A SYSTEM FOR INCREMENTAL LEARNING BASED ON ALGORITHMIC PROBABILITY

Ray J. Solomonoff *
Computer Learning Research Center
Royal Holloway, University of London
Mailing Address: P.O.B. 400404, Cambridge, Ma. 02140, U.S.A.
Email: rjsolo@ieee.org

Dec. 1989

Abstract

We have employed Algorithmic Probability Theory to construct a system for machine learning of great power and generality. The principal thrust of present research is the design of sequences of problems to train this system.

Current programs for machine learning are limited in the kinds of concepts accessible to them, the kinds of problems they can learn to solve, and in the efficiency with which they learn — both in computation time needed and/or in amount of data needed for learning.

Algorithmic Probability Theory provides a general model of the learning process that enables us to understand and surpass many of these limitations.

Starting with a machine containing a small set of concepts, we use a carefully designed sequence of problems of increasing difficulty to bring the machine to a high level of problem solving skill.

The use of training sequences of problems for machine knowledge acquisition promises to yield Expert Systems that will be easier to train and free of the brittleness that characterizes the narrow specialization of present day systems of this sort.

It is also expected that the present research will give needed insight in the design of training sequences for human learning.

*Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, Dec. 1989

Introduction

We will describe a system for machine learning that uses algorithmic probability to go beyond many of the limitations of current learning systems.

The machine starts out, like a human infant, with a set of primitive concepts. We then give it a simple problem, which it solves, using these primitive concepts. In solving the first problem, it acquires new concepts that help it solve the more difficult second problem, and so on. A suitable training sequence of problems of increasing difficulty brings the machine to a high level of problem solving skill.

The principal activity of the present research is the design of training sequences of this kind.

Current systems for machine learning have several critical limitations. We will describe these limitations and how our system overcomes them.

For each system other than our own, there are certain classes of concepts that the system can never discover, no matter how long it searches. This limitation may occur because the system has an "incomplete" set of concepts (not "universal" in the sense of "universal Turing machine"), and/or, because the search algorithm is inadequate. Our system, however, is given a complete set of concepts at an early point in its training. The use of Levin's search algorithm (Lev 73, Sol 84) then guarantees that any describable concept will eventually be discovered by the system.

Another kind of limitation in current systems is the relatively narrow range of problems they can solve. The algorithm we use is able to solve inversion problems (such as the P and NP problems of computational complexity theory) as well as time limited optimization problems (See Section 2 for definitions and discussion of these problem types). Algorithmic probability theory makes it possible to express problems of clustering, pattern discovery, design of scientific experiments, discovery of scientific laws to fit data, etc., as time limited optimization problems. Although a large fraction of problems occurring in science and mathematics are either inversion or time limited optimization problems, many of them are not solvable, or even addressable by other learning systems.

It should be noted, however, that one type of generalized time limited optimization problem does not appear to be solvable by Levin's algorithm or any other that we know of. Part of our research is to find solutions or approximate solutions to this kind of problem.

The efficiency of a system for machine learning is limited in two ways. The first is measured by the computational complexity of its solutions to problems. How much time and/or memory is required? The second is informational efficiency. How much training is needed for the system to learn? How many problems or examples are required?

In the particular training environment we have created, the use of Levin's search algorithm appears to be extremely efficient in both respects. There are heuristic arguments for its being within a factor of 4 of optimum, and the determination of the extent to which this is true is a topic of continued research.

The following sections discuss different aspects of our system:

1. What is algorithmic probability? Why is it needed? Some successful applications.

2. What kinds of problems can the system solve? Inversion problems and time limited optimization problems are defined. They cover a large fraction of problems in science and engineering.

3. What are "concepts" and how do we select a set of primitive concepts to start out with?

4. How does the machine use its concepts to solve problems?

5. After it solves a problem, how does the machine use the solution to define new concepts and update the parameters of old ones?

6. How do we write training sequences of problems that bring the machine from its initial ability to solve only the simplest of problems to a state in which it can solve very difficult ones?

7. How does this model relate to other work on machine learning?

8. What is the present state of development of the system? Near term goals, more distant goals. Time scales for these goals.

1 What is Algorithmic Probability? Why is it Needed? — Some Successful Applications.

The need to revise classical concepts of probability is strongly suggested by analysis of human problem solving. When working on a difficult problem, a person must make choices of possible courses of action. If the problem is a familiar one, the choices will all be easy. If it is not familiar, there can be much uncertainty in each choice, but choices must somehow be made. One basis for choice might be the probability that each choice leads to a quick solution. This enables us to use a simple search procedure to find the solution rapidly.

The usual method of calculating probability is by taking the ratio of the number of favorable choices to the total number of choices in the past. If the decision to use integration by parts in an integration problem has been successful in the past 43% of the time, then its present probability of success is about .43. One trouble with this method is that it has very poor accuracy if we only have one or two cases in the past, and it is undefined if the case has never occurred before. Unfortunately it is just these situations that occur most often in problem solving.

By creating an entirely new definition of probability, we have been able to resolve this difficulty as well as many others that have plagued classical concepts of probability.

The earliest description of algorithmic probability was in a formal theory of inductive inference (Sol 60, 64a, 64b). All induction problems are equivalent to the problem of extrapolating a long sequence of symbols. Formally we can do this extrapolation by Bayes' Theorem, if we are able to assign an a priori probability to any conceivable string of symbols, x.

Algorithmic probability defines the a priori probability of a string of symbols, x, as the probability that x will be produced as the output of a reference universal Turing machine having random input. One of the fundamental properties of algorithmic probability is its relative insensitivity to choice of reference machine. [1]

Though this definition seems distant from the usual, frequency based definition of probability, it has been proved that in all cases in which probability is defined by frequency, algorithmic probability converges rapidly to the same probability values.

Perhaps of most importance: If there is any describable regularity in a body of data, algorithmic probability is guaranteed to eventually discover that regularity, using a relatively small sample of the data (Sol 78). It is the only definition of probability known to have this property.

If $l(x)$ is the number of bits in the shortest program that can generate string x, then $2^{-l(x)}$ is a useful approximation to the algorithmic probability of x. The commonest way to approximate it is to find short programs for x. For any x, we can always find an "identity program" that is about as long as x itself, but if x has any pattern in its bits, we can exploit that pattern to devise a program for x that is shorter than x. A sequence of 1000 one's, for example, has a very simple pattern, and it can be generated by a very short computer program. If the shortest program for x is about the same length as x, then x has no patterns to exploit — it is completely random.

By using code compression as a criterion, algorithmic probability gives a truly objective way to detect patterns in data, and becomes a completely adequate basis for the mechanization of pattern discovery.

From considerations of this kind, Kolmogorov (Kol 65) and Martin–Löf (Mar 66) were able to develop the first adequate definition of randomness. Chaitin (Cha 74) showed how algorithmic complexity (which is the negative logarithm of algorithmic probability) could give a very clear demonstration of Gödel's theorems in mathematical logic, and suggested a much simpler proof for Turing's "Halting Problem".

Some of our earliest innovations in inductive inference theory were the use of formal languages for induction (Sol 58, 60b), and the invention of probabilistic languages for more precise prediction (Sol 59). Algorithmic probability was able to refine these ideas by giving an optimum "goodness of fit criterion" for the fitting of probabilistic grammars to data (Sol 62, 64b, 75). This was developed further by Horning (Hor 71).

Another application of algorithmic probability was to Winston's problem of learning structural descriptions from examples (Win 75) — obtaining an

[1]For a more exact definition of algorithmic probability, see Sol 78, or, more briefly, Sol 86, Pp. 476–478.

algorithm that was far better than Winston's in both speed of execution and efficiency in use of data (Sol 75, Sol 86, p. 490).

Algorithmic probability has given a unified approach to problems of universal coding, information, prediction and estimation (Ris 84). The applicability of the maximum entropy principle in statistics was greatly expanded when it was shown to be a special case of algorithmic probability (Fed 86).

An important non–probabilistic application has been its use to determine bounds on computational complexity (Pau 81).

For some time after its discovery, the use of algorithmic probability was plagued by questions of its meaningfulness and accuracy. These were finally resolved by a proof of the existence of the defined probability as a limit and a proof of its accuracy and efficiency in use of data (Sol 78). Cover (Cov 74) showed that if it was used as the basis of a gambling scheme, its yield would be extremely high.

At first, the apparent difficulty in approximating algorithmic probability limited its practical application. A real breakthrough was Levin's search procedure (Lev 73, Sol 84), using algorithmic probability to obtain solutions to a very broad class of mathematical problems within a constant factor of minimum time (These are the P and NP problems of computational complexity theory, and time limited optimization problems. They are discussed in Section 2.). Later, heuristic arguments were advanced that for critical problems in machine learning, the "constant factor" was less than 4 — giving what looks like a very practical, near optimum system for machine learning (Sol 86, pp. 475 and 482).

2 The Kinds of Problems Solved.

The first kind of problem the system solves is called an "Inversion Problem". We are given a string of symbols, s, and a program or function or computing machine, M, that quickly operates on strings to produce strings. The problem is to find, in as little time as possible, a string x, such that M operating on x, produces s — i.e. $M(x) = s$. For example, find the number x (which is a string), such that $x^2 + sin(x) = 25$. The P and NP problems of computational complexity theory are all inversion problems of this sort. Solving algebraic equations, symbolic integration and proving mathematical theorems are all examples of inversion problems.

Another kind of problem the system solves is called a "Time Limited Optimization Problem". We are given a program or function or machine M, that maps strings of symbols into real numbers. We are given a time limit, T. The problem is to find within time T, a string of symbols, x, such that $M(x)$ is as large as possible. Many engineering problems are of this sort — for example designing an automobile in 6 months satisfying certain specifications, having minimum cost. Having a limited amount of time to devise the best possible theory to fit our empirical data is also a problem of this type. Perhaps most important: The problem of improving our machine's updating algorithm (See Section 5) is a time limited optimization problem, so the machine can work on the problem of improving itself.

A large fraction of all problems in science and engineering can be expressed as either inversion or time limited optimization problems.

In the present paper we will describe the solution of inversion problems only. The methods used to solve normal time limited optimization problems are, however, very similar.

It should be noted that there is one type of generalized time limited optimization problem that is not solved very well by our system or by any other algorithm we know of. In this kind of problem, we have to find an optimum value for string x, but there is no sharp time limit. Instead, we have to maximize $G(x) \cdot F(t)$. Here t is the time we use to find x, and F is a known, decreasing function of t .

One direction of continued research attempts to obtain better approximate solutions to this type of problem.

3 What Concepts Are. How to Obtain a Primitive Set of Concepts.

We are using the term "concept" in a fairly general sense. A concept is a computer program or a fragment of a program. It can be a single computer instruction or a sequence of instructions. It can be written in machine language or in any other computer language.

Production systems such as are used in "Expert Systems" can have all of the power of computer instruction sets. In systems of this sort, a concept is a production rule or combination of such rules.

A necessary condition for the selection of a set of primitive concepts is that the set be "complete". This means that any solution of a mathematical problem is capable of being expressed as a sequence (or other suitable combination) of those primitive concepts.

We will begin the training by giving the machine a small, incomplete set of primitive concepts, that are able to express the solutions to the first set of simple problems. When the machine learns to use these concepts effectively we give it more difficult problems and, if necessary, additional primitive concepts needed to solve them. We continue to give new problems and new primitives until the machine has a complete set of concepts. After this point is reached, new problems can be given, but no new primitives are needed.

It is not difficult to devise sets of concepts that are complete. Almost all digital computers and languages for programming them have complete sets of instructions. Usually only a small subset of these instructions is needed to form a complete set.

Except for the need for completeness, the choice of primitive concepts is not critical. If a poor set of primitives is chosen, it will require a longer training sequence for the development of a set of concepts able to solve significant problems. Once this development occurs, however, the behavior of the system is fairly independent of the initial choice of primitives.

4 How Inversion Problems are Solved.

To solve an inversion problem, we construct a sequence of trial computer programs. Each of these trial programs is a sequence of primitive concepts. At first, each program takes only the description of the problem as its input and gives a candidate solution to the problem as its output. At later stages of training, input to trial programs may include information on the operation of earlier trial programs.

Suppose we are given the function $M(\,)$ and the string, s, and the problem is to find a string x such that $M(x) = s$. We take A_1 , the first in our sequence of trial programs, and have it operate on $M(\,)$ and s to produce x_1 , a candidate solution to our problem. We then test x_1 , to see if $M(x_1) = s$. If it does, we have a solution. If not, we try the next program A_2, and so on, until we find a solution.

In solving the early problems of the training sequence, the system has only a small number of primitive concepts, and the solutions to its problems require the combination of only a few of them, so the effective search space is quite small. In the preliminary training sequence for learning algebraic notation (Sol 86, Pp. 485–486) the first problems took about 343 ($= 7^3$) trials to solve — the next group took about 7 trials each and the final set were solved on the first trial.

In this initial learning phase the ordering of trials is completely independent of the nature of the problem to be solved. This "Blindness" of search is similar to that of organic evolution — but our search and update algorithms are far more efficient than the mutation and pair–wise recombination used in organic evolution.[2]

The most efficient search procedure — one whose expected time to solution is minimal — tests trial programs (i.e. strings of concepts) in order of increasing t_i/p_i. Here p_i is the probability of success of the i^{th} trial string of concepts and t_i is the time needed to generate and test that trial.

[2]In most problem domains, the exponential complexity of blind search seems to make it impractical for all but the simplest of problems. Heuristic search was devised to cope with this "exponential explosion". However, it is not difficult to prove that any conceivable heuristic search algorithm can be simulated by a blind search in a space of sufficiently powerful concepts. Our system derives much of its power from this theorem. It must be noted that in such a system a candidate program must be permitted to have more arguments than simply the present problem description. Its arguments may include any information obtained as the result of, and during the execution of, previous candidate programs.

The computation of probability values for a string of concepts, normally a very difficult task, is made tractable by application of algorithmic probability theory. We initially assign equal probabilities to each of the primitive concepts. Then a program which is a string of such concepts has a probability equal to the product of the probabilities of its components. If we have 10 primitives then each will have probability .1 and a string of 5 of them will have probability 10^{-5}.

These initial approximations to p_i are very rough. However, after the machine has solved several problems, new concepts are defined, the probabilities of old concepts are modified and some of the concepts are linked by conditional probabilities. These operations occur in the "Updating" phase of the machine's operation and are designed to optimize each p_i as an approximation to the probability of success of the i^{th} trial.

Returning to our "most efficient search procedure" using the t_i/p_i ordering, we find that while we can approximate p_i we can never know t_i before the i^{th} trial, so we can't make our trials in exact t_i/p_i order. However it is possible to obtain something like t_i/p_i order in the following manner:

First we select a small time limit, T, and we do an exhaustive test of all strings, spending at most, a time, p_iT on the i^{th} string.[3]

If we find a solution we stop. If not, we double T and go through the exhaustive testing again. The process of doubling and testing continues until a solution is found. The entire process is approximately equivalent to testing the strings in order of increasing t_i/p_i.

This search algorithm was first used by L. Levin (Lev 73) to solve a broad class of mathematical problems. One of the important properties of this algorithm is that it is easy to estimate the total search time needed to discover a particular known solution to a problem. If p_j is the probability assigned to a particular program, A_j , that solves a problem and it takes time t_j to generate and test that program, then this entire search procedure will take a time less than $2 \cdot t_j/p_j$ to discover A_j . We call t_j/p_j the "Conceptual Jump Size" (CJS) of A_j. It tells us if the machine is practically able to find a particular solution to a problem at a particular state of its development. CJS is a critical parameter in the design of training sequences and in the overall operation of the system.

After the system has solved a problem, we usually allow it to continue searching for other ways to solve that same problem, in the hope that it will find a better solution than the first one found. "Better" in this case means the time to generate and test that solution is smaller.

[3]If we only test strings whose p_iT is greater than the machine's instruction time, there will be a finite number of strings to test and the total testing time will be less than T.

5 How Updating is Done

After several problems have been solved, the machine "updates" itself by creating new concepts along with their associated probabilities and by modifying the probabilities of old concepts. This is done by taking the "best" successful program for each of the problems that has been solved. This will be the program that takes least time. The collection of "best" programs is called "the corpus" — our body of data. We want to extrapolate this corpus to obtain new programs that have high probability of giving good solutions to new problems.

We first assign a different symbol to each of the concepts, so that the programs which were strings of concepts, become strings of symbols. A very simple way to extrapolate such a corpus is to count the relative frequency with which each type of symbol occurs in the corpus. This set of frequencies gives a probability distribution on all possible strings of symbols. The probability of a string of symbols is equal to the product of the relative frequencies of its component symbols.

While this is sometimes an adequate method of extrapolating a corpus of strings, we can do much better, in the following manner: Usually there will be pairs of symbols, for example, AB, such that the frequency of the pair, AB, is significantly different from the frequency of A times the frequency of B. This indicates that A and B are somehow related — that the probability of B is somehow conditional upon whether or not it follows A. We go through the corpus and examine all possible pairs of symbols. We pick the pair, say CD, for which the frequency disparity is most significant of all those we've examined.[4]

We then define a new symbol W, which represents CD and we rewrite our entire corpus substituting W wherever the pair CD occurs. Defining the symbol W makes it possible to exploit the conditional probability of D following C in assigning probabilities to programs.

The new corpus is examined for "most significantly deviant pair frequency" as before, and we define a new symbol, X, to substitute for this pair. We repeat the operations of defining a new symbol and rewriting the corpus until we can no longer find pairs that are of "significantly deviant frequency".

The combining of a pair of symbols is equivalent to combining a pair of concepts to obtain a new, more complex concept. By iterating this procedure, we are able to define complex new concepts that combine a large number of old concepts. For example, we can define the concept $ABCD$ by first defining AB, then CD, then the combination of these two newly defined concepts to form $ABCD$.

The set of symbols in the final corpus and their frequencies of occurrence is used to obtain a probability distribution on all possible strings of symbols, and from this, a probability distribution on the computer programs that they represent.

[4]A quantitative measure of the significance of this kind of frequency deviation and a more detailed description of the updating system is given in Sol 64b, pp. 233–240.

This completes the "updating process". Our machine is now ready for the next set of problems.

The method we have used to extrapolate the corpus of "best" programs is equivalent to finding a probabilistic grammar that best fits that corpus. The grammar described is perhaps one of the simplest probabilistic grammars that can create new concepts by combining old ones. There are many other kinds of grammars that can be used. A probabilistic context free grammar (Sol 64b, Hor 71) would be able to find relationships in our corpus that can't be found using simpler grammars. It is also possible to extrapolate the corpus of programs using any data compression algorithm (Sto 88). Algorithmic probability readily transforms such algorithms into probabilistic languages.

The discovery and implementation of a better updating algorithm can be formulated as a time limited optimization problem. Thus with a suitable training sequence, it is possible to have the machine work on the problem of self–improvement.

6 How Training Sequences are Written.

The main motivation for writing training sequences, is that without incremental learning of this kind, it requires too much search time to learn to solve difficult problems. The task of designing such sequences is very similar to writing "Top Down" computer programs or writing lesson plans for human students.

To design a training sequence culminating in a particular solution to a problem, we first examine that solution and break it down into a small number of concepts or modules that can be combined to produce the solution. Each of these concepts should be applicable to the solution of other useful problems as well. We then examine each of the concepts and break each of them down into useful subconcepts as before. This breaking down into smaller subconcepts continues until we reach the primitive concepts of our machine.

By linking each concept to its set of component subconcepts, we form a tree structure which we call a "concept net", as in Figure 1. (For a discussion of the relationship of concept nets to neural nets, see Section 7).

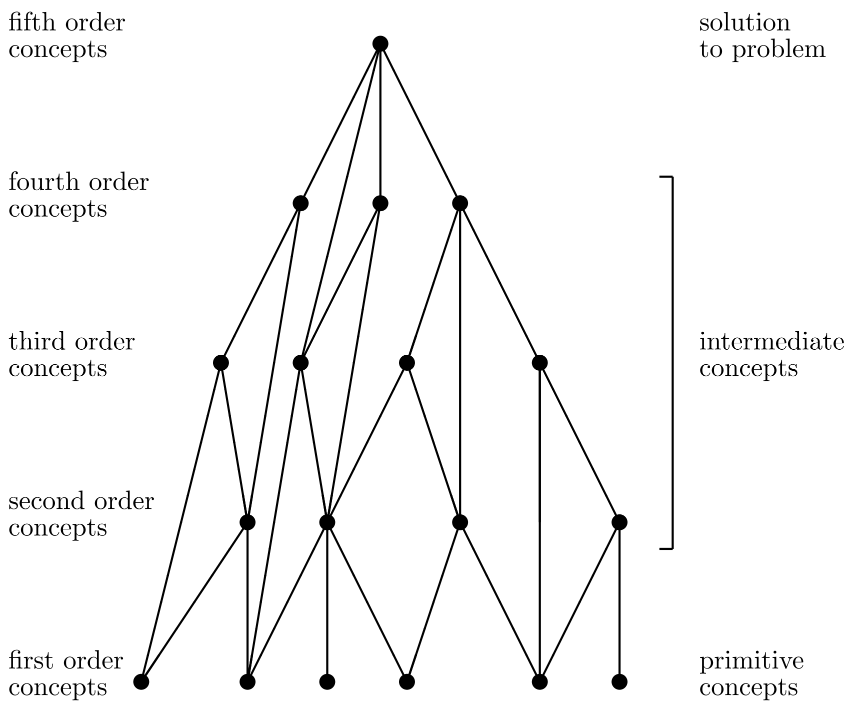

Figure 1. A SIMPLE CONCEPT NET

Let us call the primitive concepts “first order concepts”, and the concepts one level above them “second order concepts”. An n^{th} order concept is then defined to be a combination of concepts in the net that are of order less than n. The first set of problems in our training sequence will be problems that have for their solutions a combination of a few concepts. These solutions will themselves be second order concepts.

We define an n^{th} order problem to be one that has an n^{th} order concept as solution. For our training sequence, we next devise an adequate set of 3rd order problems. An “adequate” set of n^{th} order problems is one in which each $(n-1)^{th}$ order concept has been included several times in the set of solutions.

We then devise an adequate set of 4^{th} order problems, and continue devising adequate sets of problems of higher and higher order, until we arrive at the final top problem.

This set of problems in the order in which they were devised will constitute our training sequence. In the concept net shown, the entire training sequence would consist of second order problems followed by third order problems, then fourth order problems, and finally the top fifth order problem.

In the early stages of its training, the machine will be given only problems of small CJS (Conceptual Jump Size) - problems that are easy for it to solve. Such problems will also be easy for the trainer to analyze. He will be able to predict accurately the detailed response of the machine. During these stages of training, the machine's behavior will be similar to that of most "Expert Systems" — its concepts and solutions to problems will be limited to those anticipated by the trainer.

At a more advanced stage, we want the machine to be able to break out of this limitation. This can be done in several ways. One is to allow much time for "oversolving" problems — i.e. after the machine has solved a problem in a manner expected by the trainer, it is given time to try to find better solutions. Another way is to give problems for which the trainer knows only solutions of large (but practicable) CJS. In searching this large CJS space, the machine is more likely to find solutions the trainer hadn't thought of.

Each of these techniques is time consuming and expensive — but a necessary expense if we want the machine to transcend the conceptual limitations of its trainer.

Often in constructing training sequences we find that problems we would like to include have excessively high CJS — beyond the bounds of practical search times. Usually this is because the probabilities of some of the concepts in the solutions to these problems are too small.

These low probabilities are an indication of inadequate training for the concepts involved. This can be corrected either by giving the machine more problems whose solutions use those concepts, or by breaking the concepts down into subconcepts and giving an adequate set of problems involving those subconcepts.

The most difficult part of training sequence design is the construction of the concept net. It will be noted that the process of breaking down concepts into more easily learned subconcepts is a necessary skill of every good teacher. While it often requires insight and imagination, it is certainly possible to do it effectively.

7 How This Model Relates to the Work of Others.

There have been three superficially different approaches to the mechanization of cognitive processes:

Heuristic Programming (New 87, Len 82): in which new definitions are built upon older definitions that have been found to be useful.

Multilayer Neural Nets (And 88, Rum 86): in which neurons of each layer are excited by previously learned patterns of excitation of earlier layers.

The Society of Mind (Min 86): in which each new agent builds on the abilities that older agents have developed. Our research has borrowed heavily from all of them.

Each of these approaches has its own ways of combining successful abstractions to form promising new abstractions, and of using them for learning and problem solving. Algorithmic probability gives a unified technique for analyzing systems of this kind and overcoming many of their limitations. It has been used to criticize and suggest improvements in the inductive systems of Winston (Win 75) and Lenat (Len 82). Our general model of learning continues to be invaluable in the understanding, analysis and optimization of each of the aforementioned approaches to the mechanization of cognitive processes.

Of all work in machine learning, our system is most similar to Lenat's program, AM, (Len 82) but differs from it in important ways. While AM may have a complete set of primitives, its search algorithm does not exploit that completeness. There are many kinds of simple concepts that AM can never discover, no matter how long it searches. Our own system, however, by using Levin's search algorithm and a complete set of primitives, guarantees that there are no describable concepts that it will not eventually discover.

Newell, Rosenbloom and Laird's SOAR (New 87), as well as our own system is meant to be the framework of a general theory of cognition. Newell et al, however, as well as Minsky and Lenat, model their systems on their observations of human problem solving and concept discovery. Our model of cognition attempts to optimize the learning capability of our system, independently of just how people seem to work.

SOAR suffers from a common debility of systems using monotonic reasoning — the inability to get rid of a once useful concept that has become harmful in a new problem domain. In our own system, each concept has a conditional probability associated with each area of application. If a concept is found to be of little value for a certain problem domain, its conditional probability for that domain (i.e. the probability of its being applied there) decreases.

Another related area is Valiant's research on "Learnability of Concepts" (Val 88), but he considers only certain subsets of the class of concepts expressible as Boolean polynomials. In contrast, we consider the most general type of concept possible — the set of all concepts that are expressible as computer programs.

His main interest is whether a particular concept is learnable in polynomial time or not. This is certainly of theoretical interest, but knowing that a computation is polynomial doesn't tell much about its practical computability. We use CJS (conceptual jump size) to get good estimates of the time our machine needs to solve problems. It is easy to compute and gives a much better estimate than simply knowing if the computation time is polynomial.

Perhaps of most importance: in solving any problem, our system is able to use any information learned in solving previous problems, so the more problems it solves, the more skilled it becomes. In contrast, Valiant obtains for each class of concepts, a special learning algorithm, independent of what classes of problems were solved in the past. The inability of these algorithms to profit from many kinds of previous experience in problem solving seriously limits their value in studying machine or human learning.

The neural nets used for cognitive modeling (And 88, Rum 86) correspond closely to our "concept nets." The concepts themselves correspond to nodes in the neural net, and the weighted connections in the neural net correspond to conditional probabilities relating concepts. They are strengthened or weakened with training, as are the corresponding connections in neural nets.

There are, however, important differences: one is that the updating algorithms that modify the conditional probabilities in concept nets, are quite different from those used in current neural net research. The conditional probability linking two concepts can depend on what occurs anywhere else in the net any time in the past. The learning algorithms for neural nets only allow interactions that are very local in space and time.

Another difference is that in a concept net, once a concept is defined, it becomes immediately accessible at no extra cost, to all of the rest of the net, to be used in problem solving or for defining new concepts.

It is easy to realize this cost–free sharing of concepts in computers that allow subroutines to be shared by different parts of a program. In a neural net, however, if several parts of the net need to use the same operator at the same time, there will be a serious slowdown of network operation. One way to deal with this is to make duplicates of important operators in various parts of the net. Current neural net models don't have ways to implement this duplication.

A third difference is that neural nets have great difficulty learning concepts that require more than a few layers of "hidden units", in contrast to our system, which has no particular difficulty in learning concepts of great logical depth.

Nevertheless, network technology has great potential for fast, relatively inexpensive computation, and we try to keep aware of developments in this field that might enable us to adapt neural nets for more efficient hardware realization of our learning model.

8 Present State of Development of the System? Near Term Goals, More Distant Goals Time Scales for These Goals

We have a general theoretical framework for machine learning based on algorithmic probability and computational complexity theory. It overcomes many limitations of current systems for machine learning and gives what appears to be an extremely effective model for practical induction in a suitable training environment. In its present form we have been able to use it to criticize other systems for machine intelligence (such as those employing neural nets, and the systems of Winston (Win 75) and Lenat (Len 82)) and often suggest ways to deal with these criticisms.

Three things are needed to implement this system as a computer program:

First, the programming of Levin's search procedure for solving problems. We have written preliminary programs for inversion problems and for optimization problems (Sol 84). There appear to be no serious difficulties in this part of the system.

Second, the programming of the updating procedure that defines new concepts and modifies the parameters of old ones. Although this program has been written (Sol 64b, pp. 233–240), we have not yet tested it in our present system.

Third, the writing and testing of training sequences for the system. This is the main course of work in this research. These training sequences are of three types:

The first type of training sequence is for problems of small Conceptual Jump Size (CJS) — problems that are easy for the machine to solve, and simple enough for the trainer to reliably anticipate the machine's solutions of them.

We have written preliminary training sequences for learning algebraic notation (Sol 86). We will continue with sequences for learning to solve equations and sets of simultaneous equations. Parallel with this we will be writing training sequences for theorem proving in formal logic, symbolic integration, parsing sentences in formal languages and other problems that have been carefully formalized and investigated by researchers in A.I. These problems are all "adjustable" in that we can make them as easy or as difficult as we want them to be.

The second type are training sequences for problems of larger CJS. These are more difficult problems in which the machine will often discover concepts and solutions to problems unanticipated by the trainer. The problems will be otherwise similar to those of the first type.

The third type are "bootstraps training sequences" that improve the machine's general problem solving ability and/or make it easier for us to train the machine. One important sequence of this kind will train the machine to work on an unordered batch of problems — deciding itself which are the easiest, and solving them first. This makes it much easier to train the system.

Another will train the system to work on the problem of improving its updating algorithm. The kind of training needed involves more mathematics and work on various kinds of optimization problems — ultimately problems of improving computer programs.

Perhaps the most important kind of training sequence is one that teaches the system to understand English text. By "understand" we mean able to correctly answer questions (in English) about the text. This understanding need not be at all complete, but should be good enough so that ordinary English texts can be a useful source of training for the system.

This training sequence will involve formal languages of increasing complexity. The first examples of English text will cover a field that the system will already be familiar with — so that it will only have to learn the relationship of the syntax to facts it already knows.

There has been much work on language learning by machines as well as by humans (Ber 85). The formalization of learning implied by algorithmic probability usually makes it possible to translate this work into a form that will be useful for training our system.

Our method contrasts sharply with that of Lenat, who is painstakingly programming "common sense" information into his system, bit by bit (Len 86). Initially, Lenat had considered the possibility of having his machine learn English and have it discover the "common sense" concepts that were needed to make the text hold together. At that time, however, understanding of learning itself was too rudimentary for this to be a reasonable approach.

It is difficult to overestimate the importance of text comprehension. After the machine has learned to understand text to a fair degree, we can give it a very large amount to read with relatively little external guidance — just as an adequately trained human student. We expect that as the machine reads more and more, it will understand a larger and larger fraction of what it reads, partly by inventing concepts to fill in the "common sense knowledge" not stated in the text, just as a human student does.

Acknowledgements

Critical discussions with Murray Denofsky, Brewster Kahle, Jack Kotik, Grace Morton, Richard Stallman and Carlos Zapata have been invaluable in enhancing both the readability and accuracy of this paper. Many of the important ideas stem from discussions over many years with Marvin Minsky.

This work was supported in part by the United States Air Force Office of Scientific Research under Contracts AF–19(628)5975, AF–49(638)–376, and Grant AF–AFOSR 62–377; in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contracts N00014–70–A–0362–0003 and N00014–70– A–0362–0005 and in part by the Public Health Service under NIH Grant GM 11021–01.

References

(And 88) Anderson, J., and Rosenfeld, C., "Neurocomputing", Cambridge MIT Press, 1988.

(Ber 85) Berwick, R.C., "The Acquisition of Syntactic Knowledge", Cambridge MIT Press, 1985.

(Cha 74) Chaitin, G.J., "Randomness and Mathematical Proof," Scientific American, Vol 232, No. 5, pp. 47–52, May 1975.

(Cov 74) Cover, T.M. “Universal Gambling Schemes and the Complexity Measures of Kolmogorov and Chaitin,” Rep. 12, Statistics Dept., Stanford Univ., Stanford, Ca, 1974.

(Fed 86) Feder, M., “Maximum Entropy as a Special Case of the Minimum Description Length Criterion”, IEEE Transactions on Information Theory, pp. 847–849, Nov. 1988.

(Hor 71) Horning, J. “A Procedure for Grammatical Inference,” Proceedings of the IFIP Congress 71, Amsterdam, North Holland, pp. 519–523, 1971.

(Kol 65) Kolmogorov, A.N. “Three Approaches to the Quantitative Definition of Information.” *Problems Inform. Transmission* , 1(1): 1–7, 1965

(Len 82) Lenat, D., “AM: Discovery in Mathematics as Heuristic Search,” in D. Lenat and R. Davis, *Knowledge Based Systems in Artificial Intelligence* , McGraw Hill, 1982.

(Len 86) Lenat, D., Parkash, M., Shepard, M., “CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks,” *The AI Magazine* , Vol. 6, No. 4, pp. 65–85, Spring 1986.

(Lev 73a) Levin, L.A. “Universal Search Problems,” *Problemy Peredaci Informacii 9* , pp. 115–116, 1973. Translated in *Problems of Information Transmission 9* , pp. 265–266.

(Mar 66) Martin–Löf, P. “The Definition of Random Sequences,” *Information and Control* , 9:602–619, 1966.

(Min 86) Minsky, L., The Society of Mind, Simon and Schuster, 1986.

(New 87) Newell, A., Laird, J.E., Rosenbloom, P.S., “Soar: An Architecture for General Intelligence,” Artificial Intelligence, Vol. 33, pp. 1–64, 1987.

(Pau 81) Paul, W.J., Seifers, J.I., Simon, J., “An Information Theoretic Approach to time Bounds of On–Line Computation,” J. Computers and System Sciences 23, pp. 108–126, 1981.

(Ris 84) Rissanen, J., “Universal Coding, Information, Prediction and Estimation,” IEEE Transactions on Information Theory, pp. 629–636, July 1984.

(Rum 86) Rumelhart, D.E., McCleland, J.L., Parallel Distributed Processing, Cambridge MIT Press, 1986.

(Sol 58) Solomonoff, R.J., “The Mechanization of Linguistic Learning,” Second International Congress on Cybernetics, pp. 180–193, 1958.

(Sol 59) Solomonoff, R.J. “A Progress Report on Machines to Learn to Translate Languages and Retrieve Information,” *Advances in Documentation and Library Science* , Vol. III, pt. 2, pp. 941–953. (Proceedings of a conference in September 1959).

(Sol 60a) Solomonoff, R.J. “A Preliminary Report on a General Theory of Inductive Inference,” Report V–131, Zator Co., Cambridge, Mass., Feb. 4, 1960. Also “A Preliminary Report on a General Theory of Inductive Inference,” (Revision of Report V–131), Contract AF 49(639)–376, Report ZTB–138, Zator Co., Cambridge, Mass., Nov, 1960.

(Sol 60b) Solomonoff, R.J., “A New Method for Discovering the Grammars of Phrase Structure Languages,” Information Processing, Unesco, Paris, 1960.

(Sol 62) Solomonoff, R.J., "An Inductive Inference Code Employing Definitions," ZTB 141, Rockford Research, Inc., April 1962.

(Sol 64a) Solomonoff, R.J. "A Formal Theory of Inductive Inference," *Information and Control* , Part I: Vol 7, No. 1, pp. 1–22, March 1964.

(Sol 64b) Solomonoff, R.J. "A Formal Theory of Inductive Inference," *Information and Control* , Part II: Vol 7, No. 2, pp. 224–254, June 1964.

(Sol 75) Solomonoff, R.J. "Inductive Inference Theory – a Unified Approach to Problems in Pattern Recognition and Artificial Intelligence," *Proceedings of the 4th International Conference on Artificial Intelligence* , pp 274– -280, Tbilisi, Georgia, USSR, September 1975.

(Sol 78) Solomonoff, R.J. "Complexity–Based Induction Systems: Comparisons and Convergence Theorems," *IEEE Trans. on Information Theory* , Vol IT–24, No. 4, pp. 422–432, July 1978.

(Sol 84) Solomonoff, R.J. "Optimum Sequential Search," Oxbridge Research, P.O. Box 391887, Cambridge, Mass. 02139, June 1984.

(Sol 86) Solomonoff, R.J. "The Application of Algorithmic Probability to Problems in Artificial Intelligence," in *Uncertainty in Artificial Intelligence* , Kanal, L.N. and Lemmer, J.F. (Eds), Elsevier Science Publishers B.V., PP. 473–491, 1986.

(Sto 88) Storer, J.A., *Data Compression* , Computer Science Press, Maryland, 1988.

(Val 84) Valiant, L.G., "A Theory of the Learnable," Communications of the ACM 27, pp. 1134–1142, 1984.

(Val 88) Valiant, L.G., and Pitt, L., "Computational Limitations of Learning from Examples," Journal of the ACM 35, no. 4, pp. 965–984, 1988.

(Win 75) Winston, P., "Learning Structural Descriptions from Examples," in: P. Winston (Ed.), The Psychology of Computer Vision, McGraw–Hill, 1975.