"MEMO" FUNCTIONS AND MACHINE LEARNING

By
DONALD MICHIE

(*Reprinted from Nature, Vol. 218, No. 5136, pp. 19–22, April 6, 1968*)

(*Reprinted from Nature, Vol.* 218, *No.* 5136, *pp.* 19-22, *April* 6, 1968)

"Memo" Functions and Machine Learning

by
DONALD MICHIE
Experimental Programming Unit,
Department of Machine
Intelligence and Perception,
University of Edinburgh

It would be useful if computers could learn from experience and thus automatically improve the efficiency of their own programs during execution. A simple but effective rote-learning facility can be provided within the framework of a suitable programming language.

If computers could learn from experience their usefulness would be increased. When I write a clumsy program for a contemporary computer a thousand runs on the machine do not re-educate my handiwork. On every execution, each time-wasting blemish and crudity, each needless test and redundant evaluation, is meticulously reproduced.

Attempts to computerize learning processes date back little more than 10 yr. The most significant early milestone was A. L. Samuel's study[1] using the game of checkers (draughts). Samuel devised detailed procedures both of "rote-learning" and "learning by generalization". When coupled with efficient methods of lookahead and search, these procedures enabled the computer to raise itself by prolonged practice from the status of a beginner to that of a tournament player. Hence there now exists a checkers program which can learn through experience of checkers to play better checkers. What does not yet exist is any way of providing in general that all our programs will automatically raise the efficacy of their own execution, Samuelwise, as a result of repeated "plays".

Here I outline proposals[2] for enabling the programmer to "Samuelize" any functions he pleases, and so endow his program with self-improving powers—both of "rote-learning" and "learning by generalization". I shall be talking about mathematical functions. "Factorial" (of a natural number) is a function; so is "highest common factor" (of a pair of real numbers); so is "member" (of an element-set pair); so is the "reverse" (of a list). Thus *factorial* (5) = 120; *hcf* (63, 18) = 9; *member* (Queen Elizabeth, the Cabinet) = *false*; and *reverse* ([Tom Dick Harry]) = [Harry Dick Tom]. The present proposals involve a particular way of looking at functions. This point of view asserts that a function is not to be identified with the operation by which it is evaluated (the rule for finding the factorial, for instance) nor with its representation in the form of a table or other look-up medium (such as a table of factorials). A function is a function, and the means chosen to find its value in a given case is independent of the function's intrinsic meaning. This is no more than a restatement of a mathematical truism, but it is one which has been lost sight of by the designers of our programming languages. By resurrecting this truism we become free to assert: (1) that the apparatus of evaluation associated with any given function shall consist of a "rule part" (computational procedure) and a "rote part" (look-up table); (2) that evaluation in the computer shall on each given occasion proceed either by rule, or by rote, or by a blend of the two, solely as dictated by the expediency of the moment; (3) that the rule versus rote decisions shall be handled by the machine behind the scenes; and (4) that various kinds of interaction be permitted to occur between the rule part and the rote part. Thus each evaluation by rule adds a fresh entry to the rote. Updating the rote by assignment from outside the function apparatus (such as from the tape reader or other input channel) is allowed, and may alter some of the consequences of future evaluations by rule. The rule

itself may be self-modifying and seek to increase its agreement with the contents of the rote.

Parable of Self-improvement

Before describing the detailed application of this scheme, I shall take a fanciful example to show how exactly it corresponds to the way a human tackles such problems. Imagine that I am hired by an old-fashioned commercial company which does not possess, or believe in, calculating machines, "We need someone to sit under the stairs during board meetings", explains the company secretary, "someone with his wits about him. We sometimes need to know the hcf of two numbers: when I shout a pair of numbers down the stairs, you will shout the answer up as soon as you've worked it out. Our chairman is an impatient man, and you will be paid a speed bonus".

"Well", I might say, "I don't know about hcf's, but I need the money and I'll do the best I can".

So I am issued with Euclid's rule for computing the hcf; I also have the sense to demand a supply of index cards and a pencil. Whenever I apply my rule to a given pair of numbers, I enter this pair together with the result on an index card and add it to a growing pile of tabulated answers. Whenever I am asked for the hcf of a new pair, I first look through this pile, from top to bottom: if I find it I merely read out the answer, replacing this card one place higher in the pile, so that I can find it a little faster next time. Through this promotion process, rarely-occurring problems will gravitate towards the bottom and frequently occurring problems towards the top. If I do not find the required number-pair in the pile, then and only then do I have recourse to the rule. Having found the answer, in this case by calculation, I enter the result on a new card to be added to the pile. Supposing that there is a limit to the size of the pile of cards which I can handle, I discard the bottom card whenever there is need to shorten the pile.

Recursively Defined Functions

We will now re-interpret this scheme in the context of digital computing, taking the factorial function. This has the advantage that it can be defined in a recursive manner with particular ease and clarity, as will appear. I shall suppose that a "card index" regime for the evaluation of functions is available, so that we have facilities for converting functions into "memo functions"[2]. Because this can only be done at all easily using an open-ended programming language free of arbitrary constraints I shall conduct my illustration in POP-2 (ref. 3) which has the further advantage that the needed facility is actually available in the form of half a dozen POP-2 library routines[4].

We define the rule part of factorial as follows:

```
function fact n;

    if n < 0 or if not (n. isinteger) then undef else
    if n = 0 then 1 else n * fact (n–1) close
end
```

In words—"if n is negative or is not an integer then its factorial is undefined; if n is zero then its factorial is equal to 1; otherwise it is equal to n times the factorial of n–1".

To endow *fact* with the "memo" facility, using Popplestone's routines, we merely write

```
newmemo (fact, 100, nonop =) → fact;
```

This replaces the old definition of *fact* with a new one with a memo apparatus attached, such that the rote has an upper fixed limit of 100 entries and uses the "=" relation for look-up purposes. The symbol *nonop* warns the machine not to try to operate the "=" function at this stage but merely to note it for future reference.

Suppose now that the first call of the function is *fact* (3). The rote is inspected, but no entry for 3 is found there. So the rule is invoked. Is $n = 0$? No. The answer then, according to the rule is $3 \times fact$ (2). We now set out to evaluate *fact* (2). First we inspect the rote, again without success. Then we enter the rule and find that *fact* (2) is equal to $2 \times fact$ (1). The process of recursion continues until we encounter *fact* (0), which we find, by rule, is equal to 1. We can now evaluate *fact* $(1) = 1 \times fact$ (0), and hence *fact* $(2) = 2 \times fact$ (1) and hence *fact* $(3) = 3 \times fact$ (2). Each of these evaluations adds a new entry to the rote, which on exit at the end of the recursion looks like this

Argument	Value
3	6
2	2
1	1
0	1

Suppose that the next call of the function is *fact* (2). The answer is immediately found from the rote, being the second value encountered in a top-to-bottom search. On exiting, the "successful" entry is promoted one rung up the ladder (compare Samuel's "refreshing" of checker board positions stored on his dictionary tape), so that the rote now looks like this

Argument	Value
2	2
3	6
1	1
0	1

Let the next call now be *fact* (5). Is 5 in the rote ? No. Enter the rule. Is $5 = 0$? No. Then *fact* $(5) = 5 \times fact$ (4). Is 4 in the rote ? No. Enter the rule. Is $4 = 0$? No. Then *fact* $(4) = 4 \times fact$ (3). Is 3 in the rote ? Yes! *fact* $(3) = 6$ (promote this entry), so *fact* $(4) = 4 \times 6 = 24$ (make a rote entry), so *fact* $(5) = 5 \times 24 = 120$ (make a rote entry), exit. The rote now looks like this, momentarily restored, as it happens, to numerical order.

Argument	Value
5	120
4	24
3	6
2	2
1	1
0	1

Use as a Learning Mechanism

Viewed simply as speed-up aids for the evaluation of numerical functions, memo functions promise practical utility. In preliminary tests, Popplestone[4] found that a program had "learned" to evaluate a standard numerical function 10–20 times faster than normal after 200 successive calls. But the use of memo functions can be extended beyond this limited aim, so as to confer powers of learning by generalization, and of inductive reasoning. To clear the ground for these topics I shall describe an experimental study in machine learning on which I have been engaged for a number of years[5,6]. I shall then discuss how the rote-learning basis of this work can be fitted into the memo function scheme, and how memo functions can then be used to extend this basis by provision of facilities for generalization and induction.

The rote-learning algorithm is known as "boxes", and its computer program embodiment is intended to cope with adaptive control situations of the "black box" variety—that is, situations in which the physical parameters and laws of motion of the controllable system are unknown *a priori*. The program runs in an Elliott 4100 computer, connected by a high-speed link to a *PDP*-7 computer. The *PDP*-7's task is to simulate some unstable system specified by the programmer, to display a cinematic representation of this system on the cathode ray display panel and to send regular signals over the link to the Elliott 4100, these signals encoding successive state vectors of the unstable system during its evolution through time. At any instant a "fail" signal may be transmitted in place of the state vector, indicating that the simulated system has crashed. The "boxes" program running in the Elliott computer is provided with no information concerning the system being simulated at the other end of the link, apart from the number of elements of the state vector and their ranges. Its task is to receive, at each interval of time, the latest state signal from the link and to transmit a signal in return, corresponding to the selection of one out of a repertoire of control actions. Its "aim" is to learn to respond in such a way as to maximize the length of time elapsing between failure signals.

In all our experimental tests, the *PDP*-7 program has simulated a Donaldson system[7]—that is a motor-driven cart running on a track of fixed length balancing a pole so hinged at the base as to be free to fall in the plane of the track (see Fig. 1). Control is in our case by means of the switch with only two settings, "left" or "right", so that the motor operates continually at full force, changing only in sign. Failure is registered if any one of the four variables—x, the position of the cart on the track, $\dot{x}$, the velocity of the cart, θ, the angle of the pole, and $\dot{\theta}$, the rate of change of the pole's angle—goes outside pre-set bounds. In our implementation the "boxes" program running in the Elliott 4100 computer receives over the link every 1/15 s a 4-tuple of numbers x, $\dot{x}$, θ, $\dot{\theta}$, and must reply either with the number 1 (left) or 2 (right). From time to time, in place of a fresh 4-tuple, a "fail" signal arrives. From this scanty information the program must, by trial and error, construct a control function which maps state signals on to control signals so as to make the occurrence of failure states as infrequent as possible. The task is approached from a condition of complete initial ignorance about the interpretation of both state signals and control signals.

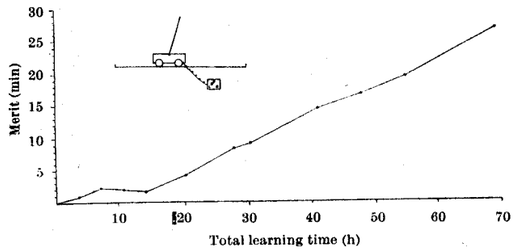

Fig. 1. The pole-and-cart system, set up for pure trial-and-error learning. The lower curve shows a smoothed average ("merit") of the time-until-crash, plotted against the total accumulated learning time.

The learning program is thus in the same situation as pre-scientific man, with a fixed repertoire of acts (control signals), a growing repertoire of experiences (state signals) and a basic assumption that there exist some unknown "laws of nature" which enable predictions to be made and strategies for survival to be devised. Progress in such a situation proceeds via the construction of a succession of models (partial descriptions) of the environment, proceeding from the crudest by successive refinement. If a given model can be used for prediction we call it a theory.

What is the most primitive pre-scientific model of all ? It is the one on which the "boxes" algorithm is based. In this model, experience of the environment is described by a rough categorization of properties—hot, warm, cool, cold; big, middle-sized, small; fast, slow, stationary; dry, moist, damp, wet; and so on. This categorization is then used to subdivide the totality of experience into a relatively small number of discrete situations. These recur in experience, so that we can attach to each separate situation a growing body of "lore", summarizing the average consequences in the past of performing this or that action in this particular situation. If the accumulating stocks of lore are now used to determine a strategy for selecting actions in future, with no attempt to generalize, then we have the essence of the "boxes" algorithm for adaptive control. Its application to the problem of evolving a cart-and-pole control strategy under "black box" conditions proceeds by reducing the totality of all possible x, $\dot{x}$, θ, $\dot{\theta}$ 4-tuples to a mere 162 discrete states defined by the following categorization

Position: left end, middle, right end;
Velocity: leftwards, roughly stationary, rightwards;
Angle of lean: extreme left, moderate left, slight left, slight right, moderate right, extreme right;
Rate of swing: leftwards, roughly stationary, rightwards.

In the "boxes" algorithm a computing process is associated with each discrete region of state-space, which has the task of accumulating by experiment a corpus of statistical lore about the expected consequences of taking, in that region, the two alternative actions. Fig. 1 shows a learning run, from which it can be concluded that even such a primitive model can be a first basis for enabling an automaton to learn what is—under "black box" constraints—a non-trivial task. We shall now step outside this primitive framework and see whether the memo function apparatus can encompass the needed extensions from pure rote-learning to learning with generalization and theory-formation.

Generalization by Approximation

The most unsatisfactory feature of the boxes algorithm described is that the discrete situations into which the total state-space is resolved are defined by an arbitrary choice of threshold settings: where is "moderate" to end and "extreme" to begin ? Further, there is no program control over these settings, so that certain obvious and attractive forms of generalization—such as the "lumping" and "splitting" procedures described elsewhere[6]—can only be tackled by *ad hoc* extensions of the program. The key to resolving state-space into discrete regions, as has been pointed out by Popplestone, lies in the function *equiv* which is left to the user to define for himself. *equiv* is used when searching the rote, to decide for each entry in turn whether it is or is not sufficiently close to the argument which we are looking for to be classified as "the same". In the case of "factorial" described earlier, *equiv* is no more than the relation of equality between integers, but for a function of real arguments, like "harmonic mean", say, it could define the number of decimal places of approximation. In this last case the reach of *equiv* could conveniently be extended, in recognition of the fact that *harm mean* $(x, y) = harm\ mean\ (y, x)$, so as to treat all x, y pairs as "the same" as their y, x opposite numbers. An example from the pole-balancing case is that it might be decreed, or discovered, that the 5-tuple x, $\dot{x}$, θ, $\dot{\theta}$, A, should be treated as equivalent to $-x$, $-\dot{x}$, $-\theta$, $-\dot{\theta}$, $-A$, so as to exploit the situation's essential symmetry (A here stands for a control action). Once we see the definition of *equiv* as modifiable by program, processes such as "lumping" and "splitting" can be taken in our stride—at the simplest level by varying in the light of experience the "tolerance" of the measure for recognizing the recurrence of "the same" event as one which has occurred before. I use the term tolerance deliberately to direct attention to a similarity to Zeeman's[8] notion of "tolerance embedding" which, however, I do not propose to pursue here.

Recasting the primitive "boxes" scheme in the new mould, we have predictor functions *pred*1 and *pred*2 corresponding to the two alternative actions "left" and "right". A predictor function takes a state-action pair as argument and produces a description of future state as a result. We also have a function called *strategy* which maps from predictions to actions. It is clearly appropriate to make *pred*1 and *pred*2 into memo functions. At the

elementary level of the current "boxes" implementation, these functions predict no more than the expected lapse of time before the next crash. In the present context we are chiefly concerned with the use of *equiv* in looking up a state-description on the rote and thus providing for approximation in state-space as a crude form of generalization. If we make the definition of *equiv* program-modifiable in the light of emergent properties of the rote, then we can ensure that states (arguments) not worth distinguishing are lumped into a single rote entry, whereas those presenting difficulties of prediction are given a finer-grained representation. For example, we can have *equiv* continuously up-dated in such a way that the result-values entered against adjacent values of those arguments which gain entry to the rote are always separated by more than a specified minimum. A thoroughgoing "lumping and splitting" facility is in this way automatically created.

Generalization by Interpolation

To see that approximation as described represents a primitive kind of generalization, we can relate it to interpolation. Figs. 2 and 3 depict the same set of data points. In the first case the gaps are partially filled by approximation. The lengths of the horizontal lines are set by the *equiv* function, which decrees which values of the x-variable are to be regarded as equivalent to each other for look-up purposes. In the second case a polynomial is fitted to the points. The question arises as to whether in some cases a list of polynomial coefficients would not be a more powerful and condensed means of

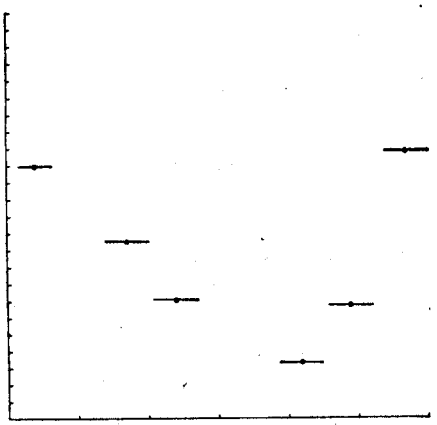

Fig. 2. An x–y plot in which y-values are obtained by look-up when the x-value falls within the vicinity of an existing point, as defined by the horizontal lines. For an x-value falling outside these intervals, a new point is computed by rule and added to the plot.

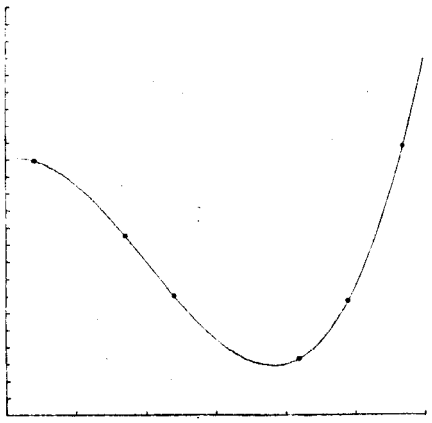

Fig. 3. The same points as in Fig. 2, with the gaps bridged by polynomial interpolation (see text).

providing for quick and approximate evaluation than a rote; also whether combinations of rote and coefficient-list could be used. Just as in the rote-learning case the rote is up-dated by simple accretion of new x–y pairs, so in the case of polynomial interpolation the coefficient list can be up-dated (and the polynomial thus revised) to accommodate each newly added x–y pair. In this way a process of induction on the growing store of examples is carried out, as demonstrated and discussed elsewhere by Fredkin[9], Pivar and Finkelstein[10], and by Popplestone[4].

I thank the Science Research Council for their generous support for this work.

Received March 12, 1968.

[1] Samuel, A. L., *IBM J. Res. Dev.*, **3**, 210 (1959).
[2] Michie, D., *Research Memorandum MIP-R-29*, *Dept. Machine Intelligence and Perception* (Edinburgh Univ., 1968).
[3] Burstall, R. M., and Popplestone, R. J., *Machine Intelligence* 2 (edit. by Dale, E., and Michie, D.), 207 (Oliver and Boyd, Edinburgh, 1968); *POP-2 Papers* (Oliver and Boyd, Edinburgh, 1968).
[4] Popplestone, R. J., *Research Memorandum MIP-R-30*, *Dept. Machine Intelligence and Perception* (Edinburgh Univ., 1968).
[5] Michie, D., and Chambers, R. A., *Machine Intelligence* 2 (edit. by Dale, E., and Michie, D.), 137 (Oliver and Boyd, Edinburgh, 1968).
[6] Michie, D., and Chambers, R. A., *Experimental Programming Reports: No. 14*, *Dept. Machine Intelligence and Perception* (Edinburgh Univ., 1968); in *Towards a Theoretical Biology* (edit. by Waddington, C. H.), **1** (Edinburgh University Press) (in the press).
[7] Donaldson, P. E. K., *Proc. Third Intern. Conf. Medical Electronics*, 173 (1960). Eastwood, E., *Proc. I.E.E.*, **115**, 203 (1968).
[8] Zeeman, E. C., and Buneman, O. P., *Towards a Theoretical Biology* (edit. by Waddington, C. H.), **1**, 140 (Edinburgh University Press, 1968).
[9] Fredkin, E., in *The Programming Language LISP: Its Operation and Applications* (edit. by Berkeley, E. C., and Bobrow, D. G.) (MIT Press, 1964).
[10] Pivar, M., and Finkelstein, M., in *The Programming Language LISP: Its Operation and Applications* (edit. by Berkeley, E. C., and Bobrow, D. G.) (MIT Press, 1964).

Printed in Great Britain by Fisher, Knight & Co., Ltd., St. Albans.