MICHAEL NEGNEVITSKY

Artificial Intelligence

A Guide to Intelligent Systems

Second Edition

ADDISON
WESLEY

Artificial Intelligence

Artificial Intelligence

A Guide to Intelligent Systems

Second Edition

Michael Negnevitsky

For my son, Vlad

Contents

Trademark notice

The following are trademarks or registered trademarks of their respective companies:

KnowledgeSEEKER is a trademark of Angoss Software Corporation; Outlook and Windows are trademarks of Microsoft Corporation; MATLAB is a trademark of The MathWorks, Inc; Unix is a trademark of the Open Group.

See Appendix for AI tools and their respective vendors.

Preface

'The only way not to succeed is not to try.'

Edward Teller

Another book on artificial intelligence ... I've already seen so many of them. Why should I bother with this one? What makes this book different from the others?

Each year hundreds of books and doctoral theses extend our knowledge of computer, or artificial, intelligence. Expert systems, artificial neural networks, fuzzy systems and evolutionary computation are major technologies used in intelligent systems. Hundreds of tools support these technologies, and thousands of scientific papers continue to push their boundaries. The contents of any chapter in this book can be, and in fact is, the subject of dozens of monographs. However, I wanted to write a book that would explain the basics of intelligent systems, and perhaps even more importantly, eliminate the fear of artificial intelligence.

Most of the literature on artificial intelligence is expressed in the jargon of computer science, and crowded with complex matrix algebra and differential equations. This, of course, gives artificial intelligence an aura of respectability, and until recently kept non-computer scientists at bay. But the situation has changed!

The personal computer has become indispensable in our everyday life. We use it as a typewriter and a calculator, a calendar and a communication system, an interactive database and a decision-support system. And we want more. We want our computers to act intelligently! We see that intelligent systems are rapidly coming out of research laboratories, and we want to use them to our advantage.

What are the principles behind intelligent systems? How are they built? What are intelligent systems useful for? How do we choose the right tool for the job? These questions are answered in this book.

Unlike many books on computer intelligence, this one shows that most ideas behind intelligent systems are wonderfully simple and straightforward. The book is based on lectures given to students who have little knowledge of calculus. And readers do not need to learn a programming language! The material in this book has been extensively tested through several courses taught by the author for the

past decade. Typical questions and suggestions from my students influenced the way this book was written.

The book is an introduction to the field of computer intelligence. It covers rule-based expert systems, fuzzy expert systems, frame-based expert systems, artificial neural networks, evolutionary computation, hybrid intelligent systems and knowledge engineering.

In a university setting, this book provides an introductory course for undergraduate students in computer science, computer information systems, and engineering. In the courses I teach, my students develop small rule-based and frame-based expert systems, design a fuzzy system, explore artificial neural networks, and implement a simple problem as a genetic algorithm. They use expert system shells (Leonardo, XpertRule, Level5 Object and Visual Rule Studio), MATLAB Fuzzy Logic Toolbox and MATLAB Neural Network Toolbox. I chose these tools because they can easily demonstrate the theory being presented. However, the book is not tied to any specific tool; the examples given in the book are easy to implement with different tools.

This book is also suitable as a self-study guide for non-computer science professionals. For them, the book provides access to the state of the art in knowledge-based systems and computational intelligence. In fact, this book is aimed at a large professional audience: engineers and scientists, managers and businessmen, doctors and lawyers – everyone who faces challenging problems and cannot solve them by using traditional approaches, everyone who wants to understand the tremendous achievements in computer intelligence. The book will help to develop a practical understanding of what intelligent systems can and cannot do, discover which tools are most relevant for your task and, finally, how to use these tools.

The book consists of nine chapters.

In Chapter 1, we briefly discuss the history of artificial intelligence from the era of great ideas and great expectations in the 1960s to the disillusionment and funding cutbacks in the early 1970s; from the development of the first expert systems such as DENDRAL, MYCIN and PROSPECTOR in the seventies to the maturity of expert system technology and its massive applications in different areas in the 1980s and 1990s; from a simple binary model of neurons proposed in the 1940s to a dramatic resurgence of the field of artificial neural networks in the 1980s; from the introduction of fuzzy set theory and its being ignored by the West in the 1960s to numerous 'fuzzy' consumer products offered by the Japanese in the 1980s and world-wide acceptance of 'soft' computing and computing with words in the 1990s.

In Chapter 2, we present an overview of rule-based expert systems. We briefly discuss what knowledge is, and how experts express their knowledge in the form of production rules. We identify the main players in the expert system development team and show the structure of a rule-based system. We discuss fundamental characteristics of expert systems and note that expert systems can make mistakes. Then we review the forward and backward chaining inference techniques and debate conflict resolution strategies. Finally, the advantages and disadvantages of rule-based expert systems are examined.

In Chapter 3, we present two uncertainty management techniques used in expert systems: Bayesian reasoning and certainty factors. We identify the main sources of uncertain knowledge and briefly review probability theory. We consider the Bayesian method of accumulating evidence and develop a simple expert system based on the Bayesian approach. Then we examine the certainty factors theory (a popular alternative to Bayesian reasoning) and develop an expert system based on evidential reasoning. Finally, we compare Bayesian reasoning and certainty factors, and determine appropriate areas for their applications.

In Chapter 4, we introduce fuzzy logic and discuss the philosophical ideas behind it. We present the concept of fuzzy sets, consider how to represent a fuzzy set in a computer, and examine operations of fuzzy sets. We also define linguistic variables and hedges. Then we present fuzzy rules and explain the main differences between classical and fuzzy rules. We explore two fuzzy inference techniques – Mamdani and Sugeno – and suggest appropriate areas for their application. Finally, we introduce the main steps in developing a fuzzy expert system, and illustrate the theory through the actual process of building and tuning a fuzzy system.

In Chapter 5, we present an overview of frame-based expert systems. We consider the concept of a frame and discuss how to use frames for knowledge representation. We find that inheritance is an essential feature of frame based systems. We examine the application of methods, demons and rules. Finally, we consider the development of a frame-based expert system through an example.

In Chapter 6, we introduce artificial neural networks and discuss the basic ideas behind machine learning. We present the concept of a perceptron as a simple computing element and consider the perceptron learning rule. We explore multilayer neural networks and discuss how to improve the computational efficiency of the back-propagation learning algorithm. Then we introduce recurrent neural networks, consider the Hopfield network training algorithm and bidirectional associative memory (BAM). Finally, we present self-organising neural networks and explore Hebbian and competitive learning.

In Chapter 7, we present an overview of evolutionary computation. We consider genetic algorithms, evolution strategies and genetic programming. We introduce the main steps in developing a genetic algorithm, discuss why genetic algorithms work, and illustrate the theory through actual applications of genetic algorithms. Then we present a basic concept of evolutionary strategies and determine the differences between evolutionary strategies and genetic algorithms. Finally, we consider genetic programming and its application to real problems.

In Chapter 8, we consider hybrid intelligent systems as a combination of different intelligent technologies. First we introduce a new breed of expert systems, called neural expert systems, which combine neural networks and rule-based expert systems. Then we consider a neuro-fuzzy system that is functionally equivalent to the Mamdani fuzzy inference model, and an adaptive neuro-fuzzy inference system (ANFIS), equivalent to the Sugeno fuzzy inference model. Finally, we discuss evolutionary neural networks and fuzzy evolutionary systems.

In Chapter 9, we consider knowledge engineering and data mining. First we discuss what kind of problems can be addressed with intelligent systems and introduce six main phases of the knowledge engineering process. Then we study

typical applications of intelligent systems, including diagnosis, classification, decision support, pattern recognition and prediction. Finally, we examine an application of decision trees in data mining.

The book also has an appendix and a glossary. The appendix provides a list of commercially available AI tools. The glossary contains definitions of over 250 terms used in expert systems, fuzzy logic, neural networks, evolutionary computation, knowledge engineering and data mining.

I hope that the reader will share my excitement on the subject of artificial intelligence and soft computing and will find this book useful.

The website can be accessed at: http://www.booksites.net/negnevitsky

Michael Negnevitsky
Hobart, Tasmania, Australia
February 2001

Preface to the second edition

The main objective of the book remains the same as in the first edition – to provide the reader with practical understanding of the field of computer intelligence. It is intended as an introductory text suitable for a one-semester course, and assumes the students have no programming experience.

In terms of the coverage, in this edition we demonstrate several new applications of intelligent tools for solving specific problems. The changes are in the following chapters:

- In Chapter 2, we introduce a new demonstration rule-based expert system, MEDIA ADVISOR.
- In Chapter 9, we add a new case study on classification neural networks with competitive learning.
- In Chapter 9, we introduce a section 'Will genetic algorithms work for my problem?'. The section includes a case study with the travelling salesman problem.
- Also in Chapter 9, we add a new section 'Will a hybrid intelligent system work for my problem?'. This section includes two case studies: the first covers a neuro-fuzzy decision-support system with a heterogeneous structure, and the second explores an adaptive neuro-fuzzy inference system (ANFIS) with a homogeneous structure.

Finally, we have expanded the book's references and bibliographies, and updated the list of AI tools and vendors in the appendix.

Michael Negnevitsky
Hobart, Tasmania, Australia
January 2004

Acknowledgements

Introduction to knowledge-based intelligent systems

<div style="text-align: right">**1**</div>

In which we consider what it means to be intelligent and whether machines could be such a thing.

1.1 Intelligent machines, or what machines can do

Philosophers have been trying for over two thousand years to understand and resolve two big questions of the universe: how does a human mind work, and can non-humans have minds? However, these questions are still unanswered.

Some philosophers have picked up the computational approach originated by computer scientists and accepted the idea that machines can do everything that humans can do. Others have openly opposed this idea, claiming that such highly sophisticated behaviour as love, creative discovery and moral choice will always be beyond the scope of any machine.

The nature of philosophy allows for disagreements to remain unresolved. In fact, engineers and scientists have already built machines that we can call 'intelligent'. So what does the word 'intelligence' mean? Let us look at a dictionary definition.

1 Someone's **intelligence** is their ability to understand and learn things.
2 **Intelligence** is the ability to think and understand instead of doing things by instinct or automatically.

<div style="text-align: right">(*Essential English Dictionary*, Collins, London, 1990)</div>

Thus, according to the first definition, intelligence is the quality possessed by humans. But the second definition suggests a completely different approach and gives some flexibility; it does not specify whether it is some**one** or some**thing** that has the ability to think and understand. Now we should discover what thinking means. Let us consult our dictionary again.

Thinking is the activity of using your brain to consider a problem or to create an idea.

<div style="text-align: right">(*Essential English Dictionary*, Collins, London, 1990)</div>

So, in order to think, some**one** or some**thing** has to have a brain, or in other words, an organ that enables some**one** or some**thing** to learn and understand things, to solve problems and to make decisions. So we can define intelligence as 'the ability to learn and understand, to solve problems and to make decisions'.

The very question that asks whether computers can be intelligent, or whether machines can think, came to us from the 'dark ages' of artificial intelligence (from the late 1940s). The goal of **artificial intelligence** (AI) as a science is to make machines do things that would require intelligence if done by humans (Boden, 1977). Therefore, the answer to the question 'Can machines think?' was vitally important to the discipline. However, the answer is not a simple 'Yes' or 'No', but rather a vague or **fuzzy** one. Your everyday experience and common sense would have told you that. Some people are smarter in some ways than others. Sometimes we make very intelligent decisions but sometimes we also make very silly mistakes. Some of us deal with complex mathematical and engineering problems but are moronic in philosophy and history. Some people are good at making money, while others are better at spending it. As humans, we all have the ability to learn and understand, to solve problems and to make decisions; however, our abilities are not equal and lie in different areas. Therefore, we should expect that if machines can think, some of them might be smarter than others in some ways.

One of the earliest and most significant papers on machine intelligence, 'Computing machinery and intelligence', was written by the British mathematician Alan Turing over fifty years ago (Turing, 1950). However, it has stood up well to the test of time, and Turing's approach remains universal.

Alan Turing began his scientific career in the early 1930s by rediscovering the Central Limit Theorem. In 1937 he wrote a paper on computable numbers, in which he proposed the concept of a universal machine. Later, during the Second World War, he was a key player in deciphering Enigma, the German military encoding machine. After the war, Turing designed the 'Automatic Computing Engine'. He also wrote the first program capable of playing a complete chess game; it was later implemented on the Manchester University computer. Turing's theoretical concept of the universal computer and his practical experience in building code-breaking systems equipped him to approach the key fundamental question of artificial intelligence. He asked: Is there thought without experience? Is there mind without communication? Is there language without living? Is there intelligence without life? All these questions, as you can see, are just variations on the fundamental question of artificial intelligence, Can machines think?

Turing did not provide definitions of machines and thinking, he just avoided semantic arguments by inventing a game, the **Turing imitation game**. Instead of asking, 'Can machines think?', Turing said we should ask, 'Can machines pass a behaviour test for intelligence?' He predicted that by the year 2000, a computer could be programmed to have a conversation with a human interrogator for five minutes and would have a 30 per cent chance of deceiving the interrogator that it was a human. Turing defined the intelligent behaviour of a computer as the ability to achieve the human-level performance in cognitive tasks. In other
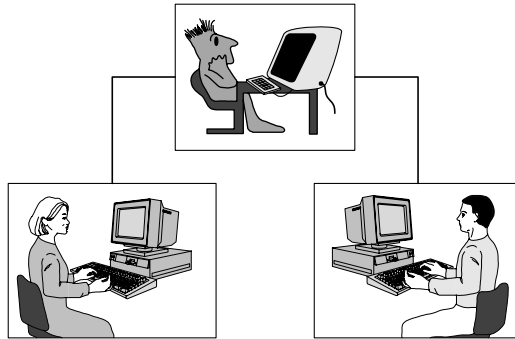
Figure 1.1   Turing imitation game: phase 1

words, a computer passes the test if interrogators cannot distinguish the machine from a human on the basis of the answers to their questions.

The imitation game proposed by Turing originally included two phases. In the first phase, shown in Figure 1.1, the interrogator, a man and a woman are each placed in separate rooms and can communicate only via a neutral medium such as a remote terminal. The interrogator's objective is to work out who is the man and who is the woman by questioning them. The rules of the game are that the man should attempt to deceive the interrogator that **he** is the woman, while the woman has to convince the interrogator that **she** is the woman.

In the second phase of the game, shown in Figure 1.2, the man is replaced by a computer programmed to deceive the interrogator as the man did. It would even be programmed to make mistakes and provide fuzzy answers in the way a human would. If the computer can fool the interrogator as often as the man did, we may say this computer has passed the intelligent behaviour test.

Physical simulation of a human is not important for intelligence. Hence, in the Turing test the interrogator does not see, touch or hear the computer and is therefore not influenced by its appearance or voice. However, the interrogator is allowed to ask any questions, even provocative ones, in order to identify the machine. The interrogator may, for example, ask both the human and the
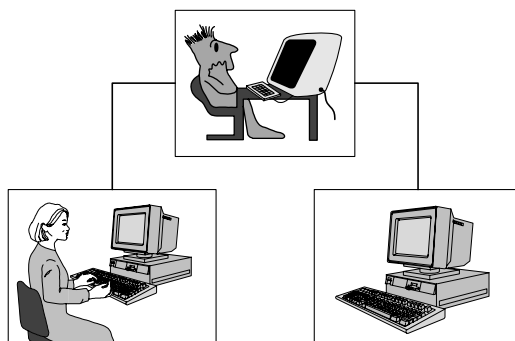
Figure 1.2   Turing imitation game: phase 2

machine to perform complex mathematical calculations, expecting that the computer will provide a correct solution and will do it faster than the human. Thus, the computer will need to know when to make a mistake and when to delay its answer. The interrogator also may attempt to discover the emotional nature of the human, and thus, he might ask both subjects to examine a short novel or poem or even painting. Obviously, the computer will be required here to simulate a human's emotional understanding of the work.

The Turing test has two remarkable qualities that make it really universal.

- By maintaining communication between the human and the machine via terminals, the test gives us an objective standard view on intelligence. It avoids debates over the human nature of intelligence and eliminates any bias in favour of humans.

- The test itself is quite independent from the details of the experiment. It can be conducted either as a two-phase game as just described, or even as a single-phase game in which the interrogator needs to choose between the human and the machine from the beginning of the test. The interrogator is also free to ask any question in any field and can concentrate solely on the content of the answers provided.

Turing believed that by the end of the 20th century it would be possible to program a digital computer to play the imitation game. Although modern computers still cannot pass the Turing test, it provides a basis for the verification and validation of knowledge-based systems. A program thought intelligent in some narrow area of expertise is evaluated by comparing its performance with the performance of a human expert.

Our brain stores the equivalent of over 10^{18} bits and can process information at the equivalent of about 10^{15} bits per second. By 2020, the brain will probably be modelled by a chip the size of a sugar cube – and perhaps by then there will be a computer that can play – even win – the Turing imitation game. However, do we really want the machine to perform mathematical calculations as slowly and inaccurately as humans do? From a practical point of view, an intelligent machine should help humans to make decisions, to search for information, to control complex objects, and finally to understand the meaning of words. There is probably no point in trying to achieve the abstract and elusive goal of developing machines with human-like intelligence. To build an intelligent computer system, we have to capture, organise and use human expert knowledge in some narrow area of expertise.

1.2 The history of artificial intelligence, or from the 'Dark Ages' to knowledge-based systems

Artificial intelligence as a science was founded by three generations of researchers. Some of the most important events and contributors from each generation are described next.

1.2.1   The 'Dark Ages', or the birth of artificial intelligence (1943–56)

The first work recognised in the field of artificial intelligence (AI) was presented by Warren McCulloch and Walter Pitts in 1943. McCulloch had degrees in philosophy and medicine from Columbia University and became the Director of the Basic Research Laboratory in the Department of Psychiatry at the University of Illinois. His research on the central nervous system resulted in the first major contribution to AI: a model of neurons of the brain.

McCulloch and his co-author Walter Pitts, a young mathematician, proposed a model of artificial neural networks in which each neuron was postulated as being in binary state, that is, in either **on** or **off** condition (McCulloch and Pitts, 1943). They demonstrated that their neural network model was, in fact, equivalent to the Turing machine, and proved that any computable function could be computed by some network of connected neurons. McCulloch and Pitts also showed that simple network structures could learn.

The neural network model stimulated both theoretical and experimental work to model the brain in the laboratory. However, experiments clearly demonstrated that the binary model of neurons was not correct. In fact, a neuron has highly non-linear characteristics and cannot be considered as a simple two-state device. Nonetheless, McCulloch, the second 'founding father' of AI after Alan Turing, had created the cornerstone of neural computing and artificial neural networks (ANN). After a decline in the 1970s, the field of ANN was revived in the late 1980s.

The third founder of AI was John von Neumann, the brilliant Hungarian-born mathematician. In 1930, he joined the Princeton University, lecturing in mathematical physics. He was a colleague and friend of Alan Turing. During the Second World War, von Neumann played a key role in the Manhattan Project that built the nuclear bomb. He also became an adviser for the Electronic Numerical Integrator and Calculator (ENIAC) project at the University of Pennsylvania and helped to design the Electronic Discrete Variable Automatic Computer (EDVAC), a **stored program** machine. He was influenced by McCulloch and Pitts's neural network model. When Marvin Minsky and Dean Edmonds, two graduate students in the Princeton mathematics department, built the first neural network computer in 1951, von Neumann encouraged and supported them.

Another of the first-generation researchers was Claude Shannon. He graduated from Massachusetts Institute of Technology (MIT) and joined Bell Telephone Laboratories in 1941. Shannon shared Alan Turing's ideas on the possibility of machine intelligence. In 1950, he published a paper on chess-playing machines, which pointed out that a typical chess game involved about 10^{120} possible moves (Shannon, 1950). Even if the new von Neumann-type computer could examine one move per microsecond, it would take 3×10^{106} years to make its first move. Thus Shannon demonstrated the need to use heuristics in the search for the solution.

Princeton University was also home to John McCarthy, another founder of AI. He convinced Martin Minsky and Claude Shannon to organise a summer

workshop at Dartmouth College, where McCarthy worked after graduating from Princeton. In 1956, they brought together researchers interested in the study of machine intelligence, artificial neural nets and automata theory. The workshop was sponsored by IBM. Although there were just ten researchers, this workshop gave birth to a new science called **artificial intelligence**. For the next twenty years the field of AI would be dominated by the participants at the Dartmouth workshop and their students.

1.2.2 The rise of artificial intelligence, or the era of great expectations (1956–late 1960s)

The early years of AI are characterised by tremendous enthusiasm, great ideas and very limited success. Only a few years before, computers had been introduced to perform routine mathematical calculations, but now AI researchers were demonstrating that computers could do more than that. It was an era of great expectations.

John McCarthy, one of the organisers of the Dartmouth workshop and the inventor of the term 'artificial intelligence', moved from Dartmouth to MIT. He defined the high-level language **LISP** – one of the oldest programming languages (FORTRAN is just two years older), which is still in current use. In 1958, McCarthy presented a paper, 'Programs with Common Sense', in which he proposed a program called the **Advice Taker** to search for solutions to general problems of the world (McCarthy, 1958). McCarthy demonstrated how his program could generate, for example, a plan to drive to the airport, based on some simple axioms. Most importantly, the program was designed to accept new axioms, or in other words new knowledge, in different areas of expertise without being reprogrammed. Thus the Advice Taker was the first complete knowledge-based system incorporating the central principles of knowledge representation and reasoning.

Another organiser of the Dartmouth workshop, Marvin Minsky, also moved to MIT. However, unlike McCarthy with his focus on formal logic, Minsky developed an anti-logical outlook on knowledge representation and reasoning. His theory of frames (Minsky, 1975) was a major contribution to knowledge engineering.

The early work on neural computing and artificial neural networks started by McCulloch and Pitts was continued. Learning methods were improved and Frank Rosenblatt proved the **perceptron convergence theorem**, demonstrating that his learning algorithm could adjust the connection strengths of a perceptron (Rosenblatt, 1962).

One of the most ambitious projects of the era of great expectations was the General Problem Solver (GPS) (Newell and Simon, 1961, 1972). Allen Newell and Herbert Simon from the Carnegie Mellon University developed a general-purpose program to simulate human problem-solving methods. GPS was probably the first attempt to separate the problem-solving technique from the data. It was based on the technique now referred to as **means-ends analysis**.

Newell and Simon postulated that a problem to be solved could be defined in terms of **states**. The means-ends analysis was used to determine a difference between the current state and the desirable state or the **goal state** of the problem, and to choose and apply **operators** to reach the goal state. If the goal state could not be immediately reached from the current state, a new state closer to the goal would be established and the procedure repeated until the goal state was reached. The set of operators determined the solution plan.

However, GPS failed to solve complicated problems. The program was based on formal logic and therefore could generate an infinite number of possible operators, which is inherently inefficient. The amount of computer time and memory that GPS required to solve real-world problems led to the project being abandoned.

In summary, we can say that in the 1960s, AI researchers attempted to simulate the complex thinking process by inventing **general methods** for solving broad classes of problems. They used the general-purpose search mechanism to find a solution to the problem. Such approaches, now referred to as **weak methods**, applied weak information about the problem domain; this resulted in weak performance of the programs developed.

However, it was also a time when the field of AI attracted great scientists who introduced fundamental new ideas in such areas as knowledge representation, learning algorithms, neural computing and computing with words. These ideas could not be implemented then because of the limited capabilities of computers, but two decades later they have led to the development of real-life practical applications.

It is interesting to note that Lotfi Zadeh, a professor from the University of California at Berkeley, published his famous paper 'Fuzzy sets' also in the 1960s (Zadeh, 1965). This paper is now considered the foundation of the fuzzy set theory. Two decades later, fuzzy researchers have built hundreds of smart machines and intelligent systems.

By 1970, the euphoria about AI was gone, and most government funding for AI projects was cancelled. AI was still a relatively new field, academic in nature, with few practical applications apart from playing games (Samuel, 1959, 1967; Greenblatt *et al.*, 1967). So, to the outsider, the achievements would be seen as toys, as no AI system at that time could manage real-world problems.

1.2.3   Unfulfilled promises, or the impact of reality (late 1960s–early 1970s)

From the mid-1950s, AI researchers were making promises to build all-purpose intelligent machines on a human-scale knowledge base by the 1980s, and to exceed human intelligence by the year 2000. By 1970, however, they realised that such claims were too optimistic. Although a few AI programs could demonstrate some level of machine intelligence in one or two toy problems, almost no AI projects could deal with a wider selection of tasks or more difficult real-world problems.

The main difficulties for AI in the late 1960s were:

- Because AI researchers were developing general methods for broad classes of problems, early programs contained little or even no knowledge about a problem domain. To solve problems, programs applied a search strategy by trying out different combinations of small steps, until the right one was found. This method worked for 'toy' problems, so it seemed reasonable that, if the programs could be 'scaled up' to solve large problems, they would finally succeed. However, this approach was wrong.

  Easy, or **tractable**, problems can be solved in polynomial time, i.e. for a problem of size n, the time or number of steps needed to find the solution is a polynomial function of n. On the other hand, hard or intractable problems require times that are exponential functions of the problem size. While a polynomial-time algorithm is considered to be efficient, an exponential-time algorithm is inefficient, because its execution time increases rapidly with the problem size. The theory of NP-completeness (Cook, 1971; Karp, 1972), developed in the early 1970s, showed the existence of a large class of non-deterministic polynomial problems (NP problems) that are NP-complete. A problem is called NP if its solution (if one exists) can be guessed and verified in polynomial time; non-deterministic means that no particular algorithm is followed to make the guess. The hardest problems in this class are NP-complete. Even with faster computers and larger memories, these problems are hard to solve.

- Many of the problems that AI attempted to solve were too broad and too difficult. A typical task for early AI was machine translation. For example, the National Research Council, USA, funded the translation of Russian scientific papers after the launch of the first artificial satellite (Sputnik) in 1957. Initially, the project team tried simply replacing Russian words with English, using an electronic dictionary. However, it was soon found that translation requires a general understanding of the subject to choose the correct words. This task was too difficult. In 1966, all translation projects funded by the US government were cancelled.

- In 1971, the British government also suspended support for AI research. Sir James Lighthill had been commissioned by the Science Research Council of Great Britain to review the current state of AI (Lighthill, 1973). He did not find any major or even significant results from AI research, and therefore saw no need to have a separate science called 'artificial intelligence'.

1.2.4 The technology of expert systems, or the key to success (early 1970s–mid-1980s)

Probably the most important development in the 1970s was the realisation that the problem domain for intelligent machines had to be sufficiently restricted. Previously, AI researchers had believed that clever search algorithms and reasoning techniques could be invented to emulate general, human-like, problem-solving methods. A general-purpose search mechanism could rely on

elementary reasoning steps to find complete solutions and could use weak knowledge about domain. However, when weak methods failed, researchers finally realised that the only way to deliver practical results was to solve typical cases in narrow areas of expertise by making large reasoning steps.

The DENDRAL program is a typical example of the emerging technology (Buchanan *et al.*, 1969). DENDRAL was developed at Stanford University to analyse chemicals. The project was supported by NASA, because an un-manned spacecraft was to be launched to Mars and a program was required to determine the molecular structure of Martian soil, based on the mass spectral data provided by a mass spectrometer. Edward Feigenbaum (a former student of Herbert Simon), Bruce Buchanan (a computer scientist) and Joshua Lederberg (a Nobel prize winner in genetics) formed a team to solve this challenging problem.

The traditional method of solving such problems relies on a generate-and-test technique: all possible molecular structures consistent with the mass spectrogram are generated first, and then the mass spectrum is determined or predicted for each structure and tested against the actual spectrum. However, this method failed because millions of possible structures could be generated – the problem rapidly became intractable even for decent-sized molecules.

To add to the difficulties of the challenge, there was no scientific algorithm for mapping the mass spectrum into its molecular structure. However, analytical chemists, such as Lederberg, could solve this problem by using their skills, experience and expertise. They could enormously reduce the number of possible structures by looking for well-known patterns of peaks in the spectrum, and thus provide just a few feasible solutions for further examination. Therefore, Feigenbaum's job became to incorporate the expertise of Lederberg into a computer program to make it perform at a human expert level. Such programs were later called **expert systems**. To understand and adopt Lederberg's knowl-edge and operate with his terminology, Feigenbaum had to learn basic ideas in chemistry and spectral analysis. However, it became apparent that Feigenbaum used not only rules of chemistry but also his own heuristics, or rules-of-thumb, based on his experience, and even guesswork. Soon Feigenbaum identified one of the major difficulties in the project, which he called the 'knowledge acquisi-tion bottleneck' – how to extract knowledge from human experts to apply to computers. To articulate his knowledge, Lederberg even needed to study basics in computing.

Working as a team, Feigenbaum, Buchanan and Lederberg developed DENDRAL, the first successful knowledge-based system. The key to their success was mapping all the relevant theoretical knowledge from its general form to highly specific rules ('cookbook recipes') (Feigenbaum *et al.*, 1971).

The significance of DENDRAL can be summarised as follows:

- DENDRAL marked a major 'paradigm shift' in AI: a shift from general-purpose, knowledge-sparse, weak methods to domain-specific, knowledge-intensive techniques.

- The aim of the project was to develop a computer program to attain the level of performance of an experienced human chemist. Using heuristics in the form of high-quality specific rules – rules-of-thumb – elicited from human experts, the DENDRAL team proved that computers could equal an expert in narrow, defined, problem areas.

- The DENDRAL project originated the fundamental idea of the new methodology of expert systems – **knowledge engineering**, which encompassed techniques of capturing, analysing and expressing in rules an expert's 'know-how'.

DENDRAL proved to be a useful analytical tool for chemists and was marketed commercially in the United States.

The next major project undertaken by Feigenbaum and others at Stanford University was in the area of medical diagnosis. The project, called MYCIN, started in 1972. It later became the Ph.D. thesis of Edward Shortliffe (Shortliffe, 1976). MYCIN was a rule-based expert system for the diagnosis of infectious blood diseases. It also provided a doctor with therapeutic advice in a convenient, user-friendly manner.

MYCIN had a number of characteristics common to early expert systems, including:

- MYCIN could perform at a level equivalent to human experts in the field and considerably better than junior doctors.

- MYCIN's knowledge consisted of about 450 independent rules of IF-THEN form derived from human knowledge in a narrow domain through extensive interviewing of experts.

- The knowledge incorporated in the form of rules was clearly separated from the reasoning mechanism. The system developer could easily manipulate knowledge in the system by inserting or deleting some rules. For example, a domain-independent version of MYCIN called EMYCIN (Empty MYCIN) was later produced at Stanford University (van Melle, 1979; van Melle *et al.*, 1981). It had all the features of the MYCIN system except the knowledge of infectious blood diseases. EMYCIN facilitated the development of a variety of diagnostic applications. System developers just had to add new knowledge in the form of rules to obtain a new application.

MYCIN also introduced a few new features. Rules incorporated in MYCIN reflected the uncertainty associated with knowledge, in this case with medical diagnosis. It tested rule conditions (the IF part) against available data or data requested from the physician. When appropriate, MYCIN inferred the truth of a condition through a calculus of uncertainty called **certainty factors**. Reasoning in the face of uncertainty was the most important part of the system.

Another probabilistic system that generated enormous publicity was PROSPECTOR, an expert system for mineral exploration developed by the Stanford Research Institute (Duda *et al.*, 1979). The project ran from 1974 to

1983. Nine experts contributed their knowledge and expertise. To represent their knowledge, PROSPECTOR used a combined structure that incorporated rules and a semantic network. PROSPECTOR had over a thousand rules to represent extensive domain knowledge. It also had a sophisticated support package including a knowledge acquisition system.

PROSPECTOR operates as follows. The user, an exploration geologist, is asked to input the characteristics of a suspected deposit: the geological setting, structures, kinds of rocks and minerals. Then the program compares these characteristics with models of ore deposits and, if necessary, queries the user to obtain additional information. Finally, PROSPECTOR makes an assessment of the suspected mineral deposit and presents its conclusion. It can also explain the steps it used to reach the conclusion.

In exploration geology, important decisions are usually made in the face of uncertainty, with knowledge that is incomplete or fuzzy. To deal with such knowledge, PROSPECTOR incorporated Bayes's rules of evidence to propagate uncertainties through the system. PROSPECTOR performed at the level of an expert geologist and proved itself in practice. In 1980, it identified a molybdenum deposit near Mount Tolman in Washington State. Subsequent drilling by a mining company confirmed the deposit was worth over $100 million. You couldn't hope for a better justification for using expert systems.

The expert systems mentioned above have now become classics. A growing number of successful applications of expert systems in the late 1970s showed that AI technology could move successfully from the research laboratory to the commercial environment. During this period, however, most expert systems were developed with special AI languages, such as LISP, PROLOG and OPS, based on powerful workstations. The need to have rather expensive hardware and complicated programming languages meant that the challenge of expert system development was left in the hands of a few research groups at Stanford University, MIT, Stanford Research Institute and Carnegie-Mellon University. Only in the 1980s, with the arrival of personal computers (PCs) and easy-to-use expert system development tools – shells – could ordinary researchers and engineers in all disciplines take up the opportunity to develop expert systems.

A 1986 survey reported a remarkable number of successful expert system applications in different areas: chemistry, electronics, engineering, geology, management, medicine, process control and military science (Waterman, 1986). Although Waterman found nearly 200 expert systems, most of the applications were in the field of medical diagnosis. Seven years later a similar survey reported over 2500 developed expert systems (Durkin, 1994). The new growing area was business and manufacturing, which accounted for about 60 per cent of the applications. Expert system technology had clearly matured.

Are expert systems really the key to success in any field? In spite of a great number of successful developments and implementations of expert systems in different areas of human knowledge, it would be a mistake to overestimate the capability of this technology. The difficulties are rather complex and lie in both technical and sociological spheres. They include the following:

- Expert systems are restricted to a very narrow domain of expertise. For example, MYCIN, which was developed for the diagnosis of infectious blood diseases, lacks any real knowledge of human physiology. If a patient has more than one disease, we cannot rely on MYCIN. In fact, therapy prescribed for the blood disease might even be harmful because of the other disease.

- Because of the narrow domain, expert systems are not as robust and flexible as a user might want. Furthermore, expert systems can have difficulty recognising domain boundaries. When given a task different from the typical problems, an expert system might attempt to solve it and fail in rather unpredictable ways.

- Expert systems have limited explanation capabilities. They can show the sequence of the rules they applied to reach a solution, but cannot relate accumulated, heuristic knowledge to any deeper understanding of the problem domain.

- Expert systems are also difficult to verify and validate. No general technique has yet been developed for analysing their completeness and consistency. Heuristic rules represent knowledge in abstract form and lack even basic understanding of the domain area. It makes the task of identifying incorrect, incomplete or inconsistent knowledge very difficult.

- Expert systems, especially the first generation, have little or no ability to learn from their experience. Expert systems are built individually and cannot be developed fast. It might take from five to ten person-years to build an expert system to solve a moderately difficult problem (Waterman, 1986). Complex systems such as DENDRAL, MYCIN or PROSPECTOR can take over 30 person-years to build. This large effort, however, would be difficult to justify if improvements to the expert system's performance depended on further attention from its developers.

Despite all these difficulties, expert systems have made the breakthrough and proved their value in a number of important applications.

1.2.5 How to make a machine learn, or the rebirth of neural networks (mid-1980s–onwards)

In the mid-1980s, researchers, engineers and experts found that building an expert system required much more than just buying a reasoning system or expert system shell and putting enough rules in it. Disillusion about the applicability of expert system technology even led to people predicting an AI 'winter' with severely squeezed funding for AI projects. AI researchers decided to have a new look at neural networks.

By the late 1960s, most of the basic ideas and concepts necessary for neural computing had already been formulated (Cowan, 1990). However, only in the mid-1980s did the solution emerge. The major reason for the delay was technological: there were no PCs or powerful workstations to model and

experiment with artificial neural networks. The other reasons were psychological and financial. For example, in 1969, Minsky and Papert had mathematically demonstrated the fundamental computational limitations of one-layer perceptrons (Minsky and Papert, 1969). They also said there was no reason to expect that more complex multilayer perceptrons would represent much. This certainly would not encourage anyone to work on perceptrons, and as a result, most AI researchers deserted the field of artificial neural networks in the 1970s.

In the 1980s, because of the need for brain-like information processing, as well as the advances in computer technology and progress in neuroscience, the field of neural networks experienced a dramatic resurgence. Major contributions to both theory and design were made on several fronts. Grossberg established a new principle of self-organisation (**adaptive resonance theory**), which provided the basis for a new class of neural networks (Grossberg, 1980). Hopfield introduced neural networks with feedback – **Hopfield networks**, which attracted much attention in the 1980s (Hopfield, 1982). Kohonen published a paper on **self-organised maps** (Kohonen, 1982). Barto, Sutton and Anderson published their work on **reinforcement learning** and its application in control (Barto *et al.*, 1983). But the real breakthrough came in 1986 when the **back-propagation learning algorithm**, first introduced by Bryson and Ho in 1969 (Bryson and Ho, 1969), was reinvented by Rumelhart and McClelland in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition* (Rumelhart and McClelland, 1986). At the same time, back-propagation learning was also discovered by Parker (Parker, 1987) and LeCun (LeCun, 1988), and since then has become the most popular technique for training multilayer perceptrons. In 1988, Broomhead and Lowe found a procedure to design **layered feedforward networks** using radial basis functions, an alternative to multilayer perceptrons (Broomhead and Lowe, 1988).

Artificial neural networks have come a long way from the early models of McCulloch and Pitts to an interdisciplinary subject with roots in neuroscience, psychology, mathematics and engineering, and will continue to develop in both theory and practical applications. However, Hopfield's paper (Hopfield, 1982) and Rumelhart and McClelland's book (Rumelhart and McClelland, 1986) were the most significant and influential works responsible for the rebirth of neural networks in the 1980s.

1.2.6 Evolutionary computation, or learning by doing (early 1970s–onwards)

Natural intelligence is a product of evolution. Therefore, by simulating biological evolution, we might expect to discover how living systems are propelled towards high-level intelligence. Nature learns by doing; biological systems are not told how to adapt to a specific environment – they simply compete for survival. The fittest species have a greater chance to reproduce, and thereby to pass their genetic material to the next generation.

The evolutionary approach to artificial intelligence is based on the computational models of natural selection and genetics. Evolutionary computation works by simulating a population of individuals, evaluating their performance, generating a new population, and repeating this process a number of times.

Evolutionary computation combines three main techniques: genetic algorithms, evolutionary strategies, and genetic programming.

The concept of genetic algorithms was introduced by John Holland in the early 1970s (Holland, 1975). He developed an algorithm for manipulating artificial 'chromosomes' (strings of binary digits), using such genetic operations as selection, crossover and mutation. Genetic algorithms are based on a solid theoretical foundation of the Schema Theorem (Holland, 1975; Goldberg, 1989).

In the early 1960s, independently of Holland's genetic algorithms, Ingo Rechenberg and Hans-Paul Schwefel, students of the Technical University of Berlin, proposed a new optimisation method called evolutionary strategies (Rechenberg, 1965). Evolutionary strategies were designed specifically for solving parameter optimisation problems in engineering. Rechenberg and Schwefel suggested using random changes in the parameters, as happens in natural mutation. In fact, an evolutionary strategies approach can be considered as an alternative to the engineer's intuition. Evolutionary strategies use a numerical optimisation procedure, similar to a focused Monte Carlo search.

Both genetic algorithms and evolutionary strategies can solve a wide range of problems. They provide robust and reliable solutions for highly complex, non-linear search and optimisation problems that previously could not be solved at all (Holland, 1995; Schwefel, 1995).

Genetic programming represents an application of the genetic model of learning to programming. Its goal is to evolve not a coded representation of some problem, but rather a computer code that solves the problem. That is, genetic programming generates computer programs as the solution.

The interest in genetic programming was greatly stimulated by John Koza in the 1990s (Koza, 1992, 1994). He used genetic operations to manipulate symbolic code representing LISP programs. Genetic programming offers a solution to the main challenge of computer science – making computers solve problems without being explicitly programmed.

Genetic algorithms, evolutionary strategies and genetic programming represent rapidly growing areas of AI, and have great potential.

1.2.7 The new era of knowledge engineering, or computing with words (late 1980s–onwards)

Neural network technology offers more natural interaction with the real world than do systems based on symbolic reasoning. Neural networks can learn, adapt to changes in a problem's environment, establish patterns in situations where rules are not known, and deal with fuzzy or incomplete information. However, they lack explanation facilities and usually act as a black box. The process of training neural networks with current technologies is slow, and frequent retraining can cause serious difficulties.

Although in some special cases, particularly in knowledge-poor situations, ANNs can solve problems better than expert systems, the two technologies are not in competition now. They rather nicely complement each other.

Classic expert systems are especially good for closed-system applications with precise inputs and logical outputs. They use expert knowledge in the form of rules and, if required, can interact with the user to establish a particular fact. A major drawback is that human experts cannot always express their knowledge in terms of rules or explain the line of their reasoning. This can prevent the expert system from accumulating the necessary knowledge, and consequently lead to its failure. To overcome this limitation, neural computing can be used for extracting hidden knowledge in large data sets to obtain rules for expert systems (Medsker and Leibowitz, 1994; Zahedi, 1993). ANNs can also be used for correcting rules in traditional rule-based expert systems (Omlin and Giles, 1996). In other words, where acquired knowledge is incomplete, neural networks can refine the knowledge, and where the knowledge is inconsistent with some given data, neural networks can revise the rules.

Another very important technology dealing with vague, imprecise and uncertain knowledge and data is **fuzzy logic**. Most methods of handling imprecision in classic expert systems are based on the probability concept. MYCIN, for example, introduced certainty factors, while PROSPECTOR incorporated Bayes' rules to propagate uncertainties. However, experts do not usually think in probability values, but in such terms as **often**, **generally**, **sometimes**, **occasionally** and **rarely**. Fuzzy logic is concerned with the use of fuzzy values that capture the meaning of words, human reasoning and decision making. As a method to encode and apply human knowledge in a form that accurately reflects an expert's understanding of difficult, complex problems, fuzzy logic provides the way to break through the computational bottlenecks of traditional expert systems.

At the heart of fuzzy logic lies the concept of a linguistic variable. The values of the linguistic variable are words rather than numbers. Similar to expert systems, fuzzy systems use IF-THEN rules to incorporate human knowledge, but these rules are fuzzy, such as:

IF speed is high THEN stopping_distance is long

IF speed is low THEN stopping_distance is short.

Fuzzy logic or **fuzzy set theory** was introduced by Professor Lotfi Zadeh, Berkeley's electrical engineering department chairman, in 1965 (Zadeh, 1965). It provided a means of computing with words. However, acceptance of fuzzy set theory by the technical community was slow and difficult. Part of the problem was the provocative name – 'fuzzy' – which seemed too light-hearted to be taken seriously. Eventually, fuzzy theory, ignored in the West, was taken seriously in the East – by the Japanese. It has been used successfully since 1987 in Japanese-designed dishwashers, washing machines, air conditioners, television sets, copiers and even cars.

The introduction of fuzzy products gave rise to tremendous interest in this apparently 'new' technology first proposed over 30 years ago. Hundreds of books and thousands of technical papers have been written on this topic. Some of the classics are: *Fuzzy Sets, Neural Networks and Soft Computing* (Yager and Zadeh, eds, 1994); *The Fuzzy Systems Handbook* (Cox, 1999); *Fuzzy Engineering* (Kosko, 1997); *Expert Systems and Fuzzy Systems* (Negoita, 1985); and also the best-seller science book, *Fuzzy Thinking* (Kosko, 1993), which popularised the field of fuzzy logic.

Most fuzzy logic applications have been in the area of control engineering. However, fuzzy control systems use only a small part of fuzzy logic's power of knowledge representation. Benefits derived from the application of fuzzy logic models in knowledge-based and decision-support systems can be summarised as follows (Cox, 1999; Turban and Aronson, 2000):

- **Improved computational power**: Fuzzy rule-based systems perform faster than conventional expert systems and require fewer rules. A fuzzy expert system merges the rules, making them more powerful. Lotfi Zadeh believes that in a few years most expert systems will use fuzzy logic to solve highly nonlinear and computationally difficult problems.

- **Improved cognitive modelling**: Fuzzy systems allow the encoding of knowledge in a form that reflects the way experts think about a complex problem. They usually think in such imprecise terms as **high** and **low**, **fast** and **slow**, **heavy** and **light**, and they also use such terms as **very often** and **almost never**, **usually** and **hardly ever**, **frequently** and **occasionally**. In order to build conventional rules, we need to define the crisp boundaries for these terms, thus breaking down the expertise into fragments. However, this fragmentation leads to the poor performance of conventional expert systems when they deal with highly complex problems. In contrast, fuzzy expert systems model imprecise information, capturing expertise much more closely to the way it is represented in the expert mind, and thus improve cognitive modelling of the problem.

- **The ability to represent multiple experts**: Conventional expert systems are built for a very narrow domain with clearly defined expertise. It makes the system's performance fully dependent on the right choice of experts. Although a common strategy is to find just one expert, when a more complex expert system is being built or when expertise is not well defined, **multiple experts** might be needed. Multiple experts can expand the domain, synthesise expertise and eliminate the need for a world-class expert, who is likely to be both very expensive and hard to access. However, multiple experts seldom reach close agreements; there are often differences in opinions and even conflicts. This is especially true in areas such as business and management where no simple solution exists and conflicting views should be taken into account. Fuzzy expert systems can help to represent the expertise of multiple experts when they have opposing views.

Although fuzzy systems allow expression of expert knowledge in a more natural way, they still depend on the rules extracted from the experts, and thus might be smart or dumb. Some experts can provide very clever fuzzy rules – but some just guess and may even get them wrong. Therefore, all rules must be tested and tuned, which can be a prolonged and tedious process. For example, it took Hitachi engineers several years to test and tune only 54 fuzzy rules to guide the Sendai Subway System.

Using fuzzy logic development tools, we can easily build a simple fuzzy system, but then we may spend days, weeks and even months trying out new rules and tuning our system. How do we make this process faster or, in other words, how do we generate good fuzzy rules automatically?

In recent years, several methods based on neural network technology have been used to search numerical data for fuzzy rules. Adaptive or neural fuzzy systems can find new fuzzy rules, or change and tune existing ones based on the data provided. In other words, data in – rules out, or experience in – common sense out.

So, where is knowledge engineering heading?

Expert, neural and fuzzy systems have now matured and have been applied to a broad range of different problems, mainly in engineering, medicine, finance, business and management. Each technology handles the uncertainty and ambiguity of human knowledge differently, and each technology has found its place in knowledge engineering. They no longer compete; rather they complement each other. A synergy of expert systems with fuzzy logic and neural computing improves adaptability, robustness, fault-tolerance and speed of knowledge-based systems. Besides, computing with words makes them more 'human'. It is now common practice to build intelligent systems using existing theories rather than to propose new ones, and to apply these systems to real-world problems rather than to 'toy' problems.

1.3   Summary

We live in the era of the knowledge revolution, when the power of a nation is determined not by the number of soldiers in its army but the knowledge it possesses. Science, medicine, engineering and business propel nations towards a higher quality of life, but they also require highly qualified and skilful people. We are now adopting intelligent machines that can capture the expertise of such knowledgeable people and reason in a manner similar to humans.

The desire for intelligent machines was just an elusive dream until the first computer was developed. The early computers could manipulate large data bases effectively by following prescribed algorithms, but could not reason about the information provided. This gave rise to the question of whether computers could ever think. Alan Turing defined the intelligent behaviour of a computer as the ability to achieve human-level performance in a cognitive task. The Turing test provided a basis for the verification and validation of knowledge-based systems.

In 1956, a summer workshop at Dartmouth College brought together ten researchers interested in the study of machine intelligence, and a new science – artificial intelligence – was born.

Since the early 1950s, AI technology has developed from the curiosity of a few researchers to a valuable tool to support humans making decisions. We have seen historical cycles of AI from the era of great ideas and great expectations in the 1960s to the disillusionment and funding cutbacks in the early 1970s; from the development of the first expert systems such as DENDRAL, MYCIN and PROSPECTOR in the 1970s to the maturity of expert system technology and its massive applications in different areas in the 1980s/90s; from a simple binary model of neurons proposed in the 1940s to a dramatic resurgence of the field of artificial neural networks in the 1980s; from the introduction of fuzzy set theory and its being ignored by the West in the 1960s to numerous 'fuzzy' consumer products offered by the Japanese in the 1980s and world-wide acceptance of 'soft' computing and computing with words in the 1990s.

The development of expert systems created knowledge engineering, the process of building intelligent systems. Today it deals not only with expert systems but also with neural networks and fuzzy logic. Knowledge engineering is still an art rather than engineering, but attempts have already been made to extract rules automatically from numerical data through neural network technology.

Table 1.1 summarises the key events in the history of AI and knowledge engineering from the first work on AI by McCulloch and Pitts in 1943, to the recent trends of combining the strengths of expert systems, fuzzy logic and neural computing in modern knowledge-based systems capable of computing with words.

The most important lessons learned in this chapter are:

- Intelligence is the ability to learn and understand, to solve problems and to make decisions.

- Artificial intelligence is a science that has defined its goal as making machines do things that would require intelligence if done by humans.

- A machine is thought intelligent if it can achieve human-level performance in some cognitive task. To build an intelligent machine, we have to capture, organise and use human expert knowledge in some problem area.

- The realisation that the problem domain for intelligent machines had to be sufficiently restricted marked a major 'paradigm shift' in AI from general-purpose, knowledge-sparse, weak methods to domain-specific, knowledge-intensive methods. This led to the development of expert systems – computer programs capable of performing at a human-expert level in a narrow problem area. Expert systems use human knowledge and expertise in the form of specific rules, and are distinguished by the clean separation of the knowledge and the reasoning mechanism. They can also explain their reasoning procedures.

Table 1.1 A summary of the main events in the history of AI and knowledge engineering

Period	Key events
The birth of artificial intelligence (1943–56)	McCulloch and Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, 1943
	Turing, *Computing Machinery and Intelligence*, 1950
	The Electronic Numerical Integrator and Calculator project (von Neumann)
	Shannon, *Programming a Computer for Playing Chess*, 1950
	The Dartmouth College summer workshop on machine intelligence, artificial neural nets and automata theory, 1956
The rise of artificial intelligence (1956–late 1960s)	*LISP* (McCarthy)
	The General Problem Solver (GPR) project (Newell and Simon)
	Newell and Simon, *Human Problem Solving*, 1972
	Minsky, *A Framework for Representing Knowledge*, 1975
The disillusionment in artificial intelligence (late 1960s–early 1970s)	Cook, *The Complexity of Theorem Proving Procedures*, 1971
	Karp, *Reducibility Among Combinatorial Problems*, 1972
	The Lighthill Report, 1971
The discovery of expert systems (early 1970s–mid-1980s)	DENDRAL (Feigenbaum, Buchanan and Lederberg, Stanford University)
	MYCIN (Feigenbaum and Shortliffe, Stanford University)
	PROSPECTOR (Stanford Research Institute)
	PROLOG – a Logic Programming Language (Colmerauer, Roussel and Kowalski, France)
	EMYCIN (Stanford University)
	Waterman, *A Guide to Expert Systems*, 1986
The rebirth of artificial neural networks (1965–onwards)	Hopfield, *Neural Networks and Physical Systems with Emergent Collective Computational Abilities*, 1982
	Kohonen, *Self-Organized Formation of Topologically Correct Feature Maps*, 1982
	Rumelhart and McClelland, *Parallel Distributed Processing*, 1986
	The First IEEE International Conference on Neural Networks, 1987
	Haykin, *Neural Networks*, 1994
	Neural Network, MATLAB Application Toolbox (The MathWork, Inc.)

Table 1.1 (cont.)

Period	Key events
Evolutionary computation (early 1970s–onwards)	Rechenberg, *Evolutionsstrategien – Optimierung Technischer Systeme Nach Prinzipien der Biologischen Information*, 1973
	Holland, *Adaptation in Natural and Artificial Systems*, 1975
	Koza, *Genetic Programming: On the Programming of the Computers by Means of Natural Selection*, 1992
	Schwefel, *Evolution and Optimum Seeking*, 1995
	Fogel, *Evolutionary Computation – Towards a New Philosophy of Machine Intelligence*, 1995
Computing with words (late 1980s–onwards)	Zadeh, *Fuzzy Sets*, 1965
	Zadeh, *Fuzzy Algorithms*, 1969
	Mamdani, *Application of Fuzzy Logic to Approximate Reasoning Using Linguistic Synthesis*, 1977
	Sugeno, *Fuzzy Theory*, 1983
	Japanese 'fuzzy' consumer products (dishwashers, washing machines, air conditioners, television sets, copiers)
	Sendai Subway System (Hitachi, Japan), 1986
	Negoita, *Expert Systems and Fuzzy Systems*, 1985
	The First IEEE International Conference on Fuzzy Systems, 1992
	Kosko, *Neural Networks and Fuzzy Systems*, 1992
	Kosko, *Fuzzy Thinking*, 1993
	Yager and Zadeh, *Fuzzy Sets, Neural Networks and Soft Computing*, 1994
	Cox, *The Fuzzy Systems Handbook*, 1994
	Kosko, *Fuzzy Engineering*, 1996
	Zadeh, *Computing with Words – A Paradigm Shift*, 1996
	Fuzzy Logic, MATLAB Application Toolbox (The MathWork, Inc.)

- One of the main difficulties in building intelligent machines, or in other words in knowledge engineering, is the 'knowledge acquisition bottleneck' – extracting knowledge from human experts.

- Experts think in imprecise terms, such as **very often** and **almost never**, **usually** and **hardly ever**, **frequently** and **occasionally**, and use linguistic variables, such as **high** and **low**, **fast** and **slow**, **heavy** and **light**. Fuzzy logic

or fuzzy set theory provides a means to compute with words. It concentrates on the use of fuzzy values that capture the meaning of words, human reasoning and decision making, and provides a way of breaking through the computational burden of traditional expert systems.

- Expert systems can neither learn nor improve themselves through experience. They are individually created and demand large efforts for their development. It can take from five to ten person-years to build even a moderate expert system. Machine learning can accelerate this process significantly and enhance the quality of knowledge by adding new rules or changing incorrect ones.

- Artificial neural networks, inspired by biological neural networks, learn from historical cases and make it possible to generate rules automatically and thus avoid the tedious and expensive processes of knowledge acquisition, validation and revision.

- Integration of expert systems and ANNs, and fuzzy logic and ANNs improve the adaptability, fault tolerance and speed of knowledge-based systems.

Questions for review

1 Define intelligence. What is the intelligent behaviour of a machine?

2 Describe the Turing test for artificial intelligence and justify its validity from a modern standpoint.

3 Define artificial intelligence as a science. When was artificial intelligence born?

4 What are **weak methods**? Identify the main difficulties that led to the disillusion with AI in the early 1970s.

5 Define expert systems. What is the main difference between weak methods and the expert system technology?

6 List the common characteristics of early expert systems such as DENDRAL, MYCIN and PROSPECTOR.

7 What are the limitations of expert systems?

8 What are the differences between expert systems and artificial neural networks?

9 Why was the field of ANN reborn in the 1980s?

10 What are the premises on which fuzzy logic is based? When was fuzzy set theory introduced?

11 What are the main advantages of applying fuzzy logic in knowledge-based systems?

12 What are the benefits of integrating expert systems, fuzzy logic and neural computing?

References

Barto, A.G., Sutton, R.S. and Anderson C.W. (1983). Neurolike adaptive elements that can solve difficult learning control problems, *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13, pp. 834–846.

Boden, M.A. (1977). *Artificial Intelligence and Natural Man*. Basic Books, New York.

Broomhead, D.S. and Lowe, D. (1988). Multivariable functional interpolation and adaptive networks, *Complex Systems*, 2, 321–355.

Bryson, A.E. and Ho, Y.-C. (1969). *Applied Optimal Control*. Blaisdell, New York.

Buchanan, B.G., Sutherland, G.L. and Feigenbaum, E.A. (1969). Heuristic DENDRAL: a program for generating explanatory hypotheses in organic chemistry, *Machine Intelligence 4*, B. Meltzer, D. Michie and M. Swann, eds, Edinburgh University Press, Edinburgh, Scotland, pp. 209–254.

Cook, S.A. (1971). The complexity of theorem proving procedures, *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, New York, pp. 151–158.

Cowan, J.D. (1990). Neural networks: the early days, *Advances in Neural Information Processing Systems 2*, D.S. Tourefzky, ed., San Mateo, CA: Morgan Kaufman, pp. 828–842.

Cox, E. (1999). *The Fuzzy Systems Handbook: A Practitioner's Guide to Building, Using, and Maintaining Fuzzy Systems,* 2nd edn. Academic Press, San Diego, CA.

Duda, R., Gaschnig, J. and Hart, P. (1979). Model design in the PROSPECTOR consultant system for mineral exploration, *Expert Systems in the Microelectronic Age*, D. Michie, ed., Edinburgh University Press, Edinburgh, Scotland, pp. 153–167.

Durkin, J. (1994). *Expert Systems Design and Development*. Prentice Hall, Englewood Cliffs, NJ.

Feigenbaum, E.A., Buchanan, B.G. and Lederberg, J. (1971). On generality and problem solving: a case study using the DENDRAL program, *Machine Intelligence 6*, B. Meltzer and D. Michie, eds, Edinburgh University Press, Edinburgh, Scotland, pp. 165–190.

Fogel, D.B. (1995). *Evolutionary Computation – Towards a New Philosophy of Machine Intelligence. IEEE Press, Piscataway, NJ.*

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley Publishing Company, Reading, MA.

Greenblatt, R.D., Eastlake, D.E. and Crocker, S.D. (1967). The Greenblatt Chess Program, *Proceedings of the Fall Joint Computer Conference*, pp. 801–810.

Grossberg, S. (1980). How does a brain build a cognitive code?, *Psychological Review*, 87, pp. 1–51.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Holland, J.H. (1995). *Hidden Order: How Adaptation Builds Complexity*. Addison-Wesley, Reading, MA.

Hopfield, J.J. (1982). Neural networks and physical systems with emergent collective computational abilities, *Proceedings of the National Academy of Sciences of the USA*, 79, pp. 2554–2558.

Karp, R.M. (1972). Reducibility among combinatorial problems, *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher, eds, Plenum, New York, pp. 85–103.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps, *Biological Cybernetics*, 43, pp. 59–69.

Kosko, B. (1993). *Fuzzy Thinking: The New Science of Fuzzy Logic*. Hyperion, New York.

Kosko, B. (1997). *Fuzzy Engineering*. Prentice Hall, Upper Saddle River, NJ.

Koza, J.R. (1992). *Genetic Programming: On the Programming of the Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

LeCun, Y. (1988). A theoretical framework for back-propagation, *Proceedings of the 1988 Connectionist Models Summer School*, D. Touretzky, G. Hilton and T. Sejnowski, eds, Morgan Kaufmann, San Mateo, CA, pp. 21–28.

Lighthill, J. (1973). Artificial intelligence: a general survey, *Artificial Intelligence: A Paper Symposium*. J. Lighthill, N.S. Sutherland, R.M. Needham, H.C. Longuest-Higgins and D. Michie, eds, Science Research Council of Great Britain, London.

McCarthy, J. (1958). Programs with common sense, *Proceedings of the Symposium on Mechanisation of Thought Processes*, vol. 1, London, pp. 77–84.

McCulloch, W.S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–137.

Medsker, L. and Leibowitz, J. (1994). *Design and Development of Expert Systems and Neural Computing*. Macmillan, New York.

Minsky, M.L. (1975). A framework for representing knowledge, *The Psychology of Computer Vision*, P. Winston, ed., McGraw-Hill, New York, pp. 211–277.

Minsky, M.L. and Papert, S.A. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

Negoita, C.V. (1985). *Expert Systems and Fuzzy Systems*. Benjamin/Cummings, Menlo Park, CA.

Newell, A. and Simon, H.A. (1961). GPS, a program that simulates human thought, *Lernende Automatten*, H. Billing, ed., R. Oldenbourg, Munich, pp. 109–124.

Newell, A. and Simon, H.A. (1972). *Human Problem Solving*. Prentice Hall, Englewood Cliffs, NJ.

Omlin, C.W. and Giles, C.L. (1996). Rule revision with recurrent neural networks, *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 183–188.

Parker, D.B. (1987). Optimal algorithms for adaptive networks: second order back propagation, second order direct propagation, and second order Hebbian learning, *Proceedings of the IEEE 1st International Conference on Neural Networks*, San Diego, CA, vol. 2, pp. 593–600.

Rechenberg, I. (1965). *Cybernetic Solution Path of an Experimental Problem*. Ministry of Aviation, Royal Aircraft Establishment, Library Translation No. 1122, August.

Rechenberg, I. (1973). *Evolutionsstrategien – Optimierung Technischer Systeme Nach Prinzipien der Biologischen Information*. Friedrich Frommann Verlag (Günther Holzboog K.G.), Stuttgart–Bad Cannstatt.

Rosenblatt, F. (1962). *Principles of Neurodynamics*. Spartan, Chicago.

Rumelhart, D.E. and McClelland, J.L., eds (1986). *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*. 2 vols, MIT Press, Cambridge, MA.

Samuel, A.L. (1959). Some studies in machine learning using the game of checkers, *IBM Journal of Research and Development*, 3(3), 210–229.

Samuel, A.L. (1967). Some studies in machine learning using the game of checkers II – recent progress, *IBM Journal of Research and Development*, 11(6), 601–617.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. John Wiley, New York.

Shannon, C.E. (1950). Programming a computer for playing chess, *Philosophical Magazine*, 41(4), 256–275.

Shortliffe, E.H. (1976). *MYCIN: Computer-Based Medical Consultations*. Elsevier Press, New York.

Turban, E. and Aronson, J.E. (2000). *Decision Support Systems and Intelligent Systems,* 6th edn. Prentice Hall, Englewood Cliffs, NJ.

Turing, A.M. (1950). Computing machinery and intelligence, *Mind*, 59, 433–460.

van Melle, W. (1979). A domain independent production-rule system for consultation programs, *Proceedings of the IJCAI 6*, pp. 923–925.

van Melle, W., Shortliffe, E.H. and Buchanan B.G. (1981). EMYCIN: A domain-independent system that aids in constructing knowledge-based consultation programs, *Machine Intelligence, Infotech State of the Art Report 9*, no. 3.

Waterman, D.A. (1986). *A Guide to Expert Systems*. Addison-Wesley, Reading, MA.

Yager, R.R. and Zadeh, L.A., eds (1994). *Fuzzy Sets, Neural Networks and Soft Computing*. Van Nostrand Reinhold, New York.

Zadeh, L. (1965). Fuzzy sets, *Information and Control*, 8(3), 338–353.

Zahedi, F. (1993). *Intelligent Systems for Business: Expert Systems with Neural Networks*. Wadsworth, Belmont, CA.

Rule-based expert systems   **2**

In which we introduce the most popular choice for building knowledge-based systems: rule-based expert systems.

2.1   Introduction, or what is knowledge?

In the 1970s, it was finally accepted that to make a machine solve an intellectual problem one had to know the solution. In other words, one has to have knowledge, 'know-how', in some specific domain.

What is knowledge?
Knowledge is a theoretical or practical understanding of a subject or a domain. Knowledge is also the sum of what is currently known, and apparently knowledge is power. Those who possess knowledge are called experts. They are the most powerful and important people in their organisations. Any successful company has at least a few first-class experts and it cannot remain in business without them.

Who is generally acknowledged as an expert?
Anyone can be considered a domain expert if he or she has deep knowledge (of both facts and rules) and strong practical experience in a particular domain. The area of the domain may be limited. For example, experts in electrical machines may have only general knowledge about transformers, while experts in life insurance marketing might have limited understanding of a real estate insurance policy. In general, an expert is a skilful person who can do things other people cannot.

How do experts think?
The human mental process is internal, and it is too complex to be represented as an algorithm. However, most experts are capable of expressing their knowledge in the form of rules for problem solving. Consider a simple example. Imagine, you meet an alien! He wants to cross a road. Can you help him? You are an expert in crossing roads – you've been on this job for several years. Thus you are able to teach the alien. How would you do this?

You explain to the alien that he can cross the road safely when the traffic light is green, and he must stop when the traffic light is red. These are the basic rules. Your knowledge can be formulated as the following simple statements:

IF   the 'traffic light' is green
THEN   the action is go

IF   the 'traffic light' is red
THEN   the action is stop

These statements represented in the IF-THEN form are called **production rules** or just **rules**. The term 'rule' in AI, which is the most commonly used type of knowledge representation, can be defined as an IF-THEN structure that relates given information or facts in the IF part to some action in the THEN part. A rule provides some description of how to solve a problem. Rules are relatively easy to create and understand.

2.2   Rules as a knowledge representation technique

Any rule consists of two parts: the IF part, called the **antecedent** (**premise** or **condition**) and the THEN part called the **consequent** (**conclusion** or **action**).
    The basic syntax of a rule is:

IF   <antecedent>
THEN   <consequent>

In general, a rule can have multiple antecedents joined by the keywords AND (conjunction), OR (disjunction) or a combination of both. However, it is a good habit to avoid mixing conjunctions and disjunctions in the same rule.

IF   <antecedent 1>
AND   <antecedent 2>
            .
            .
            .
AND   <antecedent n>
THEN   <consequent>

IF   <antecedent 1>
OR   <antecedent 2>
            .
            .
            .
OR   <antecedent n>
THEN   <consequent>

The consequent of a rule can also have multiple clauses:

```
IF      <antecedent>
THEN    <consequent 1>
        <consequent 2>
               .
               .
               .
        <consequent m>
```

The antecedent of a rule incorporates two parts: an **object** (**linguistic object**) and its **value**. In our road crossing example, the linguistic object 'traffic light' can take either the value *green* or the value *red*. The object and its value are linked by an **operator**. The operator identifies the object and assigns the value. Operators such as *is*, *are*, *is not*, *are not* are used to assign a symbolic value to a linguistic object. But expert systems can also use mathematical operators to define an object as numerical and assign it to the numerical value. For example,

```
IF      'age of the customer' < 18
AND     'cash withdrawal' > 1000
THEN    'signature of the parent' is required
```

Similar to a rule antecedent, a consequent combines an object and a value connected by an operator. The operator assigns the value to the linguistic object. In the road crossing example, if the value of *traffic light* is *green*, the first rule sets the linguistic *object* action to the value *go*. Numerical objects and even simple arithmetical expression can also be used in a rule consequent.

```
IF      'taxable income' > 16283
THEN    'Medicare levy' = 'taxable income' ∗ 1.5 / 100
```

Rules can represent relations, recommendations, directives, strategies and heuristics (Durkin, 1994).

Relation
```
IF      the 'fuel tank' is empty
THEN    the car is dead
```

Recommendation
```
IF      the season is autumn
AND     the sky is cloudy
AND     the forecast is drizzle
THEN    the advice is 'take an umbrella'
```

Directive
```
IF      the car is dead
AND     the 'fuel tank' is empty
THEN    the action is 'refuel the car'
```

Strategy

IF	the car is dead
THEN	the action is 'check the fuel tank';
	step1 is complete

IF	step1 is complete
AND	the 'fuel tank' is full
THEN	the action is 'check the battery';
	step2 is complete

Heuristic

IF	the spill is liquid
AND	the 'spill pH' < 6
AND	the 'spill smell' is vinegar
THEN	the 'spill material' is 'acetic acid'

2.3 The main players in the expert system development team

As soon as knowledge is provided by a human expert, we can input it into a computer. We expect the computer to act as an intelligent assistant in some specific domain of expertise or to solve a problem that would otherwise have to be solved by an expert. We also would like the computer to be able to integrate new knowledge and to show its knowledge in a form that is easy to read and understand, and to deal with simple sentences in a natural language rather than an artificial programming language. Finally, we want our computer to explain how it reaches a particular conclusion. In other words, we have to build an **expert system**, a computer program capable of performing at the level of a human expert in a narrow problem area.

The most popular expert systems are rule-based systems. A great number have been built and successfully applied in such areas as business and engineering, medicine and geology, power systems and mining. A large number of companies produce and market software for rule-based expert system development – expert system shells for personal computers.

Expert system shells are becoming particularly popular for developing rule-based systems. Their main advantage is that the system builder can now concentrate on the knowledge itself rather than on learning a programming language.

What is an expert system shell?

An expert system shell can be considered as an expert system with the knowledge removed. Therefore, all the user has to do is to add the knowledge in the form of rules and provide relevant data to solve a problem.

Let us now look at who is needed to develop an expert system and what skills are needed.

In general, there are five members of the expert system development team: the domain expert, the knowledge engineer, the programmer, the project
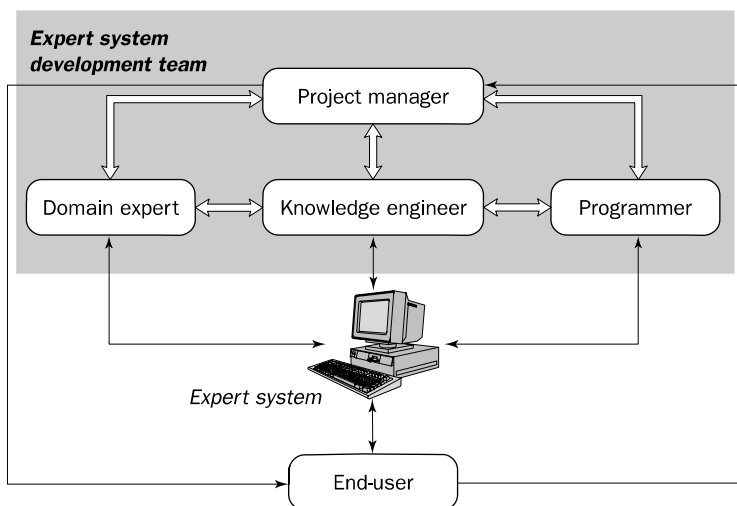
Figure 2.1   The main players of the expert system development team

manager and the end-user. The success of their expert system entirely depends on how well the members work together. The basic relations in the development team are summarised in Figure 2.1.

The **domain expert** is a knowledgeable and skilled person capable of solving problems in a specific area or **domain**. This person has the greatest expertise in a given domain. This expertise is to be captured in the expert system. Therefore, the expert must be able to communicate his or her knowledge, be willing to participate in the expert system development and commit a substantial amount of time to the project. The domain expert is the most important player in the expert system development team.

The **knowledge engineer** is someone who is capable of designing, building and testing an expert system. This person is responsible for selecting an appropriate task for the expert system. He or she interviews the domain expert to find out how a particular problem is solved. Through interaction with the expert, the knowledge engineer establishes what reasoning methods the expert uses to handle facts and rules and decides how to represent them in the expert system. The knowledge engineer then chooses some development software or an expert system shell, or looks at programming languages for encoding the knowledge (and sometimes encodes it himself). And finally, the knowledge engineer is responsible for testing, revising and integrating the expert system into the workplace. Thus, the knowledge engineer is committed to the project from the initial design stage to the final delivery of the expert system, and even after the project is completed, he or she may also be involved in maintaining the system.

The **programmer** is the person responsible for the actual programming, describing the domain knowledge in terms that a computer can understand. The programmer needs to have skills in symbolic programming in such AI

languages as LISP, Prolog and OPS5 and also some experience in the application of different types of expert system shells. In addition, the programmer should know conventional programming languages like C, Pascal, FORTRAN and Basic. If an expert system shell is used, the knowledge engineer can easily encode the knowledge into the expert system and thus eliminate the need for the programmer. However, if a shell cannot be used, a programmer must develop the knowledge and data representation structures (knowledge base and database), control structure (inference engine) and dialogue structure (user interface). The programmer may also be involved in testing the expert system.

The **project manager** is the leader of the expert system development team, responsible for keeping the project on track. He or she makes sure that all deliverables and milestones are met, interacts with the expert, knowledge engineer, programmer and end-user.

The **end-user**, often called just the **user**, is a person who uses the expert system when it is developed. The user might be an analytical chemist determining the molecular structure of soil from Mars (Feigenbaum *et al*., 1971), a junior doctor diagnosing an infectious blood disease (Shortliffe, 1976), an exploration geologist trying to discover a new mineral deposit (Duda *et al*., 1979), or a power system operator needing advice in an emergency (Negnevitsky, 1996). Each of these users of expert systems has different needs, which the system must meet: the system's final acceptance will depend on the user's satisfaction. The user must not only be confident in the expert system performance but also feel comfortable using it. Therefore, the design of the user interface of the expert system is also vital for the project's success; the end-user's contribution here can be crucial.

The development of an expert system can be started when all five players have joined the team. However, many expert systems are now developed on personal computers using expert system shells. This can eliminate the need for the programmer and also might reduce the role of the knowledge engineer. For small expert systems, the project manager, knowledge engineer, programmer and even the expert could be the same person. But all team players are required when large expert systems are developed.

2.4 Structure of a rule-based expert system

In the early 1970s, Newell and Simon from Carnegie-Mellon University proposed a production system model, the foundation of the modern rule-based expert systems (Newell and Simon, 1972). The production model is based on the idea that humans solve problems by applying their knowledge (expressed as production rules) to a given problem represented by problem-specific information. The production rules are stored in the long-term memory and the problem-specific information or facts in the short-term memory. The production system model and the basic structure of a rule-based expert system are shown in Figure 2.2.

A rule-based expert system has five components: the knowledge base, the database, the inference engine, the explanation facilities, and the user interface.
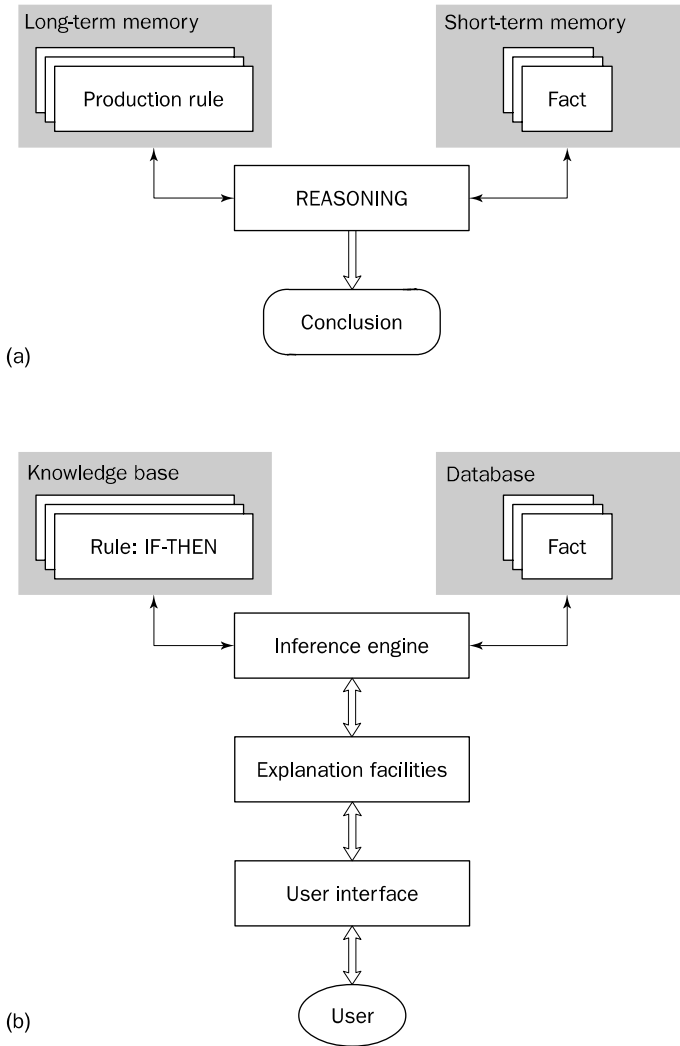
Figure 2.2 Production system and basic structure of a rule-based expert system: (a) production system model; (b) basic structure of a rule-based expert system

The **knowledge base** contains the domain knowledge useful for problem solving. In a rule-based expert system, the knowledge is represented as a set of rules. Each rule specifies a relation, recommendation, directive, strategy or heuristic and has the IF (condition) THEN (action) structure. When the condition part of a rule is satisfied, the rule is said to **fire** and the action part is executed.

The **database** includes a set of facts used to match against the IF (condition) parts of rules stored in the knowledge base.

The **inference engine** carries out the reasoning whereby the expert system reaches a solution. It links the rules given in the knowledge base with the facts provided in the database.

The **explanation facilities** enable the user to ask the expert system **how** a particular conclusion is reached and **why** a specific fact is needed. An expert system must be able to explain its reasoning and justify its advice, analysis or conclusion.

The **user interface** is the means of communication between a user seeking a solution to the problem and an expert system. The communication should be as meaningful and friendly as possible.

These five components are essential for any rule-based expert system. They constitute its core, but there may be a few additional components.

The **external interface** allows an expert system to work with external data files and programs written in conventional programming languages such as C, Pascal, FORTRAN and Basic. The complete structure of a rule-based expert system is shown in Figure 2.3.

The **developer interface** usually includes knowledge base editors, debugging aids and input/output facilities.

All expert system shells provide a simple **text editor** to input and modify rules, and to check their correct format and spelling. Many expert systems also
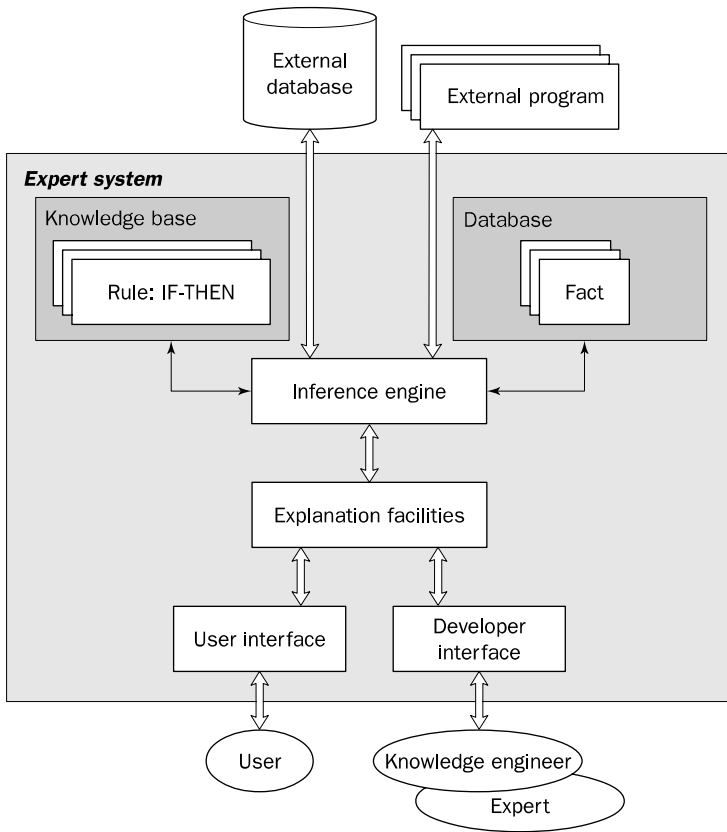
Figure 2.3   Complete structure of a rule-based expert system

include **book-keeping facilities** to monitor the changes made by the knowledge engineer or expert. If a rule is changed, the editor will automatically store the change date and the name of the person who made this change for later reference. This is very important when a number of knowledge engineers and experts have access to the knowledge base and can modify it.

Debugging aids usually consist of **tracing facilities** and **break packages**. Tracing provides a list of all rules fired during the program's execution, and a break package makes it possible to tell the expert system in advance where to stop so that the knowledge engineer or the expert can examine the current values in the database.

Most expert systems also accommodate input/output facilities such as **run-time knowledge acquisition**. This enables the running expert system to ask for needed information whenever this information is not available in the database. When the requested information is input by the knowledge engineer or the expert, the program resumes.

In general, the developer interface, and knowledge acquisition facilities in particular, are designed to enable a domain expert to input his or her knowledge directly in the expert system and thus to minimise the intervention of a knowledge engineer.

2.5 Fundamental characteristics of an expert system

An expert system is built to perform at a human expert level in a **narrow, specialised domain**. Thus, the most important characteristic of an expert system is its high-quality performance. No matter how fast the system can solve a problem, the user will not be satisfied if the result is wrong. On the other hand, the speed of reaching a solution is very important. Even the most accurate decision or diagnosis may not be useful if it is too late to apply, for instance, in an emergency, when a patient dies or a nuclear power plant explodes. Experts use their practical experience and understanding of the problem to find short cuts to a solution. Experts use rules of thumb or **heuristics**. Like their human counterparts, expert systems should apply heuristics to guide the reasoning and thus reduce the search area for a solution.

A unique feature of an expert system is its **explanation capability**. This enables the expert system to review its own reasoning and explain its decisions. An explanation in expert systems in effect traces the rules fired during a problem-solving session. This is, of course, a simplification; however a real or 'human' explanation is not yet possible because it requires basic understanding of the domain. Although a sequence of rules fired cannot be used to justify a conclusion, we can attach appropriate fundamental principles of the domain expressed as text to each rule, or at least each high-level rule, stored in the knowledge base. This is probably as far as the explanation capability can be taken. However, the ability to explain a line of reasoning may not be essential for some expert systems. For example, a scientific system built for experts may not be required to provide extensive explanations, because the conclusion it reaches

can be self-explanatory to other experts; a simple rule-tracing might be quite appropriate. On the other hand, expert systems used in decision making usually demand complete and thoughtful explanations, as the cost of a wrong decision may be very high.

Expert systems employ **symbolic reasoning** when solving a problem. Symbols are used to represent different types of knowledge such as facts, concepts and rules. Unlike conventional programs written for numerical data processing, expert systems are built for knowledge processing and can easily deal with qualitative data.

Conventional programs process data using algorithms, or in other words, a series of well-defined step-by-step operations. An algorithm always performs the same operations in the same order, and it always provides an exact solution. Conventional programs do not make mistakes – but programmers sometimes do. Unlike conventional programs, expert systems do not follow a prescribed sequence of steps. They permit inexact reasoning and can deal with incomplete, uncertain and fuzzy data.

Can expert systems make mistakes?

Even a brilliant expert is only a human and thus can make mistakes. This suggests that an expert system built to perform at a human expert level also should be allowed to make mistakes. But we still trust experts, although we do recognise that their judgements are sometimes wrong. Likewise, at least in most cases, we can rely on solutions provided by expert systems, but mistakes are possible and we should be aware of this.

Does it mean that conventional programs have an advantage over expert systems?

In theory, conventional programs always provide the same 'correct' solutions. However, we must remember that conventional programs can tackle problems if, and only if, the data is complete and exact. When the data is incomplete or includes some errors, a conventional program will provide either no solution at all or an incorrect one. In contrast, expert systems recognise that the available information may be incomplete or fuzzy, but they can work in such situations and still arrive at some reasonable conclusion.

Another important feature that distinguishes expert systems from conventional programs is that **knowledge is separated from its processing** (the knowledge base and the inference engine are split up). A conventional program is a mixture of knowledge and the control structure to process this knowledge. This mixing leads to difficulties in understanding and reviewing the program code, as any change to the code affects both the knowledge and its processing. In expert systems, knowledge is clearly separated from the processing mechanism. This makes expert systems much easier to build and maintain. When an expert system shell is used, a knowledge engineer or an expert simply enters rules in the knowledge base. Each new rule adds some new knowledge and makes the expert system smarter. The system can then be easily modified by changing or subtracting rules.

Table 2.1 Comparison of expert systems with conventional systems and human experts

Human experts	Expert systems	Conventional programs
Use knowledge in the form of rules of thumb or heuristics to solve problems in a narrow domain.	Process knowledge expressed in the form of rules and use symbolic reasoning to solve problems in a **narrow domain**.	Process data and use algorithms, a series of well-defined operations, to solve general numerical problems.
In a human brain, knowledge exists in a compiled form.	Provide a **clear separation of knowledge from its processing**.	Do not separate knowledge from the control structure to process this knowledge.
Capable of explaining a line of reasoning and providing the details.	**Trace the rules fired** during a problem-solving session and **explain how** a particular conclusion was reached and **why** specific data was needed.	Do not explain how a particular result was obtained and why input data was needed.
Use inexact reasoning and can deal with incomplete, uncertain and fuzzy information.	Permit **inexact reasoning** and can deal with incomplete, uncertain and fuzzy data.	Work only on problems where data is complete and exact.
Can make mistakes when information is incomplete or fuzzy.	**Can make mistakes** when data is incomplete or fuzzy.	Provide no solution at all, or a wrong one, when data is incomplete or fuzzy.
Enhance the quality of problem solving via years of learning and practical training. This process is slow, inefficient and expensive.	Enhance the quality of problem solving by adding new rules or adjusting old ones in the knowledge base. When new knowledge is acquired, **changes are easy** to accomplish.	Enhance the quality of problem solving by changing the program code, which affects both the knowledge and its processing, making changes difficult.

The characteristics of expert systems discussed above make them different from conventional systems and human experts. A comparison is shown in Table 2.1.

2.6 Forward chaining and backward chaining inference techniques

In a rule-based expert system, the domain knowledge is represented by a set of IF-THEN production rules and data is represented by a set of facts about the current situation. The inference engine compares each rule stored in the
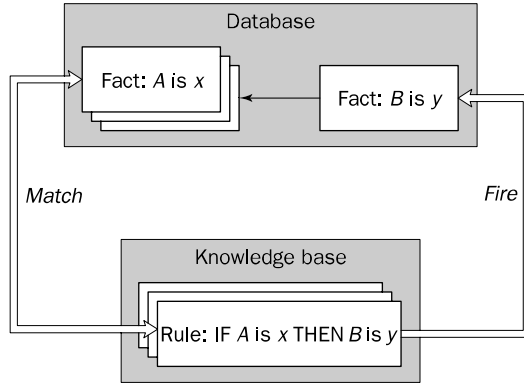
Figure 2.4   The inference engine cycles via a match-fire procedure

knowledge base with facts contained in the database. When the IF (condition) part of the rule matches a fact, the rule is **fired** and its THEN (action) part is executed. The fired rule may change the set of facts by adding a new fact, as shown in Figure 2.4. Letters in the database and the knowledge base are used to represent situations or concepts.

The matching of the rule IF parts to the facts produces **inference chains**. The inference chain indicates how an expert system applies the rules to reach a conclusion. To illustrate chaining inference techniques, consider a simple example.

Suppose the database initially includes facts A, B, C, D and E, and the knowledge base contains only three rules:

Rule 1:   IF Y is true
           AND D is true
           THEN Z is true

Rule 2:   IF X is true
           AND B is true
           AND E is true
           THEN Y is true

Rule 3:   IF A is true
           THEN X is true

The inference chain shown in Figure 2.5 indicates how the expert system applies the rules to infer fact Z. First Rule 3 is fired to deduce new fact X from given fact A. Then Rule 2 is executed to infer fact Y from initially known facts B and E, and already known fact X. And finally, Rule 1 applies initially known fact D and just-obtained fact Y to arrive at conclusion Z.

An expert system can display its inference chain to explain how a particular conclusion was reached; this is an essential part of its explanation facilities.
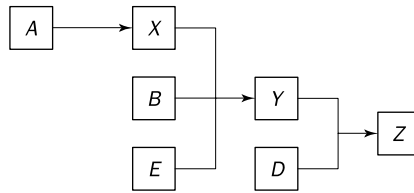
Figure 2.5   An example of an inference chain

The inference engine must decide when the rules have to be fired. There are two principal ways in which rules are executed. One is called **forward chaining** and the other **backward chaining** (Waterman and Hayes-Roth, 1978).

2.6.1   Forward chaining

The example discussed above uses forward chaining. Now consider this technique in more detail. Let us first rewrite our rules in the following form:

Rule 1: $Y \& D \rightarrow Z$

Rule 2: $X \& B \& E \rightarrow Y$

Rule 3: $A \rightarrow X$

Arrows here indicate the IF and THEN parts of the rules. Let us also add two more rules:

Rule 4: $C \rightarrow L$

Rule 5: $L \& M \rightarrow N$

Figure 2.6 shows how forward chaining works for this simple set of rules.

Forward chaining is the **data-driven** reasoning. The reasoning starts from the known data and proceeds forward with that data. Each time only the topmost rule is executed. When fired, the rule adds a new fact in the database. Any rule can be executed only once. The match-fire cycle stops when no further rules can be fired.

In the first cycle, only two rules, Rule 3: $A \rightarrow X$ and Rule 4: $C \rightarrow L$, match facts in the database. Rule 3: $A \rightarrow X$ is fired first as the topmost one. The IF part of this rule matches fact A in the database, its THEN part is executed and new fact X is added to the database. Then Rule 4: $C \rightarrow L$ is fired and fact L is also placed in the database.

In the second cycle, Rule 2: $X \& B \& E \rightarrow Y$ is fired because facts B, E and X are already in the database, and as a consequence fact Y is inferred and put in the database. This in turn causes Rule 1: $Y \& D \rightarrow Z$ to execute, placing fact Z in the database (cycle 3). Now the match-fire cycles stop because the IF part of Rule 5: $L \& M \rightarrow N$ does not match **all** facts in the database and thus Rule 5 cannot be fired.
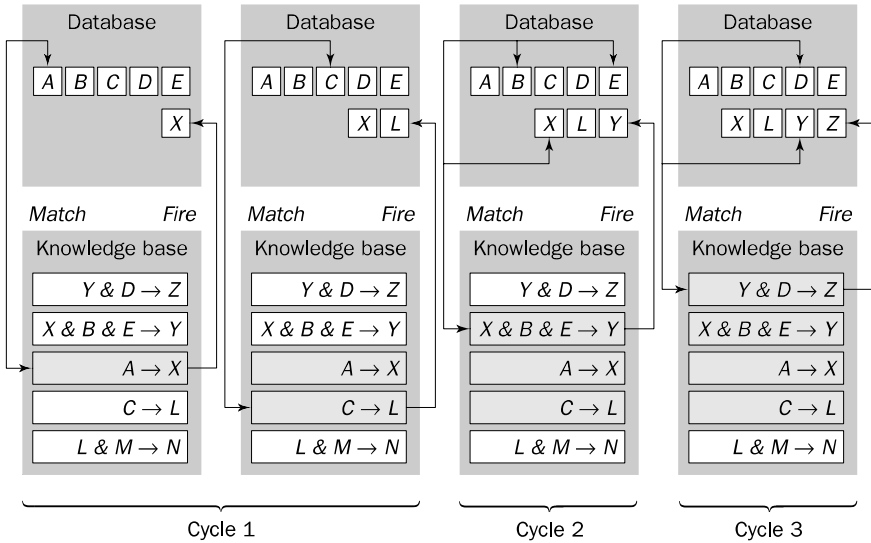
Figure 2.6 Forward chaining

Forward chaining is a technique for gathering information and then inferring from it whatever can be inferred. However, in forward chaining, many rules may be executed that have nothing to do with the established goal. Suppose, in our example, the goal was to determine fact Z. We had only five rules in the knowledge base and four of them were fired. But Rule 4: $C \rightarrow L$, which is unrelated to fact Z, was also fired among others. A real rule-based expert system can have hundreds of rules, many of which might be fired to derive new facts that are valid, but unfortunately unrelated to the goal. Therefore, if our goal is to infer only one particular fact, the forward chaining inference technique would not be efficient.

In such a situation, backward chaining is more appropriate.

2.6.2 Backward chaining

Backward chaining is the **goal-driven** reasoning. In backward chaining, an expert system has the goal (a hypothetical solution) and the inference engine attempts to find the evidence to prove it. First, the knowledge base is searched to find rules that might have the desired solution. Such rules must have the goal in their THEN (action) parts. If such a rule is found and its IF (condition) part matches data in the database, then the rule is fired and the goal is proved. However, this is rarely the case. Thus the inference engine puts aside the rule it is working with (the rule is said to **stack**) and sets up a new goal, a sub-goal, to prove the IF part of this rule. Then the knowledge base is searched again for rules that can prove the sub-goal. The inference engine repeats the process of stacking the rules until no rules are found in the knowledge base to prove the current sub-goal.
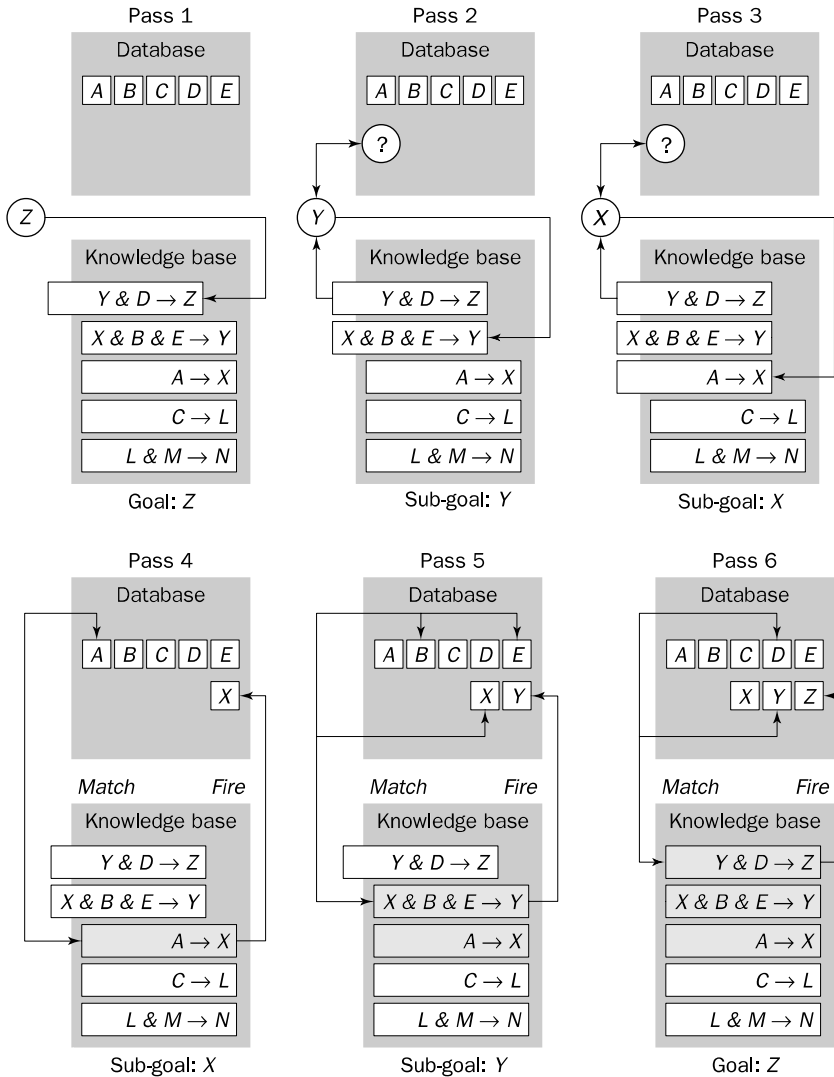
Figure 2.7 Backward chaining

Figure 2.7 shows how backward chaining works, using the rules for the forward chaining example.

In Pass 1, the inference engine attempts to infer fact Z. It searches the knowledge base to find the rule that has the goal, in our case fact Z, in its THEN part. The inference engine finds and stacks Rule 1: $Y \& D \rightarrow Z$. The IF part of Rule 1 includes facts Y and D, and thus these facts must be established.

In Pass 2, the inference engine sets up the sub-goal, fact Y, and tries to determine it. First it checks the database, but fact Y is not there. Then the knowledge base is searched again for the rule with fact Y in its THEN part. The

inference engine locates and stacks Rule 2: $X \& B \& E \rightarrow Y$. The IF part of Rule 2 consists of facts X, B and E, and these facts also have to be established.

In Pass 3, the inference engine sets up a new sub-goal, fact X. It checks the database for fact X, and when that fails, searches for the rule that infers X. The inference engine finds and stacks Rule 3: $A \rightarrow X$. Now it must determine fact A.

In Pass 4, the inference engine finds fact A in the database, Rule 3: $A \rightarrow X$ is fired and new fact X is inferred.

In Pass 5, the inference engine returns to the sub-goal fact Y and once again tries to execute Rule 2: $X \& B \& E \rightarrow Y$. Facts X, B and E are in the database and thus Rule 2 is fired and a new fact, fact Y, is added to the database.

In Pass 6, the system returns to Rule 1: $Y \& D \rightarrow Z$ trying to establish the original goal, fact Z. The IF part of Rule 1 matches all facts in the database, Rule 1 is executed and thus the original goal is finally established.

Let us now compare Figure 2.6 with Figure 2.7. As you can see, four rules were fired when forward chaining was used, but just three rules when we applied backward chaining. This simple example shows that the backward chaining inference technique is more effective when we need to infer one particular fact, in our case fact Z. In forward chaining, the data is known at the beginning of the inference process, and the user is never asked to input additional facts. In backward chaining, the goal is set up and the only data used is the data needed to support the direct line of reasoning, and the user may be asked to input any fact that is not in the database.

How do we choose between forward and backward chaining?

The answer is to study how a domain expert solves a problem. If an expert first needs to gather some information and then tries to infer from it whatever can be inferred, choose the forward chaining inference engine. However, if your expert begins with a hypothetical solution and then attempts to find facts to prove it, choose the backward chaining inference engine.

Forward chaining is a natural way to design expert systems for analysis and interpretation. For example, DENDRAL, an expert system for determining the molecular structure of unknown soil based on its mass spectral data (Feigenbaum *et al.*, 1971), uses forward chaining. Most backward chaining expert systems are used for diagnostic purposes. For instance, MYCIN, a medical expert system for diagnosing infectious blood diseases (Shortliffe, 1976), uses backward chaining.

Can we combine forward and backward chaining?

Many expert system shells use a combination of forward and backward chaining inference techniques, so the knowledge engineer does not have to choose between them. However, the basic inference mechanism is usually backward chaining. Only when a new fact is established is forward chaining employed to maximise the use of the new data.

2.7 MEDIA ADVISOR: a demonstration rule-based expert system

To illustrate some of the ideas discussed above, we next consider a simple rule-based expert system. The Leonardo expert system shell was selected as a tool to build a decision-support system called MEDIA ADVISOR. The system provides advice on selecting a medium for delivering a training program based on the trainee's job. For example, if a trainee is a mechanical technician responsible for maintaining hydraulic systems, an appropriate medium might be a workshop, where the trainee could learn how basic hydraulic components operate, how to troubleshoot hydraulics problems and how to make simple repairs to hydraulic systems. On the other hand, if a trainee is a clerk assessing insurance applications, a training program might include lectures on specific problems of the task, as well as tutorials where the trainee could evaluate real applications. For complex tasks, where trainees are likely to make mistakes, a training program should also include feedback on the trainee's performance.

Knowledge base

/* MEDIA ADVISOR: a demonstration rule-based expert system

Rule: 1
if   the environment is papers
or   the environment is manuals
or   the environment is documents
or   the environment is textbooks
then   the stimulus_situation is verbal

Rule: 2
if   the environment is pictures
or   the environment is illustrations
or   the environment is photographs
or   the environment is diagrams
then   the stimulus_situation is visual

Rule: 3
if   the environment is machines
or   the environment is buildings
or   the environment is tools
then   the stimulus_situation is 'physical object'

Rule: 4
if   the environment is numbers
or   the environment is formulas
or   the environment is 'computer programs'
then   the stimulus_situation is symbolic

Rule: 5
if   the job is lecturing
or   the job is advising
or   the job is counselling
then   the stimulus_response is oral

Rule: 6
if   the job is building
or   the job is repairing
or   the job is troubleshooting
then   the stimulus_response is 'hands-on'

Rule: 7
if   the job is writing
or   the job is typing
or   the job is drawing
then   the stimulus_response is documented

Rule: 8
if   the job is evaluating
or   the job is reasoning
or   the job is investigating
then   the stimulus_response is analytical

Rule: 9
if   the stimulus_situation is 'physical object'
and   the stimulus_response is 'hands-on'
and   feedback is required
then   medium is workshop

Rule: 10
if   the stimulus_situation is symbolic
and   the stimulus_response is analytical
and   feedback is required
then   medium is 'lecture – tutorial'

Rule: 11
if   the stimulus_situation is visual
and   the stimulus_response is documented
and   feedback is not required
then   medium is videocassette

Rule: 12
if   the stimulus_situation is visual
and   the stimulus_response is oral
and   feedback is required
then   medium is 'lecture – tutorial'

Rule: 13
if   the stimulus_situation is verbal
and   the stimulus_response is analytical
and   feedback is required
then   medium is 'lecture – tutorial'

Rule: 14
if   the stimulus_situation is verbal
and   the stimulus_response is oral
and   feedback is required
then   medium is 'role-play exercises'

/* The SEEK directive sets up the goal of the rule set

seek medium

Objects

MEDIA ADVISOR uses six linguistic objects: *environment, stimulus_situation, job, stimulus_response, feedback* and *medium*. Each object can take one of the allowed values (for example, object *environment* can take the value of *papers, manuals, documents, textbooks, pictures, illustrations, photographs, diagrams, machines, buildings, tools, numbers, formulas, computer programs*). An object and its value constitute a fact (for instance, the *environment* is *machines*, and the *job* is *repairing*). All facts are placed in the database.

Object	Allowed values	Object	Allowed values
environment	papers	job	lecturing
	manuals		advising
	documents		counselling
	textbooks		building
	pictures		repairing
	illustrations		troubleshooting
	photographs		writing
	diagrams		typing
	machines		drawing
	buildings		evaluating
	tools		reasoning
	numbers		investigating
	formulas		
	computer programs	stimulus_ response	oral
			hands-on
			documented
			analytical
stimulus_situation	verbal		
	visual		
	physical object	feedback	required
	symbolic		not required

Options

The final goal of the rule-based expert system is to produce a solution to the problem based on input data. In MEDIA ADVISOR, the solution is a medium selected from the list of four options:

  medium is workshop
  medium is 'lecture – tutorial'
  medium is videocassette
  medium is 'role-play exercises'

Dialogue

In the dialogue shown below, the expert system asks the user to input the data needed to solve the problem (the environment, the job and feedback). Based on the answers supplied by the user (answers are indicated by arrows), the expert system applies rules from its knowledge base to infer that the *stimulus_situation* is *physical object*, and the *stimulus_response* is *hands-on*. Rule 9 then selects one of the allowed values of *medium*.

What sort of environment is a trainee dealing with on the job?
⇒ **machines**

Rule: 3
if   the environment is machines
or   the environment is buildings
or   the environment is tools
then   the stimulus_situation is 'physical object'

In what way is a trainee expected to act or respond on the job?
⇒ **repairing**

Rule: 6
if   the job is building
or   the job is repairing
or   the job is troubleshooting
then   the stimulus_response is 'hands-on'

Is feedback on the trainee's progress required during training?
⇒ **required**

Rule: 9
if   the stimulus_situation is 'physical object'
and   the stimulus_response is 'hands-on'
and   feedback is required
then   medium is workshop

medium is workshop

Inference techniques

The standard inference technique in Leonardo is backward chaining with opportunistic forward chaining, which is the most efficient way to deal with the available information. However, Leonardo also enables the user to turn off either backward or forward chaining, and thus allows us to study each inference technique separately.

Forward chaining is data-driven reasoning, so we need first to provide some data. Assume that

the environment is **machines**
    '**environment**' instantiated by user input to '**machines**'

the job is **repairing**
    '**job**' instantiated by user input to '**repairing**'

feedback is **required**
    '**feedback**' instantiated by user input to '**required**'

The following process will then happen:

Rule: 3 fires   '**stimulus_situation**' instantiated by Rule: 3 to '**physical object**'
Rule: 6 fires   '**stimulus_response**' instantiated by Rule: 6 to '**hands-on**'
Rule: 9 fires   '**medium**' instantiated by Rule: 9 to '**workshop**'
No rules fire   stop

Backward chaining is goal-driven reasoning, so we need first to establish a hypothetical solution (the goal). Let us, for example, set up the following goal:

'medium' is 'workshop'

Pass 1
Trying Rule: 9   Need to find object '**stimulus_situation**'
Rule: 9 stacked   Object '**stimulus_situation**' sought as '**physical object**'

Pass 2
Trying Rule: 3   Need to find object '**environment**'
Rule: 3 stacked   Object '**environment**' sought as '**machines**'

ask environment
⇒**machines**   '**environment**' instantiated by user input to
           '**machines**'

Trying Rule: 3   '**stimulus_situation**' instantiated by Rule: 3 to
           '**physical object**'

Pass 3
Trying Rule: 9   Need to find object '**stimulus_response**'
Rule: 9 stacked   Object '**stimulus_response**' sought as '**hands-on**'

Pass 4
Trying Rule: 6   Need to find object '**job**'
Rule: 6 stacked   Object '**job**' sought as '**building**'

ask job
⇒ **repairing**   '**job**' instantiated by user input to '**repairing**'

Trying Rule: 6   '**stimulus_response**' instantiated by Rule: 6 to '**hands-on**'

Pass 5
Trying Rule: 9   Need to find object '**feedback**'
Rule: 9 stacked   Object '**feedback**' sought as '**required**'

ask feedback
⇒ **required**   '**feedback**' instantiated by user input to '**required**'

Trying Rule: 9   '**medium**' instantiated by Rule: 9 to '**workshop**'

medium is workshop

It is useful to have a tree diagram that maps a consultation session with an expert system. A diagram for MEDIA ADVISOR is shown in Figure 2.8. The root node is the goal; when the system is started, the inference engine seeks to determine the goal's value.
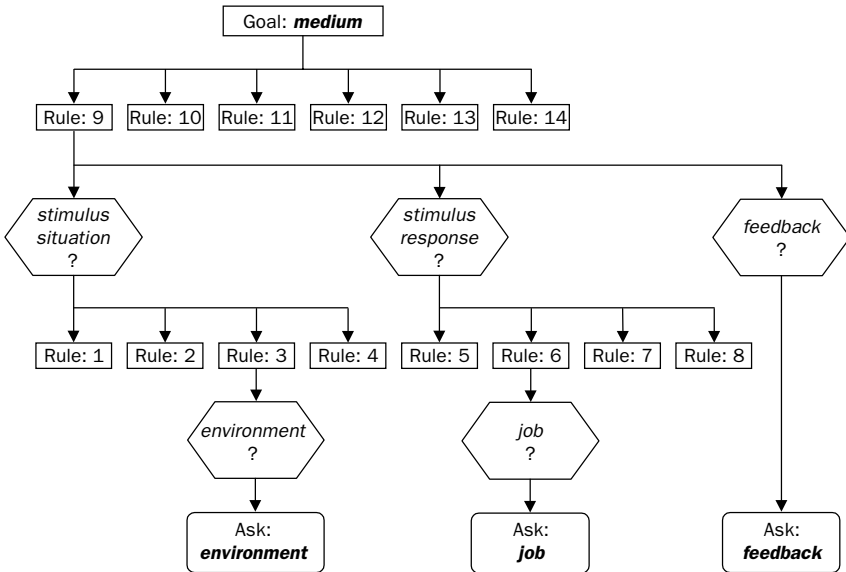
Figure 2.8   Tree diagram for the rule-based expert system MEDIA ADVISOR

Does MEDIA ADVISOR handle all possible situations?

When we start to use our expert system more often, we might find that the provided options do not cover all possible situations. For instance, the following dialogue might occur:

What sort of environment is a trainee dealing with on the job?
⇒**illustrations**

In what way is a trainee expected to act or respond on the job?
⇒**drawing**

Is feedback on the trainee's progress required during training?
⇒**required**

I am unable to draw any conclusions on the basis of the data.

Thus, MEDIA ADVISOR in its present state cannot handle this particular situation. Fortunately, the expert system can easily be expanded to accommodate more rules until it finally does what the user wants it to do.

2.8 Conflict resolution

Earlier in this chapter, we considered two simple rules for crossing a road. Let us now add a third rule. We will get the following set of rules:

Rule 1:
IF   the 'traffic light' is green
THEN   the action is go

Rule 2:
IF   the 'traffic light' is red
THEN   the action is stop

Rule 3:
IF   the 'traffic light' is red
THEN   the action is go

What will happen?

The inference engine compares IF (condition) parts of the rules with data available in the database, and when conditions are satisfied the rules are set to fire. The firing of one rule may affect the activation of other rules, and therefore the inference engine must allow only one rule to fire at a time. In our road crossing example, we have two rules, Rule 2 and Rule 3, with the same IF part. Thus both of them can be set to fire when the condition part is satisfied. These rules represent a conflict set. The inference engine must determine which rule to fire from such a set. A method for choosing a rule to fire when more than one rule can be fired in a given cycle is called **conflict resolution**.

If the traffic light is red, which rule should be executed?

In forward chaining, **both** rules would be fired. Rule 2 is fired first as the top-most one, and as a result, its THEN part is executed and linguistic object *action* obtains value *stop*. However, Rule 3 is also fired because the condition part of this rule matches the fact *'traffic light' is red*, which is still in the database. As a consequence, object *action* takes new value *go*. This simple example shows that the rule order is vital when the forward chaining inference technique is used.

How can we resolve a conflict?

The obvious strategy for resolving conflicts is to establish a goal and stop the rule execution when the goal is reached. In our problem, for example, the goal is to establish a value for linguistic object *action*. When the expert system determines a value for *action*, it has reached the goal and stops. Thus if the *traffic light* is *red*, Rule 2 is executed, object *action* attains value *stop* and the expert system stops. In the given example, the expert system makes a right decision; however if we arranged the rules in the reverse order, the conclusion would be wrong. It means that the rule order in the knowledge base is still very important.

Are there any other conflict resolution methods?

Several methods are in use (Giarratano and Riley, 1998; Shirai and Tsuji, 1982):

- Fire the rule with the highest priority. In simple applications, the priority can be established by placing the rules in an appropriate order in the knowledge base. Usually this strategy works well for expert systems with around 100 rules. However, in some applications, the data should be processed in order of importance. For example, in a medical consultation system (Durkin, 1994), the following priorities are introduced:

      Goal 1. Prescription is? Prescription

      RULE 1 Meningitis Prescription1
      (Priority 100)
      IF   Infection is Meningitis
      AND   The Patient is a Child
      THEN   Prescription is Number_1
      AND   Drug Recommendation is Ampicillin
      AND   Drug Recommendation is Gentamicin
      AND   Display Meningitis Prescription1

      RULE 2 Meningitis Prescription2
      (Priority 90)
      IF   Infection is Meningitis
      AND   The Patient is an Adult
      THEN   Prescription is Number_2
      AND   Drug Recommendation is Penicillin
      AND   Display Meningitis Prescription2

- Fire the most specific rule. This method is also known as the **longest matching strategy**. It is based on the assumption that a specific rule processes more information than a general one. For example,

  Rule 1:
  IF   the season is autumn
  AND   the sky is cloudy
  AND   the forecast is rain
  THEN   the advice is 'stay home'

  Rule 2:
  IF   the season is autumn
  THEN   the advice is 'take an umbrella'

  If the *season* is *autumn*, the *sky* is *cloudy* and the *forecast* is *rain*, then Rule 1 would be fired because its antecedent, the matching part, is more specific than that of Rule 2. But if it is known only that the *season* is *autumn*, then Rule 2 would be executed.

- Fire the rule that uses the **data most recently entered** in the database. This method relies on time tags attached to each fact in the database. In the conflict set, the expert system first fires the rule whose antecedent uses the data most recently added to the database. For example,

  Rule 1:
  IF   the forecast is rain   [08:16 PM 11/25/96]
  THEN   the advice is 'take an umbrella'

  Rule 2:
  IF   the weather is wet   [10:18 AM 11/26/96]
  THEN   the advice is 'stay home'

  Assume that the IF parts of both rules match facts in the database. In this case, Rule 2 would be fired since the fact *weather* is *wet* was entered after the fact *forecast* is *rain*. This technique is especially useful for real-time expert system applications when information in the database is constantly updated.

The conflict resolution methods considered above are simple and easily implemented. In most cases, these methods provide satisfactory solutions. However, as a program grows larger and more complex, it becomes increasingly difficult for the knowledge engineer to manage and oversee rules in the knowledge base. The expert system itself must take at least some of the burden and understand its own behaviour.

To improve the performance of an expert system, we should supply the system with some knowledge about the knowledge it possesses, or in other words, **metaknowledge**.

Metaknowledge can be simply defined as **knowledge about knowledge**. Metaknowledge is knowledge about the use and control of domain knowledge in an expert system (Waterman, 1986). In rule-based expert systems, meta-knowledge is represented by metarules. A metarule determines a strategy for the use of task-specific rules in the expert system.

What is the origin of metaknowledge?

The knowledge engineer transfers the knowledge of the domain expert to the expert system, learns how problem-specific rules are used, and gradually creates in his or her own mind a new body of knowledge, knowledge about the overall behaviour of the expert system. This new knowledge, or metaknowledge, is largely domain-independent. For example,

> Metarule 1:
> Rules supplied by experts have higher priorities than rules supplied by novices.

> Metarule 2:
> Rules governing the rescue of human lives have higher priorities than rules concerned with clearing overloads on power system equipment.

Can an expert system understand and use metarules?

Some expert systems provide a separate inference engine for metarules. However, most expert systems cannot distinguish between rules and metarules. Thus metarules should be given the highest priority in the existing knowledge base. When fired, a metarule 'injects' some important information into the database that can change the priorities of some other rules.

2.9   Advantages and disadvantages of rule-based expert systems

Rule-based expert systems are generally accepted as the best option for building knowledge-based systems.

Which features make rule-based expert systems particularly attractive for knowledge engineers?

Among these features are:

- **Natural knowledge representation**. An expert usually explains the problem-solving procedure with such expressions as this: 'In such-and-such situation, I do so-and-so'. These expressions can be represented quite naturally as IF-THEN production rules.

- **Uniform structure**. Production rules have the uniform IF-THEN structure. Each rule is an independent piece of knowledge. The very syntax of production rules enables them to be self-documented.

- **Separation of knowledge from its processing**. The structure of a rule-based expert system provides an effective separation of the knowledge base from the inference engine. This makes it possible to develop different applications using the same expert system shell. It also allows a graceful and easy expansion of the expert system. To make the system smarter, a knowledge engineer simply adds some rules to the knowledge base without intervening in the control structure.

- **Dealing with incomplete and uncertain knowledge**. Most rule-based expert systems are capable of representing and reasoning with incomplete and uncertain knowledge. For example, the rule

    IF   season is autumn
    AND   sky is 'cloudy'
    AND   wind is low
    THEN   forecast is clear   { cf 0.1 };
            forecast is drizzle   { cf 1.0 };
            forecast is rain   { cf 0.9 }

  could be used to express the uncertainty of the following statement, 'If the season is autumn and it looks like drizzle, then it will probably be another wet day today'.

  The rule represents the uncertainty by numbers called **certainty factors** {cf 0.1}. The expert system uses certainty factors to establish the degree of confidence or level of belief that the rule's conclusion is true. This topic will be considered in detail in Chapter 3.

All these features of the rule-based expert systems make them highly desirable for knowledge representation in real-world problems.

Are rule-based expert systems problem-free?

There are three main shortcomings:

- **Opaque relations between rules**. Although the individual production rules tend to be relatively simple and self-documented, their logical interactions within the large set of rules may be opaque. Rule-based systems make it difficult to observe how individual rules serve the overall strategy. This problem is related to the lack of hierarchical knowledge representation in the rule-based expert systems.
- **Ineffective search strategy**. The inference engine applies an exhaustive search through all the production rules during each cycle. Expert systems with a large set of rules (over 100 rules) can be slow, and thus large rule-based systems can be unsuitable for real-time applications.
- **Inability to learn**. In general, rule-based expert systems do not have an ability to learn from the experience. Unlike a human expert, who knows when to 'break the rules', an expert system cannot automatically modify its knowledge base, or adjust existing rules or add new ones. The knowledge engineer is still responsible for revising and maintaining the system.

2.10 Summary

In this chapter, we presented an overview of rule-based expert systems. We briefly discussed what knowledge is, and how experts express their knowledge in the form of production rules. We identified the main players in the expert

system development team and showed the structure of a rule-based system. We discussed fundamental characteristics of expert systems and noted that expert systems can make mistakes. Then we reviewed the forward and backward chaining inference techniques and debated conflict resolution strategies. Finally, the advantages and disadvantages of rule-based expert systems were examined.

The most important lessons learned in this chapter are:

- Knowledge is a theoretical or practical understanding of a subject. Knowledge is the sum of what is currently known.

- An expert is a person who has deep knowledge in the form of facts and rules and strong practical experience in a particular domain. An expert can do things other people cannot.

- The experts can usually express their knowledge in the form of production rules.

- Production rules are represented as IF (antecedent) THEN (consequent) statements. A production rule is the most popular type of knowledge representation. Rules can express relations, recommendations, directives, strategies and heuristics.

- A computer program capable of performing at a human-expert level in a narrow problem domain area is called an expert system. The most popular expert systems are rule-based expert systems.

- In developing rule-based expert systems, shells are becoming particularly common. An expert system shell is a skeleton expert system with the knowledge removed. To build a new expert system application, all the user has to do is to add the knowledge in the form of rules and provide relevant data. Expert system shells offer a dramatic reduction in the development time of expert systems.

- The expert system development team should include the domain expert, the knowledge engineer, the programmer, the project manager and the end-user. The knowledge engineer designs, builds and tests an expert system. He or she captures the knowledge from the domain expert, establishes reasoning methods and chooses the development software. For small expert systems based on expert system shells, the project manager, knowledge engineer, programmer and even the expert could be the same person.

- A rule-based expert system has five basic components: the knowledge base, the database, the inference engine, the explanation facilities and the user interface. The knowledge base contains the domain knowledge represented as a set of rules. The database includes a set of facts used to match against the IF parts of rules. The inference engine links the rules with the facts and carries out the reasoning whereby the expert system reaches a solution. The explanation facilities enable the user to query the expert system about **how** a particular conclusion is reached and **why** a specific fact is needed. The user interface is the means of communication between a user and an expert system.

- Expert systems separate knowledge from its processing by splitting up the knowledge base and the inference engine. This makes the task of building and maintaining an expert system much easier. When an expert system shell is used, a knowledge engineer or an expert simply enter rules in the knowledge base. Each new rule adds some new knowledge and makes the expert system smarter.

- Expert systems provide a limited explanation capability by tracing the rules fired during a problem-solving session.

- Unlike conventional programs, expert systems can deal with incomplete and uncertain data and permit inexact reasoning. However, like their human counterparts, expert systems can make mistakes when information is incomplete or fuzzy.

- There are two principal methods to direct search and reasoning: forward chaining and backward chaining inference techniques. Forward chaining is data-driven reasoning; it starts from the known data and proceeds forward until no further rules can be fired. Backward chaining is goal-driven reasoning; an expert system has a hypothetical solution (the goal), and the inference engine attempts to find the evidence to prove it.

- If more than one rule can be fired in a given cycle, the inference engine must decide which rule to fire. A method for deciding is called conflict resolution.

- Rule-based expert systems have the advantages of natural knowledge representation, uniform structure, separation of knowledge from its processing, and coping with incomplete and uncertain knowledge.

- Rule-based expert systems also have disadvantages, especially opaque relations between rules, ineffective search strategy, and inability to learn.

Questions for review

1 What is knowledge? Explain why experts usually have detailed knowledge of a limited area of a specific domain. What do we mean by heuristic?

2 What is a production rule? Give an example and define two basic parts of the production rule.

3 List and describe the five major players in the expert system development team. What is the role of the knowledge engineer?

4 What is an expert system shell? Explain why the use of an expert system shell can dramatically reduce the development time of an expert system.

5 What is a production system model? List and define the five basic components of an expert system.

6 What are the fundamental characteristics of an expert system? What are the differences between expert systems and conventional programs?

7   Can an expert system make mistakes? Why?

8   Describe the forward chaining inference process. Give an example.

9   Describe the backward chaining inference process. Give an example.

10 List problems for which the forward chaining inference technique is appropriate. Why is backward chaining used for diagnostic problems?

11 What is a conflict set of rules? How can we resolve a conflict? List and describe the basic conflict resolution methods.

12 List advantages of rule-based expert systems. What are their disadvantages?

References

Duda, R., Gaschnig, J. and Hart, P. (1979). Model design in the PROSPECTOR consultant system for mineral exploration, *Expert Systems in the Microelectronic Age*, D. Michie, ed., Edinburgh University Press, Edinburgh, Scotland, pp. 153–167.

Durkin, J. (1994). *Expert Systems Design and Development*. Prentice Hall, Englewood Cliffs, NJ.

Feigenbaum, E.A., Buchanan, B.G. and Lederberg, J. (1971). On generality and problem solving: a case study using the DENDRAL program, *Machine Intelligence 6*, B. Meltzer and D. Michie, eds, Edinburgh University Press, Edinburgh, Scotland, pp. 165–190.

Giarratano, J. and Riley, G. (1998). *Expert Systems: Principles and Programming*, 3rd edn. PWS Publishing Company, Boston.

Negnevitsky, M. (1996). Crisis management in power systems: a knowledge based approach, *Applications of Artificial Intelligence in Engineering XI*, R.A. Adey, G. Rzevski and A.K. Sunol, eds, Computational Mechanics Publications, South-ampton, UK, pp. 122–141.

Newell, A. and Simon, H.A. (1972). *Human Problem Solving*. Prentice Hall, Englewood Cliffs, NJ.

Shirai, Y. and Tsuji, J. (1982). *Artificial Intelligence: Concepts, Technologies and Applica-tions*. John Wiley, New York.

Shortliffe, E.H. (1976). *MYCIN: Computer-Based Medical Consultations*. Elsevier Press, New York.

Waterman, D.A. (1986). *A Guide to Expert Systems*. Addison-Wesley, Reading, MA.

Waterman, D.A. and Hayes-Roth, F. (1978). An overview of pattern-directed inference systems, *Pattern-Directed Inference Systems*, D.A. Waterman and F. Hayes-Roth, eds, Academic Press, New York.

Uncertainty management in rule-based expert systems

<div align="right">

3

</div>

In which we present the main uncertainty management paradigms, Bayesian reasoning and certainty factors, discuss their relative merits and consider examples to illustrate the theory.

3.1 Introduction, or what is uncertainty?

One of the common characteristics of the information available to human experts is its imperfection. Information can be incomplete, inconsistent, uncertain, or all three. In other words, information is often unsuitable for solving a problem. However, an expert can cope with these defects and can usually make correct judgements and right decisions. Expert systems also have to be able to handle uncertainty and draw valid conclusions.

What is uncertainty in expert systems?
Uncertainty can be defined as the lack of the exact knowledge that would enable us to reach a perfectly reliable conclusion (Stephanou and Sage, 1987). Classical logic permits only exact reasoning. It assumes that perfect knowledge always exists and the law of the excluded middle can always be applied:

    IF   *A* is true
    THEN   *A* is not false

and

    IF   *B* is false
    THEN   *B* is not true

Unfortunately most real-world problems where expert systems could be used do not provide us with such clear-cut knowledge. The available information often contains inexact, incomplete or even unmeasurable data.

What are the sources of uncertain knowledge in expert systems?
In general, we can identify four main sources: weak implications, imprecise language, unknown data, and the difficulty of combining the views of different

experts (Bonissone and Tong, 1985). Let us consider these sources in more detail.

- **Weak implications**. Rule-based expert systems often suffer from weak implications and vague associations. Domain experts and knowledge engineers have the painful, and rather hopeless, task of establishing concrete correlations between IF (condition) and THEN (action) parts of the rules. Therefore, expert systems need to have the ability to handle vague associations, for example by accepting the degree of correlations as numerical certainty factors.

- **Imprecise language**. Our natural language is inherently ambiguous and imprecise. We describe facts with such terms as **often** and **sometimes**, **frequently** and **hardly ever**. As a result, it can be difficult to express knowledge in the precise IF-THEN form of production rules. However, if the meaning of the facts is quantified, it can be used in expert systems. In 1944, Ray Simpson asked 355 high school and college students to place 20 terms like **often** on a scale between 1 and 100 (Simpson, 1944). In 1968, Milton Hakel repeated this experiment (Hakel, 1968). Their results are presented in Table 3.1.

   Quantifying the meaning of the terms enables an expert system to establish an appropriate matching of the IF (condition) part of the rules with facts available in the database.

- **Unknown data**. When the data is incomplete or missing, the only solution is to accept the value 'unknown' and proceed to an approximate reasoning with this value.

- **Combining the views of different experts**. Large expert systems usually combine the knowledge and expertise of a number of experts. For example, nine experts participated in the development of PROSPECTOR, an expert system for mineral exploration (Duda *et al.*, 1979). Unfortunately, experts seldom reach exactly the same conclusions. Usually, experts have contradictory opinions and produce conflicting rules. To resolve the conflict, the knowledge engineer has to attach a weight to each expert and then calculate the composite conclusion. However, even a domain expert generally does not have the same uniform level of expertise throughout a domain. In addition, no systematic method exists to obtain weights.

In summary, an expert system should be able to manage uncertainties because any real-world domain contains inexact knowledge and needs to cope with incomplete, inconsistent or even missing data. A number of numeric and non-numeric methods have been developed to deal with uncertainty in rule-based expert systems (Bhatnagar and Kanal, 1986). In this chapter, we consider the most popular uncertainty management paradigms: Bayesian reasoning and certainty factors. However, we first look at the basic principles of classical probability theory.

Table 3.1 Quantification of ambiguous and imprecise terms on a time-frequency scale

Ray Simpson (1944)		Milton Hakel (1968)	
Term	Mean value	Term	Mean value
Always	99	Always	100
Very often	88	Very often	87
Usually	85	Usually	79
Often	78	Often	74
Generally	78	Rather often	74
Frequently	73	Frequently	72
Rather often	65	Generally	72
About as often as not	50	About as often as not	50
Now and then	20	Now and then	34
Sometimes	20	Sometimes	29
Occasionally	20	Occasionally	28
Once in a while	15	Once in a while	22
Not often	13	Not often	16
Usually not	10	Usually not	16
Seldom	10	Seldom	9
Hardly ever	7	Hardly ever	8
Very seldom	6	Very seldom	7
Rarely	5	Rarely	5
Almost never	3	Almost never	2
Never	0	Never	0

3.2   Basic probability theory

The basic concept of probability plays a significant role in our everyday life. We try to determine the probability of rain and the prospects of our promotion, the odds that the Australian cricket team will win the next test match, and the likelihood of winning a million dollars in Tattslotto.

The concept of probability has a long history that goes back thousands of years when words like 'probably', 'likely', 'maybe', 'perhaps' and 'possibly' were introduced into spoken languages (Good, 1959). However, the mathematical theory of probability was formulated only in the 17th century.

How can we define probability?

The probability of an event is the proportion of cases in which the event occurs (Good, 1959). Probability can also be defined as a scientific measure of chance. Detailed analysis of modern probability theory can be found in such well-known textbooks as Feller (1957) and Fine (1973). In this chapter, we examine only the basic ideas used in representing uncertainties in expert systems.

Probability can be expressed mathematically as a numerical index with a range between zero (an absolute impossibility) to unity (an absolute certainty). Most events have a probability index strictly between 0 and 1, which means that

each event has **at least** two possible outcomes: favourable outcome or success, and unfavourable outcome or failure.

The probability of success and failure can be determined as follows:

$$P(\text{success}) = \frac{\text{the number of successes}}{\text{the number of possible outcomes}} \tag{3.1}$$

$$P(\text{failure}) = \frac{\text{the number of failures}}{\text{the number of possible outcomes}} \tag{3.2}$$

Therefore, if s is the number of times success can occur, and f is the number of times failure can occur, then

$$P(\text{success}) = p = \frac{s}{s+f} \tag{3.3}$$

$$P(\text{failure}) = q = \frac{f}{s+f} \tag{3.4}$$

and

$$p + q = 1 \tag{3.5}$$

Let us consider classical examples with a coin and a die. If we throw a coin, the probability of getting a head will be equal to the probability of getting a tail. In a single throw, $s = f = 1$, and therefore the probability of getting a head (or a tail) is 0.5.

Consider now a dice and determine the probability of getting a 6 from a single throw. If we assume a 6 as the only success, then $s = 1$ and $f = 5$, since there is just one way of getting a 6, and there are five ways of not getting a 6 in a single throw. Therefore, the probability of getting a 6 is

$$p = \frac{1}{1+5} = 0.1666$$

and the probability of not getting a 6 is

$$q = \frac{5}{1+5} = 0.8333$$

So far, we have been concerned with events that are independent and mutually exclusive (i.e. events that cannot happen simultaneously). In the dice experiment, the two events of obtaining a 6 and, for example, a 1 are mutually exclusive because we cannot obtain a 6 **and** a 1 simultaneously in a single throw. However, events that are not independent may affect the likelihood of one or the other occurring. Consider, for instance, the probability of getting a 6 in a

single throw, knowing this time that a 1 has not come up. There are still five ways of not getting a 6, but one of them can be eliminated as we know that a 1 has not been obtained. Thus,

$$p = \frac{1}{1 + (5 - 1)}$$

Let A be an event in the world and B be another event. Suppose that events A and B are not mutually exclusive, but occur conditionally on the occurrence of the other. The probability that event A will occur if event B occurs is called the **conditional probability**. Conditional probability is denoted mathematically as $p(A|B)$ in which the vertical bar represents GIVEN and the complete probability expression is interpreted as 'Conditional probability of event A occurring given that event B has occurred'.

$$p(A|B) = \frac{\text{the number of times } A \text{ and } B \text{ can occur}}{\text{the number of times } B \text{ can occur}} \quad (3.6)$$

The number of times A and B can occur, or the probability that both A and B will occur, is called the **joint probability** of A and B. It is represented mathematically as $p(A \cap B)$. The number of ways B can occur is the probability of B, $p(B)$, and thus

$$p(A|B) = \frac{p(A \cap B)}{p(B)} \quad (3.7)$$

Similarly, the conditional probability of event B occurring given that event A has occurred equals

$$p(B|A) = \frac{p(B \cap A)}{p(A)} \quad (3.8)$$

Hence,

$$p(B \cap A) = p(B|A) \times p(A) \quad (3.9)$$

The joint probability is commutative, thus

$$p(A \cap B) = p(B \cap A)$$

Therefore,

$$p(A \cap B) = p(B|A) \times p(A) \quad (3.10)$$

Substituting Eq. (3.10) into Eq. (3.7) yields the following equation:

$$p(A|B) = \frac{p(B|A) \times p(A)}{p(B)},$$ (3.11)

where:

$p(A|B)$ is the conditional probability that event A occurs given that event B has occurred;

$p(B|A)$ is the conditional probability of event B occurring given that event A has occurred;

$p(A)$ is the probability of event A occurring;

$p(B)$ is the probability of event B occurring.

Equation (3.11) is known as the **Bayesian rule**, which is named after Thomas Bayes, an 18th-century British mathematician who introduced this rule.

The concept of conditional probability introduced so far considered that event A was dependent upon event B. This principle can be extended to event A being dependent on a number of mutually exclusive events $B_1, B_2, \ldots, B_n$. The following set of equations can then be derived from Eq. (3.7):

$$p(A \cap B_1) = p(A|B_1) \times p(B_1)$$
$$p(A \cap B_2) = p(A|B_2) \times p(B_2)$$
$$\vdots$$
$$p(A \cap B_n) = p(A|B_n) \times p(B_n)$$

or when combined:

$$\sum_{i=1}^{n} p(A \cap B_i) = \sum_{i=1}^{n} p(A|B_i) \times p(B_i)$$ (3.12)

If Eq. (3.12) is summed over an exhaustive list of events for B_i as illustrated in Figure 3.1, we obtain

$$\sum_{i=1}^{n} p(A \cap B_i) = p(A)$$ (3.13)

It reduces Eq. (3.12) to the following conditional probability equation:

$$p(A) = \sum_{i=1}^{n} p(A|B_i) \times p(B_i)$$ (3.14)
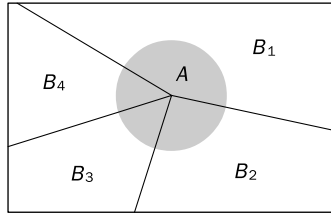
Figure 3.1   The joint probability

If the occurrence of event A depends on only two mutually exclusive events, i.e. B and NOT B, then Eq. (3.14) becomes

$$p(A) = p(A|B) \times p(B) + p(A|\neg B) \times p(\neg B), \tag{3.15}$$

where $\neg$ is the logical function NOT.

Similarly,

$$p(B) = p(B|A) \times p(A) + p(B|\neg A) \times p(\neg A) \tag{3.16}$$

Let us now substitute Eq. (3.16) into the Bayesian rule (3.11) to yield

$$p(A|B) = \frac{p(B|A) \times p(A)}{p(B|A) \times p(A) + p(B|\neg A) \times p(\neg A)} \tag{3.17}$$

Equation (3.17) provides the background for the application of probability theory to manage uncertainty in expert systems.

3.3   Bayesian reasoning

With Eq. (3.17) we can now leave basic probability theory and turn our attention back to expert systems. Suppose all rules in the knowledge base are represented in the following form:

> IF E is true
> THEN H is true {with probability p}

This rule implies that if event E occurs, then the probability that event H will occur is p.

What if event *E* has occurred but we do not know whether event *H* has occurred? Can we compute the probability that event *H* has occurred as well?

Equation (3.17) tells us how to do this. We simply use H and E instead of A and B. In expert systems, H usually represents a hypothesis and E denotes evidence to

support this hypothesis. Thus, Eq. (3.17) expressed in terms of hypotheses and evidence looks like this (Firebaugh, 1989):

$$p(H|E) = \frac{p(E|H) \times p(H)}{p(E|H) \times p(H) + p(E|\neg H) \times p(\neg H)} \qquad (3.18)$$

where:

$p(H)$ is the prior probability of hypothesis H being true;

$p(E|H)$ is the probability that hypothesis H being true will result in evidence E;

$p(\neg H)$ is the prior probability of hypothesis H being false;

$p(E|\neg H)$ is the probability of finding evidence E even when hypothesis H is false.

Equation (3.18) suggests that the probability of hypothesis H, $p(H)$, has to be defined before any evidence is examined. In expert systems, the probabilities required to solve a problem are provided by experts. An expert determines the prior probabilities for possible hypotheses $p(H)$ and $p(\neg H)$, and also the conditional probabilities for observing evidence E if hypothesis H is true, $p(E|H)$, and if hypothesis H is false, $p(E|\neg H)$. Users provide information about the evidence observed and the expert system computes $p(H|E)$ for hypothesis H in light of the user-supplied evidence E. Probability $p(H|E)$ is called the **posterior probability** of hypothesis H upon observing evidence E.

What if the expert, based on single evidence *E*, cannot choose a single hypothesis but rather provides multiple hypotheses *H*₁, *H*₂, ..., *H*ₘ? Or given multiple evidences *E*₁, *E*₂, ..., *E*ₙ, the expert also produces multiple hypotheses?

We can generalise Eq. (3.18) to take into account both multiple hypotheses $H_1, H_2, \ldots, H_m$ and multiple evidences $E_1, E_2, \ldots, E_n$. But the hypotheses as well as the evidences must be mutually exclusive and exhaustive.

Single evidence E and multiple hypotheses $H_1, H_2, \ldots, H_m$ follow:

$$p(H_i|E) = \frac{p(E|H_i) \times p(H_i)}{\sum_{k=1}^{m} p(E|H_k) \times p(H_k)} \qquad (3.19)$$

Multiple evidences $E_1, E_2, \ldots, E_n$ and multiple hypotheses $H_1, H_2, \ldots, H_m$ follow:

$$p(H_i|E_1 E_2 \ldots E_n) = \frac{p(E_1 E_2 \ldots E_n|H_i) \times p(H_i)}{\sum_{k=1}^{m} p(E_1 E_2 \ldots E_n|H_k) \times p(H_k)} \qquad (3.20)$$

An application of Eq. (3.20) requires us to obtain the conditional probabilities of all possible combinations of evidences for all hypotheses. This requirement

places an enormous burden on the expert and makes his or her task practically impossible. Therefore, in expert systems, subtleties of evidence should be suppressed and conditional independence among different evidences assumed (Ng and Abramson, 1990). Thus, instead of unworkable Eq. (3.20), we attain:

$$p(H_i|E_1E_2\ldots E_n) = \frac{p(E_1|H_i) \times p(E_2|H_i) \times \ldots \times p(E_n|H_i) \times p(H_i)}{\sum_{k=1}^{m} p(E_1|H_k) \times p(E_2|H_k) \times \ldots \times p(E_n|H_k) \times p(H_k)} \qquad (3.21)$$

How does an expert system compute all posterior probabilities and finally rank potentially true hypotheses?

Let us consider a simple example. Suppose an expert, given three conditionally independent evidences E_1, E_2 and E_3, creates three mutually exclusive and exhaustive hypotheses H_1, H_2 and H_3, and provides prior probabilities for these hypotheses – $p(H_1)$, $p(H_2)$ and $p(H_3)$, respectively. The expert also determines the conditional probabilities of observing each evidence for all possible hypotheses. Table 3.2 illustrates the prior and conditional probabilities provided by the expert.

Assume that we first observe evidence E_3. The expert system computes the posterior probabilities for all hypotheses according to Eq. (3.19):

$$p(H_i|E_3) = \frac{p(E_3|H_i) \times p(H_i)}{\sum_{k=1}^{3} p(E_3|H_k) \times p(H_k)}, \qquad i = 1, 2, 3$$

Thus,

$$p(H_1|E_3) = \frac{0.6 \times 0.40}{0.6 \times 0.40 + 0.7 \times 0.35 + 0.9 \times 0.25} = 0.34$$

$$p(H_2|E_3) = \frac{0.7 \times 0.35}{0.6 \times 0.40 + 0.7 \times 0.35 + 0.9 \times 0.25} = 0.34$$

$$p(H_3|E_3) = \frac{0.9 \times 0.25}{0.6 \times 0.40 + 0.7 \times 0.35 + 0.9 \times 0.25} = 0.32$$

Table 3.2   The prior and conditional probabilities

Probability	Hypothesis		
	$i = 1$	$i = 2$	$i = 3$
$p(H_i)$	0.40	0.35	0.25
$p(E_1|H_i)$	0.3	0.8	0.5
$p(E_2|H_i)$	0.9	0.0	0.7
$p(E_3|H_i)$	0.6	0.7	0.9

As you can see, after evidence E_3 is observed, belief in hypothesis H_1 decreases and becomes equal to belief in hypothesis H_2. Belief in hypothesis H_3 increases and even nearly reaches beliefs in hypotheses H_1 and H_2.

Suppose now that we observe evidence E_1. The posterior probabilities are calculated by Eq. (3.21):

$$p(H_i|E_1E_3) = \frac{p(E_1|H_i) \times p(E_3|H_i) \times p(H_i)}{\sum\limits_{k=1}^{3} p(E_1|H_k) \times p(E_3|H_k) \times p(H_k)}, \qquad i = 1,2,3$$

Hence,

$$p(H_1|E_1E_3) = \frac{0.3 \times 0.6 \times 0.40}{0.3 \times 0.6 \times 0.40 + 0.8 \times 0.7 \times 0.35 + 0.5 \times 0.9 \times 0.25} = 0.19$$

$$p(H_2|E_1E_3) = \frac{0.8 \times 0.7 \times 0.35}{0.3 \times 0.6 \times 0.40 + 0.8 \times 0.7 \times 0.35 + 0.5 \times 0.9 \times 0.25} = 0.52$$

$$p(H_3|E_1E_3) = \frac{0.5 \times 0.9 \times 0.25}{0.3 \times 0.6 \times 0.40 + 0.8 \times 0.7 \times 0.35 + 0.5 \times 0.9 \times 0.25} = 0.29$$

Hypothesis H_2 is now considered as the most likely one, while belief in hypothesis H_1 has decreased dramatically.

After observing evidence E_2 as well, the expert system calculates the final posterior probabilities for all hypotheses:

$$p(H_i|E_1E_2E_3) = \frac{p(E_1|H_i) \times p(E_2|H_i) \times p(E_3|H_i) \times p(H_i)}{\sum\limits_{k=1}^{3} p(E_1|H_k) \times p(E_2|H_k) \times p(E_3|H_k) \times p(H_k)}, \qquad i = 1,2,3$$

Thus,

$$p(H_1|E_1E_2E_3) = \frac{0.3 \times 0.9 \times 0.6 \times 0.40}{0.3 \times 0.9 \times 0.6 \times 0.40 + 0.8 \times 0.0 \times 0.7 \times 0.35 + 0.5 \times 0.7 \times 0.9 \times 0.25}$$
$$= 0.45$$

$$p(H_2|E_1E_2E_3) = \frac{0.8 \times 0.0 \times 0.7 \times 0.35}{0.3 \times 0.9 \times 0.6 \times 0.40 + 0.8 \times 0.0 \times 0.7 \times 0.35 + 0.5 \times 0.7 \times 0.9 \times 0.25}$$
$$= 0$$

$$p(H_3|E_1E_2E_3) = \frac{0.5 \times 0.7 \times 0.9 \times 0.25}{0.3 \times 0.9 \times 0.6 \times 0.40 + 0.8 \times 0.0 \times 0.7 \times 0.35 + 0.5 \times 0.7 \times 0.9 \times 0.25}$$
$$= 0.55$$

Although the initial ranking provided by the expert was H_1, H_2 and H_3, only hypotheses H_1 and H_3 remain under consideration after all evidences (E_1, E_2 and

E_3) were observed. Hypothesis H_2 can now be completely abandoned. Note that hypothesis H_3 is considered more likely than hypothesis H_1.

PROSPECTOR, an expert system for mineral exploration, was the first system to use Bayesian rules of evidence to compute $p(H|E)$ and propagate uncertainties throughout the system (Duda *et al.*, 1979). To help interpret Bayesian reasoning in expert systems, consider a simple example.

3.4   FORECAST: Bayesian accumulation of evidence

Let us develop an expert system for a real problem such as the weather forecast. Our expert system will be required to work out if it is going to rain tomorrow. It will need some real data, which can be obtained from the weather bureau.

Table 3.3 summarises London weather for March 1982. It gives the minimum and maximum temperatures, rainfall and sunshine for each day. If rainfall is zero it is a dry day.

The expert system should give us two possible outcomes – *tomorrow is rain* and *tomorrow is dry* – and provide their likelihood. In other words, the expert system must determine the conditional probabilities of the two hypotheses *tomorrow is rain* and *tomorrow is dry*.

To apply the Bayesian rule (3.18), we should provide the prior probabilities of these hypotheses.

The first thing to do is to write two basic rules that, with the data provided, could predict the weather for tomorrow.

Rule: 1
IF   today is rain
THEN   tomorrow is rain

Rule: 2
IF   today is dry
THEN   tomorrow is dry

Using these rules we will make only ten mistakes – every time a wet day precedes a dry one, or a dry day precedes a wet one. Thus, we can accept the prior probabilities of 0.5 for both hypotheses and rewrite our rules in the following form:

Rule: 1
IF   today is rain {LS 2.5 LN .6}
THEN   tomorrow is rain {prior .5}

Rule: 2
IF   today is dry {LS 1.6 LN .4}
THEN   tomorrow is dry {prior .5}

Table 3.3 London weather summary for March 1982

Day of month	Min. temp. °C	Max. temp. °C	Rainfall mm	Sunshine hours	Actual weather	Weather forecast
1	9.4	11.0	17.5	3.2	Rain	–
2	4.2	12.5	4.1	6.2	Rain	Rain
3	7.6	11.2	7.7	1.1	Rain	Rain
4	5.7	10.5	0.0	4.3	Dry	Rain*
5	3.0	12.0	0.0	9.5	Dry	Dry
6	4.4	9.6	0.0	3.5	Dry	Dry
7	4.8	9.4	4.6	10.1	Rain	Rain
8	1.8	9.2	5.5	7.8	Rain	Rain
9	2.4	10.2	4.8	4.1	Rain	Rain
10	5.5	12.7	4.2	3.8	Rain	Rain
11	3.7	10.9	4.4	9.2	Rain	Rain
12	5.9	10.0	4.8	7.1	Rain	Rain
13	3.0	11.9	0.0	8.3	Dry	Rain*
14	5.4	12.1	4.8	1.8	Rain	Dry*
15	8.8	9.1	8.8	0.0	Rain	Rain
16	2.4	8.4	3.0	3.1	Rain	Rain
17	4.3	10.8	0.0	4.3	Dry	Dry
18	3.4	11.1	4.2	6.6	Rain	Rain
19	4.4	8.4	5.4	0.7	Rain	Rain
20	5.1	7.9	3.0	0.1	Rain	Rain
21	4.4	7.3	0.0	0.0	Dry	Dry
22	5.6	14.0	0.0	6.8	Dry	Dry
23	5.7	14.0	0.0	8.8	Dry	Dry
24	2.9	13.9	0.0	9.5	Dry	Dry
25	5.8	16.4	0.0	10.3	Dry	Dry
26	3.9	17.0	0.0	9.9	Dry	Dry
27	3.8	18.3	0.0	8.3	Dry	Dry
28	5.8	15.4	3.2	7.0	Rain	Dry*
29	6.7	8.8	0.0	4.2	Dry	Dry
30	4.5	9.6	4.8	8.8	Rain	Rain
31	4.6	9.6	3.2	4.2	Rain	Rain

* errors in weather forecast

The value of *LS* represents a measure of the expert belief in hypothesis *H* if evidence *E* is present. It is called **likelihood of sufficiency**. The likelihood of sufficiency is defined as the ratio of $p(E|H)$ over $p(E|\neg H)$

$$LS = \frac{p(E|H)}{p(E|\neg H)} \tag{3.22}$$

In our case, *LS* is the probability of getting rain today if we have rain tomorrow, divided by the probability of getting rain today if there is no rain tomorrow:

$$LS = \frac{p(today\ is\ rain\,|\,tomorrow\ is\ rain)}{p(today\ is\ rain\,|\,tomorrow\ is\ dry)}$$

LN, as you may already have guessed, is a measure of discredit to hypothesis *H* if evidence *E* is missing. *LN* is called **likelihood of necessity** and defined as:

$$LN = \frac{p(\neg E|H)}{p(\neg E|\neg H)} \tag{3.23}$$

In our weather example, *LN* is the probability of not getting rain today if we have rain tomorrow, divided by the probability of not getting rain today if there is no rain tomorrow:

$$LN = \frac{p(today \ is \ dry \,|\, tomorrow \ is \ rain)}{p(today \ is \ dry \,|\, tomorrow \ is \ dry)}$$

Note that *LN* cannot be derived from *LS*. The domain expert must provide both values independently.

How does the domain expert determine values of the likelihood of sufficiency and the likelihood of necessity? Is the expert required to deal with conditional probabilities?

To provide values for *LS* and *LN*, an expert does not need to determine exact values of conditional probabilities. The expert decides likelihood ratios directly. High values of *LS* ($LS \gg 1$) indicate that the rule strongly supports the hypothesis if the evidence is observed, and low values of *LN* ($0 < LN < 1$) suggest that the rule also strongly opposes the hypothesis if the evidence is missing.

Since the conditional probabilities can be easily computed from the likelihood ratios *LS* and *LN*, this approach can use the Bayesian rule to propagate evidence.

Go back now to the London weather. Rule 1 tells us that if it is raining today, there is a high probability of rain tomorrow ($LS = 2.5$). But even if there is no rain today, or in other words today is dry, there is still some chance of having rain tomorrow ($LN = 0.6$).

Rule 2, on the other hand, clarifies the situation with a dry day. If it is dry today, then the probability of a dry day tomorrow is also high ($LS = 1.6$). However, as you can see, the probability of rain tomorrow if it is raining today is higher than the probability of a dry day tomorrow if it is dry today. Why? The values of *LS* and *LN* are usually determined by the domain expert. In our weather example, these values can also be confirmed from the statistical information published by the weather bureau. Rule 2 also determines the chance of a dry day tomorrow even if today we have rain ($LN = 0.4$).

How does the expert system get the overall probability of a dry or wet day tomorrow?

In the rule-based expert system, the prior probability of the consequent, $p(H)$, is converted into the prior odds:

$$O(H) = \frac{p(H)}{1 - p(H)} \tag{3.24}$$

The prior probability is only used when the uncertainty of the consequent is adjusted for the first time. Then in order to obtain the posterior odds, the prior odds are updated by *LS* if the antecedent of the rule (in other words evidence) is true and by *LN* if the antecedent is false:

$$O(H|E) = LS \times O(H) \tag{3.25}$$

and

$$O(H|\neg E) = LN \times O(H) \tag{3.26}$$

The posterior odds are then used to recover the posterior probabilities:

$$p(H|E) = \frac{O(H|E)}{1 + O(H|E)} \tag{3.27}$$

and

$$p(H|\neg E) = \frac{O(H|\neg E)}{1 + O(H|\neg E)} \tag{3.28}$$

Our London weather example shows how this scheme works. Suppose the user indicates that *today is rain*. Rule 1 is fired and the prior probability of *tomorrow is rain* is converted into the prior odds:

$$O(\text{tomorrow is rain}) = \frac{0.5}{1 - 0.5} = 1.0$$

The evidence *today is rain* increases the odds by a factor of 2.5, thereby raising the probability from 0.5 to 0.71:

$$O(\text{tomorrow is rain} \,|\, \text{today is rain}) = 2.5 \times 1.0 = 2.5$$

$$p(\text{tomorrow is rain} \,|\, \text{today is rain}) = \frac{2.5}{1 + 2.5} = 0.71$$

Rule 2 is also fired. The prior probability of *tomorrow is dry* is converted into the prior odds, but the evidence *today is rain* reduces the odds by a factor of 0.4. This, in turn, diminishes the probability of *tomorrow is dry* from 0.5 to 0.29:

$$O(\text{tomorrow is dry}) = \frac{0.5}{1 - 0.5} = 1.0$$

$$O(\text{tomorrow is dry} \,|\, \text{today is rain}) = 0.4 \times 1.0 = 0.4$$

$$p(\text{tomorrow is dry} \,|\, \text{today is rain}) = \frac{0.4}{1 + 0.4} = 0.29$$

Hence if it is raining today there is a 71 per cent chance of it raining and a 29 per cent chance of it being dry tomorrow.

Further suppose that the user input is *today is dry*. By a similar calculation there is a 62 per cent chance of it being dry and a 38 per cent chance of it raining tomorrow.

Now we have examined the basic principles of Bayesian rules of evidence, we can incorporate some new knowledge in our expert system. To do this, we need to determine conditions when the weather actually did change. Analysis of the data provided in Table 3.3 allows us to develop the following knowledge base (the Leonardo expert system shell is used here).

Knowledge base

/* FORECAST: BAYESIAN ACCUMULATION OF EVIDENCE

control bayes

Rule: 1
if   today is rain {LS 2.5 LN .6}
then   tomorrow is rain {prior .5}

Rule: 2
if   today is dry {LS 1.6 LN .4}
then   tomorrow is dry {prior .5}

Rule: 3
if   today is rain
and   rainfall is low {LS 10 LN 1}
then   tomorrow is dry {prior .5}

Rule: 4
if   today is rain
and   rainfall is low
and   temperature is cold {LS 1.5 LN 1}
then   tomorrow is dry {prior .5}

Rule: 5
if   today is dry
and   temperature is warm {LS 2 LN .9}
then   tomorrow is rain {prior .5}

Rule: 6
if   today is dry
and   temperature is warm
and   sky is overcast {LS 5 LN 1}
then   tomorrow is rain {prior .5}

/* The SEEK directive sets up the goal of the rule set

seek tomorrow

Dialogue

Based on the information provided by the user, the expert system determines whether we can expect a dry day tomorrow. The user's answers are indicated by arrows. We assume that rainfall is low if it is less than 4.1 mm, the temperature is cold if the average daily temperature is lower than or equal to 7.0°C, and warm if it is higher than 7.0°C. Finally, sunshine less than 4.6 hours a day stands for overcast.

What is the weather today?
⇒ **rain**

Rule: 1
if   today is rain {LS 2.5 LN .6}
then   tomorrow is rain {prior .5}

$$O(\text{tomorrow is rain}) = \frac{0.5}{1 - 0.5} = 1.0$$

$$O(\text{tomorrow is rain} \,|\, \text{today is rain}) = 2.5 \times 1.0 = 2.5$$

$$p(\text{tomorrow is rain} \,|\, \text{today is rain}) = \frac{2.5}{1 + 2.5} = 0.71$$

tomorrow is rain   {0.71}

Rule: 2
if   today is dry {LS 1.6 LN .4}
then   tomorrow is dry {prior .5}

$$O(\text{tomorrow is dry}) = \frac{0.5}{1 - 0.5} = 1.0$$

$$O(\text{tomorrow is dry} \,|\, \text{today is rain}) = 0.4 \times 1.0 = 0.4$$

$$p(\text{tomorrow is dry} \,|\, \text{today is rain}) = \frac{0.4}{1 + 0.4} = 0.29$$

tomorrow is rain   {0.71}
            dry   {0.29}

What is the rainfall today?
⇒ **low**

Rule: 3
if   today is rain
and   rainfall is low {LS 10 LN 1}
then   tomorrow is dry {prior .5}

$$O(\text{tomorrow is dry}) = \frac{0.29}{1 - 0.29} = 0.41$$

$$O(\text{tomorrow is dry} \,|\, \text{today is rain} \cap \text{rainfall is low}) = 10 \times 0.41 = 4.1$$

$$p(\text{tomorrow is dry} \,|\, \text{today is rain} \cap \text{rainfall is low}) = \frac{4.1}{1+4.1} = 0.80$$

tomorrow is dry   {0.80}
          rain   {0.71}

What is the temperature today?
⇒ **cold**

Rule: 4
if   today is rain
and   rainfall is low
and   temperature is cold {LS 1.5 LN 1}
then   tomorrow is dry {prior .5}

$$O(\text{tomorrow is dry}) = \frac{0.80}{1 - 0.80} = 4$$

$O(\text{tomorrow is dry} \,|\, \text{today is rain} \cap \text{rainfall is low} \cap \text{temperature is cold})$
$= 1.50 \times 4 = 6$

$p(\text{tomorrow is dry} \,|\, \text{today is rain} \cap \text{rainfall is low} \cap \text{temperature is cold})$

$$= \frac{6}{1+6} = 0.86$$

tomorrow is dry   {0.86}
          rain   {0.71}

Rule: 5
if   today is dry
and   temperature is warm {LS 2 LN .9}
then   tomorrow is rain {prior .5}

$$O(\text{tomorrow is rain}) = \frac{0.71}{1 - 0.71} = 2.45$$

$O(\text{tomorrow is rain} \,|\, \text{today is not dry} \cap \text{temperature is not warm}) = 0.9 \times 2.45 = 2.21$

$$p(\text{tomorrow is rain} \,|\, \text{today is not dry} \cap \text{temperature is not warm}) = \frac{2.21}{1 + 2.21} = 0.69$$

tomorrow is dry   {0.86}
          rain   {0.69}

What is the cloud cover today?
⇒ **overcast**

Rule: 6
if   today is dry
and   temperature is warm
and   sky is overcast {LS 5 LN 1}
then   tomorrow is rain {prior .5}

$$O(\text{tomorrow is rain}) = \frac{0.69}{1 - 0.69} = 2.23$$

$$O(\text{tomorrow is rain} \,|\, \text{today is not dry} \cap \text{temperature is not warm} \cap \text{sky is overcast})$$
$$= 1.0 \times 2.23 = 2.23$$

$$p(\text{tomorrow is rain} \,|\, \text{today is not dry} \cap \text{temperature is not warm} \cap \text{sky is overcast})$$
$$= \frac{2.23}{1 + 2.23} = 0.69$$

tomorrow is dry   {0.86}
            rain   {0.69}

This means that we have two potentially true hypotheses, *tomorrow is dry* and *tomorrow is rain*, but the likelihood of the first one is higher.

From Table 3.3 you can see that our expert system made only four mistakes. This is an 86 per cent success rate, which compares well with the results provided in Naylor (1987) for the same case of the London weather.

3.5   Bias of the Bayesian method

The framework for Bayesian reasoning requires probability values as primary inputs. The assessment of these values usually involves human judgement. However, psychological research shows that humans either cannot elicit probability values consistent with the Bayesian rules or do it badly (Burns and Pearl, 1981; Tversky and Kahneman, 1982). This suggests that the conditional probabilities may be inconsistent with the prior probabilities given by the expert. Consider, for example, a car that does not start and makes odd noises when you press the starter. The conditional probability of the starter being faulty if the car makes odd noises may be expressed as:

IF   the symptom is 'odd noises'
THEN   the starter is bad {with probability 0.7}

Apparently the conditional probability that the starter is not bad if the car makes odd noises is:

$$p(\text{starter is not bad} \,|\, \text{odd noises}) = p(\text{starter is good} \,|\, \text{odd noises}) = 1 - 0.7 = 0.3$$

Therefore, we can obtain a companion rule that states

IF   the symptom is 'odd noises'
THEN   the starter is good {with probability 0.3}

Domain experts do not deal easily with conditional probabilities and quite often deny the very existence of the **hidden implicit probability** (0.3 in our example).

In our case, we would use available statistical information and empirical studies to derive the following two rules:

IF   the starter is bad
THEN   the symptom is 'odd noises' {with probability 0.85}

IF   the starter is bad
THEN   the symptom is not 'odd noises' {with probability 0.15}

To use the Bayesian rule, we still need the prior probability, the probability that the starter is bad if the car does not start. Here we need an expert judgement. Suppose, the expert supplies us the value of 5 per cent. Now we can apply the Bayesian rule (3.18) to obtain

$$p(starter\ is\ bad\,|\,odd\ noises) = \frac{0.85 \times 0.05}{0.85 \times 0.05 + 0.15 \times 0.95} = 0.23$$

The number obtained is significantly lower than the expert's estimate of 0.7 given at the beginning of this section.

Why this inconsistency? Did the expert make a mistake?

The most obvious reason for the inconsistency is that the expert made different assumptions when assessing the conditional and prior probabilities. We may attempt to investigate it by working backwards from the posterior probability p(starter is bad | odd noises) to the prior probability p(starter is bad). In our case, we can assume that

$$p(starter\ is\ good) = 1 - p(starter\ is\ bad)$$

From Eq. (3.18) we obtain:

$$p(H) = \frac{p(H|E) \times p(E|\neg H)}{p(H|E) \times p(E|\neg H) + p(E|H)[1 - p(H|E)]}$$

where:

$p(H) = p$(starter is bad);

$p(H|E) = p$(starter is bad | odd noises);

$p(E|H) = p$(odd noises | starter is bad);

$p(E|\neg H) = p$(odd noises | starter is good).

If we now take the value of 0.7, p(starter is bad|odd noises), provided by the expert as the correct one, the prior probability p(starter is bad) would have

to be:

$$p(H) = \frac{0.7 \times 0.15}{0.7 \times 0.15 + 0.85 \times (1 - 0.7)} = 0.29$$

This value is almost six times larger than the figure of 5 per cent provided by the expert. Thus the expert indeed uses quite different estimates of the prior and conditional probabilities.

In fact, the prior probabilities also provided by the expert are likely to be inconsistent with the likelihood of sufficiency, *LS*, and the likelihood of necessity, *LN*. Several methods are proposed to handle this problem (Duda *et al.*, 1976). The most popular technique, first applied in PROSPECTOR, is the use of a piecewise linear interpolation model (Duda *et al.*, 1979).

However, to use the subjective Bayesian approach, we must satisfy many assumptions, including the conditional independence of evidence under both a hypothesis and its negation. As these assumptions are rarely satisfied in real world problems, only a few systems have been built based on Bayesian reasoning. The best known one is PROSPECTOR, an expert system for mineral exploration (Duda *et al.*, 1979).

3.6 Certainty factors theory and evidential reasoning

Certainty factors theory is a popular alternative to Bayesian reasoning. The basic principles of this theory were first introduced in MYCIN, an expert system for the diagnosis and therapy of blood infections and meningitis (Shortliffe and Buchanan, 1975). The developers of MYCIN found that medical experts expressed the strength of their belief in terms that were neither logical nor mathematically consistent. In addition, reliable statistical data about the problem domain was not available. Therefore, the MYCIN team was unable to use a classical probability approach. Instead they decided to introduce a **certainty factor** (*cf*), a number to measure the expert's belief. The maximum value of the certainty factor was $+1.0$ (definitely true) and the minimum -1.0 (definitely false). A positive value represented a degree of belief and a negative a degree of disbelief. For example, if the expert stated that some evidence was almost certainly true, a *cf* value of 0.8 would be assigned to this evidence. Table 3.4 shows some uncertain terms interpreted in MYCIN (Durkin, 1994).

In expert systems with certainty factors, the knowledge base consists of a set of rules that have the following syntax:

IF   <evidence>
THEN   <hypothesis> {*cf*}

where *cf* represents belief in hypothesis *H* given that evidence *E* has occurred.

The certainty factors theory is based on two functions: measure of belief $MB(H,E)$, and measure of disbelief $MD(H,E)$ (Shortliffe and Buchanan, 1975).

Table 3.4 Uncertain terms and their interpretation

Term	Certainty factor
Definitely not	−1.0
Almost certainly not	−0.8
Probably not	−0.6
Maybe not	−0.4
Unknown	−0.2 to +0.2
Maybe	+0.4
Probably	+0.6
Almost certainly	+0.8
Definitely	+1.0

These functions indicate, respectively, the degree to which belief in hypothesis H would be increased if evidence E were observed, and the degree to which disbelief in hypothesis H would be increased by observing the same evidence E.

The measure of belief and disbelief can be defined in terms of prior and conditional probabilities as follows (Ng and Abramson, 1990):

$$MB(H,E) = \begin{cases} 1 & \text{if } p(H) = 1 \\ \dfrac{max\,[p(H|E), p(H)] - p(H)}{max\,[1,0] - p(H)} & \text{otherwise} \end{cases} \quad (3.29)$$

$$MD(H,E) = \begin{cases} 1 & \text{if } p(H) = 0 \\ \dfrac{min\,[p(H|E), p(H)] - p(H)}{min\,[1,0] - p(H)} & \text{otherwise} \end{cases} \quad (3.30)$$

where:

$p(H)$ is the prior probability of hypothesis H being true;

$p(H|E)$ is the probability that hypothesis H is true given evidence E.

The values of $MB(H,E)$ and $MD(H,E)$ range between 0 and 1. The strength of belief or disbelief in hypothesis H depends on the kind of evidence E observed. Some facts may increase the strength of belief, but some increase the strength of disbelief.

How can we determine the total strength of belief or disbelief in a hypothesis?

To combine them into one number, the certainty factor, the following equation is used:

$$cf = \frac{MB(H,E) - MD(H,E)}{1 - min\,[MB(H,E), MD(H,E)]} \quad (3.31)$$

Thus *cf*, which can range in MYCIN from -1 to $+1$, indicates the total belief in hypothesis *H*.

The MYCIN approach can be illustrated through an example. Consider a simple rule:

IF   *A* is *X*
THEN   *B* is *Y*

Quite often, an expert may not be absolutely certain that this rule holds. Also suppose it has been observed that in some cases, even when the IF part of the rule is satisfied and object *A* takes on value *X*, object *B* can acquire some different value *Z*. In other words, we have here the uncertainty of a quasi-statistical kind.

The expert usually can associate a certainty factor with each possible value *B* given that *A* has value *X*. Thus our rule might look as follows:

IF   *A* is *X*
THEN   *B* is *Y* {*cf* 0.7};
           B is *Z* {*cf* 0.2}

What does it mean? Where is the other 10 per cent?
It means that, given *A* has received value *X*, *B* will be *Y* 70 per cent and *Z* 20 per cent of the time. The other 10 per cent of the time it could be anything. In such a way the expert might reserve a possibility that object *B* can take not only two known values, *Y* and *Z*, but also some other value that has not yet been observed. Note that we assign multiple values to object *B*.

The certainty factor assigned by a rule is then propagated through the reasoning chain. Propagation of the certainty factor involves establishing the net certainty of the rule consequent when the evidence in the rule antecedent is uncertain. The net certainty for a single antecedent rule, $cf(H,E)$, can be easily calculated by multiplying the certainty factor of the antecedent, $cf(E)$, with the rule certainty factor, *cf*

$$cf(H,E) = cf(E) \times cf \tag{3.32}$$

For example,

IF   the sky is clear
THEN   the forecast is sunny {*cf* 0.8}

and the current certainty factor of *sky is clear* is 0.5, then

$$cf(H,E) = 0.5 \times 0.8 = 0.4$$

This result, according to Table 3.4, would read as 'It may be sunny'.

How does an expert system establish the certainty factor for rules with multiple antecedents?

For conjunctive rules such as

> IF   <evidence E_1 >
> AND   <evidence E_2 >
> .
> .
> .
> AND   <evidence E_n >
> THEN   <hypothesis H > {cf}

the net certainty of the consequent, or in other words the certainty of hypothesis H, is established as follows:

$$cf(H, E_1 \cap E_2 \cap \ldots \cap E_n) = min\,[cf(E_1), cf(E_2), \ldots, cf(E_n)] \times cf \qquad (3.33)$$

For example,

> IF   sky is clear
> AND   the forecast is sunny
> THEN   the action is 'wear sunglasses' {cf 0.8}

and the certainty of *sky is clear* is 0.9 and the certainty of the *forecast is sunny* is 0.7, then

$$cf(H, E_1 \cap E_2) = min\,[0.9, 0.7] \times 0.8 = 0.7 \times 0.8 = 0.56$$

According to Table 3.4, this conclusion might be interpreted as 'Probably it would be a good idea to wear sunglasses today'.

For disjunctive rules such as

> IF   <evidence E_1 >
> OR   <evidence E_2 >
> .
> .
> .
> OR   <evidence E_n >
> THEN   <hypothesis H > {cf}

the certainty of hypothesis H, is determined as follows:

$$cf(H, E_1 \cup E_2 \cup \ldots \cup E_n) = max\,[cf(E_1), cf(E_2), \ldots, cf(E_n)] \times cf \qquad (3.34)$$

For example,

> IF   sky is overcast
> OR   the forecast is rain
> THEN the action is 'take an umbrella' {*cf* 0.9}

and the certainty of *sky is overcast* is 0.6 and the certainty of the *forecast is rain* is 0.8, then

$$cf(H, E_1 \cup E_2) = max\,[0.6, 0.8] \times 0.9 = 0.8 \times 0.9 = 0.72,$$

which can be interpreted as 'Almost certainly an umbrella should be taken today'.

Sometimes two or even more rules can affect the same hypothesis. How does an expert system cope with such situations?

When the same consequent is obtained as a result of the execution of two or more rules, the individual certainty factors of these rules must be merged to give a combined certainty factor for a hypothesis. Suppose the knowledge base consists of the following rules:

> Rule 1:   IF   *A* is *X*
>   THEN   *C* is *Z* {*cf* 0.8}
>
> Rule 2:   IF   *B* is *Y*
>   THEN   *C* is *Z* {*cf* 0.6}

What certainty should be assigned to object *C* having value *Z* if both Rule 1 and Rule 2 are fired? Our common sense suggests that, if we have two pieces of evidence (*A* is *X* and *B* is *Y*) from different sources (Rule 1 and Rule 2) supporting the same hypothesis (*C* is *Z*), then the confidence in this hypothesis should increase and become stronger than if only one piece of evidence had been obtained.

To calculate a combined certainty factor we can use the following equation (Durkin, 1994):

$$cf(cf_1, cf_2) = \begin{cases} cf_1 + cf_2 \times (1 - cf_1) & \text{if } cf_1 > 0 \text{ and } cf_2 > 0 \\ \dfrac{cf_1 + cf_2}{1 - min\,[|cf_1|, |cf_2|]} & \text{if } cf_1 < 0 \text{ or } cf_2 < 0 \\ cf_1 + cf_2 \times (1 + cf_1) & \text{if } cf_1 < 0 \text{ and } cf_2 < 0 \end{cases} \qquad (3.35)$$

where:

    cf_1 is the confidence in hypothesis *H* established by Rule 1;

    cf_2 is the confidence in hypothesis *H* established by Rule 2;

    $|cf_1|$ and $|cf_2|$ are absolute magnitudes of cf_1 and cf_2, respectively.

Thus, if we assume that

$$cf(E_1) = cf(E_2) = 1.0$$

then from Eq. (3.32) we get:

$$cf_1(H, E_1) = cf(E_1) \times cf_1 = 1.0 \times 0.8 = 0.8$$
$$cf_2(H, E_2) = cf(E_2) \times cf_2 = 1.0 \times 0.6 = 0.6$$

and from Eq. (3.35) we obtain:

$$cf(cf_1, cf_2) = cf_1(H, E_1) + cf_2(H, E_2) \times [1 - cf_1(H, E_1)]$$
$$= 0.8 + 0.6 \times (1 - 0.8) = 0.92$$

This example shows an incremental increase of belief in a hypothesis and also confirms our expectations.

Consider now a case when rule certainty factors have the opposite signs. Suppose that

$$cf(E_1) = 1 \text{ and } cf(E_2) = -1.0,$$

then

$$cf_1(H, E_1) = 1.0 \times 0.8 = 0.8$$
$$cf_2(H, E_2) = -1.0 \times 0.6 = -0.6$$

and from Eq. (3.35) we obtain:

$$cf(cf_1, cf_2) = \frac{cf_1(H, E_1) + cf_2(H, E_2)}{1 - min\left[|cf_1(H, E_1)|, |cf_2(H, E_2)|\right]} = \frac{0.8 - 0.6}{1 - min\left[0.8, 0.6\right]} = 0.5$$

This example shows how a combined certainty factor, or in other words net belief, is obtained when one rule, Rule 1, confirms a hypothesis but another, Rule 2, discounts it.

Let us consider now the case when rule certainty factors have negative signs. Suppose that:

$$cf(E_1) = cf(E_2) = -1.0,$$

then

$$cf_1(H, E_1) = -1.0 \times 0.8 = -0.8$$
$$cf_2(H, E_2) = -1.0 \times 0.6 = -0.6$$

and from Eq. (3.35) we obtain:

$$cf(cf_1, cf_2) = cf_1(H, E_1) + cf_2(H, E_2) \times [1 + cf_1(H, E_1)]$$
$$= -0.8 - 0.6 \times (1 - 0.8) = -0.92$$

This example represents an incremental increase of disbelief in a hypothesis.

The certainty factors theory provides a **practical** alternative to Bayesian reasoning. The heuristic manner of combining certainty factors is different from the manner in which they would be combined if they were probabilities. The certainty theory is not 'mathematically pure' but does mimic the thinking process of a human expert.

To illustrate the evidential reasoning and the method of certainty factors propagation through a set of rules, consider again the FORECAST expert system developed in section 3.4.

3.7 FORECAST: an application of certainty factors

The expert system is required to predict whether it will rain tomorrow, or in other words to establish certainty factors for the multivalued object *tomorrow*. To simplify our task, we use the same set of rules as in section 3.4.

Knowledge base

/* FORECAST: AN APPLICATION OF CERTAINTY FACTORS

control cf

control 'threshold 0.01'

Rule: 1
if   today is rain
then   tomorrow is rain {cf 0.5}

Rule: 2
if   today is dry
then   tomorrow is dry {cf 0.5}

Rule: 3
if   today is rain
and   rainfall is low
then   tomorrow is dry {cf 0.6}

Rule: 4
if   today is rain
and   rainfall is low
and   temperature is cold
then   tomorrow is dry {cf 0.7}

Rule: 5
if   today is dry
and   temperature is warm
then   tomorrow is rain {cf 0.65}

Rule: 6
if   today is dry
and   temperature is warm
and   sky is overcast
then   tomorrow is rain {cf 0.55}

seek tomorrow

Dialogue

To apply an inexact reasoning technique based on certainty factors, the expert system prompts the user to input not only the object value but also the certainty associated with this value. For example, using the Leonardo scale from 0 to 1.0, the following dialogue might be obtained:

What is the weather today?
⇒ **rain**

Rule: 1
if   today is rain
then   tomorrow is rain {cf 0.5}

cf (tomorrow is rain, today is rain) $= cf$ (today is rain) $\times cf = 1.0 \times 0.5 = 0.5$

tomorrow is rain   {0.50}

What is the rainfall today?
⇒ **low**

To what degree do you believe the rainfall is low? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.8**

Rule: 3
if   today is rain
and   rainfall is low
then   tomorrow is dry {cf 0.6}

cf (tomorrow is dry, today is rain $\cap$ rainfall is low)
$= min$ [cf (today is rain), cf (rainfall is low)] $\times cf = min$ [1, 0.8] $\times 0.6 = 0.48$

tomorrow is rain   {0.50}
            dry   {0.48}

What is the temperature today?
⇒ **cold**

To what degree do you believe the temperature is cold? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.9**

Rule: 4
if   today is rain
and   rainfall is low
and   temperature is cold
then   tomorrow is dry {cf 0.7}

cf (tomorrow is dry, today is rain ∩ rainfall is low ∩ temperature is cold)
$= min\,[cf$ (today is rain), cf (rainfall is low), cf (temperature is cold)$] \times cf$
$= min\,[1, 0.8, 0.9] \times 0.7 = 0.56$

tomorrow is dry   {0.56}
            rain   {0.50}

$cf\,(cf_{\text{Rule:3}}, cf_{\text{Rule:4}}) = cf_{\text{Rule:3}} + cf_{\text{Rule:4}} \times (1 - cf_{\text{Rule:3}})$
$= 0.48 + 0.56 \times (1 - 0.48) = 0.77$

tomorrow is dry   {0.77}
            rain   {0.50}

Now we would conclude that the probability of having a dry day tomorrow is almost certain; however we also may expect some rain!

3.8 Comparison of Bayesian reasoning and certainty factors

In the previous sections, we outlined the two most popular techniques for uncertainty management in expert systems. Now we will compare these techniques and determine the kinds of problems that can make effective use of either Bayesian reasoning or certainty factors.

Probability theory is the oldest and best-established technique to deal with inexact knowledge and random data. It works well in such areas as forecasting and planning, where statistical data is usually available and accurate probability statements can be made.

An expert system that applied the Bayesian technique, PROSPECTOR, was developed to aid exploration geologists in their search for ore deposits. It was very successful; for example using geological, geophysical and geochemical data, PROSPECTOR predicted the existence of molybdenum near Mount Tolman in Washington State (Campbell *et al.*, 1982). But the PROSPECTOR team could rely on valid data about known mineral deposits and reliable statistical information. The probabilities of each event were defined as well. The PROSPECTOR

team also could assume the conditional independence of evidence, a constraint that must be satisfied in order to apply the Bayesian approach.

However, in many areas of possible applications of expert systems, reliable statistical information is not available or we cannot assume the conditional independence of evidence. As a result, many researchers have found the Bayesian method unsuitable for their work. For example, Shortliffe and Buchanan could not use a classical probability approach in MYCIN because the medical field often could not provide the required data (Shortliffe and Buchanan, 1975). This dissatisfaction motivated the development of the certainty factors theory.

Although the certainty factors approach lacks the mathematical correctness of the probability theory, it appears to outperform subjective Bayesian reasoning in such areas as diagnostics, particularly in medicine. In diagnostic expert systems like MYCIN, rules and certainty factors come from the expert's knowledge and his or her intuitive judgements. Certainty factors are used in cases where the probabilities are not known or are too difficult or expensive to obtain. The evidential reasoning mechanism can manage incrementally acquired evidence, the conjunction and disjunction of hypotheses, as well as evidences with different degrees of belief. Besides, the certainty factors approach provides better explanations of the control flow through a rule-based expert system.

The Bayesian approach and certainty factors are different from one another, but they share a common problem: finding an expert able to quantify personal, subjective and qualitative information. Humans are easily biased, and therefore the choice of an uncertainty management technique strongly depends on the existing domain expert.

The Bayesian method is likely to be the most appropriate if reliable statistical data exists, the knowledge engineer is able to lead, and the expert is available for serious decision-analytical conversations. In the absence of any of the specified conditions, the Bayesian approach might be too arbitrary and even biased to produce meaningful results. It should also be mentioned that the Bayesian belief propagation is of exponential complexity, and thus is impractical for large knowledge bases.

The certainty factors technique, despite the lack of a formal foundation, offers a simple approach for dealing with uncertainties in expert systems and delivers results acceptable in many applications.

3.9 Summary

In this chapter, we presented two uncertainty management techniques used in expert systems: Bayesian reasoning and certainty factors. We identified the main sources of uncertain knowledge and briefly reviewed probability theory. We considered the Bayesian method of accumulating evidence and developed a simple expert system based on the Bayesian approach. Then we examined the certainty factors theory (a popular alternative to Bayesian reasoning) and developed an expert system based on evidential reasoning. Finally, we compared

Bayesian reasoning and certainty factors, and determined appropriate areas for their applications.

The most important lessons learned in this chapter are:

- Uncertainty is the lack of exact knowledge that would allow us to reach a perfectly reliable conclusion. The main sources of uncertain knowledge in expert systems are: weak implications, imprecise language, missing data and combining the views of different experts.

- Probability theory provides an exact, mathematically correct, approach to uncertainty management in expert systems. The Bayesian rule permits us to determine the probability of a hypothesis given that some evidence has been observed.

- PROSPECTOR, an expert system for mineral exploration, was the first successful system to employ Bayesian rules of evidence to propagate uncertainties throughout the system.

- In the Bayesian approach, an expert is required to provide the prior probability of hypothesis H and values for the likelihood of sufficiency, LS, to measure belief in the hypothesis if evidence E is present, and the likelihood of necessity, LN, to measure disbelief in hypothesis H if the same evidence is missing. The Bayesian method uses rules of the following form:

    IF E is true {LS, LN}
    THEN H is true {*prior probability*}

- To employ the Bayesian approach, we must satisfy the conditional independence of evidence. We also should have reliable statistical data and define the prior probabilities for each hypothesis. As these requirements are rarely satisfied in real-world problems, only a few systems have been built based on Bayesian reasoning.

- Certainty factors theory is a popular alternative to Bayesian reasoning. The basic principles of this theory were introduced in MYCIN, a diagnostic medical expert system.

- Certainty factors theory provides a judgemental approach to uncertainty management in expert systems. An expert is required to provide a certainty factor, *cf*, to represent the level of belief in hypothesis H given that evidence E has been observed. The certainty factors method uses rules of the following form:

    IF E is true
    THEN H is true {*cf*}

- Certainty factors are used if the probabilities are not known or cannot be easily obtained. Certainty theory can manage incrementally acquired evidence, the conjunction and disjunction of hypotheses, as well as evidences with different degrees of belief.

- Both Bayesian reasoning and certainty theory share a common problem: finding an expert able to quantify subjective and qualitative information.

Questions for review

1 What is uncertainty? When can knowledge be inexact and data incomplete or inconsistent? Give an example of inexact knowledge.

2 What is probability? Describe mathematically the conditional probability of event *A* occurring given that event *B* has occurred. What is the Bayesian rule?

3 What is Bayesian reasoning? How does an expert system rank potentially true hypotheses? Give an example.

4 Why was the PROSPECTOR team able to apply the Bayesian approach as an uncertainty management technique? What requirements must be satisfied before Bayesian reasoning will be effective?

5 What are the likelihood of sufficiency and likelihood of necessity? How does an expert determine values for *LS* and *LN*?

6 What is a prior probability? Give an example of the rule representation in the expert system based on Bayesian reasoning.

7 How does a rule-based expert system propagate uncertainties using the Bayesian approach?

8 Why may conditional probabilities be inconsistent with the prior probabilities provided by the expert? Give an example of such an inconsistency.

9 Why is the certainty factors theory considered as a **practical** alternative to Bayesian reasoning? What are the measure of belief and the measure of disbelief? Define a certainty factor.

10 How does an expert system establish the net certainty for conjunctive and disjunctive rules? Give an example for each case.

11 How does an expert system combine certainty factors of two or more rules affecting the same hypothesis? Give an example.

12 Compare Bayesian reasoning and certainty factors. Which applications are most suitable for Bayesian reasoning and which for certainty factors? Why? What is a common problem in both methods?

References

Bhatnagar, R.K. and Kanal, L.N. (1986). Handling uncertain information: A review of numeric and non-numeric methods, *Uncertainty in AI*, L.N. Kanal and J.F. Lemmer, eds, Elsevier North-Holland, New York, pp. 3–26.

Bonissone, P.P. and Tong, R.M. (1985). Reasoning with uncertainty in expert systems, *International Journal on Man–Machine Studies*, 22(3), 241–250.

Burns, M. and Pearl, J. (1981). Causal and diagnostic inferences: A comparison of validity, *Organizational Behaviour and Human Performance*, 28, 379–394.

Campbell, A.N., Hollister, V.F., Duda, R.O. and Hart, P.E. (1982). Recognition of a hidden mineral deposit by an artificial intelligence program, *Science*, 217(3), 927–929.

Good, I.J. (1959). Kinds of probability, *Science*, 129(3347), 443–447.

Duda, R.O., Hart, P.E. and Nilsson, N.L. (1976). Subjective Bayesian methods for a rule-based inference system, *Proceedings of the National Computer Conference (AFIPS)*, vol. 45, pp. 1075–1082.

Duda, R.O., Gaschnig, J. and Hart, P.E. (1979). Model design in the PROSPECTOR consultant system for mineral exploration, *Expert Systems in the Microelectronic Age*, D. Michie, ed., Edinburgh University Press, Edinburgh, Scotland, pp. 153–167.

Durkin, J. (1994). *Expert Systems Design and Development*. Prentice Hall, Englewood Cliffs, NJ.

Feller, W. (1957). *An Introduction to Probability Theory and Its Applications*. John Wiley, New York.

Fine, T.L. (1973). *Theories of Probability: An Examination of Foundations*. Academic Press, New York.

Firebaugh, M.W. (1989). *Artificial Intelligence: A Knowledge-based Approach*. PWS-KENT Publishing Company, Boston.

Hakel, M.D. (1968). How often is often? *American Psychologist*, no. 23, 533–534.

Naylor, C. (1987). *Build Your Own Expert System*. Sigma Press, England.

Ng, K.-C. and Abramson, B. (1990). Uncertainty management in expert systems, *IEEE Expert*, 5(2), 29–47.

Shortliffe, E.H. and Buchanan, B.G. (1975). A model of inexact reasoning in medicine, *Mathematical Biosciences*, 23(3/4), 351–379.

Simpson, R. (1944). The specific meanings of certain terms indicating differing degrees of frequency, *The Quarterly Journal of Speech*, no. 30, 328–330.

Stephanou, H.E. and Sage, A.P. (1987). Perspectives on imperfect information processing, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17(5), 780–798.

Tversky, A. and Kahneman, D. (1982). Causal schemes in judgements under uncertainty, *Judgements Under Uncertainty: Heuristics and Biases*, D. Kahneman, P. Slovic and A. Tversky, eds, Cambridge University Press, New York.

Fuzzy expert systems

4

In which we present fuzzy set theory, consider how to build fuzzy expert systems and illustrate the theory through an example.

4.1 Introduction, or what is fuzzy thinking?

Experts usually rely on **common sense** when they solve problems. They also use vague and ambiguous terms. For example, an expert might say, 'Though the power transformer is **slightly** overloaded, I can keep this load for **a while**'. Other experts have no difficulties with understanding and interpreting this statement because they have the background to hearing problems described like this. However, a knowledge engineer would have difficulties providing a computer with the same level of understanding. How can we represent expert knowledge that uses vague and ambiguous terms in a computer? Can it be done at all?

This chapter attempts to answer these questions by exploring the **fuzzy set theory** (or **fuzzy logic**). We review the philosophical ideas behind fuzzy logic, study its apparatus and then consider how fuzzy logic is used in fuzzy expert systems.

Let us begin with a trivial, but still basic and essential, statement: fuzzy logic is not logic that is fuzzy, but logic that is used to describe fuzziness. Fuzzy logic is the theory of fuzzy sets, sets that calibrate vagueness. Fuzzy logic is based on the idea that all things admit of degrees. Temperature, height, speed, distance, beauty – all come on a sliding scale. The motor is running **really hot**. Tom is a **very tall** guy. Electric cars are **not very fast**. **High**-performance drives require **very rapid** dynamics and **precise** regulation. Hobart is **quite a short** distance from Melbourne. Sydney is a **beautiful** city. Such a sliding scale often makes it impossible to distinguish members of a class from non-members. When does a hill become a mountain?

Boolean or conventional logic uses sharp distinctions. It forces us to draw lines between members of a class and non-members. It makes us draw lines in the sand. For instance, we may say, 'The maximum range of an electric vehicle is short', regarding a range of 300 km or less as short, and a range greater than 300 km as long. By this standard, any electric vehicle that can cover a distance of 301 km (or 300 km and 500 m or even 300 km and 1 m) would be described as

long-range. Similarly, we say Tom is tall because his height is 181 cm. If we drew a line at 180 cm, we would find that David, who is 179 cm, is small. Is David really a small man or have we just drawn an arbitrary line in the sand? Fuzzy logic makes it possible to avoid such absurdities.

Fuzzy logic reflects how people think. It attempts to model our sense of words, our decision making and our common sense. As a result, it is leading to new, more human, intelligent systems.

Fuzzy, or multi-valued logic was introduced in the 1930s by Jan Lukasiewicz, a Polish logician and philosopher (Lukasiewicz, 1930). He studied the mathematical representation of fuzziness based on such terms as **tall**, **old** and **hot**. While classical logic operates with only two values 1 (true) and 0 (false), Lukasiewicz introduced logic that extended the range of truth values to all real numbers in the interval between 0 and 1. He used a number in this interval to represent the **possibility** that a given statement was true or false. For example, the possibility that a man 181 cm tall is really tall might be set to a value of 0.86. It is **likely** that the man is tall. This work led to an inexact reasoning technique often called **possibility theory**.

Later, in 1937, Max Black, a philosopher, published a paper called 'Vagueness: an exercise in logical analysis' (Black, 1937). In this paper, he argued that a continuum implies degrees. Imagine, he said, a line of countless 'chairs'. At one end is a Chippendale. Next to it is a near-Chippendale, in fact indistinguishable from the first item. Succeeding 'chairs' are less and less chair-like, until the line ends with a log. When does a **chair** become a **log**? The concept **chair** does not permit us to draw a clear line distinguishing **chair** from **not-chair**. Max Black also stated that if a continuum is discrete, a number can be allocated to each element. This number will indicate a degree. But the question is degree of what. Black used the number to show the percentage of people who would call an element in a line of 'chairs' a **chair**; in other words, he accepted vagueness as a matter of probability.

However, Black's most important contribution was in the paper's appendix. Here he defined the first simple fuzzy set and outlined the basic ideas of fuzzy set operations.

In 1965 Lotfi Zadeh, Professor and Head of the Electrical Engineering Department at the University of California at Berkeley, published his famous paper 'Fuzzy sets'. In fact, Zadeh rediscovered fuzziness, identified and explored it, and promoted and fought for it.

Zadeh extended the work on possibility theory into a formal system of mathematical logic, and even more importantly, he introduced a new concept for applying natural language terms. This new logic for representing and manipulating fuzzy terms was called fuzzy logic, and Zadeh became the Master of fuzzy logic.

Why fuzzy?

As Zadeh said, the term is concrete, immediate and descriptive; we all know what it means. However, many people in the West were repelled by the word fuzzy, because it is usually used in a negative sense.
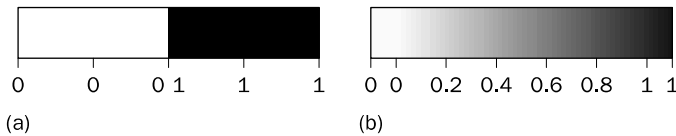
Figure 4.1 Range of logical values in Boolean and fuzzy logic: (a) Boolean logic; (b) multi-valued logic

Why logic?

Fuzziness rests on fuzzy set theory, and fuzzy logic is just a small part of that theory. However, Zadeh used the term fuzzy logic in a broader sense (Zadeh, 1965):

> Fuzzy logic is determined as a set of mathematical principles for knowledge representation based on degrees of membership rather than on crisp membership of classical binary logic.

Unlike two-valued Boolean logic, fuzzy logic is **multi-valued**. It deals with **degrees of membership** and **degrees of truth**. Fuzzy logic uses the continuum of logical values between 0 (completely false) and 1 (completely true). Instead of just black and white, it employs the spectrum of colours, accepting that things can be partly true and partly false at the same time. As can be seen in Figure 4.1, fuzzy logic adds a range of logical values to Boolean logic. Classical binary logic now can be considered as a special case of multi-valued fuzzy logic.

4.2 Fuzzy sets

The concept of a **set** is fundamental to mathematics. However, our own language is the supreme expression of sets. For example, *car* indicates the set of cars. When we say *a car*, we mean one out of the set of cars.

Let X be a classical (**crisp**) set and x an element. Then the element x either belongs to X ($x \in X$) or does not belong to X ($x \notin X$). That is, classical set theory imposes a sharp boundary on this set and gives each member of the set the value of 1, and all members that are not within the set a value of 0. This is known as the **principle of dichotomy**. Let us now dispute this principle.

Consider the following classical paradoxes of logic.

1 Pythagorean School (400 BC):
   Question: Does the Cretan philosopher tell the truth when he asserts that 'All Cretans always lie'?
   Boolean logic: This assertion contains a contradiction.
   Fuzzy logic: The philosopher does and does not tell the truth!

2 Russell's Paradox:
   The barber of a village gives a hair cut only to those who do not cut their hair themselves.

> **Question**: Who cuts the barber's hair?
> **Boolean logic**: This assertion contains a contradiction.
> **Fuzzy logic**: The barber cuts and doesn't cut his own hair!

Crisp set theory is governed by a logic that uses one of only two values: true or false. This logic cannot represent vague concepts, and therefore fails to give the answers on the paradoxes. The basic idea of the fuzzy set theory is that an element belongs to a fuzzy set with a certain degree of membership. Thus, a proposition is not either true or false, but may be partly true (or partly false) to any degree. This degree is usually taken as a real number in the interval [0,1].

The classical example in the fuzzy set theory is *tall men*. The elements of the fuzzy set 'tall men' are all men, but their degrees of membership depend on their height, as shown in Table 4.1. Suppose, for example, Mark at 205 cm tall is given a degree of 1, and Peter at 152 cm is given a degree of 0. All men of intermediate height have intermediate degrees. They are partly tall. Obviously, different people may have different views as to whether a given man should be considered as tall. However, our candidates for *tall men* could have the memberships presented in Table 4.1.

It can be seen that the crisp set asks the question, 'Is the man tall?' and draws a line at, say, 180 cm. *Tall men* are above this height and *not tall men* below. In contrast, the fuzzy set asks, 'How tall is the man?' The answer is the partial membership in the fuzzy set, for example, Tom is 0.82 tall.

A fuzzy set is capable of providing a graceful transition across a boundary, as shown in Figure 4.2.

We might consider a few other sets such as 'very short men', 'short men', 'average men' and 'very tall men'.

In Figure 4.2 the horizontal axis represents the **universe of discourse** – the range of all possible values applicable to a chosen variable. In our case, the variable is the human height. According to this representation, the universe of men's heights consists of all tall men. However, there is often room for

Table 4.1   Degree of membership of 'tall men'

| Name | Height, cm | Degree of membership | |
		Crisp	Fuzzy
Chris	208	1	1.00
Mark	205	1	1.00
John	198	1	0.98
Tom	181	1	0.82
David	179	0	0.78
Mike	172	0	0.24
Bob	167	0	0.15
Steven	158	0	0.06
Bill	155	0	0.01
Peter	152	0	0.00
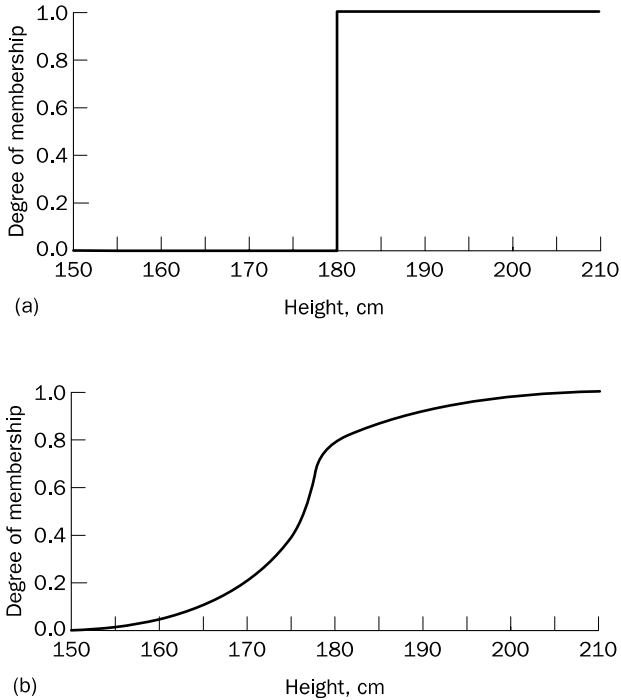
Figure 4.2   Crisp (a) and fuzzy (b) sets of 'tall men'

discretion, since the context of the universe may vary. For example, the set of 'tall men' might be part of the universe of human heights or mammal heights, or even all animal heights.

The vertical axis in Figure 4.2 represents the membership value of the fuzzy set. In our case, the fuzzy set of 'tall men' maps height values into corresponding membership values. As can be seen from Figure 4.2, David who is 179 cm tall, which is just 2 cm less than Tom, no longer suddenly becomes a *not tall* (or *short*) man (as he would in crisp sets). Now David and other men are gradually removed from the set of 'tall men' according to the decrease of their heights.

What is a fuzzy set?

A fuzzy set can be simply defined as a set with fuzzy boundaries.

Let X be the universe of discourse and its elements be denoted as x. In classical set theory, crisp set A of X is defined as function $f_A(x)$ called the **characteristic function** of A

$$f_A(x) : X \rightarrow 0, 1, \tag{4.1}$$

where

$$f_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}$$

This set maps universe X to a set of two elements. For any element x of universe X, characteristic function $f_A(x)$ is equal to 1 if x is an element of set A, and is equal to 0 if x is not an element of A.

In the fuzzy theory, fuzzy set A of universe X is defined by function $\mu_A(x)$ called the **membership function** of set A

$$\mu_A(x) : X \to [0, 1], \tag{4.2}$$

where

$\mu_A(x) = 1$ if x is totally in A;

$\mu_A(x) = 0$ if x is not in A;

$0 < \mu_A(x) < 1$ if x is partly in A.

This set allows a continuum of possible choices. For any element x of universe X, membership function $\mu_A(x)$ equals the degree to which x is an element of set A. This degree, a value between 0 and 1, represents the **degree of membership**, also called **membership value**, of element x in set A.

How to represent a fuzzy set in a computer?

The membership function must be determined first. A number of methods learned from knowledge acquisition can be applied here. For example, one of the most practical approaches for forming fuzzy sets relies on the knowledge of a single expert. The expert is asked for his or her opinion whether various elements belong to a given set. Another useful approach is to acquire knowledge from multiple experts. A new technique to form fuzzy sets was recently introduced. It is based on artificial neural networks, which learn available system operation data and then derive the fuzzy sets automatically.

Now we return to our 'tall men' example. After acquiring the knowledge for men's heights, we could produce a fuzzy set of *tall men*. In a similar manner, we could obtain fuzzy sets of *short* and *average* men. These sets are shown in Figure 4.3, along with crisp sets. The universe of discourse – the men's heights – consists of three sets: *short*, *average* and *tall men*. In fuzzy logic, as you can see, a man who is 184 cm tall is a member of the *average men* set with a degree of membership of 0.1, and at the same time, he is also a member of the *tall men* set with a degree of 0.4. This means that a man of 184 cm tall has partial membership in multiple sets.

Now assume that universe of discourse X, also called the **reference super set**, is a crisp set containing five elements $X = \{x_1, x_2, x_3, x_4, x_5\}$. Let A be a crisp subset of X and assume that A consists of only two elements, $A = \{x_2, x_3\}$. Subset A can now be described by $A = \{(x_1, 0), (x_2, 1), (x_3, 1), (x_4, 0), (x_5, 0)\}$, i.e. as a set of pairs $\{(x_i, \mu_A(x_i))\}$, where $\mu_A(x_i)$ is the membership function of element x_i in the subset A.

The question is whether $\mu_A(x)$ can take only two values, either 0 or 1, or any value between 0 and 1. It was also the basic question in fuzzy sets examined by Lotfi Zadeh in 1965 (Zadeh, 1965).
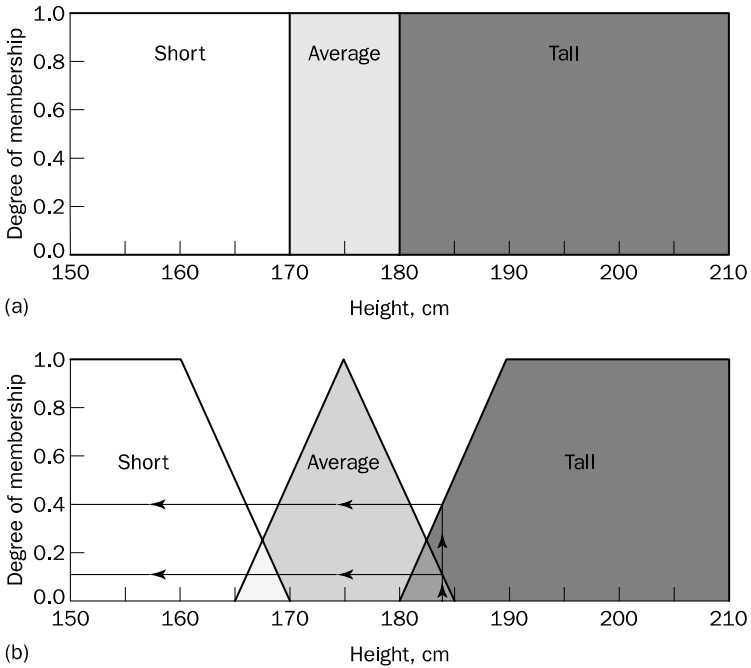
Figure 4.3   Crisp (a) and fuzzy (b) sets of short, average and tall men

If X is the reference super set and A is a subset of X, then A is said to be a fuzzy subset of X if, and only if,

$$A = \{(x, \mu_A(x))\} \qquad x \in X, \mu_A(x) : X \rightarrow [0, 1] \tag{4.3}$$

In a special case, when $X \rightarrow \{0, 1\}$ is used instead of $X \rightarrow [0, 1]$, the fuzzy subset A becomes the crisp subset A.

Fuzzy and crisp sets can be also presented as shown in Figure 4.4.
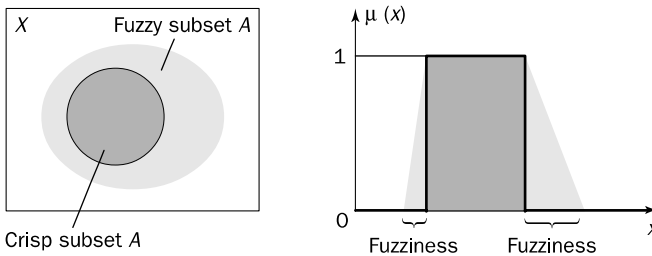
Figure 4.4   Representation of crisp and fuzzy subset of X

Fuzzy subset A of the finite reference super set X can be expressed as,

$$A = \{(x_1, \mu_A(x_1))\}, \{(x_2, \mu_A(x_2))\}, \ldots, \{(x_n, \mu_A(x_n))\} \qquad (4.4)$$

However, it is more convenient to represent A as,

$$A = \{\mu_A(x_1)/x_1\}, \{\mu_A(x_2)/x_2\}, \ldots, \{\mu_A(x_n)/x_n\}, \qquad (4.5)$$

where the separating symbol / is used to associate the membership value with its coordinate on the horizontal axis.

To represent a continuous fuzzy set in a computer, we need to express it as a function and then to map the elements of the set to their degree of membership. Typical functions that can be used are sigmoid, gaussian and pi. These functions can represent the real data in fuzzy sets, but they also increase the time of computation. Therefore, in practice, most applications use **linear fit functions** similar to those shown in Figure 4.3. For example, the fuzzy set of *tall men* in Figure 4.3 can be represented as a **fit-vector**,

tall men = (0/180, 0.5/185, 1/190) or
tall men = (0/180, 1/190)

Fuzzy sets of *short* and *average men* can be also represented in a similar manner,

short men = (1/160, 0.5/165, 0/170) or
short men = (1/160, 0/170)

average men = (0/165, 1/175, 0/185)

4.3 Linguistic variables and hedges

At the root of fuzzy set theory lies the idea of linguistic variables. A linguistic variable is a fuzzy variable. For example, the statement 'John is tall' implies that the linguistic variable *John* takes the linguistic value *tall*. In fuzzy expert systems, linguistic variables are used in fuzzy rules. For example,

IF   wind is strong
THEN   sailing is good

IF   project_duration is long
THEN   completion_risk is high

IF   speed is slow
THEN   stopping_distance is short

The range of possible values of a linguistic variable represents the universe of discourse of that variable. For example, the universe of discourse of the linguistic

variable *speed* might have the range between 0 and 220 km per hour and may include such fuzzy subsets as *very slow*, *slow*, *medium*, *fast*, and *very fast*. Each fuzzy subset also represents a linguistic value of the corresponding linguistic variable.

A linguistic variable carries with it the concept of fuzzy set qualifiers, called **hedges**. Hedges are terms that modify the shape of fuzzy sets. They include adverbs such as *very, somewhat, quite, more or less* and *slightly*. Hedges can modify verbs, adjectives, adverbs or even whole sentences. They are used as

- All-purpose modifiers, such as *very, quite* or *extremely*.
- Truth-values, such as *quite true* or *mostly false*.
- Probabilities, such as *likely* or *not very likely*.
- Quantifiers, such as *most, several* or *few*.
- Possibilities, such as *almost impossible* or *quite possible*.

Hedges act as operations themselves. For instance, *very* performs concentration and creates a new subset. From the set of *tall men*, it derives the subset of *very tall men*. *Extremely* serves the same purpose to a greater extent.

An operation opposite to concentration is dilation. It expands the set. *More or less* performs dilation; for example, the set of *more or less tall men* is broader than the set of *tall men*.

Hedges are useful as operations, but they can also break down continuums into fuzzy intervals. For example, the following hedges could be used to describe temperature: *very cold, moderately cold, slightly cold, neutral, slightly hot, moderately hot* and *very hot*. Obviously these fuzzy sets overlap. Hedges help to reflect human thinking, since people usually cannot distinguish between *slightly hot* and *moderately hot*.

Figure 4.5 illustrates an application of hedges. The fuzzy sets shown previously in Figure 4.3 are now modified mathematically by the hedge *very*. Consider, for example, a man who is 185 cm tall. He is a member of the *tall men* set with a degree of membership of 0.5. However, he is also a member of the set of *very tall men* with a degree of 0.15, which is *fairly* reasonable.
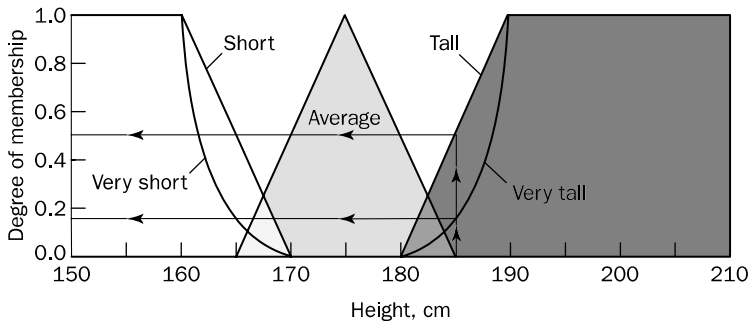
Figure 4.5   Fuzzy sets with *very* hedge

Let us now consider the hedges often used in practical applications.

- *Very*, the operation of concentration, as we mentioned above, narrows a set down and thus reduces the degree of membership of fuzzy elements. This operation can be given as a mathematical square:

$$\mu_A^{very}(x) = [\mu_A(x)]^2 \tag{4.6}$$

Hence, if Tom has a 0.86 membership in the set of *tall men*, he will have a 0.7396 membership in the set of *very tall men*.

- *Extremely* serves the same purpose as *very*, but does it to a greater extent. This operation can be performed by raising $\mu_A(x)$ to the third power:

$$\mu_A^{extremely}(x) = [\mu_A(x)]^3 \tag{4.7}$$

If Tom has a 0.86 membership in the set of *tall men*, he will have a 0.7396 membership in the set of *very tall men* and 0.6361 membership in the set of *extremely tall men*.

- *Very very* is just an extension of concentration. It can be given as a square of the operation of concentration:

$$\mu_A^{very\ very}(x) = [\mu_A^{very}(x)]^2 = [\mu_A(x)]^4 \tag{4.8}$$

For example, Tom, with a 0.86 membership in the *tall men set* and a 0.7396 membership in the *very tall men set*, will have a membership of 0.5470 in the set of *very very tall men*.

- *More or less*, the operation of dilation, expands a set and thus increases the degree of membership of fuzzy elements. This operation is presented as:

$$\mu_A^{more\ or\ less}(x) = \sqrt{\mu_A(x)} \tag{4.9}$$

Hence, if Tom has a 0.86 membership in the set of *tall men*, he will have a 0.9274 membership in the set of *more or less tall men*.

- *Indeed*, the operation of intensification, intensifies the meaning of the whole sentence. It can be done by increasing the degree of membership above 0.5 and decreasing those below 0.5. The hedge *indeed* may be given by either:
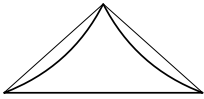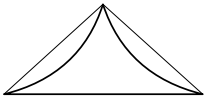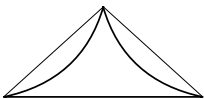
$$\mu_A^{indeed}(x) = 2[\mu_A(x)]^2 \qquad \text{if } 0 \leqslant \mu_A(x) \leqslant 0.5 \tag{4.10}$$

or

$$\mu_A^{indeed}(x) = 1 - 2[1 - \mu_A(x)]^2 \qquad \text{if } 0.5 < \mu_A(x) \leqslant 1 \tag{4.11}$$

If Tom has a 0.86 membership in the set of *tall men*, he can have a 0.9608 membership in the set of *indeed tall men*. In contrast, Mike, who has a 0.24 membership in *tall men* set, will have a 0.1152 membership in the *indeed tall men* set.

Table 4.2   Representation of hedges in fuzzy logic

Hedge	Mathematical expression	Graphical representation
A little	$[\mu_A(x)]^{1.3}$	
Slightly	$[\mu_A(x)]^{1.7}$	
Very	$[\mu_A(x)]^2$	
Extremely	$[\mu_A(x)]^3$	
Very very	$[\mu_A(x)]^4$	
More or less	$\sqrt{\mu_A(x)}$	
Somewhat	$\sqrt{\mu_A(x)}$	
Indeed	$2[\mu_A(x)]^2$ if $0 \leqslant \mu_A \leqslant 0.5$ $1 - 2[1 - \mu_A(x)]^2$ if $0.5 < \mu_A \leqslant 1$	

Mathematical and graphical representation of hedges are summarised in Table 4.2.

4.4 Operations of fuzzy sets

The classical set theory developed in the late 19th century by Georg Cantor describes how crisp sets can interact. These interactions are called **operations**.

Cantor's sets
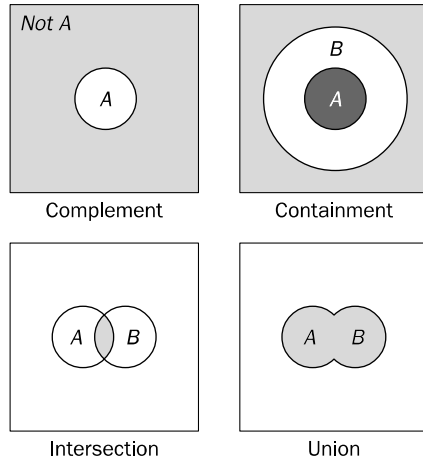
Figure 4.6   Operations on classical sets

We look at four of them: complement, containment, intersection and union. These operations are presented graphically in Figure 4.6. Let us compare operations of classical and fuzzy sets.

Complement

- Crisp sets: Who does not belong to the set?
- Fuzzy sets: How much do elements not belong to the set?

The complement of a set is an opposite of this set. For example, if we have the set of *tall men*, its complement is the set of *NOT tall men*. When we remove the tall men set from the universe of discourse, we obtain the complement. If *A* is the fuzzy set, its complement ¬*A* can be found as follows:

$$\mu_{\neg A}(x) = 1 - \mu_A(x) \tag{4.12}$$

For example, if we have a fuzzy set of *tall men*, we can easily obtain the fuzzy set of *NOT tall men*:

$$tall\ men = (0/180, 0.25/182.5, 0.5/185, 0.75/187.5, 1/190)$$
$$NOT\ tall\ men = (1/180, 0.75/182.5, 0.5/185, 0.25/187.5, 0/190)$$

Containment

- Crisp sets: Which sets belong to which other sets?
- Fuzzy sets: Which sets belong to other sets?

Similar to a Chinese box or Russian doll, a set can contain other sets. The smaller set is called the **subset**. For example, the set of *tall men* contains all tall men. Therefore, *very tall men* is a subset of *tall men*. However, the *tall men* set is just a subset of the set of *men*. In crisp sets, all elements of a subset entirely belong to a larger set and their membership values are equal to 1. In fuzzy sets, however, each element can belong less to the subset than to the larger set. Elements of the fuzzy subset have smaller memberships in it than in the larger set.

$$tall\ men = (0/180, 0.25/182.5, 0.50/185, 0.75/187.5, 1/190)$$
$$very\ tall\ men = (0/180, 0.06/182.5, 0.25/185, 0.56/187.5, 1/190)$$

Intersection

- Crisp sets: Which element belongs to both sets?
- Fuzzy sets: How much of the element is in both sets?

In classical set theory, an intersection between two sets contains the elements shared by these sets. If we have, for example, the set of *tall men* and the set of *fat men*, the intersection is the area where these sets overlap, i.e. Tom is in the intersection only if he is tall AND fat. In fuzzy sets, however, an element may partly belong to both sets with different memberships. Thus, a fuzzy intersection is the lower membership in both sets of each element.

The fuzzy operation for creating the intersection of two fuzzy sets A and B on universe of discourse X can be obtained as:

$$\mu_{A \cap B}(x) = min\,[\mu_A(x), \mu_B(x)] = \mu_A(x) \cap \mu_B(x), \quad \text{where } x \in X \tag{4.13}$$

Consider, for example, the fuzzy sets of *tall* and *average men*:

$$tall\ men = (0/165, 0/175, 0.0/180, 0.25/182.5, 0.5/185, 1/190)$$
$$average\ men = (0/165, 1/175, 0.5/180, 0.25/182.5, 0.0/185, 0/190)$$

According to Eq. (4.13), the intersection of these two sets is

$$tall\ men \cap average\ men = (0/165, 0/175, 0/180, 0.25/182.5, 0/185, 0/190)$$

or

$$tall\ men \cap average\ men = (0/180, 0.25/182.5, 0/185)$$

This solution is represented graphically in Figure 4.3.

Union

- Crisp sets: Which element belongs to either set?
- Fuzzy sets: How much of the element is in either set?

The union of two crisp sets consists of every element that falls into either set. For example, the union of *tall men* and *fat men* contains all men who are tall OR fat, i.e. Tom is in the union since he is tall, and it does not matter whether he is fat or not. In fuzzy sets, the union is the reverse of the intersection. That is, the union is the largest membership value of the element in either set.

The fuzzy operation for forming the union of two fuzzy sets A and B on universe X can be given as:

$$\mu_{A \cup B}(x) = max\,[\mu_A(x), \mu_B(x)] = \mu_A(x) \cup \mu_B(x), \quad \text{where } x \in X \tag{4.14}$$

Consider again the fuzzy sets of *tall* and *average* men:

$$tall\ men = (0/165, 0/175, 0.0/180, 0.25/182.5, 0.5/185, 1/190)$$
$$average\ men = (0/165, 1/175, 0.5/180, 0.25/182.5, 0.0/185, 0/190)$$

According to Eq. (4.14), the union of these two sets is

$$tall\ men \cup average\ men = (0/165, 1/175, 0.5/180, 0.25/182.5, 0.5/185, 1/190)$$

Diagrams for fuzzy set operations are shown in Figure 4.7.

Crisp and fuzzy sets have the same properties; crisp sets can be considered as just a special case of fuzzy sets. Frequently used properties of fuzzy sets are described below.

Commutativity

$$A \cup B = B \cup A$$
$$A \cap B = B \cap A$$

Example:

tall men OR *short men* = *short men* OR *tall men*
tall men AND *short men* = *short men* AND *tall men*

Associativity
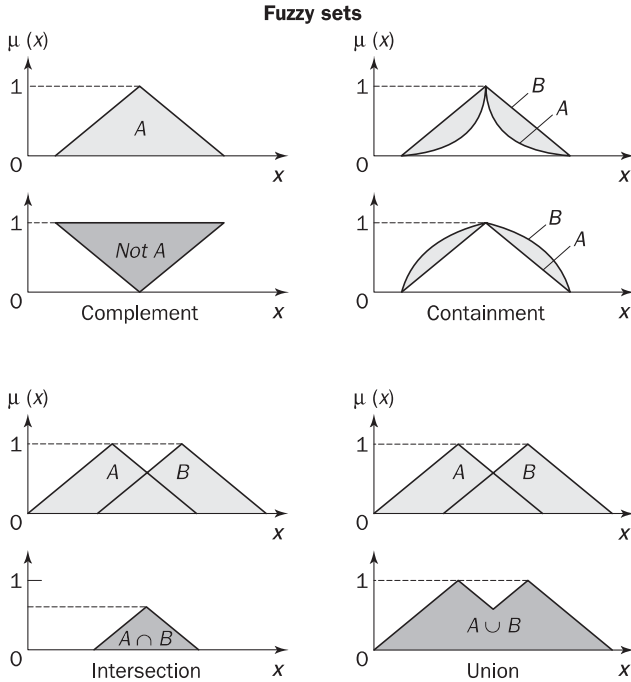
$$A \cup (B \cup C) = (A \cup B) \cup C$$
$$A \cap (B \cap C) = (A \cap B) \cap C$$

Figure 4.7   Operations of fuzzy sets

Example:

> *tall men* OR (*short men* OR *average men*) = (*tall men* OR *short men*) OR
> *average men*
> *tall men* AND (*short men* AND *average men*) = (*tall men* AND *short men*) AND
> *average men*

Distributivity

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

Example:

> *tall men* OR (*short men* AND *average men*) = (*tall men* OR *short men*) AND
> (*tall men* OR *average men*)
> *tall men* AND (*short men* OR *average men*) = (*tall men* AND *short men*) OR
> (*tall men* AND *average men*)

Idempotency

$$A \cup A = A$$
$$A \cap A = A$$

Example:

> *tall men* OR *tall men* = *tall men*
> *tall men* AND *tall men* = *tall men*

Identity

> $A \cup \emptyset = A$
> $A \cap X = A$
> $A \cap \emptyset = \emptyset$
> $A \cup X = X$

Example:

> *tall men* OR *undefined* = *tall men*
> *tall men* AND *unknown* = *tall men*
> *tall men* AND *undefined* = *undefined*
> *tall men* OR *unknown* = *unknown*

where *undefined* is an empty (**null**) set, the set having all degree of memberships equal to 0, and *unknown* is a set having all degree of memberships equal to 1.

Involution

> $\neg(\neg A) = A$

Example:

> NOT (NOT *tall men*) = *tall men*

Transitivity

> If $(A \subset B) \cap (B \subset C)$ then $A \subset C$
> Every set contains the subsets of its subsets.

Example:

> IF (*extremely tall men* $\subset$ *very tall men*) AND (*very tall men* $\subset$ *tall men*)
> THEN (*extremely tall men* $\subset$ *tall men*)

De Morgan's Laws

> $\neg(A \cap B) = \neg A \cup \neg B$
> $\neg(A \cup B) = \neg A \cap \neg B$

Example:

NOT (*tall men* AND *short men*) = NOT *tall men* OR NOT *short men*
NOT (*tall men* OR *short men*) = NOT *tall men* AND NOT *short men*

Using fuzzy set operations, their properties and hedges, we can easily obtain a variety of fuzzy sets from the existing ones. For example, if we have fuzzy set *A* of *tall men* and fuzzy set *B* of *short men*, we can derive fuzzy set *C* of *not very tall men and not very short men* or even set *D* of *not very very tall and not very very short men* from the following operations:

$$\mu_C(x) = [1 - \mu_A(x)^2] \cap [1 - (\mu_B(x)^2]$$
$$\mu_D(x) = [1 - \mu_A(x)^4] \cap [1 - (\mu_B(x)^4]$$

Generally, we apply fuzzy operations and hedges to obtain fuzzy sets which can represent linguistic descriptions of our natural language.

4.5 Fuzzy rules

In 1973, Lotfi Zadeh published his second most influential paper (Zadeh, 1973). This paper outlined a new approach to analysis of complex systems, in which Zadeh suggested capturing human knowledge in fuzzy rules.

What is a fuzzy rule?
A fuzzy rule can be defined as a conditional statement in the form:

IF   *x* is *A*
THEN   *y* is *B*

where *x* and *y* are linguistic variables; and *A* and *B* are linguistic values determined by fuzzy sets on the universe of discourses *X* and *Y*, respectively.

What is the difference between classical and fuzzy rules?
A classical IF-THEN rule uses binary logic, for example,

Rule: 1
IF   speed is > 100
THEN   stopping_distance is long

Rule: 2
IF   speed is < 40
THEN   stopping_distance is short

The variable speed can have any numerical value between 0 and 220 km/h, but the linguistic variable *stopping_distance* can take either value *long* or *short*. In

other words, classical rules are expressed in the black-and-white language of Boolean logic. However, we can also represent the stopping distance rules in a fuzzy form:

Rule: 1
IF   speed is fast
THEN   stopping_distance is long

Rule: 2
IF   speed is slow
THEN   stopping_distance is short

Here the linguistic variable *speed* also has the range (the universe of discourse) between 0 and 220 km/h, but this range includes fuzzy sets, such as *slow*, *medium* and *fast*. The universe of discourse of the linguistic variable *stopping_distance* can be between 0 and 300 m and may include such fuzzy sets as *short*, *medium* and *long*. Thus fuzzy rules relate to fuzzy sets.

Fuzzy expert systems merge the rules and consequently cut the number of rules by at least 90 per cent.

How to reason with fuzzy rules?

Fuzzy reasoning includes two distinct parts: evaluating the rule antecedent (the IF part of the rule) and *implication* or applying the result to the consequent (the THEN part of the rule).

In classical rule-based systems, if the rule antecedent is true, then the consequent is also true. In fuzzy systems, where the antecedent is a fuzzy statement, all rules fire to some extent, or in other words they fire partially. If the antecedent is true to some degree of membership, then the consequent is also true to that same degree.

Consider, for example, two fuzzy sets, 'tall men' and 'heavy men' represented in Figure 4.8.
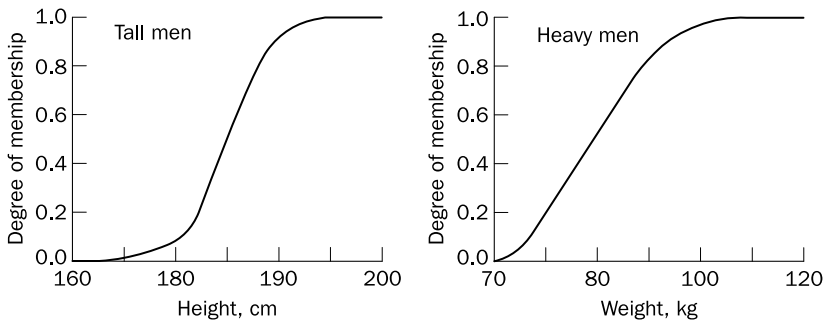
Figure 4.8   Fuzzy sets of *tall* and *heavy* men
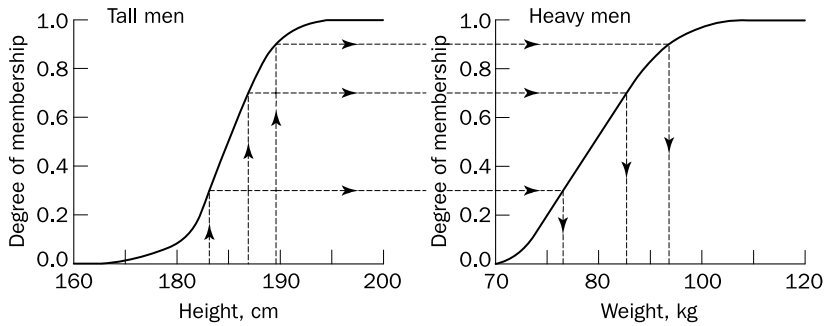
Figure 4.9   Monotonic selection of values for man weight

These fuzzy sets provide the basis for a weight estimation model. The model is based on a relationship between a man's height and his weight, which is expressed as a single fuzzy rule:

IF   height is *tall*
THEN   weight is *heavy*

The value of the output or a truth membership grade of the rule consequent can be estimated directly from a corresponding truth membership grade in the antecedent (Cox, 1999). This form of fuzzy inference uses a method called **monotonic selection**. Figure 4.9 shows how various values of men's weight are derived from different values for men's height.

Can the antecedent of a fuzzy rule have multiple parts?
As a production rule, a fuzzy rule can have multiple antecedents, for example:

IF   project_duration is long
AND   project_staffing is large
AND   project_funding is inadequate
THEN   risk is high

IF   service is excellent
OR   food is delicious
THEN   tip is generous

All parts of the antecedent are calculated simultaneously and resolved in a single number, using fuzzy set operations considered in the previous section.

Can the consequent of a fuzzy rule have multiple parts?
The consequent of a fuzzy rule can also include multiple parts, for instance:

IF   temperature is hot
THEN   hot_water is reduced;
          cold_water is increased

In this case, all parts of the consequent are affected equally by the antecedent.

In general, a fuzzy expert system incorporates not one but several rules that describe expert knowledge and play off one another. The output of each rule is a fuzzy set, but usually we need to obtain a single number representing the expert system output. In other words, we want to get a precise solution, not a fuzzy one.

How are all these output fuzzy sets combined and transformed into a single number?

To obtain a single crisp solution for the output variable, a fuzzy expert system first aggregates all output fuzzy sets into a single output fuzzy set, and then defuzzifies the resulting fuzzy set into a single number. In the next section we will see how the whole process works from the beginning to the end.

4.6   Fuzzy inference

Fuzzy inference can be defined as a process of mapping from a given input to an output, using the theory of fuzzy sets.

4.6.1   Mamdani-style inference

The most commonly used fuzzy inference technique is the so-called Mamdani method. In 1975, Professor Ebrahim Mamdani of London University built one of the first fuzzy systems to control a steam engine and boiler combination (Mamdani and Assilian, 1975). He applied a set of fuzzy rules supplied by experienced human operators.

The Mamdani-style fuzzy inference process is performed in four steps: fuzzification of the input variables, rule evaluation, aggregation of the rule outputs, and finally defuzzification.

To see how everything fits together, we examine a simple two-input one-output problem that includes three rules:

Rule: 1	Rule: 1
IF x is $A3$	IF *project_funding* is *adequate*
OR y is $B1$	OR *project_staffing* is *small*
THEN z is $C1$	THEN *risk* is *low*
Rule: 2	Rule: 2
IF x is $A2$	IF *project_funding* is *marginal*
AND y is $B2$	AND *project_staffing* is *large*
THEN z is $C2$	THEN *risk* is *normal*
Rule: 3	Rule: 3
IF x is $A1$	IF *project_funding* is *inadequate*
THEN z is $C3$	THEN *risk* is *high*

where x, y and z (*project funding*, *project staffing* and *risk*) are linguistic variables; $A1$, $A2$ and $A3$ (*inadequate*, *marginal* and *adequate*) are linguistic values

determined by fuzzy sets on universe of discourse X (*project funding*); $B1$ and $B2$ (*small* and *large*) are linguistic values determined by fuzzy sets on universe of discourse Y (*project staffing*); $C1$, $C2$ and $C3$ (*low*, *normal* and *high*) are linguistic values determined by fuzzy sets on universe of discourse Z (*risk*).

The basic structure of Mamdani-style fuzzy inference for our problem is shown in Figure 4.10.

Step 1:   ***Fuzzification***

The first step is to take the crisp inputs, $x1$ and $y1$ (*project funding* and *project staffing*), and determine the degree to which these inputs belong to each of the appropriate fuzzy sets.

What is a crisp input and how is it determined?

The crisp input is always a numerical value limited to the universe of discourse. In our case, values of $x1$ and $y1$ are limited to the universe of discourses X and Y, respectively. The ranges of the universe of discourses can be determined by expert judgements. For instance, if we need to examine the risk involved in developing the 'fuzzy' project, we can ask the expert to give numbers between 0 and 100 per cent that represent the project funding and the project staffing, respectively. In other words, the expert is required to answer to what extent the project funding and the project staffing are really adequate. Of course, various fuzzy systems use a variety of different crisp inputs. While some of the inputs can be measured directly (height, weight, speed, distance, temperature, pressure etc.), some of them can be based only on expert estimate.

Once the crisp inputs, $x1$ and $y1$, are obtained, they are fuzzified against the appropriate linguistic fuzzy sets. The crisp input $x1$ (project funding rated by the expert as 35 per cent) corresponds to the membership functions $A1$ and $A2$ (*inadequate* and *marginal*) to the degrees of 0.5 and 0.2, respectively, and the crisp input $y1$ (project staffing rated as 60 per cent) maps the membership functions $B1$ and $B2$ (*small* and *large*) to the degrees of 0.1 and 0.7, respectively. In this manner, each input is fuzzified over all the membership functions used by the fuzzy rules.

Step 2:   ***Rule evaluation***

The second step is to take the fuzzified inputs, $\mu_{(x=A1)} = 0.5$, $\mu_{(x=A2)} = 0.2$, $\mu_{(y=B1)} = 0.1$ and $\mu_{(y=B2)} = 0.7$, and apply them to the antecedents of the fuzzy rules. If a given fuzzy rule has multiple antecedents, the fuzzy operator (AND or OR) is used to obtain a single number that represents the result of the antecedent evaluation. This number (the truth value) is then applied to the consequent membership function.

To evaluate the disjunction of the rule antecedents, we use the OR fuzzy operation. Typically, fuzzy expert systems make use of the classical fuzzy operation *union* (4.14) shown in Figure 4.10 (Rule 1):

$$\mu_{A \cup B}(x) = max\,[\mu_A(x), \mu_B(x)]$$
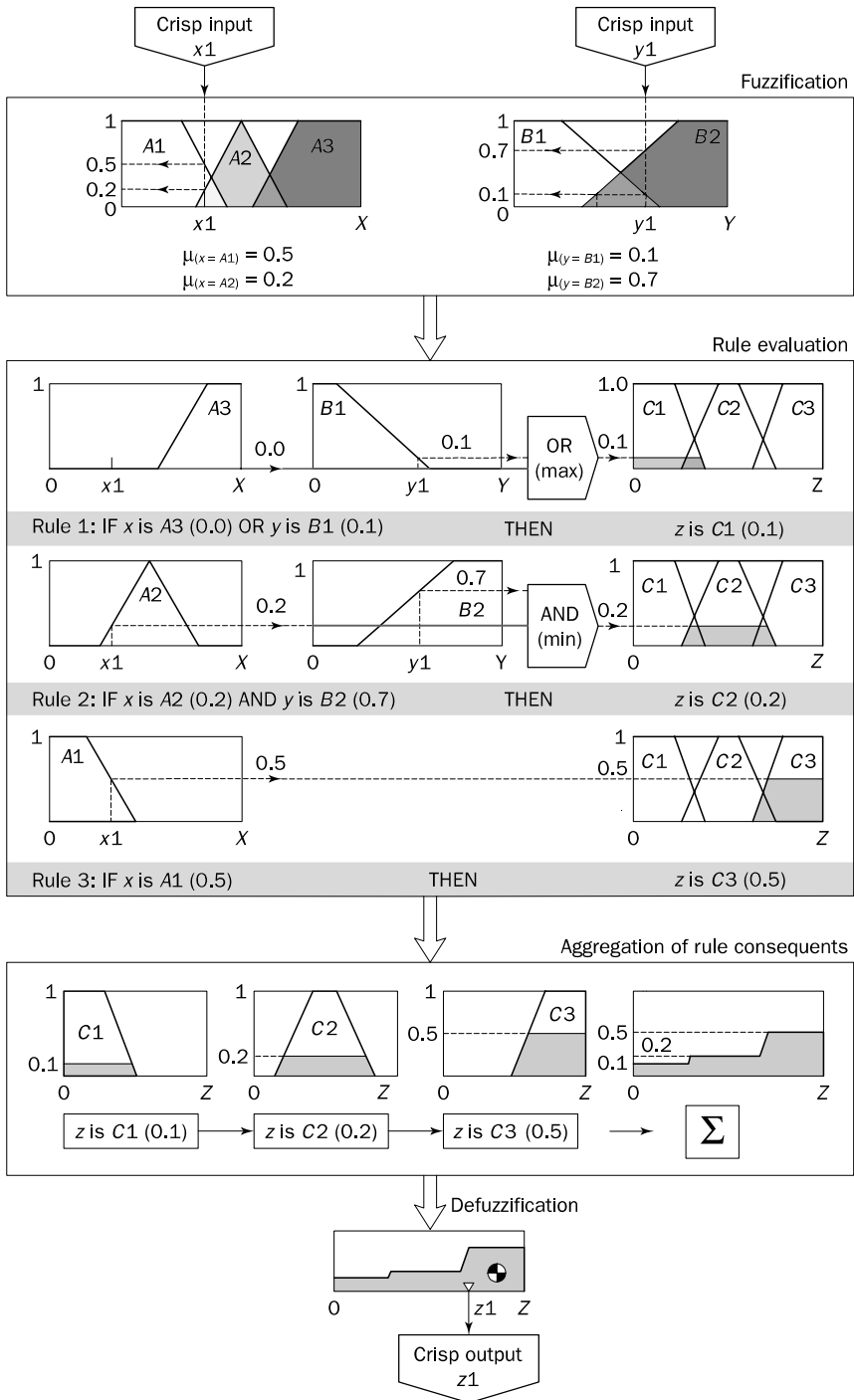
Figure 4.10 The basic structure of Mamdani-style fuzzy inference

However, the OR operation can be easily customised if necessary. For example, the MATLAB Fuzzy Logic Toolbox has two built-in OR methods: *max* and the probabilistic OR method, *probor*. The probabilistic OR, also known as the **algebraic sum**, is calculated as:

$$\mu_{A \cup B}(x) = probor\,[\mu_A(x), \mu_B(x)] = \mu_A(x) + \mu_B(x) - \mu_A(x) \times \mu_B(x) \quad (4.15)$$

Similarly, in order to evaluate the conjunction of the rule antecedents, we apply the AND fuzzy operation *intersection* (4.13) also shown in Figure 4.10 (Rule 2):

$$\mu_{A \cap B}(x) = min\,[\mu_A(x), \mu_B(x)]$$

The Fuzzy Logic Toolbox also supports two AND methods: *min* and the product, *prod*. The product is calculated as:

$$\mu_{A \cap B}(x) = prod\,[\mu_A(x), \mu_B(x)] = \mu_A(x) \times \mu_B(x) \quad (4.16)$$

Do different methods of the fuzzy operations produce different results?
Fuzzy researchers have proposed and applied several approaches to execute AND and OR fuzzy operators (Cox, 1999) and, of course, different methods may lead to different results. Most fuzzy packages also allow us to customise the AND and OR fuzzy operations and a user is required to make the choice.

Let us examine our rules again.

Rule: 1
IF   x is A3 (0.0)
OR   y is B1 (0.1)
THEN   z is C1 (0.1)

$$\mu_{C1}(z) = max\,[\mu_{A3}(x), \mu_{B1}(y)] = max\,[0.0, 0.1] = 0.1$$

or

$$\mu_{C1}(z) = probor\,[\mu_{A3}(x), \mu_{B1}(y)] = 0.0 + 0.1 - 0.0 \times 0.1 = 0.1$$

Rule: 2
IF   x is A2 (0.2)
AND   y is B2 (0.7)
THEN   z is C2 (0.2)

$$\mu_{C2}(z) = min\,[\mu_{A2}(x), \mu_{B2}(y)] = min\,[0.2, 0.7] = 0.2$$

or

$$\mu_{C2}(z) = prod\,[\mu_{A2}(x), \mu_{B2}(y)] = 0.2 \times 0.7 = 0.14$$

Thus, Rule 2 can be also represented as shown in Figure 4.11.

Now the result of the antecedent evaluation can be applied to the membership function of the consequent. In other words, the consequent membership function is **clipped** or **scaled** to the level of the truth value of the rule antecedent.

Rule 2: IF x is A2 (0.2) AND y is B2 (0.7)   THEN   z is C2 (0.14)
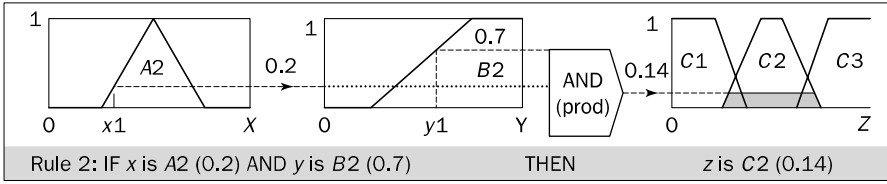
Figure 4.11   The AND *product* fuzzy operation

What do we mean by 'clipped or scaled'?

The most common method of correlating the rule consequent with the truth value of the rule antecedent is to simply cut the consequent membership function at the level of the antecedent truth. This method is called **clipping** or **correlation minimum**. Since the top of the membership function is sliced, the clipped fuzzy set loses some informa-tion. However, clipping is still often preferred because it involves less complex and faster mathematics, and generates an aggregated output surface that is easier to defuzzify.

While clipping is a frequently used method, **scaling** or **correlation product** offers a better approach for preserving the original shape of the fuzzy set. The original membership function of the rule consequent is adjusted by multiplying all its membership degrees by the truth value of the rule antecedent. This method, which generally loses less information, can be very useful in fuzzy expert systems.

Clipped and scaled membership functions are illustrated in Figure 4.12.

Step 3:   *Aggregation of the rule outputs*

Aggregation is the process of unification of the outputs of all rules. In other words, we take the membership functions of all rule conse-quents previously clipped or scaled and combine them into a single fuzzy set. Thus, the input of the aggregation process is the list of clipped or scaled consequent membership functions, and the output is one fuzzy set for each output variable. Figure 4.10 shows how the output of each rule is aggregated into a single fuzzy set for the overall fuzzy output.
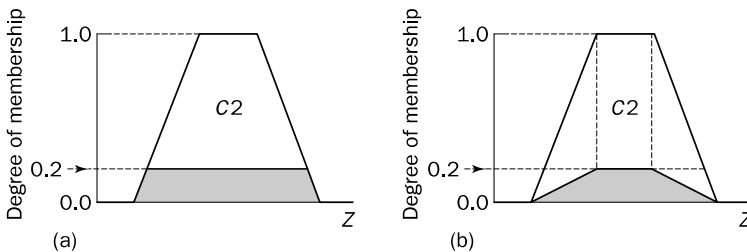
Figure 4.12   Clipped (a) and scaled (b) membership functions

Step 4:   *Defuzzification*

The last step in the fuzzy inference process is defuzzification. Fuzziness helps us to evaluate the rules, but the final output of a fuzzy system has to be a crisp number. The input for the defuzzification process is the aggregate output fuzzy set and the output is a single number.

How do we defuzzify the aggregate fuzzy set?

There are several defuzzification methods (Cox, 1999), but probably the most popular one is the **centroid technique**. It finds the point where a vertical line would slice the aggregate set into two equal masses. Mathematically this **centre of gravity** (COG) can be expressed as

$$COG = \frac{\int_a^b \mu_A(x)xdx}{\int_a^b \mu_A(x)dx} \tag{4.17}$$

As Figure 4.13 shows, a centroid defuzzification method finds a point representing the centre of gravity of the fuzzy set, *A*, on the interval, *ab*.

In theory, the COG is calculated over a continuum of points in the aggregate output membership function, but in practice, a reasonable estimate can be obtained by calculating it over a sample of points, as shown in Figure 4.13. In this case, the following formula is applied:

$$COG = \frac{\sum_{x=a}^b \mu_A(x)x}{\sum_{x=a}^b \mu_A(x)} \tag{4.18}$$

Let us now calculate the centre of gravity for our problem. The solution is presented in Figure 4.14.
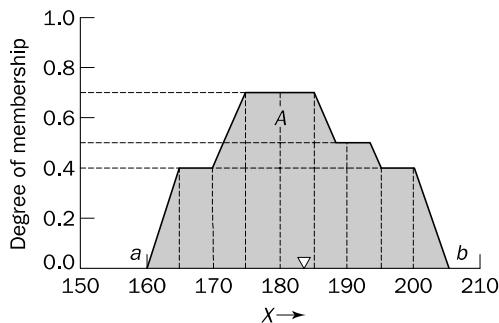
Figure 4.13   The centroid method of defuzzification
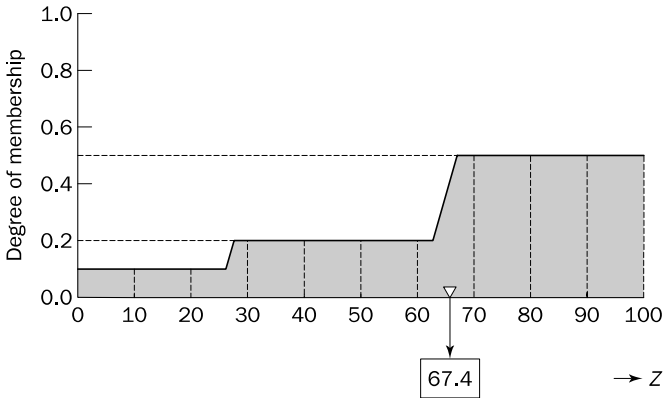
Figure 4.14   Defuzzifying the solution variable's fuzzy set

$$\text{COG} = \frac{(0+10+20) \times 0.1 + (30+40+50+60) \times 0.2 + (70+80+90+100) \times 0.5}{0.1+0.1+0.1+0.2+0.2+0.2+0.2+0.5+0.5+0.5+0.5}$$
$$= 67.4$$

Thus, the result of defuzzification, crisp output $z1$, is 67.4. It means, for instance, that the risk involved in our 'fuzzy' project is 67.4 per cent.

4.6.2   Sugeno-style inference

Mamdani-style inference, as we have just seen, requires us to find the centroid of a two-dimensional shape by integrating across a continuously varying function. In general, this process is not computationally efficient.

Could we shorten the time of fuzzy inference?

We can use a single spike, a **singleton**, as the membership function of the rule consequent. This method was first introduced by Michio Sugeno, the 'Zadeh of Japan', in 1985 (Sugeno, 1985). A singleton, or more precisely a fuzzy singleton, is a fuzzy set with a membership function that is unity at a single particular point on the universe of discourse and zero everywhere else.

Sugeno-style fuzzy inference is very similar to the Mamdani method. Sugeno changed only a rule consequent. Instead of a fuzzy set, he used a mathematical function of the input variable. The format of the Sugeno-style fuzzy rule is

IF x is A
AND y is B
THEN z is $f(x,y)$

where x, y and z are linguistic variables; A and B are fuzzy sets on universe of discourses X and Y, respectively; and $f(x,y)$ is a mathematical function.
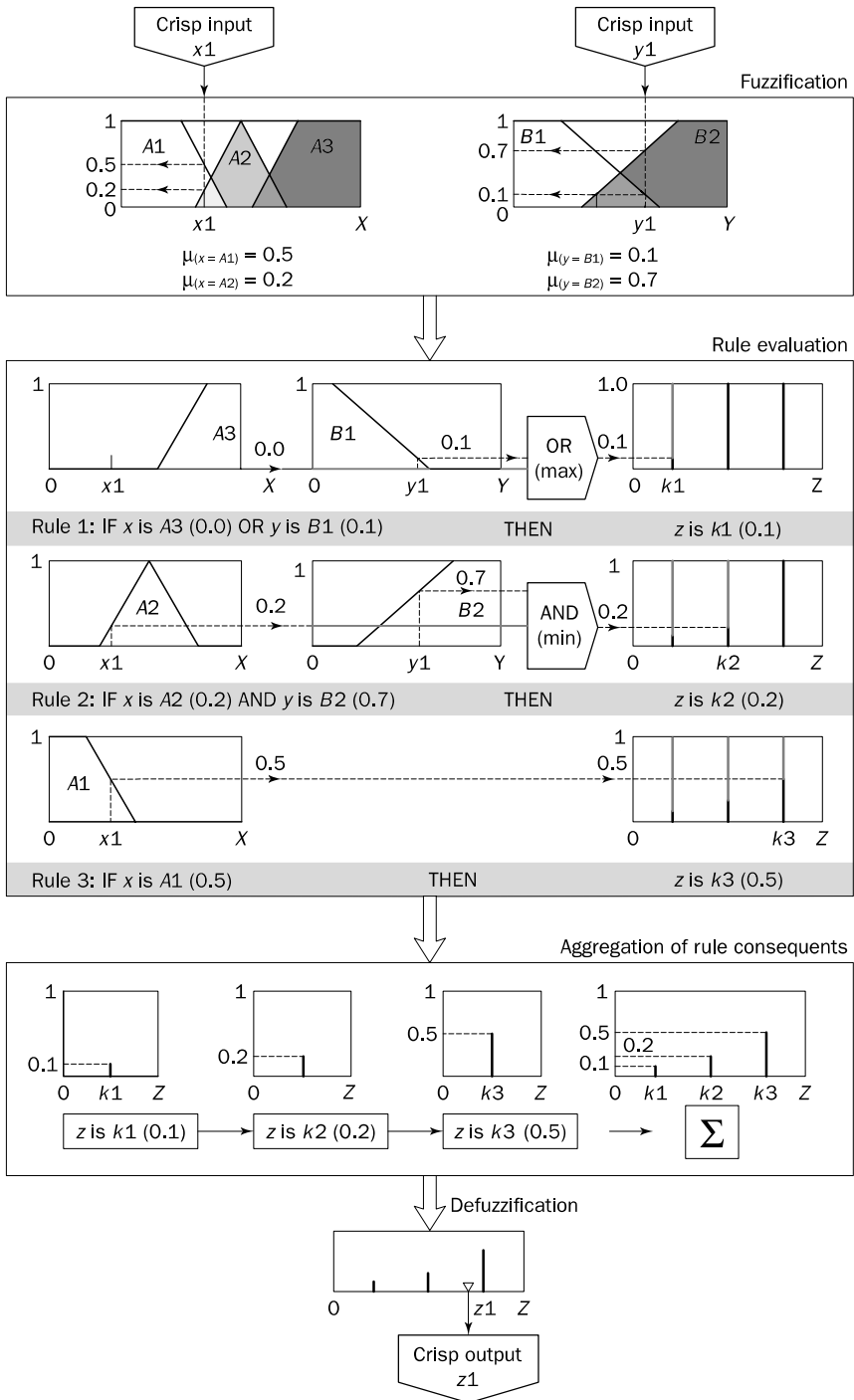
Figure 4.15   The basic structure of Sugeno-style fuzzy inference

The most commonly used **zero-order Sugeno fuzzy model** applies fuzzy rules in the following form:

IF x is A
AND y is B
THEN z is k

where k is a constant.

In this case, the output of each fuzzy rule is constant. In other words, all consequent membership functions are represented by singleton spikes. Figure 4.15 shows the fuzzy inference process for a zero-order Sugeno model. Let us compare Figure 4.15 with Figure 4.10. The similarity of Sugeno and Mamdani methods is quite noticeable. The only distinction is that rule consequents are singletons in Sugeno's method.

How is the result, crisp output, obtained?

As you can see from Figure 4.15, the aggregation operation simply includes all the singletons. Now we can find the **weighted average** (WA) of these singletons:

$$\text{WA} = \frac{\mu(k1) \times k1 + \mu(k2) \times k2 + \mu(k3) \times k3}{\mu(k1) + \mu(k2) + \mu(k3)} = \frac{0.1 \times 20 + 0.2 \times 50 + 0.5 \times 80}{0.1 + 0.2 + 0.5} = 65$$

Thus, a zero-order Sugeno system might be sufficient for our problem's needs. Fortunately, singleton output functions satisfy the requirements of a given problem quite often.

How do we make a decision on which method to apply – Mamdani or Sugeno?

The Mamdani method is widely accepted for capturing expert knowledge. It allows us to describe the expertise in more intuitive, more human-like manner. However, Mamdani-type fuzzy inference entails a substantial computational burden. On the other hand, the Sugeno method is computationally effective and works well with optimisation and adaptive techniques, which makes it very attractive in control problems, particularly for dynamic nonlinear systems.

4.7   Building a fuzzy expert system

To illustrate the design of a fuzzy expert system, we will consider a problem of operating a service centre of spare parts (Turksen *et al.*, 1992).

A service centre keeps spare parts and repairs failed ones. A customer brings a failed item and receives a spare of the same type. Failed parts are repaired, placed on the shelf, and thus become spares. If the required spare is available on the shelf, the customer takes it and leaves the service centre. However, if there is no spare on the shelf, the customer has to wait until the needed item becomes available. The objective here is to advise a manager of the service centre on certain decision policies to keep the customers satisfied.

A typical process in developing the fuzzy expert system incorporates the following steps:

1. Specify the problem and define linguistic variables.
2. Determine fuzzy sets.
3. Elicit and construct fuzzy rules.
4. Encode the fuzzy sets, fuzzy rules and procedures to perform fuzzy inference into the expert system.
5. Evaluate and tune the system.

Step 1:   *Specify the problem and define linguistic variables*

The first, and probably the most important, step in building any expert system is to specify the problem. We need to describe our problem in terms of knowledge engineering. In other words, we need to determine problem input and output variables and their ranges.

For our problem, there are four main linguistic variables: average waiting time (mean delay) m, repair utilisation factor of the service centre ρ, number of servers s, and initial number of spare parts n.

The customer's average waiting time, m, is the most important criterion of the service centre's performance. The actual mean delay in service should not exceed the limits acceptable to customers.

The repair utilisation factor of the service centre, ρ, is the ratio of the customer arrival rate, λ, to the customer departure rate, μ. Magnitudes of λ and μ indicate the rates of an item's failure (failures per unit time) and repair (repairs per unit time), respectively. Apparently, the repair rate is proportional to the number of servers, s. To increase the productivity of the service centre, its manager will try to keep the repair utilisation factor as high as possible.

The number of servers, s, and the initial number of spares, n, directly affect the customer's average waiting time, and thus have a major impact on the centre's performance. By increasing s and n, we achieve lower values of the mean delay, but, at the same time we increase the costs of employing new servers, building up the number of spares and expanding the inventory capacities of the service centre for additional spares.

Let us determine the initial number of spares n, given the customer's mean delay m, number of servers s, and repair utilisation factor, ρ. Thus, in the decision model considered here, we have three inputs – m, s and ρ, and one output – n. In other words, a manager of the service centre wants to determine the number of spares required to maintain the actual mean delay in customer service within an acceptable range.

Now we need to specify the ranges of our linguistic variables. Suppose we obtain the results shown in Table 4.3 where the intervals for m, s and n are normalised to be within the range of $[0, 1]$ by dividing base numerical values by the corresponding maximum magnitudes.

Table 4.3 Linguistic variables and their ranges

Linguistic variable: Mean delay, m		
Linguistic value	**Notation**	**Numerical range (normalised)**
Very Short	VS	[0, 0.3]
Short	S	[0.1, 0.5]
Medium	M	[0.4, 0.7]
Linguistic variable: Number of servers, s		
Linguistic value	**Notation**	**Numerical range (normalised)**
Small	S	[0, 0.35]
Medium	M	[0.30, 0.70]
Large	L	[0.60, 1]
Linguistic variable: Repair utilisation factor, ρ		
Linguistic value	**Notation**	**Numerical range**
Low	L	[0, 0.6]
Medium	M	[0.4, 0.8]
High	H	[0.6, 1]
Linguistic variable: Number of spares, n		
Linguistic value	**Notation**	**Numerical range (normalised)**
Very Small	VS	[0, 0.30]
Small	S	[0, 0.40]
Rather Small	RS	[0.25, 0.45]
Medium	M	[0.30, 0.70]
Rather Large	RL	[0.55, 0.75]
Large	L	[0.60, 1]
Very Large	VL	[0.70, 1]

Note, that for the customer mean delay m, we consider only three linguistic values – *Very Short*, *Short* and *Medium* because other values such as *Long* and *Very Long* are simply not practical. A manager of the service centre cannot afford to keep customers waiting longer than a medium time.

In practice, all linguistic variables, linguistic values and their ranges are usually chosen by the domain expert.

Step 2: ***Determine fuzzy sets***

Fuzzy sets can have a variety of shapes. However, a triangle or a trapezoid can often provide an adequate representation of the expert knowledge, and at the same time significantly simplifies the process of computation.

Figures 4.16 to 4.19 show the fuzzy sets for all linguistic variables used in our problem. As you may notice, one of the key points here is to maintain sufficient overlap in adjacent fuzzy sets for the fuzzy system to respond smoothly.
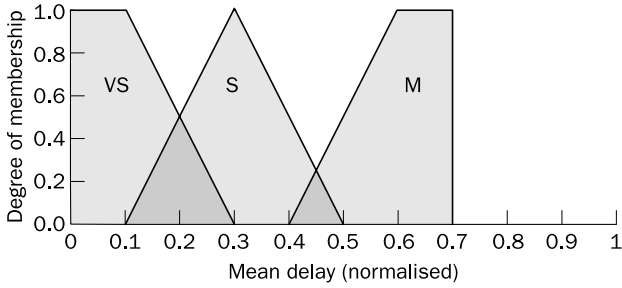
Figure 4.16   Fuzzy sets of mean delay m
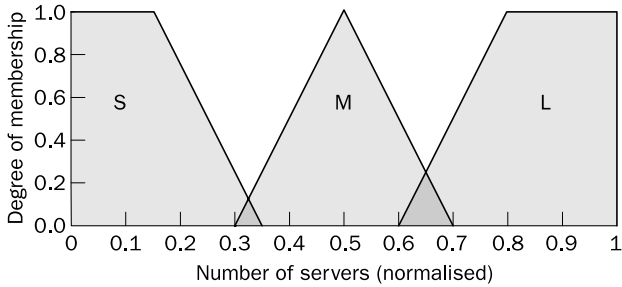
Figure 4.17   Fuzzy sets of number of servers s
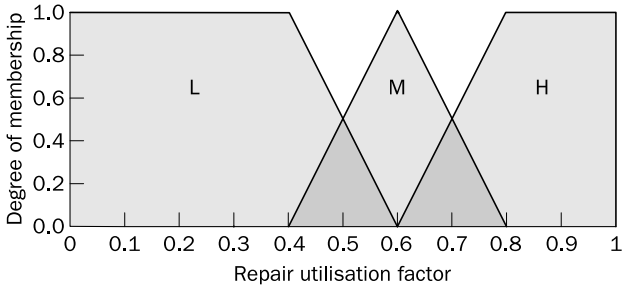
Figure 4.18   Fuzzy sets of repair utilisation factor ρ
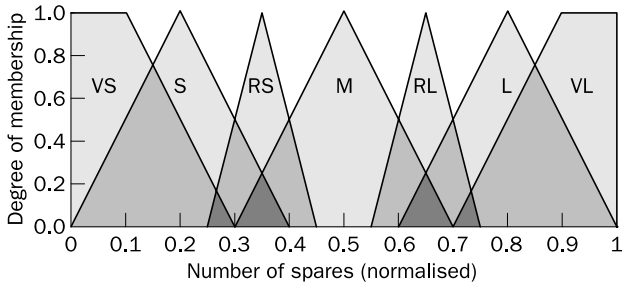
Figure 4.19   Fuzzy sets of number of spares n

Step 3: *Elicit and construct fuzzy rules*

Next we need to obtain fuzzy rules. To accomplish this task, we might ask the expert to describe how the problem can be solved using the fuzzy linguistic variables defined previously.

Required knowledge also can be collected from other sources such as books, computer databases, flow diagrams and observed human behaviour. In our case, we could apply rules provided in the research paper (Turksen *et al.*, 1992).

There are three input and one output variables in our example. It is often convenient to represent fuzzy rules in a matrix form. A two-by-one system (two inputs and one output) is depicted as an M × N matrix of input variables. The linguistic values of one input variable form the horizontal axis and the linguistic values of the other input variable form the vertical axis. At the intersection of a row and a column lies the linguistic value of the output variable. For a three-by-one system (three inputs and one output), the representation takes the shape of an M × N × K cube. This form of representation is called a **fuzzy associative memory** (FAM).

Let us first make use of a very basic relation between the repair utilisation factor ρ, and the number of spares n, assuming that other input variables are fixed. This relation can be expressed in the following form: if ρ increases, then n will not decrease. Thus we could write the following three rules:

1. If (utilisation_factor is L) then (number_of_spares is S)
2. If (utilisation_factor is M) then (number_of_spares is M)
3. If (utilisation_factor is H) then (number_of_spares is L)

Now we can develop the 3 × 3 FAM that will represent the rest of the rules in a matrix form. The results of this effort are shown in Figure 4.20.

Meanwhile, a detailed analysis of the service centre operation, together with an 'expert touch' (Turksen *et al.*, 1992), may enable us to derive 27 rules that represent complex relationships between all variables used in the expert system. Table 4.4 contains these rules and Figure 4.21 shows the cube (3 × 3 × 3) FAM representation.
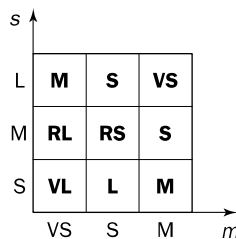
Figure 4.20 The square FAM representation

Table 4.4 The rule table

Rule	m	s	ρ	n	Rule	m	s	ρ	n	Rule	m	s	ρ	n
1	VS	S	L	VS	10	VS	S	M	S	19	VS	S	H	VL
2	S	S	L	VS	11	S	S	M	VS	20	S	S	H	L
3	M	S	L	VS	12	M	S	M	VS	21	M	S	H	M
4	VS	M	L	VS	13	VS	M	M	RS	22	VS	M	H	M
5	S	M	L	VS	14	S	M	M	S	23	S	M	H	M
6	M	M	L	VS	15	M	M	M	VS	24	M	M	H	S
7	VS	L	L	S	16	VS	L	M	M	25	VS	L	H	RL
8	S	L	L	S	17	S	L	M	RS	26	S	L	H	M
9	M	L	L	VS	18	M	L	M	S	27	M	L	H	RS
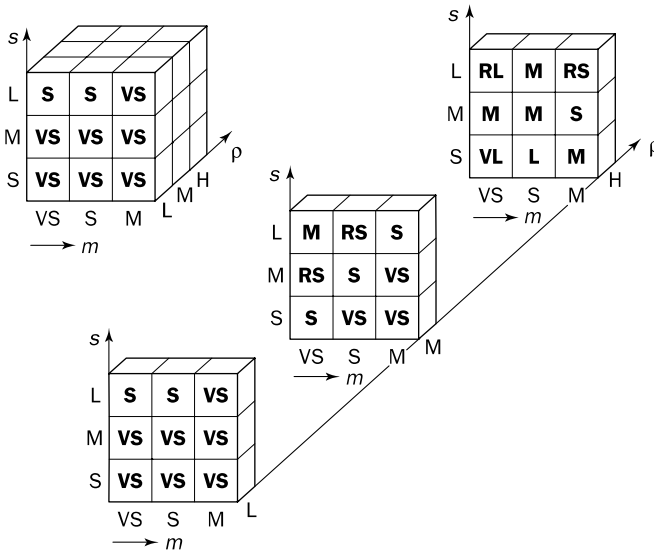
Figure 4.21 Cube FAM and sliced cube FAM representations

First we developed 12 $(3 + 3 \times 3)$ rules, but then we obtained 27 $(3 \times 3 \times 3)$ rules. If we implement both schemes, we can compare results; only the system's performance can tell us which scheme is better.

Rule Base 1

1. If (utilisation_factor is L) then (number_of_spares is S)
2. If (utilisation_factor is M) then (number_of_spares is M)
3. If (utilisation_factor is H) then (number_of_spares is L)

4. If (mean_delay is VS) and (number_of_servers is S) then (number_of_spares is VL)
5. If (mean_delay is S) and (number_of_servers is S) then (number_of_spares is L)

6. If (mean_delay is M) and (number_of_servers is S)
    then (number_of_spares is M)

7. If (mean_delay is VS) and (number_of_servers is M)
    then (number_of_spares is RL)
8. If (mean_delay is S) and (number_of_servers is M)
    then (number_of_spares is RS)
9. If (mean_delay is M) and (number_of_servers is M)
    then (number_of_spares is S)

10. If (mean_delay is VS) and (number_of_servers is L)
    then (number_of_spares is M)
11. If (mean_delay is S) and (number_of_servers is L)
    then (number_of_spares is S)
12. If (mean_delay is M) and (number_of_servers is L)
    then (number_of_spares is VS)

Rule Base 2
1. If (mean_delay is VS) and (number_of_servers is S)
    and (utilisation_factor is L) then (number_of_spares is VS)
2. If (mean_delay is S) and (number_of_servers is S)
    and (utilisation_factor is L) then (number_of_spares is VS)
3. If (mean_delay is M) and (number_of_servers is S)
    and (utilisation_factor is L) then (number_of_spares is VS)
4. If (mean_delay is VS) and (number_of_servers is M)
    and (utilisation_factor is L) then (number_of_spares is VS)
5. If (mean_delay is S) and (number_of_servers is M)
    and (utilisation_factor is L) then (number_of_spares is VS)
6. If (mean_delay is M) and (number_of_servers is M)
    and (utilisation_factor is L) then (number_of_spares is VS)
7. If (mean_delay is VS) and (number_of_servers is L)
    and (utilisation_factor is L) then (number_of_spares is S)
8. If (mean_delay is S) and (number_of_servers is L)
    and (utilisation_factor is L) then (number_of_spares is S)
9. If (mean_delay is M) and (number_of_servers is L)
    and (utilisation_factor is L) then (number_of_spares is VS)

10. If (mean_delay is VS) and (number_of_servers is S)
    and (utilisation_factor is M) then (number_of_spares is S)
11. If (mean_delay is S) and (number_of_servers is S)
    and (utilisation_factor is M) then (number_of_spares is VS)
12. If (mean_delay is M) and (number_of_servers is S)
    and (utilisation_factor is M) then (number_of_spares is VS)
13. If (mean_delay is VS) and (number_of_servers is M)
    and (utilisation_factor is M) then (number_of_spares is RS)
14. If (mean_delay is S) and (number_of_servers is M)
    and (utilisation_factor is M) then (number_of_spares is S)
15. If (mean_delay is M) and (number_of_servers is M)
    and (utilisation_factor is M) then (number_of_spares is VS)
16. If (mean_delay is VS) and (number_of_servers is L)
    and (utilisation_factor is M) then (number_of_spares is M)
17. If (mean_delay is S) and (number_of_servers is L)
    and (utilisation_factor is M) then (number_of_spares is RS)
18. If (mean_delay is M) and (number_of_servers is L)
    and (utilisation_factor is M) then (number_of_spares is S)

19. If (mean_delay is VS) and (number_of_servers is S)
    and (utilisation_factor is H) then (number_of_spares is VL)

20. If (mean_delay is S) and (number_of_servers is S)
   and (utilisation_factor is H) then (number_of_spares is L)
21. If (mean_delay is M) and (number_of_servers is S)
   and (utilisation_factor is H) then (number_of_spares is M)
22. If (mean_delay is VS) and (number_of_servers is M)
   and (utilisation_factor is H) then (number_of_spares is M)
23. If (mean_delay is S) and (number_of_servers is M)
   and (utilisation_factor is H) then (number_of_spares is M)
24. If (mean_delay is M) and (number_of_servers is M)
   and (utilisation_factor is H) then (number_of_spares is S)
25. If (mean_delay is VS) and (number_of_servers is L)
   and (utilisation_factor is H) then (number_of_spares is RL)
26. If (mean_delay is S) and (number_of_servers is L)
   and (utilisation_factor is H) then (number_of_spares is M)
27. If (mean_delay is M) and (number_of_servers is L)
   and (utilisation_factor is H) then (number_of_spares is RS)

Step 4: *Encode the fuzzy sets, fuzzy rules and procedures to perform fuzzy inference into the expert system*

The next task after defining fuzzy sets and fuzzy rules is to encode them, and thus actually build a fuzzy expert system. To accomplish this task, we may choose one of two options: to build our system using a programming language such as C or Pascal, or to apply a fuzzy logic development tool such as MATLAB Fuzzy Logic Toolbox® from the MathWorks or Fuzzy Knowledge Builder™ from Fuzzy Systems Engineering.

Most experienced fuzzy system builders often prefer the C/C++ programming language (Cox, 1999; Li and Gupta, 1995) because it offers greater flexibility. However, for rapid developing and prototyping a fuzzy expert system, the best choice is a fuzzy logic development tool. Such a tool usually provides complete environments for building and testing fuzzy systems. For example, the MATLAB Fuzzy Logic Toolbox has five integrated graphical editors: the fuzzy inference system editor, the rule editor, the membership function editor, the fuzzy inference viewer, and the output surface viewer. All these features make designing fuzzy systems much easier. This option is also preferable for novices, who do not have sufficient experience in building fuzzy expert systems. When a fuzzy logic development tool is chosen, the knowledge engineer needs only to encode fuzzy rules in English-like syntax, and define membership functions graphically.

To build our fuzzy expert system, we will use one of the most popular tools, the MATLAB Fuzzy Logic Toolbox. It provides a systematic framework for computing with fuzzy rules and graphical user interfaces. It is easy to master and convenient to use, even for new fuzzy system builders.

Step 5: *Evaluate and tune the system*

The last, and the most laborious, task is to evaluate and tune the system. We want to see whether our fuzzy system meets the requirements
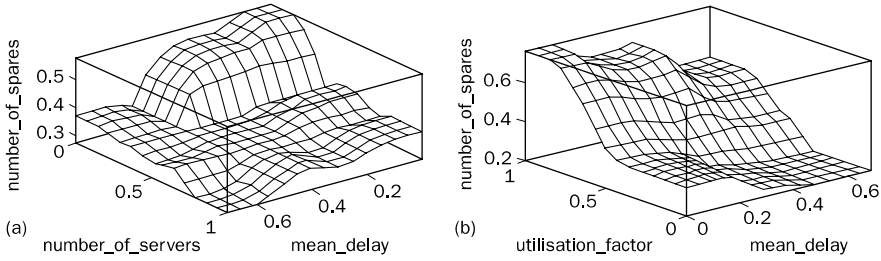
Figure 4.22   Three-dimensional plots for rule base 1

specified at the beginning. Several test situations depend on the mean delay, number of servers and repair utilisation factor. The Fuzzy Logic Toolbox can generate surface to help us analyse the system's perform-ance. Figure 4.22 represents three-dimensional plots for the two-input one-output system.

But our system has three inputs and one output. Can we move beyond three dimensions? When we move beyond three dimensions, we encounter difficulties in displaying the results. Luckily, the Fuzzy Logic Toolbox has a special capability: it can generate a three-dimensional output surface by varying any two of the inputs and keeping other inputs constant. Thus we can observe the performance of our three-input one-output system on two three-dimensional plots.

Although the fuzzy system works well, we may attempt to improve it by applying Rule Base 2. The results are shown in Figure 4.23. If we compare Figures 4.22 and 4.23, we will see the improvement.

However, even now, the expert might not be satisfied with the system performance. To improve it, he or she may suggest additional sets – *Rather Small* and *Rather Large* – on the universe of discourse *Number of servers* (as shown in Figure 4.24), and to extend the rule base according to the FAM presented in Figure 4.25. The ease with which a fuzzy system can be modified and extended permits us to follow the expert suggestions and quickly obtain results shown in Figure 4.26.
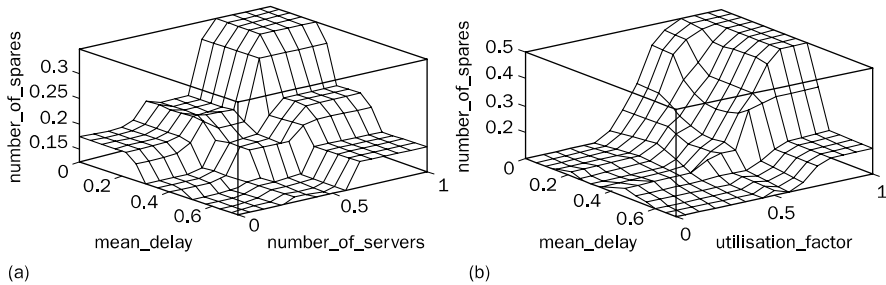
Figure 4.23   Three-dimensional plots for rule base 2
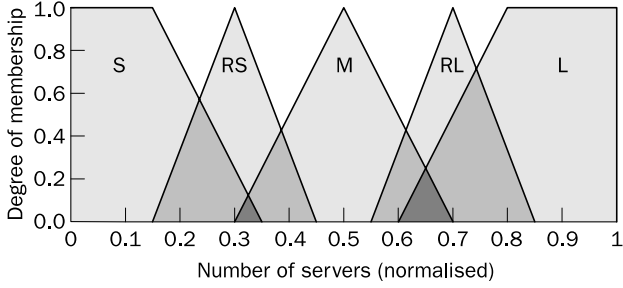
Figure 4.24   Modified fuzzy sets of number of servers *s*
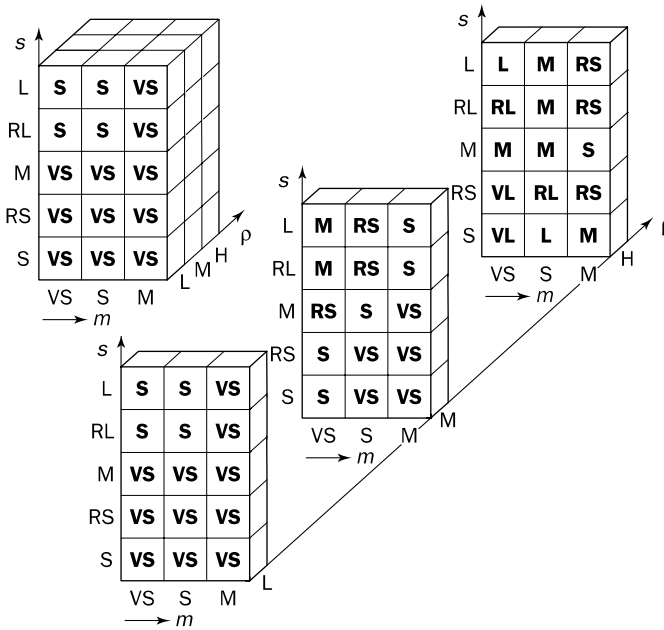
Figure 4.25   Cube FAM of rule base 3
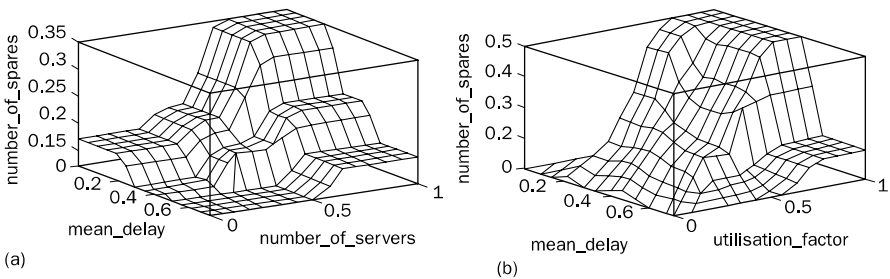
Figure 4.26   Three-dimensional plots for rule base 3

In general, tuning a fuzzy expert system takes much more time and effort than determining fuzzy sets and constructing fuzzy rules. Usually a reasonable solution to the problem can be achieved from the first series of fuzzy sets and fuzzy rules. This is an acknowledged advantage of fuzzy logic; however, improving the system becomes rather an art than engineering.

Tuning fuzzy systems may involve executing a number of actions in the following order:

1   Review model input and output variables, and if required redefine their ranges. Pay particular attention to the variable units. Variables used in the same domain must be measured in the same units on the universe of discourse.

2   Review the fuzzy sets, and if required define additional sets on the universe of discourse. The use of wide fuzzy sets may cause the fuzzy system to perform roughly.

3   Provide sufficient overlap between neighbouring sets. Although there is no precise method to determine the optimum amount of overlap, it is suggested that triangle-to-triangle and trapezoid-to-triangle fuzzy sets should overlap between 25 and 50 per cent of their bases (Cox, 1999).

4   Review the existing rules, and if required add new rules to the rule base.

5   Examine the rule base for opportunities to write hedge rules to capture the pathological behaviour of the system.

6   Adjust the rule execution weights. Most fuzzy logic tools allow control of the importance of rules by changing a weight multiplier.

In the Fuzzy Logic Toolbox, all rules have a default weight of (1.0), but the user can reduce the force of any rule by adjusting its weight. For example, if we specify

If (utilisation_factor is H) then (number_of_spares is L) (0.6)

then the rule's force will be reduced by 40 per cent.

7   Revise shapes of the fuzzy sets. In most cases, fuzzy systems are highly tolerant of a shape approximation, and thus a system can still behave well even when the shapes of the fuzzy sets are not precisely defined.

But how about defuzzification methods? Should we try different techniques to tune our system?

The centroid technique appears to provide consistent results. This is a well-balanced method sensitive to the height and width of the total fuzzy region as well as to sparse singletons. Therefore, unless you have a strong reason to believe that your fuzzy system will behave better under other defuzzification methods, the centroid technique is recommended.

4.8   Summary

In this chapter, we introduced fuzzy logic and discussed the philosophical ideas behind it. We presented the concept of fuzzy sets, considered how to represent a fuzzy set in a computer, and examined operations of fuzzy sets. We also defined linguistic variables and hedges. Then we presented fuzzy rules and explained the main differences between classical and fuzzy rules. We explored two fuzzy inference techniques – Mamdani and Sugeno – and suggested appropriate areas for their application. Finally, we introduced the main steps in developing a fuzzy expert system, and illustrated the theory through the actual process of building and tuning a fuzzy system.

The most important lessons learned in this chapter are:

- Fuzzy logic is a logic that describes fuzziness. As fuzzy logic attempts to model humans' sense of words, decision making and common sense, it is leading to more human intelligent machines.

- Fuzzy logic was introduced by Jan Lukasiewicz in the 1920s, scrutinised by Max Black in the 1930s, and rediscovered, extended into a formal system of mathematical logic and promoted by Lotfi Zadeh in the 1960s.

- Fuzzy logic is a set of mathematical principles for knowledge representation based on degrees of membership rather than on the crisp membership of classical binary logic. Unlike two-valued Boolean logic, fuzzy logic is multi-valued.

- A fuzzy set is a set with fuzzy boundaries, such as *short*, *average* or *tall* for men's height. To represent a fuzzy set in a computer, we express it as a function and then map the elements of the set to their degree of membership. Typical membership functions used in fuzzy expert systems are triangles and trapezoids.

- A linguistic variable is used to describe a term or concept with vague or fuzzy values. These values are represented in fuzzy sets.

- Hedges are fuzzy set qualifiers used to modify the shape of fuzzy sets. They include adverbs such as *very*, *somewhat*, *quite*, *more or less* and *slightly*. Hedges perform mathematical operations of concentration by reducing the degree of membership of fuzzy elements (e.g. *very* tall men), dilation by increasing the degree of membership (e.g. *more or less* tall men) and intensification by increasing the degree of membership above 0.5 and decreasing those below 0.5 (e.g. *indeed* tall men).

- Fuzzy sets can interact. These relations are called operations. The main operations of fuzzy sets are: complement, containment, intersection and union.

- Fuzzy rules are used to capture human knowledge. A fuzzy rule is a conditional statement in the form:

> IF   *x* is *A*
> THEN   *y* is *B*

where x and y are linguistic variables and A and B are linguistic values determined by fuzzy sets.

- Fuzzy inference is a process of mapping from a given input to an output by using the theory of fuzzy sets. The fuzzy inference process includes four steps: fuzzification of the input variables, rule evaluation, aggregation of the rule outputs and defuzzification.

- The two fuzzy inference techniques are the Mamdani and Sugeno methods. The Mamdani method is widely accepted in fuzzy expert systems for its ability to capture expert knowledge in fuzzy rules. However, Mamdani-type fuzzy inference entails a substantial computational burden.

- To improve the computational efficiency of fuzzy inference, Sugeno used a single spike, a singleton, as the membership function of the rule consequent. The Sugeno method works well with optimisation and adaptive techniques, which makes it very attractive in control, particularly for dynamic nonlinear systems.

- Building a fuzzy expert system is an iterative process that involves defining fuzzy sets and fuzzy rules, evaluating and then tuning the system to meet the specified requirements.

- Tuning is the most laborious and tedious part in building a fuzzy system. It often involves adjusting existing fuzzy sets and fuzzy rules.

Questions for review

1 What is fuzzy logic? Who are the founders of fuzzy logic? Why is fuzzy logic leading to more human intelligent machines?

2 What are a fuzzy set and a membership function? What is the difference between a crisp set and a fuzzy set? Determine possible fuzzy sets on the universe of discourse for *man weights*.

3 Define a linguistic variable and its value. Give an example. How are linguistic variables used in fuzzy rules? Give a few examples of fuzzy rules.

4 What is a hedge? How do hedges modify the existing fuzzy sets? Give examples of hedges performing operations of concentration, dilation and intensification. Provide appropriate mathematical expressions and their graphical representations.

5 Define main operations of fuzzy sets. Provide examples. How are fuzzy set operations, their properties and hedges used to obtain a variety of fuzzy sets from the existing ones?

6 What is a fuzzy rule? What is the difference between classical and fuzzy rules? Give examples.

7 Define fuzzy inference. What are the main steps in the fuzzy inference process?

8 How do we evaluate multiple antecedents of fuzzy rules? Give examples. Can different methods of executing the AND and OR fuzzy operations provide different results? Why?

9 What is clipping a fuzzy set? What is scaling a fuzzy set? Which method best preserves the original shape of the fuzzy set? Why? Give an example.

10 What is defuzzification? What is the most popular defuzzification method? How do we determine the final output of a fuzzy system mathematically and graphically?

11 What are the differences between Mamdani-type and Sugeno-type fuzzy inferences? What is a singleton?

12 What are the main steps in developing a fuzzy expert system? What is the most laborious and tedious part in this process? Why?

References

Chang, A.M. and Hall, L.O. (1992). The validation of fuzzy knowledge-based systems, *Fuzzy Logic for the Management of Uncertainty*, L.A. Zadeh and J. Kacprzyk, eds, John Wiley, New York, pp. 589–604.

Black, M. (1937). Vagueness: An exercise in logical analysis, *Philosophy of Science*, 4, 427–455.

Cox, E. (1999). *The Fuzzy Systems Handbook: A Practitioner's Guide to Building, Using, and Maintaining Fuzzy Systems,* 2nd edn. Academic Press, San Diego, CA.

Li, H. and Gupta, M. (1995). *Fuzzy Logic and Intelligent Systems*. Kluwer Academic Publishers, Boston.

Lukasiewicz, J. (1930). Philosophical remarks on many-valued systems of propositional logic. Reprinted in *Selected Works*, L. Borkowski, ed., Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1970, pp. 153–179.

Mamdani, E.H. and Assilian, S. (1975). An experiment in linguistic synthesis with a fuzzy logic controller, *International Journal of Man–Machine Studies*, 7(1), 1–13.

Sugeno, M. (1985). *Industrial Applications of Fuzzy Control*. North-Holland, Amsterdam.

Turksen, I.B., Tian, Y. and Berg, M. (1992). A fuzzy expert system for a service centre of spare parts, *Expert Systems with Applications*, 5, 447–464.

Zadeh, L. (1965). Fuzzy sets, *Information and Control*, 8(3), 338–353.

Zadeh, L. (1973). Outline of a new approach to the analysis of complex systems and decision processes, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(1), 28–44.

Bibliography

Chang, A.M. and Hall, L.O. (1992). The validation of fuzzy knowledge-based systems, *Fuzzy Logic for the Management of Uncertainty*, L.A. Zadeh and J. Kacprzyk, eds, John Wiley, New York, pp. 589–604.

Chiu, S. (1995). Software tools for fuzzy control, *Industrial Applications of Fuzzy Logic and Intelligent Systems*, J. Yen, R. Langari and L.A. Zadeh, eds, The Institute of Electrical and Electronics Engineers, New York, pp. 313–340.

Dubois, D. and Pride, H. (1992). Fuzzy rules in knowledge-based systems, *An Introduction to Fuzzy Logic Applications in Intelligent Systems*, R.R. Yager and L.A. Zadeh, eds, Kluwer Academic Publishers, Boston, pp. 45–68.

Dubois, D., Pride, H. and Yager, R.R. (1993). Fuzzy rules in knowledge-based systems, *Reading in Fuzzy Sets for Intelligent Systems*, Morgan Kaufmann, San Mateo, CA.

Durkin, J. (1994). *Expert Systems Design and Development*. Prentice Hall, Englewood Cliffs, NJ.

Kandel, A. (1992). *Fuzzy Expert Systems*. CRC Press, Boca Raton, FL.

Klir, G.J. and Yuan, Bo (1996). *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A. Zadeh*, Advances in Fuzzy Systems – Applications and Theory, vol. 6, World Scientific, Singapore.

Kohout, L.J. and Bandler, W. (1992). Use of fuzzy relations in knowledge representation, acquisition, and processing, *Fuzzy Logic for the Management of Uncertainty*, L.A. Zadeh and J. Kacprzyk, eds, John Wiley, New York, pp. 415–435.

Kosko, B. (1992). Fuzzy associative memory systems, *Fuzzy Expert Systems*, A. Kandel, ed., CRC Press, Boca Raton, FL, pp. 135–164.

Kosko, B. (1992). *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. Prentice Hall, Englewood Cliffs, NJ.

Kosko, B. (1993). *Fuzzy Thinking: The New Science of Fuzzy Logic*. Hyperion, New York.

Kosko, B. (1997). *Fuzzy Engineering*. Prentice Hall, Upper Saddle River, NJ.

Lee, C.C. (1990). Fuzzy logic in control systems: fuzzy logic controller, part I, *IEEE Transactions on Systems, Man, and Cybernerics*, 20(2), 404–418.

Lee, C.C. (1990). Fuzzy logic in control systems: fuzzy logic controller, part II, *IEEE Transactios on Systems, Man, and Cybernerics*, 20(2), 419–435.

Leondes, C.T. (1998). *Fuzzy Logic and Expert Systems Applications*. Academic Press, San Diego, CA.

Mukaidono, M. (2001). *Fuzzy Logic for Beginners*. World Scientific, Singapore.

Negoita, C.V. (1985). *Expert Systems and Fuzzy Systems*. Benjamin/Cummings Publishing, Menlo Park, CA.

Nguyen, H.T. and Walker, E.A. (1999). *A First Course in Fuzzy Logic*, 2nd edn. Chapman & Hall/CRC Press, London.

Pedrycz, W. and Gomide, F. (1998). *An Introduction to Fuzzy Sets: Analysis and Design*. The MIT Press, Cambridge, MA.

Ross, T.J. (1995). *Fuzzy Logic with Engineering Applications*. McGraw-Hill, Hightstown, NJ.

Tanaka, K. (1996). *An Introduction to Fuzzy Logic for Practical Applications*. Springer-Verlag, New York.

Terano, T., Asai, K. and Sugeno, M. (1992). *Fuzzy Systems Theory and its Applications*. Academic Press, London.

Tzafestas, S.G. and Venetsanopoulos, A.N. (1994). *Fuzzy Reasoning in Information, Decision and Control Systems*. Kluwer Academic Publishers, Boston.

Yager, R.R. (1992). Expert systems using fuzzy logic, *An Introduction to Fuzzy Logic Applications in Intelligent Systems*, R.R. Yager and L.A. Zadeh, eds, Kluwer Academic Publishers, Boston, pp. 27–44.

Yager, R.R. and Zadeh, L.A. (1992). *An Introduction to Fuzzy Logic Applications in Intelligent Systems*. Kluwer Academic Publishers, Boston.

Yager, R.R. and Zadeh, L.A. (1994). *Fuzzy Sets, Neural Networks and Soft Computing*. Van Nostrand Reinhold, New York.

Yen, J., Langari, R. and Zadeh, L.A. (1995). *Industrial Applications of Fuzzy Logic and Intelligent Systems*. Institute of Electrical and Electronics Engineers, New York.

Zadeh, L.A. (1975). The concept of a linguistic variable and its applications to approximate reasoning, part I, *Information Sciences*, 8, 199–249.

Zadeh, L.A. (1975). The concept of a linguistic variable and its applications to approximate reasoning, part II, *Information Sciences*, 8, 301–357.

Zadeh, L.A. (1975). The concept of a linguistic variable and its applications to approximate reasoning, part III, *Information Sciences*, 9, 43–80.

Zadeh, L.A. (1992). Knowledge representation in fuzzy logic, *An introduction to Fuzzy Logic Applications in Intelligent Systems*, R.R. Yager and L.A. Zadeh, eds, Kluwer Academic Publishers, Boston, pp. 1–26.

Zadeh, L.A. (1994). Fuzzy logic, neural networks, and soft computing, *Communications of the ACM*, 37(3), 77–84.

Zadeh, L.A. and Kacprzyk, J. (1992). *Fuzzy Logic for the Management of Uncertainty*. John Wiley, New York.

Zimmermann, H.-J. (1987). *Fuzzy Sets, Decision-Making and Expert Systems*. Kluwer Academic Publishers, Boston.

Zimmermann, H.-J. (1996). *Fuzzy Set Theory and its Applications*, 3rd edn. Kluwer Academic Publishers, Boston.

Frame-based expert systems   **5**

In which we introduce frames as one of the common methods used for representing knowledge in expert systems, describe how to develop a frame-based expert system and illustrate the theory through an example.

5.1   Introduction, or what is a frame?

Knowledge in a computer can be represented through several techniques. In the previous chapters, we considered rules. In this chapter, we will use **frames** for the knowledge representation.

What is a frame?

A frame is a data structure with typical knowledge about a particular object or concept. Frames, first proposed by Marvin Minsky in the 1970s (Minsky, 1975), are used to capture and represent knowledge in a frame-based expert system. Boarding passes shown in Figure 5.1 represent typical frames with knowledge about airline passengers. Both frames have the same structure.

Each frame has its own name and a set of **attributes**, or **slots**, associated with it. *Name*, *weight*, *height* and *age* are slots in the frame *Person*. *Model*, *processor*, *memory* and *price* are slots in the frame *Computer*. Each attribute or slot has a value attached to it. In Figure 5.1(a), for example, slot *Carrier* has value *QANTAS AIRWAYS* and slot *Gate* has value *2*. In some cases, instead of a particular value, a slot may have a procedure that determines the value.

In expert systems, frames are often used in conjunction with production rules.

Why is it necessary to use frames?

Frames provide a natural way for the structured and concise representation of knowledge. In a single entity, a frame combines all necessary knowledge about a particular object or concept. A frame provides a means of organising knowledge in slots to describe various attributes and characteristics of the object.

Earlier we argued that many real-world problems can be naturally expressed by IF-THEN production rules. However, a rule-based expert system using a

```
┌─────────────────────────────────┐   ┌─────────────────────────────────────┐
│ QANTAS BOARDING PASS            │   │ AIR NEW ZEALAND BOARDING PASS       │
│                                 │   │                                     │
│ Carrier:    QANTAS AIRWAYS      │   │ Carrier:    AIR NEW ZEALAND         │
│ Name:       MR N BLACK          │   │ Name:       MRS J WHITE             │
│ Flight:     QF 612              │   │ Flight:     NZ 0198                 │
│ Date:       29DEC               │   │ Date:       23NOV                   │
│ Seat:       23A                 │   │ Seat:       27K                     │
│ From:       HOBART              │   │ From:       MELBOURNE               │
│ To:         MELBOURNE           │   │ To:         CHRISTCHURCH            │
│ Boarding:   0620                │   │ Boarding:   1815                    │
│ Gate:       2                   │   │ Gate:       4                       │
└─────────────────────────────────┘   └─────────────────────────────────────┘
         (a)                                        (b)
```

Figure 5.1   Boarding pass frames

systematic search technique works with facts scattered throughout the entire knowledge base. It may search through the knowledge that is not relevant to a given problem, and as a result, the search may take a great deal of time. If, for example, we are searching for knowledge about Qantas frequent flyers, then we want to avoid the search through knowledge about Air New Zealand or British Airways passengers. In this situation, we need frames to collect the relevant facts within a single structure.

Basically, frames are an application of object-oriented programming for expert systems.

What is object-oriented programming?

Object-oriented programming can be defined as a programming method that uses **objects** as a basis for analysis, design and implementation. In object-oriented programming, an object is defined as a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand (Rumbaugh *et al.*, 1991). All objects have identity and are clearly distinguishable. *Michael Black*, *Audi 5000 Turbo*, *IBM Aptiva S35* are examples of objects.

An object combines both data structure and its behaviour in a single entity. This is in sharp contrast to conventional programming, in which data structure and the program behaviour have concealed or vague connections.

Object-oriented programming offers a natural way of representing the real world in a computer, and also illuminates the problem of data dependency, which is inherent in conventional programming (Taylor, 1992). When programmers create an object in an object-oriented programming language, they first assign a name to the object, then determine a set of attributes to describe the object's characteristics, and at last write procedures to specify the object's behaviour.

A knowledge engineer refers to an object as a **frame**, the term introduced by Minsky, which has become the AI jargon. Today the terms are used as synonyms.

5.2 Frames as a knowledge representation technique

The concept of a frame is defined by a collection of **slots**. Each slot describes a particular attribute or operation of the frame. In many respects, a frame resembles the traditional 'record' that contains information relevant to typical entities. Slots are used to store values. A slot may contain a default value or a pointer to another frame, a set of rules or procedure by which the slot value is obtained. In general, slots may include such information as:

1. **Frame name**.
2. **Relationship of the frame to the other frames**. The frame *IBM Aptiva S35* might be a member of the class *Computer*, which in turn might belong to the class *Hardware*.
3. **Slot value**. A slot value can be symbolic, numeric or Boolean. For example, in the frames shown in Figure 5.1, the slot *Name* has symbolic values, and the slot *Gate* numeric values. Slot values can be assigned when the frame is created or during a session with the expert system.
4. **Default slot value**. The default value is taken to be true when no evidence to the contrary has been found. For example, a car frame might have four wheels and a chair frame four legs as default values in the corresponding slots.
5. **Range of the slot value**. The range of the slot value determines whether a particular object or concept complies with the stereotype requirements defined by the frame. For example, the cost of a computer might be specified between $750 and $1500.
6. **Procedural information**. A slot can have a procedure (a self-contained arbitrary piece of computer code) attached to it, which is executed if the slot value is changed or needed. There are two types of procedures often attached to slots:
    (a) *WHEN CHANGED* procedure is executed when new information is placed in the slot.
    (b) *WHEN NEEDED* procedure is executed when information is needed for the problem solving, but the slot value is unspecified.

    Such procedural attachments are often called **demons**.

Frame-based expert systems also provide an extension to the slot-value structure through the application of **facets**.

What is a facet?
A facet is a means of providing extended knowledge about an attribute of a frame. Facets are used to establish the attribute value, control end-user queries, and tell the inference engine how to process the attribute.

In general, frame-based expert systems allow us to attach value, prompt and inference facets to attributes. **Value facets** specify default and initial values

of an attribute. **Prompt facets** enable the end-user to enter the attribute value on-line during a session with the expert system. And finally, **inference facets** allow us to stop the inference process when the value of a specified attribute changes.

What is the correct level of decomposition of a problem into frames, slots and facets?

Decomposition of a problem into frames, frames into slots and facets depends on the nature of the problem itself and the judgement of the knowledge engineer. There is no predefined 'correct' representation.

Figure 5.2 illustrates frames describing computers. The topmost frame represents the class *Computer* and the frames below describe instances *IBM Aptiva S35* and *IBM Aptiva S9C*. Two types of attributes are used here: *string* [Str] for symbolic information and *numeric* [N] for numeric data. Note default and initial value facets attached to the slots *Floppy*, *Power Supply*, *Warranty* and *Stock* in the class *Computer*. The attribute names, types, default and initial values are the properties inherited by instances.

What are the class and instances?

The word 'frame' often has a vague meaning. The frame may refer to a particular object, for example the computer *IBM Aptiva S35*, or to a group of similar objects. To be more precise, we will use the **instance-frame** when referring to a particular object, and the **class-frame** when referring to a group of similar objects.

A class-frame describes a group of objects with common attributes. *Animal*, *person*, *car* and *computer* are all class-frames. In AI, however, the abbreviation 'class' is often used instead of the term 'class-frame'.

Each frame in a frame-based system 'knows' its class. In other words, the frame's class is an implicit property of the frame. For example, instances in Figure 5.2 identify their class in the slot *Class*.

If objects are the basis of the frame-based systems, why bother with classes?

Grouping objects into classes helps us to represent a problem in an abstract form. Minsky himself described frames as 'data structures for representing **stereotyped** situations'. In general, we are less concerned with defining strictly and exhaustively the properties of each object, and more concerned with the salient properties typical for the entire class. Let us take, for example, the class of birds. Can a bird fly? A typical answer is yes. Almost all birds can fly, and thus we think of the ability to fly as being an essential property of the class of birds, even though there are birds, such as ostriches, which cannot fly. In other words, an eagle is a better member of the class *bird* than an ostrich because an eagle is a more typical representative of birds.

Frame-based systems support class inheritance. The fundamental idea of inheritance is that attributes of the class-frame represent things that are **typically** true for all objects in the class. However, slots in the instance-frames can be filled with actual data uniquely specified for each instance.
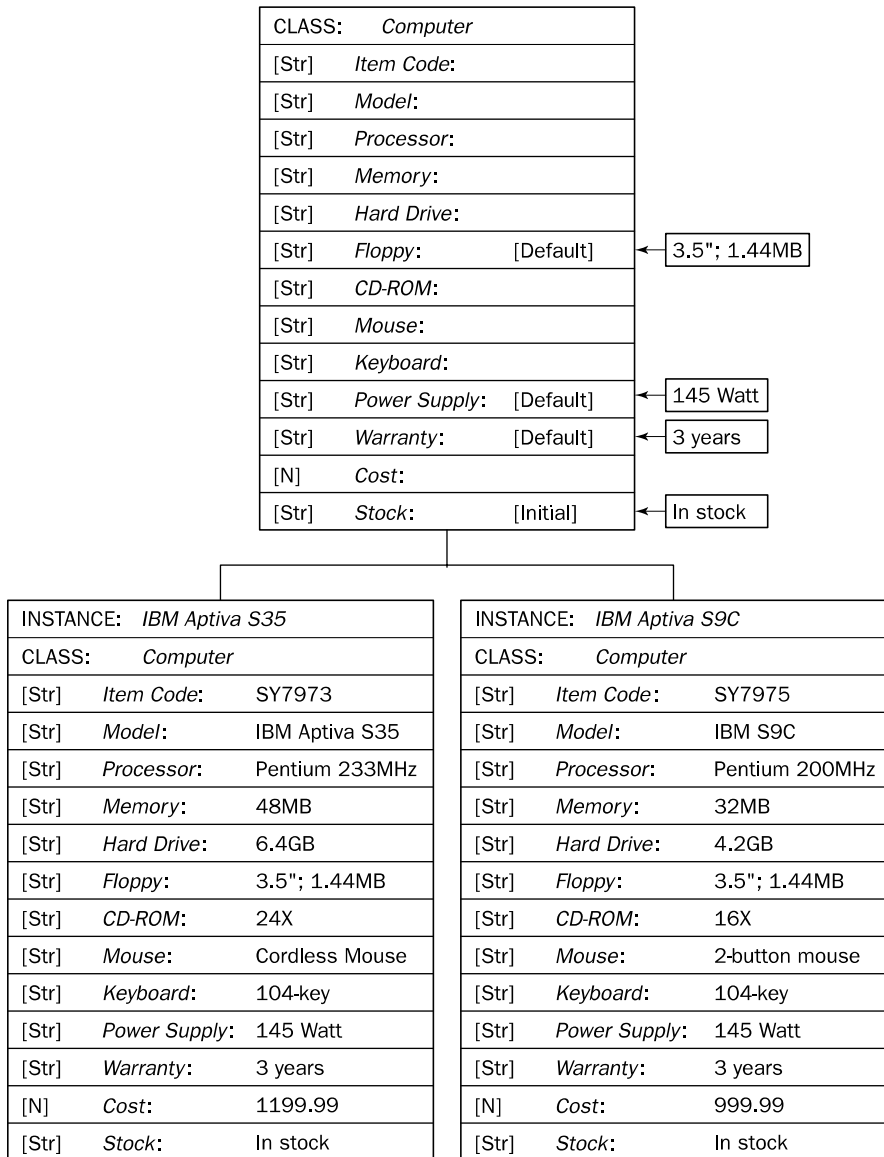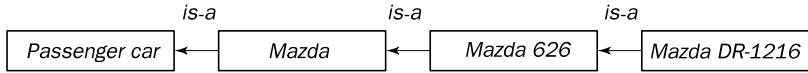
CLASS:	Computer		
[Str]	Item Code:		
[Str]	Model:		
[Str]	Processor:		
[Str]	Memory:		
[Str]	Hard Drive:		
[Str]	Floppy:	[Default]	← 3.5"; 1.44MB
[Str]	CD-ROM:		
[Str]	Mouse:		
[Str]	Keyboard:		
[Str]	Power Supply:	[Default]	← 145 Watt
[Str]	Warranty:	[Default]	← 3 years
[N]	Cost:		
[Str]	Stock:	[Initial]	← In stock

INSTANCE:	IBM Aptiva S35	
CLASS:	Computer	
[Str]	Item Code:	SY7973
[Str]	Model:	IBM Aptiva S35
[Str]	Processor:	Pentium 233MHz
[Str]	Memory:	48MB
[Str]	Hard Drive:	6.4GB
[Str]	Floppy:	3.5"; 1.44MB
[Str]	CD-ROM:	24X
[Str]	Mouse:	Cordless Mouse
[Str]	Keyboard:	104-key
[Str]	Power Supply:	145 Watt
[Str]	Warranty:	3 years
[N]	Cost:	1199.99
[Str]	Stock:	In stock

INSTANCE:	IBM Aptiva S9C	
CLASS:	Computer	
[Str]	Item Code:	SY7975
[Str]	Model:	IBM S9C
[Str]	Processor:	Pentium 200MHz
[Str]	Memory:	32MB
[Str]	Hard Drive:	4.2GB
[Str]	Floppy:	3.5"; 1.44MB
[Str]	CD-ROM:	16X
[Str]	Mouse:	2-button mouse
[Str]	Keyboard:	104-key
[Str]	Power Supply:	145 Watt
[Str]	Warranty:	3 years
[N]	Cost:	999.99
[Str]	Stock:	In stock

Figure 5.2   Computer class and instances

Consider the simple frame structure represented in Figure 5.3. The class *Passenger car* has several attributes typical for all cars. This class is too heterogeneous to have any of the attributes filled in, even though we can place certain restrictions upon such attributes as *Engine type*, *Drivetrain type* and *Transmission type*. Note that these attributes are declared as *compound* [C]. Compound attributes can assume only one value from a group of symbolic values, for

	is-a		*is-a*		*is-a*	
Passenger car	←	Mazda	←	Mazda 626	←	Mazda DR-1216

(a)

CLASS:	*Passenger car*
[C]	*Engine type* In-line 4 cylinder: V6:
[N]	*Horsepower*:
[C]	*Drivetrain type* Rear wheel drive Front wheel drive: Four wheel drive:
[C]	*Transmission type* 5-speed manual: 4-speed automatic:
[N]	*Fuel consumption (mpg)*:
[N]	*Seating capacity*:

Class:	*Mazda*
Superclass:	*Passenger car*
[C]	*Engine type* In-line 4 cylinder: V6:
[N]	*Horsepower*:
[C]	*Drivetrain type* Rear wheel drive Front wheel drive: Four wheel drive:
[C]	*Transmission type* 5-speed manual: 4-speed automatic:
[N]	*Fuel consumption (mpg)*:
[N]	*Seating capacity*:
[Str]	*Country of manufacture*: Japan

CLASS	*Mazda 626*	
Superclass:	*Mazda*	
[C]	*Engine type* In-line 4 cylinder: V6:	
[N]	*Horsepower*:	125
[C]	*Drivetrain type* Rear wheel drive Front wheel drive: Four wheel drive:	
[C]	*Transmission type* 5-speed manual: 4-speed automatic:	
[N]	*Fuel consumption (mpg)*:	22
[N]	*Seating capacity*:	5
[Str]	*Country of manufacture*:	Japan
[Str]	*Model*:	
[C]	*Colour* Glacier White: Sage Green Metallic: Slate Blue metallic: Black Onyx Clearcoat:	
[Str]	*Owner*:	

INSTANCE	*Mazda DR-1216*	
Class:	*Mazda*	
[C]	*Engine type* In-line 4 cylinder: V6:	TRUE FALSE
[N]	*Horsepower*:	125
[C]	*Drivetrain type* Rear wheel drive Front wheel drive: Four wheel drive:	FALSE TRUE FALSE
[C]	*Transmission type* 5-speed manual: 4-speed automatic:	FALSE TRUE
[N]	*Fuel consumption (mpg)*:	28
[N]	*Seating capacity*:	5
[Str]	*Country of manufacture*:	Japan
[Str]	*Model*:	DX
[C]	*Colour* Glacier White: Sage Green Metallic: Slate Blue metallic: Black Onyx Clearcoat:	FALSE TRUE FALSE FALSE
[Str]	*Owner*:	Mr Black

(b)

Figure 5.3   Inheritance of slot values in a simple frame structure: (a) relations of the car frames; (b) car frames and their slots

example the attribute *Engine type* can assume the value of either *In-line 4 cylinder* or *V6*, but not both.

The class *Mazda* is linked to its superclass *Passenger car* by the 'is-a' relation. The *Mazda* inherits all attributes of the superclass and also declares the attribute *Country of manufacture* with the default value *Japan* attached to it. The class *Mazda 626* introduces three additional attributes: *Model*, *Colour* and *Owner*. Finally, the instance-frame *Mazda DR-1216* inherits its country of manufacture from the *Mazda* frame, as the *Mazda 626* does, and establishes single values for all compound attributes.

Can an instance-frame overwrite attribute values inherited from the class-frame?

An instance-frame can overwrite, or in other words violate, some of the typical attribute values in the hierarchy. For example, the class *Mazda 626* has an average fuel consumption of 22 miles per gallon, but the instance *Mazda DR-1216* has a worse figure because it has done a lot of miles. Thus the *Mazda DR-1216* frame remains the instance of the class Mazda 626, with access to the properties further up the hierarchy, even though it violates the typical value in its class.

Relationships between frames in such a hierarchy constitute a process of specialisation. The class-frame on the top of the hierarchy represents some generic concept, class-frames further down stand for a more restricted concept and the instances are closer to exemplification.

How are objects related in a frame-based system? Is the 'is-a' relationship the only one available to us?

In general, there are three types of relationships between objects: generalisation, aggregation and association.

Generalisation denotes 'a-kind-of' or 'is-a' relationship between a superclass and its subclasses. For example, a car *is a* vehicle, or in other words, *Car* represents a subclass of the more general superclass *Vehicle*. Each subclass inherits all features of the superclass.

Aggregation is 'a-part-of' or 'part-whole' relationship in which several subclasses representing **components** are associated with a superclass representing a **whole**. For example, an engine is *a part of* a car.

Association describes some semantic relationship between different classes which are unrelated otherwise. For example, Mr Black owns a house, a car and a computer. Such classes as *House*, *Car* and *Computer* are mutually independent, but they are linked with the frame *Mr Black* through the semantic association.

Unlike generalisation and aggregation relationships, associations usually appear as verbs and are inherently bi-directional.

Does a computer own Mr Black? Of course, the name of a bi-directional association reads in a particular direction (Mr Black *owns* a computer), but this direction can be changed to the opposite. The inverse of *owns* is *belongs to*, and thus we can anticipate that a computer *belongs to* Mr Black. In fact, both directions are equally meaningful and refer to the same association.

Figure 5.4 illustrates all three types of relationships between different objects.
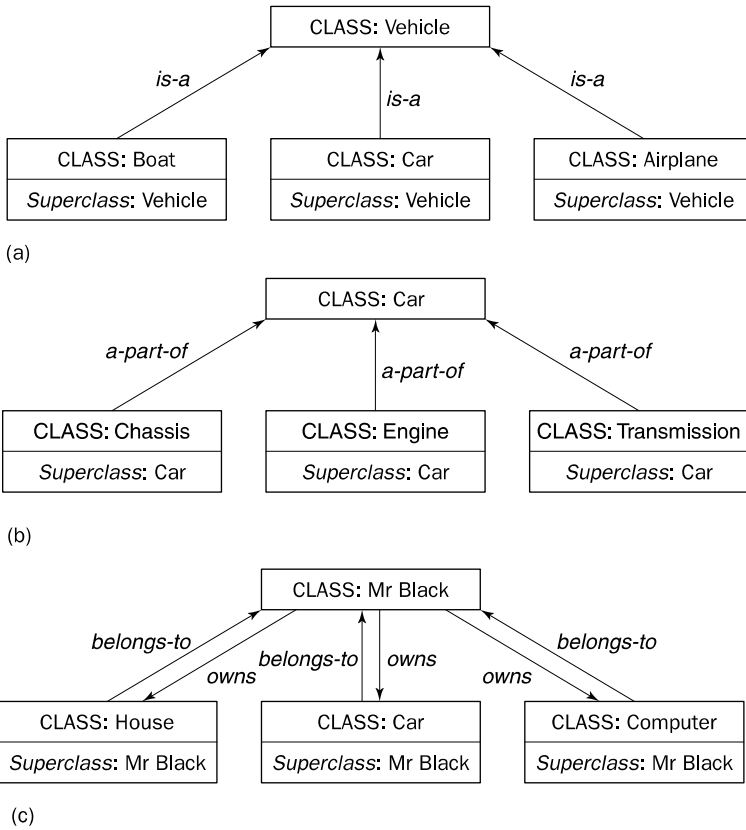
Figure 5.4   Three types of relationships among objects: (a) generalisation; (b) aggregation; (c) association

5.3   Inheritance in frame-based systems

Inheritance is an essential feature of frame-based systems. Inheritance can be defined as the process by which all characteristics of a class-frame are assumed by the instance-frame.

A common use of inheritance is to impose default features on all instance-frames. We can create just one class-frame that contains generic characteristics of some object or concept, and then obtain several instance-frames without encoding the class-level characteristics.

A hierarchical arrangement of a frame-based system can be viewed as a tree that is turned over. The highest level of abstraction is represented at the top by the root of the tree. Branches below the root illustrate lower levels of abstraction, and leaves at the bottom appear as instance-frames. Each frame inherits characteristics of all related frames at the higher levels.

Figure 5.5 shows a hierarchy of frames representing zero-emission (ZE) vehicles. The root, *ZE vehicle*, has three branches: *Electric vehicle*, *Solar vehicle*
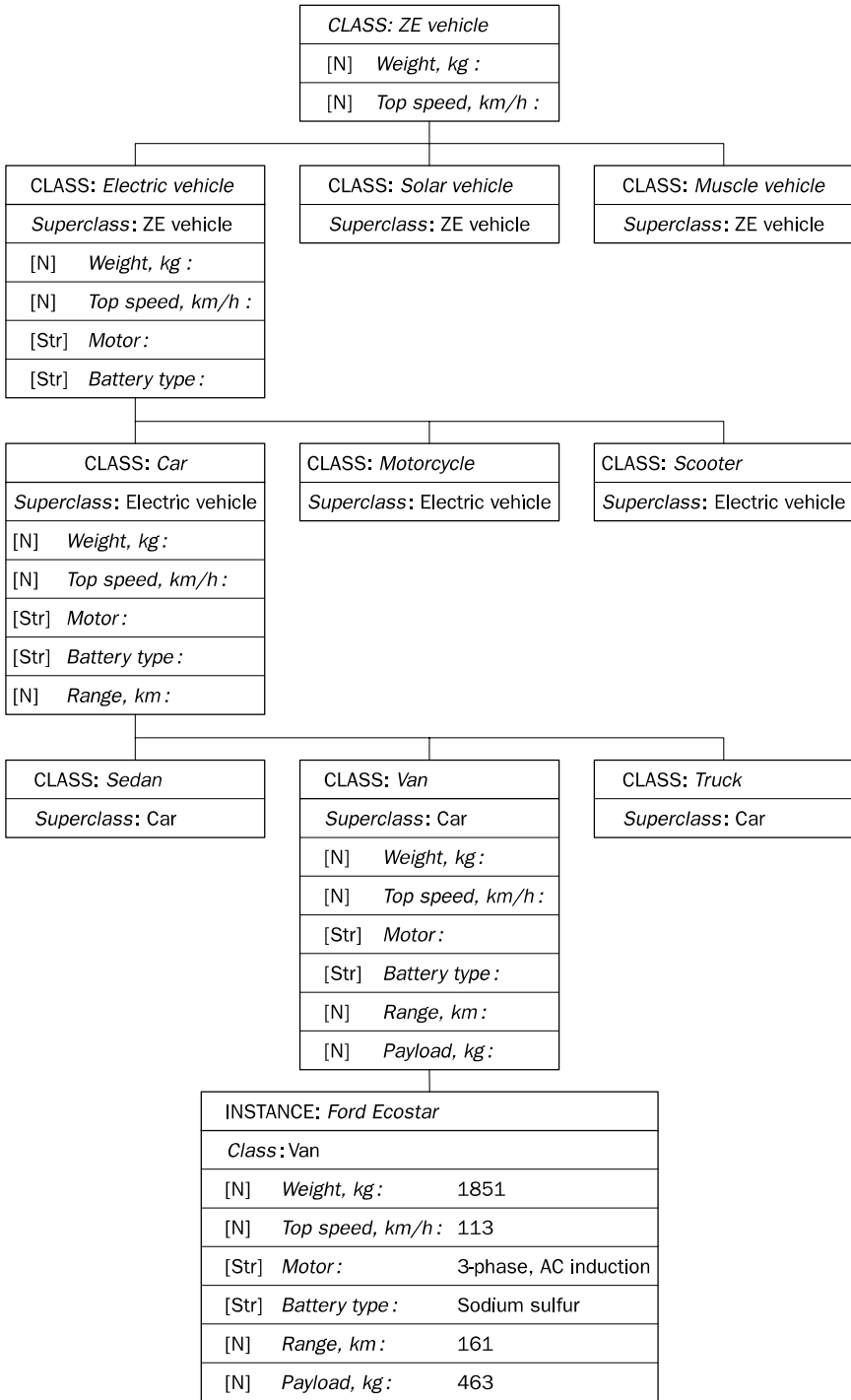
CLASS: ZE vehicle

[N]   Weight, kg :

[N]   Top speed, km/h :

CLASS: *Electric vehicle*

Superclass: ZE vehicle

[N]   Weight, kg :

[N]   Top speed, km/h :

[Str]   Motor :

[Str]   Battery type :

CLASS: *Solar vehicle*

Superclass: ZE vehicle

CLASS: *Muscle vehicle*

Superclass: ZE vehicle

CLASS: *Car*

Superclass: Electric vehicle

[N]   Weight, kg :

[N]   Top speed, km/h :

[Str]   Motor :

[Str]   Battery type :

[N]   Range, km :

CLASS: *Motorcycle*

Superclass: Electric vehicle

CLASS: *Scooter*

Superclass: Electric vehicle

CLASS: *Sedan*

Superclass: Car

CLASS: *Van*

Superclass: Car

[N]   Weight, kg :

[N]   Top speed, km/h :

[Str]   Motor :

[Str]   Battery type :

[N]   Range, km :

[N]   Payload, kg :

CLASS: *Truck*

Superclass: Car

INSTANCE: *Ford Ecostar*

Class: Van

[N]   Weight, kg :   1851

[N]   Top speed, km/h :   113

[Str]   Motor :   3-phase, AC induction

[Str]   Battery type :   Sodium sulfur

[N]   Range, km :   161

[N]   Payload, kg :   463

Figure 5.5   One-parent inheritance in the zero-emission vehicle structure

and *Muscle vehicle*. Let us now follow just one branch, the *Electric vehicle* branch. It is subdivided into *Car*, *Motorcycle* and *Scooter*. Then *Car* branches into *Sedan*, *Van* and *Truck*, and finally, the leaf, the instance-frame *Ford Ecostar*, appears at the bottom. The instance *Ford Ecostar* inherits all the characteristics of its **parent frame**.

The instance *Ford Ecostar* indeed has only one parent, the class-frame *Van*. Furthermore, in Figure 5.5, any frame except the root frame *ZE vehicle* has only one parent. In this type of structure, each frame inherits knowledge from its parent, grandparent, great-grandparent, etc.

Can a frame have more than one parent?

In many problems, it is quite natural to represent objects relating to different worlds. For example, we may wish to create a class of muscle-solar-electric vehicles. In such vehicles, people can pedal, while an electric drive system is used to travel uphill, and solar panels assist in recharging batteries for the electric system. Thus, the frame *Muscle-Solar-Electric vehicle* should combine specific properties of three classes, *Muscle vehicle*, *Solar vehicle* and *Electric vehicle*. The only requirement for multiple parent inheritance is that attributes of all parents must be uniquely specified.

In frame-based systems, several classes can use the same attribute names. However, when we use multiple inheritance, all parents must have unique attribute names. If we want, for example to create a child class *Muscle-Solar-Electric vehicle* related to parents *Muscle vehicle*, *Solar vehicle* and *Electric vehicle*, we must get rid of such properties as *Weight* and *Top speed* in the parent classes. Only then can we create the child class. In other words, to create multiple inheritance we must reconsider an entire structure of our system, as can be seen in Figure 5.6.

In frame-based systems, inheritance means code reuse, and the job of the knowledge engineer is to group similar classes together and reuse common code. The most important advantage of inheritance is the conceptual simplification, which is achieved by reducing the number of independent and specific features in the expert system.

Are there any disadvantages?

As is so often the case, much of the appealing simplicity of ideas behind the frame-based systems has been lost in the implementation stage. Brachman and Levesque (1985) argue that if we allow unrestrained overwriting of inherited properties, it may become impossible to represent either definitive statements (such as 'all squares are equilateral rectangles') or contingent universal conditions (such as 'all the squares on Kasimir Malevich's paintings are either black, red or white'). In general, frame-based systems cannot distinguish between **essential properties** (those that an instance must have in order to be considered a member of a class) and **accidental properties** (those that all the instances of a class just happen to have). Instances inherit all **typical** properties, and because those properties can be overwritten anywhere in the frame hierarchy it may become impossible to construct composite concepts when using multiple inheritance.

CLASS: *Electric vehicle*	CLASS: *Solar vehicle*	CLASS: *Muscle vehicle*
[Str] *Motor :*	[Str] *Solar panel type:*	[N] *Number of wheels :*
[Str] *Battery type :*	[Str] *Solar cell material:*	[N] *Pedalling manpower :*

CLASS: *Muscle-Solar-Electric vehicle*

Superclass: Electric vehicle
                 Solar vehicle
                 Muscle vehicle

[Str]   *Motor :*
[Str]   *Battery type :*
[Str]   *Solar panel type:*
[Str]   *Solar cell material:*
[N]   *Number of wheels :*
[N]   *Pedalling manpower :*
[N]   *Weight, kg :*
[N]   *Top speed, km/h :*

INSTANCE: *Didik Muscle Car*	
Class : Muscle-Solar-Electric vehicle	
[Str] *Motor :*	24V DC
[Str] *Battery type :*	Sealed lead acid
[Str] *Solar panel type:*	BP 140
[Str] *Solar cell material:*	Crystalline silicon
[N] *Number of wheels :*	4
[N] *Pedalling manpower :*	2
[N] *Weight, kg :*	60
[N] *Top speed, km/h :*	35

Figure 5.6   Multiple inheritance

This appears to undermine the whole idea of the frame knowledge representation. However, frames offer us a powerful tool for combining declarative and procedural knowledge, although they leave the knowledge engineer with difficult decisions to make about the hierarchical structure of the system and its inheritance paths. Appeals to so-called 'typical' properties do not always work, because they may lead us to unexpected results. Thus, although we may use frames to represent the fact that an ostrich is a bird, it is certainly not a typical bird, in the way that an eagle is. Frame-based expert systems, such as Level5 Object, provide no safeguards against creating incoherent structures. However,

such systems do provide data and control structures that are more suited for the simulation of human reasoning than any conventional programming language. Furthermore, to combine the power of both techniques of knowledge representation – rules and frames – modern frame-based expert systems use rules for interaction with information contained in the frames.

5.4   Methods and demons

As we have already discussed, frames provide us with a structural and concise means of organising knowledge. However, we expect an expert system to act as an intelligent assistant – we require it not only to store the knowledge but also to validate and manipulate this knowledge. To add actions to our frames, we need methods and demons.

What are methods and demons?

A **method** is a procedure associated with a frame attribute that is executed whenever requested (Durkin, 1994). In Level5 Object, for example, a method is represented by a series of commands similar to a macro in a spreadsheet program. We write a method for a specific attribute to determine the attribute's value or execute a series of actions when the attribute's value changes.

Most frame-based expert systems use two types of methods: WHEN CHANGED and WHEN NEEDED.

In general, a **demon** has an IF-THEN structure. It is executed whenever an attribute in the demon's IF statement changes its value. In this sense, demons and methods are very similar, and the two terms are often used as synonyms. However, methods are more appropriate if we need to write complex procedures. Demons, on the other hand, are usually limited to IF-THEN statements.

Let us now examine a WHEN CHANGED method. A WHEN CHANGED method is executed immediately when the value of its attribute changes. To understand how WHEN CHANGED methods work, we consider a simple problem adapted from Sterling and Shapiro (1994). We will use the expert system shell Level5 Object, which offers features commonly found in most frame-based expert systems and object-oriented programming languages.

The expert system is required to assist a loan officer in evaluating credit requests from small business ventures. A credit request is to be classified into one of three categories, 'Give credit', 'Deny credit' or 'Consult a superior', based on the collateral and financial rating of the business, and the bank's expected yield from the loan. When a loan officer provides a qualitative rating of the expected yield from the loan, the expert system compares the business collateral with the amount of credit requested, evaluates a financial rating based on a weighted sum of the business's net worth to assets, last year's sales growth, gross profit on sales and short-term debt to sales, and finally determines a category for the credit request.

The expert system is expected to provide details of any business venture and evaluate the credit request for the business selected by the user (a loan officer).

Figure 5.7   Input display for the request selection

The input display for the request selection is shown in Figure 5.7. The data on the display change depending on which business is selected.

The class *Action Data*, shown in Figure 5.8, is used to control the input display. The user can move to the next, previous, first or last request in the list of requests and examine the business data. The WHEN CHANGED methods here allow us to advance through a list of requests. Note that all attributes in Figure 5.8 are declared as *simple* [S]. Simple attributes can assume either a value of TRUE or FALSE. Let us examine the WHEN CHANGED method attached to the attribute *Goto Next*.

How does this method work?

In Level5 Object, any method begins with the reserved words WHEN CHANGED or WHEN NEEDED, which are followed by the reserved word BEGIN and a series of commands to be executed. The reserved word END completes a method. To refer to a particular attribute in a method, we must specify the class name as well as the attribute name. The syntax is:

   <attribute name> OF <class name>

For example, the statement *Goto Next OF Action Data* refers to the attribute *Goto Next* of the class *Action Data*.

The *Next* pushbutton on the input display is attached to the attribute *Goto Next* of the class *Action Data*. When we select this pushbutton at run time, the attribute *Goto Next* receives a value of TRUE, causing the WHEN CHANGED method attached to it to execute. The method's first command assigns the

CLASS:	*Action Data*
[S] *Goto Next*:	[WHEN CHANGED]
[S] *Goto Previous*:	[WHEN CHANGED]
[S] *Goto First*:	[WHEN CHANGED]
[S] *Goto Last*:	[WHEN CHANGED]

INSTANCE:	**Action Data 1**
CLASS:	Action Data
[S] *Goto Next*:	TRUE
[S] *Goto Previous*:	UNDETERMINED
[S] *Goto First*:	UNDETERMINED
[S] *Goto Last*:	UNDETERMINED

```
WHEN CHANGED
 BEGIN
  Current Request Number:= Request Number OF Request
  FIND Request
   LIMIT 1
   WHERE Request Number OF Request > Current Request Number
  FIND END
 END
```

```
WHEN CHANGED
 BEGIN
  Target Request Number:= Request Number OF Request – 1
  IF Target Request Number > 0 THEN
  BEGIN
   FIND Request
    LIMIT 1
    WHERE Request Number OF Request = Target Request Number
   FIND END
  END
 END
```

```
WHEN CHANGED
 BEGIN
  FIND Request
   LIMIT 1
  FIND END
 END
```

```
WHEN CHANGED
 BEGIN
  Target Request Number:= Total Number of Requests
  FIND Request
   LIMIT 1
   WHERE Request Number OF Request = Target Request Number
  FIND END
 END
```

Figure 5.8   The class *Action Data* and WHEN CHANGED methods

number of the currently selected instance of the class *Request* to the attribute *Current Request Number*, which is used as a reference point. The FIND command uses the number stored in *Current Request Number* to determine the next request in the list. The LIMIT 1 command tells Level5 Object to find the first instance that matches the search condition. The WHERE clause

WHERE Request Number OF Request > Current Request Number

locates the first instance of the class *Request* whose number is greater than the value of *Current Request Number*. The request list is maintained in increasing order to ensure that the proper instance is retrieved. If, for example, the current instance number is 6, then the FIND command will retrieve the instance with the number 7.

Let us now consider the class *Request* and its instances represented in Figure 5.9. The instances, *Request 1* and *Request 2*, have the same attributes as the class *Request*, but each instance holds specific values for these attributes. To show the attribute values on the input display, we have to create value-boxes (display items that show data) and then attach these value-boxes to the appropriate attributes. When we run the application, the value-boxes show the attribute

CLASS: *Request*
[Str] *Applicant's name*:
[Str] *Application no.*:
[N] *Requested credit*:
[N] *Currency deposits*:
[N] *Stocks*:
[N] *Mortgages*:
[N] *Net worth to assets*:
[N] *Last year's sales growth*:
[N] *Gross profits on sales*:
[N] *Short-term debt to sales*:
[C] *Expected yield*: Excellent: Reasonable: Poor:
[N] *Request Number*:

INSTANCE: **Request 1**	
CLASS: *Request*	
[Str] *Applicant's name*:	Mrs White, J.
[Str] *Application no.*:	CN001-98
[N] *Requested credit*:	50000
[N] *Currency deposits*:	50000
[N] *Stocks*:	9000
[N] *Mortgages*:	12000
[N] *Net worth to assets*:	40
[N] *Last year's sales growth*:	20
[N] *Gross profits on sales*:	45
[N] *Short-term debt to sales*:	9
[C] *Expected yield*: Excellent: TRUE Reasonable: FALSE Poor: FALSE	
[N] *Request Number*:	1

INSTANCE: **Request 2**	
CLASS: *Request*	
[Str] *Applicant's name*:	Mr Black, N.
[Str] *Application no.*:	CN002-98
[N] *Requested credit*:	75000
[N] *Currency deposits*:	45000
[N] *Stocks*:	10000
[N] *Mortgages*:	20000
[N] *Net worth to assets*:	45
[N] *Last year's sales growth*:	25
[N] *Gross profits on sales*:	35
[N] *Short-term debt to sales*:	10
[C] *Expected yield*: Excellent: FALSE Reasonable: TRUE Poor: FALSE	
[N] *Request Number*:	2

Figure 5.9   Class *Request* and its instances

values of the currently selected instance of the class *Request* and WHEN CHANGED methods cause actions to occur.

When are WHEN NEEDED methods to be used?

In many applications, an attribute is assigned to some initial or default value. However, in some applications, a WHEN NEEDED method can be used to obtain the attribute value only when it is needed. In other words, a WHEN NEEDED method is executed when information associated with a particular attribute is needed for solving the problem, but the attribute value is undetermined. We will return to this method when we discuss rules for our credit evaluation example.

5.5 Interaction of frames and rules

Most frame-based expert systems allow us to use a set of rules to evaluate information contained in frames.

Are there any specific differences between rules used in rule-based expert systems and those used in frame-based systems?

Every rule has an IF-THEN structure, and every rule relates given information or facts in its IF part to some action in its THEN part. In this sense, there are no differences between rules used in a rule-based expert system and those used in a frame-based system. However, in frame-based systems, rules often use **pattern matching** clauses. These clauses contain variables that are used for finding matching conditions among all instance-frames.

How does an inference engine work in a frame-based system? What causes rules to fire?

Let us again compare rule-based and frame-based expert systems. In a rule-based expert system, the inference engine links the rules contained in the knowledge base with data given in the database. When the goal is set up – or in other words when an expert system receives the instruction to determine a value for the specified object – the inference engine searches the knowledge base to find a rule that has the goal in its consequent (THEN part). If such a rule is found and its antecedent (IF part) matches data in the database, the rule is fired and the specified object, the goal, obtains its value. If no rules are found that can derive a value for the goal, the system queries the user to supply that value.

In a frame-based system, the inference engine also searches for the goal, or in other terms for the specified attribute, until its value is obtained.

In a rule-based expert system, the goal is defined for the rule base. In a frame-based system, rules play an auxiliary role. Frames represent here a major source of knowledge, and both methods and demons are used to add actions to the frames. Thus, we might expect that the goal in a frame-based system can be established either in a method or in a demon. Let us return to our credit evaluation example.

CLASS:	*Credit Evaluation*
[S] *Evaluate Credit* : [WHEN CHANGED]	
[C] *Collateral* : 　　　Excellent: 　　　Good: 　　　Moderate:	
[C] *Financial rating* : 　　　Excellent: 　　　Good: 　　　Medium: 　　　Bad:	
[C] *Evaluation* :　　　　[WHEN NEEDED] 　　　Give credit: 　　　Deny credit: 　　　Consult a superior:	

WHEN CHANGED
　BEGIN
　　PURSUE Evaluation OF Credit Evaluation
　END

WHEN NEEDED
　BEGIN
　　Evaluation OF Credit Evaluation IS Consult
　　superior:= TRUE
　END
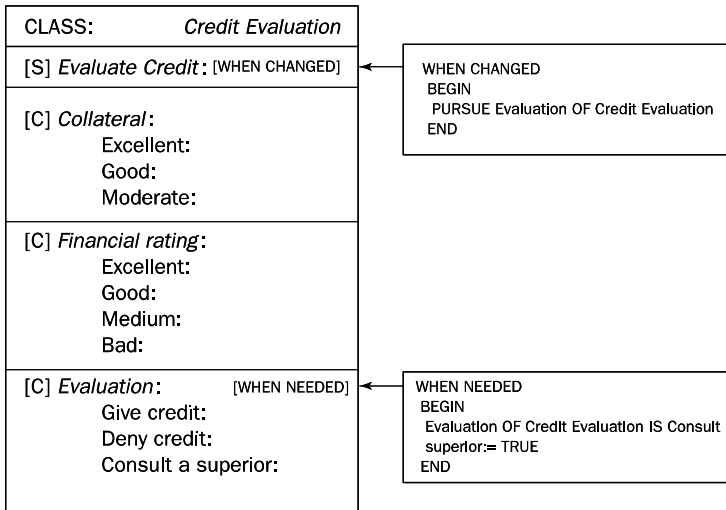
Figure 5.10　The *Credit Evaluation* class, WHEN CHANGED and WHEN NEEDED methods

Suppose we want to evaluate the credit request selected by the user. The expert system is expected to begin the evaluation when the user clicks the *Evaluate Credit* pushbutton on the input display. This pushbutton is attached to the attribute *Evaluate Credit* of the class *Credit Evaluation* shown in Figure 5.10. The attribute *Evaluate Credit* has the WHEN CHANGED method attached to it, and when we select the *Evaluate Credit* pushbutton at run time, the attribute *Evaluate Credit* receives a new value, a value of TRUE. This change causes the WHEN CHANGED method to execute. The PURSUE command tells *Level5 Object* to establish the value of the attribute *Evaluation* of the class *Credit Evaluation*. A simple set of rules shown in Figure 5.11 is used to determine the attribute's value.

How does the inference engine work here?

Based on the goal, *Evaluation OF Credit Evaluation*, the inference engine finds those rules whose consequents contain the goal of interest and examines them one at a time in the order in which they appear in the rule base. That is, the inference engine starts with RULE 9 and attempts to establish whether the attribute *Evaluation* receives the *Give credit* value. This is done by examining the validity of each antecedent of the rule. In other words, the inference engine attempts to determine first whether the attribute *Collateral* has the value of *Excellent*, and next whether the attribute *Financial rating* is *Excellent*. To determine whether *Collateral OF Credit Evaluation* is *Excellent*, the inference engine examines RULE 1 and RULE 2, and to determine whether *Financial rating OF Credit Evaluation* is *Excellent*, it looks at RULE 8. If all of the rule antecedents are valid, then the inference engine will conclude that *Evaluation OF Credit Evaluation* is *Give credit*. However, if any of the antecedents are invalid, then the conclusion is invalid. In this case, the inference engine will examine the next rule, RULE 10, which can establish a value for the attribute *Evaluation*.

RULE 1
IF Currency deposits OF Request >= Requested credit OF Request
THEN Collateral OF Credit Evaluation IS Excellent

RULE 2
IF Currency deposits OF Request >= Requested credit OF Request * 0.7
AND (Currency deposits OF Request + Stocks OF Request) >= Requested credit OF Request
THEN Collateral OF Credit Evaluation IS Excellent

RULE 3
IF (Currency deposits OF Request + Stocks OF Request) > Requested credit OF Request * 0.6
AND (Currency deposits OF Request + Stocks OF Request) < Requested credit OF Request * 0.7
AND (Currency deposits OF Request + Stocks OF Request + Mortgages OF Request) >= Requested credit OF Request
THEN Collateral OF Credit Evaluation IS Good

RULE 4
IF (Currency deposits OF Request + Stocks OF Request + Mortgages OF Request) <= Requested credit OF Request
THEN Collateral OF Credit Evaluation IS Moderate

RULE 5
IF Net worth to assets OF Request * 5 + Last year's sales growth OF Request + Gross profits on sales OF Request * 5 + Short term debt to sales OF Request * 2 <= -500
THEN Financial rating OF Credit Evaluation IS Bad

RULE 6
IF Net worth to assets OF Request * 5 + Last year's sales growth OF Request + Gross Profits on sales OF Request * 5 + Short term debt to sales OF Request * 2 >= -500
AND Net worth to assets OF Request * 5 + Last year's sales growth OF Request + Gross profits on sales OF Request * 5 + Short term debt to sales OF Request * 2 <= 150
THEN Financial rating OF Credit Evaluation IS Medium

RULE 7
IF Net worth to assets OF Request * 5 + Last year's sales growth OF Request + Gross Profits on sales OF Request * 5 + Short term debt to sales OF Request * 2 >= 150
AND Net worth to assets OF Request * 5 + Last year's sales growth OF Request + Gross profits on sales OF Request * 5 + Short term debt to sales OF Request * 2 <= 1000
THEN Financial rating OF Credit Evaluation IS Good

RULE 8
IF Net worth to assets OF Request * 5 + Last year's sales growth OF Request + Gross Profits on sales OF Request * 5 + Short term debt to sales OF Request * 2 > 1000
THEN Financial rating OF Credit Evaluation IS Excellent

RULE 9
IF Collateral OF Credit Evaluation IS Excellent
AND Financial rating OF Credit Evaluation IS Excellent
THEN Evaluation OF Credit Evaluation IS Give Credit

RULE 10
IF Collateral OF Credit Evaluation IS Excellent
AND Financial rating OF Credit Evaluation IS Good
THEN Evaluation OF Credit Evaluation IS Give Credit

RULE 11
IF Collateral OF Credit Evaluation IS Moderate
AND Financial rating OF Credit Evaluation IS Medium
THEN Evaluation OF Credit Evaluation IS Deny Credit

RULE 12
IF Collateral OF Credit Evaluation IS Moderate
AND Financial rating OF Credit Evaluation IS Bad
THEN Evaluation OF Credit Evaluation IS Deny Credit

Figure 5.11   Rules for credit evaluation

What happens if *Collateral OF Credit Evaluation* is *Good*?

Based on the set of rules provided for credit evaluation, the inference engine cannot establish the value of the attribute *Evaluation* in some cases. This is especially true when the collateral is good and the financial rating of the business is excellent or good. In fact, if we have a look at Figure 5.10, we find

cases that are not represented in the rule base. However, it is not necessary always to rely on a set of rules. We can use the WHEN NEEDED method to establish the attribute value.

The WHEN NEEDED method shown in Figure 5.10 is attached to the attribute *Evaluation*. The inference engine executes this method when it needs to determine the value of *Evaluation*. When the WHEN NEEDED method is executed, the attribute *Evaluation* receives the value *Consult a superior*.

How does the inference engine know where, and in what order, to obtain the value of an attribute?

In our case, if the WHEN NEEDED method were executed first, the attribute *Evaluation* would always receive the value *Consult a superior*, and no rules would ever be fired. Thus, the inference engine has to obtain the value from the WHEN NEEDED method only if it has not been determined from the rule base. In other words, the search order for the attribute value has to be determined first. It can be done, for example, by means of the SEARCH ORDER facet attached to an attribute that tells the inference engine where, and in what order, to obtain the value of this attribute.

In Level5 Object, a search order can be specified for every attribute, and in our credit evaluation example, we set the search order for the Evaluation value to RULES, WHEN NEEDED. It makes certain that the inference engine starts the search from the rule base.

5.6   Buy Smart: a frame-based expert system

To illustrate the ideas discussed above, we consider a simple frame-based expert system, Buy Smart, which advises property buyers.

We will review the main steps in developing frame-based systems, and show how to use methods and demons to bring frames to life. To aid us in this effort we will use the Level5 Object expert system shell.

Are there any differences between the main steps in building a rule-based expert system and a frame-based one?

The basic steps are essentially the same. First, the knowledge engineer needs to obtain a general understanding of the problem and the overall knowledge structure. He or she then decides which expert system tool to use for developing a prototype system. Then the knowledge engineer actually creates the knowledge base and tests it by running a number of consultations. And finally, the expert system is expanded, tested and revised until it does what the user wants it to do.

The principal difference between the design of a rule-based expert system and a frame-based one lies in how the knowledge is viewed and represented in the system.

In a rule-based system, a set of rules represents the domain knowledge useful for problem solving. Each rule captures some heuristic of the problem, and each new rule adds some new knowledge and thus makes the system smarter. The rule-based system can easily be modified by changing, adding or subtracting rules.

In a frame-based system, the problem is viewed in a different manner. Here, the overall hierarchical structure of the knowledge is decided first. Classes and their attributes are identified, and hierarchical relationships between frames are established. The architecture of a frame-based system should not only provide a natural description of the problem, but also allow us to add actions to the frames through methods and demons.

The development of a frame-based system typically involves the following steps:

1   Specify the problem and define the scope of the system.

2   Determine classes and their attributes.

3   Define instances.

4   Design displays.

5   Define WHEN CHANGED and WHEN NEEDED methods, and demons.

6   Define rules.

7   Evaluate and expand the system.

Step 1:   *Specify the problem and define the scope of the system*

In our Buy Smart example, we start by collecting some information about properties for sale in our region. We can identify relevant details such as the property type, location, number of bedrooms and bathrooms, and of course, the property price. We also should provide a short description and a nice photo for each property.

We expect that some of the properties will be sold and new properties will appear on the market. Thus, we need to build a database that can be easily modified and then accessed from the expert system. Level5 Object allows us to access, modify, delete and perform other actions on data within a dBASE III database.

Can we store descriptions and pictures of the properties within a database?

Property descriptions and pictures should be stored separately, descriptions as text files (*.txt) and pictures as bit-map files (*.bmp). If we then set up a display that includes a text-box and a picture-box, we will be able to view a property description and its picture in this display by reading the text file into the text-box and the bit-map file into the picture-box, respectively.

Now we create an external database file, *house.dbf*, using dBASE III or Microsoft Excel, as shown in Table 5.1.

The next step is to list all possible queries we might think of:

• What is the maximum amount you want to spend on a property?

• What type of property do you prefer?

• Which suburb would you like to live in?

• How many bedrooms do you want?

• How many bathrooms do you want?

Table 5.1 The property database *house.dbf*

Area	Suburb	Price	Type	Bedrooms
Central Suburbs	New Town	164000	House	3
Central Suburbs	Taroona	150000	House	3
Southern Suburbs	Kingston	225000	Townhouse	4
Central Suburbs	North Hobart	127000	House	3
Northern Suburbs	West Moonah	89500	Unit	2
Central Suburbs	Taroona	110000	House	3
Central Suburbs	Lenah Valley	145000	House	3
Eastern Shore	Old Beach	79500	Unit	2
Central Suburbs	South Hobart	140000	House	3
Central Suburbs	South Hobart	115000	House	3
Eastern Shore	Cambridge	94500	Unit	2
Northern Suburbs	Glenorchy	228000	Townhouse	4
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Bathrooms	Construction	Phone	Pictfile	Textfile
1	Weatherboard	(03) 6226 4212	house01.bmp	house01.txt
1	Brick	(03) 6226 1416	house02.bmp	house02.txt
2	Brick	(03) 6229 4200	house03.bmp	house03.txt
1	Brick	(03) 6226 8620	house04.bmp	house04.txt
1	Weatherboard	(03) 6225 4666	house05.bmp	house05.txt
1	Brick	(03) 6229 5316	house06.bmp	house06.txt
1	Brick	(03) 6278 2317	house07.bmp	house07.txt
1	Brick	(03) 6249 7298	house08.bmp	house08.txt
1	Brick	(03) 6228 5460	house09.bmp	house09.txt
1	Brick	(03) 6227 8937	house10.bmp	house10.txt
1	Brick	(03) 6248 1459	house11.bmp	house11.txt
2	Weatherboard	(03) 6271 6347	house12.bmp	house12.txt
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Once these queries are answered, the expert system is expected to provide a list of suitable properties.

Step 2: *Determine classes and their attributes*

Here, we identify the problem's principal classes. We begin with the general or conceptual type of classes. For example, we can talk about the concept of a **property** and describe general features that are common to most properties. We can characterise each property by its location, price, type, number of bedrooms and bathrooms, construction, picture and description. We also need to present contact details of the property, such as its address or phone number. Thus, the class

CLASS: **Property**
[Str] *Area*:
[Str] *Suburb*:
[N]   *Price*:
[Str] *Type*:
[N]   *Bedrooms*:
[N]   *Bathrooms*:
[Str] *Construction*:
[Str] *Phone*:
[Str] *Pictfile*:
[Str] *Textfile*:
[N]   *Instance Number*:

INSTANCE: **Property 1**	
CLASS: *Property*	
[Str] *Area*:	Central Suburbs
[Str] *Suburb*:	New Town
[N] *Price*:	164000
[Str] *Type*:	House
[N] *Bedrooms*:	3
[N] *Bathrooms*:	1
[Str] *Construction*:	Weatherboard
[Str] *Phone*:	(03) 6226 4212
[Str] *Pictfile*:	house01.bmp
[Str] *Textfile*:	house01.txt
[N] *Instance Number :*	1

INSTANCE: **Property 2**	
CLASS: *Property*	
[Str] *Area*:	Central Suburbs
[Str] *Suburb*:	Taroona
[N] *Price*:	150000
[Str] *Type*:	House
[N] *Bedrooms*:	3
[N] *Bathrooms*:	1
[Str] *Construction*:	Brick
[Str] *Phone*:	(03) 6226 1416
[Str] *Pictfile*:	house02.bmp
[Str] *Textfile:*	house02.txt
[N] *Instance Number*:	2

Figure 5.12   Class *Property* and its instances

Property can be presented as shown in Figure 5.12. Note that we added the attribute *Instance Number* as well. This attribute does not characterise the property but will assist Level5 Object in accessing the external database.

Step 3:   *Define instances*

Once we determined the class-frame *Property*, we can easily create its instances by using data stored in the dBASE III database. For most frame-based expert systems like Level5 Object, this task requires us to

tell the system that we want a new instance to be created. For example, to create a new instance of the class *Property*, we can use the following code:

```
MAKE Property
    WITH Area := area OF dB3 HOUSE 1
    WITH Suburb := suburb OF dB3 HOUSE 1
    WITH Price := price OF dB3 HOUSE 1
    WITH Type := type OF dB3 HOUSE 1
    WITH Bedrooms := bedrooms OF dB3 HOUSE 1
    WITH Bathrooms := bathrooms OF dB3 HOUSE 1
    WITH Construction := construct OF dB3 HOUSE 1
    WITH Phone := phone OF dB3 HOUSE 1
    WITH Pictfile := pictfile OF dB3 HOUSE 1
    WITH Textfile := textfile OF dB3 HOUSE 1
    WITH Instance Number := Current Instance Number
```

Here, the class *dB3 HOUSE 1* is used to represent the structure of the external database file *house.dbf*. Each row in the property database, shown in Table 5.1, represents an instance of the class *Property*, and each column represents an attribute. A newly created instance-frame receives the values of the current record of the database. Figure 5.12 shows instances that are created from the external database. These instances are linked to the class *Property*, and they inherit all attributes of this class.

Step 4: *Design displays*

Once the principal classes and their attributes are determined, we can design major displays for our application. We need the *Application Title Display* to present some general information to the user at the beginning of each application. This display may consist of the application title, general description of the problem, representative graphics and also copyright information. An example of the *Application Title Display* is shown in Figure 5.13.

The next display we can think of is the *Query Display*. This display should allow us to indicate our preferences by answering the queries presented by the expert system. The *Query Display* may look like a display shown in Figure 5.14. Here, the user is asked to select the most important things he or she is looking for in the property. Based on these selections, the expert system will then come up with a complete list of suitable properties.

And finally, we should design the *Property Information Display*. This display has to provide us with the list of suitable properties, an opportunity to move to the next, previous, first or last property in the list, and also a chance to look at the property picture and its description. Such a display may look like the one presented in Figure 5.15. Note that the picture-box and text-box are included in the display.

Figure 5.13 The *Application Title Display*
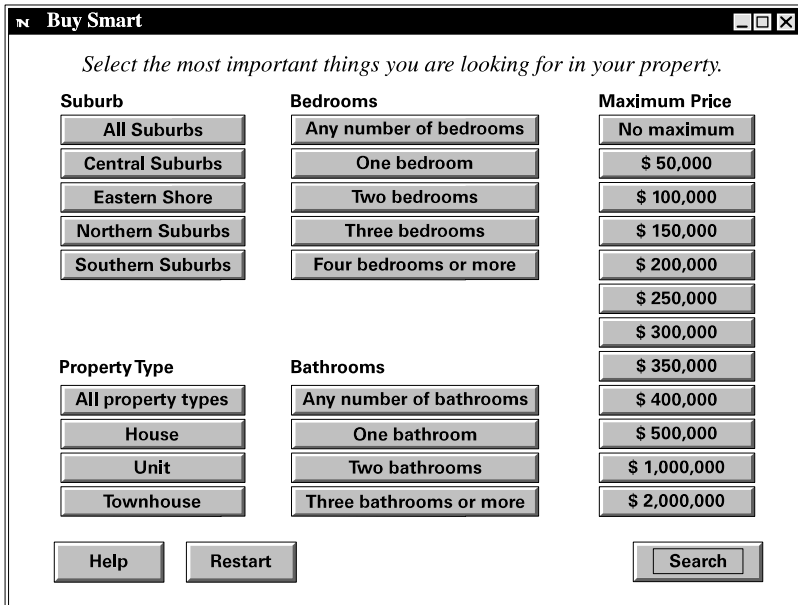
Figure 5.14 The *Query Display*
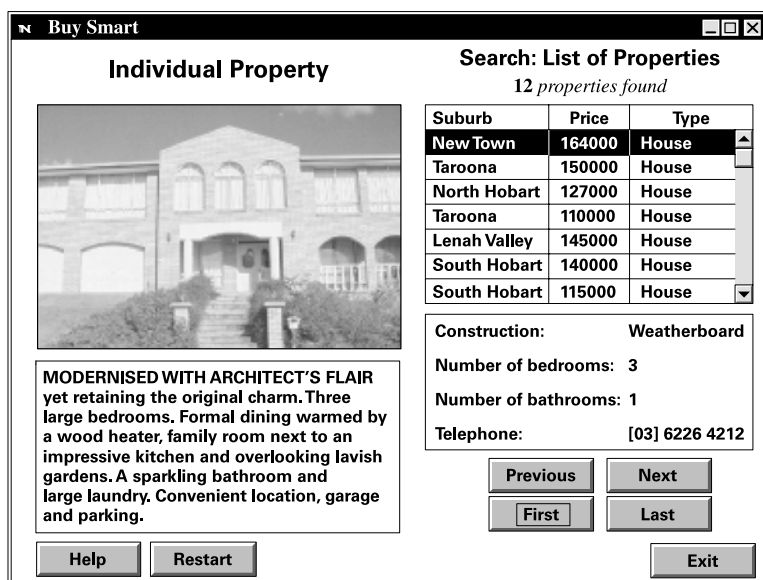
Figure 5.15   The *Property Information Display*

How are these displays linked?

Level5 Object allows us to link these displays by attaching the *Continue* pushbutton on the *Application Title Display* to the *Query Display*, and the *Search* pushbutton on the *Query Display* to the *Property Information Display*. When we run the application, clicking on either the *Continue* or *Search* pushbutton will cause a new display to appear.

Now we have to bring these displays to life.

Step 5:   ***Define WHEN CHANGED and WHEN NEEDED methods, and demons***

At this point, we have already created the problem principal classes and their attributes. We also determined the class instances, and established the mechanism for creating these instances from the external database. And finally, we designed static displays for presenting information to the user. We must now develop a way to bring our application to life. There are two ways to accomplish this task. The first one relies on WHEN CHANGED and WHEN NEEDED methods, and demons. The second approach involves pattern-matching rules. In frame-based systems, we always first consider an application of methods and demons.

What we need now is to decide when to create instances of the class *Property*. There are two possible solutions. The first one is to create all instances at once when the user clicks on the *Continue* pushbutton on the *Application Title Display*, and then remove inappropriate instances step-by-step based on the user's preferences when he or she selects pushbuttons on the *Query Display*.

The second approach is to create only relevant instances after the user has made all selections on the *Query Display*. This approach

illuminates the necessity to remove inappropriate instances of the class *Property*, but may add to the complexity of the system's design.

In our design here, we give preference to the first approach. It will provide us with an opportunity to use demons instead of rules. However, you could use the other approach.

Let us now create an additional class, the class *Action Data*, shown in Figure 5.16. The WHEN CHANGED method attached to the attribute *Load Properties* allows us to create all instances of the class *Property*.

How do we make this method work?

To make it work, we attach the *Continue* pushbutton on the *Application Title Display* to the attribute *Load Properties*. Now when we select this pushbutton at run time, the attribute *Load Properties* receives a value of TRUE, causing its WHEN CHANGED method to execute and create all instances of the class *Property*. The number of the instances created equals the number of records in the external database.

Now the *Query Display* appears (remember that we attach the *Continue* pushbutton of the *Application Title Display* to the *Query Display*), and the user is required to choose the most desirable features of the property by selecting appropriate pushbuttons. Each pushbutton here is associated with a demon that removes inappropriate instances of the class *Properties*. A set of demons is shown in Figure 5.17.

How do demons work here?

A demon does not go into action until something happens. In our application, it means that a demon is fired only if the user selects a corresponding pushbutton.
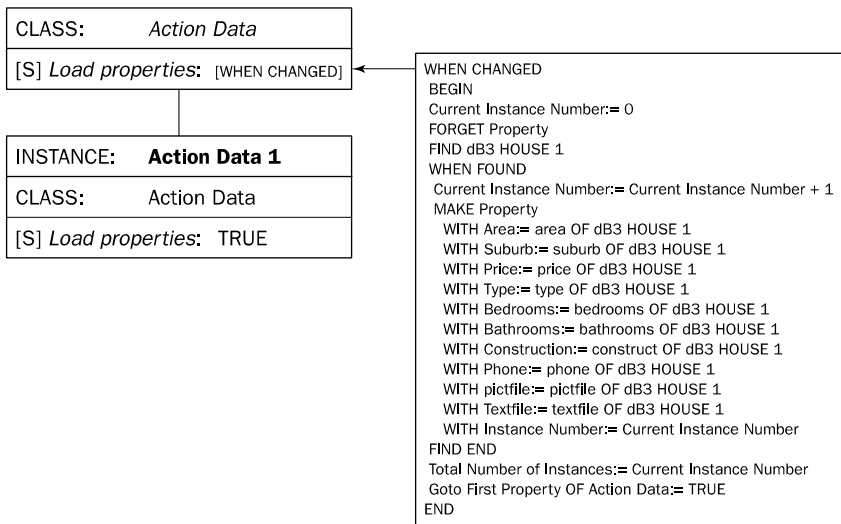
Figure 5.16 The WHEN CHANGED method of the attribute *Load Properties*

```
DEMON 1
IF selected OF Central Suburbs pushbutton
THEN FIND Property
   WHERE Area OF Property <> "Central Suburbs"
   WHEN FOUND
      FORGET CURRENT Property
 FIND END

DEMON 2
IF selected OF Eastern Shore pushbutton
THEN FIND Property
   WHERE Area OF Property <> "Eastern Shore"
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
       .
       .
       .
DEMON 5
IF selected OF House pushbutton
THEN FIND Property
   WHERE Type OF Property <> "House"
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
       .
       .
       .
DEMON 9
IF selected OF One bedroom pushbutton
THEN FIND Property
   WHERE Bedrooms OF Property <> 1
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
       .
       .
       .
DEMON 12
IF selected OF One bathroom pushbutton
THEN FIND Property
   WHERE Bathrooms OF Property <> 1
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
       .
       .
       .
DEMON 15
IF selected OF $50000 pushbutton
THEN FIND Property
   WHERE Price OF Property > 50000
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
```

Figure 5.17   Demons for the *Query Display*

Let us consider, for example, DEMON 1 associated with the *Central Suburbs* pushbutton. When the user clicks on the *Central Suburbs* pushbutton on the *Query Display*, DEMON 1 is fired. The first command of the demon consequent tells Level5 Object to find the class *Property*. The WHERE clause,

WHERE Area OF Property <> ''Central Suburbs''

finds all instances of the class *Property* that do not match the user selection. It looks for any instance where the value of the attribute

Area is not equal to *Central Suburbs*. Then, the FORGET CURRENT command removes the current instance of the class *Property* from the application.

Once the property features are selected, the user clicks on the *Search* pushbutton on the *Query Display* to obtain a list of properties with these features. This list will appear on the *Property Information Display* (recall that the *Search* pushbutton is attached to the *Property Information Display*).

Can we view pictures and descriptions of the properties?

Let us first create two more attributes, *Load Instance Number* and *Goto First Property*, for the class *Action Data* as shown in Figure 5.18. Let us also attach the *Search* pushbutton on the *Query Display* to the attribute *Load Instance Number*. Now when we click on the *Search* pushbutton at run time, the attribute *Load Instance Number* will receive a value of TRUE, causing its WHEN CHANGED method to execute. This method determines the total number of instances left in the class *Property*. It also assigns the attribute *Goto First Property* a value of TRUE, subsequently causing its WHEN CHANGED method to execute.

The method attached to the attribute *Goto First Property* ensures that we are always positioned at the first property when we enter the *Property Information Display*. It also loads the value of the attribute *Pictfile* into the display's picture-box and the value of *Textfile* into the text-box. As a result, we can see the property picture and its description as shown in Figure 5.15.
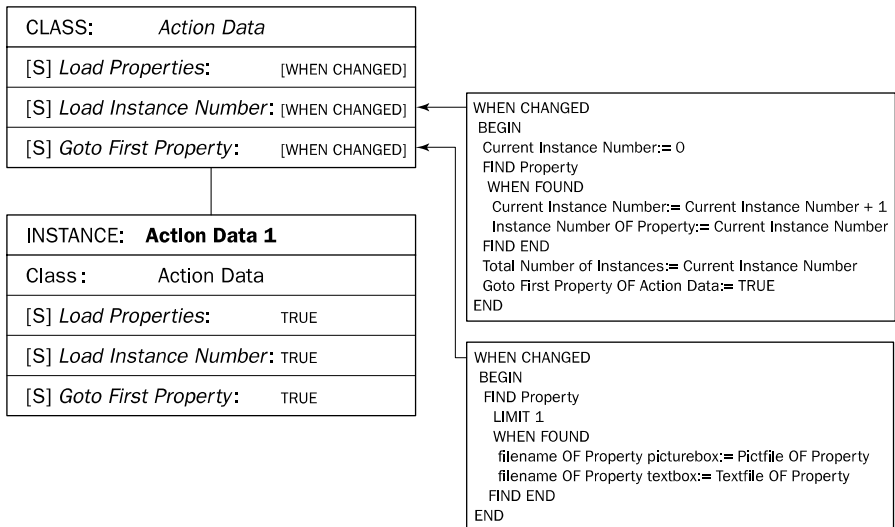
Figure 5.18   The WHEN CHANGED methods of the attributes *Load Instance Number* and *Goto First Property*

Step 6: *Define rules*

When we design a frame-based expert system, one of the most important and difficult decisions is whether to use rules or manage with methods and demons instead. This decision is usually based on the personal preferences of the designer. In our application, we use methods and demons because they offer us a powerful but simple way of representing procedures. On the other hand, in the credit evaluation example considered earlier, we applied a set of rules. In general, however, rules are not effective at dealing with procedural knowledge.

Step 7: *Evaluate and expand the system*

We have now completed the initial design of our Buy Smart expert system. The next task is to evaluate it. We want to make sure that the system's performance meets our expectations. In other words, we want to run a test case.

1. To begin the test, we click on the *Continue* pushbutton on the *Application Title Display*. The attribute *Load Properties* of the class *Action Data* receives a value of TRUE. The WHEN CHANGED method attached to *Load Properties* is executed, and all instances of the class *Property* are created.

2. The *Query Display* appears, and we make our selections, for example:

⇒ *Central Suburbs*

```
DEMON 1
IF selected OF Central Suburbs pushbutton
THEN FIND Property
   WHERE Area OF Property <> ''Central Suburbs''
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
```

⇒ *House*

```
DEMON 5
IF selected OF House pushbutton
THEN FIND Property
   WHERE Type OF Property <> ''House''
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
```

⇒ *Three bedrooms*

```
DEMON 10
IF selected OF Three bedroom pushbutton
THEN FIND Property
   WHERE Bedrooms OF Property <> 3
   WHEN FOUND
      FORGET CURRENT Property
 FIND END
```

⇒ *One bathroom*

DEMON 12
IF selected OF One bathroom pushbutton
THEN FIND Property
   WHERE Bathrooms OF Property <> 1
   WHEN FOUND
     FORGET CURRENT Property
  FIND END

⇒ *$ 200,000*

DEMON 18
IF selected OF $200,000 pushbutton
THEN FIND Property
   WHERE Price OF Property > 200000
   WHEN FOUND
     FORGET CURRENT Property
  FIND END

The demons remove those *Property* instances whose features do not match our selections.

3. Now we click on the *Search* pushbutton. The attribute *Load Instance Number* of the class *Action Data* receives a value of TRUE. The WHEN CHANGED method attached to *Load Instance Number* is executed. It determines the number of instances left in the class *Property*, and also assigns the attribute *Goto First Property* a value of TRUE. Now the WHEN CHANGED method attached to *Goto First Property* is executed. It finds the first *Property* instance, and assigns the attribute *filename* of the *Property picturebox* a value of house01.bmp, and the attribute *filename* of the *Property textbox* a value of house01.txt (recall that both the *Property picturebox* and *Property textbox* have been created on the *Property Information Display*).

4. The *Property Information Display* appears. In the example shown in Figure 5.15, we can examine 12 properties that satisfy our requirements. Note that we are positioned at the first property in the property list, the property picture appears in the picture-box and the property description in the text-box. However, we cannot move to the next, previous or last property in the property list by using the pushbuttons assigned on the display. To make them functional, we need to create additional attributes in the class *Action Data*, and then attach WHEN CHANGED methods as shown in Figure 5.19.

Now the Buy Smart expert system is ready for expansion, and we can add new properties to the external database.
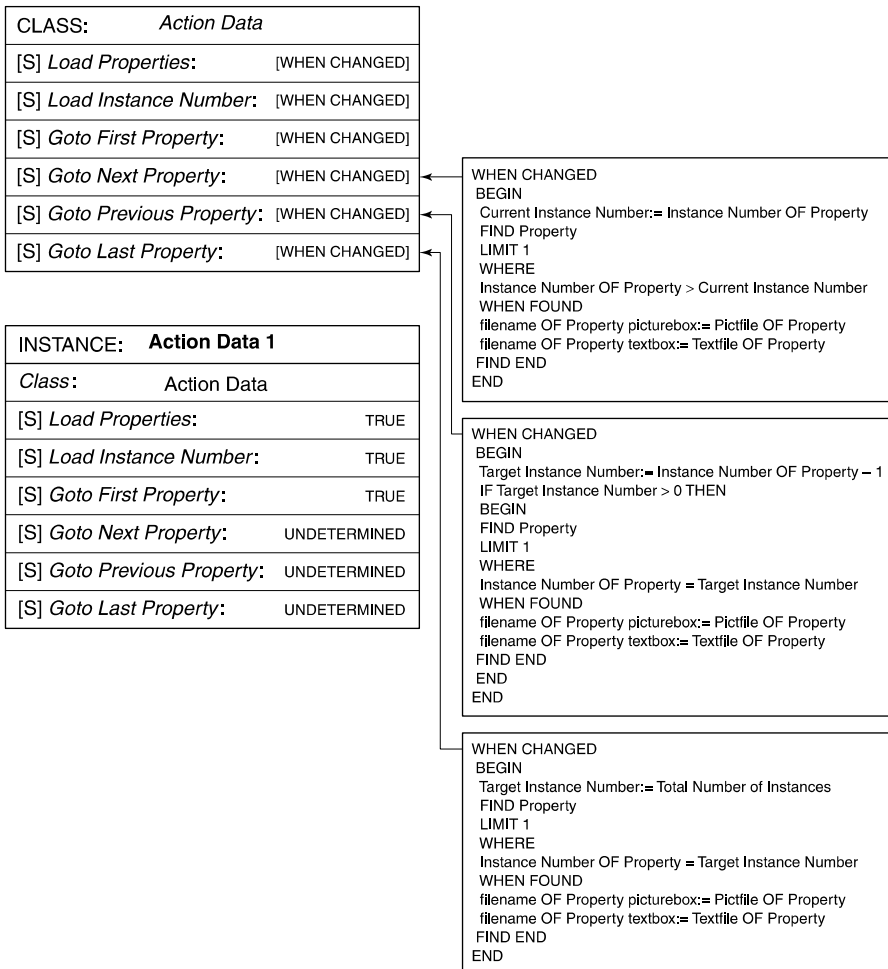
Figure 5.19   The WHEN CHANGED methods of the attributes *Goto Next Property*, *Goto Previous Property* and *Goto Last Property*

5.7   Summary

In this chapter, we presented an overview of frame-based expert systems. We considered the concept of a frame and discussed how to use frames for knowledge representation. We found that inheritance is an essential feature of the frame-based systems. We examined the application of methods, demons and rules. Finally, we considered the development of a frame-based expert system through an example.

The most important lessons learned in this chapter are:

- A frame is a data structure with typical knowledge about a particular object or concept.

- Frames are used to represent knowledge in a frame-based expert system. A frame contains knowledge of a given object, including its name and a set of attributes also called slots. *Name*, *weight*, *height* and *age* are attributes of the frame *Person*. *Model*, *processor*, *memory* and *price* are attributes of the frame *Computer*.

- Attributes are used to store values. An attribute may contain a default value or a pointer to another frame, set of rules or procedure by which the attribute value is obtained.

- Frame-based systems can also extend the attribute-value structure through the application of facets. Facets are used to establish the attribute value, control end-user queries, and tell the inference engine how to process the attribute.

- A frame may refer to a group of similar objects, or to a particular object. A class-frame describes a group of objects with common attributes. *Animal*, *person*, *car* and *computer* are all class-frames. An instance-frame describes a particular object.

- Frame-based systems support class inheritance, i.e. the process by which all characteristics of a class-frame are assumed by the instance-frame. The fundamental idea of inheritance is that attributes of the class-frame represent things that are **typically** true for all objects in the class, but slots in the instance-frames are filled with actual data that is unique for each instance.

- A frame can inherit attributes from more than one parent through multiple-parent inheritance.

- Frames communicate with each other by methods and demons. A method is a procedure associated with a frame attribute; it is executed whenever requested. Most frame-based expert systems use two types of methods: WHEN CHANGED and WHEN NEEDED. The WHEN CHANGED method is executed when new information is placed in the slot, and the WHEN NEEDED method is executed when information is needed for solving the problem but the slot value is unspecified.

- Demons are similar to methods, and the terms are often used as synonyms. However, methods are more appropriate if we need to write complex procedures. Demons, on the other hand, are usually limited to IF-THEN statements.

- In frame-based expert systems, rules often use **pattern matching** clauses. These clauses contain variables that are used for locating matching conditions among all instance-frames.

- Although frames provide a powerful tool for combining declarative and procedural knowledge, they leave the knowledge engineer with difficult decisions about the hierarchical structure of the system and its inheritance paths.

Questions for review

1 What is a frame? What are the class and instances? Give examples.

2 Design the class-frame for the object *Student*, determine its attributes and define several instances for this class.

3 What is a facet? Give examples of various types of facets.

4 What is the correct level of decomposition of a problem into frames, slots and facets? Justify your answer through an example.

5 How are objects related in frame-based systems? What are the 'a-kind-of' and 'a-part-of' relationships? Give examples.

6 Define inheritance in frame-based systems. Why is inheritance an essential feature of the frame-based systems?

7 Can a frame inherit attributes from more than one parent? Give an example.

8 What is a method? What are the most popular types of methods used in frame-based expert systems?

9 What is a demon? What are the differences between demons and methods?

10 What are the differences, if any, between rules used in rule-based expert systems and those used in frame-based systems?

11 What are the main steps in developing a frame-based expert system?

12 List some advantages of frame-based expert systems. What are the difficulties involved in developing a frame-based expert system?

References

Brachman, R.J. and Levesque, H.J. (1985). *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, CA.

Durkin, J. (1994). *Expert Systems Design and Development*. Prentice Hall, Englewood Cliffs, NJ.

Minsky, M. (1975). A framework for representing knowledge, *The Psychology of Computer Vision*, P. Winston, ed., McGraw-Hill, New York, pp. 211–277.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991). *Object-oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, NJ.

Sterling, L. and Shapiro E. (1994). *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA.

Taylor, D. (1992). *Object-Oriented Information Systems*. John Wiley, New York.

Bibliography

Aikens, J.S. (1984). A representation scheme using both frames and rules, *Rule-Based Expert Systems*, B.G. Buchanan and E.H. Shortliffe, eds, Addison-Wesley, Reading, MA, pp. 424–440.

Alpert, S.R., Woyak, S.W., Shrobe, H.J. and Arrowood, L.F. (1990). Guest Editors Introduction: Object-oriented programming in AI, *IEEE Expert*, 5(6), 6–7.

Brachman, R.J. and Levesque, H.J. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann, Los Altos, CA.

Budd, T. (2002). *An Introduction to Object Oriented Programming*, 3rd edn. Addison-Wesley, Reading, MA.

Fikes, R. and Kehler, T. (1985). The role of frame-based representation in reasoning, *Communications of the ACM*, 28(9), 904–920.

Goldstein, I. and Papert, S. (1977). Artificial intelligence, language, and the study of knowledge, *Cognitive Science*, 1(1), 84–123.

Jackson, P. (1999). *Introduction to Expert Systems*, 3rd edn. Addison-Wesley, Harlow.

Levesque, H.J. and Brachman, R.J. (1985). A fundamental trade-off in knowledge representation and reasoning. *Readings in Knowledge Representation*, R.J. Brachman and H.J. Levesque, eds, Morgan Kaufmann, Los Altos, CA.

Luger, G.F. (2002). *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 4th edn. Addison-Wesley, Harlow.

Payne, E.C. and McArthur, R.C. (1990). *Developing Expert Systems: A Knowledge Engineer's Handbook for Rules and Objects*. John Wiley, New York.

Rosson, M.B. and Alpert, S.R. (1990). The cognitive consequences of object-oriented design, *Human–Computer Interaction*, 5(4), 345–79.

Stefik, M.J. (1979). An examination of frame-structured representation systems, *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, Tokyo, Japan, August 1979, pp. 845–852.

Stefik, M.J. (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann, San Francisco, CA.

Stefik, M.J. and Bobrow, D.G. (1986). Object-oriented programming: themes and variations, *Al Magazine*, 6(4), 40–62.

Touretzky, D.S. (1986). *The Mathematics of Inheritance Systems*. Morgan Kaufmann, Los Altos, CA.

Waterman, D.A. (1986). *A Guide to Expert Systems*. Addison-Wesley, Reading, MA.

Winston, P.H. (1977). Representing knowledge in frames, Chapter 7 of *Artificial Intelligence*, Addison-Wesley, Reading, MA, pp. 181–187.

Winston, P.H. (1992). *Artificial Intelligence*, 3rd edn. Addison-Wesley, Reading, MA.

Artificial neural networks

6

In which we consider how our brains work and how to build and train artificial neural networks.

6.1 Introduction, or how the brain works

'The computer hasn't proved anything yet,' angry Garry Kasparov, the world chess champion, said after his defeat in New York in May 1997. 'If we were playing a real competitive match, I would tear down Deep Blue into pieces.'

But Kasparov's efforts to downplay the significance of his defeat in the six-game match was futile. The fact that Kasparov – probably the greatest chess player the world has seen – was beaten by a computer marked a turning point in the quest for intelligent machines.

The IBM supercomputer called Deep Blue was capable of analysing 200 million positions a second, and it appeared to be displaying intelligent thoughts. At one stage Kasparov even accused the machine of cheating!

> 'There were many, many discoveries in this match, and one of them was that sometimes the computer plays very, very human moves.
>
> It deeply understands positional factors. And that is an outstanding scientific achievement.'

Traditionally, it has been assumed that to beat an expert in a chess game, a computer would have to formulate a strategy that goes beyond simply doing a great number of 'look-ahead' moves per second. Chess-playing programs must be able to improve their performance with experience or, in other words, a machine must be capable of learning.

What is machine learning?

In general, machine learning involves adaptive mechanisms that enable computers to learn from experience, learn by example and learn by analogy. Learning capabilities can improve the performance of an intelligent system over time. Machine learning mechanisms form the basis for adaptive systems. The most popular approaches to machine learning are **artificial neural networks** and **genetic algorithms**. This chapter is dedicated to neural networks.

What is a neural network?

A neural network can be defined as a model of reasoning based on the human brain. The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called **neurons**. The human brain incorporates nearly 10 billion neurons and 60 trillion connections, **synapses**, between them (Shepherd and Koch, 1990). By using multiple neurons simultaneously, the brain can perform its functions much faster than the fastest computers in existence today.

Although each neuron has a very simple structure, an army of such elements constitutes a tremendous processing power. A neuron consists of a cell body, **soma**, a number of fibres called **dendrites**, and a single long fibre called the **axon**. While dendrites branch into a network around the soma, the axon stretches out to the dendrites and somas of other neurons. Figure 6.1 is a schematic drawing of a neural network.

Signals are propagated from one neuron to another by complex electro-chemical reactions. Chemical substances released from the synapses cause a change in the electrical potential of the cell body. When the potential reaches its threshold, an electrical pulse, **action potential**, is sent down through the axon. The pulse spreads out and eventually reaches synapses, causing them to increase or decrease their potential. However, the most interesting finding is that a neural network exhibits **plasticity**. In response to the stimulation pattern, neurons demonstrate long-term changes in the strength of their connections. Neurons also can form new connections with other neurons. Even entire collections of neurons may sometimes migrate from one place to another. These mechanisms form the basis for learning in the brain.

Our brain can be considered as a highly complex, nonlinear and parallel information-processing system. Information is stored and processed in a neural network simultaneously throughout the whole network, rather than at specific locations. In other words, in neural networks, both data and its processing are **global** rather than local.

Owing to the plasticity, connections between neurons leading to the 'right answer' are strengthened while those leading to the 'wrong answer' weaken. As a result, neural networks have the ability to learn through experience.

Learning is a fundamental and essential characteristic of biological neural networks. The ease and naturalness with which they can learn led to attempts to emulate a biological neural network in a computer.
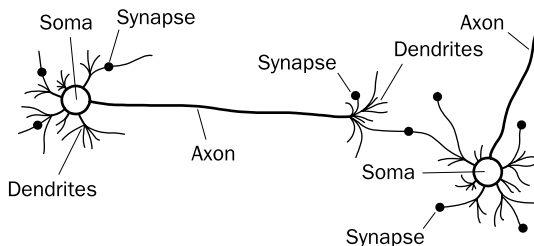
Figure 6.1 Biological neural network

Although a present-day artificial neural network (ANN) resembles the human brain much as a paper plane resembles a supersonic jet, it is a big step forward. ANNs are capable of 'learning', that is, they use experience to improve their performance. When exposed to a sufficient number of samples, ANNs can generalise to others they have not yet encountered. They can recognise hand-written characters, identify words in human speech, and detect explosives at airports. Moreover, ANNs can observe patterns that human experts fail to recognise. For example, Chase Manhattan Bank used a neural network to examine an array of information about the use of stolen credit cards – and discovered that the most suspicious sales were for women's shoes costing between $40 and $80.

How do artificial neural nets model the brain?

An artificial neural network consists of a number of very simple and highly interconnected processors, also called neurons, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links passing signals from one neuron to another. Each neuron receives a number of input signals through its connections; however, it never produces more than a single output signal. The output signal is transmitted through the neuron's outgoing connection (corresponding to the biological axon). The outgoing connection, in turn, splits into a number of branches that transmit the same signal (the signal is not divided among these branches in any way). The outgoing branches terminate at the incoming connections of other neurons in the network. Figure 6.2 represents connections of a typical ANN, and Table 6.1 shows the analogy between biological and artificial neural networks (Medsker and Liebowitz, 1994).

How does an artificial neural network 'learn'?

The neurons are connected by **links**, and each link has a **numerical weight** associated with it. Weights are the basic means of long-term memory in ANNs. They express the strength, or in other words importance, of each neuron input. A neural network 'learns' through repeated adjustments of these weights.
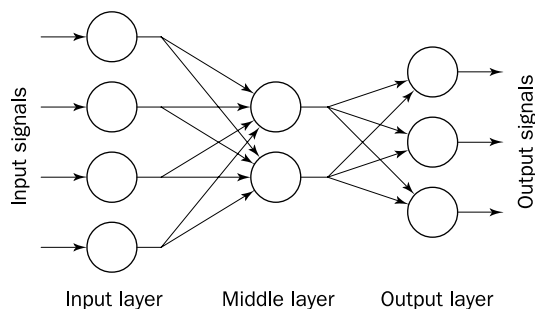
Figure 6.2 Architecture of a typical artificial neural network

Table 6.1   Analogy between biological and artificial neural networks

Biological neural network	Artificial neural network
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight

But does the neural network know how to adjust the weights?

As shown in Figure 6.2, a typical ANN is made up of a hierarchy of layers, and the neurons in the networks are arranged along these layers. The neurons connected to the external environment form input and output layers. The weights are modified to bring the network input/output behaviour into line with that of the environment.

Each neuron is an elementary information-processing unit. It has a means of computing its **activation level** given the inputs and numerical weights.

To build an artificial neural network, we must decide first how many neurons are to be used and how the neurons are to be connected to form a network. In other words, we must first choose the network architecture. Then we decide which learning algorithm to use. And finally we train the neural network, that is, we initialise the weights of the network and update the weights from a set of training examples.

Let us begin with a neuron, the basic building element of an ANN.

6.2   The neuron as a simple computing element

A neuron receives several signals from its input links, computes a new activation level and sends it as an output signal through the output links. The input signal can be raw data or outputs of other neurons. The output signal can be either a final solution to the problem or an input to other neurons. Figure 6.3 shows a typical neuron.
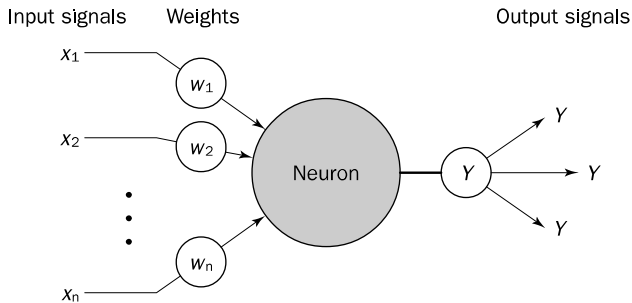
Figure 6.3   Diagram of a neuron

How does the neuron determine its output?

In 1943, Warren McCulloch and Walter Pitts proposed a very simple idea that is still the basis for most artificial neural networks.

The neuron computes the weighted sum of the input signals and compares the result with a threshold value, θ. If the net input is less than the threshold, the neuron output is -1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value $+1$ (McCulloch and Pitts, 1943).

In other words, the neuron uses the following transfer or **activation function**:

$$X = \sum_{i=1}^{n} x_i w_i \tag{6.1}$$

$$Y = \begin{cases} +1 & \text{if } X \geqslant \theta \\ -1 & \text{if } X < \theta \end{cases}$$

where X is the net weighted input to the neuron, x_i is the value of input i, w_i is the weight of input i, n is the number of neuron inputs, and Y is the output of the neuron.

This type of activation function is called a **sign function**.

Thus the actual output of the neuron with a sign activation function can be represented as

$$Y = sign\left[\sum_{i=1}^{n} x_i w_i - \theta\right] \tag{6.2}$$

Is the sign function the only activation function used by neurons?

Many activation functions have been tested, but only a few have found practical applications. Four common choices – the step, sign, linear and sigmoid functions – are illustrated in Figure 6.4.

The **step** and **sign** activation functions, also called **hard limit functions**, are often used in decision-making neurons for classification and pattern recognition tasks.

Step function	Sign function	Sigmoid function	Linear function

$Y^{step} = \begin{cases} 1, & \text{if } X \geq 0 \\ 0, & \text{if } X < 0 \end{cases}$ $Y^{sign} = \begin{cases} +1, & \text{if } X \geq 0 \\ -1, & \text{if } X < 0 \end{cases}$ $Y^{sigmoid} = \dfrac{1}{1 + e^{-X}}$ $Y^{linear} = X$
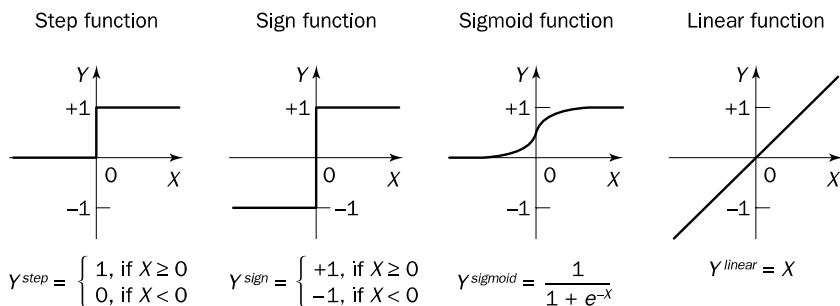
Figure 6.4   Activation functions of a neuron
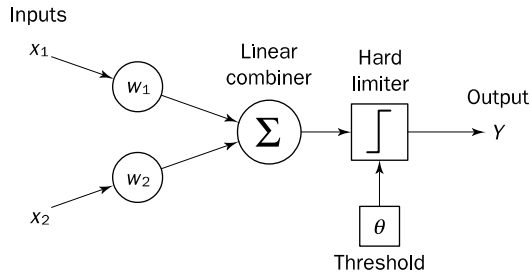
Figure 6.5   Single-layer two-input perceptron

The **sigmoid function** transforms the input, which can have any value between plus and minus infinity, into a reasonable value in the range between 0 and 1. Neurons with this function are used in the back-propagation networks.

The **linear activation function** provides an output equal to the neuron weighted input. Neurons with the linear function are often used for linear approximation.

Can a single neuron learn a task?

In 1958, Frank Rosenblatt introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron** (Rosenblatt, 1958). The perceptron is the simplest form of a neural network. It consists of a single neuron with **adjustable** synaptic weights and a **hard limiter**. A single-layer two-input perceptron is shown in Figure 6.5.

6.3   The perceptron

The operation of Rosenblatt's perceptron is based on the McCulloch and Pitts neuron model. The model consists of a linear combiner followed by a hard limiter. The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to $+1$ if its input is positive and -1 if it is negative. The aim of the perceptron is to classify inputs, or in other words externally applied stimuli $x_1, x_2, \ldots, x_n$, into one of two classes, say A_1 and A_2. Thus, in the case of an elementary perceptron, the n-dimensional space is divided by a **hyperplane** into two decision regions. The hyperplane is defined by the **linearly separable** function

$$\sum_{i=1}^{n} x_i w_i - \theta = 0 \tag{6.3}$$

For the case of two inputs, x_1 and x_2, the decision boundary takes the form of a straight line shown in bold in Figure 6.6(a). Point 1, which lies above the boundary line, belongs to class A_1; and point 2, which lies below the line, belongs to class A_2. The threshold θ can be used to shift the decision boundary.
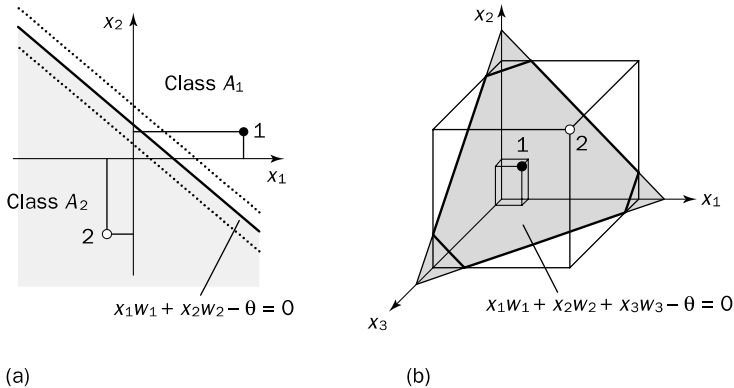
Figure 6.6   Linear separability in the perceptrons: (a) two-input perceptron;
(b) three-input perceptron

With three inputs the hyperplane can still be visualised. Figure 6.6(b) shows three dimensions for the three-input perceptron. The separating plane here is defined by the equation

$$x_1 w_1 + x_2 w_2 + x_3 w_3 - \theta = 0$$

But how does the perceptron learn its classification tasks?

This is done by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range $[-0.5, 0.5]$, and then updated to obtain the output consistent with the training examples. For a perceptron, the process of weight updating is particularly simple. If at iteration p, the actual output is $Y(p)$ and the desired output is $Y_d(p)$, then the error is given by

$$e(p) = Y_d(p) - Y(p) \qquad \text{where } p = 1, 2, 3, \ldots \tag{6.4}$$

Iteration p here refers to the pth training example presented to the perceptron.

If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$. Taking into account that each perceptron input contributes $x_i(p) \times w_i(p)$ to the total input $X(p)$, we find that if input value $x_i(p)$ is positive, an increase in its weight $w_i(p)$ tends to increase perceptron output $Y(p)$, whereas if $x_i(p)$ is negative, an increase in $w_i(p)$ tends to decrease $Y(p)$. Thus, the following **perceptron learning rule** can be established:

$$w_i(p + 1) = w_i(p) + \alpha \times x_i(p) \times e(p), \tag{6.5}$$

where α is the **learning rate**, a positive constant less than unity.

The perceptron learning rule was first proposed by Rosenblatt in 1960 (Rosenblatt, 1960). Using this rule we can derive the perceptron training algorithm for classification tasks.

Step 1: *Initialisation*

Set initial weights $w_1, w_2, \ldots, w_n$ and threshold θ to random numbers in the range $[-0.5, 0.5]$.

Step 2: *Activation*

Activate the perceptron by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired output $Y_d(p)$. Calculate the actual output at iteration $p = 1$

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p)w_i(p) - \theta\right], \tag{6.6}$$

where n is the number of the perceptron inputs, and *step* is a step activation function.

Step 3: *Weight training*

Update the weights of the perceptron

$$w_i(p + 1) = w_i(p) + \Delta w_i(p), \tag{6.7}$$

where $\Delta w_i(p)$ is the weight correction at iteration p.

The weight correction is computed by the **delta rule**:

$$\Delta w_i(p) = \alpha \times x_i(p) \times e(p) \tag{6.8}$$

Step 4: *Iteration*

Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

Can we train a perceptron to perform basic logical operations such as AND, OR or Exclusive-OR?

The truth tables for the operations AND, OR and Exclusive-OR are shown in Table 6.2. The table presents all possible combinations of values for two variables, x_1 and x_2, and the results of the operations. The perceptron must be trained to classify the input patterns.

Let us first consider the operation AND. After completing the initialisation step, the perceptron is activated by the sequence of four input patterns representing an **epoch**. The perceptron weights are updated after each activation. This process is repeated until all the weights converge to a uniform set of values. The results are shown in Table 6.3.

Table 6.2 Truth tables for the basic logical operations

Input variables		AND	OR	Exclusive-OR
x_1	x_2	$x_1 \cap x_2$	$x_1 \cup x_2$	$x_1 \oplus x_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 6.3 Example of perceptron learning: the logical operation AND

Epoch	Inputs x_1	x_2	Desired output Y_d	Initial weights w_1	w_2	Actual output Y	Error e	Final weights w_1	w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	−1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	−1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$.

In a similar manner, the perceptron can learn the operation OR. However, a single-layer perceptron cannot be trained to perform the operation Exclusive-OR.

A little geometry can help us to understand why this is. Figure 6.7 represents the AND, OR and Exclusive-OR functions as two-dimensional plots based on the values of the two inputs. Points in the input space where the function output is 1 are indicated by black dots, and points where the output is 0 are indicated by white dots.

(a) AND $(x_1 \cap x_2)$ (b) OR $(x_1 \cup x_2)$ (c) Exclusive-OR $(x_1 \oplus x_2)$
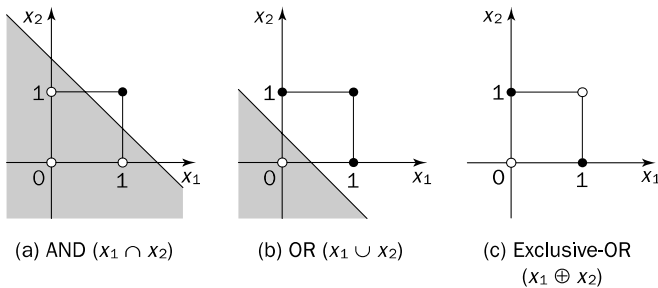
Figure 6.7 Two-dimensional plots of basic logical operations

In Figures 6.7(a) and (b), we can draw a line so that black dots are on one side and white dots on the other, but dots shown in Figure 6.7(c) are not separable by a single line. A perceptron is able to represent a function only if there is some line that separates all the black dots from all the white dots. Such functions are called **linearly separable**. Therefore, a perceptron can learn the operations AND and OR, but not Exclusive-OR.

But why can a perceptron learn only linearly separable functions?

The fact that a perceptron can learn only linearly separable functions directly follows from Eq. (6.1). The perceptron output Y is 1 only if the total weighted input X is greater than or equal to the threshold value θ. This means that the entire input space is divided in two along a boundary defined by $X = \theta$. For example, a separating line for the operation AND is defined by the equation

$$x_1 w_1 + x_2 w_2 = \theta$$

If we substitute values for weights w_1 and w_2 and threshold θ given in Table 6.3, we obtain one of the possible separating lines as

$$0.1x_1 + 0.1x_2 = 0.2$$

or

$$x_1 + x_2 = 2$$

Thus, the region below the boundary line, where the output is 0, is given by

$$x_1 + x_2 - 2 < 0,$$

and the region above this line, where the output is 1, is given by

$$x_1 + x_2 - 2 \geqslant 0$$

The fact that a perceptron can learn only linear separable functions is rather bad news, because there are not many such functions.

Can we do better by using a sigmoidal or linear element in place of the hard limiter?

Single-layer perceptrons make decisions in the same way, regardless of the activation function used by the perceptron (Shynk, 1990; Shynk and Bershad, 1992). It means that a single-layer perceptron can classify only linearly separable patterns, regardless of whether we use a hard-limit or soft-limit activation function.

The computational limitations of a perceptron were mathematically analysed in Minsky and Papert's famous book *Perceptrons* (Minsky and Papert, 1969). They proved that Rosenblatt's perceptron cannot make global generalisations on the basis of examples learned locally. Moreover, Minsky and Papert concluded that

the limitations of a single-layer perceptron would also hold true for multilayer neural networks. This conclusion certainly did not encourage further research on artificial neural networks.

How do we cope with problems which are not linearly separable?

To cope with such problems we need multilayer neural networks. In fact, history has proved that the limitations of Rosenblatt's perceptron can be overcome by advanced forms of neural networks, for example multilayer perceptrons trained with the back-propagation algorithm.

6.4 Multilayer neural networks

A multilayer perceptron is a feedforward neural network with one or more hidden layers. Typically, the network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons. The input signals are propagated in a forward direction on a layer-by-layer basis. A multilayer perceptron with two hidden layers is shown in Figure 6.8.

But why do we need a hidden layer?

Each layer in a multilayer neural network has its own specific function. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. Actually, the input layer rarely includes computing neurons, and thus does not process input patterns. The output layer accepts output signals, or in other words a stimulus pattern, from the hidden layer and establishes the output pattern of the entire network.

Neurons in the hidden layer detect the features; the weights of the neurons represent the features hidden in the input patterns. These features are then used by the output layer in determining the output pattern.

With one hidden layer, we can represent any continuous function of the input signals, and with two hidden layers even discontinuous functions can be represented.
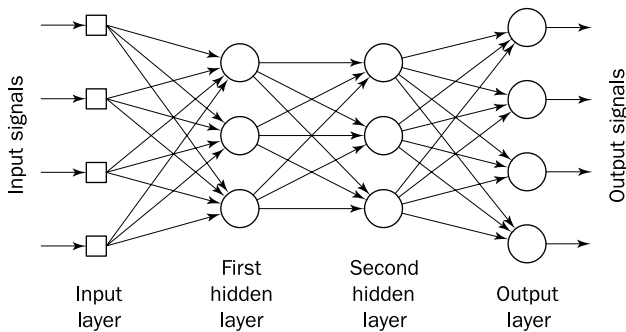
Figure 6.8   Multilayer perceptron with two hidden layers

Why is a middle layer in a multilayer network called a 'hidden' layer? What does this layer hide?

A hidden layer 'hides' its desired output. Neurons in the hidden layer cannot be observed through the input/output behaviour of the network. There is no obvious way to know what the desired output of the hidden layer should be. In other words, the desired output of the hidden layer is determined by the layer itself.

Can a neural network include more than two hidden layers?

Commercial ANNs incorporate three and sometimes four layers, including one or two hidden layers. Each layer can contain from 10 to 1000 neurons. Experimental neural networks may have five or even six layers, including three or four hidden layers, and utilise millions of neurons, but most practical applications use only three layers, because each additional layer increases the computational burden exponentially.

How do multilayer neural networks learn?

More than a hundred different learning algorithms are available, but the most popular method is back-propagation. This method was first proposed in 1969 (Bryson and Ho, 1969), but was ignored because of its demanding computations. Only in the mid-1980s was the back-propagation learning algorithm rediscovered.

Learning in a multilayer network proceeds the same way as for a perceptron. A training set of input patterns is presented to the network. The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error.

In a perceptron, there is only one weight for each input and only one output. But in the multilayer network, there are many weights, each of which contributes to more than one output.

How can we assess the blame for an error and divide it among the contributing weights?

In a back-propagation neural network, the learning algorithm has two phases. First, a training input pattern is presented to the network input layer. The network then propagates the input pattern from layer to layer until the output pattern is generated by the output layer. If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

As with any other neural network, a back-propagation one is determined by the connections between neurons (the network's architecture), the activation function used by the neurons, and the learning algorithm (or the learning law) that specifies the procedure for adjusting weights.

Typically, a back-propagation network is a multilayer network that has three or four layers. The layers are **fully connected**, that is, every neuron in each layer is connected to every other neuron in the adjacent forward layer.

A neuron determines its output in a manner similar to Rosenblatt's percep-
tron. First, it computes the net weighted input as before:

$$X = \sum_{i=1}^{n} x_i w_i - \theta,$$

where n is the number of inputs, and θ is the threshold applied to the neuron.

Next, this input value is passed through the activation function. However,
unlike a percepron, neurons in the back-propagation network use a sigmoid
activation function:

$$Y^{sigmoid} = \frac{1}{1 + e^{-X}} \tag{6.9}$$

The derivative of this function is easy to compute. It also guarantees that the
neuron output is bounded between 0 and 1.

What about the learning law used in the back-propagation networks?

To derive the back-propagation learning law, let us consider the three-layer
network shown in Figure 6.9. The indices i, j and k here refer to neurons in the
input, hidden and output layers, respectively.

Input signals, $x_1, x_2, \ldots, x_n$, are propagated through the network from left to
right, and error signals, $e_1, e_2, \ldots, e_l$, from right to left. The symbol w_{ij} denotes the
weight for the connection between neuron i in the input layer and neuron j in
the hidden layer, and the symbol w_{jk} the weight between neuron j in the hidden
layer and neuron k in the output layer.
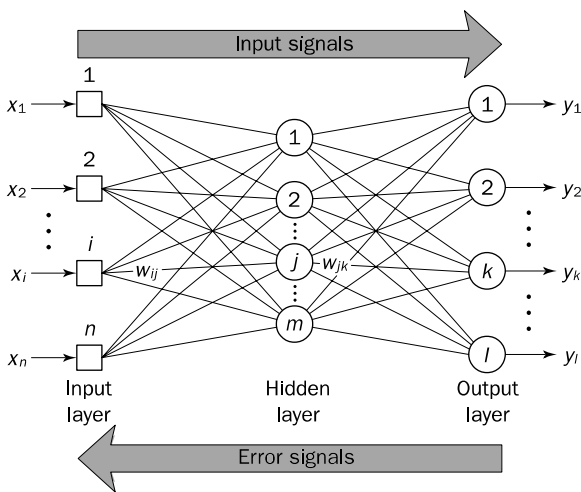
Figure 6.9 Three-layer back-propagation neural network

To propagate error signals, we start at the output layer and work backward to the hidden layer. The error signal at the output of neuron k at iteration p is defined by

$$e_k(p) = y_{d,k}(p) - y_k(p), \tag{6.10}$$

where $y_{d,k}(p)$ is the desired output of neuron k at iteration p.

Neuron k, which is located in the output layer, is supplied with a desired output of its own. Hence, we may use a straightforward procedure to update weight w_{jk}. In fact, the rule for updating weights at the output layer is similar to the perceptron learning rule of Eq. (6.7):

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p), \tag{6.11}$$

where $\Delta w_{jk}(p)$ is the weight correction.

When we determined the weight correction for the perceptron, we used input signal x_i. But in the multilayer network, the inputs of neurons in the output layer are different from the inputs of neurons in the input layer.

As we cannot apply input signal x_i, what should we use instead?

We use the output of neuron j in the hidden layer, y_j, instead of input x_i. The weight correction in the multilayer network is computed by (Fu, 1994):

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p), \tag{6.12}$$

where $\delta_k(p)$ is the error gradient at neuron k in the output layer at iteration p.

What is the error gradient?

The error gradient is determined as the derivative of the activation function multiplied by the error at the neuron output.

Thus, for neuron k in the output layer, we have

$$\delta_k(p) = \frac{\partial y_k(p)}{\partial X_k(p)} \times e_k(p), \tag{6.13}$$

where $y_k(p)$ is the output of neuron k at iteration p, and $X_k(p)$ is the net weighted input to neuron k at the same iteration.

For a sigmoid activation function, Eq. (6.13) can be represented as

$$\delta_k(p) = \frac{\partial \left\{ \dfrac{1}{1 + \exp[-X_k(p)]} \right\}}{\partial X_k(p)} \times e_k(p) = \frac{\exp[-X_k(p)]}{\{1 + \exp[-X_k(p)]\}^2} \times e_k(p)$$

Thus, we obtain:

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p), \tag{6.14}$$

where

$$y_k(p) = \frac{1}{1 + \exp[-X_k(p)]}.$$

How can we determine the weight correction for a neuron in the hidden layer?

To calculate the weight correction for the hidden layer, we can apply the same equation as for the output layer:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p), \tag{6.15}$$

where $\delta_j(p)$ represents the error gradient at neuron j in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) w_{jk}(p),$$

where l is the number of neurons in the output layer;

$$y_j(p) = \frac{1}{1 + e^{-X_j(p)}};$$

$$X_j(p) = \sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j;$$

and n is the number of neurons in the input layer.

Now we can derive the back-propagation training algorithm.

Step 1: *Initialisation*

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range (Haykin, 1999):

$$\left(-\frac{2.4}{F_i}, +\frac{2.4}{F_i} \right),$$

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Step 2: *Activation*

Activate the back-propagation neural network by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired outputs $y_{d,1}(p), y_{d,2}(p), \ldots, y_{d,n}(p)$.

(a) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = sigmoid \left[\sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j \right],$$

where n is the number of inputs of neuron j in the hidden layer, and *sigmoid* is the sigmoid activation function.

(b) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = sigmoid \left[\sum_{j=1}^{m} x_{jk}(p) \times w_{jk}(p) - \theta_k \right],$$

where m is the number of inputs of neuron k in the output layer.

Step 3:   **Weight training**

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(a) Calculate the error gradient for the neurons in the output layer:

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p)$$

where

$$e_k(p) = y_{d,k}(p) - y_k(p)$$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p + 1) = w_{jk}(p) + \Delta w_{jk}(p)$$

(b) Calculate the error gradient for the neurons in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) \times w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p)$$

Update the weights at the hidden neurons:

$$w_{ij}(p + 1) = w_{ij}(p) + \Delta w_{ij}(p)$$

Step 4:   **Iteration**

Increase iteration p by one, go back to Step 2 and repeat the process until the selected error criterion is satisfied.

As an example, we may consider the three-layer back-propagation network shown in Figure 6.10. Suppose that the network is required to perform logical operation Exclusive-OR. Recall that a single-layer perceptron could not do this operation. Now we will apply the three-layer net.

Neurons 1 and 2 in the input layer accept inputs x_1 and x_2, respectively, and redistribute these inputs to the neurons in the hidden layer without any processing:

$x_{13} = x_{14} = x_1$ and $x_{23} = x_{24} = x_2$.
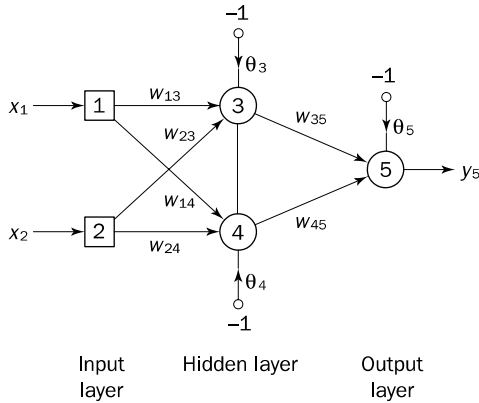
Figure 6.10   Three-layer network for solving the Exclusive-OR operation

The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to -1.

The initial weights and threshold levels are set randomly as follows:

$w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$, $\theta_3 = 0.8$, $\theta_4 = -0.1$ and $\theta_5 = 0.3$.

Consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1\times0.5+1\times0.4-1\times0.8)}] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1\times0.9+1\times1.0+1\times0.1)}] = 0.8808$$

Now the actual output of neuron 5 in the output layer is determined as

$$y_5 = sigmoid\,(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-(-0.5250\times1.2+0.8808\times1.1-1\times0.3)}] = 0.5097$$

Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, e, from the output layer backward to the input layer.

First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5(1 - y_5)e = 0.5097 \times (1 - 0.5097) \times (-0.5097) = -0.1274$$

Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \times y_3 \times \delta_5 = 0.1 \times 0.5250 \times (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \times y_4 \times \delta_5 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \times (-1) \times \delta_5 = 0.1 \times (-1) \times (-0.1274) = 0.0127$$

Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \times \delta_5 \times w_{35} = 0.5250 \times (1 - 0.5250) \times (-0.1274) \times (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \times \delta_5 \times w_{45} = 0.8808 \times (1 - 0.8808) \times (-0.1274) \times 1.1 = -0.0147$$

We then determine the weight corrections:

$$\Delta w_{13} = \alpha \times x_1 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \times x_2 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \times (-1) \times \delta_3 = 0.1 \times (-1) \times 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \times x_1 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \times x_2 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \times (-1) \times \delta_4 = 0.1 \times (-1) \times (-0.0147) = 0.0015$$

At last, we update all weights and threshold levels in our network:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

The training process is repeated until the sum of squared errors is less than 0.001.

Why do we need to sum the squared errors?

The **sum of the squared errors** is a useful indicator of the network's performance. The back-propagation training algorithm attempts to minimise this criterion. When the value of the sum of squared errors in an entire pass through all
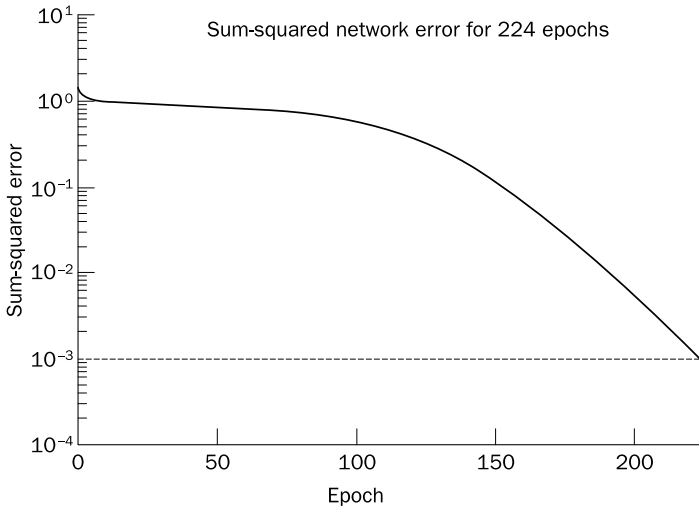
Figure 6.11 Learning curve for operation Exclusive-OR

training sets, or epoch, is **sufficiently small**, a network is considered to have **converged**. In our example, the sufficiently small sum of squared errors is defined as less than 0.001. Figure 6.11 represents a learning curve: the sum of squared errors plotted versus the number of epochs used in training. The learning curve shows how fast a network is learning.

It took 224 epochs or 896 iterations to train our network to perform the Exclusive-OR operation. The following set of final weights and threshold levels satisfied the chosen error criterion:

$w_{13} = 4.7621$, $w_{14} = 6.3917$, $w_{23} = 4.7618$, $w_{24} = 6.3917$, $w_{35} = -10.3788$, $w_{45} = 9.7691$, $\theta_3 = 7.3061$, $\theta_4 = 2.8441$ and $\theta_5 = 4.5589$.

The network has solved the problem! We may now test our network by presenting all training sets and calculating the network's output. The results are shown in Table 6.4.

Table 6.4 Final results of three-layer network learning: the logical operation Exclusive-OR

Inputs		Desired output	Actual output	Error	Sum of squared
x_1	x_2	y_d	y_5	e	errors
1	1	0	0.0155	−0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	−0.0175	

The initial weights and thresholds are set randomly. Does this mean that the same network may find different solutions?

The network obtains different weights and threshold values when it starts from different initial conditions. However, we will always solve the problem, although using a different number of iterations. For instance, when the network was trained again, we obtained the following solution:

$$w_{13} = -6.3041, w_{14} = -5.7896, w_{23} = 6.2288, w_{24} = 6.0088, w_{35} = 9.6657,$$
$$w_{45} = -9.4242, \theta_3 = 3.3858, \theta_4 = -2.8976 \text{ and } \theta_5 = -4.4859.$$

Can we now draw decision boundaries constructed by the multilayer network for operation Exclusive-OR?

It may be rather difficult to draw decision boundaries constructed by neurons with a sigmoid activation function. However, we can represent each neuron in the hidden and output layers by a McCulloch and Pitts model, using a sign function. The network in Figure 6.12 is also trained to perform the Exclusive-OR operation (Touretzky and Pomerlean, 1989; Haykin, 1999).

The positions of the decision boundaries constructed by neurons 3 and 4 in the hidden layer are shown in Figure 6.13(a) and (b), respectively. Neuron 5 in the output layer performs a linear combination of the decision boundaries formed by the two hidden neurons, as shown in Figure 6.13(c). The network in Figure 6.12 does indeed separate black and white dots and thus solves the Exclusive-OR problem.

Is back-propagation learning a good method for machine learning?

Although widely used, back-propagation learning is not immune from problems. For example, the back-propagation learning algorithm does not seem to function in the biological world (Stork, 1989). Biological neurons do not work backward to adjust the strengths of their interconnections, synapses, and thus back-propagation learning cannot be viewed as a process that emulates brain-like learning.
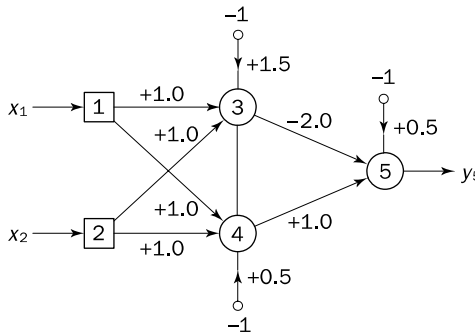
Figure 6.12   Network represented by McCulloch–Pitts model for solving the Exclusive-OR operation.
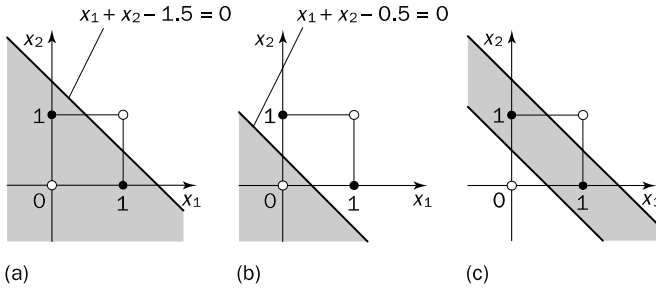
Figure 6.13   (a) Decision boundary constructed by hidden neuron 3 of the network in Figure 6.12; (b) decision boundary constructed by hidden neuron 4; (c) decision boundaries constructed by the complete three-layer network

Another apparent problem is that the calculations are extensive and, as a result, training is slow. In fact, a pure back-propagation algorithm is rarely used in practical applications.

There are several possible ways to improve the computational efficiency of the back-propagation algorithm (Caudill, 1991; Jacobs, 1988; Stubbs, 1990). Some of them are discussed below.

6.5   Accelerated learning in multilayer neural networks

A multilayer network, in general, learns much faster when the sigmoidal activation function is represented by a **hyperbolic tangent**,

$$Y^{tanh} = \frac{2a}{1 + e^{-bX}} - a, \tag{6.16}$$

where a and b are constants.

Suitable values for a and b are: $a = 1.716$ and $b = 0.667$ (Guyon, 1991).

We also can accelerate training by including a **momentum term** in the delta rule of Eq. (6.12) (Rumelhart *et al.*, 1986):

$$\Delta w_{jk}(p) = \beta \times \Delta w_{jk}(p-1) + \alpha \times y_j(p) \times \delta_k(p), \tag{6.17}$$

where β is a positive number $(0 \leqslant \beta < 1)$ called the momentum constant. Typically, the momentum constant is set to 0.95.

Equation (6.17) is called the **generalised delta rule**. In a special case, when $\beta = 0$, we obtain the delta rule of Eq. (6.12).

Why do we need the momentum constant?

According to the observations made in Watrous (1987) and Jacobs (1988), the inclusion of momentum in the back-propagation algorithm has a **stabilising effect** on training. In other words, the inclusion of momentum tends to
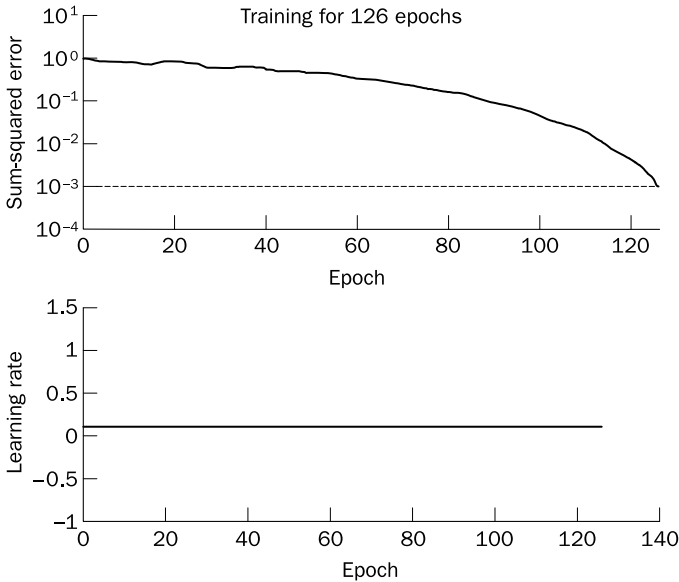
Figure 6.14 Learning with momentum

accelerate descent in the steady downhill direction, and to slow down the process when the learning surface exhibits peaks and valleys.

Figure 6.14 represents learning with momentum for operation Exclusive-OR. A comparison with a pure back-propagation algorithm shows that we reduced the number of epochs from 224 to 126.

In the delta and generalised delta rules, we use a constant and rather small value for the learning rate parameter, a. Can we increase this value to speed up training?

One of the most effective means to accelerate the convergence of back-propagation learning is to adjust the learning rate parameter during training. The small learning rate parameter, α, causes small changes to the weights in the network from one iteration to the next, and thus leads to the smooth learning curve. On the other hand, if the learning rate parameter, α, is made larger to speed up the training process, the resulting larger changes in the weights may cause instability and, as a result, the network may become oscillatory.

To accelerate the convergence and yet avoid the danger of instability, we can apply two heuristics (Jacobs, 1988):

- **Heuristic 1**. If the change of the sum of squared errors has the same algebraic sign for several consequent epochs, then the learning rate parameter, α, should be increased.

- **Heuristic 2**. If the algebraic sign of the change of the sum of squared errors alternates for several consequent epochs, then the learning rate parameter, α, should be decreased.

Adapting the learning rate requires some changes in the back-propagation algorithm. First, the network outputs and errors are calculated from the initial learning rate parameter. If the sum of squared errors at the current epoch exceeds the previous value by more than a predefined ratio (typically 1.04), the learning rate parameter is decreased (typically by multiplying by 0.7) and new weights and thresholds are calculated. However, if the error is less than the previous one, the learning rate is increased (typically by multiplying by 1.05).

Figure 6.15 represents an example of back-propagation training with adaptive learning rate. It demonstrates that adapting the learning rate can indeed decrease the number of iterations.

Learning rate adaptation can be used together with learning with momentum. Figure 6.16 shows the benefits of applying simultaneously both techniques.

The use of momentum and adaptive learning rate significantly improves the performance of a multilayer back-propagation neural network and minimises the chance that the network can become oscillatory.

Neural networks were designed on an analogy with the brain. The brain's memory, however, works by association. For example, we can recognise a familiar face even in an unfamiliar environment within 100–200 ms. We can also recall a complete sensory experience, including sounds and scenes, when we hear only a few bars of music. The brain routinely associates one thing with another.
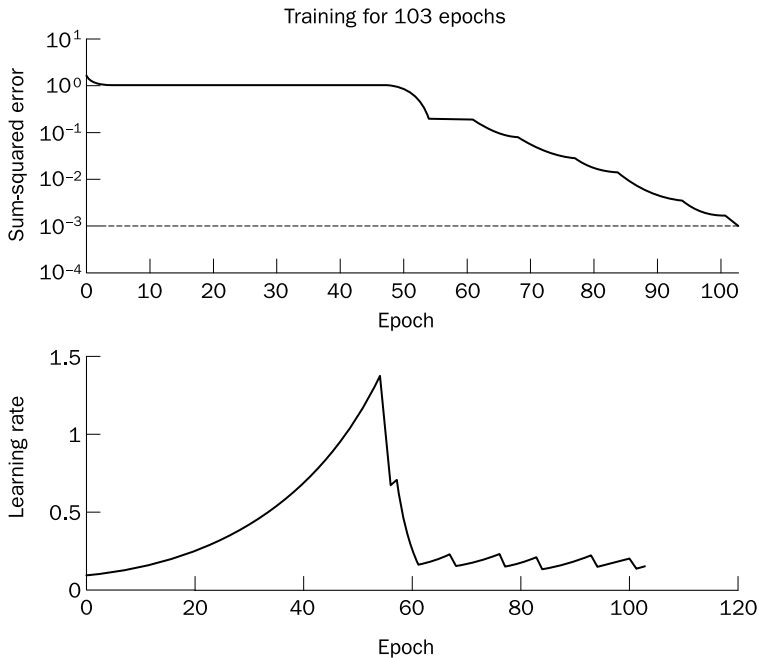
Figure 6.15   Learning with adaptive learning rate
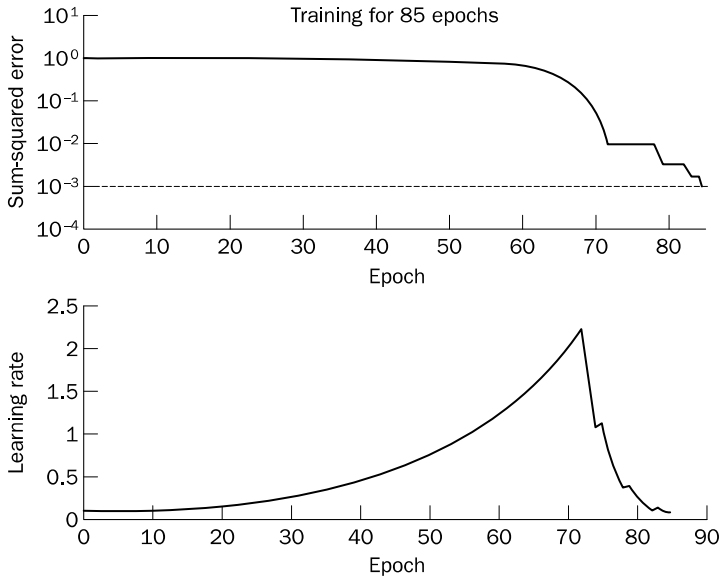
Figure 6.16 Learning with momentum and adaptive learning rate

Can a neural network simulate associative characteristics of the human memory?

Multilayer neural networks trained with the back-propagation algorithm are used for pattern recognition problems. But, as we noted, such networks are not intrinsically intelligent. To emulate the human memory's associative characteristics we need a different type of network: a **recurrent neural network**.

6.6 The Hopfield network

A recurrent neural network has feedback loops from its outputs to its inputs. The presence of such loops has a profound impact on the learning capability of the network.

How does the recurrent network learn?

After applying a new input, the network output is calculated and fed back to adjust the input. Then the output is calculated again, and the process is repeated until the output becomes constant.

Does the output always become constant?

Successive iterations do not always produce smaller and smaller output changes, but on the contrary may lead to chaotic behaviour. In such a case, the network output never becomes constant, and the network is said to be **unstable**.

The stability of recurrent networks intrigued several researchers in the 1960s and 1970s. However, none was able to predict which network would be stable,
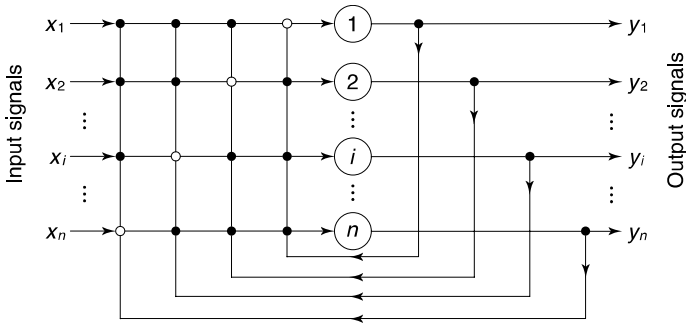
Figure 6.17   Single-layer *n*-neuron Hopfield network

and some researchers were pessimistic about finding a solution at all. The problem was solved only in 1982, when John Hopfield formulated the physical principle of storing information in a dynamically stable network (Hopfield, 1982).

Figure 6.17 shows a single-layer Hopfield network consisting of *n* neurons. The output of each neuron is fed back to the inputs of all other neurons (there is no self-feedback in the Hopfield network).

The Hopfield network usually uses McCulloch and Pitts neurons with the **sign activation function** as its computing element.

How does this function work here?

It works in a similar way to the sign function represented in Figure 6.4. If the neuron's weighted input is less than zero, the output is −1; if the input is greater than zero, the output is +1. However, if the neuron's weighted input is exactly zero, its output remains unchanged – in other words, a neuron remains in its previous state, regardless of whether it is +1 or −1.

$$
Y^{sign} = \begin{cases} +1, & \text{if } X > 0 \\ -1, & \text{if } X < 0 \\ Y, & \text{if } X = 0 \end{cases} \tag{6.18}
$$

The sign activation function may be replaced with a **saturated linear function**, which acts as a pure linear function within the region $[-1, 1]$ and as a sign function outside this region. The saturated linear function is shown in Figure 6.18.

The current state of the network is determined by the current outputs of all neurons, $y_1, y_2, \ldots, y_n$. Thus, for a single-layer *n*-neuron network, the state can be defined by the **state vector** as

$$
\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{6.19}
$$

Saturated linear function

$$Y^{satlin} = \begin{cases} X, & \text{if } -1 < X < 1 \\ +1, & \text{if } X \geq 1 \\ -1, & \text{if } X \leq -1 \end{cases}$$
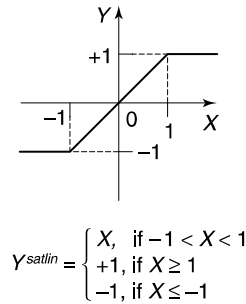
Figure 6.18   The saturated linear activation function

In the Hopfield network, synaptic weights between neurons are usually represented in matrix form as follows:

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{Y}_m \mathbf{Y}_m^T - M\mathbf{I}, \tag{6.20}$$

where M is the number of states to be memorised by the network, $\mathbf{Y}_m$ is the n-dimensional binary vector, $\mathbf{I}$ is $n \times n$ identity matrix, and superscript T denotes a matrix transposition.

An operation of the Hopfield network can be represented geometrically. Figure 6.19 shows a three-neuron network represented as a cube in the three-dimensional space. In general, a network with n neurons has 2^n possible states and is associated with an n-dimensional hypercube. In Figure 6.19, each state is represented by a vertex. When a new input vector is applied, the network moves from one state-vertex to another until it becomes stable.
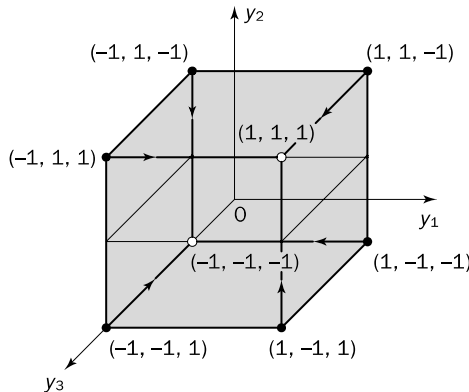
Figure 6.19   Cube representation of the possible states for the three-neuron Hopfield network

What determines a stable state-vertex?

The stable state-vertex is determined by the weight matrix $\mathbf{W}$, the current input vector $\mathbf{X}$, and the threshold matrix θ. If the input vector is partially incorrect or incomplete, the initial state will converge into the stable state-vertex after a few iterations.

Suppose, for instance, that our network is required to memorise two opposite states, $(1, 1, 1)$ and $(-1, -1, -1)$. Thus,

$$\mathbf{Y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{Y}_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix},$$

where $\mathbf{Y}_1$ and $\mathbf{Y}_2$ are the three-dimensional vectors.

We also can represent these vectors in the row, or *transposed*, form

$$\mathbf{Y}_1^T = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \text{ and } \mathbf{Y}_2^T = \begin{bmatrix} -1 & -1 & -1 \end{bmatrix}$$

The 3×3 identity matrix $\mathbf{I}$ is

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, we can now determine the weight matrix as follows:

$$\mathbf{W} = \mathbf{Y}_1 \mathbf{Y}_1^T + \mathbf{Y}_2 \mathbf{Y}_2^T - 2\mathbf{I}$$

or

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

Next, the network is tested by the sequence of input vectors, $\mathbf{X}_1$ and $\mathbf{X}_2$, which are equal to the output (or target) vectors $\mathbf{Y}_1$ and $\mathbf{Y}_2$, respectively. We want to see whether our network is capable of recognising familiar patterns.

How is the Hopfield network tested?

First, we activate it by applying the input vector $\mathbf{X}$. Then, we calculate the actual output vector $\mathbf{Y}$, and finally, we compare the result with the initial input vector $\mathbf{X}$.

$$\mathbf{Y}_m = sign\,(\mathbf{W}\,\mathbf{X}_m - \theta), \qquad m = 1, 2, \dots, M \tag{6.21}$$

where θ is the threshold matrix.

In our example, we may assume all thresholds to be zero. Thus,

$$
Y_1 = sign \left\{ \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}
$$

and

$$
Y_2 = sign \left\{ \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}
$$

As we see, $Y_1 = X_1$ and $Y_2 = X_2$. Thus, both states, $(1, 1, 1)$ and $(-1, -1, -1)$, are said to be *stable*.

How about other states?

With three neurons in the network, there are eight possible states. The remaining six states are all unstable. However, stable states (also called **fundamental memories**) are capable of attracting states that are close to them. As shown in Table 6.5, the fundamental memory $(1, 1, 1)$ attracts unstable states $(-1, 1, 1)$, $(1, -1, 1)$ and $(1, 1, -1)$. Each of these unstable states represents a single error, compared to the fundamental memory $(1, 1, 1)$. On the other hand, the

Table 6.5   Operation of the three-neuron Hopfield network

Possible state	Iteration	Inputs			Outputs			Fundamental memory
		x_1	x_2	x_3	y_1	y_2	y_3	
1 1 1	0	1	1	1	1	1	1	1 1 1
−1 1 1	0	−1	1	1	1	1	1	
	1	1	1	1	1	1	1	1 1 1
1 −1 1	0	1	−1	1	1	1	1	
	1	1	1	1	1	1	1	1 1 1
1 1 −1	0	1	1	−1	1	1	1	
	1	1	1	1	1	1	1	1 1 1
−1 −1 −1	0	−1	−1	−1	−1	−1	−1	−1 −1 −1
−1 −1 1	0	−1	−1	1	−1	−1	−1	
	1	−1	−1	−1	−1	−1	−1	−1 −1 −1
−1 1 −1	0	−1	1	−1	−1	−1	−1	
	1	−1	−1	−1	−1	−1	−1	−1 −1 −1
1 −1 −1	0	1	−1	−1	−1	−1	−1	
	1	−1	−1	−1	−1	−1	−1	−1 −1 −1

fundamental memory $(-1, -1, -1)$ attracts unstable states $(-1, -1, 1)$, $(-1, 1, -1)$ and $(1, -1, -1)$. Here again, each of the unstable states represents a single error, compared to the fundamental memory. Thus, the Hopfield network can indeed act as an **error correction network**. Let us now summarise the Hopfield network training algorithm.

Step 1: *Storage*

The n-neuron Hopfield network is required to store a set of M fundamental memories, $\mathbf{Y}_1, \mathbf{Y}_2, \ldots, \mathbf{Y}_M$. The synaptic weight from neuron i to neuron j is calculated as

$$w_{ij} = \begin{cases} \displaystyle\sum_{m=1}^{M} y_{m,i}\, y_{m,j}, & i \neq j \\ 0, & i = j \end{cases}, \tag{6.22}$$

where $y_{m,i}$ and $y_{m,j}$ are the ith and the jth elements of the fundamental memory $\mathbf{Y}_m$, respectively. In matrix form, the synaptic weights between neurons are represented as

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{Y}_m \mathbf{Y}_m^T - M\mathbf{I}$$

The Hopfield network can store a set of fundamental memories if the weight matrix is symmetrical, with zeros in its main diagonal (Cohen and Grossberg, 1983). That is,

$$\mathbf{W} = \begin{bmatrix} 0 & w_{12} & \cdots & w_{1i} & \cdots & w_{1n} \\ w_{21} & 0 & \cdots & w_{2i} & \cdots & w_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ w_{i1} & w_{i2} & \cdots & 0 & \cdots & w_{in} \\ \vdots & \vdots & & \vdots & & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{ni} & \cdots & 0 \end{bmatrix}, \tag{6.23}$$

where $w_{ij} = w_{ji}$.

Once the weights are calculated, they remain fixed.

Step 2: *Testing*

We need to confirm that the Hopfield network is capable of recalling all fundamental memories. In other words, the network must recall any fundamental memory $\mathbf{Y}_m$ when presented with it as an input. That is,

$$x_{m,i} = y_{m,i}, \qquad i = 1, 2, \ldots, n; \qquad m = 1, 2, \ldots, M$$

$$y_{m,i} = sign\left(\sum_{j=1}^{n} w_{ij}\, x_{m,j} - \theta_i\right),$$

where $y_{m,i}$ is the ith element of the actual output vector $\mathbf{Y}_m$, and $x_{m,j}$ is the jth element of the input vector $\mathbf{X}_m$. In matrix form,

$$\mathbf{X}_m = \mathbf{Y}_m, \qquad m = 1, 2, \dots, M$$
$$\mathbf{Y}_m = sign\,(\mathbf{W}\mathbf{X}_m - \boldsymbol{\theta})$$

If all fundamental memories are recalled perfectly we may proceed to the next step.

Step 3: *Retrieval*

Present an unknown n-dimensional vector (probe), $\mathbf{X}$, to the network and retrieve a stable state. Typically, the probe represents a corrupted or incomplete version of the fundamental memory, that is,

$$\mathbf{X} \neq \mathbf{Y}_m, \qquad m = 1, 2, \dots, M$$

(a) Initialise the retrieval algorithm of the Hopfield network by setting

$$x_j(0) = x_j \qquad j = 1, 2, \dots, n$$

and calculate the initial state for each neuron

$$y_i(0) = sign\left(\sum_{j=1}^{n} w_{ij}\, x_j(0) - \theta_i\right), \qquad i = 1, 2, \dots, n$$

where $x_j(0)$ is the jth element of the probe vector $\mathbf{X}$ at iteration $p = 0$, and $y_i(0)$ is the state of neuron i at iteration $p = 0$.
   In matrix form, the state vector at iteration $p = 0$ is presented as

$$\mathbf{Y}(0) = sign\,[\mathbf{W}\mathbf{X}(0) - \boldsymbol{\theta}]$$

(b) Update the elements of the state vector, $\mathbf{Y}(p)$, according to the following rule:

$$y_i(p+1) = sign\left(\sum_{j=1}^{n} w_{ij}\, x_j(p) - \theta_i\right)$$

Neurons for updating are selected **asynchronously**, that is, randomly and one at a time.
   Repeat the iteration until the state vector becomes unchanged, or in other words, a stable state is achieved. The condition for stability can be defined as:

$$y_i(p+1) = sign\left(\sum_{j=1}^{n} w_{ij}\, y_j(p) - \theta_i\right), \qquad i = 1, 2, \dots, n \tag{6.24}$$

or, in matrix form,

$$\mathbf{Y}(p+1) = sign\,[\mathbf{W}\mathbf{Y}(p) - \boldsymbol{\theta}] \tag{6.25}$$

The Hopfield network will always converge to a stable state if the retrieval is done asynchronously (Haykin, 1999). However, this stable state does not necessarily represent one of the fundamental memories, and if it is a fundamental memory it is not necessarily the closest one.

Suppose, for example, we wish to store three fundamental memories in the five-neuron Hopfield network:

$$\mathbf{X}_1 = (+1, +1, +1, +1, +1)$$
$$\mathbf{X}_2 = (+1, -1, +1, -1, +1)$$
$$\mathbf{X}_3 = (-1, +1, -1, +1, -1)$$

The weight matrix is constructed from Eq. (6.20),

$$\mathbf{W} = \begin{bmatrix} 0 & -1 & 3 & -1 & 3 \\ -1 & 0 & -1 & 3 & -1 \\ 3 & -1 & 0 & -1 & 3 \\ -1 & 3 & -1 & 0 & -1 \\ 3 & -1 & 3 & -1 & 0 \end{bmatrix}$$

Assume now that the probe vector is represented by

$$\mathbf{X} = (+1, +1, -1, +1, +1)$$

If we compare this probe with the fundamental memory $\mathbf{X}_1$, we find that these two vectors differ only in a single bit. Thus, we may expect that the probe $\mathbf{X}$ will converge to the fundamental memory $\mathbf{X}_1$. However, when we apply the Hopfield network training algorithm described above, we obtain a different result. The pattern produced by the network recalls the memory $\mathbf{X}_3$, a false memory.

This example reveals one of the problems inherent to the Hopfield network.

Another problem is the **storage capacity**, or the largest number of fundamental memories that can be stored and retrieved correctly. Hopfield showed experimentally (Hopfield, 1982) that the maximum number of fundamental memories M_{max} that can be stored in the n-neuron recurrent network is limited by

$$M_{max} = 0.15n \tag{6.26}$$

We also may define the storage capacity of a Hopfield network on the basis that **most** of the fundamental memories are to be retrieved perfectly (Amit, 1989):

$$M_{max} = \frac{n}{2 \ln n} \tag{6.27}$$

What if we want all the fundamental memories to be retrieved perfectly?
It can be shown that to retrieve **all** the fundamental memories perfectly, their
number must be halved (Amit, 1989):

$$M_{max} = \frac{n}{4 \ln n} \qquad\qquad (6.28)$$

As we can see now, the storage capacity of a Hopfield network has to be kept
rather small for the fundamental memories to be retrievable. This is a major
limitation of the Hopfield network.

Strictly speaking, a Hopfield network represents an **auto-associative** type of
memory. In other words, a Hopfield network can retrieve a corrupted or
incomplete memory but cannot associate it with another different memory.

In contrast, human memory is essentially associative. One thing may remind
us of another, and that of another, and so on. We use a chain of mental
associations to recover a lost memory. If we, for example, forget where we left an
umbrella, we try to recall where we last had it, what we were doing, and who we
were talking to. Thus, we attempt to establish a chain of associations, and
thereby to restore a lost memory.

Why can't a Hopfield network do this job?
The Hopfield network is a single-layer network, and thus the output pattern
appears on the same set of neurons to which the input pattern was applied. To
associate one memory with another, we need a recurrent neural network capable
of accepting an input pattern on one set of neurons and producing a related, but
different, output pattern on another set of neurons. In fact, we need a two-layer
recurrent network, the **bidirectional associative memory**.

6.7 Bidirectional associative memory

Bidirectional associative memory (BAM), first proposed by Bart Kosko, is a
heteroassociative network (Kosko, 1987, 1988). It associates patterns from one
set, set A, to patterns from another set, set B, and vice versa. Like a Hopfield
network, the BAM can generalise and also produce correct outputs despite
corrupted or incomplete inputs. The basic BAM architecture is shown in Figure
6.20. It consists of two fully connected layers: an input layer and an output layer.

How does the BAM work?
The input vector $\mathbf{X}(p)$ is applied to the transpose of weight matrix $\mathbf{W}^T$ to
produce an output vector $\mathbf{Y}(p)$, as illustrated in Figure 6.20(a). Then, the output
vector $\mathbf{Y}(p)$ is applied to the weight matrix $\mathbf{W}$ to produce a new input vector
$\mathbf{X}(p+1)$, as in Figure 6.20(b). This process is repeated until input and output
vectors become unchanged, or in other words, the BAM reaches a stable state.

The basic idea behind the BAM is to store pattern pairs so that when
n-dimensional vector $\mathbf{X}$ from set A is presented as input, the BAM recalls
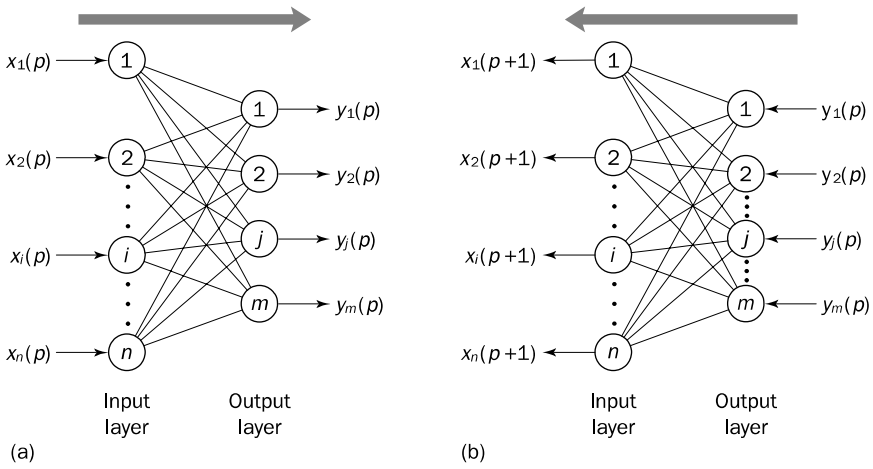
Figure 6.20   BAM operation: (a) forward direction; (b) backward direction

m-dimensional vector **Y** from set B, but when **Y** is presented as input, the BAM recalls **X**.

To develop the BAM, we need to create a correlation matrix for each pattern pair we want to store. The correlation matrix is the matrix product of the input vector **X**, and the transpose of the output vector $\mathbf{Y}^T$. The BAM weight matrix is the sum of all correlation matrices, that is,

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{X}_m \mathbf{Y}_m^T, \tag{6.29}$$

where M is the number of pattern pairs to be stored in the BAM.

Like a Hopfield network, the BAM usually uses McCulloch and Pitts neurons with the sign activation function.

The BAM training algorithm can be presented as follows.

Step 1: *Storage*

The BAM is required to store M pairs of patterns. For example, we may wish to store four pairs:

$$\text{Set } A: \; \mathbf{X}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \; \mathbf{X}_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \; \mathbf{X}_3 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \; \mathbf{X}_4 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

$$\text{Set } B: \; \mathbf{Y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \; \mathbf{Y}_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} \; \mathbf{Y}_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \; \mathbf{Y}_4 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$$

In this case, the BAM input layer must have six neurons and the output layer three neurons.

The weight matrix is determined as

$$\mathbf{W} = \sum_{m=1}^{4} \mathbf{X}_m \mathbf{Y}_m^T$$

or

$$\mathbf{W} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 \end{bmatrix}$$

$$+ \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \begin{bmatrix} -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 4 \\ 4 & 0 & 4 \\ 0 & 4 & 0 \\ 0 & 4 & 0 \\ 4 & 0 & 4 \\ 4 & 0 & 4 \end{bmatrix}$$

Step 2: *Testing*
The BAM should be able to receive any vector from set A and retrieve the associated vector from set B, and receive any vector from set B and retrieve the associated vector from set A. Thus, first we need to confirm that the BAM is able to recall $\mathbf{Y}_m$ when presented with $\mathbf{X}_m$. That is,

$$\mathbf{Y}_m = sign\,(\mathbf{W}^T \mathbf{X}_m), \qquad m = 1, 2, \ldots, M \tag{6.30}$$

For instance,

$$\mathbf{Y}_1 = sign\,(\mathbf{W}^T \mathbf{X}_1) = sign \left\{ \begin{bmatrix} 4 & 4 & 0 & 0 & 4 & 4 \\ 0 & 0 & 4 & 4 & 0 & 0 \\ 4 & 4 & 0 & 0 & 4 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right\} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Then, we confirm that the BAM recalls $\mathbf{X}_m$ when presented with $\mathbf{Y}_m$. That is,

$$\mathbf{X}_m = sign\,(\mathbf{W}\,\mathbf{Y}_m), \qquad m = 1, 2, \ldots, M \tag{6.31}$$

For instance,

$$X_3 = sign\,(W\,Y_3) = sign\left\{\begin{bmatrix} 4 & 0 & 4 \\ 4 & 0 & 4 \\ 0 & 4 & 0 \\ 0 & 4 & 0 \\ 4 & 0 & 4 \\ 4 & 0 & 4 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

In our example, all four pairs are recalled perfectly, and we can proceed to the next step.

Step 3: *Retrieval*

Present an unknown vector (probe) X to the BAM and retrieve a stored association. The probe may present a corrupted or incomplete version of a pattern from set A (or from set B) stored in the BAM. That is,

$$X \neq X_m, \qquad m = 1, 2, \ldots, M$$

(a) Initialise the BAM retrieval algorithm by setting

$$X(0) = X, \qquad p = 0$$

and calculate the BAM output at iteration p

$$Y(p) = sign\,[W^T\,X(p)]$$

(b) Update the input vector $X(p)$:

$$X(p + 1) = sign\,[W\,Y(p)]$$

and repeat the iteration until equilibrium, when input and output vectors remain unchanged with further iterations. The input and output patterns will then represent an associated pair.

The BAM is unconditionally stable (Kosko, 1992). This means that any set of associations can be learned without risk of instability. This important quality arises from the BAM using the transpose relationship between weight matrices in forward and backward directions.

Let us now return to our example. Suppose we use vector X as a probe. It represents a single error compared with the pattern X_1 from set A:

$$X = (-1, +1, +1, +1, +1, +1)$$

This probe applied as the BAM input produces the output vector Y_1 from set B. The vector Y_1 is then used as input to retrieve the vector X_1 from set A. Thus, the BAM is indeed capable of error correction.

There is also a close relationship between the BAM and the Hopfield network. If the BAM weight matrix is square and symmetrical, then $\mathbf{W} = \mathbf{W}^T$. In this case, input and output layers are of the same size, and the BAM can be reduced to the autoassociative Hopfield network. Thus, the Hopfield network can be considered as a BAM special case.

The constraints imposed on the storage capacity of the Hopfield network can also be extended to the BAM. In general, the maximum number of associations to be stored in the BAM should not exceed the number of neurons in the smaller layer. Another, even more serious problem, is incorrect convergence. The BAM may not always produce the closest association. In fact, a stable association may be only slightly related to the initial input vector.

The BAM still remains the subject of intensive research. However, despite all its current problems and limitations, the BAM promises to become one of the most useful artificial neural networks.

Can a neural network learn without a 'teacher'?

The main property of a neural network is an ability to learn from its environment, and to improve its performance through learning. So far we have considered **supervised** or **active learning** – learning with an external 'teacher' or a supervisor who presents a training set to the network. But another type of learning also exists: **unsupervised learning**.

In contrast to supervised learning, unsupervised or **self-organised learning** does not require an external teacher. During the training session, the neural network receives a number of different input patterns, discovers significant features in these patterns and learns how to classify input data into appropriate categories. Unsupervised learning tends to follow the neuro-biological organisation of the brain.

Unsupervised learning algorithms aim to learn rapidly. In fact, self-organising neural networks learn much faster than back-propagation networks, and thus can be used in real time.

6.8   Self-organising neural networks

Self-organising neural networks are effective in dealing with unexpected and changing conditions. In this section, we consider Hebbian and competitive learning, which are based on self-organising networks.

6.8.1   Hebbian learning

In 1949, neuropsychologist Donald Hebb proposed one of the key ideas in biological learning, commonly known as Hebb's Law (Hebb, 1949). Hebb's Law states that if neuron i is near enough to excite neuron j and repeatedly participates in its activation, the synaptic connection between these two neurons is strengthened and neuron j becomes more sensitive to stimuli from neuron i.
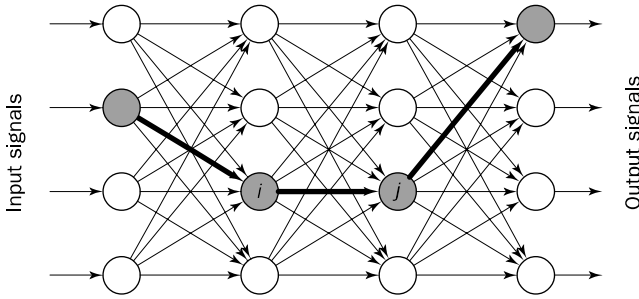
Figure 6.21 Hebbian learning in a neural network

We can represent Hebb's Law in the form of two rules as follows (Stent, 1973):

1. If two neurons on either side of a connection are activated synchronously, then the weight of that connection is increased.

2. If two neurons on either side of a connection are activated asynchronously, then the weight of that connection is decreased.

Hebb's Law provides the basis for learning without a teacher. Learning here is a local phenomenon occurring without feedback from the environment. Figure 6.21 shows Hebbian learning in a neural network.

Using Hebb's Law we can express the adjustment applied to the weight w_{ij} at iteration p in the following form:

$$\Delta w_{ij}(p) = F[y_j(p), x_i(p)], \tag{6.32}$$

where $F[y_j(p), x_i(p)]$ is a function of both postsynaptic and presynaptic activities.

As a special case, we can represent Hebb's Law as follows (Haykin, 1999):

$$\Delta w_{ij}(p) = \alpha y_j(p) x_i(p), \tag{6.33}$$

where α is the **learning rate** parameter.

This equation is referred to as the **activity product rule**. It shows how a change in the weight of the synaptic connection between a pair of neurons is related to a product of the incoming and outgoing signals.

Hebbian learning implies that weights can only increase. In other words, Hebb's Law allows the strength of a connection to increase, but it does not provide a means to decrease the strength. Thus, repeated application of the input signal may drive the weight w_{ij} into saturation. To resolve this problem, we might impose a limit on the growth of synaptic weights. It can be done by introducing a non-linear **forgetting factor** into Hebb's Law in Eq. (6.33) as follows (Kohonen, 1989):

$$\Delta w_{ij}(p) = \alpha y_j(p) x_i(p) - \phi y_j(p) w_{ij}(p) \tag{6.34}$$

where ϕ is the forgetting factor.

What does a forgetting factor mean?

Forgetting factor ϕ specifies the weight decay in a single learning cycle. It usually falls in the interval between 0 and 1. If the forgetting factor is 0, the neural network is capable only of strengthening its synaptic weights, and as a result, these weights grow towards infinity. On the other hand, if the forgetting factor is close to 1, the network remembers very little of what it learns. Therefore, a rather small forgetting factor should be chosen, typically between 0.01 and 0.1, to allow only a little 'forgetting' while limiting the weight growth.

Equation (6.34) may also be written in the form referred to as a **generalised activity product rule**

$$\Delta w_{ij}(p) = \phi\, y_j(p)[\lambda\, x_i(p) - w_{ij}(p)], \tag{6.35}$$

where $\lambda = \alpha/\phi$.

The generalised activity product rule implies that, if the presynaptic activity (input of neuron i) at iteration p, $x_i(p)$, is less than $w_{ij}(p)/\lambda$, then the modified synaptic weight at iteration $(p+1)$, $w_{ij}(p+1)$, will decrease by an amount proportional to the postsynaptic activity (output of neuron j) at iteration p, $y_j(p)$. On the other hand, if $x_i(p)$ is greater than $w_{ij}(p)/\lambda$, then the modified synaptic weight at iteration $(p+1)$, $w_{ij}(p+1)$, will increase also in proportion to the output of neuron j, $y_j(p)$. In other words, we can determine the **activity balance point** for modifying the synaptic weight as a variable equal to $w_{ij}(p)/\lambda$. This approach solves the problem of an infinite increase of the synaptic weights.

Let us now derive the generalised Hebbian learning algorithm.

Step 1:   *Initialisation*

Set initial synaptic weights and thresholds to small random values, say in an interval $[0, 1]$. Also assign small positive values to the learning rate parameter α and forgetting factor ϕ.

Step 2:   *Activation*

Compute the neuron output at iteration p

$$y_j(p) = \sum_{i=1}^{n} x_i(p)\, w_{ij}(p) - \theta_j,$$

where n is the number of neuron inputs, and θ_j is the threshold value of neuron j.

Step 3:   *Learning*

Update the weights in the network:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p),$$

where $\Delta w_{ij}(p)$ is the weight correction at iteration p.

The weight correction is determined by the generalised activity product rule:

$$\Delta w_{ij}(p) = \phi\, y_j(p)[\lambda\, x_i(p) - w_{ij}(p)]$$

Step 4: *Iteration*

Increase iteration p by one, go back to Step 2 and continue until the synaptic weights reach their steady-state values.

To illustrate Hebbian learning, consider a fully connected feedforward network with a single layer of five computation neurons, as shown in Figure 6.22(a). Each neuron is represented by a McCulloch and Pitts model with the sign activation function. The network is trained with the generalised activity product rule on the following set of input vectors:

$$\mathbf{X}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{X}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{X}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{X}_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{X}_5 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

(a)

(b)

Figure 6.22 Unsupervised Hebbian learning in a single-layer network: (a) initial and final states of the network; (b) initial and final weight matrices

Here, the input vector X_1 is the null vector. As you may also notice, input signals x_4 (in the vector X_3) and x_3 (in the vector X_4) are the only unity components in the corresponding vectors, while unity signals x_2 and x_5 always come together, as seen in the vectors X_2 and X_5.

In our example, the initial weight matrix is represented by the 5×5 identity matrix I. Thus, in the initial state, each of the neurons in the input layer is connected to the neuron in the same position in the output layer with a synaptic weight of 1, and to the other neurons with weights of 0. The thresholds are set to random numbers in the interval between 0 and 1. The learning rate parameter α and forgetting factor ϕ are taken as 0.1 and 0.02, respectively.

After training, as can be seen from Figure 6.22(b), the weight matrix becomes different from the initial identity matrix I. The weights between neuron 2 in the input layer and neuron 5 in the output layer, and neuron 5 in the input layer and neuron 2 in the output layer have increased from 0 to 2.0204. Our network has learned new associations. At the same time, the weight between neuron 1 in the input layer and neuron 1 in the output layer has become 0. The network has forgotten this association.

Let us now test our network. A test input vector, or probe, is defined as

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

When this probe is presented to the network, we obtain

$$Y = sign\,(\mathbf{W}X - \boldsymbol{\theta})$$

$$Y = sign \left\{ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2.0204 & 0 & 0 & 2.0204 \\ 0 & 0 & 1.0200 & 0 & 0 \\ 0 & 0 & 0 & 0.9996 & 0 \\ 0 & 2.0204 & 0 & 0 & 2.0204 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.4940 \\ 0.2661 \\ 0.0907 \\ 0.9478 \\ 0.0737 \end{bmatrix} \right\} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Sure enough, the network has associated input x_5 with outputs y_2 and y_5 because inputs x_2 and x_5 were coupled during training. But the network cannot associate input x_1 with output y_1 any more because unity input x_1 did not appear during training and our network has lost the ability to recognise it.

Thus, a neural network really can learn to associate stimuli commonly presented together, and most important, the network can learn without a 'teacher'.

6.8.2   Competitive learning

Another popular type of unsupervised learning is **competitive learning**. In competitive learning, neurons compete among themselves to be activated. While in Hebbian learning, several output neurons can be activated simultaneously, in competitive learning only a single output neuron is active at any time. The output neuron that wins the 'competition' is called the **winner-takes-all** neuron.

The basic idea of competitive learning was introduced in the early 1970s (Grossberg, 1972; von der Malsburg, 1973; Fukushima, 1975). However, competitive learning did not attract much interest until the late 1980s, when Teuvo Kohonen introduced a special class of artificial neural networks called **self-organising feature maps** (Kohonen, 1989). These maps are based on competitive learning.

What is a self-organising feature map?

Our brain is dominated by the cerebral cortex, a very complex structure of billions of neurons and hundreds of billions of synapses. The cortex is neither uniform nor homogeneous. It includes areas, identified by the thickness of their layers and the types of neurons within them, that are responsible for different human activities (motor, visual, auditory, somatosensory, etc.), and thus associated with different sensory inputs. We can say that each sensory input is mapped into a corresponding area of the cerebral cortex; in other words, the cortex is a self-organising computational map in the human brain.

Can we model the self-organising map?

Kohonen formulated the **principle of topographic map formation** (Kohonen, 1990). This principle states that the spatial location of an output neuron in the topographic map corresponds to a particular feature of the input pattern. Kohonen also proposed the feature-mapping model shown in Figure 6.23 (Kohonen, 1982). This model captures the main features of self-organising maps in the brain and yet can be easily represented in a computer.
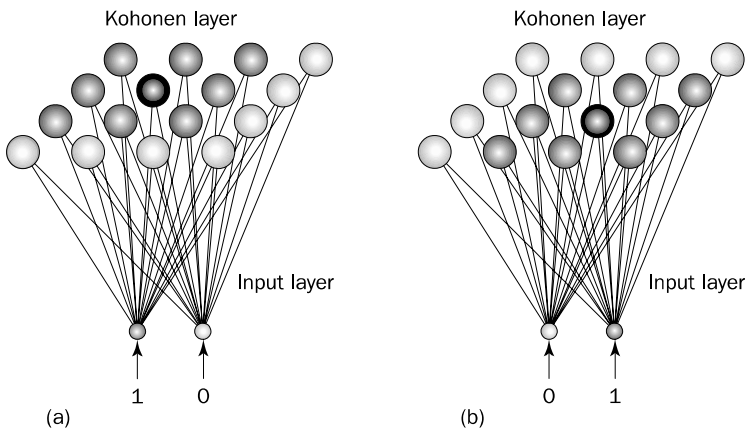
Figure 6.23   Feature-mapping Kohonen model

The Kohonen model provides a topological mapping, placing a fixed number of input patterns from the input layer into a higher-dimensional output or Kohonen layer. In Figure 6.23, the Kohonen layer consists of a two-dimensional lattice made up of 4-by-4 neurons, with each neuron having two inputs. The winning neuron is shown in black and its neighbours in grey. Here, the winner's neighbours are neurons in close physical proximity to the winner.

How close is 'close physical proximity'?

How close physical proximity is, is determined by the network designer. The winner's neighbourhood may include neurons within one, two or even three positions on either side. For example, Figure 6.23 depicts the winner's neighbourhood of size one. Generally, training in the Kohonen network begins with the winner's neighbourhood of a fairly large size. Then, as training proceeds, the neighbourhood size gradually decreases.

The Kohonen network consists of a single layer of computation neurons, but it has two different types of connections. There are **forward connections** from the neurons in the input layer to the neurons in the output layer, and also **lateral connections** between neurons in the output layer, as shown in Figure 6.24. The lateral connections are used to create a competition between neurons. The neuron with the largest activation level among all neurons in the output layer becomes the winner (the winner-takes-all neuron). This neuron is the only neuron that produces an output signal. The activity of all other neurons is suppressed in the competition.

When an input pattern is presented to the network, each neuron in the Kohonen layer receives a full copy of the input pattern, modified by its path through the weights of the synaptic connections between the input layer and the Kohonen layer. The lateral feedback connections produce excitatory or inhibitory effects, depending on the distance from the winning neuron. This is achieved by the use of a **Mexican hat function** which describes synaptic weights between neurons in the Kohonen layer.

What is the Mexican hat function?

The Mexican hat function shown in Figure 6.25 represents the relationship between the distance from the winner-takes-all neuron and the strength of the
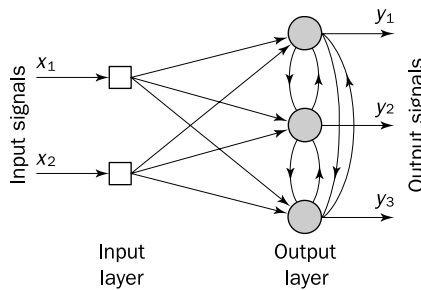
Figure 6.24   Architecture of the Kohonen network
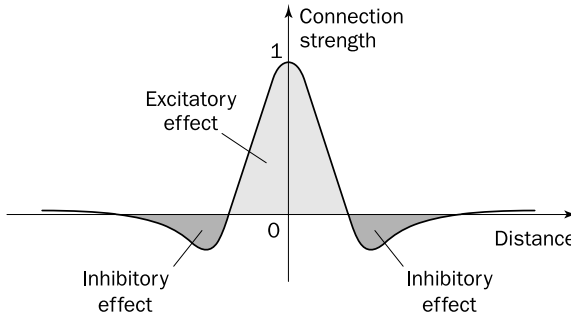
Figure 6.25   The Mexican hat function of lateral connection

connections within the Kohonen layer. According to this function, the near neighbourhood (a short-range lateral excitation area) has a strong excitatory effect, remote neighbourhood (an inhibitory penumbra) has a mild inhibitory effect and very remote neighbourhood (an area surrounding the inhibitory penumbra) has a weak excitatory effect, which is usually neglected.

In the Kohonen network, a neuron learns by shifting its weights from inactive connections to active ones. Only the winning neuron and its neighbourhood are allowed to learn. If a neuron does not respond to a given input pattern, then learning cannot occur in that particular neuron.

The output signal, y_j, of the winner-takes-all neuron j is set equal to one and the output signals of all the other neurons (the neurons that lose the competition) are set to zero.

The **standard competitive learning rule** (Haykin, 1999) defines the change Δw_{ij} applied to synaptic weight w_{ij} as

$$\Delta w_{ij} = \begin{cases} \alpha(x_i - w_{ij}), & \text{if neuron } j \text{ wins the competition} \\ 0, & \text{if neuron } j \text{ loses the competition} \end{cases} \qquad (6.36)$$

where x_i is the input signal and α is the **learning rate parameter**. The learning rate parameter lies in the range between 0 and 1.

The overall effect of the competitive learning rule resides in moving the synaptic weight vector $\mathbf{W}_j$ of the winning neuron j towards the input pattern $\mathbf{X}$. The matching criterion is equivalent to the minimum **Euclidean distance** between vectors.

What is the Euclidean distance?

The Euclidean distance between a pair of n-by-1 vectors $\mathbf{X}$ and $\mathbf{W}_j$ is defined by

$$d = \|\mathbf{X} - \mathbf{W}_j\| = \left[\sum_{i=1}^{n} (x_i - w_{ij})^2 \right]^{1/2}, \qquad (6.37)$$

where x_i and w_{ij} are the ith elements of the vectors $\mathbf{X}$ and $\mathbf{W}_j$, respectively.
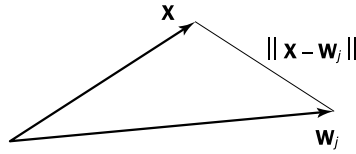
Figure 6.26 Euclidean distance as a measure of similarity between vectors **X** and **W**$_j$

The similarity between the vectors X and W_j is determined as the reciprocal of the Euclidean distance d. In Figure 6.26, the Euclidean distance between the vectors X and W_j is presented as the length of the line joining the tips of those vectors. Figure 6.26 clearly demonstrates that the smaller the Euclidean distance is, the greater will be the similarity between the vectors X and W_j.

To identify the winning neuron, j_X, that best matches the input vector X, we may apply the following condition (Haykin, 1999):

$$j_X = \min_j \|X - W_j\|, \qquad j = 1, 2, \ldots, m \qquad (6.38)$$

where m is the number of neurons in the Kohonen layer.

Suppose, for instance, that the two-dimensional input vector X is presented to the three-neuron Kohonen network,

$$X = \begin{bmatrix} 0.52 \\ 0.12 \end{bmatrix}$$

The initial weight vectors, W_j, are given by

$$W_1 = \begin{bmatrix} 0.27 \\ 0.81 \end{bmatrix} \quad W_2 = \begin{bmatrix} 0.42 \\ 0.70 \end{bmatrix} \quad W_3 = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix}$$

We find the winning (best-matching) neuron j_X using the minimum-distance Euclidean criterion:

$$d_1 = \sqrt{(x_1 - w_{11})^2 + (x_2 - w_{21})^2} = \sqrt{(0.52 - 0.27)^2 + (0.12 - 0.81)^2} = 0.73$$

$$d_2 = \sqrt{(x_1 - w_{12})^2 + (x_2 - w_{22})^2} = \sqrt{(0.52 - 0.42)^2 + (0.12 - 0.70)^2} = 0.59$$

$$d_3 = \sqrt{(x_1 - w_{13})^2 + (x_2 - w_{23})^2} = \sqrt{(0.52 - 0.43)^2 + (0.12 - 0.21)^2} = 0.13$$

Thus, neuron 3 is the winner and its weight vector W_3 is to be updated according to the competitive learning rule described in Eq. (6.36). Assuming that the learning rate parameter α is equal to 0.1, we obtain

$$\Delta w_{13} = \alpha(x_1 - w_{13}) = 0.1(0.52 - 0.43) = 0.01$$
$$\Delta w_{23} = \alpha(x_2 - w_{23}) = 0.1(0.12 - 0.21) = -0.01$$

The updated weight vector $\mathbf{W}_3$ at iteration $(p+1)$ is determined as:

$$\mathbf{W}_3(p+1) = \mathbf{W}_3(p) + \Delta\mathbf{W}_3(p) = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix} + \begin{bmatrix} 0.01 \\ -0.01 \end{bmatrix} = \begin{bmatrix} 0.44 \\ 0.20 \end{bmatrix}$$

The weight vector $\mathbf{W}_3$ of the winning neuron 3 becomes closer to the input vector X with each iteration.

Let us now summarise the competitive learning algorithm as follows (Kohonen, 1989):

Step 1: *Initialisation*

Set initial synaptic weights to small random values, say in an interval $[0, 1]$, and assign a small positive value to the learning rate parameter α.

Step 2: *Activation and similarity matching*

Activate the Kohonen network by applying the input vector X, and find the winner-takes-all (best matching) neuron $j_\mathbf{X}$ at iteration p, using the minimum-distance Euclidean criterion

$$j_\mathbf{X}(p) = \min_j \|\mathbf{X} - \mathbf{W}_j(p)\| = \left\{ \sum_{i=1}^n [x_i - w_{ij}(p)]^2 \right\}^{1/2}, \quad j = 1, 2, \ldots, m$$

where n is the number of neurons in the input layer, and m is the number of neurons in the output or Kohonen layer.

Step 3: *Learning*

Update the synaptic weights

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p),$$

where $\Delta w_{ij}(p)$ is the weight correction at iteration p.

The weight correction is determined by the competitive learning rule

$$\Delta w_{ij}(p) = \begin{cases} \alpha[x_i - w_{ij}(p)], & j \in \Lambda_j(p) \\ 0, & j \notin \Lambda_j(p) \end{cases}, \tag{6.39}$$

where α is the learning rate parameter, and $\Lambda_j(p)$ is the neighbourhood function centred around the winner-takes-all neuron $j_\mathbf{X}$ at iteration p.

The neighbourhood function Λ_j usually has a constant amplitude. It implies that all the neurons located inside the topological neighbourhood are activated simultaneously, and the relationship among those neurons is independent of their distance from the winner-takes-all neuron $j_\mathbf{X}$. This simple form of a neighbourhood function is shown in Figure 6.27.
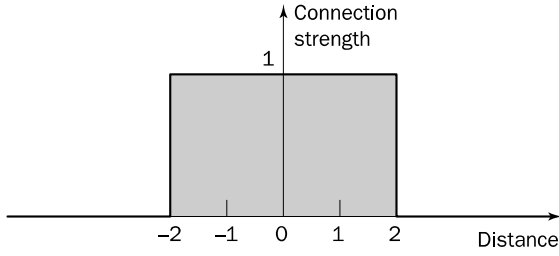
Figure 6.27   Rectangular neighbourhood function
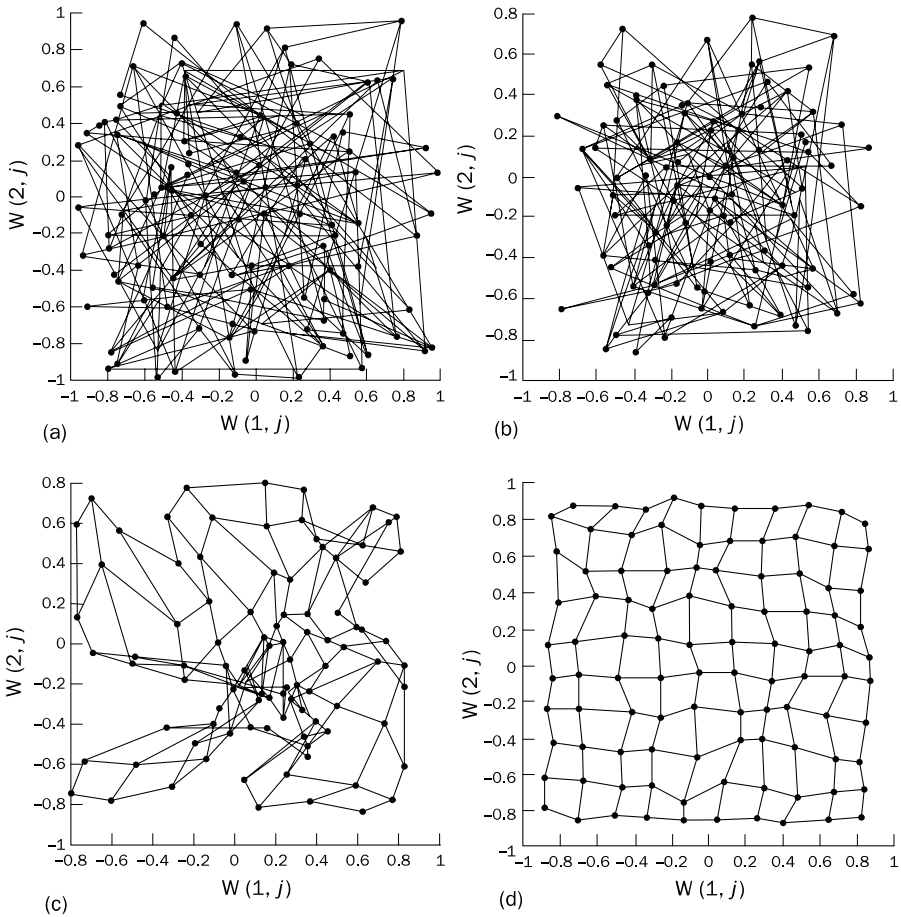
Figure 6.28   Competitive learning in the Kohonen network: (a) initial random weights; (b) network after 100 iterations; (c) network after 1000 iterations; (d) network after 10,000 iterations

The rectangular neighbourhood function Λ_j takes on a binary character. Thus, identifying the neuron outputs, we may write

$$y_j = \begin{cases} 1, & j \in \Lambda_j(p) \\ 0, & j \notin \Lambda_j(p) \end{cases} \tag{6.40}$$

Step 4: *Iteration*

Increase iteration p by one, go back to Step 2 and continue until the minimum-distance Euclidean criterion is satisfied, or no noticeable changes occur in the feature map.

To illustrate competitive learning, consider the Kohonen network with 100 neurons arranged in the form of a two-dimensional lattice with 10 rows and 10 columns. The network is required to classify two-dimensional input vectors. In other words, each neuron in the network should respond only to the input vectors occurring in its region.

The network is trained with 1000 two-dimensional input vectors generated randomly in a square region in the interval between -1 and $+1$. Initial synaptic weights are also set to random values in the interval between -1 and $+1$, and the learning rate parameter α is equal to 0.1.

Figure 6.28 demonstrates different stages in the process of network learning. Each neuron is represented by a black dot at the location of its two weights, w_{1j} and w_{2j}. Figure 6.28(a) shows the initial synaptic weights randomly distributed in the square region. Figures 6.28(b), (c) and (d) present the weight vectors in the input space after 100, 1000 and 10,000 iterations, respectively.

The results shown in Figure 6.28 demonstrate the self-organisation of the Kohonen network that characterises unsupervised learning. At the end of the learning process, the neurons are mapped in the correct order and the map itself spreads out to fill the input space. Each neuron now is able to identify input vectors in its own input space.

To see how neurons respond, let us test our network by applying the following input vectors:

$$\mathbf{X}_1 = \begin{bmatrix} 0.2 \\ 0.9 \end{bmatrix} \quad \mathbf{X}_2 = \begin{bmatrix} 0.6 \\ -0.2 \end{bmatrix} \quad \mathbf{X}_3 = \begin{bmatrix} -0.7 \\ -0.8 \end{bmatrix}$$

As illustrated in Figure 6.29, neuron 6 responds to the input vector $\mathbf{X}_1$, neuron 69 responds to the input vector $\mathbf{X}_2$ and neuron 92 to the input vector $\mathbf{X}_3$. Thus, the feature map displayed in the input space in Figure 6.29 is topologically ordered and the spatial location of a neuron in the lattice corresponds to a particular feature of input patterns.
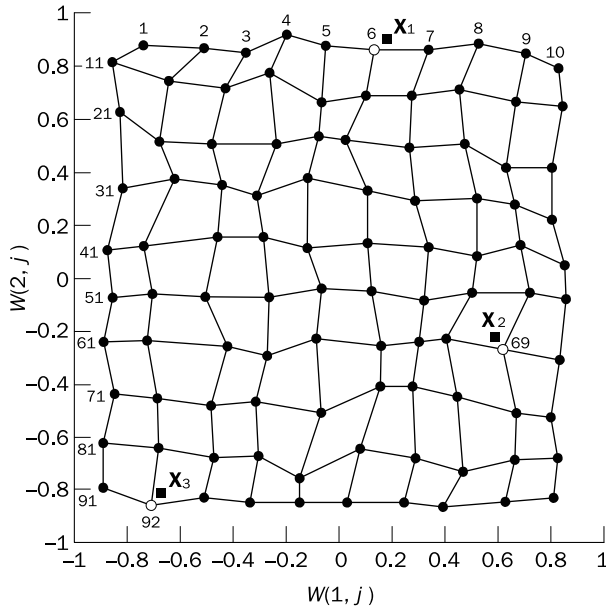
Figure 6.29 Topologically ordered feature map displayed in the input space

6.9 Summary

In this chapter, we introduced artificial neural networks and discussed the basic ideas behind machine learning. We presented the concept of a perceptron as a simple computing element and considered the perceptron learning rule. We explored multilayer neural networks and discussed how to improve the computational efficiency of the back-propagation learning algorithm. Then we introduced recurrent neural networks, considered the Hopfield network training algorithm and bidirectional associative memory (BAM). Finally, we presented self-organising neural networks and explored Hebbian and competitive learning.

The most important lessons learned in this chapter are:

- Machine learning involves adaptive mechanisms that enable computers to learn from experience, learn by example and learn by analogy. Learning capabilities can improve the performance of an intelligent system over time. One of the most popular approaches to machine learning is artificial neural networks.

- An artificial neural network consists of a number of very simple and highly interconnected processors, called neurons, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links that pass signals from one neuron to another. Each link has a numerical weight associated with it. Weights are the basic means of long-term memory in ANNs. They express the strength, or importance, of each neuron input. A neural network 'learns' through repeated adjustments of these weights.

- In the 1940s, Warren McCulloch and Walter Pitts proposed a simple neuron model that is still the basis for most artificial neural networks. The neuron computes the weighted sum of the input signals and compares the result with a threshold value. If the net input is less than the threshold, the neuron output is −1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value +1.

- Frank Rosenblatt suggested the simplest form of a neural network, which he called a perceptron. The operation of the perceptron is based on the McCulloch and Pitts neuron model. It consists of a single neuron with adjustable synaptic weights and a hard limiter. The perceptron learns its task by making small adjustments in the weights to reduce the difference between the actual and desired outputs. The initial weights are randomly assigned and then updated to obtain the output consistent with the training examples.

- A perceptron can learn only linearly separable functions and cannot make global generalisations on the basis of examples learned locally. The limitations of Rosenblatt's perceptron can be overcome by advanced forms of neural networks, such as multilayer perceptrons trained with the back-propagation algorithm.

- A multilayer perceptron is a feedforward neural network with an input layer of source neurons, at least one middle or hidden layer of computational neurons, and an output layer of computational neurons. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. The hidden layer detects the feature. The weights of the neurons in the hidden layer represent the features in the input patterns. The output layer establishes the output pattern of the entire network.

- Learning in a multilayer network proceeds in the same way as in a perceptron. The learning algorithm has two phases. First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer. If it is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

- Although widely used, back-propagation learning is not without problems. Because the calculations are extensive and, as a result, training is slow, a pure back-propagation algorithm is rarely used in practical applications. There are several possible ways to improve computational efficiency. A multilayer network learns much faster when the sigmoidal activation function is represented by a hyperbolic tangent. The use of momentum and adaptive learning rate also significantly improves the performance of a multilayer back-propagation neural network.

- While multilayer back-propagation neural networks are used for pattern recognition problems, the associative memory of humans is emulated by a different type of network called recurrent: a recurrent network, which has feedback loops from its outputs to its inputs. John Hopfield formulated the

physical principle of storing information in a dynamically stable network, and also proposed a single-layer recurrent network using McCulloch and Pitts neurons with the sign activation function.

- The Hopfield network training algorithm has two basic phases: storage and retrieval. In the first phase, the network is required to store a set of states, or fundamental memories, determined by the current outputs of all neurons. This is achieved by calculating the network's weight matrix. Once the weights are calculated, they remain fixed. In the second phase, an unknown corrupted or incomplete version of the fundamental memory is presented to the network. The network output is calculated and fed back to adjust the input. This process is repeated until the output becomes constant. For the fundamental memories to be retrievable, the storage capacity of the Hopfield network has to be kept small.

- The Hopfield network represents an autoassociative type of memory. It can retrieve a corrupted or incomplete memory but cannot associate one memory with another. To overcome this limitation, Bart Kosko proposed the bidirectional associative memory (BAM). BAM is a heteroassociative network. It associates patterns from one set to patterns from another set and vice versa. As with a Hopfield network, the BAM can generalise and produce correct outputs despite corrupted or incomplete inputs. The basic BAM architecture consists of two fully connected layers – an input layer and an output layer.

- The idea behind the BAM is to store pattern pairs so that when n-dimensional vector $\mathbf{X}$ from set A is presented as input, the BAM recalls m-dimensional vector $\mathbf{Y}$ from set B, but when $\mathbf{Y}$ is presented as input, the BAM recalls $\mathbf{X}$. The constraints on the storage capacity of the Hopfield network can also be extended to the BAM. The number of associations to be stored in the BAM should not exceed the number of neurons in the smaller layer. Another problem is incorrect convergence, that is, the BAM may not always produce the closest association.

- In contrast to supervised learning, or learning with an external 'teacher' who presents a training set to the network, unsupervised or self-organised learning does not require a teacher. During a training session, the neural network receives a number of different input patterns, discovers significant features in these patterns and learns how to classify input.

- Hebb's Law, introduced by Donald Hebb in the late 1940s, states that if neuron i is near enough to excite neuron j and repeatedly participates in its activation, the synaptic connection between these two neurons is strengthened and neuron j becomes more sensitive to stimuli from neuron i. This law provides the basis for learning without a teacher. Learning here is a local phenomenon occurring without feedback from the environment.

- Another popular type of unsupervised learning is competitive learning. In competitive learning, neurons compete among themselves to become active. The output neuron that wins the 'competition' is called the winner-takes-all

neuron. Although competitive learning was proposed in the early 1970s, it was largely ignored until the late 1980s, when Teuvo Kohonen introduced a special class of artificial neural networks called self-organising feature maps. He also formulated the principle of topographic map formation which states that the spatial location of an output neuron in the topographic map corresponds to a particular feature of the input pattern.

- The Kohonen network consists of a single layer of computation neurons, but it has two different types of connections. There are forward connections from the neurons in the input layer to the neurons in the output layer, and lateral connections between neurons in the output layer. The lateral connections are used to create a competition between neurons. In the Kohonen network, a neuron learns by shifting its weights from inactive connections to active ones. Only the winning neuron and its neighbourhood are allowed to learn. If a neuron does not respond to a given input pattern, then learning does not occur in that neuron.

Questions for review

1 How does an artificial neural network model the brain? Describe two major classes of learning paradigms: supervised learning and unsupervised (self-organised) learning. What are the features that distinguish these two paradigms from each other?

2 What are the problems with using a perceptron as a biological model? How does the perceptron learn? Demonstrate perceptron learning of the binary logic function OR. Why can the perceptron learn only linearly separable functions?

3 What is a fully connected multilayer perceptron? Construct a multilayer perceptron with an input layer of six neurons, a hidden layer of four neurons and an output layer of two neurons. What is a hidden layer for, and what does it hide?

4 How does a multilayer neural network learn? Derive the back-propagation training algorithm. Demonstrate multilayer network learning of the binary logic function Exclusive-OR.

5 What are the main problems with the back-propagation learning algorithm? How can learning be accelerated in multilayer neural networks? Define the generalised delta rule.

6 What is a recurrent neural network? How does it learn? Construct a single six-neuron Hopfield network and explain its operation. What is a fundamental memory?

7 Derive the Hopfield network training algorithm. Demonstrate how to store three fundamental memories in the six-neuron Hopfield network.

8 The delta rule and Hebb's rule represent two different methods of learning in neural networks. Explain the differences between these two rules.

9 What is the difference between autoassociative and heteroassociative types of memory? What is the bidirectional associative memory (BAM)? How does the BAM work?

10 Derive the BAM training algorithm. What constraints are imposed on the storage capacity of the BAM? Compare the BAM storage capacity with the storage capacity of the Hopfield network.

11 What does Hebb's Law represent? Derive the activity product rule and the generalised activity product rule. What is the meaning of the forgetting factor? Derive the generalised Hebbian learning algorithm.

12 What is competitive learning? What are the differences between Hebbian and competitive learning paradigms? Describe the feature-mapping Kohonen model. Derive the competitive learning algorithm.

References

Amit, D.J. (1989). *Modelling Brain Functions: The World of Attractor Neural Networks.* Cambridge University Press, New York.

Bryson, A.E. and Ho, Y.C. (1969). *Applied Optimal Control.* Blaisdell, New York.

Caudill, M. (1991). Neural network training tips and techniques, *AI Expert*, January, 56–61.

Cohen, M.H. and Grossberg, S. (1983). Absolute stability of global pattern formation and parallel memory storage by competitive networks, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13, 815–826.

Grossberg, S. (1972). Neural expectation: cerebellar and retinal analogs of cells fired by learnable or unlearned pattern classes, *Kybernetik*, 10, 49–57.

Guyon, I.P. (1991). Applications of neural networks to character recognition, *International Journal of Pattern Recognition and Artificial Intelligence*, 5, 353–382.

Fu, L.M. (1994). *Neural Networks in Computer Intelligence.* McGraw-Hill Book, Inc., Singapore.

Fukushima, K. (1975). Cognition: a self-organizing multilayered neural network, *Biological Cybernetics*, 20, 121–136.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation,* 2nd edn. Prentice Hall, Englewood Cliffs, NJ.

Hebb, D.O. (1949). *The Organisation of Behaviour: A Neuropsychological Theory.* John Wiley, New York.

Hopfield, J.J. (1982). Neural networks and physical systems with emergent collective computational abilities, *Proceedings of the National Academy of Sciences of the USA*, 79, 2554–2558.

Jacobs, R.A. (1988). Increased rates of convergence through learning rate adaptation, *Neural Networks*, 1, 295–307.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps, *Biological Cybernetics*, 43, 59–69.

Kohonen, T. (1989). *Self-Organization and Associative Memory*, 3rd edn. Springer-Verlag, Berlin, Heidelberg.

Kohonen, T. (1990). The self-organizing map, *Proceedings of the IEEE*, 78, 1464–1480.

Kosko, B. (1987). Adaptive bidirectional associative memories, *Applied Optics*, 26(23), 4947–4960.

Kosko, B. (1988). Bidirectional associative memories, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-18, 49–60.

Kosko, B. (1992). *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. Prentice Hall, Englewood Cliffs, NJ.

McCulloch, W.S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics*, 5, 115–137.

Medsker, L.R. and Liebowitz, J. (1994). *Design and Development of Expert Systems and Neural Computing*. Macmillan College Publishing Company, New York.

Minsky, M.L. and Papert, S.A. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

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

Rosenblatt, F. (1960). Perceptron simulation experiments, *Proceedings of the Institute of Radio Engineers*, 48, 301–309.

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

Shepherd, G.M. and Koch, C. (1990). Introduction to synaptic circuits, *The Synaptic Organisation of the Brain*, G.M. Shepherd, ed., Oxford University Press, New York, pp. 3–31.

Shynk, J.J. (1990). Performance surfaces of a single-layer perceptron, *IEEE Transactions on Neural Networks*, 1, 268–274.

Shynk, J.J. and Bershad, N.J. (1992). Stationary points and performance surfaces of a perceptron learning algorithm for a nonstationary data model, *Proceedings of the International Joint Conference on Neural Networks*, Baltimore, MD, vol. 2, pp. 133–139.

Stent, G.S. (1973). A physiological mechanism for Hebb's postulate of learning, *Proceedings of the National Academy of Sciences of the USA*, 70, 997–1001.

Stork, D. (1989). Is backpropagation biologically plausible?, *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, vol. 2, pp. 241–246.

Stubbs, D.F. (1990). Six ways to improve back-propagation results, *Journal of Neural Network Computing*, Spring, 64–67.

Touretzky, D.S. and Pomerlean, D.A. (1989). What is hidden in the hidden layers?, *Byte*, 14, 227–233.

Von der Malsburg, C. (1973). Self-organisation of orientation sensitive cells in the striate cortex, *Kybernetik*, 14, 85–100.

Watrous, R.L. (1987). Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimisation, *Proceedings of the First IEEE International Conference on Neural Networks*, San Diego, CA, vol. 2, pp. 619–627.

Evolutionary computation **7**

In which we consider the field of evolutionary computation, including genetic algorithms, evolution strategies and genetic programming, and their applications to machine learning.

7.1 Introduction, or can evolution be intelligent?

Intelligence can be defined as the capability of a system to adapt its behaviour to an ever-changing environment. According to Alan Turing (Turing, 1950), the form or appearance of a system is irrelevant to its intelligence. However, from our everyday experience we know that evidences of intelligent behaviour are easily observed in humans. But we are products of evolution, and thus by modelling the process of evolution, we might expect to create intelligent behaviour. Evolutionary computation simulates evolution on a computer. The result of such a simulation is a series of optimisation algorithms, usually based on a simple set of rules. Optimisation iteratively improves the quality of solutions until an optimal, or at least feasible, solution is found.

But is evolution really intelligent? We can consider the behaviour of an individual organism as an inductive inference about some yet unknown aspects of its environment (Fogel *et al.*, 1966). Then if, over successive generations, the organism survives, we can say that this organism is capable of learning to predict changes in its environment. Evolution is a tortuously slow process from the human perspective, but the simulation of evolution on a computer does not take billions of years!

The evolutionary approach to machine learning is based on computational models of natural selection and genetics. We call them **evolutionary computation**, an umbrella term that combines **genetic algorithms**, **evolution strategies** and **genetic programming**. All these techniques simulate evolution by using the processes of **selection**, **mutation** and **reproduction**.

7.2 Simulation of natural evolution

On 1 July 1858, Charles Darwin presented his theory of evolution before the Linnean Society of London. This day marks the beginning of a revolution in

biology. Darwin's classical theory of evolution, together with Weismann's theory of natural selection and Mendel's concept of genetics, now represent the neo-Darwinian paradigm (Keeton, 1980; Mayr, 1988).

Neo-Darwinism is based on processes of reproduction, mutation, competition and selection. The power to reproduce appears to be an essential property of life. The power to mutate is also guaranteed in any living organism that reproduces itself in a continuously changing environment. Processes of competition and selection normally take place in the natural world, where expanding populations of different species are limited by a finite space.

If the process of evolution is to be emulated on a computer, what is being optimised by evolution in natural life? Evolution can be seen as a process leading to the maintenance or increase of a population's ability to survive and reproduce in a specific environment (Hartl and Clark, 1989). This ability is called **evolutionary fitness**. Although fitness cannot be measured directly, it can be estimated on the basis of the ecology and functional morphology of the organism in its environment (Hoffman, 1989). Evolutionary fitness can also be viewed as a measure of the organism's ability to anticipate changes in its environment (Atmar, 1994). Thus, the fitness, or the quantitative measure of the ability to predict environmental changes and respond adequately, can be considered as the quality that is being optimised in natural life.

To illustrate fitness, we can use the concept of **adaptive topology** (Wright, 1932). We can represent a given environment by a landscape where each peak corresponds to the optimised fitness of a species. As evolution takes place, each species of a given population moves up the slopes of the landscape towards the peaks. Environmental conditions change over time, and thus the species have to continuously adjust their routes. As a result, only the fittest can reach the peaks.

Adaptive topology is a continuous function; it simulates the fact that the environment, or natural topology, is not static. The shape of the topology changes over time, and all species continually undergo selection. The goal of evolution is to generate a population of individuals with increasing fitness.

But how is a population with increasing fitness generated? Michalewicz (1996) suggests a simple explanation based on a population of rabbits. Some rabbits are faster than others, and we may say that these rabbits possess superior fitness because they have a greater chance of avoiding foxes, surviving and then breeding. Of course, some of the slower rabbits may survive too. As a result, some slow rabbits breed with fast rabbits, some fast with other fast rabbits, and some slow rabbits with other slow rabbits. In other words, the breeding generates a mixture of rabbit genes. If two parents have superior fitness, there is a good chance that a combination of their genes will produce an offspring with even higher fitness. Over time the entire population of rabbits becomes faster to meet their environmental challenges in the face of foxes. However, environmental conditions could change in favour of say, fat but smart rabbits. To optimise survival, the genetic structure of the rabbit population will change accordingly. At the same time, faster and smarter rabbits encourage the breeding

of faster and smarter foxes. Natural evolution is a continuous, never-ending process.

Can we simulate the process of natural evolution in a computer?

Several different methods of evolutionary computation are now known. They all simulate natural evolution, generally by creating a population of individuals, evaluating their fitness, generating a new population through genetic operations, and repeating this process a number of times. However, there are different ways of performing evolutionary computation. We will start with **genetic algorithms** (GAs) as most of the other evolutionary algorithms can be viewed as variations of GAs.

In the early 1970s, John Holland, one of the founders of evolutionary computation, introduced the concept of genetic algorithms (Holland, 1975). His aim was to make computers do what nature does. As a computer scientist, Holland was concerned with algorithms that manipulate strings of binary digits. He viewed these algorithms as an abstract form of natural evolution. Holland's GA can be represented by a sequence of procedural steps for moving from one population of artificial 'chromosomes' to a new population. It uses 'natural' selection and genetics-inspired techniques known as crossover and mutation. Each chromosome consists of a number of 'genes', and each gene is represented by 0 or 1, as shown in Figure 7.1.

Nature has an ability to adapt and learn without being told what to do. In other words, nature finds good chromosomes blindly. GAs do the same. Two mechanisms link a GA to the problem it is solving: **encoding** and **evaluation**.

In Holland's work, encoding is carried out by representing chromosomes as strings of ones and zeros. Although many other types of encoding techniques have been invented (Davis, 1991), no one type works best for all problems. We will use bit strings as the most popular technique.

An evaluation function is used to measure the chromosome's performance, or fitness, for the problem to be solved (an evaluation function in GAs plays the same role the environment plays in natural evolution). The GA uses a measure of fitness of individual chromosomes to carry out reproduction. As reproduction takes place, the crossover operator exchanges parts of two single chromosomes, and the mutation operator changes the gene value in some randomly chosen location of the chromosome. As a result, after a number of successive reproductions, the less fit chromosomes become extinct, while those best able to survive gradually come to dominate the population. It is a simple approach, yet even crude reproduction mechanisms display highly complex behaviour and are capable of solving some difficult problems.

Let us now discuss genetic algorithms in more detail.
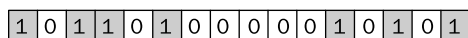
Figure 7.1   A 16-bit binary string of an artificial chromosome

7.3 Genetic algorithms

We start with a definition: genetic algorithms are a class of stochastic search algorithms based on biological evolution. Given a clearly defined problem to be solved and a binary string representation for candidate solutions, a basic GA can be represented as in Figure 7.2. A GA applies the following major steps (Davis, 1991; Mitchell, 1996):

Step 1: Represent the problem variable domain as a chromosome of a fixed length, choose the size of a chromosome population N, the crossover probability p_c and the mutation probability p_m.

Step 2: Define a fitness function to measure the performance, or fitness, of an individual chromosome in the problem domain. The fitness function establishes the basis for selecting chromosomes that will be mated during reproduction.

Step 3: Randomly generate an initial population of chromosomes of size N:

$$x_1, x_2, \ldots, x_N$$

Step 4: Calculate the fitness of each individual chromosome:

$$f(x_1), f(x_2), \ldots, f(x_N)$$

Step 5: Select a pair of chromosomes for mating from the current population. Parent chromosomes are selected with a probability related to their fitness. Highly fit chromosomes have a higher probability of being selected for mating than less fit chromosomes.

Step 6: Create a pair of offspring chromosomes by applying the genetic operators – crossover and mutation.

Step 7: Place the created offspring chromosomes in the new population.

Step 8: Repeat Step 5 until the size of the new chromosome population becomes equal to the size of the initial population, N.

Step 9: Replace the initial (parent) chromosome population with the new (offspring) population.

Step 10: Go to Step 4, and repeat the process until the termination criterion is satisfied.

As we see, a GA represents an iterative process. Each iteration is called a **generation**. A typical number of generations for a simple GA can range from 50 to over 500 (Mitchell, 1996). The entire set of generations is called a **run**. At the end of a run, we expect to find one or more highly fit chromosomes.
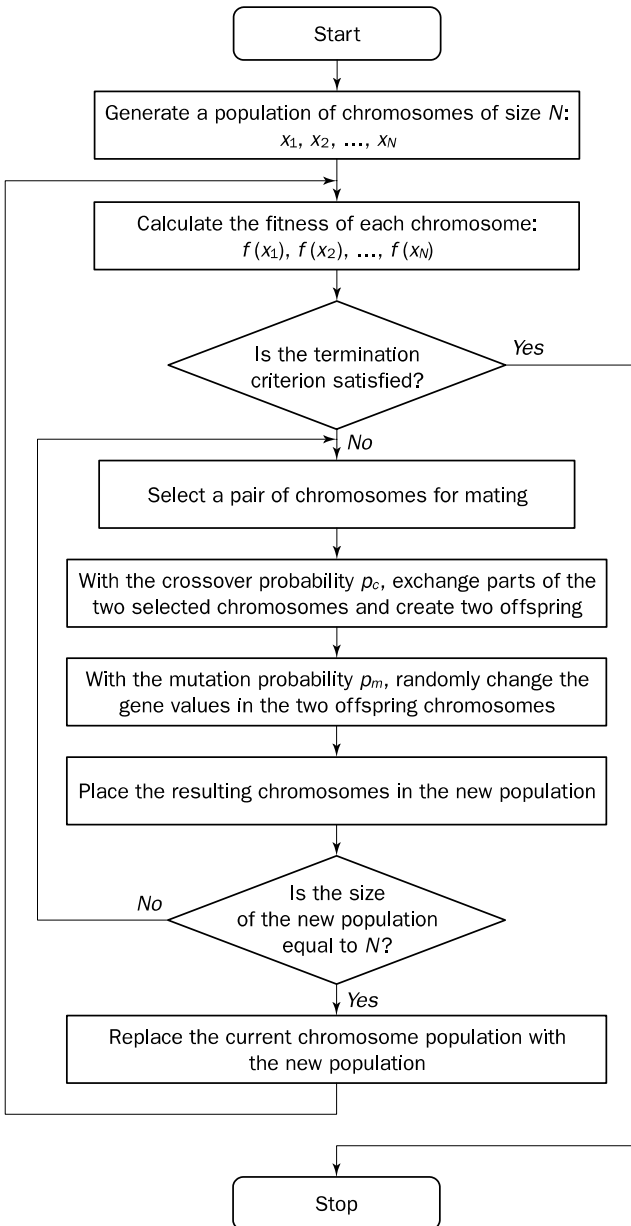
Figure 7.2   A basic genetic algorithm

Are any conventional termination criteria used in genetic algorithms?

Because GAs use a stochastic search method, the fitness of a population may remain stable for a number of generations before a superior chromosome appears. This makes applying conventional termination criteria problematic. A common practice is to terminate a GA after a specified number of generations

and then examine the best chromosomes in the population. If no satisfactory solution is found, the GA is restarted.

A simple example will help us to understand how a GA works. Let us find the maximum value of the function $(15x - x^2)$ where parameter x varies between 0 and 15. For simplicity, we may assume that x takes only integer values. Thus, chromosomes can be built with only four genes:

Integer	Binary code	Integer	Binary code	Integer	Binary code
1	0 0 0 1	6	0 1 1 0	11	1 0 1 1
2	0 0 1 0	7	0 1 1 1	12	1 1 0 0
3	0 0 1 1	8	1 0 0 0	13	1 1 0 1
4	0 1 0 0	9	1 0 0 1	14	1 1 1 0
5	0 1 0 1	10	1 0 1 0	15	1 1 1 1

Suppose that the size of the chromosome population N is 6, the crossover probability p_c equals 0.7, and the mutation probability p_m equals 0.001. (The values chosen for p_c and p_m are fairly typical in GAs.) The fitness function in our example is defined by

$$f(x) = 15x - x^2$$

The GA creates an initial population of chromosomes by filling six 4-bit strings with randomly generated ones and zeros. The initial population might look like that shown in Table 7.1. The chromosomes' initial locations on the fitness function are illustrated in Figure 7.3(a).

A real practical problem would typically have a population of thousands of chromosomes.

The next step is to calculate the fitness of each individual chromosome. The results are also shown in Table 7.1. The average fitness of the initial population is 36. In order to improve it, the initial population is modified by using selection, crossover and mutation, the genetic operators.

In natural selection, only the fittest species can survive, breed, and thereby pass their genes on to the next generation. GAs use a similar approach, but

Table 7.1   The initial randomly generated population of chromosomes

Chromosome label	Chromosome string	Decoded integer	Chromosome fitness	Fitness ratio, %
X1	1 1 0 0	12	36	16.5
X2	0 1 0 0	4	44	20.2
X3	0 0 0 1	1	14	6.4
X4	1 1 1 0	14	14	6.4
X5	0 1 1 1	7	56	25.7
X6	1 0 0 1	9	54	24.8
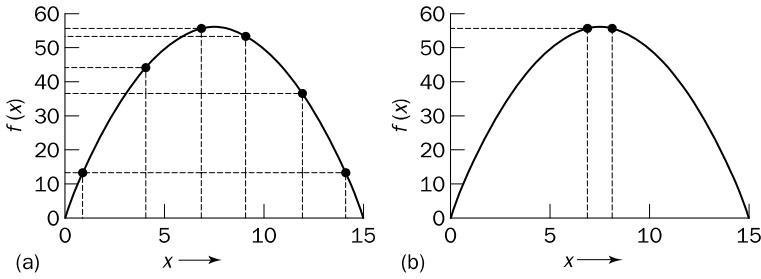
Figure 7.3 The fitness function and chromosome locations: (a) chromosome initial locations; (b) chromosome final locations

unlike nature, the size of the chromosome population remains unchanged from one generation to the next.

How can we maintain the size of the population constant, and at the same time improve its average fitness?

The last column in Table 7.1 shows the ratio of the individual chromosome's fitness to the population's total fitness. This ratio determines the chromosome's chance of being selected for mating. Thus, the chromosomes X5 and X6 stand a fair chance, while the chromosomes X3 and X4 have a very low probability of being selected. As a result, the chromosome's average fitness improves from one generation to the next.

One of the most commonly used chromosome selection techniques is the **roulette wheel selection** (Goldberg, 1989; Davis, 1991). Figure 7.4 illustrates the roulette wheel for our example. As you can see, each chromosome is given a slice of a circular roulette wheel. The area of the slice within the wheel is equal to the chromosome fitness ratio (see Table 7.1). For instance, the chromosomes X5 and X6 (the most fit chromosomes) occupy the largest areas, whereas the chromosomes X3 and X4 (the least fit) have much smaller segments in the roulette wheel. To select a chromosome for mating, a random number is generated in the interval [0, 100], and the chromosome whose segment spans the random number is selected. It is like spinning a roulette wheel where each chromosome has a segment on the wheel proportional to its fitness. The roulette wheel is spun, and
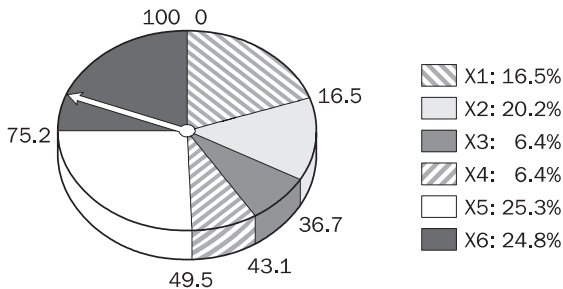
Figure 7.4 Roulette wheel selection

when the arrow comes to rest on one of the segments, the corresponding chromosome is selected.

In our example, we have an initial population of six chromosomes. Thus, to establish the same population in the next generation, the roulette wheel would be spun six times. The first two spins might select chromosomes X6 and X2 to become parents, the second pair of spins might choose chromosomes X1 and X5, and the last two spins might select chromosomes X2 and X5.

Once a pair of parent chromosomes is selected, the crossover operator is applied.

How does the crossover operator work?

First, the crossover operator randomly chooses a crossover point where two parent chromosomes 'break', and then exchanges the chromosome parts after that point. As a result, two new offspring are created. For example, the chromosomes X6 and X2 could be crossed over after the second gene in each to produce the two offspring, as shown in Figure 7.5.

If a pair of chromosomes does not cross over, then chromosome **cloning** takes place, and the offspring are created as exact copies of each parent. For example, the parent chromosomes X2 and X5 may not cross over. Instead, they create the offspring that are their exact copies, as shown in Figure 7.5.

A value of 0.7 for the crossover probability generally produces good results. After selection and crossover, the average fitness of the chromosome population has improved and gone from 36 to 42.

What does mutation represent?

Mutation, which is rare in nature, represents a change in the gene. It may lead to a significant improvement in fitness, but more often has rather harmful results.

So why use mutation at all? Holland introduced mutation as a background operator (Holland, 1975). Its role is to provide a guarantee that the search algorithm is not trapped on a local optimum. The sequence of selection and crossover operations may stagnate at any homogeneous set of solutions. Under such conditions, all chromosomes are identical, and thus the average fitness of the population cannot be improved. However, the solution might appear to become optimal, or rather locally optimal, only because the search algorithm is not able to proceed any further. Mutation is equivalent to a random search, and aids us in avoiding loss of genetic diversity.

How does the mutation operator work?

The mutation operator flips a randomly selected gene in a chromosome. For example, the chromosome X1′ might be mutated in its second gene, and the chromosome X2 in its third gene, as shown in Figure 7.5. Mutation can occur at any gene in a chromosome with some probability. The mutation probability is quite small in nature, and is kept quite low for GAs, typically in the range between 0.001 and 0.01.

Genetic algorithms assure the continuous improvement of the average fitness of the population, and after a number of generations (typically several hundred)
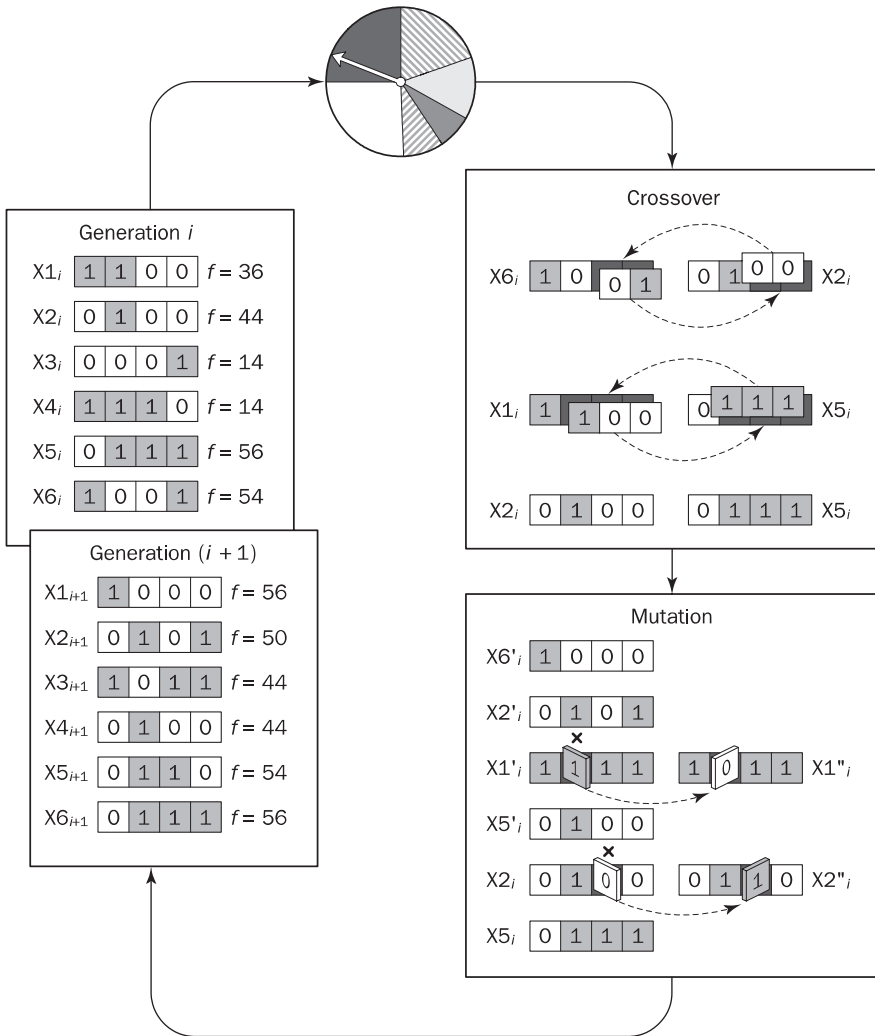
Figure 7.5 The GA cycle

the population evolves to a **near-optimal** solution. In our example, the final population would consist of only chromosomes $\boxed{0\,1\,1\,1}$ and $\boxed{1\,0\,0\,0}$. The chromosome's final locations on the fitness function are illustrated in Figure 7.3(b).

In this example, the problem has only one variable. It is easy to represent. But suppose it is desired to find the maximum of the 'peak' function of two variables:

$$f(x,y) = (1 - x)^2 e^{-x^2 - (y+1)^2} - (x - x^3 - y^3)e^{-x^2 - y^2},$$

where parameters x and y vary between -3 and 3.

The first step is to represent the problem variables as a chromosome. In other words, we represent parameters x and y as a concatenated binary string:

| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |

$\underbrace{\hspace{4cm}}_{x}$ $\underbrace{\hspace{4cm}}_{y}$
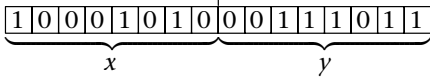
in which each parameter is represented by eight binary bits.

Then, we choose the size of the chromosome population, for instance 6, and randomly generate an initial population.

The next step is to calculate the fitness of each chromosome. This is done in two stages. First, a chromosome is decoded by converting it into two real numbers, x and y, in the interval between -3 and 3. Then the decoded values of x and y are substituted into the 'peak' function.

How is decoding done?

First, a chromosome, that is a string of 16 bits, is partitioned into two 8-bit strings:

| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | and | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
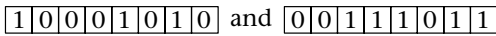
Then these strings are converted from binary (base 2) to decimal (base 10):

$$(10001010)_2 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= (138)_{10}$$

and

$$(00111011)_2 = 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$= (59)_{10}$$

Now the range of integers that can be handled by 8-bits, that is the range from 0 to $(2^8 - 1)$, is mapped to the actual range of parameters x and y, that is the range from -3 to 3:

$$\frac{6}{256 - 1} = 0.0235294$$

To obtain the actual values of x and y, we multiply their decimal values by 0.0235294 and subtract 3 from the results:

$$x = (138)_{10} \times 0.0235294 - 3 = 0.2470588$$

and

$$y = (59)_{10} \times 0.0235294 - 3 = -1.6117647$$

When necessary, we can also apply other decoding techniques, such as **Gray coding** (Caruana and Schaffer, 1988).

Using decoded values of x and y as inputs in the mathematical function, the GA calculates the fitness of each chromosome.

To find the maximum of the 'peak' function, we will use crossover with the probability equal to 0.7 and mutation with the probability equal to 0.001. As we mentioned earlier, a common practice in GAs is to specify the number of generations. Suppose the desired number of generations is 100. That is, the GA will create 100 generations of 6 chromosomes before stopping.

Figure 7.6(a) shows the initial locations of the chromosomes on the surface and contour plot of the 'peak' function. Each chromosome here is represented by a sphere. The initial population consists of randomly generated individuals that are dissimilar or **heterogeneous**. However, starting from the second generation, crossover begins to recombine features of the best chromosomes, and the population begins to converge on the peak containing the maximum, as shown in Figure 7.6(b). From then until the final generation, the GA is searching around this peak with mutation, resulting in diversity. Figure 7.6(c) shows the final chromosome generation. However, the population has converged on a chromosome lying on a local maximum of the 'peak' function.

But we are looking for the global maximum, so can we be sure the search is for the optimal solution? The most serious problem in the use of GAs is concerned with the quality of the results, in particular whether or not an optimal solution is
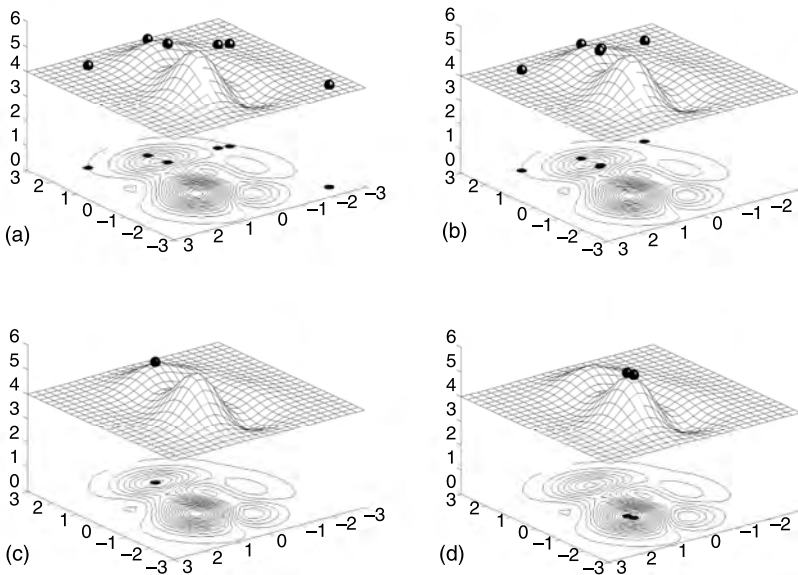
Figure 7.6 Chromosome locations on the surface and contour plot of the 'peak' function: (a) initial population; (b) first generation; (c) local maximum solution; (d) global maximum solution

being reached. One way of providing some degree of insurance is to compare results obtained under different rates of mutation. Let us, for example, increase the mutation rate to 0.01 and rerun the GA. The population might now converge on the chromosomes shown in Figure 7.6(d). However, to be sure of steady results we must increase the size of the chromosome population.
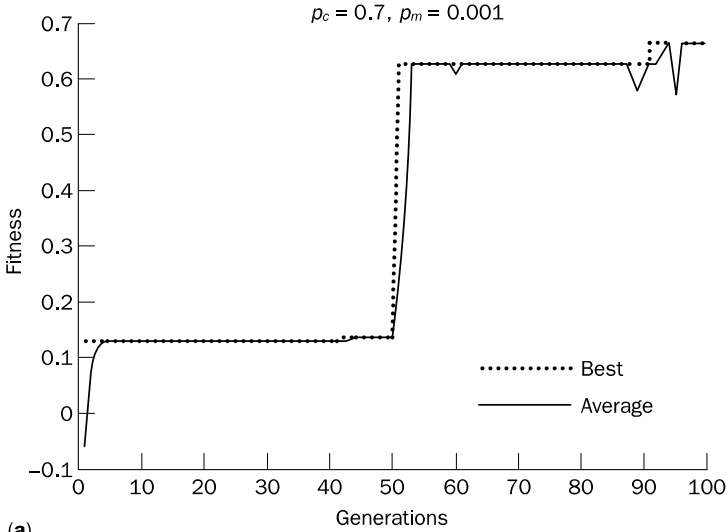
(a)
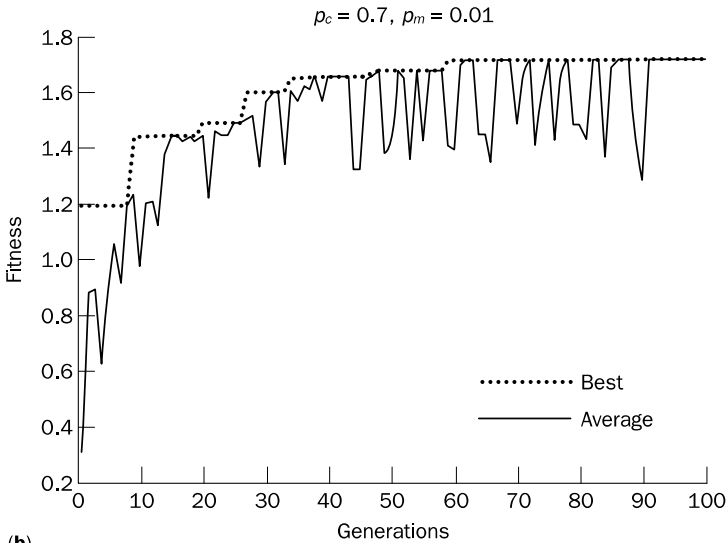
(b)

Figure 7.7 Performance graphs for 100 generations of 6 chromosomes: (a) local maximum solution and (b) global maximum solution of the 'peak' function

A surface of a mathematical function of the sort given in Figure 7.6 is a convenient medium for displaying the GA's performance. However, fitness functions for real world problems cannot be easily represented graphically. Instead, we can use **performance graphs**.

What is a performance graph?

Since genetic algorithms are stochastic, their performance usually varies from generation to generation. As a result, a curve showing the average performance of the entire population of chromosomes as well as a curve showing the performance of the best individual in the population is a useful way of examining the behaviour of a GA over the chosen number of generations.

Figures 7.7(a) and (b) show plots of the best and average values of the fitness function across 100 generations. The x-axis of the performance graph indicates how many generations have been created and evaluated at the particular point in the run, and the y-axis displays the value of the fitness function at that point.

The erratic behaviour of the average performance curves is due to mutation. The mutation operator allows a GA to explore the landscape in a random manner. Mutation may lead to significant improvement in the population fitness, but more often decreases it. To ensure diversity and at the same time to reduce the harmful effects of mutation, we can increase the size of the chromosome population. Figure 7.8 shows performance graphs for 20 generations of 60 chromosomes. The best and average curves represented here are typical for GAs. As you can see, the average curve rises rapidly at the beginning of the run, but then as the population converges on the nearly optimal solution, it rises more slowly, and finally flattens at the end.
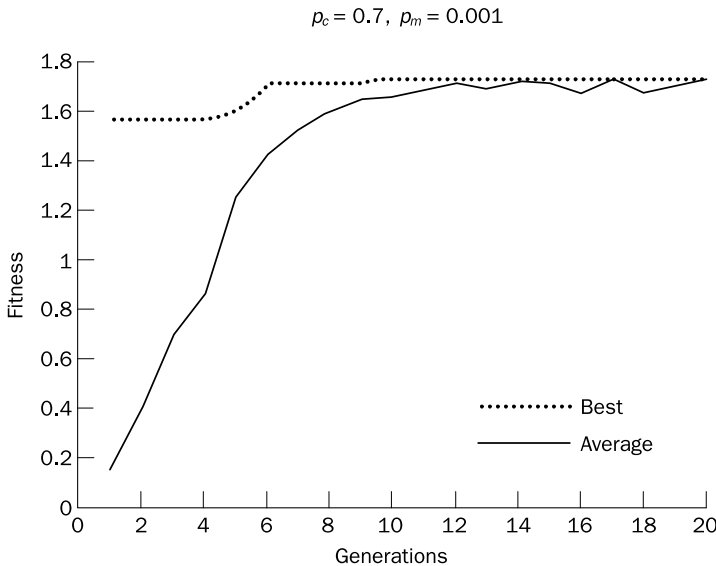
Figure 7.8   Performance graphs for 20 generations of 60 chromosomes

7.4 Why genetic algorithms work

The GA techniques have a solid theoretical foundation (Holland, 1975; Goldberg, 1989; Rawlins, 1991; Whitley, 1993). That foundation is based on the **Schema Theorem**.

John Holland introduced the notation of **schema** (Holland, 1975), which came from the Greek word meaning 'form'. A schema is a set of bit strings of ones, zeros and asterisks, where each asterisk can assume either value 1 or 0. The ones and zeros represent the fixed positions of a schema, while asterisks represent 'wild cards'. For example, the schema $\boxed{1}\boxed{*}\boxed{*}\boxed{0}$ stands for a set of 4-bit strings. Each string in this set begins with 1 and ends with 0. These strings are called **instances** of the schema.

What is the relationship between a schema and a chromosome?
It is simple. A chromosome matches a schema when the fixed positions in the schema match the corresponding positions in the chromosome. For example, the schema H

$$\boxed{1}\boxed{*}\boxed{*}\boxed{0}$$

matches the following set of 4-bit chromosomes:

$$\boxed{1}\boxed{1}\boxed{1}\boxed{0}$$
$$\boxed{1}\boxed{1}\boxed{0}\boxed{0}$$
$$\boxed{1}\boxed{0}\boxed{1}\boxed{0}$$
$$\boxed{1}\boxed{0}\boxed{0}\boxed{0}$$

Each chromosome here begins with 1 and ends with 0. These chromosomes are said to be instances of the schema H.

The number of defined bits (non-asterisks) in a schema is called the **order**. The schema H, for example, has two defined bits, and thus its order is 2.

In short, genetic algorithms manipulate **schemata** (schemata is the plural of the word schema) when they run. If GAs use a technique that makes the probability of reproduction proportional to chromosome fitness, then according to the Schema Theorem (Holland, 1975), we can predict the presence of a given schema in the next chromosome generation. In other words, we can describe the GA's behaviour in terms of the increase or decrease in the number of instances of a given schema (Goldberg, 1989).

Let us assume that at least one instance of the schema H is present in the chromosome initial generation i. Now let $m_H(i)$ be the number of instances of the schema H in the generation i, and $\hat{f}_H(i)$ be the average fitness of these instances. We want to calculate the number of instances in the next generation, $m_H(i+1)$. As the probability of reproduction is proportional to chromosome fitness, we can easily calculate the expected number of offspring of a chromosome x in the next generation:

$$m_x(i+1) = \frac{f_x(i)}{\hat{f}(i)},$$ (7.1)

where $f_x(i)$ is the fitness of the chromosome x, and $\hat{f}(i)$ is the average fitness of the chromosome initial generation i.

Then, assuming that the chromosome x is an instance of the schema H, we obtain

$$m_H(i+1) = \frac{\sum_{x=1}^{x=m_H(i)} f_x(i)}{\hat{f}(i)}, \quad x \in H$$ (7.2)

Since, by definition,

$$\hat{f}_H(i) = \frac{\sum_{x=1}^{x=m_H(i)} f_x(i)}{m_H(i)},$$

we obtain

$$m_H(i+1) = \frac{\hat{f}_H(i)}{\hat{f}(i)} m_H(i)$$ (7.3)

Thus, a schema with above-average fitness will indeed tend to occur more frequently in the next generation of chromosomes, and a schema with below-average fitness will tend to occur less frequently.

How about effects caused by crossover and mutation?

Crossover and mutation can both create and destroy instances of a schema. Here we will consider only destructive effects, that is effects that decrease the number of instances of the schema H. Let us first quantify the destruction caused by the crossover operator. The schema will survive after crossover if at least one of its offspring is also its instance. This is the case when crossover does not occur within the defining length of the schema.

What is the defining length of a schema?

The distance between the outermost defined bits of a schema is called **defining length**. For example, the defining length of $\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{1}\boxed{0}\boxed{1}\boxed{1}$ is 3, of $\boxed{*}\boxed{0}\boxed{*}\boxed{1}\boxed{*}\boxed{1}\boxed{0}\boxed{*}$ is 5 and of $\boxed{1}\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{*}\boxed{0}$ is 7.

If crossover takes place within the defining length, the schema H can be destroyed and offspring that are not instances of H can be created. (Although the schema H will not be destroyed if two identical chromosomes cross over, even when crossover occurs within the defining length.)

Thus, the probability that the schema H will survive after crossover can be defined as:

$$P_H^{(c)} = 1 - p_c \left(\frac{l_d}{l-1} \right), \tag{7.4}$$

where p_c is the crossover probability, and l and l_d are, respectively, the length and the defining length of the schema H.

It is clear, that the probability of survival under crossover is higher for short schemata rather than for long ones.

Now consider the destructive effects of mutation. Let p_m be the mutation probability for any bit of the schema H, and n be the order of the schema H. Then $(1 - p_m)$ represents the probability that the bit will not be mutated, and thus the probability that the schema H will survive after mutation is determined as:

$$P_H^{(m)} = (1 - p_m)^n \tag{7.5}$$

It is also clear that the probability of survival under mutation is higher for low-order schemata than for high-order ones.

We can now amend Eq. (7.3) to take into account the destructive effects of crossover and mutation:

$$m_H(i+1) = \frac{\hat{f}_H(i)}{\hat{f}(i)} m_H(i) \left[1 - p_c \left(\frac{l_d}{l-1} \right) \right] (1 - p_m)^n \tag{7.6}$$

This equation describes the growth of a schema from one generation to the next. It is known as the Schema Theorem. Because Eq. (7.6) considers only the destructive effects of crossover and mutation, it gives us a lower bound on the number of instances of the schema H in the next generation.

Despite crossover arguably representing a major advantage of GAs, there is as yet no theoretical basis to support the view that a GA will outperform other search and optimisation techniques just because crossover allows the combination of partial solutions.

Genetic algorithms are a very powerful tool, but need to be applied intelligently. For example, coding the problem as a bit string may change the nature of the problem being investigated. In other words, there is a danger that the coded representation becomes a problem that is different from the one we wanted to solve.

To illustrate the ideas discussed above, we consider a simple application of the GA to problems of scheduling resources.

7.5 Case study: maintenance scheduling with genetic algorithms

One of the most successful areas for GA applications includes the problem of scheduling resources. Scheduling problems are complex and difficult to solve. They are usually approached with a combination of search techniques and heuristics.

Why are scheduling problems so difficult?

First, scheduling belongs to NP-complete problems. Such problems are likely to be unmanageable and cannot be solved by combinatorial search techniques. Moreover, heuristics alone cannot guarantee the best solution.

Second, scheduling problems involve a competition for limited resources; as a result, they are complicated by many constraints. The key to the success of the GA lies in defining a fitness function that incorporates all these constraints.

The problem we discuss here is the maintenance scheduling in modern power systems. This task has to be carried out under several constraints and uncertainties, such as failures and forced outages of power equipment and delays in obtaining spare parts. The schedule often has to be revised at short notice. Human experts usually work out the maintenance scheduling by hand, and there is no guarantee that the optimum or even near-optimum schedule is produced.

A typical process of the GA development includes the following steps:

1   Specify the problem, define constraints and optimum criteria.

2   Represent the problem domain as a chromosome.

3   Define a fitness function to evaluate the chromosome's performance.

4   Construct the genetic operators.

5   Run the GA and tune its parameters.

Step 1:   *Specify the problem, define constraints and optimum criteria*
This is probably the most important step in developing a GA, because if it is not correct and complete a viable schedule cannot be obtained.

Power system components are made to operate continuously throughout their life by means of preventive maintenance. The purpose of maintenance scheduling is to find the sequence of outages of power units over a given period of time (normally a year) such that the security of a power system is maximised.

Any outage in a power system is associated with some loss in security. The security margin is determined by the system's net reserve. The net reserve, in turn, is defined as the total installed generating capacity of the system *minus* the power lost due to a scheduled outage and *minus* the maximum load forecast during the maintenance period. For instance, if we assume that the total installed capacity is 150 MW

Table 7.2 Power units and their maintenance requirements

Unit number	Unit capacity, MW	Number of intervals required for unit maintenance during one year
1	20	2
2	15	2
3	35	1
4	40	1
5	15	1
6	15	1
7	10	1

and a unit of 20 MW is scheduled for maintenance during the period when the maximum load is predicted to be 100 MW, then the net reserve will be 30 MW. Maintenance scheduling must ensure that sufficient net reserve is provided for secure power supply during any maintenance period.

Suppose, there are seven power units to be maintained in four equal intervals. The maximum loads expected during these intervals are 80, 90, 65 and 70 MW. The unit capacities and their maintenance requirements are presented in Table 7.2.

The constraints for this problem can be specified as follows:

• Maintenance of any unit starts at the beginning of an interval and finishes at the end of the same or adjacent interval. The maintenance cannot be aborted or finished earlier than scheduled.

• The net reserve of the power system must be greater than or equal to zero at any interval.

The optimum criterion here is that the net reserve must be at the maximum during any maintenance period.

Step 2: *Represent the problem domain as a chromosome*

Our scheduling problem is essentially an ordering problem, requiring us to list the tasks in a particular order. A complete schedule may consist of a number of overlapping tasks, but not all orderings are legal, since they may violate the constraints. Our job is to represent a complete schedule as a chromosome of a fixed length.

An obvious coding scheme that comes to mind is to assign each unit a binary number and to let the chromosome be a sequence of these binary numbers. However, an ordering of the units in a sequence is not yet a schedule. Some units can be maintained simultaneously, and we must also incorporate the time required for unit maintenance into the schedule. Thus, rather than ordering units in a sequence, we might build a sequence of maintenance schedules of individual units. The unit schedule can be easily represented as a 4-bit string, where each bit is a maintenance interval. If a unit is to be maintained in a particular

interval, the corresponding bit assumes value 1, otherwise it is 0. For example, the string $\boxed{0}\boxed{1}\boxed{0}\boxed{0}$ presents a schedule for a unit to be maintained in the second interval. It also shows that the number of intervals required for maintenance of this unit is equal to 1. Thus, a complete maintenance schedule for our problem can be represented as a 28-bit chromosome.

However, crossover and mutation operators could easily create binary strings that call for maintaining some units more than once and others not at all. In addition, we could call for maintenance periods that would exceed the number of intervals really required for unit maintenance.

A better approach is to change the chromosome syntax. As already discussed, a chromosome is a collection of elementary parts called genes. Traditionally, each gene is represented by only one bit and cannot be broken into smaller elements. For our problem, we can adopt the same concept, but represent a gene by four bits. In other words, the smallest indivisible part of our chromosome is a 4-bit string. This representation allows crossover and mutation operators to act according to the theoretical grounding of genetic algorithms. What remains to be done is to produce a pool of genes for each unit:

Unit 1:	1 1 0 0	0 1 1 0	0 0 1 1	
Unit 2:	1 1 0 0	0 1 1 0	0 0 1 1	
Unit 3:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 4:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 5:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 6:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Unit 7:	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1

The GA can now create an initial population of chromosomes by filling 7-gene chromosomes with genes randomly selected from the corresponding pools. A sample of such a chromosome is shown in Figure 7.9.

Step 3: ***Define a fitness function to evaluate the chromosome performance***

The chromosome evaluation is a crucial part of the GA, because chromosomes are selected for mating based on their fitness. The fitness function must capture what makes a maintenance schedule either good or bad for the user. For our problem we apply a fairly simple function concerned with constraint violations and the net reserve at each interval.

Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6	Unit 7
0 1 1 0	0 0 1 1	0 0 0 1	1 0 0 0	0 1 0 0	0 0 1 0	1 0 0 0

Figure 7.9   A chromosome for the scheduling problem

The evaluation of a chromosome starts with the sum of capacities of the units scheduled for maintenance at each interval. For the chromosome shown in Figure 7.9, we obtain:

Interval 1: $0 \times 20 + 0 \times 15 + 0 \times 35 + 1 \times 40 + 0 \times 15 + 0 \times 15 + 1 \times 10$
$= 50$

Interval 2: $1 \times 20 + 0 \times 15 + 0 \times 35 + 0 \times 40 + 1 \times 15 + 0 \times 15 + 0 \times 10$
$= 35$

Interval 3: $1 \times 20 + 1 \times 15 + 0 \times 35 + 0 \times 40 + 0 \times 15 + 1 \times 15 + 0 \times 10$
$= 50$

Interval 4: $0 \times 20 + 1 \times 15 + 1 \times 35 + 0 \times 40 + 0 \times 15 + 0 \times 15 + 0 \times 10$
$= 50$

Then these values are subtracted from the total installed capacity of the power system (in our case, 150 MW):

Interval 1: $150 - 50 = 100$
Interval 2: $150 - 35 = 115$
Interval 3: $150 - 50 = 100$
Interval 4: $150 - 50 = 100$

And finally, by subtracting the maximum loads expected at each interval, we obtain the respective net reserves:

Interval 1: $100 - 80 = 20$
Interval 2: $115 - 90 = 25$
Interval 3: $100 - 65 = 35$
Interval 4: $100 - 70 = 30$

Since all the results are positive, this particular chromosome does not violate any constraint, and thus represents a legal schedule. The chromosome's fitness is determined as the lowest of the net reserves; in our case it is 20.

If, however, the net reserve at any interval is negative, the schedule is illegal, and the fitness function returns zero.

At the beginning of a run, a randomly built initial population might consist of all illegal schedules. In this case, chromosome fitness values remain unchanged, and selection takes place in accordance with the actual fitness values.

Step 4:   *Construct the genetic operators*

Constructing genetic operators is challenging and we must experiment to make crossover and mutation work correctly. The chromosome has to be broken up in a way that is legal for our problem. Since we have already changed the chromosome syntax for this, we can use the GA operators in their classical forms. Each gene in a chromosome is represented by a 4-bit indivisible string, which consists of a possible maintenance schedule for a particular unit. Thus, any random mutation

Parent 1

| 0 1 1 0 | 0 0 1 1 | 0 0 0 1 | 1 0 0 0 | 0 1 0 0 | 0 0 1 0 | 1 0 0 0 |

Parent 2

| 1 1 0 0 | 0 1 1 0 | 0 1 0 0 | 0 0 0 1 | 0 0 1 0 | 1 0 0 0 | 0 1 0 0 |

Child 1

| 0 1 1 0 | 0 0 1 1 | 0 0 0 1 | 1 0 0 0 | 0 0 1 0 | 1 0 0 0 | 0 1 0 0 |

Child 2

| 1 1 0 0 | 0 1 1 0 | 0 1 0 0 | 0 0 0 1 | 0 1 0 0 | 0 0 1 0 | 1 0 0 0 |

(a)

×

| 1 1 0 0 | 0 1 1 0 | 0 1 0 0 | 0 0 0 1 | 0 1 0 0 | 0 0 1 0 | 1 0 0 0 |

| 1 1 0 0 | 0 1 1 0 | 0 0 0 1 | 0 0 0 1 | 0 1 0 0 | 0 0 1 0 | 1 0 0 0 |
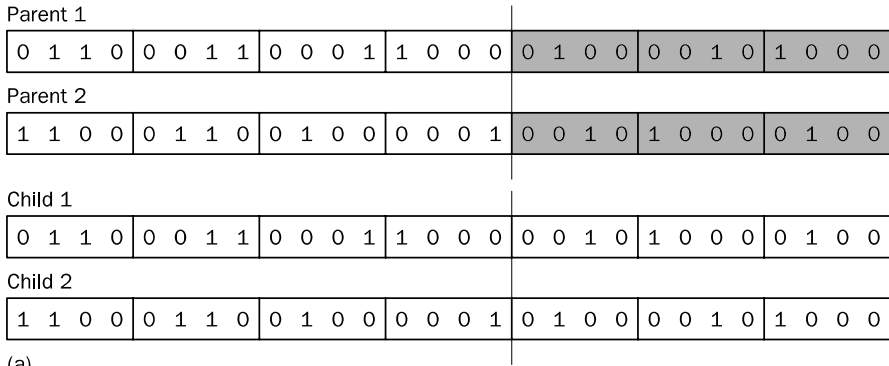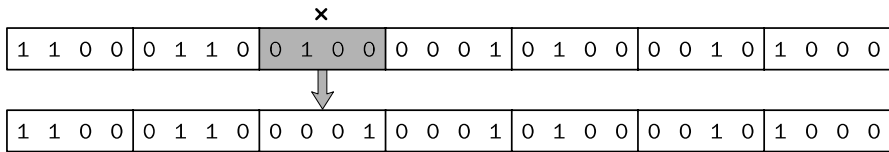
(b)

Figure 7.10   Genetic operators for the scheduling problem: (a) the crossover operator; (b) the mutation operator

of a gene or recombination of several genes from two parent chromosomes may result only in changes of the maintenance schedules for individual units, but cannot create 'unnatural' chromosomes.

Figure 7.10(a) shows an example of the crossover application during a run of the GA. The children are made by cutting the parents at the randomly selected point denoted by the vertical line and exchanging parental genes after the cut. Figure 7.10(b) demonstrates an example of mutation. The mutation operator randomly selects a 4-bit gene in a chromosome and replaces it by a gene randomly selected from the corresponding pool. In the example shown in Figure 7.10(b), the chromosome is mutated in its third gene, which is replaced by the gene $\boxed{0\,0\,0\,1}$ chosen from the pool of genes for the Unit 3.

Step 5:   *Run the GA and tune its parameters*

It is time to run the GA. First, we must choose the population size and the number of generations to be run. Common sense suggests that a larger population can achieve better solutions than a smaller one, but will work more slowly. In fact, however, the most effective population size depends on the problem being solved, particularly on the problem coding scheme (Goldberg, 1989). The GA can run only a finite number of generations to obtain a solution. Perhaps we could choose a very large population and run it only once, or we could choose a smaller population and run it several times. In any case, only experimentation can give us the answer.

(a)

(b)
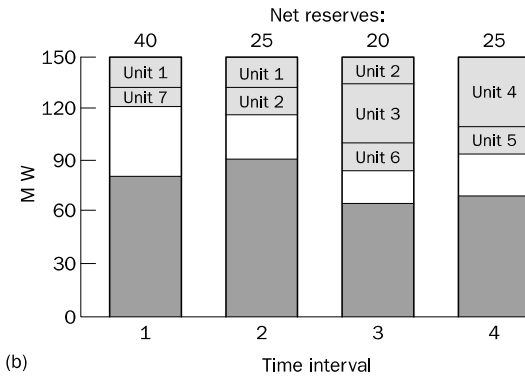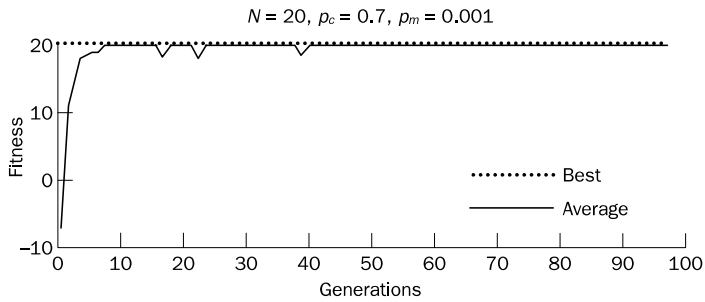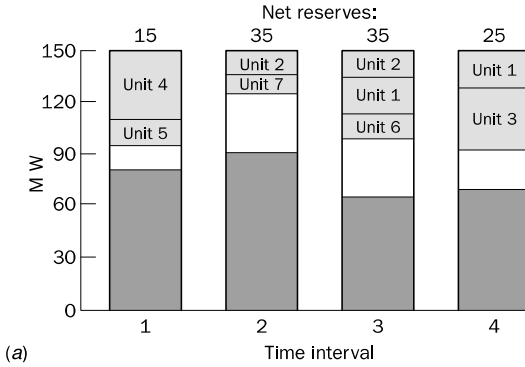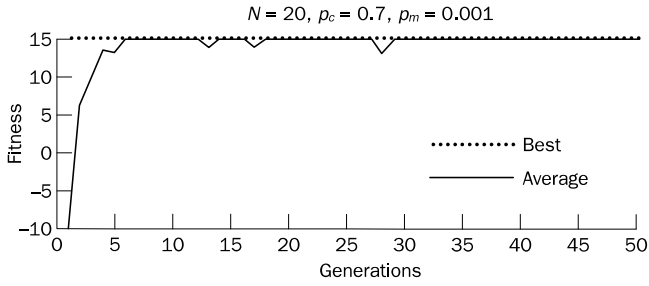
Figure 7.11 Performance graphs and the best maintenance schedules created in a population of 20 chromosomes: (a) 50 generations; (b) 100 generations

(a)

(b)
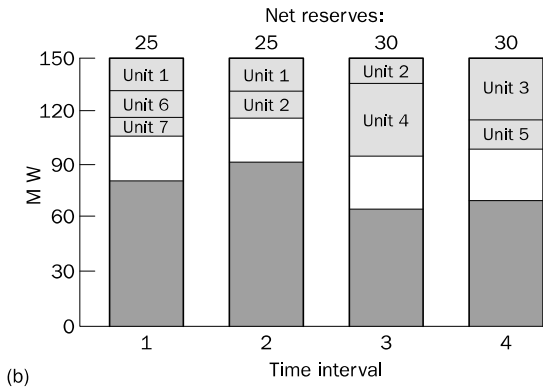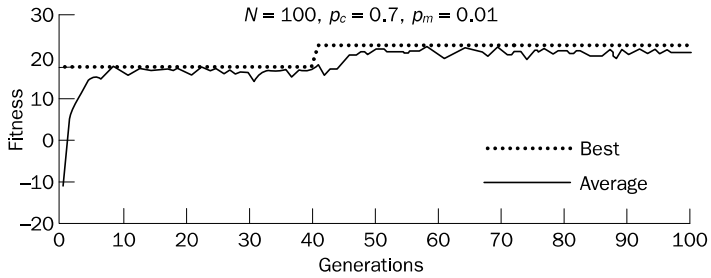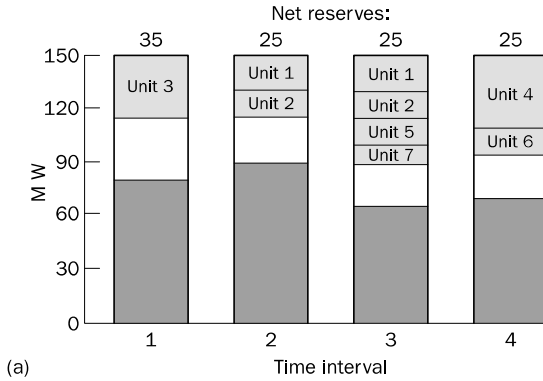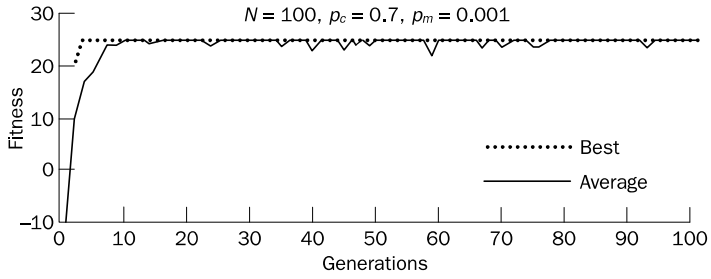
Figure 7.12   Performance graphs and the best maintenance schedules created in a population of 100 chromosomes: (a) mutation rate is 0.001; (b) mutation rate is 0.01

Figure 7.11(a) presents performance graphs and the best schedule created by 50 generations of 20 chromosomes. As you can see, the minimum of the net reserves for the best schedule is 15 MW. Let us increase the number of generations to 100 and compare the best schedules. Figure 7.11(b) presents the results. The best schedule now provides the minimum net reserve of 20 MW. However, in both cases, the best individuals appeared in the initial generation, and the increasing number of generations did not affect the final solution. It indicates that we should try increasing the population size.

Figure 7.12(a) shows fitness function values across 100 generations, and the best schedule so far. The minimum net reserve has increased to 25 MW. To make sure of the quality of the best-so-far schedule, we must compare results obtained under different rates of mutation. Thus, let us increase the mutation rate to 0.01 and rerun the GA once more. Figure 7.12(b) presents the results. The minimum net reserve is still 25 MW. Now we can confidently argue that the optimum solution has been found.

7.6 Evolution strategies

Another approach to simulating natural evolution was proposed in Germany in the early 1960s. Unlike genetic algorithms, this approach – called an **evolution strategy** – was designed to solve technical optimisation problems.

In 1963 two students of the Technical University of Berlin, Ingo Rechenberg and Hans-Paul Schwefel, were working on the search for the optimal shapes of bodies in a flow. In their work, they used the wind tunnel of the Institute of Flow Engineering. Because it was then a matter of laborious intuitive experimentation, they decided to try random changes in the parameters defining the shape following the example of natural mutation. As a result, the evolution strategy was born (Rechenberg, 1965; Schwefel, 1981).

Evolution strategies were developed as an alternative to the engineer's intuition. Until recently, evolution strategies were used in technical optimisation problems when no analytical objective function was available, and no conventional optimisation method existed, thus engineers had to rely only on their intuition.

Unlike GAs, evolution strategies use only a mutation operator.

How do we implement an evolution strategy?

In its simplest form, termed as a $(1 + 1)$-evolution strategy, one parent generates one offspring per generation by applying **normally distributed** mutation. The $(1 + 1)$-evolution strategy can be implemented as follows:

Step 1: Choose the number of parameters N to represent the problem, and then determine a feasible range for each parameter:

$$\{x_{1min}, x_{1max}\}, \{x_{2min}, x_{2max}\}, \ldots, \{x_{Nmin}, x_{Nmax}\},$$

Define a standard deviation for each parameter and the function to be optimised.

Step 2: Randomly select an initial value for each parameter from the respective feasible range. The set of these parameters will constitute the initial population of parent parameters:

$$x_1, x_2, \ldots, x_N$$

Step 3: Calculate the solution associated with the parent parameters:

$$X = f(x_1, x_2, \ldots, x_N)$$

Step 4: Create a new (offspring) parameter by adding a normally distributed random variable a with mean zero and pre-selected deviation δ to each parent parameter:

$$x_i' = x_i + a(0, \delta), \qquad i = 1, 2, \ldots, N \tag{7.7}$$

Normally distributed mutations with mean zero reflect the natural process of evolution where smaller changes occur more frequently than larger ones.

Step 5: Calculate the solution associated with the offspring parameters:

$$X' = f(x_1', x_2', \ldots, x_N')$$

Step 6: Compare the solution associated with the offspring parameters with the one associated with the parent parameters. If the solution for the offspring is better than that for the parents, replace the parent population with the offspring population. Otherwise, keep the parent parameters.

Step 7: Go to Step 4, and repeat the process until a satisfactory solution is reached, or a specified number of generations is considered.

The $(1 + 1)$-evolution strategy can be represented as a block-diagram shown in Figure 7.13.

Why do we vary all the parameters simultaneously when generating a new solution?

An evolution strategy here reflects the nature of a chromosome. In fact, a single gene may simultaneously affect several characteristics of the living organism. On the other hand, a single characteristic of an individual may be determined by the simultaneous interactions of several genes. The natural selection acts on a collection of genes, not on a single gene in isolation.

Evolution strategies can solve a wide range of constrained and unconstrained non-linear optimisation problems and produce better results than many conventional, highly complex, non-linear optimisation techniques (Schwefel, 1995). Experiments also suggest that the simplest version of evolution strategies that uses a single parent – single offspring search works best.

```
                        ┌─────────────┐
                        │    Start    │
                        └─────────────┘
                               │
                               ▼
          ┌────────────────────────────────────────┐
          │      Specify a number of generations    │
          └────────────────────────────────────────┘
                               │
                               ▼
          ┌────────────────────────────────────────┐
          │  Generate a population of parent        │
          │  parameters of size N:                  │
          │         x₁,x₂, ..., xₙ                  │
          └────────────────────────────────────────┘
                               │
                               ▼
          ┌────────────────────────────────────────┐
          │  Calculate the solution associated with │
          │  the parents:                           │
          │         X = f (x₁, x₂, ..., xₙ)         │
          └────────────────────────────────────────┘
```

Start

Specify a number of generations

Generate a population of parent parameters of size N:
$$x_1, x_2, \ldots, x_N$$

Calculate the solution associated with the parents:
$$X = f(x_1, x_2, \ldots, x_N)$$

Create a population of offspring parameters:
$$x'_1 = x_1 + a(0, \delta);$$
$$x'_2 = x_2 + a(0, \delta);$$
$$\vdots$$
$$x'_N = x_N + a(0, \delta)$$

Calculate the solution associated with the osffspring:
$$X' = f(x'_1, x'_2, \ldots, x'_N)$$

Is X' better than X?   *No*

Yes

Replace the parent population with the offspring population

Has the specified number of generations been considered?   *No*

Yes

Stop

Figure 7.13   Block-diagram of the $(1+1)$-evolution strategy

What are the differences between genetic algorithms and evolution strategies?

The principal difference between a GA and an evolution strategy is that the former uses both crossover and mutation whereas the latter uses only mutation. In addition, when we use an evolution strategy we do not need to represent the problem in a coded form.

Which method works best?

An evolution strategy uses a purely numerical optimisation procedure, similar to a focused Monte Carlo search. GAs are capable of more general applications, but the hardest part of applying a GA is coding the problem. In general, to answer the question as to which method works best, we have to experiment to find out. It is application-dependent.

7.7   Genetic programming

One of the central problems in computer science is how to make computers solve problems without being explicitly programmed to do so. Genetic programming offers a solution through the evolution of computer programs by methods of natural selection. In fact, genetic programming is an extension of the conventional genetic algorithm, but the goal of genetic programming is not just to evolve a bit-string representation of some problem but the computer code that solves the problem. In other words, genetic programming creates computer programs as the solution, while GAs create a string of binary numbers that represent the solution.

Genetic programming is a recent development in the area of evolutionary computation. It was greatly stimulated in the 1990s by John Koza (Koza, 1992, 1994).

How does genetic programming work?

According to Koza, genetic programming searches the space of possible computer programs for a program that is highly fit for solving the problem at hand (Koza, 1992).

Any computer program is a sequence of operations (functions) applied to values (arguments), but different programming languages may include different types of statements and operations, and have different syntactic restrictions. Since genetic programming manipulates programs by applying genetic operators, a programming language should permit a computer program to be manipulated as data and the newly created data to be executed as a program. For these reasons, LISP was chosen as the main language for genetic programming (Koza, 1992).

What is LISP?

LISP, or **List Processor**, is one of the oldest high-level programming languages (FORTRAN is just two years older than LISP). LISP, which was written by John McCarthy in the late 1950s, has become one of the standard languages for artificial intelligence.

LISP has a highly symbol-oriented structure. Its basic data structures are **atoms** and **lists**. An atom is the smallest indivisible element of the LISP syntax. The number *21*, the symbol *X* and the string *'This is a string'* are examples of LISP atoms. A list is an object composed of atoms and/or other lists. LISP lists are

written as an ordered collection of items inside a pair of parentheses. For example, the list

(−(∗ A B) C)

calls for the application of the subtraction function (−) to two arguments, namely the list (∗ A B) and the atom C. First, LISP applies the multiplication function (∗) to the atoms A and B. Once the list (∗ A B) is evaluated, LISP applies the subtraction function (−) to the two arguments, and thus evaluates the entire list (−(∗ A B) C).

Both atoms and lists are called symbolic expressions or **S-expressions**. In LISP, all data and all programs are S-expressions. This gives LISP the ability to operate on programs as if they were data. In other words, LISP programs can modify themselves or even write other LISP programs. This remarkable property of LISP makes it very attractive for genetic programming.

Any LISP S-expression can be depicted as a rooted point-labelled tree with ordered branches. Figure 7.14 shows the tree corresponding to the S-expression (−(∗ A B) C). This tree has five points, each of which represents either a function or a terminal. The two internal points of the tree are labelled with functions (−) and (∗). Note that the root of the tree is the function appearing just inside the leftmost opening parenthesis of the S-expression. The three external points of the tree, also called leaves, are labelled with terminals A, B and C. In the graphical representation, the branches are ordered because the order of the arguments in many functions directly affects the results.

How do we apply genetic programming to a problem?

Before applying genetic programming to a problem, we must accomplish **five preparatory steps** (Koza, 1994):

1   Determine the set of terminals.

2   Select the set of primitive functions.

3   Define the fitness function.

4   Decide on the parameters for controlling the run.

5   Choose the method for designating a result of the run.
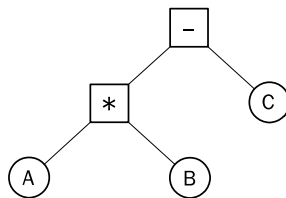
Figure 7.14   Graphical representation of the LISP S-expression (−(∗ A B) C)

Table 7.3 Ten fitness cases for the Pythagorean Theorem

Side a	Side b	Hypotenuse c	Side a	Side b	Hypotenuse c
3	5	5.830952	12	10	15.620499
8	14	16.124515	21	6	21.840330
18	2	18.110770	7	4	8.062258
32	11	33.837849	16	24	28.844410
4	3	5.000000	2	9	9.219545

The Pythagorean Theorem helps us to illustrate these preparatory steps and demonstrate the potential of genetic programming. The theorem says that the hypotenuse, c, of a right triangle with short sides a and b is given by

$$c = \sqrt{a^2 + b^2}.$$

The aim of genetic programming is to discover a program that matches this function. To measure the performance of the as-yet-undiscovered computer program, we will use a number of different **fitness cases**. The fitness cases for the Pythagorean Theorem are represented by the samples of right triangles in Table 7.3. These fitness cases are chosen at random over a range of values of variables a and b.

Step 1: *Determine the set of terminals*
The terminals correspond to the inputs of the computer program to be discovered. Our program takes two inputs, a and b.

Step 2: *Select the set of primitive functions*
The functions can be presented by standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions or domain-specific functions. Our program will use four standard arithmetic operations $+$, $-$, $*$ and $/$, and one mathematical function *sqrt*.

Terminals and primitive functions together constitute the building blocks from which genetic programming constructs a computer program to solve the problem.

Step 3: *Define the fitness function*
A fitness function evaluates how well a particular computer program can solve the problem. The choice of the fitness function depends on the problem, and may vary greatly from one problem to the next. For our problem, the fitness of the computer program can be measured by the error between the actual result produced by the program and the correct result given by the fitness case. Typically, the error is not measured over just one fitness case, but instead calculated as a sum of the absolute errors over a number of fitness cases. The closer this sum is to zero, the better the computer program.

Step 4: ***Decide on the parameters for controlling the run***

For controlling a run, genetic programming uses the same primary parameters as those used for GAs. They include the population size and the maximum number of generations to be run.

Step 5: ***Choose the method for designating a result of the run***

It is common practice in genetic programming to designate the best-so-far generated program as the result of a run.

Once these five steps are complete, a run can be made. The run of genetic programming starts with a random generation of an initial population of computer programs. Each program is composed of functions $+$, $-$, $*$, $/$ and *sqrt*, and terminals *a* and *b*.

In the initial population, all computer programs usually have poor fitness, but some individuals are more fit than others. Just as a fitter chromosome is more likely to be selected for reproduction, so a fitter computer program is more likely to survive by copying itself into the next generation.

Is the crossover operator capable of operating on computer programs?

In genetic programming, the crossover operator operates on two computer programs which are selected on the basis of their fitness. These programs can have different sizes and shapes. The two offspring programs are composed by recombining randomly chosen parts of their parents. For example, consider the following two LISP S-expressions:

$(/ (- (sqrt (+ (* a a) (- a b))) a) (* a b)),$

which is equivalent to

$$\frac{\sqrt{a^2 + (a - b)} - a}{ab},$$

and

$(+ (- (sqrt (- (* b b) a)) b) (sqrt (/ a b))),$

which is equivalent to

$$\left(\sqrt{b^2 - a} - b\right) + \sqrt{\frac{a}{b}}.$$

These two S-expressions can be presented as rooted, point-labelled trees with ordered branches as shown in Figure 7.15(a). Internal points of the trees correspond to functions and external points correspond to terminals.

Any point, internal or external, can be chosen as a crossover point. Suppose that the crossover point for the first parent is the function $(*)$, and the crossover

$(/ (- (sqrt (+(* a a)(- a b))) a) (* a b))$

(a)

$(+ (- (sqrt (- (* b b) a)) b) (sqrt (/ a b)))$

$(/ (- (sqrt (+(* a a)(- a b))) a) (sqrt (- (* b b) a)))$

(b)

$(+ (- (* a b) b) (sqrt (/ a b)))$
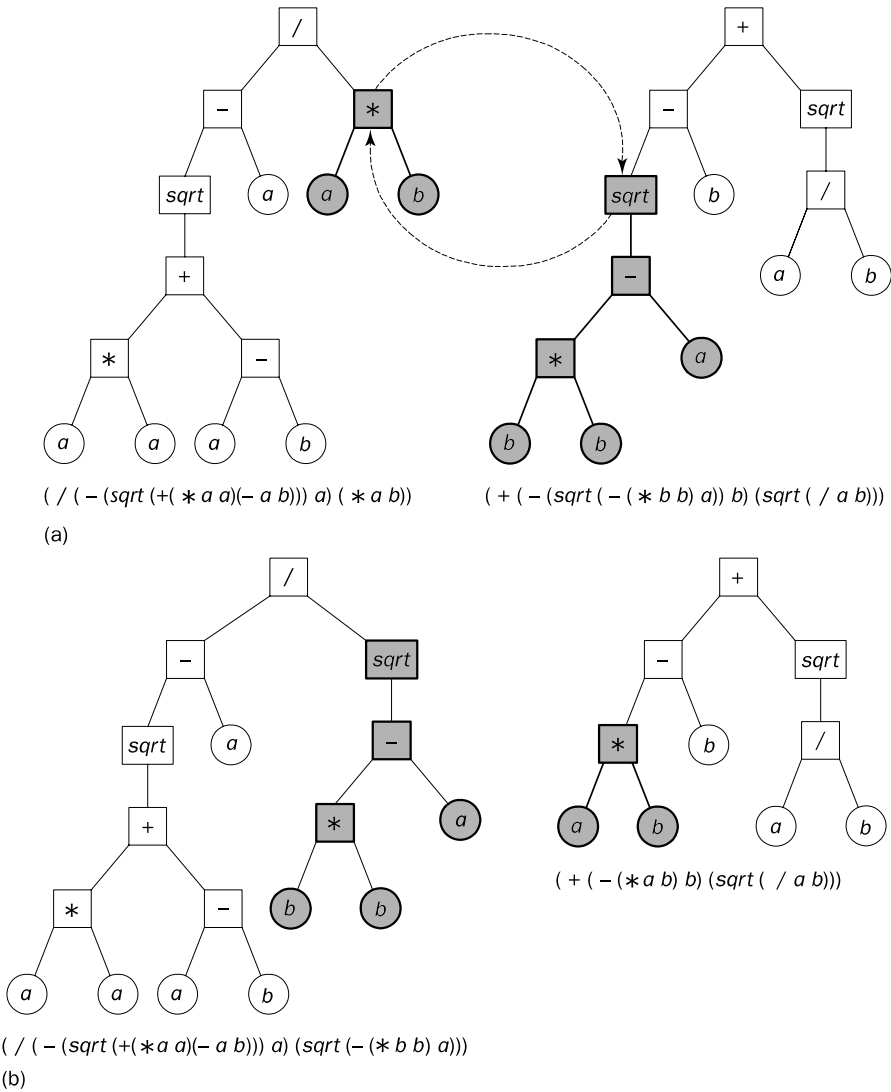
Figure 7.15   Crossover in genetic programming: (a) two parental S-expressions;
(b) two offspring S-expressions

point for the second parent is the function *sqrt*. As a result, we obtain the two
crossover fragments rooted at the chosen crossover points as shown in Figure
7.15(a). The crossover operator creates two offspring by exchanging the cross-
over fragments of two parents. Thus, the first offspring is created by inserting the
crossover fragment of the second parent into the place of the crossover fragment
of the first parent. Similarly, the second offspring is created by inserting the
crossover fragment of the first parent into the place of the crossover fragment of

the second parent. The two offspring resulting from crossover of the two parents are shown in Figure 7.15(b). These offspring are equivalent to

$$\frac{\sqrt{a^2 + (a - b)} - a}{\sqrt{b^2 - a}} \quad \text{and} \quad (ab - b) + \sqrt{\frac{a}{b}}.$$

The crossover operator produces valid offspring computer programs regardless of the choice of crossover points.

Is mutation used in genetic programming?

A mutation operator can randomly change any function or any terminal in the LISP S-expression. Under mutation, a function can only be replaced by a function and a terminal can only be replaced by a terminal. Figure 7.16 explains the basic concept of mutation in genetic programming.

In summary, genetic programming creates computer programs by executing the following steps (Koza, 1994):

Step 1: Assign the maximum number of generations to be run and probabilities for cloning, crossover and mutation. Note that the sum of the probability of cloning, the probability of crossover and the probability of mutation must be equal to one.

Step 2: Generate an initial population of computer programs of size N by combining randomly selected functions and terminals.

Step 3: Execute each computer program in the population and calculate its fitness with an appropriate fitness function. Designate the best-so-far individual as the result of the run.

Step 4: With the assigned probabilities, select a genetic operator to perform cloning, crossover or mutation.

Step 5: If the cloning operator is chosen, select one computer program from the current population of programs and copy it into a new population.

   If the crossover operator is chosen, select a pair of computer programs from the current population, create a pair of offspring programs and place them into the new population.

   If the mutation operator is chosen, select one computer program from the current population, perform mutation and place the mutant into the new population.

   All programs are selected with a probability based on their fitness (i.e., the higher the fitness, the more likely the program is to be selected).

Step 6: Repeat Step 4 until the size of the new population of computer programs becomes equal to the size of the initial population, N.

$(/ (- (sqrt (+(* a a)(- a b))) a) (* a b))$

(a)

$(+ (- (sqrt (- (* b b) a)) b) (sqrt (/ a b)))$

$(/ (+ (sqrt (+(* a a)(- a b))) a)(* a b)$

(b)

$(+ (- (sqrt (- (*b b) a)) a) (sqrt (/ a b)))$
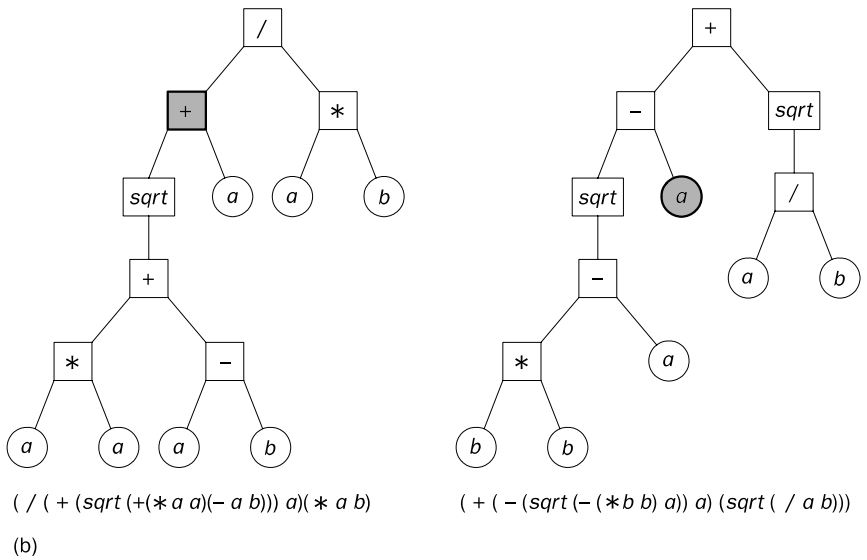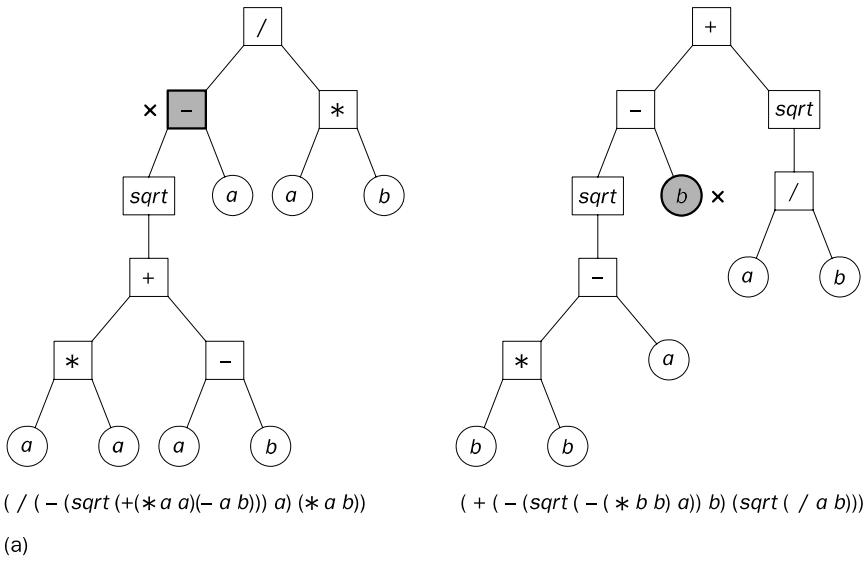
Figure 7.16   Mutation in genetic programming: (a) original S-expressions; (b) mutated S-expressions

Step 7:   Replace the current (parent) population with the new (offspring) population.

Step 8:   Go to Step 3 and repeat the process until the termination criterion is satisfied.

Figure 7.17 is a flowchart representing the above steps of genetic programming.
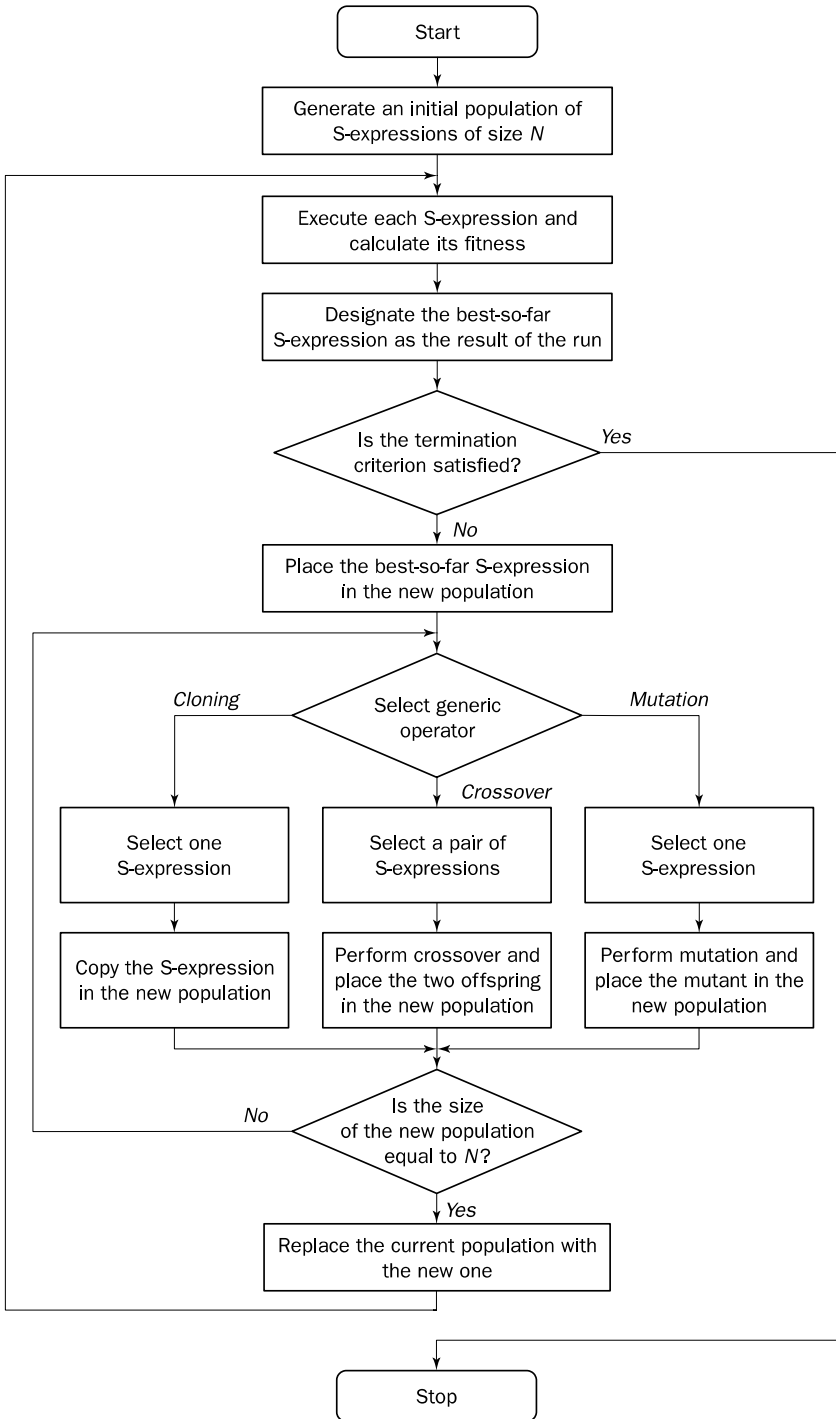
Figure 7.17 Flowchart for genetic programming
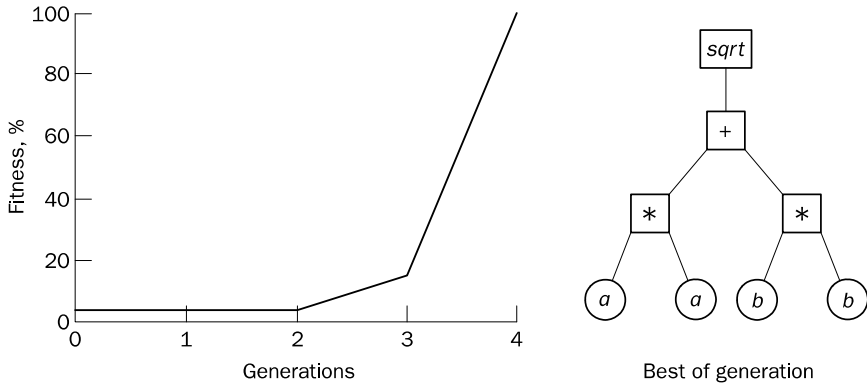
Figure 7.18   Fitness history of the best S-expression

Let us now return to the Pythagorean Theorem. Figure 7.18 shows the fitness history of the best S-expression in a population of 500 computer programs. As you can see, in the randomly generated initial population, even the best S-expression has very poor fitness. But fitness improves very rapidly, and at the fourth generation the correct S-expression is reproduced. This simple example demonstrates that genetic programming offers a general and robust method of evolving computer programs.

In the Pythagorean Theorem example, we used LISP S-expressions but there is no reason to restrict genetic programming only to LISP S-expressions. It can also be implemented in C, C++, Pascal, FORTRAN, Mathematica, Smalltalk and other programming languages, and can be applied more generally.

What are the main advantages of genetic programming compared to genetic algorithms?

Genetic programming applies the same evolutionary approach as a GA does. However, genetic programming is no longer breeding bit strings that represent coded solutions but complete computer programs that solve a particular problem. The fundamental difficulty of GAs lies in the problem representation, that is, in the fixed-length coding. A poor representation limits the power of a GA, and even worse, may lead to a false solution.

A fixed-length coding is rather artificial. As it cannot provide a dynamic variability in length, such a coding often causes considerable redundancy and reduces the efficiency of genetic search. In contrast, genetic programming uses high-level building blocks of variable length. Their size and complexity can change during breeding. Genetic programming works well in a large number of different cases (Koza, 1994) and has many potential applications.

Are there any difficulties?

Despite many successful applications, there is still no proof that genetic programming will scale up to more complex problems that require larger computer programs. And even if it scales up, extensive computer run times may be needed.

7.8 Summary

In this chapter, we presented an overview of evolutionary computation. We considered genetic algorithms, evolution strategies and genetic programming. We introduced the main steps in developing a genetic algorithm, discussed why genetic algorithms work, and illustrated the theory through actual applications of genetic algorithms. Then we presented a basic concept of evolutionary strategies and determined the differences between evolutionary strategies and genetic algorithms. Finally, we considered genetic programming and its application to real problems.

The most important lessons learned in this chapter are:

- The evolutionary approach to artificial intelligence is based on the computational models of natural selection and genetics known as evolutionary computation. Evolutionary computation combines genetic algorithms, evolution strategies and genetic programming.

- All methods of evolutionary computation work as follows: create a population of individuals, evaluate their fitness, generate a new population by applying genetic operators, and repeat this process a number of times.

- Genetic algorithms were invented by John Holland in the early 1970s. Holland's genetic algorithm is a sequence of procedural steps for moving from one generation of artificial 'chromosomes' to another. It uses 'natural' selection and genetics-inspired techniques known as crossover and mutation. Each chromosome consists of a number of 'genes', and each gene is represented by 0 or 1.

- Genetic algorithms use fitness values of individual chromosomes to carry out reproduction. As reproduction takes place, the crossover operator exchanges parts of two single chromosomes, and the mutation operator changes the gene value in some randomly chosen location of the chromosome. After a number of successive reproductions, the less fit chromosomes become extinct, while those best fit gradually come to dominate the population.

- Genetic algorithms work by discovering and recombining schemata – good 'building blocks' of candidate solutions. The genetic algorithm does not need knowledge of the problem domain, but it requires the fitness function to evaluate the fitness of a solution.

- Solving a problem using genetic algorithms involves defining constraints and optimum criteria, encoding the problem solutions as chromosomes, defining a fitness function to evaluate a chromosome's performance, and creating appropriate crossover and mutation operators.

- Genetic algorithms are a very powerful tool. However, coding the problem as a bit string may change the nature of the problem being investigated. There is always a danger that the coded representation represents a problem that is different from the one we want to solve.

- Evolution strategies were developed by Ingo Rechenberg and Hans-Paul Schwefel in the early 1960s as an alternative to the engineer's intuition. Evolution strategies are used in technical optimisation problems when no analytical objective function is available, and no conventional optimisation method exists – only the engineer's intuition.

- An evolution strategy is a purely numerical optimisation procedure that is similar to a focused Monte Carlo search. Unlike genetic algorithms, evolution strategies use only a mutation operator. In addition, the representation of a problem in a coded form is not required.

- Genetic programming is a recent development in the area of evolutionary computation. It was greatly stimulated in the 1990s by John Koza. Genetic programming applies the same evolutionary approach as genetic algorithms. However, genetic programming is no longer breeding bit strings that represent coded solutions but complete computer programs that solve a problem at hand.

- Solving a problem by genetic programming involves determining the set of arguments, selecting the set of functions, defining a fitness function to evaluate the performance of created computer programs, and choosing the method for designating a result of the run.

- Since genetic programming manipulates programs by applying genetic operators, a programming language should permit a computer program to be manipulated as data and the newly created data to be executed as a program. For these reasons, LISP was chosen as the main language for genetic programming.

Questions for review

1 Why are genetic algorithms called genetic? Who was the 'father' of genetic algorithms?

2 What are the main steps of a genetic algorithm? Draw a flowchart that implements these steps. What are termination criteria used in genetic algorithms?

3 What is the roulette wheel selection technique? How does it work? Give an example.

4 How does the crossover operator work? Give an example using fixed-length bit strings. Give another example using LISP S-expressions.

5 What is mutation? Why is it needed? How does the mutation operator work? Give an example using fixed-length bit strings. Give another example using LISP S-expressions.

6 Why do genetic algorithms work? What is a schema? Give an example of a schema and its instances. Explain the relationship between a schema and a chromosome. What is the Schema Theorem?

7 Describe a typical process of the development of a genetic algorithm for solving a real problem. What is the fundamental difficulty of genetic algorithms?

8 What is an evolution strategy? How is it implemented? What are the differences between evolution strategies and genetic algorithms?

9 Draw a block-diagram of the $(1 + 1)$ evolution strategy. Why do we vary all the parameters simultaneously when generating a new solution?

10 What is genetic programming? How does it work? Why has LISP become the main language for genetic programming?

11 What is a LISP S-expression? Give an example and represent it as a rooted point-labelled tree with ordered branches. Show terminals and functions on the tree.

12 What are the main steps in genetic programming? Draw a flowchart that implements these steps. What are advantages of genetic programming?

References

Atmar, W. (1994). Notes on the simulation of evolution, *IEEE Transactions on Neural Networks*, 5(1), 130–148.

Caruana, R.A. and Schaffer, J.D. (1988). Representation and hidden bias: gray vs. binary coding for genetic algorithms, *Proceedings of the Fifth International Conference on Machine Learning*, J. Laird, ed., Morgan Kaufmann, San Mateo, CA.

Davis, L. (1991). *Handbook on Genetic Algorithms*. Van Nostrand Reinhold, New York.

Fogel, L.J., Owens, A.J. and Walsh, M.J. (1966). *Artificial Intelligence Through Simulated Evolution*. Morgan Kaufmann, Los Altos, CA.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, Reading, MA.

Hartl, D.L. and Clark, A.G. (1989). *Principles of Population Genetics*, 2nd edn. Sinauer, Sunderland, MA.

Hoffman, A. (1989). *Arguments on Evolution: A Paleontologist's Perspective*. Oxford University Press, New York.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Keeton, W.T. (1980). *Biological Science*, 3rd edn. W.W. Norton, New York.

Koza, J.R. (1992). *Genetic Programming: On the Programming of the Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Mayr, E. (1988). *Towards a New Philosophy of Biology: Observations of an Evolutionist*. Belknap Press, Cambridge, MA.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolutionary Programs*, 3rd edn. Springer-Verlag, New York.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

Rawlins, G. (1991). *Foundations of Genetic Algorithms*. Morgan Kaufmann, San Francisco, CA.

Rechenberg, I. (1965). *Cybernetic Solution Path of an Experimental Problem*. Ministry of Aviation, Royal Aircraft Establishment, Library Translation No. 1122, August.

Schwefel, H.-P. (1981). *Numerical Optimization of Computer Models*. John Wiley, Chichester.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. John Wiley, New York.

Turing, A.M. (1950). Computing machinery and intelligence, *Mind*, 59, 433–460.

Whitley, L.D. (1993). *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Francisco, CA.

Wright, S. (1932). The roles of mutation, inbreeding, crossbreeding, and selection in evolution, *Proceedings of the 6th International Congress on Genetics*, Ithaca, NY, vol. 1, pp. 356–366.

Bibliography

Arnold, D.V. and Beyer, H.-G. (2002). *Noisy Optimization with Evolution Strategies*. Kluwer Academic Publishers, Boston.

Beyer, H.-G. (2001). *The Theory of Evolution Strategies*. Springer-Verlag, Heidelberg.

Cantu-Paz, E. (2000). *Designing Efficient Parallel Genetic Algorithms*. Kluwer Academic Publishers, Boston.

Christian, J. (2001). *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann, San Francisco, CA.

Coley, D.A. (1999). *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific, Singapore.

Davidor, Y. (1990). *Genetic Algorithms and Robotics*. World Scientific, Singapore.

Gen, M. and Cheng, R. (1997). *Genetic Algorithms and Engineering Design*. John Wiley, New York.

Gen, M. and Cheng, R. (1999). *Genetic Algorithms and Engineering Optimization*. John Wiley, New York.

Goldberg, D.E. (2002). *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Boston.

Haupt, R.L. and Haupt, S.E. (1998). *Practical Genetic Algorithms*. John Wiley, New York.

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J.R., Bennett III, F.H., Andre, D. and Keane, M.A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, San Francisco, CA.

Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J. and Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Boston.

Langdon, W.B. (1998). *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming*! Kluwer Academic Publishers, Amsterdam.

Langdon, W.B. and Poli, R. (2002). *Foundations of Genetic Programming*. Springer-Verlag, Berlin.

Man, K.F., Tang, K.S., Kwong, S. and Halang, W.A. (1997). *Genetic Algorithms for Control and Signal Processing*. Springer-Verlag, London.

Man, K.F., Tang, K.S. and Kwong, S. (1999). *Genetic Algorithms: Concepts and Designs*. Springer-Verlag, London.

O'Neill, M. and Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Boston.

Vose, M.D. (1999). *The Simple Genetic Algorithm: Foundations and Theory*. MIT Press, Cambridge, MA.

Hybrid intelligent systems   **8**

In which we consider the combination of expert systems, fuzzy logic, neural networks and evolutionary computation, and discuss the emergence of hybrid intelligent systems.

8.1 Introduction, or how to combine German mechanics with Italian love

In previous chapters, we considered several intelligent technologies, including probabilistic reasoning, fuzzy logic, neural networks and evolutionary computation. We discussed the strong and weak points of these technologies, and noticed that in many real-world applications we would need not only to acquire knowledge from various sources, but also to combine different intelligent technologies. The need for such a combination has led to the emergence of **hybrid intelligent systems**.

A hybrid intelligent system is one that combines at least two intelligent technologies. For example, combining a neural network with a fuzzy system results in a hybrid neuro-fuzzy system.

The combination of probabilistic reasoning, fuzzy logic, neural networks and evolutionary computation forms the core of **soft computing** (SC), an emerging approach to building hybrid intelligent systems capable of reasoning and learning in an uncertain and imprecise environment.

The potential of soft computing was first realised by Lotfi Zadeh, the 'father' of fuzzy logic. In March 1991, he established the Berkeley Initiative in Soft Computing. This group includes students, professors, employees of private and government organisations, and other individuals interested in soft computing. The rapid growth of the group suggests that the impact of soft computing on science and technology will be increasingly felt in coming years.

What do we mean by 'soft' computing?
While traditional or 'hard' computing uses **crisp values**, or numbers, soft computing deals with **soft values**, or fuzzy sets. Soft computing is capable of operating with uncertain, imprecise and incomplete information in a manner that reflects human thinking. In real life, humans normally use soft data

represented by words rather than numbers. Our sensory organs deal with soft information, our brain makes soft associations and inferences in uncertain and imprecise environments, and we have a remarkable ability to reason and make decisions without using numbers. Humans use words, and soft computing attempts to model our sense of words in decision making.

Can we succeed in solving complex problems using words?

Words are inherently less precise than numbers but precision carries a high cost. We use words when there is a tolerance for imprecision. Likewise, soft computing exploits the tolerance for uncertainty and imprecision to achieve greater tractability and robustness, and lower the cost of solutions (Zadeh, 1996).

We also use words when the available data is not precise enough to use numbers. This is often the case with complex problems, and while 'hard' computing fails to produce any solution, soft computing is still capable of finding good solutions.

What is the difference between soft computing and artificial intelligence?

Conventional artificial intelligence attempts to express human knowledge in symbolic terms. Its corner-stones are its rigid theory for symbol manipulation and its exact reasoning mechanisms, including forward and backward chaining. The most successful product of conventional artificial intelligence is the expert system. But an expert system is good only if explicit knowledge is acquired and represented in the knowledge base. This substantially limits the field of practical applications for such systems.

However, during the last few years, the domain of artificial intelligence has expanded rapidly to include artificial neural networks, genetic algorithms and even fuzzy set theory (Russell and Norvig, 2002). This makes the boundaries between modern artificial intelligence and soft computing vague and elusive. The objective of this chapter, however, is not to argue when one becomes part of the other, but to provide the reader with an understanding of the main principles of building hybrid intelligent systems.

What exactly are we trying to combine in a hybrid system?

Lotfi Zadeh is reputed to have said that a good hybrid would be 'British Police, German Mechanics, French Cuisine, Swiss Banking and Italian Love'. But 'British Cuisine, German Police, French Mechanics, Italian Banking and Swiss Love' would be a bad one. Likewise, a hybrid intelligent system can be good or bad – it depends on which components constitute the hybrid. So our goal is to select the right components for building a good hybrid system.

Each component has its own strengths and weaknesses. Probabilistic reasoning is mainly concerned with uncertainty, fuzzy logic with imprecision, neural networks with learning, and evolutionary computation with optimisation. Table 8.1 presents a comparison of different intelligent technologies. A good hybrid system brings the advantages of these technologies together. Their synergy allows a hybrid system to accommodate common sense, extract knowledge from raw data, use human-like reasoning mechanisms, deal with uncertainty and imprecision, and learn to adapt to a rapidly changing and unknown environment.

Table 8.1   Comparison of expert systems (ES), fuzzy systems (FS), neural networks (NN) and genetic algorithms (GA)

	ES	FS	NN	GA
Knowledge representation	○	●	□	■
Uncertainty tolerance	○	●	●	●
Imprecision tolerance	□	●	●	●
Adaptability	□	■	●	●
Learning ability	□	□	●	●
Explanation ability	●	●	□	■
Knowledge discovery and data mining	□	■	●	○
Maintainability	□	○	●	○

The terms used for grading are: □ bad, ■ rather bad, ○ rather good and ● good

8.2   Neural expert systems

Expert systems and neural networks, as intelligent technologies, share common goals. They both attempt to imitate human intelligence and eventually create an intelligent machine. However, they use very different means to achieve their goals. While expert systems rely on logical inferences and decision trees and focus on modelling human reasoning, neural networks rely on parallel data processing and focus on modelling a human brain. Expert systems treat the brain as a black-box, whereas neural networks look at its structure and functions, particularly at its ability to learn. These fundamental differences are reflected in the knowledge representation and data processing techniques used in expert systems and neural networks.

Knowledge in a rule-based expert system is represented by IF-THEN production rules collected by observing or interviewing human experts. This task, called knowledge acquisition, is difficult and expensive. In addition, once the rules are stored in the knowledge base, they cannot be modified by the expert system itself. Expert systems cannot learn from experience or adapt to new environments. Only a human can manually modify the knowledge base by adding, changing or deleting some rules.

Knowledge in neural networks is stored as synaptic weights between neurons. This knowledge is obtained during the learning phase when a training set of data is presented to the network. The network propagates the input data from layer to layer until the output data is generated. If it is different from the desired output, an error is calculated and propagated backwards through the network. The synaptic weights are modified as the error is propagated. Unlike expert systems, neural networks learn without human intervention.

However, in expert systems, knowledge can be divided into individual rules and the user can see and understand the piece of knowledge applied by the system. In contrast, in neural networks, one cannot select a single synaptic weight as a discrete piece of knowledge. Here knowledge is embedded in the

entire network; it cannot be broken into individual pieces, and any change of a synaptic weight may lead to unpredictable results. A neural network is, in fact, a black-box for its user.

An expert system cannot learn, but can explain how it arrives at a particular solution. A neural network can learn, but acts as a black-box. Thus by combining the advantages of each technology we can create a more powerful and effective expert system. A hybrid system that combines a neural network and a rule-based expert system is called a **neural expert system** (or a **connectionist expert system**). Learning, generalisation, robustness and parallel information processing make neural networks a 'right' component for building a new breed of expert systems.

Figure 8.1 shows the basic structure of a neural expert system. Unlike a rule-based expert system, the knowledge base in the neural expert system is represented by a trained neural network.

A neural expert system can extract IF-THEN rules from the neural network, which enables it to justify and explain its conclusion.

The heart of a neural expert system is the **inference engine**. This controls the information flow in the system and initiates inference over the neural knowledge base. A neural inference engine also ensures approximate reasoning.

What is approximate reasoning?

In a rule-based expert system, the inference engine compares the condition part of each rule with data given in the database. When the IF part of the rule matches the data in the database, the rule is fired and its THEN part is executed.
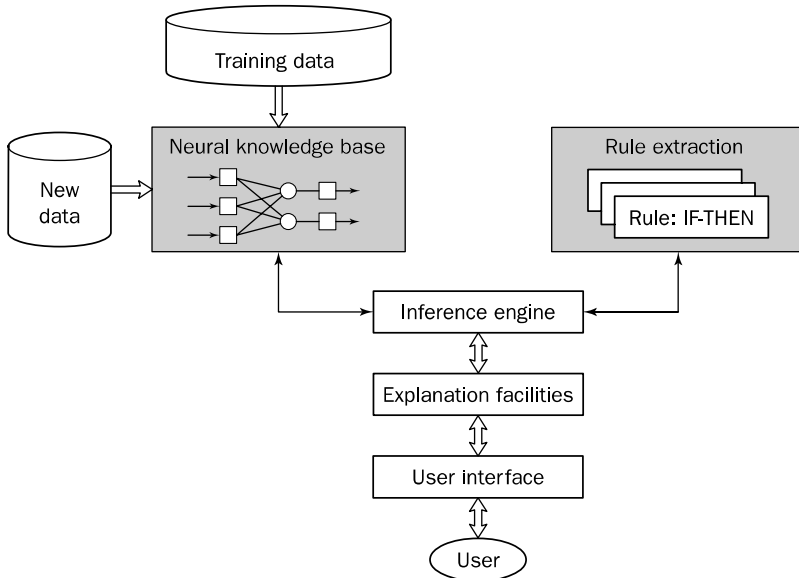
Figure 8.1   Basic structure of a neural expert system

In rule-based expert systems, the **precise** matching is required. As a result, the inference engine cannot cope with noisy or incomplete data.

Neural expert systems use a trained neural network in place of the knowledge base. The neural network is capable of generalisation. In other words, the new input data does not have to precisely match the data that was used in network training. This allows neural expert systems to deal with noisy and incomplete data. This ability is called approximate reasoning.

The **rule extraction** unit examines the neural knowledge base and produces the rules **implicitly** 'buried' in the trained neural network.

The **explanation facilities** explain to the user how the neural expert system arrives at a particular solution when working with the new input data.

The **user interface** provides the means of communication between the user and the neural expert system.

How does a neural expert system extract rules that justify its inference?

Neurons in the network are connected by links, each of which has a numerical weight attached to it. The weights in a trained neural network determine the strength or importance of the associated neuron inputs; this characteristic is used for extracting rules (Gallant, 1993; Nikolopoulos, 1997; Sestito and Dillon, 1991).

Let us consider a simple example to illustrate how a neural expert system works. This example is an object classification problem. The object to be classified belongs to either birds, planes or gliders. A neural network used for this problem is shown in Figure 8.2. It is a three-layer network fully connected between the first and the second layers. All neurons are labelled according to the concepts they represent.

The first layer is the **input layer**. Neurons in the input layer simply transmit external signals to the next layer. The second layer is the **conjunction layer**. The neurons in this layer apply a sign activation function given by

$$Y^{sign} = \begin{cases} +1, & \text{if } X \geqslant 0 \\ -1, & \text{if } X < 0 \end{cases}, \tag{8.1}$$

where X is the net weighted input to the neuron,

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

x_i and w_i are the value of input i and its weight, respectively, and n is the number of neuron inputs.

The third layer is the **output layer**. In our example, each output neuron receives an input from a single conjunction neuron. The weights between the second and the third layers are set to unity.

You might notice that IF-THEN rules are mapped quite naturally into a three-layer neural network where the third (disjunction) layer represents the consequent parts of the rules. Furthermore, the strength of a given rule, or its
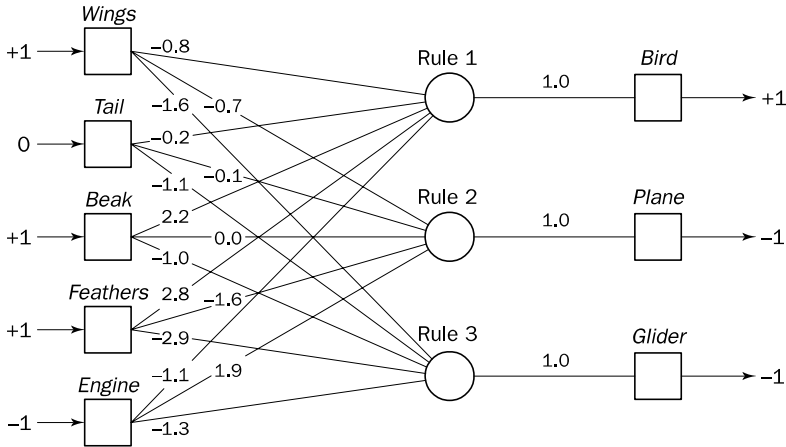
Figure 8.2   The neural knowledge base

certainty factor, can be associated with the weight between respective conjunction and disjunction neurons (Fu, 1993; Kasabov, 1996). We will discuss specific aspects of mapping rules into a neural network later, but now we shall return to our example.

The neural knowledge base was trained with a set of training examples; Figure 8.2 shows the actual numerical weights obtained between the first and the second layers. If we now set each input of the input layer to either $+1$ (true), -1 (false), or 0 (unknown), we can give a semantic interpretation for the activation of any output neuron. For example, if the object has *Wings* $(+1)$, *Beak* $(+1)$ and *Feathers* $(+1)$, but does not have *Engine* (-1), then we can conclude that this object is *Bird* $(+1)$:

$$X_{Rule1} = 1 \times (-0.8) + 0 \times (-0.2) + 1 \times 2.2 + 1 \times 2.8 + (-1) \times (-1.1)$$
$$= 5.3 > 0;$$
$$Y_{Rule1} = Y_{Bird} = +1.$$

We can similarly conclude that this object is not *Plane*,

$$X_{Rule2} = 1 \times (-0.7) + 0 \times (-0.1) + 1 \times 0.0 + 1 \times (-1.6) + (-1) \times 1.9$$
$$= -4.2 < 0;$$
$$Y_{Rule2} = Y_{Plane} = -1.$$

and not *Glider*,

$$X_{Rule3} = 1 \times (-0.6) + 0 \times (-1.1) + 1 \times (-1.0) + 1 \times (-2.9) + (-1) \times (-1.3)$$
$$= -4.2 < 0;$$
$$Y_{Rule3} = Y_{Glider} = -1.$$

Now by attaching a corresponding question to each input neuron,

Neuron: *Wings*   Question: Does the object have wings?
Neuron: *Tail*   Question: Does the object have a tail?
Neuron: *Beak*   Question: Does the object have a beak?
Neuron: *Feathers*   Question: Does the object have feathers?
Neuron: *Engine*   Question: Does the object have an engine?

we can enable the system to prompt the user for initial values of the input variables. The system's goal is to obtain the most important information first and to draw a conclusion as quickly as possible.

How does the system know what the most important information is, and whether it has enough information to draw a conclusion?

The importance of a particular neuron input is determined by the absolute value of the weight attached to this input. For example, for neuron *Rule* 1, the input *Feathers* has a much greater importance than the input *Wings*. Thus, we might establish the following dialogue with the system:

PURSUING:
> Bird
ENTER INITIAL VALUE FOR THE INPUT FEATHERS:
> +1

Our task now is to see whether the acquired information is sufficient to draw a conclusion. The following heuristic can be applied here (Gallant, 1993):

An inference can be made if the known net weighted input to a neuron is greater than the sum of the absolute values of the weights of the unknown inputs.

This heuristic can be expressed mathematically as follows:

$$\sum_{i=1}^{n} x_i w_i > \sum_{j=1}^{n} |w_j| \tag{8.2}$$

where $i \in \text{KNOWN}$, $j \notin \text{KNOWN}$ and n is the number of neuron inputs.

In our example, when the input *Feathers* becomes known, we obtain

KNOWN $= 1 \times 2.8 = 2.8$
UNKNOWN $= |-0.8| + |-0.2| + |2.2| + |-1.1| = 4.3$
KNOWN < UNKNOWN

Thus, the inference for neuron *Rule* 1 cannot be made yet, and the user is asked to provide a value for the next most important input, input *Beak*:

ENTER INITIAL VALUE FOR THE INPUT BEAK:
> +1

Now we have

KNOWN $= 1 \times 2.8 + 1 \times 2.2 = 5.0$
UNKNOWN $= |-0.8| + |-0.2| + |-1.1| = 2.1$
KNOWN > UNKNOWN

And thus, according to the heuristic (8.2), the following inference can be made:

CONCLUDE: BIRD IS TRUE

While KNOWN gives the acquired net weighted input to neuron *Rule* 1, UNKNOWN indicates how this net input might change based upon the worst possible combination of values of the unknown inputs. In our example, the net weighted input cannot change more than ±2.1. Therefore, the output of neuron *Rule* 1 will be greater than 0 regardless of the values of the known inputs, and we can make the inference that *Bird* must be true.

Now it is time to examine how a single rule can be extracted to justify an inference. We will use a simple algorithm concerned only with neurons directly connected to the neuron in question (Gallant, 1988). Let us again consider the example shown in Figure 8.2 and justify the inference that *Bird* is true. Because all neurons in the first layer are directly connected to neuron *Rule* 1, we might expect that the rule to be extracted may involve all five neurons – *Wings*, *Tail*, *Beak*, *Feathers* and *Engine*.

First, we determine all contributing inputs and the size of each contribution (Gallant, 1993). An input *i* is considered to be contributing if it does not move the net weighted input in the opposite direction. The size of this contribution is determined by the absolute value of the weight $|w_i|$ of the contributing input *i*.

Now we arrange all contributing inputs according to their sizes in a descending order. In our example, the list of inputs contributing to the inference *Bird is true* looks as follows:

Input: *Feathers*   Size: 2.8
Input: *Beak*   Size: 2.2
Input: *Engine*   Size: 1.1
Input: *Tail*   Size: 0.2

This list enables us to create a rule in which the condition part is represented by the contributing input with the largest contribution:

IF   *Feathers* is true
THEN   *Bird* is true

The next step is to verify this rule. In other words, we need to make sure that the rule passes the validity test. It can be done by applying the heuristic (8.2):

KNOWN $= 1 \times 2.8 = 2.8$
UNKNOWN $= |-0.8| + |-0.2| + |2.2| + |-1.1| = 4.3$
KNOWN < UNKNOWN

The rule is not valid yet, and thus we need to add the 'second best' contributing input as a clause in the condition part of our rule:

IF   *Feathers* is true
AND   *Beak* is true
THEN   *Bird* is true

Now we have:

KNOWN $= 1 \times 2.8 + 1 \times 2.2 = 5.0$
UNKNOWN $= |-0.8| + |-0.2| + |-1.1| = 2.1$
KNOWN $>$ UNKNOWN

This rule has passed the validity test. It is also a maximally general rule, that is a removal of any condition clause results in an invalid rule.

Similarly, we can obtain rules to justify the inferences that *Plane* is false, and *Glider* is false:

IF   *Engine* is false
AND   *Feathers* is true
THEN   *Plane* is false

IF   *Feathers* is true
AND   *Wings* is true
THEN   *Glider* is false

This example also illustrates that the neural expert system can make useful deductions even when the data is incomplete (for instance, *Tail* is unknown in our example).

In our example, we assume that the neural expert system has a properly trained neural knowledge base. In the real world, however, the training data is not always adequate. We also assume that we do not have any prior knowledge about the problem domain. In fact, we might have some knowledge, although not often perfect. Can we determine an initial structure of the neural knowledge base by using domain knowledge, train it with a given set of training data, and then interpret the trained neural network as a set of IF-THEN rules?

As we mentioned before, a set of IF-THEN rules that represent domain knowledge can be mapped into a multi-layer neural network. Figure 8.3 illustrates a set of rules mapped into a five-layer neural network. The weights between conjunction and disjunction layers indicate the strengths of the rules, and thus can be regarded as certainty factors of the associated rules.

As soon as we have established the initial structure of the neural knowledge base, we can train the network according to a given set of training data. This can be done by using an appropriate training algorithm such as back-propagation. When the training phase is completed, we can examine the neural network knowledge base, extract and, if necessary, refine the set of initial

Rule 1:
IF *a*1 AND *a*3 THEN *b*1 (0.8)

Rule 5:
IF *a*5 THEN *b*3 (0.6)

Rule 2:
IF *a*1 AND *a*4 THEN *b*1 (0.2)

Rule 6:
IF *a*1 AND *b*3 THEN *c*1 (0.7)

Rule 3:
IF *a*2 AND *a*5 THEN *b*2 (–0.1)

Rule 7:
IF *b*2 THEN *c*1 (0.1)

Rule 4:
IF *a*3 AND *a*4 THEN *b*3 (0.9)

Rule 8:
IF *b*2 AND *b*3 THEN *c*2 (0.9)
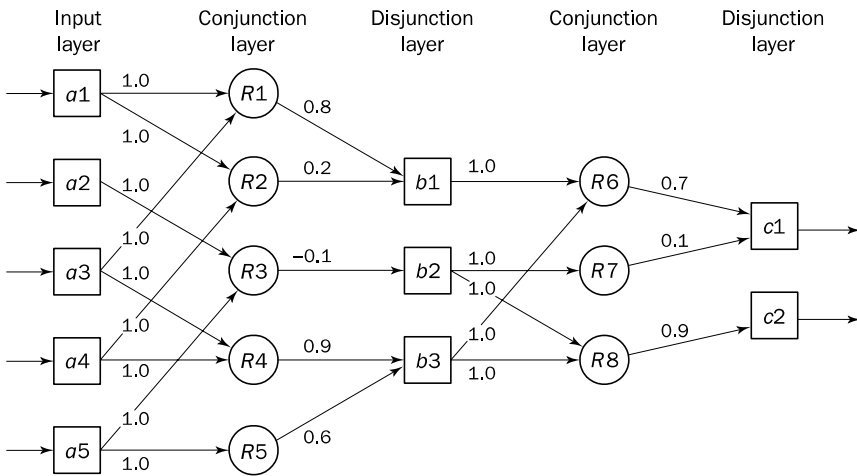
Figure 8.3   An example of a multi-layer knowledge base

IF-THEN rules. Thus, neural expert systems can use domain knowledge represented as IF-THEN rules as well as a set of numerical data. In fact, neural expert systems provide a bi-directional link between neural networks and rule-based systems.

Unfortunately, neural expert systems still suffer from the limitations of Boolean logic, and any attempt to represent continuous input variables may lead to an infinite increase in the number of rules. This might significantly limit the area of application for neural expert systems. The natural way of overcoming this limitation is to use fuzzy logic.

8.3   Neuro-fuzzy systems

Fuzzy logic and neural networks are natural complementary tools in building intelligent systems. While neural networks are low-level computational

structures that perform well when dealing with raw data, fuzzy logic deals with reasoning on a higher level, using linguistic information acquired from domain experts. However, fuzzy systems lack the ability to learn and cannot adjust themselves to a new environment. On the other hand, although neural networks can learn, they are opaque to the user. The merger of a neural network with a fuzzy system into one integrated system therefore offers a promising approach to building intelligent systems. Integrated neuro-fuzzy systems can combine the parallel computation and learning abilities of neural networks with the human-like knowledge representation and explanation abilities of fuzzy systems. As a result, neural networks become more transparent, while fuzzy systems become capable of learning.

A neuro-fuzzy system is, in fact, a neural network that is functionally equivalent to a fuzzy inference model. It can be trained to develop IF-THEN fuzzy rules and determine membership functions for input and output variables of the system. Expert knowledge can be easily incorporated into the structure of the neuro-fuzzy system. At the same time, the connectionist structure avoids fuzzy inference, which entails a substantial computational burden.

How does a neuro-fuzzy system look?

The structure of a neuro-fuzzy system is similar to a multi-layer neural network. In general, a neuro-fuzzy system has input and output layers, and three hidden layers that represent membership functions and fuzzy rules.

Figure 8.4 shows a Mamdani fuzzy inference model, and Figure 8.5 a neuro-fuzzy system that corresponds to this model. For simplicity, we assume that the fuzzy system has two inputs – $x1$ and $x2$ – and one output – y. Input $x1$ is represented by fuzzy sets $A1$, $A2$ and $A3$; input $x2$ by fuzzy sets $B1$, $B2$ and $B3$; and output y by fuzzy sets $C1$ and $C2$.

Each layer in the neuro-fuzzy system is associated with a particular step in the fuzzy inference process.

Layer 1 is the **input layer**. Each neuron in this layer transmits external crisp signals directly to the next layer. That is,

$$y_i^{(1)} = x_i^{(1)},\tag{8.3}$$

where $x_i^{(1)}$ is the input and $y_i^{(1)}$ is the output of input neuron i in Layer 1.

Layer 2 is the **input membership** or **fuzzification layer**. Neurons in this layer represent fuzzy sets used in the antecedents of fuzzy rules. A fuzzification neuron receives a crisp input and determines the degree to which this input belongs to the neuron's fuzzy set, as follows.

The activation function of a membership neuron is set to the function that specifies the neuron's fuzzy set. In the example presented in Figure 8.4, we use triangular sets. Therefore, the activation functions for the neurons in Layer 2 are set to the triangular membership functions (although fuzzification neurons may have any of the membership functions normally used in fuzzy systems). A
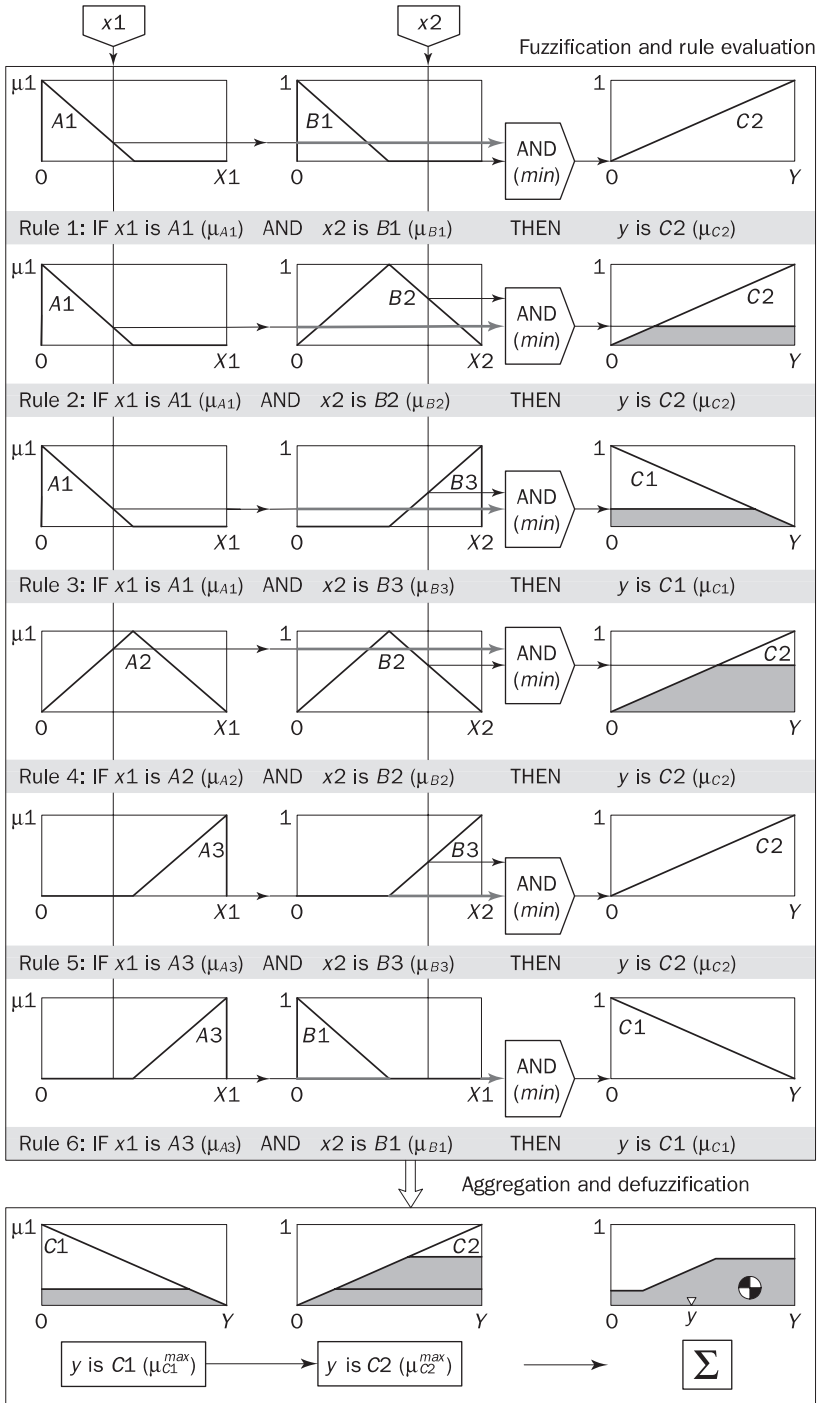
Figure 8.4 Mamdani fuzzy inference system
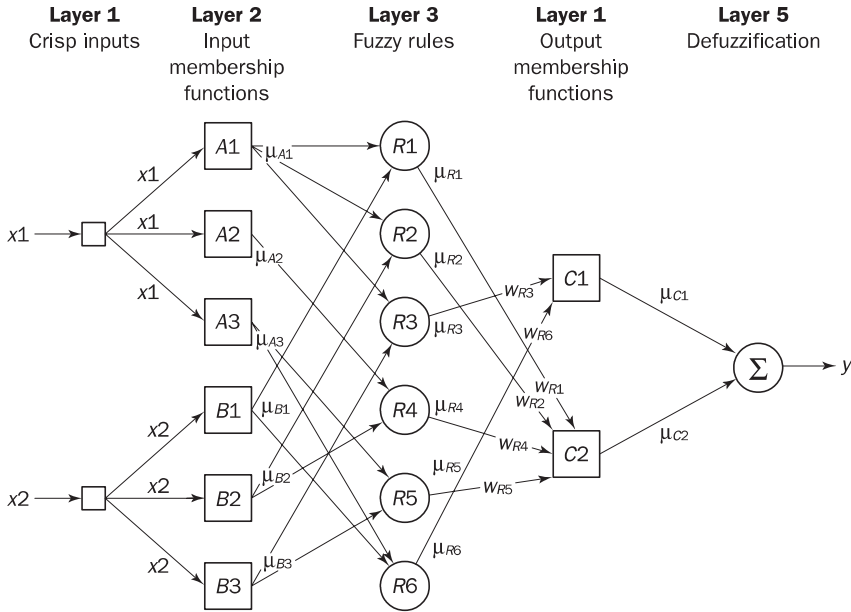
Figure 8.5   Neuro-fuzzy equivalent system

triangular membership function can be specified by two parameters $\{a, b\}$ as follows:

$$
y_i^{(2)} = \begin{cases} 0, & \text{if } x_i^{(2)} \leqslant a - \dfrac{b}{2} \\[2mm] 1 - \dfrac{2|x_i^{(2)} - a|}{b}, & \text{if } a - \dfrac{b}{2} < x_i^{(2)} < a + \dfrac{b}{2} \\[2mm] 0, & \text{if } x_i^{(2)} \geqslant a + \dfrac{b}{2} \end{cases} \tag{8.4}
$$

where a and b are parameters that control the centre and the width of the triangle, respectively, $x_i^{(2)}$ is the input and $y_i^{(2)}$ is the output of fuzzification neuron i in Layer 2.

Figure 8.6 illustrates a triangular function and the effect caused by the variation of parameters a and b. As we can see, the output of a fuzzification neuron depends not only on its input, but also on the centre, a, and the width, b, of the triangular activation function. The neuron input may remain constant, but the output will vary with the change of parameters a and b. In other words, parameters a and b of the fuzzification neurons can play the same role in a neuro-fuzzy system as synaptic weights in a neural network.

Layer 3 is the **fuzzy rule layer**. Each neuron in this layer corresponds to a single fuzzy rule. A fuzzy rule neuron receives inputs from the fuzzification neurons that represent fuzzy sets in the rule antecedents. For instance, neuron $R1$, which corresponds to Rule 1, receives inputs from neurons $A1$ and $B1$.
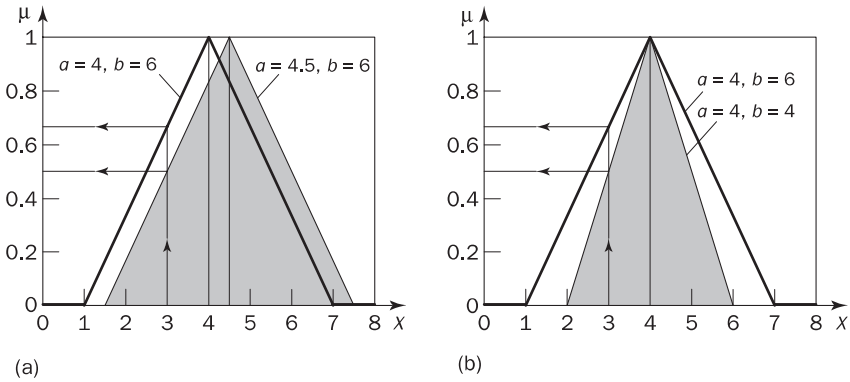
Figure 8.6 Triangular activation functions of the fuzzification neurons: (a) effect of parameter *a*; (b) effect of parameter *b*

In fuzzy systems, if a given rule has multiple antecedents, a fuzzy operator is used to obtain a single number that represents the result of the antecedent evaluation. The conjunction of the rule antecedents is evaluated by the fuzzy operation **intersection**. The same fuzzy operation can be used to combine multiple inputs to a fuzzy rule neuron. In a neuro-fuzzy system, intersection can be implemented by the product operator. Thus, the output of neuron *i* in Layer 3 is obtained as:

$$y_i^{(3)} = x_{1i}^{(3)} \times x_{2i}^{(3)} \times \ldots \times x_{ki}^{(3)}, \tag{8.5}$$

where $x_{1i}^{(3)}, x_{2i}^{(3)}, \ldots, x_{ki}^{(3)}$ are the inputs and $y_i^{(3)}$ is the output of fuzzy rule neuron *i* in Layer 3. For example,

$$y_{R1}^{(3)} = \mu_{A1} \times \mu_{B1} = \mu_{R1}$$

The value of μ_{R1} represents the firing strength of fuzzy rule neuron *R*1.

The weights between Layer 3 and Layer 4 represent the **normalised degrees of confidence** (known as certainty factors) of the corresponding fuzzy rules. These weights are adjusted during training of a neuro-fuzzy system.

What is the normalised degree of confidence of a fuzzy rule?

Different rules represented in a neuro-fuzzy system may be associated with different degrees of confidence. In Figure 8.4, an expert may attach the degree of confidence to each fuzzy IF-THEN rule by setting the corresponding weights within the range of $[0, 1]$. During training, however, these weights can change. To keep them within the specified range, the weights are normalised by dividing their respective values by the highest weight magnitude obtained at each iteration.

Layer 4 is the **output membership layer**. Neurons in this layer represent fuzzy sets used in the consequent of fuzzy rules. An output membership neuron receives inputs from the corresponding fuzzy rule neurons and combines them

by using the fuzzy operation **union**. This operation can be implemented by the probabilistic OR (also known as the algebraic sum). That is,

$$y_i^{(4)} = x_{1i}^{(4)} \oplus x_{2i}^{(4)} \oplus \ldots \oplus x_{li}^{(4)},$$ (8.6)

where $x_{1i}^{(4)}, x_{2i}^{(4)}, \ldots, x_{li}^{(4)}$ are the inputs, and $y_i^{(4)}$ is the output of output membership neuron i in Layer 4. For example,

$$y_{C1}^{(4)} = \mu_{R3} \oplus \mu_{R6} = \mu_{C1}$$

The value of μ_{C1} represents the integrated firing strength of fuzzy rule neurons $R3$ and $R6$. In fact, firing strengths of neurons in the output membership layer are combined in the same way as truth values of the fuzzy rules in Figure 8.4.

In the Mamdani fuzzy system, output fuzzy sets are clipped by the truth values of the corresponding fuzzy rules. In the neuro-fuzzy system, we clip activation functions of the output membership neurons. For example, the membership function of neuron $C1$ is clipped by the integrated firing strength μ_{C1}.

Layer 5 is the **defuzzification layer**. Each neuron in this layer represents a single output of the neuro-fuzzy system. It takes the output fuzzy sets clipped by the respective integrated firing strengths and combines them into a single fuzzy set.

The output of the neuro-fuzzy system is crisp, and thus a combined output fuzzy set must be defuzzified. Neuro-fuzzy systems can apply standard defuzzification methods, including the centroid technique. In our example, we will use the **sum-product composition** method (Jang *et al.*, 1997), which offers a computational shortcut for the Mamdani-style inference.

The sum-product composition calculates the crisp output as the weighted average of the centroids of all output membership functions. For example, the weighted average of the centroids of the clipped fuzzy sets $C1$ and $C2$ is calculated as,

$$y = \frac{\mu_{C1} \times a_{C1} \times b_{C1} + \mu_{C2} \times a_{C2} \times b_{C2}}{\mu_{C1} \times b_{C1} + \mu_{C2} \times b_{C2}},$$ (8.7)

where a_{C1} and a_{C2} are the centres, and b_{C1} and b_{C2} are the widths of fuzzy sets $C1$ and $C2$, respectively.

How does a neuro-fuzzy system learn?

A neuro-fuzzy system is essentially a multi-layer neural network, and thus it can apply standard learning algorithms developed for neural networks, including the back-propagation algorithm (Kasabov, 1996; Lin and Lee, 1996; Nauck *et al.*, 1997; Von Altrock, 1997). When a training input-output example is presented to the system, the back-propagation algorithm computes the system output and compares it with the desired output of the training example. The difference (also called the error) is propagated backwards through the network from the output
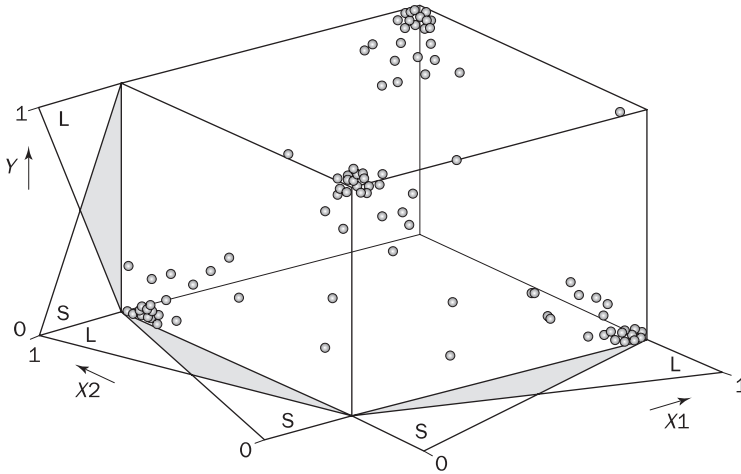
Figure 8.7   Training patterns in the three-dimensional input-output space

layer to the input layer. The neuron activation functions are modified as the error is propagated. To determine the necessary modifications, the back-propagation algorithm differentiates the activation functions of the neurons.

Let us demonstrate how a neuro-fuzzy system works on a simple example. Figure 8.7 shows the distribution of 100 training patterns in the three-dimensional input-output space $X1 \times X2 \times Y$. Each training pattern here is determined by three variables: two inputs $x1$ and $x2$, and one output y. Input and output variables are represented by two linguistic values: *small* (*S*) and *large* (*L*).

The data set of Figure 8.7 is used for training the five-rule neuro-fuzzy system shown in Figure 8.8(a). Suppose that fuzzy IF-THEN rules incorporated into the system structure are supplied by a domain expert. **Prior** or existing knowledge can dramatically expedite the system training. Besides, if the quality of training data is poor, the expert knowledge may be the only way to come to a solution at all. However, experts do occasionally make mistakes, and thus some rules used in a neuro-fuzzy system may be false or redundant (for example, in Figure 8.8(a), either Rule 1 or Rule 2 is wrong because they have exactly the same IF parts, while their THEN parts are different). Therefore, a neuro-fuzzy system should also be capable of identifying bad rules.

In Figure 8.8(a), initial weights between Layer 3 and Layer 4 are set to unity. During training the neuro-fuzzy system uses the back-propagation algorithm to adjust the weights and to modify input and output membership functions. The training continues until the sum of squared errors is less than 0.001. As can be seen from Figure 8.8(b), weight w_{R2} becomes equal to 0 while other weights remain high. This indicates that Rule 2 is certainly false and can be removed without any harm to the neuro-fuzzy system. It leaves the system with four rules that, as you may notice, represent the behaviour of the Exclusive-OR (XOR) operation.

(a)

(b)
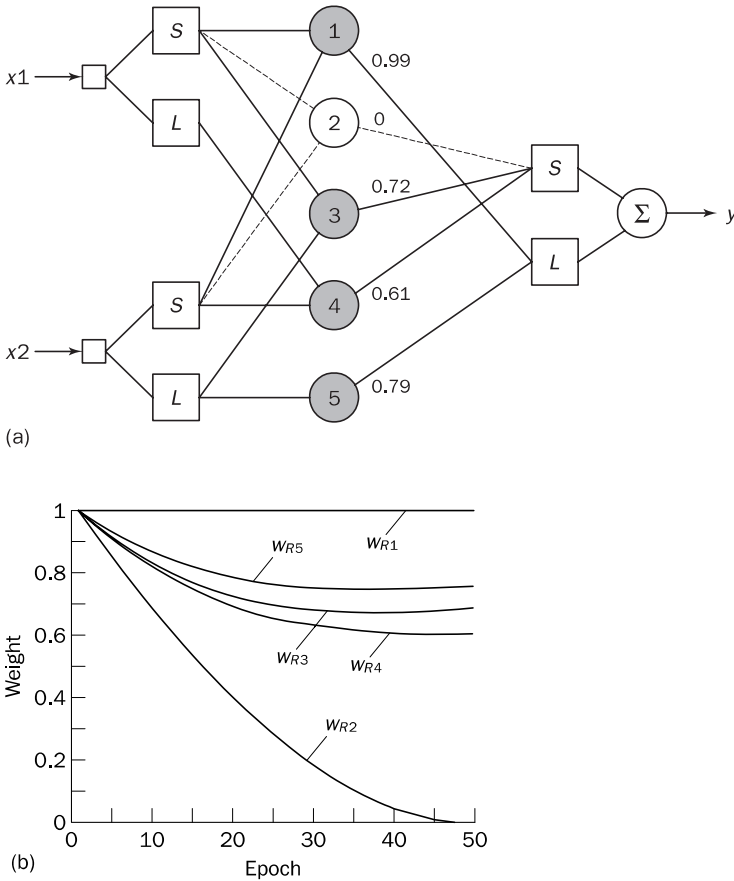
Figure 8.8   Five-rule neuro-fuzzy system for the Exclusive-OR operation:
(a) five-rule system; (b) training for 50 epochs

The training data used in this example includes a number of 'bad' patterns inconsistent with the XOR operation. However, the neuro-fuzzy system is still capable of identifying the false rule.

In the XOR example, an expert gives us five fuzzy rules, one of which is wrong. On top of that, we cannot be sure that the 'expert' has not left out a few rules. What can we do to reduce our dependence on the expert knowledge? Can a neuro-fuzzy system extract rules directly from numerical data?

Given input and output linguistic values, a neuro-fuzzy system can automatically generate a complete set of fuzzy IF-THEN rules. Figure 8.9 demonstrates the system created for the XOR example. This system consists of $2^2 \times 2 = 8$ rules. Because expert knowledge is not embodied in the system this time, we set all initial weights between Layer 3 and Layer 4 to 0.5. After training we can eliminate all rules whose certainty factors are less than some sufficiently small number, say 0.1. As a result, we obtain the same set of four fuzzy IF-THEN rules that represents

(a)

(b)
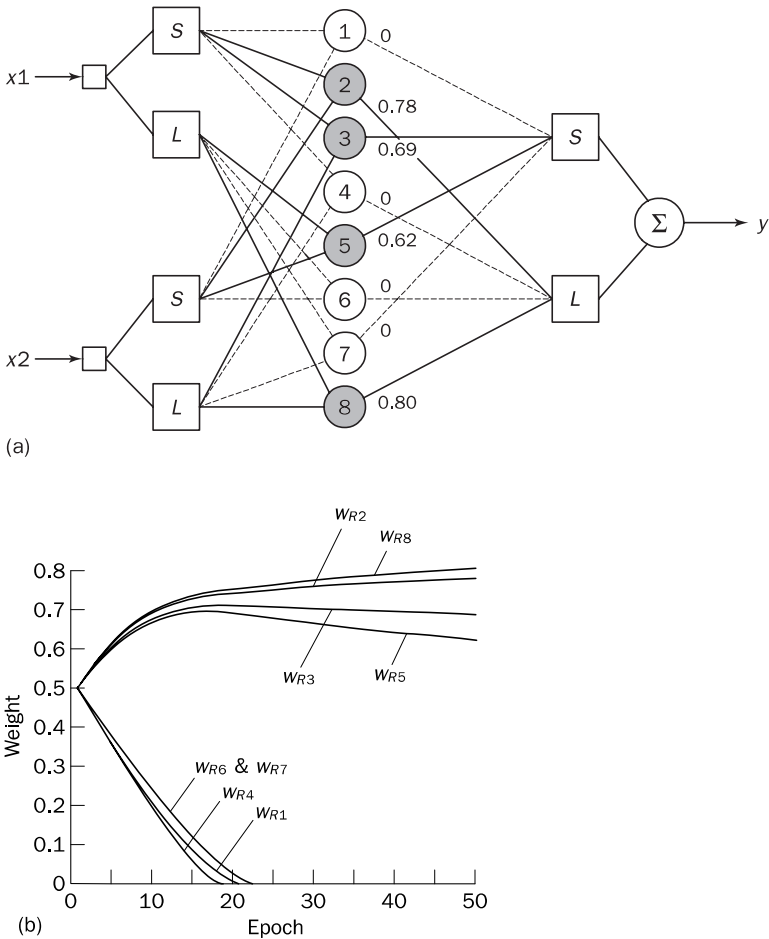
Figure 8.9 Eight-rule neuro-fuzzy system for the Exclusive-OR operation: (a) eight-rule system; (b) training for 50 epochs

the XOR operation. This simple example demonstrates that a neuro-fuzzy system can indeed extract fuzzy rules directly from numerical data.

The combination of fuzzy logic and neural networks constitutes a powerful means for designing intelligent systems. Domain knowledge can be put into a neuro-fuzzy system by human experts in the form of linguistic variables and fuzzy rules. When a representative set of examples is available, a neuro-fuzzy system can automatically transform it into a robust set of fuzzy IF-THEN rules, and thereby reduce our dependence on expert knowledge when building intelligent systems.

So far we have discussed a neuro-fuzzy system that implements the Mamdani fuzzy inference model. However, the Sugeno model is by far the most popular candidate for data-based fuzzy modelling.

Recently, Roger Jang from the Tsing Hua University, Taiwan, proposed a neural network that is functionally equal to a Sugeno fuzzy inference model (Jang, 1993). He called it an **Adaptive Neuro-Fuzzy Inference System** or **ANFIS**.

8.4 ANFIS: Adaptive Neuro-Fuzzy Inference System

The Sugeno fuzzy model was proposed for a systematic approach to generating fuzzy rules from a given input-output data set. A typical Sugeno fuzzy rule can be expressed in the following form:

IF x_1 is A_1
AND x_2 is A_2
· · · · ·
AND x_m is A_m
THEN $y = f(x_1, x_2, \ldots, x_m)$

where $x_1, x_2, \ldots, x_m$ are input variables; $A_1, A_2, \ldots, A_m$ are fuzzy sets; and y is either a constant or a linear function of the input variables. When y is a constant, we obtain a zero-order Sugeno fuzzy model in which the consequent of a rule is specified by a singleton. When y is a first-order polynomial, i.e.

$$y = k_0 + k_1 x_1 + k_2 x_2 + \ldots + k_m x_m$$

we obtain a first-order Sugeno fuzzy model.

Jang's ANFIS is normally represented by a six-layer feedforward neural network. Figure 8.10 shows the ANFIS architecture that corresponds to the first-order Sugeno fuzzy model. For simplicity, we assume that the ANFIS has two inputs – $x1$ and $x2$ – and one output – y. Each input is represented by two fuzzy
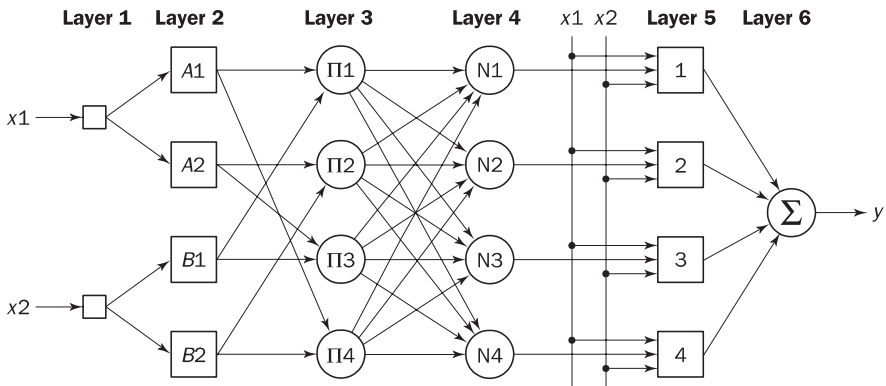
Figure 8.10   Adaptive Neuro-Fuzzy Inference System (ANFIS)

sets, and the output by a first-order polynomial. The ANFIS implements four rules:

Rule 1:
IF $x1$ is $A1$
AND $x2$ is $B1$
THEN $y = f_1 = k_{10} + k_{11}x1 + k_{12}x2$

Rule 2:
IF $x1$ is $A2$
AND $x2$ is $B2$
THEN $y = f_2 = k_{20} + k_{21}x1 + k_{22}x2$

Rule 3:
IF $x1$ is $A2$
AND $x2$ is $B1$
THEN $y = f_3 = k_{30} + k_{31}x1 + k_{32}x2$

Rule 4:
IF $x1$ is $A1$
AND $x2$ is $B2$
THEN $y = f_4 = k_{40} + k_{41}x1 + k_{42}x2$

where $x1$, $x2$ are input variables; $A1$ and $A2$ are fuzzy sets on the universe of discourse $X1$; $B1$ and $B2$ are fuzzy sets on the universe of discourse $X2$; and k_{i0}, k_{i1} and k_{i2} is a set of parameters specified for rule i.

Let us now discuss the purpose of each layer in Jang's ANFIS.

Layer 1 is the **input layer**. Neurons in this layer simply pass external crisp signals to Layer 2. That is,

$$y_i^{(1)} = x_i^{(1)}, \tag{8.8}$$

where $x_i^{(1)}$ is the input and $y_i^{(1)}$ is the output of input neuron i in Layer 1.

Layer 2 is the **fuzzification layer**. Neurons in this layer perform fuzzification. In Jang's model, fuzzification neurons have a **bell activation function**.

A bell activation function, which has a regular bell shape, is specified as:

$$y_i^{(2)} = \frac{1}{1 + \left(\dfrac{x_i^{(2)} - a_i}{c_i} \right)^{2b_i}}, \tag{8.9}$$

where $x_i^{(2)}$ is the input and $y_i^{(2)}$ is the output of neuron i in Layer 2; and a_i, b_i and c_i are parameters that control, respectively, the centre, width and slope of the bell activation function of neuron i.

Layer 3 is the **rule layer**. Each neuron in this layer corresponds to a single Sugeno-type fuzzy rule. A rule neuron receives inputs from the respective fuzzification neurons and calculates the firing strength of the rule it represents. In an ANFIS, the conjunction of the rule antecedents is evaluated by the operator **product**. Thus, the output of neuron i in Layer 3 is obtained as,

$$y_i^{(3)} = \prod_{j=1}^{k} x_{ji}^{(3)}, \tag{8.10}$$

where $x_{ji}^{(3)}$ are the inputs and $y_i^{(3)}$ is the output of rule neuron i in Layer 3.

For example,

$$y_{\Pi 1}^{(3)} = \mu_{A1} \times \mu_{B1} = \mu_1,$$

where the value of μ_1 represents the firing strength, or the truth value, of Rule 1.

Layer 4 is the **normalisation layer**. Each neuron in this layer receives inputs from all neurons in the rule layer, and calculates the **normalised firing strength** of a given rule.

The normalised firing strength is the ratio of the firing strength of a given rule to the sum of firing strengths of all rules. It represents the contribution of a given rule to the final result.

Thus, the output of neuron i in Layer 4 is determined as,

$$y_i^{(4)} = \frac{x_{ii}^{(4)}}{\sum\limits_{j=1}^{n} x_{ji}^{(4)}} = \frac{\mu_i}{\sum\limits_{j=1}^{n} \mu_j} = \bar{\mu}_i, \tag{8.11}$$

where $x_{ji}^{(4)}$ is the input from neuron j located in Layer 3 to neuron i in Layer 4, and n is the total number of rule neurons. For example,

$$y_{N1}^{(4)} = \frac{\mu_1}{\mu_1 + \mu_2 + \mu_3 + \mu_4} = \bar{\mu}_1$$

Layer 5 is the **defuzzification layer**. Each neuron in this layer is connected to the respective normalisation neuron, and also receives initial inputs, x_1 and x_2. A defuzzification neuron calculates the weighted consequent value of a given rule as,

$$y_i^{(5)} = x_i^{(5)}[k_{i0} + k_{i1}x1 + k_{i2}x2] = \bar{\mu}_i[k_{i0} + k_{i1}x1 + k_{i2}x2], \tag{8.12}$$

where $x_i^{(5)}$ is the input and $y_i^{(5)}$ is the output of defuzzification neuron i in Layer 5, and k_{i0}, k_{i1} and k_{i2} is a set of consequent parameters of rule i.

Layer 6 is represented by a single **summation neuron**. This neuron calculates the sum of outputs of all defuzzification neurons and produces the overall ANFIS output, y,

$$y = \sum_{i=1}^{n} x_i^{(6)} = \sum_{i=1}^{n} \bar{\mu}_i[k_{i0} + k_{i1}x1 + k_{i2}x2] \tag{8.13}$$

Thus, the ANFIS shown in Figure 8.10 is indeed functionally equivalent to a first-order Sugeno fuzzy model.

However, it is often difficult or even impossible to specify a rule consequent in a polynomial form. Conveniently, it is not necessary to have any prior knowledge of rule consequent parameters for an ANFIS to deal with a problem. An ANFIS learns these parameters and tunes membership functions.

How does an ANFIS learn?

An ANFIS uses a hybrid learning algorithm that combines the least-squares estimator and the gradient descent method (Jang, 1993). First, initial activation functions are assigned to each membership neuron. The function centres of the neurons connected to input x_i are set so that the domain of x_i is divided equally, and the widths and slopes are set to allow sufficient overlapping of the respective functions.

In the ANFIS training algorithm, each epoch is composed from a forward pass and a backward pass. In the forward pass, a training set of input patterns (an input vector) is presented to the ANFIS, neuron outputs are calculated on the layer-by-layer basis, and rule consequent parameters are identified by the least-squares estimator. In the Sugeno-style fuzzy inference, an output, y, is a linear function. Thus, given the values of the membership parameters and a training set of P input-output patterns, we can form P linear equations in terms of the consequent parameters as:

$$
\begin{cases}
y_d(1) = \bar{\mu}_1(1)f_1(1) + \bar{\mu}_2(1)f_2(1) + \ldots + \bar{\mu}_n(1)f_n(1) \\
y_d(2) = \bar{\mu}_1(2)f_1(2) + \bar{\mu}_2(2)f_2(2) + \ldots + \bar{\mu}_n(2)f_n(2) \\
\qquad\qquad\qquad \vdots \\
y_d(p) = \bar{\mu}_1(p)f_1(p) + \bar{\mu}_2(p)f_2(p) + \ldots + \bar{\mu}_n(p)f_n(p) \\
\qquad\qquad\qquad \vdots \\
y_d(P) = \bar{\mu}_1(P)f_1(P) + \bar{\mu}_2(P)f_2(P) + \ldots + \bar{\mu}_n(P)f_n(P)
\end{cases}
\tag{8.14}
$$

or

$$
\begin{cases}
y_d(1) = \bar{\mu}_1(1)[k_{10} + k_{11}x_1(1) + k_{12}x_2(1) + \ldots + k_{1m}x_m(1)] \\
\qquad + \bar{\mu}_2(1)[k_{20} + k_{21}x_1(1) + k_{22}x_2(1) + \ldots + k_{2m}x_m(1)] + \ldots \\
\qquad + \bar{\mu}_n(1)[k_{n0} + k_{n1}x_1(1) + k_{n2}x_2(1) + \ldots + k_{nm}x_m(1)] \\
y_d(2) = \bar{\mu}_1(2)[k_{10} + k_{11}x_1(2) + k_{12}x_2(2) + \ldots + k_{1m}x_m(2)] \\
\qquad + \bar{\mu}_2(2)[k_{20} + k_{21}x_1(2) + k_{22}x_2(2) + \ldots + k_{2m}x_m(2)] + \ldots \\
\qquad + \bar{\mu}_n(2)[k_{n0} + k_{n1}x_1(2) + k_{n2}x_2(2) + \ldots + k_{nm}x_m(2)] \\
\qquad\qquad\qquad \vdots \\
y_d(p) = \bar{\mu}_1(p)[k_{10} + k_{11}x_1(p) + k_{12}x_2(p) + \ldots + k_{1m}x_m(p)] \\
\qquad + \bar{\mu}_2(p)[k_{20} + k_{21}x_1(p) + k_{22}x_2(p) + \ldots + k_{2m}x_m(p)] + \ldots \\
\qquad + \bar{\mu}_n(p)[k_{n0} + k_{n1}x_1(p) + k_{n2}x_2(p) + \ldots + k_{nm}x_m(p)] \\
\qquad\qquad\qquad \vdots \\
y_d(P) = \bar{\mu}_1(P)[k_{10} + k_{11}x_1(P) + k_{12}x_2(P) + \ldots + k_{1m}x_m(P)] \\
\qquad + \bar{\mu}_2(P)[k_{20} + k_{21}x_1(P) + k_{22}x_2(P) + \ldots + k_{2m}x_m(P)] + \ldots \\
\qquad + \bar{\mu}_n(P)[k_{n0} + k_{n1}x_1(P) + k_{n2}x_2(P) + \ldots + k_{nm}x_m(P)]
\end{cases}
\tag{8.15}
$$

where m is the number of input variables, n is the number of neurons in the rule layer, and $y_d(p)$ is the desired overall output of the ANFIS when inputs $x_1(p)$, $x_2(p), \ldots, x_m(p)$ are presented to it.

In the matrix notation, we have

$$y_d = A k, \tag{8.16}$$

where y_d is a $P \times 1$ desired output vector,

$$y_d = \begin{bmatrix} y_d(1) \\ y_d(2) \\ \vdots \\ y_d(p) \\ \vdots \\ y_d(P) \end{bmatrix}$$

A is a $P \times n(1+m)$ matrix,

$$A = \begin{bmatrix} \bar{\mu}_1(1) & \bar{\mu}_1(1)x_1(1) & \cdots & \bar{\mu}_1(1)x_m(1) & \cdots & \bar{\mu}_n(1) & \bar{\mu}_n(1)x_1(1) & \cdots & \bar{\mu}_n(1)x_m(1) \\ \bar{\mu}_1(2) & \bar{\mu}_1(2)x_1(2) & \cdots & \bar{\mu}_1(2)x_m(2) & \cdots & \bar{\mu}_n(2) & \bar{\mu}_n(2)x_1(2) & \cdots & \bar{\mu}_n(2)x_m(2) \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ \bar{\mu}_1(p) & \bar{\mu}_1(p)x_1(p) & \cdots & \bar{\mu}_1(p)x_m(p) & \cdots & \bar{\mu}_n(p) & \bar{\mu}_n(p)x_1(p) & \cdots & \bar{\mu}_n(p)x_m(p) \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ \bar{\mu}_1(P) & \bar{\mu}_1(P)x_1(P) & \cdots & \bar{\mu}_1(P)x_m(P) & \cdots & \bar{\mu}_n(P) & \bar{\mu}_n(P)x_1(P) & \cdots & \bar{\mu}_n(P)x_m(P) \end{bmatrix}$$

and k is an $n(1+m) \times 1$ vector of unknown consequent parameters,

$$k = [k_{10} \ k_{11} \ k_{12} \ldots k_{1m} \ k_{20} \ k_{21} \ k_{22} \ldots k_{2m} \ldots k_{n0} \ k_{n1} \ k_{n2} \ldots k_{nm}]^T$$

Usually the number of input-output patterns P used in training is greater than the number of consequent parameters $n(1+m)$. It means that we are dealing here with an overdetermined problem, and thus exact solution to Eq. (8.16) may not even exist. Instead, we should find a least-square estimate of k, k^*, that minimises the squared error $\|A k - y_d\|^2$. It is done by using the pseudoinverse technique:

$$k^* = (A^T A)^{-1} A^T y_d, \tag{8.17}$$

where A^T is the transpose of A, and $(A^T A)^{-1} A^T$ is the pseudoinverse of A if $(A^T A)$ is non-singular.

As soon as the rule consequent parameters are established, we can compute an actual network output vector, y, and determine the error vector, e,

$$e = y_d - y \tag{8.18}$$

In the backward pass, the back-propagation algorithm is applied. The error signals are propagated back, and the antecedent parameters are updated according to the chain rule.

Let us, for instance, consider a correction applied to parameter a of the bell activation function used in neuron $A1$. We may express the chain rule as follows:

$$\Delta a = -\alpha \frac{\partial E}{\partial a} = -\alpha \frac{\partial E}{\partial e} \times \frac{\partial e}{\partial y} \times \frac{\partial y}{\partial (\bar{\mu}_i\, f_i)} \times \frac{\partial (\bar{\mu}_i\, f_i)}{\partial \bar{\mu}_i} \times \frac{\partial \bar{\mu}_i}{\partial \mu_i} \times \frac{\partial \mu_i}{\partial \mu_{A1}} \times \frac{\partial \mu_{A1}}{\partial a}, \tag{8.19}$$

where α is the learning rate, and E is the instantaneous value of the squared error for the ANFIS output neuron, i.e.,

$$E = \frac{1}{2}e^2 = \frac{1}{2}(y_d - y)^2 \tag{8.20}$$

Thus, we have

$$\Delta a = -\alpha\,(y_d - y)(-1)f_i \times \frac{\bar{\mu}_i(1 - \bar{\mu}_i)}{\mu_i} \times \frac{\mu_i}{\mu_{A1}} \times \frac{\partial \mu_{A1}}{\partial a} \tag{8.21}$$

or

$$\Delta a = \alpha\,(y_d - y)f_i\,\bar{\mu}_i(1 - \bar{\mu}_i) \times \frac{1}{\mu_{A1}} \times \frac{\partial \mu_{A1}}{\partial a}, \tag{8.22}$$

where

$$\frac{\partial \mu_{A1}}{\partial a} = -\frac{1}{\left[1 + \left(\dfrac{x1 - a}{c}\right)^{2b}\right]^2} \times \frac{1}{c^{2b}} \times 2b \times (x1 - a)^{2b-1} \times (-1)$$

$$= \mu_{A1}^2 \times \frac{2b}{c} \times \left(\frac{x1 - a}{c}\right)^{2b-1}$$

Similarly, we can obtain corrections applied to parameters b and c.

In the ANFIS training algorithm suggested by Jang, both antecedent parameters and consequent parameters are optimised. In the forward pass, the consequent parameters are adjusted while the antecedent parameters remain fixed. In the backward pass, the antecedent parameters are tuned while the consequent parameters are kept fixed. However, in some cases, when the input-output data set is relatively small, membership functions can be described by a human expert. In such situations, these membership functions are kept fixed throughout the training process, and only consequent parameters are adjusted (Jang *et al.*, 1997).

Let us now demonstrate an application of an ANFIS for function approximation. In this example, an ANFIS is used to follow a trajectory of the non-linear function defined by the equation

$$y = \frac{\cos(2\,x1)}{e^{x2}}.$$

First, we choose an appropriate architecture for the ANFIS. An ANFIS must have two inputs – $x1$ and $x2$ – and one output – y.

We decide on the number of membership functions to be assigned to each input by choosing the smallest number of membership functions that yields a 'satisfactory' performance. Thus, the experimental study may begin with two membership functions assigned to each input variable.

To build an ANFIS, we choose either a programming language, for example C/C++, or a neuro-fuzzy development tool. We will use one of the most popular tools – the MATLAB Fuzzy Logic Toolbox. It provides a systematic framework for building neuro-fuzzy inference systems and defines rules automatically based on the number of membership functions assigned to each input variable. Thus, in our example, the ANFIS is defined by four rules, and in fact has the structure shown in Figure 8.10.

The ANFIS training data includes 101 training samples. They are represented by a 101×3 matrix $[x1\ x2\ y_d]$, where $x1$ and $x2$ are input vectors, and y_d is a desired output vector. The first input vector, $x1$, starts at 0, increments by 0.1 and ends at 10. The second input vector, $x2$, is created by taking the sine of each element of vector $x1$. Finally, each element of the desired output vector, y_d, is determined by the function equation.

An actual trajectory of the function and the ANFIS's output after 1 and 100 epochs of training are depicted in Figure 8.11. Note that Figure 8.11(a) represents results after the least-squares estimator identified the rule consequent para-
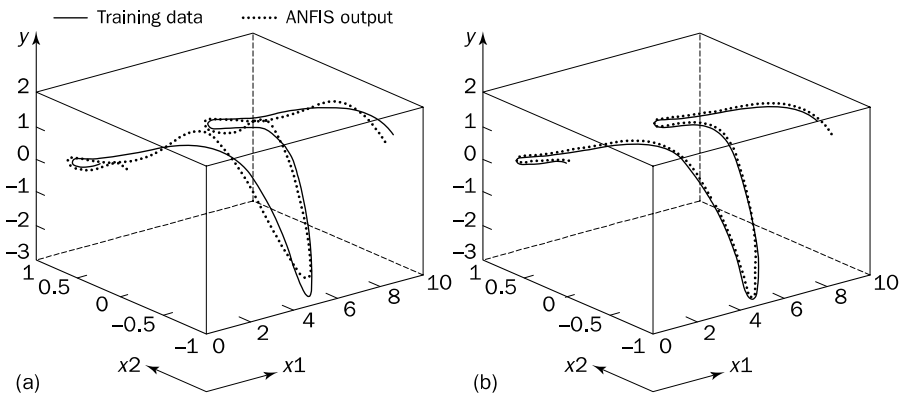
Figure 8.11 Learning in an ANFIS with two membership functions assigned to each input: (a) one epoch; (b) 100 epochs
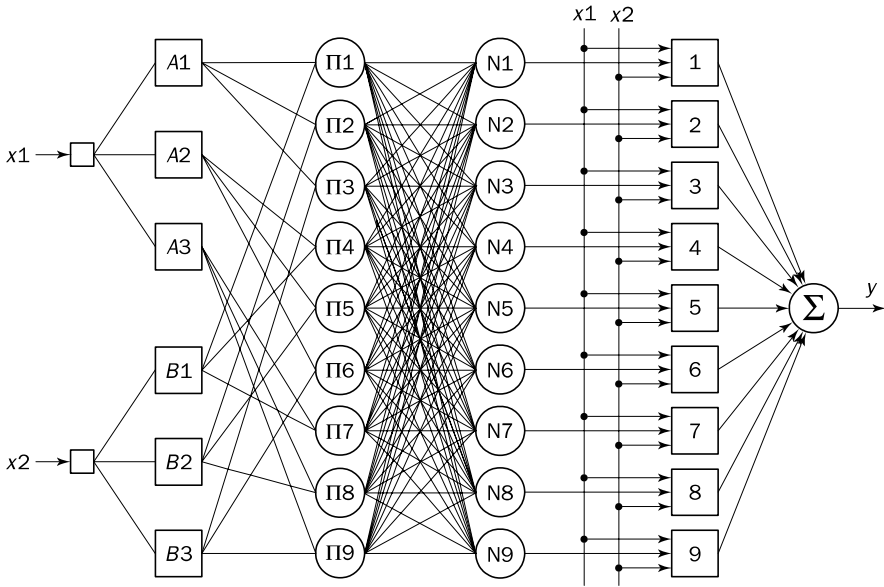
Figure 8.12   An ANFIS model with nine rules

meters for the first time. As we can see, the ANFIS's performance is not always adequate even after 100 epochs of training.

We can achieve some improvement in an ANFIS's performance by increasing the number of epochs, but much better results are obtained when we assign three membership functions to each input variable. In this case, the ANFIS model will have nine rules, as shown in Figure 8.12.

Figure 8.13 shows that the ANFIS's performance improves significantly, and even after one epoch its output quite accurately resembles the desired trajectory. Figure 8.14 illustrates the membership functions before and after training.
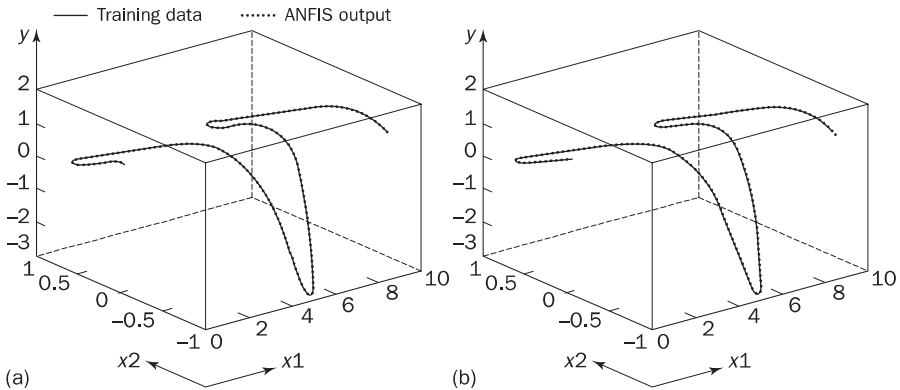
Figure 8.13   Learning in an ANFIS with three membership functions assigned to each input: (a) one epoch; (b) 100 epochs
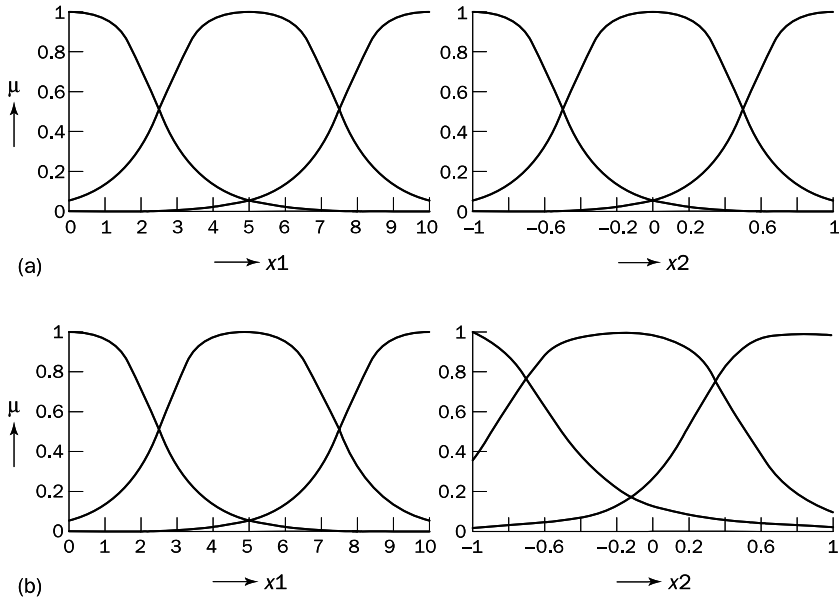
Figure 8.14   Initial and final membership functions of the ANFIS: (a) initial membership functions; (b) membership functions after 100 epochs of training

The ANFIS has a remarkable ability to generalise and converge rapidly. This is particularly important in on-line learning. As a result, Jang's model and its variants are finding numerous applications, especially in adaptive control.

8.5   Evolutionary neural networks

Although neural networks are used for solving a variety of problems, they still have some limitations. One of the most common is associated with neural network training. The back-propagation learning algorithm that is often used because it is flexible and mathematically tractable (given that the transfer functions of neurons can be differentiated) has a serious drawback: it cannot guarantee an optimal solution. In real-world applications, the back-propagation algorithm might converge to a set of sub-optimal weights from which it cannot escape. As a result, the neural network is often unable to find a desirable solution to a problem at hand.

Another difficulty is related to selecting an optimal topology for the neural network. The 'right' network architecture for a particular problem is often chosen by means of heuristics, and designing a neural network topology is still more art than engineering.

Genetic algorithms are an effective optimisation technique that can guide both weight optimisation and topology selection.

Let us first consider the basic concept of an evolutionary weight optimisation technique (Montana and Davis, 1989; Whitley and Hanson, 1989; Ichikawa and
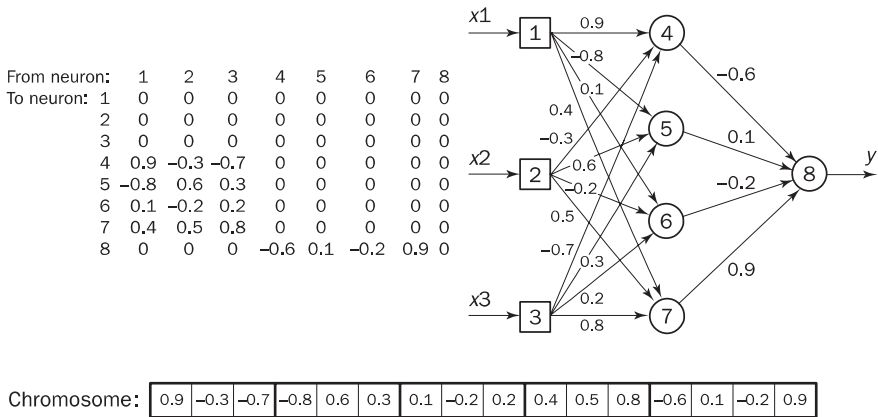
Figure 8.15　Encoding a set of weights in a chromosome

Sawa, 1992). To use genetic algorithms, we first need to represent the problem domain as a chromosome. Suppose, for example, we want to find an optimal set of weights for the multilayer feedforward neural network shown in Figure 8.15.

Initial weights in the network are chosen randomly within some small interval, say $[-1, 1]$. The set of weights can be represented by a square matrix in which a real number corresponds to the weighted link from one neuron to another, and 0 means that there is no connection between two given neurons. In total, there are 16 weighted links between neurons in Figure 8.15. Since a chromosome is a collection of genes, a set of weights can be represented by a 16-gene chromosome, where each gene corresponds to a single weighted link in the network. Thus, if we string the rows of the matrix together, ignoring zeros, we obtain a chromosome.

In addition, each row now represents a group of all the incoming weighted links to a single neuron. This group can be thought of as a functional building block of the network (Montana and Davis, 1989), and therefore should be allowed to stay together passing genetic material from one generation to the next. To achieve this, we should associate each gene not with a single weight but rather with a group of all incoming weights of a given neuron, as shown in Figure 8.15.

The second step is to define a fitness function for evaluating the chromosome's performance. This function must estimate the performance of a given neural network. We can apply here a fairly simple function defined by the reciprocal of the sum of squared errors. To evaluate the fitness of a given chromosome, each weight contained in the chromosome is assigned to the respective link in the network. The training set of examples is then presented to the network, and the sum of squared errors is calculated. The smaller the sum, the fitter the chromosome. In other words, the genetic algorithm attempts to find a set of weights that minimises the sum of squared errors.

The third step is to choose the genetic operators – crossover and mutation. A crossover operator takes two parent chromosomes and creates a single child with
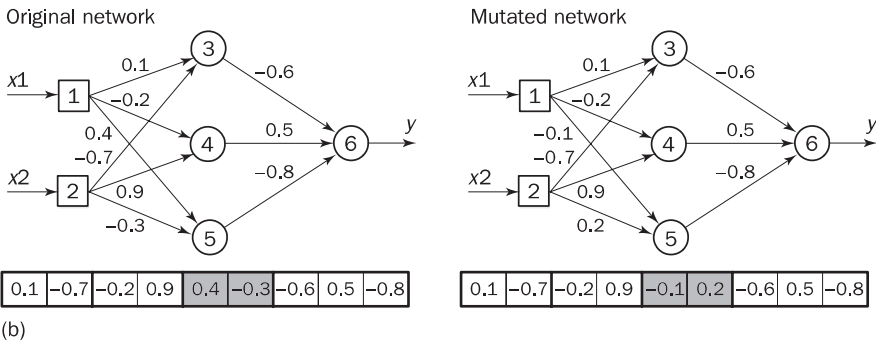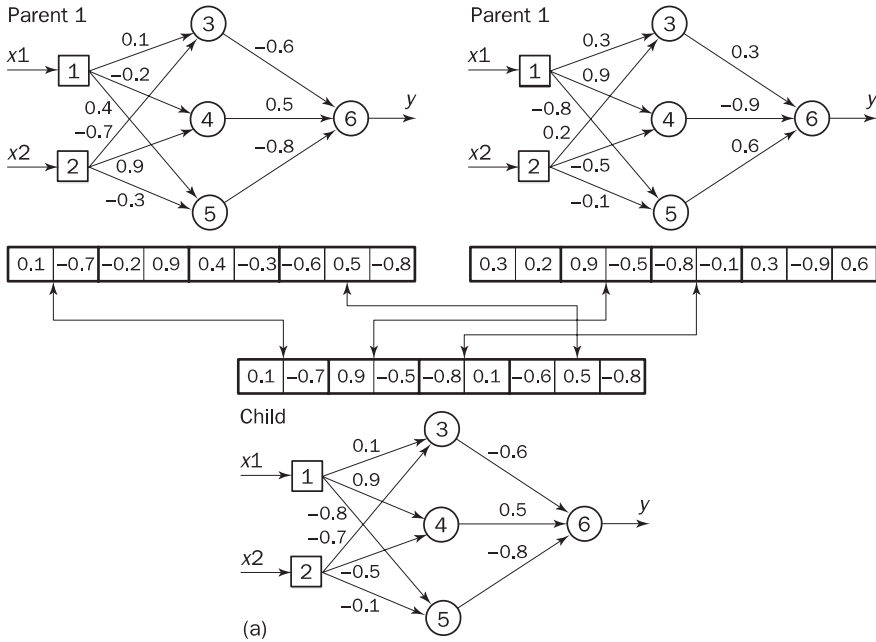
Figure 8.16 Genetic operations in neural network weight optimisation: (a) crossover; (b) mutation

genetic material from both parents. Each gene in the child's chromosome is represented by the corresponding gene of the randomly selected parent. Figure 8.16(a) shows an application of the crossover operator.

A mutation operator randomly selects a gene in a chromosome and adds a small random value between -1 and 1 to each weight in this gene. Figure 8.16(b) shows an example of mutation.

Now we are ready to apply the genetic algorithm. Of course, we still need to define the population size, i.e. the number of networks with different weights, the crossover and mutation probabilities and the number of generations.

```
From neuron:   1  2  3  4  5  6
To neuron: 1   0  0  0  0  0  0
           2   0  0  0  0  0  0
           3   1  1  0  0  0  0
           4   1  0  0  0  0  0
           5   0  1  0  0  0  0
           6   0  1  1  1  1  0
```

Chromosome:

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
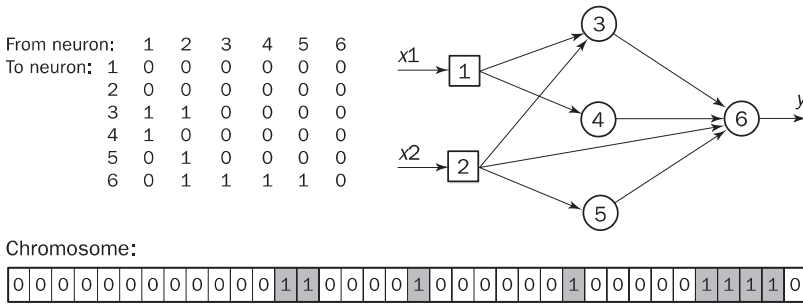
Figure 8.17   Direct encoding of the network topology

So far we have assumed that the structure of the network is fixed, and evolutionary learning is used only to optimise weights in the given network. However, the architecture of the network (i.e. the number of neurons and their interconnections) often determines the success or failure of the application. Usually the network architecture is decided by trial and error; there is a great need for a method of automatically designing the architecture for a particular application. Genetic algorithms may well help us in selecting the network architecture.

The basic idea behind evolving a suitable network architecture is to conduct a genetic search in a population of possible architectures (Miller *et al.*, 1989; Schaffer *et al.*, 1992). Of course, we must first choose a method of encoding a network's architecture into a chromosome.

There are many different ways to encode the network's structure. The key is to decide how much information is required for the network representation. The more parameters of the network architecture, the greater the computational cost. As an illustration, we can consider a simple direct method of encoding (Miller *et al.*, 1989). Although direct encoding is a restricted technique, and can be applied only to feedforward networks with a fixed number of neurons, it demonstrates how a connection topology is evolved.

The connection topology of a neural network can be represented by a square connectivity matrix, as shown in Figure 8.17. Each entry in the matrix defines the type of connection from one neuron (column) to another (row), where 0 means no connection and 1 denotes connection for which the weight can be changed through learning. To transform the connectivity matrix into a chromosome, we need only to string the rows of the matrix together, as shown in Figure 8.17.

Given a set of training examples and a binary string representation for possible network architectures, a basic GA can be described by the following steps:

Step 1: Choose the size of a chromosome population, the crossover and mutation probabilities, and define the number of training epochs.

Step 2: Define a fitness function to measure the performance, or fitness, of an individual chromosome. In general, the network's fitness should be

based not only on its accuracy, but also on its learning speed, size and complexity. However, the network's performance is much more important than its size, and therefore the fitness function can still be defined by the reciprocal of the sum of squared errors.

Step 3: Randomly generate an initial population of chromosomes.

Step 4: Decode an individual chromosome into a neural network. Since our networks are restricted to be feedforward, ignore all feedback connections specified in the chromosome. Set initial weights of the network to small random numbers, say in the range $[-1, 1]$. Train the network on a training set of examples for a certain number of epochs using the back-propagation algorithm. Calculate the sum of squared errors and determine the network's fitness.

Step 5: Repeat Step 4 until all the individuals in the population have been considered.

Step 6: Select a pair of chromosomes for mating, with a probability proportionate to their fitness.

Step 7: Create a pair of offspring chromosomes by applying the genetic operators crossover and mutation.

A crossover operator randomly chooses a row index and simply swaps the corresponding rows between two parents, creating two offspring. A mutation operator flips one or two bits in the chromosome with some low probability, say 0.005.

Step 8: Place the created offspring chromosomes in the new population.

Step 9: Repeat Step 6 until the size of the new chromosome population becomes equal to the size of the initial population, and then replace the initial (parent) chromosome population with the new (offspring) population.

Step 10: Go to Step 4, and repeat the process until a specified number of generations has been considered.

An evolutionary cycle of evolving a neural network topology is presented in Figure 8.18.

In addition to neural network training and topology selection, evolutionary computation can also be used to optimise transfer functions and select suitable input variables. Evolving a set of critical inputs from a large number of possible input variables with complex or unknown functional relationships is an area of current research that has a great potential for evolutionary neural networks. Further topics on new areas of evolutionary computation research in neural systems can be found in Bäck *et al* (1997).
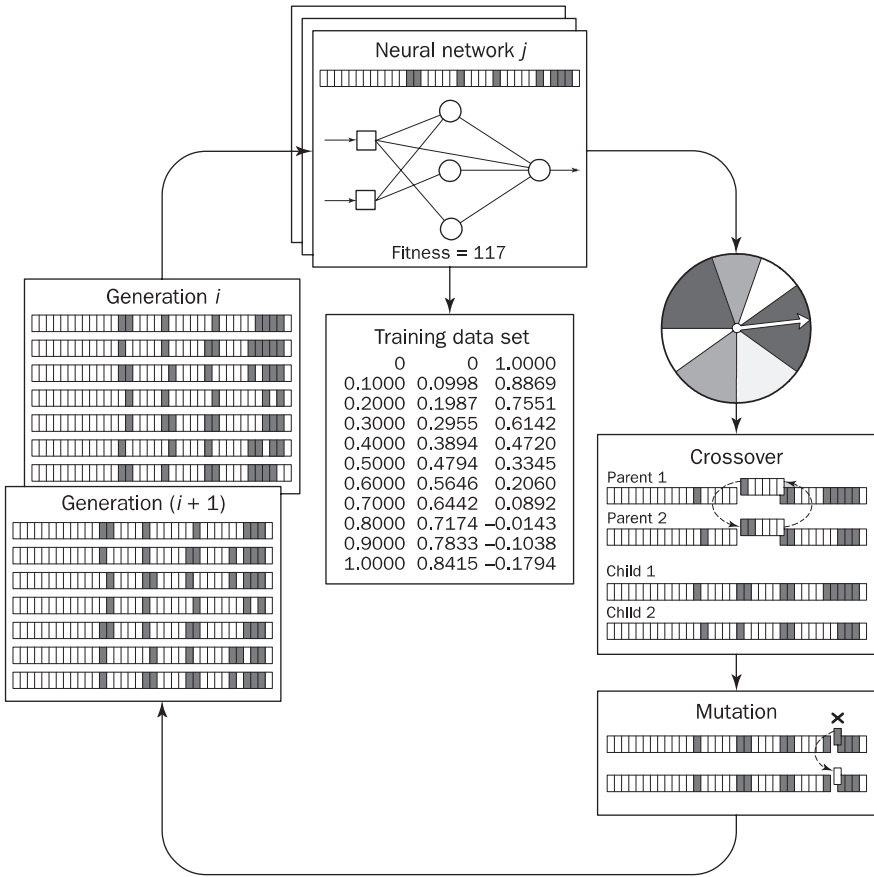
Figure 8.18   The evolutionary cycle of evolving a neural network topology

8.6   Fuzzy evolutionary systems

Evolutionary computation is also used in the design of fuzzy systems, particularly for generating fuzzy rules and adjusting membership functions of fuzzy sets. In this section, we introduce an application of genetic algorithms to select an appropriate set of fuzzy IF-THEN rules for a classification problem (Ishibuchi *et al.*, 1995).

To apply genetic algorithms, we need to have a population of feasible solutions – in our case, a set of fuzzy IF-THEN rules. We need to obtain this set. For a classification problem, a set of fuzzy IF-THEN rules can be generated from numerical data (Ishibuchi *et al.*, 1992). First, we use a **grid-type fuzzy partition** of an input space.

Figure 8.19 shows an example of the fuzzy partition of a two-dimensional input space into 3×3 fuzzy subspaces. Black and white dots here denote the training patterns of Class 1 and Class 2, respectively. The grid-type fuzzy
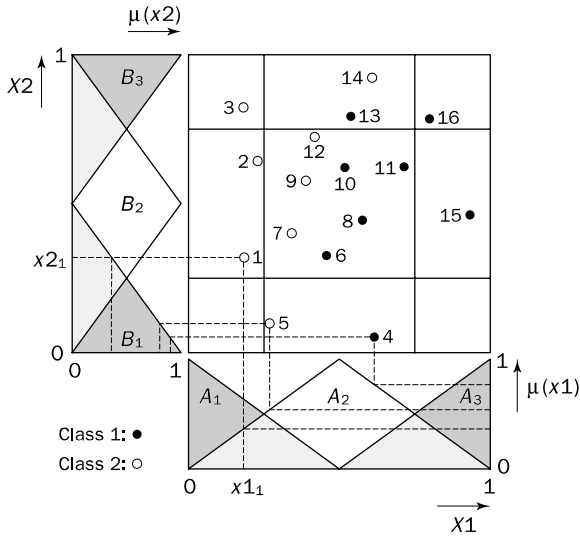
Figure 8.19 Fuzzy partition by a 3 × 3 fuzzy grid

partition can be seen as a rule table. The linguistic values of input $x1$ (A_1, A_2 and A_3) form the horizontal axis, and the linguistic values of input $x2$ (B_1, B_2 and B_3) form the vertical axis. At the intersection of a row and a column lies the rule consequent.

In the rule table, each fuzzy subspace can have only one fuzzy IF-THEN rule, and thus the total number of rules that can be generated in a $K \times K$ grid is equal to $K \times K$. Fuzzy rules that correspond to the $K \times K$ fuzzy partition can be represented in a general form as:

Rule R_{ij}:
IF $\quad x1_p$ is A_i $\qquad\qquad\qquad\qquad i = 1, 2, \ldots, K$
AND $\quad x2_p$ is B_j $\qquad\qquad\qquad\qquad j = 1, 2, \ldots, K$
THEN $\quad \mathbf{x}_p \in C_n \{CF_{A_iB_j}^{C_n}\} \qquad \mathbf{x}_p = (x1_p, x2_p), p = 1, 2, \ldots, P;$

where K is the number of fuzzy intervals in each axis, $\mathbf{x}_p$ is a training pattern on input space $X1 \times X2$, P is the total number of training patterns, C_n is the rule consequent (which, in our example, is either Class 1 or Class 2), and $CF_{A_iB_j}^{C_n}$ is the certainty factor or likelihood that a pattern in fuzzy subspace A_iB_j belongs to class C_n.

To determine the rule consequent and the certainty factor, we use the following procedure:

Step 1: Partition an input space into $K \times K$ fuzzy subspaces, and calculate the **strength** of each class of training patterns in every fuzzy subspace.

Each class in a given fuzzy subspace is represented by its training patterns. The more training patterns, the stronger the class. In other

words, in a given fuzzy subspace, the rule consequent becomes more certain when patterns of one particular class appear more often than patterns of any other class. The strength of class C_n in fuzzy subspace A_iB_j can be determined as:

$$\beta_{A_iB_j}^{C_n} = \sum_{\substack{p=1 \\ \mathbf{x}_p \in C_n}}^{P} \mu_{A_i}(x1_p) \times \mu_{B_j}(x2_p), \quad \mathbf{x}_p = (x1_p, x2_p), \tag{8.23}$$

where $\mu_{A_i}(x1_p)$ and $\mu_{B_j}(x2_p)$ are degrees of membership of training pattern $\mathbf{x}_p$ in fuzzy set A_i and fuzzy set B_j, respectively.

In Figure 8.19, for example, the strengths of Class 1 and Class 2 in fuzzy subspace A_2B_1 are calculated as:

$$\begin{aligned}
\beta_{A_2B_1}^{Class1} &= \mu_{A_2}(\mathbf{x}_4) \times \mu_{B_1}(\mathbf{x}_4) + \mu_{A_2}(\mathbf{x}_6) \times \mu_{B_1}(\mathbf{x}_6) + \mu_{A_2}(\mathbf{x}_8) \times \mu_{B_1}(\mathbf{x}_8) \\
&\quad + \mu_{A_2}(\mathbf{x}_{15}) \times \mu_{B_1}(\mathbf{x}_{15}) \\
&= 0.75 \times 0.89 + 0.92 \times 0.34 + 0.87 \times 0.12 + 0.11 \times 0.09 = 1.09
\end{aligned}$$

$$\begin{aligned}
\beta_{A_2B_1}^{Class2} &= \mu_{A_2}(\mathbf{x}_1) \times \mu_{B_1}(\mathbf{x}_1) + \mu_{A_2}(\mathbf{x}_5) \times \mu_{B_1}(\mathbf{x}_5) + \mu_{A_2}(\mathbf{x}_7) \times \mu_{B_1}(\mathbf{x}_7) \\
&= 0.42 \times 0.38 + 0.54 \times 0.81 + 0.65 \times 0.21 = 0.73
\end{aligned}$$

Step 2: Determine the rule consequent and the certainty factor in each fuzzy subspace. As the rule consequent is determined by the strongest class, we need to find class C_m such that,

$$\beta_{A_iB_j}^{C_m} = max \left[\beta_{A_iB_j}^{C_1}, \beta_{A_iB_j}^{C_2}, \ldots, \beta_{A_iB_j}^{C_N} \right] \tag{8.24}$$

If a particular class takes the maximum value, the rule consequent is determined as C_m. For example, in fuzzy subspace A_2B_1, the rule consequent is Class 1.

Then the certainty factor can be calculated:

$$CF_{A_iB_j}^{C_m} = \frac{\beta_{A_iB_j}^{C_m} - \bar{\beta}_{A_iB_j}}{\sum\limits_{n=1}^{N} \beta_{A_iB_j}^{C_n}}, \tag{8.25}$$

where

$$\bar{\beta}_{A_iB_j} = \frac{\sum\limits_{\substack{n=1 \\ n \neq m}}^{N} \beta_{A_iB_j}^{C_n}}{N-1} \tag{8.26}$$

For example, the certainty factor of the rule consequent corresponding to fuzzy subspace A_2B_1 can be calculated as:

$$CF_{A_2B_1}^{Class2} = \frac{1.09 - 0.73}{1.09 + 0.73} = 0.20$$
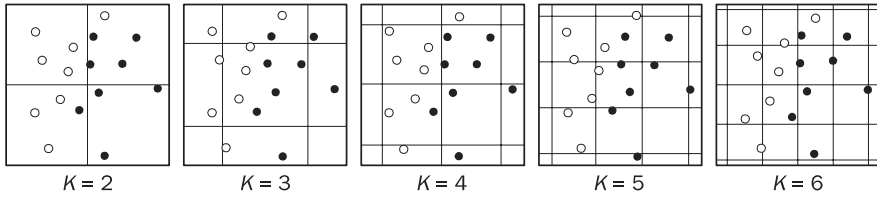
Figure 8.20 Multiple fuzzy rule tables

How do we interpret the certainty factor here?

The certainty factor specified by Eq. (8.25) can be interpreted as follows. If all the training patterns in fuzzy subspace A_iB_j belong to the same class C_m, then the certainty factor is maximum and it is certain that any new pattern in this subspace will belong to class C_m. If, however, training patterns belong to different classes and these classes have similar strengths, then the certainty factor is minimum and it is uncertain that a new pattern will belong to class C_m.

This means that patterns in fuzzy subspace A_2B_1 can be easily misclassified. Moreover, if a fuzzy subspace does not have any training patterns, we cannot determine the rule consequent at all. In fact, if a fuzzy partition is too coarse, many patterns may be misclassified. On the other hand, if a fuzzy partition is too fine, many fuzzy rules cannot be obtained, because of the lack of training patterns in the corresponding fuzzy subspaces. Thus, the choice of the density of a fuzzy grid is very important for the correct classification of an input pattern.

Meanwhile, as can be seen in Figure 8.19, training patterns are not necessarily distributed evenly in the input space. As a result, it is often difficult to choose an appropriate density for the fuzzy grid. To overcome this difficulty, we use multiple fuzzy rule tables (Ishibuchi *et al.*, 1992); an example of these is shown in Figure 8.20. The number of these tables depends on the complexity of the classification problem.

Fuzzy IF-THEN rules are generated for each fuzzy subspace of multiple fuzzy rule tables, and thus a complete set of rules can be specified as:

$$S_{ALL} = \sum_{K=2}^{L} S_K, \qquad K = 2, 3, \ldots, L \tag{8.27}$$

where S_K is the rule set corresponding to a fuzzy rule table K.

The set of rules S_{ALL} generated for multiple fuzzy rule tables shown in Figure 8.20 contains $2^2 + 3^3 + 4^4 + 5^5 + 6^6 = 90$ rules.

Once the set of rules S_{ALL} is generated, a new pattern, $\mathbf{x} = (x1, x2)$, can be classified by the following procedure:

Step 1: In every fuzzy subspace of the multiple fuzzy rule tables, calculate the degree of compatibility of a new pattern with each class:

$$\alpha_{K\{A_iB_j\}}^{C_n} = \mu_{K\{A_i\}}(x1) \times \mu_{K\{B_j\}}(x2) \times CF_{K\{A_iB_j\}}^{C_n} \tag{8.28}$$

$$n = 1, 2, \ldots, N; \quad K = 2, 3, \ldots, L; \quad i = 1, 2, \ldots, K; \quad j = 1, 2, \ldots, K$$

Step 2: Determine the maximum degree of compatibility of the new pattern with each class:

$$\alpha^{C_n} = max\left[\alpha^{C_n}_{1\{A_1B_1\}}, \alpha^{C_n}_{1\{A_1B_2\}}, \alpha^{C_n}_{1\{A_2B_1\}}, \alpha^{C_n}_{1\{A_2B_2\}},\right. \tag{8.29}$$

$$\alpha^{C_n}_{2\{A_1B_1\}}, \dots, \alpha^{C_n}_{2\{A_1B_K\}}, \alpha^{C_n}_{2\{A_2B_1\}}, \dots, \alpha^{C_n}_{2\{A_2B_K\}}, \dots, \alpha^{C_n}_{2\{A_KB_1\}}, \dots, \alpha^{C_n}_{2\{A_KB_K\}}, \dots,$$

$$\left.\alpha^{C_n}_{L\{A_1B_1\}}, \dots, \alpha^{C_n}_{L\{A_1B_K\}}, \alpha^{C_n}_{L\{A_2B_1\}}, \dots, \alpha^{C_n}_{L\{A_2B_K\}}, \dots, \alpha^{C_n}_{L\{A_KB_1\}}, \dots, \alpha^{C_n}_{L\{A_KB_K\}}\right]$$

$$n = 1, 2, \dots, N$$

Step 3: Determine class C_m with which the new pattern has the highest degree of compatibility, that is:

$$\alpha^{C_m} = max\left[\alpha^{C_1}, \alpha^{C_2}, \dots, \alpha^{C_N}\right] \tag{8.30}$$

Assign pattern $\mathbf{x} = (x1, x2)$ to class C_m.

The number of multiple fuzzy rule tables required for an accurate pattern classification may be quite large. Consequently, a complete set of rules S_{ALL} can be enormous. Meanwhile, the rules in S_{ALL} have different classification abilities, and thus by selecting only rules with high potential for accurate classification, we can dramatically reduce the size of the rule set.

The problem of selecting fuzzy IF-THEN rules can be seen as a combinatorial optimisation problem with two objectives. The first, more important, objective is to maximise the number of correctly classified patterns; the second is to minimise the number of rules (Ishibuchi *et al.*, 1995). Genetic algorithms can be applied to this problem.

In genetic algorithms, each feasible solution is treated as an individual, and thus we need represent a feasible set of fuzzy IF-THEN rules as a chromosome of a fixed length. Each gene in such a chromosome should represent a fuzzy rule in S_{ALL}, and if we define S_{ALL} as:

$$S_{ALL} = 2^2 + 3^3 + 4^4 + 5^5 + 6^6$$

the chromosome can be specified by a 90-bit string. Each bit in this string can assume one of three values: 1, -1 or 0.

Our goal is to establish a compact set of fuzzy rules S by selecting appropriate rules from the complete set of rules S_{ALL}. If a particular rule belongs to set S, the corresponding bit in the chromosome assumes value 1, but if it does not belong to S the bit assumes value -1. **Dummy rules** are represented by zeros.

What is a dummy rule?

A dummy rule is generated when the consequent of this rule cannot be determined. This is normally the case when a corresponding fuzzy subspace has no training patterns. Dummy rules do not affect the performance of a classification system, and thus can be excluded from rule set S.

How do we decide which fuzzy rule belongs to rule set *S* and which does not?

In the initial population, this decision is based on a 50 per cent chance. In other words, each fuzzy rule has a 0.5 probability of receiving value 1 in each chromosome represented in the initial population.

A basic genetic algorithm for selecting fuzzy IF-THEN rules includes the following steps (Ishibuchi *et al.*, 1995):

Step 1: Randomly generate an initial population of chromosomes. The population size may be relatively small, say 10 or 20 chromosomes. Each gene in a chromosome corresponds to a particular fuzzy IF-THEN rule in the rule set defined by S_{ALL}. The genes corresponding to dummy rules receive values 0, and all other genes are randomly assigned either 1 or −1.

Step 2: Calculate the performance, or fitness, of each individual chromosome in the current population.

The problem of selecting fuzzy rules has two objectives: to maximise the accuracy of the pattern classification and to minimise the size of a rule set. The **fitness function** has to accommodate both these objectives. This can be achieved by introducing two respective weights, w_P and w_N, in the fitness function:

$$f(S) = w_P \frac{P_s}{P_{ALL}} - w_N \frac{N_S}{N_{ALL}}, \qquad (8.31)$$

where P_s is the number of patterns classified successfully, P_{ALL} is the total number of patterns presented to the classification system, N_S and N_{ALL} are the numbers of fuzzy IF-THEN rules in set S and set S_{ALL}, respectively.

The classification accuracy is more important than the size of a rule set. This can be reflected by assigning the weights such that,

$$0 < w_N \leqslant w_P$$

Typical values for w_N and w_P are 1 and 10, respectively. Thus, we obtain:

$$f(S) = 10 \frac{P_s}{P_{ALL}} - \frac{N_S}{N_{ALL}} \qquad (8.32)$$

Step 3: Select a pair of chromosomes for mating. Parent chromosomes are selected with a probability associated with their fitness; a better fit chromosome has a higher probability of being selected.

Step 4: Create a pair of offspring chromosomes by applying a standard crossover operator. Parent chromosomes are crossed at the randomly selected crossover point.

Step 5: Perform mutation on each gene of the created offspring. The mutation probability is normally kept quite low, say 0.01. The mutation is done by multiplying the gene value by −1.

Step 6: Place the created offspring chromosomes in the new population.

Step 7: Repeat Step 3 until the size of the new population becomes equal to the size of the initial population, and then replace the initial (parent) population with the new (offspring) population.

Step 9: Go to Step 2, and repeat the process until a specified number of generations (typically several hundreds) is considered.

The above algorithm can dramatically reduce the number of fuzzy IF-THEN rules needed for correct classification. In fact, several computer simulations (Ishibuchi *et al.*, 1995) demonstrate that the number of rules can be cut down to less than 2 per cent of the initially generated set of rules. Such a reduction leaves a fuzzy classification system with relatively few significant rules, which can then be carefully examined by human experts. This allows us to use fuzzy evolutionary systems as a knowledge acquisition tool for discovering new knowledge in complex databases.

8.7 Summary

In this chapter, we considered hybrid intelligent systems as a combination of different intelligent technologies. First we introduced a new breed of expert systems, called neural expert systems, which combine neural networks and rule-based expert systems. Then we considered a neuro-fuzzy system that was functionally equivalent to the Mamdani fuzzy inference model, and an adaptive neuro-fuzzy inference system, ANFIS, equivalent to the Sugeno fuzzy inference model. Finally, we discussed evolutionary neural networks and fuzzy evolutionary systems.

The most important lessons learned in this chapter are:

- Hybrid intelligent systems are systems that combine at least two intelligent technologies; for example, a combination of a neural network and a fuzzy system results in a hybrid neuro-fuzzy system.

- Probabilistic reasoning, fuzzy set theory, neural networks and evolutionary computation form the core of soft computing, an emerging approach to building hybrid intelligent systems capable of reasoning and learning in uncertain and imprecise environments.

- Both expert systems and neural networks attempt to emulate human intelligence, but use different means. While expert systems rely on IF-THEN rules and logical inference, neural networks use parallel data processing. An expert system cannot learn, but can explain its reasoning, while a neural network can learn, but acts as a black-box. These qualities make them good candidates for building a hybrid intelligent system, called a neural or connectionist expert system.

- Neural expert systems use a trained neural network in place of the knowledge base. Unlike conventional rule-based expert systems, neural expert systems

can deal with noisy and incomplete data. Domain knowledge can be utilised in an initial structure of the neural knowledge base. After training, the neural knowledge base can be interpreted as a set of IF-THEN production rules.

- A neuro-fuzzy system corresponding to the Mamdani fuzzy inference model can be represented by a feedforward neural network consisting of five layers: input, fuzzification, fuzzy rule, output membership and defuzzification.

- A neuro-fuzzy system can apply standard learning algorithms developed for neural networks, including the back-propagation algorithm. Expert knowledge in the form of linguistic variables and fuzzy rules can be embodied in the structure of a neuro-fuzzy system. When a representative set of examples is available, a neuro-fuzzy system can automatically transform it into a set of fuzzy IF-THEN rules.

- An adaptive neuro-fuzzy inference system, ANFIS, corresponds to the first-order Sugeno fuzzy model. The ANFIS is represented by a neural network with six layers: input, fuzzification, fuzzy rule, normalisation, defuzzification and summation.

- The ANFIS uses a hybrid learning algorithm that combines the least-squares estimator with the gradient descent method. In the forward pass, a training set of input patterns is presented, neuron outputs are calculated on a layer-by-layer basis, and rule consequent parameters are identified by the least-squares estimator. In the backward pass, the error signals are propagated back and the rule antecedent parameters are updated according to the chain rule.

- Genetic algorithms are effective for optimising weights and selecting the topology of a neural network.

- Evolutionary computation can also be used for selecting an appropriate set of fuzzy rules for solving a complex classification problem. While a complete set of fuzzy IF-THEN rules is generated from numerical data by using multiple fuzzy rule tables, a genetic algorithm is used to select a relatively small number of fuzzy rules with high classification power.

Questions for review

1 What is a hybrid intelligent system? Give an example. What constitutes the core of soft computing? What are the differences between 'hard' and 'soft' computing?

2 Why is a neural expert system capable of approximate reasoning? Draw a neural knowledge base for a three-class classification problem. Suppose that an object to be classified is either an apple, an orange or a lemon.

3 Why are fuzzy systems and neural networks considered to be natural complementary tools for building intelligent systems? Draw a neuro-fuzzy system corresponding to the Sugeno fuzzy inference model for the implementation of the AND operation. Assume that the system has two inputs and one output, and each of them is represented by two linguistic values: small and large.

4 Describe the functions of each layer in a neuro-fuzzy system. How is fuzzification done in this system? How does a fuzzy rule neuron combine its multiple inputs? How is defuzzification done in neuro-fuzzy systems?

5 How does a neuro-fuzzy system learn? What system parameters are learned or tuned during training? How does a neuro-fuzzy system identify false rules given by a human expert? Give an example.

6 Describe the functions of each layer of an ANFIS. What are activation functions used by fuzzification neurons in Jang's model? What is a normalised firing strength of a fuzzy rule?

7 How does an ANFIS learn? Describe a hybrid learning algorithm. What are the advantages of this algorithm?

8 How should we change the ANFIS architecture shown in Figure 8.10 if we want to implement a zero-order Sugeno fuzzy model?

9 What are the differences between a neuro-fuzzy system corresponding to the Mamdani fuzzy inference model and an ANFIS?

10 How is a set of weights of a neural network encoded in a chromosome? Give an example. Describe the genetic operations used to optimise the weights of a neural network.

11 How is a neural network topology encoded in a chromosome? Give an example. Outline the main steps of a basic genetic algorithm for evolving an optimal neural network topology.

12 What is a grid-fuzzy partition? Give an example. Why are multiple fuzzy rule tables needed for a complex pattern classification problem? Describe a genetic algorithm for selecting fuzzy IF-THEN rules.

References

Bäck, T., Fogel, D.B. and Michalewicz, Z., eds (1997). *Handbook of Evolutionary Computation*. Institute of Physics Publishing, Bristol, Philadelphia and Oxford University Press, New York.

Fu, L.M. (1993). Knowledge-based connectionism for revising domain theories, *IEEE Transactions on Systems, Man and Cybernetics*, 23(1), 173–182.

Gallant, S.I. (1988). Connectionist expert systems, *Communications of the ACM*, 31(2), 152–169.

Gallant, S.I. (1993). *Neural Network Learning and Expert Systems*. MIT Press, Cambridge, MA.

Ichikawa, Y. and Sawa, T. (1992). Neural network application for direct feedback controllers, *IEEE Transactions on Neural Networks*, 3(2), 224–231.

Ishibuchi, H., Nozaki, K. and Tanaka, H. (1992). Distributed representation of fuzzy rules and its application to pattern classification, *IEEE Transactions on Fuzzy Systems*, 3(3), 260–270.

Ishibuchi, H., Nozaki, K., Yamamoto, N. and Tanaka, H. (1995). Selecting fuzzy If-Then rules for classification problems using genetic algorithms, *Fuzzy Sets and Systems*, 52, 21–32.

Jang, J.-S.R. (1993). ANFIS: Adaptive Network-based Fuzzy Inference Systems, *IEEE Transactions on Systems, Man and Cybernetics*, 23(3), 665–685.

Jang, J.-S.R., Sun, C.-T. and Mizutani, E. (1997). *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice Hall, Englewood Cliffs, NJ.

Kasabov, N. (1996). *Foundations of Neural Networks, Fuzzy Logic, and Knowledge Engineering*. MIT Press, Cambridge, MA.

Lin, C.T. and Lee, G. (1996). *Neural Fuzzy Systems: A Neuro-Fuzzy Synergism to Intelligent Systems*. Prentice Hall, Englewood Cliffs, NJ.

Miller, G.F., Todd, P.M. and Hedge, S.U. (1989). Designing neural networks using genetic algorithms, *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer, ed., Morgan Kaufmann, San Mateo, CA, pp. 379–384.

Montana, D.J. and Davis, L. (1989). Training feedforward networks using genetic algorithms, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Mateo, CA, pp. 762–767.

Nauck, D., Klawonn, F. and Kruse, R. (1997). *Foundations of Neuro-Fuzzy Systems*. John Wiley, Chichester.

Nikolopoulos, C. (1997). *Expert Systems: Introduction to First and Second Generation and Hybrid Knowledge Based Systems*. Marcel Dekker, Inc., New York.

Russell, S.J. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach,* 2nd edn. Prentice Hall, Englewood Cliffs, NJ.

Schaffer, J.D., Whitley, D. and Eshelman, L.J. (1992). Combinations of genetic algorithms and neural networks: a survey of the state of the art, *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks, COGANN-92*, D. Whitley and J.D. Schaffer, eds, IEEE Computer Society Press, Baltimore, MD, pp. 1–37.

Sestito, S. and Dillon T. (1991). Using single layered neural networks for the extraction of conjunctive rules, *Journal of Applied Intelligence*, no. 1, 157–173.

Von Altrock, C. (1997). *Fuzzy Logic and NeuroFuzzy Applications in Business and Finance*. Prentice Hall, Upper Saddle River, NJ.

Whitley, D. and Hanson, T. (1989). Optimizing neural networks using faster, more accurate genetic search, *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer, ed., Morgan Kaufmann, San Mateo, CA, pp. 391–396.

Zadeh, L. (1996). Computing with words – A paradigm shift, *Proceedings of the First International Conference on Fuzzy Logic and Management of Complexity*, Sydney, Australia, 15–18 January, vol. 1, pp. 3–10.

Knowledge engineering and data mining

9

In which we discuss how to pick the right tool for the job, build an intelligent system and turn data into knowledge.

9.1 Introduction, or what is knowledge engineering?

Choosing the right tool for the job is undoubtedly the most critical part of building an intelligent system. Having read this far, you are now familiar with rule- and frame-based expert systems, fuzzy systems, artificial neural networks, genetic algorithms, and hybrid neuro-fuzzy and fuzzy evolutionary systems. Although several of these tools handle many problems well, selecting the one best suited to a particular problem can be difficult. Davis's law states: 'For every tool there is a task perfectly suited to it' (Davis and King, 1977). However, it would be too optimistic to assume that for every task there is a tool perfectly suited to it. In this chapter, we suggest basic guidelines for selecting an appropriate tool for a given task, consider the main steps in building an intelligent system and discuss how to turn data into knowledge.

The process of building an intelligent system begins with gaining an understanding of the problem domain. We first must assess the problem and determine what data are available and what is needed to solve the problem. Once the problem is understood, we can choose an appropriate tool and develop the system with this tool. The process of building intelligent knowledge-based systems is called **knowledge engineering**. It has six basic phases (Waterman, 1986; Durkin, 1994):

1 Problem assessment

2 Data and knowledge acquisition

3 Development of a prototype system

4 Development of a complete system

5 Evaluation and revision of the system

6 Integration and maintenance of the system

Phase 1: Problem assessment

- Determine the problem's characteristics
- Identify the main participants in the project
- Specify the project's objectives
- Determine the resources needed for building the system

Phase 2: Data and knowledge acquisition

- Collect and analyse data and knowledge
- Make key concepts of the system design more explicit

Phase 3: Development of a prototype system

- Choose a tool for building an intelligent system
- Transform data and represent knowledge
- Design and implement a prototype system
- Test the prototype with test cases

Phase 4: Development of a complete system

- Prepare a detailed design for a full-scale system
- Collect additional data and knowledge
- Develop the user interface
- Implement the complete system

Phase 5: Evaluation and revision of the system

- Evaluate the system against the performance criteria
- Revise the system as necessary

Phase 6: Integration and maintenance of the system

- Make arrangements for technology transfer
- Establish an effective maintenance program

Figure 9.1   The process of knowledge engineering

The process of knowledge engineering is illustrated in Figure 9.1. Knowledge engineering, despite its name, is still more art than engineering, and a real process of developing an intelligent system is not as neat and clean as Figure 9.1 might suggest. Although the phases are shown in sequence, they usually overlap considerably. The process itself is highly iterative, and at any time we may engage in any development activities. Let us now examine each phase in more detail.

9.1.1 Problem assessment

During this phase we determine the problem's characteristics, identify the project's participants, specify the project's objectives and determine what resources are needed for building the system.

To characterise the problem, we need to determine the problem type, input and output variables and their interactions, and the form and content of the solution.

The first step is to determine the problem type. Typical problems often addressed by intelligent systems are illustrated in Table 9.1. They include **diagnosis**, **selection**, **prediction**, **classification**, **clustering**, **optimisation** and **control**.

The problem type influences our choice of the tool for building an intelligent system. Suppose, for example, we develop a system to detect faults in an electric circuit and guide the user through the diagnostic process. This problem clearly belongs to *diagnosis*. Domain knowledge in such problems can often be represented by production rules, and thus a rule-based expert system might be the right candidate for the job.

Of course, the choice of a building tool also depends on the form and content of the solution. For example, systems that are built for diagnostic tasks usually need explanation facilities – the means that enable them to justify their solutions. Such facilities are an essential component of any expert system, but are not available in neural networks. On the other hand, a neural network might be a good choice for *classification* and *clustering* problems where the results are often more important than understanding the system's reasoning process.

The next step in the problem assessment is to identify the participants in the project. Two critical participants in any knowledge engineering project are

Table 9.1 Typical problems addressed by intelligent systems

Problem type	Description
Diagnosis	Inferring malfunctions of an object from its behaviour and recommending solutions.
Selection	Recommending the best option from a list of possible alternatives.
Prediction	Predicting the future behaviour of an object from its behaviour in the past.
Classification	Assigning an object to one of the defined classes.
Clustering	Dividing a heterogeneous group of objects into homogeneous subgroups.
Optimisation	Improving the quality of solutions until an optimal one is found.
Control	Governing the behaviour of an object to meet specified requirements in real-time.

the knowledge engineer (a person capable of designing, building and testing an intelligent system) and the domain expert (a knowledgeable person capable of solving problems in a specific area or domain).

Then we specify the project's objectives, such as gaining a competitive edge, improving the quality of decisions, reducing labour costs, and improving the quality of products and services.

Finally, we determine what resources are needed for building the system. They normally include computer facilities, development software, knowledge and data sources (human experts, textbooks, manuals, web sites, databases and examples) and, of course, money.

9.1.2 Data and knowledge acquisition

During this phase we obtain further understanding of the problem domain by collecting and analysing both data and knowledge, and making key concepts of the system's design more explicit.

Data for intelligent systems are often collected from different sources, and thus can be of different types. However, a particular tool for building an intelligent system requires a particular type of data. Some tools deal with continuous variables, while others need to have all variables divided into several ranges, or to be normalised to a single range, say from 0 to 1. Some handle symbolic (textual) data, while others use only numerical data. Some tolerate imprecise and noisy data, while others require only well-defined, clean data. As a result, the data must be transformed, or **massaged**, into the form useful for a particular tool. However, no matter which tool we choose, there are three important issues that must be resolved before massaging the data (Berry and Linoff, 1997).

The first issue is **incompatible** data. Often the data we want to analyse store text in EBCDIC coding and numbers in packed decimal format, while the tools we want to use for building intelligent systems store text in the ASCII code and numbers as integers with a single- or double-precision floating point. This issue is normally resolved with data transport tools that automatically produce the code for the required data transformation.

The second issue is **inconsistent** data. Often the same facts are represented differently in different databases. If these differences are not spotted and resolved in time, we might find ourselves, for example, analysing consumption patterns of carbonated drinks using data that do not include Coca-Cola just because they were stored in a separate database.

The third issue is **missing** data. Actual data records often contain blank fields. Sometimes we might throw such incomplete records away, but normally we would attempt to infer some useful information from them. In many cases, we can simply fill the blank fields in with the most common or average values. In other cases, the fact that a particular field has not been filled in might itself provide us with very useful information. For example, in a job application form, a blank field for a business phone number might suggest that an applicant is currently unemployed.

Our choice of the system building tool depends on the acquired data. As an example, we can consider a problem of estimating the market value of a property based on its features. This problem can be handled by both expert system and neural network technologies. Therefore, before deciding which tool to apply, we should investigate the available data. If, for instance, we can obtain recent sale prices for houses throughout the region, we might train a neural network by using examples of previous sales rather than develop an expert system using knowledge of an experienced appraiser.

The task of data acquisition is closely related to the task of knowledge acquisition. In fact, we acquire some knowledge about the problem domain while collecting the data.

What are the stages in the knowledge acquisition process?

Usually we start with reviewing documents and reading books, papers and manuals related to the problem domain. Once we become familiar with the problem, we can collect further knowledge through interviewing the domain expert. Then we study and analyse the acquired knowledge, and repeat the entire process again. Knowledge acquisition is an inherently iterative process.

During a number of interviews, the expert is asked to identify four or five typical cases, describe how he or she solves each case and explain, or 'think out loud', the reasoning behind each solution (Russell and Norvig, 2002). However, extracting knowledge from a human expert is a difficult process – it is often called the 'knowledge acquisition bottleneck'. Quite often experts are unaware of what knowledge they have and the problem-solving strategy they use, or are unable to verbalise it. Experts may also provide us with irrelevant, incomplete or inconsistent information.

Understanding the problem domain is critical for building intelligent systems. A classical example is given by Donald Michie (1982). A cheese factory had a very experienced cheese-tester who was approaching retirement age. The factory manager decided to replace him with an 'intelligent machine'. The human tester tested the cheese by sticking his finger into a sample and deciding if it 'felt right'. So it was assumed the machine had to do the same – test for the right surface tension. But the machine was useless. Eventually, it turned out that the human tester subconsciously relied on the cheese's smell rather than on its surface tension and used his finger just to break the crust and let the aroma out.

The data and knowledge acquired during the second phase of knowledge engineering should enable us to describe the problem-solving strategy at the most abstract, conceptual, level and choose a tool for building a prototype. However, we **must not** make a detailed analysis of the problem before evaluating the prototype.

9.1.3 Development of a prototype system

This actually involves creating an intelligent system – or, rather, a small version of it – and testing it with a number of test cases.

What is a prototype?

A prototype system can be defined as a small version of the final system. It is designed to test how well we understand the problem, or in other words to make sure that the problem-solving strategy, the tool selected for building a system, and techniques for representing acquired data and knowledge are adequate to the task. It also provides us with an opportunity to persuade the sceptics and, in many cases, to actively engage the domain expert in the system's development.

After choosing a tool, massaging the data and representing the acquired knowledge in the form suitable for that tool, we design and then implement a prototype version of the system. Once it is built, we examine (usually together with the domain expert) the prototype's performance by testing it with a variety of test cases. The domain expert takes an active part in testing the system, and as a result becomes more involved in the system's development.

What is a test case?

A test case is a problem successfully solved in the past for which input data and an output solution are known. During testing, the system is presented with the same input data and its solution is compared with the original solution.

What should we do if we have made a bad choice of the system-building tool?

We should throw the prototype away and start the prototyping phase over again – any attempt to force an ill-chosen tool to suit a problem it wasn't designed for would only lead to further delays in the system's development. The main goal of the prototyping phase is to obtain a better understanding of the problem, and thus by starting this phase with a new tool, we waste neither time nor money.

9.1.4 Development of a complete system

As soon as the prototype begins functioning satisfactorily, we can assess what is actually involved in developing a full-scale system. We develop a plan, schedule and budget for the complete system, and also clearly define the system's performance criteria.

The main work at this phase is often associated with adding data and knowledge to the system. If, for example, we develop a diagnostic system, we might need to provide it with more rules for handling specific cases. If we develop a prediction system, we might need to collect additional historical examples to make predictions more accurate.

The next task is to develop the user interface – the means of delivering information to a user. The user interface should make it easy for users to obtain any details they need. Some systems may be required to explain its reasoning process and justify its advice, analysis or conclusion, while others need to represent their results in a graphical form.

The development of an intelligent system is, in fact, an evolutionary process. As the project proceeds and new data and knowledge are collected and added to

the system, its capability improves and the prototype gradually evolves into a final system.

9.1.5 Evaluation and revision of the system

Intelligent systems, unlike conventional computer programs, are designed to solve problems that quite often do not have clearly defined 'right' and 'wrong' solutions. To evaluate an intelligent system is, in fact, to assure that the system performs the intended task to the user's satisfaction. A formal evaluation of the system is normally accomplished with the test cases selected by the user. The system's performance is compared against the performance criteria that were agreed upon at the end of the prototyping phase.

The evaluation often reveals the system's limitations and weaknesses, so it is revised and relevant development phases are repeated.

9.1.6 Integration and maintenance of the system

This is the final phase in developing the system. It involves integrating the system into the environment where it will operate and establishing an effective maintenance program.

By 'integrating' we mean interfacing a new intelligent system with existing systems within an organisation and arranging for technology transfer. We must make sure that the user knows how to use and maintain the system. Intelligent systems are knowledge-based systems, and because knowledge evolves over time, we need to be able to modify the system.

But who maintains the system?
Once the system is integrated in the working environment, the knowledge engineer withdraws from the project. This leaves the system in the hands of its users. Thus, the organisation that uses the system should have in-house expertise to maintain and modify the system.

Which tool should we use?
As must be clear by now, there is no single tool that is applicable to all tasks. Expert systems, neural networks, fuzzy systems and genetic algorithms all have a place and all find numerous applications. Only two decades ago, in order to apply an intelligent system (or, rather, an expert system), one had first to find a 'good' problem, a problem that had some chance for success. Knowledge engineering projects were expensive, laborious and had high investment risks. The cost of developing a moderate-sized expert system was typically between $250,000 and $500,000 (Simon, 1987). Such 'classic' expert systems as DENDRAL and MYCIN took 20 to 40 person-years to complete. Fortunately, the last few years have seen a dramatic change in the situation. Today, most intelligent systems are built within months rather than years. We use commercially available expert system shells, fuzzy, neural network and evolutionary computation toolboxes, and run our applications on standard PCs. And most

importantly, adopting new intelligent technologies is becoming problem-driven, rather than curiosity-driven as it often was in the past. Nowadays an organisation addresses its problems with appropriate intelligent tools.

In the following sections, we discuss applications of different tools for solving specific problems.

9.2 Will an expert system work for my problem?

Case study 1: Diagnostic expert systems

I want to develop an intelligent system that can help me to fix malfunctions of my Mac computer. Will an expert system work for this problem?

There is an old but still useful test for prime candidates for expert systems. It is called the *Phone Call Rule* (Firebaugh, 1988): 'Any problem that can be solved by your in-house expert in a 10–30 minute phone call can be developed as an expert system'.

Diagnosis and troubleshooting problems (of course, computer diagnosis is one of them) have always been very attractive candidates for expert system technology. As you may recall, medical diagnosis was one of the first areas to which expert systems were applied. Since then, diagnostic expert systems have found numerous applications, particularly in engineering and manufacturing.

Diagnostic expert systems are relatively easy to develop – most diagnostic problems have a finite list of possible solutions, involve a rather limited amount of well-formalised knowledge, and often take a human expert a short time (say, an hour) to solve.

To develop a computer diagnostic system, we need to acquire knowledge about troubleshooting in computers. We might find and interview a hardware specialist, but for a small expert system there is a better alternative – to use a troubleshooting manual. It provides step-by-step procedures for detecting and fixing a variety of faults. In fact, such a manual contains knowledge in the most concise form that can be directly used in an expert system. There is no need to interview an expert, and thus we can avoid the 'knowledge acquisition bottle-neck'.

Computer manuals often include troubleshooting sections, which consider possible problems with the system start-up, computer/peripherals (hard disk, keyboard, monitor, printer), disk drives (floppy disk, CD-ROM), files, and network and file sharing. In our example, we will consider only troubleshooting the Mac system start-up. However, once the prototype expert system is developed, you can easily expand it.

Figure 9.2 illustrates the troubleshooting procedure for the Macintosh computer. As you can see, troubleshooting here is carried out through a series of visual inspections, or tests. We first collect some initial information (the system does not start), infer from it whatever can be inferred, gather additional information (power cords are OK, Powerstrip is OK, etc.) and finally identify

Section 1: System start-up	
Problem	**Action**
1.1. System does not start	• Check power cords (both ends). • Check Powerstrip. • Check screen brightness. • Check phonet connectors. • Check keyboard connectors. • Check pin connectors.
1.2. System starts and then freezes	• Restart the Mac. • Unhook all SCSI devices and restart the Mac. • Restart with the shift key down (turns off extensions). • Remove all extensions and add them back one at a time. Restart the Mac after each addition. If using System 7.5 or higher, use Extensions Manager.
1.3. System starts with a question mark	• Restart the Mac. • Unhook all SCSI devices and restart the Mac. • Restart with the shift key down (turns off extensions). • Zap the PRAM. If this works, turn virtual memory and the disk cache on one at a time (restarting the computer each time). Try making the disk cache smaller, or reduce virtual memory. • Start with Disk Tools and if the hard drive icon shows up, see if there are two system suitcases and/or two finders. If the hard drive icon does not show up, call AV repair. • Run Disk First Aid, MacCheck, or Norton's Disk Doctor. • Do a clean reinstall of the system.
1.4. System starts with a Sad Mac.	• Start with Disk Tools and do a clean reinstall of the system. • Run Disk First Aid, MacCheck, or Norton's Disk Doctor. • Start with Disk Tools, and if the hard drive icon does not show up, call AV repair.
1.5. System starts with wrong 'music'	• Check cables.

Figure 9.2   Troubleshooting the system start-up for Macintosh computers

the cause of the system's malfunction. This is essentially data-driven reasoning, which can be best realised with the forward-chaining inference technique. The expert system should first ask the user to select a particular task, and once the task is selected, the system should direct troubleshooting by asking the user for additional information until the fault is found.

Let us develop a general rule structure. In each rule, we need to include a clause that identifies the current task. Since our prototype is limited to the

Mac system start-up, the first clause of all rules will identify this task. For example,

Rule: 1
if   task is 'system start-up'
then   ask problem

Rule: 2
if   task is 'system start-up'
and   problem is 'system does not start'
then   ask 'test power cords'

Rule: 3
if   task is 'system start-up'
and   problem is 'system does not start'
and   'test power cords' is ok
then   ask 'test Powerstrip'

All the other rules will follow this structure. A set of rules to direct troubleshooting when the Mac system does not start (in Leonardo code) is shown in Figure 9.3.

Now we are ready to build a prototype, or in other words to implement the initial set of rules using an expert system development tool.

How do we choose an expert system development tool?

In general, we should match the features of the problem with the capabilities of the tool. These tools range from high-level programming languages such as LISP, PROLOG, OPS, C and Java, to expert system shells. High-level programming languages offer a greater flexibility and can enable us to meet any project requirements, but they do require high-level programming skills. On the other hand, shells, although they do not have the flexibility of programming languages, provide us with the built-in inference engine, explanation facilities and the user interface. We do not need any programming skills to use a shell – we just enter rules in English in the shell's knowledge base. This makes shells particularly useful for rapid prototyping.

So how do we choose a shell?

The Appendix provides some details of a few commercial expert systems shells currently available on the market. This can help you to choose an appropriate tool; however the internet is rapidly becoming the most valuable source of information. Many vendors have Web sites, and you can even try and evaluate their products over the Web.

In general, when selecting an expert system shell, you should consider how the shell represents knowledge (rules or frames), what inference mechanism it uses (forward or backward chaining), whether the shell supports inexact reasoning and if so what technique it uses (Bayesian reasoning, certainty factors or fuzzy logic), whether the shell has an 'open' architecture allowing access to

```
/* Mac Troubleshooting Expert System

ask task

/*************************************************
/* Section 1. System Start-up
/*************************************************


Rule: 1
if     task is 'system start-up'
then ask problem

/*************************************************
/* Section 1.1. System does not start
/*************************************************


Rule: 1.1.
if     task is 'system start-up'
and  problem is 'system does not start'
then ask 'test power cords'

Rule: 1.2
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test power cords' is ok
then ask 'test Powerstrip'

Rule: 1.3
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test power cords' is not ok
then troubleshooting is done

Rule: 1.4
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test Powerstrip' is ok
then ask 'test screen brightness'

Rule: 1.5
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test Powerstrip' is not ok
then troubleshooting is done
```

```
Rule: 1.6
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test screen brightness' is ok
then ask 'test phonet connectors'

Rule: 1.7
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test screen brightness' is not ok
then troubleshooting is done

Rule: 1.8
if     task is 'system start-up'
and  problem is 'system does not start'
and  test 'phonet connectors' is ok
then ask 'test keyboard connectors'

Rule: 1.9
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test phonet connectors' is not ok
then troubleshooting is done

Rule: 1.10
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test keyboard connectors' is ok
then ask 'test pin connectors'

Rule: 1.11
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test keyboard connectors' is not ok
then troubleshooting is done

Rule: 1.12
if     task is 'system start-up'
and  problem is 'system does not start'
and  'test pins connectors' is ok
then troubleshooting is 'Call AV Repair'

/*****************************************
/* The SEEK directive sets up the goal

seek troubleshooting
```

Figure 9.3   Rules for a prototype of the Mac troubleshooting expert system

external data files and programs, and how the user will interact with the expert system (graphical user interface, hypertext).

Today you can buy an expert system shell for less than $500 and run it on your PC or Mac. You can also obtain an expert system shell for free (for example, CLIPS). However, you should clearly understand your licence obligations, especially whether you need to have a distribution licence allowing the end-user to use your expert system once it is developed.

An important area for consideration in choosing a tool is the stability of the company supplying the tool.

What are indicators of a company's stability?

There are several important indicators, such as the year founded, number of employees, total gross income, gross income from intelligent systems products, and number of products sold. Similar indicators represent the stability of a particular product. When was the product officially released? How many versions have been released? How many installations have been made? These are important questions for determining the development stage of the product.

However, probably the best method for evaluating both products and vendors is to obtain a list of users, successful applications and installation sites. Just a few minutes on the phone with the tool's user brings to light the strengths and weaknesses of the product and its supplier.

Case study 2: Classification expert systems

I want to develop an intelligent system that can help me to identify different classes of sail boats. Will an expert system work for this problem?

This is a typical classification problem (to identify a boat means to assign it to one of the defined classes) and, as we discussed earlier, such problems can be handled well by both expert systems and neural networks. If you decide to build an expert system, you should start with collecting some information about mast structures and sail plans of different sailing vessels. As an example, consider Figure 9.4, which shows eight classes of sail boats. Each boat can be uniquely identified by its sail plans.
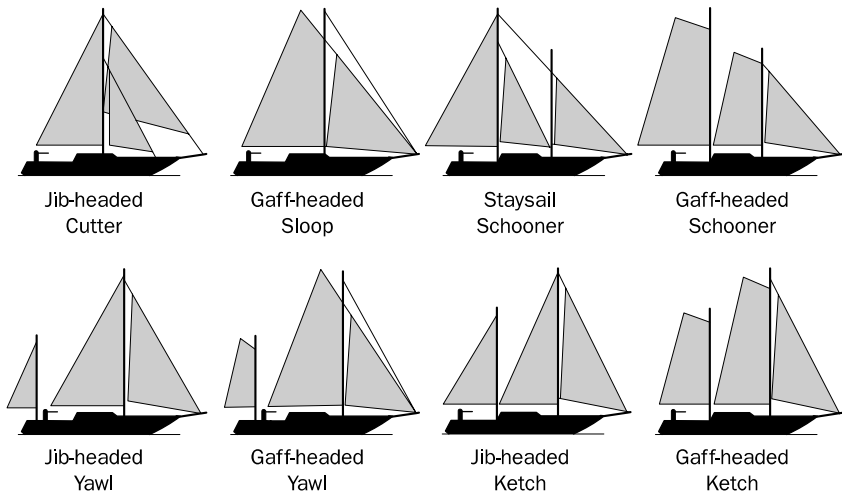
Figure 9.4   Eight classes of sailing vessels

```
             /* Sailing Vessel Classification Expert System: Mark 1
Rule: 1      if    'the number of masts' is one
             and   'the shape of the mainsail' is triangular
             then  boat is 'Jib-headed Cutter'
Rule: 2      if    'the number of masts' is one
             and   'the shape of the mainsail' is quadrilateral
             then  boat is 'Gaff-headed Sloop'
Rule: 3      if    'the number of masts' is two
             and   'the main mast position' is 'forward of the short mast'
             and   'the short mast position' is 'forward of the helm'
             and   'the shape of the mainsail' is triangular
             then  boat is 'Jib-headed Ketch'
Rule: 4      if    'the number of masts' is two
             and   'the main mast position' is 'forward of the short mast'
             and   'the short mast position' is 'forward of the helm'
             and   'the shape of the mainsail' is quadrilateral
             then  boat is 'Gaff-headed Ketch'
Rule: 5      if    'the number of masts' is two
             and   'the main mast position' is 'forward of the short mast'
             and   'the short mast position' is 'aft the helm'
             and   'the shape of the mainsail' is triangular
             then  boat is 'Jib-headed Yawl'
Rule: 6      if    'the number of masts' is two
             and   'the main mast position' is 'forward of the short mast'
             and   'the short mast position' is 'aft the helm'
             and   'the shape of the mainsail' is quadrilateral
             then  boat is 'Gaff-headed Yawl'
Rule: 7      if    'the number of masts' is two
             and   'the main mast position' is 'aft the short mast'
             and   'the shape of the mainsail' is quadrilateral
             then  boat is 'Gaff-headed Schooner'
Rule: 8      if    'the number of masts' is two
             and   'the main mast position' is 'aft the short mast'
             and   'the shape of the mainsail' is 'triangular with two foresails'
             then  boat is 'Staysail Schooner'
             /*********************************************************************
             /* The SEEK directive sets up the goal
             seek boat
```

Figure 9.5   Rules for the boat classification expert system

A set of rules (in Leonardo code) for the sailing vessel classification is shown in Figure 9.5. During a dialogue session with the user, the system obtains the number and position of masts on the unknown vessel as well as the shape of its mainsail, and then uniquely identifies each of the eight boats shown in Figure 9.4.

No doubt when the sky is blue and the sea is calm, this system will help us to identify a sail boat. But this is not always the case. On a rough sea or in foggy conditions, it is difficult, or even impossible, to see clearly the position of masts and the shape of the mainsail. Despite the fact that solving real-world classification problems often involves inexact and incomplete data such as these, we still can use the expert system approach. However, we need to deal with uncertainties. Let us apply the certainty factors theory to our problem. This

```
          /* Sailing Vessel Classification Expert System: Mark 2
          control cf
Rule: 1   if     'the number of masts' is one
          then   boat is 'Jib-headed Cutter'      {cf 0.4};
                 boat is 'Gaff-headed Sloop'      {cf 0.4}

Rule: 2   if     'the number of masts' is one
          and    'the shape of the mainsail' is triangular
          then   boat is 'Jib-headed Cutter'      {cf 1.0}

Rule: 3   if     'the number of masts' is one
          and    'the shape of the mainsail' is quadrilateral
          then   boat is 'Gaff-headed Sloop'      {cf 1.0}

Rule: 4   if     'the number of masts' is two
          then   boat is 'Jib-headed Ketch'       {cf 0.1};
                 boat is 'Gaff-headed Ketch'      {cf 0.1};
                 boat is 'Jib-headed Yawl'        {cf 0.1};
                 boat is'Gaff-headed Yawl'        {cf 0.1};
                 boat is 'Gaff-headed Schooner'   {cf 0.1};
                 boat is ' Staysail Schooner'     {cf 0.1}

Rule: 5   if     'the number of masts' is two
          and    'the main mast position' is 'forward of the short mast'
          then   boat is 'Jib-headed Ketch'       {cf 0.2};
                 boat is'Gaff-headed Ketch'       {cf 0.2};
                 boat is 'Jib-headed Yawl'        {cf 0.2};
                 boat is 'Gaff-headed Yawl'       {cf 0.2}

Rule: 6   if     'the number of masts' is two
          and    'the main mast position' is 'aft the short mast'
          then   boat is 'Gaff-headed Schooner' {cf 0.4};
                 boat is 'Staysail Schooner'      {cf 0.4}

Rule: 7   if     'the number of masts' is two
          and    'the short mast position' is 'forward of the helm'
          then   boat is 'Jib-headed Ketch'       {cf 0.4};
                 boat is 'Gaff-headed Ketch'      {cf 0.4}

Rule: 8   if     'the number of masts' is two
          and    'the short mast position' is 'aft the helm'
          then   boat is 'Jib-headed Yawl'        {cf 0.2};
                 boat is 'Gaff-headed Yawl'       {cf 0.2};
                 boat is 'Gaff-headed Schooner' {cf 0.2};
                 boat is 'Staysail Schooner'      {cf 0.2}

Rule: 9   if     'the number of masts' is two
          and    'the shape of the mainsail' is triangular
          then   boat is 'Jib-headed Ketch'       {cf 0.4};
                 boat is 'Jib-headed Yawl'        {cf 0.4}

Rule: 10  if     'the number of masts' is two
          and    'the shape of the mainsail' is quadrilateral
          then   boat is 'Gaff-headed Ketch'      {cf 0.3};
                 boat is 'Gaff-headed Yawl'       {cf 0.3};
                 boat is 'Gaff-headed Schooner' {cf 0.3}

Rule: 11  if     'the number of masts' is two
          and    'the shape of the mainsail' is 'triangular with two foresails'
          then   boat is 'Staysail Schooner'      {cf 1.0}

          seek boat
```

Figure 9.6   Uncertainty management in the boat classification expert system

theory, as you may recall, can manage incrementally acquired evidence, as well as information with different degrees of belief.

Figure 9.6 shows a complete set of rules for solving the sailing vessel classification problem with certainty factors. The expert system is required to classify a boat, or in other words to establish certainty factors for a multivalued object *boat*. To apply the evidential reasoning technique, the expert system prompts the user to input not only the object value but also the certainty associated with this value. For example, using the Leonardo scale from 0 to 1, we might obtain the following dialogue (the user's answers are indicated by arrows; also note the propagation of certainty factors through the set of rules):

What is the number of masts?
⇒ **two**

To what degree do you believe that the number of masts is two? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.9**

Rule: 4
if   'the number of masts' is two
then   boat is 'Jib-headed Ketch'   {cf 0.1};
       boat is 'Gaff-headed Ketch'   {cf 0.1};
       boat is 'Jib-headed Yawl'   {cf 0.1};
       boat is 'Gaff-headed Yawl'   {cf 0.1};
       boat is 'Gaff-headed Schooner' {cf 0.1};
       boat is 'Staysail Schooner'   {cf 0.1}

cf (boat is 'Jib-headed Ketch') = cf ('number of masts' is two) $\times$ 0.1 = 0.9 $\times$ 0.1 = 0.09
cf (boat is 'Gaff-headed Ketch') = 0.9 $\times$ 0.1 = 0.09
cf (boat is 'Jib-headed Yawl') = 0.9 $\times$ 0.1 = 0.09
cf (boat is 'Gaff-headed Yawl') = 0.9 $\times$ 0.1 = 0.09
cf (boat is 'Gaff-headed Schooner') = 0.9 $\times$ 0.1 = 0.09
cf (boat is 'Staysail Schooner') = 0.9 $\times$ 0.1 = 0.09

boat is   Jib-headed Ketch   {cf 0.09}
          Gaff-headed Ketch   {cf 0.09}
          Jib-headed Yawl   {cf 0.09}
          Gaff-headed Yawl   {cf 0.09}
          Gaff-headed Schooner   {cf 0.09}
          Staysail Schooner   {cf 0.09}

What is the position of the main mast?
⇒ **aft the short mast**

To what degree do you believe that the main mast position is aft the short mast? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.7**

Rule: 6
if   'the number of masts' is two
and   'the main mast position' is 'aft the short mast'
then   boat is 'Gaff-headed Schooner' {cf 0.4};
       boat is 'Staysail Schooner'   {cf 0.4}

$cf_{Rule:6}$ (boat is 'Gaff-headed Schooner') = min [cf ('the number of masts' is two), cf ('the main mast position' is 'aft the short mast')] × 0.4
= min [0.9, 0.7] × 0.4 = 0.28

$cf_{Rule:6}$ (boat is 'Staysail Schooner') = min [0.9, 0.7] × 0.4 = 0.28

cf (boat is 'Gaff-headed Schooner') = $cf_{Rule:4}$ + $cf_{Rule:6}$ × (1 − $cf_{Rule:4}$)
= 0.09 + 0.28 × (1 − 0.09) = 0.34

cf (boat is 'Staysail Schooner') = 0.09 + 0.28 × (1 − 0.09) = 0.34

boat is		
Gaff-headed Schooner	{cf 0.34}	
Staysail Schooner	{cf 0.34}	
Jib-headed Ketch	{cf 0.09}	
Gaff-headed Ketch	{cf 0.09}	
Jib-headed Yawl	{cf 0.09}	
Gaff-headed Yawl	{cf 0.09}	

What is the position of the short mast?
⇒ **forward of the helm**

To what degree do you believe that the short mast position is forward of the helm? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.6**

Rule: 7
if   'the number of masts' is two
and   'the short mast position' is 'forward of the helm'
then   boat is 'Jib-headed Ketch'   {cf 0.4};
        boat is 'Gaff-headed Ketch'   {cf 0.4}

$cf_{Rule:7}$ (boat is 'Jib-headed Ketch') = min [cf ('the number of masts' is two), cf ('the short mast position' is 'forward of the helm')] × 0.4
= min [0.9, 0.6] × 0.4 = 0.24

$cf_{Rule:7}$ (boat is 'Gaff-headed Ketch') = min [0.9, 0.6] × 0.4 = 0.24

cf (boat is 'Jib-headed Ketch') = $cf_{Rule:6}$ + $cf_{Rule:7}$ × (1 − $cf_{Rule:6}$)
= 0.09 + 0.24 × (1 − 0.09) = 0.30

cf (boat is 'Gaff-headed Ketch') = 0.09 + 0.24 × (1 − 0.09) = 0.30

boat is		
Gaff-headed Schooner	{cf 0.34}	
Staysail Schooner	{cf 0.34}	
Jib-headed Ketch	{cf 0.30}	
Gaff-headed Ketch	{cf 0.30}	
Jib-headed Yawl	{cf 0.09}	
Gaff-headed Yawl	{cf 0.09}	

What is the shape of the mainsail?
⇒ **triangular**

To what degree do you believe that the shape of the mainsail is triangular? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.8**

Rule: 9
if   'the number of masts' is two
and   'the shape of the mainsail' is triangular
then   boat is 'Jib-headed Ketch'   {cf 0.4};
         boat is 'Jib-headed Yawl'   {cf 0.4}

$cf_{Rule:9}$ (boat is 'Jib-headed Ketch') = min [cf ('the number of masts' is two),
cf ('the shape of the mainsail' is triangular)] $\times$ 0.4
= min [0.9, 0.8] $\times$ 0.4 = 0.32

$cf_{Rule:9}$ (boat is 'Jib-headed Yawl') = min [0.9, 0.8] $\times$ 0.4 = 0.32

cf (boat is 'Jib-headed Ketch') = $cf_{Rule:7}$ + $cf_{Rule:9}$ $\times$ (1 $-$ $cf_{Rule:7}$)
= 0.30 + 0.32 $\times$ (1 $-$ 0.30) = 0.52

cf (boat is 'Jib-headed Yawl') = 0.09 + 0.32 $\times$ (1 $-$ 0.09) = 0.38

boat is   Jib-headed Ketch   {cf 0.52}
            Jib-headed Yawl   {cf 0.38}
            Gaff-headed Schooner   {cf 0.34}
            Staysail Schooner   {cf 0.34}
            Gaff-headed Ketch   {cf 0.30}
            Gaff-headed Yawl   {cf 0.09}

Now we can conclude that the boat is probably a *Jib-headed Ketch* and almost certainly not a *Gaff-headed Ketch* or *Gaff-headed Yawl*.

9.3   Will a fuzzy expert system work for my problem?

We need to decide which problem is a good candidate for fuzzy technology. The basic approach here is simple: if you cannot define a set of exact rules for each possible situation, then use fuzzy logic. While certainty factors and Bayesian probabilities are concerned with the imprecision associated with the outcome of a well-defined event, fuzzy logic concentrates on the imprecision of the event itself. In other words, inherently imprecise properties of the problem make it a good candidate for fuzzy technology.

Fuzzy systems are particularly well suited for modelling human decision making. We often rely on common sense and use vague and ambiguous terms while making important decisions. Doctors, for example, do not have a precise threshold in mind when they decide whether a patient in a post-operative recovery area should be sent to a general hospital floor. Although hypothermia is a significant concern after surgery and the patient's body temperature often plays a vital role in the doctor's decision, such factors as the stability of the patient's blood pressure, and his or her perceived comfort at discharge are also taken into account. A doctor makes an accurate assessment not from the precision of a single parameter (say, a body temperature), but rather from evaluating several parameters, some of which are expressed in ambiguous terms (for instance, the patient's willingness to leave the post-operative recovery unit).

Although, most fuzzy technology applications are still reported in control and engineering, an even larger potential exists in business and finance (Von Altrock, 1997). Decisions in these areas are often based on human intuition, common sense and experience, rather than on the availability and precision of data. Decision-making in business and finance is too complex and too uncertain to lend itself to precise analytical methods. Fuzzy technology provides us with a means of coping with the 'soft criteria' and 'fuzzy data' that are often used in business and finance.

Case study 3: Decision-support fuzzy systems

I want to develop an intelligent system for assessing mortgage applications. Will a fuzzy expert system work for this problem?

Mortgage application assessment is a typical problem to which decision-support fuzzy systems can be successfully applied (Von Altrock, 1997).

To develop a decision-support fuzzy system for this problem, we first represent the basic concept of mortgage application assessment in fuzzy terms, then implement this concept in a prototype system using an appropriate fuzzy tool, and finally test and optimise the system with selected test cases.

Assessment of a mortgage application is normally based on evaluating the market value and location of the house, the applicant's assets and income, and the repayment plan, which is decided by the applicant's income and bank's interest charges.

Where do membership functions and rules for mortgage loan assessment come from?

To define membership functions and construct fuzzy rules, we usually need the help of experienced mortgage advisors and also bank managers, who develop the mortgage granting policies. Figures 9.7 to 9.14 show fuzzy sets for linguistic variables used in our problem. Triangular and trapezoidal membership functions can adequately represent the knowledge of the mortgage expert.
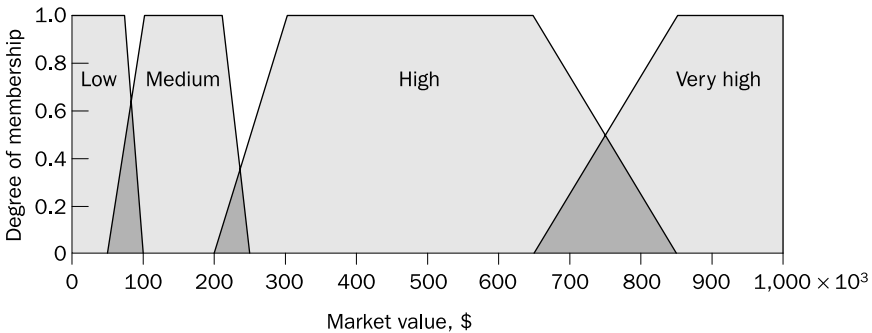
Figure 9.7   Fuzzy sets of the linguistic variable *Market value*
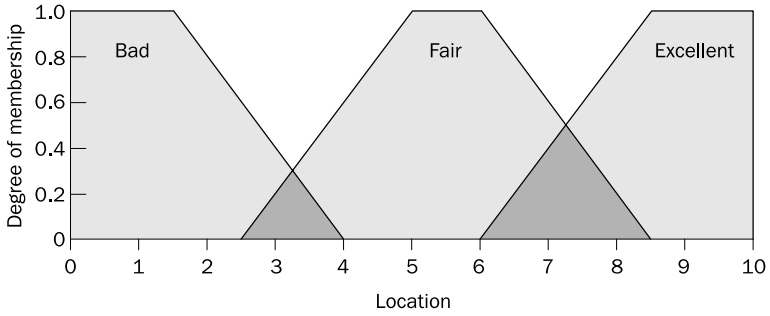
Figure 9.8 Fuzzy sets of the linguistic variable *Location*
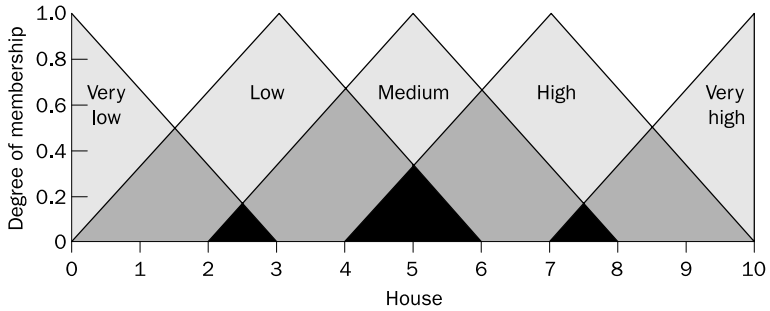
Figure 9.9 Fuzzy sets of the linguistic variable *House*
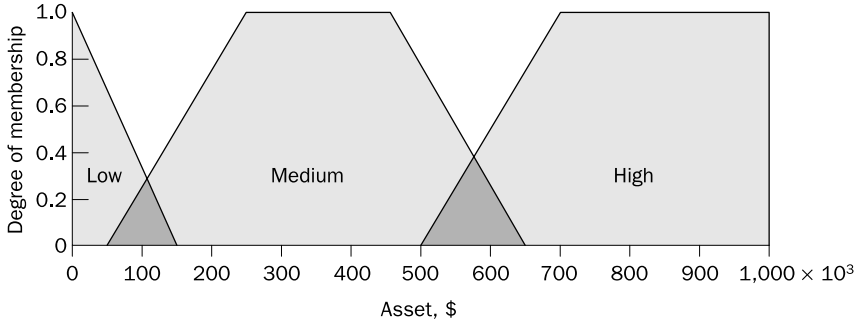
Figure 9.10 Fuzzy sets of the linguistic variable *Asset*
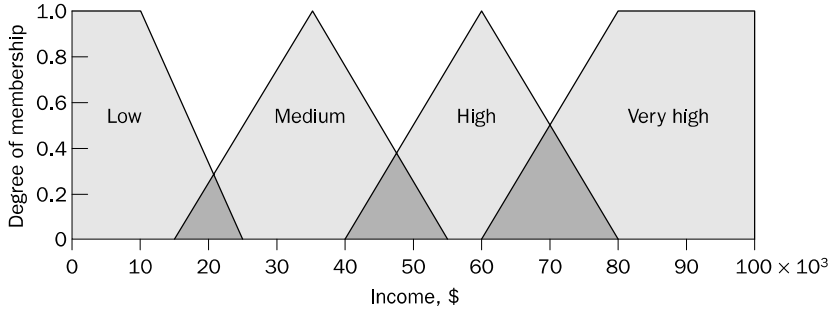
Figure 9.11 Fuzzy sets of the linguistic variable *Income*
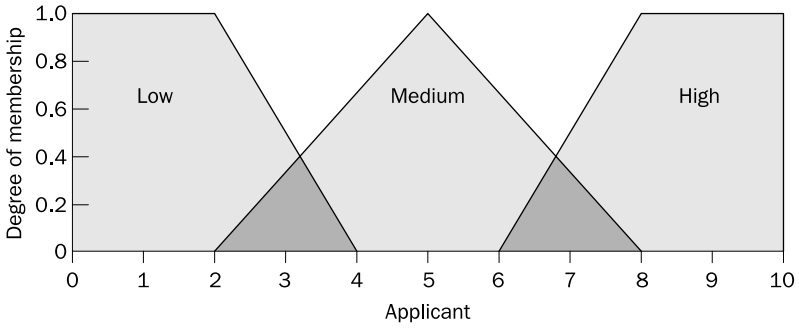
Figure 9.12   Fuzzy sets of the linguistic variable *Applicant*
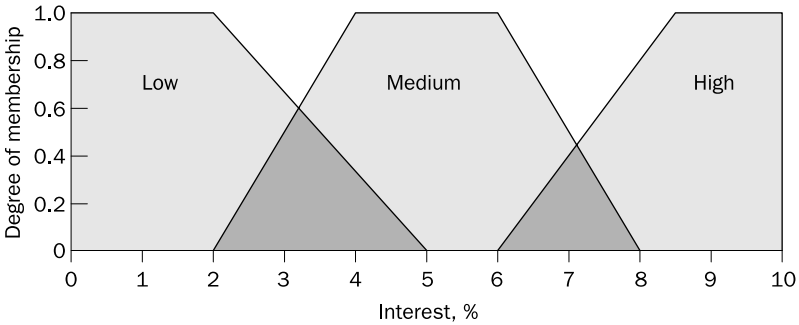
Figure 9.13   Fuzzy sets of the linguistic variable *Interest*
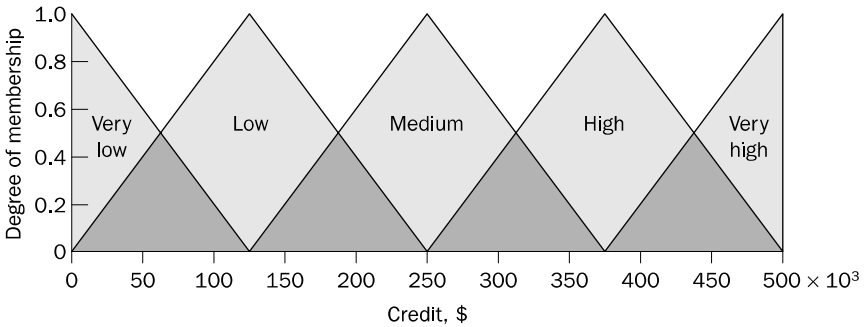
Figure 9.14   Fuzzy sets of the linguistic variable *Credit*

Next we obtain fuzzy rules. In our case, we simply adapt some of the basic rules used by Von Altrock in his fuzzy model for mortgage loan assessment (Von Altrock, 1997). These rules are shown in Figure 9.15.

Complex relationships between all variables used in the fuzzy system can be represented best by the hierarchical structure shown in Figure 9.16.

To build our system we use the MATLAB Fuzzy Logic Toolbox, one of the most popular fuzzy tools currently on the market.

The last phase in the development of a prototype system is its evaluation and testing.

Rule Base 1: Home Evaluation

1. If (Market_value is Low) then (House is Low)
2. If (Location is Bad) then (House is Low)
3. If (Location is Bad) and (Market_value is Low) then (House is Very_low)
4. If (Location is Bad) and (Market_value is Medium) then (House is Low)
5. If (Location is Bad) and (Market_value is High) then (House is Medium)
6. If (Location is Bad) and (Market_value is Very_ high) then (House is High)
7. If (Location is Fair) and (Market_value is Low) then (House is Low)
8. If (Location is Fair) and (Market_value is Medium) then (House is Medium)
9. If (Location is Fair) and (Market_value is High) then (House is High)
10. If (Location is Fair) and (Market_value is Very_high) then (House is Very_high)
11. If (Location is Excellent) and (Market_value is Low) then (House is Medium)
12. If (Location is Excellent) and (Market_value is Medium) then (House is High)
13. If (Location is Excellent) and (Market_value is High) then (House is Very_high)
14. If (Location is Excellent) and (Market_value is Very_high) then (House is Very_high)

Rule Base 2: Applicant Evaluation

1. If (Asset is Low) and (Income is Low) then (Applicant is Low)
2. If (Asset is Low) and (Income is Medium) then (Applicant is Low)
3. If (Asset is Low) and (Income is High) then (Applicant is Medium)
4. If (Asset is Low) and (Income is Very_high) then (Applicant is High)
5. If (Asset is Medium) and (Income is Low) then (Applicant is Low)
6. If (Asset is Medium) and (Income is Medium) then (Applicant is Medium)
7. If (Asset is Medium) and (Income is High) then (Applicant is High)
8. If (Asset is Medium) and (Income is Very_high) then (Applicant is High)
9. If (Asset is High) and (Income is Low) then (Applicant is Medium)
10. If (Asset is High) and (Income is Medium) then (Applicant is Medium)
11. If (Asset is High) and (Income is High) then (Applicant is High)
12. If (Asset is High) and (Income is Very_high) then (Applicant is High)

Rule Base 3: Credit Evaluation

1. If (Income is Low) and (Interest is Medium) then (Credit is Very_low)
2. If (Income is Low) and (Interest is High) then (Credit is Very_low)
3. If (Income is Medium) and (Interest is High) then (Credit is Low)
4. If (Applicant is Low) then (Credit is Very_low)
5. If (House is Very_low) then (Credit is Very_low)
6. If (Applicant is Medium) and (House is Very_low) then (Credit is Low)
7. If (Applicant is Medium) and (House is Low) then (Credit is Low)
8. If (Applicant is Medium) and (House is Medium) then (Credit is Medium)
9. If (Applicant is Medium) and (House is High) then (Credit is High)
10. If (Applicant is Medium) and (House is Very_high) then (Credit is High)
11. If (Applicant is High) and (House is Very_low) then (Credit is Low)
12. If (Applicant is High) and (House is Low) then (Credit is Medium)
13. If (Applicant is High) and (House is Medium) then (Credit is High)
14. If (Applicant is High) and (House is High) then (Credit is High)
15. If (Applicant is High) and (House is Very_high) then (Credit is Very_high)

Figure 9.15 Rules for mortgage loan assessment

To evaluate and analyse the performance of a fuzzy system, we can use the output surface viewer provided by the Fuzzy Logic Toolbox. Figures 9.17 and 9.18 represent three-dimensional plots of the fuzzy system for mortgage loan assessment. Finally, the mortgage experts would try the system with several test cases.

Decision-support fuzzy systems may include dozens, and even hundreds, of rules. For example, a fuzzy system for credit-risk evaluation developed by BMW
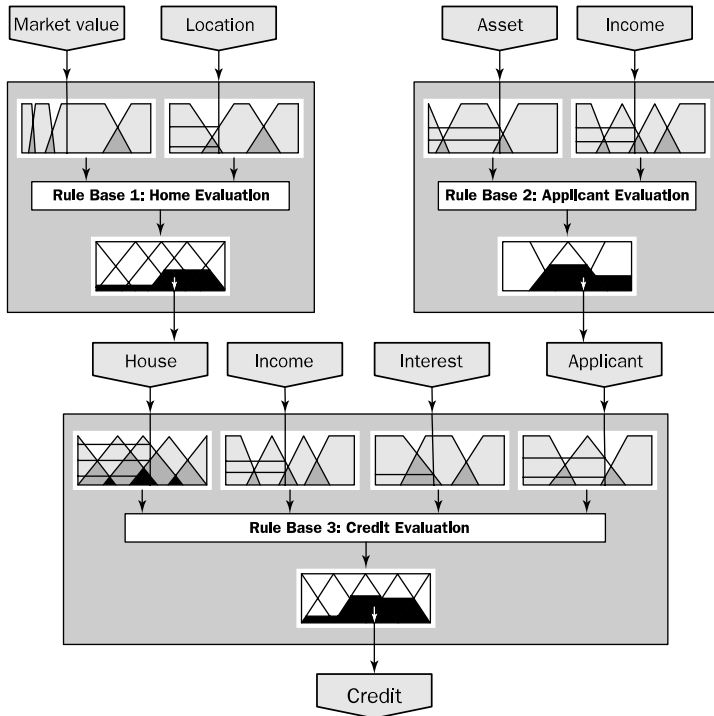
Figure 9.16   Hierarchical fuzzy model for mortgage loan assessment

Bank and Inform Software used 413 fuzzy rules (Güllich, 1996). Large knowledge bases are usually divided into several modules in a manner similar to that shown in Figure 9.16.

In spite of the often large number of rules, decision-support fuzzy systems can be developed, tested and implemented relatively quickly. For instance, it took just two person-years to develop and implement the fuzzy system for credit-risk evaluation. Compare this effort with the 40 person-years it took to develop MYCIN.
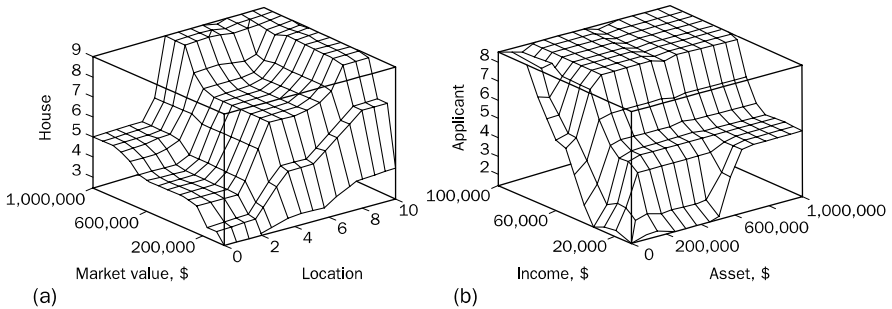
Figure 9.17   Three-dimensional plots for Rule Base 1 and Rule Base 2
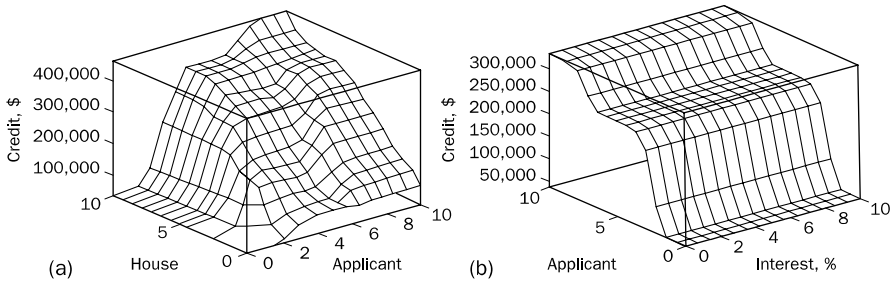
Figure 9.18   Three-dimensional plots for Rule Base 3

9.4   Will a neural network work for my problem?

Neural networks represent a class of very powerful, general-purpose tools that have been successfully applied to prediction, classification and clustering problems. They are used in a variety of areas, from speech and character recognition to detecting fraudulent transactions, from medical diagnosis of heart attacks to process control and robotics, from predicting foreign exchange rates to detecting and identifying radar targets. And the areas of neural network applications continue to expand rapidly.

The popularity of neural networks is based on their remarkable versatility, abilities to handle both binary and continuous data, and to produce good results in complex domains. When the output is continuous, the network can address prediction problems, but when the output is binary, the network works as a classifier.

Case study 4: Character recognition neural networks

I want to develop a character recognition system. Will a neural network work for this problem?

Recognition of both printed and handwritten characters is a typical domain where neural networks have been successfully applied. In fact, **optical character recognition** systems were among the first commercial applications of neural networks.

What is optical character recognition?
It is the ability of a computer to translate character images into a text file, using special software. It allows us to take a printed document and put it into a computer in editable form without the need of retyping the document.

To capture the character images we can use a desktop scanner. It either passes light-sensitive sensors over the illuminated surface of a page or moves a page through the sensors. The scanner processes the image by dividing it into hundreds of pixel-sized boxes per inch and representing each box by either 1 (if the box is filled) or 0 (if the box is empty). The resulting matrix of dots is called a **bit map**. Bit maps can be stored, displayed and printed by a computer,
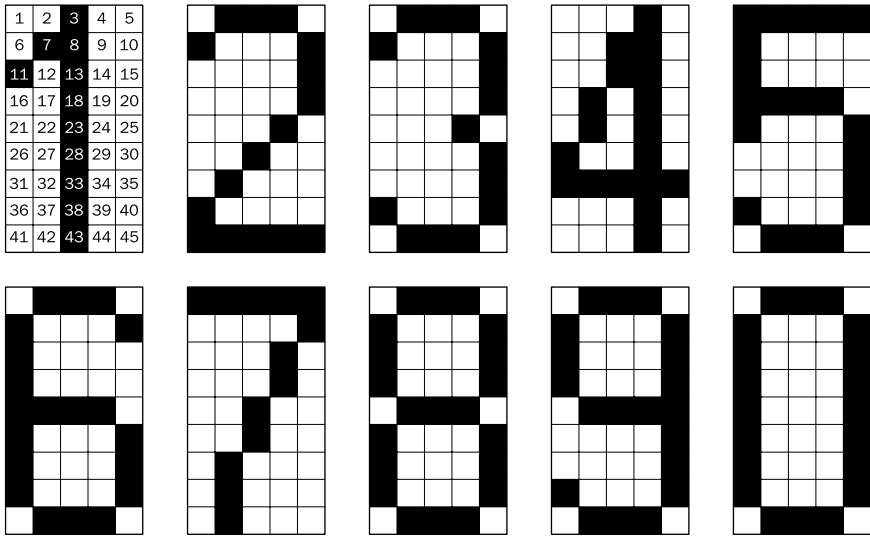
Figure 9.19   Bit maps for digit recognition

but we cannot use a word processor to edit the text – the patterns of dots have to be recognised as characters by the computer. This is the job for a neural network.

Let us demonstrate an application of a multilayer feedforward network for printed character recognition. For simplicity, we can limit our task to the recognition of digits from 0 to 9. In this application, each digit is represented by a 5×9 bit map, as shown in Figure 9.19. In commercial applications, where a better resolution is required, at least 16×16 bit maps are used (Zurada, 1992).

How do we choose the architecture of a neural network for character recognition?

The architecture and size of a neural network depend on the complexity of the problem. For example, handwritten character recognition is performed by rather complex multilayer networks that may include three, or even four, hidden layers and hundreds of neurons (Zurada, 1992; Haykin, 1999). However, for the printed digit recognition problem, a three-layer network with a single hidden layer will give sufficient accuracy.

The number of neurons in the input layer is decided by the number of pixels in the bit map. The bit map in our example consists of 45 pixels, and thus we need 45 input neurons. The output layer has 10 neurons – one neuron for each digit to be recognised.

How do we determine an optimal number of hidden neurons?

Simulation experiments indicate that the number of neurons in the hidden layer affects both the accuracy of character recognition and the speed of training the network. Complex patterns cannot be detected by a small number of hidden neurons; however too many of them can dramatically increase the computational burden.

Another problem is **overfitting**. The greater the number of hidden neurons, the greater the ability of the network to recognise existing patterns. However, if the number of hidden neurons is too big, the network might simply memorise all training examples. This may prevent it from generalising, or producing correct outputs when presented with data that was not used in training. For instance, the overfitted character recognition network trained with Helvetica-font examples might not be able to recognise the same characters in the Times New Roman font.

The practical approach to preventing overfitting is to choose the smallest number of hidden neurons that yields good generalisation. Thus, at the starting point, an experimental study could begin with as little as two neurons in the hidden layer. In our example, we will examine the system's performance with 2, 5, 10 and 20 hidden neurons and compare results.

The architecture of a neural network (with five neurons in the hidden layer) for the character recognition problem is shown in Figure 9.20. Neurons in the hidden and output layers use a sigmoid activation function. The neural network is trained with the back-propagation algorithm with momentum; the momentum constant is set to 0.95. The input and output training patterns are shown in Table 9.2. The binary input vectors representing the bit maps of the respective digits are fed directly into the network.

The network's performance in our study is measured by the sum of squared errors. Figure 9.21 demonstrates the results; as can be seen from Figure 9.21(a), a neural network with two neurons in the hidden layer cannot converge to a solution, while the networks with 5, 10 and 20 hidden neurons learn relatively fast. In fact, they converge in less than 250 epochs (each epoch represents an entire pass through all training examples). Also note that the network with 20 hidden neurons shows the fastest convergence.

Once the training is complete, we must test the network with a set of test examples to see how well it performs.
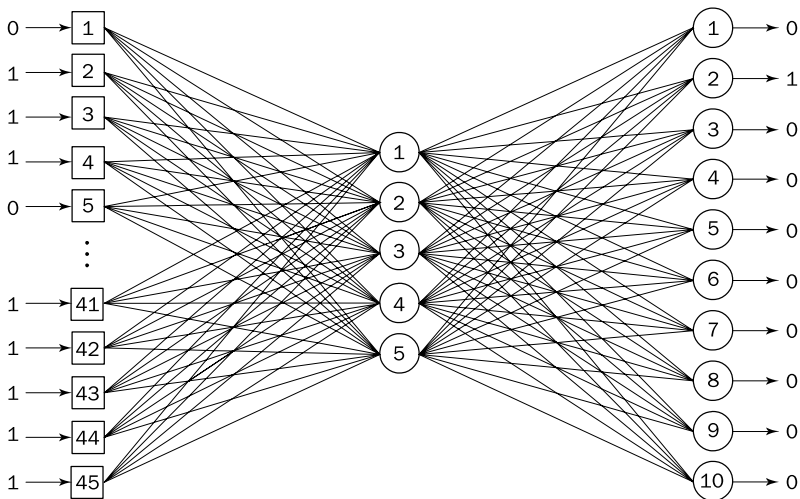
Figure 9.20 Neural network for printed digit recognition

Table 9.2 Input and desired output patterns for the digit recognition neural network

| Digit | Input patterns Rows in the pixel matrix | | | | | | | | | Desired output patterns |
	1	2	3	4	5	6	7	8	9	
1	00100	01100	10100	00100	00100	00100	00100	00100	00100	1000000000
2	01110	10001	00001	00001	00010	00100	01000	10000	11111	0100000000
3	01110	10001	00001	00001	00010	00001	00001	10001	01110	0010000000
4	00010	00110	00110	01010	01010	10010	11111	00010	00010	0001000000
5	11111	10000	10000	11110	10001	00001	00001	10001	01110	0000100000
6	01110	10001	10000	10000	11110	10001	10001	10001	01110	0000010000
7	11111	00001	00010	00010	00100	00100	01000	01000	01000	0000001000
8	01110	10001	10001	10001	01110	10001	10001	10001	01110	0000000100
9	01110	10001	10001	10001	01111	00001	00001	10001	01110	0000000010
0	01110	10001	10001	10001	10001	10001	10001	10001	01110	0000000001
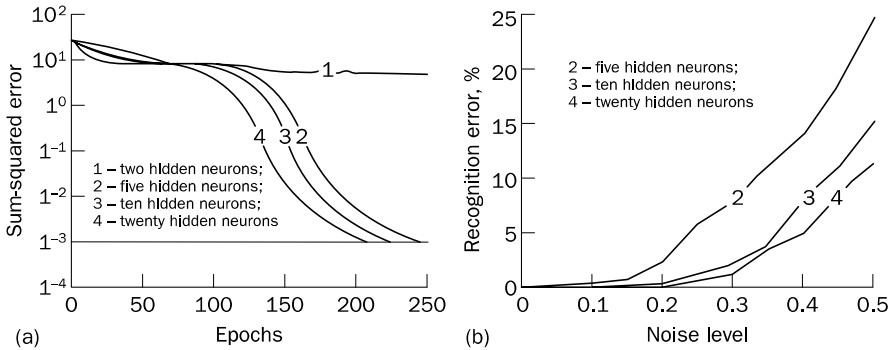
Figure 9.21 Training and performance evaluation of the digit recognition three-layer neural networks: (a) learning curves; (b) performance evaluation

What are the test examples for character recognition? Are they the same that were used for neural network training?

A test set has to be strictly independent from the training examples. Thus, to test the character recognition network, we must present it with examples that include 'noise' – the distortion of the input patterns. This distortion can be created, for instance, by adding some small random values chosen from a normal distribution to the binary input vectors representing bit maps of the ten digits. We evaluate the performance of the printed digit recognition networks with 1000 test examples (100 for each digit to be recognised). The results are shown in Figure 9.21(b).

Although the average recognition error of the network with 20 hidden neurons is the lowest, the results do not demonstrate significant differences between the networks with 10 and 20 hidden neurons. Both networks can sustain similar levels of noise without sacrificing their recognition performance. On this basis, we may conclude that for the digit recognition problem described here, the use of 10 hidden neurons is adequate.

Can we improve the performance of the character recognition neural network?

A neural network is as good as the examples used to train it. Therefore, we can attempt to improve digit recognition by feeding the network with 'noisy' examples of digits from 0 to 9. The results of such an attempt are shown in Figure 9.22. As we expected, there is some improvement in the performance of the digit recognition network trained with 'noisy' data.

This case study illustrated one of the most common applications of multilayer neural networks trained with the back-propagation algorithm. Modern character recognition systems are capable of processing different fonts in English, French, Spanish, Italian, Dutch and several other languages with great accuracy. Optical character recognition is routinely used by office workers, lawyers, insurance clerks, journalists – in fact anybody who wants to take a printed (or even
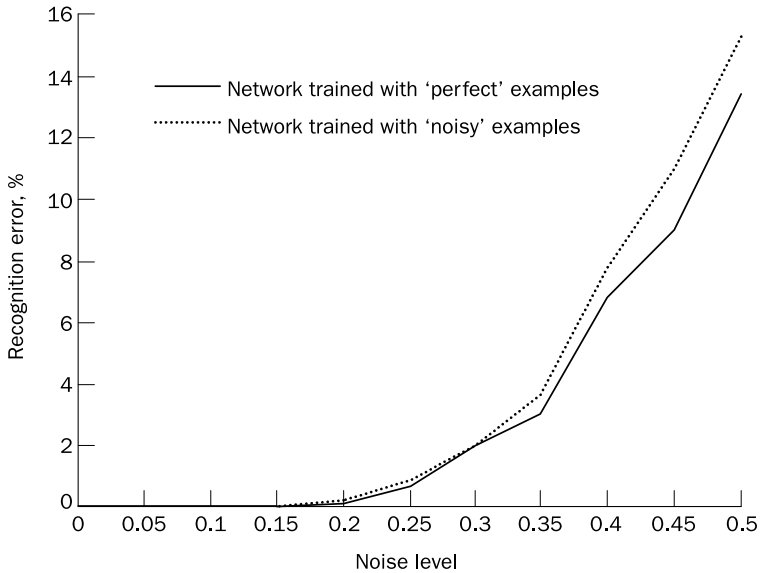
Figure 9.22 Performance evaluation of the digit recognition network trained with 'noisy' examples

handwritten) document and load it into their computer as an editable file. Handwritten digit recognition systems are widely used in processing zip codes on mail envelopes (LeCun *et al.*, 1990).

Case study 5: Prediction neural networks

I want to develop an intelligent system for real-estate appraisal. Will a neural network work for this problem?

Real-estate appraisal is a problem of predicting the market value of a given house based on the knowledge of the sales prices of similar houses. As we mentioned earlier, this problem can be solved with expert systems as well as neural networks. Of course, if we choose to apply a neural network, we will not be able to understand how an appraisal of a particular house is reached – a neural network is essentially a black-box to the user and rules cannot be easily extracted from it. On the other hand, an accurate appraisal is often more important than understanding how it was done.

In this problem, the inputs (the house location, living area, number of bedrooms, number of bathrooms, land size, type of heating system, etc.) are well-defined, and normally even standardised for sharing the housing market information between different real estate agencies. The output is also well defined – we know what we are trying to predict. Most importantly, there are many examples we can use for training the neural network. These examples are the features of recently sold houses and their sales prices.

Choosing training examples is critical for an accurate prediction. A training set must cover the full range of values for all inputs. Thus, in the training set for real estate appraisal, we should include houses that are large and small, expensive and inexpensive, with and without garages, etc. And the training set has to be sufficiently large.

But how do we determine when the size of a training set is 'sufficiently large'?

A network's ability to generalise is influenced by three main factors: the size of the training set, the architecture of the network, and the complexity of the problem. Once the network architecture is decided, the issue of generalisation is resolved by the adequacy of the training set. An appropriate number of training examples can be estimated with **Widrow's rule of thumb**, which suggests that, for a good generalisation, we need to satisfy the following condition (Widrow and Stearns, 1985; Haykin, 1999):

$$N = \frac{n_w}{e},\qquad(9.1)$$

where N is the number of training examples, n_w is the number of synaptic weights in the network, and e is the network error permitted on test.

Thus, if we allow an error of, say, 10 per cent, the number of training examples should be approximately 10 times bigger than the number of weights in the network.

In solving prediction problems, including real-estate appraisal, we often combine input features of different types. Some features, such as the house's condition and its location, can be arbitrarily rated from 1 (least appealing) to 10 (most appealing). Some features, such as the living area, land size and sales price, are measured in actual physical quantities – square metres, dollars, etc. Some features represent counts (number of bedrooms, number of bathrooms, etc.), and some are categories (type of heating system).

A neural network works best when all its inputs and outputs vary within the range between 0 and 1, and thus all the data must be massaged before we can use them in a neural network model.

How do we massage the data?

Data can be divided into three main types: continuous, discrete and categorical (Berry and Linoff, 1997), and we normally use different techniques to massage different types of data.

Continuous data vary between two pre-set values – minimum and maximum, and can be easily mapped, or massaged, to the range between 0 and 1 as:

$$massaged\ value = \frac{actual\ value - minimum\ value}{maximum\ value - minimum\ value}\qquad(9.2)$$

For instance, if the living areas of the houses in training examples range between 59 and 231 square metres, we might set the minimum value to 50 and the maximum to 250 square metres. Any value lower than the minimum is mapped to the minimum, and any value higher than the maximum to the maximum. Thus, a living area of, say, 121 square metres would be massaged as:

$$massaged\ value_{121} = \frac{121 - 50}{250 - 50} = 0.355$$

This method works well for most applications.

Discrete data, such as the number of bedrooms and the number of bathrooms, also have maximum and minimum values. For example, the number of bedrooms usually ranges from 0 to 4. Massaging discrete data is simple – we assign an equal space to each possible value on the interval from 0 to 1, as shown in Figure 9.23.

A neural network can now handle a feature like the number of bedrooms as a single input. For example, a three-bedroom house would be represented by the input value of 0.75.

This approach is sufficient for most applications with discrete features that have up to a dozen possible values. However, if there are more than a dozen values, a discrete feature should be treated like a continuous one.

Categorical data, such as gender and marital status, can be massaged by using **1 of N coding** (Berry and Linoff, 1997). This method implies that each categorical value is handled as a separate input. For example, marital status, which can be either single, divorced, married or widowed, would be represented by four inputs. Each of these inputs can have a value of either 0 or 1. Thus, a married person would be represented by an input vector $[0\,0\,1\,0]$.

Let us now construct a feedforward neural network for real-estate appraisal. Figure 9.24 represents a simplified model that was set up by using training examples with features of the houses recently sold in Hobart.

In this model, the input layer, which includes 10 neurons, passes the massaged input values to the hidden layer. All input features, except *type of heating system*, are treated as single inputs. The type of heating system represents a categorical type of data, which is massaged with 1 of N coding.
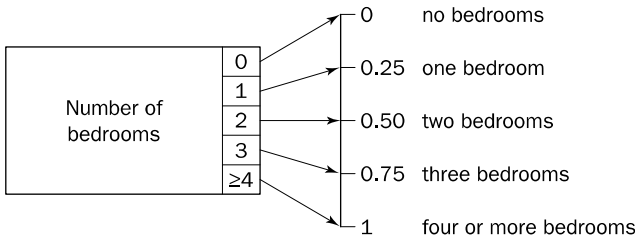
Figure 9.23 Massaging discrete data

Location rating	7	0.7000	→	1
House condition	8	0.8000	→	2
Number of bedrooms	3	0.7500	→	3
Number of bathrooms	1	0.5000	→	4
Living area	121.00 m²	0.3550	→	5
Land size	682.00 m²	0.4682	→	6
Garage	double	1.0000	→	7
		1.0000	→	8
Type of heating system	wood heater	0.0000	→	9
		0.0000	→	10
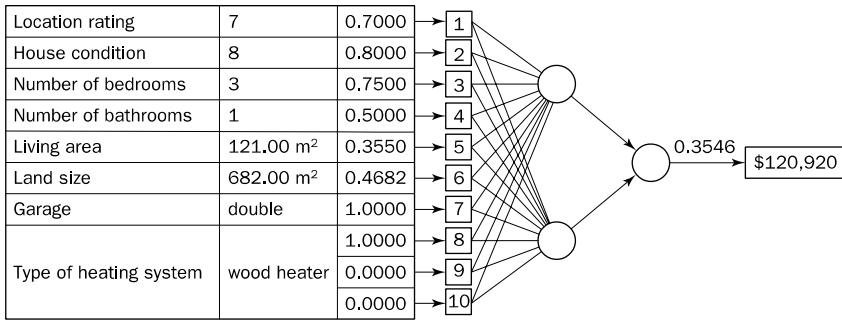
Figure 9.24 Feedforward neural network for real-estate appraisal

The hidden layer includes two neurons, and the output layer is represented by a single neuron. Neurons in the hidden and output layers apply sigmoid activation functions.

The neural network for real-estate appraisal determines the value of a house, and thus the network output can be interpreted in dollars.

But how do we interpret the network output?
In our example, the network output is represented by continuous values in the range between 0 and 1. Thus, to interpret the results, we can simply reverse the procedure we used for massaging continuous data. Suppose, for instance, that in the training set, sales prices range between \$52,500 and \$225,000, and the output value is set up so that \$50,000 maps to 0 and \$250,000 maps to 1. Then, if the network output is 0.3546, we can compute that this value corresponds to:

$$actual\ value_{0.3546} = 0.3546 \times (\$250,000 - \$50,000) + \$50,000 = \$120,920$$

How do we validate results?
To validate results, we use a set of examples never seen by the network. Before training, all the available data are randomly divided into a training set and a test set. Once the training phase is complete, the network's ability to generalise is tested against examples of the test set.

A neural network is opaque. We cannot see how the network derives its results. But we still need to grasp relationships between the network inputs and the results it produces. Although current research into rule extraction from trained neural networks will eventually bring adequate outcomes, the non-linear characteristics of neurons may prevent the network from producing simple and understandable rules. Fortunately, to understand the importance of a particular input to the network output, we do not need rule extraction. Instead we can use a simple technique called **sensitivity analysis**.

Sensitivity analysis determines how sensitive the output of a model is to a particular input. This technique is used for understanding internal relationships in opaque models, and thus can be applied to neural networks. Sensitivity analysis is performed by measuring the network output when each input is set

(one at a time) to its minimum and then its maximum values. Changes in some inputs may have little effect on the network output – the network is not sensitive to these inputs. Changes in other inputs have a much greater effect on the network output – the network is sensitive to these inputs. The amount of change in the network output represents the network's sensitivity to a respective input. In many cases, sensitivity analysis can be as good as the rules extracted from the trained neural network.

Case study 6: Classification neural networks with competitive learning

I want to develop an intelligent system that can divide a group of iris plants into classes and then assign any iris plant to one of these classes. I have a data set with several variables but I have no idea how to separate it into different classes because I cannot find any unique or distinctive features in the data. Will a neural network work for this problem?

Neural networks can discover significant features in input patterns and learn how to separate input data into different classes. A neural network with competitive learning is a suitable tool to accomplish this task.

The competitive learning rule enables a single-layer neural network to combine similar input data into groups or **clusters**. This process is called **clustering**. Each cluster is represented by a single output. In fact, clustering can be defined as the process of dividing an input space into regions, each of which is associated with a particular output (Principe *et al.*, 2000).

For this case study, we will use a data set of 150 elements that contains three classes of iris plants: *Iris setosa, versicolor* and *virginica* (Fisher, 1950). Each plant in the data set is represented by four variables: sepal length, sepal width, petal length and petal width. The sepal length ranges between 4.3 and 7.9 cm, sepal width between 2.0 and 4.4 cm, petal length between 1.0 and 6.9 cm, and petal width between 0.1 and 2.5 cm.

In a competitive neural network, each input neuron corresponds to a single input, and each competitive neuron represents a single cluster. Thus, the network for the iris plant classification problem will have four neurons in the input layer and three neurons in the competitive layer. The network's architecture is shown in Figure 9.25.

However, before the network is trained, the data must be massaged and then divided into training and test sets.

The iris plant data are continuous, vary between some minimum and maximum values, and thus can easily be massaged to the range between 0 and 1 using Eq. (9.2). Massaged values can then be fed to the network as its inputs.

The next step is to generate training and test sets from the available data. The 150-element iris data is randomly divided into a training set of 100 elements and a test set of 50 elements.

Now we can train the competitive neural network to divide input vectors into three classes. Figure 9.26 illustrates the learning process with the learning rate of
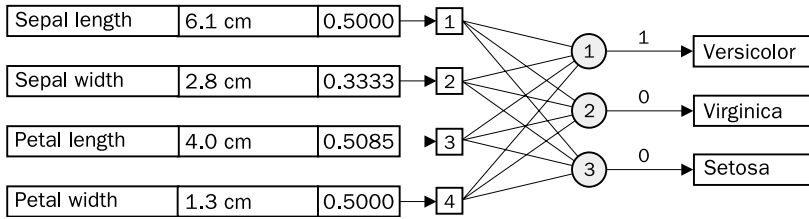
Figure 9.25   Neural network for iris plant classification

0.01. Black dots here represent the input patterns and three spheres denote the weight vectors of the competitive neurons. The location of each sphere is determined by the neuron's weights in the four-dimensional input space.

Initially all weights of the competitive neurons are assigned the same value of 0.5, and thus only one sphere appears in the centre of the input space, as shown in Figure 9.26(a). After training, the weight vectors correspond to the positions of the cluster centres, so that each competitive neuron can now respond to input data in a particular region.
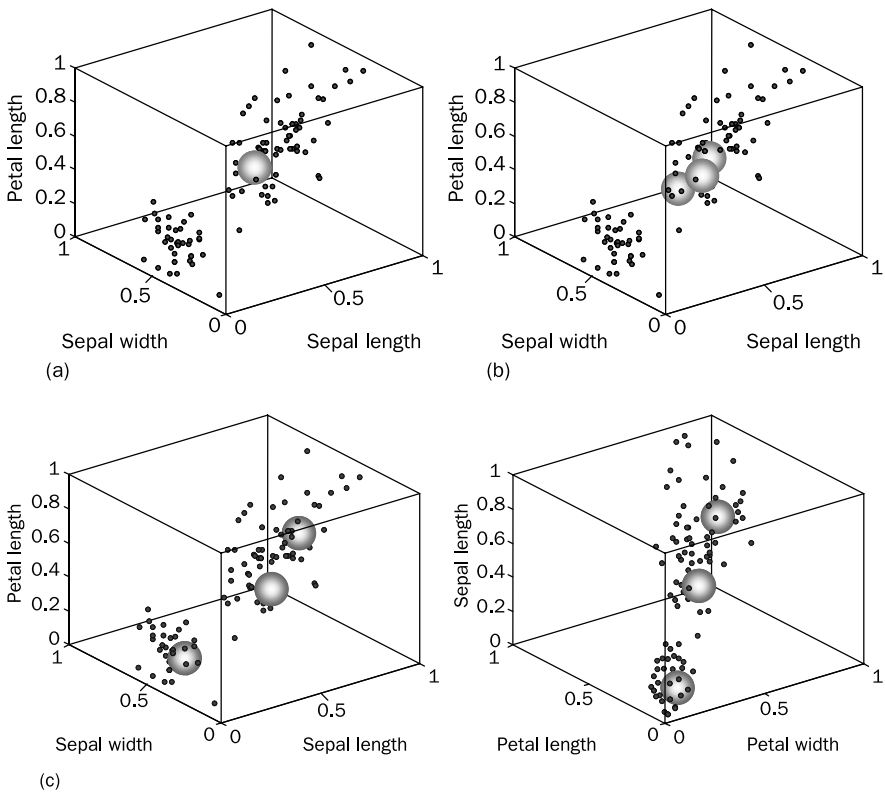
Figure 9.26   Competitive learning in the neural network for iris plant classification: (a) initial weights; (b) weight after 100 iterations; (c) weight after 2000 iterations

How do we know when the learning process is complete?
In a competitive neural network, unlike a multilayer perceptron trained with the back-propagation algorithm, there is no obvious way of knowing whether the learning process is complete or not. We do not know what the desired outputs are, and thus cannot compute the sum of squared errors – a criterion used by the back-propagation algorithm. Therefore, we should use the **Euclidean distance** criterion instead. When no noticeable changes occur in the weight vectors of competitive neurons, a network can be considered to have converged. In other words, if the motion of competitive neurons in the input space remains sufficiently constrained for several subsequent epochs, then we can assume that the learning process is complete.

Figure 9.27 shows the dynamics of the learning process for competitive neurons of the iris classification neural network. The network was trained with two different learning rates. As can be seen in Figure 9.27(b), if the learning rate is too high, the behaviour of competitive neurons may become erratic, and the network may never converge. However, in order to accelerate learning, we can still use large initial values of the learning rate, but as training progresses the learning rate must gradually decrease.

How can we associate an output neuron with a particular class? How do we know, for example, that the competitive neuron 1 represents class Versicolor?
Competitive neural networks enable us to identify clusters in input data. However, since clustering is an unsupervised process, we cannot use it directly for labelling output neurons. In fact, clustering is just a preliminary stage of classification.

In most practical applications, the distribution of data that belong to the same cluster is rather dense, and there are usually natural valleys between different clusters. As a result, the position of the centre of a cluster often reveals distinctive features of the corresponding class. On the other hand, the weight vectors of the competitive neurons after training provide us with the coordinates of these centres in the input space. Thus, a competitive neuron can be associated with a particular class through its weights. Table 9.3 contains the final weights of

(a) *Learning rate 0.01*   (b) *Learning rate 0.5*
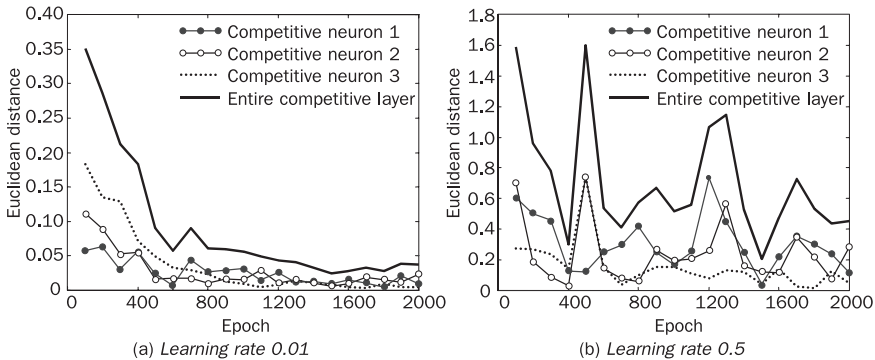
Figure 9.27   Learning curves for competitive neurons of the iris classification neural network

Table 9.3 Labelling the competitive neurons

Neuron	Weights		Dimensions of the iris palnt, cm		Class of the iris plant
1	W_{11}	0.4355	Sepal length	5.9	*Versicolor*
	W_{21}	0.3022	Sepal width	2.7	
	W_{31}	0.5658	Petal length	4.4	
	W_{41}	0.5300	Petal width	1.4	
2	W_{12}	0.6514	Sepal length	6.7	*Virginica*
	W_{22}	0.4348	Sepal width	3.0	
	W_{32}	0.7620	Petal length	5.5	
	W_{42}	0.7882	Petal width	2.0	
3	W_{13}	0.2060	Sepal length	5.0	*Setosa*
	W_{23}	0.6056	Sepal width	3.5	
	W_{33}	0.0940	Petal length	1.6	
	W_{43}	0.0799	Petal width	0.3	

the competitive neurons, decoded values of these weights and the corresponding classes of the iris plant.

How do we decode weights into iris dimensions?
To decode the weights of the competitive neurons into dimensions of the iris plant we simply reverse the procedure used for massaging the Iris data. For example,

$$\text{Sepal length } W_{11} = 0.4355 \times (7.9 - 4.3) + 4.3 = 5.9 \text{ cm}$$

Once the weights are decoded, we can ask an iris plant expert to label the output neurons.

Can we label the competitive neurons automatically without having to ask the expert?
We can use a test data set for labelling competitive neurons automatically. Once training of a neural network is complete, a set of input samples representing the same class, say class *Versicolor*, is fed to the network, and the output neuron that wins the competition most of the time receives a label of the corresponding class.

Although a competitive network has only one layer of competitive neurons, it can classify input patterns that are not linearly separable. In classification tasks, competitive networks learn much faster than multilayer perceptrons trained with the back-propagation algorithm, but they usually provide less accurate results.

9.5   Will genetic algorithms work for my problem?

Genetic algorithms are applicable to many optimisation problems (Haupt and Haupt, 1998). Optimisation is essentially the process of finding a better solution to a problem. This implies that the problem has more than one solution and the solutions are not of equal quality. A genetic algorithm generates a population of competing candidate solutions and then causes them to evolve through the process of natural selection – poor solutions tend to die out, while better solutions survive and reproduce. By repeating this process over and over again, the genetic algorithm breeds an optimal solution.

Case study 7: The travelling salesman problem

I want to develop an intelligent system that can produce an optimal itinerary. I am going to travel by car and I want to visit all major cities in Western and Central Europe and then return home. Will a genetic algorithm work for this problem?

This problem is well known as the **travelling salesman problem** (TSP). Given a finite number of cities, *N*, and the cost of travel (or the distance) between each pair of cities, we need to find the cheapest way (or the shortest route) for visiting each city exactly once and returning to the starting point.

Although the TSP became known as early as the eighteenth century, only in the late 1940s and early 1950s was the problem studied seriously and then publicised as a typical NP-hard problem (Dantzig *et al.*, 1954; Flood, 1955). Such problems are hard to solve by combinatorial search techniques. The search space for the TSP includes all possible combinations of *N* cities, and thus the size of the search space equals *N*! (the factorial of the number of cities). Because the number of cities can be quite large, examining the alternative routes one by one is not feasible.

The TSP is naturally represented in numerous transportation and logistics applications such as arranging routes for school buses to pick up children in a school district, delivering meals to home-bound people, scheduling stacker cranes in a warehouse, planning truck routes to pick up parcel post, and many others. A classic example of the TSP is the scheduling of a machine to drill holes in a circuit board. In this case, the holes are the cities, and the cost of travel is the time it takes to move the drill head from one hole to the next.

Over the years the size of the TSP has grown dramatically, moving from the solution of a 49-city problem (Dantzig *et al.*, 1954) up to the recent solution of a 15,112-city problem (Applegate *et al.*, 2001).

Researchers apply different techniques to solve this problem. These techniques include simulated annealing (Laarhoven and Aarts, 1987), discrete linear programming (Lawler *et al.*, 1985), neural networks (Hopfield and Tank, 1985), branch-and-bound algorithms (Tschoke *et al.*, 1995), Markov chains (Martin *et al.*, 1991) and genetic algorithms (Potvin, 1996). Genetic algorithms are

particularly suitable for the TSP because they can rapidly direct the search to promising areas of the search space.

How does a genetic algorithm solve the TSP?

First, we need to decide how to represent a route of the salesman. The most natural way of representing a route is path representation (Michalewicz, 1996). Each city is given an alphabetic or numerical name, the route through the cities is represented as a chromosome, and appropriate genetic operators are used to create new routes.

Suppose we have nine cities numbered from 1 to 9. In a chromosome, the order of the integers represents the order in which the cities will be visited by the salesman. For example, a chromosome

| 1 | 6 | 5 | 3 | 2 | 8 | 4 | 9 | 7 |
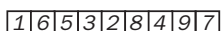
represents the route shown in Figure 9.28. The salesman starts at *City* 1, visits all the other cities once and returns to the starting point.

How does the crossover operator work in the TSP?

The crossover operator in its classical form cannot be directly applied to the TSP. A simple exchange of parts between two parents would produce illegal routes containing duplicates and omissions – some cities would be visited twice while others would not be visited at all. For example, exchanging parts of the two parent chromosomes

Parent 1: | 1 | 6 | 5 | 3 | 2 | 8 | 4 | 9 | 7 | *Parent 2:* | 3 | 7 | 6 | 1 | 9 | 4 | 8 | 2 | 5 |
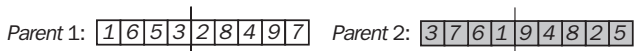
would produce one route with *City* 5 visited twice and *City* 7 omitted, and the other with *City* 7 visited twice and *City* 5 omitted.

Child 1: | 1 | 6 | 5 | 3 | 9 | 4 | 8 | 2 | 5 | *Child 2:* | 3 | 7 | 6 | 1 | 2 | 8 | 4 | 9 | 7 |

Clearly, the classical crossover with a single crossover point does not work in the TSP. To overcome this problem, a number of two-point crossover operators were proposed (Goldberg, 1989). For example, Goldberg and Lingle (1985) suggested the *partially mapped crossover*, and Davis (1985) introduced the *ordered*
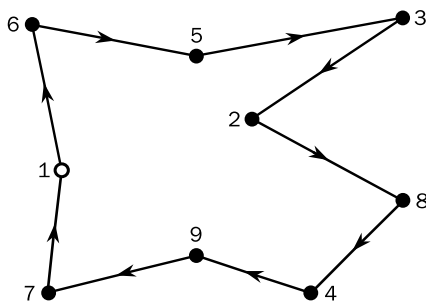
Figure 9.28   An example of the salesman's route

Step 1

Parent 1: | 1 | 6 | 5 | 3 | 2 | 8 | 4 | 9 | 7 | Parent 2: | 3 | 7 | 6 | 1 | 9 | 4 | 8 | 2 | 5 |

Child 1: | * | * | * | 1 | 9 | 4 | 8 | | Child 2: | * | * | * | 3 | 2 | 8 | 4 | * | * |

Step 2

Parent 1: | | 6 | 5 | 3 | 2 | | | | 7 | Parent 2: | | 7 | 6 | 1 | 9 | | | | 5 |

Step 3

Child 1: | 6 | 5 | 3 | 1 | 9 | 4 | 8 | 2 | 7 | Child 2: | 7 | 6 | 1 | 3 | 2 | 8 | 4 | 9 | 5 |
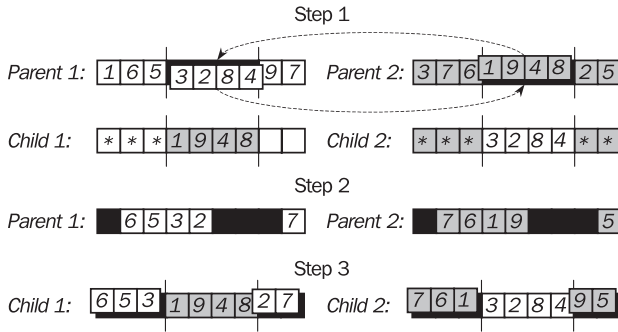
Figure 9.29 Crossover operators for the TSP

crossover. However, most operators are based on creating an offspring by choosing a part of a route from one parent and preserving the order of cities from the other parent. Figure 9.29 demonstrates how the crossover operator works.

First, two crossover points (marked by the | character) are chosen uniformly at random along the strings of two parent chromosomes. Chromosome material between crossover points defines swap sections. Two offspring chromosomes are created by interchanging the swap sections between the parents; in Figure 9.29 asterisks represent yet undecided cities. Next, the original cities from each parent are placed in their original order, omitting cities present in the other parent's swap section. For example, cities 1, 9, 4 and 8, which appear in the swap section from the second parent, are removed from the first parent. The remaining cities are then placed in the offspring, preserving their original order. As a result, an offspring represents a route partly determined by each of its parents.

How does the mutation operator works in the TSP?

There are two types of mutation operators: reciprocal exchange and inversion (Michalewicz, 1996). Figures 9.30(a) and (b) show how they work. The reciprocal exchange operator simply swaps two randomly selected cities in the chromosome. The inversion operator selects two random points along the chromosome string and reverses the order of the cities between these points.

How do we define a fitness function in the TSP?

While creating genetic operators for the TSP is not trivial, the design of a fitness function is straightforward – all we need to do is to evaluate the total length of the

Reciprocal exchange   Inversion

✗   ✗

| 1 | 6 | 5 | 3 | 2 | 8 | 4 | 9 | 7 | | 1 | 6 | 5 | 3 | 2 | 8 | 4 | 9 | 7 |
(a)

| 1 | 6 | 8 | 3 | 2 | 5 | 4 | 9 | 7 | | 1 | 2 | 3 | 5 | 6 | 8 | 4 | 9 | 7 |
(b)
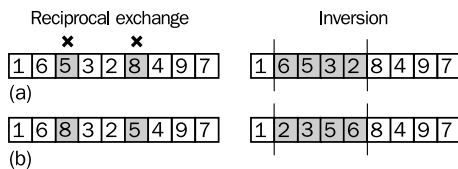
Figure 9.30 Mutation operators for the TSP: (a) original chromosomes; (b) mutated chromosomes
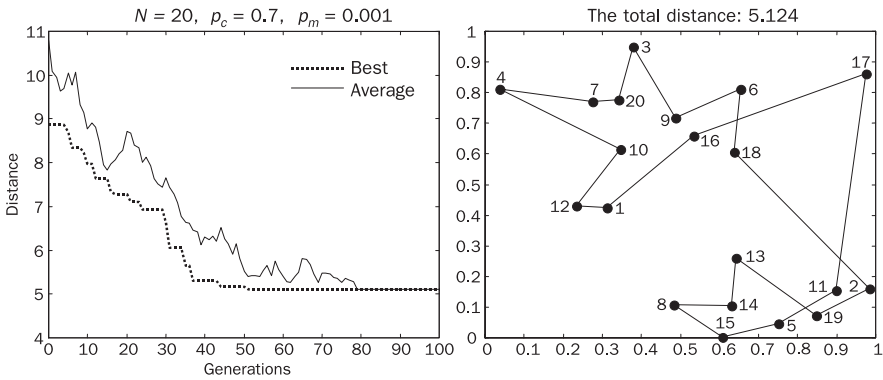
Figure 9.31 Performance graph and the best salesman's routes created in a population of 20 chromosomes after 100 generations

route. The fitness of each individual chromosome is determined as the reciprocal of the route length. In other words, the shorter the route, the fitter the chromosome.

Once the fitness function is defined and genetic operators are constructed, we can implement and run the GA.

As an example, let us consider the TSP for 20 cities placed in a 1×1 square. First, we choose the size of a chromosome population and the number of generations to run. We might start with a relatively small population and after a few generations examine the solutions obtained. Figure 9.31 shows the best route created by 20 chromosomes after 100 generations. As we can see, this route is not optimal and obviously can be improved. Let us increase the size of the chromosome population and run the GA again. Figure 9.32(a) demonstrates the results. The total length of the route decreases by 20 per cent – a very significant improvement.

But how do we know that the GA has actually found the optimal route?
The fact is, we don't. Only further trial runs on different sizes of chromosome populations with different rates of crossover and mutation can provide the answer.

Let us, for example, increase the mutation rate up to 0.01. Figure 9.32(b) shows the results. Although the total distance decreased slightly, the salesman's route is still similar to the one shown in Figure 9.32(a). Perhaps we might now attempt to increase the size of the chromosome population and rerun the GA. However, it is highly unlikely that we would achieve a noticeably better solution. Can we be sure that this route is *the optimal* one? Of course, we wouldn't bet on it! However, after several runs, we can be absolutely sure that the route we obtained is a good one.

9.6 Will a hybrid intelligent system work for my problem?

Solving complex real-world problems requires an application of complex intelligent systems that combine the advantages of expert systems, fuzzy logic, neural networks and evolutionary computation. Such systems can integrate
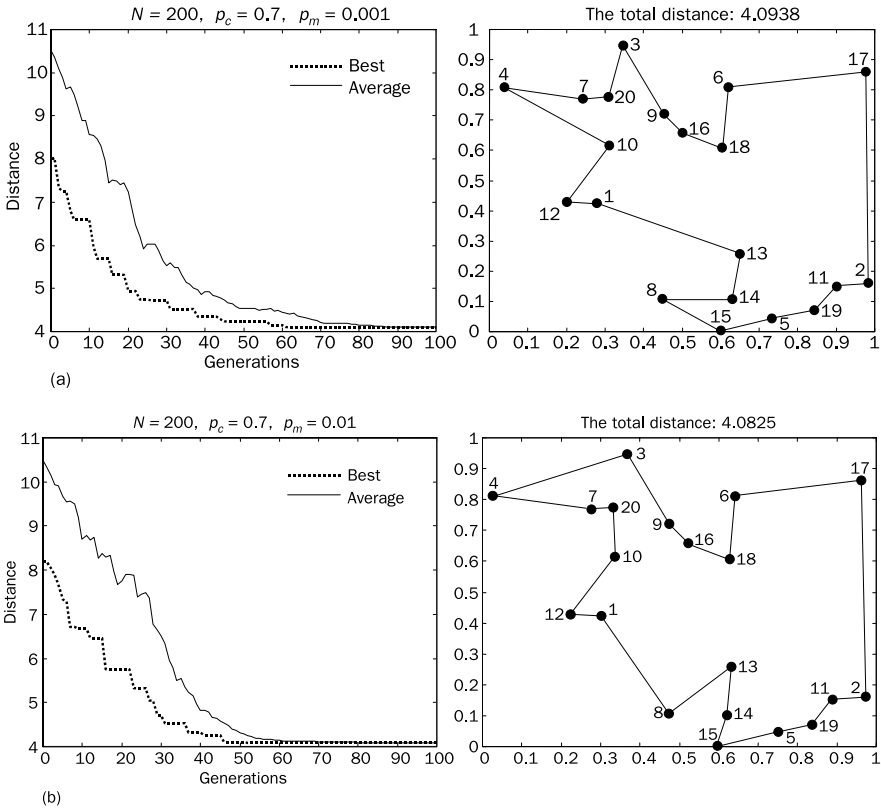
Figure 9.32 Performance graphs and the best routes created in a population of 200 chromosomes: (a) mutation rate is 0.001; (b) mutation rate is 0.01

human-like expertise in a specific domain with abilities to learn and adapt to a rapidly changing environment.

Although the field of hybrid intelligent systems is still evolving, and most hybrid tools are not yet particularly effective, neuro-fuzzy systems have already matured as an advanced technology with numerous successful applications. While neural networks can learn from data, the key benefit of fuzzy logic lies in its ability to model decision-making of humans.

Case study 8: Neuro-fuzzy decision-support systems

I want to develop an intelligent system for diagnosing myocardial perfusion from cardiac images. I have a set of cardiac images as well as the clinical notes and physician's interpretation. Will a hybrid system work for this problem?

Diagnosis in modern cardiac medicine is based on the analysis of SPECT (Single Proton Emission Computed Tomography) images. By injecting a patient with

radioactive tracer, two sets of SPECT images are obtained: one is taken 10–15 minutes after the injection when the stress is greatest (stress images), and the other is taken 2–5 hours after the injection (rest images). The distribution of the radioactive tracer in the cardiac muscle is proportional to the muscle's perfusion. Thus by comparing stress and rest images, a cardiologist can often detect abnormalities in the heart function.

The SPECT images are usually represented by high-resolution two-dimensional black-and-white pictures with up to 256 shades of grey. Brighter patches on the image correspond to well-perfused areas of the myocardium, while darker patches may indicate the presence of an ischemia. Unfortunately a visual inspection of the SPECT images is highly subjective; physicians' interpretations are therefore often inconsistent and susceptible to errors. Clearly an intelligent system that can help a cardiologist to diagnose cardiac SPECT images would be of great value.

For this study, we use 267 cardiac diagnostic cases. Each case is accompanied by two SPECT images (the stress image and the rest image), and each image is divided into 22 regions. The region's brightness, which in turn reflects perfusion inside this region, is expressed by an integer number between 0 and 100 (Kurgan *et al.*, 2001). Thus, each cardiac diagnostic case is represented by 44 continuous features and one binary feature that assigns an overall diagnosis – normal or abnormal.

The entire SPECT data set consists of 55 cases classified as normal (positive examples) and 212 cases classified as abnormal (negative examples). This set is divided into training and test sets. The training set has 40 positive and 40 negative examples. The test set is represented by 15 positive and 172 negative examples.

Can we train a back-propagation neural network to classify the SPECT images into normal and abnormal?

A back-propagation neural network can indeed address the SPECT image classification problem – the size of the training set appears to be sufficiently large, and the network can work here as a classifier. The number of neurons in the input layer is determined by the total number of regions in the stress and rest images. In this example, each image is divided into 22 regions, so we need 44 input neurons. Since SPECT images are to be classified as either normal or abnormal, we should use two output neurons. Experimental trials show that a good generalisation can be obtained with as little as 5 to 7 neurons in the hidden layer. The back-propagation network learns relatively fast and converges to a solution.

However, when we test the network on the test set, we find that the network's performance is rather poor – about 25 per cent of normal cardiac diagnostic cases are misclassified as abnormal and over 35 per cent of abnormal cases are misclassified as normal; the overall diagnostic error exceeds 33 per cent. This indicates that the training set may lack some important examples (a neural network is only as good as the examples used to train it). Despite this, we still can significantly improve the accuracy of the diagnosis.

First, we need to redefine the problem. To train the network, we use the same number of positive and negative examples. Although in real clinical trials the ratio between normal and abnormal SPECT images is very different, the misclassification of an abnormal cardiac case could lead to infinitely more serious consequences than the misclassification of a normal case. Therefore, in order to achieve a small classification error for abnormal SPECT images, we might agree to have a relatively large error for normal images.

The neural network produces two outputs. The first output corresponds to the possibility that the SPECT image belongs to the class *normal*, and the second to the possibility that the image belongs to the class *abnormal*. If, for example, the first (*normal*) output is 0.92 and the second (*abnormal*) is 0.16, the SPECT image is classified as normal, and we can conclude that the risk of a heart attack for this case is low. On the other hand, if the *normal* output is low, say 0.17, and the *abnormal* output is much higher, say 0.51, the SPECT image is classified as abnormal, and we can infer that the risk of a heart attack in this case is rather high. However, if the two outputs are close – say the *normal* output is 0.51 and the *abnormal* 0.49 – we cannot confidently classify the image.

Can we use fuzzy logic for decision-making in medical diagnosis?

Doctors do not keep precise thresholds in mind when they classify SPECT images. A cardiologist examines perfusion across all regions in the diagnosed image and also compares the brightness of the corresponding myocardium regions on the stress and rest images. In fact, doctors often rely on their experience and intuition in detecting abnormalities of the myocardium. Fuzzy logic provides us with a means of modelling how the cardiologist assesses the risk of a heart attack.

To build a fuzzy system, we first need to determine input and output variables, define fuzzy sets and construct fuzzy rules. For our problem, there are two inputs (*NN output 1* and *NN output 2*) and one output (the *risk* of a heart attack). The inputs are normalised to be within the range of [0, 1], and the output can vary between 0 and 100 per cent. Figures 9.33, 9.34 and 9.35 demonstrate fuzzy sets of the linguistic variables used in the fuzzy system. Fuzzy rules are shown in Figure 9.36.

Figure 9.37 represents a complete structure of the neuro-fuzzy decision-support system for assessing the risk of a cardiac decease. To build this system, we can use the MATLAB Neural Network and MATLAB Fuzzy Logic Toolboxes.
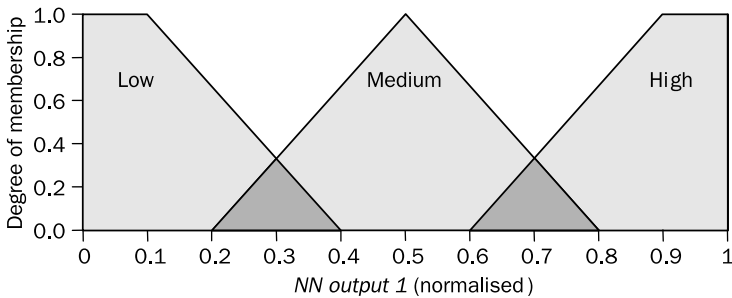
Figure 9.33   Fuzzy sets of the neural network output *normal*
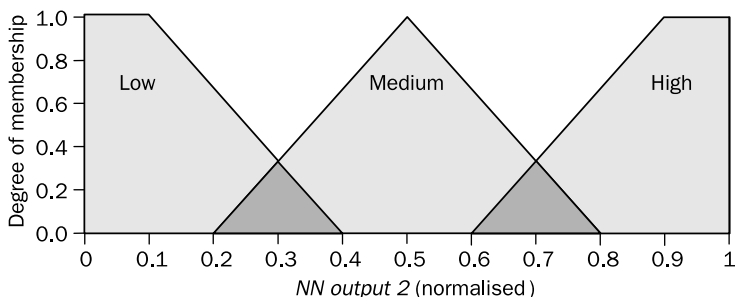
Figure 9.34 Fuzzy sets of the neural network output *abnormal*
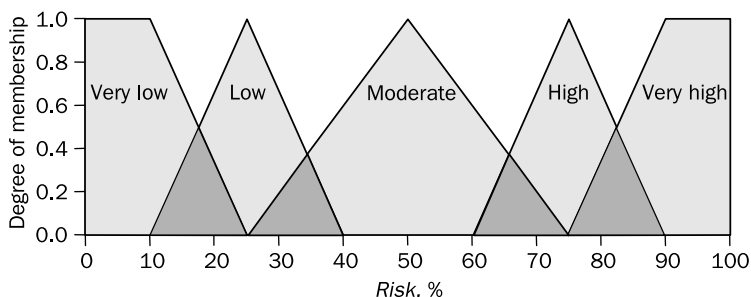
Figure 9.35 Fuzzy sets of the linguistic variable *Risk*

Once the system is developed, we can study and analyse its behaviour on the three-dimensional plot shown in Figure 9.38.

The system's output is a crisp number that represents a patient's risk of a heart attack. Based on this number, a cardiologist can now classify cardiac cases with greater certainty – when the risk is quantified, a decision-maker has a much better chance of making the right decision. For instance, if the risk is low, say, smaller than 30 per cent, the cardiac case can be classified as *normal*, but if the risk is high, say, greater than 50 per cent, the case is classified as *abnormal*.

However, cardiac cases with the risk between 30 and 50 per cent cannot be classified as either *normal* or *abnormal* – rather, such cases are *uncertain*.

1. If (NN_output1 is Low) and (NN_output2 is Low) then (Risk is Moderate)
2. If (NN_output1 is Low) and (NN_output2 is Medium) then (Risk is High)
3. If (NN_output1 is Low) and (NN_output2 is High) then (Risk is Very_high)
4. If (NN_output1 is Medium) and (NN_output2 is Low) then (Risk is Low)
5. If (NN_output1 is Medium) and (NN_output2 is Medium) then (Risk is Moderate)
6. If (NN_output1 is Medium) and (NN_output2 is High) then (Risk is High)
7. If (NN_output1 is High) and (NN_output2 is Low) then (Risk is Very_low)
8. If (NN_output1 is High) and (NN_output2 is Medium) then (Risk is Low)
9. If (NN_output1 is High) and (NN_output2 is High) then (Risk is Moderate)

Figure 9.36 Fuzzy rules for assessing the risk of a heart decease

Can we still classify at least some of these uncertain cases using the knowledge of an experienced cardiologist?

An experienced cardiologist knows that, in normal heart muscle, perfusion at maximum stress is usually higher than perfusion at rest for the same region of the muscle. Thus we can expect to make an uncertain case more certain when we apply the following heuristics to all corresponding regions:

1. If perfusion inside region *i* at stress is higher than perfusion inside the same region at rest, then the risk of a heart attack should be decreased.
2. If perfusion inside region *i* at stress is not higher than perfusion inside the same region at rest, then the risk of a heart attack should be increased.
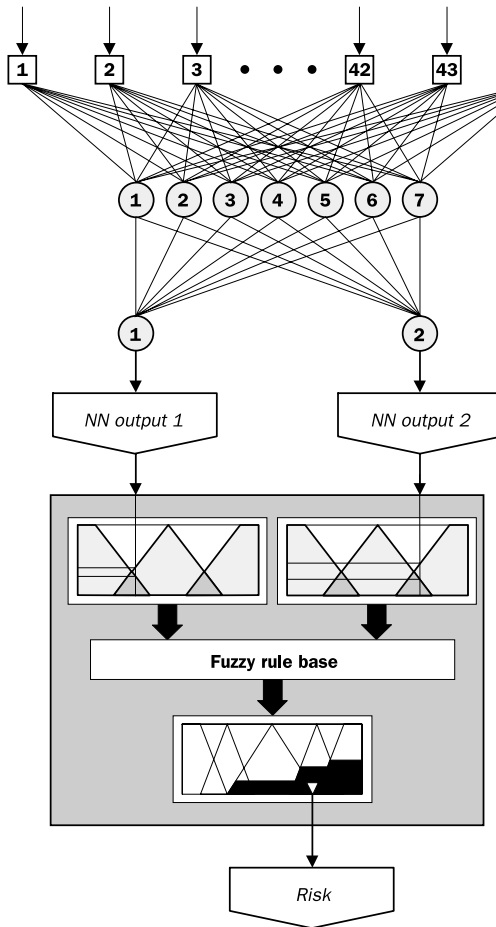
Figure 9.37   Hierarchical structure of the neuro-fuzzy system for risk assessment of the cardiac decease
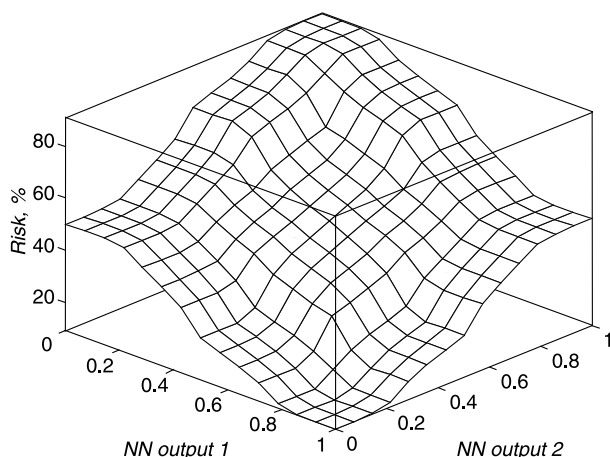
Figure 9.38 Three-dimensional plot for the fuzzy rule base

These heuristics can be implemented in the diagnostic system as follows:

Step 1: Present the neuro-fuzzy system with a cardiac case.

Step 2: If the system's output is less than 30, classify the presented case as *normal* and then stop. If the output is greater than 50, classify the case as *abnormal* and stop. Otherwise, go to Step 3.

Step 3: For region 1, subtract perfusion at rest from perfusion at stress. If the result is positive, decrease the current risk by multiplying its value by 0.99. Otherwise, increase the risk by multiplying its value by 1.01. Repeat this procedure for all 22 regions and then go to Step 4.

Step 4: If the new risk value is less than 30, classify the case as *normal*; if the risk is greater than 50, classify the case as *abnormal*; otherwise, classify the case as *uncertain*.

When we now apply the test set to the neuro-fuzzy system, we find that the accuracy of diagnosis has dramatically improved – the overall diagnostic error does not exceed 5 per cent, while only 3 per cent of abnormal cardiac cases are misclassified as normal. Although we have not improved the system's perform-ance on normal cases (over 30 per cent of normal cases are still misclassified as abnormal), and up to 20 per cent of the total number of cases are classified as uncertain, the neuro-fuzzy system can actually achieve even better results in classifying SPECT images than a cardiologist can. Most importantly, the abnor-mal SPECT images can now be recognised with much greater accuracy.

In this example, the neuro-fuzzy system has a heterogeneous structure – the neural network and fuzzy system work as independent components (although they cooperate in solving the problem). When a new case is presented to the diagnostic system, the trained neural network determines inputs to the fuzzy

system. Then the fuzzy system, using predefined fuzzy sets and fuzzy rules, maps the given inputs to an output, and thereby obtains the risk of a heart attack.

Are there any successful neuro-fuzzy systems with a homogeneous structure?

A typical example of a neuro-fuzzy system with a homogeneous structure is an Adaptive Neuro-Fuzzy Inference System (ANFIS). It cannot be divided into two independent and distinct parts. In fact, an ANFIS is a multilayer neural network that performs fuzzy inferencing.

Case study 9: Time-series prediction

I want to develop a tool to predict an aircraft's trajectory during its landing aboard an aircraft carrier. I have a database of landing trajectories of various aircraft flown by different pilots, and I also can use RADAR numerical data, which provide real-time trajectories of landing aircraft. My goal is to predict an aircraft's trajectory at least 2 seconds in advance, based on the aircraft's current position. Will a neuro-fuzzy system work for this problem?

The landing of an aircraft, particularly aboard aircraft carriers, is an extremely complex process. It is affected by such variables as the flight deck's space constraints and its motions (both pitch and roll), the aircraft's ordinance and fuel load, continuous mechanical preparations, and the most critical of all – time constraints. The ship may be heaving 10 feet up and 10 feet down, making a 20-foot displacement from a level deck. In addition, it is difficult to see approaching aircraft at night or during stormy conditions.

Responsibility for the aircraft's final approach and landing lies with the Landing Signal Officer (LSO). In fact, the LSO, not the pilot, makes the most important decision to wave-off (i.e. abort landing). When an aircraft is within 1 nautical mile of the landing deck, which roughly corresponds to 60 seconds in real time, the aircraft's flight is carefully observed and guided. During this critical time, the LSO needs to predict the aircraft's position at least 2 seconds ahead. Such problems are known in mathematics as **time-series prediction** problems.

What is a time series?

A time series can be defined as a set of observations, each one being recorded at a specific time. For instance, a time series can be obtained by recording the aircraft's positions over a time interval of, say, 60 seconds before landing. Real-world time-series problems are non-linear and often exhibit chaotic behaviour, which make them hard to model.

Prediction of the aircraft's landing trajectory is mainly based on the experience of the LSO (all LSOs are trained pilots). An automatic prediction system can use aircraft-position data given by the ship's RADAR, and also data records of previous landings executed by pilots flying different types of aircraft (Richards, 2002). The system is trained off-line with the past data. Then it is presented
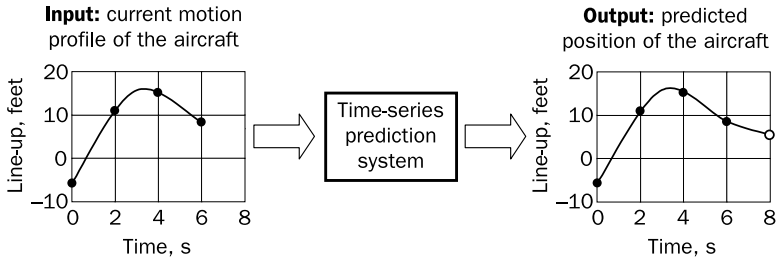
Figure 9.39 On-line time-series predication of an aircraft's trajectory

on-line with the current motion profile, and required to predict the aircraft's motion in the next few seconds. A time-series prediction of an aircraft's trajectory is shown in Figure 9.39.

To predict an aircraft's position on-line we will use an ANFIS. It will learn from time-series data of given landing trajectories in order to determine the membership function parameters that best allow the system to track these trajectories.

What do we use as ANFIS inputs?

To predict a future value for a time series, we use values that are already known. For example, if we want to predict an aircraft's position 2 seconds ahead, we may use its current position data as well as data recorded, say, 2, 4 and 6 seconds before the current position. These four known values represent an input pattern – a four-dimensional vector of the following form:

$$\mathbf{x} = [x(t-6) \quad x(t-4) \quad x(t-2) \quad x(t)],$$

where $x(t)$ is the aircraft position recorded at the point in time t.

The ANFIS output corresponds to the trajectory prediction: the aircraft's position 2 seconds ahead, $x(t+2)$.

For this case study, we will use 10 landing trajectories – five for training and five for testing. Each trajectory is a time series of the aircraft's position data points recorded every half a second over a time interval of 60 seconds before landing. Thus, a data set for each trajectory contains 121 values.

How do we build a data set to train the ANFIS?

Let us consider Figure 9.40; it shows an aircraft trajectory and a 3×5 training data set created from the trajectory data points sampled every 2 seconds. Input variables x_1, x_2, x_3 and x_4 correspond to the aircraft's flight positions at $(t-6)$, $(t-4)$, $(t-2)$ and t, respectively. The desired output corresponds to the two-second-ahead prediction, $x(t+2)$. The training data set shown in Figure 9.40 is built with t equal to 6.0 s (the first row), 6.5 s (the second row) and 7.0 s (the third row).

By applying the same procedure to a landing trajectory recorded over a time interval of 60 seconds, we obtain 105 training samples represented by a 105×5
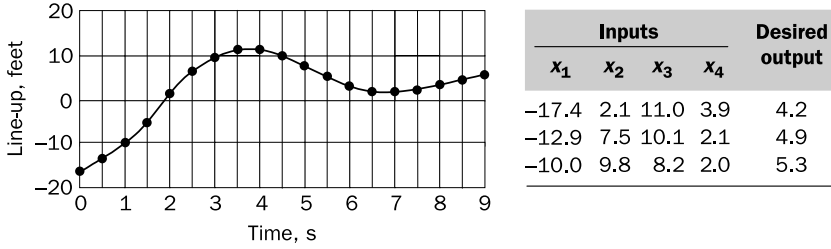
Figure 9.40   An aircraft trajectory and a data set built to train the ANFIS

matrix. Thus, the entire data set, which we use for training the ANFIS, is represented by a 525×5 matrix.

How many membership functions should we assign to each input variable?
A practical approach is to choose the smallest number of membership functions. Thus, we may begin with two membership functions assigned to each input variable.

Figure 9.41 shows an actual aircraft trajectory and the ANFIS's output after 1 and 100 epochs of training. As can be seen, even after 100 epochs, the ANFIS's

(a)

(b)
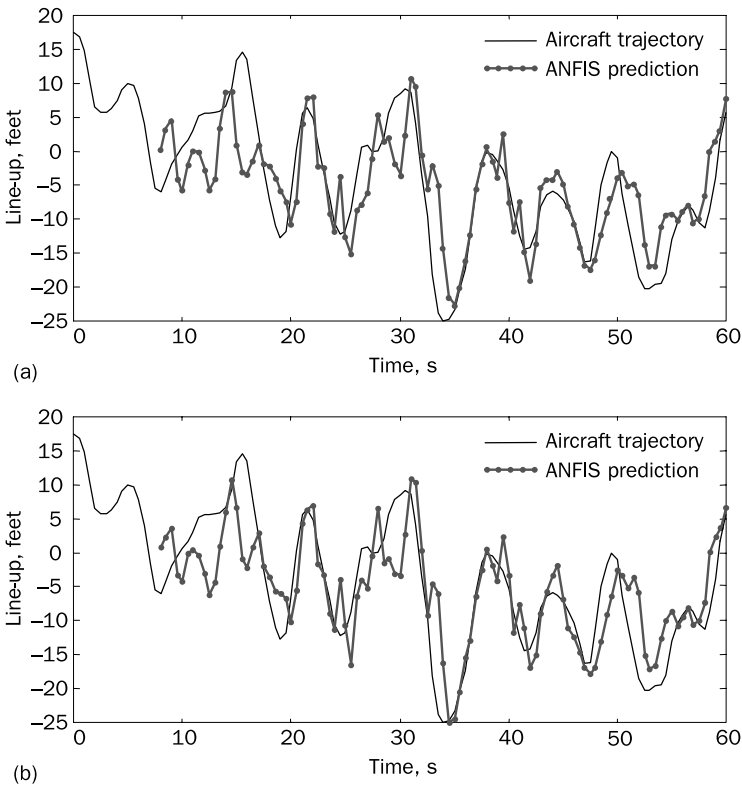
Figure 9.41   Performance of the ANFIS with four inputs and two membership functions assigned to each input: (a) one epoch; (b) 100 epochs
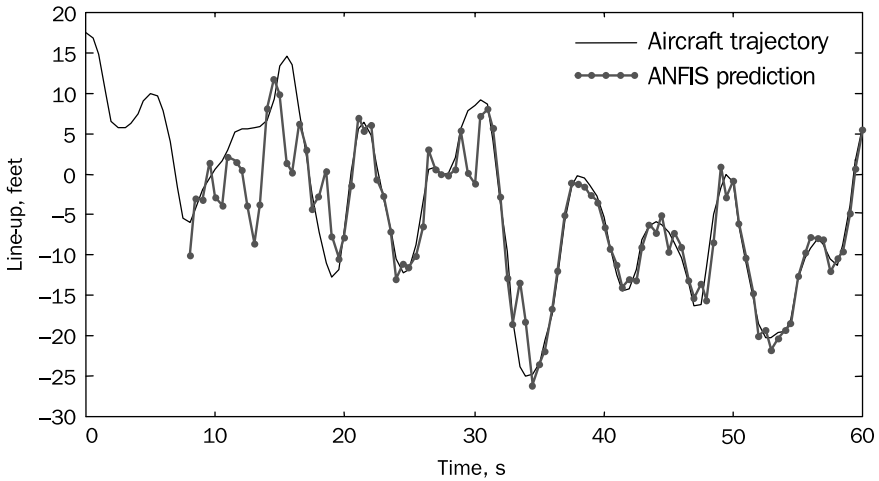
Figure 9.42   Performance of the ANFIS with four inputs and three membership functions assigned to each input after one epoch of training

performance is still unsatisfactory. We can also see that we have not achieved any significant improvement by increasing the number of epochs.

How can we improve the ANFIS's performance?
The ANFIS's performance can be significantly improved by assigning three membership functions to each input variable. Figure 9.42 demonstrates the results: after only one epoch of training the ANFIS predicts the aircraft's trajectory more accurately than it did in the previous study after 100 epochs.

Another way of improving time-series prediction is to increase the number of input variables. Let us, for example, examine an ANFIS with six inputs that correspond to the aircraft's flight positions at $(t-5)$, $(t-4)$, $(t-3)$, $(t-2)$, $(t-1)$ and t, respectively. The ANFIS output is still the two-second-ahead prediction. The training data set is now represented by a 535×7 matrix.

Once we assign the smallest number of membership functions to each input variable, we can train the ANFIS for 1 epoch and observe its performance on a test trajectory. Figure 9.43 shows the results. As we can see, the six-input ANFIS outperforms the ANFIS with four inputs, and provides satisfactory predictions after just one epoch of training.

9.7   Data mining and knowledge discovery

Data is what we collect and store, and knowledge is what helps us to make informed decisions. The extraction of knowledge from data is called **data mining**. Data mining can also be defined as the exploration and analysis of **large** quantities of data in order to discover meaningful patterns and rules (Berry and Linoff, 2000). The ultimate goal of data mining is to discover knowledge.

(a)

(b)
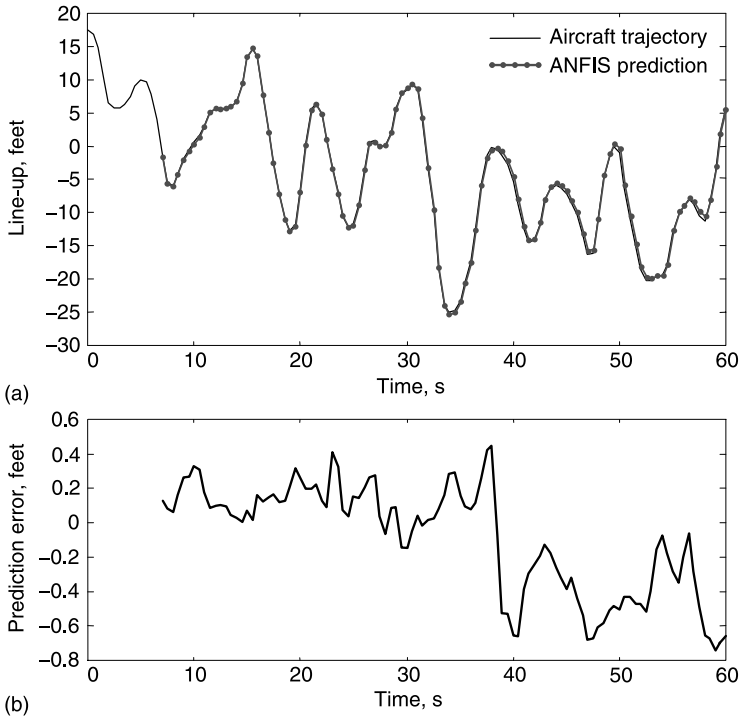
Figure 9.43   Performance of the ANFIS with six inputs and two membership functions assigned to each input: (a) prediction after one epoch; (b) prediction errors

We live in a rapidly expanding universe of data. The quantity of data in the modern world roughly doubles every year, and we often have enormous difficulties in finding the information we need in huge amounts of data. NASA, for example, has more data than it can analyse. Human Genome Project researchers have to store and process thousands of bytes for each of the three billion DNA bases that make up the human genome. Every day hundreds of megabytes of data are circulated via the Internet, and we need methods that can help us to extract meaningful information and knowledge from it.

Data mining is often compared with gold mining. Large quantities of ore must be processed before the gold can be extracted. Data mining can help us to find the 'hidden gold' of knowledge in raw data. Data mining is fast becoming essential to the modern competitive business world.

Modern organisations must respond quickly to any change in the market. This requires rapid access to current data normally stored in operational databases. However, an organisation must also determine which trends are relevant, and this cannot be accomplished without access to historical data that are stored in large databases called **data warehouses**.

What is a data warehouse?

The main characteristic of a data warehouse is its capacity. A data warehouse is really big – it includes millions, even billions, of data records. The data stored

in a data warehouse is **time dependent** – linked together by the times of recording – and **integrated** – all relevant information from the operational databases is combined and structured in the warehouse (Adriaans and Zantinge, 1996).

A data warehouse is designed to support decision making in the organisation. The information needed can be obtained with traditional query tools. These tools might also help us in discovering important relationships in the data.

What is the difference between a query tool and data mining?

Traditional query tools are **assumption**-based – a user must ask the **right** questions. Let us consider an example. Suppose we obtained data from a study on high blood pressure. Such data normally includes information on each person's age, gender, weight and height, sport activities, and smoking and drinking habits. With a query tool, a user can select a specific variable, say smoking, that might affect the outcome, in our case, blood pressure. The user's aim here is to compare the number of smokers and non-smokers among people with high blood pressure. However, by selecting this variable, the user **makes an assumption** (or even knows) that there is a strong correlation between high blood pressure and smoking.

With a data mining tool, instead of assuming certain relationships between different variables in a data set (and studying these relationships one at a time), we can determine the most significant factors that influence the outcome. Thus, instead of assuming a correlation between blood pressure and smoking, we can automatically identify the most significant risk factors. We can also examine different groups, or clusters, of people with high blood pressure. Data mining does not need any hypotheses – it discovers hidden relationships and patterns automatically.

The structured representation of data in a data warehouse facilitates the process of data mining.

How is data mining applied in practice?

Although data mining is still largely a new, evolving field, it has already found numerous applications in banking, finance, marketing and telecommunication. Many companies use data mining today, but refuse to talk about it. A few areas in which data mining is used for strategic benefits are direct marketing, trend analysis and fraud detection (Groth, 1998; Cabena *et al*., 1998).

In direct marketing, data mining is used for targeting people who are most likely to buy certain products and services. In trend analysis, it is used to determine trends in the marketplace, for example, to model the stock market. In fraud detection, data mining is used to identify insurance claims, cellular phone calls and credit card purchases that are most likely to be fraudulent.

How do we mine data?

Long before recorded history, people were gathering and analysing data. They observed the sun, the moon and the stars and discovered patterns in their movements; as a result, they created calendars.

Traditionally, data has been analysed with user-driven techniques, where a user formulates a hypothesis and then tests and validates it with the available data. A query tool is, in fact, one such technique. However, as we already know, the success of a query tool in discovering new knowledge is largely based on the user's ability to hypothesise, or in other words, on the user's hunch. Moreover, even experts are not capable of correlating more than three or, at best, four variables, while in reality, a data warehouse may include dozens of variables, and there may be hundreds of complex relationships among these variables.

Can we use statistics to make sense of the data?

Statistics is the science of collecting, organising and utilising numerical data. It gives us general information about data: the average and median values, distribution of values, and observed errors. Regression analysis – one of the most popular techniques for data analysis – is used to interpolate and extrapolate observed data.

Statistics is useful in analysing numerical data, but it does not solve data mining problems, such as discovering meaningful patterns and rules in large quantities of data.

What are data mining tools?

Data mining is based on intelligent technologies already discussed in this book. It often applies such tools as neural networks and neuro-fuzzy systems. However, the most popular tool used for data mining is a **decision tree**.

What is a decision tree?

A decision tree can be defined as a map of the reasoning process. It describes a data set by a tree-like structure. Decision trees are particularly good at solving classification problems.

Figure 9.44 shows a decision tree for identifying households that are likely to respond to the promotion of a new consumer product, such as a new banking service. Typically, this task is performed by determining the demographic characteristics of the households that responded to the promotion of a similar product in the past. Households are described by their owner-ship, income, type of bank accounts, etc. One field in the database (named *Household*) shows whether a household responded to the previous promotion campaign.

A decision tree consists of **nodes**, **branches** and **leaves**. In Figure 9.44, each box represents a node. The top node is called the **root node**. The tree always starts from the root node and grows down by splitting the data at each level into new nodes. The root node contains the entire data set (all data records), and child nodes hold respective subsets of that set. All nodes are connected by **branches**. Nodes that are at the end of branches are called **terminal nodes**, or **leaves**.

Each node contains information about the total number of data records at that node, and the distribution of values of the **dependent variable**.
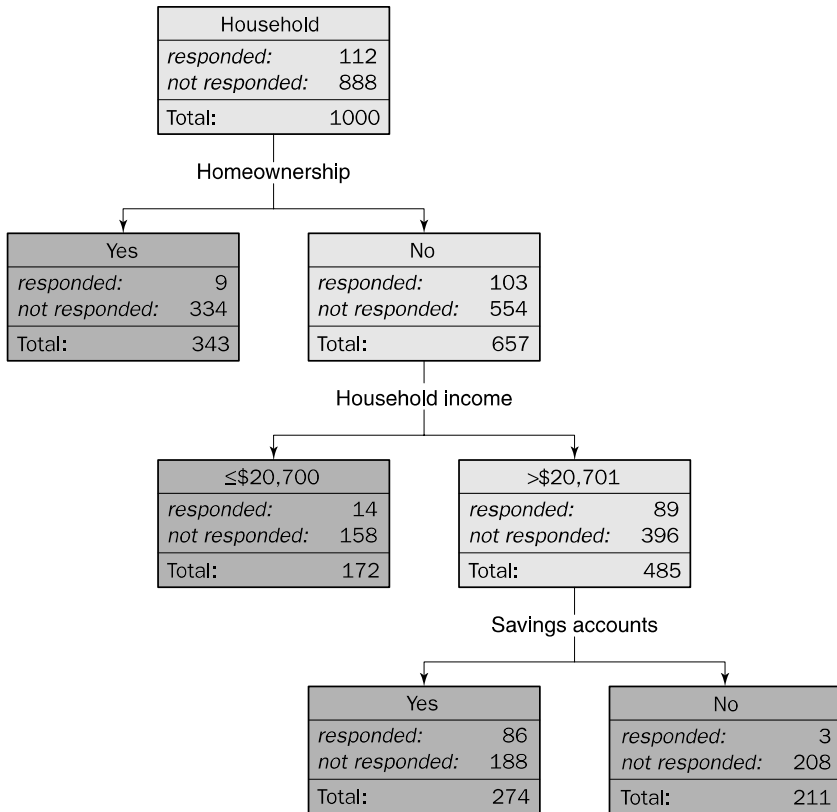
Figure 9.44   An example of a decision tree

What is the dependent variable?

The dependent variable determines the goal of the study; it is chosen by the user. In our example, *Household* is set up as the dependent variable, and it can have a value of either *responded* or *not responded*.

Below the root node we find the next level of the tree. Here, the tree selects variable *Homeownership* as a **predictor** for the dependent variable, and separates all households according to the predictor's values. The separation of data is called a **split**. In fact, *Homeownership* is just one of the fields in the database.

How does the decision tree select a particular split?

A split in a decision tree corresponds to the predictor with the maximum separating power. In other words, the best split does the best job in creating nodes where a single class dominates.

In our example *Homeownership* best splits households that responded to the previous promotion campaign from those that did not. In Figure 9.44, we can see that, while only 11.2 per cent of all households responded, a great majority of them were **not** home owners.

There are several methods of calculating the predictor's power to separate data. One of the best known methods is based on the Gini coefficient of inequality.

What is the Gini coefficient?

The Gini coefficient is, essentially, a measure of how well the predictor separates the classes contained in the parent node.

Corrado Gini, an Italian economist, introduced a rough measure of the amount of inequality in the income distribution in a country. Computation of the Gini coefficient is illustrated in Figure 9.45. The diagonal corresponds to an absolutely equal distribution of wealth, and the curve above it represents a real economy, where there is always some inequality in the income distribution. The curve's data is ordered from the richest to the poorest members of the society. The Gini coefficient is calculated as the area between the curve and the diagonal divided by the area below the diagonal. For a perfectly equal wealth distribution, the Gini coefficient is equal to zero. For complete inequality when only one person has all the income, the Gini coefficient becomes unity.

Classification and Regression Trees (CART) use the Gini's measure of inequality for selecting splits (Breiman *et al.*, 1984). Let us compare two alternative trees shown in Figure 9.46. Suppose, at the root node, we have two classes, *Class A* and *Class B*. A decision tree strives to isolate the largest class, that is, to pull out the data records of *Class A* into a single node. This ideal, however, can rarely be achieved; in most cases, a database field that clearly separates one class from the others does not exist. Therefore, we need to choose among several alternative splits.

A tree shown in Figure 9.46(a) is grown automatically with splits being selected by the Gini measure of inequality. In Figure 9.46(b), we select the splits using our own judgements or informed guesswork. The resulting trees are compared on a **gain chart** (also called a **lift chart**) shown in Figure 9.47. The chart maps the cumulative percentage of instances of *Class A* at a terminal node to the cumulative percentage of the total population at the same node. The diagonal line here represents the outcome if each terminal node contained a
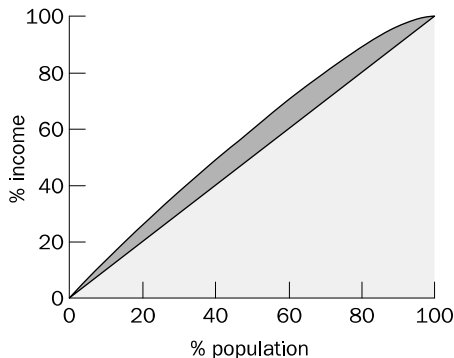
Figure 9.45 Computation of the Gini coefficient of inequality
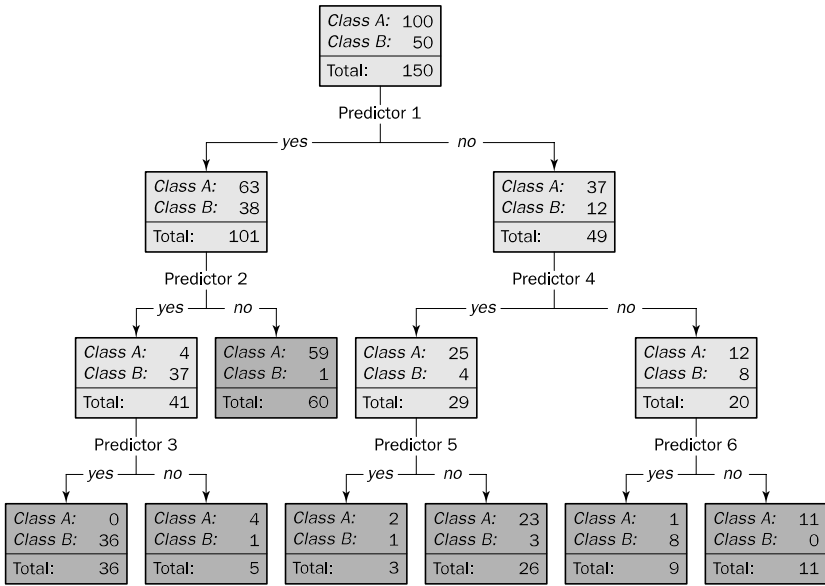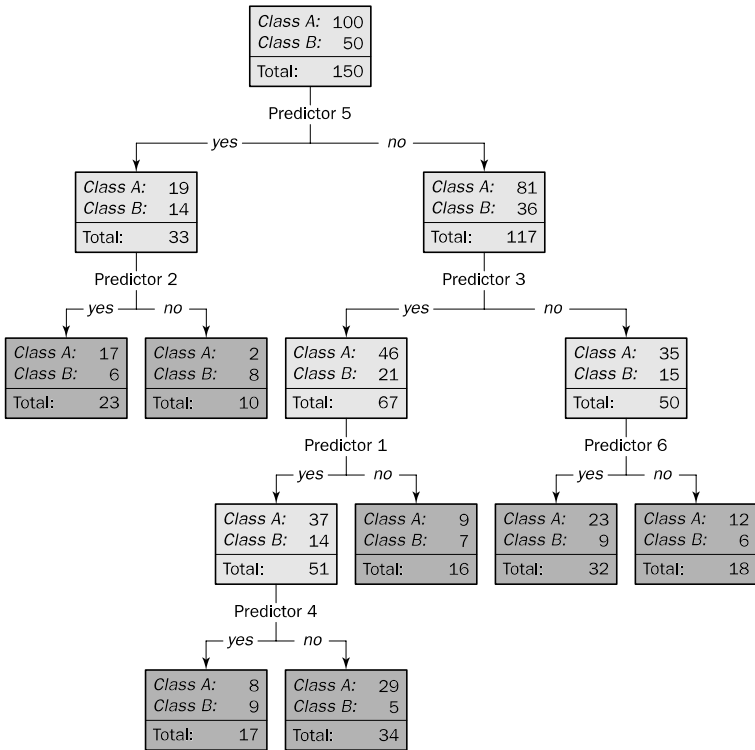
Figure 9.46 Selecting an optimal decision tree: (a) splits selected by Gini; (b) splits selected by guesswork
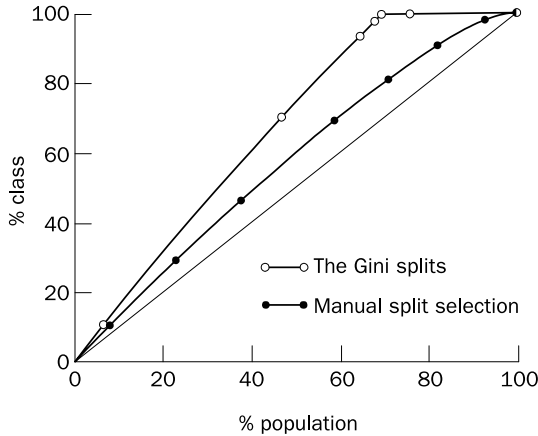
Figure 9.47   Gain charts of *Class A*

random sample of the population. The results clearly demonstrate the advantages of the tree constructed with the Gini splits.

Can we extract rules from a decision tree?

The pass from the root node to the bottom leaf reveals a decision rule. For example, a rule associated with the right bottom leaf in Figure 9.46(a) can be represented as follows:

if   (Predictor 1 = *no*)
and   (Predictor 4 = *no*)
and   (Predictor 6 = *no*)
then   class = *Class A*

Case study 10: Decision trees for data mining

I have the results of a community health survey, and I want to understand which people are at a greater risk of having high blood pressure. Will decision trees work for this problem?

A typical task for decision trees is to determine conditions that may lead to certain outcomes. This makes decision trees a good choice for profiling people with high blood pressure, and community health surveys can provide us with the necessary data.

High blood pressure, also called hypertension, occurs when the body's smaller blood vessels narrow. This causes the heart to work harder to maintain the pressure, and although the body can tolerate increased blood pressure for months and even years, eventually the heart may fail.

Blood pressure can be categorised as optimal, normal or high. Optimal pressure is below 120/80, normal is between 120/80 and 130/85, and a hypertension is diagnosed when blood pressure is over 140/90. Figure 9.48 shows an example of a data set used in a hypertension study.

Community Health Survey: Hypertension Study (California, USA)	
Gender	☑ Male ☐ Female
Age	☐ 18–34 years ☐ 35–50 years ☑ 51–64 years ☐ 65 or more years
Race	☑ Caucasian ☐ African American ☐ Hispanic ☐ Asian or Pacific Islander
Marital status	☐ Married ☐ Separated ☑ Divorced ☐ Widowed ☐ Never married
Household income	☐ Less than $20,700 ☐ $20,701–$45,000 ☑ $45,001–$75,000 ☐ $75,001 and over
Alcohol consumption	☐ Abstain from alcohol ☐ Occasional (a few drinks per month) ☑ Regular (one or two drinks per day) ☐ Heavy (three or more drinks per day)
Smoking	☐ Nonsmoker ☐ 1–10 cigarettes per day ☑ 11–20 cigarettes per day ☐ More than one pack per day
Caffeine intake	☐ Abstain from coffee ☑ One or two cups per day ☐ Three or more cups per day
Salt intake	☐ Low-salt diet ☑ Moderate-salt diet ☐ High-salt diet
Physical activities	☐ None ☑ One or two times per week ☐ Three or more times per week
Height **Weight**	178 cm 93 kg
Blood pressure	☐ Optimal ☐ Normal ☑ High

Obesity	☑ Obese ☐ Not obese

Figure 9.48 A data set for a hypertension study

Decision trees are as good as the data they represent. Unlike neural networks and fuzzy systems, decision trees do not tolerate noisy and polluted data. Therefore, the data must be cleaned before we can start data mining.

Almost all databases are polluted to some degree. In a hypertension study, we might find that such fields as *Alcohol Consumption* or *Smoking* have been left blank or contain incorrect information. We must also check our data for possible inconsistencies and typos. However, no matter how hard we try, we can rarely remove all the pollution in advance – some abnormalities in the data can only be discovered during the data mining process itself.

We might also attempt to enrich the data. We have, for example, such variables as *weight* and *height*, from which we can easily derive a new variable, *obesity*. This variable is calculated with a body-mass index (BMI), that is, the weight in kilograms divided by the square of the height in metres. Men with BMIs of 27.8 or higher and women with BMIs of 27.3 or higher are classified as obese.

Once data for the hypertension study is prepared, we can choose a decision tree tool. In our study, we use **KnowledgeSEEKER** by Angoss – a comprehensive tool for building classification trees.

KnowledgeSEEKER starts a decision tree with the root node for the dependent variable *Blood Pressure* and divides all respondents into three categories: *optimal*, *normal* and *high*. In this study, 319 people (32 per cent) have optimal, 528 people (53 per cent) normal, and 153 people (15 per cent) high blood pressure.

Then KnowledgeSEEKER determines the influence of each variable on blood pressure, and makes a ranked list of the most important variables. In our study, *age* emerges at the top of the list, and KnowledgeSEEKER creates the next level of the tree by splitting respondents by their age, as shown in Figure 9.49. As we can see, the risk of high blood pressure increases as one ages. Hypertension is significantly more prevalent after age 50.

We grow the tree by creating new splits. Let us, for example, make the second level node for age group 51–64. KnowledgeSEEKER splits this group by *Obesity*. This is because, in our example, *Obesity* is found to be a key indicator of whether someone of age 51 to 64 has high blood pressure. In Figure 9.49, we can see that 48 per cent of obese individuals in this group suffer from hypertension. In fact, the increase in blood pressure in an ageing population may be due primarily to weight gain.

As we continue growing the tree node by node, we might find that African Americans have a much higher risk of hypertension than any other group, and smoking and heavy drinking increase this risk even further.

Can we look at a specific split?

Decision tree tools, including KnowledgeSEEKER, allow us to look at any split. Figure 9.50 shows splits by *Gender* created for age groups 35–50 and 51–64. As you can see, the results reveal that a higher percentage of men than women have hypertension before age 51, but after that the ratio reverses, and women are more likely to have high blood pressure than are men.
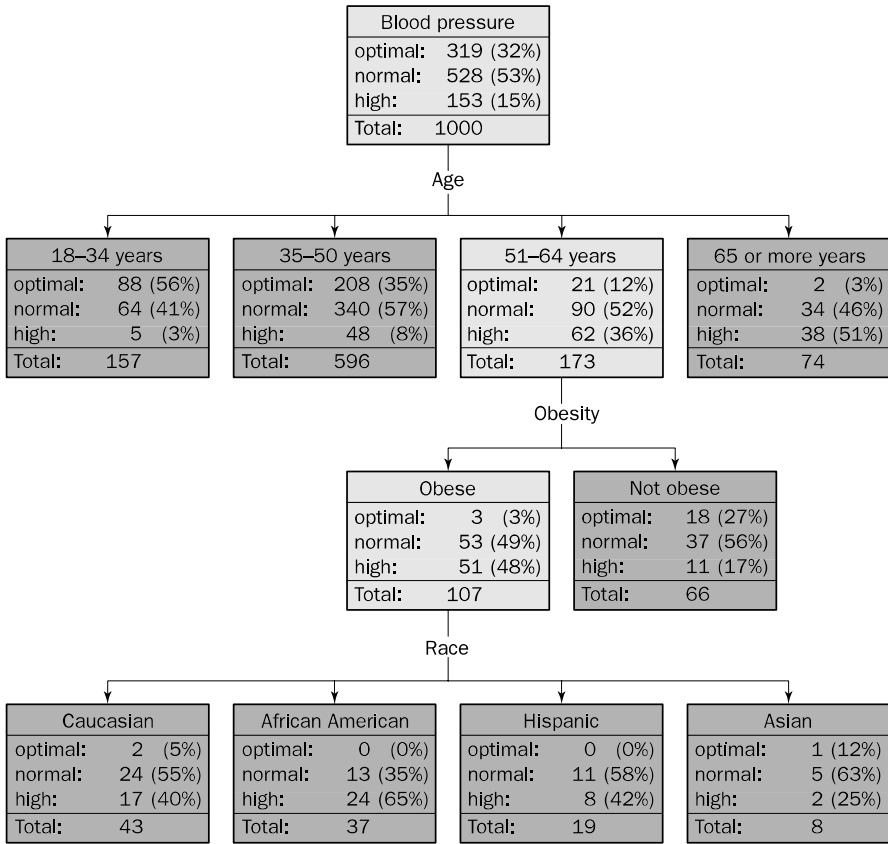
Figure 9.49   Hypertension study: growing a decision tree

The main advantage of the decision-tree approach to data mining is that it visualises the solution; it is easy to follow any path through the tree. Relationships discovered by a decision tree can be expressed as a set of rules, which can then be used in developing an expert system.

Decision trees, however, have several drawbacks. Continuous data, such as age or income, have to be grouped into ranges, which can unwittingly hide important patterns.

Another common problem is handling of missing and inconsistent data – decision trees can produce reliable outcomes only when they deal with 'clean' data.

However, the most significant limitation of decision trees comes from their inability to examine more than one variable at a time. This confines trees to only the problems that can be solved by dividing the solution space into several successive rectangles. Figure 9.51 illustrates this point. The solution space of the hypertension study is first divided into four rectangles by *age*, then age group 51–64 is further divided into those who are overweight and those who are not.
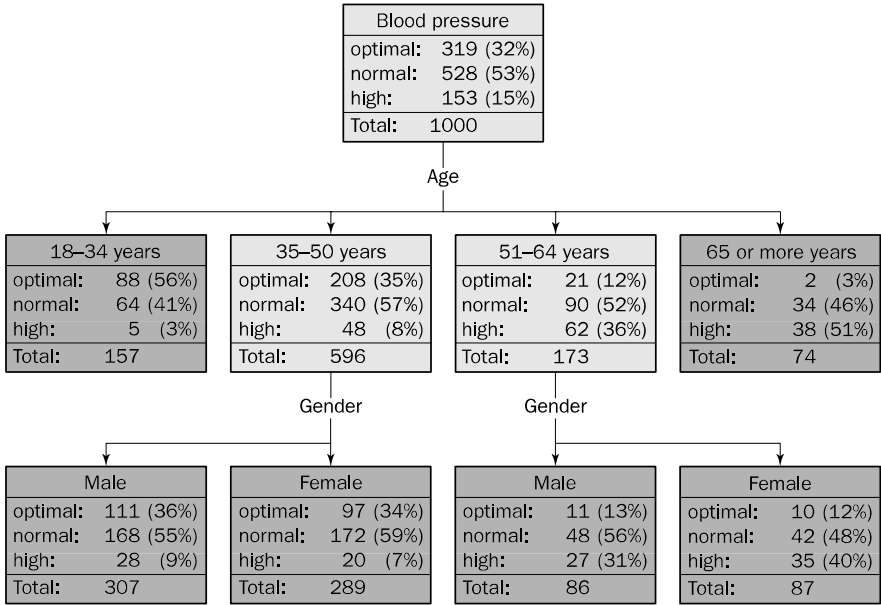
Figure 9.50 Hypertension study: forcing a split

And finally, the group of obese people is divided by *race*. Such a 'rectangular' classification may not correspond well with the actual distribution of data. This leads to data fragmentation, when the tree is so large and the amount of data passing from the root node to the bottom leaves is so small that discovering meaningful patterns and rules becomes difficult. To minimise fragmentation, we often need to trim back some of the lower nodes and leaves.

In spite of all these limitations, decision trees have become the most successful technology used for data mining. An ability to produce clear sets of rules make decision trees particularly attractive to business professionals.
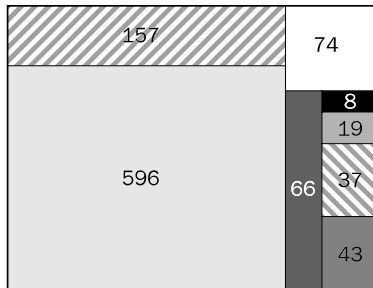
Figure 9.51 Solution space of the hypertension study

9.8 Summary

In this chapter, we considered knowledge engineering and data mining. First we discussed what kind of problems can be addressed with intelligent systems and introduced six main phases of the knowledge engineering process. Then we studied typical applications of intelligent systems, including diagnosis, classification, decision support, pattern recognition and prediction. Finally, we examined an application of decision trees in data mining.

The most important lessons learned in this chapter are:

- Knowledge engineering is the process of building intelligent knowledge-based systems. There are six main steps: assess the problem; acquire data and knowledge; develop a prototype system; develop a complete system; evaluate and revise the system; and integrate and maintain the system.

- Intelligent systems are typically used for diagnosis, selection, prediction, classification, clustering, optimisation and control. The choice of a tool for building an intelligent system is influenced by the problem type, availability of data and expertise, and the form and content of the required solution.

- Understanding the problem's domain is critical for building an intelligent system. Developing a prototype system helps us to test how well we understand the problem and to make sure that the problem-solving strategy, the tool selected for building a system, and the techniques for representing acquired data and knowledge are adequate to the task.

- Intelligent systems, unlike conventional computer programs, are designed to solve problems that quite often do not have clearly defined 'right' and 'wrong' solutions. Therefore, the system is normally evaluated with test cases selected by the user.

- Diagnostic and troubleshooting problems are very attractive candidates for expert systems. Diagnostic expert systems are easy to develop because most diagnostic problems have a finite list of possible solutions, involve a limited amount of well-formalised knowledge, and usually take a human expert a short time to solve.

- Solving real-world classification problems often involves inexact and incomplete data. Expert systems are capable of dealing with such data by managing incrementally acquired evidence as well as information with different degrees of belief.

- Fuzzy systems are well suited for modelling human decision-making. Important decisions are often based on human intuition, common sense and experience, rather than on the availability and precision of data. Fuzzy technology provides us with a means of coping with the 'soft criteria' and 'fuzzy data'. Although decision-support fuzzy systems may include dozens, even hundreds, of rules, they can be developed, tested and implemented relatively quickly.

- Neural networks represent a class of general-purpose tools that are successfully applied to prediction, classification and clustering problems. They are used in such areas as speech and character recognition, medical diagnosis, process control and robotics, identifying radar targets, predicting foreign exchange rates and detecting fraudulent transactions. The areas of neural network applications are expanding very rapidly.

- Data mining is the extraction of knowledge from data. It can also be defined as the exploration and analysis of **large** quantities of data in order to discover meaningful patterns and rules. The ultimate goal of data mining is to discover knowledge.

- Although data mining is still largely a new, evolving field, it has already found numerous applications. In direct marketing, data mining is used for targeting people who are most likely to buy certain products and services. In trend analysis, it is used to identify trends in the marketplace by, for example, modelling the stock market. In fraud detection, data mining is used to identify insurance claims, cellular phone calls and credit card purchases that are most likely to be fraudulent.

- The most popular tool for data mining is a decision tree – a tool that describes a data set by a tree-like structure. Decision trees are particularly good at solving classification problems. The main advantage of the decision-tree approach to data mining is that it visualises the solution; it is easy to follow any path through the tree. The tree's ability to produce clear sets of rules makes it particularly attractive for business professionals.

Questions for review

1 What is knowledge engineering? Describe the main steps in knowledge engineering. Why is choosing the right tool for the job the most critical part of building an intelligent system?

2 What are the stages in the knowledge acquisition process? Why is knowledge acquisition often called a bottleneck of the process of knowledge engineering? How can the acquired data affect our choice of the system building tool?

3 What is a prototype? What is a test case? How do we test an intelligent system? What should we do if we have made a bad choice of system-building tool?

4 Why is adopting new intelligent technologies becoming problem-driven, rather than curiosity-driven, as it often was in the past?

5 What makes diagnosis and troubleshooting problems so attractive for expert system technology? What is a phone call rule?

6 How do we choose a tool to develop an expert system? What are the advantages of expert system shells? How do we choose an expert system shell for building an intelligent system?

7 Why are fuzzy systems particularly well suited for modelling human decision-making? Why does fuzzy technology have great potential in such areas as business and finance?

8 What is the basis for the popularity of neural networks? What are the most successful areas of neural network applications? Explain why and give examples.

9 Why do we need to massage data before using them in a neural network model? How do we massage the data? Give examples of massaging continuous and discrete data. What is 1 of N coding?

10 What is data mining? What is the difference between a query tool and data mining? What are data mining tools? How is data mining applied in practice? Give examples.

11 What is a decision tree? What are dependent variables and predictors? What is the Gini coefficient? How does a decision tree select predictors?

12 What are advantages and limitations of the decision-tree approach to data mining? Why are decision trees particularly attractive to business professionals?

References

Adriaans, P. and Zantinge, D. (1996). *Data Mining*. Addison-Wesley, Harlow.

Applegate, D., Bixby, R., Chvátal, V. and Cook, W. (2001). TSP cuts which do not conform to the template paradigm, *Computational Combinatorial Optimization*, M. Junger and D. Naddef, eds, Springer-Verlag, Berlin, pp. 261–304.

Berry, M. and Linoff, G. (1997). *Data Mining Techniques: For Marketing, Sales, and Customer Support*. John Wiley, New York.

Berry, M. and Linoff, G. (2000). *Mastering Data Mining*. John Wiley, New York.

Breiman, L., Friedman, J.H., Olshen, R.A. and Stone, C.J. (1984). *Classification and Regression Trees*. Wadsworth, Belmont, CA.

Cabena, P., Hadjinian, P., Stadler, R., Verhees, J. and Zanasi, A. (1998). *Discovering Data Mining: From Concept to Implementation*. Prentice Hall, Upper Saddle River, NJ.

Dantzig, G., Fulkerson, R. and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem, *Operations Research*, 2, 393–410.

Davis, L. (1985). Applying adaptive algorithms to epistatic domains, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, A. Joshi, ed., Morgan Kaufmann, Los Angeles, CA, pp. 162–164.

Davis, R. and King, J. (1977). An overview of production systems, *Machine Intelligence*, 8, 300–322.

Durkin, J. (1994). *Expert Systems Design and Development*. Prentice Hall, Englewood Cliffs, NJ.

Firebaugh, M. (1988). *Artificial Intelligence: A Knowledge-Based Approach*. Boyd & Fraser, Boston.

Fisher, R.A. (1950). *Contributions to Mathematical Statistics*. John Wiley, New York.

Flood, M.M. (1955). The traveling salesman problem, *Operations Research*, 4, 61–75.

Goldberg, D.E. (1989). *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, Boston.

Goldberg, D.E. and Lingle, R. (1985). Alleles, loci and the traveling salesman problem, *Proceedings of the 1st International Conference on Genetic Algorithms*, J.J. Grefenstette, ed., Lawrence Erlbaum Associates, Pittsburgh, PA, pp. 154–159.

Groth, R. (1998). *Data Mining: A Hands-On Approach for Business Professionals (Data Warehousing Institute Series)*. Prentice Hall, Upper Saddle River, NJ.

Güllich, H.-P. (1996). Fuzzy logic decision support system for credit risk evaluation, *EUFIT Fourth European Congress on Intelligent Techniques and Soft Computing*, pp. 2219–2223.

Haupt, R.L. and Haupt, S.E. (1998). *Practical Genetic Algorithms*. John Wiley, New York.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*, 2nd edn. Prentice Hall, Englewood Cliffs, NJ.

Hopfield, J.J. and Tank, D.W. (1985). Neural computation of decisions in optimization problems, *Biological Cybernetics*, 52, 141–152.

Kurgan, L.A., Cios, K.J., Tadeusiewicz, R., Ogiela, M. and Goodenday, L. (2001). Knowledge discovery approach to automated cardiac SPECT diagnosis, *Artificial Intelligence in Medicine*, 23(2), 149–169.

Laarhoven, P.J.M. and Aarts, E.H.L. (1987). *Simulated Annealing: The Theory and Application*. Kluwer Academic Publishers, Norwell, MA.

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B. (1985). *The Traveling Salesman Problem*. John Wiley, Chichester.

LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. and Jackel, L.D. (1990). Handwritten digit recognition with a back-propagation network, *Advances in Neural Information Processing Systems*, D.S. Touretzky, ed., Morgan Kaufmann, San Mateo, CA, vol. 2, pp. 396–404.

Martin, O., Otto, S.W. and Felten, E.W. (1991). Large-step Markov chains for the traveling salesman problem, *Complex Systems*, 5(3), 299–326.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolutionary Programs*, 3rd edn. Springer-Verlag, New York.

Michie, D. (1982). The state of the art in machine learning, *Introductory Readings in Expert Systems*, Gordon and Breach, New York, pp. 209–229.

Potvin, J.V. (1996). Genetic algorithms for the traveling salesman problem, *Annals of Operations Research*, 63, 339–370.

Principe, J.C., Euliano, N.R. and Lefebvre, W.C. (2000). *Neural and Adaptive Systems: Fundamentals Through Simulations*. John Wiley, New York.

Richards, R. (2002). Application of multiple artificial intelligence techniques for an aircraft carrier landing decision support tool, *Proceedings of the IEEE International Conference on Fuzzy Systems, FUZZ-IEEE'02*, Honolulu, Hawaii.

Russell, S.J. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*, 2nd edn. Prentice Hall, Englewood Cliffs, NJ.

Simon, R. (1987). The morning after, *Forbes*, October 19, pp. 164–168.

Tschoke, S., Lubling, R. and Monien, B. (1995). Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network, *Proceedings of the 9th IEEE International Parallel Processing Symposium*, Santa Barbara, CA, pp. 182–189.

Von Altrock, C. (1997). *Fuzzy Logic and NeuroFuzzy Applications in Business and Finance*. Prentice Hall, Upper Saddle River, NJ.

Waterman, D.A. (1986). *A Guide to Expert Systems*. Addison-Wesley, Reading, MA.

Widrow, B. and Stearns, S.D. (1985). *Adaptive Signal Processing*. Prentice Hall, Englewood Cliffs, NJ.

Zurada, J.M. (1992). *Introduction to Artificial Neural Systems*. West Publishing Company, St Paul, MN.

Glossary

The glossary entries are coded using the following abbreviations:

es = expert systems
fl   = fuzzy logic
nn = neural networks
ec = evolutionary computation
dm = data mining
ke = knowledge engineering

Action potential
An output signal (also called *nerve impulse*) of a biological neuron that does not lose strength over long distances. When an action potential occurs, the neuron is said to 'fire an impulse'. [nn]

Activation function
A mathematical function that maps the net input of a **neuron** to its output. Commonly used activation functions are: **step**, **sign**, **linear** and **sigmoid**. Also referred to as **Transfer function**. [nn]

Adaptive learning rate
A **learning rate** adjusted according to the change of **error** during **training**. If the error at the current **epoch** exceeds the previous value by more than a predefined ratio, the learning rate is decreased. However, if the error is less than the previous one, the learning rate is increased. The use of an adaptive learning rate accelerates **learning** in a **multilayer perceptron**. [nn]

Aggregate set
A **fuzzy set** obtained through **aggregation**. [fl]

Aggregation
The third step in **fuzzy inference**; the process of combining **clipped** or **scaled** consequent **membership functions** of all **fuzzy rules** into a single **fuzzy set** for each output variable. [fl]

Algorithm
A set of step-by-step instructions for solving a problem.

AND
A logical operator; when used in a **production rule**, it implies that all antecedents joined with AND must be true for the rule **consequent** to be true. [es]

Antecedent
A conditional statement in the IF part of a **rule**. Also referred to as **Premise**. [es]

a-part-of

An **arc** (also known as **'part-whole'**) that associates subclasses representing components with a superclass representing the whole. For example, an engine is *a-part-of* a car. [es]

Approximate reasoning

Reasoning that does not require a precise matching between the IF part of a **production rule** with the **data** in the **database**. [es]

Arc

A directed labelled link between nodes in a **semantic network** that indicates the nature of the connection between adjacent nodes. The most common arcs are **is-a** and **a-part-of**. [es]

Architecture

see **Topology**. [nn]

Artificial neural network (ANN)

An information-processing paradigm inspired by the structure and functions of the human brain. An ANN consists of a number of simple and highly interconnected processors, called **neurons**, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links that pass signals from one neuron to another. While in a biological neural network, learning involves adjustments to the **synapses**, ANNs learn through repeated adjustments of the **weights**. These weights store the **knowledge** needed to solve specific problems. [nn]

Artificial intelligence (AI)

The field of computer science concerned with developing machines that behave in a way that would be considered intelligent if observed in humans.

Assertion

A **fact** derived during **reasoning**. [es]

Associative memory

The type of memory that allows us to associate one thing with another. For example, we can recall a complete sensory experience, including sounds and scenes, when we hear only a few bars of music. We can also recognise a familiar face even in an unfamiliar environment. An associative **ANN** recalls the closest 'stored' training pattern when presented with a similar input pattern. The **Hopfield network** is an example of the associative **ANN**. [nn]

Attribute

A property of an **object**. For example, the object 'computer' might have such attributes as 'model', 'processor', 'memory' and 'cost'. [es]

Axon

A single long branch of a biological **neuron** that carries the output signal (**action potential**) from the cell. An axon may be as long as a metre. In an **ANN**, an axon is modelled by the neuron's output. [nn]

Backward chaining

An **inference technique** that starts with a hypothetical solution (a goal) and works backward, matching **rules** from the **rule base** with **facts** from the **database** until the **goal** is either verified or proven wrong. Also referred to as **Goal-driven reasoning**. [es]

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Back-propagation

see **Back-propagation algorithm**. [nn]

Back-propagation algorithm

The most popular method of **supervised learning**. The algorithm has two phases. First, a training input pattern is presented to the **input layer**. The network propagates the input pattern from layer to layer until the output pattern is generated by the **output layer**. If this pattern is different from the desired output, an **error** is calculated and then propagated backwards through the network from the output layer to the input layer. The **weights** are modified as the error is propagated. Also referred to as **Back-propagation**. [nn]

Bayesian reasoning

A statistical approach to uncertainty management in **expert systems** that propagates uncertainties throughout the system based on a **Bayesian rule** of evidence. [es]

Bayesian rule

A statistical method for updating the probabilities attached to certain **facts** in the light of new evidence. [es]

Bidirectional associative memory (BAM)

A class of **neural networks** that emulates characteristics of **associative memory**; proposed by Bart Kosko in the 1980s. The BAM associates patterns from one set to patterns from another set, and vice versa. Its basic architecture consists of two fully connected **layers** – an **input layer** and an **output layer**. [nn]

Bit

A binary digit. The smallest unit of information. **Data** stored in a computer is composed of bits. [ke]

Bit map

A representation of an image by rows and columns of dots. Bit maps can be stored, displayed and printed by a computer. Optical scanners are used to transform text or pictures on paper into bit maps. The scanner processes the image by dividing it into hundreds of **pixels** per inch and representing each pixel by either 1 or 0. [ke]

Black-box

A model that is opaque to its user; although the model can produce correct results, its internal relationships are not known. An example of a black-box is a **neural network**. To understand the relationships between outputs and inputs of a black-box, **sensitivity analysis** can be used. [ke]

Boolean logic

A system of logic based on Boolean algebra, named after George Boole. It deals with two truth values: 'true' and 'false'. The Boolean conditions of true and false are often represented by 0 for 'false' and 1 for 'true'.

Branch

A connection between **nodes** in a **decision tree**. [dm]

Building block

A group of **genes** that gives a **chromosome** a high **fitness**. According to the building block hypothesis, an optimal solution can be found by joining several building blocks together in a single chromosome. [ec]

---

dm = data mining   ke = knowledge engineering

Byte
A set of eight **bits** that represents the smallest addressable item of information in a modern computer. The information in a byte is equivalent to a letter in a word. One gigabyte is about 1,000,000,000 (2^{30} or 1,073,741,824) bytes, approximately equal to 1000 novels. [ke]

C
A general-purpose programming language, originally developed at Bell Labs along with the UNIX operating system.

C++
An object-oriented extension of **C**.

CART (Classification and Regression Trees)
A tool for **data mining** that uses **decision trees**. CART provides a set of **rules** that can be applied to a new data set for predicting outcomes. CART segments **data records** by creating binary splits. [dm]

Categorical data
The **data** that fits into a small number of discrete categories. For example, gender (male or female) or marital status (single, divorced, married or widowed). [ke]

Centroid technique
A **defuzzification** method that finds the point, called the *centroid* or *centre of gravity*, where a vertical line would slice the **aggregate set** into two equal masses. [fl]

Certainty factor
A number assigned to a **fact** or a **rule** to indicate the certainty or confidence one has that this fact or rule is valid. Also referred to as **Confidence factor**. [es]

Certainty theory
A theory for managing uncertainties in **expert systems** based on inexact **reasoning**. It uses **certainty factors** to represent the level of belief in a hypothesis given that a particular event has been observed. [es]

Child
see **Offspring**. [ec]

Child
In a **decision tree**, a child is a **node** produced by splitting the **data** of a node located at the preceding hierarchical level of the tree. A child node holds a subset of the data contained in its **parent**. [dm]

Chromosome
A string of **genes** that represent an individual. [ec]

Class
A group of **objects** with common **attributes**. *Animal*, *person*, *car* and *computer* are all classes. [es]

Class-frame
A frame that represents a **class**. [es]

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Clipping

A common method of correlating the **consequent** of a **fuzzy rule** with the truth value of the rule **antecedent**. The method is based on cutting the consequent **membership function** at the level of the antecedent truth. Since the top of the membership function is sliced, the clipped **fuzzy set** loses some information. [fl]

Cloning

Creating an **offspring** that is an exact copy of a **parent**. [ec]

Clustering

The process of dividing a heterogeneous group of objects into homogeneous subgroups. Clustering **algorithms** find groups of items that are similar in some respect. For example, clustering can be used by an insurance company to group customers according to age, assets, income and prior claims. [ke]

Coding

The process of transforming information from one scheme of representation to another. [ec]

Cognitive science

The interdisciplinary study of how **knowledge** is acquired and used. Its contributing disciplines include **artificial intelligence**, psychology, linguistics, philosophy, neuro-science, and education. Also, the study of **intelligence** and intelligent systems, with reference to intelligent behaviour as computation.

Common-sense

A general **knowledge** of how to solve real-world problems, usually obtained through practical experience. [ke]

Competitive learning

Unsupervised learning in which **neurons** compete among themselves such that only one neuron will respond to a particular input pattern. The neuron that wins the 'competition' is called the winner-takes-all neuron. **Kohonen self-organising feature maps** are an example of an **ANN** with competitive learning. [nn]

Complement

In classical **set theory**, the complement of set *A* is the set of elements that are not members of *A*. In the **fuzzy set theory**, the complement of a set is an opposite of this set. [fl]

Confidence factor

see **Certainty factor**. [es]

Conflict

A state in which two or more **production rules** match the **data** in the **database**, but only one rule can actually be fired in a given cycle. [es]

Conflict resolution

A method for choosing which **production rule** to fire when more than one rule can be **fired** in a given cycle. [es]

Conjunction

The logical operator **AND** that joins together two **antecedents** in a **production rule**. [es]

---

dm = data mining   ke = knowledge engineering

Connection

A link from one **neuron** to another to transfer signals. Also referred to as **synapse**, which is often associated with the **weight** that determines the strength of the transferred signal. [nn]

Consequent

A conclusion or action in the IF part of a **rule**. [es]

Continuous data

The **data** that takes an infinite number of possible values on some interval. Examples of continuous data include height, weight, household income, the living area of a house. Continuous variables are usually measurements, and do not have to be integers. [ke]

Convergence

An **ANN** is said to have converged when the **error** has reached a preset threshold indicating that the network has learned the task. [nn]

Convergence

A tendency of individuals in the population to be the same. A **genetic algorithm** is said to have converged when a solution has been reached. [ec]

Crossover

A **reproduction** operator that creates a new **chromosome** by exchanging parts of two existing chromosomes. [ec]

Crossover probability

A number between zero and one that indicates the probability of two chromosomes crossing over. [ec]

Darwinism

Charles Darwin's theory that states that **evolution** occurs through natural **selection**, coupled with random changes of inheritable characteristics. [ec]

Data

Facts, measurements, or observations. Also, a symbolic representation of facts, measurements, or observations. Data is what we collect and store.

Database

A collection of structured **data**. Database is the basic component of an **expert system**. [es]

Data-driven reasoning

see **Forward chaining**. [es]

Data cleaning

The process of detecting and correcting obvious errors and replacing missing **data** in a **database**. Also referred to as **Data cleansing**. [dm]

Data cleansing

see **Data cleaning**. [dm]

Data mining

The extraction of **knowledge** from **data**. Also, the exploration and analysis of large

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

amounts of data in order to discover meaningful patterns and **rules**. The ultimate goal of data mining is to discover knowledge. [dm]

Data record
A set of values corresponding to the **attributes** of a single **object**. A data record is a row in a **database**. Also referred to as **Record**. [dm]

Data visualisation
The graphical representation of **data** that helps the **user** in understanding the structure and meaning of the information contained in the data. Also referred to as **Visualisation**. [dm]

Data warehouse
A large **database** that includes millions, even billions, of **data records** designed to support decision-making in organisations. It is structured for rapid on-line queries and managerial summaries. [dm]

Decision tree
A graphical representation of a data set that describes the **data** by tree-like structures. A decision tree consists of **nodes**, **branches** and **leaves**. The tree always starts from the **root node** and grows down by splitting the data at each level into new nodes. Decision trees are particularly good at solving classification problems. Their main advantage is **data visualisation**. [dm]

Decision-support system
An interactive computer-based system designed to help a person or a group of people to make decisions in a specific **domain**. [es]

Deductive reasoning
Reasoning from the general to the specific. [es]

Defuzzification
The last step in **fuzzy inference**; the process of converting a combined output of **fuzzy rules** into a crisp (numerical) value. The input for the defuzzification process is the **aggregate set** and the output is a single number. [fl]

Degree of membership
A numerical value between 0 and 1 that represents the degree to which an element belongs to a particular **set**. Also referred to as **Membership value**. [fl]

Delta rule
A procedure for updating **weights** in a **perceptron** during **training**. The delta rule determines the weight correction by multiplying the neuron's input with the **error** and the **learning rate**. [nn]

Demon
A **procedure** that is attached to a **slot** and executed if the slot value is changed or needed. A demon usually has an IF-THEN structure. Demon and **method** are often used as synonyms. [es]

DENDRAL
A rule-based expert system developed at Stanford University in the late 1960s for analysing chemicals, based on the mass spectral **data** provided by a mass spectrometer. DENDRAL

---

dm = data mining   ke = knowledge engineering

marked a major 'paradigm shift' in **AI**: a shift from general-purpose, knowledge-sparse methods to domain-specific, knowledge-intensive techniques. [es]

Dendrite
A branch of a biological **neuron** that transfers information from one part of a cell to another. Dendrites typically serve an input function for the cell, although many dendrites also have output functions. In an **ANN**, dendrites are modelled by inputs to a neuron. [nn]

Deterministic model
A mathematical model that postulates exact relationships between **objects** (no random variables are recognised). Given a set of input **data**, the deterministic model determines its output with complete certainty. [es]

Discrete data
The **data** that takes only a finite number of distinct values. Discrete data are usually (but not necessarily) counts. Examples of discrete data include the number of children in a family, the number of bedrooms in a house, the number of masts of a sailing vessel. [ke]

Disjunction
The logical operator **OR** that joins together two **antecedents** in a **production rule**. [es]

Domain
A relatively narrow problem area. For example, diagnosing blood diseases within the medical diagnostics field. **Expert systems** work in well-focused specialised domains. [es]

Domain expert
see **Expert**. [es]

EMYCIN
Empty **MYCIN**, an **expert system shell** developed at Stanford University in the late 1970s. It has all features of the MYCIN system except the **knowledge** of infectious blood diseases. EMYCIN is used to develop diagnostic **expert systems**. [es]

End-user
see **User**. [es]

Epoch
The presentation of the entire **training set** to an **ANN** during **training**. [nn]

Error
The difference between the actual and desired outputs in an **ANN** with **supervised learning**. [nn]

Euclidean distance
The shortest distance between two points in space. In Cartesian coordinates, the Euclidean distance between two points, (x_1, y_1) and (x_2, y_2), is determined by the Pythagorean theorem $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Evolution
A series of genetic changes by which a living organism acquires characteristics that distinguish it from other organisms. [ec]

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Evolution strategy

A numerical optimisation procedure similar to a focused Monte Carlo search. Unlike **genetic algorithms**, evolution strategies use only a **mutation** operator, and do not require a problem to be represented in a coded form. Evolution strategies are used for solving technical optimisation problems when no analytical objective function is available, and no conventional optimisation method exists. [ec]

Evolutionary computation

Computational models used for simulating **evolution** on a computer. The field of evolutionary computation includes **genetic algorithms**, **evolution strategies** and **genetic programming**. [ec]

Exhaustive search

A problem-solving technique in which every possible solution is examined until an acceptable one is found. [es]

Expert

A person who has deep **knowledge** in the form of **facts** and **rules** and strong practical experience in a particular **domain**. Also referred to as **Domain expert**. [es]

Expert system

A computer program capable of performing at the level of a human **expert** in a narrow **domain**. Expert systems have five basic components: the **knowledge base**, the **database**, the **inference engine**, the **explanation facilities** and the **user interface**. [es]

Expert system shell

A skeleton **expert system** with the **knowledge** removed. Also referred to as **Shell**. [es]

Explanation facility

A basic component of an **expert system** that enables the **user** to query the expert system about how it reached a particular conclusion and why it needs a specific **fact** to do so. [es]

Facet

A means of providing extended **knowledge** about an **attribute** of a **frame**. Facets are used to establish the attribute value, control the **user** queries, and tell the **inference engine** how to process the attribute. [es]

Fact

A statement that has the property of being either true or false. [es]

Feedback neural network

A **topology** of an **ANN** in which **neurons** have feedback loops from their outputs to their inputs. An example of a feedback network is the **Hopfield network**. Also referred to as **Recurrent network**. [nn]

Feedforward neural network

A **topology** of an **ANN** in which **neurons** in one **layer** are connected to the neurons in the next layer. The input signals are propagated in a forward direction on a layer-by-layer basis. An example of a feedforward network is a multilayer **perceptron**. [nn]

Field

A space allocated in a **database** for a particular **attribute**. (In a spreadsheet, fields are called

dm = data mining   ke = knowledge engineering

cells.) A tax form, for example, contains a number of fields: your name and address, tax file number, taxable income, etc. Every field in a database has a name, called the field name. [dm]

Firing a rule

The process of executing a **production rule**, or more precisely, executing the THEN part of a rule when its IF part is true. [es]

Fitness

The ability of a living organism to survive and reproduce in a specific environment. Also, a value associated with a **chromosome** that assigns a relative merit to that chromosome. [ec]

Fitness function

A mathematical function used for calculating the **fitness** of a **chromosome**. [ec]

Forward chaining

An **inference technique** that starts from the known **data** and works forward, matching the **facts** from the **database** with **production rules** from the **rule base** until no further rules can be fired. Also referred to as **Data-driven reasoning**. [es]

Frame

A **data** structure with typical **knowledge** about a particular **object**. Frames are used to represent knowledge in a **frame-based expert system**. [es]

Frame-based expert system

An **expert system** in which **frames** represent a major source of **knowledge**, and both **methods** and **demons** are used to add actions to the frames. In frame-based systems, **production rules** play an auxiliary role. [es]

Fuzzification

The first step in **fuzzy inference**; the process of mapping crisp (numerical) inputs into degrees to which these inputs belong to the respective **fuzzy sets**. [fl]

Fuzzy expert system

An **expert system** that uses **fuzzy logic** instead of **Boolean logic**. A fuzzy expert system is a collection of **fuzzy rules** and **membership functions** that are used to reason about **data**. Unlike conventional expert systems, which use **symbolic reasoning**, fuzzy expert systems are oriented towards numerical processing. [fl]

Fuzzy inference

The process of reasoning based on **fuzzy logic**. Fuzzy inference includes four steps: **fuzzification** of the input variables, **rule evaluation**, **aggregation** of the rule outputs and **defuzzification**. [fl]

Fuzzy logic

A system of logic developed for representing conditions that cannot be easily described by the binary terms 'true' and 'false'. The concept was introduced by Lotfi Zadeh in 1965. Unlike **Boolean logic**, fuzzy logic is multi-valued and handles the concept of partial truth (truth values between 'completely true' and 'completely false'). Also referred to as **Fuzzy set theory**. [fl]

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Fuzzy rule

A conditional statement in the form: IF *x* is *A* THEN *y* is *B*, where *x* and *y* are **linguistic variables**, and *A* and *B* are **linguistic values** determined by **fuzzy sets**. [fl]

Fuzzy set

A **set** with fuzzy boundaries, such as 'short', 'average' or 'tall' for men's height. To represent a fuzzy set in a computer, we express it as a function and then map the elements of the set to their **degree of membership**. [fl]

Fuzzy set theory

see **Fuzzy logic**. [fl]

Fuzzy singleton

A **fuzzy set** with a **membership function** equal to unity at a single point on the **universe of discourse** and zero everywhere else. Also referred to as **Singleton**. [fl]

Fuzzy variable

A quantity that can take on **linguistic values**. For example, the fuzzy variable 'temperature', might have values such as 'hot', 'medium' and 'cold'. [fl]

Gene

A basic unit of a **chromosome** that controls the development of a particular feature of a living organism. In Holland's chromosome, a gene is represented by either 0 or 1. [ec]

General Problem Solver (GPS)

An early **AI** system that attempted to simulate human methods of problem solving. The GPS was the first attempt to separate the problem-solving technique from the **data**. However, the program was based on the general-purpose **search** mechanism. This approach, now referred to as a weak method, applied weak information about the problem domain, and resulted in weak performance of the program in solving real-world problems. [es]

Generation

One iteration of a **genetic algorithm**. [ec]

Generalisation

The ability of an **ANN** to produce correct results from **data** on which it has not been trained. [nn]

Genetic algorithm

A type of **evolutionary computation** inspired by Darwin's theory of **evolution**. A genetic algorithm generates a population of possible solutions encoded as **chromosomes**, evaluates their **fitness**, and creates a new population by applying genetic operators – **crossover** and **mutation**. By repeating this process over many **generations**, the genetic algorithm breeds an optimal solution to the problem. [ec]

Genetic programming

An application of **genetic algorithms** to computer programs. Genetic programming is most easily implemented where the programming language permits a program to be manipulated as data and the newly created data to be executed as a program. This is one of the reasons why **LISP** is used as the main language for genetic programming. [ec]

dm = data mining   ke = knowledge engineering

Genetic operator

An operator in **genetic algorithms** or **genetic programming**, which acts upon the **chromosome** in order to produce a new individual. Genetic operators include **crossover** and **mutation**. [ec]

Global minimum

The lowest value of a function over the entire range of its input parameters. During **training**, the **weights** of an **ANN** are adjusted to find the global minimum of the **error** function. [nn]

Global optimisation

Finding the true optimum in the entire **search space**. [ec]

Goal

A hypothesis that an **expert system** attempts to prove. [es]

Goal-driven reasoning

see **Backward chaining**. [es]

Hard limit activation function

An **activation function** represented by the **step** and **sign functions**. Also referred to as **Hard limiter**. [nn]

Hard limiter

see **Hard limit activation function**. [nn]

Hebb's Law

The **learning** law introduced by Donald Hebb in the late 1940s; it states that if **neuron** *i* is near enough to excite neuron *j* and repeatedly participates in its activation, the synaptic **connection** between these two neurons is strengthened and neuron *j* becomes more sensitive to stimuli from neuron *i*. This law provides the basis for **unsupervised learning**. [nn]

Hebbian learning

Unsupervised learning that relates a change in the **weight** of the synaptic **connection** between a pair of **neurons** to a product of the incoming and outgoing signals. [nn]

Hedge

A qualifier of a **fuzzy set** used to modify its shape. Hedges include adverbs such as 'very', 'somewhat', 'quite', 'more or less' and 'slightly'. They perform mathematical operations of concentration by reducing the **degree of membership** of fuzzy elements (e.g. very tall men), dilation by increasing the degree of membership (e.g. more or less tall men) and intensification by increasing the degree of membership above 0.5 and decreasing those below 0.5 (e.g. indeed tall men). [fl]

Heuristic

A strategy that can be applied to complex problems; it usually – but not always – yields a correct solution. Heuristics, which are developed from years of experience, are often used to reduce complex problem solving to more simple operations based on judgment. Heuristics are often expressed as **rules of thumb**. [es]

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Heuristic search

A search technique that applies **heuristics** to guide the **reasoning**, and thus reduce the **search space** for a solution. [es]

Hidden layer

A **layer** of **neurons** between the **input** and **output layers**; called 'hidden' because neurons in this layer cannot be observed through the input/output behaviour of the **neural network**. There is no obvious way to know what the desired output of the hidden layer should be. [nn]

Hidden neuron

A **neuron** in the **hidden layer**. [nn]

Hopfield network

A single-layer **feedback neural network**. In the Hopfield network, the output of each **neuron** is fed back to the inputs of all other neurons (there is no self-feedback). The Hopfield network usually uses **McCulloch and Pitts neurons** with the **sign activation function**. The Hopfield network attempts to emulate characteristics of the **associative memory**. [nn]

Hybrid system

A system that combines at least two intelligent technologies. For example, combining a **neural network** with a fuzzy system results in a hybrid neuro-fuzzy system. [ke]

Hypothesis

A statement that is subject to proof. Also, a **goal** in **expert systems** that use **backward chaining**. [es]

Individual

A single member of a **population**. [ec]

Inductive reasoning

Reasoning from the specific to the general. [es]

Inference chain

The sequence of steps that indicates how an **expert system** applies **rules** from the **rule base** to reach a conclusion. [es]

Inference engine

A basic component of an **expert system** that carries out **reasoning** whereby the expert system reaches a solution. It matches the **rules** provided in the **rule base** with the **facts** contained in the **database**. Also referred to as **Interpreter.** [es]

Inference technique

The technique used by the **inference engine** to direct **search** and **reasoning** in an **expert system**. There are two principal techniques: **forward chaining** and **backward chaining**. [es]

Inheritance

The process by which all characteristics of a **class-frame** are assumed by the **instance-frame**. Inheritance is an essential feature of **frame-based systems**. A common use of inheritance is to impose default features on all instance-frames. [es]

---

dm = data mining   ke = knowledge engineering

Initialisation

The first step of the **training** algorithm that sets **weights** and **thresholds** to their initial values. [nn]

Input layer

The first **layer** of **neurons** in an **ANN**. The input layer accepts input signals from the outside world and redistributes them to neurons in the next layer. The input layer rarely includes computing neurons and does not process input patterns. [nn]

Input neuron

A **neuron** in the **input layer**. [nn]

Instance

A specific **object** from a **class**. For example, class 'computer' may have instances *IBM Aptiva S35* and *IBM Aptiva S9C*. In **frame-based expert systems**, all characteristics of a class are inherited by its instances. [es]

Instance

A member of the **schema**. For example, **chromosomes** $\boxed{1\,1\,1\,0}$ and $\boxed{1\,0\,1\,0}$ are the instances of the schema $\boxed{1\, *\, *\, 0}$. [ec]

Instance-frame

A **frame** that represents an **instance**. [es]

Instantiation

The process of assigning a specific value to a variable. For example, 'August' is an instantiation of the object 'month'. [es]

Intelligence

The ability to learn and understand, to solve problems and to make decisions. A machine is thought intelligent if it can achieve human-level performance in some cognitive task.

Interpreter

see **Inference engine**. [es]

Intersection

In classical **set theory**, an intersection between two **sets** contains elements shared by these sets. For example, the intersection of *tall men* and *fat men* contains all men who are tall *and* fat. In **fuzzy set theory**, an element may partly belong to both sets, and the intersection is the lowest **membership value** of the element in both sets. [fl]

is-a

An **arc** (also known as **'a-kind-of'**) that associates a superclass with its subclasses in a **frame-based expert system**. For example, if *car is-a vehicle*, then *car* represents a subclass of more general superclass *vehicle*. Each subclass inherits all features of the superclass. [es]

Knowledge

A theoretical or practical understanding of a subject. Knowledge is what helps us to make informed decisions.

Knowledge acquisition

The process of acquiring, studying and organising **knowledge**, so that it can be used in a **knowledge-based system**. [ke]

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Knowledge base

A basic component of an **expert system** that contains knowledge about a specific **domain**. [es]

Knowledge-based system

A system that uses stored **knowledge** for solving problems in a specific **domain**. A knowledge-based system is usually evaluated by comparing its performance with the performance of a human **expert**. [es]

Knowledge engineer

A person who designs, builds and tests a **knowledge-based system**. The knowledge engineer captures the **knowledge** from the **domain expert**, establishes reasoning methods and chooses the development software. [ke]

Knowledge engineering

The process of building a **knowledge-based system**. There are six main steps: assess the problem; acquire data and knowledge; develop a prototype system; develop a complete system; evaluate and revise the system; integrate and maintain the system. [ke]

Knowledge representation

The process of structuring **knowledge** to be stored in a **knowledge-based system**. In **AI**, **production rules** are the most common type of knowledge representation. [ke]

Kohonen self-organising feature maps

A special class of **ANNs** with **competitive learning** introduced by Teuvo Kohonen in the late 1980s. The Kohonen map consists of a single **layer** of computation **neurons** with two types of **connections**: forward connections from the neurons in the input layer to the neurons in the output layer, and lateral connections between neurons in the output layer. The lateral connections are used to create a competition between neurons. A neuron learns by shifting its **weights** from inactive connections to active ones. Only the winning neuron and its neighbourhood are allowed to learn. [nn]

Layer

A group of **neurons** that have a specific function and are processed as a whole. For example, a **multilayer perceptron** has at least three layers: an **input layer**, an **output layer** and one or more **hidden layers**. [nn]

Leaf

A bottom-most **node** of a **decision tree**; a node without **children**. Also referred to as a **Terminal node**. [dm]

Learning

The process by which **weights** in an **ANN** are adjusted to achieve some desired behaviour of the network. Also referred to as **Training**. [nn]

Learning rate

A positive number less than unity that controls the amount of changes to the **weights** in the **ANN** from one iteration to the next. The learning rate directly affects the speed of network **training**. [nn]

Learning rule

A procedure for modifying **weights** during **training** in an **ANN**. [nn]

---

dm = data mining   ke = knowledge engineering

Linear activation function

An **activation function** that produces an output equal to the net input of a **neuron**. Neurons with the linear activation function are often used for linear approximation. [nn]

Linguistic variable

A variable that can have values that are language elements, such as words and phrases. In **fuzzy logic**, terms linguistic variable and **fuzzy variable** are synonyms. [fl]

Linguistic value

A language element that can be assumed by a **fuzzy variable**. For example, the fuzzy variable 'income' might assume such linguistic values as 'very low', 'low', 'medium', 'high' and 'very high'. Linguistic values are defined by **membership functions**. [fl]

LISP (LISt Processor)

One of the oldest high-level programming languages. LISP, which was developed by John McCarthy in the late 1950s, has become a standard language for **artificial intelligence**.

Local minimum

The minimum value of a function over a limited range of its input parameters. If a local minimum is encountered during **training**, the desired behaviour of an **ANN** may never be achieved. The usual method of getting out of a local minimum is to randomise the **weights** and continue training. [nn]

Machine learning

An adaptive mechanism that enable computers to learn from experience, learn by example and learn by analogy. Learning capabilities improve the performance of an intelligent system over time. Machine learning is the basis of adaptive systems. The most popular approaches to machine learning are **artificial neural networks** and **genetic algorithms**.

Massaging data

The process of modifying the **data** before it is applied to the **input layer** of an **ANN**. [nn]

McCulloch and Pitts neuron model

A **neuron** model proposed by Warren McCulloch and Walter Pitts in 1943, which is still the basis for most **artificial neural networks**. The model consists of a linear combiner followed by a **hard limiter**. The net input is applied to the hard limiter, which produces an output equal to $+1$ if its input is positive and -1 if it is negative. [nn]

Membership function

A mathematical function that defines a **fuzzy set** on the **universe of discourse**. Typical membership functions used in **fuzzy expert systems** are triangles and trapezoids. [fl]

Membership value

see **Degree of membership**. [fl]

Metaknowledge

Knowledge about knowledge; knowledge about the use and control of **domain** knowledge in **expert systems**. [es]

Metarule

A **rule** that represents **metaknowledge**. A metarule determines a strategy for the use of task-specific rules in the **expert system**. [es]

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Method

A **procedure** associated with an **attribute** of a **frame**. A method can determine the attribute's value or execute a series of actions when the attribute's value changes. Most **frame-based expert systems** use two types of methods: **WHEN CHANGED** and **WHEN NEEDED**. Method and **demon** are often used as synonyms. [es]

Momentum constant

A positive constant less than unity included in the **delta rule**. The use of momentum accelerates **learning** in a **multilayer perceptron** and helps to prevent it from getting caught in a **local minimum**. [nn]

Multilayer perceptron

The most common **topology** of an **ANN** in which **perceptrons** are connected together to form **layers**. A multilayer perceptron has the **input layer**, at least one **hidden layer** and the **output layer**. The most popular method of training a multilayer perceptron is **back-propagation**. [nn]

Multiple inheritance

The ability of an **object** or a **frame** to inherit information from multiple superclasses. [es]

Mutation

A **genetic operator** that randomly changes the **gene** value in a **chromosome**. [ec]

Mutation probability

A number between zero and one that indicates the probability of **mutation** occurring in a single **gene**. [ec]

MYCIN

A classic **rule-based expert system** developed in the 1970s for the diagnosis of infectious blood diseases. The system used **certainty factors** for managing uncertainties associated with **knowledge** in medical diagnosis. [es]

Natural selection

The process by which the most fit individuals have a better chance to mate and reproduce, and thereby to pass their genetic material on to the next generation. [ec]

Neural computing

A computational approach to modelling the human brain that relies on connecting a large number of simple processors to produce complex behaviour. Neural computing can be implemented on specialised hardware or with software, called **artificial neural networks**, that simulates the structure and functions of the human brain on a conventional computer. [nn]

Neural network

A system of processing elements, called **neurons**, connected together to form a network. The fundamental and essential characteristic of a biological neural network is the ability to learn. **Artificial neural networks** also have this ability; they are not programmed, but learn from examples through repeated adjustments of their **weights**. [nn]

Neuron

A cell that is capable of processing information. A typical neuron has many inputs (**dendrites**) and one output (**axon**). The human brain contains roughly 10^{12} neurons.

---

dm = data mining   ke = knowledge engineering

Also, a basic processing element of an **ANN** that computes the weighted sum of the input signals and passes the result through its **activation function** to generate an output. [nn]

Node

A decision point of a **decision tree**. [dm]

Noise

A random external disturbance that affects a transmitted signal. Noisy **data** contain errors associated with the way the data was collected, measured and interpreted. [dm]

NOT

A logical operator used for representing the negation of a statement. [es]

Object

A concept, abstraction or thing that can be individually selected and manipulated, and that has some meaning for the problem at hand. All objects have identity and are clearly distinguishable. *Michael Black*, *Audi 5000 Turbo*, *IBM Aptiva S35* are examples of objects. In **object-oriented programming**, an object is a self-contained entity that consists of both **data** and **procedures** to manipulate the data. [es]

Object-oriented programming

A programming method that uses **objects** as a basis for analysis, design and implementation. [es]

Offspring

An individual that was produced through **reproduction**. Also referred to as a **child**. [ec]

Operational database

A **database** used for the daily operation of an organisation. **Data** in operational databases is regularly updated. [dm]

OPS

A high-level programming language derived from **LISP** for developing **rule-based expert systems**. [es]

Optimisation

An iterative process of improving the solution to a problem with respect to a specified objective function. [ec]

OR

A logical operator; when used in a **production rule**, it implies that if any of the **antecedents** joined with OR is true, then the rule **consequent** must also be true. [es]

Overfitting

A state in which an **ANN** has memorised all the training examples, but cannot generalise. Overfitting may occur if the number of **hidden neurons** is too big. The practical approach to preventing overfitting is to choose the smallest number of hidden neurons that yields good **generalisation**. Also referred to as **Over-training**. [nn]

Over-training

see **Overfitting**. [nn]

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Output layer
The last **layer** of **neurons** in an **ANN**. The output layer produces the output pattern of the entire network. [nn]

Output neuron
A **neuron** in the **output layer**. [nn]

Parallel processing
A computational technique that carries out multiple tasks simultaneously. The human brain is an example of a parallel information-processing system: it stores and processes information simultaneously throughout the whole biological **neural network**, rather than at specific locations. [nn]

Parent
An individual that produces one or more other individuals, known as **offspring** or **child**. [ec]

Parent
In a **decision tree**, a parent node is a **node** that splits its data between nodes at the next hierarchical level of the tree. The parent node contains a complete data set, while **child** nodes hold subsets of that set. [dm]

Pattern recognition
Identification of visual or audio patterns by computers. Pattern recognition involves converting patterns into digital signals and comparing them with patterns already stored in the memory. **Artificial neural networks** are successfully applied to pattern recognition, particularly in such areas as voice and character recognition, radar target identification and robotics. [nn]

Perceptron
The simplest form of a **neural network**, suggested by Frank Rosenblatt. The operation of the perceptron is based on the **McCulloch and Pitts neuron model**. It consists of a single **neuron** with adjustable **synaptic weights** and a **hard limiter**. The perceptron learns a task by making small adjustments in the weights to reduce the difference between the actual and desired outputs. The initial weights are randomly assigned and then updated to obtain an output consistent with the training examples. [nn]

Performance
A statistical evaluation of **fitness**. [ec]

Performance graph
A graph that shows the average **performance** of the entire **population** and the performance of the best individual in the population over the chosen number of **generations**. [ec]

Pixel
Picture Element; a single point in a graphical image. Computer monitors display pictures by dividing the screen into thousands (or millions) of pixels arranged into rows and columns. The pixels are so close together that they appear as one image. [ke]

Population
A group of individuals that breed together. [ec]

---

dm = data mining   ke = knowledge engineering

Premise
see **Antecedent**. [es]

Probability
A quantitative description of the likely occurrence of a particular event. Probability is expressed mathematically as a number with a range between zero (an absolute impossibility) to unity (an absolute certainty). [es]

Procedure
A self-contained arbitrary piece of computer code. [es]

Production
A term often used by cognitive psychologists to describe a **rule**. [es]

Production rule
A statement expressed in the IF (antecedent) THEN (consequent) form. If the **antecedent** is true, then the **consequent** is also true. Also referred to as **Rule**. [es]

PROLOG
A high-level programming language developed at the University of Marseilles in the 1970s as a practical tool for programming in logic; a popular language for **artificial intelligence**.

PROSPECTOR
An **expert system** for mineral exploration developed by the Stanford Research Institute in the late 1970s. To represent **knowledge**, PROSPECTOR used a combined structure that incorporated **production rules** and a **semantic network**. [es]

Query tool
Software that allows a **user** to create and direct specific questions to a **database**. A query tool provides the means for extracting the desired information from a database. [dm]

Reasoning
The process of drawing conclusions or inferences from observations, **facts** or assumptions. [es]

Record
see **Data record**. [dm]

Recurrent network
see **Feedback network**. [nn]

Reproduction
The process of creating **offspring** from **parents**. [ec]

Root
see **Root node**. [dm]

Root node
The top-most **node** of a **decision tree**. The tree always starts from the root node and grows down by splitting the **data** at each level into new nodes. The root node contains the entire data set (all **data records**), and **child** nodes hold subsets of that set. Also referred to as **Root**. [dm]

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

Roulette wheel selection

A method of selecting a particular individual in the **population** to be a **parent** with a probability equal to its **fitness** divided by the total fitness of the population. [ec]

Rule

see **Production rule**. [es]

Rule base

The **knowledge base** that contains a set of **production rules**. [es]

Rule-based expert system

An **expert system** whose **knowledge base** contains a set of **production rules**. [es]

Rule evaluation

The second step in **fuzzy inference**; the process of applying the fuzzy inputs to the **antecedents** of **fuzzy rules**, and determining the **truth value** for the antecedent of each rule. If a given rule has multiple antecedents, the fuzzy operation of **intersection** or **union** is carried out to obtain a single number that represents the result of evaluating the antecedent. [fl]

Rule of thumb

A **rule** that expresses a **heuristic**. [es]

Scaling

A method of correlating the **consequent** of a **fuzzy rule** with the **truth value** of the rule **antecedent**. It is based on adjusting the original **membership function** of the rule consequent by multiplying it by the truth value of the rule antecedent. Scaling helps to preserve the original shape of the **fuzzy set**. [fl]

Search

The process of examining a set of possible solutions to a problem in order to find an acceptable solution. [es]

Search space

The set of all possible solutions to a given problem. [es]

Self-organised learning

see **Unsupervised learning**. [nn]

Semantic network

A method of **knowledge representation** by a graph made up of labelled nodes and **arcs**, where the nodes represent **objects** and the arcs describe relationships between these objects. [es]

Set

A collection of elements (also called members).

Set theory

The study of **sets** or **classes** of **objects**. The set is the basic unit in mathematics. Classical set theory does not acknowledge the **fuzzy set**, whose elements can belong to a number of sets to some degree. Classical set theory is bivalent: the element either does or does not belong to a particular set. That is, classical set theory gives each member of the set the value of 1, and all members that are not within the set a value of 0.

dm = data mining   ke = knowledge engineering

Schema

A bit string of ones, zeros and asterisks, where each asterisk can assume either value 1 or 0. For example, the schema $\boxed{1 * * 0}$ stands for a set of four 4-bit strings with each string beginning with 1 and ending with 0. [ec]

Schema theorem

A theorem that relates the expected number of **instances** of a given **schema** in the consequent **generation** with the **fitness** of this schema and the average fitness of **chromosomes** in the current generation. The theorem states that a schema with above-average fitness tends to occur more frequently in the next generation. [ec]

Selection

The process of choosing **parents** for **reproduction** based on their **fitness**. [ec]

Sensitivity analysis

A technique of determining how sensitive the output of a model is to a particular input. Sensitivity analysis is used for understanding relationships in opaque models, and can be applied to **neural networks**. Sensitivity analysis is performed by measuring the network output when each input is set (one at a time) to its minimum and then its maximum values. [ke]

Shell

see **Expert system shell**. [es]

Sigmoid activation function

An **activation function** that transforms the input, which can have any value between plus and minus infinity, into a reasonable value in the range between 0 and 1. **Neurons** with this function are used in a **multilayer perceptron**. [nn]

Sign activation function

A **hard limit activation function** that produces an output equal to $+1$ if its input is positive and -1 if it is negative. [nn]

Singleton

see **Fuzzy singleton**. [fl]

Slot

A component of a **frame** in a **frame-based system** that describes a particular **attribute** of the frame. For example, the frame 'computer' might have a slot for the attribute 'model'. [es]

Soma

The body of a biological **neuron**. [nn]

Step activation function

A **hard limit activation function** that produces an output equal to $+1$ if its input is positive and 0 if it is negative. [nn]

Supervised learning

A type of **learning** that requires an external teacher, who presents a sequence of training examples to the **ANN**. Each example contains the input pattern and the desired output pattern to be generated by the network. The network determines its actual output and compares it with the desired output from the training example. If the output from

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

the network differs from the desired output specified in the training example, the network **weights** are modified. The most popular method of supervised learning is **back-propagation**. [nn]

Survival of the fittest
The law according to which only individuals with the highest **fitness** can survive to pass on their genes to the next generation. [ec]

Symbol
A character or a string of characters that represents some **object**. [es]

Symbolic reasoning
Reasoning with **symbols**. [es]

Synapse
A chemically mediated **connection** between two **neurons** in a biological **neural network**, so that the state of the one cell affects the state of the other. Synapses typically occur between an **axon** and a **dendrite**, though there are many other arrangements. See also **Connection**. [nn]

Synaptic weight
see **Weight**. [nn]

Terminal node
see **Leaf**. [dm]

Test set
A data set used for testing the ability of an **ANN** to generalise. The test data set is strictly independent of the **training set**, and contains examples that the network has not previously seen. Once **training** is complete, the network is validated with the test set. [nn]

Threshold
A specific value that must be exceeded before the output of a **neuron** is generated. For example, in the **McCulloch and Pitts neuron model**, if the net input is less than the threshold, the neuron output is -1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value $+1$. Also referred to as **Threshold value**. [nn]

Threshold value
see **Threshold**. [nn]

Topology
A structure of a **neural network** that refers to the number of **layers** in the neural network, the number of **neurons** in each layer, and **connections** between neurons. Also referred to as **Architecture**. [nn]

Toy problem
An artificial problem, such as a game. Also, an unrealistic adaptation of a complex problem. [es]

Training
see **Learning**. [nn]

dm = data mining   ke = knowledge engineering

Training set

A data set used for **training** an **ANN**. [nn]

Transfer function

see **Activation function**. [nn]

Truth value

In general, the terms **truth value** and **membership value** are used as synonyms. The truth value reflects the truth of a fuzzy statement. For example, the fuzzy proposition x is A (0.7) suggests that element x is a member of **fuzzy set** A to the degree 0.7. This number represents the truth of the proposition. [fl]

Turing test

A test designed to determine whether a machine can pass a behaviour test for **intelligence**. Turing defined the intelligent behaviour of a computer as the ability to achieve human-level performance in cognitive tasks. During the test a human interrogates some*one* or some*thing* by questioning it via a neutral medium such as a remote terminal. The computer passes the test if the interrogator cannot distinguish the machine from a human.

Union

In classical **set theory**, the union of two **sets** consists of every element that falls into either set. For example, the union of *tall men* and *fat men* contains all men who are either tall *or* fat. In **fuzzy set theory**, the union is the reverse of the **intersection**, that is, the union is the largest **membership value** of the element in either set. [fl]

Universe of discourse

The range of all possible values that are applicable to a given variable. [fl]

Unsupervised learning

A type of **learning** that does not require an external teacher. During **learning** an **ANN** receives a number of different input patterns, discovers significant features in these patterns and learns how to classify input data into appropriate categories. Also referred to as **Self-organised learning**. [nn]

User

A person who uses a **knowledge-based system** when it is developed. For example, the user might be an analytical chemist determining the molecular structures, a junior doctor diagnosing an infectious blood disease, an exploration geologist trying to discover a new mineral deposit, or a power system operator seeking an advice in an emergency. Also referred to as **End-user**. [es]

User interface

A means of communication between a **user** and a machine. [es]

Visualisation

see **Data visualisation**. [dm]

Weight

The value associated with a **connection** between two **neurons** in an **ANN**. This value

---

es = expert systems   fl = fuzzy logic   nn = neural networks   ec = evolutionary computation

determines the strength of the connection and indicates how much of the output of one neuron is fed to the input of another. Also referred to as **Synaptic weight**. [nn]

WHEN CHANGED method

A **procedure** attached to a **slot** of a **frame** in a **frame-based expert system**. The WHEN CHANGED method is executed when new information is placed in the slot. [es]

WHEN NEEDED method

A **procedure** attached to a **slot** of a **frame** in a **frame-based expert system**. The WHEN NEEDED method is executed when information is needed for the problem solving, but the slot value is unspecified. [es]

---

dm = data mining   ke = knowledge engineering

Appendix:
AI tools and vendors

Expert system shells

ACQUIRE

A knowledge acquisition and expert system development tool. Knowledge is represented by production rules and pattern-based action tables. ACQUIRE does not require special training in building expert systems. A domain expert can create a knowledge base and develop applications without help from the knowledge engineer.

Acquired Intelligence Inc.
Suite 205, 1095 McKenzie Avenue
Victoria, BC, Canada V8P 2L5
Phone: +1 (250) 479-8646
Fax: +1 (250) 479-0764

http://www.aiinc.ca/acquire/acquire.shtml

Blaze Advisor

A sophisticated tool for developing rule-based object-oriented expert systems. Advisor has two components: *Advisor Builder* (a development tool with visual editors, powerful debugging facilities and wizards, which integrate rule-based applications with databases, Java objects and COBRA objects) and *Advisor Engine* (a high-performance inference engine). Advisor includes mechanisms for servicing simultaneous users, scheduling deployments, performing dynamic load balancing, and reducing memory requirements.

Fair Isaac Corporation
200 Smith Ranch Road
San Rafael, CA 94903, USA
Phone: +1 (415) 472-2211
Fax: +1 (415) 492-5691

http://www.fairisaac.com/Fairisaac/Solutions/Product+Index/Blaze+Advisor/

Exsys CORVID

An expert system development tool for converting complex decision-making processes into a form that can be incorporated into a Web page. CORVID, which is based on the Visual Basic model, provides an object-oriented structure. It also uses logic blocks – supersets of rules and trees – which can be run by forward or backward chaining. CORVID applications are delivered via a small Java applet that allows robust interface design options.

EXSYS, Inc.
2155 Louisiana Blvd NE, Suite 3100
Albuquerque, NM 87110, USA
Phone: +1 (505) 888-9494

http://www.exsys.com/

Flex

A frame-based expert system toolkit. Supports frame-based reasoning with inheritance, rule-based programming and data-driven procedures. Flex has its own English-like knowledge specification language (KSL). The main structures in Flex are frames and instances with slots for organising objects, default and current values for storing data, demons and constraints for adding functionality to slot values, rules and relations for expressing knowledge and expertise, functions and actions for defining imperative processes, and questions and answers for end-user interaction. The KSL supports mathematical, Boolean and conditional expressions.

Logic Programming Associates Ltd
Studio 4, RVPB, Trinity Road
London SW18 3SX, England
Phone: +44 (0) 20-8871-2016
Fax: +44 (0) 20-8874-0449
e-mail: support@lpa.co.uk

http://www.lpa.co.uk/

G2

An interactive object-oriented, graphical environment for the development and on-line deployment of intelligent systems. Objects are organised in hierarchical classes with multiple inheritance. Developers can model an application by representing and connecting objects graphically. Expert knowledge is expressed by rules. G2 uses forward chaining to respond automatically whenever new data arrive, and backward chaining to invoke rules or procedures. G2 works efficiently in real time.

Gensym Corporation
52 Second Avenue
Burlington, MA 01803, USA
Phone: +1 (781) 265-7100
Fax: +1 (781) 265-7101
e-mail: info@gensym.com

http://www.gensym.com/manufacturing/g2_overview.shtml

GURU

A rule-based expert system development environment that offers a wide variety of information processing tools. GURU uses fuzzy logic and certainty factors to handle uncertainties in human knowledge. At the core of GURU is KGL, a knowledge and object-based fourth-generation programming language, including a self-contained relational database.

Micro Data Base Systems, Inc.
Research Park, 1305 Cumberland Ave
PO Box 2438, West Lafayette, IN 47996-2438, USA

Phone: +1 (765) 463-7200
Fax: +1 (765) 463-1234

http://www.mdbs.com/html/guru.html

Intellix

A comprehensive tool developed by combining neural network and expert system technologies. The tool provides a user-friendly environment where no programming skills are required. Domain knowledge is represented by production rules and examples. The system uses a combined technique of pattern matching (neural networks) and rule interpretation, and is capable of learning in real time.

Intellix Denmark
Nikolaj Plads 32, 2
DK-1067 Copenhagen K, Denmark
Phone: +45 3314-8100
Fax: +45 3314-8130
e-mail: info@intellix.com

http://www.intellix.com/products/designer/designer.html

JESS

The Java Expert System Shell (JESS) is available as a free download (including its complete Java source code) from Sandia National Laboratories. JESS was originally inspired by CLIPS (C Language Integrated Production System), but has grown into a complete tool of its own. The JESS language is still compatible with CLIPS – JESS scripts are valid CLIPS scripts and vice versa. JESS adds many features to CLIPS, including backward chaining and the ability to manipulate and directly reason about Java objects. Despite being implemented in Java, JESS runs faster than CLIPS.

Sandia National Laboratories, California
PO Box 969
Livermore, CA 94551, USA
e-mail: casmith@sandia.gov

http://herzberg.ca.sandia.gov/jess

Level5 Object

A tool for developing frame-based expert systems. Objects in a knowledge base are created via class declarations. Rules and demons describe rules-of-thumb and cause-and-effect relationships for making decisions and triggering certain events or actions during a session. Databases are managed by the *Object-Oriented Database Management System*, which allows the system to obtain attribute values of a class from an external database.

Rule Machines Corporation
51745 396th Ave
Frazee, MN 56544, USA
Phone: +1 (218) 334-3960
Fax: +1 (218) 334-3957
e-mail: info@RuleMachines.com

http://www.rulemachines.com/

M.4

A powerful tool for developing rule-based expert systems. Domain knowledge is represented by production rules. M.4 employs both backward and forward chaining inference techniques. It uses certainty factors for managing inexact knowledge, and supports object-oriented programming within the system.

Teknowledge
1810 Embarcadero Road
Palo Alto, CA 94303, USA
Phone: +1 (650) 424-0500
Fax: +1 (650) 493-2645
e-mail: info@teknowledge.com

http://www.teknowledge.com/m4/

Visual Rule Studio

Visual Rule Studio is based on the *Production Rule Language* (PRL) and inference engines of Level5 Object. The language and inference engines of Visual Rule Studio are compatible with Level5 Object. Visual Rule Studio is built specifically for Visual Basic developers – Visual Rule Studio installs into Visual Basic as an ActiveX Designer. It allows developers to create intelligent objects as reusable components.

Rule Machines Corporation
51745 396th Ave
Frazee, MN 56544, USA
Phone: +1 (218) 334-3960
Fax: +1 (218) 334-3957
e-mail: info@RuleMachines.com

http://www.rulemachines.com/VRS/Index.htm

XMaster

The system consists of two basic packages: *XMaster Developer* and *XMaster User*. With *XMaster Developer* the user creates a knowledge base simply by building up a list of possible *hypotheses* and a list of *items of evidence*. The items of evidence are then associated with the relevant hypotheses. XMaster also enables the user to incorporate uncertain or approximate relationships into the knowledge base. It uses Bayesian reasoning for managing uncertainties.

Chris Naylor Research Limited
14 Castle Gardens
Scarborough, North Yorkshire
YO11 1QU, England
Phone: +44 (1) 723-354-590
e-mail: ChrisNaylor@ChrisNaylor.co.uk

http://www.chrisnaylor.co.uk/

XpertRule

A tool for developing rule-based expert systems. Domain knowledge is represented by decision trees, examples, truth tables and exception trees. Decision trees are the main knowledge representation method. Examples relate outcomes to attributes. A truth table is an extension to examples – it represents a set of examples covering every possible combination of cases. From examples, truth tables and exception trees, XpertRule automatically

generates a decision tree. XpertRule also uses fuzzy reasoning, which can be integrated with crisp reasoning and with GA optimisation.

Attar Software UK
Newlands Road
Leigh WN7 4HN, England
Phone: +44 (0) 870-60-60-870
Fax: +44 (0) 870-60-40-156
e-mail: info@attar.co.uk

Intellicrafters (Attar Software USA)
Renaissance International Corporation
Newburyport, MA 01950, USA
Phone: +1 (978) 465-5111
Fax: +1 (978) 465-0666
e-mail: info@IntelliCrafters.com

http://www.attar.com/

Fuzzy logic tools

CubiCalc
A software tool for creating and using fuzzy rules. With CubiCalc, the user can write English-like IF-THEN rules and use a graphical editor to define fuzzy sets. The user can then apply the rules to data or use them in a simulated dynamic scenario. CubiCalc is particularly useful for rapid prototyping. No programming is needed to set up plots, numeric displays, input and output data files, and interactive data-entry windows.

HyperLogic Corporation
PO Box 300010
Escondido, CA 92030-0010, USA
Phone: +1 (760) 746-2765
Fax: +1 (760) 746-4089

http://www.hyperlogic.com/cbc.html

FIDE
The Fuzzy Inference Development Environment (FIDE) is a complete environment for developing a fuzzy system. It supports all phases of the development process, from the concept to the implementation. FIDE serves as the developer's guide in creating a fuzzy controller, including its implementation as a software or hardware solution. Hardware solutions are realised in the Motorola microcontroller units; the code is generated automatically. FIDE also supports C code by creating ANSI C code for a fuzzy inference unit.

Aptronix, Inc.
PO Box 70188
Sunnyvale, CA 94086-0188, USA
Phone: +1 (408) 261-1898
Fax: +1 (408) 490-2729
e-mail: support@aptronix.com

http://www.aptronix.com/fide/

FlexTool
FlexTool offers the Genetic Algorithm, Neural Network and Fuzzy System MATLAB Toolbox for building intelligent systems. The readable full-source code is included with the toolbox; it can be easily customised as well as tailored to the user's needs. FlexTool (Fuzzy System) facilitates the development of fuzzy expert systems, fuzzy predictors and fuzzy controllers. It provides a graphical user interface for tuning membership functions.

CynapSys, LLC
160 Paradise Lake Road
Birmingham, AL 35244, USA
Phone/Fax: +1 (530) 325-9930
e-mail: info@cynapsys.com

http://www.flextool.com/

FLINT

The Fuzzy Logic INferencing Toolkit (FLINT) is a versatile fuzzy logic inference system that makes fuzzy rules available within a sophisticated programming environment. FLINT supports the concepts of fuzzy variables, fuzzy qualifiers and fuzzy modifiers (linguistic hedges). Fuzzy rules are expressed in a simple, uncluttered syntax. Furthermore, they can be grouped into matrices, commonly known as fuzzy associative memory (FAM). FLINT provides a comprehensive set of facilities for programmers to construct fuzzy expert systems and decision-support applications on all LPA-supported hardware and software platforms.

Logic Programming Associates Ltd
Studio 4, RVPB, Trinity Road
London SW18 3SX, England
Phone: +44 (0) 208-871-2016
Fax: +44 (0) 208-874-0449
e-mail: support@lpa.co.uk

http://www.lpa.co.uk/

FuzzyCLIPS

FuzzyCLIPS is an extension of the CLIPS (C Language Integrated Production System) from NASA, which has been widely distributed for a number of years. It enhances CLIPS by providing a fuzzy reasoning capability such that the user can represent and manipulate fuzzy facts and rules. FuzzyCLIPS can deal with exact, fuzzy and combined reasoning, allowing fuzzy and normal terms to be freely mixed in the rules and facts of an expert system. The system uses two basic inexact concepts: fuzziness and uncertainty. FuzzyCLIPS is available as a free download.

Integrated Reasoning Group
NRC Institute for Information Technology
1200 Montreal Road, Building M-50
Ottawa, ON Canada, K1A 0R6
Phone: +1 (613) 993-8557
Fax: +1 (613) 952-0215
e-mail: Bob.Orchard@nrc-cnrc.gc.ca

http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyClips/fuzzyCLIPSIndex.html

Fuzzy Control Manager

The Fuzzy Control Manager (FCM) provides a graphical user interface (GUI) that allows the user to display any relevant data while developing, debugging and optimising a fuzzy system. Offers the point-and-click rule editor and graphical editor of membership functions. The FCM enables the user to generate a source code in C assembler or binary codes.

TransferTech GmbH
Cyriaksring 9A
D-38118 Braunschweig, Germany
Phone: +49 (531) 890-255
Fax: +49 (531) 890-355
e-mail: info@transfertech.de

http://www.transfertech.de/wwwe/fcm/fcme_gen.htm

FuzzyJ Toolkit

The FuzzyJ Toolkit is a set of Java classes that provide the capability for handling fuzzy reasoning. It is useful for exploring fuzzy logic in a Java setting. The work is based on earlier experience building the FuzzyCLIPS extension to the CLIPS Expert System Shell. The toolkit can be used stand-alone to create fuzzy rules and do reasoning. It can also be used with JESS, the Java Expert System Shell from Sandia National Laboratories. FuzzyJ is available as a free download.

Integrated Reasoning Group
NRC Institute for Information Technology
1200 Montreal Road, Building M-50
Ottawa, ON Canada, K1A 0R6
Phone: +1 (613) 993-8557
Fax: +1 (613) 952-0215
e-mail: Bob.Orchard@nrc-cnrc.gc.ca

http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit.html

Fuzzy Judgment Maker

A tool for developing fuzzy decision-support systems. It breaks down the decision scenario into small parts that the user can focus on and input easily. It then uses theoretically optimal methods of combining the scenario pieces into a global interrelated solution. The Judgment Maker provides graphical tools for negotiating decisions and making the consensus from two decisions.

Fuzzy Systems Engineering
12223 Wilsey Way
Poway, CA 92064, USA
Phone: +1 (858) 748-7384
e-mail: mmcneill@fuzzysys.com

http://www.fuzzysys.com/

Fuzzy Query

Fuzzy Query is an application based on Win32. It allows the user to query a database using the power and semantic flexibility of the *Structured Query Language* (SQL) – the most popular method for retrieving information from databases. Fuzzy Query provides information beyond the strict restrictions of Boolean logic. The user not only sees the candidates that best meet some specified criteria, but can also observe the candidates that just barely miss the cut. Each record returned by a Fuzzy Query shows data ranked by the degree to which it meets the specified criteria.

Fuzzy Systems Solutions
Sonalysts Inc.

215 Parkway North
Waterford, CT 06385, USA
Phone: +1 (860) 526-8091
Fax: +1 (860) 447-8883
e-mail: FuzzyQuery@Sonalysts.com

http://fuzzy.sonalysts.com/

*Fuzzy*TECH

*Fuzzy*TECH is the world's leading family of software development tools for fuzzy logic and neural–fuzzy solutions. It provides two basic products: *Editions* for technical applications and *Business* for applications in finance and business. The tree view enables the structured access to all components of a fuzzy logic system under design in the same way that Windows Explorer lets users browse the structure of their PCs. The Editor and Analyser windows allow each component of a fuzzy system to be designed graphically.

Inform Software Corporation	INFORM GmbH
222 South Riverside Plaza	Pascalstrasse 23
Suite 1410 Chicago, IL 60606, USA	D-52076 Aachen, Germany
Phone: +1 (312) 575-0578	Phone: +49 2408-945-680
Fax: +1 (312) 575-0581	Fax: +49 2408-945-685
e-mail: office@informusa.com	e-mail: hotline@inform-ac.com

http://www.fuzzytech.com/

Mathematica Fuzzy Logic Package

The package represents built-in functions that facilitate in defining inputs and outputs, creating fuzzy sets, manipulating and combining fuzzy sets and relations, applying fuzzy inference functions, and incorporating defuzzification routines. Experienced fuzzy logic designers find it easy to use the package to research, model, test and visualise highly complex systems. Fuzzy Logic requires Mathematica 4 or 5 and is available for Windows, Mac OS X, Linux and most Unix platforms.

Wolfram Research, Inc.
100 Trade Center Drive
Champaign, IL 61820-7237, USA
Phone: +1 (217) 398-0700
Fax: +1 (217) 398-1108

http://www.wolfram.com/products/applications/fuzzylogic/

MATLAB Fuzzy Logic Toolbox

Features a simple point-and-click interface that guides the user through the steps of fuzzy design, from set-up to diagnosis. It provides built-in support for the latest fuzzy logic methods, such as fuzzy clustering and adaptive neuro-fuzzy learning. The toolbox's interactive graphics let the user visualise and fine-tune system behaviour.

The MathWorks
3 Apple Hill Drive
Natick, MA 01760-2098, USA
Phone: +1 (508) 647-7000
Fax: +1 (508) 647-7001

http://www.mathworks.com/products/fuzzylogic/

rFLASH

rFLASH (Rigel's Fuzzy Logic Applications Software Helper) is a code generator that creates a set of subroutines and tables in the MCS-51 assembly language to implement Fuzzy Logic Control (FLC) applications. The code generated runs on the 8051 family of microcontrollers. rFLASH software includes a code generator and a simulator. As a code generator, rFLASH creates the FLC code directly from a high-level Control Task Description File (CTDF). As a simulator, rFLASH generates the outputs from given inputs on the PC. The simulator can test several inputs and fine-tune the terms or rules accordingly.

> Rigel Corporation
> PO Box 90040
> Gainesville, FL 32607, USA
> Phone: +1 (352) 384-3766
> e-mail: techsupport@rigelcorp.com
>
> http://www.rigelcorp.com/flash.htm

TILShell

The windows-based software development tool for designing, debugging and testing fuzzy expert systems, including embedded control systems. It offers real-time on-line debugging and tuning fuzzy rules, membership functions and rule weights; 3-D visualisation tools; fully integrated graphical simulation of fuzzy systems and conventional methods; and ANSI and Keil C code generation from the Fuzzy-C compiler.

> Ortech Engineering Inc.
> 16250 Highway 3, Suite B6
> Webster, TX 77598, USA
> Phone: +1 (281) 480-8904
> Fax: +1 (281) 480-8906
> e-mail: togai@ortech-engr.com
>
> http://www.ortech-engr.com/fuzzy/TilShell.html

Neural network tools

Attrasoft Predictor & Attrasoft DecisionMaker

Neural-network-based tools that use the data in databases or spreadsheets to detect subtle changes, predict results and make business decisions. DecisionMaker is especially good for applications to terabyte or gigabyte databases because of its accuracy and speed. The software does not require any special knowledge of building neural networks.

> Attrasoft
> PO Box 13051
> Savannah, GA 31406, USA
> Fax: +1 (510) 652-6589
> e-mail: webmaster@attrasoft.com
>
> http://attrasoft.com/products.htm

BackPack Neural Network System

Designed for users interested in developing solutions to real business problems using state-of-the-art data mining tools. This system uses a back-propagation algorithm. It reads ASCII

text files and dBASE database files. The system has built-in data preprocessing capabilities, including fuzzy sets, 1-of-N, built-in graphical analysis tools for model evaluation and explanation, thermometer transforms, and training data set creation. A working trial version of BackPack is available as a free download.

Z Solutions, Inc.
6595G Roswell Rd, Suite 662
Atlanta, GA 30328, USA
e-mail: info@zsolutions.com

http://www.zsolutions.com/backpack.htm

BrainMaker

The neural network software for business and marketing forecasting; stock, bond, commodity and futures prediction; pattern recognition; medical diagnosis – almost any activity where the user needs special insight. The user does not need any special programming or computer skills. With more than 25,000 systems sold, BrainMaker is the world's best-selling software for developing neural networks.

California Scientific Software
10024 Newtown Rd
Nevada City, CA 95959, USA
Phone: +1 (530) 478-9040
USA toll free: 1-800-284-8112
Fax: +1 (530) 478-9041
e-mail: sales@calsci.com

http://www.calsci.com/

EasyNN-plus

EasyNN-plus is a neural network software system for Microsoft Windows. It can generate multilayer neural networks from imported files. Numerical data, text or images can be used to create the neural networks. The neural networks can then be trained, validated and queried. All diagrams, graphs and input/output data produced or used by the neural networks can be displayed. The graphs, grid and network diagrams are updated dynamically, so the user can see how everything is working. Neural networks can then be used for data analysis, prediction, forecasting, classification and time-series projection.

Stephen Wolstenholme
18 Seymour Road
Cheadle Hulme
United Kingdom
e-mail: steve@tropheus.demon.co.uk

http://www.easynn.com/easynnplus.html

MATLAB Neural Network Toolbox

The Neural Network Toolbox is a complete neural network engineering environment within MATLAB. It has a modular, open and extensible design that provides comprehensive support for many proven network paradigms such as multilayer perceptrons with back-propagation learning, recurrent networks, competitive layers and self-organising maps. The toolbox has a GUI for designing and managing the networks.

The MathWorks
3 Apple Hill Drive
Natick, MA 01760-2098, USA
Phone: +1 (508) 647-7000
Fax: +1 (508) 647-7001

http://www.mathworks.com/products/neuralnet/

NeuJDesk

NeuJDesk is a general-purpose framework for Java applications. Each application is tailored to the specific JBi bean. Data can be loaded, passed to the bean and results obtained with minimal user knowledge. NeuJDesk supports such paradigms as multilayer perceptron, Kohonen, KMeans, Bayesian classifier, case-based reasoning, and principal components analysis.

Neusciences
Unit 2 Lulworth Business Centre
Nutwood Way Totton
Southampton
SO40 3WW, United Kingdom
Phone: +44 (0) 238-06-64-011
Fax: +44 (0) 238-08-73-707
e-mail: sales@neusciences.com

http://www.ncs.co.uk/Products/NeuJDesk%20introduction.htm

NeuralWorks Predict

NeuralWorks Predict is an integrated, state-of-the-art tool for creating and deploying prediction and classification applications. Predict combines neural network technology with genetic algorithms, statistics and fuzzy logic to find optimal or near-optimal solutions automatically for a wide range of problems. Predict requires no prior knowledge of neural networks. For advanced users, Predict also offers direct access to all key training and network parameters. In Microsoft Windows environments, NeuralWorks Predict can be run either as an add-in for Microsoft Excel to take advantage of Excel's data-handling capabilities, or as a command line program. In Unix and Linux environments, Neural-Works Predict runs as a command line program.

NeuralWare
230 East Main Street, Suite 200
Carnegie, PA 15106-2700, USA
Phone: +1 (412) 278-6280
Fax: +1 (412) 278-6289
e-mail: info@neuralware.com

http://www.neuralware.com/products.jsp

NeuroCoM

The Neuro Control Manager (NeuroCoM) is a high-performance tool for developing and testing neural networks. The NeuroCoM has a window-oriented GUI that facilitates both neural network training and its analysis. This interface also helps to visualise the neural network architecture, transfer functions and the learning process. The NeuroCoM can generate a source code in C.

TransferTech GmbH
Cyriaksring 9A
D-38118 Braunschweig, Germany
Phone: +49 (531) 890-255
Fax: +49 (531) 890-355
e-mail: info@transfertech.de

http://www.transfertech.de/wwwe/ncm/ncme_scr.htm

NeuroForecaster

NeuroForecaster (NF) is a windows-based, user-friendly and intelligent neural network forecasting tool. It incorporates neural network and fuzzy computing. The NF can be used for time-series forecasting (e.g. stock and currency market forecasts, GDP forecast), classification (e.g. stock selection, bond rating, credit assignment, property valuation) and indicator analysis. The NF can read Excel, MetaStock, CSI, Computrac and ASCII data files directly.

NewWave Intelligent Business Systems, NIBS Inc.
e-mail: info@kDiscovery.com

http://web.singnet.com.sg/~midaz/nfga.htm

NeuroShell 2

Combines powerful neural network architectures, a user interface driven by Microsoft Windows icons, sophisticated utilities and popular options to give users the ultimate neural network experimental environment. It is recommended for academic users, or for users who are concerned with classic neural network paradigms like back-propagation. Users interested in solving real problems should consider the NeuroShell Predictor, NeuroShell Classifier or the NeuroShell Trader.

Ward Systems Group, Inc.
Executive Park West
5 Hillcrest Drive
Frederick, MD 21703, USA
Phone: +1 (301) 662-7950
Fax: +1 (301) 663-9920
e-mail: sales@wardsystems.com

http://www.wardsystems.com/products.asp

NeuroSolutions

This software combines a modular, icon-based network-design interface with an implementation of learning procedures, such as recurrent back-propagation and back-propagation through time. Other features include GUI and C++ source-code generation. There are six levels of NeuroSolutions: the Educator, the entry level intended for those who want to learn about neural networks; the Users level, which extends the Educator with a variety of neural models for static pattern recognition applications; and the Consultants level that offers enhanced models for dynamic pattern recognition, time-series prediction and process control.

NeuroDimension, Inc.
1800 N. Main Street, Suite D4
Gainesville, FL 32609, USA

Phone: +1 (352) 377-5144
USA toll free: 1-800-634-3327
Fax: +1 (352) 377-9009
e-mail: info@nd.com

http://www.neurosolutions.com/products/ns

Partek Discover and Partek Predict

Partek Discover provides visual and numerical analyses of clusters in the data. Also useful for mapping high-dimensional data to a lower dimension for visualisation, analysis or modelling. Partek Predict is a tool for predictive modelling that determines an optimal set of variables to be used. It provides several methods for variable selection, including statistical methods, neural networks and genetic algorithms.

Partek Incorporated
4 Research Park Drive, Suite 100
St Charles, MO 63304, USA
Phone: +1 (636) 498-2329
Fax: +1 (636) 498-2331
e-mail: information@partek.com

http://www.partek.com/html/products/products.html

STATISTICA Neural Networks

STATISTICA Neural Networks is the most technologically advanced and best-performing neural networks application on the market. It offers numerous unique advantages and will appeal not only to neural network experts (by offering them an extraordinary selection of network types and training algorithms), but also to new users in the field of neural computing (via the unique *Intelligent Problem Solver*, a tool that can guide the user through the procedures for creating neural networks).

StatSoft, Inc.
2300 East 14th Street
Tulsa, OK 74104, USA
Phone: +1 (918) 749-1119
Fax: +1 (918) 749-2217
e-mail: info@statsoft.com

http://www.statsoft.com/stat_nn.html

THINKS and ThinksPro

THINKS is a personal neural development environment. It can also be used as an excellent teaching tool. With options on network architecture and processing element definition, the experienced user can quickly experiment with novel network configurations. ThinksPro is a professional neural network development environment. It offers dynamic graphing and visualisation tools to view continually inputs, weights, states and outputs in a number of formats, illustrating the learning process. A 30-day trial version of ThinksPro is available as a free download.

Logical Designs Consulting, Inc.
Advanced Investment Technologies Center
5666 La Jolla Blvd, Suite 107

La Jolla, CA 92037, USA

http://www.sigma-research.com/bookshelf/rtthinks.htm

Evolutionary computation tools

ActiveX Genetic Programming Control!

Enables the user to build his/her own 'genetic programs' with any OCX or ActiveX programming language. The user just has to provide the grammar in a plain text file and add his/her raw fitness evaluation function. A manual and a sample application are available as free downloads.

Hanke & Hörner Software Solutions
Pouthongasse 12/26
A-1150 Vienna, Austria
Phone: +43 (1) 789-5117
Fax: +43 (1) 789-5117-11
e-mail: info@hhsoft.com

http://www.hhsoft.com/

Evolutionary Optimizer

The Evolutionary Optimizer (EVO) is a generic tool for optimising system properties determined by numerical parameters. The system provides a graphical user-friendly interface and requires no knowledge of programming.

TransferTech GmbH
Cyriaksring 9A
D-38118 Braunschweig, Germany
Phone: +49 (531) 890-255
Fax: +49 (531) 890-355
e-mail: info@transfertech.de

http://www.transfertech.de/wwwe/evo/evoe_scr.htm

Evolver

An optimisation add-in for Microsoft Excel. Evolver uses genetic algorithms to quickly solve complex optimisation problems in finance, distribution, scheduling, resource allocation, manufacturing, budgeting, engineering, and more. Virtually any type of problem that can be modelled in Excel can be solved by Evolver. It requires no knowledge of programming or genetic algorithm theory and ships with a fully illustrated manual, several examples and free, unlimited technical support.

Palisade Corporation
31 Decker Road
Newfield, NY 14867, USA
Phone: +1 (607) 277-8000
USA/Canada toll-free: 1-800-432-7475
Fax: +1 (607) 277-8001
sales@palisade.com

http://www.palisade.com/html/evolver.html

GEATbx

The Genetic and Evolutionary Algorithm Toolbox (GEATbx) for use with MATLAB is the most comprehensive implementation of evolutionary algorithms in MATLAB. A broad range of operators is fully integrated into the environment, which constitutes a powerful optimisation tool applicable to a wide range of problems.

> T&R Computer-Vertrieb GmbH
> Klaistower Strasse 64/65
> D-14542 Glindow, Germany
> Phone: +49 (3) 327-468-0189
> Fax: +49 (3) 327-434-89
> e-mail: t&r@geatbx.com
>
> http://www.geatbx.com/

GeneHunter

A powerful solution for optimisation problems. GeneHunter includes an Excel add-in which allows the user to run an optimisation problem from an Excel Release 7, Excel 97 or Excel 2000 spreadsheet, as well as a dynamic link library of genetic algorithm functions that may be called from programming languages such as Microsoft Visual Basic or C.

> Ward Systems Group, Inc.
> Executive Park West
> 5 Hillcrest Drive
> Frederick, MD 21703, USA
> Phone: +1 (301) 662-7950
> Fax: +1 (301) 663-9920
> e-mail: sales@wardsystems.com
>
> http://www.wardsystems.com/products.asp

Generator

A general-purpose genetic algorithm program. It is useful for solving a wide variety of problems, such as optimisation, curve fitting, scheduling, stock market projections, electronic circuit design, neural network design, business productivity and management theories.

> New Light Industries, Ltd
> 9715 W. Sunset Highway
> Spokane, WA 99224, USA
> Phone: +1 (509) 456-8321
> Fax: +1 (509) 456-8351
>
> http://myweb.iea.com/~nli/

Genetic Server and Genetic Library

Provides a general-purpose API for genetic algorithm design. The Genetic Server is an ActiveX component that can be used easily to build a custom genetic application in Visual Basic. Genetic Library is a C++ library that can be used to build custom genetic applications in C++.

> NeuroDimension, Inc.
> 1800 N. Main Street, Suite D4
> Gainesville, FL 32609, USA
> Phone: +1 (352) 377-5144
> USA toll free: 1-800-634-3327

Fax: +1 (352) 377-9009

e-mail: info@nd.com

http://www.nd.com/products/genetic.htm

GenSheet

Implements genetic algorithms as fast C-coded dynamic link libraries. GenSheet supports genetic operations for binary, integer, real and permutation representations, and includes special commands for constrained non-linear optimisation, genetic classifiers, job-shop scheduling and computing minimum variance portfolio. GenSheet requires Microsoft Excel. All GenSheet commands are configured in an easy-to-use Excel menubar. GenSheet provides interactive help and a tutorial.

Inductive Solutions, Inc.

380 Rector Place, Suite 4A

New York, NY 10280, USA

Phone: +1 (212) 945-0630

Fax: +1 (212) 945-0367

e-mail: roy@inductive.com

http://www.inductive.com/softgen.htm

Sugal

Sugal is the SUnderland Genetic ALgorithm system. The aim of Sugal is to support research and implementation in genetic algorithms on a common software platform. It is written in ANSI C; the source code is provided. Sugal supports multiple data types: bit strings, integers, real numbers, symbols (from arbitrarily sized alphabets) and permutations. It provides a platform-independent GUI, including fitness and diversity graphing facilities. The Sugal 2.1 source code and manual are available as free downloads.

Dr Andrew Hunter

University of Durham

South Road

Durham DH1 3LE

United Kingdom

Phone: +44 (1) 91-334-1723

Fax: +44 (1) 91-334-1701

e-mail: andrew1.hunter@durham.ac.uk

http://www.dur.ac.uk/andrew1.hunter/Sugal/

XpertRule

An expert system shell with embedded genetic algorithms. The system combines the power of genetic algorithms in evolving solutions with the power of rule-based expert systems in solving scheduling and optimisation problems.

Attar Software UK	Intellicrafters (Attar Software USA)
Newlands Road	Renaissance International Corporation
Leigh WN7 4HN, England	Newburyport, MA 01950, USA
Phone: 44 (0) 870-60-60-870	Phone: +1 (978) 465-5111
Fax: 44 (0) 870-60-40-156	Fax: +1 (978) 465-0666
e-mail: info@attar.co.uk	e-mail: info@IntelliCrafters.com

http://www.attar.com/

Index