

ARTIFICIAL INTELLIGENCE – OR NOT ?

by

G. Alan Creak

(alan@cs.auckland.ac.nz)
(<http://www.cs.auckland.ac.nz/~alan>)

Technical Report 169

Computer Science Department,
Auckland University,
New Zealand

1999 December 22

CONTENTS

Contents		page 2
Foreword		page 3
CHAPTER 1 :	Introducing "intelligent" machines	page 4
CHAPTER 2	First example : Symbolic methods	page 6
CHAPTER 3	Second example : Cellular Automata and Genetic Algorithms	page 10
CHAPTER 4	Third example : Neural networks ("connectionism")	page 16
CHAPTER 5	Conclusion ?	page 22
References		page 31

FOREWORD

The material presented here was conceived, in a manner of speaking, on 1st September 1994, when a letter came forth from the university's Centre for Continuing Education in the direction of me and a few other people. It conveyed a proposal that there should be a "trans-disciplinary conference" entitled "Intelligence, Free Will And Behaviour : Is Humanity so Special Anyway ?" (what's wrong with "so" that it alone should be confined to lower case ?), and an invitation to take part. The date was set – auspiciously or suspiciously – as 1st April 1995. Who could resist such a beguiling opportunity ?

I could. I was not short of things to do, and I had tangled with the Centre for Continuing Education a year or two earlier, with unedifying results. Nevertheless, I spoke to the luminary who had sent the letter, and allowed myself to be persuaded. Later in September, I sent him a summary of my proposed talk, whereupon nothing happened.

Until 14th November. On that day, the luminary sent both electronic mail and a letter, curiously dated "14 Monday November", enclosing (with the letter) copies of other participants' summaries and the information that he was leaving the country permanently on 16th November, and identifying a second luminary who would take over the administration for the event. There was also the rather surprising information that one of my research students (David Norman, who appears again later) would cover the event for the press. On enquiry, David explained that luminary 1 was a contemporary and close friend, and that was a surprise too, for I'd expected that events of the sort suggested would be organised by someone significantly more experienced.

I never heard from luminary 2. The next communication was from luminary 3, on 14th March 1995. By this time, I had put in quite a lot of work on the talk, and – my expectation of non-shortage of things to do having been, if anything, more than fully justified by events – had taken some days I could ill afford to make sure it was properly finished. I had also begun to feel restive about the lack of communication from the Centre for Continuing Education (surely someone should be worrying about publicity material or handouts or something ?), but had done nothing about it because of my permanent state of frantic activity.

Luminary 3's letter, addressed to me at Computer (sic) Science, said that the event (now a mere seminar) had been postponed indefinitely. My misgivings about 1st April had clearly been well founded. It would be managed by yet another luminary, who would be in touch "towards the end of Term 11" (also sic). Luminary 4 did indeed get in touch, in April, and eventually set a new date for the event : 22nd July. This time, he got as far as sending a contract to be signed and returned.

And he got in touch again, in June, to say that the seminar had been cancelled because of a lack of enrolments. In his confirmatory letter, he said that he hoped "to offer the seminar again at a later date". Fat chance. And at around that time this report, springing phoenix-like from the ashes of both seminar and trans-disciplinary conference, started life – appropriately, just about nine months after its conception.

It is not clear to me whether the metaphors of the phoenix and nine months' gestation are compatible, but there had been some rather early stirrings of life around April and May which could perhaps be used to choose between them, if one only knew more of the life-cycle of the phoenix. At that time, feeling fairly disgusted with the Centre for Continuing Education, I'd presented a version of the seminar to the Computer Science Department under this facetious banner :

YAPOOMI – Yet Another Personal Opinion On Machine Intelligence.

I shall present and discuss, rather briefly, three computer programmes which use different techniques to learn from experience. I shall present an argument which suggests that the three techniques cover the whole possible range, and conclude *either* that the prospect for machine intelligence looks bleak, *or* that we cannot escape symbolic methods. Finally, I shall present a view which makes the rest irrelevant anyway.

This report is, in effect, the notes of that seminar, tidied up, and with some material added or expanded to fill in the gaps left by the disappearance of expected context from a whole trans-disciplinary conference.

CHAPTER 1

Introducing "intelligent" machines

Can machines think ? That is hardly an original question, and it has certainly been ardently discussed since we first began to build machines which could, just occasionally, present a vestige of a suggestion of a hint of intelligent behaviour.

I shall not attempt to present my ideas on the subject in the context of the continuing philosophical debate. I choose this approach not because I believe that the philosophers are too clever for me – false modesty is not one of my failings – nor because I believe that I am too clever for them – though true modesty is not necessarily one of my virtues. I present these thoughts as a purely personal view because I don't know enough about the continuing philosophical debate to comment on it. It is therefore quite likely that I shall match the hardly original question with a hardly original answer; if so, I'd quite like someone to tell me about it.

More specifically, I shall offer an opinion on a perhaps different question : are machines intelligent ? I am not sure what the difference between the two questions is, but it seems clear from my dictionary¹ that my uncertainty is indefensible, as the two notions have nothing in common at all :

intelligence, *n.* The exercise of the understanding; intellectual power; capacity for the higher functions of the intellect; acquired knowledge; quickness or sharpness of intellect; ...

think, *v.t.* To regard or examine in the mind, to reflect, to ponder (over, etc.); to consider, to be of opinion, to believe; to design, to intend, to effect by thinking; ...

(There is more to both definitions, but I've played fair : neither includes any mention of the other's keyword.) It is true that intellect is defined as "The faculty of the human mind by which it receives and comprehends ...", while mind is defined as "The intellectual powers in man; the understanding, the intellect ...", so there's a link (albeit confusingly recursive) somewhere, but I am not sure that any of that helps me a lot.

Fortunately, this has no bearing on my decision to address the intelligence question. The relationship between intelligence and thinking might be confused, but the question about intelligence fits my answer better. In any case, according to the literature, the question about thinking has been settled² : "Can a machine think ? The answer to this old chestnut is certainly yes ...". If that's so easy, though, why do we hear so much about artificial intelligence, or machine intelligence, but very little about artificial thinking ?

I shall approach the question obliquely. I shall demonstrate that – at least for as long as machines are designed by people – we can (subject to the reliability of the Church-Turing thesis³) continually improve their intelligent behaviour. Then, in an elegant dénouement, I shall suggest that this has nothing to do with intelligence anyway.

- AND THIS IS WHERE THE STORY REALLY STARTS

To illustrate my point, I shall describe three approaches to a very elementary task – learning from experience. Does that amount to intelligence ? It's hard to say : *not* learning from experience is regarded as stupid, but that doesn't mean that learning is necessarily regarded as intelligent. On the other hand, it's normal human behaviour, and we certainly expect intelligent people to learn things, so there's presumably some connection. As an example of intelligence, learning has one advantage over other possible tasks : it is open-ended, in the sense that, though we have to set up a machine so that it is likely to learn, we can avoid predetermining exactly what it will learn, so we can be surprised. We'll accept it in default of anything else.

My three examples illustrate three significantly different sorts of learning, each associated with different computing techniques. These are :

- "SYMBOLIC" PROGRAMMING conventional computer programming.
- CELLULAR AUTOMATA and GENETIC ALGORITHMS evolving intelligence following the patterns of biological evolution ?
- NEURAL NETWORKS modelling the brain – electronic brains ? (at last !)

I didn't select these examples with great care; they are three rather different examples of learning behaviour which I've encountered in some of the various activities in which I've engaged from time to time. It happens that they illustrate three quite different approaches to learning, which is appropriate in attempting to answer a general question.

After discussing the examples, I'll return to the general considerations again. I shall then suggest that, though different, the examples in one sense fall into a single pattern, which exhaustively covers the possibilities for an intelligent machine, so far as machines that we can design are concerned. This pattern is forced upon us by our human limitations, and restricts us to some form of symbolic argument, no matter what building blocks we might choose for our systems.

CHAPTER 2

First example : Symbolic methods

Symbolic methods are those we use when we argue consciously; they are the stuff of logic and debate, mathematical proof and detective stories. We use symbols (typically words, or mathematical notation, or objects in diagrams) to represent the things we are discussing, and manipulate them according to formal schemes. Roughly speaking, symbolic methods correspond to methods we understand.

Computer programming lies comfortably within the class of symbolic methods. What's often called (usually disparagingly) "traditional artificial intelligence" is largely expressed in terms of symbolic arguments and implemented in computer programmes. A learning programme is constructed by following a procedure something like this :

1 : THINK ABOUT LEARNING : Introspection.

The first step is to decide what learning is. Must we be guided by psychological models, or can we go it alone ? What does learning feel like ? How do we expect that a system will change as it learns ? Is it a matter of simply storing that which is to be learnt in a file somewhere, or can it be done by making a few changes to some simple data structures, or is there more to it than that ? We have to make these decisions first, because without them we can't decide how to proceed in the later steps – or, to put it another way, once these decisions are made the problem of learning is reduced to a problem of computer programming. It is here that we define the symbols which make this a symbolic approach, and describe their behaviour.

2 : WORK OUT A PLAUSIBLE MECHANISM : Ingenuity.

Now we must decide in detail how learning works – or, at least, how it plausibly could work. The choice of mechanism will be guided by the model we constructed in the first step. It might be a psychological model, or it might be the result of introspection, or it might be a set of data structures which we believe will show the required behaviour. In any case, we must convert the model into a precise description of how memories are stored and manipulated.

3 : WRITE A COMPUTER PROGRAMME : Interpretation.

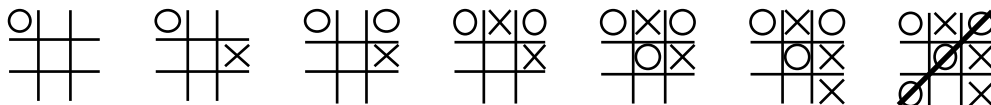
Now we have a full specification. How can we simulate it in action ? We choose a programming language (or, more commonly, just use the one we know best), and write a programme which we know will behave in the way we have specified.

I have spelt that out at some length to emphasise that every detail is determined by us. In that sense, the system can't do anything new, though it can do its predictable things in ways we didn't expect. Even that, though, is more a matter of our limited foresight than of real novelty generated by the system.

Whether or not the system behaves in any sense like we do depends completely on the first step, and in particular on whether what we think we do when we think really is what we do when we think. If it isn't, then we really do have **Artificial** Intelligence – or, at least, artificial something.

THE EXAMPLE.

My example of symbolic intelligence is a programme which learns to play noughts-and-crosses. (I understand that "tic-tac-toe" is another name for the same game.) Just in case anyone doesn't know the rules, here's an illustrative game :



There are two players. Each player is assigned one of the marks, nought or cross. Nought begins, and the players take turns thereafter. At each turn, the active player puts his mark in one of the vacant squares of the grid; a player who completes a line of three of his own marks wins. The game ends when one player wins, or when the grid is full. I have also assumed that a player can resign.

I invented it (the programme, not the game) in (I think) 1970; it seems very likely that many other people have invented it before and since, so I claim no precedence. If you sit and think about the problem for half an hour or so, the method I used is the obvious way to learn noughts-and-crosses – or

practically anything that falls in the same class. (That's not to say that it's a very practicable method in all cases; as the programme operates by, in effect, making a representation of the complete game tree, chess would take a little longer, and go would never stop.)

It works by following this very simple algorithm :

Remember the last position you moved to by choice, and, if you lose, take care never to get into that position again.

The argument is that anything which happened since you last made a choice was forced upon you, so, no matter how far in the past the choice was taken, that was your last chance to avoid defeat.

In more detail : You (you being the programme) begin with a knowledge of the geography of the board, a knowledge of the permissible moves, a memory which can accommodate one image of the board, and a list of bad positions, which contains no bad positions. You don't know the rules of the game, but there is a referee who will tell you when it is your turn to move, and when you win or lose. Then, at each move, follow these instructions :

- 1 : Count the possible moves. (That's the number of empty squares on the board.) If there is no possible move, the game is over, and was an honourable draw; go to step 5.
- 2 : Consider, one by one, the results of making each of the possible moves. Check the potential new position against your list of bad positions, and eliminate it from consideration if it is on the list.
- 3 : Count the number of moves which haven't been eliminated. There are three possibilities :
 - There is no move left. All possible moves are on your list of moves to avoid, which implies that your opponent can force a win whatever you do. Resign, and add the last remembered choice to your list of bad moves, for there is a forced win for the opponent from that position.
 - There is just one move left. Make it.
 - There are several moves left. Choose any, and record in your memory the state of the game after making the move as the most recent choice. (Any previous memory is overwritten by this action.)
- 4 : Await the next pronouncement from the referee. There are three possibilities* :
 - "It is your move" : go back to step 1.
 - "You have won the game" : apologise to your opponent for your sheer good luck.
 - "You have lost the game" : congratulate your opponent on his greatly superior skill, and add the last remembered choice to your list of bad positions.
- 5 : Forget everything but your list of bad positions, set up the board for a new game, and go back to step 1.

This can be considered an example of rote learning, but it's learning nonetheless. It is hardly sophisticated learning, but it's close to a model of learning which works with a dog : hit it when it does wrong. Even with a dog, it isn't a recommended teaching method – it's better to encourage the dog when it does right – but in the noughts-and-crosses case there's nothing to say that a move is right, so we can't do that; we're reduced to learning from mistakes. That's an example of the *credit-assignment* problem : if you're only

* These are the old British rules. Those who are neither old nor British may prefer more histrionic versions of the second and third possibilities.

told the result at the end of a long sequence of actions, how do you work out which of the actions was important in producing the result ?

IS IT INTELLIGENT ?

(Before thinking about that question, forget that you know how the machine works; everything is simple when you know the trick !)

It works. It's rather slow, but you can never beat it in the same way twice, and now and then – more frequently as it becomes more expert – it surprises you by knowing how to react when you are using what you thought was a new approach. And if you keep on for long enough, it will – given a wide variety of games from its opponents – become perfect. (Or it should. I never had the patience to keep trying for sufficiently long.) That's not a bad trick.

But we can do better than that. Indeed, we can devise a way to keep on doing better until we reach a state of real intelligence – or for ever, whichever happens sooner. Consider this dialogue :

Me :	I have constructed this intelligent machine. Please test it.	
	<i>Pause for testing.</i>	
You :	I have done so.	
Me :	Do you agree that the machine is intelligent ?	
You :	<i>EITHER</i>	Yes. (<i>STOP – I have succeeded.</i>)
	<i>OR</i>	No – That machine isn't intelligent !
Me :	How do you know ?	
You :	Because it doesn't < <i>do something</i> >.	
	<i>Lapse of time, then repeat.</i>	

The point of that algorithm is to force you either to admit that I have written an intelligent programme, or to identify some specific reason why the programme isn't intelligent in such a way that I can remedy the perceived defect. I don't accept vague answers ("It sort of doesn't behave intelligently"), but press you for something more specific. If you can't find anything more specific, then I claim that you are fudging the issue, and inventing imaginary "reasons" for denying that my machine is intelligent. If you do come up with something reasonably precise ("It doesn't look ahead", "It takes no account of the symmetry of the board", etc.), I occupy the *lapse of time* by making it < *do something* > which corrects the defect which you identified. After so doing, I bring you an improved version which meets your objection. For the two reasons suggested as examples, the *lapse of time* will be quite short, and the improvement is very marked; more complicated reasons might take a little longer, but I claim that I'll be able to do it provided that you give me a sufficiently precise definition of your objection.

I am rather confident about that, because of the Church-Turing thesis³. That's an unprovable assertion, but one which is intuitively reasonable and to which we have so far discovered no exception : it can be roughly expressed as "if you can describe a way to do it, a computer can do it". I'll keep pushing you to give more details until I can see how it could be programmed, then I'll stop pushing and confidently expect to be able to do the programming.

I thought that I'd invented that argument, but perhaps I was wrong. (This is not an uncommon discovery.) This passage is taken from "Programming for poets"⁴, an interesting and often reflective introduction to programming :

Statement : A computer can certainly never exhibit property X.

Response : Precisely what do you mean by property X ?

– followed by a discussion not unlike mine. In defence I offer my assertion that though the book has been on my shelves for some years, I didn't read it until well after writing the body of this report.

It's interesting to look a little closer into the two objections which I suggested. In both cases, there is a sense in which you, the objector, are expecting the machine to do something which it cannot do because of its limited nature. As we, people, learn to play noughts-and-crosses, we eventually begin to generalise; instead of thinking at the level of individual moves and positions, we recognise that a line of three similar symbols is a significant abstraction, and we begin to use this idea in our planning. But the

noughts-and-crosses programme can't see the line of three; it has no way to perceive an almost complete row, for example, and to expect it to learn to do so is about as sensible as expecting a car to grow wings when you drive it over a cliff. If the abstraction is what you want, then I can build it in and make it work.

This sort of thing is the essence of symbolic programming. It depends on our devising a possible mechanism for the effect we want, whereupon we can programme it.

A CURIOUS OBSERVATION – OR PERHAPS NOT ?

Did you notice an interesting fact about the two algorithms which I've discussed in this chapter ? I did, and it took me only several years to do so. It is that the two are essentially identical, both conforming to this pattern :

```
Repeat
  Do the experiment;
  Find what went wrong;
  Take steps to prevent it in future
until happy.
```

I was very impressed with my discovery, and saw it as a deeply significant event.

On reflection, I became less impressed. Consider the quite general problem of how to improve the performance of anything. Either we know how to do it (we have some relevant theory, or someone to ask, or some other source of existing knowledge about the problem), or we don't. If we know how to do it, there is no problem (in principle, if not necessarily in practice); if we don't know how to do it, all we can do is try to find something which we do know how to handle – or we can just make sure that the problem doesn't arise again by avoiding it. If we don't take avoiding action, we must perform some sort of experiment, and consider what happened. If nothing went wrong, we perform another experiment. If something went wrong, then perhaps it was in a part of the system which we do understand, so we know what to do; otherwise either we perform further experiments to try to probe the details until we find something which we can understand, or we can simply put a fence round the area so that it doesn't happen again.

But that's what I just said, a couple of sentences ago.

That isn't a very tidy description, but perhaps it makes the point that if we don't know what to do, whether it's playing noughts-and-crosses or creating artificial intelligence, we have no choice but to follow something very like the pattern which I thought I'd discovered – there is simply no other way to do it. Further, the only steps we can take are either to avoid the problem, or to investigate further until we find something we know how to handle. That's all we can do; and it's also all our machines can do, so unless they have means to investigate and knowledge with which to handle, they can only avoid. Search techniques are the simplest possible (I think) sorts of "means to investigate" – just try whatever's available.

And I don't know whether that's profoundly significant or trivially obvious. Or both.

CHAPTER 3

Second Example : Cellular Automata and Genetic Algorithms

Once we abandon symbolic methods, there is a sense in which we lose control. That's not to say that we shall build machines which run riot; on the contrary, we might be able to predict precisely every individual change in the machine's configuration. What we give up is our knowledge of what the changes mean, and how the machine uses them to perform in the way we intend. In the noughts-and-crosses example, we have an explicit representation of the board, and of the memory of bad positions, and of how the system uses the memory to avoid bad moves. In the examples described in this chapter and in the next, we provide machinery which the system can use in order to generate certain sorts of behaviour, and methods which encourage it to move towards the desired behaviour, but we prescribe only general features of the movement without defining details. The result is machines which evolve their own representations of the knowledge which they use, and these representations do not necessarily fit neatly into our conventional ideas of how to do things.

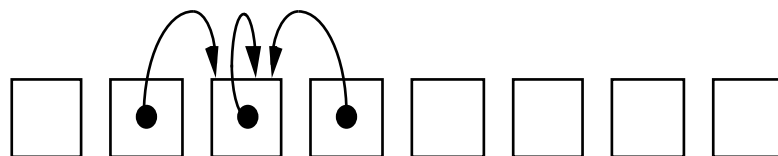
In this chapter, we use a cellular automaton as a machine which can in a rather primitive way be programmed to accomplish a simple task, and we use a genetic algorithm as a means of developing a programme for the automaton.

CELLULAR AUTOMATA.

A cellular automaton is a machine composed of a number (possibly a large number) of identical cells arranged in a regular array. There are many possible variations on this theme, but the most common, and the only one I'll consider, is this : each cell is a processor with no internal state, except for a single output register. The processor's inputs are the outputs of all cells in its *neighbourhood* (a finite set of local cells), and its output is a function of its input determined by a *rule*. The rule is just a list of all possible input combinations, each associated with the corresponding defined output; all cells of the automaton follow the same rule, and all cells compute their new values in synchrony.

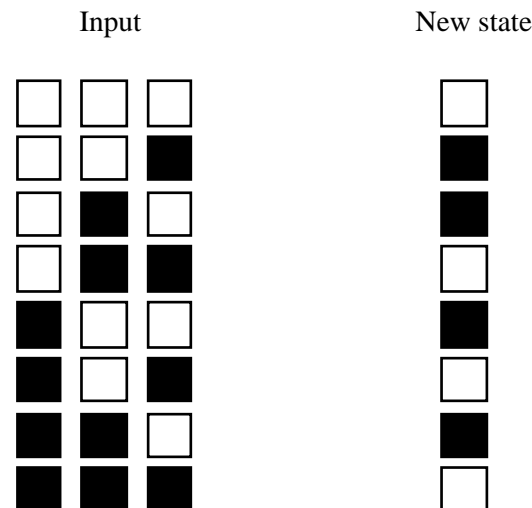
The best known cellular automaton is undoubtedly the "Game of Life". This is a two-dimensional automaton, each cell of which can see itself and its immediate neighbours. A pattern imposed on the set of cells will evolve as the cells compute successive outputs from their changing states which they receive from their local environments, but the automaton itself doesn't change, so, in particular, it doesn't learn. That's why we need the genetic algorithms; but before describing those there is more to say about the cellular automata.

I shall discuss one-dimensional automata, because they're simpler. Here's a picture of a one-dimensional automaton :



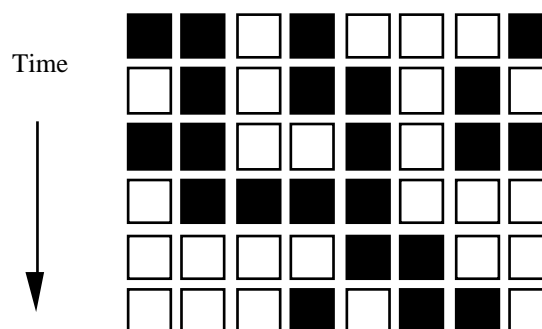
It has eight cells, which I shall always assume to be arranged in a ring in order to avoid end effects – so the cell drawn in the leftmost position is considered to be the right neighbour of the cell drawn in the rightmost position. This assumption makes no difference to the arguments. The arrows are intended to suggest that the third cell from the left is influenced by itself and by its two neighbours; we say that it has a neighbourhood of three cells, which I assume to be symmetrical. (If it isn't symmetrical, the only change to the behaviour is that the patterns shift systematically in one direction or the other at each step.) At each step, therefore, the third cell from the left computes its new state from the states of the three cells in its neighbourhood, and all the other cells do likewise.

Further to simplify the system, I shall assume that each cell has just two states, which I shall call *black* and *white*. A rule for the cells might be defined like this :



It will immediately be apparent to alert readers that I have drawn the rule in such a way that the list of inputs is redundant; with an obvious interpretation of black and white, the input column is simply the integers from zero to seven written as binary numbers. While there is no reason to suppose that the order thus determined is any more, or less, significant than any other order, it forms a convenient standard which I shall assume henceforth.

The effect of the rule in action can be seen in this diagram, which shows a few steps in the evolution of the state of the automaton as a whole :



Each row is a state of the automaton as a whole, commonly called a *configuration*, computed (except for the first row) from the previous row by applying the rule to each cell; the time axis points downwards. The sequence is not particularly interesting, but bear in mind what the diagram means, for another diagram of the same type will shortly make its appearance.

What can we do with cellular automata ? Approximately two things : we can study their behaviour, theoretically or experimentally, in the hope of gaining some understanding of how they work, or we can think of them as computing machines and try to find ways to programme them to carry out some useful computation. It is probably fair to say that in neither of these directions has investigation so far met with conspicuous success. Even if we know the rule obeyed by the cells of an automaton, we do not in general know how to calculate anything much about its behaviour by any means except laboriously working through the stepwise changes of configuration; and the converse, programming, problem – given a desired behaviour, work out a rule which produces it – is even less tractable.

That does not sound encouraging as a base for constructing a system which learns from experience. Surely we must begin with some system which we can set up in some way to change into some improved form as it finds out new things – but we do not know how to write a programme to make an automaton perform a specified task, so we can't set it up at all; and, even if we did, we have already observed that the rule is fixed, and cannot be changed.

At the same time, if we can forget these inconvenient facts for a moment, a cellular automaton has its attractions as a means of implementing a learning system. It is a very simple system with all its behaviour controlled by a single rule, and it is capable of exceedingly complex behaviour. That we cannot programme it is not necessarily a bad thing – particularly if we seek non-symbolic learning, when we can regard this property as an insurance device which preserves us from inadvertently doing just a little programming to help it along.

But how are we to change the rule ? Well, all is a matter of viewpoint. If the rule of an automaton cannot be changed, then we will have to construct new automata; and even if we do not know how to construct a rule to solve a problem, perhaps we can find a way to improve a rule. The improvement method cannot be derived from such cellular automaton theory as exists (or we would have something equivalent to a programming technique) – so we must find one from some other source.

The method I shall discuss relies on a genetic algorithm. This procedure doesn't exactly improve the existing rule in any direct and obvious way; instead, it constructs a lot of new rules which might or might not be better than the old ones, tests them all, and grabs any better ones that happen to turn up.

My choice of this method isn't arbitrary; it was used in a series of publications found by David Norman while he was working on his M.Sc. thesis⁵. The publications emanated from the Santa Fe Institute, and were by various sets of authors, but the sets usually included J.P. Crutchfield and M. Mitchell – so we came to call them the C&M examples. I shall come back to a specific example of their work shortly, but a digression into the nature of genetic algorithms is appropriate first.

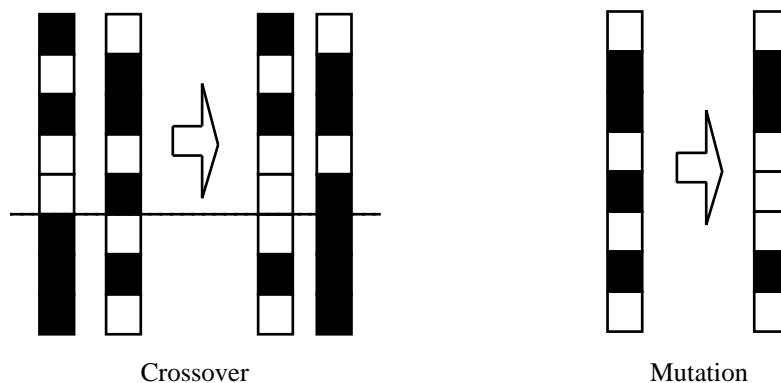
GENETIC ALGORITHMS.

There are several varieties of genetic algorithm⁶, all using techniques based on some feature of genetic processes as found in (usually sexual) reproduction. The aim is to use selective breeding to produce a result which satisfies your requirements. The trick can be worked with various sorts of problem, but as I'll only be interested in programmes I'll suppose that's what we're discussing from now on; it simplifies the description.

To evolve a programme by this means, you have to begin with suitable breeding stock. As the breeding is based on performance, your stock must be able to perform; if you want to breed executable programmes, therefore, you must start with a group of programmes which might be able to accomplish the task you want. In principle, you could begin with any set that satisfies that requirement, but in practice however much you evolve a 50-character Fortran programme, you're unlikely to develop anything impressively intelligent. (That isn't quite true, for if you build into your system some provision for extending the abilities of its stock beyond the range defined by the original pattern there's no obvious limit⁷, but that's an unusual sort of algorithm which I shall not pursue here.) Then you test the abilities of your stock, kill off those which are farthest from what you want, and breed the rest. And you repeat that process until a programme which is good enough for your requirements is produced, or you decide that it isn't going to work.

The interesting part of this process is the breeding machinery. The principle is to derive the programme from an analogue of the living cell's chromosome. This must be a linear string of symbols which in some way determines the form (and therefore the behaviour) of the programme, just as biological genetic material determines the form of the living creature which grows. In general, the chromosome is not the programme, so if you really wanted to use Fortran programmes you would devise a linear code in such a way that every possible string would define some Fortran programme, and use the code as your chromosome. In our case, matters are simpler, for we have a ready-made chromosome in the form of the cellular automaton's rule; it is a linear string of binary symbols, and certainly determines the behaviour of the automaton.

To breed the automata, we take their genetic material and subject it to analogues of the biological processes which lead to variation in properties. Two processes are commonly used, *crossover* and *mutation*. Here's a picture of how they work; a brief description follows.



In *crossover*, which forms the principal evolutionary mechanism in most genetic algorithm applications, two genetic strings exchange their material at some point along their lengths, which is

arbitrarily chosen but the same for both. In the diagram, the left-hand string of the result pair is composed of the upper part of the original left-hand string and the lower part of the original right-hand string, while the right-hand string of the result pair is composed of the complementary parts. This is seen as a good thing to do, because it allows for some variation in properties while preserving much of the original structure intact. That's important, because desirable behaviour might well be determined by the combined effect of values at several different points along the string, which must not be disturbed if the behaviour is to be preserved.

In *mutation*, a single component of a single genetic string is changed into something else. In the diagram, the fifth component from the top of the string is changed from black to white. This is seen as a good thing to do, because it ensures that – eventually – all combinations of genetic possibilities are accessible. To illustrate the point, consider the extreme case in which it happens that every genetic string in the original population begins with the same symbol – then crossover alone could never produce any string with a different symbol in the first position. It is also true that too much mutation can destroy the structures painstakingly built up by selection over generations of crossover, so it can be overdone, but a little mutation is a sort of insurance against getting locked into various sorts of unsatisfactory state.

These operations can be combined in various ways, but a typical example – that used by C&M – is easily described. The algorithm works like this :

Select N initial rules.

Repeat until satisfied :

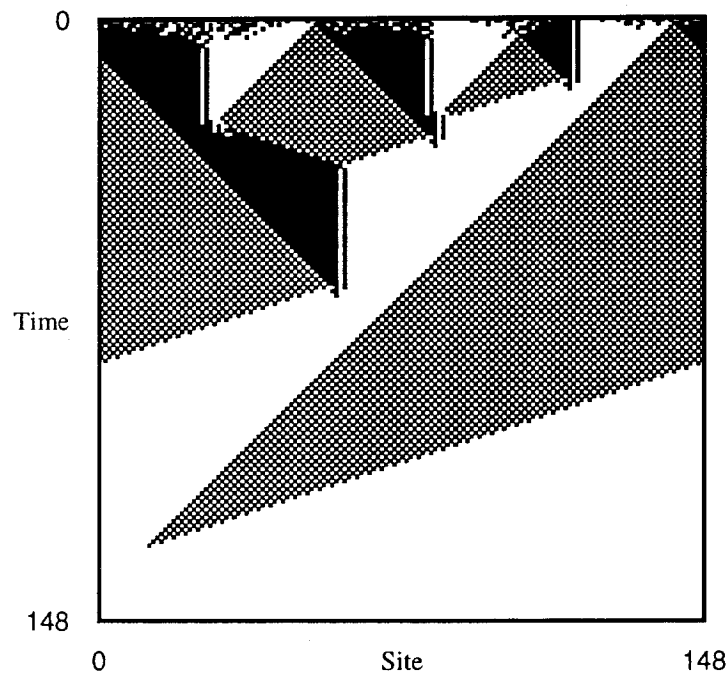
- Evaluate the rules : run the automata on a random sample of possible initial configurations and grade the results.
- Select the best M (for example, $N/2$).
- "Breed" them to get new rules, using some combination of crossover (a lot) and mutation (a little).
- Replace the worst old rules by the newly bred rules.

The repeat loop typically continues until the performance doesn't seem to be improving any more, which might take many thousands of iterations. The interesting fact is that the performance *does* improve, so we can at least imagine that it's learning. But *what's* learning ? It isn't any individual cellular automaton, nor the genetic algorithm, for neither of these ever changes. The only change is in the genetic material, but it's hard to identify that with learning as we know it, for the system shows no sign of understanding – but then, neither did the noughts-and-crosses algorithms.

C&M'S EXPERIMENTS.

C&M studied the evolution of a rule which would cause a cellular automaton to perform some well-defined computation. There is a limit to the sort of computation which you can expect from anything as stupid as a cellular automaton, but one plausible sort of problem is a classification. You identify some binary property which can be determined for any configuration of black and white states of the automaton's cells; then you define the function which converts all initial configurations into one or other of two standard configurations according to the value of the binary property.

C&M used the "majority problem" : given a cellular automaton with an odd number of cells, and with a size much greater than the neighbourhood size of the cells, determine whether there's a majority of black or white cells, and turn all cells of the automaton to the majority colour. The interesting thing about this problem is that it can't be solved without some sort of communication between the cells, for no individual cell can see the whole automaton. They used an automaton with 149 cells, each with a neighbourhood of seven cells, and started with a random set of rules. Here's an example of the performance of one of the rules which they derived after many generations of genetic refinement⁸ :



It works. More precisely, it usually works – but, considering that the genetic algorithm started from nothing, that's fairly impressive.

HOW DOES IT WORK ?

In a fascinating series of experiments, C&M show that their cellular automata evolve from random actions to effective performance through a clearly defined sequence of stages. (The *sequence* is reproducible; now, that's *really* fascinating !) First, the system learns (that's begging the question, but it's the natural word to use in the circumstances) to produce rules which change the automaton state always to either all white or all black; that guarantees it 50% success. Next, it learns rules which expand large areas of uniform state; this is not a correct algorithm, as a predominantly black configuration can easily contain larger white sequences than black, but it's more likely to be right than wrong, so the performance improves. Eventually, it learns rules which produce the sort of behaviour illustrated in the diagram, characterised by sweeping boundaries moving systematically across the automaton.

What is happening at the boundaries ? C&M describe the phenomenon in terms of *particles*, likening the moving boundaries to fundamental particles of physics. I have offered an alternative interpretation⁹ in terms of the logical operations carried out. Very briefly, consider the lowest grey-white boundary in the figure : as the configuration changes (moving downwards in the figure), the change which occurs at the boundary can be summarised as grey + white → white. In the grey area, there are equal number of black and white states; if we regard a white area as encoding the result "more white than black", which is precisely what we have required it to mean in the final configuration, then this is sound logic, for if we combine an area predominantly white with an area of equal proportions, then the result must be predominantly white.

To make this work, the rule must satisfy many constraints. In particular, the all-white and grey states must be stable unless disturbed, and a cell must be able to notice that it is at the boundary between the two areas (which means that its neighbourhood must be large enough to be distinguishably different in this case) and adjust its state accordingly. Such a rule relies for its operation on a disconnected pattern of bits along its whole length; for example, to encode the logic of this operation alone using a rule with a neighbourhood of five, this pattern must be encoded :

Index	Input	New state
0	WWWWW	W
8	WBWWW	W
10	WBWBW	B
16	BWWWW	W
20	BWBWW	W
21	BWBWB	W

Item 0 stabilises the white areas; items 10 and 21 stabilise the grey area (with an oscillating pattern, as in the C&M figure; a static grey pattern is obtained by interchanging the new states); while the remaining three deal with the boundary between grey on the left and white on the right.

That's how it works, which was the question I asked. Are we any the wiser ? Not much. Grubbing about in the detail of the machinery is not very enlightening. Can we take a broader view of events ? Perhaps it might be better to ask what the system is doing, rather than how it works.

IS IT INTELLIGENT ?

What, then, is the system doing ? Is it, for example, solving the problem ? Not really, if we seek anything resembling an algorithmic solution. It seems fairly unlikely that the logical reasoning I've identified can be elaborated to cover all cases⁹. A better description of the behaviour might be that the system is exploring its universe – developing "knowledge" which works using the only machinery it has. Can we see that as the first steps of science ? I would prefer not to, because the system does not – and, indeed, cannot – formulate models, or otherwise interpret its observations. But we can make out a much more plausible case for engineering; the system experiments in its universe, finds evidences of useful behaviour, and deploys its knowledge to produce artefacts which perform according to predefined specifications. Whether that constitutes intelligence might depend on your point of view.

CHAPTER 4

Third Example : Neural networks ("connectionism")

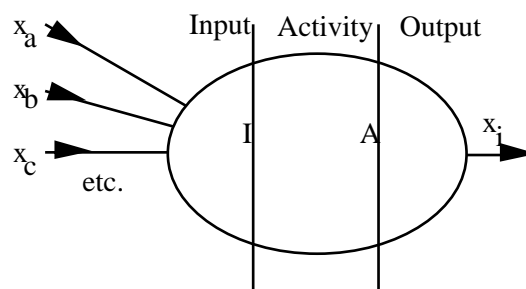
I've left neural networks until last because they're the machines which it's most fashionable to associate with machine intelligence these days. That's partly because they are the machines which resemble most closely the entity which we associate naturally with intelligence – the brain. Both brain and neural network are constructed from units which pass information from one to the other, and can be connected together in arbitrarily complex ways. These are called *connectionist systems*.

A cellular automaton is a sort of connectionist system, but one which doesn't change, and therefore can't learn. Neural networks are so designed that they can change their behaviour in response to external stimuli, so we don't have to introduce any separate learning contrivances such as genetic algorithms. (We can if we want, and some people do, but I shall ignore that possibility.)

NEURAL NETWORKS.

A neural network is, oddly enough, a network built of neurons. The neurons perform (fairly) simple calculations, while the network conveys the outputs of neurons to the inputs of other neurons. The networks which I shall describe all have inputs and outputs, and the job of the network is to convert the input signals into corresponding outputs in some defined manner. More significantly, its job is (again restricting the discussion to the case I shall discuss) to learn to perform the conversion, given examples of the desired behaviour.

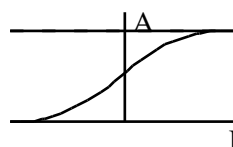
The neurons are loosely based on biological neurons. They come in many varieties, but share the properties of accepting several inputs and producing one output. Here's a picture of a general neuron, with inputs on the left and output on the right.



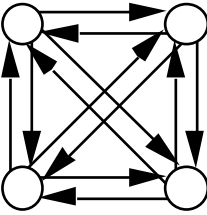
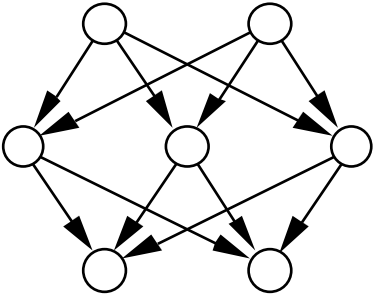
With appropriate definitions for the input, activity, and output functions, that covers most, if not all, sorts of neuron in anything like common use, but there's a specific set of functions which are very widely used, and are indeed used in the system which I shall describe as my example. They are :

- All signals between neurons are in the range 0 to 1;
- Input : $I = w_a x_a + w_b x_b + w_c x_c + \dots$
- Activity : $A = f(I)$
- Output : $x_i = A$

Two features of that description have not yet been defined. They are the *coefficients*, w_a , w_b , w_c , etc., and the *activity function* $f()$. The coefficients are variable weights, which don't appear on the diagram because they are more properly associated with the links between neurons. In biological terms, they are properties of the synapses through which signals flow from cell to cell, and because of that they are sometimes called synaptic coefficients. The weights are very important in neural networks, for they are the basis of the networks' programmability : it is by changing the values of the weights that we can change the behaviour of the networks. The activity function converts the total input to the neuron into a value which can reasonably be thought of as the internal state of the neuron; it is common to use sigmoid functions of the general form shown in the diagram :



Just as certain sorts of neuron are favoured in practice, so, though neurons can be connected in any way at all, certain patterns of connection have been more thoroughly investigated than others. Here are two examples :

<i>Topology</i>		
<i>Description</i>	Fully connected	Feedforward
<i>Name</i>	Hopfield	Multilayer Perceptron
<i>Function</i>	Associative memory	Function evaluation

The multilayer perceptron topology, illustrated on the right of the figure, is the most common in current fashion, and the only pattern which I shall discuss further.

Neither the structure of the neurons nor the topology of the network contributes directly to the way in which we can change the network's behaviour, and therefore cause it to learn. In a neural network, the neurons themselves always carry out the same computation, and have nothing resembling the cellular automaton's "programme"; instead, we can adjust the weights of the connections between the neurons. The usual method depends on knowing the mathematical form of the relationship between the neuron's inputs and its output, and it proceeds by adjusting the weights in the direction which will cause the observed behaviour of the network to approach the required behaviour. The result is an adaptive system which learns from examples. Typically, a largish set of examples of input-output pairs exemplifying the required behaviour is obtained, and this algorithm is executed :

```

Repeat until it works :
  Repeat for each example ( i )
    Present input i;
    Observe the network's output i;
    Work out how to change the weights to bring the
      output closer to the desired output i;
    Change the weights slightly in the indicated
      direction.

```

If all is well (which can often be arranged) we end up with a network which is trained to associate each input pattern in its training set with the corresponding output pattern, and also to interpolate sensibly between the input patterns. This perfectly ordinary consequence of a process which is, after all, just a matter of fitting a rather complicated function to a set of points by trial and error, is renamed "generalisation", and regarded with awe as an example of an *emergent property*, a mystical outcome of connectionism. (I'm not decrying generalisation; it is a very valuable property of the system. I am questioning its elevation to the realm of wonder.)

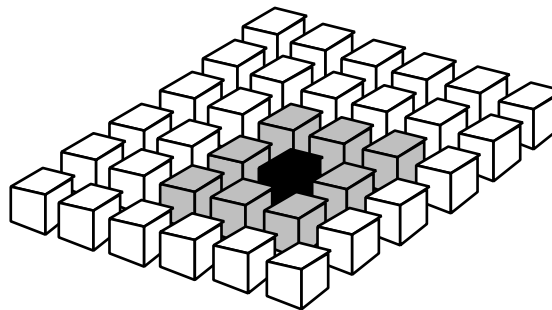
It is not unreasonable to regard this process as an example of learning. By practice, and – in effect – being told when it's wrong, the network acquires an ability which it didn't have before. (Of course, it also loses an ability which it did have before – that of producing whatever response to the input patterns it did produce before we started training it.) Most neural network techniques rely on this learning ability. I shall introduce a rather complicated example.

TIM STUCKE'S NETWORK : THE MIND'S EYE.

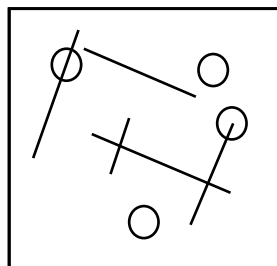
This network was developed by Tim Stucke as part of his doctorate research¹⁰. "The Mind's Eye" was Tim's name for it; I don't think he ever explained why. It's an anagram of "Hey, Tim's need", but I don't think that's significant. It learns to see "naturally variable" objects – that is, objects which have a common structure but which always differ in detail. Faces are a good example; they almost always have eyes, ears, a nose, a mouth of approximately the same shape in roughly the same relative positions, but they are nevertheless almost all easily distinguished from each other. Tim worked with leaves, which have the same property of being the same even though different.

Because there was no mechanism for moving the image on the network's "retina", such as we use when we direct our gaze straight towards an object of interest, Tim's network had to do something which few computer vision systems (or, for that matter, real eyes) can manage effectively : if it was reliably to recognise a leaf wherever it appeared in the input picture, it had to be able to recognise any part of a leaf wherever it turned up. In other words, the network had to be able to identify leaf edges, and leaf veins, and leafy substance, and so on at every point of its retina – indeed, every point had to be functionally identical with every other point so far as recognition was concerned.

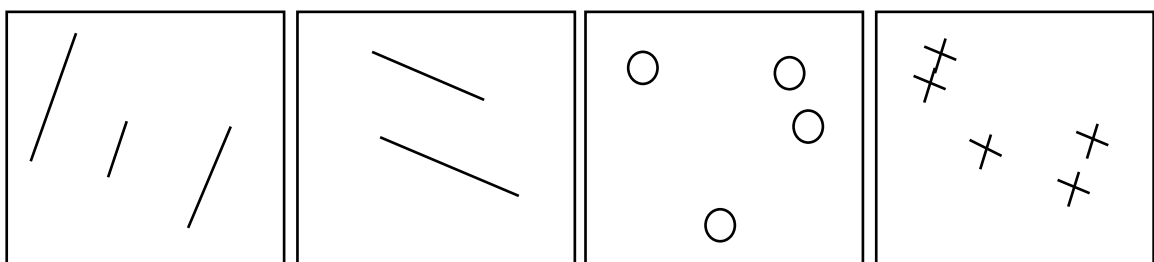
To be more precise, speaking in terms of identifying veins, etc., at each point of the retina is inaccurate, as there isn't enough information at a point to identify the local structure of the image. It's therefore necessary to take into account a certain neighbourhood in each case. At every point, then, we want some mechanism which can inspect a small local area of the image and decide what sort of leaf component it might be. Tim achieved that part of the specification by associating a neural network with every point of the retina. All the neural networks were the same, and they all had a fairly limited field of view; each could see the picture at its own point of the retina, and outwards a few steps in all directions over the surrounding points. That's the *receptive field* of the network. Here's a diagram; each cube is a neural network, and we shall imagine that the retina, with one point for each neural network, is below the network layer. The grey cubes illustrate the receptive field – the range of retina points "visible" to the black cube – so, in network terms, the black network receives one input from each of the nine retina points more or less beneath it.



Recall that each network must classify what it sees into the different sorts of part of a leaf which it has to recognise. It's easier to demonstrate with something more schematic than a leaf, so here's a picture composed of lines in two directions, circles, and crosses :



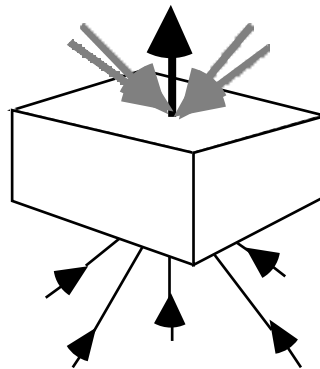
To deal with pictures of that type, each local network in Tim's system would have four outputs, one corresponding to each sort of image feature; we say there are four *feature classes*. A signal on one output would mean that the network could see a circle, on another that it could see a cross, and so on. That means that if you were to collect together the circle outputs of all the networks, and the cross outputs, and so on, you could make four pictures of the input, each showing only one of the features. They would look like this :



And they might indeed look quite like that with my nice clean synthetic pictures using the system as I've described it so far; but natural objects are not so neat and tidy, and in practice it's not at all easy to identify the parts. Local images might be blurred or obscured, contrast might be poor, shapes might be rather

different from anything previously seen. Each network needs more information to make its decision – but it can get more information by asking its neighbours. In effect, the process works in this way. (And, no, I don't really believe that the networks are thinking; consider it to be a metaphor introduced for ease of exposition.) A network thinks that it might be able to see an east-west line, but it isn't sure. It asks its neighbour to the east what *it* can see. The neighbour reports that it thinks that it might be able to see an east-west line, but it isn't sure. That's consistent, so they both gain confidence. If the neighbour on the west agrees too, and those to the north and south don't think that they can see a line, then the hypothesis is much strengthened.

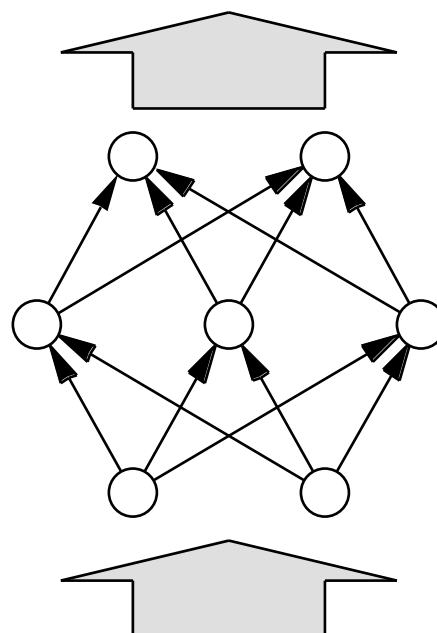
To make this work, each of the little cubes in the picture above must have quite a number of connections with various objects in its environment. Here is a representation of the connections :



The thin lines coming in from underneath the cube represent the inputs from the retina. I've only drawn five lines, but there may be more – the number depends on the size and shape of the receptive field of the cell, and that can be as large or as small as you choose. Much of Tim's work was done with a square receptive field of nine elements. The thick lines going into and out of the cube at the top represent connections with what amounts to a second receptive field of feature classes; each line represents several connections, one to a single point of each feature class. As the feature classes are defined by the networks' outputs, this arrangement gives each network information on what its neighbours have decided. If the system were set up to deal with my simple example, there would be four connections in each thick line; the black line leading from the cube conveys the output from the cube to its own point in each of the feature class arrays, while the grey lines leading into the cube are its inputs from the four neighbouring points of each of the four feature class arrays. In Tim's work, there were in fact eight such inputs, not four – that's one from each other cell in the nine-cell square, which happens to be just the same connection scheme as that used for the receptive field for the retina, but that's an accident.

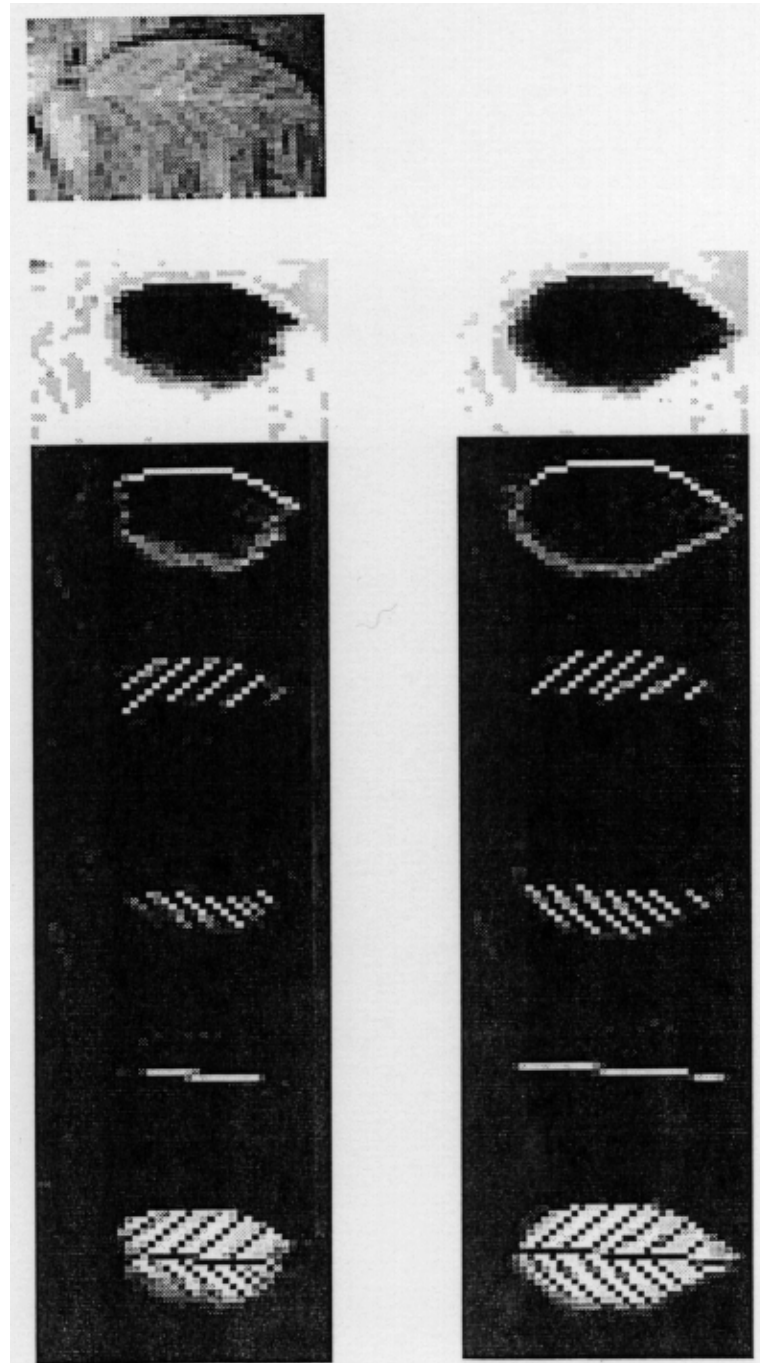
Inside each of the cubes is a neural network, and the diagram below shows how the connections shown above on the cube were made to the network itself. The network is shown only schematically; Tim's networks were very significantly bigger than that illustrated.

1 output to each feature class array :



8 inputs from each feature array,
9 inputs from the image :

Extending this structure to the whole array of networks, this leads to a sort of decision-making by consensus – and it works. Here's an illustration¹¹. The top frame shows the input to the system's retina. Tim used six feature classes, and their outputs are shown below. Reading from the top, they represent the background, the leaf boundary, the upper veins, the lower veins, the large central spine, and the leafy substance between the veins. The first column of pictures shows the output after two cycles of consensus-making; the second is after eight cycles, and the improvement is very clear.



HOW DOES IT WORK ?

I've described how the network recognises images, but I still haven't said where the learning comes in – therefore, considering that this discussion is supposed to be about learning systems, I had better do so.

In principle, the learning proceeds as I described for the multilayer perceptron. A few pictures of leaves are analysed (by people) into their component feature classes, so we can start with the pictures and the outputs we want in each case. At a lower level, this means that for each elementary network we know the input from the retina, and we know the output we want to see in the feature classes. Because we train every network identically, each single picture gives as many training examples as there are networks in the system – typically, 900 examples. We therefore only need a few pictures to give a very thorough set of training examples.

The position is slightly complicated by the feedback from the feature arrays to the networks' input layers. While we know the desired final states of the feature layers, we don't know what their states might be in the intermediate stages of the computations, and we should really take into account the intermediate values during the learning phase. Fortunately, we don't need to go into the details here, but I wanted to point out that the consensus requirement wasn't trivial. Even more fortunately, Tim's algorithm does work.

IS IT INTELLIGENT ?

Tim's network is by far the most complicated of my examples, and it's worth pausing to make a couple of points before going on to review the exercise.

First, recall the picture of the network as a two-dimensional array of cubes which are all identical inside and communicate with their neighbours, and over a period of time convert an input into a specific output. Does that sound familiar ? Does it sound, perhaps, rather like a two-dimensional cellular automaton ?

Yes, it does. It isn't quite a cellular automaton, if only because the original input is always there, but it's close. There is a sense in which Tim did build a programmable cellular automaton, but before we get too excited we should observe that it's computing a very simple function; though it uses communication to refine its decisions, each cell's output is determined largely by its own immediate input from the retina, and there's nothing like the majority problem in his experiments. Even so, it's an interesting, and essentially accidental, link between the two approaches.

The second point of interest is that this network is beginning to behave in a way which approaches human behaviour. It approximates, it guesses, it muddles through. It has a long way to go before it can muddle like, say, a politician, but it's a beginning, and it's different from the much more precise sorts of computation which we find in more conventional computing.

It learns to carry out a very subtle task, which is much more complicated than those tackled in my other examples. It achieves a sort of perception – indeed, it can do surprisingly well even when given a rather poor image. Whether that constitutes intelligence is, again, a matter of definition, but the learning is there, and perception is one of the characteristics of entities which we consider to be intelligent.

The case of perception, and vision in particular, is interesting. In the earlier, and more sanguine, days of artificial intelligence, it was widely supposed that the really hard questions in intelligence were those for which we award people doctorates – reasoning, logic, mathematics, and so on. In practice it has turned out that those are the easy ones, because they can all be formulated very precisely and economically. They're not trivial, but they're fairly tractable. The hard ones are the things we take for granted, like vision and hearing and finding our way about. We take them for granted because we have an enormous amount of machinery in our heads specifically devoted to just these problems. Reproducing the functions of this machinery has turned out to be very hard, and Tim's network performs rather well in comparison with other systems.

CHAPTER 5

Conclusion ?

In this chapter, I should draw together the insights we have gained from our three examples, scrutinise, compare, contrast, and otherwise express from them the essence of their implications for our discussion of artificial intelligence. I shall try to do a little of that, and I'll come back to a "definition" of intelligence at the end of my remarks. All in all, though, this is a warning against expecting too much : it isn't a simple question, as many others have found before I started to worry about it. I think that there is something to be learnt from the examples, but it might have more to do with our limitations than with machines' intelligence or lack thereof.

Indeed, I shall start by forgetting about intelligence completely. Consider instead the more general – and much less contentious ! – question of building a machine to solve any very complex problem. How do we do it ? We invariably do it by connecting together a lot of (comparatively) simple parts. For very complex problems, we might apply the algorithm recursively, achieving the machine by assembling a series of submachines.

COMPLEX MACHINES.

Consider a specific example. A steam locomotive (chosen only because I know a little more about them than I do about some other fairly complicated machines) of the generations which flourished just before the species became almost extinct is composed of many parts, all carefully integrated to convert a suitable fuel into a very large tractive force – but it seems quite likely that none of the parts would constitute a long-term puzzle for, say, an engineer who designed and supervised the building of one of the Egyptian pyramids.

The example is, of course, slightly cooked. I've chosen a specifically mechanical example, and offered the puzzle to someone who would have a good qualitative grasp of mechanics, though lacking knowledge of the detailed reasoning which leads to the design of the locomotive as a whole, but I think that's fair. The Egyptian engineer would have a much harder task with electronics, but that's more a matter of complete unfamiliarity with the nature of the field concerned than a consequence of the intrinsic complexity of the devices.

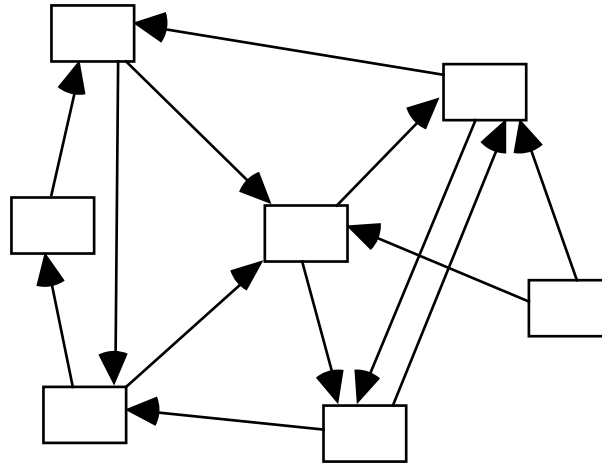
It might be that our preference for building complex objects from simple components is only a reflection of our stupidity. Many enlightened souls urge us to seek holistic solutions to complex problems, regarding any attempt at analysis as a guarantee that the true nature of the problem will be lost. Perhaps if we were able to comprehend each problem completely we could indeed find a direct solution and bypass the laborious analysis and synthesis. In practice, I find that this route is not open to me, and I have to plod along with my analysis and synthesis. It seems that many other people are similarly limited.

We are not alone. In the natural world, things are built from other things, down to the level of atoms and below. My knowledge of biology is ludicrously incomplete, but I have read of marvellous unicellular organisms which seem to approach the holistic ideal : the single cell manages to discharge a large number of functions simultaneously, with rather little obvious internal structure to support the separate activities. It is true that we know of various organelles and other entities within the cells, and even have some idea of how they cooperate to carry out the cell's functions, but there remains a soupiness to the protoplasm which gives little impression of structure, even if it exists in the subtleties. Nevertheless, there exist multicellular organisms, which are magnificent examples of complex objects synthesised from simple components, with specialisation of function evident throughout. The cells themselves, just like the parts of the locomotive, are specialised to perform various different functions; and at a higher level bodies have limbs, sense organs, digestive systems, brains, and other parts specialised to perform their own functions.

Whether there is any significance in this natural precedent is another matter. Despite impressions which one might gain from some sources, the argument from nature has no moral force. There is no implication that synthesis from specialised parts is the *right* way to go; all we can say is that it's a *possible* way, and that it does seem to work in some circumstances. It is relevant to my argument in that the intelligent systems we know best are constructed in exactly that way, and it seems to work in those circumstances – but I said that I wasn't going to mention intelligence yet, so let us proceed to think about complex systems, accepting only that the synthetic route does seem to be effective, and that we don't know a better one.

BUILDING THINGS FROM PARTS.

What sorts of part are available ? At the lowest level, I think that just two sorts will suffice : boxes, and connections between boxes (which I shall call lines). Given a sufficiently liberal interpretation of the connections, that's little more than a restatement of the notion of parts – except that it emphasises that some of the parts are active, while others are more concerned with communication.



That's a picture of an arbitrary network. Generally, each box does something which affects values associated with its connections. That's a very noncommittal sentence, because there are very many ways in which the values can be affected¹². I've shown all the connections as directed, implying that boxes have inputs and outputs, but even that is restrictive; a lever is a sort of box with three connections, representing the positions of its two ends and its fulcrum. The function of the box is to establish a fixed relationship between these three variables – and, as you can get hold of any of the parts of lever and move it, there's no distinction between input and output.

In all the examples I have discussed in previous chapters, it is easy to distinguish inputs and outputs. I don't know whether that's essential for artificial intelligence, but it seems to work on the limited range of examples I've described. It isn't essential in all sorts of intelligent activity, for explicit inputs and outputs aren't defined in mathematical and logical entities which amount to statements of fact. As in the case of the lever, many logical and arithmetic assertions have no explicit direction in that any term can be derived if the others are known – so if $A + B + C = 0$, a specification of any two of the variables defines the third. This behaviour can be incorporated into computer languages (Prolog is a well known example), but in practice it's very hard to implement consistently and completely. On the other hand, when we build structures to do things, we usually assume that inputs and outputs are essential. It's interesting that in general systems theory components are assumed to have defined outputs¹³. Here I'll assume that we can talk about inputs and outputs, but I conjecture that the argument, such as it is, doesn't depend on that assumption.

MAKING MACHINES DO WHAT WE WANT.

There are just two ways to acquire a machine to perform a specified task : you can build a special-purpose machine which does the job, or you can start with a general-purpose machine and change it, which is what we mean by programming. Either way, if the boxes-and-lines view is correct, there are three things we can do to get the desired behaviour :

- Decide which boxes and connections to have.
- Determine the functions of the boxes.
- Determine the functions of the connections.

It is a marvellous, though not necessarily significant, fact that these operations correspond to the three examples which I introduced, more or less, even though I chose the examples before working out this analysis. This table summarises the correspondence :

ARCHITECTURE	How many boxes, how they're connected	Computer programmes – build the structure.
BEHAVIOUR OF BOXES	How the boxes are programmed	Cellular automata – work out the rules.
BEHAVIOUR OF LINES	How information is transmitted	Neural networks – adjust the weights.

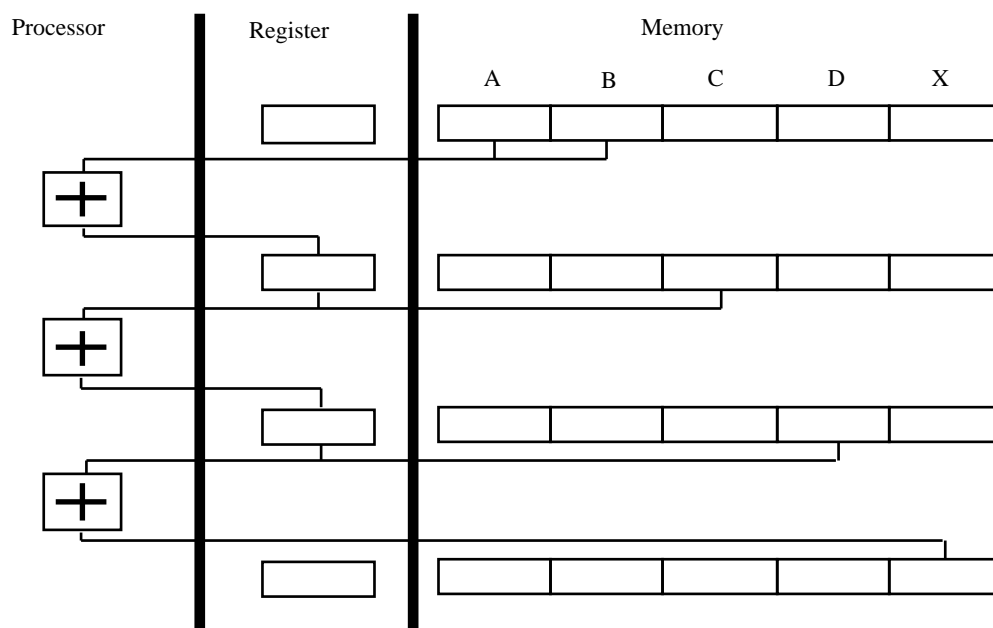
Computer programmes.

Of the three implementation methods, the use of computer programmes to change the architecture of the system itself is perhaps the least obvious. Nevertheless, that is essentially what a computer programme does.

It's instructive to think a little more about that. For any computing task, it's possible in principle to design a special-purpose machine to do the job. This is a complicated machine, so we build it in a thoroughly satisfactory lines-and-boxes way, using electrical connectors for the lines and standard components (registers, logic gates, adders, comparators, and so on) for the boxes. In practice, though, to do that is prohibitively expensive, both in money and time, so we achieve the same end by a different route, using general-purpose programmable computers.

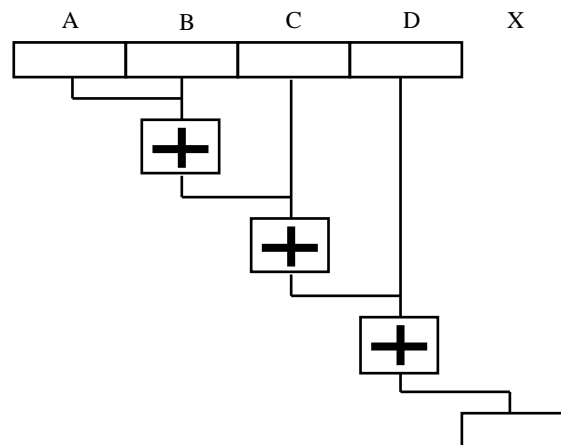
The common digital computers work by switching connections as directed by their programmes, in effect, if not in fact, making and destroying connections as they go. To execute one instruction, certain gates are opened and others are closed, thereby becoming a (usually rather small) piece of electronic machinery for the duration of the instruction. Then the results of that stage are saved, and the next instruction reconfigures the hardware so that it becomes a different piece of electronic machinery. And so on. Eventually, after many such cycles of operation, the computation is completed.

What's really doing the job ? It's the "machine" you get by taking all these momentary machines and connecting them together so that there is a wired connection for each transfer of information from momentary machine to momentary machine. (And a few more bits to deal with timing problems and suchlike, but the principle is unchanged.) That's a special-purpose computer to do the required job; the programmed computer is a very clever way to provide any number of different special-purpose machines, each doing a different job – but for our purposes it's really the special machines we should be thinking about. Here's an illustration. This is a (simplified, but honest) representation of how a conventional digital computer could go about performing the calculation $X = A + B + C + D$:



In the first step, the values of A and B are fetched from the memory, added in the processor, and stored in a register. In the next step, the contents of the register are added to the value of C, fetched from the memory, and the result is again stored in the register. Finally, the contents of the register are added to the fetched value of D, and the result stored in the memory location allocated to X.

But now take that diagram, cut out the intermediate register stages which contribute nothing to the computation, and suppose we can use as many adders as we want. This is what we get :



This circuit would do just the same job as the computer, and it would be much more efficient – and you could make it even more efficient very easily. What you couldn't do is build it in a couple of minutes, but you can write the computer programme to simulate it faster than that. In effect, the computer programme has generated the hard-wired network, and to change a programme is to change its generated network.

That's what the noughts-and-crosses programme does, though in a slightly more complicated way. It's interesting that it works by performing the same trick twice :

- First, because it's hard to build a special-purpose machine to do the job, we write an ordinary computer programme which will play the game according to the rules we saw earlier;
- But to make a machine that changes its behaviour means that the programme has to keep changing itself, which is also hard – so we write the programme itself to be programmable, by causing its behaviour to depend on its data in a well organised way.

Now by changing the data (that's the memory of bad positions) we change the behaviour of the programme, and that in turn in effect constructs a different effective hardware machine, so the final effect is what we wanted – a machine of variable architecture, which for convenience is simulated by a digital computer.

Cellular automata.

If we want to build a learning system, but don't want to change the system architecture, we have to change some parameters of the components of a given architecture. A cellular automaton is a set of simple fixed boxes with a fixed set of connections, but in this case the connections don't change, while the box functions do. We think of a large number of separate little and very simple computers connected together, all going at once, and all executing the same programme.

There is no fundamental reason for restricting all the programmes to be the same, except for the suggestion of mathematical elegance which it imparts to the model. While I haven't tried to do it, it seems rather likely that if one were allowed to pick the programmes for the cells individually one might be able to exploit the cyclic structure to do interesting things, though it could be that the limited connectivity would pose interesting problems.

But the homogeneity suits my argument very well, for it gives a system in which a form of behaviour modification can be achieved essentially untouched by human hands, which is just what I want. A heterogeneous system would have to be programmed somehow, either by hand or automatically. As I shall explain, once we start to programme machines we're back with symbolic methods, which isn't what I want – but, while it is possible to contemplate automatically programming all the cells in a cellular automaton independently, the sheer size of the problem becomes astronomical. To search for a good rule which is some permutation of a 128-element binary string (which is what C&M did in their experiments) entails a search space of 2^{128} elements, which is tedious, but a comparable approach to an automaton of n cells in which each cell is independently programmed is in effect to search a space of 2^{128n} elements, which is ridiculous.

A second, though incidental, advantage of the homogeneous rule for the cells is that it simplifies the job of constructing the automaton, which for convenience is usually simulated by a digital computer.

Neural networks.

Neural networks are also simple fixed boxes with a fixed set of connections. As with the cellular automata, we think of a large number of separate little and very simple computers connected together, all going at once, and all executing the same programme, but in this case the programme doesn't change. The variation in behaviour is achieved by changing the strengths of the connections between the boxes.

Apart from that basic difference, neural networks differ from cellular automata in two significant ways : there is more variety in the pattern of connections between the boxes, and, as there is no coordination of the values of the weights, there is no analogue of the cellular automaton's uniformity of behaviour over the whole network system.

In practice, the methods for "training" neural networks adjust the values of all the weights independently, and it is usual to begin with random values for the weights to ensure that the behaviour of different parts of the system is not uniform. This greater diversity is possible because the behaviour of the neural network can usually be improved by methods such as steepest descent techniques without involving a large search space. These are built into the model of the neural network, which for convenience is usually simulated by a digital computer.

SO MUCH FOR COMPLEXITY – NOW WHAT ABOUT INTELLIGENCE ?

You might recall that I've been discussing complex systems, and, in particular, the notion that any complex system which we build is likely to be made of lots of bits tied together, if only because (if my suggestion is accepted) it's the only way we have of building complex things. I've shown, after a fashion, that each of the three examples of a learning system is constructed in this way, and, further, that together they illustrate the only ways we have. At least, that's consistent; if learning is a form of intelligence, and if intelligence is a complex problem, then any machines which we build to do it will inevitably be like that.

Such considerations at least raise questions about arguments which suggest that a structural similarity between machine and brain in some way predisposes the machine to intelligence. It is reasonable to suggest that they are both complex because they must perform complex tasks, but the complexity need not be specially related to intelligence. Perhaps a fair conclusion is that the people who built the machines and the evolutionary process which produced the brain are about equally stupid. (Perhaps the people are stupid faster ?)

The ubiquity of symbolic methods.

Alert readers will have observed that the final phrases in my three accounts of different methods above are identical : all the methods are commonly implemented by simulating them using computer programmes. Is that a demonstration that they're all symbolic anyway ? No – it's only a demonstration that we can use symbolic methods to simulate each system. The systems themselves could be arbitrarily complex. You can simulate the growth of a tree, or the flow of air in a wind tunnel, or indeed the inner machinery of a cell¹⁴ to a high degree of precision, but there's certainly nothing symbolic about the simulated phenomena.

The symbolicness or otherwise of the methods is clear from a consideration of their various ways of remembering things. The programmed system is completely predefined, right down to the representation of interesting facts in its memory; given the programme and a slice of the memory, we could work out the meaning of the memory contents. While details could vary depending on the way in which the system was exposed to new knowledge, it would be possible to say right from the start how some specified notion would be represented.

That isn't so of either the neural network or the cellular automaton. In both those cases, even with the whole computer programme available, there is no possibility of interpreting a listing of the memory (the set of weights, or the cell rule) without, in effect, running the programme. The programmes don't tell the system how to represent memories; they only specify what the system must do in order to store memories.

But how far will that go ? Tim Stucke's little networks learn to classify a few sorts of patterns, and C&M's automata learn to classify something even simpler, and neither can do anything else. The noughts-and-crosses programme is also a sort of simple pattern classifier – but it's easy to extend it to do lots of other things too if we want to. It can draw its board on the screen, keep statistics of its games, identify and reject silly inputs, and so on, *still using the same sort of machinery*. To think of trying to train the cellular automaton to do anything similar is a bit of a joke; and even the neural network, which is in some ways much more clever, wouldn't be able to do the job. You could extend the programmes to incorporate those

features in both cases, so far as it makes sense, but you wouldn't change the simulated automaton or network to do it.

To cause a neural network to solve more complex problems, the usual method is to use a *modular network* of some sort, with components specialised for specific tasks and joined together as appropriate^{15,16,17,18}. (No one sane would even dream of trying to make a cellular automaton do anything more complicated.) Tim Stucke's leaf recogniser is of just this design : it is built of a large number of small networks which are trained by real nonsymbolic means, but they are tied together by connections designed by Tim in a very symbolic way; we know what each of the connections is supposed to do, and why they are there, and how the intercell cooperation is intended to work.

Why didn't Tim begin with an enormous network and just train it on many leaves ? Because he just didn't believe that it would work. Even his simpler networks contained thousands of neurons and tens, or hundreds, of thousands of connections; perhaps it could have been trained in time, but time would certainly have been needed.

Generally, to build a neural network with a complicated function, we begin by working out what sorts of module we'll need, and how they should be connected. That's a symbolic step, because that's the only way we know how to do it. Only after that step is complete need we worry about the details of the networks. What we are doing is straightforward symbolic design, which looks like something else because we're incorporating neural networks as a new sort of box with some attractive properties – but the overall function of the network is determined by the symbolic design component.

It seems likely, therefore, that we're stuck with symbolic systems, even though we build them from neural networks. Perhaps some day we'll find other ways to train vast amorphous systems to carry out complex jobs, but for the present we have to do the design, so we can't escape symbolic methods.

What about brains ?

I didn't include brains among my examples, partly because I've been discussing machines, but really because I know very little about them other than hearsay and rumour. Even from that rather limited standpoint, though, it's interesting to see how the brain fits into the categories I've identified.

All the methods for building boxes-and-lines machines seem to apply at one stage or other of the brain's development. In young developing animals, connections are made as the brain grows to deal with incoming perceptions. Later, synaptic weights seem to be changed, and perhaps some of the connections die. Programming by changing the cells' functions is perhaps not so well represented, but it seems that not all cells behave identically, and there are cells with different sorts of behaviour in different contexts.

What does that mean ? Nothing very much, perhaps, except perhaps as a sort of existence proof : the phenomena which I've been discussing do happen somewhere around the most intelligent device we know. That doesn't demonstrate that they're necessary for intelligence, but it does weakly suggest that they might be useful. (The brain also demonstrates that there is a nonsymbolic way to build an intelligent system. It is not encouraging that it takes hundreds of millions of years.)

Comparing machines and brains.

So far, I've been trying to make machines do brain-like things in the hope that they will thereby somehow be seen as intelligent. With all three examples, I've achieved some sort of success; the machines have learnt to play noughts-and-crosses perfectly, developed communications techniques, recognised leaves. If intelligence at all, it's admittedly intelligence of a very low order, but we're only just beginning. If, as I've suggested, we are constrained to rely on symbolic techniques to tackle ambitious projects, we might see that as an obstacle – but if we are using symbolic techniques we can also look to my earlier argument that they offer a way of improving the intelligence of our systems without obvious limit. This train of thought proves nothing, but suggests that there is nothing to prevent us from building intelligent machines.

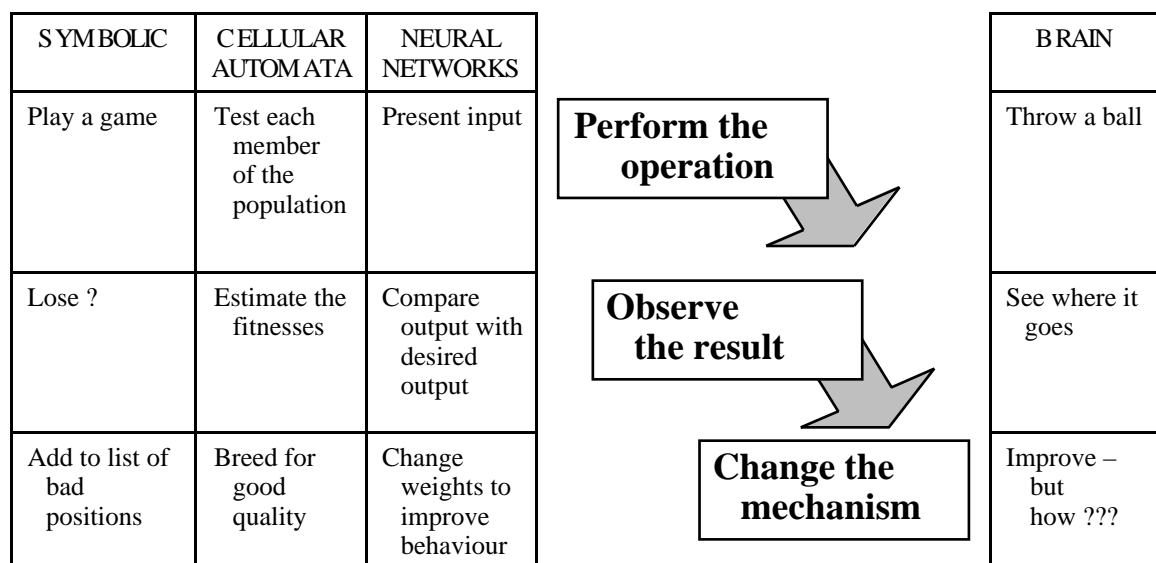
Can we discern any inevitable differences between machines and brains which might pose problems ? The difficulty with that question is that we still know rather little about the brain. (Or so they tell me; that's more hearsay, but it seems plausible.) On the other hand, we can use the symbolic argument again : as we discover describable mechanisms in the brain, we'll be able to write programmes to simulate them.

A significant difference at our present stage of physiological knowledge and computing expertise is that the brains look after themselves, and the computers don't. Brains can slip comfortably from learning to using to learning a bit more to using a bit more to tweaking the learnt bit to improve it, and so on; our computer programmes are in contrast incredibly clumsy, and most of them have to be prodded to stop learning when they've learnt enough, or they can distinguish A from B but can't recognise that C is

neither of the two, or they exhibit other sorts of very primitive behaviour. Even so, it is far from obvious that brain-like behaviour is impossible to reproduce in a machine, and there seems to be no reason to suppose that we won't get there eventually.

And the differences might not be as great as they appear on the surface. Consider how learning is managed in the three examples, and in the brain. When a system learns its behaviour must in some way change, so the connectivity, or the function of the boxes or the connections, must change to reflect this. How is this change effected ?

In an artificial system, it is usually effected by an additional agent which we might call a learning engine. The symbolic programme plays a game of noughts and crosses, the neural network tries to recognise a leaf, the cellular automaton produces its encoded solution to the problem it is required to solve. But then some other entity – the learning engine – takes over, evaluates the result, and changes the machine. The learning engine is able to look at the machine and the result from outside, make some decision on which parts of the machine have to be changed in order to improve the performance, and poke its fingers into the machine to make the required changes. Here's an illustration, comparing the four "intelligent" systems; observe that in all cases the pattern matches that of the universal algorithm which I put forward at the end of Chapter 2.



We don't know everything about the machinery which the brain uses, but there's a consistent pattern which is observable through all four learning systems. The brain looks after the switch from step to step by itself; so does the symbolic system, so that level of autonomy isn't confined to the brain. We see the other methods as requiring intervention, but that's mainly because we have to leave the neural network or cellular automaton and use some other machinery to do the learning – but if I'm correct in my expectation that we'll need symbolic intelligence as well as nonsymbolic methods, that isn't an objection.

INTELLIGENCE AT LAST ?

So, back to intelligence. Are computers (that's any of the machines) intelligent ? – I still haven't answered the question. When you take them apart, where's the intelligence ? We can't yet take the brain apart while still knowing exactly what we're doing, but it does seem quite likely that we won't find any atoms of intelligence there either.

We don't believe that the boxes themselves are intelligent, so presumably it means that the intelligence is associated with the connections between the boxes – but how ? You can evade the question by saying that the intelligence is "emergent", but is that an answer ? – or is it a synonym for "we don't know" ? People have analysed the neural background to cockroaches walking¹⁹; that's similarly to do with connections between neurons, but there's no big todo about walking being emergent behaviour.

Have we found a definition ? No, I don't think we have. If we could, then – if my symbolic argument works – we could make a computer do it. Perhaps if we start from there and work backwards, so to speak, we'll be able to see any differences more clearly. Consider, therefore, a task which both people and computers can do, and which people think is intelligent.

From time to time, I have inverted matrices. I learnt to do it while studying mathematics, and have long since forgotten the details. But suppose that I was required for some reason to invert a matrix, and (equally obscurely) had to do it by hand. I would get out a textbook which told me how to do it, and

work through the procedure carefully step by step, and then I would take the result and multiply it by the original matrix – and as the unit matrix appeared, element by element, I would feel rather satisfied with myself, and consider that I'd exhibited a sort of intelligent behaviour.

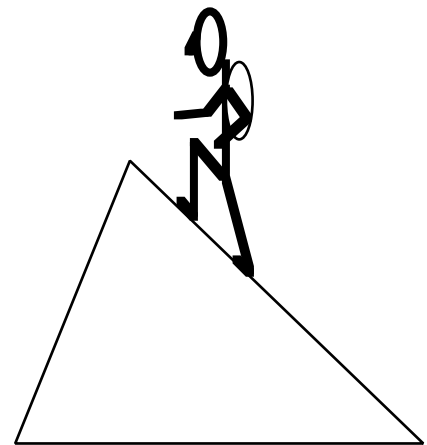
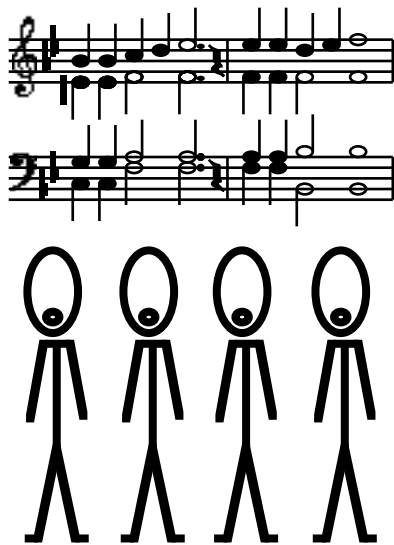
But now if a computer were magically made available, I could take the same textbook and programme the computer to follow through exactly the steps which I took, including the final multiplication to check my answer. I could then run the programme, using the same initial values that I had used, and the computer would (should) reach the same results. But would I consider that the computer had acted intelligently ? No, I wouldn't. Why not ?

On the face of it, my position seems to be untenable. Both the computer and I have completed the same task in essentially the same way; both of us have worked from a programme, so I can't claim to have created anything new. Both of us have behaved, so far as our different anatomies will permit, in exactly the same way – but I still think that I'm intelligent, and the computer isn't.

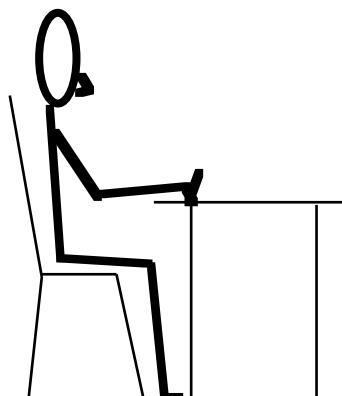
Has anything else changed to account for this apparent inconsistency ? Yes, it has : I don't believe that the computer felt satisfied with itself. I therefore conjecture that there might be a connection between these two phenomena.

So I offer this opinion : Intelligence is something I feel. I ascribe intelligence to other people as a courtesy, and maybe to a limited degree to some other animals, but I have no difficulty in believing that people and animals feel emotions. I do have difficulty in ascribing intelligence to machines, because I don't believe that they have emotions.

It isn't a matter of behaviour. Until about two paragraphs ago, I've been assuming that the indicator of intelligence is intelligent behaviour, but now I'm suggesting that it's an æsthetic experience, not dissimilar to that associated with other exercises of skill and achievement, such as singing, or climbing a mountain.



Intelligence is similar. You sit and think, and you solve a problem. At the end, you lean back and say "Ah !", and you feel good.



And if that's true, then I have no hope of ever writing an intelligent computer programme, because I have no idea at all how to go about making a computer feel good. Consider this alternative version of the imaginary conversation which I put forward during the noughts-and-crosses discussion :

You : That machine isn't intelligent !

Me : How do you know ?

You : It isn't feeling intelligent

?

REFERENCES

- 1 : A.L. Hayward, J.J. Sparkes : *Cassell's English dictionary* (Cassell, nineteenth edition, 1962).
- 2 : O.G. Selfridge, U. Neisser : "Pattern recognition by machine", *Scientific American* **203#3**, 60 (1960).
- 3 : H.R. Lewis, C.H. Papadimitriou : *Elements of the theory of computation* (Prentice-Hall, 1981), page 223.
- 4 : R. Conway, J. Archer, R. Conway : *Programming for poets* (Winthrop Publishers, 1980), page 300.
- 5 : D.H. Norman : *Emergence in artificial intelligence* (M.Sc. Thesis, Auckland University Computer Science Department, 1995).
- 6 : Z. Michalewicz : *Genetic algorithms + data structures = evolution programs* (Springer-Verlag, 1996).
- 7 : I. Harvey : "The SAGA cross : the mechanics of recombination for species with variable-length genotypes", in *Parallel Problem Solving from Nature 2* (R. Männer, B. Manderick (eds), North-Holland, 1992), page 269; also Cognitive Science Research Paper CSRP 223, University of Sussex.
- 8 : R. Das, M. Mitchell, J.P. Crutchfield : *A genetic algorithm discovers particle-based computation in cellular automata*, preprint for the Third Parallel Problem-Solving from Nature Conference, 2 March 1994.
- 9 : G.A. Creak : *Cellular automata with a purpose*, unpublished working note AC93 (December, 1994).
- 10 : T.J. Stucke : *The mind's eye* (PhD Thesis, Auckland University, 1994).
- 11 : T.J. Stucke, G.G. Coghill, G.A. Creak : "The mind's eye: extracting structure from naturally variable objects", *Neural, Parallel and Scientific Computations* **2#1**, 93-103 (March, 1994).
- 12 : G.A. Creak : *Semantics of block diagrams*, unpublished working note AC69 (January, 1990).
- 13 : T.H. Athey : *Systematic systems approach* (Prentice-Hall, 1982), page 13.
- 14 : R.G. Traub, R. Miles, R.K.S. Wong : "Large scale simulations of the hippocampus", *IEEE Engineering in Medicine and Biology* **7#4**, 31-38 (December, 1998).
- 15 : G.A. Creak : *Describing modular networks*, unpublished working note AC87 (September, 1993).
- 16 : M.R. Scaletti : *Modular neural networks for object recognition* (MSc Thesis, Auckland University Computer Science Department, 1994).
- 17 : E.J.W. Boers, H. Kuiper : *Biological metaphors and the design of modular artificial neural networks*, (Masters' thesis, Departments of Computer Science and Experimental and Theoretical Psychology, Leiden University, 1992).
- 18 : E. Ronco, P. Gawthrop : *Modular neural networks : a state of the art* (Technical Report CSC-95026, Centre for System and Control University of Glasgow, 1995).
- 19 : R.D. Beer, R.D. Quinn, H.J. Chiel, R.E. Ritzman : "Biologically inspired approaches to robotics", *Comm.ACM* **40#3**, 31-38 (March, 1997).