COMPLETE INDUCTION OF RECURRENT NEURAL NETWORKS

PETER J. ANGELINE
IBM Federal Systems Company, Rt 17C
Owego, New York 13827

GREGORY M. SAUNDERS and JORDAN B. POLLACK
Laboratory for Artificial Intelligence Research, The Ohio State University,
Columbus, Ohio 43210

ABSTRACT

It has occurred to many researchers to apply genetic algorithms to the training of recurrent neural networks. These studies generally avoid total network induction, i.e., inducing both the topology and parametric values of a network, favoring instead simple parametric learning. Also, they often rely on standard forms of crossover to manipulate network structures, a process which actually inhibits network evolution. In addition, the common commitment to bit string representations introduces an artificial limitation on the range of networks that can be created. In this paper, an evolutionary program, called GNARL, is presented that performs total network induction. GNARL's ability to induce a range of appropriate solutions is demonstrated on an interesting control task.

1. Introduction

Applying evolutionary computations to the training of neural networks has occurred to many researchers. Studies that use genetic algorithms[11] to evolve connectionist networks generally avoid total network induction, i.e., inducing both the topology and parametric values of a network, favoring instead simple parametric learning.[3,12,17] Genetic algorithm studies that do induce network topology allow only limited structural changes.[16,13] In addition, these studies rely on generic forms of crossover to manipulate network structure, a process which this paper argues actually inhibits network evolution.

Evolutionary programming,[7,5] an alternative evolutionary optimization technique, embodies principles that better respect the complexities of evolving neural networks. This paper presents GNARL, an evolutionary program that induces both the topology and parametric values for recurrent neural networks. GNARL is demonstrated on an interesting control task.

2. Evolving Networks with Genetic Algorithms

Genetic algorithms create new individuals by recombining the representational components of two population members, usually represented as bit strings which are interpreted as the representation to be evaluated. They therefore rely on two distinct representational spaces, one in which the search is performed, called the *search representation*, and one in which candidate solutions are evaluated, called the *evaluated representation*.[1,2] An *interpretation function* maps between elements of these distinct representational spaces.[1,2] For instance, when the task is to evolve connectionist networks, the interpretation function maps individuals from the chosen search representation into the

evaluated connectionist network. Clearly, the choice of interpretation function introduces a strong bias into the search, typically by excluding many potentially interesting and useful networks. The benefits of having such a dual representation hinge on supplying an interpretation function that makes crossover an appropriate evolutionary operator for the task. Otherwise, the need to translate between dual representations is an unnecessary complication.

Characterizing tasks for which crossover is beneficial is an open question. Crossover has been hypothesized to be most effective in tasks where the fitness of a member of the population is reasonably correlated with the expected ability of its representational components.[9] Environments where this is not true are called *deceptive*.[8] Deceptive problems are generally more difficult for a genetic algorithm to solve. Often, the interpretation function is specially crafted to reduce or remove the deceptiveness of a task. The central question for evolving connectionist networks is then does a suitable interpretation function exist that adequately compensates for the deception inherent in the task of evolving connectionist networks so that crossover is a good operator.

There are three forms of deception when using crossover to evolve connectionist networks. The first involves identical networks, i.e., ones that share both a common topology and common weights when evaluated. Because the interpretation function may be many-to-one, identical networks need not have the same search representation. In such a situation, crossover will tend to create offspring that contain repeated components, thus losing the computational ability of some of the parents' hidden units. Consequently, the resulting networks will tend to perform worse than their parents since they are not in possession of key computational components for the task. This has been called the *competing conventions problem*.[17] The second form of deception involves two networks with identical topologies but different weights. It is well known that for a given task, a single connectionist topology affords multiple solutions for a task, each implemented by a unique *distributed representation* spread across the hidden units.[10,18] The computational role each node plays in the overall representation of the task solution is determined purely by the presence and strengths of its interconnections. As a result, there need be no correlation between distinct distributed representations over a particular network architecture for a given task. This greatly reduces the possibility that an arbitrary crossover operation between distinct distributed representations will construct viable offspring *regardless of the interpretation function used*. Finally, deception can occur when the parents differ topologically. The types of distributed representations that can develop in a network vary widely with the number of hidden units and the network's connectivity. Thus, the distributed representations of topologically distinct networks have a greater chance of being incompatible parents.

In short, for crossover to be an appropriate operator for evolving networks, the interpretation function must somehow compensate for the various types of deception described above. The discussion above suggests that for general network induction the complexity of an appropriate interpretation function will more than rival the complexity of the original learning problem. Thus, the prospect of evolving connectionist networks with crossover goes against the current theory associated with both genetic algorithms and

connectionist networks, and better results should be expected with reproduction heuristics that respect the uniqueness of the independently developing distributed representations. This point has been tacitly validated in the genetic algorithms literature by a trend towards a reduced reliance on binary representations when evolving networks.[14,4] However, the juxtaposition of distributed representations via crossover is still commonplace. For complete network induction without unnecessarily restricting the search space of architectures, alternative operators that respect the idiosyncracies of the developing architectures are required.

3. Using an Evolutionary Program to Evolve Connectionist Networks

3.1 Advantages of Evolutionary Programming

In evolutionary programming (EP),[7,5] there is no dual representation scheme as in genetic algorithms. The actual representation evaluated by the fitness function is manipulated directly. Thus no interpretation function is required. Further, evolutionary programs generally avoid recombination of any form, choosing instead to manipulate structures by representation specific mutation operators. EP mutation operators are more complex than the weak form of mutation used in genetic algorithms and often are designed to respect the idiosyncracies of the chosen representation. This provides a significant advantage for EP methods since EP operators are supplied more task specific knowledge than typically used in standard genetic operators.[1]

By using informed representation specific operators and avoiding recombination, EP is a much better paradigm for evolving the weights and topology of recurrent connectionist networks. These qualities allow the developing distributed representations that are unique to each network in the population to be manipulated in a manner consistent with their nature. Previous EP studies that evolve connectionist networks[5,6] typically fall into the same traps of architecture restriction as genetic algorithms. In the next section, we describe GNARL, an evolutionary program that performs complete network induction.

3.2 The GNARL Algorithm

The GNARL (*GeNeralized Acquisition of Recurrent Links)* algorithm begins with an initial population of *n* randomly generated networks. The number of input nodes (*num-in*) and number of output nodes (*num-out*) are fixed for a given task; the number of hidden nodes as well as the connections among them are free to vary. Self-links as well as general recurrent loops are permitted.

In each generation of search, the networks are ranked by a user-supplied fitness function f: $\boldsymbol{N} \rightarrow \Re$. The flexibility of evolutionary methods allows a number of fitness metrics to be used. After ranking, the best $^{n}/_{2}$ individuals are saved and allowed to reproduce with mutations each generation. Reproduction entails modifying both the weights and topology of each parent network *N*. First, the *temperature* of a network *T(N)* is calculated:

$$T(N) = \text{'U}(0,1)\left(1 - \frac{f(N)}{f_{max}}\right) \qquad \text{Eq. (1)}$$

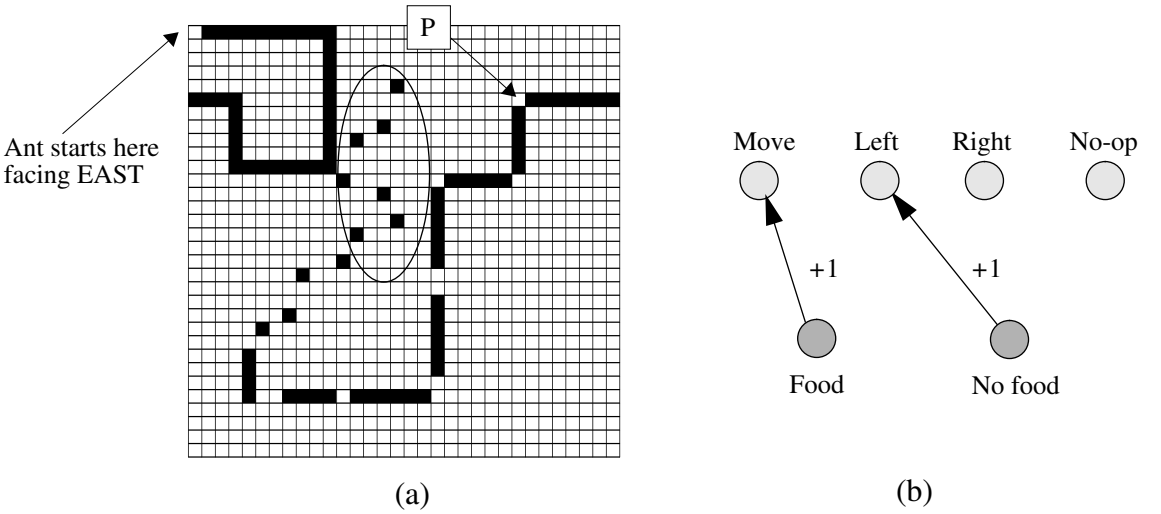

Figure 1: *The Tracker Task. (a) The trail is connected initially, but becomes progressively more difficult to follow. The underlying 2-d grid is toroidal, so that position "P" is the first break in the trail. The ellipse indicates the 7 pieces of food that the network of the second run failed to reach. (b) The semantics of the I/O units for the ant network. The first input node denotes the presence of food in the square directly in front of the ant; the second denotes the absence of food in this same square. No-op, from Jefferson, allows the network to stay in one position while activation flows through recurrent links. This particular network "eats" 42 pieces of food before spinning endlessly in place at position P, illustrating a very deep local minimum in the search space.*

where f_{max} (provided by the user) is the maximum possible fitness for a given task and U(0,1) is a uniform random variable over the interval (0,1). This measure of *N*'s performance is used to anneal both the structural and parametric similarity between parent and offspring according to the network's proximity to correct performance. Networks with a high temperature, and thus poor performance, are mutated more severely while those with a low temperature, and thus nearer to a solution, are mutated only slightly. This measure encourages a coarse-grained search initially and a finer-grained search as a network approaches a solution. The uniform random variable in the equation allows even a network that is far from optimal to be mutated less severely on occasion.

A parametric mutation perturbs each weight, *w*, in a network, *N*, with gaussian noise as follows:

$$w \leftarrow w + \text{Normal}\,(0, T(N)), \quad \forall w \in N \qquad \text{Eq. (2)}$$

Structural mutations modify the topology of *N* according to the following:

- add k_1 hidden nodes with probability $p_{add\text{-}node}$
- delete k_2 hidden nodes with probability $p_{delete\text{-}node}$
- add k_3 links with probability $p_{add\text{-}link}$
- delete k_4 links with probability $p_{delete\text{-}link}$

where each k_i is selected uniformly from a user-defined range, again annealed by *T(N)*. When a node is added, it is initialized without connections; when a node is deleted, all its incident links are removed. All new links are initialized to 0. Another paper[2] describes the GNARL algorithm in more detail.

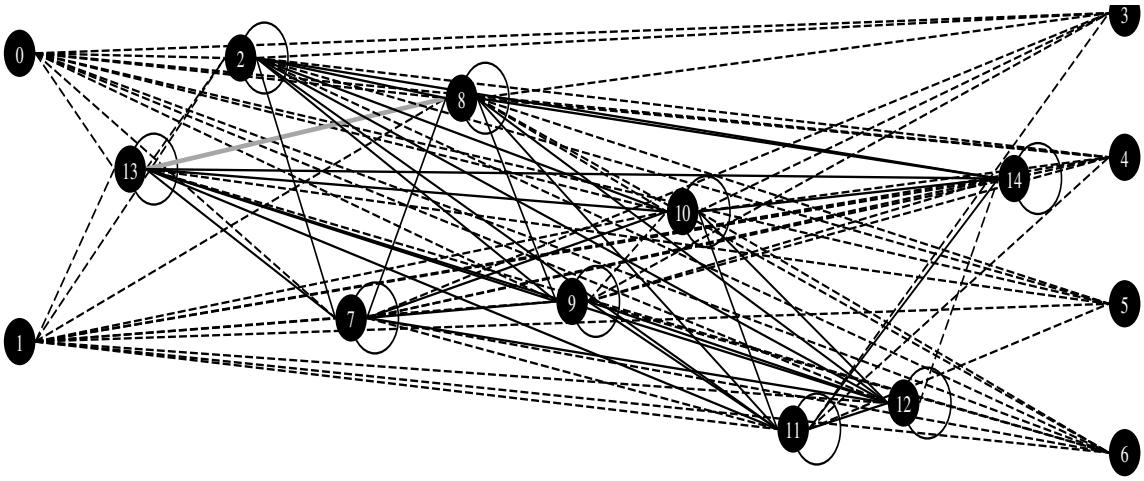

Figure 2: *The* Tracker Task*, first run. Network induced by GNARL after 2090 generations. Forward links are dashed; bidirectional links & loops are solid. The light gray connection between nodes 8 and 13 is the sole backlink. This network clears the trail in 319 epochs.*

3.3 Experiments with GNARL

GNARL was tested on a simple control task called the *Tracker* task.[12] In this problem, a simulated ant is placed on a two-dimensional toroidal grid and must maximize the number of food pieces it collects in a given time period (Figure 1a). Each ant is controlled by a recurrent network with two input nodes and four output nodes (Figure 1b). Following the original study,[12] input nodes signify either the presence or absence of food directly in front of the ant while output nodes signify the actions MOVE, turn LEFT, turn RIGHT, and NO-OP. At each step, the action whose corresponding output node has maximum activation is executed. Fitness of a network is the number of grid positions cleared within 200 time steps. A run was considered completed when a network cleared more than 80 of the 89 pieces of food on the grid. This experiment used a population size of 100 networks.

In the first run, GNARL created a network (Figure 2) that cleared 81 grid positions within the 200 time steps. To create this network, GNARL generated a total of 104,600 networks over 2090 generations. Figure 3 shows the state of the output units of the network over three different sets of inputs. Each point is a triple determined by the activations of the MOVE, LEFT, and RIGHT nodes of the network.* Figure 3a shows the result of supplying 200 "food" inputs to the network – a fixed point that always executes MOVE. Figure 3b shows the sequence of states reached when 200 "no food" signals are supplied to the network – a collection of points describing a limit cycle of length 5 that repeatedly executes the sequence RIGHT, RIGHT, RIGHT, RIGHT, MOVE. These two attractors determine the response of the network to the task (Figures 3c,d). The additional points in Figure 3c are transients encountered as the network alternates between these attractors.

However, not all evolved network behaviors are so simple.[15] In a second run, GNARL induced a network that cleared 82 grid points within the 200 time steps. In this run,

* NO-OP is not shown since it was never used in the final network.

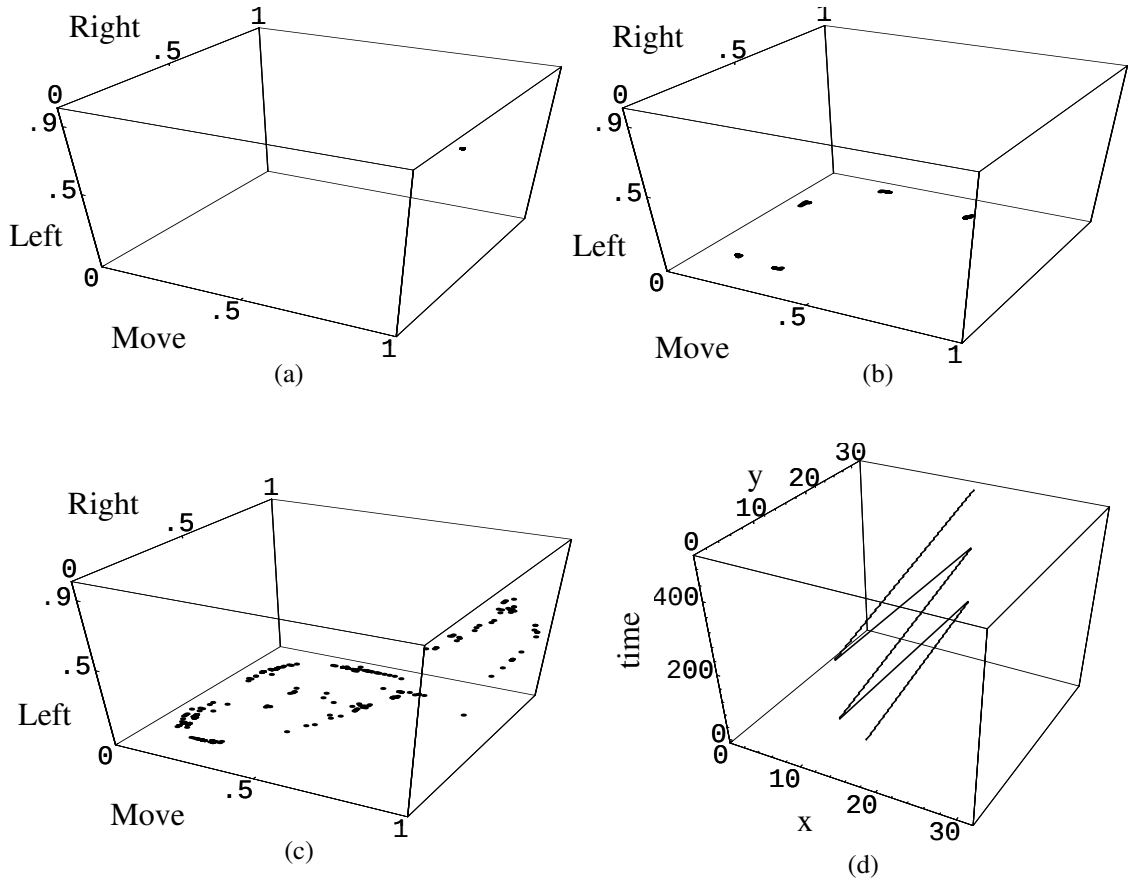

Figure 3: *Limit behavior of the network that clears the trail in 319 steps. Graphs show the state of the output units Move, Right, Left. (a) Fixed point attractor that results for sequence of 500 "food" signals; (b) Limit cycle attractor that results when a sequence of 500 "no food" signals is given to network; (c) All states visited while traversing the trail; (d) The path of the ant on an empty grid. The z axis represents time. Note that x is fixed, and y increases monotonically at a fixed rate. The large jumps in y position are artifacts of the toroidal grid.*

GNARL created 79,850 networks over the 1595 generations. Figure 4 illustrates the behavior of this network. Once again, the "food" attractor, shown in Figure 4a, is a single point in the space that always executes MOVE. The "no food" behavior, however, is not describable as an FSA; instead, it is a quasiperiodic trajectory of points shaped like a "D" in output space (Figure 4b). The placement of the "D" is in the MOVE / RIGHT corner of the space and encodes a complex alternation between these two operations (Figure 4d).

4. Discussion and Conclusions

In the original study, Jefferson et al.[12] use a genetic algorithm, and hence crossover, to evolve only the parameters of a recurrent neural network with five hidden units for the Tracker task. They report that 1,123,942 networks were generated to evolve a network of similar ability to the networks evolved by GNARL in the two runs. This is 10.74 and 14.07 times the number of recurrent individual networks generated respectively in the two

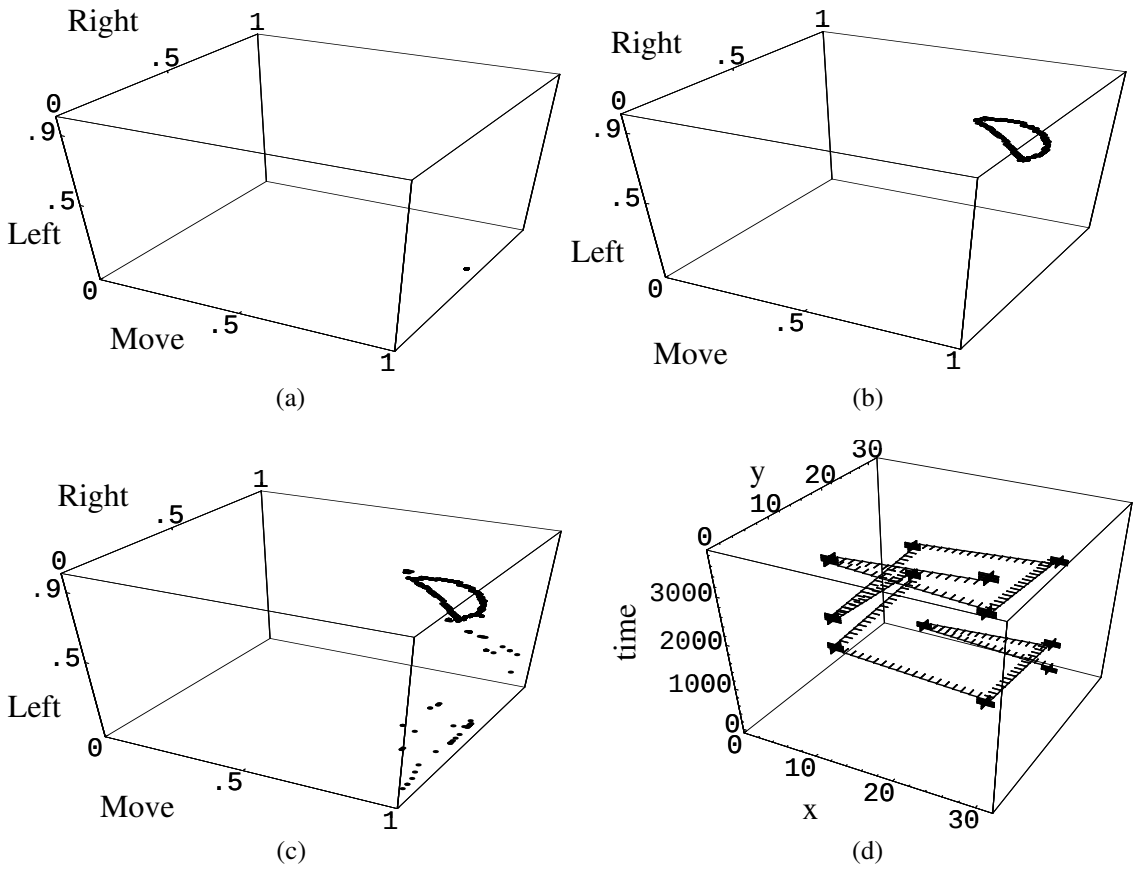

Figure 4: *Limit behavior of the network of the second run. Graphs show the state of the output units Move, Right, Left. (a) Fixed point attractor that results for sequence of 500 "food" signals; (b) Limit cycle attractor that results when a sequence of 500 "no food" signals is given to network; (c) All states visited while traversing the trail; (d) The x position of the ant over time when run on an empty grid.*

runs of GNARL. Thus GNARL constructed an order of magnitude fewer recurrent networks in both runs to solve the problem. The fact that GNARL was also manipulating the architecture of the recurrent network while the architecture was static in Jefferson et al.[12] makes this difference especially significant.

While it is tempting to use the above results to validate our claim of the inappropriateness of crossover for evolving networks, there may be other explanations. For instance, the synergy of manipulating both the parameters and topology of the network may also contribute to the quicker learning. Consider that a fixed network architecture, if chosen poorly, can inhibit the prompt evolution of a solution. Next, consider that in GNARL, architectures are indirectly selected for how quickly they improve over the task. This encourages the identification of architectures that are conducive to manipulation by the selected mutation operators, potentially leading to quicker learning. However, by the arguments in Section 2 of this paper, we feel that the mismatch between crossover and the network representation should be the most significant reason for the speed-up shown by GNARL.

The experiments above, along with additional experiments in a more complete study[2]

demonstrate GNARL's ability to evolve both the architecture and parameters for recurrent neural networks for a variety of tasks without unduly restricting the architectural search space.

5. Acknowledgments

This research was partially supported by ONR grant N00014-92-J-1195.

6. References

1. P. J. Angeline (1993). *Evolutionary algorithms and emergent intelligence*. Ph. D. thesis, The Ohio State University, Laboratory for Artificial Intelligence Research, Columbus Ohio.
2. P. J. Angeline, G. M. Saunders and J. B. Pollack (1993). *An evolutionary algorithm that constructs recurrent neural networks*. *IEEE Transactions on Neural Networks*, **5** (2).
3. R. K. Belew, J. McInerney, and N. N. Schraudolph (1990). Evolving networks: Using the genetic algorithm with connectionist learning. In *Artificial Life II*, C. Langton, C. Taylor, J. Farmer and S. Rasmussen (eds.), Reading, MA: Addison-Wesley Publishing Company, Inc., pp. 511-548.
4. R. Collins and D. Jefferson (1992). An artificial neural network representation for artificial organisms. In H. P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*. Springer-Verlag.
5. D. Fogel (1992). *Evolving Artificial Intelligence.* Ph. D. thesis, University of California, San Diego.
6. D. Fogel, L. Fogel and V. Porto (1990). Evolving neural networks. *Biological Cybernetics*, **63**, pp. 487-493.
7. L. Fogel, A. Owens and M. Walsh (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York.
8. D. Goldberg (1989a). Genetic algorithms and Walsh functions: Part 2, Deception and its analysis. *Complex Systems*, **3**, pp. 153–171.
9. D. Goldberg (1989b). Genetic algorithms and Walsh functions: Part 1, A gentle introduction. *Complex Systems*, **3**, pp. 129–152.
10. G. Hinton, J. McClelland and D. Rumelhart (1986). Distributed representations. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: Foundations. pp. 77–109. MIT Press, Cambridge, MA,
11. J. Holland (1975). *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, MI.
12. D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor and A. Wang (1991). Evolution as a theme in artificial life: The genesys/tracker system. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II: Proceedings of the Workshop on Artificial Life.* pages 549–577. Addison-Wesley.
13. N. Karunanithi, R. Das, and D. Whitley (1992). Genetic cascade learning for neural networks. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks.*
14. J. Koza and J. Rice (1991). Genetic generation of both the weights and architecture for a neural network. In *IEEE International Joint Conference on Neural Networks*, pages II-397 – II-404, Seattle, WA, IEEE Press.
15. J. B. Pollack (1991). The induction of dynamical recognizer. *Machine Learning*. **7**, pp. 227-252.
16. M. A. Potter (1992). A genetic cascade-correlation learning algorithm. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks.*
17. J. D. Schaffer, D. Whitley and L. J. Eshelman (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks.*
18. T. Sejnowski and C. Rosenberg (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, **1**, pp. 145–168.