Autonomous Theory Building Systems

W.J. Paul, R.J. Solomonoff
Computer Science Department
University
D-6600 Saarbruecken
Germany

October 1990

1 Introduction

We are interested in very general systems which are programmed once and which from then on learn autonomously all sorts of things simply by observing a sequence of input data. In this preliminary note we give examples of techniques which apparently permit to deal with two basic aspects related to such systems: basic drive and complexity of learning steps.

A system which learns autonomously must have a criterion by which to decide what is worth learning. This criterion provides the basic drive for the system. We study very general criteria of this nature. They are related to the concept of building theories about the input data.

A theory about any set of data D is for us simply an algorithm t which reproduces the data. The theory t is nontrivial if the length of t is less than the length of the data D (both measured in the same unit, say bits). We will focus on two aspects of the complexity of nontrivial theories t: the amount of time it takes to find t and the length of t.

In sections 2 and 4 we will formally define theory building systems as well as some concepts of teaching and learning related to these systems. Sections 3, 5, 6 and 7 contain some techniques for constructing such systems. We feel that with proper teaching remarkably much can be learned by such systems with large but not astronomical computational effort.

We are trying to substantiate this by building such systems [BPT, B et al, BKP]. In general we are not able to predict what exactly will belearned in what time by these systems. If we were we would not bother to run experiments. We are however in certain situations able to give upper bounds on the time in which such systems will make certain progress. This allows to estimate the run time of the experiments and hence it is of great help in designing such systems. Several examples of this nature are presented.

We feel that large theory building systems, i.e. systems which have already learned many kinds of things, will have to develop nontrivial internal data structures in order to continue learning at a reasonable speed. Like the whole system these data structures have to develop autonomously out of certain building blocks. Algorithms which compute functions with a small range — like 0, 1 — should be very useful in this, for instance because they can be used as conditions in tests. As these algorithms have to be build by the system itself we need a general criterion about the usefulness of such algorithms. In section 7 we will define a test to be useful if it helps to make the overall theory shorter. We will illustrate this with an example.

2 Theory building systems, tests and training time

The formal definition of a theory building system has to be based on a machine model which provides complexity measures for computation time and storage space. A measure for storage space is only needed on I/O-devices. In order to be explicit we choose random access machines with uniform cost measure and with several seperate input tapes and output tapes. Basic operations are $+$ and $-$.

A *theory building system* now consists of 2 machines M1 and M2. The task of M1 is to build theories about its input data. The task of M2 is to reconstruct the input data for M1 from the theory built by M1. Formally M1 works in rounds r.

The start of a new round is signalled to M1 by toggling the bit under the head on input tape 2. This is the only function of input tape 2.

Let Σ be the tape alphabet of machines M1 and M2 and let $\#$ be a symbol not in Σ. In each round r some string $w(r) \in \Sigma^*$ is presented to

M1 on input tape 1. The sequence of all inputs $W(r) = w(1)\#\ldots\#w(r)$ is called the *current corpus*. For each r let $l(r)$ be the length of round r measured in steps performed by M1. The pair $e(r) = (w(r), l(r))$ is called the *current exercise* and the sequence $\epsilon(r) = (e(1), \ldots, e(r))$ is called the *current training sequence*. The sum of the lengths of all rounds is called the *length* of $\epsilon(r)$. During each round r machine M1 from time to time overwrites its output tape with a new string t. This string t is called the *current theory* or alternatively the *current description of the corpus*. It codes $W(r)$ in the following sense: machine M2 started with input t produces output $W(r)$ and halts.

It is well known that the capability of compressing data observed so far can be used to predict future data, but we will not pursue this here.

Informally speaking the purpose of theory building systems is to search for algorithms which help to compress the current description of the corpus. By looking at the output tape of M1 we have in a sense complete information about the algorithms discovered by M1. We might however not be able to understand them. Thus we check on the progress of the system by other means: we extend the current training sequence by a sequence S of exercises and measure the number b of alphabet symbols by which the curent theory grows while M1 processes S. The pair $T = (S, b)$ is called a *test* of the theory building system.

Next we would like to measure the amount of time necessary to train a system to pass a test T provided the system starts in configuration C. We call this amount of time $Tr(C, T)$, the *training time from C to T*. We define it tentatively as the length of the shortest training sequence E such that after E the system is able to pass test T. In section 8 we will discuss this definition further.

3 Incremental learning

Generally in order to be able to pass tests the system will have to find certain algorithms. Systematic enumeration of all programs of some programming language will eventually produce any program p as a candidate solution of problems but the number of trials grows exponentially with the length of the program p. For all but the most simple programs this is completely impractical.

But suppose the system is able to construct new programs p from old programs $p', p", \ldots$ which it already knows. Suppose moreover that the programs $p', p", \ldots$ are themselves useful for doing compression of certain inputs. Then one could try a training sequence which starts out with inputs for which $p', p", \ldots$ are useful. This approach divides the problem of discovering p into the subproblems of discovering $p', p", \ldots$ and finally of discovering a proper way to combine these programs in order to obtain p. In case programs $p', p", \ldots$ are too long to be found by enumeration, then the same approach can be used to find these programs.

We believe that very much can be learned this way. Certain basic concepts are so simple such that they can be found by enumeration. During incremental learning concepts which have been found to be useful in the past are combined to new useful concepts in extremely simple ways. We study an example for this related to the evaluation of arithmetic expressions in polish notation.

On the input tape of M1 we use the following alphabet: $A = \{0, 1, +, -, *, ,, =\}$. Inputs $w(r)$ are of the form $E(r) = V(r)$ where $E(r)$ is an expression in polish notation and $V(r)$ is the value of the expression. Numbers are coded in binary. A simple example would be $101, 10, +, =, 111$.

Theories of M1 will have the form $t = (d, c)$. Here $d = (d_0, \ldots, d_{k-1})$ is a sequence of programs in a programming language L which will be defined shortly. After the r'th round c is a sequence $c = (c_1, \ldots, c_r)$. For each $i \leq r$ component c_i is a triple $c_i = (E(i), j(i), F(i))$ where $j(i) \in \{0, 1\}$ and $V(i) = F(i)$ if $j(i) = 0$, $V(i) =$ the output of $d_{F(i)}$ started with input $E(i)$ if $j(i) = 1$. Machine M(2) is not much more than an interpreter of programs in L.

Programs in L are interpreted as commands for a pushdown machine P. Machine P is capable of storing in a single cell of its pushdown store elements from $A' = \{0, 1\}^* \cup \{+, -, *\}$. Inputs for P are strings in A^*. They are interpreted as sequences $h(1), ..., h(s)$ of elements $h(i) \in A' \cup \{=\}$ separated by commas. Programs for P are generated by the following context free grammar:

$\langle$program$\rangle$ ::= $\langle$statement$\rangle$ —
$\langle$statement$\rangle$;$\langle$program$\rangle$

$\langle$statement$\rangle$ ::= if $\langle$condition$\rangle$ then action —

while ⟨condition⟩ do ⟨program⟩ od —
call(⟨number⟩)

⟨action⟩ ::= add — sub — mul — push

⟨condition⟩ ::= num — + — − — ∗ — = — not ⟨condition⟩

⟨number⟩ ::= 0 — 1 — 0⟨number⟩ — 1⟨number⟩

Let P' = the number of productions which have ⟨program⟩ as left hand side and let $P = 1/P'$. Define S, A, C similarly for ⟨statement⟩, ⟨action⟩ and ⟨condition⟩. Thus we have $P = 1/2, S = 1/3, A = 1/4$ and $C = 1/6$.

We explain the semantics of programs in L. At any time the head on the input tape touches some $h(i)$. Actions add, sub and mul operate on the two top elements of the stack in the obvious way and the head on the input tape is advanced such that it touches $h(i+1)$. Action push pushes $h(i)$ on the stack and advances the head on the input tape such that it touches $h(i+1)$. Condition num is true if $h(i) \in \{0,1\}^*$, condition s is true if $h(i)$ is s for $s \in \{+, -, *, =\}$. In a statement of the form call(j) the number j refers to program d_j in d. The result of a computation is the top element of the stack at the end of the computation.

Machine M1 will generate canditate programs starting from ⟨program⟩ by applying randomly and independently productions in the grammar. For nonterminals other than ⟨number⟩ the possible right hand sides are chosen with equal probability. If the sequence d currently contains k numbers then nonterminal ⟨number⟩ is replaced by the binary representation of a number between 0 and $k-1$. All numbers are picked with equal likelihood $1/k$.

Much more sophisticated ways of generating programs are possible. We will consider some of them later.

We would like M1 to discover a program p like

```
while not = do if num then push;
          if + then add; if − then sub; if ∗ then mul
od
```

We compute the probability of discovering this program. We have to multiply the following probabilities:

Expanding ⟨program⟩ to ⟨statement⟩ : P

Expanding this to while ⟨condition⟩ do ⟨program⟩ od : S

¿From ⟨condition⟩ to not = : C^2

¿From ⟨program⟩ to ⟨statement⟩;⟨statement⟩;⟨statement⟩ : P^3

¿From the first ⟨statement⟩ to while num then push: $S \times C \times A$

The same for the next two occurences of ⟨statement⟩.

Thus in each try one finds p only with probability $P^4 \times S \times C^2 \times (S \times C \times A)^3 = 1/(2^4 \times 3^4 \times 4^3 \times 6^5)$ which is roughly $1/(645 \times 10^6)$.

The following arguments suggest that by incremental learning one can speed this up very much. In what follows we will often speak of the *length* of objects which are not strings. What is meant is the length of some standard encoding of the object. In the case of natural numbers we mean the length of the binary representation without leading zeros.

In each round r machine M1 will work in the following way.

Let $t = (d, c)$ be the current theory and $w(r) = (E = V)$. First M1 extends c by the triple $(E, 0, V)$. Next for all programs d_j in d program d_j is applied to E. If it produces V and if the length of j is less than the length of V then $(E, 0, V)$ is replaced by $(E, 1, j)$.

Suppose d contains k programs all of which fail. Let $l =$ the length of V.

$(*)$: M1 tries to generate programs d' such that length(d')+length$(k)+1 < l$ and which compute V from E. Programs which contain a while loop which is executed more than length(E) times are aborted (much more general ways exist for dealing with programs that don't halt [L, S85], but they are not needed here). If such a program is found then d is extended by d' (this increases the length of d by length$(d') + 1$ for the comma before d') and the last component of c is replaced by $(E, 1, k)$. The bound l is replaced by the length of the new program d_k and k is replaced by $k + 1$. The search for even shorter programs continues at $(*)$.

Much more sophisticated methods to produce theories are possible. We will consider some of them later.

We now present the inputs $w(r)$ of a training sequence and a sequence of programs d_j which ends with a program equivalent to the program p presented above. For each j we compute the probability $p(j)$ of finding d_j in a single trial provided $d = (d_0, \ldots, d_{j-1})$. All programs in the language L run fast or are aborted. Therefore for each j the probability $p(j)$ is a reasonable measure of the amount of computation needed to expand the sequence $(d_0, \ldots, d_{j-1})$ by d_j, and we will not bother to determine the length

$l(r)$ of the rounds of the training sequence exactly. We will be interested in sequences where the minimum of the $p(j)$ is as large as possible, i.e. where the hardest learning step is as easy as possible.

Let d_0 be the program: if num then push. Let n be a number such that $\text{length}(n) > \text{length}(d_0) + \text{length}(0) + 1$. Let $w(1)$ be the input $n, =, n$. Then in each try program d_0 is generated with probability $p(0) = P \times S \times C \times A = (1/2) \times (1/3) \times (1/6) \times (1/4) = 1/144$. If d_0 is generated and d is still empty then d_0 is included into d. This follows by the construction of M1.

Next let $d_1 = \text{call}(0)$; call(0) ; if + then add. Let a, b and c be numbers such that $a + b = c$ and such that $\text{length}(c) > \text{length}(d_1) + \text{length}(1) + 1$. Let $w(1)$ be $a, b, +, =, c$. In each try expansion of $\langle\text{program}\rangle$ to call $\langle\text{number}\rangle$; call $\langle\text{number}\rangle$; if + then add happens with probability $P^3 \times S^3 \times C \times A = (1/8) \times (1/3)^3 \times (1/6) \times (1/4)$. Expansion of $\langle\text{number}\rangle$ to 0 happens with probability 1 because there is no choice. Thus $p(1) = 1/5184$.

Let $d_2 = \text{call}(0)$; call(0) ; if $-$ then sub. Choose a, b, c such that $a - b = c$ and such that $\text{length}(c) > \text{length}(d_2) + \text{length}(2) + 1$. Let $w(2) = a, b, -, =, c$. In each try program d_2 is found with probability $p(2) = (1/2)^2 \times p(1)$. The reason is that in the expansion of $\langle\text{number}\rangle$ there are now the two choices 0 and 1. 2 Let $d_3 = \text{call}(0)$; call (0) ; if $*$ then mul. We find $p(3) = (1/3)^2 \times p(1)$ in an analogous way.

Finally let d_4 = while not = do call(1) ; call(2) ; call(3) od. Choose $a, \ldots, h$ such that $((a + b - c) * d + e - f) * g = h$ and such that $\text{length}(h) > \text{length}(d_4) + \text{length}(4) + 1$. Choose $w(4) = a, b, +, c, -, d, *, e, +, f, -, g, *, =, h$. In each try expansion from $\langle\text{program}\rangle$ to while not = do $\langle\text{statement}\rangle$; $\langle\text{statement}\rangle$; $\langle\text{statement}\rangle$ od happens with probability $P \times S \times C^2 \times P^3 = (1/2) \times (1/3) \times (1/6)^2 \times (1/8) = 1/1728$. Each statement is expanded to call $\langle\text{number}\rangle$ and then to the right call(j) with probability $S \times (1/k) = (1/3) \times (1/4)$. Thus $p(4) = (1/1728) \times (1/12)^3 = 1/2985984$.

Although this is still bad it is much better than the probability of finding p in our original example above.

4 Training plans

The calculation in section 3 is a heuristic argument in support of the usefulness of incremental learning. It does not imply any reasonable upper bound about the training time from the initial configuration to a test T where knowledge of program d_4 is useful (such a test might consist of an input $(E, =, V)$ similar to $w(4)$ above and a bound b very slightly larger than the length of E). For example we did not take into account the possibility, that completely different programs are enumerated and included into d.

On the other hand we also have ignored certain things which contribute to faster learning. For instance any permutation of the three call statements in d_4 would give a program which does the desired job. Hence if $d = (d_0, \dots, d_3)$ then the probability to find in one try some program equivalent to d_4 is at least $6 \times p(4)$.

We will discuss the topic of actually proving upper bounds on training time further in section 8.

Actually in section 3 we have only analyzed one particular possible development of configurations which according to our plans the training could produce. We could have checked on the progress of that plan with tests every now and then. If even one of the probabilities, which we computed is very small, then there is reason to worry that the training might take very long. In this sense the minimum of the probabilities $p(i)$ is a measure for the difficulty of the plan.

Of course there might be short cuts we did not think about. On the other hand the training might result in the discovery of programs which are completely different from the ones which we expected. In case one uses as in section 5 below mutations of known programs as a strategy to generate new programs, then this might lead to a situation where we plan for the system to modify a program which it never discovered in the first place.

Thus exact prediction of what will happen seems to be extremely difficult. On the other hand the difficulties above are quite similar to those occuring in the training say of children. Moreover with the probabilities $p(i)$ or related quantities we even have a heuristic quantitative measure for the difficulty of our plan and we can use these probabilities for estimating the time one has to give the system before it can pass the next test.

Formally we define a *training plan* as a sequence of triples (E_i, C_i, T_i) where for all i component E_i is a training sequence, C_i is a configuration of the theory building system and T_i is a test which can be passed if the system starts in configuration C_i. The probabilities $p(i)$ depend on the training plan we have in mind and not only on the training sequence. They are related to the concept of conceptual jump size in [S89].

In the next sections we will analyze a few more training plans. We will present two techniques which help to improve the situation in section 3 dramatically.

5 Mutating programs

Probability $p(4)$ was so bad, because three statements had to expanded simultaneously in the right way. Things might be better if after d_3 we could proceed to teach the following programs

d_4 : while not = do call(1) od

d_5 : while not = do call(1) ; call (2) od

d_6 : while not = do call(1) ; call (3) ; call (4) od

In order to make this possible we provide machine M1 from section 4 with a second strategy to generate programs: mutation. The old strategy is now called pure generation. At the beginning of any try to generate a new programs one of the two strategies is chosen, each with equal probability $M = 1/2$.

The strategy 'mutation' first randomly chooses a program d' in d. This is the program which will be mutated. All programs are chosen with equal probability. Next in the derivation tree of d' an occurrence ω of the nonterminal ⟨statement⟩ is chosen. Each occurence is chosen with equal probability. The whole subtree with root ω is replaced by the nonterminal ⟨program⟩. From then on strategy 'pure generation' is applied to complete the program.

Let us return to the examples above. Probabilities $p(1)$ through $p(3)$ must now be multiplied with M because of the choice between the two strategies. Teach d_4 without using mutation. Choose an input, where sufficiently many large numbers are summed. Then

$p(4) = M \times P \times S \times C^2 \times P \times S \times (1/k) = (1/2) \times (1/2) \times (1/3) \times (1/6)^2 \times (1/2) \times (1/3) \times (1/4) = 1/10368$.

Now in order to teach d_5 pick an input where several numbers are added and subtracted. Program d_5 can be obtained from d_4 by mutation. In the derivation tree of d_4 there are $s = 2$ occurences of the nonterminal statement. Pick the one within the while statement. Then expand ⟨program⟩ to call(0) ; call(1). One gets

$p(5) = M \times (1/s) \times P^2 \times (S \times (1/k))^2 = (1/2) \times (1/2) \times (1/2)^2 \times (1/3)^2 \times (1/4)^2 = 1/2304$.

Finally in order to obtain d_6 from d_5 one has to replace the occurence of ⟨statement⟩ which produces call(2) by ⟨program⟩ and to expand this to call(2) ; call(3). One can choose between 3 occurences of ⟨statement⟩ and there are now 5 choices for the parameters j of call(j). Thus

$p(6) = p(5) \times (2/3) \times (4/5)^2 = 1/5400$.

6 Compression of old inputs and renaming of programs

Let d' be some program that is tried in round r. So far in order to get included into the sequence d program d' had to be able to help compress the description of the actual input $w(r)$. Now we extend machine M1 further. Every program d' that is generated is also tried on previous inputs. Suppose d contains k programs, suppose program d' is generated, and let I be a set of indices i such that (1) and (2) hold:

(1) application of d' to $w(i)$ gives $V(i)$ for all $i \in I$.
(2) $\text{length}(d') + |I| \times \text{length}(k) + 1 < \sum \text{length}((i))$, where the summation is over all $i \in I$.

Then it makes sense for M1 to extend d by d' and replace $(j(i), F(i))$ by $(1, k)$ for all $i \in I$.

Next, let us consider an example, where $j(i) = 1$ for all $i < r$, i.e. in all rounds so far $F(i)$ is some number $< k$ of some program. Suppose moreover that d' is not shorter than the progams found so far. Then condition (2) is false and d' would not be added to d, even if condition (1) would be true for $I = \{1, \ldots, r\}$. Intuitively it seems wrong that a theory building system should fail in such a situation to include d' into d. This motivates the following extension of M1.

Theories have now the form (d, π, c) where components d and c have the same format as before. Suppose d has k elements. Component π is a permutation of $\{1, \ldots, k\}$. The particular code used for π is not important to illustrate our examples. The meaning of the components $F(i)$ in c changes: we write $F(i) = b$ if program $d_{\pi(b)}$ applied to $E(i)$ gives $V(i)$. This permits to give short numbers to programs which are useful more often than others. It should be clear how to modify M1 in order to support this.

Even without using mutations we can now treat the example of sections 4 and 5 in the following way. Teach d_0 through d_3 as before. Next present a long sequence of inputs which alternate between the forms $a, b, +, =, c$ and $a, b, -, =, c$ and $a, b, *, =, c$. Let d_4 be the following program: call(1); call(2) ; call(3). For this program we have $p(4) = P^3 \times S^3 \times (1/k)^3 = (1/2)^3 \times (1/3)^3 \times (1/4)^3 = 1/13824$.

Choose π such that $\pi(0) = 4$. Suppose $F(i) = 2$ for at least $l = \text{length}(d_4) + \text{length}(\pi) + 2$ indices i. Then including π and d_4 will not increase the length of the current theory by more than $l - 1$, even if the length of (the code of) the old π was 0. But now all $F(i)$ can be made 0. Because in binary representation we have $\text{length}(2) = 2 > 1 = \text{length}(0)$ the length of at least l components in c decreases by 1.

If one uses mutations then one can teach this in two steps, each with an even higher probability.

Now let d_5 be the program while not = do call(4) od. Choose an input as for d_4 in section 4. We have $p(5) = P \times S \times C^2 \times P \times S \times (1/k) = (1/2) \times (1/3) \times (1/6)^2 \times (1/2) \times (1/3) \times (1/5) = 1/6480$.

7 Towards structuring the sequence of programs learned

As the sequence d becomes longer and longer there are more and more choices for the parameters of call statements. If d contains k programs and one wants to generate a new program whch contains m new call statements, then the probability of getting them right is proportional to k^{-m}. This quickly becomes a problem.

For reasons like this it is desirable to structure the internal collection of programs of a theory building system into something like directories. There should be simple tests to be performed on the input data which determine the directory to work in. Each directory should only contain a limited number of programs.

There should be ways for M1 to generate directories and candidates for tests. We will not address here the question how to do this. We will only focus on the question how to decide what is a good combination of directories and tests. In the previous sections we have used a general philosophy to judge the quality of a program: good programs are those which help to compress one or several input words $w(i)$. If we have directories $D_1, \ldots, D_m$ then tests should map the current input word $w(r)$ to the number $n \in \{1, \ldots, m\}$ of an appropriate directory. Thus tests are of no direct use to compress input words and the above criterion cannot be applied directly.

We will now extend the previous example of a theory building system such that it comes into a situation where it is intuitively plausible to introduce two directories and a test to decide which one to use. We will show how the introduction of such a structure helps to shorten the current theory. This immediately implies a measure of quality of the structure.

We extend the set of tasks which the theory building system is supposed to learn: instead of 3 different operator symbols $+, -, *$ we want the system now to be able to learn to deal with a large even number n of operator symbols $s_1, \ldots, s_n$. In language L there are now n operations $op_1, \ldots, op_n$ which replace add, sub and mul.

Program d_0 is as before. One now teaches for all i the following programs d_i: call(0) ; call(0) ; if s_i then op_i.

Let $S_1 = \{s_1, \ldots, s_{\frac{n}{2}}\}$ and let $S_2 = \{s_{\frac{n}{2}+1}, \ldots, s_n\}$. We from now on present only inputs with several operators where for each r the operator symbols in $w(r)$ are either all in S_1 or all in S_2. The system is supposed to find for all $i < n/2$ successively the following programs

e_i : call(1) ; $\ldots$; call(i) and

f_j : call($n/2+1$); $\ldots$; call($n/2+i$)

Finally it is supposed to arrive at programs

while not = do call(1) ; $\ldots$; call ($n/2$) od and

while not = do call($n/2+1$) ; $\ldots$; call(n) od.

For the purposes of our example we view a directory simply as a way to specify a small subset D of a large set d such that the following holds: if we know that we are in D then we can specify elements in D with few bits. Suppose d has k elements and D has $K < k$ elements. Then we can view D as a function $D : \{0, \ldots, K-1\} \to \{0, \ldots, k-1\}$.

We extend the theories t of our theory building system further to include directories. In our example there will be the following two directories.

$D_1, D_2 : \{0, \ldots, n/2-1\} \to \{0, \ldots, k-1\}$

with $D_1(i) = i+1$ and $D_2(i) = n/2+1$ for all i.

There is a test which depends only of the first operation symbol s occuring in the current input word. It activates D_1 if $s \in S$; otherwise it activates D_2. The semantics of a statement call(j) is now made dependent on the current directory D, namely call(j) refers to $d_{D(j)}$.

Clearly even in the most brute force way the directories can be specified with $O(n \log n)$ bits. But they allow to shorten each parameter in a call statement of the programs f_i by one bit. Because there are $1+2+\ldots+n/2 = \Theta(n^2)$ such statements inclusion of the directories and the corresponding test into the current theory will at some point actually allow to shorten the length of the current theory.

8 Final remarks

We have presented some techniques which are useful for constructing theory building systems and a way to estimate the rate of progress these systems make in a training plan. The techniques for constructing M1 are very general and can be applied for all kinds of problems (see e.g. [BKP]). The language L was chosen to match very closely the problem of evaluating expressions. In general it is clearly desirable to use a universal language. The techniques presented here work for any language, but in a richer language the probabilities $p(i)$ become quickly very small.

Therefore it will be of crucial importance to develop a self organizing directory structure which depending on the current input narrows down the possibilities for generating programs. In section 7 we have seen that there are very general ways to judge the quality of such a structure.

Another important goal is to set up a mechanism which allows a system to discover and apply laws like for example commutativity.

We hope that a small number of such mechanisms and large but not astronomical computational resources will permit to construct theory building systems which exhibit surprising beheaviour.

So far we have not excluded the possibility of programming a theory building system explicitely. For exaple we could set up the system such that in every 100'th round it adds the current input to it's collection of programs. Then of course the training time $Tr(C,T)$ from any configuration C to a test T which can be passed if the system knows program d' would be $O(\text{length}(d'))$.

There is little hope that a more restrictive definition of theory building systems will get rid of this difficulty: if a theory building system is not in principle able to learn any recursive function, then it is very restrictive. Otherwise an interpreter for a universal language can be learned with training time $O(1)$. From then on explicit programming is possible. To define formally (and then to forbid) explicit programming seems to be the only way to make the function Tr nontrivial, but we do not know how to do this.

On the other hand this state of affairs is not terribly disturbing. The explicit teaching of algorithms (with no proof of correctness) is used to a considerable extend in the education of people, e.g. when one learns in elementary school how to multiply long numbers. Thus it is not even clear that we want to forbid this. If we are willing to allow it, then the argument captures the simple and true fact that learning an explicitely presented program is a trivial matter for a machine.

In our view however the most interesting situation in machine learning arises when we do not know ahead of time what program will solve a given problem and where the machine discovers the program itself. It seems to be very hard to find out much about this by doing theory alone. Running experiments seems to be crucial.

References

[BPT] Bergmann P., Paul W.J., Thiele L.,“An Information Theoretic Approach to Computer Vision”, Dynamical Networks, Berlin, 1989, pp. 52–58.

[B et al] Bergmann P., Keller J., Malter T., M”uller S.M., Paul W.J., Pöschel T., Schlüter O., Thiele L.,“Implementierung eines informationstheoretischen Ansatzes zur Bilderkennung”, Proc. Innovative Informations–Infrastrukturen, III–Forum, Saarbrücken, 1988, pp. 187–197.

[BKP] Bergmann P., Keller J., Paul W.J.,“A Selforganizing System for Image Recognition”, Proc. Neural Networks and Machine Learning, IASTED–Conference, New York 1990.

[L] Levin L.A.,“Universal Search Problems”, Problemy Peredaci Informacii 9, pp. 115–116, Translated in: Problems of Information Transmission 9, pp. 265–266.

[S78] Solomonoff R.J.,“Complexity–Based Induction Systems: Comparisons and Convergence Theorems”, IEEE Transactions on Information Theory, Vol. IT–24, No. 4, 7/1978.

[S85] Solomonoff R.J.,"Optimum Sequential Search", Oxbridge Research Report, Oxbridge Research, Box 559, Cambridge, Mass. 02238, Rev., 1985.

[S89] Solomonoff R.J.,"A System for Incremental Learning Based on Algorithmic Probability", Proc. of the 6th Israeli Conference on Artificial Intelligence, (Computer)Vision and Pattern Recognition, DEC 26–27 1989, pp. 515–527.

Problems and Issues are stated in section 8 of the paper.

Workgroup suggestions and comments

The first authors presentation of the above ideas in the workshop was quite informal (to put it mildly). Partly because of this but also partly because of an apparent urge to speak about the matter the work group discussed the sociological and philosophical consequences which would arise if one would succeed in the construction of smart machines.

The ideas discussed were reasonably disturbing but this is perhaps due to the fact that the ideas were based on speculation rather than on technical under standing. Only one centuriy ago one would have found the idea of machines made out of silicon and capable of multiplying numbers very disturbing. After all the capability to multiply distinguishes man from animal and one would have suspected that a soul is needed in order to do it.