Machine Learning, 22, 11–33 (1996)
© 1996 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Efficient Reinforcement Learning through Symbiotic Evolution

DAVID E. MORIARTY AND RISTO MIIKKULAINEN moriarty,risto@cs.utexas.edu
Department of Computer Sciences, The University of Texas at Austin. Austin, TX 78712

Received October 20, 1994 ; Revised October 6, 1995

Editor: Leslie Pack Kaelbling

Abstract. This article presents a new reinforcement learning method called SANE (Symbiotic, Adaptive Neuro-Evolution), which evolves a population of neurons through genetic algorithms to form a neural network capable of performing a task. Symbiotic evolution promotes both cooperation and specialization, which results in a fast, efficient genetic search and discourages convergence to suboptimal solutions. In the inverted pendulum problem, SANE formed effective networks 9 to 16 times faster than the Adaptive Heuristic Critic and 2 times faster than Q-learning and the GENITOR neuro-evolution approach without loss of generalization. Such efficient learning, combined with few domain assumptions, make SANE a promising approach to a broad range of reinforcement learning problems, including many real-world applications.

Keywords: Neuro-Evolution, Reinforcement Learning, Genetic Algorithms, Neural Networks.

1. Introduction

Learning effective decision policies is a difficult problem that appears in many real-world tasks including control, scheduling, and routing. Standard supervised learning techniques are often not applicable in such tasks, because the domain information necessary to generate the target outputs is either unavailable or costly to obtain. In reinforcement learning, agents learn from signals that provide some measure of performance and which may be delivered after a sequence of decisions have been made. Reinforcement learning thus provides a means for developing profitable decision policies with minimal domain information. While reinforcement learning methods require less *a priori* knowledge than supervised techniques, they generally require a large number of training episodes and extensive CPU time. As a result, reinforcement learning has been limited to laboratory-scale problems.

This article describes a new reinforcement learning system called SANE (Symbiotic, Adaptive Neuro-Evolution), with promising scale-up properties. SANE is a novel neuro-evolution system that can form effective neural networks quickly in domains with sparse reinforcement. SANE achieves efficient learning through *symbiotic evolution*, where each individual in the population represents only a partial solution to the problem: complete solutions are formed by combining several individuals. In SANE, individual neurons are evolved to form complete neural networks. Because no single neuron can perform well alone, the population remains diverse and the genetic algorithm can search many different areas of the solution space concurrently. SANE can thus find solutions faster, and to harder problems than standard neuro-evolution systems.

An empirical evaluation of SANE was performed in the inverted pendulum problem, where it could be compared to other reinforcement learning methods. The learning speed and generalization ability of SANE was contrasted with those of the single-layer Adaptive Heuristic Critic (AHC) of Barto et al. (1983), the two-layer Adaptive Heuristic Critic of Anderson (1987, 1989), the Q-learning method of Watkins and Dayan (1992), and the GENITOR neuro-evolution system of Whitley et al. (1993). SANE was found to be considerably faster (in CPU time) and more efficient (in training episodes) than the two-layer AHC, Q-learning, and GENITOR implementations. Compared to the single-layer AHC, SANE was an order of magnitude faster even though it required more training episodes. The generalization capability of SANE was comparable to the AHC and GENITOR and was superior to Q-learning. An analysis of the final populations verifies that SANE finds solutions in diverse, unconverged populations and can maintain diversity in prolonged evolution. SANE's efficient search mechanism and resilience to convergence should allow it to extend well to harder problems.

The body of this article is organized as follows. After a brief review of neuro-evolution in section 2, section 3 presents the basic idea of symbiotic evolution. Section 4 describes the SANE method and its current implementation. The empirical evaluation of SANE in the inverted pendulum problem is presented in section 5, followed by an empirical analysis of the population dynamics in 6. Section 7 discusses related work, and 8 briefly describes other tasks where SANE has been effectively applied and outlines future areas of future research.

2. Neuro-Evolution

Genetic algorithms (Holland 1975; Goldberg 1989) are global search techniques patterned after Darwin's theory of natural evolution. Numerous potential solutions are encoded in strings, called *chromosomes*, and evaluated in a specific task. Substrings, or *genes*, of the best solutions are then combined to form new solutions, which are inserted into the population. Each iteration of the genetic algorithm consists of solution evaluation and recombination and is called a *generation*. The idea is that structures that led to good solutions in previous generations can be combined to form even better solutions in subsequent generations.

By working on a legion of solution points simultaneously, genetic algorithms sample many different areas of the solution space concurrently. Such parallel search can be very advantageous in multimodal (multi-peaked) search spaces that contain several good but suboptimal solutions. Unlike gradient methods, which perform a point-to-point search and must search each peak sequentially, genetic algorithms may evolve several distinct groups of solutions, called *species*, that search multiple peaks in parallel. Speciation can create a quicker, more efficient search as well as protect against convergence at false peaks. However, for speciation to emerge the population must contain a diverse collection of genetic material, which prevents convergence to a single solution.

Since genetic algorithms do not require explicit credit assignment to individual actions, they belong to the general class of reinforcement learning algorithms. In genetic algorithms, the only feedback that is required is a general measure of proficiency for each potential solution. Credit assignment for each action is made implicitly, since poor solutions generally choose poor individual actions. Thus, which individual actions are most responsible for a good/poor solution is irrelevant to the genetic algorithm, because by selecting against poor solutions, evolution will automatically select against poor actions.

Recently there has been much interest in evolving artificial neural networks with genetic algorithms (Belew et al., 1990; Jefferson et al., 1991; Kitano, 1990; Koza and Rice, 1991; Nolfi and Parisi, 1991; Schaffer et al., 1992, Whitley et al., 1990). In most applications of neuro-evolution, the population consists of complete neural networks and each network is evaluated independently of other networks in the population. During evolution, the population converges towards a single dominant network. Such convergence is desirable if it occurs at the global optimum, however, often populations *prematurely converge* to a local optimum. Instead of multiple parallel searches through the encoding space, the search becomes a random walk using the mutation operator. As a result, evolution ceases to make timely progress and neuro-evolution is deemed pathologically slow.

Several methods have been developed to prevent premature convergence including fitness sharing (Goldberg and Richardson, 1987), adaptive mutation (Whitley et al., 1990), crowding (Dejong, 1975), and local mating (Collins and Jefferson, 1991). Each of these techniques limits convergence through external operations that are often computationally expensive or produce a less efficient search. In the next section, a new evolutionary method will be presented that maintains diverse populations without expensive operations or high degrees of randomness.

3. Symbiotic Evolution

Normal evolutionary algorithms operate on a population of full solutions to the problem. In symbiotic evolution, each population member is only a *partial solution*. The goal of each individual is to form a partial solution that can be combined with other partial solutions currently in the population to build an effective full solution. For example in SANE, which applies the idea of symbiotic evolution to neural networks, the population consists of individual neurons, and full solutions are complete neural networks. Because single neurons rely on other neurons in the population to achieve high fitness levels, they must maintain a symbiotic relationship.

The fitness of an individual partial solution can be calculated by summing the fitness values of all possible combinations of that partial solution with other current partial solutions and dividing by the total number of combinations. Thus, an individual's fitness value reflects the average fitness of the full solutions in which the individual participated. In practice, however, evaluating all possible full solutions is intractable. The average fitness values are therefore approximated by evaluating n random subsets of partial solutions (n full solutions).

Partial solutions can be characterized as *specializations*. Instead of solving the entire problem, partial solutions specialize towards one aspect of the problem. For example, in an animal classification task one specialization may learn to recognize a mammal, while another specialization may learn to recognize a reptile. Whereas alone each specialization forms a poor classification system, conjunctively such specializations can form a complete animal classification system. Specialization will emerge because (1) individual fitness values are based on the performance of full solutions, and (2) individuals cannot delineate full solutions.

Specialization ensures diversity which prevents convergence of the population. A single partial solution cannot "take over" a population since to achieve high fitness values, there must be other specializations present. If a specialization becomes too prevalent, its members will not always be combined with other specializations in the population. Thus, redundant partial solutions do not always receive the benefit of other specializations and will incur lower fitness evaluations. Evolutionary pressures are therefore present to select against members of dominant specializations. This is quite different from standard evolutionary approaches, which always converge the population, hopefully at the global optimum, but often at a local one. In symbiotic evolution, solutions are found in diverse, *unconverged* populations.

Different specializations optimize different objective functions. In the animal classification example, recognizing mammals is different from recognizing reptiles. Evolution will, in effect, conduct separate, parallel searches in each specialization. This concurrent, divide-and-conquer approach creates a faster, more efficient search, which allows the population to discover better solutions faster, and to more difficult problems.

4. The SANE Implementation

SANE employs symbiotic evolution on a population of neurons that interconnect to form a complete neural network. More specifically, SANE evolves a population of hidden neurons for a given type of architecture such as a 2-layer-feedforward network (2 layers of weights). The basic steps in one generation of SANE are as follows (table 1): During the evaluation stage, random subpopulations of neurons of size ζ are selected and combined to form a neural network. The network is evaluated in the task and assigned a score, which is subsequently added to each selected neuron's fitness variable. The process continues until each neuron has participated in a sufficient number of networks. The average fitness of each neuron is then computed by dividing the sum of its fitness scores by the number of networks in which it participated. The neurons that have a high *average* fitness have cooperated well with other neurons in the population. Neurons that do not cooperate and are detrimental to the networks they form receive low fitness scores and are selected against.

Table 1. The basic steps in one generation of SANE.

1.	Clear all fitness values from each neuron.
2.	Select ζ neurons randomly from the population.
3.	Create a neural network from the selected neurons.
4.	Evaluate the network in the given task.
5.	Add the network's score to each selected neuron's fitness variable.
6.	Repeat steps 2-5 a sufficient number of times.
7.	Get each neuron's average fitness score by dividing its total fitness values by the number of networks in which it was implemented.
8.	Perform crossover operations on the population based on the average fitness value of each neuron.

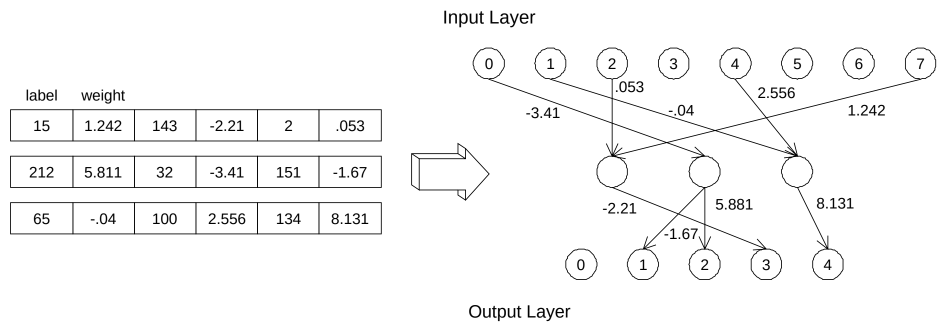

Figure 1. Forming a simple 8 input, 3 hidden, 5 output unit neural network from three hidden neuron definitions. The chromosomes of the hidden neurons are shown to the left and the corresponding network to the right. In this example, each hidden neuron has 3 connections.

Once each neuron has a fitness value, crossover operations are used to combine the chromosomes of the best-performing neurons. Mutation at low levels introduces genetic material that may have been missing from the initial population or lost during crossover operations. In other words, mutation is used only as an insurance policy against missing genetic material, not as a mechanism to create diversity.

Each neuron is defined in a bitwise chromosome that encodes a series of connection definitions, each consisting of an 8-bit label field and a 16-bit weight field. The value of the label determines where the connection is to be made. The neurons connect only to the input and the output layer, and every specified connection is connected to a valid unit. If the decimal value of the label, D, is greater than 127, then the connection is made to output unit D mod O, where O is the total number of output units. Similarly, if D is less than or equal to 127, then the connection is made to input unit D mod I, where I is the total number of input units. The weight field encodes a floating point weight for the connection. Figure 1 shows how a neural network is formed from three sample hidden neuron definitions.

Once each neuron has participated in a sufficient number of networks, the population is ranked according to the average fitness values. A mate is selected for each neuron in the top quarter of the population by choosing a neuron with an equal or higher average fitness value. A one-point crossover operator is used to combine the two neurons' chromosomes, creating two offspring per mating. The offspring replace the worst-performing neurons in the population. Mutation at the rate of 0.1% is performed on the new offspring as the last step in each generation.

Selection by rank is employed instead of the standard fitness-proportionate selection to ensure a bias towards the best performing neurons. In fitness-proportionate selection, a string s is selected for mating with probability f_s/F, where f_s is the fitness of string s and F is the average fitness of the population. As the average fitness of the strings increase, the variance in fitness decreases (Whitley, 1994). Without sufficient variance between the best and worst performing strings, the genetic algorithm will be unable to assign significant bias towards the best strings. By selecting strings based on their overall rank in the population, the best strings will always receive significant bias over the worst strings even when performance differences are small.

The current implementation of SANE has performed well, however, SANE could be implemented with a variety of different neuron encodings and even network architectures that allow recurrency. More advanced encodings and evolutionary strategies may enhance both the search efficiency and generalization ability and will be a subject of future research.

5. Empirical Evaluation

To evaluate SANE, it was implemented in the standard reinforcement learning problem of balancing a pole on a cart, where its learning speed and generalization ability could be compared to previous reinforcement learning approaches to this problem.

5.1. The Inverted Pendulum Problem

The inverted pendulum or pole-balancing problem is a classic control problem that has received much attention in the reinforcement learning literature (Anderson, 1989; Barto et al., 1983; Michie and Chambers, 1968; Whitley et al., 1993). A single pole is centered on a cart (figure 2), which may move left or right on a horizontal track. Naturally, any movements to the cart tend to unbalance the pole. The objective is to push the cart either left or right with a fixed-magnitude force such that the pole remains balanced and the track boundaries are avoided. The controller receives reinforcement only after the pole has fallen, which makes this task a challenging credit assignment problem for a reinforcement learning system.

The controller is afforded the following state information: the position of the cart (ρ), the velocity of the cart ($\dot{\rho}$), the angle of the pole (θ), and the angular velocity of the pole ($\dot{\theta}$). At each time step, the controller must resolve which direction the

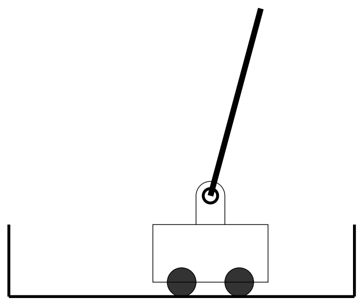

Figure 2. The cart-and-pole system in the inverted pendulum problem. The cart is pushed either left or right until it reaches a track boundary or the pole falls below 12 degrees.

cart is to be pushed. The cart and pole system can be described with the following second order equations of motion:

$$\ddot{\theta}_t = \frac{mg\ sin\theta_t - cos\theta_t[F_t + m_p l\dot{\theta}_t^2\ sin\theta_t]}{(4/3)ml - m_p l\ cos^2\theta_t},\quad \ddot{\rho}_t = \frac{F_t + m_p l[\dot{\theta}_t^2\ sin\theta_t - \ddot{\theta}_t\ cos\theta_t]}{m},$$

where

ρ:	The position of the cart.
$\dot{\rho}$:	The velocity of the cart.
θ:	The angle of the pole.
$\dot{\theta}$:	The angular velocity of the pole.
l:	The length of the pole = 0.5 m.
m_p:	The mass of the pole = 0.1 kg.
m:	The mass of the pole and cart = 1.1 kg.
F:	The magnitude of force = 10 N.
g:	The acceleration due to gravity = 9.8.

Through Euler's method of numerical approximation, the cart and pole system can be simulated using discrete-time equations of the form $\theta(t+1) = \theta(t) + \tau\dot{\theta}(t)$, with the discrete time step τ normally set at 0.02 seconds. Once the pole falls past 12 degrees or the cart reaches the boundary of the 4.8 meter track, the trial ends and a reinforcement signal is generated. The performance of the controller is measured by the number of time steps in which the pole remains balanced. The above parameters are identical to those used by Barto et al. (1983), Anderson (1987), and Whitley et al. (1993) in this problem.

5.2. Controller Implementations

Five different reinforcement learning methods were implemented to form a control strategy for the pole-balancing problem: SANE, the single-layer Adaptive Heuristic

Critic (AHC) of Barto et al. (1983), the two-layer AHC of Anderson (1987, 1989), the Q-learning method of Watkins and Dayan (1992), and the GENITOR system of Whitley et al. (1993). The original programs written by Sutton and Anderson were used for the AHC implementations, and the simulation code developed by Pendrith (1994) was used for the Q-learning implementation. For GENITOR, the system was reimplemented as described in (Whitley et al., 1993). A control strategy was deemed successful if it could balance a pole for 120,000 time steps.

5.2.1. SANE

SANE was implemented to evolve a 2-layer network with 5 input, 8 hidden, and 2 output units. Each hidden neuron specified 5 connections giving each network a total of 40 connections. The number of hidden neurons was chosen so that the total number of connections was compatible with the 2-layer AHC and GENITOR implementations. Each network evaluation consisted of a single balance attempt where a sequence of control decisions were made until the pole fell or the track boundaries were reached. Two hundred networks were formed and evaluated per generation, which allowed each neuron to participate in 8 networks per generation on average. The input to the network consisted of the 4 state variables $(\theta, \dot{\theta}, \rho, \dot{\rho})$, normalized between 0 and 1 over the following ranges:

$$\begin{array}{rl} \rho: & (-2.4, 2.4) \\ \dot{\rho}: & (-1.5, 1.5) \\ \theta: & (-12^\circ, 12^\circ) \\ \dot{\theta}: & (-60^\circ, 60^\circ) \end{array}$$

To make the network input compatible with the implementations of Whitley et al. (1993) and Anderson (1987), an additional bias input unit that is always set to 0.5 was included.

Each of the two output units corresponded directly with a control choice (left or right). The output unit with the greatest activation determined which control action was to be performed. The output layer, thus, represented a ranking of the possible choices. This approach is quite different from most neuro-control architectures, where the activation of an output unit represents a probability of that action being performed (Anderson, 1987; Barto et al., 1983; Whitley et al., 1993). For example, a decision of "move right" with activation 0.9 would move right only 90% of the time. Probabilistic output units allow the network to visit more of the state space during training, and thus incorporate a more global view of the problem into the control policy (Whitley et al., 1993). In the SANE implementation, however, randomness is unnecessary in the decision process since there is a large amount of state space sampling through multiple combinations of neurons.

Table 2. Implementation parameters for each method.

	1-AHC	2-AHC	QL		GENITOR	SANE
Action Learning Rate (α):	1.0	1.0		Population Size:	100	200
Critic Learning Rate (β):	0.5	0.2	0.2	Mutation Rate:	Adaptive	0.1%
TD Discount Factor (γ):	0.95	0.9	0.95	Chromosome Length:	35 (floats)	120 (bits)
Decay Rate (λ):	0.9	0		Subpopulation (ζ):		8

5.2.2. The Adaptive Heuristic Critic

The Adaptive Heuristic Critic is one of the best-known reinforcement learning methods, and has been shown effective in the inverted pendulum problem. The AHC framework consists of two separate networks: an *action network* and an *evaluation network*. The action network receives the current problem state and chooses an appropriate control action. The evaluation network receives the same input, and evaluates or critiques the current state. The evaluation network is trained using the temporal difference method (Sutton, 1988) to predict the expected outcome of the current trial given the current state and the action network's current decision policy. The differences in predictions between consecutive states provide effective credit assignment to individual actions selected by the action network. Such credit assignment is used to train the action network using a standard supervised learning algorithm such as backpropagation.

Two different AHC implementations were tested: A single-layer version (Barto et al., 1983) and a two-layer version (Anderson, 1987). Table 2 lists the parameters for each method. Both implementations were run directly from pole-balance simulators written by Sutton and Anderson, respectively. The learning parameters, network architectures, and control strategy were thus chosen by Sutton and Anderson and presumably reflect parameters that have been found effective.

Since the state evaluation function to be learned is non-monotonic (Anderson, 1989) and single-layer networks can only learn linearly-separable tasks, Barto et al. (1983) discretized the input space into 162 nonoverlapping regions or "boxes" for the single-layer AHC. This approach was first introduced by Michie and Chambers (1968), and it allows the state evaluation to be a linear function of the input. Both the evaluation and action network consist of one unit with a single weight connected to each input box. The output of the unit is the inner product of the input vector and the unit's weight vector, however, since only one input box will be active at one time, the output reduces to the weight corresponding to the active input box.

In the two-layer AHC, discretization of the input space is not necessary since additional hidden units allow the network to represent any non-linear discriminant function. Therefore, the same continuous input that was used for SANE was also used for the two-layer AHC. Each network (evaluation and action) in Anderson's implementation consists of 5 input units (4 input variables and one bias unit set at 0.5), 5 hidden units, and one output unit. Each input unit is connected to every hidden unit and to the single output unit. The two-layer networks are trained using a variant of backpropagation (Anderson, 1989). The output of the action network is interpreted as the probability of choosing that action (push left or right) in both the single and two-layer AHC implementations.

5.2.3. Q-learning

Q-learning (Watkins, 1989; Watkins and Dayan 1992) is closely related to the AHC and is currently the most widely-studied reinforcement learning algorithm. In Q-learning, the Q-function is a predictive function that estimates the expected return from the current state and action pair. Given accurate Q-function values, called Q values, an optimal decision policy is one that selects the action with the highest associated Q value (expected payoff) for each state. The Q-function is learned through "incremental dynamic programming" (Watkins and Dayan, 1992), which maintains an estimate $\hat{Q}$ of the Q values and updates the estimates based on immediate payoffs and estimated payoffs from subsequent states.

Our Q-learning simulations were run using the simulation code developed by Pendrith (1994), which employs one-step updates as described by Watkins and Dayan (1992). In this implementation, the Q-function is a look-up table that receives the same discretized input that Barto et al. created for the single-layer AHC. Actions on even-numbered steps are determined using the stochastic action selector described by Lin (1992). The action on odd-numbered steps is chosen deterministically according to the highest associated Q-value. Pendrith (1994) found that such interleaved exploration and exploitation greatly improves Q-learning in the pole-balancing domain. Our experiments confirmed this result: when interleaving was disabled, Q-learning was incapable of learning the pole-balancing task.

5.2.4. GENITOR

The motivation for comparing SANE to GENITOR is twofold. GENITOR is an advanced genetic algorithm method that includes external functions for ensuring population diversity. Diversity is maintained through *adaptive mutation*, which raises the mutation rate as the population converges (section 7.1). Comparisons between GENITOR's and SANE's search efficiency thus test the hypothesis that symbiotic evolution produces an efficient search without reliance on additional randomness. Since GENITOR has been shown to be effective in evolving neural networks for the inverted pendulum problem (Whitley et al.,1993), it also provides a state-of-the-art neuro-evolution comparison.

GENITOR was implemented as detailed in (Whitley et al.,1993) to evolve the weights in a fully-connected 2-layer network, with additional connections from each input unit to the output layer. The network architecture is identical to the two-layer AHC with 5 input units, 5 hidden units and 1 output unit. The input to the network consists of the same normalized state variables as in SANE, and the activation of the output unit is interpreted as a probabilistic choice as in the AHC.

5.3. Learning-Speed Comparisons

The first experiments compared the time required by each algorithm to develop a successful network. Both the number of pole-balance attempts required and the CPU time expended were measured and averaged over 50 simulations. The number of balance attempts reflects the number of training episodes required. The CPU time was included because the number of balance attempts does not describe the amount of overhead each algorithm incurs. The CPU times should be treated as ballpark estimates because they are sensitive to the implementation details. However, the CPU time differences found in these experiments are large enough to indicate real differences in training time among the algorithms. Each implementation was written in C and compiled using the cc compiler on an IBM RS6000 25T workstation with the -O2 optimization flag. Otherwise, no special effort was made to optimize any of the implementations for speed.

The first comparison (table 3) was based on the static start state of Barto et al. (1983). The pole always started from a centered position with the cart in the middle of the track. Neither the pole nor the cart had any initial velocity. The second comparison (table 4), was based on the random start states of Anderson (1987, 1989) and Whitley et al. (1993). The cart and pole were both started from random positions with random initial velocity. The positions and velocities were selected from the same ranges that were used to normalize the input variables, and could specify a state from which pole balancing was impossible. With random initial states, a network was considered successful if it could balance the pole from any single start state.

5.3.1. Results

The results show the AHCs to require significantly more CPU time than the other approaches to discover effective solutions. While the single-layer AHC needed the lowest number of balance attempts on average, its long CPU times overshadow its efficient learning. Typically, it took over two minutes for the single-layer AHC to find a successful network. This overhead is particularly large when compared to the genetic algorithm approaches, which took only five to ten seconds. The two-layer AHC performed the poorest, exhausting large amounts of CPU time and requiring at least 5, but often 10 to 20, times more balance attempts on average than the other approaches.

The experiments confirmed Whitley's observation that the AHC trains inconsistently when started from random initial states. Out of the 50 simulations, the single-layer AHC failed to train in 3 and the two-layer AHC failed in 14. Each unsuccessful simulation was allowed to train for 50,000 pole balance attempts before it was declared a failure. The results presented for the AHC in tables 3 and 4 are averaged over the successful simulations, excluding the failures.

Q-learning and GENITOR were comparable across both tests in terms of mean CPU time and average number of balance attempts required. The differences in

Table 3. The CPU time and number of pole balance attempts required to find a successful network starting from a centered pole and cart in each attempt. The number of pole balance attempts refers to the number of training episodes or "starts" necessary. The numbers are computed over 50 simulations for each method. A training failure was said to occur if no successful network was found after 50,000 attempts. The differences in means are statistically significant ($p < .01$), except the number of pole balance attempts between Q-learning and GENITOR.

	CPU Seconds				Pole Balance Attempts				
Method	Mean	Best	Worst	SD	Mean	Best	Worst	SD	Failures
1-layer AHC	130.6	17	3017	423.6	232	32	5003	709	0
2-layer AHC	99.1	17	863	158.6	8976	3963	41308	7573	4
Q-learning	19.8	5	99	17.9	1975	366	10164	1919	0
GENITOR	9.5	4	45	7.4	1846	272	7052	1396	0
SANE	5.9	4	8	0.6	535	70	1910	329	0

Table 4. The CPU time and number of pole balance attempts required to find a successful network starting from random pole and cart positions with random initial velocities. The differences in means between Q-learning and GENITOR are not significant ($p < .01$); the other mean differences are.

	CPU Seconds				Pole Balance Attempts				
Method	Mean	Best	Worst	SD	Mean	Best	Worst	SD	Failures
1-layer AHC	49.4	14	250	52.6	430	80	7373	1071	3
2-layer AHC	83.8	13	311	61.6	12513	3458	45922	9338	14
Q-learning	12.2	4	41	7.8	2402	426	10056	1903	0
GENITOR	9.8	4	54	7.9	2578	415	12964	2092	0
SANE	5.2	4	9	1.1	1691	46	4461	984	0

CPU times between the two approaches are not large enough to discount implementation details, and when started from random start states the difference is not statistically significant. Both Q-learning and GENITOR were close to an order of magnitude faster than the AHCs and incurred no training failures.

SANE expended one half of the CPU time of Q-learning and GENITOR on average and required significantly fewer balance attempts. Like Q-learning and GENITOR, SANE found solutions in every simulation. In addition, the time required to learn the task varied the least in the SANE simulations. When starting from random initial states, 90% of the CPU times (in seconds) fall in the following ranges:

1-layer AHC:	$[17, 136]$
2-layer AHC:	$[17, 124]$
Q-learning:	$[4, 20]$
GENITOR:	$[4, 17]$
SANE:	$[4, 6]$

Thus, while the AHC can vary as much as 2 minutes among simulations and Q-learning and GENITOR about 15 seconds, SANE consistently finds solutions in 4 to 6 seconds of CPU time, making it the fastest and most consistent of the learning methods tested in this task.

5.3.2. Discussion

The large CPU times of the AHC are caused by the many weight updates that they must perform after every action. Both the single and two-layer AHCs adjust every weight in the neural networks after each activation. Since there are thousands of activations per balance attempt, the time required for the weight updates can be substantial. The Q-learning implementation reduces this overhead considerably by only updating a single table entry after every step, however, these continuous updates still consume costly CPU cycles. Neither SANE nor GENITOR require weight updates after each activation, and do not incur these high overhead costs.

Note that the Q-function can be represented efficiently as a look-up table only when the state space is small. In a real-world application, the enormous state space would make explicit representation of each state impossible. Larger applications of Q-learning are likely to use neural networks (Lin 1992), which can learn from continuous input values in an infinite state space. Instead of representing each state explicitly, neural networks form internal representations of the state space through their connections and weights, which allows them to generalize well to unobserved states. Like the AHC, a neural network implementation of Q-learning would require continuous updates of all neural network weights, which would exhaust considerably more CPU time than the table look-up implementation.

Both the single-layer AHC and Q-learning had the benefit of a presegmented input space, while the two-layer AHC, GENITOR, and SANE methods received only undifferentiated real values of the state variables. Barto et al. (1983) selected the input boxes according to prior knowledge of the "useful regions" of the input variables and their compatibility with the single-layer AHC. This information allowed the single-layer AHC to learn the task in the least number of balance attempts. The input partitioning, however, did not extend well to the Q-learner, which required as many pole-balance attempts as the methods receiving real-valued inputs.

Interestingly, the results achieved with GENITOR were superior to those reported by Whitley et al. (1993). This disparity is probably caused by the way the input variables were normalized. Since it was unclear what ranges Whitley et al. (1993) used for normalization, the input vectors could be quite different. On average, our implementation of GENITOR required only half of the attempts, which suggests that Whitley et al. may have normalized over an overly broad range.

The comparison between SANE and GENITOR confirms our hypothesis that symbiotic evolution can perform an efficient genetic search without relying on high mutation rates. It appears that in GENITOR, the high mutation rates brought on through adaptive mutation may be causing many disruptions in highly-fit schemata (genetic building blocks), resulting in many more network evaluations required to learn the task.

Table 5. The generalization ability of the networks formed with each method. The numbers show the percentage of random start states balanced for 1000 time steps by a fully-trained network. Fifty networks were formed with each method. There is a statistically significant difference ($p < .01$) between the mean generalizations of Q-learning and those of the single-layer AHC, GENITOR, and SANE. The other differences are not significant.

Method	Mean	Best	Worst	SD
1-layer AHC	50	76	2	16
2-layer AHC	44	76	5	20
Q-learning	41	61	13	11
GENITOR	48	81	2	23
SANE	48	81	1	25

5.4. Generalization Comparisons

The second battery of tests explored the generalization ability of each network. Networks that generalize well can transfer concepts learned in a subset of the state space to the rest of the space. Such behavior is of great benefit in real-world tasks where the enormous state spaces make explicit exploration of all states infeasible. In the pole balancing task, networks were trained until a network could balance the pole from a single start state. How well these networks could balance from other start states demonstrates their ability to generalize.

One hundred random start states were created as a test set for the final network of each method. The network was said to successfully balance a start state if the pole did not fall below 12° within 1000 time steps. Table 5 shows the generalization performance over 50 simulations. Since some initial states contained situations from which pole balancing was impossible, the best networks were successful only 80% of the time.

Generalization was comparable across the AHCs and the genetic algorithm approaches. The mean generalization of the Q-learning implementation, however, was significantly lower than those of the single-layer AHC, GENITOR, and SANE. This disparity is likely due to the look-up table employed by the Q-learner. In the single-layer AHC, which uses the same presegmented input space as the Q-learner, all weights are updated after visiting a single state, allowing it to learn a smoother approximation of the control function. In Q-learning, only the weight (i.e. the table value) of the currently visited state is updated, preventing interpolation to unvisited states.

Whitley et al. (1993) speculated that an inverse relationship exists between learning speed and generalization. In their experiments, solutions that were found in early generations tended to have poorer performance on novel inputs. Sammut and Cribb (1990) also found that programs that learn faster often result in very specific strategies that do not generalize well. This phenomenon, however, was not observed in the SANE simulations. Figure 3 plots the number of network evaluations incurred before a solution was found against its generalization ability for

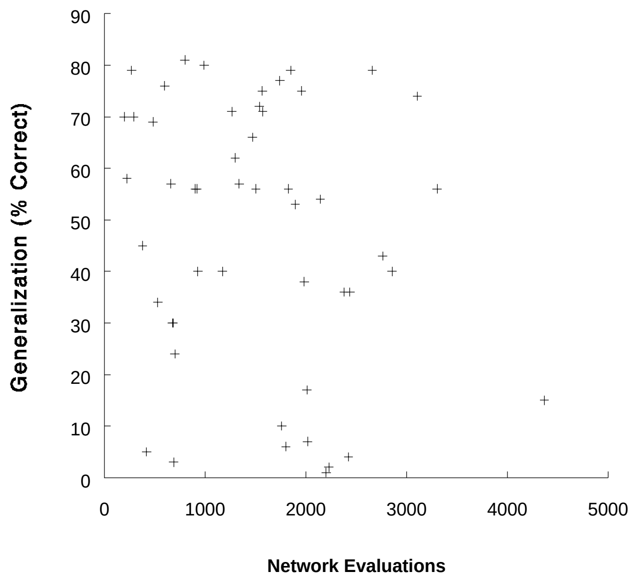

Figure 3. A plot of the learning speed versus generalization for the final networks formed in the 50 SANE simulations. The learning speed is measured by the number of network evaluations (balance attempts) during the evolution of the network, and the generalization is measured by the percentage of random start states balanced for 1000 time steps. The points are uniformly scattered indicating that learning speed does not affect generalization.

each of the 50 SANE simulations. As seen by the graph, no correlation appears to exist between learning speed and generalization. These results suggest that further optimizations to SANE will not restrict generalization.

6. Population Dynamics in Symbiotic Evolution

In section 3, we hypothesized that the power of the SANE approach stems from its ability to evolve several specializations concurrently. Whereas standard approaches converge the population to the desired solution, SANE forms solutions in diverse, unconverged populations. To test this hypothesis, an empirical study was conducted where the diversity levels of populations evolved by SANE were compared to those of an otherwise identical approach, but one that evolved a population of networks. Thus, the *only* difference between the two approaches was the underlying evolutionary method (symbiotic vs. standard). Whereas in SANE each chromosome consisted of 120 bits or one neuron definition, in the standard approach each chromosome contained 960 bits or 8 (the value of ζ) neuron definitions. All other parameters including population size, mutation rate, selection strategy, and number of networks evaluated per generation (200) were identical.

The comparisons were performed in the inverted pendulum problem starting from random initial states. However, with the standard parameter settings, (section 5.1), SANE found solutions so quickly that diversity was not even an issue. Therefore, the pendulum length was extended to 2.0 meters. With a longer pole, the angular acceleration of the pole $\ddot{\theta}$ is increased, because the pole has more mass and the pole's center of mass is farther away from the cart. As a result, some states that were previously recoverable no longer are. The controllers receive more initial states from which pole balancing is impossible, and consequently require more balance attempts to form an effective control policy. A more difficult problem to learn prolongs the evolution and thereby makes the population more susceptible to diversity loss.

Ten simulations were run using each method. Once each simulation established a successful network, the diversity of the population, Φ, was measured by taking the average Hamming distance between every two chromosomes and dividing by the length of the chromosome:

$$\Phi = \frac{2\sum_{i=1}^{n}\sum_{j=i+1}^{n} H_{i,j}}{n(n-1)l},$$

where n is the population size, l is the length of each chromosome, and $H_{i,j}$ is the Hamming distance between chromosomes i and j. The value Φ represents the probability that a given bit at a specific position on one chromosome is different from a bit at the same position on a different chromosome. Thus, a random population would have $\Phi = 0.5$ since there is a 50% probability that any two bits in the same position differ.

Figure 4 shows the average population diversity Φ as a function of each generation. A significant loss of diversity occurred in the standard approach in early generations as the populations quickly converged. After only 50 generations, 75% of any two chromosomes were identical ($\Phi = 0.25$). After 100 generations, 95% of two chromosomes were the same. In the symbiotic approach, the diversity level decreased initially but reached a plateau of 0.35 around generation 100. The symbiotic diversity level never fell below 0.32 in any simulation. SANE was able to form solutions in every simulation, while the standard approach found solutions in only 3. SANE found its solutions between 10 and 201 generations (67 on average), with an average final diversity $\Phi = 0.38$. The three solutions found by the standard approach were at generations 69, 76, and 480, with an average diversity of 0.14. The failed simulations were stopped after 1000 generations (i.e. 200,000 pole-balance attempts).

These results confirm the hypothesis that symbiotic evolution establishes solutions in diverse populations and can maintain diversity in prolonged evolution. Whereas evolving full solutions caused the population to converge and fail to find solutions, the symbiotic approach always found a solution and in an unconverged population. This cooperative, efficient, genetic search is the hallmark of symbiotic evolution and should allow SANE to extend to more difficult problems.

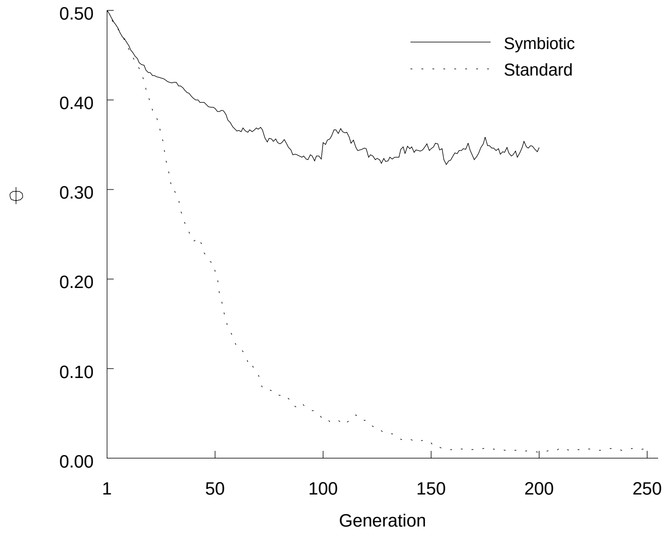

Figure 4. The average population diversity Φ after each generation. The diversity measures were averaged over 10 simulations using each method. The symbiotic method maintained high levels of diversity, while the standard genetic algorithm quickly converged.

7. Related Work

Work most closely related to SANE can be divided into two categories: genetic reinforcement learning and coevolutionary genetic algorithms.

7.1. Genetic Reinforcement Learning

Several systems have been built or proposed for reinforcement learning through genetic algorithms, including both symbolic and neural network approaches. The SAMUEL system (Grefenstette et al.,1990) uses genetic algorithms to evolve rule-based classifiers in sequential decision tasks. Unlike most classifier systems where genetic algorithms evolve individual rules, SAMUEL evolves a population of classifier systems or "tactical plans" composed of several action rules. SAMUEL is a model-based system that is designed to evolve decision plans offline in a simulation of the domain and then incrementally add the current best plan to the actual domain. SAMUEL has been shown effective in several small problems including the evasive maneuvers problem (Grefenstette et al.,1990) and the game of cat-and-mouse (Grefenstette, 1992), and in more recent work, SAMUEL has been extended to the task of mobile robot navigation (Grefenstette and Schultz, 1994). The main difference between SAMUEL and SANE lies in the choice of representation. Whereas SAMUEL evolves a set of rules for sequential decision tasks, SANE evolves neural networks. The interpolative ability of neural networks should allow SANE to learn tasks quicker than SAMUEL, however, it is easier to incorporate pre-existing knowledge of the task into the initial population of SAMUEL.

GENITOR (Whitley and Kauth, 1988; Whitley, 1989) is an "aggressive search" genetic algorithm that has been shown to be quite effective as a reinforcement learning tool for neural networks. GENITOR is considered aggressive because it uses small populations, large mutation rates, and rank-based selection to create greater variance in solution space sampling. In the GENITOR neuro-evolution implementation, each network's weights are concatenated in a real-valued chromosome along with a gene that represents the crossover probability. The crossover allele determines whether the network is to be mutated or whether a crossover operation is to be performed with a second network. The crossover allele is modified and passed to the offspring based on the offspring's performance compared to the parent. If the offspring outperforms the parent, the crossover probability is decreased. Otherwise, it is increased. Whitley refers to this technique as adaptive mutation because it tends to increase the mutation rate as populations converge.

There are several key differences between GENITOR and SANE. The main difference, however, is that GENITOR evolves full networks and requires extra randomness to maintain diverse populations, whereas SANE ensures diversity by building it into the evaluation itself. As demonstrated in the pole-balancing simulations, GENITOR's high mutation rates can lead to a less efficient search than SANE's symbiotic approach. Another difference between SANE and Whitley's approach lies in the network architectures. In the current implementation of GENITOR (Whitley et al.,1993), the network architecture is fixed and only the weights are evolved. The implementor must resolve a priori how the network should be connected. In SANE, the topology of the network evolves together with the weights, granting more freedom to the genetic algorithm to manifest useful neural structures.

7.2. Coevolutionary Genetic Algorithms

Symbiotic evolution is somewhat similar to implicit fitness sharing or co-adaptive genetic algorithms (Smith et al., 1993; Smith and Gray, 1993). In their immune system model, Smith et al. (1993) evolved artificial antibodies to recognize or match artificial antigens. Since each antibody can only match one antigen, a diverse population of antibodies is necessary to effectively guard against a variety of antigens. The co-adaptive genetic algorithm model, however, is based more on competition than cooperation. Each antibody must compete for survival with other antibodies in the subpopulation to recognize the given antigen. The fitness of each individual reflects how well it matches its opposing antigen, not how well it cooperates with other individuals. The antibodies are thus not dependent on other antibodies for recognition of an antigen and only interact implicitly through competition. Horn et al. (1994) characterize this difference as weak cooperation (co-adaptive GA) vs. strong cooperation (symbiotic evolution). Since both approaches appear to have similar effects in terms of population diversity and speciation, further research is necessary to discover the relative strengths and weaknesses of each method.

Smith (1994) has recently proposed a method where a learning classifier system (LCS) can be mapped to a neural network. Each hidden node represents a classifier rule that must compete with other hidden nodes in a winner-take-all competition. Like SANE, the evolution in the LCS/NN is performed on the neuron level instead of at the network level. Unlike SANE, however, the LCS/NN does not form a complete neural network, but rather relies on a gradient descent method such as backpropagation to "tune" the weights. Such reliance precludes the LCS/NN from most reinforcement learning tasks where sparse reinforcement makes gradient information unavailable.

Potter and De Jong have developed a symbiotic evolutionary strategy called Cooperative Coevolutionary Genetic Algorithms (CCGA) and have applied it to both neural network and rule-based systems (Potter and De Jong, 1995a; Potter et al., 1995b). The CCGA evolves partial solutions much like SANE, but distributes the individuals differently. Whereas SANE keeps all individuals in a single population, the CCGA evolves specializations in distinct subpopulations or *islands*. Members of different subpopulations do not interbreed across subpopulations, which eliminates haphazard, destructive recombination between dominant specializations, but also removes information-sharing between specializations.

Evolving in distinct subpopulations places a heavier burden on a priori knowledge of the number of specializations necessary to form an effective complete solution. In SANE, the number and distribution of the specializations is determined implicitly throughout evolution. For example, a network may be given eight hidden neurons but may only require four *types* of hidden neurons. SANE would evolve four different specializations and redundantly select two from each for the final network. While two subpopulations in the CCGA could represent the same specialization, they cannot share information and therefore are forced to find the redundant specialization independently. Potter and De Jong (1995a) have proposed a method that automatically determines the number of partial solutions necessary by incrementally adding random subpopulations. This approach appears promising, and motivates further research comparing the single population and incremental subpopulation approaches.

8. Extending SANE

SANE is a general reinforcement learning method that makes very few domain assumptions. It can thus be implemented in a broad range of tasks including real-world decision tasks. We have implemented SANE in two such tasks in the field of artificial intelligence: value ordering in constraint satisfaction problems and focusing a minimax search (Moriarty and Miikkulainen, 1994a, 1994b).

Value ordering in constraint satisfaction problems is a well-studied task where problem-general approaches have performed inconsistently. A SANE network was used to decide the order in which types or classes of cars were assigned on an assembly line, which is an NP-complete problem. The network was implemented in a chronological backtrack search and the number of backtracks incurred determined each network's score. The final SANE network required 1/30 of the backtracks of random value ordering and 1/3 of the backtracks of the commonly-used maximization-of-future-options heuristic.

In the second task, SANE was implemented to focus minimax search in the game of Othello. SANE formed a network to decide which moves from a given board situation are promising enough to be evaluated. Such decisions can establish better play by effectively hiding bad states from the minimax search. Using the powerful evaluation function from Bill (Lee and Mahajan, 1990), the SANE network was able to generate better play while examining 33% fewer board positions than a normal, full-width minimax search using the same evaluation function.

Future work on SANE includes applying it to larger real-world domains with multiple decision tasks. Possible tasks include local area network routing and scheduling, robot control, elevator control, air and automobile traffic control, and financial market trading. Since SANE makes few domain assumptions, it should be applicable in each of these domains as well as many others. An important question to be explored in such domains is: can SANE simultaneously evolve networks for separate decision tasks? For example, in a local area network, can neurons involved in priority queuing be simultaneously evolved with neurons for packet routing? Evolving neurons to form many different networks should not be any different than for a single network, since even in a single network SANE must develop neurons that specialize and serve very different roles. To evolve multiple networks, the input layers and output layers of each network could be concatenated to form a single, multi-task network. Which input units are activated and which output units are evaluated would depend on which decision task was to be performed. Since a hidden neuron could establish connections to any input or output unit, it could specialize its connections to a single decision task or form connections between sub-networks that perform different tasks. Such inter-network connections could produce interesting interactions between decision policies.

A potential key advantage of symbiotic evolution not yet fully explored is the ability to adapt quickly to changes in the environment. Often in control applications, fluctuations in the domain may require quick adaptation of the current decision policy. In gradient or point-to-point searches, adaptation can be as slow as retraining from a random point. Similarly in standard genetic algorithms which converge the population to a single solution, the lack of diverse genetic material makes further traversals of the solution space extremely slow. In SANE, however, the population does not converge. SANE's population should therefore remain highly adaptive to any changes in the fitness landscape.

While we believe that symbiotic evolution is a general principle, applicable not only to neural networks but to other representations as well, not all representations may be compatible with this approach. Symbiosis emerges naturally in the current representation of neural networks as collections of hidden neurons, but preliminary experiments with other types of encodings, such as populations of individual network connections, have been unsuccessful (Steetskamp, 1995). An important facet of SANE's neurons is that they form complete input to output mappings, which makes every neuron a primitive solution in its own right. SANE can thus form subsumption-type architectures (Brooks, 1991), where certain neurons provide very crude solutions and other neurons perform higher-level functions that fix problems in the crude solutions. Preliminary studies in simple classification tasks have uncovered some subsumptive behavior among SANE's neurons. An important focus for future research will be to further analyze the functions of evolved hidden neurons and to study other symbiotic-conducive representations.

9. Conclusion

SANE is a new reinforcement learning system that achieves fast, efficient learning through a new genetic search strategy called symbiotic evolution. By evolving individual neurons, SANE builds effective neural networks quickly in diverse populations. In the inverted pendulum problem, SANE was faster, more efficient, and more consistent than the earlier AHC approaches, Q-learning, and the GENITOR neuro-evolution system. Moreover, SANE's quick solutions do not lack generalization as suggested by Whitley et al. (1993). Future experiments will extend SANE to more difficult real-world problems where the ability to perform multiple concurrent searches in an unconverged population should allow SANE to scale up to previously unachievable tasks.

Acknowledgments

Thanks to Mark Pendrith and Claude Sammut for providing their implementation of Q-learning for pole balancing and to Rich Sutton and Charles Anderson for making their AHC simulation code available. This research was supported in part by the National Science Foundation under grant #IRI-930973.

References

Anderson, C. W. (1987). Strategy learning with multilayer connectionist representations. Technical Report TR87-509.3, GTE Labs, Waltham, MA.

Anderson, C. W. (1989). Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9:31–37.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846.

Belew, R. K., McInerney, J., and Schraudolph, N. N. (1991). Evolving networks: Using the genetic algorithm with connectionist learning. In Farmer, J. D., Langton, C., Rasmussen, S., and Taylor, C., editors, *Artificial Life II*. Reading, MA: Addison-Wesley.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

Collins, R. J., and Jefferson, D. R. (1991). Selection in massively parallel genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 249–256. San Mateo, CA: Morgan Kaufmann.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, The University of Michigan, Ann Arbor, MI.

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

Goldberg, D. E., and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, 148–154. San Mateo, CA: Morgan Kaufmann.

Grefenstette, J., and Schultz, A. (1994). An evolutionary approach to learning in robots. In *Proceedings of the Machine Learning Workshop on Robot Learning, Eleventh International Conference on Machine Learning.* New Brunswick, NJ.

Grefenstette, J. J. (1992). An approach to anytime learning. In *Proceedings of the Ninth International Conference on Machine Learning*, 189–195.

Grefenstette, J. J., Ramsey, C. L., and Schultz, A. C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5:355–381.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Ann Arbor, MI: University of Michigan Press.

Horn, J., Goldberg, D. E., and Deb, K. (1994). Implicit niching in a learning classifier system: Nature's way. *Evolutionary Computation*, 2(1):37–66.

Jefferson, D., Collins, R., Cooper, C., Dyer, M., Flowers, M., Korf, R., Taylor, C., and Wang, A. (1991). Evolution as a theme in artificial life: The genesys/tracker system. In Farmer, J. D., Langton, C., Rasmussen, S., and Taylor, C., editors, *Artificial Life II.* Reading, MA: Addison-Wesley.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476.

Koza, J. R., and Rice, J. P. (1991). Genetic generalization of both the weights and architecture for a neural network. In *International Joint Conference on Neural Networks*, vol. 2, 397–404. New York, NY: IEEE.

Lee, K.-F., and Mahajan, S. (1990). The development of a world class Othello program. *Artificial Intelligence*, 43:21–36.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8(3):293–321.

Michie, D., and Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E., and Michie, D., editors, *Machine Intelligence.* Edinburgh, UK: Oliver and Boyd.

Moriarty, D. E., and Miikkulainen, R. (1994a). Evolutionary neural networks for value ordering in constraint satisfaction problems. Technical Report AI94-218, Department of Computer Sciences, The University of Texas at Austin.

Moriarty, D. E., and Miikkulainen, R. (1994b). Evolving neural networks to focus minimax search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1371-1377. Seattle, WA: MIT Press.

Nolfi, S., and Parisi, D. (1992). Growing neural networks. *Artificial Life III*, Reading, MA: Addison-Wesley.

Pendrith, M. (1994). On reinforcement learning of control actions in noisy and non-Markovian domains. Technical Report UNSW-CSE-TR-9410, School of Computer Science and Engineering, The University of New South Wales.

Potter, M., and De Jong, K. (1995a). Evolving neural networks with collaborative species. In *Proceedings of the 1995 Summer Computer Simulation Conference.* Ottawa, Canada.

Potter, M., De Jong, K., and Grefenstette, J. (1995b). A coevolutionary approach to learning sequential decision rules. In *Proceedings of the Sixth International Conference on Genetic Algorithms.* Pittsburgh, PA.

Sammut, C., and Cribb, J. (1990). Is learning rate a good performance criterion for learning? In *Proceedings of the Seventh International Conference on Machine Learning*, 170–178. Morgan Kaufmann.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*. Baltimore, MD.

Smith, R. E. (1994). Is a learning classifier system a type of neural network? *Evolutionary Computation*, 2(1).

Smith, R. E., Forrest, S., and Perelson, A. S. (1993). Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1(2):127–149.

Smith, R. E., and Gray, B. (1993). Co-adaptive genetic algorithms: An example in Othello strategy. Technical Report TCGA 94002, Department of Engineering Science and Mechanics, The University of Alabama.

Steetskamp, R. (1995) Explorations in symbiotic neuro-evolution search spaces. Masters Stage Report, Department of Computer Science, University of Twente, The Netherlands.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

Syswerda, G. (1991). A study of reproduction in generational and steady-state genetic algorithms. In Rawlings, G., editor, *Foundations of Genetic Algorithms*, 94–101. San Mateo, CA: Morgan-Kaufmann.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* PhD thesis, University of Cambridge, England.

Watkins, C. J. C. H., and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Whitley, D. (1989). The GENITOR algorithm and selective pressure. In *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufman.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85.

Whitley, D., Dominic, S., Das, R., and Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284.

Whitley, D., and Kauth, J. (1988). GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, 118–130. Denver, CO.

Whitley, D., Starkweather, T., and Bogart, C. (1990). Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14:347–361.