

Neurons, Neural Networks, and Linear Discriminants

We've spent enough time with the concepts of machine learning, now it is time to actually see it in practice. To start the process, we will return to our demonstration that learning is possible, which is the squishy thing that your skull protects.

3.1 THE BRAIN AND THE NEURON

In animals, learning occurs within the brain. If we can understand how the brain works, then there might be things in there for us to copy and use for our machine learning systems. While the brain is an impressively powerful and complicated system, the basic building blocks that it is made up of are fairly simple and easy to understand. We'll look at them shortly, but it's worth noting that in computational terms the brain does exactly what we want. It deals with noisy and even inconsistent data, and produces answers that are usually correct from very high dimensional data (such as images) very quickly. All amazing for something that weighs about 1.5 kg and is losing parts of itself all the time (neurons die as you age at impressive/depressing rates), but its performance does not degrade appreciably (in the jargon, this means it is **robust**).

So how does it actually work? We aren't actually that sure on most levels, but in this book we are only going to worry about the most basic level, which is the processing units of the brain. These are nerve cells called **neurons**. There are lots of them (100 billion = 10^{11} is the figure that is often given) and they come in lots of different types, depending upon their particular task. However, their general operation is similar in all cases: transmitter chemicals within the fluid of the brain raise or lower the electrical potential inside the body of the neuron. If this **membrane potential** reaches some threshold, the neuron **spikes** or **fires**, and a pulse of fixed strength and duration is sent down the **axon**. The axons divide (**arborise**) into connections to many other neurons, connecting to each of these neurons in a **synapse**. Each neuron is typically connected to thousands of other neurons, so that it is estimated that there are about 100 trillion ($= 10^{14}$) synapses within the brain. After firing, the neuron must wait for some time to recover its energy (the **refractory period**) before it can fire again.

Each neuron can be viewed as a separate processor, performing a very simple computation: deciding whether or not to fire. This makes the brain a massively parallel computer made up of 10^{11} processing elements. If that is all there is to the brain, then we should be able to model it inside a computer and end up with animal or human intelligence inside a computer. This is the view of **strong AI**. We aren't aiming at anything that grand in this

book, but we do want to make programs that learn. So how does learning occur in the brain? The principal concept is **plasticity**: modifying the **strength** of synaptic connections between neurons, and creating new connections. We don't know all of the mechanisms by which the strength of these synapses gets adapted, but one method that does seem to be used was first postulated by Donald Hebb in 1949, and that is what is discussed now.

3.1.1 Hebb's Rule

Hebb's rule says that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. So if two neurons consistently fire simultaneously, then any connection between them will change in strength, becoming stronger. However, if the two neurons never fire simultaneously, the connection between them will die away. The idea is that if two neurons both respond to something, then they should be connected. Let's see a trivial example: suppose that you have a neuron somewhere that recognises your grandmother (this will probably get input from lots of visual processing neurons, but don't worry about that). Now if your grandmother always gives you a chocolate bar when she comes to visit, then some neurons, which are happy because you like the taste of chocolate, will also be stimulated. Since these neurons fire at the same time, they will be connected together, and the connection will get stronger over time. So eventually, the sight of your grandmother, even in a photo, will be enough to make you think of chocolate. Sound familiar? Pavlov used this idea, called **classical conditioning**, to train his dogs so that when food was shown to the dogs and the bell was rung at the same time, the neurons for salivating over the food and hearing the bell fired simultaneously, and so became strongly connected. Over time, the strength of the synapse between the neurons that responded to hearing the bell and those that caused the salivation reflex was enough that just hearing the bell caused the salivation neurons to fire in sympathy.

There are other names for this idea that synaptic connections between neurons and assemblies of neurons can be formed when they fire together and can become stronger. It is also known as **long-term potentiation** and **neural plasticity**, and it does appear to have correlates in real brains.

3.1.2 McCulloch and Pitts Neurons

Studying neurons isn't actually that easy. You need to be able to extract the neuron from the brain, and then keep it alive so that you can see how it reacts in controlled circumstances. Doing this takes a lot of care. One of the problems is that neurons are generally quite small (they must be if you've got 10^{11} of them in your head!) so getting electrodes into the synapses is difficult. It has been done, though, using neurons from the giant squid, which has some neurons that are large enough to see. Hodgkin and Huxley did this in 1952, measuring and writing down differential equations that compute the membrane potential based on various chemical concentrations, something that earned them a Nobel prize. We aren't going to worry about that, instead, we're going to look at a mathematical model of a neuron that was introduced in 1943. The purpose of a mathematical model is that it extracts only the bare essentials required to accurately represent the entity being studied, removing all of the extraneous details. McCulloch and Pitts produced a perfect example of this when they modelled a neuron as:

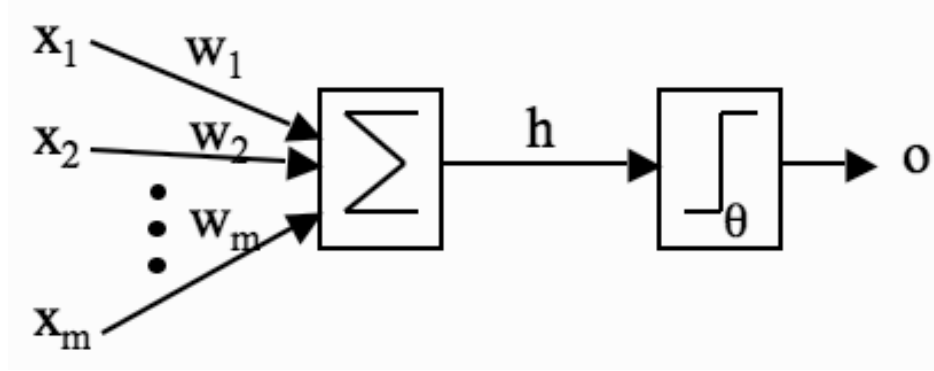


FIGURE 3.1 A picture of McCulloch and Pitts' mathematical model of a neuron. The inputs x_i are multiplied by the weights w_i , and the neurons sum their values. If this sum is greater than the threshold θ then the neuron fires; otherwise it does not.

- (1) **a set of weighted inputs** w_i that correspond to the synapses
- (2) **an adder** that sums the input signals (equivalent to the membrane of the cell that collects electrical charge)
- (3) **an activation function** (initially a threshold function) that decides whether the neuron fires ('spikes') for the current inputs

A picture of their model is given in Figure 3.1, and we'll use the picture to write down a mathematical description. On the left of the picture are a set of input nodes (labelled x_1, x_2, \dots, x_m). These are given some values, and as an example we'll assume that there are three inputs, with $x_1 = 1, x_2 = 0, x_3 = 0.5$. In real neurons those inputs come from the outputs of other neurons. So the 0 means that a neuron didn't fire, the 1 means it did, and the 0.5 has no biological meaning, but never mind. (Actually, this isn't quite fair, but it's a long story and not very relevant.) Each of these other neuronal firings flowed along a synapse to arrive at our neuron, and those synapses have strengths, called **weights**. The strength of the synapse affects the strength of the signal, so we multiply the input by the weight of the synapse (so we get $x_1 \times w_1$ and $x_2 \times w_2$, etc.). Now when all of these signals arrive into our neuron, it adds them up to see if there is enough strength to make it fire. We'll write that as

$$h = \sum_{i=1}^m w_i x_i, \quad (3.1)$$

which just means sum (add up) all the inputs multiplied by their synaptic weights. I've assumed that there are m of them, where $m = 3$ in the example. If the synaptic weights are $w_1 = 1, w_2 = -0.5, w_3 = -1$, then the inputs to our model neuron are $h = 1 \times 1 + 0 \times -0.5 + 0.5 \times -1 = 1 + 0 - 0.5 = 0.5$. Now the neuron needs to decide if it is going to fire. For a real neuron, this is a question of whether the membrane potential is above some threshold. We'll pick a threshold value (labelled θ), say $\theta = 0$ as an example. Now, does our neuron fire? Well, $h = 0.5$ in the example, and $0.5 > 0$, so the neuron does fire, and produces output 1. If the neuron did not fire, it would produce output 0.

The McCulloch and Pitts neuron is a binary threshold device. It sums up the inputs (multiplied by the synaptic strengths or weights) and either fires (produces output 1) or does not fire (produces output 0) depending on whether the input is above some threshold. We can write the second half of the work of the neuron, the decision about whether or not to fire (which is known as an **activation function**), as:

$$o = g(h) = \begin{cases} 1 & \text{if } h > \theta \\ 0 & \text{if } h \leq \theta. \end{cases} \quad (3.2)$$

This is a very simple model, but we are going to use these neurons, or very simple variations on them using slightly different activation functions (that is, we'll replace the threshold function with something else) for most of our study of neural networks. In fact, these neurons might look simple, but as we shall see, a network of such neurons can perform any computation that a normal computer can, provided that the weights w_i are chosen correctly. So one of the main things we are going to talk about for the next few chapters is methods of setting these weights.

3.1.3 Limitations of the McCulloch and Pitts Neuronal Model

One question that is worth considering is how realistic is this model of a neuron? The answer is: not very. Real neurons are much more complicated. The inputs to a real neuron are not necessarily summed linearly: there may be non-linear summations. However, the most noticeable difference is that real neurons do not output a single output response, but a **spike train**, that is, a sequence of pulses, and it is this spike train that encodes information. This means that neurons don't actually respond as threshold devices, but produce a graded output in a continuous way. They do still have the transition between firing and not firing, though, but the threshold at which they fire changes over time. Because neurons are biochemical devices, the amount of neurotransmitter (which affects how much charge they required to spike, amongst other things) can vary according to the current state of the organism. Furthermore, the neurons are not updated sequentially according to a computer clock, but update themselves randomly (**asynchronously**), whereas in many of our models we will update the neurons according to the clock. There are neural network models that are asynchronous, but for our purposes we will stick to algorithms that are updated by the clock.

Note that the weights w_i can be positive or negative. This corresponds to **excitatory** and **inhibitory** connections that make neurons more likely to fire and less likely to fire, respectively.

Both of these types of synapses do exist within the brain, but with the McCulloch and Pitts neurons, the weights can change from positive to negative or vice versa, which has not been seen biologically—synaptic connections are either excitatory or inhibitory, and never change from one to the other. Additionally, real neurons can have synapses that link back to themselves in a feedback loop, but we do not usually allow that possibility when we make networks of neurons. Again, there are exceptions, but we won't get into them.

It is possible to improve the model to include many of these features, but the picture is complicated enough already, and McCulloch and Pitts neurons already provide a great deal of interesting behaviour that resembles the action of the brain, such as the fact that networks of McCulloch and Pitts neurons can memorise pictures and learn to represent functions and classify data, as we shall see in the next couple of chapters. 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.

3.2 NEURAL NETWORKS

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**.

Before we worry too much about this, let's think about what learning is. In the Introduction 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 3.1) 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; they are the model of the synapse! Getting excited about neurons turns out to be 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.

3.3 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 3.2. 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

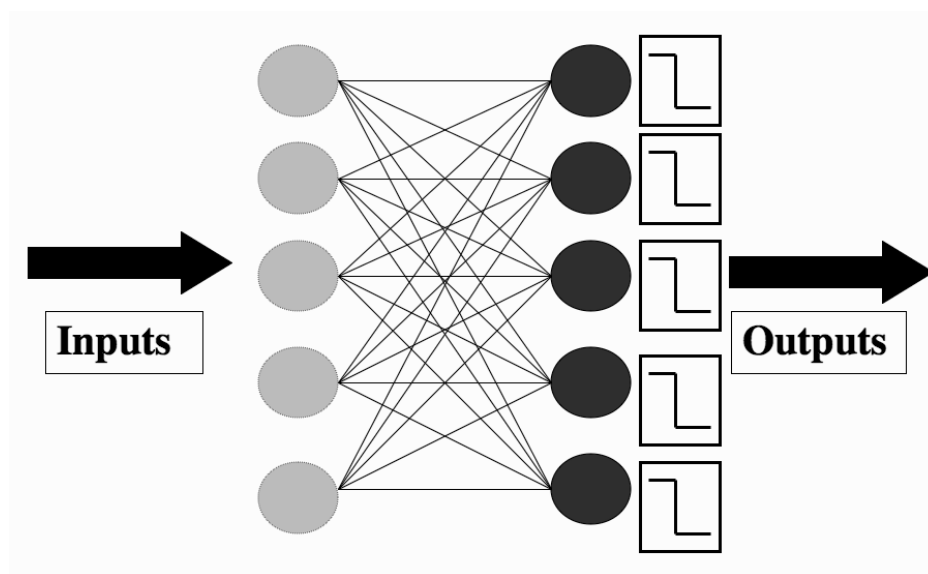


FIGURE 3.2 The Perceptron network, consisting of a set of input nodes (left) connected to McCulloch and Pitts neurons using weighted connections.

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 3.2 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 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 (3.1) and (3.2) 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 3.2, 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 4, 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 $y_k - t_k$ (the difference between the output y_k , which is what the neuron did, and the target for that neuron, t_k , which is what the neuron should have done. This is a possible **error function**). If it is negative then the neuron should have fired and didn't, so we make the weights bigger, and vice versa if it is positive, which we can do by subtracting the error value. 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: $\Delta w_{ik} = -(y_k - t_k) \times x_i$, 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 3.3.2. 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} :

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i. \quad (3.3)$$

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 4.3.3), but for now we will predefine the maximum number of iterations, T . Of course, if the network got all of the inputs correct, then this would also be a good time to stop.

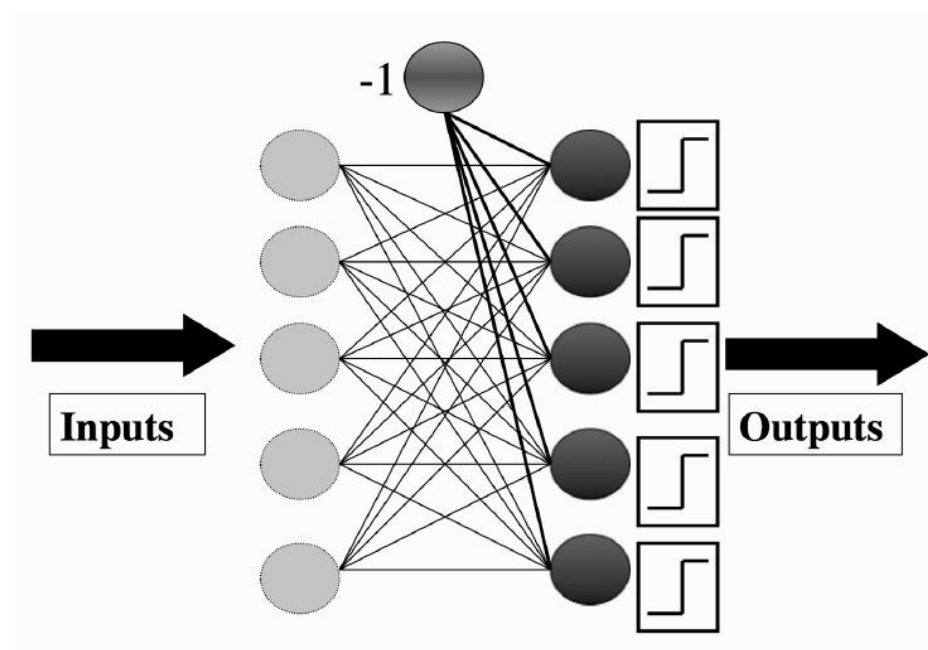


FIGURE 3.3 The Perceptron network again, showing the bias input.

3.3.1 The Learning Rate η

Equation (3.3) 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 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. It doesn't matter for the Perceptron algorithm, but for many of the algorithms that we will see in the book, the learning rate is a crucial parameter.

3.3.2 The Bias Input

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; in this book I'm going to use -1 to make it stand out, but any non-zero value will do). 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 called a **bias node**, and its weights are usually given a 0 subscript, so that the weight connecting it to the j th neuron is w_{0j} .

3.3.3 The Perceptron Learning Algorithm

We are now ready to write our first learning algorithm. It might be useful to keep Figure 3.3 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 or until all the outputs are correct:
 - * for each input vector:
 - compute the activation of each neuron j using activation function g :

$$y_j = g\left(\sum_{i=0}^m w_{ij}x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^m w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^m w_{ij}x_i \leq 0 \end{cases} \quad (3.4)$$

- update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \quad (3.5)$$

- **Recall**

- compute the activation of each neuron j using:

$$y_j = g\left(\sum_{i=0}^m w_{ij}x_i\right) = \begin{cases} 1 & \text{if } w_{ij}x_i > 0 \\ 0 & \text{if } w_{ij}x_i \leq 0 \end{cases} \quad (3.6)$$

Note that the code on the website for the Perceptron has a different form, as will be discussed in Section 3.3.5.

Computing the computational complexity of this algorithm is very easy. The recall phase

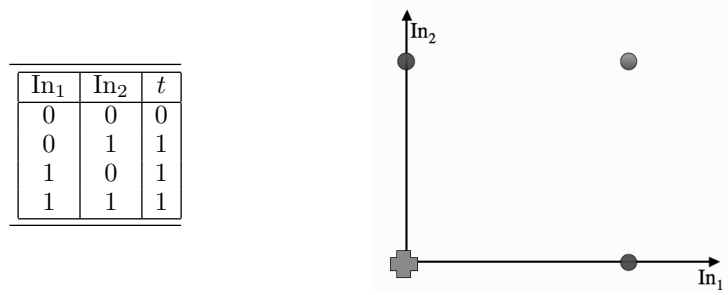


FIGURE 3.4 Data for the OR logic function and a plot of the four datapoints.

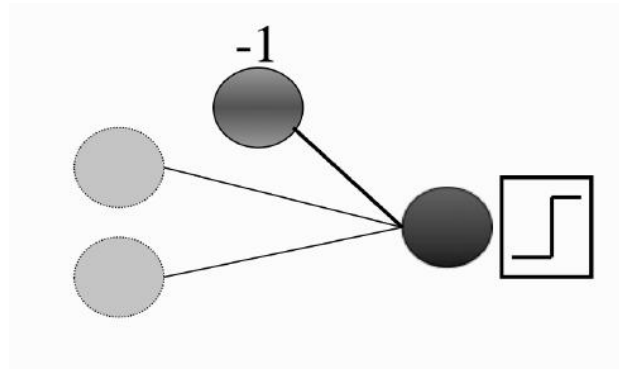


FIGURE 3.5 The Perceptron network for the example in Section 3.3.4.

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.

3.3.4 An Example of Perceptron Learning: Logic Functions

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 3.4; 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 3.5.

As you can see from Figure 3.5, 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 , so the value that reaches the neuron is $-0.05 \times -1 +$

$-0.02 \times 0 + 0.02 \times 0 = 0.05$. 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 (3.3) to each of the weights separately (we'll pick a value of $\eta = 0.25$ for the example):

$$w_0 : -0.05 - 0.25 \times (1 - 0) \times -1 = 0.2, \quad (3.7)$$

$$w_1 : -0.02 - 0.25 \times (1 - 0) \times 0 = -0.02, \quad (3.8)$$

$$w_2 : 0.02 - 0.25 \times (1 - 0) \times 0 = 0.02. \quad (3.9)$$

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:

$$w_0 : 0.2 - 0.25 \times (0 - 1) \times -1 = -0.05, \quad (3.10)$$

$$w_1 : -0.02 - 0.25 \times (0 - 1) \times 0 = -0.02, \quad (3.11)$$

$$w_2 : 0.02 - 0.25 \times (0 - 1) \times 1 = 0.27. \quad (3.12)$$

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.

3.3.5 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 (Appendix A 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 (this code can be found on the book website):

```
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 `np.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 **A** and **B** where **A** is size $m \times n$, then the size of **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 **AB** (but not necessarily **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 **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 **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 **B**, rotating it to match the direction, and multiplying it by the first row of **A**, and so on. As an example:

$$\begin{pmatrix} 3 & 4 & 5 \\ 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{pmatrix} \quad (3.13)$$

$$= \begin{pmatrix} 3 \times 1 + 4 \times 2 + 5 \times 3 & 3 \times 3 + 4 \times 4 + 5 \times 5 \\ 2 \times 1 + 3 \times 2 + 4 \times 3 & 2 \times 3 + 3 \times 4 + 4 \times 5 \end{pmatrix} \quad (3.14)$$

$$= \begin{pmatrix} 26 & 50 \\ 20 & 38 \end{pmatrix} \quad (3.15)$$

NumPy can do this multiplication for us, using the `np.dot()` function (which is a rather strange name mathematically, but never mind). So to reproduce the calculation above, we use (where `>>>` denotes the Python command line, and so this is code to be typed in, with the answers provided by the Python interpreter shown afterwards):

```
>>> import numpy as np
>>> a = np.array([[3,4,5],[2,3,4]])
>>> b = np.array([[1,3],[2,4],[3,5]])
>>> np.dot(a,b)
array([[26, 50],
       [20, 38]])
```

The `np.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 `np.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,

```
>>> np.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 of code as:

```
# Compute activations
activations = np.dot(inputs,self.weights)

# Threshold the activations
return np.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 `pcnfw` 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 `np.dot(inputs,targets - activations)` we need to turn the `inputs` matrix around so that it is $m \times N$. This is done using the `np.transpose()` function, which swaps the rows and columns over (so using matrix `a` above again) we get:

```
>>> np.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):

```
self.weights -= eta*np.dot(np.transpose(inputs),self.activations-targets)
```

Assuming that you make sure in advance that all your input matrices are the correct size (the `np.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 `np.concatenate()` function, making a 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 = np.concatenate((inputs,-np.ones((self.nData,1))),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 4.2.2. 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 = np.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`. Note that this is a different version of the algorithm because it is a **batch** version: all of the inputs go forward through the algorithm, and then the error is computed and the weights are changed. This is different to the **sequential** version that was written down in the first algorithm. The batch version is simpler to write in Python and often works better.

We now move on to seeing the code working, 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 print-out 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).

```
>>> import numpy as np
>>> inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
>>> targets = np.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
[[-0.03755646]
 [ 0.01484562]
 [ 0.21173977]]
Final outputs are:
[[0]
 [0]
 [0]
 [0]]
Iteration: 1
[[ 0.46244354]
 [ 0.51484562]
 [-0.53826023]]
Final outputs are:
[[1]
 [1]
 [1]
 [1]]
Iteration: 2
[[ 0.46244354]
 [ 0.51484562]
 [-0.28826023]]
```

```
Final outputs are:
```

```
[[1]
 [1]
 [1]
 [1]]
```

```
Iteration: 3
```

```
[[ 0.46244354]
 [ 0.51484562]
 [-0.03826023]]
```

```
Final outputs are:
```

```
[[1]
 [1]
 [1]
 [1]]
```

```
Iteration: 4
```

```
[[ 0.46244354]
 [ 0.51484562]
 [ 0.21173977]]
```

```
Final outputs are:
```

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

```
Iteration: 5
```

```
[[ 0.46244354]
 [ 0.51484562]
 [ 0.21173977]]
```

```
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 3.6 shows 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 3.4.

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 -1 s on in this case, using:

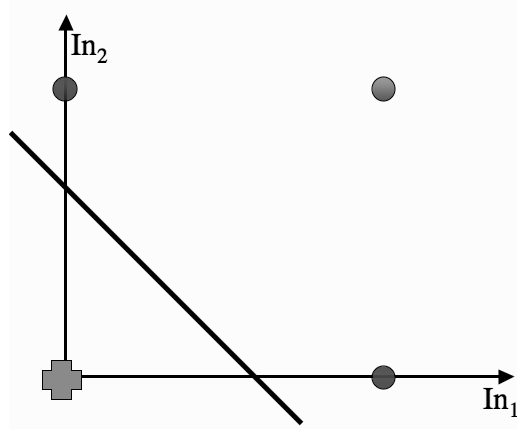


FIGURE 3.6 The decision boundary computed by a Perceptron for the OR function.

```
>>> # Add the inputs that match the bias node
>>> inputs_bias = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),2
axis=1)
>>> pcn.pcnfwd(inputs_bias,weights)
```

The results on this **test data** are what you can use in order to compute the accuracy of the training algorithm using the methods that were described in Section 2.2.

In terms of learning about a set of data we have 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.

3.4 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 3.4, you should be able to draw a straight line that separates out the crosses from the circles without difficulty (it is done in Figure 3.6). 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 3.7.

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

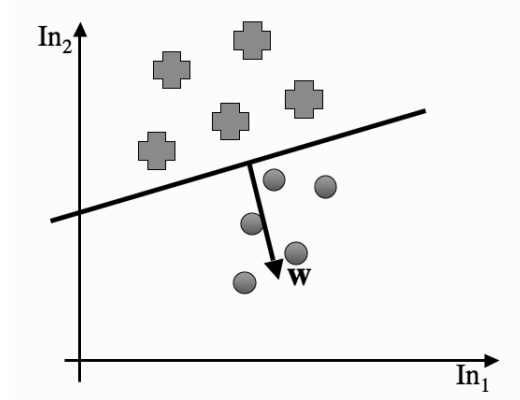


FIGURE 3.7 A decision boundary separating two classes of data.

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, $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$, where θ is the angle between \mathbf{a} and \mathbf{b} and $\|\mathbf{a}\|$ 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 `np.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:

$$\mathbf{x}_1 \cdot \mathbf{w}^T = \mathbf{x}_2 \cdot \mathbf{w}^T \quad (3.16)$$

$$\Rightarrow (\mathbf{x}_1 - \mathbf{x}_2) \cdot \mathbf{w}^T = 0. \quad (3.17)$$

What does this last equation mean? In order for the inner product to be 0, either $\|\mathbf{a}\|$ or $\|\mathbf{b}\|$ or $\cos \theta$ needs to be zero. There is no reason to believe that $\|\mathbf{a}\|$ or $\|\mathbf{b}\|$ 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 3.7.

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 line are called **linearly separable** cases. What happens if the classes that we want to learn about are not linearly separable? It turns out that making such a function is very easy: there is even one that matches a logic function. Before we have a look at it, it is worth thinking about what happens when we have more than one output neuron. The weights for each neuron separately describe a straight line, so by putting together several neurons we get several straight lines that each try to separate different parts of the space. Figure 3.8 shows an example of decision boundaries computed by a Perceptron with four neurons; by putting them together we can get good separation of the classes.

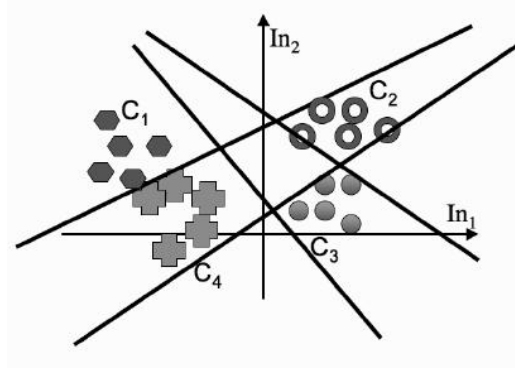


FIGURE 3.8 Different decision boundaries computed by a Perceptron with four neurons.

3.4.1 The Perceptron Convergence Theorem

Actually, it is not quite true that we have reached 1969. There is one more important fact that was known: Rosenblatt's 1962 proof that, given a linearly separable dataset, the Perceptron will converge to a solution that separates the classes, and that it will do it after a finite number of iterations. In fact, the number of iterations is bounded by $1/\gamma^2$, where γ is the distance between the separating hyperplane and the closest datapoint to it. The proof of this theorem only requires some algebra, and so we will work through it here. We will assume that the length of every input vector $\|\mathbf{x}\| \leq 1$, although it isn't strictly necessary provided that they are bounded by some constant R .

First, we know that there is some weight vector \mathbf{w}^* that separates the data, since we have assumed that it is linearly separable. The Perceptron learning algorithm aims to find some vector \mathbf{w} that is parallel to \mathbf{w}^* , or as close as possible. To see whether two vectors are parallel we use the inner product $\mathbf{w}^* \cdot \mathbf{w}$. When the two vectors are parallel, the angle between them is $\theta = 0$ and so $\cos \theta = 1$, and so the size of the inner product is a maximum. If we therefore show that at each weight update $\mathbf{w}^* \cdot \mathbf{w}$ increases, then we have nearly shown that the algorithm will converge. However, we do need a little bit more, because $\mathbf{w}^* \cdot \mathbf{w} = \|\mathbf{w}^*\| \|\mathbf{w}\| \cos \theta$, and so we also need to check that the length of \mathbf{w} does not increase too much as well.

Hence, when we consider a weight update, there are two checks that we need to make: the value of $\mathbf{w}^* \cdot \mathbf{w}$ and the length of \mathbf{w} .

Suppose that at the t th iteration of the algorithm, the network sees a particular input \mathbf{x} that should have output y , and that it gets this input wrong, so $y\mathbf{w}^{(t-1)} \cdot \mathbf{x} < 0$, where the $(t-1)$ index means the weights at the $(t-1)$ st step. This means that the weights need to be updated. This weight update will be $\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} + y\mathbf{x}$ (where we have set $\eta = 1$ for simplicity, and because it is fine for the Perceptron).

To see how this changes the two values we are interested in, we need to do some computation:

$$\begin{aligned}
 \mathbf{w}^* \cdot \mathbf{w}^{(t)} &= \mathbf{w}^* \cdot (\mathbf{w}^{(t-1)} + y\mathbf{x}) \\
 &= \mathbf{w}^* \cdot \mathbf{w}^{(t-1)} + y\mathbf{w}^* \cdot \mathbf{x} \\
 &\geq \mathbf{w}^* \cdot \mathbf{w}^{(t-1)} + \gamma
 \end{aligned} \tag{3.18}$$

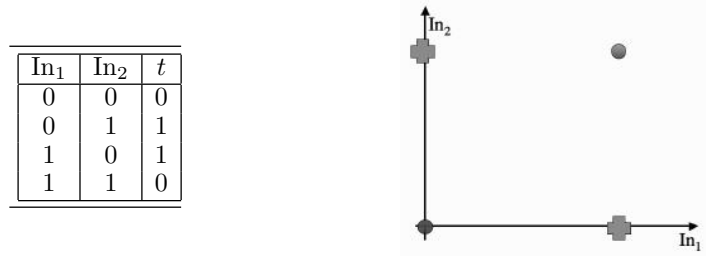


FIGURE 3.9 Data for the XOR logic function and a plot of the four datapoints.

where γ is that smallest distance between the optimal hyperplane defined by \mathbf{w}^* and any datapoint.

This means that at each update of the weights, this inner product increases by at least γ , and so after t updates of the weights, $\mathbf{w}^* \cdot \mathbf{w}^{(t)} \geq t\gamma$. We can use this to put a lower bound on the length of $\|\mathbf{w}^{(t)}\|$ by using the Cauchy–Schwartz inequality, which tells us that $\mathbf{w}^* \cdot \mathbf{w}^{(t)} \leq \|\mathbf{w}^*\| \|\mathbf{w}^{(t)}\|$ and so $\|\mathbf{w}^{(t)}\| \geq t\gamma$.

The length of the weight vector after t steps is:

$$\begin{aligned}
 \|\mathbf{w}^{(t)}\|^2 &= \|\mathbf{w}^{(t-1)} + y\mathbf{x}\|^2 \\
 &= \|\mathbf{w}^{(t-1)}\|^2 + y^2\|\mathbf{x}\|^2 + 2y\mathbf{w}^{(t-1)} \cdot \mathbf{x} \\
 &\leq \|\mathbf{w}^{(t-1)}\|^2 + 1
 \end{aligned} \tag{3.19}$$

where the last line follows because $y^2 = 1$, $\|\mathbf{x}\| \leq 1$, and the network made an error, so the $\mathbf{w}^{(t-1)}$ and \mathbf{x} are perpendicular to each other. This tells us that after t steps, $\|\mathbf{w}^{(t)}\|^2 \leq k$.

We can put these two inequalities together to get that:

$$t\gamma \leq \|\mathbf{w}^{(t-1)}\| \leq \sqrt{k}, \tag{3.20}$$

and so $t \leq 1/\gamma^2$. Hence after we have made that many updates the algorithm must have converged.

We have shown that if the weights are linearly separable then the algorithm will converge, and that the time that this takes is a function of the distance between the separating hyperplane and the nearest datapoint. This is called the **margin**, and in Chapter 8 we will see an algorithm that uses this explicitly. Note that the Perceptron stops learning as soon as it gets all of the training data correct, and so there is no guarantee that it will find the largest margin, just that if there is a linear separator, it will find it. Further, we still don't know what happens if the data are not linearly separable. To see that, we will move on to just such an example.

3.4.2 The Exclusive Or (XOR) Function

The XOR has the same four input points as the OR function, but looking at Figure 3.9, you should be able to convince yourself that you can't draw a 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 analysis above is correct, then the Perceptron will fail to get the correct answer, and using the Perceptron code above we find:

```
>>> targets = np.array([[0],[1],[1],[0]])
>>> pcn.pctrain(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]
 [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.

3.4.3 A Useful Insight

From the discussion in Section 3.4.2 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 3.10. 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

In ₁	In ₂	In ₃	Output
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	1

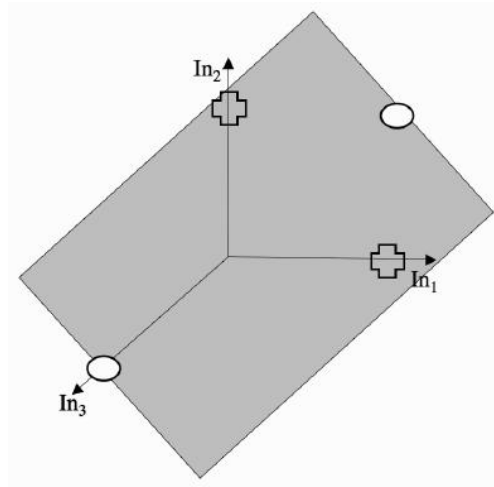


FIGURE 3.10 A decision boundary (the shaded plane) solving the XOR problem in 3D with the crosses below the surface and the circles above it.

dimension. So the truth table for the function is the one shown on the left side of Figure 3.10 (where ‘In₃’ has been added, and only affects the point at (0,0)).

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

```
>>> inputs = np.array([[0,0,1],[0,1,0],[1,0,0],[1,1,0]])
>>> pcn.pctrain(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 function, provided that you **project** the data into the correct set of dimensions. There is 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 8.

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 3.11 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

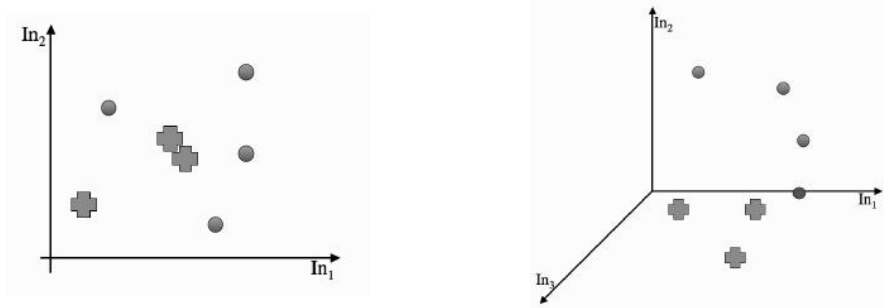


FIGURE 3.11 *Left:* Non-separable 2D dataset. *Right:* The same dataset with third coordinate $x_1 \times x_2$, which makes it separable.

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 3.5, 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.

3.4.4 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 (where not all of the **import** lines are used immediately, but will be required as more code is developed):

```
>>> import os
>>> import pylab as pl
>>> import numpy as np
>>> import pcn

>>> os.chdir('/Users/srmarsla/Book/Datasets/pima')
>>> pima = np.loadtxt('pima-indians-diabetes.data',delimiter=',')
>>> np.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 **np.loadtxt()** command the **delimiter** specifies which character is used to separate out the datapoints. The **np.shape()** method tells that there are 768

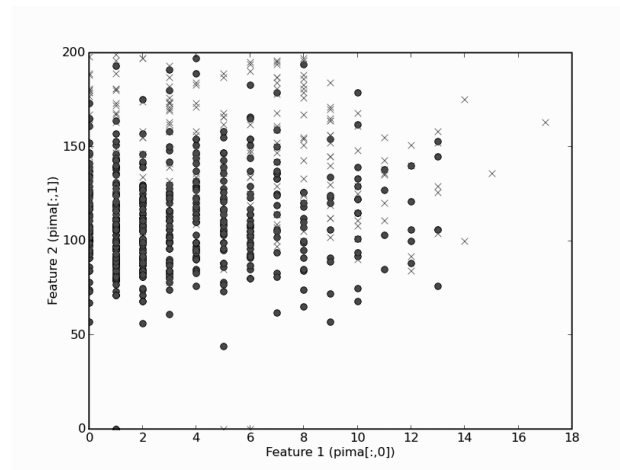


FIGURE 3.12 Plot of the first two dimensions of the Pima Indians dataset showing the two classes as 'x' and 'o'.

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 `np.where` command. Once you have worked that out, you will be able to plot them with different shapes and colours. The `pl.plot` command is in Matplotlib, so you'll need to import that (using `import pylab as pl`) 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:

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

to plot the first two dimensions as green circles and red crosses, which (up to colour, of course) should look like Figure 3.12. The `pl.ion()` command ensures that the data is actually plotted, and might not be needed depending upon your precise software setup; this is also true of the `pl.show()` command, which ensures that the graph does not vanish when 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.pcctrain(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, and we have already seen that this is unfair in Section 2.2, 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]
traingt = pima[:,2,8:9]
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 by preparing the data a little, or **preprocessing** it.

3.4.5 Preprocessing: Data Preparation

Machine learning algorithms tend to learn much more effectively if the inputs and targets are prepared for analysis before the network is trained. As the most basic example, the neurons that we are using give outputs of 0 and 1, and so if the target values are not 0 and 1, then they should be transformed so that they are. 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 input data 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**. While normalisation is not essential for every algorithm, but it is usually beneficial, and for some of the other algorithms that we will see, the normalisation will be essential.

In NumPy it is very easy to perform the normalisation by using the built-in `np.mean()` and `np.var()` functions; the only place where care is needed is along which axis the mean and variance are computed: `axis=0` sums down the columns and `axis=1` sums across the rows. Note that only the input variables are normalised in this code. 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.

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

There is one thing to be careful of, 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, 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.

Normalisation can be done without knowing anything about the dataset in advance. However, there is often useful preprocessing that can be done by looking at the data. For example, in the Pima dataset, 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 `np.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[np.where(pima[:,0]>8),0] = 8

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

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 2.4.2. We can also consider methods of **dimensionality reduction**, which produce lower dimensional representations of the data that still include the relevant information; see Chapter 6 for more details.

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.

3.5 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

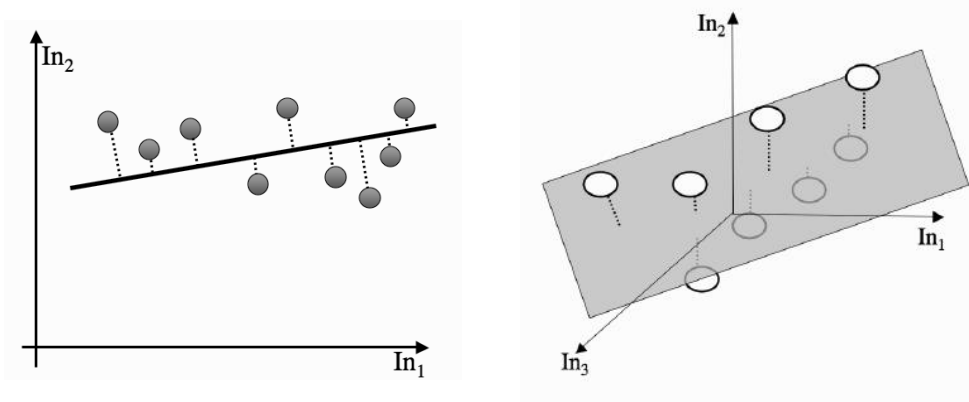


FIGURE 3.13 Linear regression in two and three dimensions.

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 3.13 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 Pythagoras' theorem to know the distance. Now, we can try to minimise an **error function** that measures the sum of all these distances. If we 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:

$$\sum_{j=0}^N \left(t_j - \sum_{i=0}^M \beta_i x_{ij} \right)^2. \quad (3.21)$$

This can be written in matrix form as:

$$(\mathbf{t} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{t} - \mathbf{X}\boldsymbol{\beta}), \quad (3.22)$$

where \mathbf{t} is a column vector containing 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 (column) parameter vector $\boldsymbol{\beta}$ and setting the derivative to 0, which means that $\mathbf{X}^T(\mathbf{t} - \mathbf{X}\boldsymbol{\beta}) = 0$ (to see this, expand out the brackets, remembering that $\mathbf{A}\mathbf{B}^T = \mathbf{B}^T\mathbf{A}$ and note that the term $\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{t} = \mathbf{t}^T\mathbf{X}\boldsymbol{\beta}$ since they are

both a scalar term), which has the solution $\beta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$ (assuming that the matrix $\mathbf{X}^T \mathbf{X}$ can be inverted). Now, for a given input vector \mathbf{z} , the prediction is $\mathbf{z}\beta$. 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 as columns) and its determinant is non-zero.

Computing this is very simple in Python, using the `np.linalg.inv()` function in NumPy. In fact, the entire function can be written as (where the `\n` symbol denotes a linebreak in the text, so that the command continues on the next line):

```
def linreg(inputs,targets):
    inputs = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),\n
axis=1)
    beta = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(inputs),\n
inputs)),np.transpose(inputs)),targets)

    outputs = np.dot(inputs,beta)
```

3.5.1 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, trying to fit a surface to a set of 0 and 1 points. Worse, we will view it as an error if we get say 1.25 and the output should be 1, so points that are in some sense too correct will receive a penalty! 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 a 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 `np.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 `np.loadtxt` that they are comments, using:

```
auto = np.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 = np.concatenate((testin, -np.ones((np.shape(testin)[0], 1))), axis=1)
testout = np.dot(testin, beta)
error = np.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 6.1 once we have built up the understanding we need.

FURTHER READING

If you are interested in real brains and want to know more about them, then there are plenty of popular science books that should interest you, including:

- Susan Greenfield. *The Human Brain: A Guided Tour*. Orion, London, UK, 2001.
- S. Aamodt and S. Wang. *Welcome to Your Brain: Why You Lose Your Car Keys but Never Forget How to Drive and Other Puzzles of Everyday Life*. Bloomsbury, London, UK, 2008.

If you are looking for something a bit more formal, then the following is a good place to start (particularly the 'Roadmaps' at the beginning):

- Michael A. Arbib, editor. *The Handbook of Brain Theory and Neural Networks*, 2nd edition, MIT Press, Cambridge, MA, USA, 2002.

The original paper by McCulloch and Pitts is:

- W.S. McCulloch and W. Pitts. A logical calculus of ideas imminent in nervous activity. *Bulletin of Mathematics Biophysics*, 5:115–133, 1943.

There is a very nice motivation for neural network-based learning in:

- V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA, USA, 1984.

If you want to know more about the history of neural networks, 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.
- B. Widrow and M.A. Lehr. 30 years of adaptive neural networks: Perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

Textbooks that cover the same material, although from different viewpoints, include:

- Chapter 5 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.
- Sections 3.1–3.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.

PRACTICE QUESTIONS

Problem 3.1 Consider a neuron with 2 inputs, 1 output, and a threshold activation function. If the two weights are $w_1 = 1$ and $w_2 = 1$, and the bias is $b = -1.5$, then what is the output for input $(0, 0)$? What about for inputs $(1, 0)$, $(0, 1)$, and $(1, 1)$?

Draw the discriminant function for this function, and write down its equation. Does it correspond to any particular logic gate?

Problem 3.2 Work out the Perceptrons that construct logical NOT, NAND, and NOR of their inputs.

Problem 3.3 The parity problem returns 1 if the number of inputs that are 1 is even, and 0 otherwise. Can a Perceptron learn this problem for 3 inputs? Design the network and try it.

Problem 3.4 Test out both the Perceptron and linear regressor code from the website on the parity problem.

Problem 3.5 The Perceptron code on the website is a batch update algorithm, where the whole of the dataset is fed in to find the errors, and then the weights are updated afterwards, as is discussed in Section 3.3.5. Convert the code to run as sequential updates and then compare the results of using the two versions.

Problem 3.6 Try to think of some interesting image processing tasks that cannot be performed by a Perceptron. (Hint: You need to think of tasks where looking at individual pixels isn't enough to allow classification.)

Problem 3.7 The decision boundary hyperplane found by the Perceptron has equation $y(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b = 0$. For a point \mathbf{x}' , minimise $\|\mathbf{x} - \mathbf{x}'\|^2$ to show that the shortest distance from the point to the hyperplane is $|y(\mathbf{x}')|/\|\mathbf{w}\|$.

Problem 3.8 There is a link to a very large dataset of handwritten figures on the book website (the MNIST dataset). Download it and use a Perceptron to learn about the dataset.

Problem 3.9 For the prostate data available via the website, use both the Perceptron and logistic regressor and compare the results.

Problem 3.10 In the Perceptron Convergence Theorem proof we assumed that $\|\mathbf{x}\| \leq 1$. Modify the proof so that it only assumes that $\|\mathbf{x}\| \leq R$ for some constant R .