Machine Learning, 13, 259-284 (1993)
© 1993 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Genetic Reinforcement Learning for Neurocontrol Problems

DARRELL WHITLEY, STEPHEN DOMINIC, RAJARSHI DAS, AND
CHARLES W. ANDERSON WHITLEY@CS.COLOSTATE.EDU
Computer Science Department, Colorado State University, Fort Collins, CO 80523

Abstract. Empirical tests indicate that at least one class of genetic algorithms yields good performance for neural network weight optimization in terms of learning rates and scalability. The successful application of these genetic algorithms to supervised learning problems sets the stage for the use of genetic algorithms in reinforcement learning problems. On a simulated inverted-pendulum control problem, "genetic reinforcement learning" produces competitive results with AHC, another well-known reinforcement learning paradigm for neural networks that employs the temporal difference method. These algorithms are compared in terms of learning rates, performance-based generalization, and control behavior over time.

Keywords. Genetic algorithms, reinforcement learning, neural networks, adaptive control

1. Introduction

Recent applications of genetic algorithms to neural network weight optimization produce results roughly competitive with standard back propagation (Montana & Davis, 1989; Whitley, Starkweather, & Bogart, 1990c); but at the same time, there are supervised training paradigms that are significantly faster than back propagation. The application of genetic algorithms to the evolution of neural network topologies has also produced interesting results, but has been largely limited to small problems (Harp, Samad, & Guha, 1990; Whitley & Bogart, 1990a; Schaffer, 1990; Miller & Todd, 1989). There are domains, however, where genetic algorithms can make a unique contribution to neural network learning. In particular, because genetic algorithms do not require or use derivative information, the most appropriate applications are problems where gradient information is unavailable or costly to obtain. Reinforcement learning is one example of such a domain.

The goals of this article are twofold. Our first goal is to demonstrate how genetic algorithms can be used to train neural networks for reinforcement learning and "neurocontrol" applications (Werbos, 1989). As background, we review genetic algorithms and discuss the general problem of optimizing neural network weights using genetic algorithms. A variant of the genetic algorithm is used to optimize the weights of a neural network for the simulated control task of balancing an inverted pendulum. We have chosen this problem because it has been studied using a number of different approaches. The second goal of this article is to compare *genetic reinforcement learning* against another well-known method for training neural networks for reinforcement learning, the "Adaptive Heuristic Critic" (AHC) algorithm (Barto, Sutton & Anderson, 1983; Sutton, 1988; Anderson, 1987). The Adaptive Heuristic Critic (AHC) belongs to a class of adaptive critic reinforcement-learning algorithms

that rely upon both a learned evaluation function and a learned action function. Adaptive critic algorithms are designed for reinforcement learning with delayed rewards. The AHC algorithm uses the temporal difference method to train an evaluation network that learns to predict failure. The prediction is then used to heuristically generate plausible target outputs at each time step, thereby allowing the use of back propagation in a separate neural network that maps state variables to output actions. The comparisons offered in this article include not only learning rates but also performance tests designed to measure generalization, as well as graphs displaying the control behavior over time of specific individual networks in specific situations.

Although not often thought of in this way, genetic algorithms are, in a sense, inherently a reinforcement learning technique; the only feedback used by the algorithm is information about the relative performance of different strings representing different potential solutions. Genetic algorithms and reinforcement learning have been linked in the past, but the connection has largely been in the context of classifier systems. Classifier systems use genetic algorithms to optimize and discover simple pattern matching rules using a production system paradigm (Holland, 1986; Goldberg, 1989; Booker, Goldberg, & Holland, 1989). Grefenstette (1989, 1990, 1991) has also used genetic algorithms to optimize and discover sets of (symbolic) rules for sequential decision-making tasks. In the work reported here, genetic algorithms are not used for training classifier systems or other rule-based systems, but rather to train a neural network using only sparse feedback from the environment concerning performance.

Section 2 of this article briefly reviews genetic algorithms and highlights the major issues related to applying genetic algorithms to neural network weight optimization. The inverted pendulum problem is outlined in section 3, while section 4 delineates the genetic reinforcement learning paradigm. A short description of the Adaptive Heuristic Critic algorithm is presented in section 5. Results of comparative tests between the two approaches are reported in section 6.

2. Background: Genetic algorithms and neural networks

Traditionally, genetic algorithms search by manipulating a population of binary strings that represent different potential solutions, each corresponding to a sample point from the search space. The initial population is randomly generated. New sample points are generated by recombining information from two parent strings drawn from the current population. Consider the following binary strings: 1101001100101101 and yxyyxyxxyyyxyxxy, where x and y are used to represent 0 and 1. Using one or more randomly chosen "crossover points," recombination could occur as follows:

```
11010 \ / 01100101101 → 11010yxxyyyxyxxy
yxyyx / \ yxxyyyxyxxy → yxyyx01100101101
```

Another key element of genetic algorithms is the use of selective pressure to allocate reproductive trials. By allowing strings representing above-average solutions in the current population to "reproduce" more often than strings representing below-average solutions,

the genetic algorithm will allocate more trials to regions in hyperspace that tend to contain above-average solutions. Genetic algorithms are capable of performing a global search of a space because they can rely on hyperplane sampling to guide the search instead of searching along the gradient of a function.

When the string encoding is binary, the search space can be modeled as a binary hypercube. Different regions in hyperspace are represented by schemata. For example, the schema 0**1****** represents an "order-2" hyperplane, since two bits are specified in the schema. The " * " symbol represents a "don't care" operator. This hyperplane contains 25% of all strings in the search space: all strings with a 0 as the first bit and a 1 as the fourth bit. Recombination generates new sample points while preserving hyperplane information through inheritance. For example, if 10101100 and 11011110 are recombined without mutation, both parents and the offspring reside in the hyperplane 1***11*0. The fundamental theorem of genetic algorithms (Holland, 1975; Schaffer, 1987; Goldberg, 1989) demonstrates how it is possible to sample individual hyperplans and allocate either an increased representation in the population if they represent regions of above-average fitness or decreased representations for regions of below-average fitness. That the genetic search efficiently samples numerous hyperplanes in parallel is referred to as *implicit parallelism*. Theoretically, the result is a robust search method capable of searching nonlinear multimodal functions without using gradient information.

In all of our experiments, we use a version of the genetic algorithm we refer to as the GENITOR algorithm. The mechanics of the algorithm are as follows. A population of strings is randomly generated. Each string is evaluated and the population is sorted by ranking strings in terms of their evaluations. A random selection function with a linear bias towards the higher-ranked strings is used to stochastically choose two parents for recombination. The parents are recombined so as to produce a single offspring (i.e., two offspring are generated and one is randomly discarded). After the offspring is evaluated, it replaces the lowest-ranked string in the population and is inserted into its appropriate rank location. The differences between this and other genetic algorithms, as well as their comparative performance on several optimization problems, are discussed in Whitley and Kauth (1988) and Whitley and Hanson (1989).

Several researchers have attempted to apply genetic algorithms to neural network weight optimization problems. The most straightforward way of doing this is to encode each weight in the network as a binary substring; the entire binary encoding would be a string composed of these concatenated substrings. In our previous experiments using genetic algorithms with binary encodings, we have found that genetic algorithms that reply on recombination and limited mutation easily solve small neural network problems such as exclusive-or and 4-2-4 encoders. Given large populations (5000 strings) and a large number of recombinations (e.g., 1 million) we have been able to optimize more complex networks such as a network to add two two-bit numbers using four hidden units and a coding of 280 bits (eight bits per weight). Nevertheless, the resulting search time is dramatically slower than back propagation, and we have not been able to scale these results to significantly larger problems (Whitley & Starkweather, 1990b).

Our experiments suggest that one problem with genetic algorithms for larger neural network problems is that multiple symmetric representations exist for any single neural network. Recombining encodings for functionally dissimilar neural networks can result in

inconsistent feedback to the genetic algorithm in the form of inconsistent hyperplane samples. In other words, recombining two good but dissimilar networks can result in offspring that are not viable. For example, consider two small neural networks with two hidden nodes each. Both the networks may successfully learn to compute the same function mapping, but the two solutions may not be compatible because one network learns to compute subfunction A in hidden node $H1$ and subfunction B in hidden node $H2$, while the other network learns just the opposite assignment of functionality to the hidden units. Recombining functionally dissimilar strings will tend to produce offspring that redundantly represent some hidden nodes (functionality) and fail to represent the functionality of other hidden nodes. It is also not just the assignment of functionality to hidden units that may be different. The functionality can also be different in the sense that the sets of weights corresponding to two neural networks can also be scaled differently. Again, the recombination of networks with dissimilar sets of weights can result in offspring with poor performance. We refer to the problem of recombining dissimilar networks as the *structural/functional mapping problem* (Whitley et al., 1990c).

Some basic changes in the GENITOR algorithm resulted in the ability to handle relatively large neural net training problems; the results we have obtained on supervised learning problems (Whitley et al., 1990c) are similar to those reported by Montana and Davis (1989). Only three major implementation differences exist between the algorithms that have failed to optimize larger networks and those we have used to produce positive results. First, the problem encoding is a real-valued instead of binary. This means that each parameter (weight) is represented by a single real value and that recombination can only occur between weights. Second, a much higher level of mutation is used; traditional genetic algorithms are largely driven by recombination, not mutation. Third, a small population is used (e.g., 50 individuals). The use of small populations reduces the exploration of the multiple (representationally dissimilar) solutions for the same net. The stronger reliance on mutation also helps to avoid this problem, since no recombination is involved when mutation occurs.

Another way of trying to deal with the structural/function mapping problem is to try to identify the functionality of each hidden unit and then to swap similar hidden units. However, the experiments of Montana and Davis (1989) suggested there is little difference in performance when using operators that crossover weights or sets of nodes, or operators that attempt to extract hidden unit functionality and swap similar units. Therefore, we merely recombine the weights without attempting to restrict crossover or identify similar hidden units. Thus, our approach does not solve the structural–functional problem directly; instead, it bypasses the problem by using a small population and high mutation rates. Although the implementation details are not so different from conventional genetic algorithms, the empirical evidence suggests that the result is a type of stochastic hill-climbing algorithm that we refer to as a *genetic hill-climber* (Whitley et al., 1991). This algorithm has solved some large supervised learning problems (a neural network for signal detection with approximately 500 weighted connections) in approximately the same amount of time as back propagation (Whitley et al., 1990c).

The details of the real-coded genetic algorithm used in this article are given in figure 1. The use of adaptive crossover and mutation rates is again based on the work of Davis (1989). While it is somewhat unusual to include the crossover and mutation probabilities as an "allele" in the problem encoding, the idea was to have an expedient means of implementing

**** Initialization Phase: For each individual do the following ****

- Set all weights in the network to a random value between ± 2.5. Set one allele representing the probability of crossover to a random value between 0 and 1. Evaluate each individual and sort the population according to the fitness.

**** Iteration Phase ****

(1) Select two individuals according to relative fitness using linear-bias selection. Crossover with probability determined by the crossover probability allele of the string selected as parent 1; otherwise perform mutation on parent 1.

(2) The offspring always inherits the crossover probability of parent 1. If parent 1 has a higher fitness than the offspring, increment the offspring's crossover probability by a factor of 0.10 (to maximum 0.95); otherwise decrease the crossover probability by a factor of 0.10 (to minimum 0.05).

(3) Evaluate the new offspring and insert in the population according to fitness. Continue "Iteration" until error is acceptable or MAX-ITERATIONS = True.

**** Operators ****

- **Mutation:** Mutate all weights on the first selected individual by adding a random value with range ± 10.0.

- **Crossover:** Perform no crossover if the parents differ by two or fewer alleles. Otherwise, recombine the strings using one-point crossover between the first and last positions at which the parents have different weight values.

Figure 1. Implementation details of the real coded genetic algorithm.

adaptive operators. We conjecture that the exact nature of the adaptive operators is not critical as long as it has the effect that the algorithm automatically increases the probability of mutation when the population is converging. Note that crossover and mutation are mutually exclusive operators in the current implementation, so that the probability of crossover decreases as the population converges. We also note that the mutation operator is somewhat extreme: the offspring produced by mutation is a new random point located within a specific radius of the parent string.

While the genetic hill-climber appears to be roughly competitive with back propagation, it fails to be competitive when compared with faster supervised learning methods such as cascade correlation (Fahlman, 1990). Our tests on an adder for two-bit numbers and a large signal pulse detection problem show the genetic hill-climber to be much slower than cascade correlation. Nevertheless, genetic algorithms can make a unique and valuable contribution to the field of neural networks, especially for learning problems when gradient information is unavailable or costly to obtain. Currently, most neural networks are feed forward networks that use sigmoid transfer functions or radial basis functions. These choices make gradient information relatively easy to obtain. However, if one wishes to use other, more complex kinds of transfer units, such as "product units" or even splines—or if one

wished to train fully recurrent networks—then computing gradient information becomes far more costly. For these kinds of networks, genetic algorithms may represent a viable alternative. Similarly, training networks in situations where only sparse reinforcement is available is a difficult learning problem because one does not know in advance what the "correct" output of the network should be at each time step; therefore, the derivative information needed to drive gradient descent methods is not directly available. Most current methods for training neural networks using sparse feedback rely on a second network that either heuristically estimates plausible target outputs or that generates targets by back propagating errors through time. Both approaches involve considerable computational overhead.

3. Reinforcement learning for balancing an inverted pendulum

Control systems represent an important application for reinforcement learning algorithms because for some problems it is not feasible to analytically derive controllers; as a result, the training data needed for supervised learning such as simple back propagation may not be directly available. Werbos (1989) provides a brief review of neurcontrol learning paradigms; a more complete discussion is found in Barto (1990) and Werbos (1990). The inverted pendulum problem (which is also sometimes referred to as pole-balancing and cart-centering) is a classic control task. Several researchers have looked at ways of training neural networks for this problem. Of particular relevance are those methods that treat this as a reinforcement learning problem by using algorithms that rely on the relatively uninformative failure signal for feedback, such as Michie and Chambers (1968), Barto, Sutton, and Anderson (1983), Sutton (1988), Selfridge, Sutton, and Barto (1985) and Anderson (1987, 1989).

The inverted pendulum problem involves controlling an inherently unstable mechanical system of a cart and a pole that is constrained to move within a vertical plane. The objective is to keep the pole balanced and to avoid track boundaries. Training a neural network to solve the inverted pendulum problem cannot be directly accomplished using standard supervised training if we wish to use only performance information during training. Supervised training implies that for each input in the training set there is a known desired output. But in this problem we would like to learn a control strategy without knowing the desired output of the system at each time step in advance. Consider a sequence of actions on the cart, followed by a failure signal that means the pole has fallen. Let a 1 indicate a push to the right and a 0 to the left; "F" indicates failure. Consider the following sequence: 100011110000F. A classic credit assignment problem exists: which actions contributed to success and which actions contributed to failure?

At any time, the available state information includes the angle of the pole, θ, and the angular velocity of the pole, $\dot{\theta}$, as well as the position of the cart, ρ, and the velocity of the cart, $\dot{\rho}$. Using θ, $\dot{\theta}$, ρ, and $\dot{\rho}$ as inputs, the problem is to learn a decision policy for applying one of two actions to the cart at each time step: full-push left or full-push right, as in bang-bang control. The system is simulated by numerically approximating the equations of motion using Euler's method with a time step of $\tau = 0.02$ seconds and discrete time equations of the form $\theta(t + 1) = \theta(t) + \tau \dot{\theta}(t)$. The sampling rate of the system's state variables is the same as the rate of application of the control force, which is equal to 50 Hertz.

The equations of motion for the above system are as follows:

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

where

ρ is the cart position
$\dot{\rho}$ is the cart velocity
θ is the pole angle
$\dot{\theta}$ is the angular velocity of the pole
l is the length of the pole $= 0.5$ m
m_p is the mass of the pole $= 0.1$ kg
m is the mass of the cart and pole $= 1.1$ kg
F is the control force $= \pm 10$ N
g is the acceleration due to gravity $= 9.8 \text{m/sec}^2$

A failure signal is associated with θ falling past a particular angle or with the cart running into the ends of the track at ± 2.4 meters from center. The control problem is approximately linear for small values of θ, which is typically limited to a 12-degree range. Although the 12-degree limit is commonly used to generate a failure signal, some of our experiments attempt to balance the pole over a much larger range of angles and allows starting states with θ ranging up to ± 74 degrees.

4. Genetic reinforcement learning

Figure 2 illustrates the basic components of our genetic reinforcement learning paradigm. A real-valued string determines the weights in a neural network of predefined size and connectivity. The network is then applied to the reinforcement learning problem. The "system" could refer to either a real or simulated model of the task to be performed. In the pole-balancing problem, each real-value string in the population is decoded to form a network with five input units, five hidden units, and one output unit. The network is fully connected between the input layer and the hidden layer; the input layer is also fully and directly connected to the output unit. All five hidden units also feed into the output unit. This network configuration is the same as that used by Anderson (1989) with the AHC algorithm, thus facilitating comparisons. Since there are 35 links in the network, each string used by the genetic search includes (35 + 1) real values concatenated together.[1] Before any input is applied to the network, the four state variables are normalized between 0 and 1. A bias unit fixed at 0.5 is also used as a fifth input to the net; a weight from the bias unit to a hidden node (or output node) in effect changes the threshold behavior of that node. The action of the neural network for a particular set of inputs is determined from the activation of the output unit.

A random start state is supplied to the network, which determines the action on the cart. The new state of the system is then calculated and reintroduced as a new input to the net.

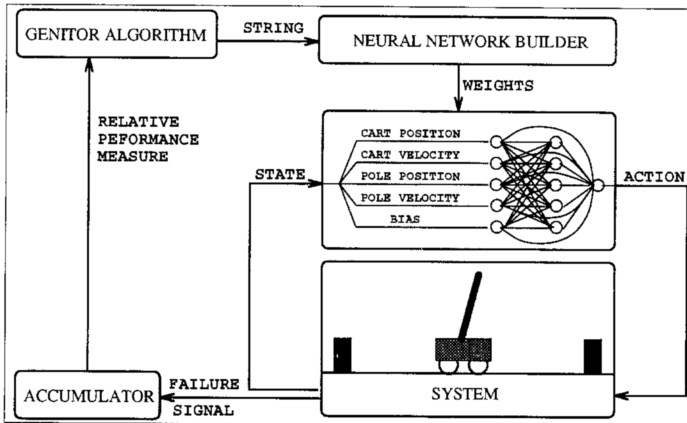


Figure 2. The genetic reinforcement learning paradigm.

This continues until a failure occurs. Feedback takes the form of an accumulator that determines how long the pole stays up and how long the cart avoids the end of the track; this is used as a relative measure of fitness.

Based on Anderson's earlier work (1989), we stopped learning when a network was found that was able to maintain the system without generating a failure signal for 120,000 time steps (40 minutes of simulated time). One potential problem with such a simple evaluation criterion is that a favorable or unfavorable start state may bias the fitness ranking of an individual net. In other words, the evaluation function is noisy. We would like to assign a fitness value to a string based on its ability to perform across all possible start states. In reality, this is not practical. In our initial experiments we started from one start state and interpreted the output action deterministically; the networks learned, but what was learned was not a full solution to the problem over all possible states. Networks starting from a favorable position may easily learn to keep the pole in position, but may not learn to deal with unfavorable conditions. We adopted the idea of interpreting the output action probabilistically from the AHC algorithms (Anderson, 1987) and found that this improved the performance of the controller. In other words, an output of +0.75 did not automatically mean that the action should be to push right, but rather that the probability of pushing right should be 0.75. This interpretation of the output action allows the network to visit more of the state space and hence to learn about more of the problem space.

The problem is to learn a total mapping policy from a limited sample of input training data that may be relatively uninformative. For genetic algorithms, this generalization problem is also a noisy "fitness" evaluation problem. Some sets of initial state variables guarantee failure for any sequence of control actions. Networks that receive poor starting states may be ranked lower than networks that receive good starting positions; some networks that represent good control policies will be lost. Methods to deal with this noise in the fitness evaluation function remain a topic of interest and are discussed later in this article. Fitzpatrick and Grefenstette (1988) show that the overall efficiency of a genetic algorithm may be improved by reducing the time spent evaluating each individual in favor of evaluating

more individuals from a larger population. As will be shown in later sections, not all of our methods or results are directly comparable to Fitzpatrick and Grefenstette's work; however our results do support the notion that increasing population size can be a robust mechanism to obtain better generalization.

4.1. Initial results using genetic reinforcement learning

Initial experiments were carried out with a failure signal occurring when the pole angle falls past 12 degrees or the cart hits the end of the track. In these experiments the "fitness" of a string is based on the accumulated number of steps until failure, starting from a single random initial state vector. The genetic algorithm did not have any difficulty with this problem: the system learned to balance the pole on every attempt, as shown in table 1.

Experiments with supervised learning problems indicate that the genetic algorithm's tendency is to converge more quickly but less reliably using smaller populations (Whitley et al., 1991). A similar trend exists for the inverted pendulum problem. While the genetic algorithm learns in every case, smaller populations tend toward faster learning (as indicated by the median), but with poorer worst-case behavior. This sometimes skews the mean and results in a higher variance. Data supporting these observations are given in table 1 for population sizes (PopSize) 5, 50, and 100.

We also ran a second set of 50 experiments to determine if we could improve the learning speed of the genetic algorithm using some form of population initialization. Grefenstette (1987) discusses an initialization strategy where the population is seeded with heuristically improved strings. Our seeding strategy involved loading the initial population with strings that have already demonstrated the ability to balance the pole for 100 time steps. This was done by randomly generating strings until enough strings satisfying the entry requirement are obtained. This strategy helped to eliminate extreme worst-case behavior for small populations. However, since seeding the initial population results in additional computation for most individual runs of the genetic algorithm, it uniformly increases the median learning time across experiments using different population sizes. The experimental results in tables 1 and 2 for populations of 5, 50, and 100 strings suggest that for this problem the genetic hill-climber is fairly robust without seeding. In subsequent tests we used the genetic algorithm with a random initial population.

Table 1. "Learning rates" as measured by Best, Median, Worst, and Mean refer to the number of "starts" (evaluations) required to reach the first occurrence of a net that is able to balance the pendulum for 120,000 time steps. SD is the standard deviation of the number of starts.

Genetic learning rates using random initial populations Pole angle ± 12 degrees			Sample size: 50		
PopSize	Best	Worst	Median	Mean	SD
5	242	50255	1845	4544	9650
50	478	6308	2580	2855	1238
100	886	11481	3579	4097	2205

Table 2. The effect of seeding on learning rate for the genetic algorithm using population sizes (PopSize) of 5, 50, and 100. The figures refer to the total number of strings evaluated, including those evaluated while seeding the population.

Genetic learning rates for seeding experiments Pole angle ±12 degrees Sample size: 50					
PopSize	Best	Worst	Median	Mean	SD
5	880	14,287	2657	3491	2752
50	7357	16,915	11,427	11,512	1896
100	14,592	25,655	20,321	20,244	2399

4.2. Other genetic approaches to control problems

The use of genetic algorithms for control applications have been studied by several researchers. In reviewing this work we have limited our attention to three kinds of approaches. First are approaches that use a genetic algorithm to directly associate actions with all possible states in a discretized mapping of the state space. Second are approaches where genetic algorithms are used to evolve control policies in the form of rules. Third are approaches using genetic algorithms to train neural networks for control application.

Odetayo and McGregor (1989) as well as Thierens and Vercauteren (1990) have applied genetic algorithms to the pole-balancing problem by discretizing the space in a way that is similar to earlier approaches used by Michie and Chambers (1968) and Barto et al. (1983). When genetic search is applied to this discretized problem, it is used to find an appropriate action for each state-space partition. The genetic encoding is merely a binary string, where each bit represents an action (push left or right) for each partition of the space. One notable point is that the entire binary vector does not have to be correct for the entire space to keep the pole balanced; the pole need only be started in a favorable position and kept in a favorable position. However, this represents a failure to generalize the control strategy to all portions of the input space. Also, for problems that have a large number of variables, it becomes infeasible to discretize the space. Consider a problem with 30 variables; even if each variable is binary, the discretized space of 2^{30} partitions is unreasonable.

The work of Grefenstette and collaborators (Grefenstette, Ramsey, & Schultz, 1990; Grefenstette, 1991) on SAMUEL (Strategy Acquistion Method Using Empirical Learning) is relevant to the work presented here for several reasons. First, SAMUEL is explicitly designed to develop a strategy for sequential decision-making. One problem to which SAMUEL has been applied is the *Evasive Maneuvers* game, involving an agent trying to avoid an approaching predator. Second, while SAMUEL uses a symbolic rule-based approach to learning rather than a neural network, it still uses a genetic algorithm as the main learning component of the system. SAMUEL and genetic reinforcement learning therefore face many of the same issues related to learning: how to initialize the population, the use of different selection strategies, the choice of genetic operators (crossover and mutation) and—probably most important—the problem of noisy fitness evaluation. Noisy fitness evaluation is very relevant to control problems because we are using the genetic algorithm to evaluate the control strategy, represented by an encoded string, as to its ability to function

for all possible inputs, while only actually testing the string on a small sample of cases. Whether the string is converted into a set of symbolic rules or connectionist weights is a secondary issue. While we have not compared our system to SAMUEL, such a comparison would be valuable.

More comparable to our own work are experiments carried out by Weiland (1990, 1991), who uses a standard generational genetic algorithm with a binary encoding of weights to train a fully recurrent network for several versions of the pole-balancing problem. Among these more difficult versions is a problem in which two poles are attached to the cart and simultaneously balanced; another problem involves balancing a jointed pole (i.e., one pole is attached to the top of another). These experiments also used the value of the output unit to determine the magnitude of the force applied to the cart.

The binary encoded genetic algorithms used by Weiland as well as the mapping of functionality to the recurrent network—and even the scaling of the binary encodings to the weights—are different enough that it is hard to compare Wieland's results with our own work. However, the limited success reported by Whitley, et al. (1990c) using binary encodings for the optimization of neural networks needs to be carefully reexamined in light of Weiland's results. We highly recommend Weiland's work to the reader interested in applying genetic algorithms to control problems.

5. AHC: Adaptive heuristic critic

In its search for a good network, the genetic reinforcement learning algorithm evaluates each candidate network by applying it to the balancing problem and measuring the time until failure. A new network is constructed after each balancing attempt. An alternative is to make small adjustments to one network after every step in a balancing attempt. The AHC algorithm (Barto et al., 1983; Sutton, 1988) takes this approach. Adjustments following each step are guided by information provided by a second network. This new network is called the evaluation network, while the original network that generates actions is called the action network. Figure 3 is a diagram of the evaluation and action networks. The action network has the same structure as the network in the genetic reinforcement learning model of figure 2. The evaluation network learns a function whose value indicates how often and how soon the inverted pendulum has failed after being in states similar to its current state. Thus, the AHC provides an evaluation of every state. A single action during a balancing attempt is rewarded or punished by comparing the evaluations of the state preceding an action and the state following the action. The difference between these evaluations is called a temporal-difference, or TD, error. Sutton (1988) defines a general class of TD methods for learning predictions. The AHC algorithm we used for our experiments is a member of this class. It is a general solution to the problem of distributing credit among the actions of a sequence that precedes a reinforcement. Sutton's analysis of linear temporal methods indicates that they are capable of learning a Markov model of the underlying system process. In the linear case there is a single "prediction-weight" associated with each state, and the output is a linear combination of the prediction-weight vector multiplied by the state vector. The result corresponds to the expected payoff associated with different absorbing states in the Markov model, where the payoff is adjusted to reflect the probabilities of entering

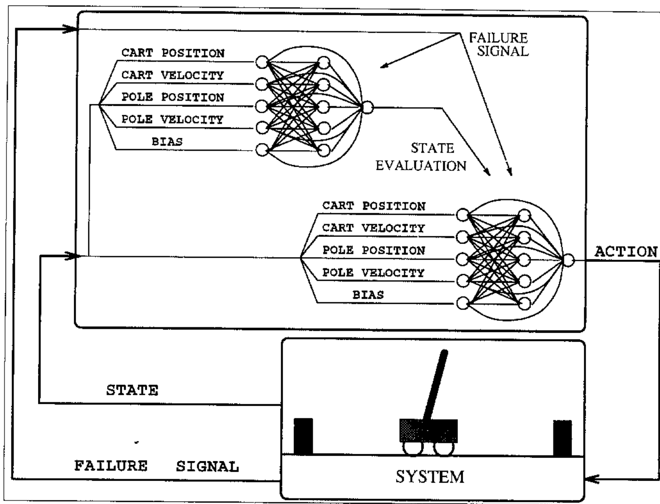


Figure 3. The adaptive heuristic critic learning paradigm.

the different absorbing states. Extending this to the nonlinear problem of predicting failure in the inverted pendulum problem, we would like the predictions associated with different state configurations to reflect the expected probability of making a sequence of moves that lead to failure, where a failure state is a terminal state with a payoff of −1 and all other states are nonterminal states with no payoff value.

After learning an approximately correct prediction of the failure signals, the evaluation network produces values between 0 and −1, with lower values indicating states from which failure is more likely. Barto et al. (1983) demonstated the use of the AHC for the pole-balancing problem when the cart-pole states are discretized. For the problem studied here without discretized states, reinforcement prediction as a function of cart-pole states is nonlinear. A multilayer AHC evaluation network with five hidden units in one hidden layer successfully learned this function. The hidden units of the evaluation network were trained using a modified error-propagation scheme where the error was a temporal-difference error. The AHC algorithm as described in Anderson (1987) is outlined in figure 4.

The action network has the same structure as the evaluation network, with five hidden units in one hidden layer. The hidden units of the action network are trained using a modified form of the back propagation algorithm (Anderson, 1987). Instead of having a predefined target output, the evaluation network is used to obtain evaluations of the current state variables. The difference between consecutive predictions in the evaluation network serves as a reinforcement of individual actions in the action network at each time step. The observed output of the action network is a continuous value giving the probability of pushing to the right. An action is chosen based on this probability. The difference between the actual action and the observed output is analogous to the difference between the target output and observed output terms normally used in back propagation. If the output action results in a state transition that reduces the prediction of failure, then the temporal difference error will be positive and the action will be positively reinforced. If the output action results

**** Initialization Phase ****

(1) Set each weight in the action network and the evaluation network to a random number from a uniform distribution from −0.1 to 0.1.

**** Iteration Phase ****

(2) Repeat until number of trials equals max-trials or number of steps equals max-steps:

(a) Forward pass through the evaluation network, calculating an evaluation, v_t, for the current state.
(b) Forward pass through the action network, calculating the probability, p_t, of pushing right. Using this probability an action is chosen, a_t, with value 0 or 1, representing pushes to the left or right, respectively.
(c) Apply action a_t to the pole simulation producing new pole state.
(d) Forward pass through evaluation network with new pole state, calculating an evaluation, v_{t+1}, for the new state.
(e) Calculate temporal difference between evaluations of new state and previous state. The temporal-difference error is $\gamma v_{t+1} - v_t$, where γ is a discount factor set to 0.9 in our experiments (c.f., Sutton 1988). If pole exceeds its bounds, $\gamma = 1$ and $v_{t+1} = -1$.
(f) Update the evaluation network's weights by back propagating the temporal-difference error. Normal back propagation calculates hidden nodes errors using the weights between the hidden nodes and the nodes they feed. This same calculation is applied, except only the signs of the weight value (± 1) are used.
(g) Update action network's weights by back propagating the product of the temporal-difference error and an action-comparison term given by $a_t - p_t$.
(h) If failure occurs, reset the pole state to random state and reset step counter to zero.

(3) Report results as the number of trials before a trial of max-steps has occurred.

Figure 4. The adaptive critic heuristic algorithm.

in a state transition that reduces the prediction of failure, then the temporal difference error will be positive and the action will be positively reinforced. If the output action results in a state transition that increases the prediction of failure, then the temporal difference error will be negative and the action is negatively reinforced. (For an alternative approach to reinforcement back propagation, see Ackley and Littman (1990).

The term "Adaptive Heuristic Critic" (AHC) is sometimes used in a more restrictive sense to refer only to the evaluation network that serves as a predictor of failure. However, the term has also been used in a more general sense to refer to the combined evaluation network and action network (e.g., Sutton, 1988, 1991). References to the "AHC algorithm" in this article also refer to the combined evaluation and action network when making comparisons to the genetic algorithm.

6. Comparative tests

We have attempted to compare genetic reinforcement learning to the adaptive-critic AHC architecture as used by Anderson (1987). It would be short-sighted to view these tests as

merely a way of deciding if one algorithm is better than another. Our goal is to study the behavior of the learned action networks under different circumstances and to discover the respective strengths and weaknesses in both algorithms. The algorithms are compared at two levels. First, the consistency, speed, and reliability of learning itself is evaluated. Second, once the networks have been trained, how well do they generalize? Put another way, how can we characterize the performance of the networks across the application domain after training is complete? Besides offering comparisons of learning rates and performance, we also look at ways of improving performance.

6.1. Comparative test 1: Learning at 12, 35, and 74 degrees

The experiments of most researchers only attempt to learn to balance the pole within the 12-degree range. Additional experiments were carried out with failure signals occurring at three different positions of the pole: 12 degrees, 35 degrees, and 74 degrees. In the implementation used here, changing the angle at which the failure signal occurs also changes the range of the input variable representing the pole angle. We are therefore learning using a different representation of state space. The 12-degree restriction means that the inverted pendulum problem has a solution that is approximately linear. At 35 degrees the problem is nonlinear and contains many start states where it is impossible to balance the pole; empirically, we have found it is possible to balance the pole for states up to 39 degrees from vertical, but only if other states variables are favorable. Our empirical tests suggests that the pole will always fall when the pole angle is beyond 45 degrees, even if all state variables are favorable. Thus, at 74 degrees the space is dominated by start states from which it is impossible to balance the pole.

One fact that emerged from these tests is that AHC sometimes fails to learn. The genetic algorithm converged to a solution in every experiment, regardless of whether the failure signal occurred at 12, 35, or 74 degrees. Learning rates (the number of trials required for learning) for tests with failure signal set to 12 and 35 degrees are given in table 3. AHC learned a control strategy in 33 out of 50 attempts (66%) on the 12-degree problem. We also ran experiments using both algorithms with the failure signal extended to 35 and 74 degrees to increase the difficulty of the problem. AHC successfuly converged in 43 out of 50 attempts for the 35-degree problem. Both 12- and 35-degree experiments were run out to a maximum of 45,000 trials. Additional batteries of tests composed of 50 runs for both 12- and 35-degree problems showed that the convergence rate continued to fluctuate

Table 3. Comparison of learning rates for the AHC algorithm and the genetic algorithm, GA-100 (population size 100), using 12- and 35-degree failure signals. Results are averaged over 50 experiments for the genetic algorithm, and only over those cases that converged for the AHC.

	Learning Rates at 12 degrees					Learning Rates at 35 degrees				
Method	Best	Worst	Median	Mean	SD	Best	Worst	Median	Mean	SD
AHC	3182	13543	4529	5433	2390	2106	12076	2758	3922	2452
GA-100	886	11481	3579	4097	2205	820	7221	4231	4206	1777

between 66% and 86%, but that the convergence rate is independent of the bounds on the pole angle. In all of our experiments, if the AHC algorithm failed to learn by 15,000 trials, it never converged no matter what version of the problem was being learned.[2]

When the problem was extended to use a failure signal at 74 degrees, the AHC algorithm converged to a successful solution in only 1 out of 50 attempts (learning time, 8057 trials) using up to 100,000 learning trials. On the 74-degree problem, the learning rates of the genetic algorithm using a population of 100 are as follows: best: 6055, worst: 37,032, median: 17,972, mean: 18,921, SD: 7355.

The differences in the learning rates reported here are not statistically significant, but the genetic algorithm is clearly competitive with the AHC algorithm. Because the AHC does not reliably converge to a solution, we conclude that the genetic algorithm is more robust with respect to the rate of convergence.

6.2. Comparative test 2: Performance-based generalization

The most straightforward test of performance is a test of the networks trained by the AHC algorithm and by genetic reinforcement learning across both random and fixed test points drawn as samples from the entire state space. Initially each system was tested for different "fixed" initial state variables, with each of the four normalized state variables having the following values: 0.05, 0.275, 0.50, 0.725, 0.95. This resulted in 625 different initial positions. We also compared the networks on 625 randomly chosen start positions and found similar results. All subsequent comparisons used the 625 randomly chosen positions. All initial states are chosen by assigning random values to the state variables within the normalized input range; thus, the initial pole angle is within the 12-degree (or 35-degree) bounds. It might come as a surprise to find that so many of the starting positions result in a failure even for those networks that display the best performance. It would appear that there are many initial positions that are irrecoverable for any control strategy. Not all AHC action networks were tested; rather, networks were tested only for those cases where the learning criteria was successfully met.

Table 4 provides statistics about the performance of the neural networks found by each method. Each network is tested starting at 625 different random initial states; we then count in how many of the 625 tests each neural network is able to balance the pole for 1000 time

Table 4. Performance is measured by the number of start states for which each network successfully balances the pole for 1000 time steps when tested over 625 random start states. Results are shown for both the 12- and 35-degree problems. The genetic algorithm using a population of 100 (GA-100) is compared to the AHC algorithm. Only those networks that converged for the AHC algorithm were used in these comparisons.

	Performance Metrics for 625 Starting Positions									
	12-deg. Failure Signal					35-deg. Failure Signal				
Method	Best	Worst	Median	Mean	SD	Best	Worst	Median	Mean	SD
AHC	372	70	196	192	79	406	19	296	271	108
GA-100	446	24	315	297	89	430	92	337	304	92

Table 5. Performance is measured by the number of start states for which each network successfully balances the pole for 1000 time steps when tested over 625 random start states. This table shows the effect on performance of varying the population size (PopSize) used by the genetic algorithm for the 12-degree problem.

PopSize vs. Performance at 12 degrees					
PopSize	Best	Worst	Median	Mean	SD
5	424	78	266	262	81
50	411	63	281	253	102
100	446	24	315	297	89

steps. This is enough time to either recover or fail in a difficult situation. The best networks are able to balance the pole from more than 400 of the 625 initial start states.

The same failure criteria used for training were also used for testing. Several things are interesting about the performance results. First, the data indicate that performance generally increased for the genetic algorithm using the larger population of 100 strings (see table 5). A population of 100 found the best overall net, as well as the most desirable mean and median case behavior. In all of our tests the genetic algorithm produced results than are competitive with the AHC algorithm, in terms of learning rates, convergence behavior, and performance.

6.3. Comparative test 3; Tracking control behavior over time

We developed several ways of tracking the control behavior of these networks over time. Obviously, one can animate the cart and pole system—and we did this. While animation was useful, we found that graphs that mapped the input and output behavior of the networks over time were much more informative. Figures 5 and 6 are made up of two different subgraphs. The top subgraph tracks the four *normalized* state variables (which serve as the neural network inputs) as a function of time. The bottom subgraph tracks the activation of the output unit, which is continuous. During training, the output is determined probabilistically depending on the activation of the output unit. During testing, the action applied to the system is obtained by deterministically thresholding the activation value of the output unit. If the activation value is greater than 0.5, then output a 1 and push right; if it is less than or equal to 0.5, then output a 0 and push left.

If the pole is vertical and the cart is centered and the velocities are 0, then all state variables will have the normalized value 0.5. When the system is started in an "ideal" state (all state variables are initialized to 0.5), then a successfully trained network will maintain the state variables close to the 0.5 level. It is not possible to balance the pole and avoid the track terminals from all possible initial states; however, when started from an initial state from which recovery is possible, a perfectly trained network should drive all state variables back to the 0.5 level representing the ideal state of the system.

In figures 5 and 6, the cart is at the far right end of the track with the pole leaning 32 degrees to the left; the 12-degree failure signal is disabled for these tests. The cart velocity and pole velocity are initialized at 0 (i.e., 0.5 is the normalized form). This initial state

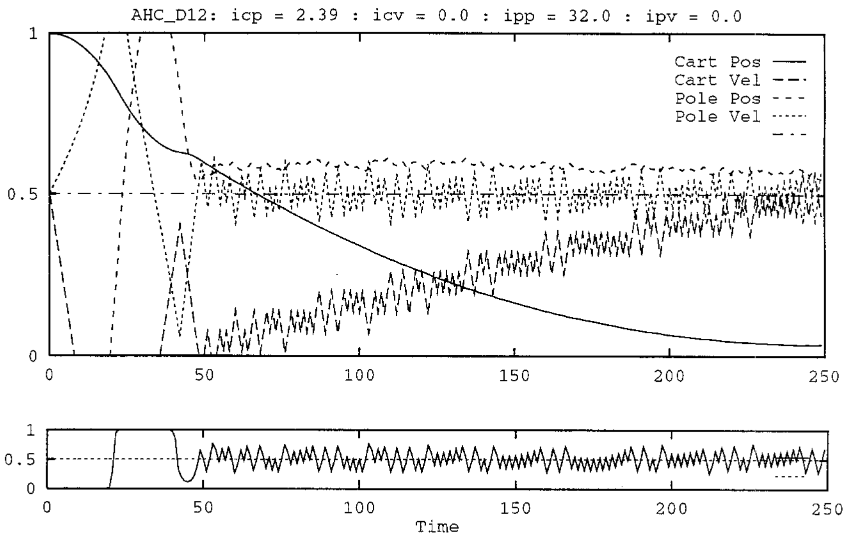


Figure 5. Control behavior of the best AHC-trained network for the 12-degree problem. The top subgraph tracks the four normalized state variables as a function of time. The bottom subgraph tracks the activation of the output unit, which determines the direction the cart is pushed. The cart is initially positioned at the far right end of the track with the pole leaning 32 degrees to the left. Since the 32-degree starting angle of the pole exceeds 12 degrees, the plot of the pole angle is initially beyond the range used in this graph. The initial cart and pole velocities are both set to zero.

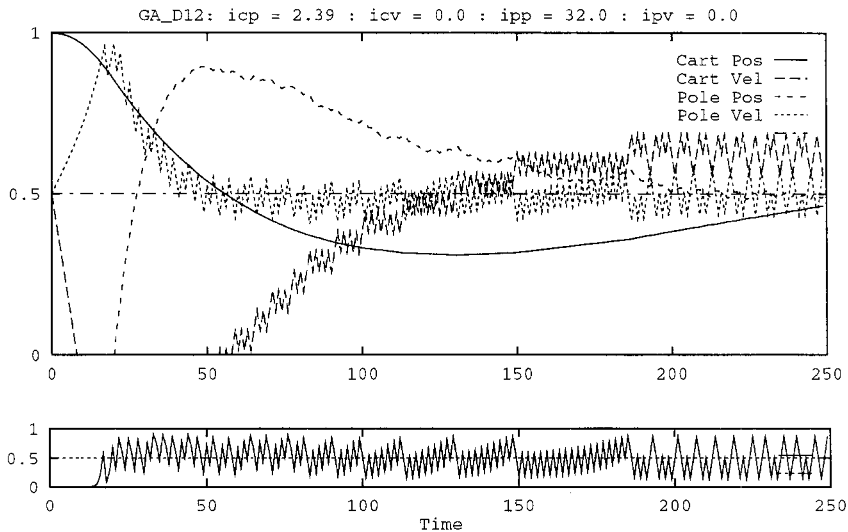


Figure 6. Control behavior of the best genetically trained network for the 12-degree problem. The cart is initially positioned at the far right end of the track with the pole leaning 32 degrees to the left.

constitutes a position from which it is difficult for the system to recover. Both the AHC-trained network and the genetically trained network used to produce these graphs are the best networks obtained for the 12-degree problem, as indicated by the performance tests summarized in table 4. Figure 6 shows that the genetically trained network gets all of the input variables into tolerable ranges fairly quickly, whereas the AHC-trained network (figure 5) takes longer. The AHC-trained network quickly dampens pole velocity and reduces oscillation in the pole position, but in doing so the cart almost crashes into the opposite end of the track. The genetically trained network handles problems with starting pole angles beyond 32 degrees, but the AHC-trained network does not. Results from the next section help to explain this behavior.

Figure 7 and figure 8 show results for an AHC-trained network and a genetically trained network using a failure signal at 35 degrees during learning. Again we use the best networks according to the performance data reported in table 4. These graphs indicate that both the AHC-trained network and the genetically trained network exploit similar information to determine the output activation levels and that they employ similar control strategies. The networks trained at 35 degrees proved to be more similar across a wider range of start states, but as the difficulty of the initial start states is increased the AHC-trained networks fail sooner than the genetically trained networks. In these graphs, the system is started with the cart in the same far right position and the pole leaning 34 degrees to the left. Cart velocity and pole velocity are initially 0.

These graphs make it evident that both networks track pole velocity by varying the magnitude of the output value. The correlation between pole velocity and the output activation is not as discernible in the first 50 to 100 time steps because the system is recovering from a difficult initial situation; correlation between the pole velocity and the output activation

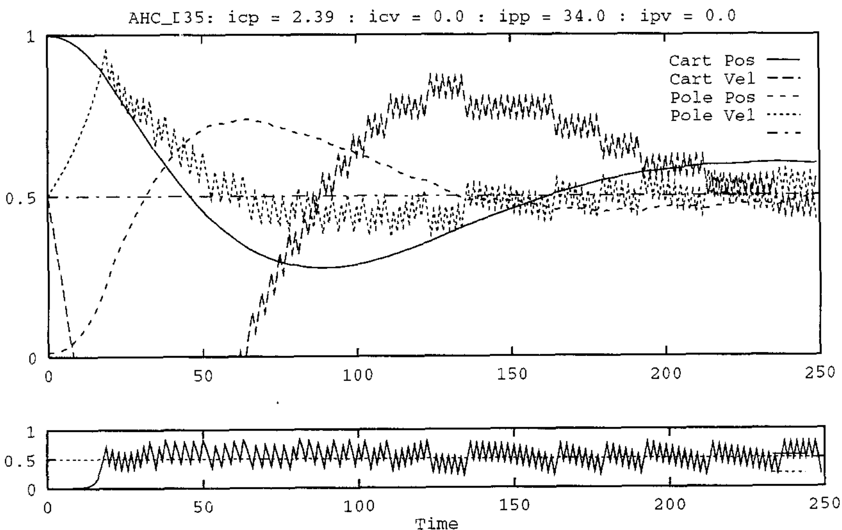


Figure 7. Control behavior of the best AHC-trained network for the 35-degree problem. The cart is initially positioned at the far right end of the track with the pole leaning 34 degrees to the left.

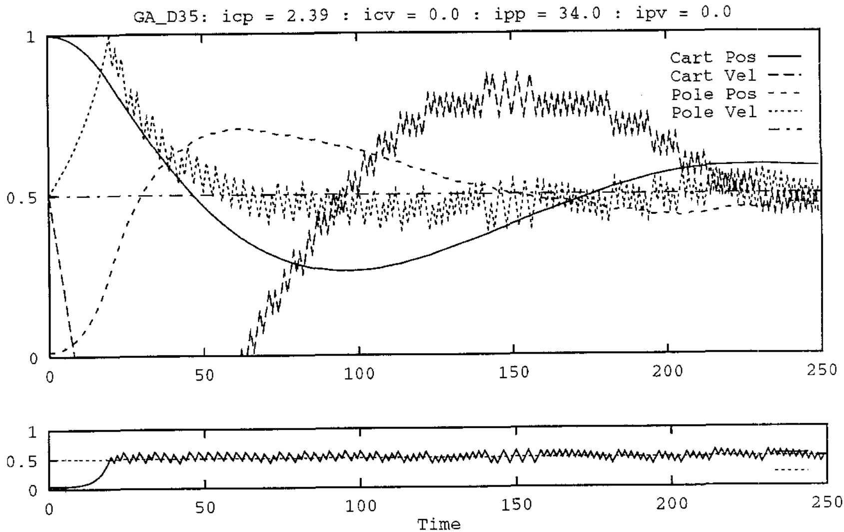


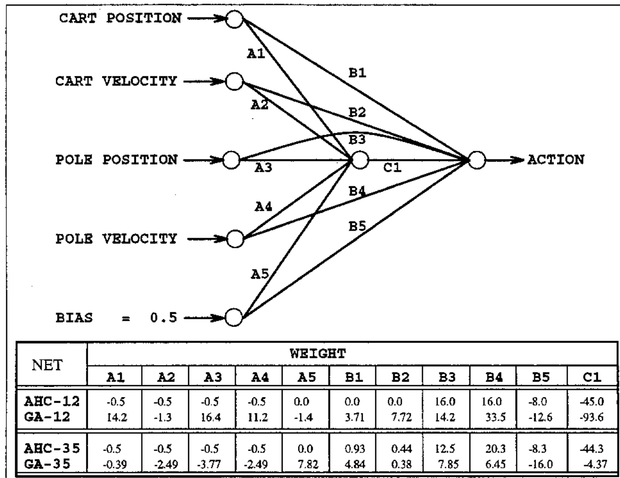
Figure 8. Control behavior of the best genetically trained network for the 35-degree problem. The cart is initially positioned at the far right end of the track with the pole leaning 34 degrees to the left.

is much more pronounced as the networks begin to bring the system under control. Also notable is that cart velocity and pole velocity tend to be negatively correlated. Given the input definitions used in our experiments, cart velocity and pole velocity have a similar, but opposite relationship. In the next section we prune the neural networks in order to isolate possible functional relationships between the state variables and the output unit's activation.

6.4. Comparative test 4: Pruning networks to find critical variables

To better understand the control strategies developed by the two algorithms, we attempted to reduce several successful networks to their minimal form. One way to prune a network is to remove any hidden unit whose activation is approximately constant for all inputs. Such a hidden unit can be eliminated by combining it with the bias unit (which always has a constant activation value). The weights that fan out from the eliminated unit are absorbed by appropriately modifying the weights that fan out from the bias unit. Another way to reduce the number of hidden units in the network is to combine two units with similar output behavior. A hidden unit p can be removed when its output is approximately equal to the output of another hidden unit q for all input pattern. For all units r, which receive input from both p and q, the value of the weight w_{pr} is added to the weight w_{qr}. The unit p can now be eliminated along with all weights associated with it. Sietsma and Dow (1991) give a more detailed explanation of these kinds of methods that allow a user to hand-prune a neural network.

The networks that were reduced are the best networks from table 4 for a failure signal of both 12 degrees and 35 degrees. In each case we were able to reduce the number of



NET	WEIGHT										
	A1	A2	A3	A4	A5	B1	B2	B3	B4	B5	C1
AHC-12	-0.5	-0.5	-0.5	-0.5	0.0	0.0	0.0	16.0	16.0	-8.0	-45.0
GA-12	14.2	-1.3	16.4	11.2	-1.4	3.71	7.72	14.2	33.5	-12.6	-93.6
AHC-35	-0.5	-0.5	-0.5	-0.5	0.0	0.93	0.44	12.5	20.3	-8.3	-44.3
GA-35	-0.39	-2.49	-3.77	-2.49	7.82	4.84	0.38	7.85	6.45	-16.0	-4.37

Figure 9. The pruned forms of the neural networks produced by the AHC and genetic algorithms.

hidden units down to one; this pruning still allowed performance to remain within $\pm 1\%$ of performance before pruning as determined using the 625 fixed test points.

Figure 9 shows the reduced networks. An examination of numerous networks trained by the AHC algorithm reveals that the action networks always have a similar weight pattern. Weights leading from the input layer to the hidden layer are always close to the -0.5 range (see connections A1–A4). Larger weights are typically found on connections that feed into the output, particularly the direct input/output connections (e.g., B3–B5). This suggests that the computation of the AHC networks may be dominated by these linear associations. The weights for the genetically trained networks were more difficult to analyze and prune.

The weights in figure 9 suggest that the genetically trained network for the 12-degree problem uses more information about cart velocity and position than the AHC trained network (where B1 and B2 are 0). To further test this hypothesis, we started a cart on the far right of the track with the pole angle at 36 degree to the left and a high cart velocity to the left. We also reset the failure signal to occur at ± 90 degrees. The networks trained at 12 degrees were than tested. The genetically trained network got the pole up using the favorable velocity, then got the velocity under control, thereby recovering from a difficult situation. The AHC-trained network was able to get the pole up, but failed to control the velocity of the cart, which crashed into the left track terminal. None of the AHC networks trained at 12 degrees could recover if the pole angle fell past 32 degrees.

The graphs in section 6.3 suggest that the computation of the genetic trained networks and the AHC-trained networks are similar, but also indicate that the genetically trained networks are better able to recover from extreme initial states. The weights obtained by pruning the neural networks suggest that this difference may be due to better exploitation of state information about cart position and velocity.

6.5. Comparative test 5: A tougher learning criteria

Both the genetic and AHC algorithms show a good deal of variability in what is learned from one experiment to the next. This is perhaps not surprising, considering the stopping criteria. Controlling the inverted pendulum so that it successfully balances for 120,000 time steps from a single random starting position does not represent a consistent metric of success from one experiment to the next because the ability to balance for 120,000 time steps is influenced by the initial state vector. There is some evidence to suggest that action networks from the experiments with the fastest learning times have poorer performance. Sammut and Cribb (1990) conjecture that in general that there is a tradeoff between learning rate and the ability to generalize. Specifically, they found experimentally that programs that very rapidly learned to perform a task resulted in very specific control strategies that could not be transferred to other initial start states.

Since the fastest learning times may be correlated with fortuitious starting positions, a stopping criteria that should be more indicative of overall performance was chosen. Both the AHC and genetic algorithms were stopped only after demonstrating success at the learning task multiple times. The first increment in the stopping criteria was to do two consecutive balancing attempts successfully for 120,000 steps (the "2/2 rule"). Each additional attempt to balance the pole is carried out from a new random initial start state. This appears to result in improved performance behavior for the genetic algorithm, but no clear trend emerged for the AHC algorithm. The stopping criteria was further extended to three and four consecutive balances (the "3/3 rule" and the "4/4 rule"). As can be seen in table 6, an additional improvement in mean performance and reduced variance is again observed for the genetic algorithm.

Using a more difficult stopping criteria resulted in modest increases in training time. But the data in table 6 makes it clear that simple comparisons of learning speed can sometimes be very misleading if the effect on performance is not also considered. Using the genetic algorithm, the mean performance of the resulting networks improved and the performance variance decreased as the stopping criteria became more challenging. One other notable point, which we have not explored, is the potential for further improving the performance

Table 6. Results of using a stricter stopping criteria. The genetic algorithm (GA-100) uses a population size of 100.

Using Different Stopping Criteria at 12 degrees								
	Learning Rates				Performance Metrics			
Method	Best	Mean	Median	SD	Best	Mean	Median	SD
AHC 1/1 rule	3182	4529	5433	2390	372	196	191	79
AHC 2/2 rule	3190	4628	5331	2437	377	183	193	88
AHC 3/3 rule	3600	4740	5204	1754	324	192	188	79
AHC 4/4 rule	3228	4702	5384	1763	369	175	180	72
GA-100 1/1 rule	886	3579	4097	2205	446	315	291	89
GA-100 2/2 rule	689	4760	5148	3323	425	328	307	82
GA-100 3/3 rule	232	4720	5226	2727	445	330	326	66
GA-100 4/4 rule	1526	5507	7071	7602	428	371	360	46

of the genetically trained networks by using a larger population size during learning. However, much larger population sizes could aggravate the *structural/functional mapping problem* discussed earlier in this article.

Convergence rates continued to fluctuate between 66% and 86% for the AHC using different stopping criteria. The criteria for stopping do not appear to be factors in the probability of convergence, however. We found *if an AHC network balanced the pendulum once, it always continued learning to a more difficult criterion.* We again note that all of the AHC-trained networks that we have developed either learn before 15,000 evaluations or entirely fail to learn. The fact that longer training times did not help with the convergence problem may also mean that it is not beneficial to lengthen training time by using a stricter stopping criterion.

7. Genetic algorithms and noisy fitness functions

As one might expect there appears to be a relationship between larger population sizes and more reliable control functions being learned by the genetic algorithm. Work by Fitzpatrick and Grefenstette (1988) on an image registration problem using a noisy evaluation function also indicates that fewer samples are needed to estimate the "correct" fitness of a string using a noisy evaluation when larger populations are used. This is because in the larger populations we expect more schemata samples; thus the effect of the noise on the fitness of each (implicitly estimated) schema is averaged out at the schema level rather than at the string level. This implies that a tradeoff exists between relying on a smaller number of more accurate (and costly) evaluations versus relying on a larger number of less accurate (and less costly) evaluations. Grefenstette et al. (1990) report similar findings in their application of genetic learning to the *evasive maneuvers* control problem.

When an evaluation function is noisy, it may be possible to average the performance from several initial state vectors to obtain a more accurate evaluation. We therefore thought that the performance of the genetic algorithm could perhaps be improved for the inverted pendulum problem by averaging fitness over three random start positions. The results were somewhat surprising. There was no improvement in performance, and the training time more than doubled (although the total number of "fitness evaluations" was reduced). The control policies of the resulting action networks were no more robust than controllers trained normally. One possible hypothesis that could explain our results relates to the interpretation of the output of the network as a probabilistic value. By interpreting the output of the network probabilistically, we presumably visit more of the state space, which is perhaps analogous to averaging over multiple starts using a deterministic interpretation of the output. Experiments to test this hypothesis could be designed and would provide valuable information. Overall, the use of the stricter stopping criteria was more effective at improving performance than averaging fitness over multiple tests of the neural net.

Several factors complicate the current experiments so that it is more difficult to consider the effects of noise on schema samples. First, it should be noted that the image registration work (Fitzpatrick & Grefenstette, 1988) was done in the context of a standard genetic algorithm using generational replacement; as recently noted by Davis (1991), the effects of noisy evaluation in the context of a one-at-a-time replacement scheme remains an open issue.

In a generational model, strings are evaluated each generation; if the evaluation function is noisy, a more "correct" estimate of the average value of strings in a particular hyperplane region can be obtained by using larger populations to average out the noise and by generational evaluation to reevaluate the strings. The use of the GENITOR algorithm with one-at-a-time reproduction and replacement could pose a potential problem, because once a string is evaluated, it is ranked in the population and never reevaluated. If the fitness function is noisy, as it is in this case, this means the ranking will include errors. However, in this particular application, noisy evaluation did not prevent the algorithm from learning. We believe this is because the noise largely had a conservative effect: some good networks are lost because of poor start states, but it is more difficult for a poor net to obtain a good ranking.

8. Discussion and conclusions

There are several differences in the information used by the AHC and genetic algorithms, at least in the learning component. In both cases, the action network requires input information about the current state in order to produce an output. For the AHC algorithm, the evaluation network also requires state information to learn, since without state information there is no prediction of failure. But the genetic algorithm does not require state information at each time step; it only needs feedback about how long the pole stayed up in order to rank competing sets of weights. The reinforcement back propagation that occurs in the AHC algorithm means that learning continues even when failures are not occurring. Furthermore, the use of the temporal difference method means that the evaluation network is also being updated, even when failures are not occurring. In the genetic approach used here, updates to the action network occur only after one or more failures: learning is not continuous. Another difference is that the genetic approach used in our experiments will assign an equal evaluation to two networks that avoid failure for an equal number of time steps. However, the AHC algorithm evaluates networks by the trajectory of states that are experienced. The evaluations associated with any two networks would differ when the AHC training algorithm is used, favoring the network that drives the cart-pole through more highly valued states. The restriction of the search to highly valued states may also explain why the performance of AHC trained networks did not improve when the stricter stopping criterion was used.

Yet another difference between the genetic algorithm and the AHC algorithm is the lack of success using the AHC algorithm when the failure signal occurs at wider pole bounds. This possibly results from the combination of the incremental learning algorithm used to adjust the weights and the way the AHC evaluation network generalizes. A good prediction of failure is hard to learn when the majority of start states lead to failure. Without a good failure prediction function, a successful control strategy cannot be learned using the reinforcement-driven back propagation. The genetic algorithm, because it ranks each network based on performance, is able to ignore those cases where the pole cannot be balanced; only the successful cases will obtain the chance to engage in genetic reproduction. For the AHC evaluation network, however, the preponderance of failures may cause all states to overpredict failure. This problem can be corrected either by selectively sampling the

space to extract a better balance of success and failure information or by tuning the AHC algorithm to place more emphasis on positive results and less on failure.

Our purpose in the current study is (1) to demonstrate that genetic hill-climbing algorithms can be used for training neural networks for control problems, and (2) to examine methods for comparing the learning behavior and performance of algorithms for neurocontrol problems. The differences between genetic reinforcement learning and reinforcement learning using other methods must be explored more carefully, especially in other application domains. Anderson and Miller (1990) discuss several challenging control problems that could provide an initial test bed. We are also interested in ways in which these two different algorithms might be combined or hybridized. Recently, Ackley and Littman (1991) have combined genetic methods and neural networks for control problems in a different way. They use a genetic algorithm to train an evaluation net, then use the output of the evaluation net to do reinforcement error propagation on the action net. This represents a strategy that is intermediate between the methods compared in this article.

The results that we report here represent just one approach to genetically train neural networks for control problems. These are numerous questions that remain to be answered. There are other forms of adaptive critics, such as Q-learning algorithms (Watkins, 1989; Sutton, 1991), to which we could compare results. There is also a critical need to test different approaches across a suite of test problems that display varying levels of difficulty. Overall, we are very encouraged by the results obtained so far. The comparative methods we have used in this article also highlight the need for a broadly based comparative perspective when studying different reinforcement learning algorithms for control problems.

Acknowledgments

This research was supported in part by NSF grant IRI-9010546 and in part by a grant from the Colorado Institute of Artificial Intelligence (CIAI). CIAI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the State of Colorado.

Notes

1. The extra real value encodes the crossover probability of the string.
2. In the final stages of preparing this article, we encountered one AHC-trained network for a 35-degree problem that converged after 39,000 trials; this was the only such occurrence out of several hundred tests.

References

Ackley, D., & Littman, M. (1990). Generalization and scaling in reinforcement learning. In D. Touretzky (Ed.), *Advances in neural information processing systems* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Ackley, D., & Littman, M. (1991) *Interactions between learning and evolution*. Morristown, NJ: Cognitive Science Research Group, Bellcore.

Anderson, C.W. (1987). *Strategy learning with multilayer connectionist representations* (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.

Anderson, C.W., & Miller, W.T. (1990). A challenging set of control problems. In T. Miller, R. Sutton, & P. Werbos (Eds.), *Neural networks for control*. Cambridge, MA: MIT Press.

Barto, A.G. (1990). Connectionist learning for control. In T. Miller, R. Sutton, & P. Werbos (Eds.), *Neural networks for control*. Cambridge, MA: MIT Press.

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

Booker, L., Goldberg, D., & Holland, J. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, *40(1–3)*, 235–282.

Davis, L. (1989). Adapting operator probabilities in genetic search. *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 61–69). Fairfax, VA: Morgan Kaufmann.

Davis, L. (1991). *The handbook of genetic algorithms*. New York: Van Nostrand Reinhold.

Fahlman, S., & Lebiere, C. (1990). The cascade correlation learning architecture. In D. Touretzky (Ed.), *Advances in neural information processing systems* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Fitzpatrick, J., & Grefenstette, J. (1988). Genetic algorithms in noisy environments. *Machine Learning*, *3(2–3)*, 101–120.

Goldberg, D. (1989). *Genetic algorithms in search, optimization and machine learning*. Reading, MA: Addison-Wesley.

Grefenstette, J. (1987). Incorporating problem specific knowledge into genetic algorithms. In L. Davis (Ed.), *Genetic algorithms and simulated annealing*. London: Pitman/Morgan Kaufmann.

Grefenstette, J. (1989). A system for learning control strategies using genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms* (p. 183–190). Fairfax, VA: Morgan Kaufmann.

Grefenstette, J. (1991). Strategy acquisition with genetic algorithms. In L. Davis (Ed.), *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

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

Harp, S., Samad, T., & Guha, A. (1990). Designing application-specific neural networks using the genetic algorithm. *Neural Information Processing Systems* (Vol. 2). San Mateo, CA: Morgan Kaufman.

Holland, J. (1975) *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press.

Holland, J. (1986). Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-basd systems. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine learning* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Michie, D., & Chambers, R. (1968). BOXES: An experiment in adaptive control. In E. Dale & D. Michie (Eds.), *Machine intelligence* (Vol. 2). Edinburgh: Oliver and Boyd.

Miller, G., Todd, P., & Hegde, S. (1989). Designing neural networks using genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 379–384). Fairfax, VA: Morgan Kaufmann.

Montana, D., & Davis, L. (1989). Training feedforward neural networks using genetic algorithms. *Proceedings of the 1989 International Joint Conference on Artificial Intelligence* (pp. 762–767).

Odetayo, M.O., & McGregor, D.R. (1989). Genetic algorithm for inducing control rules for a dynamic system. *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 177–182). Fairfax, VA: Morgan Kaufmann.

Sammut, C., & Cribb, J. (1990). Is learning rate a good performance criterion for learning. *Machine Learning: Proceedings of the 7th International Conference* (pp. 170–178). San Mateo, CA: Morgan Kaufmann.

Schaffer, D. (1987). Some effects of selection procedures on hyperplane sampling by genetic algorithms. In L. Davis (Ed.), *Genetic algorithms and simulated annealing*. London: Pitman/Morgan Kaufmann.

Schaffer, J.D., Caruana, R.A., & Eshelman, L.J. (1990). Using genetic search to exploit the emergent behavior of neural networks. *Physica D*, *42*, 244–248.

Selfridge, O.G., Sutton, R.S., & Barto, A.G. (1988). Training and tracking in robotics. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 437–443). San Mateo, CA: Morgan Kaufmann.

Sietsma, J., & Dow, R. (1991). Creating artificial neural networks that generalize. *Neural Networks*, *4*, 67–79.

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

Sutton, R. (1991). Reinforcement learning architectures for animats. In J. Meyers & S. Wilson (Eds.), *Simulation of adaptive behavior: From animals to animats* (pp. 288–296). Cambridge, MA: MIT Press.

Thierens, D., & Vercauteren, L. (1990). A topology exploiting genetic algorithms to control dynamical systems. In H.P. Schwefel & R. Manners (Eds.), *Parallel problems solving from nature* (pp. 104–108). Springer/Verlag.

Watkins, C. (1990). *Learning with delayed rewards*. Ph.D. dissertation, Psychology Department, Cambridge University, Cambridge, England.

Weiland, A. (1990). Evolving controllers for unstable systems. In D. Touretzky, J. Elman, T. Sejnowski & G. Hinton (Eds.), *Connectionist models: Proceedings of the 1990 Summer School* (p. 91–102). San Mateo, CA: Morgan Kaufmann

Weiland, A. (1991). Evolving neural network controllers for unstable systems. *1991 International Joint Conference on Neural Networks, 2* (pp. 667–673). Seattle.

Werbos, P. (1989). Backpropagation and neurocontrol: A review and prospectus. *1989 International Joint Conference on Neural Networks, 1* (pp. 209–215). Washington, DC.

Werbos, P. (1990). A menu of designs for reinforcement learning over time. In T. Miller, R. Sutton, & P. Werbos (Eds.), *Neural networks for control*. Cambridge, MA: MIT Press.

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

Whitley, D., & Hanson, T. (1989). Optimizing neural nets using faster, more accurate genetic search. *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 391–396). Fairfax, VA: Morgan Kaufmann.

Whitley, D., & Bogart, C. (1990a). The evolution of connectivity: Pruning neural networks using genetic algorithms. *1990 International Joint Conference on Neural Networks, 1* (p. 134–137). Washington, DC.

Whitley, D., & Starkweather, T. (1990b). Optimizing small neural networks using a distributed genetic algorithm. *1990 International Joint Conference on Neural Networks, 1* (pp. 206–209). Washington, DC.

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

Whitley, D., Dominic, S., & Das, R. (1991). Genetic reinforcement learning with multilayered neural networks. In R. Belew & L. Booker (Eds.), *Proceedings of the 4th International Conference on Genetic Algorithms* (pp. 562–569). San Diego, CA: Morgan Kaufmann.

Received November 21, 1991
Accepted March 24, 1992
Final Manuscript July 23, 1992