Some Experiments in Machine Learning

HOWARD CAMPAIGNE†

EVER since the development of automatic sequence computers it has been possible for the machine to modify its own instructions, and this ability is the greatest single faculty in the complex that tempts the term "giant brain." Friedberg[1] demonstrated a new technique in the modification of instructions; he allowed the machine to make alterations at "random," and lent direction to the maneuver by monitoring the result. This technique is far from being a feasible way to program a computer, for it took several hundred thousand errors before the first successful trial, and this was for one of the simplest tasks he could imagine. A simple principle of probabilities shows that a task compounded of two tasks of this same complexity would take several hundred thousand times as long, perhaps a million computer hours. It is the object of this study to examine techniques for abbreviating this process.

The work reported here is not complete, nor is it likely to be for several years. The field is immense, the search proceeds slowly, and there are few clues as to where to look. This paper is, therefore, in the nature of a preliminary report.

The memory of a computer can be pictured as a piece of scratch paper, ruled into numbered cells, on which notes can be written or overwritten. I set aside a portion of this memory in which the computer is supposed, somehow, to write a program. Some of this dedicated space is for data and some for instructions. I programmed a generator of random numbers, and instructed the computer to use these numbers to write a program (and later to modify a program already written). The method of using the random numbers is such that all addresses generated refer to the data, and all instructions generated are from an approved list. After a little thought one sees that by putting sufficient limitations on what the computer is allowed to write the result will be a foregone conclusion. The object is to design an experiment which leaves the computer relatively free from limitations but which will still lead to a meaningful result. Friedberg did this.

One way for the machine to write the instructions is to have it write random bits into a form word, the form and the dedicated spaces being selected to be coherent.

Now a simple task is envisioned and a monitor routine written to test whether the randomly generated program (which Friedberg christened HERMAN) has accomplished the task. If it has it is tried again until the probability is high that the task is being done correctly. If at any step the task is not done then a change is made before another trial. It is clear that once a routine which can do the task has been arrived at no further changes will be made. The procedure can be pictured as a random walk. The number of possible programs is finite; in one of my experiments it was 2^{96}. The rule for going from one trial to another can be chosen in various ways, but in the same experiment there were just 64 alternatives at each step. The number of routines which satisfy the test must be, judging from my results, on the order of $2^{96}/2^{12}=2^{84}$. For some tasks and for some repertories of instructions it can be estimated directly.

AN EXAMPLE OF A MACHINE WRITTEN PROGRAM

Operation		*Operand*	*Address*	*Data*
QJP 27	00	0447	0440	13351604
COQ 24	00	0443	0441	77777777
LDQ 22	00	0453	0442	0
STQ 23	00	0457	0443	57312317
COQ 24	00	0452	0444	0
QJP 27	00	0452	0445	77777777
LDQ 22	00	0445	0446	77777777
QJP 27	00	0457	0447	77777777
STQ 23	00	0444	0450	0
LDQ 22	00	0444	0451	0
STQ 23	00	0450	0452	0
QJP 27	00	0454	0453	77777777
COQ 24	00	0445	0454	54040476
STQ 23	00	0442	0455	77777777
COQ 24	00	0447	0456	77777777
LDQ 22	00	0451	0457	0

In designing these experiments one has a tremendous number of choices. There is the repertory of instructions from which the machine chooses to make up its routine. The instructions of this repertory need not be selected with equal probability; some can be used more often than others. There is the size of the area of the memory dedicated to the random routine. There is the task to be performed. There is the time allowed the routine to make its trial. And there are a multitude of other variations, some of which will be mentioned later.

I first used the simplest repertory to be found capable of performing the Sheffer "stroke" function. It is:

LOAD from y into the accumulator,
STORE at y from the accumulator,
COMPLEMENT the accumulator,
JUMP to y if the accumulator is positive.

This differs drastically from that used by Friedberg.

The area of the memory devoted to the random routine must be in two parts, one for data and the other for instructions; otherwise the machine would try to execute data and come to an intolerable halt. Therefore the address y of the JUMP order must be interpreted differently from those of the LOAD and STORE orders.

A problem related to that which leads to the separation of the two dedicated areas is that endless loops are highly probable and intolerable. This problem can be

† American University, Washington, D. C.
[1] R. M. Friedberg, "A learning machine," pt. I, *IBM J. Res. and Dev.*, vol. 2, p. 2; January, 1958.

solved by the routine which interprets the jump addresses. At the occasion of a JUMP a calculation is made about the time of running; if the time is excessive then the trial is terminated and called a failure. If the time is acceptable then the address to which the jump is made is interpreted.

In an earlier experiment I used a clock to time the routines and adjusted the jump addresses with a B-box. The use of a clock made checking and debugging very difficult and eventually forced me to the later method.

I have used several sizes of dedicated spaces and plan to try still more, but most of the trials have been with sixteen lines of coding. Friedberg used sixty-four lines. Clearly one must allow enough lines to permit coding the task. More than enough slows the "learning" process. To allow just enough prejudices the event and is not "fair." A felicitous solution to this quandary would be to find a way to expedite the "learning" so that very large areas of memory can be dedicated, thus guaranteeing that the answer has been supplied by the machine. It is the object of this study to find such a solution.

The time allowed to execute the routine can be varied. Its absolute minimum is that for three instructions. The maximum could be quite large if the routine were retraced several times. Oddly enough the time allowed on each trial has only a small effect on the number of trials before learning. This may be because when the machine has extra time it uses that time to destroy what it may already have accomplished. If so then two effects tend to neutralize each other; the freedom of more time tends to lengthen the learning period, and the greater number of potential solutions tends to shorten it (Table I).

TABLE I

THE EFFECT OF TIME ON LEARNING

Time	Median number of trials	Number of experiments
14	4400	31
10	5800	47
7	6800	30
4	15,500	13

These experiments differed only in the time allowed each trial. The time is measured by the number of program steps possible. Each experiment was run until the machine had demonstrated that it succeeded (one hundred consecutive successes) and then the number of trials was recorded. In this set of experiments about 38 per cent of the trials appeared by accident to be successes. Thus the number of different programs tried was about 62 per cent of the total number of trials. As the time was shortened it became more difficult to find a successful routine.

The median has been quoted here to avoid a bias which might affect the average. The range of the number of trials is large, and there might be some prejudice against the longest runs, such as stopping them to check for faults.

The task selected was to transfer a word from one place to another. This was chosen because of its simplicity. An easier task was to copy a word at a prescribed spot from one of several places (redundant input). A harder task would be to copy a word at several places (redundant output). With redundant input (the key word written at two accessible places) the median number of trials was 2600, compared with 5800 in non-redundant experiments.

The procedure for the machine making changes in the routine offers many opportunities to be different. In one series of trials the routine was rewritten completely after each failure. In another only one instruction was rewritten, the instructions being taken in turn in successive tests. In another series just one instruction was rewritten, this time chosen by an elaborate procedure involving "success numbers." The success numbers were accounts, one for each instruction of the routine, which were increased when a success or what appeared to be a success happened, and decreased when a failure occurred. This procedure was roughly the same as that of Friedberg's, although not quite as elaborate.

A comparison of the results of these alternative methods were that "learning" occurred most rapidly with the first method, about twenty-five hundred errors before a success. It was less rapid with the second, about six thousand trials, and slowest with the last, over one hundred thousand trials. The "learning" is an abrupt process, a "flash of insight." The procedure with success numbers resembles that used in other experiments with self-improving programs, such as those used by Oettinger.[2] It seems to be inappropriate here since a line of coding is not itself a unit. If the program were organized into units to which success numbers could be appropriately applied then it would begin to appear that the answer to our problem was being built into our approach. Success numbers will have an important place in the ultimate "learning" machines, but some sort of self-improving without them will also be required.

A SAMPLE OF EXPERIMENTS ARRANGED IN ORDER

No. of Trials	*Apparently Right*	*No. of Trials*	*Apparently Right*
20	20	576	273
38	32	592	280
54	37	598	303
68	39	612	317
72	46	614	297
76	40	618	290
76	47	680	331
88	53	800	385
156	77	996	509
160	89	1090	526
162	88	1100	526
234	131	1246	595
222	124	1324	629
276	130	1412	688
294	153	1420	678
330	160	1528	753
338	173	1616	782
352	160	2016	990
386	181	2872	1366
521	249	3198	1522

[2] A. G. Oettinger, "Programming a digital computer to learn," *Phil. Mag.*, vol. 43, pp. 1243–1263; December, 1952.

TABLE II
SYNOPSIS OF EXPERIMENTS

Type of change		Length	Time	Task		Competing twins?		Median no. of trials before learning	Notes
All	One at a time			Transfer word	Choose an exit	Yes	No		
	X	2	2		X		X	548	
X		14	10		X		X	960	Reversed exit
	X	14	10		X		X	1530	Input redundant
X		16	12	X			X	1567	
	X	16	12	X		X		1775	
	X	8	4	X		X		1902	
X		16	10		X		X	2097	
	X	16	10		X	X		2210	
	X	8	4	X		X		2280	
X		16	10		X		X	2608	Reversed exit
	X	16	10		X		X	2683	
X		16	10		X		X	2998	Reversed exit
	X	16	10		X		X	3125	Input redundant
	X	16	10		X		X	3610	Input redundant
X		16	10		X		X	4200	
	X	16	12	X			X	4300	
	X	16	10		X		X	4401	
X		16	10		X		X	4406	
	X	8	4	X			X	4988	
	X	16	8	X		X		5177	
	X	8	4	X			X	5280	
	X	16	10		X	X		6700	
	X	16	7		X		X	6800	
	X	16	10		X	X		7273	
	X	16	10		X		X	8600	
	X	16	8		X	X		9266	

In these experiments the minimum possible space, two orders, was allowed for the program. The learning was judged complete when 20 successive right answers were given. Thus there is a chance of something less than one in a million that a given experiment did not succeed, despite appearances. Notice the wide range, the slowest taking 160 times as long as the fastest (Table II).

These experiments throw some light on this kind of "learning," and particularly on which versions learn fastest. The process is analogous to evolution, since a routine survives only if it meets the challenge of its environment. This analogy suggests some further experiments where each trial is built on two successful trials, perhaps by taking lines of coding from each, much as genes are taken from chromosomes. I have hopes that routines can be built in this manner to meet complex environments and that the number of trials will be only the sum of the number for each individual requirement rather than the product.

In the experiments described above once a task has been "learned" there is no further improvement. It would be desirable for the routine to improve its performance even after it had demonstrated acceptable skill. This can only be done by some sort of flexible criterion of satisfaction, and by some way of keeping progress already made. I have tried to do this by twin learning routines. The twins compete to see which will first learn the task. When one of them has learned the other continues to try, but now it must complete the task more quickly than its sister. In this way some improvement takes place. It is not quicker than the alternative of insisting on a high standard *ab initio*.

Other techniques need to be tested. One of these is to use as components not lines of coding but subroutines. In this way the average coherence of the trials should be raised. Another is some way of accumulating successes and then using them cooperatively to meet more and more complex environments. This resembles biological history, where evolution has produced increasingly complex organisms which become more and more effective in dealing with their environment.

If someone could invent a technique which produced programs as effective as organisms in a time which is electronic rather than biological it would be a revolution in programming.