Artificial Neural Networks and Information Theory

Colin Fyfe,
Department of Computing and Information Systems,
The University of Paisley.
Room H242
Phone 848 3305.

Edition 1.2,
2000

Contents

Chapter 1

Introduction

1.1 Objectives

After this chapter, you should

1. understand the basic building blocks of artificial neural networks (ANNs)
2. understand the two modes of operation in ANNs
3. understand the importance of learning in ANNs
4. be able to use a simple rule to create learning in an ANN
5. begin to understand the importance of linearly inseparable problems
6. know some of the problems on which ANNs have been used.

1.2 Intelligence

This course comprises an advanced course to those new information processing simulations which are intended to emulate the information processors which we find in biology.

Traditional artificial intelligence is based on high-level symbol processing i.e. logic programming, expert systems, semantic nets etc all rely on there being in existence some high level representation of knowledge which can be manipulated by using some type of formal syntax manipulation scheme - the rules of a grammar. Such approaches have proved to be very successful in emulating human prowess in a number of fields e.g.

- we now have software which can play chess at Grand Master level
- we can match professional expertise in medecine or the law using expert systems
- we have software which can create mathematical proofs for solving complex mathematical problems.

Yet there are still areas of human expertise which we are unable to mimic using software e.g. our machines have difficulty reliably reading human handwriting, recognising human faces or exhibiting common sense. Notice how low-level the last list seems compared to the list of achievements: it has been said that the difficult things have proved easy to program whereas the easy things have proved difficult.

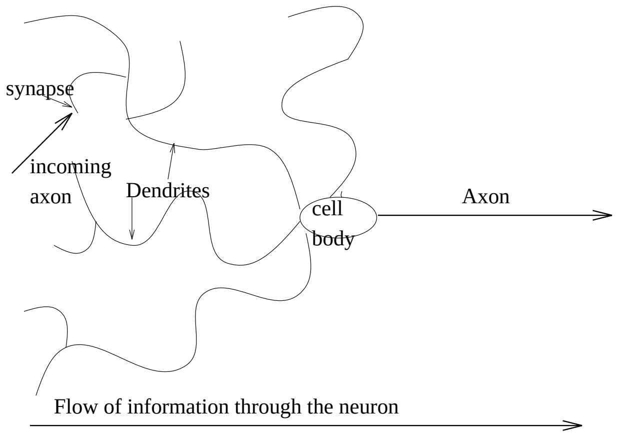


Figure 1.1: A simplified neuron

1.3 Artificial Neural Networks

Now tasks such as those discussed above seemingly require no great human expertise - young children are adept at many of these tasks. This explains the underlying presumption of creating artificial neural networks (ANNs): that the expertise which we show in this area is due to the nature of the hardware on which our brains run. Therefore if we are to emulate biological proficiencies in these areas we must base our machines on hardware (or simulations of such hardware) which seems to be a silicon equivalent to that found within our heads.

First we should be clear about what the attractive properties of human neural information processing are. They may be described as :

- Biological information processing is robust and fault-tolerant: early on in life[1], we have our greatest number of neurons yet though we daily lose many thousands of neurons we continue to function for many years without an associated deterioration in our capabilities
- Biological information processors are flexible: we do not require to be reprogrammed when we go into a new environment; we adapt to the new environment, i.e. we learn.
- We can handle fuzzy, probabilistic, noisy and inconsistant data in a way that is possible with computer programs but only with a great deal of sophisticated programming and then only when the context of such data has been analysed in detail. Contrast this with our innate ability to handle uncertainty.
- The machine which is performing these functions is highly parallel, small, compact and dissipates little power.

We shall therefore begin our investigation of these properties with a look at the biological machine we shall be emulating.

1.4 Biological and Silicon Neurons

A simplified neuron is shown in Figure 1.1. Information is received by the neuron at synapses on its dendrites. Each synapse represents the junction of an incoming axon from another neuron with a dendrite of the neuron represented in the figure; an electro-chemical transmission occurs at the synapse which allows information to be transmitted from one neuron to the next. The information is then transmitted along the dendrites till it reaches the cell body where a summation of the electrical impulses reaching the body takes place and some function of this sum is performed. If

[1]Actually several weeks before birth

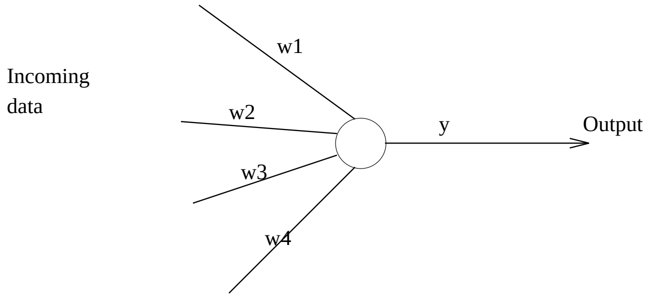


Figure 1.2: The artificial neuron. The weights model the synaptic efficiencies. Some form of processing not specified in the diagram will take place within the cell body.

this function is greater than a particular threshold the neuron will fire: this means that it will send a signal (in the form of a wave of ionisation) along its axon in order to communicate with other neurons. In this way, information is passed from one part of the network of neurons to another. It is crucial to recognise that synapses are thought to have different efficiencies and that these efficiencies change during the neuron's lifetime. We will return to this feature when we discuss learning.

We generally model the biological neuron as shown in Figure 1.2. The inputs are represented by the input vector **x** and the synapses' efficiencies are modelled by a weight vector **w**. Therefore the single output value of this neuron is given by

$$y = f(\sum_i w_i x_i) = f(\mathbf{w}.\mathbf{x}) = f(\mathbf{w}^T\mathbf{x}) \tag{1.1}$$

You will meet all 3 ways of representing the operation of summing the weighted inputs. Sometimes f() will be the identity function i.e. f(**x**)=**x**. Notice that if the weight between two neurons is positive, the input neuron's effect may be described as excitatory; if the weight between two neurons is negative, the input neuron's effect may be described as inhibitory.

Consider again Figure 1.2. Let $w_1 = 1, w_2 = 2, w_3 = -3$ and $w_4 = 3$ and let the activation function, f(), be the Heaviside (threshold) function such that

$$f(t) = \begin{cases} 1 & \text{if } t > 0 \\ 0 & \text{if } t \leq 0 \end{cases} \tag{1.2}$$

Now if the input vector $\mathbf{x} = (x_1, x_2, x_3, x_4)$ = (1,2,1,2). Then the activation of the neuron is **w.x** = $\sum_j w_j x_j$ = 1*1 + 2*2 + 1*(-3) + 2*3 = 8 and so y = f(8) =1. However if the input vector is (3,1,2,0), then the activation is 3*1 + 1*2 + 2*(-3) + 0*3 = -1 and so y = f(-1) =0.

Therefore we can see that the single neuron is an extremely simple processing unit. The power of neural networks is believed to come from the accumulated power of adding many of these simple processing units together - i.e. we throw lots of simple and robust power at a problem. Again we may be thought to be emulating nature, as the typical human has several hundred billion neurons. We will often imagine the neurons to be acting in concert in layers such as in Figure 1.3.

In this figure, we have a set of inputs (the input vector, **x**) entering the network from the left-hand side and being propagated through the network via the weights till the activation reaches the output layer. The middle layer is known as the hidden layer as it is invisible from outwith the net: we may not affect its activation directly.

1.5 Learning in Artificial Neural Networks

There are two modes in artificial neural networks:

1. activation transfer mode when activation is transmitted throughout the network

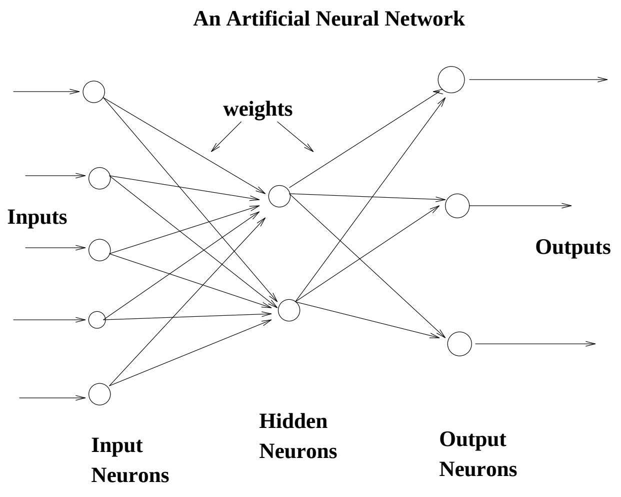


Figure 1.3: A typical artificial neural network consisting of 3 layers of neurons and 2 conecting layers of weights

2. learning mode when the network organises usually on the basis of the most recent activation transfer.

We will now consider the learning mode.

We stated that neural networks need not be programmed when they encounter novel environments. Yet their behaviour changes in order to adapt to the new environment. Such behavioural changes are due to changes in the weights in the network. We call the changes in weights in a neural network learning. The changes in weights in an artificial neural network are intended to model the changing synaptic efficiencies in real neural networks: it is believed that our learning is due to changes in the efficiency with which synapses pass information between neurons.

There are 3 main types of learning in a neural network:

Supervised learning: with this type of learning, we provide the network with input data and the correct answer i.e. what output we wish to receive given that input data. The input data is propagated forward through the network till activation reaches the output neurons. We can then compare the answer which the network has calculated with that which we wished to get. If the answers agree, we need make no change to the network; if, however, the answer which the network is giving is different from that which we wished then we adjust the weights to ensure that the network is more likely to give the correct answer in future if it is again presented with the same (or similar) input data. This weight adjustment scheme is known as supervised learning or learning with a teacher. The tutorial at the end of this chapter gives an example of supervised learning.

Unsupervised learning: with this type of learning, we only provide the network with the input data. The network is required to self-organise (i.e. to teach itself) depending on some structure in the input data. Typically this structure may be some form of redundancy in the input data or clusters in the data.

Reinforcement learning: is a half-way house between these two: we provide the network with the input data and propagate the activation forward but only tell the network if it has produced a right or a wrong answer. If it has produced a wrong answer some adjustment of the weights is done so that a right answer is more likely in future presentations of that particular piece of input data.

For many problems, the interesting facet of learning is not just that the input patterns may be learned/classified/identified precisely but that this learning has the capacity to generalise. That is, while learning will take place on a set of *training patterns* an important property of the learning is that the network can generalise its results on a set of *test patterns* which it has not seen during learning. One of the important consequences here is that there is a danger of overlearning a set of training patterns so that new patterns which are not part of the training set are not properly classified.

For much of this course we will concentrate on unsupervised learning; the major exceptions occur in Chapters 5 and 8 in which we will use supervised learning methods.

1.6 Typical Problem Areas

The number of application areas in which artificial neural networks are used is growing daily. Here we simply produce a few representative types of problems on which neural networks have been used

Pattern completion: ANNs can be trained on sets of visual patterns represented by pixel values. If subsequently, a part of an individual pattern (or a noisy pattern) is presented to the network, we can allow the network's activation to propagate through the network till it converges to the original (memorised) visual pattern. The network is acting like a content-addressable memory. Typically such networks have a recurrent (feedback as opposed to a feedforward) aspect to their activation passing. You will sometimes see this described as a network's *topology.*

Classification: An early example of this type of network was trained to differentiate between male and female faces. It is actually very difficult to create an algorithm to do so yet an ANN has been shown to have near-human capacity to do so.

Optimisation: It is notoriously difficult to find algorithms for solving optimisation problems. A famous optimisation problem is the Travelling Salesman Problem in which a salesman must travel to each of a number of cities, visiting each one once and only once in an optimal (i.e. least distance or least cost) route. There are several types of neural networks which have been shown to converge to 'good-enough' solutions to this problem i.e. solutions which may not be globally optimal but can be shown to be close to the global optimum for any given set of parameters.

Feature detection: An early example of this is the phoneme producing feature map of Kohonen: the network is provided with a set of inputs and must learn to pronounce the words; in doing so, it must identify a set of features which are important in phoneme production.

Data compression: There are many ANNs which have been shown to be capable of representing input data in a compressed format losing as little of the information as possible; for example, in image compression we may show that a great deal of the information given by a pixel to pixel representation of the data is redundant and a more compact representation of the image can be found by ANNs.

Approximation: Given examples of an input to output mapping, a neural network can be trained to approximate the mapping so that a future input will give approximately the correct answer i.e. the answer which the mapping should give.

Association: We may associate a particular input with a particular output so that given the same (or similar) output again, the network will give the same (or a similar) output again.

Prediction: This task may be stated as: given a set of previous examples from a time series, such as a set of closing prices for the FTSE, to predict the next (future) sample.

Control: For example to control the movement of a robot arm (or truck, or any non-linear process) to learn what inputs (actions) will have the correct outputs (results).

1.7 A short history of ANNs

The history of ANNs started as long ago as 1943 when McCullogh and Pitts showed that simple neuron-like building blocks were capable of performing all possible logic operations. At that time too, Von Neumann and Turing discussed interesting aspects of the statistical and robust nature of brain-like information processing, but it was only in the 1950s that actual hardware implementations of such networks began to be produced. The most enthusiastic proponent of the new learning machines was Frank Rosenblatt who invented the *perceptron* machine, which was able to perform simple pattern classification.

However, it became apparant that the new learning machines were incapable of solving certain problems and in 1969 Minsky and Papert wrote a definitive treatise, 'Perceptrons', which clearly demonstrated that the networks of that time had limitations which could not be transcended. The core of the argument against networks of that time may be found in their inability to solve XOR problems (see Chapter 5). Enthusiasm for ANNs decreased till the mid '80s when first John Hopfield, a physicist, analysed a particular class of ANNs and proved that they had powerful pattern completion properties and then in 1986 the subject really took off when Rumelhart, McClelland and the PDP Group rediscovered powerful learning rules which transcended the limitations discussed by Minsky and Papert.

1.8 A First Tutorial

This tutorial will emphasise learning in neural nets. Recall that learning is believed to happen at the synapses (the meeting points between neurons) and that we model synaptic efficiency with weights.

You are going to hand simulate a simple neural net (see Figure 1.4) performing classification according to the AND (see table) OR and XOR rules. We will use a network with three input neurons - the two required for the input values and a third neuron known as the bias neuron. The bias always fires 1 (i.e. is constant).

You will work in groups of 3 - one person in charge of selecting the input, one in charge of calculating the feedforward value of the neuron, and one person in charge of calculating the change in weights.

To begin with, the group selects random (between 0 and 1) values for the three weights shown in the figure.

1. The INPUTER selects randomly from the set of patterns shown in the table and places the cards in the appropriate places on the table.

2. The FEEDFORWARDER multiplies the weights by the input patterns to calculate the output.

$$y = \sum_{i=0}^{2} w_i x_i \tag{1.3}$$

   Then y = 1 if $y > 0$, y = -1 if $y < 0$.

3. The WEIGHTCHANGER changes the weights *when the value of y is not the same as the target* y_T according to the formula

$$w_i = w_i + \eta * (y_T - y) * x_i \tag{1.4}$$

Steps (1)-(3) are repeated as often as necessary.

1.8.1 Worked Example

Let us have initial values $w_0 = 0.5, w_1 = 0.3, w_2 = 0.7$ and let η be 0.1.

Bias	first input	second input	target output
1	1	1	1
1	1	-1	-1
1	-1	1	-1
1	-1	-1	-1

Table 1.1: The values for the AND patterns

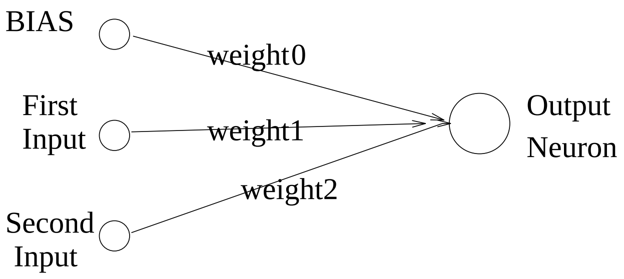


Figure 1.4: The Simple Neural Network

1. "Randomly" select pattern 1. The FEEDFORWARDER calculates y = 0.5 +0.3 +0.7 =1.5. So y=1 which is the same as the target and so the WEIGHTCHANGER does nothing.

2. Imagine pattern 2 is chosen. The FEEDFORWARDER calculates y = 0.5+0.3 -0.7 = 0.1. So y=1. Now y_T = -1 and so WEIGHTCHANGER calculates

$$\begin{aligned} w_0 &= w_0 + 0.1 * (-2) * 1 = 0.5 - 0.2 = 0.3 \\ w_1 &= w_1 + 0.1 * (-2) * 1 = 0.3 - 0.2 = 0.1 \\ w_2 &= w_2 + 0.1 * (-2) * (-1) = 0.7 + 0.2 = 0.9 \end{aligned}$$

3. Now pattern 3 is chosen. The FEEDFORWARDER calculates y = 0.3 - 0.1 + 0.9 = 1.1. So y =1 and y_T = -1 and so WEIGHTCHANGER calculates

$$\begin{aligned} w_0 &= w_0 + 0.1 * (-2) * 1 = 0.3 - 0.2 = 0.1 \\ w_1 &= w_1 + 0.1 * (-2) * (-1) = 0.1 + 0.2 = 0.3 \\ w_2 &= w_2 + 0.1 * (-2) * 1 = 0.9 - 0.2 = 0.7 \end{aligned}$$

4. Now pattern 4 is chosen. The FEEDFORWARDER calculates y = 0.1 - 0.3 - 0.7 = -0.9. So y = -1 and y_T = -1. Therefore the WEIGHTCHANGER does nothing.

5. Now select pattern 2. The FEEDFORWARDER calculates y = 0.1 - 0.3 + 0.7 =0.5. Then y =1 but y_T = -1. WEIGHTCHANGER calculates

$$\begin{aligned} w_0 &= w_0 + 0.1 * (-2) * 1 = 0.1 - 0.2 = -0.1 \\ w_1 &= w_1 + 0.1 * (-2) * (-1) = 0.3 + 0.2 = 0.5 \\ w_2 &= w_2 + 0.1 * (-2) * 1 = 0.7 - 0.2 = 0.5 \end{aligned}$$

6. Now all of the patterns give the correct response (try it).

We can draw the solution found by using the weights as the parameters of the line $ax_1+bx_2+c = 0$. Using $a = w_1, b = w_2, c = w_0$ we get

$$0.5x_1 + 0.5x_2 - 0.1 = 0 \qquad (1.5)$$

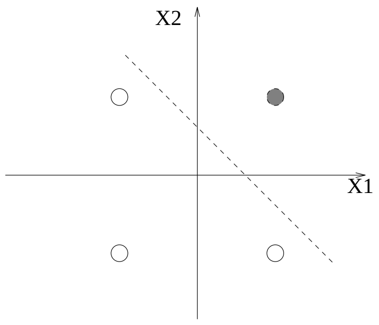


Figure 1.5: The line joining (0,0.2) and (0.2,0) cuts the 4 points into 2 sets correctly

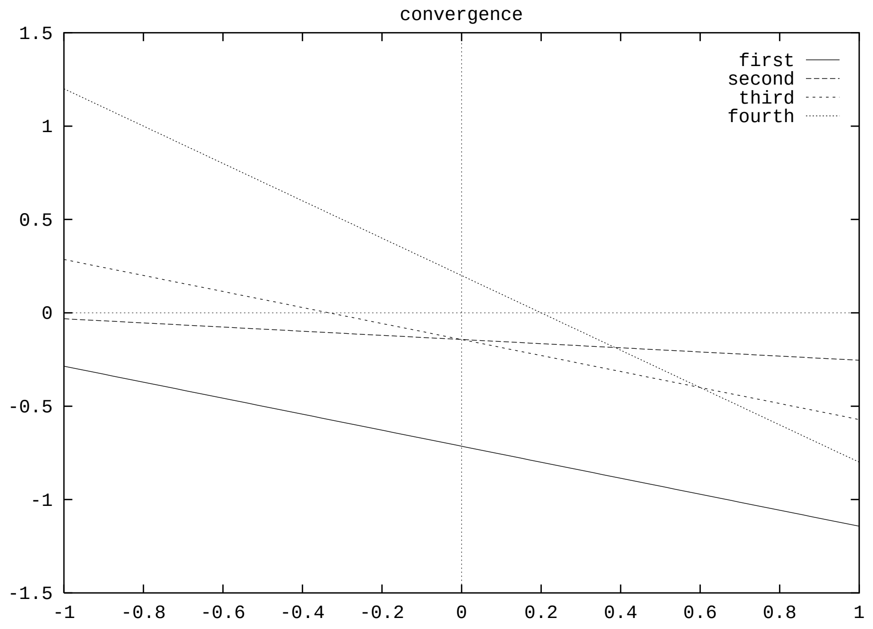


Figure 1.6: The iterative convergence of the network to a set of weights which can perform the correct mapping is shown diagrammatically here.

which we can draw by getting two points. If $x_1 = 0$, then $0.5x_2 = 0.1$ and so $x_2 = 0.2$ which gives us one point (0,0.2). Similarly we can find another point (0.2,0) which is drawn in Figure 1.5. Notice the importance of the BIAS weight: it allows a solution to be found which does not go through the origin; without a bias we would have to have a moving threshold.

We can see the convergence of $w_0 + w_1x_1 + w_2x_2 = 0$ in Figure 1.6. Notice that initially 2 patterns are correctly classified, very quickly a third is correctly classified and only on the fourth change are all 4 patterns correctly classified.

1.8.2 Exercises

1. Repeat the worked example with different initial values for the weights. (Objectives 3, 4).
2. Repeat for the OR patterns. (Objectives 3, 4).
3. Experiment with different initial values, learning rates. (Objectives 3, 4).
4. Now do the XOR problem. Don't spend too long on it - it's impossible. (Objective 5).

Nut	type A - 1	type A - 2	type A - 3	type B - 1	type B - 2	type B - 3
Length (cm)	2.2	1.5	0.6	2.3	1.3	0.3
Weight (g)	1.4	1.0	0.5	2.0	1.5	1.0

Table 1.2: The lengths and weights of six instances of two types of nuts

5. Draw the XOR coordinates and try to understand why it is impossible. (Objective 5).
6. Now we will attempt to train a network to classify the data shown in Table 1.2. Then we will train a network with the input vectors , **x**, equal to
   (1, 2.2, 1.4) with associated training output 1 (equal to class A)
   (1, 1.5, 1.0) with associated training output 1
   (1, 0.6, 0.5) with associated training output 1
   (1, 2.3, 2.0) with associated training output -1 (equal to class B)
   (1, 1.3,1.5) with associated training output -1
   (1, 0.3,1.0) with associated training output -1
   Note that the initial 1 in each case corresponds to the bias term
7. Describe in your own words an Artificial Neural Network and its operation (Objectives 1, 2).
8. Describe some typical problems which people have used ANNs to solve. Attempt to describe what features of the problems have made them attractive to ANN solutions. (Objective 6).

Chapter 2

Information Theory and Statistics

In this chapter, we review Statistics, Probability and Information Theory and then investigate the topic of Principal Component Analysis(PCA).

2.1 Probability

Probability deals with the study of random variations. We define the probability of an event which is sure to happen to be 1 and the probability of an event which will certainly not happen to be 0. All other probabilities lie between 0 and 1.

If we have a finite number of equally likely events which can happen, we can define the probability of an outcome happening to be

$$P(E) = \frac{N}{M}$$

where N is the number of events which result in the outcome happening and M is the total number of possible events.Therefore

$$P(Heads) = \frac{1}{2}$$

where Heads is the event of a coin falling down with a head face up and

$$P(King) = \frac{4}{52} = \frac{1}{13}$$

where King is the event of a pack of cards being cut at a King.

We see that if we toss two coins we have four events, HH,HT,TH and TT, so that the probability of two Heads is

$$P(H,H) = \frac{1}{4}$$

Note also that if an outcome's probability is x, then the probability of having the opposite of the outcome is 1-x. In the above, the probability of not getting two Heads is

$$P(Not(H,H)) = \frac{3}{4}$$

We write the joint probability of two outcomes, A and B, as $P(A \cap B)$. Thus if we are interested in the two outcomes :

1. We cut a King from a pack of 52 playing cards

2. We cut a spade from a pack of 52 playing cards

we can then define the joint probability

$$P(King \cap Spade) = \frac{1}{52}$$

since there is only a single card for which both outcomes are jointly possible.

We write the conditional probability of an event A given B has happened as $P(A|B)$. It can be shown that

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

If events are independent, then

$$P(A \cap B) = P(A).P(B)$$

You will notice that this happened in our example of cutting the King of Spades in which $P(King) = \frac{1}{13}$ and $P(Spade) = \frac{1}{4}$. Notice that this does not necessarily happen since e.g. $P(BlackCard) = \frac{1}{2}$ but $P(Spade \cap BlackCard) \neq \frac{1}{8}$. These events are not independent.

We are sometimes interested in the marginal distribution of a random variable which is jointly varying with another e.g. let the joint distributions vary according to the table[1]

	1	2	3	4	X
1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$	
2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$	
3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	
4	$\frac{1}{4}$	0	0	0	
y					

Then the marginal distribution of X (the distribution of X over all values of Y) is $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ while the marginal distribution of Y is $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$

2.2 Statistics

A variable whose outcome is determined by the outcome of an experiment with a random component is known as a random variable. Consider the two coins tossing experiment. Let the random variable X be the number of heads obtained. Then X can take on the values 0,1 and 2. Here X is a discrete random variable. We are often interested in the distribution of the possible outcomes which we can think of as a function from the set of outcomes to the interval [0,1]. Thus in our two coins experiment we have

X	0	1	2
Probability Function	0.25	0.5	0.25

Notice that if we have a generic table

X , the number of heads	x_0	x_1	x_2	$\cdots$	x_n
f(x) = P(X=x)	f(x_0)	f(x_1)	f(x_2)	$\cdots$	f(x_n)

then the $f(x_i)$ must satisfy

1. $f(x_i) \geq 0, \forall i$
2. $\sum_i f(x_i) = 1$

This is extensible to deal with the situation of a continuous probability distribution such as would be found were we to measure the mass of sugar in a 1 kg bag of sugar: we might expect that the mean weight of sugar would be 1 kg but would not expect every single bag of sugar to weigh in at exactly 1 kg. We might plot the difference between the 1 kg weight which is stated on the bag and the actual weight of the sugar as in Figure 2.1. Because we can never get a weight to infinite accuracy, it makes little sense to talk of the probability of a particular weight. Also the generalisation of the features of the distribution above must hold:

[1]This example is taken from Cover and Thomas

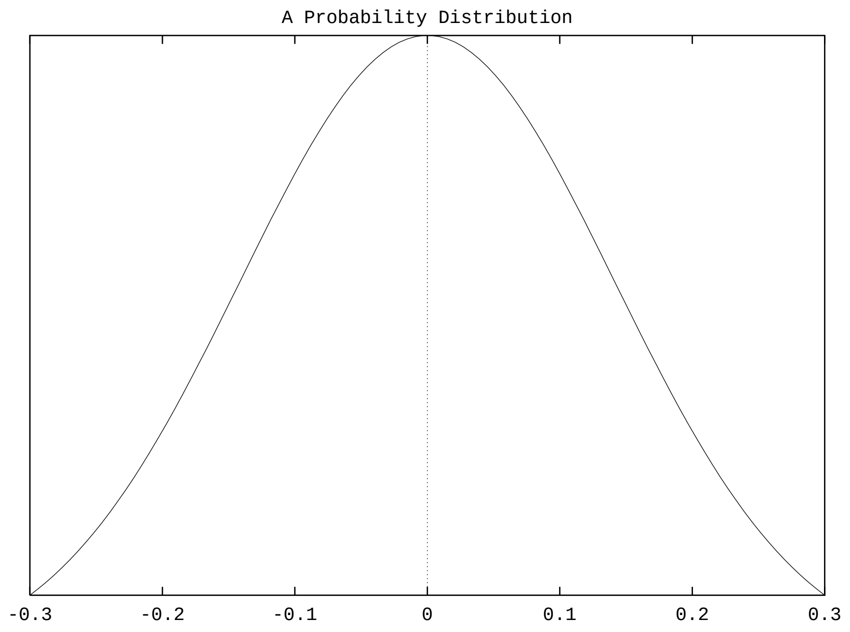


Figure 2.1: A general probability distribution

1. $f(x) \geq 0, \forall x$
2. $\int f(x)dx = 1$, i.e. the total area under the curve must sum to 1.

To find the actual probability associated with any interval, we simply integrate across the interval. Thus, the probability that x lies between the values a and b is

$$P(a < x < b) = \int_a^b f(x)dx$$

We usually call $f(x)$ the probability density function (PDF) of the distribution.

We will often be interested in the expected value of a function of a random variable and will use E(f(X)) to denote this value. The simplest expectation to take is the mean. For discrete variables, we know that

$$\mu_X = E(X) = \sum_i x_i f(x_i)$$

For example in our two coins experiment we have the expected number of heads,

$$\mu(H) = 0.25 * 0 + 0.5 * 1 + 0.25 * 2 = 0 + 0.5 + 0.5 = 1$$

which is what we would have expected! For continuous variables, we have

$$\mu_X = E(X) = \int xf(x)dx$$

to get the usual mean or average.

Expectation is a linear operator in that E(aX + bY) = aE(X) + bE(Y).

The variance of a distribution defines the amount of spread in the distribution. It is calculated from the expected value of the square of the difference of the X-value from the mean:

$$\sigma^2 = Var(X) = E([X - \mu]^2)$$

σ itself is known as the standard deviation. It is easily shown that

$$\sigma^2 = E(X^2) - [E(X)]^2$$

which is often more convenient for on-line calculation since it can be done in 1 pass through the data (as opposed to a first pass to calculate the mean and a second to calculate the squared divergence from the mean). So to calculate the variance of our two coins experiment, we have

$$\sigma^2 = 0.25 * (0-1)^2 + 0.5 * (1-1)^2 + 0.25 * (2-1)^2 = 0.5$$

2.2.1 The Gaussian or Normal Distribution

A continuous random variable X is said to be normally distributed if its probability density function is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\{-\frac{(x-\mu)^2}{2\sigma^2}\}$$

As a probability density function, f(x) must satisfy $\int_{-\infty}^{\infty} f(x)dx = 1$. The value of μ is the mean of the distribution. The value of σ is the standard deviation of the distribution. Large values of σ give a widely spread distribution; small values of σ give a narrowly peaked distribution. The normal distribution is symmetrical about the mean. We sometimes describe the distribution as $N(\mu, \sigma^2)$; a Gaussian distribution is totally described by these two parameters. Figure 2.1 is actually a diagrammatic representation of a Normal distribution. The effect of varying μ and σ is shown in Figure 2.2.

The standard normal distribution is N(0,1); there are tables in many statistical books which give the probability of an outcome being less than a particular value in terms of the standard normal distribution.

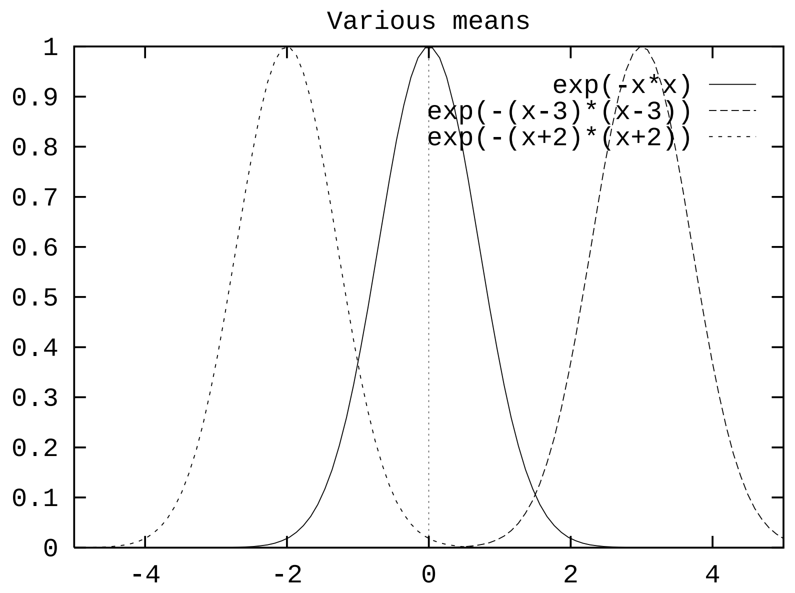
We can of course have distributions of variables in more than one dimension. So if we have a Gaussian distribution of a two dimensional variable which has the same variance in both directions we could view this as a bell-shaped dome rising to its maximum height over the mean (centre) of the distribution. If one direction has greater spread than the other we have a distribution function which looks oval from outside; it will still rise to its greatest height above the mean but will be narrower than it is long. Sometimes there is correlation between the variables in which the distribution is measured and in this case we describe the variance as a matrix, though externally, the distribution will still look bell shaped as before. The only difference now is that the axes of the bell will not correspond to the axes of the measurements.

2.3 Quantification of Information

2.3.1 Logarithms

A quick statement of some facts about logarithms of which you should be aware

1. $y = \log_a(x) \longleftrightarrow a^y = x$
2. $$\begin{aligned} \log_a(x) &= 0 \longleftrightarrow x = 1 \\ \log_a(x) &> 0 \longleftrightarrow x > 1 \\ \log_a(x) &< 0 \longleftrightarrow x < 1 \end{aligned}$$
3. $\log_a(x)$ is not defined for $x < 0$.
4. $\log_a(xy) = \log_a(x) + \log_a(y)$
5. $\log_a(\frac{x}{y}) = \log_a(x) - \log_a(y)$



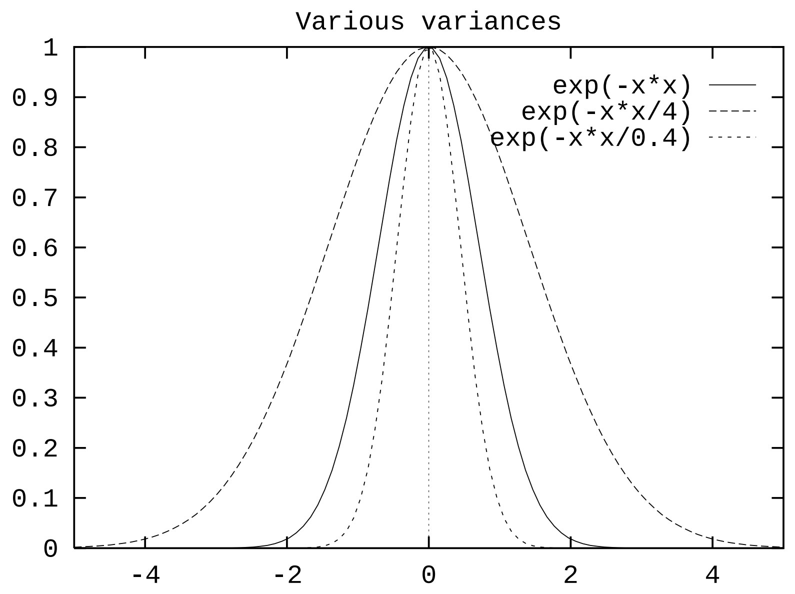


Figure 2.2: The top diagram shows the effect of different values of μ; the bottom of different values of σ. Note that in the bottom figure we have not used the $\frac{1}{\sqrt{\ }}$ term to normalise the height and

6.

$$\begin{aligned}\log_a(x^2) &= 2\log_a(x)\\ \log_a(x^3) &= 3\log_a(x)\\ \log_a(x^n) &= n\log_a(x)\end{aligned}$$

7. $\log_a(a) = 1$

8.

$$\begin{aligned}\log_a(a^2) &= 2\log_a(a) = 2\\ \log_a(a^n) &= n\log_a(a) = n\end{aligned}$$

9. $\log_a(\sqrt{a}) = \frac{1}{2}\log_a(a) = \frac{1}{2}$

10.

$$\begin{aligned}\log_a(\frac{1}{a^2}) &= \log_a(a^{-2}) = -2\log_a(a) = -2\\ \log_a(\frac{1}{a^n}) &= \log_a(a^{-n}) = -n\log_a(a) = -n\end{aligned}$$

These are the most important facts about logarithms you must know. They can be found in any elementary university mathematics book and will not be proved here.

We will begin with a consideration of a finite system of discrete events.

2.3.2 Information

Shannon devised a measure of the information content of an event in terms of the probability of the event happening. He wished to quantify the intuitive concept that the occurrance of an unlikely event tells you more than that of a likely event. He defined the information in an event i, to be $-\log p_i$ where p_i is the probability that the event labelled i occurs.

This satisfies our intuitive concept that if an event which is unlikely (has a low probability) happens, we learn more than if an event which has a high probability happens. If we toss a coin and it turns up Heads we have learned something - we can now rule out the outcome Tails; but when we cut a pack of cards and find we have the King of Spades, we have learned a lot more - we can rule out the other 51 cards.

We see that if an event is bound to happen (and so its probability is 1) that the information we get from it is 0 since log(1) =0. Similarly the closer the probability of an event is to 0, the larger the information quantity we get when the event happens.

If we work in logarithms to base two, we say our information is in bits. Thus, if we toss a coin, P(H) = 0.5 and so the information we get when we see a Head is

$$I = -log_2\frac{1}{2} = \log_2 2 = 1bit$$

which satisfies us as computer scientists in that we know that it takes 1 bit to hold the information that A rather than B has happened. With our previous event of tossing two coins, P(HH) = $\frac{1}{4}$ and so the information we get when we find the event HH is

$$I = -\log_2\frac{1}{4} = \log_2 4 = \log_2 2^2 = 2\log_2 2 = 2bits$$

If we had 3 coins, P(HHH) = $\frac{1}{8}$ and so the information we get from this event is

$$I = -\log_2\frac{1}{8} = \log_2 8 = \log_2 2^3 = 3\log_2 2 = 3bits$$

i.e. we have gained more information from the less likely event.

If we work in log to base 10 our results are in digits; if we work in log to base e we have nats. We will assume $\log_2$ unless otherwise stated but you can use digits for your tutorial work e.g. if we have the coin-tossing experiment, the information gained in digits when a Head is revealed is

$$I(H) = -\log_{10} 0.5 = 0.3$$

while in the two coin experiment,

$$I(HH) = -log_{10}0.25 = 0.6$$

and in the three coin experment,

$$I(HHH) = -log_{10}0.125 = 0.9$$

Notice that while these last three figures are all information measured in digits, they are in the same proportion as the information measured in bits that we used earlier.

2.3.3 Entropy

Using information, we define the entropy (or uncertainty or information content) of a set of N events to be

$$H = -\sum_{i=1}^{N} p_i \log p_i$$

That is, the entropy is the information we would expect to get from one event happening where this expectation is taken over the ensemble of possible outcomes. It is the mean information we get from the dataset.

Thus if we toss a single coin, we have two possible events, each with associated probability of 0.5. So the entropy associated with tossing a coin is

$$\begin{aligned} H &= p(H)I(H) + p(T)I(T) \\ &= -\sum_{i=1}^{2} \frac{1}{2}\log_2\frac{1}{2} = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = log_2 2 = 1bit \end{aligned}$$

which is what we would expect - we need 1 bit of data to describe the result of a single coin tossing experiment.

Now we can consider an example with a non-uniform distribution: let us imagine a horse race with 8 horses taking part. Let the probability of each horse winning be $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64})$. Then the entropy of the horse race is

$$H = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{4}\log\frac{1}{4} - \frac{1}{8}\log\frac{1}{8} - \frac{1}{16}\log\frac{1}{16} - 4\frac{1}{64}\log\frac{1}{64} = 2bits$$

Notice that if we had 8 horses of equal probability, $H = -8 * \frac{1}{8}log\frac{1}{8} = 3bits$ i.e. there is more uncertainty if every horse in the race has equal probability of winning. This is a general rule: there is more uncertainty in a situation if every outcome is equally probable.

Entropy is always ≥ 0 since $0 \leq P(X) \leq 1$ and so $\log\frac{1}{P(X)} \geq 0$.

A Worked Example in Digits

Consider a bag with 8 balls in it. Let there originally be 4 red and 4 black balls and let us not be able to differentiate between the red balls nor within the set of black balls. Then

$$\begin{aligned} P(Red) &= \frac{4}{8} = \frac{1}{2} \\ P(Black) &= \frac{4}{8} = \frac{1}{2} \end{aligned}$$

Then the information we get when we pick a ball is either

$$\begin{aligned} I(Red) &= -\log_{10}\frac{1}{2} = 0.3\text{digits or} \\ I(Black) &= -\log_{10}\frac{1}{2} = 0.3\text{digits} \end{aligned}$$

Then the entropy (or uncertainty or expected information we get when we pick a ball) is

$$\begin{aligned} H &= -\sum_{i=1}^{2} p_i \log_{10} p_i \text{digits} \\ &= -(\frac{1}{2}\log_{10}\frac{1}{2} + \frac{1}{2}\log_{10}\frac{1}{2}) \\ &\approx -(\frac{1}{2} * (-0.3) + \frac{1}{2} * (-0.3)) \\ &= 0.3\text{digits} \end{aligned}$$

Now let us change the experiment: we now have 2 red balls and 6 black balls and so

$$\begin{aligned} P(Red) &= \frac{2}{8} = \frac{1}{4} \\ P(Black) &= \frac{6}{8} = \frac{3}{4} \end{aligned}$$

Then the information we get when we pick a ball is either

$$\begin{aligned} I(Red) &= -\log_{10}\frac{1}{4} = 0.6\text{digits or} \\ I(Black) &= -\log_{10}\frac{3}{4} = 0.12\text{digits} \end{aligned}$$

Then the entropy is

$$\begin{aligned} H &= -\sum_{i=1}^{2} p_i \log_{10} p_i \text{digits} \\ &= -(\frac{1}{4}\log_{10}\frac{1}{4} + \frac{3}{4}\log_{10}\frac{3}{4}) \\ &\approx -(\frac{1}{4} * (-0.6) + \frac{3}{4} * (-0.12)) \\ &= 0.24\text{digits} \end{aligned}$$

Notice the drop in entropy since we have some prior knowledge of what we expect to see when we pick a ball and so we gain less information when we pick it.

Let us continue in this way: we now have 1 red balls and 7 black balls and so

$$\begin{aligned} P(Red) &= \frac{1}{8} \\ P(Black) &= \frac{7}{8} \end{aligned}$$

Then the information we get when we pick a ball is either

$$\begin{aligned} I(Red) &= -\log_{10}\frac{1}{8} = 0.9\text{digits or} \\ I(Black) &= -\log_{10}\frac{7}{8} = 0.058\text{digits} \end{aligned}$$

	1	2	3	4	X
1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$	
2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$	
3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	
4	$\frac{1}{4}$	0	0	0	
y					

Table 2.1: The joint distribution of X and Y.

Then the entropy is

$$\begin{aligned} H &= -\sum_{i=1}^{2} p_i \log_{10} p_i \text{digits} \\ &= -(\frac{1}{8}\log_{10}\frac{1}{8} + \frac{7}{8}\log_{10}\frac{7}{8}) \\ &\approx -(\frac{1}{8} * (-0.9) + \frac{7}{8} * (-0.058)) \\ &= 0.16\text{digits} \end{aligned}$$

2.3.4 Joint and Conditional Entropy

For a pair of random variables X and Y, if $p(i,j)$ is the joint probability of X taking on the i^{th} value and Y taking on the j^{th} value, we define the entropy of the joint distribution as:

$$H(X,Y) = -\sum_{i,j} p(i,j)\log p(i,j)$$

Similarly, we can define the conditional entropy (or equivocation or remaining uncertainty in x if we are given y) as:

$$H(X|Y) = -\sum_{i,j} p(i,j)\log p(i|j)$$

This may seem a little strange since we use the joint probability times the logarithm of the conditional probability, but note that

$$\begin{aligned} H(X|Y) &= \sum_j p(j)H(X|Y=j) \\ &= -\sum_j p(j)\sum_i p(i|j)\log p(i|j) \\ &= -\sum_{i,j} p(i,j)\log p(i|j) \end{aligned}$$

We can relate the conditional and joint entropy using the following formula

$$H(X,Y) = H(X) + H(Y|X) \tag{2.1}$$

A diagrammatic representation of three distributions is shown in Figure 5.10.

A Worked Example

We previously met the joint probability distribution shown in Table 2.1.

We noted that the marginal distribution of X (the distribution of X over all values of Y) is $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ while the marginal distribution of Y is $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$. We will use this now to calculate the associated entropies.

Notice first that

$$\begin{aligned}
H(X) &= -\sum_i p_i \log p_i \\
&= -(\frac{1}{2}\log\frac{1}{2} + \frac{1}{4}\log\frac{1}{4} + 2 * \frac{1}{8}\log\frac{1}{8}) \\
&= \frac{7}{4} \text{ bits} \\
\text{Similarly, } H(Y) &= 2 \text{ bits}
\end{aligned}$$

Now the conditional entropy is given by

$$\begin{aligned}
H(X|Y) &= \sum_{i=1}^{4} p(Y=i)H(X|Y=i) \\
&= \frac{1}{4}H(\frac{1}{2},\frac{1}{4},\frac{1}{8},\frac{1}{8}) + \frac{1}{4}H(\frac{1}{4},\frac{1}{2},\frac{1}{8},\frac{1}{8}) \\
&\quad \frac{1}{4}H(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}) + \frac{1}{4}H(1,0,0,0) \\
&= \frac{1}{4} * \frac{7}{4} + \frac{1}{4} * \frac{7}{4} + \frac{1}{4} * 2 + \frac{1}{4} * 0 \\
&= \frac{11}{8} \text{ bits}
\end{aligned}$$

Similarly we can show that $H(Y|X) = \frac{13}{8}$ bits.

To calculate H(X,Y) we will calculate

$$\begin{aligned}
H(X,Y) &= -\sum_{i,j} p(i,j)\log p(i,j) \\
&= -(\frac{1}{8}\log\frac{1}{8} + \frac{1}{16}\log\frac{1}{16} + ... \\
&= \frac{27}{8} \text{ bits}
\end{aligned}$$

2.3.5 Mutual Information

The mutual information between two random variables X and Y is denoted by I(X,Y).

Shannon also showed that if x is a transmitted signal and y is the received signal, then the information which receiving y gives about x is

$$\begin{aligned}
I(x;y) &= H(x) - H(x|y) && (2.2) \\
\text{or } I(x;y) &= H(y) - H(y|x) && (2.3) \\
\text{or } I(x;y) &= H(x) + H(y) - H(x,y) && (2.4)
\end{aligned}$$

Because of the symmetry of the above equations, this term is known as the mutual information between x and y.

For the example in section 2.3.4, we can calculate the mutual information between X and Y as

$$\begin{aligned}
I(x;y) &= H(x) - H(x|y) \\
&= (\frac{7}{4} - \frac{11}{8}) \text{ bits} \\
&= \frac{3}{8} bits
\end{aligned}$$

This accords well with our intuition that there is some information available about X when we know Y (look at the last line of the table) but not really that much (look at the third line).

The channel capacity is defined to be the maximum value over all possible values of x and y of this mutual information.

In summary, the basic facts in which we will take an interest are:

- Because the occurance of an unlikely event has more information than that of a likely event, it has a higher information content.
- Hence, a data set with high variance is liable to contain more information than one with small variance.
- A channel of maximum capacity is defined by 100% mutual information i.e. $I(x;y) = H(x)$

2.3.6 The Kullback Leibler Distance

We may use entropy-based measures to measure the difference between two different probability distributions. The most common method is to use the Kullback Leibler Distance which is sometimes known as relative entropy.

Relative entropy is defined as

$$D(p \parallel q) = \sum_i p(i) \log \frac{p(i)}{q(i)} = E(\log \frac{p(i)}{q(i)}) \tag{2.5}$$

It can be thought of as a measure of the inefficiency of assuming that the distribution is q when the true distribution is p. e.g. if we know the true distribution of a set of finite symbols, we can construct the optimal (Huffman) coding for that set. But if we assume that the distribution is q when in fact it is p, we will construct an inefficient coding. We can in fact show that in this case the average length of the code (which should be H(p)) will in fact be H(p) + $D(p \parallel q)$.

Relative entropy has the following properties:

- It is always non-negative; in other words, we always lose some efficiency when we assume the wrong probability distribution.
- It is not commutative. $D(p \parallel q) \neq D(q \parallel p)$. Notice that this means that it is not a true distance measure.

An Example

Let us consider an experiment in which there are two possible outcomes 0 and 1. Let us believe that the probability distribution of outcomes is p(0) = 0.5 , p(1) =0.5) when the real distribution is p(0) = 0.25, p(1) = 0.75. Then

$$\begin{aligned} D(q||p) &= \frac{3}{4}\log\frac{\frac{1}{2}}{\frac{3}{4}} + \frac{1}{4}\log\frac{\frac{1}{4}}{\frac{1}{2}} \\ &= 0.1887 \text{ bits} \end{aligned}$$

This is a measure of the penalty we incur if we assume the wrong distribution. Notice that

$$\begin{aligned} D(p||q) &= \frac{1}{2}\log\frac{\frac{1}{2}}{\frac{3}{4}} + \frac{1}{2}\log\frac{\frac{1}{2}}{\frac{1}{4}} \\ &= 0.2075 \text{ bits} \end{aligned}$$

2.3.7 Entropy from a Continuous Distribution

To parallel the discrete distributions, we can define the *differential entropy* of a distribution as

$$h(X) = -\int_S f(x) \log f(x) dx \tag{2.6}$$

where S is the support set of the random variable X. For example if we consider a random variable distributed uniformly between 0 and a, so that its density function is $\frac{1}{a}$ from 0 to a and 0 elsewhere, then its differential entropy is

$$h(X) = -\int_0^a \frac{1}{a}\log\frac{1}{a} = \log a \tag{2.7}$$

This is intuitively appealing since, if a has a large value (the distribution has a large spread) the distribution's entropy is large.

All of those same extensions to the basic idea which we met when we used discrete entropy are valid for differential entropy but we should note that some of our answers are not necessarily finite. We are, however, most often interested in the relationship e.g. between the outputs of a neural network and the inputs and it can be shown that the infinite part of their differential entropy will cancel out meaning that the mutual information between inputs and outputs is a relevant finite quantity to use to measure the effectiveness of the network.

2.3.8 Entropy and the Gaussian Distribution

Let us attempt to find the distribution which has greatest entropy. This task means little in this form since we can merely keep adding points to the distribution to increase the uncertainty/entropy in the distribution. We must constrain the problem in some way before it makes sense.

Haykin puts it this way:

> With the differential entropy of a random variable x defined by
>
> $$h(x) = -\int_{-\infty}^{\infty} f(x)\log f(x)dx \tag{2.8}$$
>
> find the probability density function f(x) for which h(x) is a maximum, subject to the two constraints
>
> $$\int_{-\infty}^{\infty} f(x)dx = 1 \tag{2.9}$$
>
> and
>
> $$\int_{-\infty}^{\infty} (x-\mu)^2 f(x)dx = \sigma^2 = \text{ a constant} \tag{2.10}$$
>
> where μ is the mean of the distribution and σ^2 is its variance.

The first constraint simply ensures that the function f() is a proper probability density function; the second constrains the variance of the distribution. We will show that the distribution with greatest entropy for a given variance (spread) is the Gaussian distribution. There is more uncertainty/information in a Gaussian distribution than in any other comparable distribution!

So we have an optimisation problem (maximise the entropy) under certain constraints. We incorporate the constraints into the optimisation problem using Lagrange multipliers so that we wish to find the maximum of

$$\begin{aligned} & -\int_{-\infty}^{\infty} f(x)\log f(x)dx + \lambda_1\int_{-\infty}^{\infty} f(x)dx + \lambda_2\int_{-\infty}^{\infty}(x-\mu)^2 f(x)dx \\ = & -\int_{-\infty}^{\infty}\{f(x)\log f(x) + \lambda_1 f(x) + \lambda_2(x-\mu)^2 f(x)\}dx \end{aligned}$$

where λ_1 and λ_2 are the Lagrange multipliers. This maximum is achieved when the derivative of the integrand with respect to the function f(x) is zero. i.e. when

$$\begin{aligned} 0 &= -1 - \log f(x) + \lambda_1 + \lambda_2(x-\mu)^2 \\ \log f(x) &= -1 + \lambda_1 + \lambda_2(x-\mu)^2 \\ f(x) &= \exp(-1 + \lambda_1 + \lambda_2(x-\mu)^2) \end{aligned} \tag{2.11}$$

Substituting this into equations 2.9 and 2.10 gives

$$\begin{aligned}\int_{-\infty}^{\infty} \exp(-1+\lambda_1+\lambda_2(x-\mu)^2)dx &= 1 \\ \int_{-\infty}^{\infty} (x-\mu)^2 \exp(-1+\lambda_1+\lambda_2(x-\mu)^2)dx &= \sigma^2\end{aligned}$$

which gives us two equations in the two unknowns λ_1 and λ_2 which can be solved to give

$$\begin{aligned}\lambda_1 &= 1-\log(2\pi\sigma^2) \\ \lambda_2 &= -\frac{1}{2\sigma^2}\end{aligned}$$

which can be inserted in equation 2.11 to give

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{2.12}$$

the probability density function of a Gaussian distribution. When we use this to calculate the entropy of the Gaussian distribution we get

$$h(x) = \frac{1}{2}\{1+\log(2\pi\sigma^2)\} \tag{2.13}$$

In summary, we have shown

1. The Gaussian distribution is the distribution with the greatest entropy for a given variance: if x and y are both random variables with a given variance σ^2 and if x is a Gaussian random variable, then
$$h(x) \geq h(y) \tag{2.14}$$
2. The entropy of a Gaussian random variable is totally determined by its variance. We will later see that this is not true for other distributions.

2.4 Information Theory and the Neuron

Linsker has analysed the effect of noise on a neuron's processing capabilities.

He begins by stating the Infomax principle (we will discuss his network in a later chapter) which can be stated approximately as: it is the responsibility of the neurons in the output layer of a network to jointly maximise the information at the outputs about the input activations of the network. i.e. we should maximise the average mutual information between inputs **x** and outputs **y**. However he shows that this maximisation cannot be done without taking account of noise in the system.

2.4.1 One Neuron with Noise on the Output

Consider the situation in Figure 2.3. We have a single ouput neuron which is receiving the weighted sum of its inputs but whose firing is corrupted by some processing noise. It is known that real neurons are noisy[2] and that the source of this noise may come from a variety of sources. We will not consider the chemical or electrical source of such noise, but simply model the system as

$$y = (\sum_j w_j x_j) + \nu \tag{2.15}$$

The general problem cannot be analysed but we can make some assumptions to make the analysis of (2.15) tractable: let us assume that

[2]Indeed the noise may be essential to the information passing of a real neuron

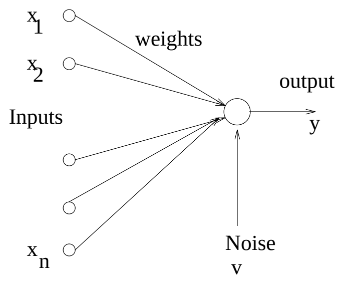


Figure 2.3: A single output neuron takes the weighted sum of the inputs but its firing is corrupted by noise.

- the output of the neuron, y, is a Gaussian random variable with variance σ_y^2.
- the processing noise is also a zero mean Gaussian random variable with variance σ_ν^2.
- the noise is uncorrelated with any of the inputs i.e.

$$E(\nu x_j) = 0, \forall j \tag{2.16}$$

Now the mutual information between inputs and outputs is given by

$$I(\mathbf{x}; y) = h(y) - h(y|\mathbf{x}) \tag{2.17}$$

which is simply stating that the information in the output about the inputs is the expected information in the output minus the entropy/uncertainty in the output when we know the inputs. Now this last term is solely due to the noise since the neuron is otherwise deterministic and so $h(y|\mathbf{x}) = h(\nu)$. Therefore

$$I(\mathbf{x}; y) = h(y) - h(\nu) \tag{2.18}$$

Now with our Gaussian assumptions, we have

$$\begin{aligned} h(y) &= \frac{1}{2}\{1 + \log(2\pi\sigma_y^2)\} \\ h(\nu) &= \frac{1}{2}\{1 + \log(2\pi\sigma_\nu^2)\} \end{aligned}$$

Notice that σ_y depends on σ_ν. Then the information at the output about the inputs is given by

$$\begin{aligned} I(\mathbf{x}; y) &= h(y) - h(\nu) \\ &= \frac{1}{2}\{1 + \log(2\pi\sigma_y^2)\} - \frac{1}{2}\{1 + \log(2\pi\sigma_\nu^2)\} \\ &= \frac{1}{2}\log\frac{\sigma_y^2}{\sigma_\nu^2} \end{aligned}$$

The ratio $\frac{\sigma_y^2}{\sigma_\nu^2}$ is the signal-to-noise ratio of the neuron. Usually we can do little about the variance of the noise and so to improve this ratio we must increase the variance of the output. i.e. to maximise the information at the output about the inputs, we must (under the Gaussian assumptions and with noise only on the outputs) increase the variance of the outputs. We can do this by allowing the weights to grow in magnitude; since increase in the value of the weights does not affect the noise, this can be done independently of any noise effects. Again, intuitively appealing.

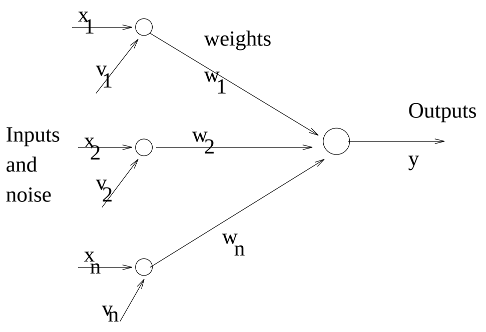


Figure 2.4: Now the noise is affecting the inputs and is therefore being transmitted by the weights.

2.4.2 Noise on the Inputs

However this simple situation does not often prevail; more commonly we have the situation in Figure 2.4 in which there is noise on our inputs. Intuitively we might expect that the fact that the weights are being used to transmit the noise as well as the inputs might limit the effectiveness of weight growth described in the last section. We shall see that this is so.

Let us make the same assumptions about the nature of the distributions and the noise where such assumptions are made with respect to each input noise. Now we have

$$\begin{aligned} y &= \sum_j w_j(x_j+\nu_j) \\ &= \sum_j w_j x_j + \sum_j w_j \nu_j \\ &= \sum_j w_j x_j + \rho \end{aligned}$$

where the summed noise ρ is a zero mean Gaussian distribution whose variance is given by

$$\begin{aligned} \sigma_\rho^2 &= \int \sum_j (w_j\nu_j - 0)^2 f(\rho_j) d\rho_j \\ &= \sum_j w_j^2 \int \nu_j^2 f(\rho_j) d\rho_j \\ &= \sum_j w_j^2 \sigma_\nu^2 \end{aligned}$$

and so the entropy of the noise this time is given by

$$h(\rho) = \frac{1}{2}\{1 + 2\pi\sigma_\nu^2 (\sum_j w_j)^2\} \tag{2.19}$$

and so the mutual information between inputs and output is given by

$$I(\mathbf{x};y) = \frac{1}{2}\log(\frac{\sigma_y^2}{\sigma_\nu^2(\sum_j w_j)^2}) \tag{2.20}$$

Again we must assume that we have no control over the magnitude of the noise and so we must maximise $\frac{\sigma_y^2}{(\sum_j w_j)^2}$. Now in this case it is not sufficient merely to increase the magnitude of the weights since by doing so we are also increasing the effect of the noise on the denominator. More sophisticated tactics must be employed on a neuron-by-neuron basis.

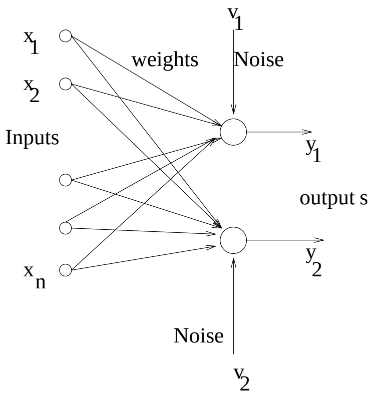


Figure 2.5: Two outputs attempting to jointly convey as much information as possible about the inputs.

2.4.3 More than one output neuron

Consider the network shown in Figure 2.5. Each output neuron's activation is given by

$$y_i = (\sum_j w_{ij} x_j) + \nu_i \tag{2.21}$$

We will use the same assumptions as previously. Since the noise terms are uncorrelated and Gaussian, they are also independent and so

$$h(\nu) = h(\nu_1, \nu_2) = h(\nu_1) + h(\nu_2) = 1 + \log(2\pi\sigma_\nu^2) \tag{2.22}$$

Now since the output neurons are both dependent on the same input vector **x** there will be a correlation between their outputs. Let the correlation matrix be R. Then

$$R = E(\mathbf{y}\mathbf{y}^T) = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \begin{pmatrix} y_1 & y_2 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{pmatrix} \tag{2.23}$$

where $r_{ij} = E(y_i y_j)$. i.e. if the outputs are zero mean, the main diagonal contains the variances of each output while the off-diagonal terms contain the covariances.

$$\begin{aligned} r_{11} &= \sigma_1^2 + \sigma_\nu^2 \\ r_{12} = r_{21} &= \sigma_1 \sigma_2 \rho_{12} \\ r_{22} &= \sigma_2^2 + \sigma_\nu^2 \end{aligned}$$

where $\sigma_i^2, i = 1, 2$ is the variance of each output neuron *in the absence of noise* while ρ_{12} is the correlation coefficient of the output signals also in the absence of noise. Note that in the calculation of r_{21}, we use the fact that $E(\nu_1\nu_2) = 0$ etc.

Now the general case of a multivariate Gaussian distribution has entropy

$$h(\mathbf{y}) = 1 + \log(2\pi \det(R)) \tag{2.24}$$

and so the mutual information is given by

$$I(\mathbf{x}; \mathbf{y}) = \log(\frac{\det(R)}{\sigma_\nu^2}) \tag{2.25}$$

So we again can't do anything about the power in the noise, σ_ν^2 and so to maximise mutual information we must maximise

$$\begin{aligned} \det(R) &= r_{11} r_{22} - r_{12} r_{21} \\ &= \sigma_\nu^4 + \sigma_\nu^2(\sigma_1^2 + \sigma_2^2) + \sigma_1^2 \sigma_2^2 (1 - \rho_{12}^2) \end{aligned} \tag{2.26}$$

So we can identify two separate situations:

Large noise variance If σ_ν^2 is large, we can ignore the third term in equation (2.26) since it is independent of σ_ν^2. Thus since σ_ν^2 is a given (we cannot affect it) we must maximise the central term, $\sigma_1^2 + \sigma_2^2$ which can be done by maximising the variance of either neuron independently of the other. In other words, each neuron is on its own trying to maximise the information which it is giving about the inputs.

Low noise variance But if σ_ν^2 is small, the third term becomes more important. Now there is a trade off between maximising the variance on each output neuron and keeping the correlation coefficient between the neurons as small as possible.

Therefore in a low noise situation, it pays a network to ensure that each output neuron is uncorrelated with the others i.e. is conveying different information about the inputs from the information conveyed by the others. On the other hand when there is a lot of noise in the system, it pays to have redundancy in the outputs: we only ensure that each neuron is conveying as much information about the inputs as possible.

Clearly there exists a continuum of possible answers between the low and high noise limits.

2.5 Principal Component Analysis

Firstly recall that matrix A has an eigenvector **c** if when we multiply **c** by A we get a vector whose direction is the same as **c** though its length may be (usually is) different. We can write this as

$$A\mathbf{c} = \lambda\mathbf{c} \tag{2.27}$$

λ is a scalar known as the eigenvalue.

Inputs to a neural net generally exhibit high dimensionality i.e. the N input lines can each be viewed as 1 dimension so that each pattern will be represented as a coordinate in N dimensional space.

A major problem in analysing data of high dimensionality is identifying patterns which exist across dimensional boundaries. Such patterns may become visible when a change of basis of the space is made, however an *a priori* decision as to which basis will reveal most patterns requires fore-knowledge of the unknown patterns.

A potential solution to this impasse is found in Principal Component Analysis which aims to find that orthogonal basis which maximises the data's variance for a given dimensionality of basis. The usual tactic is to find that direction which accounts for most of the data's variance - this becomes the first basis vector (the first Principal Component direction). One then finds that direction which accounts for most of the remaining variance - this is the second basis vector and so on. If one then projects data onto the Principal Component directions, we perform a dimensionality reduction which will be accompanied by the retention of as much variance (or information) in the data as possible.

In general, it can be shown that the k^{th} basis vector from this process is the same as the k^{th} eigenvector of the co-variance matrix, **C** where

$$c_{ij} = E[(x_i - E(x))(x_j - E(x))]$$

For zero-mean data, the covariance matrix is equivalent to a simple correlation matrix.

Now, if we have a set of weights which are the eigenvectors of the input data's covariance matrix,C, then these weights will transmit the largest values to the outputs when an item of input data is in the direction of the largest correlations which corresponds to those eigenvectors with the largest eigenvalues. Thus, if we can create a situation in an Artificial Neural Network where one set of weights (into a particular output neuron) converges to the first eigenvector (corresponding to the largest eigenvalue), the next set of weights converges to the second eigenvector and so on, we will be in a position to maximally recreate at the outputs the directions with the largest variance in the input data.

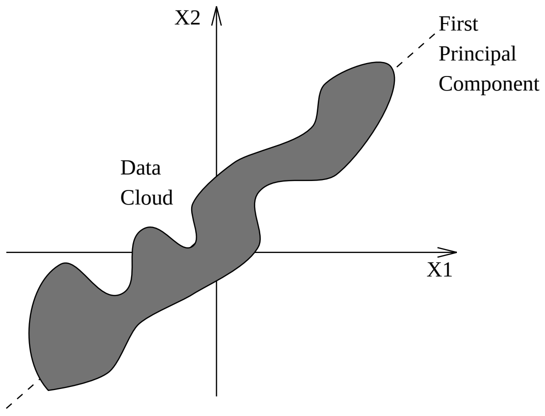


Figure 2.6: There is a correlation between the information in the two directions given. PCA has extracted the first principal component direction which contains most of the variance in the data.

Note that representing data as coordinates using the basis found by a PCA means that the data will have greatest variance along the first principal component, the next greatest variance along the second, and so on. While it is strictly only true to say that information and variance may be equated in Gaussian distributions, it is a good rule-of-thumb that a direction with more variance contains more information than one with less variance. Thus PCA provides a means of compressing the data whilst retaining as much information within the data as possible. A diagrammatical representation is shown in Figure 2.6: here we show the points of a two dimensional distribution on the plane; we therefore require two coordinates to describe each point exactly but if we are only allowed a single coordinate, our best bet is to choose to use the coordinate axis labelled "first principal component". This axis will allow us to represent each point as accurately as possible with a single coordinate. It is the best possible linear compression of the information in the data.

It can be shown that if a set of input data has eigenvalues $\{\lambda_1, \lambda_2, ..., \lambda_n\}$ and if we represent the data in coordinates on a basis spanned by the first m eigenvectors, the loss of information due to the compression is

$$E = \sum_{i=m+1}^{n} \lambda_i \tag{2.28}$$

Artificial Neural Networks and PCA come together in 2 ways:

1. There are some networks which use Principal Components as an aid to learning e.g. by compressing the data on the principal components we are discarding noise in the data and retaining the essential variance/information in the data. We are then using PCA to preprocess the data before letting the network loose on it.

2. Some networks have been explicitly designed to calculate Principal Components

Our interest will lie mainly in the latter type of network.

2.6 A Silly Example

Following Kohonen and Ritter, we represent each of 16 animals/birds by a 29-bit vector, the first 16 bits of which were 0 except for the bit which identified the particular animal. The other bits were associated with the animal's attributes as shown in Table 2.2. The output data were plotted in the obvious manner - each output vector was identified as a binary number (1/0) and converted to decimal to give the coordinates in each direction.

	dove	hen	duck	goose	owl	hawk	eagle	fox	dog	wolf	ca t	tiger	lion	horse	zebra	cow
small	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0
medium	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
big	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
2 legs	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
4 legs	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
hair	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
hooves	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
mane	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
feathers	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
hunt	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
run	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0
fly	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
swim	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.2: Animal names and their attributes

Attribute/class	First PC	Second PC	Third PC
dove	0.017	-0.071	0.054
hen	0.013	-0.057	0.059
duck	0.014	-0.062	0.083
goose	0.018	-0.077	0.078
owl	0.027	-0.077	-0.022
hawk	0.027	-0.077	-0.022
eagle	0.028	-0.060	-0.116
fox	0.044	0.008	-0.155
dog	0.044	0.021	-0.066
wolf	0.061	0.020	-0.118
cat	0.044	-0.009	-0.062
tiger	0.057	0.021	-0.019
lion	0.065	0.026	0.005
horse	0.059	0.036	0.140
zebra	0.059	0.036	0.140
cow	0.041	0.024	0.102
small	0.161	-0.431	0.166
medium	0.177	-0.012	-0.457
big	0.281	0.143	0.369
2 legs	0.146	-0.482	0.113
4 legs	0.474	0.183	-0.034
hair	0.474	0.183	-0.034
hooves	0.159	0.096	0.383
mane	0.243	0.116	0.167
feathers	0.146	-0.482	0.112
hunt	0.354	-0.149	-0.512
run	0.345	0.159	0.081
fly	0.118	-0.364	-0.029
swim	0.032	-0.139	0.161

Table 2.3: The first 3 Principal Components of the input data's covariance matrix. It is easily verified that they form an orthonormal basis of the 3 dimensional subspace

Table 2.3 shows the first 3 Principal Components of the covariance matrix of the data of Table 2.2 as identified by the interneuron weights of the middle layer. It is easily verified that the vectors form an orthonormal basis of a 3 dimensional subspace of the data.i.e. the length of each vector is 1 and the product of any two vectors is 0.

- The first Principal Component is most strongly identifying animal type features: animals tend to be big, have 4 legs and hair; some have hooves or a mane; somewhat more hunt and run.
- The second Principal Component completes the job: the birds all are represented by a negative component as are the small, medium, 2 legs, feathers, hunt, fly and swim attributes. Also, it can be seen that the more prototypical bird-like features have larger absolute values e.g. $|\text{fly}| > |\text{swim}|$ since the prototypical bird is more likely to fly than swim. Note also that the cat has a small negative value brought about by its prototypical bird-like attribute of smallness.
- The Third Principal Component seems to be mainly differentiating the hunters from the non-hunters, though differentiation in size and between fliers and s wimmers is also taking place

Note that the components defining e.g. horse and zebra are identical in all 3 directions as there is nothing in the input data provided which will allow us to discriminate between these groups. Similarly, the attributes "4 legs" and "hair" are identically represented as they are identically distributed in the information we have given.

2.7 Exercise

1. Write out all the possibilities in a table when you roll two dice. What is the probablity of getting a 7? (Have a red die and a black one).
2. Calculate the probability of cutting
   (a) An ace
   (b) A heart
   (c) The ace of hearts
   (d) An ace or a heart.

   Check that P(Ace or Heart) = P(Ace) +P(Heart) - P(Ace and Heart).
   This is a general law: P(A or B) = P(A) + P(B) - P(A and B).
3. An urn contains 3 red balls and 4 black ones and we pick a ball blind from it
   (a) What is the probability that a red ball is picked
   (b) Given that a red ball is picked what is the probability that a black ball is now picked.
   (c) If we now have a second urn containing 4 red balls and 5 black ones and we select an urn at random, what is the probability that a black ball is picked?
4. The probability of getting a first is $\frac{1}{4}$ if you take ANNs, $\frac{1}{3}$ if you take SSM and $\frac{1}{6}$ if you take GBH. If you select your course at random, what is the probability that you do not get a first?
5. A fair die is rolled. Find the mean and variance of the random number X of the value appearing on top.
6. How much information (in bits) do you gain when you cut the ace of spades?

7. How much information do you gain when you throw 10 coins simultaneously? 8. Let p(x,y) be given by

X : Y	0	1
0	1/3	1/3
1	0	1/3

Find

   (a) H(X) and H(Y)
   (b) H(X—Y) and H(Y—X)
   (c) H(X,Y)
   (d) H(Y) - H(Y—X)
   (e) I(X;Y)
   (f) Draw a Venn Diagram for the above

8. A fair coin is flipped until the first head occurs. Let X denote the number of flips required. Find the entropy H(X) in bits. You may find the following expressions useful:

$$\begin{aligned}\sum_{n=1}^{\infty} r^n &= \frac{r}{1-r}\\ \sum_{n=1}^{\infty} nr^n &= \frac{r}{(1-r)^2}\end{aligned}$$

Chapter 3

Hebbian Learning

3.1 Simple Hebbian Learning

The aim of unsupervised learning is to present a neural net with raw data and allow the net to make its own representation of the data - hopefully retaining all information which we humans find important. Unsupervised learning in neural nets is generally realised by using a form of Hebbian learning which is based on a proposal by Donald Hebb who wrote:

When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

Neural nets which use Hebbian learning are characterised by making the activation of a unit depend on the sum of the weighted activations which feed into the unit. They use a learning rule for these weights which depends on the strength of the simultaneous activation of the sending and receiving neuron. With respect to the network depicted in Figure 3.1, these conditions are usually written as

$$\begin{aligned} y_i &= \sum_j w_{ij} x_j && (3.1) \\ \text{and } \Delta w_{ij} &= \alpha x_j y_i && (3.2) \end{aligned}$$

the latter being the learning mechanism. Here y_i is the output from neuron i, x_j is the j^{th} input, and w_{ij} is the weight from x_j to y_i. α is known as the learning rate and is usually a small scalar which may change with time. Note that the learning mechanism says that if x_j and y_i fire simultaneously, then the weight of the connection between them will be strengthened in proportion to their strengths of firing.

It is possible to introduce a (non-linear) function into the system using

$$\begin{aligned} y_i &= g(\sum_j w_{ij} x_j) && (3.3) \\ \text{and } \Delta w_{ij} &= \alpha x_j y_i && (3.4) \end{aligned}$$

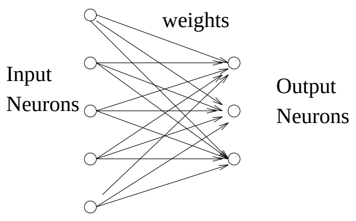


Figure 3.1: A one layer network whose weights can be learned by simple Hebbian learning.

for some function, $g()$; we will still call this Hebb learning.

Substituting Equation (3.1) into Equation (3.2), we can write the Hebb learning rule as

$$\begin{aligned}\Delta w_{ij} &= \alpha x_j \sum_k w_{ik} x_k \\ &= \alpha \sum_k w_{ik} x_k x_j \qquad (3.5)\end{aligned}$$

If we take α to be the time constant, Δt and divide both sides by this constant we get

$$\begin{aligned}\frac{\Delta w_{ij}}{\Delta t} &= \sum_k w_{jk} x_k x_j \\ \text{which, as } \Delta t \rightarrow 0 & \quad \text{is equivalent to} \qquad (3.6) \\ \frac{d}{dt}\mathbf{W}(t) &\propto \mathbf{C}\mathbf{W}(t) \qquad (3.7)\end{aligned}$$

where C_{ij} is the correlation coefficient calculated over all input patterns between the i^{th} and j^{th} terms of the inputs and $\mathbf{W}(t)$ is the matrix of weights at time t. In moving from the stochastic equation (3.5) to the averaged differential equation (3.7), we must place certain constraints on the process particularly on the learning rate α which we will not discuss in this course. We are using the notation

$$\frac{d}{dt}\mathbf{W} = \begin{pmatrix} \frac{dw_{11}}{dt} & \frac{dw_{12}}{dt} & \dots & \frac{dw_{1n}}{dt} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{dw_{m1}}{dt} & \frac{dw_{m2}}{dt} & \dots & \frac{dw_{mn}}{dt} \end{pmatrix} \qquad (3.8)$$

The advantage of this formulation is that it emphasises the fact that the resulting weights depend on the second order statistical properties of the input data i.e. the covariance matrix is the crucial factor.

3.1.1 Stability of the Simple Hebbian Rule

At convergence the weights should have stopped changing. Thus at a point of convergence *if this point exists* $E(\Delta W)$ =0 or in terms of the differential equations $\frac{d}{dt}W = 0$.

However, a major difficulty with the simple Hebb learning rule is that unless there is some limit on the growth of the weights, the weights tend to grow without bound: we have a positive feedback loop - a large weight will produce a large value of y (Equation 3.1) which will produce a large increase in the weight (Equation 3.2). Let us examine mathematically the Hebb rule's stability:

Recall first that a matrix $\mathbf{A}$ has an eigenvector $\mathbf{x}$ with a corresponding eigenvalue λ if

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

In other words, multiplying the vector $\mathbf{x}$ or any of its multiples by $\mathbf{A}$ is equivalent to multiplying the whole vector by a scalar λ. Thus the direction of $\mathbf{x}$ is unchanged - only its magnitude is affected.

Consider a one output-neuron network and assume that the Hebb learning process does cause convergence to a stable direction, $\mathbf{w}^*$; then if w_k is the weight vector linking x_k to y, at convergence,

$$0 = E(\Delta w_i^*) = E(yx_i) = E(\sum_j w_j x_j x_i) = \sum_j R_{ij} w_j$$

where R is the correlation matrix of the distribution. Now this happens for all i , so $\mathbf{R}\mathbf{w} = 0$. Now the correlation matrix, R, is a symmetric, positive semi-definite matrix and so all its eigenvalues are non-negative. But the above formulation shows that $\mathbf{w}^*$ must have eigenvalue 0. Now consider a small disturbance, ϵ, in the weights in a direction with a non-zero (i.e. positive) eigenvalue. Then

$$E(\Delta w*) = R(\mathbf{w}^* + \epsilon) = R\epsilon > 0$$

i.e. the weights will grow in any direction with non-zero eigenvalue (and such directions must exist). Thus there exists a fixed point at **W=0** but this is an unstable fixed point: if all weights happen to be zeroa single small change from this will cause all the weights to move from zero. In fact, it is well known that in time, the weight direction of nets which use simple Hebbian learning tend to be dominated by the direction corresponding to the largest eigenvalue.

We will now discuss one of the major ways of limiting this growth of weights while using Hebbian learning and review its important side effects.

3.2 Weight Decay in Hebbian Learning

As noted in Section 3.1, if there are no constraints placed on the growth of weights under Hebbian learning, there is a tendancy for the weights to grow without bounds. It is possible to renormalise weights after each learning epoch, however this adds an additional operation to the network's processing. By renormalising, we mean making the length of the weight vector always equal to 1 and so after each weight change we divide the weight vector by its length; this preserves its direction but makes sure that its length is 1. We then have the two stage operation:

$$\begin{aligned}\mathbf{w}_j &= \mathbf{w}_j + \mathbf{\Delta w}_j \\ \mathbf{w}_j &= \frac{\mathbf{w}_j}{\|\mathbf{w}_j\|}\end{aligned}$$

Another possibility is to allow the weights to grow until each reaches some limit, e.g. have an upper limit of w^+ and a lower limit of w^- and clip the weights when they reach either of these limits. Clearly a major disadvantage of this is that if all weights end up at one or other of these limits[1] the amount of information which can be retained in the weights is very limited.

A third possibility is to prune weights which do not seem to have importance for the network's operation. However, this is an operation which must be performed using non-local knowledge - typically which weights are of much smaller magnitude than their peers.

Hence, interest has grown in the use of decay terms embedded in the learning rule itself. Ideally such a rule should ensure that no single weight should grow too large while keeping the total weights on connections into a particular output neuron fairly constant. One of the simplest forms of weight decay was developed as early as 1968 by Grossberg and was of the form:

$$\frac{dw_{ij}}{dt} = \alpha y_i x_j - w_{ij} \tag{3.9}$$

It is clear that the weights will be stable (when $\frac{dw_{ij}}{dt} = 0$) at the points where $w_{ij} = \alpha E(y_i x_j)$. Using a similar type of argument to that employed for simple Hebbian learning, we can show that at convergence we must have $\alpha \mathbf{C}\mathbf{w} = \mathbf{w}$. Thus $\mathbf{w}$ would have to be an eigenvector of the correlation matrix of the input data with corresponding eigenvalue $\frac{1}{\alpha}$. We shall be interested in a somewhat more general result.

Grossberg went on to develop more sophisticated learning equations which use weight decay e.g. for his instar coding, he has used

$$\frac{dw_{ij}}{dt} = \alpha\{y_i - w_{ij}\}x_j \tag{3.10}$$

where the decay term is gated by the input term x_j and for outstar coding

$$\frac{dw_{ij}}{dt} = \alpha\{x_j - w_{ij}\}y_i \tag{3.11}$$

where the decay term is gated by the output term y_i. These, while still falling some way short of the decay in which we will be interested, show that researchers of even 15 years ago were beginning to think of both differentially weighted decay terms and allowing the rate of decay to depend on the statistics of the data presented to the network.

[1]This will certainly happen if simple Hebbian learning is used

3.3 Principal Components and Weight Decay

Miller and MacKay have provided a definitive study of the results of a decay term on Hebbian learning. They suggest an initial distinction between Multiplicative Constraints and Subtractive Constraints.

They define Multiplicative Constraints as those satisfying

$$\frac{d}{dt}\,\mathbf{w}(t) = \mathbf{C}\mathbf{w}(t) - \gamma(\mathbf{w})\mathbf{w}(t)$$

where the decay in the weights is governed by the product of a function of the weights, $\gamma(\mathbf{w})$, and the weights, $\mathbf{w}$(t), themselves. The decay term can be viewed as a feedback term which limits the rate of growth of each weight in proportion to the size of the weight itself while the first term defines the Hebbian learning itself.

Subtractive Constraints are satisfied by equations of the form

$$\frac{d}{dt}\mathbf{w}(t) = \mathbf{C}\mathbf{w}(t) - \epsilon(\mathbf{w})\mathbf{n}$$

where the decay in the weights is governed by the product of a function of the weights ,$\epsilon(\mathbf{w})$, and a constant vector,$\mathbf{n}$, (which is often $\{1, 1, ..1\}^T$).

They prove that

- Hebb rules whose decay is governed by Multiplicative Constraints will, in cases typical of Hebb learning, ensure that the weights will converge to a stable point
- This stable point is a multiple of the principal eigenvector of the covariance matrix of the input data
- Hebb rules governed by Subtractive Constraints will tend to lead to saturation of the weights at their extreme permissible values.
- Under Subtractive Constraints, there is actually a fixed point within the permitted hypercube of values but this is unstable and is only of interest in anti-Hebbian learning(see below).
- If specific limits (w^+ and w^-) do not exist, weights under Subtractive Constraints will tend to increase without bound.

In summary then, Subtractive Constraints offer little that cannot be had from simple clipping of the weights at preset upper and lower bounds. Multiplicative Constraints, however, seem to give us not just weights which are conveniently small, but also weights which are potentially useful since

$$y_i = \sum_j w_{ij}x_j = \mathbf{w_i}.\mathbf{x}$$

where $\mathbf{w_i}$ is the vector of weights into neuron y_i and $\mathbf{x}$ is the vector of inputs. But,

$$\mathbf{w_i}.\mathbf{x} = |\mathbf{w_i}||\mathbf{x}|\cos\theta$$

where $|\mathbf{d}|$ is the length of $\mathbf{d}$ and θ is the angle between the 2 vectors.

This is maximised when the angle between the vectors is 0. Thus, if $\mathbf{w}_1$ is the weight into the first neuron which converges to the first Principal Component, the first neuron will maximally transmit information along the direction of greatest correlation, the second along the next largest, etc. In Section 2.5, we noted that these directions were those of greatest variance which from Section 2.3, we are equating with those of maximal information transfer through the system.

Given that there are statistical packages which find Principal Components, we should ask why it is necessary to reinvent the wheel using Artificial Neural Networks. There are 2 major advantages to PCA using ANNs:

1. Traditional statistical packages require us to have available prior to the calculation, a batch of examples from the distribution being investigated. While it is possible to run the ANN models with this method - "batch mode" - ANNs are capable of performing PCA in real-time i.e. as information from the environment becomes available we use it for learning in the network. We are, however, really calculating the Principal Components of a sample, but since these estimators can be shown to be unbiased and to have variance which tends to zero as the number of samples increases, we are justified in equating the sample PCA with the PCA of the distribution. The adaptive/recursive methodology used in ANNs is particularly important if storage constraints are important.

2. Strictly, PCA is only defined for stationary distributions. However, in realistic situations, it is often the case that we are interested in compressing data from distributions which are a function of time; in this situation, the sample PCA outlined above is the solution in that it tracks the moving statistics of the distribution and provides as close to PCA as possible in the circumstances.

   However, most proofs of convergence of ANNs which find Principal Components require the learning rate to converge to 0 in time and, in practice, it is the case that convergence is often more accurate when the learning rate tends to decrease in time. This would preclude an ANN following a distribution's statistics, an example of the well-known trade-off between tracking capability and accuracy of convergence.

We now look at several ANN models which use weight decay with the aim of capturing Principal Components. We will make no attempt to be exhaustive since that would in itself require a thesis; we do however attempt to give representative samples of current network types.

3.4 Oja's One Neuron Model

There were a number of ANN models developed in the 1980s which used Hebbian learning. The most important was Oja's.

Oja proposed a model which extracts the largest principal component from the input data. He suggested a single output neuron which sums the inputs in the usual fashion

$$y = \sum_{i=1}^{m} w_i x_i$$

His variation on the Hebb rule, though, is

$$\Delta w_i = \alpha(x_i y - y^2 w_i)$$

Note that this is a rule defined by Multiplicative Constraints ($y^2 = \gamma(w)$) and so will converge to the principal eigenvector of the input covariance matrix. The weight decay term has the simultaneous effect of making $\sum w_i^2$ tend towards 1 i.e. the weights are normalised.

However, this rule will find only the first eigenvector (that direction corresponding to the largest eigenvalue) of the data. It is not sufficient to simply throw clusters of neurons at the data since all will find the same (first) Principal Component; in order to find other PCs, there must be some interaction between the neurons. Other rules which find other principal components have been identified by subsequent research, an example of which is shown in the next Section.

3.4.1 Derivation of Oja's One Neuron Model

If we have a one neuron model whose activation is modelled by

$$y = \sum_j w_j x_j \tag{3.12}$$

and use simple Hebbian learning with renormalisation

$$\begin{aligned} w_j(t+1) &= w_j(t) + \alpha y(t) x_j(t) \\ w_j(t+1) &= \frac{w_j(t) + \alpha y(t) x_j(t)}{\{\sum_k (w_k(t) + \alpha y(t) x_k(t))^2\}^{\frac{1}{2}}} \end{aligned}$$

If α is very small, we can expand the last equation as a power series in α to get

$$w_j(t+1) = w_j(t) + \alpha y(t)(x_j(t) - y(t) w_j(t)) + O(\alpha^2) \tag{3.13}$$

where the last term, $O(\alpha^2)$ denotes terms which contain a term in the square or higher powers of α which we can ignore if $\alpha << 1$.

Therefore we can look at Oja's rule as an approximation to the simple Hebbian learning followed by an explicit renormalisation.

3.5 Recent PCA Models

We will consider 3 of the most popular PCA models. It is of interest to begin with the development of Oja's models over recent years.

3.5.1 Oja's Subspace Algorithm

The One Neuron network reviewed in the last section is capable of finding only the first Principal Component. While it is possible to use this network iteratively by creating a new neuron and allowing it to learn on the data provided by the residuals left by subtracting out previous Principal Components, this involves several extra stages of processing for each new neuron.

Therefore Oja's Subspace Algorithm provided a major step forward. The network has N output neurons each of which learns using a Hebb type rule with weight decay. Note however that it does not guarantee to find the actual directions of the Principal Components; the weights *do* however converge to an orthonormal basis of the Principal Component Space. We will call the space spanned by this basis the Principal Subspace. The learning rule is

$$\Delta w_{ij} = \alpha(x_j y_i - y_i \sum_k w_{kj} y_k) \tag{3.14}$$

which has been shown to force the weights to converge to a basis of the Principal Subspace.

Two sets of typical results from an experiment are shown in Table 3.2. The results shown in Table 3.2 are from a network with 5 inputs each of zero mean random Gaussians, where x_1's variance is largest, x_2's variance is next largest, and so on. Sample input data is shown in Table 3.1. You should be able to see that there is more spread in the data the further to the left you look. All data is zero mean.

Therefore, the largest eigenvalue of the input data's covariance matrix comes from the first input, x_1, the second largest comes from x_2 and so on. The advantage of using such data is that it is easy to identify the principal eigenvectors (and hence the principal subspace). There are 3 interneurons in the network and it can be seen that the 3-dimensional subspace corresponding to the first 3 principal components has been identified by the weights. There is very little of each vector outside the principal subspace i.e. in directions 4 and 5. The left matrix represents the results from the interneuron network[2], the right shows Oja's results. The lower ($W^T W$) section shows that the product of any two weights vectors is 0 while the product of a weight vector with itself is 1.Therefore the weights form an orthonormal basis of the space (i.e. the weights are at right angles to one another and of length 1). The upper (W) section shows that this space is almost entirely defined by the first 3 eigenvectors.

[2]which is equivalent to the subspace algorithm(see later)

First	Second	Third	Fourth	Fifth
-4.28565	-3.91526	3.13768	0.0433859	-0.677345
4.18636	3.43597	2.81336	-1.08159	-1.63082
-6.56829	0.849423	6.22813	-2.35614	-0.903031
5.97706	6.2691	-1.70276	-4.28273	0.626593
-2.54685	2.1765	-4.60265	2.34825	0.00339159
4.48306	-3.4953	3.50614	-2.22695	0.107193
-3.92944	-0.0524066	-4.75939	-0.816988	-1.10556
-1.37758	-2.22294	-0.108765	1.19515	1.84522
0.849091	0.189594	-3.75911	0.597238	1.73941
2.60213	-0.952078	-0.542339	0.58135	0.459956
6.21475	-0.48011	-1.31189	-2.50365	-0.809325
-4.33518	2.53261	1.47284	-4.52822	1.6673
10.1211	-4.96799	3.61302	0.00288919	0.48462
1.1967	3.71773	0.214127	0.105751	-0.343055
-8.72964	8.72083	1.2801	-1.41662	1.21766
10.8954	-7.03958	-2.00256	-2.27068	-2.1738
-2.1017	-0.779569	3.09251	1.51042	-2.11619
1.63661	2.40136	-4.79798	0.190769	-0.225283
-0.736219	0.389274	1.65305	1.79372	-0.133088
2.51133	-3.50206	-2.2774	2.13589	1.01751

Table 3.1: Each column represents the values input to a single input neuron. Each row represents the values seen by the network at any one time.

W			W		
0.249	**0.789**	**0.561**	**0.207**	**-0.830**	**0.517**
0.967	**-0.234**	**-0.100**	**-0.122**	**0.503**	**0.856**
-0.052	**-0.568**	**0.821**	**0.970**	**0.241**	-0.003
0.001	0.002	0.016	-0.001	0.001	0.001
-0.001	0.009	0.005	0.000	0.000	-0.001
W^TW			W^TW		
1.001	0.000	0.000	**1.000**	0.000	0.000
0.000	**1.000**	0.000	0.000	**1.000**	0.000
0.000	0.000	**1.000**	0.000	0.000	**1.000**

Table 3.2: Results from the simulated network and the reported results from Oja *et al.* The left matrix represents the results from the negative feedback network (see next Chapter), the right from Oja's Subspace Algorithm. Note that the weights are very small outside the principal subspace and that the weights form an orthonormal basis of this space. Weights above 0.1 are shown in bold font.

W			W		
1.000	-0.036	-0.008	**1.054**	-0.002	-0.002
0.036	**0.999**	-0.018	0.002	**1.000**	0.001
0.010	0.018	**1.000**	0.003	-0.002	**0.954**
-0.002	-0.002	0.016	-0.001	0.001	-0.002
0.010	0.003	0.010	0.001	-0.001	0.000
W^TW			W^TW		
1.001	0.000	0.000	**1.111**	0.000	0.000
0.000	**1.000**	0.000	0.000	**1.000**	0.000
0.000	0.000	**1.000**	0.000	0.000	**0.909**

Table 3.3: Results from the interneuron network (left) and from Oja (right).
Both methods find the principal eigenvectors of the input data covariance matrix. The interneuron algorithm has the advantage that the each vector is equally weighted.

One advantage of this model compared with some other networks is that it is completely homogeneous i.e. the operations carried out at each neuron are identical. This is essential if we are to take full advantage of parallel processing.

The major disadvantage of this algorithm is that it finds only the Principal Subspace of the eigenvectors not the actual eigenvectors themselves.

3.5.2 Oja's Weighted Subspace Algorithm

The final stage is the creation of algorithms which find the actual Principal Components of the input data. In 1992, Oja *et al* recognised the importance of introducing asymmetry into the weight decay process in order to force weights to converge to the Principal Components. The algorithm is defined by the equations

$$y_i = \sum_{j=1}^{n} w_{ij} x_j$$

where a Hebb-type rule with weight decay modifies the weights according to

$$\Delta w_{ij} = \alpha y_i (x_j - \theta_i \sum_{k=1}^{N} y_k w_{kj})$$

Ensuring that $\theta_1 < \theta_2 < \theta_3 < ...$ allows the neuron whose weight decays proportional to θ_1 (i.e. whose weight decays least quickly) to learn the principal values of the correlation in the input data. That is, this neuron will respond maximally to directions parallel to the principal eigenvector, i.e. to patterns closest to the main correlations within the data. The neuron whose weight decays proportional to θ_2 cannot compete with the first but it is in a better position than all of the others and so can learn the next largest chunk of the correlation, and so on.

It can be shown that the weight vectors will converge to the principal eigenvectors in the order of their eigenvalues. The algorithm clearly satisfies Miller and Mackay's definition of Multiplicative Constraints with $\gamma(w_i) = \theta_i \sum_k y_k w_{ki} x_i$. To compare the results with Oja's Weighted Subspace Algorithm, we repeat ed the above experiment with the algorithm. The results are shown in Table 3.3; the left set is from the negative feedback network (next Chapter), the right from Oja's Weighted Subspace Algorithm.

Clearly both methods find the Principal eigenvectors. We note that the interneuron results have the advantage of equally weighting each eigenvector.

3.5.3 Sanger's Generalized Hebbian Algorithm

Sanger has developed a different algorithm (which he calls the "Generalized Hebbian Algorithm") which also finds the actual Principal Components. He also introduces asymmetry in the decay term of his learning rule:

$$\Delta w_{ij} = \alpha(x_j y_i - y_i \sum_{k=1}^{i} w_{kj} y_k) \tag{3.15}$$

Note that the crucial difference between this rule and Oja's Subspace Algorithm is that the decay term for the weights into the i^{th} neuron is a weighted sum of the first i neurons' activations. Sanger's algorithm can be viewed as a repeated application of Oja's One Neuron Algorithm by writing it as

$$\Delta w_{ij} = \alpha([x_j y_i - y_i \sum_{k=1}^{i-1} w_{kj} y_k] - y_i^2 w_{ij}) \tag{3.16}$$

We see that the central term comprises the residuals after the first j-1 Principal Components have been found, and therefore the rule is performing the equivalent of One Neuron learning on subsequent residual spaces. So that the first neuron is using

$$\Delta w_{1i} = \alpha(x_i y_1 - y_1^2 w_{1i}) \tag{3.17}$$

while the second is using

$$\Delta w_{2i} = \alpha([x_i y_2 - y_2 w_{1i} y_1] - y_2^2 w_{2i}) \tag{3.18}$$

and so on for all the rest.

However, note that the asymmetry which is necessary to ensure convergence to the actual Principal Components, is bought at the expense of requiring the j^{th} neuron to 'know' that it is the j^{th} neuron by subtracting only j terms in its decay. It is Sanger's contention that all true PCA rules are based on some measure of deflation such as shown in this rule.

3.6 The InfoMax Principle in Linsker's Model

Linsker has developed a Hebb learning ANN model which attempts to realise the InfoMax principle - the neural net created should transfer the maximum amount of information possible between inputs and outputs subject to constraints needed to inhibit unlimited growth. Linsker notes that this criterion is equivalent to performing a principal component analysis on the cell's inputs.

Although Linsker's model is a multi-layered model, it does not use a supervised learning mechanism; he proposes that the information which reaches each layer should be processed in a way which maximally preserves the information. That this does not, as might be expected, lead to an identity mapping, is actually due to the effect of noise. Each neuron "responds to features that are statistically and information-theoretically most significant". He equates the process with a Principal Component Analysis.

Linsker's network is shown in Figure 3.2. Each layer comprises a 2-dimensional array of neurons. Each neuron in layers from the second onwards receives input from several hundred neurons in the previous layer and sums these inputs in the usual fashion. The region of the previous layer which sends input to a neuron is called the receptive field of the neuron and the density of distribution of inputs from a particular region of the previous layer is defined by a Gaussian distribution i.e. we can imagine two layers of neurons with the a neuron in the second layer receiving most activation from those neurons which lie directly below it and less activation from those neurons further away from directly beneath it. At the final layer, lateral connections within the layer are allowed.

The Hebb-type learning rule is

$$\Delta w_{ij} = a(x_j - E(x))(y_i - E(y)) + b$$

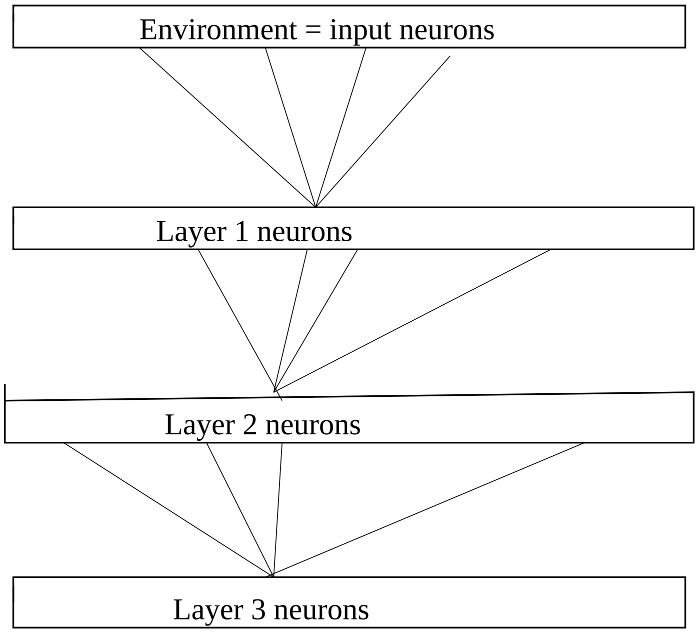


Figure 3.2: Linsker's model

where a and b are constants.

In response to the problem of unlimited growth of the network weights, Linsker uses a hard limit to the weight-building process i.e. the weights are not allowed to exceed w^+ nor decrease beyond w^- where $w^- = -w^+$.

Miller and MacKay have observed that Linsker's model is based on Subtractive Constraints, i.e.

$$\Delta w_{ij} = ax_j y_i - aE(x)y_i - aE(y)(x_j - E(x))$$

Both y_i and $E(y)$ are functions of w, but in neither case are we multiplying these by w itself. Therefore, as noted earlier, the weights will not tend to a multiple of the principal eigenvector but will saturate at the bounds (w_{ij}^+ *or* w_{ij}^-) of their permissible values.

Because the effects of the major eigenvectors will still be felt, there will not be a situation where a weight will tend to w^- in a direction where the principal eigenvector has a positive correlation with the other weights. However, the directions of the weight matrix will, in general, bear little resemblence to any eigenvector of the correlation matrix. The model will not, in general, enable maximal information transfer through the system.

Linsker showed that after several layers, this model trained on noise alone, developed

- center-surround cells - neurons which responded optimally to inputs of a bright spot surrounded by darkness or vice-versa
- bar detectors - neurons which responded optimally to lines of activity in certain orientations

Such neurons exist in the primary visual cortex of mammals. They have been shown to respond even before birth to their optimal inputs though their response is refined by environmental influences i.e. by experience.

3.7 Regression

Regression comprises finding the best estimate of a dependent variable, y, given a vector of predictor variables, **x**. Typically, we must make some assumptions about the form of the predictor

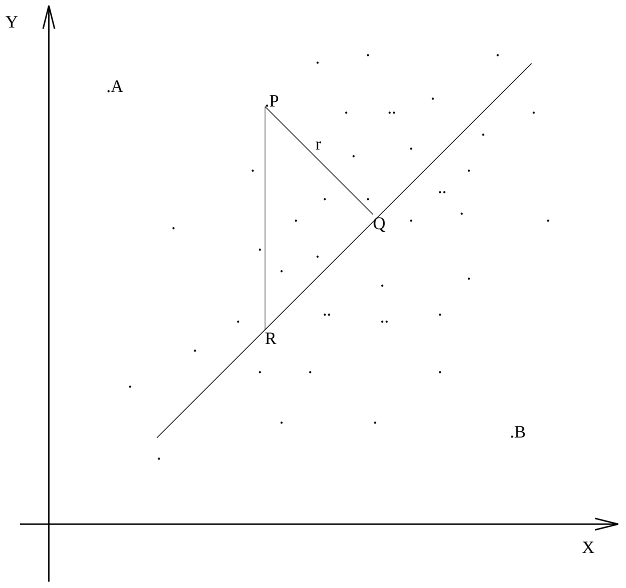


Figure 3.3: The vertical lines will be minimised by the Least Squares method. The shortest distances, r_i, will be minimised by the Total Least Squares method.

surface e.g. that the surface is linear or quadratic, smooth or disjoint etc.. The accuracy of the results achieved will test the validity of our assumptions.

This can be more formally stated as: let (X,Y) be a pair of random variables such that $X \in R^n, Y \in R$. Regression aims to estimate the response surface,

$$f(x) = E(Y|X = x) \tag{3.19}$$

from a set of p observations, $\mathbf{x}_i, y_i, i = 1, ..., p$.

The usual method of forming the optimal surface is the Least (Sum of) Squares Method which minimises the Euclidean distance between the actual value of y and the estimate of y based on the current input vector, $\mathbf{x}$. Formally, if we have a function, f, which is an estimator of the predictor surface, and an input vector, $\mathbf{x}$, then our best estimator of y is given by minimising

$$E = \min_f \sum_i^N (y_i - f(\mathbf{x}_i))^2 \tag{3.20}$$

i.e. the aim of the regression process is to find that function f which most closely matches y with the estimate of y based on using f() on the predictor, $\mathbf{x}$, for all values (y,$\mathbf{x}$).

For a linear function of a scalar x, we have $y = mx + c$, and so the search for the best estimator, f, is the search for those values of m and c which minimise

$$E_1 = \min_{m,c} \sum_i (y_i - mx_i - c)^2$$

For each sample point in Figure 3.3, this corresponds to finding that line which minimises the sum of the vertical lengths such as PR from all actual y-values to the best-fitting line, $y = mx + c$.

However, in minimising this distance, we are making an assumption that only the y-values contain errors while the x-values are known accurately. This is often not true in practical situations in which, for example, which variable constitutes the response variable and which the predictor variables is often a matter of choice rather than being a necessary feature of the problem. Therefore, the optimal line will be that which minimises the distance, r, i.e. which minimises the shortest distance from each point, (x_i, y_i) to the best fitting line. Obviously, if we know the relative magnitude of the errors in x and y, we will incorporate that into the model; however here we assume no foreknowledge of the magnitudes of errors. Thus, we are seeking those values of m and c which minimise

$$E_2 = \min_{m,c} \sum_i r_i^2 = \min_{m,c} \sum_i \frac{(y_i - mx_i - c)^2}{1+m^2}$$

This is the so-called Total Least Squares method. Because of the additional computational burden introduced by the non-linearity in calculating E_2, TLS is less widely used than LS although the basic idea has been known for most of this century.

3.7.1 Minor Components Analysis

Xu *et al.* have shown that the TLS fitting problem can be solved by performing a Minor Component Analysis of the data: i.e. finding those directions which instead of containing maximum variance of the data contain minimum variance. Since there may be errors in both y and x we do not differentiate between them and indeed incorporate y into the input vector $\mathbf{x}$. Therefore we reformulate the problem as: find the direction $\mathbf{w}$ such that we minimise E_2 i.e.

$$\begin{aligned} E_2 &= \min_{\mathbf{w}} \frac{(\mathbf{w}.\mathbf{x}+c)^2}{\mathbf{w}^2} \text{ over all inputs } \mathbf{x} \\ &= \min_{\mathbf{w}} \sum_{i=1}^{N} \frac{(\mathbf{w}.\mathbf{x}_i+c)^2}{\mathbf{w}^2} \\ &= N \min_{\mathbf{w}} \frac{\mathbf{w}^T\mathbf{R}\mathbf{w} + 2c\mathbf{w}^T\mathbf{E}(\mathbf{x}) + c^2}{\mathbf{w}^T\mathbf{w}} \end{aligned}$$

where $R = \frac{1}{N}\sum_{i=1}^N \mathbf{x_i}\mathbf{x_i}^T$, the autocorrelation matrix of the data set and $E(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^N \mathbf{x}_i$, the mean vector of the data set. Since, at convergence, $\frac{dE_2}{d\mathbf{w}} = 0$, we must have

$$R\mathbf{w} + cE(\mathbf{x}) - \lambda\mathbf{w} = 0 \tag{3.21}$$

where $\lambda = \frac{\mathbf{w}^T R\mathbf{w} + 2c\mathbf{w}^T E(\mathbf{x}) + c^2}{\mathbf{w}^T\mathbf{w}}$. Now we wish to find a hyperplane of the form

$$\mathbf{w}.\mathbf{x} + c = 0$$

So, taking expectations of this equation we have $c = -\mathbf{w}.\mathbf{E}(\mathbf{x})$ which we can substitute

$$C\mathbf{w} - \lambda\mathbf{w} = 0 \tag{3.22}$$

where now $\lambda = \frac{\mathbf{w}^T C\mathbf{w}}{\mathbf{w}^T\mathbf{w}}$ where C is the covariance matrix $= R - E(\mathbf{x}\mathbf{x}^T)$. From this we can see that every eigenvector is a solution of the minimisation of E_2.

Using a similar technique to that used earlier to show that only the greatest Principal Component was stable for Oja's one neuron rule, we can now show that only the smallest Principal Component is stable for this rule.

As an example of the network in operation, we show in the first line of Table 3.4 the converged values of the weights of an MCA network when sample points are drawn from the line and both x and y coordinates are subject to noise. Clearly the algorithm has been successful.

For the lines shown in Table 3.4, points were drawn uniformly from only the first (both x and y positive) quadrant of the distribution determined by the line in each case. The first 3 lines show the direction to which the network converged when the distribution was affected by only white noise in both x and y direction drawn from N(0,0.05). Clearly the degree of accuracy of the convergence depends very greatly on the relative proportion of the amount of variance due to the length of the distribution from which points were drawn and the white noise. In the third case, the noise was of the same order as the variance due to the spread of points on the line and the convergence was severly disrupted.

Actual Distribution	Direction Found	Outliers
3x + 2y = 10	0.300x + 0.200y = 1	none
3x + 2y = 1	2.970x + 2.018y = 1	none
3x + 2y = 0.1	24.3x + 23.7y =1	none
3x + 2y = 1	3.424x + 1.714y = 1	1% in y direction
3x + 2y = 1	2.459x + 2.360y = 1	1% in x direction

Table 3.4: Directions converged to when the points from the distribution were disturbed by noise drawn from N(0,0.05)

3.8 Your Practical Work

3.8.1 Annealing of Learning Rate

The mathematical theory of learning in Principal Component Nets requires the learning rate to be such that $\alpha_k \geq 0, \sum \alpha_k^2 < \infty, \sum \alpha_k = \infty$. In practise, we relax these requirements somewhat and we find that we can generally find an approximation to the Principal Components when we use a small learning rate.

However for more accurate results we can anneal the learning rate to zero during the course of the experiment. Different annealing schedules have been tried - e.g. to subtract a small constant from the learning rate at each iteration, to multiply the learning rate by a number < 1 during the course of each iteration, to have a fixed learing rate for the first 1000 iterations and then to anneal it and so on. All of these have been successfully used in practise and it is reccommended that your simulations use one of these methods.

3.8.2 The Data

The theory of Principal Components is most easily applied when we have Gaussian distributions. Not all compilers come with a built-in Gaussian distribution but most usually have a means of creating samples from a uniform distribution from 0 to 1: e.g. in C on the Unix workstations, we typically use drand48(). Donald Knuth has an algorithm which provides a means of creating a pseudo-Gaussian distribution from a uniform distribution. The C++ code for doing so is shown below:

```
#define A1  3.949846138
#define A3  0.252408784
#define A5  0.076542912
#define A7  0.008355968
#define A9  0.029899776

double cStat::normalMean(double mean,double stdev)
{
        int j;
        float r,x;
        float rsq;

        r=0;
        for(j=0;j<12;j++) r += drand48();
        r = (r-6)/4;

        rsq = r*r;
        x = ((((A9*rsq+A7)*rsq + A5)*rsq + A3)*rsq+A1)*r;
        return mean+x*stdev;
}
```

This is a function which takes in two doubles (the mean and standard deviation of the distribution you wish to model) and returns a single value from that distribution. It uses 5 values, $A_1 - A_9$, which are constant for the life of the program. You should call the function with e.g.
x[0] = aStat.normalMean(0,7);
x[1]= aStat.normalMean(0,6); etc

Chapter 4

Anti-Hebbian Learning

All the ANNs we have so far met have been feedforward networks - activation has been propagated only in one direction. However, many real biological networks are characterised by a plethora of recurrent connections. This has led to increasing interest in networks which, while still strongly directional, allow activation to be transmitted in more than one direction i.e. either laterally or in the reverse direction from the usual flow of activation. One interesting idea is to associate this change in direction of motion of activation with a minor modification to the usual Hebbian learning rule called Anti-Hebbian learning.

If inputs to a neural net are correlated, then each contains information about the other. In information theoretical terms, there is redundancy in the inputs ($I(x;y) > 0$).

Anti-Hebbian learning is designed to decorrelate input values. The intuitive idea behind the process is that more information can be passed through a network when the nodes of the network are all dealing with different data. The less correlated the neurons' responses, the less redundancy is in the data transfer. Thus the aim is to produce neurons which respond to different signals. If 2 neurons respond to the same signal, there is a measure of correlation between them and this is used to affect their responses to future similar data. Anti-Hebbian learning is sometimes known as lateral inhibition as this type of learning is generally used between members of the same layer and not between members of different layers. The basic model(Figure 4.1) is defined by

$$\Delta w_{ij} = -\alpha y_i y_j$$

Therefore, if initially y_i and y_j are highly correlated then the weights between them will grow to a large negative value and each will tend to turn the other off.

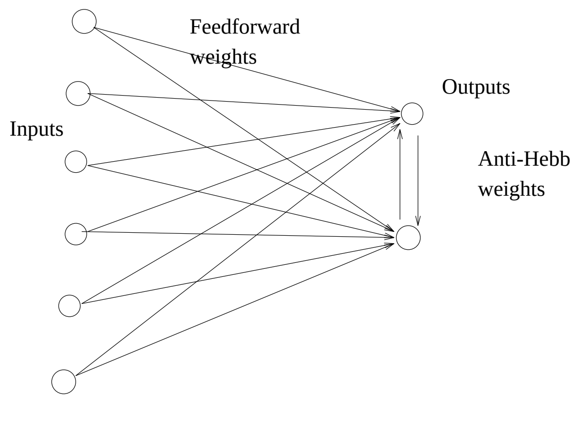


Figure 4.1: Anti-Hebbian Weights
Negative decorrelating weights between neurons in the same layer are learned using an "anti-Hebbian" learning rule

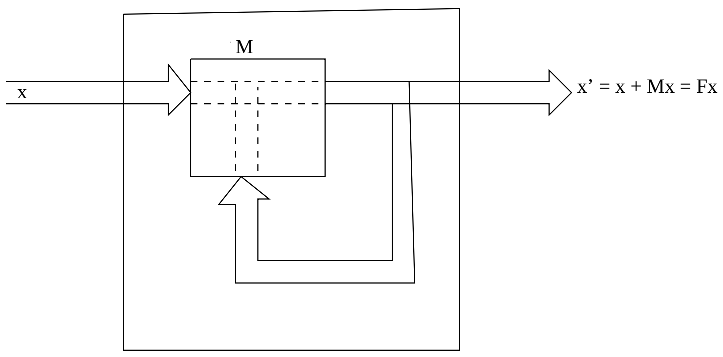


Figure 4.2: The System Model of the Novelty Filter

It is clear that there is no need for weight decay terms or limits on anti-Hebbian weights as they are automatically self-limiting, provided decorrelation can be attained. When the outputs have been decorrelated, we have $E(y_i y_j) = 0$ and

$$(E(y_i.y_j) \to 0) \Longrightarrow (E(\Delta w_{ij}) \to 0) \tag{4.1}$$

i.e. weight change stops when the outputs are decorrelated. Success in decorrelating the outputs results in weights being stabilised.

Several authors have developed Principal Component models using a mixture of one of the above PCA methods (often Oja's One Neuron Rule) and Anti-Hebbian weights between the output neurons.

We first note a similarity between the aims of PCA and anti-Hebbian learning: the aim of anti-Hebbian learning is to decorrelate neurons. If a set of neurons performs a Principal Component Analysis, their weights form an orthogonal basis of the space of principal eigenvectors. Thus, both methods perform a decorrelation of the neurons' responses.

Further, in information theoretic terms, decorrelation ensures that the maximal amount of information possible for a particular number of output neurons is transferred through the system. We will consider only noise-free information-transfer since if there is some noise in the system, some duplication of information may be beneficial to optimal information transfer.

4.1 The Novelty Filter

The role of negative feedback in static models has most often been as the mechanism for competition often based on biological models of activation transfer and sometimes based on psychological models.

An interesting early model was proposed by Kohonen who uses negative feedback in a number of models, the most famous of which (at least of the simple models) is the so-called "novelty filter" (see Figure 4.2). Here we have an input vector **x** which generates feedback gain by the vector of weights, M. Each element of M is adapted using anti-Hebbian learning:

$$\begin{aligned} \frac{dm_{ij}}{dt} &= -\alpha x_i^{'} x_j^{'} \\ \text{where } \mathbf{x}^{'} &= \mathbf{x} + \mathbf{M}\mathbf{x}^{'} \\ \text{and so } (I+M)\mathbf{x}^{'} &= \mathbf{x} \end{aligned}$$

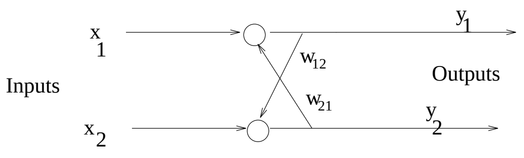


Figure 4.3: Földiák's First Model

$$\text{Therefore } \mathbf{x}' = (\mathbf{I}-\mathbf{M})^{-1}\mathbf{x} = \mathbf{F}\mathbf{x} \tag{4.2}$$

"It is tentatively assumed $(I-M)^{-1}$ always exists." Kohonen shows that, under fairly general conditions on the sequence of $\mathbf{x}$ and the initial conditions of the matrix M, the values of F always converge to a projection matrix under which the output $\mathbf{x}'$ approaches zero although F does not converge to the zero matrix i.e. F converges to a mapping whose kernel (that bit of the space which is mapped to 0) is the subspace spanned by the vectors $\mathbf{x}$. Thus any new input vector $\mathbf{x_1}$ will cause an output which is solely a function of the novel features in $\mathbf{x_1}$.

4.2 Földiák's First Model

Földiák has suggested many neural net models, several of which combine anti-Hebbian learning and weight decay. Here, we will examine the first 2 as they are examples of solely anti-Hebbian learning.

The first model is shown diagrammatically in Figure 4.3 and has anti-Hebbian connections between the output neurons.

The equations which define its dynamical behaviour are

$$y_i = x_i + \sum_{j=1}^{n} w_{ij} y_j$$

with learning rule

$$\Delta w_{ij} = -\alpha y_i y_j \text{ for } i \neq j$$

In matrix terms, we have

$$\begin{aligned} \mathbf{y} &= \mathbf{x} + \mathbf{W}\mathbf{y} \\ \text{And so, } \mathbf{y} &= (\mathbf{I}-\mathbf{W})^{-1}\mathbf{x} \end{aligned}$$

Therefore we can view the system as a transformation, T, from the input vector $\mathbf{x}$ to the output $\mathbf{y}$ given by

$$\mathbf{y} = \mathbf{T}\mathbf{x} = (\mathbf{I}-\mathbf{W})^{-1}\mathbf{x} \tag{4.3}$$

Now the matrix W must be symmetric and has only non-zero non-diagonal terms i.e. if we consider only a two input, two output net as in the diagram,

$$W = \begin{pmatrix} 0 & w \\ w & 0 \end{pmatrix} \tag{4.4}$$

so that T is given by

$$T = (I-W)^{-1} = \begin{pmatrix} 1 & -w \\ -w & 1 \end{pmatrix}^{-1} = \frac{1}{1-w^2}\begin{pmatrix} 1 & w \\ w & 1 \end{pmatrix} \tag{4.5}$$

Now let the two dimensional input vector have correlation matrix

$$C_{xx} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \tag{4.6}$$

where ρ is the correlation coefficient. Now the correlation matrix for y can be calculated since $\mathbf{y} = \mathbf{Tx}$ we have $C_{yy} = E(yy^T) = E(T\mathbf{x}.(T\mathbf{x})^T) = TC_{xx}T^T$. Then

$$C_{yy} = \frac{1}{(w^2-1)^2}\begin{pmatrix} \sigma_1^2 + 2w\rho\sigma_1\sigma_2 + w^2\sigma_2^2 & \rho\sigma_1\sigma_2(w^2+1) + (\sigma_1^2+\sigma_2^2)w \\ \rho\sigma_1\sigma_2(w^2+1) + (\sigma_1^2+\sigma_2^2)w & w^2\sigma_1^2 + 2w\rho\sigma_1\sigma_2 + \sigma_2^2 \end{pmatrix} \tag{4.7}$$

The anti-Hebb rule reaches equilibrium when the the units are decorrelated and so the terms $w_{12} = w_{21} = 0$. Notice that this gives us a quadratic equation in w (which naturally we can solve).

Let us consider the special case that the elements of x have the same variance so that $\sigma_1 = \sigma_2 = \sigma$. Then the cross correlation terms become $\rho\sigma^2w^2 + 2\sigma^2w + \rho\sigma^2$ and so we must solve the quadratic equation

$$\rho w^2 + 2w + \rho = 0 \tag{4.8}$$

which has a zero at

$$w_f = \frac{-1 + \sqrt{1-\rho^2}}{\rho} \tag{4.9}$$

Földiák further shows that this is a stable point in the weight space.

4.2.1 An Example

Consider data which has correlation matrix

$$C_{xx} = \begin{pmatrix} 1 & 0.957 \\ 0.957 & 1 \end{pmatrix} \tag{4.10}$$

Notice that the cross correlations are very high. This data was actually created by using the algorithm

$$\begin{aligned} a &= N(0,5) \\ x_1 &= N(0,1) + a \\ x_2 &= N(0,1) + a \end{aligned}$$

where N(0,d) is a normal distribution with zero mean and standard deviation d.

When we use Földiák's algorithm on this data we find that the w weight converges to -0.753. The decorrelation can be seen graphically in Figure 4.4: the high correlation in the input data is clearly visible - a high positive value of x_1 means a high positive value for x_2 and vice versa - while there is no apparent structure in the output values.

4.2.2 Földiák's Second Model

Földiák suggests a further model by allowing all neurons to receive their own outputs with weight 1.

$$\Delta w_{ii} = \alpha(1 - y_i y_i)$$

which can be written in matrix form as

$$\Delta W = \alpha(I - \mathbf{y}\mathbf{y}^T) \tag{4.11}$$

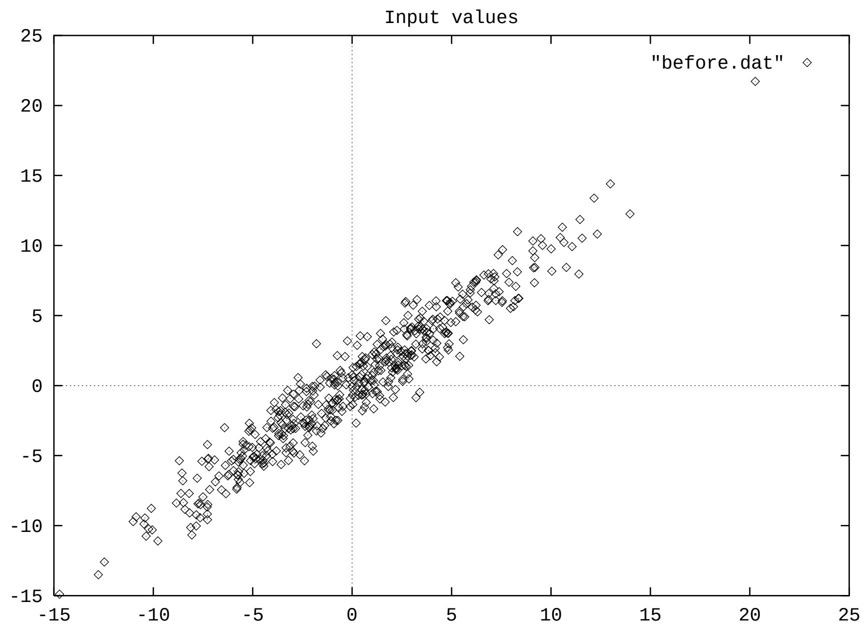
where I is the identity matrix.

This net will converge when the outputs are decorrelated (due to the off-diagonal anti-Hebbian learning) and when the expected variance of the outputs is equal to 1. i.e. this learning rule forces each network output to take responsibility for the same amount of information since the entropy of each output is the same.

This is generalisable to

$$\Delta w_{ij} = \alpha(\theta_{ij} - y_i y_j)$$

where $\theta_{ij} = 0$ for $i \neq j$. The value of θii for all i, will determine the variance on that output and so we can manage the information output of each neuron.



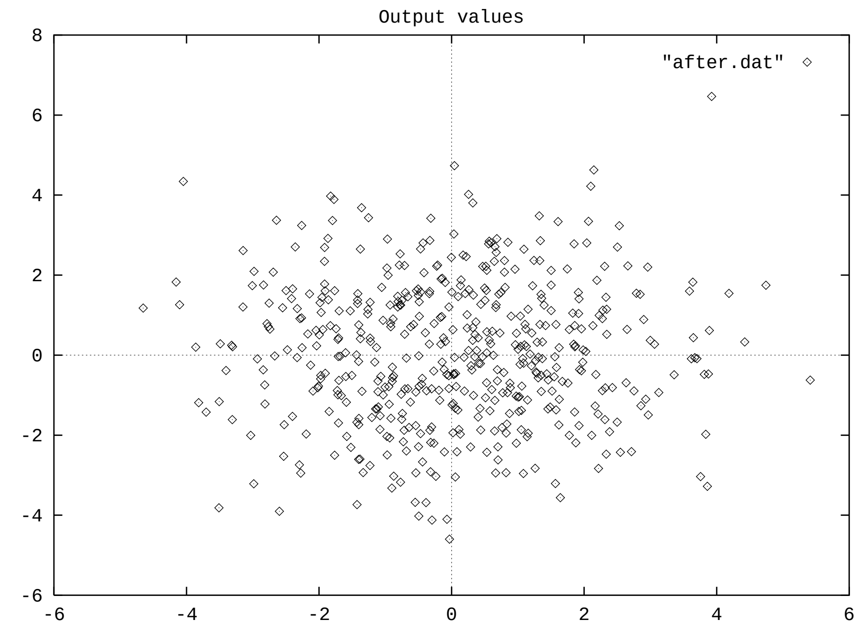


Figure 4.4: The original data is highly correlated. The output data has been decorrelated.

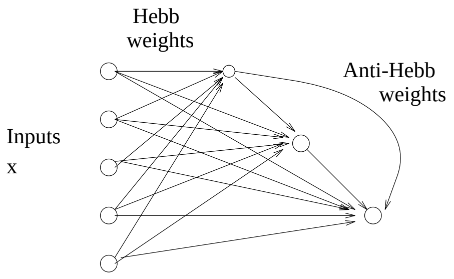


Figure 4.5: The weights from the input layer to the outputs are adjusted by Hebbian learing; meanwhile each output neuron is trying to turn off those succeeding it using weights which are adjusted by anti-Hebbian learing.

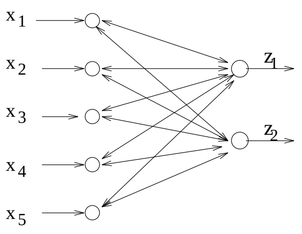


Figure 4.6: The negative feedback network.

4.3 Rubner and Schulten's Model

Several authors have combined Hebbian and Anti-Hebbian learning with a view to extracting all the principal components of a data set. One such example is shown in Figure 4.5.

Rubner and Schulten have a model in which the feedforward weights use either

- simple Hebbian learning with a renormalisation after each learning cycle or
- Hebbian learning of the one neuron Oja type

Either of which will guarantee to find the first principal component of the input data. However each output neuron is joined to every *subsequent* output neuron with a weight which is adjusted by anti-Hebbian learning. So the output of the second output neuron is determined by the Principal component of the input data but also by the need to decorrelate itself from the first output neuron. The net result is that it learns to respond to the second principal component. The third neuron is attempting in its turn to extract the maximum information from the data but also must decorrelate its output with the first two output neurons (which have learned the first two principal components); it then learns to respond maximally to the third principal component direction etc..

4.4 The Negative Feedback Model

The negative feedback network is shown in Figure 4.6. The input neurons are at the left hand side; activation is propagated through weights to the output neurons where a summation takes place and then the activation is fedback *as inhibition* to the input neurons. It can be shown that not only does the negative feedback stop the positive feedback loop in simple Hebbian networks (i.e. stops the weights from growing without bound) but it also causes the weights to converge to the Principal Components of the input data.

The rules governing the organisation of the network are

$$z_i = \sum_{j=1}^{N} w_{ij} x_j \tag{4.12}$$

$$\begin{aligned} x_j(t+1) &\leftarrow x_j(t) - \sum_{k=1}^{M} w_{kj} z_k & (4.13) \\ \Delta w_{ij} &= \alpha_t z_i x_j(t+1) & (4.14) \end{aligned}$$

where $x_p(t)$ is the firing of the p^{th} input neuron at time t.

There is no explicit weight decay, normalisation or clipping of weights in the model. The subtraction of the weighted sum of the output neuron values acts like anti-Hebbian learning. We will consider the network as a transformation from inputs x to outputs z; by considering the effects of these rules on individual neurons, we can show that the resultant network is equivalent to Oja's Subspace Algorithm.

Substituting (4.13) into (4.14) we get

$$\begin{aligned} \Delta w_{ji} &= \alpha x_i(t+1) z_j \\ &= \alpha (x_i(t) - \sum_k w_{ki} z_k) z_j \\ &= \alpha (x_i z_j - z_j \sum_k w_{ki} z_k) & (4.15) \end{aligned}$$

where we have dropped the time (t) notation for simplicity.

This last formulation of the learning rule (4.15) is exactly the learning rule for the Subspace Algorithm, Equation (3.14). The comparative results given in the various tables in Chapter 3 were from a negative feedback network.

4.4.1 Biological Interneurons

Because this network is similar to that found in biological networks, we have in the past called the network "The Interneuron Network": there are in the cortex negative feedback neurons called interneurons which inhibit the neurons which cause them to fire. The results of the last section have one major drawback when considered as a model of biological systems: the weights of the connections from the interneuron,z, to the summing neuron,y, are assumed to be identical to those from the summing neuron,y, to the interneuron,z. This is biologically implausible. We therefore have proposed a model where these weights are initially different and then learn independently from each other (albeit on the same data).

$$\begin{aligned} \mathbf{y} &= \mathbf{x} - \mathbf{V}\mathbf{z} & (4.16) \\ \mathbf{z} &= \mathbf{W}\mathbf{y} = \mathbf{W}\mathbf{x} & (4.17) \\ \mathbf{\Delta W} &= \alpha_w \mathbf{y}\mathbf{z}^T & (4.18) \\ \mathbf{\Delta V}^T &= \alpha_v \mathbf{y}\mathbf{z}^T & (4.19) \end{aligned}$$

where the initial values of both $\mathbf{V}^T$ and $\mathbf{W}$ are small random numbers not correlated in any way with each other.

Note that both learning rules for $\mathbf{W}$ and $\mathbf{V}$ are identical up to the learning rate and use only simple Hebbian learning. The results of an experiment identical to the last ones but with different feedforward and feedback weights is shown in Table 4.1.

4.4.2 Extensions to the Interneuron Network

The major difficulty with the basic interneuron network is the same as that which applies to the Oja's Subspace Algorithm - it finds only the subspace of the principal components not the principal component directions themselves.

Inter	neuron	model	VW	Model				
W			W			V		
1.000	-0.036	-0.008	**0.985**	-0.041	-0.003	**1.013**	-0.017	-0.024
0.036	**0.999**	-0.018	-0.019	**1.033**	0.031	-0.027	**0.965**	0.032
0.010	0.018	**1.000**	0.022	-0.032	**1.028**	0.020	-0.017	**0.969**
-0.002	-0.002	0.016	-0.024	-0.041	0.038	-0.007	-0.034	0.037
0.010	0.003	0.010	0.098	-0.007	-0.011	0.010	0.000	0.002

Table 4.1: Results from the interneuron network (left) with symmetric weights,W. and for the V and W vectors from the VW Model(see text)

Phased Creation of Interneurons

The first results in each table given when we discussed Oja's models are those from the negative feedback network. The second set of results (those which find the actual Principal Components, Table 3.3) are from a network using the following algorithm: the system is created with 1 interneuron; this interneuron finds the first principal component using the above learning rule. It then loses its plasticity i.e. its weights will not subsequently change. We then create a second interneuron. Since the first neuron has found and subtracted the first principal component, the second neuron will find the largest remaining principal component. It too now loses its plasticity. Then the third interneuron is created etc.. Therefore, we have introduced our asymmetry in the time dimension; note that whereas to do so with e.g. Oja's Single Neuron Network would have required the introduction of an extra mechanism - that of subtracting the projection of the data onto the subspace already found - we do not require this here as the network automatically finds and subtracts this subspace.

Four factors make the interneuron network especially exciting as a PCA network:

simplicity - there are no logistic or hyperbolic functions to be calculated; there is no additional computation within the learning rule; there is no sequential passing back of errors or decay terms.

homogeneity - every interneuron is performing exactly the same calculation as its neighbours; every summing neuron is performing exactly the same calculation as its neighbours.

locality of information - each interneuron uses only the information which it receives from its own connections; similarly with the summing neurons which calculate the y values

parallelism - each operation at each interneuron is independent of what is happening at any other interneuron; similarly with the summing neurons

Lateral Inhibition

However, the phased creation of neurons described in the last section does not utilise the inherent potential of this network for parallel information processing. We now develop learning algorithms which do this while retaining as much as possible of the other features. We amend the basic network by allowing the inhibitory effect of each interneuron to act on the other interneurons as well as the summing neurons.

The first type of network will be characterised by

$$\begin{aligned}
\mathbf{z}' &= \mathbf{W}\mathbf{x} \\
\mathbf{z} &= \mathbf{z}' - \mathbf{U}\mathbf{z}' \\
\mathbf{x} &\leftarrow \mathbf{x} - \mathbf{V}\mathbf{z} \\
\mathbf{\Delta W} &= \eta_w \mathbf{x}\mathbf{z}^T \\
\mathbf{\Delta V} &= \eta_v \mathbf{x}\mathbf{z}^T \\
\mathbf{\Delta U} &= \gamma \mathbf{z}\mathbf{z}^T
\end{aligned}$$

	Input 1	Input 2	Input 3	Input 4	Input 5
Interneuron 1	**0.552**	0	0.004	0.000	0.001
Interneuron 2	**0.700**	0	0.005	0.000	0.001
Interneuron 3	0.035	**0.991**	0.004	0.000	0.000
Interneuron 4	**0.450**	0	0.003	0.000	0.001

Table 4.2: A 5 input, 4 interneuron network with the same type of input data as previously

where $\mathbf{z}^{'}$ is the initial activation of the interneuron before receiving the lateral inhibition from other interneurons and $\mathbf{U}$ is the matrix of weights between the interneurons.

We do not however allow self-connections from interneurons to themselves. Two methods have been used with this amended network in order to create the necessary asymmetry:

- in the first, we allow the network weights to be upgraded at different rates;
- in the second, we use different activation functions to force convergence to the Principal Components.

We note that we have now a 3-phase operation:

1. The activation is fed forward from the summing neurons to the interneurons
2. The interneurons feed their activation to their peers and recalculate their activations
3. The activation is fed back to the summing neurons from the interneurons

While this is more computationally complex than before, we only require $O(m^2)$ additional calculations, where m is the number of interneurons. Further all learning processes continue to use simple Hebbian learning.

Now Oja has proved the importance of asymmetry in a network if you wish to find the actual Principal Components rather than the subspace of PCs. We can show that the lateral inhibition is not in itself sufficient to cause convergence to the actual Principal Components but it does however give us an alternative means of inserting the asymmetry (e.g. by having the lateral feedback affect the other neurons in an asymmetric manner or allowing the learning rates to be different).

Non-negative Weights

There is one obvious asymmetry used in nature which we have not used as yet: it is believed that signals from neurons may be excitatory or inhibitory but not both i.e. a neuron's output can excite (positively) other neurons or it can inhibit (negatively) other neurons; what cannot happen is that it excites some and inhibits others. The results reported in previous Chapters were based on a model where the weights were allowed to take any value positive or negative and so a neuron could be exciting some neurons while inhibiting others. In fact, it is possible for a neuron to switch from excitatory activation to inhibitory as its weight changes from positive to negative. If we allow only non-negative weights i.e. ensure that if a weight, while learning, never takes a negative value, we have the following interesting situation:

Assume that two weights of our converged network have values $\mathbf{w}_1 = a\mathbf{c_i} + b\mathbf{c_j}$ and $\mathbf{w}_2 = c\mathbf{c_i} + d\mathbf{c_j}$, where the $\mathbf{c}_p$ are the eigenvectors of the data's covariance matrix. Then since the weights converge to an orthogonal basis of the space, $ac + bd = 0$. Now if none of the terms a,b,c or d can be negative, then at least 2 must be zero (one from each term ac and bd). In other words, this constraint swings the weight vectors through the weight space to the actual Principal Components themselves. Since we are not directing the process, situations where several sets of weights converge to the same Principal Component tend to appear. An extreme example is shown in Table 4.2 in which we report the results of a simulation on the same type of data as previously but where the basic VW interneuron network was set up and the weights allowed to learn concurrently. Clearly, the weights of interneurons 1,2 and 4 have all converged to the same

Direction	1	2	3	4	5	Value
First PC	**0.584**	**0.811**	0.000	-0.002	-0.002	59.3
Second PC	-0.006	0.001	**-0.469**	**-0.617**	**-0.632**	33.7
Third PC	**-0.811**	**0.584**	-0.010	0.005	0.011	7.1
Fourth PC	0.012	-0.008	**-0.876**	**0.235**	**0.421**	2.4
Fifth PC	0.002	-0.001	**0.111**	**-0.751**	**0.650**	0.5

Table 4.3: Principal Components of the new data calculated using a standard statistical package

Interneuron 1	0.005	0.000	**0.465**	**0.616**	**0.635**
Interneuron 2	**0.391**	**0.518**	0.001	0.000	0.000
Interneuron 3	**0.324**	**0.467**	0.001	0.000	0.000
Interneuron 4	**0.296**	**0.409**	0.001	0.000	0.000

Table 4.4: A 5-4 interneuron circuit operating on the data of the previous table

Principal Component. Note that the weights marked only "0" have been stopped from becoming negative.

To further investigate the network's potential, data from a distribution whose Principal Components are shown in Table 4.3 was used as input to the network: it should be clear that there is a sharp division in the data between the first two directions and the last three. It might seem to be possible for the network to converge to a mixture of the above weights e.g. the directions {0.584,0,0.469,0,0} and {0,0.811,0,0.617,0.632} span the subspace of the first two Principal Components. This does not happen; the network converges to the first 2 Principal Components themselves (see analysis in the next Section).

It is impossible for the network using the positive weight constraint to converge to any direction containing a negative component i.e. from the third onwards. To find out how the network would respond to a situation where there were more degrees of freedom than possible directions to be found, we used the network with these 5 inputs and 4 interneurons (with the constraint that no weights are allowed to become negative). The results are shown in Table 4.4.

It is clear that the first interneuron has found the second Principal Component while the second, third and fourth interneurons have found the first Principal Component. This is a general finding with this type of network with the non-negative weight constraints.

This form of information extraction may be of importance if the data has been preprocessed in order to have isolated the "texture" data from the "colour" data from the "smell" data etc.. This type of distributed data-processing is known to happen in biological neural networks. However, this type of data-processing cannot be an initial data-processing function. The information must first be differentiated into disjoint dimensions: if there is any overlap between the dimensions in which the data exists, no more than one Principal Component per data set is possible.

We note that the length of the total vector of weights into interneurons 2, 3 and 4 is one unit. This is convenient in that it dispels the end of "grandmother cells"- that elusive neuron which would recognise only your grandmother. A grandmother cell is inconvenient in that damage to such a cell might lead to your never recognising your grandmother again. If such recognition is spread over a group of neurons such as is shown here, this provides a robustness in the network which has been missing up till now.

Asymmetry in Distance

Another possible model is suggested by the innate asymmetry in real biological neural networks in terms of the distances between neurons. This will manifest itself as different times to respond to a signal depending on the distance which the signal must travel (assuming that there is some uniformity in the speed of information transfer).

This differential is used in a new model where different interneurons take different lengths of times to respond to the input signal $\mathbf{x}$. Therefore while the activation from the input neurons

V			W		
1.000	0.006	-0.010	**1.000**	0.006	-0.010
-0.000	**-1.000**	0.013	-0.000	**-1.000**	0.013
0.012	0.023	**1.000**	0.012	0.023	**1.000**
0.000	-0.003	0.004	0.000	-0.002	0.004
-0.002	-0.004	-0.001	-0.002	-0.004	-0.001

Table 4.5: Results of the Differential Distance Model; each column shows the converged weights between one interneuron and the input neurons after learning on data from independent zero mean Gaussians with descending variances

is transmitted *to* all interneurons at the same time, each interneuron's response takes a different length of time to feedback to the input neurons. Thus the negative feedback is felt and used in a phased manner and learning takes place immediately the returned signal is received. Therefore, we embed the learning process in the feedback loop, so that we now postulate a learning and activation-transmission process which takes place in the order in which the following equations are given.

$$\text{initial value of } \mathbf{x} = \mathbf{x}(0) = \mathbf{x} \tag{4.20}$$

$$\mathbf{z} = W\mathbf{x} \tag{4.21}$$

$$\mathbf{x}(t) = \mathbf{x}(t-1) - \mathbf{v_i}(t-1)z_i \tag{4.22}$$

$$\Delta\mathbf{w_i}(t) = \alpha(t)z_i(t)\mathbf{x}^T(t) \tag{4.23}$$

$$\Delta\mathbf{v_i}(t) = \alpha(t)z_i(t)\mathbf{x}^T(t) \tag{4.24}$$

where e.g. $\mathbf{v_i}(t-1)$ indicates the value of the vector of weights $\mathbf{v_i}$ at the time t-1 into the i^{th} output neuron. The process (defined by Equations (4.22), (4.23) and (4.24)) is repeated for each interneuron in turn. This corresponds to the feedback from the interneurons being received at different times (perhaps depending on the physical distance which the activation must traverse, perhaps depending on the efficiency of transmission of the interneuron). This process results in the weights of the first (fastest) interneuron learning the first Principal Component, the second fastest interneuron learns the second Principal Component etc.. Experimental results from a network with 5 inputs and 3 interneurons are given in Table 4.5. In order to demonstrate the effect of the network, we have carried out our simulations on the same type of data as previously. Clearly the first 3 principal components have been found by the 3 interneurons. Note that the crucial difference between this model and previous models is the embedding of the learning process in the activation reception process. When this is done, the resulting network is more similar to a Sanger-type network rather than an Oja-type network. The k^{th} interneuron is learning to extract the maximum amount of information which is left after the previous (k-1) interneurons have extracted their information.

Equivalence to Sanger's Algorithm

Sanger's algorithm has, as a learning rule

$$\Delta w_{ij} = \alpha y_i (x_j - \sum_{k=1}^{i} y_k w_{kj})$$

in a totally feedforward architecture, where the outputs at y are given by

$$y_i = \sum_j w_{ij} x_j$$

We can show that the interneuron network using the rules determined by Equations 4.20 - 4.24 is equivalent to Sanger's algorithm:

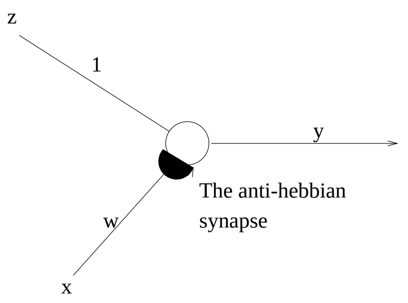


Figure 4.7: The synapse from x is modified using anti-Hebbian learning.

Let the **x** values be indexed with the time of feedback from the interneurons. Then,
$x_j(0)$ is the initial value of x_j at time 0. i.e. $x_j(0) = x_j$ originally
$x_j(1)$ is the value of x_j after receiving the feedback activation from the first (and hence closest) interneuron. i.e. $x_j(1) = x_j(0) - v_{1j}z_1$. Note that the time values are only ordinal indices - they do not imply equal intervals between feedback activations.

Similarly, if $x_j(2)$ is the value of x_j after receiving feedback from the first 2 interneurons, then

$$x_j(2) = x_j(1) - v_{2j}z_2 = x_j(0) - \sum_{k=1}^{2} v_{kj}z_k \tag{4.25}$$

In general, if $x_j(i)$ is the value of x_j after receiving feedback from the first i interneurons,

$$x_j(i) = x_j(0) - \sum_{k=1}^{i} v_{kj}z_k = x_j - \sum_{k=1}^{i} v_{kj}z_k \tag{4.26}$$

Therefore,

$$\begin{aligned} \Delta v_{ij} = \Delta w_{ij} &= \alpha x_j(i) z_i \\ &= \alpha(x_j(0) - \sum_{k=1}^{i} v_{kj}z_k)z_i \\ &= \alpha z_i(x_j - \sum_{k=1}^{i} v_{kj}z_k) \end{aligned}$$

4.5 The Anti-Hebbian Synapse as Energy Minimiser

Palmieri derives the anti-hebb synapse as a self-organising energy system using the following argument. Figure 4.7 shows a single idealised neuron with one linear synapse. The output of the neuron is given by

$$y = 1.z + w.x = z + wx \tag{4.27}$$

Notice that there is only one weight which can be modified. Anti-hebbian learning changes the strength of the synaptic connection proportional to the activation of the input x and the output y but with a negative sign.

$$\Delta w = -\alpha xy \tag{4.28}$$

We can view the anti-Hebbian synapse as attempting to turn off the neuron to which it is attached. If we treat x and z as random variables (each instances of distributions which may or may not be related to one another), the aim is to change the value of w so that we reach the minimum expected value of the squared output y i.e. we are changing w to reach $\min_w E(y^2)$, the minimum energy output. Now we can change w to get to this minimum by calculating

$$\begin{aligned}\frac{\partial E(y^2)}{\partial w} &= \frac{\partial}{\partial w}E((wx+z)^2)\\ &= 2E((wx+z)x)\\ &= 2(wE(x^2)+E(xz))\end{aligned}$$

To reach a minimum we change the weights proportional to the negative of this derivative i.e. $\frac{dw}{dt} \propto -\frac{\partial E(y^2)}{\partial w}$, and at a minimum, when $\frac{\partial E(y^2)}{\partial w} = 0$, we have the optimal w^* equal to

$$w^* = -\frac{E(xz)}{E(x^2)} \tag{4.29}$$

This assumes of course that such a minimum exists. We then have the minimum value of the output energy as

$$\begin{aligned}\min E(y^2) &= E(z^2) + 2w^*E(xz) + (w^*)^2E(x^2)\\ &= E(z^2) - \frac{E(xz).E(xz)}{E(x^2)}\\ &= E(z^2) + w^*E(xz)\end{aligned}$$

Note that, at the optimal value, w^*, we have x and y decorrelated since

$$E(xy) = E(x(w^*x+z)) = w^*E(x^2) + E(xz) = 0 \tag{4.30}$$

since $w^* = -\frac{E(xz)}{E(x^2)}$. So in attempting to minimise the output energy we have decorrelated the x and y values.

Chapter 5

Objective Function Methods

This chapter deals with a general class of methods which involve setting an objective function for the network and then optimising the value of that function using gradient based learning. The commonest type of objective function network uses minimisation of error as the the criterion and so such networks are said to be performing error descent. As we shall shortly see this is not the only possible criterion for ANNs.

Error descent methods are usually associated with supervised learning in which we must provide the network with a set of example data *and* the answer we expect to get when the network is presented with that data. So we present the input to the neural net, feed the activation forward through the weights currently in the network and then compare the actual output we get with the target output which we knew we wanted with this set of input data. We can then adjust the weights in the network so that the answer we wish to get (the target answer) is more likely next time the network is presented with this or similar data. The introductory tutorial we met in Chapter 1 was based on supervised learning. In this chapter we will use error descent sometimes in such a way that it may be considered an exercise in unsupervised learning e.g. by using the network for autoassociation which is sometimes described as self-supervision.

5.1 The Adaline

The Adaline is a simple one-layered neural network.

Let the P^{th} input pattern be $\mathbf{x}^P$, with corresponding output o^P and target t^P. So $o^P = \sum_j w_j x_j$ is the output of a single output neuron network. Then the sum squared error from using the Adaline on all training patterns is given by

$$E = \sum_P E^P = \frac{1}{2} \sum_P (t^P - o^P)^2 \tag{5.1}$$

where the fraction is included due to inspired hindsight. Now, if our Adaline is to be as accurate as possible, we wish to minimise the squared error. To minimise the error, we can find the gradient of the error with respect to the weights and move the weights in the opposite direction. If the gradient is positive, the error would be increased by changing the weights in a positive direction and therefore we change the weights in a negative direction. If the gradient is negative, in order to decrease the error we must change the weights in a positive direction. This is shown diagrammatically in Figure 5.1. Formally $\Delta_P w_j = -\gamma \frac{\partial E^P}{\partial w_j}$.

We say that we are searching for the Least Mean Square error and so the rule is called the LMS or Delta rule or Widrow-Hoff rule. Now, for an Adaline with a single output, o,

$$\frac{\partial E^P}{\partial w_j} = \frac{\partial E^P}{\partial o^P} \cdot \frac{\partial o^P}{\partial w_j} \tag{5.2}$$

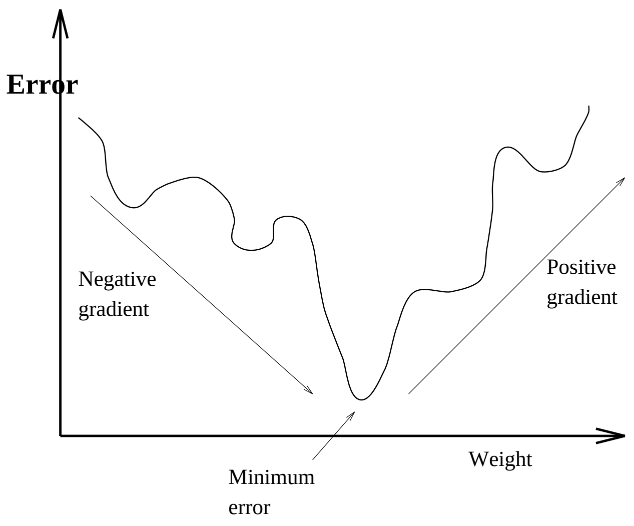


Figure 5.1: A schematic diagram showing error descent. In the negative gradient section, we wish to increase the weight; in the positive gradient section, we wish to decrease the weight

and because of the linearity of the Adaline units (i.e. $o = \sum_j w_j x_j$),

$$\begin{aligned} \frac{\partial o^P}{\partial w_j} &= x_j^P \\ \frac{\partial E^P}{\partial o^P} &= -(t^P - o^P) \text{ and so} \\ \Delta_P w_j &= \gamma(t^P - o^P).x_j^P \end{aligned}$$

This has proved to be a most powerful rule and is at the core of almost all current supervised learning methods. But it should be emphasised that nothing we have written has guaranteed that the method will cause the weights to converge i.e. for learing to cease. It can in fact be proved that this method will give the best (in a least mean square error sense) approximation to the function being modelled.

The tutorial we met in Chapter 1 walks through an example problem using the Delta rule.

5.2 The Backpropagation Network

An example of a multi-layered perceptron (MLP) is shown in Figure 1.3. Activity in the network is propagated forwards via weights from the input layer to the hidden layer where some function of the net activation is calculated. Then the activity is propagated via more weights to the output neurons. Now two sets of weights must be updated - those between the hidden and output layers and those between the input and hidden layers. The error due to the first set of weights is calculated using the Least Mean Square rule which we used in the Adaline network; however, now we require to propagate backwards that part of the error due to the errors which exist in the second set of weights and assign the error proportionately to the weights which cause it. You may see that we have a problem - **the credit assignment problem** - in that we must decide how much effect each weight in the first layer of weights has on the final output of the network. This assignment is the core result of the **backprop** method.

We may have any number of hidden layers which we wish since the method is quite general; however, the limiting factor is usually training time which can be excessive for many-layered networks. In addition, it has been shown that networks with a single hidden layer are sufficient to approximate any continuous function (or indeed any function with only a finite number of discontinuities) provided we use non-linear (differentiable) activation functions in the hidden layer.

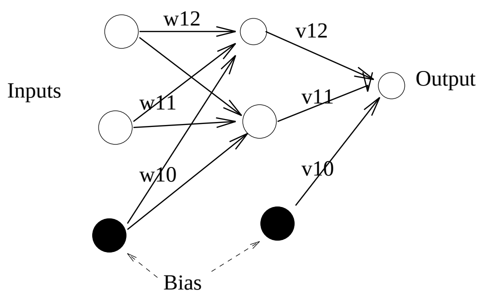


Figure 5.2: The net which will be used for the solution of the XOR problem using backpropagation

5.2.1 The Backpropagation Algorithm

A derivation of the backpropagation algorithm is given in an Appendix C. You will only require to know the algorithm not derive it.

The whole algorithm is now given in a 'how-to-do-it' form:

1. Initialise the weights to small random numbers
2. Choose an input pattern, $\mathbf{x}$, and apply it to the input layer
3. Propagate the activation forward through the weights till the activation reaches the output neurons
4. Calculate the δs for the output layer $\delta_i^P = (t_i^P - o_i^P)f'(Act_i^P)$ using the desired target values for the selected input pattern.
5. Calculate the δs for the hidden layer using $\delta_i^P = \sum_{j=1}^N \delta_j^P w_{ji}.f'(Act_i^P)$
6. Update all weights according to $\Delta_P w_{ij} = \gamma.\delta_i^P.o_j^P$
7. Repeat steps 2 to 6 for all patterns.

A final point is worth noting: the actual update rule after the errors have been backpropagated is local. This makes the backpropagation rule a candidate for parallel implementation.

5.2.2 The XOR problem

You will use the net shown in Figure C.1 to solve the XOR problem. The procedure is

Initialisation .

- Initialise the W-weights and V-weights to small random numbers.
- Initialise the learning rate, η to a small value e.g. 0.001.
- Choose the activation function e.g. tanh().

Select Pattern It will be one of only 4 patterns for this problem. Note that the pattern chosen determines not only the inputs but also the target pattern.

Feedforward to the hidden units first, labelled 1 and 2.

$$\begin{aligned} act_1 &= w_{10} + w_{11}x_1 + w_{12}x_2 \\ act_2 &= w_{20} + w_{21}x_1 + w_{22}x_2 \\ o_1 &= tanh(act_1) \\ o_2 &= tanh(act_2) \end{aligned}$$

Now feedforward to the output unit which we will label 3

$$\begin{aligned} act_3 &= v_{10} + v_{11}o_1 + v_{12}o_2 \\ o_3 &= tanh(act_3) \end{aligned}$$

Feedback errors calculate error at output

$$\delta_3 = (t - o_3) * f'(o_3) = (t - o_3)(1 - o_3^2)$$

and feedback error to hidden neurons

$$\begin{aligned} \delta_1 &= \delta_3 v_{11} f'(o_1) = \delta_3 v_{11}(1 - o_1^2) \\ \delta_2 &= \delta_3 v_{12} f'(o_2) = \delta_3 v_{12}(1 - o_2^2) \end{aligned}$$

Change weights

$$\begin{aligned} \Delta v_{11} &= \eta.\delta_3.o_1 \\ \Delta v_{12} &= \eta.\delta_3.o_2 \\ \Delta v_{10} &= \eta.\delta_3.1 \\ \Delta w_{11} &= \eta.\delta_1.x_1 \\ \Delta w_{12} &= \eta.\delta_1.x_2 \\ \Delta w_{10} &= \eta.\delta_1.1 \\ \Delta w_{21} &= \eta.\delta_2.x_1 \\ \Delta w_{22} &= \eta.\delta_2.x_2 \\ \Delta w_{20} &= \eta.\delta_2.1 \end{aligned}$$

Go back to Select Pattern

5.2.3 Backpropagation and PCA

Consider a two-layer (of weights) neural network such as in Figure 1.3. We are going to use such a network for autoassociation i.e. to train a network to associate each input with itself. We use a network with n inputs, n outputs and m hidden neurons where m<n. For autoassociation, we present the input data to the input neurons, propagate the data forward through weights to the hidden neurons and then through the output neuron's weights to the output neurons. The target pattern is then equal to the input pattern. We will show that such a network performs Principal Components Analysis of the data *when the network is a linear network* i.e. where each neuron merely performs a weighted summation of the inputs. Another way of describing this situation is to state that the activation function f() is the identity function i.e. f(Act) = Act.

Let us denote by A the matrix of weights from inputs to hidden neurons and by B the matrix of weights from hidden neurons to outputs. Then we have

$$\begin{aligned} h_i &= \sum_j a_{ij}x_j \\ y_i &= \sum_j b_{ij}h_j \\ \text{Then } \mathbf{y} &= BA\mathbf{x} \end{aligned}$$

Let the target pattern be $\mathbf{t}$ when the input pattern is $\mathbf{x}$. Then the error is given by

$$E_{A,B} = E((\mathbf{t} - BA\mathbf{x})^2) \tag{5.3}$$

So in the special case of autoassociation, we wish to minimise the mean reconstruction error

$$E_{A,B} = E((\mathbf{x} - BA\mathbf{x})^2) \tag{5.4}$$

We may write W = BA since we have only linear processes and so $\mathbf{y}$ = BA$\mathbf{x}$ = W$\mathbf{x}$. Now one definition of the principal component projection is that it is equal to that matrix P which minimises the mean square projection error i.e. which minimises

$$E((\mathbf{x} - P\mathbf{x})^2) \tag{5.5}$$

which is exactly our aim with the backprop rule above. This suggests that the linear backprop network converges only when BA=P and this can in fact be proved. Now A is m*n and B is n*m so that BA is n*n, a square matrix which from the above must have m columns equal to (a rotation of) the first m eigenvectors of the covariance matrix of the input data. But notice that this is not a uniquely determined minimum: the actual minimum is found for any rotation of the subspace.

So the minimum value of the function $E_{A,B}$ will be found when BA defines a projection of the data on the Principal Subspace of dimension m, where m is the number of hidden neurons. Notice it follows that the loss of information is equal to the projection of the data on the subspace we have discarded i.e. on the other n-m directions. So the information loss is

$$\sum_{i=m+1}^{n} \lambda_i \tag{5.6}$$

where λ_i is the i^{th} eigenvector of the data's covariance matrix.

5.3 Using Principal Components

As well as having neural networks which can find Principal Components, we can use PCs to facilitate a neural network's learning.

5.3.1 Preprocessing

The backpropagation network can be slow to converge. One of the reasons for this is that the weight changing process often consists of conflicting demands due to the interacting nature of the network weights on the error signal. This interaction is not relieved by momentum terms or adaptive learning rates.

One possibility is to use Principal Components Analysis as a preprocessing tool for backpropagation networks. I.e. we will project the input data onto its Principal Components before feeding it to the backpropagation network. Then we would have e.g. x[1] = the projection of the input data onto the first PC
x[2] = the projection of the input data onto the second PC etc
This is effective since the Principal Components are uncorrelated with each other and so the projections of the data on the PC directions are orthogonal to one another and so the learning for each weight interferes less with that of other weights. It can in fact be shown that the Hessian [1] of the error with respect to the weights is more nearly diagonal when we use principal component preprocessing and so we can use different learning rates appropriate to the size of the error in each PC direction: if a PC direction has a large error we use a large learning rate in this direction.

This type of preprocessing is of most use when we have high volume, high dimensional data when the PCA is used to cut the dimensionality of the data in such a way as to minimise information loss. Speech signal processing and image proceessing using backpropagation networks have both benefited from PCA preprocessing; in the former case, the processed signals have then been categorised as particular vowels; in the latter, face recognition has been a typical problem.

In both cases there has been a substantial speed up in the rate of learning.

[1]the matrix of second derivatives of the error with respect to the weights

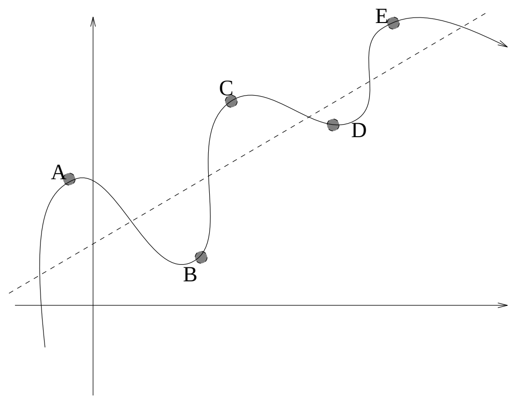


Figure 5.3: A set of data points may be approximated by either the straight line or the curve. Either would seem to fit the data; however, the line may give a better generalisation performance (on the test set) than the curve which is actually producing a lower error on the training set.

5.3.2 Principal Components Pruning

When training and testing a neural network, we usually have separate training and testing data sets. While we wish to see as good a performance as possible on the training set, we are even more interested in the network's performance on the test set; we already know how we wish the network to perform on the training set, what we are interested is how well it will perform on new data i.e. data which it has not seen during training. So we are looking for a measure of how well the network generalises. There is a trade-off between accuracy on the training set and accuracy on the test set.

Now a perfect memory of the patterns which are met during training is essentially a look-up table and look-up tables are discontinuous in that the item looked-up either is found to correspond to a particular result or not. Also generalisation is important not only because we wish a network to perform on new data which it has not seen during learning but also because we are liable to have data which is noisy, distorted or incomplete. Consider the set of 5 training points in Figure 5.3. We have shown two possible models for these data points - a linear model (perhaps the line minimising the squared error) and a polynomial fit which models the five given points exactly.

The problem with the more exact representation given by the curve is that it may be misleading in positions other than those directly on the curve. If a neural network has a large number of weights (each weight represents a degree of freedom), we may be in danger of overfitting the network to the training data which will lead to poor performance on the test data. To avoid this danger we may remove connections/weights but how do we decide which weights are the important ones and which are the ones which it is possible to remove without damaging the power of the network?

First attempts at pruning a network used the criterion of removing weights which had small absolute value. This however may remove important weights since such weights can have a crucial responsibility in correctly approximating the data set. In addition some weights may have a large magnitude but be unnecessary in that other weights also have learned the same information.

Levin, Leen and Moody have developed a method known as Principal Components Pruning:

1. Train the network using the backpropagation algorithm

2. Calculate the correlation matrix of the input training data.

3. Rank the Principal Components starting with those of largest eigenvalues.

4. Remove the lowest eigen-nodes. This is equivalent to projecting the data onto the Principal eigenvectors.

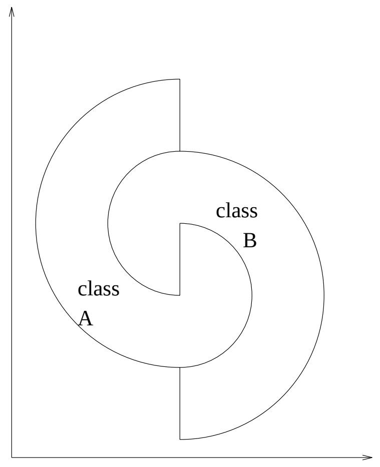


Figure 5.4: The Double Convex Regions.

5. Check the effect of the removal on the test data. If the validation error is not increased accept the removal. Otherwise not.

6. Project the weights out of the layer onto the Principal Subspace validated above. $W \leftarrow WC_lC_l^T$ where C_l is the matrix whose columns are the largest l eigenvectors of the correlation matrix.

7. Repeat this process through all layers till all eigennodes have been pruned.

Notice that when we prune an "eigen-node" we are not actually removing a node from the network, we are merely removing that part of the node's activation which is not a projection onto the first(largest) l Principal Components.

We can discuss the algorithm in the light of our previous analysis of linear backpropagation: we see that we are basically making sure that the network is capturing those directions in the input data which contain most information and we are simply slicing off those eigen-nodes which have responsibility for low information directions.

This provides a fast easily implementable algorithm.

The Double Convex Classification

One of the standard classification tasks for supervised learning networks is that of classifying points from two classes which interact in the way shown in Figure 5.4. The task is to train a backpropagation network to differentiate the two classes given examples from the classes. The task is a difficult one since the two classes are non-linearly separable.

Girolami used initially a network with 2 inputs (the x and y coordinates), H hidden neurons and 1 output neuron where the output has value 0 if the network classifies the input as belonging to class A and 1 if the network classifies the point as belonging to class B; however a network with two outputs was found to perform better. The value of H is the crucial factor: if H is too large the network will learn to model the individual points; if H is too small the network will not have enough power to model the distribution. It is possible to use trial and error to show that a value of about 45 is optimal for this problem.

Thus there are two inputs to the network comprising the x and y coordinate of the input and two outputs each of which represents one of the classes. Girolami showed that starting with a 2*90*2 network, there was virtually no increase in test error till the network had been pruned to under 50 eigennodes at which point the error began to increase substantially.

5.4 Cross-Entropy as the Objective Function

We have previously used only minimisation of error as the criterion to determine convergence of the backpropagation algorithm. However the algorithm is quite general: you can use it to maximise or minimise any objective function which is differentiable with respect to the parameters (weights) of the function.

Consider first the case in which we wish to use an ANN to distinguish between the members of two classes, Θ_1 and Θ_2. It would be possible to have two outputs, one for each class. But on the grounds of parsimony alone we will opt for a network with only a single output; we would like the output of that neuron to represent the posterior probability for one class, e.g. Θ_1 so that

$$\begin{aligned} P(\Theta_1|\mathbf{x}) &= y \\ P(\Theta_2|\mathbf{x}) &= 1-y \end{aligned}$$

where $\mathbf{x}$ is the input data. If we have a target coding scheme which states that t=1 if the input comes from class Θ_1 and t=0 if the input comes from class Θ_2, these can be combined into the single expression:

$$p(t|\mathbf{x}) = y^t(1-y)^{1-t} \tag{5.7}$$

remembering that $u^0 = 1, \forall u$.

Consider first if we have a data set comprising only two points, $(t_1, \mathbf{x}_1), (t_2, \mathbf{x}_2)$. Then we wish to maximise the probability of this distribution i.e. to maximise

$$p(t_1|\mathbf{x}_1 \text{ and } t_2|\mathbf{x}_2) = y_1^{t_1}(1-y_1)^{1-t_1} * y_2^{t_2}(1-y_2)^{1-t_2} \tag{5.8}$$

since these events are independent.

In general, if we have P different patterns, subscripted by the letter i so that y_i is the actual output when the target output is t_i then the probability of obtaining the complete set of training data and corresponding targets (assuming the training examples are drawn independently from the the distribution of examples) is

$$P(\text{training set}) = \prod_{i=1}^{P} y_i^{t_i}(1-y_i)^{1-t_i} \tag{5.9}$$

Our aim is to maximise this probability. We usually find it more convenient to work with the logarithm of this value which will be negative since all the individual values are ≤ 1. So we will attempt to minimise the negative of the logarithm of the above probability i.e. to minimise

$$\begin{aligned} E &= -\ln \prod_{i=1}^{P} y_i^{t_i}(1-y_i)^{1-t_i} \\ &= -\sum_{i=1}^{P} \{\ln (y_i)^{t_i} + \ln (1-y_i)^{(1-t_i)}\} \\ &= -\sum_{i=1}^{P} \{t_i \ln y_i + (1-t_i)\ln (1-y_i)\} \end{aligned}$$

by using gradient descent. The term E is known as the *cross entropy* and is the one we wish to minimise.

Now if we differentiate this error function with respect to y_i (as required by the backpropagation algorithm) we get

$$\begin{aligned} \frac{\partial E}{\partial y_i} &= -(\frac{t_i}{y_i} - \frac{1-t_i}{1-y_i}) \\ &= -\frac{t_i - t_i y_i - y_i + t_i y_i}{y_i(1-y_i)} \\ &= \frac{y_i - t_i}{y_i(1-y_i)} \end{aligned}$$

and so the absolute minimum of E occurs when $y_i = t_i, \forall i$ since this is the point at which $\frac{\partial E}{\partial y_i} = 0$. The double helix classification problem of the last section would be one which could use this objective function easily.

A nice property of this error function is that if we are using the logistic function $y = g(a) = \frac{1}{1+\exp(-a)}$ we find the derivative of the error with respect to the activation, a is

$$\delta_i = \frac{\partial E}{\partial a_i} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial a_i} = \frac{y_i - t_i}{y_i(1-y_i)}.y_i(1-y_i) = y_i - t_i \tag{5.10}$$

where we have used the special properties of the logistic function that $g'(a) = g(a)(1-g(a))$. This very simple form of the δ_i which is the term which is backpropagated in the algorithm makes this measure particularly appealing.

At the minimum, (where $t_i = y_i$), the cross-entropy is equal to

$$E = -\sum_i \{t_i \ln t_i + (1-t_i)\ln(1-t_i)\} \tag{5.11}$$

It is possible to extend this measure of classification to both several classes and to continuous distributions. For example, if we have a continuous distribution in which p($\mathbf{x}$) is the probability distribution of the target and q($\mathbf{x}$) is the probability distribution of the output given the input $\mathbf{x}$, the general measure of cross-entropy is given by

$$\begin{aligned} E &= -\int p(\mathbf{x})\ln q(\mathbf{x}) \\ &= -\int p(\mathbf{x})\ln q(\mathbf{x}) + \int p(\mathbf{x})\ln p(\mathbf{x}) - \int p(\mathbf{x})\ln p(\mathbf{x}) \\ &= -\int p(\mathbf{x})\ln\frac{q(\mathbf{x})}{p(\mathbf{x})} - \int p(\mathbf{x})\ln p(\mathbf{x}) \\ &= D(p||q) + H(p) \end{aligned}$$

From this we can see that minimising the cross-entropy between target and actual outputs is equivalent to minimising the Kullback-Leibler distance between the distributions (given that the entropy of the target distribution, the last term in the above sequence, is unchanged by the learning procedure). In other words, we are making the output probabilities as close as possible to the probability distribution of the target values.

Finally we note that this error function requires an output between 0 and 1 and so a sigmoid (logistic) function at the output layer is needed as well as a sigmoid at the hidden layer.

5.5 The I-Max Model

Becker and Hinton have an interesting model using error descent learning. They begin with the question "What is the best way to model the input distribution to encode interesting features in the data" and come up with the observation that they wish to constrain the learning problem by restricting the features of interest to those which are liable to be useful for later perceptual processing. In a general non-specific environment, there are regularities ("coherence") in that any part of the environment is very likely to be predictable from other close parts of the environment e.g. any object has a finite compact surface area and so there exists a set of points all physically close to one another which share visually similar features. Similarly there exists temporal coherence in our environment - only in Star Trek can objects vanish from one location and reappear in another. Also there is a coherence across sensory modalities - we generally see and smell and feel an orange at a single instant in time. This suggests that we should use coherence to extract information from the input data; one objective that might be appropriate for a network would be the extraction of redundancy (which gives rise to coherence) in raw sensory data since we do not, for example, have to use sight, smell and touch of an orange in order to identify the orange.

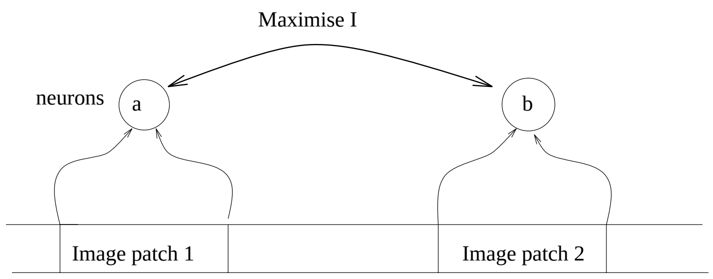


Figure 5.5: Two neurons receive inputs from adjacent parts of the image. Learning attempts to maximise the mutual information between the neurons' outputs.

This is shown diagrammatically in Figure 5.5 in which the two output neurons are attempting to reduce the redundancy in their outputs based on their reaction to the input data which comes from the same source.

We could perform error descent on the squared error of the difference between the outputs but one difficulty with this is that the network could simply learn to output a constant value at both neurons a and b. So we need to force the neurons to extract as much information as possible but still ensure that they are agreeing. This suggests that the optimisation criterion should be to maximise the mutual information between the two neurons

$$\begin{aligned} I_{a,b} &= H(a) + H(b) - H(a,b) \\ &= H(a) - H(a|b) \\ (\text{or} &= H(b) - H(b|a)) \end{aligned}$$

Written this way we can see that by maximising the mutual information between the neurons, we are maximising the entropy (the expected information output) of each neuron while minimising the conditional entropy (the uncertainty left about each neuron's output) given the other's value. So we wish each neuron to be as informative as possible while also telling us as little as possible about the other neuron's outputs.

Now if we have two binary (1/0) probabilistic units, we can estimate the mutual information by sampling their activity over a large set of input cases. This gives us an estimate of their individual and joint probabilities and so the mutual information can be calculated. If the expected output of the j^{th} output neuron on the K^{th} training example is s_j^K i.e. $s_j = E(y_j)$ when the input pattern is K, Becker and Hinton show that the partial derivative of the mutual information with respect to the expected output on training case K is

$$\frac{\partial I_{y_i;y_j}}{\partial s_i^K} = -P^K (\log \frac{s_i}{s_{\bar{i}}} - s_j^K \log \frac{s_{ij}}{s_{\bar{i}j}} - s_{\bar{j}} \log \frac{s_{i\bar{j}}}{s_{\overline{ij}}}) \tag{5.12}$$

but I will not expect you to memorise this formula! Notice that here P^K is the probability of training case K and $s_{\bar{i}} = E(1 - y_i)$ etc.

The point which we wish to highlight is that we have a method of changing the weights (e.g. by least mean square learning) to maximise the mutual information.

$$\Delta w = \propto \frac{\partial I}{\partial w} = \frac{\partial I}{\partial s} . \frac{\partial s}{\partial w} \tag{5.13}$$

It is possible to extend the IMax algorithm to both continuous variables and to multi-valued spatially coherent features; the latter is interesting in that it uses a general method of ensuring that every output is a valid probability between 0 and 1. If we have a discrete random variable A$\in \{a_1, ..., a_N\}$, we can define the probability of the i^{th} event as

$$P(A = a_i) = \frac{\exp{(x_i)}}{\sum_{j=1}^{N} \exp{(x_j)}} \tag{5.14}$$

which is guaranteed to give a value between 0 and 1 and to ensure that the probabilities sum to 1.

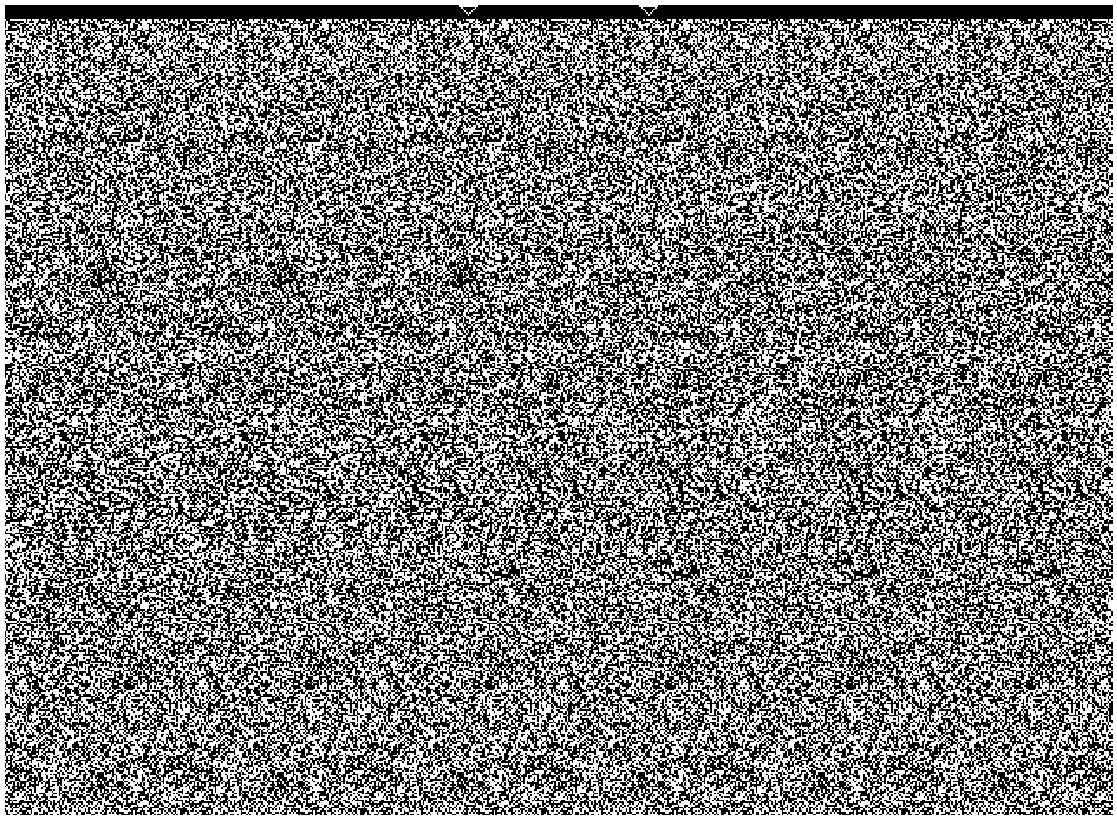


Figure 5.6: The Starship Enterprise.

5.5.1 An Example

Becker has used this type of network on simplified random-dot stereograms. A random dot stereogram is shown in Figure 5.6. We can focus on the Enterprise by allowing our eyes to focus beyond the plane of the picture and get the coherence in the image from the non-coherent random dot stereogram; the input to each eye is maximally informative about the input to the other when we focus on the starship. However the randomness need not be in dot format(Figure 5.7).

Becker has performed a number of experiments such as having networks learn particular frequnces from a number of different sets of frequencies or finding the coherence in a random dot stereogram image. One of the simplest however was using a set of simple, binary random-binary stereograms such as shown in Figure 5.8a.

Each input vector consists of a one dimensional random binary strip which corresponds to the right image and a shifted version of this which corresponds to the left image. The right image in each case is totally random and the left image is generated by choosing a randomly chosen global shift. The only local property is the invariant between images shift corresponding to the depth in a random dot stereogram. Now the neurons are attempting to maximise their mutual information and so the only way that they can do this is by representing the shift information. The multi-layered version is shown in Figure 5.8b. Two different global shifts were used - one pixel forward or one backward.

Becker found that if she used a small training set, the neurons learned features about the training set as well as the global shift. Since the features of the training set were randomly generated this is irrelevant information. As she increased the size of the training set, the network learned only the global shift. The probability that only the shift will be learned can be increased by increasing the number of receptive fields while the lower level neurons can respond to more local features.

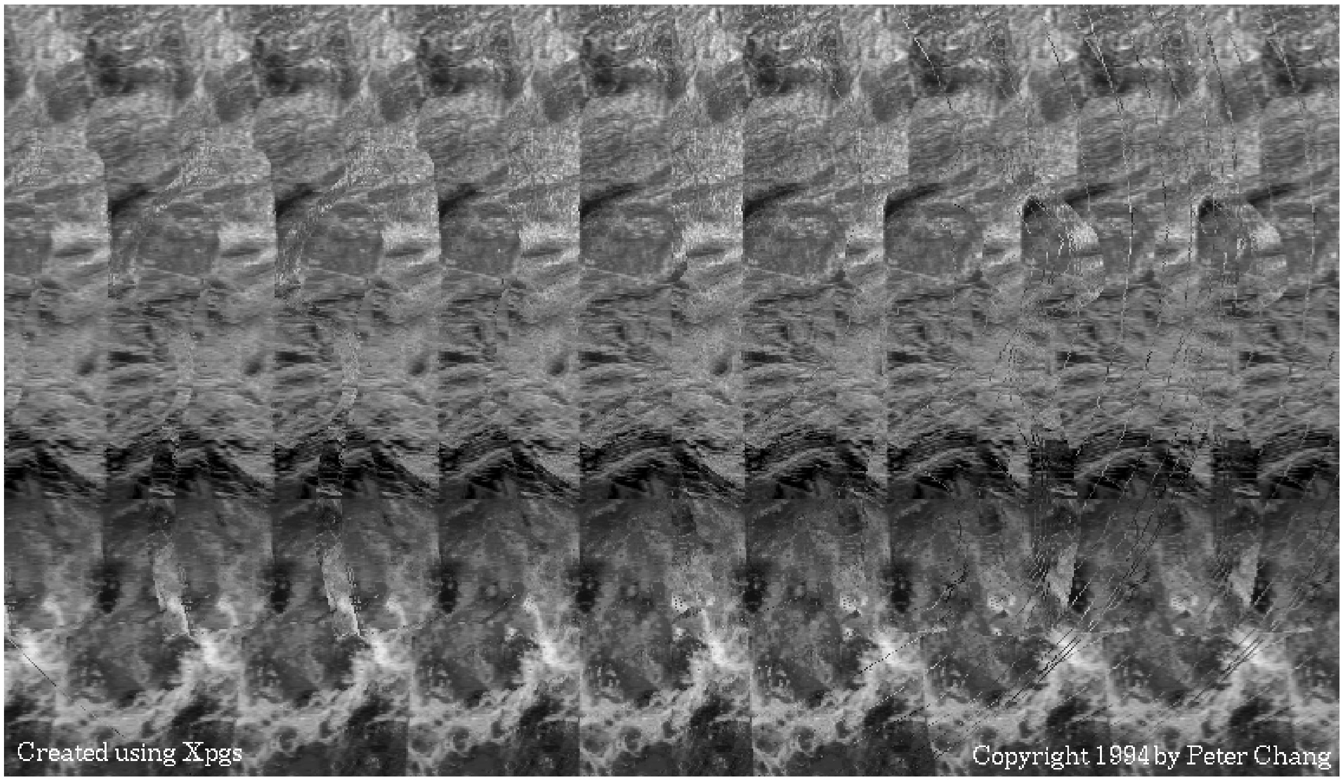


Figure 5.7: Extraction of coherent information

However since shift is a higher order feature of the data - it cannot be learned from any single pair of inputs - we require a multi-layered perceptron to accurately detect the shift operator. In this case the best results are achieved by maximising the mutual information at the top layer only and then backpropagating the error to the first layers. The output neurons are in a position to respond to global features.

If we simply extend the architecture in Figure 5.8a, to multiple receptive fields with one unit receiving input from each receptive field, each neuron now tries to maximise the sum of its pairwise mutual information from each of the other neurons. Interestingly, once one pair of neurons "catches on" to the shift feature, their mutual information becomes very large and convergence accelerates which provides a stronger shift-tuned signal for the other neurons and helps their convergence.

5.6 Contextual Information

Kay and Phillips[2] have developed a neural network which attempts to use contextual information to help extract information from input data. It is clear that we use such information e.g. to give ourselves clues about the exact meaning of the input data. Since their interest is in modelling biological processes, they use only local processing and investigate ways of helping neurons distinguish relevant from irrelevant information at early stages of processing.

They wish neurons to use contextual information in a way that is not confused with the input data and so have two entirely different sets of inputs to each neuron: the normal feedforward data connections and lateral context connections.

They envisage a network with m receptive field units and n contextual field units and have a probabilistic mechanism for the firing of the neuron. If the inputs are **x** and the contexts are **c**

[2]working in the University of Stirling, to give you contextual information.

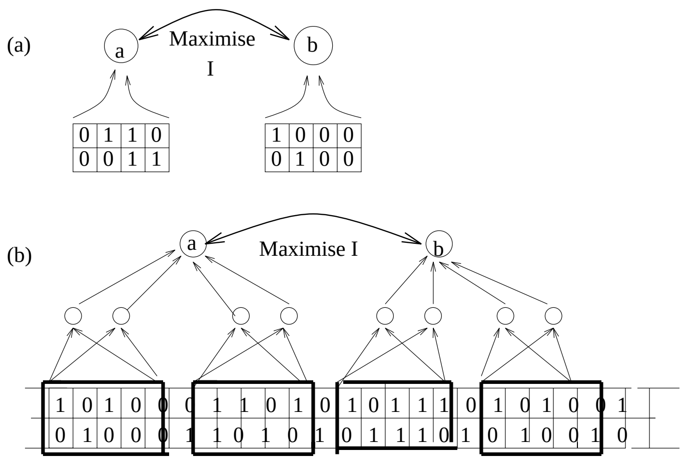


Figure 5.8: In (a), two neurons recieve as input random binary patterns in which the left half is a shifted version of the right half. The inputs to the two neurons contains no other information (other than the common shift). The learning algorithm adjusts the weights of the network to maximise the mutual information between neurons over the whole set of training patterns.
In (b), the first layer of weights is trained to extract some low order features and the next layer hierarchically combines these into more accurate feature detectors.

then they define the activation derived from each by two separate values

$$\begin{aligned} s_x &= \sum_{i=1}^{m} w_i x_i \\ s_c &= \sum_{i=1}^{n} v_i c_i \end{aligned}$$

They now wish to combine these inputs but in a way that allows the normal input to drive the firing of the output neuron but allows the context to affect the magnitude of the output neuron's response. Specifically, the activation function is such that

1. If the weighted sum of inputs, s_x, is zero there is no output.
2. If the weighted sum of contexts, s_c, is zero, the output is the weighted sum of inputs, s_x.
3. If the contexts and inputs agree (the sign of s_x and s_c agree) the output should be greater than that produced by the inputs alone.
4. If the contexts and inputs disagree(the sign of s_x and s_c are different) the output should be less than that which would be produced by the inputs alone.
5. Only the inputs should determine the sign of the output i.e. the context cannot change the direction (positive or negative) of the output decision.

They use the following activation function:

$$f(s_x, s_c) = \frac{1}{2} s_x (1 + \exp(2 s_x s_c)) \tag{5.15}$$

which has the above qualities. Now they use a binary probabilistic neuron which is more likely to fire the larger the above function is:

$$p = P(Y = 1 | X = \mathbf{x}, C = \mathbf{c}) = \frac{1}{1 + \exp(-f(s_x, s_c))} \tag{5.16}$$

Since the neuron is binary, its output is either 1 or 0. If the value of the activation function is large, the probability of firing (given the current input $\mathbf{x}$ and context $\mathbf{c}$) is large since the bottom line of the right hand side will then tend to 1. If on the other hand the activation function is largely negative the probability will tend to 0. Finally if the activation function is around 0 (which can only happen when s_x is around zero or the context is strongly disagreeing with the inputs) the probability of firing is around $\frac{1}{2}$. The probability density function of firing is shown in Figure 5.9 as a function of both inputs and context.

5.6.1 Information Transfer and Learning Rules

We see from Figure 5.10 the relationships between the various sets of information possible. We can require a number of different objective functions to be optimised depending on our particular criterion of success. For example, we may require the network to minimise the uncertainty in the distribution of outputs given the inputs and context units information. This is equivalent to minimisation of $H(Y|X, C)$. Now this conditional entropy can be calculated since we know that the probability that a particular neuron will fire is p. Thus

$$H(Y|X, C) = -E(p \log p + (1 - p) \log(1 - p))_{X,C} \tag{5.17}$$

where the expectation is taken over the joint distribution of inputs, X, and contexts, C. Now we calculate

$$\begin{aligned} \frac{\partial H(Y|X, C)}{\partial w} &= -E(p(1 - p) \log \frac{p}{1 - p} \frac{\partial f}{\partial s_x} X)_{X,C} \\ \frac{\partial H(Y|X, C)}{\partial v} &= -E(p(1 - p) \log \frac{p}{1 - p} \frac{\partial f}{\partial s_c} C)_{X,C} \end{aligned}$$

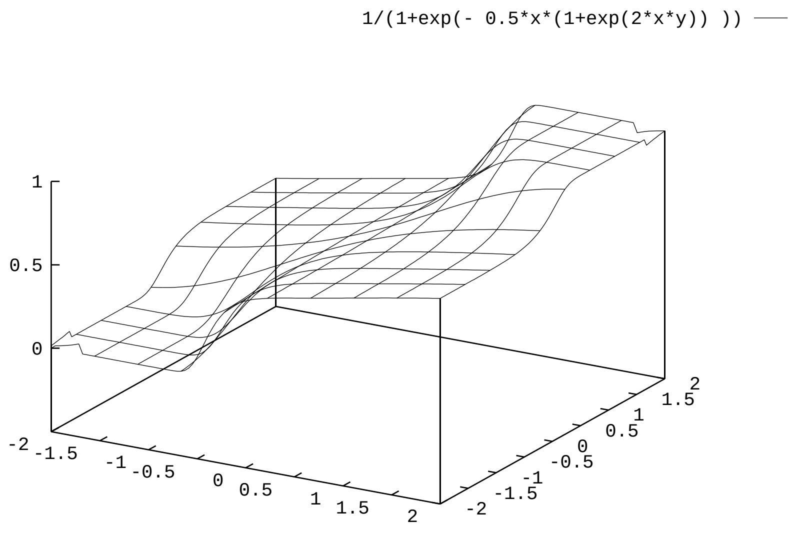


Figure 5.9: The probability function associated with the embedding of contextual information in input activation.

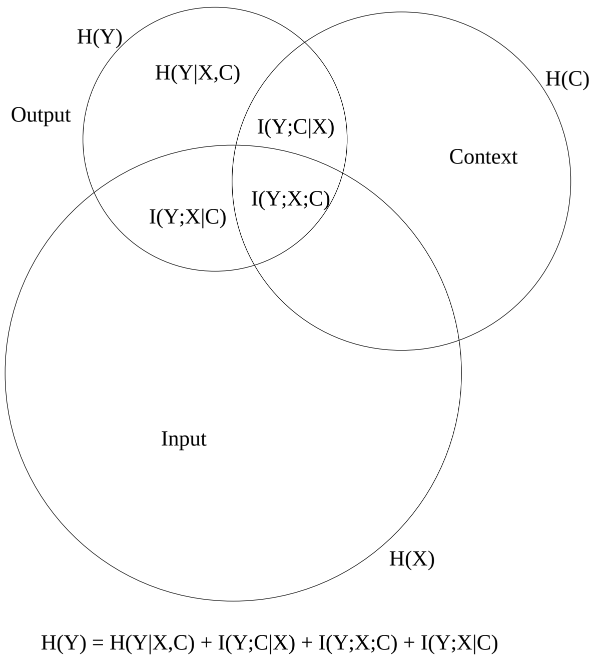


Figure 5.10: The information in the output can be decomposed into the various conditional and unconditional mutual informations and the residual entropy.

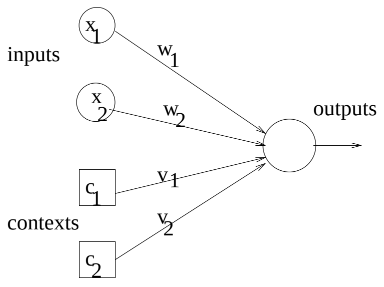


Figure 5.11: A simple network showing a single output neuron with two inputs and two inputs from contextual units.

w_1	w_2	v_1	v_2
1.2	0	0.75	0

Table 5.1: The trained network has learned to associate the input with the context with which it was correlated.

Again I do not expect you to memorise these equations but you should know the general method.

Kay and Phillips in fact go further and define a class of optimisation criteria based on

$$F = I(Y;X;C) + a_1 I(Y;X|C) + a_2 I(Y;C|X) + a_3 H(Y|X,C) \tag{5.18}$$

By adjusting the various parameters, a_1, a_2, a_3 we can adjust the importance of the various sub-criteria which make up F. They extend the above derivation of $\frac{\partial H(Y|X,C)}{\partial w}$ and $\frac{\partial H(Y|X,C)}{\partial v}$ to the other sub-criteria which provides a general model for optimising the learing rules of the network in the usual way.

5.6.2 Results

Consider the network shown in Figure 5.11. We have two bipolar (1/-1) input neurons and two bipolar (1/-1) context inputs thus giving 16 possible patterns. Only the X_1 and C_1 neurons' values were correlated. We select one of the simplest of the family of objective functions described above, that of maximising $F = I(Y;X;C)$ i.e. maximising the joint information between inputs, output and context units. When we set the correlation between C_1 and X_1 to be 0.8 (a very high correlation), we find that the weights in Table 5.1 are learned.

Essentially the network has learned the correlation between the input and context unit which were correlated but has ignored the non-information in the other two units. In this case any new data is more liable to be classified in the first class if it also is presented with the same context. If we create data such that a context unit and an input have negative correlation, this is reflected in the weights - one being positive, the other negative while, as before, the other two remain about 0.

5.7 The Paisley Connection

5.7.1 Introduction

Canonical Correlation Analysis is used when we have two data sets which we believe have some underlying correlation. Consider two sets of input data, from which we draw iid samples to form a pair of input vectors, $\mathbf{x}_1$ and $\mathbf{x}_2$. Then in classical CCA, we attempt to find the linear combination of the variables which gives us maximum correlation between the combinations. Let

$$y_1 = \mathbf{w}_1 \mathbf{x}_1 = \sum_j w_{1j} x_{1j} \tag{5.19}$$

$$y_2 = \mathbf{w}_2\mathbf{x}_2 = \sum_j w_{2j}x_{2j} \tag{5.20}$$

Then we wish to find those values of $\mathbf{w}_1$ and $\mathbf{w}_2$ which maximise the correlation between y_1 and y_2. Whereas Principal Components Analysis and Factor Analysis deals with the interrelationships within a set of variables, CCA deals with the relationships between two sets of variables. If the relation between y_1 and y_2 is believed to be causal, we may view the process as one of finding the best predictor of the set $\mathbf{x}_2$ by the set $\mathbf{x}_1$ and similarly of finding the most predictable criterion in the set $\mathbf{x}_2$ from the $\mathbf{x}_1$ data set. Thus, we later review a data set in which a set of exam results are split into those achieved by students when they had access to their books and those marks obtained when the students were denied their books during the exam. We might wish to use a student's open book exams to predict how well he/she might do in the closed book exams.

One way to view canonical correlation analysis is as an extension of multiple regression. Recall that in multiple regression analysis the variables are partitioned into an $\mathbf{x}_1$-set containing q variables and a x_2-set containing p =1 variable. The regression solution involves finding the linear combination of $\mathbf{x}_1$ which is most highly correlated with x_2.

Let $\mathbf{x}_1$ have mean μ_1 and $\mathbf{x}_2$ have mean μ_2. Then the standard statistical method lies in defining

$$\Sigma_{11} = E\{(\mathbf{x}_1 - \mu_1)(\mathbf{x}_1 - \mu_1)^T\} \tag{5.21}$$
$$\Sigma_{22} = E\{(\mathbf{x}_2 - \mu_2)(\mathbf{x}_2 - \mu_2)^T\} \tag{5.22}$$
$$\Sigma_{12} = E\{(\mathbf{x}_1 - \mu_1)(\mathbf{x}_2 - \mu_2)^T\} \tag{5.23}$$
$$\text{and } K = \Sigma_{11}^{-\frac{1}{2}}\Sigma_{12}\Sigma_{22}^{-\frac{1}{2}} \tag{5.24}$$

where T denotes the transpose of a vector. We then perform a Singular Value Decomposition of K to get

$$K = (\alpha_1, \alpha_2, ..., \alpha_k)D(\beta_1, \beta_2, ..., \beta_k)^T \tag{5.25}$$

where α_i and β_i are the standardised eigenvectors of KK^T and K^TK respectively and D is the diagonal matrix of eigenvalues.

Then the first canonical correlation vectors (those which give greatest correlation) are given by

$$\mathbf{w}_1 = \Sigma_{11}^{-\frac{1}{2}}\alpha_1 \tag{5.26}$$
$$\mathbf{w}_2 = \Sigma_{22}^{-\frac{1}{2}}\beta_1 \tag{5.27}$$

with subsequent canonical correlation vectors defined in terms of the subsequent eigenvectors, α_i and β_i.

In this chapter, we present a neural implementation of CCA which adaptively learns the optimal weights to maximise correlations between the data sets.

5.7.2 The Canonical Correlation Network

The input data comprises two vectors $\mathbf{x}_1$ and $\mathbf{x}_2$. Activation is fed forward from each input to the corresponding output through the respective weights, $\mathbf{w}_1$ and $\mathbf{w}_2$ (see Figure 5.12 and equations (5.19) and (5.20)) to give outputs y_1 and y_2.

We wish to maximise the correlation $E(y_1y_2)$ where $E()$ denotes the expectation which will be taken over the joint distribution of $\mathbf{x}_1$ and $\mathbf{x}_2$. We may regard this problem as that of maximising the function $g_1(\mathbf{w}_1|\mathbf{w}_2) = E(y_1y_2)$ which is defined to be a function of the weights, $\mathbf{w}_1$ given the other set of parameters, $\mathbf{w}_2$. This is an unconstrained maximisation problem which has no finite solution and so we must constrain the maximisation. Typically in CCA, we add the constraint that $E(y_1^2 = 1)$ and similarly with y_2 when we maximise $g_2(\mathbf{w}_2|\mathbf{w}_1)$. Using the method of Lagrange

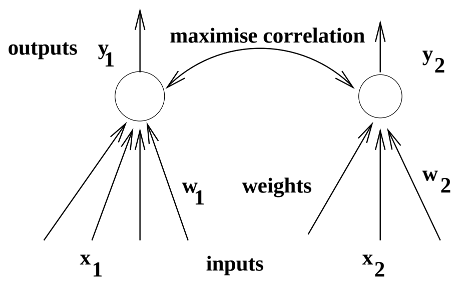


Figure 5.12: The CCA Network. By adjusting weights, w_1 and w_2, we maximise correlation between y_1 and y_2
.

multipliers, this yields the constrained optimisation functions,

$$\begin{aligned} J_1 &= E(y_1y_2) + \frac{1}{2}\lambda_1(1-y_1^2) \text{ and} \\ J_2 &= E(y_1y_2) + \frac{1}{2}\lambda_2(1-y_2^2) \end{aligned}$$

We may equivalently use

$$J = E(y_1y_2) + \frac{1}{2}\lambda_1(1-y_1^2) + \frac{1}{2}\lambda_2(1-y_2^2)$$

but it will be more convenient in the following sections to regard these as separate criteria which can be optimised independently by implicitly assuming that $\mathbf{w}_1$ is constant when we are changing $\mathbf{w}_2$ and vice-versa. We wish to find the optimal solution using gradient ascent and so we find the derivative of the instantaneous version of each of these functions with respect to both the weights, $\mathbf{w}_1$ and $\mathbf{w}_2$, and the Lagrange multipliers, λ_1 and λ_2. By changing the Lagrange multipliers in proportion to the derivates of J we are changing the relative strength of the constraint compared to the function we are optimising; this allows us to smoothly maximise that function in the region in which we are satisfying the constraint.

Noting that

$$\frac{\partial g_1(\mathbf{w}_1|\mathbf{w}_2)}{\partial \mathbf{w}_1} = \frac{\partial(y_1y_2)}{\partial \mathbf{w}_1} = \frac{\partial(\mathbf{w}_1\mathbf{x}_1y_2)}{\partial \mathbf{w}_1} = \mathbf{x}_1y_2 \tag{5.28}$$

these yield respectively

$$\begin{aligned} \frac{\partial J_1}{\partial \mathbf{w}_1} &= \mathbf{x}_1y_2 - \lambda_1y_1\mathbf{x}_1 = \mathbf{x}_1(y_2 - \lambda_1y_1) \\ \frac{\partial J_1}{\partial \lambda_1} &\propto (1-y_1^2) \end{aligned}$$

Similarly with the J_2 function, $\mathbf{w}_2$ and λ_2. This gives us a method of changing the weights and the Lagrange multipliers on an online basis. We use the joint learning rules

$$\begin{aligned} \Delta w_{1j} &= \eta x_{1j}(y_2 - \lambda_1y_1) \\ \Delta\lambda_1 &= \eta_0(1-y_1^2) \\ \Delta w_{2j} &= \eta x_{2j}(y_1 - \lambda_2y_2) \\ \Delta\lambda_2 &= \eta_0(1-y_2^2) \end{aligned} \tag{5.29}$$

where w_{1j} is the j^{th} element of weight vector, $\mathbf{w}_1$ etc. It has been found empirically that best results are achieved when $\eta_0 >> \eta$.

However, just as a neural implementation of Principal Component Analysis (PCA) e.g. the Subspace Algorithm may be very interesting but not a generally useful method of finding Principal Components, so a neural implementation of CCA may be only a curiosity. However, it has been shown that nonlinear extensions of PCA networks are able to search for the independent components of a data set (ICA) and that such extensions are therefore justifiable as engineering tools for investigating data sets. We therefore extend our neural implementation of CCA by maximising the correlation between outputs when such outputs are a nonlinear function of the inputs. We investigate a particular case of maximisation of $E(y_1 y_2)$ when the values y_i are a nonlinear function of the inputs, $\mathbf{x}_i$.

For the nonlinear optimisation, we use e.g. $y_3 = \sum_j w_{3j} f_3(v_{3j} x_{1j}) = \mathbf{w}_3 \mathbf{f}_3$ where the function $f_3()$ is applied on an element-wise basis to give the vector $\mathbf{f}_3$.

The equivalent optimisation function for the nonlinear problem is then

$$\begin{aligned} J_3(\mathbf{w}_3|\mathbf{w}_4) &= E(y_3 y_4) + \frac{1}{2}\lambda_3(1 - y_3^2) \\ J_4(\mathbf{w}_4|\mathbf{w}_3) &= E(y_4 y_3) + \frac{1}{2}\lambda_4(1 - y_4^2) \end{aligned}$$

One solution is discussed in section 5.7.8.

5.7.3 Experimental Results

We report simulations on both real and artificial data sets of increasing complexity. We begin with data sets in which there is a linear correlation and we demonstrate the effectiveness of the network on Becker's random dot stereogram data. We then extend the method in two ways not possible with standard statistical techniques:

1. We maximise correlations between more than two input data sets.

2. We consider maximising correlations where such correlations may be on nonlinear projections of the data.

5.7.4 Artificial Data

Our first experiment comprises an artificial data set: $\mathbf{x}_1$ is a 4 dimensional vector, each of whose elements is drawn from the zero-mean Gaussian distribution, N(0,1); $\mathbf{x}_2$ is a 3 dimensional vector, each of whose elements is also drawn from N(0,1). In order to introduce correlations between the two vectors, $\mathbf{x}_1$ and $\mathbf{x}_2$, we generate an additional sample from N(0,1) and add it to the first elements of each vector. Thus there is no correlation between the two vectors other than that existing between the first element of each.

Using an initial learning rate of 0.0001 which is decreased linearly to 0 over 50000 iterations, the weights converge to the vectors (**0.679**, 0.023, -0.051, -0.006) and (**0.681**, 0.004, 0.005). This clearly illustrates the high correlation between the first elements of each of the vectors and also the fact that this is the only correlation between the vectors. We may compare the reported results with the optimal values of ($\sqrt{0.5}$,0,0,0) and ($\sqrt{0.5}$,0,0).

The effect of the constraint on the variance of the outputs is clearly seen when we change the distribution from which all samples are drawn to N(0, 5). The weight vectors converge to (**0.141**, 0.002, 0.003, 0.002) and (**0.141**, 0.002, -0.001) - the optimal results are ($\sqrt{0.02}$,0,0,0) and ($\sqrt{0.02}$,0,0). The differences in magnitude are due to the constraint, $E(y_i^2) = 1$ since

$$E(y_i^2) = 1 \Longleftrightarrow E(\mathbf{w}_i \mathbf{x}\mathbf{x}^T \mathbf{w}_i^T) = \mathbf{w}_i R_{xx} \mathbf{w}_i^T = 1 \qquad (5.30)$$

where R_{xx} is the covariance matrix of the input data.

Standard Statistics Maximum Correlation	0.6962		
$\mathbf{w}_1$	0.0260	0.0518	
$\mathbf{w}_2$	0.0824	0.0081	0.0035
Neural Network Maximum Correlation	0.6630		
$\mathbf{w}_1$	0.0264	0.0526	
$\mathbf{w}_2$	0.0829	0.0098	0.0041

Table 5.2: The converged weights from the neural network are compared with the values reported from a standard statistical technique.

5.7.5 Real data

Our second experiment uses a real data set; it comprises 88 students' marks on 5 module exams. The exam results can be partitioned into two data sets: two exams were given as close book exams (C) while the other three were opened book exams (O). The exams were on the subjects of Mechanics(C), Vectors(C), Algebra(O), Analysis(O), and Statistics(O). We thus split the five variables (exam marks) into two sets-the closed-book exams (x_{11}, x_{12}) and the opened-book exams (x_{21}, x_{22}, x_{23}). One possible quantity of interest here is how highly a student's ability on closed-book exams is correlated with his ability on open-book exams. Alternatively, one might try to use the open-book exam results to predict the closed-book results (or vice versa).

The results shown in Table 5.2 were found using a learning rate of 0.0001 and 50000 iterations. We have reported our results to 4 decimal places in this section to facilitate comparison with those reported in a standard statistical text which were found by standard statistical batch methods. The $\mathbf{w}_1$ vector consists of the weights from the closed book exam data to y_1 while the $\mathbf{w}_2$ vector consists of the weights from the open book exam data to y_2. We note the excellent agreement between the methods. The highest correlations are given by a weighted average of x_{11} and x_{12} with the former receiving half the weight of the latter (since $\mathbf{w}_{11} \approx \mathbf{w}_{12}$) and the average of x_{21}, x_{22} and x_{23} heavily weighted on x_{21} (since $\mathbf{w}_{21} >> \mathbf{w}_{22}, \mathbf{w}_{23}$).

5.7.6 Random Dot Stereograms

It has been suggested that one of the goals of sensory information processing may be the extraction of common information between different sensors or across sensory modalities. One reason that this is possible is because of the coherence which exists in time and place in sensory input data. We may view the above network as a means of merging two different data streams - $\mathbf{x}_1$ and $\mathbf{x}_2$ - which may be either representatives of two different modalities or as different representatives of the same modality where such representatives may be different in either time or place. Becker has developed this idea and experimented on a data set which is an abstraction of random dot stereograms: an example is shown graphically in Figure 5.13. The central idea behind this is that two different neural units or neural network modules should learn to extract features that are coherent across their inputs. If there is any feature in common across the two inputs, it should be discovered, while features which are independent across the two inputs will be ignored.

Each input vector consists of a one dimensional random strip which corresponds to the left image and a shifted version of this which corresponds to the right image. The left image has components drawn with equal probability from the set $\{-1, 1\}$ and the right image is generated by choosing a randomly chosen global shift - either one pixel left or one pixel right - and applying it to the left image. We wish to find the maximum linear correlation between y_1 and y_2 which are themselves linear combinations of $\mathbf{x}_1$ and $\mathbf{x}_2$. Because the shifts are chosen with equal probability, there are two equal sets of correlations corresponding to left-shift and right-shift. In order to find these, we require two pairs of outputs and the corresponding pairs of weights ($\mathbf{w}_1$, $\mathbf{w}_2$) and ($\mathbf{w}_3$, $\mathbf{w}_4$). The learning rules for $\mathbf{w}_3$ and $\mathbf{w}_4$ in this experiment are analagous to those for $\mathbf{w}_1$ and $\mathbf{w}_2$; at each presentation of a sample of input data, a simple competition between the products $y_1 y_2$ and $y_3 y_4$ determine which weights will learn on the current input samples: if $y_1 y_2 > y_3 y_4$, then $\mathbf{w}_1$, $\mathbf{w}_2$ are updated, else $\mathbf{w}_3$, $\mathbf{w}_4$ are updated.

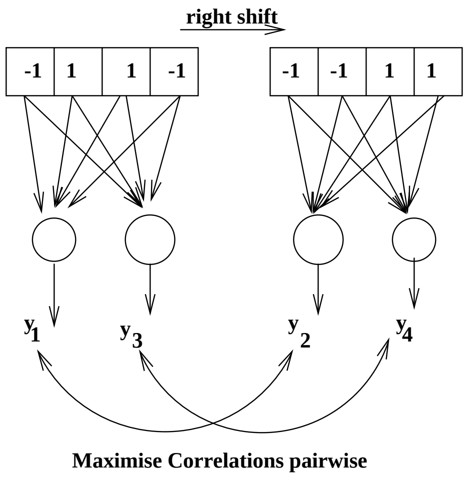


Figure 5.13: The Random dot stereogram data and network. The $\mathbf{x}_2$ set is either a left shifted or a right shifted (as shown) version of the $\mathbf{x}_1$ set. We find that $\mathbf{w}_1$ and $\mathbf{w}_2$ reliably find the left shift and $\mathbf{w}_3$ and $\mathbf{w}_4$ the right shift or vice versa.

$\mathbf{w}_1$	-0.002	**1.110**	0.007	-0.009
$\mathbf{w}_2$	0.002	0.025	**0.973**	0.020
$\mathbf{w}_3$	-0.014	0.026	**1.111**	-0.002
$\mathbf{w}_4$	0.013	**0.984**	0.003	-0.007

Table 5.3: The converged weights clearly show that the first pair of neurons has learned a right shift while the second pair has learned a left shift.

Using a learning rate of 0.001 and 100000 iterations, the weights converge to the vectors shown in Table 5.3.

The first pair of weights $\mathbf{w}_1$ and $\mathbf{w}_2$ have identified the second element of $\mathbf{x}_1$ and the third element of $\mathbf{x}_2$ as having maximum correlation while other inputs are ignored (the weights from these are approximately 0). This corresponds to a right shift. This first pair of outputs has a (sample) correlation of 0.512. Similarly the second pair of weights has idenfied the third element of $\mathbf{x}_1$ and the second element of $\mathbf{x}_2$ as having maximum correlation while other inputs are ignored. The second pair has a (sample) correlation of 0.530 and corresponds to an identification of left shift.

Now for some patterns there will be ambiguity since it is possible that by chance a right-shifted pattern will happen to match the weights $\mathbf{w}_3, \mathbf{w}_4$ and therefore a bank of pairs of pairs of neurons is required to perform as well as Becker's IMAX network. For best results, each set of four neurons as above should see as input a slightly different part of the input data (though with a global left or right shift).

The table thus demonstrates that these high correlations come from one pair learning the shift-left transformation while the other learns the shift-right.

It should be noted at this point that Becker only uses a subset of 12 of the 16 possible patterns. Those which are ambiguous (such as (-1,1,-1,1)) are removed whereas we are drawing our data from all 16 possible patterns. In addition, Becker uses computationally expensive backpropagation of the derivatives of mutual information. We are able to find both correlations with a very simple network.

Comparing the CCA network with Kay and Phillips' network, we might consider e.g. from the first data set, $\mathbf{x}_1$ as the input data to the network and $\mathbf{x}_2$ as the contextual input. The results from the CCA network and the Kay and Phillips network are similar too, though Kay and Phillips use a probabilistic network with an nonlinear activation function designed to manage the effect of contextual information on the response of the network to input data.

Finally, the CCA network presented here may be criticised as a model of biological information processing in that it appears as a non-local implementation of Hebbian learning i.e. the $\mathbf{w}_1$ weights use the magnitude of y_2 as well as y_1 to self-organise. One possibility is to postulate non-learning connections which join y_2 to y_1 thus providing the information that $\mathbf{w}_1$ requires for learning. Alternatively we may describe the λ_1 parameter as a lateral weight from y_2 to y_1 and so the learning rules become

$$\begin{aligned} \Delta w_{1j} &= (\eta\lambda_1)x_{1j}(\frac{y_2}{\lambda_1} - y_1) \\ \Delta\lambda_1 &= \eta_0(1 - y_1^2) \end{aligned}$$

where we have had to incorporate a λ_1 term into the learning rate. Perhaps the second of these suggestions is more plausible than the first as a solution to the non-local feature of the previous learning rules since non-learning connections hardwires some activation passing into the cortex. This is an area of further investigation.

5.7.7 More than two data sets

In integrating information from different sensory modalities, the cortex may be presented with the problem of integrating that from more than two data sets simultaneously. Therefore we extend the algorithm without introducing unnecessarily complex activation passing which would become biologially implausible.

We create an artificial data set which comprises three vectors each of whose first elements have correlations of equal magnitude: $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$ are each 3 dimensional vectors, each of whose elements is initially independently drawn from N(0,1). We now draw a sample from N(0,1) and add it to the first element of each of $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$ and attempt to maximise the correlation between y_1, y_2 and y_3. We opt to maximise three separate constrained objective functions:

$$\begin{aligned} J_1 &= E(y_1y_2) + \frac{1}{2}\lambda_1(1 - y_1^2) \text{ and} \\ J_2 &= E(y_2y_3) + \frac{1}{2}\lambda_2(1 - y_2^2) \text{ and} \\ J_3 &= E(y_3y_1) + \frac{1}{2}\lambda_3(1 - y_3^2) \end{aligned}$$

We use gradient ascent on the instantaneous version of each of these functions with respect to both the weights, $\mathbf{w}_1$, $\mathbf{w}_2$ and $\mathbf{w}_3$, and the Lagrange multipliers, λ_1, λ_2 and λ_3. This gives us the learning rules

$$\begin{aligned} \frac{\partial J_1}{\partial \mathbf{w}_1} &= \mathbf{x}_1y_2 - \lambda_1y_1\mathbf{x}_1 = \mathbf{x}_1(y_2 - \lambda_1y_1) \\ \frac{\partial J_2}{\partial \mathbf{w}_2} &= \mathbf{x}_2y_3 - \lambda_2y_2\mathbf{x}_2 = \mathbf{x}_2(y_3 - \lambda_2y_2) \\ \frac{\partial J_3}{\partial \mathbf{w}_3} &= \mathbf{x}_3y_1 - \lambda_3y_3\mathbf{x}_3 = \mathbf{x}_3(y_1 - \lambda_3y_3) \end{aligned}$$

The derivates with respect to the Lagrange multipliers are similar to the previous rules (5.29) though we have found empirically that the best result are achieved when the λ's learning rate is again very greatly increased (now $\eta_0 \approx 200\eta$ to 1000η).

Using η =0.0001, $\eta_0 = 0.05$ and 100000 iterations, the weights converge to the values shown in Table 5.4. The three way correlation derived by this method is equal to three pairwise correlations.

5.7.8 Non-linear Correlations

Now while the data set in Section 5.7.6 provides us with an abstraction of random dot stereograms, it is not a complete and accurate abstraction of how the cortex extracts depth information from

$\mathbf{w}_1$	**0.812**	0.013	0.027
$\mathbf{w}_2$	**0.777**	-0.014	0.030
$\mathbf{w}_3$	**0.637**	0.007	0.012

Table 5.4: The weights of the converged three input vectors network. The network has clearly identified the correlation between the first element in each vector.

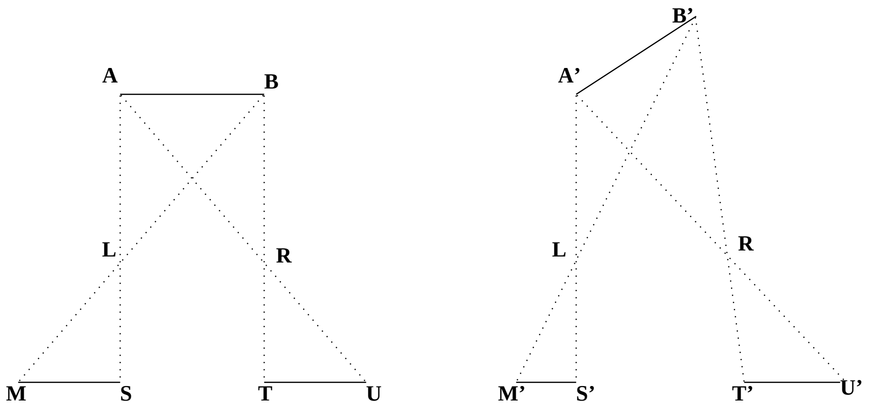


Figure 5.14: The left figure represents visual information from a surface AB which is passed through pupils R and L to flat "retinas" MS and TU. The right figure represents the same scene when the external surface A'B' is not parallel to the plane of the retinas.

surfaces: at any one time, we view not just single points but segments of (at least) a surface. Consider Figure 5.14. We see that the relationship between the projection of the surface on the retinas is a function of the angle between the plane of the surface and the plane of the retinas. We do not wish to determine the precise relationship in any specific case since we wish to create a general purpose depth analyser which does not depend on e.g. both the retinas and the surface being flat, the pupils having a precise relationship with the limits of the viewed surfaces.

Therefore in our final experiment, we investigate the general problem of maximising correlations between two data sets when there may be an underlying nonlinear relationship between the data sets: we generate artificial data according to the prescription:

$$x_{11} = 1 - \sin\theta + \mu_1 \tag{5.31}$$

$$x_{12} = cos\theta + \mu_2 \tag{5.32}$$

$$x_{21} = \theta - \pi + \mu_3 \tag{5.33}$$

$$x_{22} = \theta - \pi + \mu_4 \tag{5.34}$$

where θ is drawn from a uniform distribution in $[0, 2\pi]$ and $\mu_i, i = 1, ..., 4$ are drawn from the zero mean Gaussian distribution N(0, 0.1). Equations (5.31) and (5.32) define a circular manifold in the two dimensional input space while equations (5.33) and (5.34) define a linear manifold within the input space where each manifold is only approximate due to the presence of noise ($\mu_i, i = 1, ..., 4$). The subtraction of π in the linear manifold equations is merely to centre the data.

Thus $\mathbf{x}_1 = \{x_{11}, x_{12}\}$ lies on or near the circular manifold $x_{11}^2 + x_{12}^2 = 1$ while $\mathbf{x}_2 = \{x_{21}, x_{22}\}$ lies on or near the line $x_{21} = x_{22}$.

We wish to test whether the network can find nonlinear correlations between the two data sets, $\mathbf{x}_1$ and $\mathbf{x}_2$, and test whether such correlations are greater than the maximum linear correlations. To do this we train two pairs of output neurons:

- We train one pair of weights $\mathbf{w}_1$ and $\mathbf{w}_2$ using rules (5.29)
- We train a second pair of outputs, y_3 and y_4 which are calculated from

$$y_3 = \sum_j w_{3j} \tanh(v_{3j} x_{1j}) = \mathbf{w}_3 \mathbf{f}_3 \text{ and} \tag{5.35}$$

$$y_4 = \sum_j w_{4j} \tanh(v_{4j} x_{2j}) = \mathbf{w}_4 \mathbf{f}_4 \tag{5.36}$$

This gives us another pair of adaptable parameters which may be changed to maximise the correlation. In particular, since the weights, $\mathbf{v}_3$ and $\mathbf{v}_4$ are used prior to the use of the nonlinearity, this gives us extra flexibility in maximising correlations.

The correlation between y_1 and y_2 neurons was maximised using the previous linear operation (11) while that between y_3 and y_4 used the functions

$$\begin{aligned} J_3 &= E(y_3 y_4) + \frac{1}{2}\lambda_3(1 - y_3^2) \text{ and} \\ J_4 &= E(y_3 y_4) + \frac{1}{2}\lambda_4(1 - y_4^2) \end{aligned}$$

whose derivatives give us (taking into account the nonlinearities (5.35) and (5.36))

$$\begin{aligned} \frac{\partial J_3}{\partial \mathbf{w}_3} &= \mathbf{f}_3 y_4 - \lambda_3 y_3 \mathbf{f}_3 = \mathbf{f}_3(y_4 - \lambda_3 y_3) \\ \frac{\partial J_3}{\partial \mathbf{v}_3} &= \mathbf{w}_3 y_4 (1 - \mathbf{f}_3^2)\mathbf{x}_1 - \lambda_3 \mathbf{w}_3 (1 - \mathbf{f}_3^2)\mathbf{x}_1 y_3 \\ &= \mathbf{w}_3(1 - \mathbf{f}_3^2)\mathbf{x}_1(y_4 - \lambda_3 y_3) \end{aligned} \tag{5.37}$$

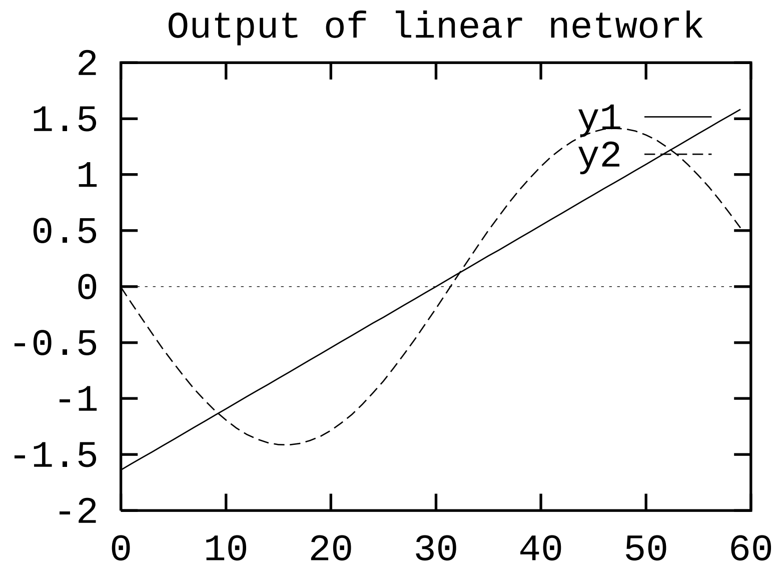
Similarly with the J_4 function, $\mathbf{w}_4$, $\mathbf{v}_4$, and λ_4. This gives us a method of changing the weights and the Lagrange multipliers on an online basis. We use the joint learning rules

$$\begin{aligned} \Delta \mathbf{w}_3 &= \eta \mathbf{f}_3(y_4 - \lambda_3 y_3) \\ \Delta \mathbf{w}_4 &= \eta \mathbf{f}_4(y_3 - \lambda_4 y_4) \\ \Delta \mathbf{v}_{3i} &= \eta \mathbf{x}_{1i} \mathbf{w}_{3i}(y_4 - \lambda_3 y_3)(1 - \mathbf{f}_3^2) \\ \Delta \mathbf{v}_{4i} &= \eta \mathbf{x}_{2i} \mathbf{w}_{4i}(y_3 - \lambda_4 y_4)(1 - \mathbf{f}_4^2) \end{aligned}$$

We use a learning rate of 0.001 for all weights and learn over 100000 iterations. The network finds a sample linear correlation between the data sets of 0.623 (equal to the correlation between y_1 and y_2) while the nonlinear neurons, y_3 and y_4, have a sample correlation of 0.865. In putting the data sets through the nonlinear tanh() function we have created a relationship whose correlations are greater than the linear correlations of the original data set. We show a test of the outputs from both the linear and nonlinear networks in Figure 5.15 in which we graph the output values of the trained network from each pair of neurons against inputs where the θ values in equations (5.31)-(5.34) range from -3 to 3 . We see that the linear network is aligning the outputs as well as may be expected but the nonlinear network's outputs are very closely aligned with each other over much more of the data set.

5.8 Speeding up Error Descent

We have, however, used error descent as our chosen method of supervised weight adaption. But the backpropagation method as described so far is innately slow. The reason for this is shown diagrammatically in Figure 5.16. In this Figure, we show (in two dimensions) the contours of constant error. Since the error is not the same in each direction we get elipses rather than circles. If we are following the path of steepest descent, which is perpendicular to the contours of constant error, we get a zig-zag path as shown. The axes of the elipse can be shown to be parallel to the eigenvectors of the Hessian matrix. The greater the difference between the largest and the smallest eigenvalues, the more eliptical the error surface is and the more zig-zag the path that the fastest descent algorithm takes.



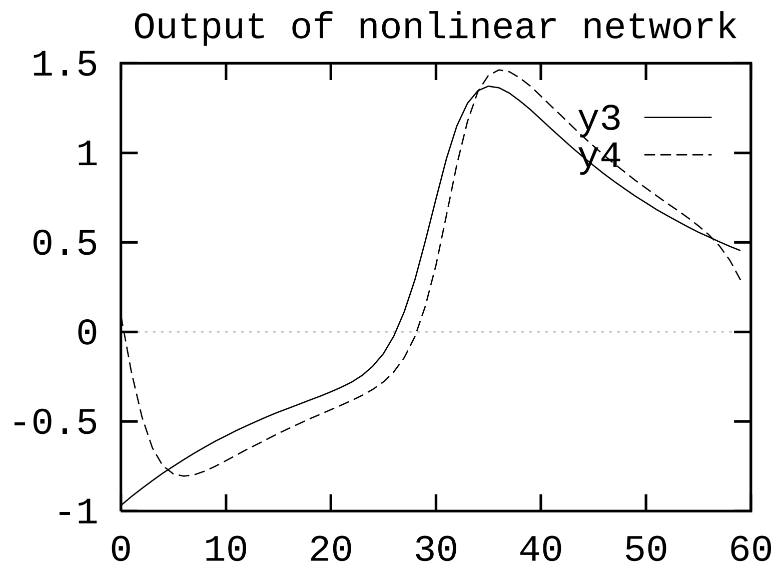


Figure 5.15: The top diagram shows the outputs, y_1 and y_2, of the linear network, while the lower diagram shows the outputs, y_3 and y_4, of the nonlinear network. Visual inspection would suggest that the outputs from the nonlinear network are more correlated. The actual sample corrrelation values achieved were 0.623 (linear) and 0.865 (nonlinear).

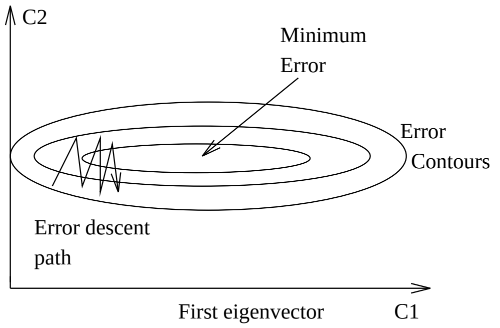


Figure 5.16: The path taken by the route of fastest descent is not the path directly to the centre (minimum) of the error surface.

5.8.1 Mathematical Background

The matrix of second derivatives is known as the Hessian and may be written as

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_m} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_m} & \frac{\partial^2 E}{\partial w_2 \partial w_m} & \cdots & \frac{\partial^2 E}{\partial w_m^2} \end{bmatrix} \tag{5.38}$$

If we are in the neighbourhood of a minimum $\mathbf{w*}$, we can consider the (truncated) Taylor series expansion of the error as

$$E(\mathbf{w}) = E(\mathbf{w}^*) + (\mathbf{w} - \mathbf{w}^*)^T \nabla\mathbf{E} + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \tag{5.39}$$

where $\mathbf{H}$ is the Hessian matrix at $\mathbf{w*}$ and $\nabla\mathbf{E}$ is the vector of derivatives of E at $\mathbf{w*}$. Now at the minimum ($\mathbf{w*}$), $\nabla\mathbf{E}$ is zero and so we can approximate equation 5.39 with

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \tag{5.40}$$

The eigenvectors of the Hessian,$\mathbf{c}_i$ are defined by

$$\mathbf{H}\mathbf{c}_i = \lambda_i \mathbf{c}_i \tag{5.41}$$

and form an orthonormal set (they are perpendicular to one another and have length 1) and so can be used as a basis of the $\mathbf{w}$ space. So we can write

$$\mathbf{w} - \mathbf{w}^* = \sum_i \alpha_i \mathbf{c}_i \tag{5.42}$$

Now since $\mathbf{H}(\mathbf{w} - \mathbf{w}^*) = \sum_i \lambda_i \alpha_i \mathbf{c}_i$ we have

$$E(\mathbf{w}) = E(\mathbf{w}*) + \frac{1}{2} \sum_i \lambda_i \alpha_i^2 \tag{5.43}$$

In other words, the error is greatest in directions where the eigenvalues of the Hessian are greatest. Or alternatively, the contours of equal error are determined by $\frac{1}{\sqrt{\lambda_i}}$. So the long axis in Figure 5.16 has radius proportional to $\frac{1}{\sqrt{\lambda_1}}$ and the short axis has radius proportional to $\frac{1}{\sqrt{\lambda_2}}$. Ideally we would like to take this information into account when converging towards the minimum.

Now we have $\Delta E = \sum_i \alpha_i \lambda_i \mathbf{c}_i$ and we have $\Delta \mathbf{w} = \sum_i \Delta \alpha_i \mathbf{c}_i$ so since we wish to use $\Delta \mathbf{w} = -\eta \Delta E$, we have

$$\Delta \alpha_i = -\eta \lambda_i \alpha_i \tag{5.44}$$

and so

$$\alpha_i^{new} = (1 - \eta \lambda_i) \alpha_i^{old} \tag{5.45}$$

which gives us a means of adjusting the distance travelled along the eigenvector in each direction. So by taking a larger learning rate η we will converge quicker to the minimum error point in weight space. However, there are constraints in that the changes to α_i form a geometric sequence,

$$\begin{aligned}
\alpha_i^{(1)} &= (1 - \eta \lambda_i) \alpha_i^{(0)} \\
\alpha_i^{(2)} &= (1 - \eta \lambda_i) \alpha_i^{(1)} = (1 - \eta \lambda_i)^2 \alpha_i^{(0)} \\
\alpha_i^{(3)} &= (1 - \eta \lambda_i) \alpha_i^{(2)} = (1 - \eta \lambda_i)^3 \alpha_i^{(0)} \\
\alpha_i^{(T)} &= (1 - \eta \lambda_i) \alpha_i^{(T-1)} = (1 - \eta \lambda_i)^T \alpha_i^{(0)}
\end{aligned}$$

This will diverge if $|1 - \eta \lambda_i| > 1$. Therefore we must choose a value of η as large as possible but not so large as to break this bound. Therefore $\eta < \frac{2}{\lambda_1}$ where λ_1 is the greatest eigenvalue of the Hessian. But note that this means that the convergence along other directions will be at best proportional to $(1 - \frac{2\lambda_i}{\lambda_1})$ i.e. convergence is determined by the ratio of the smallest to the largest eigenvalues.

Thus gradient descent is *inherently* a slow method of finding the minimum of the error surface. We can now see the effect of momentum diagrammatically since the momentum is built up in direction $\mathbf{c}_1$ while the continual changes of sign in direction $\mathbf{c}_2$ causes little overall change in the momentum in this direction.

5.8.2 QuickProp

Fahlman has developed an heuristic which attempts to take into account the curvature of the error surface at any point by defining

$$\Delta w_{ij}(k) = \begin{cases} \alpha_{ij}(k) \Delta w_{ij}(k-1), & \text{if } \Delta w_{ij}(k-1) \neq 0 \\ \eta_0 \frac{\partial E}{\partial w_{ij}}, & \text{if } \Delta w_{ij}(k-1) = 0 \end{cases} \tag{5.46}$$

where

$$\alpha_{ij}(k) = \min\left(\frac{\frac{\partial E(k)}{\partial w_{ij}}}{\frac{\partial E(k-1)}{\partial w_{ij}} - \frac{\partial E(k)}{\partial w_{ij}}}, \alpha_{max} \right) \tag{5.47}$$

The idea is to approximate the error surface with a parabola and to use two successive evaluations of the error and an evaluation of is gradient (the rate of change of the error). It is usual to use the algorithm with an upper bound set on the step size.

5.8.3 Line Search

If we know the direction in which we wish to go - the direction in which we will change the weights - we need only determine how far along the direction we wish to travel. We therefore choose a value of λ in

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda^{(t)} \mathbf{d}^{(t)} \tag{5.48}$$

in order to minimise

$$E(\lambda) = E(\mathbf{w}^{(t)} + \lambda^{(t)} \mathbf{d}^{(t)}) \tag{5.49}$$

We show E as a function of λ in Figure 5.17. If we start at a point a and have points b and c such that $E(a) > E(b)$ and $E(c) > E(b)$ then it follows that there must be a minimum between a and c. So we now fit a parabola to a, b and c and choose the minimum of that parabola to get d. Now we can choose 3 of these four points (one of which must be d) which also satisfy the above relation and iterate the parabola fitting and minimum finding.

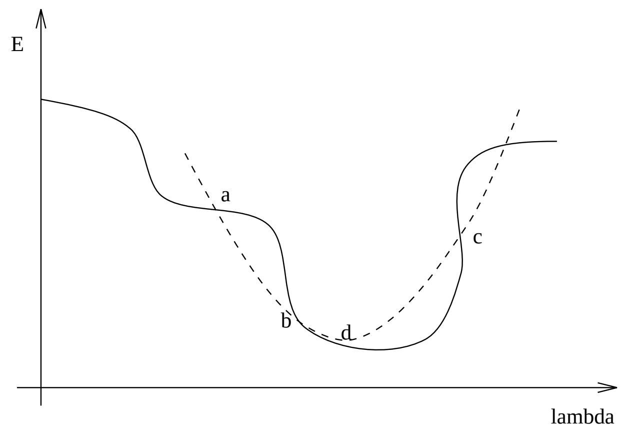


Figure 5.17: The error function as a function of λ. The (unknown) error function is shown as a solid line. The fitted parabola is shown as a dotted line with minimum value at d.

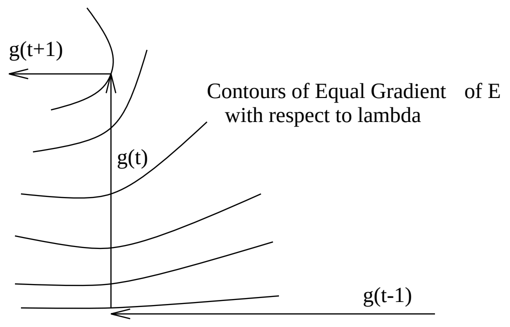


Figure 5.18: After minimising error in one direction, the new direction of fastest descent is perpendicular to that just traversed. This leads to a zigzagging approach to the minimum.

5.8.4 Conjugate Gradients

Now we must find a method to find the direction **d** in which to search for the minimum. Our first attempt might be to find the best (minimum error) point along one direction and then start from there to find the best point along the fastest descent direction from that point. However as shown in Figure 5.18, we see that this leads to zig-zagging and so the minimisation of the error function proceeds very slowly.

We require *conjugate* or non-interfering directions: we choose the direction $\mathbf{d}^{(t+1)}$ such that the component of the direction $\mathbf{d}^{(t)}$ is (approximately) unaltered.

The slowness of the vanilla backpropagation method is due partly at least to the interaction between different gradients. Thus the method zig-zags to the minimum of the error surface. The conjugate-gradient method avoids this problem by creating an intertwined relationship between the direction of change vector and the gradient vector. The method is

- Calculate the gradient vector **g** for the batch of patterns as usual. Call this $\mathbf{g}(0)$, the value of **g** at time 0 and let $\mathbf{p}(0) = \mathbf{g}(0)$.
- Update the weights according to

$$\Delta \mathbf{w}(n) = \eta(n)\mathbf{p}(n) \tag{5.50}$$

- Calculate the new gradient vector **g**(n+1) with the usual method
- Calculate the parameter $\beta(n)$ (see below).

- Recalculate the new value of **p** using

$$\mathbf{p}(n+1) = -\mathbf{g}(n+1) + \beta(n)\mathbf{p}(n) \tag{5.51}$$

- Repeat from step 2 till convergence

The step left undefined is the calculation of the parameter $\beta(n)$. This can be done in a number of ways but the most common (and one of the easiest) is the Fletcher-Reeves formula

$$\beta(n) = \frac{\mathbf{g}^T(n+1)\mathbf{g}(n+1)}{\mathbf{g}^T(n)\mathbf{g}(n)} \tag{5.52}$$

The calculation of the parameter $\eta(n)$ (which is equivalent to λ in the last section in that it determines how far a jump will be made) is done to minimise the cost function

$$E(\mathbf{w}(n) + \eta\mathbf{p}(n)) \tag{5.53}$$

As with the Newton method, convergence using this method is much faster but computationally more expensive.

5.9 Least Mean Square Error Reconstruction

Finally it is worth stating that many attempts have been made to derive the PCA algorithms using an error descent mechanism. Consider the negative feedback netork discussed in Chapter 3. Then let us wish to minimise the error, **e**, at **x** after the output neuron's activation is returned.

We wish to minimise

$$J(\mathbf{W}) = \frac{1}{2}\mathbf{1}^T E(\mathbf{e}^2|\mathbf{W}) = \frac{1}{2}\mathbf{1}^T E(\mathbf{x} - \mathbf{W}\mathbf{W}^T\mathbf{x})^{\mathbf{2}}|\mathbf{W}) \tag{5.54}$$

where **1** is the vector of 1s.

Consider the j^{th} component of the reconstruction error, e_j.

$$e_j = x_j - \sum_{i=1}^{M} w_{ij}\mathbf{w_i}.\mathbf{x} \tag{5.55}$$

where, as before, $\mathbf{w_i}$ is the vector of weights into the i^{th} output neuron. Then we wish to find stationary point(s) of the derivative of $J(\mathbf{W})$ i.e. where

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{w}_m} = \sum_{j=1}^{M} e_j \frac{\partial e_j}{\partial \mathbf{w}_m} = 0 \tag{5.56}$$

Now,

$$\frac{\partial e_j}{\partial \mathbf{w}_m} = -w_{mj}\mathbf{x} - (\mathbf{w_m}.\mathbf{x})[0, 0, ..., 1, 0, ..0]^T \tag{5.57}$$

where the last vector has a 1 in only the j^{th} position. Then,

$$\begin{aligned} \frac{\partial J(\mathbf{W})}{\partial \mathbf{w}_m} &= -\sum_{j=1}^{M}(x_j - \sum_{i=1}^{M} w_{ij}\mathbf{w_i}.\mathbf{x}).\{w_{mj}\mathbf{x} + \mathbf{w_m}.\mathbf{x}[0, 0, .., 1, 0.., 0]^T)\} \\ &= -(\mathbf{x} - \mathbf{W}^T\mathbf{W}\mathbf{x})\mathbf{w}_m.\mathbf{x} - (\mathbf{x} - \mathbf{W}^T.\mathbf{W}\mathbf{x})(\mathbf{w}_m.x)\mathbf{1}^T \end{aligned} \tag{5.58}$$

This can be used in the usual way in the gradient ascent algorithm

$$\Delta\mathbf{W} \propto \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

to give a learning rule

$$\Delta \mathbf{W} = \mathbf{x}^T \mathbf{x}(\mathbf{I} - \mathbf{W}^T \mathbf{W})\mathbf{W} + (\mathbf{x} - \mathbf{W}^T \mathbf{W}\mathbf{x})(\mathbf{W}\mathbf{x})^T \tag{5.59}$$

Now while this last equation is not quite the algorithm we have used previously, Xu has shown that "on the average", the scalar product of our negative feedback algorithm (or equivalently Oja's subspace algorithm) and the above learning rule is positive. Thus "on the average", the negative feedback network can be thought of as minimising the residuals at **x**.

It should be stated that this is not the only way to attempt to derive a PCA network: we could for instance attempt to derive a network which maximised variance under the constraint that the network weights have unit length.

5.10 Conclusion

In this chapter we have seen a number of different features associated with finding an objective function and using the criterion of optimising this function to determine the course of adaption of the weights of the network. Some of these methods are sometimes described as unsupervised learning but it is more accurate to state that the methods involve some features of self-supervision.

In the two cases of Becker and Hinton's IMax and Kay and Phillips' Contextual network, we looked at explicit optimisation of information theoretical objectives which gave fascinating insights into the capabilities of the networks concerned.

Finally we have looked at two methods which are used to quicken the rate of convergence of supervised learning methods without affecting the generalisation characteristics of the networks using them; in general such methods tend to be computationally expensive but do manage to speed up the convergence to the optimum value of the objective function.

Chapter 6

Identifying Independent Sources

6.1 The Trouble with Principal Components

Figure 6.1 illustrates some disadvantages of attempting to use Principal Components to discover the information in data. The top diagram shows a situation in which there is a non-linear relationship between the variables; the bottom one shows a distribution consisting of two distinct sub-distributions.

Recall first that PCA is a linear operation and so the principal component directions are all straight lines. Thus in the first case, the first PC would form a chord of the circular distribution, while in the second case, the first PC would be a diagonal of the rectangle approximately parallel to the x-axis. The trouble in each case is that projecting the data onto the first PC completely hides the structure in the data - you cannot see either the circular dependency in the first case or that, in the second, there are two different blobs to the distribution. We will concentrate on the latter situation in this chapter in which a distribution has two or more underlying causes; we are looking for neural algorithms which identify independent sources of the observed data.

The neurons we use will have non-linear activation functions since wholly linear networks have a very limited set of capabilities. In addition, we will be discussing our networks in the context of biological neurons which tend to have non-linear activation functions: typically a neuron will not fire till a particular threshold has been reached and will saturate (not exceed a particular firing rate) when a specific input level has been attained.

6.1.1 Independent Codes

Barlow has developed a theory of learning based on the neuron as a "suspicious coincidence detector": if input A is regularly met in conjunction with input B this represents a suspicious coincidence; there is something in the neuron's environment which is worth investigating. A crude example might be the coincidence of mother's face and food and warmth for a young animal. We can note the similarity of this question to that asked by Becker and Hinton (last chapter) but we will see that a very different tactic will be taken by the researchers discussed in this chapter.

The types of codes which are developed using the learing rules in this chapter are sometimes known as "factorial codes": we have lots of different symbols representing the different parts of the environment and the occurrence of a particular input is simply the product of probabilities of the individual code symbols. So if neuron 1 says that it has identified a sheep and neuron 2 states that it has identified blackness, then presenting a black sheep to the network will cause neurons 1 and 2 to both fire. Also such a coding should be invertible: if we know the code we should be able to go to the environment and identify precisely the input which caused the code reaction from the network. So when we see neurons 1 and 2 firing we know that it was due to a black sheep.

We will maintain a close connection with psychological principles which should suggest that we are using a biologically plausible rule such as the Hebb rule. We have seen that the Hebb rule will extract information from the environment. What we need to do is modify the Hebb rule so

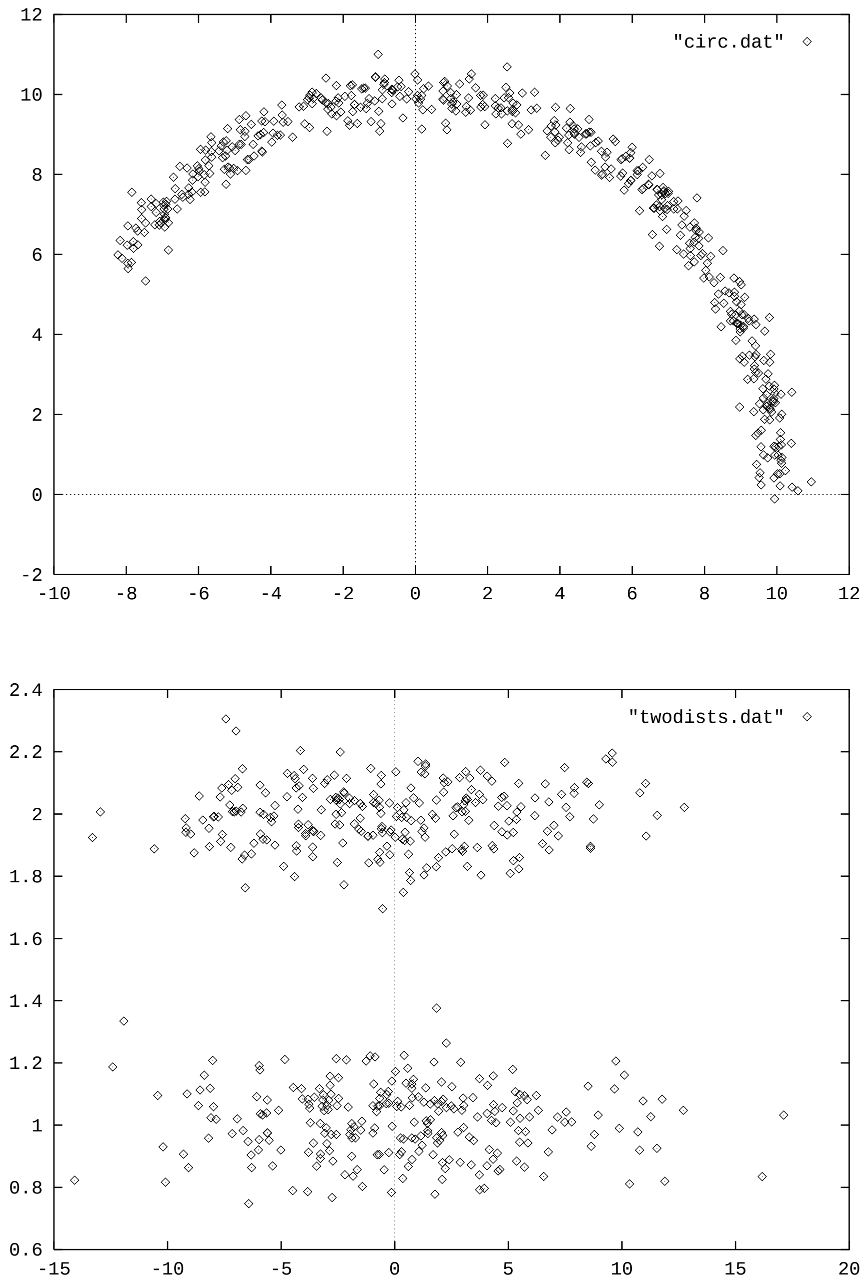


Figure 6.1: Two distributions where Principal Component Analysis does not find the structure in the data: in the first distribution, the first PC would be an almost horizontal chord of the arc; in the second, it would lie on the diagonal of the rectangle. So projecting onto either principal component axis would hide the structure in the data.

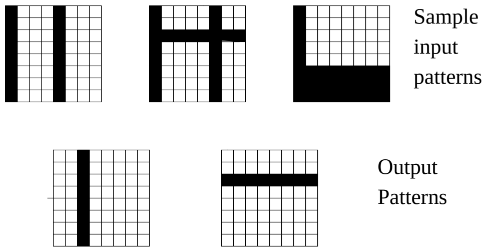


Figure 6.2: The top line shows sample data (input values) presented to the network. The second layer shows the independent sources which we hope will also be the network's response. If this is so, it has clearly identified the suspicious coincidences of lines.

that each neuron responds to a particular set of inputs which is unique to itself. One way to do this is with competitive learning.

6.1.2 A Typical Experiment

Often in artificial neural network research, we create artificial data and run our networks on them before attempting experiments with real world data. The advantages of artificial data are

1. We have more control over our data. Real world data is often noisy with outliers and, in addition, we are also relying on the efficacy of our sampling methods.
2. We can more easily understand our results. We will create data with the specific aim of aiding our understanding of the network.
3. We know the answers we hope to get from our network. Our data has been specifically created to help test a particular hypothesis.
4. We can control the magnitude of our data set. We will use a limited number of representative samples.

The input data here consists of a square grid of input values where $x_i = 1$ if the i^{th} square is black and 0 otherwise. However the patterns are not random patterns: each input consists of a number of randomly chosen horizontal or vertical lines. The network must identify the existence of these lines. The important thing to note is that each line can be considered as an independent source of blackening a pixel on the grid: it may be that a particular pixel will be twice blackened by both a horizontal and a vertical line at the same time but we need to identify both of these sources.

Typically, on an 8*8 grid, each of the 16 possible lines are drawn with a fixed probability of $\frac{1}{8}$ independently from each of the others. The data set then is highly redundant in that there exists 2^{64} possible patterns and we are only using at most 2^{16} of these. We will typically have 16 output neurons whose aim is to identify (or respond optimally to) one of the input lines. Thus from any pattern composed of some of the set of 16 lines, we can identify exactly which of the 16 lines were used to create the pattern. Note the factorial nature of the coding we are looking for: neurons 1, 3 and 10 will fire if and only if the input is composed of a pattern from sources 1,3 and 10. Note also the code's reversibility: given that neurons 1,3 and 10 are firing we can recreate the input data exactly.

If we use a principal component net on this data, our first principal component will be a small magnitude uniform vector over all 64 positions. i.e. we get a global smearing of the patterns which does not reveal how each pattern came to be formed.

6.2 Competitive Learning

One of the non-biological aspects of the basic Hebbian learning rule is that there is no limit to the amount of resources which may be given to a synapse. This is at odds with real neural growth in that it is believed that there is a limit on the number and efficiency of synapses per neuron. In other words, there comes a point during learning in which if one synapse is to be strengthened, another must be weakened. This is usually modelled as a competition for resources.

In competitive learning, there is a competition between the output neurons after the activity of each neuron has been calculated and only that neuron which wins the competition is allowed to fire. Such output neurons are often called **winner-take-all** units. The aim of competitive learning is to **categorize** the data by forming **clusters**. However, as with the Hebbian learning networks, we provide no correct answer (i.e. no labelling information) to the network. It must self-organise on the basis of the structure of the input data. The method attempts to ensure that the similarities of instances within a class is as great as possible while the differences between instances of different classes is as great as possible.

6.2.1 Simple Competitive Learning

The basic mechanism of simple competitive learning is to find a winning unit and update its weights to make it more likely to win in future should a similar input be given to the network. We first have the activity transfer equation

$$y_i = \sum_j w_{ij} x_j, \forall i$$

which is followed by a competition between the output neurons and then

$$\Delta w_{ij} = \eta(x_j - w_{ij}), \text{ for the winning neuron i}$$

Note that the change in weights is a function of the *difference* between the weights and the input. This rule will move the weights of the winning neuron directly towards the input. If used repeatedly over a distribution, the weights will tend to the mean value of the distribution since $\Delta w_{ij} \rightarrow 0 \Longleftrightarrow w_{ij} \rightarrow E(x_j)$, the mean value of the j^{th} input. We can actually describe this rule as a variant of the Hebb learning rule if we state that y_i =1 for the winning i^{th} neuron and y_i =0 otherwise. Then the learning rule can be written $\Delta w_{ij} = \eta y_i(x_j - w_{ij})$ i.e. a Hebbian learning rule with weight decay. A geometric analogy is often given to aid understanding simple competitive learning. Consider Figure 6.3: we have two groups of points lying on the surface of the sphere and the weights of the network are represented by the two radii. The weights have converged to the mean of each group and will be used to classify any future input to one or other group.

A potential problem with this type of learning is that some neurons can come to dominate the process i.e. the same neuron continues to win all of the time while other neurons (**dead neurons**) never win. While this can be desirable if we wish to preserve some neurons for possible new sets of input patterns it can be undesirable if we wish to develop the most efficient neural network. It pays in this situation to ensure that all weights are normalised at all times (and so already on the surface of the sphere) so that one neuron is not just winning because it happens to be greater in magnitude than the others. Another possibility is **leaky learning** where the winning neuron is updated and so too by a lesser extent are all other neurons. This encourages all neurons to move to the areas where the input vectors are to be found. The amount of the leak can be varied during the course of a simulation. Another possibility is to have a variable threshold so that neurons which have won often in the past have a higher threshold than others. This is sometimes known as learning with a **conscience**. Finally noise in the input vectors can help in the initial approximate weight learning process till we get approximate solutions. As usual an annealing schedule is helpful.

6.3 Anti-Hebbian and Competitive Learning

6.3.1 Sparse Coding

Hertz *et al* point out that simple competitive learning leads to the creation of **grandmother** cells, the proverbial neuron which would fire if and only if your grandmother hove in sight. The major

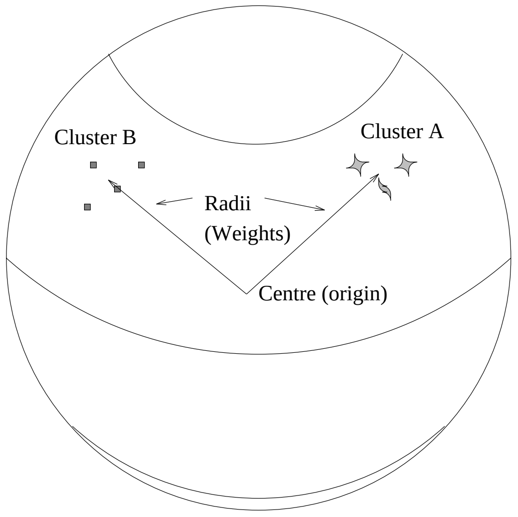


Figure 6.3: The input vectors are represented by points on the surface of a sphere and the lines represent the directions to which the weights have converged. Each is pointing to the mean of the group of vectors surrounding it.

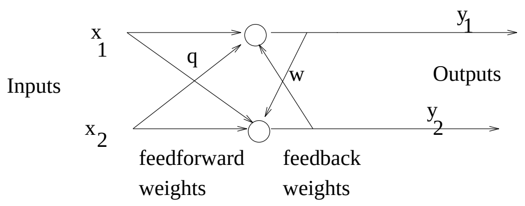


Figure 6.4: Foldiak's last model: we have feedforward weights from inputs to outputs and then feedback (or lateral) connections between the outputs.

difficulty with such neurons is their lack of robustness: if you lose your grandmother cell, will you never again recognise your grannie. In addition, we should note that if we have N grandmother cells we can only recognise N categories whereas if we are using a binary code, we could distinguish between 2^N categories.

So simple competitive learning leads to a single neuron firing in response to an input pattern. At the other extreme, when a large number of neurons are firing for each input pattern, subsequent layers have to work much harder to identify the information being presented by that layer. So if our layer is used to classify a full 2^N categories, it may allow a very compact representation but make the subsequent identification of the individual categories difficult.

Földiák has suggested that an appropriate compromise between these competing constraints would be to have the neurons in a layer forming a sparse coding of the input data. i.e. each pattern is represented by a set of m firing neurons where $m > 1$ but $m << n$, the number of neurons in the layer. He believes that such a representation potentially trades off the benefits of increased representational capacity to be had with a distributed representation with the simplicity to be had with a completely local representation. It is this balance between cooperation (so that a set of output neurons can represent the input pattern which is currently being presented) and competition (so that not all outputs are used to represent all patterns) that seems necessary for the extraction of salient features of the problem.

6.3.2 Földiák's Sixth Model

The major difference between this model and Földiák's previous ones is that the neurons are non-linear units, each with an adjustable threshold over which its activation must be before the neuron can fire. In order to get a sparse coding, each neuron tries to keep its probability of firing down by adjusting its own threshold. If a neuron has been firing frequently, its threshold will increase to make it more difficult for the neuron to fire in the future; if it has been firing only occasionally, its threshold will decrease making it easier for the neuron to fire in future. This mechanism does have some biogical plausibility in that neurons do become habituated to inputs and stop responding so strongly to repeated sets of inputs. An obvious example of which you will be aware is your reaction when you emerge from a dark tavern into broad daylight; initially you will be blinded because there is too much environmental information being processed by the neurons in your retina. This effect soon disappears as you become used to your new environment.

Let there be n inputs and m output neurons (the representational layer). Then

$$y_i = f(\sum_{j=1}^{n} q_{ij}x_j + \sum_{j=1}^{m} w_{ij}y_j - t_i) \tag{6.1}$$

where q_{ij} is the weight of the feedforward connection from the j^{th} input x_j, w_{ij} is the weight of the lateral connection from the j^{th} output neuron to the i^{th} in that layer and t_i is the adjustable threshold for the i^{th} output neuron. Both sets of weights and the threshold are adjustable by competitive type learning:

$$\begin{aligned}
\Delta w_{ij} &= \begin{cases} -\alpha(y_i y_j - p^2) \text{ if } i \neq j \\ 0 \text{ if } i = j \text{ or } w_{ij} < 0 \end{cases} \\
\Delta q_{ij} &= \beta y_i (x_j - q_{ij}) \\
\Delta t_i &= \gamma (y_i - p)
\end{aligned}$$

where α, β, γ are adjustable learning rates. The feedforward weights, q_{ij}, use simple competitive learning. The lateral learning rule for the w weights will stabilise when $E(y_i y_j) = p^2$. i.e. each pair of units will tend to fire together a fixed proportion of the time. This rule will interact with the rule which changes the threshold: the long term effect of this rule should be to set the threshold t_i to a set value to ensure $E(y_i) = p$. By choosing the value of p appropriately we can determine the level of sparseness of the firing pattern. For example suppose $p = \frac{1}{10}$. Each neuron will fire about $\frac{1}{10}$ of the time and will fire at the same time as each of its neighbours about $\frac{1}{100}$ of the time. So if we have 500 output neurons, then each input pattern would elicit a response from about 50 output neurons which is a distributed representation and not a grandmother cell response but one in which the number of neurons firing at any one time is much less than the number of output neurons.

6.3.3 Implementation Details

Földiák actually describes his feedforward rule with the differential equation

$$\frac{dy_i^*}{dt} = f(\sum_{j=1}^{n} q_{ij}x_j + \sum_{j=1}^{m} w_{ij}y_j^* - t_i) - y_i^* \tag{6.2}$$

which is viewing the neuron as a dynamically changing neuron responding to a mixture of feed-forward inputs and lateral inputs which are themselves changing in time. It can be shown that , provided the feedback is symmetric, the network is guaranteed to settle after a brief transient. Földiák simulates this transient by numerically solving the differential equations. He uses initial conditions

$$y_i^*(0) = f(\sum_{j=1}^{n} q_{ij}x_j - t_i) \tag{6.3}$$

Network output	Input pattern
1000000000000000	t
0000100000000000	i or !
0010011000001000	J

Table 6.1: Some of the codes found by Foldiak's network after being presented with 8000 letters. t appears frequently in the text and so t uses a small number of firing neurons while J appears infrequently and so can use a larger number of firing neurons. Note also that i and ! share a code.

with the function f(u) equal to the logistic function $\frac{1}{1+\exp(-\lambda u)}$. The values at the stable state are now rounded to 1 or 0 depending on whether $y_i^* > 0.5$ or not. Feedforward weights are initially random but normalised so that $\sum_j q_{ij}^2 = 1$ and the feedback weights are zero.

I suggest that you repeat this experiment but do not numerically simulate the transient but operate a method which selects one output randomly at a time to update. Continue for a set number of iterations (2 or 10 or 100?) and investigate under what conditions the network converges.

6.3.4 Results

On the experimental data discussed previously, the network learns to find the lines so that the feedforward weights match a single possible line. The code generated in this case is optimal in that there is no redundancy between neurons - every neuron represents a different line - and all information in the input is preserved - the set of output neurons will identify exactly which of the input squares is on by identifying which lines are represented in the pattern.

An extension to this experiment was the presentation of input patterns consisting of images of letters presented in a fixed position on an 8*15 raster display. During training the letters were presented in a random order with the same probabilities that they appeared in a piece of English text. The results were as you might have hoped (Table 6.1) had you hand-designed a code (e.g. á la Huffmann): frequent letters had fewer active bits than infreqent ones since otherwise the correlations introduced by simultaneous frequent firing would force the decorrelating lateral weights to increase inhibition between the neurons. Another feature was that no two frequent letters were assigned to the same output though some rare letters did share an output. Finally the code has the nice property that similar inputs e.g. O and Q are mapped to similar output neurons.

Finally note the interaction between the lateral weights and the threshold; this will be a theme of this chapter.

6.4 Competitive Hebbian Learning

White has attempted to play off the extreme focussing of competitive learing with the broad smearing of Hebbian learing with a model which attempts to combine both effects.

Consider a one layer network with outputs

$$y_i = f(\sum_j w_{ij} x_j) - \frac{1}{2} \tag{6.4}$$

where $f(u) = \frac{1}{1+\exp(-u)}$ and x_j is the j^{th} input. Then each output satisfies $-\frac{1}{2} \leq y_i \leq \frac{1}{2}$. The information given by the outputs is proportional to

$$F = \sum_{i=1}^{N} y_i^2 \tag{6.5}$$

If we simply wish to maximise F we can use gradient ascent so that

$$\Delta w_{ij} = \frac{\partial F}{\partial w_{ij}} = 2 f_i' y_i x_j \tag{6.6}$$

Now since $2f_i'$ is constant for the whole weight vector into output neuron i, we can ignore this factor (it alters the magnitude of the change but not the direction) to get simple Hebbian learning:

$$\Delta w_{ij} = \alpha y_i x_j \tag{6.7}$$

where α, the learning rate, contains the $2f_i'$ factor. But we know that this causes every output neuron to move to the principal component of the input data - i.e. all are covering the same information. We introduce a penalty term to discourage this and the simplest is that the mean square correlation between output neurons should be as low as possible: if we get totally decorrelated outputs then

$$E(g_{ik}) = E((y_i y_k)^2) = 0 \tag{6.8}$$

We can incorporate the constraint into the optimisation by using the method of Lagrange multipliers: now we try to maximise the function G given by

$$G = F + \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1,k\neq i}^{N} \lambda_{ik} g_{ik} \tag{6.9}$$

which gives us

$$\Delta w_{ij} = \frac{\partial G}{\partial w_{ij}} \propto \alpha y_i x_j \{1 + \sum_{k\neq i} \lambda_{ik} y_k^2\} \tag{6.10}$$

White shows that one solution to this (for the λ values) occurs when all λ_{ik} are equal to -4. Thus the weight change equation becomes

$$\Delta w_{ij} = \alpha y_i x_j \{1 - 4\sum_{k\neq i} y_k^2\} \tag{6.11}$$

This rule is known as the Competitive Hebbian Rule: it is used with a hard limit to the overall length of the vector of weights into a particular output neuron and in addition we do not allow the term in brackets to become negative. There are therefore three phases to the learning in this network:

Neuron outputs are weak In this case there is essentially no interaction between nodes (the term in the bracket $<< 1$) and all nodes will use simple Hebbian learning to learn the first Principal Component direction. But note that when they have done so, (or are doing so) they enter a region where the this interaction is not insignificant and so some interaction between output neurons begins to take place - the neurons have learned their way out of the uniform region of weak outputs.

Neuron outputs are in the intermediate range This is the effective range of learning for the Competitive Hebbian Algorithm. A winning node - one which happens to be firing strongly - will tend to turn off the learning for the other nodes. This will result in different neurons learning different regions of the input space.

Neuron outputs are too strong When the competitive learning factor (the bracketed term) becomes negative, no learning takes place since we simply make the last term 0. If a system gets to this stage, there is no way it can learn its way out of the region and we must simply stop the algorithm and restart.

The Competitive Hebbian Algorithm had some success on our horizontal and vertical lines problem but has proved difficult to stabilise and indeed the actual parameters used in the algorithm have to be hand tuned for a particular problem to get the best results.

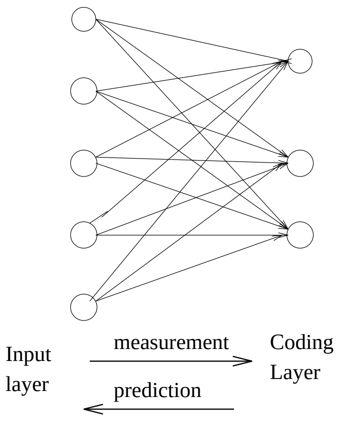


Figure 6.5: Saund's model has activation propagating forwards and prediction propagating backwards.

6.5 Multiple Cause Models

The algorithms discussed in this chapter are sometimes known as "Multiple Cause Models" since the aim is to discover a vocabulary of independent causes or generators such that each input can be completely accounted for by the action of a few of these possible generators e.g. that any set of the 2^{16} possible patterns can be accounted for by some combination of the 16 output neurons, each of which represents one of the 16 possible independent inputs.

6.5.1 Saund's Model

Saund has focussed his attention on the activation function. His network is shown in Figure 6.5. We have a binary input vector $\{x_1, x_2, ..., x_j, ..., x_n\}$ which is joined by weights w_{ij} to an output/coding layer $\{ y_1, ..., y_i, ..., y_m\}$. The weights are thought of as associating an input vector with activity at a cluster centre determined by the weights into that centre. This should remind you of competitive learning which is the simplest way to view this part of the process.

The cluster centre then corresponds to an output neuron which is in a position to predict the input that caused it to react. This prediction is then returned to the input layer. Saund views the outputs as performing a voting rule associating each prediction with the firing of the various output neurons: if the i^{th} neuron is firing strongly, then it must be because the inputs to it approximate the data vector which is currently being presented to the network. We can view such firing as the probability that it is caused by an item of data which is close to the weights into that neuron. Thus he defines the prediction of the network for the j^{th} input value as

$$r_j = 1 - \prod_k (1 - w_{kj}.y_k) \tag{6.12}$$

Clearly if all weighted sums of the outputs are close to zero then the prediction is close to zero. On the other hand as soon as a single weighted feedback is close to 1, the prediction of the network goes to one - rather like the neuron saying "I've found it". The prediction function for 2D inputs is shown in Figure 6.6. It is sometimes known as a noisy-OR function since it is a smoothing of the Boolean OR function.

Saund identifies an objective function equal to the log likelihood (derived in a similar way to that discussed in the last chapter)

$$g_i = \sum_j \log\left(x_{i,j} r_{i,j} + (1 - x_{i,j})(1 - r_{i,j})\right) \tag{6.13}$$

where the i subscript identifies the i^{th} pattern and the j the j^{th} input neuron. If $x_{i,j}$ and $r_{i,j}$ simultaneously approach 1 or 0 then the objective function tends to zero. At all other times the

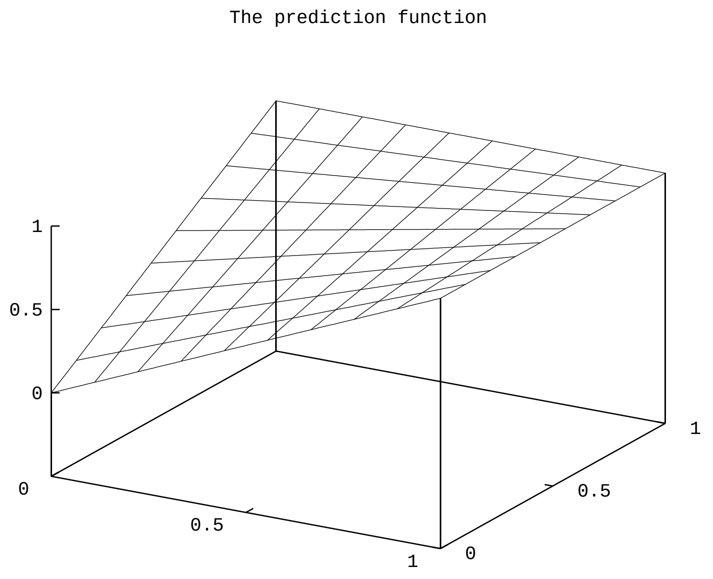


Figure 6.6: The prediction function for a two dimensional data set.

objective function will be negative. A global objective function is merely the sum of these objective functions over all input patterns and the optimal weights are found by batch gradient ascent.

Saund shows that the network will identify the independent horizontal or vertical lines and points out the inter-dependencies in the training regime: the optimal weights cannot be calculated independently for each hidden unit but are dependent on one another's predictions.

On the left hand side of Figure 6.7, we show a single neuron with cluster centre (1,1,1) is attempting to predict the pattern (1,1,0). The best it can do is to predict (0.666,0.666,0.666) which minimises the error function. However a second neuron with centre (1,1,0) can minimise this error (in fact send it to zero) when it comes on line leaving the first neuron free to continue responding maximally to (1,1,1). Note that each neuron has responded to a different pattern and each is minimising the error for a different part of the input space.

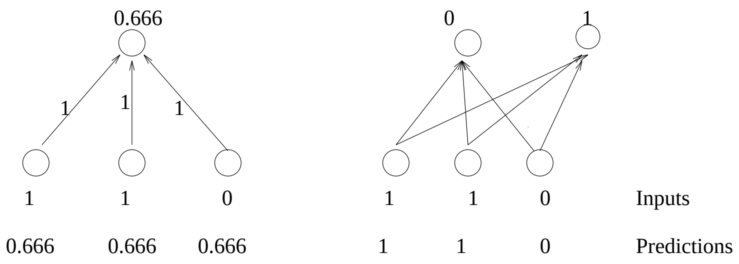


Figure 6.7: If there is only a single cluster centre at (1,1,1) the output neuron cannot respond accurately to the input (1,1,0) since the incorrect prediction of a 1 on the third input would give an error of magnitude 1. The best that can be done is to predict $\frac{2}{3}$ for each input. But if we have two centres, the second centre can concentrate its resources on this input leaving the other centre to react to the pattern (1,1,1).

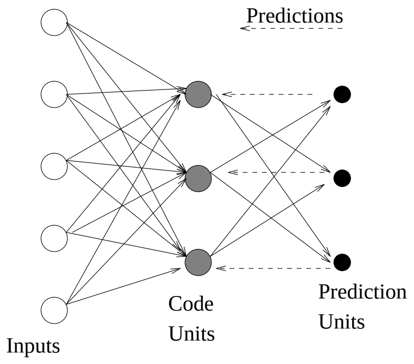


Figure 6.8: Schmidhuber's Predictability Minimisation network: input patterns are coded across the (large) code neurons. Each code neuron feeds forward to all the other prediction units *but not its own*. Each prediction unit attempts to predict the corresponding code unit's output. Each code neuron is meanwhile attempting to avoid being predictable.

6.5.2 Dayan and Zemel

Dayan and Zemel view the problem as one of finding a set of prior probability and conditional probabilities that (approximately) minimize the description length of a set of examples drawn from the distribution: the minimum description length refers to the number of output neurons required to describe the input in an invertible code. Their first model uses a backpropagation network with a single hidden layer for autoassociation. They use, however, a cross-entropy error to judge the accuracy of the reconstructions at the output layer. Using this on the lines data, they had some success but found that the network tended to get stuck in a local minima about 73% of the time in which a particular output would have responsibility for more than one line.

6.6 Predictability Minimisation

Schmidhuber has developed a network (Figure 6.8)for extraction of independent sources based on the principle of predictability minimisation. The core idea is that each prediction layer (the last layer in the Figure) attempts to predict each code neuron's output based only on the output of the other code neurons while simultaneously each code unit is attempting to become as unpredictable as possible by representing input properties which are independent from the features which are being represented by other code neurons.

Notice that each prediction neuron is connected in a feedforward manner to all code neurons *except* the code neuron whose output it is attempting to predict. Let the output of the i^{th} code neuron be y_i. Then

$$y_i = f(\sum_j w_{ij} x_j) \tag{6.14}$$

where $f(u) = \frac{1}{1+\exp(-u)}$ and x_j is the j^{th} input. So $0 \leq y_i \leq 1$. The output of the prediction units is given by

$$P_k = \sum_{j \neq k} v_{kj} y_j \tag{6.15}$$

So P_k sees only the response of the other code neurons and not the code neuron which it is trying to predict. Now we have two pass training (both using conventional backprop).

Pass 1 Each set of weights is trained to minimise the mean squared prediction error over all sets of inputs. i.e. to minimise $\sum_p \sum_i (P_i^p - y_i^p)^2$, where P_i^p is the output of the i^{th} prediction neuron to the p^{th} input pattern etc. If it does minimise this function, the P_i will then be the conditional expectation $E(y_i|\{y_k, k \neq j\})$ of the prediction unit given the output of all

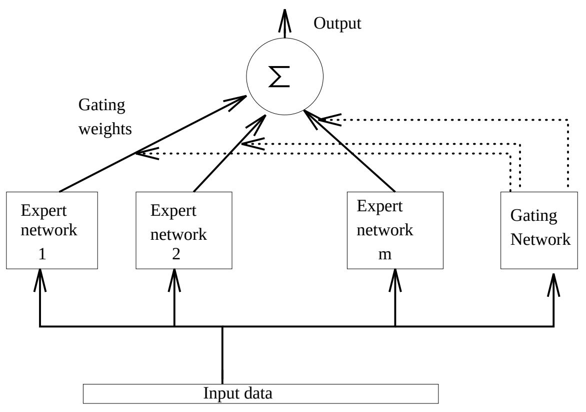


Figure 6.9: Each expert and the gating network sees all input data. Each expert learns to take responsibility for a particular set of input data and the gating expert learns how much confidence to have in each network's output for that input.

the other code neurons. We can see that this conditional expectation will not be the same as actually predicting the value of the code unit. e.g. if the code unit fires 1 one third of the time and 0 the other two thirds, the conditional expectation would be 0.333 in this context.

Pass 2 Now in the second pass the weights are again adjusted so that the code units are attempting to maximise essentially the same objective function which was previously being minimised i.e. to make them as unpredictable as possible. But now we only need change the w weights into the code units.

The two criteria co-evolve by fighting each other. Note that each w weight is maximising only the local prediction error while each v weight is being updated to minimise the global prediction error.

Schmidhuber states that the code evolved by his network is quasi-binary: each coding neuron is either 0/1 for each pattern or responds with a constant value to each input pattern in which case it is simply giving a "don't know" response to each pattern.

6.7 Mixtures of Experts

We now introduce a method known as the mixtures of experts: the desire is to have each expert (which is typically itself a backpropagation neural network) learn to take responsibility for a particular subset of the input data. The experts are then assigned a weight which can be thought of as the confidence that a gating network has in each expert's responsibility for responding to that input data.

Consider the network shown in Figure 6.9. Each expert sees all the input data and is itself a network which can be trained by supervised learning (error descent) methods. The gating network also sees the input data and is trained by supervised learning. Its responsibility is to put a probability on the chance that each network is the one to take responsibility for that input.

A First Attempt

A first attempt might be to use as error descent on

$$E^c = \| \mathbf{t}^c - \sum_i p_i^c \mathbf{y}_i^c \|^2 \tag{6.16}$$

where $\mathbf{y}_i^c$ is the output vector of expert i on case c , p_i^c is the proportional contribution of expert i to the combined output vector and $\mathbf{t}^c$ is the target output on case c. Notice that it is the gating expert which determines p_i^c while the experts independently calculate their outputs,$\mathbf{y_i}$.

The problem with this error function is that this introduces a strong coupling between the experts causing them to act together to try to reduce the error. So each expert attempts to minimise the residual errors when all the other experts have had their say which tends to lead to situations where several experts are combining to take responsibility for each case.

An Improvement

What we really require is for our gating expert to make the decision about which single expert should be used in each case. This suggests an error term of the form

$$E^c = E(\| \mathbf{t}^c - \mathbf{y}_i^c \|^2) = \sum_i p_i^c \| \mathbf{t}^c - \mathbf{y}_i^c \|^2 \tag{6.17}$$

Here the error is the expected value of the residual error where each expert is expected to provide the whole of the approximation to the target and the gating network evaluates how close this whole approximation is. There will be still some indirect coupling since when another expert changes its weights the gating network may alter its assessment of responsibility apportionment but this is much less direct than before. Simulations have shown that this network can evolve to find independent sources.

When an expert network gives less error than the weighted average of the errors over all experts, its responsibility for that case is increased (the gating network increases the value of p_i^c) and if it does worse, its responsibility is decreased.

There is still a difficulty with this error measure however which we see when we consider the rate of change of the error with the output,

$$\frac{\partial E^c}{\partial \mathbf{y}_i^c} = -2p_i^c(\mathbf{t}^c - \mathbf{y}_i^c) \tag{6.18}$$

from which we can see that the rate of change for terms which are far away from the current target is greater than those closer. While this is gated by the probability vector, it still has an adverse effect on convergence.

The Final Error Measure

Consider the error measure

$$E^c = -\log \sum_i p_i^c \exp\left(-\frac{1}{2} \| \mathbf{t}^c - \mathbf{y}_i^c \|^2\right) \tag{6.19}$$

which can be derived by assuming that all output vectors can be described by a Gaussian distribution and we are maximising the negative log probability of their independent joint distributions. When we now calculate the derivative of the error function with respect to the experts' outputs we get

$$\frac{\partial E^c}{\partial \mathbf{y}_i^c} = -\frac{p_i^c \exp(-\frac{1}{2} \| \mathbf{t}^c - \mathbf{y}_i^c \|^2)}{\sum_j p_j^c \exp(-\frac{1}{2} \| \mathbf{t}^c - \mathbf{y}_j^c \|^2)}.(\mathbf{t}^c - \mathbf{y}_i^c) \tag{6.20}$$

This term has been shown to have much better performance due to the fact that the first fractional term takes into account how well the i^{th} expert is performing compared to all other experts.

6.7.1 An Example

Jacobs *et al* performed an experiment in which 4 vowel sounds from a 75 speakers were to be discriminated by an ANN. They compared the results using the Mixture of Experts network with a backpropagation network and showed that latter typically took about twice as long to train. The actual convergence of the weights in the network is interesting: initially each expert in the network attempts to minimise all the error in the network over every example. There is, at this stage, no

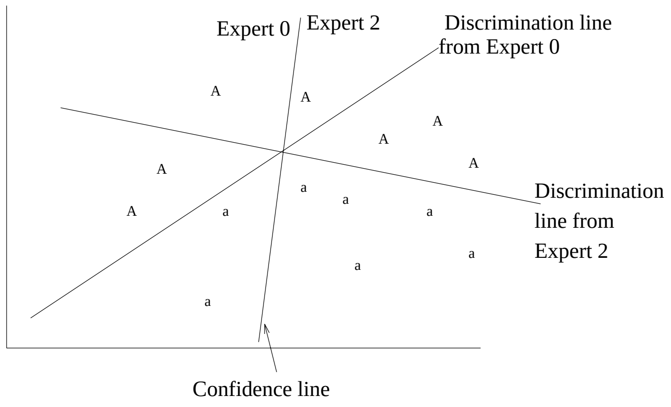

Expert 0
Expert 2
Discrimination line from Expert 0
Discrimination line from Expert 2
Confidence line
A
a

Figure 6.10: An exaggerated view of how the gating network and the mixtures of experts work together. The almost vertical line is the gating networks' view of the experts: on examples to the left of the line it assigns responsibility to Expert 0 while to the right it assigns responsibility to expert 0. Each of these experts have a linear discriminant function which is accurate for its region of responsibility but inaccurate outside. In this way the combined network acts together to give a non-linear boundary between classes.

cooperation in the network and each network is attempting to deal with the whole problem itself. However a stage is reached when one network begins to receive more cases from a particular class than the others and quickly assumes responsibility for this class. This leaves the other experts free to concentrate on error minimisation for the other classes.

Finally we see an exaggerated version of converged weights in Figure 6.10 where we have experts and the gating network working together to model the problem. It can be seen that each of the networks has a region of the input space where it is effective and a region where it is ineffective. The gating network has learned these regions and assigns responsibility for the network's output to the expert for problems in the area where the expert is effective.

6.8 The Paisley Connection

6.8.1 Non negative Weights and Factor Analysis

Charles and Fyfe have shown that a simple change to the Subspace Algorithm results in a network which will find the independent components of the bars data: we simply do not allow the weights to go negative. The rationale for this rule is that neurons can inhibit other neurons or excite other neurons but what cannot happen is that a neuron which at one time excites another neuron changes its synaptic connections (weights) so that it now inhibits that neuron. Recall that we used this simple rule with Gaussian data and showed that this rule enabled convergence to the actual Principal Components of the simple Gaussian data. We now state that this change leads to the weights converging to the bars of the bars data.

Table 6.2 shows an example of the patterns typically learned (each output identifies a separate bar).

A standard method of finding independent sources of this type is the statistical technique of Factor Analysis (FA). PCA and FA are closely related statistical techniques both of which achieve an efficient compression of the data but in a different manner. They can both be described as methods to explain the data set in a smaller number of dimensions but FA is based on assumptions about the nature of the underlying data whereas PCA is model free.

We can also view PCA as an attempt to find a transformation from the data set to a compressed code, whereas in FA we try to find the linear transformation which takes us from a set of hidden factors to the data set. Since PCA is model free, we make no assumptions about the form of the

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0.351583	0.351481	0.342942	0.349674	0.344767	0.344997	0.343597	0.344819
0	0	0	0	0	0	0	0

Table 6.2: The weights into a single output neuron. The weights have been arranged to display their connection with the input grid.

	Output 1	Output 2
Input 1	0.000755	**0.708634**
Input 2	0.001283	**0.705238**
Input 3	0.021043	0.021338
Input 4	**0.708208**	0.002265
Input 5	**0.705604**	0.001692

Table 6.3: The converged weights of the network showing that the underlying sources have been found.

data's covariance matrix. However FA begins with a specific model which is usually constrained by our prior knowledge or assumptions about the data set. The general FA model can be described by the following relationship:

$$\mathbf{x} = L\mathbf{f} + \mathbf{u} \tag{6.21}$$

where $\mathbf{x}$ is a vector representative of the data set, $\mathbf{f}$ is the vector of factors, L is the matrix of factor loadings and $\mathbf{u}$ is the vector of sp ecific (unique) factors.

We now view the output vector, $\mathbf{y}$, in our network as the vector of factors. It is well known that factor analysis does not give us a unique set of factors or loadings since any rotation of the factors and a corresponding rotation of the factor loadings explains the data set equally efficiently. Therefore, typically the underlying factors are constrained dependent on the researcher's prior knowledge or the type of solution is constrained by some more or less *ad hoc* arrangement.

As an extension to the factor analysis on discrete data, we use 5 mixtures of sine waves as input data to our network so that

$$\begin{aligned} x_0 &= \sin(t) + \sin(2t) \\ x_1 &= \sin(t + \frac{\pi}{8}) + \sin(2t + \frac{\pi}{4}) \\ x_2 &= \sin(3t + \frac{3\pi}{7}) \\ x_3 &= \sin(4t + \frac{4\pi}{3}) + sin(5t) \\ x_4 &= 2\sin(5t) \end{aligned}$$

The first two mixtures, x_0 and x_1, are identical but slightly out of phase, the 3rd is a totally independent sine wave and the last two contain the same sine wave, however one has another mixed on to it. Therefore the relationship between the outputs of the sources is straightforward in the case of x_3 and x_4 but time-varying in the case of x_0 and x_1 where the underlying source is emitting different phase signals.

6.8.2 Non negative Outputs

A further possibility is to only allow the outputs to be non-negative: this has the same effect of forcing the weights to learn the individual bars.

A final possibility is to use a function of the outputs in the nonlinear PCA algorithm which ignores negative weights. e.g.

$$\begin{aligned} \Delta w_{ij} &= \eta\{y_i x_j - y_i \sum_k w_{kj} y_k\} \\ \text{where } y_i &= \exp(\sum_j w_{ij} x_j) \end{aligned}$$

An alternative function is

$$y_i = \frac{1}{1 + \exp(\{-\sum_j w_{ij} x_j\} + a)} \tag{6.22}$$

where a is chosen to move the sigmoid parallel to the x-axis so that it is close to 0 at x=0 and gets closer to 0 as x becomes more negative.

All of these methods used with the subspace rule cause convergence of the weights to find the individual bars.

We may show that for every n-dimensional space, there exists a basis with n+1 basis vectors in which every point can be represented by positive coordinates. i.e. intuitively, if we have a 3-dimensional space, we can choose an origin and then every point can be coded with 4 positive numbers which will explicitly identify its position. Such bases are known as overcomplete bases - you have more vectors than you need to form a basis.

6.8.3 Additive Noise

However, the outputs of the network as described in the previous section will learn partial bars when there are more outputs than causes - i.e. the bars are shared between the outputs. If there are sixteen causes in the input space then only sixteen outputs should be used to code these individual causes regardless of the dimensionality of the output space. By adding noise to our network after the application of the non-linearity we can ensure that only as many outputs respond to the data as there are causes. An added bonus is that this may be interesting from a biological perspective as real neurons tend to operate in a noisy environment. There are already a number of a biologically plausible aspects to the network such as Hebbian learning, local learning, sparse coding, as well as thresholding and the possibility of topographical mapping.

Non-linear PCA can be shown to be an approximation to the minimization of $\mathbf{J} = \boldsymbol{E}(\mathbf{x} - \mathbf{W}\mathbf{y})^2 = \boldsymbol{E}(\mathbf{x} - \mathbf{W}\mathbf{f}(\mathbf{a}))^2$, where E() is the expectation operator. Now we add noise to the process so that $\mathbf{y} = \mathbf{f}(\mathbf{a}) + \mu$ where μ is a vector of independently drawn noise from a zero mean distribution. So defining $\mathbf{f} = \mathbf{f}(\mathbf{a})$, let

$$\begin{aligned} \mathbf{J}' &= E(\mathbf{x} - \mathbf{W}\mathbf{y})^2 && (6.23) \\ &= J + \sum \sigma_i^2 \mathbf{w}_i^2 && (6.24) \end{aligned}$$

Terms containing single expectations of μ can be removed from the equation as they are drawn from zero mean noise and thus are independent of $\mathbf{x}$. Intuitively, we see that, with a low magnitude of added noise to the network the first term of (6.24) dominates and so non-linear PCA is performed in the normal manner. If the noise level is increased then the learning is moderated by this additional weighted noise term which has the effect of forcing some weight vectors to zero (the degenerate solution). Extracting a weight update rule from the energy equation (6.24), we have

$$\Delta\mathbf{W} \propto \mathbf{f}(\mathbf{a})(\mathbf{x} - \mathbf{W}\mathbf{f}(\mathbf{a})) - \mathbf{\Lambda}\mathbf{W}^{\mathbf{T}} \tag{6.25}$$

As the amplitude of the noise becomes larger then the right term has more effect and pushes the weight values downwards. So the introduction of the noise into the network at the outputs has the effect of introducing a second pressure into the learning rule of the non-linear PCA algorithm. This is a natural way in which to introduce a sparsification term on to the weights.

The major advantage of additive noise is that it reduces the number of basis terms used to code an input: we are finding the Minimum Overcomplete Basis.

Additive noise has a further advantage in that it can be added in a number of ways. For example, if it is added uniformly on to all outputs (the diagonal values of $\mathbf{\Lambda}$ matrix are all the same) then all of the weight vectors are penalised equally and only features that are strong enough to dominate the noise are learned. So in the case of the bars if we have 20 outputs in are network trying to learn 16 individual bars then only 16 outputs will respond, the weight vectors connected to the other 4 all having zero weight values.

Noise may also be added in a graduated manner, i.e. the first output is given a small magnitude of noise (e.g. zero mean, gaussian noise of standard deviation 0.001) and this is increased by *noise* x *output number* (where the first output is number one and the last is twenty) for every subsequent output. That is the diagonal values of $\mathbf{\Lambda}$ matrix become increasingly larger. So if the first output is on the left then the effect of adding noise in the manner is to force the bars to be learned at the first 16 outputs starting from the left side. In this way we can begin control the location in which factors are learned. In Section 6.8.5, we show how that high magnitude noise can be added locally on modules of the output space to force features to be learned on the outputs between the areas of high magnitude noise. By introducing temporal context, we can then force the network to learn vertical and horizontal bars in separate areas of the output space. Once we have separate 'modules' of the output space responding to these different types of bar then it is easy to from the concepts of 'vertical' and 'horizontal' at a higher processing level.

6.8.4 Topographical Ordering and Modular Noise.

By introducing temporal context via lateral connections between the outputs (6.11), we can achieve an ordering in the coding of the bars at the outputs. This is achieved by giving values to the lateral weights proportional to the distance between any particular pair of outputs and increasing an output's activation in proportion to the previous values of the other outputs weighted by the lateral connections. The lateral connections come into play after the feedforward stage of the algorithm but before the application of the non-linearity. So $a_i \leftarrow a_i(t) + \sum_{j=-k, j\neq 0}^{k} (a_{i+j}(t-1)\tau_j)$, where τ_j is a lateral weight. The lateral weights can be set in a number of ways, for example using a gaussian or difference of gaussians, or gaussian weighted cosines or sines, however the method that we use here is simply to connect an output only to its nearest four neighbours. Lateral weight values to the left of an output are -0.2 and -0.7 (nearest) and to the right 0.2 and 0.7 (nearest). If the bars appear as part of horizontally or vertically moving sequences then this network now can order the outputs so that temporally close features are coded spatially close at the output neurons. Now using these lateral connections along with modular noise we can force vertical and horizontal bars to be learned in different modules of the output space. This is achieved by creating two "wells of attraction" at the outputs by adding zero mean gaussian noise after the application of the non-linearity proportional to $|\cos(2 * \pi *$(number of output/total number of outputs$|$ where each of the outputs are numbered 1,2,3 ... . This has the effect of encouraging one set of features to be coded around the output that is one third from the left and one set of features to be coded around the output that is one third from the right. This happens because the noise is of lower magnitude at these points and so the error minimisation term dominates.

6.8.5 Results

In this section a variety of aspects of our network are illustrated with the benchmark "bars data" before testing on real image data. Unless otherwise stated the the networks are trained over 50000 presentations of the data with a learning rate of 0.05 which is annealed linearly to zero over the training period. The squares in the following figures are individual weight vectors each connecting to one output, arranged 2-dimensionially for the convenience of viewing the results. The diameter of circles within the squares represent the individual weight values where black is a positive weight and white is a negative weight.

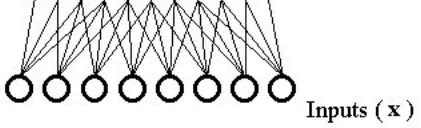


Figure 6.11: Lateral connections are added between neighbouring outputs and the activations of these are fed to their neighbour at the same time as the neighbouring output receives the feedforward signal from the inputs. In this way temporal context is incorporated into the network so as to encourage neighbouring outputs to learn factors which are temporally close on the inputs. Note that in this example only the closest neighbours are connected.

Random Mixes of Horizontal and Vertical Bars

We use the more difficult form of this data in horizontal and vertical bars may appear together (Note that we have already shown this network to be resistent to noisy versions of this data). Each bar (horizontal or vertical) may appear at the input to the network with a probability of 1/8 as described previously.

The converged weights of our network when using using the straightforward rectification of the outputs is shown in Figures 6.12 and 6.13. In the case where 24 outputs are used then all of the bars are found but some of the bars are shared by two or three outputs. The threshold implementation of the network is more successful at identifying the bars in this case, Figures 6.14 and 6.15 show the converged weights when using this implementation of the network on the data. That is when more outputs than bars are used in the network then all of the individual bars are indentified, redundant weight vectors simply contain noisy values on the weights. We use the *soft threshold* function with the artificial data here as it is very flexible to work with and forgiving of non-optimal network parameters. The other threshold functions from this family once optimised, however, yield results that are virtually indentical.

We have found that the soft threshold non-linearity is more effective than the plain rectification on the outputs. With exactly as many outputs as bars then both networks identify all of the bars easily, whereas when there are more outputs in the network than bars that make up the data set then all of the bars are identified but in the case of the rectified network some bars are identified

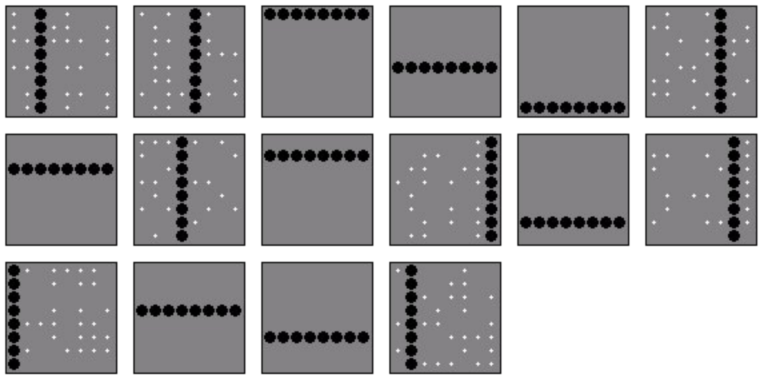


Figure 6.12: Converged weights of the non-linear PCA network with the straightforword rectification $[a]^+$ - 16 outputs

more than once, and junctions of of bars or combinations of bars are also found.

6.8.6 Additive Noise

Additive noise, as can be seen from the results presented here below (zero mean, gaussian noise of standard deviation 0.01 is added to every output) is beneficial in all of these networks when added to the outputs after the application of the non-linearity. Figure 6.16 show that the additive noise enables each of these networks to identify all of the individual bars. That is, only as many outputs are used as are required in the coding; the weights connected to the other outputs each learn values that are close to zero.

As stated earlier we can add noise in a graduated way across the outputs so that the first output has zero mean gaussian noise of standard deviation 0.001 added to it and every subsequent output having double the amount of noise as the previous output. In a network with 20 outputs this has the effect of forcing the first 16 outputs to learn all of the bars but the last 4 to learn nothing. In this way we can control the location on the outputs where factors may be learned.

Illusory Causes

In the situation where bar patterns are non-sparse (the bars appear with a random probability of 7/8 each) then networks with built in sparse priors cannot be expected to identify the individual bars. With our threshold network the results (Figure 6.17) indicate that the network converges to a very sparse representation - i.e. the illusory bars between the actual bars patterns. As the network operates so as to find a sparse response with the minimum descriptive length and because the individual bars are appearing in dense patterns together, then the network cannot learn the individual bars. Instead the network learns the spaces between bars patterns which is the appropriate sparse response (illusory bars). Note that in this experiment that the horizontal

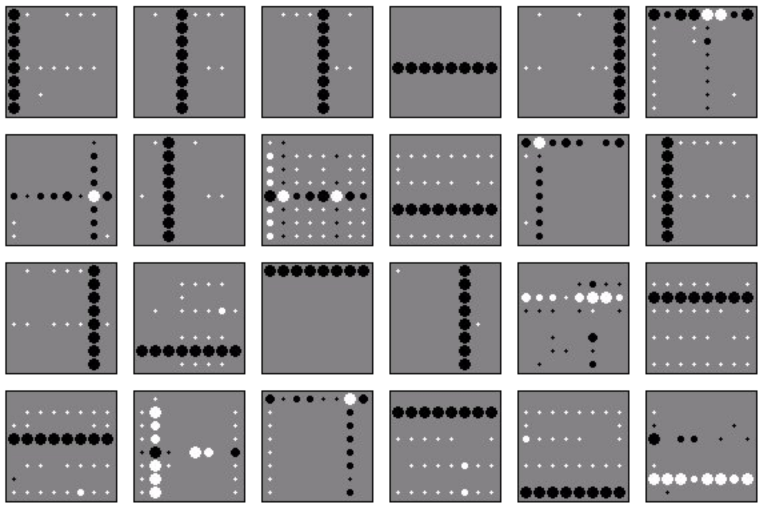


Figure 6.13: Converged weights of the non-linear PCA network with the straightforword rectification $[a]^+$ - 24 outputs

and vertical bars are not mixed so as to allow the weights to learn a more visually interesting response.

Although most of the weight vectors have prominent negative values, one of the weight vectors has small positive values on all of its weights. This weight vector is used to ensure that the output values respond with a significant positive value (normally a magnitude of over 3) to the illusory bars.

Using Noise to Modularise the Network Response

Additive noise on the outputs has another useful property - that of enabling the network to learn related features in modules of output space. We create two "wells of attraction" at the outputs of the lateral connected network (Section 6.8.4) by adding local zero mean gaussian noise after the application of the non-linearity proportional to |cos(2*π*(number of output/total number of outputs| where each of the outputs are numbered 1,2,3 ... . This has the effect of encouraging one set of features to be coded around the output that is one third from the left and one set of features to be coded around the output that is one third from the right (6.18). This happens because the noise is of lower magnitude at these points and so the error minimisation term of (3) dominates.

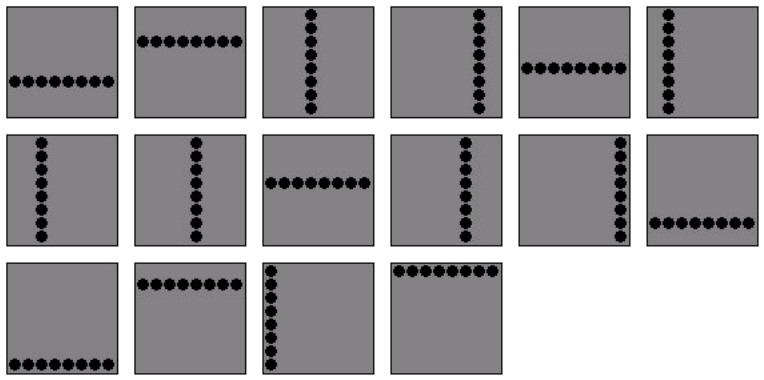


Figure 6.14: Trained threshold network - 16 outputs

Real Image Data

In this section we test our network on some real images with noise added in a variety of ways to the outputs of the network before feedback. We use the shifted logistic function here because, as it saturates at '1', the visual effect of reconstructed patches having different brightnesses is less prominent. The value for the threshold parameter θ that we use in the non-linearity is 4 and we set the slope parameter $\sigma = 0.4$. The learning rate is kept the same in all of the experiments at 0.05 and training conducted over 100 000 samples of the input images (8x8 samples). In each case the learned codes are tested on a new but similar image by sampling (again 8x8 patches) over this image completely so that each area is covered at least once and the sampled patches overlap by 6 pixels (this is used to even out the contrast between neighbouring reconstructed image patches - it helps with the visualisation process but has a slightly negative effect on the reconstruction error).

Striped Wood Data

The first real image that we test our network with is an image of striped wood, shown in Figure 6.19. The structure in this example is very clear, i.e. it is made up largely of slightly off-vertical stripes. One would expect in this case that the structure in this data set could be learned by only a few outputs in a network. The results of using a PCA network confirm this to some extent because we find that there are only 5 or 6 significant principal components (Figure 6.19). By adding noise either uniformly (Figure 6.20) or in a graduated manner (Figure 6.21) to our shifted logistic network we can force the network to learn only the most significant causes in the data. The reconstruction of the striped wood image is shown in each of the following examples; this is to show that even though the reconstruction of the image with fewer codes is less detailed the structure of the original image is still clearly visable. Our purpose is not to form codes that can reconstruct the image with minimal error but to identify the fundamental underlying factors in the data. For example we do not normally want to reconstruct noise in an image. It could be said

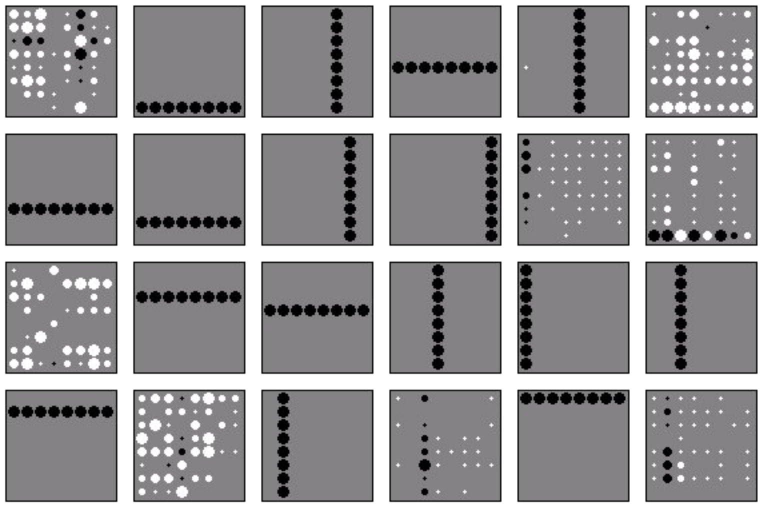


Figure 6.15: Trained threshold network - 24 outputs

that we are adding noise to remove noise from an image.

6.9 Probabilistic Models

Hinton, Gharamani, Dayan *et al* have developed models which attempt to use a generative model = top down connections from underlying reasons for the input image i.e. the top down connections create the image from an abstraction of the image. This is based on the view that "Visual perception consists of inferring the underlying state of the stochastic graphics model using the false but useful assumption that the observed sensory input was generated by the model."

Learning is done by maximising the likelihood that the observed data came from the generative model. The simplest generative model is the Mixtures of Gaussians model.

6.9.1 Mixtures of Gaussians

- Each data point has an associated probability of being generated by a mixure of Gaussian distributions.
- Given the current parameters of the model, we calculate the probability that any data point

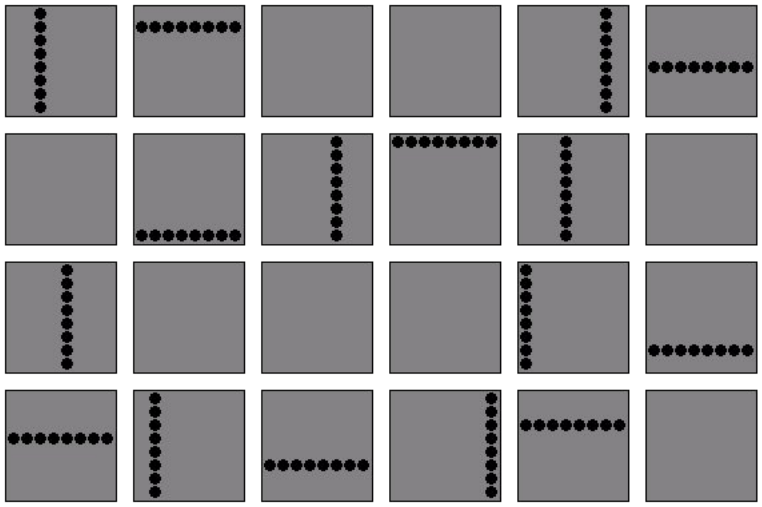


Figure 6.16: Trained soft threshold network with additive noise - 24 outputs. Note that some weight vectors have no significant weight values

came from the distributions. *The posterior probability*

- The learing process adjusts the parameters of the model - the means, variances and mixing proportions (weights) of the Gaussians - to maximise the likelihood that the model produced the points.

So when we generate a data point

- Pick a hidden neuron (the underlying cause of the data). Give it a state of 1, set all other hidden neurons' states to 0.
- Each hidden neuron will have probability of being picked of π_j - *a prior probability.*
- Feed back to input weights through weight vector $\mathbf{g}_j$. The $\mathbf{g}_j$ is the center of the Gaussian $= \{g_{j1}, g_{j2}, ..., g_{jn}\}$.
- Add local independent zero mean Gaussian noise to each input.
- This means that each data point is a Gaussian cloud with mean $\mathbf{g}_j$ and variance σ_i^2.

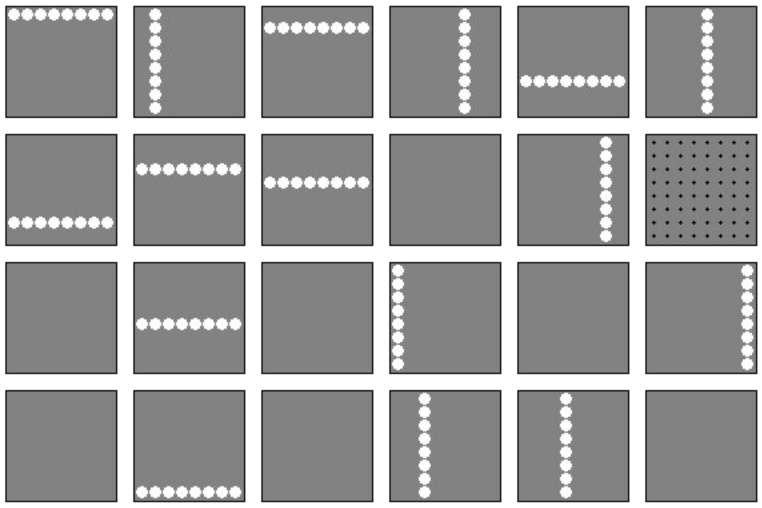


Figure 6.17: The noisy soft threshold network discovers illusory bars - 24 outputs

$$p(\mathbf{d}) = \sum_j \pi_j \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} e^{-(d_i - g_{ji})^2/2\sigma_i^2} \tag{6.26}$$

Interpreting Data - Expectation Step

1. Compute the probability density for each data point (assuming the model is correct.)

$$p(\mathbf{d}|s_j = 1) = \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} e^{-(d_i - g_{ji})^2/2\sigma_i^2} \tag{6.27}$$

2. Weight these with the prior probabilities π_j.

3. Use Bayes theorem to calculate the probability of the data.

$$p(s_j = 1|\mathbf{d}) = \frac{\pi_j p(\mathbf{d}|s_j = 1)}{\sum_k \pi_k p(\mathbf{d}|s_k = 1)} \tag{6.28}$$

We have now calculated the posterior probabilities of the hidden states given the data ("perceptual inference") - the E-step of the EM Algorithm and we now perform the M step, which involves changing the parameters to maximise the Expectation.

Learning as Expectation Maximisation

We use the existing parameters to calculate the optimal new parameters.

$$\begin{aligned} \mathbf{g}_j &= \frac{E\{p(s_j = 1|\mathbf{d})\mathbf{d}\}}{E\{p(s_j = 1|\mathbf{d})\}} \\ \sigma_i^2 &= \frac{E\{p(s_j = 1|\mathbf{d})(d_i - g_{ji})^2\}}{E\{p(s_j = 1|\mathbf{d})\}} \\ \pi_j &= E\{p(s_j = 1|\mathbf{d})\} \end{aligned}$$

We have now a means of maximising the expectation - the M-step but we can also use incremental learning which involves gradient ascent such as

$$\Delta g_{ji} = \epsilon p(s_j = 1|\mathbf{d})(d_i - g_{ji}) \tag{6.29}$$

6.9.2 A Logistic Belief Network

Multiple layers of binary stochastic neurons whose state s_j is based on top down expectations $\hat{s}_j$ from the layers above.

$$p(s_j = 1) = \hat{s}_j = \sigma(g_{0j} + \sum_k s_k g_{kj}) \tag{6.30}$$

If the configuration of a network is α,

$$P^\alpha = \prod_i p(s_i^\alpha | pa(i, \alpha)) \tag{6.31}$$

where $pa(i, \alpha)$ is the states of i's parents in configuration α and s_i^α is the state of the i^{th} neuron in configuration α.

Using negative log probabilities as an energy measure, we have

$$E^\alpha = -\ln P^\alpha = -\sum_u (s_u^\alpha \ln \hat{s}_u^\alpha + (1 - s_u^\alpha)\ln(1 - \hat{s}_u^\alpha)) \tag{6.32}$$

where $\hat{s}_u^\alpha$ is the top-down expectation for unit u.

We use the ratio of

$$\Delta E_u^\alpha = E^{\alpha|s_u=0} - E^{\alpha|s_u=1} \tag{6.33}$$

to chose the new state of the u^{th} neuron:

$$p(s_u = 1|\alpha) = \sigma(\Delta E_u^\alpha) \tag{6.34}$$

Hinton shows that this can be factorised into top down effects and knock on effects from below.

Given Gibbs sampling,

$$\Delta g_{ji} = \epsilon s_j (s_i - \hat{s}_i) \tag{6.35}$$

6.9.3 The Helmholtz Machine and the EM Algorithm

As prelude to the discussion in the next a discussion of the Helmholtz machine will be useful. The stochastic Helmholtz machine consists of a pair of Bayesian networks that are fitted to training data using an algorithm that approximates generalised Expectation Maximisation (EM).

The EM algorithm is a general approach to iterative computation of maximum-likelihood estimates when the observed data can be viewed as incomplete data. The term incomplete data has two implications :

1. The existence of two sample spaces X and Y represented by the observed data vector x and the complete data vector y, respectively.

2. The one-to-many mapping $y \rightarrow x(y)$ from space Y to X.

The complete vector y is not observed directly, but only through the vector x. The EM algorithm is executed in two steps: first an expectation step (E), followed by a maximisation step (M). During the E step the complete-data log-likelihood, P(Y—x,M) , given the observed data vector x and the current model, M , is calculated.

The Helmholtz machine then, based on a generalised EM algorithm, has a generative network P(x,y—V) and a recognition network Q(y—x,W), where V and W may be thought of as generative and recognition weight parameters respectively. The recognition model is used to infer a probability distribution over the underlying causes from the sensory input. The separate generative model is used to train the recognition model.

6.9.4 The Wake-Sleep algorithm

The Wake-Sleep algorithm was designed as an improvement on the Helmholtz machine. The main disadvantage of the Helmholtz machine is that a recognition network that is compatible with the generative network must be estimated and this is often a difficult task. In the Wake-Sleep algorithm rather than using Gibbs sampling, a separate set of bottom-up recognition connections are used to pick binary states for units in the layer below. The learning for the top-down generative weights is the same as for a Logistic Belief Net. This learning rule follows the gradient of the penalised log-likelihood where the penalty term is the Kullback-Liebler divergence between the true posterior distribution and the distribution produced by the recognition process. The penalised log-likelihood acts as a lower bound on the log-likelihood of the data and the effect of learning is to improve this lower bound. In attempting to raise the bound, the learning tries to adjust the generative model so that the true posterior distribution is as close as possible to the distribution actually computed. The recognition weights are learned by introducing a sleep phase in which the generative model is run top-down to produce fantasy data. The network knows the true causes of this fantasy data and attempts to maximise the log-likelihood of recovering these causes by adjusting the recognition weights. Frey provides a clear description of the mathematical process for the Wake-Sleep algorithm which may be summarised by the following analysis.

The Rectified Gaussian Belief Net

Hinton and Ghahramani's network is based on the Rectified Gaussian Belief Network (RGBN) which is well described in and and is an improvement in some ways on the Wake-Sleep algorithm. The RGBN uses units with states that are either positive real values or zero, so it can represent real-valued latent variables directly. The main disadvantage of the network is that the recognition process requires Gibbs sampling.

The generative model for the RGBN consists of multiple layers of units each of which has a real-valued unrectified state, y_j, and a rectified state, $[y_j]+ = \max(y_j, 0)$. The value of y_j is gaussian distributed with a standard deviation σ_j and a mean $\hat{y}_j$ that is determined by the generative bias g_{0j}, and the combined effects of the rectified states of units, k, in the layer above:

$$\hat{y}_j = g_{oj} + \sum_k [y_k]^+ g_{kj} \tag{6.36}$$

Given the states of its parents, then the rectified state $[y_j]+$ has a gaussian distribution above zero, but all of the mass that falls below zero is concentrated in an infinitely dense spike at zero. This form of density is a problem for sampling and so Gibbs sampling is performed on the unrectified states. Now, consider a unit in an intermediate layer of a multi-layer RGBN. With the unrectified states of all the other units in the network, then Gibbs sampling is performed to select a value for y_j according to its posterior distribution given the unrectified states of all the other units. In terms of energies, which are defined as to negative log probabilities, then the rectified states of the units in the layer above contribute a quadratic energy term by determining $\hat{y}_j$. The unrectified states of units, i, in the layer below contribute nothing if $[y_j]+$ is 0, and if $[y_j]+$ is positive then they each contribute because of the effect of $[y_j]+$ on $\hat{y_j}$. The energy function may then be written as

$$E(y_j) = \frac{(y_j - \hat{y}_j)^2}{2\sigma_j^2} + \sum_i \frac{y_i - \sum_k [y_k]^+ g_{ki}}{2\sigma_i^2} \tag{6.37}$$

where h is an index over all of the units in the same layer as j including j itself; so y_j influences the right hand side of this energy function by $[y_j]+ = \max(y_j, 0)$. H and G show that learning rules for generative, recognition and lateral connections may be formed that not only identify the underlying causes in a data set but also that a topographical mapping may also be formed on data sets such as in the stereo disparity problem. Because sampling is used, this method is considerably slower than e.g. Charles' method. Additionally, because of the top-down mechanism of learning it must be assumed that either horizontal mixes or vertical mixes of bars are present (in the case of the bars data) at the inputs, that is there must not be a mix of both types in the data.

Attias provides a review of current probabilistic approaches to Factor Analysis and related areas and Frey provides a comprehensive overview of the probabilistic theory and techniques related to this area of research.

6.10 Conclusion

It is of interest that in each of these models there is a balancing of competing criteria: e.g. cooperation between outputs (finding several causes per input) is balanced with competition (separation of responsibilities for coding different independent sources). This seems to be an essential part of each solution but we still seem to be lacking an overall rationale for this observation.

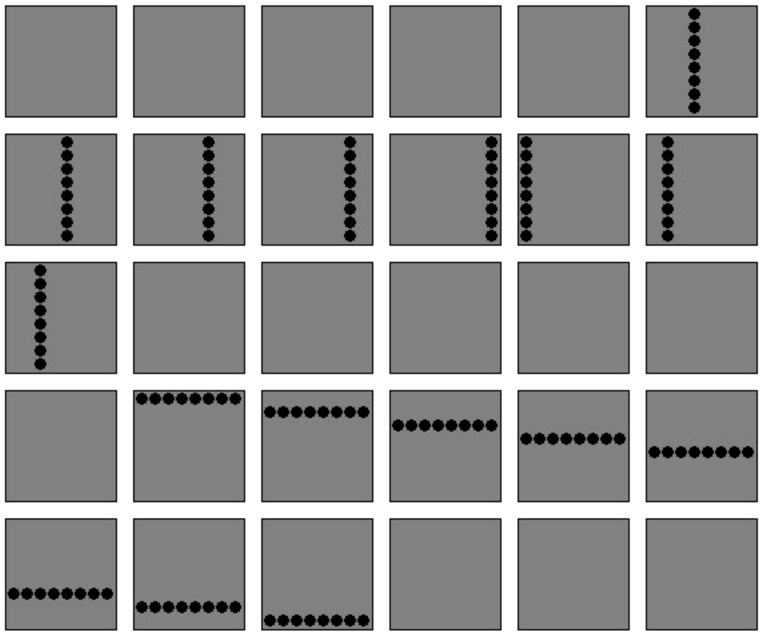


Figure 6.18: Trained weights of noisy soft threshold network using noise to modularize the response of the network

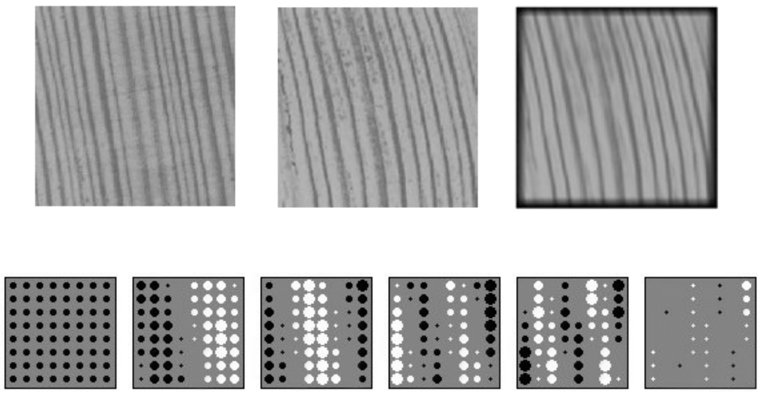


Figure 6.19: The six smaller squares are the visual representation of the weights of a PCA network when trained on the Striped Wood example. There are only 5 significant principal components for this data so only 6 outputs were used in the network. The top three squares from left to right are; the Striped Wood training sample, test sample, and reconstructed image.

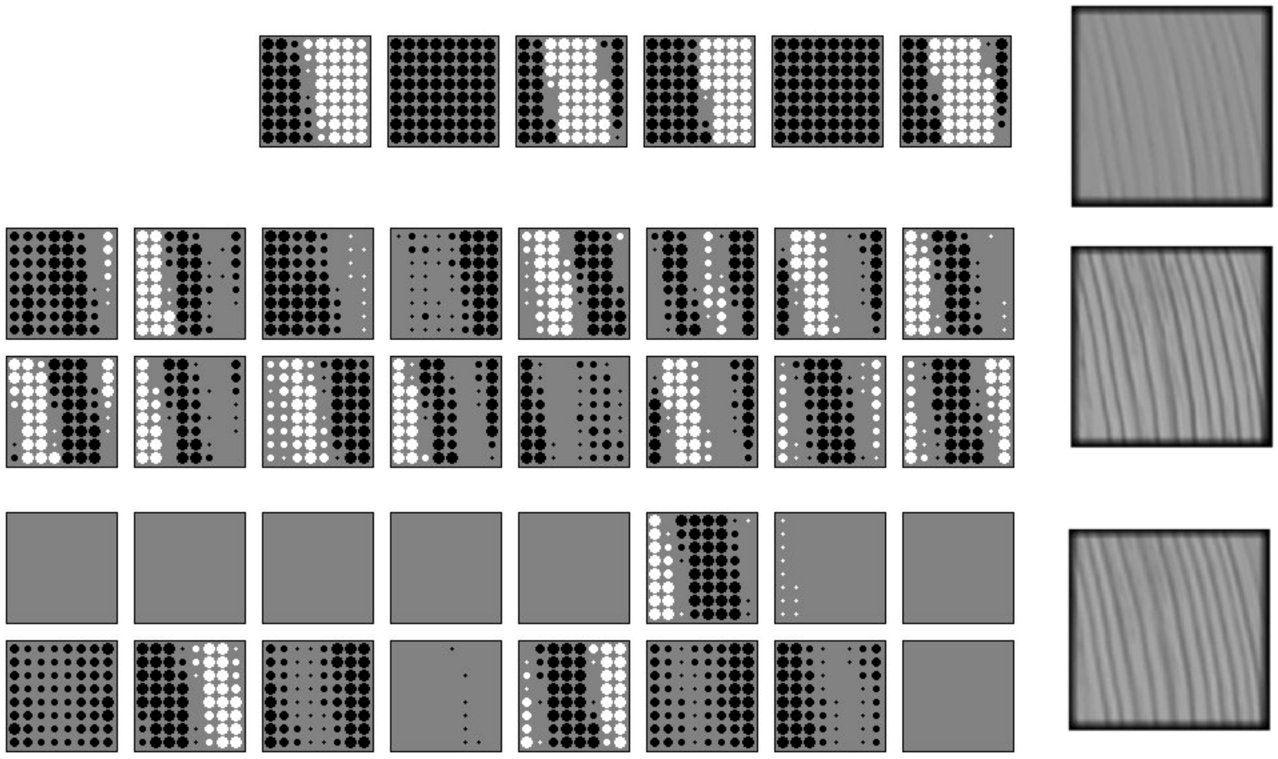


Figure 6.20: The first 2 sets of weight vectors shown above are learned by the shifted logistic network without added noise and are shown with the reconstructed images generated with codes from the test image (the first using 6 outputs, the second 16 outputs). The bottom set of 16 weights are learned when noise is added uniformly on to all of the outputs of the network (zero mean, gaussian noise of 0.05 standard deviation)

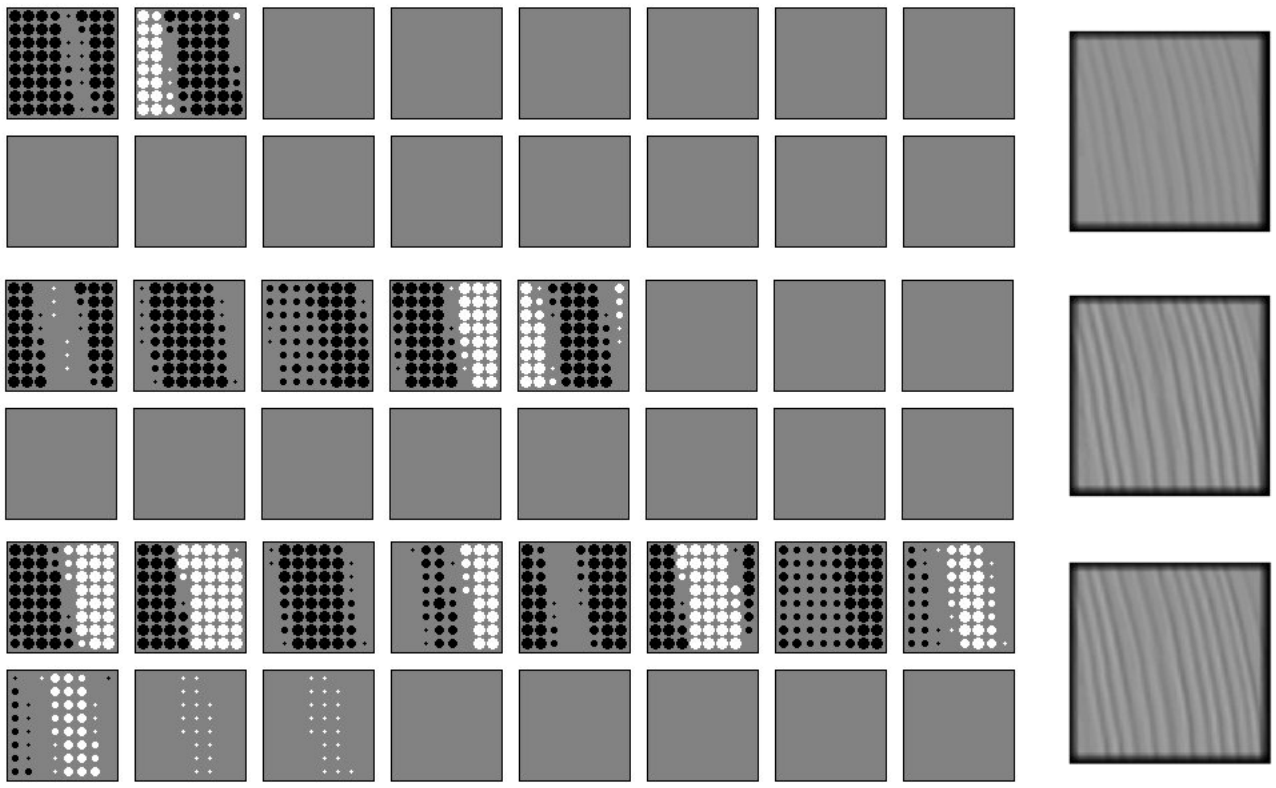


Figure 6.21: This figure shows 3 sets of weights (16 in each set) learned in three different noise implimentations our noisy shifted logistic network when noise is added gradually on to the outputs (also shown are the reconstructed images). The first output recieves the smallest amount of noise and each subsequent output recieves *initial noise*(distance of output from first output + 1)*. The initial noise for the three sets of results shown here are (from top to bottom) - 0.05, 0.02 and 0.005 respectively.

Chapter 7

Independent Component Analysis

There is however a second strand of research using artificial neural networks on this problem of separating out a single source from a mixture of sources. This second strand deals with continuous signals - as opposed to the binary signals we used in the last chapter - and has its roots in the world of signal processing. The problem is generally known as the "blind separation of sources" or sometimes "the cocktail party problem". The latter name is a reference to the human ability to extract a single voice from a mixture of voices: there is no simple algorithmic solution to this problem yet people have no difficulty following a conversation even when the conversation is embedded in multiple other conversations. The former name is in more general use: we wish to separate out a single source signal from a mixture of sources and/or noise. The problem is known as "blind" since we make (almost) no assumptions about the signals.

We will consider only a linear mixture of signals - this problem is difficult enough; polynomial and other non-linear mixtures are beyond the scope of this course (or this lecturer).

The problem may be set up as follows: let there be N independent non-Gaussian signals $(s_1, s_2, ..s_N)$ which are mixed using a (square) mixing matrix A to get N vectors $,x_i$, each of which is an unknown mixture of the independent signals,

$$\mathbf{x} = \mathbf{A}\mathbf{s} \tag{7.1}$$

There may in addition be noise added to the mixing process but we shall ignore that for the time being. Then the aim is to use an artificial neural network to retrieve the original input signals when the only information presented to the network is the unknown mixture of the signals. The weights in the network will be W such that

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{7.2}$$

where the elements of $\mathbf{y}$ are the elements of the original signal *in some order* i.e. we are not insisting that the first output of our neural network is equal to the first signal, the second equal to the second signal and so on. We merely insist that neuron i's output is one of the N signals uncontaminated by any of the other signals. Neural and quasi-neural methods of performing this task are known as Independent Component Analysis networks (ICA) and are often thought of as extensions of PCA networks.

However we did make one assumption when we defined the problem which was that the signals should be non-Gaussian. The reason for this is that if we add together two Gaussian signals we simply get a third Gaussian signal. Therefore if two or more of our signals (or noise sources) were Gaussian distributed there is no way to disentangle them. This is less an assumption than an incontrovertible fact which cannot be side-stepped in our form of life.

7.1 A Restatement of the Problem

Let us take another look at the problem. In Figure 7.1, we show two dimensional data points each of which were drawn independently from the uniform distribution within the parallelogram. The

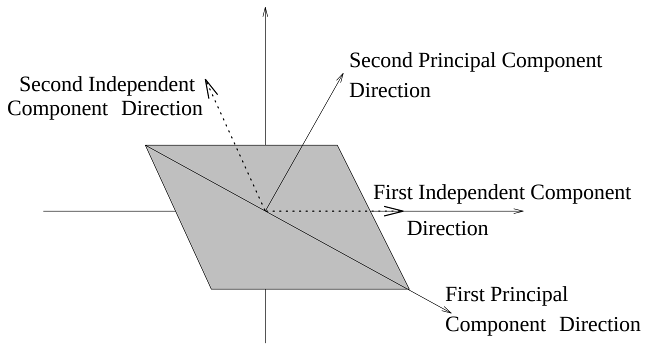


Figure 7.1: The data were points drawn independently from the uniform parallelogram distribution shown. The first Principal Component is the direction with greatest spread - the long axis of the parallelogram. The second is of necessity perpendicular to that. The independent component directions however are parallel to the sides of the parallelogram.

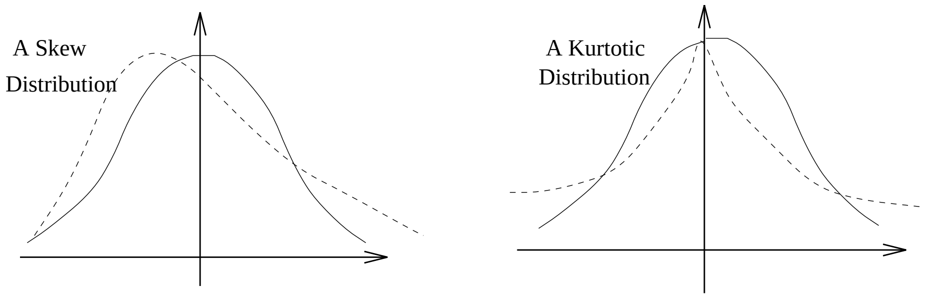


Figure 7.2: Deviations from Gaussian distributions: the dotted line on the left represents a negatively skewed distribution; that on the right represents a positively kurtotic distribution; in each case, the solid line represents a Gaussian distribution

first Principal Component is the direction with greatest spread - the long axis of the parallelogram. The second is of necessity perpendicular to that; we have no choice with two dimensional data; the second PC must be perpendicular to the first and so in a plane (with 2D data) we must draw the second PC as shown. The independent component directions however are parallel to the sides of the parallelogram. The first Independent Component gives no information about the direction of the second; they truly are independent. Each however finds the underlying causes of the distribution in that each finds the independent directions of the uniform two dimensional distribution.

We will see that there are two major methods used to solve this problem - one uses information theory while the other uses the higher order moments of the data. We have already used the first two moments of a set of data:

1. The first moment is the mean. The mean can be calculated from

$$\mu = E(X) = \int p(x)x dx \tag{7.3}$$

2. The second moment is the variance. The variance can be calculated from

$$\sigma^2 = E((X-\mu)^2) = \int p(x)(x-\mu)^2 dx \tag{7.4}$$

For a Gaussian distribution, that is all there is to know about the distribution. For other distributions, you may well be interested in higher moments:

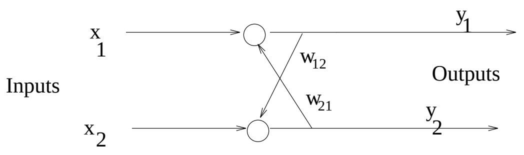


Figure 7.3: Jutten and Herault's model.

- The third moment measures skewness (see Figure 7.2) in a distribution:

$$E((X-\mu)^3) = \int p(x)(x-\mu)^3 dx \tag{7.5}$$

  If a distribution is perfectly symmetrical, this will evaluate to 0.

- The fourth moment measures the kurtosis of a distribution. This is a measure of the proportion of the distribution which is in the tails of the distribution:

$$E((X-\mu)^4) - 3 = \int p(x)(x-\mu)^4 dx - 3 \tag{7.6}$$

  The term "-3" is added to ensure that a Gaussian distribution has 0 kurtosis.

It can be shown that if two distributions are independent, then their higher moments satisfy the same constraint that we saw with the second order statistics when we decorrelate the distributions:

$$E((XY)^p) = E(X^p).E(Y^p), \forall p \tag{7.7}$$

This fact is used in some algorithms

Before we look at some algorithms which have been proposed for this problem we might ask what type of applications might benefit from being able to perform a blind separation. One area is the automatic retrieval of a single voice from a noisy environment; one reason delaying the introduction of speech communication with auto-teller machines (those nice machines which dish out bank notes) is that these machines tend to be situated in busy high streets where there are lots of people and lots of traffic and so an individual speaking into such a machine will certainly have competition. Secondly there has been an enormous increase recently in the use of the telephone for e.g. home banking. Again while most people ensure that they can use the phone in a quiet area of their home we would like to make our communication robust against interruptions from children, door bells etc.

7.2 Jutten and Herault

Jutten and Herault proposed a neural network architecture (Figure 7.3) similar to Földiák's first model. The feedforward of activation and the lateral inhibition are defined by:

$$y_i = x_i - \sum_{j=1}^{n} w_{ij} y_j$$

As before, in matrix terms, we can write

$$\begin{aligned} \mathbf{y} &= \mathbf{x} - \mathbf{W}\mathbf{y} \\ \text{And so, } \mathbf{y} &= (\mathbf{I}+\mathbf{W})^{-1}\mathbf{x} \end{aligned}$$

This looks very similar to models we discussed earlier (see Földiák's models) but we shall see that crucially Jutten and Herault introduce an non-linearity into the learning rule.

Before we look whether we can find a learning rule which will adaptively solve the problem, let us first consider if it is possible that such a network can separate two mixed sources.

7.2.1 An Example Separation

Consider the 2*2 problem. Let us have two signals $s_1(t), s_2(t)$ which are functions of time and let them be mixed using a matrix A to get the mixed signals $x_1(t), x_2(t)$. Then

$$\begin{aligned} x_1(t) &= a_{11}s_1(t) + a_{12}s_2(t) \\ x_2(t) &= a_{21}s_1(t) + a_{22}s_2(t) \end{aligned}$$

Now we have outputs from the network satisfying

$$\begin{aligned} y_1 &= x_1 - w_{21}y_2 \\ y_2 &= x_2 - w_{12}y_1 \end{aligned}$$

Substituting we get

$$\begin{aligned} y_1 &= x_1 - w_{21}(x_2 - w_{12}y_1) \\ \text{and rearranging gives } y_1 &= \frac{x_1 - w_{21}x_2}{1 - w_{21}w_{12}} \\ \text{Similarly, } y_2 &= \frac{x_2 - w_{12}x_1}{1 - w_{21}w_{12}} \end{aligned}$$

Substituting for the x_i values gives

$$\begin{aligned} y_1(t) &= \frac{(a_{11} - w_{21}a_{21})s_1 + (a_{12} - w_{21}a_{22})s_2}{1 - w_{21}w_{12}} \\ y_2(t) &= \frac{(a_{21} - w_{12}a_{11})s_1 + (a_{22} - w_{12}a_{12})s_2}{1 - w_{21}w_{12}} \end{aligned}$$

From this we can see two pairs of solutions at which the y values are a function only of a single signal s value:

- If $w_{21} = \frac{a_{11}}{a_{21}}$ and $w_{12} = \frac{a_{22}}{a_{12}}$ then we get the solution:

$$\begin{aligned} y_1(t) &= \frac{(a_{12} - w_{21}a_{22})s_2}{1 - w_{21}w_{12}} \\ &= \frac{(a_{12} - \frac{a_{11}}{a_{12}}a_{22})s_2}{1 - \frac{a_{11}}{a_{12}}\frac{a_{22}}{a_{12}}} \\ &= \frac{a_{12}(a_{12}a_{12} - a_{11}a_{22})}{a_{12}a_{12} - a_{11}a_{22}}s_2 \\ &= a_{12}s_2(t) \\ \text{Similarly, } y_2(t) &= \frac{(a_{21} - w_{12}a_{11})s_1}{1 - w_{21}w_{12}} \\ &= a_{21}s_1(t) \end{aligned}$$

- Alternatively, if $w_{21} = \frac{a_{12}}{a_{22}}$ and $w_{12} = \frac{a_{21}}{a_{11}}$ then we get the solution:

$$\begin{aligned} y_1(t) &= \frac{(a_{11} - w_{21}a_{21})s_1}{1 - w_{21}w_{12}} \\ &= a_{11}s_1(t) \\ y_2(t) &= \frac{(a_{22} - w_{12}a_{12})s_2}{1 - w_{21}w_{12}} \\ &= a_{22}s_2(t) \end{aligned}$$

In either case the network is extracting the single signals from the mixture - each output is a function of only one $s_i(t)$. However this only shows that the solution is possible; it does not give us an algorithm for finding these optimal weights. We must find a learning algorithm which will adapt the weights during learning to find these optimal weights.

7.2.2 Learning the weights

Having got a network which is theoretically capable of separating the two sources, we require to find a (neural network) algorithm which will adjust the parameters (the weights) till the separation actually happens.

The learning rule Jutten and Herault use is

$$\Delta w_{ij} = -\alpha f(y_i) g(y_j) \text{ for } i \neq j \tag{7.8}$$

which is clearly an extension of Hebbian learning.

Notice that if we use identity functions for f() and g() we are using exactly simple Hebbian learning which we know will decorrelate the outputs. Recall that when we decorrelate the outputs, $E(y_i y_j) = 0$. If we have two independent sources the expected value of all joint higher order moments will be equal to the product of individual higher order moments. i.e. $E(y_i^n y_j^m) = E(y_i^n)E(y_j^m)$ for all values of m and n.

Jutten and Herault suggest using two different odd functions in the learning rule 7.8. An odd function is one which satisfies f(-x) = -f(x) such as $f(x) = x^3$ so that e.g. f(-2) = - 8 = -f(2). Since the functions f() and g() are odd functions, their Taylor series expansion will consist solely of the odd terms e.g.

$$f(x) = \sum_{j=0}^{\infty} a_{2j+1} x^{2j+1}, \text{ and } g(x) = \sum_{j=0}^{\infty} b_{2j+1} x^{2j+1} \tag{7.9}$$

The familiar tanh() function (note its oddness) is one such and its expansion is

$$tanh(s) = s - \frac{s^3}{3} + \frac{2s^5}{15} - .. \tag{7.10}$$

Therefore the change due to the learing rule for a two output network is of the form

$$\begin{aligned} \Delta w_{ij} &= -\alpha f(y_1) g(y_2) \\ &= -\alpha \sum_j \sum_k a_j b_k y_1^{2j+1} y_2^{2k+1} \end{aligned}$$

Convergence is reached when all the moments $E(y_1^{2j+1} y_2^{2k+1}) = 0, \forall j, k$. Now statistical independence occurs when

$$E(y_1^{2j+1} y_2^{2k+1}) = E(y_1^{2j+1}).E(y_2^{2k+1}) \tag{7.11}$$

J and H state that since most audio signals have an even distribution, their odd moments are zero and hence at the above state of convergence we have the independence criterion (7.11) satisfied.

In practice the signal separation properties seem to work when separating 2 or perhaps 3 voices from a mixture but no more than that; also the process is not robust and requires careful parameter setting.

Another example of using the network was given in the image processing field: the input data was words written on card but with a sloping style. The network successfully converted the image to one in which the writing was parallel to the edges of the paper. The result is due to the fact that a sloped line introduces dependencies between the x and y coordinates. This dependency is minimised when the lines are either horizontal or vertical. In fact if the original writing is closer to the vertical than horizontal orientation, the output will be a vertical line of text.

7.3 Non-linear PCA

Oja's Subspace Algorithm was shown earlier to find the Principal Components of the input data which we know means decorrelation rather than independence of the outputs. The learning rule is repeated here for convenience:

$$\Delta w_{ij} = \alpha (x_i y_j - y_j \sum_k w_{ik} y_k) \tag{7.12}$$

Since it finds an approximation to true PCA and PCA gives us the least error in reconstruction of the data from a linear operation, the Oja network can be thought of as finding the best linearly compressed form of the data.

Karhunen and Joutsensalo have derived from equation 7.12 a non-linear equivalent:

$$\Delta w_{ij} = \alpha(x_i f(y_j) - f(y_j) \sum_k w_{ik} f(y_k)) \tag{7.13}$$

This can be derived as an approximation to the best non-linear compression of the data. While there is no 100% secure derivation of this algorithm as a solution to the ICA of a data set, it has been found experimentally that the algorithm does indeed find independent sources of some mixtures of signals (see below). Also the addition of a non-linearity breaks the symmetry which we found using the original subspace algorithm: with the original algorithm, the individual principal components were not found (indeed it is experimentally found that the algorithm tends to divide the variance up evenly between the output neurons). Therefore the original linear algorithm finds only a basis of the subspace *not* the actual principal components themselves. However this non-linear algorithm (7.13) finds the independent sources exactly not just a linear combination of the sources.

7.3.1 Simulations and Discussion

Karhunen and Joutsensalo have shown that the algorithm derived above is capable of separating signals into their significant subsignals. As an example, we repeat their experiment to separate samples of a sum of sinusoids into its component parts: the experimental data consists of N samples of a signal composed of the sum of 2 sinusoids in noise:

$$x(t) = \sum_{j=1}^{2} A_j cos(2\pi f_j t - \theta_j) + \omega_t \tag{7.14}$$

The amplitudes, A_j, frequencies, f_j and phases θ_j are unknown and must be estimated by the algorithm. We use initially white noise, $\omega_t \sim N(0, 0.05)$ where t denotes time.

Our input vector is a vector comprising a randomly drawn instance of $x(t)$ and that of the 14 subsequent times, $t+1, ..t+14$. We can show that a network whose output neurons use a non-linear function are better able to separate the input signal into its component parts while those using linear functions are less able to differentiate the individual subsignals. The original signal is shown in Figure 7.4 while the output of the non-linear output neurons are shown in Figure 7.5. This capability is not affected by coloured noise. Clearly the output neuron has identified one of independent sinusoids. But in general, this network's performance on e.g. voice data has not yielded very good results.

7.4 Information Maximisation

Bell and Sejnowski have developed a network based on the desire to maximise mutual information between inputs X and outputs Y:

$$I(X;Y) = H(Y) - H(Y|X) \tag{7.15}$$

They reason, however, that $H(Y|X)$ is independent of the weights W and so

$$\frac{\partial I(X;Y)}{\partial w} = \frac{\partial H(Y)}{\partial w} \tag{7.16}$$

Now comes the interesting part: the entropy of a distribution is maximised when all outcomes are equally likely. Therefore we wish to choose an activation function at the output neurons which equalises each neuron's chances of firing and so maximises their collective entropy. An example

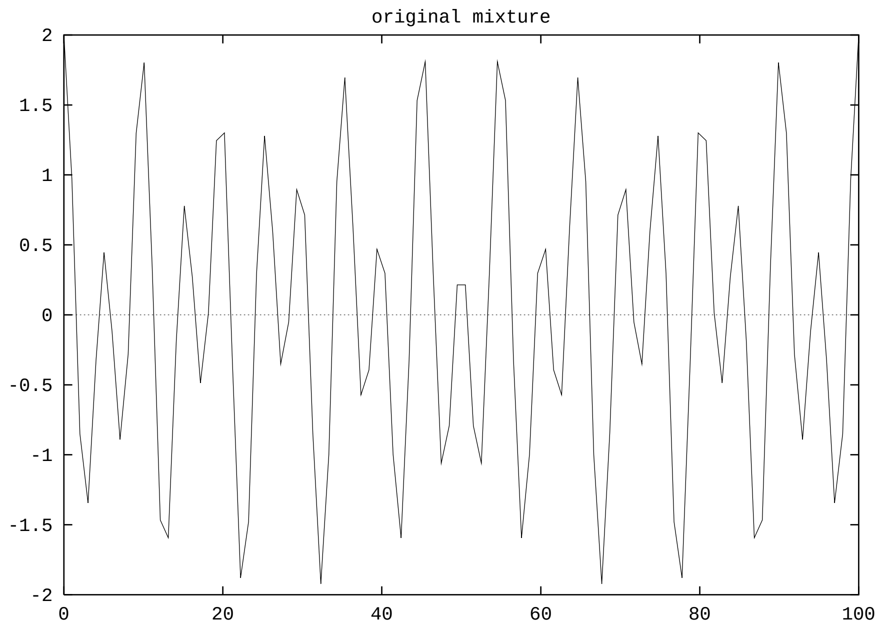


Figure 7.4: The original signal comprising a mixture of sinusoids

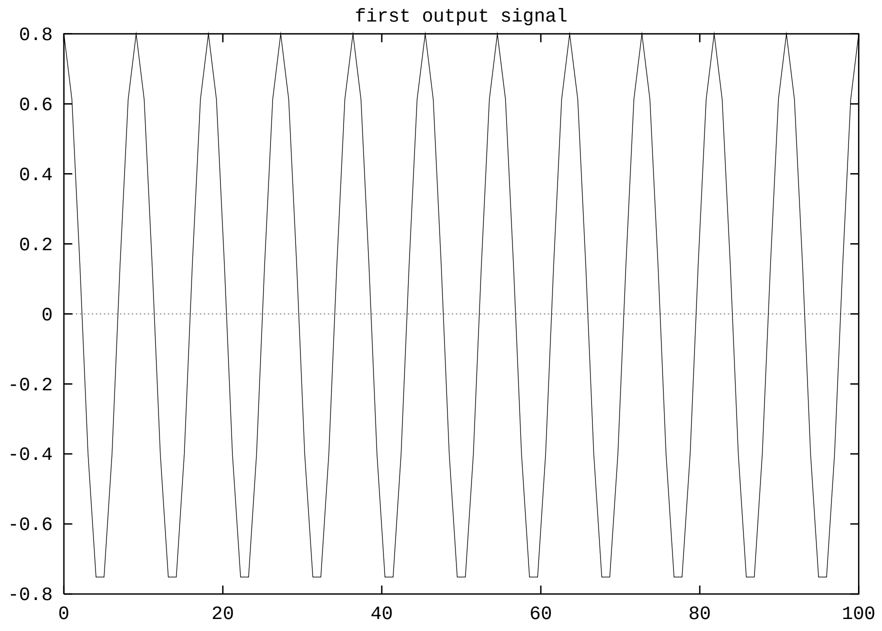


Figure 7.5: The output from the first interneuron after training when the output neuron's output is a non-linear function of its inputs

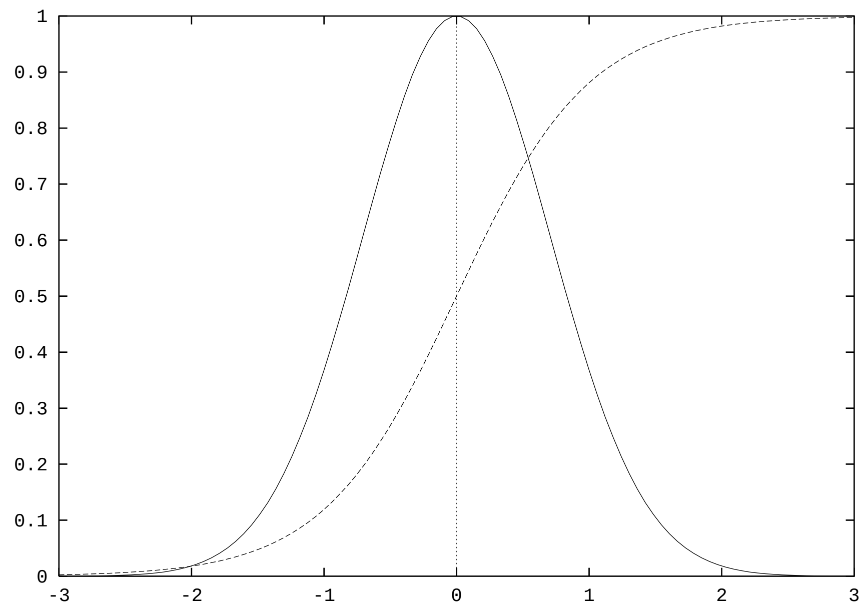


Figure 7.6: The solid line shows a Gaussian probability distribution. The sigmoid is the optimal function for evening out the output distribution so that all outputs are equally likely.

is given in Figure 7.6 in which we show a Gaussian distribution and a sigmoid. Note that at the points of the distribution at which there are maximum values the slope of the sigmoid is greatest: the high density parts of the probability density function of the inputs is matched with the highly sloping parts of the sigmoid; this evens out the distribution at the outputs.

If we consider a single output Y with a single input X joined by a single weight w, the distribution of the Y values is shown in Figure 7.7 over the weights w and inputs X where

$$y = \frac{1}{1 + \exp(-wx)} \tag{7.17}$$

For large values of w (=1) the Gaussian nature of the input distribution is clear. For negative values of w the distribution becomes a bipolar distribution with equal probability for each of the two limiting values; but there is an intermediate value of w at which the output distribution is uniform. This last is the optimal value of w for which our learning algorithms should search.

Notice that this value depends on the actual distribution of the input data which is the sort of relationship that suits neural learning since it inherently responds to the statistics of the input distribution.

7.4.1 The Learning Algorithm

We can write the probability density function of an output y as

$$f_y(y) = \frac{f_x(x)}{\frac{\partial y}{\partial x}} \tag{7.18}$$

where $f_y()$ is the probability density function of y and $f_x()$ is the probability density function of x. Then the entropy of the output is

$$H(y) = -E(\ln f_y(y)) = E(\ln|\frac{\partial y}{\partial x}|) - E(\ln f_x(x)) \tag{7.19}$$

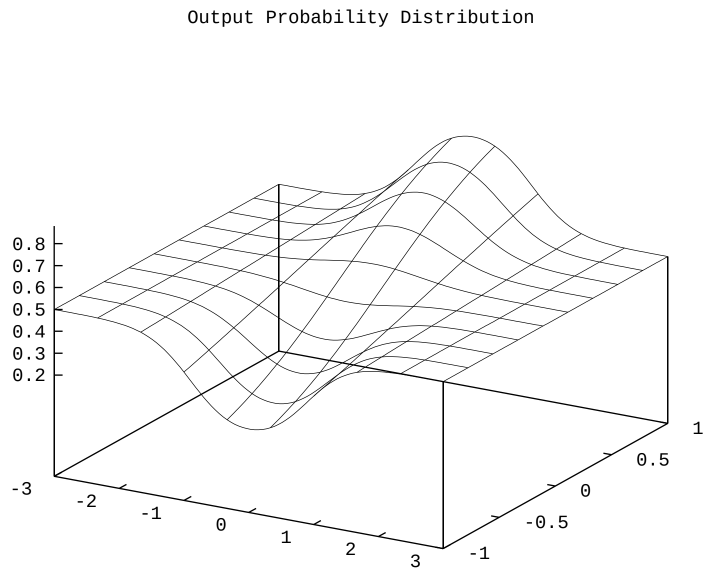


Figure 7.7: The distribution of the outputs as a function of inputs and weights. The weight which gives the most uniform distribution is about 0 in this case. Note that when the weight increases towards 1, the distribution becomes more peaked (actually Gaussian). When the weight decreases we tend to get a bimodal distribution.

Now the second term here is unaffected by any change in the weights and therefore we can concentrate on maximising the first term with respect to w:

$$\Delta w \propto \frac{\partial H}{\partial w} = \frac{\partial}{\partial w}(\ln|\frac{\partial y}{\partial x}|) = (\frac{\partial y}{\partial x})^{-1}\frac{\partial}{\partial w}\frac{\partial y}{\partial x} \tag{7.20}$$

In the special case of the logistic transfer function,

$$\begin{aligned} act &= wx + w_0 \\ y &= \frac{1}{1+e^{-act}} \end{aligned}$$

Then

$$\begin{aligned} \frac{\partial y}{\partial x} &= wy(1-y) && (7.21) \\ \frac{\partial}{\partial w}\frac{\partial y}{\partial x} &= y(1-y)(1+wx(1-2y)) && (7.22) \end{aligned}$$

Dividing (7.22) by (7.21) give the learning rule for the logistic function of

$$\Delta w \propto (\frac{\partial y}{\partial x})^{-1}\frac{\partial}{\partial w}\frac{\partial y}{\partial x} = \frac{1}{w} + x(1-2y) \tag{7.23}$$

A similar argument gives the derivation of the weight update rule for the bias term

$$\Delta w_0 \propto 1 - 2y \tag{7.24}$$

Notice the effect of these two rules:

1. at convergence, $\Delta w_0 = 0$ and so the expected output is $\frac{1}{2}$. This in effect moves the sigmoid horizontally along its graph till the centre of the sigmoid (the steepest part) is above the peak of the input distribution $f_x()$.
2. Meanwhile the first rule is shaping the output distribution:
   - the $\frac{1}{w}$ part acts like an anti-decay term and moves the weights away from one of the uninformative situations, where w is zero and y is constant regardless of the input. i.e.
     $$y = \frac{1}{1+e^{-w_0}} \tag{7.25}$$
   - the other term is anti-Hebbian. This keeps the rule from moving to the other uninformative solution which is that the y output is always 1

These forces balance out (recall the comments in the last chapter) and cause the output distribution to stabilise at the maximum entropy solution.

We do not reproduce the derivation of the many input-many output rule but merely give the results:

$$\begin{aligned} \Delta W &\propto (W^T)^{-1} + (\mathbf{1} - 2\mathbf{y})\mathbf{x}^T \\ \Delta \mathbf{w}_0 &\propto \mathbf{1} - 2\mathbf{y} \end{aligned}$$

It is possible to use very flexible sigmoids in which e.g. the top and bottom parts of the sigmoid are independently matched to the input signal's statistics. Bell and Sejnowski show that such a network can extract 10 voices from a linear mixture of voices with great precision. Their algorithm has only failed under two conditions:

- When more than one of the sources was white Gaussian noise
- When the mixing matrix was almost singular

In the first case, there is no possible algorithm which can extract a single Gaussian from a mixture of Gaussians since the mixture of Gaussians is itself a Gaussian. In the second case the problem is ill defined since we have n signals and < n independent inputs to the neural network.

It has been shown that removing the correlations (the second order statistics) from the input data greatly increases the speed of convergence of the network: Hebbian (and anti-Hebbian) learning responds mostly to the correlations in the input data; by removing these the network can concentrate on the other facets of learning. This is sometimes known as sphering the data - if we were outside the data set we would see a high dimensional sphere (rather than an ellipse) - or whitening the data - if we plot the data as a time series it would change approximately equally at all frequencies.

7.5 The Paisley Dimension

The group of methods based on Projection Pursuit is based on one central idea: rather than solving the difficult problem of identifying structure in high dimensional data, project the data onto a low dimensional subspace and look for structure in the projection. However not all projections will reveal the data's structure equally well. Therefore we define an index that measures how "interesting" a given projection is, and then represent the data in terms of the projections that maximise the index and are therefore maximally "interesting".

We will initially restrict our attention to one dimensional subspaces i.e. we will identify an index for each line in the space and attempt to maximise the index in order to make projections of the raw data onto the line as interesting as possible.

Clearly the choice of index is the crucial factor in Projection Pursuit, and the index is specified by our desire to identify interesting directions. Therefore we must define what we mean by "interesting directions".

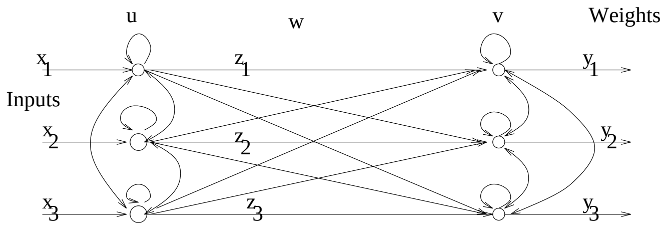


Figure 7.8: The extended exploratory projection pursuit network.
The first layer of weights, U, decorrelates the inputs; the interaction between the second layer of weights, W and the third, V eliminates statistical dependencies from the z values.

Friedman notes that what constitutes an interesting direction is more difficult to define than what constitutes an uninteresting direction. The idea of "interestingness" is usually defined in relation to the oft-quoted observation of Diaconis and Freedman that most projections of high-dimensional data onto arbitrary lines through most multi-dimensional data give almost Gaussian distributions. This would suggest that if we wish to identify "interesting" features in data, we should look for those directions, projections onto which are as non-Gaussian as possible. The negative feedback network can be used to find interesting directions by using it to identify directions which are most kurtotic or most skewed (using the fourth and third moments respectively).

Girolami and Fyfe have developed an extension of the negative feedback network described in Chapter 4.

The first layer of weights perform exactly as we saw in Foldiak's second model; the equations are

$$z_i = x_i + \sum_{j=1}^{n} u_{ij} z_j$$

with learning rule

$$\Delta u_{ij} = -\alpha(1 - z_i z_j)$$

The net result is that the outputs, the components of **z**, are decorrelated with about equal variance. Note that since if the signals (such as voice signals) are not Gaussian this does not lead to separation of the independent sources. The net result is the removal of the second order statistics from the data - the covariance matrix of the z values should be diagonal.

Now **z** is fed forward through the W weights to the output neurons where there is a second layer of lateral inhibition. However before the activation is passed through this layer it is passed back to the originating z values as inhibition and then a non-linear function of the inputs is calculated:

$$\begin{aligned} act_i &= \sum_j w_{ij} z_j \\ z_j &\leftarrow z_j - w_{ij} act_i \\ s_i &= act_i - tanh(act_i) \end{aligned}$$

Now we pass this output through the lateral inhibition to get the final output y values

$$y_i = s_i + \sum_j v_{ij} s_j \tag{7.26}$$

and then the new weights are calculated

$$\begin{aligned} \Delta w_{ij} &= \beta z_j y_i \\ \Delta v_{ij} &= \gamma y_i y_j \end{aligned}$$

The net result is the removal of any dependence from the output signals.

0.65	0.20	-0.43	0.60	0.467
-0.3	-0.49	0.7	-0.3	0.57
0.68	1.5	-0.8	0.41	1.34
-0.234	0.38	0.35	0.45	-0.76
0.85	-0.43	0.6	-0.7	0.4

Table 7.1: The Mixing Matrix used to create a babble of voices.

Male 1	0.011347
Male 2	0.001373
Female 1	0.000288
Female 2	0.000368
Female 3	0.000191

Table 7.2: Fourth Order Cumulants of the individual voices.

7.5.1 Example

One of the descriptions of this type of problem is that of the "cocktail party problem". Girolami has therefore used this network to extract a single voice from a linear mixture of voices.

Five samples of five seconds of natural speech was recorded using the standard telecom sampling rate of 8khz. Two adult male and female voices were used along with that of a female child. The speakers each spoke their name and a six digit number. The samples were then linearly mixed using the 5 x 5 mixing matrix shown in Table 7.1 which is well conditioned with a determinant value of 1.42. The fourth order statistics of the original signals are shown in Table 7.2.

The output signals played back are clear with no residual of the mixture as shown in Figure 7.9. When we look at the converged weight matrices, we see that both U and V are diagonal and symmetric as would be expected. The magnitudes of the values in U indicate the large correlations in the incoming raw data, with the off-diagonal terms being typically within an order of magnitude less than the diagonal terms. Compare this with the V weight matrix where the off-diagonal terms are all three orders of magnitude less than the diagonal terms, indicative of the whitened input to the layer of neurons.

7.6 Penalised Minimum Reconstruction Error

We conclude this chapter with a new network which is an extension of a PCA network but which has been used to find Multiple Causes; we do this to emphasise that this chapter and the previous one are really variations on a theme. Both are approaching the same problem but from different perspectives.

7.6.1 The Least Mean Square Error Network

We discussed in Chapter 5 how Xu has recently derived a neural network which we can describe as one which feeds activation forward to outputs and then back to inputs through the same weights. He envisages the returning signal from the outputs as trying to reconstruct the input vector. He aims to minimise the least mean square error at its inputs and shown that it is a PCA network. The error term is

$$\begin{aligned} J(\mathbf{W}) &= E(\parallel \mathbf{x} - \hat{\mathbf{x}} \parallel) \\ &= \int p(\mathbf{x}) \parallel \mathbf{x} - \mathbf{W}^T\mathbf{W}\mathbf{x} \parallel d\mathbf{x} \end{aligned}$$

Starting from this (which is one of the definitions of PCA), Xu derives the learning rule

$$\Delta\mathbf{W} = \mu\mathbf{x}^T\mathbf{x}(\mathbf{I} - \mathbf{W}^T\mathbf{W})\mathbf{W} + (\mathbf{x} - \mathbf{W}^T\mathbf{W}\mathbf{x})(\mathbf{W}\mathbf{x})^T \tag{7.27}$$

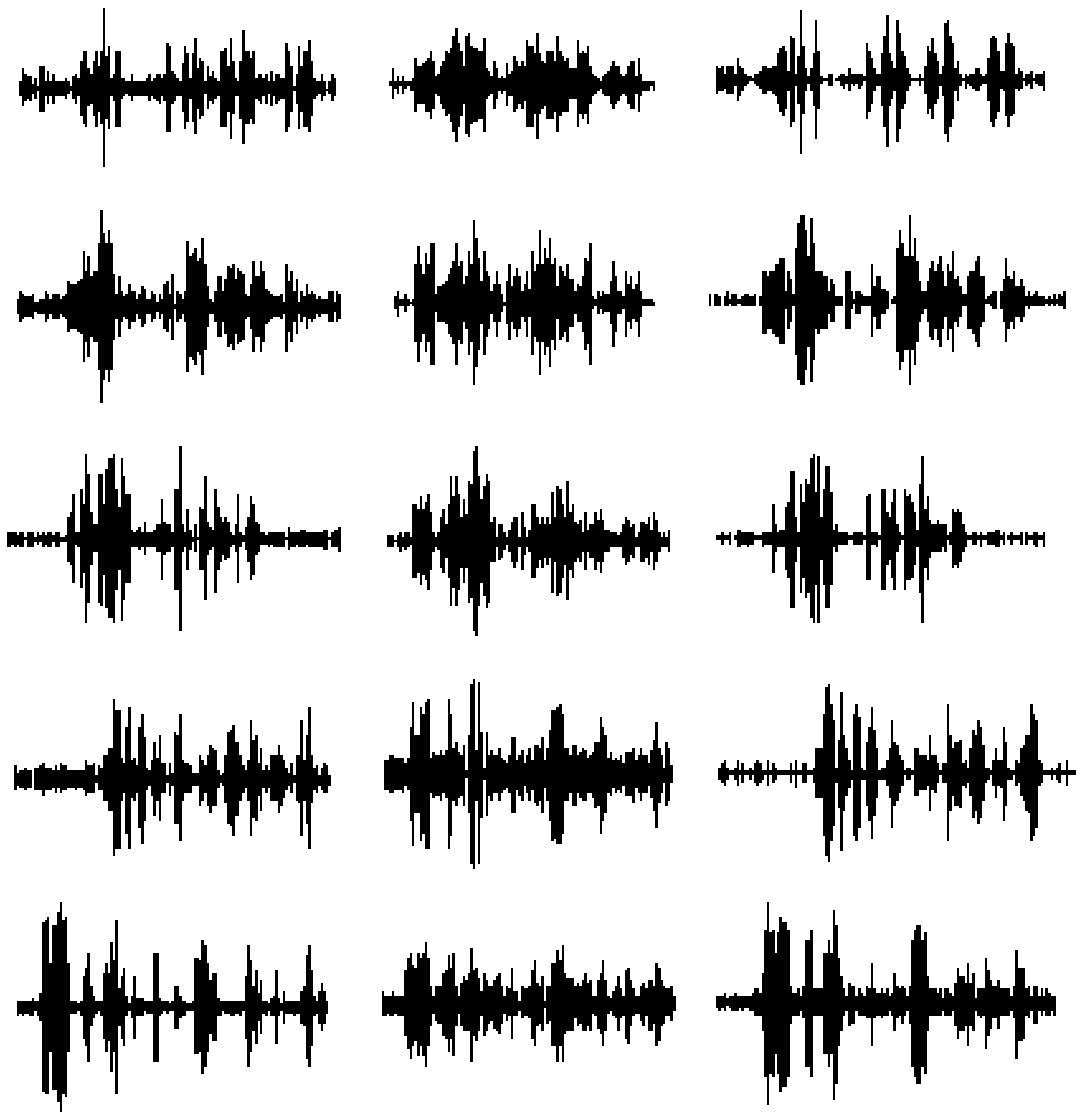


Figure 7.9: The original signals, how they were seen by the network after the mixing matrix has mixed them and the retrieved signals heard by the network.

If we put this into the feedforward and feedback terms, we may use $\mathbf{y} = \mathbf{W}\mathbf{x}$ and $\mathbf{e} = \mathbf{x} - \mathbf{W}^T\mathbf{W}\mathbf{x}$ and so state the rule as

$$\Delta\mathbf{W} = \mu(\mathbf{x}\mathbf{e}^T\mathbf{W}\mathbf{y}' + \mathbf{e}\mathbf{y}^T) \tag{7.28}$$

7.6.2 Adding Competition

Now in the context of the discussion in the last chapter, we have a network which is wholly given over to cooperation - the neurons are taking the maximum information out of the data set. We require to add some competition to the network to find the independent sources.

One way to do this is to simply add a non-linearity to the learning rule to get

$$\Delta\mathbf{W} = \mu(\mathbf{x}\mathbf{e}^T f(\mathbf{W}\mathbf{y}') + \mathbf{e}f(\mathbf{y})^T) \tag{7.29}$$

This does have some effect in finding the individual sources (this is the same method Karhunen and Joutsensalo use) but the competition is quite weak. A better solution, developed by Zhang, Xu and Fu, is to explicitly add a penalty term to the cost function to get

$$E(\mathbf{W}) = J(\mathbf{W}) + \lambda G(\mathbf{W}) \tag{7.30}$$

where J() is the best reconstruction criterion as before and G() is a competition criterion which will penalise cooperation. The Lagrange parameter λ will trade-off competition for cooperation so we get compromise between these two criteria.

The next section will consider the nature of the competition function G().

The Nature of the Competition Function

Zhang *et al* develop a competition function based on information theory. Firstly let the i^{th} activation be h_i which will then be passed through an activation function f() to get the output y_i. Then, in the absence of noise on the inputs, the mutual information between inputs and outputs satisfies

$$I(\mathbf{y}, \mathbf{x}) = I(\mathbf{y}, \mathbf{h}) \tag{7.31}$$

If the postsynaptic potentials are factorial (recall the previous Chapter),

$$p(\mathbf{h}) = \prod_{i=1}^{M} p(h_i) \tag{7.32}$$

One possible index might be to reduce the pairwise correlation over the outputs. Thus we might have

$$\begin{aligned} G &= \sum_{i=1}^{M} \sum_{j=1, j\neq i}^{M} g_{ij} \text{ where} \\ g_{ij} &= \int (y_i y_j) p(\mathbf{x}) d\mathbf{x} \end{aligned}$$

This would certainly work for Gaussian distributions but is not sufficient for non-Gaussian distributions and so we might extend this to

$$g_{ij} = \int (y_i y_j)^k p(\mathbf{x}) d\mathbf{x} \tag{7.33}$$

Then e.g. taking $k = 2$ we would have

$$\Delta\mathbf{w}_j = \mu(\mathbf{x}\mathbf{e}^T\mathbf{w}_j y_j + y_j\mathbf{e} - \lambda y_j'\mathbf{x}\sum_{l\neq j} y_i^2) \tag{7.34}$$

thereby introducing higher order statistics into the learning rule.

We can imagine a number of competition functions all of which should share the property that when the output of one neuron increases the other's output must decrease. i.e. if $\sigma_i()$ is the output function of the i^{th} neuron, then

$$\frac{\partial \sigma_i}{\partial \mathbf{y}_j} < 0, \forall j \neq i \tag{7.35}$$

7.7 Conclusion

We should view the last two chapters as two different facets of the same problem -that of identifying underlying causes from data which is a mixture of independent data from the underlying causes. The methods in the previous chapter tend to concentrate on the play-off between two neural methods - competition and cooperation. The methods in this chapter are based more on the statistical properties of independent distributions.

There is as yet no complete solution to this problem and research is continuing on networks like those discussed in these chapters. It is a fascinating area of research and one which may well bear fruit in the near future.

Chapter 8

Learning

In this concluding chapter, we shall investigate analytically issues associated with learning in artificial neural networks. Many of these issues, in fact, arose within other communities e.g. machine learning or statistical inference but are very valid for ANNs. Our concern is always the operation of our networks on the test set of data rather than the training set. We referred to this in Chapter 5 as the network's generalisation properties.

We will first investigate a particular problem with learning machines known as the "Bias-Variance Dilemma" and use the analysis to investigate whether it is possible to improve the generalisation performance of out neural networks. We will then consider two specific measure of performance in neural networks and introduce a new type of network whose generalisation performance is suspect. We will investigate ways to improve it.

8.1 The Bias-Variance Dilemma

Consider again Figure 5.3. We noted then that a network could be overtrained in that it could be trained to fit the training data exactly but would then not perform so well on the test data. We stated that one reason for this poor performance could be that the network had too many parameters (weights or neurons) for the problem and discussed one method of pruning the network. In this section, we are going to investigate the source of the problem more analytically. Much of the discussion is based on the article by Geman *et al.*

Consider the double convex problem of Chapter 5. We wish to make an inference with respect to which class any particular input point belongs. There are two main classes of methods for doing so:

1. Parametric modelling: we create a model of the input data and then must simply adjust the parameters of the model to get the minimum error on the training set. The simplest model would be to use a linear model of the form

$$y = w_1x_1 + w_2x_2 + w_3 \tag{8.1}$$

   which can be thought of a simple one layer linear neural network used to classify the inputs (x_1, x_2) according to the value of y. The LMS training algorithm could be used to find the optimal value of w_1, w_2 and w_3. Clearly this would not be a very successful model because the data is simply not linearly separable but it would have the advantage of being very quick to train. This is typical of parametric models: if you have the right model, it is certainly the most efficient method to use; if you have the wrong model, the inherent errors in the difference between the model and reality are insuperable. This type of error is known as *bias*. Notice that we can still get the best possible estimator within the constraints set by the model but that even this estimator will be subject to bias.

2. Non-parametric estimation: sometimes non-parametric estimation is known as model-free estimation since we do not, in advance, make any suppositions about the type of model. We can, during learning, change both the parameters of the current model (the weights) and change the nature of the model itself by e.g. changing the activation functions or adding new neurons or adding new layers. Backpropagation learning is usually described as non-parametric estimation since we can change not only the weights into the output layer but also the weights into and hence the activation of the hidden layer. Estimation of this type is inherently slow. Also we shall see that such models are very much at the mercy of the data set - a different data set would produce a different model and so errors on the training set do not necessarily give good estimates of the possible errors on the test set. In addition, if we have too many parameters in our current model, we are liable to the type of error discussed in Chapter 5 in which the ANN simply memorises the input data and performs no generalisation: this type of error is known as *variance*. A different set of training data would create a different final model.

The bias-variance dilemma is the result of having to play off one of these types of error against the other in any real world estimation problem. We will see that reducing the error in one will have an adverse effect on the other.

If an estimator can be shown to asymptotically converge to the object of estimation it is known as a *consistent* estimator. So if our ANN can be shown to be capable of modelling the distribution given enough training examples it is consistent. However, in practice, the number of training examples can be excessively large. In addition, when we have only a limited number of samples, a parametric estimator can outperform a non-parametric one which will be very dependent on the actual data points seen during training.

It is important to be clear that non-parametric estimation is not without parameters; it is simply that its parameters may not have a valid meaning in terms of the model's existence whereas in parametric modelling, the parameters have, in some way, a meaning for our model.

8.1.1 Decomposition of the Error

Let us consider a family of estimators which has a mean value, E(y), of outputs with which we are attempting to estimate the target value,t. Let there be a training set D $=\{(\mathbf{x}_1,t_1),...,(\mathbf{x}_N,t_N)\}$. Then the error that we wish to minimise is

$$E((t-y)^2|\mathbf{x},D) \tag{8.2}$$

where the expectation is taken over the training set and the input distribution. Now

$$\begin{aligned}
E((t-y)^2|\mathbf{x},D) &= E(((t-E(t|\mathbf{x}))+(E(t|\mathbf{x})-y))^2|\mathbf{x},D) \\
&= E((t-E(t|\mathbf{x}))^2|\mathbf{x},D)+E(E(t|\mathbf{x})-y)^2|\mathbf{x},D) \\
&\quad +2E((t-E(t|\mathbf{x})|\mathbf{x},D).(E(t|\mathbf{x},D)-y)|\mathbf{x},D)) \\
&= E((t-E(t|\mathbf{x}))^2|\mathbf{x},D)+E((E(t|\mathbf{x})-y)^2|\mathbf{x},D)
\end{aligned}$$

The first term does not depend on the data set, D or on the output y. It is simply the variance of t given **x**. So no estimator can get rid of this term - the innate variance in the data cannot be removed. So the last term, the squared distance of the estimator to the expected value of t given **x** is a natural predictor of the effectiveness of y as a predictor for t. We concentrate on this term therefore

$$\begin{aligned}
E((y-E(t|\mathbf{x}))^2) &= E(((y-E(y))+(E(y)-E(t|\mathbf{x})))^2) \\
&= E((y-E(y))^2)+E(E(y)-E(t|\mathbf{x}))^2) \\
&\quad +2E(y-E(y)).(E(y)-E(t|\mathbf{x}))) \\
&= E((y-E(y))^2)+E(E(y)-E(t|\mathbf{x}))^2)
\end{aligned}$$

The first term is known as the "variance" - the amount of squared error due to the distance which the current estimator is from the mean estimator; the second term is known as the bias - the amount by which the expected estimator y differs from the expected target for this particular input value **x**. If the bias is non-zero, we have a biased estimator. The algorithm is derived using a more general notation in Section 8.1.2 since you will meet this convention in books (Geman *et al* used it) but I will not expect you to reproduce it though the equivalence of the two derivations should be clear.

8.1.2 General Statistical Derivation

We will consider the above problem in a slightly more general form: we wish to choose a function f() of the inputs **x** which minimises the sum of squares of the *observed* errors i.e. those on the training set alone. It can be shown that this estimator is the best mean squared error predictor of y given **x**.

It can be shown that, among all possible estimators, the function f() which minimises

$$E((t-f(\mathbf{x}))^2|\mathbf{x}) \tag{8.3}$$

(where t is the target output when the input is **x**) is the best mean-squared-error estimator and recall that this was what our learning algorithm was intended to minimise.

Let there be a training set D $=\{(\mathbf{x}_1,t_1),...,(\mathbf{x}_N,t_N)\}$. Then the error that we wish to minimise is

$$E((t-y)^2|\mathbf{x},D) = E((t-f(\mathbf{x},D))^2|\mathbf{x},D) \tag{8.4}$$

where we have explicitly shown the dependence which f() has on the training set D. Now

$$\begin{aligned}
E((t-f(\mathbf{x},D))^2|\mathbf{x},D) &= E((t-E(t|\mathbf{x}))-(E(t|\mathbf{x})-f(\mathbf{x},D))^2|\mathbf{x},D)\\
&= E((t-E(t|\mathbf{x}))^2|\mathbf{x},D)+E(E(t|\mathbf{x})-f(\mathbf{x},D))^2|\mathbf{x},D)\\
&\quad +2E((t-E(t|\mathbf{x})|\mathbf{x},D).(E(t|\mathbf{x},D)-f(\mathbf{x})))\\
&= E((t-E(t|\mathbf{x}))^2|\mathbf{x},D)+E(E(t|\mathbf{x})-f(\mathbf{x},D))^2|\mathbf{x},D)
\end{aligned}$$

The first term does not depend on the data set, D or on the estimator f(). It is simply the variance of t given **x**. So the squared distance of the estimator to the expected value of t given **x** is a natural predictor of the effectiveness of f() as a predictor for t. We concentrate on this term therefore

$$\begin{aligned}
E((f(\mathbf{x};D)-E(t|\mathbf{x}))^2) &= E((f(\mathbf{x};D)-E(f(\mathbf{x};D)))+(E(f(\mathbf{x};D))-E(t|\mathbf{x}))^2)\\
&= E((f(\mathbf{x};D)-E(f(\mathbf{x};D)))^2)+E(E(f(\mathbf{x};D))-E(t|\mathbf{x}))^2)\\
&\quad +2E((f(\mathbf{x};D)-E(f(\mathbf{x};D))).(E(f(\mathbf{x};D))-E(t|\mathbf{x})))\\
&= E((f(\mathbf{x};D)-E(f(\mathbf{x};D)))^2)+E(E(f(\mathbf{x};D))-E(t|\mathbf{x}))^2)
\end{aligned}$$

Again the first term is the variance of the particular estimator; the second is the bias associated with the family of estimators.

8.1.3 An Example

From the same paper, we repeat a low dimensional example which is itself drawn from one developed by Wahba for ease of exposition: it has a one dimensional input and one dimensional output. We draw 100 data points from the function

$$g(x) = 4.26(e^{-x}-4e^{-2x}+3e^{-3x})+\eta \tag{8.5}$$

where η is a random number drawn from a zero mean Gaussian distribution of standard deviation 0.2. The input data and the underlying function (without the noise) are shown in Figure 8.1.

The data was used to train a backpropagation network with one input, one output and a varied number of hidden neurons. When we use a single hidden neuron, the network is not

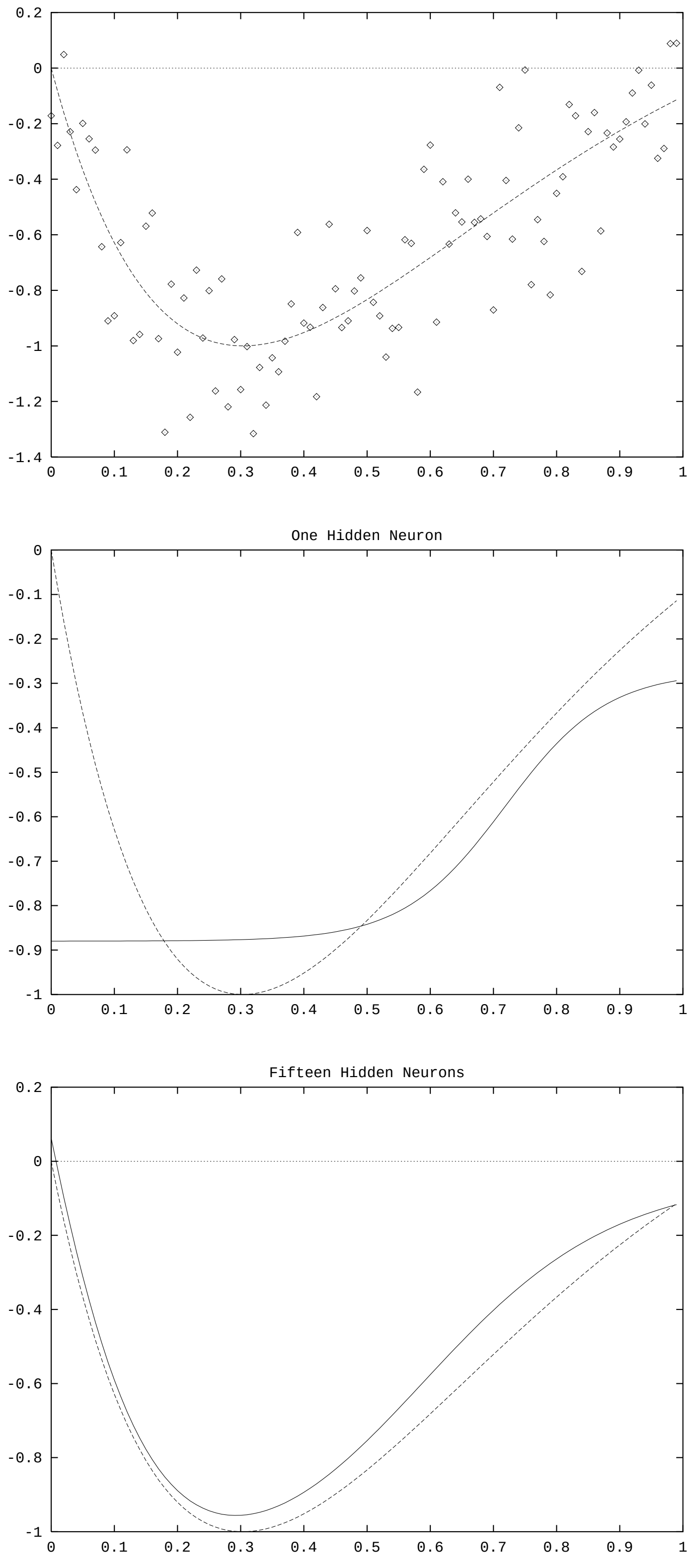


Figure 8.1: The top diagram shows the input data to the backpropagation network.
The second diagram shows the output when using a one hidden neuron network. The model is not powerful enough to model the data; most of the resulting error is due to the bias. The last diagram shows the output from a 15 hidden neuron output. The model is more powerful and more accurately models the data. In each case the noise-free underlying function is shown with a dotted line.

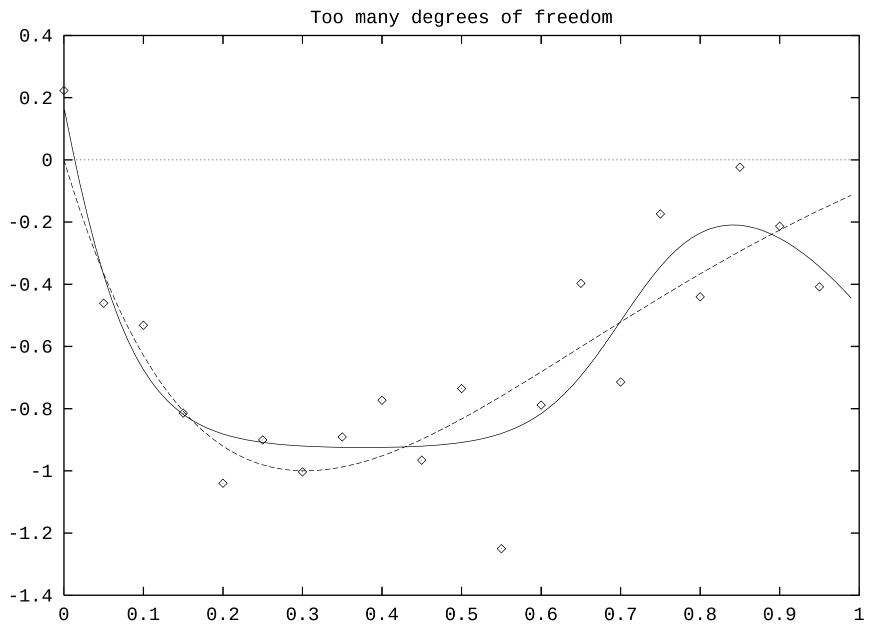


Figure 8.2: The same network as previously but trained on as subset of only 10 points. Clearly the network is too powerful for this data set and has memorised the data points rather than extracted the underlying function.

powerful enough to model the data and there is a large residual error due to the bias in the model. When we use 50 hidden neurons, the network has too many degrees of freedom (weights) and the network is able to model the actual data points and the underlying model is hidden.

Rather than choosing to show this we have shown in Figure 8.2 the results when the same 15 hidden neuron network is trained on a subset of the data points. Since it has much less training data to work on but still has the same number of degrees of freedom (weights) available to it, it can afford to memorise the data.

This would suggest correctly that one tactic which could be used to limit the damage done by both bias and variance simultaneously would be to simply increase the number of data points: as we increase the number of points, we can afford to use more complex models so reducing bias but at the same time the use of more points more heavily constrains the model and so variance is reduced. Of course this only works if we have more representative data.

Another tactic which is sometimes used to minimise the variance is to employ a regularisation tactic which smoothes the target data set. This will be discussed in a later section.

8.2 The VC Dimension

Let S denote the set of N points in the n-dimensional space of input vectors.

$$S = \{\mathbf{x}_i, i = 1, 2, ..., N\} \tag{8.6}$$

A *dichotomy* is a rule (or neural network) which splits the set by classifying some points to set A and all the rest to set B. So for a neural net, we could have

$$y = \begin{cases} 0 & \text{if } \mathbf{x} \in A \\ 1 & \text{if } \mathbf{x} \in B = S \cap A' \end{cases} \tag{8.7}$$

If we denote by $\Delta(S)$ the number of distinct dichotomies implementable by the neural network i.e. the number of distinct A and B sets into which the network can partition the set S. Now the total

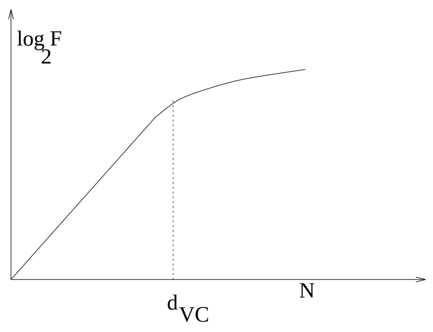


Figure 8.3: The general form of the growth function F(N). Notice that the vertical axis is a log axis so the function grows initially as 2^N till it reaches the VC dimension when it tails off to grow like a power law.

number of distinct pairs of sets is $2^{|S|}$ where $|S|$ is the number of elements in S. i.e. the number of dichotomies possible for any neural network is less than or equal to $2^{|S|}$. If a network can be trained to make the $2^{|S|}$ different dichotomies, we say the set S is shattered by the network.

Then looking at this from the point of view of the network, there is a largest size of set S which the network is capable of shattering. This largest size is one measure of the power of the network and is known as the Vapnik-Chervonenkis Dimension which, for some reason, tends to be shortened to the VC dimension.

We can think of the VC dimension as the maximum number of training examples which a network can learn to classify for all possible labellings of the classification function. It should be emphasised that the "dimension" term does not refer to a geometrical concept. Sometimes its value can be related to the number of weights in the network but often is not easy to evaluate computationally.

Now this VC dimension is all very well for telling us how many totally random pattern sets can be classified by the neural network. But what about more typical sets of patterns - such as those in which there is some structure? Also it is not clear that the VC dimension is very useful in general since we know that if a data set can be totally recalled by an ANN it is simply acting like a look-up table. Therefore we require to increase the number of patterns beyond the VC dimension (see Figure 8.3) so that generalisation can take place and it is only when we have a network which is correctly classifying a number of patterns much greater than its VC dimension that we can be hopeful that it is responding to some structure in the data and simply memorising the individual points.

It can be shown that if a network has M neurons and W weights then the VC dimension must satisfy

$$d_{VC} \leq 2W \log_2(eM) \tag{8.8}$$

where e is the base of natural logarithms (≈ 2). From this it can be calculated that the minimum number of patterns must satisfy

$$N_{Min} \approx \frac{W}{\epsilon} \tag{8.9}$$

where ϵ is a small number representing twice the number of incorrectly classified examples on the training set. E.g. to be confident in the generalisation properties of the network when we have 95% corrrectness on our training set we require about ten times as many training examples as weights in the network.

We will state without proving the main result with respect to the VC Dimension: if the VC Dimension of a set of functions is finite, each function in the set can be learned.

8.3 PAC Learning

One investigation of learning and generalisation is based on the Probably Approximately Correct model developed by Valiant in the 1980s. This discussion is based on that of Sonntag.

Suppose we have a set of data $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)\}$ which are drawn independently and at random from an unknown probability distribution. There is a fixed *but unknown* function f such that $y_i = f(\mathbf{x}_i)$ and f() belongs to some known class (or assumed known class) of functions,F. If the class of functions which are used for learning, i.e. the assumptions in the learning rule, do not agree with the underlying reality of the data set, there will be a "bias" built in to the learning system.

Consider the case of a binary classification problem so that $y_i \in \{0, 1\}$ and let the current estimate of f be $\hat{f}$. Then if we present a new $\mathbf{x}$ to our current learning machine/neural network, we can define the *error* of the network to be the probability that it misclassifies $\mathbf{x}$. i.e.

$$Error_{\hat{f}} = P(\hat{f}(\mathbf{x}) \neq y) \tag{8.10}$$

where y is the correct response to input $\mathbf{x}$ i.e. y = f($\mathbf{x}$). We are assuming that $\mathbf{x}$ is selected from the distribution according to the same procedures for selecting the rest of the training examples.

Now on what does this probability depend?

1. Since we must of necessity select only a limited number of training examples, in general this will not be enough to differentiate between all possible functions f().

2. Also there is the possibility that the set of training examples might not be representative of the set of test examples.

These both contribute to the variance element in the learning process. However, if we have enough samples, the first error is very unlikely and becomes increasingly so as we increase the number of samples. It is still possible but with increasingly low probability. For the second type of error, we must ensure that the method used to draw the test and training examples are identical. If we do this then the prediction error on new test samples will be small. So the learned function $\hat{f}$ will Probably be Approximately Correct.

If we wish to show that every member of the set F can be learned we can write that for all $\epsilon > 0$, there exists a $\delta > 0$ such that for every $f \in F$,

$$P(Error_{\hat{f}} > \epsilon) < \delta \tag{8.11}$$

i.e. we can choose to limit the probability of misclassification by puting a confidence limit on our probability of misclassification.

8.3.1 Examples

Again following Sonntag, consider a very simple example where we have a one dimensional data set, $x \in [0, 1]$ i.e. we draw the input x randomly from the unit interval. We wish to learn the function

$$f(x) = \begin{cases} 0 & \text{when } x < a \\ 1 & \text{when } x > a \end{cases} \tag{8.12}$$

where $a \in [0, 1]$. Identifying this simple function means identifying the cut-off point a. If we take a large enough sample, there will be enough pairs of examples (x_i, y_i) with x_i near a so that a good estimate of a can be obtained e.g. by choosing our estimate as the midpoint of the interval $[a_1, a_2]$ where a_1 is the largest x_i with corresponding $y_i = 0$ and a_2 is the smallest x_j with corresponding $y_j = 1$. It could be that there is an error due to bad sampling but this error decreases as the number of samples increases. It could be that there is an error because our new x lies within the interval $[a_1, a_2]$ but, again, as we take more and more samples, the expected length of this interval decreases thus decreasing the probability of choosing a test value in this set.

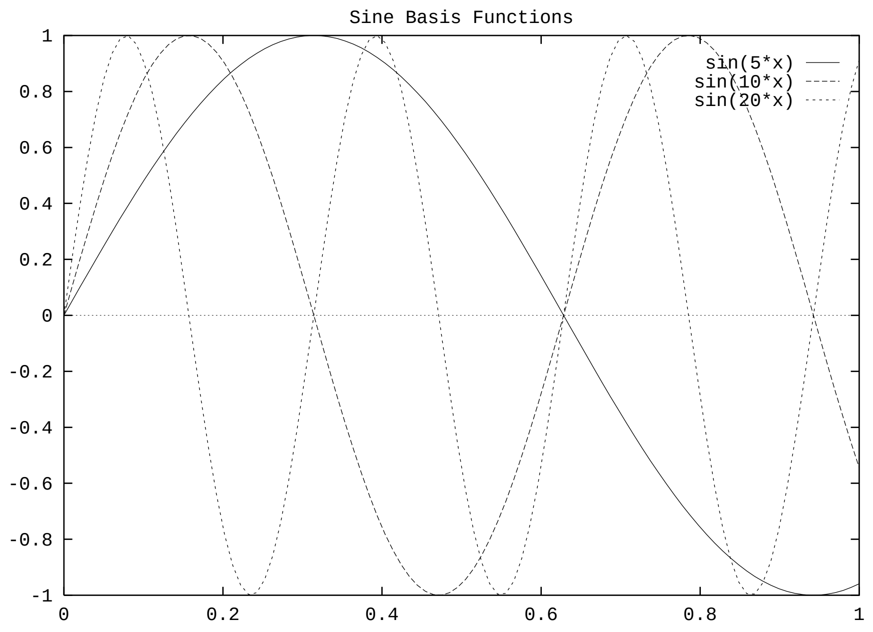


Figure 8.4: Three of the sine basis functions.

However consider the situation where of set of functions F is known to consist of the set of all functions $f_k(x) = sin(kx)$ over the same interval. So our classification procedure is known to satisfy

$$f(x) = \begin{cases} 0 & \text{when } f_k(x) \le 0 \\ 1 & \text{when } f_k(x) > 0 \end{cases} \tag{8.13}$$

Now given any sequence (x_i, y_i) we cannot be confident in our current prediction since if our current prediction is $f = f_i$ there exist an infinite number of possible alternatives to f_i. This is illustrated in Figure 8.4. We can make a prediction but not with any degree of probability.

8.4 Regularisation

We stated that one tactic which might be used is to smooth the response of the network in the region of any data points, thus getting rid of any tendency to model the noise in the data. This tactic has a long history in statistical analyis and so we will first review regularisation in a statistical context.

8.4.1 Regression

As stated previously, regression comprises finding the best estimate of a dependent variable, y, given a vector of predictor variables, $\mathbf{x}$. With parametric regression, we must make some assumptions about the form of the predictor surface e.g. that the surface is linear or quadratic, smooth or disjoint etc.. The accuracy of the results achieved will test the validity of our assumptions.

This can be more formally stated as: let (X,Y) be a pair of random variables such that $X \in R^n, Y \in R$. Regression aims to estimate the response surface,

$$f(x) = E((Y|X = x)) \tag{8.14}$$

from a set of p observations, $\mathbf{x}_i, y_i, i = 1, ..., p$.

The usual method of forming the optimal surface is the Least (Sum of) Squares Method which minimises the Euclidean distance between the actual value of y and the estimate of y based on the current input vector, **x**. Formally, if we have a function, f, which is an estimator of the predictor surface, and an input vector, **x**, then our best estimator of y is given by minimising

$$E = \min_f \sum_i^N (y_i - f(\mathbf{x}_i))^2 \tag{8.15}$$

i.e. the aim of the regression process is to find that function f which most closely matches y with the estimate of y based on using f() on the predictor, **x**, for all values (y,**x**). However, this leaves the possibility that the function may be merely approximated accurately at the N values of **x** and y. It may be far from a good approximator at other points. Intuitively, it would seem to be a good idea to force the function to smoothly interpolate between the points. Therefore to enforce a smoothness constraint on the solution, a penalty for roughness is often imposed: e.g.

$$E_R = \min_f \sum_i (y_i - f(\mathbf{x}_i))^2 + \lambda \int (f''(t))^2 dt \tag{8.16}$$

The intuitive reason behind the use of this smoothing criterion is that the second derivative of a function tells us how much the function jumps about. λ is the parameter which trades off the accuracy of the solution at the N points with the smoothness. This is sometimes known as the penalized least squares fitting algorithm.

The general form is

$$E_R = E + \lambda\Omega \tag{8.17}$$

where Ω is the penalty term. This is only one penalty from a family of penalty functions known as regularisers.

8.4.2 Weight Decay

If we choose

$$\Omega = \frac{1}{2}\sum_i w_i^2 \tag{8.18}$$

then we are penalising large weights. When a weight has a large absolute value, it will contribute greatly to the function, E_R which we are trying to minimise. It is an empirical finding that such a regularising term can add greatly to a network's performance in terms of generalisation.

An intuitively appealing reason for this is that small weights will keep the net inputs to a neuron small and so the activation function will tend to be used in its central region where it is closer to a linear activation function than at the extreme values. But we know that to model all noisy points requires sharp changes of direction which in turn demands the non-linearity found at these extreme values and so our penalty term is smoothing out the demands of different points by keeping to a linear(ish) interpolation between these points.

More analytically, consider what happens to the error term when we change w:

$$\frac{\partial E_R}{\partial w} = \frac{\partial E}{\partial w} + \lambda\frac{\partial \Omega}{\partial w} \tag{8.19}$$

Considering only the effect of the regularising term on a single weight we have

$$\frac{dw_i}{dt} = -\eta\lambda\frac{\partial \Omega}{\partial w_i} = -2\eta\lambda w_i \tag{8.20}$$

when $\Omega = \sum_i w_i^2$. Clearly the weights will only stop changing when $w_i = 0$. The solution of this differential equation is actually

$$w_i(t) = w_i(0)e^{-\eta\lambda t} \tag{8.21}$$

showing that the decay to zero is exponential and depends on the initial value of the weight, $w_i(0)$, the learning rate and the relative importance which is given to the penalty term - this being determined by the λ parameter.

So $\frac{dw}{dt} \propto \frac{\partial E_R}{\partial w}$ will be a compromise between the two factors.

8.4.3 Eigenvector Analysis of Weight Decay

This section owes much to Bishop's book(p257). We will consider the situation without the regularising term and then with such a term.

Without weight decay

Consider the truncated Taylor series expansion of E($\mathbf{w}$) in the neighbourhood of some particular value of the weights e.g. $\hat{\mathbf{w}}$.

$$E(\mathbf{w}) = E(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})\mathbf{b} + \frac{1}{2}(\mathbf{w} - \hat{\mathbf{w}})^T \mathbf{H}(\mathbf{w} - \hat{\mathbf{w}}) \tag{8.22}$$

where $\mathbf{b}$ is vector of first derivatives of E with respect to $\mathbf{w}$ at the point $\hat{\mathbf{w}}$.

$$\mathbf{b} = \frac{\partial E}{\partial \mathbf{w}}|_{\hat{\mathbf{w}}} \tag{8.23}$$

and $\mathbf{H}$ is the Hessian matrix of double derivatives whose (i,j)th element is

$$\mathbf{H}_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}|_{\hat{\mathbf{w}}} \tag{8.24}$$

For points close to $\hat{\mathbf{w}}$, the truncation of the Taylor series is accurate since $(\mathbf{w} - \hat{\mathbf{w}})$ is very small and so higher power terms containing $(\mathbf{w} - \hat{\mathbf{w}})^k$ for $k > 2$ can be ignored.

Now we can find the local minimum of the function in the neighbourhood of $\hat{\mathbf{w}}$ by calculating the derivative of this function. Noting that the first term is constant, we get

$$\frac{\partial E}{\partial \mathbf{w}} = \mathbf{b} + \mathbf{H}(\mathbf{w} - \hat{\mathbf{w}}) \tag{8.25}$$

Now if there is a minimum $\mathbf{w}^*$ in this neighbourhood, then $\mathbf{b}$=0 since there is no rate of change of the function at the minimum. Let $\hat{\mathbf{w}}^*$ be the current estimate of this optimum $\mathbf{w}^*$ and so the expansion can be written

$$E(\mathbf{w}^*) = E(\hat{\mathbf{w}}^*) + \frac{1}{2}(\mathbf{w}^* - \hat{\mathbf{w}})^T \mathbf{H}(\mathbf{w}^* - \hat{\mathbf{w}}) \tag{8.26}$$

where the Hessian must be evaluated at $\mathbf{w}^*$.

Let us now consider the eigenvalues-eigenvectors of H i.e.

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{8.27}$$

Since the eigenvectors form an orthonormal basis of the space, we can now express the term $(\mathbf{w} - \mathbf{w}^*)$ in this basis as

$$(\mathbf{w} - \mathbf{w}^*) = \sum_i \alpha_i \mathbf{u}_i \tag{8.28}$$

for some scalars α_i. Substituting this into (8.26),

$$\begin{aligned} E(\mathbf{w}^*) &= E(\hat{\mathbf{w}}^*) + \frac{1}{2}(\sum_i \alpha_i \mathbf{u}_i)\mathbf{H}(\sum_i \alpha_i \mathbf{u}_i) \\ &= E(\hat{\mathbf{w}}^*) + \frac{1}{2}(\sum_i \alpha_i \mathbf{u}_i)(\sum_i \lambda_i \alpha_i \mathbf{u}_i) \\ &= E(\hat{\mathbf{w}}^*) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2 \end{aligned}$$

since the $\mathbf{u}_i$ are orthonormal. So in general the weight change rule has most effect in those directions with greatest spread and in which the difference between the current weight and the optimal is greatest.

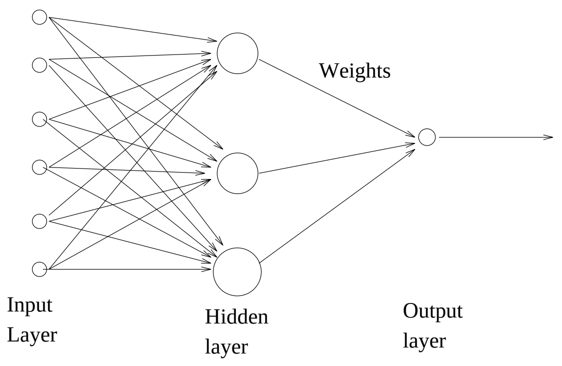


Figure 8.5: A typical radial basis function network. Activation is fed forward from the input layer to the hidden layer where a (basis) function of the Euclidean distance between the inputs and the centres of the basis functions is calculated. The weighted sum of the hidden neuron's activations is calculated at the single output neuron

Using Regularisation

Let us now consider the effect of the regularisation term. Let the minimum of the complete function,E_R, be moved to a point $\tilde{\mathbf{w}}$ by the use of the regularisation term in the weight change rule. Now the minimum is achieved when

$$\mathbf{b} + \mathbf{H}(\tilde{\mathbf{w}} - \mathbf{w}) + \lambda(\tilde{\mathbf{w}} - \mathbf{w}) = 0 \tag{8.29}$$

Now we expand $\mathbf{w}^*$ and $\tilde{\mathbf{w}}$ in the eigenvector basis to get

$$\begin{aligned} \mathbf{w} - \mathbf{w}^* &= \sum_i \alpha_i \mathbf{u}_i \text{ as before, and} \\ \mathbf{w} - \tilde{\mathbf{w}} &= \sum_i \beta_i \mathbf{u}_i \end{aligned}$$

which we can substitute into equation (8.29) to get

$$\beta_j = \frac{\lambda_j}{\lambda_j + \lambda} \alpha_j \tag{8.30}$$

So that along directions where the individual eigenvalues are much greater than the decay parameter, $\lambda_j >> \lambda$, the minimum of the error function is little changed. On the other hand, in directions where the decay parameter is relatively large compared with the eigenvalue,$\lambda >> \lambda_j$, the corresponding components of change are virtually suppressed and so any change will take place only in the former directions. So if the noise in the data really has small variance (small λ), the training algorithm will ignore that noise.

8.5 Radial Basis Functions

We noted in Chapter 5 that, while the multilayer perceptron is capable of approximating any continuous function, it can suffer from excessively long training times. In this section we will investigate a different type of ANN which can substantially shorten the time necessary for supervised learning.

A typical radial basis function (RBF) network is shown in Figure 8.5. The input layer is simply a receptor for the input data. The crucial feature of the RBF network is the function calculation which is performed in the hidden layer. This function performs a *non-linear* transformation from the input space to the hidden-layer space. The hidden neurons' functions form a basis for the input

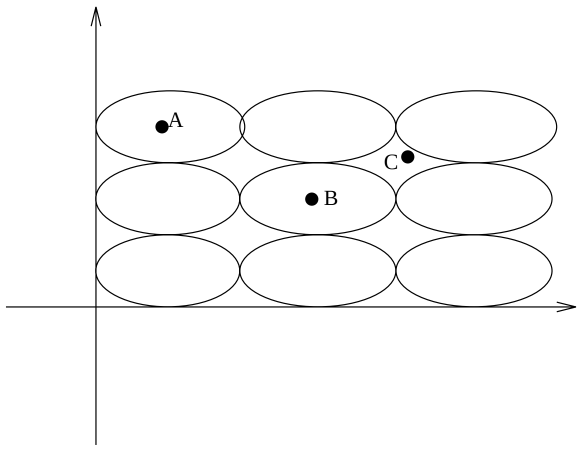


Figure 8.6: A "tiling" of a part of the plane

vectors and the output neurons merely calculate a linear (weighted) combination of the hidden neurons' outputs.

An often-used set of basis functions is the set of Gaussian functions whose mean and standard deviation may be determined in some way by the input data (see below). Therefore, if $\phi(\mathbf{x})$ is the vector of hidden neurons' outputs when the input pattern $\mathbf{x}$ is presented and if there are M hidden neurons, then

$$\begin{aligned} \phi(\mathbf{x}) &= (\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), ..., \phi_M(\mathbf{x}))^T \\ \text{where } \phi_i(\mathbf{x}) &= \exp\left(-\lambda_i \parallel \mathbf{x} - \mathbf{c}_i \parallel^2\right) \end{aligned}$$

where the centres $\mathbf{c}_i$ of the Gaussians will be determined by the input data. Note that the terms $\parallel \mathbf{x} - \mathbf{c}_i \parallel$ represent the Euclidean distance between the inputs and the i^{th} centre. For the moment we will only consider basis functions with $\lambda_i = 0$. The output of the network is calculated by

$$y = \mathbf{w}.\phi(\mathbf{x}) = \mathbf{w}^T\phi(x) \tag{8.31}$$

where $\mathbf{w}$ is the weight vector from the hidden neurons to the output neuron.

To get some idea of the effect of basis functions consider Figure 8.6. In this figure we have used an eliptical tiling of a portion of the plane; this could be thought of as a Gaussian tiling as defined above but with a different standard deviation in the vertical direction from that in the horizontal direction. We may then view the lines drawn as the 1 (or 2 or 3 ...) standard deviation contour. Then each basis function is centred as shown but each has a non-zero effect elsewhere. Thus we may think of
A as the point (1,0,0,0,0,0,0,0,0)
and B as (0,0,0,0,1,0,0,0,0)
Since the basis functions actually have non-zero values everywhere this is an approximation since A will have some effect particularly on the second, fourth and fifth basis functions (the next three closest) but these values will be relatively small compared to 1, the value of the first basis function.

However the value of the basis functions marked 2,3,5 and 6 at the point C will be non-negligible. Thus the coordinates of C in this basis might be thought of as (0,0.2,0.5,0,0.3,0.4,0,0,0) i.e. it is non-zero over 4 dimensions.

Notice also from this simple example that we have increased the dimensionality of each point by using this basis.

8.5.1 RBF and MLP as Approximators

We examine the approximation properties of both a multi-layered perceptron and a radial basis function on data drawn from a noisy version of a simple trigonometric function. Consider the

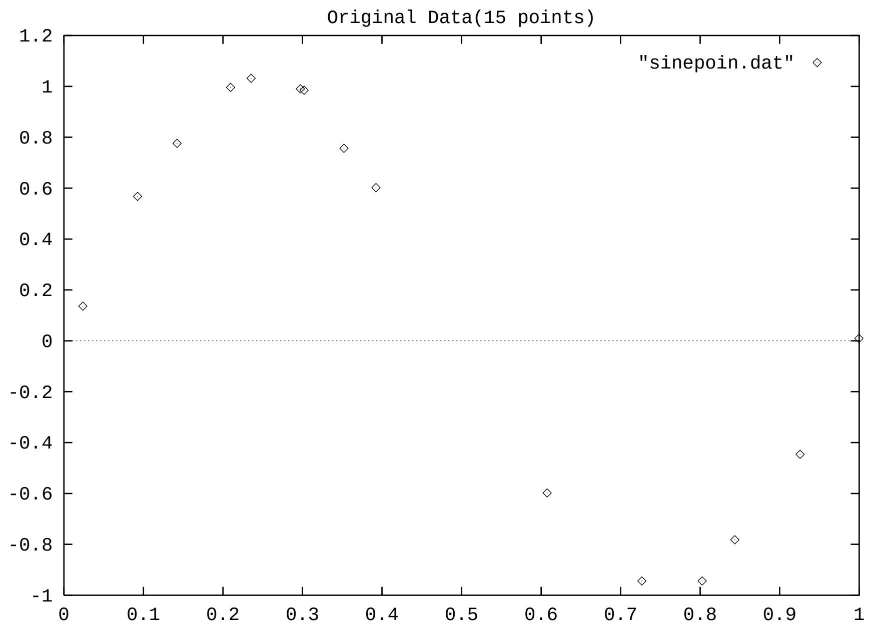


Figure 8.7: 15 data points drawn from a noisy version of $\sin(2\pi x)$.

set of points shown in Figure 8.7 which are drawn from $\sin(2\pi x)$ + noise. The convergence of radial basis function networks is shown in Figure 8.8. In all cases the centres of the basis functions were set evenly across the interval [0,1]. It is clear that the network with 1 basis function is not powerful enough to give a good approximation to the data; we recognise from the previous discussion that the error is due to bias introduced because our parametric model is not sufficiently accurate. That with 3 basis functions makes a much better job while that with 5 is better yet. Note that in thelast cases the approximation near the end points (0 and 1) is much worse than that in the centre of the mapping. This illustrates the fact that RBFs are better at interpolation than extrapolation: where there is a region of the input space with little data, an RBF cannot be expected to approximate well.

The above results might suggest that we should simply create a large RBF network with a great many basis functions, however if we create too many basis functions the network will begin to model the noise rather than try to extract the underlying signal from the data. An example is shown in Figure 8.9. In order to compare the convergence of the RBF network with an MLP we repeat the experiment with the same data but with a multi-layered perceptron with linear output units and a tanh() nonlinearity in the hidden units. The results are shown in Figure 8.10.

Notice that in this case we were *required* to have biases on both the hidden neurons and the output neurons and so the nets in the Figure had 1, 3 and 5 hidden neurons *plus* a bias neuron in each case. This is necessary because

- In an RBF network, activation contours (where the hidden neurons fire equally are circular (or ellipsoid if the function has a different response in each direction).
- In an MLP network, activation contours are planar - the hidden neurons have equal responses to a plane of input activations which must go through the origin if there is no bias.

However the number of basis neurons is not the only parameter in the RBF. We can also change its properties when we move the centres or change the width of the basis functions. We illustrate this last in Figure 8.11 in which we illustrate this fact on the same type of data as before but use a

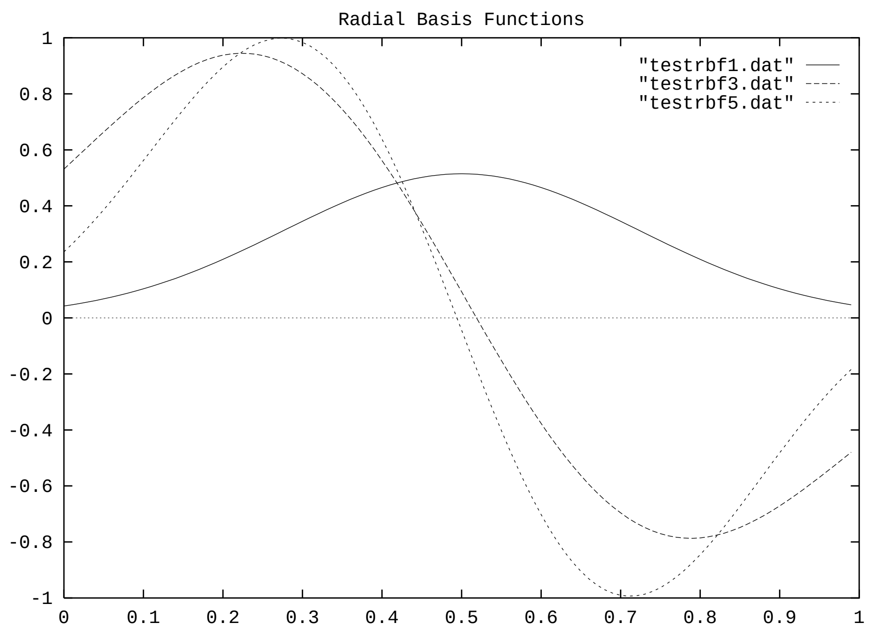


Figure 8.8: Approximation of the above data using radial basis function networks with 1, 3 and 5 basis functions.

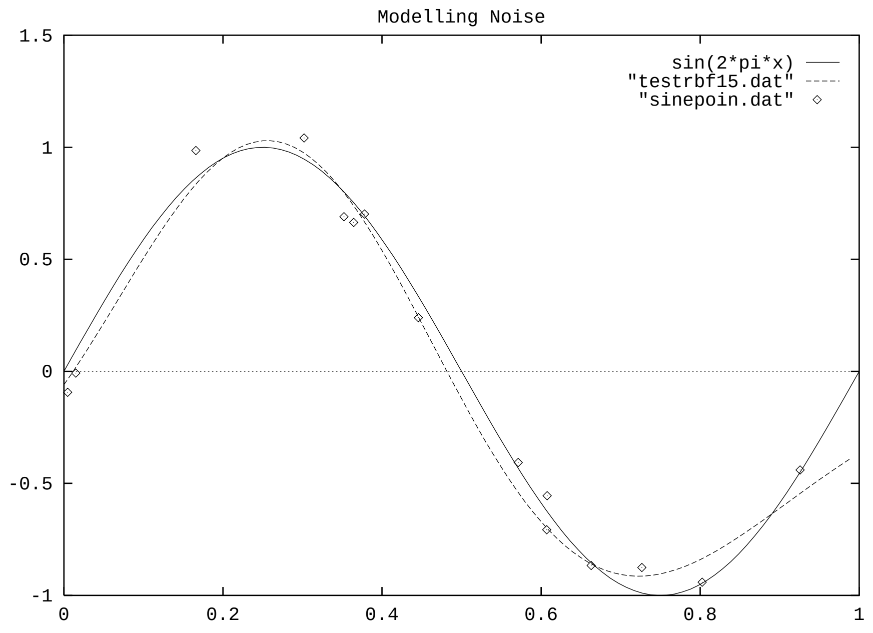


Figure 8.9: The data points which were corrupted by noise, the underlying signal and the network approximation by a radial basis function net with 15 basis functions.

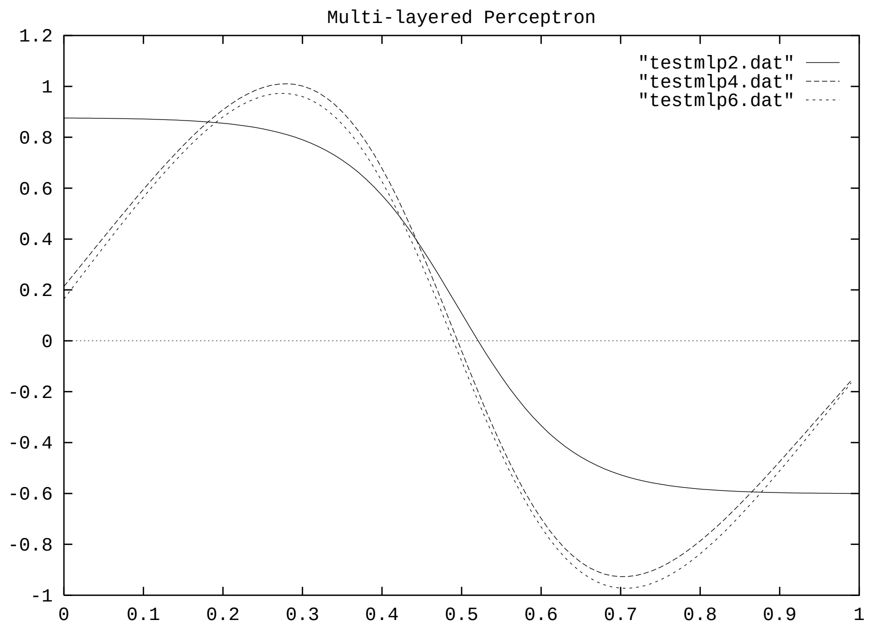


Figure 8.10: A comparison of the network convergence using multilayered perceptrons on the same data.

value of $\lambda = 1$ and $\lambda = 100$ for the parameter λ when calculating the output of the basis neurons.

$$y = exp(-\lambda \parallel \mathbf{x}_i - \mathbf{c}_i \parallel^2) \tag{8.32}$$

We see clearly that by introducing these extra possible parameters which can be changed in our model we have introduced another possible complication to the bias-variance dilemma. A small value of λ smears out the response of individual neurons over the whole data set but in doing so introduces bias into the model; a large value of λ means that each neuron's response is very localised but allows the network to model the noise thus introducing variance. This λ can also be thought of as a type of regularisation parameter.

We will, in section 8.6, investigate learning in radial basis functions with respect to the various types of parameters.

8.5.2 Comparison with MLPs

Both RBFs and MLPs can be shown to be universal approximators i.e. each can arbitrarily closely model continuous functions. There are however several important differences:

1. The neurons of an MLP generally all calculate the same function of the neurons' activations e.g. all neurons calculate the logistic function of their weighted inputs. In an RBF, the hidden neurons perform a non-linear mapping whereas the output layer is always linear.

2. The non-linearity in MLPs is generally monotonic; in RBFs we use a radially decreasing function.

3. The argument of the MLP neuron's function is the vector product $\mathbf{w}.\mathbf{x}$ of the input and the weights; in an RBF network, the argument is the distance between the input and the centre of the radial basis function, $\parallel \mathbf{x} - \mathbf{w} \parallel$.

4. MLPs perform a global calculation whereas RBFs find a sum of local outputs. Therefore MLPs are better at finding answers in regions of the input space where there is little data

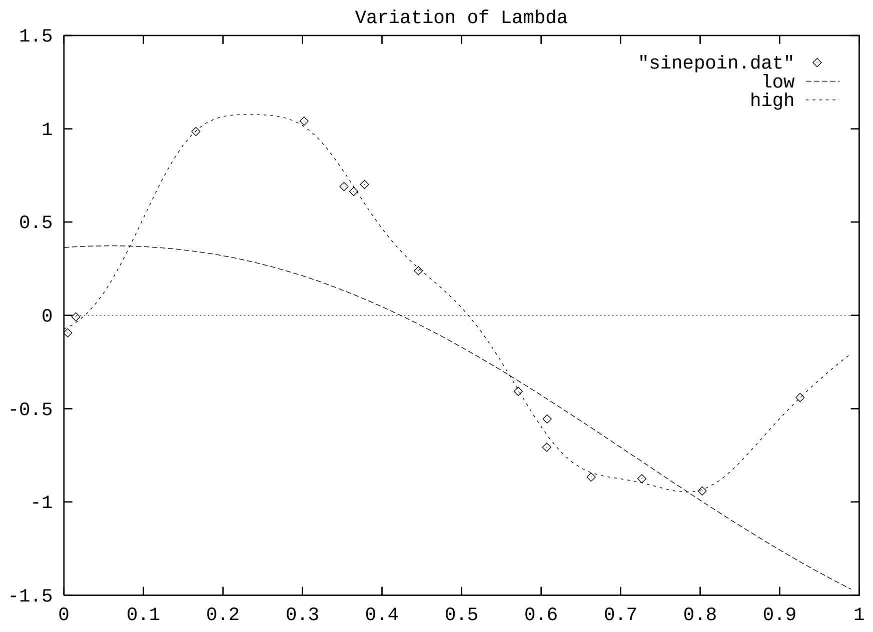


Figure 8.11: Using a basis function with a wide neighbourhood is equivalent to smoothing the output. A narrower neighbourhood function will more closely model the noise.

in the training set. If accurate results are required over the whole training space, we may require many RBFs i.e. many hidden neurons in an RBF network. However because of the local nature of the model, RBFs are less sensitive to the order in which data is presented to them.

5. MLPs must pass the error back in order to change weights progressively. RBFs do not do this and so are much quicker to train.

8.6 Learning in Radial Basis Functions

We have said that radial basis functions can be used as universal discriminators. However we still have to find the actual parameters which determine the slope of the discrimination line. These are the weights between the basis functions (in the hidden layer) and the output layer, the position of the centers and the spread of the hidden neurons. We will largely follow Haykin's analysis in this.

We first note that these changes can be viewed as happening on different timescales. Also the hidden layer's activation functions will be learned using a non-linear optimisation strategy while the output layer's weights can be adjusted through a fast linear optimisation.

8.6.1 Fixed Centers selected at Random

Let us first choose the centres of the basis functions randomly to instances of input data drawn from the data set and set the spread of all basis functions to be equal and constant. If we assume that the drawing of data points from the data set is not biased, and we have no other information, this is the best strategy. We choose an isotropic Gaussian whose standard deviation depends on the spread of the centres: let the current centre of the i^{th} basis function be $\mathbf{c}_i$

$$\phi_i(\mathbf{x}) = G(\parallel \mathbf{x} - \mathbf{c}_i \parallel) = \exp(-\frac{M}{d^2} \parallel \mathbf{x} - \mathbf{c}_i \parallel^2) \tag{8.33}$$

where M is the number of basis functions and d the maximum distance between the chosen centres. So we are fixing the standard deviation of the Gaussian as

$$\sigma = \frac{d}{\sqrt{2M}} \tag{8.34}$$

This type of network is known as a linear parametric model since we are only interested in changing the value of the weights.

The LMS Algorithm

We may train the network now using the simple LMS algorithm (see Chapter 5) in the usual way. If the sum of the errors over all N patterns is

$$E = \frac{1}{2}\sum_{i=1}^{N}(t^i - o^i)^2 = \frac{1}{2}\sum_{i=1}^{N}(t^i - \sum_j w_j\phi_j(\mathbf{x}^i))^2 = \frac{1}{2}\sum_{i=1}^{N} e_i^2 \tag{8.35}$$

where, as before, t^i represents the target output for the i^{th} input pattern, then we can represent the instantaneous error (the error on presentation of a single pattern) by

$$E^i = \frac{1}{2}(t^i - \sum_j w_j\phi_j(\mathbf{x}^i))^2 = \frac{1}{2}e_i^2 \tag{8.36}$$

and so we can create an on-line learning algorithm using

$$\frac{\partial E^i}{\partial w_k} = -(t^i - \sum_j w_j\phi_j(\mathbf{x}^i))\phi_k(\mathbf{x}^i) = -e^i.\phi_k(\mathbf{x}^i) \tag{8.37}$$

Therefore we will change the weights after presentation of the i^{th} input pattern $\mathbf{x}^i$ by

$$\Delta w_k = -\frac{\partial E^i}{\partial w_k} = e^i.\phi_k(\mathbf{x}^i) \tag{8.38}$$

Using Pseudo-Inverses

Because we only need to learn one set of parameters in this linear part of the network we can use a pseudo-inverse method. Let us begin by constructing the matrix G comprised of the responses of each hidden neuron to each input vector in the training set. Thus

$$g_{ij} = \exp(-\frac{M}{d^2} \parallel \mathbf{x}^i - \mathbf{c}_j \parallel^2), i = 1,2,...,N; j = 1,2,...,M \tag{8.39}$$

i.e. the response of the j^{th} hidden neuron to the i^{th} training pattern. Then the pseudo-inverse of G is given by

$$G^+ = (G^TG)^{-1}G^T \tag{8.40}$$

which can always be calculated since (G^TG) is a square non-singular matrix. (Even if it is singular because of redundancy in the input data, there are fast efficient methods of calculating the pseudo-inverse.) Now it is a property of pseudo-inverses that $G^+G = I$, the identity matrix, and so since

$$\begin{aligned} \mathbf{t} &= \mathbf{w}^TG \\ \text{Then } \mathbf{w} &= \mathbf{t}G^+ \end{aligned}$$

This method is not a true neural method since it requires no learning but is efficient.

8.6.2 Self-organised Selection of Centres

We can however opt to move the location of our basis functions at the same time as training the weights. When we do so the model becomes a non-linear one and the closed form (matrix) solutions no longer become possible. A radial basis function is non-linear if the basis functions can move, change size or change basis functions in any way.

One possibility for training such a network is to use an unsupervised training method on the weights (while continuing to use the supervised learning on the weights). We could choose to allocate each point to a particular radial basis function (i.e. such that the greatest component of the hidden layer's activation comes from a particular neuron) according to the *k-nearest neighbours rule*. In this rule, a vote is taken among the k-nearest neighbours as to which neuron's centre they are closest and the new input is allocated accordingly. The centre of the neuron is moved so that it remains the average of the inputs allocated to it.

Note that now it does not pay to use a pseudo-inverse method for the output weights since the parameters of the pseudo-inverse are changing at the same time

8.6.3 Supervised Selection of Centres

In this section we allow the positions of the centres to change but do it using a supervised learning approach. We can use a the same error function as before with a generalisation of the LMS rule:

$$\Delta c_i = -\eta_2 \frac{\partial E}{\partial c_i} \tag{8.41}$$

This unfortunately is not guaranteed to converge (unlike the equivalent weight change rule) since the cost function E is not convex with respect to the centres and so a local minimum is possible. Note that the learing rate need not be the same as that used to update the weights.

It is usual to combine this with the supervised adaption of the width parameter. So in this approach, all the parameters of the network are dynamically changed using supervised learning.

In general it has been found that radial basis networks all of whose parameters are modified using supervised learning procedures have greater generalisation properties than those which either do not change the location of the centres or those which change their centres using an unsupervised algorithm.

8.7 Cross-Validation

We have seen that to minimise an error function, it is essential to

- determine the correct model
- determine the optimal parameters for this model

We have used error descent typically as our chosen method for the second of these but one message of this chapter is that it is essential to pay close attention to the first issue. But these two issues are features not only of the training data but also of the test data - we are above all seeking good generalisation which means that the network must perform as well as possible on the test set as well as the training set. But how can we in advance state what is optimal performance on the test set? One method is the method of cross-validation.

The basic form of cross-validation is to split the p members of the data set into two subsets, one containing p-1 members which will be used for training, and the other single element which will be used to evaluate the validation error. We repeat this with each of the p members being omitted in turn and calculate the total validation error. This is known as leave-one-out validation. It has the obvious disadvantage that it requires us to retrain the network p times.

We may have sufficient data (or be so short of processing time) that it is possible to consider splitting the p members of the set into only $\frac{p}{N}$ subsets each of size N. Now our validation is leave-N-out validation.

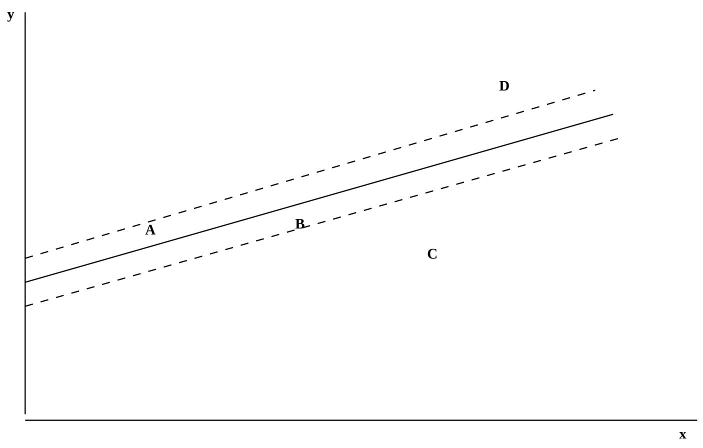


Figure 8.12: Only the points C and D will be used to find the correct direction of w and intercept b. A and B are close enough.

8.8 Support Vector Machines

One of the most popular recent methods has been that of Support Vector Machines(SVMs). The SVM tries to formalise the intuitive idea that only some of the data points which are used to train a supervised network are actually necessary: some points are redundant as training points and only add to the length of time a simulation takes without improving the accuracy of the final result.

Let us imagine that we are given a set of training points $\{(\mathbf{x}_1, y_1), \mathbf{x}_2, y_2), ..., \mathbf{x}_l, y_l)\}$ which we wish to use to train a neural network. Let us begin by describing the situation where there is a linear relationship between inputs and outputs. Then we have

$$y = f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b \tag{8.42}$$

where $\langle , \rangle$ denotes the dot product. Now we wish to have functions which are as flat as possible which in the above case means that we wish to have w as small as possible. One way to do this is to minimise the Euclidean distance

$$||\mathbf{w}||^2 \tag{8.43}$$

subject to the constraints that we wish the resulting function to approximate the true function i.e. subject to the constraint that the difference at each data point between y_i and $f(\mathbf{x}_i)$ is (almost) 0. This results in a situation such as shown in Figure 8.13. The points A and B are close enough to the correct line that they can be ignored while C and D are further away and must be used to find $\mathbf{w}$ and b.

8.8.1 Classification

Consider the special case where the function $f(\mathbf{x}_i) = 1$ or $-1, \forall i$. i.e. we are classifying all points as belonging to one of two classes. Then we are looking for $\mathbf{w}$ and b such that

$$\mathbf{x}_i.\mathbf{w} + b \geq +1 \text{ if } y_i = +1 \tag{8.44}$$
$$\mathbf{x}_i.\mathbf{w} + b \leq -1 \text{ if } y_i = -1 \tag{8.45}$$

These can be combined into one set of inequalities:

$$y_i(\mathbf{x}_i.\mathbf{w} + b) - 1 \geq 0 \forall i \tag{8.46}$$

Now the points for which the equality in equation 8.44 holds lie on the hyperplane $H_1 : \mathbf{x}_i.\mathbf{w} + b = 1$ which has perpendicular distance from the origin $= \frac{1-b}{||\mathbf{w}||}$ where $||\mathbf{w}||$ is the euclidean norm (length) of $\mathbf{w}$. Similarly equation 8.45 leads to the hyperplane $H_2 : \mathbf{x}_i.\mathbf{w} + b = -1$ which has perpendicular distance from the origin $= \frac{-1-b}{||\mathbf{w}||}$. So the width of the margin is $\frac{1}{||\mathbf{w}||}$ and what we wish to do is to minimise this width while still performing accurate classification. Also the training points which just lie on these hyperplanes are the only ones necessary to determine the parameters $\mathbf{w}$ and b and they are known as the *support vectors*. We can solve this problem using the method of Lagrange multipliers: we must minimise

$$L_p = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{l} \alpha_i y_i(\mathbf{w}.\mathbf{x}_i + b) + \sum_{i=1}^{l} \alpha_i \tag{8.47}$$

where the last constraint ensures that the $\alpha_i \geq 0$. Note that we are now required to minimise L_p with respect to $\mathbf{w}$, b and α_i and that at the optimal point the derivative of L_p with respect to these parameters should vanish.

Because the problem is convex, we can equivalently optimise the "dual" problem: *maximise* L_p subject to the constraints that the gradient of L_p with respect to $\mathbf{w}$ and b vanish. Differentiating L_p we get

$$\begin{aligned} \frac{\partial L_p}{\partial \mathbf{w}} &= \mathbf{w} - \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i \\ \text{and so } \mathbf{w} &= \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i \end{aligned}$$

at the optimum point. Similarly finding the derivative with respect to b and setting this equal to 0 gives

$$\sum_i \alpha_i y_i = 0 \tag{8.48}$$

which we can substitute into the problem to give us

$$L_D = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j \tag{8.49}$$

which is the dual problem which we must maximise.

Support vector training in the linear separable case then involves using the last equation to solve the original optimisation problem. Only those points which are on the separating hyperplanes will have values of α_i not equal to zero. All points which are currently correctly classified will lie within acceptable bounds and will not be used for determining the parameters.

8.8.2 A Case Study

Our goal was to analyse a two-dimensional data-set and try to discover the best-fitting set of regression curves that both generalised the data and followed the laws of probability, while also taking into account large variances in the sample sizes associated with each set of observations.

In particular, we analysed a dataset of soccer results where the independent variable is the bookmaker's evaluation of the winning chan-ces of the favourite at the end of the match, and the dependent variable is the actual frequency of a particular 'double-result' occurring given the bookmaker's prior evaluation of the odds. A 'double-result' is the combination of the outcomes at the end of the first and second halves and so, as we have three possible different outcomes of home win, draw, or away win at the end of each half, we have nine possible double-results in all. It should be noted that the use of the full-time favourite odds as independent variables are a good enough approximation to a smooth and consistent scale for our dependent variables to be considered viable - however one of our main difficulties is that we have far larger sample sizes for some dependent variables than others, and also some double-results occur very infrequently so an accurate regression fit may be difficult.

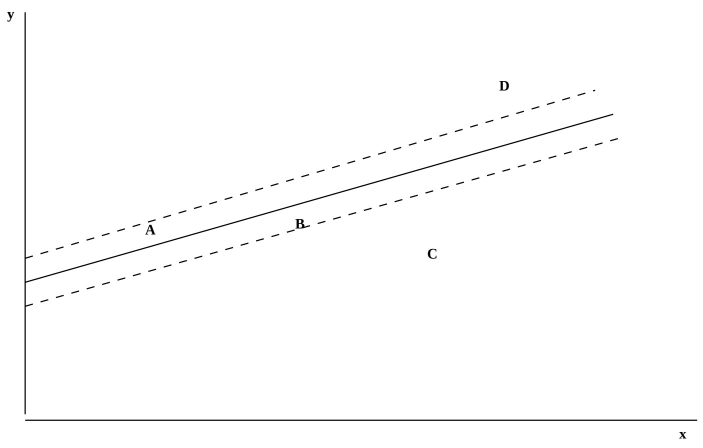


Figure 8.13: Only the points C and D will be used to find the correct direction of w and intercept b. A and B are close enough.

The independent variable is based on odds of favourites between 6/4 and 1/3. This is turned into an independent variable range of between -1 and 1 where -1 is equivalent to the probability that the away team has 100% chance of winning and 1 is equal to the probability that the home team has 100% chance of winning, where such probabilities are given by the bookmaker.

However some independent variable ranges are very poorly represented e.g. there are very few independent variables near the extremities of ± 1 (see Table 8.1. In fact the majority of our data exists for values between -0.2 and +0.5; outside of this range, there is a very sharp decline in the number of observations.

This section discusses an application demonstrating how to build into a Support Vector Machine (SVM) [2, 14] knowledge of which points are more reliable than others while keeping the good generalisation and smoothing properties of SVMs.

The ϵ parameter is the focus of our attention. If we prescribe in advance the required degree of accuracy, we may set ϵ to an appropriate value. Sometimes we simply wish to investigate what degree of accuracy gives us best generalisation over the data. However for our data set, we have knowledge about the data set additional to the dependent and independent variables: we know that some values of the (x_i, y_i) data set are based on a greater number of samples than others. Therefore we build in this knowledge by altering the value of ϵ at each data point. Therefore we now have a vector of parameters, $\epsilon_i, i = 1, ..., N$, one for each data point. To calculate ϵ_i we use the sample size, s_i from which the dependent variable was calculated. Define

$$t_i = \frac{s_i}{\sum_j s_j} \tag{8.50}$$

Let t_{min} be the smallest value of t_i so calculated and t_{max} be the largest. We chose ϵ_{max} and ϵ_{min} to be the maximum and minimum permitted values of ϵ_i respectively and set

$$\epsilon_i = \frac{t_i - t_{min}}{t_{max} - t_{min}} * (\epsilon_{max} - \epsilon_{min}) + \epsilon_{min} \tag{8.51}$$

We then use the interior point optimiser, LOQO [15] to solve the quadratic optimisation problem.

8.8.3 Simulations

The number of observations for each (x_i, y_i) pair is shown in Figure 8.14 as is the corresponding value of ϵ_i. We see that the greatest number of samples and hence the smallest values of ϵ_i occur in the central region between -0.2 and 0.5. In Figure 8.15 we show one set of regression curves for the

Fav. Odds	Sample	H-H	H-D	H-A	D-H	D-D	D-A	A-H	A-D	A-A
1/5	3	0	0	0	2	1	0	0	0	0
2/9	7	3	0	0	2	1	0	0	0	1
1/4	21	11	1	0	4	3	0	1	0	1
2/7	36	15	1	1	12	3	1	0	1	2
1/3	35	22	1	1	5	1	2	0	0	3
4/11	26	14	0	0	1	10	1	0	0	0
2/5	58	28	4	0	13	4	4	1	2	2
4/9	87	52	4	0	12	6	4	5	0	4
1/2	70	33	6	0	13	8	2	4	1	3
8/15	74	26	2	1	15	15	4	3	5	3
4/7	164	57	10	2	40	23	10	3	5	14
8/13	200	72	11	3	45	28	10	8	8	15
4/6	229	104	4	2	46	31	13	5	7	17
8/11	303	83	16	1	50	60	32	14	18	29
4/5	251	75	15	5	38	46	29	0	17	26
5/6	112	32	5	1	23	19	7	5	5	15
10/11	235	69	6	4	38	48	25	3	12	30
1/1	200	57	8	2	32	41	16	8	18	18
11/10	200	48	19	3	25	34	20	4	14	33
6/5	230	53	15	5	33	46	31	3	10	34
5/4	322	69	25	6	43	58	40	9	18	54
11/8	236	59	14	5	33	45	23	5	5	47
6/4	232	56	12	1	24	50	28	5	11	45

Table 8.1: The odds and outcomes of English Premier division games in one season.

variable ϵ_i method (top) and the corresponding curve for the single ϵ method. On each graph we also show the regression curve from an alternative method based on a second degree polynomial which also takes into account the sample sizes and which has been shown to be successful at making its creator money! The variable ϵ_i method is clearly closer to the proven technique. We conjecture that the end points are having a more pivotal effect in the latter case.

Figure 8.14: The number of data points and corresponding value of ϵ at each data pair.

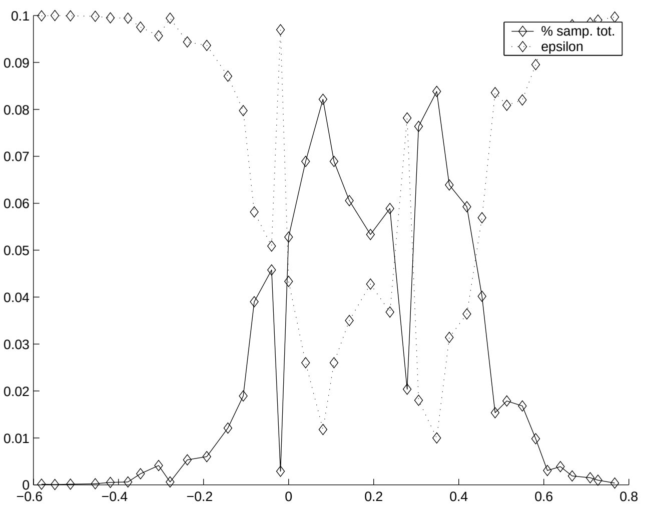
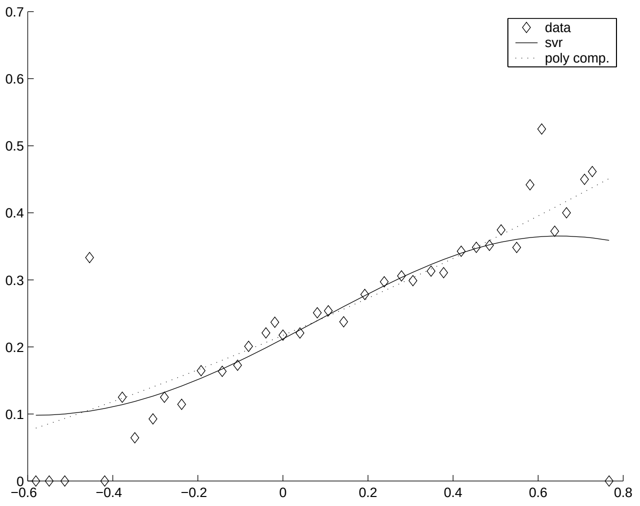


Figure 8.15: The top diagram shows the regression curve from the variable ϵ_i case. The bottom shows that from the constant ϵ case on the same data set. Each graph also shows the regression curve from a tried and trusted polynomial regression which also takes into account the sample sizes.



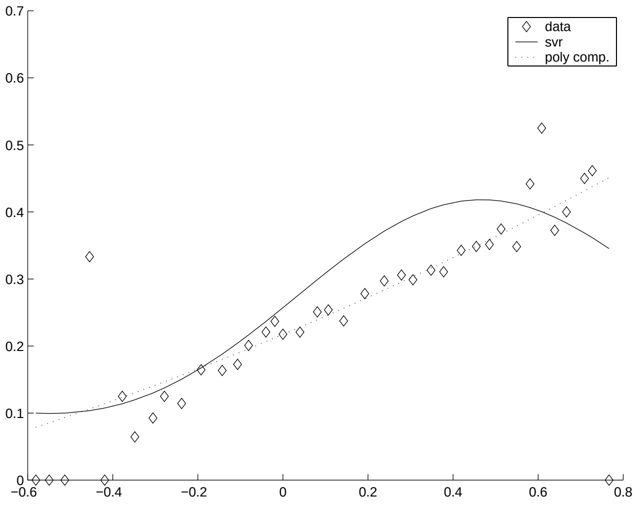
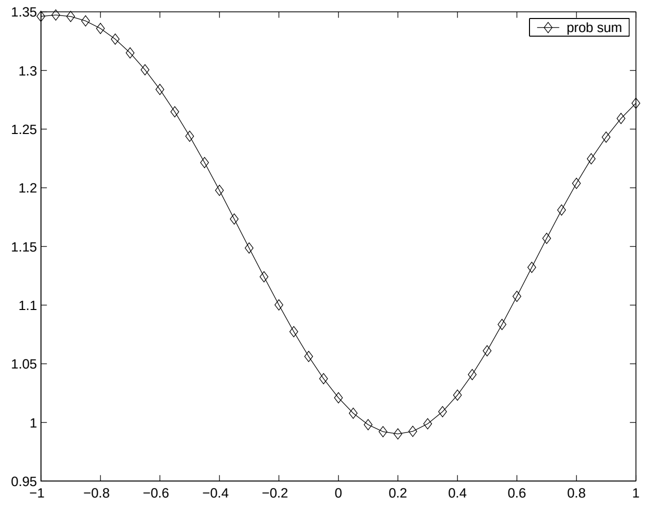


Figure 8.16: The sum of the probabilities using the variable ϵ method. The probabilities sum to approximately 1 in the region in which we have most data but diverge substantially from 1 at the extremities.



Bibliography

[1] D. Charles and C. Fyfe. Modelling multiple cause structure using rectification constraints. *Network: Computation in Neural Systems*, 1998.

[2] N Christiani and J Shawe-Taylor. *An Introduction to Support Vector Machines and other kernel-based learning methods.* Cambridge University Press, 2000.

[3] C. Fyfe and R. Baddeley. Non-linear data structure extraction using simple hebbian networks. *Biological Cybernetics*, 72(6):533–541, 1995.

[4] S. Mika, B. Scholkopf, A. Smola, K.-R. Muller, M. Scholz, and G. Ratsch. Kernel pca and de-noising in feature spaces. In *Advances in Neural Processing Systems, 11*, 1999.

[5] P.L.Lai and C. Fyfe. A neural network implementation of canonical correlation analysis. *Neural Networks*, 12(10):1391–1397, Dec. 1999.

[6] S. Romdhani, S. Gong, and A. Psarrou. A multi-view nonlinear active shape model using kernel pca. In *BMVC99*, 1999.

[7] B. Scholkopf, S. Mika, C. Burges, P. Knirsch, K.-R. Muller, G. Ratsch, and A. J. Smola. Input space vs feature space in kernel-based methods. *IEEE Transactions on Neural Networks*, 10:1000–1017, 1999.

[8] B. Scholkopf, S. Mika, A. Smola, G. Ratsch, and K.-R. Muller. Kernel pca pattern reconstruction via approximate pre-images. In L. Niklasson M. Boden R. Ziemke, editor, *Proceedings of 8th International Conference on Artificial Neural Networks*, pages 147–152. Springer Verlag, 1998.

[9] B. Scholkopf, A. Smola, and K.-R. Muller. Nonlinear component analysis as a kernel eigenvalue problem. Technical Report 44, Max Planck Institut fur biologische Kybernetik, Dec 1996.

[10] B. Scholkopf, A. Smola, and K.-R. Muller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.

[11] B. Scholkopf, A. Smola, and K.-R. Muller. *Support Vector Machines*, chapter Kernel Principal Component Analysis, pages 327–370. 1999.

[12] A. J. Smola, O. L. Mangasarian, and B Scholkopf. Sparse kernel feature analysis. Technical Report 99-04, University of Wiscosin Madison, 1999.

[13] A. J. Smola, S. Mika, B. Scholkopf, and R. C. Williamson. Regularized principal maniforlds. *Machine Learning*, pages 1–28, 2000. (submitted).

[14] A. J. Smola and B. Scholkopf. A tutorial on support vector regression. Technical Report NC2-TR-1998-030, NeuroCOLT2 Technical Report Series, Oct. 1998.

[15] R. J. Vanderbei. Loqo: An interior point code for quadratic programming. Technical Report SOR-94-15, Princeton University, Sept. 1998.

[16] V Vapnik. *The nature of statistical learning theory.* Springer Verlag, New York, 1995.

8.9 Conclusion

The theme of this Chapter is an investigation of the efficacy of training a network on one set of data and testing it on another.

We have investigated two aspects of this both of which can be used to quantify the generalisation properties of ANNs. The VC dimension is an abstract tool which provides a worst case scenario for analysing the power of a neural network; as such it often seems to have little relevance to everyday neural network construction. The PAC system is created with a similar intention in that it is used to give confidence in a network's outputs.

The Bias-Variance trade-off on the other hand is very real and we have used the medium of a different type of supervised neural network to show its relevance.

We have also met the technique of regularisation which is intended to ensure that we find a smooth function and so hopefully good interpolation for data which was not seen during training.

Chapter 9

Unsupervised Learning using Kernel Methods

9.1 Introduction

In this Chapter, we use radial kernels to learn mappings in an unsupervised manner. The use of radial kernels has been derived from the work of Vapnik [16], Burges [?] etc in the field of Support Vectors Machines. Support Vector regression for example, performs a nonlinear mapping of the data set into some high dimensional feature space in which we may then perform linear operations. Since the original mapping was nonlinear, any linear operation in this feature space corresponds to a nonlinear operation in data space.

We first review recent work on Kernel Principal Component Analysis (KPCA) [12, 13, 9, 10, 7, 6, 11, 4, 8] which has been the most frequently reported linear operation involving unsupervised learning in feature space. We then extend the method to perform other Kernel-based operations: Kernel Principal Factor Analysis, Kernel Exploratory Projection Pursuit and Kernel Canonical Correlation Analysis. For each operation, we derive the appropriate rules and give exemplar simulation results. Since this book is on Radial Basis Functions, we will report results only using radial kernels, however the theory is quite general and many interesting results may be had using non-radial kernels.

9.2 Kernel PCA

This section is based very much on the analysis in [9, 10]. A very good Matlab simulation of Kernel PCA can be found at

`http://svm.first.gmd.de`

In the next section, we show that sample Principal Component Analysis (PCA) may be performed on the samples of a data set in a particular way which will be useful in the performance of PCA in the nonlinear feature space.

9.2.1 The Linear Kernel

PCA finds the eigenvectors and corresponding eigenvalues of the covariance matrix of a data set. Let $\chi = \{\mathbf{x}_1, ..., \mathbf{x}_M\}$ be iid (independent, identically distributed) samples drawn from a data source. If each $\mathbf{x}_i$ is n-dimensional, $\exists$ at most n eigenvalues/eigenvectors. Let C be the covariance matrix of the data set; then C is $n \times n$. Then the eigenvectors, $\mathbf{e}_i$, are n dimensional vectors which are found by solving

$$C\mathbf{e} = \lambda\mathbf{e} \tag{9.1}$$

where λ is the eigenvalue corresponding to $\mathbf{e}$. We will assume the eigenvalues and eigenvectors are arranged in non-decreasing order of eigenvalues and each eigenvector is of length 1. We will use the sample covariance matrix as though it was the true covariance matrix and so

$$C \approx \frac{1}{M}\sum_{j=1}^{M}\mathbf{x}_j\mathbf{x}_j^T \tag{9.2}$$

Now each eigenvector lies in the span of χ; i.e. the set $\chi = \{\mathbf{x}_1, ..., \mathbf{x}_M\}$ forms a basis set (normally overcomplete since $M > n$) for the eigenvectors. So each $\mathbf{e}_i$ can be expressed as

$$\mathbf{e}_i = \sum_j \alpha_i^j \mathbf{x}_j \tag{9.3}$$

Now if we wish to find the principal components of a new data point $\mathbf{x}$ we project it on the eigenvectors previously found: the first principal component is $(\mathbf{x}.\mathbf{e}_1)$, the second is $(\mathbf{x}.\mathbf{e}_2)$, etc. These are the coordinates of $\mathbf{x}$ in the eigenvector basis. There are only n eigenvectors (at most) and so there can only be n coordinates in the new system: we have merely rotated the data set.

Now consider projecting one of the data points from χ on the eigenvector $\mathbf{e}_1$; then

$$\mathbf{x}_k.\mathbf{e}_1 = \mathbf{x}_k.\sum_j \alpha_1^j \mathbf{x}_j = \ \alpha_1.\sum_j \mathbf{x}_k\mathbf{x}_j \tag{9.4}$$

Now let K be the matrix of dot products. Then $K_{ij} = \mathbf{x}_i\mathbf{x}_j$.

Multiplying both sides of (9.1) by $\mathbf{x}_k$ we get

$$\mathbf{x}_k C\mathbf{e}_1 = \lambda\mathbf{e}_1.\mathbf{x}_k \tag{9.5}$$

and using the expansion for $\mathbf{e}_1$, and the definition of the sample covariance matrix, C, gives

$$\frac{1}{M}K^2\alpha_1 = \lambda_1 K\alpha_1 \tag{9.6}$$

Now it may be shown [10] that all interesting solutions of this equation are also solutions of

$$K\alpha_1 = M\lambda_1\alpha_1 \tag{9.7}$$

whose solution is that α_1 is the principal eigenvector of K.

9.2.2 Non linear Kernels

Now we preprocess the data using $\Phi : \chi \rightarrow F$. So F is now the space spanned by $\Phi(\mathbf{x}_1), ..., \Phi(\mathbf{x}_M)$. The above arguments all hold and the eigenvectors of the dot product matrix $K_{ij} = (\Phi(\mathbf{x}_i).\Phi(\mathbf{x}_j))$. But now the *Kernel Trick*: provided we can calculate K we don't need the individual terms $\Phi(\mathbf{x}_i)$.

In this Chapter, we will exclusively use Gaussian kernels so that

$$K_{ij} = (\Phi(\mathbf{x}_i).\Phi(\mathbf{x}_j)) = \exp(-(\mathbf{x}_i - \mathbf{y}_i)^2/(2\sigma^2)) \tag{9.8}$$

This kernel has been shown [10] to satisfy the conditions of Mercer's theorem and so can be used as a kernel for some function $\Phi(.)$. One issue that we must address in feature space is that the eigenvectors should be of unit length. Let $\mathbf{v}_i$ be an eigenvector of C. Then $\mathbf{v}_i$ is a vector in the space F spanned by $\Phi(\mathbf{x}_1), ..., \Phi(\mathbf{x}_M)$ and so can be expressed in terms of this basis. This is an at most M-dimensional subspace of a possibly infinite dimensional space which gives computational tractibility to the kernel algorithms. Then

$$\mathbf{v}_i = \sum_{j=1}^{M}\alpha_j^i\Phi(\mathbf{x}_j) \tag{9.9}$$

for eigenvectors $\mathbf{v}_i$ corresponding to non-zero eigenvalues. Therefore

$$\begin{aligned}
\mathbf{v}_i^T \mathbf{v}_i &= \sum_{j,k=1}^{M} \alpha_j^i \Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_k) \alpha_k^i \\
&= \sum_{j,k=1}^{M} \alpha_j^i K_{jk} \alpha_k^i \\
&= \alpha^i.(K\alpha^i) \\
&= \lambda_i \alpha^i.\alpha^i
\end{aligned}$$

Now α^i are (by definition of the eigenvectors of K) of unit magnitude. Therefore since we require the eigenvectors to be normalised in feature space, F, i.e. $\mathbf{v}_i^T \mathbf{v}_i = 1$, we must normalise the eigenvectors of K, α^i, by dividing each by the square root of their corresponding eigenvalues.

Now we can simply perform a principal component projection of any new point $\mathbf{x}$ by finding its projection onto the principal components of the feature space,F. Thus

$$\mathbf{v}_i.\Phi(\mathbf{x}) = \sum_{j=1}^{M} \alpha_i^j \Phi(\mathbf{x}_j).\Phi(\mathbf{x}) = \sum_{j=1}^{M} \alpha_i^j K(\mathbf{x}_j, \mathbf{x}) \tag{9.10}$$

And the above argument shows that any operation which can be defined in terms of dot products can be Kernelised. We will in subsequent sections use similar arguments with Factor Analysis, Exploratory Projection Pursuit and Canonical Correlation Analysis; however first we give an illustratory example.

There are many examples of KPCA in the literature (e.g. [10, 7, 6]) and we will in this Chapter only give results using KPCA later when we wish to compare it with other Kernel methods.

9.2.3 The Curse of Kernel Dimensionality

One of the difficulties associated with unsupervised kernel methods in general is that the nonlinear mapping, $\Phi()$ maps the input data to a feature space of high, possibly infinite, dimensionality. Now one of the advantages kernel methods are said to have is that they are actually working in a space of dimensionality equal only to the number of data points. This is true but in practise the dimensionality may be even less than that if two or more data points coincide. Then we have a reduced rank K matrix. Perhaps more common is the situation when two or more points lie very close to one another (in feature space) and then we have an ill-conditioned K matrix whose lower eigenvectors are very susceptible to noise. It is our finding that kernel methods are typically plagued with problems of this type, a fact which should not be surprising given that we are estimating M eigenvectors from M points. Some methods for creating a reduced set of vectors have been suggested in the past. In addition to addressing this problem, such methods also alleviate the computational problems associated with large matrices. We now suggest further methods to find a reduced set and perform an empirical study of some of these methods.

The data set we will use consists of 65 colour spectra of 115 asteroids used by [?]. The data set is composed of a mixture of the 52-colour survey by [?], together with the 8-colour survey conducted by [?] providing a set of asteroid spectra spanning 0.3-2.5mm. Success is measured in terms of the accuracy of classification of asteroids into the correct data types.

We report results in Table 9.1 (Kernel Factor Analysis is discussed in the next section- the results are shown here for completeness). The table shows the effectiveness of performing a linear classification of the projections onto feature space of the data. The methods for choosing a reduced set of points (called initialisation methods in the Table) were:

Random We simply randomly select a subset of the data set to use as the points which we use to perform a Kernel PCA. Typically we use 100 out of the 116 points.

Initialisation method	Kernel method	Dataset	Accuracy
Random	Principal Component Analysis	Asteroid	63%
k-Means	Principal Component Analysis	Asteroid	68%
SOM	Principal Component Analysis	Asteroid	74%
MoG	Principal Component Analysis	Asteroid	67%
Random	Principal Factor Analysis	Asteroid	65%
k-Means	Principal Factor Analysis	Asteroid	70%
SOM	Principal Factor Analysis	Asteroid	75%
MoG	Principal Factor Analysis	Asteroid	68%

Table 9.1: The percentage accuracy of two kernel methods when different means of creating the data vectors used to determine the kernels.

k-Means We select 50 centres selected using the k-means algorithm. These centres became 'virtual points' on which the KPCA algorithm was performed. Often we found that some centres coincided in which case one was removed (see discussion on ill-conditioned covariance matrices).

SOM We begin with a 10*10 grid of output neurons in a Self-Organising Map [?] and train the network on this data set. Typically around half the nodes are not responding to a part of the data space and are discarded. The centres from the other half are used as a form of 'virtual data' on which to perform the KPCA.

MoG We assume the data was formed from a Mixtures of Gaussian causes (we actually used diagonal covariance matrices for the Gaussian covariance matrices) and optimised the parameters of the model using the EM Algorithm. Typically we found that the data set was best explained by 20-30 Gaussians whose means were used as 'virtual data' on which the KPCA was performed. The centres were found using the EM algorithm.

We see from Table 9.1 that the SOM provided the best points (in terms of accuracy of the clustering) on which KPCA was performed. Perhaps the most interesting result is the difference between the SOM and the k-means algorithm; this may best be explained by the fact that the centres in the k-means algorithm are totally competitive and tend to move to the regions of greatest mass. In the SOM, the topology preservation causes some neurons to map to regions of lower probability density, something which in other guises is a drawback when the mapping has neurons whose centres are in areas of low mass. This study is part of a larger study over different data sets, however these results seem to be typical.

9.3 Kernel Principal Factor Analysis

A standard method of finding independent sources in a data set is the statistical technique of Factor Analysis (FA). PCA and FA are closely related statistical techniques both of which achieve an efficient compression of the data but in a different manner. They can both be described as methods to explain the data set in a smaller number of dimensions but FA is based on assumptions about the nature of the underlying data whereas PCA is model free.

We can also view PCA as an attempt to find a transformation from the data set to a compressed code, whereas in FA we try to find the linear transformation which takes us from a set of hidden factors to the data set. Since PCA is model free, we make no assumptions about the form of the data's covariance matrix. However FA begins with a specific model which is usually constrained by our prior knowledge or assumptions about the data set. The general FA model can be described by the following relationship:

$$\mathbf{x} = L\mathbf{f} + \mathbf{u} \tag{9.11}$$

where $\mathbf{x}$ is a vector representative of the data set, $\mathbf{f}$ is the vector of factors, L is the matrix of factor loadings and $\mathbf{u}$ is the vector of specific (unique) factors.

The usual assumptions built into the model are that:

- $E(\mathbf{f}) = 0$, Var($\mathbf{f}$) =I i.e. the factors are zero mean, of the same power and uncorrelated with each other.
- $E(\mathbf{u}) = 0$, Cov($\mathbf{u}_i, \mathbf{u}_j$) = 0, $\forall i, j$, i.e the specific factors are also zero mean and uncorrelated with each other
- Cov($\mathbf{f}, \mathbf{u}$) = 0 i.e. the factors and specific factors are uncorrelated

Let $C = E(\mathbf{x}\mathbf{x}^T)$ be the covariance matrix of $\mathbf{x}$ (again assuming zero mean data). Then $C = \Lambda\Lambda^T + \Phi$ where Φ is the covariance matrix of the specific factors, $\mathbf{u}$ and so Φ is a diagonal matrix, $diag\{\Phi_{11}, \Phi_{22}, , \Phi_{MM}\}$. Now whereas PCA attempts to explain C without a specific model, FA attempts to find parameters Λ and Φ which explain C and only if such models can be found will a Factor Analysis be successful.

Estimations of the Factor loading is usually done by means of one of two methods - Maximum Likelihood Estimation or Principal Factor Analysis [?]. Since Principal Factor Analysis is a method which may be performed using dot products, this is the method in which we shall be interested in this Chapter.

9.3.1 Principal Factor Analysis

Expanding $C = E(\mathbf{x}\mathbf{x}^T)$, then

$$C_{ii} = \sum_j \lambda_{ij}^2 + \Phi_{ii}^2 = h_i^2 + \Phi_{ii}^2 \tag{9.12}$$

i.e. the variance of the data set can be broken into two parts the first of which is known as the communality and is the variance of x_i which is shared via the factor loadings with the other variables. The second is the specific or unique variance associated with the i^{th} input.

In Principal Factor Analysis (PFA), an initial estimate of the communalities is made. This is inserted into the main diagonal of the data covariance matrix and then a PCA is performed on the "reduced correlation matrix". A commonly used estimate of the communalities is the maximum of the square of the multiple correlation coefficient of the i^{th} variable with every other variable.

We have previously [1] derived a neural network method of performing Principal Factor Analysis.

9.3.2 The Varimax Rotation

In PCA the orthogonal components are arranged in descending order of importance and a unique solution is always possible. The factor loadings in FA are not unique and there are likely to be substantial loadings on more than one factor that may be negative or positive. This often means that the results in standard FA are difficult to interpret. To overcome these problems it is possible to perform a rigid rotation of the axes of the factor space and so identify a more simplified structure in the data that is more easily interpretable. One well-known method of achieving this is the Varimax rotation [?]. This has as its rationale that factors should be formed with a few large loadings and as many near zero loadings as possible, normally achieved by an iterative maximization of a quadratic function of the factor loadings. It is worth noting that the Varimax rotation aims for a sparse response to the data and this has acknowledged as an efficient form coding. In the experiments presented here it can be seen that by using a Varimax rotation we can gain more straightforward, interpretable results than with PCA in kernel space.

9.3.3 Kernel Principal Factor Analysis

Let $\chi = \mathbf{x}_1, ..., \mathbf{x}_M$ be iid (independent, identically distributed) samples drawn from a data source. Let C be the covariance matrix of the data set and let us define $C^- = C - D$ where we will assume that D is a diagonal matrix of the form $D = \mu I$. We are thus stripping out the same amount of variance from each element in the diagonal of the covariance matrix of the data.

Then the eigenvectors of this reduced covariance matrix, $\mathbf{e}_i$, are n dimensional vectors which are found by solving

$$C^{-}\mathbf{e}_i = \lambda_i \mathbf{e}_i \tag{9.13}$$

where λ_i is the eigenvalue corresponding to $\mathbf{e}_i$. We will assume the eigenvalues and eigenvectors are arranged in non-decreasing order of eigenvalues and each eigenvector is of length 1. We will use the sample covariance matrix as though it was the true covariance matrix and so

$$C^{-} \approx \frac{1}{M}\sum_{j=1}^{M} \mathbf{x}_j \mathbf{x}_j^T - \mu I \tag{9.14}$$

Then

$$\begin{aligned} C^{-}\mathbf{e}_i &= \frac{1}{M}\sum_{j=1}^{M} \mathbf{x}_j \mathbf{x}_j^T \mathbf{e}_i - \mu I \mathbf{e}_i \\ \text{i.e. } \lambda_i \mathbf{e}_i &= \frac{1}{M}\sum_{j=1}^{M} \mathbf{x}_j \mathbf{x}_j^T \mathbf{e}_i - \mu \mathbf{e}_i \\ \text{Thus } \sum_{j=1}^{M} \mathbf{x}_j(\mathbf{x}_j^T \mathbf{e}_i) &= M(\mu + \lambda_i)\mathbf{e}_i \qquad (9.15) \end{aligned}$$

Thus each eigenvector lies in the span of χ; i.e. the set $\chi = \mathbf{x}_1, ..., \mathbf{x}_M$ forms a basis set for the eigenvectors. So each $\mathbf{e}_i$ can be expressed as

$$\mathbf{e}_i = \sum_j \alpha_{ij}\mathbf{x}_j \tag{9.16}$$

Now if we wish to find the principal components of a new data point $\mathbf{x}$ we project it on the eigenvectors previously found: the first principal component is $\mathbf{x}.\mathbf{e}_1$, the second is $\mathbf{x}.\mathbf{e}_2$, etc. These are the coordinates of $\mathbf{x}$ in the eigenvector basis. There are only n eigenvectors (at most) and so there can only be n coordinates in the new system: we have merely rotated the data set.

Now consider projecting one of the data points from χ on the eigenvector $\mathbf{e}_1$; then

$$\mathbf{x}_k.\mathbf{e}_1 = \mathbf{x}_k.\sum_j \alpha_{1j}\mathbf{x}_j = \alpha_1.\sum_j \mathbf{x}_k\mathbf{x}_j \tag{9.17}$$

Now let K be the matrix of dot products. Then $K_{ij} = \mathbf{x}_i\mathbf{x}_j$.

Multiplying both sides of (9.13) by $\mathbf{x}_k$ we have

$$\begin{aligned} \mathbf{x}_k^T C^{-}\mathbf{e}_i &= \lambda_i \mathbf{e}_i.\mathbf{x}_k \\ \mathbf{x}_k^T(\frac{1}{M}\sum_{j=1}^{M}\mathbf{x}_j\mathbf{x}_j^T - \mu I)\sum_p \alpha_i^p \mathbf{x}_p &= \lambda_1 \sum_p \alpha_i^p \mathbf{x}_p.\mathbf{x}_k \\ \frac{1}{M}K^2\alpha_i - \mu K \alpha_i &= \lambda_i K \alpha_i \end{aligned}$$

Now it may be shown [10] that all interesting solutions of this equation are also solutions of

$$K\alpha_1 = M(\lambda_i + \mu)\alpha_i \tag{9.18}$$

whose solution is that α_i is a principal eigenvector of K with eigenvalue $\gamma_i = M(\lambda_i + \mu)$. Now, if we have a nonlinear mapping to a feature space, F, then $C^{-} \approx \frac{1}{M}\sum_{j=1}^{M}\Phi(\mathbf{x}_j)\Phi^T(\mathbf{x}_j) - \mu I$ and the above arguments continue to hold.

Note that the direction of each eigenvector is exactly the same as the KPCA solution but the corresponding eigenvalue is different. Since we wish our $\mathbf{e}_i$ to be of unit length we have that

$$\begin{aligned}
\mathbf{e}_i^T \mathbf{e}_i &= \sum_{j,k=1}^{M} \alpha_i^j \Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_k) \alpha_i^k \\
&= \sum_{j,k=1}^{M} \alpha_i^j K_{jk} \alpha_i^k \\
&= \alpha_i.(K\alpha_i) \\
&= \gamma_i \alpha_i.\alpha_i
\end{aligned}$$

where we have used α_i^k to be the k^{th} element of the i^{th} eigenvector.

Now α_i are (by definition of the eigenvectors of K) of unit magnitude. Therefore since we require the eigenvectors to be normalised in feature space, F, i.e. $\mathbf{e}_i^T \mathbf{e}_i = 1$, we must normalise the eigenvectors of K, α_i, by dividing each by the square root of their corresponding eigenvalues, γ_i. But note now that these eigenvalues contain both a term from the eigenvalues of the covariance matrix and also μ (see (9.18)). Thus the Varimax rotation may have a different solution when applied in the KPFA space compared with that found in the KPCA space.

However this assumption of noise in the feature space is less interesting than the assumption of noise in the data space which is then transformed by passing the noise through the nonlinearity into the feature space. Thus we now consider the situation in which the input data contains the noise and the nonlinear function, $\Phi()$ acts on both signal and noise. There seems to be no generic solution to this and so we must consider the effect of the noise through different functions.

- Consider the feature space of all monomials of degree 2 on two dimensional data. i.e.

$$\Phi(\mathbf{x}) = \Phi((x_1, x_2)) = (x_1^2, x_1 x_2, x_2^2) \tag{9.19}$$

  Then if our model is such that $(x_1, x_2) = (x_1^- + \mu_1, x_2^- + \mu_2)$, then

$$\Phi(\mathbf{x}) = ((x_1^- + \mu_1)^2, (x_1^- + \mu_1)(x_2^- + \mu_2), (x_2^- + \mu_2)^2) \tag{9.20}$$

  which, for zero mean noise, has expectation

$$\Phi(\mathbf{x}) = ((x_1^-)^2 + \mu^2, x_1^- x_2^-, (x_2^-)^2 + \mu^2) \tag{9.21}$$

  So the noise in this space does not satisfy the conditions,

$$C^- \approx \frac{1}{M} \sum_{j=1}^{M} \Phi(\mathbf{x}_j) \Phi^T(\mathbf{x}_j) - \mu I$$

- Consider the space formed from the Gaussian function: now there is no known nonlinear function but we may consider the kernel matrix directly

$$\begin{aligned}
K_{ij} &= \Phi(\mathbf{x}_i, \mathbf{x}_j) = \exp(-(\mathbf{x}_i - \mathbf{x}_j)^2/\sigma) \\
&= \exp(-(\mathbf{x}_i^- + \mu_i - \mathbf{x}_j^- - \mu_j)^2/\sigma)
\end{aligned}$$

  where we have used μ_i and μ_j as vectors of noise. Then while it is true that the expected value of this is given by

$$K_{ij} = exp(-(\mathbf{x}_i^- - \mathbf{x}_j^-)^2/\sigma) \tag{9.22}$$

  this is true only of the expectation. Considering this on a data point by data point basis,

$$\exp(-(\mathbf{x}_i^- + \mu_i - \mathbf{x}_j^- - \mu_j)^2/\sigma) \neq exp(-(\mathbf{x}_i^- - \mathbf{x}_j^-)^2/\sigma) \tag{9.23}$$

  other than the special case when $i = j$. Thus we have an interesting analogy to subtracting out a common term from the C covariance matrix: the K matrix is only absolutely correct along its main diagonal.

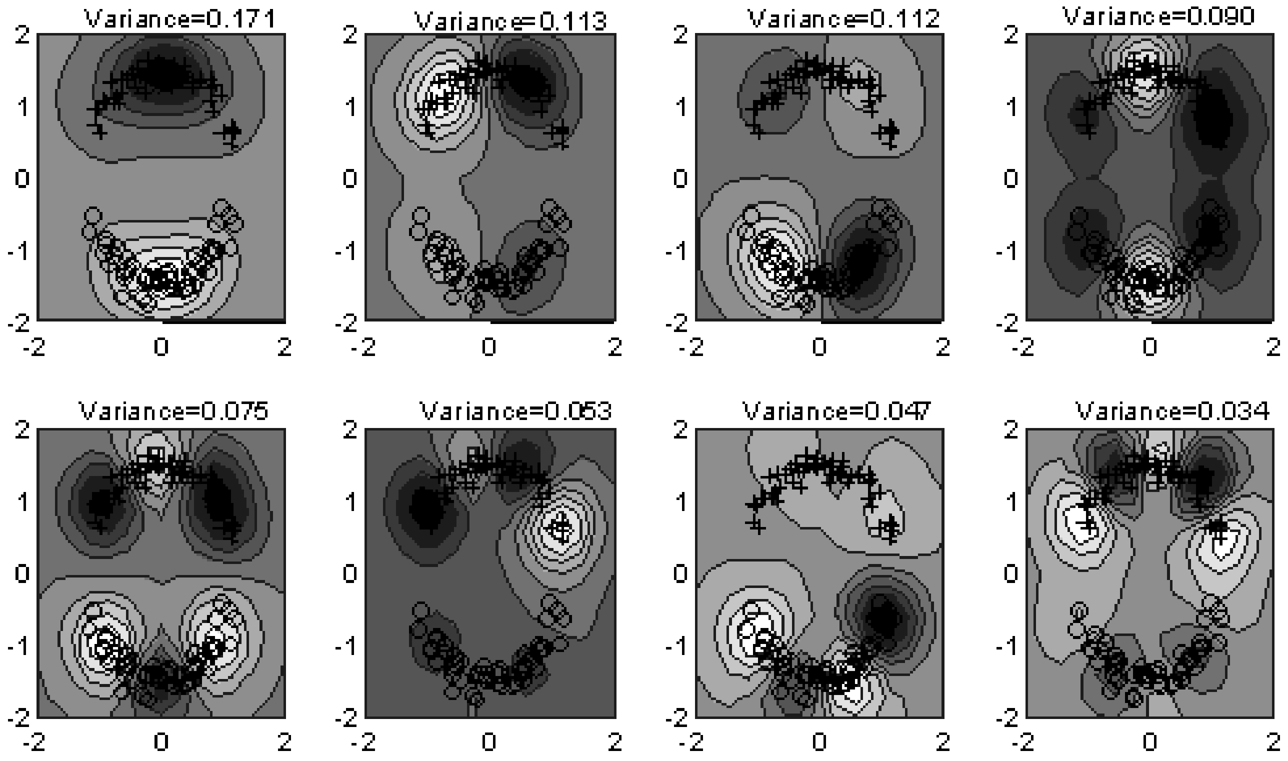


Figure 9.1: The first eigenvectors found by Kernel Principal Component Analysis on the data set shown.

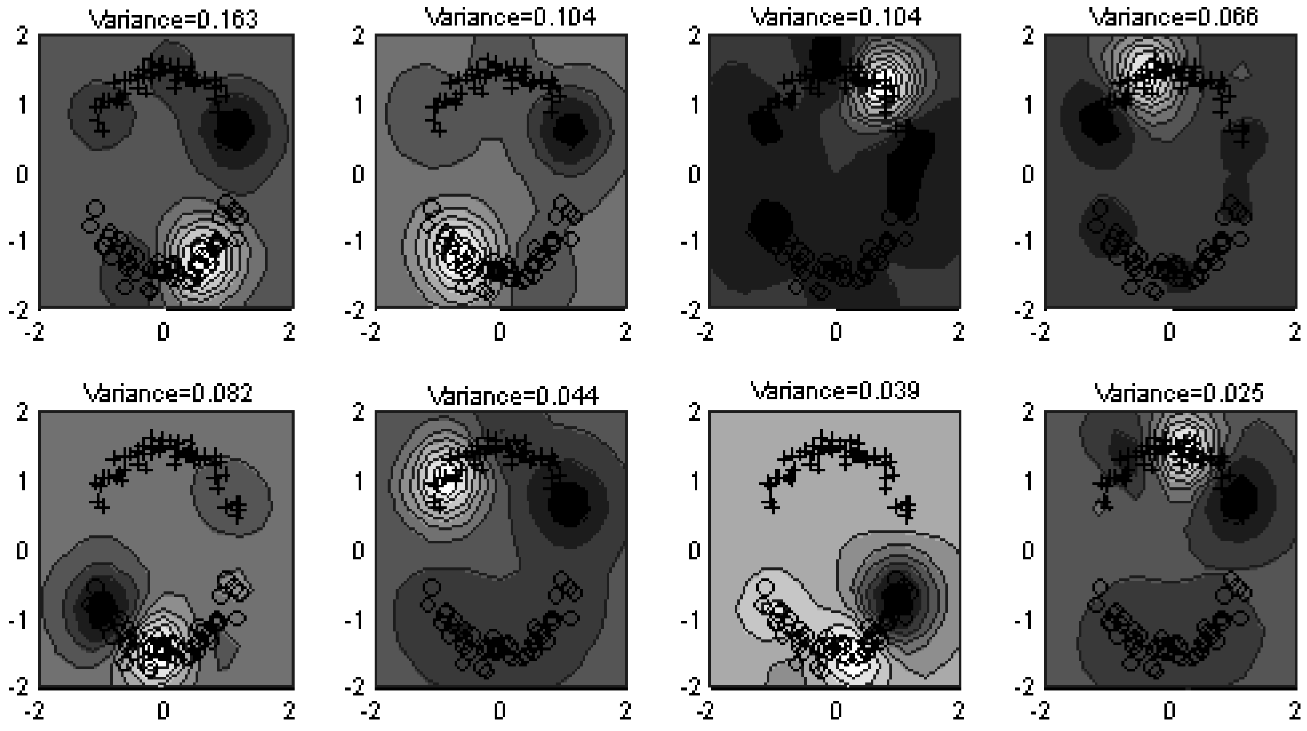


Figure 9.2: The directions found by Kernel Principal Factor Analysis. We may compare this with Figure 9.1.

9.3.4 Simulations

We first create artificial data in the form of two noisy arcs of a cirle. The filters found by KPCA are shown in Figure 9.1 while those found by KPFA are shown in Figure 9.2 ; the FA results were achieved after a Varimax rotation. We see that the results from KPFA are more tightly defined (more local) than those from KPCA and that it concentrates on one of the two arcs at one time.

9.4 Kernel Exploratory Projection Pursuit

9.4.1 Exploratory Projection Pursuit

Exploratory Projection Pursuit attempts to project the data onto a low dimensional subspace in order to look for structure in the projection. However not all projections reveal the data's structure equally well; so we define an index that measures how "interesting" a given projection is, and then represent the data in terms of the projections that are maximally interesting.

Friedman [?] notes that what constitutes an interesting direction is more difficult to define than what constitutes an uninteresting direction. The idea of "interestingness" is usually defined in relation to the oft-quoted observation of Diaconis and Freedman([?]) that most projections through most multi-dimensional data are almost Gaussian. This would suggest that if we wish to identify "interesting" features in data, we should look for those projections which are as non-Gaussian as possible.

Thus we require a method for moving a filter (a weight vector) so that it finds linear combinations of the data set which are as non-Gaussian as possible. Some possible measures of non-Gaussianity lie in the higher order statistics of the data: for example, the third moment measures skewness in a data set while the fourth measures the volume of the distribution found in the tails of the distribution. Since we are not interested in the mean or the variance of the distribution (one Gaussian is an uninteresting as any other Gaussian) we sphere (or whiten) the data to remove the first and second moments; we can envisage this as translating the data till its mean is over the origin and squeezing in or teasing out the data in each direction till it has equal spread in each direction. It is then with this sphered data that we look for interesting filters. We have previously given a neural implementation of Exploratory Projection Pursuit [3].

9.4.2 Kernel Exploratory Projection Pursuit

[12] have introduced the term Kernel Projection Pursuit: they argue that rather than finding the vector, $\mathbf{v}_1$ which maximises the variance of the projections in feature space

$$\mathbf{v}_1 = \arg\max_{\mathbf{v}\in V} \frac{1}{m} \sum_{i=1}^{m} (\mathbf{v}.\Phi(\mathbf{x}_i))^2 \tag{9.24}$$

we may choose any other function of the projection to maximise. Thus we may choose

$$\mathbf{v}_1 = \arg\max_{\mathbf{v}\in V} \frac{1}{m} \sum_{i=1}^{m} q(\mathbf{v}.\Phi(\mathbf{x}_i)) \tag{9.25}$$

where $q(.)$ may be a higher power or indeed we can use more general functions of all variable projections so that we have

$$\mathbf{v}_1 = \arg\max_{\mathbf{v}\in V} Q(\{\mathbf{v}.\Phi(\mathbf{x}_1), \mathbf{v}.\Phi(\mathbf{x}_2), ..., \mathbf{v}.\Phi(\mathbf{x}_m)\}) \tag{9.26}$$

where $Q(.)$ may now be any function with a finite maximum. However, there is no necessity or capability when using that single operation to perform sphering of the data which the EPP method demands. Therefore we perform a two-stage operation just as we would if we were operating in data space.

The operation of KPCA gives us principal components in feature space. Since we are already centering these Kernels, all that is required for sphering is to divide each projection by the square root of its eigenvalue and we have a set of coordinates in feature space which is sphered. Now we may choose any method to search for higher order structure in feature space. We choose to use a neural method which we have already shown to perform EPP on sphered data in data space [3]. Let $\mathbf{z}_i$ be the projection of the data point $\mathbf{x}_i$ onto feature space after the sphering has been carried out. Now $\mathbf{z}$ is fed forward through weights, W, to output neurons to give a vector. Now the activation is passed back to the originating z values as inhibition and then a non-linear function of the inputs is calculated:

$$\begin{aligned} y_i &= \sum_j w_{ij} z_j \\ z_j &\leftarrow z_j - w_{ij} y_i \\ s_i &= tanh(y_i) \end{aligned}$$

and then the new weights are calculated using simple Hebbian learning

$$\Delta w_{ij} = \beta z_j y_i$$

Note at this stage the operation of our algorithm is identical to that used previously in data space. Because we are using the sphered projections onto the eigenvectors, it is irrelevant as to whether these eigenvectors are eigenvectors in the data space or eigenvectors in the (nonlinear) feature space.

9.4.3 Simulations

We use the Iris data set to compare KPCA (Figure 9.3) and KEPP (Figure 9.4). Figure 9.3 shows the projection of the iris data set on the first two principal component directions found by Kernel PCA. We can see that one cluster has been clearly found but the other two clusters are very intertwined. In Figure 9.4 we see the projections of the same data set onto the filters of the first two Exploratory Projection Pursuit directions. One cluster has been split from the other two and in addition, the separation between the second and third clusters is very much greater than with KPCA. It is worth noting that this result was found with great repeatability which is rather unlike many EPP methods working in data space. We require more investigation of this feature.

9.5 Canonical Correlation Analysis

Canonical Correlation Analysis is a statistical technique used when we have two data sets which we believe have some underlying correlation. Consider two sets of input data; $\mathbf{x}_1$ and $\mathbf{x}_2$. Then in classical CCA, we attempt to find that linear combination of the variables which give us maximum correlation between the combinations. Let

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{w}_1 \mathbf{x}_1 = \sum_j w_{1j} x_{1j} \\ \mathbf{y}_2 &= \mathbf{w}_2 \mathbf{x}_2 = \sum_j w_{2j} x_{2j} \end{aligned}$$

where we have used $\mathbf{x}_{1j}$ as the j^{th} element of $\mathbf{x}_1$.

Then we wish to find those values of $\mathbf{w}_1$ and $\mathbf{w}_2$ which maximise the correlation between $\mathbf{y}_1$ and $\mathbf{y}_2$. If the relation between $\mathbf{y}_1$ and $\mathbf{y}_2$ is believe to be causal, we may view the process as one of finding the best predictor of the set $\mathbf{x}_2$ by the set $\mathbf{x}_1$ and similarly of finding the best predictable criterion in the set $\mathbf{x}_2$ from the set $\mathbf{x}_1$ data set.

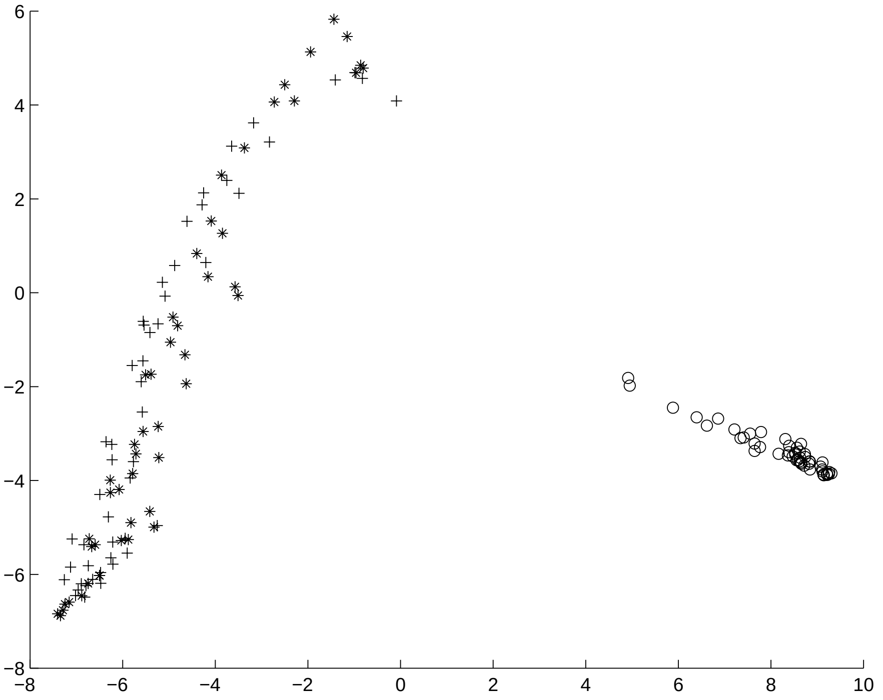


Figure 9.3: The projections on the first two kernel principal component directions of the iris data set; one cluster has clearly been identified but the other two are very intertwined.

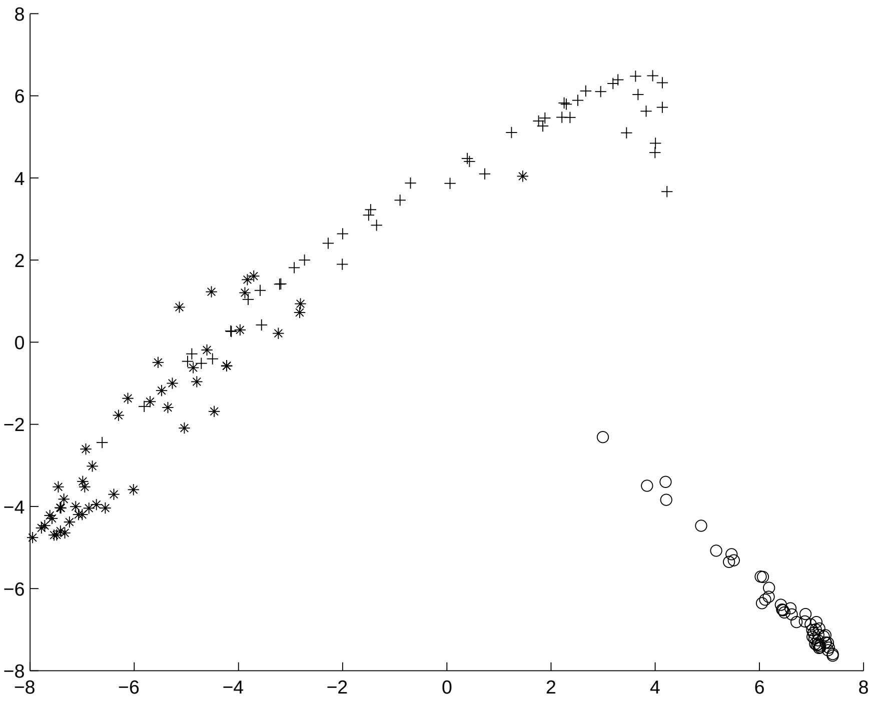


Figure 9.4: The projections of the first two kernel exploratory projection pursuit directions of the iris data set; one cluster has clearly been identified and the second and third clusters have been partially separated.

Then the standard statistical method (see [?]) lies in defining

$$\begin{aligned}
\Sigma_{11} &= E\{(\mathbf{x}_1 - \mu_1)(\mathbf{x}_1 - \mu_1)^T\} \\
\Sigma_{22} &= E\{(\mathbf{x}_2 - \mu_2)(\mathbf{x}_2 - \mu_2)^T\} \\
\Sigma_{12} &= E\{(\mathbf{x}_1 - \mu_1)(\mathbf{x}_2 - \mu_2)^T\} \\
\text{and } K &= \Sigma_{11}^{-\frac{1}{2}} \Sigma_{12} \Sigma_{22}^{-\frac{1}{2}} \qquad (9.27)
\end{aligned}$$

where T denotes the transpose of a vector. We then perform a Singular Value Decomposition of K to get

$$K = (\alpha_1, \alpha_2, ..., \alpha_k) D (\beta_1, \beta_2, ..., \beta_k)^T \qquad (9.28)$$

where α_i and β_i are the standardised eigenvectors of KK^T and $K^T K$ respectively and D is the diagonal matrix of eigenvalues.

Then the first canonical correlation vectors (those which give greatest correlation) are given by

$$\mathbf{w}_1 = \Sigma_{11}^{-\frac{1}{2}} \alpha_1 \qquad (9.29)$$

$$\mathbf{w}_2 = \Sigma_{22}^{-\frac{1}{2}} \beta_1 \qquad (9.30)$$

with subsequent canonical correlation vectors defined in terms of the subsequent eigenvectors, α_i and β_i.

9.5.1 Kernel Canonical Correlation Analysis

Consider mapping the input data to a high dimensional (perhaps infinite dimensional) feature space, F. Now,

$$\begin{aligned}
\Sigma_{11} &= E\{(\Phi(\mathbf{x}_1) - \mu_1)(\Phi(\mathbf{x}_1) - \mu_1)^T\} \\
\Sigma_{22} &= E\{(\Phi(\mathbf{x}_2) - \mu_2)(\Phi(\mathbf{x}_2) - \mu_2)^T\} \\
\Sigma_{12} &= E\{(\Phi(\mathbf{x}_1) - \mu_1)(\Phi(\mathbf{x}_2) - \mu_2)^T\}
\end{aligned}$$

where now $\mu_i = E(\Phi(\mathbf{x}_i))$ for $i = 1, 2$. Let us assume for the moment that the data has been centred in feature space (we actually will use the same trick as [10] to centre the data later). Then we define

$$\begin{aligned}
\Sigma_{11} &= E\{\Phi(\mathbf{x}_1)\Phi(\mathbf{x}_1)^T\} \\
\Sigma_{22} &= E\{\Phi(\mathbf{x}_2)\Phi(\mathbf{x}_2)^T\} \\
\Sigma_{12} &= E\{\Phi(\mathbf{x}_1)\Phi(\mathbf{x}_2)^T\}
\end{aligned}$$

and we wish to find those values $\mathbf{w}_1$ and $\mathbf{w}_2$ which will maximise $\mathbf{w}_1^T \Sigma_{12} \mathbf{w}_2$ subject to the constraints $\mathbf{w}_1^T \Sigma_{11} \mathbf{w}_1 = 1$ and $\mathbf{w}_2^T \Sigma_{22} \mathbf{w}_2 = 1$.

In practise we will approximate Σ_{12} with $\frac{1}{n}\sum_i \Phi(\mathbf{x}_{1i})\Phi(\mathbf{x}_{2i})$, the sample average.

At this stage we can see the similarity with our nonlinear CCA: if we consider an instantaneous hill-climbing algorithm, we would derive precisely our NLCCA algorithm for the particular nonlinearity involved.

Now $\mathbf{w}_1$ and $\mathbf{w}_2$ exist in the feature space which is spanned by $\{\Phi(\mathbf{x}_{11}), \Phi(\mathbf{x}_{12}), ..., \Phi(\mathbf{x}_{1n}), \Phi(\mathbf{x}_{21}), ..., \Phi(\mathbf{x}_{2n})\}$ and therefore can be expressed as

$$\begin{aligned}
\mathbf{w}_1 &= \sum_{i=1}^{n} \alpha_{1i}\Phi(\mathbf{x}_{1i}) + \sum_{i=1}^{n} \alpha_{2i}\Phi(\mathbf{x}_{2i}) \\
\mathbf{w}_2 &= \sum_{i=1}^{n} \beta_{1i}\Phi(\mathbf{x}_{1i}) + \sum_{i=1}^{n} \beta_{2i}\Phi(\mathbf{x}_{2i})
\end{aligned}$$

With some abuse of the notation we will use $\mathbf{x}_i$ to be the i^{th} instance from the set of data i.e. from either the set of values of $\mathbf{x}_1$ or from those of $\mathbf{x}_2$ and write

$$\begin{aligned}\mathbf{w}_1 &= \sum_{i=1}^{2n} \alpha_i \Phi(\mathbf{x}_i)\\ \mathbf{w}_2 &= \sum_{i=1}^{2n} \beta_i \Phi(\mathbf{x}_i)\end{aligned}$$

Therefore substituting this in the criteria we wish to optimise, we get

$$(\mathbf{w}_1^T \Sigma_{12} \mathbf{w}_2) = \frac{1}{n} \sum_{k,i} \alpha_k . \Phi^T(\mathbf{x}_k)\Phi(\mathbf{x}_{1i}) \sum_l \beta_l \Phi^T(\mathbf{x}_{2i})\Phi(\mathbf{x}_l) \tag{9.31}$$

where the sums over i are to find the sample means over the data set. Similarly with the constraints and so

$$\begin{aligned}\mathbf{w}_1^T \Sigma_{11} \mathbf{w}_1 &= \frac{1}{n} \sum_{k,i} \alpha_k . \Phi^T(\mathbf{x}_k)\Phi(\mathbf{x}_{1i}) . \sum_l \alpha_l \Phi^T(\mathbf{x}_{1i})\Phi(\mathbf{x}_l)\\ \mathbf{w}_2^T \Sigma_{22} \mathbf{w}_2 &= \frac{1}{n} \sum_{k,i} \beta_k . \Phi^T(\mathbf{x}_k)\Phi(\mathbf{x}_{2i}) . \sum_l \beta_l \Phi^T(\mathbf{x}_{2i})\Phi(\mathbf{x}_l)\end{aligned}$$

Using $(K_1)_{ij} = \Phi^T(\mathbf{x}_i)\Phi(\mathbf{x}_{1j})$ and $(K_2)_{ij} = \Phi^T(\mathbf{x}_i)\Phi(\mathbf{x}_{2j})$ we then have that we require to maximise $\alpha^T K_1 K_2^T \beta$ subject to the constraints $\alpha^T K_1 K_1^T \alpha = 1$ and $\beta^T K_2 K_2^T \beta = 1$. Therefore if we define $\Sigma_{11} = K_1 K_1^T$, $\Sigma_{22} = K_2 K_2^T$ and $\Sigma_{12} = K_1 K_2^T$ we solve the problem in the usual way: by forming matrix $K = \Sigma_{11}^{-\frac{1}{2}} \Sigma_{12} \Sigma_{22}^{-\frac{1}{2}}$ and performing a singular value decomposition on it as before to get

$$K = (\gamma_1, \gamma_2, ..., \gamma_k) D(\theta_1, \theta_2, ..., \theta_k)^T \tag{9.32}$$

where γ_i and θ_i are again the standardised eigenvectors of KK^T and K^TK respectively and D is the diagonal matrix of eigenvalues [1]

Then the first canonical correlation vectors in feature space are given by

$$\alpha_1 = \Sigma_{11}^{-\frac{1}{2}} \gamma_1 \tag{9.33}$$

$$\beta_1 = \Sigma_{22}^{-\frac{1}{2}} \theta_1 \tag{9.34}$$

with subsequent canonical correlation vectors defined in terms of the subsequent eigenvectors, γ_i and θ_i.

Now for any new values $\mathbf{x}_1$, we may calculate

$$\mathbf{w}_1 . \Phi(\mathbf{x}_1) = \sum_i \alpha_i \Phi(\mathbf{x}_i) \Phi(\mathbf{x}_1) = \sum_i \alpha_i K_1(\mathbf{x}_i, \mathbf{x}_1) \tag{9.35}$$

which then requires to be centered as before. We see that we are again performing a dot product in feature space (it is actually calculated in the subspace formed from projections of $\mathbf{x}_i$).

The optimal weight vectors are vectors in a feature space which we may never determine. We are simply going to calculate the appropriate matrices using the kernel trick - e.g. we may use Gaussian kernels so that

$$K_1(\mathbf{x}_{1i}, \mathbf{x}_{1j}) = exp(-(\mathbf{x}_{1i} - \mathbf{x}_{1j})^2) \tag{9.36}$$

which gives us a means of calculating K_{11} without ever having had to calculate $\Phi(\mathbf{x}_{1i})$ or $\Phi(\mathbf{x}_{1j})$ explicitly.

[1]This optimisation is applicable for all symmetric matrices (Theorem A.9.2, [?]).

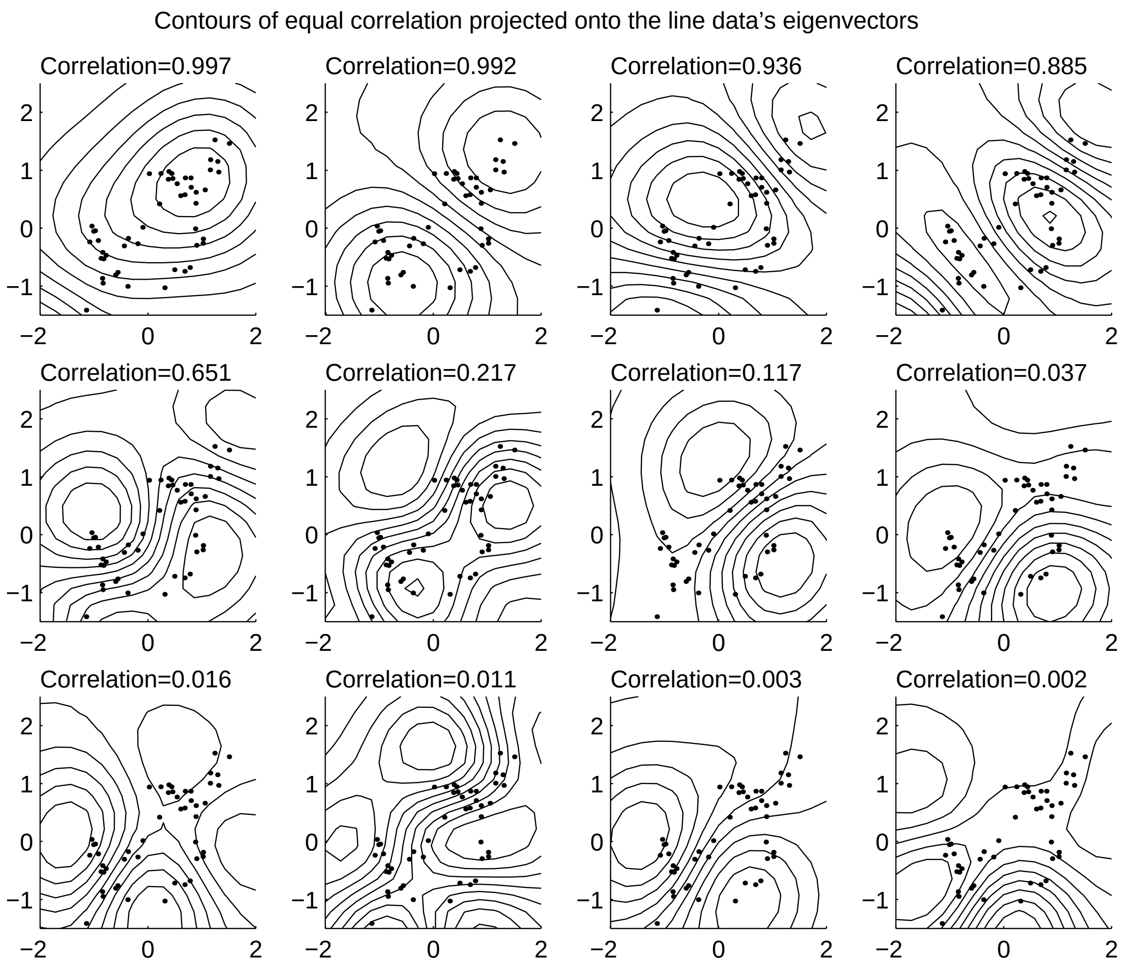


Figure 9.5: The contours of equal correlation when projected onto the first twelve cca directions of the first data set (the circle data)

9.5.2 Simulations

We have previously [5] tested a neural implementation of CCA on artificial data comprising a line and a circle. The best linear correlation of the data set was 0.623 while the nonlinear neural method was able to get a correlation of 0.865. We show in Figures 9.5 and 9.6 two simulations with noisy versions of the data: the first shows the contours of the first twelve directions of maximum correlation of the circle data; the second shows the contours of maximum correlation of the line data. We see first that we have more than two non-zero correlation projections which is impossible with a linear method. We also see that we can achieve remarkably high correlations with this data set.

We have to be a little careful with these results since it is perfectly possible to create perfect correlations using radial kernels by simply setting the width of these kernels to ∞! Perhaps a better example is shown in Figure 9.7 in which we show the results of this technique on a set of data discussed in ([?],p290); it comprises 88 students' marks on 5 module exams. The exam results can be partitioned into two data sets: two exams were given as close book exams (C) while

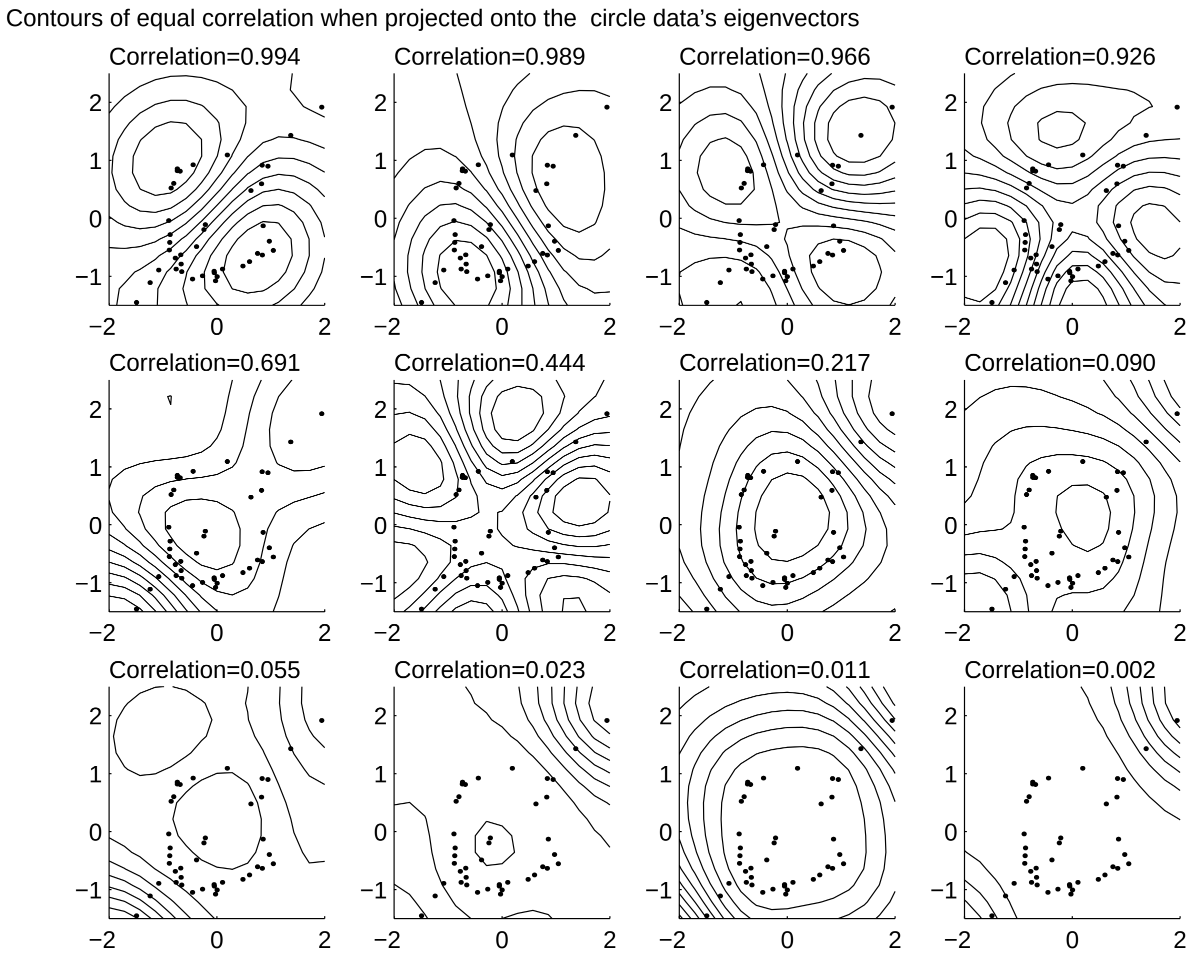


Figure 9.6: The contours of equal correlation when projected onto the first twelve cca directions of the second data set (the line data)

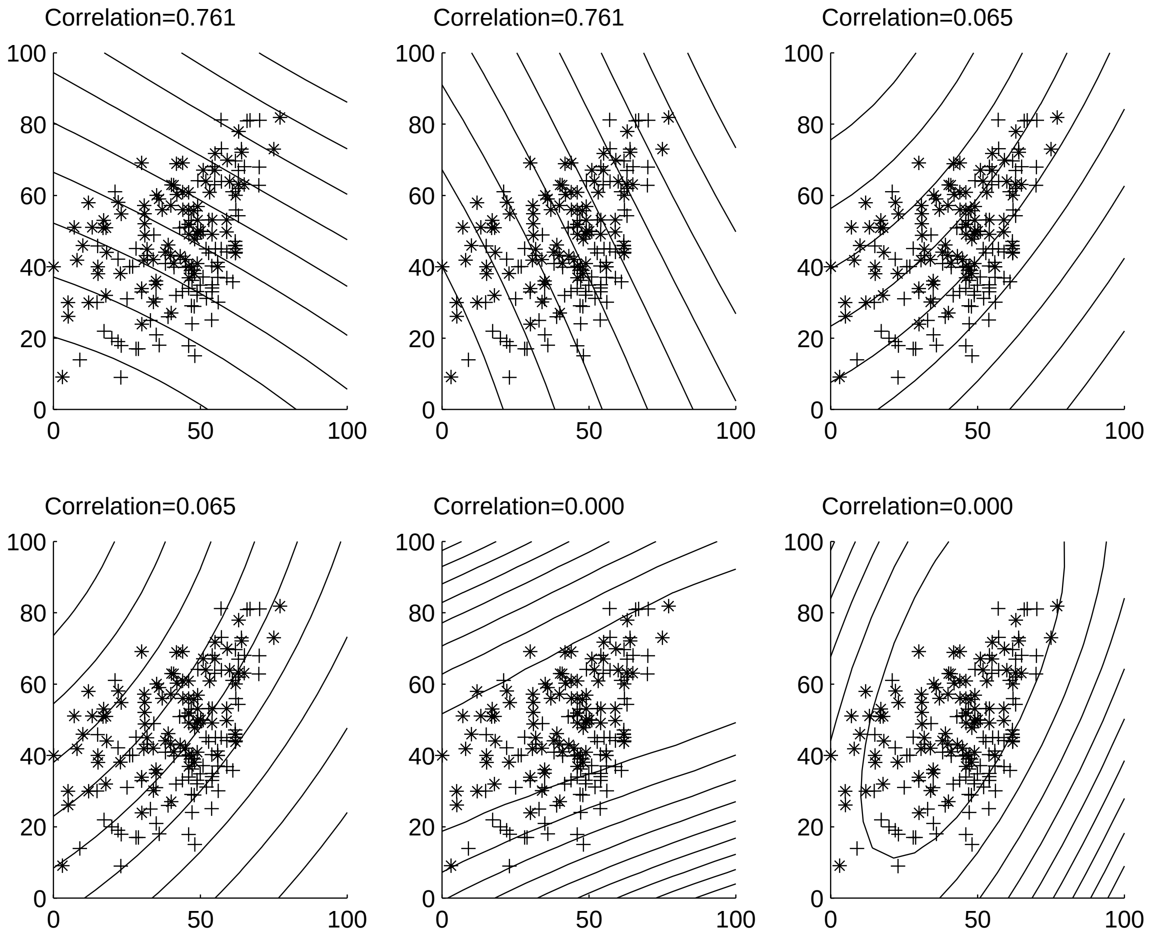


Figure 9.7: The kernel canonical correlation directions found using radial kernels. The contour lines are lines of equal correlation. Each pair of diagrams shows the equal correlation contours from the perspective of one of the data sets.

the other three were opened book exams (O). The exams were on the subjects of Mechanics(C), Vectors(C), Algebra(O), Analysis(O), and Statistics(O). We thus split the five variables (exam marks) into two sets-the closed-book exams (x_{11}, x_{12}) and the opened-book exams (x_{21}, x_{22}, x_{23}). One possible quantity of interest here is how highly a student's ability on closed-book exams is correlated with his ability on open-book exams. Alternatively, one might try to use the open-book exam results to predict the closed-book results (or vice versa).

The results in Figure 9.7 are clearly very good but come with two caveats:

1. The method requires a dot product between members of the data set $\mathbf{x}_1$ and $\mathbf{x}_2$ and therefore the vectors must be of the same length. Therefore in the exam data, we must discard one set of exam marks.

2. The method requires a matrix inversion and the data sets may be such that one data point may be repeated (or almost) leading to a singularity or badly conditioned matrices. One solution is to add noise to the data set; this is effective in the exam data set, is a nice solution if we were to consider biological information processors but need not always work.

An alternative is to add μI, where I is the identity matrix to Σ_{11} and Σ_{22} - a method which was also used in [?]. This gives robust and reliable solutions.

9.6 Conclusion

We have reviewed the technique of Kernel Principal Component Analysis and extended the use of kernels to three other methods of unsupervised investigation of structure in data:

1. Principal Factor Analysis
2. Exploratory Projection Pursuit
3. Canonical Correlation Analysis

Each of these methods may be expressed in terms of a dot product. We have restricted our examples to simulations using Gaussian kernels but there are many other possible kernels (both radial and otherwise). It is an open research question as to which kernel is optimal in different situations.

Bibliography

[1] D. Charles and C. Fyfe. Modelling multiple cause structure using rectification constraints. *Network: Computation in Neural Systems*, 1998.

[2] N Christiani and J Shawe-Taylor. *An Introduction to Support Vector Machines and other kernel-based learning methods.* Cambridge University Press, 2000.

[3] C. Fyfe and R. Baddeley. Non-linear data structure extraction using simple hebbian networks. *Biological Cybernetics*, 72(6):533–541, 1995.

[4] S. Mika, B. Scholkopf, A. Smola, K.-R. Muller, M. Scholz, and G. Ratsch. Kernel pca and de-noising in feature spaces. In *Advances in Neural Processing Systems, 11*, 1999.

[5] P.L.Lai and C. Fyfe. A neural network implementation of canonical correlation analysis. *Neural Networks*, 12(10):1391–1397, Dec. 1999.

[6] S. Romdhani, S. Gong, and A. Psarrou. A multi-view nonlinear active shape model using kernel pca. In *BMVC99*, 1999.

[7] B. Scholkopf, S. Mika, C. Burges, P. Knirsch, K.-R. Muller, G. Ratsch, and A. J. Smola. Input space vs feature space in kernel-based methods. *IEEE Transactions on Neural Networks*, 10:1000–1017, 1999.

[8] B. Scholkopf, S. Mika, A. Smola, G. Ratsch, and K.-R. Muller. Kernel pca pattern reconstruction via approximate pre-images. In L. Niklasson M. Boden R. Ziemke, editor, *Proceedings of 8th International Conference on Artificial Neural Networks*, pages 147–152. Springer Verlag, 1998.

[9] B. Scholkopf, A. Smola, and K.-R. Muller. Nonlinear component analysis as a kernel eigenvalue problem. Technical Report 44, Max Planck Institut fur biologische Kybernetik, Dec 1996.

[10] B. Scholkopf, A. Smola, and K.-R. Muller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.

[11] B. Scholkopf, A. Smola, and K.-R. Muller. *Support Vector Machines*, chapter Kernel Principal Component Analysis, pages 327–370. 1999.

[12] A. J. Smola, O. L. Mangasarian, and B Scholkopf. Sparse kernel feature analysis. Technical Report 99-04, University of Wiscosin Madison, 1999.

[13] A. J. Smola, S. Mika, B. Scholkopf, and R. C. Williamson. Regularized principal maniforlds. *Machine Learning*, pages 1–28, 2000. (submitted).

[14] A. J. Smola and B. Scholkopf. A tutorial on support vector regression. Technical Report NC2-TR-1998-030, NeuroCOLT2 Technical Report Series, Oct. 1998.

[15] R. J. Vanderbei. Loqo: An interior point code for quadratic programming. Technical Report SOR-94-15, Princeton University, Sept. 1998.

[16] V Vapnik. *The nature of statistical learning theory.* Springer Verlag, New York, 1995.

Appendix A

Linear Algebra

A.1 Vectors

We can show a two-dimensional vector $\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$ in the (x,y)-plane as in Figure A.1. We can write this as $\mathbf{a} = (a_1, a_2)^T$ where the T stands for the transpose of the vector(see below). This system can be extended to

- 3D vectors so that $\mathbf{a} = (a_1, a_2, a_3)^T$ where a_3 can be thought to be the projection of the vector on the z-axis.
- 4D vectors so that $\mathbf{a} = (a_1, a_2, a_3, a_4)^T$. Now it is difficult to visualise $\mathbf{a}$.
- n-dimensional vectors $\mathbf{a} = (a_1, a_2, ..., a_n)^T$. Now no one can visualise the vectors!

A.1.1 Same direction vectors

In 2D, the vector $\mathbf{a} = (a_1, a_2)^T$ and the vector $2\mathbf{a} = (2a_1, 2a_2)^T$ are parallel - and the second vector is twice the length of the first. In general, the vector $\mathbf{a} = (a_1, a_2, \cdots, a_n)^T$ and the vector $k\mathbf{a} = (ka_1, ka_2, \cdots, ka_n)^T$ are parallel.

A.1.2 Addition of vectors

If $\mathbf{a} = (a_1, a_2)^T$ and $\mathbf{b} = (b_1, b_2)^T$ then we may write $\mathbf{a} + \mathbf{b} = (a_1 + b_1, a_2 + b_2)^T$. In general, if

$$\begin{aligned} \mathbf{a} &= (a_1, a_2, ..., a_n)^T \text{ and} \\ \mathbf{b} &= (b_1, b_2, ..., b_n)^T \text{ then} \\ \mathbf{a} + \mathbf{b} &= (a_1 + b_1, a_2 + b_2, ..., a_n + b_n)^T \end{aligned}$$

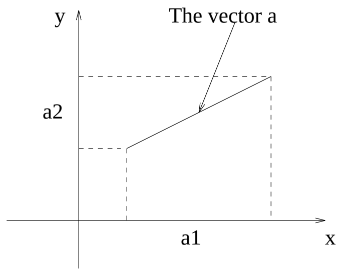


Figure A.1: A two dimensional vector

A.1.3 Length of a vector

In two dimensions we know that the length of the vector, **a** can be found by

$$|\mathbf{a}| = \sqrt{a_1^2 + a_2^2} \tag{A.1}$$

This is extendable to

$$\begin{aligned} |\mathbf{a}| &= \sqrt{a_1^2 + a_2^2 + a_3^2} \text{ in 3 dimensions} \\ |\mathbf{a}| &= \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2} \text{ in n dimensions} \end{aligned}$$

To **normalise** a vector (i.e. to give it length 1 but keep its direction unchanged), divide by its length to give $\frac{\mathbf{a}}{|\mathbf{a}|}$. In ANN books, you will often meet $\| \mathbf{a} \|$. This is identical to $|\mathbf{a}|$.

A.1.4 The Scalar Product of 2 Vectors

In 2D the scalar product of two vectors, **a** and **b** is given by

$$\mathbf{a}.\mathbf{b} = a_1.b_1 + a_2.b_2 \tag{A.2}$$

This can be extended to 3-dimensional vectors as

$$\mathbf{a}.\mathbf{b} = a_1.b_1 + a_2.b_2 + a_3.b_3 \tag{A.3}$$

and, in general,

$$\mathbf{a}.\mathbf{b} = a_1.b_1 + a_2.b_2 + \cdots + a_n.b_n \tag{A.4}$$

The scalar product of **a** and **b** can be viewed as the projection of **a** onto **b**.

A.1.5 The direction between 2 vectors

In 2D the direction between two vectors, **a** and **b** is given by

$$\begin{aligned} \cos\theta &= \frac{\mathbf{a}.\mathbf{b}}{|\mathbf{a}|.|\mathbf{b}|} \\ &= \frac{a_1.b_1 + a_2.b_2 + \cdots + a_n.b_n}{\sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}.\sqrt{b_1^2 + b_2^2 + \cdots + b_n^2}} \end{aligned}$$

A.1.6 Linear Dependence

Consider vector $\mathbf{a} = (1,3)^T$, $\mathbf{b} = (1,-1)^T$, and $\mathbf{c} = (5,4)^T$, then we can write
$\mathbf{c} = 2.25\ \mathbf{a} + 2.75\ \mathbf{b}$
We say that **c** is a linear combination of **a** and **b** or that the set of vectors **a**, **b** and **c** are in the space spanned by**a** and **b**. We call **a** and **b** the basis of the space just as the X-axis and the Y-axis are the basis of the usual Cartesian plane. Indeed we can note that the unit vectors $(1,0)^T$ and $(0,1)^T$ in these directions is the usual basis of that plane.

If we have a set of vectors which cannot be shown to have the property that one of them is a linear combination of the others, the set is said to exhibit linear independence. Such a set can always form a basis of the space in which the vectors lie.

A.1.7 Neural Networks

We often consider a weight vector **w** and a set of inputs **x** and are interested in the weighted sum of the inputs

$$Act = \sum_i w_i.x_i = \mathbf{w}^T\mathbf{x} = \mathbf{w}.\mathbf{x} \tag{A.5}$$

A.2 Matrices

A matrix is an array (of numbers) such as

$$A = \begin{bmatrix} 2 & 3.5 & 4 & 0.1 \\ 2 & 1.2 & 7 & 9 \\ 9 & 0.6 & 5.99 & 1 \end{bmatrix} \tag{A.6}$$

A.2.1 Transpose

The transpose of a matrix, A, is that matrix whose columns are the rows of the original matrix and is usually written A^T. If A is as above, then

$$A^T = \begin{bmatrix} 2 & 2 & 9 \\ 3.5 & 1.2 & 0.6 \\ 4 & 7 & 5.99 \\ 0.1 & 9 & 1 \end{bmatrix} \tag{A.7}$$

Clearly if the array A is m*n then A^T is n*m.

A.2.2 Addition

To add matrices, add corresponding entries. It follows that the matrices must have the same order. Thus if A is as above and

$$B = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 4 & 3 & 1 & 7 \\ 2 & 1 & 0 & 0 \end{bmatrix} \tag{A.8}$$

then A + B is the matrix

$$A + B = \begin{bmatrix} 3 & 3.5 & 4 & 1.1 \\ 6 & 4.2 & 8 & 16 \\ 11 & 1.6 & 5.99 & 1 \end{bmatrix} \tag{A.9}$$

A.2.3 Multiplication by a scalar

If again,

$$A = \begin{bmatrix} 2 & 3.5 & 4 & 0.1 \\ 2 & 1.2 & 7 & 9 \\ 9 & 0.6 & 5.99 & 1 \end{bmatrix} \tag{A.10}$$

then 3A is that matrix each of whose elements is multiplied by 3

$$3A = \begin{bmatrix} 6 & 10.5 & 12 & 0.3 \\ 6 & 3.6 & 21 & 27 \\ 27 & 1.8 & 17.97 & 3 \end{bmatrix} \tag{A.11}$$

A.2.4 Multiplication of Matrices

We multiply the elements of the rows of the first matrix by the elements in the columns of the second. Let C be the matrix

$$C = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{A.12}$$

Then CB is the matrix

$$CB = \begin{bmatrix} 3 & 1 & 0 & 1 \\ 6 & 4 & 1 & 7 \end{bmatrix} \tag{A.13}$$

A.2.5 Identity

Additive Identity

The additive m*n identity matrix is that m*n matrix each of whose entries is 0.

Thus $A + 0_{m*n} = A = 0_{m*n} + A, \forall A$ which are m*n matrices.

Multiplicative Identity

The multiplicative identity is usually denoted by the letter I and is such that $A * I = I * A = A, \forall A$ such that the multiplication is possible.

Then

$$\begin{aligned} I_{2*2} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ I_{3*3} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{etc.} \end{aligned}$$

A.2.6 Inverse

The inverse of a matrix is that matrix which when the operation is applied to the matrix and its inverse, the result is the identity matrix.

Additive Inverse

Therefore we are looking for the matix -A such that A+(-A) = 0. Clearly if $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ then $-A = \begin{bmatrix} -a & -b \\ -c & -d \end{bmatrix}$

Multiplicative Inverse

We are now looking for that matrix A^{-1} such that $AA^{-1} = A^{-1}A = I$ *when this matrix exists.*

A.3 Eigenvalues and Eigenvectors

A matrix **A** has an eigenvector **x** with a corresponding eigenvalue λ if

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

In other words, multiplying the vector **x** or any of its multiples by **A** is equivalent to multiplying the whole vector by a scalar λ. Thus the direction of **x** is unchanged - only its magnitude is affected.

Appendix B

Calculus

B.1 Introduction

Consider a function y = f(x). Then $\frac{dy}{dx}$, the derivative of y with respect to x, gives the rate of change of y with respect to x. Then as we see in Figure B.1, the ratio $\frac{\Delta y}{\Delta x}$ is the tangent of the angle which the triangle makes with the x-axis i.e. gives a value of the average slope of the curve at this point. Now if we take the $\lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \frac{dy}{dx}$ we get the gradient of the curve.

If $|\frac{\Delta y}{\Delta x}|$ is large we have a steep curve; if $|\frac{\Delta y}{\Delta x}|$ is small we have a gently sloping curve.

Since we are often interested in change in weights with respect to time we often use $\frac{dw}{dt}$. In a simulation, we cannot change weights in an infinitesimally small instance of time and so we use the notation $\Delta w \propto \frac{dw}{dt}$ for the change in w. Now we are often using systems of learning which are changing weights according to the error descent procedure $\Delta w \propto -\frac{dE}{dw}$. In other words, (see Figure B.2) if $\frac{dE}{dw}$ is large and negative we will be making large increases to the value of w while if $\frac{dE}{dw}$ is large and positive we will be making large decreases to w. If $\frac{dE}{dw}$ is small we will only be making a small change to w. Notice that at the minimum point $\frac{dE}{dw} = 0$.

B.1.1 Partial Derivatives

Often we have a variable which is a function of two or more other variables. For example our (instantaneous) error function can be thought of as a function both of the weights, w, and of the inputs, x. To show the derivative of E with respect to w, we use $\frac{\partial E}{\partial w}$ which should be interpreted as the rate of change of E with respect to w when x, the input, is held constant. We can think of this as a mountain surface with grid lines on it. Then $\frac{\partial E}{\partial w}$ is the rate of change in one direction

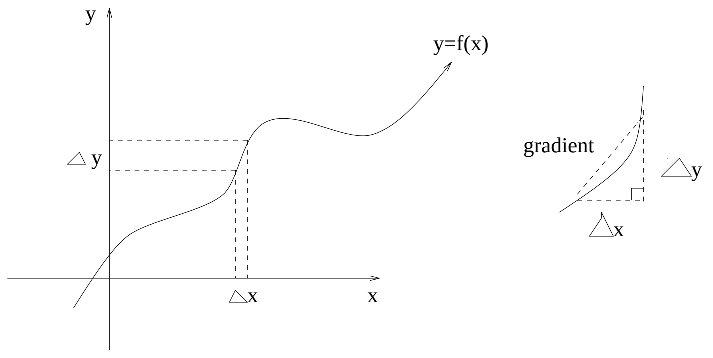


Figure B.1: The ratio $\frac{\Delta y}{\Delta x}$ is the tangent of the angle which the triangle makes with the x-axis i.e. gives the average slope of the curve at this point

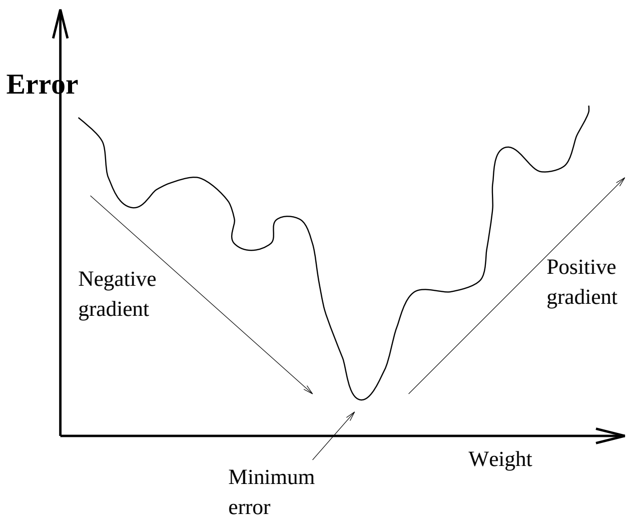


Figure B.2: A schematic diagram showing error descent. In the negative gradient section, we wish to increase the weight; in the positive gradient section, we wish to decrease the weight

while the other (orthogonal) direction is kept constant.

B.1.2 Second Derivatives

Now let $\mathbf{g} = \frac{\partial E}{\partial w}$, the rate of change of E with respect to $\mathbf{w}$. We may be interested in how quickly $\mathbf{g}$ changes as we change $\mathbf{w}$. To find this we would calculate $\frac{\partial \mathbf{g}}{\partial w}$; now note that this gives us the rate of change of the derivative of E with respect to $\mathbf{w}$ (or the rate of change of the rate of change of E with respect to $\mathbf{w}$.

For the error descent methods, we are often interested in regions of the error space where the rate of descent of the error remains constant over a region. To do so we shall be interested in this value, $\frac{\partial \mathbf{g}}{\partial w}$. To emphasise that this may be found by differntiating E with respect to $\mathbf{w}$ and then differentiating it again, we will write this rate as $\frac{\partial^2 E}{\partial w^2}$. This is its second derivative.

Appendix C

Backpropagation Derivation

We must first note that our activation functions will be non-linear in this chapter: if we were to be using linear activation functions, our output would be a linear combination of linear combinations of the inputs i.e. would simply be linear combinations of the inputs and so we would gain nothing by using a three layer net rather than a two layer net.

As before, consider a particular input pattern, $\mathbf{x}^P$, we have an output o^P and target t^P. Now however we will use a non-linear activation function, $f()$. $o_i = f(Act_i) = f(\sum_j w_{ij} o_j)$ where we have taken any threshold into the weights as before. Notice that the o_j represents the outputs of neurons in the preceeding layer. Thus if the equation describes the firing of a neuron in the (first) hidden layer, we revert to our previous definition where $o_i = f(Act_i) = f(\sum_j w_{ij} x_j)$ while if we wish to calculate the firing in an output neuron the o_j will represent the firing of hidden layer neurons. However now f() must be a differentiable function (unlike the perceptron) and a non-linear function (unlike the Adaline). Now we still wish to minimise the sum of squared errors,

$$E = \sum_P E^P = \frac{1}{2} \sum_P (t^P - o^P)^2 \tag{C.1}$$

at the outputs. To do so, we find the gradient of the error with respect to the weights and move the weights in the opposite direction. Formally, $\Delta_P w_{ij} = -\gamma \frac{\partial E^P}{\partial w_{ij}}$.

Now we have, for all neurons,

$$\frac{\partial E^P}{\partial w_{ij}} = \frac{\partial E^P}{\partial Act_i^P} . \frac{\partial Act_i^P}{\partial w_{ij}} \tag{C.2}$$

and $\frac{\partial Act_i^P}{\partial w_{ij}} = o_j$. Therefore if we define $\delta_i^P = -\frac{\partial E^P}{\partial Act_i^P}$ we get an update rule of

$$\Delta_P w_{ij} = \gamma . \delta_i^P . o_j^P \tag{C.3}$$

Note how like this rule is to that developed in the previous chapter (where the last o is replaced by the input vector, $\mathbf{x}$). However, we still have to consider what values of δ are appropriate for individual neurons. We have

$$\delta_i^P = -\frac{\partial E^P}{\partial Act_i^P} = -\frac{\partial E^P}{\partial o_i^P} . \frac{\partial o_i^P}{\partial Act_i^P} \tag{C.4}$$

for all neurons. Now, for all output neurons, $\frac{\partial E^P}{\partial o^P} = -(t^P - o^P)$, and $\frac{\partial o_i^P}{\partial Act_i^P} = f'(Act_i^P)$. This explains the requirement to have an activation function which is differentiable. Thus for output neurons we get the value $\delta_i^P = (t_i^P - o_i^P) f'(Act_i^P)$.

However, if the neuron is a hidden neuron, we must calculate the responsibility of that neuron's weights to the final error. To do this we take the error at the output neurons and propagate this

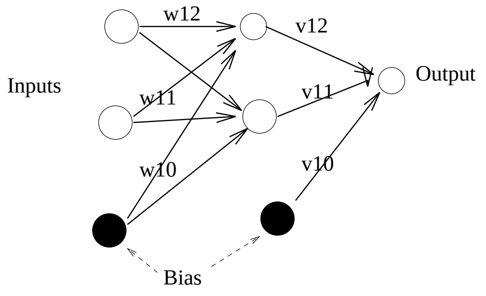


Figure C.1: The net which will be used for the solution of the XOR problem using backpropagation

backward through the current weights (the very same weights which were used to propagate the activation forward). Consider a network with N output neurons and H hidden neurons. We use a chain rule to calculate the effect on unit i in the hidden layer:

$$\frac{\partial E^P}{\partial o_i^P} = \sum_{j=1}^{N} \frac{\partial E^P}{\partial Act_j^P}.\frac{\partial Act_j^P}{\partial o_i^P} = \sum_{j=1}^{N} \frac{\partial E^P}{\partial Act_j^P}.\frac{\partial}{\partial o_i^P}\sum_{k=1}^{H} w_{jk} o_k^P = \sum_{j=1}^{N} \frac{\partial E^P}{\partial Act_j^P}.w_{ji} = -\sum_{j=1}^{N} \delta_j^P.w_{ji} \quad \text{(C.5)}$$

Note that the terms $\frac{\partial E^P}{\partial Act_j^P}$ represent the effect of change on the error from the change in activation in the output neurons. On substitution, we get

$$\delta_i^P = -\frac{\partial E^P}{\partial Act_j^P} = -\frac{\partial E^P}{\partial o_i^P}.\frac{\partial o_i^P}{\partial Act_j^P} = \sum_{j=1}^{N} \delta_j^P w_{ji}.f'(Act_i^P) \quad \text{(C.6)}$$

This may be thought of as assigning the error term in the hidden layer proportional to the hidden neuron's contribution to the final error as seen in the output layer.

C.1 The XOR problem

We can use the net shown in Figure C.1 to solve the XOR problem. The procedure is

Initialisation .

- Initialise the W-weights and V-weights to small random numbers.
- Initialise the learning rate, η to a small value e.g. 0.001.
- Choose the activation function e.g. tanh().

Select Pattern It will be one of only 4 patterns for this problem. Note that the pattern chosen determines not only the inputs but also the target pattern.

Feedforward to the hidden units first, labelled 1 and 2.

$$\begin{aligned} act_1 &= w_{10} + w_{11}x_1 + w_{12}x_2 \\ act_2 &= w_{20} + w_{21}x_1 + w_{22}x_2 \\ o_1 &= tanh(act_1) \\ o_2 &= tanh(act_2) \end{aligned}$$

Now feedforward to the output unit which we will label 3

$$\begin{aligned} act_3 &= v_{10} + v_{11}o_1 + v_{12}o_2 \\ o_3 &= tanh(act_3) \end{aligned}$$

Feedback errors calculate error at output

$$\delta_3 = (t - o_3) * f'(o_3) = (t - o_3)(1 - o_3^2)$$

and feedback error to hidden neurons

$$\begin{aligned}\delta_1 &= \delta_3 v_{11} f'(o_1) = \delta_3 v_{11}(1 - o_1^2) \\ \delta_2 &= \delta_3 v_{12} f'(o_2) = \delta_3 v_{12}(1 - o_2^2)\end{aligned}$$

Change weights

$$\begin{aligned}\Delta v_{11} &= \eta.\delta_3.o_1 \\ \Delta v_{12} &= \eta.\delta_3.o_2 \\ \Delta v_{10} &= \eta.\delta_3.1 \\ \Delta w_{11} &= \eta.\delta_1.x_1 \\ \Delta w_{12} &= \eta.\delta_1.x_2 \\ \Delta w_{10} &= \eta.\delta_1.1 \\ \Delta w_{21} &= \eta.\delta_2.x_1 \\ \Delta w_{22} &= \eta.\delta_2.x_2 \\ \Delta w_{20} &= \eta.\delta_2.1\end{aligned}$$

Go back to Select Pattern