Robust Non-Linear Control through Neuroevolution

Faustino J. Gomez and Risto Miikkulainen
Department of Computer Sciences
The University of Texas
Austin, TX 78712, U.S.A
(inaki,risto@cs.utexas.edu)

TR AI-2002-292

Abstract

Many complex control problems require sophisticated solutions that are not amenable to traditional controller design. Not only is it difficult to model real world systems, but often it is unclear what kind of behavior is required to solve the task. Reinforcement learning (RL) approaches have made progress by utilizing direct interaction with the task environment, but have so far not scaled well to large state spaces and environments that are not fully observable. In recent years, neuroevolution, the artificial evolution of neural networks, has had remarkable success in tasks that exhibit these two properties, but, like RL methods, requires solutions to be discovered in simulation and then transferred to the real world. To ensure that transfer is possible, evolved controllers need to be robust enough to cope with discrepancies between these two settings. In this paper, we demonstrate how a method called Enforced SubPopulations (ESP), for evolving recurrent neural network controllers, can facilitate this transfer. The method is first compared to a broad range of reinforcement learning algorithms on very difficult versions of the pole balancing problem that involve large (continuous, high-dimensional) state spaces and hidden state. ESP is shown to be significantly more efficient and powerful than the other methods on these tasks. We then present a model-based method that allows controllers evolved in a learned model of the environment to successfully transfer to the real world. We test the method on the most difficult version of the pole balancing task, and show that the appropriate use of noise during evolution can improve transfer significantly by compensating for inaccuracy in the model.

1 Introduction

In many decision making processes such as manufacturing, aircraft control, and robotics researchers are faced with the problem of controlling systems that are highly complex, noisy, and unstable. A controller or *agent* must be built that observes the state of the system, or *environment*, and outputs a control signal that affects future states of the environment in some desirable way. For example, a guidance system designed to stabilize a rocket in flight must modulate the thrust of several engines in order to maximize altitude under variable atmospheric conditions. To succeed, this controller needs to be general and robust enough to respond effectively to conditions not explicitly considered or completely modeled by the designer.

The problem with designing or programming such controllers by conventional engineering methods is threefold:

I. **No mathematical model**. The environment is usually so high-dimensional, non-linear, and noisy that it is impossible to obtain the kind of accurate and tractable mathematical model required by these methods.

II. **No examples of correct behavior**. The task is complex enough that there is very little *a priori* knowledge of what constitutes a reasonable, much less optimal, control strategy. For a sophisticated task like rocket guidance, the designer knows the controller's general objective, but does not know how it should act from moment to moment in order to best achieve the objective.

III. **The transfer problem**. If a good control strategy is found, will it work on the real system being modeled? In general, it is not possible to predict how an agent will behave until it has begun to interact with its environment. Consequently, deciding what control features, designed in isolation, will yield the desired behavior when transferred to the real world can be very difficult.

The first two problems have compelled researchers to explore methods based on reinforcement learning (RL; Sutton and Barto 1998). Instead of trying to pre-program a response to every likely situation, the agent is made to learn the task by interacting with the environment. This way, the agent is said to be *grounded* in its environment (Harnad 1990); the actions that become part of the agent's behavior arise from and are validated by how they contribute to improved performance. In principle, RL methods can solve problems I and II: they do not require a mathematical model (i.e. the state transition probabilities) of the environment and can solve many problems where examples of correct behavior are not available. However, in practice, they have not scaled well to large state spaces or non-Markov tasks where the state of the environment is not fully observable to the agent. This is a serious problem because the real world is continuous (i.e. there are an infinite number of states) and artificial agents, like natural organisms, are necessarily constrained in their ability to fully perceive their environment.

Recently, methods based on evolutionary adaptation have shown promising results on continuous, non-Markov tasks (Gomez and Miikkulainen 1997, 1999; Nolfi and Parisi 1995; Yamauchi and Beer 1994). The first goal of this paper is to demonstrate that an architecture called Enforced Subpopulations (ESP) where neurons are evolved in separate subpopulations to form effective neural networks, is a particularly effective method. On a set of very difficult pole balancing tasks, we compare the performance of ESP to a wide range of learning systems including value function, policy search, and other evolutionary methods.

However, problem III is still an open issue in RL. Even though RL does not require a model, direct interaction with the environment is usually too slow (costly) due to the high data requirements of these methods (Bertsekas and Tsitsiklis 1996), and often too risky since the stability of most learning agent architectures cannot be guaranteed. Consequently, the control policy must first be learned off-line in a simulator or *simulation environment* and then be transferred to the actual *target environment* where it is ultimately meant to operate. Evolutionary approaches are just as dependent on simulation as other RL methods since it is impractical to evaluate entire populations of controllers in the real world. So far, transfer of evolved mobile robot controllers has been shown to be possible, but there is very little research on transfer in other classes of tasks, such as the control of unstable systems. The second goal of this paper is to analyze what factors influence transfer and show that transfer is possible even in high-precision tasks in unstable environments, such as the most difficult pole balancing task.

The paper is organized as follows: First, in section 2 we review the reinforcement learning problem and the conventional, single-agent methods of solving it that are in current use (2.1). Then we describe a fundamentally different RL approach, neuroevolution (2.2), that searches the space of neural network policies using a genetic algorithm. Cooperative coevolution (reviewed in section 2.3) is an advanced evolutionary method that evolves interacting subproblems to solve tasks more efficiently. Neuroevolution and cooperative coevolution are combined in the SANE algorithm (2.4) which forms the basis for our system, ESP. The last background section (2.5) reviews the current state of technology in the transfer of evolved controllers. In section 3, we present the ESP algorithm, and in section 4 the pole balancing task. ESP is compared to a variety of RL methods in section 5, and shown to solve harder versions of the problem faster. A methodology for transferring controllers to the real world is developed in section 6, showing that trajectory noise during training results in robust transfer.

2 Background and Related Work

Before introducing ESP, we first review four topics that our work is based on: reinforcement learning, neuroevolution, cooperative coevolution, and, most directly, the SANE algorithm.

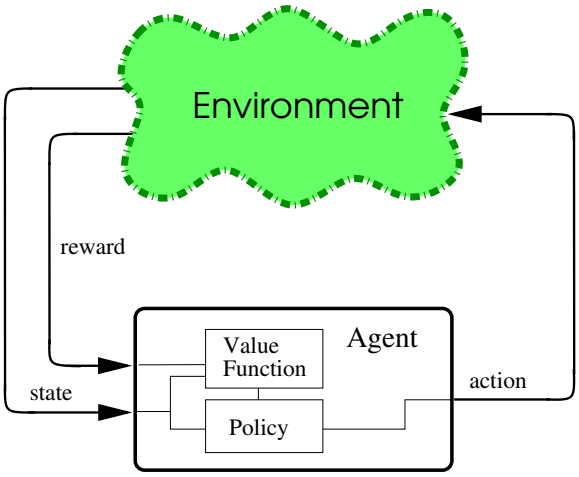


Figure 1: **The value function approach (color figure).** The agent is composed of a value-function and a policy. The value function tells the agent how much reward can be expected from each state if the best known policy is followed. The policy maps states to actions based on information from the value function.

2.1 The Reinforcement Learning Problem

Reinforcement learning refers to a class of algorithms for solving problems in which a sequence of decisions is made in order to maximize some measure of reward or *reinforcement* received from the environment. At each decision point, the agent, starting at some state $s \in S$, where S is the set of possible states, selects an action a from a set of possible actions, A, that transitions the environment to the next state s' and imparts a reinforcement signal, r, to the agent. Starting with little or no knowledge of how to solve the task, the agent explores the environment by trial-and-error. By associating reward with certain actions in each situation, the agent can gradually learn a course of action or *policy* that leads to favorable outcomes. This learning process is difficult because, unlike supervised learning tasks, the desired response in each state is not known in advance. An action that seems good in the short run may prove bad or even catastrophic down the road. Conversely, an action that is not good in terms of immediate payoff may prove beneficial or even essential for larger payoffs in the future. Therefore the agent must explore the state-space to try to associate actions to consequences in order to determine the best policy.

The best understood and most widely used learning methods for solving these problems are based on Dynamic Programming (Howard 1960). Essential to these methods is the *value function* V, which maps each problem state to its utility or *value* with respect to the task being learned (figure 1). This value is an estimate of the reward the agent can expect to receive if it starts in a particular state and follows the currently best known policy. As the agent explores the environment, it updates the value of each visited state according to the reward it receives. Given a value function that accurately computes the utility of every state, a controller can act optimally by merely selecting the action at each state that leads to the

subsequent state with the highest value. Therefore, the key to RL is finding the optimal value function for a given task.

RL control methods such as the popular Q-learning (Watkins 1989; Watkins and Dayan 1992), Sarsa (Rummery and Niranjan 1994), and TD(λ) (Sutton 1988) algorithms provide incremental procedures for computing V that are attractive because they do not require a model of the environment, can learn a policy by direct interaction, are naturally suited to stochastic environments, and are guaranteed to converge under certain conditions. These methods are based on Temporal Difference learning (Sutton and Barto 1998) in which the value of each state is updated by

$$V(s) := V(s) + \alpha[r + \gamma V(s') - V(s)]. \tag{1}$$

The estimate of the value of state s, $V(s)$, is incremented by the reward r from transitioning to state s' plus the difference between the discounted value of the next state $\gamma V(s')$ and $V(s)$, where α is the learning rate, γ is the discount factor and $0 \leq \alpha, \gamma \leq 1$. Rule 1 improves $V(s)$ by moving it towards the "target" $r + \gamma V(s')$ which is more likely to be correct because it uses the real reward r. To allow selecting the best action in each state (i.e. control) methods like Q-learning and Sarsa actually learn a Q-function instead of V, which gives the value of each state-action pair. The Q-function, in effect, caches the lookahead that would have to be performed to find the best action using V, allowing best policy for a given Q-function to be simply:

$$\underset{a \in A}{\operatorname{argmax}}\ Q(s, a), \tag{2}$$

In early research, these methods were investigated in very simple environments with very few states and actions. Subsequent work has focused on extending these methods to larger, high-dimensional and/or continuous environments. When the number of states and actions is relatively small, look-up tables can be used to represent V efficiently. But even with an environment of modest size this approach quickly becomes impractical and a function approximator is needed to map states to values. Typical choices range from local approximators such as the CMAC, case-based memories, and radial basis functions (Santamaria et al. 1998; Sutton 1996), to neural networks (Crites and Barto 1996; Lin 1993; Tesauro and Sejnowski 1987).

Despite substantial progress, value-function methods can be very slow, especially when reinforcement is sparse or when the environment is not completely observable. If the agent's sensory system does not provide enough information to determine the state (i.e. the *global* or *underlying process state*) then the decision process is non-Markov, and the agent must utilize a history or short-term memory of *observations*. This is important for most tasks of interest since a controller's sensors usually have limited range, resolution, and fidelity, causing *perceptual aliasing* where many observations that require different actions look the same. Next, we look at an approach that promises to be less susceptible to the problems outlined in this section.

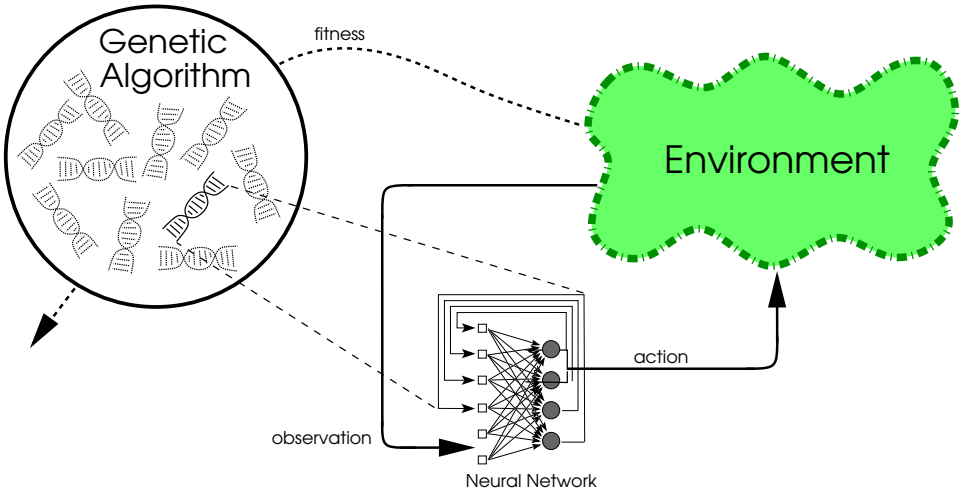


Figure 2: **Neuroevolution (color figure).** Each chromosome is transformed into a neural network phenotype and evaluated on the task. The agent receives input from the environment (observation) and propagates it through its neural network to compute an output signal (action) that affects the environment. At the end of the evaluation, the network is assigned a fitness according to its performance. The networks that perform well on the task are mated to generate new networks.

2.2 Neuroevolution

Neuroevolution (NE) presents a fundamentally different approach to reinforcement learning tasks. The basic idea of NE is to search the space of neural network policies directly using a genetic algorithm (figure 2). In contrast to conventional *ontogenetic* learning involving a single agent such as RL, evolutionary methods use a population of solutions. The individual solutions are not modified during evaluation; instead, adaptation arises through repeatedly recombining the population's most fit individuals in a kind of collective or *phylogenetic* learning. The population gradually improves as a whole until a sufficiently fit individual is found.

By searching the space of policies directly, NE eliminates the need for a value function and its costly computation. Instead, neural network controllers map observations from the environment directly to actions. This mapping is potentially powerful: neural networks are universal function approximators that can generalize and tolerate noise. Networks with feedback connections (i.e. recurrent networks) can maintain internal state extracted from a history of inputs, allowing them to solve non-Markov tasks. By evolving these networks instead of training them, NE avoids the problems of computational complexity and diminishing error gradients that affect recurrent network learning algorithms (Bengio et al. 1994). For NE to work, the environment need not satisfy any particular constraints—it can be continuous and non-Markov. All that concerns a NE system is that the network representations be large enough to solve the task and that there is an effective way to evaluate the relative quality of candidate solutions.

NE approaches differ primarily in how they encode neural network specifications into genetic strings. We will therefore use this dimension to classify and discuss NE methods. In NE, a chromosome can encode any relevant network parameter including synaptic weight values, size, connectivity (topology), learning rate, etc. The choice of encoding scheme affects the structure of the search space, the behavior of the search algorithm, and how the network genotypes are transformed into their phenotypes for evaluation.

There are two basic kinds of encoding schemes: direct and indirect. In direct encoding, the parameters are represented explicitly on the chromosome as binary or real numbers that are mapped directly to the phenotype. Many methods encode only the synaptic weight values (Belew et al. 1991; Gomez and Miikkulainen 1997; Jefferson et al. 1991) while others evolve topology as well (Moriarty 1997). Our method, ESP, uses a direct encoding scheme that does not evolve topology. However, since ESP evolves fully connected networks, virtually any topology of a given size can be represented by having some weights evolve to a value of zero.

Indirect encodings operate at a higher level of abstraction. Some simply provide a coarse description such as delineating a neuron's receptive field (Mandischer 1993) or connective density (Harp et al. 1989), while others are more algorithmic providing growth rules in the form of graph generating grammars (Kitano 1990; Voigt et al. 1993). These schemes have the advantage that very large networks can be represented without requiring large chromosomes. Cellular Encoding (CE; Gruau et al. 1996a,b) is a promising indirect method which we compare to ESP in the experiments below.

Whichever encoding scheme is used, neural network specifications are usually very high-dimensional so that large populations are required to find good solutions before convergence sets in. The next section reviews an evolutionary approach that potentially makes the search more efficient by decomposing the search space into smaller interacting spaces.

2.3 Cooperative Coevolution

In natural ecosystems, organisms of one species compete and/or cooperate with many other different species in their struggle for resources and survival. The fitness of each individual changes over time because it is coupled to that of other individuals inhabiting the environment. As species evolve they specialize and co-adapt their survival strategies to those of other species. This phenomenon of *coevolution* has been used to encourage complex behaviors in GAs.

Most coevolutionary problem solving systems have concentrated on competition between species (Darwen 1996; Miller and Cliff 1994; Paredis 1994; Pollack et al. 1996; Rosin 1997). These methods rely on establishing an "arms race" with each species producing stronger and stronger strategies for the others to defeat. This is a natural approach in areas such as game-playing where an optimal opponent is not available.

A very different kind of coevolutionary model emphasizes cooperation. Cooperative co-

evolution is motivated, in part, by the recognition that the complexity of difficult problems can be reduced through modularization (e.g. the human brain; Grady 1993). In cooperative coevolutionary algorithms the species represent solution subcomponents. Each individual forms a part of a complete solution but need not represent anything meaningful on its own. The subcomponents are evolved by measuring their contribution to complete solutions and recombining those that are most beneficial to solving the task. Cooperative coevolution can potentially improve the performance of artificial evolution by dividing the task into many smaller problems.

Early work in this area was done by Holland and Reitman (1978) in Classifier Systems. A population of rules was evolved by assigning a fitness to each rule based on how well it interacted with other rules. This approach has been used in learning classifiers implemented by a neural network, in coevolution of cascade correlation networks, and in coevolution of radial basis functions (Eriksson and Olsson 1997; Horn et al. 1994; Paredis 1995; Whitehead and Choate 1995). More recently, Potter and De Jong (1995) developed a method called Cooperative Coevolutionary GA (CCGA) in which each of the species is evolved independently in its own population. As in Classifier Systems, individuals in CCGA are rewarded for making favorable contributions to complete solutions, but members of different populations (species) are not allowed to mate. A particularly powerful idea is to combine cooperative coevolution with neuroevolution so that the benefits of evolving neural networks can be enhanced further through improved search efficiency. This is the approach taken by the SANE algorithm, described next.

2.4 Symbiotic, Adaptive Neuroevolution (SANE)

Conventional NE systems evolve genotypes that represent complete neural networks. SANE (Moriarty 1997; Moriarty and Miikkulainen 1996a) is a cooperative coevolutionary system that instead evolves neurons (i.e. partial solutions; figure 3). SANE evolves two different populations simultaneously: a population of neurons and a population of *network blueprints* that specify how the neurons are combined to form complete networks. Each generation of networks is formed using the blueprints, and evaluated on the task.

In SANE, neurons compete on the basis of how well, on average, the networks in which they participate perform. A high average fitness means that the neuron contributes to forming successful networks and, consequently, suggests that it cooperates well with other neurons. Over time, neurons will evolve that result in good networks. The SANE approach has proven faster and more efficient than other reinforcement learning methods such as Adaptive Heuristic Critic, Q-Learning, and standard neuroevolution, in, for example, the basic pole balancing task and in the robot arm control task (Moriarty and Miikkulainen 1996a,b).

SANE evolves good networks more quickly because the network sub-functions are allowed to evolve independently. Since neurons are not tied to one another on a single chromosome (i.e. as in conventional NE) a neuron that may be useful is not discarded if it happens to

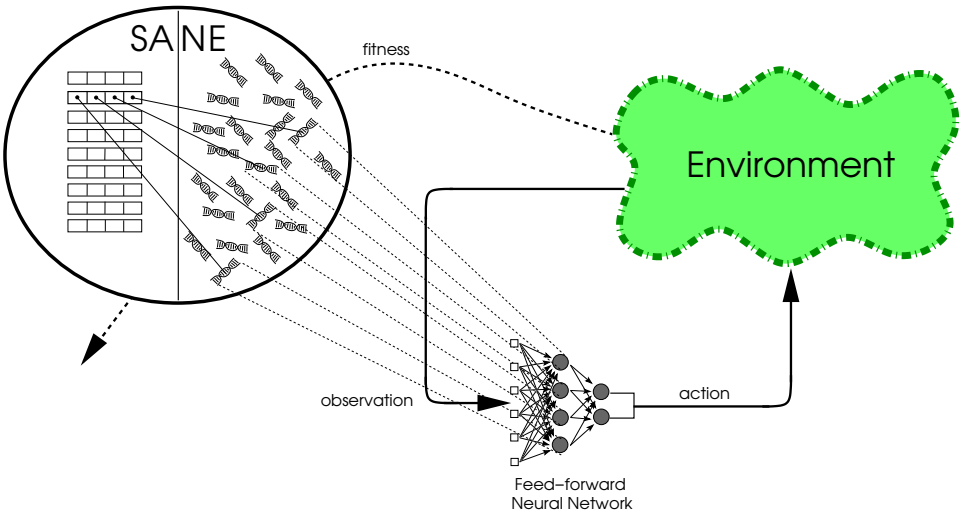


Figure 3: **Symbiotic, Adaptive Neuroevolution (color figure).** The algorithm maintains two distinct populations, one of network blueprints (left), and one of neurons (right). Networks are formed by combining neurons according to the blueprints. Networks are evaluated in the task, and the fitness is distributed among all the neurons that participated in the network. After all neurons are evaluated this way, recombination is performed on both populations.

be part of a network that performs poorly. Thus, more paths to a winning solution are maintained. Likewise, bad neurons do not get "free rides" by being part of a high scoring network. The system breaks the problem down to that of finding the solution to smaller, interacting subproblems.

Evolving neurons instead of full networks also maintains diversity in the population. If one type of neuron genotype begins to take over the population, networks will often be formed that contain several copies of that genotype. Because difficult tasks usually require several different hidden neurons, such networks cannot perform well. They incur low fitness, and the dominant genotype will be selected against, bringing diversity back into the population. In the advanced stages of SANE evolution, instead of converging the population around a single individual like a standard GA, the neuron population forms clusters of individuals that perform specialized functions in the target behavior (Moriarty 1997). This kind of implicit and automatic speciation is similar to more explicit methods such as fitness sharing that reduce the fitness of individuals that occupy crowded regions of the search space (Mahfoud 1995).

A key problem with SANE is that because it does not discriminate between the evolving specializations when it constructs networks and selects neurons for reproduction, evaluations can be very noisy. This limits its ability to evolve recurrent networks. A neuron's behavior in a recurrent network depends critically upon the neurons to which it is connected, and in SANE it cannot rely on being combined with similar neurons in any two trials. A neuron that behaves one way in one trial may behave very differently in another, and SANE cannot

obtain accurate fitness information. Without the ability to evolve recurrent networks, SANE is restricted to reactive tasks where the agent can learn to select the optimal action in each state based solely on its immediate sensory input. This is a serious drawback since most interesting tasks require memory. The method presented in section 3, ESP, extends cooperative neuroevolution to tasks that make use of short-term memory.

2.5 Transfer

Reinforcement learning requires a continuous interaction with the environment. In most tasks interaction is not feasible in the real world, and simulated environments must be used instead. However, no matter how rigorously they are developed, simulators cannot faithfully model all aspects of a target environment. Whenever the target environment is abstracted in some way to simplify evaluation, spurious features are introduced into the simulation. If a controller relies on these features to accomplish the task, it will fail to transfer to the real world where the features are not available (Mataric and Cliff 1996). Since some abstraction is necessary to make simulators tractable, such a "reality gap" can prevent controllers from performing in the physical world as they do in simulation.

Studying factors that lead to successful transfer is difficult because testing potentially unstable controllers can damage expensive equipment or put lives in danger. One exception is Evolutionary Robotics (ER), where the hardware is relatively inexpensive and the tasks have, up to now, not been safety critical in nature. Researchers in ER are well aware that it is often just as hard to transfer a behavior as it is to evolve it in simulation, and have devoted great effort to overcoming the transfer problem. Given the extensive body of work in this field, this section reviews the key issues and advances in transfer methods in ER.

By far the most widely used platform in ER is the Khepera robot (Mondada et al. 1993). Khepera is very popular because it is small, inexpensive, reliable, and easy to model due to its simple cylindrical design. Typically, behaviors such as phototaxis or "homing" (Ficici et al. 1999; Floreano and Mondada 1996; Jakobi et al. 1995; Lund and Hallam 1996; Meeden 1998), avoidance of static obstacles (Chavas et al. 1998; Jakobi et al. 1995; Miglino et al. 1995a), exploring (Lund and Hallam 1996), or pushing a ball (Smith 1998) are first evolved for a simulated Khepera controlled by a neural network that maps sensor readings to motor voltage values. The software controller is then downloaded to the physical robot where performance is measured by how well the simulated behavior is preserved in the real world.

Although these tasks (e.g. homing and exploring) are simple enough to solve with hand-coded behaviors, many studies have demonstrated that solutions evolved in idealized simulations transfer poorly. The most direct and intuitive way to improve transfer is to make the simulator more accurate. Instead of relying solely on analytical models, researchers have incorporated real-world measurements to empirically validate the simulation. Nolfi et al. (1994) and Miglino et al. (1995a) collected sensor and position data from a real Khepera robot and used to compute the sensor values and movements of its simulated counterpart. This approach improved the performance of transferred controllers dramatically by ensuring

a that the controller would experience states in simulation that actually occurred in the real world.

Unfortunately, as the complexity of tasks and the agents that perform them increases enough, it will not be possible to achieve sufficiently accurate simulations, and a fundamentally different approach is needed. Instead of trying to eliminate inaccuracies from the simulation, why not make the controllers less susceptible to them? For example, if noise is added to the controller's sensors and actuators during evaluation, they become more tolerant of noise in the real world and, therefore, less sensitive to discrepancies between the simulator and the target environment. The key is to find the right amount of noise: if there is not enough noise, the controller will rely on unrealistically accurate sensors and actuators. On the other hand, too much noise can amplify an irrelevant signal in the simulator that the controller will then not be able to find in the real world (Mataric and Cliff 1996). Correct noise levels are usually determined experimentally.

The most formal investigation of the factors that influence transfer was carried out by Jakobi (1993; 1998; 1995). He proposed using *minimal simulations* that concentrate on isolating a *base set* of features in the environment that are necessary for correct behavior. These features need to be made noisy to obtain robust control. Other features that are not relevant to the task are classified as *implementation* features which must me made unreliable (random) in the simulator so that the agent can not use them to perform the task. This way the robot will be very reliable with respect to the features that are critical to the task and not misled by those that are not. Minimal simulations provide a principled approach that can greatly reduce the complexity of simulations and improve transfer. However, so far they have only been used in relatively simple tasks. It is unclear whether this approach will be possible in more complex tasks where the set of critical features (i.e. the base set) is large or not easily identified (Watson et al. 1999).

While significant advances have been made in the transfer of robot controllers, it should be noted that the robots and environments used in ER are relatively "transfer friendly." Most significantly, robots like the Khepera are stable: in the absence of a control signal the robot will either stay in its current state or quickly converge to a nearby state due to momentum. Consequently, a robot can often perform competently in the real world as long as its behavior is preserved qualitatively after transfer. This is not the case with a great many systems of interest such as rockets, aircraft, and chemical plants that are inherently unstable. In such environments, the controller must constantly output a precise control signal to maintain equilibrium and avoid failure. Therefore, controllers for unstable systems may be less amenable to techniques that have worked for transfer in robots.

The transfer experiments in section 6 aim at providing a more general understanding of the transfer process including challenging problems in unstable environments. We have chosen pole balancing as the test domain for two reasons: (1) it embodies the essential elements of unstable systems while being simple enough to study in depth, and (2) it has been studied extensively, but in simulation only. This paper represents the first attempt to systematically study transfer outside of the mobile robot domain. However, before transfer

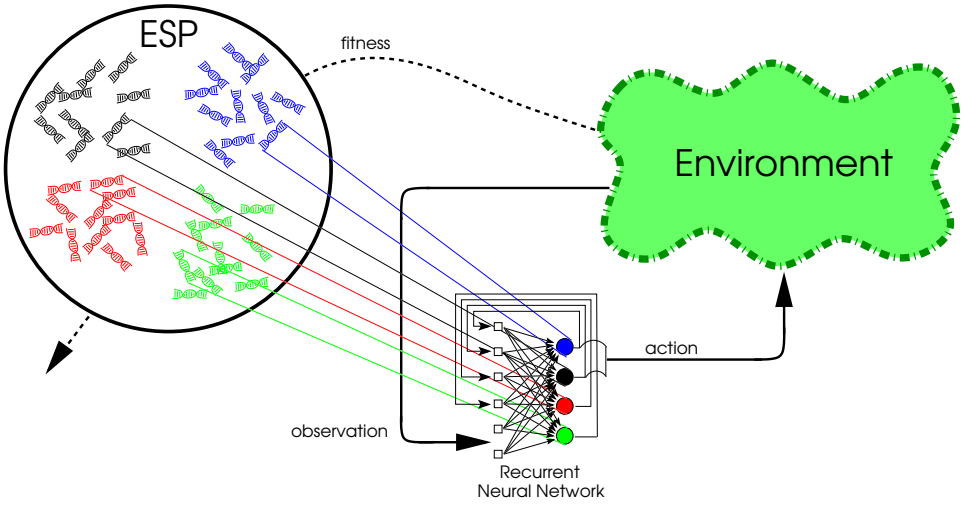


Figure 4: **The Enforced Subpopulations Method (ESP; color figure).** The population of neurons is segregated into subpopulations shown here in different colors. Networks are formed by randomly selecting one neuron from each subpopulation. As with SANE, a neuron accumulates a fitness score by adding the fitness of each network in which it participated. This score is then normalized and the best neurons within each subpopulation are mated to form new neurons.

can be studied, we will have to develop an RL method strong enough to robustly solve such problems. This will be done in the next three sections.

3 Enforced Subpopulations (ESP)

In the Enforced Subpopulations[1] (Gomez and Miikkulainen 1997, 1998, 1999) method, as in SANE, the population consists of individual neurons instead of full networks, and a subset of neurons are put together to form a complete network. However, in contrast to SANE, ESP makes use of explicit subtasks; a separate subpopulation is allocated for each of the u units in the network, and a neuron can only be recombined with members of its own subpopulation (figure 4). This way the neurons in each subpopulation can evolve independently, and rapidly specialize into good network sub-functions.

Evolution in ESP proceeds as follows:

1. Initialization. The number of hidden units u in the networks that will be formed is specified and a subpopulation of neuron chromosomes is created. Each chromosome encodes the input and output connection weights of a neuron with a random string of real numbers.

[1]The ESP package is available at:
`http://www.cs.utexas.edu/users/nn/pages/software/abstracts.html#esp-cpp`

2. Evaluation. A set of u neurons is selected randomly, one neuron from each subpopulation, to form a hidden layer of a feedforward network. The network is submitted to a *trial* in which it is evaluated on the task and awarded a fitness score. The score is added to the *cumulative fitness* of each neuron that participated in the network. This process is repeated until each neuron has participated in an average of e.g. 10 trials.

3. Recombination. The average fitness of each neuron is calculated by dividing its cumulative fitness by the number of trials in which it participated. Neurons are then ranked by average fitness within each subpopulation. Each neuron in the top quartile is recombined with a higher-ranking neuron using 1-point crossover and mutation at low levels to create the offspring to replace the lowest-ranking half of the subpopulation.

4. The Evaluation–Recombination cycle is repeated until a network that performs sufficiently well in the task is found.

ESP can evolve recurrent networks because the subpopulation architecture makes the evaluations more consistent, in two ways: first, the subpopulations that gradually form in SANE are already present by design in ESP. The species do not have to organize themselves out of a single large population, and their progressive specialization is not hindered by recombination across specializations that usually fulfill relatively orthogonal roles in the network. Second, because the networks formed by ESP always consist of a representative from each evolving specialization, a neuron is always evaluated on how well it performs its role in the context of all the other players. A neuron's recurrent connection weight r_i will always be associated with neurons from subpopulation S_i. As the subpopulations specialize, neurons evolve to expect, with increasing certainty, the kinds of neurons to which they will be connected. Therefore, the recurrent connections to those neurons can be adapted reliably.

Figure 5 illustrates the specialization process. The plots show the distribution of the neurons in the search space throughout the course of a typical evolution. Each point represents a neuron chromosome projected onto 2-D using Principal Component Analysis. In the initial population (Generation 1) the neurons, regardless of subpopulation, are distributed throughout the space uniformly. As evolution progresses, the neurons begin to form clusters which eventually become clearly defined and represent the different specializations used to form good networks.

The accelerated specialization caused by segregating neurons into in subpopulations not only allows for recurrent connections, it also makes ESP more efficient than SANE. The tradeoff is that diversity in ESP declines over the course of evolution like that of a normal GA. This can be a problem because a converged population cannot easily adapt to a new task. To deal with premature convergence ESP is combined with *burst mutation*. The idea is to search for optimal modifications of the current best solution. When performance has stagnated for a predetermined number of generations, new subpopulations are created by adding noise to each of the neurons in the best solution. Each new subpopulation contains neurons that represent differences from the best solution. Evolution then resumes, but now

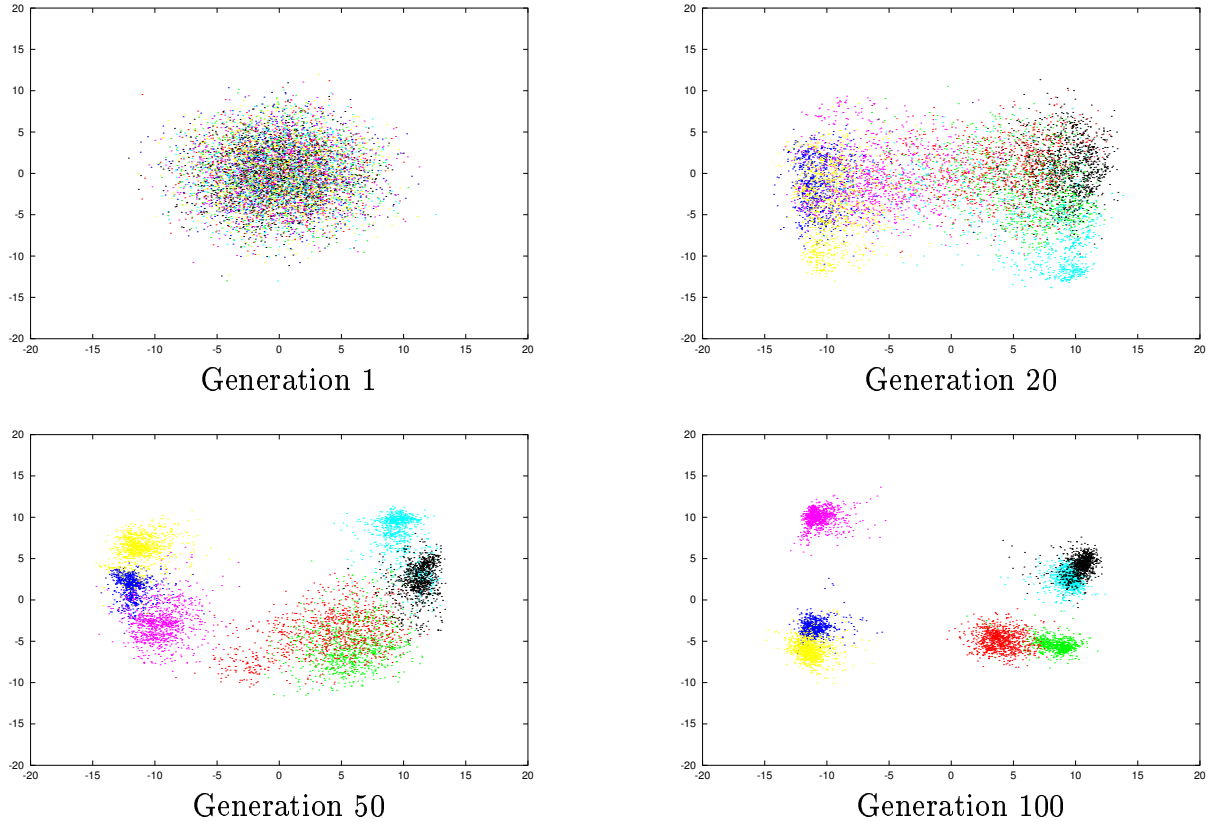


Figure 5: **Evolution of specializations in ESP (color figure).** The plots show a 2-D projection of the neuron weight vectors after Principal Component Analysis (PCA) transformation. Each subpopulation is shown in a different color. As evolution progresses, the subpopulations cluster into their own region of the search space. Each subpopulation represents a different neuron specialization that can be combined with others to form good networks.

searching the space in a "neighborhood" around the best previous solution. Burst mutation can be applied multiple times, with successive invocations representing differences to the previous best solution. Assuming the best solution already has some competence in the task, most of its weights will not need to be changed radically. To ensure that most changes are small while allowing for larger changes to some weights ESP uses the Cauchy distribution to generate noise:

$$f(x) = \frac{\alpha}{\pi(\alpha^2 + x^2)} \tag{3}$$

With this distribution 50% of the values will fall within the interval $\pm\alpha$ and 99.9% within the interval $318.3 \pm \alpha$. This technique of "recharging" the subpopulations keeps diversity in the population so that ESP can continue to make progress toward a solution even in prolonged evolution.

Burst mutation is similar to the Delta-Coding technique of Whitley et al. (1991) which was developed to improve the precision of genetic algorithms for numerical optimization

problems. Because our goal is to maintain diversity, we do not reduce the range of the noise on successive applications of burst mutation and we use Cauchy rather that uniformly distributed noise.

ESP does not evolve network topology because fully connected networks can effectively represent any topology of a given size (section 2.2). However, ESP does adapt the size of the networks. It is well known that when neural networks are trained using gradient-descent methods such as backpropagation, too many or too few hidden units can seriously affect learning and generalization. Having too many units can cause the network to memorize the training set, resulting in poor generalization. Having too few will slow down learning or prevent it altogether. Similar observations can be made when networks are evolved by ESP. With too few units (i.e. too few subpopulations) the networks will not be powerful enough to solve the task. If the task requires fewer units than have been specified, two things can happen: either each neuron will make only a small contribution to the overall behavior or, more likely, some of the subpopulations will evolve neurons that do nothing. The network will not necessarily overfit to the environment. However, too many units is still a problem because the evaluations will be slowed down unnecessarily, and evaluations will be noisier than necessary because a neuron will be sampled in a smaller percentage of all possible neuron combinations. Both of these problems result in inefficient search.

For these reasons ESP uses the following mechanism to add and remove subpopulations as needed: When evolution ceases to make progress, even after some number of burst mutations, take the best network found so far and evaluate it after removing each of its neurons in turn. If the fitness of the network does not fall below a threshold when missing neuron i, then i is not critical to the performance of the network and its corresponding subpopulation is removed. If no neurons can be removed, add a new subpopulation of random neurons and evolve networks with one more neuron.

This way, ESP will enlarge networks when the task is too difficult for the current architecture and prune units (subpopulations) that are found to be ineffective. Overall, with growth and pruning, ESP will be more robust in dealing with environmental changes and tasks where the appropriate network size is difficult to determine.

4 Experimental Setup

We conducted three different types of experiments using the pole balancing domain. Sections 4.1 and 4.2 describe the domain and task setup in detail. The first set of experiments, in section 5, evaluates how efficiently ESP can evolve effective controllers. We compare ESP to a broad range of learning algorithms on a sequence of increasingly difficult versions of the pole balancing task. This scheme allows us to compare methods at different levels of task complexity, exposing the strengths and limitations of each method with respect to specific challenges introduced by each succeeding task. In section 6, we present a model-based approach to applying ESP to real-world tasks, and evaluate how enhancing robustness in

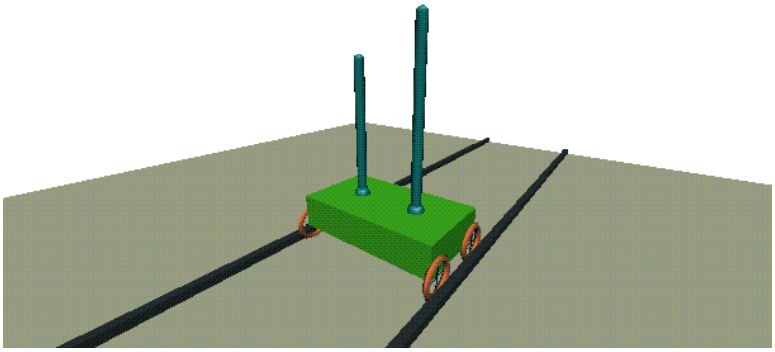


Figure 6: **The double pole balancing system (color figure).** Both poles must be balanced simultaneously by applying a continuous force to the cart. The system becomes more difficult to control as the poles assume similar lengths and if the velocities are not provided to the controller. The figure is a snapshot of a 3D real-time simulation. Demo available at `http://www.cs.utexas.edu/users/inaki/esp/two-pole-demo.`

simulation facilitates transfer to the target environment.

4.1 The Pole Balancing Problem

The basic pole balancing or inverted pendulum system consists of a pole hinged to a wheeled cart on a finite stretch of track. The objective is to apply a force to the cart at regular intervals such that the pole is balanced indefinitely and the cart stays within the track boundaries. This problem has long been a standard benchmark for artificial learning systems. For over 30 years researchers in fields ranging from control engineering to reinforcement learning have tested their systems on this task (Anderson 1989; Michie and Chambers 1968; Schaffer and Cannon 1966). There are two primary reasons for this longevity: (1) Pole balancing has intuitive appeal. It is a continuous real-world task that is easy to understand and visualize. It can be performed manually by humans and implemented on a physical robot. (2) It embodies many essential aspects of a whole class of learning tasks that involve temporal credit assignment. The controller is not given a strategy to learn, but instead must discover its own from the reinforcement signal it receives every time it fails to control the system. In short, it is an elegant artificial learning environment that is a good surrogate for a more general class of problems involving unstable systems like bipedal robot walking (Vukobratovic 1990), and satellite attitude control (Dracopoulos 1997).

This long history notwithstanding, the relatively recent success of modern reinforcement learning methods on control learning tasks has rendered the basic pole balancing problem obsolete. It can now be solved so easily that it provides little or no insight about a system's ability. This is especially true for neuroevolution systems which often find solutions in the initial random population (Gomez and Miikkulainen 1997; Moriarty and Miikkulainen 1996a).

To make it challenging for modern methods, a variety of extensions to the basic pole-balancing task have been suggested. Wieland (1991) presented several variations that can be grouped into two categories: (1) modifications to the mechanical system itself, such as adding a second pole either next to or on top of the other, and (2) restricting the amount of state information that is given to the controller; for example, only providing the cart position and the pole angle. The first category renders the task more difficult by introducing non-linear interactions between the poles. The second makes the task non-Markovian, requiring the controller to employ short term memory to disambiguate underlying process states. Together, these extensions represent a family of tasks that can effectively test algorithms designed to solve difficult control problems.

The most challenging of the pole balancing versions is a double pole configuration (figure 6) where two poles of unequal length must be balanced simultaneously. Even with complete state information, this problem is very difficult, requiring precise control to solve. When state information is incomplete, the task is even more difficult because the controller must in addition infer the underlying state.

The sequence of comparisons presented below begins with a single pole version and then moves on to progressively more challenging variations. The final task is a two-pole version that involves perceptual aliasing due to incomplete state information. This task allows controllers to be tested on perceptual aliasing not by isolating it in a discrete toy environment, but by including it as an additional dimension of complexity in an already difficult, non-linear, high-dimensional, and continuous control task. The next section describes the general setup that was used for all experiments in this paper.

4.2 Task Setup

The pole balancing environment was implemented using a realistic physical model with friction, and fourth-order Runge-Kutta integration with a step size of 0.01s (see Appendix A for the equations of motion and parameters used). The state variables for the system are the following:

x : position of the cart.
$\dot{x}$: velocity of the cart.
θ_i : angle of the i-th pole ($i = 1, 2$).
$\dot{\theta}_i$: angular velocity of the i-th pole.

Figure 7 shows how the network controllers interact with the pole balancing environment. At each time-step (0.02 seconds of simulated time) the network receives the state variable values scaled to [-1.0, 1.0]. This input activation is propagated through the network to produce a signal from the output unit that represents the amount of force used to push the the cart. The force is then applied and the system transitions to the next state which becomes the new input to the controller. This cycle is repeated until a pole falls or the cart

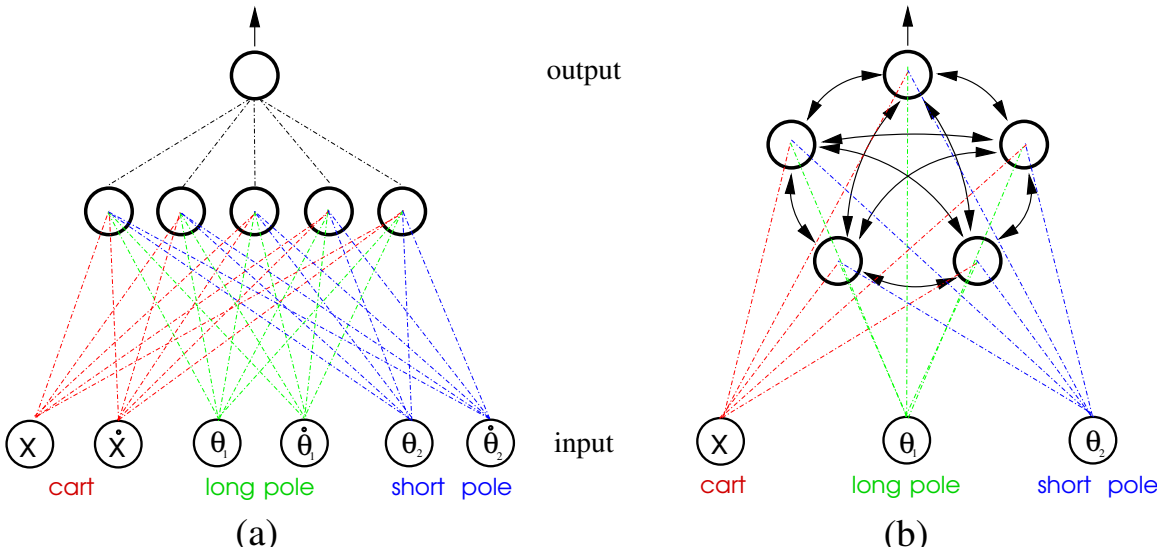


Figure 7: **Neural network control of the pole balancing system (color figure).** At each time step the network receives the current state of the cart-pole system $(x, \dot{x}, \theta_1, \dot{\theta_1}, \theta_2, \dot{\theta_2})$ through its input layer. For the feed-forward networks (a) used in the Markov tasks (1a and 2a), the input layer activation is propagated forward through the hidden layer of neurons to the output unit. For the recurrent networks (b) used in the non-Markov tasks (1b and 2b), the neurons do not receive the velocities $(\dot{x}, \dot{\theta_1}, \dot{\theta_2})$, instead they must use their feedback connections to determine which direction the poles are moving. The feedback connections provide each neuron with the activation of the other neurons from the previous time step. A force is applied to the cart according to the activation of the output unit. For the single pole version the network only has inputs for the cart and long pole.

goes off the end of the track. In keeping with the setup in several other papers (e.g. Gruau et al. 1996a; Wieland 1991) we restrict the force to be no less than $\pm 1/256 \times 10$ Newtons so that the controllers cannot maintain the system in unstable equilibrium by outputting a force of zero when the poles are vertical.

5 Pole Balancing Comparisons

Many algorithms have been developed for solving reinforcement learning problems. However, to our knowledge there have been no comparisons that both span a cross-section of current technology and address tasks of significant difficulty. The comparisons below are intended to close this gap: they establish a ranking of various approaches with respect to a challenging set of benchmark tasks. The tasks include the following four pole balancing configurations of increasing difficulty:

1. One Pole
   (a) Complete state information

(b) Incomplete state information

2. Two Poles
   (a) Complete state information
   (b) Incomplete state information

Task 1a is the classic one-pole configuration except that the control signal (i.e. action space) is continuous rather than "bang-bang." In 1b, the controller only has access to 2 of the 4 state variables: it does not receive the velocities $(\dot{x}, \dot{\theta})$. In 2a, the system now has a second pole next to the first, making the state-space 6-dimensional. Task 2b, like 1b, is non-Markov with the controller only seeing x, θ_1, and θ_2. Fitness was determined by the number of time steps a network could keep both poles within a specified angle from vertical and the cart between the ends of the track. A task was considered solved if a network could do this for 100,000 time steps, which is equal to over 30 minutes in simulated time. All simulations were run on a 600mHz Intel Pentium III.

5.1 Other Methods

ESP was compared to eight other learning methods in the pole balancing domain. The first three are value-function methods for which we ran our own simulations. Q-MLP is our implementation, and the two SARSA methods are implementations of Santamaria et al. (1998), adapted to the pole balancing problem. The other five methods are policy search methods. With the exception of SANE and CNE, the results for these methods were taken from the published work cited below. Data was not available for all methods on all tasks: however, in all such cases the method is shown to be significantly weaker already in a previous, easier task. The parameter settings for each method on each task are listed in Appendix B.

Value Function Methods

The three value function methods each use a different kind of function approximator to represent a Q-function that can generalize across the continuous space of state-action pairs. Although these approximators can compute a value for any state-action pair, they do not implement true continuous control since the policy is not explicitly stored. Instead, continuous control is approximated by discretizing the action space at a resolution that is adequate for the problem. In order to select the optimal action a for a given state s, a *one-step* search in the action space is performed. The control agent selects actions according to an *ϵ-greedy* policy: with probability $1 - \epsilon$, $0 \leq \epsilon < 1$, the action with the highest value (equation 2) is selected, and with probability ϵ, the action is random. This policy allows some exploration so that information can be gathered for all actions. In all simulations the controller was tested every 20 trials with ϵ=0 and learning turned off to determine whether a solution had been found.

Q-learning with MLP (Q-MLP): This method is the basic Q-learning algorithm (Watkins and Dayan 1992) that uses a Multi-Layer Perceptron (i.e. an artificial neural network) to map state–action pairs to values $Q(s, a)$. The input layer of the network has one unit per state variable and one unit per action variable. The output layer consists of a single unit indicating the Q-value. Values are learned through gradient descent on the prediction error using the backpropagation algorithm. This kind of approach has been studied widely with success in tasks such as pole-balancing (Lin and Mitchell 1992), pursuit-evasion games (Lin 1992), and backgammon (Tesauro 1992).

Sarsa(λ) with Case-Based function approximator (SARSA-CABA; Santamaria et al. 1998): This method consists of the Sarsa on-policy Temporal Difference control algorithm with eligibility traces that uses a case-based memory to approximate the Q-function. The memory explicitly records state-action pairs (i.e. cases) that have been experienced by the controller. The value of a new state-action pair not in the memory is calculated by combining the values of the k-nearest neighbors. A new case is added to the memory whenever the current query point is further than a specified *density threshold*, t_d away from all cases already in the memory. The case-based memory provides locally-linear model of the Q-function that concentrates its resources on the regions of the state space that are most relevant to the task and expands its coverage dynamically according to t_d.

Sarsa(λ) with CMAC function approximator (SARSA-CMAC; Santamaria et al. 1998): This is the same as SARSA-CABA except that it uses a Cerebellar Model Articulation Controller (CMAC; Albus 1975; Sutton 1996) instead of a case-based memory to represent the Q-function. The CMAC partitions the state-action space with a set of overlapping tilings. Each tiling divides the space into a set of discrete *features* which maintain a value. When a query is made for a particular state-action pair, its Q-value is returned as the sum of the value in each tiling corresponding to the feature containing the query point. SARSA-CABA and SARSA-CMAC have both been applied to the pendulum swing-up task and the double-integrator task.

Policy Search Methods

Policy search methods search the space of action policies directly without maintaining a value function. In this study, all except VAPS are evolution based approaches that maintain a population of candidate solutions and use genetic operators to transform this set of search points into a new, possibly better, set. VAPS is a single solution (agent) method, and, therefore, shares much in common with the value-function methods.

Value and Policy Search (VAPS; Meuleau et al. 1999) extends the work of Baird and Moore (1999) to policies that can make use of memory. The algorithm uses stochastic gradient descent to search the space of finite policy graph parameters. A policy graph

is a state automaton that consists of nodes labeled with actions that are connected by arcs labeled with observations. When the system is in a particular node the action associated with that node is taken. This causes the underlying Markov environment to transition producing an observation that determines which arc is followed in policy graph to the next action node.

Symbiotic, Adaptive Neuro-Evolution (SANE; Moriarty 1997) described in section 2.4.

Conventional Neuroevolution (CNE) is our implementation of single-population Neuroevolution similar to the algorithm used in Wieland (1991). In this approach, each chromosome in the population represents a complete neural network. CNE differs from Wieland's algorithm in that (1) the network weights are encoded with real instead of binary numbers, (2) it uses rank selection, and (3) it uses burst mutation. CNE is like ESP except that it evolves at the network level instead of the neuron level, and therefore provides a way to isolate the performance advantage of cooperative coevolution (ESP) over a single population approach (CNE).

Evolutionary Programming (EP; Saravanan and Fogel 1995) is a general mutation-based evolutionary method that can be used to search the space of neural networks. Individuals are represented by two n dimensional vectors (where n is the number of weights in the network): $\vec{x}$ contains the synaptic weight values for the network, and $\vec{\delta}$ is a vector of standard deviation values of $\vec{x}$. A network is constructed using the weights in $\vec{x}$, and offspring are produced by applying Gaussian noise to each element $\vec{x}(i)$ with standard deviation $\vec{\delta}(i), i \in \{1..n\}$.

Cellular Encoding (CE; Gruau et al. 1996a,b) uses Genetic Programming (GP; Koza 1991) to evolve graph-rewriting programs. The programs control how neural networks are constructed out of "cell." A cell represents a neural network processing unit (neuron) with its input and output connections and a set of registers that contain synaptic weight values. A network is built through a sequence of operations that either copy cells or modify the contents of their registers. CE uses the standard GP crossover and mutation to recombine the programs allowing evolution to automatically determine an appropriate architecture for the task and relieve the investigator from this often trial-and-error undertaking.

5.2 Balancing One Pole

This task is the starting point for our comparisons. Balancing one pole is a relatively easy problem that gives us a base performance measurement before we move to the much harder two-pole task. It has also been solved with many other methods and therefore serves to put the results in perspective with prior literature.

Method	Evaluations	CPU time (sec)
Q-MLP	2056	53
SARSA-CMAC	540	487
SARSA-CABA	965	1713
CNE	352	5
SANE	302	5
ESP	289	4

Table 1: **One pole with complete state information.** Comparison of various learning methods on the basic pole balancing problem with continuous control. Results for all methods are averages of 50 runs.

Complete State Information

Table 1 shows the results for the single pole balancing task where the controller receives complete state information. Although this is the easiest task in the suite, it is more difficult than the standard bang-bang control version commonly found in the literature. Here the controller must output a control signal within a continuous range.

None of the methods has a clear advantage if we look at the number of evaluations alone. However, in terms of CPU time the evolutionary methods show remarkable improvement over the value-function methods. The computational complexity associated with evaluating and updating values can make value-function methods slow, especially in continuous action spaces. With continuous actions the function approximator must be evaluated $O(|A|)$ times per state transition to determine the best action-value estimate (where A is a finite set of actions). Depending on the kind of function approximator such evaluations can prove costly. In contrast, evolutionary methods do not update any agent parameters during interaction with the environment and only need to evaluate a function approximator once per state transition since the policy is represented explicitly.

The notable disparity in CPU time between Q-MLP and the SARSA methods is related to the efficiency of their respective function approximators. The MLP provides a compact representation that can be evaluated quickly, while the CMAC and case-based memory are coarse-codings whose memory requirements and evaluation cost grow exponentially with the dimensionality of the state space.

The performance of the CNE, SANE and ESP is nearly indistinguishable: this task poses very little difficulty for the NE methods.

Incomplete State Information

This task is identical to the first task except the controller only senses the cart position x and pole angle θ. Therefore, the underlying states $\{x, \dot{x}, \theta, \dot{\theta}\}$ are hidden and the networks need to be recurrent so that the velocities can be computed internally using feedback connections. This makes the task significantly harder since it is more difficult to control the system

Method	Evaluations	CPU time	%Success
VAPS	(500000)	(5days)	(0)
Q-MLP	11331	340	100
SARSA-CMAC	13562	2034	59
SARSA-CABA	15617	6754	70
CNE	724	15	100
ESP	589	11	100

Table 2: **One pole with incomplete state information.** The table shows the number of evaluations, CPU time, and success rate of the various methods. Results are the average of 50 simulations, and all differences are statistically significant. The results for VAPS are in parenthesis since only a single unsuccessful run according to our criteria was reported by Meuleau et al. (1999).

when the concomitant problem of velocity calculation must also be solved. And, for the NE methods, the number of connections in the networks is necessarily larger which expands the size of the search space.

To allow Q-MLP and the SARSA methods to solve this task, we extended their inputs to include the immediately previous cart and pole positions (x_{t-1}, θ_{t-1}) in addition to x_t and θ_t. This *delay window* of depth 1 is sufficient to disambiguate process states (Lin and Mitchell 1992). For VAPS, the state-space was partitioned into unequal intervals, 8 for x and 6 for θ, with the smaller intervals being near the center of the value ranges (Meuleau et al. 1999).

Table 2 compares the various methods in this task. The table shows the number of evaluations and average CPU time for the successful runs. The rightmost column is the percentage of simulations that resulted in the pole being balanced for 10^6 time steps (%Success).

The results for VAPS are in parenthesis in the table because only a single run was reported by Meuleau et al. (1999). It is clear, however, that VAPS is the slowest method in this comparison, only being able to balance the pole for around 1 minute of simulated time after several days of computation (Meuleau et al. 1999). The evaluations and CPU time for the SARSA methods are the average of the successful runs only. Of the value-function methods Q-MLP fared the best, reliably solving the task and doing so much more rapidly than SARSA.

ESP and CNE were two orders of magnitude faster than VAPS and SARSA and one order of magnitude faster that Q-MLP. The performance of the two NE methods degrades only slightly compared to the previous task. ESP was able to balance the pole for over 30 minutes of simulated time usually within 10 seconds of learning CPU time, and do so reliably.

5.3 Balancing Two Poles

The first two tasks show that the single pole environment poses no challenge for modern reinforceme nt learning methods. The double pole problem is a more adequate test envi-

Method	Evaluations	CPU time (sec)
Q-MLP	10,582	153
CNE	22,100	73
EP	307,200	—
SANE	12,600	37
ESP	3,800	22

Table 3: **Two poles with complete state information.** The table shows the number of pole balancing attempts (evaluations) and CPU time required by each method to solve the task. Evolutionary Programming data is taken from Saravanan and Fogel (1995). Q-MLP, CNE, SANE, and ESP data are the average of 50 simulations, and all differences are statistically significant.

ronment for these methods, representing a significant jump in difficulty. Here the controller must balance two poles of different lengths (1m and 0.1m) simultaneously. The second pole adds two more dimensions to the state-space $(\theta_2, \dot{\theta}_2)$ and complex non-linear interactions between the poles.

Complete State Information

For this task, ESP was compared with Q-MLP, CNE, SANE and the published results of EP. Despite extensive experimentation with many different parameter settings, we were unable to get the SARSA methods to solve this task within 12 hours of computation.

Table 3 shows the results for the two-pole configuration with complete state information. These results suggest that NE methods based on partial solutions (i.e. SANE and ESP) are superior to the other neuroevolution methods in terms of learning speed. Q-MLP compares very well to the NE methods with respect to evaluations, in fact, better than on task 1b, but again lags behind ESP and SANE by nearly an order of magnitude in CPU time. Finally, ESP is faster than SANE by a factor of 2.

Incomplete State Information

Although the previous task is difficult, the controller has the benefit of complete state information. In this task, as in task 1b, the controller does not have access to the velocities, i.e. it does not know how fast or in which direction the poles are moving. To compensate, it has to disambiguate the state based on memory.

Gruau et al. (1996a) is the only study we know that has solved the two-pole problem without velocity information. SANE is not suited to non-Markov problems and none of the value-function methods we tested made noticeable progress on the task after approximately 12 hours of computation. Therefore, in this task, only CNE, ESP, and the Cellular Encoding (CE) method are compared.

The same fitness function was used as in Gruau et al. (1996a). The function $\mathcal{F}$ is the weighted sum of two separate fitness measurements $(0.1f_1 + 0.9f_2)$ taken over a simulation

Method	Evaluations	Generalization
CE	840,000	300
CNE	786,227	285
ESP	169,466	289

Table 4: **Two poles with incomplete state information.** Table shows the number of evaluations and the number of test cases out of 625 that could be controlled (generalization) for each of the three methods. Results for CNE and ESP are the average of 20 simulations. Results for CE taken from Gruau et al. (1996a).

of 1000 time steps:

$$f_1 = t/1000, \tag{4}$$

$$f_2 = \begin{cases} 0 & \text{if } t < 100 \\ \left(\dfrac{K}{\sum_{i=t-100}^{t} (|x^i| + |\dot{x}^i| + |\theta_1^i| + |\dot{\theta}_1^i|)} \right) & \text{otherwise,} \end{cases} \tag{5}$$

where t is the number of time steps the pole was balanced, K is a constant (set to 0.75), and the denominator in equation 5 is the sum of the absolute values of the cart and long pole state variables, summed over the last 100 time steps of the run. This complex fitness is intended to force the network to compute the pole velocities by favoring controllers that can keep the poles near the equilibrium point and minimize the amount of oscillation. Gruau et al. (1996a) devised this fitness because the simple fitness measure could produce networks that would balance the poles by merely swinging them back and forth (i.e. without calculating the velocities).

According to Gruau et al., the main benefit of Cellular Encoding is that it liberates the user from having to search for and specify an appropriate network architecture for the problem. While ESP does incrementally resize networks it does not recombine networks of different size (i.e. it does not search in the space of network topologies). ESP begins evolution with networks of a user-defined size, and therefore this could give it an unfair advantage by hiding the purported work needed to discover a good architecture for this task. To remedy this, the number of hidden units u is chosen at random at the beginning of each evolution instead of being prescribed by the user; effectively taking the user out of the decision making process. In keeping with the network size constraint in Gruau et al. (1996a), we restrict u to $\{1..9\}$. For consistency, CNE was used with this method as well.

In addition to learning speed, the robustness of the solutions was also tested. A total of 625 test cases were generated by allowing the state variables $x, \dot{x}, \theta_1$, and $\dot{\theta}_1$ to take on the values: 0.05, 0.25, 0.5, 0.75, 0.95, scaled to the appropriate range for each variable ($5^4 = 625$). These ranges were ± 2.16m for x, ± 1.35m/s for $\dot{x}$, ± 3.6 deg for θ_1, and ± 8.6 deg/s for $\dot{\theta}_1$. This test, first introduced by Dominic et al. (1991), has become a standard for evaluating the generality of solutions in pole balancing. A high score indicates that a solution has

competence in a wide area of the state space.

Table 4 compares the performance of ESP, CNE, and CE in this task. The column labeled "Generalization" refers to each method's average score on this test. A successful controller is awarded a point for each of the 625 different initial states from which it is able to control the system for 1000 time steps. To determine if the task had been solved, we tested the most fit individual from each generation to see if it could balance the pole for 100,000 time steps and score at least 250 on the generalization test describe below. This was necessary because we found that fitness $\mathcal{F}$ did not correlate well with the ability to generalize to novel initial states.

ESP required approximately five times fewer evaluations than both CNE and CE to produce controllers with comparable generalization.

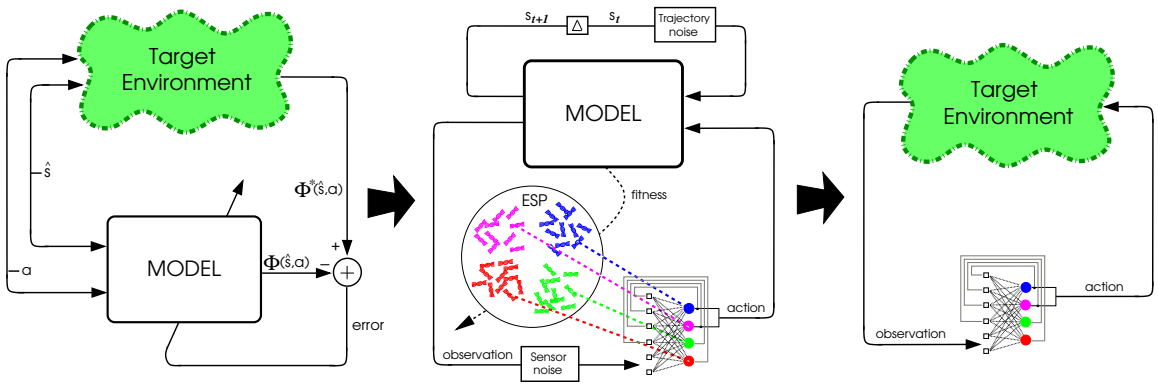
5.4 Summary of comparisons

The results of the comparisons show that neuroevolution is more efficient and more powerful than value function methods in this set of tasks. The best value function method in task 1a required an order of magnitude more CPU time than NE, and the transition from 1a to 1b represented a significant challenge, causing some of them to fail and others to take 30 times longer than NE. Only Q-MLP was able to solve task 2a and none of the value function methods could solve task 2b. In contrast, all of the evolutionary methods scaled up to the most difficult tasks, with ESP increasing its lead the more difficult the task became.

While it is important to show that ESP can efficiently evolve controllers that work in simulation, the purpose of doing so is to eventually put them to use in the real world. Successful transfer is critical to the viability of evolutionary methods as a useful problem solving paradigm.

6 Transferring to the target environment

In the experiments so far, a controller was considered successful if it solved the task during evolution in the simulator. That is, the *simulation environment* used for training was also the *target environment* used to test the final solution. For real-world tasks, these two environments are necessarily distinct because it is too time consuming to evaluate populations of candidate controllers through direct interaction with the target environment. Moreover, interaction with the real system is not possible in many domains where evaluating the potentially unstable controllers during evolution can be costly or dangerous. Therefore, in order to apply NE (or other RL methods) to real-world control problems, controllers must be evolved in a model of the target environment and then transferred to the actual target environment. In order to ensure that transfer is successful, the controllers must be robust enough to tolerate discrepancies that may exist between the simulation and target environments.



1: Learn model 2: Evolve using model 3: Transfer controller

Figure 8: **The model-based neuroevolution approach (color figure).** Step 1: Learn a model of the target environment; in this case, the double pole balancing system. The model (e.g. a supervised neural network) receives the a state $\hat{s}$ and an action a as its input and produces a prediction of the next state $\Phi(s, a)$ as its output. The error between the prediction and the correct next state $\Phi^*(s, a)$ is used to improve the prediction. Step 2: Evolve a controller using the model instead of the target environment. The boxes labeled "trajectory noise" and "sensory noise" add uniform noise to the signal passing through them. These noise sources make controllers more robust and facilitate transfer. Step 3: Test the controller on the target environment. This methodology allows controllers for complex, poorly understood dynamical systems to be evolved in simulation and then transferred successfully.

In this section, we study the problem of transferring controllers for unstable, non-linear systems. We continue with the non-Markov two-pole task (task 2b), but instead of using the analytical system (i.e. the differential equations in Appendix A) as the simulation environment it is now the target environment.

Figure 8 shows the three steps that constitute our approach. First, the target environment is modeled with a supervised neural network to produce a simulation environment (step 1). Second, ESP uses the model as the simulation environment to evolve a controller (step 2). Third, this controller is transferred to the target environment (step 3). In the usual application of this method, the target environment is the physical system, and nothing needs to be known about it as long as its interaction with a controller can be observed. However, in this study, by treating the analytical system as if it were the real two-pole system, we can test controller transfer exactly with complete knowledge of the system. This way it is possible to systematically study conditions for successful transfer in a controlled setting and gain insights that would be difficult to obtain from an actual transfer to a physical system.

Sections 6.1, 6.2, and 6.3 present steps 1, 2, and 3 of our method respectively. In section 6.4 we test the robustness of transferred solutions, and in section 6.5 analyze results of the experiment.

6.1 Learning the simulation model

The purpose of building a model is to provide a simulation environment so that evolution can be performed efficiently. While it is possible to develop a tractable mathematical model for some systems, most real world environments are too complex or poorly understood to be captured analytically. A more general approach is to learn a *forward model* using observations of target environment behavior. The model, Φ, approximates the discrete-time state-transition mapping, Φ^*, of the target environment:

$$\widehat{s}_{t+1} = \Phi^*_\delta(\widehat{s}_t, a_t), \quad \forall \widehat{s} \in \widehat{S}, a \in A, \tag{6}$$

where $\widehat{S}$ is the set of possible states in the target environment, and $\widehat{s}_{t+1}$ is the state of the system time $\delta > 0$ in the future if action a is taken in state $\widehat{s}_t$. The function Φ^* is in general unknown and can only be sampled. The parameter δ controls the period between the time an action is taken and the time the resulting state is observed. Since the objective is to control the system it makes sense to set δ to the time between control actions. Using Φ, ESP can simulate the interaction with the target environment by iterating

$$s_{t+1} = \Phi_\delta(s_t, \pi(s_t)), \tag{7}$$

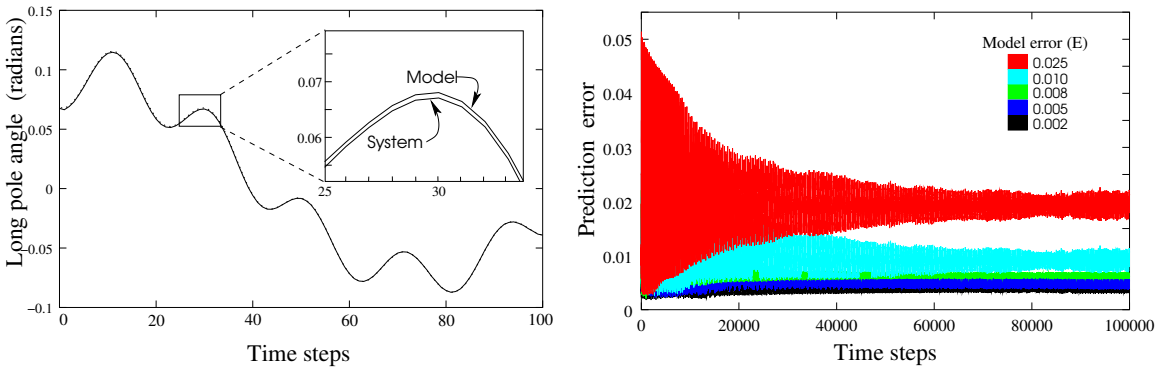
where π is the control policy and $s \in S$ are the states of the simulator.

In modeling the two-pole system we followed a standard neural network system identification approach (Barto 1990). Figure 8a illustrates the basic procedure for learning the model. The model is represented by a feed-forward neural network (MLP) that is trained on a set of state transition examples obtained from Φ^*. State-action pairs are presented to the model, which outputs a prediction $\Phi(\widehat{s}, a)$ of the next state $\Phi^*(\widehat{s}, a)$. The error between the correct next state and the model's prediction is used to improve future predictions using backpropagation.

The MLP is a good choice of representation for modeling dynamical systems because it makes few assumptions about the structure of Φ^*, except that it be a continuously differentiable function. This architecture allows us to construct a model quickly using relatively few examples of correct behavior. Furthermore, when modeling a real system these examples can be obtained from an existing controller (i.e. the controller upon which we are trying to improve).

A training set of 500 example state transitions was generated by sampling the state space, $\widehat{S} \subset \Re^6$, and action space, $A \subset \Re$, of the two-pole system using the Sobol sequence (Press et al. 1992), which distributes quasi-random points evenly throughout the 7 dimensional ($\widehat{S} \times A \subset \Re^7$) space. For each point $(\widehat{s}^i, a^i)$, $\widehat{s} \in \widehat{S}, a \in A$, we generated the next state $\Phi^*_\delta(\widehat{s}^i, a^i)$ with $\delta = 0.02$ seconds (i.e. one time-step). The training set is then $\{ ((\widehat{s}^i, a^i), \Phi^*_\delta(\widehat{s}^i, a^i)) \}, i = 1..500$. For clarity, hereafter we drop δ from the notation as it remains fixed.

The accuracy of the model was tested periodically during training on a separate test set of 500 examples to determine how well it could generalize to state-action pairs not found in



(a) Local error (b) Prediction error

Figure 9: **Model accuracy (color figure).** Plot (a) shows the trajectory of the long pole angle for the real system (target environment) and model $M_{0.002}$ (simulation environment). A network is controlling the real system and each time step the model is also fed the state and action to produce a one-step prediction. The inset shows that the model is locally very accurate. Plot (b) shows the prediction error for each model. The large errors seen at the beginning of the trial are due to the high velocities of the poles swinging back and forth as the controller gradually stabilizes the system. When velocities are high successive states are further apart and prediction errors are more likely. Models with more error E have greater error in practice.

the training set. We measured accuracy in terms of model error E:

$$E = \frac{1}{N}\sum_{i=1}^{N} \|\Phi^*(\hat{s}^i, a^i) - \Phi(\hat{s}^i, a^i)\|, \tag{8}$$

where $\|\cdot\|$ is the L_1-norm. This is the average error in the model's prediction across the entire test set of $N = 500$ samples. The highest accuracy we achieved was $E = 0.002$ using a network with 20 hidden units.

Our model is global in the sense that a single function approximator is used to represent Φ for the entire state space, but it is local in terms of its temporal predictions. If the model is used to make predictions one time-step into the future, then the predictions will be very accurate. That is, if we have two state trajectories, one generated by the system:

$$\hat{s}_1,\ \hat{s}_2,\ \hat{s}_3, \ldots, \qquad \text{where } \hat{s}_{i+1} = \Phi^*(\hat{s}_i, \pi(\hat{s}_i)), \tag{9}$$

and the other by the model using the system states $\hat{s}_i$ to make one-step predictions:

$$\hat{s}_1,\ \Phi(\hat{s}_1, \pi(\hat{s}_1)),\ \Phi(\hat{s}_2, \pi(\hat{s}_2)), \ldots, \tag{10}$$

where π is the same policy in both equation 9 and 10, then the trajectories will be very similar (figure 9a). However, as we shall see below, even these small local errors can prevent successful transfer when the environment is unstable.

In order to study the effect of model error on transfer, we saved the weights of the model at five points during training to obtain a set of models with different levels of error: 0.002, 0.005, 0.008, 0.010, 0.025. Figure 9b shows the one-step prediction error (i.e. $\|\widehat{s}_{t+1} - \Phi(\widehat{s}_t, a_t)\|$) for each of the five models ($M_{0.002}, M_{0.005}, M_{0.008}, M_{0.010}, M_{0.025}$, where M_e is a model with $E = e$) in a trial where the target environment is being controlled successfully. The graph shows that E, the error based on the test set, is a reliable indicator of the relative amount of prediction error that will actually be encountered by controllers during evolution. That is, for two models M_x and M_y, $x > y$ implies that the prediction errors of M_x will generally be greater than those of M_x. It is important that this be true, so that E can be used to rank the models correctly for the experiments that follow.

The next section describes how the set of models is used to evolve robust controllers that can transfer to the target environment despite inevitable local errors. We examine how model inaccuracy affects transfer and use two techniques that use noise to improve it.

6.2 Evolving with the Model

If the simulation environment perfectly replicates the dynamics of the target environment, as it did in the comparisons of section 5, then a model-evolved controller will behave identically in both settings, and successful transfer is guaranteed. Unfortunately, since such an ideal model is unattainable, the relevant questions are: (1) how do inaccuracies in the model affect transfer and (2) how can successful transfer be made more likely given an imperfect model? To answer these questions we evolved controllers at different levels of model error to study the relationship between E and transfer, and to find ways in which transfer can be improved.

The controllers were evolved under three conditions:

No noise. The controllers are evolved as in section 5, except instead of interacting with the analytical system (Appendix A), they interact with the model according to equation 7. This experiment provides a baseline for measuring how well controllers transfer from simulation to the target environment. Each of the five models ($M_{0.002}, M_{0.005}, M_{0.008}, M_{0.010}, M_{0.025}$) was used in 20 simulations for a total of 100 simulations. The network fitness was equal to the number of time steps the poles could be balanced.

Sensor noise. The controllers are evolved as above except that their inputs are perturbed by noise. The agent-environment interaction is defined by

$$s_{t+1} = \Phi(s_t, \pi(s_t + v)), \tag{11}$$

where $v \in \Re^6$ is a random vector with components $v(i)$ distributed uniformly over the interval $[-\alpha, \alpha]$. Sensor noise can be interpreted as perturbations in the physical quantities being measured, imperfect measurement of those quantities, or, more generally, a combination of both. This kind of noise has been shown to produce more general

and robust evolved controllers in the mobile robot domain (Jakobi et al. 1995; Miglino et al. 1995b; Reynolds 1994).

In figure 8b, the box labeled "sensor noise" represents this noise source. Note that because the controller does not have access to the velocities at all in task 2b, only x, θ_1, and θ_2 are distorted by noise. As in the "no noise" case, 20 simulations were run for each of the five models, this time with three sensor noise levels: 5%, 7%, and 10%, for a total of 300 simulations. These noise levels far exceed the sensor error that would be expected from a real mechanical system, and are not intended to model the noise of the target environment. Instead, we use sensor noise to try to encourage robust strategies that will be more likely to transfer.

Trajectory noise. In this case, instead of distorting the controllers' sensory input the noise is applied to the dynamics of the simulation model. The agent-environment interaction is defined by

$$s_{t+1} = \Phi(s_t, \pi(s_t)) + w, \tag{12}$$

where $w \in \Re^6$ is a uniformly distributed random vector with $w(i) \in [-\beta, \beta]$. At each state transition the next state is generated by the model and is then perturbed by noise before it is fed back in for the next iteration. Although, a similar effect could be produced by adding noise to the actuators, $s_{t+1} = \Phi(s_t, \pi(s_t)+w)$, equation 12 ensures that the the trajectory of the model remains stochastic even in the absence of a control signal, $\pi(s) = 0$.

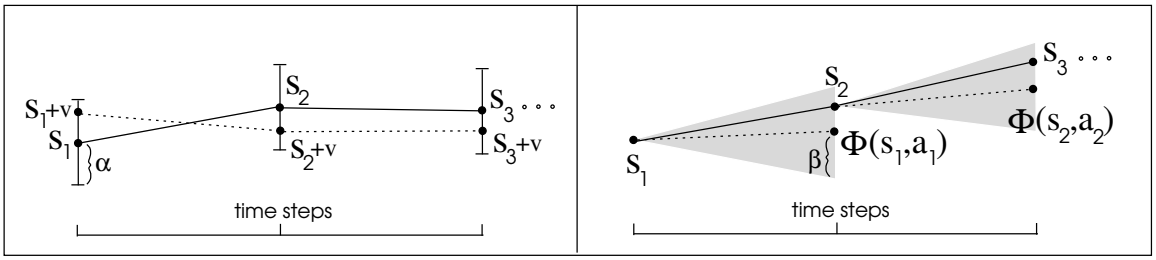
We used 8 different levels of trajectory noise $\{0.5\%, 0.6\%, \ldots, 1.2\%\}$. As with the sensor noise simulations, 20 simulations were run for each of the five models at each trajectory noise level, for a total of 800 simulations.

Figure 10 gives examples of how state transitions occur for the two kinds of noise for a hypothetical 1-dimensional system.

In addition to affecting how networks interact with the simulation environment, noise also affects performance at the evolutionary level by increasing *evaluation noise*. Since a given controller will behave differently from trial to trial due to noisy inputs, its underlying fitness can only be approximated. This noise in the evaluation of networks can mislead ESP by causing it to select solutions that are not truly the best individuals in the population. Therefore, ESP must be able to tolerate evaluation noise in order to search the space of controllers effectively.

6.3 Transfer Results

After evolving controllers with the model, the solution found by each simulation was submitted to a single trial in the target environment. Our criteria for successful transfer was whether a controller could perform the same task in the target environment that it performed



(a) Sensor noise (b) Trajectory noise

Figure 10: **State transitions.** For sensor noise (a), the actual state of the simulation environment s_i is not observable to the controller. Instead, the controller selects an action based on $s_i + v$ (s_i distorted by noise), which transitions the environment to s_{i+1} according to equation 11. The vertical bars at each state represent the range of possible distortions to s_i. The dotted line is the state trajectory that the controller sees, the solid line is the actual state trajectory. For trajectory noise (b), the controller sees the correct state of the environment, but instead of the next state being determined by Φ (dotted line), noise is added to the transition making the dynamics stochastic. The trajectory from s_i to s_{i+1} will lie within the shaded triangle marking the range of possible transitions for a given level of trajectory noise.

in simulation. That is, balance the poles for 100,000 time steps using the same setup (i.e. initial state, pole lengths, etc.) used during evolution, but with no noise. This conforms to the conventional definition of transfer used in many studies (see section 2.5). However, here it constitutes a minimum requirement that a controller must satisfy before the more rigorous examination in the next section.

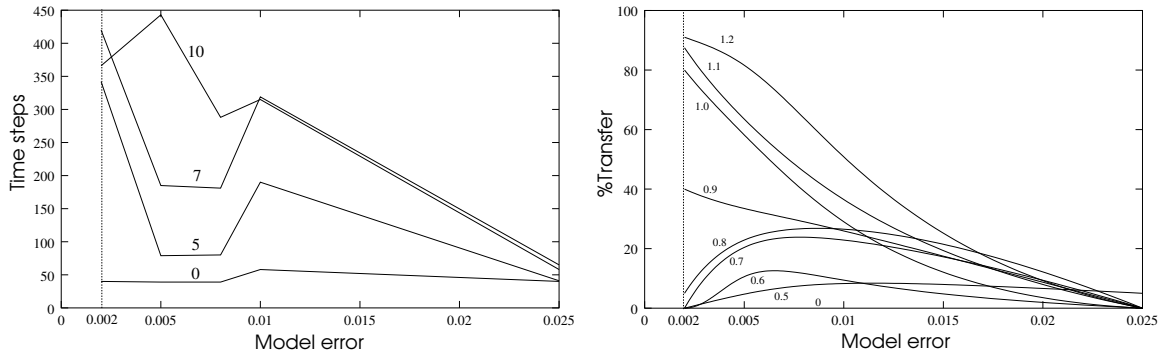
Sensor Noise

Figure 11a shows the transfer results for controllers evolved with sensor noise. The plot shows the average number of time-steps that the networks at each level of model error could control the target environment. Successful transfers were very rare in this case (occurring only twice in all 400 simulations); in effect, the curves show the performance of networks that did not transfer.

Without noise, all of the transferred controllers failed within 50 time steps i.e. almost immediately (curve "0"). As sensor noise was added, performance improved significantly, especially when model error was low, but controllers were still far from being able to stabilize the target environment. Therefore sensor noise, even at high levels, is not useful for transfer in this kind of domain.

Trajectory Noise

On the other hand, trajectory noise had a much more favorable affect on transfer. Because successes were frequent, a different plot is used. Figure 11b shows the percentage of networks that transferred successfully for different levels of model error and trajectory noise. Each



(a) Sensor noise (b) Trajectory noise

Figure 11: **Transfer results.** Plot (a) shows the number of time steps the controllers evolved at different levels of model error and sensor noise could balance the poles after transfer to the target environment. Each curve corresponds to a different level of noise. Sensor noise improves performance slightly but does not produce successful transfers (a transfer was considered successful if the network could control the system for 100,000 time steps). Plot (b) shows the percentage of controllers that successfully transferred to the target environment for different levels of model error and trajectory noise. Each curve is a smoothed average of 20 runs and corresponds to a different percent trajectory noise. Lower error (i.e. more accurate local model) and higher trajectory noise produces controllers more likely to transfer from the model to the real world.

curve corresponds to a different level of trajectory noise, with the "0" curve again indicating transfer without noise. The plot shows that trajectory noise compensates for model error and ensures better transfer. To achieve very reliable transfer, low model error needs to be combined with higher levels of trajectory noise. The best results were achieved with 1.2% trajectory noise and a model error of 0.002, yielding a successful transfer rate of 91%. Moreover, we found that these "untransferred" controllers were "near-misses," and could be adapted to the target environment quite easily through local random search (section 6.5). These results show that transfer is indeed possible despite significant model inaccuracy.

6.4 Evaluating controller robustness

Since during evolution a controller can only be exposed to a subset of the conditions that can exist in the real world, how will it perform under conditions that differ from those encountered during evolution? In this section, we analyze the quality of transferred controllers in 3 respects: (1) generalization to novel starting states, (2) resistance to external disturbances, and (3) resistance to changes in the environment. This analysis goes beyond any other study in testing neuroevolved controllers in realistic situations. Since only the controllers that were evolved with trajectory noise transferred successfully, our analysis pertains only to those controllers. Also, because different levels of trajectory noise had different transfer rates (figure 11b), we evolved additional controllers at each noise level until we had a minimum of 20 successfully transferred controllers for each noise level.

Generalization

In the transfer experiments, the controllers were evaluated from the same starting state in both the simulation and target environments. Therefore, successful transfer gives little insight into how well a controller can stabilize the system from states not visited during evolution. Here we use the generalization test described above (section 5, task 2b) to measure how trajectory noise affects the performance of transferred controllers in a broad range of initial states.

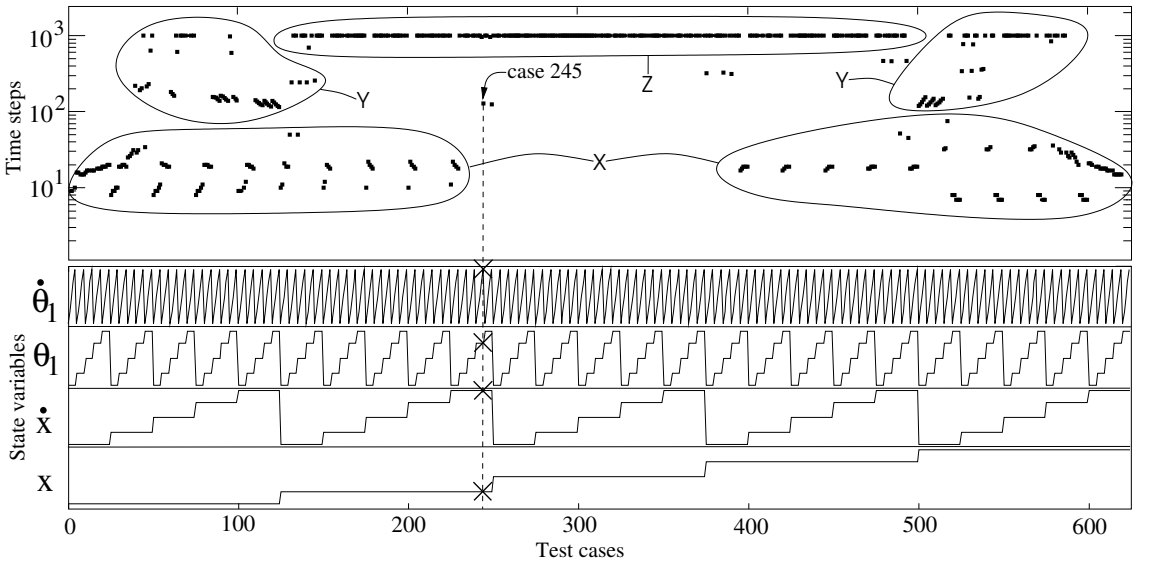
Figure 12a is a visualization of a controller's performance on this test. Each dot in the upper half of the graph identifies the number of time steps the poles could be balanced for a particular start state, i.e. test case. Each test case is denoted by a unique setting of the four state variables $x, \dot{x}, \theta_1, \dot{\theta}_1$ in the lower half of the graph (θ_2 and $\dot{\theta}_2$ were always set to zero). Drawing a vertical line through the graph at a given dot gives the state variable values for that case. For example, the balance time for case 245 ($x = -1.080$, $\dot{x} = 1.215$, $\theta_1 = 0.031$, $\dot{\theta}_1 = 0.135$; the labeled point in the figure) is 129 time steps for this particular controller which solved 353 of the 625 cases. A case was considered solved if the poles where balanced for 1000 time steps.

The graph reveals at least 3 qualitatively different regions of controllability in the state space which we have grouped together and labeled X, Y, and Z. Starting states near the edge of the track (in the extreme left and right of the plot) are difficult to solve because they leave little room to maneuver, especially when the long pole is leaning toward the near edge of the track (area X). When instead the long pole leans toward the center of the track, balancing improves, especially if the cart is not moving in the opposite direction (area Y). States near the center of the track give the controller more space to recover from all settings of $\dot{x}$ and $\dot{\theta}_1$, and also from all settings of θ_1 when in the very center of the track (area Z), and therefore these states are highly successful.

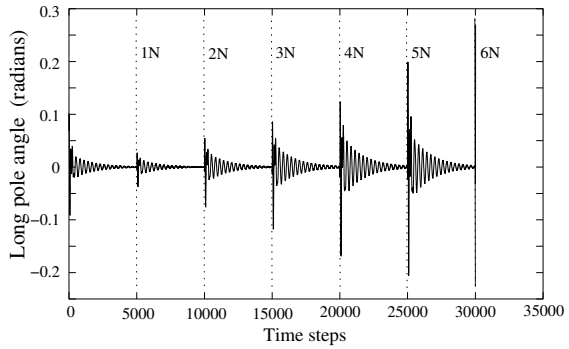
Figure 13a shows the quantitative results for the generalization test. Solutions evolved with more trajectory noise generalize to a larger number of novel initial states. Recall that the fitness function used here simply measures the number of time steps the poles stay balanced. It is therefore almost devoid of domain knowledge, and places no restriction (bias) on the control strategy. Still, the use of trajectory noise produces solutions that generalize nearly as well in the target environment as those tested above in the simulation environment where a much more complex fitness function designed specifically to encourage stable solutions (equations 4 and 5).

External Disturbances

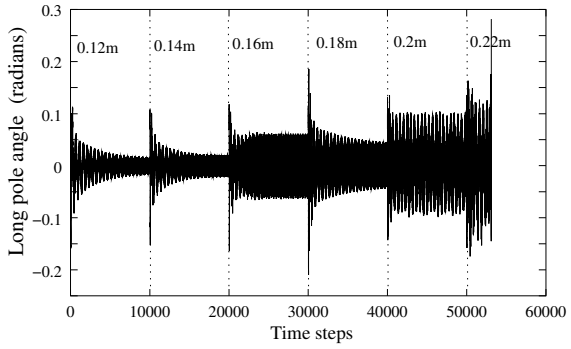
The generalization test measures how well networks behave across a large region of the state space, but how well will these solutions operate in "unprotected" environments where external disturbances, not modeled in simulation, are present? To answer this question, the networks were subjected to external forces that simulate the effect of wind gusts buffeting the cart-pole system. Each network was started in the center of the track with the long pole



(a) Generalization

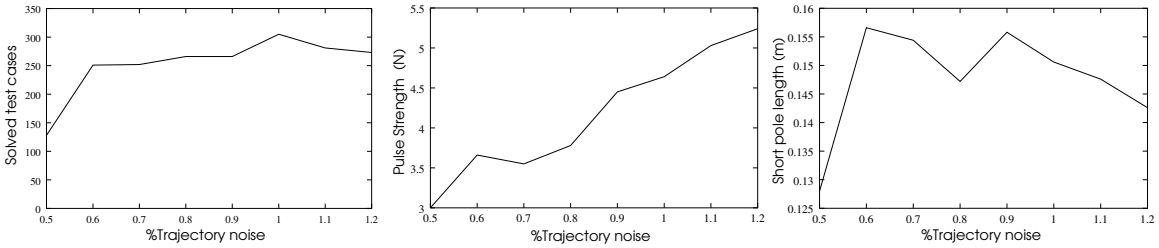


(b) External Disturbances



(c) Changing Environments

Figure 12: **Examples of controller behavior on the robustness tests.** The plots quantify the performance of a particular controller evolved with 1.0% trajectory noise on the three robustness tests. In plot (a) the lower half of the plot shows the values of $x, \dot{x}, \theta_1$, and $\dot{\theta}_1$ for each of the 625 cases. The upper half shows the number of time steps the poles were balanced starting from each case. The cases are divided into qualitatively similar groups labeled X,Y,Z. This controller solved 353 of the cases. Plot (b) shows the trajectory of the long pole angle for a disturbance test. Each vertical dotted line marks the onset of an external pulse of the shown intensity (in Newtons). The controller is able to recover from disturbances of up to 5 Newtons, but fails when the force reaches 6 Newtons. Plot (c) shows the trajectory of the long pole angle for a changing environment test. Each vertical dotted line marks each lengthening of the short pole. The controller is able to balance the poles using increasingly wide oscillations as the short pole is lengthened. However, at 0.22 meters (i.e. almost double the original length) the system becomes unstable.



(a) Generalization (b) External Disturbances (c) Changing Environments

Figure 13: **Robustness results.** The plots show the performance of successfully transferred controllers evolved at different levels of trajectory noise on the three different tests for robustness. Plot (a) shows the number of generalization test cases, on average, solved by the controllers. In plot (b), the y-axis is the average force in Newtons a that network could withstand. In plot (c), the y-axis is the average short pole length that could be balanced for 10,000 time steps. High levels of trajectory noise generally yield more robust behavior in transferred controllers.

at 4.5 degrees (the small pole was vertical). After every 5,000 time steps, a force was applied to the cart for 2 time steps (0.04 sec). The magnitude of this pulse was started at 1 Newton and increased by 1 Newton on each pulse.

Figure 12b shows the angle of the long pole for a typical disturbance test. In the first 5,000 time steps the controller stabilizes the system from its unstable initial state. After the first pulse hits, the system recovers rapidly. As the pulse is strengthened, the controller takes longer to recover, until the force becomes too large causing the system to enter into an unrecoverable state. Figure 13b shows the average maximum force that could be handled for each level of trajectory noise. Above 1.1% noise controllers could withstand an average pulse of over 5 Newtons. This magnitude of disturbance is very large as it is equal to more than half of the maximum force that can be exerted by the controller. Higher levels of trajectory noise lead to networks that are less affected by external forces.

Changes to the Environment

The two previous tests present novel conditions (initial states, external forces) that take place roughly within the same environmental dynamics found during evolution. But what if the dynamics themselves change significantly, as they could in the real world due to mechanical wear, damage, adjustments, and modifications? Evolved controllers that adapt specifically to the narrow conditions present during evolution are likely to fail in environments that do not conform exactly to the simulation model or are non-stationary.

An interesting and convenient aspect of the double pole system is that it is more difficult to control as the poles assume similar lengths. By lengthening the short pole during testing, we can test how well the controllers can cope with a change to the dynamics of the environment. For this test, each network was started controlling the system with a short pole length of 0.12 meters, 0.02 meters longer than the length used during evolution. If after 10,000 time steps the trial had not ended in failure, the cart-pole system was restarted

from the same initial state but with the short pole elongated by another 0.02 meters. This cycle was repeated until the network failed to control the system for 10,000 time steps. A network's score was the short pole length that it could successfully control for 10,000 time steps. Figure 12c shows the behavior of the long pole during one of these tests.

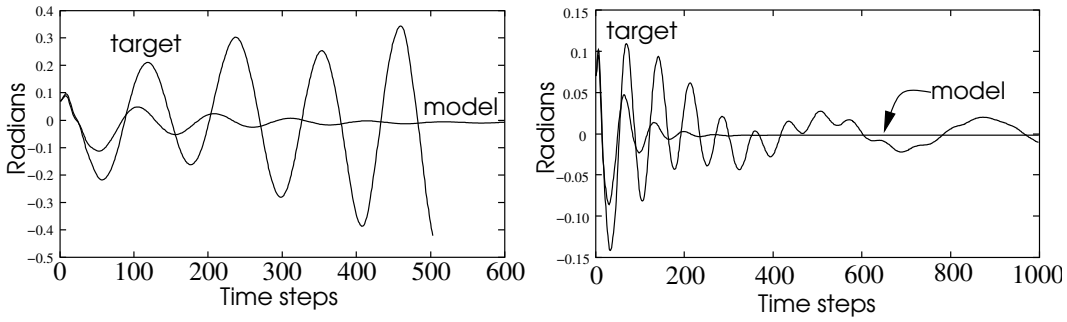
Figure 13c shows the average short pole length versus trajectory noise. For low noise levels ($\approx$0.5%), the networks adapt only to relatively small changes (0.03m, or 30%). At and above 0.6%, networks tolerate up to as much as a 57% increase (to 0.157m) on average. These are very large changes especially because not only do the dynamics change, but also the system becomes harder and harder to control with each extension of the short pole. Trajectory noise prepares controllers for conditions that can deviate significantly from those to which they were exposed during evolution, and, consequently, produces high-performance solutions that can better endure the rigors of operating in the real world.

6.5 Analysis of transfer results

Whenever a behavior is learned in one environment some adaptation is necessary to preserve an acceptable level of performance in another, different environment. The idea of using noise is, in a sense, pre-adapting the agent to a range of possible deviations from the simulation model. It is not surprising that controllers evolved without noise did not transfer, even when model error was low. But why is there such a disparity between the sensor and trajectory noise results?

Let us take a typical solution evolved with sensor noise and use it to control the simulation environment *without* sensor noise. The resulting behavior is shown by the "model" curve in figure 14a. Sensor noise forces ESP to select against strategies that balance the poles by swinging them back and forth. Such strategies are too precarious when the state of the environment is uncertain since they cause the system to periodically traverse states with high velocities where mistakes can more easily lead to failure. Therefore, high-performance solutions quickly stabilize the environment by dampening oscillations. This behavior, however, does not help networks control the target environment. Because the target environment reacts differently from the model, a policy that would stabilize the simulation environment can soon cause the target environment to diverge and become unstable (the "target" curve in figure 14a).

This result differs from the experience of many researchers (e.g. Miglino et al. 1995b; Reynolds 1994) that have used sensor noise to make robots more robust and facilitate transfer. We believe that sensor noise was not effective because of the difficulty of the non-Markov two pole task. Unlike the relatively simple robot navigation tasks reviewed in section 2.5, where small inaccuracies in actuator values do not affect the transferred behavior qualitatively, the pole balancing domain requires very precise control and is much less forgiving due to its inherent instability: a sequence of poor or even imprecise actions can quickly lead to failure. Also sensor noise is much less of an issue here compared to the robot domain where sensor error is notoriously problematic. The sensor error in a real cart-pole system would be



(a) Sensor noise (b) Trajectory noise

Figure 14: **Comparison of controller behavior before and after transfer.** The graphs show typical behavior (long pole angle) of a network evolved with sensor noise (a) versus a network evolved with trajectory noise (b), when controlling either the model or the target envirenment. With sensor noise (a), the trajectories coincide initially, but after about 50 time steps the target environment becomes unstable, although the model is quickly damped into equilibrium. With trajectory noise (b), the trajectories also do not coincide after about 50 time steps but the network is still able to control the target environment because it has been evolved to cope with a range of deviations from the model trajectory.

negligible using readily available linear and rotary position encoders.

The success of controllers evolved with trajectory noise can be explained by looking at the space of state trajectories that are possible by making noisy state transitions. If we let $\{u_i\}$ and $\{l_i\}$ be the sequence of states that form the upper and lower bound on the possible state trajectories for a given policy π, such that

$$u_{i+1} = \max_{s_i \in [l_i, u_i]} \|\Phi(s_i, \pi(s_i)) + \overline{w}\|, \tag{13}$$

$$l_{i+1} = \min_{s_i \in [l_i, u_i]} \|\Phi(s_i, \pi(s_i)) - \overline{w}\|, \tag{14}$$

where

$$\overline{w} = \operatorname*{argmax}_{w} \|w\|, \quad s_1 = l_1 = u_1,$$

then it can be shown, by the continuity of Φ and π, that every state $\|l_i\| \leq \|s\| \leq \|u_i\|$ can be visited by some sequence of state transitions generated by equation 12. State sequences $\{u_i\}$ and $\{l_i\}$ form a trajectory envelope for a particular controller.

All of the trajectories that can be followed from an initial state s_1 will fall inside this envelope (figure 15). Although each network evaluation involves only a single trajectory of finite length, the number of state transitions in a successful trial (100,000) is large enough that it represents a relatively dense sampling of the trajectory space. So the controller is effectively trained to respond to a range of possible transitions at each state (figure 14b). The more noise, the larger the envelope, and the more likely it is that the controller is prepared for differences between the simulation and target environments that could otherwise lead to instability.

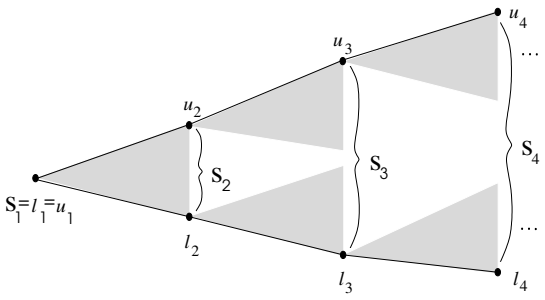
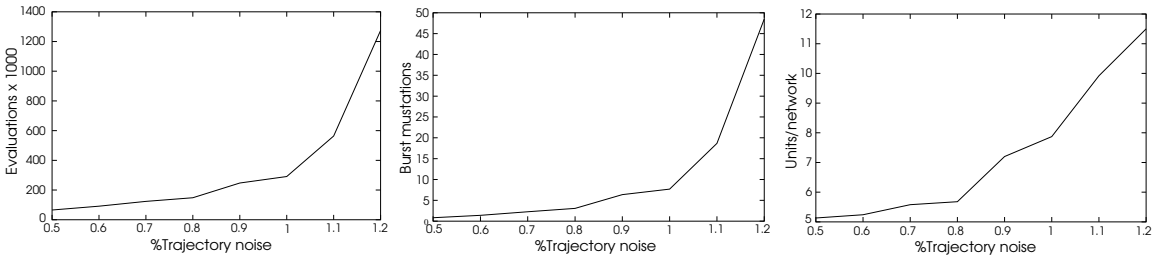


Figure 15: **Trajectory envelope.** Trajectories $\{l_i\}$ and $\{u_i\}$ are the upper and lower bounds on the possible trajectories that the model can take for policy π and a given amount of trajectory noise. The shaded areas show the possible state transitions from l_i and u_i due to noise. The sets S_i are the sets of states $\|l_i\| \leq \|s\| \leq \|u_i\|$. Every state in S_i is reachable from S_1 by some trajectory generated using equation 12. A controller that successfully completes a trial will have sampled a large number of transitions within the envelope and will be more likely to transfer to the target environment.

We should note that no level of trajectory noise can guarantee transfer, and if noise is increased too much the state transitions will become so noisy that the task cannot be solved. Plotting how the performance of ESP scales with trajectory noise (figure 16) we see that an increase in noise incurs a sharply increasing computational cost, but does not yield a proportional improvement in transfer rate. The figure shows the average number of evaluations, burst mutations, and network size required by ESP to solve the task at each noise level. As noise increases, all three grow quadratically. For this reason, we bounded trajectory noise at 1.2% in our experiments—more noise, even just 0.1% more, would yield a marginal increase in transfer rate, but would require a projected 3 million evaluations per controller.

Similarly, it might seem reasonable to assume that transfer will reach 100% as $E \to 0$, but this is not guaranteed either since E is defined on a training set of finite size (equation 8). Even for $E = 0$, $\|\Phi^*(s,a) - \Phi(s,a)\|$ can be greater than zero for some state s not in the training set. Therefore, neither high trajectory noise nor low model error can guarantee that controllers evolved using the neural network model will transfer. However, as our experiments have shown, a combination of the two should make transfer possible in practice.

Although these experiments have focused on direct transfer, in domains where unsuccessful controllers can be tested, a simple post-transfer search can be used to find a successful controller. If a network fails to transfer but performs relatively well when tested, it is likely that it is close to a good solution in the weight space. For instance, a network that can balance the poles for 10 minutes before failing has a good chance of being in the vicinity of a network that can control the system indefinitely. Therefore, such controllers can potentially be adapted to the target environment by local search. For each unsuccessful network from the transfer experiments we perturbed the weights with Cauchy noise to produce a new network that was tested in the target environment. If the new network could not stabilize the system, then the original network was perturbed again until a successful network was found. Using this simple procedure, all of the controllers evolved with trajectory noise over 0.9% and $E \leq 0.005$ could control the target environment after only 68 evaluations on av-



(a) (b) (c)

Figure 16: **Learning performance of ESP with increasing trajectory noise.** Plot (a) shows the number of evaluations needed to solve the task at each trajectory noise level. Plot (b) shows the number of burst mutations required to solve the task. Plot (c) displays the size of the solution network. Each data point is the average of 20 simulations. All three grow quadratically with trajectory noise. Noise levels above 1.2% will incur a very high cost without significantly improving transfer.

erage. This means that evolving with high trajectory noise and low model error produces "better failures" that can be transferred by just making a few small random adjustments to the controller. Given the small number of direct evaluations required to find a successful controller, it may be possible to apply this technique with real systems in a manner similar to that of refining mobile robot controllers by evolving for a few generations in real world (Miglino et al. 1995a; Nolfi et al. 1994).

In domains where testing is not feasible, it may be possible to determine the stability of the controller through analytical tools such as those developed for robust neurocontrol by Suykens et al. (1993) and Kretchmar (2000). Then, only controllers that pass a stability test would be allowed to interact with the environment, thereby reducing the chance of failure.

7 Discussion and Future Work

The complexity of real world systems often defies mathematical analysis, and, most interesting tasks in these environments are too hard for designing a controller strategy by hand. Both of these problems can be avoided by learning from direct interaction given two essential components: a simulator that behaves like the environment, and a learning mechanism that is powerful enough to solve the task. However, in practice simulators are only approximations of the real world so that unexpected behavior can occur when the controller is transferred. The transfer methodology we presented in this paper is a blueprint for efficiently developing controllers in simulation that will work in the real world. By learning a neural network model of the environment we can simulate high-dimensional, non-linear systems with a high degree of local accuracy. By evolving recurrent neural networks with ESP, we can solve difficult tasks efficiently in the simulation environment. And by using trajectory noise in the

model we can evolve controllers that are more insensitive to model inaccuracy and therefore transfer well to the real world.

The comparisons of ESP with other methods show that it is an efficient method for developing robust controllers. ESP was significantly faster and more powerful than the other methods we studied across the entire suite of tasks; it solved even the most difficult tasks in which several of the methods could make little or no progress. The most challenging of the tasks exhibit many of the dimensions of difficulty found in real-world control problems:

1. high-dimensionality,
2. continuous state and action spaces,
3. partial observability (non-Markov),
4. non-linearity,
5. stochasticity.

The first three are problematic for conventional reinforcement learning methods because they either complicate the representation of the value function or the access to it. Neuroevolution deals with them by evolving recurrent networks; the networks can compactly represent arbitrary temporal, non-linear mappings. The success of ESP on tasks of this complexity suggests that it can be applied to the control of real systems that manifest similar properties—specifically, high-dimensional, non-linear, continuous systems such as aircraft control, satellite detumbling, and robot bipedal walking. Other types of environments that are discrete or discontinuous, such as game-playing, job-shop scheduling, and resource allocation may be better served by other learning or optimization strategies.

Our experiments in simulating transfer revealed that a relatively accurate model alone may not be sufficient to ensure transfer when the target environment is inherently unstable. Transforming the deterministic model into a stochastic one by injecting trajectory noise, however, improved the transferred controllers significantly. They were able to cope with a wide variety of conditions that where not present during evolution. These experiments demonstrate how such transfer can be achieved in principle. For a large class of problems involving unstable systems this technique should prove useful.

As a corollary, we should note that the performance of ESP scaled well with evaluation noise. For high levels of both sensor and trajectory noise, ESP solved the non-Markov two-pole problem in a small number of evaluations. This is an important finding since most real environments are stochastic. ESP needs to be able to make progress when fitness measurements are only sample points of an individual's true competence.

The pole balancing problem studied in this paper is a good surrogate for challenging problems in the real world. In ongoing work, we are applying ESP to the active guidance of sounding rockets, a problem with dynamics similar to that of the inverted pendulum. These rockets are used for sub-orbital scientific research such as high-G-force testing, meteorology, environmental sampling, and micro-gravity experimentation. We are utilizing a very accurate

aerodynamic model of a finless version of the Interorbital Systems RSX2[2], and a realistic simulation of aircraft dynamics that includes atmospheric modeling. Finless rockets have the potential to reach much higher altitudes than finned versions, but are very unstable. The objective is to evolve a controller that controls engine thrust in order to counteract dynamic instability (caused by wind and the absence of fins) and maximize final altitude. If such a controller can be developed, the techniques described in this paper will be used to transfer it to the actual rocket launch. Similar applications should be possible in other domains, leading to more accurate and robust non-linear control in the real world.

8 Conclusion

Controlling real-world dynamical systems is difficult. Such environments are highly non-linear and effective control strategies are often not known in advance. Conventional reinforcement learning methods provide a means to *discover* such strategies by interacting with the environment, but scale poorly to large state-spaces and non-Markov environments. In this paper, we have shown that ESP can solve these problems by evolving recurrent neural networks, and do so more efficiently than other methods. For ESP to succeed as a general method for reinforcement learning tasks, the controllers evolved in simulation must transfer successfully to the real world. Our transfer experiments demonstrate that combining an empirical model with trajectory noise allows crossing the reality gap even in unstable environments. The approach therefore provides a powerful platform for applying evolutionary methods to challenging real world control problems.

[2] www.interorbital.com/Sounding_Rockets.htm

Appendix A: Pole-balancing equations

The equations of motion for N unjointed poles balanced on a single cart are

$$\ddot{x} = \frac{F - \mu_c sgn(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i},$$

$$\ddot{\theta}_i = -\frac{3}{4l_i}(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i l_i}),$$

where $\tilde{F}_i$ is the effective force from the ith pole on the cart,

$$\tilde{F}_i = m_i l_i \dot{\theta}_i^2 \sin\theta_i + \frac{3}{4} m_i \cos\theta_i (\frac{\mu_{pi}\dot{\theta}_i}{m_i l_i} + g\sin\theta_i),$$

and $\tilde{m}_i$ is the effective mass of the ith pole,

$$\tilde{m}_i = m_i(1 - \frac{3}{4}\cos^2\theta_i).$$

Parameters used for the single pole problem:

Sym.	Description	Value
x	Position of cart on track	[-2.4,2.4] m
θ	Angle of pole from vertical	[-12,12] deg.
F	Force applied to cart	-10,10 N
l	Half length of pole	0.5m
M	Mass of cart	1.0 kg
m	Mass of pole	0.1 kg

Parameters for the double pole problem.

Sym.	Description	Value
x	Position of cart on track	[-2.4,2.4] m
θ	Angle of pole from vertical	[-36,36] deg.
F	Force applied to cart	[-10,10] N
l_i	Half length of ith pole	$l_1 = 0.5$m $l_2 = 0.05$m
M	Mass of cart	1.0 kg
m_i	Mass of ith pole	$m_1 = 0.1$ kg $m_2 = 0.01$ kg
μ_c	Coefficient of friction of cart on track	0.0005
μ_p	Coefficient of friction if ith pole's hinge	0.000002

Appendix B: Parameter settings used in pole balancing comparisons

Below are the parameters used to obtain the results for Q-MLP, SARSA-CABA, SARSA-CMAC, CNE, SANE, and ESP in section 5. The parameters for VAPS, EP, and CE along with a detailed description of each method can be found in the papers from which their results were taken: VAPS (Meuleau et al. 1999), EP (Saravanan and Fogel 1995), CE (Gruau et al. 1996a).

Table B.1 describes the parameters common to all of the value function methods.

Parameter	Description
ϵ	greediness of policy
α	learning rate
γ	discount rate
λ	eligibility

Table B.1. All parameters have a range of (0,1).

Q-MLP

Parameter	Task		
	1a	1b	2a
ϵ	0.1	0.1	0.05
α	0.4	0.4	0.2
γ	0.9	0.9	0.9
λ	0	0	0

For all Q-MLP experiments the Q-function network had 10 hidden units and the action space was quantized into 26 possible actions: $\pm 0.1, 0.25, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

SARSA-CABA

Parameter	Task	
	1a	1b
τ_d	0.03	0.03
τ_k^x	0.05	0.05
τ_k^u	0.1	0.1
ϵ	0.05	0.05
α	0.4	0.1
γ	0.99	0.99
λ	0.4	0.4

τ_d is the density threshold, τ_k^x and τ_k^u are the smoothing parameters for the input and output spaces, respectively. See Santamaria et al. (1998) for a more detailed description of the Case-Based Memory architecture.

SARSA-CMAC

Parameter	Task	
	1a	1b
ϵ	0.05	0.05
α	0.4	0.1
γ	0.9	0.9
λ	0.5	0.3
No. of tilings	45: 10 based on $x, \dot{x}, \theta_1$ 5 based on x, θ 5 based on $x, \dot{\theta}$ 5 based on $\dot{x}, \dot{\theta}$ 5 based on x 5 based on $\dot{x}$ 5 based on θ 5 based on $\dot{\theta}$	50 : 10 based on x_t, x_{t-1}, θ_t 10 based on $x, \theta_t, \theta_{t-1}$ 5 based on x_t, θ_t 5 based on x_{t-1}, θ_{t-1} 5 based on x_t 5 based on x_{t-1} 5 based on θ_t 5 based on θ_{t-1}

where x_t and θ_t are the cart position and pole angle at time t. Each variable was divided in to 10 intervals in each tiling. For a more complete explanation of the CMAC architecture see Santamaria et al. (1998).

SANE

Parameter	Task	
	1a	2a
no. of neurons	100	200
no. of blueprints	50	100
evals per generation	200	400
size of networks	5	7

The mutation rate for all runs was set to 10%.

CNE

Parameter	Task			
	1a	1b	2a	2b
no. of networks	200	200	400	1000
size of networks	5	5	5	rand [1..9]
burst threshold	10	10	10	15

The mutation rate for all runs was set to 20%. Burst threshold is the number of generations after which burst mutation is activated if the best network found so far is not improved upon. CNE evaluates each of the networks in its population once per generation.

ESP

Parameter	Task			
	1a	1b	2a	2b
initial no. of subpops	5	5	5	rand [1..9]
size of subpopulations	20	20	40	100
evals per generation	200	200	400	1000
burst threshold	10	10	10	15

The mutation rate for all runs was set to 40%. Burst threshold is the number of generations after which burst mutation is activated if the best network found so far is not improved upon.

References

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*, 97(3):220–227.

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

Baird, L. C., and Moore, A. W. (1999). Gradient descent reinforcement learning. In *Advances in Neural Information Processing Systems 12.*

Barto, A. G. (1990). Connectionist learning for control. In 3rd, W. T. M., Sutton, R. S., and Werbos, P. J., editors, *Neural Networks for Control*, chapter 1, 5–58. Cambridge, MA: MIT Press.

Belew, R. K., McInerney, J., and Schraudolph, N. N. (1991). Evolving networks: Using the genetic algorithm with connectionist learning. In (Langton et al. 1991).

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bertsekas, D. P., and Tsitsiklis, J. N. (1996). *Neuro-dynamic programming.* Number 3 in the Optimization and Neural Computation Series. Belmont, Mass: Athena Scientific.

Brooks, R. A., and Maes, P., editors (1994). *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life IV).* Cambridge, MA: MIT Press.

Chavas, J., Corne, C., Horvai, P., Kodjabachian, J., and Meyer, J.-A. (1998). Incremental evolution of neural controllers for robust obstacle-avoidance in khepera. In *EvoRobots*, 227–247.

Crites, R. H., and Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In (Touretzky et al. 1996), 1017–1023.

Darwen, P. J. (1996). *Co-Evolutionary Learning by Automatic Modularization with Speciation.* PhD thesis, University College, University of South Wales.

Dominic, S., Das, R., Whitley, D., and Anderson, C. (1991). Genetic reinforcement learning for neural networks. In (IJCNN 1991), 71–76.

Dracopoulos, D. C. (1997). *Evolutionary Learning Algorithms for Neural Adaptive Control.* Perspectives in neural computing. Springer.

Eriksson, R., and Olsson, B. (1997). Cooperative coevolution in inventory control optimization. In *Proceedings of 3rd International Conference on Artificial Neural Networks and Genetic Algorithms*.

Ficici, S. G., Watson, R. A., and Pollack, J. B. (1999). Embodied evolution: A response to challenges in evolutionary robotics. In Wyatt, J. L., and Demiris, J., editors, *Eighth European Workshop on Learning Robots*, 14–22.

Floreano, D., and Mondada, F. (1996). Evolution of homing navigation in a real mobile robot. In *IEEE transactions on systems, man, and cybernetics: part B; cybernetics*, vol. 26, 396–407. IEEE.

Gomez, F., and Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342.

Gomez, F., and Miikkulainen, R. (1998). 2-D pole-balancing with recurrent evolutionary networks. In *Proceedings of the International Conference on Artificial Neural Networks*, 425–430. Berlin; New York: Springer-Verlag.

Gomez, F., and Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*. Denver, CO: Morgan Kaufmann.

Grady, D. (1993). The vision thing: Mainly in the brain. *Discover*, 14:57–66.

Gruau, F., Whitley, D., and Pyeatt, L. (1996a). A comparison between cellular encoding and direct encoding for genetic neural networks. Technical Report NC-TR-96-048, NeuroCOLT.

Gruau, F., Whitley, D., and Pyeatt, L. (1996b). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, 81–89. Cambridge, MA: MIT Press.

Harnad, S. (1990). The symbol grounding problem. *Physica D*, 42:335–346.

Harp, S. A., Samad, T., and Guha, A. (1989). Towards the genetic synthesis of neural networks. In *Proceedings of the Third International Conference on Genetic Algorithms*, 360–369.

Holland, J. H., and Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In Waterman, D. A., and Hayes-Roth, F., editors, *Pattern-Directed Inference Systems*. New York: Academic Press.

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

Howard, R. A. (1960). *Dynamic Programming and Markov Processes.* Cambridge, MA: MIT Press.

IJCNN (1991). *Proceedings of the International Joint Conference on Neural Networks* (Seattle, WA). Piscataway, NJ: IEEE.

Jakobi, N. (1993). Half-baked, ad-hoc, and noisy: Minimal simulations for evolutionary robotics. In Husbands, P., and Harvey, I., editors, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life IV)*, 348–357. Morgan Kaufmann.

Jakobi, N. (1998). *Minimal Simulations for Evolutionary Robotics.* PhD thesis, University of Sussex.

Jakobi, N., Husbands, P., and Harvey, I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. In *Proceedings of the Third European Conference on Artificial Life.* Springer-Verlag.

Jefferson, D., Collins, R., Cooper, C., Dyer, M., Flowers, M., Korf, R., Taylor, C., and Wang, A. (1991). Evolution as a theme in artificial life: The Genesys/Tracker system. In (Langton et al. 1991).

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

Koza, J. R. (1991). *Genetic Programming.* Cambridge, MA: MIT Press.

Kretchmar, R. M. (2000). *A Synthesis of Reinforcement Learning and Robust Control Theory.* PhD thesis, Department of Computer Science, Colorado State University, Fort Collins, Colorado.

Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors (1991). *Proceedings of the Workshop on Artificial Life (ALIFE '90).* Reading, MA: Addison-Wesley.

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

Lin, L.-J. (1993). *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, CMU, Pittsburg.

Lin, L.-J., and Mitchell, T. M. (1992). Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, Carnegie Mellon University, School of Computer Science.

Lund, H. H., and Hallam, J. (1996). Sufficient neurocontrollers can be surprisingly simple. Technical Report Research Paper 824, Department of Artificial Intelligence, University of Edinburgh.

Mahfoud, S. W. (1995). *Niching Methods for Genetic Algorithms.* PhD thesis, University of Illinois at Urbana-Champaign.

Mandischer, M. (1993). Representation and evolution of neural networks. In Albrecht, R., Reeves, C., and Steele, N., editors, *Proceedings of the Conference on Artificial Neural Nets and Genetic Algorithms at Innsbruck, Austria*, 643–649. Springer-Verlag.

Mataric, M., and Cliff, D. (1996). Challenges in evolving controllers for physical robots. *Robotics and Autonomous Systems*, 19(1):67–83.

Meeden, L. (1998). Bridging the gap between robot simulations and reality with improved models of sensor noise. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 824–831. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

Meuleau, N., Peshkin, L., Kim, K.-E., and Kaelbling, L. P. (1999). Learning finite state controllers for partially observable environments. In *15th International Conference of Uncertainty in AI.*

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

Miglino, O., Lund, H. H., and Nolfi, S. (1995a). Evolving mobile robots in simulated and real environments. *Artificial Life*, 2:417–434.

Miglino, O., Lund, H. H., and Nolfi, S. (1995b). Evolving mobile robots in simulated and real environments. Technical report, Institute of Psychology, C.N.R, Rome, Rome, Italy.

Miller, G., and Cliff, D. (1994). Co-evolution of pursuit and evasion i: Biological and game-theoretic foundations. Technical Report CSRP311, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK.

Mondada, F., Franzi, E., and Ienne, P. (1993). Mobile robot miniaturization: A tool for investigation in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics*, 501–513.

Moriarty, D. E. (1997). *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin. Technical Report UT-AI97-257.

Moriarty, D. E., and Miikkulainen, R. (1996a). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32.

Moriarty, D. E., and Miikkulainen, R. (1996b). Evolving obstacle avoidance behavior in a robot arm. Technical Report AI96-243, Department of Computer Sciences, The University of Texas at Austin.

Nolfi, S., Floreano, D., Miglino, O., and Mondada, F. (1994). How to evolve autonomous robots: Different approaches in evolutionary robotics. In (Brooks and Maes 1994), 190–197.

Nolfi, S., and Parisi, D. (1995). Learning to adapt to changing environments in evolving neural networks. Technical Report 95-15, Institute of Psychology, National Research Council, Rome, Italy.

Paredis, J. (1994). Steps towards co-evolutionary classification neural networks. In (Brooks and Maes 1994), 102–108.

Paredis, J. (1995). Coevolutionary computation. *Artificial Life*, 2:355–375.

Pollack, J. B., Blair, A. D., and Land, M. (1996). Coevolution of a backgammon player. In Langton, C. G., and Shimohara, K., editors, *Proceedings of the 5th International Workshop on Artificial Life: Synthesis and Simulation of Living Systems (ALIFE-96)*. Cambridge, MA: MIT Press.

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

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C*. Cambridge University Press. Second edition.

Reynolds, C. W. (1994). Evolution of obstacle avoidance behaviour: using noise to promote robust solutions. In Kenneth E. Kinnear, J., editor, *Advances in Genetic Programming*, chapter 10. MIT Press.

Rosin, C. D. (1997). *Coevolutionary Search Among Adversaries*. PhD thesis, University of California, San Diego, San Diego, CA.

Rummery, G. A., and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR-166, Engineering Department, Cambridge University.

Santamaria, J. C., Sutton, R. S., and Ram, A. (1998). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2):163–218.

Saravanan, N., and Fogel, D. B. (1995). Evolving neural control systems. *IEEE Expert*, 23–27.

Schaffer, J., and Cannon, R. (1966). On the control of unstable mechanical systems. In *Automatic and Remote Control III: Proceedings of the Third Congress of the International Federation of Automatic Control.*

Smith, T. M. C. (1998). Blurred vision: Simulation-reality transfer of a visually guided robot. In *EvoRobots*, 152–164.

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

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In (Touretzky et al. 1996), 1038–1044.

Sutton, R. S., and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

Suykens, J., Moor, B. D., and Vandewalle, J. (1993). Stabilizing neural controllers: a case study for swinging up a double inverted pendulum. In *International Symposium on Nonlinear Theory and its Application (NOLTA'93)*, 411–414.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–277.

Tesauro, G., and Sejnowski, T. J. (1987). A "neural" network that learns to play backgammon. In Anderson, D. Z., editor, *Neural Information Processing Systems.* New York: American Institute of Physics.

Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors (1996). *Advances in Neural Information Processing Systems 8.* Cambridge, MA: MIT Press.

Voigt, H. M., Born, J., and Santibanez-Koref, I. (1993). Evolutionary structuring of artificial neural networks. Technical report, Technical University Berlin, Bio- and Neuroinformatics Research Group.

Vukobratovic, M. (1990). *Biped locomotion : dynamics, stability, control, and applications.* Number 7 in Scientific fundamental of robotics. Springer-Verlag.

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

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

Watson, R. A., Ficici, S. G., and Pollack, J. B. (1999). Embodied evolution: Embodying an evolutionary algorithm in a population of robots. In Angeline, Michalewicz, Schoenauer, Yao, and Zalzala, editors, *Congress on Evolutionary Computation*, 335–342. IEEE.

Whitehead, B. A., and Choate, T. D. (1995). Cooperative–competitive genetic evolution of radial basis function centers and widths for time series prediction. *IEEE Transactions on Neural Networks*.

Whitley, D., Mathias, K., and Fitzhorn, P. (1991). Delta-Coding: An iterative search strategy for genetic algorithms. In Belew, R. K., and Booker, L. B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, 77–84. San Francisco, CA: Morgan Kaufmann.

Wieland, A. (1991). Evolving neural network controllers for unstable systems. In (IJCNN 1991), 667–673.

Yamauchi, B., and Beer, R. D. (1994). Integrating reactive, sequential, and learning behavior using dynamical neural networks. In Cliff, D., Husbands, P., Meyer, J.-A., and Wilson, S. W., editors, *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, 382–391. Cambridge, MA: MIT Press.