Evolving artificial neural networks to control chaotic systems

Eric R. Weeks* and John M. Burgess†
Center for Nonlinear Dynamics and Department of Physics, University of Texas at Austin, Austin, Texas 78712

(Received 7 April 1997)

We develop a genetic algorithm that produces neural network feedback controllers for chaotic systems. The algorithm was tested on the logistic and Hénon maps, for which it stabilizes an unstable fixed point using small perturbations, even in the presence of significant noise. The network training method [D. E. Moriarty and R. Miikkulainen, Mach. Learn. **22**, 11 (1996)] requires no previous knowledge about the system to be controlled, including the dimensionality of the system and the location of unstable fixed points. This is the first dimension-independent algorithm that produces neural network controllers using time-series data. A software implementation of this algorithm is available via the World Wide Web. [S1063-651X(97)05308-7]

PACS number(s): 05.45.+b, 07.05.Mh

I. INTRODUCTION

Chaotic behavior in a dynamical system can be suppressed by periodically applying small, carefully chosen perturbations, often with the goal of stabilizing an unstable periodic orbit of the system [1–3]. Control of this type has been successfully applied to many experimental systems [3–7]. In this paper we demonstrate a robust method of training neural networks to control chaos. The method makes no assumptions about the system; the training algorithm operates without knowledge of either the dimensionality of the dynamics or the location of any unstable fixed points. The structure of the controller is not fixed and the neural network is free to adopt nonlinear forms.

After reviewing previous work in Sec. II, we present the details of our method in Sec. III. We use a modified version of Symbiotic Adaptive Neuro-Evolution (SANE), developed by Moriarty and Miikkulainen [8–10]. This method uses genetic algorithms to create neural networks with desirable characteristics, in this case the ability to stabilize unstable fixed points of a Poincaré map. SANE has proved to be fast and efficient in a variety of cases unrelated to chaos control [8–10]. In Sec. IV we present results for control of the one-dimensional logistic map and the two-dimensional Hénon map. In Sec. V we discuss extensions of our method and conclusions.

II. PREVIOUS WORK

A commonly applied method for control of chaotic dynamical systems was discovered by Ott, Grebogi, and Yorke (OGY) [1]. The OGY method requires an analytical description of the linear map describing the behavior near the fixed point [2]. This map is used to determine small perturbations that, when periodically applied, use the system's own dynamics to send the system towards an unstable fixed point. Continued application of these perturbations keeps the system near the fixed point, thereby stabilizing the unstable fixed point even in a noisy system.

The OGY method generally is inadequate when the system is far from the fixed point and the linear map is no longer valid. The original OGY method is also limited to controlling only one- or two-dimensional maps. However, the method has been extended to higher dimensions [11–13] and to cases where multiple control parameters are available [14].

Several other analytical methods exist for controlling chaos. Hunt developed the method of occasional proportional feedback, a modification of the OGY algorithm [5]. Pyragas developed a method that provides small control perturbations for continuous systems [15]. This elegant method uses a linear feedback based on an external signal or time-delayed feedback and requires little analytical preparation. Hübler reported on the ''dynamical key'' method in which natural system dynamics may be time reversed and converted into perturbations to drive the system back to an unstable fixed point [16]. Petrov and Showalter presented a nonlinear method that relies on the construction of an extended-dimension, nonlinear control surface (effectively a lookup table) by taking into account the final state of the system after the application of a perturbation [17].

An alternative to analytical control algorithms involves the use of neural networks to discover (possibly novel) control techniques. Neural networks are well known for providing solutions to some complex problems, even when an analytical solution cannot be found [18]. Neural networks have been used to control chaos in various systems, including the logistic map [19–21], the Hénon map [22,23], other maps [23], and continuous systems [24–27]. Several of these methods require the perturbation to be large [21,23–26] in contrast to methods such as the OGY algorithm. In some methods additional computation is required, such as preprocessing to find the algorithm to train the network [22,26,27], postprocessing to translate the output of a network into a desired perturbation [20,21], or an additional ''switch'' to activate the network when the system is close to the fixed point [20,23]. In some cases, the location of the fixed point must be precisely specified [19,20,22,27]. In other cases, target dynamics such as a limit cycle must be specified [21,25]. In one case all of the system variables must be available and controllable, although several different maps were controllable by this algorithm [23].

*Electronic address: weeks@chaos.ph.utexas.edu
†Electronic address: jburgess@chaos.ph.utexas.edu

In this paper we present a method that combines the advantages of several of these methods. Our method uses small perturbations to stabilize unstable fixed points in chaotic maps with no additional computation required before or after the network has been trained. The training algorithm does not know the correct perturbations that need to be applied, does not use the location of the target state, and is dimension independent. In addition, we use an algorithm [8] that provides for great flexibility in specifying the network topology. The method is easily extended to systems with all system variables known, with the location of a fixed point known, or with multiple control parameters available. Previous methods often require careful study to choose the structure of the network for a given system [22,27]. Our training algorithm is used to avoid this concern. Our method is distinct from methods that use a neural network to model the system and then analytically determine a control formula for the neural network model [20,28]. To our knowledge, no previous method can train controller neural networks for multiple systems without requiring system-dependent modifications to the training algorithm and/or the neural network structure.

III. OUR METHOD

We give a brief introduction to neural networks in Sec. III A and to genetic algorithms in Sec. III B. Our implementation of these ideas is based on SANE, discussed in Sec. III C. Our use of SANE to solve the chaos control problem is discussed in detail in Secs. III D and III E.

A. Neural networks

A neural network is a biologically motivated computational construct [18]. A network may be hardware or software based and consists of several nodes, or *neurons*, connected by weighted communication lines. A neural network is a structure whose ith neuron has input value x_i, output value $y_i=g(x_i)$, and connections to other neurons described by weights w_{ij}. The envelope function $g(x_i)$ is commonly a sigmoidal function $g(x)=(1+e^x)^{-1}$. The input value x_i of neuron i is given by the formula $x_i=\Sigma_{j\neq i}w_{ij}y_j$.

We use a feed-forward network, in which the neurons are organized into layers: an input layer, hidden layer(s), and output layer. The input layer input values are set by the environment, while the output layer output values are returned to the environment (see Fig. 1). For example, the output information may be interpreted as a control signal. The hidden layers have no external connections: they only have connections with other layers in the network. In a feed-forward network, a weight w_{ij} is nonzero only if neuron i is in one layer and neuron j is in the previous layer. This ensures that information flows forward through the network, from the input layer to the hidden layer(s) to the output layer. More complicated forms for neural networks exist and can be found in standard textbooks such as Ref. [18]. Training a neural network involves determining the weights w_{ij} such that an input layer presented with information results in the output layer having a correct response. This training is the fundamental concern when attempting to construct a useful network. The training method we use is presented in Sec. III C.

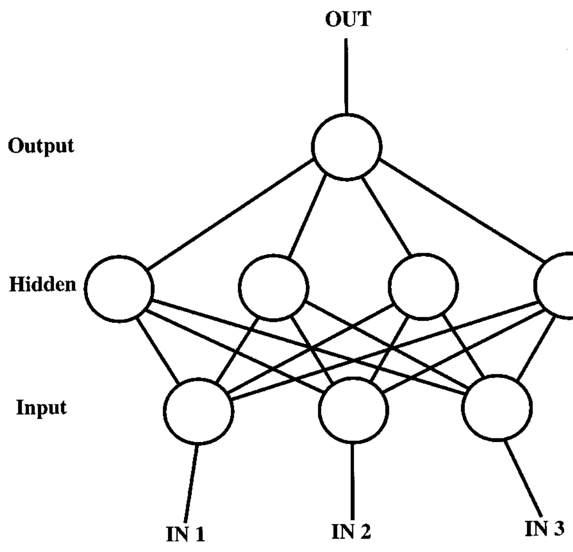

FIG. 1. Typical neural network. This feed-forward network consists of an input layer, one hidden layer, and an output layer. The lines represent the weighted connections between nodes in successive layers. ''IN'' represents information given to the neural network and ''OUT'' is the information the network returns to the environment; these are the only external connections.

Neural networks are reasonable candidates for providing a control algorithm to a chaotic system. System observations are presented to the network via the input layer. The output layer determines a perturbation to the system, whose modified behavior is then presented to the network as a new set of input information. This process is iterated to stabilize the system. Methods such as the OGY method rely on a linear description of the system near the fixed point, whereas the nonlinearity of a neural network [due to the nonlinear function $g(x)$] allows for the possibility of a nonlinear description of the system. By varying the structure of the input and output layers of a neural network, it can be easily modified to cases for which different numbers of system variables can be observed and cases for which multiple control parameters are available (see Sec. III E). We use the network topology with three layers shown in Fig. 1; this topology is system independent.

B. Genetic algorithms

A genetic algorithm [29,30] is a method for rapidly and efficiently searching through the space of all possible solutions to a given problem. In many problems the *fitness* of a given solution can be determined; a solution with a high fitness value is better than a solution with a low fitness value, although the maximum possible fitness might not be known. Genetic algorithms perform well in cases for which gradient information is not available; in such situations, algorithms such as the conjugate-gradient method cannot be used to find maxima in the solution space.

The basic idea of a genetic algorithm is to consider an ensemble, or *population*, of possible solutions to the problem. For our particular method, the population consists of hidden layer neurons from which we construct controller networks; see Sec. III C. For any genetic algorithm, a population of randomly generated solutions is formed and the fitness of each solution is determined. After the population has been evaluated, the best solutions are copied. These copies are changed slightly, with the hope that some of these random changes will produce a better solution; this process is *mutation*. Additionally, randomly chosen elements from pairs of good solutions are combined to form new solutions; this process is *crossover*. A new population is assembled from the mutated copies, the new solutions formed from crossover, and the best solutions from the original population.

The fitness evaluation of the population and the creation of new solutions is repeated until an adequate solution is found. Each evaluation of the population is a *generation*. In this manner, the process is ''evolving'' the population into one containing solutions that are better suited for the task at hand. Genetic algorithms have the advantage of searching multiple areas in solution space in parallel, which often prevents focusing on a solution that is a local, rather than a global, maximum in fitness. A genetic algorithm then consists of a description of solutions, a *fitness function* to evaluate the solutions, and methods (such as crossover and mutation) to produce new solutions from old solutions.

C. Symbiotic adaptive neuro-evolution

We use a genetic algorithm to evolve neurons that are used to form networks that can stabilize an unstable fixed point. Our genetic algorithm method is a replacement for other methods of determining network weights (see Ref. [18] for a discussion of backpropagation, which requires gradient information, and other training methods). The advantage of the genetic algorithm approach for the chaos control problem is that the network weights can be found by examining the performance of a network as a controller rather than by providing correct control signals for various input data.

The specific method we use is based on symbiotic adaptive neuro-evolution [8–10]. The crux of SANE is to use a genetic algorithm to evolve a population of individual hidden layer neurons rather than networks as a whole. Each hidden layer neuron is specified by its weights w_{ij} fully connecting it to the input and output layers. Each hidden layer neuron specification is independent since neurons are not laterally interconnected. A network is formed with several neurons selected from the population; this selection can be either random or directed (see the Appendix for details). This algorithm may result in the production of specialized neurons that work symbiotically with each other to produce useful networks [8]. The fitness of a network is used to determine the fitness of the individual neurons. The advantage of SANE is that it is specifically designed to maintain a diverse population of neurons, which aids the parallel searching power of genetic algorithms [8]. In addition, resultant networks are able to ignore useless inputs and the size of the hidden layer does not need to be precisely predetermined (as the specialization of neurons can result in neurons that are redundant).

Our specific algorithm differs slightly from the original SANE algorithm. The primary difference is that our algorithm has a short-term memory that forms some networks from neurons that worked well together in the previous generation or from their transformed copies. This allows focused testing of neurons that work well together. See the Appendix for the details of our algorithm. Note that we are not directly evolving the networks; the evolving population consists of hidden layer neurons from which we construct the networks.

D. Fitness function for controlling chaos

The fitness function is a statement of the goals of a genetic algorithm. The proper choice of the fitness function determines the speed with which the genetic algorithm can converge on the correct solution. In our algorithm the fitness function evaluates the ability of an individual neural network to control a dynamical system. To better generalize the control method to many chaotic systems, we use a fitness function that assumes little about the dynamics of the system and is robust even in the presence of significant noise. The goal of our algorithm is to find a neural network that can control a specific system such that the dynamics eventually reaches a fixed point. A period-1 fixed point is defined by $X_n \approx X_{n-1}$ at each step, where X is an observable variable and n is the nth observation. (The actual location of the fixed point is not a required part of the fitness function.) The fitness function evaluates a network through observations of its attempts to control the system.

The fitness function we use has three parts: the fitness F is given by $F \equiv AF_1 + BF_2 + CF_3$, where A, B, and C are adjustable parameters that are system independent. The map to be controlled is iterated many times (typically 1000) and the value of each part of the fitness function is set by the behavior of the map and the network.

F_1 is determined by whether the network has stabilized a fixed point by the end of the 1000 iterations. The differences in a system variable X are examined for the last few iterations of the map. To stabilize a period-1 fixed point, define $\Delta_n \equiv |X_n - X_{n-1}|/S$, where S is the size of the uncontrolled attractor of the system. $F_1 \equiv 1 - \langle \Delta_n \rangle$, where the average is taken over the last few iterations of the map (typically 40). Thus small values of Δ_n (successive iterations remaining near each other) result in a larger fitness F_1.

F_2 quantifies the growth rate of Δ_n near the fixed point. If Δ_n is small (less than 0.01), the values of Δ_n are stored until Δ_n grows larger than 0.10. $F_2 \equiv 1 - \ln(\lambda)$, where λ is the geometric mean of the quantities Δ_{n+1}/Δ_n for all Δ_n that have been stored. λ is similar to the largest Lyapunov exponent, although for higher-dimensional systems it may be influenced by other Lyapunov exponents. When the fixed point is successfully stabilized by the neural network, $\lambda \approx 1$. If Δ_n is never smaller than 0.01, λ is undefined and $F_2 = 0$. A large number of iterations (1000) is used to allow the system ample opportunity to be near the fixed point ($\Delta_n < 0.01$) so that F_2 can be determined.

F_3 rewards networks that are optimal controllers, which rely on small perturbations to establish control. F_3 depends on the behavior of the network itself and not the dynamics of the system. Many randomly chosen network weights lead to a network that applies the largest possible perturbation δP to the system. However, chaos control usually requires a smaller perturbation; for example, in the absence of noise, if the system is exactly on the fixed point, then the necessary perturbation is $\delta P = 0$. Near the fixed point only small perturbations are needed [1]. Thus a reward is given to networks that favor smaller perturbations. Define F_3 to be the fraction of iterations for which $|\delta P|$ is smaller than $0.95\delta P_{\text{max}}$, where δP_{max} is the magnitude of the maximum allowed perturbation. (δP_{max} is chosen before evolution begins; in practical applications, the size of δP_{max} may be limited by the system.)

We find that the size of δP is less important than the behavior near the fixed point, so if the system is able to get near the fixed point (and thus F_2 is nonzero) F_3 is set to zero. Trials with both F_3 and F_2 nonzero occasionally resulted in networks that had $\delta P \equiv 0$, independent of the inputs to the network. Such networks were trapped at a local maximum of fitness space because they were rewarded by F_3; variations due to F_2 were insufficient to move the networks away from this local maximum. Setting F_3 to zero when F_2 is nonzero solved this problem.

The algorithm's ability to find good networks depends on the parameters A, B, and C. We use $A=200$, $B=1000$, and $C=20$. This results in each piece of the fitness function being roughly the same order of magnitude, with an emphasis on F_2, which is the most sensitive to small improvements in the control ability of a network. Small variations in these three parameters do not change the performance of the genetic algorithm. Note that the fitness function F is used only to determine the appropriate ranking of neurons, so that F is defined only to within a linear transformation; $F' = aF + b$ will give the same ranking and work equally well as a fitness function. In this sense, one of the parameters A, B, and C can be rescaled: the fitness function has only two independent parameters. These parameters appear to be system independent; see the discussion in Sec. IV D.

In noisy systems, the same network can be tested multiple times with different results. To make the fitness measurement more robust, each network is tested twice and is assigned an average fitness.

The fitness function can be easily adapted to find orbits with higher periodicity. By redefining $\Delta_n = |X_n - X_{n-p}|/S$ a period p orbit can be found. An additional redefinition of Δ_n is useful if the exact location of the fixed point is known. By defining $\Delta_n = |X_n - X_{\text{fixed}}|/S$, the evolution proceeds much faster (typically reducing the number of generations needed to create a successful network by an order of magnitude). This is also useful if there exist multiple unstable period-1 fixed points within the normal dynamics of the system and stabilization of a specific fixed point is desired.

Given this fitness function, we can calculate the maximum achievable network fitness. A successful network will have $\langle \Delta_n \rangle \approx 0$ ($F_1 \approx 1$), $\lambda \approx 1$ ($F_2 \approx 1$), and $F_3 = 0$. Therefore $F_{\text{max}} \approx A + B = 1200$. In practice, $\langle \Delta_n \rangle > 0$ and λ can be slightly less than 1, so a successful network can have a fitness slightly different from 1200. We measured the fitness of the OGY method by replacing the neural network with the correct analytical OGY algorithm for the Hénon map. We found that the OGY method has a fitness of approximately 1200.

E. Parameters

The structure of the neural network and parameters for the evolution must be set before generating the initial population of neurons. We use four input neurons, seven hidden neurons, and one output neuron, although the results appear to be nearly independent of the number of input and hidden neurons (see Sec. IV C for details). Each hidden neuron receives input from all neurons in the input layer and outputs to all neurons in the output layer. The input neurons are assigned the values X_n, X_{n-1}, X_{n-2}, and X_{n-3} (where X is a variable from the system and the four values of X are taken from four successive measurements of X). These lagged variables provide the network with a useful description of the system [31]. The value y of the output neuron sets the perturbation applied to the system $\delta P = \delta P_{\text{max}}(2y-1)$. The envelope function $y = g(x)$ (see Sec. III A) is a sigmoidal function, so that $0 < y < 1$ and $|\delta P| < \delta P_{\text{max}}$.

TABLE I. Typical parameters for the evolution.

Population size	100 neurons
Neurons preserved each generation	30% of population
Neurons formed by mutation	60% of population
Neurons formed by crossover	10% of population
Networks formed	100 per generation

The typical details of the genetic algorithm parameters are listed in Table I. These parameters were chosen to be similar to the parameters for the original pole-balancing work discussed in Ref. [8]. We find our results were not sensitive to reasonable changes in these parameters. Some variations from these parameters are discussed in Sec. IV D.

IV. RESULTS

A. Logistic map

The logistic map is a one-dimensional map that exhibits chaotic behavior. The map is given by

$$X_{n+1} = P_0 X_n (1 - X_n). \tag{1}$$

We consider $3 \leqslant P_0 \leqslant 4$ (for which the map has a nontrivial unstable period-1 fixed point) and $0 \leqslant X \leqslant 1$ (see Fig. 2). Noise is added to the system by adding a random variable η_n to X_n for each iteration of the map; η_n is chosen uniformly to be in the range $(-\epsilon/2, \epsilon/2)$. We restrict $0 < X_n + \eta_n < 1$. With $\epsilon < 0.001$ and $\delta P_{\text{max}} = 0.01$, our algorithm is successful; for higher noise levels, typically no networks were found that could successfully control chaos. The algorithm usually takes 300–700 generations to find a successful network.

For $\delta P_{\text{max}} = 0.01$, the unstable period-1 fixed point can be stabilized for $P_0 > 3.66$. For $3 \leqslant P_0 \leqslant 3.66$, the natural dynamics of the logistic map is never sufficiently near the fixed point to allow the small perturbations to stabilize it (see Fig. 2). However, for $P_0 \leqslant 3.66$ and $\delta P_{\text{max}} = 0.1$, the algorithm finds successful networks. In these cases, the F_2 portion of the fitness function remains zero until the neurons have evolved significantly (see Sec. III D). The F_1 portion acts to squeeze $|X_n - X_{n-1}|$ to smaller values until X is near X_{fixed}, at which point F_2 begins to reward good networks.

Typically the stabilized fixed points are not identical to the fixed points of the uncontrolled system. The fitness function penalizes networks that produce extremely large perturbations but does not reward networks for having small per-

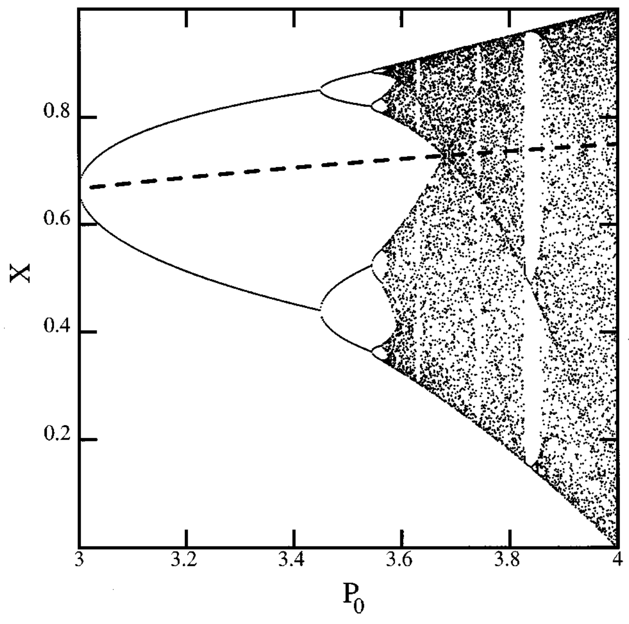

FIG. 2. Bifurcation diagram for the logistic map. The dashed line indicates the location of the unstable period-1 fixed point.

turbations, so δP does not tend exactly to zero for the controlled system. With $\delta P \neq 0$, the fixed point location is shifted slightly from the uncontrolled location. We have tried modifying the fitness function to reward networks that produce small perturbations, but this often resulted in networks that had $\delta P \equiv 0$ independent of the inputs. In situations for which the fixed-point location is known, the fitness function can use this knowledge as discussed in Sec III D. In such cases the controllers found do not change the location of the fixed point.

B. Hénon map

The Hénon map is a two-dimensional map that exhibits chaotic behavior [32]. The map is given by

$$X_{n+1} = a_0 + b_0 Y_n - X_n^2, \tag{2}$$

$$Y_{n+1} = X_n . \tag{3}$$

For many values of a_0 and b_0, the dynamics of this map are chaotic. Figure 3 shows for the Hénon map a strange attractor, an attracting set with fractal dimension [33]. Embedded within this attractor is an unstable period-1 fixed point where $X_n = X_{n+1}$ (see Fig. 3). By applying small perturbations to the parameter a_0, we wish to stabilize the fixed point located at $X \approx 0.8$. As for the logistic map, we add noise to the X variable at each iteration, typically with $\epsilon = 0.001$.

We consider the range $1.130 \leqslant a_0 \leqslant 1.415$ and $b_0 = 0.3$; the bifurcation diagram for this range is shown in Fig. 4. (For $a_0 > 1.415$ the Hénon map with noise is often unstable; for $a_0 < 1.130$ the natural dynamics of the map is never sufficiently near the fixed point to allow small perturbations to stabilize it.) Our algorithm successfully stabilizes the unstable fixed point in the Hénon map in this range without changing any training parameters. For the same network topology and fitness function as used for the logistic map, the algorithm finds successful neural networks, typically taking

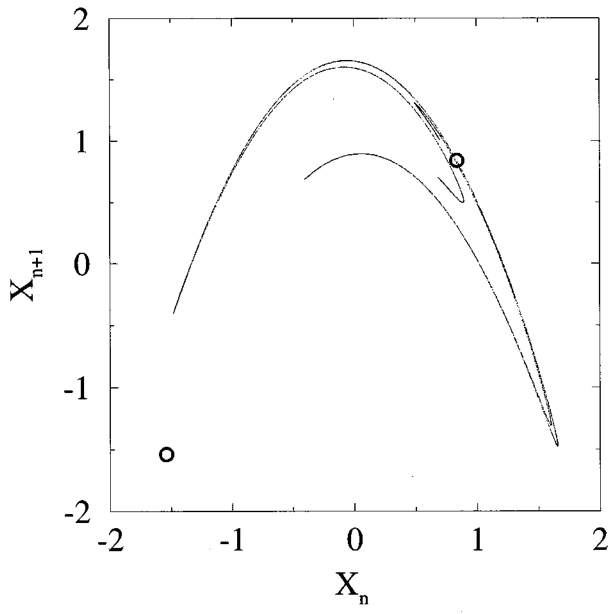

FIG. 3. Strange attractor for the Hénon map ($a_0 = 1.29$, $b_0 = 0.3$). The circled points are the period-1 fixed points of the system.

200–1100 generations. The evolution is slower when a_0 is near 1.24, where the unperturbed dynamics are period-7 rather than chaotic, but the algorithm is still able to stabilize the period-1 behavior.

The number of generations needed for control may be of concern. In real world experiments, parameters may drift slightly during the course of the experiment. We examined parameter drift by changing a over the course of the evolution. Evolution succeeds even in cases with a varying sinusoidally with a period of 10–1000 generations, as well as cases with a chosen randomly from the interval (1.29,1.40)

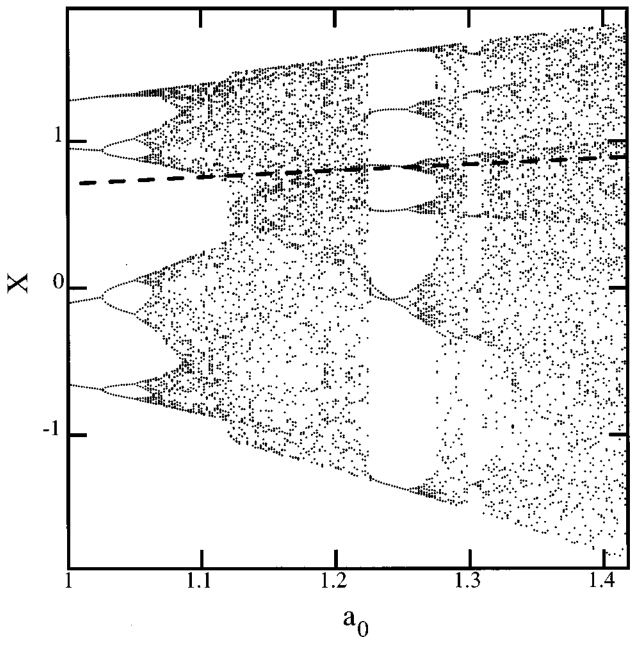

FIG. 4. Bifurcation diagram for the Hénon map. The dashed line indicates the location of the unstable period-1 fixed point.

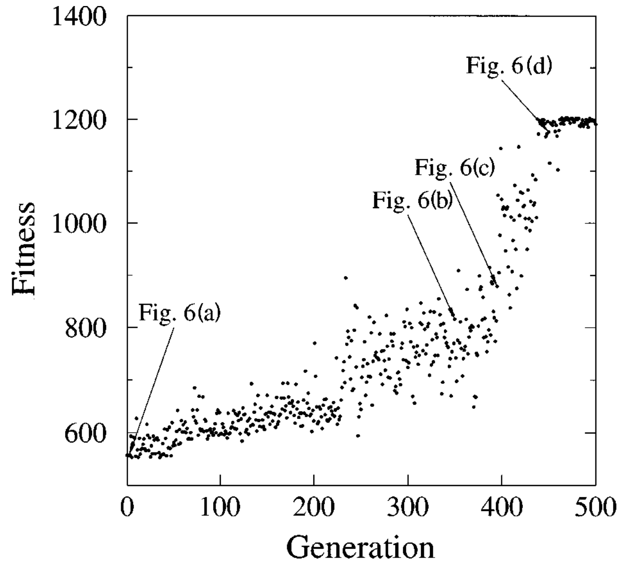

FIG. 5. History of the evolution for the Hénon map with $a_0 = 1.29$, $b_0 = 0.3$, $\delta a_{\max} = 0.01$, and a noise level of $\epsilon = 0.001$. For other parameters see the text. As the fitness value increases, the best networks in the population have better performance, but a successful controller is only found after the rapid increase seen around generation 400. Note that the maximum possible fitness is obtained, $F_{\max} \approx 1200$. The results shown are similar to the evolution for the logistic map.

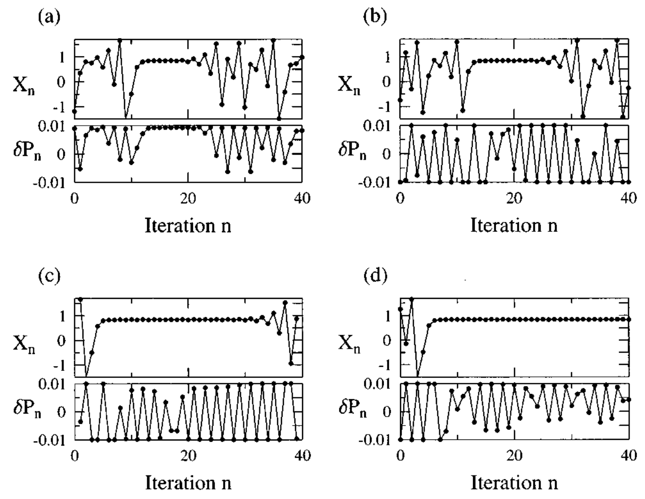

FIG. 6. Performance of the best network constructed after (a) 2 generations, (b) 350 generations, (c) 395 generations, and (d) 450 generations. These data correspond to Fig. 5. The origins of the abscissas are arbitrary; the portion of the data shown is chosen to illustrate the behavior near the fixed point. Iterations before the dynamics approach the fixed point are not pictured. As the evolution proceeds, the neural networks are learning to apply appropriate perturbations when the system is near the fixed point. Note in (d) that the control perturbations needed are small once control has been established.

each generation. For trials discussed in this paper, all parameters are held fixed for the course of the evolution unless otherwise noted. We also examined some controllers found with fixed parameters and tested them with parameter values for which they were not trained. We found that the networks could still control a system for which the parameter was within 10% of the training parameter.

Figure 5 shows the typical evolution of the population of neurons for the Hénon map with $a_0 = 1.29$, $b_0 = 0.3$, and perturbations limited to $\delta a_{\max} = 0.01$. In the example shown, the population took approximately 400 generations to construct neural networks that could stabilize the unstable fixed point of the Hénon map. A series of 100 runs with identical conditions (but different random number seeds) were performed, and 100–1700 generations were needed to accomplish results similar to Fig. 6(d); the mean was 550 generations and the median was 412 generations. Four of these runs failed to find a successful controller in 2000 generations.

The progress of the evolution is seen in Fig. 6. Initially networks are formed randomly and the behavior of the system looks similar to the unperturbed dynamics [Fig. 6(a)]. After 250 generations, the best networks formed in each generation can slow the escape of the system from the unstable fixed point, but the dynamics remain uncontrolled [Fig. 6(b)]. Around generation 400, a rapid change is seen in the evolution (Fig. 5). At this point, the best networks are learning to keep the system near the unstable fixed point for many iterations [Fig. 6(c)]. By generation 450, the best networks are able to stabilize the system as soon as the system is sufficiently close to the unstable fixed point. Small perturbations are maintained to continue the control despite the presence of noise.

Figure 7 shows the graph of the number of generations needed to evolve successful controllers for varying noise level ϵ. The spread in the required number of generations needed (200–1100) reflects the underlying uncertainty of the evolution process; the variation appears independently of noise level. For $\epsilon \geqslant 0.01$ no successful networks are formed in the 2000 generations allowed. An additional 10 000-generation trial with $\epsilon = 0.02$ failed to form a successful network (although see Sec. IV C). Above $\epsilon \approx 0.17$, noise fre-

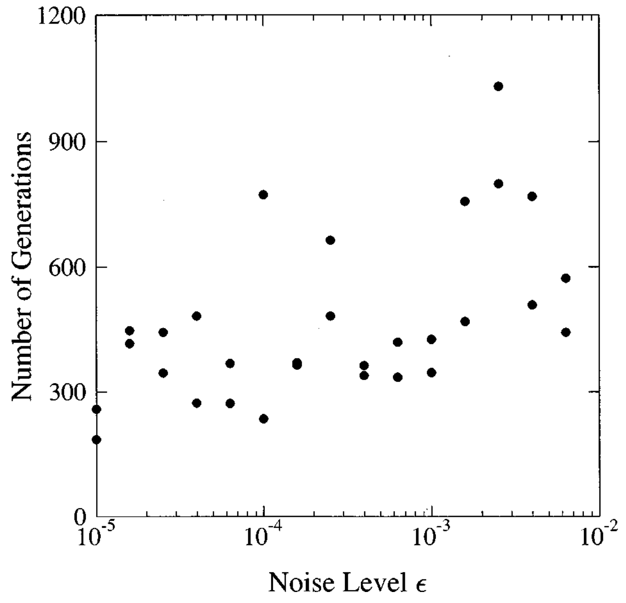

FIG. 7. Number of generations needed to control the system versus the noise level imposed. For $\epsilon \geqslant 0.01$, no networks were produced in 2000 generations that could successfully control the system. For these data, $a_0 = 1.29$ and $b_0 = 0.3$, and $|\delta a|$ was limited to 0.01. For larger $|\delta a|$ allowed, systems with higher noise levels could be controlled; see the text.

quently kicks the Hénon map dynamics out of the basin of attraction, which often results in $X_n \rightarrow -\infty$. Such behavior is unphysical and not considered here.

We test the robustness of these results for the Hénon and logistic map by varying the structure of the neural network and varying the genetic algorithm parameters. These variations are discussed in the following two subsections.

C. Variations in the neural network

Three variations of the neural network architecture are considered: varying the input, hidden, and output layers. The results discussed for the logistic map and the Hénon map are for networks that typically include three or four input neurons, seven hidden layer neurons, and one output neuron.

Several trials were done with five or more input neurons. In the first set of trials, the input neurons were given additional lagged variables X_n. The evolution was slowed in most cases by approximately 20%, but still found networks that successfully controlled the system. This is promising for cases for which the dimension of the dynamics is unknown, as a larger number of lagged variables can be used (a larger input layer). If the system is high dimensional, then these lagged variables may all be useful [31]. If the system is low dimensional, the extra variables are ignored. We also tested the performance of the networks with underspecified inputs. The Hénon map could be controlled even with only one input neuron, a somewhat surprising result given that the map is two dimensional.

In a second set of trials, the additional input neurons were assigned the values of the previous perturbations δP_n without significantly changing the results; Refs. [2,17] show that the optimal control method typically does depend on earlier perturbations.

In a third set of trials, additional input neurons were added that were assigned random numbers as input. Again, the evolution was slowed slightly, but the networks learned to ignore these meaningless inputs. Thus the evolution appears fairly robust to over specified input layers.

The results additionally appear robust to changes in the number of neurons in the hidden layer. Typically evolution was faster if the number of hidden layer neurons was approximately twice the number of inputs. A strength of the SANE algorithm is that even if a hidden layer has more neurons than needed, it will still find a solution (with some neurons acting redundantly) [8]. The neurons are evaluated for their ability to work in a network with each other, so neurons will be selected that work well in a network with superfluous hidden neurons. This is a useful feature as the necessary number of hidden neurons is often unknown.

The output layer was the one critical part of the neural networks. As the only information returning to the system is from the controller, changes to the representation of such information could have significant consequences on the performance of the network. Trials with different arrangements for the output layer (such as using two or three neurons to specify the perturbation in some way) were unsuccessful and often could not control the system at all. However, the network was successful when allowed to control two parameters of the Hénon map. For example, control is possible if the network has two output neurons, specifying δa and δb, with

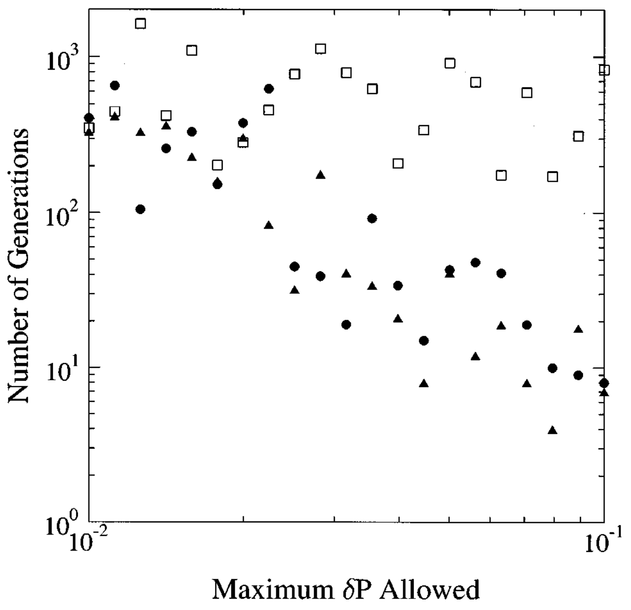

FIG. 8. Number of generations needed to stabilize the Hénon map with one control parameter (circle), the Hénon map with two control parameters (triangle), and the logistic map (square), versus the maximum allowed perturbation size. For the Hénon map results, $a_0 = 1.29$, $b_0 = 0.3$, and the abscissa is $\delta a_{\max}$, and for the two-parameter Hénon map results, $\delta b_{\max} = 0.3 \delta a_{\max}$. For the logistic map results, $P_0 = 3.7$. The noise level for these cases is $\epsilon = 0.001$.

$\delta a_{\max}$ and $\delta b_{\max}$ limited to 1% of a_0 and b_0, respectively.

In addition to varying the layers of the neural network, we can ease the restriction on the perturbation size $\delta P_{\max}$. With larger perturbations, many control methods can stabilize the system for larger noise levels and achieve faster control. With our algorithm, we found that for the Hénon map, with $\delta a_{\max} = 0.03$ a noise level up to $\epsilon = 0.03$ could be controlled; with $\delta a_{\max} = 0.05$ a noise level up to $\epsilon = 0.05$ could be controlled. We tested the OGY algorithm and found identical results. As seen in Fig. 8, our algorithm often finds good neural networks faster when larger perturbations are allowed. Once found, the good neural networks have the advantages any control method has when allowed larger perturbations. Figure 9 shows the typical number of iterations needed by networks to control chaos; larger allowed perturbations generally result in faster control. These are typical results; the best networks found were often twice as fast, as Fig. 9 indicates.

D. Variations in the genetic algorithm

The parameters listed in Table I were varied in order to test the robustness of the evolutionary algorithm. We typically use a population of 100 neurons to form the networks in each generation. The results are fairly independent of the population size, as long as the number of networks tested in each generation is roughly the same as the population size. This ensures that each individual neuron gets tested approximately a number of times equal to the number of hidden layer neurons each generation (see the Appendix for details).

Both mutation and crossover appear to be important for successful evolution. In most cases a good solution is not found if either of these operations is removed. Varying the

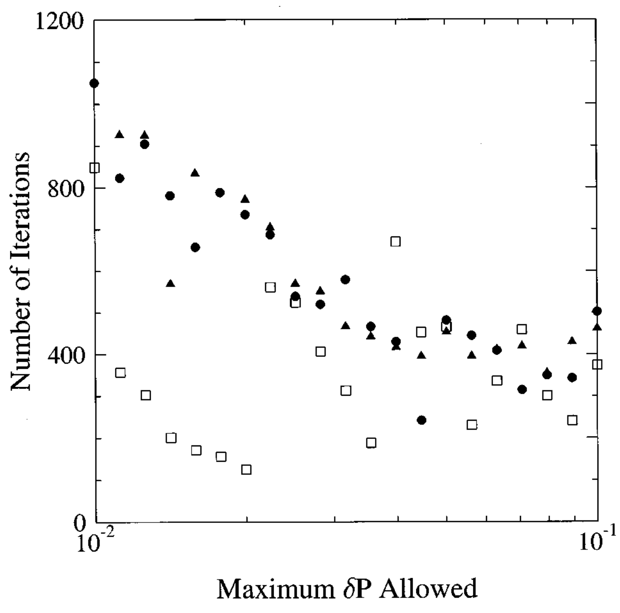

FIG. 9. Average number of iterations needed by the best neural networks to stabilize the Hénon map with one control parameter (circle), the Hénon map with two control parameters (triangle), and the logistic map (square), versus the maximum allowed perturbation size. For the Hénon map results, $a_0 = 1.29$, $b_0 = 0.3$, and the abscissa is $\delta a_{\max}$, and for the two-parameter Hénon map results, $\delta b_{\max} = 0.3\delta a_{\max}$. For the logistic map results, $P_0 = 3.7$. The noise level is $\epsilon = 0.001$.

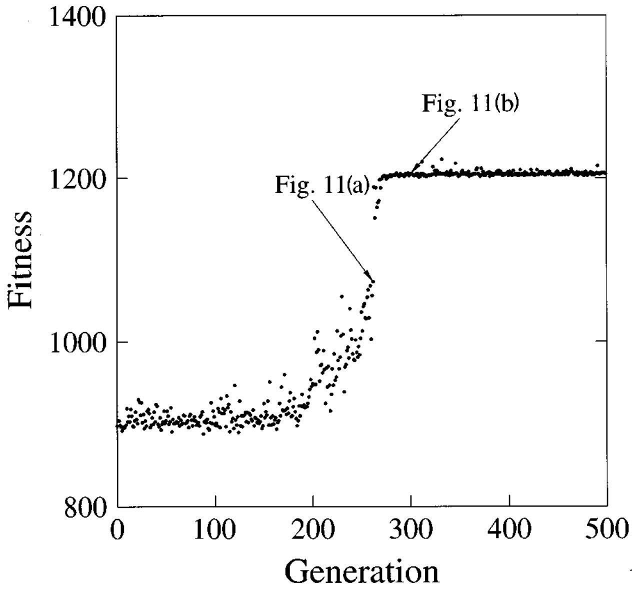

FIG. 10. Fitness of the best network each generation for the logistic map with $P_0 = 3.7$, $\delta P_{\max} = 0.03$, and a noise level of $\epsilon = 0.001$. For this trial, the goal is to stabilize a period-2 orbit. The fitness of the networks shown in Fig. 11 is indicated.

number of neurons created with each operation (see Table I) does not affect the results significantly, although it appears that a low level of crossover is sufficient.

Some variations of the fitness function were checked. Independent variations of $\pm 20\%$ to the parameters A, B, and C in the fitness function do not change the results. In most cases larger variations prevent the algorithm from evolving a successful controller for one or both of the two maps. The parameter C, related to the size of the applied perturbation δP, is the least important of the parameters, although it seems necessary in cases for which $\delta P_{\max}$ is allowed to be large. Additionally, stabilization of higher-periodicity orbits is possible by modifying the fitness function appropriately (see Sec. III D). Results are shown in Figs. 10 and 11.

V. CONCLUSION

We have developed a method for controlling chaos using neural networks formed with a genetic algorithm. This method requires a fitness function that, for any given chaotic system, can direct the creation of a successful neural network controller. Our fitness function works effectively with both the one-dimensional logistic map and the two-dimensional Hénon map. The fitness function appears to depend neither on the size of the controlling perturbations allowed nor on the dimensionality of the map. Future work will further investigate the applicability to other systems. The method works both in chaotic cases (the natural dynamics of the system approach a fixed point many times) as well as in periodic cases when large perturbations are allowed (the natural dynamics never approach a fixed point); see Sec. IV. The method is reasonably robust to noise. The locations of unstable fixed points of the dynamical system are not provided for the training of the networks.

The neural networks developed by our method have useful characteristics. They are able to use small perturbations to control a system and the network inputs depend only on previous observations of a system variable, similar to other control methods [1,3]. Other characteristics of our controllers are not reproduced by some control methods. For example, the networks succeed even when the system starts far from the fixed point, without having to be activated only near the fixed point. A possible use for our method is to examine the behavior of the neural networks to determine the control method the network is applying to a given system; this could lead to a better understanding of that system or to ideas for other control algorithms.

A concern about using our method to control a physical system is the time needed to find a good controlling network. For the Hénon map, the algorithm takes approximately 500 generations to find a network that effectively stabilizes the unstable fixed point. Each generation 100 networks are each evaluated twice and each evaluation requires the Hénon map to be iterated 1000 times (see the Appendix). Thus 10^8 iterations of the map are needed, which for experimental observation frequencies ranging from 1 kHz to 1 Hz would require from 1 day to 3 yr of training. However, information about the physical system to be controlled may supplement our current assumptions, allowing the training process to proceed faster. First, loosening the constraint of small perturbations may speed the evolution; see Fig. 8. Second, if the location of the unstable fixed point is known and used in the fitness function, evolution is faster by approximately a factor of 10. Third, many physical systems can be modeled and the neural networks pretrained on the model. Future work will focus on combining neural networks with a Petrov-Showalter modeling technique [17] to obtain an algorithm that trains rapidly while retaining the advantages of our current method (ignorance of dimensionality and of fixed point location).

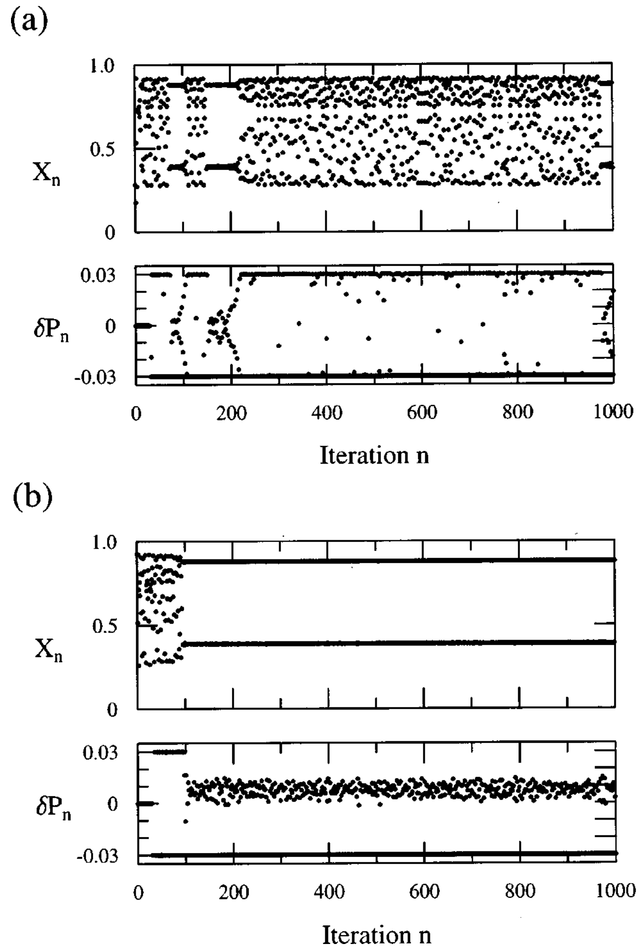

FIG. 11. Performance of the best network after (a) 262 generations and (b) 300 generations. For the first 30 iterations, the output of the neural network is ignored and $\delta P = 0$ is used to allow possible transients from the initial state to disappear. Note in (b) that for every second map iteration a maximal perturbation ($\delta P = -0.03$) is applied. The spread in δP around zero reflects the variations needed to control against the noise in the system. These data correspond to Fig. 10.

The computer program to implement our algorithm is available on the World Wide Web at http://chaos.ph.utexas.edu/~weeks/dsane/. We have tested our program on UNIX computers. We believe that our approach will allow enough flexibility to apply this method to many different chaotic systems and may lead to a ‘‘black box’’ controller that can find a control method for a system without any fine tuning.

ACKNOWLEDGMENTS

We thank Risto Miikkulainen and David E. Moriarty for the original pole-balancing computer code implementation of SANE and for helpful discussions. E.R.W. thanks Stephanie Forrest and the 1995 Complex Systems Summer School for a helpful introduction to genetic algorithms. E.R.W. acknowledges the financial support of the ONR and the University of Texas. J.M.B. acknowledges the support of the ONR. The authors also thank Joseph Breeden, Michael Marder, Valery Petrov, and Harry Swinney for helpful discussions.

APPENDIX: DETAILS OF DIRECTED SYMBIOTIC ADAPTIVE NEURO-EVOLUTION

We use a variation of the original SANE algorithm [8,9], which we term directed SANE. This appendix details the software implementation of this method.

The networks are formed with the connections shown schematically in Fig. 1. We can vary the number of neurons in each layer; see Sec. IV C. The individual elements in the evolving population are the hidden layer neurons, which have weights fully connecting them to the input and output layers. The program starts by creating a population of neurons with random weights. The neurons are evaluated and then evolved.

(i) N_H (typically 7) neurons from the population are used to construct the hidden layer of a network.

(ii) The evolutionary fitness of the network is determined (see Sec. III D for details).

(iii) Steps (i) and (ii) are repeated many times (typically 100) to test all neurons in the current population. The fitness of a neuron is the fitness of the best network in which it has participated.

(iv) The population of neurons is ranked by fitness and divided into two subpopulations, the ‘‘good’’ and the ‘‘bad.’’ All of the neurons in the good subpopulation are retained for the next generation. The neurons in the bad subpopulation are replaced.

(a) Mutation. A portion of the replacement neurons are formed by making copies of good neurons and multiplying each weight by a random number close to 1.0 (and occasionally changing the sign of the weight).

(b) Crossover. The rest of the replacement neurons are formed by selecting two good neurons and assigning the new weights by choosing each weight from one of the two good neurons at random.

(v) The two subpopulations are then taken as the new population for the next generation. Note that all of the neurons in the good subpopulation remain unchanged in the new population. With this new population, the process [steps (i)–(iv)] is restarted.

(vi) This process is repeated for as many generations as needed in order to produce a single network that has a fitness that is satisfactory. When this occurs, that particular network is retained as the solution to the problem.

The selection of N_H neurons from the population to form a network can be either random or directed. By considering which networks performed well in the previous generation, we may bias the selection of neurons to favor previously successful networks. This is the reason we name our method directed SANE.

We store four *network definitions*, the N_H indices of the neurons that composed the four best networks in the previous generation. Half of the new networks are created using these network definitions with either the original or newly transformed neurons. Thus new networks resemble the previous generation’s outstanding networks and may perform better. In principle, the network definitions created in a given generation are unrelated to those formed in the previous generation; these definitions act only as a short-term memory. In practice, often the best network in a given generation is formed with a network definition from the previous generation and thus there is some continuity in the definitions as a side effect. Our short-term memory is similar in concept to Moriarty and Miikkulainen's ''hierarchical SANE'' [9], although different in implementation. The short-term nature of the memory allows for great flexibility as the evolution proceeds, allowing new improvements to be extensively tested as better networks are found. We find that forming half of the networks randomly and the other half from the definitions works better than forming all of the networks from either method exclusively.

Directed SANE differs in three other ways from the original SANE algorithm. (For the details of their method, see Refs. [8,9,34].) First, SANE neurons are encoded as bit strings; directed SANE neurons are encoded as weights and each weight is stored by the program as a floating point variable. For example, in SANE, bit strings describing weights could be broken in the middle by crossover. In directed SANE, the crossover transformation selects weights from both parent neurons and treats each individual weight as an unbreakable piece. Thus crossover does not result in loss of the information represented by the weight. Second, in SANE, at each generation the entire population of neurons is mutated. In directed SANE, unchanged copies of the best neurons remain in the population for the next generation. The best neurons in a population are frequently better than most mutated versions of those neurons; it is best to replace neurons only with neurons that have been tested and found to be better. Third, the fitness of a neuron is the fitness of the best network it participated in rather than the average of all of the networks. These three changes result in faster evolution for our problem.

[1] E. Ott, C. Grebogi, and J. A. Yorke, Phys. Rev. Lett. **64**, 1196 (1990).

[2] U. Dressler and G. Nitsche, Phys. Rev. Lett. **68**, 1 (1992).

[3] T. Shinbrot, C. Grebogi, E. Ott, and J. A. Yorke, Nature (London) **363**, 411 (1993).

[4] W. L. Ditto, S. N. Rauseo, and M. L. Spano, Phys. Rev. Lett. **65**, 3211 (1990).

[5] E. R. Hunt, Phys. Rev. Lett. **67**, 1953 (1991).

[6] R. Roy, T. W. Murphy, Jr., T. D. Maier, Z. Gills, and E. R. Hunt, Phys. Rev. Lett. **68**, 1259 (1992).

[7] J. Starrett and R. Tagg, Phys. Rev. Lett. **74**, 1974 (1995).

[8] D. E. Moriarty and R. Miikkulainen, Mach. Learn. **22**, 11 (1996).

[9] D. E. Moriarty and R. Miikkulainen, University of Texas at Austin Technical Report No. AI96-242, 1996 (unpublished).

[10] D. E. Moriarty and R. Miikkulainen, in *From Animals to Animats 4*, Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, edited by P. Maes, M. Mataric, J.-A. Meyer, and J. Pollack (MIT Press, Cambridge, MA, 1996).

[11] D. Auerbach, C. Gebogi, E. Ott, and J. A. Yorke, Phys. Rev. Lett. **69**, 3479 (1992).

[12] P. So and E. Ott, Phys. Rev. E **51**, 2955 (1995).

[13] M. Ding, W. Yang, V. In, W. L. Ditto, M. L. Spano, and B. Gluckman, Phys. Rev. E **53**, 4334 (1996).

[14] E. Barreto and C. Grebogi, Phys. Rev. E **52**, 3553 (1995).

[15] K. Pyragas, Phys. Lett. A **170**, 421 (1992).

[16] A. Hübler, Helv. Phys. Acta **62**, 343 (1989).

[17] V. Petrov and K. Showalter, Phys. Rev. Lett. **76**, 3312 (1996).

[18] S. Haykin, *Neural Networks—A Comprehensive Foundation* (Macmillan, New York, 1994).

[19] R. Der and M. Herrmann, in *1994 IEEE International Conference on Neural Networks* (IEEE, New York, 1994), Vol. 4, pp. 2472–2475.

[20] K. Otawara and L. T. Fan, in *Proceedings of the IEEE International Conference on Fuzzy Systems* (IEEE, New York, 1995), pp. 1943–1948.

[21] H. Szu and R. Henderson, in *, Proceedings of the International Joint Conference on Neural Networks* (IEEE, New York, 1993), pp. 1781–1784.

[22] P. M. Alsing, A. Gavrielides, and V. Kovanis, Phys. Rev. E **49**, 1225 (1994).

[23] K. Konishi and H. Kokame, Physica D **100**, 423 (1997).

[24] T. W. Frison, *1992 IEEE International Conference on Neural Networks* (IEEE , New York, 1992), Vol. 2, pp. 75–80.

[25] G. Chen and X. Dong, in *Proceedings of the IEEE International Symposium on Circuits and Systems* (IEEE, New York, 1995), pp. 1177–1182.

[26] A. Mulpur, S. Mulpur, and B. Jang, in *Proceedings of the Fourth IEEE Conference on Control Applications* (IEEE, New York, 1995), pp. 560–565.

[27] D. Lebender, J. Müller, and F. W. Schneider, J. Phys. Chem. **99**, 4992 (1995).

[28] R. Bakker, J. C. Schouten, F. Takens, and C. M. van den Bleek, Phys. Rev. E **54**, 3545 (1996).

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

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

[31] F. Takens, in *Dynamical Systems and Turbulence: Proceedings of the 1979–1980 Warwick Symposium*, edited by D. A. Rand and L.-S. Young, Lectures Notes in Mathematics Vol. 898 (Springer-Verlag, Berlin, 1981), p. 366.

[32] M. Hénon, Commun. Math. Phys. **50**, 69 (1976).

[33] E. A. Jackson, *Perspectives of Nonlinear Dynamics* (Cambridge University Press, New York, 1990), Vols. 1 and 2.

[34] The source code for the original version of SANE is available on the World Wide Web at http://net.cs.utexas.edu/users/nn/pages/software/software.html#sane