Contents

AI lecture 7

Jocelyn Ireson-Paine

February 14, 1996

1 Introduction — The limits to computation

My final lecture will be about some of the criticisms of AI. I need to prepare for that by talking about the limits to computation: what computers can't do. Of course, the average computer can't transmute lead, generate antigravity, or even make toast. What we're interested in is restrictions on the type of computation that can be carried out.

I'll start with two examples of limitations on neural nets, and how they imply the idea of limits to what a learner can learn. Then I'll move to the most general kind of computer known, and what it can or can't do.

2 Some limits on neural networks — linear associator

In this section, I look at an example of a limit on what kinds of thing can be learnt.

Linear associators are one of the simplest kinds of neural net, and you've probably met them already. For those who haven't, here's a brief description. See page 161 of *Understanding Cognitive Science* by McTear (PSY KH:M 025) for a nice intro. Note that although I talk about neurons and synapses, these nets (and indeed most connectionist models) are not at all like real neurons.

Consider the following type of net. There are three inputs, I_1, I_2, and I_3. There are three output neurons, O_1, O_2, and O_3. Each input has synaptic connections to each of the three outputs. Thus, I_1 has one connection to O_1, another to O_2, and a third to O_3. The same for I_2 and I_3. Altogether, this gives nine connections.

Each connection has a given numeric *weight*. In general, I'll use W_{ij} to denote the connection from I_i to O_j. Each output is calculated as follows. Multiply each of the inputs by the corresponding weight, and then add them. In symbols:

$$O_{ij} = \sum_{i=1,j=1}^{3,3} W_{ij} I_i \tag{1}$$

This architecture can be extended to any number, not just three. In fact, the number of inputs does not need to be the same as the number of outputs: the principles I describe below will still work. By setting the weights suitably ("training" the net), such nets can be trained to act as recognisers, such that if you put pattern P in, you get a one on O_1 and zeros elsewhere; if you put pattern Q in, in, you get a one on O_2 and zeros elsewhere; and so on.

Question 1: assume we have a linear associator with two inputs and one output. Can we train it to make the following association:

I_1	I_2	O_1
0	0	1
0	1	0
1	0	0
1	1	1

i.e. to give an output only when the inputs are the same? You may recognise this as the "exclusive-or" problem.

Answer 1: No. All linear associators are what mathematicians call *linear* — hence the name. This means they obey two rules:

- If you multiply each input by the same number k, then the outputs will all be multiplied by k. I.e. if the net gives a pattern X of outputs for a pattern P of inputs, it will give kX for kP.
- Suppose that for pattern P of inputs, the net gives pattern X of outputs; and that for pattern Q, it gives pattern Y. Then for inputs $P + Q$, we get outputs $X + Y$.

The input–output mappings in the exclusive-or table don't follow that rule: hence we know the net can never produce them. Mathematically, the net is a *linear transformation* (matrix) which acts on an input vector to give an output vector. The only distinctions it can make are those which are *linearly separable*.

The original Perceptron suffered from such a limitation, so you'll find stuff about linear separability in most accounts of the Perceptron, e.g. *Introduction to the theory of neural computation* by Herz et al (PSY KH:H 044). You could not for example train the perceptron to recognise all patterns containing exactly one dot while rejecting all other images — see Crevier p 105.

Question 2: You read a paper whose author claims to have made a pattern recogniser that gives the output $(1, 1, 1, 1)$ for the input $(45, 23, -1, 4)$ and gives $(0, 0, 0, 0)$ for every other input. He's done this by connecting 37 linear associators in sequence. Do you believe him?

Answer 2: This is also impossible. If you connect any number of linear associators together, the result is still linear (and can be represented as just one associator). [1] I once heard (maybe apocryphally) of an academic who wrote several papers claiming to have implemented non-linear operations by combining linear nets!

3 Some limits on neural networks — Hopfield net

In this section, I look at an example of limits on the amount of information that can be learnt.

Now to a different kind of net, the *Hopfield net*. We have a set of neurons. Each neuron has a synaptic connection to every other neuron. These connections also have weights, and the weights are symmetric, in that neuron 1 affects neuron 2 in the same way that 2 affects 1. There's an updating rule by which pairs of neurons adjust their activations so as to minimise their "computational energy". This rule is repeatedly applied until the whole system is in a stable state.

The best way to think of this is via a physical analogy (which Hopfield nets were developed from). Consider a set of compass needles arranged on a board. A needle's position corresponds to its activation. Each needle affects each other, via their magnetic fields. The strength of this connection is usually fixed, determined by the rate at which magnetic fields attenuate in the air. However, it could be changed by

[1] Each net corresponds to a matrix. The result of connecting them corresponds to the result of multiplying the matrices. This is itself a matrix, and is linear.

interposing something which blocks off such fields, such as soft iron. If you position two needles so that they repel each other, they will be in a state of high potential energy. They'll move so as to minimise their PE by moving to attract each other, i.e. one's south pole next to the other's north pole. If you have more than two needles, the whole set will adjust itself to find the state of lowest PE.

You can consider all possible positions of the needles, and plot PE against these positions. This gives you a *potential energy landscape*. The points of lowest energy or *minima* are, not surprisingly, the lowest points in the landscape.

Going back to nets, these points of lowest energy correspond to stored memories or patterns. Start the net off anywhere else, and it will "roll down the energy landscape" until it falls into one of the minima. The shape of the energy landscape is determined by the synaptic weights, and is determined during training. Retrieving a memory from a partial input corresponds to starting somewhere high on the landscape, and rolling into the nearest minimum.

So how many memories can be stored? You can do a mathematical analysis which shows that if you try to store too many, the local minima interfere. In effect, different landscapes get superimposed, and create false minima which don't correspond to real memories. Worse, the positions where the memories should be may no longer be minima anyway.

Furthermore, in any real system of compass needles, the needles will never be completely still, because of thermal excitation. If the temperature is too great, then once again, this will cause the same kind of effect as above. The same kind of thing happens in a Hopfield net. This is explained in detail, with diagrams, on page 40 of Herz. For a nice general intro to Hopfield nets, with explanation of potential energy landscapes, see the article on *Neural networks* in the *Encyclopaedia of AI*. [2]

4 Limits on learnability

We can see the first (exclusive-or) result as a limit on *learnability*. The linear associator's architecture is restricted in such a way that there are some things it just can never learn. Many such results have been derived for individual learning systems, whether connectionist or not: we're now beginning to see general theories of learnability. The idea is that we can see learning as *function-building*: our system has to learn a certain mapping M from its inputs to its outputs. But it only has a certain set of primitive functions available, those provided by its hardware and software. If the function M to be learnt can't be built up from these primitive functions, it's logically impossible for the system to learn it.

5 Limits on space

My result about the Hopfield net concerned *storage*. For any net, there is only a finite number of associations it can learn. That's not surprising since the net is finite. In general, any computational system will display limits of this kind.

6 Limits on speed — complexity theory

There are also results on *speed* — how fast a computer can perform certain tasks. Any computer, including a connectionist one, will have its own speed limits, but

[2] For a different computational system based on magnetism, see *A non-deterministic approach to analogy, involving the Ising model of ferromagnetism* by Hofstadter, AI box copy H114. This treats the solution of analogy problems as a "phase change", like the change from solid to liquid or liquid to gas.

I'll switch now to conventional serial machines. The idea is that, given a task such as sorting numbers, we can write a number of different *algorithms* (programs) for it. The time taken by each program will increase with the size of its input: it takes longer to sort 40 numbers than to sort 20. (Note that this has to be a statistical measure. If the 20 numbers are exactly in reverse order, and the 40 are exactly in the right order, some algorithms might process the 40 more quickly. But, on average, the time will increase with the size of input.)

What complexity theory studies is:

- Given a specific algorithm, how does *its* time vary with size of input? Is it as the square of the size, or the cube, or perhaps exponentially? And what are the worst- and best-case possibilities?
- How does the *best* algorithm for a task vary? Is it possible to write a sorting program whose time increases more slowly than the square of input size, for instance?

6.1 Sorting

One simple algorithm: the *bubble-sort*. Given a sequence of numbers, scan along it until you find a pair that's out of order. Interchange them. Go back to the start, and scan again. Repeat until all numbers are sorted. The time this takes increases (on average) with the square of the number of numbers.

Can we do better? Yes. I think the best possible algorithm that works by comparing pairs of numbers has a time which increases as $nlogn$, where n is the number of numbers input.

6.2 The travelling salesman problem

The problem: you have a number of cities, and you know the distance from each city to every other. A salesman has to construct a route which takes him through every city once and only once. Find the shortest route.

This is an *exponential* problem. It is logically impossible to construct an algorithm that solves the problem exactly whose computation time goes up more slowly than the exponential of the number of cities. There are many problems of this kind, often concerned with packing objects in a way that minimise the number of gaps. For example, scheduling advertising slots. See *Emperor's New Mind* by Penrose, pp 140–146.

7 Complexity theory and psychology

Complexity applies to parsing algorithms as to any other. Any parser will take a time which varies in some way on the complexity of its input. Hence one could try to infer the nature of the human parser by looking at how the time taken to respond to sentences varies with the input. I haven't found any references to this, but I think some psycholinguists have tried it.

8 Complete impossibilities

I've discussed problems that can't be solved in less than a given time. But there are also problems that can't be solved at all.

- *Diophantine equations*. A Diophantine equation is a polynomial equation such as $x + x^2 + 47y = 3$, where the solutions are restricted to being *integers*. It is impossible to write an algorithm that will tell you, for *any* Diophantine equation, whether it has a solution. *Emperor's New Mind* p 129.

- *Tiling the plane*. Is there an algorithm that will tell you whether, given any set of polygons, it's possible to cover an infinite plane with no gaps and no overlaps? Obviously one can do this for some sets, e.g. that containing just one square. But can it be done in general? Again, it can't. *Emperor's New Mind* p 132.

- *Topological equivalence of manifolds*. Informally, a *manifold* is (in two dimensions) a closed surface. For example: the surface of a sphere; the surface of a cup; the surface of a doughnut. Two manifolds are equivalent if you can deform one into the other without cutting or tearing. Think of each being made of infinitely elastic rubber. Can you write a program that takes two manifolds and tells you whether they're equivalent. For two-dimensional manifolds, yes. For four-dimensional ones and higher, no. For three-dimensional ones, no-one knows whether it's possible or not (as far as I know). It's very odd that one dimension should cause a switch from a soluble to an insoluble problem. *Emperor's New Mind* p 130.

- *The word problem*. You have an alphabet of letters. A *word* is just a sequence of letters. You also have a set of rules by which parts of words can be transformed into other words. Is it possible to write a program which tells you whether, given any words $w1$ and $w2$, you can go from $w1$ to $w2$ by using these rules? Once again, no. Again, it can't. *Emperor's New Mind* p 131.

9 The decision problem

You may think such problems aren't particularly important, though physicists would disagree, at least concerning the equivalence of manifolds. But another impossibility result is very significant, because it puts a very general limit on what we can ever prove with formal reasoning.

A bit of history first. Many philosophers and mathematicians in the early part of the century were logical positivists, i.e. took the viewpoint that a statement was only meaningful if one could specify a sequence of operations which would decide its truth or falsity. So "the moon is made of green cheese" is meaningful, because one can specify spectroscopic tests to determine its consituents, but "God is good" isn't, as there are no operations one can perform to examine God. Bertrand Russell was one of the best known positivists.

This stance was closely related to the *formalist* position in mathematics. This was the view (or hope) that all of mathematics could be reduced to a set of symbol manipulations. Given any set of axioms, and a conclusion that you wanted to prove or disprove, all you needed to do was to manipulate the symbols according to certain formal rules for constructing proofs. There was no need to appeal to any other notion of truth. In other words, an operational method should exist for determining the truth or falsity of mathematical operations.

The German mathematician David Hilbert was a formalist. In the 1920s, he posed the *decision problem* (also known as the *entscheidungsproblem*, from the German), as part of his programme for the foundation of mathematics. Is this formalist stance valid? Given a finite list of logical premises and a conclusion, is there a set of rules (i.e. an algorithm) that will tell you — for every possible set of premises and conclusion — whether the conclusion can be proven from the premises?

In 1931, Gödel answered part of this. Given any logical system that's powerful enough to describe arithmetic, it's impossible to show, by using the system's own rules of inference, that it is both consistent and complete. *Consistent* means that the system can't prove something to be both true and false. *Complete* means that it's possible to prove *every* statement either true or false. What Gödel showed was that in any logical system that's powerful enough to describe arithmetic, there is a statement that you can show to be true by reasoning outside the system, but that the system's own rules of inference can't prove.

Hilbert's programme also called for solution to whether any axiomatic system can be *decideable* — i.e. whether there exists a definite method which in principle can be applied to determine the truth of any assertion. Gödel had answered part of this for arithmetic, but the general problem — whether for any logical system, you could prove every statement — was still unresolved. Turing was attracted by this, and decided to work on it. In 1936, he showed the answer that there wasn't. In doing so, he also defined and answered the *halting problem*. So I need now to describe the halting problem, and to talk about Turing's notion of "definite method" or computation.

Incidentally, I am typing this in a great hurry, and even if it's intelligible, it may not be completely accurate. There are various references. A book which should shortly appear in the library is *Searching for Certainty* by John Casti. Chapter Six is a popular account of Turing machines, Gödel's theorem, the limitations on theories, and so on. AI photocopy D77, *What is a Computation?* by Martin Davis (from *Mathematics Today* edited by L.A.Steen, Springer-Verlag 1978), gives a slightly more technical account, including the proof of the halting problem for Turing machines, and an unsolveable word problem.

10 Turing-equivalent computers, and Turing-computable problems

I've mentioned that one can't write algorithms to solve the Entscheidungsproblem, the word problem, the Diophantine equation problem, the problem of whether any set of polygons will tile the plane, or the topological equivalence of manifolds in 4 dimensions or more. To anticipate what I'm going to say later, Turing showed that all computers were equivalent to each other, and to an "ideal" computer called a Turing machine. If an algorithm doesn't exist for one computer, it won't for any. So I'll say in future that problems like the above, for which there exists no algorithm, are *non Turing-computable*. Sometimes, I'll say *non-computable* for short. Other problems are Turing-computable, or computable for short.

I'll say that any computer which is equivalent in power to a Turing machine is *Turing equivalent*. This covers all digital computers. It is also an upper limit on connectionist systems. Some (e.g. the linear associator) may have less power than a Turing machine, in that they can't compute something it can. But none have more power.

No-one has yet built a computer that can compute more than a Turing machine can. But some people believe that one could do so using quantum effects, and I'll talk about that next week. Some people, such as Penrose, also believe that the brain uses such effects, and hence can solve non-Turing-computable problems.

11 The halting problem

In this section, I describe the *halting problem*. As mentioned above, this is another non-computable problem. What's the relevance to AI? See next week. But briefly, if the brain can solve non-Turing-computable problems, then complete AI will be impossible to engineer on Turing-equivalent computers, and any cognitive model running on a Turing-equivalent computer will give an incomplete account of the mind.

Now to the halting problem. Suppose yourself to be a programmer. A client asks you to write a program, "H" (in any language, and running on any computer, of your choice) whose specification is as follows. The client is allowed to give it the text of any program as input. Your program, H, must work out (by any means you want) whether the program it was given as input will eventually stop, or go on for ever. If the former, it must print "Halts". Otherwise, it must print "Doesn't halt".

So, if H is given these programs, it will print "Halts":

```
Program 1:
           i := sin(3.1415);

Program 2:
           i := 1;
    next:  if i mod 2 = 0 then goto exit;
           i := i + 1;
           goto next;
    exit:
           write("Found an even number");
```

But with the programs below, it will print "Doesn't halt".

```
Program 1:
    next:  write("Looping");
           goto next;

Program 2:
           i := 0;
    next:  if i>0 and i<0 then goto exit;
           i := i + 1;
           goto next;
    exit:  write("Found a negative positive number");
```

Turing proved that program H is impossible to write. It's a logical impossibility, not a restriction imposed by technology or skill, in the same way that no number can be both odd and even, or no person can be both taller than 6 feet and shorter than 5. To prove it, Turing showed that if such a program existed, it would, when given itself as input, both halt and run forever. Since this is impossible, program H must be impossible too.

Obviously, you can analyse that *some* programs will or won't halt — e.g. by looking for patterns like `l: goto l`. But that's impossible in all cases. To see why, consider a program constructed to generate the digits of π, and stop only if it finds a string of 9 9's. It would not be easy to write a pattern recogniser that tells whether this would ever halt!

But it's easy to state what H must do: I just have. Therefore, there is at least one task you can define, but not make a computer do.

11.1 A short proof

I've taken this from a letter by Christopher Strachey to the *Computer Journal* for January 1965, volume 7, number 4. There's a copy as AI photocopy S170. He used a programming language called CPL which is now extinct, but the basic idea works in any programming language where functions (procedures, subroutines) can take other functions as arguments. I've written the proof using Pascal-like notation, but without the types.

Assume we have written a function called `T` (T for "terminates"). `T` takes another function as argument, and returns 1 if that function will eventually stop, and 0 if it will go on for ever. For example, suppose we have these two functions `f` and `g`:

```
function f;
begin
    f := 1;
end;

function g;
begin
next: goto next;
end;
```

Then `T(f)` is 1; `T(g)` is 0.

Now we write another function `P`:

```
function P;
begin
    l: if T(P) = 1 then goto l;
end;
```

Now, what's `T(P)`? It has to be either 0 or 1, so start by assuming it's 0. Look at the definition of `P`. If `T(P)` is 0, `P` won't goto `l`, but will just return. So `P` terminates. But that's a contradiction, because `T(P) = 0` means that `T` is saying that `P` won't terminate.

So assume instead that `T(P)` is 1. Look again at the definition of `P`. Now, `P` will goto `l` and will loop forever. So `P` does not terminate. But that's also a contradiction, because `T(P) = 1` means that `T` is saying that `P` will terminate.

So either way, we hit a contradiction. Our initial assumption — that we could write a function `T` — must therefore be false.

Note: strictly speaking, this only works in a language where a function can take *any* other function as an argument. If you know Pascal, you'll know that's not the case there, because of type-checking (though I think it is in C). In such cases, you need to make a more indirect proof. This involves encoding the text of the procedure as a number, and using that number as an index into a catalogue of all possible functions. But the basic structure is still the same.

11.2 Other proofs

All the proofs I've seen work in a similar way, by reductio ad absurdum. They assume a termination-testing program can exist, show that if it does, then a contradiction always arises, and therefore conclude the initial assumption to be false. In most cases, the contradiction is a program that stops if and only if it doesn't stop.

What Machines can and cannot do by Nievergelt and Farrer, *Computing Surveys* volume 4 number 2, June 1972. Available as AI photocopy N 21. Their proof has the same structure as in Strachey's letter, but the termination tester is passed the index number for a program rather than the program itself.

Incomputability by Allison and Hoare, from *Computing Surveys* volume 4 number 3, September 1972. Available as AI photocopy H136.

12 What is computation? The Turing machine

Turing made his proof of the decision problem or entscheidungsproblem in 1936, *before* there were any programmable computers. He was concerned with what can be computed or calculated by a human who is following definite rules, i.e. by a human doing mathematics as the formalists say it should be done.

He first had to formalise the notion of a "definite method". He did this by constructing a mathematical model of somebody following a definite set of rules. This model was the *Turing machine*. It:

- used a finite alphabet of symbols. Clearly, this is what mathematicians use when working.
- wrote these on tape divided into squares, each square capable of holding one symbol. This is an idealisation of the 2-d piece of paper that the mathematician works on. Its two-dimensionality adds nothing essential, so without loss of generality, we can model it as a 1-d tape.
- had a finite number of possible "states of mind". If the number of states of mind were infinite, then some states would have to be arbitrarily close, and hence could not be distinguished. (This also applies to the alphabet of symbols.)
- can at any time inspect one symbol, that which is in the tape square under its "eye". (The "eye" is usually termed a *read head*.)
- has a behaviour which is determined solely by the current symbol under its "eye", plus its current "state of mind". In effect, the machine contains a "state table" which maps every combination of these two to a new state plus some behaviour. This is the assumption of physical determinism. You can think of many machines which embody such state tables: e.g. a coffee machine, whose current "state of mind" is determined by the value of the coins inside it, or a washing machine, whose current "state of mind" is determined by the signals from its timer.
- builds its behaviour from very simple operations. It can write a symbol, change its state of mind, move the tape left or right. We lose no generality by restricting ourselves to such simple operations, because we can combine them to produce arbitrarily complex behaviours.

See the essay by Andrew Hodges in *The Universal Turing Machine* edited by Herken (PSY KD:H 42) for a summary of the history behind Turing's work. Other essays describe the mathematical background, but may be too technical. Penrose pages 47–48 explains why the Turing machine is an adequate model of what human mathematicians do when following rules.

12.1 The Universal Turing Machine

Turing also showed that he could define a *Universal Turing Machine*. By writing suitable symbols on its tape, you could program this to emulate the behaviour (the state table) of any other Turing machine. In the same way, we can program a computer to simulate coffee machines, washing machines, and any other system.

Turing then proved the halting problem by considering a UTM program which was to test whether every other Turing machine would eventually halt. By applying this program to itself, he arrived at a contradiction, similar to the one I demonstrated earlier.

13 Uncomputable numbers

Turing used the halting problem to demonstrate that there are some functions (or numbers) that one can describe but not compute. To see this, consider that, given any programming language, one can assign a unique integer to each program. E.g. by sorting the programs in order of length, and numbering them from 1 upwards. Now, imagine the real number h whose Nth digit it 1 if the Nth program halts and 0 otherwise. Since the halting problem can't be solved, it's impossible to compute this number. Nevertheless, it is true in fact that every program *does* either halt or fail to halt. Hence every digit of h must be either 0 or 1. But we can never know which.

14 Uncomputable numbers and infinity

Another way to see that some numbers are uncomputable is by pairing them off against programs. Basically, there are more numbers than programs. So there are some numbers that can't be produced by any program, i.e. can't be computed.

To see this, number all programs, as in the previous section. Then say that the nth program P_n computes a real number r_n. Now, make a number whose first digit is different from the first digit of the number r_1, i.e. whose first digit is different from that output by the first program. Similarly, let its second digit be different from the second digit of r_2, i.e. from the second digit output by the second program. No, since r's first digit is different from that output by P_1, r is not the number that P_1 produces. Similarly, it isn't the number that P_2 produces. And so on. So r is a number that can't be produced by any of our programs. [3]

[3] Continuing the above, we can use similar arguments to show that there are some physical phenomena that we can't describe by any theory. In essence, we can view a theory as a set of rules (i.e. a program) that predicts (i.e. outputs) the observations it explains. But this theory (program) will be stored in a brain or a computer (a finite machine). But a finite machine can hold only a finite number of programs. So it can predict only a finite number of different observations. If the observations we're trying to predict don't fall in that set, no program (theory) our machine can hold will be capable of explaining them. See page 357 of Casti. Is psychology too complex to be described by any theory we can store?