UNIT-II

Linear Discriminants:

The Perceptron,

Linear Separability,

Linear Regression

Multilayer Perceptron (MLP):

Going Forwards,

Backwards, MLP in practices,

Deriving back

Propagation SUPPORT Vector Machines:

Optimal Separation,

Kernels

UNIT-II

Chapter 2

I. Linear Discriminants

PART

In the last chapter we saw a simple model of a neuron that simulated what seems to be the most important function of a neuron—deciding whether or not to fire—and ignored the nasty biological things like chemical concentrations, refractory periods, etc. Having this model is only useful if we can use it to understand what is happening when we learn, or use the model in order to solve some kind of problem. We are going to try to do both in this chapter, although the learning that we try to understand will be machine learning rather than animal learning.

One thing that is probably fairly obvious is that one neuron isn't that interesting. It doesn't do very much, except fire or not fire when we give it inputs. In fact, it doesn't even learn. If we feed in the same set of inputs over and over again, the output of the neuron never varies—it either fires or does not. So to make the neuron a little more interesting we need to work out how to make it learn, and then we need to put sets of neurons together into neural networks so that they can do something useful.

The question we need to think about first is how our neurons can learn. We are going to look at supervised learning for the next few chapters, which means that the algorithms will learn by example: the dataset that we learn from has the correct output values associated with each datapoint. At first sight this might seem pointless, since if you already know the correct answer, why bother learning at all? The key is in the concept of generalisation that we saw in Section 1.2. Assuming that there is some pattern in the data, then by showing the neural network a few examples we hope that it will find the pattern and predict the other examples correctly. This is sometimes known as pattern recognition.

patte recog

Before we worry too much about this, let's think about what learning is. In the previous chapter it was suggested that you learn if you get better at doing something. So if you can't program in the first semester and you can in the second, you have learnt to program. Something has changed (adapted), presumably in your brain, so that you can do a task that you were not able to do previously. Have a look again at the McCulloch and Pitts neuron (e.g., in Figure 1.6) and try to work out what can change in that model. The only things that make up the neuron are the inputs, the weights, and the threshold (and there is only one threshold for each neuron, but lots of inputs). The inputs can't change, since they are external, so we can only change the weights and the threshold, which is interesting since it tells us that most of

the learning is in the weights, which aren't part of the neuron at all; th
are the model of the synapse! Getting excited about neurons turns out to b
missing something important, which is that the learning happens *between* the
neurons, in the way that they are connected together. So in order to make
a neuron learn, the question that we need to ask is:

How should we change the weights and thresholds of the neurons so that the network gets the right answer more often?

Now that we know the right question to ask we'll have a look at our very first neural network, the space-age sounding Perceptron, and see how we can use it to solve the problem (it really was space-age, too: created in 1958). Once we've worked out the algorithm and how it works, we'll look at what it can and cannot do, and then see how statistics can give us insights into learning as well.

2.1 Preliminaries

Now is probably a good time to set up some terminology that we will use throughout the book. We've already seen a bit of it in the previous chapter. We will talk about **inputs and input vectors** for our learning algorithms. Likewise, we will talk about the **outputs** of the algorithm. The inputs are the data that is fed into the algorithm. In general, machine learning algorithms all work by taking a set of input values, producing an output (answer) for that input vector, and then moving on to the next input. The input vector will typically be several real numbers, which is why it is described as a vector: it is written down as a series of numbers, e.g., $(0.2, 0.45, 0.75, -0.3)$. The size of this vector, i.e., the number of elements in the vector, is called the **dimensionality** of the input. This is because if we were to plot the vector as a point, we would need one dimension of space for each of the different elements of the vector, so that the example above has 4 dimensions. We will talk about this much more in Section 4.1.1.

We will often write equations in vector and matrix notation, with lowercase boldface letters being used for vectors and uppercase boldface letters for matrices. A vector $\mathbf{x}$ has elements $(x_1, x_2, \ldots, x_m)$. We will use the following notation in the book:

Inputs An input vector is the data given as one input to the network. Written as $\mathbf{x}$, with elements x_i, where i runs from 1 to the number of input dimensions, m.

Weights w_{ij}, which is the weighted connection between nodes i and j. These weights are equivalent to the synapses in the brain. They are arranged into a matrix $\mathbf{W}$.

Outputs The output vector is $\mathbf{y}$, with elements y_j, where j runs from 1 to the number of output dimensions, n. We can write $\mathbf{y}(\mathbf{x}, \mathbf{W})$ to remind ourselves that the output depends on the inputs to the algorithm and the current set of weights of the network.

Targets The target vector $\mathbf{t}$, with elements t_j, where j runs from 1 to the number of output dimensions, n, are the extra data that we need for supervised learning, since they provide the 'correct' answers that the algorithm is learning about.

Activation Function $g(\cdot)$ is a mathematical function that describes the firing of the neuron as a response to the weighted inputs, such as the threshold function described in Section 1.5.2.

Error E, a function that computes the inaccuracies of the network as a function of the outputs $\mathbf{y}$ and targets $\mathbf{t}$.

2.2 The Perceptron

The Perceptron is nothing more than a collection of McCulloch and Pitts neurons together with a set of inputs and some weights to fasten the inputs to the neurons. The network is shown in Figure 2.1. On the left of the figure, shaded in light grey, are the input nodes. These are not neurons, they are just a nice schematic way of showing how values are fed into the network, and how many of these input values there are (which is the dimension (number of elements) in the input vector). They are almost always drawn as circles, just like neurons, which is rather confusing, so I've shaded them a different colour. The neurons are shown on the right, and you can see both the additive part (shown as a circle) and the thresholder. In practice nobody bothers to draw the thresholder separately, you just need to remember that it is part of the neuron.

Notice that the neurons in the Perceptron are completely independent of each other: it doesn't matter to any neuron what the others are doing, it works out whether or not to fire by multiplying together its own weights and the input, adding them together, and comparing the result to its own threshold, regardless of what the other neurons are doing. Even the weights that go into each neuron are separate for each one, so the only thing they share is the inputs, since every neuron sees all of the inputs to the network.

In Figure 2.1 the number of inputs is the same as the number of neurons, but this does not have to be the case — in general there will be m inputs and n neurons. The number of inputs is determined for us by the data, and so is the number of outputs, since we are doing supervised learning, so we want

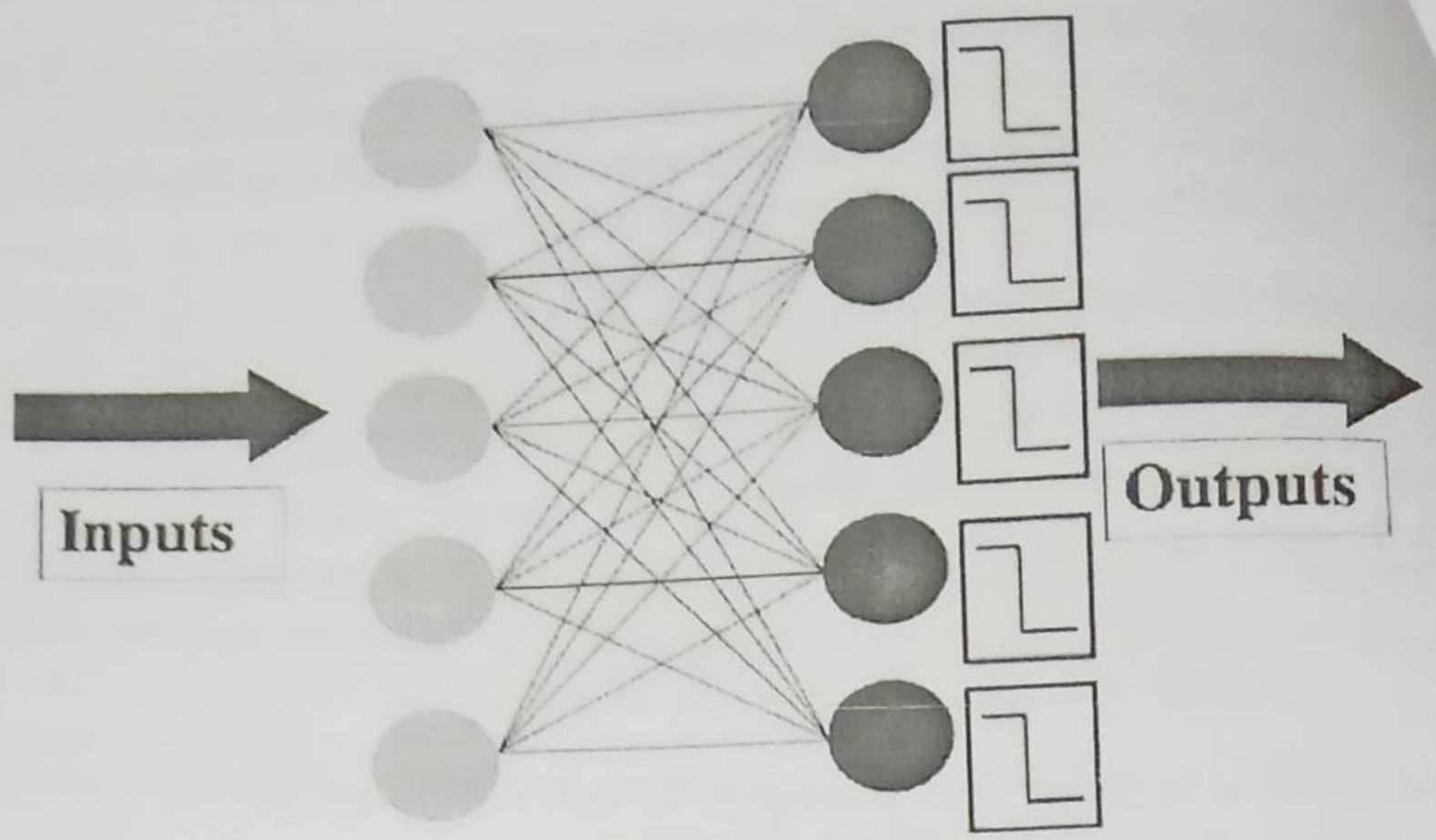


FIGURE 2.1: The Perceptron network, consisting of a set of input nodes (left) connected to McCulloch and Pitts neurons using weighted connections.

the Perceptron to learn to reproduce a particular target, that is, a pattern of firing and non-firing neurons for the given input.

When we looked at the McCulloch and Pitts neuron, the weights were labelled as w_i, with the i index running over the number of inputs. Here, we also need to work out which neuron the weight feeds into, so we label them as w_{ij}, where the j index runs over the number of neurons. So w_{32} is the weight that connects input node 3 to neuron 2. When we make an implementation of the neural network, we can use a two-dimensional array to hold these weights.

Now, working out whether or not a neuron should fire is easy: we set the values of the input nodes to match the elements of an input vector and then use Equations (1.1) and (1.2) for each neuron. We can do this for all of the neurons, and the result is a pattern of firing and non-firing neurons, which looks like a vector of 0s and 1s, so if there are 5 neurons, as in Figure 2.1, then a typical output pattern could be $(0, 1, 0, 0, 1)$, which means that the second and fifth neurons fired and the others did not. We compare that pattern to the target, which is our known correct answer for this input, to identify which neurons got the answer right, and which did not.

For a neuron that is correct, we are happy, but any neuron that fired when it shouldn't have done, or failed to fire when it should, needs to have its weights changed. The trouble is that we don't know what the weights should be—that's the point of the neural network, after all, so we want to change the weights so that the neuron gets it right next time. We are going to talk about this in a lot more detail in Chapter 3, but for now we're going to do something fairly simple to see that it is possible to find a solution.

Suppose that we present an input vector to the network and one of the

neurons gets the wrong answer (its output does not match the target). There are m weights that are connected to that neuron, one for each of the input nodes. If we label the neuron that is wrong as k, then the weights that we are interested in are w_{ik}, where i runs from 1 to m. So we know which weights to change, but we still need to work out how to change the values of those weights. The first thing we need to know is whether each weight is too big or too small. This seems obvious at first: some of the weights will be too big if the neuron fired when it shouldn't have, and too small if it didn't fire when it should. So we compute $t_k - y_k$ (the difference between the target for that neuron t_k, which is what the neuron should have done, and the output y_k, which is what the neuron did. This is a possible error function). If it is positive then the neuron should have fired and didn't, so we make the weights bigger, and vice versa if it is negative. Hold on, though. That element of the input could be negative, which would switch the values over: so if we wanted the neuron to fire we'd need to make the value of the weight negative as well. To get around this we'll multiply those two things together to see how we should change the weight: $\Delta w_{ik} = (t_k - y_k) \times x_i$, and the new value of the weight is the old value plus this value.

Note that we haven't said anything about changing the threshold value of the neuron. To see how important this is, suppose that a particular input is 0. In that case, even if a neuron is wrong, changing the relevant weight doesn't do anything (since anything times 0 is 0): we need to change the threshold. We will deal with this in an elegant way in Section 2.2.2. However, before we get to that, the learning rule needs to be finished—we need to decide how much to change the weight by. This is done by multiplying the value above by a parameter called the learning rate, usually labelled as η. The value of the learning rate decides how fast the network learns. It's quite important, so it gets a little subsection of its own (next), but first let's write down the final rule for updating a weight w_{ij}:

$$w_{ij} \leftarrow w_{ij} + \eta(t_j - y_j) \cdot x_i. \tag{2.1}$$

The other thing that we need to realise now is that the network needs to be shown every training example several times. The first time the network might get some of the answers correct and some wrong, the next time it will hopefully improve, and eventually its performance will stop improving. Working out how long to train the network for is not easy (we will see more methods in Section 3.3.6), but for now we will predefine the maximum number of iterations, T.

2.2.1 The Learning Rate η

Equation (2.1) above tells us how to change the weights, with the parameter η controlling how much to change the weights by. We could miss it out, which would be the same as setting it to 1. If we do that, then the weights change

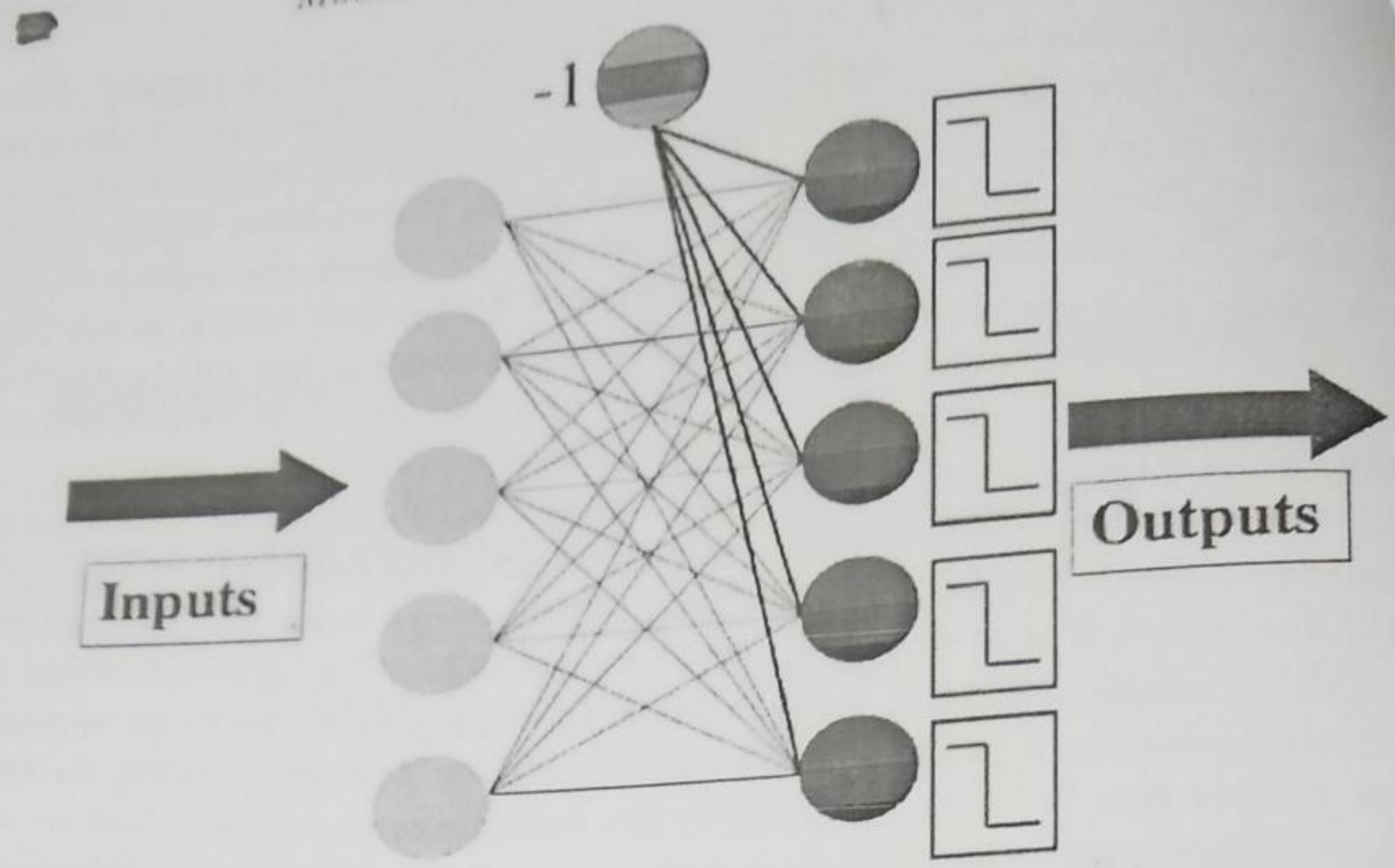


FIGURE 2.2: The Perceptron network again, showing the bias input.

a lot whenever there is a wrong answer, which tends to make the network unstable, so that it never settles down. The cost of having a small learning rate is that the weights need to see the inputs more often before they change significantly, so that the network takes longer to learn. However, it will be more stable and resistant to noise (errors) and inaccuracies in the data. We therefore use a moderate learning rate, typically $0.1 < \eta < 0.4$, depending upon how much error we expect in the inputs.

2.2.2 The Bias Input (-1)

When we discussed the McCulloch and Pitts neuron, we gave each neuron a firing threshold θ that determined what value it needed before it should fire. This threshold should be adjustable, so that we can change the value that the neuron fires at. Suppose that all of the inputs to a neuron are zero. Now it doesn't matter what the weights are (since zero times anything equals zero), the only way that we can control whether the neuron fires or not is through the threshold. If it wasn't adjustable and we wanted one neuron to fire when all the inputs to the network were zero, and another not to fire, then we would have a problem. No matter what values of the weights were set, the two neurons would do the same thing since they had the same threshold and the inputs were all zero.

The trouble is that changing the threshold requires an extra parameter that we need to write code for, and it isn't clear how we can do that in terms of the weight update that we worked out earlier. Fortunately, there is a neat

way around this problem. Suppose that we fix the value of the threshold for the neuron at zero. Now, we add an extra input weight to the neuron, with the value of the input to that weight always being fixed (usually the value of -1 is chosen). We include that weight in our update algorithm (like all the other weights), so we don't need to think of anything new. And the value of the weight will change to make the neuron fire or not fire, whichever is correct—when an input of all zeros is given, since the input on that weight is always -1, even when all the other inputs are zero. This input is often called a **bias** node, and its weights are usually given a 0 subscript, so that the weight connecting it to the jth neuron is w_{0j}.

2.2.3 The Perceptron Learning Algorithm

We are now ready to write our first learning algorithm. It might be useful to keep Figure 2.2 in mind as you read the algorithm, and we'll work through an example of using it afterwards. The algorithm is separated into two parts: a **training** phase, and a **recall** phase. The recall phase is used after training, and it is the one that should be fast to use, since it will be used far more often than the training phase. You can see that the training phase uses the recall equation, since it has to work out the activations of the neurons before the error can be calculated and the weights trained.

The Perceptron Algorithm

- **Initialisation**
  - set all of the weights w_{ij} to small (positive and negative) random numbers
- **Training**
  - for T iterations:
    - for each input vector:
      - compute the activation of each neuron j using activation function g:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 \text{ if } w_{ij}x_i > 0 \\ 0 \text{ if } w_{ij}x_i \leq 0 \end{cases} \tag{2.2}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} + \eta(t_j - y_j) \cdot x_i \tag{2.3}$$

In_1	In_2	t
0	0	0
0	1	1
1	0	1
1	1	1

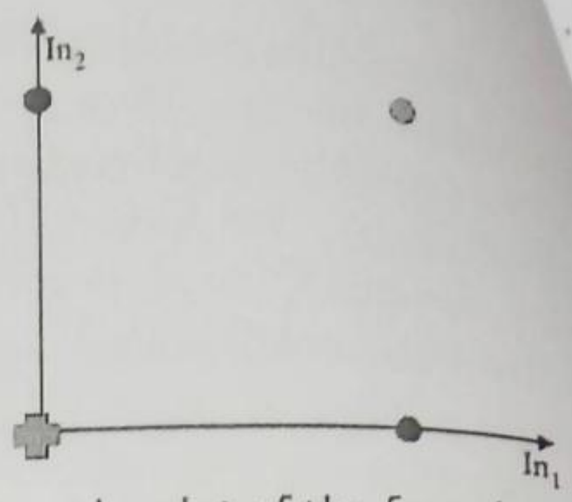


FIGURE 2.3: Data for the OR logic function and a plot of the four datapoints.

- **Recall**
  - compute the activation of each neuron j using:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 \text{ if } w_{ij}x_i > 0 \\ 0 \text{ if } w_{ij}x_i \leq 0 \end{cases} \quad (2.4)$$

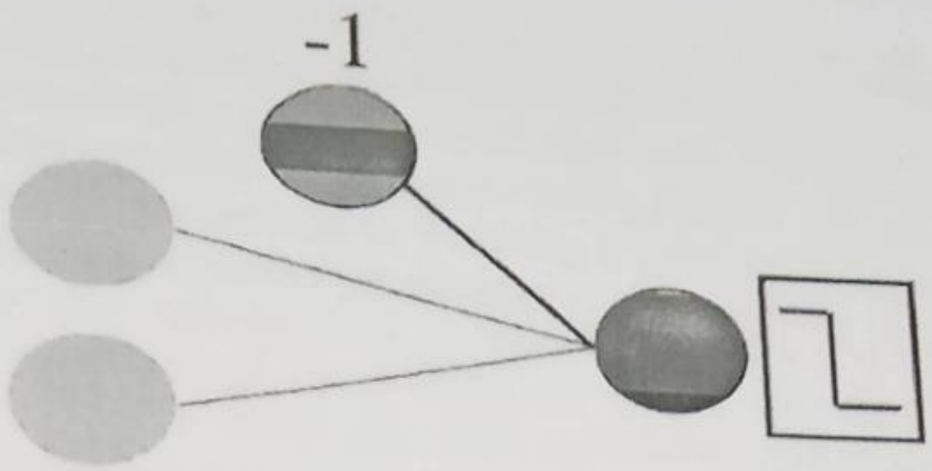
Computing the computational complexity of this algorithm is very easy. The recall phase loops over the neurons, and within that loops over the inputs, so its complexity is $\mathcal{O}(mn)$. The training part does this same thing, but does it for T iterations, so costs $\mathcal{O}(Tmn)$.

It might be the first time that you have seen an algorithm written out like this, and it could be hard to see how it can be turned into code. Equally, it might be difficult to believe that something as simple as this algorithm can learn something. The only way to fix these things is to work through the algorithm by hand on an example or two, and to try to write the code and then see if it does what is expected. We will do both of those things next, first working through a simple example by hand.

2.2.4 An Example of Perceptron Learning

The example we are going to use is something very simple that you already know about, the logical OR. This obviously isn't something that you actually need a neural network to learn about, but it does make a nice simple example. So what will our neural network look like? There are two input nodes (plus the bias input) and there will be one output. The inputs and the target are given in the table on the left of Figure 2.3; the right of the figure shows a plot of the function with the circles as the true outputs, and a cross as the false one. The corresponding neural network is shown in Figure 2.4.

As you can see from Figure 2.4, there are three weights. The algorithm tells us to initialise the weights to small random numbers, so we'll pick $w_0 = -0.05, w_1 = -0.02, w_2 = 0.02$. Now we feed in the first input, where both inputs are 0: $(0, 0)$. Remember that the input to the bias weight is always -1.



I_1	I_2	t
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE 2.4: The Perceptron network for the example in Section 2.2.4.

so the value that reaches the neuron is $-0.05\times-1+-0.02\times0+-0.02\times0 = 0.05$. This value is above 0, so the neuron fires and the output is 1, which is incorrect according to the target. The update rule tells us that we need to apply Equation (2.1) to each of the weights separately (we'll pick a value of $\eta = 0.25$ for the example):

$$w_0 : -0.05 + 0.25 \times (0 - 1) \times -1 = 0.2, \quad (2.5)$$

$$w_1 : -0.02 + 0.25 \times (0 - 1) \times 0 = -0.02, \quad (2.6)$$

$$w_2 : 0.02 + 0.25 \times (0 - 1) \times 0 = 0.02. \quad (2.7)$$

Now we feed in the next input $(0, 1)$ and compute the output (check that you agree that the neuron does not fire, but that it should) and then apply the learning rule again:

$$w_0 : 0.2 + 0.25 \times (1 - 0) \times -1 = -0.05, \quad (2.8)$$

$$w_1 : -0.02 + 0.25 \times (1 - 0) \times 0 = -0.02, \quad (2.9)$$

$$w_2 : 0.02 + 0.25 \times (1 - 0) \times 1 = 0.27. \quad (2.10)$$

For the $(1, 0)$ input the answer is already correct (you should check that you agree with this), so we don't have to update the weights at all, and the same is true for the $(1, 1)$ input. So now we've been through all of the inputs once. Unfortunately, that doesn't mean we've finished—not all the answers are correct yet. We now need to start going through the inputs again, until the weights settle down and stop changing, which is what tells us that the algorithm has finished. For real world applications the weights may never stop changing, which is why you run the algorithm for some pre-set number of iterations, T.

So now we carry on running the algorithm, which you should check for yourself either by hand or using computer code (which we'll discuss next), eventually getting to weight values that settle and stop changing. At this

point the weights stop changing, and the Perceptron has correctly learnt all of the examples. Note that there are lots of different values that we can assign to the weights that will give the correct outputs: the ones that the algorithm finds depend on the learning rate, the inputs, and the initial starting values. We are interested in finding a set that works; we don't necessarily care what the actual values are, providing that the network generalises to other inputs.

2.2.5 Implementation

Turning the algorithm into code is fairly simple: we need to design some data structures to hold the variables, then write and test the program. Data structures are usually very basic for machine learning algorithms: here we need an array to hold the inputs, another to hold the weights, and then two more for the outputs and the targets. When we talked about the presentation of data to the neural network we used the term input vectors. The vector is a list of values that are presented to the Perceptron, with one value for each of the nodes in the network. When we turn this into computer code it makes sense to put these values into an array. However, the neural network isn't very exciting if we only show it one datapoint: we will need to show it lots of them. Therefore it is normal to arrange the data into a two-dimensional array, with each row of the array being a datapoint. In a language like C or Java, you then write a loop that runs over each row of the array to present the input, and a loop within it that runs over the number of input nodes (which does the computation on the current input vector).

Written this way in Python syntax (Chapter 16 provides a brief introduction to Python), the recall code that is used after training for a set of `nData` datapoints arranged in the array `inputs` looks like:

```
for data in range(nData): # loop over the input vectors
    for n in range(N): # loop over the neurons
        # Compute sum of weights times inputs for each neuron
        # Set the activation to 0 to start
        activation[data][n] = 0
        # Loop over the input nodes (+1 for the bias node)
        for m in range(M+1):
            activation[data][n] += weight[m][n] * inputs[data][m]

        # Now decide whether the neuron fires or not
        if activation[data][n] > 0:
            activation[data][n] = 1
        else
            activation[data][n] = 0
```

However, Python's numerical library NumPy provides an alternative method, because it can easily multiply arrays and matrices together (MATLAB and R have the same facility). This means that we can write the code with fewer loops, making it rather easier to read, and also means that we write less code. It can be a little confusing at first, though. To understand it, we need a little bit more mathematics, which is the concept of a matrix. In computer terms, matrices are just two-dimensional arrays. We can write the set of weights for the network in a matrix by making an `array` that has $m + 1$ rows (the number of input nodes + 1 for the bias) and n columns (the number of neurons). Now, the element of the matrix at location (i, j) contains the weight connecting input i to neuron j, which is what we had in the code above.

The benefit that we get from thinking about it in this way is that multiplying matrices and vectors together is well defined. You've probably seen this in high school or somewhere but, just in case, to be able to multiply matrices together we need the inner dimensions to be the same. This just means that if we have matrices $\mathbf{A}$ and $\mathbf{B}$ where $\mathbf{A}$ is size $m \times n$, then the size of $\mathbf{B}$ needs to be $n \times p$, where p can be any number. The n is called the inner dimension since when we write out the size of the matrices in the multiplication we get $(m \times n) \times (n \times p)$.

Now we can compute $\mathbf{AB}$ (but not necessarily $\mathbf{BA}$, since for that we'd need $m = p$, since the computation above would then be $(n \times p) \times (m \times n)$). The computation of the multiplication proceeds by picking up the first column of $\mathbf{B}$, rotating it by 90° anti-clockwise so that it is a row not a column, multiplying each element of it by the matching element in the first row of $\mathbf{A}$ and then adding them together. This is the first element of the answer matrix. The second element in the first row is made by picking up the second column of $\mathbf{B}$, rotating it to match the direction, and multiplying it by the first row of $\mathbf{A}$, and so on. As an example:

$$\begin{pmatrix} 3 & 4 & 5 \\ 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{pmatrix} \tag{2.11}$$

$$= \begin{pmatrix} 3\times1+4\times2+5\times3 & 3\times3+4\times4+5\times5 \\ 2\times1+3\times2+4\times3 & 2\times3+3\times4+4\times5 \end{pmatrix} \tag{2.12}$$

$$= \begin{pmatrix} 26 & 50 \\ 20 & 38 \end{pmatrix} \tag{2.13}$$

NumPy can do this multiplication for us, using the `dot()` function (which is a rather strange name mathematically, but never mind). So to reproduce the calculation above, we use:

```
>>> from numpy import *
>>> a = array([[3,4,5],[2,3,4]])
>>> b = array([[1,3],[2,4],[3,5]])
```

```
>>> dot(a,b)
array([[26, 50],
       [20, 38]])
```

The `array()` function makes the NumPy array, which is actually a matrix here, made up of an array of arrays: each row is a separate array, as you can see from the square brackets within square brackets. Note that we can enter the 2D array in one line of code by using commas between the different rows, but when it prints them out, NumPy puts each row of the matrix on a different line, which makes things easier to see.

This probably seems like a very long way from the Perceptron, but we are getting there, I promise! We can put the input vectors into a two-dimensional array of size $N \times m$, where N is the number of input vectors we have and m is the number of inputs. The weights array is of size $m \times n$, and so we can multiply them together. If we do, then the output will be an $N \times n$ matrix that holds the values of the sum that each neuron computes for each of the N input vectors. Now we just need to compute the activations based on these sums. NumPy has another useful function for us here, which is `where(condition,x,y)`. (`condition` is a logical condition and `x` and `y` are values) that returns a matrix that has value `x` where `condition` is true and value `y` everywhere else. So using the matrix `a` that was used above,

```
>>> where(a>3,1,0)
array([[0, 1, 1],
       [0, 0, 1]])
```

The upshot of this is that the entire section of code for the recall function of the Perceptron can be rewritten in two lines as:

```
activations = dot(inputs,weights)
activations = where(activations>0,1,0)
```

The training section isn't that much harder really. You should notice that the first part of the training algorithm is the same as the recall computation, so we can put them into a function (I've called it `pcnfwd` in the code because it consists of running forwards through the network to get the outputs). Then we just need to compute the weight updates. The weights are in an $m \times n$ matrix, the activations are in an $N \times n$ matrix (as are the targets) and the inputs are in an $N \times m$ matrix. So to do the multiplication `dot(inputs,targets - activations)` we need to turn the `inputs` matrix around so that it is $m \times N$. This is done using the `transpose()` function, which swaps the rows and columns over (so using matrix `a` above again) we get:

```
>>> transpose(a)
array([[3, 2],
       [4, 3],
       [5, 4]])
```

Once we have that, the weight update for the entire network can be done in one line (where `eta` is the learning rate, η):

```
weights += eta*dot(transpose(inputs),targets-activations)
```

Assuming that you make sure in advance that all your input matrices are the correct size (the `shape()` function, which tells you the number of elements in each dimension of the array, is helpful here), the only things that are needed are to add those extra -1's onto the input vectors for the bias node, and to decide what values we should put into the weights to start with. The first of these can be done using the `concatenate()` function, making an one-dimensional array that contains -1 as all of its elements, and adding it on to the inputs array. Note that `nData` in the code is equivalent to N in the text.

```
inputs = concatenate((-ones((nData,1)),inputs),axis=1)
```

The last thing we need to do is to give initial values to the weights. It is possible to set them all to be zero, and the algorithm will get to the right answer. However, instead we will assign small random numbers to the weights, for reasons that will be discussed in Section 3.2.2. Again, NumPy has a nice way to do this, using the built-in random number generator (with `nin` corresponding to m and `nout` to n):

```
weights = random.rand(nIn+1,nOut)*0.1-0.05
```

At this point we have seen all the snippets of code that are required, and putting them together should not be a problem. The entire program is available from the book website as `pcn.py`. What is interesting is to see the code working, and that is what we will do next, starting with the OR example that was used in the hand-worked demonstration.

Making the OR data is easy, and then running the code requires importing it using its filename (`pcn`) and then calling the `pcntrain` function. The printout below shows the instructions to set up the arrays and call the function, and the output of the weights for 5 iterations of a particular run of the program, starting from random initial points (note that the weights stop changing after the 1st iteration in this case, and that different runs will produce different values).

```
>>> from numpy import *
>>> inputs = array([[0,0],[0,1],[1,0],[1,1]])
>>> targets = array([[0],[1],[1],[1]])
>>> import pcn_logic_eg

>>> p = pcn_logic_eg.pcn(inputs,targets)
>>> p.pcntrain(inputs,targets,0.25,6)

Iteration:  0
 [[-0.22546505]
  [ 0.03035344]
  [ 0.2684798 ]]
 Iteration:  1
 [[ 0.02453495]
  [ 0.03035344]
  [ 0.2684798 ]]
 Iteration:  2
 [[ 0.02453495]
  [ 0.03035344]
  [ 0.2684798 ]]
 Iteration:  3
 [[ 0.02453495]
  [ 0.03035344]
  [ 0.2684798 ]]
 Iteration:  4
 [[ 0.02453495]
  [ 0.03035344]
  [ 0.2684798 ]]
 Iteration:  5
 [[ 0.02453495]
  [ 0.03035344]
  [ 0.2684798 ]]
 Final outputs are:
[[0]
 [1]
 [1]
 [1]]
```

We have trained the Perceptron on the four datapoints $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$. However, we could put in an input like $(0.8,0.8)$ and expect to get an output from the neural network. Obviously, it wouldn't make any sense from the logic function point-of-view, but most of the things that we do with neural networks will be more interesting than that, anyway. Figure 2.5 shows

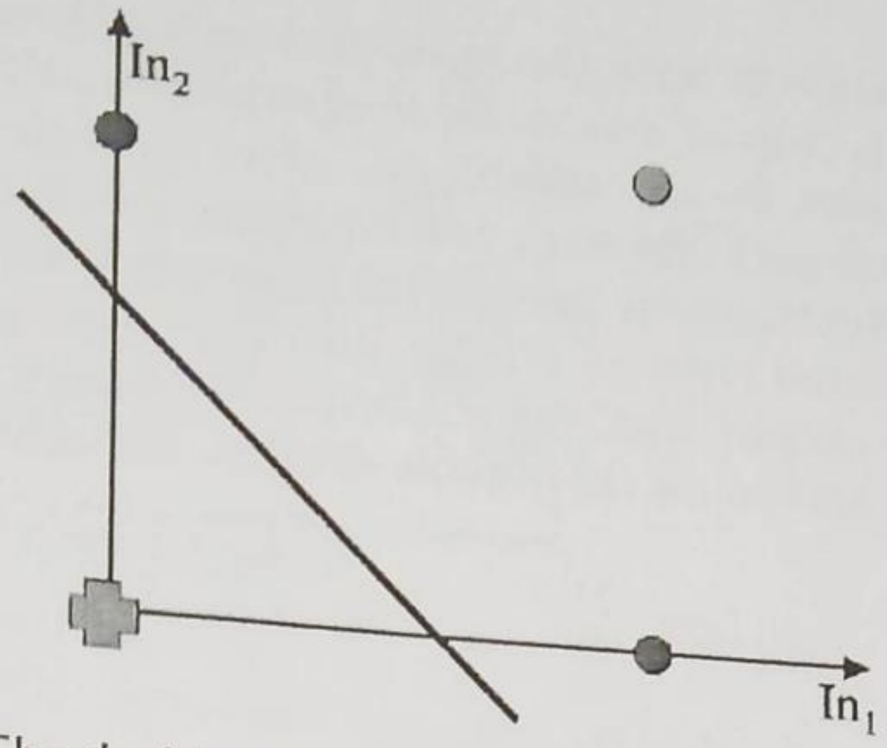


FIGURE 2.5: The decision boundary computed by a Perceptron for the OR function.

the decision boundary, which shows when the decision about which class to categorise the input as changes from crosses to circles. We will see why this is a straight line in Section 2.3.

2.2.6 Testing the Network

Before returning the weights, the Perceptron algorithm above prints out the outputs for the trained inputs. You can also use the network to predict the outputs for other values by using the `pcnfwd` function. However, you need to manually add the -1s on in this case, using:

```
>>> inputs_bias = concatenate((-ones((shape(inputs)[0],1)),
inputs),axis=1)
>>> pcn.pcnfwd(inputs_bias,weights)
```

This brings us to an interesting question, which is how do you decide whether or not the network has learnt well? The first thing that we can do is look at the error on the training set. We asked the Perceptron to learn about the OR data, and it got the predictions 100% correct. However, we want a neural network to generalise to examples that it has not seen in the training set, and we can't test this by using the training set. So we need some different data, a **test set** to test it on as well. This isn't very easy for this example, but for real datasets, you separate the data into a training set and a separate test set. This will be covered in more detail in Section 3.3.5.

Regardless of what data we use to test the network, we still need to work out whether or not the result is good. We will look here at a method that is suitable for classification problems that is known as the **confusion matrix**. It is a nice simple idea, which is to make a square matrix that contains all

the possible classes in both the horizontal and vertical directions. We list the classes along the top of a table as the outputs, and then down the left-hand side as the targets. So for example, the element of the matrix at (i, j) tells us how many input patterns were put into class i in the targets, but class j by the network. Anything on the leading diagonal is a correct answer. Suppose that we have three classes: C_1, C_2, and C_3. Now we count the number of times that the output was class C_1 when the target was C_1, then when the target was C_2, and so on until we've filled in the table:

	Outputs		
	C_1	C_2	C_3
C_1	5	1	0
C_2	1	4	1
C_3	2	0	4

This table tells us that, for the three classes, most examples were classified correctly, but two examples of class C_3 were misclassified as C_1, and so on. Writing the code to compute this is not too difficult, and for a small number of classes, it is a nice way to look at the outputs. If you just want one number, then it is possible to divide the sum of the elements on the leading diagonal by the sum of all of the elements in the matrix, which gives the fraction of correct responses.

We've now reached the stage that neural networks were up to in 1969. Then, two researchers, Minsky and Papert, published a book called "Perceptrons." The purpose of the book was to stimulate neural network research by discussing the learning capabilities of the Perceptron, and showing what the network could and could not learn. Unfortunately, the book had another effect: it effectively killed neural network research for about 20 years. To see why, we need to think about how the Perceptron learns in a different way.

2.3 Linear Separability

What does the Perceptron actually compute? For our one output neuron example of the OR data it tries to separate out the cases where the neuron should fire from those where it shouldn't. Looking at the graph on the right side of Figure 2.3, you should be able to draw a straight line that separates out the crosses from the circles without difficulty (it is done in Figure 2.5). In fact, that is exactly what the Perceptron does: it tries to find a straight line (in 2D, a plane in 3D, and a hyperplane in higher dimensions) where the neuron fires on one side of the line, and doesn't on the other. This line is called the decision boundary or discriminant function, and an example of one is given in Figure 2.6.

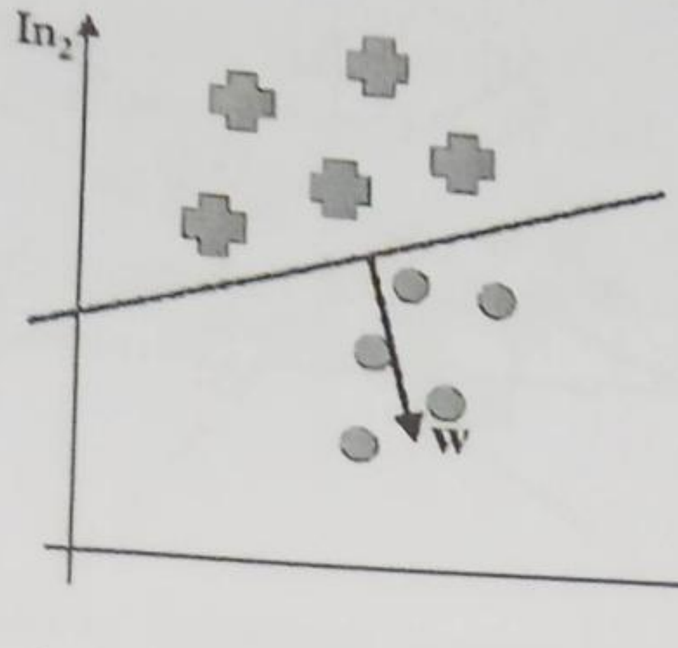


FIGURE 2.6: A decision boundary separating two classes of data.

To see this, think about the matrix notation we used in the implementation, but consider just one input vector $\mathbf{x}$. The neuron fires if $\mathbf{x} \cdot \mathbf{w}^T \geq 0$ (where $\mathbf{w}$ is the row of $\mathbf{W}$ that connects the inputs to one particular neuron: they are the same for the OR example, since there is only one neuron, and $\mathbf{w}^T$ denotes the transpose of $\mathbf{w}$ and is used to make both of the vectors into column vectors). The $\mathbf{a} \cdot \mathbf{b}$ notation describes the inner or scalar product between two vectors. It is computed by multiplying each element of the first vector by the matching element of the second and adding them all together. As you might remember from high school, $\mathbf{a} \cdot \mathbf{b} = \|a\|\|b\| \cos\theta$, where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$ and $\|a\|$ is the length of the vector $\mathbf{a}$. So the inner product computes a function of the angle between the two vectors, scaled by their lengths. It can be computed in NumPy using the `inner()` function.

Getting back to the Perceptron, the boundary case is where we find an input vector $\mathbf{x}_1$ that has $\mathbf{x}_1 \cdot \mathbf{w}^T = 0$. Now suppose that we find another input vector $\mathbf{x}_2$ that satisfies $\mathbf{x}_2 \cdot \mathbf{w}^T = 0$. Putting these two equations together we get:

$$\mathbf{x}_1 \cdot \mathbf{w}^T = \mathbf{x}_2 \cdot \mathbf{w}^T \tag{2.14}$$

$$\Rightarrow (\mathbf{x}_1 - \mathbf{x}_2) \cdot \mathbf{w}^T = 0. \tag{2.15}$$

What does this last equation mean? In order for the inner product to be 0, either $\|a\|$ or $\|b\|$ or $\cos\theta$ needs to be zero. There is no reason to believe that $\|a\|$ or $\|b\|$ should be 0, so $\cos\theta = 0$. This means that $\theta = \pi/2$ (or $-\pi/2$), which means that the two vectors are at right angles to each other. Now $\mathbf{x}_1 - \mathbf{x}_2$ is a straight line between two points that lie on the decision boundary, and the weight vector $\mathbf{w}^T$ must be perpendicular to that, as in Figure 2.6.

So given some data, and the associated target outputs, the Perceptron simply tries to find a straight line that divides the examples where each neuron fires from those where it does not. This is great if that straight line exists, but is a bit of a problem otherwise. The cases where there is a straight

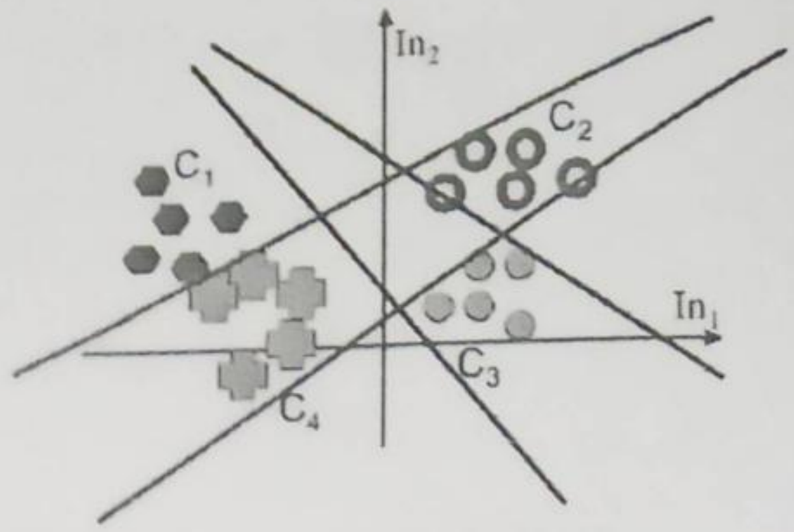


FIGURE 2.7: Different decision boundaries computed by a Perceptron with four neurons.

In_1	In_2	t
0	0	0
0	1	1
1	0	1
1	1	0

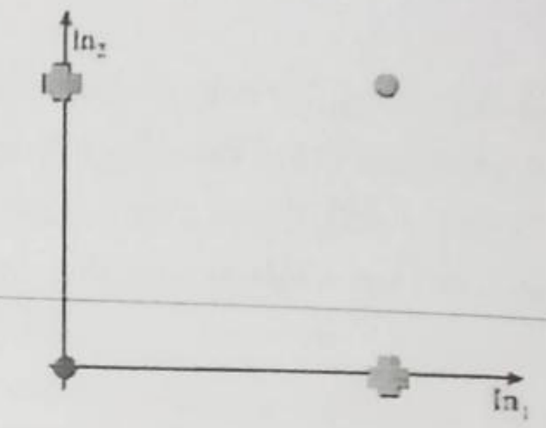


FIGURE 2.8: Data for the XOR logic function and a plot of the four data points.

line are called linearly separable cases. What happens if the classes that w want to learn about are not linearly separable? It turns out that makin such a function is very easy: there is even one that matches a logic functio Before we have a look at it, it is worth thinking about what happens when w have more than one output neuron. The weights for each neuron separatel describe a straight line, so by putting together several neurons we get severa straight lines that each try to separate different parts of the space. Figure 2. shows an example of decision boundaries computed by a Perceptron with fou neurons: by putting them together we can get good separation of the classe

2.3.1 The Exclusive Or (XOR) Function

The XOR has the same four input points as the OR function, but lookin at Figure 2.8, you should be able to convince yourself that you can't draw straight line on the graph that separates true from false (crosses from circles In our new language, the XOR function is not linearly separable. If the ana ysis above is correct, then the Perceptron will fail to get the correct answe and using the Perceptron code above we find:

```
>>> targets = array([[0],[1],[1],[0]])
>>> pcn.pcntrain(inputs,targets,0.25,15)
```

which gives the following output (the early iterations have been missed out):

```
Iteration:  11
[[ 0.45946905]
 [-0.27886266]
 [-0.25662428]]
Iteration:  12
[[-0.04053095]
 [-0.02886266]
 [-0.00662428]]
Iteration:  13
[[ 0.45946905]
 [-0.27886266]
 [-0.25662428]]
Iteration:  14
[[-0.04053095]
 [-0.02886266]
 [-0.00662428]]
Final outputs are:
[[0]
 [0]
 [0]]
```

You can see that the algorithm does not converge, but keeps on cycling through two different wrong solutions. Running it for longer does not change this behaviour. So even for a simple logical function, the Perceptron can fail to learn the correct answer. This is what was demonstrated by Minsky and Papert in "Perceptrons," and the discovery that the Perceptron was not capable of solving even these problems, let alone more interesting ones, is what halted neural network development for so long. There is an obvious solution to the problem, which is to make the network more complicated—add in more neurons, with more complicated connections between them, and see if that helps. The trouble is that this makes the problem of training the network much more difficult. In fact, working out how to do that is the topic of the next chapter.

In_1	In_2	In_3	Output
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	1

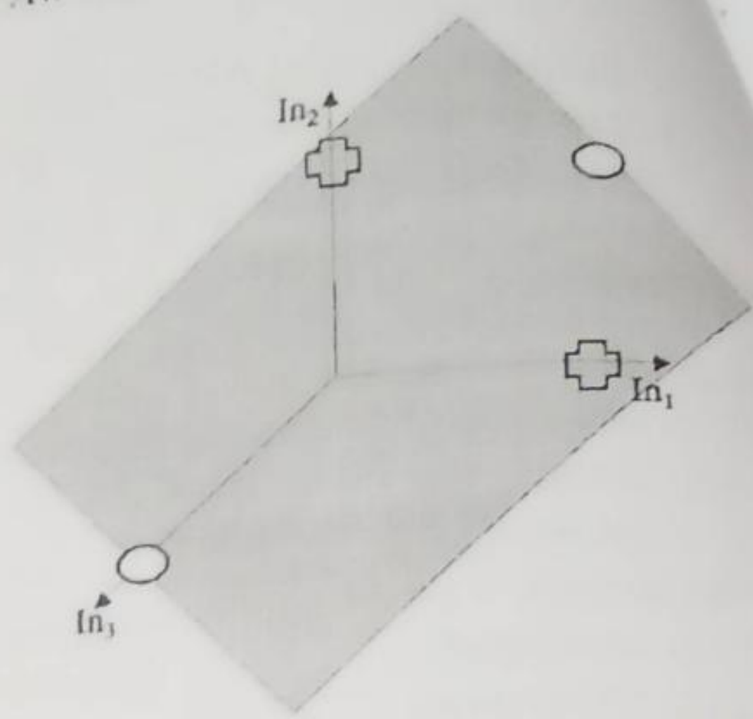


FIGURE 2.9: A decision boundary (the shaded plane) solving the XOR problem in 3D with the crosses below the surface and the circles above it.

2.3.2 A Useful Insight

From the discussion in Section 2.3.1 you might think that the XOR function is impossible to solve using a linear function. In fact, this is not true. If we rewrite the problem in three dimensions instead of two, then it is perfectly possible to find a plane (the 2D analogue of a straight line) that can separate the two classes. There is a picture of this in Figure 2.9. Writing the problem in 3D means including a third input dimension that does not change the data when it is looked at in the (x, y) plane, but moves the point at $(0, 0)$ along a third dimension. So the truth table for the function is the one shown on the left side of Figure 2.9 (where 'In_3' has been added, and only affects the point at $(0, 0)$).

To demonstrate this, the following listing uses the same Perceptron code:

```
>>> inputs = array([[0,0,1],[0,1,0],[1,0,0],[1,1,0]])
>>> pcn.pcntrain(inputs,targets,0.25,15)
Iteration:  14
[[-0.27757663]
 [-0.21083089]
 [-0.23124407]
 [-0.53808657]]
Final outputs are:
[[0]
 [1]
 [1]
 [0]]
```

In fact, it is always possible to separate out two classes with a linear func-

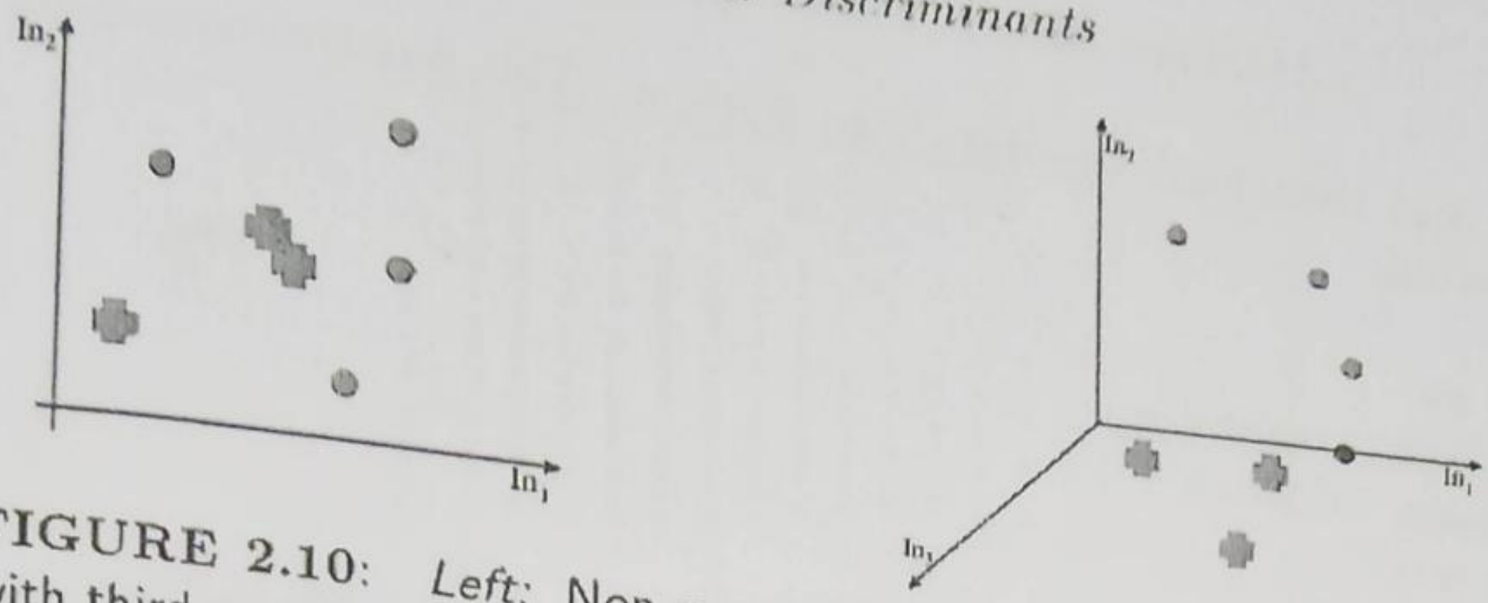


FIGURE 2.10: *Left:* Non-separable 2D dataset. *Right:* The same dataset with third coordinate $x_1 \times x_2$, which makes it separable.

tion, provided that you project the data into the correct set of dimensions. There are a whole class of methods for doing this reasonably efficiently, called kernel classifiers, which are the basis of support vector machines, which are the subject of Chapter 5. For now, it is sufficient to point out that if you want to make your linear Perceptron do non-linear things, then there is nothing to stop you making non-linear variables. For example, Figure 2.10 shows two versions of the same dataset. On the left side, the coordinates are x_1 and x_2, while on the right side the coordinates are x_1, x_2 and $x_1 \times x_2$. It is now easy to fit a plane (the 2D equivalent of a straight line) that separates the data.

Statistics has been dealing with problems of classification and regression for a long time, before we had computers in order to do difficult arithmetic for us, and so straight line methods have been around in statistics for many years. They provide a different (and useful) way to understand what is happening in learning, and by using both statistical and computer science methods we can get a good understanding of the whole area. We will see the statistical method of linear regression in Section 2.4, but first we will work through another example of using the Perceptron. This is meant to be a tutorial example, so I will give some of the relevant code and results, but leave places for you to fill in the gaps.

2.3.3 Another Example: The Pima Indian Dataset

The UCI Machine Learning Repository (`http://archive.ics.uci.edu/ml/`) holds lots of datasets that are used to demonstrate and test machine learning algorithms. For the purposes of testing out the Perceptron and Linear Regressor, we are going to use one that is very well known. It provides eight measurements of a group of American Pima Indians living in Arizona in the USA, and the classification is whether or not each person had diabetes. The dataset is available from the UCI repository (called `Pima`) and there is a file inside the folder giving details of what the different variables mean.

Once you have downloaded it, import the relevant modules (NumPy to use the array methods, PyLab to plot the data, and the Perceptron from the book

website) and then load the data into Python. This requires something like the following:

```
import os
from pylab import *
from numpy import *
import pcn

os.chdir('~/Book/Datasets/pima')
pima = loadtxt('pima-indians-diabetes.data',delimiter=',')
shape(pima)
(768, 9)
```

where the path in the `os.chdir` line will obviously need to be changed to wherever you have saved the dataset. In the `loadtxt()` command the `delimiter` specifies which character is used to separate out the datapoints. Note that PyLab is imported before NumPy. This should not matter, in general, but there are some commands in PyLab that overwrite some in NumPy, and we want to use the NumPy ones, so you need to import them in that order. The `shape()` method tells that there are 768 datapoints, arranged as rows of the file, with each row containing nine numbers. These are the eight dimensions of data, with the class being the ninth element of each line (indexed as 8 since Python is zero-indexed). This arrangement, with each line of a file (or row of an array) being a datapoint is the one that will be used throughout the book.

You should have a look at the dataset. Obviously, you can't plot the whole thing at once, since that would require being able to visualise eight dimensions. But you can plot any two-dimensional subset of the data. Have a look at a few of them. In order to see the two different classes in the data in your plot, you will have to work out how to use the `where` command. Once you have worked that out, you will be able to plot them with different shapes and colours. The `plot` command is in Matplotlib, so you'll need to import that (as `pylab`) beforehand. Assuming that you have worked out some way to store the indices of one class in `indices0` and the other in `indices1` you can use:

```
ion()
plot(pima[indices0,0],pima[indices0,1],'go')
plot(pima[indices1,0],pima[indices1,1],'rx')
show()
```

to plot the first two dimensions as green circles and red crosses, which (up to colour, of course) should look like Figure 2.11. The `ion()` command ensures that the data is actually plotted, while the `show()` command is only required if you are using Eclipse, and ensures that the graph does not vanish when

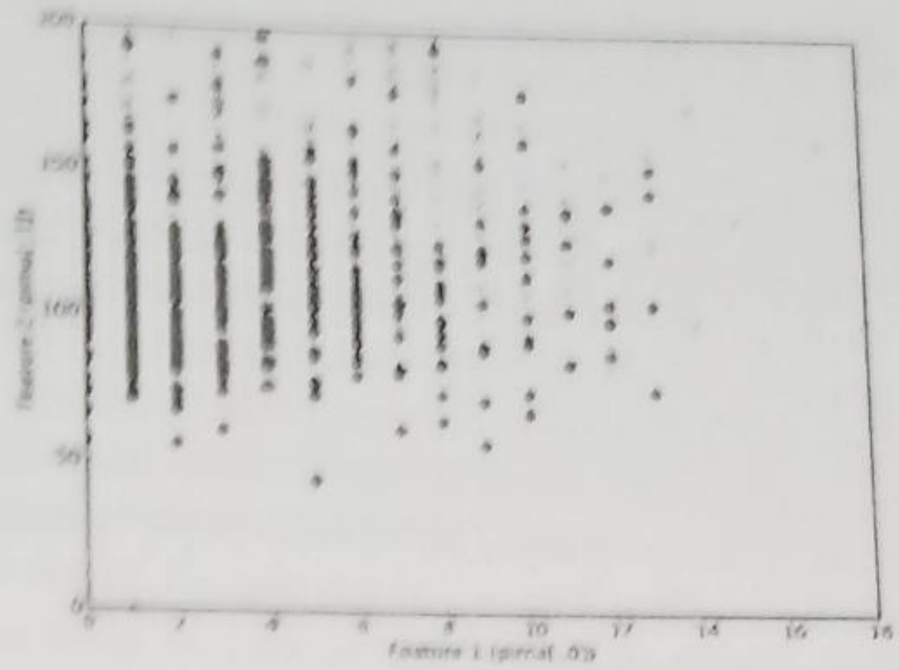


FIGURE 2.11: Plot of the first two dimensions of the Pima Indians dataset showing the two classes as 'x' and 'o'.

the program terminates. Clearly, there is no way that you can find a linear separation between these two classes with these features. However, you should have a look at some of the other combinations of features and see if you can find any that are better.

The next thing to do is to try using the Perceptron on the full dataset. You will need to try out different values for the learning rate and the number of iterations for the Perceptron, but you should find that you can get around 50-70% correct (use the confusion matrix method `confmat()` to get the results). This isn't too bad, but it isn't that good, either. The results are quite unstable, too: sometimes the results have only 30% accuracy—worse than chance—which is rather depressing.

```
p = pcn.pcn(pima[:,:8],pima[:,8:9])
p.pcntrain(pima[:,:8],pima[:,8:9],0.25,100)
p.confmat(pima[:,:8],pima[:,8:9])
```

This is, of course, unfair testing, since we are testing the network on the same data we were training it on. We will talk about this more in Section 3.3.5, but we will do something quick now, which is to use even-numbered datapoints for training, and odd-numbered datapoints for testing. This is very easy using the : operator, where we specify the start point, the end point, and the step size. NumPy will fill in any that we leave blank with the beginning or end of the array as appropriate.

```
trainin = pima[::2,:8]
testin = pima[1::2,:8]
traintgt = pima[::2,8:9]
```

II - 23

```
testtgt = pima[1::2,8:9]
```

For now, rather than worrying about training and testing data, we are more interested in working out how to improve the results. And we can do better. The first thing to do is to have a proper look at some of the data. For example, column 0 is the number of times that the person has been pregnant (did I mention that all the subjects were female?) and column 7 is the age of the person. Taking the pregnancy variable first, there are relatively few subjects that were pregnant 8 or more times, so rather than having the number there, maybe they should be replaced by an 8 for any of these values. Equally, the age would be better quantised into a set of ranges such as 21–30, 31–40, etc. (the minimum age is 21 in the dataset). This can be done using the `where` function again, as in this code snippet. If you make these changes and similar ones for the other values, then you should be able to get massively better results.

```
pima[where(pima[:,0]>8),0] = 8

pima[where(pima[:,7]<=30),7] = 1
pima[where((pima[:,7]>30) & (pima[:,7]<=40)),7] = 2
#You need to finish this data processing step
```

There is another thing that can improve the results markedly, which is to normalise the data, sometimes also known as standardisation. We will look at this more in Section 3.3.1, but the basic idea is to ensure that the values in the data are not too large, because then the weights will have to be very small. The most common method of normalisation is to subtract off the mean of each variable (so that they each have zero mean) and divide by the variance. This is very easy in NumPy once you have worked out which `axis` is which: `axis=0` sums down the columns and `axis=1` sums across the rows. Note that only the input variables are normalised here. This is not always true, but here the target variable already has values 0 and 1, which are the possible outputs for the Perceptron, and we don't want to change that.

```
pima[:,:8] = pima[:,:8]-pima[:,:8].mean(axis=0)
pima[:,:8] = pima[:,:8]/pima[:,:8].var(axis=0)
```

There is one thing to be careful of here, which is that if you normalise the training and testing sets separately in this way then a datapoint that is in both sets will end up being different in the two, if since the mean and variance are probably different in the two sets. For this reason it is a good idea to normalise the dataset before splitting it into training and testing.

The last thing that we can do for now is to perform a basic form of feature selection and to try training the classifier with a subset of the inputs by missing out different features one at a time and seeing if they make the results better. If missing out one feature does improve the results, then leave it out completely and try missing out others as well. This is a simplistic way of testing for correlation between the output and each of the features. We will see better methods when we look at covariance in Section 8.2.2.

Now that we have seen how to use the Perceptron on a better example than the logic functions, we will look at another linear method, but coming from statistics, rather than neural networks.

2.4 Linear Regression

As is common in statistics, we need to separate out regression problems, where we fit a line to data, from classification problems, where we find a line that separates out the classes, so that they can be distinguished. However, it is common to turn classification problems into regression problems. This can be done in two ways, first by introducing an indicator variable, which simply says which class each datapoint belongs to. The problem is now to use the data to predict the indicator variable, which is a regression problem. The second approach is to do repeated regression, once for each class, with the indicator value being 1 for examples in the class and 0 for all of the others. Since classification can be replaced by regression using these methods, we'll think about regression here.

The only real difference between the Perceptron and more statistical approaches is in the way that the problem is set up. For regression we are making a prediction about an unknown value y (such as the indicator variable for classes or a future value of some data) by computing some function of known values x_i. We are thinking about straight lines, so the output y is going to be a sum of the x_i values, each multiplied by a constant parameter: $y = \sum_{i=0}^{M} \beta_i x_i$. The β_i define a straight line (plane in 3D, hyperplane in higher dimensions) that goes through (or at least near) the datapoints. Figure 2.12 shows this in two and three dimensions.

The question is how we define the line (plane or hyperplane in higher dimensions) that best fits the data. The most common solution is to try to minimise the distance between each datapoint and the line that we fit. We can measure the distance between a point and a line by defining another line that goes through the point and hits the line. School geometry tells us that this second line will be shortest when it hits the line at right angles, and then we can use Pythagorus' theorem to know the distance. Now, we can try to minimise an error function that measures the sum of all these distances. If we

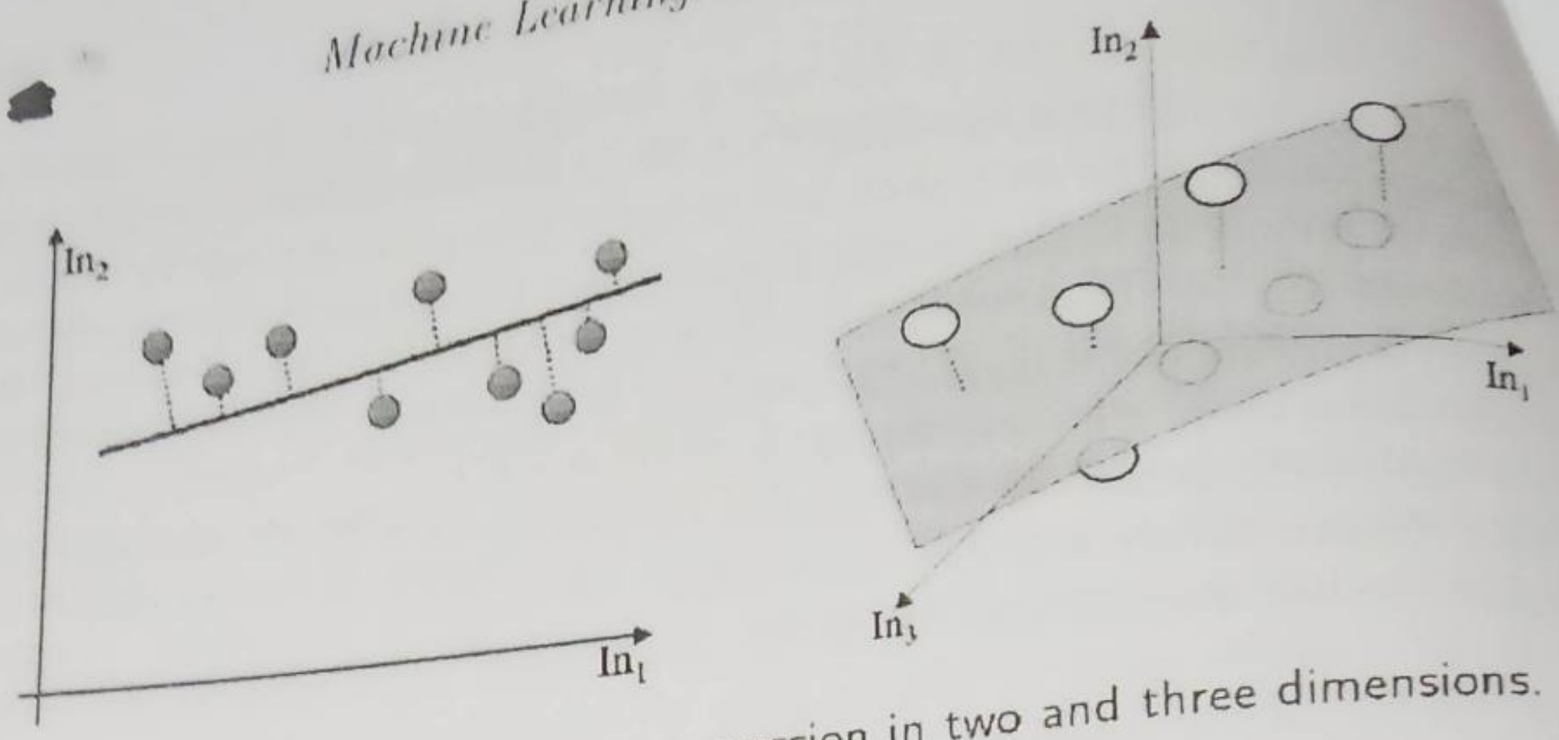


FIGURE 2.12: Linear regression in two and three dimensions.

ignore the square roots, and just minimise the sum-of-squares of the errors, then we get the most common minimisation, which is known as least-squares optimisation. What we are doing is choosing the parameters in order to minimise the squared difference between the prediction and the actual data value, summed over all of the datapoints. That is, we have:

$$\sum_{j=0}^{N} \left(t_j - \sum_{i=0}^{M} \beta_i x_{ij} \right)^2 . \tag{2.16}$$

This can be written in matrix form as:

$$(\mathbf{t} - \mathbf{X}\boldsymbol{\beta})(\mathbf{t} - \mathbf{X}\boldsymbol{\beta})^T, \tag{2.17}$$

where $\mathbf{t}$ is the targets and $\mathbf{X}$ is the matrix of input values (even including the bias inputs), just as for the Perceptron. Computing the smallest value of this means differentiating it with respect to the parameter vector $\boldsymbol{\beta}$ and setting the derivative to 0, which means that $\mathbf{X}^T(\mathbf{t}-\mathbf{X}\boldsymbol{\beta}) = 0$, which has the solution $\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ (assuming that the matrix $\mathbf{X}^T\mathbf{X}$ can be inverted). Now for a given input vector $\mathbf{z}$, the prediction is $\mathbf{z}\boldsymbol{\beta}$. The inverse of a matrix $\mathbf{X}$ is the matrix that satisfies $\mathbf{X}\mathbf{X}^{-1} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix, the matrix that has 1s on the leading diagonal and 0s everywhere else. The inverse of a matrix only exists if the matrix is square (has the same number of rows and columns) and its determinant is non-zero.

Computing this is very simple in Python, using the `linalg.inv()` function that is available in NumPy. In fact, the entire function can be written as:

```
from numpy import *

def linreg(inputs,targets):

    inputs = concatenate((-ones((shape(inputs)[0],1)),inputs),
```

```
axis=1)
beta = dot(dot(linalg.inv(dot(transpose(inputs),inputs)),
transpose(inputs)),targets)

outputs = dot(inputs,beta)
```

2.4.1 Linear Regression Examples

Using the linear regressor on the logical OR function seems a rather strange thing to do, since we are performing classification using a method designed explicitly for regression. However, we can do it, and it gives the following outputs:

```
[[ 0.25]
 [ 0.75]
 [ 0.75]
 [ 1.25]]
```

It might not be clear what this means, but if we threshold the outputs by setting every value less than 0.5 to 0 and every value above 0.5 to 1, then we get the correct answer. Using it on the XOR function shows that this is still a linear method:

```
[[ 0.5]
 [ 0.5]
 [ 0.5]
 [ 0.5]]
```

A better test of linear regression is to find a real regression dataset. The UCI database is useful here, as well. We will look at the `auto-mpg` dataset. This consists of a collection of number of datapoints about certain cars (weight, horsepower, etc.), with the aim being to predict the fuel efficiency in miles per gallon (mpg). This dataset has one problem. There are missing values in it (labelled with question marks '?'). The `loadtxt()` method doesn't like these, and we don't know what to do with them, anyway, so after downloading the dataset, manually edit the file and delete all lines where there is a ? in that line. The linear regressor can't do much with the names of the cars either, but since they appear in quotes (") we will tell `loadtxt` that they are comments, using:

```
axis=1)
beta = dot(dot(linalg.inv(dot(transpose(inputs),inputs)),
transpose(inputs)),targets)

outputs = dot(inputs,beta)
```

2.4.1 Linear Regression Examples

Using the linear regressor on the logical OR function seems a rather strange thing to do, since we are performing classification using a method designed explicitly for regression. However, we can do it, and it gives the following outputs:

```
[[ 0.25]
 [ 0.75]
 [ 0.75]
 [ 1.25]]
```

It might not be clear what this means, but if we threshold the outputs by setting every value less than 0.5 to 0 and every value above 0.5 to 1, then we get the correct answer. Using it on the XOR function shows that this is still a linear method:

```
[[ 0.5]
 [ 0.5]
 [ 0.5]
 [ 0.5]]
```

A better test of linear regression is to find a real regression dataset. The UCI database is useful here, as well. We will look at the `auto-mpg` dataset. This consists of a collection of number of datapoints about certain cars (weight, horsepower, etc.), with the aim being to predict the fuel efficiency in miles per gallon (mpg). This dataset has one problem. There are missing values in it (labelled with question marks '?'). The `loadtxt()` method doesn't like these, and we don't know what to do with them, anyway, so after downloading the dataset, manually edit the file and delete all lines where there is a ? in that line. The linear regressor can't do much with the names of the cars either, but since they appear in quotes (") we will tell `loadtxt` that they are comments, using:

```
auto = loadtxt('/Users/srmarsla/Book/Datasets/auto-mpg/auto-
mpg.data.txt',comments='"')
```

You should now separate the data into training and testing sets, and then use the training set to recover the β vector. Then you use that to get the predicted values on the test set. However, the confusion matrix isn't much use now, since there are no classes to enable us to analyse the results. Instead, we will use the sum-of-squares error, which consists of computing the difference between the prediction and the true value, squaring them so that they are all positive, and then adding them up, as is used in the definition of the linear regressor. Obviously, small values of this measure are good. It can be computed using:

```
beta = linreg.linreg(trainin,traintgt)

testin = concatenate((-ones((shape(testin)[0],1)),testin),
axis=1)
testout = dot(testin,beta)
error = sum((testout - testtgt)**2)
```

Now you can test out whether normalising the data helps, and perform feature selection as we did for the Perceptron. There are other more advanced linear statistical methods. One of them, Linear Discriminant Analysis, will be considered in Section 10.1 once we have built up the understanding we need.

Further Reading

If you are interested in the historical aspects of the field, then the original paper on the Perceptron and the book that showed the requirement of linear separability (and that some people blame for putting the field back 20 years) still make interesting reads. Another paper that might be of interest is the review article written by Widrow and Lehr, which summarises some of the seminal work:

- F. Rosenblatt. The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6): 386–408, 1958.

- M.L. Minsky and S.A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge MA, 1969.

II UNIT-II Multi Layer perceptron

PART

In the last chapter we saw that while linear models are easy to understand and use, they come with the inherent cost that is implied by the word 'linear' that is they can only identify straight lines, planes, or hyperplanes. And this is not usually enough, because the majority of interesting problems are not linearly separable. In Section 2.3 we saw that problems can be made linearly separable if we can work out how to transform the features suitably. We will come back to this idea in Chapter 5, but in this chapter we will instead consider making more complicated networks.

We have pretty much decided that the learning in the neural network happens in the weights. So, to perform more computation it seems sensible to add more weights. There are two things that we can do: add some backward connections, so that the output neurons connect to the inputs again, or add more neurons. The first approach leads into recurrent networks. These have been studied, but are not that commonly used. We will instead consider the second approach. We can add neurons between the input nodes and the outputs, and this will make more complex neural networks, such as the one shown in Figure 3.1.

Recurrent networks

We will think about why adding extra layers of nodes makes a neural network more powerful in Section 3.3.3, but for now, to persuade ourselves that it is true, we can check that a prepared network can solve the two-dimensional XOR problem, something that we have seen is not possible for a linear model like the Perceptron. A suitable network is shown in Figure 3.2. To check that it gives the correct answers, all that is required is to put in each input and work through the network, treating it as two different Perceptrons, first computing the activations of the neurons in the middle layer (labelled as C and D in Figure 3.2) and then using those activations as the inputs to the single neuron at the output. As an example, I'll work out what happens when you put in $(1, 0)$ as an input; the job of checking the rest is up to you.

Input $(1, 0)$ corresponds to node A being 1 and B being 0. The input to neuron C is therefore $-1 \times 0.5 + 1 \times 1 + 0 \times 1 = -0.5 + 1 = 0.5$. This is above the threshold of 0, and so neuron C fires, giving output 1. For neuron D the input is $-1 \times 1 + 1 \times 1 + 0 \times 1 = -1 + 1 = 0$, and so it does not fire, giving output 0. Therefore the input to neuron E is $-1 \times 0.5 + 1 \times 1 + 0 \times -1 = 0.5$, so neuron E fires. Checking the result of the inputs should persuade you that neuron E fires when inputs A and B are different to each other, but does not

Chapter 3

The Multi-Layer Perceptron

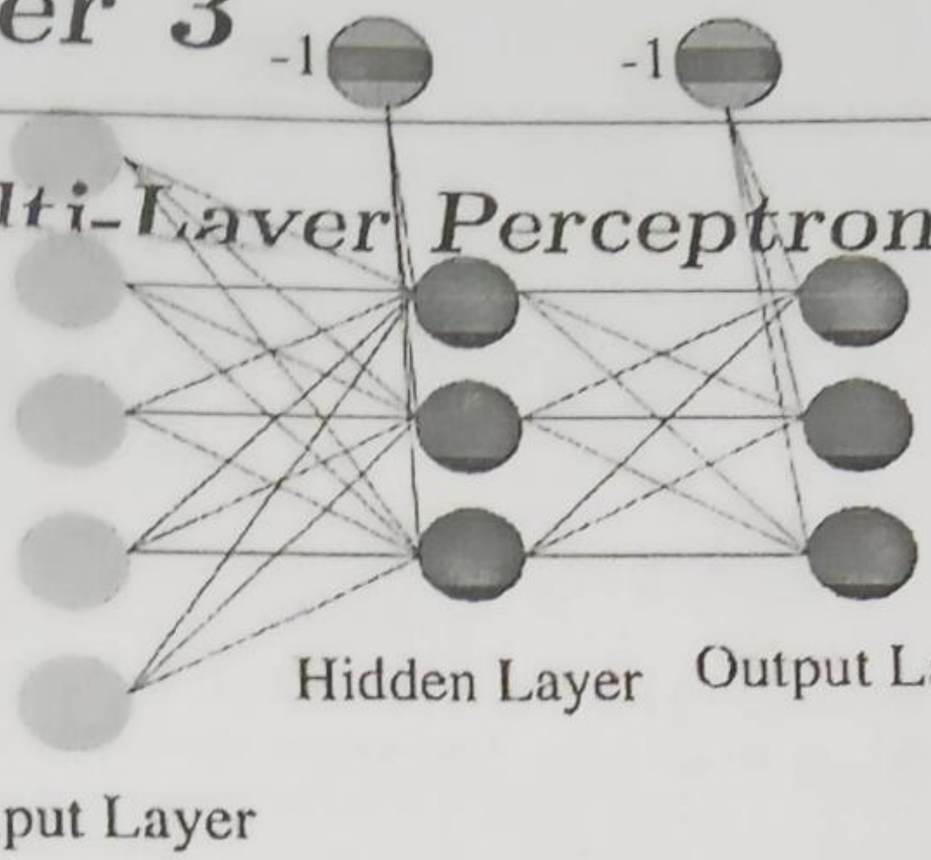


FIGURE 3.1: The Multi-Layer Perceptron network, consisting of multiple layers of connected neurons.

fire when they are the same, which is exactly the XOR function (it doesn't matter that the fire and not fire have been reversed).

So far, so good. Since this network can solve a problem that the Perceptron cannot, it seems worth looking into further. However, now we've got a much more interesting problem to solve, namely how can we train this network so that the weights are adapted to generate the correct (target) answers? If we try the method that we used for the Perceptron we need to compute the error at the output. That's fine, since we know the targets there, so we can compute the difference between the targets and the outputs. But now we don't know which weights were wrong: those in the first layer, or the second? Worse, we don't know what the correct activations are for the neurons in the middle of the network. This fact gives the neurons in the middle of the network their name, they are called the hidden layer (or layers), because it isn't possible to examine and correct their values directly.

It took a long time for people who studied neural networks to work out how to solve this problem. In fact, it wasn't until 1986 that Rumelhart, Hinton, and McClelland managed it. However, a solution to the problem was already known by statisticians and engineers they just didn't know that it was a problem in neural networks! In this chapter we are going to look at the neural network solution proposed by Rumelhart, Hinton, and McClelland, the Multi-Layer Perceptron (MLP), which is still one of the most commonly used machine learning methods around. Getting to the stage where we understand how it works and what we can do with it is going to take us into lots of different areas of statistics, mathematics and computer science, so we'd better get started.

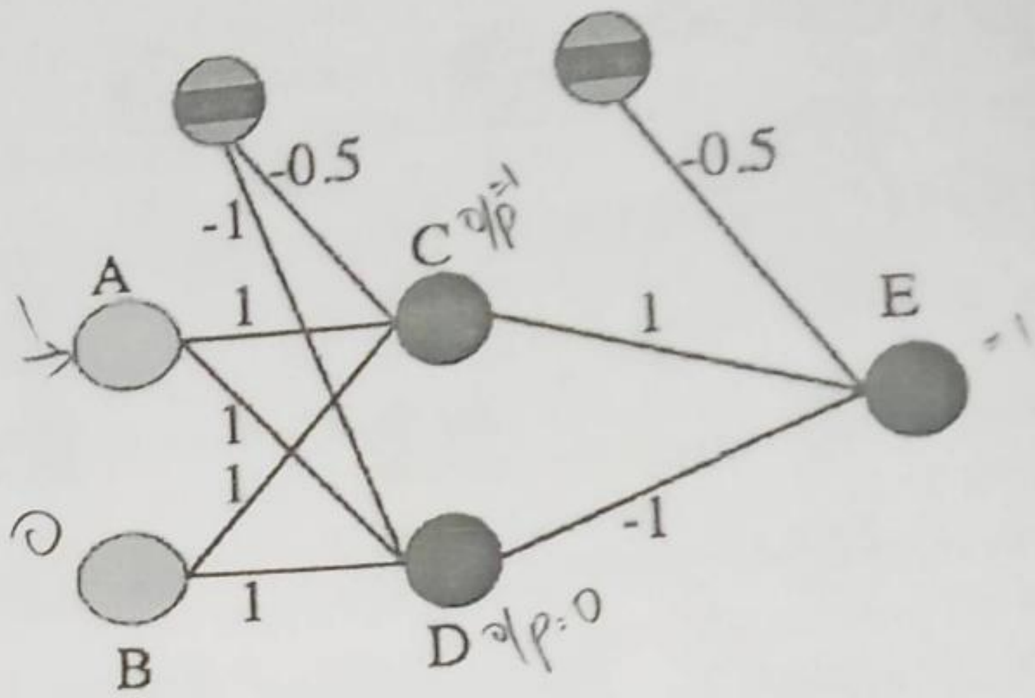


FIGURE 3.2: A Multi-Layer Perceptron network showing a set of weights that solve the XOR problem.

3.1 Going Forwards

Just like in the Perceptron, training the MLP consists of two parts: working out what the outputs are for the given inputs and the current weights, and then updating the weights according to the error, which is a function of the difference between the outputs and the targets. These are generally known as going forwards and backwards through the network. We've already seen how to go forwards for the MLP when we saw the XOR example above, which was effectively the recall phase of the algorithm. It is pretty much just the same as the Perceptron, except that we have to do it twice, once for each set of neurons, and we need to do it layer by layer, because otherwise the input values to the second layer don't exist. In fact, having made an MLP with two layers of nodes, there is no reason why we can't make one with 3, or 4, or 20 layers of nodes (we'll discuss whether or not you might want to in Section 3.3.3). This won't even change our recall (forward) algorithm much, since we just work forwards through the network computing the activations of one layer of neurons and using those as the inputs for the next layer.

So looking at Figure 3.1, we start at the left by filling in the values for the inputs. We then use these inputs and the first level of weights to calculate the activations of the hidden layer, and then we use those activations and the next set of weights to calculate the activations of the output layer. Now that we've got the outputs of the network, we can compare them to the targets and compute the error.

3.1.1 Biases

Just like in the Perceptron case, we need to include a bias input to each neuron. We do this in the same way, by having an extra input that is permanently set to -1, and adjusting the weights to each neuron as part of the training. Thus, each neuron in the network (whether it is a hidden layer or the output) has 1 extra input, with fixed value.

3.2 Going Backwards: Back-Propagation of Error

It is in the backwards part of the algorithm that things get tricky. Computing the errors at the output is no more difficult than it was for the Perceptron, but working out what to do with those errors is more difficult. The method that we are going to look at is called **back-propagation of error**, which makes it clear that the errors are sent backwards through the network. It is a form of **gradient descent** (which is described briefly below, and also given its own section in Chapter 11).

The best way to describe back-propagation properly is mathematically, but this can be intimidating and difficult to get a handle on at first. I've therefore tried to compromise by using words and pictures in the main text, but putting all of the mathematical details into Section 3.6. While you should look at that section and try to understand it, it can be skipped if you really don't have the background. Although it looks complicated, there are actually three things that you need to know, all of which are from differential calculus: the derivative of $\frac{1}{2}x^2$, the fact that if you differentiate a function of x with respect to some other variable t, then the answer is 0, and the chain rule, which tells you how to differentiate composite functions.

When we talked about the Perceptron, we changed the weights so that the neurons fired when the targets said they should, and didn't fire when the targets said they shouldn't. Although we didn't say it like this then, what we did was to choose an **error function** $E = t - y$, and tried to make it as small as possible. Since there was only one set of weights in the network, this was sufficient to train the network.

We still want to do the same thing—minimise the error, so that neurons fire only when they should—but, with the addition of extra layers of weights, this is harder to arrange. The problem is that when we try to adapt the weights of the Multi-Layer Perceptron, we have to work out which weights caused the error. This could be the weights connecting the inputs to the hidden layer, or the weights connecting the hidden layer to the output layer (for more complex networks, there could be extra weights between nodes in hidden layers. This isn't a problem—the same method works—but it is more confusing to talk about, so I'm only going to worry about one hidden layer here).

FIGURE 3.3: The weights of the network are trained so that the error goes downhill until it reaches a local minimum, just like a ball rolling under gravity.

The error function that we used for the Perceptron was $E = t - y$. However, suppose that we make two errors. In the first, the target is bigger than the output, while in the second the output is bigger than the target. If these two errors are the same size, then if we add them up we could get 0, which means that the algorithm thinks that no error was made. To get around this we need to make all errors have the same sign. We can do this in a few different ways, but the one that will turn out to be best is the sum-of-squares error function, which calculates the difference between t and y for each node, squares them, and adds them all together:

$$E(\mathbf{t}, \mathbf{y}) = \frac{1}{2}\sum_{k=1}^{n}(t_k - y_k)^2. \tag{3.1}$$

You might have noticed the $\frac{1}{2}$ at the front of that equation. It doesn't matter that much, but it makes it easier when we differentiate the function, and that is the name of the game here: if we differentiate a function, then it tells us the gradient of that function, which is the direction along which it increases and decreases the most. So if we differentiate an error function, we get the gradient of the error. Since the purpose of learning is to minimise the error, following the error function downhill (in other words, in the direction of the negative gradient) will give us what we want. Imagine a ball rolling around on a surface that looks like the line in Figure 3.3. Gravity will make the ball roll downhill (follow the downhill gradient) until it ends up in the bottom of one of the hollows. These are places where the error is small, so that is exactly what we want. This is why the algorithm is called gradient descent. So what should we differentiate with respect to? There are only three things in the network that change: the inputs, the activation function that decides whether or not the node fires, and the weights. The first and second are out of our control when the algorithm is running, so only the weights matter, and therefore they are what we differentiate with respect to.

Having mentioned the activation function, this is a good time to point out a little problem with the threshold function that we have been using for

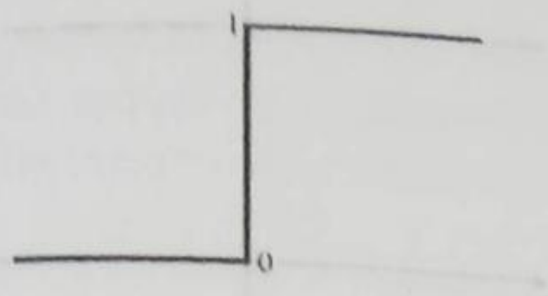


FIGURE 3.4: The threshold function that we used for the Perceptron. Note the discontinuity where the value changes from 0 to 1.

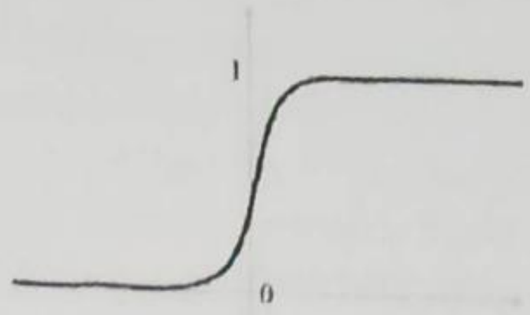


FIGURE 3.5: The sigmoid function, which looks qualitively fairly similar, but varies smoothly and differentiably.

our neurons so far, which is that it is discontinuous (see Figure 3.4; it has a sudden jump in the middle) and so differentiating it at that point isn't possible. The problem is that we need that jump between firing and not firing to make it act like a neuron. We can solve the problem if we can find an activation function that looks like a threshold function, but is differentiable so that we can compute the gradient. If you squint at a graph of the threshold function (for example, Figure 3.4) then it looks kind of S-shaped. There is a mathematical form of S-shaped functions, called sigmoid functions (see Figure 3.5). They have another nice property, which is that their derivative also has a nice form, as is shown in Section 3.6.3 for those who know some mathematics. The most commonly used form of this function (where β is some positive parameter) is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}. \tag{3.2}$$

In some texts you will see the activation function given a different form, as:

$$a = g(h) = \tanh(h) = \frac{\exp(h) - \exp(-h)}{\exp(h) + \exp(-h)}, \tag{3.3}$$

which is the hyperbolic tangent function. This is a different but similar function: it is still a sigmoid function, but it saturates (reaches its constant values) at ± 1 instead of 0 and 1, which is sometimes useful. It also has a relatively simple derivative: $\frac{d}{dx}\tanh x = (1 - \tanh^2(x))$. We can convert between the two easily, because if the saturation points are (± 1), then we can convert to $(0, 1)$ by using $0.5 \times (x + 1)$.

So now we've got a new form of error computation and a new activation function that decides whether or not a neuron should fire. We can differentiate it, so that when we change the weights, we do it in the direction that is downhill for the error, which means that we know we are improving the error

function of the network. As far as an algorithm goes, we've fed our inputs forward through the network and worked out which nodes are firing. Now, at the output, we've computed the errors as the sum-squared difference between the targets and the outputs (Equation (3.1) above). What we want to do next is to compute the gradient of these errors and use them to decide how much to update each weight in the network. We will do that first for the nodes connected to the output layer, and after we have updated those, we will work *backwards* through the network until we get back to the inputs again. There are just two problems:

- for the output neurons, we don't know the inputs.
- for the hidden neurons, we don't know the targets; for extra hidden layers, we know neither the inputs nor the targets, but even this won't matter for the algorithm we derive.

So we can compute the error at the output, but since we don't know what the inputs were that caused it, we can't update those second layer weights the way we did for the Perceptron. If we use the chain rule of differentiation that you all (possibly) remember from high school then we can get around this problem. Here, the chain rule tells us that if we want to know how the error changes as we vary the weights, we can think about how the error changes as we vary the inputs to the weights, and multiply this by how those input values change as we vary the weights. This is useful because it lets us calculate all of the derivatives that we want to: we can write the activations of the output nodes in terms of the activations of the hidden nodes and the output weights, and then we can send the error calculations back through the network to the hidden layer to decide what the target outputs were for those neurons. Note that we can do exactly the same computations if the network has extra hidden layers between the inputs and the outputs. It gets harder to keep track of which functions we should be differentiating, but there are no new tricks needed.

Chain Rule

All of the relevant equations are derived in Section 3.6, and you should read that section carefully, since it is quite difficult to describe exactly what is going on here in words. The important thing to understand is that we compute the gradients of the errors with respect to the weights, so that we change the weights so that we go downhill, which makes the errors get smaller. We do this by differentiating the error function with respect to the weights, but we can't do this directly, so we have to apply the chain rule and differentiate with respect to things that we know. This leads to two different update functions, one for each of the sets of weights, and we just apply these backwards through the network, starting at the outputs and ending up back at the inputs.

II - 3

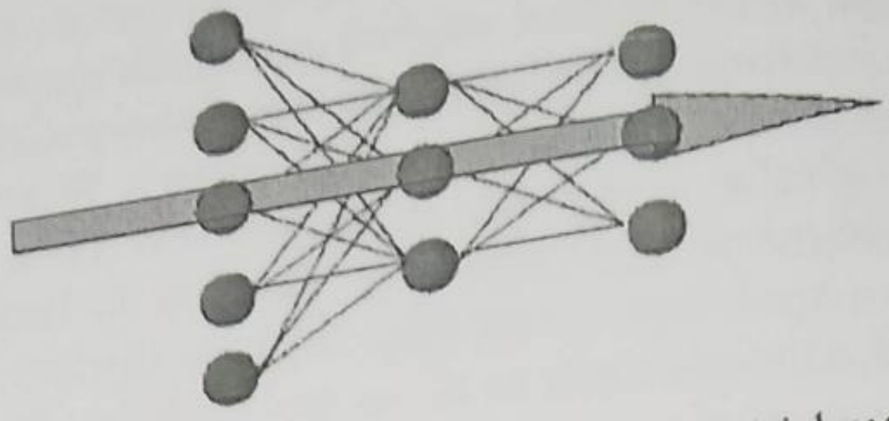


FIGURE 3.6: The forward direction in a Multi-Layer Perceptron.

3.2.1 The Multi-Layer Perceptron Algorithm

We'll get into the details of the basic algorithm here, and then, in the next section, have a look at some practical issues, such as how much training data is needed, how much training time is needed, and how to choose the correct size of network. The MLP is one of the most common neural networks in use. It is often treated as a 'black box,' in that people use it without understanding how it works, which often results in fairly poor results. Here is a quick summary of how the algorithm works, and then the full MLP training algorithm using back-propagation of error is described.

1. an input vector is put into the input nodes
2. the inputs are fed *forward* through the network (Figure 3.6)
   - the inputs and the first-layer weights (here labelled as v) are used to decided whether the hidden nodes fire or not. The activation function $g(\cdot)$ is the sigmoid function given in Equation (3.2) above
   - the outputs of these neurons and the second-layer weights (labelled as w) are used to decide if the output neurons fire or not
3. the *error* is computed as the sum-of-squares difference between the network outputs and the targets
4. this error is fed *backwards* through the network in order to
   - first update the second-layer weights by using the δ_o errors
   - and then afterwards, the first-layer weights by using the δ_h errors

---

The Multi-Layer Perceptron Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values

- **Training**
  - repeat:
    - for each input vector:
      Forwards phase:
      - compute the activation of each neuron j in the hidden layer(s) using:

$$h_j = \sum_i x_i v_{ij} \tag{3.4}$$

$$a_j = g(h_j) = \frac{1}{1 + \exp(-\beta h_j)} \tag{3.5}$$

      - work through the network until you get to the output layers, which have activations:

$$h_k = \sum_j a_j w_{jk} \tag{3.6}$$

$$y_k = g(h_k) = \frac{1}{1 + \exp(-\beta h_k)} \tag{3.7}$$

      Backwards phase:
      - compute the error at the output using:

$$\delta_{ok} = (t_k - y_k)\, y_k (1 - y_k) \tag{3.8}$$

      - compute the error in the hidden layer(s) using:

$$\delta_{hj} = a_j (1 - a_j) \sum_k w_{jk} \delta_{ok} \tag{3.9}$$

      - update the output layer weights using:

$$w_{jk} \leftarrow w_{jk} + \eta \delta_{ok} a_j^{\text{hidden}} \tag{3.10}$$

      - update the hidden layer weights using:

$$v_{ij} \leftarrow v_{ij} + \eta \delta_{hj} x_i \tag{3.11}$$

    * randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops (see Section 3.3.6)
- **Recall**
  - use the Forwards phase in the training section above

This provides a description of the basic algorithm. There are a couple of things that weren't mentioned beforehand, including the randomisation of the order of the input vectors. It turns out that this can significantly improve the speed with which the algorithm learns. NumPy has a useful function that assists with this, `random.shuffle()`, which takes a list of numbers and reorders them. It can be used like this:

```
random.shuffle(change)
inputs = inputs[change,:]
targets = targets[change,:]
```

As with the Perceptron, a NumPy implementation can take advantage of various matrix multiplications, which makes things easy to read and faster to compute. The implementation on the website is a batch version of the algorithm, so that weight updates are made after all of the input vectors have been presented (as is described in Section 3.2.4). The central weight update computations for the algorithm can be implemented as:

```
deltao = (targets-self.outputs)*self.outputs*(1.0-self.outputs)
deltah = self.hidden*(1.0-self.hidden)*(dot(deltao,transpose(
self.weights2)))

updatew1 = zeros((shape(self.weights1)))
updatew2 = zeros((shape(self.weights2)))

updatew1 = eta*(dot(transpose(inputs),deltah[:,:-1]))
updatew2 = eta*(dot(transpose(self.hidden),deltao))
self.weights1 += updatew1
self.weights2 += updatew2
```

There are a few improvements that can be made to the algorithm, and there are some important things that need to be considered, such as how many training datapoints are needed, how many hidden nodes should be used, and how much training the network needs. We will look at the improvements first, and then move on to practical considerations in Section 3.3. There are lots of details that are given in this section because it is one of the early examples in the book; later on things will be skipped over more quickly.

The first thing that we can do is check that this MLP can indeed learn the logic functions, especially the XOR. We can do that with this code (which is function `logic` on the website):

```
from numpy import *
import mlp

anddata = array([[0,0,0],[0,1,0],[1,0,0],[1,1,1]])
xordata = array([[0,0,0],[0,1,1],[1,0,1],[1,1,0]])

p = mlp.mlp(anddata[:,0:2],anddata[:,2:3],2)
p.mlptrain(anddata[:,0:2],anddata[:,2:3],0.25,1001)
p.confmat(anddata[:,0:2],anddata[:,2:3])
```

```
q = mlp.mlp(xordata[:,0:2],xordata[:,2:3],2)
q.mlptrain(xordata[:,0:2],xordata[:,2:3],0.25,5001)
q.confmat(xordata[:,0:2],xordata[:,2:3])
```

The outputs that this produces is something like:

```
Iteration:  0  Error:  0.367917569871
Iteration:  1000  Error:  0.0204860723612
Confusion matrix is:
[[ 3.  0.]
 [ 0.  1.]]
Percentage Correct:  100.0
Iteration:  0  Error:  0.515798627074
Iteration:  1000  Error:  0.499568173798
Iteration:  2000  Error:  0.498271692284
Iteration:  3000  Error:  0.480839047738
Iteration:  4000  Error:  0.382706753191
 Iteration:  5000  Error:  0.0537169253359
Confusion matrix is:
[[ 2.  0.]
 [ 0.  2.]]
Percentage Correct:  100.0
```

There are a few things to notice about this. One is that it does work, producing the correct answers, but the other is that even for the AND we need significantly more iterations than we did for the Perceptron. So the benefits of a more complex network come at a cost, because it takes substantially more computational time to fit those weights to solve the problem, even for linear examples. Sometimes, even 5000 iterations are not enough for the XOR function, and more have to be added.

3.2.2 Initialising the Weights

The MLP algorithm suggests that the weights are initialised to small random numbers, both positive and negative. The question is how small is small, and does it matter? One way to get a feeling for this would be to experiment with the code, setting all of the weights to 0, and seeing how well the network learns, then setting them all to large numbers and comparing the results. However, to understand why they should be small we can look at the shape of the sigmoid. If the initial weight values are close to 1 or -1 (which is what we mean by large here) then the inputs to the sigmoid are also likely to be close to ± 1 and so the output of the neuron is either 0 or 1 (the sigmoid has saturated, reached its maximum or minimum value). If the weights are very

small (close to zero) then the input is still close to 0 and so the output of the neuron is just linear, so we get a linear model. Both of these things can be useful for the final network, but if we start off with values that are inbetween it can decide for itself.

Choosing the size of the initial values needs a little more thought, then. Each neuron is getting input from n different places (either input nodes if the neuron is in the hidden layer, or hidden neurons if it is in the output layer). If we view the values of these inputs as having uniform variance, then the typical input to the neuron will be $w\sqrt{n}$, where w is the initialisation value of the weights. So a common trick is to set the weights in the range $-1/\sqrt{n} < w < 1/\sqrt{n}$, where n is the number of nodes in the input layer to those weights. This makes the total input to a neuron have a maximum size of about 1. We use random values in this range so that the learning starts off from different places for each run, and we keep them all about the same size because we want all of the weights to reach their final values at about the same time. This is known as uniform learning and it is important because otherwise the network will do better on some inputs than others.

3.2.3 Different Output Activation Functions

In the algorithm described above, we used sigmoid neurons in the hidden layer and the output layer. This is fine for classification problems since there we can make the classes be 0 and 1. However, we might also want to perform regression problems, where the output needs to be from a continuous range, not just 0 or 1. The sigmoid neurons at the output are not very useful in that case. We can replace the output neurons with linear nodes that just sum the inputs and give that as their activation (so $g(h) = h$ in the notation of Equation (3.2)). This does not mean that we change the hidden layer neurons, they stay exactly the same. These output nodes are not models of neurons anymore, since they don't have the characteristic fire/don't fire pattern. Even so, they can be useful, for example for regression problems, where we want a real number out, not just a 0/1 decision.

There is a third type of output neuron that is also used, which is the soft-max activation function. This is most commonly used for classification problems where the 1-of-N output encoding is used, as is described in Section 3.4.2. The soft-max function rescales the outputs by calculating the exponential of the inputs to that neuron, and dividing by the total sum of the inputs to all of the neurons, so that the activations sum to 1 and all lie between 0 and 1. As an activation function it can be written as:

$$y_k = g(h_k) = \frac{\exp(h_k)}{\sum_{\hat{h}_k} \exp(\hat{h}_k)}. \tag{3.12}$$

Of course, if we change the activation function, then the derivative of the activation function will also change, and so the learning rule will be different.

The changes that need to be made to the algorithm are in Equations (3.7) and (3.8). For the linear activation function the first is replaced by:

$$y_k = g(h_k) = h_k, \tag{3.13}$$

while the second is replaced by:

$$\delta_{ok} = (t_k - y_k). \tag{3.14}$$

For the soft-max activation, the update equation that replaces (3.8) is (3.14), just as for the linear output. Computing these update equations requires computing the error function that is being optimised, and then differentiating it. These additions can be added into the code by allowing the user to specify the type of output activation, which has to be done twice, once in the `mlpfwd` function, and once in the `mlptrain` function. In the former, the new piece of code can be written as:

```
if self.outtype == 'linear':
    return outputs
elif self.outtype == 'logistic':
    return 1.0/(1.0+exp(-self.beta*outputs))
elif self.outtype == 'softmax':
    normalisers = sum(exp(outputs),axis=1)*ones((1,shape(
    outputs)[0]))
    return transpose(transpose(exp(outputs))/normalisers)
else:
    print "error"
```

3.2.4 Sequential and Batch Training

The MLP is designed to be a batch algorithm. All of the training examples are presented to the neural network, the average sum-of-squares error is then computed, and this is used to update the weights. Thus there is only one set of weight updates for each epoch (pass through all the training examples). This means that we only update the weights once for each iteration of the algorithm, which means that the weights are moved in the direction that most of the inputs want them to move, rather than being pulled around by each input individually. The batch method performs a more accurate estimate of the error gradient, and will thus converge to the local minimum more quickly.

The algorithm that was described earlier was the sequential version, where the errors are computed and the weights updated after each input. This is not guaranteed to be as efficient in learning, but it is simpler to program when using loops, and it is therefore much more common. Since it does not

converge as well, it can also sometimes avoid local minima, thus potentially reaching better solutions. While the description of the algorithm is sequential, the NumPy implementation on the book website is a batch version, because the matrix manipulation methods of NumPy make that easy. It is, however, relatively simple to modify it to use sequential update.

3.2.5 Local Minima

The driving force behind the learning rule is the minimisation of the network error by gradient descent (using the derivative of the error function to make the error smaller). This means that we are performing an optimisation: we are adapting the values of the weights in order to minimise the error function. As should be clear by now, the way that we are doing this is by approximating the gradient of the error and following it downhill so that we end up at the bottom of the slope. However, following the slope downhill only guarantees that we end up at a local minimum, a point that is lower than those close to it. If we imagine a ball rolling down a hill, it will settle at the bottom of a dip. However, there is no guarantee that it will have stopped at the lowest point – only the lowest point locally. There may be a much lower point over the next hill, but the ball can't see that, and it doesn't have enough energy to climb over the hill and find the global minimum (have another look at Figure 3.3 to see a picture of this).

Gradient descent works in the same way in two or more dimensions, and has similar (and worse) problems. The problem is that efficient downhill directions in two dimensions and higher are harder to compute locally. Standard contour maps provide beautiful images of gradients in our three-dimensional world, and if you imagine that you are walking in a hilly area aiming to get to the bottom of the nearest valley then you can get some idea of what is going on. Now suppose that you close your eyes, so that you can only feel which direction to go by moving one step and checking if you are higher up or lower down than you were. There will be places where going downwards as steeply as possible at the current point will not take you much closer to the valley bottom. There can be two reasons for this. The first is that you find a nearby local minimum, while the second is that sometimes the steepest direction is effectively across the valley, not towards the global minimum. This is shown in Figure 3.7.

All of these things are true for most of our optimisation problems, including the MLP. We don't know where the global minimum is because we don't know what the error landscape looks like; we can only compute local features of it for the place we are in at the moment. Which minimum we end up in depends on where we start. If we begin near the global minimum, then we are very likely to end up in it, but if we start near a local minimum we will probably end up there. In addition, how long it will take to get to the minimum that we do find depends upon the exact appearance of the landscape at the current point.

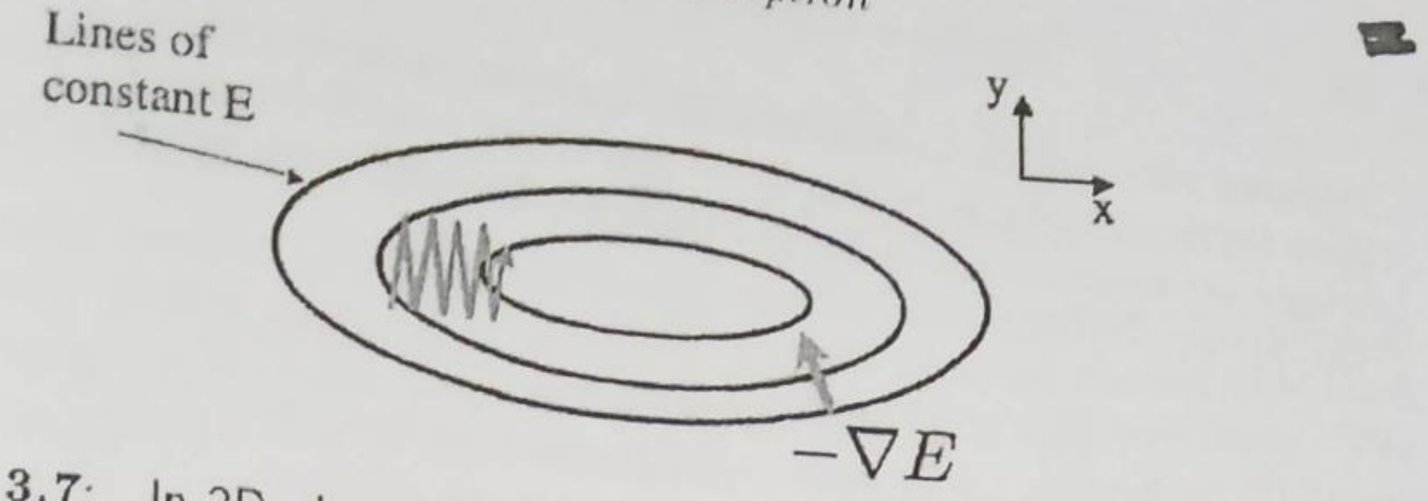


FIGURE 3.7: In 2D, downhill means at right angles to the lines of constant contour. Imagine walking down a hill with your eyes closed. If you find a direction that stays flat, then at right angles to that there may well be uphill or downhill. However, this is not the direction that takes you directly towards the local minimum.

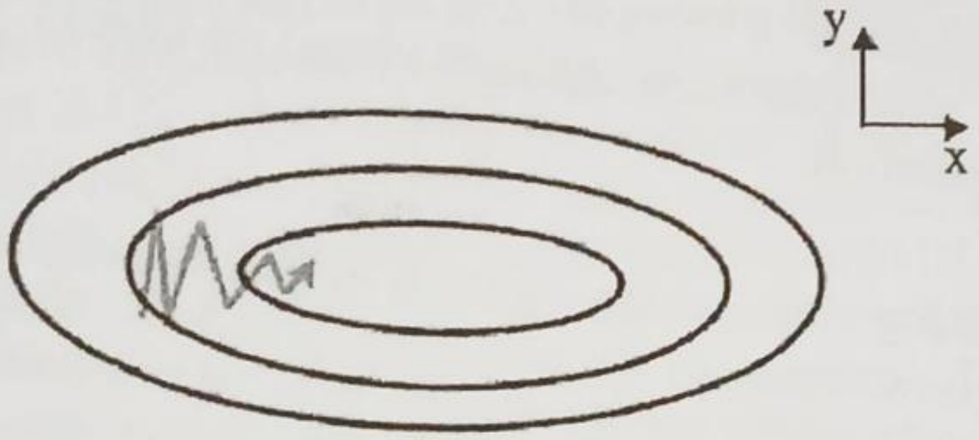


FIGURE 3.8: Adding momentum can help to avoid local minima, and also makes the dynamics of the optimisation more stable, improving convergence.

We can make it more likely that we find the global minimum by trying out several different starting points by training several different networks, and this is commonly done. However, we can also try to make it less likely that the algorithm will get stuck in local minima. There is a moderately effective way of doing this, which is discussed next.

3.2.6 Picking Up Momentum

Let's go back to the analogy of the ball rolling down the hill. The reason that the ball stops rolling is because it runs out of energy at the bottom of the dip. If we give the ball some weight, then it will generate momentum as it rolls, and so it is more likely to overcome a small hill on the other side of the local minimum, and so more likely to find the global minimum. We can implement this idea in our neural network learning by adding in some contribution from the previous weight change that we made to the current one. In two dimensions it will mean that the ball rolls more directly towards the valley bottom, since on average that will be the correct direction, rather than being controlled by the local changes. This is shown in Figure 3.8.

There is another benefit to momentum. It makes it possible to use a smaller

learning rate, which means that the learning is more stable. The only change that we need to make to the MLP algorithm is in Equations (3.10) and (3.11), where we need to add a second term to the weight updates so that they have the form:

$$w_{ij}^t = w_{ij}^{t-1} + \eta\delta_o a_j^{\text{hidden}} + \alpha\Delta w_{ij}^{t-1}, \tag{3.15}$$

where t is used to indicate the current update and $t-1$ is the previous one. Δw_{ij}^{t-1} is the previous update that we made to the weights (so $\Delta w_{ij}^t = +\eta\delta_o a_j^{\text{hidden}} + \alpha\Delta w_{ij}^{t-1}$) and $0 < \alpha < 1$ is the momentum constant. Typically a value of $\alpha = 0.9$ is used. This is a very easy addition to the code, and can improve the speed of learning a lot.

```
updatew1 = eta*(dot(transpose(inputs),deltah[:,:-1])) +
momentum*updatew1
updatew2 = eta*(dot(transpose(hidden),deltao)) +
momentum*updatew2
```

Weight decay

Another thing that can be added is known as weight decay. This reduces the size of the weights as the number of iterations increases. The argument goes that small weights are better since they lead to a network that is closer to linear (since they are close to zero, they are in the region where the sigmoid is increasing linearly), and only those weights that are essential to the non-linear learning should be large. After each learning iteration through all of the input patterns, every weight is multiplied by some constant $0 < \epsilon < 1$. This changes the learning quite a lot, but since it makes the network simpler it can often produce improved results. Unfortunately, it isn't fail-safe: occasionally it can make the learning significantly worse, so it should be used with care. Setting the value is typically done experimentally.

3.2.7 Other Improvements

Reduce Learning rate as the Algorithm progresses

There are a few other things that can be done to improve the convergence and behaviour of the back-propagation algorithm. One is to reduce the learning rate as the algorithm progresses. The reasoning behind this is that the network should only be making large-scale changes to the weights at the beginning, when the weights are random. If it is still making large weight changes later on, then something is wrong.

Something that results in much larger performance gains is to include information about the second derivatives of the error with respect to the weights. In the back-propagation algorithm we use the first derivatives to drive the learning. However, if we have knowledge of the second derivatives as well, we can use them as well to improve the network. Unfortunately, calculating the second derivatives is often computationally costly. These things are considered in more detail in Chapter 11.

3.3 The Multi-Layer Perceptron in Practice

The previous section looked at the design and implementation of the MLP network itself. In this section, we are going to look more at choices that can be made about the network in order to use it for solving real problems. We will then apply these ideas to using the MLP to find solutions to four different types of problem: regression, classification, time-series prediction, and data compression.

3.3.1 Data Preparation

The MLP, and indeed, pretty much every machine learning algorithm, tends to learn much more effectively if some preprocessing of the inputs and targets is performed before the network is trained. We saw some of these methods before, in Section 2.3.3, where we saw an example of using the Perceptron. For the targets it is fairly obvious that if using sigmoidal activation functions for the output, then the only possible target values should be 0 and 1. In fact, it is normal to scale the targets to lie between 0 and 1 no matter what kind of activation function is used for the output layer neurons. This helps to stop the weights from getting too large unnecessarily. Scaling the inputs also helps to avoid this problem. The most common approach to scaling the data for the MLP is to treat each data dimension independently, and then to either make each dimension have zero mean and unit variance in each dimension, or simply to scale them so that maximum value is 1 and the minimum -1. Both of these scalings have similar effects, but the first is a little bit better as it does not allow outliers to dominate as much. These scalings are commonly referred to as data normalisation, or sometimes standardisation. In fact, normalisation is not essential for the MLP, although it is usually beneficial. For some of the other networks that we will see, the normalisation will be essential.

In NumPy it is very easy to perform the normalisation by using the built-in `mean()` and `var()` functions; the only place where care is needed is along which axis the mean and variance are computed. For a dataset in the variable `data`, with each row being a datapoint, and targets `targets`:

```
data = (data - data.mean(axis=0))/data.var(axis=0)
targets = (targets - targets.mean(axis=0))/targets.var(axis=0)
```

3.3.2 Amount of Training Data

For the MLP with one hidden layer there are $(m+1) \times n + (n+1) \times p$ weights, where m, n, p are the number of nodes in the input, hidden, and output layers, respectively. The extra +1s come from the bias nodes, which

also have adjustable weights. This is a potentially huge number of adjustable parameters that we need to set during the training phase. Setting the values of these weights is the job of the back-propagation algorithm, which is driven by the errors coming from the training data. Clearly, the more training data there is, the better for learning, although the time that the algorithm takes to learn increases. Unfortunately, there is no way to compute what the minimum amount of data required is, since it depends on the problem. A rule of thumb that has been around for almost as long as the MLP itself is that you should use a number of training examples that is at least 10 times the number of weights. This is probably going to be a very large number of examples, so neural network training is a fairly computationally expensive operation, because we need to show the network all of these inputs lots of times.

3.3.3 Number of Hidden Layers

There are two other considerations concerning the number of weights that are inherent in the calculation above, which is the choice of the number of hidden nodes, and the number of hidden layers. Making these choices is obviously fundamental to the successful application of the algorithm. For reasons that we will see shortly, two hidden layers is the most that you ever need for normal MLP learning, but the bad news for choosing the number of hidden nodes is that there is no theory to guide it. You just have to experiment by training networks with different numbers of hidden nodes and then choosing the one that gives the best results, as we will see in Section 3.4.

We can use the back-propagation algorithm for a network with as many layers as we like, although it gets progressively harder to keep track of which weights are being updated at any given time. Fortunately, as was mentioned above, we will never normally need more than three layers (that is, two hidden layers and the output layer). This is because we can approximate any smooth functional mapping using a linear combination of localised sigmoidal functions. There is a sketchy demonstration of this using pictures in Figure 3.9. The basic idea is that by combining sigmoid functions we can generate ridge-like functions, and by combining ridge-like functions we can generate functions with a unique maximum. By combining these and transforming them using another layer of neurons, we obtain a localised response (a 'bump' function), and any functional mapping can be approximated to arbitrary accuracy using a linear combination of such bumps. The way that the MLP does this is shown in Figure 3.10. We will use this idea again when we look at approximating functions, for example using radial basis functions in Chapter 4.

Two hidden layers are sufficient to compute these bump functions for different inputs, and so if the function that we want to learn (approximate) is continuous, the network can compute it. It can therefore approximate any decision boundary, not just the linear one that the Perceptron computed.

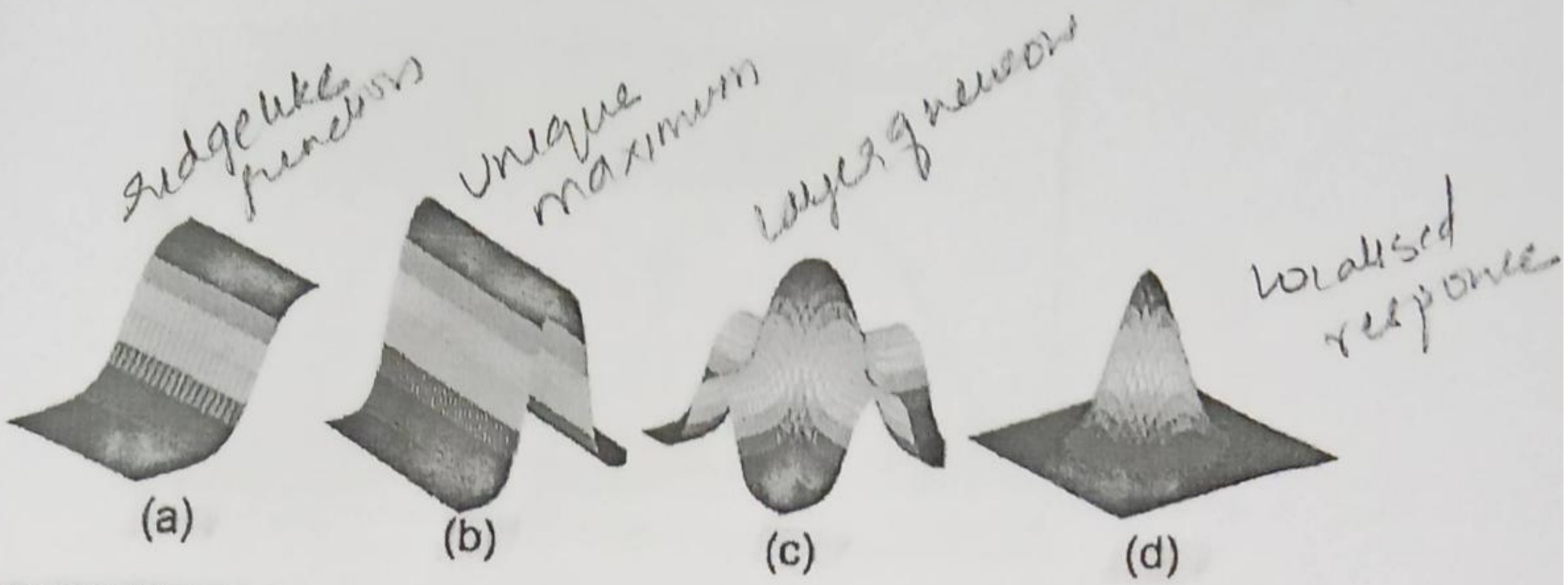


FIGURE 3.9: The learning of the MLP can be shown as (a) the output of a single sigmoidal neuron can be added to others (b), including reversed ones, to get a hill shape. Adding another hill at 90° produces (c) a bump, which can be sharpened (d) to any extent we want, with the bumps added together in the output layer. Thus the MLP learns a local representation of individual inputs.

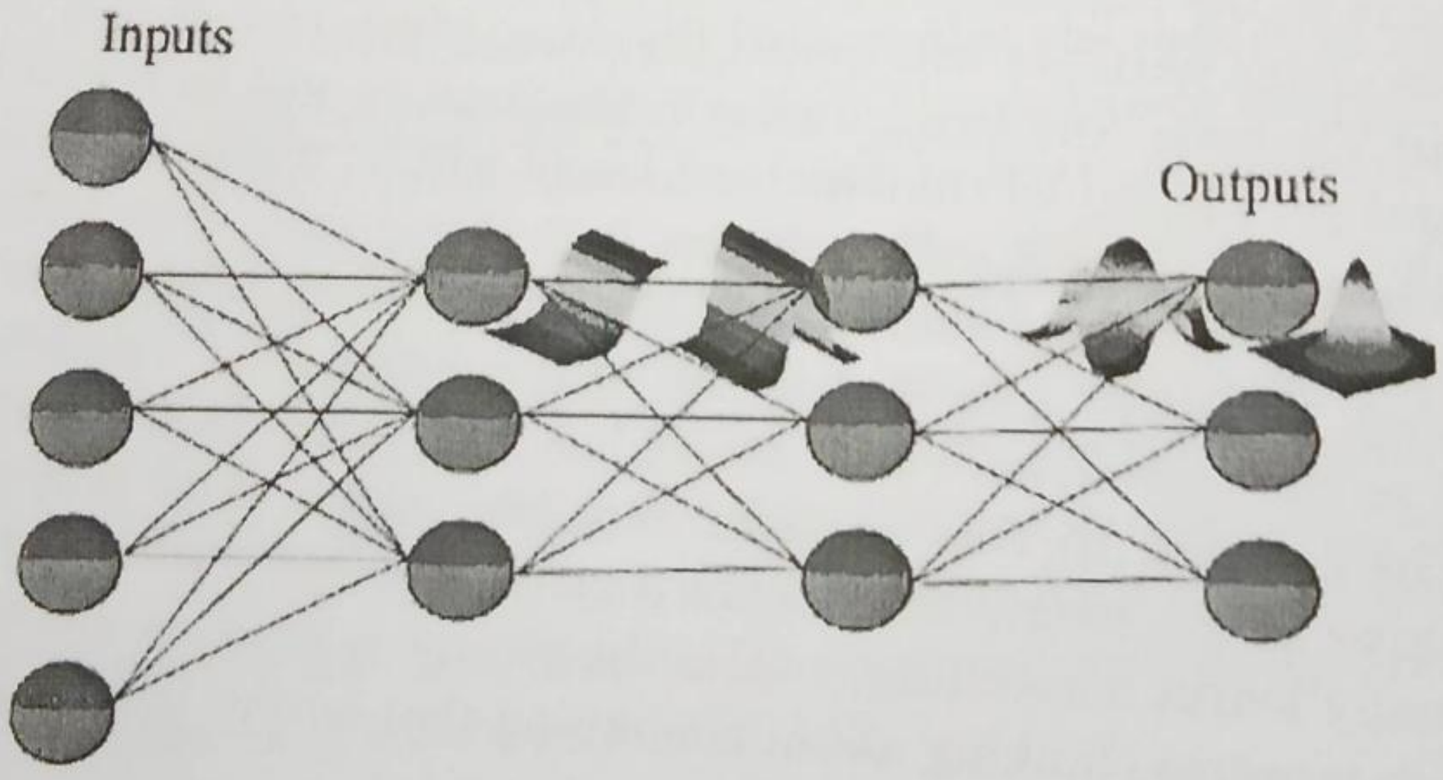


FIGURE 3.10: Schematic of the effective learning shape at each stage of the MLP.

II-48

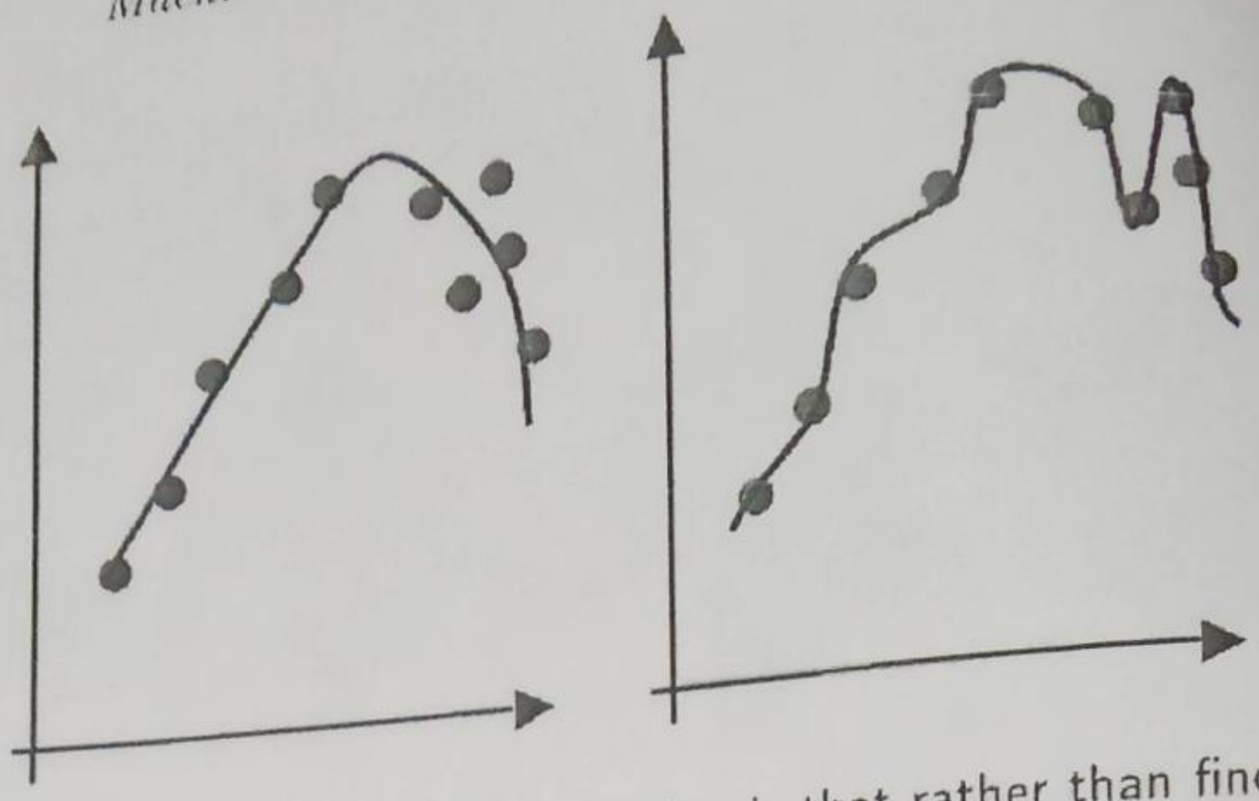


FIGURE 3.11: The effect of overfitting is that rather than finding the generating function (as shown on the left), the neural network matches the inputs perfectly, including the noise in them (on the right). This reduces the generalisation capabilities of the network.

3.3.4 Generalisation and Overfitting

The whole purpose of using a neural network is to generalise from the training examples to all possible inputs. We need to make sure that we do enough training that the network generalises well. However, there is at least as much danger in over-training the network as there is in under-training it. The number of degrees of variability in a neural network is huge — as discussed above, every weight can be varied. This is undoubtedly more variation than there is in the function we are learning, so we need to be careful: if we train for too long, then we will overfit the data, which means that we have learnt about the noise and inaccuracies in the data as well as the actual function. Therefore, the model that we have learnt will be much too complicated, and won't be able to generalise. Figure 3.11 shows this. Two different networks have learnt about the same data, but the one shown on the right has overfitted so that the curve goes through all of the datapoints, matching the noise as well as the underlying curve, which has been found in the graph on the left.

The solution to this problem is in two parts. The first is stopping the training before overfitting occurs (but not too early, so that the network has actually learnt something), while the second is working out when to do this, which requires thinking about something that is very important for the whole of machine learning: testing.

3.3.5 Training, Testing, and Validation

Whenever we train an MLP we are obviously going to test how well it works. We can compute the error of the trained network by computing the sum-of-squares error between the output and the target, just like we use to

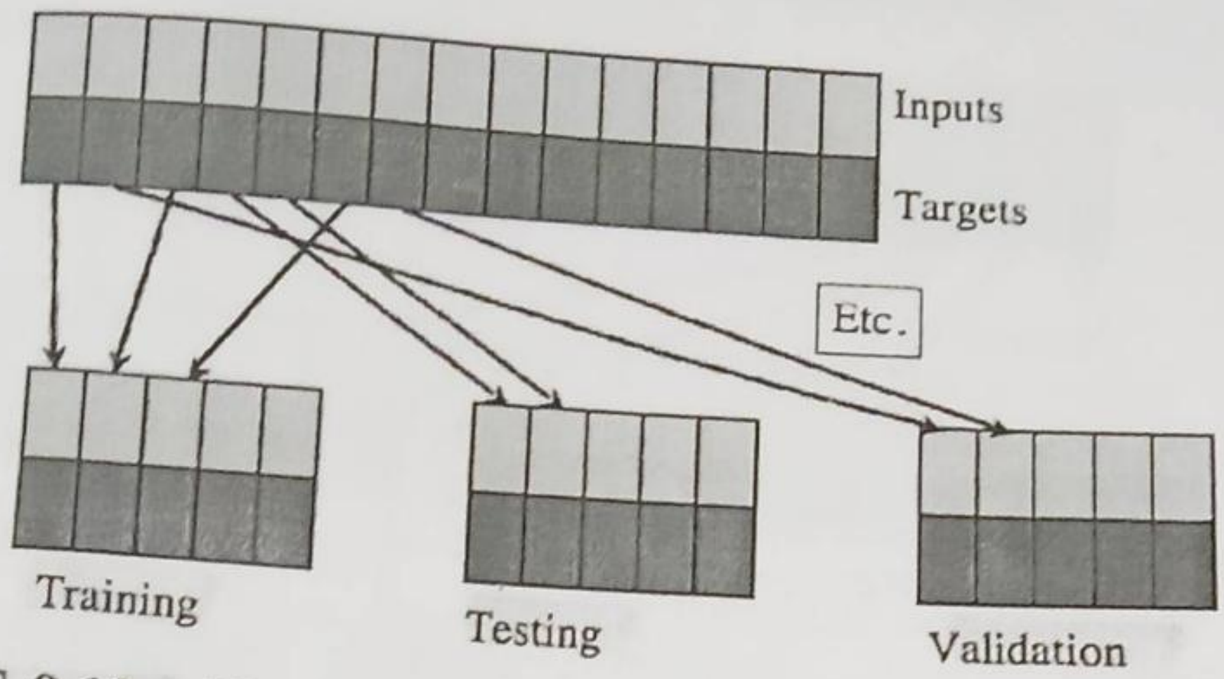


FIGURE 3.12: The dataset is split into different sets, some for training, some for validation, and some for testing.

drive the learning. What data should we test the network on? Clearly, it is not sensible to test using the same data that we trained on, since it would not tell us anything at all about how well the network generalises, nor anything about whether or not overfitting had occurred. We therefore need to keep a test set of (input, target) pairs in reserve that we don't use for training. The only problem with this is that it reduces the amount of data that we have available for testing, but that is something that we will just have to live with.

Things get more complicated when we consider checking how well the network is learning during training, so that we can decide when to stop. We can't use the training data for this, because we wouldn't detect overfitting, but we can't use the testing data either, because we're saving that for the final tests. We therefore keep a third set of data back, called the validation set because we're using it to validate the learning so far. This is known as cross-validation in statistics. The exact proportion of training to testing to validation data is up to you, but it is typical to do something like 50:25:25 if you have plenty of data, and 60:20:20 if you don't. How you do the splitting can also matter. Many datasets are presented with the first set of datapoints being in class 1, the next in class 2, and so on. If you pick the first few points to be the training set, the next the test set, etc. then the results are going to be pretty bad, since the training did not see all the classes. This can be dealt with by randomly reordering the data first, or by assigning each datapoint randomly to one of the sets, as is shown in Figure 3.12.

If you are really short of training data, so that if you have a separate validation set there is a worry that the network won't be sufficiently trained, then it is possible to perform leave-some-out, multi-fold cross-validation. The idea is shown in Figure 3.13. The dataset is randomly partitioned into K subsets, and one subset is used as a validation set, while the neural network is trained on all of the others. A different subset is then left out and a new network is trained on that subset, repeating the same process for all of the

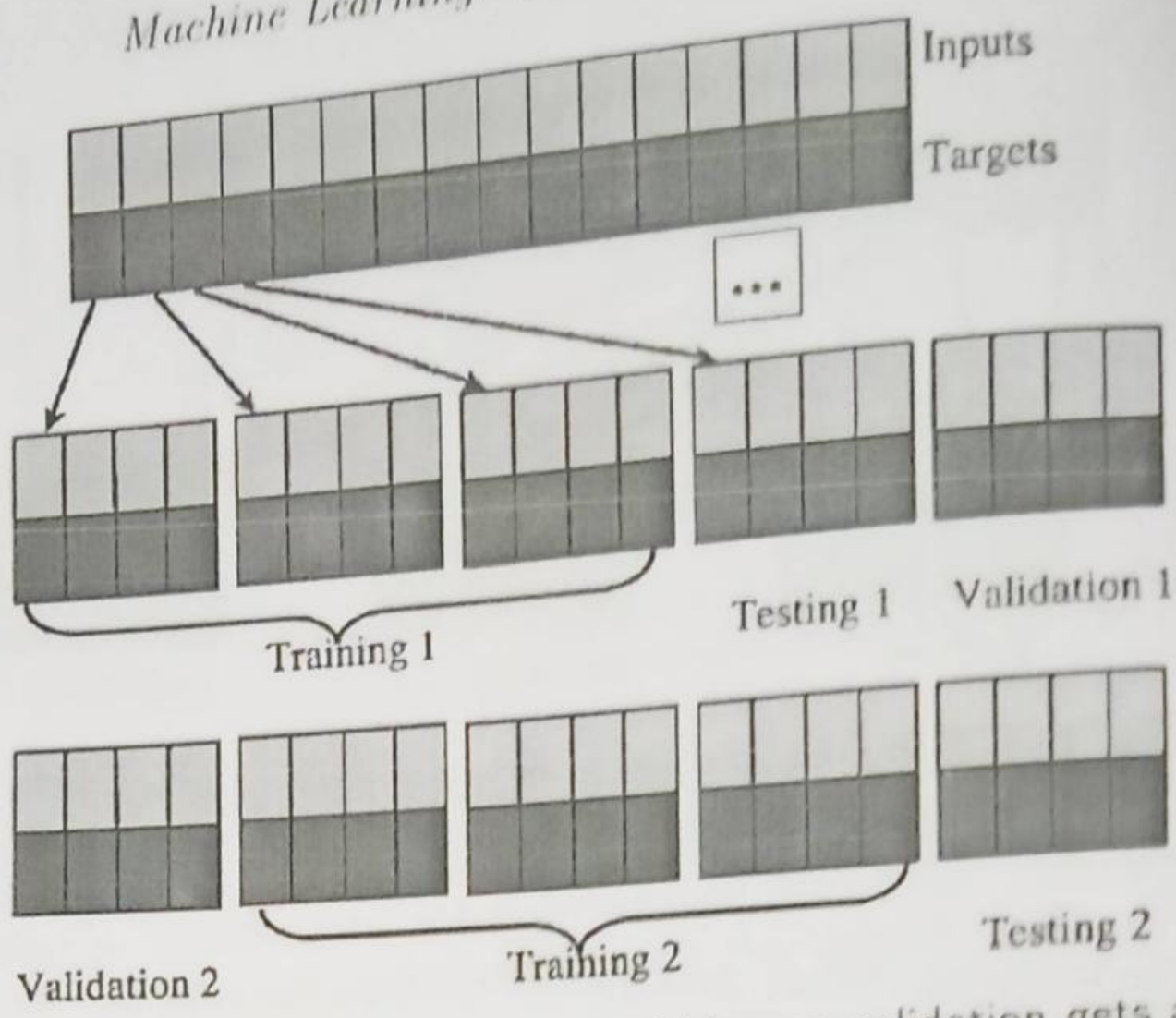


FIGURE 3.13: Leave-some-out, multi-fold cross-validation gets around the problem of data shortage by training many networks. It works by splitting the data into sets, training a network on most sets and holding one out for validation (and another for testing). Different networks are trained with different sets being held out.

different subsets. Finally, the network that produced the lowest validation error is tested and used. We've traded off data for computation time, since we've had to train K different networks instead of just one. In the most extreme case of this there is leave-one-out cross-validation, where the network is validated on just one piece of data, training on all of the rest.

3.3.6 When to Stop Learning

The training of the MLP requires that the algorithm runs over the entire dataset many times, with the weights changing as the network makes errors in each iteration. The question is how to decide when to stop learning, and this is a question that we are now ready to answer. It is unfortunate that the most obvious options are not sufficient: setting some predefined number N of iterations, and running until that is reached runs the risk that the network has overfitted by then, or not learnt sufficiently, and only stopping when some predefined minimum error is reached might mean the algorithm never terminates, or that it overfits. Using both of these options together can help, as can terminating the learning once the error stops decreasing.

However, the validation set we just created gives us something rather more useful, since we can use it to monitor the generalisation ability of the network at its current stage of learning. If we plot the sum-of-squares error during

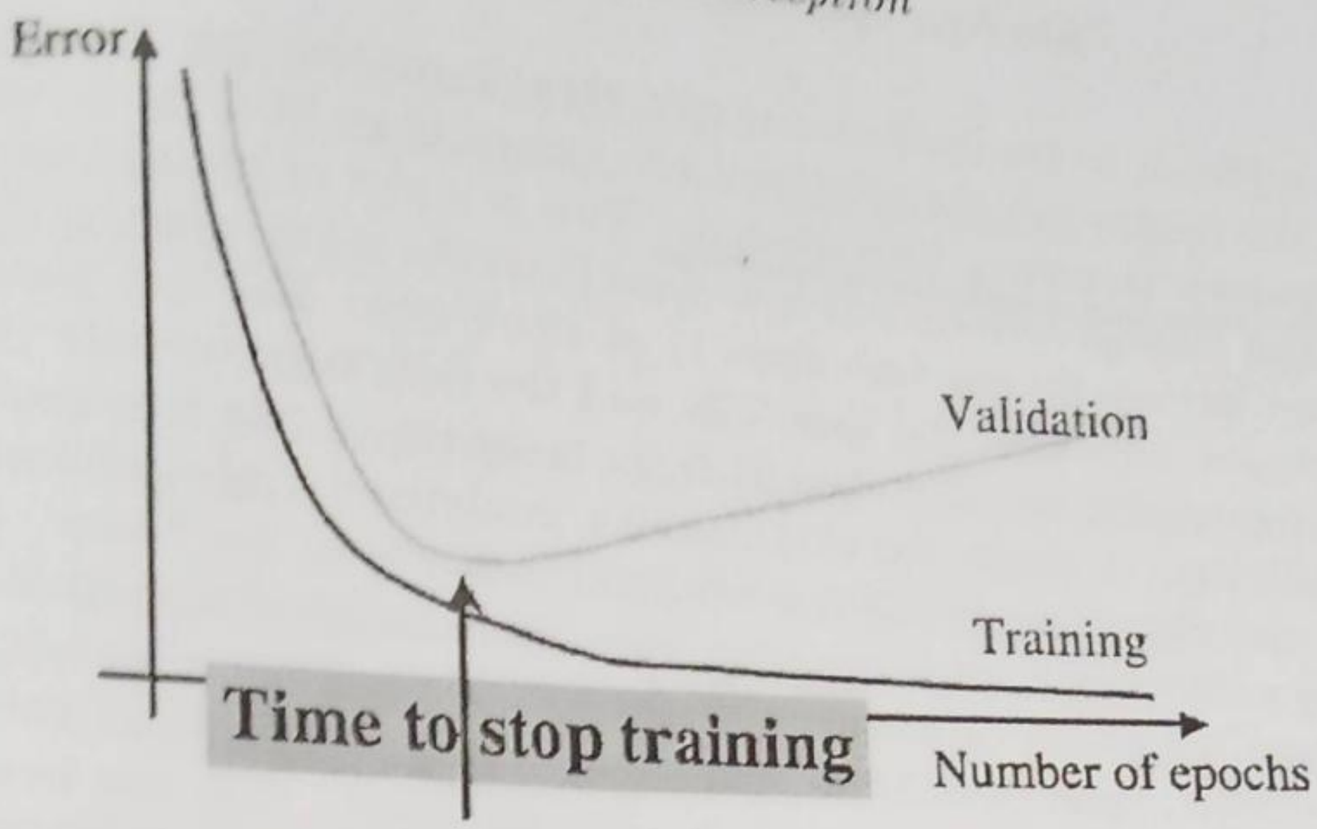


FIGURE 3.14: The effect of overfitting on the training and validation error curves, with the point at which early stopping will stop the learning marked.

training, it typically reduces fairly quickly during the first few training iterations, and then the reduction slows down as the learning algorithm performs small changes to find the exact local minimum. We don't want to stop training until the local minimum has been found, but, as we've just discussed, keeping on training too long leads to overfitting of the network. This is where the validation set comes in useful. We train the network for some predetermined amount of time, and then use the validation set to estimate how well the network is generalising. We then carry on training for a few more iterations, and repeat the whole process. At some stage the error on the validation set will start increasing again, because the network has stopped learning about the function that generated the data, and started to learn about the noise that is in the data itself (shown in Figure 3.14). At this stage we stop the training. This technique is called **early stopping**.

3.3.7 Computing and Evaluating the Results

We've now talked about everything that we need to run the algorithm and make it learn, but we haven't really considered how to report and analyse the results. This involves using the **test set** which has been held separate so far and running the network forward on this data. The error can then be evaluated by comparing the prediction with the target outputs. The question of what to do then depends upon the type of problem that we are solving. For a classification problem it is possible to compute the number of cases that the network predicted correctly for each class (in fact, we saw this in Section 2.2.6 with the **confusion matrix**), while for regression problems the only thing that is generally useful is the sum-of-squares error that we used to drive the training. We will see these methods being used as we look at examples.

In addition to the confusion matrix there is another way that we can evaluate the results of a classifier, which is known as an ROC curve, which stands for Receiver Operating Characteristic. This is a plot of the percentage of true positives (things correctly put into class 1) on the y axis against false positives (things incorrectly put into class 1) on the x axis. The true positive rate is sometimes known as the specificity and the false negative rate (things that were incorrectly put into class 2) is the sensitivity, so the false positive rate is 1-sensitivity. A single run of a classifier produces a single point on the ROC plot, and the closer to the top-left-hand corner it is, the better. If you were to use a fair coin to pick the class, then you would end up with a line on the diagonal from bottom-left to top-right.

The key to getting a curve rather than a point on the ROC curve is to use cross-validation. If you use 10-fold cross-validation, then you have 10 classifiers, with 10 different test sets, and you also have the true labels. The true labels can be used to produce a ranked list of the different cross-validation-trained results, which can be used to specify a curve through the 10 datapoints on the ROC curve that correspond to the results of this classifier. By producing an ROC curve for each classifier it is possible to compare their results.

3.4 Examples of Using the MLP

This section is intended to be practical, so you should follow the examples at a computer, and add to them as you wish. The MLP is rather too complicated to enable us to work through the weight changes as we did with the Perceptron. Instead, we shall look at some demonstrations of how to make the network learn about some data. As was mentioned above, we shall look at the four types of problems that are generally solved using an MLP: regression, classification, time-series prediction, and data compression/data denoising.

3.4.1 A Regression Problem

The regression problem we will look at is a very simple one. We will take a set of samples generated by a simple mathematical function, and try to learn the generating function (that describes how the data was made) so that we can find the values of any inputs, not just the ones we have training data for.

The function that we will use is a very simple one, just a bit of a sine wave. We'll make the data in the following way (the reason why the `x` computation requires the `ones()` call is because of some idiosyncracies in the way that *NumPy* produces arrays). Make sure that you have NumPy imported first:

```
x = ones((1,40))*linspace(0,1,40)
t = sin(2*pi*x) + cos(4*pi*x) + random.randn(40)*0.2
x = transpose(x)
t = transpose(t)
```

You can plot this data to see what it looks like (the results of which are shown in Figure 3.15) using:

```
>>> from pylab import *
>>> plot(x,t,'.')
```

We can now train an MLP on the data. There is one input value, `x` and one output value `t`, so the neural network will have one input and one output. Also, because we want the output to be the value of the function, rather than 0 or 1, we will use linear neurons at the output. We don't know how many hidden neurons we will need yet, so we'll have to experiment to see what works.

Before getting started, we need to normalise the data using the method shown in Section 3.3.1, and then separate the data into training, testing, and validation sets. For this example there are only 40 datapoints, and we'll use half of them as the training set, although that isn't very many and might not be enough for the algorithm to learn effectively. We can split the data in the ratio 50:25:25 by using the odd-numbered elements as training data, the even numbered ones that do not divide by 4 for testing, and the rest for validation:

```
train = x[0::2,:]
test = x[1::4,:]
valid = x[3::4,:]
traintarget = t[0::2,:]
testtarget = t[1::4,:]
validtarget = t[3::4,:]
```

With that done, it is just a case of making and training the MLP. To start with, we will construct a network with three nodes in the hidden layer, and run it for 101 iterations with a learning rate of 0.25, just to see that it works:

```
>>> import mlp
>>> net = mlp.mlp(train,traintarget,3,outtype='linear')
>>> net.mlptrain(train,traintarget,0.25,101)
```

11-5

The output from this will look something like:

```
Iteration:  0  Error:  12.3704163654
Iteration:  100  Error:  8.2075961385
```

so we can see that the network is learning, since the error is decreasing. We now need to do two things: work out how many hidden nodes we need, and decide how long to train the network for. To solve the first problem, we need to test out different networks and see which get lower errors, but to do that properly we need to know when to stop training. So we'll solve the second problem first, which is to implement early stopping.

We train the network for a few iterations (let's make it 10 for now). then evaluate the validation set error by running the network forward (i.e., the recall phase). Learning should stop when the validation set error starts to increase. We'll write a Python program that does all the work for us. The important point is that we keep track of the validation error and stop when it starts to increase. The following code is a function within the MLP on the book website. It keeps track of the last two changes in validation error to ensure that small fluctuations in the learning don't change it from early stopping to premature stopping:

```
old_val_error1 = 100002
old_val_error2 = 100001
new_val_error = 100000

count = 0
while (((old_val_error1 - new_val_error) > 0.001) or ((
old_val_error2 - old_val_error1)>0.001)):
    count+=1
    self.mlptrain(inputs,targets,0.25,100)
    old_val_error2 = old_val_error1
    old_val_error1 = new_val_error
    validout = self.mlpfwd(valid)
    new_val_error = 0.5*sum((validtargets-validout)**2)

print "Stopped", new_val_error,old_val_error1, old_val_error2
```

Figure 3.16 gives an example of the output of running the function. It plots the training and validation errors. The point at which early-stopping makes the learning finish is the point where there is a missing validation datapoint. I ran it on after that so you could see that the validation error did not improve after that, and so early-stopping found the correct point.

We can now return to the problem of finding the right size of network. There is one important thing to remember, which is that the weights are

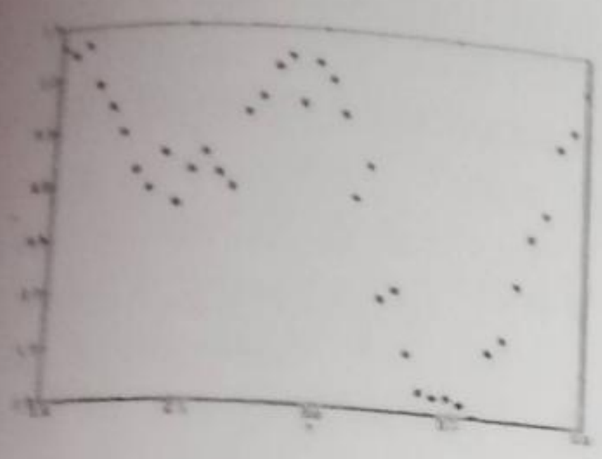


FIGURE 3.15: The data that we will learn using an MLP, consisting of some samples from a sine wave with Gaussian noise added.

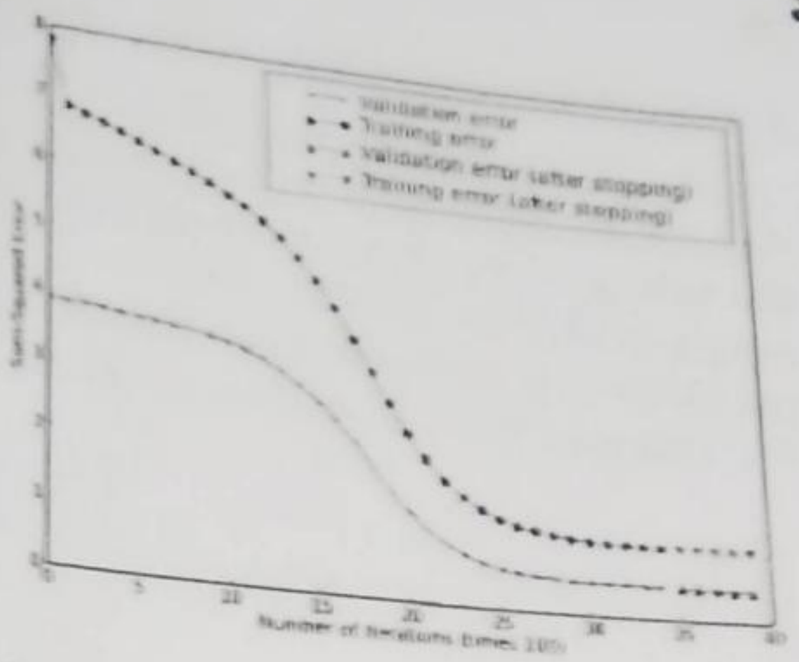


FIGURE 3.16: Plot of the error as the MLP learns (top line is total error on training set, bottom line is on validation set; the reason why it is larger on the training set is that there are more datapoints in this set). Early-stopping halts the learning at the point where there is no line, where the crosses become triangles. The learning was continued to show that the error got slightly worse afterwards.

initialised randomly, and so the fact that a particular size of network gets a good solution once does not mean it is the right size, it could have been a lucky starting point. So each network size is run 10 times, and the average is monitored. The following table shows the results of doing this, reporting the sum-of-squares validation error, for a few different sizes of network:

No. of hidden nodes	1	2	3	5	10	25	50
Mean error	2.21	0.52	0.52	0.52	0.55	1.35	2.56
Standard deviation	0.17	0.00	0.00	0.02	0.00	1.20	1.27
Max error	2.31	0.53	0.54	0.54	0.60	3.230	3.66
Min error	2.10	0.51	0.50	0.50	0.47	0.42	0.52

Based on these numbers, we would select a network with a small number of hidden nodes, certainly between 2 and 10 (and the smaller the better, in general), since their maximum error is much smaller than a network with just 1 hidden node. Note also that the error increases once too many hidden nodes are used, since the network has too much variation for the problem. You can also do the same kind of experimentation with more hidden layers.

II-5

3.4.2 Classification with the MLP

Using the MLP for classification problems is not radically different once the output encoding has been worked out. The inputs are easy: they are just the values of the feature measurements (suitably normalised). There are a couple of choices for the outputs. The first is to use a single linear node for the output, y, and put some thresholds on the activation value of that node. For example, for a four-class problem, we could use:

$$\text{Class is: } \begin{cases} C_1 \text{ if } y \leq -0.5 \\ C_2 \text{ if } -0.5 < y \leq 0 \\ C_3 \text{ if } 0 < y \leq 0.5 \\ C_4 \text{ if } y > 0.5 \end{cases} \tag{3.16}$$

However, this gets impractical as the number of classes gets large, and the boundaries are artificial: what about an example that is very close to a boundary, say $y = 0.5$? We arbitrarily guess that it belongs to class C_3, but the neural network doesn't give us any information about how close it was to the boundary in the output, so we don't know that this was a difficult example to classify. A more suitable output encoding is called 1-of-N encoding. A separate node is used to represent each possible class, and the target vectors consist of zeros everywhere except for in the one element that corresponds to the correct class, e.g., $(0, 0, 0, 1, 0, 0)$ means that the correct result is the 4th class out of 6. We are therefore using binary output values (we want each output to be either 0 or 1).

Once the network has been trained, performing the classification is easy: simply choose the element y_k of the output vector that is the largest element of $\mathbf{y}$ (in mathematical notation, pick the y_k for which $y_k > y_j \forall j \neq k$; $\forall$ means for all, so this statement says pick the y_k that is bigger than all other possible values y_j). This generates an unambiguous decision, since it is very unlikely that two output neurons will have identical largest output values. This is known as the hard-max activation function (since the neuron with the highest activation is chosen to fire and the rest are ignored). An alternative is the soft-max function, which we saw in Section 3.2.3, and which has the effect of scaling the output of each neuron according to how large it is in comparison to the others, and making the total output sum to 1. So if there is one clear winner, it will have a value near 1, while if there are several values that are close to each other, they will each have a value of about $\frac{1}{p}$, where p is the number of output neurons that have similar values.

There is one other thing that we need to be aware of when performing classification, which is true for all classifiers. Suppose that we are doing two class classification, and 90% of our data belongs to class 1. (This can happen: for example in medical data, most tests are negative in general.) In that case, the algorithm can learn to always return the negative class, since it will be right 90% of the time, but still a completely useless classifier! So you should generally make sure that you have approximately the same number

```
order = range(shape(iris)[0])
random.shuffle(order)
iris = iris[order,:]
target = target[order,:]

train = iris[::2,0:4]
traint = target[::2]
valid = iris[1::4,0:4]
validt = target[1::4]
test = iris[3::4,0:4]
testt = target[3::4]
```

We're now finally ready to set up and train the network. The commands should all be familiar from earlier:

```
>>> import mlp
>>> net = mlp.mlp(train,traint,5,outtype='softmax')
>>> net.earlystopping(train,traint,valid,validt,0.1)
>>> net.confmat(test,testt)
Confusion matrix is:
[[ 16.   0.   0.]
 [  0.  12.   2.]
 [  0.   1.   6.]]
Percentage Correct:  91.8918918919
```

This tells us that the algorithm got nearly all of the test data correct, misclassifying just two examples of class 2 and one of class 3.

3.4.4 Time-Series Prediction

There is a common data analysis task known as time-series prediction, where we have a set of data that show how something varies over time, and we want to predict how the data will vary in the future. It is quite a difficult task, but a fairly important one. It is useful in any field where there is data that appears over time, which is to say almost any field. Most notable (if often unsuccessful) uses have been in trying to predict stock markets and disease patterns. The problem is that even if there is some regularity in the time-series, it can appear over many different scales. For example, there is often seasonal variation—if we plotted average temperature over several years, we would notice that it got hotter in the summer and colder in the winter, but we might not notice if there was a overall upward or downward trend to the summer temperatures, because the summer peaks are spread too far apart in the data.

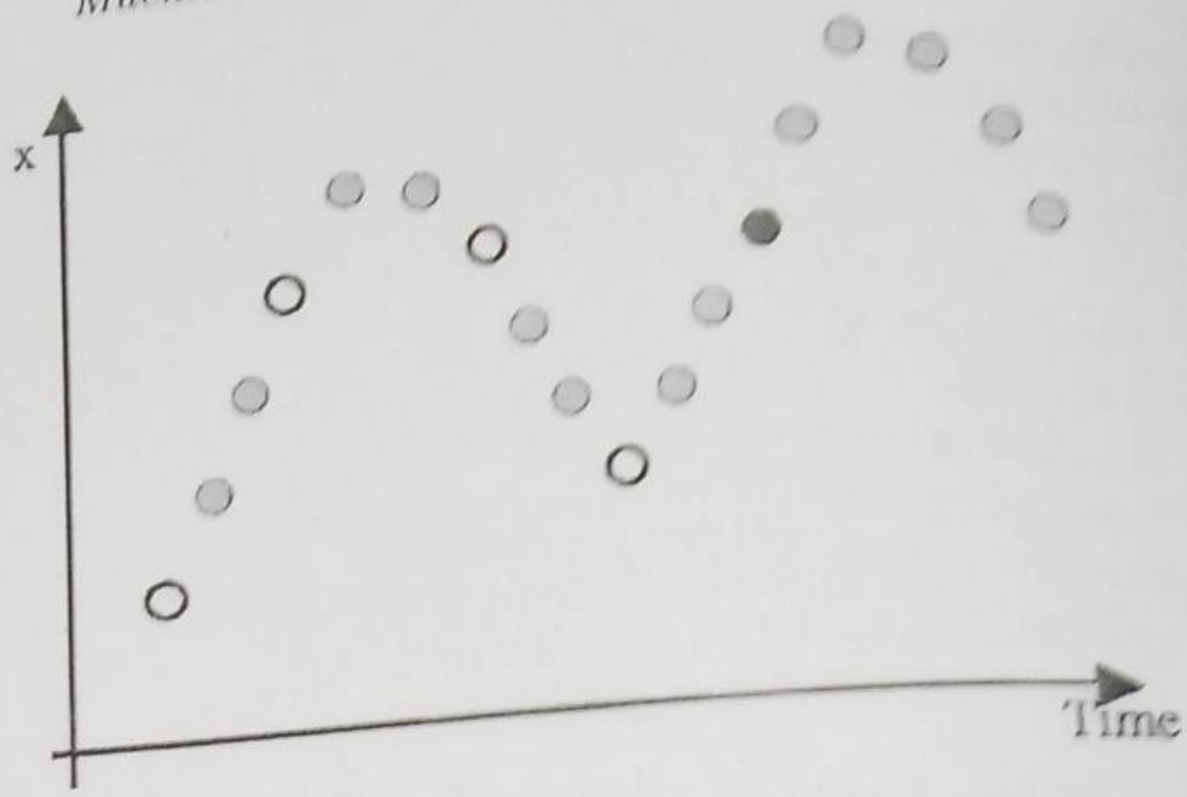


FIGURE 3.17: Part of a time-series plot, showing the datapoints and the meanings of τ and k.

The other problems with the data are practical. How many datapoints should we look at to make the prediction (i.e., how many inputs should there be to the neural network) and how far apart in time should we space those inputs (i.e., should we use every second datapoint, every 10th, or all of them)? We can write this as an equation, where we are predicting y using a neural network that is written as a function $f(\cdot)$:

$$y = x(t+\tau) = f(x(t), x(t-\tau), \ldots, x(t-k\tau)), \tag{3.17}$$

where the two questions about how many datapoints and how far apart they should be come down to choices about τ and k.

The target data for training the neural network is simple, because it comes from further up the time-series, and so training is easy. Suppose that $\tau = 2$ and $k = 3$. Then the first input data are elements $1, 3, 5$ of the dataset, and the target is element 7. The next input vector is elements $2, 4, 6$, with target 8, and then $3, 5, 7$ with target 9. You train the network by passing through the time-series (remembering to save some data for testing), and then press on into the future making predictions. Figure 3.17 shows an example of a time-series with $\tau = 3$ and $k = 4$, with a set of datapoints that make up an input vector marked as white circles, and the target coloured black.

The dataset I am going to use is available on the book website. It provides the daily measurement of the thickness of the ozone layer above Palmerston North in New Zealand (where I live) between 1996 and 2004. Ozone thickness is measured in Dobson Units, which are 0.01 mm thickness at 0 degrees Celcius and 1 atmosphere of pressure. I'm sure that I don't need to tell you that the reduction in stratospheric ozone is partly responsible for global warming and the increased incidence of skin cancer, and that in New Zealand we are fairly

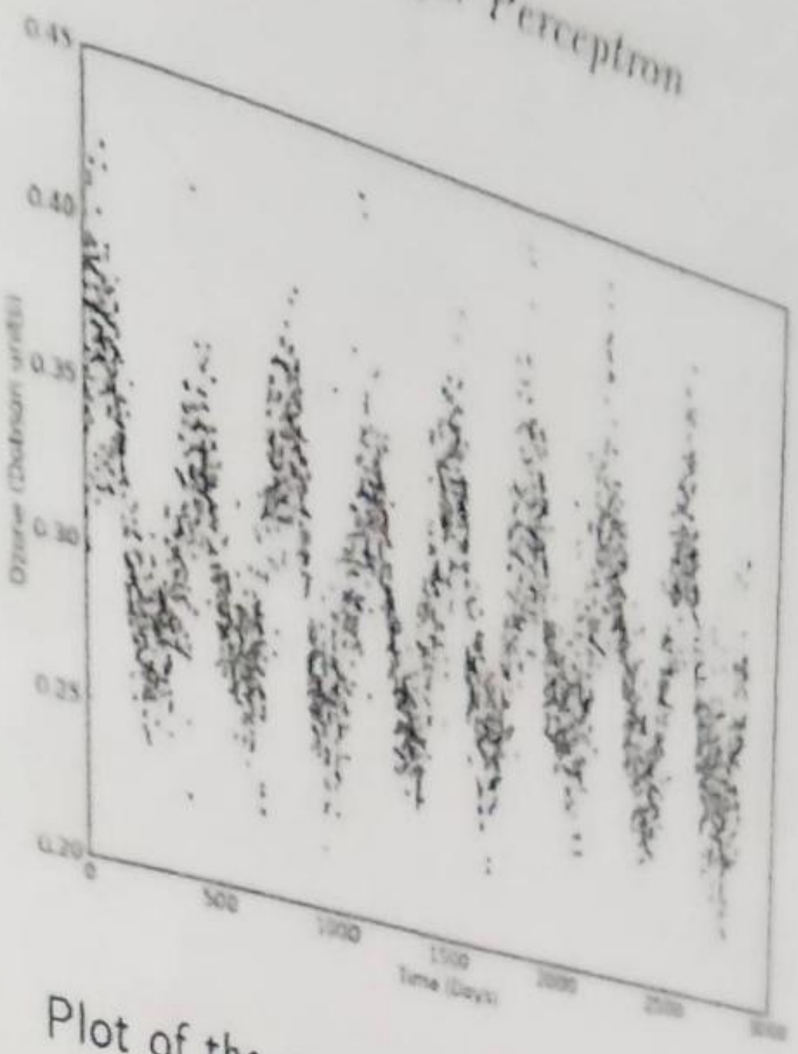


FIGURE 3.18: Plot of the ozone layer thickness above Palmerston North in New Zealand between 1996 and 2004.

close to the large hole over Antarctica. What you might not know is that the thickness of the ozone layer varies naturally over the year. This should be obvious in the plot shown in Figure 3.18. A typical time-series problem is to predict the ozone levels into the future and see if you can detect an overall drop in the mean ozone level.

You can load the data using `PNoz = loadtxt('PNOz.dat')` (once you've downloaded it from the website), which will load the data and stick it into an array called PNoz. There are 4 elements to each vector: the year, the day of the year, and the ozone level and sulphur dioxide level, and there are 2855 readings. To just plot the ozone data so that you can see what it looks like, use `plot(arange(shape(PNoz)[0]),PNoz[:,2],'.')`.

The difficult bit is assembling the input vector from the time-series data. The first thing is to choose values of τ and k. Then it is just a question of picking k values out of the array with spacing τ, which is a good use for the slice operator, as in this code:

```
test = inputs[-800:,:]
testtargets = targets[-800,:]
train = inputs[:-800:2,:]
traintargets = targets[:-800:2]
valid = inputs[1:-800:2,:]
validtargets = targets[1:-800:2]
```

You then need to assemble training, testing, and validation sets. However, some care is needed here since you need to ensure that they are not picked systematically into each group, (for example, if the inputs are the even-indexed datapoints, but some feature is only seen at odd datapoint times, then it will be completely missed). This can be averted by randomising the order of the

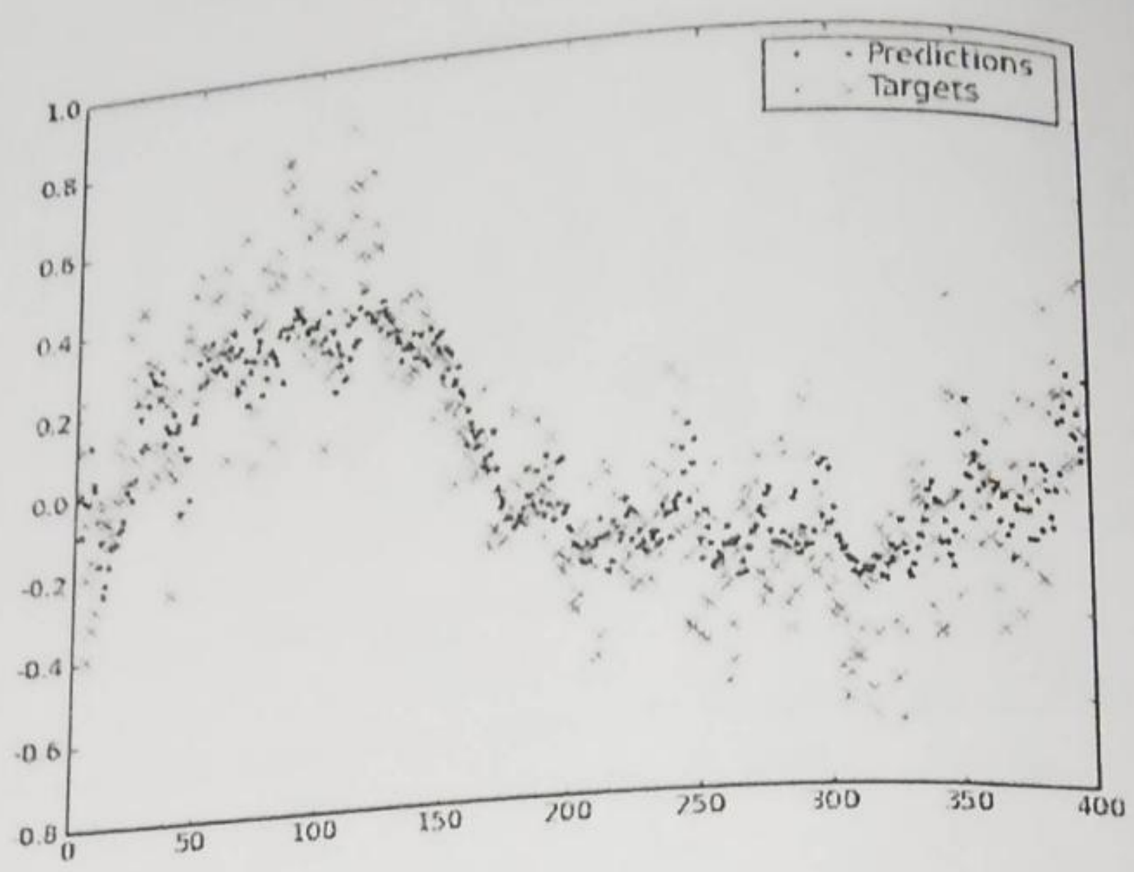


FIGURE 3.19: Plot of 400 predicted and actual output values of the ozone data using the MLP as a time-series predictor with $k = 3$ and $\tau = 2$.

datapoints first. However, it is also common to use the datapoints near the end as part of the test set; some possible results from using the MLP in this way are shown in Figure 3.19.

From here you can treat time-series as regression problems: the output nodes need to have linear activations, and you aim to minimise the sum-of-squares error, since there are no classes the confusion matrix is not useful. The only extra work is that in addition to testing MLPs with different numbers of input nodes and hidden nodes, you also need to consider different values of τ and k.

3.4.5 Data Compression: The Auto-Associative Network

We are now going to consider an interesting variation of the MLP. Suppose that we train the network to reproduce the inputs at the output layer (called auto-associative learning; sometimes the network is known as an autoencoder). The network is trained so that whatever you show it at the input is reproduced at the output, which doesn't seem very useful at first, but suppose that we use a hidden layer that has fewer neurons than the input layer (see Figure 3.20). This bottleneck hidden layer has to represent all of the information in the input, so that it can be reproduced at the output. It therefore performs some compression of the data, representing it using fewer dimensions than were used in the input. This gives us some idea of what the hidden layers of the MLP are doing: they are finding a different (often lower dimensional) representation of the input data that extracts important components of the data, and ignores

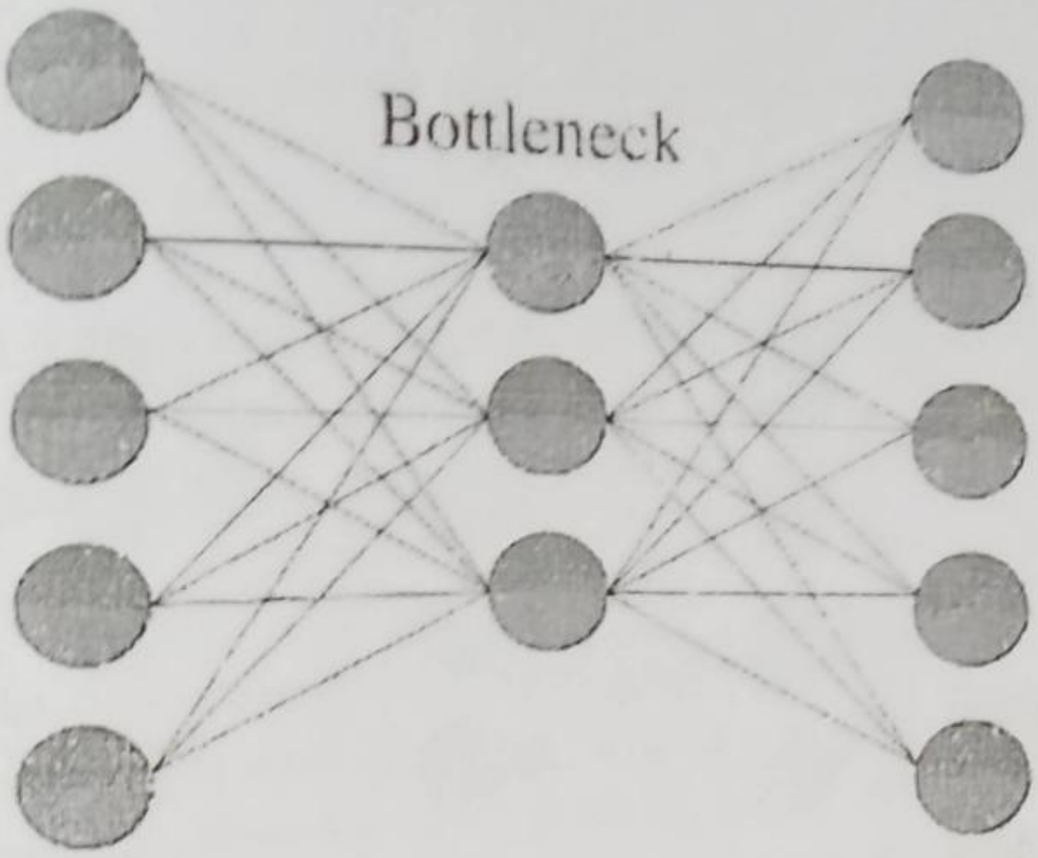


FIGURE 3.20: The auto-associative network. The network is trained to reproduce the inputs at the outputs, passing them through the bottleneck hidden layer that compresses the data.

ıe noise.

This auto-associative network can be used to compress images and other ata. A schematic of this is shown in Figure 3.21: the 2D image is turned ıto a 1D vector of inputs by cutting the image into strips and sticking the rips into a long line. The values of this vector are the intensity (colour) lues of the image, and these are the input values. The network learns to eproduce the same image at the output, and the activations of the hidden odes are recorded for each image. After training, we can throw away the ıput nodes and first set of weights of the network. If we insert some values the hidden nodes (their activations for a particular image: see Figure 3.22). ıen by feeding these activations forward through the second set of weights. ıe correct image will be reproduced on the output. So all we need to store re the set of second-layer weights and the activations of the hidden nodes for ıch image, which is the compressed version.

Auto-associative networks can also be used to denoise images, since after raining the network will reproduce the trained image that best matches the urrent (noisy) input. We don't throw away the first set of weights this time. it if we feed in a noisy version of the image into the inputs, then the network ill produce the image that is closest to the noisy version at the outputs, which ill be the version it learnt on, which is uncorrupted by noise.

You might be wondering what this representation in the hidden nodes looks ke. In fact, what the network learns to compute are the Principal Components f the input data. Principal Component Analysis (PCA) is a useful dimensionality eduction technique, and is described in Section 10.2.

II-64

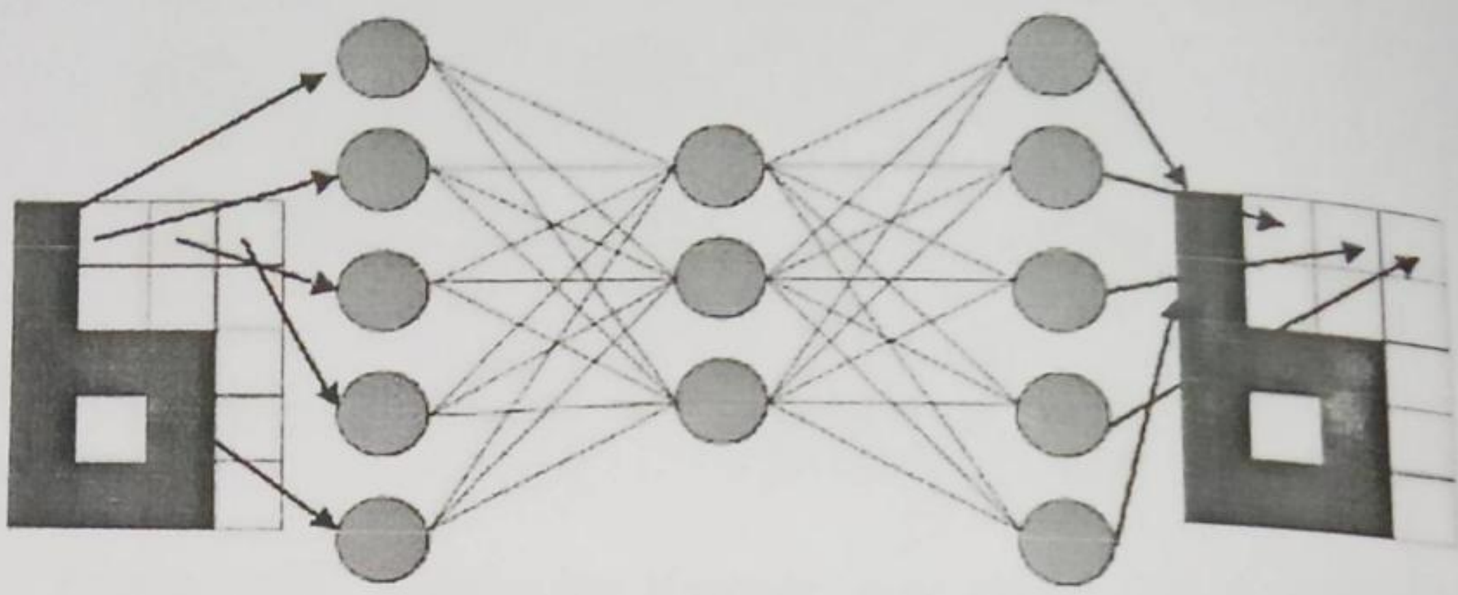


FIGURE 3.21: Schematic showing how images are fed into the auto-associative network for compression.

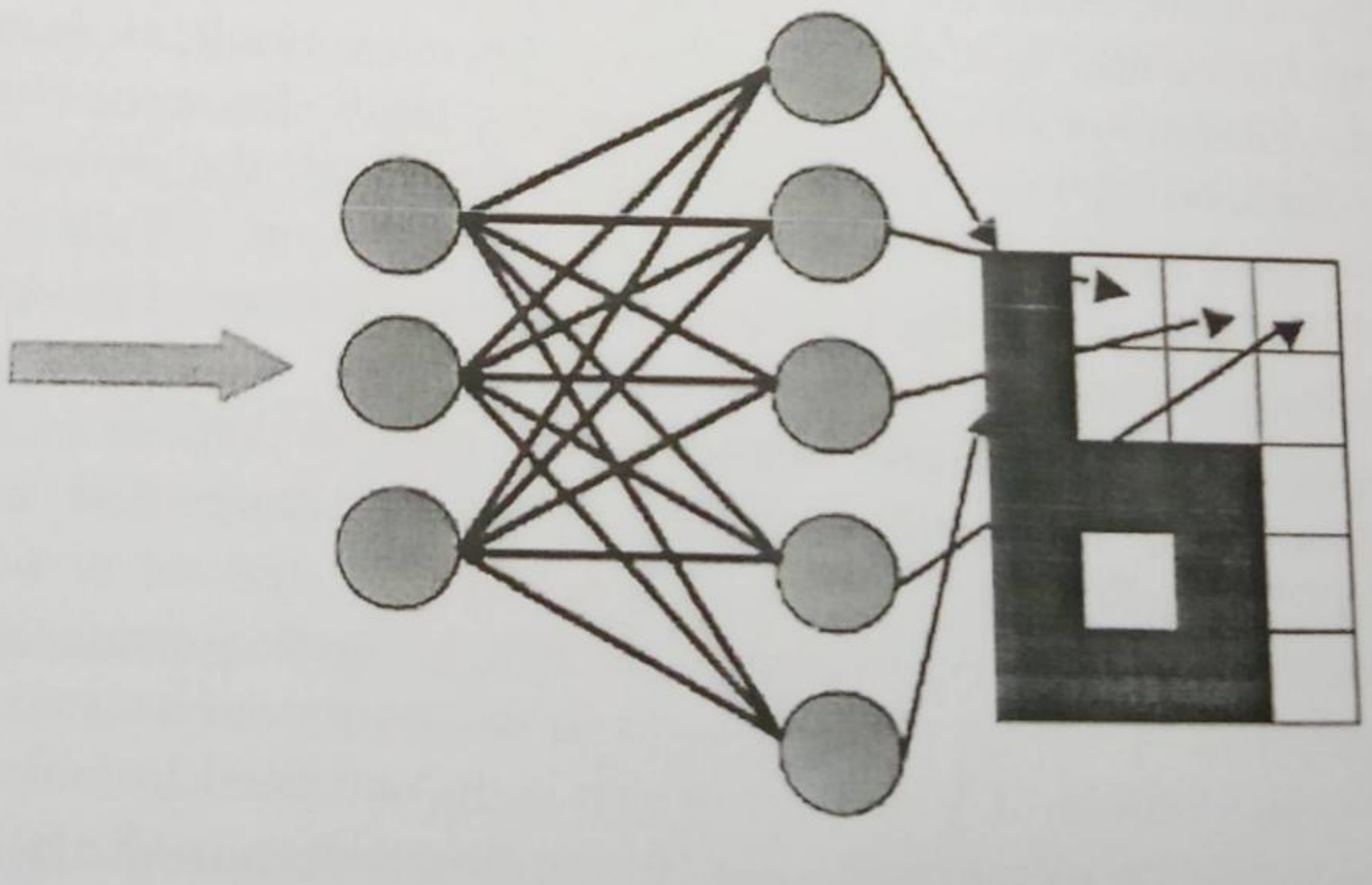


FIGURE 3.22: Schematic showing how the hidden nodes and second layer of weights can be used to regain the compressed images after the network has been trained.

3.5 Overview

We have covered a lot in this chapter, so I'm going to give you a 'recipe' for how to use the Multi-Layer Perceptron in practice. This is, by necessity, a simplification of the problem, but it should serve to remind you of many of the important features.

Select inputs and outputs for your problem Before anything else, you need to think about the problem you are trying to solve, and make sure that you have data for the problem, both input vectors and target outputs. At this stage you need to choose what features are suitable for the problem (something we'll talk about more in other chapters) and decide on the output encoding that you will use — standard neurons, or linear nodes. These things are often decided for you by the input features and targets that you have available to solve the problem. Later on in the learning it can also be useful to re-evaluate the choice by training networks with some input feature missing to see if it improves the results at all.

Normalise inputs Rescale the data by subtracting the mean value from each element of the input vector, and divide by the variance (or alternatively, either the maximum or minus the minimum, whichever is greater).

Split the data into training, testing, and validation sets You cannot test the learning ability of the network on the same data that you trained it on, since it will generally fit that data very well (often too well, over-fitting and modelling the noise in the data as well as the generating function). We generally split the data into three sets, one for training, one for testing, and then a third set for validation, which is testing how well the network is learning during training. The ratio between the sizes of the three groups depends on how much data you have, but is often around 50:25:25.

Where there is not enough data for three sets, a technique called cross-validation can be useful. In its most extreme form, leave-one-out cross-validation, this consists of training the network on all but one piece of the training data and then validating it on the final piece. You then train another network on the training data again, but leaving out a different piece of data. You select one of the networks that gets the final piece correct. You still need a separate test set.

Select a network architecture You already know how many input nodes there will be, and how many output neurons. You need to consider whether you will need a hidden layer at all, and if so how many neurons

it should have in it. You might want to consider more than one hidden layer. The more complex the network, the more data it will need to be trained on, and the longer it will take. It might also be more subject to overfitting. The usual method of selecting a network architecture is to try several with different numbers of hidden nodes and see which works best.

Train a network The training of the neural network consists of applying the multi-layer Perceptron algorithm to the training data. This is usually run in conjunction with early-stopping, where after a few iterations of the algorithm through all of the training data, the generalisation ability of the network is tested by using the validation set. The neural network is very likely to have far too many degrees of freedom for the problem, and so after some amount of learning it will stop modelling the generating function of the data, and start to fit the noise and inaccuracies inherent in the training data. At this stage the error on the validation set will start to increase, and learning should be stopped.

Test the network Once you have a trained network that you are happy with, it is time to use the test data for the first (and only) time. This will enable you to see how well the network performs on some data that it has not seen before, and will tell you whether this network is likely to be usable for other data, for which you do not have targets.

3.6 Deriving Back-Propagation

This section derives the back-propagation algorithm. This is important to understand how and why the algorithm works. There isn't actually that much mathematics involved except some slightly messy algebra. In fact, there are only three things that you really need to know. One is the derivative (with respect to x) of $\frac{1}{2}x^2$, which is x, and another is the chain rule, which says that $\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx}$. The third thing is very simple: $\frac{dy}{dx} = 0$ if y is not a function of x. With those three things clear in your mind, just follow through the algebra, and you'll be fine. We'll work in simple steps.

3.6.1 The Network Output and the Error

The output of the neural network (the end of the forward phase of the algorithm) is a function of three things:

- the current input ($\mathbf{x}$)
- the activation function $g(\cdot)$ of the nodes of the network

- the weights of the network (v for the first layer and w for the second)

We can't change the inputs, since they are what we are learning about, nor can we change the activation function as the algorithm learns. So the weights are the only things that we can vary to improve the performance of the network, i.e., to make it learn. However, we do need to think about the activation function, since the threshold function that we used for the Perceptron is not differentiable (it has a discontinuity at 0). We'll think about a better one in Section 3.6.3, but first we'll think about the error of the network. Remember that we have run the algorithm forwards, so that we have fed the inputs ($\mathbf{x}$) into the algorithm, used the first set of weights ($\mathbf{v}$) to compute the activations of the hidden neurons, then those activations and the second set of weights ($\mathbf{w}$) to compute the activations of the output neurons, which are the outputs of the network ($\mathbf{y}$). Note that I'm going to use i to be an index over the input nodes, j to be an index over the hidden layer neurons, and k to be an index over the output neurons.

3.6.2 The Error of the Network

When we discussed the Perceptron learning rule in the previous chapter we motivated it by minimising the error function $E = \sum_{i=1}^{N} \mathbf{t}_i - \mathbf{y}_i$. We then invented a learning rule that made this error smaller. We are going to do much better this time, because everything is computed from the principles of gradient descent.

To begin with, let's think about the error of the network. This is obviously going to have something to do with the difference between the targets $\mathbf{t}$ and the outputs $\mathbf{y}$, but I'm going to write it as $E(\mathbf{v}, \mathbf{w})$ to remind us that the only things that we can change are the weights $\mathbf{v}$ and $\mathbf{w}$, and that changing the weights changes the output, which in turn changes the error. For the Perceptron we computed the error as $E = \sum_{i=1}^{N} \mathbf{t}_i - \mathbf{y}_i$, but there are some problems with this: if $\mathbf{t}_i > \mathbf{y}_i$, the sign of the error is different to if $\mathbf{y}_i > \mathbf{t}_i$, so if we have lots of output nodes that are all wrong, but some have positive sign and some have negative sign, then they might cancel out. Instead, we'll choose the sum-of-squares error function, which calculates the difference between $\mathbf{t}_i$ and $\mathbf{y}_i$ for each node, squares them, and adds them together (I've missed out the $\mathbf{v}$ in $E(\mathbf{w})$ because we don't use them here):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^{N} (t_k - y_k)^2 \tag{3.18}$$

$$= \frac{1}{2} \sum_{k} \left[t_k - g\left(\sum_{j} w_{jk} a_j \right) \right]^2 \tag{3.19}$$

The second line adds in the input from the hidden layer neurons and the

a_j is output - which

second-layer weights to decide on the activations of the output neurons. For now we're going to think about the Perceptron, so Equation (3.19) will be replaced by:

$$\frac{1}{2}\sum_k \left[t_k - g\left(\sum_j w_{jk} x_j \right) \right]^2 \tag{3.20}$$

Now we can't differentiate the threshold function, which is what the Perceptron used for $g(\cdot)$, because it has a discontinuity (sudden jump) at the threshold value. So I'm going to miss it out completely for the moment. Also, for the Perceptron there are no hidden neurons, and so the activation of an output neuron is just $y_k = \sum_j w_{jk} x_j$ where x_j is the value of an input node.

We are going to use a gradient descent algorithm that adjusts each weight w_{jk} in the direction of the gradient of $E(\mathbf{w})$. In what follows, the notation ∂ means the partial derivative, and is used because there are lots of different functions that we can differentiate E with respect to all of the different weights. If you don't know what a partial derivative is, think of it as being the same as a normal derivative, but taking care that you differentiate in the correct direction. The gradient that we want to know is how the error function changes with respect to the different weights:

$$\frac{\partial E}{\partial w_{ik}} = \frac{\partial}{\partial w_{ik}} \left(\frac{1}{2} \sum_k (t_k - y_k)^2 \right) \tag{3.21}$$

$$= \frac{1}{2} \sum_k 2(t_k - y_k) \frac{\partial}{\partial w_{ik}} \left(t_k - \sum_j w_{jk} x_j \right) \tag{3.22}$$

t_k is not a function of w_{ik}, so $\frac{\partial t_k}{\partial w_{ik}} = 0$,

and the only part of $\sum_j w_{jk} x_j$ that is a function of w_{ik} is when $i = j$, that is, w_{ik} itself. Hence:

$$\frac{\partial E}{\partial w_{ik}} = \sum_k (t_k - y_k)(-x_i). \tag{3.23}$$

Now the idea of the weight update rule is that we follow the gradient downhill, that is, in the direction $-\frac{\partial E}{\partial w_{ik}}$. So the weight update rule (when we include the learning rate η) is:

$$w_{ik} \leftarrow w_{ik} + \eta (t_k - y_k) x_i. \tag{3.24}$$

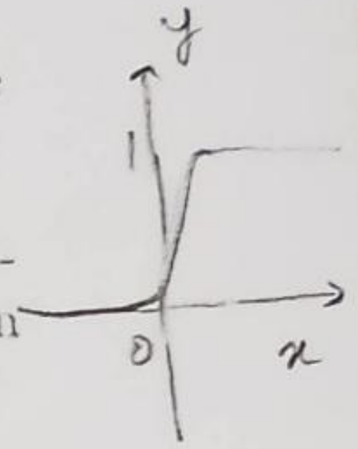
which hopefully looks familiar (see Equation (2.1)). It isn't actually identical, because we are computing y_k differently, and for the Perceptron, we used the

threshold activation threshold function, whereas in the work above we ignored the thresholder. This isn't very useful if we want units that act like neurons, because neurons either fire or do not fire, rather than varying continuously. However, if we want to be able to differentiate the output in order to use gradient descent, then we need a differentiable activation function, so that's what we'll talk about now.

3.6.3 A Suitable Activation Function

We want an activation function that has the following properties:

- it must be differentiable so that we can compute the gradient
- it should saturate (become constant) at both ends of the range, so that the neuron either fires or does not fire
- it should change between the saturation values fairly quickly in the middle



There is a family of functions called **sigmoid functions** because they are S-shaped (see Figure 3.5) that satisfy all those criteria perfectly. The form in which it is generally used is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}, \tag{3.25}$$

where β is some positive parameter. One happy feature of this function is that its derivative has an especially nice form:

$$g'(h) = \frac{dg}{dh} = \frac{d}{dh}(1 + e^{-\beta h})^{-1} \tag{3.26}$$

$$= -1(1 + e^{-\beta h})^{-2}\frac{de^{-\beta h}}{dh} \tag{3.27}$$

$$= -1(1 + e^{-\beta h})^{-2}(-\beta e^{-\beta h}) \tag{3.28}$$

$$= \frac{\beta e^{-\beta h}}{(1 + e^{-\beta h})^2} \tag{3.29}$$

$$= \beta g(h)(1 - g(h)) \tag{3.30}$$

$$\boxed{g'(h) = \beta a(1 - a)} \tag{3.31}$$

We'll be using this derivative later, except that we can ignore the factor of β, since this is just a scaling. So we've now got an error function and an activation function that we can compute derivatives of. The next things to do is work out how to use them in order to adjust the weights of the network.

3.6.4 Back-Propagation of Error

It is now that we'll need the chain rule that I reminded you of earlier. In the form that we want, it looks like this:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial h_k}\frac{\partial h_k}{\partial w_{jk}}, \tag{3.32}$$

where $h_k = \sum_l w_{lk}a_l$ is the input to output-layer neuron k, that is, the sum of the activations of the hidden-layer neurons multiplied by the relevant (second-layer) weights. So what does Equation (3.32) say? It tells us that if we want to know how the error at the output changes as we vary the second-layer weights, we can think about how the error changes as we vary the input to the output neurons, and also about how those input values change as we vary the weights.

Let's think about the second term first (in the third line we use the fact that $\frac{\partial w_{lk}}{\partial w_{jk}} = 0$ for all values of l except $l = j$):

$$\frac{\partial h_k}{\partial w_{jk}} = \frac{\partial \sum_l w_{lk}a_l}{\partial w_{jk}} \tag{3.33}$$

$$= \sum_l \frac{\partial w_{lk}a_l}{\partial w_{jk}} \tag{3.34}$$

$$= a_j. \tag{3.35}$$

Now we can worry about the $\frac{\partial E}{\partial h_k}$ term. This term is important enough to get its own term, which is the error or delta term:

$$\delta_o = \frac{\partial E}{\partial h_k}. \tag{3.36}$$

Let's start off by trying to compute this error for the output. We can't actually compute it directly, since we don't know much about the inputs to a neuron, we just know about its output. That's fine, because we can use the chain rule again:

$$\delta_o = \frac{\partial E}{\partial h_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial h_k}. \tag{3.37}$$

Now the output of output layer neuron k is

$$y_k = g(h_k^{\text{output}}) = g\left(\sum_j w_{jk}a_j^{\text{hidden}}\right), \tag{3.38}$$

where $g(\cdot)$ is the activation function. We've chosen to use the sigmoid function given in Equation (3.25), but for now I'm going to leave it as a function to make it a little bit more general. I've also started labelling whether h refers

to an output or hidden layer neuron, just to avoid any possible confusion. We don't need to worry about this for the activations, because we use y for the activations of output neurons and a for hidden neurons. In Equation (3.41) I've substituted in the expression for the error at the output, which we computed in Equation (3.19):

$$\delta_o = \frac{\partial E}{\partial g\left(h_k^{\text{output}}\right)} \frac{\partial g\left(h_k^{\text{output}}\right)}{\partial h_k^{\text{output}}} \tag{3.39}$$

$$= \frac{\partial E}{\partial g\left(h_k^{\text{output}}\right)} g'\left(h_k^{\text{output}}\right) \tag{3.40}$$

$$= \frac{\partial}{\partial g\left(h_k^{\text{output}}\right)} \left[\frac{1}{2}\sum_k \left(g(h^{\text{output}}) - t_k\right)^2\right] g'\left(h_k^{\text{output}}\right) \tag{3.41}$$

$$= \left(g(h_k^{\text{output}}) - t_k\right) g'(h_k^{\text{output}}) \tag{3.42}$$

$$\delta_o = (y_k - t_k) g'(h_k^{\text{output}}), \tag{3.43}$$

where $g'(h_k)$ denotes the derivative of g with respect to h_k. We know exactly what that is for the sigmoid functions that we are using: we computed it in Equation (3.31). So we can put everything together to compute the precise update rule for the second-layer weights, $w_{jk} \leftarrow w_{jk} - \eta \frac{\partial E}{\partial w_{jk}}$ (we are using the minus sign because we want to go downhill), where:

$$\frac{\partial E}{\partial w_{jk}} = \delta_o a_j \tag{3.44}$$

$$= (y_k - t_k) y_k (1 - y_k) a_j. \tag{3.45}$$

Having got through all this, we don't actually need to do too much more work to get to the first layer of v_{jk} weights, which connect the inputs to the hidden nodes. We need the chain rule (Equation (3.32)) one more time to get to these weights, remembering that we are working backwards through the network:

$$\delta_h = \sum_k \frac{\partial E}{\partial h_k^{\text{output}}} \frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}} \tag{3.46}$$

$$= \sum_k \delta_o \frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}}. \tag{3.47}$$

In the first line, k runs over the output nodes, and we obtain the second line by using Equation (3.40). We now need a nicer expression for that derivative.

The important thing that we need to remember is that inputs to the output layer neurons come from the activations of the hidden layer neurons multiplied by the second layer weights:

$$h_k^{\text{output}} = g\left(\sum_l w_{lk} h_l^{\text{hidden}}\right), \tag{3.48}$$

which means that:

$$\frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}} = \frac{\partial g\left(\sum_l w_{lk} h_l^{\text{hidden}}\right)}{\partial h_j^{\text{hidden}}}. \tag{3.49}$$

We can now use a fact that we've used before, which is that $\frac{\partial h_l}{\partial h_j} = 0$ unless $l = j$. So:

$$\frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}} = w_{jk} g'(a_j) \tag{3.50}$$

$$= w_{jk} a_j (1 - a_j) \tag{3.51}$$

which allows us to compute:

$$\delta_h = a_j(1 - a_j) \sum_k \delta_o w_{jk} \tag{3.52}$$

and so get to the update rule for $v_{ij} \leftarrow v_{ij} - \eta \frac{\partial E}{\partial v_{ij}}$, by computing:

$$\frac{\partial E}{\partial v_{ij}} = a_j(1 - a_j) \left(\sum_k \delta_o w_{jk}\right) x_i. \tag{3.53}$$

Note that we can do exactly the same computations if the network has extra hidden layers between the inputs and the outputs. It gets harder to keep track of which functions we should be differentiating, but there are no new tricks needed.

Further Reading

The original papers describing the back-propagation algorithm are listed here, along with a well-known introduction to neural networks:

- D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323(99):533–536. 1986a.

UNIT-II

Chapter 5

PART-III

Support Vector Machines

Back in Chapter 2 we looked at the Perceptron, a set of McCulloch and Pitts neurons arranged in a single layer. We identified a method by which we could modify the weights so that the network learned, and then saw that the Perceptron was rather limited in that it could only identify straight line classifiers, that is, it could only separate out groups of data if it was possible to draw a straight line (hyperplane in higher dimensions) between them. This meant that it could not learn the difference between the two truth classes of the 2D XOR function. However, in Section 2.3.2, we saw that it was possible to modify the problem so that the Perceptron could solve the problem, by changing the data so that it used more dimensions than the original data.

This chapter is concerned with a method that makes use of that insight, amongst other things. The main idea is one that we have seen before, in Section 4.4, which is to modify the data by changing its representation. However, the terminology is different here, and we will introduce kernel functions rather than bases. In principle, it is always possible to transform any set of data so that the classes within it can be separated linearly. To get a bit of a handle on this, think again about what we did with the XOR problem in Section 2.3.2: we added in an extra dimension and moved a point that we could not classify properly into that additional dimension so that we could linearly separate the classes. The problem is how to work out which dimensions to use, and that is what kernel methods, which is the class of algorithms that we will talk about in this chapter, do.

We will focus on one particular algorithm, the Support Vector Machine (SVM), which is one of the most popular algorithms in modern machine learning. They were introduced by Vapnik in 1992 and have taken off radically since then, principally because they often (but not always) provide significantly better classification performance than other machine learning algorithms on reasonably sized datasets (they do not work well on extremely large datasets, since they involve a data matrix inversion, which is computationally very expensive). This should be sufficient motivation to master the (quite complex) concepts that are needed to understand the algorithm. We won't be using any code in this chapter, since implementing an SVM is probably something you wouldn't want to implement for yourself: some of the details of the algorithm, particularly the optimisation routine, are difficult to implement. There are several different implementations of the SVM available on the Internet,

II - 74

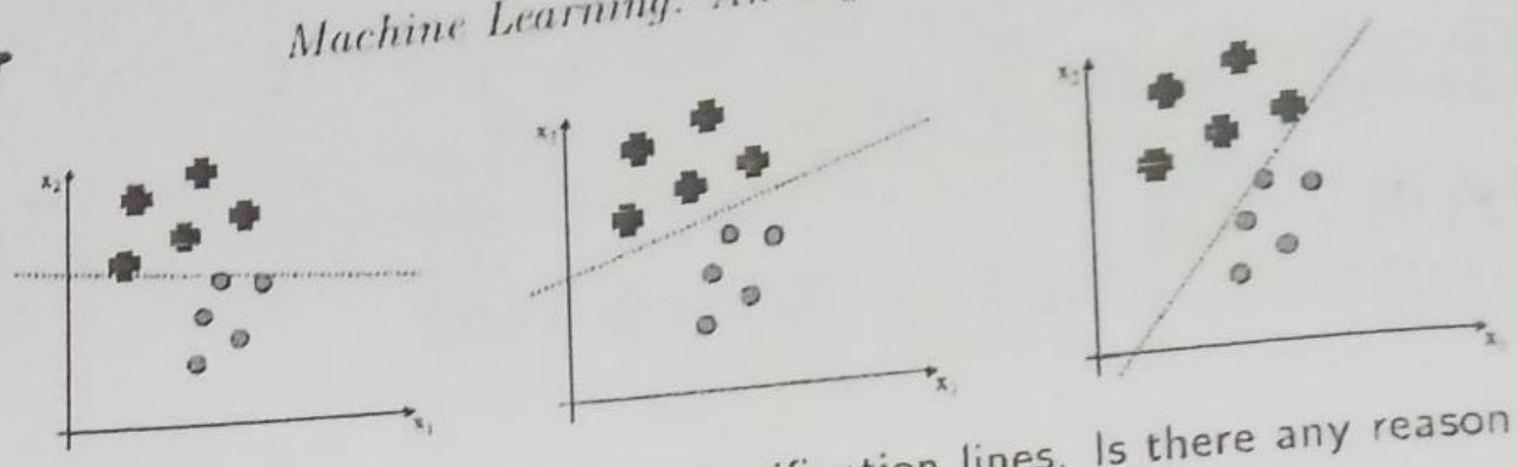


FIGURE 5.1: Three different classification lines. Is there any reason why one is better than the others?

and there are references to some of the more popular ones at the end of the chapter. Some of them include wrappers for Python so that they can be used from within Python.

There is rather more to the SVM than the kernel method: the algorithm also reformulates the classification problem in such a way that we can tell a good classifier from a bad one, even if they both give the same results on a particular dataset. It is this distinction that enables these advanced algorithms to be derived, so that is where we will start.

5.1 Optimal Separation

Figure 5.1 shows a simple classification problem with three different possible linear classification lines. All three of the lines that are drawn separate out the two classes, so in some sense they are 'correct.' However, if you had to pick one of the lines to act as the classifier for a set of test data, I'm guessing that most of you would pick the line shown in the middle picture. It's probably hard to describe exactly why you would do this, but somehow we prefer a line that runs through the middle of the separation between the datapoints from the two classes, staying approximately equidistant from the data in both classes. Of course, if you were feeling smart then you might have asked what criteria you were meant to pick a line based on, and why one of the lines should be any better than the others.

To answer that, we are going to try to define why the line that runs halfway between the two sets of datapoints is better, and then work out some way to quantify that so we can identify the 'optimal' line, that is, the best line according to our criteria. The data that we have used to identify the classification line is our training data. We believe that these data are indicative of some underlying process that we are trying to learn, and that the testing data that the algorithm will be evaluated on after training comes from the same underlying process. However, we don't expect to see exactly the same datapoints in the test dataset, and inevitably some of the points will be closer to the

classifier line, and some will be further away. If we pick the lines shown in the left or right graphs of Figure 5.1 then there is a chance that a datapoint from one class will be on the wrong side of the line, just because we have put the line tight up against some of the datapoints we have seen in the training set. The line in the middle picture doesn't have this problem: like the baby bear's porridge in Goldilocks, it is 'just right.'

How can we quantify this? We can measure the distance that we have to travel away from the line (in a direction perpendicular to the line) before we hit a datapoint. Imagine that we put a 'no-man's land' around the line (shown in Figure 5.2), so that any point that lies within that region is declared to be too close to the line to be accurately classified. This region is symmetric about the line, so that it forms a cylinder about the line in 3D, and a hyper-cylinder in higher dimensions. How large could we make the radius of this cylinder until we started to put points into a no-man's land, where we don't know which class they are from? This largest radius is known as the margin, labelled M. The classifier in the middle of Figure 5.1 has the largest margin of the three. It has the imaginative name of the maximum margin (linear) classifier. The datapoints in each class that lie closest to the classification line have a name as well. They are called support vectors. Using the argument that the best classifier is the one that goes through the middle of a no-man's land, we can now make two arguments: first that the margin should be as large as possible, and second that the support vectors are the most useful datapoints because they are the ones that we might get wrong. This leads to an interesting feature of these algorithms: after training we can throw away all data except for the support vectors, and use them for classification.

Now that we've got a measurement that we can use to find the optimal classification line, we just need to work out how to actually compute it from a given set of datapoints. Let's start by reminding ourselves of some of the things that we worked out in Chapter 2. We can use the standard equation of a straight line to write down our classifier, it is $y = \mathbf{w} \cdot \mathbf{x} + b$, where we are using the same notation as in Chapter 2: the $\mathbf{w}$ is the weight vector (it is a vector, not a matrix, since there is only one output) and $\mathbf{x}$ is the particular input vector, with b being the contribution from the bias weight. We use the classifier line by saying that any $\mathbf{x}$ value that gives a positive value for $\mathbf{w} \cdot \mathbf{x} + b$ is above the line, and so is an example of the '+' class, while any $\mathbf{x}$ that gives a negative value is in the 'o' class. In our new version of this we want to include our no-man's land. So instead of just looking at whether the value of $\mathbf{w} \cdot \mathbf{x} + b$ is positive or negative, we also check whether the absolute value is less than our margin M. Remember that $\mathbf{w} \cdot \mathbf{x}$ is the inner or scalar product, $\mathbf{w} \cdot \mathbf{x} = \sum_i w_i x_i$.

For a given margin value M we can say that any point $\mathbf{x}$ where $\mathbf{w} \cdot \mathbf{x} + b \geq M$ is a plus, and any point where $\mathbf{w} \cdot \mathbf{x} + b \leq -M$ is a circle. Now suppose that we pick a point $\mathbf{x}^+$ that lies on the '+' class boundary line, so that $\mathbf{w} \cdot \mathbf{x}^+ = M$. This is a support vector. If we want to find the closest point that lies on the boundary line for the 'o' class, then we travel perpendicular to the '+'

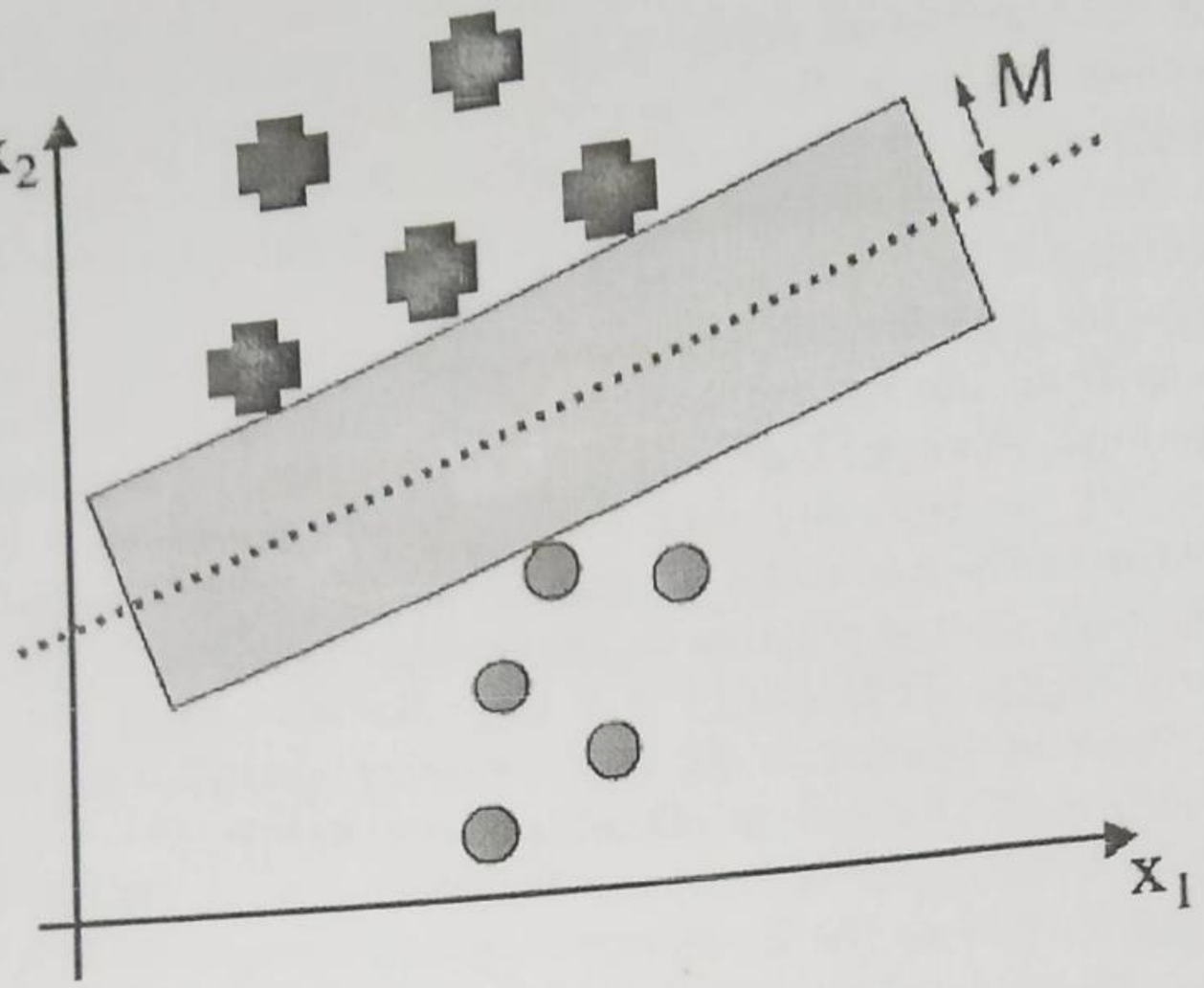


FIGURE 5.2: The margin is the largest region we can put that separates the classes without there being any points inside.

boundary line until we hit the 'o' boundary line. The point that we hit is the closest point, and we'll call it $\mathbf{x}^-$. How far did we have to travel in this direction? Figure 5.2 hopefully makes it clear that the distance we travelled is $2M$. We can use this fact to write down the margin size M in terms of $\mathbf{w}$ if we remember one extra thing from Chapter 2, namely that the weight vector $\mathbf{w}$ is perpendicular to the classifier line. If it is perpendicular to the classifier line, then it is obviously perpendicular to the '+' and 'o' boundary lines too, so the direction we travelled in from $\mathbf{x}^+$ to $\mathbf{x}^-$ is along $\mathbf{w}$, or writing it as an equation (where ν is some distance along the line):

$$\mathbf{x}^- = \mathbf{x}^+ + \nu\mathbf{w}. \tag{5.1}$$

We know that $|\mathbf{x}^- - \mathbf{x}^+| = 2M$, and so we can use the equation above to compute that:

$$M = \frac{1}{2|\mathbf{w}|} = \frac{1}{2\sqrt{\mathbf{w}\cdot\mathbf{w}}}. \tag{5.2}$$

So now, given a classifier line (that is, the vector $\mathbf{w}$ and scalar b that define the line $\mathbf{w}\cdot\mathbf{x} + b$) we can compute the margin M. We can also check that it puts all of the points on the right side of the classification line. Of course, that isn't actually what we want to do: we want to find the $\mathbf{w}$ and b that give us the biggest possible value of M. Equation (5.2) tells us that making M as large as possible is the same as making $\mathbf{w}\cdot\mathbf{w}$ as small as possible. If

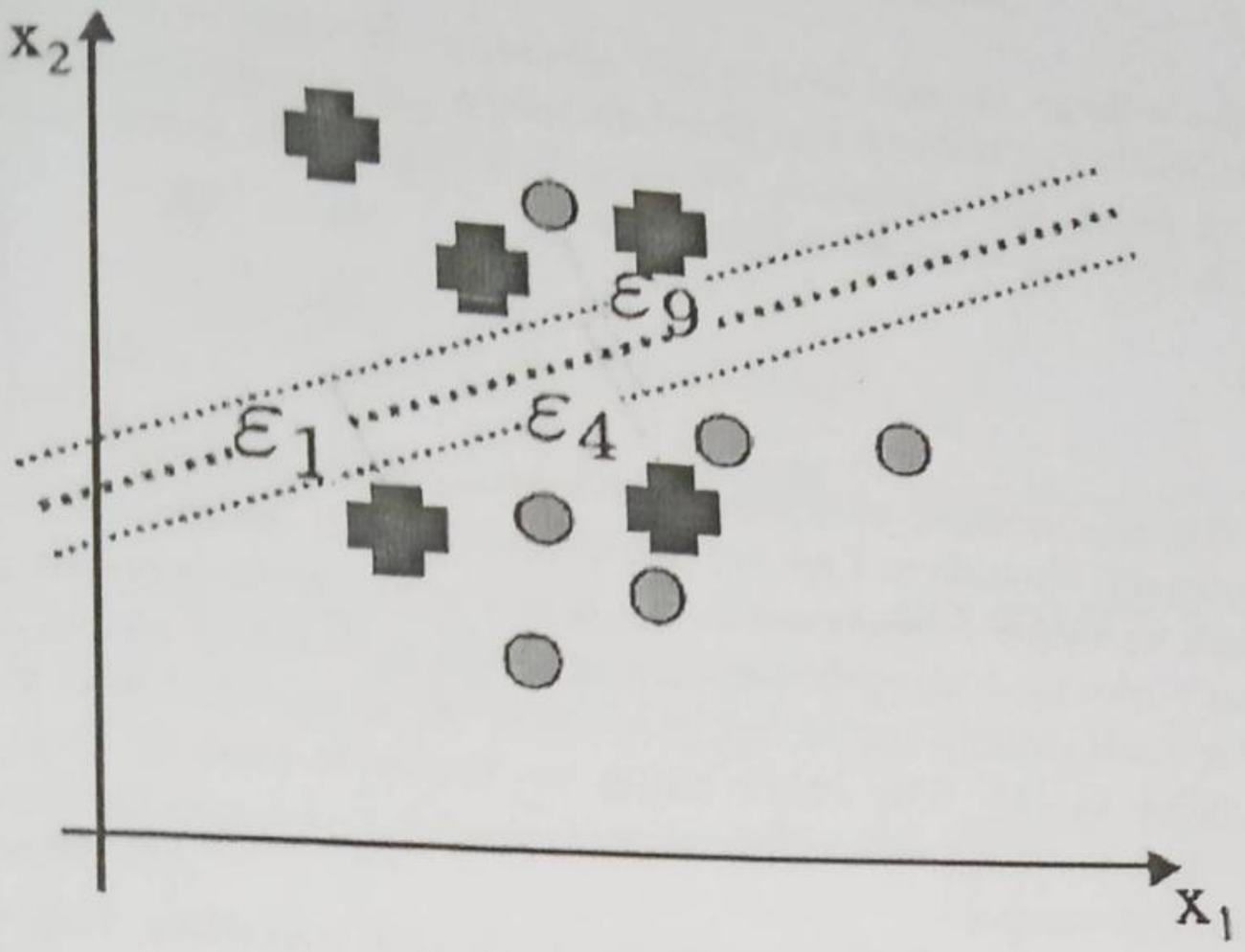


FIGURE 5.3: If the classifier makes some errors, then the distance by which the points are over the border should be used to weight each error in order to decide how bad the classifier is.

that was the only constraint, then we could just set $\mathbf{w} = 0$, and problem would be solved, but we also want the classification line to separate out the '+' data from the 'o', that is, actually act as a classifier. So we are going to need to try to satisfy two problems simultaneously: find a decision boundary that classifies well, while also making $\mathbf{w} \cdot \mathbf{w}$ as small as possible.

How do we decide whether or not a classifier is any good? Obviously, the fewer mistakes that it makes, the better. So we can write down a set of constraints that say that the classifier should get the answer right. To do this we make the target answers for our two classes be ± 1, rather than 0 and 1. We can then write down $t_i \times y_i$, that is, the target multiplied by the output, and this will be positive if the two are the same and negative otherwise. We can write down the equation of the straight line again, which is how we computed y, to see that we require that $t_i(\mathbf{w} \cdot \mathbf{x} + b) \geq 1$.

When comparing classifiers, we should consider the case where one classifier makes a mistake by putting a point just on the wrong side of the line, and another puts the same point a long way onto the wrong side of the line. It can be argued that the first classifier is better than the second, because the mistake was not as serious, so we should include this information in our minimisation criterion. We can do this by modifying the problem. In fact, we have to do major surgery, since we want to add a term into the minimisation problem so that we will now minimise $\mathbf{w} \cdot \mathbf{w} + \lambda \times$ (distance of misclassified points from the correct boundary line). Here, λ is a trade-off parameter that decides how much weight to put onto each of the two criteria—small λ means

we prize a large margin over a few errors, while large λ means the opposite. This actually transforms the problem into a soft-margin classifier, since we are allowing for a few mistakes. Writing this in a more mathematical way, the function that we want to minimise is:

$$L(\mathbf{w}, \epsilon) = \mathbf{w} \cdot \mathbf{w} + \lambda \sum_{i=1}^{R} \epsilon_i, \tag{5.3}$$

where R is the number of misclassified data points, and each ϵ_i is the distance to the correct boundary line for the missing point, which is sometimes known as a slack variable. The constraints don't quite work anymore, either, since they don't mention the possibility of getting something wrong. We now want to say for each point that $\mathbf{w} \cdot \mathbf{x}_i \geq 1 - \epsilon_i$ if the target is 1 and $\mathbf{w} \cdot \mathbf{x}_i \leq -1 + \epsilon_i$ if the target is -1. The other thing to notice is that ϵ_i is a distance, and therefore has to be a positive number. This has to be specified as an additional constraint for each i.

We've made a lot of effort to write down this equation, but we don't know how to solve it. We could use gradient descent, but we would have to put a lot of effort into making it enforce the constraints, and it would be very, very slow and inefficient for the problem. There is a method that is much better suited, which is quadratic programming, which takes advantage of the fact that the problem we have described is quadratic and therefore convex. and has linear constraints. If you want to understand these terms, and don't, then a book on numerical optimisation would be a good start: a couple are given in the references at the end of the chapter. The practical upshot of these facts for us is that the problem can be solved directly and efficiently (i.e., in polynomial time) for the problems that we wish to solve. There are very effective quadratic programming solvers available, but it is not an algorithm that we will consider writing ourselves, being well beyond the scope of this book.

We will, however, work out how to formulate the problem so that it can be presented to a quadratic program solver. This involves transforming the form of the problem by using the technique of Lagrange multipliers, which means that we treat λ in Equation (5.3) as a parameter instead of a constant, and find the minimiser by looking at the derivatives of the function with respect to each of the parameters independently and setting them to zero. The method of Lagrange multipliers is a very useful one for optimisation, and good introductions to the method can be found in many textbooks.

There is another modification to the problem that we will make as well, which is to change it into a maximisation problem by finding its dual. This is an alternative representation of a constraint problem that has the same solution, but swaps the constraints and the objective, producing a version of the problem that is more efficient for quadratic optimisation, and will have certain other benefits later on. Constructing the dual function requires that we use the Karush-Kuhn-Tucker construction to eliminate the $\mathbf{w}$ from Equation (5.3)

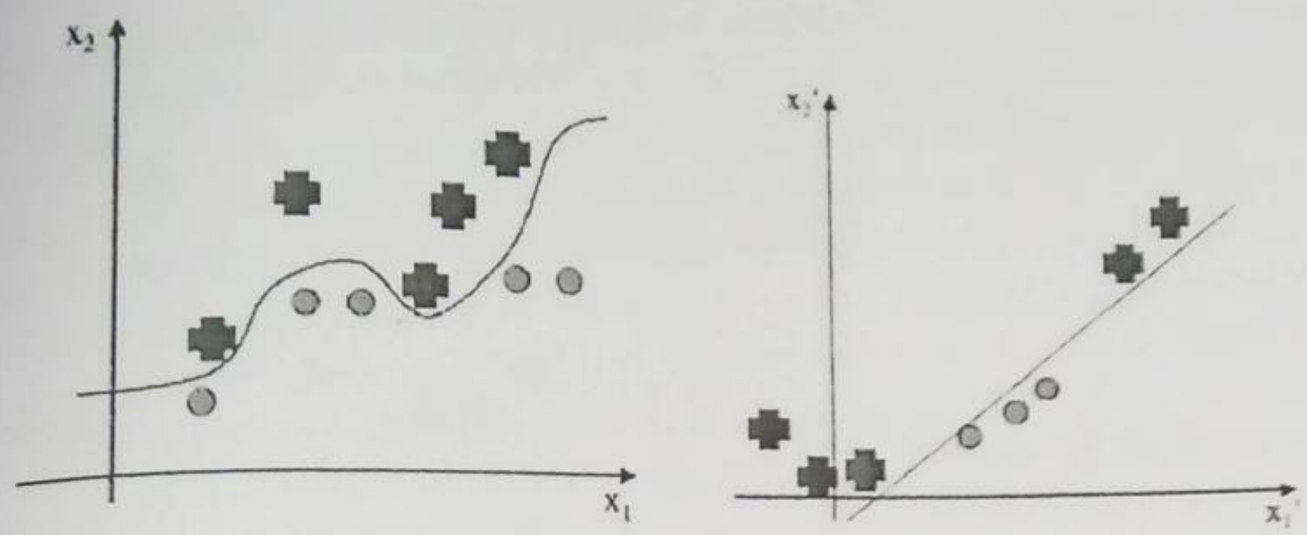


FIGURE 5.4: By modifying the features we hope to find spaces where the data are linearly separable.

(by differentiating with respect to it and setting the derivatives equal to 0) to get:

$$L(\epsilon) = \max \sum_{i-1}^{R} \alpha_i - \frac{1}{2} \sum_{i-1}^{R} \sum_{j-1}^{R} \alpha_i \alpha_j t_i t_j \mathbf{x}_i \cdot \mathbf{x}_j, \tag{5.4}$$

subject to the constraints $0 \leq \alpha_i \leq \lambda$ and $\sum_{i-1}^{R} \alpha_i \mathbf{x}_k = 0$.

5.2 Kernels

Although we've done lots of work up to this point, we haven't actually changed things very much, because we are still finding straight line boundary conditions in the input space of the data. So while the decision boundary that is found could be better than that found by the Perceptron, if there isn't a straight line solution then, just like the Perceptron, our current method won't work. Not ideal for something that's taken lots of effort to work out! It's time to pull our extra piece of magic out of the hat: transformation of the data. To see the idea, have a look at Figure 5.4. It is basically the idea that if we modify the features in some way then we might be able to linearly separate the data, as we did for the XOR problem: if we can use more dimensions then we might be able to find a linear decision boundary that separates the classes. What extra dimensions can we use? We can't invent new data, so the new features will have to be derived from the current ones in some way. Just like in Section 4.4, we are going to introduce new functions $\phi(\mathbf{x})$ of our input features.

We still need to pick what functions to use, of course. If we knew something about the data, then we might be able to identify functions that would be a good idea, but this kind of domain knowledge is not always going to be

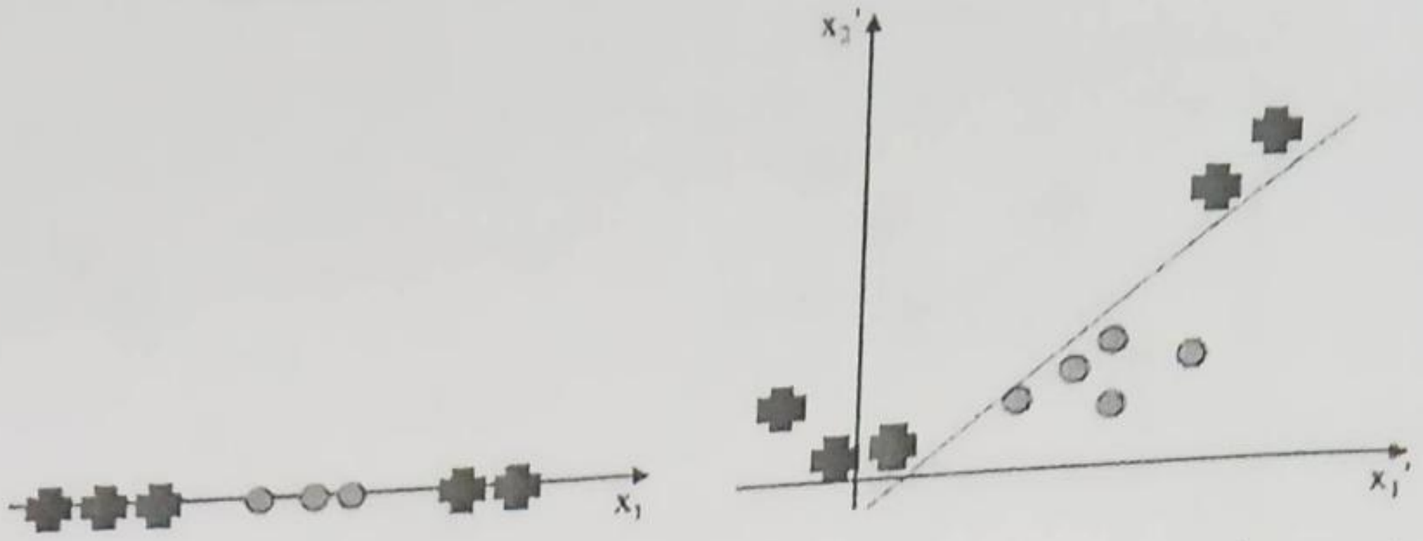


FIGURE 5.5: Using x_1^2 as well as x_1 allows these two classes to be separated.

around, and we would like to automate the algorithm. For now, let's think about a basis that consists of the polynomial of everything up to degree 2. It contains the constant value 1, each of the individual (scalar) input elements $x_1, x_2, \ldots, x_d$, and then the squares of each input element $x_1^2, x_2^2, \ldots, x_d^2$, and finally, the products of each pair of elements $x_1x_2, x_1x_3, \ldots, x_{d-1}x_d$. The total input vector made up of all these things is generally written as $\Phi(\mathbf{x})$: it contains about $d^2/2$ elements. The right of Figure 5.5 shows a 2D version of this (with the constant term suppressed), and I'm going to write it out for the case $d = 3$, with a set of $\sqrt{2}$s in there (the reasons for them will become clear soon):

$$\Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_3, x_1^2, x_2^2, x_3^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \sqrt{2}x_2x_3). \quad (5.5)$$

If there was just one feature, x_1, then we would have changed this from a one-dimensional problem into a three-dimensional one $(1, x_1, x_1^2)$.

Have another look at Equation (5.4). There is no reason why we can't modify it so that the $\mathbf{x}$ variables look like $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$. (Note the slight notational intricacy here: $\mathbf{x}_i$ is the ith input vector, while x_i is the ith element of an input vector.) This seems great, since we don't have to modify our derivation of that equation at all. Except that now the function $\Phi(\mathbf{x}_i)$ has $d^2/2$ elements, and we need to multiply it with another one the same size, and we need to do this R^2 times. This is rather computationally expensive, and if we need to use the powers of the input vector greater than 2 it will be even worse. There is one last piece of trickery that will get us out of this hole: it turns out that we don't actually have to compute $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$. To see how this works, let's work out what $\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y})$ actually is:

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}) = 1 + 2\sum_{i=1}^{d} x_i y_i + \sum_{i=1}^{d} x_i^2 y_i^2 + 2\sum_{i=1}^{m} x_i x_j y_i y_j. \quad (5.6)$$

You might not recognise that you can factorise this equation, but fortunately somebody did: it can be written as $(1 + \mathbf{x} \cdot \mathbf{y})^2$. The dot product here

is in the original space, so it only requires d multiplications, which is obviously much better—this part of the algorithm has now been reduced from $\mathcal{O}(d^2)$ to $\mathcal{O}(d)$. The same thing holds true for the polynomials of any degree s that we are making here, where the cost of the naïve algorithm is $\mathcal{O}(d^s)$. The important thing is that we remove the problem of computing the dot products of all the extended basis vectors, which is expensive, with the computation of a kernel matrix (also known as the Gram matrix) K that is made from the dot product of the original vectors, which is only linear in cost. This is sometimes known as the kernel trick. It means that you don't even have to know what $\Phi(\cdot)$ is, provided you know a kernel. These kernels are the fundamental reason why these methods work, and the reason why we went to all that effort to produce the dual formulation of the problem. They produce a transformation of the data so that they are in a higher-dimensional space, but because the datapoints only appear inside those inner products, we don't actually have to do any computations in those higher-dimensional spaces, only in the original (relatively cheap) low-dimensional space.

So how do we go about finding a suitable kernel? Any symmetric function that is positive definite (meaning that it enforces positivity on the integral of arbitrary functions) can be used as a kernel. This is a result of Mercer's theorem, which also says that it is possible to convolve kernels together and the result will be another kernel. However, there are three different types of basis functions that are commonly used, and they have nice kernels that correspond to them:

1. • polynomials up to some degree s in the elements x_k of the input vector (e.g., x_3^3 or $x_1 \times x_4$) with kernel:

$$\mathrm{K}(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x} \cdot \mathbf{y})^s \tag{5.7}$$

2. • sigmoid functions of the x_ks with parameters κ and δ, and kernel:

$$\mathrm{K}(\mathbf{x}, \mathbf{y}) = \tanh(\kappa \mathbf{x} \cdot \mathbf{y} - \delta) \tag{5.8}$$

3. • radial basis function expansions of the x_ks with parameter σ and kernel:

$$\mathrm{K}(\mathbf{x}, \mathbf{y}) = \exp\left(-(\mathbf{x} - \mathbf{y})^2 / 2\sigma^2\right) \tag{5.9}$$

Choosing which kernel to use and the parameters in these kernels is a tricky problem. While there is some theory based on something known as the Vapnik-Chernik dimension that can be applied, most people just experiment with different values and find one that works, using a validation set as we did for the MLP in Chapter 3.

There are two things that we still need to worry about for the algorithm. One is something that we've discussed in the context of other machine learning algorithms: overfitting, and the other is how we will do testing. The second

one is probably worth a little explaining. We used the kernel trick in order to reduce the computations for the training set. We still need to work out how to do the same thing for the testing set, since otherwise we'll be stuck with doing the $\mathcal{O}(d^s)$ computations. In fact, it isn't too hard to get around this problem, because the forward computation for the weights is $\mathbf{w} \cdot \Phi(\mathbf{x})$, where:

$$\mathbf{w} = \sum_{i \text{ where } \alpha_i > 0} \alpha_i y_i \Phi(\mathbf{x}_i). \tag{5.10}$$

So we still have the computation of $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$, which we can replace using the kernel as before.

The overfitting problem goes away because of the fact that we are still optimising $\mathbf{w} \cdot \mathbf{w}$ (remember that from somewhere a long way back?), which tries to keep $\mathbf{w}$ small, which means that many of the parameters are kept close to 0.

5.2.1 Example: XOR

We motivated the SVM by thinking about how we solved the XOR function in Section 2.3.2. So will the SVM actually solve the problem? We'll need to modify the problem to have targets -1 and 1 rather than 0 and 1, but that is not difficult. Then we'll introduce a basis of all terms up to quadratic in our two features: $1, \sqrt{2}x_1, \sqrt{2}x_2, x_1x_2, x_1^2, x_2^2$, where the $\sqrt{2}$ is to keep the multiplications simple. Then Equation (5.4) looks like:

$$\sum_{i=1}^{4} \alpha_i - \sum_{i,j}^{4} \alpha_i \alpha_j t_i t_j \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j), \tag{5.11}$$

subject to the constraints that $\alpha_1 - \alpha_2 + \alpha_3 - \alpha_4 = 0, \alpha_i \geq 0 \;\; i = 1 \ldots 4$. Solving this (which can be done algebraically) tells us that the classifier line is at $x_1x_2 = 0$. The margin that corresponds to this is $\sqrt{2}$. Unfortunately we can't plot it, since our four points have been transferred into a six-dimensional space. We know that this is not the smallest number that it can be solved in, since we did it in three, but the dimensionality of the kernel space doesn't matter, as all the computations are in the 2D space anyway.

5.2.2 Extensions to the Support Vector Machine

We've talked about SVMs in terms of two-class classification. You might be wondering how to use them for more classes, since we can't use the same methods as we have done to work out the current algorithm. In fact, you can't actually do it in a consistent way. The SVM only works for two classes. This might seem like a major problem, but with a little thought it is possible to find ways around the problem. For the problem of N-class classification,

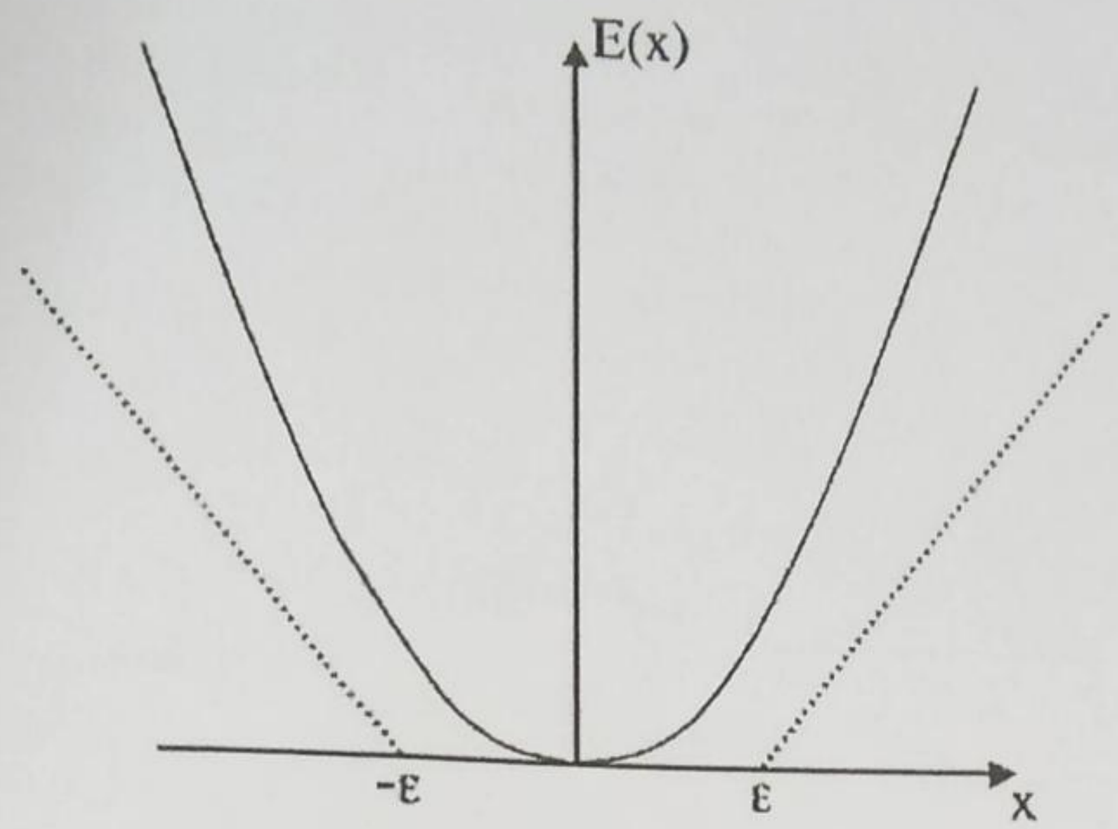


FIGURE 5.6: The ϵ-insensitive error function is zero for any error below ϵ.

you train an SVM that learns to classify class one from all other classes, then another that classifies class two from all the others. So for N-classes, we have N SVMs. This still leaves one problem: how do we decide which of these SVMs is the one that recognises the particular input? The answer is just to choose the one that makes the strongest prediction, that is, the one where the basis vector input point is the furthest into the positive class region.

Interestingly, it is also possible to use the SVM for regression. The key is to take the usual least-squares error function (with the regulariser that keeps the norm of the weights small):

$$\frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2 + \frac{1}{2}\lambda\|\mathbf{w}\|^2, \tag{5.12}$$

and transform it using what is known as a ϵ-insensitive error function (E_ϵ) that gives 0 if the difference between the target and output is less than ϵ (and subtracts ϵ in any other case for consistency). Figure 5.6 shows the form of this error function, which is:

$$\sum_{i=1}^{N} E_\epsilon(t_i - y_i) + \lambda\frac{1}{2}\|\mathbf{w}\|^2. \tag{5.13}$$

You might see this written in other texts with the constant λ in front of the second term replaced by a C in front of the first term. This is equivalent up to scaling. The picture to think of now is almost the opposite of Figure 5.3: we want the predictions to be inside the tube of radius ϵ that surrounds the correct line. To allow for errors, we again introduce slack variables for each datapoint (ϵ_i for datapoint i) with their constraints and follow the same procedure of

introducing Lagrange multipliers, transferring to the dual problem, using a kernel function and solving the problem with a quadratic solver.

There is a lot of advanced work on kernel methods and SVMs. This includes lots of work on the optimisation, including Sequential Minimal Optimisation, and extensions to compute posterior probabilities instead of hard decisions, such as the Relevance Vector Machine. There are some references in the Further Reading section.

There are SVM implementations available via the Internet. They are mostly written in C, but some include wrappers to be called from other languages, including Python. An Internet search will find you some possibilities to try.

Further Reading

The treatment of SVMs here has only skimmed the surface of the topic. There is a useful tutorial paper on SVMs at:

- C.J. Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2): 121–167, 1998.

If you want more information, then any of the following books will provide it (the first is by the creator of SVMs):

- V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, Berlin, Germany, 1995.
- B. Schölkopf, C.J.C. Burges, and A.J. Smola. *Advances in Kernel Methods: Support Vector Learning*. MIT Press, Cambridge, MA, USA, 1999.
- J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 2004.

If you want to know more about quadratic programming, then a good reference is:

- S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.

Other machine learning books that give useful coverage of this area are:

- Chapter 12 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, Berlin, Germany, 2001.
- Chapter 7 of C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.