Artificial Neural Networks

Colin Fyfe,
Department of Computing and Information Systems,
The University of Paisley.
Room L117
Phone 848 3305.

Edition 1.1,
1996

Contents

1 Introduction **5**
- 1.1 Objectives . . . 5
- 1.2 Intelligence . . . 5
- 1.3 Artificial Neural Networks . . . 6
- 1.4 Biological and Silicon Neurons . . . 6
- 1.5 Learning in Artificial Neural Networks . . . 7
- 1.6 Typical Problem Areas . . . 9
- 1.7 A short history of ANNs . . . 9
- 1.8 Genetic Algorithms . . . 10
- 1.9 Tutorial . . . 11
  - 1.9.1 Worked Example . . . 11
  - 1.9.2 Exercises . . . 13

2 Associative Memory **15**
- 2.1 Objectives . . . 15
- 2.2 Memory . . . 15
- 2.3 Heteroassociation . . . 16
- 2.4 Creating the Memory Matrix . . . 17
- 2.5 Recall and Crosstalk . . . 17
  - 2.5.1 Example . . . 18
- 2.6 Bidirectional Autoassociative Memories . . . 20
  - 2.6.1 An Example . . . 20
  - 2.6.2 Lyapunov's Method . . . 22
- 2.7 Conclusion . . . 23
- 2.8 Exercises . . . 23

3 Simple Supervised Learning **25**
- 3.1 Objectives . . . 25
- 3.2 Introduction . . . 25
- 3.3 The perceptron . . . 25
- 3.4 The perceptron learning rule . . . 27
- 3.5 An example problem solvable by a perceptron . . . 27
  - 3.5.1 And now the bad news . . . 28
- 3.6 The Delta Rule . . . 29
  - 3.6.1 The learning rule: error descent . . . 29
  - 3.6.2 Non-linear Neurons . . . 30
  - 3.6.3 XOR revisited . . . 31
  - 3.6.4 Nuts Revisited . . . 32
  - 3.6.5 Stochastic Neurons . . . 33
- 3.7 Multiple Discrimination . . . 34
- 3.8 Exercises . . . 34

4 The Multilayer Perceptron: backprop **35**
4.1 Objectives . . . 35
4.2 Introduction . . . 35
4.3 The Backpropagation Algorithm . . . 36
4.3.1 The XOR problem . . . 36
4.4 Backpropagation Derivation . . . 37
4.5 Issues in Backpropagation . . . 38
4.5.1 Batch vs On-line Learning . . . 38
4.5.2 Activation Functions . . . 39
4.5.3 Initialisation of the weights . . . 39
4.5.4 Momentum and Speed of Convergence . . . 39
4.5.5 Stopping Criteria . . . 39
4.5.6 Local Minima . . . 40
4.5.7 Weight Decay and Generalisation . . . 40
4.5.8 Adaptive parameters . . . 41
4.5.9 The number of hidden neurons . . . 41
4.6 Application - A classification problem . . . 41
4.6.1 Theory . . . 41
4.6.2 An Example . . . 42
4.7 Linear Discrimination . . . 42
4.7.1 Multilayered Perceptrons . . . 45
4.8 Function Approximation . . . 46
4.8.1 A Prediction Problem . . . 47
4.8.2 Practical Issues . . . 50
4.9 Data Compression . . . 52
4.10 Exercises . . . 52

5 Unsupervised learning **53**
5.1 Objectives . . . 53
5.2 Unsupervised learing . . . 53
5.3 Hebbian learning . . . 53
5.3.1 The InfoMax Principle in Linsker's Model . . . 54
5.3.2 The Stability of the Hebbian Learning Rule . . . 55
5.4 Hebbian Learning and Information Theory . . . 56
5.4.1 Quantification of Information . . . 56
5.4.2 Principal Component Analysis . . . 56
5.4.3 Weight Decay in Hebbian Learning . . . 57
5.4.4 Oja's One Neuron Model . . . 57
5.4.5 Oja's Subspace Algorithm . . . 58
5.4.6 Oja's Weighted Subspace Algorithm . . . 58
5.4.7 Sanger's Generalized Hebbian Algorithm . . . 59
5.4.8 Summary of Hebbian Learning . . . 60
5.4.9 Applications . . . 60
5.5 Anti-Hebbian Learning . . . 60
5.5.1 The Novelty Filter . . . 62
5.6 Competitive learning . . . 62
5.6.1 Simple Competitive Learning . . . 63
5.6.2 Learning Vector Quantisation . . . 64
5.6.3 ART Models . . . 64
5.6.4 The Kohonen Feature Map . . . 65
5.7 Applications . . . 69
5.8 Exercises . . . 69

6 Genetic Algorithms **71**
6.1 Objectives . . . 71
6.2 Optimisation . . . 71
6.3 Genetic Algorithms . . . 72
6.3.1 Natural evolution . . . 72
6.3.2 The Simple Genetic Algorithm . . . 72
6.3.3 The Knapsack problem . . . 73
6.3.4 Evaluation of Fitness . . . 73
6.3.5 Reproduction . . . 74
6.3.6 Crossover . . . 74
6.3.7 Mutation . . . 74
6.3.8 Inversion . . . 74
6.3.9 Selection . . . 75
6.3.10 Comparison of Operators . . . 75
6.3.11 The Schema Theorem . . . 76
6.4 A First Example . . . 77
6.4.1 Premature Convergence . . . 78
6.5 Extensions to the Simple GA . . . 78
6.5.1 Deceptive problems . . . 78
6.5.2 The Structured GA . . . 79
6.5.3 Tabu Search . . . 81
6.5.4 Population-based Incremental Learning . . . 81
6.5.5 Evolution Strategies . . . 82
6.5.6 Representation and Mutation . . . 82
6.6 Representation and Operators . . . 83
6.6.1 The Iterated Prisoner's Dilemma . . . 83
6.6.2 The Traveling Salesman Problem . . . 84
6.7 Applications . . . 89
6.8 Exercises . . . 90

7 Recurrent Networks **93**
7.1 Objectives . . . 93
7.2 The Hopfield Network . . . 93
7.2.1 Activation in the Hopfield Network . . . 94
7.2.2 Storage Phase . . . 95
7.2.3 Retrieval Phase . . . 95
7.2.4 An Example . . . 95
7.2.5 Energy Minimisation . . . 97
7.3 Simulated Annealing . . . 97
7.3.1 The Metropolis Algorithm . . . 98
7.4 The Boltzmann Machine . . . 98
7.4.1 The Clamped Phase . . . 99
7.4.2 Free-running phase . . . 100
7.4.3 Learning phase . . . 100
7.4.4 Supervised learning . . . 100
7.5 Applications . . . 101
7.6 Cellular Automata . . . 101
7.6.1 Description of the Parameters . . . 103
7.6.2 Transients and cycles . . . 104
7.7 The Game of Life . . . 105
7.8 Artificial Life . . . 106
7.8.1 Emergent Behaviour . . . 106
7.9 Complex Behaviour in Robots . . . 107

7.10 Application - Cryptography . . . 107
7.10.1 Cryptography and complex systems theory . . . 108
7.10.2 Cryptosystems based on cellular automata . . . 108
7.11 Exercises . . . 109

8 Faster Supervised Learning **111**
8.1 Objectives . . . 111
8.2 Radial Basis Functions . . . 111
8.2.1 XOR Again . . . 112
8.2.2 Learning weights . . . 113
8.2.3 Approximation problems . . . 114
8.2.4 RBF and MLP as Approximators . . . 114
8.2.5 Comparison with MLPs . . . 115
8.2.6 Finding the Centres of the RBFs . . . 118
8.3 Error Descent . . . 118
8.3.1 Mathematical Background . . . 119
8.3.2 QuickProp . . . 120
8.4 Line Search . . . 120
8.4.1 Conjugate Gradients . . . 120
8.5 Cascade Correlation . . . 122
8.6 Reinforcement Learning . . . 123
8.7 Second Order Methods . . . 124
8.8 Exercises . . . 124

A The perceptron learning Theorem **125**

B Linear Algebra **127**
B.1 Vectors . . . 127
B.1.1 Same direction vectors . . . 127
B.1.2 Addition of vectors . . . 127
B.1.3 Length of a vector . . . 128
B.1.4 The Scalar Product of 2 Vectors . . . 128
B.1.5 The direction between 2 vectors . . . 128
B.1.6 Linear Dependence . . . 128
B.1.7 Neural Networks . . . 128
B.2 Matrices . . . 129
B.2.1 Transpose . . . 129
B.2.2 Addition . . . 129
B.2.3 Multiplication by a scalar . . . 129
B.2.4 Multiplication of Matrices . . . 129
B.2.5 Identity . . . 130
B.2.6 Inverse . . . 130

C Calculus **131**
C.1 Introduction . . . 131
C.1.1 Partial Derivatives . . . 131
C.1.2 Second Derivatives . . . 132

D Laboratory Exercises **133**

Preface

This course is an introduction to the subject of Artificial Neural Networks and Genetic Algorithms, two very new subjects forming part of Distributed Artificial Intelligence.

As you leaf through these notes you will notice that they are full of mathematical equations. The reason is simple: these subjects are inherently mathematical. However the course and assessments are such that it will be possible for you to pass if you do not touch the equations. However if you wish to gain a good pass you must attempt to master the equations. In addition, by doing so, you will gain a deeper insight into the operation of these two exciting technologies.

The aims of the course are that you should be able to

1. identify tasks which can be solved by these methods
2. identify and use the appropriate specific neural network or genetic algorithm for specific tasks
3. implement the method

Since these are broad aims, we begin each Chapter with a statement of objectives. We make explicit reference to these objectives in the Exercises at the end of each Chapter but expect you to be aware of these objectives when you are in the laboratory.

The laboratory work is designed to both teach you the contents of the course and assess your understanding of the algorithms. You will have one two-hour laboratory session each week. The first few laboratories are using a simulator which you can run by simply pointing and clicking the mouse. This is to allow you to concentrate on the high level features of the Artificial Neural Network you are using and not get involved in the details which you have to be aware of when you are programming a network. However in subsequent weeks you will have to program your own nets. This is an essential and assessable part of the course.

You will have two lectures per week which will be mainly devoted to me telling you about the charms of ANNs and GAs. There will however be a 5-10 minute slot in each lecture for daft questions. The "Daft Question" slot is important so please use it to ask any question (about the course) which comes to mind. All questions will be treated as equally daft.

You will have one tutorial per week. All tutorials are activity based - you will be directed to a piece of work each week and expected to report on the outcome of your (group's) deliberations during that tutorial.

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 introductory course to those new information processing simulations which are intended to emulate the information processors which we find in biology. These are often categorised as

1. Neuron based information processing
2. Genetic information processing
3. Immunity building informaton processing

This course will introduce the first two of these as aspects of Artificial Intelligence.

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.

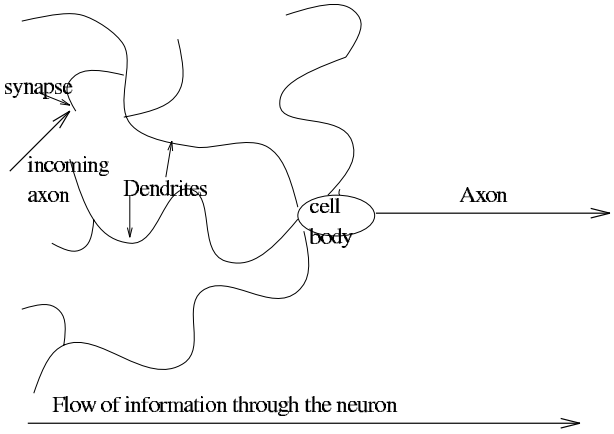

Figure 1.1: A simplified neuron

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 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

[1] Actually several weeks before birth

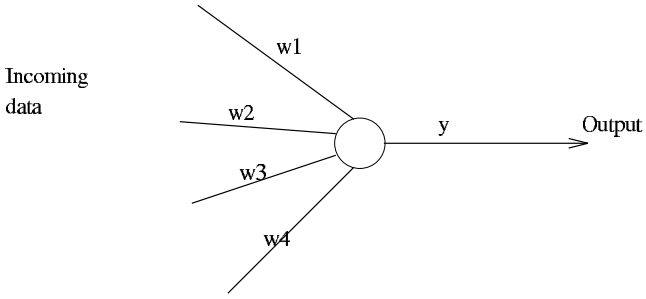

Figure 1.2: The artificial neuron. The weights model the synaptic efficiencies. Some form of processing not specified in the diagram will take place within the cell body.

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 $\mathbf{x}$ and the synapses' efficiencies are modelled by a weight vector $\mathbf{w}$. 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($\mathbf{x}$)=$\mathbf{x}$. 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 $\mathbf{w}.\mathbf{x} = \sum_j w_j x_j$ = 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, $\mathbf{x}$) 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

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

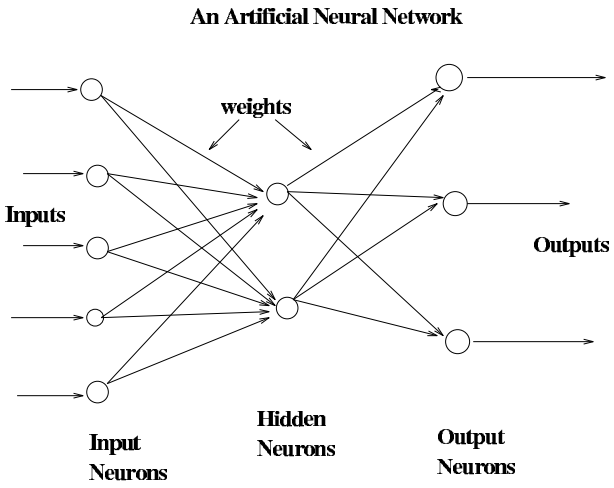

Figure 1.3: A typical artificial neural network consisting of 3 layers of neurons and 2 conecting layers of weights

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.

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 3). 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.

During the last decade, the topic has received increasing attention from

- computer scientists who believe that a new method of computation may result from emulating the brain
- statsticians who believe that neural processing allows novel statisitical approaches to problems which are not solvable using traditional techniques
- physicists who believe that their techniques for understanding sub-atomic particles may be appropriate for understanding the operations of the brain
- psychologists who believe the new science will allow them to develop psychological theories based both on empirical evidence and neural network theories to create a real physics of the brain
- neuro-physiologists who believe that the new science offers a potential theoretical underpinning of their databases of neural behaviours.
- scientists from other disciplines who use the new techniques in their armoury of data investigation methods
- commercial/industrial groups who wish to evaluate the techniques

This course will provide an introduction to the topic. You are advised to read Haykin, Chapter 1.

1.8 Genetic Algorithms

This course will also introduce Genetic Algorithms.

Evolution is an astonishingly good problem solver. Genetic algorithms are a class of simulations which attempt to mimic natural evolution as a strategy in problem solving. Genetic information processing is like neural information processing in that it too seems to be robust - errors in individual codes are often detected and corrected before any fatal consequences occur. However, genetic information processing takes place on a far longer timescale to that of neural processing.

In nature the basic mechanism forcing change *on a species* is believed to be **survival of the fittest** which gives greater probability of survival to those individuals in a population which are most fit; with greater probability of survival comes greater probability of success in any competition to reproduce; and so, finally, if an individual has greater probability of success in this competition, he/she will have greater chance of passing on his/her genes. Thus the species as a whole becomes more fit for its environment.

We use genetic algorithms often for complex optimisation problems: we first create a coding scheme which allows us to evaluate how fit a particular individual is in terms of optimising the task it has been set. We then create randomly a population of individuals so coded and evaluate their fitness. We must allow the population to have the opportunity to develop greater fitness and this we do by allowing the genes to both combine (2 parents forming new offspring) and to mutate. We then evaluate the fitness of the whole population and employ some sort of survival of the fittest routine over the population.

Typical GA problems include scheduling problems, the Travelling Salesman Problem and process control maximisation problems. The inherent parallelism of the operation is thought to be an attractive feature of GAs particularly since such problems typically are not solvable by any algorithm.

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

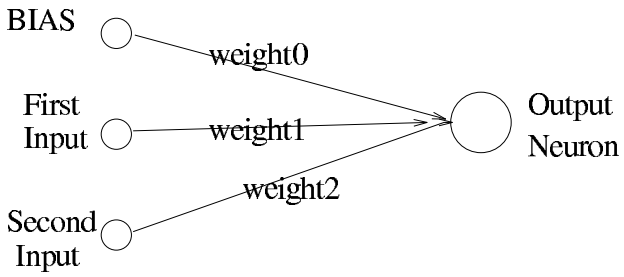

Figure 1.4: The Simple Neural Network

1.9 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.

1.9.1 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.

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.

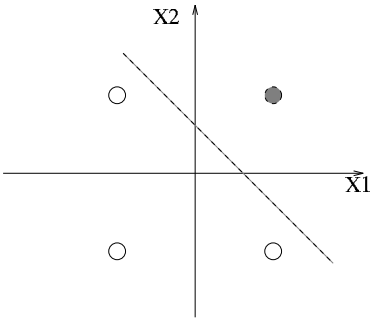

Figure 1.5: The line joining (0,0.2) and (0.2,0) cuts the 4 points into 2 sets correctly

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

$$0.5x_1 + 0.5x_2 - 0.1 = 0 \tag{1.5}$$

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.

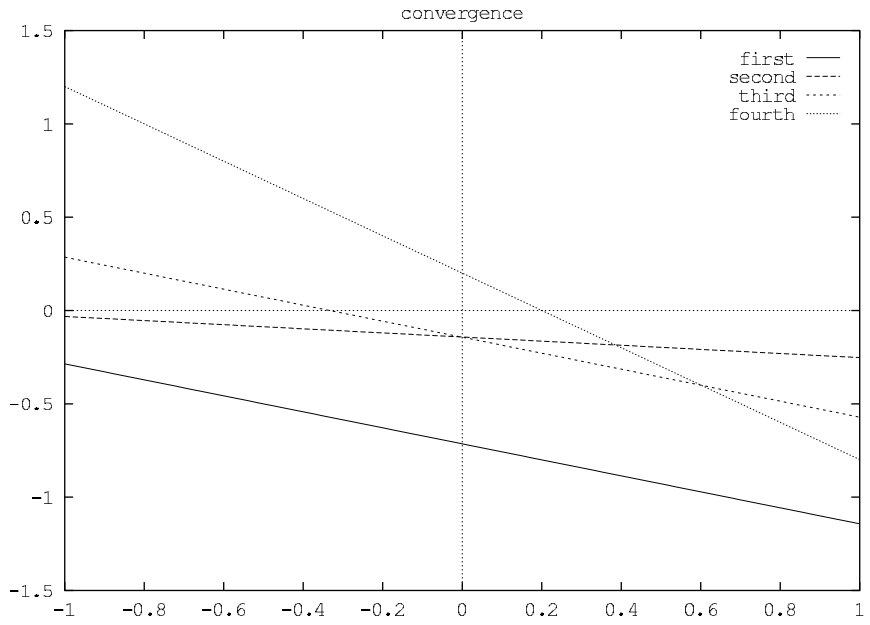

Figure 1.6: The iterative convergence of the network to a set of weights which can perform the correct mapping is shown diagrammatically here.

1.9.2 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).
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 , $\mathbf{x}$, 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).

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

Chapter 2

Associative Memory

2.1 Objectives

After this Chapter, you should

1. understand the terms "associative memory", "autoassociation", "heteroassociation", "recall" and "crosstalk".

2. be able to create a simple memory matrix using Hebb's rule.

3. understand the significance and reasons for crosstalk.

4. be able to use Lyapunov's method in simple situations.

5. be able to use the vector and matrix notation for simple memories.

2.2 Memory

We cannot remember an event before it happens. Therefore an event happens, some change takes place in our brains so that subsequently we can remember the event. So memory is inherently bound up in the learning proces.

In this chapter, we will meet a simple example of associative memory: the type of memory which allows us to associate a particular taste with a particular sight and, at a different level, a particular tune with a particular person etc. Therefore we will develop an artificial neural network (see Figure 2.1) which performs a mapping from an input vector (e.g. a set of sights) to an output vector (e.g. a taste). The memory which links these is found in the weights which join input to output and the process of learning this memory is equivalent to changing the weights.

There are two types of association:

Autoassociation: where a pattern vector is associated with itself. This is most useful for pattern completion where a partial pattern (a pair of eyes) or a noisy pattern (a blurred image) is associated with its complete and accurate representation (the whole face).

Heteroassociation: where a vector is associated with another vector which may have different dimensionality. We may still hope that a noisy or partial input vector will retrieve the complete output vector.

This chapter relates to Haykin Chapter 3, pages 90-114. The exercises on p114 are very good.

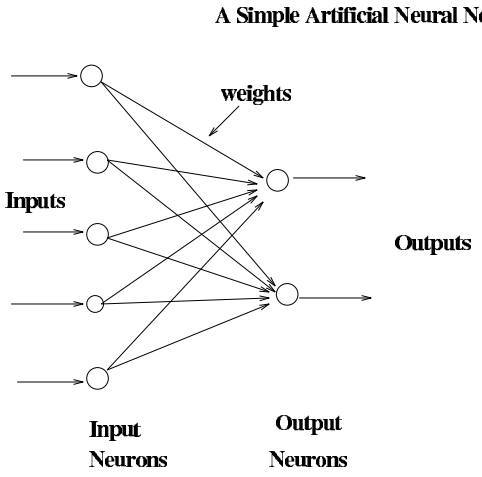

Figure 2.1: This simple two layer network can be used to hold memories associating input data with outputs.

2.3 Heteroassociation

Let us assume that the input vector, $\mathbf{x}$ is n-dimensional and the output vector, $\mathbf{y}$ is m-dimensional. Then the input data (the stimulus) is represented by the firing of n different input neurons and the output vector (the response) is held on m output neurons. There is a set of weights between the input neurons and the output neurons which is modelled by a matrix, W which is m*n. Then we have

$$\begin{aligned} \mathbf{x} &= (x_1, x_2, ..., x_n)^T \\ \mathbf{y} &= (y_1, y_2, ..., y_m)^T \end{aligned}$$

linked by W, where w_{ij} is the weight from x_j to y_i. Then if we present $\mathbf{x}_k$, the k^{th} input pattern, to the network, we wish to recall the k^{th} output pattern, $\mathbf{y}_k$. Therefore we must have the output neurons firing in the appropriate pattern when we input $\mathbf{x}_k$. i.e. $\mathbf{y}_k = W\mathbf{x}_k$ for all k.

$$\begin{bmatrix} y_{k1} \\ y_{k2} \\ \vdots \\ y_{km} \end{bmatrix} = \begin{bmatrix} w_{11}(k) & w_{12}(k) & \cdots & w_{1n}(k) \\ w_{21}(k) & w_{22}(k) & \cdots & w_{2n}(k) \\ \vdots & \vdots & \vdots & \vdots \\ w_{m1}(k) & w_{m2}(k) & \cdots & w_{mn}(k) \end{bmatrix} \begin{bmatrix} x_{k1} \\ x_{k2} \\ \vdots \\ x_{kn} \end{bmatrix} \tag{2.1}$$

Now this set of weights would, we hope, give us the required output vector (or associated memory) when we enter the k^{th} input vector. But we wish to recall p distinct patterns when presented with p distinct stimuli or input patterns. Therefore we will use a m*n memory matrix, M, defined by

$$\begin{aligned} M &= \sum_{k=1}^{p} W(k) \text{ where} \\ W(k) &= \begin{bmatrix} w_{11}(k) & w_{12}(k) & \cdots & w_{1n}(k) \\ w_{21}(k) & w_{22}(k) & \cdots & w_{2n}(k) \\ \vdots & \vdots & \vdots & \vdots \\ w_{m1}(k) & w_{m2}(k) & \cdots & w_{mn}(k) \end{bmatrix} \end{aligned}$$

the matrix memory for the k^{th} pattern alone.

2.4 Creating the Memory Matrix

Firstly we note that we can recursively calculate the memory matrix using

$$M_k = M_{k-1} + W_k \tag{2.2}$$

if we define M_0 to be 0, the zero matrix (i.e. the memory matrix initially holds no memories). Therefore if we have a memory matrix holding k-1 patterns we can simply add in the memories for the k^{th} pattern. Thus our problem is one of calculating W_k, the matrix of memories of the k^{th} pattern.

The simplest way of doing this is by using Hebb's learing rule which we shall meet in detail in Chapter 5. Hebbian learning is so-called after Donald Hebb who in 1949 conjectured:

> When an axon of a 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.

We generally equate the efficiency of a synapse in a biological network with the weight of a connection in an artificial neural network. Therefore the rule is written as $w_{ij}(k) = x_j y_i$. That is the weight between each input and output neuron is increased proportional to the magnitude of the simultaneous firing of these neurons. We can write this in matrix terms as

$$\begin{aligned} W(k) &= \mathbf{y}_k \mathbf{x}_k^T \\ &= \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \\ &= \begin{bmatrix} y_1x_1 & y_1x_2 & \cdots & y_1x_n \\ y_2x_1 & y_2x_2 & \cdots & y_2x_n \\ \vdots & \vdots & \vdots & \vdots \\ y_mx_1 & y_mx_2 & \cdots & y_mx_n \end{bmatrix} \end{aligned}$$

Note that an n-dimensional input vector and an m-dimensional output vector give an m*n dimensional weight matrix. If we were associating the input patterns, $\mathbf{y}$ with the output patterns $\mathbf{x}$, we would have the n*m weight matrix which is the transpose of the current matrix.

We can now see that the output from the system, the vector $\mathbf{y}$, is

$$\begin{aligned} \mathbf{y} &= W(k)\mathbf{x}_k \\ &= \mathbf{y}_k \mathbf{x}_k^T \mathbf{x}_k \\ &= \mathbf{y}_k \end{aligned}$$

if the input patterns have been normalised to length 1.[1]

Therefore we can see that using the vector W(k) we get a truly associative memory which will associate the correct output vector with the k^{th} input vector.

Therefore the sequence W(k) seems to fulfil our requirements for an associative memory. However, we have ignored the possibility that when we add the successive W(k), we may get interference between sets of patterns. This interference is usually called **crosstalk**.

2.5 Recall and Crosstalk

We can investigate the effect of crosstalk as follows. Let the vector $M = \sum_{k=1}^{p} W(k)$. Then let the i^{th} input pattern, $\mathbf{x}_i$ evoke the response $\mathbf{y}$. Then we have

$$\mathbf{y} = M\mathbf{x}_i$$

[1]When the input patterns have not been normalised, our outputs are in the same direction as the associated vector $\mathbf{y}_k$ but may be larger or smaller

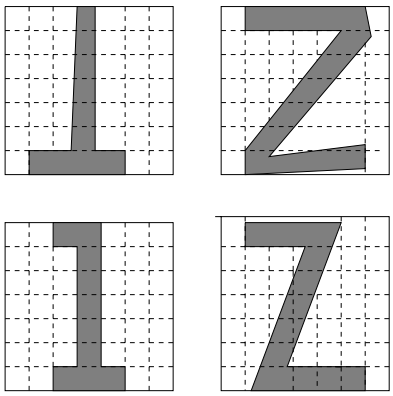

Figure 2.2: Sample patterns of ones and twos. Each pattern is converted to a 49 dimensional vector in which a dimension has the value one if more than half the pixel is contained in the figure and 0 otherwise

$$\begin{aligned} &= \sum_k W_k \mathbf{x}_i \\ &= \sum_k \mathbf{y}_k \mathbf{x}_k^T \mathbf{x}_i \\ &= \mathbf{y}_i \mathbf{x}_i^T \mathbf{x}_i + \sum_{k \neq i} \mathbf{y}_k \mathbf{x}_k^T \mathbf{x}_i \\ &= \mathbf{y}_i + \mathbf{c}_i \end{aligned}$$

where $\mathbf{c}_i$ is the crosstalk due to the effect of the other patterns. These two terms can be thought of as signal and noise respectively so what we would like would be to minimise the noise so that the recalled pattern is as close to the associated pattern $\mathbf{y}_i$ as possible. One way to do this is to make the input patterns orthogonal (or as orthogonal as possible). Note that the cosine of the angle between two (normalised) input vectors is given by

$$\cos(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{x}_i . \mathbf{x}_j}{\| \mathbf{x}_i \| . \| \mathbf{x}_j \|} = \frac{\mathbf{x}_i^T \mathbf{x}_j}{\| \mathbf{x}_i \| . \| \mathbf{x}_j \|} = \mathbf{x}_i^T \mathbf{x}_j \tag{2.3}$$

since the input vectors have been normalised to have unit length, $\| \mathbf{x}_i \| = 1, \forall i$. So we can see that the crosstalk term

$$\mathbf{c}_i = \sum_{k \neq i} \mathbf{y}_k \cos(\mathbf{x}_k, \mathbf{x}_i) \tag{2.4}$$

which will be minimised when the angle between the vectors is as close to a right angle as possible and will be zero when the vectors are orthogonal. Note that if the vectors are not orthogonal, there will inevitably be some crosstalk between the vectors. We say that the associative memory works perfectly only when the input patterns form an **orthonormal** set i.e.

$$\begin{aligned} \mathbf{x}_k^T \mathbf{x}_i &= 0 \text{ when } i \neq k \\ \mathbf{x}_i^T \mathbf{x}_i &= 1 \end{aligned}$$

2.5.1 Example

We would like to train an artificial neural network to differentiate between sets of patterns on a square grid of pixels. Consider Figure 2.2. We show 4 patterns two of which we wish to associate with pattern "one" and two of which we will identify as pattern "two".

To train a neural network to differentiate between these two sets of patterns (and we assume that we have many examples of "ones" and "twos"), we set up a neural network with 49 inputs and 2 outputs. Each of the 49 inputs corresponds to one pixel so that if the example drawn on the grid is such that pixel 18 (say) is more than half inside the figure, the 18^{th} element of the vector is a 1; if the 18^{th} pixel on the other hand

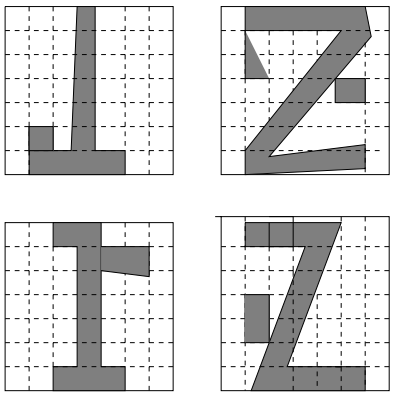

Figure 2.3: Noisy versions of the previous data which when input to the network will be classed as "ones" and "twos" as appropriate.

is more than half outside the figure, the 18^{th} element of the vector is a 0. Therefore our input data to the neural network is a 49 dimensional vector composed only of ones and zeroes.

The output vector is only a two dimensional vector which will be $(1,0)^T$ if the input pattern is a "one" and $(0,1)^T$ if the input pattern is "two".

Then for the k^{th} pattern,

$$W(k) = \mathbf{y}_k \mathbf{x}_k^T \tag{2.5}$$

So if the pattern is a "one", the second row of W is all zeros while if the pattern is a "two" the first row is a row of zeroes. Also the other row in each case is simply a row containing the 49 dimensional input vector. e.g. for the first "two" pattern above

$$W = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \tag{2.6}$$

So if we have p patterns

$$\begin{aligned} M &= \sum_{k=1}^{p} W(k) \\ &= \begin{bmatrix} \sum_{i \in P_1} \mathbf{x}_i^T \\ \sum_{i \in P_2} \mathbf{x}_i^T \end{bmatrix} \end{aligned}$$

where P_i is the set of training patterns of type i.

So the fact that the y-values are orthogonal to one another means that, in the weight matrix, the memories associated with "ones" do not interfere with the memories associated with "twos". However all the "ones" memories are interfering with each other; in other words, the memory which will be retained will be that which is formed by the crosstalk between patterns of the same type - the crosstalk is actually helpful in forming an idealised version of "one" and of "two".

Thus we can see that this matrix memory is proportional to the mean of the training patterns of each type which it meets during the network training. Therefore the ability of this type of network to recognise new examples of "ones" and "twos" depends on how typical were the examples on which it was trained. In general, if we put an input vector of a noisy "one" (see Figure 2.3) into the network we would hope that the output vector would more closely resemble $(1,0)^T$ than $(0,1)^T$ i.e. we should be able to identify which pattern the input is closest to by looking at the activation of the output vector. The strength of firing of the appropriate output neuron gives a measure of how closely the input corresponds to the prototypical input vector of that class.

More information on associative memories can be found in Haykin p90-98.

Student	Hard	Endur	Smart	Consc	Intu	Crea	Course	Math	Comp	Phys	EE
A_1			X	X		X	B_1	X			X
A_2	X			X	X		B_2	X	X		
A_3			X	X	X		B_3	X		X	

Table 2.1: Students attributes and their subjects.

2.6 Bidirectional Autoassociative Memories

The usual aim of autoassociation is to place a memory in a network in such a way that a noisy or partial memory will cause the original complete memory to be recalled. This will not generally be the case with the simple memory outlined above since if we have created the mapping $\mathbf{x}_k \rightarrow \mathbf{y}_k$, then if we enter a noisy version of $\mathbf{x}_k$, say $\mathbf{x}'_k = \mathbf{x}_k + \mu_k$, then we will not in general retrieve the memory $\mathbf{y}_k$ but will retrieve $M\mathbf{x}'_k = M\mathbf{x}_k + M\mu_k = \mathbf{y}_k + M\mu_k$ since we have a linear memory.

One possibility is to allow the activation to pass backward and forward through the matrix M in the hope that the divergence from the original memory will decrease in time. Therefore using $\mathbf{x}_k(0)$ to denote the k^{th} input pattern at time 0, we would get a set of iteratively closer solutions:

$$\begin{aligned}
\mathbf{y}_k(0) &= M\mathbf{x}_k(0) \text{ forward pass} \\
\mathbf{x}_k(1) &= M^T\mathbf{y}_k(0) \text{ backward pass} \\
\mathbf{y}_k(1) &= M\mathbf{x}_k(1) \text{ forward pass} \\
\mathbf{x}_k(2) &= M^T\mathbf{y}_k(1) \text{ backward pass} \\
&\vdots \\
\mathbf{y}_k(f-1) &= M\mathbf{x}_k(f-1) \text{ forward pass} \\
\mathbf{x}_k(f) &= M^T\mathbf{y}_k(f-1) \text{ backward pass}
\end{aligned} \tag{2.7}$$

Now, this sequence will not be useful for the purely linear relationships described above since e.g. $\mathbf{x}_k(1) = M^T\mathbf{y}_k(0) = M^TM\mathbf{x}_k(0) = \mathbf{x}_k(0)$ (see Question 4) i.e. we have gained nothing from the back and forth activation passing. Therefore we introduce a non-linear activation function at both the input and output neurons; we introduce the simplest - a threshold function. Then

$$\begin{aligned}
\mathbf{y}_{k,j} &= 1, \text{ if } \mathbf{M}_j.\mathbf{x} > 0 \\
\mathbf{y}_{k,j} &= 0, \text{ if } \mathbf{M}_j.\mathbf{x} < 0 \\
\mathbf{x}_i &= 1, \text{ if } \mathbf{M}_i^T.\mathbf{y} > 0 \\
\mathbf{x}_i &= 0, \text{ if } \mathbf{M}_i^T.\mathbf{y} < 0
\end{aligned}$$

where we have used $\mathbf{y}_{k,j}$ to designate the output of the j^{th} output neuron to the k^{th} input pattern and M_j is the j^{th} row of the matrix, M.

So all patterns are now binary 1/0 patterns. It will be shown by the method of Lyapunov that every matrix M is bidirectionally stable (i.e. the process 2.7 converges to a stable solution) for every input.

2.6.1 An Example

Let us create a BAM for the pair of memories shown in Table 2.1 in which an X in any position identifies that the student has that quality or that the course contains that class. We wish to create a memory which associates Student i with Course i.

First we may represent each student and course by a vector e.g.

$$\begin{aligned}
A_1 &= (0,0,1,1,0,1)^T & B_1 &= (1,0,0,1)^T \\
A_2 &= (1,0,0,1,1,0)^T & B_2 &= (1,1,0,0)^T \\
A_3 &= (0,0,1,1,1,0)^T & B_3 &= (1,0,1,0)^T
\end{aligned}$$

The BAM is easier to construct with a bipolar coding and so we use

$$X_1 = (-1,-1,1,1,-1,1)^T \quad Y_1 = (1,-1,-1,1)^T$$
$$X_2 = (1,-1,-1,1,1,-1)^T \quad Y_2 = (1,1,-1,-1)^T$$
$$X_3 = (-1,-1,1,1,1,-1)^T \quad Y_3 = (1,-1,1,-1)^T$$

We can now find correlation matrices:

$$X_1 Y_1^T = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

$$X_2 Y_2^T = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

$$X_3 Y_3^T = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Let us consider initially a matrix containing only the first two pairs of memories, $A_1 - B_1$ and $A_2 - B_2$. Then we get

$$M = X_1 Y_1^T + X_2 Y_2^T = \begin{bmatrix} 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & 2 \\ 2 & 0 & -2 & 0 \\ 0 & 2 & 0 & -2 \\ 0 & -2 & 0 & 2 \end{bmatrix} \tag{2.8}$$

which can be used in recall with

$$A_1^T M = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & 2 \\ 2 & 0 & -2 & 0 \\ 0 & 2 & 0 & -2 \\ 0 & -2 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 2 & -4 & -2 & 4 \end{bmatrix}$$

which can be thresholded to give (1001). We can similarly feed B_1 to the network to get A_1 using $B_1^T M^T$. Alternatively we can use MB_1 e.g.

$$MB_1 = \begin{bmatrix} 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & 2 \\ 2 & 0 & -2 & 0 \\ 0 & 2 & 0 & -2 \\ 0 & -2 & 0 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \\ 2 \\ 2 \\ -2 \\ 2 \end{bmatrix}$$

which can again be thresholded to give A_1. In addition we can input slightly wrong stimuli and hope to get a correct response - this is the pattern completion properties of this type of network. Let us input $A_*^T = (1,0,1,1,0,1)$. Then we have

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & 2 \\ 2 & 0 & -2 & 0 \\ 0 & 2 & 0 & -2 \\ 0 & -2 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 2 & -2 & -2 & 2 \end{bmatrix}$$

which is again thresholded to give B_1 which can be fed back as before to retrieve A_1. So a noisy version of a pattern can be fed forward and back through the network to give an accurate version of the pattern.

Now we add in the third pattern $X_3 - Y_3$. Then M now becomes

$$\begin{bmatrix} 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & 2 \\ 2 & 0 & -2 & 0 \\ 0 & 2 & 0 & -2 \\ 0 & -2 & 0 & 2 \end{bmatrix} + \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 3 & -1 & -1 \\ -3 & 1 & 1 & 1 \\ 1 & -3 & 1 & 1 \\ 3 & -1 & -1 & -1 \\ 1 & 1 & 1 & -3 \\ -1 & -1 & -1 & 3 \end{bmatrix}$$

Now if we try to recall B_1 from A_1 we get

$$A_1^T M = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -1 & 3 & -1 & -1 \\ -3 & 1 & 1 & 1 \\ 1 & -3 & 1 & 1 \\ 3 & -1 & -1 & -1 \\ 1 & 1 & 1 & -3 \\ -1 & -1 & -1 & 3 \end{bmatrix} = \begin{bmatrix} 3 & -5 & -1 & 3 \end{bmatrix}$$

which is again precisely B_1. Similarly with A_2, A_3 and even the noisy version of A_1, A_*. However the vectors used in the example were chosen with care, notice how the Y values are pairwise orthogonal to one another. As we add more and more memories to our matrix we will experience crosstalk and be unable to retrieve complete versions of our memories. Add in a fourth memory to the matrix M and attempt to retrieve all four memories.

2.6.2 Lyapunov's Method

Lyapunov's method is

1. Find a function which changes monotonically when the system is changed. i.e. find a function which consistently decreases (increases) *every time* the state of the network changes.

2. Show that the function is bounded below (above).

If we can find such a function then we know that the learning rules cause convergence though we cannot say to which final values convergence will take place.

For the Bidirectional Associative Memory (BAM), we choose the function E(**x**,**y**) such that

$$\begin{aligned} E(\mathbf{x},\mathbf{y}) &= -XMY^T \\ &= -\sum_i \sum_j x_i y_j m_{ij} \end{aligned}$$

Then E(**x**,**y**) is bounded below since $E(\mathbf{x},\mathbf{y}) \geq -\sum_i \sum_j |m_{ij}|$.

Now consider the effect of a change at the input(X) neurons. Let Δx_i be the change due to the feedback from the previous output (Y) neurons at the i^{th} input neuron. Then Δx_i is -1, 0 or 1 at any iteration, and so

$$\begin{aligned} \Delta E(\mathbf{x},\mathbf{y}) &= -\Delta X M Y^T \\ &= -\sum_i \Delta \mathbf{x}_i \sum_j y_j m_{ij} \end{aligned}$$

Now if $\sum_j y_j m_{ij} > 0$ then $\Delta \mathbf{x}_i > 0$, and so $\Delta E(\mathbf{x}_i,\mathbf{y}) < 0$. Similarly, if $\sum_j y_j m_{ij} < 0$ then $\Delta \mathbf{x}_i < 0$, and so $\Delta E(\mathbf{x}_i,\mathbf{y}) < 0$. A similar argument applies to change at the output neurons. In other words, any change will cause E to decrease and we have seen that E is bounded below so that the network must converge. Therefore regardless of the actual identity of matrix M the network will stabilise.

Example

As an extension of the previous example, we wish to teach a network how to count. Specifically if we input x we wish it to output x+1. Therefore both input and output vectors are 49 dimensional binary vectors and we train the network to associate an input vector representing "one" with an output vector representing "two", an input vector representing "two" with an output vector representing "three" etc..

Now when we enter a noisy version of a "one", it will elicit a response of a noisy version of a "two" which will be fed back to the inputs to elicit a less noisy version of a "one" etc. From the above, we know that the network will settle into a stable state. Provided the initial vector is not too noisy (and so could not be mistaken for e.g. a "seven") the stable pattern will be "one"-"two".

2.7 Conclusion

We have met two simple memories and differentiated between autoassociation and heteroassociation. We can make a further distinction between the networks presented in this Chapter: the first is a static memory whereas the second network, the BAM, is of a dynamic nature - there is a certain settling time in which the activation within the network passes back and forth eventually settling, we hope, on one of the remembered patterns. Because of this we can view the activation in the network as "short term memory" while the "long term memory" is held in the weights of the network.

The major points to note from this chapter

- The memory is contained in the weights.
- The memory is distributed.
- Both the stimulus (input pattern) and response (output pattern) are high dimensional vectors
- The recalled memory consists of a pattern of activities across the output neurons.
- The memory is robust
- There may be crosstalk between individual memories (associations)

2.8 Exercises

1. Calculate the weights used in an associative memory with which we will associate the three input patterns and output patterns shown in Table 2.2. (Objectives 2, 5).

   (a) Show that there is no crosstalk. (Objectives 1,3, 5).

   (b) Calculate the weight matrix of the memory. (Objectives 2 5).

   To use this set of patterns as a backwards memory we must normalise the output patterns and threshold at the largest.

Pattern No.	Input vector	Output vector
1	$(1,0,0)^T$	$(2,3,-1)^T$
2	$(0,1,0)^T$	$(1,-1,3)^T$
3	$(0,0,1)^T$	$(0,1,-3)^T$

Table 2.2: Three input-output pairs

Pattern No.	Input vector	Output vector
1	$(1,-1,1,-1,1,-1)^T$	$(1,1,-1,-1)^T$
2	$(1,1,1,-1,-1,-1)^T$	$(1,-1,1,-1)^T$

Table 2.3: Two input-output vectors with which to construct a BAM

2. Construct a simple BAM with the patterns given in Table 2.3. Show that the network performs perfect autoassociation with the given patterns. Investigate what happens when we enter the patterns

   (a) $(1,1,1,-1,1,-1)^T$

   (b) $(-1,1,-1,1,-1,1)^T$

   (Objectives 2, 3, 5).

3. Given a set of orthogonal m-dimensional vectors $\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n$, calculate the network memory which will perform autoassociation. What is the maximum number of vectors, n, which can be stored in such a memory? (Objectives 2, 3,5).

4. To show the need for a non-linearity in the BAM, we have assumed that $MM^T = I$, the identity matrix. Prove this. (Objective 5).

5. Explain with diagrams Lyapunov's method. (Objective 4).

Chapter 3

Simple Supervised Learning

3.1 Objectives

After this Chapter, you should

1. understand simple supervised learning.
2. be able to implement a perceptron.
3. be able to implement an Adaline.
4. know the difference between the perceptron learning rule and the LMS rule
5. know simple activation functions commonly used and their properties .
6. know the role of the bias in simple supervised learning.
7. be able to explain linear separability geometrically.
8. understand error descent procedures .

3.2 Introduction

We will consider a simple two layer[1] feedforward ANN in this chapter. We shall be interested in supervised learning (learning with a teacher) in which the network is trained on examples whose target output is known. We therefore must have a training set for which we already know 'the answer' to our questions to the network.

3.3 The perceptron

The simple perceptron was first investigated by Rosenblatt [Rosenblatt,1962]. We will define the activation passing of the perceptron by $Act_i = \sum_j w_{ij} x_j$, where
Act_i is the activation of the i^{th} output neuron,
x_j is the firing (or output) of the j^{th} input neuron
and w_{ij} is the weight from the j^{th} input neuron to the i^{th} output neuron.

We calculate the output of the jth output neuron by

$$o_i = f(Act_i) = f(\sum_j w_{ij} x_j) \tag{3.1}$$

[1]We will use the convention that this refers to the number of layers of neurons; some authors refer to the number of layers of weights, in which case the networks of this chapter are 1 layer networks

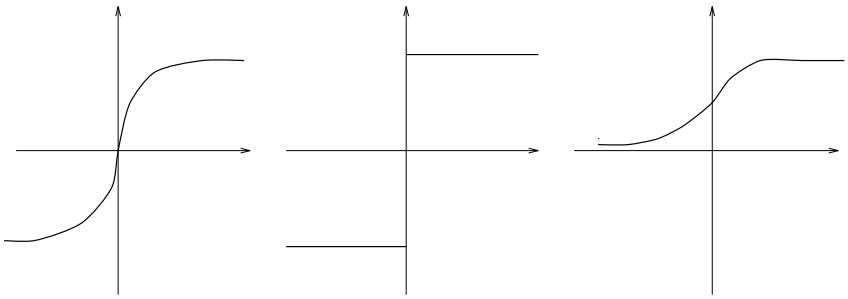

Figure 3.1: The tanh(), Heaviside, and logistic functions

where typically f() is a non-linear function such as a threshold function, a sigmoid or a semi-linear function (see below). We will also write this as

$$o_i = f(Act_i) = f(\mathbf{w}_i.\mathbf{x}) \tag{3.2}$$

where $\mathbf{w}_i = [w_{i0}, w_{i1}, ..., w_{iN}]^T$ is the vector of weights into the i^{th} output neuron and $\mathbf{x} = [x_0, x_1, x_2, ..., x_N]^T$ is the vector of input activations.

We will often omit any mention of a **threshold** in the networks we use since we can, if we need a threshold, select a particular input (often x_0) to be on (i.e. equal to 1) all of the time and chose weights $w_{i0} = \theta_i$, to give threshold θ_i since

$$o_i = f(Act_i) = f(\sum_{j=0}^{N} w_{ij}x_j) = f(\sum_{j=1}^{N} w_{ij}x_j + \theta_i) \tag{3.3}$$

The functions $f()$ are known as **activation functions.** Typical activation functions are shown in Figure 3.1

The Heaviside function, or step function, or sgn function is defined by

$$\begin{aligned} f(x) &= 1, \text{ if } \quad x > 0 \\ f(x) &= -1, \text{ if } \quad x < 0 \end{aligned}$$

The other two functions are continuous functions which asymptote at large absolute values of x. The logistic function is defined by

$$f(x) = \frac{1}{1 + e^{-ax}} \tag{3.4}$$

i.e. $f \to 0$ when x is very negative and $f \to 1$ when x is large and positive. The tanh() function similarly asymptotes at -1 and 1. A half-way stage between the step function and the sigmoid functions is the semi-linear function defined by

$$\begin{aligned} f(x) &= 0, && \text{if } x < a \\ f(x) &= 1, && \text{if } x > b \\ f(x) &= \tfrac{(x-a)}{(b-a)}, && \text{for } a < x < b. \end{aligned}$$

We will be using supervised learning with this type of ANN and so when we present the P^{th} input pattern we must also present the network with the P^{th} target pattern. We then attempt to ensure that $o_i^P = t_i^P$ for all output neurons. i.e. the learning process must use the fact that on presentation with the P^{th} input vector we desire to get the P^{th} output vector equal to the P^{th} target vector.

It is possible that the input and output patterns are the same in which case we say that we are performing **autoassociation**; otherwise we have **hereroassociation.** Typically, for this type of feedforward network, we shall be using different patterns at inputs and outputs and are thus performing heteroassociation.

3.4 The perceptron learning rule

In their simplest form, perceptrons consist of binary units; consider a perceptron with N input neurons and a single output neuron. Then the perceptron must learn the mapping $T : \{-1,1\}^N \longrightarrow \{-1,1\}$ based on samples of input vectors ,**x**. An example is the mapping $T : \{-1,1\}^3 \longrightarrow \{-1,1\}$ defined by

$$\begin{aligned} T : (-1,1,1) &\longrightarrow 1 \\ T : (1,1,-1) &\longrightarrow 1 \\ T : (1,-1,-1) &\longrightarrow -1 \end{aligned}$$

In this mapping we have only defined the mapping on some of the possible inputs - the others are don't care values. However, in order to be deemed to have learned the mapping, the perceptron must get the above three values correct.

The output neuron of the simple perceptron is a linear threshold unit which takes the value 1 or -1 according to the rule

$$\begin{aligned} o &= f(\textstyle\sum_{j=1}^{N} w_j x_j + \theta) = 1, \quad \text{if } \sum_{j=1}^{N} w_j x_j + \theta > 0 \\ o &= f(\textstyle\sum_{j=1}^{N} w_j x_j + \theta) = -1, \quad \text{if } \sum_{j=1}^{N} w_j x_j + \theta < 0 \end{aligned}$$

i.e. f() is the simple step function. The generalisation to an M-output perceptron is

$$\begin{aligned} o_i &= f(\textstyle\sum_{j=1}^{N} w_{ij} x_j + \theta_i) = 1, \quad \text{if } \sum_{j=1}^{N} w_{ij} x_j + \theta_i > 0 \\ o_i &= f(\textstyle\sum_{j=1}^{N} w_{ij} x_j + \theta_i) = -1, \quad \text{if } \sum_{j=1}^{N} w_{ij} x_j + \theta_i < 0 \end{aligned}$$

Rosenblatt showed in 1959 that if it was possible for a mapping T to exist, then the perceptron learning algorithm could be guaranteed to converge to it. A proof of the theorem is given in Appendix A.

The algorithm can be described by:

1. begin with the network in a randomised state: the weights between all neurons are set to small random values (between -1 and 1).
2. select an input vector, **x**, from the set of training examples
3. propagate the activation forward through the weights in the network to calculate the output o.
4. if $o^P = t^P$, (i.e. the network's output is correct) return to step 2.
5. else change the weights according to $\Delta w_i = \eta x_i^P (t^P - o^P)$ where η is a small positive number known as the **learning rate.** Return to step 2.

Thus we are adjusting the weights in a direction intended to make the output, o, more like the target value, t, the next time an input like **x** is given to the network.

We must emphasise the importance of this rule is that it is guaranteed to converge to the answer in a finite time if the answer exists.

3.5 An example problem solvable by a perceptron

Consider a simple classification problem. We wish to train a perceptron to differentiate between two different types of nuts. We have 3 examples of each type of nut and can measure 2 characteristics of each nut: its

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 3.1: The lengths and weights of six instances of two types of nuts

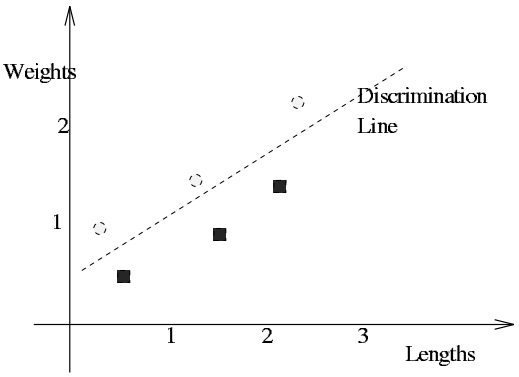

Figure 3.2: The two classes of nut and a discrimination line which can differentiate between the classes

length and its weight. Then our data is shown in tabular form in Table 3.1. Then we will train a perceptron 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. Its trainable weight will converge to the bias for the output neuron. So we have a network with three inputs and a single output which will tell us with a 1 if the nut is A or a -1 if the nut is a B. Therefore our network has 3 weights.

Now we set our initial weights to small random values. Select randomly one of the patterns e.g. the third. Feed the activation forward through the weights and perform a thresholding at the output. If the output neuron is firing 1, do nothing. If, however, the output neuron is firing -1, it is wrongly classifying the nut and its weights must be changed. Change each of the three weights according to the perceptron learning rule.

The weights in this simple problem are guaranteed to converge to something like $w_0 = 0.5, w_1 = 2, w_2 = 3$. A diagrammatic representation of the solution is shown in Figure 3.2.

Now it is clear that it is in fact trivial to differentiate between these two classes on the basis of these six instances. However the perceptron has been shown to be capable of solving more difficult problems though we must in any real problem ensure that the data on which we are training the network is representative of the data as a whole. Only then can we state with confidence that we have a perceptron capable of distinguishing the classes.

Statisticians would say that we have created a **decision surface**, or **discrimination line**, between the two classes. Discriminant functions for the classes would be $g_A(x,y)$ and $g_B(x,y)$ where we would state that a point (x,y) belonged to class A if and only if $g_A(x,y) > g_B(x,y)$; otherwise, (x,y) belongs to class B. The line (or plane) which differentiates the classes is $g_A(x,y) - g_B(x,y) = 0$ which is the line shown in the Figure.

3.5.1 And now the bad news

Now a crucial part of the above description is the '*if it was possible*' part. Minsky and Papert showed that there exist many functions which cannot be solved by this simple mapping. So, following Hertz et al, we can show this diagrammatically in Figure 3.3.

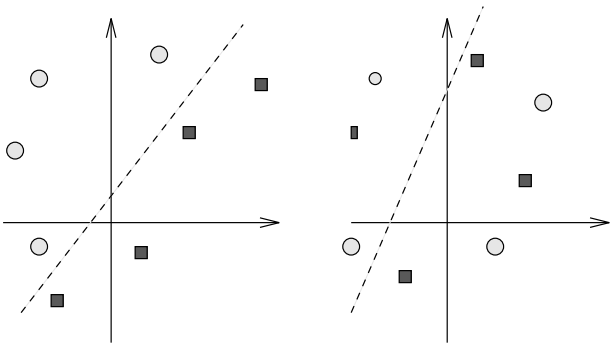

Figure 3.3: In the left part of the diagram we show that we can use a single line to categorise all rectangles as belonging to the region below and to the right of the line while all ovals lie above and to the left of the line ; in the right diagram, the line can no longer discriminate between the classes.

X and	X	1	1	0	0
	Y	**1**	**0**	**1**	**0**
	Y	1	0	0	0

X OR	X	1	1	0	0
	Y	**1**	**0**	**1**	**0**
	Y	1	1	1	0

X XOR	X	1	1	0	0
	Y	**1**	**0**	**1**	**0**
	Y	0	1	1	0

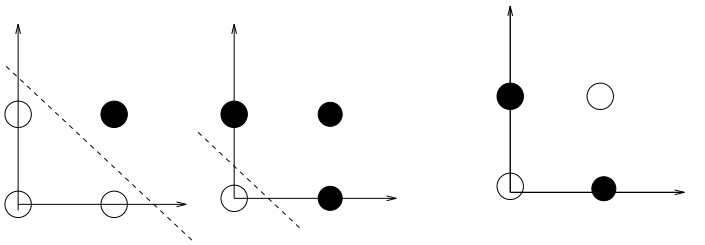

Figure 3.4: A diagrammatic representation of AND, OR and XOR with the corresponding truth tables where 0 is false and 1 is true

Notice that in the right half of the diagram no line can be found which will allow us to discriminate between all rectangles (representing one class) and all ovals (representing the other class). We say that the classes in the left diagram are **linearly separable** while those in the right diagram are **linearly inseparable.**

The simple example which Minsky and Papert used was the XOR pattern which is readily shown not to be linearly separable (see Figure 3.4.

The simple perceptron cannot differentiate between the XOR set of patterns (you will test this in Laboratory 2). Minsky and Papert showed that this problem is the simplest example of a set of problems which may be characterised as belonging to the parity problem (The XOR truth table above shows even parity).

3.6 The Delta Rule

Another important early network was the Adaline (ADAptive LINear Element). The Adaline calculates its output as $o = \sum_j w_j x_j + \theta$, with the same notation as before. You will immediately note the difference between this network and the perceptron is the lack of thresholding. The interest in the network was partly due to the fact that it is easily implementable as a set of resistors and switches.

3.6.1 The learning rule: error descent

For a particular input pattern, $\mathbf{x}^P$, we have an output o^P and target t^P. Then the sum squared error from using the Adaline on all training patterns is given by

$$E = \sum_P E^P = \frac{1}{2} \sum_P (t^P - o^P)^2 \tag{3.5}$$

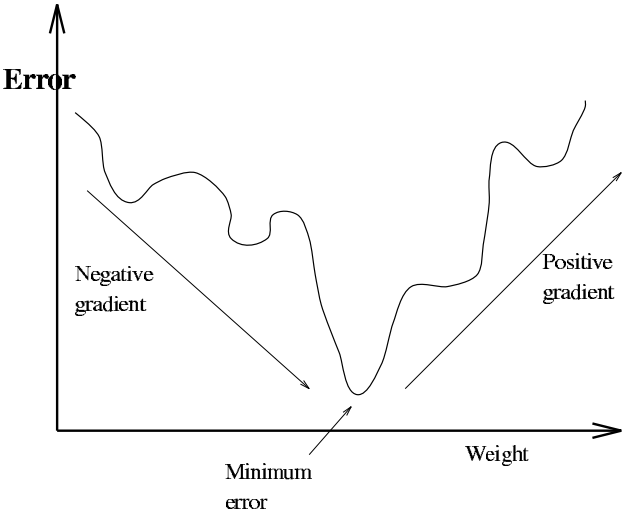

Figure 3.5: A schematic diagram showing error descent. In the negative gradient section, we wish to increase the weight; in the positive gradient section, we wish to decrease the weight

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 3.5. 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,

$$\frac{\partial E^P}{\partial w_j} = \frac{\partial E^P}{\partial o^P} . \frac{\partial o^P}{\partial w_j} \tag{3.6}$$

and because of the linearity of the Adaline units, $\frac{\partial o^P}{\partial w_j} = x_j^P$. Also, $\frac{\partial E^P}{\partial o^P} = -(t^P - o^P)$, and so $\Delta_P w_j = \gamma(t^P - o^P).x_j^P$. Notice the similarity between this rule and the perceptron learning rule; however, this rule has far greater applicability in that it can be used for both continuous and binary neurons. 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 the conditions for guaranteed convergence which we used in proving the perceptron learning theorem do not now pertain. Therefore there is nothing to prevent learning in principle never converging.

However, since it is so central we will consider 2 variations of the basic rule in the sections below.

3.6.2 Non-linear Neurons

The extension to non-linear neurons is straightforward. The firing of an output neuron is given by $o = f(\sum_j w_j x_j)$, where f() is some non-linear function of the neuron's activation. Then we have

$$E = \sum_P E^P = \frac{1}{2}\sum_P (t^P - o^P)^2 = \frac{1}{2}\sum_P (t^P - f(\sum_j w_j x_j^P))^2 \tag{3.7}$$

and so

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial o} . \frac{\partial o}{\partial act} \frac{\partial act}{\partial w_j}$$

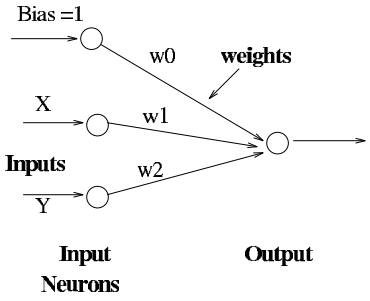

Figure 3.6: The network with a single layer of weights which we will use to attempt to solve the XOR problem

$$\frac{\partial E}{\partial w_j} = -\sum_P (t^P - f(\sum_j w_j x_j^P)).f'(\sum_j w_j x_j^P).x_j^P$$

So using the error descent rule, $\Delta w_j = -\gamma \frac{\partial E}{\partial w_j}$, we get the weight update rule

$$\begin{aligned} \Delta w_j &= \gamma \delta x_j \\ \text{where } \delta &= \sum_P (t^P - f(\sum_j w_j x_j^P)).f'(\sum_j w_j x_j^P) \end{aligned}$$

The sole difference between this rule and that presented in the last section is the f' term. Its effect is to increase the rate of change of the weights in regions of the weight space where f' is large i.e. where a small change in the activation makes a large change in the value of the function. You can see from the diagrams of the activation functions given earlier that such regions tend to be in the centre of the function's range rather than at its extreme values.

This learning rule is a **batch** learning rule i.e. the whole set of training patterns are presented to the network and the total error (over all patterns) is calculated and only then is there any weight update. A more usual form is to have **on-line** learning where the weights are updated after the presentation of each pattern in turn. This issue will be met again in the next chapter. The online version of the learning rule is

$$\begin{aligned} \Delta_P w_j &= \gamma \delta^P x_j^P \\ \text{where } \delta^P &= (t^P - f(\sum_j w_j x_j^P)).f'(\sum_j w_j x_j^P) \end{aligned}$$

Typical activation functions are the logistic function and the tanh() function both of which are differentiable. Further, their derivatives can be easily calculated:

- if f(x) = tanh(bx), then f'(a) = b(1 - f(a)*f(a)); i.e. the derivative of the function at a point a is an easily calculable function of its value at a.
- similarly, if f(x) = 1/(1+exp(-bx)) then f'(a)=bf(a)(1 - f(a)).

As noted earlier, these functions have the further important property that they asymptote for very large values.

3.6.3 XOR revisited

Consider the basic network shown in Figure 3.6. X and Y represent the (binary) inputs while x_0 is set always to 1 and will be the constant bias term. In the case of linear (Adaline) neurons, the best which this network can do is to set $w_0 = w_1 = w_2 = 0$; in this case, the least squared error is 4 (see Table 3.2).

i.e. the total error, E=4. The network is basically giving 4 'don't know' answers.

Now if we repeat the experiment with a tanh() nonlinearity being calculated at the output neuron, we will find that another four more states become possible. An example is shown in Table 3.3; to produce these

X	Y	X XOR Y	NETWORK OUTPUT	SQUARED ERROR
0	0	-1	0	1
1	0	1	0	1
0	1	1	0	1
1	1	-1	0	1

Table 3.2: A network giving 4 "don't know" answers

X	Y	X XOR Y	NETWORK OUTPUT	SQUARED ERROR
0	0	-1	-1	0
1	0	1	1	0
0	1	1	1	0
1	1	-1	1	4

Table 3.3: A network giving 3 correct answers and a single wrong answer. Note that the total squared error is the same as before.

results, the network converged to $w_2 = w_1 \to \infty$ and $w_0 \to -\infty$. This could be thought to be a slightly superior answer since the network is giving 3 correct answers (though 1 wrong answer).

This solution has the same sum squared error as the previous one and may be thought of as a **local minimum** which the introduction of a non-linearity makes possible as a point of convergence.

3.6.4 Nuts Revisited

We use the same data (Table 3.1) that we used before i.e. we have the same inputs and targets. So we will train a non-linear net with the input vectors , $\mathbf{x}$, equal to
$(1, 2.2, 1.4)^T$ with associated target output 1 (equal to class A)
$(1, 1.5, 1.0)^T$ with target output 1
$(1, 0.6, 0.5)^T$ with target output 1
$(1, 2.3, 2.0)^T$ with associated target output -1 (equal to class B)
$(1, 1.3, 1.5)^T$ with target output -1
$(1, 0.3, 1.0)^T$ with target output -1

Each cycle will have 3 stages:

1. Select randomly an input pattern and associated target pattern
2. Feed the input pattern forward through the current weights
3. Change the weights

The cycles will be repeated till the actual outputs are within 0.1 of the target outputs for all patterns.

Consider the simple network in Figure 3.6. We will use the tanh() function as an activation function since it asymptotes at 1 and -1.

Feedforward:

$$\begin{aligned} y &= f(act) = f(\sum_j w_j x_j) \\ &= tanh(act) = tanh(\sum_j w_j x_j) \end{aligned}$$

Learning:

$$\begin{aligned} \Delta w_j &= \eta(t - f(\sum_j w_j x_j)).f'(\sum_j w_j x_j).x_j \\ &= \eta(t-y).f'(\mathbf{w}.\mathbf{x}).x_j \\ &= \eta(t-y).(1-y*y).x_j \end{aligned}$$

when f() is the tanh() function.

Randomly initialise weights: let $w_0 = 0.7$, $w_1 = 0.5$, $w_2 = 0.5$. Let the learning rate be 1.

Randomly select a pattern: pattern 3.

Feedforward:

$$\begin{aligned} Act &= w_0 * 1 + w_1 * 0.6 + w_2 * 0.5 \\ &= 0.7 + 0.3 + 0.25 = 1.25 \end{aligned}$$

Now y = tanh(act) = tanh(1.25) ≈ 0.8

Change weights:

$$\begin{aligned} \Delta w_0 &= 1 * (1 - 0.8) * (1 - 0.8 * 0.8) * 1 \approx 0.06 \\ w_0 &\rightarrow 0.76 \\ \Delta w_1 &= 1 * (1 - 0.8) * (1 - 0.8 * 0.8) * 0.6 \approx 0.04 \\ w_1 &\rightarrow 0.54 \\ \Delta w_2 &= 1 * (1 - 0.8) * (1 - 0.8 * 0.8) * 0.5 \approx 0.03 \\ w_2 &\rightarrow 0.53 \end{aligned}$$

Randomly select a pattern: pattern 5.

Feedforward:

$$\begin{aligned} Act &= w_0 * 1 + w_1 * 1.3 + w_2 * 1.5 \\ &= 0.76 + 0.54 * 1.3 + 0.53 * 1.5 \approx 2.257 \end{aligned}$$

Change weights:

$$\begin{aligned} \Delta w_0 &= 1 * (-1 - 0.98) * (1 - 0.98 * 0.98) * 1 \approx -0.1 \\ w_0 &\rightarrow 0.76 - 0.1 = 0.66 \\ \Delta w_1 &= 1 * (-1 - 0.98) * (1 - 0.98 * 0.98) * 1.3 \approx -0.13 \\ w_1 &\rightarrow 0.54 - 0.13 = 0.41 \\ \Delta w_2 &= 1 * (-1 - 0.98) * (1 - 0.98 * 0.98) * 1.5 \approx -0.15 \\ w_1 &\rightarrow 0.53 - 0.15 = 0.381 \end{aligned}$$

Stop: when for all patterns the error is less than 0.1. Sometimes a cumulative rule is used e.g. when the average of the last 5 cycles was less than a particular value.

Note: The rate of change of the weights depends on the error at the output (the first term) and the magnitude of the input(the last term) *but also* crucially on the derivative of the activation function - when the tanh() function approaches 1 or -1, the rate of change of the weights slows down.

It is essential to note that here our test for goodness of the current solution occurs outside the learning rule. In the perceptron, we test if we have the correct output and do not change the weights if we have. With the LMS rule, we change the weights every time since we have a floating point output which will never (with probability 1) be exactly equal to the output. The test criterion is a criterion for stopping the simulation not a criterion as to whether we should change the weights.

3.6.5 Stochastic Neurons

It is known that in biological neural networks, the firing of a particular neuron is not always deterministic: there seems to be a probabilistic element to their firing. We can easily model such effects by introducing stochastic neurons whose probability of firing at a particular time depends on their net activation at that time. e.g.

$$P(o_j^P = \pm 1) = \frac{1}{1 + \exp(\mp 2\beta act_i^P)} \tag{3.8}$$

where β is a parameter determining the slope of the probability function. This leads to an expected firing rate of $\langle (o_j^P) \rangle = tanh(\beta \sum_j w_{ij} x_j)$ which can be used in the weight update rule $\Delta_P w_j = \gamma \delta^P x_j^P$, by setting $\delta^P = (t^P - \langle o_j^P \rangle)$ where the angled brackets indicate an average value over all input patterns.

3.7 Multiple Discrimination

So far we have only discussed networks in which there has been a single output neuron. Such a network can only differentiate between two classes. However by creating a network with more than one output neuron, we can create multiple decision surfaces for a single network. But note that such a machine is only possible if the individual classes are pairwise separable. Therefore to differentiate between N classes, there must exist N linear discriminant functions, $g_i(\mathbf{x},\mathbf{y})$, for i=1,...,N. Then we will believe that a particular $(\mathbf{x},\mathbf{y})$ is an instance of class i if $g_i(\mathbf{x},\mathbf{y}) > g_j(\mathbf{x},\mathbf{y}), \forall j \neq i$. Such a categorisation can be achieved with the simple binary perceptron, if we allocate a single output neuron to each class. Then the output of neuron i will be 1 if and only if $g_i(\mathbf{x},\mathbf{y}) > 0$ which will be the case for only inputs from class i. All other neurons will be -1 at this time i.e. $g_j(\mathbf{x},\mathbf{y}) < 0$ for all other neurons. This representation of the classes is a local representation since a single neuron is firing to show the classification to a single class. This can be contrasted with a distributed representation in which the class membership is shown by a pattern of firing over the set of output neurons.

3.8 Exercises

1. For the four patterns of the AND rule, use a simple perceptron (pencil and paper) learning rule with $\eta = 0.5$ to calculate the weights of the perceptron after presentation of the 4 patterns in the order second, fourth, third and first (see Figure 3.4). Let the initial values of the weights be w_0, the bias, $= 0.3$, $w_1 = -0.2$, $w_2 = 0.1$. (Objective 1, 2).

2. Repeat Question 1 with a LMS rule using an activation function of your choice. (Objectives 1,2, 3, 4, 5, 6, 8).

3. (Jagota, 1995) Consider all boolean functions $f : \{0,1\}^2 \rightarrow \{-1,1\}$. Which of them are linearly separable? For each of the linearly separable ones hand-construct a simple perceptron with two input units and one output unit with a hard threshold activation function. Use any real valued threshold θ for the output neuron. (Objective 7).

4. The XOR problem is the simplest of a set of problems known as the parity problems. e.g. for 6 dimensional inputs, $(100011)^T$ has odd parity. Therefore output -1.
   $(110011)^T$ has even parity. Therefore output 1.

   Draw a 3 dimensional unit cube and explain why the perceptron cannot solve the 3-D parity problem. (Objective 7).

5. (Jagota, 1995) Consider a simple perceptron with N input units and M output units, each with a hard threshold activation function (whose range is $\{-1,+1\}$) with threshold $\theta_i = 0$. Restrict each of the weights to have the values -1 or +1. How many functions from $\{-1,1\}^N$ to $\{-1,1\}^M$ are represented in this family of networks? Prove your answer. (Objective 2).

Chapter 4

The Multilayer Perceptron: backprop

4.1 Objectives

After this chapter, you should

1. understand the backpropagation algorithm.
2. be able to implement the backprop algorithm.
3. understand
   (a) batch vs on-line learning.
   (b) the importance of differentiable activation functions. .
   (c) methods of speeding convergence.
   (d) stopping criteria .
   (e) local minima.
   (f) generalisation.
4. be able to list potential applications of the algorithm.

4.2 Introduction

As we have seen, the Perceptron (and Adeline) proved to be powerful learning machines but there were certain mappings which were (and are) simply impossible using these networks. Such mappings are characterised by being linearly inseparable. Now it is possible to show that many linearly inseparable mappings may be modelled by multi-layered perceptrons; this indeed was known in the 1960s but what was not known was a rule which would allow such networks to learn the mapping. Such a rule appears to have been discovered independently several times[Werbos,1974; Parker, 1985] but has been spectacularly popularised by the PDP(Parallel Distributed Processing) Group [Rumelhart et al, 1986] under the name backpropagation.

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 clearly calculable by the previously described LMS rule; 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.

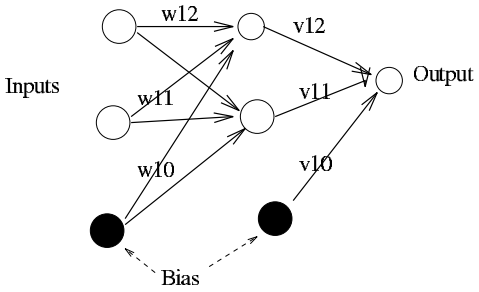

Figure 4.1: The net which will be used for the solution of the XOR problem using backpropagation

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.

4.3 The Backpropagation Algorithm

Because it is so important, we will repeat the whole algorithm in a 'how-to-do-it' form and then give a simple walk through for the algorithm when it is trained on the XOR problem:

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.

4.3.1 The XOR problem

You will use the net shown in Figure 4.1 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

4.4 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,

$$E = \sum_P E^P = \frac{1}{2}\sum_P (t^P - o^P)^2 \tag{4.1}$$

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,

$$\frac{\partial E^P}{\partial w_{ij}} = \frac{\partial E^P}{\partial Act_i^P} \cdot \frac{\partial Act_i^P}{\partial w_{ij}} \tag{4.2}$$

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

$$\Delta_P w_{ij} = \gamma . \delta_i^P . o_j^P \tag{4.3}$$

Note how like this rule is to that developed in the previous chapter (where the last o is replaced by the input vector, **x**). However, we still have to consider what values of δ are appropriate for individual neurons. We have

$$\delta_i^P = -\frac{\partial E^P}{\partial Act_i^P} = -\frac{\partial E^P}{\partial o_i^P} \cdot \frac{\partial o_i^P}{\partial Act_i^P} \tag{4.4}$$

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 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:

$$\frac{\partial E^P}{\partial o_i^P} = \sum_{j=1}^{N} \frac{\partial E^P}{\partial Act_j^P} \cdot \frac{\partial Act_j^P}{\partial o_i^P} = \sum_{j=1}^{N} \frac{\partial E^P}{\partial Act_j^P} \cdot \frac{\partial}{\partial o_i^P} \sum_{k=1}^{H} w_{jk} o_k^P = \sum_{j=1}^{N} \frac{\partial E^P}{\partial Act_j^P} . w_{ji} = -\sum_{j=1}^{N} \delta_j^P . w_{ji} \tag{4.5}$$

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

$$\delta_i^P = -\frac{\partial E^P}{\partial Act_j^P} = -\frac{\partial E^P}{\partial o_i^P} \cdot \frac{\partial o_i^P}{\partial Act_j^P} = \sum_{j=1}^{N} \delta_j^P w_{ji} . f'(Act_i^P) \tag{4.6}$$

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.

4.5 Issues in Backpropagation

4.5.1 Batch vs On-line Learning

The backpropagation algorithm is only theoretically guaranteed to converge if used in batch mode i.e. if all patterns in turn are presented to the network, the total error calculated and the weights updated in a separate stage at the end of each training epoch. However, it is more common to use the on-line (or pattern) version where the weights are updated after the presentation of each individual pattern. It has been found empirically that this leads to faster convergence though there is the theoretical possibility of entering a cycle of repeated changes. Thus in on-line mode we usually ensure that the patterns are presented to the network in a random and changing order.

The on-line algorithm has the advantage that it requires less storage space than the batch method. On the other hand the use of the batch mode is more accurate: the on-line algorithm will zig-zag its way to the final solution. It can be shown that the expected change (where the expectation is taken over all patterns) in weights using the on-line algorithm is equal to the batch change in weights.

4.5.2 Activation Functions

The most popular activation functions are the logistic function and the tanh() function. Both of these functions satisfy the basic criterion that they are differentiable. In addition, they are both monotonic and have the important property that their rate of change is greatest at an intermediate values and least at extreme values. This makes it possible to saturate a neuron's output at one or other of their extreme values. The final point worth noting is the ease with which their derivatives can be calculated:

- if f(x) = tanh(bx), then f'(a) = b(1 - f(a)*f(a));
- similarly, if f(x) = 1/(1+exp(-bx)) then f'(a)=bf(a)(1 - f(a)).

There is some evidence to suggest that convergence is faster when tanh() is used rather than the logistic function. Note that in each case the target function must be within the output range of the respective functions. If you have a wide spread of values which you wish to approximate, you must use a linear output layer.

4.5.3 Initialisation of the weights

The initial values of the weights will in many cases determine the final converged network's values. Consider an energy surface with a number of energy wells; then, if we are using a batch training method, the initial values of the weights constitute the only stochastic element within the training regime. Thus the network will converge to a particular value depending on the basin in which the original vector lies. There is a danger that, if the initial network values are sufficiently large, the network will initially lie in a basin with a small basin of attraction and a high local minimum; this will appear to the observer as a network with all weights at saturation points (typically 0 and 1 or +1 and -1). It is usual therefore to begin with small weights uniformly distributed inside a small range. Haykin recommends (p162) the range $(-\frac{2.4}{F_i}, +\frac{2.4}{F_i})$ where F_i is the fan-in of the i^{th} unit.

4.5.4 Momentum and Speed of Convergence

The basic backprop method described above is not known for its fast speed of convergence. Note that though we could simply increase the learning rate, this tends to introduce instability into the learning rule causing wild oscillations in the learned weights. It is possible to speed up the basic method in a number of ways. The simplest is to add a momentum term to the change of weights. The basic idea is to make the new change of weights large if it is in the direction of the previous changes of weights while if it is in a different direction make it smaller. Thus we use $\Delta w_{ij}(t+1) = (1-\alpha).\delta_j.o_i + \alpha\Delta w_{ij}(t)$, where the α determines the influence of the momentum. Clearly the momentum parameter α must be between 0 and 1. The second term is sometimes known as the 'flat spot avoidance' term since them momentum has the additonal property that it helps to slide the learning rule over local minima(see below).

4.5.5 Stopping Criteria

We must have a stopping criterion to decide when our network has solved the problem in hand. It is possible to stop when:

1. the Euclidean norm of the gradient vector reaches a sufficiently small value since we know that at the minimum value, the rate of change of the error surface with respect to the weight vector is zero. There are two disadvantages with this method:
   - it may lead to excessively long training times
   - it requires calculating the gradient vector of the error surface with respect to the weights
2. the rate of change of the mean squared error is sufficiently small
3. the mean squared error is sufficiently small

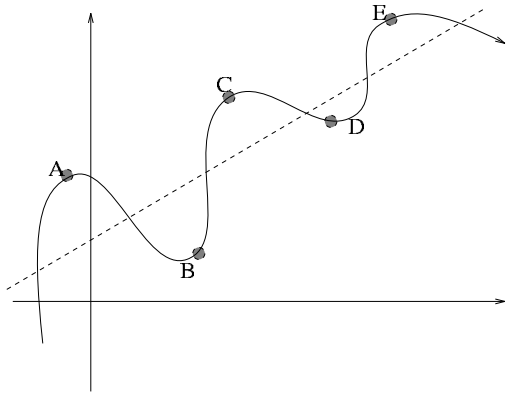

Figure 4.2: A set of data points may be approximated by either the straight line or the curve. Either would seem to fit the data; however, the line may give a better generalisation performance (on the test set) than the curve which is actually producing a lower error on the training set.

4. a mixture of the last two criteria

Typically we will stop the learning before each pattern has been perfectly learned (see Section 4.5.7) so that learning will stop when outputs are greater than 0.9 or less than 0.1.

4.5.6 Local Minima

Error descent is bedeviled with local minima. You may read that local minima are not much problem to ANNs, in that a network's weights will typically converge to solutions which, even if they are not globally optimal, are good enough. There is as yet little analytical evidence to support this belief. An heuristic often quoted is to ensure that the initial (random) weights are such that the average input to each neuron is approximately unity (or just below it). This suggests randomising the initial weights of neuron j around the value $\frac{1}{\sqrt{N}}$, where N is the number of weights into the j_{th} neuron. A second heuristic is to introduce a little random noise into the network either on the inputs or with respect to the weight changes. Such noise is typically decreased during the course of the simulation. This acts like an annealing schedule(see Chapter 7).

4.5.7 Weight Decay and Generalisation

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 since this is a measure of how well the network generalises. Remember the training set is composed of instances for which we already have the answer. We wish the network to give accurate results on data for which we do not already know the answer. There is a trade-off between accuracy on the training set and accuracy on the test set.

Note also that 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 4.2. 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 explicit 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 either remove connections explicitly or we may give each weight a tendancy to decay towards zero. The simplest method

is $w_{ij}^{new} = (1-\epsilon)w_{ij}^{old}$ after each update of the weights. This does have the disadvantage that it discourages the use of large weights in that a single large weight may be decayed more than a lot of small weights. More complex decay routines can be found which will encourage small weights to disappear.

4.5.8 Adaptive parameters

A heuristic sometimes used in practice is to assign learning rates to neurons in output layers a smaller value than those for hidden layers since the last layers usually have larger local gradients than the early layers and we wish all neurons to learn at the same rate.

Since it is not easy to choose the parameter values a priori one approach is to change them dynamically. e.g. if we are using too small a learning rate, we will find that the error E is decreasing consistently but by too little each time. If our learning rate is too large, we will find that the error is decreasing and increasing haphazardly. This suggests adapting the learning rate according to a schedule such as

$$\begin{aligned} \Delta\eta &= +a, \quad \text{if } \Delta E < 0 \text{ consistently} \\ \Delta\eta &= -b\eta, \quad \text{if } \Delta E > 0 \end{aligned} \tag{4.7}$$

4.5.9 The number of hidden neurons

The number of hidden nodes has a particularly large effect on the generalisation capability of the network: networks with too many weights (too many degrees of freedom) will tend to memorise the data; networks with too few will be unable to perform the task allocated to it. Therefore many algorithms have been derived to create neural networks with a smaller number of hidden neurons. Two obvious methods present themselves

1. Prune weights which are small in magnitude. Such weights can only be refining classifications which have already been made and are in danger of modelling the finest features of the input data
2. Grow networks till their performance is sufficiently good on the test set

4.6 Application - A classification problem

4.6.1 Theory

Consider an m-class classification problem. Let us create a network with m output neurons and require each neuron to output 1 when an input $\mathbf{x}$ is in its class and 0 otherwise. Now input pattern $\mathbf{x}^\mu$. Then the response of the trained MLP is $\mathbf{y}^\mu$ where

$$\mathbf{y}^\mu = (F_1(\mathbf{x}^\mu), F_2(\mathbf{x}^\mu), ..., F_m(\mathbf{x}^\mu)) \tag{4.8}$$

Then we can view the F values as forming an m-dimensional vector and write

$$\mathbf{y}^\mu = \mathbf{F}(\mathbf{x}^\mu) \tag{4.9}$$

where the function $\mathbf{F}$ is derived as a minimisation of the function

$$E = \frac{1}{2N}\sum_{\mu=1}^{N} \| \mathbf{t}^\mu - \mathbf{F}(\mathbf{x}^\mu) \|^2 \tag{4.10}$$

where $\mathbf{t}^\mu$ is the training signal when $\mathbf{x}^\mu$ is presented to the network. Note that $\mathbf{t}^\mu$ is an m-dimensional vector with 0 everywhere except the j^{th} position where j is the class of input $\mathbf{x}^\mu$. If we have enough training examples, then we can invoke the law of large numbers to show that the minimisation of the above function is equivalent to minimising the function over the whole input distribution. Also the optimum vector over all has the property that the converged weight $\mathbf{w}*$ is equal to the conditional expectation of the training examples, $E(\mathbf{t}^j|\mathbf{x})$, the *a posteriori* class probability of class j. This suggests the classification rule:

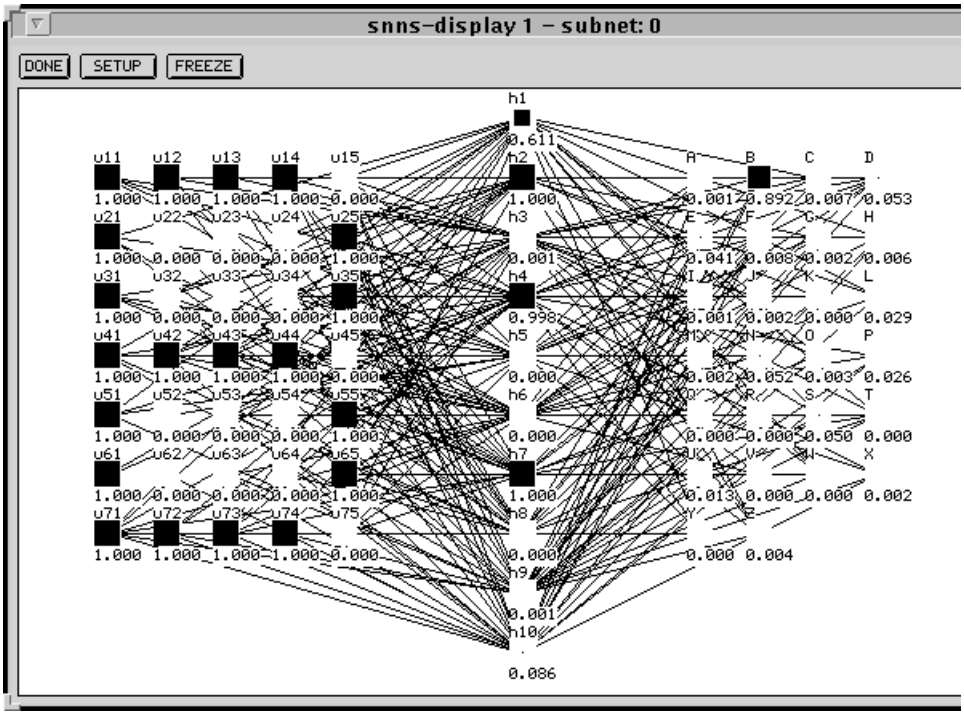

Figure 4.3: The Stuttgart Neural Network Simulator has been trained on the 26 letters. Here we show the response of the network to the presentation on the 5*7 display of the letter B

Classify input $\mathbf{x}$ in class C_j if

$$F_j(\mathbf{x}) > F_k(\mathbf{x}), \text{ for all } k \neq j \tag{4.11}$$

If the class distributions are disjoint, this will give us an unambiguous classification rule since we will have only a single neuron firing.

4.6.2 An Example

Using the Stuttgart Neural Network Simulator, a network was trained to identify all of the 26 roman letters, a,..,z presented on a 5*7 display. Training was stopped sufficiently early that some degree of generalisation was possible. Figure 4.3 shows the network's response during training to the letter B while Figure 4.4 shows the network's response to a noisy B.

Notice how large the effect of just one "pixel" change is on the network's output: the network is "hedging its bets" between classifying the letter as a B or a D. The single "pixel" was specifically chosen because it has just this effect - other "pixels" have a much smaller effect. This "pixel" was chosen for this demonstration precisely because it is so important to the classification process particularly to the differentiation of Bs and Ds. Of interest in this example also is the effect on the hidden neurons of the single pixel change in the input layer. Clearly this pixel was inhibiting pixels 1 and 10 in the hidden layer; in this case we can see quite explicitly how the network is differentiating between the two patterns B and D.

4.7 Linear Discrimination

A linear feedforward network with a single output neuron is defined by the rule

$$y = \mathbf{w}^T\mathbf{x} + w_0 \tag{4.12}$$

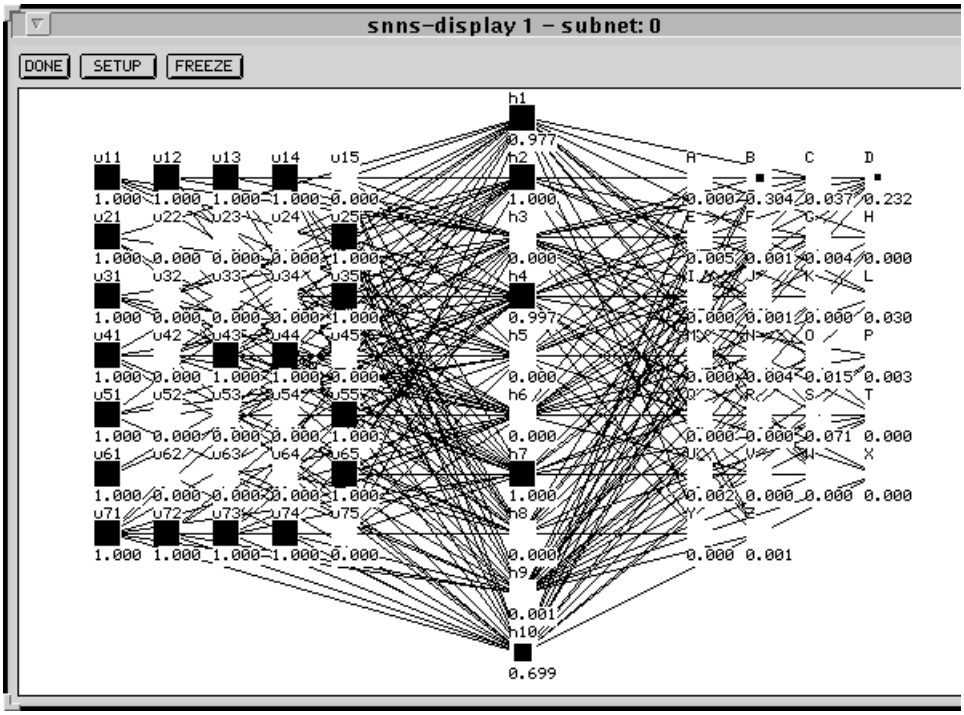

Figure 4.4: The input pattern B has been changed in just one pixel yet the effect on the network's response is fairly large. The network is responding maximally to the pattern as both B and D yet an observer could not be sure that it was seeing either

The parameter w_0 is known as the bias (or threshold).

We can view this as a discriminant function if we assign a vector $\mathbf{x}$ to class C_1 when $y > 0$ and to class C_2 when $y < 0$. It can be shown that if we have 2 classes whose instances are drawn from Gaussian distributions and if the classes have equal covariance matrices (which implies in particular that their variances are equal) this function is the best we can get to discriminate between the classes and y=0 is that line(actually hyperplane) on one side of which we have class C_1 and on the other side of which we have class C_2.

Then we have, on the discrimination line, $\mathbf{w}^T\mathbf{x} + w_0 = 0$. Now if $\mathbf{x}_A$ and $\mathbf{x}_B$ are any two points lying on the discrimination line, then $\mathbf{w}^T\mathbf{x}_A + w_0 = 0$ and $\mathbf{w}^T\mathbf{x}_B + w_0 = 0$ and so $\mathbf{w}^T(\mathbf{x}_A - \mathbf{x}_B) = 0$. So $\mathbf{w}$ is perpendicular to the discrimination line and so determines the orientation of the discrimination line. The w_0 parameter on the other hand determines the distance of the line from the origin (see Figure 4.5) since this distance, $\| \mathbf{x} \|$ is determined by

$$\begin{aligned} \| \mathbf{x} \| \| \mathbf{w} \| \cos\theta &= \mathbf{w}.\mathbf{x} = \mathbf{w}^T\mathbf{x} \\ \text{And so, } \| \mathbf{x} \| &= \frac{\mathbf{w}^T\mathbf{x}}{\| \mathbf{w} \|} = \frac{-w_0}{\| \mathbf{w} \|} \end{aligned}$$

So the line is completely determined by the weights. Alternatively, we can view the data as a set of 3 dimensional data comprising $(1, \mathbf{x})$ where the 1 is the bias. Now we have a two dimensional discrimination plane which passes through the origin in a 3 dimensional space (see Figure 4.6) and the classes are defined with respect to the intersection of the two planes as shown.

We can introduce several classes by having several output neurons each with its own weight vector. Then we have a set of boundaries (discrimination lines) between classes dependent on the weights and biases of the respective neurons. We use the simple rule that $\mathbf{x} \in C_i \iff i = arg\max_i y_i$.

It is readily shown that the regions formed are convex (i.e. if A and B are in a region, so is every point on the line AB). Consider Figure 4.7. Then let A be represented by the vector $\mathbf{x}_A$ and B be represented by

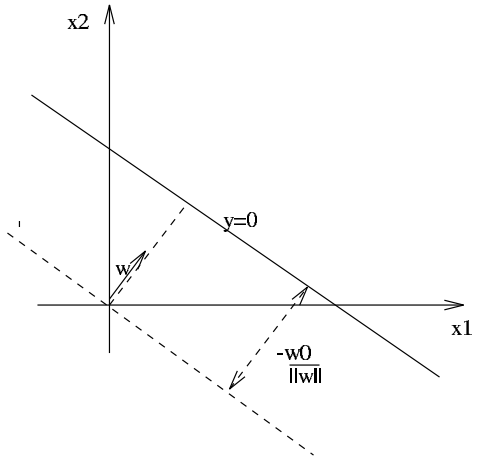

Figure 4.5: A linear discriminant line in a two dimensional space

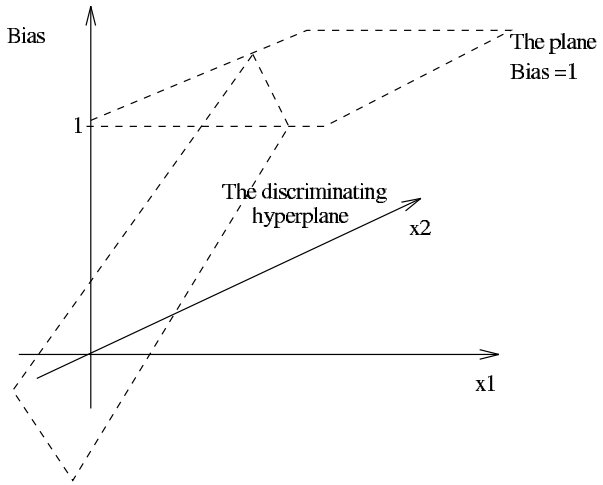

Figure 4.6: The slanting hyperplane discriminates between the two classes. We are interested in those (x_1, x_2) values which are on the plane Bias $=1$ which determines the classes

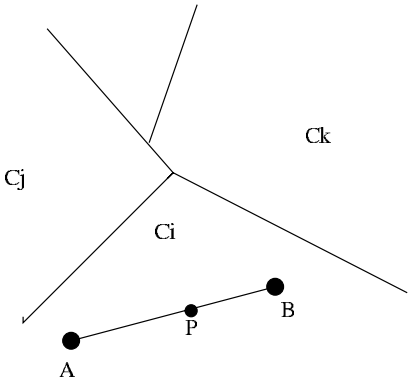

Figure 4.7: A representation of several classes which show convex regions

$\mathbf{x}_B$. Then P can be represented by

$$\mathbf{x}_P = \alpha \mathbf{x}_A + (1-\alpha)\mathbf{x}_B \tag{4.13}$$

where $0 \leq \alpha \leq 1$. Since both A and B are in C_i, they satisfy $y_i(\mathbf{x}_A) > y_j(\mathbf{x}_A), \forall j$ and $y_i(\mathbf{x}_B) > y_j(\mathbf{x}_B), \forall j$. So

$$\begin{aligned} y_i(\mathbf{x}_P) &= \alpha y_i(\mathbf{x}_A) + (1-\alpha) y_i(\mathbf{x}_B) \\ &> \alpha y_j(\mathbf{x}_A) + (1-\alpha) y_j(\mathbf{x}_B), \forall j \\ &= y_j(\mathbf{x}_P), \forall j \end{aligned}$$

So P is also classified as being in Class i.

Now the perceptron is guaranteed to converge to a line which will discriminate between classes if such a line can be found. The LMS rule can be shown to find the best linear discriminant between the classes under the conditions stated above. Notice that if we add a non-linearity into the network (such as the logistic function or tanh()) we still have a linear discriminant function (since these functions are monotonic). i.e. the network performing

$$y_i = f(\mathbf{w}_i^T \mathbf{x} + w_{i0}) \tag{4.14}$$

still gives a line (or hyperplane) between the classes where y=0 since

$$\begin{aligned} f(\mathbf{w}_i^T \mathbf{x}) &> f(\mathbf{w}_j^T \mathbf{x}) \\ \text{if and only if} & \\ \mathbf{w}_i^T \mathbf{x} &> \mathbf{w}_j^T \mathbf{x} \end{aligned}$$

though you will change the position of the discriminant function. This corresponds to the non-linear LMS rule from chapter 3.

4.7.1 Multilayered Perceptrons

Notice however that we can consider an alternative nonlinearity e.g.

$$y_i = \mathbf{w}_i^T \mathbf{f}(\mathbf{x}) \tag{4.15}$$

Here the function $\mathbf{f}()$ are also vectors. If we use different functions for each f() we call them basis functions and usually write for a single output neuron network

$$y = \sum_{j=0}^{M} w_j \phi_j(\mathbf{x}) \tag{4.16}$$

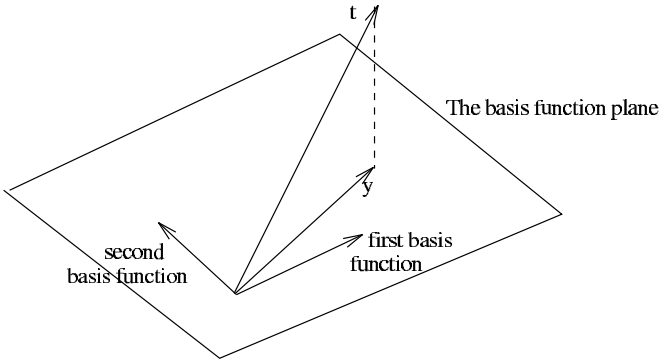

Figure 4.8: The target value, t, is outwith the basis function plane. Its best projection is y which can be learned by a simple LMS algorithm.

An example of this is the network defined by

$$y = \sum_{j=0}^{M} w_j \tanh(\mathbf{v}.\mathbf{x}) \tag{4.17}$$

Notice that this corresponds to a backpropagation network with weights $\mathbf{v}$ between the input and hidden neurons, a non-linear activation function at the hidden layer and an identity function for the output layer neurons. This type of backprop network can be shown to be capable of approximating any continuous function (or even a continuous function with a finite number of discontinuities). A geometric interpretation is shown in Figure 4.8. We can view the non-linearities in the hidden layer as forming a new basis (set of axes) for the data and then the feedforward through the network acts like a projection onto this new basis. Then the error between the output and the target values can be given by

$$E = \frac{1}{2} \parallel \sum_{j=0}^{M} w_j \phi_j(\mathbf{x}) - \mathbf{t} \parallel^2 \tag{4.18}$$

As usual we can minimise this by error descent using

$$\frac{\partial E}{\partial w_i} = \phi_i(\mathbf{x})(y - t) \tag{4.19}$$

Now if E can be made 0, it can only be when $(\mathbf{y} - \mathbf{t}) = 0$ i.e. (see Figure) when the target lies precisely in the basis function's plane. If there is a residual error it is caused by the continuing existence of a perpendicular distance between the basis function plane and some of the targets.

For the multi-output neuron case we can have

$$\frac{\partial E}{\partial w_{ij}} = \phi_i^T(\mathbf{x})(\mathbf{y} - \mathbf{t}) = 0 \tag{4.20}$$

i.e. the learning process stops when the difference between $\mathbf{y}$ and $\mathbf{t}$ is perpendicular to the basis vectors. i.e. in 2 dimensions, when $\mathbf{y}$ is directly under $\mathbf{t}$.

4.8 Function Approximation

We can approximate any continuous function using sets of tanh() sigmoid functions. Since we can make the tanh() function arbitrarily close to a step function, we can see the outline of an approximate proof in Figure 4.9. Then

$$f(x) \approx f(x_0) + \sum_{j=0}^{N} (f(x_{i+1}) - f(x_i)) H(x - x_i) \tag{4.21}$$

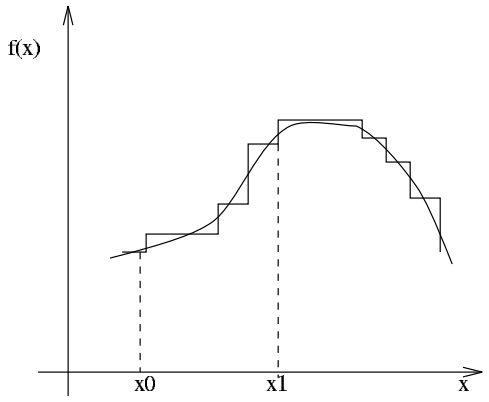

Figure 4.9: An approximation of a continuous function with a set of step functions.

where H() is the Heaviside function.

We can show the increasing accuracy of multi-layered perceptrons when we add more hidden neurons with the following small experiment: we use a noisy version of a simple trigonometric function and consider the set of points shown in Figure 8.4 which are drawn from $\sin(2\pi x)$ + noise. We train a multi-layered perceptron with linear output units and a tanh() nonlinearity in the hidden units. The results are shown in Figure 4.11.

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.

Clearly the network with a single sigmoid hidden neuron is not capable of modelling the sine function adequately whereas the nets with more hidden neurons are more successful. This might suggest that our best tactic is simply to throw lots of simple processing power (i.e. lots of hidden neurons) at a problem. So now we show the MLP with 15 hidden neurons (plus the bias) also responding to the data in Figure 4.12. It can be seen that the MLP is responding to the noise as much as the underlying signal in the data. It has sufficient degrees of freedom (the network weights) to "memorise" the data; it must be made to work harder in order to extract the underlying signal from the data and the way to do this is to cut down on the number of hidden neurons in the network.

4.8.1 A Prediction Problem

We are going to use a network such as shown in Figure 4.13 to predict the outcome of the next day's trading on the Stock Market. Our inputs to the network correspond to the closing prices of day (t-1), day (t-2), day (t-3) etc and we wish to output the closing price on day t. We have arbitrarily chosen to input 5 days information here and only used a training set of 100 days. (We will not make our fortune out of this network).

For this problem we choose a network which has a nonlinearity (actually tanh()) at the hidden layer and linear output neurons. We chose to make the output neuron linear since we wish it to take values in the range e.g. 0 - 3800 or whatever value the stock market might achieve. It would have been possible to have sigmoids in the output layer but we would then have had to scale the target values accordingly so that the network outputs could approximate the target values.

In practice, we find it easiest to take out the trend information from the raw data before feeding it to a network and so we adopt a very simple procedure of subtracting the previous days' data from the current days' data. Similarly with the outputs and targets. So what we are actually predicting is whether the market will go up or down and by how much.

We show in Figure 4.14 the results of using a single hidden neuron (plus a bias term at the hidden layer) on the financial data. The results are clearly not good since the network is only finding out that in general the market is going up.

As we might expect the results are very much dependent on the size of network chosen and so we show in Figure 4.15 a network with 5 hidden neurons. The results are clearly much better but again we must take

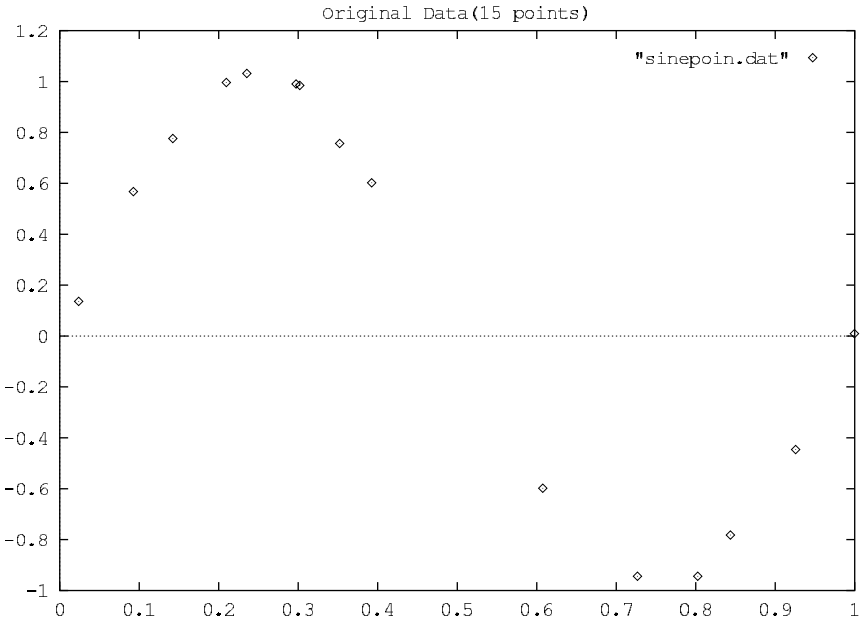

Figure 4.10: 15 data points drawn from a noisy version of $\sin(2\pi x)$.

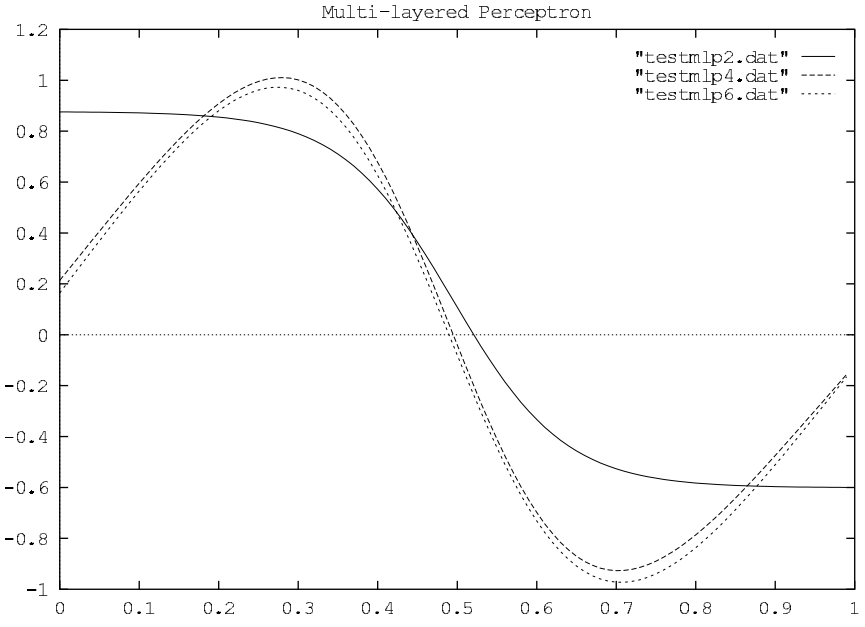

Figure 4.11: A comparison of the network convergence using multilayered perceptrons on the trigonometric data.

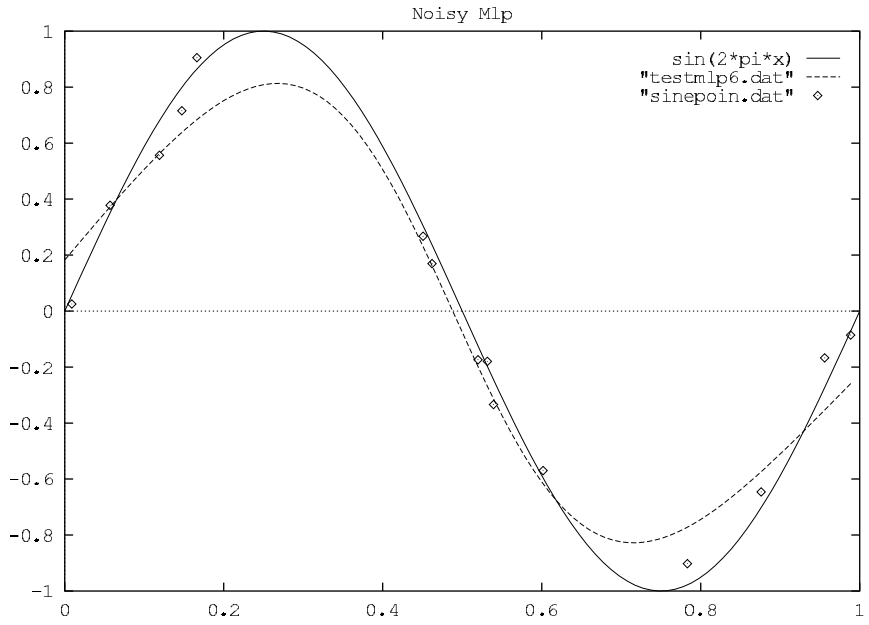

Figure 4.12: An MLP with 15 hidden neurons will also model the noise rather than the underlying signal.

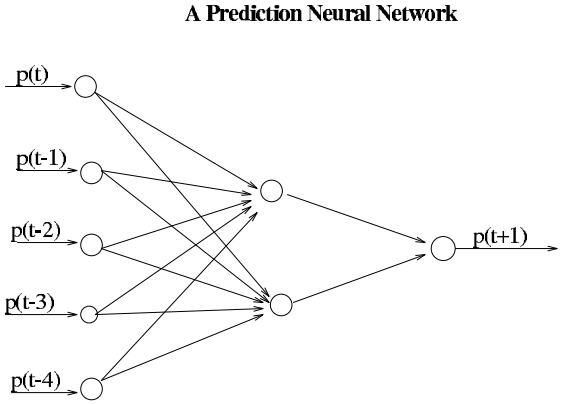

Figure 4.13: The inputs to the neural network comprise the last 5 day's closing prices and the network must predict the next day's closing prices.

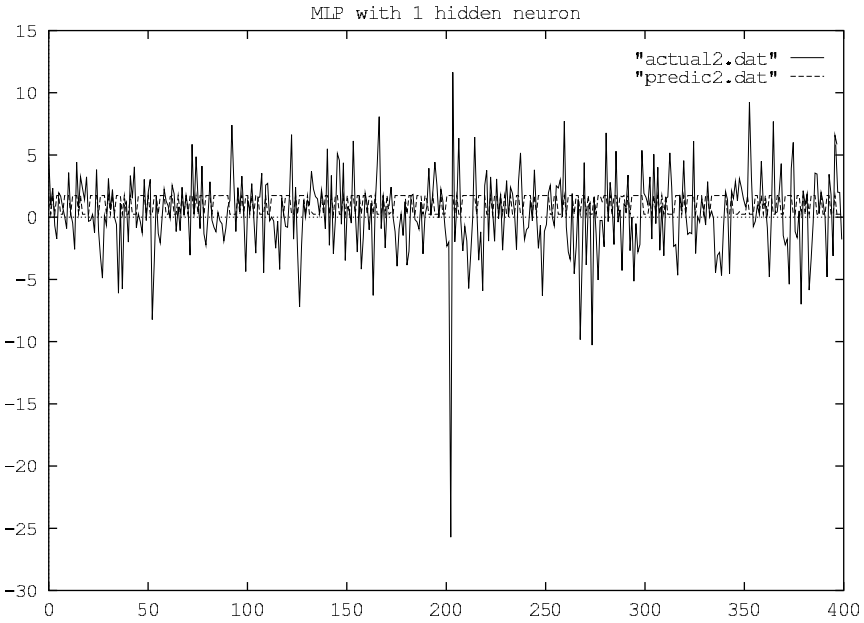

Figure 4.14: An MLP with a single non-linear hidden neuron is attempting to predict the closing price of the stock market on 100 days.

care about willy-nilly adding neurons since with 15 neurons the results (Figure 4.15) are very much worse (cf the sin() graph problem).

4.8.2 Practical Issues

You have been asked to create an artificial neural network to perform prediction on the FTSE index. What factors must you take into account? Some of the obvious ones are:

- Number of Inputs: we cannot specify in advance the number of days of inputs which are valuable in training the network optimally. So we would have to try different lengths of input vectors under the criteria below and find the best.
- Number of Hidden neurons: the smaller the better. Depends on the number of inputs etc.. We will use the least possible number of neurons which gives us a good predictor since if we add too many hidden neurons (== too many degrees of freedom) we will have an over-powerful neural network which will model the noise as well as any underlying series. Probably stick with a single hidden layer.
- Number of outputs is not an issue since we only wish one step look ahead.
- Output Activation Function will probably be linear since financial data is up to 4000 etc. Other possibility is to normalise the data between 0 and 1 and use logistic function.
- Learning rate: trial and error but almost certainly small (¡0.1). Maybe annealed to 0 during the course of the experiment which seems to give better accuracy.
- Split historical data into test set/training set. Division under a number of different regimes i.e. not just train on first 1000 samples, test on last 100. Stopping criterion: probably the least mean square error or least mean absolute error on the test set.
- Momentum: test a momentum term to see if we get better results.

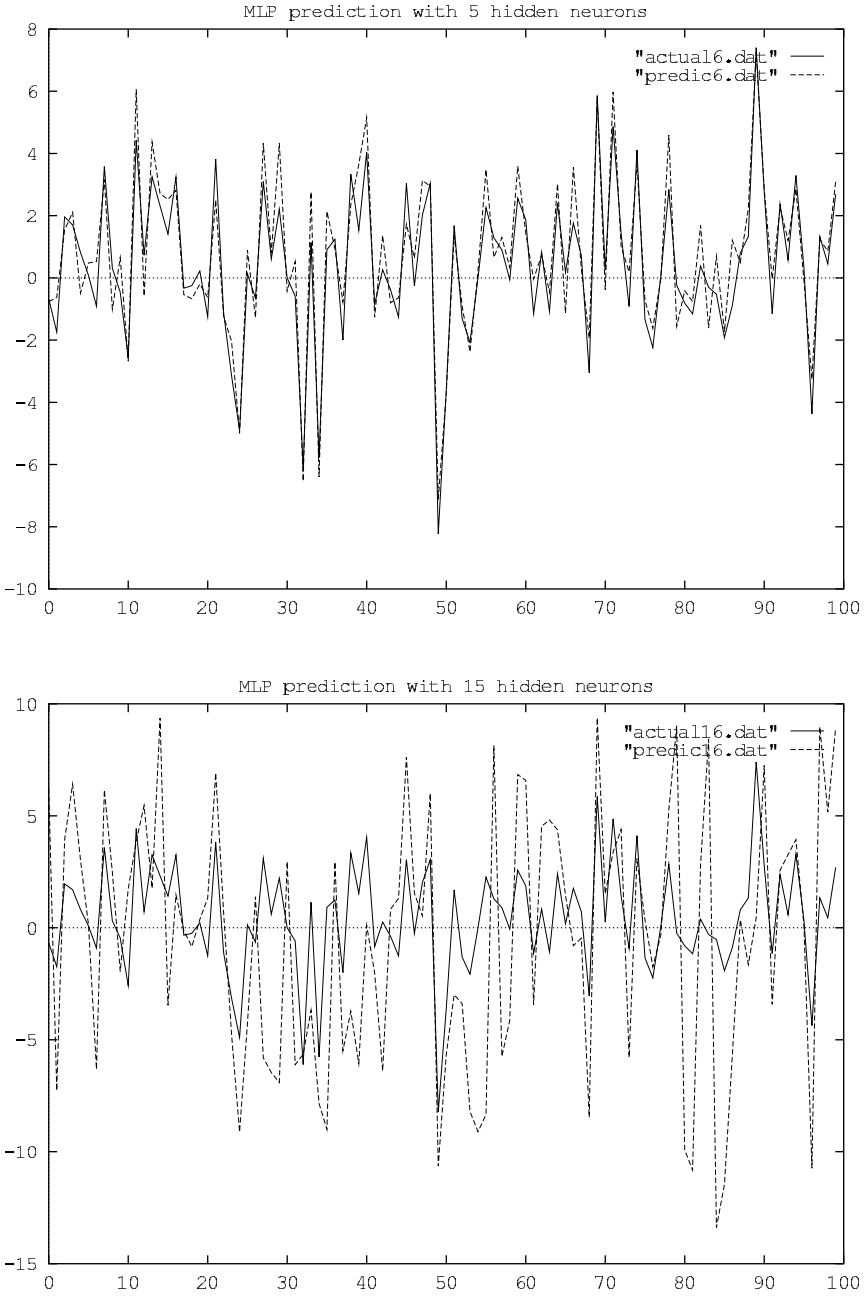

Figure 4.15: With 5 hidden neurons (plus a bias) the prediction properties of the network have vastly improved but with 15 hidden neurons the ntwork's powers of prediction have deteriorated again

4.9 Data Compression

We have stated that a network with no non-linear activation functions is no more use than a single layer linear network. There is one situation in which this is not quite true: when we have an auto-association network which has a hidden layer which is of smaller dimensionality than the input and output layers. In this specific case the hidden layer can be shown to form the optimal linear representation (the one which throws away least information) of the data. i.e. we are performing the optimal linear compression of the data.

We say that the network is performing a Principal Component Analysis. We will meet this in more detail in the next Chapter. For the time being consider a network which has been trained to autoassociate on a data set. Further let us consider the situation in which both we and a remote site have a copy of the network (or the data on which to train a network). Then if we wish to transmit that data with as few bits as possible losing as little information as possible we can feed the data to a network, extract the hidden neurons' activations and merely transmit them. When they are received at the remote site these activations can be inserted to the hidden nodes of *its* neural network and out pops the best (least-lossy) representation of the data.

4.10 Exercises

1. Consider the problem of data compression. We wish to create an 8-3-8 network which performs an encoding of the 8-dimensional input patterns into $\log_2(8) = 3-$dimensional hidden layer and which will then be decoded to an 8-dimensional output pattern. Use an autoassociation backpropagation network associating (1,0,0,0,0,0,0,0) with (1,0,0,0,0,0,0,0),
(0,1,0,0,0,0,0,0) with (0,1,0,0,0,0,0,0) etc.
Show a set of possible activations of the hidden values for each input pattern. (Objectives 4).

2. (Jagota,1995) Hidden units in multi-layer feedforward networks can be viewed as playing one or more of the following roles:

   (a) They serve as a set of basis functions

   (b) They serve as a means of transforming the input space into a more suitable space

   (c) Sigmoidal hidden units serve as approximations to linear threshold functions

   (d) Sigmoidal hidden units serve as feature detectors with soft boundaries

   Explain the benefits of the individual roles in helping us understand the theory and applications of multi-layer networks. Which roles help understand the theory and which roles ar insightful for which applications? (Objectives 1,3b, 3f).

3. Which of these two-layer feedforward networks is more powerful: one whose hidden units are sigmoidal or one whose hidden units are linear? The output units are linear in both cases. Explain your answer. (Objectives 1, 3b).

Chapter 5

Unsupervised learning

5.1 Objectives

After this Chapter, you should

1. understand the power of unsupervised learning.
2. be able to describe the problem with simple Hebbian learning and some solutions.
3. be able to describe anti-Hebbian learning and its uses.
4. be able to describe competitive learning and its variants.
5. be able to implement a Kohonen network, Art network and LVQ network.
6. be able to describe applications of competitive nets.

5.2 Unsupervised learing

With the networks which we have met so far, we must have a training set on which we already have the answers to the questions which we are going to pose to the network. Yet humans appear to be able to learn (indeed some would say can only learn) without explicit supervision. The aim of unsupervised learning is to mimic this aspect of human capabilities and hence this type of learning tends to use more biologically plausible methods than those using the error descent methods of the last two chapters. The network must self-organise and to do so, it must react to some aspect of the input data - typically either redundancy in the input data or clusters in the data; i.e. there must be some structure in the data to which it can respond. There are two major methods used

1. Hebbian learning
2. Competitive learning

We shall examine each of these in turn.

5.3 Hebbian learning

Hebbian learning is so-called after Donald Hebb who in 1949 conjectured:

> When an axon of a 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.

In the sort of feedforward neural networks we have been considering, this would be interpreted as the weight between an input neuron and an output neuron is very much strengthened when the input neuron's activation when passed forward to the output neuron causes the output neuron to fire strongly. We can see that the rule favours the strong: if the weights between inputs and outputs are already large (and so an input will have a strong effect on the outputs) the chances of the weights growing is large.

More formally, consider the simplest feedforward neural network which has a set of input neurons with associated input vector, $\mathbf{x}$, and a set of output neurons with associated output vector, $\mathbf{y}$. Then we have, as before, $y_i = \sum_j w_{ij} x_j$, where now the (Hebbian) learning rule is defined by $\Delta w_{ij} = \eta x_j y_i$. That is the weight between each input and output neuron is increased proportional to the magnitude of the simultaneous firing of these neurons.

We may note now that the simple associative memories which we met in Chapter 2 constitute a simple form of Hebbian learning.

Now we can substitute into the learning rule the value of y calculated by the feeding forward of the activity to get $\Delta w_{ij} = \eta x_j \sum_k w_{ki} x_k = \eta \sum_k w_{ki} x_k x_j$.

Writing the learning rule in this way emphasises the statistical properties of the learning rule i.e. that the learning rule depends on the correlation between different parts of the input data's vector components. It does however, also show that we have a difficulty with the basic rule as it stands which is that we have a positive feedback rule which has an associated difficulty with lack of stability: if the i^{th} input and j^{th} output neurons are tending to fire strongly together, the weight between them will tend to grow strongly; if the weight grows strongly, the j^{th} output neuron will fire more strongly the next time the i^{th} input neuron fires and this will cause an increased value in the rate of change of the weights which will.... If we do not take some preventative measure, the weights will grow without bound. Such preventative measures include

1. Clipping the weights i.e. insisting that there is a maximum, w_{max} and minimum w_{min} within which the weights must remain.
2. Specifically normalising the weights after each update. i.e. $\Delta w_{ij} = \eta x_j y_i$ is followed by $w_{ij} = \frac{w_{ij}+\Delta w_{ij}}{\sqrt{(\sum_k w_{ik}+\Delta w_{ik})^2}}$ which ensures that the weights into each output neuron have length 1.
3. Having a weight decay term within the learing rule to stop it growing too large e.g. $\Delta w_{ij} = \eta x_j y_i - \gamma(w_{ij})$, where the decay function $\gamma(w_{ij})$ represents a monotonically increasing function of the weights i.e. if the weights grow large so does the weight decay term.
4. Create a network containing a negative feedback of activation

5.3.1 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".

Linsker's network is shown in Figure 5.1. 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. At the final layer, lateral connections within the layer are allowed.

The Hebb-type learning rule is

$$\Delta w_{ij} = a(x_i - < x >)(y_j - < y >) + b$$

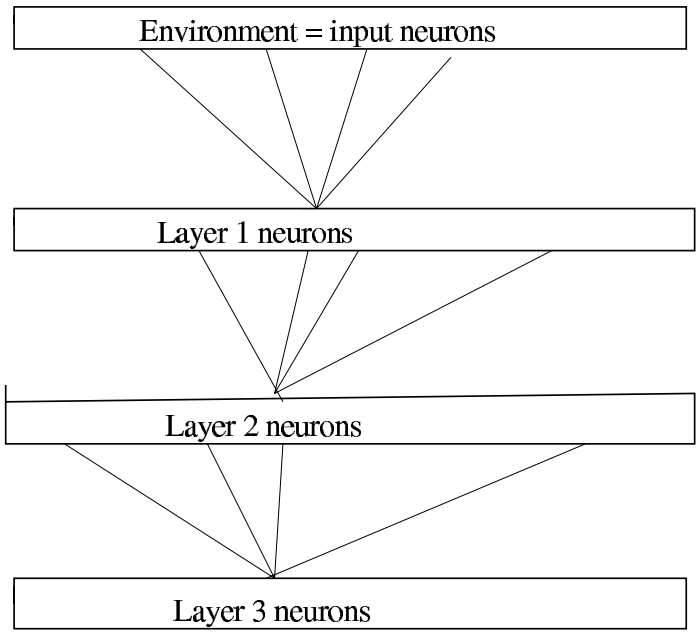

Figure 5.1: Linsker's model

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 $w^- = -w^+$.

We see that the learning rule is equivalent to a Hebbian learning rule with some decay terms based on the expected values of the inputs and outputs.

$$\Delta w_{ij} = a x_i y_j - a < x > y_j - a < y > (x_i - < x >)$$

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.

5.3.2 The Stability of the Hebbian Learning Rule

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

In other words, multiplying the vector $\mathbf{x}$ or any of its multiples by 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,

$$0 = \langle \Delta w_i^* \rangle = \langle y x_i \rangle = \langle \sum_j w_j x_j x_i \rangle = \sum_j R_{ij} w_j \tag{5.1}$$

where the anled brackets indicate the expected value taken over the whole distribution and R is the correlation matrix of the distribution. Now this happens for all i , so $R\mathbf{w}^* = 0$.

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 **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

$$\langle \Delta \mathbf{w}^* \rangle = R(\mathbf{w}^* + \epsilon) = R\epsilon > 0 \tag{5.2}$$

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. In fact, in time, the weights of nets which use simple Hebbian learning tend to be dominated by the direction corresponding to the largest eigenvalue.

5.4 Hebbian Learning and Information Theory

5.4.1 Quantification of Information

Shannon devised a measure of the information content of an event in terms of the probability of the event happening. He wished to make precise 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.

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

$$H = -\sum_{i=1}^{N} p_i \log(p_i) \tag{5.3}$$

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.

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.

5.4.2 Principal Component Analysis

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. A useful tactic is to find that direction which accounts for most of the data's variance - this becomes the first basis vector. 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 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 defined by $c_{ij} = \langle (x_i - \langle x \rangle)(x_j - \langle x \rangle) \rangle$ where the angled brackets indicate an ensemble average. 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

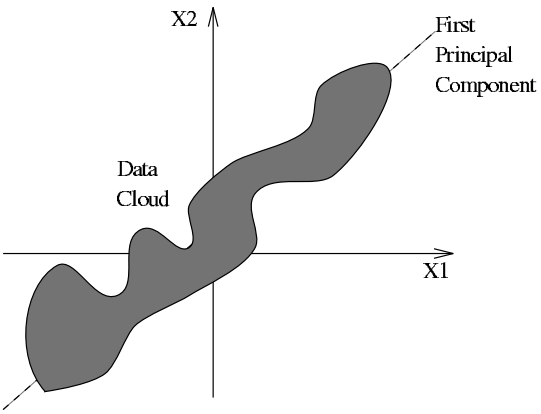

Figure 5.2: A cloud of data points and the corresponding first Principal Component

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 5.2 shows a simple example of a principal component analysis. The original data is two dimensional (in the X1,X2 plane). If you wished to know the position of each data point as accurately as possible using only a single coordinate, you would be best to give each point's position in terms of the First Principal Component shown in the diagram. We can see that the first PC is the direction of greatest spread (variance) of the input 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. 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 $E = \sum_{i=m+1}^{N} \lambda_i$.

5.4.3 Weight Decay in Hebbian Learning

As noted earlier, 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.

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:

$$\frac{dw_{ij}}{dt} = \eta y_i x_j - w_{ij} \tag{5.4}$$

It is clear that the weights will be stable (when $\frac{dw_{ij}}{dt} = 0$) at the points where $w_{ij} = \eta \langle y_i x_j \rangle$ where the angled brackets indicate an ensemble average. Using a similar type of argument to that employed for simple Hebbian learning, we see that at convergence we must have $\eta C\mathbf{w} = \mathbf{w}$. Thus $\mathbf{w}$ would have to be an eigenvector of the correlation matrix of the input data with corresponding eigenvalue $\frac{1}{\eta}$. (In fact, $\mathbf{w} = 0$ is a possible solution, but, as before, we can show that this is an unstable solution).

5.4.4 Oja's One Neuron Model

Oja developed a model of Hebbian learning with weight decay which not only stopped the weights growing without bound, it also caused the weights to converge to the Principal Components of the input data. In other words, the weight decay had the amazing effect of not only causing the weight increase to stabilise,

it also caused the convergence of the weights to values that extracted the maximum information from the input data.

He used a single output neuron which sums the inputs in the usual fashion

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

but used a Hebbian learning neuron with weight decay,

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

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.

5.4.5 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

$$\Delta w_{ij} = \alpha(x_i y_j - y_j \sum_k w_{ik} y_k) \tag{5.5}$$

which has been shown to force the weights to converge to a basis of the Principal Subspace. In other words, this network will act so that, depending on the number of output neurons, the network weights will converge on all the major information filters possible.

One advantage of this model 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. This need not be a disadvantage for biological networks but for engineering applications it is preferable to be able to find the actual Principal Components themselves.

5.4.6 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

$$y_j = \sum_{i=1}^{n} w_{ij} x_i$$

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

$$\Delta w_{ij} = \eta y_j (x_i - \theta_j \sum_{k=1}^{N} y_k w_{kj})$$

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. Now we have a network which will extract the direction with maximum information using output neuron 1, the direction of second greatest information using output neuron 2 and so on.

5.4.7 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:

$$\Delta w_{ij} = \alpha(x_i y_j - y_j \sum_{k=1}^{j} w_{ik} y_k) \tag{5.6}$$

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

$$\begin{aligned} \Delta w_{ij} &= \alpha(x_i y_j - y_j \sum_{k=1}^{j-1} w_{ik} y_k - y_j w_{ij} y_j) \\ &= \alpha([x_i y_j - y_j \sum_{k=1}^{j-1} w_{ik} y_k] - y_j^2 w_{ij}) \\ &= \alpha([x_i - \sum_{k=1}^{j-1} w_{ik} y_k] y_j - y_j^2 w_{ij}) \end{aligned} \tag{5.7}$$

Now the term inside brackets is the projection of the output neurons' values onto the first i-1 principal components; it is basically stating that the first neurons have captured those (i-1) directions with most information. We can look at the effect on the weights into the first output neuron (Neuron 1):

$$\begin{aligned} \Delta w_{1i} &= \alpha(x_i y_1 - y_1 \sum_{k=1}^{1} w_{ki} y_k) \\ &= \alpha(x_i y_1 - y_1 w_{1i} y_1) \\ &= \alpha(x_i y_1 - w_{1i} y_1^2) \end{aligned} \tag{5.8}$$

So the first principal component will be grabbed by the first output neuron's weights since this is exactly Oja's one neuron rule.

Now the second output neuron is using the rule

$$\begin{aligned} \Delta w_{2i} &= \alpha(x_i y_2 - y_2 \sum_{k=1}^{2} w_{ki} y_k) \\ &= \alpha(x_i y_2 - y_2 w_{1i} y_1 - y_2 w_{2i} y_2) \\ &= \alpha([x_i - w_{1i} y_1] y_2 - w_{2i} y_2^2) \\ &= \alpha(x_i' y_2 - w_{2i} y_2^2) \end{aligned}$$

Now the bit inside the square brackets is the original x-value minus the projection of the first y (output) neuron on the first principal component. This subtracts out the direction of greatest information (the first principal component) and leaves the second neuron performing Oja's one neuron rule on the subspace left after the first principal component has been removed i.e. the x' space. It will then converge to the second principal component.

The same argument holds for the third output neuron:

$$\begin{aligned}\Delta w_{3i} &= \alpha(x_i y_3 - y_3 \sum_{k=1}^{3} w_{ki} y_k) \\ &= \alpha(x_i y_3 - y_3 w_{1i} y_1 - y_3 w_{2i} y_2 - y_3 w_{3i} y_3) \\ &= \alpha([x_i - w_{1i} y_1 - w_{2i} y_2] y_3 - w_{3i} y_3^2)) \\ &= \alpha(x_i'' y_3 - w_{3i} y_3^2)\end{aligned}$$

where x'' is the remainder after the first two directions of most information are subtracted.

In general, 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. 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.

5.4.8 Summary of Hebbian Learning

Hebbian learning is rarely used in its simple form; typically a weight decay term is included in the learning rule which has the property that it stops the weights growing indefinitely. If the decay term is of the correct form, it also causes the weights to converge to the principal component directions. In other words, we now have a very simple neural network which is extracting as much information as possible from a set of raw data in an unsupervised manner. The network is responding solely to the redundancy in the input data to find an optimal compression of the input data which retains as much information in the data as possible in as few dimensions as possible. This is an important property for any information processing machine -carbon or silicon - to have.

5.4.9 Applications

Principal component networks are primarily useful when we wish to compress data losing as little of the information in the data as possible. Then by projecting the individual items of raw data onto the principal components (of the ensemble of input data) we will on average be able to represent the data most accurately in as concise a form as possible. A second use for such networks is in exploratory data investigations. If we have to search for structure in data in a high-dimensional space it cannot be done by eye and the automatic identification of structure may be inhibited by the high-dimensionality of the data. By projecting the data onto the lower dimensional subspace spanned by the principal components we hope to retain as much structure in the data as possible while making it easier to spot the structure. Finally principal component networks are finding uses in preprocessing data for other networks (such as the backprop network of the last chapter) since the principal components directions are orthogonal to one another we do not have interference between the sets of patterns in each direction. Thus projecting inputs onto the principal component directions and then using a standard multi-layer perceptron can greatly speed the perceptron's learning.

A typical application of principal component networks has been in the area of image coding in which a set of images can require an extremely high bandwidth channel in order to be transmitted perfectly. However the human eye does not require (nor can it handle) infinite precision and so we may transmit images over a lossy channel. The optimal linear compression of the image data is that determined by the projection of the data onto its principal components. We may then receive image data which is good enough for human recognition though it has lost some of its precision in the transmission.

5.5 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.

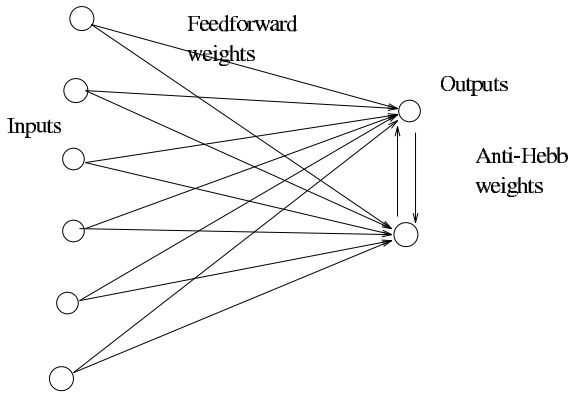

Figure 5.3: Anti-Hebbian Weights
Negative decorrelating weights between neurons in the same layer are learned using an "anti-Hebbian" learning rule

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 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.

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.

$$(\langle y_i . y_j \rangle \to 0) \Longrightarrow (w_{ij} \to 0) \tag{5.9}$$

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

It has been shown that not only does anti-Hebbian learning force convergence in the particular case of a deflationary algorithm but that the lateral connections do indeed vanish.

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 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. This is true only for noise-free information-transfer since if there is some noise in the system, some duplication of information may be beneficial to optimal information transfer.

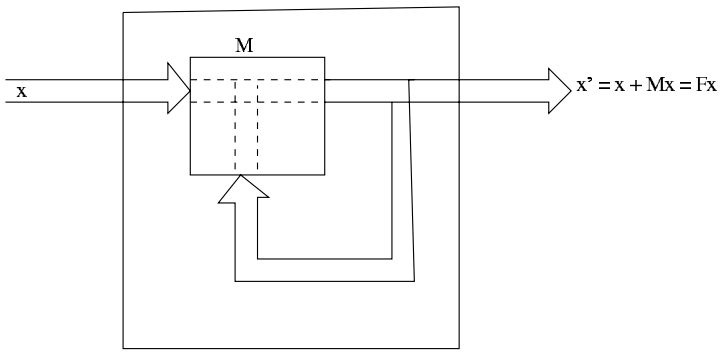

Figure 5.4: The System Model of the Novelty Filter

5.5.1 The Novelty Filter

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 5.4). Here we have an input vector $\mathbf{x}$ which generates feedback gain by the vector of weights, M. Each element of M is adapted using anti-Hebbian learning:

$$\frac{dm_{ij}}{dt} = -\alpha x_i^{'} x_j^{'} \tag{5.10}$$

$$\text{where } \mathbf{x}^{'} = \mathbf{x} + \mathbf{M}\mathbf{x}^{'} \tag{5.11}$$

$$= (\mathbf{I} - \mathbf{M})^{-1}\mathbf{x} = \mathbf{F}\mathbf{x} \tag{5.12}$$

"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 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 $\mathbf{x_1}$.

Thus the "Novelty Filter" knows what patterns it has seen and extracts information from any pattern which is equivalent to patterns which it has seen; the residuals are the "novelties".

5.6 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.

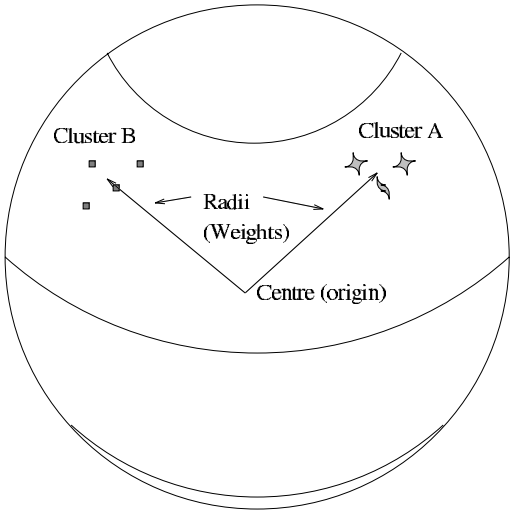

Figure 5.5: The input vectors are represented by points on the surface of a sphere and the lines represent the directions to which the weights have converged. Each is pointing to the mean of the group of vectors surrounding it.

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 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.

5.6.1 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 over a distribution, the weights will tend to the mean value of the distribution since $\Delta w_{ij} \to 0 \Longleftrightarrow w_{ij} \to \langle x_j \rangle$, where the angled brackets indicate the ensemble average. 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 5.5: 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. We now consider three important variations on Competitive learning:

1. Learning Vector Quantisation
2. The ART models
3. The Kohonen feature map

5.6.2 Learning Vector Quantisation

In vector quantisation, we create a codebook of vectors which represent as well as possible the vectors of each class. One obvious way to represent vectors is to use the mean of each class of vectors and competitive learning can be used to find the mean. If we use this method we can be sure that there is no vector codebook of equal length which is better able to represent the input vectors.

Kohonen has suggested a *supervised* quantisation method called learning vector quantisation (LVQ) which means that the classes must be known in advance (so that we have a set of labelled input data). We feedforward the activation through the weights as before but now we are in a position to tell if the correct unit has won the competition. If it has, we use the same learning rule as before but if neuron i has won the competition erroneously we move its vector of weights away from the input pattern. Formally the learning rule becomes

$$\begin{aligned} \Delta w_{ij} &= \eta(x_j - w_{ij}) \text{ , if class i is correct} \\ \Delta w_{ij} &= -\eta(x_j - w_{ij}) \text{ , if class i is wrong} \end{aligned}$$

An improved algorithm called LVQ2 only changes the weights negatively if

1. the input vector is misclassified *and*
2. the next nearest neighbour is the correct class *and*
3. the input vector is reasonably close to the decision boundary between the weights

Note that this is a supervised learning method but we include it in this chapter since it is a variant of Competitive learning.

5.6.3 ART Models

There are a number of ART models all developed in response to what Grossberg calls the **stability-plasticity dilemma**. This refers to the conflicting desires for our neural networks to remain stable in the condition to which they have converged (i.e. to retain their memories of what has been learned) while at the same time being receptive to new learning. Typically we ensure stability by reducing the learning rate during the course of a simulation but this has the obvious effect that the network cannot then respond e.g. to changing input distributions.

The ART models are based on having a mixture of top-down and bottom-up competition. The simplest model can be described as:

1. Select randomly an input from the input distribution
2. Propagate its activation forwards

3. Select the output neuron with greatest activation. If the activation is not high enough, create a new neuron and make its weights equal to the normalised input pattern, $\mathbf{x}$. Go back to step 1.

4. Check that the match between the winning neuron and the input pattern is good enough by calculating the ratio $\frac{\mathbf{w}_i.\mathbf{x}}{\|\mathbf{x}\|}$. If this does not exceed the vigilance parameter, ρ, the winning unit is discarded as not good enough. It is disabled and the competition is carried out again with the same input over all other output neurons (i.e. go back to step 3).

5. Adjust the weights of the winning neuron using the simple competitive learning rule.

Notice how the algorithm solves the stability-plasticity problem: in order that it can continue to learn we assume a set of unused neurons which are created as needed. On the other hand, once a neuron has learned a set of patterns (i.e. is responding maximally to those patterns) it will continue to respond to them and to patterns like them. It is possible to show that all weight changes stop after a finite number of steps.

Note also the interaction between the two criteria:

- The winning neuron is that which is maximally firing
- The vigilance parameter checks whether the winning neuron is actually close enough to the input pattern.

It is the interaction between these two criteria which gives ART its power. We are actually searching through the prototype vectors to find one which is close enough to the current input to be deemed a good match. A local algorithm with top-down (checking vigilance) and bottom-up (finding winning neuron) neurons has been created.

One major difficulty with ART networks is that convergence can be extremely slow for any problem greater than a toy problem.

5.6.4 The Kohonen Feature Map

The interest in feature maps stems directly from their biological importance. A feature map uses the "physical layout" of the output neurons to model some feature of the input space. In particular, if two inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ are close together with respect to some distance measure in the input space, then if they cause output neurons y_a and y_b to fire respectively, y_a and y_b must be close together in some layout of the output neurons. Further we can state that the opposite should hold: if y_a and y_b are close together in the output layer, then those inputs which cause y_a and y_b to fire should be close together in the input space. When these two conditions hold, we have a feature map. Such maps are also called **topology preserving maps.**

Examples of such maps in biology include

the retinotopic map which takes input from the retina (at the eye) and maps it onto the visual cortex (back of the brain) in a two dimensional map

the somatosensory map which maps our touch centres on the skin to the somatosensory cortex

the tonotopic map which maps the responses of our ears to the auditory cortex.

Each of these maps is believed to be determined genetically but refined by usage. e.g. the retinotopic map is very different if one eye is excluded from seeing during particular periods of development.

Hertz *et al* distinguish between

- those maps which map continuous inputs from single(such as one ear) inputs or a small number of inputs to a map in which similar inputs cause firings on neighbouring outputs (left half of Figure 5.6)
- with those maps which take in a broad array of inputs and map onto a second array of outputs (right half of Figure 5.6).

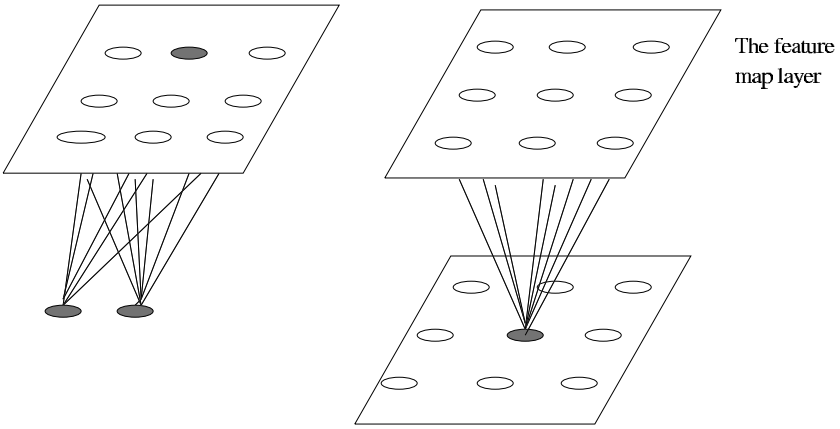

Figure 5.6: Two types of feature maps (a) a map from a small number of continuous inputs (b) a map from one layer spatially arranged to another

There are several ways of creating feature maps - the most popular is Kohonen's.

Kohonen's algorithm is exceedingly simple - the network is a simple 2-layer network and competition takes place between the output neurons; however now not only are the weights into the winning neuron updated but also the weights into its neighbours. Kohonen defined a neighbourhood function $f(i, i^*)$ of the winning neuron i^*. The neighbourhood function is a function of the distance between i and i^*. A typical function is the Difference of Gaussians function; thus if unit i is at point $\mathbf{r}_i$ in the output layer then

$$f(i,i^*) = a\exp(\frac{-|r_i - r_{i^*}|^2}{2\sigma^2}) - b\exp(\frac{-|r_i - r_{i^*}|^2}{2\sigma_1^2}) \tag{5.13}$$

The single Gaussian can be seen in Figure 5.7 while the Mexican Hat function is shown in Figure 5.8. Notice that this means that a winning neurons' chums - those neurons which are "close" to the winning neuron in the output space - are also dragged out to the input data while those neurons further away are pushed slightly in the opposite direction.

Results from an example experiment is shown in Figure 5.9. The experiment consists of a neural network with two inputs and twenty five outputs. The two inputs at each iteration are drawn from a uniform distribution over the square from -1 to 1 in two directions e.g typical inputs would be (0.3,0.5), (-0.4,0.9), (0.8,0.8) or (-0.1,-0.5). The algorithm is

1. Select at random an input point (with two values).

2. There is a competition among the output neurons. That neuron whose weights are closest to the input data point wins the competition:

$$\text{winning neuron, } i^* = \arg\min(\| \mathbf{x} - \mathbf{w_i} \|) \tag{5.14}$$

   Use the distance formula to find that distance which is smallest,

$$dist = \sqrt{(x_1 - w_{i1})^2 + (x_2 - w_{i2})^2} \tag{5.15}$$

3. Now update all neurons' weights using

$$\Delta w_{ij} = \alpha(x_j - w_{ij}) * f(i, i^*) \tag{5.16}$$

   where

$$f(i,i^*) = a\exp(\frac{-|r_i - r_{i^*}|^2}{2\sigma^2}) - b\exp(\frac{-|r_i - r_{i^*}|^2}{2\sigma_1^2}) \tag{5.17}$$

4. Go back to the start.

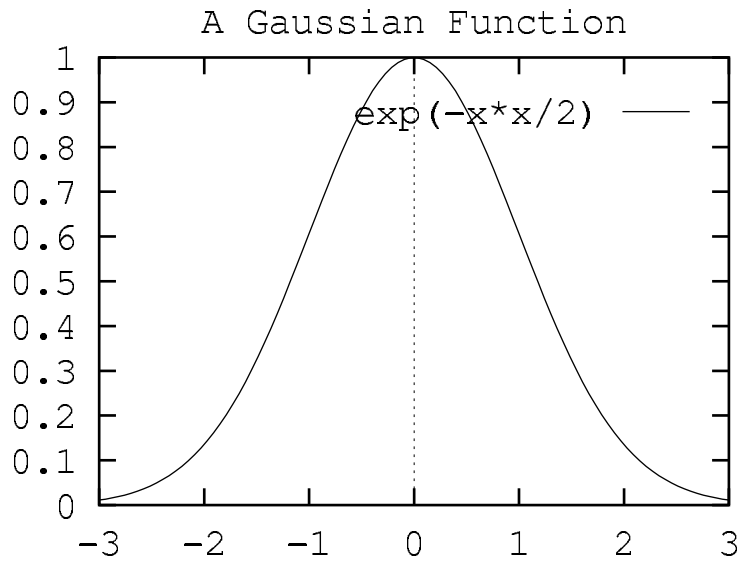

Figure 5.7: The Gaussian function.

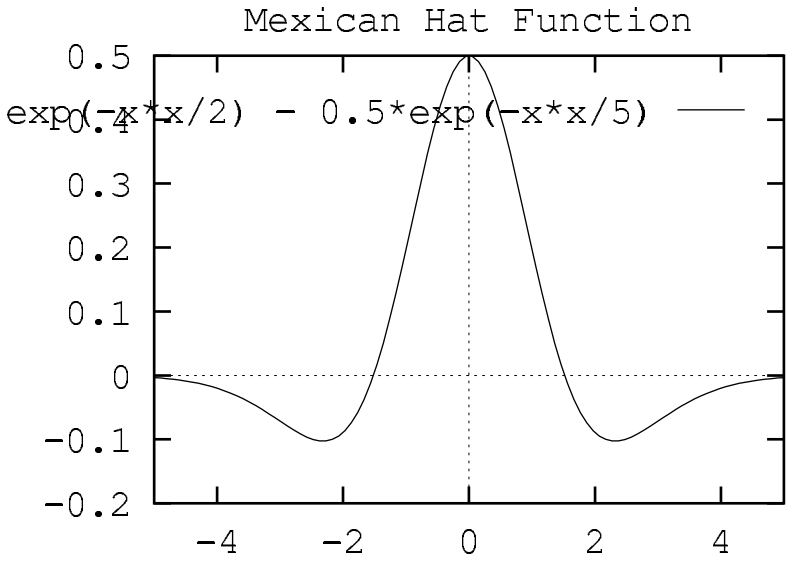

Figure 5.8: The difference of Gaussian function.

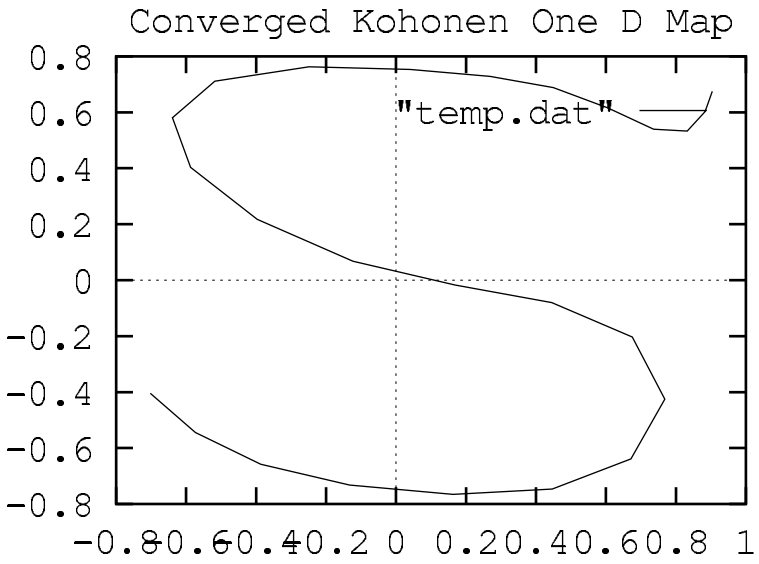

Figure 5.9: A one dimensional mapping of the two dimensional input space

Kohonen typically keeps the learning rate constant for the first 1000 iterations or so and then slowly decreases it to zero over the remainder of the experiment (we can be talking about 100 000 iterations for self-organising maps). Two dimensional maps can be created by imagining the output neurons laid out on a rectangular grid (we then require a two D neighbourhood function) or sometimes a hexagonal grid. An example of such a mapping is shown in Figure 5.10 in which we are making a two dimensional mapping of a two dimensional surface. This is a little redundant and is for illustration only.

Like the ART algorithms, Kohonen feature maps can take a long while to converge. Examples of the converged mappings which a Kohonen SOFM can find are shown in Figure 5.10. We can see that the neurons have spread out evenly when the input distribution is uniform over a rectangular space but with the less regular distribution in the right half of the figure, there are several dead neurons which are performing no useful function - they will respond to areas of the input space which are empty.

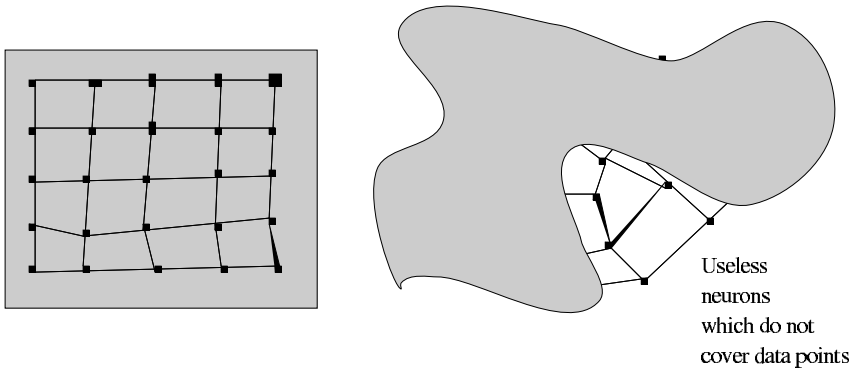

Figure 5.10: A 5*5 feature map which conforms well to a set of inputs drawn from a rectangular distribution but less well to a less regular distribution

5.7 Applications

The SOFM has been used in modelling features of the human brain such as visual maps and somatosensory maps. In terms of engineered applications some of interest are

1. Vector quantisation in which we wish to identify all vectors close to a particular prototypical vector of a particular class with the prototype. This can be useful e.g. in terms of data compression since if we transmit the code for the prototype vector (i.e. the information that it is prototype 3 rather than 2 or 4) we may be giving enough information for the vector to be reconstructed at the receiving location with very little loss of information. This can be very useful e.g. in image reconstruction where the bandwidth necessary to fully reconstruct an image may be prohibitively high.
2. Control of robot arms. Inverse kinematics problems can be extremely computationally expensive to solve. A SOFM allows an unsupervised learning of a set of standard positions of the robot end-receptors in terms of the positions of the individual angles and lengths of levers which make up a robot arm.
3. Classification of e.g. clouds from raw data.

5.8 Exercises

1. Hebbian learning is sometimes defined in terms of a covariance-type rule:

$$\Delta w_{ij}^p = \eta(y_j^p - \langle y_j \rangle)(x_i^p - \langle x_i \rangle) + k \tag{5.18}$$

where Δw_{ij}^p is the change in the weight w_{ij} with respect to the p^{th} input pattern, y_j^p is the output of the j^{th} output neuron when the p^{th} input pattern is presented and $\langle y_j \rangle$ is the expected value of the j^{th} output neuron over all patterns. By expanding Equation 5.18, compare the parameters with the standard Hebbian rule. (Objectives 1, 2).

2. (Nikovski, 1995) Compute the covariance matrix C of the following two vectors:

$$\begin{aligned} \mathbf{x}_1 &= (3,4) \\ \mathbf{x}_2 &= (12,5) \end{aligned}$$

Calculate the eigenvalues and corresponding eigenvectors of the covariance matrix. Why is only one of them not zero? What is the direction of the first principal component? How many principal components are there for only these two vectors $\mathbf{x}_1$ and $\mathbf{x}_2$. (Objective 2).

3. A conscience mechanism is a mechanism which limits the number of patterns to which a neuron can respond. (A successful neuron develops a conscience and so stops winning so much). Discuss. (Objective 4).
4. (Haykin, p435) It is said that the SOFM algorithm based on competitive learning lacks any tolerance against hardware failure, yet the algorithm is error-tolerant in that a small perturbation applied to the input vector causes the output to jump from the winning neuron to a neighbouring one. Discuss the implications of these statements. (Objectives 4, 5)
5. Discuss the effect of the Mexican hat function on the formation of the activity bubble in the SOFM. Show that the function can be calculated as the sum of two Gaussians. (Objectives 4, 5)
6. Prove that the simple competitive learning rule can be derived as an error descent rule for the function

$$E^P = \frac{1}{2}\sum_j (w_{kj} - x_j^P)^2 \tag{5.19}$$

where k is the winning neuron. (Objective 4)

Chapter 6

Genetic Algorithms

6.1 Objectives

After this Chapter, you should

1. be able to describe the GA.
2. be able to implement a simple GA.
3. understand the limits to the simple GA.
4. be able to describe PBIL, Tabu search and the Structured GA.
5. understand the difficulty found coding certain problems.
6. describe typical problem areas.

6.2 Optimisation

The methods in this chapter were developed in response to the need for general purpose methods for solving complex optimisation problems. A typical problem addressed is the Travelling Salesman Problem in which a salesman must visit each of n cities once and only once in an optimum order - that which minimises his travelling. There are two traditional methods for tackling optimisation problems:

Enumeration : basically we test the whole search space with respect to some optimisation function to find the best point with respect to this function. This can be a long process for substantial problems.

Calculus-based methods : we may divide these into

- *direct methods* which use the gradient at each point in the space to guide in which direction to search next. This is comparable to the error descent methods we used earlier and is effective in well-behaved problems (problems for which there are no local minima and which have continuous cost functions).
- *indirect methods* which attempt to solve the nonlinear set of differential equations to get those points where the gradient is zero i.e. stationary points of the evaluation function. Such solutions are often difficult or even impossible to find.

Because of these drawbacks, alternative methods which may be characterised as involving **guided random searches** have arisen. Such methods are based on enumerative techniques but allow some knowledge of the domain to permeate the search process. We will look at one of these types of methods which is based on processes which seem to involve the solution of difficult problems in the real world.

6.3 Genetic Algorithms

The attraction of simulating **evolution** as a problem solver is similar in many respects to the attraction of simulating neurons: evolution seems to offer a robust method of information processing. We will begin by briefly examining natural evolution and then consider the aspects of such evolution which are deemed to be important when we simulate it .

6.3.1 Natural evolution

Evolution is a process of change for living beings. But whereas learning is a process which happens to an individual in the space of his/her lifetime, evolution is a process which changes species over a much longer timescale. Notice that the individual is unaware of evolution: evolution is happening on a different timescale to the individual's time-scale. In addition, evolution has no memory itself: it does not matter what happened last generation or the generation before, the only material with which evolution has to play is the current version of life.

It is important to be clear about the fact that evolution acts upon **chromosomes** rather than on living beings. We can view chromosomes as a code which determines life. Therefore we require a process which will decode the chromosomes. The individual parts of a chromosome are **genes**. It is the postioning of certain genes in specific positions in the chromosome which determine the features of the resulting life. In natural evolution, there is an alphabet of only 4 values; GAs typically use a binary 1/0 alphabet.

Natural selection is a process by which those chromosomes which encode successful life-forms get more opportunity to reproduce than others. The method which is most commonly used is **survival of the fittest**. In this method, those individuals which are most fit for their current environment get most chance to reproduce and thereby transmit their chromosomes to the next generation.

In going from one generation to the next there is a process of reproduction in which there is some explicit probability of the genetic material which exists in this generation being passed on to the next generation. All high level organisms on earth use sexual reproduction which requires parents of two sexes. Some very simple organisms use asexual reproduction in which the genetic material from the single parent is passed onto its children.

6.3.2 The Simple Genetic Algorithm

Genetic algorithms were invented single-handedly by Holland in the 1970s. His algorithm is usually known as the simple GA now since many of those now using GAs have added bells and whistles. Holland's major breakthrough was to code a particular optimisation problem in a binary string - a string of 0s and 1s. We can by analogy with the natural world call the individual bit positions **alleles**. He then created a random population of these strings and evaluated each string in terms of its fitness with respect to solving the problem. Strings which had a greater fitness were given greater chance of reproducing and so there was a greater chance that their chromosomes (strings) would appear in the next generation. Eventually Holland showed that the whole population of strings converged to satisfactory solutions to the problem. Since it is so important, we reproduce the algorithm in how-to-do-it form.

1. Initialise a population of chromosomes randomly.
2. Evaluate the fitness of each chromosome (string) in the population
3. Create new chromosomes by mating. Mutate some bits randomly.
4. Delete the less fit members of the current population.
5. Evaluate the new (child) chromosomes and insert them into the population
6. Repeat steps 3-5 till convergence of the population.

Notice that the population's overall fitness increases as a result of the increase in the number of fit individuals in the population. Notice, however, that there may be just as fit (or even fitter) individuals in the population

Object Number	Object Value	Object Weight	Optimal String (W=50)
1	70	31	1
2	20	10	0
3	39	20	0
4	37	19	1
5	7	4	0
6	5	3	0
7	10	6	0
Total:	188	93	2

Table 6.1: The weights and values of 7 objects and the optimal objects held when the total weight is limited to 50

at time t-1 as there are at time t. In evolution, we only make statements about populations rather than individuals.

There are several aspects of the problem which merit closer scrutiny. We will do so in the context of a particular optimisation problem. The description given below owes much to that of [Davis,1991].

6.3.3 The Knapsack problem

The knapsack problem is one of those deceptively simple problems for which no algorithm is known for the solution of the general problem (other than exhausive enumeration of the potential solutions). Consider a knapsack which can hold at most W kg.. We wish to load the knapsack with as many objects as we can in order to maximise their value subject to this weight constraint. (We may in addition have a constraint on the total number of objects which the knapsack can hold). Then, letting v_i be the value and w_i the weight of the i^{th} object, we wish to maximise $\sum_i v_i$, subject to the constraint $\sum_i w_i < W$.

In the simple knapsack problem, we are only allowed one of each type of object and so we can model this as a binary string which has value 1 in the i^{th} bit if object i is selected for the knapsack and 0 otherwise. An example 7-object knapsack problem is shown in Table 6.1.

The knapsack problem is characterised by having optima which are often far apart in Hamming distance and so is a difficult problem for calculus-based methods. Notice how simple the coding is in this case. There is no necessity to use a binary alphabet though most GAs use only 0s and 1s.

6.3.4 Evaluation of Fitness

The evaluation of fitness in this problem is extremely simple: if the weight constraint has been violated we give the chromosome a large negative value; otherwise the fitness value of the chromosome is the value of the objects which it has loaded into the string. Practitioners of GAs generally talk about hill-climbing (as opposed to error descent). The basic idea is that the population of chromosomes here are climbing up a hill corresponding to a surface measuring their fitness. Some problems are simple particularly those consisting of a single hill; in this case, a calculus-based method would be capable of finding the summit of the hill. Of more interest, though, are problems in which there are a number of non-related peaks. Many knapsack problems are of this form. e.g. the string 1001000 shown above has fitness (value) = 107 whereas the next best string 1100011 has fitness 105. Notice that the Hamming distance between these strings is 4 or over half the maximum distance across the space. In a high-dimensional problem this can have serious consequences.

There are a number of refinements to this naive method of evaluating fitnesses which may be found useful in specific problems:

Windowing : find the minimum fitness in a population and then assign as fitness the amount by which a chromosome's fitness exceeds this value. Sometimes a minimum value is set so that every chromosome has some non-zero fitness value.

Linear normalisation : Order the chromosomes by decreasing fitness values; assign a maximum and minimum fitness for the population and then assign intermediate values between these two according to the position of the chromosome in the assigned order.

The second technique is especially useful if there is a possibility of a **super-individual** in the population. A super-individual is so called because its evaluated fitness greatly exceeds that of its contemporaries. This can be a major problem if, by chance, a super-individual appears in the population early on. It will have no real competitors and will soon dominate the population - clones of the super-individual will far outweigh any other chromosome. However, the super-individual may only be relatively super - it may represent the coding of a local maximum of the problem.

6.3.5 Reproduction

Typically we imagine reproduction to be carried out in two stages: the parent chromosomes are copied into the child chromosome bits and then some functions which alter the child chromosomes are applied to each child. The three original operations are crossover, mutation and reversal. The last of these has fallen from favour but is included here for completeness. Crossover is the major power behind genetic algorithms and usually is assigned high probability; mutation is less important and is given a lower probability. Finally the parameters are often varied during the simulation: mutation becomes more important as the simulation progresses and the population becomes less varied. Reproduction with more than two parents has not been found to be an improvement.

6.3.6 Crossover

Crossover is the driving force of a Genetic Algorithm. It mimics sexual reproduction which is used by almost all of the earth's complex species. The basic GA uses a random number generator to select a point in each parent at which the crossover of genetic material takes place. e.g. if we have two parents
PARENT A 01010101 and
PARENT B 11110000.
Then if the point between bit 2 and bit 3 is the crossover point we will get two new chromosomes
CHILD A **11**010101 and
CHILD B 01**110000**
where the bold font is used for the bits which have come from parent B. Notice that bit 4 has not changed in either child since it was the same in both parents. The crossover described here is also known as one-point crossover; it is entirely possible to allow crossover at two or more locations.

6.3.7 Mutation

Unfortunately if we only allow crossover in our population, there may be regions of the search space which are never explored; even if every part of the search space is potentially in our original population, it may be that the direction of increasing fitness initially may remove potentially valuable bits from our chromosomes. Thus we have a mutation operator which assigns at reproduction time a small though non-zero probability of change in any single bit position. This keeps diversity in our population ensuring that convergence to a sub-optimal minimum is not complete.

Note that we can define mutation as the probability of changing a bit's value or the probability of randomly selecting a bit's value; with a binary alphabet, the latter description leads to a probability of change equal to half that of the former.

6.3.8 Inversion

Inversion was an operator which was used with the original GA because of its biological plausibility. Its operation is
Original 01**10101**010 invert beteen bits 3 and 6 to get
New chromosome 01**01011**010.

There is conjecture that inversion may be found to be useful with longer chromosomes.

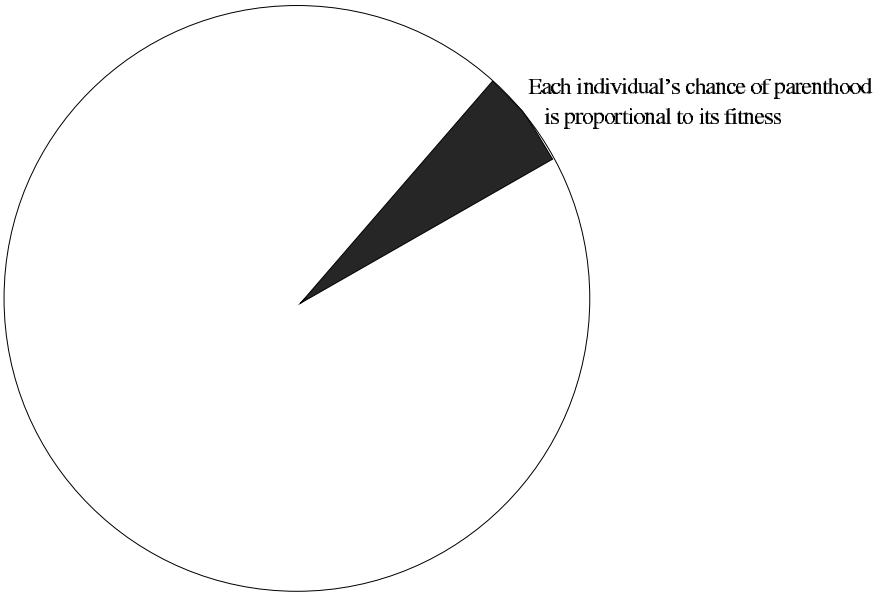

Figure 6.1: We envisage a roulette wheel in which the ball has equal chance of stopping anywhere on the circumference. Each individual is given a slice of the wheel in proportion to its fitness.

6.3.9 Selection

There are a variety of strategies for selecting which members of the population should reproduce. The original GA used **roulette-wheel selection** (see Figure 6.1).

Each individual is given a chance to reproduce which is proportional to its fitness - a fit individual should be given more chance to reproduce than an individual which is unfit. This is a mirror of real evolutionary situations in that, in order to reproduce, an individual must survive. Therefore a fitter individual which survives longer will have more chance to reproduce. We calculate the fitness of the whole population and then set the probability of an individual being selected for reproduction to be equal to the fraction of its fitness over the whole population's fitness. Therefore if we randomly generate a number between e.g. 0^0 and 360^0 and determine in which slice the random number falls, we will give each individual the appropriate chance. Now we simply select two individuals at random from the population and mate them, repeating this operation as often as is necessary till we have a replacement population.

Other possible strategies include **elitism** in which we ensure that we do not lose the best individuals from each generation by copying immediately the best of each generation into the next generation.

Another possible strategy is **steady-state reproduction** in which we only allow a few individuals at a time to reproduce. We may refine this by using **steady-state with no duplicates** in which we do not allow children which are duplicates of strings which already exist in the population. (By insisting that there are no duplicates at all in the population, we are ensuring genetic diversity). This has the great advantage of ensuring no premature convergence of the algorithm. It has however been found that steady-state reproduction methods do not work well when the evaluation function is noisy; in this case, a bad chromosome can get a spuriously good evaluation which will allow it to propagate its equally bad offspring throughout the gene-pool; if the evaluation function is deterministic, however, this problem does not arise.

6.3.10 Comparison of Operators

It is generally held that while crossover is essential to GAs, mutation alone is not sufficient to account for the algorithm's effectiveness. Note that crossover produces children which can be very different from their parents while mutation alone produces children which are only changed in one bit (or at most a few bits) from their parents. One argument for the relative importance of crossover goes like this: consider a population which uses mutation alone to change. Then if the population as a whole has a certain set chromosome values

in certain positions and if it requires two changes to its chromosomes to give an individual an edge in the reproduction game, then it may take a very long while to get an individual with both of these changes at once. If P(good mutation) =0.001 for each position, then P(both mutations at once) = 0.000001. However, if we allow crossover in our reproduction cycle we need only have P(both mutations at once) $\propto$ P(good mutation), a considerable improvement.

6.3.11 The Schema Theorem

(The fundamental theorem of GAs)

Holland has developed a theorem which describes the effectiveness of the GA known as the Schema Theorem. This roughly states that the probability of useful strings being generated and kept in the population's chromosomes depends on the length of the strings (since longer strings are more likely to be disrupted themselves) and on the distance across the string from the first to last position in the string. It must be emphasised that the Schema Theorem is not universally accepted.

We first note that we may describe a particular template or **schema** using the alphabet which we are using for the chromosome with the addition of a '*' for don't care. Thus if we are particularly interested in strings with a 1 in position 3 , a 0 in position 4 and a 1 in position 8, we describe the schema, H, of this state as **10***1** for the 10 bit string. The **schema order**, O(H), is defined as the number of fixed positions (1s and 0s in the binary alphabet); thus in the above example, the schema order is 3.

The **schema defining length** $\delta(H)$ is the distance from the first to the last fixed position in the schema. Thus in the schema above, $\delta(H)$ =5.

Holland's argument runs as follows: each individual chromosome is a representative of a great number of schema. e.g. the string 10101 represents the schema H_1 = 1**** and H_2 =*01** etc. Thus the GA may be thought of as a parallel search mechanism which is evaluating a great number of schema simulaneously. Now suppose at a particular time, t, there exists m examples of a particular schema, H, within the population.. Let the average fitness of the schema be f and the average fitness of the population be F; then at time t +1 we would expect to have m_1 examples of schema H where $m_1 = \frac{m*f}{F}$. i.e. a schema's incidence grows as its relative fitness with respect to the population as a whole grows. So schema with above average fitness will come to dominate the population.

This much seems qualitatively obvious. Now we try to quantify the effect of the two operators, crossover and mutation. Consider a string 010001010. Then it is representative of (among others) the schema *1**0**** and *1*****1*. If the probability of crossover occurring at any point is P_c then the probability that the first schema will be untouched by crossover in this generation is $(1 - P_c)^3$ while the probability that crossover will not disrupt the second schema is $(1 - P_c)^6$. Clearly longer strings are more likely to be disrupted. Therefore we can see that the defining length of a schema is essential to its chances of survival into the next generation - short schema have greater chances than long schema.

Alternatively we may say that the survival probability of a schema with respect to simple crossover is given by $P_s = 1 - P_c.\frac{\delta(h)}{(l-1)}$ where l is the length of the chromosome. Now if reproduction and crossover occur independently, then we have $m' = m.\frac{f}{F}(1 - P_c\frac{\delta(H)}{l-1})$.

Now we must consider the effects of mutation. Let the probability of mutation at any gene position be P_m. Then since there are O(H) positions of importance for each schema (those which are specified by 0s and 1s) we have the probability of a particular schema surviving mutation = $(1 - P_m)^{O(H)}$ Now since $P_m << 1$, high order powers of P_m can be ignored and so we get the survival probability as approximately $(1 - O(H).P_m$

Therfore our completed expected number of instances of the schema H in the new generation is given by

$$m' = m.\frac{f}{F}(1 - P_c\frac{\delta(H)}{l-1} - O(H).P_m) \tag{6.1}$$

This is known as the Fundamental Theorem of Genetic Algorithms or the Schema Theorem.

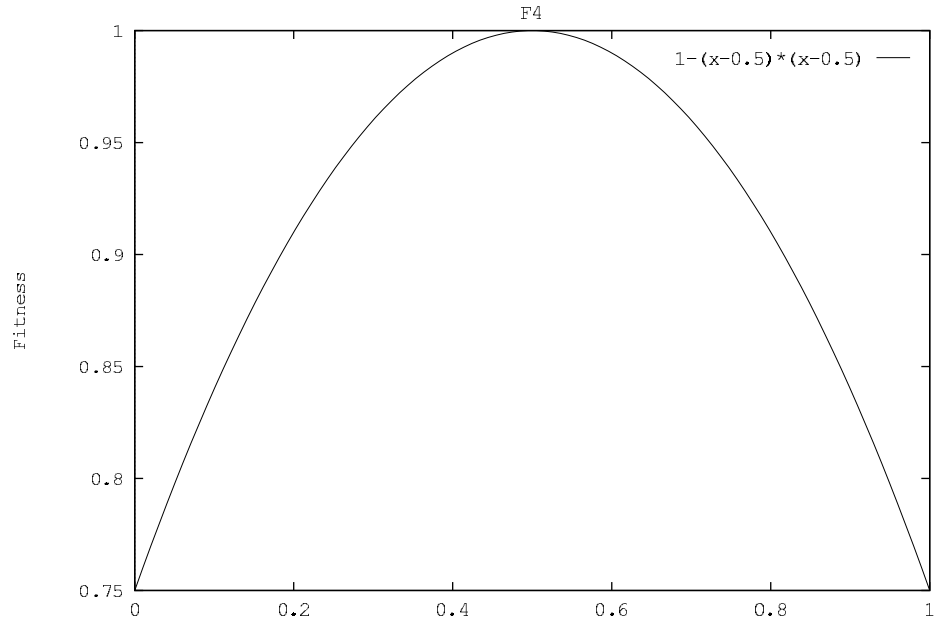

Figure 6.2: A simple hill-climbing algorithm will find this maximum

6.4 A First Example

Consider the fitness function shown in Figure 6.2. It represents a fitness function defined by fitness, $F = 1-(x-0.5)^2, 0 \leq x \leq 1$. This is a trivial problem to solve using calculus based methods since we have an explicit derivative at each point and can use

$$\Delta x = \eta \frac{dF}{dx} = \eta - 2(x - 0.5) \tag{6.2}$$

So we start off with a random value of x in the interval [0,1] and if x is less than 0.5, Δx is positive and so x will increase towards 0.5 while if x is more than 0.5, Δx is negative and so x will decrease towards 0.5.

However we will use this easy problem to illustrate the GA solution. We first have to decide the correct coding. Let us use a binary coding and wish to have the answer correct to 6 decimal places. The range of possible answers is from 0 to 1 and so we wish to slice this range up into enough bit positions to get the desired accuracy. We are looking for that value m which is such that $2^{m-1} < 10^6 < 2^m$, i.e. that value of m which is just big enough to give the required degree of accuracy. Therefore m must be 20 and so our chromosome will be 20 bits long. When we decode a chromosome we will use $x = \sum_j 2^{-j-1} bit_j$.

If our range of possible solutions had been [a,b], we would have required the value of m such that $2^{m-1} < (b-a) * 10^6 < 2^m$, and decoded chromosomes using $x = a + \sum_j 2^{-j-1} bit_j * (b-a)$.

Now we generate a number of solutions randomly.

Consider a population of 3 comprising
$\mathbf{v}_1 = 10101010101010101010$
$\mathbf{v}_2 = 11001101010100101100$
$\mathbf{v}_3 = 00101001011011100101$

Then $\mathbf{v}_1$ is equivalent to the point ≈ 0.66667 which has fitness $F = 1-(x-0.5)^2 = 1-(0.66667-0.5)^2 = 0.972222$. Similarly $\mathbf{v}_2$ is equivalent to the point ≈ 0.80204 which has fitness $F = 1-(x-0.5)^2 = 1-(0.80204-0.5)^2 = 0.908$. Similarly $\mathbf{v}_3$ is equivalent to the point ≈ 0.16183 which has fitness $F = 1-(x-0.5)^2 = 1-(0.16183-0.5)^2 = 0.8856$.

A roulette wheel selection procedure simply adds the three fitnesses together and allocates the probability of each gene being selected for parenting of the next generation as $p_I = \frac{F(I)}{\sum_j F(j)}$. So $p_1 = \frac{0.972222}{2.7658}, p_2 = \frac{0.908}{2.7658}, p_3 = \frac{0.8856}{2.7658}$. Clearly the definition of fitness has a lot to do with the probability that genes are selected in the next generation.

6.4.1 Premature Convergence

There is a trade-off between fast convergence of a GA and searching the whole search space: the danger is that fast convergence will lead to a convergence point which is only locally optimal. On the other hand, we live in a real world where we do not have infinite time or an infinitely large population. Premature convergence is often associated with a super-individual in the population- an individual whose fitness is so much greater than the fitness of those around him that he will breed so quickly that the population will become dominated by him. Sometimes GAs insist that each new generation totally replaces the old. At other times, it may make more sense to retain the best individual between generations - this is known as elitism.

6.5 Extensions to the Simple GA

Several extensions to GAs have been proposed usually in response to situations in which the simple GA does not perform particularly well. Two such situations are

1. Deceptive problems
2. Changing Environments

6.5.1 Deceptive problems

Much recent research has been in the area of deceptive problems: the simplest deceptive problem is as follows:

Consider a population in which the string "11" is most fit and the following conditions characterise the fitness landscape:

$$\begin{aligned} f(11) &> f(00) \\ f(11) &> f(01) \\ f(11) &> f(10) \\ f(*0) &> f(*1) \text{ or } f(0*) > f(1*) \end{aligned}$$

Then the order 1 schemata do not contain the optimal string 11 as an instance and lead the GA away from 11. Now in this simple example, the structure of the problem itself occasioned the deception; however, the nonlinearity in crossover and mutation, the finiteness of population sizes and poor problem representation have all been shown to contribute to deception in a problem. These are hard problems for the Simple GA to solve and so attempts have been made to modify the simple GA.

A particular type of deceptive problem is known as a *trap* function which is based on the *unitation* or number of 1's in the representation of the problem. Let the unitation of a string of length l be u. Then define the fitness of a chromosome with u unitation as

$$f(u) = \begin{cases} \frac{a}{z}(z-u), & \text{if } u \leq z \\ \frac{b}{l-z}(u-z), & \text{otherwise} \end{cases} \tag{6.3}$$

where the parameters, a,b and z determine the shape of the function. An example is shown in Figure 6.3. Clearly by altering the parameter z, the function can be made more or less deceptive as the basin of attraction of the maximum at a increases or decreases.

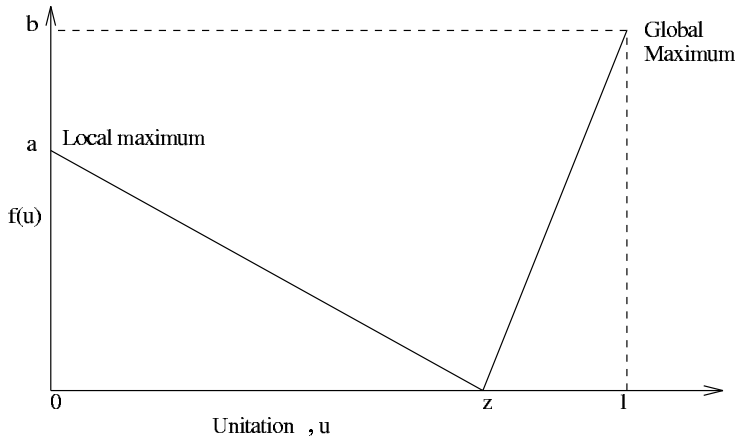

Figure 6.3: A fitness function which is deceptive. The basin of attraction of the maximum at a is much larger (dependent on z and l) than that of the maximum at b.

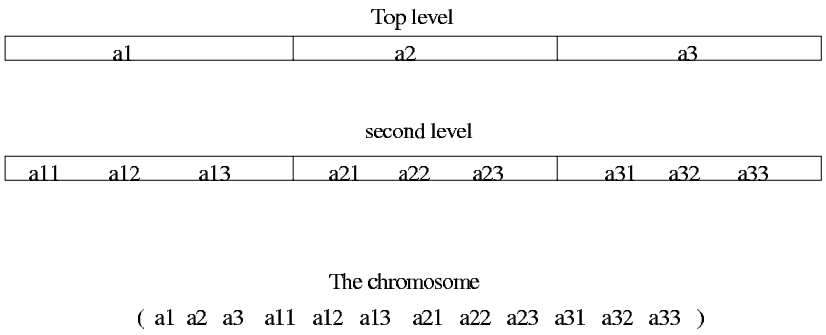

Figure 6.4:

6.5.2 The Structured GA

A number of extensions to the basic GA have been proposed; they are often based on the fact that real biological material has great redundancy within it - it has been estimated that only a small fraction of genetic material is really used in creating a new individual (a figure of 80% of genetic material being junk has been quoted). Further biologists have long realised that as cells differentiate they switch some genes on and others off making it possible for a single fertilised egg to self-organise to create a new bird, bee or baboon. Yet the differences in DNA are remarkably small e.g. a chimpanzee shares 99% of our genetic material with us or put another way a change of only 1% in our genes would make us chimpanzees. Consider how little differences then must there be between the genetic material of different humans.

One example of such an extension is the Structured Genetic Algorithm in which we imagine a two-layered genetic algorithm such as shown in Figure 6.4. Here we have a two layered genetic structure. Genes at either level can be either active or passive; high level genes either activate or deactivate the sets of lower level genes i.e. only if gene a1 is active will genes (a1 a2 a3) determine a phenotype. Therefore a single change in a gene at a high level represents multiple changes at the second level in terms of genes which are active. Genes which are not active (passive genes) do not, however, disappear since they remain in the chromosome structure and are carried invisibly to subsequent generations with the individual's string of genes. Therefore a single change at the top level of the network can create a huge difference in the realised phenotype whereas with the simple GA we would require a number of changes in the chromosome to create the same effect. As the number of changes required increases, the probability of such a set of changes becomes vanishingly small.

Such extensions have been found to be most helpful in problems in which the simple genetic algorithm does not perform well. For example the Structured Genetic Algorithm has been shown to perform well in dynamic situations i.e. when the environment is changing within the course of an experiment. For example,

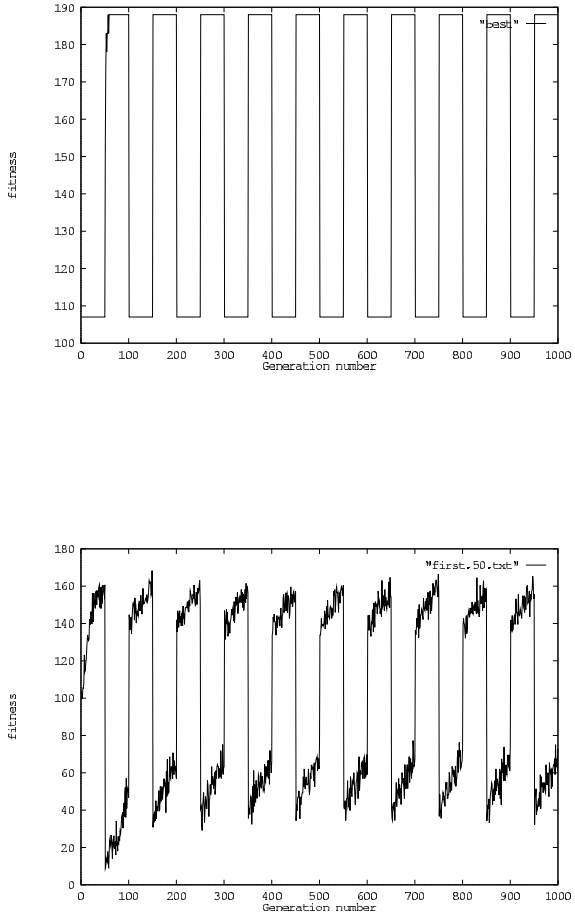

Figure 6.5: The best in each generation using the SGA and the average fitness during that generation when the constraints were changed every 50 generations.

in the knapsack problem described above, let us allow the simulation to converge (so that the population has converged on some solution of the problem (maximisation of value) under the constraint that the weight in the knapsack does not exceed 50Kg). Now change the constraint so that individuals must now optimise their value under the constraint that the weights do not exceed 75Kg. The simple GA will not perform well in this change of environment since the individuals of the population have been optimised for a different environment. The Structured GA, on the other hand, retains enough genetic diversity within its string to quickly converge to optimal values. Further if we set up an experiment so that the the environment changes systematically and regularly, the structured GA will quickly adapt to these conditions since even the phenotypes which are fittest with respect to the 50Kg constraints may retain (as passive genes) genetic material which would make their offspring fit with respect to the 75Kg constraints (and recall that these genes can be switched on with a single change at the top level). An example of the convergence of such a GA is shown in Figure 6.5 in which the constraints were changed every 50 generations.

While we have considered only 2 level chromosomes here, clearly a multi-layered approach would be possible.

6.5.3 Tabu Search

In this section and the next we consider algorithms related to GAs but different in some way from a GA. Tabu search is not a genetic algorithm yet shares some similarities with genetic algorithms particularly in the encoding of the problem. Consider again the knapsack problem. A tabu system will encode the problem in exactly the same way as the genetic algorithm. However the tabu search uses only a single chromosome randomly initialised.

First Algorithm

Since we have only a single chromosome there can be no crossover; therefore mutation plays the role of sole provider of genetic change. The algorithm can be described as:

1. Randomly select a single point on the chromosome and mutate the gene at that point. Using our binary alphabet we will be changing a 0 to a 1 or vice-versa.
2. Test the fitness of the new gene: if it is fitter than its predecessor (parent seems the wrong term) the change is accepted. If it is not fitter, the change is rejected.
3. In either case, we return to stage 2.

The search is continued till no change in the fitness of the gene is recorded after a given number of generations or until the fitness is deemed to be good enough. (We cannot get caught in a loop since we are always lowering fitness at each stage; if we opt to accept all changes that do not leave the phenotype less fit, we must test explicitly for loops.)

Notice the second crucial difference between this search method and GAs: it is a directed search in that we only accept changes to the gene which make the phenotype fitter. This is both a strength and a weakness in that, on the one hand the convergence of this algorithm is typically much faster than a GA, but on the other hand we may not find a global minimum since the whole search space may not be reachable from our initial position. Since it is very easy for a tabu search to get stuck in a local optimum, the tabu search is usually repeated from many different starting points and the best final answer taken.

A Second Tabu Search

We have not yet explained the term "tabu": in this type of search we typically retain a short term memory of previous changes and ensure that a change cannot be performed if it reverses one which has recently been done - such a change is tabu. We do not have to be black and white about tabus: we can classify the quantity of tabuness of a move numerically. Consider the problem (from Glover, 1992) of using 7 modules of insulating material to create the best insulator over all. The quality of the final insulator is dependent on the order in which the layers are laid down. An initial gene might be 2-5-7-3-4-6-1. Let us consider the operator "pairwise exchange". So to swap modules 5 and 4 would result in 2-4-7-3-5-6-1. We do not then allow the reswap of modules 5 and 4 for (say) 3 iterations. Using this method, we can allow some swaps which will decrease the overall fitness of the string *if there is no alternative*. One option is to evaluate all possible swaps and opt for the best non-tabu swap. Sometimes the tabu rule is overuled if the new solution is better than any solution which has been found previously (the aspiration criterion). Occasionally we use longer term statistics to guide the search as well as the short term memory.

6.5.4 Population-based Incremental Learning

The Tabu search described above is a hill-climbing algorithm: the chromosome at time t+1 is always better than the algorithm at time t. However the algorithm has lost the property to search the whole search space since there is no crossover operations which will allow the chomosome to jump to a different region of the space. We would like to retain the GAs stochastic elements which enable all points of the space to be reachable from each starting point but improve the convergence properties of the GA.

One possibility is the Population-based Incremental Learning(PBIL) algorithm. This abstracts out the operations of crossover and mutation and yet retains the stochastic search elements of the GA by

- Creating a vector whose length is the same as the required chromosome and whose elements are the probabilities of a 1 in the corresponding bit postion of the chromosome.
- Generate a number of samples from the vector where the probability of a 1 in each bit position of each vector is determined by the current probability vector.
- Find the fittest chromosome from this population.
- Amend the probability vector's elements so that the probability of a 1 is increased in positions in which the fittest chromosomes have a 1

The process is initialised with a probability vector each of whose elements is 0.5 and terminated whene each element of the vector approaches 1 or 0. The update of the probability vector's elements is done using a supervised learning method

$$\Delta p_i = \eta(E_{bestj}(chromosome_{ji}) - p_i) \tag{6.4}$$

where $E_{bestj}(chromosome_{ji})$ is the mean value of the i^{th} chromosome bit taken over the fittest chromosomes and p_i is the probability of a 1 in position i in the current generation.

Therefore to transmit information from generation to generation we have a single vector - the probability vector - from which we can generate instances (chromosomes) from the appropriate distribution. Notice that

$$\Delta p_i \to 0 \longleftrightarrow p_i \to E_{bestj}(chromosome_{ji}) \tag{6.5}$$

This would suggest that the number of vectors chosen to represent each generation might be a factor which will play off the speed of hill climbing with the need to search the whole search space.

6.5.5 Evolution Strategies

Evolution strategies (ES) or evolutionary algorithms were developed independently from GAs but have a similar basis in biological evolution. ES are programs designed to optimise a particular function (whereas GAs were developed as general search techniques which are used in function optimisation). Therefore ES use a floating point number representation. Consider again the optimisation of $F(x) = 1 - (x - 0.5)^2, 0 \leq x \leq 1$. Then it is more natural to represent x with a floating point number rather than a discretised version.

6.5.6 Representation and Mutation

ES represents a potential solution of a problem by a pair of real-valued vectors $(\mathbf{x}, \sigma)$. The first vector $\mathbf{x}$ represents a point in the search space. In the example, x would in fact be a scalar. The second vector σ) is a vector of standard deviations which measures the spread of a distribution. If σ is large, we have a large spread in our distribution; if σ is small, the distribution is very compact. The σ parameter is used in mutating the current vector $\mathbf{x}$ to its new value:

$$\mathbf{x}(t+1) = \mathbf{x}(t) + N(0, \sigma) \tag{6.6}$$

$N(0, \sigma)$ represents a normal (Gaussian) distribution with zero mean and standard deviations σ). If the new vector is fitter than its parent (and satisfies any constraints within the problem) it replaces its parent. If not, it is rejected. In our simple example, let us have x(t) = 0.3 and $\sigma = 0.2$. Then we generate a random number from the Gaussian distribution N(0,0.2) = 0.15 say and add this to x(t) to get x(t+1) = 0.45. Since the fitness of x(t+1) is greater than x(t), it is accepted and x(t) is removed from the population.

A refinement is to have a population of N individuals and generate M children. We can then keep the best of the (M+N) individuals in the new population - this is known as (M+N)-ES or delete the N parents and keep only the best N of the M children- this is known as (M,N)-ES.

It can be shown under certain not-too-severe conditions, that the simple ES described above will converge to the optimal function value with probability 1.

The parameter σ is usually adapted dynamically during the run of the program dependent on the fraction of mutations which prove to be successful. If you are having a lot of successful mutations, this suggests that your current solution is far from an optimum and we can increase the amount of mutation. If on the other

hand you are having very few successful mutations, you should decrease the magnitude of the mutations in order to search the local space more intensely.

When we use a multi-membered ES, we can introduce crossover e.g.

$$\begin{aligned}(\mathbf{x}_1, \sigma_1) &= (x_{11}, x_{12}, x_{13}..., x_{1n}, \sigma_{11}, \sigma_{12}, ..., \sigma_{1n}) \\ (\mathbf{x}_2, \sigma_2) &= (x_{21}, x_{22}, x_{23}..., x_{2n}, \sigma_{21}, \sigma_{22}, ..., \sigma_{2n})\end{aligned}$$

Then we can either select child vectors based on discrete crossover or can use an averaging operator e.g.

$$\begin{aligned}(\mathbf{x}_3, \sigma_3) &= (x_{21}, x_{12}, x_{23}..., x_{2n}, \sigma_{11}, \sigma_{22}, ..., \sigma_{2n}) \\ (\mathbf{x}_2, \sigma_2) &= (\frac{x_{21}+x_{11}}{2}, \frac{x_{12}+x_{22}}{2}, ..., \frac{x_{1n}+x_{2n}}{2}, \frac{\sigma_{11}+\sigma_{21}}{2}, ..., \frac{\sigma_{1n}+\sigma_{2n}}{2})\end{aligned}$$

Comparison with GAs

- GAs were developed as general purpose adaptive search techniques, ESs were developed as real valued optimisers.
- GAs use discrete representations, ESs use floating point representations
- In GAs each individual has a particular probability of being selected for parenthood whereas in ESs each individual will be a parent of a child in the next generation i.e. it is deterministic.
- Similarly in GAs a fit individual has a chance of being selected a great many times for parenthood. In ESs each individual is selected exactly once.
- In ESs we have recombination - selection - fitness evaluation. In GAs we have fitness evaluation - selection - recombination.
- In GAs the reproduction parameters remain constant whereas in ESs the parameters are varied continually.
- In GAs, constraint violations are usually dealt with by adding a penalty to the fitness function; in ESs violators are removed from the population.

Having made these points, there is a convergence between these two sets of algorithms and the underlying methods certainly come under the general heading of algorithms based on evolution.

6.6 Representation and Operators

This chapter concludes with a look at issues which we have largely glossed over: the representation of a problem. The examples we have looked at now have had representations which are largely self-evident. We now consider two examples which illustrate opposite features of GAs: the "Prisoner's Dilemma" which illustrates the power of a GA (though the representation requires some thought) and the "Travelling Salesman Problem" in which the representation is a major problem and raises issues of the best operator for a particular representation.

6.6.1 The Iterated Prisoner's Dilemma

The prisoner's dilemma is a simple game played by two players. Each is asked independently if he will help the authorities - 'grass' the other. If both refuse to comply, both will be jailed for 1 year; if one complies and the other does not, the complior gets his freedom, the non-complior gets 10 years imprisonment; if both comply, each gets 3 years imprisonment. We can put this in score-terms as shown in Table 6.2. The important feature of the scoring is that the benefits of both cooperating outweigh the benefits of only one cooperating *for the population.*

We wish to have a GA learn the optimal strategy for the population. Notice that the iteration is an important point in the game: if there is to be only one game, then deception is better than cooperation.

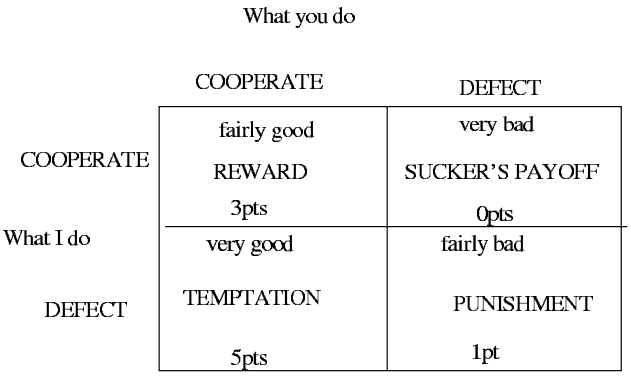

Figure 6.6: The prisoners' dilemma - the four outcomes of a single game

Player 1	Player 2	P_1	P_2
Defect	Defect	1	1
Defect	Cooperate	5	0
Cooperate	Defect	0	5
Cooperate	Cooperate	3	3

Table 6.2: The prisoners' dilemma. The important feature is that the maximal gain for the population of prisoners is if both cooperate

Representation

We consider only deterministic strategies. Assume each move made by an individual takes account of the previous 3 moves. Since there are 4 possible outcomes for each move we have 4*4*4 = 64 different histories for the last 3 moves. Therefore we require 64 bits each of which will represent a strategy for one of the histories. We also have to specify what strategy to use initially i.e. whether to Cooperate or Defect on the first move and use 6 bits to represent its premises about the three hypothetical moves which preceded the game.

Axelrood used a tournament to optimise the fitness of the population. Each player's fitness was determined by its score when it played against other members of the population. The player's fitness is its average score. In selecting players to breed, a player with an average score is allowed one mating, one with a score one standard deviation above the average is given two; one with a score one standard deviation below the average does not breed. Successful players are paired off randomly and crossover and mutation are performed.

Axelrood used a GA to breed a population of nice guys who did much better on average than the bad guys.

Results

Axelrood found that the population converged to a population which cooperated rather than defected on average. i.e. after (CC)(CC)(CC) C(ooperate). But the population was not wholly gullable since this would allow a rogue D(efector) to gain ground. Thus after a defection, the individual should also defect. Thus after (CC)(CC)(CD) D(efect). But it was also the case that the best individuals were forgiving i.e. after (CD)(DC)C(ooperate). These are usually summarised as 'do unto others'...

6.6.2 The Traveling Salesman Problem

One of the most oft-used NP-hard problems is the TSP in which a salesman is to visit a number of cities in a tour in the shortest possible time. There is no difficulty defining a fitness function for the TSP - the obvious one is the length of the tour. There is a difficulty in representation - if we represent the tour with

a chromosome of city numbers, we cannot simply allow crossover of bits of tours since this would lead to illegal tours such as tours with the same city visited more than once and one in which a particular city is never visited. We must ensure that all tours are legal.

We require therefore to limit our chromosome to being a permutation of the cities in the tour. Therefore if we number the cities (1,2,...,n) then every possible tour must be of the form $(i_1, i_2, ..., i_n)$ where each element is an unique identifier of the city.

With this representation or otherwise devise a TSP genetic algorithm solution.

Representation and Operators

This section is based on the analysis of Michalewicz. Firstly it would seem that a simple binary representation is not well suited to the TSP since there are ordering dependencies that mean that an external agency must be called so that the order of a new tour does not conflict with the requirements that each city must be visited once and only once.

Adjacency Representation

One possible tour uses a list of n cities where j is in position i if and only if the tour leads from city i to city j. Thus the tour
1-3-4-9-2-5-8-7-6
would be represented by
(3 5 4 9 8 1 6 7 2) so that position 5 contains 8 since the city 5 is followed by position 8 on the tour. Now some strings are illegal e.g. (3 * 4 1 * * * * *) is illegal since it represents the closed loop
1-3-4-1 . Therefore we must use a different crossover operator usually with the option of selecting a random city from the set of unvisited cities:

Alternating edges crossover has the new offspring taking an edge from one of its parents then another edge from the other parent etc. If a new edge introduces a cycle, the operator selects a random edge from the set of remaining edges which does not introduce cycles.

Subtour-chunks crossover is as above but each parent supplies a chunk of chromosome at a time.

Heuristic crossover selects a random starting point and then compares the two edges (from both parents) leaving this city and selects the shorter. Again if there are cycles, a random city from the unvisited ones is selected.

Ordinal Representation

Each chromosome is based on its reference to a set list of the cities such as
C=(1 2 3 4 5 6 7 8 9)
A chromosome such as (5 2 1 4 2 3 1 1 1) can be interpreted as follows:

1. The first number in the chromosome is 5 so we select the fifth element of C and delete it from C. So we have the first line in Table 6.3 in which the tour is started at 5 .

2. The second number in the chromosome is 2. So we select the second element from our reference vector and delete it from the current version of C.

3. Continue in this way till the reference vector is empty.

The main advantage of this representation is that classical crossover works.

Path Representation

So we revert to the most natural representation of all i.e. (3 4 2 7 6 5 1 9 8) represents the path 3-4-2-7-6-5-1-9-8. We consider three crossover operators:

Stage	Tour	Remaining C
1	5	(1 2 3 4 6 7 8 9)
2	5-2	(1 3 4 6 7 8 9)
3	5-2-1	(3 4 6 7 8 9)
4	5-2-1-7	(3 4 6 8 9)
5	5-2-1-7-4	(3 6 8 9)
6	5-2-1-7-4-8	(3 6 9)
7	5-2-1-7-4-8-3	(6 9)
8	5-2-1-7-4-8-3-6	(9)
9	5-2-1-7-4-8-3-6-9	()

Table 6.3: The processing of the ordinal representation TSP

Partially-mapped Crossover, PMX creates a part of a tour from the second parent while preserving the order and position of as many cities as possible from the first. e.g.

$$\begin{aligned} \text{Let } p_1 &= (123|4567|89) \\ \text{and } p_2 &= (245|1687|39) \end{aligned}$$

where the vertical lines show the crossover points. Then we begin by swapping the crossed over portions to get 2 children:

$$\begin{aligned} c_1 &= (***|1687|**) \\ c_2 &= (***|4567|**) \end{aligned}$$

Now we can fill in those other cities for which there is no conflict to get

$$\begin{aligned} c_1 &= (*23|1687|*9) \\ c_2 &= (2**|4567|39) \end{aligned}$$

Finally we note that in the crossover we replaced 4 in the first chromosome with 1 and so we perform the opposite replacement now in the first chromosme i.e. where a 1 would go (but for the conflict) we put a 4. Similarly a 6 goes where the 8 would have been and an identical transformation for the second child gives

$$\begin{aligned} c_1 &= (423|1687|69) \\ c_2 &= (218|4567|39) \end{aligned}$$

Order Crossover, OX uses sub-sequences of a tour from one parent and preseves the relative ordering from the other e.g.

$$\begin{aligned} \text{Let } p_1 &= (123|4567|89) \\ \text{and } p_2 &= (245|1687|39) \end{aligned}$$

is first crossed over as before to give

$$\begin{aligned} c_1 &= (***|1687|**) \\ c_2 &= (***|4567|**) \end{aligned}$$

Now we start from that allele in the second string after the cut (position 8) and copy into the first string the values in order (but omitting duplicates which would cause errors) to get

$$\begin{aligned} c_1 &= (345|1687|92) \\ c_2 &= (218|4567|39) \end{aligned}$$

Cycle Crossover, CX Consider again our two parents:

$$\begin{aligned} \text{Let } p_1 &= (123|4567|89) \\ \text{and } p_2 &= (245|1687|39) \end{aligned}$$

Then we begin the first child by using the value at the first allele of mum:

$$c_1 = (1********) \tag{6.7}$$

Now choose the value under the 1 value in parent 2 - it is a 2. So enter the 2 in its position in the child to get

$$c_1 = (12********) \tag{6.8}$$

Now the position under the 2 in parent 2 has value 4. So enter a 4 in its position in child 1 to get:

$$c_1 = (12*4*****) \tag{6.9}$$

Now under the 4 is a 1 which would give us a completed subtour so now we fill the remaining positions from the other parent to get

$$c_1 = (125468739) \tag{6.10}$$

We can perform an identical operation with the second child.

Matrix Representation

The most obvious way to create a matrix representation of the TSP is to put a 1 at position (i,j) - the intersection of the i^{th} row and j^{th} column - when there is a path from city i to city j and a 0 otherwise. Thus the tour (3 5 2 7 6 1 8 4 9) would be represented by the matrix

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.11}$$

Notice that, in a legitimate circular tour, each city has one road out of it and one route into it which corresponds in our matrix to having a single 1 in each row and a single 1 in each column. However this alone is not enough to specify a correct tour since e.g.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{6.12}$$

represents two disjoint subtours (1 3 8 2) and (4 5 7 9 6). Therefore while it is possible to have mutation (of a single bit or a single column) and crossover (by taking a set of columns from one parent and placing these in the other) these must be followed by some remedial action to ensure that a legitimate tour develops. The "repair algorithms" which are developed often smack of hacks. An alternative is to constrict the mutation

and crossover operators e.g. let the mutation operator randomly select a subset of the rows and columns, remove the set bits from these rows and replace them randomly in a different configuration but ensuring that they still agree with the summative properties which exist in the original matrix. Let us select rows 3,5,7 and 9 of the first matrix and columns 2,3,4, 5 and 6. This gives us a submatrix of

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{6.13}$$

which can be mutated to give

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.14}$$

i.e. preserving the subarray row and column values. Now such a new array cannot be guaranteed not to provide subtours and so some remedial action must be specified. An alternative is to allow subtours and hope that the algorithm will eventually converge to a complete (or nearly so) tour. We can similarly define crossover as an operation involving

- Stage 1: Perform the AND of two parents. i.e. the child has a 1 only in those positions where its parents agree.
- Stage 2: Alternatively copy one bit from each parent till no bits can be copied from either parent without violating the constraints.
- Stage 3: Randomly fill in the remaining necessary bit positions to make a tour (or set of subtours).

A second possible representation is to place 1's in position (i,j) in the matrix if city i is before city j in the route. E.g. if the route is (4 6 3 7 1 2 8 9 5), the matrix would be

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.15}$$

Now the matrix satisfies

1. The number of 1's is exactly $\frac{n(n-1)}{2}$.
2. The main diagonal is composed of zeros
3. If $w_{ij} = 1$ and $w_{jk} = 1$, then $w_{ik} = 1$.

Now the intersection (AND)of two parent matrices is guaranteed to satisfy these constraints and so comprise a legal tour. So we find the intersection of two parents (which is where they agree) and then add in other 1's by alternatively choosing rows from each parent. E,g. if the tour above is added to the tour (1 2 3 4 5 6

7 8 9) represented by the matrix

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.16}$$

we get the tour (where the parents agree)

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.17}$$

This provides a partial ordering on the set of cities: we know that

- city 1 is before 2 which is before 5,8 and 9
- city 3 is also before 5,8 and 9
- city 4 is before cities 5-9 etc

A second possible operator is the union operator provided it is restricted to disjoint subsets of the matrix. This is useful if we have subtours which do not intersect with one another.

An ongoing area of research is the application of such matrix evolution techniques to timetables in which the obvious representation is a matrix.

6.7 Applications

GAs can be used in most situations in which we wish to optimise the value of a function (under a set of constraints). We must note, however, that if there exists a calculus based method (or any closed form, calculable method) then such a method is liable to outperform the GA and should be used. However to prove the ability of GAs they have been used in a great number of different situations such as:

1. As a classifier system: an image is digitised onto a 25*25 grid giving 625 pixels where each pixel can take only a binary 0/1 value (and is thus capable of only differentiating between light and dark). A set of feature detectors is chosen such that each detector is a known subset of the set of pixels. During training, images of known classification are presented to the system and lists of detector states are stored. During recognition, an unknown image is presented and its score is calculated and a list of the ranked classifications is produced

2. Optimisation of process control parameters. e.g. optimisation of the parameters in a pipeline operation in order to maximise the flow of oil (or other fluid).

3. Scheduling problems in which we have various constraints such as the number of machines which are available, the time by which various jobs must be completed, the order in which sub-tasks of the jobs must be done etc. GAs have been used with success for such problems since there is no good mathematical method for their solution.

4. Game strategy evolution and strategy acquisition in general
5. Evolution of optimal robot trajectories with respect to a given task.
6. Evolution of robot behaviours
7. Evolution of most fit neural networks for a given task.
8. Fault diagnosis.
9. And of course, the travelling salesman problem.

6.8 Exercises

1. Discuss with examples deceptive problems in GAs. Is a deceptive problem alleviated by using a Structured GA? Explain. (Objectives 3, 4).
2. (Goldberg) Six strings have fitness function values: 5,10,15,25,50,100. Under roulette wheel selection, calculate the expected number of copies of each string in the mating pool if a constant population size, n=100, is maintained. (Objectives 1, 2).
3. Using this chapter's data on the knapsack problem, hand simulate with your neighbour the convergence (or otherwise) of a Tabu Search on random initial strings. Ask your neighbour to call out random numbers to initialise the string and to decide which gene to change and you calculate the fitness each time. Repeat several times. (Objective 4).
4. If the probability of mutation at any gene is 0.1, what is the probability that a 10 bit string will survive a single generation without change. Repeat with a 20 bit string. Repeat with a mutation rate of 0.01. (Objective 1).
5. What additions could be made to the PBIL algorithm to optimise the functions in Figures 6.7- 6.8. (Objective 4.)
6. Describe the difficulties encountered when coding the TSP. Repeat with the problem of coding a GA to evolve a timetable. Discuss other problems which are difficult to code. (Objective 5).

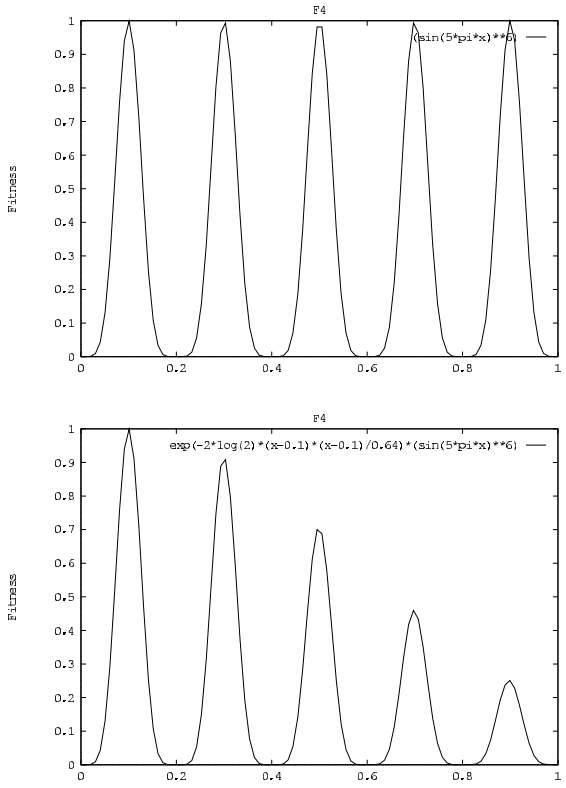

Figure 6.7: Deb's first two functions , F1() and F2().

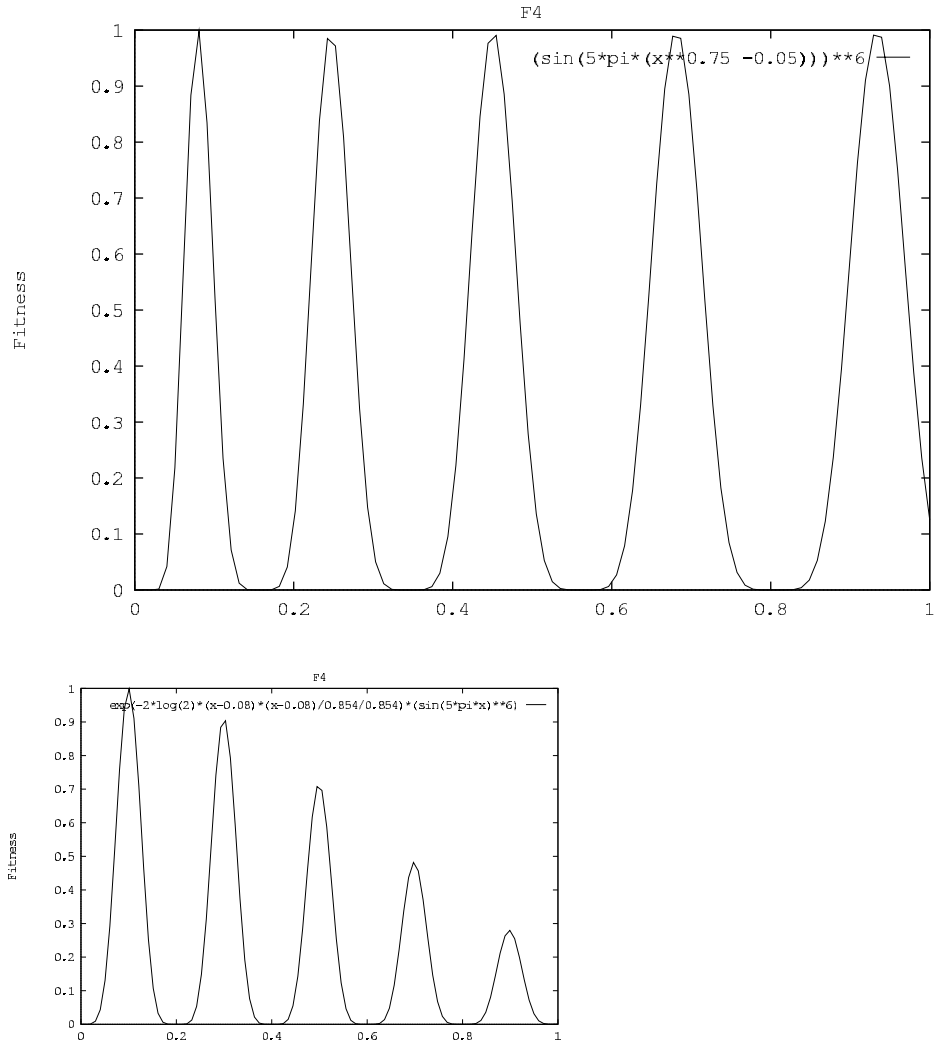

Figure 6.8: Debs functions, F3() and F4().

Chapter 7

Recurrent Networks

7.1 Objectives

In this chapter, you will

1. understand the Hopfield network
2. understand the Bolzmann Machine
3. understand simulated annealing
4. understand Cellular Automata
5. be able to create a Hopfield network
6. understand the place of dynamic systems in modern AI

7.2 The Hopfield Network

The Hopfield network is so-called after John Hopfield who from 1982-85 introduced the network and gave it a formal basis on the foundations of statistical physics. We shall try to discuss the main aspects of the network without delving too deeply into the physics.

The Hopfield network is designed to meet the need for biologically realistic models of neural networks which takes the dynamics of activation-passing into account. For example, in the MLP network we implicitly assume that all activation is passed forward at the same time. It seems biologically implausible that nature would sanction such dependencies. Also, in the models of the last few chapters, we have considered in the main feedforward networks which are strongly directional. In biological neural networks, on the other hand, there are many connections between neurons which can be described as feedback connections. (A more accurate description of the connections in the networks discussed in this Chapter is that these are lateral connections - connections between neurons which are in a single layer).

The Hopfield model is a type of associative memory which acts as a **content addressable memory** in that we can create the memory and use a partial or noisy stimulus to recall the stored memory which is stored in the weights. The model therefore acts as a error-correcting mechanism for incorrect stimuli. The core of the method is to create a fixed point of a dynamical system corresponding to a memory of the network. A dynamical system is one in which

$$\mathbf{x}(t+1) = F(\mathbf{x}(t)) \tag{7.1}$$

where F() is some (non-linear) function of the state-vector $\mathbf{x}(t)$ at time t. Thus e.g. if $F(x) = \frac{x+1}{2}$ and the initial state of the system is x=3, then we have

$$x(1) = F(x(0)) \quad = \quad \frac{x(0)+1}{2} = 2$$

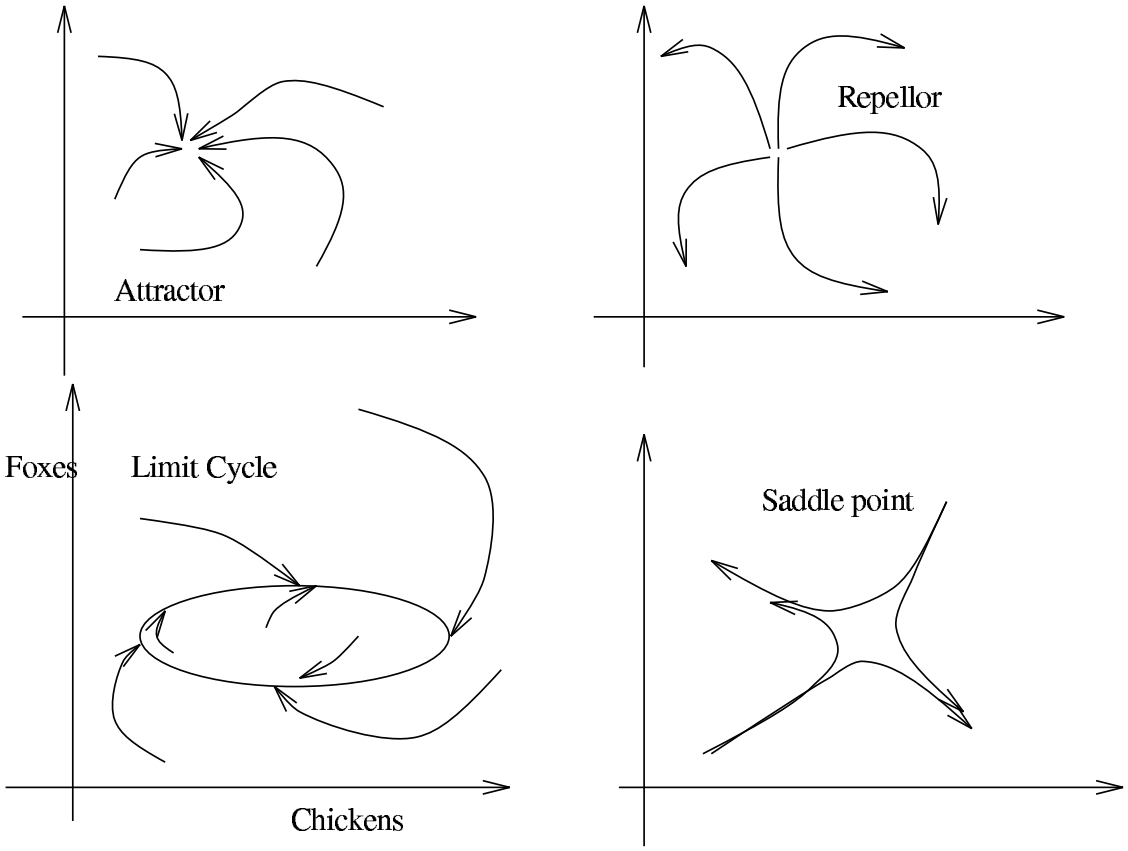

Figure 7.1: Some possible events in a dynamical system

$$x(2) = F(x(1)) = \frac{x(1)+1}{2} = 1.5$$
$$x(3) = F(x(2)) = \frac{x(2)+1}{2} = 1.25$$

A fixed point of this system is x = 1 since

$$x = F(x) = F(1) = \frac{1+1}{2} = 1 \tag{7.2}$$

Note that this does not mean that the system will necessarily converge to 1 (though the above system does) but does mean that if the system gets to the value 1, it will stay there.

The stable points of the Hopfield network are the prototypes of particular memories which have been placed within the network. Then when a partial or noisy pattern is presented to the network, it should iteratively converge to a fixed point - the prototypical memory.

Examples of possible events when a dynamical system is set in motion are shown in Figure 7.1. These types of diagrams are not specific to Hopfield networks but can be thought of as representing any dynamical system.

There is also the possibility of a strange attractor (the butterfly wings seen on the covers of most popular science books on chaos).

7.2.1 Activation in the Hopfield Network

The Hopfield network can be considered a single layer of binary neurons which are either on (firing +1) or off (firing -1). Therefore a single pattern in the network, the network's state, is an N dimensional vector of

+1s and -1s. Then if the firing of neuron j is denoted S_j, we have

$$Act_j = \sum_{i=1, i \neq j}^{N} w_{ji} S_i - \theta_j \tag{7.3}$$

Then we perform simple binary thresholding so that

$$S_j = sgn(Act_j) = \begin{cases} +1 & \text{if } Act_j > 0 \\ -1 & \text{if } Act_j < 0 \end{cases} \tag{7.4}$$

If Act_j is zero, we typically leave the sign of S_j in the same state as it was before.

7.2.2 Storage Phase

Consider a set of p patterns, $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_p$ which we wish to memorise. Each pattern is N dimensional. Then we set the weight between neurons i and j such that it is proportional to the expected joint firing of the neurons i and j over the p patterns:

$$w_{ji} = \frac{1}{N} \sum_{\mu=1}^{p} x_{\mu,j} x_{\mu,i} \tag{7.5}$$

where $x_{\mu,j}$ is the firing of the j^{th} neuron for the μ^{th} pattern. While this is **one-shot learning** (as opposed to the iterative methods which we have used till now), it is usually described as Hebbian learning.

We ensure that $w_{ii} = 0, \forall i$ which is important since self-feedback can adversely affect stability and so we can create a single weight matrix W using

$$W = \frac{1}{N} \sum_{\mu=1}^{p} x_{\mu,j} x_{\mu,i} - \frac{p}{N} I \tag{7.6}$$

where I is the identity matrix. A further aspect of the network which is important for stability is that the network is symmetric ($w_{ij} = w_{ji}, \forall i, j$).

7.2.3 Retrieval Phase

Assume we have a noisy or incomplete input pattern and we wish to retrieve the original memory. We set the pattern on the network by making the neuron's firing +1 or -1 as appropriate. We may choose to update synchronously or asynchronously.

- Synchronous updating: we calculate the new activation of each neuron in turn and only then update the firing of each neuron. There is a possiblity that this system will lead to a cycle of states.
- Asynchronous updating: we select a neuron at random and update its activation and firing and repeat till the network state never changes. It can be shown that such a system will always converge to a fixed state. The proof uses the method of Lyapunov (see Chapter 2 and Section 7.2.5)

7.2.4 An Example

We will consider an example from Haykin (p294). Consider the network shown in Figure 7.2. Let the weight matrix be

$$W = \frac{1}{3} \begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix} \tag{7.7}$$

Then there are 8 possible states for the network to be in at any time. We can show that e.g. $\mathbf{y}$ =(-1,1,-1) is a stable state since

$$W\mathbf{y} = \frac{1}{3} \begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} -4 \\ 4 \\ -4 \end{bmatrix} \tag{7.8}$$

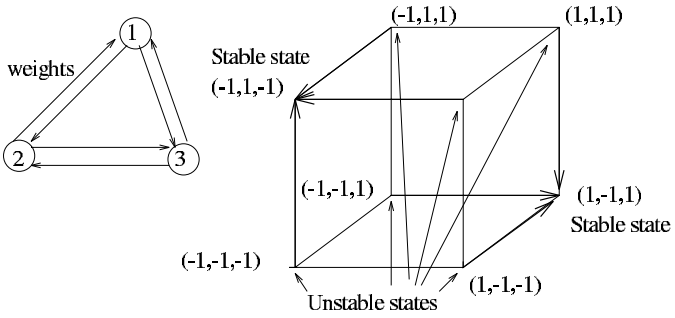

Figure 7.2: The left diagram shows a layer of three neurons, each of which is connected to the other two via (symmetric) weights. The right diagram shows the stable and unstable states and the transitions from unstable states to stable states.

to give the vector of activation values. Now thresholding these gives the original vector

$$\mathbf{y} = sgn(W\mathbf{y}) = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \tag{7.9}$$

Also the state (-1,-1,-1) converges to the same state since

$$W\mathbf{y} = \frac{1}{3}\begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} 0 \\ 4 \\ 0 \end{bmatrix} \tag{7.10}$$

Now using the rule that if the activation is 0 the firing state of the neuron is unchanged we can see that the state (-1,-1,-1) converges to (-1,1,-1) in one iteration (where we know it will be stable).

We can show (see Question 1) that the network has 2 fixed states corresponding to the 2 stored memories (-1,1,-1) and (1,-1,1). Hence the network weight matrix, the memory matrix is calculated by

$$\begin{aligned} W &= \frac{1}{3}\begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix}\begin{bmatrix} +1 & -1 & +1 \end{bmatrix} + \frac{1}{3}\begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix}\begin{bmatrix} -1 & +1 & -1 \end{bmatrix} \\ &\quad -\frac{2}{3}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{1}{3}\begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix} \end{aligned}$$

which is how we calculated the memory in the first place.

A more realistic computer experiment is given in Haykin p296-302 where a Hopfield network converging to various fixed points is shown. However it is shown that we cannot guarantee *a priori* that the net will converge to the state which we wish. It may be that there will be another state to which it will converge. We should also be aware of the fact that, in creating a Hopfield network's memory we also create

- **reversed patterns**: if we store a pattern we also create a memory of the reversed pattern - the negative of the stored pattern. Therefore if we are storing the pattern (-1,1,-1,1,1,1,-1) we are also storing the pattern (1,-1,1, -1,-1,-1,1). It can be shown that a pattern and its reverse have the same energy (see below).

- **spurious states** which are mixtures of an odd number of memories. Therefore if we are storing the patterns (-1,1,-1,1,-1,1,-1), (1,1,1,-1,-1,-1,1) and (-1,-1,1,-1,1,-1,1) we will also be storing the pattern

(-1,1,1,-1,-1,-1,1) which is the pattern formed by using a majority rule on the 3 patterns. Fortunately it can be shown that such spurious states have a higher energy (see below) than the states of the stored patterns and so the network is less liable to converge to them.

7.2.5 Energy Minimisation

We again use the method of Lyapunov.

Let the state of the i^{th} neuron in a Hopfield network be given by s_i. Then we can define the energy of the current state of the Hopfield network as

$$E = -\frac{1}{2} \sum_{i=1, i \neq j}^{N} \sum_{j=1}^{N} w_{ji} s_i s_j \tag{7.11}$$

We may drop the restriction $i \neq j$ since $w_{ii} = 0, \forall i$.

Note that the energy function is bounded below by the same value as we found with the BAM.

Then if we change the network state by flipping the state of the j^{th} neuron, there will be a change in the energy of the network given by

$$\Delta E = -\Delta s_j \sum_{i=1}^{N} w_{ji} s_i \tag{7.12}$$

Now notice that the change Δs_j is positive if $\sum_{i=1}^{N} w_{ji} s_i$, which is the activation of the j^{th} neuron, is positive and negative if $\sum_{i=1}^{N} w_{ji} s_i$ is negative. In other words, ΔE is monotonically decreasing.

Therefore if we present any pattern to the network, activation will be passed through the network (the network will settle) till a stable point is reached with (locally) minimum energy. The local minima of the energy landscape are known as the *attractors* of the system - when the network reaches one of these states, it will stay there. It can be shown that the fundamental patterns learned by the Hopfield network are attractors and indeed attractors with large basins of attraction and low energy states.

7.3 Simulated Annealing

Annealing is a method for hardening metals: the metals start off at a very high temperature (i.e. they are molten) and the temperature is allowed to drop slowly. This method has been shown to produce metals with consistent properties throughout the sample. Simulated annealing attempts to mimic this property for optimisation problems by creating a variable which by analogy with real annealing is known as the temperature. During the lifetime of the simulation, this parameter is decreased and as it is decreased the properties of the simulation converge to optimise some desired criterion.

With deterministic algorithms using gradient descent (e.g. backprop or LMS) if the algorithm causes the simulation to reach a local minimum, there is no way out - the algorithm must cause change to occur in a direction which causes the (error) function to decrease and if there is no direction in which this is possible, the local minimum is a fixed point. Simulated annealing gets over this problem by stipulating merely that, on average, descent takes place on the (error) function surface[1]. Therefore it is possible to climb out of local minima since the algorithm permits uphill climbs on occasion. The algorithm has a temperature parameter (NOT the temperature of a real simulation/neural network, merely an abstract parameter). When the temperature is high, as it is initially, it is easy to climb out of local minima (there is a high probability of going uphill); on the other hand, when the temperature is low, it is very difficult to climb out of local minima. During the simulation, the temperature is started at a high level and slowly decreased. Thus, initially all states of the network are very possible (including local minima with relatively high energy) but as the network cools, it becomes harder to jump to states with high energy and the low energy states prevail. It often appears that the simulation is responding to the coarse features of the data at high temperatures and as the temperature drops becomes more involved in differentiating between the finer features of the input data.

[1]You may find it useful to equate states with high (error) function values with states with high energy.

7.3.1 The Metropolis Algorithm

The Metropolis algorithm provides a simple method of simulating annealing. As noted above, simulated annealing is most useful in constraint satisfaction problems. We have previously met the Travelling Salesman Problem in which the constraints are that each city must be visited once and once only. To solve the TSP using simulated annealing we will equate the energy E of the current state of the simulation with the cost (in time or money) to make the journey which the current route would require.

We randomly select a section of the route to change e.g. if currently the route is $A \to B \to C \to D$, we may change this to $A \to C \to B \to D$. In doing so, we will change the energy (cost) of the simulation. If this change is negative ($\Delta E \leq 0$) the change is accepted. (Therefore we are using gradient descent). If the change in energy is positive, it is accepted with probability

$$P(\Delta E) = \exp\left(-\frac{\Delta E}{T}\right) \tag{7.13}$$

where T is the temperature. Note that it is this feature which allows us to climb out of local minima. Thus changes which result in very small increases in energy have a greater chance of being accepted than changes which result in a large change in energy. Also the probabilities of acceptance depend on the temperature since, if the temperature is high, all changes have a high probability of being accepted while as the temperature drops so the probability of accepting these uphill moves decreases.

Now the difficulty with the algorithm is that the temperature must be decreased very slowly and at each temperature the system must be allowed to reach thermal equilibrium. This can lead to very long simulations.

The change schedule described above has the interesting property that at equilibrium the probability of a system being in any state α with associated energy E_α when the temperature of the system is T is given by the Boltzmann distribution

$$P(\alpha) = \frac{\exp\left(-\frac{E_\alpha}{T}\right)}{\sum_\beta \exp\left(-\frac{E_\beta}{T}\right)} \tag{7.14}$$

where the summation is done over all states β which are possible states for the system. At high temperature, each state has nearly uniform probability whereas at low temperature, states with lower energy become much more probable than the high energy states.

This distribution occurs again with an artificial neural network of stochastic neurons; the general ANN of this type is known as the Boltzmann Machine.

7.4 The Boltzmann Machine

Consider a function which is bounded from below i.e. we know that a minimum value exists. Then, as in the last section, we wish to investigate the landscape of the function in a guided manner using error descent but not exclusively since otherwise we may land prematurely in a local minimum. Therefore we introduce a stochastic element into the change of weights.

A simple Boltzmann Machine is shown in Figure 7.3. The Boltzmann Machine is similar to the Hopfield network in that

- Each neuron fires with binary values (usually ± 1)
- All connections are symmetric
- In activation passing, the next neuron whose state we wish to update is selected randomly.
- There is no self-feedback (connections from a neuron to itself).

However, there are important differences. In particular,

- A Boltzmann Machine uses hidden neurons. This allows a much richer representation of the input data than is possible in the Hopfield network.

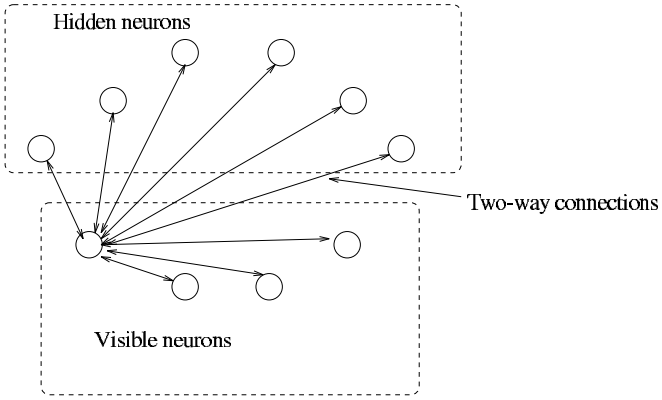

Figure 7.3: A Boltzmann Machine with hidden and visible neurons. The network is fully connected with symmetric connections (though only connections to a single neuron are shown).

- The Boltzmann Machine uses stochastic neurons: at any time there is a probability attached to whether the neuron fires whereas the Hopfield network is based on deterministic principles.
- The Boltzmann Machine may use either supervised or unsupervised learning.

We will concentrate on the unsupervised learning method. There are three phases in the operation of the network:

1. the clamped phase in which a pattern is clamped onto the visible neurons and the activation of the hidden neurons is allowed to settle.
2. the free running phase in which the clamping is removed and all activation is allowed to settle.
3. the learning phase

These phases iterate till learning has created a Boltzmann Machine which can be said to have learned the input patterns and will converged to the learned patterns when a noisy or incomplete pattern is presented.

7.4.1 The Clamped Phase

The network weights are generally initially randomly set to values uniformly within a small range e.g. -0.5 to +0.5. Then an input pattern is presented to the network and *clamped* to the visible neurons. Now perform a simulated annealing on the network by chosing a (hidden) neuron at random and flipping its state from s_j to $-s_j$ with probability

$$P(s_j \rightarrow -s_j) = \frac{1}{1+\exp(-\frac{\Delta E}{T})} \tag{7.15}$$

where the energy of the network is given by

$$E = -\frac{1}{2}\sum_{i \neq j}\sum_{j=1}^{N} w_{ji} s_j s_i \tag{7.16}$$

As with the Hopfield network, the activation passing can be shown to continue till the network reaches a low energy state. However the stochastic nature of the Boltzmann Machine means that we cannot specify a single state which will be the attractor of the system. Instead we may state that the network will reach a state of *thermal equilibrium* in which individual neurons will change state but the probability of any single state can be calculated. In fact we may show that at equilibrium, the states of the Boltzmann Machine obey

the Boltzmann distribution. i.e. the probability of a system being in any state α with associated energy E_α when the temperature of the system is T is given by the distribution

$$P(\alpha) = \frac{\exp(-\frac{E_\alpha}{T})}{\sum_\beta \exp(-\frac{E_\beta}{T})} \tag{7.17}$$

Notice that the updating may be thought of as a local operation since

$$\Delta E = -\Delta s_j \sum_{i=1}^{N} w_{ji} s_i = -2 * |v_j| \tag{7.18}$$

where $|v_j|$ is the absolute value of the j^{th} neuron's activation.

Now the temperature is gradually dropped and at each temperature the network is allowed to relax to as low an energy state as it may at that temperature. At the final temperature, the correlations between the firing of pairs of neurons are noted.

$$\rho_{ij}^{+} = \langle s_j s_i \rangle^{+} \tag{7.19}$$

where the + indicates that the correlation is being calculated when the visible neurons are in a clamped state.

7.4.2 Free-running phase

Now we repeat the calculations of the previous section but this time we do not clamp the visible neurons. After presentation of the input patterns all neurons are allowed to update their states and the annealing schedule is performed as before. Again at the final temperature we calculate the correlations between the firing of pairs of neurons:

$$\rho_{ij}^{-} = \langle s_j s_i \rangle^{-} \tag{7.20}$$

where the - indicates that the correlation is being carried out when the visible neurons are *not* in a clamped state.

7.4.3 Learning phase

Now we use the Boltzmann Machine's learning rule to update the weights:

$$\Delta w_{ij} = \eta(\rho_{ij}^{+} - \rho_{ij}^{-}), \forall i, j \tag{7.21}$$

The first term is very like a Hebbian learning term while the second is a decay or forgetting term. The derivation of the rule is beyond the scope of this course but is given in (Haykin, pp 322-327).

It can be shown that the rule is equivalent to modifying the weights till the distance between the statistics of the distribution which the network receives in the clamped phase is matched (as closely as possible) by the knowledge that the Boltzmann Machine has of this distribution and so the pattern completion property of the Boltzmann Machine may be established.

7.4.4 Supervised learning

A diagrammatic representation of the Boltzmann Machine when used in supervised learning mode is shown in Figure 7.4. Now we see that the set of visible neurons is split into input and output neurons and the machine will be used to associate an input pattern with an output pattern.

During the clamped phase, the input and output patterns are clamped to the appropriate units. As before the hidden neurons' activations are allowed to settle at the various temperatures. However during the free-running phase, only the input neurons are clamped - both the output neurons and the hidden neurons are allowed to pass activation round till the activation in the network settles.

There is one other difference which is that the learning rule requires to be modulated by the probability of the patterns seen at the input. Thus

$$\Delta w_{ij,\alpha} = \eta P_\alpha(\rho_{ij}^{+} - \rho_{ij}^{-}), \forall i, j \tag{7.22}$$

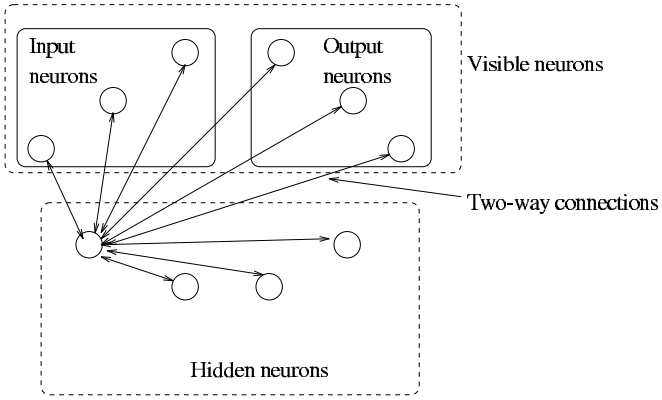

Figure 7.4: A supervised Boltzmann Machine. The visible units are split into input and output neurons. During the free-running phase, only the input neurons are clamped; during the clamped phase, both input and output neurons are clamped.

where P_α is the *a priori* probability of state α at the input neurons. We must sum the learing thereafter over all states α.

7.5 Applications

We have noted that the Hopfield network can be used as a content addressable memory. In fact, the BAM of Chapter 2 can be thought of as a special case of the Hopfield network.

The Boltzmann Machine has been described as capable of solving optimisation problems and yet even the simple Hopfield network has been used for this (though often with a stochastic element to its settling). Examples of such problems are given in [Chapter 4, Hertz, Krogh and Palmer] e.g.

1. The weighted matching problem. Here we have a set of N points with a known "distance" between each. Let d_{ij} be the distance between the i^{th} and j^{th} points. Then the problem is to link the points together in pairs so as to minimise the total length of the links. An example of this problem may be the optimal scheduling for two machines.

2. The Travelling Salesman Problem yet again.

3. Graph bipartitioning in which we have a set of points which we wish to split into two disjoint sets with as low an associated cost as possible.

4. Image reconstruction problems in which we wish to reconstruct the image as accurately as possible. Such reconstruction will often use the fact that images are locally homogeneous i.e. remain constant over a small area. Therefore we attach a cost to discontinuities and attempt to relax the network till it minimises the cost.

7.6 Cellular Automata

Perhaps the simplest dynamical system is that produced by cellular automata.

This section will investigate their properties; it is directly taken from the Santa Fe Institute notes.

Cellular Automata are discrete space/time logical universes, obeying their own local physics. Space in CA is partitioned into discrete volume elements called "cells" and time progresses in discrete steps. Each cell of space is in one of a finite number of states at any one time. The physics of this logical universe is a deterministic, local physics. "Deterministic" means that once a local physics and an initial state of a CA has been chosen, its future evolution is uniquely determined. "Local" means that the state of a cell at time

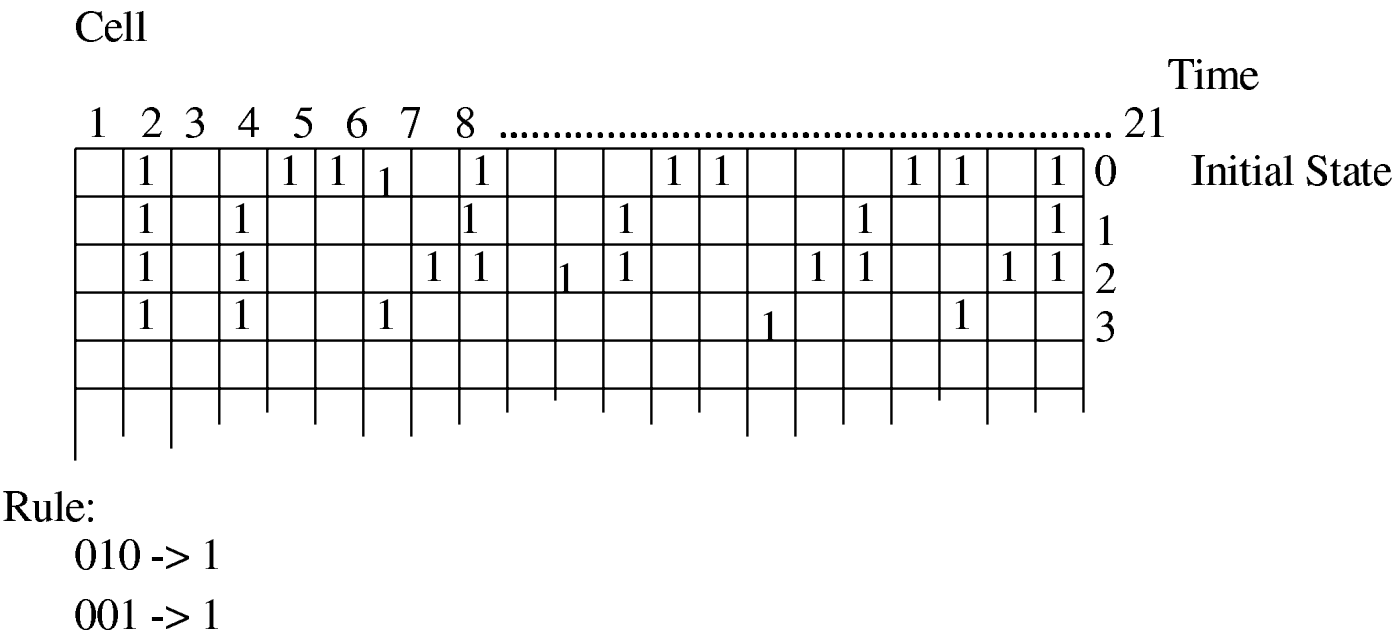

Figure 7.5: A simple one dimensional CA with a neighbourhood of only one cell. The simple rule only maps two patterns to 1 and all others to 0

000	0	0
001	1	1
010	2	1
011	3	0
100	4	1
101	5	0
110	6	0
111	7	0

Table 7.1: A nearest neighbour rule: e.g. if self and the squares on either side are 0 (top line) stay at 0; if self and square to the left are zero but square to the right is 1, change to 1 (second line). etc.

t+1 is a function only of its own state and the states of its immediate neighbors at time t. Notice that time is also discrete in this universe.

The local physics is determined (typically) by an explicit mapping from all possible local states of a predefined neighborhood template (e.g., the cells bordering on a cell), to the state of one of those cells at the next time-step. For example, for a 2-state, 1D CA, with a neighborhood template that includes a cell and its immediate neighbors to the left and right, there will be 8 possible neighborhood states (000....111) and for each we must explicitly state whether the cell itself will map to a 1 or a 0, implying that there will be $2^{2^3} = 256$ possible local physics under these constraints. For an 8-state, nearest neighbor 2D CA, there will be 8^5 possible neighborhood states, and a choice of 8 states to map to for each of those, yielding a total of 8^{8^5} or 8^{32K} possible physics.

A Transition Rule defined for a nearest-neighbor two-state CA is shown in Table 7.1. The last column becomes the code for the rule which can then be numbered from 0 to 255. For instance, the rule shown is Rule 22 (== binary 00010110). Rule 22 has been shown to exhibit complex behaviour. Others are very much simpler. Wolfram addressed the issue by dividing cellular automata into four broad, largely qualitative classes:

Type I all configurations map to a homogeneous state

Type II all configurations map eventually to simple separated periodic structures

Type III all configurations may to chaotic patterns, making large-time prediction of behavior impossible, though precise statistical statements may be made

Type IV configurations may produce propagating structures, sometimes soliton-like, may perform useful computations, have behavior which is provably undecidable etc.

This classification is a summary of observations based on the simulation of many automata, it is purely phenomenological, and offers no means to construct rules exhibiting a given behavior.

Cellular automata can process information when their collective global behavior exceeds the sum of the behaviors of their individual cells. Information processing emerges from cooperation of the individuals, who must be able to communicate in a meaningful manner. Class I, II, and III automata fail to provide a substrate for meaningful communication. In classes I and II, information is confined locally, and is simply not transmitted. In class III, by contrast, information is meaningfully transmitted only over short distances in space and time. Deterministic chaos injects noise at a significant rate, washing out all signal. Class IV automata form the intermediate case, not too hot, not too cold.

One of the most well-known methods of investigating CAs is that of Langton. Langton employs a parameter, λ, which is a measure of the distribution of state transitions in the rule table. For CA with two states is particularly simple: it is the fraction of rule table containing neighborhoods which lead to a 1 under the rule. He found that for low values of λ, simple rules, of Wolfram type I and II, were mostly generated. For high values, the rules are typically chaotic, of Wolfram type III. At the cross-over between simple and chaotic, complex rules become prevalent. For rules with a small number of states there is a great deal of variability in the behavior or rules with a specified parameter value. It is only in the limit of an infinite number of states that the transition becomes sharp.

A good deal of work in CA has been devoted to understanding what classes of behavior and/or structure might be found in such astronomically large spaces of possible local physics.

7.6.1 Description of the Parameters

These parameters are those pertaining to the Santa Fe Institute's simulator and are repeated to give you an idea of what may be changed when you run a public domain simulator.

World Size: Number of cells in the 1-D world. It must be a positive integer and is limited to no greater than 1024.

Number of Iterations: Number of times the world is updated. It must be a positive integer no greater than 2048, number of iterations = [1,2048].

Number of States: Number of possible states a cell can be in, i.e. for a 2 state CA, each cell is either in state 0 or 1. We have limited the number of states to a maximum of 16. We have also limited both number of states and radius of neighborhood by the following constraint:

$$(numberofstates)^{(2*radius+1)} < 2^{16} \tag{7.23}$$

where $(numberofstates)^{(2*radius+1)}$ is the size of the rule table.

Radius of Neighborhood: The neighborhood radius is the number of cells to each side of a cell that affect the future state of that cell. For example, with a radius of 2, the next state of any cell is determined my the current state of the cell, two cells on its left and two cells on its right.

Rule Number: Allows for input of a specific rule wished to be explored. Base is specifiable. The most well known examples are the base 10 rules of Wolfram. Given larger number of states or neighborhood radius than the elementary 1-d CA, rule specification grows decreasingly useful amd informative. It is suggested lambda is used to explore CAs with large rule tables instead.

Lambda: Lambda is the percentage of local transitions leading to non-quiescient states. The lambda options allow for varyiing degrees of quiescience to be selected as well as the specification of symmetrical or non-symmetrical rule selection.

Initial Conditions: Defines the percentage of non-quiescient states in the world for the initial configuration. There is also an option to begin with only one non-quiescient state.

Display: Each cell is represented as a square on the screen. This option allows for the specification of the cell scale factor of the square. The parameter represents the pixel size of one side of the square cell. Thus, larger positive integers will make each cell larger. There is a bound on the maximum pixel size given by: pixel size ¡ (1024 / world size). Thus, with a world size of 1024, the maximum pixel resolution for an individual cell is a 1x1 pixel.

7.6.2 Transients and cycles

The most evident feature of a state transition graph is the size of its base cycle and transient trees. A number of studies have aimed to discover laws describing how the size of these features scales with the size of the lattice on which the cellular automaton operates. It appears generally that transient and cycle lengths are constant for simple rules, exponential for chaotic rules, and power-law (or some more exotic scaling) for complex rules. Some evidence suggests that transients and cycles for a given chaotic rule normally scale following the same law, while complex rules may exhibit features which scale in different ways.

One of the main services CA can render to complexity theory is to help connect physics to computation theory. One way to apply computation theory to physics is to find a map from the behavior of the physical system onto the operation of a computing machine. CA, like many physical systems, are governed by local, translationally invariant, interactions. The connections with computers, in particular parallel computers, derive from considering each cell in the cellular automaton as performing logical operations on its inputs. In the interpretation of CA as computers, problems are posed coded into the initial configuration of cell states on the lattice, and solutions decoded from some configuration ultimately achieved under evolution of the automaton.

One method used in computation theory to characterize the complexity of a computing machine is to measure the time it takes the machine to produce an answer to a problem, at which point the operation of the machine halts. One studies how this halting time depends on the length of the problem in some suitable encoding. We can thus think of a machine composed of the cellular automaton itself operating on a finite lattice with periodic boundary conditions, along with some mechanism for recognizing that a configuration has been visited twice. The problem posed to this machine is to determine the projection under the CA rule of the initial configuration onto the invariant set; otherwise said, the point at which an initial configuration hits a cycle. The machine operates by applying the cellular automaton repeatedly to an initial configuration until some configuration has been visited twice; that configuration is the solution. Any initial configuration must eventually enter a cycle, hence this machine will always halt. Topologically, the halting time is simply the size of the set of configurations, H, reachable from a given initial configuration. The cycle time is the size of the largest subset, C, in H whose image under the rule, is equal to itself, i.e., such that . The transient time is the size of the complement, T, of C in H. The halting configuration is then , or the initial configuration itself if it is chosen on a cycle. These times may thus be discussed independently of any particular model or method of computation. However, the program described above is quite reasonably the shortest which computes the halting configuration for arbitrary CA on arbitrary initial conditions. Hence, the halting time is related to the "logical depth" of the halting configuration.

In complex rules the scaling may depend markedly on which statistic is chosen. Distributions with long tails, such as those characteristic of rule 20, are common in physics. It may be suggested, then, that when computation-theoretic measures of complexity are applied to physical problems, the possibility that averages may not be sufficient measures of distributions (indeed may not exist) should be explicitly taken into account.

The connection between complexity as defined in terms of behavior on finite systems with complexity as defined in terms of the statistical properties of infinite systems is subtle and deserves further exploration. The rules showing exponential growth with system size of topological transients and cycles exhibit rapid

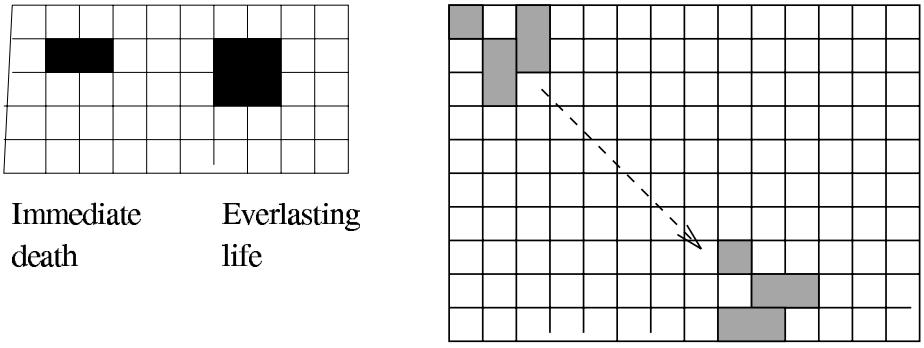

Figure 7.6: The object on the left suffers immediate extinction. That in the centre is in stasis. The right hand object is a glider which moves across screen by changing its shape.

convergence to invariant statistical properties both in simulation experiments on large systems and in generalized mean-field approximations. Complex rules, on the other hand, show sub-exponential growth of topological transients, coupled with slow statistical convergence.

7.7 The Game of Life

Cellular automata were invented by John von Neumann when he became interested in creating self-replicating machines (he was thinking of mathematical machines). His central insight is that what you require to do is provde the machine with its own blueprint (cf DNA).

Probably the most widely discussed and investigated cellular automata is that known as the Game of Life which was developed by John Conway. The Game is played on a square draughts-like board (and so each cell has precisely 8 neighbours) with only three very simple rules:

1. A cell that is white at one instant becomes black at the next if it has precisely three black neighbours

2. A cell that is black at one instant becomes white at the next if it has four or more black neighbours

3. A cell that is black at one instant becomes white at the next if it has one or no black neighbours

All other cells retain their colour.

Life is started with a small black object and the rest of the board white. Two very simple starting shapes are shown in Figure 7.6. These are of no great interest since the first immediately dies while the second reproduces itself without change for all time.

More complicated (and more interesting) objects are possible, however, such as a "glider" which moves across the screen changing its shape in a regular manner. One particular glider is shown on the right of the diagram. Further we can create "glider-guns" which emit regular streams of gliders. Finally we can position our streams of gliders so that one knocks out the other. It is using objects such as these that we can prove that CAs are capable of being thought of as computers. For example if we wish to represent the (binary) number 1100111, we could do so using

```
glider glider noglider noglider glider glider glider
```

where the nogliders have been removed from a stream of gliders by a collision with another stream. By positioning glider-guns in the appropriate positions, we can perform any logic operation.

7.8 Artificial Life

The Game of Life described in the last section is the quintessential reductionist system; every outcome in the game is absolutely deterministic and all depends on only three simple rules. Physicists would be delighted with this universe - there is an absolutely provable Theory of Everything.

However recently researchers have grown interested in modelling behaviour of a number of autonomous agents all interacting together with a variety of emergent behaviours evolving. Again from the Santa Fe Institute:

7.8.1 Emergent Behaviour

Reviewing the different home-grown software "laboratories" developed by complexity researchers, it becomes apparent that most of the complex systems under study share a common "swarm-like" architecture. The essential characteristic of this kind of system is a non-centralized collection of relatively autonomous entities interacting with each other and a dynamic environment. Typically, there is no central authority dictating the behavior of the collection of individuals: each of the many individuals making up the "swarm" makes its own behavioral choices on the basis of its own sampling and evaluation of the world, its own internal state, and through communication with other individuals. Entities that may appear at first glance to be central authorities, such as the ruler of a country, are often just as much caught up in the group dynamic as the rest of the agents.

We use the term "swarm" in a general sense to refer to any such loosely structured collection of interacting agents. The classic example of a swarm is a swarm of bees, but the metaphor of a swarm can be extended to other systems with a similar architecture. An ant colony can be thought of as a swarm whose individual agents are ants, a flock of birds is a swarm whose agents are birds, traffic is a swarm of cars, a crowd is a swarm of people, an immune system is a swarm of cells and molecules, and an economy is a swarm of economic agents. Although the notion of a swarm suggests an aspect of collective motion in space, as in the swarm of a flock of birds, we are interested in all types of collective behavior, not just spatial motion.

What makes swarms scientifically interesting, and often mathematically intractable, is the coupling between the individual and the group behaviors. Although the individuals are usually relatively simple, their collective behavior can be quite complex. Swarms allow us to focus directly on the fundamental roots of complexity: they capture the point at which simplicity becomes complexity.

The behavior of a swarm as a whole emerges in a highly nonlinear manner from the behaviors of the individuals. This emergence involves a critical feedback loop between the behavior of the individuals and the behavior of the whole collection. In a swarm, the combination of individual behaviors determines the collective behavior of the whole group. In turn, the behavior of the whole group determines the conditions (spatial and temporal patterns of information) within which each individual makes its behavioral choices (Figure 1). These individual choices again collectively determine the overall group behavior, and on and on, in a never-ending loop.

This inter-level feedback loop makes traditional linear analysis difficult, if not impossible. "Analysis" literally means to take a system apart into its constituent pieces and then compose our understanding of the pieces to form an understanding of the whole system. But for systems that depend critically on the interactions between parts, analysis can miss essential aspects of the system. Synthesis is the act of putting pieces together to study what they do collectively. Understanding complex systems requires synthesis as well as analysis.

For most systems, the effects of parts on each other must be determined by general computation, not just by simple numerical techniques such as solution of differential equations. Swarm-like architectures constitute a computational paradigm in their own right, but one which has never been made accessible to researchers in a usable, general-purpose form.

The evolution of swarm-like behaviour is a current area of active research.

7.9 Complex Behaviour in Robots

One of the major areas where we see complex behaviour emerging from simple rules is in the field of robotics in which Brooks from MIT has determined a new way of creating interesting robotic behaviours. As with natural language processing and machine vision, GOFAI uses a top-down symbolic approach to defining robot behaviours and, as with these topics, has run into a wall with respect to creating intelligent robots.

Brooks developed a methodology based on learning to do simple things first and learning them well. Now more complex behaviours are learned but are built on top of the simple learned behaviours. The simple behaviours are never altered after they have been learned. "When something works don't mess with it; build on top of it". Thus a sequence of more sophisticated behaviour might be

1. Avoid contact with objects
2. Wander aimlessly
3. Explore world
4. Build internal map
5. Notice changes in the environment
6. Formulate travel plans
7. Anticipate and modify plans accordingly

But if there is a conflict the simpler (lower level) action kicks in. For example, if `explore world` is in charge but there is a danger of hitting another object, `avoid contact with objects` kicks in.

The basic idea has been likened to evolution in that code is never changed, merely ignored.

Brooks also uses the effect of an action as the learning medium for the action. He describes using the real world as a communication medium between the parts. In the above example the robot's sensors would be in communication with the `avoid contact with objects` module and this would initiate its resumption of control.

Such ideas are having a big impact on the creation of intelligent software agents - "softbots" - whose behaviours are layered and adaptable.

7.10 Application - Cryptography

The reader unfamiliar with these concepts should take a moment to consider the cryptanalysis of the so-called Caesar cipher, reputed to have been used by Caesar to communicate with his troops. It consists of a pair of concentric rings. On each ring the letters of the alphabet are written in order. The key of the system is the displacement of the outer ring with respect to the inner ring. To send an encrypted message, the sender emits in sequence the letters on the inner ring which correspond to the letters on the outer ring contained in the message. The receiver reverses the process, reading off from the outer ring letters which correspond to the letters on the inner ring received. While a fair amount of ciphertext might be required in a passive ciphertext-only attack before the key is guessed, a ciphertext-plaintext pair for a single letter reveals the key in any other attack.

A cryptographer's ideal encryption scheme is an operation on a message which renders the message fully meaningless to anyone who does not possess a decryption key, yet in no way degrades the meaning extractable by anyone who does possess the key. An ideal practical code is in addition fast on both encryption and decryption, uses a key of manageable size, and produces no expansion of the data upon encryption.

A provable ideal practical encryption method in this sense in not yet known. Cryptographic research proceeds by the proposition of cryptosystems, followed by attempts to break the the proposed systems. To break a cryptosystem means to discover the meaning of messages encrypted by the system without being handed the secret key. It is generally assumed in academic cryptography that the mechanism of encryption in all its detail is known to the cryptanalyst, the only information lacking being the secret key. Breaking a cryptosystem means reconstructing the key through observations of the cryptosystem in operation. The

type of observations on and manipulations of the cryptosystem which are performed determine the mode of cryptanalytic attack. The first kind of attack is passive attack, in which the cryptanalyst can only make observations on the cryptosystem as it performs. In a ciphertext-only attack, the cryptanalyst has access only to a stream of ciphertext coming from a cryptosystem loaded with its secret key. The cryptanalyst attempts to find statistical regularities in the stream of ciphertext, departures from randomness which might reveal the nature of the key. All but the most naive cryptosystems produce ciphertext with a high degree of randomness, so that a cryptosystem which falls prey to this kind of attack is considered very weak. A stronger passive attack allows the cryptanalyst observations both of a stream of ciphertext and the corresponding message stream which produced it. This is called a known-plaintext attack. Again, cryptography has progressed to the point where cryptosystems susceptible to a known-plaintext attack hold little interest.

More important are the active attacks. Here cryptanalysts can opt to have plaintext of their choosing encrypted and see the ciphertext which results (a chosen-plaintext attack). Similarly, a chosen-ciphertext attack permits ciphertext of the cryptanalyst's design to be compared with the corresponding plaintext. By current cryptographic standards, a good cryptosystem must resist attacks which permit both plaintext and ciphertext to be chosen, and according to any strategy preferred by the cryptanalyst.

7.10.1 Cryptography and complex systems theory

When we describe a dynamical system as "complex" we mean that aspects of its behavior bear a cryptic relationship with the simple evolution laws which define it. One way to unpack the meaning of "cryptic" in complex systems theory is to connect it with more precise meanings in cryptography. To bridge dynamical systems and cryptography one may attempt to make codes based on dynamical systems and use methods of cryptography to evaluate them. Cryptography and complex systems theory are linked by methodology as well as subject matter. It should be obvious that all of the cryptographer's methods of attack have analogies in the methods of a dynamical systems theorist. The analogies are particularly tight in the study of so-called iterated cryptosystems. An iterated cryptosystem is one in which a cryptographically weak transformation is applied repeatedly to a message, so that the composed transformation is strong. The most well-known and well-used cryptosystem is an iterated cryptosystem. It is known as the Data Encryption Standard, or DES. The DES encryption/decryption algorithm consists of 16 rounds of a transformation designed to fully mix message information together with random key information.

7.10.2 Cryptosystems based on cellular automata

The following thoughts on how to produce an ideal practical encryption scheme must be natural as they have occurred independently to a number of students of dynamical systems: The future state of a (chaotic) dynamical system depends sensitively on its initial state. After enough time has elapsed the initial condition is forgotten. Yet, since the system is deterministic, the same trajectory will always be traced out from the same initial condition. Let the secret key of the cryptosystem be the initial state of a publicly known dynamical system. Then a user sharing a key with another can send a secret message by combining it with the trajectory swept out by the dynamical system operating on the secret initial state. Anyone who does not know the initial state would not be able to recreate the trajectory and thus would not be able to disentangle it from the encrypted message.

One version of this idea employs cellular automaton rule 30 to generate temporal sequences which have a high degree of randomness[18]. The secret key is the initial state of the system. Senders run the initial state forward under the rule, and XOR's the temporal sequence generated with their message to produces a ciphertext. Receivers then run the same cellular automaton forward on the same initial state to reproduce the temporal sequence. XORing again recovers the plaintext. Note that: 1) the key is a state of the system, 2) the message is encrypted by combining it with an information stream generated by forward iteration of the system, and 3) the message is decrypted by combining the ciphertext with an information stream again generated by forward iteration of the system.

Consideration of inverse as well as forward iteration of dynamical systems opens up some new ways to use dynamical systems for encryption. One possibility, which again has occurred independently to a number of investigators, is to concentrate on reversible dynamical systems. Using a reversible dynamical system, a message can be encrypted by encoding it as a state of the system and then running the system forward in

time some distance. The resulting state is the ciphertext. To decrypt the ciphertext, the system is inverse iterated the same number of time steps as were used in encryption, recovering the plaintext as a state of the system. Note the contrast with the systems considered above in which only forward iteration is used. In those systems, the key is a state of the system and the system is fixed. When forward and inverse iteration is used, the key is the dynamical system itself. The key operates directly on the message to encrypt and decrypt it, while in the previous systems the information generated by the dynamical system is combined indirectly, so to speak, externally, with the message information.

Reversible CA may be particularly valuable in public-key encryption. A public-key cryptosystem uses two keys, one key is used for encryption, the other for decryption. One key is held in private, the other rendered public. Kari has proposed a system in which the public key is a cellular automaton inverse to the private key cellular automaton. The security of public-key cryptosystems depends on the difficulty of finding the private key given knowledge of the public key and/or chosen plain and ciphertext. Kari bases much of his reasoning on a result of his which shows that even deciding whether a given cellular automaton in more than one dimension has an inverse is impossible. In general, the mathematical theory of cellular automata which is relevant to cryptography is more well-developed than the theory of irreversible cellular automata. In particular, one knows that if a cellular automaton has an inverse, that inverse is also a cellular automaton. Many practical problems remain, however, concerning how to choose good public-key/private-key pairs, how these should be applied to encrypt a message, etc.

7.11 Exercises

1. Check that the other patterns for the Hopfield net discussed on page 78 converge as stated. Objectives 1, 5.

2. Create a Hopfield network to store the patterns

$$\begin{aligned} \mathbf{x}_1 &= (1,1,1,1,1) \\ \mathbf{x}_2 &= (-1,-1,-1,1,1) \\ \mathbf{x}_3 &= (1,-1,1,-1,1) \end{aligned}$$

   Show that, with asynchronous updating, each point is stable. To what will the network converge when the input pattern (-1,1,1,1,1) is presented? Find the spurious state of this network comprised of mixtures of the three patterns. Show that it is stable. Show that the reverse of each of the fundamental memories is a stable pattern. Objectives 1, 5.

3. Simulated Annealing and Genetic Algorithms are designed to tackle the same type of problems. Compare them. Objective 3

4. Create a one dimensional CA of length 32 with a radius of 1, time 10 and rule

   (a) Rule 22

   (b) Rule 30

   (c) Rule 110

   (d) Rule 54

   Objective 4.

Chapter 8

Faster Supervised Learning

8.1 Objectives

After this Chapter, you should

1. understand the innate difficulty with error descent.
2. know several methods of improving convergence.
3. be able to implement a Radial Basis Function network.
4. be able to compare supervised learning methods.

8.2 Radial Basis Functions

We noted in Chapter 4 that, while the multilayer perceptron is capable of approximating any continuous function, it can suffer from excessively long training times. In this chapter we will investigate methods of shortening training times for artificial neural networks using supervised learning.

A typical radial basis function (RBF) network is shown in Figure 8.1. 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 vectors and the output neurons merely calculate a linear (weighted) combination of the hidden neurons' outputs.

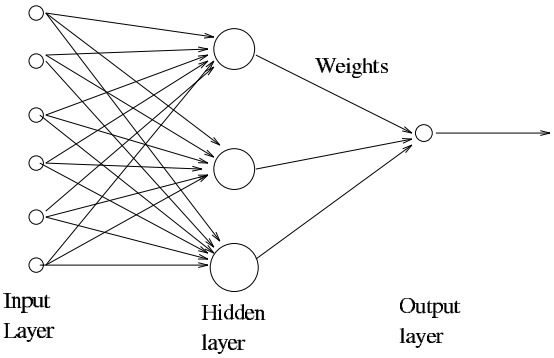

Figure 8.1: A typical radial basis function network. Activation is fed forward from the input layer to the hidden layer where a (basis) function of the Euclidean distance between the inputs and the centres of the basis functions is calculated. The weighted sum of the hidden neuron's activations is calculated at the single output neuron

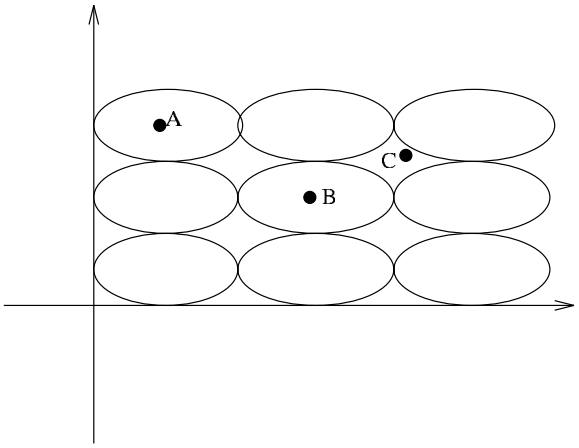

Figure 8.2: A "tiling" of a part of the plane

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

$$y = \mathbf{w}.\phi(\mathbf{x}) = \mathbf{w}^T\phi(x) \tag{8.1}$$

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 8.2. 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.

We will use the XOR function to demonstrate that expressing the input patterns in the hidden layer's basis permits patterns which were not linearly separable in the original (input) space to become linearly separable in the hidden neurons' space (see Haykin page 241).

8.2.1 XOR Again

We will use the XOR pattern which is shown diagrammatically in Figure 3.4. We noted earlier that this set of patterns is not linearly separable (in the input (X,Y)-space). Let us consider the effect of mapping the

Input pattern	$\phi_1(\mathbf{x})$	$\phi_2(\mathbf{x})$
(0,0)	0.135	1
(0,1)	0.368	0.368
(1,0)	0.368	0.368
(1,1)	1	0.135

Table 8.1: The activation functions of the hidden neurons for the 4 possible inputs for the XOR problem

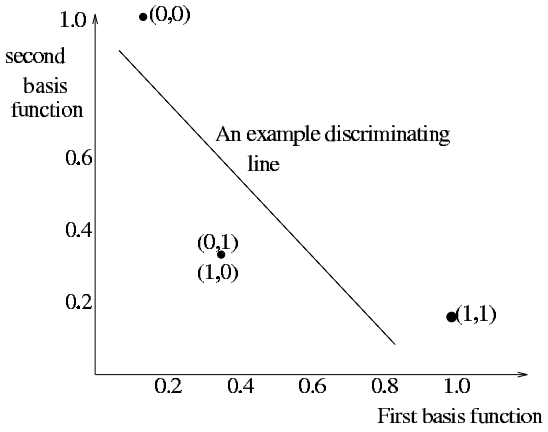

Figure 8.3: The hidden neurons' activations are plotted on a graph whose axes are the neuron's functions. One (of many possible) discriminating lines is drawn.

inputs to a hidden layer with two neurons with

$$\begin{aligned} \phi_1(\mathbf{x}) &= \exp(-\| \mathbf{x} - \mathbf{c}_1 \|^2), \text{ where } \mathbf{c}_1 = (1,1) \\ \phi_2(\mathbf{x}) &= \exp(-\| \mathbf{x} - \mathbf{c}_2 \|^2), \text{ where } \mathbf{c}_2 = (0,0) \end{aligned}$$

Then the two hidden neurons' outputs on presentation of the four input patterns is shown in Table 8.1. Now if we plot the hidden neuron's outputs in the ϕ_1, ϕ_2 basis, we see (Figure 8.3) that the outputs are linearly separable.

8.2.2 Learning weights

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.

We may train the network now using the simple LMS algorithm (see Chapter 3) in the usual way. If the sum of the errors over all N patterns is

$$E = \frac{1}{2}\sum_{i=1}^{N}(t^i - o^i)^2 = \frac{1}{2}\sum_{i=1}^{N}(t^i - \sum_j w_j\phi_j(\mathbf{x}^i))^2 = \frac{1}{2}\sum_{i=1}^{N} e_i^2 \tag{8.2}$$

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

$$E^i = \frac{1}{2}(t^i - \sum_j w_j\phi_j(\mathbf{x}^i))^2 = \frac{1}{2}e_i^2 \tag{8.3}$$

and so we can create an on-line learning algorithm using

$$\frac{\partial E^i}{\partial w_k} = -(t^i - \sum_j w_j\phi_j(\mathbf{x}^i))\phi_k(\mathbf{x}^i) = -e^i.\phi_k(\mathbf{x}^i) \tag{8.4}$$

Therefore we will change the weights after presentation of the i^{th} input pattern $\mathbf{x}^i$ by

$$\Delta w_k = -\frac{\partial E^i}{\partial w_k} = e^i.\phi_k(\mathbf{x}^i) \tag{8.5}$$

8.2.3 Approximation problems

We may view the problem of finding the weights which minimise the error at the outputs as an approximation problem. Then the learning process described above is equivalent to finding that line in the hidden layer's space which is optimal for approximating an unknown function. Note that this straight line (or in general hyperplane) in the hidden layer space is equivalent to a curve or hypersurface in the original space. Now, as before, our aim is not solely to make the best fit to the data points on which we are training the network; our overall aim is to have the network perform as well as possible on data which it has not seen during learning. Previously we described this as generalisation. In the context of the RBF network, we may here view it as interpolation: we are fitting the RBF network to the actual values which it sees during training but we are doing so with a smooth enough set of basis functions that we can interpolate between the training points and give correct (or almost correct) responses for points not in the training set.

If the number of points in the training set is less than the number of hidden neurons, this problem is underdetermined and there is the possibility that the hidden neurons will map the training data precisely and be less useful as an approximation function of the underlying distribution. Ideally the examples from which we wish to generalise must show some redundancy, however if this is not possible we can add some constraints to the RBF network to attempt to ensure that any approximation it might perform is valid. A typical constraint is that the second derivative of the basis functions with respect to the input distribution is sufficiently small. If this is small, the weighted sum of the functions does not change too rapidly when the inputs change and so the output values (the y's) should be reasonable approximations of the true values of the unknown function at intermediate input values (the x's) between the training values.

8.2.4 RBF and MLP as Approximators

We examine the approximation properties of both a multi-layered perceptron and a radial basis function on a problem which we met in Chapter 4: a noisy version of a simple trigonometric function. Consider the set of points shown in Figure 8.4 which are drawn from $\sin(2\pi x)$ + noise. The convergence of radial basis function networks is shown in Figure 8.5. 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. 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 8.6. In order to compare the convergence of the RBF network with an MLP we repeat the experiment performed in Chapter 4 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 8.7.

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.

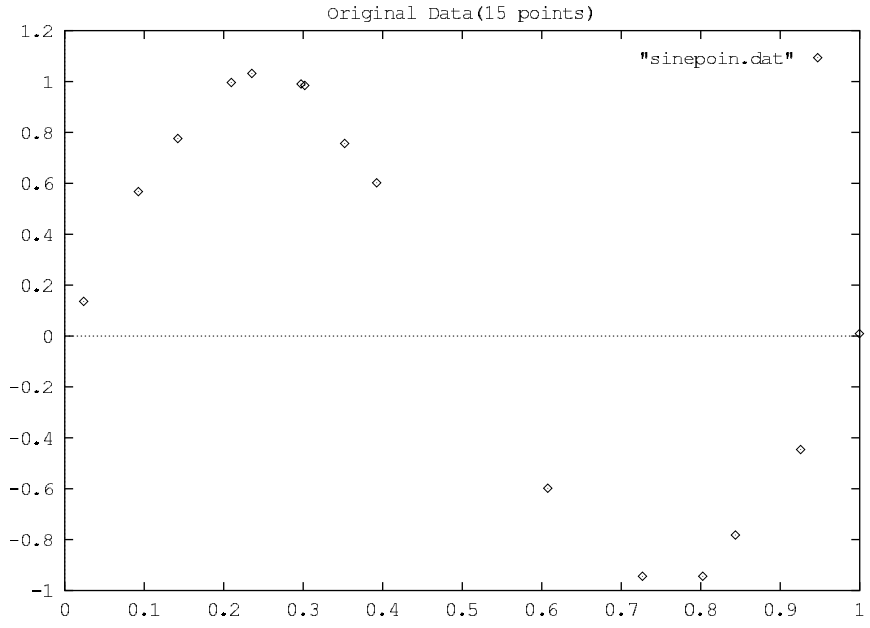

Figure 8.4: 15 data points drawn from a noisy version of $\sin(2\pi x)$.

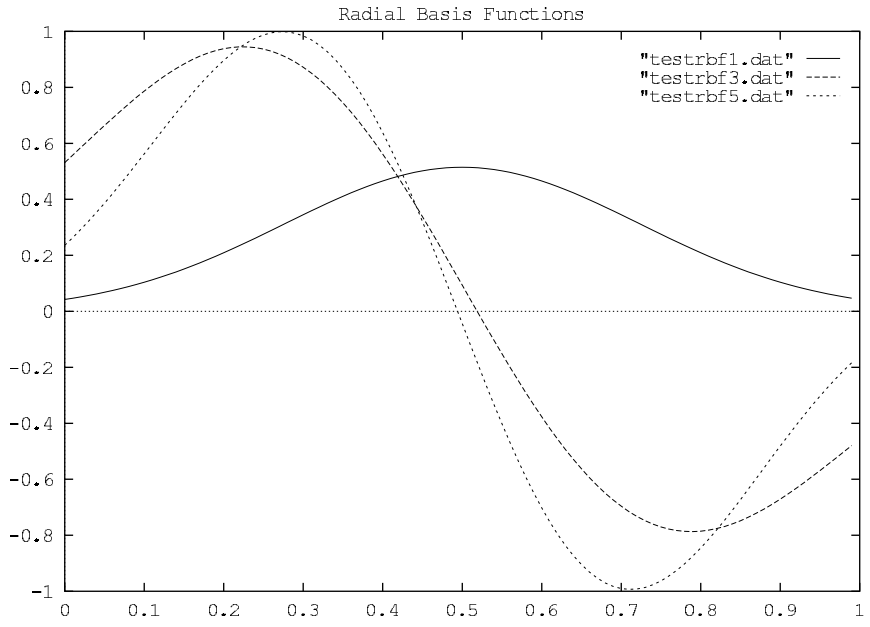

Figure 8.5: Approximation of the above data using radial basis function networks with 1, 3 and 5 basis functions.

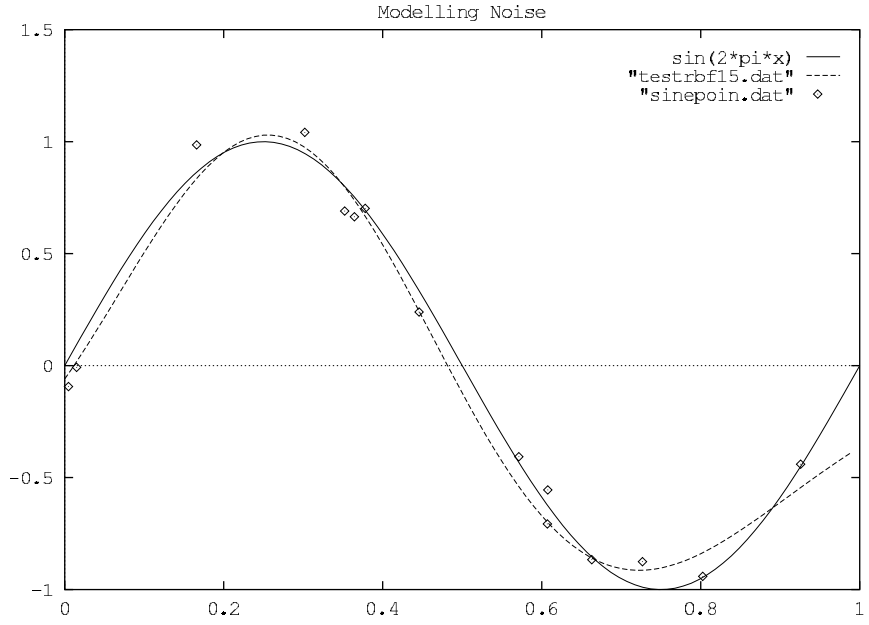

Figure 8.6: The data points which were corrupted by noise, the underlying signal and the network approximation by a radial basis function net with 15 basis functions.

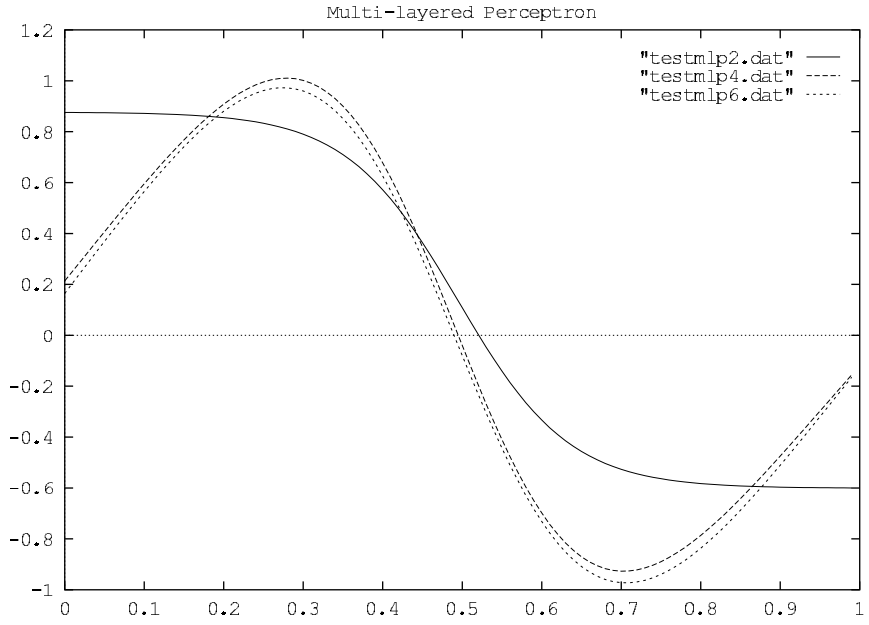

Figure 8.7: A comparison of the network convergence using multilayered perceptrons on the same data.

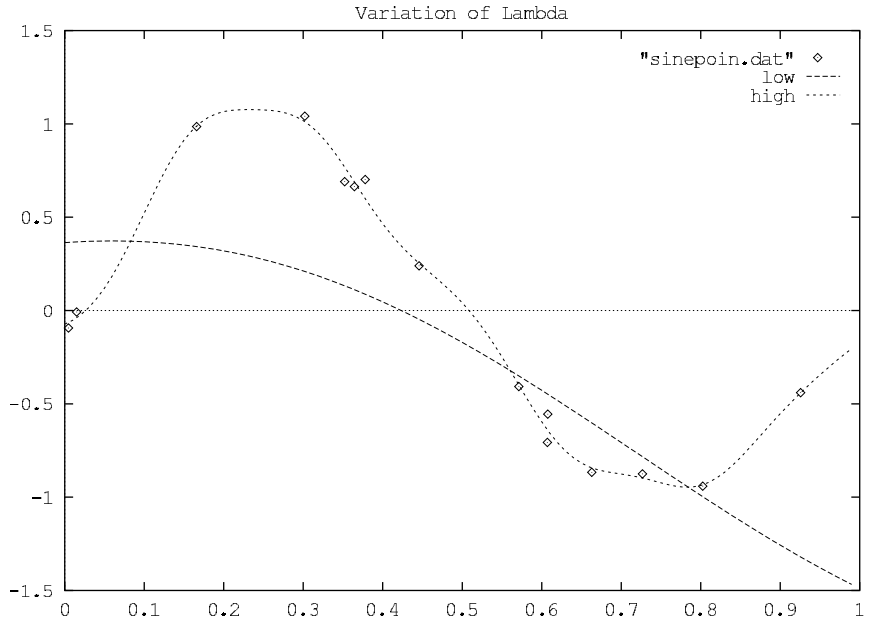

Figure 8.8: Using a basis function with a wide neighbourhood is equivalent to smoothing the output. A narrower neighbourhood function will more closely model the noise.

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 8.8 in which we illustrate this fact on the same type of data as before but use a value of $\lambda = 1$ and $\lambda = 100$ for the parameter λ when calculating the output of the basis neurons.

$$y = exp(-\lambda \parallel \mathbf{x}_i - \mathbf{c}_i \parallel^2) \tag{8.6}$$

8.2.5 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 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.

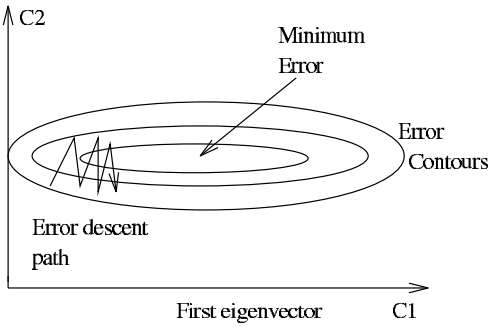

Figure 8.9: The path taken by the route of fastest descent is not the path directly to the centre (minimum) of the error surface.

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.

8.2.6 Finding the Centres of the RBFs

If we have little data, we may have no option but to position the centres of our radial basis functions at the data points. However, as noted earlier, such problems may be ill-posed and lead to poor generalisation. If we have more training data, several solutions are possible:

1. Choose the centres of the basis functions randomly from the available training data.

2. 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.

3. We can use a generalisation of the LMS rule:

$$\Delta c_i = -\frac{\partial E}{\partial c_i} \tag{8.7}$$

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.

8.3 Error Descent

The backpropagation method as described so far is innately slow. The reason for this is shown diagrammatically in Figure 8.9. 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.

8.3.1 Mathematical Background

The matrix of second derivatives is known as the Hessian and may be written as

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_m} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_m} & \frac{\partial^2 E}{\partial w_2 \partial w_m} & \cdots & \frac{\partial^2 E}{\partial w_m^2} \end{bmatrix} \tag{8.8}$$

If we are in the neighbourhood of a minimum $\mathbf{w^*}$, we can consider the (truncated) Taylor series expansion of the error as

$$E(\mathbf{w}) = E(\mathbf{w}^*) + (\mathbf{w} - \mathbf{w}^*)^T \nabla \mathbf{E} + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H} (\mathbf{w} - \mathbf{w}^*) \tag{8.9}$$

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 8.9 with

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H} (\mathbf{w} - \mathbf{w}^*) \tag{8.10}$$

The eigenvectors of the Hessian,$\mathbf{c}_i$ are defined by

$$\mathbf{H}\mathbf{c}_i = \lambda_i \mathbf{c}_i \tag{8.11}$$

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

$$\mathbf{w} - \mathbf{w}^* = \sum_i \alpha_i \mathbf{c}_i \tag{8.12}$$

Now since $\mathbf{H}(\mathbf{w} - \mathbf{w}^*) = \sum_i \lambda_i \alpha_i \mathbf{c}_i$ we have

$$E(\mathbf{w}) = E(\mathbf{w}*) + \frac{1}{2} \sum_i \lambda_i \alpha_i^2 \tag{8.13}$$

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 8.9 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

$$\Delta \alpha_i = -\eta \lambda_i \alpha_i \tag{8.14}$$

and so

$$\alpha_i^{new} = (1 - \eta \lambda_i) \alpha_i^{old} \tag{8.15}$$

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

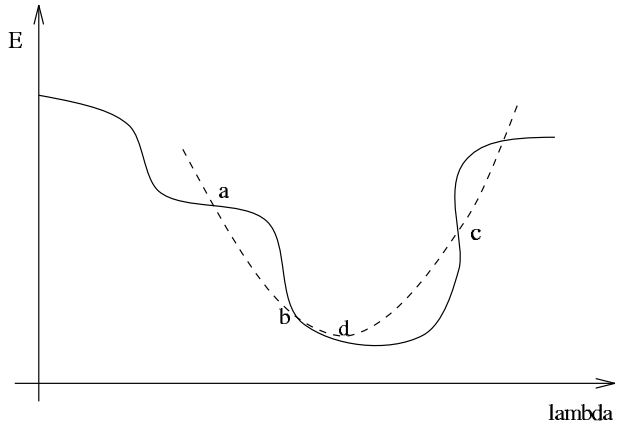

Figure 8.10: The error function as a function of λ. The (unknown) error function is shown as a solid line. The fitted parabola is shown as a dotted line with minimum value at d.

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.

8.3.2 QuickProp

Fahlman has developed an heuristic which attempts to take into account the curvature of the error surface at any point by defining

$$\Delta w_{ij}(k) = \begin{cases} \alpha_{ij}(k)\Delta w_{ij}(k-1), & \text{if } \Delta w_{ij}(k-1) \neq 0 \\ \eta_0 \frac{\partial E}{\partial w_{ij}}, & \text{if } \Delta w_{ij}(k-1) = 0 \end{cases} \tag{8.16}$$

where

$$\alpha_{ij}(k) = \min\left(\frac{\frac{\partial E(k)}{\partial w_{ij}}}{\frac{\partial E(k-1)}{\partial w_{ij}} - \frac{\partial E(k)}{\partial w_{ij}}}, \alpha_{max}\right) \tag{8.17}$$

8.4 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

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda^{(t)}\mathbf{d}^{(t)} \tag{8.18}$$

in order to minimise

$$E(\lambda) = E(\mathbf{w}^{(t)} + \lambda^{(t)}\mathbf{d}^{(t)}) \tag{8.19}$$

We show E as a function of λ in Figure 8.10. 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.

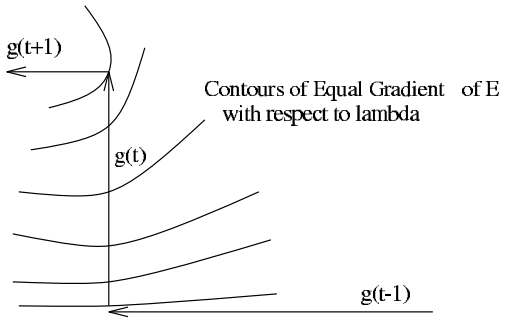

Figure 8.11: After minimising error in one direction, the new direction of fastest descent is perpendicular to that just traversed. This leads to a zigzagging approach to the minimum.

8.4.1 Conjugate Gradients

Now we must find a method to find the direction $\mathbf{d}$ 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 8.11, 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 $\mathbf{g}$ for the batch of patterns as usual. Call this $\mathbf{g}(0)$, the value of $\mathbf{g}$ at time 0 and let $\mathbf{p}(0) = \mathbf{g}(0)$.
- Update the weights according to
$$\Delta\mathbf{w}(n) = \eta(n)\mathbf{p}(n) \tag{8.20}$$
- Calculate the new gradient vector $\mathbf{g}$(n+1) with the usual method
- Calculate the parameter $\beta(n)$.
- Recalculate the new value of $\mathbf{p}$ using
$$\mathbf{p}(n+1) = -\mathbf{g}(n+1) + \beta(n)\mathbf{p}(n) \tag{8.21}$$
- 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

$$\beta(n) = \frac{\mathbf{g}^T(n+1)\mathbf{g}(n+1)}{\mathbf{g}^T(n)\mathbf{g}(n)} \tag{8.22}$$

The calculation of the parameter $\eta(n)$ is done to minimise the cost function

$$E(\mathbf{w}(n) + \eta\mathbf{p}(n)) \tag{8.23}$$

As with the Newton method (see below), convergence using this method is much faster but computationally more expensive.

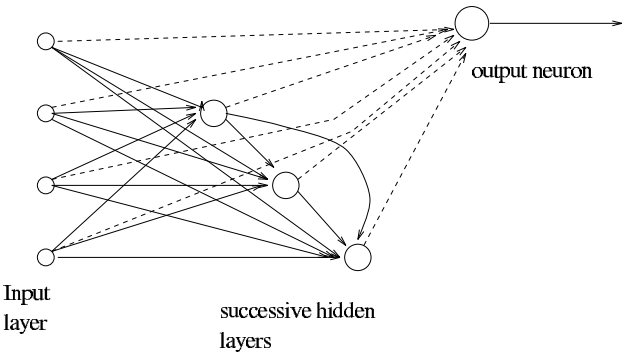

Figure 8.12: A cascade correlation network. Note that there are many hidden "layers" each containing only a single neuron. The dotted lines show weights which continue to be trainable during the whole simulation. The solid lines represent weights which are frozen after the initial learning period.

8.5 Cascade Correlation

One of the themes of our investigation of supervised learning networks has been the need to ensure that the networks perform well not just on the training set but also on data on which the network has not seen during training. We have shown that this can be achieved by cutting down on the number of parameters (weights) in the networks: if there are enough parameters, the network will simply act as a look-up table since it will have memorised the data.

Cascade correlation networks attempt to start off with a very simple network and then add neurons only when they are needed. A typical cascade correlation network is shown in figure 8.12. The algorithm is

1. Start with a network consisting of only an input and an output layer (in Figure 8.12, the output layer is only a single neuron).
2. Connect all input neurons to all output neurons.
3. Train the weights using the LMS algorithm till the error stops decreasing.
4. If the error is not small enough, generate a pool of "candidate neurons". Every candidate neuron is connected to all input neurons and all existing hidden neurons.
5. Try to maximise the correlation between the activation of each candidate neuron and the residual error in the network by training all the links leading to the candidate neurons. Stop training when the learning does not decrease the residual error.
6. Choose the candidate neuron with maximum correlation with the error at the outputs. Freeze its incoming weights and add it to the network creating a new link between it and the output neurons.
7. Loop back to 3 till the error is sufficiently small

It is important to note that the candidate neurons do not interact with each other and so they can be tested in isolation. To maximise the correlation (actually the covariance) between the candidate and the previous residual error, we first define

$$S = \sum_i |\sum_p (y^p - \langle y \rangle)(E_i^p - \langle E_i \rangle)| \tag{8.24}$$

where y^p is the output of the candidate neuron when the network is presented with the p^{th} pattern, $\langle y \rangle$ is its average value (taken over all input patterns), E_i^p is the residual error at the i^{th} output when the network is presented with the p^{th} pattern and $\langle E_i \rangle$ is its average value. Then just as with other supervised learning methods we differentiate this with respect to the weights but since we wish to *maximise* S, we use

$$\Delta w_j = \frac{\partial S}{\partial w_j} = \sum_i \sum_p \sigma_i (E_i^p - \langle E_i \rangle) f'^p x_j^p \tag{8.25}$$

where σ_i is the sign of the correlation between the candidate's value and the residual error, f'^p is the derivative of the candidate neuron's activation function with respect to the sum of its inputs (for pattern p) and x_j^p is the input the candidate receives from unit j for pattern p.

The cascade correlation architecture takes many of the heuristics out of network creation. It learns fast since we do not have global learning taking place at any time. It will build deep nets in which, it is claimed, the hidden neurons act as high order feature detectors.

8.6 Reinforcement Learning

Reinforcement learning is sometimes distinguished from supervised learning since the teacher does not provide the correct answer to the current set of inputs. The teacher merely provides a training signal in terms of "right" or "wrong" to the network. Therefore the input data is presented to the network, activation is passed through the network just as in a multi-layered perceptron and the teacher then looks at the network's response to that input data. If the network has made the correct response, no action is taken; however, if the network has made an incorrect response, an error signal is sent to the network and it must adjust its weights to make the correct response more likely the next time.

This has been linked to biological learning [Haykin]:

> If an action taken by a learning system is followed by a satisfactory state of affairs, then the tendency of the system to produce that particular action is strengthened or reinforced. Otherwise, the tendency of the system to produce that action is weakened.

We could view GAs as an example of *nonassociative* reinforcement learning since the probability of reproducing may be thought of as the feedback signal from the environment. The field in which reinforcement learning has found its most prominent place is that of control. A typical problem is the pole-balancing experiment in which a vertical pole must be held in a vertical position using only the "bang-bang" backward/forward forces available to a simple cart on which the pole is balanced. A failure is easy to see while success is keeping the pole balanced for as long as possible. A second experiment is the truck-backing up experiment in which articulated vehicles must be reversed into a variety of parking positions. Such tasks take a prohibitively long time to learn using standard supervised learning techniques but have been learned using reinforcement learning.

Typical of reinforcement learning methods (and true of the two examples given above) is that the system is expected to probe its environment and learn from its mistakes. The pole starts of from an almost vertical position and the system attempts to prolong its success learning as it does so. Often the reward may be delayed thereby raising the **credit assignment problem** - how to decide at what stage a set of actions became less than useful.

8.7 Second Order Methods

The most obvious second order method is based on Newton's method. A feature of error descent is that if the slope (gradient of the error with respect to the weights) does not change much locally we can use a large learning rate while if the slope is very changeable we should use a small learning rate or face the danger of overshooting our minimum. Now the rate of change of the slope is the second derivative of the error with respect to the weights. Thus if the second derivative is high we wish a small learning rate while if the second derivative is small we can use a high learning rate. The matrix of second derivatives is known as the Hessian and may be written as

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_m} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_m} & \frac{\partial^2 E}{\partial w_2 \partial w_m} & \cdots & \frac{\partial^2 E}{\partial w_m^2} \end{bmatrix} \tag{8.26}$$

Therefore we use the weight update rule

$$\Delta \mathbf{w} \propto H^{-1} \frac{\partial E}{\partial \mathbf{w}} \tag{8.27}$$

This method is certain to cause faster convergence but the extra computational cost of calculating the Hessian at each iteration may make it infeasible as a practical proposition.

8.8 Exercises

1. The inverse multiquadric function

$$\phi(r) = \frac{1}{(r^2 + c^2)^{\frac{1}{2}}}, c > 0 \tag{8.28}$$

has been suggested as a possible basis function where r is the distance from the centre of the function. Draw a 1 dimensional example of this function and discuss its properties as a basis function. (Objective 3).

2. Find another set of values (centres, weights) with which a radial basis function may solve the XOR problem. (Objective 3).

3. Compare the learning methods of this chapter with standard backprop. (Objectives 2,4).

Appendix A

The perceptron learning Theorem

We review the algorithm here for completness:

1. begin with the network in a randomised state: the weights between all neurons are set to small random values (between -1 and 1).

2. select an input vector, $\mathbf{x}$, from the set of training examples

3. propagate the activation forward through the weights in the network to calculate the output o.

4. if $o^P = t^P$, (i.e. the network's output is correct) return to step 2.

5. else change the weights according to

$$\Delta w_i = \eta . x_i^P (t^P - o^P) \tag{A.1}$$

   where η is a small positive number known as the learning rate. Return to step 2.

Notice that the update rule may be written as

$$\begin{aligned} \Delta w_i &= \eta . x_i^P (t^P - o^P) \text{ or} \\ \Delta w_i &= \eta . x_i^P (2t^P) \\ \Delta w_i &= \eta . x_i^P T(\mathbf{x}^P) \end{aligned}$$

where T() is the mapping we wish the perceptron to learn.

Theorem *If there exists a set of connection weights,* $\mathbf{w*}$*, which is capable of performing the transformation T, the perceptron learning rule will converge to a solution (not necessarily* $\mathbf{w*}$*) in a finite number of steps from any initial choice of weights.*

Note: This is really a remarkable theorem as it shows that we are guaranteed convergence of the weights to a solution of the problem from *any* starting set of weights in a *guaranteed finite* length of time.

Proof: Since the update rule uses the Heaviside or sgn function to calculate the output of the neuron, the length of the vector $\mathbf{w*}$, does not matter; therefore we take $\| \mathbf{w}* \| = 1$. Because $\mathbf{w*}$ is a correct solution, $|\mathbf{w} * .\mathbf{x}| > 0$;i.e. there exists a $\delta > 0$ such that $|\mathbf{w} * .\mathbf{x}| > \delta$ for all inputs,$\mathbf{x}$. Let the current weight vector be $\mathbf{w}$. Then the angle between $\mathbf{w*}$ and $\mathbf{w}$ is defined by $\cos(\alpha) = \mathbf{w} * .\mathbf{w} / \| \mathbf{w} \|$.

Then when we use the perceptron learning rule to update $\mathbf{w}$, for a given $\mathbf{x}$, we get $\mathbf{w'} = \mathbf{w} + \mathbf{\Delta w}$ where $\mathbf{\Delta w} = T(\mathbf{x}).\mathbf{x}$, taking $\eta = 1$. Then

$$\begin{aligned} \mathbf{w'}.\mathbf{w}* &= \mathbf{w}.\mathbf{w}* + T(\mathbf{x}).\mathbf{w} * .\mathbf{x} \\ &= \mathbf{w}.\mathbf{w}* + sgn(\mathbf{w} * .\mathbf{x})\mathbf{w} * .\mathbf{x} \\ &> \mathbf{w}.\mathbf{w}* + \delta \end{aligned}$$

Now,

$$\begin{aligned} \| \mathbf{w}' \|^2 &= \| \mathbf{w} + T(\mathbf{x}).\mathbf{x} \|^2 \\ &= \mathbf{w}^2 + 2T(\mathbf{x})\mathbf{w}.\mathbf{x} + \mathbf{x}^2 \\ &< \mathbf{w}^2 + \mathbf{x}^2, \text{ since } T(\mathbf{x}) = -sgn(\mathbf{w}.\mathbf{x}) \\ &= \mathbf{w}^2 + K \end{aligned}$$

After t modifications, we have

$$\begin{aligned} \mathbf{w}(t).\mathbf{w}* &> \mathbf{w}.\mathbf{w}* + t\delta \\ \text{i.e. } \| \mathbf{w}(t) \|^2 &< \mathbf{w}^2 + tK \text{ and so} \\ \cos(\alpha(t)) &= \frac{\mathbf{w}*.\mathbf{w}(t)}{\| \mathbf{w}(t) \|} > \frac{\mathbf{w}*.\mathbf{w} + t\delta}{\sqrt{\mathbf{w}^2 + tK}} \end{aligned}$$

Therefore $\lim_{t\to\infty} \cos(\alpha(t)) \propto \sqrt{t} \to \infty$. But this is impossible since $\cos(\alpha(t))$ has maximum value 1 and so there must be a maximum time for t i.e. the process of learning must stop at some finite time at which point the perceptron must be performing the mappint T() correctly.

Appendix B

Linear Algebra

B.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 B.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 **a.**
- n-dimensional vectors $\mathbf{a} = (a_1, a_2, ..., a_n)^T$. Now no one can visualise the vectors!

B.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.

B.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

$$\begin{aligned} \mathbf{a} &= (a_1, a_2, ..., a_n)^T \text{ and} \\ \mathbf{b} &= (b_1, b_2, ..., b_n)^T \text{ then} \\ \mathbf{a}+\mathbf{b} &= (a_1 + b_1, a_2 + b_2, ..., a_n + b_n)^T \end{aligned}$$

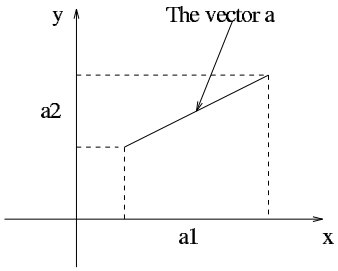

Figure B.1: A two dimensional vector

B.1.3 Length of a vector

In two dimensions we know that the length of the vector, **a** can be found by

$$|\mathbf{a}| = \sqrt{a_1^2 + a_2^2} \tag{B.1}$$

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 $|\mathbf{a}|$.

B.1.4 The Scalar Product of 2 Vectors

In 2D the scalar product of two vectors, **a** and **b** is given by

$$\mathbf{a}.\mathbf{b} = a_1.b_1 + a_2.b_2 \tag{B.2}$$

This can be extended to 3-dimensional vectors as

$$\mathbf{a}.\mathbf{b} = a_1.b_1 + a_2.b_2 + a_3.b_3 \tag{B.3}$$

and, in general,

$$\mathbf{a}.\mathbf{b} = a_1.b_1 + a_2.b_2 + \cdots + a_n.b_n \tag{B.4}$$

The scalar product of **a** and **b** can be viewed as the projection of **a** onto **b.**

B.1.5 The direction between 2 vectors

In 2D the direction between two vectors, **a** and **b** is given by

$$\begin{aligned} \cos\theta &= \frac{\mathbf{a}.\mathbf{b}}{|\mathbf{a}|.|\mathbf{b}|} \\ &= \frac{a_1.b_1 + a_2.b_2 + \cdots + a_n.b_n}{\sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}.\sqrt{b_1^2 + b_2^2 + \cdots + b_n^2}} \end{aligned}$$

B.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.

B.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

$$Act = \sum_i w_i.x_i = \mathbf{w}^T\mathbf{x} = \mathbf{w}.\mathbf{x} \tag{B.5}$$

B.2 Matrices

A matrix is an array (of numbers) such as

$$A = \begin{bmatrix} 2 & 3.5 & 4 & 0.1 \\ 2 & 1.2 & 7 & 9 \\ 9 & 0.6 & 5.99 & 1 \end{bmatrix} \tag{B.6}$$

B.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

$$A^T = \begin{bmatrix} 2 & 2 & 9 \\ 3.5 & 1.2 & 0.6 \\ 4 & 7 & 5.99 \\ 0.1 & 9 & 1 \end{bmatrix} \tag{B.7}$$

Clearly if the array A is m*n then A^T is n*m.

B.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

$$B = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 4 & 3 & 1 & 7 \\ 2 & 1 & 0 & 0 \end{bmatrix} \tag{B.8}$$

then A + B is the matrix

$$A+B = \begin{bmatrix} 3 & 3.5 & 4 & 1.1 \\ 6 & 4.2 & 8 & 16 \\ 11 & 1.6 & 5.99 & 1 \end{bmatrix} \tag{B.9}$$

B.2.3 Multiplication by a scalar

If again,

$$A = \begin{bmatrix} 2 & 3.5 & 4 & 0.1 \\ 2 & 1.2 & 7 & 9 \\ 9 & 0.6 & 5.99 & 1 \end{bmatrix} \tag{B.10}$$

then 3A is that matrix each of whose elements is multiplied by 3

$$3A = \begin{bmatrix} 6 & 10.5 & 12 & 0.3 \\ 6 & 3.6 & 21 & 27 \\ 27 & 1.8 & 17.97 & 3 \end{bmatrix} \tag{B.11}$$

B.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

$$C = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{B.12}$$

Then CB is the matrix

$$CB = \begin{bmatrix} 3 & 1 & 0 & 1 \\ 6 & 4 & 1 & 7 \end{bmatrix} \tag{B.13}$$

B.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

$$I_{2*2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$I_{3*3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ etc.}$$

B.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.*

Appendix C

Calculus

C.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 C.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 C.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 $\frac{dE}{dw} = 0$.

C.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 while the other (orthogonal) direction is kept constant.

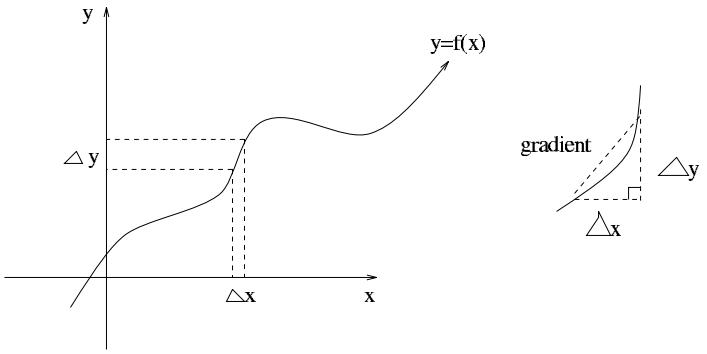

Figure C.1: The ratio $\frac{\Delta y}{\Delta x}$ is the tangent of the angle which the triangle makes with the x-axis i.e. gives the average slope of the curve at this point

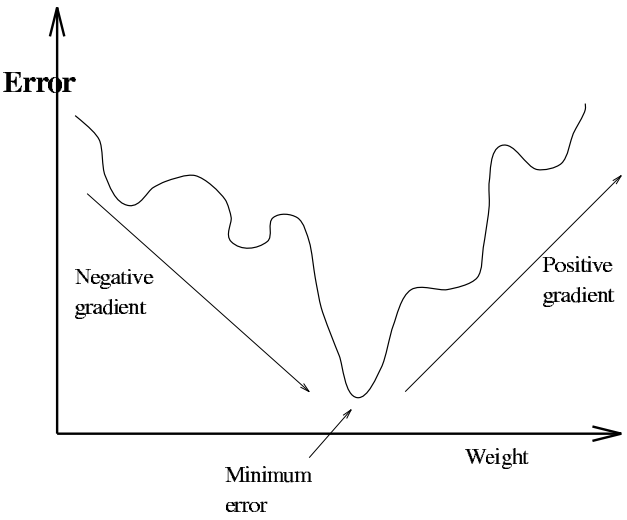

Figure C.2: A schematic diagram showing error descent. In the negative gradient section, we wish to increase the weight; in the positive gradient section, we wish to decrease the weight

C.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 D

Laboratory Exercises