Accepted for publication in the Proceedings of the Conference on Emergent Computation, Los Alamos, NM, May 22-26, 1989, which will appear as a special issue of Physica D.

Perceptron Redux[*]: Emergence of Structure

Stewart W. Wilson
The Rowland Institute for Science
Cambridge, MA 02142
(wilson@think.com)

Abstract

Perceptrons were evolved that computed a rather difficult nonlinear Boolean function. The results with this early and basic form of emergent computation suggested that when genetic search is applied to its structure, a perceptron can learn more complex tasks than is sometimes supposed. The results also suggested, in the light of related work on classifier systems, that to hasten the *emergence* of an emergent computation it is desirable to provide evaluative feedback at a level as close as possible to that of the constituent local computations.

Keywords: emergent computation, perceptron, genetic algorithm, learning, credit assignment

* "redux…2. Brought back, restored…" *Oxford English Dictionary.*

1. Introduction

This paper investigates the application of genetic search to the generation of effective emergent computation. We shall say that a system embodies an emergent computation if it has global information processing behavior that arises from the interaction of many small, local computations. The emphasis is on "local": no aspect of the system sees more than a fraction of the rest of the system or the environment, yet significant overall behavior results. Within this definition, many variations are possible, from cases where the local computations and their interaction are linear, e.g., the simplest linear associator, to human society, where both the component individuals and their interactions are highly nonlinear. Our focus will be on the three-layer perceptron [1-2], an early and basic example of emergent computation in which a global decision is reached by weighing and combining the results of many small computations each looking at only a part of the input (Figure 1). Note that while the decision computation receives the results of the others, it, too, is local and limited in that it sees none of the system's environment. Related to the perceptron is the Pandemonium model [3], in which lower-level "demons" shout their computational results to higher-level demons. We shall apply genetic search to the problem of evolving perceptron structure, a process that itself forms an emergent computation.

In Minsky and Papert's [2] terminology, the perceptron computes a global predicate, i.e., a truth function with respect to the bits of its input. They showed that certain global predicates such as connectedness and parity could not be computed by a perceptron whose local computations, or *partial predicates*, were of restricted order, i.e., were based on fewer than the total number of input bits. However, there remain, as Minsky and Papert also showed, many useful and sometimes surprising global predicates that can be computed by order-restricted perceptrons.

The perceptron's attractiveness as a computing device is further enhanced by the *perceptron convergence theorem*, which states that if for a given perceptron there exists a choice of weights on its partial predicate outputs that would permit it to compute a certain global predicate correctly, then a weight adjustment procedure called the *perceptron learning algorithm* (PLA) will bring the system to perfect performance in a finite number of input trials. The drawback, of course, is that, for a given task, no such set of weights may exist, and therefore the PLA will be of no use. The existence of appropriate weights is itself a function of the perceptron's partial predicates—that is, do the local computations or experiments on the input sufficiently recode it so that a linear weighting scheme can compute the global predicate? While the PLA could modify the partial predicates' output weights, at the time of *Perceptrons* no scheme was known for modifying the predicates themselves. One proposal was to modify weights placed *within* the partial predicates, but no technique for this was widely available until the PDP group [4] introduced back-propagation.

In this paper we investigate a quite different approach to the predicate modification problem, namely, genetic search over the space of possible perceptron structures. In proposing that the demons might "mutate" and undergo "fusion" to form new demons, Selfridge [3] made a related proposal for Pandemonium. Our approach will be to evolve a *population* of perceptrons under a version of the genetic algorithm [5] in the context of a specific task. A general characteristic of the GA is that partial solutions to a problem, called *schemata*, are distributed throughout the population, implicitly tested, and gradually combined to form the superior individuals that eventually emerge. Thus our approach has a two-fold emergence: (1) that of the GA in finding good structures through distributed testing and interaction of their constituent schemata; and (2) that of the perceptron computation itself.

Since the perceptron is a fundamental example of an artificial neural network, our investigation is related to several others in which network characteristics were evolved genetically. In some (e.g., [6-7]), the emphasis was on evolving connection weights. Others (e.g., [8-9]) addressed the question of evolving general network structure and are thus more related to the present work.

We shall offer some answers to the following questions: (1) Can genetic search over "perceptron space" indeed lead to higher performance? (2) Will the predicates in the evolved perceptrons have an interesting form? (3) How efficient is the evolutionary process in this case? (4) More generally, what do the results say for the generation of goal-oriented emergent computation? In the present example of evolution under the GA, the evaluated entity is an entire perceptron so that we are in effect designing by providing feedback solely at the top or system level. Such a regime puts "emergence" to the test, so we are interested in its efficiency relative to schemes in which evaluation is more local.

2. Chromosome representation

Genetic search takes place over a space of genotypes or chromosomes that encode relevant characteristics of the phenotypes or organisms whose worth or *fitness* is to be optimized. In the present problem, the phenotype is the perceptron itself, consisting of partial predicates, predicate output weights, and threshold decision unit. For the predicates (which he called *association cells*), Rosenblatt [1] generally used units that summed and thresholded inputs that could be excitatory or inhibitory—that is, positive or negative. We have simplified this somewhat by assuming that the inputs to the perceptron's input layer are binary variables, and that each predicate computes the logical conjunct (AND) of some subset of the set of those variables and their complements (given the right predicates, such a perceptron can compute any Boolean function of its inputs). However, there is no reason why threshold units, or any other fixed computation should not also be investigated. We assume that when a predicate is True, its output is +1; when False, its output is -1. The weights multiplying these values are allowed to be any real numbers. The threshold decision unit sums the weighted predicate outputs and itself outputs a 1 (True) if the sum is greater than or equal to 0, otherwise 0 (False).

This specification of the phenotype permits a straightforward genotype encoding. Notice that all such perceptrons have identical decision units. Notice also that the weights are not inherent to the perceptron structure, but result from training in a certain problem environment. Thus only the number and structure of the predicates need be encoded. We encode a particular predicate using an ordered string in which "1" means the predicate unit is connected to the corresponding input bit, a "0" means connected but through an inverter (the bit is complemented), and "#" means that bit is not involved in the predicate's computation. Thus if there were six input bits in all, the string "01#10#" would encode a predicate that computed the conjunct $x'_0 x_1 x_3 x'_4$, where the primes signify negation. As another example, the predicate second from the top in Figure 1 would have the encoding "0##1###1#...#" and computes $x'_0 x_3 x_7$ (reading input bits from the top down, the circled tilde indicating an inverter). To encode a whole perceptron, we simply concatenate, in any order, the encodings of its predicates.

3. Evolution under the genetic algorithm

The major steps of the standard genetic algorithm are shown in Figure 2. The initial population P(0) usually consists of *N* fixed length chromosomes generated at random. In the Evaluation step, each chromosome is decoded to its phenotype and the phenotype is subjected to a procedure that determines or estimates its fitness with respect to the task or other problem environment at hand. The same fitness value is then attached to the underlying chromosome. In the Selection step, a new population of *N* members is formed by copying (reproducing) chromosomes by some procedure in which the fitter chromosomes receive more offspring. In the Genetic Operators step, some members of the new population exchange substrings (crossover) or undergo random changes at random string positions (mutation). Then the overall process iterates until the Stop condition is satisfied, which may occur after a predetermined number of generations, or after some condition within the population has occurred, etc. For details on the genetic algorithm, see Goldberg [10]. Our experiments followed most of this pattern, with parameters and variations to be described later.

The GA's evaluation step is the task-dependent part of the algorithm. In the present case, we wish to determine "how good" is the perceptron encoded by a given chromosome. To do so, we construct the phenotype and train it in the problem environment, using some measure of performance as the fitness (Figure 3). "Training" means presenting the perceptron with random input strings, obtaining its decision, then adjusting the weights according to the degree to which the decision matches the desired or correct decision for the global predicate that the trainer has in mind.

4. Task description

As a task or global predicate for the perceptrons to compute, we chose the six-bit Boolean multiplexer. This function is highly nonlinear, and has been used as a task with several different machine learning systems [11-16], providing a basis for comparing them. The six-multiplexer belongs to a family of progressively harder multiplexer functions, so that scale-up effects can be examined. In disjunctive normal form (DNF), the six-multiplexer is as follows:

$$F_6 = x'_0 x'_1 x_2 + x'_0 x_1 x_3 + x_0 x'_1 x_4 + x_0 x_1 x_5 \ .$$

Bits x_0 and x_1 can be thought of as forming an address that selects one of the remaining bits as the function's value. In general, there is a multiplexer function for strings of length $L = k + 2^k$, with $k > 0$.

5. Results

After a series of experiments incorporating several changes that produced improvements, we obtained results of the sort illustrated in Figure 4. The graph plots population average score versus the total number of chromosome evaluations. The population contained 100 chromosomes, so that 100 evaluations constituted one generation. The score or individual performance of a perceptron was its percentage of correct decisions on the last 40 of 100 training trials, and these individual scores were averaged over the population. The quantity plotted is the average of three runs of the experiment. By about 40 generations, the population performance was close to 100%, meaning that essentially all of the 100 chromosomes encoded perceptrons that, once trained, easily solved the problem. Note that the performance of the initial population was about 64%, indicating that training a perceptron with random predicates produced better than random performance, but clearly did not solve the problem.

Besides performance, it was interesting to know what kinds of predicates were being evolved. Figure 5 shows a typical set of predicates from one of the chromosomes at the end of a run. For purposes of illustration, the predicates have been sorted into two groups. The upper group consists of predicates that turn out to correspond exactly to the DNF terms of F_6 and its complement. The lower group contains the remaining predicates from the chromosome. With each predicate is its weight after training. Note how strong are the weights of the upper group vs. the lower—the latter predicates appear to be “chaff” resulting from the GA’s stochastic nature. Ignoring the chaff and the duplicates in the upper group, it can be seen that the system discovered the most efficient possible solution in the sense that (1) when a predicate is True, the global predicate is either always True or always False, and (2) the predicates form a minimal non-overlapping cover of the input space.

One of the important experimental changes that helped lead to these results was to make the chromosome longer than needed. A chromosome eight predicates long has just enough room for the eight predicates corresponding to the DNF terms of F_6 and its complement. In fact, we used a chromosome that had room for 16 predicates, as is clear from Figure 5. With an eight-predicate chromosome, the system typically did not evolve more than about four of the DNF predicates before converging, and performance was lower. Of course, in an arbitrary environment, one would not so conveniently know in advance how long to make the chromosome, but a more sophisticated system might employ chromosomes of variable length, along the lines of Smith’s [17] work, so that the proper length could be found adaptively. But we are not sure just why extra length is needed. There was some informal evidence that evolving predicates compete for chromosomal “slots” during the early part of a run. With fewer slots, it is more likely that a proto-predicate that loses such a competition would later be unable to establish itself elsewhere. The effect deserves greater study, and may be related to Holland’s [5] discussion of gene variation through intra-chromosomal duplication.

A second improvement in basic GA technique was to use so-called *reduced surrogate* crossover [18]. Instead of picking the crossover point at random, this technique picks at random but only among sites where the offspring are guaranteed to be different in at least one position from the parents. The result was to raise performance in the middle part of the evolution, so that the final level of performance was arrived at more quickly. The technique had no effect in the early stages, where totally random crossover is almost sure to be fruitful. It also did not raise final performance over that of standard crossover.

Two improvements resulted from changes in the evaluation procedure. In the first place, we were unsure just how to base *fitness* on a perceptron's performance during training. Initially, the fitness was the same as the performance measure: the average score on the last 40 of 100 training trials. It seemed reasonable to let the creature learn for a while before grading it. Eventually, however, we discovered that better results came from setting fitness equal to the average score over the entire training period, or all 100 trials. Perceptrons with superior predicates probably learn faster, and averaging from the very beginning may help detect them.

The largest single improvement came from changing the training algorithm from PLA to the Widrow-Hoff algorithm [19]. The difference was to move final performance levels from approximately 95% to 100%. In PLA, no weight adjustment occurs if the perceptron is correct, whereas Widrow-Hoff adjusts whenever the output of the decision unit summer differs from either +1 or -1, and the adjustment is such as to make the error zero. Because W-H is essentially always active, it may in some cases cause faster learning than PLA [20]. We did not observe this, but did observe the above final performance increase. Hinton [21] suggests that W-H yields weight settings closer to optimal when the predicates are suboptimal. That would produce more accurate fitnesses in our case, and may explain the improvement.

Several more mundane aspects of the experiments should be included for completeness. The chromosomes were not encoded directly in the ternary {1,0,#} vocabulary described in Section 2, but were instead pure binary, 11 standing for the 1, 00 standing for 0, and both 01 and 10 standing for #. This was based on the idea that binary encodings maximize the number of schemata processed by the GA [10]; empirical observations of the superiority of binary over ternary encodings were made by Schaffer [22], among others. The crossover rate in our experiments was 0.8, meaning that pairs of offspring were recombined with that probability. The mutation rate was 0.005 per allele (bit), the rate that gave the best results in an experiment with crossover turned off. The selection method used was a version of Wetzel's *tournament selection* [10] as modified by David Goldberg and the author. Tournament selection has a relatively simple implementation, and automatically eliminates problems of fitness scaling that can occur early and late in a run under more traditional selection methods. No direct comparison was made with traditional stochastic selection and its variants, but a version of Whitley and Kauth's [23] GENITOR algorithm was found to give performance similar to tournament selection along with somewhat less interesting final predicates.

6. Discussion

The basic impression gained from these experiments is that excellent performance and internal representation were achieved on a rather difficult problem, but the process was very lengthy. If, in Figure 3, we take 40 generations as the point at which a criterion performance was reached, then this required (40 generations) x (100 evaluations per generation) x (100 trials per evaluation) = 400,000 input trials. In comparison, *classifier systems* have been shown to solve the six-multiplexer problem much more quickly. A classifier system also employs genetic search, but the population consists of individual classifiers, somewhat analogous to a population of individual predicates. Results like the present ones have typically been achieved within approximately 5,000 input trials or fewer [13, 16] and have been extended to as few as 500 trials [15]. A neural network system using back-propagation [12] solved the six-multiplexer in approximately 5,000 input trials. Quinlan's [14] system learned the six-multiplexer rapidly, but that system cannot be described as emergent.

These comparisons are not quite fair to the present approach because the performance measure in Figure 4 is the population average performance, whereas high-performing individuals appeared in the population much earlier. In the runs of Figure 4, for example, the populations contained individuals scoring over 99% before 2,000 evaluations, or about twice as quickly as the population average. Such individuals, however, did not yet have the clear DNF representation of Figure 5.

The question of evolution times looms especially large in connection with scale-up to bigger problems. Among other things, a "bigger" problem usually means a longer input string, and the size of the input space grows exponentially with string length. Consequentially, an exponentially growing number of trials per evaluation are required in order to sample a fixed fraction of the input space. Grefenstette & Fitzpatrick [24] presented evidence that the GA can sample a relatively small fraction of possible inputs and still get good results. We experimented with this concept on the six-multiplexer, trying as few as 16 random trials for evaluating fitness. Results held up quite well for 64 and 32 trials, but fell off sharply for 16. Since there are 64 possible input strings for the six-multiplexer, this may indicate that one should not sample less than about half of the input space. The input space of the 11-multiplexer has 2048 members, so this would indicate sampling at least 1000 points, which in itself would mean learning times for the 11-multiplexer an order of magnitude longer than for the six-multiplexer. Very long times were indeed found in preliminary experiments with the 11-multiplexer. In the best experiment, a population average score of 87% required (80 generations) x (200 evaluations per generation) x (300 trials per evaluation) = 4,800,000 trials. By contrast, in unpublished work of the author, a classifier system reached a 90% score on the 11-multiplexer within 3,000 random trials. We do not know of any neural net investigation (using, e.g., back-propagation) of the 11-multiplexer.

7. Conclusion

Perceptrons competent to solve quite difficult problems appear to be evolvable by genetic search—suggesting greater versatility for the perceptron architecture than is sometimes thought—but the process is long and does not scale up well. A clue to the shortcomings may lie in the comparison with classifier systems. There, each classifier receives evaluation whereas in the present work a single evaluation applies to a whole perceptron, and the individual predicates making up the perceptron are evaluated only indirectly. Since a predicate corresponds logically to a classifier, this would suggest the hypothesis that evaluation at the whole-system level, for systems as large as these perceptrons, is simply too diffuse—and is therefore less efficient than—evaluation at the level of the system's principal components. Of course, there may be situations where a component interaction more complex than simple linear summation requires that the whole system be evaluated as a unit. An example may be recent work on a control problem [25] in which each population member is a set of rules. But in that case, too, the evolution times were long, on the order of one million trials.

An emergent system is one that produces a global result through interaction of strictly local computations. Given an interaction scheme, design of such a system means design of the local computations. Such a process is suitable for genetic search, as has been illustrated by the perceptron example, because the GA is good at altering and recombining pieces of solutions. An emergent system designed by a GA may in fact be one of the best approaches to maximizing performance in an incompletely understood environment. However, the present results plus those on classifier systems suggest broadly that the "design time" of an emergent system will be significantly improved if evaluation occurs not at the system level but at the level of the system's component computations. *Sic semper* credit assignment!

Acknowledgements

The author would like to thank David Goldberg and Stephanie Forrest for valuable conversations. The suggestions of the anonymous reviewers are also appreciated.

References

[1] F. Rosenblatt, The perceptron: a probabilistic model for information storage and organization in the brain, Psych. Rev. 65 (1958) 386.

[2] M. Minsky and S. Papert, Perceptrons: An Introduction to Computational Geometry (MIT Press, Cambridge, MA, 1969).

[3] O. G. Selfridge, Pandemonium: a paradigm for learning, in: Mechanisation of Thought Processes: Proceedings of a Symposium Held at the National Physical Laboratory (HMSO, London, 1958).

[4] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, Parallel Distributed Processing: Explorations in the Microstructure of Cognition (MIT Press, Cambridge, 1986).

[5] J. H. Holland, Adaptation in Natural and Artificial Systems (Univ. of Michigan Press, Ann Arbor, 1975).

[6] D. Whitley and T. Hanson, Optimizing neural networks using faster, more accurate genetic search, in: Proc. Third Internat. Conf. on Genetic Algorithms, J. D. Schaffer, ed. (Morgan

Kaufmann, San Mateo, CA, 1989).

[7] D. Montana and L. Davis, Training feedforward neural networks using genetic algorithms, in: Proc. Eleventh Internat. Joint Conference on Artificial Intelligence (1989).

[8] S. A. Harp, T. Samad, and A. Guha, Towards the genetic synthesis of neural networks, in: Proc. Third Internat. Conf. on Genetic Algorithms, J. D. Schaffer, ed. (Morgan Kaufmann, San Mateo, CA, 1989).

[9] G. F. Miller, P. M. Todd, and S. U. Hegde, Designing neural networks using genetic algorithms, in: Proc. Third Internat. Conf. on Genetic Algorithms, J. D. Schaffer, ed. (Morgan Kaufmann, San Mateo, CA, 1989).

[10] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning (Addison-Wesley, Reading, MA, 1989).

[11] A. G. Barto, Learning by statistical cooperation of self-interested neuron-like computing elements, Human Neurobiology 2 (1985) 229.

[12] C. W. Anderson, Learning and Problem Solving with Multilayer Connectionist Systems (Ph.D. Dissertation, Computer and Information Science, Univ. of Massachusetts, 1986).

[13] S. W. Wilson, Classifier systems and the animat problem, Machine Learning 2 (1987) 199.

[14] J. R. Quinlan, An empirical comparison of genetic and decision-tree classifiers, in: Proc. Fifth Intern. Conf. on Machine Learning (Morgan Kaufmann, San Mateo, CA, 1988).

[15] S. Sen, Classifier system learning of multiplexer function, Unpublished report available from Sandip Sen, Dept. of EECS, CSE Division, Univ. of Michigan, Ann Arbor, MI 48197.

[16] L. B. Booker, Triggered rule discovery in classifier systems, in: Proc. Third Internat. Conf. on Genetic Algorithms, J. D. Schaffer, ed. (Morgan Kaufmann, San Mateo, CA, 1989).

[17] S. Smith, A Learning System Based on Genetic Algorithms (Ph. D. Dissertation, Computer Science, Univ. of Pittsburgh, 1980).

[18] L. B. Booker, Improving search in genetic algorithms, in: Genetic Algorithms and Simulated Annealing, L. Davis, ed. (Pitman, London, 1987).

[19] B. Widrow and M. Hoff, Adaptive switching circuits, in: IRE WESCON Convention Record (IRE, New York, 1960).

[20] J. A. Anderson and E. Rosenfeld, Neurocomputing: Foundations of Research (MIT Press, Cambridge, MA, 1988).

[21] G. E. Hinton, Connectionist Learning Procedures, Technical Report CMU-CS-87-115 (Carnegie-Mellon Univ., Computer Science Dept., Pittsburgh, 1987).

[22] J. D. Schaffer, Learning multiclass pattern discrimination, in: Proc. First Intern. Conf. on Genetic Algorithms, J. Grefenstette, ed. (Lawrence Erlbaum Assoc., Hillsdale, NJ, 1985).

[23] D. Whitley and J. Kauth, GENITOR: a different genetic algorithm, in: Proc. Rocky Mountain Conf. on Artificial Intelligence (Denver, CO, 1988).

[24] J. J. Grefenstette and J. M. Fitzpatrick, Genetic search with approximate function evaluations, in: Proc. First Intern. Conf. on Genetic Algorithms, J. Grefenstette, ed. (Lawrence Erlbaum Assoc., Hillsdale, NJ, 1985).

[25] J. J. Grefenstette, A system for learning control strategies with genetic algorithms, in: Proc. Third Internat. Conf. on Genetic Algorithms, J. D. Schaffer, ed. (Morgan Kaufmann, San Mateo, CA, 1989).

[26] J. D. Schaffer and G. G. Grefenstette, A Critical Review of Genetic Algorithms, Report No. TR-88-009 (Philips Laboratories, Briarcliff Manor, NY, 1988).

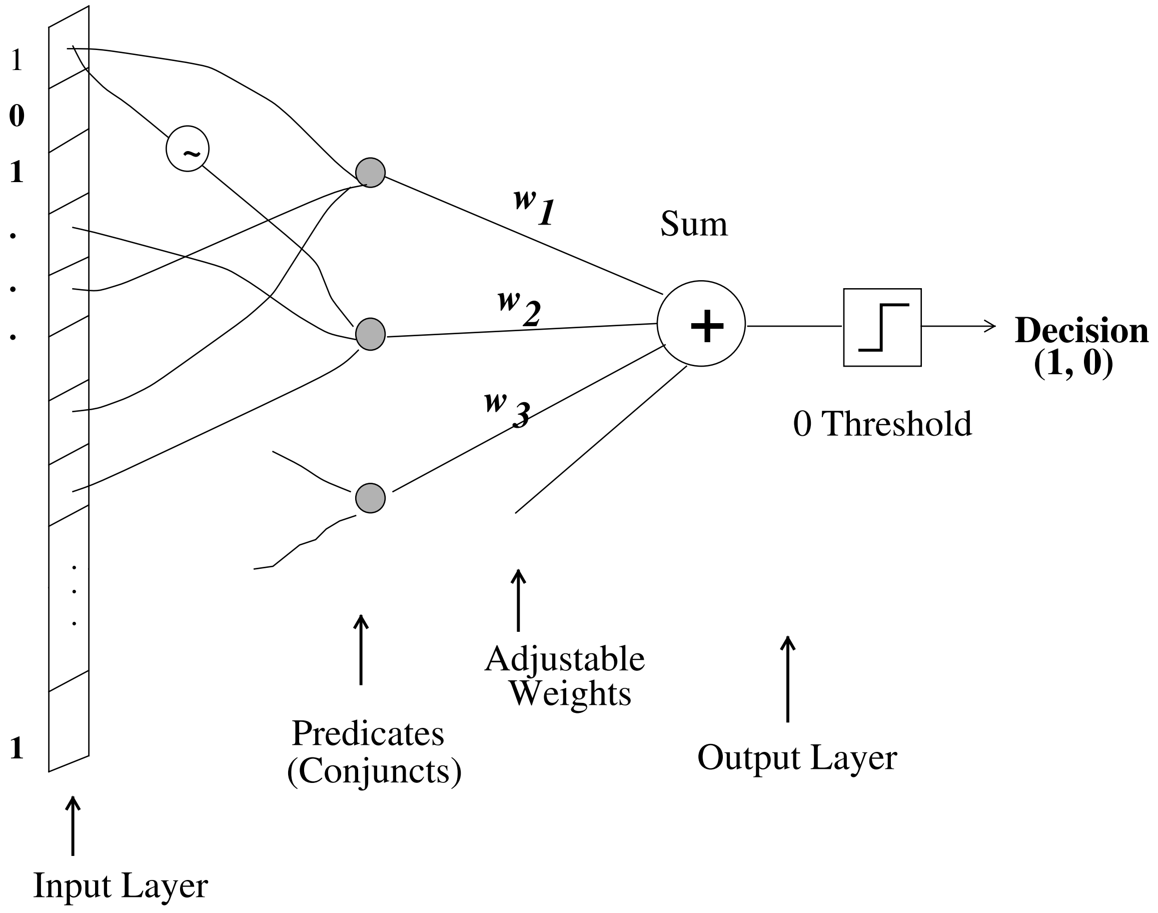

Figure 1. A 3-layer perceptron with binary input layer and predicates that compute the AND of subsets of the set of the inputs and their complements.

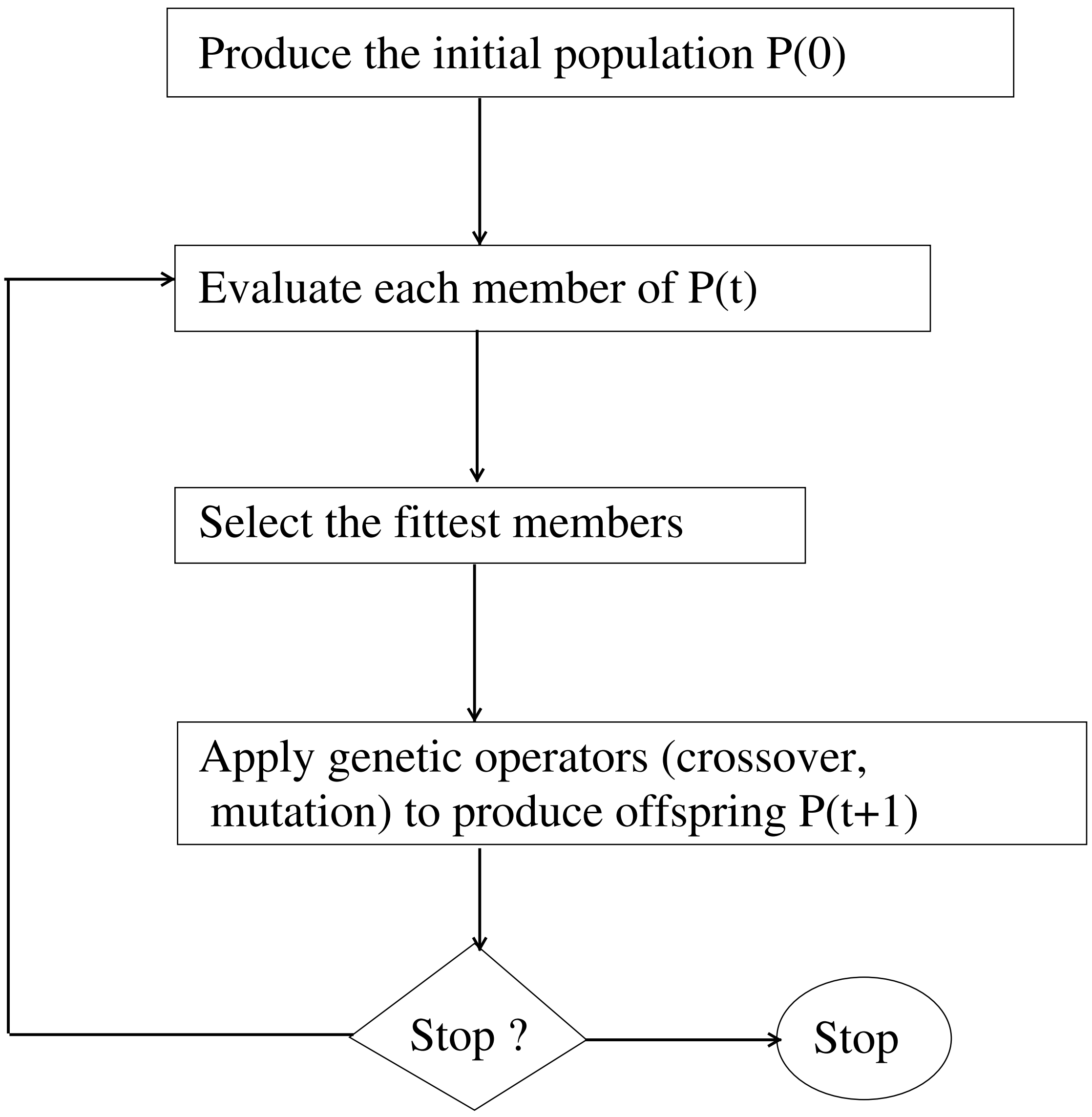
Produce the initial population P(0)
Evaluate each member of P(t)
Select the fittest members
Apply genetic operators (crossover, mutation) to produce offspring P(t+1)
Stop ?
Stop

Figure 2. A basic genetic algorithm (after a diagram in [26]).

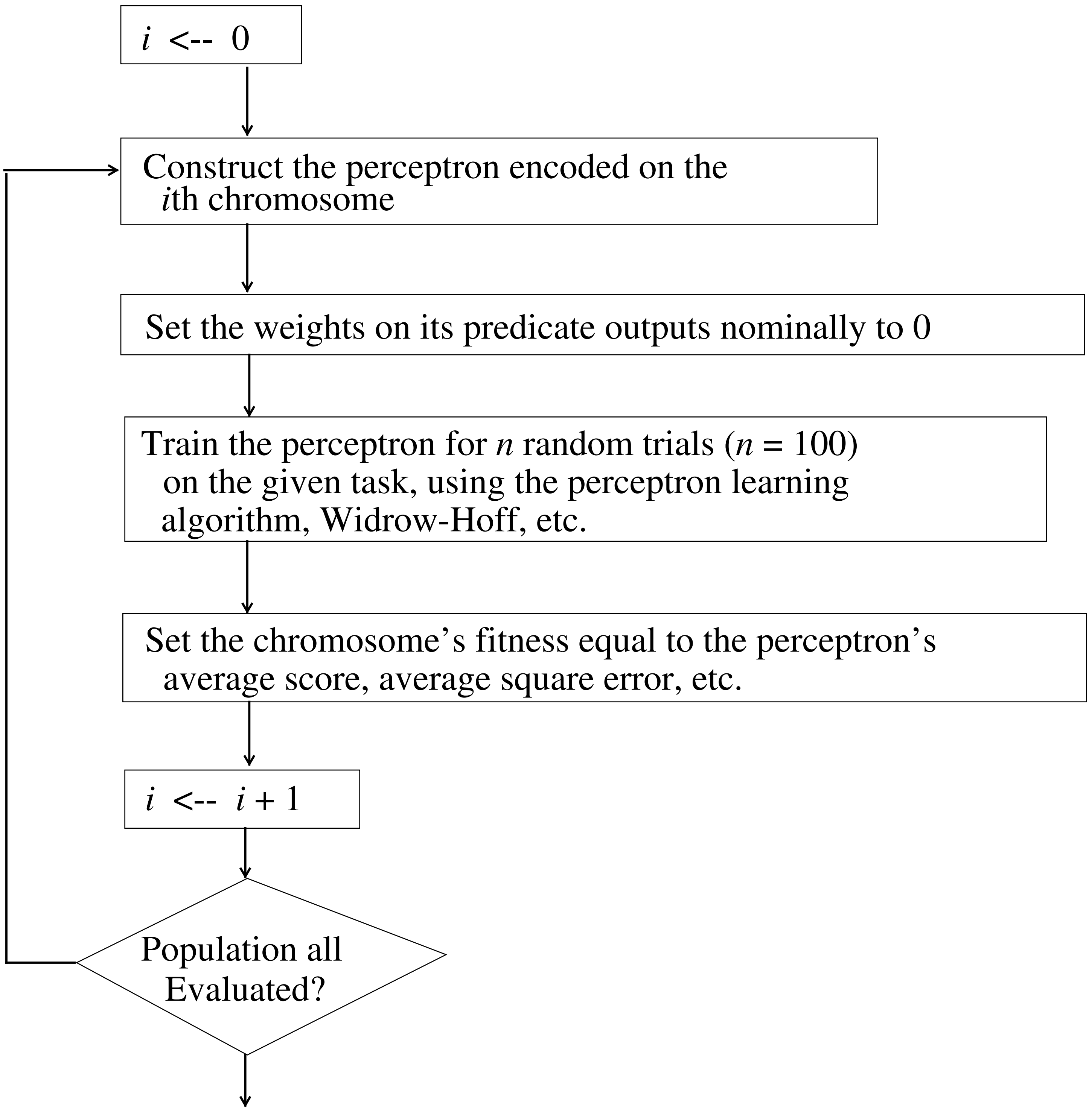
i <-- 0
Construct the perceptron encoded on the ith chromosome
Set the weights on its predicate outputs nominally to 0
Train the perceptron for n random trials (n = 100) on the given task, using the perceptron learning algorithm, Widrow-Hoff, etc.
Set the chromosome's fitness equal to the perceptron's average score, average square error, etc.
i <-- i + 1
Population all Evaluated?

Figure 3. Inside the GA's Evaluate box for the perceptron problem.

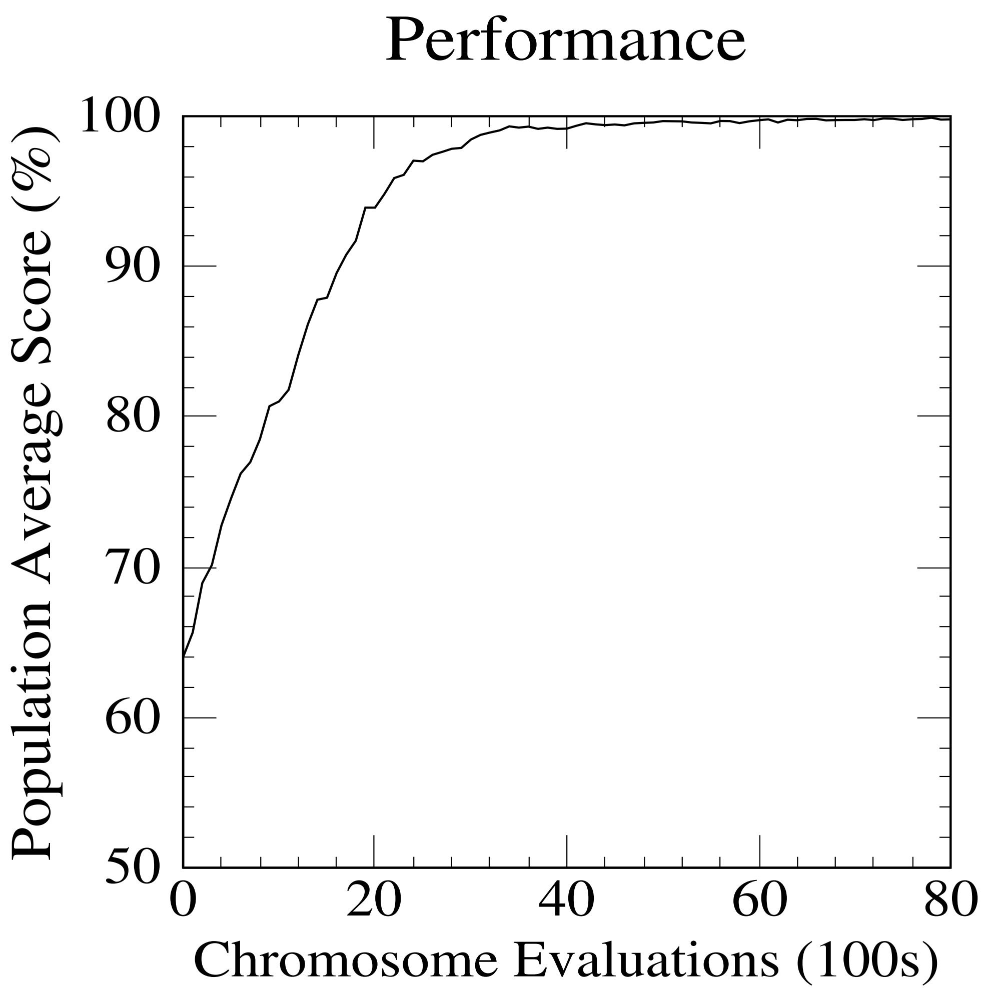

Figure 4. Population performance. Average over the population of the score of each chromosome's perceptron on the last 40 of 100 training trials *versus* number of chromosome evaluations. Curve averages over three runs, each starting with a different random initial population.

000###	-.244
000###	-.244
001###	+.269
001###	+.269
01#0##	-.424
01#1##	+.410
10##0#	-.441
10##1#	+.262
10##1#	+.262
11###0	-.399
11###1	+.487

Reflect DNF terms of F_6 and its complement.

0##0##	-.063
0111##	+.104
1####0	-.087
101###	-.003
11##10	-.084

Other predicates. Note small weights.

Figure 5. Evolved predicates (left column) from a chromosome in one of the populations of Figure 4. Weights (right column) are after training.