Chapter II

An Introduction to Neural Networks

Objectives

After reading this chapter you should be able to:

- describe what neural networks are and how they are used;
- compute with simple cellular automata;
- solve a simple classification problem with a perceptron;
- understand the delta learning rule; and
- appreciate the limits of simple neural networks.

II.1 Overview

We have learned some of the procedures and notation important for working with vectors and matrices. We did this to be able to program a neural network. Before we do that, we should review what we mean by the term "neural network." What is a neural network? More importantly, what are the key ideas, at a non-mathematical level, which motivate this type of work? We will examine these two questions in this chapter and work towards a simple spreadsheet-based implementation.

II.2 What Are Neural Networks?

The answer to this question depends on context and scientific specialty. In biology, a neural network might well refer to a collection of real, physical neurons. For example, one could grow a collection of nerve cells on a microscope slide and see if they established synapses. If so, then they would have formed a network, and you could study its properties. In the computational literature a neural network usually refers to a collection of simple processing elements that are "wired" together in some way. Here the term "neural" is used to emphasize the neurobiological inspiration, but the connection to neurobiology may be quite remote. The term *connectionism* or *connectionist network* may be used to label a neural network that is only compared against behavioral, and not biological, data; an example of this is the interactive activation model of single word reading (McClelland & Rumelhart, 1981). The term *artificial neural network* is sometimes limited to refer to models that measure themselves against neuroscientific or neuropsychological data, and not behavior alone.

Neural networks can loosely be thought of as functions. Neural networks take in a vector or matrix of "inputs" and output a transformed version. We learned in the previous chapter that transformations of vectors may be matrices. A neural network can sometimes be thought of as a matrix, a matrix of weights. We will clarify this statement shortly. At their heart neural networks are mathematical methods for input:output mappings. In practice they usually share some subset of the following features:

1. They are inspired by biology, but are not expected to slavishly emulate biology.
2. They are implemented as computer programs.
3. They have "nodes" that play the role of neurons, collections of neurons, brain structures, or cognitive modules. Their role depends on the modeller's intent and the problem being modeled.
4. Nodes are interconnected: they have functional equivalents of "dendrites" and "axons."
5. The individual nodes produce outputs from inputs. The network's output is the collection of all these local operations.

There is not one canonical kind of neural network. Rather, there are many varieties of neural networks, all having some family resemblance according to the above criteria.

While most neural networks are inspired by some aspect of biology, many applications of neural networks are not intended to be models of biological or cognitive systems. Such neural network applications are simply meant to be practical solutions to practical questions. Their success is judged by performance and not biological verisimilitude. You might retrieve such articles when doing a literature search. It is important to remember this when determining if a particular approach is relevant for pursuing a neuroscientific or psychological question. We are usually interested in neural network approaches that emphasize neurological or cognitive phenomena and that are not simply used for a generic engineering purpose.

It may be natural at a first reading to think of the single node in a neural network as the network's representation for a neuron, but this should be reconsidered. It may be possible to regard a node in a neural network as depicting an individual neuron, and the inputs as analogues of individual dendritic subtrees, but it may be just as "correct" to think of the node as representing a population of neurons or even that it represents a cognitive module. For examples of this last sort it may not matter much if the neural network's procedures violate biology at the level of the local unit since we are not trying to match the network's node with our brain's neurons, but with some more abstract idea of neural or cognitive functioning.

11.3 Some Neural Network History

Treating neurons as computational abstractions has a history that precedes the computing age. One of the first advances was made by a collaboration between the neurophysiologist

Warren McCullough and Walter Pitts (McCulloch & Pitts, 1943). In 1943 they provided a mathematical demonstration that things like neurons, formally conceived, could compute, in principle, any computable function (see also Section 17.2 and the entry about Turing on page 164). While these results preceded the rise of the digital computer, it was with the digital computer that neural networks began to receive a wider hearing and broad application. The father of digital computing, John Von Neumann, was aware of the resemblance between digital computers and brains and of the work of McCullough and Pitts and their bipolar, on or off, formal neurons. Von Neumann discussed these relations in a series of lectures he was working on at the time of his early death from a brain tumor (Von Neumann, 1958). This work, written by one of the greatest minds in the history of humankind, is still worth reading for its ideas and insights. A brief biography of von Neumann is in the box below.

Von Neumann was explicit in discussing the connection between collections of simple off–on elements and the brain. It was Frank Rosenblatt who brought them into the heart of psychology (read Rosenblatt, 1960, for an introduction, and consult Rosenblatt, 1958, for a complete treatment). He emphasized the connection between computation and learning. Learning could be automatic and rule-based and still lead to interesting varieties of behavior. The dependence of this research on computing can be seen by looking at Rosenblatt's institutional affiliation: the Cornell Aeronautical Laboratory.

John von Neumann (1903–1957)

John von Neumann was a unique combination of intellect and *joie de vivre*, and one of a generation of great Hungarian mathematicians born in the early 1900s. He was a child prodigy who learned calculus by the age of 8 and mastered the German and Greek taught to him by a governess. He was taught in public schools and privately tutored, and published mathematical papers before he was 20. He acquired a PhD in mathematics and simultaneously, to please his father, a degree in chemical engineering. The fact that the universities from which he earned these two degrees were in two separate countries seems not to have been too much of a problem, since he obtained his PhD by age 22.

In 1930, he came to the Institute for Advanced Studies in Princeton, NJ (where Albert Einstein and Kurt Gödel were also appointed), and later became a US citizen. He continued his seminal work in pure mathematics, but also contributed to a wide range of other areas. Together with Oskar Morgenstern, he largely invented the field of game theory and made explicit game theory's relations to economics and other practical domains. He contributed to the US efforts to construct a nuclear weapon, and had time to play the *bon vivant* at frequent cocktail parties where he was a wit and good host. In addition to all this, he can be said to have invented computer science. He understood the possibility of self-reproducing machines, a simple example of which are cellular automata.

He died from brain cancer. Von Neumann was not religious, and his uncertainty in the face of death was extreme and a source of great anxiety and discomfort. Even on his death bed, his eidetic memory remained intact, and he is said to have entertained himself and his brother by reciting from memory the lines on specified pages of Göethe's *Faust*.

11.4 Global Structure from Local Interactions

Neural networks have global and local aspects. Just as the entire network produces an output for an input, so does each of the constituent nodes. The individual nodes typically have no access to the global state of the network or even information regarding what problem the network is programmed to solve. The component neurodes have only their local inputs and their individual rules as to how to process that input. The input comes into each node from other nodes within the system. The nodes send their output locally to other nodes through specified connections. In summary, a network can be thought of as a collection of little elements that each have a local rule for processing inputs to produce outputs. In addition, there is a matrix that encodes the connections between nodes in the network. The "meaning" of a network comes from outside the network. It comes from us, the users or programmers. Barring a bug in the programming, networks always give the "right" answer. We just may not like the answer. Whether the answer is *useful* is determined by us and the requirements of our application.

To rephrase, neural networks are collections of individual elements following local rules that give rise to global behavior of an interesting sort. How interesting depends on the person outside the network. An analogy is seeing the "wave" when it goes through the crowd at a football stadium. In this case, each gentleman spectator is following a local rule. He stands up when the people on one side of him stand up, and he sits down when they sit down. From his perspective the rule is local: "if these three people stand, I stand; if they sit down, I sit." The spectator receives an input, that is, they are standing or they are sitting, and he produces an output, that is, he stands or he sits. But viewed from outside this local perspective we can see a wave undulating around the perimeter of the stadium. This is the global behavior produced by the application of local rules in an interconnected network. We, looking from outside (or on the stadium's big screen), can see the global result, but it is emergent from our local actions and interactions.

11.5 Cellular Automata

My intention in this section is to give you a practical hands-on experience with the procedures of neural networks. We will observe interesting global structure produced by the labor of local, ignorant elements. We will do this by using pencil and paper to implement a simple rule-based system. The action of each cell in our network will be determined by the actions of nearby neighbors. None of these local elements will know what they are trying to produce. We can be sure of this, because you will be doing all the calculations for them and you will have no idea what the goal is, so how could a square on a sheet of graph paper?

This exercise uses cellular automata. Cellular automata have a lot in common with neural networks: local elements, local rules, connections, and global behavior. Cellular automata have been suggested as a general framework for computation and were featured in von Neumann's book on the brain.

For this exercise you will need a sheet of graph paper and a *rule*. Begin with the top row of the graph paper. Your rule will specify how the color of a square of the graph paper depends on the color of neighboring cells in the row immediately above

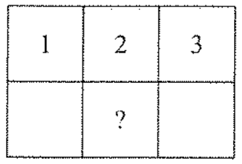


Figure 11.1 For the cellular automata classroom exercise, you decide whether a particular cell in the grid on your graph paper should be colored or not based on the three squares immediately above it and to the left and right.

(Figure 11.1). Our rule depends on three cells only and is like the gentleman at the football match deciding whether to stand or sit when making the wave. Each square in the grid of your graph paper decides whether to be blank or colored-in based on the three squares above it.

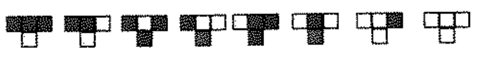
Exercise: Cellular Automata

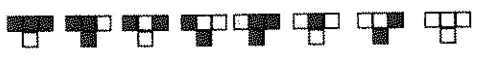
1. Pick one of the rules from the choices in Figure 11.2.
2. Color in the very center square of the very top row of the graph paper.
3. Proceeding from left to right, color every square in the second row based on the rule you selected. In this row every cell will be left uncolored except for those near the center, because these rules all specify that a cell with three uncolored grids above remains uncolored. But how you treat the ones near the center will depend on the rule. For example, for rule 60, the cell immediately beneath the center, colored, square of row 1 will be colored, as will the one immediately to its right.
4. Repeat this process working down until you can clearly see the pattern, or give up in despair.
5. Compare your results from using different rules.

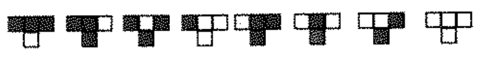
What this exercise illustrates is that remarkable global structure can emerge from the consistent application of simple rules. By analogy think of a neuron computing whether or not to spike based on the input it receives from a small set of neighbors. A spike becomes a decision, like whether or not to color a grid of the graph paper. The neuron does not need to know the global objective in order to do its job properly. The global structure takes care of itself.

How many rules are there? This is an exercise in combinatorics, the field of mathematics for computing combinations. You know how many inputs there can be: there are three cells and each can be white or gray, $2 \times 2 \times 2 = 8$. But how many output patterns are there that each rule can be matched with?

This example also emphasizes the components of many neural networks. Each grid square is a node. It took inputs, the colors of the three squares above it, and it (or rather you) computed its output: its color. The connectivity was implicit in the structure we used. Each node (grid cell) was wired to the three nodes, and only those three nodes, above it (for its input) and wired for output to the node directly beneath it and one to either


(a) Rule 60


(b) Rule 90


(c) Rule 110

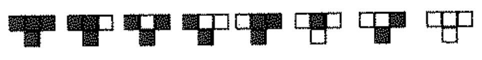

(d) Rule 250

Figure 11.2 For the cellular automata classroom exercise, you decide whether a particular cell in the grid of your graph paper should be colored or not based on the three squares immediately above it and to the left and right. Take rule 90 as an example. If all three squares on one line are colored, then the square on the next line below that falls directly underneath is uncolored. For each line, and working left to right, look at the three cells directly above, one to the left, and one to the right. Then, based on the patterns in this figure, color in, or leave white, the cell you are working on. Then move one cell to the right, and repeat. At the end of each line, move over to the far left, drop down one line, and repeat.

side. If we change the connections, we change the network. If we change the rule for coloring, we change the network. There is nothing special or necessary about choosing to use three cells. In fact, von Neumann used a different, more complex architecture. Some mathematicians argue that such simple programs can give rise to all of our complex world (Wolfram, 2002).

11.6 The Perceptron

Although the cellular automata example captures many of the features of a neural network, it is not typically what people mean when they refer to a neural network. To begin our programming of a neural network, we will start with one of the oldest and simplest examples: Rosenblatt's *perceptron*.*

Rosenblatt took the rules that we have already seen, and added an important amendment. He provided a way for the elements to change or learn. He expanded the purview of the automatic local rules to include instructions for altering the output of a node based on experience. This idea was extended for many other types of networks. Often the focus is on the connection strengths between nodes. This is true of Rosenblatt's network too, but we must see the connections as existing between a layer of nodes that communicate the inputs from the outside world into our perceptron

Neural network learning mechanisms divide neural networks into two classes: supervised and unsupervised. In *supervised* learning there is a right answer. After each run of the network, the performance of the network is assessed by comparing the output of the

*Named, as you might guess, to suggest a combination of automaton and perception.

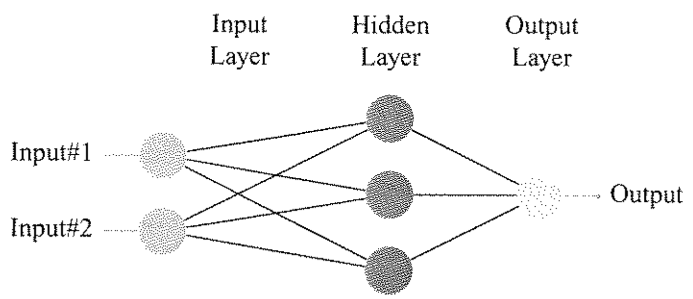


Figure 11.3 A simple neural network architecture. Two input units pass the data along to each of the three intermediate, "Hidden", nodes. In turn, after processing their input, each of the three hidden nodes sends its output to the output unit. This unit computes the final output of the network. For a supervised network this output is compared to the desired output and used to update the connection strengths between nodes.

network to the correct output. This comparison results in some feedback to the network that adjusts its future output to the same input so that it more closely approximates the correct answer. This adjustment procedure may be automated, but the idea of supervision invokes something external to the network that "knows" what the network is supposed to output and teaches it, as a teacher supervises and instructs a classroom. *Unsupervised* learning algorithms adjust future network outputs via an automated method that does not require the network to be given access to the true, correct answers. How this might work will be seen in the next chapter, which covers Hopfield networks (Chapter 13).

The Connection Between Neural Networks and Vectors and Matrices

It is common to conceptualize a neural network as a graphical-like structure or even as a physical "thing," some collage of balls and strings (Figure 11.3). While this can be useful for planning purposes, the actual programming of a network benefits from thinking of the actual computations. Neural networks are ultimately just numbers and equations. And those numbers and equations live in the world of linear algebra.

For the graphical structure in Figure 11.3 the input would be two numbers. We can represent these two numbers as a 2-D vector, for example,

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

where each of the input nodes connects to each of the hidden nodes. We imagine a sort of gate that filters the input to the hidden node. This is called a weight. How do we represent the weights that regulate the strength of each input node's connection to each hidden unit? Remember that we described a neural network as transforming input vectors to new outputs. The output of the hidden layer will be, in this case, a 3-D vector. What transforms vectors? Matrices. The weights are represented as a weight matrix. For example,

$$\begin{bmatrix} 0.5 & 0.2 \\ -0.3 & 0.7 \\ 0.01 & -1.1 \end{bmatrix}$$

These numbers are not special. I just made them up to make the example concrete. Since we are dealing with vectors and matrices, we have to think about whether a vector is a row or a column of the matrix. Since we learned that the order of multiplication is important, we also have to decide if the weight matrix will be to the left of the vector or to its right. These are conventions we decide at the outset when programming our network in the language of linear algebra.

In this example, I have set each row of the weight matrix so that each hidden unit is a row, and the two numbers in each row represent the weight of its connection to the top and bottom input units. Each of the input units connects to each of the hidden units. For example, the element in the second row, first column represents the weight connecting our first input to the second hidden unit. To find the input activation for a hidden node we multiply the matrix (on the left) against the column vector (on the right). (Can you figure out the answer? It is in the note in the margin.) This column vector has one element for each of the three hidden nodes. By chaining operations like these together, we can have a compact representation that would take a lot of space to present graphically. Also, this gives us clear guidance on how to program a neural network.

Neural Networks and Geometry

The answer to the question about the activity of the hidden units is: 0.5, −0.3, 0.01?

In the prior chapter, I emphasized the value of thinking of a vector as a geometric object, an arrow with direction and length, and to think of matrices as transformations. This geometric orientation becomes useful for understanding the qualitative behavior of neural networks. This type of thinking, though, is easier said than done, and it is a challenge in the beginning. If we succeed, however, it means we will visualize our inputs as little arrows pointing in some direction in a space. Similarly, we will see the weight matrix as a way to move our vector around in space. The repeated application of weights to inputs will create a trajectory: a path of migration for our vector in space. We can visualize the learning process as moving one or another of the components of our network through this space. Before we get too high level, though, it is always good to make sure we understand how to perform the basic operations. We begin our work with the perceptron using a pencil and paper example:

Perceptron Learning Rule

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

$$y = \begin{cases} +1, & \text{if } I \geq T \\ -1, & \text{if } I < T \end{cases}$$

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \beta \mathbf{y}\mathbf{x}$$

$$\beta = \begin{cases} +1, & \text{if answer correct} \\ -1, & \text{if answer incorrect} \end{cases}$$

Table 11.1 Data for perceptron classroom exercise (adapted from Caudill & Butler, 1992).

Class	Input 1	Input 2	Correct output
A	0.3	0.7	1
B	−0.6	0.3	−1
A	0.7	0.3	1
B	−0.2	−0.8	−1

I will try to explain the rule in words. Between the equations and the description you should be able to puzzle it out. The "sigma" is the sign to add up things. The things to be added up are specified by the sub- and superscripts of the $\sum$ sign. The rule is for a single perceptron. The weight connecting it to the input is multiplied against the value of the input and all are summed up. This is like a dot product. Then this total is compared to a threshold, T, and the output of the perceptron is determined by whether it is above the threshold. We could make the output binary (1 or 0), but it is more convenient here to use the bipolar system of 1 or −1. The threshold is ours to choose, but we will often use zero. Next, we compare the output of our perceptron to what it should have been. This makes the learning rule a supervised learning rule. The value of the weight of the perceptron gets adjusted up or down depending on what we wanted, what we got, and what weight we were using before. From a repetitive application of this rule we will home in on the correct answer.

This rule highlights another characteristic of many neural network procedures. The processing of the nodes is typically a two-stage process. First, each node computes the net input it receives. This is the matrix operation of multiplying its weight vector against the input vector to get a weighted sum of the inputs. This calculation of the input activation is then followed by a *non-linearity*. This is a function that takes the net input and converts it to an activation that becomes the nodes' output. It is called a non-linearity because the plot of how inputs becomes activations would not generate the graph of a line. In this case all the values less than zero on the x axis would be −1 and then there would be a straight line up to +1 at T. Such a function is called a "step function."

Our example problem will use a single perceptron. Admittedly, a network of one unit is a very small neural network, but when tackling a new computational topic it is often advisable to start with the simplest example one can think of, and scale up as the problem requires and as one's understanding improves. If it helps our pride, we can think of this example as a two-layer network in which the first layer is the layer of units that translate from the outside world into the language of our perceptron. These are the input units.

Exercise: Manual Calculation of a Perceptron

Using the following inputs (Table 11.1) and the matching answers, train a single perceptron to solve this classification problem starting with weights = $(-0.6, 0.8)$.

This exercise can be a bit of challenge to get started with. Here is the first step worked through:

$$\vec{\mathbf{w}} \cdot \mathbf{A}_1^{\mathrm{T}} = -0.6 \times 0.3 + 0.8 \times 0.7 = -0.18 + 0.56 = 0.38$$

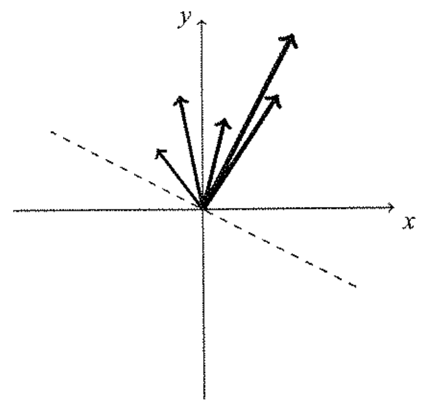


Figure 11.4 Interpreting the perceptron's weight vector. This plot uses the two elements of the weight vector as the (x,y) pair for points on a Cartesian plane. At each iteration of the perceptron learning process, the line gets thicker, and you can see how its direction changes. This allows you to exercise your geometric interpretation of vectors as arrows pointing in a space. The dashed line shows the perpendicular to the final weight vector. This is the decision plane. Every point on the same side as the weight vector will yield a positive number when multiplied against the weights, and every point on the other side will yield a negative number. The basis for this fact lies in trigonometry. There is a relation between the dot product of two vectors and the cosine of the angle between them (see Equation 9.2; page 74).

Since $I = 0.38$ is greater than or equal to zero, $y = +1$ and $\beta = +1$ the answer is correct, that is, we observed the desired output listed in our table. To update the weights we apply the perceptron rule:

$$[-0.6, 0.8] + (+1)(+1)[0.3, 0.7] = [-0.3, 1.5]$$

After doing the same for each of the three remaining patterns, test to see if the final weights correctly classify the four test patterns into their two respective groups (they should). You perform this by deriving y and comparing it to the desired output without further changes to the weight vector (Figure 11.4).

11.7 Another Learning Rule: The Delta Rule

The different types of neural networks can vary along a number of different dimensions. One way in which different flavors of neural networks come about is to change the learning rule the units use. The delta rule is an example of this. It makes a more elaborate use of Perceptron-like nodes. The delta rule is an example of using an *error signal*. An error signal is the amount that we are in error when generating the output. We compute the output our neural network unit gives us and subtract this from the output we wanted, and this difference is our error. We update our weights based on this value. To some degree, the use of the delta rule is similar to a linear regression model. In linear regression we weight the independent variables and sum them together to get an estimate

of our dependent variable. With the delta rule we are finding the right weight for our inputs so as to produce the proper output. This analogy is not simply conceptual.

Because we are using the data to train the network, we risk the problem of over-fitting. We might train our neural network to solve a problem using a particular set of training data, and it might be perfect, but all we might have done is succeed in training it to detect some random variation in the particular examples we have selected for training. What we want to do is to use our neural network to extract some general characteristics of the data that distinguish two classes of inputs, for all examples of those input classes, not just the particular examples we trained on.

To control for this risk, it is common practice to have both a training set and a validation set. If data are not abundant, one can take the available data and arbitrarily divide these into two portions: one for testing and one for validation.

We will explore the use of both the validation samples and the delta rule in this next example.

Exercise: Delta Rule, Testing, and Validation

This exercise has several steps and will require patience on your part. It may be easier to follow the steps if you first have an overall view of the exercise's goals. I want you to develop the idea of *surrogate* data: data created by a procedure that matches the process of interest. I want you to develop test and validation samples. I want you to use random numbers to generate your data rather than hand-coding a particular example. This allows you to run the same process over and over, to test for consistency. And I want you to train a simple neural network classifier.

For these goals we will calculate a line of random slope and intercept. This line will be the "true" division between our two classes of points. We will shift some points of the line up and others down to create our two stimulus classes. We will use ten of this set (five examples from each class: the ups and the downs) to train and another ten (again five of each) to validate.

1. Use the random function in your spreadsheet program to generate a random slope and intercept for a line (e.g., `=rand()`). After doing this, you will need to copy and paste *values* to keep the spreadsheet from changing the random values every time you update the spreadsheet.
2. Use the random function to generate 20 random x values. You might wish to scale these numbers to be from −20 to 20 (or something similar, e.g., `=rand() * 40 - 20`).
3. Use the equation of a line ($y = mx + b$, substituting your random slope and intercept from above) to calculate the location on the line for each of your randomly selected x positions.
4. Again, use the random function, and remembering to cut and paste *values* shift half the y values above the line by a random amount, and the other half below by a random amount.

(Continued)

(Continued)

5. Create a column that will code the classes of your points: +1 for above the line, and −1 for below the line.
6. Use the first five examples of each set as the training set, holding the other ten in reserve for validation.
7. The neurode you will be building will have weights for both inputs and *a bias term*. Therefore, in addition to your two inputs x and y you will have a third input called the bias that also will have a weight. You can set the bias to 1 for each pattern.
8. To compute the activation of your unit multiply the three inputs for a pattern (x, y, and bias) against three, initially random, weights (w1,w2,w3), and sum the results. Set a threshold of 0 and code the outputs as greater than or equal to the threshold or less than the threshold.
9. Use the delta rule (Equation 11.1) for updating the weight vector after each pattern.
10. You use only a single, 3-D weight vector for your classifier. Those three weights change after it has seen each training pattern.
11. Continue to loop through all the training patterns until the neurode classifies all the training patterns correctly.
12. Then "freeze" the weights. This means they no longer change. Test how well your network did on classifying each of the validation patterns that you kept in reserve.

$$\Delta w_i = x_i \eta (\text{desired} - \text{observed}) \tag{11.1}$$

How do we know that the delta rule guarantees that the weights will stop changing if the perceptron matches all the inputs to their correct answers?

What the delta rule says in words is that we take the value we should have gotten (assume we are training on a class +1 pattern), and subtract what we got, say 0, from it. Then we multiply this error signal by a learning rate called η. η (pronounced "eta" not "nu") that is usually a small number, say 0.1. We multiply the result against the input. Since our input has multiple elements we need to keep track of which one we are using, hence the subscript i. We multiply all this together to get the amount that we change the weight for that input. All of this can be done in a spreadsheet. If you know how to use macros, you can make repeating the steps easier. In an intermezzo (Chapter 12), I show you how this can be programmed using Octave.

Additional Explorations

Once you have the basic method implemented, examine how the accuracy of your classification of the validation patterns improves if you put the training points close to the decision line, or far away. How does this interact with the number of training patterns? Plot the line that is at a 90 degree angle to the final weight vector and see how closely

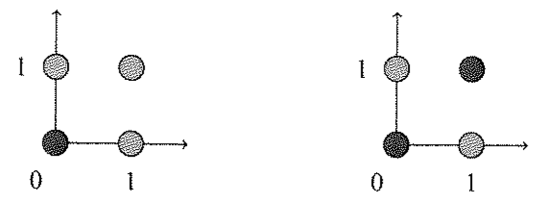


Figure 11.5 Linear separability. On the left we have a schematic illustration of the OR function. For each of two items, think apples and oranges, we have two possibilities: we have them or we do not. If not, we enter a zero; if so, a one. Each point corresponds to one of the situations. If a point is gray it means that the statement that we have apples OR oranges is true. On the right we illustrate the XOR Boolean operation. It is only true if we have apples or oranges, but not both (this makes it exclusively OR). On this side it is impossible to draw a single line that separates the black and gray dots.

it tracks the true answer (which we know to be the random line you used to generate the division between the two classes of data). Why isn't the fit perfect? An additional exploration would be to look at the similarity between perceptrons and *support vector machines* (Collobert & Bengio, 2004). Support vector machines (SVMs) are a popular method for classifying and clustering data that relies on finding the right vectors to properly partition a space with a large number of vectors from more than one class. SVMs are widely used in machine learning applications, and would make an excellent topic for extending the ideas learned in our perceptron section.

11.8 Why Aren't Perceptrons the Only Type of Neural Networks?

To get the answer to this question, it helps to first consider a simpler one: Why did we use a line as the basis for creating our two populations in the last exercise? When we use a line to create the border between two classes, the classes become *linearly separable*. Linear separability is very important in the field of neural networks. Figure 11.5 demonstrates linearly separability.

When we use a perceptron-like neural network unit, the weight vector of that unit creates a virtual decision plane at a 90 degree angle to itself. All points on one side are above a zero threshold, and all values on the other side are below a zero threshold. Because of this, a perceptron can only correctly classify data that are able to be separated by some plane running in some direction. Perceptrons are limited to solving linearly separable problems.

In a famous book, *Perceptrons*, Minsky and Papert (1969) demonstrated that perceptrons were limited to this class of linearly separable problems. What is more, they showed that many apparently easy problems were not linearly separable. The *XOR* problem, illustrated on the right of Figure 11.5, is one example of a non-linearly separable problem. Because it was felt that interesting problems were likely to be non-linearly separable, a cold wind blew over the field of neural networks and limited the amount of research and the interest in neural networks for nearly 20 years.

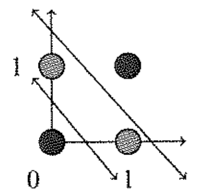


Figure 11.6 Multi-layer perceptron. This is the XOR again. Note that these two lines isolate the gray points between them. If we had an OR perceptron to classify all points above the lower line, and another perceptron (what would it represent in Boolean terminology (see Chapter 17 for more on Boolean functions)?) to classify the points that fell below it, we could feed the output of each to another perceptron that would classify points that were above line 1 AND below line 2, and we would have solved the XOR problem.

The way to overcome this limitation of single layer perceptron networks is to develop multi-layer perceptron networks. While a single perceptron layer is limited to a single decision plane, multiple layers can use multiple planes, and the combinations of these planes can effectively carve the space into the compartments needed to solve many non-linearly separable problems.

Figure 11.6 illustrates how this works. We use multiple perceptrons to create multiple subsets of our data. Then we test for combinations of those divisions with yet other perceptrons to find the appropriate set of conjunctions. The construction of multi-layered perceptrons makes the problem-solving capabilities of perceptrons more powerful and more general. However, their practical utility is limited by the fact that we have to know how to train each of the subsets. We need to know the correct mappings in order to build the correct separations.

To test your understanding, you can try your hand at solving the XOR problem manually. If you have a good understanding of the decision planes needed, then with a little trial and error you could manually discover sets of weights sufficient for a three-perceptron network (two in layer 1, and one in layer 2) to be able to solve all the inputs of the XOR problem.

11.9 Summary

Neural networks are a fascinating addition to the computational toolkit of neuroscientists and psychologists. They can be used for very practical problems, such as discovering clusters in data or sub-categorizing complex data sets, and as we will see in the next chapter, they can probe in highly original ways basic cognitive functions. They may point the way for discovering, at a functional level, principles of cognitive and neural operations.

However, we have only scratched the surface by beginning with one of the earliest and most basic examples of neural networks. Still, it is sufficient to demonstrate some of the advantages and disadvantages of this approach. The advantages include that fact that it provides a concrete demonstration that simple rules alone, when appropriately connected and trained, can produce complex behavior. A proof of concept, if you will, that maybe all we are is neurons. Practical disadvantages include the fact that actual simulations on interesting problems can take a *long* time. Further, although a particular network may solve a problem, the "reasons" why a network works may be opaque since it is, in a sense, contained in the pattern of weights (connection strengths) between units. An additional disadvantage is that if we are not careful we can make networks that are too good. They succeed because we have over-fit and are trained on the noise in our data. Such networks will not generalize.

In the next chapter we develop the Hopfield network. This is a network that works through its interconnections. It has the ability to remember and to correct errors, and it is an example of how interdisciplinary research expands understanding. The Hopfield network shows how the tools of mathematical physics can be brought to psychology, and the result is an ability to prove things with a definiteness that had eluded all workers since Fechner.

Computational Neuroscience and Cognitive Modelling

a student's introduction to methods and procedures

Britt Anderson

SAGE

Los Angeles | London | New Delhi
Singapore | Washington DC

Ak 1355

SAGE

Los Angeles | London | New Delhi
Singapore | Washington DC

SAGE Publications Ltd
1 Oliver's Yard
55 City Road
London EC1Y 1SP

SAGE Publications Inc.
2455 Teller Road
Thousand Oaks, California 91320

SAGE Publications India Pvt Ltd
B 1/I 1 Mohan Cooperative Industrial Area
Mathura Road
New Delhi 110 044

SAGE Publications Asia-Pacific Pte Ltd
3 Church Street
#10-04 Samsung Hub
Singapore 049483

Editor: Michael Carmichael
Editorial assistant: Keri Dicken
Production editor: Imogen Roome
Copyeditor: Neville Hankins
Proofreader: Audrey Scriven
Marketing manager: Alison Borg
Cover design: Wendy Scott

Library of Congress Control Number: 2013940255

British Library Cataloguing in Publication data

A catalogue record for this book is available from the British Library

ISBN 978-1-4462-4929-1
ISBN 978-1-4462-4930-7

Contents

Preface x

1 Introduction **1**
Objectives 1
1.1 Overview 1
1.2 Why Model? 2
1.3 Can We Make A Brain? 2
1.4 Computational Models as Experiments 3
1.5 Do Models Need to Be Biologically Plausible? 5
1.6 Do Computational Models Require Computers? 6
1.7 How Do You Evaluate a Model? 7
1.8 Do Models Need Assumptions? 8
1.9 Additional Questions 8
1.10 Road Map 9

I Modelling Neurons **11**

2 What Is a Differential Equation? **13**
Objectives 13
2.1 Overview 13
2.2 Modelling Individual Neurons 14
2.3 Differential Equations: The Mathematical Method of the Hodgkin and Huxley Model 14
2.4 The D in DE 16
2.5 Summary 21

3 Numerical Application of a Differential Equation **23**
Objectives 23
3.1 Overview 23
3.2 First, We Assume a Spherical Chicken 23
3.3 Numerical Simulations 26
3.4 Numerical Integration and Differential Equations: Euler's Method 28
3.5 Solving the Spring Differential Equation 30
3.6 Summary 31