Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 317 (2004) 13–30
Theoretical Computer Science
www.elsevier.com/locate/tcs

Toward a theory of intelligence

Peter Kugel*
Computer Science Department, Boston College, Chestnut Hill, MA 02467-3808, USA

Received 16 May 2003; received in revised form 19 October 2003

Abstract

In 1950, Turing suggested that intelligent behavior might require "a departure from the completely disciplined behavior involved in computation", but nothing that a digital computer could not do. In this paper, I want to explore Turing's suggestion by asking what it is, beyond computation, that intelligence might require, why it might require it and what knowing the answers to the first two questions might do to help us understand artificial and natural intelligence.

Keywords: Intelligence; Artificial intelligence; Hypercomputers; Super-recursive algorithms

1. Introduction

In 1950, Turing [24] wrote a famous paper in which he planned to discuss the relationship between computing machinery and intelligence. He changed his mind before he got very far when he realized that, although he had a precise definition of "computing machinery", in terms of what we now call the "Turing machine" [22], he lacked a precise definition of "intelligence". So, instead of discussing intelligence, he changed the subject [15] and discussed what he called the "imitation game" [24] instead. [1]

If we try to use the ability to play the imitation game as a definition of "intelligence" (as some people have done), we quickly notice that it is imprecise and (as I shall suggest) misleading. Turing's purpose in suggesting it was not to define "intelligence" but to draw "a fairly sharp line" [24] between what counted toward intelligence and

* Tel.: +1-617-552-3979; fax: +1-617-552-6790.
E-mail address: kugel@bc.edu (P. Kugel).

[1] The imitation game is played by a computer against a person. Both players are allowed to communicate with a human judge only by means of a computer terminal and both try to convince the judge that they are the person in the game. The computer wins if it can fool the judge for a specified period of time.

0304-3975/$ - see front matter
doi:10.1016/j.tcs.2003.12.002

what did not. In other words, it was intended primarily to say what intelligence is not, rather than what it is. That was the best Turing thought he could do.

But, toward the end of his paper (and elsewhere) he did manage to say a few things about what he thought intelligence might be and how it might be related to computing machinery. In effect, he said that:

- The machinery of the computer is probably powerful enough to produce intelligent behavior [24]. In other words, intelligence does not require any new, more-powerful, hardware. (In other words, it does not require what Copeland and Proudfoot [6] have called a *hypercomputer*.)
- But the machinery of the computer will probably have to be allowed to do more than compute before it can be made to behave intelligently. As Turing [24] put it, "Intelligent behavior presumably consists in a departure from the completely disciplined behavior involved in computation, but a rather slight one, which does not give rise to random behavior, or to pointless repetitive loops". (In other words, it might require what Burgin [3] has called *super-recursive algorithms*.)
- The required "departure" will probably require allowing the computer to make mistakes because, again in Turing's words [26], "if a machine is expected to be infallible, it cannot also be intelligent".

In this paper, I want to suggest a mathematical model of the mind in terms of which I then want to try to develop a characterization of "intelligence" that is more precise and, hopefully, more accurate than the imitation game. I am not going to argue that the resulting characterization is necessarily what Turing had in mind, but I am going to suggest that it might be both interesting and useful.

2. Intelligence

The *American Heritage Dictionary* defines *intelligence* as the ability to "acquire and apply knowledge". Although dictionaries can be wrong, let us take this definition seriously (at least for the duration of this paper) and think of potentially intelligent machines as having two basic components. One component (the *learner*) acquires knowledge. The other (the *doer*) applies the knowledge that the learner acquires. If we assume that knowledge can be represented by computer programs (or something like them), we can think of the learner as a system that generates programs and the doer as a system that runs the programs generated by the learner to do useful things (see Fig. 1).

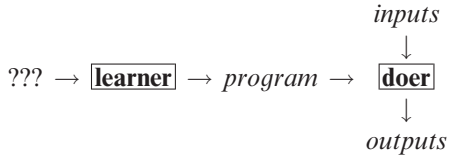

Fig. 1. The two components of an intelligent system.

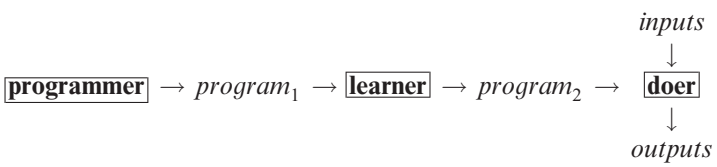

Fig. 2. Today's "intelligent" systems.

It is not enough, of course, for a machine merely to have these two components. Before a *learner*/*doer* system can be considered intelligent, those components will have to perform at a certain level and one of the main questions I want to ask in this paper is "What level might that be?"

The level reached by much of today's work in Artificial Intelligence is not, I will argue, high enough. Consider, for example, a computer program that uses some form of tree search to play chess well enough to convince people who believe that intelligence is simply the ability to simulate intelligent behavior that it is intelligent. Such a system (see Fig. 2) has a powerful doer that does the actual playing and produces the required behavior.

But that is not enough for a learner/doer system because its learning component is too weak. All it does is translate a program, given to it by a human programmer, from one language (a "high-level" language) to another (a "low-level" language). That is not a job that seems to require a lot of intelligence (although writing a program that does that job—a *compiler*—may require a great deal).

Perhaps, it is because the learner contributes so little to the system's total intelligence that some critics complain that, however intelligent such a chess-playing system might appear, much of its apparent intelligence is provided by the humans who wrote the program that the learner only translates.

It makes a certain amount of sense to assume that intelligence requires a learner that can get along with instructions that provide fewer details than such a chess-playing program requires. (We do tend to think of a person who has to be told exactly what to do as not very intelligent.) But what?

People seem to learn to do things (such as play chess) from a variety of sources—from examples, from vague instructions, from analogies and the like. Although machines can learn from any of these sources, I propose to focus on systems that learn from examples.

That is how a child first learns to use its native language, an ability that it needs before it can deal with instructions and analogies. It is also an ability one can be asked to demonstrate on an intelligence tests when one is asked to continue such series as $2, 4, 6, \ldots$. To do this one presumably has to first come up with an algorithm that would generate, not only the given part of the series, but the rest of it and then use that algorithm to generate a few more.

We can think of such a learner-from-examples as a system that tries to develop a program that can simulate the behavior of the device whose inner workings it cannot examine. All it has to work with is the behavior (inputs and outputs) of the device. (see Fig. 3)

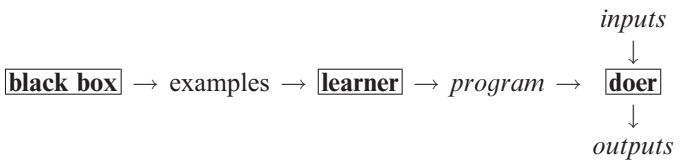

Fig. 3. A prototypical intelligent system.

Such a system might learn to play legal chess by watching a person play or by playing games against a strong opponent.

Is it not easier just to tell a system what the rules are? It can be. It is probably easier to tell people the rules of chess than to ask them to figure them out by watching games. But there are situations in which people simply cannot be told what the algorithm is (When they begin learning their native language, there seems to be no language in terms of which they can be told.) and situations in which learning from examples is easier. Thus, for example, people may learn the rules of chess by being told them, but few learn how to play chess well only by being told what to do. Being told what to do can help, but lots of playing (which is to say, lots of examples) generally help more.

Few children, if any, have ever learned the meaning of the word "dog" from detailed instructions that tell them how to recognize one. Even if children could learn from such instructions, few parents could produce them. Most do not try. They just point to a few examples and let the child's mind do the rest. They can do that, presumably, because the child is intelligent—because it can learn from examples.

3. If intelligence is computable, then...

We can think of a learner that tries to develop programs from examples, as a system that tries to solve what has been called a *black box identification problem*. In such a problem, a learner is given the behavior of a device it cannot look inside of (a *black box*) and asked to *identify* it by coming up with a program that exactly duplicates its behavior. This problem is not particularly difficult (in theory) if the behavior is finite and the learner has access to all of it. But it becomes more difficult if the behavior is (at least potentially) infinite, because the learner must come up with a program that duplicates all of it in finite time. That means that it must do so after it has seen only a finite part of that behavior. Thus, the program it comes up with must characterize all of the black box's behavior, including the parts the learner has not yet seen. This means that it must "go beyond the information given" in a way that, I shall argue, computations cannot.

Let me introduce a bit of terminology. Let me call the black box to be identified the *sample-device*. The sample-device will generally be drawn from a set of possible devices that I propose to call the *domain*. (Intuitively, the sample-device represents "this world" whereas the domain represents "the set of all possible worlds".) If the

learner succeeds in coming up with a (finite) program that exactly reproduces the (usually infinite) behavior of the sample-device, we will say that it has *acquired* that sample-device. It can then pass its program (or *theory* of the black box's behavior) to the doer. The doer can then use that program to predict the future behavior of the sample-device and use those predictions as it sees fit. When the learner/doer acquires a sample-device it can predict its future behavior and that can be useful.

In acquiring a sample-device, the learner, in effect, turns the infinite behavior of the sample-device into a finite program, or theory. This process of compressing infinitely many examples, most of which the learner has not yet seen, into general theories is a model of what is often called *induction*. (We see a hundred swans, all of which are white, and infer, by induction, that "All swans are white".)

I am going to try to characterize what we might call, following Chomsky [5], the abstract *competence* that underlies this process rather than what he called the concrete *performance*. So I am going to assume, among other things, that the inputs to the learner are error-free and that the learner/doer never makes clerical or computing errors. In exchange, I am going to assume that the learner must be able to produce the future (and past) behavior of the sample-device without error before it can be said to have acquired it. (We can consider the effects of errors and other realistic limitations after the model has been developed, much as we consider the effects of air resistance on falling bodies only after we have developed a theory that ignores it.)

We will call the set of all devices that a given learner can successfully acquire its *range*. And we will say that a learner can *cover* a set of devices (or domain) if it has that set as its range.

To explore what all this means, let's look at an example. Suppose we want to design a learner to deal with a rather restricted domain—the set of all finite automata that generate infinite sequences of 0's and 1's. Let us call such devices *Zero-one generating finite automata* (or *Zog-fa's*). A Zog-fa has no inputs and its outputs can be represented by a one-way (to the right) infinite tape that contains only 0's and 1's.

We will call such a tape an *oracle* (following Turing [23]). With each such tape we can associate a representing function, f, such that $f(n)=0$ if the nth symbol of the tape is 0 and $f(n)=1$ if it is 1. We can then think of a learner that tries to acquire a Zog-fa as a system whose input is the oracle of such a machine and whose output is a program that generates that oracle.

If we ignore the inner division of a learner/doer into two components, and assume that the doer only executes the program passed to it by the learner, the outward behavior of the learner/doer system does not seem to be particularly interesting. When it succeeds in acquiring an oracle, its input is that oracle and its output is a program that can compute that oracle. It is because such systems produce finite programs that are portable and because they can output the "next" symbol of the oracle before they read it that such systems are useful. Those finite programs correspond to what biological organisms carry around with them and use to predict the future behavior of things in their environment. (If the object such a learner/doer is dealing with is a predator, the resulting predictions can help the system predict the predator's behavior and use those predictions to try to avoid it, thus helping the system to survive.)

Our oracles resemble Turing's oracles in that they allow a Turing machine with access to the oracle associated with the function f to answer questions of the form "What is the value of $f(n)$?" for arbitrary n's. Our oracles differ from Turing's oracles in that, whereas his were limited to uncomputable functions, ours are limited, at least in this paper, to computable ones. Such oracles are particularly well-suited to represent the inputs to a learner trying to solve a black-box identification problem because a learner, with access only to an oracle, cannot "look" inside the machine that generated it.

(*Notes*: In order to talk about such oracles, I will denote the sequence consisting of all ones by "1^*" and, more generally, the sequence that consists of the infinite repetition of the finite sequence s by "s^*" [13]. I will denote the sequence that consists of n repetitions of s by "s^n", the first n symbols of s by "${}_ns$" and the result of concatenating one sequence, s, to another, t, by "st".)

Before looking at possible designs for learners that can acquire Zog-fa's, let me state an easily-proved lemma [13] about such Zog-fa's that will prove useful in what follows.

Lemma. *For any sequence of the form* $s(t)^*$, *where* s *and* t *are finite sequences of* 0's *and* 1's, *there is a Zog-fa that generates it and, conversely, any infinite oracle generated by a Zog-fa must have this form.*

As this lemma suggests, the Zog-fa's are a particularly simple domain and I want to begin by looking at how well learners can deal with it by using computations alone. The defining features of an ordinary computation are (1) that it may use only the machinery of the Turing machine and (2) that it may produce (at most) one output and that it must halt when it has done so. When I want to distinguish ordinary computations from other kinds (such as the limiting computations to be discussed later), I will explicitly refer to them as *ordinary* (or *recursive*) *computations*.

One of the big problems with learners limited to ordinary (recursive) computations is that they have very limited scopes. For example:

Theorem 1. *No learner, limited to ordinary computing, can cover even so simple a set as the set of all Zog-fa's.*

Proof. Assume there is a recursive-computing learner that can. Since it covers the whole set, it must be able to acquire a program for the Zog-fa whose outputs are all 1's from its oracle 1^*—the tape containing an infinite sequence of 1's. (Obviously, there is a Zog-fa that generates such a tape.)

Now consider what happens when a computing learner acquires the device whose oracle is this sequence of all 1's. By the definition of *computation* there must come some point in its operation at which it outputs a program for this oracle, announces that it has found it, and stops. And this point must come after it has read some finite number of symbols (say n) of the input oracle. Since, for the rest of this argument, the size of n does not matter, let us assume $n = 10$. That means that our learner's output is fixed after it has read the sequence of ten 1's (or 1^{10}). Which, in turn, means that

it cannot come up with the right program for the tape $(1^{10})0^*$. But clearly there is a Zog-fa that generates this tape too (by our lemma). □

Corollary 1.1. *For any Zog-fa a recursive-computing learner can acquire, there are infinitely many Zog-fa's it cannot.*

In the above case, it cannot acquire any of the Zog-fa's whose oracles have the form $(1^n)0^*$, with $n \geqslant 10$. Clearly each of these oracles can be generated by a Zogfa (by our lemma again) and there are infinitely many of them.

Since any computing learner that could cover a superset of the set of all Zog-fa's could also cover the set of all Zog-fa's, it follows that:

Corollary 1.2. *No recursive-computing learner can cover any superset of the set of Zog-fa's. Therefore, no computing learner can cover the set of all finite automata, all Turing machines, all primitive recursive functions, and many other domains.*

Oracles are infinite sequences of 0's and 1's that can be ordered lexicographically. Let us say that one oracle, O_1, is *less* then another O_2 (or $O_1 < O_2$) if O_1 is earlier in this ordering than O_2. Let us call the set of all oracles between the two different oracles, O_i and O_j, an *interval*. Let us call an interval a *gap* in a learner's range if that learner can acquire at most one oracle in it. Gaps are clearly undesirable since any interval has infinitely many Zog-fa's in it and, if it is a gap, a learner can acquire at most one of them. So it is another strike against our computing learner that:

Corollary 1.3. *A recursive-computing learner, whose domain is the set of all Zog-fa's, has at least one gap in its domain (and therefore, infinitely many).*

Proof. If it acquires an oracle, s, a computing learner must do so after it has read n symbols of that oracle, or ${}_ns$. But, since it has only one chance to come up with a program, it must come up with the same program for all oracles that start with this sequence. That is precisely the set of oracles in the interval between ${}_ns(0)^*$ and ${}_ns(1)^*$. Since the one program (at most) that it can come up with is correct for only one oracle in that interval, that interval is a gap. □

Perhaps the most serious drawback of recursive-computing learners is that learner/doers that use them can be very bad at predicting future symbols of their input oracles. Let us call a system consisting of a learner and a doer related to each other in the manner shown in Fig. 1 (above) a *predictor* if it reads an oracle, a symbol at a time (from left to right), and uses the latest program output by its learner to predict the next symbol of that oracle before it reads it. (If the learner has not yet produced a program, let it predict 0.)

Let us say that a predictor is *good* for an oracle if it predicts infinitely many symbols of that oracle correctly (and only finitely many incorrectly) and *bad* if it predicts infinitely many symbols of that oracle incorrectly (and only finitely many correctly).

Corollary 1.4. *For every oracle that the learning component of a computing predictor acquires, there are infinitely many oracles of Zog-fa's for which it is bad.*

Proof. Consider a predictor, P, that computes a program for an oracle, s, after reading the finite sequence ${}_ns$ (the first n symbols of s) for some n. Let s_n be all the symbols of s after the first n symbols and, for any sequence u, let $(u)'$ be the result of replacing every 1 in u with a 0 and every 0 with a 1. It is not hard to see that (a) P will predict the symbols of any oracle of the form $({}_ns)(1^m)(s_{n+m})'$ (where $m \geqslant 0$ and s_{n+m} is the sequence of the input oracle after the first $n+m$ symbols) incorrectly infinitely often and correctly only finitely often and (b) that any such oracle is in its domain because it can be generated by a Zog-fa. □

This "badness" is not limited to learners dealing with the set of Zog-fa's. It applies to any computing learner that deals with a domain that is *dense* in the sense that, between any two distinct oracles in that domain, there is another that is also in the domain. (More generally, a domain is dense if no finite initial sequence uniquely determines an oracle in that domain. That condition is met by practically all interesting domains.) It is not hard to see that:

Corollary 1.5. *No recursive-computing learner can cover a dense domain.*

Because recursive-computing predictors have to stick to a program long after the evidence has shown it to be incorrect (since they are not allowed to "take back" a result) they can seem somewhat "pigheaded". The computable learner that produced a program that generates 1^* as its theory after seeing 1^n, forces the associated doer to predict 1, even if the input was 1^n0^*. So, if n is 10, it is still required to predict that the next symbol will be a 1 after it has seen $1^{10}0^{999990}$, which should make the odds of a 1 appearing next "one in a million". It is difficult to call that "intelligent".

The reason computing learners do so badly with such domains is that computations must, by the definition of *computation*, satisfy what we might call the *announcement condition*. It is not enough for them to produce a result in finite time. They must also announce when they have produced it. If we drop that requirement, we can get machines that do more than compute without changing anything else. Machines that do not have to satisfy the announcement condition can execute what Burgin [3] has called *super-recursive algorithms*—algorithms that can evaluate uncomputable functions. In particular they can execute *limiting-computable algorithms* [10,19].

The distinction between what we might call *recursive computing* (which is the traditional kind) and *limiting computing* (which is one kind of super-recursive computing) can be stated quite simply. When we use computing machinery to *compute recursively*, we count the *first* output it produces as its *result*. In contrast, when we use that same machinery to *limiting-compute* or to *compute in the limit*, we count its *last* output as its *result*. Note that we do not require that it announce when it has produced that last output nor to halt after it has done so (which would, of course, "announce" it).

Putnam [19] has referred to predicates that can be computed in the limit as *trial-and-error* predicates. In that spirit, let us call Turing machines that are allowed to compute

in the limit *trial-and-error machines*. We can think of such a machine as a computing machine that prints a sequence of outputs. We say that its "result" is the last output it prints, if there is a last. (If there is no last—or no first—the output is said to be *undefined*.) Notice that trial-and-error machines (and the limiting-computable algorithms they execute) produce their results in finite time, but that it can take infinitely long to determine that a given output is a result because they can always "change their minds".

4. If intelligence is limiting computable, then...

Trial-and-error machines use precisely the same "hardware" as the Turing machines, but because they do not have to satisfy the announcement condition, they can use it differently. The fact that they do not have to satisfy the announcement condition makes them useless for many purposes, including most of the purposes for which we use computers today. But it makes them more powerful than machines limited to recursive computations and this additional power can suit them for other things. Thus, for example, it can make them better at generating programs (or theories) from examples.

First, to see that limiting-computable algorithms (and the trial-and-error machines that execute them) are more powerful than regular (recursive) computations, note that they can solve the halting problem which, Turing [22] proved, no regular computation can.

Recall that the *halting problem* is the problem of finding a single procedure that, given a program, Prog, and an input, Inp, will tell us whether or not Prog(Inp) (or Prog running on the input Inp) will or will not halt. A limiting computation can solve this problem using the following algorithm:

Limiting computable algorithm for solving the halting problem: Given a program, Prog, and an input Inp, output NO (to indicate that Prog(Inp) will not halt). Then run a simulation of Prog(Inp). (Turing [22] showed that such a simulation is always possible.) If the simulation halts, output YES to indicate that Prog(Inp) really does halt.

Clearly the last output that this procedure produces solves the halting problem, if you are willing to accept results arrived at "in the limit". Which proves that a limiting computation can do things no ordinary, or recursive, computation can.

Of course, if you are interested in using your solution to detect cases in which a program is not worth running because it will not output any result at all, this "solution" will not help you. Although you can use its YES answers for your purpose, you cannot use the NO answers. Because they do not satisfy the announcement condition there will not, in general, come a time at which you can say of a program "This program will not halt because the algorithm told me so". And, of course, for this purpose, it is the NO answers that matter.

Failure to meet the announcement condition is the main drawback of limiting computations. They may give you right answers, but you will not always know when an output is the final answer.

But limiting computations can be useful when we are dealing with problems that have no final answers—problems for which the announcement condition cannot reasonably

be met. The problem of generating programs for dense domains (or sets of oracles) is such a problem. It is not hard to see why. A computing learner can only see finitely many symbols of an oracle before it has to come up with its final decision and, if the set of oracles is dense, the decision it makes can always be wrong because no finite subsequence of an oracle uniquely specifies it. In contrast, a limiting computable learner need not make final decisions which allows it to do things that a (recursively) computable learner cannot. Thus they can cover dense domains. For instance:

Theorem 2. *A limiting-computable learner can cover the set of all Zog-fa's.*

Proof. A limiting-computable learner can do this using what has come to be known as an *enumeration method* [10,1]. An enumeration method uses a sub-program that generates a list of totally computable programs $(a_1, a_2, a_3, \ldots)$ such that, for any machine in the domain, there is at least one program in the list that generates it. (The fact that each program generated must be totally computable is important.)

An enumeration method uses the list generated by its enumerating sub-program as follows:

- It begins by outputting the first program in the list, a_1, as its (tentative) result.
- As it reads the symbols of the oracle it is trying to identify, one by one, it checks the nth symbol it reads against the nth symbol generated by the program that is its current result (a_i). If the symbols match, it does nothing and reads the next ($n+1$st) symbol of the oracle. If the symbols do not match, it goes on down the enumeration from its current theory, a_i, to consider $a_{i+1}, a_{i+2}, \ldots$ in turn. It then outputs the first program in this list whose first n symbols match the first n symbols of the input oracle—the symbols that it has already seen—if there is one.

It is not hard to see that there is a program that computably enumerates a set of programs such that (a) for any Zog-fa, there is at least one program in the enumeration whose output is its oracle and (b) all the programs in the enumeration are total (because the halting problem for finite automata is recursively solvable).

Suppose that a limiting computation is given an oracle for an arbitrary Zog-fa as its input. It will compute a correct program in the limit for this oracle because (a) a correct program for the input oracle will appear in the enumeration, (b) every incorrect program earlier in the enumeration will eventually be discarded and (c) once a correct program is reached, it will never get discarded. Which proves the theorem. □

Notice that, although the process of determining whether or not the current (tentative) result produced by such a system is correct (at any particular moment in time), and the process of finding a replacement for it if it is not, are both totally computable, the process of finding the right theory for an oracle, of which they are the primary components, is not.

It is not hard to verify the following "good news" about such a simple enumeration process:

(a) It will get the right program in finite time (even though, its user will, in general, not be able to tell when this has happened).

(b) Its final program will be "perfect" in the sense that, if it is used to predict the symbols of the oracle it will predict all of them correctly.
(c) The learner will leave no gaps in the set of all Zog-fa's.
(d) If a learner/doer system, trying to identify a Zog-fa, uses its latest program to predict the next symbol of the input tape before it reads it, it will predict infinitely many of its symbols correctly and only finitely many of them incorrectly.

But there are also two pieces of "bad news":

(e) The process will not tell the user if an input oracle cannot be produced by any Zog-fa. If it runs into such an oracle, it will only way it will signal that fact is by failing to produce a last output—in effect, by "changing its mind" infinitely often.
(f) The announcement condition cannot be met. The learner cannot announce when it has finally found the right theory of its input oracle. Which means that the user cannot, in general, be sure when the system's current theory is the right one.

The set of all Zog-fa's is dense so, by Corollary 1.5, the announcement condition cannot be met. As a result we lose the certainty that a recursive-computing algorithm can give us. We cannot be sure, at any point, that its current output is going to be its final result. But, when we are dealing with a dense domain, we do not really have a choice.

The set of all Zog-fa's is not the only domain that the enumeration method can be used to cover. It can cover any set of machines that can be recursively enumerated in such a way that all the machines in the enumeration are totally computable. So we have the following [10].

Corollary 4.1. *A learner that computes in the limit can cover the set of primitive recursive functions, the set of finite automata whose outputs are limited to a finite alphabet, the set of regular grammars, and a variety of other domains.*

Because limiting computable learners can cover some sets of sample-devices whose oracles are dense (such as the set of all Zog-fa's) it is natural to ask whether such learners can cover all possible dense sets of that can be computably enumerated. The answer is that they cannot.

To see why not, consider first systems that use the enumeration method to develop programs for oracles that they examine one symbol at a time, in order from left to right. Let us call them *simple enumeration learners*. Now consider the set of all oracles generated, not by finite automata, but by Turing machines that generate sequences of 0's and 1's without inputs. Let us call them *Zero-one generating Turing machines* or *Zog-tm's*. Such machines generate oracles by computing their component symbols, one after the other, from left to right. If, at any time, a computation of the "next" symbol fails to produce an output, the sequence (but not necessarily the computation producing it) stops and the oracle is finite.

Lemma. *No simple enumeration learner can cover the set of all Zog-tm's.*

Proof. Suppose there is one that can. Call it M. Consider the Zog-tm that generates an oracle, D_M (where "D" stands for "diagonal"), defined in terms of M as follows:

0. The first symbol of D_M is a 0.

n. If, after reading the first n symbols of D_M, M's latest program for D_M predicts that the $(n+1\text{st})$ symbol of D_M will be a 1, then the $(n+1\text{st})$ symbol of D_M is a 0. If it predicts a 0, its $(n+1\text{st})$ symbol is a 1.

It is not hard to see that D_M can be generated by a Zog-tm (if M is a simple enumeration learner) because its nth symbol can be computed for any n. And it is easy to see, from the construction, that D_M cannot be identified by M because it will change its output infinitely often. □

Theorem 3 (Gold [10]). *No limiting-computable learner can cover the set of all Zog-tm's.*

Proof. The only difference between an arbitrary limiting-computable learner and one based on an enumeration method is that the former may not always have a "current" program and that they need not read the symbols of the oracle in order from left to right or, for that matter, in any fixed order. It is not hard to alter the construction of D_M to take those problems into account. □

Let us call a learner *universal* if it can cover the set of all Zog-tm's and *partial* if it cannot. Theorem 3 says that limiting-computable learner can be universal.

The idea of a limiting computation is not new. Such a computation was used by Gödel [9] in his proof of the completeness of the predicate calculus. Turing [25] seems to have suggested that something like a limiting computation might be necessary for machine intelligence when he wrote that "There are indications that it is possible to make the (computer) display intelligence at the risk of its making occasional serious mistakes". Gold [10] and Putnam [19] showed how this idea could be formalized. The use of limiting computable procedures to do induction has been studied by many, including Burgin [2] and Kugel [14].

5. Therefore...

I have suggested that machine intelligence (and perhaps the human kind) might involve two separate abilities—the ability to acquire programs and the ability to apply them. I have looked at how this suggestion plays out in machines that deal with "worlds" that contain only Zog-fa's and Zog-tm's. And I have tried to show that, in these "worlds", acquiring programs at a level that we would be willing to call "intelligent"—because they do not lead to pigheaded predictors nor produce "gaps" in which a predictor is helpless—requires at least the ability to compute in the limit.

But, of course, we want to be able to talk about the intelligence of machines that deal with many other kinds of "worlds" (or domains). Fortunately, because our proofs depend only on a few properties of the domains with which machines deal, it is not difficult to show that they can be used to derive comparable theorems about machines that deal with many other kinds of domains.

For example, Theorem 1, which tells us that computable learners have some serious drawbacks, is not limited to learners that deal with (a) input-free, (b) zero–one generating, (c) finite automata or Turing machines. Similar theorems can be proved (in similar ways) about learners that deal with (a) machines with inputs as well as outputs, (b) machines that have a wide variety of different powers and (c) machines whose inputs and outputs are made up of more complex objects than just 0's and 1's.

Thus we can say that most, if not all, learners will have serious defects if they are restricted to (recursive) computing alone and have to deal with domains that are *dense* in the more general sense that any finite behavior—or sets of input–output pairs—of oracles of machines in the domain can be produced by at least two different machines in the domain. These defects have to arise if we expect the system doing the induction to satisfy the announcement condition of recursive computation.

Theorem 2 tells us that these shortcomings can be overcome by allowing a system to compute in the limit, which is one way to specify what kind of "departure from the completely disciplined behavior involved in computation" Turing [24] might have had in mind (albeit perhaps only vaguely) when he suggested that is what intelligence might require.

Since fallibility is built into limiting computations (even when the data and the operation of the machine are both error-free), I take this to be one way to specify what Turing [26] might have had in mind (again, perhaps only vaguely) when he wrote that "if a machine is expected to be infallible, it cannot be expected to be intelligent".

Theorem 3 tells us that, although limiting computable learners can handle some dense domains (containing only computing machines), they cannot handle them all with such methods. This is an important limitation and there are two ways we might deal with it. We might try to develop more powerful methods that can be universal, or we might accept partiality and do the best we can with it.

If we try to develop universal algorithms we might want to look into types of super-recursive algorithms that are more powerful than limiting-computable ones and that can, therefore, cover the set of all Turing machines. Burgin [4] has called abstract machines that use them "inductive Turing machines of the second order" and I have called them "hyper trial and error" machines [14]. A more general, and more abstract, discussion of such procedures in logical, rather than machine, terms can be found in Kleene's classic paper on the arithmetic hierarchy [12].

It is hard to see how we could use hypercomputers in place of super-recursive algorithms for this purpose [7]. Although hypercomputers (such as the neural nets of Siegelmann [21] or the quantum computers of Kieu [11]) can evaluate super-recursive algorithms, their ability to do so assumes that all the data they will need is potentially available at the start.

This allows them (in theory) to solve such classical problems as Turing's halting problem or Hilbert's 10th problem [17]—the problem of finding roots of Diophantine equations. We might want to say that learner/doers that could use such machinery in their doer components were more intelligent than learner/doers that could not. (After all, it need not be only the learning component of a learner/doer that determines the level of a machine's intelligence.) But it is not clear how their use could do much for the learner.

A typical hypercomputer solves a problem like the halting problem by (in essence) looking at the answers to an infinity of questions in finite time, asking "Does it stop after step 1"?, "Does it stop after step 2?" and so forth. This can be done (in principle, if not in fact) because the answer to each of these individual questions can be computed from the information given. The hypercomputer can compute the answers to all such questions and "look" at them in finite time. Based on what it "sees", it can come to a conclusion and satisfy the announcement condition.

Unless a learner/doer has some sort of device for seeing all the values of an oracle in some way—some sort of "crystal ball"—it is hard to see how such a hypercomputer could help.

Hyper trial-and-error algorithms and hypercomputers can be universal but they can be hard to implement or to use for learning from examples. In contrast, super-recursive algorithms are relatively easy to implement. They do, however, have two significant disadvantages. One is that their results are always tentative. The other is that they are (as Theorem 3 says) not universal. There is no single universal algorithm that can do anything any limiting-computable algorithm can do. Nothing that corresponds to Turing's "universal machine" or its concrete implementation, the general purpose digital computer. So, if intelligence is in the province of limiting-computing algorithms, an intelligent machine may have to run more than one algorithm at a time. Different algorithms to learn about different domains.

For the study of "natural" intelligence this suggests that the brain may have more than one learning module to deal with the world. So it is satisfying to see that there seems to be at least one case where this seems to be the case. The human brain seems to have one module for thinking about (and therefore, presumably, learning about) inanimate objects and another for dealing with humans [20]. These two modules seem to be physically distinct and therefore could use different algorithms. If the theory I am proposing here makes sense, we should expect to see more such modules in the brain.

For the development of "artificial" intelligence it suggests that we might want to recognize the fact that any learning algorithms we develop will only be able to deal with restricted domains. Learners that "only" cover the set of all finite automata are a particularly tempting domain to focus on at the start. They are simple. They might be powerful enough for many purposes, and they could provide a potentially useful test bed for studying limiting-computing learning algorithms.

If we use simple enumeration learners for this purpose, we will have to deal with the fact that they are enormously inefficient at deciding what program to consider when their current program is disproved by the evidence. (Looking through a fixed enumeration can result in a lot of very bad "guesses". Finding the right theory under such circumstances can be a bit like trying to write *Hamlet* by setting a bunch of monkeys to work on typewriters.)

The basic problem is that, when such a learner discards its current theory, it has to "decide what to think of next". There is no general "best" order in which to choose the next theory to consider and it is not easy to make the job of finding that next theory efficient. One way to speed up the search for the next theory would be to use knowledge of the domain.

For example, we know that any infinite tape generated by a Zog-fa will be eventually periodic, which is to say that it must be of the form $t(s^*)$ where t is a finite (possibly empty) sequence and s^* is a finite sequence repeated infinitely often. A learner that looks for such a t and s in the part of the input oracle it has seen so far can find the "next" program to consider more efficiently than one that works "mindlessly" with a fixed enumeration.

Another way to speed the search through the enumeration is by excluding possible programs or by reordering the enumeration. People who take intelligence tests, for example, are sometimes asked to continue sequences like $2, 4, \ldots$ If the set of allowable continuations were not severely limited, such a question would make little if any sense. But, in fact, in IQ tests continuations of such sequences are usually implicitly limited to sequences built up from the rules people learn in school. And those rules are also ordered. So the sequence 2,4,6,8, built on addition, is usually thought to be a better continuation of $2, 4, \ldots$ than $2, 4, 8, 16, \ldots$, built on multiplication, because addition is thought to be "easier" than multiplication. And sequences like $2, 4, 5286, 35432, \ldots$ are not considered at all because they have no simple arithmetic explanation. Such limitations (and orderings) make it much easier (and faster) for a learner to "decide" what to "think of" next.

As people who work in artificial intelligence know, sometimes searches can be speeded up by changing the representation. Today's programming languages describe programs in terms of what the computer has to do to carry them out. If the algorithms were written in a language that defined algorithms in terms of what they accomplished, rather than the steps in terms of which they accomplished it, it might relatively easy to compute the next program to consider from the description of the current program and the datum that caused its rejection. If, for example, a program to control a golf-playing robot has a numerical parameter that determines how hard the ball is to be hit and a practice shot falls short, it is relatively easy to decide what to change in its program to take the new evidence—the shot that fell short—into account. But, in our case, changing numerical parameters only will probably narrow the domain too much.

Once we develop ways to increase the efficiency of limiting-computable learners, we might try putting them to practical use. Among the possible uses to which implementations of such learners might be put are the following:

Programs that generate programs from examples: Today, we program computers by telling them, step by step what to do. That is not how we usually "program" people. Limiting computable learners—if they could be made efficient—might make it possible for us to program computers by showing them examples of what we want them to do and letting them develop their own programs from those examples.

Programs that adapt to their users: A program that develops programs from examples can be used to develop models of that program's users from the way they behave to that program. For example, a teaching program might use the behavior of its students to try to develop a theory of what was going on in the minds of those students. And then it might use those theories to try to adapt its behavior to their needs.

Programs that generate theories from evidence: Predictors with limiting computable learners model Popper's [18] account of the scientific method and they might, therefore, be used to automate certain kinds of theory generation. Popper argued that, when

scientists try to come up with general theories from specific evidence, they have disproof procedure but not a proof procedure. In other words, they have a way to tell whether any particular swan they see disproves the theory that "All swans are white". But they have no way to prove that every swan they see will be white. So they effectively (by Popper's account) compute in the limit—holding on to a theory as long as it is not disproved by the evidence. By using simple enumeration methods, computers might be able to simulate the scientific method (if Popper was right about how it works). Popper's method is not the only way that theories can be generated from evidence and approaches based on other philosophical accounts of the scientific method, such as that of Kuhn [16], might also be worth trying to automate.

Programs that generate programs from hints and incomplete or vague instructions: Applying a hint goes beyond the information given and might, therefore, also profit from algorithms that compute in the limit. And the "explicit" instructions given to humans (in such natural languages as English) are not totally explicit (in contrast to computer programs) so that, to follow them, a system also has to go beyond the information they give. As a result, they might profit from using super-recursive algorithms.

When we study human intelligence we often try to compare the intelligence of systems. We seem less interested in doing this with machines but we still do it from time to time. Thus people sometimes say that one chess-playing program is more intelligent than another because it plays better chess. But that compares the using components.

When we do that, we might want to say that one system, A, is $more_1$*intelligent* than another, B, if its using component is larger or faster or more powerful in some other way than B's. It is this sense of comparative intelligence that makes a world-championship chess program seem intelligent to some.

Alternatively we might focus on the learning component and say that A is $more_2$ *intelligent* than B if its scope properly includes that of B or if it converges to the right programs more quickly.

One way to extend the scope of a learner is to allow it to use more than one algorithm at a time. It might for example, run several sub-learners, each using a different strategy to acquire programs. It could then (somehow) choose the best result that any sub-system came up with.

Gardner [8] has suggested that human intelligence may come in different "flavors" or "intelligences", each suited to a different domain. Given a particular problem, the mind might set several of these intelligences (or learners) to work, leaving it to some sort of supervisory system (which could be thought of as a model of what we call *consciousness*, if you like) to decide which of the results of its learners to convey to its user. A machine version of this idea might be worth exploring.

6. Conclusion

People tend to feel that intelligence is a good thing, even if they are unable to say exactly what it is. But it presence in a machine might not be an unmitigated blessing. If, as I have suggested, you cannot have real intelligence without giving up

the announcement condition, there are purposes for which we might not want machines to be intelligent.

Today we tend to use computers to do what we want them to do in the way we want them to do it. If we were to let them behave intelligently, we would (if the account I have given of intelligence here is correct) have to let them develop their own way of doing what we ask them to do and we would have to let them to come up with results that they could, later, change.

There are many applications for which we would want to avoid such independence and flexibility for very much the same reason that we might want to avoid them in human office clerks. But there are some applications for which intelligence—in the sense in which I have tried to define it here—might be a good thing. [2]

References

[1] D. Angluin, C.H. Smith, Inductive inference: theory and methods, Comput. Surv. 3 (1983) 237–269.
[2] M.S. Burgin, Inductive Turing machines, Sov. Math. Dokl. 270 (1983) 1289–1293.
[3] M.S. Burgin, Super-recursive algorithms as a tool for high performance computing, Proc. High Performance of Computer Symposium, San Diego, 1999, pp. 224–228.
[4] M.S. Burgin, How we know what technology can do, Comm. ACM 44 (2001) 82–88.
[5] N. Chomsky, Aspects of the Theory of Syntax, MIT Press, Cambridge, 1965.
[6] B.J. Copeland, D. Proudfoot, Alan Turing's forgotten ideas in computer science, Sci. Amer. 280 (1999) 76–81.
[7] P. Cotogno, Hypercomputation and the physical Church–Turing thesis, British J. Philos. Sci. 54 (2003) 181–223.
[8] H. Gardner, Frames of Mind: The Theory of Multiple Intelligences, Basic Books, New York, 1983.
[9] K. Gödel, Die Vollständikeit der Axiome des logischen Funktionenkalküls, Monatsh Math. Phys. 37 (1930) 349–360.
[10] E.M. Gold, Limiting recursion, J. Symbolic Logic 30 (1965) 28–48.
[11] T.D. Kieu, Quantum algorithm for the Hilbert's tenth problem, Contemp. Phys. 44 (2003) 51–71.
[12] S.C. Kleene, Recursive predicates and quantifiers, Trans. Amer. Math. Soc. 53 (1943) 41–73.
[13] S.C. Kleene, Representation of events in nerve nets and finite automata, in: C.E. Shannon, J. McCarthy (Eds.), Automata Studies, Annals of Mathematics Studies, Vol. 34, Princeton University Press, Princeton, NJ, 1956, pp. 3–41.
[14] P. Kugel, Induction pure and simple, Inform. and Control 35 (1977) 276–336.
[15] P. Kugel, Computing machines can't be intelligent (...and Turing said so), Minds Mach. 12 (4) (2002) 563–579.
[16] T. Kuhn, The Structure of Scientific Revolutions, University of Chicago Press, Chicago, 1962.
[17] Yu.V. Matiyasevich, Solution of the tenth problem of Hilbert, Mat. Lapok 21 (1970) 83–87.
[18] K. Popper, The Logic of Scientific Discovery, (translation of Logik der Forschung), Hutchinson, London, 1959.
[19] H. Putnam, Trial and error predicates and the solution of a problem of Mostowski, J. Symbolic Logic 20 (1965) 49–57.
[20] R. Saxe, N. Kanwisher, People thinking about thinking people. The role of the temporo-partietal junction in "theory of mind", Neuroimage 19 (2003) 1835–1842.
[21] H. Siegelmann, Computation beyond the Turing limit, Science 268 (1995) 545–548.
[22] A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, Proc. London Math. Soc. Ser. 2 42 (1936) 232–265.

[2] I want to thank Mark Burgin and the referee for helpful suggestions. They should, of course, be granted the customary absolutions.

[23] A.M. Turing, Systems of logic based on ordinals, Proc. London Math. Soc. Ser. 2 45 (1939) 161–228.
[24] A.M. Turing, Computing machinery and intelligence, Mind 59 (N.S. 236) (1950) 433–460.
[25] A.M. Turing, Proposals for the development in the Mathematics Division of an Automatic Computing Engine (ACE), in: B.E. Carpenter, R.N. Doran (Eds.), A.M. Turing's ACE Report and Other Papers, MIT Press, Cambridge, MA, 1986.
[26] A.M. Turing, Lecture to The London Mathematical Society on 20 February 1947, in: B.E. Carpenter, R.N. Doran (Eds.), A.M. Turing's ACE Report and Other Papers, MIT Press, Cambridge, MA, 1986.