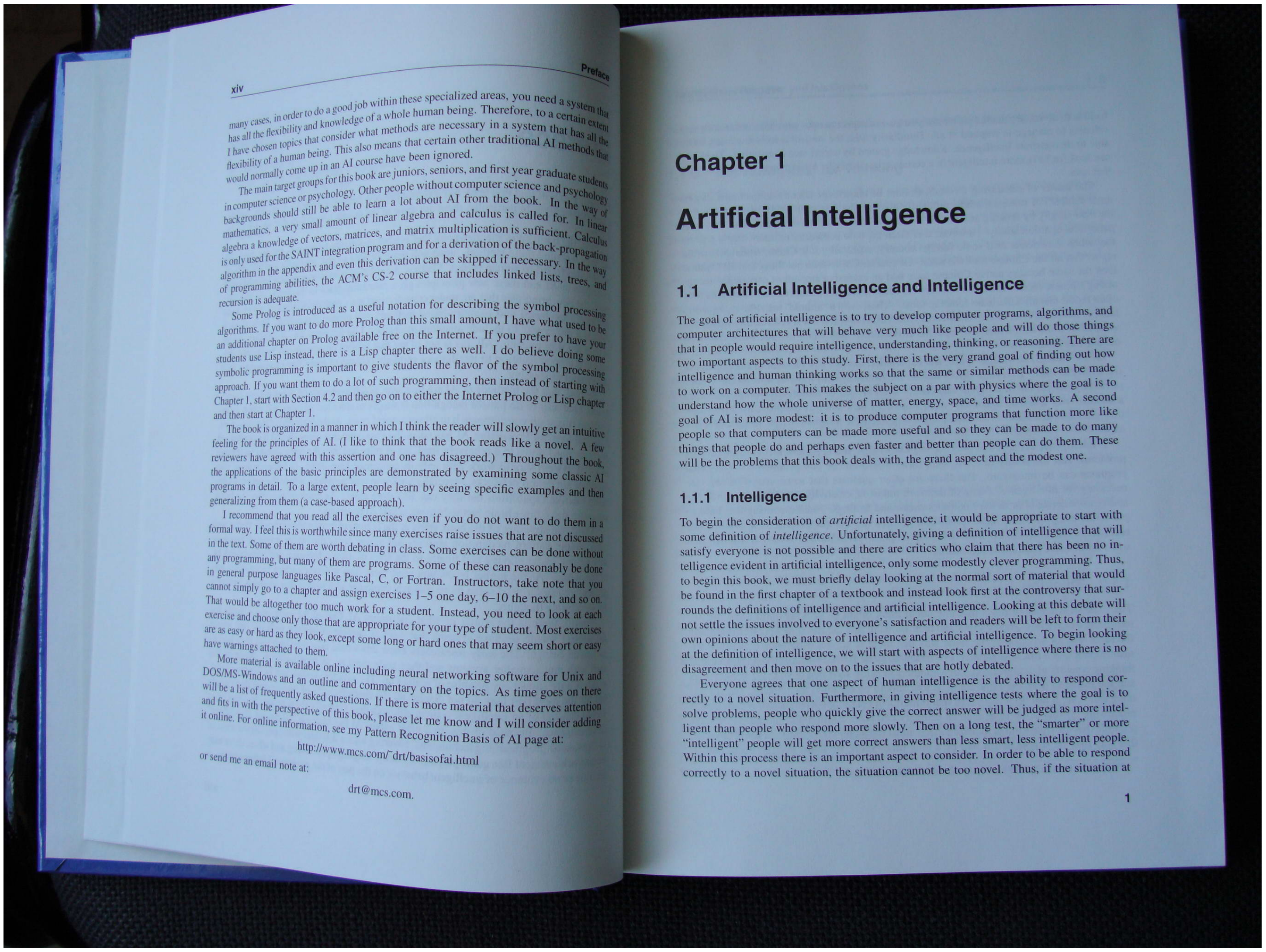


many cases, in order to do a good job within these specialized areas, you need a system that has all the flexibility and knowledge of a whole human being. Therefore, to a certain extent I have chosen topics that consider what methods are necessary in a system that has all the flexibility of a human being. This also means that certain other traditional AI methods that would normally come up in an AI course have been ignored.

The main target groups for this book are juniors, seniors, and first year graduate students in computer science or psychology. Other people without computer science and psychology backgrounds should still be able to learn a lot about AI from the book. In the way of mathematics, a very small amount of linear algebra and calculus is called for. In linear algebra a knowledge of vectors, matrices, and matrix multiplication is sufficient. Calculus is only used for the SAINT integration program and for a derivation of the back-propagation algorithm in the appendix and even this derivation can be skipped if necessary. In the way of programming abilities, the ACM's CS-2 course that includes linked lists, trees, and recursion is adequate.

Some Prolog is introduced as a useful notation for describing the symbol processing algorithms. If you want to do more Prolog than this small amount, I have what used to be an additional chapter on Prolog available free on the Internet. If you prefer to have your students use Lisp instead, there is a Lisp chapter there as well. I do believe doing some symbolic programming is important to give students the flavor of the symbol processing approach. If you want them to do a lot of such programming, then instead of starting with Chapter 1, start with Section 4.2 and then go on to either the Internet Prolog or Lisp chapter and then start at Chapter 1.

The book is organized in a manner in which I think the reader will slowly get an intuitive feeling for the principles of AI. (I like to think that the book reads like a novel. A few reviewers have agreed with this assertion and one has disagreed.) Throughout the book, the applications of the basic principles are demonstrated by examining some classic AI programs in detail. To a large extent, people learn by seeing specific examples and then generalizing from them (a case-based approach).

I recommend that you read all the exercises even if you do not want to do them in a formal way. I feel this is worthwhile since many exercises raise issues that are not discussed in the text. Some of them are worth debating in class. Some exercises can be done without any programming, but many of them are programs. Some of these can reasonably be done in general purpose languages like Pascal, C, or Fortran. Instructors, take note that you cannot simply go to a chapter and assign exercises 1–5 one day, 6–10 the next, and so on. That would be altogether too much work for a student. Instead, you need to look at each exercise and choose only those that are appropriate for your type of student. Most exercises are as easy or hard as they look, except some long or hard ones that may seem short or easy have warnings attached to them.

More material is available online including neural networking software for Unix and DOS/MS-Windows and an outline and commentary on the topics. As time goes on there will be a list of frequently asked questions. If there is more material that deserves attention and fits in with the perspective of this book, please let me know and I will consider adding it online. For online information, see my Pattern Recognition Basis of AI page at:

http://www.mcs.com/˜drt/basisofai.html

or send me an email note at:

drt@mcs.com.

Chapter 1

Artificial Intelligence

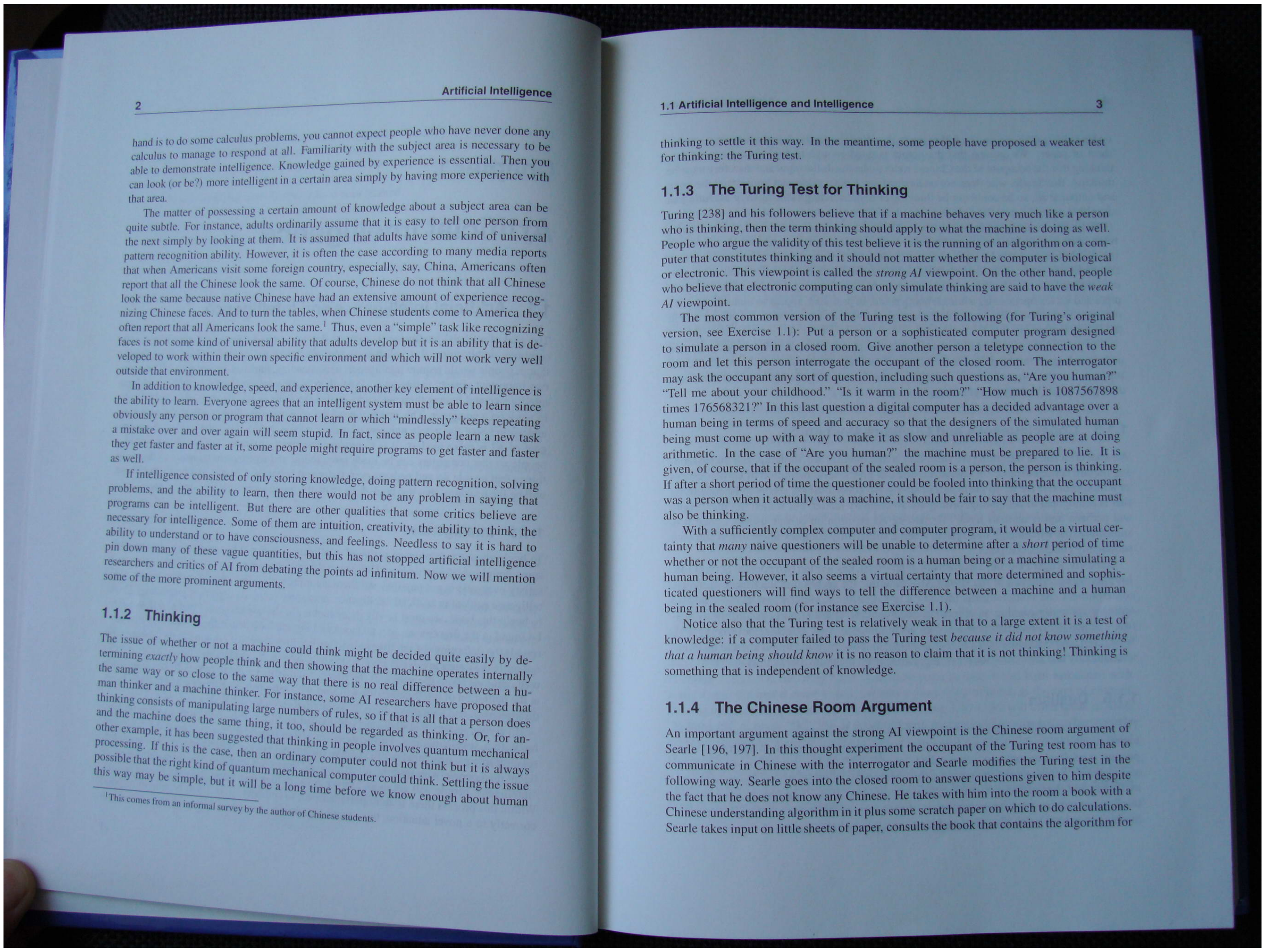
1.1 Artificial Intelligence and Intelligence

The goal of artificial intelligence is to try to develop computer programs, algorithms, and computer architectures that will behave very much like people and will do those things that in people would require intelligence, understanding, thinking, or reasoning. There are two important aspects to this study. First, there is the very grand goal of finding out how intelligence and human thinking works so that the same or similar methods can be made to work on a computer. This makes the subject on a par with physics where the goal is to understand how the whole universe of matter, energy, space, and time works. A second goal of AI is more modest: it is to produce computer programs that function more like people so that computers can be made more useful and so they can be made to do many things that people do and perhaps even faster and better than people can do them. These will be the problems that this book deals with, the grand aspect and the modest one.

1.1.1 Intelligence

To begin the consideration of *artificial* intelligence, it would be appropriate to start with some definition of *intelligence*. Unfortunately, giving a definition of intelligence that will satisfy everyone is not possible and there are critics who claim that there has been no intelligence evident in artificial intelligence, only some modestly clever programming. Thus, to begin this book, we must briefly delay looking at the normal sort of material that would be found in the first chapter of a textbook and instead look first at the controversy that surrounds the definitions of intelligence and artificial intelligence. Looking at this debate will not settle the issues involved to everyone's satisfaction and readers will be left to form their own opinions about the nature of intelligence and artificial intelligence. To begin looking at the definition of intelligence, we will start with aspects of intelligence where there is no disagreement and then move on to the issues that are hotly debated.

Everyone agrees that one aspect of human intelligence is the ability to respond correctly to a novel situation. Furthermore, in giving intelligence tests where the goal is to solve problems, people who quickly give the correct answer will be judged as more intelligent than people who respond more slowly. Then on a long test, the "smarter" or more "intelligent" people will get more correct answers than less smart, less intelligent people. Within this process there is an important aspect to consider. In order to be able to respond correctly to a novel situation, the situation cannot be too novel. Thus, if the situation at



hand is to do some calculus problems, you cannot expect people who have never done any calculus to manage to respond at all. Familiarity with the subject area is necessary to be able to demonstrate intelligence. Knowledge gained by experience is essential. Then you can look (or be?) more intelligent in a certain area simply by having more experience with that area.

The matter of possessing a certain amount of knowledge about a subject area can be quite subtle. For instance, adults ordinarily assume that it is easy to tell one person from the next simply by looking at them. It is assumed that adults have some kind of universal pattern recognition ability. However, it is often the case according to many media reports that when Americans visit some foreign country, especially, say, China, Americans often report that all the Chinese look the same. Of course, Chinese do not think that all Chinese look the same because native Chinese have had an extensive amount of experience recognizing Chinese faces. And to turn the tables, when Chinese students come to America they often report that all Americans look the same.[1] Thus, even a "simple" task like recognizing faces is not some kind of universal ability that adults develop but it is an ability that is developed to work within their own specific environment and which will not work very well outside that environment.

In addition to knowledge, speed, and experience, another key element of intelligence is the ability to learn. Everyone agrees that an intelligent system must be able to learn since obviously any person or program that cannot learn or which "mindlessly" keeps repeating a mistake over and over again will seem stupid. In fact, since as people learn a new task they get faster and faster at it, some people might require programs to get faster and faster as well.

If intelligence consisted of only storing knowledge, doing pattern recognition, solving problems, and the ability to learn, then there would not be any problem in saying that programs can be intelligent. But there are other qualities that some critics believe are necessary for intelligence. Some of them are intuition, creativity, the ability to think, the ability to understand or to have consciousness, and feelings. Needless to say it is hard to pin down many of these vague quantities, but this has not stopped artificial intelligence researchers and critics of AI from debating the points ad infinitum. Now we will mention some of the more prominent arguments.

1.1.2 Thinking

The issue of whether or not a machine could think might be decided quite easily by determining *exactly* how people think and then showing that the machine operates internally the same way or so close to the same way that there is no real difference between a human thinker and a machine thinker. For instance, some AI researchers have proposed that thinking consists of manipulating large numbers of rules, so if that is all that a person does and the machine does the same thing, it too, should be regarded as thinking. Or, for another example, it has been suggested that thinking in people involves quantum mechanical processing. If this is the case, then an ordinary computer could not think but it is always possible that the right kind of quantum mechanical computer could think. Settling the issue this way may be simple, but it will be a long time before we know enough about human

[1] This comes from an informal survey by the author of Chinese students.

thinking to settle it this way. In the meantime, some people have proposed a weaker test for thinking: the Turing test.

1.1.3 The Turing Test for Thinking

Turing [238] and his followers believe that if a machine behaves very much like a person who is thinking, then the term thinking should apply to what the machine is doing as well. People who argue the validity of this test believe it is the running of an algorithm on a computer that constitutes thinking and it should not matter whether the computer is biological or electronic. This viewpoint is called the *strong AI* viewpoint. On the other hand, people who believe that electronic computing can only simulate thinking are said to have the *weak AI* viewpoint.

The most common version of the Turing test is the following (for Turing's original version, see Exercise 1.1): Put a person or a sophisticated computer program designed to simulate a person in a closed room. Give another person a teletype connection to the room and let this person interrogate the occupant of the closed room. The interrogator may ask the occupant any sort of question, including such questions as, "Are you human?" "Tell me about your childhood." "Is it warm in the room?" "How much is 1087567898 times 176568321?" In this last question a digital computer has a decided advantage over a human being in terms of speed and accuracy so that the designers of the simulated human being must come up with a way to make it as slow and unreliable as people are at doing arithmetic. In the case of "Are you human?" the machine must be prepared to lie. It is given, of course, that if the occupant of the sealed room is a person, the person is thinking. If after a short period of time the questioner could be fooled into thinking that the occupant was a person when it actually was a machine, it should be fair to say that the machine must also be thinking.

With a sufficiently complex computer and computer program, it would be a virtual certainty that *many* naive questioners will be unable to determine after a *short* period of time whether or not the occupant of the sealed room is a human being or a machine simulating a human being. However, it also seems a virtual certainty that more determined and sophisticated questioners will find ways to tell the difference between a machine and a human being in the sealed room (for instance see Exercise 1.1).

Notice also that the Turing test is relatively weak in that to a large extent it is a test of knowledge: if a computer failed to pass the Turing test *because it did not know something that a human being should know* it is no reason to claim that it is not thinking! Thinking is something that is independent of knowledge.

1.1.4 The Chinese Room Argument

An important argument against the strong AI viewpoint is the Chinese room argument of Searle [196, 197]. In this thought experiment the occupant of the Turing test room has to communicate in Chinese with the interrogator and Searle modifies the Turing test in the following way. Searle goes into the closed room to answer questions given to him despite the fact that he does not know any Chinese. He takes with him into the room a book with a Chinese understanding algorithm in it plus some scratch paper on which to do calculations. Searle takes input on little sheets of paper, consults the book that contains the algorithm for understanding Chinese, and by following its directions he produces some output on another sheet of paper. We assume that the output is good enough to fool almost anyone into thinking that the occupant of the Chinese room understands the input and therefore must be thinking. But Searle, who does not understand any Chinese does not understand the input and output at all, so he could not be thinking or understanding. Thus merely executing an algorithm, even if it gets the right answers, should not constitute understanding or thinking.

Believers in strong AI then reply that while Searle does not understand, it is the whole room, including Searle, the algorithm in the book, and the scratch paper that is understanding. Searle counters this by saying he could just as well memorize the rules, do away with the pencil and paper, and do all the calculations in his head, but he still would not be understanding Chinese. Searle takes the point even farther by noting that a room full of water pipes and valves operated by a human being could, in principle, appear to understand without actually understanding as a real Chinese person would understand if that person was in the Turing test room.

The point of the argument is that merely executing some algorithm should not constitute understanding or thinking, understanding and thinking require something more. Searle supposes that the something more in people comes from having the right kind of hardware, the right kind of biology and chemistry.

1.1.5 Consciousness and Quantum Mechanics

Another criticism of the strong AI viewpoint is that intelligence, thinking, and understanding require consciousness. Of course, no one can give a solid definition of consciousness or a foolproof test for it. To the critics of strong AI, consciousness seems to be something that is orthogonal to computing, orthogonal to ordinary matter, but something that people and perhaps higher animals have. The strong AI position on consciousness is that it is something that will emerge in a system when a sufficiently complex algorithm is run on a sufficiently complex computer.

Recently, Roger Penrose, a mathematical physicist, has written two popular books [148, 149] giving his criticism of the strong AI viewpoint. He argues that intelligence requires consciousness and consciousness involves a nonalgorithmic element, an element that no ordinary computer running an algorithm can duplicate. Furthermore, according to Penrose, the nonalgorithmic element involves quantum mechanical effects. Lockwood [101], Wolf [263], and Nanopoulos [133] also speculate on how the mind might operate quantum mechanically and how consciousness might arise from quantum mechanical effects.

1.1.6 Dualism

The 17th century philosopher and mathematician, Rene Descartes, was a proponent of the idea that there is more to a human being than just plain matter, there is an additional component, a spiritual component, often called "mind-stuff." In his conception, the spiritual and material components of the mind can interact with each other. A few researchers such as Eccles and Popper (see [34]) take this position now. If thinking, consciousness, and intelligence require a spiritual component, then it may be difficult or impossible to get a machine to behave much like a human being.

With all this disagreement on what constitutes intelligence, thinking, and understanding, it will be some time before satisfactory definitions are worked out.

1.2 Association

The principle of association may be the most important principle used in intelligence. Briefly put, given that a set of ideas is present in the mind, this set will cause some new idea to come to mind, an idea that has been associated with the set of ideas in the past. This most important principle has been known for hundreds or even thousands of years, but perhaps the best early detailed description was given by a famous 19th century philosopher/psychologist, William James, in his two-volume series, *The Principles of Psychology* [72] (and the abridged one volume version, *Psychology* [73]). In effect, James and other psychologists and philosophers had a very high-level solution to the AI problem by the 19th century, however, they were unfortunate in that there were no computers available at the time with which they could make their ideas more concrete.

In the excerpt below, James gives the principles of association:

> *When two elementary brain processes have been active at the same time, or in immediate succession, one of them, upon reawakening, tends to propagate its excitement into the other.*
>
> But, as a matter of fact, every elementary process has unavoidably found itself at different times excited in conjunction with *many* other processes. Which of these others it shall now awaken becomes a problem. Shall *b* or *c* be aroused by the present *a*? To answer this, we must make a further postulate, based on the fact of *tension* in nerve tissue, and on the fact of summation of excitements, each incomplete or latent in itself, into an open resultant. The process *b*, rather than *c*, will awake, if in addition to the vibrating tract *a* some other tract *d* is in a state of subexcitement, and formerly was excited with *b* alone and not with *a*.

Instead of thinking of summing up "tension" in nerve tissue, today we would think of summing up voltages or currents.

These principles can be better understood with Figure 1.1. If we activate the idea, a, and its only associations in the past are with the idea b with an activation strength of 0.25 and with c with an activation strength of 0.40, then c must come to mind. Think, as James does, as if "tension" or electrical current is flowing from a into both b and c. This is shown in Figure 1.1(a). On the other hand, if at some point in time, b had been associated with a and d, and if a *and* d come to life, the idea c must come to mind, as the sum of currents flowing into it is the greatest. This is shown in Figure 1.1(b). We will often speak of ideas "lighting up" or being "lit." This is in accord with conventional terminology where people often say that ideas "light up" in their mind. We will also talk about the "brightest" (highest rated) idea as the one that comes to mind.

Some examples of this summation process are worth looking at. One example from James is what occurs when the old poem, "Locksley Hall," is memorized. Two different lines of the poem are as follows:

I, the heir of all *the ages* in the foremost files of time,

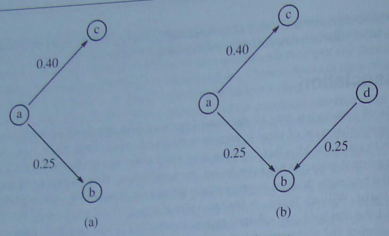


Figure 1.1: Summation of excitements.

and

For I doubt not through *the ages* one increasing purpose runs.

We focus in on the words, "the ages." If a person had memorized this poem and started reciting the first line, and got to the phrase, "the ages," why should he continue with the words, "in the foremost files of time," rather than "one increasing purpose runs?" The answer is simple. While "the ages" points to, or suggests, *both* "in the foremost files of time" and "one increasing purpose runs," there are those words *before* "the ages" that also, but to a smaller extent, point to "in the foremost files of time." The summation of the "the ages" with "I, the heir of all" produces a larger value for "in the foremost files of time" than for "one increasing purpose runs."

A second example from James is the following:

> The writer of these pages has every year to learn the names of a large number of students who sit in alphabetical order in a lecture-room. He finally learns to call them by name, as they sit in their accustomed places. On meeting one in the street, however, early in the year, the face hardly ever recalls the name, but it may recall the place of its owner in the lecture-room, his neighbors' faces, and consequently his general alphabetical position: and then, usually as the common associate of all these combined data, the student's name surges up in his mind.

The principles of association also form the basis for most TV game shows. The principles have been seen there in their purest form in the shows *Password*, *Password Plus*, and *Super Password*. In the simplest version, *Password*, there are two teams of two players each. One player on each team is given a secret word, short phrase, or name and the object of the game is for this person to say a word that will induce the player's teammate to say the secret word, phrase, or name. Whichever team gets the right answer scores some points. For example, in one game the secret name was "Jesse James." The first clue given in the game was "western" but the other person's response was "John Wayne." This is fairly reasonable since in many people's minds, John Wayne is very closely associated

with Westerns. Some other reasonable responses might be "cowboys," "Indians," or "eastern." Since the response was wrong, the other team gets a chance and this time the clue was "train." Adding together the clues "western" and "train," some reasonable responses might be "Santa Fe," "Union Pacific," or "Central Pacific," all famous western train companies, but again the contestant got the wrong answer. Finally, after the clues, "frank," "brother," and "robber" were given, a contestant got the right answer, "Jesse James," a famous train robber in the Old West who had a brother named Frank. The game is summarized in Figure 1.2.

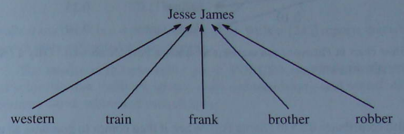


Figure 1.2: A simple game of *Password*.

It is easy to model the process of combining ideas in the following manner. Suppose we assign numeric values to the strength of the associations between ideas. Suppose the associations with "western" are:

John Wayne	0.40
cowboys	0.30
Indians	0.25
Santa Fe	0.25
Jesse James	0.15
Frank James	0.10
Union Pacific	0.05
Central Pacific	0.05

and the associations for "train" are:

Amtrak	0.30
electric	0.25
Santa Fe	0.25
Union Pacific	0.25
Central Pacific	0.25
Jesse James	0.15

If you want to combine the effects of two different clues, like "western" and "train," one simple solution is to simply add up how much is being contributed to each of the other ideas in the lists of ideas. In Figure 1.3 we show how "western" and "train" combine to activate all the ideas with which they have been associated in the past. The numbers to the right in Figure 1.3 show the summations. For instance, "western" contributes 0.25 to "Santa Fe"

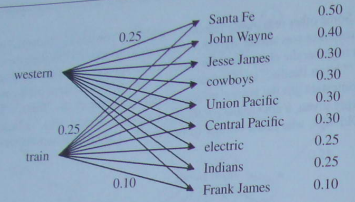


Figure 1.3: How clues in *Password* can combine to produce possible answers. Only a few of the association strengths are shown.

and "train" also contributes 0.25 to "Santa Fe." When it then comes to guessing an answer, the idea with the highest rating is "Santa Fe."

For a final example of these principles we now look at their actual use in a simple AI program. Walker and Amsler [245, 246] created a program called FORCE4 whose purpose is to look at newspaper stories and figure out roughly what the story is about. The program does not acquire any kind of detailed understanding of what the article is about, but only tries to put it in a general category, such as weather, law, politics, manufacturing, and so forth. It makes use of a set of subject codes assigned to specialized word senses in the Longman Dictionary of Contemporary English. For instance, the word "heavy" is often associated with food (coded as FO), meteorology (ML), and theater (TH). "Rainfall" is associated with meteorology (ML). "High" is associated with motor vehicles (AU), drugs and drug experiences (DGXX), food (FO), meteorology (ML), religion (RLXX), and sounds (SN). "Wind" suggests hunting (HFZH), physiology (MDZP), meteorology (ML), music (MU), and nautical (NA).

One story given to FORCE4 was the following:

> Heavy rainfall and high winds clobbered the California coast early today, while a storm system in the Southeast dampened the Atlantic Seaboard from Florida to Virginia.
>
> Travelers' advisories warned of snow in California's northern mountains and northwestern Nevada. Rain and snow fell in the Dakotas, northern Minnesota and Upper Michigan.
>
> Skies were cloudy from Tennessee through the Ohio Valley into New England, but generally clear from Texas into the mid-Mississippi Valley.

For each important word, the program counts one point for each of its associated ideas.

When you apply this procedure to the above story, you get the following counts:

10	ML (Meteorology)
4	GOZG (Geographical terms)
4	DGXX (Drugs and drug experiences)
3	NA (Nautical)
2	MI (Military)
2	FO (Food)
2	GO (Geography)
1	TH (Theatre)

The results show that it is a weather-related story. Walker [245] reports that when over 100 news stories were submitted to the program,

> The results were remarkably good: FORCE4 works well over a variety of subjects—law, military, sports, radio and television—and several different formats—text, tables and even recipes.

1.3 Neural Networking

When scientists became aware that nerve cells pass around electrical pulses, most of them assumed that this activity was used for thinking. Relatively little is known about how networks of nerve cells operate, and determining how networks of nerve cells operate is a major part of the field of neural networking. The second major part of neural networking research centers on the study of computer models of simplified nerve cells. In this book we will deal almost exclusively with the computer-based models.

1.3.1 Artificial Neural Networks

Artificial neural networks represent a way of organizing a large number of simple calculations so that they can be executed in parallel. The calculations are performed by relatively simple processors typically called nodes, artificial neurons, or just neurons. Artificial neurons have a number of input connections and a number of output connections. The input connections serve to activate, or excite, or we may say, "light up" a neuron or they might also try to turn off or inhibit a neuron. An excited neuron then passes this excitement on to other neurons through its output connections. Figures 1.2 and 1.3 can be regarded as diagrams of neural networks. In Figure 1.3 there are the input nodes, "western" and "train," and the outputs are "Santa Fe," "John Wayne," and so forth. The connections between inputs and outputs are called *weights* in neural networking terminology and the value of a weight is what we call the "strength of association."

A simple artificial neuron is shown in Figure 1.4. For artificial neurons, each connection has associated with it a real value called a weight and each neuron has an activation value. The typical algorithm for activating an artificial neuron, j, given a set of input neurons, subscripted by i, and the set of weights, w_{ij}, works as follows. First, find the quantity, net_j, the total input to neuron j by the following formula:

$$net_j = \sum_i w_{ij} o_i$$

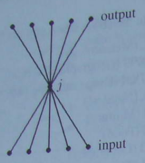


Figure 1.4: A simple artificial neuron, j, with inputs from one set of neurons and outputs to another set.

When the activation value of neuron i, o_i, times the weight w_{ij} is positive, then unit i
serves to activate unit j. On the other hand, when the value of unit i times the weight
w_{ij} is negative, the unit i serves to inhibit unit j. The activation value of neuron, j, is
given by some function, $f(net_j)$. The function, f, may be called an *activation function*,
transfer function, or *squashing function*. One simple activation function is simply to let the
sum of the inputs, net_j, be the activation value of the neuron. This is how the *Password*
network in the last section worked. A second common activation function is to test if net_j
is greater than some threshold (minimum) value, and if it is, the neuron turns on (usually
with an activation value of +1), otherwise it stays off (usually with an activation value of
0). The neural network in Figure 1.5 computes the exclusive-or function and it uses the
activation function, $1/(1+e^{-net_j})$. While this function only reaches 0 and 1 at $-\infty$ and
∞ respectively, when the outputs are close enough to 0 or 1 they are counted as being the
same as 0 or 1. There are many other possible activation functions that can be used.

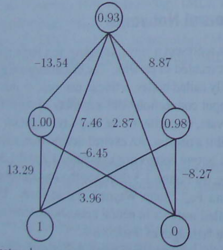


Figure 1.5: This simple network computes the exclusive-or function. The two inputs are made on the bottom layer and the top layer has the answer to within 0.1 of the desired value. In this case the input values are 1 and 0 and the output is 0.93.

Usually the neurons in artificial neural systems have the units arranged in layers as shown in Figures 1.4 and 1.5. These networks have the input layer at the bottom, a *hidden layer* in the middle, and an output layer at the top. The hidden layer gets its name from the

fact that if you view the network as a black box with inputs and outputs that you can monitor, you cannot see the hidden inner workings. When the flow of activation or inhibition goes from the input up to higher-level layers only, the network is said to be a *feed-forward* network. Most often the connections between units are between units in adjacent layers, but it is also possible to have connections between nonadjacent layers and connections within each layer. If there are connections that allow the activation or inhibition to spread down to earlier layers, the network is said to be *recurrent*.

Learning in artificial neural systems is accomplished by modifying the values of the weights connecting the neurons and sometimes by adding extra neurons and weights. There have been a number of learning algorithms that have been proposed and tested for neural networks but the most powerful and most generally useful algorithm is the back-propagation algorithm described in Chapter 3.

Currently, neural networks can be used for many pattern recognition applications, such as recognizing letters, rating loan applications, choosing moves to make in a game, and they can even do simple language processing tasks. One system has been used to automatically drive a van along interstate highways. So far at least, networks are not very well suited to doing complex symbol processing tasks like arithmetic, algebra, understanding natural language, or any task that requires more than a single step of pattern recognition. To produce a system capable of doing much of what human beings can do will require at the very least more complex models and a multitude of different subsystems, each tuned to perform slightly different tasks and all working together.

1.3.2 Biological Neural Networks

As mentioned above, relatively little is known about how biological neural networks operate. For quite a long time it has been assumed that the neurons in the human brain act like the artificial neurons in that they pass around electrical signals, but whereas an artificial neuron receives a single real-value input from each of its input neurons, the biological neurons pass around simple pulses, pulses that are either present or not present. The number of pulses per second going along a connection is an indication of the weight of the connection—more pulses mean a higher weight, fewer pulses indicate a lower weight. In this theory each neuron acts like a little switch, when enough pulses are input a neuron outputs a pulse. The estimates are that there are around 100 billion neurons in the brain with about 1000 connections per neuron and each neuron switches about 100 times per second. This gives a processing rate of around 10^{16} bits per second.[2] But biological neurons are more complicated than simple switches since they are influenced by chemicals within the cell. One recent discovery is that at least some cells involved in vision are not just sending out plain pulses but in fact are passing around coded messages.[3] The shape of the pulse codes the message.

Theories by Hameroff et al. [57] and [133] have each cell acting as a small computer rather than as just a simple switch. The computing would be done in microtubules that make up the cell's cytoskeleton. In this case they estimate a single neuron is processing

[2] This estimate is taken from the article by Hameroff et al. [57] which in turn was taken from [127].

[3] A simple description can be found in "A New View of Vision" by Christopher Vaughan, *Science News*, Volume 134, July 23, 1988, pages 58–60. There are many other articles on this topic. See [171] for a list of other references.

about 10^{13} bits per second and the whole brain would be processing at least about 10^{23} bits per second assuming there is some redundancy.

From time to time people have made estimates of how many bits of information the human brain can store based on certain assumptions, but since it is most certainly not known how information is stored and processed in the brain, none of these estimates can be taken too seriously.

In short, little is known about what is really going on in the human brain but new research may soon shed a lot of light on what is going on. Whatever is happening, it is much more complicated than the processing done in the current set of artificial neural network models.

1.4 Symbol Processing

For most of the history of artificial intelligence the symbol processing approach has been the most important one. There are several reasons why symbol processing has been the dominant approach to the subject. First, there were a number of highly impressive symbol processing programs done in the early 1960s. Two of these early systems are described in Chapter 8, the SAINT program of Slagle that could do symbolic integration and the Geometry Theorem Proving system of Gelernter and others. In addition, it seemed obvious to process natural language this way since language consists of symbols. The second reason symbol processing has been dominant is that it seemed as if it would be a very long time before artificial neural networks could be designed that could do such impressive things.

The advocates of the symbol processing approach to AI have proposed the Physical Symbol System Hypothesis (PSSH) (see [138], [139], and [40]). It states that symbols, structures of symbols, and rules for combining and manipulating symbols and structures of symbols are the necessary *and sufficient* criteria for creating intelligence. This means that these features and only these features are required for producing intelligent behavior. Advocates of PSSH assume that the human brain is doing nothing more than manipulations of collections of symbols. In current computers the manipulations are done sequentially, but advocates of this position assume that human minds actually do parallel processing of symbols. It is the *Physical* Symbol System Hypothesis because advocates assume that there are physical states in the brain that correspond to the kind of structures that a symbol processing computer program uses. PSSH advocates also assume that although neural hardware implements the symbol processing abilities of the brain, this hardware is too low level to have to worry about. So, just as Pascal programmers do not have to worry about integrated circuits, symbol processing can concern itself with symbols and structures of symbols without worrying about the underlying neural hardware. Of course, symbol processing adherents acknowledge that neural networking *is* important for lower-level tasks like vision and movement.

The techniques used in symbol processing are very similar to those used in programming in conventional languages such as Pascal and Fortran, however symbol processing emphasizes list processing and recursion and symbol processing methods use symbols rather than numbers. Because in the beginning almost all AI was done in symbol processing languages, some people have defined artificial intelligence as symbolic computing. The most important computer language for AI programs has been Lisp (for list processing

language) and a newer language is Prolog (for programming in logic). For the most part we will use Prolog as a notation for some symbol processing algorithms later in the book because Prolog has some built-in pattern recognition capabilities that Lisp does not have.

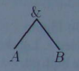
Symbols are defined as unique marks on a piece of paper and in a computer each symbol is represented by a different integer. Two symbols can be equal or not equal, but there are no other relations defined between them. Notice then, that even though symbols are implemented as integers in computer programs, symbols are simpler than the integers that represent the symbols. In addition to being used individually, symbols can also be combined into structures of symbols such as lists or trees. One example of this might be the expression:

$$A\&B.$$

Inside a computer we might find this as a linked list:

$$A \rightarrow \& \rightarrow B \rightarrow nil$$

or as a tree:



Another part of the symbol processing approach is the assumption that there are rules which specify how symbols and structures of symbols are manipulated. Logic and arithmetic provide perfect examples of symbols and how they can be manipulated and combined using rules. Take for example the logic expression, A & B & A. Now within logic, there is a rewrite rule that says that this expression can be rewritten as A & A & B. Moreover, there is a rule that says that A & A can be rewritten as just A. Some rules from arithmetic for manipulating symbols and expressions are: x/x can be replaced by 1, $1 * x$ can be replaced by x, and $x + (-x)$ can be replaced by 0.

The use of rules in symbol processing methods is a key element of the symbol processing approach because everyone who studies human behavior agrees that people exhibit rulelike behavior. For an example of rulelike behavior consider the following case. Suppose a child learns the meaning of a sentence like:

The cat is on the mat.

The child, knowing what a cat is and what a mat is, and what a cat on a mat is, seems to deduce some rules (or form a theory) about how sentences are constructed. Thus, the child can apply the rules and come up with statements like the following:

The dog is on the mat.

The boy is on the mat.

The block is on the mat.

The cat is on the floor.

Some other rules that people form would be "if something is a bird then it can fly" or "if you drop something it will fall." Such facts are typically coded in a rule format something like this:

if bird(X) then flies(X)

if drop(X) then fall(X)

where X stands for the something. For another example concerning language processing, some researchers have studied how children learn to construct the past tenses of verbs and they have come to the conclusion that the errors that children make show that they are producing rules.

From examples like the above and others, traditional AI researchers have concluded that people have some kind of unconscious machinery that deduces rules as well as some kind of symbol processing architecture that applies them.

Notice, though, that rules are little neural networks where the input and output units have symbolic labels as in this rule:

if *a* and *b* then *c*.

The corresponding network is shown in Figure 1.6. It is a two layer network with inputs labeled *a* and *b* and the output unit labeled *c*. Let the two weights be 1 and let the threshold for unit *c* be 1.5. Now if unit *a* and *b* are both on, (= 1), net_c will be 2. Since net_c is greater than 1.5, unit *c* turns on, otherwise it stays off, (= 0). And so it turns out that a key element of symbol processing can be regarded as a form of neural networking.

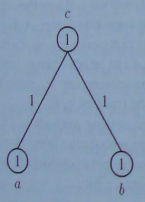


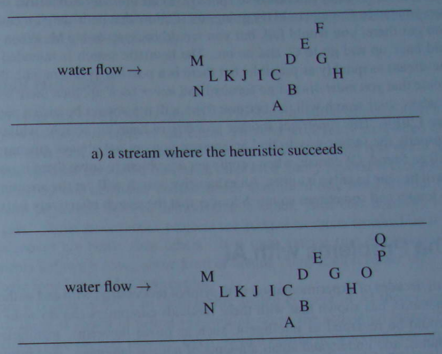
Figure 1.6: The rule, if *a* and *b* then *c*, can be regarded as a neural network where the units *a*, *b*, and *c* can take on the values 0 or 1, the two weights are +1, and the threshold for unit *c* is 1.5.

Symbol processing techniques have been somewhat successful at doing a number of very narrow but useful tasks involving reasoning and processing natural language.

1.5 Heuristic Search

When people encounter a problem they typically have to do some trial and error work on the problem to find the solution. People look at some of the most likely possible solutions to the problem, not every possible solution. However, a simple computer program is dumb in that it does not have any way of evaluating the possible solutions to determine which are

the likely ones. This type of program must do an *exhaustive search* of all the possibilities. Very early on it was recognized that for computer programs to solve problems as human beings do, the programs must be able to look at only the likely possibilities. Programs that use some method to evaluate the possibilities are said to do *heuristic search*.



a) a stream where the heuristic succeeds

b) a stream where the heuristic fails

Figure 1.7: In trying to cross a stream in a heavy fog by stepping on rocks, one heuristic is to keep trying to move forward and never back up. Rocks are indicated by letters. In a) the heuristic succeeds but in b) never backing up will produce a failure.

As an example, suppose you were hiking through the wilderness and you came upon a small stream that you needed to cross. Suppose there are small flat rocks in the river and you can step from one rock to an adjacent rock. For example, there is the situation shown in Figure 1.7(a) where each rock is indicated by a letter. Suppose you are on the river bank at the bottom and you want to get to the river bank at the top. A human being will "eyeball" the situation and have it solved in a second. The best (and only) path is from A to B to C to D to E to F. How would you have a computer look at the same situation and find that path? To find one computer solution we could make the problem harder. Suppose you come upon this place in the river, but there is fog and the fog is so thick that you can only see one rock ahead of you. You are clearly going to have to start making guesses as to which steps to take. You will realize that *probably* the best thing to do is to keep going forward as much as possible. What sense would it make to go back in the direction that you came from, or up or down the river? You could tell which way was forward by noticing that the river flows from left to right, so when you make a move, you should try to keep upstream on your left and downstream on your right. Therefore, when you get to rock C, the best thing to do is to go on to D and not to I. Again, when you get to rock E, you will go on to F, rather than back up by going to G. A computer program could use the same strategy for finding a path across the river and it would find a path as easily as a person lost in a fog. Both you and the computer were doing a heuristic search of a tree, looking for a goal node. If you did an ordinary search of the tree rather than a heuristic search of the tree, you would find a path across, but probably not nearly as quickly. In an ordinary exhaustive search of the tree, when you get to C, you could try going to I. Follow that path and you could go to N. When you got there, you would fail, but you would back up to try M. When that failed, you would back up and go to C, and so on. The heuristic search is intended to get you across the stream as quickly as possible, but there is a possible problem with this method. If you decide that you *must always* go forward and *never* back up, then there will be some locations where your search will fail because there will not always be such a path available (see Figure 1.7(b)). This illustrates another property of heuristic search: while a heuristic search is usually the fastest way to find an answer, you are not always guaranteed that an answer will be found. Of course, when people get a problem to solve there is no guarantee that they will be able to solve it either. An exhaustive search will get the answer but it may take much longer and sometimes so much longer that the search effectively fails.

1.6 The Problems with AI

The results of decades of experimentation with symbol processing, with and without heuristic search methods, has shown that with these methods computers can do *some* tasks that in people would be regarded as intelligent, such as prove theorems, manipulate mathematical formulas, and understand small amounts of natural language. According to some researchers' definitions of intelligence, these programs display intelligence. On the other hand, such programs typically do not learn from their activities and since learning is a key factor in intelligence critics do not see any intelligence in such programs. Then too, the level of understanding that these programs have is severely limited. For example, if you give a program a statement like "John ate up the street," the program might easily conclude that John was eating asphalt. Or, given that "John was in the 100-meter butterfly," a program might think that John was inside a large insect rather than in a swimming event. People say that such programs that do not have the common sense knowledge that people have are *brittle*. In response to this criticism, most symbol processing researchers say they believe their basic methods are valid and the problems can be eliminated by just producing much larger systems. At the moment a very ambitious project known as CYC (see [99]) is attempting to produce a program with a very large number of facts and rules about the world that hopefully will not be brittle. The early estimate was that the program would need about a million rules. As of 1993,[4] the program had two million and work is continuing at the present time.

1.7 The New Proposals

So while AI has problems, some AI researchers remain optimistic about symbol processing methods but other AI researchers are not and they have started looking into a variety of

[4] *Computerworld*, May 10, 1993, pages 104–105.

new proposals. Most of these new proposals come to mind quite easily by just denying the elements of the Physical Symbol System Hypothesis. These ideas are that thinking and intelligence require the use of real numbers, not just symbols; that people use images or pictures, not just structures of symbols; and that they use specific cases or memories, not just rules. In addition, there is another proposal that human thinking involves quantum mechanics and quantum mechanics adds extra capabilities that ordinary computing, not even analog computing, can account for.

1.7.1 Real Numbers

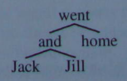
Symbol processing works only with symbols and the only relation defined between symbols is equality, two symbols are equal or not equal. This can work fairly well when the answer to every question is a nice true or false, but in many situations in the real world judgments are fuzzy. The accused person on trial must be proved guilty beyond a reasonable doubt. Music produced by one composer sounds better than the music from another composer. One crime will be judged as more heinous than another. New cars normally look better than older cars. Some government projects are judged as more worthwhile than others. Some chess moves are better than others. It is only natural that researchers think that such judgments are made using some kind of analog computation rather than the simple true/false logic found in symbol processing.

There are a couple of ways to include this analog concept in AI theory, but the most important of these is found in neural networking when the activation values of the units and the weights take on real values. Even though neural networking was present at the beginning of AI research, it was quickly abandoned in favor of symbol processing methods and so it has hardly been investigated until recently. When neural networking methods are applied to AI-type problems it is called *connectionist AI*. One variation on connectionist AI is *parallel distributed processing*, or PDP for short. It uses a specific type of coding within networks.

These methods are fairly good at doing a single step of pattern recognition, but they present connectionist AI researchers with quite a problem as to how to store complicated facts because unlike symbol processing AI where you can use a tree structure to store a fact, in a neural network the facts must be represented as a vector or matrix of real numbers. So for example, if you needed to store away the fact that:

Jack and Jill went home,

it is very straightforward in a digital computer to produce one kind or another of tree structure to represent this such as:



So far there is no established good way to represent this tree structure as a vector or matrix of real numbers, although there are some proposals along these lines.

One position on neural networking is that networks do have some features that are required for intelligence, thinking and reasoning, but conventional symbol processing is also necessary. In this case the Physical Symbol System Hypothesis is wrong at the point where it says that symbol processing is *sufficient*. One proposal is that the mind may be basically a connectionist computer architecture, but it simulates a symbol processing architecture to do those tasks that are most suited for symbol processing, while still using connectionist methods for other types of problems.

A more extreme position is that neural networking is necessary and *sufficient* and the only reason that symbol processing methods are somewhat successful is that they just approximate what is happening in the mind. To get better performance, connectionist methods are needed.

1.7.2 Picture Processing

MacLennan [102, 103][5] has argued that the important features of connectionist AI are the use of real numbers, that the large number of neurons in the brain and eye can be treated mathematically the same as fields (as in magnetic, electric, and gravitational) in physics (see [104]) and that image processing or picture processing is going on in the human mind.

For an example of this suppose we are watching the movie "Jack and Jill's Greatest Adventure," with that familiar story:

> Jack and Jill went up the hill to fetch a pail of water. Jack fell down and broke his crown and Jill came tumbling after him.

Just watching the movie gives you *images* that are stored away and it is rather difficult to argue that people store these images as some kind of symbolic tree-type representation. Then too, just reading the words will develop images in your mind. Moreover, as Jack starts to fall down you have to predict based on the images that Jack might suffer some damage that will require medical attention, so just working from the images you can do some reasoning. Why use symbols, structures of symbols, and formal rules to do this when picture-based processing will work?

The fact that people do store many memories as pictures and do at least some of their reasoning about the world using pictures ought to be one of the most obvious principles of all, yet it has been neglected, in part due to the predominance of symbol processing and in part due to the fact that processing pictures is hard compared to processing symbols. Unfortunately, at this point in time image processing is still fairly underdeveloped and has not been used in conjunction with representing the real world in programs where the goal of the program is to reason about the real world. Of course, simple image processing has been used by robots and in programs to recognize patterns such as handwritten or typed digits and letters of the alphabet.

1.7.3 Memories

The final key feature of the Physical Symbol System Hypothesis that can be criticized is the idea that people take in large amounts of experience from the real world, condense all these specific instances down to a handful of rules, and then people work from these rules to

[5] In addition to discussing how connectionist programs might store knowledge about the real world, MacLennan also reviews the symbol processing position in these papers so they are quite worthwhile and they are online.

solve new problems. An example of this is that when you drop something, call it, X, where X may be a rock, a piece of paper, or a feather. If you have done a lot of experimenting with dropping various things, then you will derive the rule: *if you are holding something and you let go, it will fall straight down*. In a symbol processing representation you are likely to code this as something like:

```
if letgo(X) then fall(X).
```

Yet there is a problem with such a rule because it only applies under certain conditions. If the air is moving and X is a feather or a flat piece of paper, then it will not fall straight down and it may remain in the air for quite some time before reaching the ground. But, if a piece of paper is crumpled up it will fall faster than if it is flat. If the air is moving very rapidly, even a rock will not fall straight down. Then what about the case we have all seen on TV where an astronaut on board a spaceship in orbit around the Earth lets go of something and rather than falling[6] it simply floats in midair? So a humanly coded set of rules is subject to the same problem that comes up in conventional computer programming where you must consider every possible permutation of the input data. Thus a rule-based program where the programmer has neglected to take into account wind velocity could conclude that if someone dropped a feather in a tornado the feather will fall straight to the ground, another example of the brittleness of conventional programs. So far, generating rules from data has not worked especially well either except for very small problem domains, domains much smaller than the real world domain.

One way to eliminate the problems involved with finding and using rules is to just not bother with the rules. If you have done your experiments of dropping various things under various conditions and seen the experiments done in space, then when someone asks you what will happen if you drop something all you have to do is reference your memories of your experiments to get the answer. This idea that people use simple memories to solve many ordinary real world problems is now getting a lot of attention although it is being done in the context of symbol-based methods, not in a picture-based context. These methods are called *case-based* and *memory-based*.

1.7.4 Quantum Mechanics

The possible application of quantum mechanics to thinking comes up in a number of ways. As already mentioned, Penrose in his two well-known popular books [148, 149] argues that consciousness is necessary for intelligence and quantum mechanics is responsible for consciousness, and moreover, that QM contains a nonalgorithmic component that cannot be duplicated by digital computers. Second, Vitiello [240] has proposed a quantum mechanical memory system that has the useful property that no matter how many memories this system has stored, one more can always be added without damaging any of the old memories, so in effect you get an unlimited memory. Finally, there is the idea that quantum mechanics might allow faster than light communication and this would explain the persistent reports of mind reading and predicting the future. For an argument in favor of this see [79]. Nanopoulos [133] has a quantum mechanical theory of brain function that

[6] The physics people will explain it by saying the object, the person, and the spaceship are really all falling at the same rate.

fits the psychological theories of William James and Sigmund Freud. Unfortunately, the application of quantum mechanics to thinking is still in a very early stage of development, it is more of a hope than any sort of concrete, testable proposal.

1.8 The Organization of the Book

The book starts with some of the lowest-level vision problems in Chapter 2 and then, generally speaking, the book goes on to cover higher and higher level problems until this progression ends with natural language processing in Chapter 10. The principles found at the beginning in vision systems can be found in slightly different forms all the way up to the highest levels. First, Chapters 2 and 3 illustrate the most important and useful pattern recognition and neural networking methods. Chapters 4 and 5 then give the approximate symbolic equivalents to the material in Chapters 2 and 3. The theme of Chapter 6 is that the methods presented so far are much too simple to produce programs with humanlike behavior. What is really needed is a much more complex architecture *and* a method for storing and retrieving knowledge in that architecture. Chapter 7 is to some extent an extension of the knowledge storage and retrieval problem in that storing, retrieving, and using cases rather than the traditional classical method of using rules is the theme. To a large extent Chapters 8, 9, and 10 are examples and applications of the principles given in Chapters 1 through 7, although, of course, Chapters 8 and 9 also develop the heuristic search theme as well.

One of the key ideas in this book is, of course, that the new methods need to be studied and worked on in order to get programs to achieve human levels of performance in dealing with problems, especially in dealing with the whole range of real world problems. However, all these methods, the symbolic and the neural and the memory-based all represent different ways of doing *pattern recognition*. Pattern recognition can be defined as the ability to classify patterns like the letters of the alphabet, other written symbols, or objects of various sorts; however, pattern recognition can also be used to try and find the more abstract and hidden patterns that exist within economic data or social behavior. Pattern recognition can also be used to describe the process of finding patterns that are close to each other in situations where the goal is not to do formal classification. Pattern recognition is also a formal academic field of study, typically found in electrical engineering or computer science departments, where the goal is once again to recognize patterns, either the visual ones or the more abstract ones.

1.9 Exercises

1.1. In the original Turing test [238], two people, a man and a woman go into the Turing test room while a third person asks them questions through a teletype system. Suppose the man is A and the woman is B but the third person knows them as X and Y. The problem for the third person is to try to determine whether X is male and Y is female or if X is female and Y is male. In the game, Y will try to help the questioner make the correct identifications but X will try to confuse the questioner. Then Turing says:

> We now ask the question, What will happen when a machine takes the part of A in this game? Will the interrogator decide wrongly as often when the game is played like this as he does when the game is played between a man and woman? These questions replace our original, "Can machines think?"

Would this version of the Turing test be any better at identifying thinking than the usual version where the game is simply to determine whether the entity in the Turing test room is a person or a computer?

Consider this too: one person posted a note in the Usenet comp.ai.philosophy newsgroup saying that Turing was noted for being a playful man and that maybe this whole Turing test was just a playful joke.[7]

You might actually want to try the human only version of this test in class. One way that is said to be very effective in determining who is male and who is female is to ask X and Y false questions such as "What is a Lipetz-head screwdriver?" Firschein[8] reports that in his experiments with the test: "Once this false question approach is discovered, few students can successfully fool the class."

1.2. The strong AI position is that certain types of computing are thinking. Suppose this is true. Does this mean that computers will be able to write great music, great poetry, create great art, and so on, or are these things something that only people can do?

1.3. Here are some examples of third and fourth grade arithmetic word problems:

> Matt has 5 cents. Karen has 6 cents. How many cents do they have altogether?
>
> Kathy is 29 years old. Her sister Karen is 25 years old. How much older is Kathy than Karen?
>
> If Mary sells 5 pencils at 6 cents each, how many cents will she have altogether?

It is not very hard to get a computer program to do a fair job of reading and solving these problems. These problems are quite simple to program because all children know at this grade level is how to add, subtract, multiply, and divide integers. They do not know about negative numbers or fractions. All they (or a program) have to do is pick off the numbers and then decide which operation to apply. Subtraction must always produce a positive number and division must always produce an integer. Also, the problems are loaded with phrases such as: "have altogether," "how much more," and "at this rate, how many."

Write a program that will look at the words in problems, much as the FORCE4 program did in Section 1.2 and then have it decide on what operation to apply to get the answer. Consult third and fourth grade textbooks for more problems.

Also consider the following alternative strategy. Instead of using individual words alone to choose the operation to apply, try using each *pair* of adjacent words in the problem. For example, "much more" will suggest subtraction and "many altogether" will suggest addition or maybe multiplication. Of course this will produce a longer list of items to store, but see if it produces better results.

[7] From Kenneth Colby, UCLA Computer Science Department, Message-ID: <3k4iub$p8n@oahu.cs.ucla.edu>, 14 Mar 1995 09:14:51 -0800.

[8] "Letters to the Editor," Oscar Firschein, *AI Magazine*, Fall 1992.

In fact, it is easy to create a fairly small program that can *learn* to do these problems. Give the program some sample problems and let it break the text down into pairs of words. For the first problem above, you get the pairs, "Matt has," "has 5," "5 cents," "cents," and so forth. Associate addition with each of these pairs. Expose your program to many such problems, and then test it with some of the problems you have trained it on, as well as on some unknown ones, and see how effective it is.

Whether you program this problem or not, you can still evaluate the effectiveness of the techniques that have been suggested as well as suggest more techniques that may work. Consider whether or not you could use these techniques or similar ones to do harder problems like:

> John went to the store and decided to buy 4 pieces of candy at 10 cents each. He gave the clerk 50 cents. How much change should he receive?

1.4. Rather than trying to classify arithmetic word problems you may want to classify Usenet news articles on two or more topics. It is probably best to choose articles from two very different newsgroups. After you train your program on the two classes, give the program some additional articles to see how well it classifies them.

1.5. For the network in Figure 1.5, show that when any two binary digits are given to the two input units the correct value (to within 0.1) of the exclusive-or of the two inputs appears on the output unit. Compute by hand and give the hidden unit values as well.

1.6. If we use a neural network where output units have real values for the threshold values and weights, show the networks corresponding to these two rules:

if a or b then c
if (a or b) and not c then d

1.7. Is the heuristic search suggested for crossing the river on rocks a realistic model of how people would find a path across the river if there was no fog? If it is not, how do people do it?

1.8. For some extra background on AI, read and summarize one or more of the following articles, all found in the Winter, 1988 issue of *Daedalus*:

"One AI or Many?" by Seymour Papert,
"Making a Mind vs. Modeling a Brain" by Stuart and Hubert Dreyfus,
"Natural and Artificial Intelligence" by Robert Sokolowski,
"Much Ado About Nothing" by Hilary Putnam,
"When Philosophers Encounter Artificial Intelligence" by Daniel C. Dennett.

1.9. Stuart and Hubert Dreyfus are two noted critics of artificial intelligence and they give their criticism in the book, *Mind Over Machine* [28]. Read this book and summarize their criticisms and then state whether or not you agree with them and why. (A good due date would be near the end of the course.)

Chapter 2

Pattern Recognition I

In this chapter we will be examining algorithms, especially neural networking algorithms, that can be used for pattern recognition. These algorithms will use the neural and associationist principles discussed in the first chapter. To illustrate the use of the principles we will start by looking at programs that can recognize letters of the alphabet, then look at a method for recognizing words, and finally show how the same principles are involved in higher-level thought as well. One of the results of this study will be that recognizing even simple patterns like letters requires more knowledge about the world than you might at first suspect is necessary. The problems encountered with recognizing letters spill over to higher-level cognitive activities such as understanding natural language and pose difficulties for all AI programs.

2.1 A Simple Pattern Recognition Algorithm

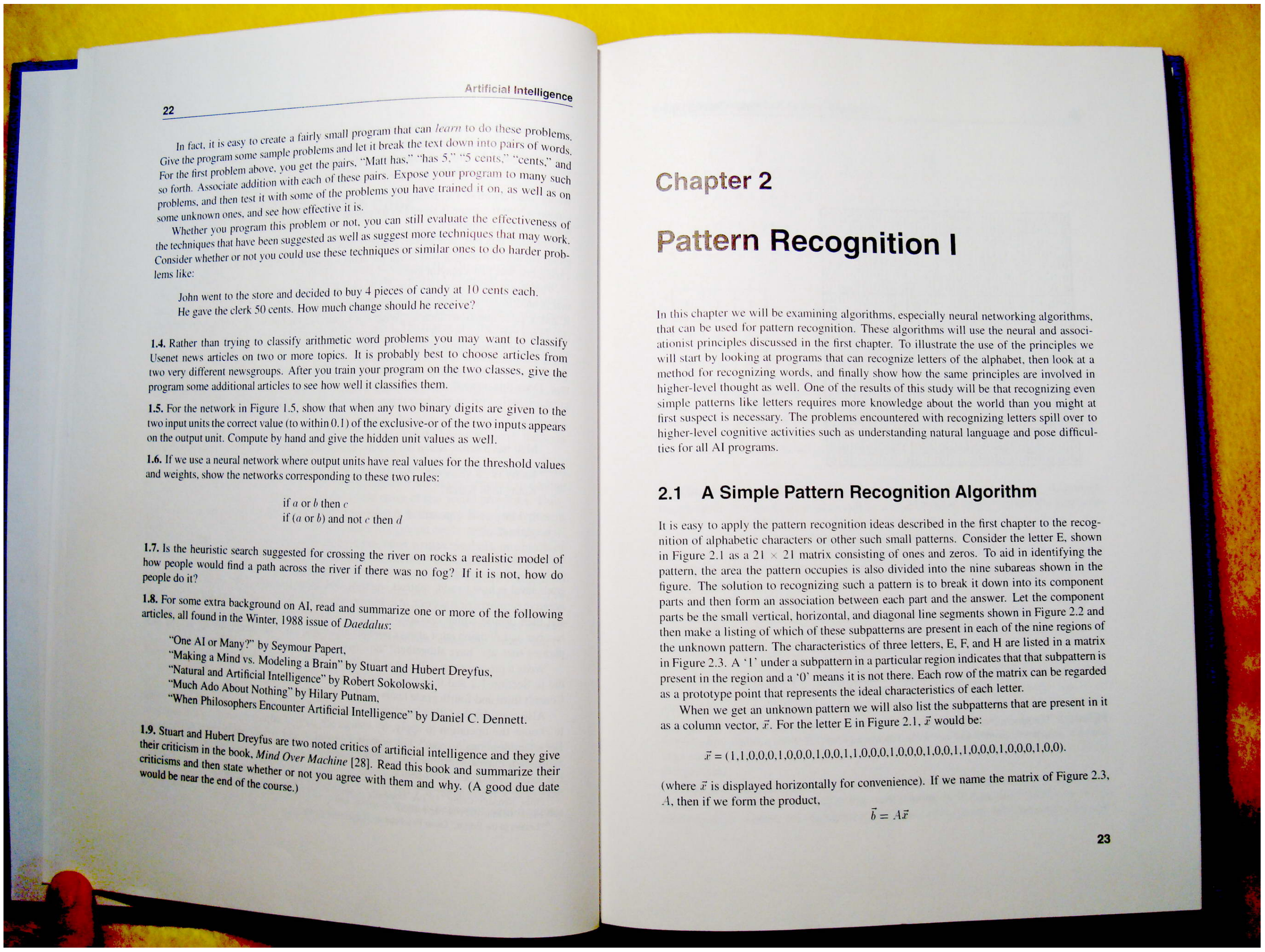
It is easy to apply the pattern recognition ideas described in the first chapter to the recognition of alphabetic characters or other such small patterns. Consider the letter E, shown in Figure 2.1 as a 21 × 21 matrix consisting of ones and zeros. To aid in identifying the pattern, the area the pattern occupies is also divided into the nine subareas shown in the figure. The solution to recognizing such a pattern is to break it down into its component parts and then form an association between each part and the answer. Let the component parts be the small vertical, horizontal, and diagonal line segments shown in Figure 2.2 and then make a listing of which of these subpatterns are present in each of the nine regions of the unknown pattern. The characteristics of three letters, E, F, and H are listed in a matrix in Figure 2.3. A '1' under a subpattern in a particular region indicates that that subpattern is present in the region and a '0' means it is not there. Each row of the matrix can be regarded as a prototype point that represents the ideal characteristics of each letter.

When we get an unknown pattern we will also list the subpatterns that are present in it as a column vector, $\vec{x}$. For the letter E in Figure 2.1, $\vec{x}$ would be:

$$\vec{x} = (1,1,0,0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,0,0,1,0,0).$$

(where $\vec{x}$ is displayed horizontally for convenience). If we name the matrix of Figure 2.3, A, then if we form the product,

$$\vec{b} = A\vec{x}$$



In fact, it is easy to create a fairly small program that can *learn* to do these problems. Give the program some sample problems and let it break the text down into pairs of words. For the first problem above, you get the pairs, "Matt has," "has 5," "5 cents," "cents," and so forth. Associate addition with each of these pairs. Expose your program to many such problems, and then test it with some of the problems you have trained it on, as well as on some unknown ones, and see how effective it is.

Whether you program this problem or not, you can still evaluate the effectiveness of the techniques that have been suggested as well as suggest more techniques that may work. Consider whether or not you could use these techniques or similar ones to do harder problems like:

> John went to the store and decided to buy 4 pieces of candy at 10 cents each.
> He gave the clerk 50 cents. How much change should he receive?

1.4. Rather than trying to classify arithmetic word problems you may want to classify Usenet news articles on two or more topics. It is probably best to choose articles from two very different newsgroups. After you train your program on the two classes, give the program some additional articles to see how well it classifies them.

1.5. For the network in Figure 1.5, show that when any two binary digits are given to the two input units the correct value (to within 0.1) of the exclusive-or of the two inputs appears on the output unit. Compute by hand and give the hidden unit values as well.

1.6. If we use a neural network where output units have real values for the threshold values and weights, show the networks corresponding to these two rules:

if a or b then c
if (a or b) and not c then d

1.7. Is the heuristic search suggested for crossing the river on rocks a realistic model of how people would find a path across the river if there was no fog? If it is not, how do people do it?

1.8. For some extra background on AI, read and summarize one or more of the following articles, all found in the Winter, 1988 issue of *Daedalus*:

"One AI or Many?" by Seymour Papert,
"Making a Mind vs. Modeling a Brain" by Stuart and Hubert Dreyfus,
"Natural and Artificial Intelligence" by Robert Sokolowski,
"Much Ado About Nothing" by Hilary Putnam,
"When Philosophers Encounter Artificial Intelligence" by Daniel C. Dennett.

1.9. Stuart and Hubert Dreyfus are two noted critics of artificial intelligence and they give their criticism in the book, *Mind Over Machine* [28]. Read this book and summarize their criticisms and then state whether or not you agree with them and why. (A good due date would be near the end of the course.)

Chapter 2

Pattern Recognition I

In this chapter we will be examining algorithms, especially neural networking algorithms, that can be used for pattern recognition. These algorithms will use the neural and associationist principles discussed in the first chapter. To illustrate the use of the principles we will start by looking at programs that can recognize letters of the alphabet, then look at a method for recognizing words, and finally show how the same principles are involved in higher-level thought as well. One of the results of this study will be that recognizing even simple patterns like letters requires more knowledge about the world than you might at first suspect is necessary. The problems encountered with recognizing letters spill over to higher-level cognitive activities such as understanding natural language and pose difficulties for all AI programs.

2.1 A Simple Pattern Recognition Algorithm

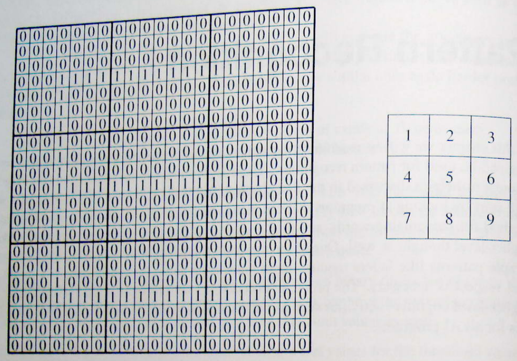
It is easy to apply the pattern recognition ideas described in the first chapter to the recognition of alphabetic characters or other such small patterns. Consider the letter E, shown in Figure 2.1 as a 21×21 matrix consisting of ones and zeros. To aid in identifying the pattern, the area the pattern occupies is also divided into the nine subareas shown in the figure. The solution to recognizing such a pattern is to break it down into its component parts and then form an association between each part and the answer. Let the component parts be the small vertical, horizontal, and diagonal line segments shown in Figure 2.2 and then make a listing of which of these subpatterns are present in each of the nine regions of the unknown pattern. The characteristics of three letters, E, F, and H are listed in a matrix in Figure 2.3. A '1' under a subpattern in a particular region indicates that that subpattern is present in the region and a '0' means it is not there. Each row of the matrix can be regarded as a prototype point that represents the ideal characteristics of each letter.

When we get an unknown pattern we will also list the subpatterns that are present in it as a column vector, $\vec{x}$. For the letter E in Figure 2.1, $\vec{x}$ would be:

$$\vec{x} = (1,1,0,0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,0,0,1,0,0).$$

(where $\vec{x}$ is displayed horizontally for convenience). If we name the matrix of Figure 2.3, A, then if we form the product,

$$\vec{b} = A\vec{x}$$



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1	2	3
4	5	6
7	8	9

Figure 2.1: The letter E as a matrix of zeros and ones. The area is divided into nine subareas, numbered 1 through 9 as shown on the right.

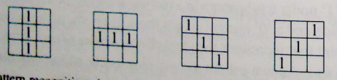


Figure 2.2: The pattern recognition algorithm will start by looking for each of these vertical, horizontal, or diagonal subpatterns and then will list their presence (1) or absence (0) in a vector.

```
         1          2          3          4          5          6          7          8          9
     | – \ /    | – \ /    | – \ /    | – \ /    | – \ /    | – \ /    | – \ /    | – \ /    | – \ /

E:   1 1 0 0    0 1 0 0    0 1 0 0    1 1 0 0    0 1 0 0    0 1 0 0    1 1 0 0    0 1 0 0    0 1 0 0

F:   1 1 0 0    0 1 0 0    0 1 0 0    1 1 0 0    0 1 0 0    0 1 0 0    1 0 0 0    0 0 0 0    0 0 0 0

H:   1 0 0 0    0 0 0 0    1 0 0 0    1 1 0 0    0 1 0 0    1 1 0 0    1 0 0 0    0 0 0 0    1 0 0 0
```

letter:	E	F	H
pattern is E:	12	9	6
pattern is F:	9	9	6
pattern is H:	6	6	9

Figure 2.3: The above matrix is derived simply by listing the presence (1) or absence (0) of each feature (vertical line segment, horizontal line segment, and two diagonal line segments) in each of the nine regions of a pattern. When you multiply this matrix, A, times the vector $\vec{x}$ that lists the features in the unknown picture, you get a vector that lists the number of votes for the letters, E, F, and H. This is basically the algorithm of Walker and Amsler in the last chapter. This works fairly well, but it gives the same score (9) for an E and an F when an F is presented to the algorithm.

by matrix multiplication, the ith row of the vector, $\vec{b}$, will give the number of "votes" for the ith letter. For the letter E there will be a total of twelve votes, the letter F will have nine votes and H will have six, so the answer must be that the unknown is the letter E. The scores you get from this algorithm when you submit an ideal letter E, an ideal letter F, and an ideal letter H are also shown in the figure. This algorithm is really the same as the one proposed by Walker and Amsler to categorize newspaper stories except instead of using words to contribute votes, it is the line segments in each region that contribute votes.

This algorithm is a simple one and you may already have noticed a problem with it. If the unknown letter happened to be the letter F, doing the matrix multiplication produces a score of 9 for E and 9 for F. A solution to this problem is to change the weights in the matrix and there are simple algorithms that will find a matrix that will work correctly, however, for now there are two simple changes that can be made to correct the problem. First change the values of the weights so that the perfect letter E will score 1.0, the perfect F will be 1.0, and the perfect H will be 1.0. To do this, divide each element of the first row by 12 and each element in the F and H rows by 9. Second, when an unknown pattern has features that the ideal pattern should not have, the score for the letter should be decreased by some amount. For the first row of the matrix change each 0 to –1/12 and for the second and third rows change each 0 to –1/9. This gives the matrix shown in Figure 2.4 and it corrects the E-F problem.

Note that with this algorithm even if the unknown is distorted a little from its ideal pattern, so that one or two subpatterns were missing or if there were one or two extra characteristics present, the algorithm is still likely to come up with the correct answer.

$$\begin{array}{cccccccccccccccccccccccccccccccccccc}
 & 1 & & & & 2 & & & & 3 & & & & 4 & & & & 5 & & & & 6 & & & & 7 & & & & 8 & & & & 9 & & \\
\vert & - & \backslash & / & \vert & - & \backslash & / & \vert & - & \backslash & / & \vert & - & \backslash & / & \vert & - & \backslash & / & \vert & - & \backslash & / & \vert & - & \backslash & / & \vert & - & \backslash & / & \vert & - & \backslash & / \\
\frac{1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{-1}{12} & \frac{1}{12} & \frac{-1}{12} & \frac{-1}{12} \\
\frac{1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} \\
\frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{1}{9} & \frac{-1}{9} & \frac{-1}{9} & \frac{-1}{9}
\end{array}$$

letter:	E	F	H
pattern is E:	1.000	0.667	0.000
pattern is F:	0.750	1.000	0.333
pattern is H:	0.250	0.333	1.000

Figure 2.4: The above matrix works better than the previous one.

Notice too, that the pattern in Figure 2.1 can be moved around some within the 21×21 matrix and the algorithm still gives the same set of 36 values and therefore will get the same answer.

This algorithm can also be presented as a neural algorithm. A diagram of the algorithm as a neural network is shown in Figure 2.5. This network has 36 nodes in the bottom layer, one for each of the 36 pattern features, and 3 units for the possible answers, E, F, and H in the output layer. The weights come from the values found in the matrix A, in Figure 2.4. The output values are computed in the standard neural way, where the value of each output unit, o_k is computed as follows:

$$o_k = \sum_{j=1,36} w_{j,k} i_j$$

and where the i_j are the 36 input values and $w_{j,k}$ is the weight on the connection between input unit j and output unit k. The portion of the algorithm where you first have to search for the presence or absence of a feature in the unknown is still not 'neural,' however, in the next section we will show how this too can be organized as a neural networking algorithm. The change will be that more layers can be added below the input layer in Figure 2.5 to find the values for the 36 input units.

2.2 A Short Description of the Neocognitron

The program in the last section was a simple illustration of how a program can recognize patterns, however, there have been many more sophisticated programs designed to recognize letters and other symbols. One recent important set of experiments has been done by Fukushima. His first program was the Cognitron [41]. This was followed by later programs, all called the Neocognitron [42, 43, 44, 45, 46, 47]. Each version of the Neocognitron varies slightly in its details. In this section we will mention some of the key features of the Neocognitron without studying its weight adjusting algorithm.

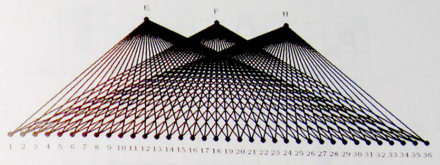


Figure 2.5: A neural interpretation of the algorithm. The bottom layer contains nodes that represent the features found in the unknown letter. Each node will then be 1 or 0 depending on whether the feature was present or not. The top layer contains three nodes for the three letters to be recognized. Whichever one lights up brightest is the answer. The lines connecting nodes in the two layers contain the weights shown in Figure 2.4.

2.2.1 Detecting Short Lines

In the last section we assumed that the short line segments were found by a very conventional algorithm. Finding short line segments in a figure is actually quite easy to do neurally but it requires a large number of neurons and interconnections. First, suppose the characters we have to identify are again E, F, and H, and again they are contained in a 21×21 matrix of zeros and ones as in Figure 2.6. The network we will use will have an input layer, two layers called the S and C layers (following Neocognitron terminology), and an output layer that gives the identity of the pattern. This network is shown in Figure 2.7 and it is much smaller than a typical Neocognitron network. The typical Neocognitron network uses many more layers and we will look at a larger network later, but for now the network in Figure 2.7 will suffice.

In this network we will have it look for the four short line segments shown in Figure 2.8. To do this there will be four sets of S layer neurons, one set dedicated to finding each of the four types of line segments. Each set is represented by a square matrix of neurons as shown in Figure 2.7. Each neuron in each of the four sets connects to the 9 neurons in a 3×3 area of the input matrix just beneath it. Figure 2.9 shows the area of the input matrix that one S level neuron connects to. In this figure the set of S level neurons is looking for the small horizontal line pattern, as a set of three black squares in the middle line of a 3×3 matrix. In the Neocognitron the interconnection weights are trained to turn on when they see these different line segments, but here let us suppose that they are handwired and an S layer cell turns on when it finds the correct pattern below it.

When a picture is given to the network, all the S layer cells look for their respective patterns at every possible location. Now the C layer cells look at the cells on the S layer. Suppose, as in the algorithm in Section 2.1, that we want to flag the existence of each type of line segment in each of the nine areas of the input matrix so that we end up with the vector of 36 values we used in that algorithm. Let each of the C cells connect to a 7×7 area of the S level matrices and have a C cell turn on if it receives any activation from one or more of the 49 S cells it connects to. For instance, in Figure 2.9 the C cell at the bottom and center of the picture turns on because at least one of its S cells is on. Notice then, that the whole pattern on the input matrix could be shifted several rows or columns and

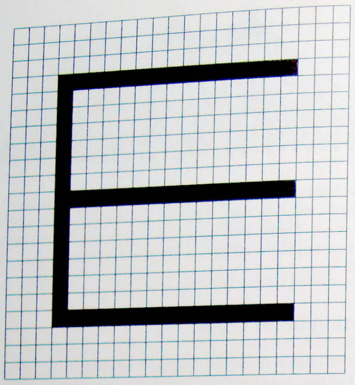


Figure 2.6: The letter E where black squares are a 1 and white squares are a 0.

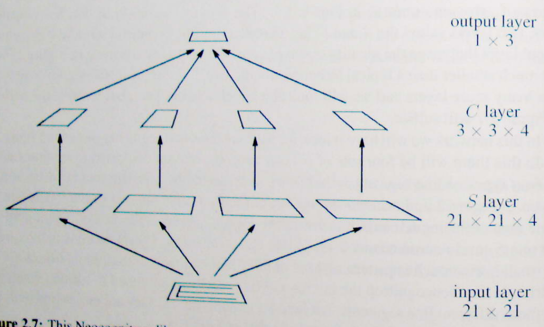


Figure 2.7: This Neocognitron-like network has four layers: the input layer to hold the pattern to be identified, an S layer to scan for the four subpatterns, a C layer to group the subpatterns into nine regions, and finally the output layer. There are too many neurons to display individually and each rectangle represents a matrix consisting of a large number of neurons. Also, there are many more interconnections between neurons than can be drawn here so whole groups of these connections will be represented by arrows. The input layer has 21×21 neurons. The S layer contains four sets of 21×21 neurons each, and the C layer contains four sets of 3×3 neurons each. Each of the four sets in these layers is used for each of the four subpatterns that must be recognized. The C layer will contain the 36 values that would be on the input layer of Figure 2.5, however the values will be in a different order. The output layer will contain three neurons, one each for E, F, and H.

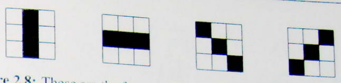


Figure 2.8: These are the four patterns the network will be looking for.

the same C cell will still turn on. In the typical Neocognitron this tolerance for a shifted pattern is taken into account over many layers rather than just two.

2.2.2 A Typical Neocognitron

Figures 2.10 and 2.11 give the outlines of one particular Neocognitron [43] designed to recognize the digits 0 through 9. It has four pairs of S and C layers and an input layer for a total of nine layers. The S_1 layer looks for the twelve different subpatterns shown in Figure 2.12 that represent the patterns you get from horizontal, vertical, and diagonal lines and from the four lines in Figure 2.13. Notice that these latter four lines shown in Figure 2.13 give rise to eight different patterns that the network needs to look for. Since the twelve patterns of Figure 2.12 only represent lines at eight possible angles, the C_1 layer only has to have eight sets of neurons, not twelve. Each of the four pairs of patterns on the right in Figure 2.12 will map into one C_2-layer cell. Each of the eight sets of C_1-layer neurons is 11×11 and these cells turn on when any of the S_1-layer cells they connect to are on. This arrangement is used to take into account patterns that are slightly translated within the input matrix. Figure 2.11 shows a vertical slice of the network and how the units in one layer connect to the previous layer. Most of the C_1 neurons connect to a 5×5 area of the S_1 layer while the C_1 neurons near the edge connect to fewer S_1 neurons.

At the S_2 and C_2 layers the network looks for more complex patterns that can be formed from the short line segments detected by the S_1 layer. Figure 2.14 shows some of the 38 features that this pair of layers looks for. Again, some of the 38 S_2 features that are found can be mapped into just 22 categories just as the 12 subpatterns of layer S_1 can be mapped into just eight categories. Again, to take into account the possible translations of input figures, the 11×11 array in the S_2 layer is mapped into just a 7×7 array in the C_2 layer.

At the S_3 and C_3 layers still more complex features are detected like those in Figure 2.15.

2.2.3 Training the Neocognitron

The Neocognitron can be trained in either of two ways. First, it can be *trained with a teacher* and in this mode the human trainer selects specific patterns that each layer of neurons needs to learn, for instance, 12 at the S_1-C_1 layer, 38 at the S_2-C_2 layer, and so on. In training layer S_1, one of the 12 patterns in Figure 2.12 is placed on the input layer, the trainer selects a cell, called the seed cell, in one of the arrays and trains it to respond to the pattern on the input layer, then all the cells in the array get the same set of weights. The training is done by adjusting the weights in the network but we will not go into that aspect of it. Next, another of the 12 patterns is selected and the weights are adjusted and so on. The same procedure is done for layer S_2 only now the patterns in Figure 2.14 are given to the input layer. The training continues this way for layers S_3 and S_4. In *training without a*

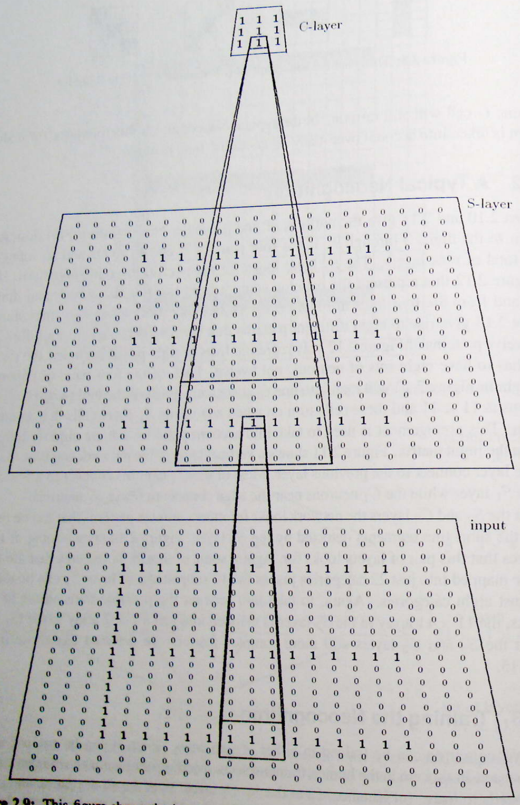


Figure 2.9: This figure shows the input matrix on the bottom that contains the letter E. The next layer up is one of the four S-level sets of neurons, this set is designed to detect the short horizontal line. Each neuron here looks at a 3×3 area of the input matrix. The next layer up is the C layer that contains a 3×3 matrix of neurons that will list which of the nine portions of the input pattern contains a horizontal line. Each of these nine neurons is connected to a 7×7 area in the S-level matrix. A C-layer neuron turns on if it finds at least one occurrence of a 1 within the area it scans. In this case all nine neurons will be 1.

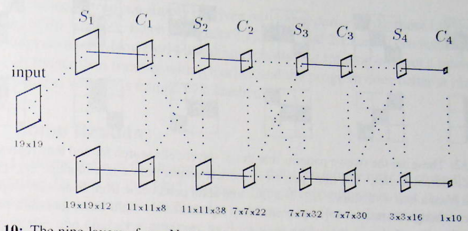


Figure 2.10: The nine layers of one Neocognitron network [43] designed to recognize the digits 0 through 9. The input layer is 19×19. Layer S_1 looks for 12 features, but these 12 map into just 8 in layer C_1. Layer S_2 looks for 38 features, but these map into just 22 in layer C_2. Layer S_3 looks for 32 features that map into 30 in the C_3 layer. Finally, layer S_4 looks for 16 features and the answer appears on layer C_4. S-layer neurons must scan all the C-level arrays of neurons while the C-level sets look at only one S-array of neurons.

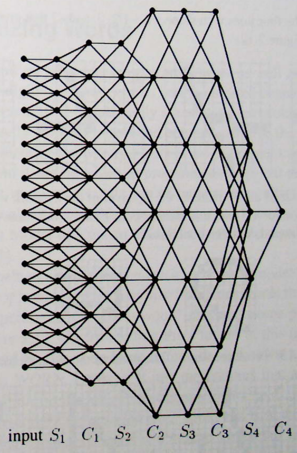


Figure 2.11: This slice through the network shows how cells are interconnected. The figure shows that most of the S_1 cells vertically scan three of the input cells except near the edge of the input where only two cells are scanned. Most of the C_1 cells vertically scan five of the S_1 cells, most of the S_2 cells vertically scan three of the C_1 cells, and so on.

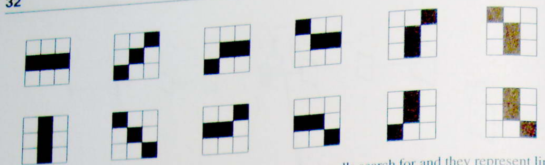


Figure 2.12: These are the twelve patterns that the S_1-layer cells search for and they represent lines with eight different angles. The third pair from the left comes from a line of slope 1/2, the fourth pair comes from a line with slope −1/2. The fifth and sixth pairs come from lines with slopes 2 and −2. So, even though there are twelve patterns, they represent only eight lines, therefore only eight C_1-layer cells are needed.

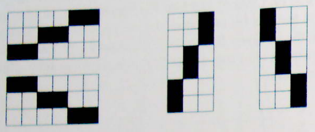


Figure 2.13: These are the four lines with slope 1/2, −1/2, 2, and −2 that give rise to the four pairs of patterns on the right of Figure 2.12.

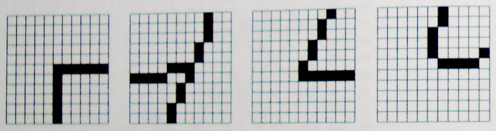


Figure 2.14: These are four of the 38 patterns the S_2 layer of the network will be looking for. Most of the 38 features consist of curved lines and intersections of lines, but some of them (not shown) are simply plain vertical, horizontal, and diagonal lines.

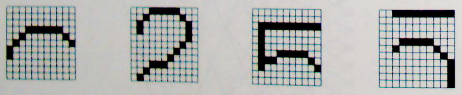


Figure 2.15: These are some of the patterns the S_3 layer looks for.

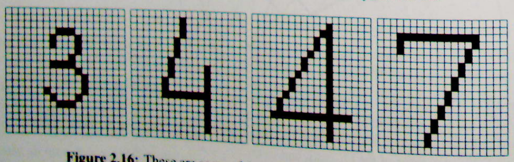


Figure 2.16: These are some of the patterns the S_4 layer looks for.

teacher the Neocognitron can simply be given *only input patterns* like those in Figure 2.16 and *not any of the features shown* in Figures 2.12, 2.14, and 2.15 and it can actually learn to classify them correctly. However, Fukushima reports that training without a teacher takes much much longer than training with a teacher and the program does better at classifying distorted patterns when it is trained by a teacher.

2.2.4 Some Results

Figure 2.17 shows some of the distorted digits that the Neocognitron can correctly classify. An interesting feature in the 1987 version of the Neocognitron is that it can look at a picture containing more than one character and first identify one character and then another. In this scheme one character emerges as dominant and all the neural pathways involved in identifying this character become active and inhibit the pathways needed to recognize other characters. At the flip of a switch these active pathways can be inhibited so that some other pathways will find another character in the input. Figure 2.18 shows a time series of the Neocognitron doing this. While the Neocognitron was developed to do visual pattern recognition, Fukushima expects that it could easily be modified to do speech recognition as well.

2.3 Recognizing Words

Various pattern recognition programs can perform almost as well as people at recognizing individual letters and digits. Among people, badly distorted characters all by themselves can easily be misinterpreted, but people rarely encounter such problems in practice because the characters are usually seen in context and context can give the extra amount of information necessary to identify the pattern. This is an important capability of human data processing systems and it can be added to programs as well, but adding this capability to programs reveals that to do an accurate job of pattern recognition requires that the program know quite a lot about the world. The particular work we are going to look at is "An Interactive Activation Model of Context Effects in Letter Perception," by Rumelhart and McClelland [180, 181].

The work by Rumelhart and McClelland was an effort to model certain results obtained by experiments on people who had to recognize four letter words that are briefly flashed on a screen. In these experiments the words would also have some parts of letters obscured or missing as in Figure 2.19. As for the first three letters in this figure, there is no doubt that they are W, O, and R. Based on these letters, the unknown word may be WORD or WORK or WORM or WORN. Looking at just the features that are visible in the fourth letter we see a vertical line on the left. That suggests the letters E, F, K, R, and others. The diagonal line in the lower right suggests the letter could be a K or an R. The short horizontal line is consistent with the letters K and R and others. Given our knowledge about words, we can safely conclude that the incomplete letter is a K and that the unknown word is the word "WORK." The individual features suggest letters, the letters suggest words, the words suggest possible letters, and the letters even suggest individual features. This phenomenon shows why when people write text (especially computer programs!) it is very common for them to miss many mistakes in the text just because they know what the text is supposed to

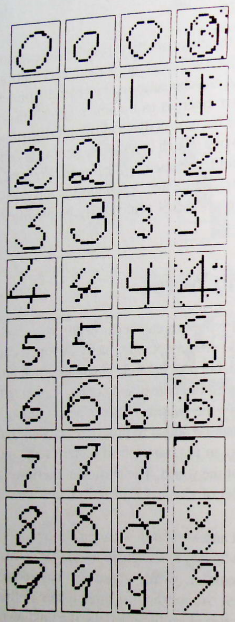


Figure 2.17: These are some of the patterns the Neocognitron has been able to recognize. A 1991 version of the program has been able to recognize 35 characters.

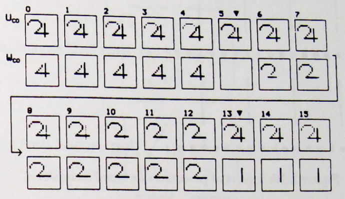


Figure 2.18: A pattern containing 1, 2, and 4. The top row shows the activation state of the input cells. The row below shows another layer of cells that isolates the parts of the input that correspond to particular figures. First, the network "sees" the 4. At time = 5, a switch is flipped that resets the network. The "4" pattern is inhibited and now the network sees the 2. At time = 13, the switch is flipped again and now the network sees the 1.

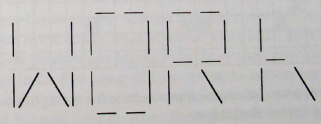


Figure 2.19: An input to the program, an incomplete version of a word. In reality the program does not "look at" a picture, instead the program gets a vector that flags which line segments are present in each position.

say and they know what the words are supposed to be. To really catch mistakes you need some other person or program to proofread the text.

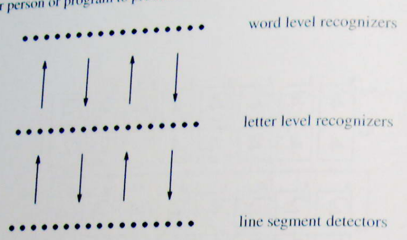


Figure 2.20: Layers of processors for word recognition.

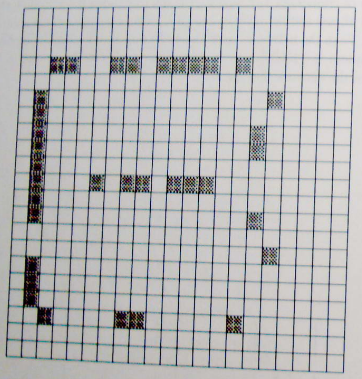


Figure 2.21: Even with quite a lot missing from this letter B it is easy to interpret a small number of dots as forming curved lines or straight lines.

To model the word recognition process we will have three layers of neurons arranged as shown in Figure 2.20. In this arrangement, the detection of certain line segments influences the letters that are recognized and the letters that are recognized influence the words that are recognized. When processing proceeds this way from the primitive data to conclusions it is referred to as *bottom-up* or *data driven* processing. Most pattern recognition has traditionally been bottom-up in character. On the other hand, if a person expects a particular word, the lighting up of the word can influence what letters are seen. Furthermore, any bias we

may have toward a particular letter will also influence whether or not we see a collection of dots as forming a curved or straight line as in Figure 2.21. Processing that proceeds from expectations is referred to as *expectation driven* or *top-down* processing. When information flows both from the bottom up and from the top down it is called *interactive* processing.

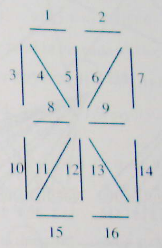


Figure 2.22: The 16 features used to construct the letters.

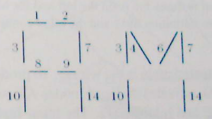


Figure 2.23: The word AM, with its features numbered.

To make this concrete we will construct a network designed to recognize just two words, AM and AN. Each letter will be constructed from a set of 16 short line segments. These segments are shown in Figure 2.22. Figure 2.23 shows how the word AM is constructed using these line segments. Figure 2.24 shows the neural network designed to recognize the two words. The top layer has the nodes, AM and AN, and either AM or AN lights up when their features are present in the middle layer of nodes. The middle layer has nodes A1, M2, and N2 for A in the first position and M and N in the second position. A1 will light up when there are features in the first group of 16 units on the bottom layer that suggest that an A is present. M2 will light up when there are features for an M present in the second group of 16 units and N2 will light up when there are features for an N present in the second group. In Figure 2.24 the connections between nodes that end in a dot are used to indicate an inhibition link (negative weight) and all these connections are bidirectional. The connections without a dot are activation links and they are also bidirectional. For instance, the presence of an M in the second position activates AM and inhibits AN. Notice that the network will have AM and AN inhibit each other. This was done so that when one of these words begins to be recognized the network will inhibit the other. This design makes it possible for just one word to clearly win and this type of network is an example of a *winner-take-all* network. In a winner-take-all network the output nodes compete with each

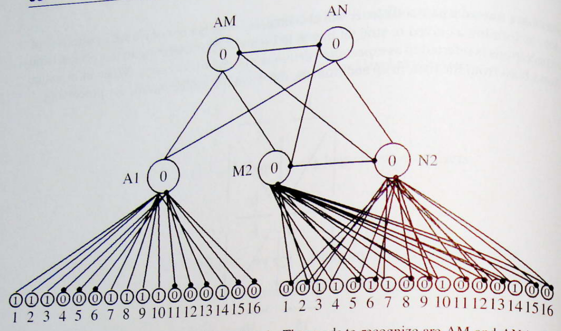


Figure 2.24: A simple word recognition network. The words to recognize are AM and AN in the top layer. The letters to recognize are A in position 1 (A1), M in position 2 (M2) and N in position 2 (N2) in the middle layer. At the start of the process the middle and upper layer nodes will be 0. The lower layer has the 16 numbered features for A on the left and for M on the right. The connections between nodes that end in a ● are inhibition links and the other connections are activation links.

other and the most highly activated node is the winner. To produce top-down activation and inhibition, the word node AM activates the letter nodes A1 and M2 and inhibits N2. The node AN activates A1 and N2 and inhibits M2. The input-feature-level nodes activate the letters that they are part of and inhibit the letters they are not part of. These connections also work in reverse, so as letters turn on they activate or inhibit the input level features. The activation weights will all have a value of +0.02 and the inhibition links will all have weights of –0.045.

In the previous networks we have looked at we took the input and used it to activate higher layers in the network until the activation reached the output layer. Now the activation procedure will be more complicated in several ways. The input features on the lower level will all have initial values of either 0, meaning the feature is not present, or 1, meaning that the feature is present. The other nodes in the network will start out at 0. The value of a node such as M2 will be calculated using a formula that will only increase or decrease its activation *by a small amount*. At the same time that node N2 is calculating its activation value all the other nodes will be doing the same thing. The net result is that all values will go up or down by a small amount (or sometimes they may be unchanged). We then repeat this process over and over again until each node reaches a stable value. The formula we use to light up nodes will keep the values of the nodes between 0 and 1.

Each node in the network is updated using the following algorithm. First, compute the net input to each node, j, at time, t by in the usual way:

$$net_j(t) = \sum_i w_{i,j} a_i(t). \tag{2.1}$$

where $w_{i,j}$ is the value of the connection from node i to node j and $a_i(t)$ is the activation level of node i at time t. To compute the activation value of node j, first, if $net_j(t)$ is greater than 1.0 then set $net_j(t)$ to 1.0 and if it is less than −1.0 set it to −1.0. If the input is positive then the new value of the node at time $t + \Delta t$ will be:

$$a_j(t + \Delta t) = a_j(t) + net_j(t)(1.0 - a_j(t)), \tag{2.2}$$

while if net_j is negative the new value of the node will be:

$$a_j(t + \Delta t) = a_j(t) + net_j(t)a_j(t). \tag{2.3}$$

Notice, how, if the value of a node is close to 1.0, say 0.9, and if the input is large, say 0.9, the new value of the node will not go over 1.0:

$$a_j(t + \Delta t) = 0.9 + 0.9(1.0 - 0.9) = 0.99.$$

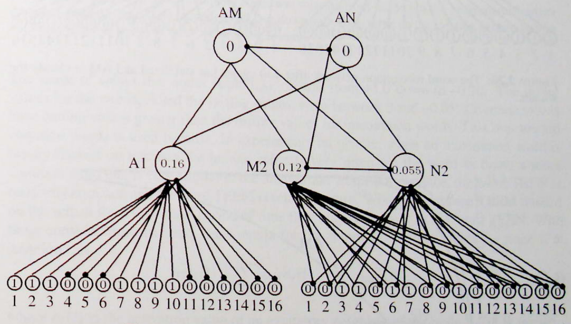


Figure 2.25: The word recognition network after one cycle.

For an example of the process we will give the network the word AM and show how the network lights up. Figure 2.24 shows the values of the nodes for this example at the start of the process and Figure 2.25 shows the values after the first iteration. After 130 cycles the values have stabilized at those shown in Figure 2.26 with A1, M2, and AM clearly the winners. The activation of some nodes as a function of time is shown in Figure 2.27. In another experiment with the network we can turn off the input feature number 3 in the letter M. Figure 2.28 shows that top-down activation from the M node turns it on anyway. In effect, the network is trying to "see" the feature.

This example embodies the most important aspects of the Rumelhart and McClelland algorithm and we can now look at the somewhat more complicated algorithm they used.

In the experiments done by Rumelhart and McClelland they used 1,179 four-letter words. At the letter level there are 26 letter nodes for each of the four positions in a word.

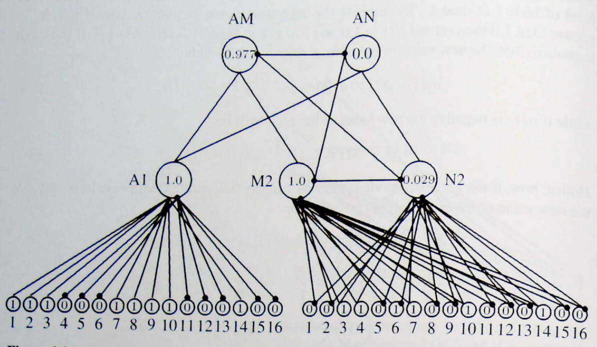


Figure 2.26: The word recognition network after 130 cycles has stabilized and AM is clearly the winner.

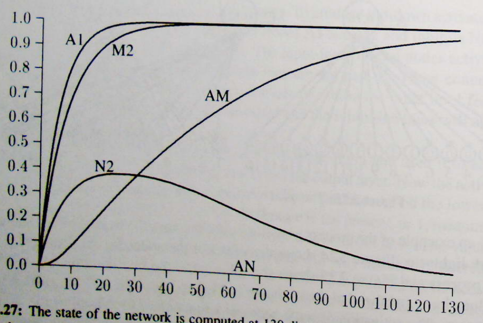


Figure 2.27: The state of the network is computed at 130 discrete time intervals and smooth lines are drawn between the points. A1 rises the fastest because it is supported by eight low-level features and the word AM. M2 rises almost as quickly but it is inhibited slightly by N2 and it is only being supported by six low-level features. N2 initially rises but its low-level support is less than M2 and it is inhibited by M2 and AM so after a while it starts to decline. The word AM slowly lights up during the process but AN never gets enough activation to rise above 0.

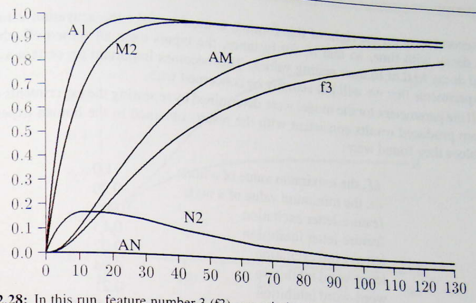


Figure 2.28: In this run, feature number 3 (f3) was missing from the M but the program recognizes the word AM anyway. As M is recognized, top-down activation turns on the missing feature so you could say that it is trying to "see" the missing feature.

The range of values that nodes could take on was from +1.0 down to –0.20. The initial values for the words, called the resting values, were between 0 and –0.05. Common words have resting values greater than the resting values for uncommon words. This bias toward common words is used because, in experiments on people, when an uncommon word is briefly flashed on a screen the human subjects very often see the word as being a more common word that is very similar to the uncommon word. For instance, the word TEEN is relatively common and the word TEEM is relatively rare. When subjects see TEEM flashed on the screen for a very short period of time they may well see it as the word TEEN. With these considerations in mind, the formula for the summation of the inputs for node i, at iteration t, is:

$$net_i(t) = \sum_j \alpha_{ij} e_j(t) - \sum_k \gamma_{ik} i_k(t), \qquad (2.4)$$

where $e_j(t)$ is the activation value of an excitatory neighbor of the node, $i_k(t)$ is the activation value of an inhibitory neighbor of the node, and α_{ij} and γ_{ik} are the weights. When $net_j(t)$ is positive, we calculate a quantity, $\eta_j(t)$, by:

$$\eta_j(t) = net_j(t)(M - a_j(t)), \qquad (2.5)$$

where M is the maximum activation level of the unit. As we said earlier, this value is simply 1.0. When $net_j(t)$ is negative, we calculate $\eta_j(t)$ by:

$$\eta_j(t) = n_j(t)(a_j(t) - m), \qquad (2.6)$$

where m is the minimum activation of the unit. The value for m in their experiments was –0.20. The new value of the activation of node j, at time $t + \Delta t$ is given by:

$$a_j(t + \Delta t) = a_j(t) - \Theta_j(a_j(t) - r_j) + \eta_j(t). \qquad (2.7)$$

The second term on the right of this formula is used to make the activation value of the node decay with time, so that if, for instance, the inputs were all turned off, the nodes would decay to 0 or to their resting values. This becomes important for certain aspects of the experiments that we will not actually be concerned with.

All the parameters for the model were determined by repeating the experiments until the program produced results consistent with the results obtained in the human experiments. The values they found were:

M, the maximum value of a node	1.0
m, the minimum value of a node	–0.20
feature-letter excitation	0.005
feature-letter inhibition	0.15
letter-word excitation	0.07
letter-word inhibition	0.04
word-word inhibition	0.21
letter-letter inhibition	0
word-letter excitation	0.30
word-letter inhibition	0
Θ_j, the decay rate	0.07
r_j, the resting value for letter nodes	0
r_j, the resting value for word nodes	variable

Since the weights between words and letters, features and letters, and so forth are all the same, regardless of the particular word, letter, or feature, the weights for every individual connection do not need to be stored and this saves a lot of memory space. Rumelhart and McClelland's experiments also included another part where the probability that the computer gives a response to a given input is estimated but it is not necessary to describe that portion of their work here.

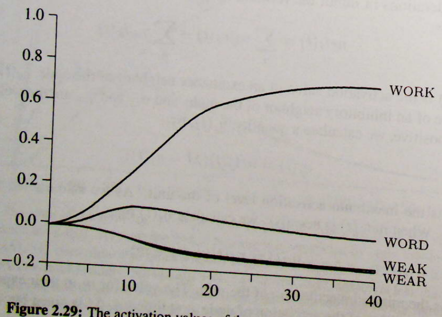


Figure 2.29: The activation values of the words as a function of time.

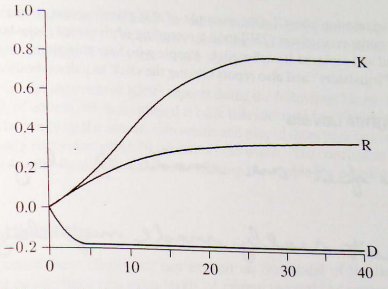


Figure 2.30: The activation values of the letters K, R, and D as a function of time.

Figure 2.29 shows how the words WORK, WORD, WEAK, and WEAR are activated by the algorithm given the incomplete word, WORK, shown in Figure 2.19. WEAK and WEAR, while they have features in common with WORK, are inhibited by the more likely words, WORK and WORD, and these inhibited words are pushed down below their resting levels. WORD first increases a little and then decreases to a near zero value. With respect to the letter activations, as you can see in Figure 2.30, K is the winner. This is because K is receiving support up from the features and down from the word, WORK, while R only receives support from the bottom up.

As we said earlier on, this program was designed to model results obtained by flashing words and portions of words on a screen. Rumelhart and McClelland discuss their results in their two papers on the matter. It is not important for us to discuss those results here. It is enough to say that their method produces very similar results to those obtained from tests done on human subjects.

2.4 Expanding the Pattern Recognition Hierarchy

The top-down and bottom-up effects used in recognizing letters also extend to how we interpret the sounds we hear. Moreover, the interpretation of words we hear and see depends on still higher level patterns, namely sentences and our knowledge of the world. In this section we look at these phenomena and how they make producing an artificial system that is equivalent in capability to a human being very difficult.

2.4.1 Hearing

The last section demonstrated that people can fill in obscured or uncertain visual inputs based on context using their knowledge of words. An analog to this visual phenomenon also occurs in auditory processing where, for example, there is a phenomena known as the "phonemic restoration effect." One example of this phenomenon comes from an experiment where some researchers [250] took a recording of the word "legislature," edited out the "s" sound and replaced it with a click. People who hear this altered recording hear the entire word "legislature" and also report hearing the click as a disembodied sound.

2.4.2 Higher Levels

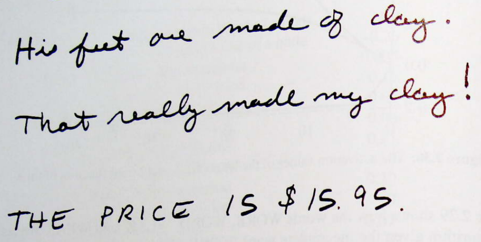


Figure 2.31: The interpretation of identical curves is highly dependent on the context in which they are seen.

The filling in ability of the mind also extends beyond the word level to the higher levels of sentence processing and interpreting events in the world. For instance, suppose we have the sentence:

The ___ broke the window.

where the blank space is either blank or so badly printed as to be completely illegible. First, our knowledge of sentence structure prompts us to think that the blank space should contain a noun. Also, our knowledge of the world prompts us to think that the noun may represent a person or some heavy object such as a rock. Another example of this phenomenon is shown in Figure 2.31 where the identity of a curve is highly dependent on the context in which it is seen. Likewise, the interpretation of sounds is also influenced by the context as shown in this story:

> In Center Harbor, Maine, local legend recalls the day some 10 years ago when Walter Cronkite steered his boat into port. The avid sailor, as it's told was amused to see in the distance a small crowd of people on shore waving their arms to greet him. He could barely make out their excited shouts of "Hello Walter . . . Hello Walter."
>
> As his boat sailed closer, the crowd grew larger, still yelling "Hello Walter . . . Hello Walter." Pleased at the reception, Cronkite tipped his white captain's hat, waved back, even took a bow.

> But before reaching dockside, Cronkite's boat abruptly jammed aground. The crowd stood silent. The veteran news anchor suddenly realized what they'd been shouting: "Low water . . . low water." [1]

In a more scientific experiment Klatt[2] reports doing the following: He recorded continuous speech and, of course, when he played it back listeners could recognize all the words. However, when he broke up the speech into words and played them back in a random order, listeners could only recognize about 70 percent of the words. The conclusion is that meaning and word order play a significant role in making it possible for people to understand speech.

2.4.3 The Hierarchy

To take into account these effects we can expand on the model of the last section and hypothesize that people have two extra levels of pattern recognizing processors as shown in Figure 2.32. We will designate them as the "sentence level recognizers" and "event level recognizers." Although this hierarchy makes sense, there is as yet no complete system that uses it, so this arrangement must still be considered highly theoretical.

While it was quite easy to model word recognition by having a node for each of 1,179 four-letter words, it is clearly not so simple to adopt that plan for the sentence recognition level. Keeping an example of every kind of sentence is not reasonable. Researchers in natural language processing have generally held that in trying to understand the meaning of sentences the proper way to proceed is to look for patterns of nouns, verbs, adjectives, and so forth. They believed that only after determining these grammatical aspects could you go on to consider the meaning of a sentence. Years of research, however, has found problems with this approach. Consider, for instance, the simple sentence:

Time flies like an arrow.

People readily discern the meaning: time passes quickly. However, there are other interpretations that are possible to a machine that only knows about nouns, verbs, adjectives, and so forth. Flies could be interpreted as a noun, and then time must be an adjective, giving a possible meaning to a naive machine that creatures known as "time flies" like arrows. Or time could be interpreted as an imperative verb and flies as a noun giving the meaning that "You should time flies the same way as you time arrows." It is human knowledge of the world that enables people to figure out the meanings of sentences and to figure out whether or not a particular word is being used as a noun or a verb or an adjective.

It is far from settled exactly how to do sentence recognition and neither does anyone really know how to represent the real world information, however, there is one project done by Waltz and Pollack [247, 157] that can make sense of ambiguous words in sentences by crudely taking into account facts about the world. Their research shows how easily an interactive activation model can potentially be used to sort out meanings of sentences and

[1] Don Oldenberg, *Chicago Sun-Times*, June 10, 1987, and the *Washington Post* February 27, 1987, reprinted with permission from the Washington Post.

[2] This report comes from the article by White [257] where the author gives this report as being from a personal communication with Dennis Klatt.

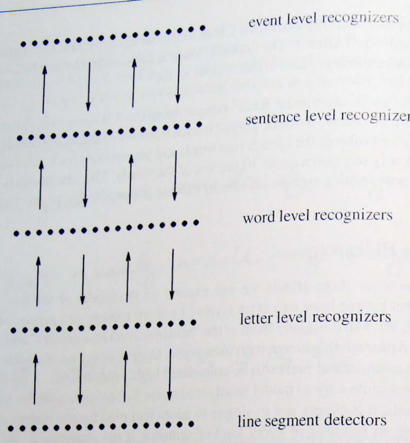


Figure 2.32: Human pattern recognition depends not only on low level features but on higher level patterns as well.

correctly identify nouns, verbs, and so on. One example from Waltz and Pollack uses the sentence:

John shot some bucks.

There are many possible meanings for the words “shot” and “bucks.” The possible interpretations for shot are “worn out” (tired, an adjective), “fire” (shoot with a gun, a verb), “bullet” (a unit of ammunition, a noun), or “waste” (as in squander, a verb). “Bucks” could mean either “resist” (opposes, a verb), “throw off” (a bucking horse, a verb), “dollar” (a unit of currency, a noun), or “deer” (male deer, a noun). People who think that John went to Las Vegas will interpret “shot” as “waste” and “bucks” as “dollars” while people who think that John went hunting will interpret “shot” as “fire” and “bucks” as “male deer.”

The Waltz and Pollack program works by looking up the possible meanings of all the words in a sentence and constructing an interactive activation network that will be used to settle on a consistent meaning for all the words. The network for “John shot some bucks.” is shown in Figure 2.33 and it consists of four parts. First, the top part is a *parse tree*, a way of diagramming a sentence that shows how it can be generated from a formal grammar.[3] The second part of the network is the set of words used in the sentence. The third part is the most complicated. It contains the possible meanings of the words. The meanings of

[3] Parse trees and formal grammars will be described in Chapter 10 but understanding these concepts is not really important now.

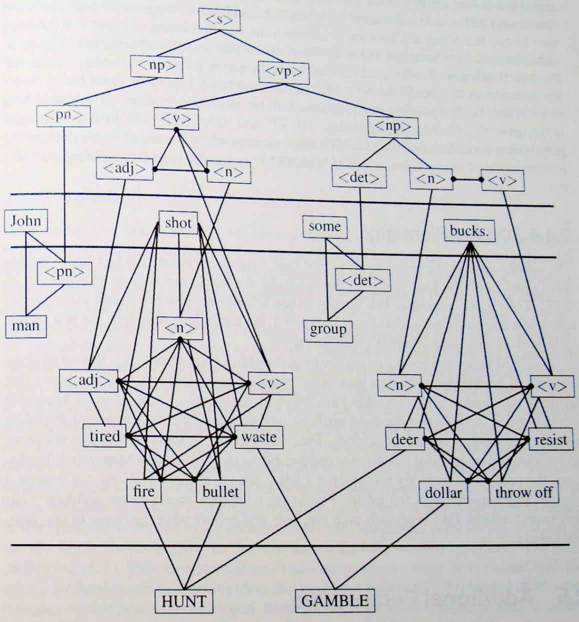


Figure 2.33: The Waltz and Pollack program constructed this network to determine the meaning of the sentence, “John shot some bucks.” It consists of four parts: at the top a network designed to resolve the syntax of the sentence, the next layer down lists the words in the sentence and these are activated from left to right, below this are subnetworks to determine the parts of speech of each word (lexical analysis), and finally at the bottom two nodes give the possible contexts in which the sentence is seen. In the syntactic part, the notation, <s> means a sentence; <np>, a noun-phrase; <vp>, a verb-phrase; <v>, a verb; <n>, a noun; <adj>, an adjective; <pn>, a proper noun; and <det>, a determiner.

the words "John" and "some" do not pose a problem. "John" is a proper noun, the name of a man, and "some" is a simple determiner for a group. In the "shot" subnetwork there are nodes to indicate that "shot" is being used as an adjective, a noun, or a verb. Given the context that a verb is the only candidate that could follow the noun "John," it is quite easy to see that when the network is activated the verb meanings of "shot" are the only ones that could come up, so either the "fire" or "waste" nodes will come on. Just as in the "shot" subnetwork where "shot" could only be a verb based on the order of words in the sentence, in the "buck" subnetwork, "bucks" could only be a noun, again based on the order of words. The possible interpretations will be "deer" and "dollars." The bottom part of the network contains only two nodes, "HUNT" and "GAMBLE" and these represent the possible contexts that apply. If "HUNT" is on, the network will ultimately give the hunting interpretation of the sentence, while if "GAMBLE" is on, the gambling meaning will win out.

2.4.4 On the Hierarchy

The simple networking hierarchy proposed here accurately reflects the fact that a human mind takes in all the facts and then tries to make the maximum amount of sense out of them. There is no guarantee that human minds actually achieve this result using simple interactive activation networks. In all likelihood, the algorithms seen here are just simple models of a much more complex architecture.

A second important point that comes from the above considerations is that to produce an artificial intelligence with near human performance capabilities will require giving that artificial intelligence a considerable knowledge of the world. Knowing *a lot* about the world is not something that the early artificial intelligence researchers expected would be necessary to create intelligent machines. This "knowledge" factor clearly makes the problem of creating an artificial intelligence comparable to that of a human being much harder. Even the process of correctly recognizing a letter can get complicated and can require a considerable knowledge of the world. People see what they expect to see and hear what they expect to hear. People perhaps even see, hear, and believe what they *want* to see, hear, and believe.

2.5 Additional Perspective

The pattern recognition methods discussed in this chapter have been chosen because the principles involved are representative of how vision programs work; however, they do not represent all that can be said about this subject so in this final section we will add some perspective by mentioning other issues and systems.

2.5.1 Other Systems

Other algorithms to recognize hand-printed and typed characters are also being developed. A recent system by Le Cun et al. [96] used the back-propagation algorithm (described in the next chapter) and various other techniques and it managed to achieve a 1 percent error

rate with a 9 percent rejection rate[4] on hand-printed characters taken from zip codes on actual US mail. A number of other such projects report similar results. In addition, Le Cun et al. constructed a system consisting of a video camera to scan digits which were then processed by a PC equipped with a special digital signal processor board. The system can process 10 to 12 digits per second.

Simple and efficient algorithms also exist to take handwritten characters from a tablet. These algorithms are faster and more accurate than scanning already existing characters because the system can detect the beginning and ending points of individual strokes as the strokes are made. For an example of one, see the Ledeen character recognizer described in Appendix VIII of [221].

In the way of hardware implementations, Mead has produced a "silicon retina" that behaves much like a human retina (see [110] and [111]).

2.5.2 Realism

Some researchers like Fukushima believe that their models of visual processing are fairly realistic, however, it is not really known *exactly* how the eye and nervous system process the data that a retina receives. One interesting result of studying human visual hardware is that it tries to get by with sending a minimum of information to the brain. For instance, suppose we had a retina consisting of a 100 × 100 array of cells. This comes to 10,000 cells and it represents a lot of information to be passed along the optic nerve. However, the cells of the retina are designed to only send information when they detect a change in the color or intensity of light hitting them. Figure 2.34 shows a pattern, half black and half white. If we place this picture on the retina, then, after about a tenth of a second, most of the cells will stop sending information except for the cells along the border between the two colors. The cells along the border will keep sending information because the eye actually oscillates with a frequency of about 10 cycles per second so the border between the two colors is actually moving back and forth on the cells of the retina. These cells with changing inputs keep reporting while the other cells stop reporting. Now if the retina was 100 × 100 cells, instead of 10,000 cells reporting, only several hundred will have to report. In the brain, the brain fills in the enclosed areas with the correct color so you continue to see the whole picture despite the fact that most cells have stopped reporting. (For more on this see [82].) This experimental result and many others cannot be explained with the simple artificial neural networking ideas that have been seen in this chapter and much more complex models need to be developed. Some more complex and realistic models of human visual processing can be found in the first four chapters of [54] and they can explain many visual illusions.

Remember, too, as mentioned in Chapter 1, it has been recently discovered that at least some of the messages being passed between neurons involved in vision processing are not simple activation values, they are in fact coded messages.

2.5.3 Bigger Problems

[4]When the difference between the ratings for the two highest patterns is not large enough, the network does not attempt to classify the unknown pattern and these very close calls are rejected rather than being classified. By rejecting close calls you lower the error rate.

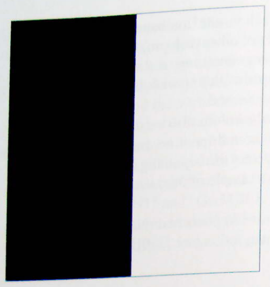


Figure 2.34: When this pattern is placed on a retina all the cells stop reporting to the brain after about a tenth of a second except for cells that are near the border between black and white.

In this chapter we have been working with simple letters and numerals because it is quite easy to do so; however, recognizing three-dimensional objects and finding objects in a scene has also been an important research area. The principles involved in recognizing more complex objects are the same as those involved in recognizing letters, that is, small edges are detected, these edges are then used to detect more complex features, and the more complex features point toward the identity of the object. Some methods only work from the small features upward while other methods also work from the top down. Perhaps the most well-known method for recognizing 3D objects has been developed by Marr [105], however Grossberg (see [54], Chapter 2) argues that Marr's methods are unrealistic and he has developed other methods to do the processing that he argues are more realistic.

2.6 Exercises

2.1. In Section 2.1 the pattern recognition method was illustrated using only the patterns E, F, and H. Expand this base of patterns to include at least five more letters and determine how well your program works by submitting a few test cases to the program. If you have a computer or a terminal with some graphics capability where you can draw letters on the screen, expand on the previous exercise and write a program so that people can draw letters on the screen and then have the program identify them.

2.2. Figure 2.5 did not have enough room to show the weights on the connections. Using the matrix in Figure 2.4, jot down the the values of the weights that go from input nodes 1, 2, and 3 to the E, F, and H nodes.

2.3. Given the letter E shown in Figure 2.6, produce the four S- and four C-layer matrices the algorithm will produce. One S-layer matrix and one C-layer matrix are shown in Figure 2.9.

2.4. In Section 2.1 the algorithm located the presence of subpatterns or operators in a very conventional way by searching the entire matrix to find them. If you did Exercise 2.1 modify the program so that it uses sets of neurons the way the Neocognitron does. Do this with the four subpatterns and nine regions used in the algorithm in Section 2.1.

2.5. Working by hand, determine what the state of the AM/AN network will be after the second and third iterations, given the values shown in Figure 2.25.

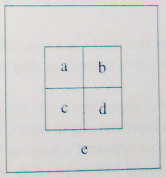
2.6. If the network in Figure 2.24 is given the features of an M, minus feature 6, will the program recognize the word as AM?

2.7. Program either the simplified network algorithm (used with the AM and AN example) or the slightly more complicated version actually used by Rumelhart and McClelland. In either case, use data for the WORK/WORD example where part of the last letter is missing. Use data consisting of the short line segments used in the text. Since this algorithm is useful for other exercises in this chapter you may want to produce a general purpose implementation of the algorithm rather than one that is specifically tailored to the WORK/WORD example.[5]

2.8. If you are interested in the effects that occur in the Rumelhart and McClelland word recognition network that follow the results on human subjects, read those portions of the Rumelhart and McClelland papers that discuss this and then give a brief listing and summary of the effects.

2.9. It seems that the word recognition network of Rumelhart and McClelland may be quite a nice way to recognize misspelled words. Consider how well it could do this if the misspelled words included missing and extra characters. Consider what would be necessary in the way of a sequential algorithm to recognize misspelled words that include missing and extra characters. Compare the two methods.

2.10. Below is a small map consisting of regions a, b, c, d, and e:



The map can be colored using three colors, say red, green, and blue, such that a region of one color never has a common border with a region of the same color, except possibly, if the regions meet at a point. If, for instance, a is green then b, c, and e cannot be green. The constraints involved in coloring the map can be wired into an interactive activation network. For instance, there can be a node that stands for region a being green, a node for

[5] This program is available on the Internet.

region a being red, a node for region c being green, and so on. If the node for a being green lights up, then this should inhibit regions b, c, and e from being green, as well as activating the possibility that b is red, b is blue, c is red, c is blue, e is red, e is blue, and so on. Devise a network and a set of weights that will find the colors of regions b, c, d, and e given that a is green. If you are not given the color of any region can this method still find solutions?

2.11. An interactive activation network can be used to choose tic-tac-toe moves. Create such a network and show how it can play an entire game starting from an empty board. Here is an outline of how this can be done.

First, here is a way to represent the input game board. Suppose the board is the following:

X		O
	O	
		X

The positions can be represented using 18 nodes divided up as follows:

1 0 0
0 0 0
0 0 1
	0 0 1
	0 1 0
	0 0 0

Values of the nodes 1 through 9 representing the presence (1) or absence (0) of an X in that square.	Values of the nodes 10 through 18 representing the presence (1) or absence (0) of an O in that square.

Naturally, for the output layer of the network you will need 9 nodes and the goal will be to light up the one output node that will be the move to make.

To generate moves, some important principles must be wired into the network:

a) First, if a square is already occupied, the network cannot be allowed to select this square as its move.

b) Second, if you already have two marks in a row, column, or diagonal and there is an empty space available in the row, column, or diagonal, you should take the empty space so as to win.

c) Third, if your opponent has two marks in a row, column, or diagonal and there is an empty space available in the row, column, or diagonal, then you must block the win by your opponent.

First, work out a network that will meet these basic conditions. It will be convenient to have a layer of units between the input and output layers. The units in this layer can detect pairs of your nodes and pairs of your opponent's nodes as well as possibly other features of the game board. One additional useful feature you may want to add to your network is to put a threshold on each unit so that it does not even begin to turn on until the input exceeds a certain threshold. This principle makes it easy for units in the middle layer to stay off unless they receive input from two sources.

Besides the basic requirements in b) and c), you will need to devise some way to make other reasonably good moves early in the game when these important conditions are not present. One useful feature here would be to have some of the middle layer units start out on (= 1.0) instead of off. This is especially useful to have when the game board is empty.

In addition, use a random updating scheme. In the Rumelhart and McClelland model, all the units update at once. A certain amount of randomness can be brought into the network by choosing a unit at random and updating it. Then randomly choose another unit and update it and then another, and so on. Demonstrate that your network can decide to make different moves given the same board configuration.

Chapter 3

Pattern Recognition II

In this chapter we will look at more mathematically rigorous approaches to doing pattern recognition. The most important of these will be the back-propagation algorithm. This remarkable algorithm can be used to do a variety of highly useful pattern recognition tasks. A number of applications of this algorithm will be shown.

3.1 Mathematics, Pattern Recognition, and the Linear Pattern Classifier

The pattern recognition algorithms that have been presented here so far were designed by their creators simply because they thought the algorithms would work. Much of AI is simply done that way. You believe something will work and then you test it to see if it does. To more mathematically inclined people this experimental proof of success, by itself, is not completely acceptable. If at all possible, algorithms should be proven correct. The side effects of such proofs should also give an insight into why the algorithm works, under what (if any) special conditions it works, and hopefully too, some way of estimating how long it will take to work.

Most pattern recognition work actually has been a mathematical problem with the following form. Some characteristics of an unknown pattern are measured and these characteristics are listed in a vector we will call $\vec{u}$. If, for example, you were trying to predict the weather, the measurements might include the barometric pressure, wind direction, wind speed, temperature, cloud cover, and humidity. If, for example, you are trying to predict the stock market, you would probably want to list at least the changes for the last few days, the interest rate, inflation rate, price/earnings ratios, and so on. In any case, an operator, we will call it Op, is then applied to the vector $\vec{u}$ and it gives the identity of the unknown pattern. In a mathematical format it is simply:

$$answer = Op(\vec{u}).$$

Much of pattern recognition consists of finding and studying good operators.

3.1.1 The Linear Pattern Classifier

To be more concrete about the matter we will look at an instance of this approach in a very simple case, the linear pattern classifier. Take, for example, a set of four items of the class

A and another set of four items of the class B. These might, for instance, be four examples of the letter E and four examples of the letter F. There will be only two characteristics measured for each item. For items from class A, let the characteristics be:

$$(-6, 4) \qquad (-6, -1) \qquad (-2, -2) \qquad (4, 2)$$

Let the characteristics measured for the items in class B be:

$$(7, -1) \qquad (4, -2) \qquad (-1, -4) \qquad (-4, -7)$$

These eight points are plotted in the xy-plane in Figure 3.1. They have been chosen so that they are *linearly separable*. A linearly separable set of patterns is one in which a line, a plane, or a hyperplane can be drawn between two different sets such that all the patterns in one set are on one side of the line, plane, or hyperplane, while all the patterns in the second set are on the other side of the line, plane, or hyperplane.

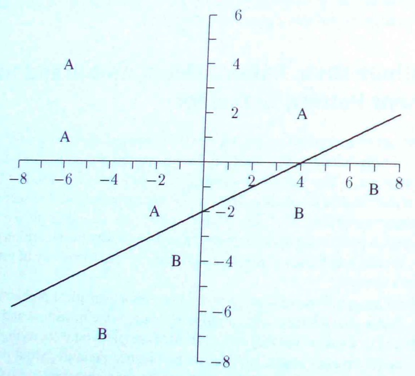


Figure 3.1: A set of linearly separable points.

From just looking at the graph of the points in each class, we can conclude that the line $y = x/2 - 2$ (and many other nearby lines as well) can be used to separate the region of A's from the region of B's. Writing the equation as $x - 2y - 4 = 0$, the coefficients give a *weight vector*, $\vec{w} = (1, -2, -4)$, that can be used to separate the two classes of objects. The algorithm requires that we augment each member of the set of eight patterns by adding a third coordinate with a constant value of +1. The examples of class A are now:

$$(-6, 4, 1) \qquad (-6, -1, 1) \qquad (-2, -2, 1) \qquad (4, 2, 1)$$

and the examples of class B are now:

$$(7, -1, 1) \qquad (4, -2, 1) \qquad (-1, -4, 1) \qquad (-4, -7, 1)$$

When we take $\vec{w}$ and form the dot product of it with a member of class A, we will get a negative value. Dotting it with a member of class B will give a positive value. For example, $(1, -2, -4) \cdot (-6, 4, 1)$ gives -18 and $(1, -2, -4) \cdot (-1, -4, 1)$ gives +3. Figure 3.2 shows how the algorithm can be formulated as a neural network. In the figure, the points $(-6, 4, 1)$ and $(-1, -4, 1)$ are submitted to the network and the output, whether positive or negative, gives the pattern classification.

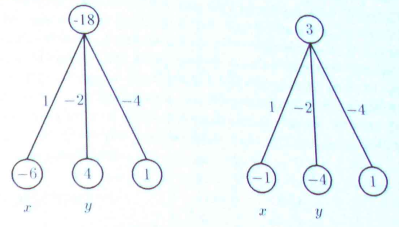


Figure 3.2: The linear pattern classifier can be viewed as a neural algorithm. The output unit looks at its inputs and sums them. The coefficients in the linear pattern classifier are the weights in the network. The coordinates of a point to be classified are input to the units in the input layer. On the left, the network takes in the point, $(-6, 4, 1)$ and puts it in class A. On the right, the same network takes in the point $(-1, -4, 1)$ and puts it in class B.

In the above example it was easy to find a weight vector by looking at the points laid out in two dimensions, however, it is not so easy if the points are in three-, four-, or n-dimensional space. Fortunately, however, a learning algorithm exists that is guaranteed to find a weight vector that can be used to separate linearly separable classes. The training procedure works by taking members from each set and dotting them with an estimate for the weight vector. Any initial value for the weight vector will work. If the weight vector gives the wrong answer, it is changed, but if it gives the correct answer it remains unchanged. The details as to how to change the weights are as follows. If $\vec{w} \cdot \vec{u}$ gives a negative or zero value for a pattern $\vec{u}$, and this is wrong, change the weight vector according to the formula:

$$\vec{w} \leftarrow \vec{w} + c\vec{u}$$

where c is some positive constant. On the other hand, if $\vec{w} \cdot \vec{u}$ gives a non-negative value and the value should be negative, change $\vec{w}$ by:

$$\vec{w} \leftarrow \vec{w} - c\vec{u}.$$

Members of both sets are continually submitted to the algorithm until it comes up with a value for w that works for all the points. It can be shown that for any positive value of c, the training process will converge. For a proof see [142].

As an example of how the weight vector converges to a value that will separate the two classes, we start with the weight vector, $\vec{w} = (0, 0, 0)$ and keep modifying it until it can separate all eight points correctly. The results of the computations are shown in Figure 3.3. The result is a weight vector, $\vec{w} = (4, -5, -7)$ that represents a line quite close to the one we chose by inspection.

PATTERN VECTOR, $\vec{u}$	WEIGHT VECTOR, $\vec{w}$	$\vec{w} \cdot \vec{u}$	CORRECT RESPONSE	NEW WEIGHT VECTOR
-6 4 1	0 0 0	0	−	6 -4 -1
7 -1 1	6 -4 -1	45	+	6 -4 -1
4 -2 1	6 -4 -1	31	+	6 -4 -1
-6 -1 1	6 -4 -1	-33	−	6 -4 -1
-2 -2 1	6 -4 -1	-5	−	6 -4 -1
-1 -4 1	6 -4 -1	9	+	6 -4 -1
4 2 1	6 -4 -1	15	−	2 -6 -2
-4 -7 1	2 -6 -2	32	+	2 -6 -2
-6 4 1	2 -6 -2	-38	−	2 -6 -2
7 -1 1	2 -6 -2	18	+	2 -6 -2
4 -2 1	2 -6 -2	18	+	2 -6 -2
-6 -1 1	2 -6 -2	-8	−	2 -6 -2
-2 -2 1	2 -6 -2	6	−	4 -4 -3
-1 -4 1	4 -4 -3	9	+	4 -4 -3
4 2 1	4 -4 -3	5	−	0 -6 -4
-4 -7 1	0 -6 -4	38	+	0 -6 -4
-6 4 1	0 -6 -4	-28	−	0 -6 -4
7 -1 1	0 -6 -4	2	+	0 -6 -4
4 -2 1	0 -6 -4	8	+	0 -6 -4
-6 -1 1	0 -6 -4	2	−	6 -5 -5
-2 -2 1	6 -5 -5	-7	−	6 -5 -5
-1 -4 1	6 -5 -5	9	+	6 -5 -5
4 2 1	6 -5 -5	9	−	2 -7 -6
-4 -7 1	2 -7 -6	35	+	2 -7 -6
-6 4 1	2 -7 -6	-46	−	2 -7 -6
7 -1 1	2 -7 -6	15	+	2 -7 -6
4 -2 1	2 -7 -6	16	+	2 -7 -6
-6 -1 1	2 -7 -6	-11	−	2 -7 -6
-2 -2 1	2 -7 -6	4	−	4 -5 -7
-1 -4 1	4 -5 -7	9	+	4 -5 -7
4 2 1	4 -5 -7	-1	−	4 -5 -7
-4 -7 1	4 -5 -7	12	+	4 -5 -7
-6 4 1	4 -5 -7	-51	−	4 -5 -7
7 -1 1	4 -5 -7	26	+	4 -5 -7
4 -2 1	4 -5 -7	19	+	4 -5 -7
-6 -1 1	4 -5 -7	-26	−	4 -5 -7
-2 -2 1	4 -5 -7	-5	−	4 -5 -7

Figure 3.3: Responses during training of the simple linear pattern classifier.

3.1.2 ADALINEs and MADELINEs

Some other versions of linear pattern classifiers have also been researched. Widrow re-searched linear pattern classifiers from the standpoint of using them as adaptive filters in electronics. He called his linear classifier an ADALINE for ADAptive LInear NEuron. Widrow also experimented with putting many ADALINEs together. These were called MADALINEs for Multiple ADALINEs. One very interesting application of an ADALINE was done by Widrow [258] at Stanford in the early 1960s where he used a linear pattern classifier to predict whether or not during the rainy season in San Francisco it would rain today, tonight, or tomorrow. The input data was a set of barometric pressure readings and changes in pressure readings in the Pacific Ocean from Alaska down to almost the equator. The results of the experiment were that the program was able to predict rain for San Francisco just as accurately as Weather Bureau meteorologists. Widrow also designed an "artificial neuron" in which the input weights could be changed by plating or unplating copper on very thin pencil lead.

3.1.3 Perceptrons

Another important researcher into pattern recognition using neuronlike elements was Frank Rosenblatt [177]. His systems of linear neuronlike elements were known as *perceptrons*. (The term perceptron is often applied to the linear pattern classifier as well.) The learning algorithm employed was known as the perceptron convergence procedure. Its learning rule is known as the *delta rule*. (This rule is derived in Appendix A.) The ability of perceptrons and the delta rule and any linear pattern classifier to learn complex functions is limited, however. In 1969, Minsky and Papert produced a book, *Perceptrons* (reprinted and expanded in 1988 [125]), in which they showed some of the limitations of perceptrons. This book is often credited with almost completely stopping research in the field. Since then, however, researchers have produced improved networks with nonlinear activation functions that are capable of learning nonlinearly separable patterns. The back-propagation procedure described in Section 3.4 can learn such patterns.

3.2 Separating Nonlinearly Separable Classes

Most pattern recognition problems cannot be solved by linear neurons because the surfaces separating the patterns are not linear. Since researchers realized the limitations of networks of linear neurons, most pattern recognition research has tried to find ways to separate pattern classes using more complex pattern recognition operators. Thus, there are operators that can take a set of points like the ones shown in Figure 3.4 and separate the points into two different classes using a surface more complicated than a straight line. We will neglect looking at these more complex operators and instead describe a few simple methods for dealing with nonlinearly separable patterns.

3.2.1 The Nearest Neighbor Algorithm

Possibly the simplest way to classify an unknown pattern at the point (x, y) is to compute the Euclidean distance from this point to every other known data point and find its nearest

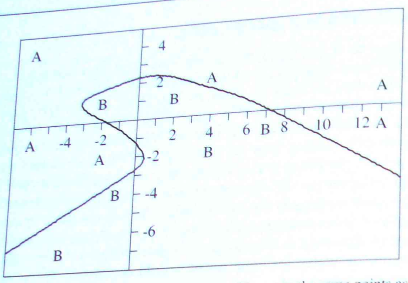


Figure 3.4: A set of points that is not linearly separable. These are the same points as in Figure 3.1 plus the points (13,1) and (13,-1) in class A and the points (-2,1) and (2,1) in class B. The line dividing the two classes is just one way to divide the space.

neighbor. We then assume the unknown has the same identity as its nearest neighbor. For instance, in Figure 3.5 we can assume that the point marked as 'X' is from class A and the point marked as 'Y' is from class B. Figure 3.5 also shows how the space is then divided up using the 12 data points. In actual use, the more data points you have available, the better the classification accuracy will be. If 1,000 points are generated at random from the distribution shown in Figure 3.4 and another 1,000 points are used as test cases, about 98 percent of them will be classified correctly.

Besides classifying an unknown point by finding its nearest neighbor, you can find its k nearest neighbors and let each neighbor contribute one vote toward identifying an unknown. This is known as the *k-nearest neighbor algorithm*. Another variation is to let each of the k nearest neighbors contribute an amount that decreases with its distance from the unknown point.

A nearest neighbor algorithm by Simard, Le Cun, and Denker using a different distance measure (not Euclidean) has managed to do better than all other algorithms on two difficult databases of handwritten digits [208].

3.2.2 Learning Vector Quantization Methods

There has also been a recent series of algorithms called Learning Vector Quantization algorithms: LVQ1, LVQ2, LVQ2.1, and LVQ3 (see [84, 85, 86]) and Decision Surface Mapping (DSM) from Geva and Sitte [51], that use the nearest neighbor algorithm but instead of storing a large number of points and searching through them to find the nearest neighbor, you store only a relatively small number of pattern vectors called *codebook vectors* or *prototype points* that are highly representative of the patterns in each class. In these methods you start with a initial set of prototype points and then move these points around to try to increase the classification performance of the nearest neighbor algorithm. Because the LVQ algorithms are new there have been few applications of them so far, but in a speech recognition

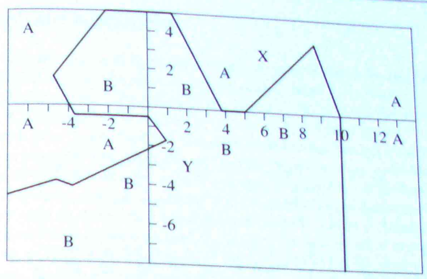


Figure 3.5: Using the nearest neighbor algorithm, point X will be classified as an A and point Y will be classified as a B. This figure also shows how the space is divided using the nearest neighbor classifier with the 12 data points.

problem Kohonen [85] reports better results than with any other algorithm he has tried.

The DSM algorithm is even simpler than the LVQ algorithms and it is reported to train faster and give better results than the LVQ algorithms for some types of problems. It works as follows. Let prototype point i be labeled as p_i and a training set point i, be t_i, then:

> Find the nearest prototype point, p_n.
>
> If p_n is the same class as t_i (a right answer), do nothing,
>
> otherwise, move p_n away from t_i using the formula:
>
> $$p_n = p_n - a(t_i - p_n),$$
>
> where a is around 0.3 or less and it slowly decreases with time. Furthermore, find the closest prototype point, p_c, that gives the correct answer and move it closer to t_i using the formula:
>
> $$p_c = p_c + a(t_i - p_c),$$
>
> where again, a decreases slowly with time. Repeat these steps for several passes through the training set.

As an example,[1] we will take 10 prototype points at random from the distribution given in Figure 3.3. These points are shown in Figure 3.6(a). We will then generate 1,000 points for training and 1,000 more points for testing. These points will be the same ones we mentioned using the simple nearest neighbor algorithm. The initial performance on the training and test sets is around 70 percent. We then make seven passes through the training set using the values of 0.3, 0.2, 0.1, 0.05, 0.025, 0.01, and 0.005 for a. The final result is shown in Figure 3.6(b) where about 98 percent of the training and test set points are classified correctly. Notice that with the simple nearest neighbor algorithm, approximately 100 times more prototype points were used to reach the same level of classification accuracy.

[1] Results will vary according to the random points generated for training and testing.

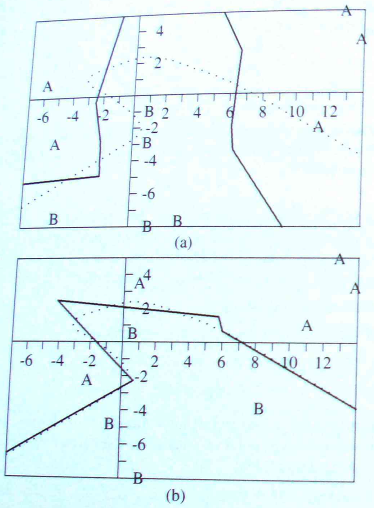


Figure 3.6: To demonstrate how DSM works, 10 prototype points were generated at random and these are shown in part (a). They give a 70% correct classification on the test set of 1000 points generated at random. After applying DSM the points have been moved to the locations shown in part (b) and they now give a 98.5% correct classification on the same test set. One of the B prototype points was moved below the area shown. The dotted lines in each part show the correct boundary.

3.3 Hopfield Networks

We will now look at a theoretically important type of network called the Hopfield network after Hopfield [68] who originated it. It is important because it is possible to prove that under the particular assumed conditions of the network, the network *will* converge to *an* answer, even if it is perhaps not the best possible answer. Knowing that a network will converge to an answer is quite important because some kinds of networks *never* converge. They may just constantly keep changing and so they never do settle down to any conclusion. Proving that a Hopfield network will converge is quite easy to do because there is a mathematical quantity associated with the network called the *computational energy* that will always decrease or, at the very worst, remain constant. When the computational energy of a Hopfield network stops decreasing, the network will be at *an* energy minimum, although not necessarily at the lowest energy minimum. The lowest minimum represents the best answer. To deal with this problem there is another network updating algorithm, the *Boltzman machine* relaxation algorithm that we will cover. Besides the theoretical importance of Hopfield networks, they are also important because they are designed to store items, or memories, in the weights connecting the units and so they are a candidate to model human memory as well.

3.3.1 The Hopfield Network

In a Hopfield network, the processing units take on only the two values 0 or 1. (Another version of the network has them take on the values −1 or 1.) Each unit has a threshold value associated with it such that if the input to a unit exceeds its threshold, the unit will turn on (become 1) or stay on, and if the input to a unit is less than the threshold, it will turn off (become 0) or stay off. The weights on the connections between the units take on the continuous set of values from $-\infty$ to $+\infty$. Also, if the connection weight from unit i to unit j is designated as w_{ij}, then the connection weight from unit j to unit i, w_{ji}, will equal w_{ij}. Such weights are said to be *symmetric*. Finally, the updating of the values for each unit is done *asynchronously*. This means that units in the network sort of take it upon themselves to update when they get the urge to do so (at random). If you "take pictures" of the state of the system at small enough time intervals, you will find only one unit is doing an update at a time. In the interactive activation network the updates for all the units take place at the same time, hence, in that type of network it is said that the updates take place *synchronously*.

The energy function for the n units in a Hopfield network is:

$$E = -\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i .$$

where w_{ij} is the weight between unit i and unit j, s_i is the state of unit i, s_j is the state of unit j, and θ_i is the threshold of unit i. To examine the meaning of the energy definition in detail, consider the simple network shown in Figure 3.7. We will first neglect the second term of the energy function by setting all the θ_is equal to 0. Here, the connection between a and b is +3/4, meaning that, if unit a is on, unit b should likely be on as well or if b is on then a should also be on. If we assume that a is currently on and b is currently off, the

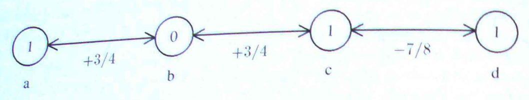


Figure 3.7: A simple Hopfield network with four nodes, a, b, c, and d with their values and weights as shown.

contribution of this pair to the energy of the system will be –3/4 × 1 × 0, but if b is turned on, the contribution will be –3/4 × 1 × 1. So now, if we do turn on unit b, the energy of the system will decrease by 3/4. Now we look at the link between c and d. The –7/8 says that if d is on, it is quite likely that c should be off or if c is on, it is quite likely that d should be off. In terms of the computational energy, if both c and d are on, the contribution of the c-d link is +7/8. The energy could be decreased by turning either c or d or both of them off. In effect, the formula contributes penalties to the total computational energy when units that are off should in fact be on, as well as when units that are on should in fact be off. The goal of the algorithm is to remove penalties and therefore decrease the energy of the network.

To see why there is the $\sum_i \theta_i s_i$ term in the energy definition, notice that the difference in the computational energy between the kth unit being on and the kth unit being off is:

$$\Delta E_k = \sum_i w_{ki} s_i - \theta_k$$

If this term is positive, it means that the input to the unit k from the other units ($\sum_i w_{ki} s_i$) is greater than the threshold of the unit k. As it turns out, this is exactly the rule used to decide whether or not the unit k turns on. So, the system can go about its computations just by having each unit look at the inputs from other units and all the time the system will be moving toward a minimum energy state.

As a concrete example of the process we consider the network of Figure 3.7 with the initial values shown. Let the thresholds for all nodes be 1/2. The initial energy of the network will then be 19/8. The node b will be getting 3/4 of an energy unit from node a and 3/4 of an energy unit from node c. Additionally, the θ term will contribute –1/2. This total is 1 and is greater than 0, so node b will be turned on. This changes the energy to 11/8. We now look at whether or not to change node c. The input it will be receiving from b is +3/4 and the input from d will be –7/8 and the threshold term will be –1/2, giving a total of –5/8, and because this is less than the threshold it will cause c to turn off. The network energy will now be 6/8. Figure 3.8 shows how the energy level of the network changes as the updates are made. Note that the energy levels are quantized. In a very large network the energy levels will approach a continuum. The updating process that moves a network from a higher state of energy to a lower state is also known as a *relaxation algorithm*.

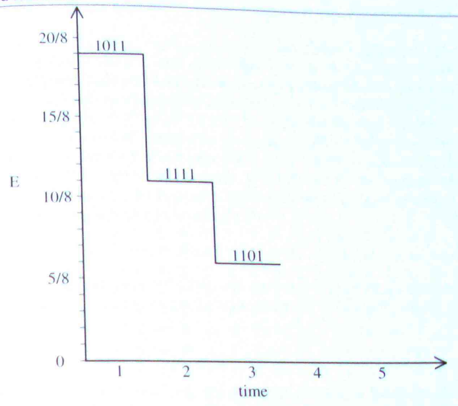


Figure 3.8: Starting with the initial state of 1011, updates to the network lower the energy level. This plot shows the progress after b and c are updated. How far will the energy drop if the updates are continued?

3.3.2 Storing Patterns

$$\begin{array}{ccccc} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{array} \qquad \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{array}$$

Figure 3.9: Two patterns that will be stored into a Hopfield network, a lowercase e and π. The units are numbered as shown on the right.

When a Hopfield network stores a pattern the pattern will be stored at a minimum of the energy function. Given part of a pattern, the network will update its units and move downhill to the minimum that represents the whole pattern. To illustrate how this works and to show how it is possible for the network to become stuck in a local minimum, let us take two small patterns in a 5 × 5 matrix, a lowercase e and the Greek letter π as shown in Figure 3.9. There will be 25 units and each unit will be connected to all the other units, meaning there will be a total of 25 × 25, or 625 connections, each with a weight. One way to determine the weights in a network is the following.[2] If a unit in the π pattern, say unit 1, is a 1 we

[2] Another way is given in the next section.

0	0	0	0	2	-2	2	0	2	-2	-2	0	-2	0	0	-2	2	0	2	0	0	0	-2	0	0
0	0	2	2	0	0	0	-2	0	0	0	2	0	2	-2	0	0	-2	0	-2	-2	2	0	2	-2
0	2	0	2	0	0	0	-2	0	0	0	2	0	2	-2	0	0	-2	0	-2	-2	2	0	2	-2
0	2	2	0	0	0	0	-2	0	0	0	2	0	2	-2	0	0	-2	0	-2	-2	2	0	2	-2
2	0	0	0	0	-2	2	0	2	-2	-2	0	-2	0	0	-2	2	0	2	0	0	0	-2	0	0
-2	0	0	0	-2	0	-2	0	-2	2	2	0	2	0	0	2	-2	0	-2	0	0	0	2	0	0
2	0	0	0	2	-2	0	0	2	-2	-2	0	-2	0	0	-2	2	0	2	0	0	0	-2	0	0
0	-2	-2	-2	0	0	0	0	0	0	0	-2	0	-2	2	0	0	2	0	2	2	-2	0	-2	2
2	0	0	0	2	-2	2	0	0	-2	-2	0	-2	0	0	-2	2	0	2	0	0	0	-2	0	0
-2	0	0	0	-2	2	-2	0	-2	0	2	0	2	0	0	2	-2	0	-2	0	0	0	2	0	0
-2	0	0	0	-2	2	-2	0	-2	2	0	0	2	0	0	2	-2	0	-2	0	0	0	2	0	0
0	2	2	2	0	0	0	-2	0	0	0	0	0	2	-2	0	0	-2	0	-2	-2	2	0	2	-2
-2	0	0	0	-2	2	-2	0	-2	2	2	0	0	0	0	2	-2	0	-2	0	0	0	2	0	0
0	2	2	2	0	0	0	-2	0	0	0	2	0	0	-2	0	0	-2	0	-2	-2	2	0	2	-2
0	-2	-2	-2	0	0	0	2	0	0	0	-2	0	-2	0	0	0	2	0	2	2	-2	0	-2	2
-2	0	0	0	-2	2	-2	0	-2	2	2	0	2	0	0	0	-2	0	-2	0	0	0	2	0	0
2	0	0	0	2	-2	2	0	2	-2	-2	0	-2	0	0	-2	0	0	2	0	0	0	-2	0	0
0	-2	-2	-2	0	0	0	2	0	0	0	-2	0	-2	2	0	0	0	0	2	2	-2	0	-2	2
2	0	0	0	2	-2	2	0	2	-2	-2	0	-2	0	0	-2	2	0	0	0	0	0	-2	0	0
0	-2	-2	-2	0	0	0	2	0	0	0	-2	0	-2	2	0	0	2	0	0	2	-2	0	-2	2
0	-2	-2	-2	0	0	0	2	0	0	0	-2	0	-2	2	0	0	2	0	2	0	-2	0	-2	2
0	2	2	2	0	0	0	-2	0	0	0	2	0	2	-2	0	0	-2	0	-2	-2	0	0	2	-2
-2	0	0	0	-2	2	-2	0	-2	2	2	0	2	0	0	2	-2	0	-2	0	0	0	0	0	0
0	2	2	2	0	0	0	-2	0	0	0	2	0	2	-2	0	0	-2	0	-2	-2	2	0	0	-2
0	-2	-2	-2	0	0	0	2	0	0	0	-2	0	-2	2	0	0	2	0	2	2	-2	0	-2	0

Figure 3.10: The matrix produced for the π/e example.

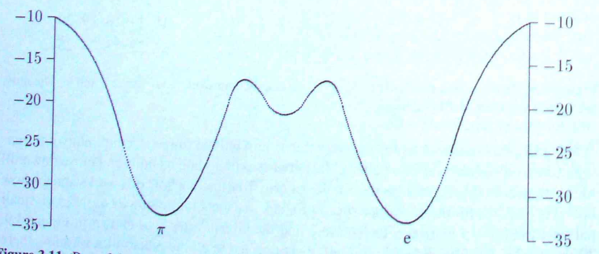


Figure 3.11: Part of the energy landscape for the π/e problem with π and e both at −33 and another shallower minimum at −21 in between. In reality the landscape is 26-dimensional and the energy levels are discrete, not continuous.

take a look at the other 24 units. Unit 2 is on so there should be an activation link with a value of +1 from unit 1 to unit 2 as well as from unit 2 to unit 1. (Notice how this recalls James' rule: "When two ideas have been active at once, or in immediate succession, one of them on re-occurring tends to propagate its excitement into the other.") There also will be +1 links from unit 1 to units 3, 4, 5, 7, 9, 12, 14, 17, 19, 22, and 24. Now when unit 1 is on, unit 6 should be off. This can be arranged by having an inhibition link (value −1) between unit 1 and unit 6. Notice that under this plan, every other 1 in the π pattern will also work to inhibit unit 6 as well as work to inhibit every other unit that is zero. (James did not say this!) Finally, if a pair of units are both zero, the weight between them is set to 1. These weights are conveniently defined by the formula:

$$w_{ij} = (2s_i - 1)(2s_j - 1).$$

Let the weights for the e be the matrix W_1, and the weights for π be the matrix W_2. The weights for the network that stores both the e and the π are obtained by simply adding the two matrices, W_1 and W_2 together:

$$W = W_1 + W_2$$

giving the matrix shown in Figure 3.10. We still need a threshold value for each unit and let us (rather arbitrarily) choose a value of +3 for every unit. Given all these weights, we end up with what we might call an "energy landscape" that contains two large minimas, one for π and one for e, and unfortunately many other low spots that the network can be trapped in. Representing the energy landscape for this problem is rather difficult because it is a 26-dimensional space; however by simplifying it, part of it looks like the one shown in Figure 3.11. The minimas for π and e are at −33 and in between them is another minimum with a value of −21.

Figure 3.12 shows the reconstruction process for the initial state shown in the upper-left-hand corner of the figure. The initial state has the characteristics of an incomplete e pattern and we clamp the units that are 1 at 1 while allowing the other units to update themselves. The relaxation process completes the e when the minimum of −33 is reached. In Figure 3.13, starting with an incomplete π pattern, the relaxation process becomes stuck at an energy value of −18. In fact, if the 1 units were not frozen at 1, the network would turn off unit 1 so it could fall down in the minimum, −21, shown in Figure 3.11. To get the reconstruction process moving again we can set unit 5 equal to 1, in effect giving the network another clue that the pattern is a π, this is shown in Figure 3.14. This raises the energy level to −17, but by updating the rest of the units the π pattern is completed. Because Hopfield networks are oriented toward storing patterns or memories in the network weights, it is sometimes said that when a network falls into a local minimum it is experiencing a kind of *deja vu*, in which it remembers something that it has never really seen.

3.3.3 The Boltzman Machine

In order to try and avoid local minima, researchers have devised a plan, where at the start of the updating procedure a lot of jumps uphill are made at random, and then as time goes on, the chance of taking a step uphill gradually decreases. This is the strategy of an abstract kind of machine, known as a *Boltzman machine* [66]. In it, the high probability of taking

$\begin{matrix}0&1&1&1&0\\1&0&0&0&0\\1&0&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ initial pattern $E = 6$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&0\\1&0&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 1 $\Delta E = -9$ $E = 6$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&0\\1&0&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 5 $\Delta E = -9$ $E = 6$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&0\\1&0&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 7 $\Delta E = -9$ $E = 6$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&0\\1&0&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 8 $\Delta E = -9$ $E = 6$
$\begin{matrix}0&1&1&1&0\\1&0&0&0&0\\1&0&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 9 $\Delta E = -9$ $E = 6$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&0&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 10 $\Delta E = 3$ $E = 3$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&0&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 12 $\Delta E = 3$ $E = 0$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&0&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 13 $\Delta E = 5$ $E = -5$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 14 $\Delta E = 5$ $E = -10$
$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 15 $\Delta E = -13$ $E = -10$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 17 $\Delta E = -13$ $E = -10$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 18 $\Delta E = -13$ $E = -10$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 19 $\Delta E = -13$ $E = -10$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 20 $\Delta E = -13$ $E = -10$
$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&0&0&0&0\end{matrix}$ test 21 $\Delta E = -13$ $E = -10$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&1&0&0&0\end{matrix}$ test 22 $\Delta E = 7$ $E = -17$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&1&1&0&0\end{matrix}$ test 23 $\Delta E = 7$ $E = -24$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&1&1&1&0\end{matrix}$ test 24 $\Delta E = 9$ $E = -33$	$\begin{matrix}0&1&1&1&0\\1&0&0&0&1\\1&1&1&1&0\\1&0&0&0&0\\0&1&1&1&0\end{matrix}$ test 25 $\Delta E = -17$ $E = -33$

Figure 3.12: Starting with a partial letter "e" shown in the frame in the upper-left-hand corner, the Hopfield network reconstructs the missing parts and the energy minimum is –33.

$\begin{matrix}1&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ initial pattern $E = -9$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 3 $\Delta E = 9$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 5 $\Delta E = -1$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 6 $\Delta E = -5$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 7 $\Delta E = -1$ $E = -18$
$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 8 $\Delta E = -17$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 9 $\Delta E = -1$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 10 $\Delta E = -5$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 11 $\Delta E = -5$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 13 $\Delta E = -5$ $E = -18$
$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 15 $\Delta E = -17$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 16 $\Delta E = -5$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 17 $\Delta E = -1$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 18 $\Delta E = -17$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 19 $\Delta E = -1$ $E = -18$
$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 20 $\Delta E = -17$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 21 $\Delta E = -17$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 23 $\Delta E = -5$ $E = -18$	$\begin{matrix}1&1&1&1&0\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$ test 25 $\Delta E = -17$ $E = -18$	

Figure 3.13: For the incomplete pattern shown in the upper-left-hand corner the network gets stuck in a local minimum with an energy of –18.

$\begin{matrix}1&1&1&1&1\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&0&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&0&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$
initial pattern	test 6	test 7	test 8	test 9
	$\Delta E = -7$	$\Delta E = 1$	$\Delta E = -17$	$\Delta E = 3$
$E = -17$	$E = -17$	$E = -18$	$E = -18$	$E = -21$

$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&0&0&0&0\\0&1&0&1&0\end{matrix}$
test 10	test 11	test 13	test 15	test 16
$\Delta E = -11$	$\Delta E = -11$	$\Delta E = -11$	$\Delta E = -17$	$\Delta E = -11$
$E = -21$	$E = -21$	$E = -21$	$E = -21$	$E = -21$

$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&0&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\end{matrix}$
test 17	test 18	test 19	test 20	test 21
$\Delta E = 5$	$\Delta E = -17$	$\Delta E = 7$	$\Delta E = -17$	$\Delta E = -17$
$E = -26$	$E = -26$	$E = -33$	$E = -33$	$E = -33$

$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\end{matrix}$	$\begin{matrix}1&1&1&1&1\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\\0&1&0&1&0\end{matrix}$
test 23	test 25
$\Delta E = -15$	$\Delta E = -17$
$E = -33$	$E = -33$

Figure 3.14: With an extra kick the network completes the π pattern. The kick consisted of setting unit 5 = 1.

a step uphill reflects a high "temperature" while as the system "cools down," the chance of a random step uphill decreases until the system "freezes" at a minimum in the energy landscape. This process of cooling the system is often referred to as "simulated annealing" because it is analogous to what happens in metals and other materials when they are melted and then cooled slowly. The atoms or molecules in the material have electric charges that repel and attract each other. When these atoms or molecules lose energy slowly enough they have a chance to align themselves in such a way as to minimize the energy of the solid. In this case, large crystals of the material will be formed, whereas if the material is cooled very rapidly, only very small microscopic crystals will form.

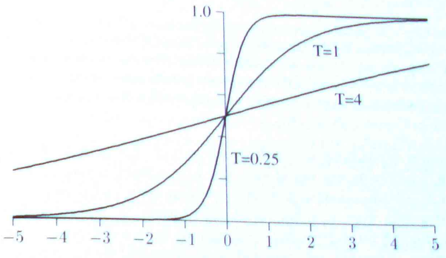


Figure 3.15: The probability of turning a unit on as a function of the unit's input for temperatures of 0.25, 1.0, and 4.0.

A simple modification of the Hopfield updating procedure leads to the Boltzman machine recall plan. The modification is that the probability, p_k, of a unit k becoming 1 (or staying at 1) at temperature T is given by:

$$p_k = \frac{1}{1 + e^{-\Delta E_k / T}}.$$

A graph of this function for several temperatures is shown in Figure 3.15. Notice that at high temperatures and negative values of ΔE_k, the probability of a unit turning on and increasing the computational energy is much greater than for low temperatures. As the temperature goes down, units that do not fit within the pattern are less likely to turn on. Even then, however, sometimes a unit with $\Delta E_k < 0$ will turn on. At very low temperatures you approach the Hopfield relaxation procedure where a unit only turns on if its input is greater than its threshold. The Boltzman procedure in effect lets the network sample a great many portions of the energy landscape. If in so doing the network happens to find a deep minimum, it is unlikely that taking a step upward will bring it out, so thereafter it will continue to settle down, whereas if it is in a shallow minimum, one step up could easily get the network out. The theory behind the Boltzman machine also shows that as the temperature approaches zero *slowly*, the probability that the network will be in the global minimum approaches 1.

The effectiveness of the Boltzman machine recall algorithm is illustrated in Figure 3.16 using the partial π pattern shown in the first frame of Figure 3.13. The network's initial

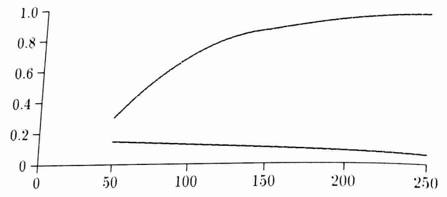


Figure 3.16: Given the partial π pattern shown in the first frame of Figure 3.13, the upper curve shows the probability of being in the global minimum of −33 and the lower curve shows the probability of being in the local minimum of −18 as a function of time. The time is the number of units that were updated at random while decreasing the temperature linearly from 4 to 0.25. When the network is cooled rapidly it often ends up in states other than −33 or −18 so the probabilities do not add to 1.

temperature was 4.0 and it was cooled linearly to 0.25. The amount of time taken to cool the network is simply the number of units that were updated at random as the system was cooled. The results show that the probability of being in the global minimum of −33 is only 0.3 when 50 updates were made but it rises to 0.93 when 250 updates were made.

Another method for overcoming local minima is to redefine the system so that the activation values of the units take on continuous values and the updates are done as in the interactive activation model [69]. This procedure does not guarantee that the system will settle into a minimum, however.

3.3.4 Pattern Recognition

We have been considering how Hopfield networks can do pattern completion in this section, but the methods can easily do pattern recognition as well. In the π/e example we could add two extra units to each network that code for the identity of the pattern. Unit 26 could indicate a π and unit 27 could indicate an e. Now given a pattern, even an incomplete one or one with a small amount of noise present, the network will complete it and identify it as well by turning on unit 26 for a π or unit 27 for an e.

3.3.5 Harmony

Some researchers have also defined a quantity similar to computational energy, called "goodness of fit," which their systems then attempt to maximize. Another similar concept is called "harmony," which a network also attempts to maximize [217]. When the object is to maximize a quantity, the searching process is called *hill climbing*.

3.3.6 Comparison with Human Thinking

The Hopfield/Boltzman models represent a much different way of computing than is found in standard von Neumann machines and they have some aspects that they share with people.

0	1	1	1	0
0	0	0	0	0
0	1	0	1	0
0	0	0	0	0
0	1	0	1	0

Figure 3.17: An ambiguous pattern that could be either π or e.

First, the Hopfield/Boltzman machines are, like people, somewhat unpredictable. In the π/e network, depending on the exact order in which the units update themselves, the network could reach different conclusions. Given the pattern in Figure 3.17 that contains only the features common to both π and e, and updating at random, it is clear that sometimes the network will settle down to the e minimum, but with a different sequence of updates it might settle down to the π minimum. This is interesting because if you give this problem or many similar kinds of problems like this to people, some people will find one answer and other people will find another, and even the same person may give one result on one occasion and the other result on another occasion. From the last chapter, recall the sentence, "John shot some bucks." If you do not know whether John was hunting or gambling, your mind could easily slip into either the hunting or the gambling interpretation. Because this algorithm produces the kind of random results you get from people, it makes some researchers feel that this model of artificial thinking may be close to how people's minds actually operate.

Second, another phenomenon that happens in human problem solving is that people often seem to get stuck in local minima. First, they may get stuck on a problem and not be able to find any solution at all, or second, they may find a solution to a problem that satisfies many of the required constraints, but not all of them. When this happens to people the solution is to start over fresh in hopes that a better solution will just pop into their minds. Starting over in Boltzman machine terminology corresponds to reheating the system and hoping that through random updates it will settle into a better solution. It is easy to see how this happens in the Hopfield/Boltzman networks, but it is not at all obvious how this phenomenon can happen in a classical von Neumann architecture running a classical algorithm with classical data structures. The von Neumann architecture is thoroughly predictable and there is no reason why the system should get stuck anywhere in the process or give different responses to the same input at different times. In addition to people not finding correct solutions to problems, when people are rushed to find an answer they will not necessarily find the best solution, but only an adequate solution. Again, in Boltzman machine terminology, being rushed corresponds to cooling the system quickly and then there is an increased chance that the best solution will not be found.

Although the Hopfield/Boltzman algorithms are extremely interesting from the standpoint of being much different than the classical symbol processing algorithms, very little has been done with them so far. One interesting application is by Halici and Sungur [56] where they use the Boltzman machine algorithm to solve the SOMA puzzle problem.

3.4 Back-Propagation

Back-propagation has become the single most useful neural networking algorithm. It is a generalization of the delta rule and its learning rule is sometimes called the generalized delta rule. Back-propagation networks are sometimes referred to as multilayer perceptrons. In this section we give only some informal justification for the formulas, however there is a derivation of the back-propagation algorithm formulas in Appendix A. Until very recently it was thought unlikely that anything like back-propagation occurs in the brain, however there is now some speculation as to how the brain might implement back-propagation (see [224], [65], and [26]). Users of the algorithm call it backprop.

3.4.1 History

Back-propagation has been discovered a number of times. At this time it appears as if the first derivation of the algorithm was by Robbins and Monro in 1951 [176]. Their discovery was reported by White in 1989 [256]. Also in 1989, Hecht-Nielsen [65] noted that it was discovered by Bryson and Ho by 1969 [16]. In 1974, Werbos independently rederived it [252]. Since that time, Werbos has been studying the use of back-propagation in various economic modeling and artificial intelligence problems (for a listing of articles see [253] or [254]). Unfortunately, Werbos' work was not discovered by the AI community until 1987. Back-propagation was again independently rederived by Parker [146, 147] and finally made well known in 1986 by Rumelhart, Hinton, and Williams [180].

3.4.2 The Network

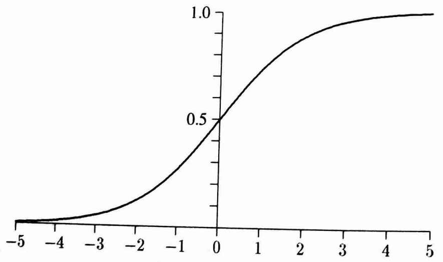


Figure 3.18: A plot of the most commonly used back-propagation activation function, $1/(1+e^{-x})$.

One of the specific reasons Minsky and Papert gave in their 1969 book, *Perceptrons*, for being pessimistic about neural networking was that networks with any number of layers and where the neurons use a linear activation function (as in the linear pattern classifier) could not learn many important and useful functions. For example, they could not compute the exclusive-or (XOR) of two binary inputs. Back-propagation uses a nonlinear activation

function and it can be used to do the XOR problem and other problems where nonlinear curves and surfaces are necessary, such as the problem of separating the nonlinearly separable regions shown in Figure 3.4. The back-propagation algorithm will work with many activation functions but the most commonly used one is:

$$o_j = \frac{1}{1+e^{-net_j}}$$

where net_j is the sum of the inputs to neuron j and o_j is the activation value of neuron j. This function is shown in Figure 3.18. Functions like this with an S-shaped character are referred to as *sigmoids*. net_j is defined as:

$$net_j = \sum_i w_{ij} o_i + \theta_j$$

where w_{ij} is the weight connecting unit i in a previous layer with unit j and o_i is the activation value of unit i. The term θ_j represents the weight from a *bias unit* that is always on (=1.0) and it functions like the threshold value in other networks.

This activation function has some important properties. If $net_j = 0$, so that there is no activation at all coming into node j, then $o_j = 0.5$. A value of 0.5 for a node therefore means the unit is "undecided." Also notice that to raise the activation value of a unit to exactly +1, net_j would need to be infinite and to achieve a value of 0, net_j must be negative infinity. Plus and minus infinity are rather hard values to reach so in reality people usually consider the output for a pattern to be correct if the values on its output node(s) that are supposed to be 1 are at least 0.9 and the values of its output node(s) that are supposed to be 0 are below 0.1. Less stringent limits are used at times. When a network is being tested with unknown patterns, any value greater than 0.5 is often considered to be on and any value less than 0.5 is considered to be off. Usually in classification problems the output unit with the largest value is considered to be the answer whether the largest value is greater than 0.5 or not.

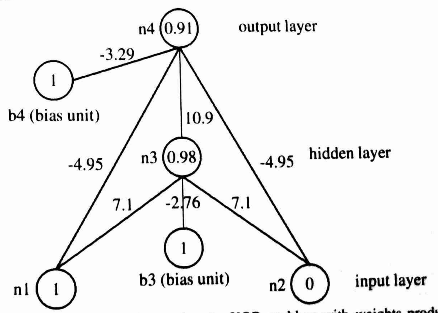


Figure 3.19: A three-layer network to solve the XOR problem with weights produced by back-propagation.

Figure 3.19 shows a network that has been trained to compute the XOR function. The topology of the network was chosen by hand but the weights for the connections have been found by the back-propagation algorithm. The layer of units in between the input and output units is called a *hidden layer* since if the network is viewed as a black box with input units and output units the hidden layer units cannot be seen. Unit b3 is the bias unit for the hidden layer unit, n3, and b4 is the bias unit for the output layer unit, n4. The patterns to be learned and the actual network responses to these patterns are:

n1	n2	n4 (desired)	n4 (actual)
1	0	1	0.91
0	0	0	0.08
0	1	1	0.91
1	1	0	0.10

This particular XOR network has connections from the first to the third layer. Typically, back-propagation networks only have connections between adjacent layers but there is one report that adding extra connections from the input to the output layer improves the algorithm [220]. Networks can also have connections between units in a single layer and the number of hidden layers is unlimited in principle. In practice, three-layer networks work well for most problems. Networks are often described by the number of units in each layer. A network with 19 input units, 12 hidden units, and 7 output units is then described as: 19-12-7. In this description the bias units are not counted. Ordinarily, bias units are not shown in diagrams of networks either.

3.4.3 Computing the Weights

To find the proper weights for a network, all the weights are started out at 0 or small random values. Then, a training pattern is placed on the input level units and this produces a response on the output units. This output pattern computed by the network is compared with what the answer should be and modifications of the weights throughout the network are made in order to make the computed answer come closer to the correct answer. Every weight in the network can be regarded as a variable or a dimension in space. This process of changing weights to try to minimize the error is then called a search through "weight space." The search to try to minimize the error is much the same as trying to minimize energy in a Hopfield network and the process is subject to the same problem as in the Hopfield network: it is possible to come to a local minimum that it is impossible to get out of if you only try to go down. Moreover, a second problem exists in back-propagation: if the weight changes are too large it is possible for the errors to increase as shown in Figure 3.20, whereas in the Hopfield network there was a guarantee that the energy would never increase.

As an example of the weight changing process, we will start with the network shown in Figure 3.21 where all the weights are initially 0. When we place the pattern (1,0) on the input units, the output unit has a value of 0.5 instead of the target value of 1. We now need some formulas that will change the weights so that the next time this pattern is presented,

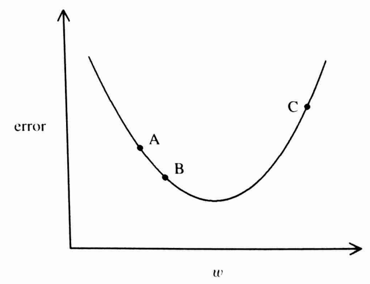


Figure 3.20: The error on the output units plotted as a function of a weight, w in the network. When the network is at point A, the error can be decreased a little by slightly increasing the value of the weight. On the other hand, a large change might put the network at the point C on the other side of the valley where the error is larger. The simplest way to avoid this problem is to make the weight changes fairly small.

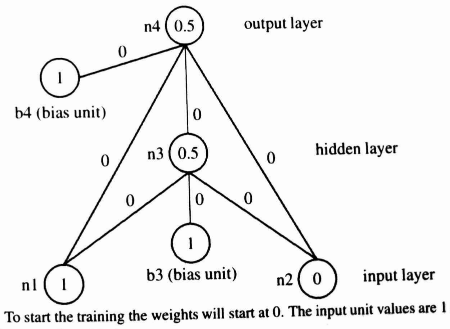


Figure 3.21: To start the training the weights will start at 0. The input unit values are 1 and 0, n3 and n4 are 0.5. The target for n4 is 1.

n4 will be a little closer to 1.0. In order to raise the value of the output unit there are two things that can be done. First, the weights coming from n1, n3, and b4 can all be increased. The second thing that can be done to increase the value of n4 is to raise the activation levels of nodes that feed into n4. In this case, however, the only node that can have its activation level raised is n3. The other nodes are frozen at values of 0 or 1. Raising the activation level of n3 can be done by raising the values of the weights leading into n3. First, we will look at raising the weights leading into n4. The back-propagation formula to modify these weights consists of several definitions:

$$\delta_k = (t_k - o_k) f'(net_k) \tag{3.1}$$

$$f'(net_k) = o_k (1 - o_k) \tag{3.2}$$

$$\Delta w_{jk} = \eta \delta_k o_j \tag{3.3}$$

The quantity δ_k in Equation (3.1) is called the error signal, t_k is the target value for unit k, and o_k is the actual activation value of node k. Note that the difference $(t_k - o_k)$ determines the extent of the error and it is called the simple error. Large differences between what you have as an answer and what you should have will result in larger corrections being made to the weights. The $f'(net_k)$ term given in Equation (3.2) is the derivative of the activation function, f, with respect to net_k. It comes as a consequence of the derivation. Finally, in Equation (3.3), the change to the weight Δw_{jk} that goes from a unit j in a lower layer to unit k in the output layer is the product of δ_k, o_j, the activation value of the unit j, and a small constant η, that controls the learning rate. This constant must be relatively small so that the error decreases. In our XOR example we will choose $\eta = 0.1$ and then the weight changes we compute are as follows:

$$\delta_{n4} = (1 - 0.5) \times 0.5 \times (1 - 0.5) = 0.125,$$

$$\Delta w_{n1n4} = 0.1 \times 0.125 \times 1 = 0.0125,$$

$$\Delta w_{b4n4} = 0.1 \times 0.125 \times 1 = 0.0125,$$

$$\Delta w_{n3n4} = 0.1 \times 0.125 \times 0.5 = 0.00625,$$

$$\Delta w_{n2n4} = 0.1 \times 0.125 \times 0 = 0.$$

Now, without making any weight changes just yet we proceed to the hidden layer units and compute weight changes for weights leading into these units. In this way, we will raise or lower the activation of units in this hidden layer. To do this, we need to give some error values to the hidden layer units. If we now let δ_k be the error found for the kth unit in the output layer (or any layer above), then the δ_j term for the jth hidden unit is:

$$\delta_j = f'(net_j) \sum_k \delta_k w_{jk}.$$

The w_{jk} are the weights that lead from unit j to the output units. The change in the weight w_{ij} between unit i in the layer below the hidden layer and unit j is Δw_{ij}, which is given by:

$$\Delta w_{ij} = \eta \delta_j o_i.$$

Recalling that the error signal for n4 was 0.125, and that the weight between n3 and n4 is still 0, δ_{n3} for the hidden unit n3 will be:

$$\delta_{n3} = 0.5 \times (1 - 0.5) \times 0.125 \times 0 = 0$$

and the weight changes will be:

$$\Delta w_{n1n3} = 0.1 \times 0 \times 1 = 0,$$

$$\Delta w_{b3n3} = 0.1 \times 0 \times 0 = 0,$$

$$\Delta w_{b3n3} = 0.1 \times 0 \times 1 = 0.$$

All the network weights can then be modified by these amounts according to:

$$w_{jk} \leftarrow w_{jk} + \Delta w_{jk}.$$

If the same pattern is now input to the network again, n4 will be 0.507, which is of course a little closer to the correct answer.

The above method is called the "online" or "continuous" update method because changes are made continuously as each pattern is input. Another method for doing weight changes is to collect all the weight changes for all the patterns in the training set and then do them only once for the whole set instead of once for each pattern. The advantage to this method is that less arithmetic needs to be done, but there is a disadvantage to this because then you may need to make more passes through the training patterns. On large problems this second option is usually better. This second option is called the "batch" or "periodic" update method.

Another important consideration when doing weight changes with the periodic update method is that for some pattern sets like the XOR problem all the weight changes will cancel out if all the weights start out at 0 and the result is that no learning takes place. This can be solved by starting out the weights with small random values in the range from about −1 to +1 rather than starting them with 0. The best range of starting weights to use varies from problem to problem and may be larger or smaller than this range. In practice, back-propagation networks are almost always started out with small random values for the weights no matter how the weight changes are managed because this will speed convergence.

3.4.4 Speeding Up Back-Propagation

With $\eta = 0.1$ and starting each weight at 0 and using 32-bit floating point weights, it requires 25,496 iterations through the pattern set to train the network to within 0.1 of its targets. With $\eta = 0.5$ it requires 3,172 iterations and with $\eta = 1$, it requires 1,381 iterations. All these values are much too large to be acceptable for a problem as small as this one. By employing a certain trick it is almost always possible to make a network converge much faster. The trick consists of using a different Δw, we will call it Δw_{better}, that is equal to the Δw calculated for this weight plus α times the value of the Δw that was obtained the last time the weight was updated. In equation form this is:

$$\Delta w_{better} = \Delta w + \alpha \Delta w_{previouschange}.$$

This second extra term is called a "momentum term" because it keeps the process moving in a consistent direction. Typically, α is about 0.9. Using the best combination of η and α the XOR problem can be solved in under 30 iterations.[3] At this time there is no theory that will give the best values for η and α for a given problem, but after you acquire some experience with different types of problems you can usually choose some reasonable values fairly quickly. For a problem with n patterns, a reasonable guess for η is around $1/n$ or $2/n$.

A very large number of methods, both simple and complex, have been devised to speed up back-propagation. Perhaps two of the best are Rprop [172, 173] and Quickprop [37], both of which vary the learning rates automatically for each weight as training goes on.

3.4.5 Dealing with Local Minima

From time to time a network may get stuck in a local minimum and be unable to learn the desired answers. There are not yet any thoroughly researched methods on how best to kick a back-propagation network out of a local minimum. However, one simple method is to assume that this problem has occurred because weights that are positive are too positive and weights that are negative are too negative. Therefore, one simple solution is to take weights that are positive and decrease them by a random amount while weights that are negative can be increased by a random amount. This method can often take a network out of a local minimum.[4] Another method to escape a local minimum is to simply add one or more hidden units to the network. Ash [4] reports that a method he used for automatically adding one node at a time always found a solution for the problems on which he tried the method.

3.4.6 Using Back-Propagation to Train Hopfield/Boltzman Networks

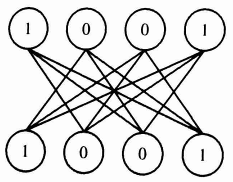


Figure 3.22: Using back-propagation to produce a network with symmetric weights.

It is a simple matter to create a version of back-propagation to produce the kinds of networks with symmetric links that are used in the Hopfield and Boltzman networks. In Figure 3.22 we have a two-layer network with the same number of units in both layers. Given a pattern on the input layer, we will want to produce the same pattern on the output layer.

[3] Various other changes can get this down to around 6. See [239].
[4] Unpublished research by the author.

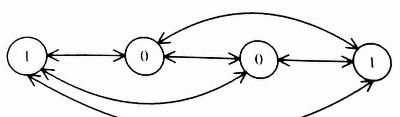


Figure 3.23: The network in Figure 3.22 can be transformed into a Hopfield/ Boltzman type network.

There is one trivial solution to this problem that needs to be eliminated. When input unit 1 is on we can turn output unit 1 on just by having a large weight from input unit 1 to output unit 1; all the weights from the other input units to output unit 1 can be zero. This solution is not acceptable, so to eliminate this possibility, we can either freeze the links from 1 to 1, from 2 to 2, and so forth at zero or eliminate the weights altogether. Now, to enforce the requirement that the weights be symmetric, the weight from input unit 1 to output 2 *will be the same weight* as the weight from input unit 2 to output unit 1, and so on. This procedure also then cuts the number of weights needed in half. Then, whenever we modify the weight from output unit 1 to input unit 2, the weight from output unit 2 to input unit 1 is contained in the same memory location so it is modified at the same time. While we started the problem thinking of it as a two-layer network, this resulting set of interconnections can also be viewed as a Hopfield-style network as shown in Figure 3.23. In training the network, you can use either the traditional back-propagation activation function or a linear one.

3.5 Pattern Recognition and Curve Fitting

It is important to keep in mind that when back-propagation is applied to a pattern recognition problem the algorithm will try to construct a surface that will separate the input data into the correct classes. We will now look at a few examples to illustrate the surfaces that are formed and also note that back-propagation can approximate real-valued functions as well.

3.5.1 Pattern Recognition as Curve Fitting

The XOR problem is a discrete problem because the input and output values are all integers. As such, it does not make sense to try to find XOR of (0.5,0.5) or XOR of any other combination of real-valued inputs. However, if we neglect the fact that such combinations are undefined and go ahead and plot the surface, $z = XOR(x, y)$, for $x = 0$ to 1 and $y = 0$ to 1, we get the surface plotted in Figure 3.24. The network created a valley that runs from (0,0) to (1,1) to solve the problem and the points (0,1) and (1,0) overlook the valley.

In Figure 3.25, we show how a 2-1 network divided up the space for the eight linearly separable points in Figure 3.1. In part (a) of the figure, the line separating the two classes is on a sigmoid-shaped slope at a height of 0.5. Part (b) of the figure shows a cross section of the slope.

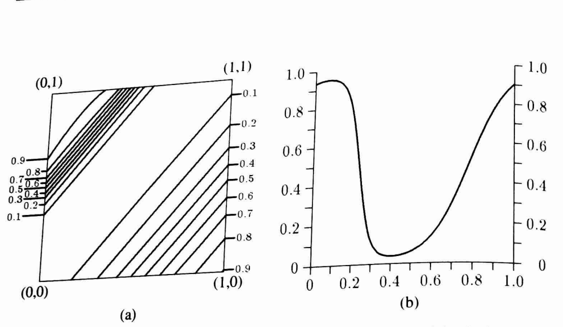


Figure 3.24: On the left in part (a) is a contour plot of the XOR function giving its height as a function of x and y. There is a wide valley between (0,0) and (1,1). On the right in part (b) is a cross section of the surface that runs from (0,1) to (1,0).

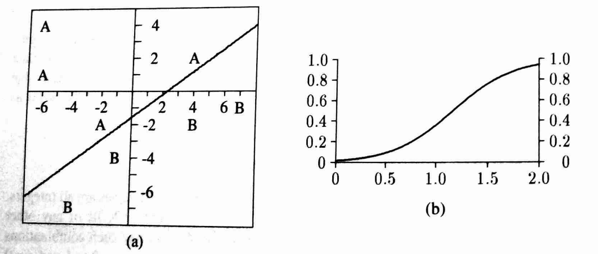


Figure 3.25: A 2-1 back-propagation network produces a sigmoid-shaped surface to separate the eight linearly separable points. The $z = 0.5$ contour line is shown in (a) and part (b) shows a cross section of the surface from $(0, 0)$ to $(2, -2)$.

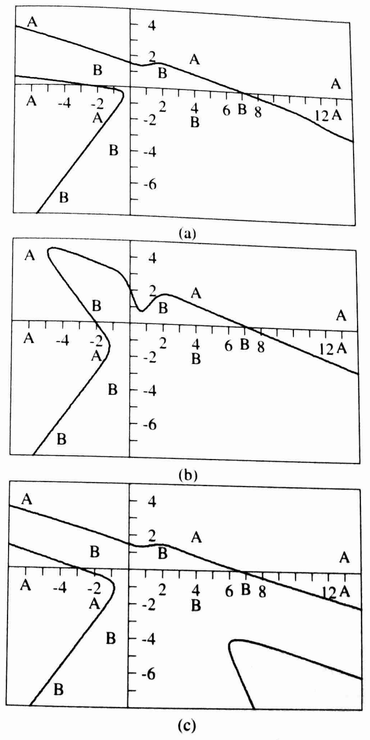


Figure 3.26: These are three different ways three different 2-4-1 back-propagation networks decided to divide up the space using the 12 nonlinearly separable points. If you run a large number of trials, most networks will produce the division shown in part (a), however other divisions also occur as in (b) and (c). Note that in (c), the network made a large area on the lower-right part of class A (less than 0.5), despite the fact that there are no examples of class A nearby.

In Figure 3.26, we show three different ways that different 2-4-1 networks decided to divide the pattern space given the 12 nonlinearly separable points found in Figure 3.4. In all cases the networks tend to see two lines of B points that meet near the center of the space, however, there are major differences between all three solutions. If a very large number of data points was available to train the network, the network is likely to find a mapping that would fit the data very closely.

3.5.2 Approximating Real-Valued Functions

In most instances in this book we will apply back-propagation to problems where the inputs and outputs are 0 or 1, however, it is also possible to use back-propagation for functions where the input and output values can be any real number. For functions with output values beyond the range of the standard sigmoid's range of 0 to 1, the most common solution is to make the output layer activation function linear.

Funahashi [48] has proved that three-layer feed-forward networks can approximate any continuous mapping to any degree of accuracy and Hornik et al. [70] have proven that they can approximate an arbitrary function and its derivatives to any degree of accuracy. Leshno et al. [100] extend the results to show that continuous functions can be approximated if and only if the hidden layer activation function is not a polynomial.

3.5.3 Overfitting

Figure 3.27 shows an example of how backprop can *overfit* a function. The function to be fit is the line, $y = x + 1$ and in the plots this is the straight line shown. The training data consists of seven points that are slightly above or below the ideal straight line and these points are marked with an asterisk. In real world applications it is normal for data points to fall above and below the best line and such data is said to be *noisy*. A test set was made of 301 points along the ideal line.[5] The network chosen was a 1-3-1 network with a linear output unit. After 2,700 iterations of training the fit is quite close to the ideal line, but as training continues the network begins to overfit the data by bending its solution very close to all the points. Notice that while the error continues to go down on the training set, the test set error goes up. Test set error is a much better measure of the performance of a network than the training set error; for serious applications you need a training set and a test set. To minimize overfitting behavior you should have many more training patterns than weights. One way to do this is to use as few hidden layer units as possible. There are other ways to deal with this problem as well.

3.6 Associative Memory and Generalization

An important characteristic of human behavior is that as people gather together a number of cases of some phenomenon or objects, they generalize about them. In the traditional symbol processing approach to artificial intelligence the theory has been that people actually develop explicit rules about the phenomenon or objects. Furthermore, it was generally

[5] Using points along the ideal line instead of more noisy points is unrealistic of course, but easy to do.

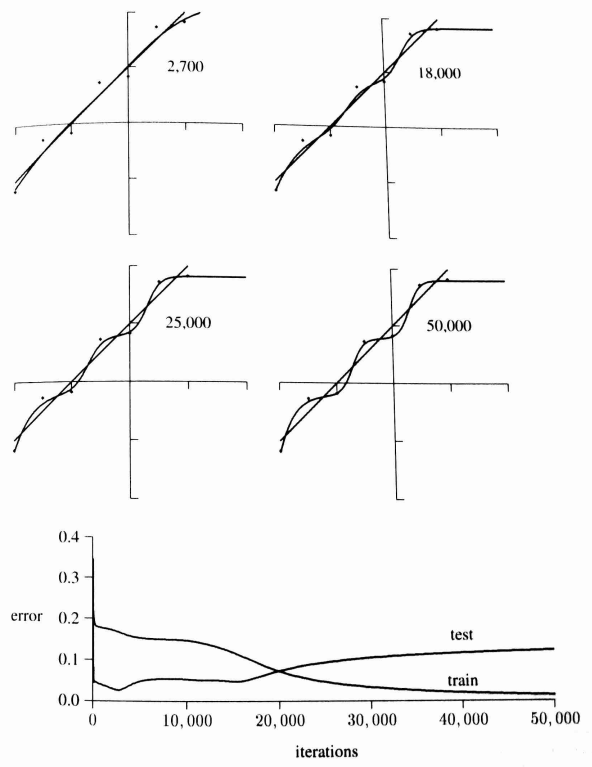


Figure 3.27: The fit for the seven points with a 1-3-1 network at 2,700, 18,000, 25,000, and 50,000 iterations. The graph at the bottom shows the error for the training set and the test set. The minimum on the test set error was at 2,700.

assumed that people then used these rules in something close to a von Neumann style architecture. We noted this in Chapter 1 and the idea will be developed further in the next and later chapters. In this section we will show how neatly neural networks can generalize to give rulelike behavior and do some elementary reasoning without using explicit rules and a von Neumann style architecture. In the neural framework it is the way items are stored in memory that gives rise to generalization and some simple reasoning phenomena. Neural networking researchers believe these methods are more realistic than the traditional symbolic approaches.

3.6.1 Associative Memory

One way to view the XOR back-propagation network introduced in Section 3.4 is that, given some inputs it computes the value of the function, however, another way to view it is as a memory, a device that gives the contents of a memory location when it is given an address (the input pattern). From this perspective it is quite similar to a memory reference operation in a conventional von Neumann style computer, however, when a neural network memory stores and retrieves memories you get some interesting additional effects. Neural networks are a way to implement an *associative memory*. (There are other ways as well.) An associative memory is one that, given a stimulus pattern, produces a response pattern or patterns that have been associated with the stimulus. An associative memory differs from the conventional von Neumann computer memory because an ordered set of stimulus patterns (addresses) is not used and because multiple answers are possible. It is also different in that it can give answers to addresses that are only similar to addresses that it has seen.

```
1010 101 100000000000   01000
1010 010 010000000000   11000
0101 101 001000000000   00001
0100 010 000100000000   00100
1000 101 000010000000   00100
0101 011 000001000000   00001
1010 011 000000100000   11000
0101 010 000000010000   00011
1000 000 000000001000   00100
0100 000 000000000100   00100
1010 100 000000000010   01000
0101 100 000000000001   00001
```

Figure 3.28: The data for an associative memory. The first four bits represent: from Chicago, from
New York, Cubs fan, Mets fan. The next three positions represent whether or not the person is a
Democrat, a Republican, and likes lemonade. The next 12 bits give each person a unique name.
These first 19 bits will be the input pattern (rather like an address). The last 5 bits will be the output
of the network and represent whether or not the person is a Sox fan, a Bears fan, likes tennis, is a
Yankees fan, and is a Jets fan.

To illustrate some properties of an associative memory we consider an example where
the input (address) will be 19 bits. The last 12 bits represent names for 12 different people
and the first 7 bits represent properties of these people. Bit 1 is 1 when the person is from

Chicago, bit 2 is 1 when the person is from New York, bit 3 is 1 when the person is a
Chicago Cubs baseball fan, bit 4 is 1 when the person is a New York Mets baseball fan.
We let bit 5 represent whether or not the person is a Democrat, bit 6 represents whether
or not the person is a Republican, and bit 7 represents whether or not the person likes
lemonade. The output patterns consist of 5 bits. Bit 1 represents that the person in question
is a Chicago White Sox baseball fan, bit 2 represents that the person is a Chicago Bears
football fan, bit 3 represents that the person likes tennis, bit 4 represents that the person
is a New York Yankees baseball fan, and bit 5 represents that the person is a New York
Jets football fan. These inputs and outputs for 12 people are shown in Figure 3.28. The
properties of the people involved show the patterns you would expect: people from Chicago
are never fans of New York teams and New Yorkers are never fans of Chicago teams. Half
the Cubs fans are White Sox fans. Three out of four Mets fans are not Yankees fans. All
Cubs fans are Bears fans. All Mets fans are Jets fans. In addition, Chicagoans and New
Yorkers who are not baseball fans like tennis. Bits 5, 6, and 7 have little correlation with
the last 5 bits, except Republican Cubs fans are also Sox fans. It is easy to train a two-layer
network to remember these patterns.

It is not interesting that a network can learn to recall these facts. The interesting part is what happens when you use input patterns, or addresses, that the network has never seen but which are very similar to patterns it has seen. For instance, give the network a nameless person (all name bits = 0) from Chicago who likes the Cubs:

$$1010\ 000\ 000000000000.$$

We get:

$$0.44\ 0.88\ 0.23\ 0.03\ 0.02.$$

The network effectively compares its input with the similar patterns it has experience with and returns a pattern that is close to what those similar input patterns would have produced. We can take these output values to be rough estimates as to whether or not a Cubs fan from Chicago will like the Sox, Bears, tennis, Yankees, and Jets. The results indicate that there is some chance that the person likes the Sox, a high probability that the person likes the Bears, and little or no chance that the person likes the Yankees, Jets, or tennis. This is a rather pleasant surprise considering all the network did was to store "values" at "addresses." If we use Democrat Cubs fans we get:

$$0.11\ 0.88\ 0.14\ 0.01\ 0.02.$$

If we use Republican Cubs fans we get:

$$0.77\ 0.92\ 0.13\ 0.05\ 0.02.$$

For Cubs fans who like lemonade:

$$0.46\ 0.87\ 0.17\ 0.01\ 0.02.$$

Evidently, from the examples it has seen, the network has "come to the conclusion" that Republican Cubs fans also like the Sox while Democrat Cubs fans do not.

We take a look at the network's response to a nameless Mets fan from New York:

0.01 0.02 0.18 0.26 0.85.

meaning that the person is certainly a Jets fan, just possibly a Yankees fan, but no fan of tennis, the Sox, or the Bears. Also, a nameless New Yorker who does not like the Mets, makes the network think of tennis:

0.03 0.05 0.80 0.11 0.26.

What the back-propagation algorithm did during training was to notice certain regularities between the input and output sequences. When it now sees "from Chicago" and "likes the Cubs," it activates "likes the Bears" and inhibits "likes tennis," and so forth. The network found that being a Republican, Democrat, or lemonade liker had no correlation with being a Bears fan. The network is behaving *as if* it had *explicitly* come to the conclusion that the following rule applies:

IF	the person is a Chicagoan and
	the person is a Cubs fan
THEN	that person is a Bears fan,

while all it was doing was trying to look up a value at an address. We may say that networks like these have the important ability to generalize from their experience and come to know the important characteristics of a class of objects. Notice also, that the network came up with, we might say, fuzzy sorts of rules, like, that the typical Chicago Cubs fan is sometimes a Sox fan and that a New York Mets fan is not usually a Yankees fan.

One problem with the linear pattern classifier network and the nearest neighbor algorithm is that they assume that there are specific places at which to divide up the space between pattern classes. In real life problems the divisions are not always so easy. For instance, one highly qualified loan officer may decide to grant a loan where another equally qualified one will deny it. In pattern recognition terms, there is a region of space where the decision is uncertain and it is unrealistic to find a hard division between two classes. However, back-propagation networks produce smooth surfaces that will minimize the overall error and then the numbers that come out of the network correspond to an estimate of probability. This shows up in the above network's ability to estimate whether or not Cubs fans are Sox fans or that Mets fans are Yankees fans.

It is important to note, however, that back-propagation networks do not always find the generalizations that you expect them to find. For instance given the 12 nonlinearly separable points used in creating Figure 3.26, three different runs of back-propagation produced three different divisions of the space that no human being is likely to produce. One simple safeguard in using networks is to train a number of them on the same data, submit the same unknown patterns to them for classification, and then average the results. This is the analog to collecting a set of opinions from a number of human experts (say, for instance, doctors) and weighing them. For a mathematical analysis of this see the article by Perrone and Cooper [150].

Of course, another interesting aspect of the problem is that networks that *do not* find the generalizations that people would expect may actually find better generalizations than people or provide people with an entirely different perspective on the situation. Remember,

too, that people have often produced incorrect generalizations, such as that the Earth is flat or that the fundamental elements are earth, air, fire, and water. The scientific method of observing data, forming a theory, *and testing as many other cases as possible* is designed to minimize such errors. It cannot be emphasized enough that backprop is not capable of uncovering every sort of regularity that can be found in a set of data. It has very specific limitations. For a discussion of these limitations and some ways they can be overcome see [19].

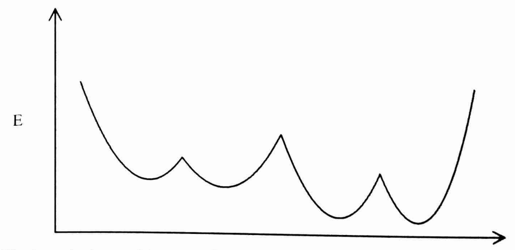


Figure 3.29: A rough picture of the energy landscape near the four Cubs fans. The valley on the left contains the two Democrat Cubs fans and the valley on the right has the two Republican Cubs fans.

In the above sports example we chose to emphasize the memory lookup capability of a network by declaring the input to be an address and the output to be the contents of the address. It is also possible to use back-propagation to produce pattern completion networks for use with the Hopfield and Boltzman recall algorithms. We can put all 24 characteristics of each sports fan on the input layer of a symmetric network and require the same pattern to appear on the output units. After training, we can put part of a pattern on the input units and the network will try to complete the pattern on the output units. A network that simply tries to reproduce its input pattern on the output units is called ***auto-associative***. (A network can be auto-associative without being a Hopfield network.) We can also use the Boltzman machine recall algorithm to complete the pattern. For instance, we may give the recall program the pattern:

1010 000 000000000000 00000

and through random updates of the Boltzman machine relaxation algorithm, the system will settle into a minimum having the characteristics of one of the four Cubs fans. If the cooling is done enough times, the network will recall the characteristics of all four Cubs fans. Thinking in terms of computational energy, the four Cubs fans occupy a valley in the 25-dimensional computational energy landscape. Within that valley is a subvalley for the two Republican Cubs fans and a subvalley for the two Democrat Cubs fans. This is illustrated in Figure 3.29. Each subvalley has two minimums in it. Similar valleys exist for Mets fans and tennis fans.

3.6.2 Local and Distributed Representations

There are two representations for objects that can be used in neural networks. They are local and distributed representations. The term *parallel distributed processing* is derived from the distributed representation. Distributed representations have the important ability to generalize from patterns and we will now look at the differences between local and distributed representations in detail. At first glance, the distributed and local representations appear quite different, yet under certain circumstances the differences are not clear-cut.

A good example of a local representation is the Waltz and Pollack activation network seen in Section 2.4. In that network, different and complex concepts such as hunting, gambling, John, bucks, and deer are each allocated a specific node. By way of contrast, using a distributed representation, we might code the pattern for a deer, not as a single unit, but as a pattern of activity over a whole series of units. For instance the pattern for a deer could be:

000 000 011 000 111 010 101 0.

Each of these 22 units represents a characteristic of animals in general. Some of these features might be color, size, has four legs, has two arms and two legs, has a tail, and so on. If this distributed type of representation is used by people, we would expect there to be many more than 22 such features, including possibly a pattern representing a mental picture of the animal, a pattern representing the sound of the name of the animal, a pattern representing how to speak the name of the animal, and so on. Each feature represents a small part of the animal and in a distributed representation each small feature is called a *microfeature*.

chimpanzees:	111 000 000 111 000 000 000 0
gorillas:	000 111 000 111 000 000 000 0
orangutans:	000 000 111 111 000 000 000 0
dogs:	000 000 000 000 111 111 000 0
cats:	000 000 000 000 111 000 111 0

Figure 3.30: The distributed representations for five animals.

An important aspect of the distributed representation is that it is quite easy to represent many animals by just listing their characteristics in a vector. For instance, we might list the characteristics of chimpanzees, gorillas, orangutans, dogs, and cats as shown in Figure 3.30. If we used this representation in a program it is quite easy to add new animals simply by producing new codes listing their characteristics. This is in marked contrast to what would be necessary if the memory was using a local representation. In that case, for each new animal added, a new node would need to be created and links would need to be created connecting the animal with its characteristics.

If the differences between the two representations were as clear-cut as it sounds from above, all would be well, however, there are situations where the differences are not so obvious. Suppose we have a network as shown in Figure 3.31 with nodes for chimpanzee, gorilla, and orangutan plus nodes for their 22 characteristics as shown in Figure 3.30. The nodes numbered 1 to 22 represent the 22 microfeatures of animals. There are activation and inhibition links between the animal units and the microfeature units and we assume the

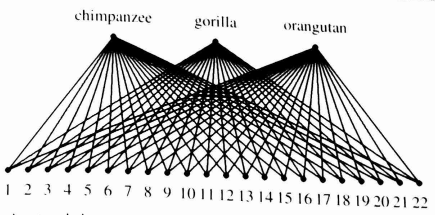


Figure 3.31: A network that seems to use local representations, but if you light up the chimpanzee unit and apply the interactive activation algorithm to the network, a set of feature nodes (1-22) will also come on so 'chimpanzee' ends up looking like a distributed representation after all.

network uses the interactive activation method. If we activate the chimpanzee node then units 1, 2, 3, 10, 11, and 12 will come on giving, in effect, a distributed representation in the network. Thus, distinguishing between local and distributed representations can be a problem at times.

3.6.3 Reasoning within a Network

We can now look at the ability of distributed representations to do some elementary reasoning. We will use the data in Figure 3.30 about chimpanzees, gorillas, orangutans, dogs, and cats and we designate the 22nd bit as indicating whether or not the animal likes onions. The data at the moment shows that none of them do. However, suppose that someone tells us that chimpanzees do in fact like onions. As human beings, we would also suspect that gorillas and orangutans also like onions because these animals are all quite similar. Dogs and cats, however, are very different from apes, so knowing that chimpanzees like onions will not raise our expectations about dogs and cats liking onions by very much, if it raises them any at all. If we later learned that gorillas and orangutans do not like onions we would file that away while still leaving liking onions as a unique characteristic of chimpanzees.

To demonstrate that a network can also do the reasoning process described above we will first train a network using back-propagation to remember the characteristics of the five animals in Figure 3.30. Column 1 of Figure 3.32 shows the activation values for the "likes onions" unit and they are all very close to 0. Now, to have the network learn that chimpanzees do like onions, we retrain this *existing* network of weights with only one pattern rather than with the complete set of five patterns:

111 000 000 111 000 000 000 1

This pattern is the same as we had before for chimpanzees except in this case we have chimpanzees liking onions. The network will use those units that are 1s in the input layer to turn on the "likes onions" unit in the output layer. Column 2 of Figure 3.32 clearly shows that chimpanzees like onions and that orangutans and gorillas register an increased probability that these animals like onions. There is very little indication that dogs and cats like onions.

	"likes onions" after initial training	"likes onions" after learning that chimps like onions	"likes onions" after learning that gorillas and orangutans do not like onions
chimpanzees	0.02	0.96	0.68
gorillas	0.02	0.37	0.08
orangutans	0.02	0.36	0.07
dogs	0.02	0.05	0.04
cats	0.02	0.05	0.04

Figure 3.32: This data shows how a network can so some elementary reasoning. The slight changes for dogs and cats are due to changes in the weights from the bias units.

We can now tell the network that gorillas and orangutans do not like onions by training it on two patterns, one for gorillas and one for orangutans. Column 3 of Figure 3.32 shows the results. The estimates for gorillas and orangutans liking onions has been lowered to below 0.1, while the rating for chimpanzees remains quite high.

The automatic reasoning ability of the above network was made possible by the fact that each animal was represented as a pattern of activity distributed over a number of units. Knowledge about one animal spills over to the other animals to the extent that their active input units overlap. The more the overlap the more the spillover.

The above demonstration made use of another interesting property of this type of network. All the patterns do not necessarily always have to be added to the memory all at once. They can sometimes be added one at a time and up to a certain point they will not get in each other's way. This is possible because when a new pattern is added there are many weights to be changed and therefore each weight only needs to be changed by a little bit to store the new pattern. In addition, the total effects of all the small changes are likely to cancel each other out. For these reasons, new patterns can often be added without destroying the old ones. There are instances, however, when storing one extra pattern *will* destroy the facts in the original network. With only 22 units for this example, the changes to each weight were still fairly significant. However, by adding one fact at a time, eventually a network will reach a point where old patterns get weaker and weaker and finally the old patterns will be lost forever.

The fact that in parallel distributed networks the generalization of knowledge comes automatically just as a consequence of storing and retrieving knowledge from a network has made some researchers think that this is a better model of generalization than the symbolic approach of constructing actual rules. These networks produce rulelike behavior without the expense of producing actual rules.

3.7 Applications of Back-Propagation

To illustrate the usefulness of back-propagation networks we will now look at some of the early applications as well as mention a few experimental systems. In addition to the

applications we mention here, there are very many applications of back-propagation being made to very many types of problems that can be solved by doing a single step of pattern recognition.

3.7.1 Interpreting Sonar Returns

One experimental back-propagation system by Gorman and Sejnowski [52, 53] was designed to classify sonar targets. The two targets used in the experiment were a metal cylinder and a cylindrically shaped rock. Both targets were about five feet long and both were placed on a sandy ocean floor. Sonar returns were collected from the objects at a range of 10 meters and from various angles. Data for each signal was preprocessed in a manner thought to be similar to how human beings hear sound patterns (see [53]) and then it was normalized to fit between 0.0 and 1.0.[6] There were 60 input values. These patterns were used as input to networks with 0, 2, 3, 6, 12, and 24 hidden units. The output layer consisted of two units, one for the rock and one for the cylinder.

	Aspect-Angle Independent Series		Aspect-Angle Dependent Series	
Number of Units in Hidden Layer	Average Performance on Training Sets	Average Performance on Testing Sets	Average Performance on Training Sets	Average Performance on Testing Sets
0	89.4%	77.1%	79.3%	73.1%
2	96.5%	81.9%	96.2%	85.7%
3	98.8%	82.0%	98.1%	87.6%
6	99.7%	83.5%	99.4%	89.3%
12	99.8%	84.7%	99.8%	90.4%
24	99.8%	84.5%	100.0%	89.2%

Figure 3.33: The results of two series of tests done by Sejnowski and Gorman to train networks to distinguish between the sonar returns from a metal cylinder and a cylindrically shaped rock.

Two series of experiments were performed, one set using angle independent data and another using angle dependent data. The results are shown in Figure 3.33. These results were averaged over 130 networks for each number of hidden units. The performance in this series of tests approached 100 percent on the training data and 84.5 percent on testing sets that the program had not seen. Three-layer networks with 12 and 24 units gave the best performance. An analysis by Sejnowski and Gorman found that this relatively poor performance was due to the fact that there were examples of patterns used in the testing set that were not found in the training set. In addition to doing the back-propagation analysis, the authors used another technique to estimate the performance of a nearest neighbor classifier and found it would be 82.7 percent correct on the unknown patterns. In the angle

[6] Backprop networks can accept any real values as input, however, the best results come when the magnitude of the inputs is more or less 1.

dependent series of tests the networks performed near 100 percent on the training set and up to 90.4 percent on the test sets. Again, performance reached a peak at 12 hidden units.

In tests on human subjects, the human subjects tended to perform from slightly better than to slightly worse than the networks but there was less testing done with the human subjects than with the networks.

3.7.2 Reading Text

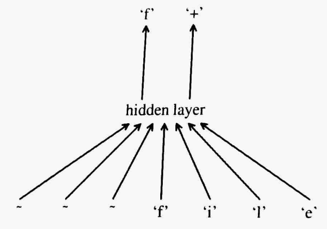


Figure 3.34: The general structure of NETtalk. There are seven letters in the input layer. In this case, the first three letters are blank (˜). The output layer codes for the correct pronunciation of the middle letter of the seven. The left set of output units represents the identity of the phoneme and the right set codes the stress field. In effect, the network has to learn to give the proper response for the input letter 'f' given the fact it is surrounded by three other items on each side.

Another experimental network, this one researched by Sejnowski and Rosenberg, is known as NETtalk [199]. NETtalk consists of a network of processors that learns to read aloud. The program has been interfaced to a speech synthesis system. With the speech synthesis system engaged, the program babbles like an infant to start with, but after 16 hours of practice it learns to read aloud at the level of a 6-year-old child. The general arrangement of NETtalk is shown in Figure 3.34. It is a three-layer network where the input units contain a seven letter portion of a word (called a window) and the output units contain first, a code for the proper pronunciation (the phoneme) for the letter in the middle of the seven input letters and second, a stress field for the middle letter. The stress field takes on the value "1" for primary stress, "2" for secondary stress, "0" for unstressed, "+" for rising, and "-" for falling. Figure 3.34 shows the network looking up the pronunciation for the "f" in "file." The first three characters on the input layer are blank. For patterns selected from the training set, NETtalk produces the correct phoneme 94 percent of the time. For patterns it has not seen before it gets the correct phoneme 78 percent of the time. The number of correctly pronounced words is much less. A system similar to NETtalk was analyzed by Seidenberg and McClelland [198] and they found that it behaved much the same way as people.

Waltz and Stanfill have also attempted this pronunciation task using another method in a system called MBRtalk. MBRtalk is described briefly in Section 7.2. Waltz and Stanfill point out that perfect or near perfect results with the methods employed by NETtalk

and MBRtalk are not possible for several reasons. First, while these systems can learn regularities, there are also many words with irregular pronunciations that a system cannot know about unless it has been specifically told about them. Second, English has borrowed many words from other languages. Unless the system knows the language in which the word originated it cannot guess the correct pronunciation. Waltz and Stanfill cite such examples as targET versus filET, piZZa versus fiZZy, and vILLA versus tortILLA.

In addition to MBRtalk and NETtalk, Wolpert [264] has produced a system simpler than MBRtalk that he says "has a generalization error rate of less than 1/3 of that of NETtalk" and this system is described briefly in Section 7.2. Another system for reading text aloud is DECtalk designed by Digital Equipment Corporation. DECtalk was constructed using conventional programming techniques and it is superior to NETtalk, however, DECtalk required 15 years to develop while NETtalk was developed over a summer.

3.7.3 Speech Recognition

Speech recognition is another important area where back-propagation networks may be useful. Waibel et al. [241, 242, 243, 244] have been working on the problem of training networks to recognize spoken sounds. In one early set of experiments a specialized back-prop network was trained to recognize the sounds, b, d, g, p, t, and k and the network managed to get around 98 percent correct on a test set.

3.7.4 Detecting Bombs

One back-propagation-based system in actual use was commissioned by the Federal Aviation Agency and produced by Science Applications International Corporation (see Shea et al. [205, 206]). The system is designed to detect explosives in suitcases by quickly scanning the suitcases in a normal airport environment. In particular, the object is to detect unusual amounts of nitrogen in the suitcases because most ordinary explosives contain large amounts of nitrogen. The system works by first bombarding the luggage with low energy neurons. The neutrons are absorbed by atoms within a suitcase and the atoms then emit gamma rays at various energy levels. Each type of atom will emit a characteristic set of gamma rays which are measured and the results of the measurements are then submitted to a back-propagation network for analysis.

	Percent Explosives Detected	False Alarms
linear discriminant	98.0	11.6
back-propagation	98.0	7.8

Figure 3.35: The explosive detection rates and false alarm rates for a linear discriminant solution and back-propagation.

As in so many real world applications, it was not possible to get perfect detection. Sometimes a bag with explosives will not be detected and sometimes a bag without explosives will be flagged as containing explosives. The goal in the project was to detect at least

90 percent of the suitcases that contain explosives while minimizing the number of false alarms. A practical aspect of such systems is that you can increase the probability of detection but at the cost of increasing the number of false alarms. In operation, every piece of luggage that is suspect must be examined in more detail and this is very time consuming. There were two major approaches that were tried, a linear discriminant analysis system (a linear pattern classifier) and the back-propagation network. Both proved to be equally good at detecting bags with explosives, but the back-propagation system did much better at eliminating false alarms. Some tests were made at John F. Kennedy airport in October 1989 using actual luggage and luggage with simulated explosives and they gave the results shown in Figure 3.35. These results came from a network with one hidden layer. Some experiments were run with a four-layer network where the probability of a false alarm was decreased by one percentage point, but the training required 14,000 to 20,000 cycles, whereas the three-layer network only needed 2,000 to 4,000 cycles.

3.7.5 Economic Analysis

A number of systems have been designed to do economic analysis ranging from stock market analysis, to commodity futures analysis, to rating bonds, to loan underwriting. Kimoto, Asakawa, Yoda, and Takeoka [83] report producing a stock market prediction system that "showed an excellent profit." On the other hand, White [255] attempted to use networks to predict the daily rates of return on IBM stock and he reported that in the experiments he tried, the networks have failed to uncover any predictable fluctuations. Refenes et al. [168] report that neural networks perform better by an order of magnitude than classical statistical techniques for forecasting within the framework of the arbitrage pricing theory model for stock ranking.

In another example of economic analysis, Collard [21] trained a network on commodity futures data using a year's worth of data. The network was then tested on another nine months of data. Collard reports that, given an initial investment of $1,000, the network would have made a $10,301 profit.

Dutta and Shekhar [207] have experimented with networks to rate bonds. They report that their networks function better than the typical mathematical procedures used by bond raters, but the networks do not always produce the same rating that human bond raters produce. They also found that two-layer networks performed as well as networks with hidden layers. Experiments done by Surkan and Singleton [225] found that three-layer networks performed better than discriminant analysis at rating bonds and four-layer networks were better than three-layer networks.

Smith [215] produced a network to do loan underwriting and he reports that "It is delivering an 18 percent increase in profit for its user, compared with the performance of previous decision technology (a point scoring system based on multivariate discriminant analysis)."

McCann [106] used a recurrent backprop network to predict the highs and lows of the gold market.

3.7.6 Learning to Drive

In a project conducted by Pomerleau, Jochem, and Thorpe [160, 161, 162, 163, 164, 74], back-propagation networks have been trained to drive a car down interstate highways, one and two lane roads, and suburban neighborhood streets. The project is called ALVINN for Autonomous Land Vehicle In a Neural Network and it is used to drive a vanlike vehicle called NAVLAB. Pictures from a video camera on the vehicle are digitized to form a 30 × 32 input matrix. This is fed into a hidden layer with 4 units and then to a 30 unit output layer. The output layer units code for how hard the network should steer to the right or left. The system learns to drive by monitoring the driving of a human being for about three minutes. For each different type of road a different network must be trained and the system must monitor which network is giving the best results on how to drive. It switches between networks as necessary to give the best performance. The higher the degree of confidence the system has, the faster the system can drive. ALVINN has managed to drive up to 70 miles per hour and has averaged 60 miles per hour over a 90 mile stretch of highway. Research is now being done on getting ALVINN to handle more complex driving situations such as intersections, off-ramps, and changing lanes. To do this the researchers are working on dynamically moving the camera as well as narrowing its field of view so it can concentrate on the important details of the scene.[7]

3.7.7 DNA Analysis

Lapedes[8] has trained a back-propagation network to predict whether or not short DNA fragments contain codes to produce proteins. Its accuracy rate is 80 percent rather than the 50 percent obtained with conventional pattern recognition techniques. The network was trained by presenting it with 900 examples of fragments that do code for protein production and 900 examples that do not. Lapedes says that the network appears to have learned some fundamental rules about genetics that may possibly have eluded biologists.

3.8 Additional Perspective

In this chapter we have looked at a few of the most common and useful pattern recognition algorithms. For the most part, these algorithms were chosen because they will be useful in later chapters. In particular, back-propagation can be modified to handle an amazing number of tasks. However, over the years a very large number of pattern recognition algorithms have been proposed and used and it is not possible or useful in an AI book to look at any more of them. Even though back-propagation is only a relatively recent addition there are already an amazing number of variations on back-propagation that have been developed and most of these give better results than backprop.

[7] Some of these details were taken from a Usenet posting in comp.ai.neural-nets by Dean Pomerleau, Message-ID: <753213973/pomerlea@POMERLEA.BOLTZ.CS.CMU.EDU>, Date: Sat, 13 Nov 1993 18:06:00 GMT. More information on ALVINN and related research can be found in the Robotics Institute Technical Report WWW page at http://www.cs.cmu.edu/afs/cs.cmu.edu/project/alv/member/www/navlab_home_page.html.

[8] This report comes from *Science News*, Volume 132, Number 5, August 1, 1987, page 76.

3.9 Exercises

3.1. Without using a program show that a linear pattern classifier cannot do the XOR problem, that is, it cannot put the XOR outputs of +1 in one class and the XOR outputs of 0 in another class. Let the 0 outputs be −1.

3.2. Without using a program determine whether or not the following patterns can be classified correctly using the linear pattern classifier:

input	output
1 0 0	1
0 0 0	−1
0 1 0	1
1 1 1	−1

3.3. Show that if a three or more layer backprop network uses ONLY the linear activation function that it is equivalent to a two-layer network, so that nothing can be gained in terms of pattern recognition ability by having more than two layers if a network uses linear neurons. This is particularly easy to show if you use matrix algebra.

3.4. Program the linear pattern classifier[9] of Section 3.1 and train it to learn the difference between an E and an F using the data shown below. After training, examine the weight vector to see how the program discriminates between an E and an F.

1	1	1	1	1
1	0	0	0	0
1	1	1	1	1
1	0	0	0	0
1	1	1	1	1

1	1	1	1	1
1	0	0	0	0
1	1	1	1	1
1	0	0	0	0
1	0	0	0	0

3.5. In Section 2.1 the letter recognition algorithm looked for four different subpatterns in each of nine regions and then applied a formula to determine the unknown. Now, program the same problem but use the simple Euclidean nearest neighbor algorithm described in Section 3.2. Use at least eight different patterns. Test the program with a number of letters that are distorted from their ideal shapes. If you programmed Exercise 2.1 compare those results with the results you get here.

Another variation you may want to try is to collect a large number of patterns of just two very similar hand-printed letters (for instance, E and F, C and G, P and R, F and P, and so forth). Use the nearest neighbor algorithm and put about half the patterns into the database and use the other half as test cases. You can also repeat this with DSM and the back-propagation algorithm described in Section 3.4.

3.6. Starting with the network in Figure 3.7 and the activation values shown there, update the units in this order: b, c, d, a, b, c, d. Also try this series of updates: a, c, d.

[9] Available online.

3.7. Program the Boltzman machine and Hopfield network relaxation algorithms.[10] Use the following patterns of the letter J and the number 4:

0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	0	0	0	1
1	1	1	1	1

1	0	0	0	1
1	0	0	0	1
1	1	1	1	1
0	0	0	0	1
0	0	0	0	1

Storing the weights in a 25 × 25 matrix will be convenient or C programmers can use malloc to create arbitrary size arrays. Run simulations to determine how often the following patterns end up in the global minimum as a function of how long the network stays at each temperature. In addition, vary the threshold value to see how it affects your results.

0	0	0	0	0
0	0	0	0	1
0	0	0	1	1
0	0	0	0	1
0	0	0	0	0

0	0	0	0	0
0	0	0	0	1
0	0	0	0	1
1	0	0	0	1
0	0	0	0	0

3.8. Philosophical Question: Is the concept of finding and getting stuck in a local minimum applicable to other aspects of the world? For instance, do people, societies, and nations get stuck in local minima so that they become incapable of change, even change for the better? Could alcoholism be a local minimum? Are these valid examples and are there other examples? If you think these are valid examples, then how do you get people, societies, and nations out of a local minimum?

3.9. Use backprop to solve the XOR problem, then vary at least one of the weights to see how the error changes. Plot the error for a wide range of values around the minimum value found by the network.

3.10. With the XOR problem, vary η and α and the range of the initial weights and see how few iterations you need to solve the problem. To get reliable estimates, average at least 10 or better still, 25 cases for each parameter setting. If you are more ambitious, do the same for as many different variations of the backprop algorithm as you have available. (Doing a large number of simulations will take some time.)

3.11. Given the following set of weights for the XOR network shown in Figure 3.14, sketch the surface the network found to solve the problem:

```
 7.1252470016   * from n1 to n3
 7.1284189224   * from n2 to n3
-2.7686207294   * n3 bias weight
-4.9701967239   * from n1 to n4
```

[10] Also available online.

```
-4.9693980217   * from n2 to n4
10.9574527740   * from n3 to n4
-3.3010675907   * n4 bias weight
```

3.12. Give the assumptions the author made in drawing the line separating the two classes in Figure 3.4 that the back-propagation networks did not make in Figure 3.21.

3.13. Within 6-bit binary integers there are eight palindromes, bit strings that read the same from left to right and right to left. Use back-propagation with two units in the hidden layer to create a network to recognize the 6-bit palindromes. Explain how the network works. You will probably need to train the network on all 64 possible 6-bit patterns. (Also see the next exercise.)

3.14. Neural networks do not always generalize correctly. For example, the data below contains the eight 6-bit palindromes plus some nonpalindromes. Use the patterns and the initial weights shown below and train a back-propagation network with $\eta = 0.5$ and $\alpha = 0.0$. (If you use the integer-based version of the software available with the text it should take exactly 151 iterations to train the patterns to within 0.1 of their targets and the network will not generalize correctly for all 64 patterns. This will happen most of the time for other parameters and initial weights as well.) Test the resulting network on nonpalindromes and see how it classifies them. Try to figure out how the network does classify patterns.

```
m 6 2 1       * use a 6-2-1 network
A dd up       * use the differential step size derivative term and
              * periodic updates
e 0.5         * eta
a 0.0         * alpha
n 15          * the 15 patterns
0 0 0 0 0 0     1
0 1 0 0 1 0     1
0 0 1 0 0 1     0
1 0 1 1 0 1     1
1 0 0 0 1 1     0
0 1 0 1 0 1     0
1 1 1 1 1 1     1
0 0 1 0 0 0     0
0 0 1 1 0 0     1
1 0 0 0 0 1     1
1 1 0 0 0 1     0
1 1 0 0 1 1     1
0 0 0 1 1 1     0
1 1 0 0 0 1     0
0 1 1 1 1 0     1

R             * restore the file of weights
r 152 152     * run 152 iterations (stopping after 151 updates)
```

Here are the weights to start the program with:

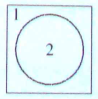
```
0r file = pal
```

```
 0.1494140625
-0.3457031250
 0.0507812500
 0.4843750000
-0.1660156250
-0.3867187500
 0.3945312500
-0.0048828125
-0.3603515625
 0.1132812500
-0.2812500000
-0.4521484375
 0.2226562500
 0.2783203125
 0.2304687500
-0.1962890625
 0.0683593750
```

3.15. Train a back-propagation network to learn $sin(x)$.

3.16. Store the data for the 12 sports fans in Section 3.6 in a Hopfield network matrix and save these weights. You can either use the simple Hopfield matrix algorithm given in the text or use the back-propagation software that is available with the text (or any such software). With the back-propagation software, reasonable learning parameters are $\eta = 0.25$ and $\alpha = 0.5$. A threshold value of 3 will work well whichever method you use. Input the weights to a Boltzman machine simulator and give the simulator the pattern that represents a person from Chicago who likes the Cubs and have the program do several dozen or so relaxations. The purpose of this is to have the program complete the Cubs fan pattern by supplying the rest of the data for each Cubs fan. Because the updates are done at random, sometimes the process will "remember" one Cubs fan and sometimes it will remember another. Note how often each person comes up. Explain why some names come up more often than others. If people also remember things using a Boltzman machine algorithm then does your explanation have any implications for human memory?

3.17. Suppose we have a simple two-dimensional pattern classification problem with two classes divided up like so:



If the radius of the circle is 1 and the length of the side of the square is approximately 2.5, then the areas for both classes 1 and 2 are approximately equal. With this configuration it is easy to generate points at random and assign them to one class or another. Train 2-n-2 backprop networks with 50, 100, and 1,000 training points and test the network with 1,000 points. Vary n as well as the algorithm and algorithm parameters to get very close convergence for the training set. While training, test the network at regular intervals to see how well the network is doing. Repeat the training a number of times for each parameter setting to get a good average value for how long the training takes. Graph the results. (This exercise can take a lot of human and computer time.)

3.18. Do Exercise 3.17 with the same training and testing points but use the nearest neighbor algorithm to classify patterns. Compare your results with those from back-propagation. (This will go much faster than Exercise 3.17.)

3.19. Do Exercise 3.17 with the same training and testing points but use the Decision Surface Mapping (DSM) algorithm. Compare your results with those from back-propagation and the simple nearest neighbor approach. (This, too, will go much faster than Exercise 3.17.)

3.20. If you have some real world data of any sort, such as weather data, stock market data, sports data, or scientific data, apply back-propagation and/or DSM and/or the nearest neighbor algorithm to see how well the algorithm can predict answers. A lot of real world data is available via the Internet including the Gorman and Sejnowski sonar data.[11] For other real world data you can look, for instance, in the directory, pub/machine-learning-database on the system ics.uci.edu and the comp.ai.neural-nets FAQ.

[11] The sonar data comes with the backprop software found at http://www.mcs.com/~drt/svbp.html.

Chapter 4

Rule-Based Methods

4.1 Introduction

In this chapter we will look at some standard symbol processing techniques, in particular those techniques associated with rule processing. In order to do this we will first look at some elementary Prolog so that it can be used as a notation both in this chapter and in later chapters for symbol processing algorithms. Prolog was chosen for this task rather than the also very popular symbol processing language, Lisp, because Prolog has built-in pattern matching capabilities that Lisp does not have. Prolog gets much of its power by using rules so the language itself is an illustration of rule processing.

In this chapter we will look at examples of a type of AI program known as an *expert system*. An expert system is an AI program that is capable of doing the work of a highly skilled human expert. The programs mentioned in Section 3.7 that rate bonds or analyze sonar echos would be considered expert systems, while a program that can recognize letters of the alphabet would not normally be considered an expert system since any human being can recognize letters of the alphabet. The classification of expert/nonexpert system is made this way even if the internal operation of the letter recognizer and the expert system is the same. A number of famous rule-based expert systems will be described. A cautionary note needs to be kept in mind regarding expert systems. It is that they are rarely as good as genuine human experts. The most famous criticism of the abilities of expert systems comes from Dreyfus and Dreyfus [28] where they list instances of expert systems that are not as good as the best human experts. They maintain that because expert systems are not as good as the best human experts they should not be called *expert* systems at all, but only *competent* systems. Furthermore, they argue that real human experts do not use rules.

4.2 Some Elementary Prolog

Logic has been one of the most appealing approaches used to try to produce artificial intelligence. One of the reasons for this is that people are often viewed as being logical creatures. Another reason is that logic is the means for doing mathematical proofs and many computer scientists are former mathematicians. A third reason is that it is very easy to program computers to do simple examples of logic. The most commonly used form of logic is *predicate calculus*. Predicate calculus statements use a formal functionlike notation to give facts (statements) about a problem. Working with these statements, you can prove setting to get a good average value for how long the training takes. Graph the results. (This exercise can take a lot of human and computer time.)

3.18. Do Exercise 3.17 with the same training and testing points but use the nearest neighbor algorithm to classify patterns. Compare your results with those from back-propagation. (This will go much faster than Exercise 3.17.)

3.19. Do Exercise 3.17 with the same training and testing points but use the Decision Surface Mapping (DSM) algorithm. Compare your results with those from back-propagation and the simple nearest neighbor approach. (This, too, will go much faster than Exercise 3.17.)

3.20. If you have some real world data of any sort, such as weather data, stock market data, sports data, or scientific data, apply back-propagation and/or DSM and/or the nearest neighbor algorithm to see how well the algorithm can predict answers. A lot of real world data is available via the Internet including the Gorman and Sejnowski sonar data.[11] For other real world data you can look, for instance, in the directory, pub/machine-learning-database on the system ics.uci.edu and the comp.ai.neural-nets FAQ.

[11] The sonar data comes with the backprop software found at http://www.mcs.com/~drt/svbp.html.

Chapter 4

Rule-Based Methods

4.1 Introduction

In this chapter we will look at some standard symbol processing techniques, in particular those techniques associated with rule processing. In order to do this we will first look at some elementary Prolog so that it can be used as a notation both in this chapter and in later chapters for symbol processing algorithms. Prolog was chosen for this task rather than the also very popular symbol processing language, Lisp, because Prolog has built-in pattern matching capabilities that Lisp does not have. Prolog gets much of its power by using rules so the language itself is an illustration of rule processing.

In this chapter we will look at examples of a type of AI program known as an *expert system*. An expert system is an AI program that is capable of doing the work of a highly skilled human expert. The programs mentioned in Section 3.7 that rate bonds or analyze sonar echos would be considered expert systems, while a program that can recognize letters of the alphabet would not normally be considered an expert system since any human being can recognize letters of the alphabet. The classification of expert/nonexpert system is made this way even if the internal operation of the letter recognizer and the expert system is the same. A number of famous rule-based expert systems will be described. A cautionary note needs to be kept in mind regarding expert systems. It is that they are rarely as good as genuine human experts. The most famous criticism of the abilities of expert systems comes from Dreyfus and Dreyfus [28] where they list instances of expert systems that are not as good as the best human experts. They maintain that because expert systems are not as good as the best human experts they should not be called *expert* systems at all, but only *competent* systems. Furthermore, they argue that real human experts do not use rules.

4.2 Some Elementary Prolog

Logic has been one of the most appealing approaches used to try to produce artificial intelligence. One of the reasons for this is that people are often viewed as being logical creatures. Another reason is that logic is the means for doing mathematical proofs and many computer scientists are former mathematicians. A third reason is that it is very easy to program computers to do simple examples of logic. The most commonly used form of logic is *predicate calculus*. Predicate calculus statements use a formal functionlike notation to give facts (statements) about a problem. Working with these statements, you can prove the truth or falsity of other statements. Statements that you try to prove correct are often called theorems and the calculating of new results is therefore often called *theorem proving*. It is also called *automated reasoning*. The greatest theoretical success in this area of automated reasoning is a proof technique known as Resolution. The next chapter considers the more general case of Resolution. Prolog is a programming language that uses only a subset of the Resolution technique, it basically consists of processing simple rules of the form:

```
if A and B then C
```

and so it is quite easy to understand a Prolog program without studying Resolution.

4.2.1 Stating Facts

Prolog is a programming language very much unlike conventional languages such as Pascal and C. One of the major differences is that it has few of the traditional control structures found in conventional languages. For another, the usual method of using Prolog is interactive. Statements in Prolog represent facts or rules that are true. Here is a series of facts, stated in Prolog:

```
/* 1 */   likes(matt,mets).
/* 2 */   likes(carol,cubs).
/* 3 */   likes(bob,cubs).
/* 4 */   likes(bob,bears).
/* 5 */   likes(mary,mets).
/* 6 */   likes(mary,yankees).
/* 7 */   likes(nancy,lemonade).
```

The numbers between /* and */ are comments and are not required. These statements are quite English-like and they simply look like sentences where the verb has been removed from the middle of the sentence and placed in the front. All statements and rules must end with a period. We define the first statement to mean "Matt likes the Mets." Usually, most people will translate English to Prolog this way by simply moving the verb to the front of the statement, however, this is not always done, so a Prolog programmer might define the first statement to mean "The Mets like Matt."

4.2.2 Syntax

Prolog programs consist of *terms*. A term is either a *constant*, *variable*, or a *structure*. Constants are *atoms* or *numbers*. Numbers are integers or reals. Atoms consist of any sequence of characters enclosed in single quotes (') such as these atoms:

'St. Louis Cardinals'
'095'
'+-+'

or an atom can consist of a sequence that begins with a lowercase letter followed by letters, digits, and the underscore character (_) such as these atoms:

st_louis_cardinals
stl
x99

or an atom consists of a sequence of the special characters:

+ - * / \ ^ > < = ' ~ : . ? @ # $ &

such as:

->
:-
?-

Variables are another type of term and they start with an uppercase character or an underscore followed by letters, digits, and underscores such as the following:

X
Team
_abc
St_Louis
_

The variable consisting of just an underscore by itself (_) is called the *anonymous variable* and it is reserved for a special use.

The final type of term is a structure, such as in the facts we have already seen. In the term:

```
likes(matt,mets)
```

`likes` is called the *functor*, and `matt` and `mets` are its components. When a functor is used to structure data it is called a functor (an example will come up later), however, when a functor is used to express facts or rules it is usually called a *predicate*.

4.2.3 Asking Questions

The set of facts, or clauses, 1 through 7 above might be contained in a file and be read by a Prolog command, or the user might type these facts directly into the Prolog system. With this set of facts in memory, we can go on to ask questions such as the following, where the `?-` at the beginning of the line represents a prompt for a question that is printed by the system:

```
?- likes(nancy,lemonade).
```

In this request, we have asked the system if Nancy likes lemonade. Prolog attempts to answer this question by simply looking through its database of facts for this particular fact. It starts at the top of the database and tries to match this one question against the set of facts in the database one after another until it either finds the fact, or finds the end of the database. In this case, Prolog responds:

```
yes
```

If the question had been:

```
?- likes(nancy,mets).
```

Prolog would respond with:

```
no
```

This "no" means that, *given the facts that Prolog has available to it, it cannot prove that Nancy likes the Mets. This is different from proving that Nancy does not like the Mets.*

Here is a slightly more complicated question that we can ask the system:

```
?- likes(X,cubs).
```

This question means: "Is there any person, X, who likes the Cubs? If so, report the name of that person." Initially, the variable, X, has no value and is said to be an *uninstantiated* variable. In the pattern matching process an uninstantiated variable can match anything. This request is also a simple pattern matching problem. Prolog starts at the top of the database and looks to see if it can match this pattern with anything. The first fact fails to match. The second fact does match the pattern. The variable, X, now becomes an *instantiated* variable and has the value, "carol" and Prolog prints out:

```
X = carol
```

The interpreter leaves the cursor at the end of this line and waits for the user to respond. If the user types a carriage return, Prolog assumes the user is happy with this one response and gives the prompt for another question. If the user types a ";" at this point, Prolog goes further on into the database looking for another possible solution. We type the ";" and Prolog continues searching from where it left off and X loses the value "carol" and again becomes an uninstantiated variable. The third item matches the pattern so Prolog prints out:

```
X = bob
```

Prolog again waits for a response from the user. A carriage return would end this search for solutions. A ";" will cause Prolog to search further. If we type the latter, it cannot find any more matches so it reports, "no" and prompts the user for another question. As a special case, the anonymous variable matches anything but it is never instantiated to any value. So, to ask if anybody likes the Mets without getting back the name of the person, use:

```
?- likes(_,mets).
```

It is also possible to ask the question, "What does Mary like?" by saying:

```
?- likes(mary,X).
```

and to ask "Who likes what?" by saying:

```
?- likes(X,Y).
```

In this case, `likes(X,Y)` will end up matching every item in the database.

Prolog is also capable of answering questions such as: "Is there anyone who likes the Cubs and likes the Bears?" This is stated in Prolog as:

```
?- likes(X,cubs),likes(X,bears).
```

The "," between the two clauses is read as 'and.' Again, in this case, Prolog begins searching the database from the top trying to match the first clause in this question. For the time being, Prolog leaves the second clause alone. Searching through the database, Prolog finds: "likes(carol,cubs)." Having found this, the variable, X, in the statement is set to "carol." The second part of the problem, therefore, becomes: "likes(carol,bears)." To do an orderly search of all the possibilities, Prolog marks the clause, "likes(carol,cubs)," as being as far as it has searched through the database so far in an effort to satisfy the first clause. To keep this straight ourselves, we mark this clause with a '1' as follows:

```
/* 1 */   likes(matt,mets).
/* 2 */   likes(carol,cubs).          1
/* 3 */   likes(bob,cubs).
/* 4 */   likes(bob,bears).
/* 5 */   likes(mary,mets).
/* 6 */   likes(mary,yankees).
/* 7 */   likes(nancy,lemonade).
```

Prolog now works on satisfying the second clause by starting at the top of the database once again. Effectively, Prolog goes off on another call of its pattern matching procedure. Prolog will now search through the database and try to find "likes(carol,bears)." This will fail and so Prolog will return to its first call of its pattern matching procedure and continue trying to match, "likes(X,cubs)," beginning just after the clause we marked with a '1.' The variable, X, is no longer instantiated to the value, "carol." Prolog now finds that "likes(bob,cubs)" matches "likes(X,cubs)" so now X will be instantiated to "bob." Prolog will mark this place in the database, again with a '1' like so:

```
/* 1 */   likes(matt,mets).
/* 2 */   likes(carol,cubs).
/* 3 */   likes(bob,cubs).            1
/* 4 */   likes(bob,bears).
/* 5 */   likes(mary,mets).
/* 6 */   likes(mary,yankees).
/* 7 */   likes(nancy,lemonade).
```

Prolog now goes off on another call of its pattern matching procedure and its problem is to try to find out if "likes(bob,bears)" is true. Prolog quickly finds that it is true and since there are no more clauses in the question to check on, Prolog reports:

```
X = bob
```

and waits for a response from the user. Again, typing a carriage return will end this search process and typing a ";" will continue it. Typing in a ";" is effectively like telling Prolog its answer is a failure and it should continue looking for another answer. We type the ";" and Prolog continues searching on from the fourth clause in the second call of the matching routine. It fails to find another way to match "likes(bob,bears)" and so it returns to the first call of its pattern matching routine. Prolog resumes searching for another Cubs fan starting after the clause marked with the '1.' This fails, and Prolog then reports "no" and asks for another question.

We now want to consider briefly what happens if we ask the longer question:

```
?- likes(X,cubs),likes(X,bears),likes(X,lemonade).
```

That is, is there anybody who likes the Cubs, likes the Bears, and likes lemonade? We have already done part of this problem, so instead of starting at the beginning, we start at the point where the first two clauses have been matched and it is time to go on to the third clause. At this point we will have placed a '1' at the clause "likes(bob,cubs)." Also the second clause in the question has matched the fourth clause in the database, so we mark this fourth clause with a '2':

```
/* 1 */   likes(matt,mets).
/* 2 */   likes(carol,cubs).
/* 3 */    likes(bob,cubs).              1
/* 4 */    likes(bob,bears).               2
/* 5 */    likes(mary,mets).
/* 6 */    likes(mary,yankees).
/* 7 */    likes(nancy,lemonade).
```

We can now go on to try to look at the third clause, "likes(bob,lemonade)." This search will be happening in the third call of the pattern matching routine. It will, of course, fail and this failure causes Prolog to return to the second call of the pattern matching routine and resume searching the database just after the clause we labeled '2.' This part of the search fails and Prolog returns to the first call of its pattern matching routine, restarting its search at statement 4. Ultimately, this searching fails after more attempts and Prolog reports, "no."

4.2.4 Rules

Prolog can not only handle facts, it can also handle rules. Rules are counted as clauses as well. If we want to write the rule that "All Yankee fans like lemonade," it is:

```
/* 8 */    likes(X,lemonade) :- likes(X,yankees).
```

This rule can be read as: "If X likes the Yankees, then X likes lemonade" or "X likes lemonade if X likes the Yankees." The conditions on a rule are called the *antecedents* and the conclusion that would then be true is the *consequent*. Using the first seven facts and with this rule placed at the end of the database, we now ask the question:

```
?- likes(Y,lemonade).
```

Prolog starts at the top of the database trying to match this pattern. It first finds that Nancy likes lemonade and prints:

```
Y = nancy
```

and waits for a response from the user. We type a ";" and Prolog goes on to the rest of the database. Prolog will find that the left-hand-side of the rule matches the pattern it is looking for. The two, still uninstantiated variables, X and Y are now said to *share*. When one of them acquires a value, the other one acquires the same value. The problem of figuring out whether or not a person likes lemonade is now the problem of trying to find out if that person is a Yankees fan. We mark our place in the database:

```
/* 1 */    likes(matt,mets).
/* 2 */    likes(carol,cubs).
/* 3 */    likes(bob,cubs).
/* 4 */    likes(bob,bears)
/* 5 */    likes(mary,mets).
/* 6 */    likes(mary,yankees).
/* 7 */    likes(nancy,lemonade).
/* 8 */    likes(X,lemonade) :- likes(X,yankees).    1
```

and start the pattern matching routine at the top of the list. It will be looking for the pattern "likes(X,yankees)." It will find that Mary likes the Yankees, and so in answer to the question of "likes(Y,lemonade)," it will print out:

```
Y = mary
```

and wait for a response from the user.

Keeping variables straight when processing Prolog rules can be difficult to do so it is probably a good idea to look at how Prolog handles variables internally. For instance, when we asked the question:

```
?- likes(Y,lemonade).
```

internally, it was made into something like this:

```
likes(_1,lemonade)
```

where the Y has been replaced with an interpreter generated variable, _1. When Prolog was searching through its database and it came upon the rule:

```
likes(X,lemonade) :- likes(X,yankees),
```

it created a variable, _2, that stands for X in this rule. The pattern for the rule is now:

```
likes(_2,lemonade) :- likes(_2,yankees).
```

Therefore, even if our original question had been phrased as:

```
?- likes(X,lemonade).
```

it would still have been translated to:

```
likes(_1,lemonade)
```

so Prolog would recognize the X in the question as being a different X than the X in the rule. This means that each variable in a rule is confined to that rule, the same as local variables are confined to the procedures in which they are defined in languages such as Pascal. There are, therefore, no global variables in Prolog. This method of creating new variables for each "call of a rule" also occurs in rules that call themselves recursively.

4.2.5 Recursion

We now turn to some recursive programming in Prolog. The standard example of recursive programming is, of course, the factorial function. Below we show the definition of factorial in Prolog:

```
/* 1 */    factorial(0,1).
/* 2 */    factorial(N,M) :- X is N - 1,
                             factorial(X,Y),
                             M is N * Y.
```

Line 1 says that "factorial of 0 is 1" and line 2 says that "N factorial is M, where M is defined by the clauses on the right hand side of the rule." The notation here for doing arithmetic is a bit unusual. The expression "N – 1" can be written in this infix notation for the convenience of the user, however, internally, Prolog considers it to be:

```
-(N,1),
```

so "N – 1" is just another structure and "–" is called the more general term, functor, rather than a predicate. *Writing N – 1 does not instruct Prolog to do any arithmetic*, rather, it is the 'is' operator that forces arithmetic to be done on the structure "N – 1." The execution of the factorial program proceeds in a straightforward way. If we want Prolog to find 3 factorial, we enter:

```
?- fact(3,A),
```

When Prolog tries to match this against line 1, it fails because it cannot match 3 with 0. It is possible to match 3 with N in the second line of the program. Prolog then sets X to 2 and does the call, fact(2,Y). This recursive call sets up other recursive calls, but eventually, in this current call, Y is set to 2, M becomes 6, and Prolog goes on to report:

```
A = 6
```

From time to time, people have said that the statements in a Prolog program can go into the database in any order. Most of the time that is true, however, recursive definitions like the one for factorial must have their statements go into the database in the correct order or infinite recursion will result.

4.2.6 List Processing

We can now look at some list processing in Prolog. One of the really important features of symbol processing is its reliance on processing lists of symbols. In general purpose languages like Pascal, lists of items are often implemented using arrays and accessed by specifying subscripts. In Lisp and Prolog, arrays have not always been implemented and the standard method of implementing lists is by using linked lists. In Prolog the notation for a list containing one symbol, a, is:

```
[a]
```

and it is shorthand for the structure, .(a,[ ]), where . is a functor called dot and [ ] is the *empty list*. The notation for the list containing the three symbols, a, b, and c is:

```
[a, b, c]
```

which is a shorthand for the structure .(a,.(b,.(c,[ ]))). A diagram of how these lists are stored in memory is shown in Figure 4.1. It is also possible to have lists within lists in Prolog, thereby producing trees. An example of this is:

```
[[a, b], [c, d]]
```

Here, the first item in the list is: [a,b], while the rest of the list is: [[c,d]]. This list is diagrammed in Figure 4.2.

If we find it necessary to work with the first symbol in a list, we would call up a predicate that would try to match the list against the pattern:

```
[First | Rest]
```

The vertical bar, "|" can be viewed as an operator that causes the pattern matcher to try to split a list into a first element on the left and the whole rest of the list on the right. Thus, in matching this pattern against [a, b, c], First would become instantiated to the value, a, while Rest would be instantiated to the remainder of the list, or: [b, c]. When we finish

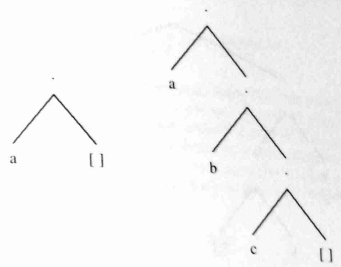


Figure 4.1: The internal tree structure of the lists [a] and [a,b,c]. The notation [a] is shorthand for the structure .(a,[ ]), and [a,b,c] is shorthand for the structure .(a,.(b,.(c,[ ]))), where . is a functor and [ ] is the empty list.

whatever processing needs to be done with a and we want to move on to the next element in the list, the standard means of doing so is to recursively pass this list, [b, c], on to the same predicate. This new call of the predicate again splits the first element off this shorter list. Splitting one element at a time off the list, we eventually encounter the empty list.

As an example of this recursive processing of lists, we will look at a predicate, member, that determines whether or not a particular constant is a member of a particular list. For instance, to ask if the constant, c, is a member of the list, [a,b,c], we will write:

```
?- member(c, [a,b,c]).
```

Member will be defined like so:

```
member(X, [First | Rest]) :- X = First.
member(X, [First | Rest]) :- member(X,Rest).
```

The first line can be translated to English as, "X is a member of the list, [First|Rest], if X and First are equal" or "X is a member of the list [First|Rest], if X is the first symbol in the list." The second line can be translated as: "X is a member of the list, [First|Rest] if X is a member of the list, Rest." In working the problem, member(c,[a,b,c]), Prolog tries to start with the first definition of member. It can match X with c so X is instantiated to c. It then moves on to try to match the pattern in the second argument of member with the pattern:

```
[First | Rest]
```

This is easily done by setting First = a and Rest = [b,c]. Prolog moves on to make the test, X = First and fails since c is not the same as a. The whole statement fails and Prolog moves on to the second definition for member. Again, X = c, First = a, and Rest = [b,c], but now, the test to perform is a recursive call of member. In this call, we will be asking if c is a member of the list, [b,c]:

```
member(c, [b, c]).
```

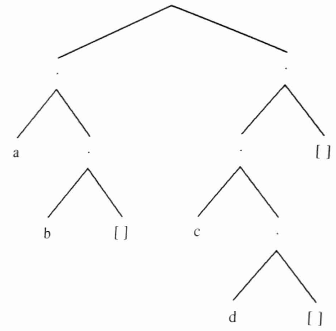


Figure 4.2: The tree structure of the list, [[a,b],[c,d]].

Prolog starts at the top of the list of definitions for member and finds that the first one fails because c is not equal to b. In the second statement, we find X = c, First = b, and Rest = [c]. We do another recursive call of member and now we will have X = c, First = c, and Rest = [ ]. This match succeeds and so ultimately, Prolog reports: yes.

In case we had asked the question:

```
?- member(d, [a, b, c])
```

Prolog will behave as above but it would make another recursive call to member, this time the call will be:

```
member(d, [])
```

This call will fail because when Prolog attempts to pattern match the empty list against [First|Rest], it finds it cannot split the empty list into a "first part" and a "rest part."

For another example of list processing, we consider the problem of looking at a list of integers and producing a list of all the positive numbers in it. For instance, if we name the predicate that creates the list of positive numbers, pos, and we give Prolog the question:

```
?- pos([-1, 44, 97, -300, 10], X)
```

we want Prolog to respond:

```
X = [44, 97, 10]
```

This is easily done with the following definition for pos:

```
/* 1 */  pos([],[]).
/* 2 */  pos([A|B],[A|C]) :- A > 0, pos(B,C).
/* 3 */  pos([A|B],C) :- pos(B,C).
```

Line 1 says that the list of positive numbers in an empty list is the empty list. Line 2 says that if you split the list of integers into a first element, A, and the rest of the list is B, and A is greater than 0, then the list of positive numbers in the list [A|B] is the list with A at the front followed by C ([A|C]), where C is the list of positive numbers found in B. If this second line should fail, the third line applies, and it says that the list of positive numbers in the list [A|B] will just be the list, C, where C is the list of positive numbers in B.

```
1 start:  pos([-1,44,97,-300,10],    X )

2 rule 3: pos([-1|44,97,-300,10], _C1 ) :- pos([44,97,-300,10],_C1).

3 rule 2: pos([44|97,-300,10],[44|_C2]) :- pos([97,-300,10],    _C2).

4 rule 2: pos([97|-300,10],    [97|_C3]) :- pos([-300,10],      _C3).

5 rule 3: pos([-300|10],            _C4 ) :- pos([10],          _C4).

6 rule 2: pos([10 | ],              _C5 ) :- pos([],            _C5).

7 rule 1: pos([],                    [] ))
```

Figure 4.3: A trace of how Prolog interprets the call: pos([-1,44,97,-300,10],X).

The English description of pos is quite neat, but to get a better idea of what is happening in the process we will look at a handmade trace of the process. An actual trace provided by a Prolog interpreter will probably not look like this. Also, the use of variables in this description is not quite the way a Prolog interpreter would go about creating and naming them. The sequence of steps is shown in Figure 4.3. In this description line 1 is the initial problem. Prolog tries to find a rule to apply. It finds that rules 1 and 2 fail. At line 2 in the figure, Prolog examines rule 3 and finds a way to break up the initial list. The "_C1" term is meant to represent the uninstantiated variable named C, in rule 3. Each recursive call will have a C variable with a different number. Still in line 2, rule 3 can be satisfied if its right-hand-side can be satisfied. The right-hand-side consists of the problem:

```
pos([44,97,-300,10],_C1).
```

This is passed to Prolog and in line 3 of the figure, Prolog comes upon rule 2. It creates another uninstantiated variable, "_C2" to represent the instance of C in this rule. Notice how in this case the second argument will be: [A|C] or [44 | _C2]. What this notation says is that the second argument will become a list with 44 at the front of it and there will be something on the tail of the list. That something will be whatever _C2 eventually becomes. Of course, we know that _C2 will eventually become [97,10], so [A|C] will become: [44,97,10], but at this time, Prolog does not know this. Prolog now goes on to notice that rule 2 can be satisfied if the right-hand-side of the rule can be satisfied. This right side now becomes the problem:

```
pos([97,-300,10],_C2).
```

These calls continue in this manner with Prolog attempting to verify either rule 2 or rule 3 until Prolog gets the problem in line 6:

```
pos([],_C5).
```

This problem matches rule 1. Prolog now knows that _C5 is [ ]. With all the necessary matching completed, Prolog can look up the value of the variable, X, in the original question. It works like this:

```
X matched with _C1,
_C1 matched with [44 | _C2],
_C2 matched with [97 | _C3],
_C3 matched with _C4,
_C4 matched with [10 | _C5],
_C5 matched with [],
giving X = [44, 97, 10].
```

After printing out the value for X, Prolog waits for input from the user. If the user types a ";" Prolog continues the search from this current point. If the user types a carriage return, Prolog returns from all the recursive calls and prompts the user for a new question.

Both member and pos follow the typical pattern found in recursive list processing algorithms: you break off the first item, deal with it, and then deal recursively with the rest of the list. At each level where you broke off an item, you combine your result with the results from the recursive call that processed the rest of the list. Recursion is really quite nice in that if you can simply list all the possible cases that can come up, together with their answers, the problem is solved. Recursion is not normally very efficient but most AI researchers have never really been concerned with efficiency.

4.2.7 Other Predicates

Some other Prolog predicates will prove useful. Two of these are assert and retract. Assert adds facts to the database while retract removes them. The following line adds the fact that Fred likes the Cubs to the database at the end of the likes predicates:

```
assert(likes(fred,cubs)).
```

The statement:

```
retract(likes(X,cubs)).
```

will cause Prolog to start looking at the top of the database for a fact that matches likes(X,cubs) and if it finds one, then that one fact will be removed from the database.

Two other useful predicates are write, that writes out a single argument without an end-of-line marker, and nl, that writes the end-of-line marker. Here is an example of write and nl:

```
?- write(hello),nl,write(world),nl.
```

This produces:

```
hello
world
```

Another important Prolog predicate to mention is the not predicate.[1] Naturally, it takes a true value and changes it to false and takes a false value and changes it to true. For example, if it is given that Matt likes the Mets, then giving Prolog the question:

```
?- not(likes(matt,mets)).
```

will produce: no. If nothing at all is stated about Bob liking or disliking the Cardinals, then:

```
?- not(likes(bob,cardinals)).
```

will produce: yes. This is a somewhat peculiar result for people well versed in predicate-calculus-based theorem proving from which Prolog was derived. In predicate calculus, without having the appropriate information about Bob and his likes and dislikes, it is impossible to prove the proposition that Bob does not like the Cardinals. Therefore, the Prolog not predicate is not really the same as the logical not found in predicate calculus.

4.3 Rules and Basic Rule Interpretation Methods

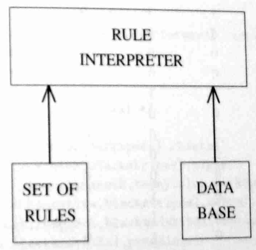


Figure 4.4: Rules and facts are submitted to a rule interpreter or inference engine.

When using rules, in general, the rule base and the facts are fed into a rule interpreter, sometimes also called an *inference engine*, because it works through the rules and facts and attempts to reach new conclusions. A simple diagram of this kind of system is shown in Figure 4.4. In general, the rules can be placed in any order and the facts can also be listed in any order and the system will still reach the same conclusions. Actually, however, there are conditions when the order of the rules and facts may make a difference, but we will

[1] Some Prologs use the predicate, \\+, rather than 'not.'

save that problem for a later section. Programming the system focuses on creating a set of rules. The people that create the rules are known as *knowledge engineers*. The set of rules is referred to as the *knowledge base* and the systems themselves are called *knowledge-based systems*. Rule-based systems are also sometimes referred to as *production systems* and *pattern directed inference systems*. To a certain extent, a rule-based system can be made to handle new problems just by having new sets of rules plugged in, a very flexible arrangement. Sometimes, however, a given rule interpreter will be inadequate and will need some reprogramming or even need to be replaced by one more suited to the particular problem.

There are a number of ways for a rule interpreter to deal with a system of rules. In this section we will program the two simplest methods, *forward chaining* and *backward chaining*, and then in later sections look at more complex methods. In this section the methods will be illustrated with a small animal identifying system.

4.3.1 A Small Rule-Based System

	albatross	penguin	ostrich	giraffe	cheetah	zebra	tiger
1 has hair	0	0	0	1	1	1	1
2 has claws	0	0	0	0	1	0	1
3 gives milk	0	0	0	1	1	1	1
4 eyes forward	0	0	0	0	1	0	1
5 has feathers	1	1	1	0	0	0	0
6 has hoofs	0	0	0	1	0	1	0
7 flies	1	0	0	0	0	0	0
8 chews cud	0	0	0	1	0	1	0
9 lays eggs	1	1	1	0	0	0	0
10 tawny color	0	0	0	1	1	0	1
11 pointed teeth	0	0	0	0	1	0	1
12 dark spots	0	0	0	1	1	0	0
13 black stripes	0	0	0	0	0	1	1
14 long legs	0	0	1	1	0	0	0
15 long neck	0	0	1	1	0	0	0
16 black & white	0	0	1	0	0	1	0
17 swims	0	1	0	0	0	0	0
18 flies well	1	0	0	0	0	0	0
19 eats meat	0	0	0	0	1	0	1

Figure 4.5: A set of animals with their characteristics listed as vectors.

The classic small example of a rule-based system has become the animal recognition system of Winston [262], where given some characteristics of an animal, the goal is to determine which of seven possible animals it is. Figure 4.5 lists the seven animals and their characteristics. It would be simple to come up with a set of Prolog language rules for this problem. For example, we could have a rule that an animal is an albatross if it is a bird and it flies well:

```
albatross :- bird, flieswell.
```

Instead of doing it this way, the identification rules will be listed as a series of facts. We will then write our own simple interpreters to work on these rules. For an example of this notation, here is how the rule for identifying an albatross will be coded:

```
idrule([bird,flieswell],albatross).
```

The first item in a rule will be the list of antecedents and the second item will be the consequent. As it happens in all the rules we will use, there will only be a single consequent, but, in general, there may be more than one. The top part of Figure 4.6 shows the set of rules we will use and below them some data for an unknown animal. Those rules with the predicate, 'rule,' are rules designed to "light up" intermediate conclusions while the rules labeled 'idrule' are used to give the identity of the unknown animal.

```
   /*  Rules for intermediate conclusions */

rule([hashair],mammal).
rule([givesmilk],mammal).
rule([flies,layseggs],bird).
rule([hasfeathers],bird).
rule([eatsmeat,mammal],carnivore).
rule([mammal,pointedteeth,hasclaws,eyesforward],carnivore).
rule([mammal,hashoofs],ungulate).
rule([mammal,chewscud],ungulate).

   /*  Rules to identify the animal */

idrule([carnivore,tawnycolor,darkspots],cheetah).
idrule([carnivore,tawnycolor,blackstripes],tiger).
idrule([ungulate,longlegs,longneck,tawnycolor,darkspots],giraffe).
idrule([ungulate,blackandwhite,blackstripes],zebra).
idrule([bird,longlegs,longneck,blackandwhite],ostrich).
idrule([bird,swims,blackandwhite],penguin).
idrule([bird,flieswell],albatross).

   /* some sample data */

flies.
layseggs.
flieswell.
```

Figure 4.6: A set of rules to identify the animals and an example of what the data for an unknown animal will look like.

4.3.2 Forward Chaining

The first major way of interpreting rules is called *forward chaining* and it works much the same way as a feed-forward neural network in that processing begins with the input data and lower-level rules. This method is also known as *bottom-up reasoning*, *data driven reasoning* and *antecedent reasoning*. It works by searching through the rule base in a systematic way for rules whose conditions are true. In general, the search may find more than one rule that could apply, but for now we will simply search through the database and apply the first rule whose conditions are true. In the next section we will deal with the problem of having more than one rule that could be applied.

First, we will illustrate the principle of forward chaining using the following data for the unknown animal:

```
flies.
layseggs.
flieswell.
```

We begin searching through the rules to see if any rule has all its conditions true. At this point we will be using only the rules labeled as 'rule,' and not any of the rules labeled 'idrule.' We come across the rule that if an animal flies and lays eggs then it must be a bird. This rule fires and adds to the database the fact that the animal is a bird by using the Prolog assert clause. The next step will be to remove the conditions that produced this conclusion. If these conditions were not removed, the interpreter would find the same rule again and an infinite loop would occur. After removing the conditions that caused the rule to fire, we begin again and look for another rule to fire. All these operations are easily stated in Prolog as shown below where the predicate f, is for forward chaining:

```
f :- rule(Conditions,Conclusion),
     fcheck(Conditions),   /* check for the antecedents */
     assert(Conclusion),
     remove(Conditions),
     f.
```

For example, with the data given above for an albatross, we call the predicate, f. First, f tries to match the 'rule' predicate and this rule about mammals is found:

```
rule([hashair],mammal).
```

The fcheck predicate looks to see if all the conditions in the list are true, but they are not. Prolog backs up to the rule predicate and finds another rule about mammals. This fails and Prolog finds a rule about birds:

```
rule([flies,layseggs],bird).
```

Fcheck finds that flies and layseggs are true, so bird is asserted. The facts, flies and layseggs are removed from the database and f is called again from inside f. Notice that this will put a long chain of calls to f on the stack. The process could be done more plainly and less wastefully without recursion if Prolog supported iteration. At any rate, in the second call, f finds the first rule about mammals fails, the second rule about mammals fails, the third rule about birds fails, and in fact, every rule fails because there are no more conclusions that can be reached.

When all the possible rules labeled 'rule' have been fired, we have the interpreter move on to a second definition of f:

```
f :- idrule(Conditions,Animal),
     fcheck(Conditions),
     remove(Conditions),
     writeln(Animal).
```

Here, the program looks for the characteristics of a particular animal and if it finds the answer, it prints out the identity of the animal. With the given data, the idrule about albatross will have its conditions met, and the answer will be printed. If the animal could not be identified, we have the Prolog interpreter fall through to this third definition of f:

```
f :- writeln('unknown animal').
```

The definitions of fcheck, remove, and writeln are given below:

```
/* Fcheck checks the first condition in the list of
   antecedents and then recursively checks the rest
   of them if the first one is true.  */

fcheck([]).
fcheck([First | Rest]) :- First, fcheck(Rest).

/* Remove removes the first fact in the list and goes on
   to remove the rest of them.  */

remove([]).
remove([First | Rest]) :- retract(First), remove(Rest).

/* Writeln, a handy statement to write answers with.  */

writeln(X) :- write(X), nl.
```

4.3.3 Backward Chaining

The second fundamental rule interpretation method is to use the Prolog method of backward chaining. It is also known as *top-down reasoning*, *goal-based reasoning*, and *consequent reasoning*, and it is the way Prolog itself operates. The Prolog predicate, b, defined below, finds a rule that identifies a particular animal:

```
b :- idrule(Conditions,Animal),
     bcheck(Conditions),
     remove(Conditions),
     writeln(Animal).
```

This definition works by selecting an identification rule, a rule that will identify an animal, and then the bcheck predicate checks if the conditions of the rule are true. If the conditions are true, they are removed and the answer is printed, otherwise, the failure causes Prolog to back up and find another identification rule to test. To cover the case when the animal cannot be identified because all the identification rules have failed, we need this second definition for b placed after the above one:

```
b :- writeln('unknown animal').
```

The bcheck procedure is more complex than fcheck. Bcheck first looks to see if the First condition in the list is already true. If it is, the rest of the conditions in the list are checked recursively. If the First condition is not in the database, Prolog moves on to try to find a way to prove the condition is true by selecting a rule that will show that the First condition is true. The definition of bcheck is as follows:

```
bcheck([]).
bcheck([First | Rest]) :- First, bcheck(Rest).
bcheck([First | Rest]) :- rule(Conditions,First),
                          bcheck(Conditions),
                          remove(Conditions),
                          assert(First),
                          bcheck(Rest).
```

As an example, suppose the data is once again that the animal in question flies, lays eggs, and flies well. We call the predicate, b and b will end up selecting the rule for a cheetah as its first candidate:

```
idrule([carnivore,tawnycolor,darkspots],cheetah).
```

When b calls bcheck, it passes on the list of requirements for an animal to be a cheetah. The first rule of bcheck will fail. The second rule will break off the carnivore condition, and ask if carnivore is true. It will not be true, so Prolog will move on to the third bcheck rule. This rule looks for a way to prove carnivore true. The rule predicate in bcheck finds this rule about carnivores:

```
rule([eatsmeat,mammal],carnivore).
```

Another call is made to bcheck with this new list of requirements. The first two rules for bcheck will fail. In the third rule, Prolog will try to find a way to prove the First condition, eatsmeat. This fails and Prolog backtracks.... Eventually, every possible way to prove that the animal is a cheetah fails. After this, more idrules fail. Eventually, albatross is selected as the goal to pursue and this proof finally succeeds.

Forward and backward chaining constitute the two major methods of searching through a rule base. Clearly, backward chaining can be very time consuming if there are many possible final conclusions that can be reached, however, forward chaining can also be time consuming because the interpreter may spend a great deal of time reaching a large number of conclusions that have no bearing whatsoever on solving a particular problem.

4.4 Conflict Resolution

The animal identifying expert system we have been using so far has avoided a problem that occurs in larger expert systems, that of *conflict resolution*. In the forward chaining version of the animal identifying expert system we assumed, and reasonably so, that whenever we found a rule whose antecedents were true we could simply apply the rule. In general,

however, in searching through a rule base there could easily be several rules that have their antecedents satisfied. These rules that could be applied are said to be *triggered*. Whichever rule is finally applied is said to *fire* but it becomes a problem to decide which of them should fire. For example, we may have the following set of rules:

IF A and B and C THEN X (1)
IF A and B THEN Y (2)
IF A THEN Z (3)

and all the conditions A, B, and C are true. In this case, rule (2)'s conditions are a superset of rule (3)'s conditions and rule (1)'s conditions are a superset of both rule (2)'s and rule (3)'s conditions. The programmer must tell the interpreter how to choose which rule to fire. The final answer that the expert system produces may well be different depending on which of these rules actually fires.

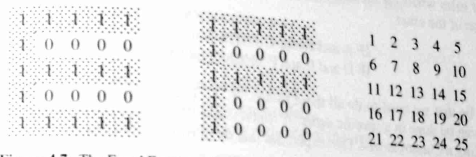


Figure 4.7: The E and F patterns in 5 × 5 matrices and the numbering of the units.

Notice how this problem is the same as the problem discussed in Section 2.1 where we developed a simple matrix multiplication technique to use to discriminate between the letters E, F, and H. Instead of using the larger representations of the letters as in Section 2.1, we will use the two smaller 5 × 5 versions of the letters E and F shown in Figure 4.7. To identify E and F we could have the following rules, where the number indicates the bit is a 1:

IF	1 & 2 & 3 & 4 & 5 & 6 & 11 & 12 & 13 & 14 & 15 & 16 & 21 & 22 & 23 & 24 & 25
THEN	the letter is E
IF	1 & 2 & 3 & 4 & 5 & 6 & 11 & 12 & 13 & 14 & 15 & 16 & 21
THEN	the letter is F

Again, using this rule-based plan, if the unknown letter we want to identify is an F, there is no problem, but if the unknown letter is an E, both these rules are triggered and we are left with the problem of which one should fire. In Section 2.1 we were counting the number of votes for each letter. If we follow that example here, we will declare the letter to be an E. Choosing the rule to fire that has the most specific set of antecedents true is called *specificity ordering*.

Another solution to rule conflicts is to simply take the first rule that is encountered as the one to fire. Of course, this is easily done without looking for other rules that may also be triggered, but this leaves the programmer with the problem of placing the rules in the knowledge base in just the right order.

Other solutions to rule conflicts involve keeping track of how often and when each of the triggered rules has fired. Then a tie is broken by using either the most commonly used rule or the most recently used rule, the least commonly used rule or the least recently used rule. As an example of using the most commonly used rule, suppose you go to your doctor and complain of flulike symptoms. If the flu is quite common at the time, it is likely that the doctor will decide you have just the ordinary flu because (1) it is what everyone else has at the moment and (2), statistically speaking, it is the most likely disease that fits the symptoms. Of course there are occasions where this will not be correct. You could have just come back from a trip up the Amazon river and during the trip you could have acquired a rare tropical disease with symptoms just like those of the flu.

Another method of conflict resolution is to prevent it from happening altogether by adding some additional conditions to the rules. For instance, we may have the following pair of rules where all the antecedents are true and where one set of antecedents is not a superset of the other:

IF A and B and C THEN W
IF D and E and F THEN X and Y and Z

It may be that we need to do all these things, W, X, Y, and Z, but it is possible that these things can be done in a specific order. A concrete example of this might be the problem of decorating a Christmas tree. One expert has this set of rules for decorating Christmas trees:

IF there is a string of lights available
THEN put on the string of lights

IF there is a large ornament available
THEN put the large ornament at the bottom of the tree

IF there is a medium ornament available
THEN put the medium ornament in the middle of the tree

IF there is a small ornament available
THEN put the small ornament at the top of the tree

IF there is some tinsel available
THEN put the tinsel on

At the beginning of the process there are lots of strings of lights, lots of every size ornament, and lots of tinsel available so every rule is triggered. This problem can be avoided by sequentializing the process. We can break the problem into a series of steps, the "putting on lights phase," the "putting on large ornaments phase," the "putting on medium ornaments phase," the "putting on small ornaments phase," and the "putting on tinsel phase." For each phase, there will be a variable associated with it, such as "lights_phase" for the putting on lights phase. It will be 1 during this phase and 0 otherwise. When one phase is finished, a rule can be used to change the state to the next phase:

IF lights_phase = 1 and
there is a string of lights available
THEN put on the string of lights

IF lights_phase = 1
THEN lights_phase ← 0 and
large_ornaments_phase ← 1

IF large_ornaments_phase = 1 and
there is a large ornament available
THEN put the large ornament at the bottom of the tree

IF large_ornaments_phase = 1
THEN large_ornaments_phase ← 0 and
medium_ornaments_phase ← 1

IF medium_ornaments_phase = 1 and
there is a medium ornament available
THEN put the medium ornament in the middle of the tree

IF medium_ornaments_phase = 1
THEN medium_ornaments_phase ← 0 and
small_ornaments_phase ← 1

IF small_ornaments_phase = 1 and
there is a small ornament available
THEN put the small ornament at the top of the tree

IF small_ornaments_phase = 1
THEN small_ornaments_phase ← 0 and
tinsel_phase ← 1

IF tinsel_phase = 1 and
there is some tinsel available
THEN put the tinsel on the tree

The process has to start with lights_phase = 1. Using this plan, there will be only two rules to choose from at a time, the one that adds something to the tree and the one that changes the state of the problem. A tie can be broken by taking the rule with the longer list of antecedents.

It is also important to note that, whereas we can break the one set of rules for decorating a Christmas tree into five different phases, we can also declare each phase to be a separate little expert system by itself. This has the nice advantage that fewer rules have to be checked in each part of the process. XCON is a famous expert system that configures VAX computer systems and it breaks the process into different phases that are handled sequentially.

Still another alternative for dealing with rule conflicts is to try to find more specific conditions for each rule so that the right consequences are produced without any conflicts. One way to do this using the first example in this section is to transform the rules into these:

IF A and B and C THEN X
IF A and B and not C THEN Y
IF A and not B and not C THEN Z

These rules reflect the kind of data that would have to be submitted to a back-propagation network where every input must have a value.

The final alternative is to admit that you do not know enough to say for sure which rule to use, so choose one at random in the hope that it works out. If this choice fails, back up to the point where you made the arbitrary choice and try another rule.

4.5 More Sophisticated Rule Interpretation

Problems sometimes occur with the simple forward and backward chaining methods we have described, but these can be dealt with using more complex techniques. There are also other capabilities that rule interpreters need and we will mention some of them in this section.

4.5.1 Dealing with Incomplete Data by Asking Questions

One problem is that an expert system may need to operate with an incomplete set of data. This could be because some data is unobtainable or because the person supplying the data has just been careless by not giving all the facts that are available. For instance, a person may walk up to the animal identifying expert system and ask it to identify some animal with a tawny color, long legs, a long neck, and dark spots. Obviously the animal is a giraffe, but without the program having the additional information that the animal has hair and has hoofs, the system will not be able to identify the animal.

One solution to this problem is to make the rule interpreter a little more sophisticated by having it ask the user about facts that the user has not given. This can be done while the system is searching through the list of rules. Let us suppose we had the interpreter using forward chaining and the data that it has on the animal is that it has a tawny color, long legs, a long neck, and dark spots. When the interpreter picks a rule to try to apply, it first looks in the database to see if the antecedents of the rule are true. If they are, there is no problem and the rule fires, but if one or more antecedents are not there, we let the program ask the user if the particular missing facts in the antecedent about the animal are true. With the data for this problem arranged in the order given for rules in Section 4.3, the first rule we happen upon is to try to determine if the animal is a mammal by checking if the animal has hair. The interpreter would then ask the user if the animal has hair. In this case it happens to be a sensible question since this rule will contribute directly to the problem of determining that the animal is a giraffe. In most cases we can expect that the questions asked by systems will not be very sensible. The system might just as easily have found the rule about proving an animal is a bird as the first rule. So, while we can say that this method of getting additional information from the user will eventually work, the interpreter will usually display a considerable lack of "good sense." In addition, the sheer number of questions it might ask could be very annoying to the user.

Of course, asking questions can also be done in a backward chaining expert system and here the questions can be a little more focused. The program can start with the assumption that the animal is an albatross and ask relevant questions for that hypothesis rather than jumping around between rules at random.

4.5.2 Other Activation Functions

When a human expert is given incomplete data the person will not ask a lot of irrelevant questions the way a simple forward or backward chaining system would. A better system is to use another activation function to pick good questions to ask, an analog function that will let you rate the possible answers. In the animal identification problem, one solution that immediately comes to mind is to keep a list of each animal and its characteristics and then search through the list looking for the highest percentage of matches, take the most likely candidates, and do backward chaining, asking questions as necessary about characteristics that the user has not given. So, given that the animal has a tawny color, long legs, a long neck, and dark spots, this should activate the giraffe candidate more than any other.

If your data is incomplete and you cannot ask about the missing features, you may still want a program that will give you some indication of how likely some conclusion is, and again for that you need some real-valued activation function. Some other techniques and functions that can help are described below.

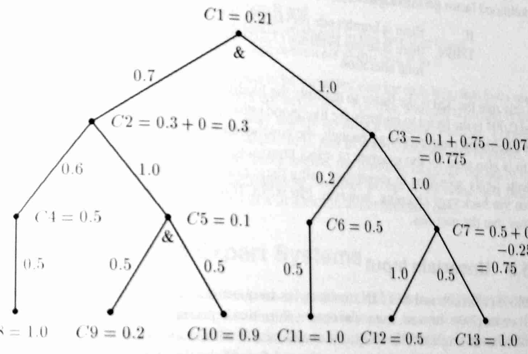


Figure 4.8: An example illustrating MYCIN's method of propagating certainty factors through its network. The nodes marked with & are AND nodes while the rest are OR nodes.

One of the most commonly used activation functions comes from an early experimental expert system called MYCIN [24], a program designed for analyzing bacterial infections. MYCIN uses rules that are used by an inference engine to construct and evaluate a network. In MYCIN the network is an AND/OR tree like that shown in Figure 4.8. Each node receives values from its children in the same way that nodes in neural networks do by taking the weight on the link between nodes and multiplying it by the value of the child's node. In MYCIN the weights are called attenuation factors because they are in the range from 0 to 1.0. The activation value of a node is called the *certainty factor* and it also runs from 0 to 1 and corresponds to a rating factor like a probability. At the AND nodes the value of the node is calculated by taking the smallest incoming value as the value of the node. At the OR nodes the value of the node is calculated by taking two incoming values, a and b, adding them together and subtracting their product:

$$certaintyfactor = a + b - a \times b.$$

This formula can be generalized to more than two incoming values. Values for nodes less than some arbitrary threshold, say 0.2, are set to zero. This is a very ad hoc method but it works for many applications. Other, more sophisticated methods have been proposed by more mathematically inclined people who want to make the certainty factors as close as possible to the precise mathematical definition of probability.

Another different method of rating rules and conclusions was developed for the PROSPECTOR expert system [30]. This system was designed to evaluate the prospects for mineral deposits. It gives all its possible conclusions an initial rating of 1. In PROSPECTOR, the rating is called a *likelihood ratio*. As each conclusion gains evidence, this ratio is multiplied by a factor greater than one. If key evidence for the conclusion is missing, the likelihood factor gets multiplied by a value less than 1. One rule from PROSPECTOR is:

```
IF      there is hornblende pervasively altered to biotite
THEN    there is strong evidence (320,0.001) for potassic
        zone alteration
```

In this rule the 320 is the factor to multiply the likelihood ratio by if the antecedent is true and 0.001 is the factor to multiply the likelihood ratio by if the antecedent is not true. When the likelihood ratio gets large enough, the conclusion is considered true. The likelihood ratio is also used by the program to select promising rules to investigate. PROSPECTOR selects likely scenarios to investigate using these ratings and then goes on to investigate them via backward chaining, however, PROSPECTOR is also willing to investigate any theory that the user has.

4.5.3 Uncertain Input

In PROSPECTOR and MYCIN, the answers to questions do not have to be a plain yes (1.0) or no (0.0). Instead, users can enter values that represent their degree of confidence in the answer. In PROSPECTOR, the confidence intervals run in integral values from –5 for definitely not, to 0 for unknown, and to +5 for absolutely certain. In PROSPECTOR, the rules can also have confidence intervals set so that certain rules should not even be considered by the program unless the response from the user is within a certain range, say, perhaps, from +2 to +5.

To cope with the fuzziness of values, several mathematical methods have been proposed. One of them is based on the theory of fuzzy sets proposed by Zadeh. (For a short introduction see [268].)

4.5.4 Extra Facilities for Rule Interpreters

The expert system capabilities described so far have really been only those associated with the pattern recognition and control aspects of the problem. In practical inference engines, you typically need many of the capabilities found in general purpose computer languages, such as having variables that you can do arithmetic on. For instance, we might need an

expert system for packing Christmas tree ornaments in boxes. Suppose six big ornaments and four small ones can fit in a box. To do this problem, rules like these will be necessary:

```
IF      the large ornament count in box B < 6 and
        there is a large ornament L that needs to be packed
THEN    put the large ornament L in box B and
        increment the large ornament count in box B by 1.

IF      box B has 6 large ornaments and
        there are < 4 small ornaments in box B and
        there is a small ornament S that needs to be packed
THEN    put the small ornament in box B and
        increment the small ornament count in box B by 1.

IF      box B has 6 large ornaments and
        box B has 4 small ornaments and
        there are more ornaments that need to be packed
THEN    get another empty box B and
        set the number of small ornaments in this box B to 0 and
        set the number of large ornaments in this box B to 0
```

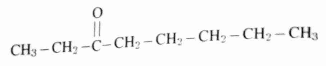
Therefore, in this program there will need to be variables, one for each box, that keep track of how many large and small ornaments are in the box. For this, a record structure comes to mind. To "get another empty box" may require creating a dynamic variable. Looping, conventional if statements, arithmetic calculations, and I/O operations are also required. As additions like these are made to the inference engine, the resulting language begins to look very much like a general purpose programming language except it has some new pattern recognition oriented control structures in it in addition to the ordinary ones.

4.6 The Famous Expert Systems

In this section we want to look at the kinds of problems that have been handled by traditional rule-based expert systems by looking at some of the more well-known systems. Two of these early expert systems are DENDRAL for analyzing mass spectrograms and MYCIN for analyzing bacterial infections. These were early systems that demonstrated the principles involved, but they were never good enough to have a technical or economic impact. The systems that made headlines because of their economic benefit were PROSPECTOR, used for mineral exploration, and XCON (alias R1), used to configure VAX computer systems. We also look briefly at the ACE system from AT&T that is used to diagnose faults in telephone cables. For the most part, expert systems are developed using specialized languages and commercial shells that have facilities to make the job easier. Two such languages mentioned below are OPS4 and OPS5. Expert systems are rarely done in the traditional AI languages such as Lisp and Prolog.

4.6.1 DENDRAL

The first famous expert system we will look at is DENDRAL [17]. DENDRAL was designed to analyze mass spectrograms of certain classes of organic compounds. The goal of DENDRAL is to deduce the molecular structure of the compounds. For instance, given some compound like: $C_8H_{16}O$, the program can take in information from a mass spectrogram and conclude that the structure is:



The compound is called 3-octanone.

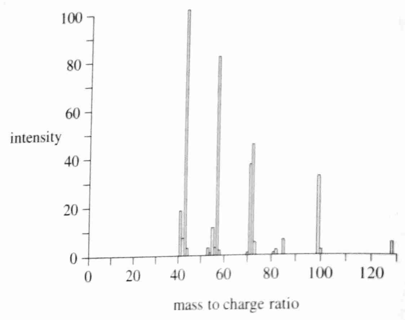


Figure 4.9: A plot of intensity as a function of the mass/charge ratio for the compound 3-octanone. The pattern of peaks allows chemists (and DENDRAL) to guess what substructures are present.

The process begins by taking the compound in question and heating it in an oven so that the molecules break into pieces. The pieces are charged ions that are accelerated by an electric field and then deflected by a magnetic field onto a photographic plate. Naturally, heavy ions are deflected less than light ones, and ions with more charge are deflected more than ions with little charge. The apparatus to do this is much like the system in a CRT except a CRT only uses electrons. The darkness at a particular spot on the photographic plate indicates how many of one particular kind of ion is hitting the plate at that spot. A plot of some results is shown in Figure 4.9.

With the information available from the mass spectrogram, it is possible for an expert to determine what ions are present. DENDRAL uses rules for this like:

```
IF      there is a high peak at X, Y, and Z and any size peak at W
THEN    the substructure U is present
```

This information about substructures that are present is used by another part of the program to generate possible ways of putting together molecules that contain the necessary constituents. In the next phase, the list of possible candidates is passed on to another part of the program that knows how these kinds of compounds will break up when they

are heated. For each of these candidates, the program computes the mass spectrogram that will result, then these simulated spectrograms are compared with the one from the unknown compound. The simulated mass spectrogram that most closely matches the actual spectrogram is chosen as the answer. DENDRAL was actually only capable of dealing with a small number of compounds and it has never been used commercially. Another program called CONGEN is widely used, however, Dreyfus and Dreyfus [28] report that it does not use any techniques that would be regarded as artificial intelligence techniques and so they say it should not be regarded as an expert system.

4.6.2 MYCIN

The MYCIN expert system was designed to analyze information on bacterial infections and give advice on the best possible treatments. It was a backward chaining system and had about 500 rules. The following is part of a sample session with MYCIN from [24]. Comments within curly brackets are not part of the actual dialog. User responses come after a pair of asterisks:[2]

```
- - - - - - - - PATIENT-1 - - - - - - - -
(1) Patient's name:  (first-last)
**FRED SMITH
                                          {background data}
(2) Sex:
**MALE
(3) Age:
**55
(4) Have you been able to obtain positive cultures from a site at
    which Fred Smith has an infection?
**YES
 - - - - - - - - INFECTION-1 - - - - - - - -
 (5) What is the infection?
 **PRIMARY-BACTEREMIA
 (6) Please give the date and approximate time when signs or symptoms
     of the primary-bacteremia (INFECTION-1) first appeared.  (mo/da/yr)
 **MAY 5, 1975
 The most recent positive culture associated with the primary-bacteremia
 (INFECTION-1) will be referred to as:
   - - - - - - - - CULTURE-1 - - - - - - - -
   (7) From what site was the specimen for CULTURE-1 taken?
   **BLOOD
   (8) Please give the date and time when this blood culture (CULTURE-1)
       was obtained (mo/da/yr)
   **MAY 9, 1975
   The first significant organism from this blood culture (CULTURE-1)
   will be referred to as:
     - - - - - - - - ORGANISM-1 - - - - - - - -
     (9) Enter the identity of ORGANISM-1:
     **UNKNOWN
            {typically the identity is not known and must be inferred}
     (10) is ORGANISM-1 a rod or coccus (etc.):
```

[2]Reprinted from "Production Rules as a Representation for a Knowledge-Based Consultation Program," by Randall Davis, Bruce Buchanan, and Edward Shortliffe, in *Artificial Intelligence*, Volume 8, 1977 with permission from the publisher, Elsevier Science Publishers B. V.

```
     **ROD
     (11) The gram stain of ORGANISM-1
     **GRAMNEG
     (12) Have there been POSITIVE cultures from sites which are NOT
          associated with infections about which you are seeking advice?
     **NO
     (13) Did ORGANISM-1 grow aerobically?
     **YES (8)
     (14) Did ORGANISM-1 grow anaerobically?
     **YES
     (15) What is the suspected portal of entry of ORGANISM-1
          into this sterile site?
     **GI (6)
       .  .  .

 My therapy recommendation will be based on the following probable
 infections(s) and potential causative organism(s):

 INFECTION-1 IS PRIMARY-BACTEREMIA
                                 {possible identities}
 <item 1> The identity of ORGANISM-1 may be PSEUDOMONAS-AERUGINOSA
 <item 2> The identity of ORGANISM-1 may be KLEBSIELLA-PNEUMONIAE
 <item 3> The identity of ORGANISM-1 may be E. COLI
 <item 4> The identity of ORGANISM-1 may be BACTEROIDES-FRAGILIS
 <item 5> The identity of ORGANISM-1 may be ENTEROBACTER
 <item 6> The identity of ORGANISM-1 may be PROTEUS-NON-MIRABILIS

  (46) On a scale of 0 to 4, where higher numbers indicate increasing
       severity how would you rate Fred Smith's degree of sickness?
  **3
  (47) Does Fred Smith have a clinically significant allergic reaction
       to any antimicrobial agent?
  **NO
  (48) Fred Smith's weight in kilograms (or <number> POUNDS):
  **70
                        {note that 2 drugs cover for 6 potential organisms}
[Rec 1] My preferred therapy recommendation is as follows:
  In order to cover for Items <1 2 3 5 6>:
    Give:  GENTAMICIN
    Dose:  119 mg (1.7 mg/kg) q8h IV [or IM] for 10 days
    Comments:  Modify dose in renal failure
  In order to cover for ITEM <4>:
    Give:  CLINDAMYCIN
    Dose:  595 mg (8.5 mg/kg) q6h IV [or IM] for 14 days
    Comments:  If diarrhea or other GI symptoms develop, patient
      should be evaluated for possible pseudomembranous colitis.
Do you wish to see the next choice therapy?
                       {alternative choices are available}
**Y
Which, if any, of the above drugs do you wish eliminated from
consideration?
**CLINDAMYCIN

[Rec 2] Next best therapy recommendation:
  In order to cover for items <2 3 4 5 6>:
    Give:  CHLORAMPHENICOL
```

```
    Dose:  1 gm (15 mg/kg) q6h IV for 14 days
    Comments:  Monitor patient's white count
  In order to cover for Item <1>:
    Give:  GENTAMYCIN
    Dose:  119 mg (1.7 mg/kg) q8h IV [or IM] for 10 days
    Comments:  Modify dose in renal failure
```

MYCIN was only an experimental system and it has never been used in practical situations. Davis [25] has experimented with an automated knowledge acquisition system for MYCIN called TEIRESIAS.

4.6.3 PROSPECTOR

The PROSPECTOR system was developed by Stanford Research Institute with the help of grants from the Office of Resource Analysis of the US Geological Survey and the National Science Foundation. PROSPECTOR's main claim to fame is its possible discovery of a molybdenum ore deposit in Washington state. There are actually many versions of this story. One version is that the system predicted that there would be a molybdenum deposit at a certain spot near Mt. Tolman and that no one had ever suspected that there was such a deposit there. Another version is that people strongly suspected that there was a deposit in the area, but the program predicted where it was and where other such deposits could also be found. Another version is that the deposit was known to be there in the first place. The concentration of molybdenum ore found by PROSPECTOR was not enough to justify a mining operation at the time. Below are two excerpts from a conversation with PROSPECTOR. The first excerpt is from an early version where the confidence interval for answers given by the user ran from -2 to +2 in integral amounts. In a later version of the program the interval runs from -5 to +5. Here is the sample conversation with PROSPECTOR:[3]

```
--------------Program execution is now starting -------------------
   Do you want to volunteer any evidence?  YES

   A. Space name of evidence:    SPACE-25L
        New likelihood of (Widespread Igneous Rocks):  2
   B. Space name of evidence:  NIL

   Proceeding to establish the likelihood of (Massive Sulfide
      Deposit):

    1.  Do you have anything to say about (Volcanic province and
major fault zone)?  1
    2.  Do you have anything to say about (Mineralization)?  0
    3.  Do you have to say about (Near shore depositional sequences
of andesites, rhyolites, or dacites)?  0
    4.  Do you have anything to say about (Pillow structures) 2
    5.  Do you have anything to say about (Breccia)?  NO
    6.  Do you have anything to say about (Rhyolite or dacite plug)?
YES
```

[3] Reprinted from [30] "Semantic Network Representations in Rule-Based Inference Systems," by Richard Duda, Peter Hart, Nils Nilsson, and Georgia Sutherland, in *Pattern-Directed Inference Systems*, 1978, with permission from the publisher, Academic Press.

```
  6a.  Have you anything to say about an entity with the composition
of (OR RHYOLITE DACITE)? 1

This entity will hereafter be referred to as ENTITY-1

  6b.  Have you anything to say about whether ENTITY-1 has the form
of PLUG? 2
  7.  Do you have anything to say about (Galena, sphalerite, or
chalcopyrite filled cracks in rhyolite or dacite)?  YES
  7a.  Have you anything to say about an entity with the composition
of (OR GALENA SPHALERITE CHALCOPYRITE)? 2

This entity will hereafter be referred to as ENTITY-2

  7b.  Have you anything to say about whether ENTITY-1 has the property
of CONTAINING-CRACKS? 2
  7c.  Have you anything to say about whether ENTITY-2 is CONTAINED-IN
ENTITY-1?  1
  .
  .
  .
 10.    Do you have anything to say about (Olivine or alkaline
andesite)?  2
 11.    Do you have anything to say about (Calc-alkaline andesite)?
NO
  .
  .
  .
 14.    Do you have anything to say about (Clay Minerals)?  0
 15.    Do you have anything to say about (Reduction process)?  0
 16.    Do you have anything to say about (Bleaching of rocks)?  YES
 16a.    Have you anything to say about an entity with the composition
of ROCKS? 2

This entity will hereafter be referred to as ENTITY-4

  16b.    Have you anything to say about whether ENTITY-4 has the
property of BLEACHED?  1
  .
  .
  .
 22.    Do you have anything to say about (Prospect within a few
miles of known MDS)? NO

  In summary, the following have been observed:

  The most likely deposit type is (Massive Sulfide Deposit)  Its
current probability is 0.01465
  The most encouraging observations were:  (Galena, sphalerite, or
chalcopyrite filled cracks in rhyolite or dacite) and (Rhyolite or
dacite plug)
  The only discouraging observation was:  (Olivine or alkaline
andesite).
  The most useful additional information would be:    (Mineralization)
and (Breccia)
----------------------------DONE-----------------------------------
```

In the following example (from [31]), note how the user can get a geological education by asking WHY?

```
13-- Do you know the age of the continental-margin mobile belt?  WHY

The vast majority of porphyry copper deposits are younger than 200
million years.   Major age groups include:

  1.    An older group of 200 to 150 m.y.  (million year) porphyries in
        British Columbia;

  2.    An important younger group of 70 to 50 m.y.  deposits in the
        Mideast, British Columbia (Skeena Arch), and Arizona-New Mexico-
        Sonora (Laramide) copper province;

  3.    The major belt of 60 to 4 m.y.  deposits in the Andes of Argentina,
        Chile, Peru, Ecuador, and Colombia.
          .  .  .
```

4.6.4 ACE

The next system to look at is ACE, short for automated cable expertise, and it may have been the first expert system to be used commercially. It was developed at AT&T Bell Laboratories and it uses forward chaining and has about 300 rules. It is written in OPS4, Lisp, C, and Unix shell. Its job is to look at a database that contains reports of subscriber telephone problems and analyze the causes of the problems. In scanning the reports, ACE does nightly what used to require a human being a month to do. Being able to work this fast is an important ability. Previously, it could take a human expert a month to analyze problems. This meant that the repair staff was involved in doing a large number of short-term fixes that actually resulted from a single larger problem that was undetected.[4]

4.6.5 XCON

The last system to look at is XCON used by Digital Equipment Corporation to properly configure VAX computer systems. XCON stands for eXpert CONfigurer and it is also sometimes known as R1. It is perhaps the most important program that can be credited with fueling interest in expert systems because it reportedly saves DEC millions of dollars a year while doing a better job of configuring VAX computer systems than human beings can do. Besides the cost savings involved, there is also the increased customer satisfaction that comes from having systems correctly configured on the first try. Our descriptions of XCON come from [200].[5]

The history of XCON began in 1974 when a DEC engineer suggested that a program be written to check all customer orders for PDP-11 computers. Some early parts of the

[4] This information comes from a talk given by Dr. George V. Otto, AT&T Bell Laboratories.
[5] Reprinted from *The Artificial Intelligence Experience: An Introduction*, by Susan J. Scown, 1985, with permission from the publisher, Digital Press.

system were done in Fortran and Basic, but ultimately, after consulting with members of the Computer Science Department at Carnegie-Mellon University, it was decided to try an artificial intelligence approach. One important consideration involved in the decision was the fact that the number of components used in VAX computers and the specifications for those components were constantly changing. A program that used traditional programming methodology would have to be constantly changed as new and improved components were added. With the AI methodology, these changes could be made very simply by just adding new rules. Scown reports that "Most of Digital's development team agree that the AI solution has been proven and is ultimately the best approach for this problem." In late 1985, XCON had about 4,200 rules and by late 1988 this was up to over 10,000. This large number of rules makes for an interesting new problem: there are very few people who are qualified to modify XCON. Any modifications must be checked to insure that the new fixes do not ruin the parts of the system that are known to work correctly. Researchers are currently looking into this problem and have come up with a knowledge acquisition system for XCON called RIME (see [109] and [7]).

The major researcher and developer of XCON has been Professor John McDermott. Below is a description excerpted from the book, *The Artificial Intelligence Experience* written by Scown (a DEC employee) and published by Digital Press, that describes some of how the system works and the effectiveness of it.

* How XCON Works

XCON accepts as input a list of items on a customer order, configures them into a system, notes any additions, deletions, or changes needed in the order to make the system complete and functional, and prints out a set of detailed diagrams showing the spatial relationships among the components as they should be assembled in the factory.

The users are

* Technical editors who are responsible for seeing that only configurable orders are committed to the manufacturing flow.

* The assemblers and technicians in Digital's manufacturing organization who assemble the systems on the plant floor.

* Sales people, who use XCON in conjunction with XSEL, an expert system that helps them prepare accurate quotes for customers. This can be done on a dialup basis from the customer's site.

* Scheduling personnel who use information from XCON to decide how to combine options for the most efficient configurations.

* Technicians who assemble systems at the customer's site.

Configuration tasks like that of XCON can be thought of as heuristic searches for an acceptable configuration through a search space of possible configurations. McDermott used a "match search" pattern-matching method that does not deviate from the solution path; backtracking is rarely ever required because XCON's knowledge is usually sufficient to determine an acceptable next step.

Elements of the system include

* OPS5's production memory (knowledge in condition/action rule form about how to configure the systems), the embodiment of the heuristic knowledge base.

* The working memory, which starts with the set of customer-ordered components but by the end of processing has accumulated descriptions of partial configurations that will be used to complete the full configuration.

* The inference engine, which is the OPS5 interpreter. The interpreter selects and applies rules.

* An additional component database (descriptions of each of the components that may be configured in systems).

* User interface software that allows the user to interactively enter and modify orders, review XCON output for those orders, and enter problem reports.

* Traditional software for database access, the collection of statistics on hardware resource utilization and functional accuracy, and for automatically routing problem reports entered by users to the support organization.

The major subtasks within XCON are

* Checking the order for gross errors, such as missing prerequisites, wrong voltage or frequency, no central processing unit, etc. The first subtask is also concerned with unbundling line items to the configurable level, assigning devices to controllers, and distributing modules among multiple secondary buses.

* Placing the components in the central processing unit cabinets and then finding an acceptable configuration of the secondary bus by placing modules in backplanes, backplanes in boxes, and boxes in cabinets.

* Configuring the rest of the components on the secondary bus-panels, continuity cards, and unused backplanes. Also, computing vector and address locations for all modules.

* Laying out the system on the floor and determining how to cable it together.

The production rules in XCON that describe the different subtasks are grouped together. The rules are separated into subtasks both for easier maintenance and to increase efficiency because the interpreter need only consider the rules in a single subtask at any given time.

An example of an XCON rule translated from OPS5 into normal English follows:

R1-Panel-Space

If: The most current active context is panel-space for module-x And module-x has a line-type and requires cabling And the cabling that module-x requires is to a panel And there is space available for a panel in the current cabinet And there is no panel already assigned to the current cabinet with space for module-x And there is no partial configuration relating module-x to available panel space

Then: Mark the panel space in the cabinet as used And assign it the same line type as module-x And create a partial configuration relating module-x to the panel space.

XCON runs in batch mode, processing an order every minute or two. XCON looks into its database for an order to configure and if it finds an order, XCON configures it, updates the database with its output, and looks for another order.

* Testing

At the beginning, Digital used to create a test set of orders, either 25 customer orders or sample orders generated internally. The idea was to test the vast majority of rules on the most difficult orders. Once XCON could run all of these, the developers were confident that the rules functioned well. As new orders came through from customers, the developers would then see where XCON had failed and add those problems to the test cases, constantly making the tests tougher and tougher. These test cases were run against each rule change and/or each formal release. Now, as developers change rules in XCON, they run only those tests they believe are necessary. But before each new release of updated software to the production environment, a complete set of regression tests is run....

* XCON's Performance

Success for XCON has always been difficult to define. At the beginning, the development team had long and heated debates about the defining criteria. They decided that XCON would have to examine all orders, including the most difficult ones. The degree of XCON's accuracy, as judged by human experts was initially 75 percent and rose toward a goal of 95 percent over a period of about a year and a half. To increase accuracy was quite a difficult task because the development team was constantly adding new products and finding more and more "hidden" details about how to properly configure a VAX system. Success also became hard to define because the experts often disagreed on what was correct and what was not.

Another measure of success was acceptance by the technical editors and the engineers in the factory. The technical editors and engineers were at first unwilling to accept XCON as a software product. Later, after the development team had run a large number of orders through XCON, the technical editors and engineers accepted the fact that the system worked.

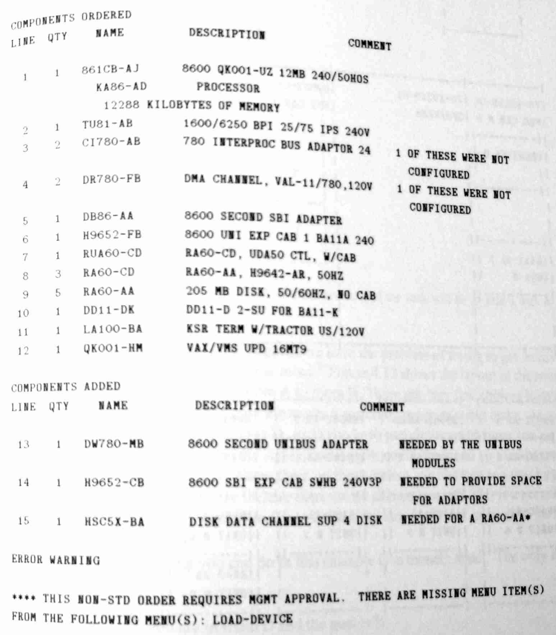
The average runtime required to configure an order is currently 2.3 minutes. Small PDP-11s sometimes take less than 1 minute to configure, while a 200 line item VAX 8600 cluster may take many minutes. Before XCON and before the complications of clusters, technical editors required 25-35 minutes for an average order and got 70 percent of them correct. XCON provides several hundred pieces of information per order, and creates a usable configuration 98 percent of the time. XCON performs about six times as many functions as the technical editors used to perform.

* XCON's Impact

XCON has allowed Digital to avoid costs that it would have incurred if Digital had been forced to hire more technical editors as the volume of systems sold increased.

XCON has also made possible another significant cost-avoidance and efficiency measure in the manufacturing process. Before XCON's accurate configuration plans were available, systems were sometimes assembled up to a point at which a problem was discovered, and then the system had to be dismantled and reconfigured. This wasted floor space in the assembly plants as well as time. With XCON, the configurations are more dependable, so the manufacturing process can, in turn, be more efficient.

A sample report generated by XCON is shown in Figures 4.10 and 4.11.



```
COMPONENTS ORDERED
LINE  QTY    NAME            DESCRIPTION
                                                   COMMENT

  1    1   861CB-AJ        8600 QK001-UZ 12MB 240/50HOS
             KA86-AD         PROCESSOR
              12288 KILOBYTES OF MEMORY
  2    1   TU81-AB         1600/6250 BPI 25/75 IPS 240V
  3    2   CI780-AB        780 INTERPROC BUS ADAPTOR 24    1 OF THESE WERE NOT
                                                            CONFIGURED
  4    2   DR780-FB        DMA CHANNEL, VAL-11/780,120V    1 OF THESE WERE NOT
                                                            CONFIGURED
  5    1   DB86-AA         8600 SECOND SBI ADAPTER
  6    1   H9652-FB        8600 UNI EXP CAB 1 BA11A 240
  7    1   RUA60-CD        RA60-CD, UDA50 CTL, W/CAB
  8    3   RA60-CD         RA60-AA, H9642-AR, 50HZ
  9    5   RA60-AA         205 MB DISK, 50/60HZ, NO CAB
 10    1   DD11-DK         DD11-D 2-SU FOR BA11-K
 11    1   LA100-BA        KSR TERM W/TRACTOR US/120V
 12    1   QK001-HM        VAX/VMS UPD 16MT9

COMPONENTS ADDED
LINE  QTY    NAME            DESCRIPTION              COMMENT


 13    1   DW780-MB        8600 SECOND UNIBUS ADAPTER    NEEDED BY THE UNIBUS
                                                          MODULES
 14    1   H9652-CB        8600 SBI EXP CAB SWHB 240V3P  NEEDED TO PROVIDE SPACE
                                                          FOR ADAPTORS
 15    1   HSC5X-BA        DISK DATA CHANNEL SUP 4 DISK  NEEDED FOR A RA60-AA*

ERROR WARNING

**** THIS NON-STD ORDER REQUIRES MGMT APPROVAL.  THERE ARE MISSING MENU ITEM(S)
FROM THE FOLLOWING MENU(S): LOAD-DEVICE
```

Figure 4.10: Part of the report generated for an order by XCON. The rest is shown in the next figure.

```
CABINET LAYOUT

|------------|
|CONSOLE     |
|LA100-BA 0  |
|            |
|            |
|            |
|            |
|------------|


|------------|--------------------|------------|------------|
|70-19218-01 |70-19219-01         |H9652-CB 1  |H9652-FB 1  |
|FEC CAB # 0 |CPU/KA86            |SBI CAB # 1 |UEC CAB # 1 |
||----------||                    |            |            |
||RL02-FK 0 ||                    |            |            |
||          ||                    |            |            |
||----------||                    |            |            |
|            |                    |            |            |
|            |                    |            |            |
||----------||                    |            ||----------||
||BA11-AL 1 ||                    |            ||BA11-AM 1 ||
||UBA 0     ||                    |            ||UBA 1     ||
||----------||                    |            ||----------||
|            |                    |            |            |
|            |                    |            |            |
|------------|--------------------|------------|------------|


|------------|  |------------|  |------------|  |------------|  |------------|
|H9642-AR 1  |  |H9642-AR 2  |  |H9642-AR 3  |  |H9642-AR 4  |  |TU81-AB 1   |
||----------||  ||----------||  ||----------||  ||----------||  |            |
||RA60-AA 1 ||  ||RA60-AA 1 ||  ||RA60-AA 1 ||  ||RA60-AA 1 ||  |            |
||          ||  ||          ||  ||          ||  ||          ||  |            |
||----------||  ||----------||  ||----------||  ||----------||  |            |
||RA60-AA   ||  ||RA60-AA   ||  ||RA60-AA   ||  ||RA60-AA   ||  |            |
||UNIT # 4  ||  ||UNIT # 3  ||  ||UNIT # 2  ||  ||UNIT # 1  ||  |            |
||----------||  ||----------||  ||----------||  ||----------||  |            |
|            |  |            |  |            |  ||RA60-AA   ||  |            |
|            |  |            |  |            |  ||UNIT # 0  ||  |            |
|------------|  |------------|  |------------|  ||----------||  |------------|
```

Figure 4.11: The rest of the report generated by XCON.

4.7 Learning Rules in SOAR

Learning has always been a weak point of symbolic rule-based AI programs. A notable exception to this has been a program called SOAR.[6] SOAR was originally designed to be a heuristic search system with the initial goal to design as general purpose a program as possible to do as many problems as possible. Normally then, SOAR is discussed in terms of its searching ability, however, in this section we will look at the ability of newer versions of SOAR to learn rules as it searches. Some experiments with SOAR show that its performance gives the Power Law of Practice results first discovered in studies of human learning.

4.7.1 A Searching Example

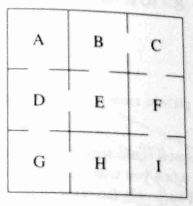


Figure 4.12: This figure shows the layout of some rooms and the task will be to find a way to go from room A to room H.

For a simple example of searching, suppose we have the problem of trying to get from one particular room to some other particular room.[7] Figure 4.12 shows the layout of the rooms. Suppose the goal is to move from room A to room H. There are very few choices here and the tree in Figure 4.13 shows the possible moves a searching program could make. At room A the only room you could move to is D. At D you have two choices and since you do not have any heuristics to lead you in the right direction you could choose to go to either room E or G. Suppose you move to E. From there on there are not any choices and you have to move along to B, C, F, and finally I where there are no alternatives left. This is a good time to back up and try moving from D to G. At G the only move you can make is to go to H and that solves the problem.

The amount of learning you can do in this example is extremely small. The only thing worth learning is:

IF you are in room D and the goal is H
THEN go to room G

[6] The SOAR homepage at: http://www.cs.cmu.edu/afs/cs/project/soar/public/www/home-page.html contains information on SOAR including binaries for Intel and Macintosh systems.
[7] This example is adapted from an article by Smith and Johnson [216].

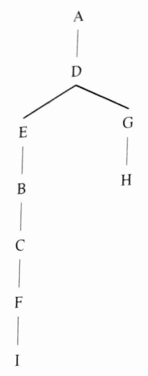


Figure 4.13: The search for a way to get from room A to room H is straightforward.

Notice that there are various other things you could remember from this experience like when you were in A you can go to D or when you were in B you can go to C, however, unlike people, SOAR attempts to find and keep only the relevant memories, not memories that are unnecessary. In SOAR, the relevant memories are typically called rules, but they might easily be called relevant memories. These rules are also called *chunks* and the process of forming them is called *chunking*.[8]

SOAR has been able to solve an impressive array of problems from simple puzzles up to real world type tasks. One system learned the MYCIN rules [251][9] while another learned some of the hardest parts of the R1 VAX computer configuration task [95].

4.7.2 The Power Law of Practice

In a work published in 1926, Snoddy [219] showed that the time, t, it takes for a human being to trace geometric figures in a mirror depends on the amount of time, N, used to practice the task. The exact relationship was given by the formula:

$$t = bN^{-\alpha}$$

where b and α are constants that have to be determined experimentally. This formula says that the time it takes to do the task will initially be quite long with little practice (a small value for N), but as the task is practiced more, the time to do the task becomes shorter.

[8] The terms chunk and chunking come from the research of Miller [122], however, it is not clear if these chunks are exactly the same as Miller's chunks.

[9] Code is available at: http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/areas/planning/systems/soar/neomycin/0.html.

Since then, other researchers have found this same power law of practice holds for many other activities as well such as recalling facts, editing with text editors, checking proofs, and playing solitaire [140]. Newell and others have shown that SOAR programs also produce this power law of practice [141]. In essence, what happens in a SOAR program at the start of learning is that searching takes up quite a lot of time, however, as the chunks build up, the searching time decreases because previously saved results minimize the need to search. Because SOAR programs also generate the power law of practice results, Newell believed that searching and chunking were perhaps the most fundamental aspects of intelligent behavior.[10]

4.8 Rules versus Networks

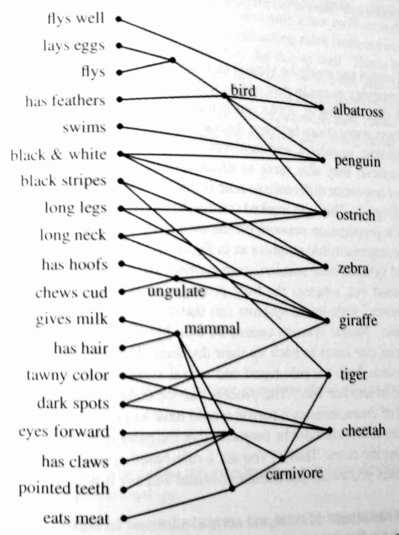


Figure 4.14: The rules for identifying animals can be put together to form a large, sparse, feed-forward network. The weights and thresholds for the nodes are not shown. The bird, ungulate, mammal, and carnivore nodes are OR nodes while the rest are AND nodes.

layer	unit	unit value	weight
2	1	0.94134	-3.09056
2	2	0.00147	-0.38760
2	3	0.42901	-4.29914
2	4	0.66736	-0.28191
2	5	0.02098	-0.96560
2	6	0.00156	-2.02424
2	7	0.61722	-0.62367
2	8	0.01811	7.90404
2	9	0.05367	0.51293
3	b	1.00000	-0.78887

Figure 4.15: This figure shows some of the weights in a 19-9-7 backprop network trained to do the animal identification problem. In particular, these are the weights leading into the first output unit. Unlike a rule-based network, there is no easy way to determine the meaning of the weights. For that matter, the "hidden units" in the rule-based network also mean something. The unit value column above lists the hidden unit values in this network and there is no simple interpretation for this collection of values.

As noted in Chapter 1, rules can easily be viewed as small neural networks. If we take the rules in the animal identifying example they can be assembled into the one larger network shown in Figure 4.14 that, of course, looks much like a conventional back-propagation network but with far fewer connections between the nodes. The introduction of hidden units to recognize birds, mammals, ungulates, and carnivores can cut down on the overall amount of processing and of course, they also serve to divide animals into groups with similar characteristics. Another important difference to note is that the units and their connections can easily be stated in English. This is in marked contrast to the type of network you would get from training a back-propagation network on the animal data where you end up with only a collection of incomprehensible numbers as in Figure 4.15. Clark [18] has said that the symbolic rule-based systems have *semantic transparency* because the meaning of each component is easy to read out, whereas the back-propagation type of network does not have semantic transparency. Rule-based systems can therefore quote the rules they used in reaching their decision. Neural systems cannot do this, but if the database of training data is available, they can cite cases to back up their decision. In some applications like, say, a loan approval system, both the rule-based and neural systems can give reasons for rejecting an application in another way. The process can be done by varying the inputs and showing what kind of characteristics a person would have to possess in order to have their loan approved. Then too, it must not be forgotten that the rules in a rule-based system ultimately came from specific cases. Thus, if you ask a rule-based system why it made the decision it did and it quotes you a rule, you should go ahead and ask it where the rule came from!

It is hard to do a fair assessment of rules and networks for use on a given application. For one thing, there are very few expert system projects that have been implemented both

[10] See [141], page 96, however, Newell also noted that an argument could be made that pattern match processes were equally fundamental.

ways. For another, it is also possible to do a poor job with one version and a better job with the other and then the results do not mean much. To get fair results from a humanly built network of rules, the expert system creation process should be done independently by a number of different teams and this would be a very expensive experiment to do. Also, the results you get from backprop networks can often vary by quite a lot and there are many techniques that can be applied to improve the generalization of a network. To get fair results for backprop networks you really have to try all these options.

In one report, Saito and Nakano [185] compared a symbolic medical diagnostic expert system with a back-propagation-based one. The network was trained on only 300 cases involving headaches as the only symptom and with 23 possible diagnoses. On a set of test data the network was correct 67 percent of the time versus 70 percent for the symbolic system.

In another experiment, Bradshaw, Fozzard, and Ceci [13] report on two systems to predict solar flares. Both the symbolic and back-propagation versions predict solar flares equally as well as human forecasters. One big difference was that the symbolic system required over one man year of work (over 700 rules) while the network was developed in less than a week. A second big difference is that the symbolic system takes five minutes to do a single prediction while the network takes only a few milliseconds.

There have also been experiments in extracting rules from networks. In the Saito and Nakano experiment mentioned above they did this as well. Thrun has also experimented with extracting rules [234].[11] Another approach by Shavlik and Towell [237, 204][12] is to start with rules, transform the rules into a backprop network, train the network, and then extract rules from the network. In this experiment they found that the final rule set performed better than the network and the network performed better than the initial rules.

4.9 Exercises

4.1. Given data such as:

```
father(john,mary).
father(john,ted).
father(ted,larry).
father(ted,bill).
mother(mary,alice).
mother(alice,carol).
```

Define a rule that will be able to answer questions about who is the grandfather of whom, such as this one:

```
?- grandfather(john,X).
```

Also give rules that will define the grandmother, great grandmother, aunt, uncle, brother, sister, and sibling relationships.

4.2. When the factorial predicate is given the problem: fact(3,A), it reports back that A = 6. What happens if you type a ";" now, instead of a carriage return?

[11] For related publications see: http://www.informatik.uni-bonn.de/~thrun/publications.html.
[12] For related publications see: http://www.cs.wisc.edu/trs.html.

4.3. Write a Prolog program to find the *n*th Fibonacci number.

4.4. For the following structures in list notation, give the structure in the notation that uses . as a functor that composes trees. Then diagram the trees.

```
[a, [b, c]]
[[a, b], [c, d]]
[[c, [d, [e, f]]], a, b]
```

4.5. Write a Prolog function that will determine if all the members of a list, L, are also members of a list, X. For example, are all the members of the list, [b,1], present in the list, [a,1,b,c,x,4,z]?

4.6. Write a Prolog function, vowel, that will take a list of letters and return a list of all the vowels in the original list. For example:

```
?- vowel([c,h,a,p,t,e,r],X).
```

should give:

```
X = [a,e].
```

Also write a function, consonant, that will return all the letters that are not vowels.

4.7. Write a Prolog function, nodupl, that will remove duplicate entries from a list. For example:

```
?- nodupl([h,e,l,l,o,w,o,r,l,d],X).
```

should give:

```
X = [h,e,l,o,w,r,d].
```

4.8. Here is a function, append, that runs its first two arguments (lists) together to form a new list in the third argument:

```
append([],L,L).
append([A|L1],L2,[A|L3]) :- append(L1,L2,L3).
```

So, for instance:

```
?- append([a,b],[c,d],X).
```

gives X = [a,b,c,d]. Trace through the above function call giving all the intermediate results.

4.9. In the text we programmed simple forward and backward chaining interpreters to handle the rule base for the animal identification program. It is a little simpler to let Prolog do the interpretation by giving it the rules about animals directly, such as in:

```
albatross(X) :- bird(X), flieswell(X).
bird(X) :- hasfeathers(X).
bird(X) :- layseggs(X), flies(X).
```

In the first case, this says that X is an albatross if X is a bird and X flieswell. Write a Prolog program that will do the animal identification problem using backward chaining. Write a second program using data in this format that will use forward chaining.

4.10. Program the forward and backward chaining versions of the animal identifying problem using whatever language is available and convenient. For the backward chaining version, have the program ask questions about whether or not an animal has a certain characteristic if there is no rule available to deduce that the characteristic is true. Make your program ask a question about a given characteristic only once. For instance, if early in the identification process the program asks if the animal has feathers and the user says no, then have the program remember this fact so that it will not ask it again.

4.11. Suppose you are doing the animal identification problem and you have to deal with incomplete data about the unknown animal and you want to ask the user more questions about the animal. How well will the MYCIN activation function work on this problem?

4.12. Suppose you have the data about the seven animals in the animal identification problem as vectors. Find out how well a simple nearest neighbor approach using Euclidean distance will perform at identifying unknown animals when one or two characteristics of each animal are omitted from the description of the unknown animal. Compare this with the results you get from a back-propagation network.

4.13. Hilary Putnam, a skeptic with regards to the accomplishments of AI, has said that expert systems are "just high-speed data-base searchers [165]." Is this true? If it is true, is this bad?

4.14. The rules at the end of Section 4.5 for packing Christmas tree ornaments do not consider what happens if:

1) you run out of large ornaments to be packed,

2) you run out of small ornaments to be packed.

Write a more complete set of rules to take these possibilities into account. Do not neglect the case where a box may end up with less than six large ornaments so that there is more room for small ornaments. Assume three small ornaments occupy the same space as one large ornament. You can invent new procedures and variables as necessary. If you know Prolog or Lisp then you may want to program this system with one of these languages. Another possible way to do this problem would be to train a number of cases into a back-propagation network and let the network choose which move to make. How well would this work?

4.15. Instead of having a rule interpreter, why not just write IF-THEN statements in a general purpose program in a language such as C? Consider the impact this would have in doing problems in:

1) a forward chaining manner,

2) a backward chaining manner, and

3) what happens with rule conflicts.

4.16. Get a copy of a book for identifying wildflowers, such as: *A Field Guide to Wildflowers of Northeastern and North-Central North America* by Peterson and McKenny. Produce an expert system to identify *as many* wildflowers as is possible and convenient. Produce a convenient interface for the system so that users can type in characteristics of the flowers as words. In case the user gives an insufficient number of details about an unknown flower, arrange to have the program ask questions about other possible characteristics. If a person submits an unknown flower that does not match any flowers that the program knows about, have the program say it is not sure and have it indicate which known flower the unknown is closest to. There are many ways to do this problem. Choose one or more of the following methods to implement this system. In a class, you may want to assign different students to program different methods and then the results can be compared. Some of the factors that should be used to compare the methods are the ease of programming, size of the code, and execution time. Here are the methods:

a) Use one large back-propagation network.

b) Try using more than one network to learn all the flowers. This gives you the problem of dividing up the flowers among the networks in some appropriate way.

c) Use the Euclidean nearest neighbor scheme or a nearest neighbor scheme based on the number of characteristics the unknown has in common with flowers in the database.

d) Find out how well this variation on a nearest neighbor scheme works: use a nearest neighbor scheme to locate the *n* nearest neighbors to the unknown flower, where *n* is perhaps 5 or 10. If there is a perfect match, let this be the answer, otherwise train these *n* candidates into a back-propagation network and submit the unknown to this network to find out the most likely answer.

e) Use forward and backward chaining in Lisp, Prolog, or some other convenient language.

f) Use any other promising method(s) you can think of.

Again, try to use as many different flowers as is possible and convenient. Note the advantages and limitations of each method as the number of flowers in the system grows. If you do not have time to program any of these methods, you could still try to evaluate the methods without doing any programming. Also, if you do not want to identify wildflowers, you can evaluate the methods for any other application area you can think of.

4.17. One major reason for a company to produce an expert system is to capture the expertise of its human experts who may retire or quit. How else could a human expert transfer his expertise to other people? How was this done before computers?

Chapter 5

Logic

As was stated earlier, Prolog uses a subset of a type of logic called the predicate calculus. The full predicate calculus is much more powerful than Prolog. In this chapter we want to look at the notation and capabilities of predicate calculus and compare them with Prolog. In Prolog, the user asks questions about the database of facts and rules. In predicate calculus the questions are regarded as theorems to be proved so the subject is often called automatic theorem proving. The subject is also often referred to as automated reasoning. The notation we will use comes from a public domain automated reasoning program called Otter[1] with Prolog style variables. Some examples are taken from a collection of problems called the "Thousands of Problems for Theorem Provers" (TPTP) collection of Geoff Sutcliffe and Christian Suttner.[2]

5.1 Standard Form and Clausal Form

Predicate calculus is a form of logic that can express facts about a problem and that uses specific rules to reach valid new conclusions. Of course some of the standard relations in this logic are the familiar "and," "or," and "not," but it also includes the relations "implies" ($\rightarrow$), "is equivalent to" ($\leftrightarrow$), and the quantifiers for variables, "for all" ($\forall$) and "there exists" ($\exists$). One notation in predicate calculus called *standard form* uses all these symbols, however, statements written in this form can be translated into another form called *clausal form* where these latter four symbols are not used. It turns out that clausal form is much more convenient for computer manipulation than standard form so that is the form we will be using here, except we will begin by showing how "implies," "is equivalent to," "for all," and "there exists" can be eliminated from predicate calculus statements.

In the way of notation, we will be using '&' for 'and,' ' | ' for 'or,'[3] and '–' for 'not.'

[1]Otter is from Argonne National Laboratory. It includes C code, a 32-bit DOS binary, a Macintosh version, user manuals in various formats, and some sample problems. It is available at http://www.mcs.anl.gov/home/-mccune/ar/otter/index.html and ftp://info.mcs.anl.gov/pub/Otter.

[2]This collection is rather large, a gzipped file of 1M which, when gunzipped comes to 18M. In all likelihood the only problems you will ever want are the ones in the puzzles directory. This package is available from ftp://flop.informatik.tu-muenchen.de/pub/tptp-library, http://wwwjessen.informatik.tu-muenchen.de/~suttner/tptp.html, ftp://coral.cs.jcu.edu.au/pub/research/tptp-library, and http://coral.cs-jcu.edu.au/-users/GSutcliffe/WWW/TPTP.HTML.

[3]If predicate calculus used lists as in Prolog you might get confused between the two different uses for | , however the only use for | in this chapter will be as the 'or' symbol. It is quite common to use ∨ for 'or,' however Otter uses | not ∨.