Robust Non-linear Control through Neuroevolution

Faustino John Gomez

Report AI-TR-03-303 August 2003

inaki@cs.utexas.edu
http://www.cs.utexas.edu/users/nn

Artificial Intelligence Laboratory
The University of Texas at Austin
Austin, TX 78712

The Dissertation Committee for Faustino John Gomez
certifies that this is the approved version of the following dissertation:

Robust Non-linear Control through Neuroevolution

Committee:

Risto Miikkulainen, Supervisor

Bruce W. Porter

Raymond J. Mooney

Benjamin J. Kuipers

John M. Scalo

Robust Non-linear Control through Neuroevolution

by

Faustino John Gomez, B.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2003

Acknowledgments

I would first like to thank my advisor Risto Miikkulainen without whose encouragement (insistence) this work would not have even begun. Equally crucial, my brother and best friend, Oliver, and my family. Others: my Ph.D. committee, Stephanie Schoonover, Marty and Coquis Mayberry, Jim Bednar, Yoonsuck Choe, Ken Stanley, Bobby Bryant, Gloria Ramirez, Daniel Polani, los Fox-Bakers, Carlos (ostia!), Sonia Barreiro, Marta Lois, quen mais...

I would also like to thank Eric Gullichsen of Interorbital Systems for his collaboration and invaluable assistance in the rocket guidance project. Doug Burger and Hrishikesh Murukkathampoondi for their help in setting up the SimpleScalar parameters, generating the traces, and overall advice in the CMP project.

Lifelong inspiration: Ian Anderson, Martin Barre, Andy Latimer, Albert Camus, Chuck Schuldiner (RIP), Keith Jarrett, Jan Garbarek, Woody Allen, Robert Fripp, Steve Howe, Steve Morse.

This research was supported in part by the National Science Foundation under grants IIS-0083776 and IRI-9504317, and the Texas Higher Education Coordinating Board under grant ARP-003658-476-2001.

FAUSTINO JOHN GOMEZ

The University of Texas at Austin
August 2003

Robust Non-linear Control through Neuroevolution

Publication No. _____

Faustino John Gomez, Ph.D.
The University of Texas at Austin, 2003

Supervisor: Risto Miikkulainen

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 approaches have made progress in such problems, but have so far not scaled well. Neuroevolution, has improved upon conventional reinforcement learning, but has still not been successful in full-scale, non-linear control problems. This dissertation develops a methodology for solving real world control tasks consisting of three components: (1) an efficient neuroevolution algorithm that solves difficult non-linear control tasks by coevolving neurons, (2) an incremental evolution method to scale the algorithm to the most challenging tasks, and (3) a technique for making controllers robust so that they can transfer from simulation to the real world. The method is faster than other approaches on a set of difficult learning benchmarks, and is used in two full-scale control tasks demonstrating its applicability to real world problems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many real world control problems are so complex that designing controllers by conventional means is either impractical or results in poor performance. Take, for example, the problem of flying a rocket on a stable trajectory (figure 1.1). Rockets normally have fins to provide passive guidance and keep them from tumbling, but fins add weight and drag, and the rocket will fly much higher without them. A guidance system will then be required to keep it flying on a straight path (i.e. stabilizing the rocket). However, developing such a guidance system is a difficult and expensive. This dissertation provides a methodology for designing such controllers automatically by evolving neural network controllers using a method called Enforced SubPopulations (ESP).

Figure 1.1: **The RSX-2 Rocket of Interorbital Systems, Inc.** Without fins the RSX-2 would be unstable. Developing an active guidance system for a finless version of the RSX-2 is a challenging problem, and the subject of chapter 9.

1.1 Motivation

For some relatively simple control problems, effective or even optimal controllers can be designed by hand, using classical feedback control theory. The household thermostat is the classic example: provided the temperature outside the home does not change rapidly, a simple linear control law will maintain the temperature inside close to a desired level.

For most interesting real world problems, however, the situation is more complicated because the environment is often highly non-linear. Non-linearity is a problem because modern controller design methods rely on linear mathematical models: first, a linear model of the environment is constructed, then the controller is designed for the model and implemented in the environment. Even if a strategy for solving a particular control task is known in advance (i.e. what action should be taken in each state), the simplifying assumptions required to build a tractable (linear) model can limit the level of performance that can be achieved. If we extend the thermostat example to a more general climate control system, a linear controller will not be able to regulate the temperature adequately. The system consists of many non-linear components (e.g. heating coils, fans, dampers) that interact, and cannot be captured effectively using a linear model.

Non-linear models and controllers such as those based on neural networks can be used to produce systems that can better cope with non-linearity (Miller et al. 1990; Suykens et al. 1996), but a more fundamental problem exists when a satisfactory control strategy is not known. In this case, the control task is not a matter of correctly implementing a known strategy, but rather one of *discovering* a strategy that solves the task. For example, imagine a robot whose mission is to navigate an office space collecting trash as efficiently as possible while avoiding moving obstacles (e.g. people, chairs, and so on) that can impede its progress or cause a collision. Performing one sequence of actions may cause the robot to avoid one obstacle cleanly but hit another soon after. Another sequence of actions may cause the robot to narrowly avoid collision with the first object, but allow it to avoid the second one. In general, it is not possible to predict which action in each situation will benefit the robot the most over the long run.

To solve problems where effective strategies are not easily specified, researchers have explored methods based on reinforcement learning (RL; Sutton and Barto 1998). Instead of trying to pre-program a correct response to every likely situation, the designer only provides a reward or *reinforcement* signal that is correlated with the desired behavior. The controller or *agent* then learns to perform the task by interacting with the environment to maximize the reinforcement it receives. This way the actions that become part of the agent's behavior arise from, and are validated by, how they contribute to improved performance.

In theory, RL methods can solve many problems where examples of correct behavior are not available. They can also be used for tasks where control strategies are known. In such cases, instead of trying to implement a known strategy, RL can be used to optimize a higher level specification of the desired behavior (i.e. a cost function). Since RL places few restrictions on the kind of strategy that can be employed, the learning agent can explore

potentially more efficient and robust strategies that would otherwise not be considered by the designer, or be too complex to design.

Unfortunately, in practice, conventional RL methods 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 learning agents, like natural organisms, are necessarily constrained in their ability to fully perceive their environment.

Recently, significant progress has been made in solving continuous, non-Markov reinforcement learning problems using methods that evolve neural networks or *neuroevolution* (NE; Yamauchi and Beer 1994; Nolfi and Parisi 1995; Yao 1993; Moriarty 1997). Instead of adapting a single agent to solve the task, a population of neural networks is used to search the space of possible controllers according to the principles of natural selection. A successful controller is evolved by allowing each member of the population to attempt the task, and then selecting and reproducing those that perform best with respect to a quantitative measure or *fitness*. NE has been used extensively to evolve simple mobile robot navigation and obstacle avoidance behaviors, but it has not yet scaled to more complex tasks or environments that require high-precision, non-linear control such as the rocket guidance problem.

A critical issue that affects all controllers whether designed, learned, or evolved is robustness. Controllers cannot be developed in actual contact with the system they are meant to control because doing so is usually too inefficient, costly, or dangerous. Therefore, they must be developed in a model or simulator of the environment, and then be *transferred* to the real world. In order to apply neuroevolution to real world problems not only must the method be powerful enough to evolve non-linear controllers in simulation, but also the controllers must be robust enough to transfer to the real world. These are the main issues this dissertation is intended to address.

1.2 Research Goals

The overall aim of this dissertation is to provide a methodology for applying neuroevolution to real-time control tasks. These tasks encompass a broad range of problems in process control, manufacturing, aerospace, and robotics where the controller must continuously monitor the state of the system and execute actions at short, regular intervals to achieve an objective. While neuroevolution is applicable to other types of problems, from classification to more deliberative tasks such as game-playing, my contention is that real-time control problems are those best suited for neuroevolution.

The main advantage of using neuroevolution in these tasks is that it allows the designer to ignore details about the structure of the environment, and about how the task should be solved. Instead of having to build an analytical model of the environment, all that is needed is a simulator that can approximate its behavior, and provide a setting for evaluating controllers. Therefore, the development process is greatly simplified by eliminating the need for a formal analysis of the environment. For systems that are currently controlled by conventional designs, this means that NE can be used to optimize performance by evolving non-linear controllers that do not make *a priori* assumptions about how the task should be solved. The process will be demonstrated in this dissertation using the pole balancing benchmark.

More significantly, NE can be employed to solve tasks for which there are currently no satisfactory solutions, and encourage the exploration of new and more challenging control problems such as the finless rocket. Instead of having to use a strategy based on heuristics or dictated by control theory, the designer only has to supply a scalar fitness measure that quantifies the relative competence of all possible behaviors. This is usually easier to determine than the correct strategy itself.

1.3 Approach

Applying artificial evolution to real world tasks involves two steps: first, a controller is evolved in a simulator, and then the controller is connected to the physical system it is intended to control. I claim that for this process to succeed, three components are essential:

1. An evolutionary algorithm that is capable of efficiently searching the space of controller representations that are powerful enough to solve non-linear tasks.

2. A method for scaling the evolutionary algorithm for tasks that are too difficult to be solved directly.

3. A technique to ensure that controllers are robust enough to transfer.

Each of these components is addressed in this dissertation. First, the Enforced Sub-Populations algorithm is used to automatically design non-linear controllers. ESP is a neuroevolution method that is based on Symbiotic, Adaptive Neuroevolution (SANE; Moriarty 1997). Like SANE, ESP evolves network components or *neurons* instead of complete neural networks. However, instead of using a single population of neurons to form networks, ESP designates a separate subpopulation of neurons for each particular structural location

(i.e. unit) in the network. The subpopulation architecture makes neuron evaluations more consistent so that the neurons specialize more rapidly into useful network sub-functions. An equally important side effect of accelerated specialization is that it allows ESP to evolve recurrent networks. These networks are necessary for tasks that require memory, and cannot be evolved reliably using SANE.

Second, an incremental evolution technique is presented which can be used with any evolutionary algorithm to solve tasks that are too difficult to solve directly. Instead of trying to solve a difficult task "head-on," a solution is first evolved for a much easier related task. In steps, the ultimate goal task is solved by transitioning through a sequence of intermediate tasks of increasing difficulty.

Third, to learn how to perform transfer it must be studied in a controlled experimental setting. The problem of transfer is studied by simulating the process of testing a controller in the real world after it has been evolved in a simulator. I analyze two techniques that use noise to make controllers more robust and prepare them for transfer by compensating for inaccuracies in the simulator. Together, these three components form a methodology that can be used to solve difficult non-linear control tasks.

1.4 Overview of Dissertation

The chapters are grouped into five parts: Introduction and Foundations (**Chapters 1** and **2**), ESP (**Chapter 3**), Comparisons (**Chapters 4**, **5**, and **6**), Applications (**Chapters 7**, **8**, and **9**), and Discussion and Conclusion (**Chapters 10** and **11**).

In **Chapter 2**, I lay the foundation for the three components of the method (ESP, incremental evolution, and transfer) by providing background material on control, reinforcement learning, artificial evolution, shaping, and controller transfer.

Chapter 3 presents the core contribution of the dissertation, ESP. The three chapters that follow (**4**, **5**, and **6**) each demonstrate a component of the method using the pole balancing domain as a testbed. In **Chapter 4**, ESP is compared to a broad range of learning algorithms on a suite of pole balancing tasks that includes difficult non-Markov versions. This comparison represents the most comprehensive evaluation of reinforcement learning methods that has been conducted to date, including both single-agent and evolutionary approaches. ESP is shown to solve harder versions of the problem more efficiently than the other methods.

In **Chapter 5**, incremental evolution is formally introduced and used to push further the most difficult task from chapter 4, showing how gradually raising task difficulty can dramatically improve the efficiency of ESP, and allow it to solve harder tasks than would

otherwise be possible.

The problem of transferring controllers to the real world is analyzed in **Chapter 6**. While transfer has been studied in the mobile robot domain, this research is the first to look at transfer in a high-precision unstable control problem.

In **Chapter 7**, ESP and incremental evolution are applied to an Artificial Life pursuit-evasion task to demonstrate how ESP can be used to solve tasks that require short-term memory. In **Chapter 8**, ESP is applied to the first of two real world applications, the problem of managing the memory cache resources for a chip-multiprocessor. The second application, in **Chapter 9**, is the stabilization of a finless version of the Interorbital Systems RSX-2 rocket mentioned in section 1.1. This is the most challenging task undertaken in this dissertation, representing a scale-up to the full complexity of a real world control problem.

Chapter 10 discusses the contributions and outlines some promising directions for future work, and **Chapter 11** concludes the dissertation.

Appendix A contains the equations of motion for pole balancing domain, **Appendix B** contains the parameter settings used by the various methods in chapter 4, and **Appendix C** contains the equation of motion for the environment used in chapter 7.

Chapter 2

Foundations

This chapter provides the background material and literature review that relates to the three components of the approach outlined above (ESP, incremental evolution, and controller transfer). The first section discusses the basic concept of control and the standard engineering approach to solving common control tasks. The next four sections focus on learning control, starting with general reinforcement learning methods, then focusing on neuroevolution, neuroevolution based on cooperative coevolution, and finally Symbiotic, Adaptive Neuroevolution, the method on which ESP is based. The next section covers shaping, the concept underlying the incremental evolution approach used to scale ESP to difficult tasks. The last section discusses the topic of controller transfer.

2.1 Control

This section serves to define many of the control concepts and terms used throughout the dissertation. Control is a process that involves two components: an *environment* and a *controller* (figure 2.1). At any given instant, the environment is in one of a potentially infinite number of *states* $s \in S \subset \Re^n$, and the dynamics of the environment are governed by some arbitrary, unknown function f,

$$s_{t+1} = f(s_t, a_t), \tag{2.1}$$

of the state s_t and the *action* $a_t \in A \subset \Re^m$ at time t, where S and A are known as the *state-space* and *action-space*, respectively. The action is generated by the controller according to:

$$a_t = \pi(o_t), \;\; o_t = h(s_t), \tag{2.2}$$

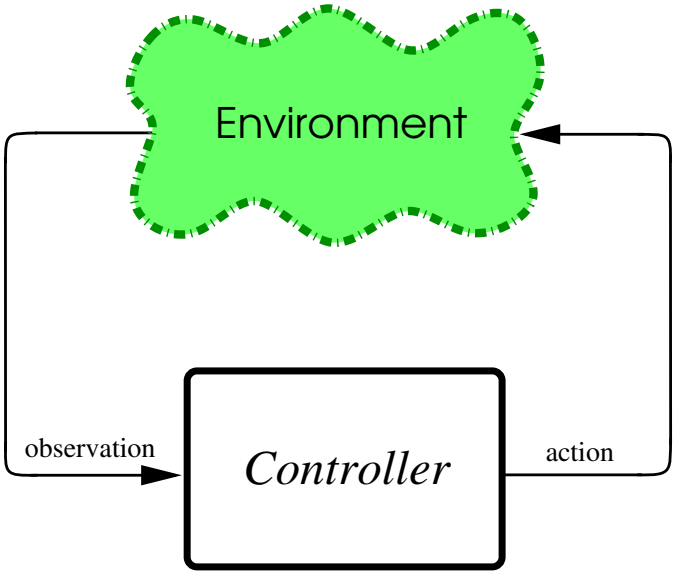


Figure 2.1: **Control.** At regular intervals the controller receives information about the state of the environment (i.e. an observation) and immediately outputs an action that potentially affects the next state of the environment. The controller's objective is to keep the environment on some desirable trajectory.

where π is the control function or *policy*, and o_t is the controller's *observation* of the state. The function h can be thought of as the controller's sensory system; because some of the state variables may not be measurable, and sensors are always noisy and of limited resolution, the observation o_t is generally an imperfect measurement of the state. Note that although real systems are continuous in time, the discrete-time treatment used here is more appropriate because controllers are almost always developed and implemented using digital hardware.

An environment starting in some initial state s_0 and acted upon by some policy, follows a sequence of states or *trajectory* through the state space:

$$s_0, s_1, s_2, \ldots \tag{2.3}$$

Let Ω be the set of all possible trajectories that can occur as a consequence of all possible sequences of actions $\{a_t\}$ starting in s_0. The subset of Ω that is generated by a particular controller will be referred to as the controller's *behavior*. The behavior can contain more than one trajectory because f,π, or h may be stochastic. The objective of control is to find a controller whose behavior solves a particular *task*. In this dissertation, task refers to a pairing of a particular environment and objective. For example, a robot following a wall in

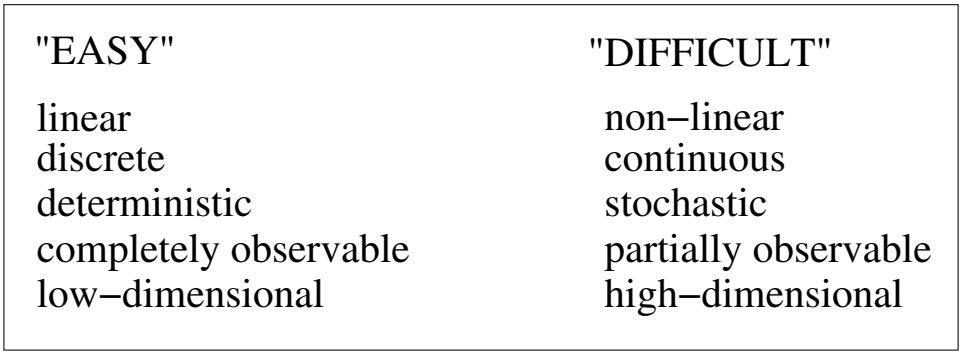


Figure 2.2: **Dimensions of environment complexity.** Environments that exhibit the properties on the left are usually considered easier to control than environments exhibiting the properties on the right. Non-linear environments are usually a problem for classical control theory, and continuous, high-dimensional, and partially observable environments cause problems for conventional reinforcement learning methods. The focus of this dissertation will be primarily on "difficult" environments.

an office space is clearly a different task from the robot collecting trash in the same office space. Likewise, a robot collecting trash outdoors is performing a different task than a robot pursuing the same objective in an office space.

The latter example is less intuitive because we tend to think of proficiency in a task as a general competency. While the ultimate goal in many control problems is to produce controllers with such general behavior, it is often the case that a controller that can accomplish an objective in one environment is not able to in another slightly different environment. This definition is useful for the purpose of discussing both incremental evolution (chapter 5) where a controller is evolved in a sequence of environments with the same objective, and transfer (chapter 6) where a controller evolved in a simulator may not be able to accomplish the same objective in the real world due to inevitable differences between the two environments.

One way to classify control tasks is by the complexity of their environments. Figure 2.2 lists several dimensions that are commonly used to describe environments. The characteristics on the left are considered easy for controller development methods while those on the right are found in most challenging control tasks in the real world. This dissertation will focus primarily on tasks that exhibit "difficult" properties.

Almost all controllers in operation today are designed using methods derived from classical control theory, most commonly linear feedback control (figure 2.3). For these methods to work, a *reference signal* must be available that specifies the desired state of the environment at each point in time. The controller is designed to *track* the reference

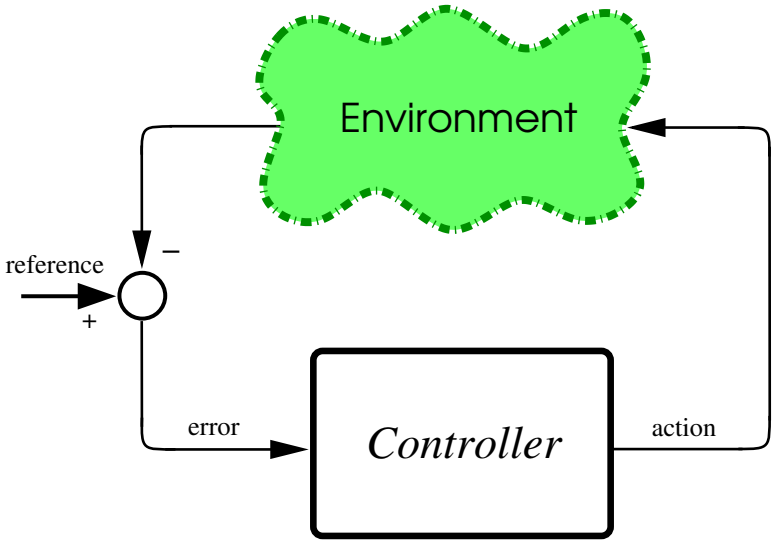


Figure 2.3: **Feedback Control.** The controller outputs actions to try to minimize the error between the state of the environment and an external reference signal that prescribes the desired trajectory of the environment.

signal by minimizing the error between the signal and the actual state of the environment. Feedback control can be very practical and effective provided that the environment is approximately linear and time-invariant (i.e. the dynamics do not change over time). Because real world systems are rarely linear, modern control theory such as Adaptive control and Robust control have focused on extending linear methods to environments that are non-linear.

Although, non-linearity is a key issue in most control tasks, a more fundamental problem exists when there is no reference signal. Instead, there is only a more high-level specification of what the behavior should be. Problems that exhibit this characteristic are known as reinforcement learning problems, and the most common method for solving them is Reinforcement Learning discussed in next section.

2.2 Reinforcement Learning

Reinforcement learning refers to a class of algorithms for solving problems in which a sequence of decisions must be made to maximize a reward or *reinforcement* received from the environment. At each decision point, the learning agent in state $s \in S$, selects an action $a \in A$ that transitions the environment to the next state s' and imparts a reinforcement sig-

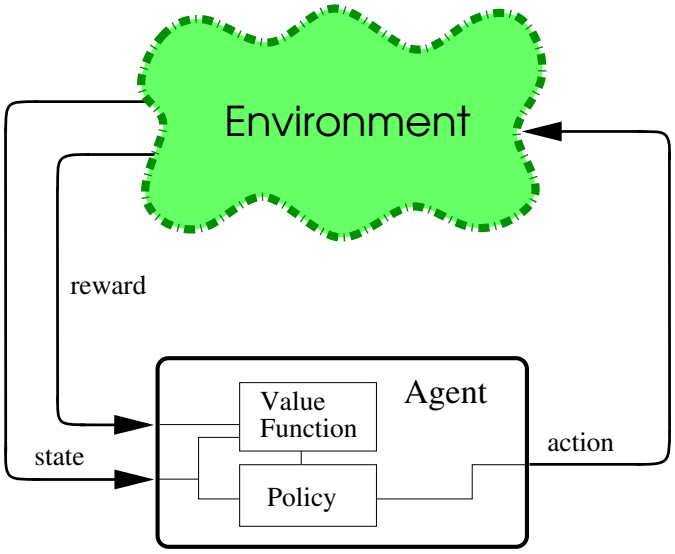


Figure 2.4: **The value function approach.** The agent is composed of a value-function and a policy. The value function tells the agent how much reward can be expected from each state if the best known policy is followed. The policy maps states to actions based on information from the value function.

nal, 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 gradually learning policy that leads to favorable outcomes by associating reward with certain actions in each state it visits. This learning process is difficult because, unlike in 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.

The best understood and widely used learning methods for solving control tasks are based on Dynamic Programming (Howard 1960). These methods dominate this area of research to such an extent that they have become nearly synonymous with the term "reinforcement learning" (RL; Sutton and Barto 1998). Essential to RL methods is the *value function* V (figure 2.4) which maps each problem state to its utility or *value* with respect to the task being learned. 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 selecting at each state the action 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 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 (1) do not require a model of the environment, (2) can learn by direct interaction, (3) are naturally suited to stochastic environments, and (4) 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 $V(s)$ is updated using the value of the successive state $V(s')$:

$$V(s) := V(s) + \alpha[r + \gamma V(s') - V(s)]. \tag{2.4}$$

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 2.4 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.

In early research, these methods were studied in simple environments with 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 (Sutton 1996; Santamaria et al. 1998), to neural networks (Lin 1993; Tesauro and Sejnowski 1987; Crites and Barto 1996).

Despite substantial progress in recent years, 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 because a controller's sensors usually have limited range, resolution, and fidelity, causing *perceptual aliasing* where many observations that require different actions look the same. The next section looks at an approach that is less susceptible to the problems outlined here.

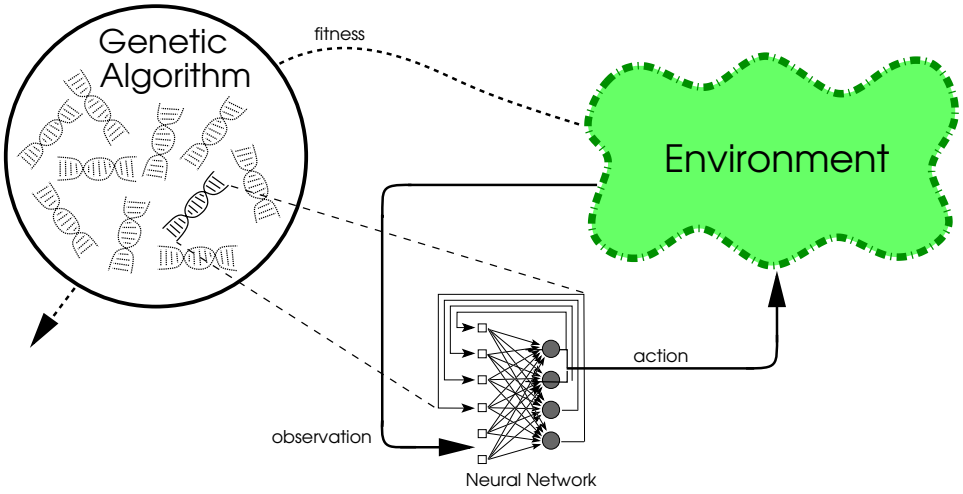


Figure 2.5: **Neuroevolution.** Each chromosome is transformed into a neural network phenotype and evaluated on the task. The agent receives input from the environment (observation) and propagates it through its neural network to compute an output signal (action) that affects the environment. At the 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.

2.3 Neuroevolution

Neuroevolution (NE) presents a fundamentally different approach to reinforcement learning tasks. Neuroevolution leverages the strengths of two biologically inspired areas of artificial intelligence: Artificial Neural Networks and Genetic Algorithms. The basic idea of NE is to search the space of neural network policies directly by using a genetic algorithm (figure 2.5). In contrast to conventional *ontogenetic* learning involving a single agent, evolutionary methods use a population of solutions. These solutions are not modified during evaluation; instead, adaptation arises through the repeated recombination of 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.

2.3.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are parallel distributed processors modeled on biological nervous systems (Haykin 1994). Neural Networks are composed of many simple processing elements or *neurons* that are connected to form a layered structure. A network (figure 2.6) receives information from the environment in the form of a vector $\mathbf{x}$ that activates its input layer. This activation is then propagated to the next layer via weighted

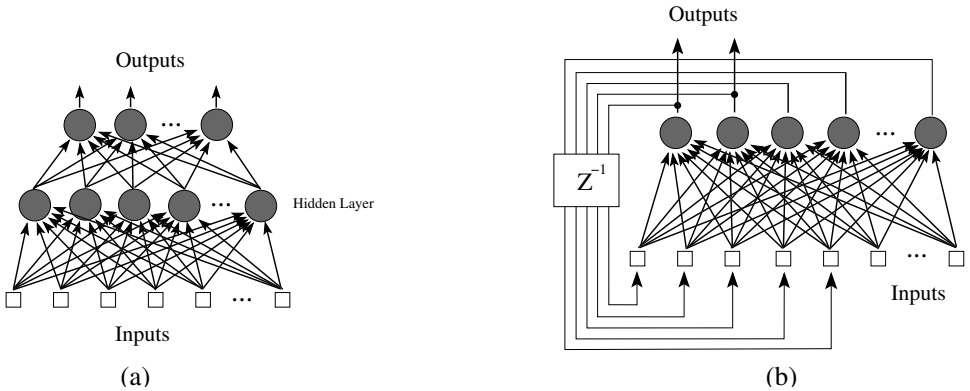


Figure 2.6: **Neural network architectures.** (a) A feedforward network. Information enters the network through the input layer and activation is propagated forward to compute the output. There are no feedback connections. (b) A recurrent network. The network has feedback connections that provide it with information from previous activations.

connections or *synapses*. Each neuron in a non-input layer sums the weighted output from all the units in the previous layer and then applies a non-linear squashing or threshold activation function. For a given input vector $\mathbf{x}$, the output y of each unit j is computed by:

$$y_j = \varphi\left(\sum_i w_{ij} x_i\right) \tag{2.5}$$

Where w_{ij} is the weight from node i to node j, and φ is the sigmoid function $\frac{1}{1+e^{-x}}$. ANNs can be *feedforward* with information flowing only from input to output or the they can be *recurrent* and have feedback connections.

ANNs are a natural choice for representing controllers because of the following properties:

- **Universal function approximation.** Neural Networks are capable of uniformly approximating any differentiable function (Cybenko 1989). This property is useful because it allows neural networks to represent the high-dimensional, non-linear mappings that are required to solve complex control tasks.

- **Generalization.** Neural networks can generalize to novel input patterns. This means that networks can be used as controllers in large state/action spaces without having to expose them to all possible situations.

- **Memory.** Networks with feedback connections (figure 2.6b) can retain information from previous input patterns and allow it to affect the current output. Such recurrent

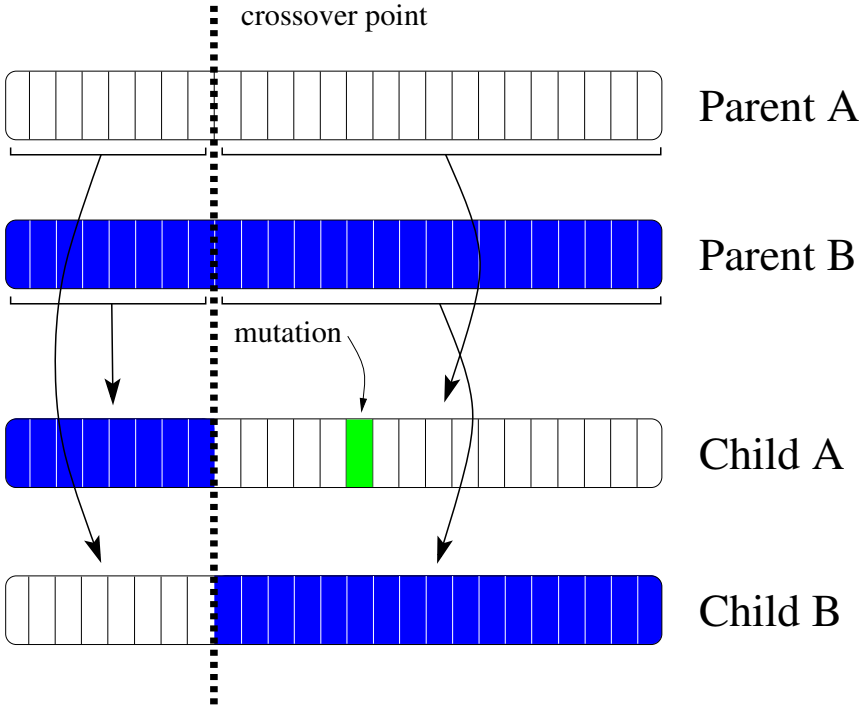


Figure 2.7: **The crossover and mutation operators.** Crossover produces two *children* from two *parents* by cutting the parent chromosomes at a random crossover point and exchanging the segments on either side of the of the cut. The two children are composed of genetic material obtained from both parents. Mutation works by randomly changing one of the alleles in a child's chromosome.

networks can be used to solve non-Markov tasks where the correct output at time t depends not only on the input at time t but also previous input patterns some unknown distance in the past.

Neural networks cannot be designed by hand. Instead, they are trained with gradient-descent algorithms such as backpropagation (Rumelhart et al. 1986) that use examples of correct input/output behavior (i.e. training patterns). Any single-agent method that utilizes neural networks to approximate a value-function or policy must contend with the *stability-plasticity problem* (Carpenter and Grossberg 1987) where learning new patterns can cause previously learned patterns to be forgotten. Also, gradient-descent algorithms are susceptible local minima and notoriously slow and unreliable when used to train recurrent neural networks (Bengio et al. 1994).

2.3.2 Genetic Algorithms

Genetic Algorithms (GAs; Holland 1975; Goldberg 1989) are a class of stochastic search procedures founded on the principles of natural selection. Unlike conventional search methods that iteratively improve a single solution, a GA maintains a set or "population" of *candidate* solutions that sample the search space at multiple points. These solutions are encoded as strings called *chromosomes* that represent the *genotype* of the solution. The chromosomes are usually composed of a fixed number of *genes* that can take on some set of values called *alleles*.

Following a process analogous to natural evolution, each genotype is transformed into its *phenotype* and evaluated on a given problem to assess its *fitness*. Those genotypes with high fitness are then mated using crossover and mutation at low levels to produce new solutions or *offspring*. Figure 2.7 illustrates how crossover and mutation work. Crossover produces two offspring from two parents by exchanging chromosomal substrings on either side of a random crossover point—each offspring is a concatenation of contiguous gene segments from both parents. When an offspring is mutated, one of its alleles is randomly changed to a new value. By mating only the most fit individuals, the hope is that the favorable traits of both parents will be transmitted to the offspring resulting in a higher scoring individual, and eventually leading to a solution.

Because GAs sample many points in the search space simultaneously, they are less susceptible to local minima than single solution methods, and are capable of rapidly locating high payoff regions of high dimensional search spaces. Figure 2.8, shows a hypothetical fitness landscape to illustrate how a GA operates. The fitness of each individual in the population is represented by its position on the landscape. In a single solution method, if the initial search point (the yellow circle) happens to fall in the neighborhood of a local maxima, the algorithm can become trapped because it has only local information with which to make a next guess and improve the solution. Therefore, it will climb the gradient towards the local maxima. In a GA, although some individuals (the red circles) may reside near local maxima, it is less likely to get trapped because the population provides global information about the landscape. There is a better chance that some individual will be near the global maxima, and the genetic operators allow the GA to move the population in large jumps to focus the search in the most fruitful regions of the landscape.

For these reasons, GAs are well suited for searching the space of neural networks. Instead of training a network by performing gradient-descent on an error surface, the GA samples the space of networks and recombines those that perform best on the task in question.

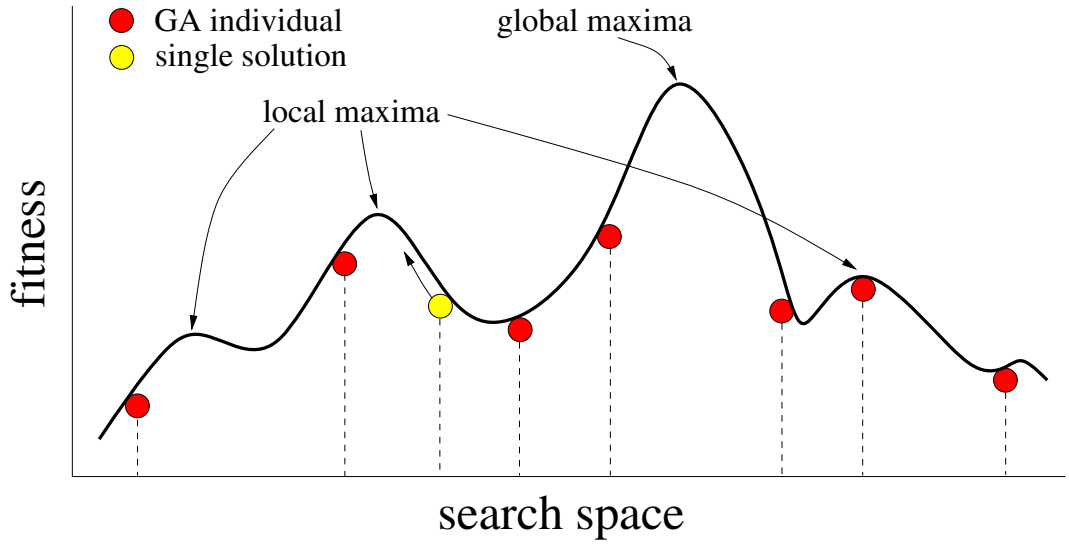


Figure 2.8: **Fitness landscape.** The figure shows a hypothetical search space being searched by both a single solution algorithm and a genetic algorithm. If the single solution algorithm's initial guess (yellow circle) is near a local maxima, it will climb the gradient and get stuck. In contrast, the GA samples many points (red circles), and can therefore identify the region around the global maxima more reliably.

2.3.3 Evolving Neural Networks

NE combines the generalization, function approximation, and temporal capabilities of artificial neural networks with an efficient parallel search method. The GA replaces unreliable learning algorithms and allows adaptation in the absence of targets. By searching the space of policies directly, NE eliminates the need for a value function and its costly computation. Neural network controllers map observations from the environment directly to actions without explicitly assessing their utility.

For NE to work, the environment need not satisfy any particular constraints—it can be continuous and partially observable. All that concerns a NE system is that there be an effective way to evaluate the relative quality of candidate solutions. If the environment contains sufficient regularity for a task to be solvable, and the phenotype representations are sufficiently powerful, then NE can find a solution.

The recurrent neural network offers one such representation that is naturally suited to continuous state/action spaces and tasks that require memory. By evolving these networks instead of training them, NE circumvents the many problems associated with recurrent network learning algorithms.

NE approaches differ from each other primarily by how they encode neural network specifications into strings. I will therefore use this dimension to classify and discuss these methods. In NE, a chromosome can encode any relevant network parameter including synaptic weight values, number of hidden units, connectivity (topology), learning rate, etc. The choice of encoding scheme can play a significant role in shaping 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; Jefferson et al. 1991; Gomez and Miikkulainen 1997) while others such as Symbolic, Adaptive Neuroevolution (SANE; Moriarty 1997) and Neuroevolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen 2002) evolve topology as well.

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.

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. The experiments in chapter 4, compare ESP (a direct fixed topology method) with NEAT, (a direct method that evolves topology), and Cellular Encoding (CE; Gruau et al. 1996a,b, an indirect method that evolves topology) on a difficult non-Markov task.

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.

2.4 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 envi-

ronment. 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; Pollack et al. 1996; Paredis 1994; Miller and Cliff 1994; 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 coevolution 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.

2.4.1 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 2.9). SANE evolves two dif-

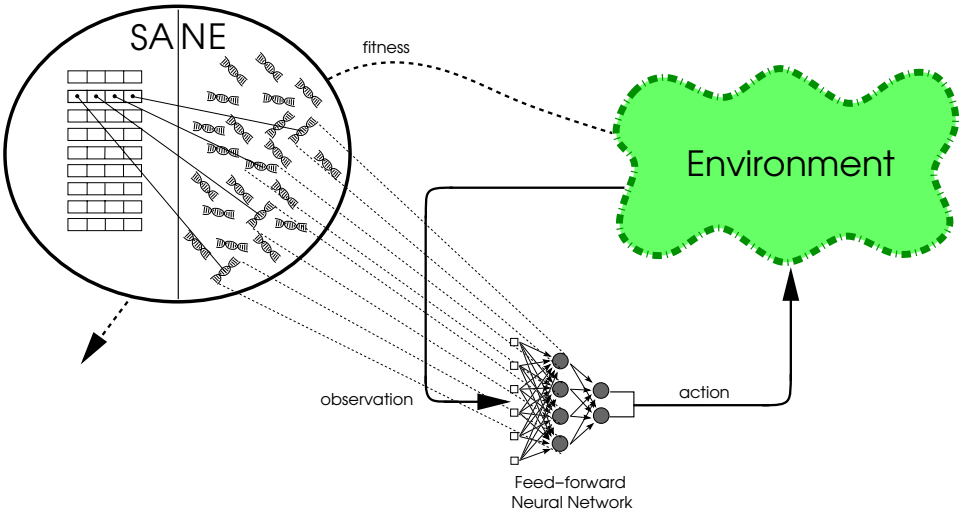


Figure 2.9: **Symbiotic, Adaptive Neuroevolution (color figure).** The algorithm maintains two distinct populations, one of network blueprints (left), and one of neurons (right). Networks are formed by combining neurons according to the blueprints. Networks are evaluated in the task, and the fitness is distributed among all the neurons that participated in the network. After all neurons are evaluated this way, recombination is performed on both populations.

ferent 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, networks are formed according to 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 robot navigation and robot arm control tasks (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 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 types of neurons, such networks cannot perform well; they incur low fitness so that the dominant genotype is selected against, bringing diversity back into the population. In the advanced stages of SANE evolution, instead of converging 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 chapter 3, ESP, extends cooperative neuroevolution to tasks that make use of short-term memory.

2.5 Shaping

There is a general consensus in the artificial learning and robotics communities that in order to scale existing paradigms to significantly more complex tasks, some form of external bias is needed to guide learning (Kaelbling et al. 1996; Perkins and Hayes 1996; Dorigo and Colombetti 1998; Großmann 2001). Rather than trying to solve a difficult problem from scratch with a single monolithic system, it is usually better to exploit any available knowledge to decompose the problem into more accessible units. This may involve breaking the problem down into subproblems, using a learning schedule, incorporating specific domain knowledge, or some combination of the three. The general principal of structuring learning to make tasks more feasible is often referred to as *shaping*.

A number of researchers have used shaping to make learning complex tasks tractable (Colombetti and Dorigo 1992; Jacobs et al. 1991; Perkins and Hayes 1996; Singh 1992).

Typically, in these approaches the complex task is broken into simpler components or *subtasks* that are each learned by separate systems (e.g. GAs or rule-bases) and then combined to achieve the goal task. In incremental evolution, the technique presented in chapter 5, a single system learns a succession of tasks. Such an adaptation process is similar to continual (or lifelong) learning (Elman 1991; Ring 1994; Thrun 1996), and motivated by staged learning in real life where each stage of learning provides a bias or adaptive advantage for learning the next, more advanced, stage. This approach is discussed in more detail and tested in chapter 5, and used effectively to scale ESP in chapters 7 and 9.

2.6 Controller 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, the next subsection reviews the key issues and advances in transfer methods in ER.

2.6.1 Transfer in Evolutionary Robotics

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" (Meeden 1998; Ficici et al. 1999; Jakobi et al. 1995; Lund and Hallam 1996; Floreano and Mondada 1996), avoidance of static obstacles (Jakobi et al. 1995; Miglino et al. 1995a; Chavas et al.

1998), 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 be solved 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 and used it to compute the sensor values and movements of the robot's simulated counterpart. This approach improved the performance of transferred controllers dramatically by ensuring 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, the controller becomes 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; 1995; 1998). 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 be 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 be 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).

2.6.2 Transfer in Unstable Control Tasks

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 only case of successful controller transfer in an unstable domain is, to my knowledge, the work of Bagnell and Schneider (2001). They used a model-based policy search method to learn a hovering behavior for a small scale helicopter. While their results are a very significant achievement, the hovering task was performed in an approximately linear region of the state space where linear (PD) control could be used effectively to stabilize the helicopter. Furthermore, the task that was learned is one that can be solved by conventional engineering approaches (Eck et al. 2001) and supervised neural networks (Buskey et al. 2002).

The transfer experiments in chapter 6 aim at providing a more general understanding of the transfer process including challenging problems in unstable environments. Pole balancing was chosen 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 work represents the first attempt to systematically study transfer outside of the mobile robot domain.

Chapter 3

Enforced Subpopulations

This chapter presents the core contribution of this dissertation, the Enforced SubPopulations[1] (ESP) algorithm. ESP, like SANE, is a neuron-level cooperative coevolution method. That is, the individuals that are evolved are 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 units in the network, and a neuron can only be recombined with members of its own subpopulation (figure 3.2). This way the neurons in each subpopulation can evolve independently, and rapidly specialize into good network sub-functions. This specialization allows ESP to search more efficiently than SANE, and also evolve recurrent networks.

Section 3.1 describes the ESP algorithm in detail, and section 3.2 discusses the advantages of using subpopulations of neurons instead of a single population of neurons (i.e. SANE).

3.1 The ESP Algorithm

ESP can be used to evolve any type of neural network that consists of a single hidden layer, such as feed-forward, simple recurrent (Elman), fully recurrent, and second-order networks. The networks are fully-connected, i.e. every unit in a layer is connected to every unit in the next layer. The neuron chromosomes consist of a string of real numbers that represent the synaptic weights. Figure 3.1 illustrates the correspondence between the values in the genotype and the weights in the phenotype. Evolution in ESP proceeds as

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

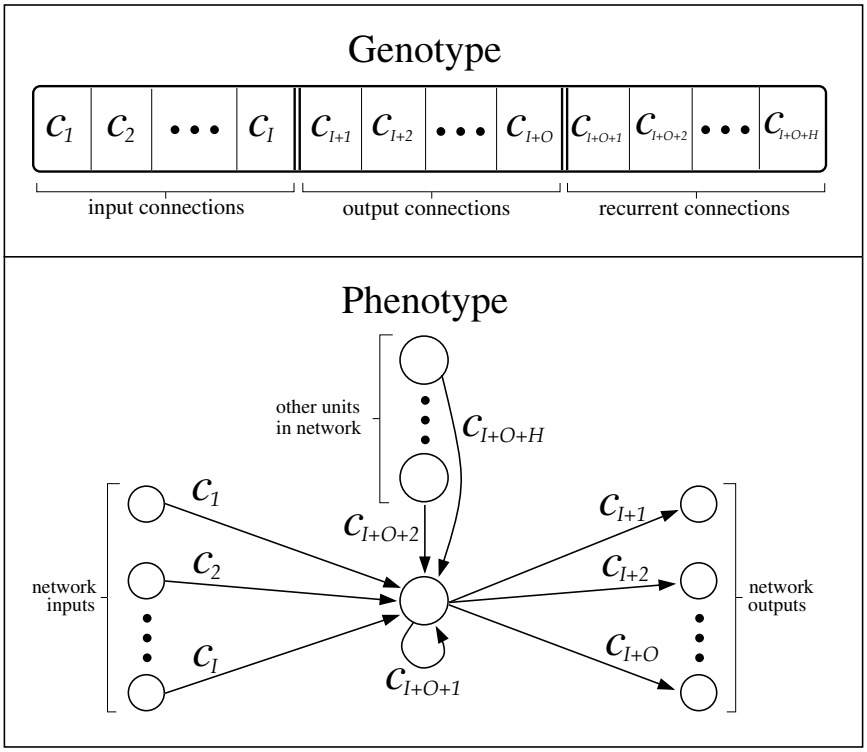


Figure 3.1: **Neuron genotype encoding.** The ESP genotypes encode the synaptic connection weights, c_i, of a single neuron as real numbers. The figure illustrates the mapping from genotype (top) to phenotype (bottom). Each chromosome has I input connections, O output connections and R connections from the other neurons in the network if the network is recurrent, where R is equal to the number of subpopulations (network size), h. If the network is fully recurrent then the activation of some of neurons serves as the network output and the output connections are not needed.

follows:

1. Initialization. The number of hidden units h is specified and h subpopulations of n neuron chromosomes are created. Each chromosome encodes the input, output, and possibly recurrent connection weights of a neuron with a random string of real numbers.

2. Evaluation. A set of h neurons is selected at random, one neuron from each subpopulation, to form the hidden layer of a network of the specified type. 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

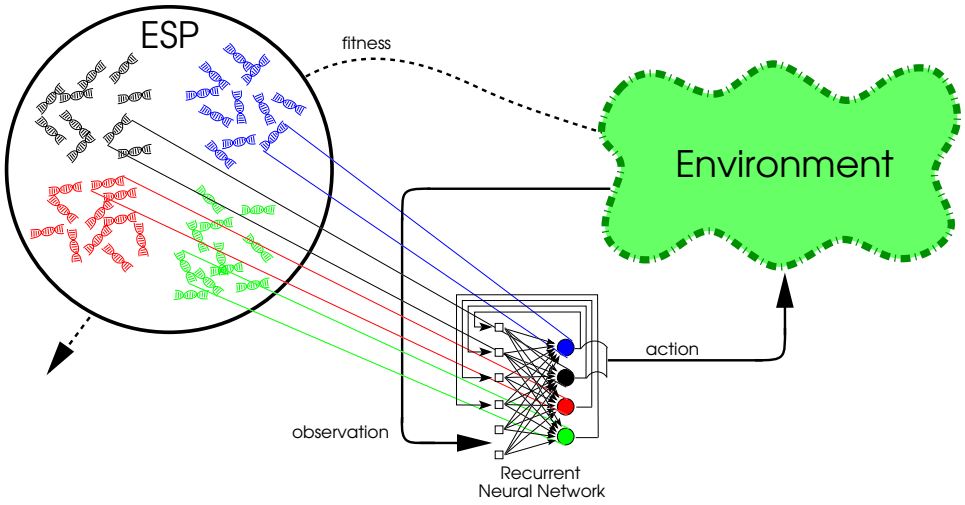


Figure 3.2: **The Enforced Subpopulations method (ESP; color figure).** The population of neurons is segregated into subpopulations shown here in different colors. Networks are formed by randomly selecting one neuron from each subpopulation. As with SANE, a neuron accumulates a fitness score by adding 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.

e.g. 10 trials.

3. Check Stagnation. If the fitness of the best network found so far has not improved in b generations burst mutation is performed. If the fitness has not improved after two burst mutations the network size is adapted.

4. 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. The offspring replace the lowest-ranking half of the subpopulation.

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

Steps 1, 2, 4, and 5 form the backbone of the algorithm. Step 3 incorporates two features that are used in the event that ESP converges prematurely: burst mutation and the adaptation of network size which are discussed in the next two sections. Figure 3.3 presents basic the algorithm in pseudocode form. Figure 3.4 presents the subroutines used in step 3, and the genetic operators.

h	:	number of subpopulations
n	:	number of neurons in each subpopulation
m	:	mutation rate, $[0, 1]$
b	:	number of generations before burst mutation is invoked
$goal{-}fitness$	:	the fitness value at which the task is considered solved
$best{-}fitness$	:	best fitness found so far
S_i	:	subpopulation $i,\ i = 1..h$
$\mathcal{N}^*$	:	the best network found so far
$\mathcal{N}(i)$	:	the i-th neuron in network $\mathcal{N}$
$\eta_{i,j}$	:	neuron j from $S_i,\ j = 1..n$
$\eta_{i,j}(k)$	:	allele (weight) $k = 1..l,$ of neuron $\eta_{i,j}$

ESP(h, n, m, b)

while $best{-}fitness < goal{-}fitness$ **do**
  INITIALIZATION
    create n random neurons for each subpopulation S_i
  EVALUATION
    for $trial \leftarrow 1$ **to** $n \times 10$
      do for $i \leftarrow 1$ **to** h /* create random network */
        do $j \leftarrow$ RAND(n)
          $\mathcal{N}(i) \leftarrow \eta_{i,j}$
        $fitness \leftarrow$ EVALUATE($\mathcal{N}$) /* evaluate network */
        for $i \leftarrow 1$ **to** h /* add fitness to each neuron */
          do $\mathcal{N}(i) \leftarrow \mathcal{N}(i) + fitness$
        if $fitness > best{-}fitness$ /* save best fitness */
          then $best{-}fitness \leftarrow fitness$
  CHECK STAGNATION
    if $best{-}fitness$ has not improved in b generations
      if this is the second consecutive time
        then ADAPT-NETWORK-SIZE()
        else BURST-MUTATE()
  RECOMBINATION
    for $i \leftarrow 1$ **to** h
      sort neurons in S_i by normalized fitness
      do for $j \leftarrow 1$ **to** $n/4$ /* mate top quartile */
        CROSSOVER($\eta_{i,j}, \eta_{i,rand(j)}, \eta_{i,j*2}, \eta_{i,j*2+1}$)
      do for $j \leftarrow n/2$ **to** n /* mutate offspring */
        MUTATE($m, \eta_{i,j}$)

Figure 3.3: **The ESP algorithm.**

```
RNDCAUCHY()  :  Cauchy distributed noise generator
        l    :  number of genes in the neuron chromosomes
 threshold   :  the criteria for removing a subpopulation, [0, 1]
-------------------------------------------------------------------

    CROSSOVER(η1, η2, η3, η4)
1       crosspoint ← RAND(l)                  /* select random crossover point */
2       for k ← 1 to l
3           if k < crosspoint                 /* exchange chromosomal segments */
4               then η3(k) ← η1(k)
5           else
6               η4(k) ← η2(k)

    MUTATE(m, η)
1       if RAND(1.0) < m                      /* if neuron η is selected for mutation */
2           then η(RAND(l)) ← RNDCAUCHY()     /* add noise to one of its weights */

    BURST-MUTATE()
1       for i ← 1 to h                        /* for each subpopulation */
2           do for j ← 1 to n                 /* for each neuron in subpopulation Si */
3               do for k ← 1 to l             /* add noise to each weight of best neuron */
4                   do ηi,j(k) ← N*(i)(k)+RNDCAUCHY()   /* assign it to ηi,j */

    ADAPT-NETWORK-SIZE()
1       fitness ← EVALUATE(N*)                /* get unlesioned fitness of best network */
2       for i ← 1 to h
3           do lesion N*(i)                   /* lesion each neuron in turn */
4           lesioned–fitness ← EVALUATE(N*)   /* get its lesioned fitness */
5           if lesioned–fitness > fitness × threshold
6               then remove N*(i)             /* decrement number of subpopulations */
7                   delete subpopulation Si
8                   h ← h − 1
9       if no neuron was removed
10          then h ← h + 1                    /* increment number of subpopulations */
11              add a subpopulation Sh
```

Figure 3.4: **ESP algorithm subroutines.**

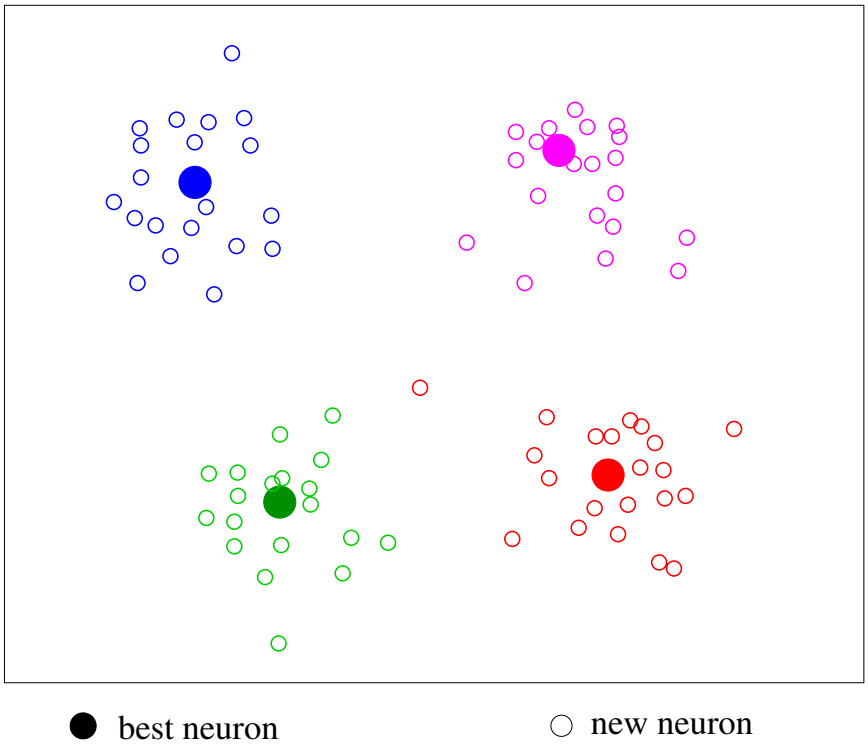


Figure 3.5: **Burst mutation.** When ESP stops making progress toward a solution, new subpopulations are created by adding Cauchy-distributed noise to the neuron chromosomes of the best network found so far. The large filled circles in the figure represent the neurons of the best network for a hypothetical 2D space ($l = 2$). The smaller circles represent the neurons in each of the new subpopulations after burst mutation is invoked. The new neurons are distributed in a region around each of the best neurons.

3.1.1 Burst Mutation

The idea of *burst mutation* is to search for optimal modifications of the current best solution. When performance has stagnated for a predetermined number of generations, the best network is saved and new subpopulations are created by adding noise to each of the neurons in the best solution (figure 3.5). Each new subpopulation contains neurons that represent differences from the best solution. Evolution then resumes, but now searching 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

the noise:

$$f(x) = \frac{\alpha}{\pi(\alpha^2 + x^2)} \tag{3.1}$$

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 maintains diversity 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 the goal is to maintain diversity, the range of the noise is not reduced on successive applications of burst mutation and Cauchy rather than uniformly distributed noise is used.

3.1.2 Adapting the Network Size

ESP does not evolve network topology: as was described in section 2.3 fully connected networks can effectively represent any topology of a given size by having some weights evolve to very small values. However, ESP can adapt the size of the networks. 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 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 burst mutation), the best network found so far is evaluated 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.

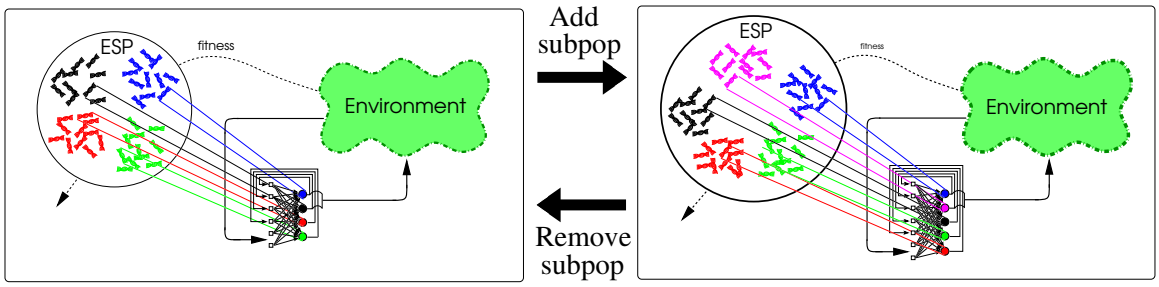


Figure 3.6: **Adapting the network size.**

If no neurons can be removed, add a new subpopulation of random neurons and evolve networks with one more neuron (figure 3.6).

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 is more robust in dealing with environmental changes and tasks where the appropriate network size is difficult to determine.

3.2 Advantages of Segregating Neurons

Both ESP and SANE evolve neurons, but, as discussed in section 2.4.1, SANE cannot reliably evolve recurrent networks. SANE does not make explicit use of the neuron specializations, and therefore it obtains noisy information about the utility of a particular neuron. In contrast, 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. In contrast, SANE forms networks that can contain multiple members of some specializations and omit members of others. 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.

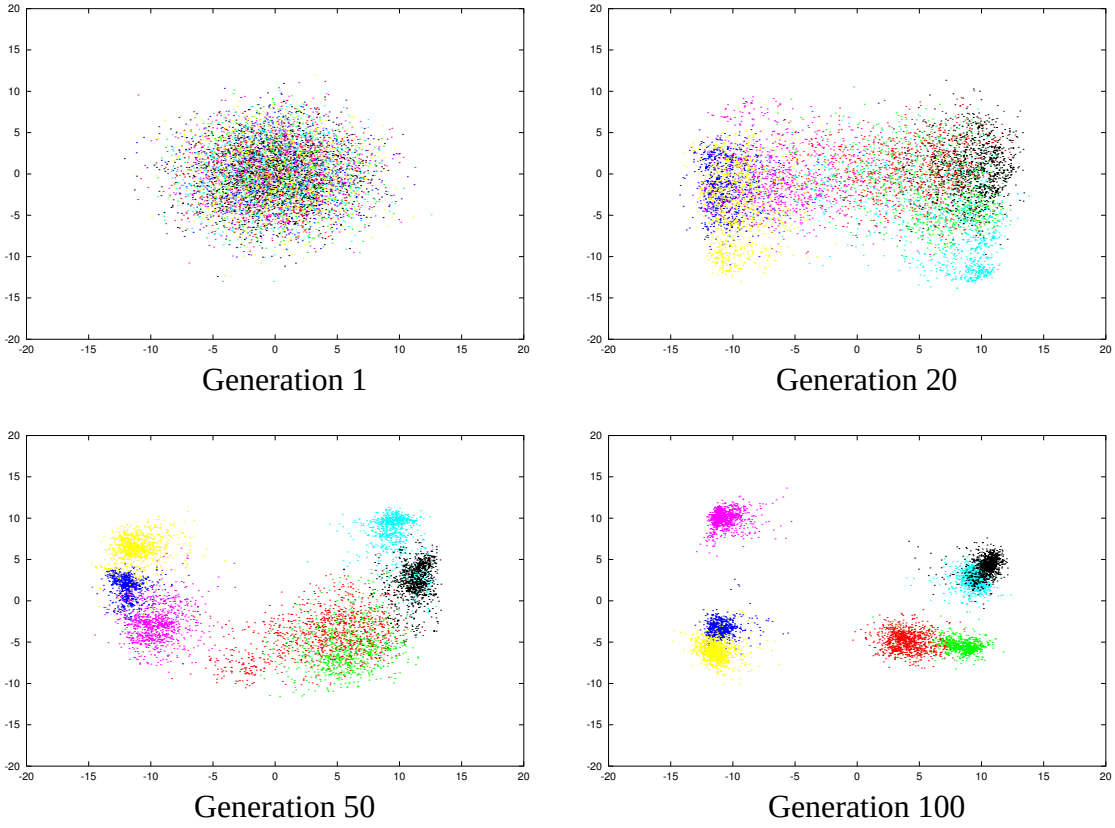


Figure 3.7: **Evolution of specializations in ESP (color figure).** The plots show a 2-D projection of the neuron weight vectors after Principal Component Analysis (PCA) transformation. Each subpopulation is shown in a different color. As evolution progresses, the subpopulations cluster into their own region of the search space. Each subpopulation represents a different neuron specialization that can be combined with others to form good networks.

Figure 3.7 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.

Chapter 4

Pole Balancing Comparisons

To evaluate how efficiently ESP can evolve effective controllers, this chapter compares ESP to a broad range of learning algorithms on a sequence of increasingly difficult versions of the pole balancing task. This scheme allows for methods to be compared at different levels of task complexity, exposing the strengths and limitations of each method with respect to specific challenges introduced by each succeeding task. Sections 4.1 and 4.2 describe the domain and task setup in detail.

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 (Schaffer and Cannon 1966; Michie and Chambers 1968; Anderson 1989). 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 unstable control problems such as bipedal robot walking (Vukobratovic 1990), and satellite attitude control (Dracopoulos 1997).

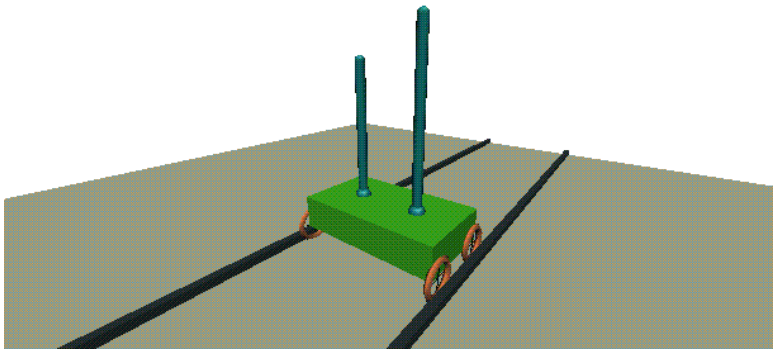


Figure 4.1: **The double pole balancing system (color figure).** Both poles must be balanced simultaneously by applying a continuous force to the cart. The system becomes more difficult to control as the poles assume similar lengths and if the velocities are not provided to the controller. The figure is a snapshot of a 3D real-time simulation. Demo available at `http://www.cs.utexas.edu/users/inaki/esp/two-pole-demo`.

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 (Moriarty and Miikkulainen 1996a; Gomez and Miikkulainen 1997).

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 4.1), 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.

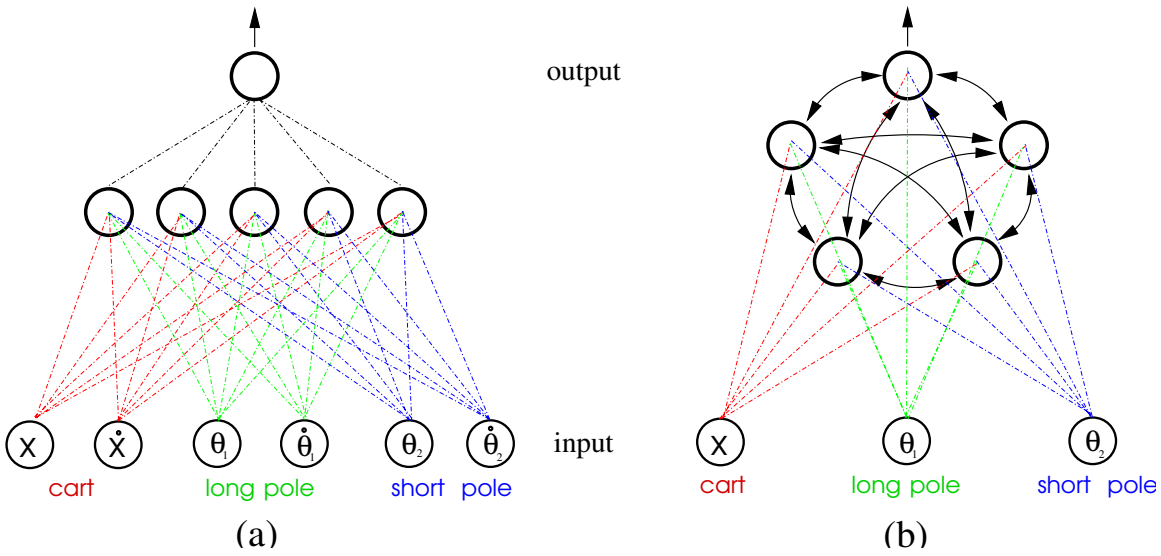


Figure 4.2: **Neural network control of the pole balancing system (color figure).** At each time step the network receives the current state of the cart-pole system $(x, \dot{x}, \theta_1, \dot{\theta_1}, \theta_2, \dot{\theta_2})$ through its input layer. For the feed-forward networks (a) used in the 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.

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 chapter.

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 4.2 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 goes off the end of the track. In keeping with the setup in prior work (e.g. Wieland 1991; Gruau et al. 1996a) the force is restricted 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.

4.3 Pole Balancing Experiments

Many algorithms have been developed for solving reinforcement learning problems. However, to my 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.

4.3.1 Other Methods

ESP was compared to nine other learning methods in the pole balancing domain. The first three are value-function methods for which I ran my own simulations. Q-MLP is my 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, CNE, and NEAT 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 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. a feed-forward 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 a 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 the policy graph to the next action node.

Symbiotic, Adaptive Neuro-Evolution (SANE; Moriarty 1997) described in section 2.4.1.

Conventional Neuroevolution (CNE) is my 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.

NeuroEvolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen 2002) is another NE method that evolves topology as well as synaptic weights, but unlike CE it uses a direct encoding. NEAT starts with a population of minimal networks (i.e. no hidden units) that can increase in complexity by adding either new connections or units through mutation. Every time a new gene appears, a *global innovation number* is incremented and assigned to that gene. Innovation numbers allow NEAT to keep track of the historical origin of every gene in the population so that (1) crossover can be performed between networks with different topologies, and (2) the networks can be grouped into "species" based on topological similarity.

Whenever two networks crossover, the genes in both chromosomes with the same innovation numbers are lined up. Those genes that do not match are either *disjoint* or *excess*, depending on whether they occur within or outside the range of the other parent's innovation numbers, and are inherited from the more fit parent.

The number of disjoint and excess genes is used to measure the distance between genomes. Using this distance, the population is divided into species so that individuals compete primarily within their own species instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before they have to compete with other species in the population.

4.3.2 Balancing One Pole

This task is the starting point for the comparisons. Balancing one pole is a relatively easy problem that gives us a base performance measurement before moving on 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.

Complete State Information

Table 4.1 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 eval-

Method	Evaluations	CPU time (sec)
Q-MLP	2,056	53
SARSA-CMAC	540	487
SARSA-CABA	965	1,713
CNE	352	5
SANE	302	5
NEAT	743	7
ESP	289	4

Table 4.1: **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.

uated $O(|A|)$ 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 can be attributed the superior efficiency with which the MLP is updated. 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.

This task poses very little difficulty for the NE methods. However, NEAT required more than twice as many evaluations as CNE, SANE, and ESP because it explores different topologies that initially behave poorly and require time to develop. For this task the speciation process is an overkill—the task can be solved more efficiently by devoting resources to searching for weights only. The observed performance difference between CNE, SANE and ESP is not statistically significant.

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 when the concomitant problem of velocity calculation must also be solved.

Method	Evaluations	CPU time	%Success
VAPS	(500,000)	(5days)	(0)
Q-MLP	11,331	340	100
SARSA-CMAC	13,562	2,034	59
SARSA-CABA	15,617	6,754	70
CNE	724	15	100
NEAT	1523	15	100
ESP	589	11	100

Table 4.2: **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 my criteria was reported by Meuleau et al. (1999).

To allow Q-MLP and the SARSA methods to solve this task, their inputs are extended to include the immediately previous cart position, pole angle, and action $(x_{t-1}, \theta_{t-1}, a_{t-1})$ in addition to x_t, θ_t, and a_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 4.2 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, one order of magnitude faster than Q-MLP, and approximately twice as fast as NEAT. The performance of the three 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.

Method	Evaluations	CPU time (sec)
Q-MLP	10,582	153
CNE	22,100	73
EP	307,200	—
SANE	12,600	37
NEAT	3,600	31
ESP	3,800	22

Table 4.3: **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, NEAT, and ESP data are the average of 50 simulations, and all differences are statistically significant ($p < 0.001$) except the number of evaluations for NEAT and ESP.

4.3.3 Balancing Two Poles

The first two tasks show that the single pole environment poses no challenge for modern reinforcement learning methods. The double pole problem is a better test environment 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 non-linear interactions between the poles.

Complete State Information

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

Table 4.3 shows the results for the two-pole configuration with complete state information.

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. ESP and NEAT are statistically even in terms of evaluations, requiring roughly three times fewer evaluations than SANE. In terms of CPU time, ESP has a slight but statistically significant advantage over NEAT.

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

Method	Evaluations	
	Standard fitness	Damping fitness
CE	—	(840,000)
CNE	76,906	87,623
NEAT	20,918	24,543
ESP	20,456	26,342

Table 4.4: **Two poles with incomplete state information.** The table shows the number of evaluations for CNE, NEAT, and ESP using the standard fitness function (middle column), and using the damping fitness function, equations 4.1 and 4.2 (right column). Results are the average of 50 simulations for all methods except CE which is from a single run. All results are statistically significant except for the difference between ESP and NEAT using the standard fitness function.

velocities, i.e. it does not know how fast or in which direction the poles are moving.

Gruau et al. (1996a) were the first to tackle the two-pole problem without velocity information. Although they report the performance for only one simulation, their results are included to put the performance of the other methods in perspective. SANE is not suited to non-Markov problems and none of the value-function methods that were tested made noticeable progress on the task after approximately 12 hours of computation. Therefore, in this task, only ESP, CNE, NEAT, and the Cellular Encoding (CE) method are compared.

To accomodate a comparison with CE, controllers were evolved using both the standard fitness function used in the previous tasks and also the "damping" fitness function used by Gruau et al. (1996a). The damping fitness is the weighted sum of two separate fitness measurements ($0.1f_1 + 0.9f_2$) taken over a simulation of 1000 time steps:

$$f_1 = t/1000, \tag{4.1}$$

$$f_2 = \begin{cases} 0 & \text{if } t < 100 \\ \left(\dfrac{K}{\sum_{i=t-100}^{t} (|x^i| + |\dot{x}^i| + |\theta_1^i| + |\dot{\theta}_1^i|)} \right) & \text{otherwise,} \end{cases} \tag{4.2}$$

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 4.2 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 standard fitness measure could produce networks that would balance the poles by merely swinging them back and forth (i.e. without calculating the velocities).

Table 4.4 compares the four neuroevolution methods for both fitness functions. To determine when the task was solved for the damping fitness function, the best controller from each generation was tested using the standard fitness to see if it could balance the poles for 100,000 time steps. The results for CE are in parenthesis in the table because only a single run was reported by Gruau et al. (1996a).

Using the damping fitness, ESP, CNE, and NEAT required an order of magnitude fewer evaluations than CE. ESP and NEAT were three times faster than CNE using either fitness function, with CNE failing to solve the task about 40% of the time.

The performance of ESP versus NEAT was again very similar. Both methods solved the problem quickly and reliably. This is an interesting result because the two methods take such different approaches to evolving neural networks. NEAT is based on searching for an optimal topology, whereas ESP optimizes a single, general, powerful topology (i.e. fully recurrent networks). At least in the difficult versions of the pole balancing task, the two approaches are equally strong.

One significant practical difference between ESP and NEAT is the number of user parameters that must be set. In ESP, there are five, whereas NEAT requires 23 (see Appendix B). Such simplicity may make ESP a more general method, allowing it to be adapted more easily to other tasks. On the other hand, it is possible that topology optimization is crucial in some domains. The relative advantages of these two approaches constitute a most interesting direction of future study.

4.3.4 Summary of Comparisons

The results of the comparisons show that neuroevolution is more efficient than the 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 NEAT and ESP increasing their lead the more difficult the task became. While these methods are roughly equivalent on the hardest tasks, ESP provides a simpler approach with only a few user parameters.

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. The next two chapters will show that to do so (1) an incremental evolution approach is needed to scale the evolutionary search to tasks that are too hard to solve directly, and (2) controllers

must be made robust enough to successfully make the transition from simulation to the real world.

Chapter 5

Incremental Evolution

The experiments in chapter 4 demonstrated how ESP can be used to solve difficult control problems efficiently. However, there will always be tasks that are too challenging for even the most efficient NE system. In such cases, it may be possible to still make progress by shifting the focus away from algorithm design to the task itself. If the task can be transformed in a way that preserves its essential features while making the desired behavior easier to evolve, then solving this transformed task can provide a springboard for solving the original task. This is the basic idea of the incremental evolution described in this chapter.

Section 1.1 develops the basic concept of incremental evolution and explains how the process guides evolution to solve otherwise intractable tasks. Section 1.2 presents experiments that compare incremental evolution to conventional, direct evolution using the most difficult task in chapter 4 (double pole balancing without velocity information). The results demonstrate how tasks that cannot be solved directly may be solved efficiently by decomposing the problem into a sequence of increasingly difficult tasks.

5.1 The Limits of Direct Evolution

Evolution is a process that relies on diversity to make progress—without sufficient variation in the fitness of individuals, evolution cannot clearly identify promising regions of the search space. However, variation alone is not enough: in order to solve a given task, the most fit individuals need to be situated in a region of the search space that is near a solution, and there must be a path from those individuals to a solution (i.e. a sequence of genetic operations). If the percentage of the search space that constitutes a solution is very small, and the fitness landscape very rugged, then the probability of producing

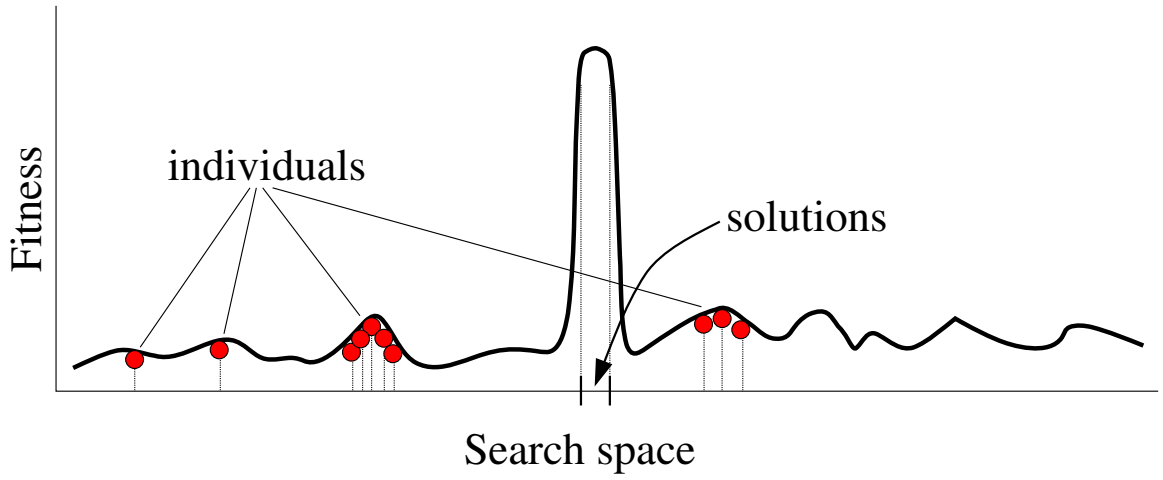


Figure 5.1: **A difficult fitness landscape.** The curve represents the fitness value of each point in a hypothetical 1-dimensional search space. The red circles are the individual members of the population. Because the solutions represent only a small part of the search space and the fitness landscape is rugged, the initial population may not sample the fitness near this region. As a result the population converges prematurely to a local maxima.

such individuals in the initial random population will be low, and evolution will not make progress. All individuals in the initial population perform poorly and the population gets trapped in suboptimal regions of the fitness landscape during the early stages of evolution (figure 5.1). Enlarging the population will make it more likely that fruitful regions of the search space are sampled, but given the very high dimensionality of neural networks (i.e. hundreds or even thousands of synaptic weights) a prohibitively large population may be required in order to make progress.

One way to scale neuroevolution to tasks that are too difficult to evolve directly is to begin by viewing the task we want to solve or *goal task* as a member of a family of tasks. Each member of the family is parameterized by the *free parameters* in the environment, i.e. the variables that are not under the controller's direct control (Mataric and Cliff 1996). For instance, in a robot navigation task the free parameters might consist of the size and speed of moving obstacles, the maximum velocity of the robot, the amount of sensor noise, etc. The space of all possible parameter settings is called the *configuration space* of the problem (Schultz 1991). Each point in the space represents a specific case (configuration) of the conditions in which a controller can be evaluated. All configurations have the same objective, but some configurations are more difficult than others to solve using an evolutionary algorithm, i.e. some will require larger populations and more evaluations than others (Figure 5.2). Returning to the robot navigation example: large, fast moving

obstacles are harder to navigate through than small, slower ones.

The normal, *direct* approach to evolving controllers is to evaluate the population using the configuration that corresponds to the conditions expected in the real world. However, if this goal task proves too difficult to solve directly, configurations that may not be useful or even possible in the real world can be used during evolution as *evaluation tasks* to make the goal task easier to solve.

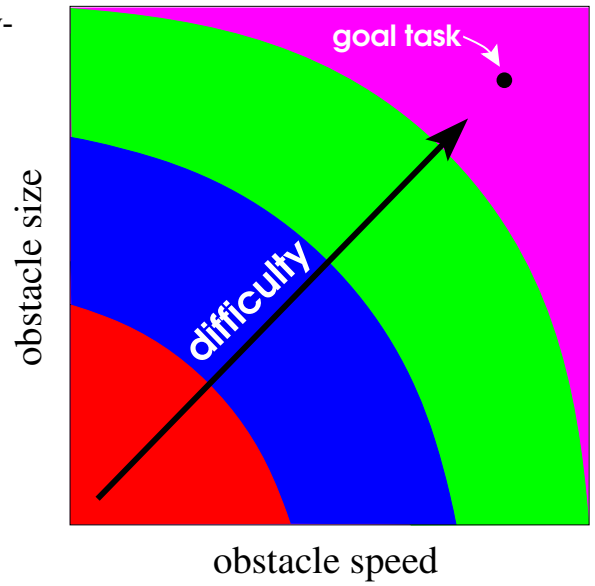


Figure 5.2: **Configuration space.** In this 2-D configuration space for a hypothetical robot navigation task, the difficulty increases with the free parameters, obstacle size and speed. The goal task is in the most difficult region where obstacles are large and move quickly.

Instead of evolving a solution to the goal task directly, evolution starts in a part of the configuration space that can be solved more easily. Once this first evaluation task, t_1, is solved, it is updated to a more difficult task, t_2, that is closer to the goal task in the configuration space. Therefore, evolution proceeds incrementally by solving a sequence of evaluation tasks $\{t_1, t_2, t_3, \ldots, t_n\}$ that traverse the configuration space until the goal task, t_n, is solved.

For example, in figure 5.2, the goal task is in the most difficult part of the configuration space. If this task is too difficult to solve directly, then evolution can be started somewhere in the "easy," red region and then move through the increasingly difficult blue and green regions until the goal is reached. Whenever incremental evolution is used throughout this dissertation, the evaluation tasks are referred to by the notation $e_{p_1,\ldots,p_n}$, where each p_i is one of n free parameters of interest.

Each evaluation task provides a bias for solving the next task by situating the population on the gradient to the next solution. Figure 5.3 is a conceptual visualization of this process. The goal task, t_n, is too hard to solve directly because the solution set occupies a small part of the search space and fitness drops steeply in the neighborhood around the solution set. A very different situation exists in t_1 (i.e. the initial task) where the solution set is large and there is a gentle gradient in the fitness landscape leading to the solution set. Evolution can more easily find this region because even if no individual is in the solution set, individuals near the set will have higher than average fitness and drive the population

towards the solution region.

As tasks get more difficult, the solution set becomes smaller, but because successive tasks are related, each task positions the population in a good region of the space to solve the next task. For instance, individuals that are situated on the gradient or in the solution set of t_1 will have a good chance of also being on the gradient leading to the solution set of t_2. Eventually, if the tasks are generated properly the goal task can be achieved. However, if evolution is unable to make the transition from t_i to t_{i+1}, then t_{i+1} is too far away in the configuration space for the amount of diversity in the population. In order to continue making progress toward the goal task in this event, either additional diversity must be introduced or t_{i+1} must be brought closer to t_i. Both of these mechanisms are used in the incremental evolution experiments below.

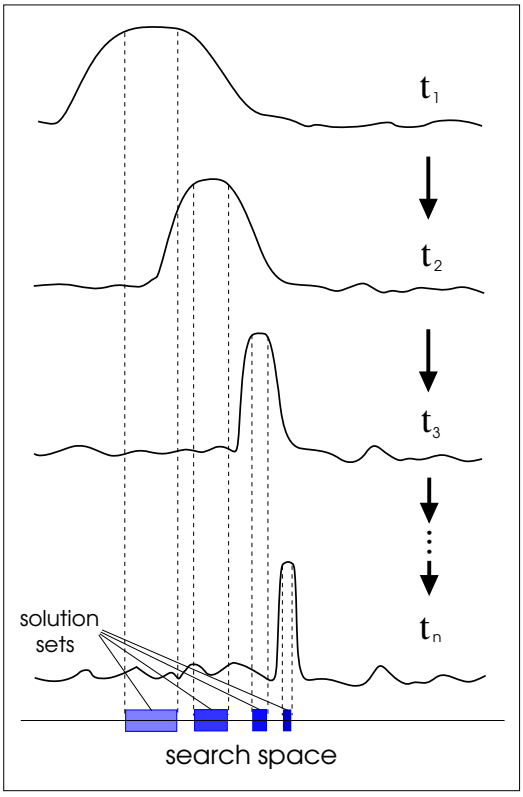


Figure 5.3: **Incremental fitness landscapes.** The figure illustrates, for a 1-dimensional search space, how incremental evolution works by gradually reshaping the fitness landscape to guide the population toward a solution to the goal task. The initial task, t_1, provides an easy target for evolution, which positions the population in the correct region to approach t_2. Successive tasks do the same until the goal task t_n is reached.

Note that in general the goal task will not be a single point in the configuration space but rather a set of configurations. This is because the values of the free parameters in the target environment are often not known exactly or vary over time. Therefore, controllers need to be evaluated in multiple trials with different configurations to reduce evaluation noise, and to ensure that the range of conditions likely to occur in the target environment are covered. To keep the focus of this chapter on the incremental evolutionary process, the experiments below use one configuration (trial) per network evaluation. The problem of evolving robust controllers to ensure successful transfer is left for chapter 6.

The next section demonstrates the power of incremental evolution by comparing it to the direct approach in the pole balancing domain.

5.2 Experiments

An interesting aspect of the double pole system is that it becomes more difficult to control as the poles assume similar lengths (Higdon 1963). When the short pole is more than half the length of the long pole, the system is extremely delicate and requires a level of precision that tests the limits of direct evolution. By keeping the length of the long pole constant, and making the length of the short pole, l, a free parameter, the system provides a 1-dimensional configuration space in which to study incremental evolution. Starting with l at a relative easy value, the length of the shorter pole can be increased gradually until the desired goal task is reached. The experiments in this section compare the performance of direct versus incremental evolution in solving progressively harder versions of the non-Markov double pole balancing task (i.e. task 2b in Chapter 4).

5.2.1 Experimental Setup

For the direct evolution experiments, the length of the short pole, l, was fixed throughout the course of evolution. Eight sets of 50 simulations were run, each with a different value of l, from 0.15 meters up to 0.5 meters in 0.05 meter increments. Each set will be referred to by its corresponding evaluation task, e_l: $e_{0.15}, e_{0.2}, e_{0.25}, \ldots, e_{0.5}$. All other parameters settings for the pole balancing system were identical to those used in task 2b.

For the incremental experiments, 50 simulations were run using the following rule to schedule the sequence of tasks (figure 5.4): Evolution begins with $e_{0.1}$ as the initial evaluation task t_1. If this task is solved, the short pole is lengthened by a predefined increment (P) to generate the next evaluation task $e_{0.1+P}$ or t_2. If t_2 is solved the short pole is lengthened again yielding $e_{0.1+2P}$, and so on.

However, if along the way ESP is unable make the transition from some task t_i to t_{i+1} after two burst mutations, then P is halved and subtracted from the length of the short pole. New subpopula-

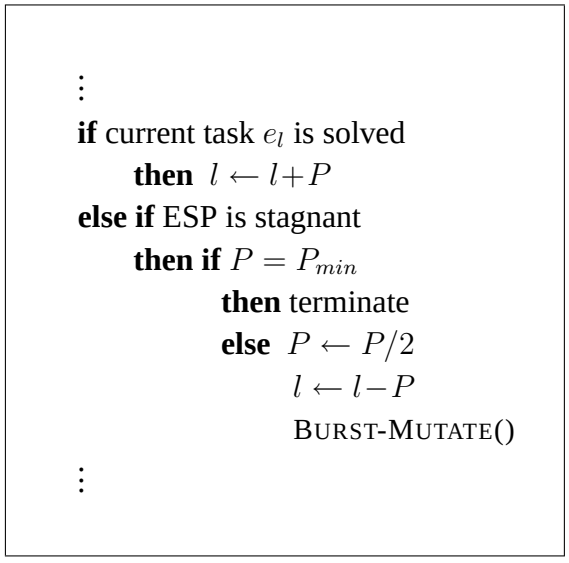


Figure 5.4: **Task scheduling rule for incremental pole balancing.**

tions are created by burst mutating around the solution to t_i and evolution continues with the new value for P. Tasks can be repeatedly simplified, with P decreasing monotonically, until either a transition occurs or a lower bound P_{min} is reached. If $P \leq P_{min}$, and ESP cannot transition to the next task after two burst mutations or before the total number of generations reaches 1000, then the simulation terminates.

The task schedule for incremental evolution can be illustrated best with an example. Normally with this method the task differences are quite large at first. As the networks move on to harder tasks, P tends to shrink, and more task transitions are required for a given increase in the length of the short pole. For the initial value for P of 0.02 used in these experiments, a typical evolution schedule might look like:

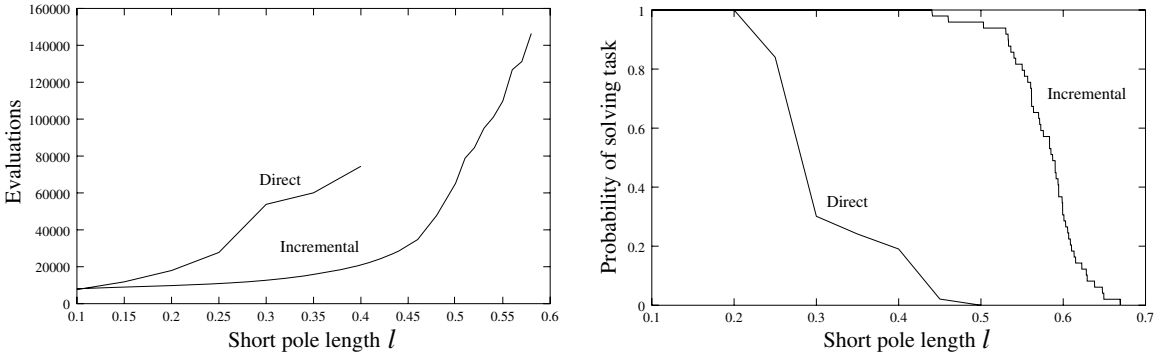
$$\overbrace{e_{0.1} \rightarrow e_{0.12} \rightarrow \cdots \rightarrow e_{0.30} \rightarrow (e_{0.32})}^{P=0.02}$$
$$\searrow$$
$$\overbrace{e_{0.31} \rightarrow e_{0.32} \rightarrow \cdots \rightarrow e_{0.35} \rightarrow (e_{0.36})}^{P=0.01}$$
$$\searrow$$
$$\overbrace{e_{0.355} \rightarrow e_{0.36} \rightarrow \cdots}^{P=0.005}$$

where the parenthesis denote a task that could not be solved within two burst mutations, and a diagonal arrow indicates that evolution has backtracked to an easier task.

5.2.2 Results

Figure 5.5 compares the performance of direct and incremental evolution. In plot (a), each data point in the direct evolution curve is the average number of evaluations each set of experiments required to solve its corresponding task, e_l. The incremental evolution curve is the average evaluations required to solve each e_l starting with $e_{0.1}$ and transitioning to harder tasks according to the task scheduling rule (figure 5.4). Plot (b) shows the probability of solving each task for the two approaches. The two curves in plot (a) stop at the value of l that has less than a 0.2 probability in plot (b).

Direct evolution was only able to solve $e_{0.4}$ 20% of the time (i.e. with probability 0.2 in plot (b)), $e_{0.45}$ once out of 50 runs, and could not solve $e_{0.5}$. When the task is this hard the population converges prematurely in a low payoff region of the search space. No individual does well enough to guide the search, and ESP selects genotypes that are slightly better than others in terms of the fitness scalar but are not necessarily any closer to the goal.



(a) (b)

Figure 5.5: **Results for incremental vs. direct evolution.** Plot (a) shows the average number of evaluations required to solve each task for both the direct and incremental approaches. Each data point is the average of those simulations in each set of 50 that were able to solve the task within 1000 generations. Plot (b) shows the probability of solving each task e_l within 1000 generations. Incremental evolution can solve much harder tasks in significantly fewer evaluations.

In contrast, incremental evolution was always able to solve $e_{0.44}$, and solved $e_{0.5}$ with a probability of 0.96. Moreover, the approach was able to solve more difficult tasks (up to $e_{0.66}$) than were possible directly, and required 75% fewer evaluations to solve $e_{0.4}$, the hardest task solved with any reliability by direct evolution.

Each incremental evolution simulation used an average of 198 successful task transitions. Figure 5.6, shows the average value of P, the change in l, for each task. For the first eleven task transitions P remained at its initial value of 0.02, always reaching $e_{0.4}$ before decrementing P from its initial value. This means that after solving the initial task $e_{0.1}$ the short pole will be increased by 20% to $e_{0.12}$. These changes to the environment are significant: other approaches that have applied shaping to the easier double pole task (with velocities) have incremented the short pole by only 1% (Wieland 1991; Saravanan and Fogel 1995). After reaching $e_{0.4}$, the average P drops sharply as the task assumes a level of difficulty beyond that which could be solved reliably by direct evolution.

In summary, evolving incrementally allows ESP to solve more difficult tasks, it also dramatically improves the efficiency with which tasks that are accessible to direct evolution are solved.

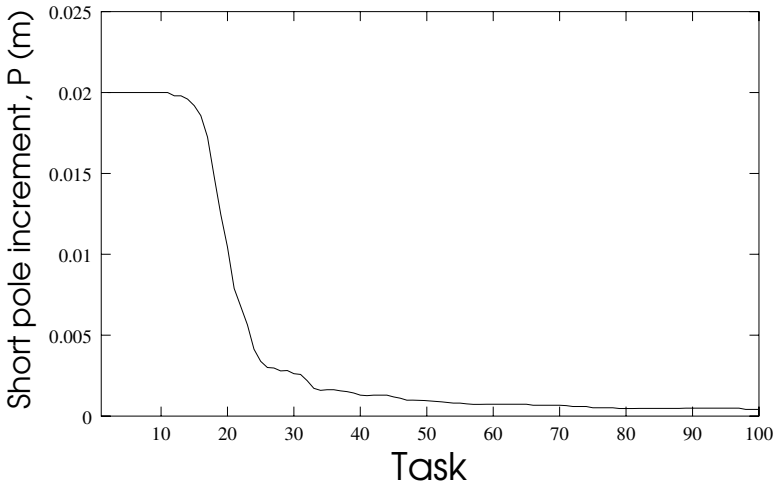


Figure 5.6: **Average increment P in short pole length.** The plot shows the value of P for each task t_i, averaged over the 50 incremental evolution simulation. Over about the first ten task transitions, P remains at its initial value or 0.02 meters. As the short pole approaches 0.4 meters in length, the task transitions become difficult. Consequently, P is adjusted automatically so that successive tasks are closer together in the configuration space, allowing further transitions to be made.

5.3 Discussion

Incremental evolution makes it possible to solve more complex tasks. This is not to say that behaviors evolved incrementally cannot be evolved directly. Such behaviors are points in the network weight space that, in principle, can be found by direct search. However, such a search requires larger populations and more evaluations. Therefore, incremental evolution extends the range of tasks that can be solved with a given amount of computational resources.

The gradual change to the fitness landscape in incremental evolution is related to the process that occurs in competitive coevolution where the fitness landscape of one population is *deformed* by that of the other competing populations. That is, the fitness of an individual in population A is coupled to that of its "opponents" in population B. If all goes well, the landscapes adapt to promote better and better individuals on both sides. From the perspective of an individual in one population, the task changes from one generation to the next. This automatic scaling of the fitness function can be useful in domains such as game playing where the opponent (i.e. the environment) is not known in advance and an optimal opponent usually does not exist. However, incremental evolution allows the designer to use available knowledge and intuition about the problem domain to guide evolution toward a

solution that is relevant to a set of specific operating conditions.

Although the most obvious application of the incremental approach is in scaling evolutionary algorithms, it is not restricted to this purpose. With burst mutation, any neuro-controller (whether trained or evolved) can provide a starting point for adapting an existing behavior to new conditions or operational objectives.

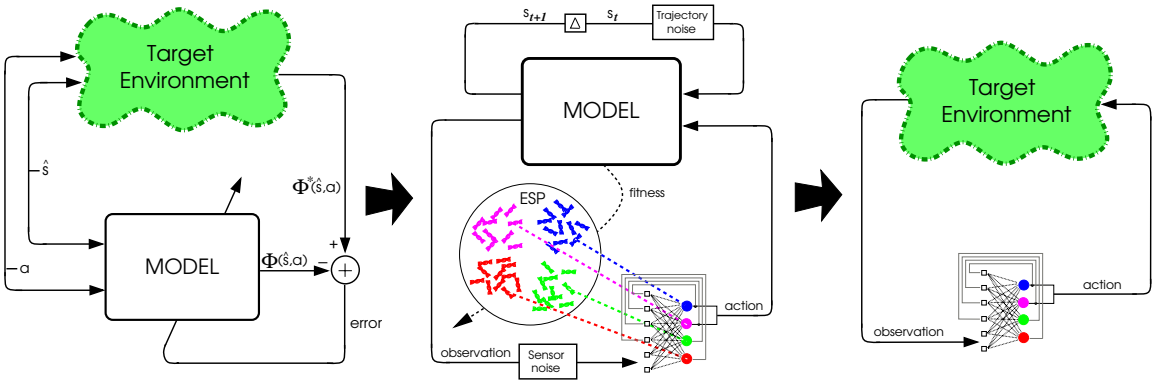
Chapter 6

Controller Transfer

In the experiments conducted so far in chapters 4 and 5 , 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.

In this chapter, the problem of transferring controllers for unstable, non-linear systems is studied continuing with the non-Markov two-pole task (task 2b). However, instead of using the analytical system (i.e. the differential equations in Appendix A) as the simulation environment it is now the target environment, and the simulation environment is an approximation of it.

Figure 6.1 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 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, controller



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

Figure 6.1: **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 $\widehat{s}$ 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.

transfer can be tested 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 the method respectively. In section 6.4 the robustness of transferred solutions is tested, and in section 6.5 the results of the experiment are analyzed.

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:

$$\widehat{s}_{t+1} = \Phi^*_\delta(\widehat{s}_t, a_t), \quad \forall \widehat{s} \in \widehat{S}, a \in A, \tag{6.1}$$

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

$$s_{t+1} = \Phi_\delta(s_t, \pi(s_t)), \tag{6.2}$$

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

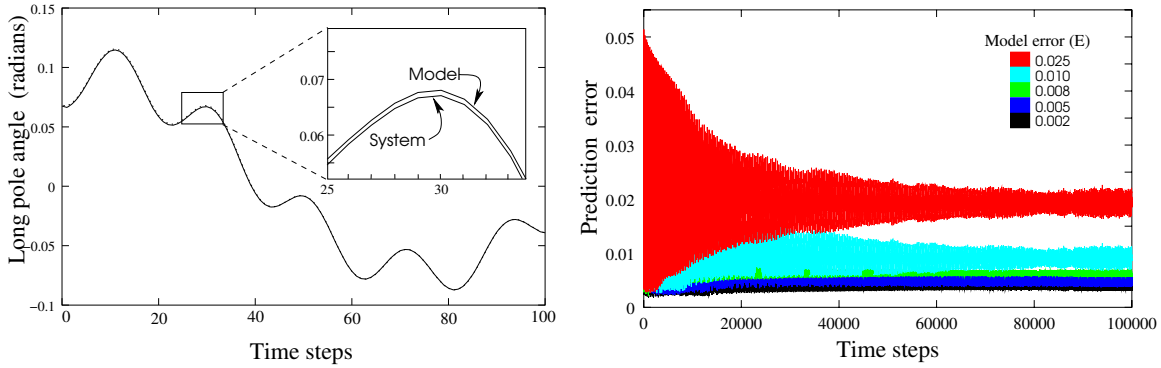
The modeling of the two-pole system followed a standard neural network system identification approach (Barto 1990). Figure 6.1a illustrates the basic procedure for learning the model. The model is represented by a feed-forward neural network (MLP) that is trained on a set of state transition examples 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 $(\widehat{s}^i, a^i)$, $\widehat{s} \in \widehat{S}, a \in A$, the next state $\Phi^*_\delta(\widehat{s}^i, a^i)$ was generated 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 δ is dropped 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 the training set. We measured accuracy in terms of model error E:

$$E = \frac{1}{N} \sum_{i=1}^{N} \|\Phi^*(\widehat{s}^i, a^i) - \Phi(\widehat{s}^i, a^i)\|^2, \tag{6.3}$$



(a) Local error (b) Prediction error

Figure 6.2: **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.

This is the average squared error in the model's prediction across the entire test set of $N = 500$ samples. The highest accuracy achieved was $E = 0.002$ using a network with 20 hidden units.

The 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:

$$\widehat{s}_1,\ \widehat{s}_2,\ \widehat{s}_3, \ldots, \qquad \text{where } \widehat{s}_{i+1} = \Phi^*(\widehat{s}_i, \pi(\widehat{s}_i)), \tag{6.4}$$

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

$$\widehat{s}_1,\ \Phi(\widehat{s}_1, \pi(\widehat{s}_1)),\ \Phi(\widehat{s}_2, \pi(\widehat{s}_2)), \ldots, \tag{6.5}$$

where π is the same policy in both equation 6.4 and 6.5, then the trajectories will be very similar (figure 6.2a). 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, the weights of the model were saved 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 6.2b 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_y. 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. I 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 4.3, 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 controllers were evolved 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 4.3, except instead of interacting with the analytical system (Appendix A), they interact with the model according to equation 6.2. 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

$$s_{t+1} = \Phi(s_t, \pi(s_t + v)), \tag{6.6}$$

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; Reynolds 1994b; Miglino et al. 1995b).

In figure 6.1b, 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, sensor noise is used 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

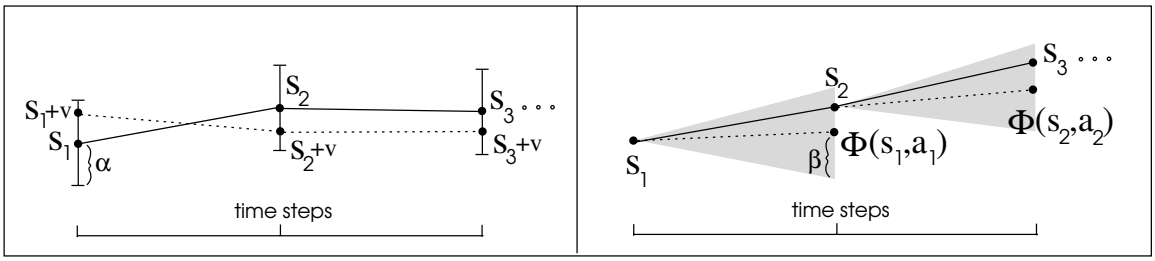
$$s_{t+1} = \Phi(s_t, \pi(s_t)) + w, \tag{6.7}$$

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 6.7 ensures that the the trajectory of the model remains stochastic even in the absence of a control signal, $\pi(s) = 0$.

Eight different levels of trajectory noise $\{0.5\%, 0.6\%, \ldots, 1.2\%\}$ were used. 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 6.3 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



(a) Sensor noise (b) Trajectory noise

Figure 6.3: **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 6.6. 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.

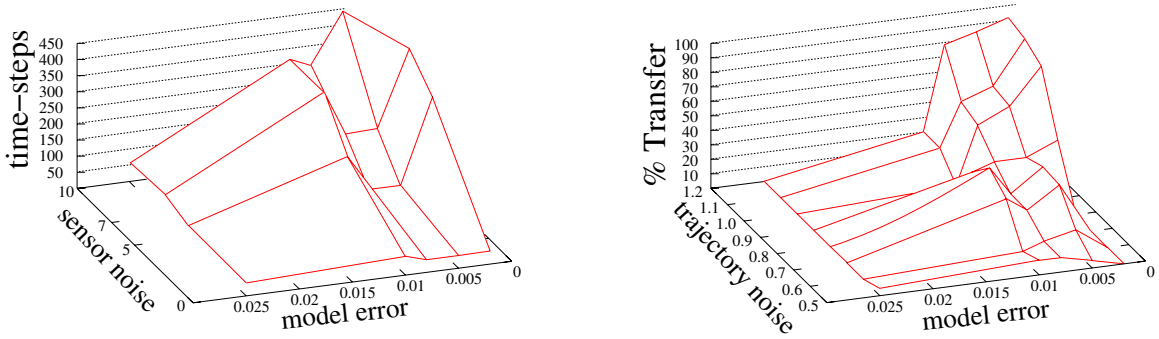
population. Therefore, ESP must be able to tolerate evaluation noise in order to search the space of controllers effectively.

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. The criteria for successful transfer was whether a controller could perform the same task in the target environment that it performed 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 2.6). However, here it constitutes a minimum requirement that a controller must satisfy before the more rigorous examination in the next section.

Sensor Noise

Figure 6.4a 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



(a) Sensor noise (b) Trajectory noise

Figure 6.4: **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.

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 effect on transfer. Because successes were frequent, a different plot is used. Figure 6.4b shows the percentage of networks that transferred successfully for different levels of model error and trajectory noise. Each 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, most of these "untransferred" controllers were found to be "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

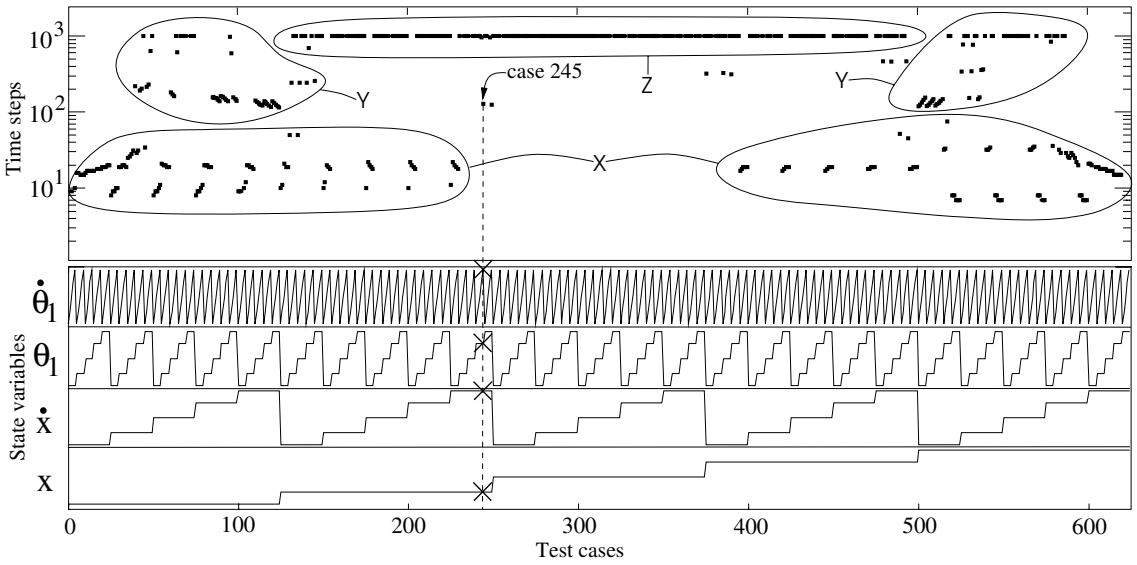
During evolution a controller can only be exposed to a subset of the conditions that are possible in the real world. Therefore, how will it perform under conditions that differ from those encountered during evolution? In this section, I analyze the quality of transferred controllers in 3 respects: (1) 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, this analysis pertains only to those controllers. Also, because different levels of trajectory noise had different transfer rates (figure 6.4b), additional controllers were evolved at each noise level until a minimum of 20 successfully transferred controllers were obtained for each noise level.

Generalization

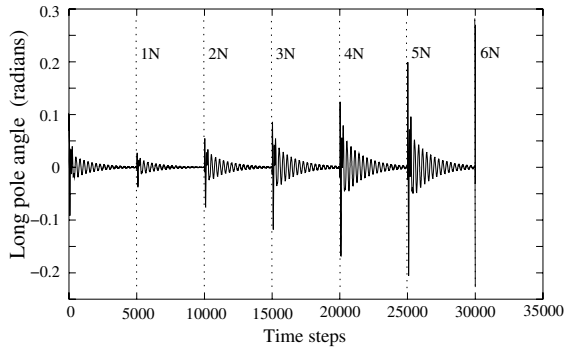
In the transfer experiments, the controllers were evaluated from the same starting state in both the simulation and target environments. Therefore, successful transfer gives little insight into how well a controller can stabilize the system from states not visited during evolution.

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 $\dot{x}$, $\pm 3.6\,\mathrm{deg}$ for θ_1, and $\pm 8.6\,\mathrm{deg}$/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. Here the generalization test is used to measure how trajectory noise affects the performance of transferred controllers in a broad range of initial states. 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.

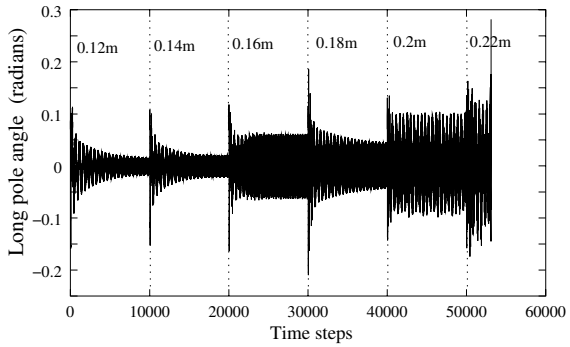
Figure 6.5a is a visualization of a controller's performance on this test. Each dot in the upper half of the graph identifies the number of time steps the poles could be balanced for a particular start state, i.e. test case. Each test case is denoted by a unique setting of the four state variables $x, \dot{x}, \theta_1, \dot{\theta}_1$ 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



(a) Generalization

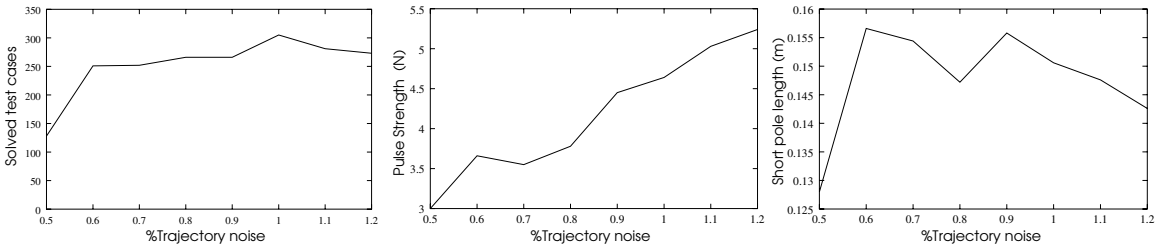


(b) External Disturbances



(c) Changing Environments

Figure 6.5: **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 6.6: **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 that a 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.

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.

The graph reveals at least 3 qualitatively different regions of controllability in the state space. They are 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 6.6a 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 to a large number of cases in the target environment that were not experienced in the simulation environment.

External Disturbances The generalization test measures how well networks behave across a large region of the state space. Another important question is: how well will these solu-

tions 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 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.

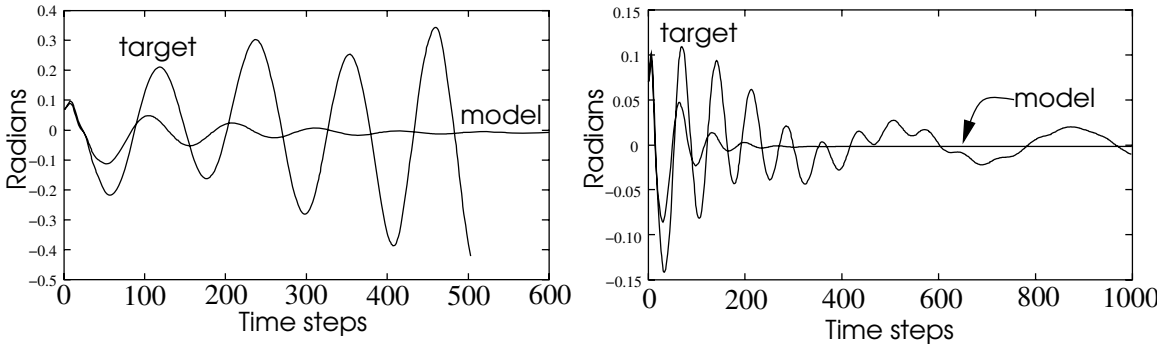
Figure 6.5b shows the angle of the long pole for a typical disturbance test. In the first 5,000 time steps the controller stabilizes the system from its unstable initial state. After the first pulse hits, the system recovers rapidly. As the pulse is strengthened, the controller takes longer to recover, until the force becomes too large causing the system to enter into an unrecoverable state. Figure 6.6b 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: it is 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 6.5c shows the behavior of the long pole during one of these tests.

Figure 6.6c 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%).



(a) Sensor noise (b) Trajectory noise

Figure 6.7: **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.

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. 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 6.7a. 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 6.7a).

This result differs from the experience of many researchers (e.g. Reynolds 1994b; Miglino et al. 1995b) that have used sensor noise to make robots more robust and facilitate transfer. I believe that sensor noise was not effective in the non-Markov two pole task. In robot navigation tasks (reviewed in section 2.6), small inaccuracies in actuator values do not affect the transferred behavior qualitatively. In contrast, the pole balancing domain requires very precise control and is much less forgiving because it is inherently unstable: a sequence of poor or even imprecise actions can quickly lead to failure. Also sensor noise is much less of an issue in the pole balancing domain compared to the robot domain where sensor error is notoriously problematic. The sensor error in a real cart-pole system would be 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

$$u_{i+1} = \max_{s_i \in [l_i, u_i]} \|\Phi(s_i, \pi(s_i)) + \overline{w}\|, \tag{6.8}$$

$$l_{i+1} = \min_{s_i \in [l_i, u_i]} \|\Phi(s_i, \pi(s_i)) - \overline{w}\|, \tag{6.9}$$

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 6.7. 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 6.8). 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 6.7b).

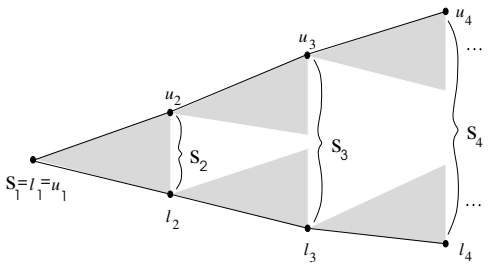


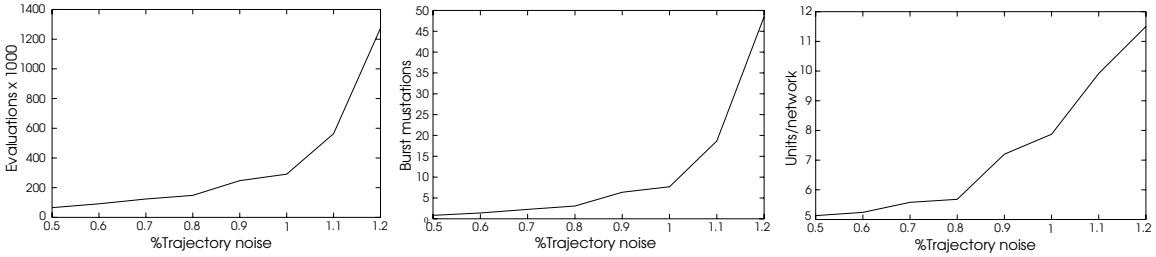
Figure 6.8: **Trajectory envelope.** Trajectories $\{l_i\}$ and $\{u_i\}$ are the upper and lower bounds on the possible trajectories that the model can take for policy π and a given amount of trajectory noise. The shaded areas show the possible state transitions from l_i and u_i due to noise. The sets 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 6.7. 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.

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.

It should be noted 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 6.9) 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, trajectory noise was bounded at 1.2% in the 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 $E \rightarrow 0$. This is not guaranteed either since E is defined on a training set of finite size (equation 6.3). 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 the 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



(a) (b) (c)

Figure 6.9: **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.

can potentially be adapted to the target environment by local search. For each unsuccessful network from the transfer experiments the weights were perturbed 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 average. This means that evolving with high trajectory noise and low model error produces "near misses" 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 (Nolfi et al. 1994; Miglino et al. 1995a).

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.

Chapter 7

Prey Capture Task

This chapter demonstrates ESP in an Artificial Life setting that emphasizes the role of short-term memory in the development of complex general behavior. Unlike the continuous, deterministic pole balancing domain used throughout the previous three chapters, the *prey capture* task used here is discrete and stochastic, and places greater demands on the ability of evolved networks to cope with temporal dependencies between observations of the environment. The approach uses the incremental evolution technique presented in chapter 5 to scale multiple dimensions of problem difficulty. The experiments compare the performance of direct versus incremental evolution and show that all else being equal, incremental evolution allows ESP to evolve behaviors that are not accessible directly.

7.1 Background and Motivation

Artificial Life (ALife; Langton 1988) is a field that seeks to gain a deeper understanding of biological and evolutionary processes by building computational models of natural systems. Alife systems attempt not only to replicate existing organisms, behaviors, and ecosystems, but also suggest new possibilities for life that can shed light on the underlying properties of growth, development, social organization, and group behavior.

A common Alife environment is the *pursuit-evasion* scenario consisting of two entities, a predator and a prey, with conflicting objectives: the predator moves through the environment trying to capture the prey while the prey attempts to avoid capture by fleeing from the predator. For either side to be successful, its strategy must take into account the potentially changing strategy of its opponent. Pursuit and evasion contests are of interest in many areas such as adaptive behavior and robotics (Reynolds 1994a; Haynes and Sen 1995; Floreano and Nolfi 1997; Floreano et al. 1998; Strens and Moore 2002; Whiteson

et al. 2003) because they embody the kind of complex general behavior that is ubiquitous in natural ecosystems, but challenging to produce by artificial means. Pursuit and evasion also constitutes an area of game theory, differential games (Isaacs 1965), that has important implications for difficult optimal control problems like air combat (Pesch 1992; Breitner et al. 1993).

The difficulty in evolving an effective predator or prey depends on the relative strengths and weaknesses of the two adversaries. If the entity we wish to evolve suffers from a tactical disadvantage, then the task may be too hard for evolution to make progress. For example, if the predator is limited in its ability to perceive the prey, then in order to catch the prey, its behavior must consist of more than just a reactive policy. The predator will need to use some kind of short-term memory to predict the future location of the prey based not just on immediate sensory stimulus, but also on previous experience. However, since such a memory-dependent strategy is unlikely to be present in the initial random population, all of the individuals will perform too poorly to give evolution a clear indication of where to search (i.e. which individuals should be selected for recombination). Instead of gradually evolving increasingly sophisticated behaviors, "mechanical" strategies emerge (e.g. repeating a pattern of movement through the environment). Such strategies make some headway in terms of maximizing a fitness measure, but do not exhibit the kind of intelligent responsiveness required to ultimately accomplish the objective. Mechanical strategies are easy to evolve because they will often have better fitness than standing still or moving very little, and therefore, can trap the population in a locally maximal region of the solution space.

One way to solve this problem is to gradually scale the difficulty of the task over the course of evolution so that the necessary skills required to solve the goal task are more likely to be acquired. This incremental evolution approach is taken in the prey capture experiments below.

7.2 Prey Capture Experiments

The prey capture task involves a predator (i.e. the control agent) that "lives" in a bounded, square arena and whose objective is to chase down a prey that moves unpredictably within a limited amount of time (figure 7.1). The predator can detect the prey only within a limited distance. When the prey is within sensor range, the predator must move toward the prey to capture it, but when the prey moves outside the sensory range, the predator no longer receives direct sensory stimulus from the prey.

This task is easy to describe yet requires a kind of behavior that is difficult to evolve

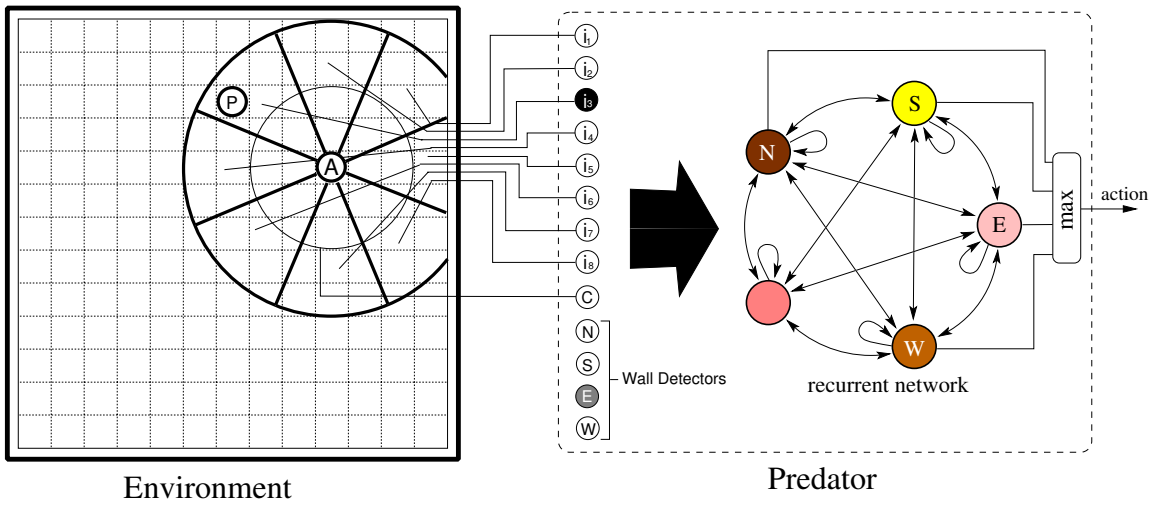


Figure 7.1: **The prey capture environment and the predator network.** The grid world (a) is occupied by the predator (agent A) and the prey (P). The sectors and circles around the predator represent its sensory array. There are 8 sectors divided into two levels of proximity. Each sector is represented by a node in the input of the neural network controlling the predator (b). An input unit i_j is activated when the prey enters the corresponding sector. If the predator brings the prey within the inner circle of the array, the C unit will also be activated. Each of the wall detector units is activated proportional to the predator's distance to the wall in that direction, provided the wall is within the sensor range. In the situation shown here, the input unit i_3 is activated but C is not, because the prey is in the far NW area. The E wall-detector unit is also activated by a small amount because the east wall is just within sensor range. This input is fed into the fully recurrent neural network along with the network activation from the previous time-step. In this case, the predator will move north because the north (N) output unit has the highest activation.

directly due to its complexity. Since the predator does not have a global view of the environment, it must remember where the prey was last detected to decide which action will bring the prey back into sensor range. This means that the networks evolved by ESP have to be recurrent. Furthermore, because the prey does not move deterministically, no amount of memory will allow the predator to predict exactly how the prey will move. The following sections describe the simulation environment, the predator representation, and the experimental setup in detail.

7.2.1 Simulation Environment

The environment consists of a square grid-world (figure 7.1a). Both the predator and the prey occupy a single grid space and can move in one of four directions {N,S,E,W} at each

time-step. The predator is considered to have captured the prey when they both occupy the same grid-cell. The prey moves probabilistically with a tendency to move away from the predator that grows stronger as the predator gets closer to it (see Appendix A for a definition of the enemy algorithm, due to Lin (1992)). The prey moves at a speed s that is set between 0 and 1. This value is the probability of the prey taking an action at each time-step. If the prey has a speed of 0.5 it will do nothing 50% of the time. Note that if the environment were continuous, a speed of 0.5 would make the task quite easy because the prey would always be moving at the same leisurely rate. In this discrete world, however, a prey moving at a speed of 0.5 is really moving at the same speed as the predator but only part of the time.

7.2.2 Control Architecture

The predator is controlled by a fully connected recurrent neural network with sigmoidal units (Figure 7.1b). At each time step each unit receives input from the input layer and from all other units. Such recurrency allows the predator to maintain temporal information that is necessary for performing the task.

As the predator moves through the environment it can detect the presence of the prey within a specified sensor range (figure 7.1a). There is one input unit (i_j) assigned to each of the 8 sectors in the sensory array. When the prey is in an area covered by the sensory array, the unit corresponding to that sector is set to 1. An additional unit (C) is set when the prey is within the closer half of the sector. The units i_1 through i_8 and C therefore afford a coarse encoding of relative prey position. The radial nature of the sensory apparatus gives greater sensitivity to prey movement at close range, where it is most crucial. Four more units are used to detect the walls in the N, S, E, and W directions. As a wall comes within sensor range the corresponding unit is activated to a degree that is proportional to the wall's distance from the predator. There is one output unit for each of the four possible actions. At each time step the predator selects the action corresponding to the unit with the highest activation. This representation provides the predator with sensory input that is both imprecise and of limited range.

7.2.3 Experimental Setup

The effectiveness of incremental evolution was tested in the prey capture task and compared to direct evolution. To determine how difficult tasks could be solved, the prey speed and short-term memory requirements of the task were varied.

The parameter settings used in the prey capture experiments are listed in figure 7.2. At each generation during evolution, 400 networks are constructed and evaluated in (M = 3) trials. A trail consists of the following: the predator is placed in the center of a 24×24 grid world and the prey is placed in a random position just within the predator's sensor range. The predator and prey alternate in taking an action each time-step until either the prey has been captured or a maximum number of time-steps (N = 60) has been reached. If the predator captures the prey, the prey is moved to a new initial position just within the sensor range, and the predator is allowed another N moves to capture the prey. This cycle repeats until the predator fails to capture the prey within N moves (the value N can thus be interpreted e.g. as the maximum time that the predator can survive without feeding). The total number of times the predator captures the prey in the M trials is used as its fitness score. Multiple trials were used to reduce evaluation noise. For a predator to receive high fitness, it must be able to catch the prey from many initial states and deal favorably with the prey's non-deterministic behavior. The task is difficult because with a sensor range of 5 there are $24^4 = 331,776$ states, but the predator only receives 612 unique observations.

Parameter	Value
Environment	
size of grid	24$\times$ 24
sensor range	5
number or trials (M)	3
number or moves (N)	60
ESP	
no. of subpops	5
size of subpops	40
mutation rate	10%

Figure 7.2: **Prey capture parameters.**

The difficulty of the task can be controlled by adjusting the value of two free parameters: the prey's speed, s, (i.e. the probability of it making a move), and the number of moves, m, the prey is allowed to make before the predator is allowed to make its first move (during these m time steps the prey moves at maximum speed, $s = 1.0$). The prey's head start guarantees that each trial will contain situations that require memory. Following the convention introduced in chapter 5, an evaluation task with a particular setting of m and s will be referred to by the notation $e_{m,s}$.

Predators were evolved both directly and incrementally to accomplish the goal task $e_{4,1.0}$, i.e. where the prey makes four initial moves before the predator is allowed to move, and then continues to move at the same speed as the predator. A predator is considered to have accomplished the task if it can capture the prey more than $M \times 100 = 300$ times in single trial.

For the Direct evolution simulations, the evaluation task remains constant throughout evolution. In other words, the networks are subjected to the goal task $e_{4,1.0}$ from the beginning. For incremental evolution the population is first evolved on the task $e_{0,0.0}$, i.e. capturing a stationary prey within its sensory range. Once this initial task has been accomplished, the best-performing network is saved and burst mutation is invoked to evolve $e_{2,0.0}$. After $e_{2,0.0}$, the number of initial steps m is further increased to 3 and 4, and then the prey speed from 0.0 to 1.0 in four steps. In other words, the incremental evolution schedule is:

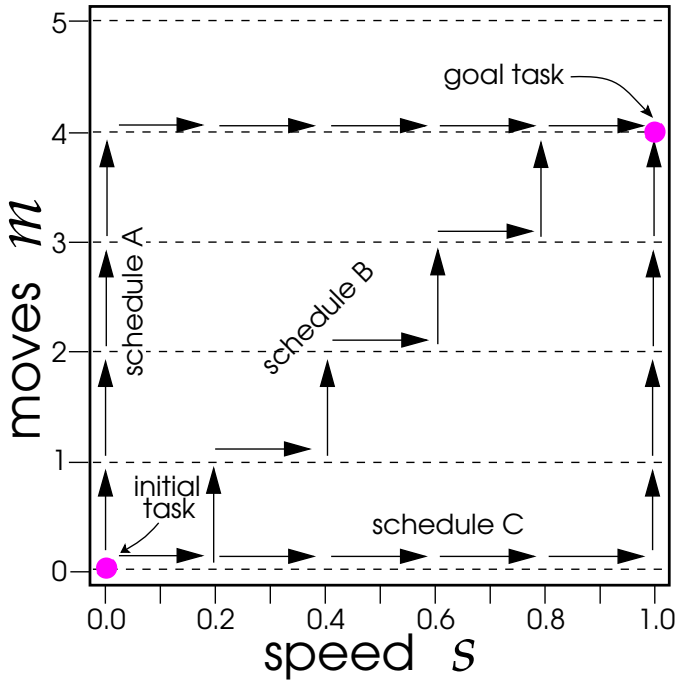


Figure 7.3: **Prey capture configuration space.** The dashed lines represent the space of evaluation tasks. Many paths from the initial evaluation task to the goal task are possible by following different schedules. Schedule A is the path used in the experiments.

$$e_{0,0.0} \longrightarrow e_{2,0.0} \longrightarrow e_{3,0.0} \longrightarrow e_{4,0.0} \longrightarrow e_{4,0.3} \longrightarrow e_{4,0.6} \longrightarrow e_{4,0.8} \longrightarrow e_{4,1.0}$$

This sequence of tasks forces the predator to first develop its short-term memory and then learn to deal with a fast moving prey. Figure 7.3, shows the configuration space for the task, and the path implied by the chosen schedule (schedule A). While other schedules such as B and C in the figure are possible, this is a natural one. To be able to pursue a prey at all, the predator first has to be able to know where it is.

7.3 Results

Figure 7.4 summarizes the prey capture results for both direct and incremental evolution. As can be seen from the figure, direct evolution (lower plot) makes little progress in solving $e_{4,1.0}$. All of the networks in the first generation perform too poorly to provide adequate selective pressure for reproduction; the environment is simply too difficult for any single

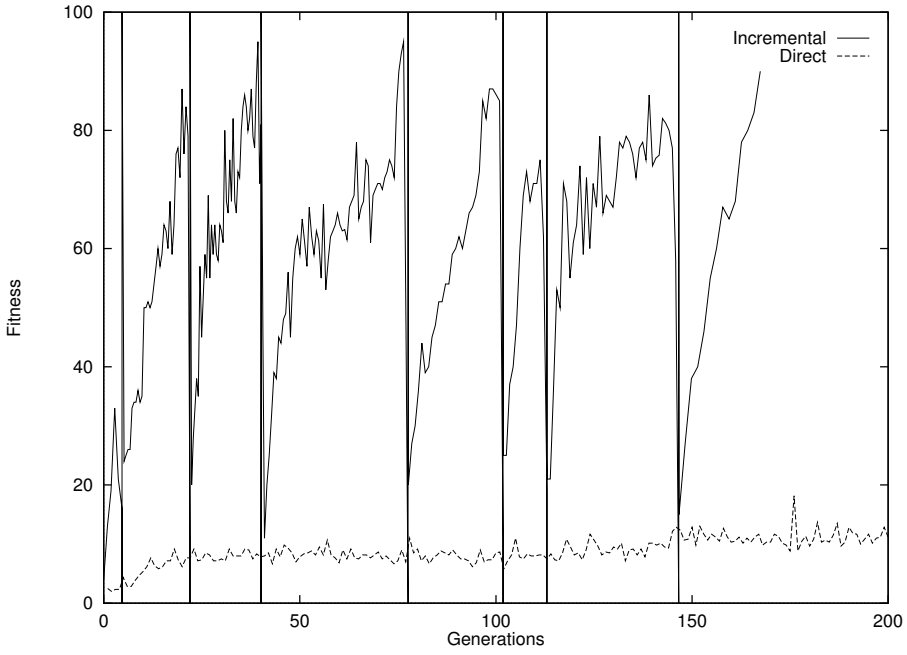


Figure 7.4: **Performance of direct and incremental evolution in the prey capture task.** The maximum fitness per generation is plotted for each of the two approaches. The direct evolution (bottom plot) makes slight progress at first but stalls after about 20 generations. The plot is an average of 10 simulations. Incremental evolution, however, proceeds through several task transitions (seen as abrupt drop-offs in the plot), and eventually solves the goal task. The incremental plot is an average of 10 simulations. Each of the simulations included a different number of generations for each evaluation task, so time was stretched or compressed for each simulation so that the transitions could be lined up.

individual to perform significantly above average. The networks improve slightly over the first 20 generations but become trapped in a region of the weight space where the sub-populations have converged before basic task skills have been acquired. The best of these individuals move around the environment a few times in a mechanical fashion. In order for an individual to perform well it must know both how to chase a fast-moving prey and how to remember its location. The likelihood of encountering an individual with such proficiency in a random population is extremely low—the direct evolution failed in every simulation.

The upper plot in figure 7.4 shows the performance of the incremental approach. In the following, the progress of this approach toward $e_{4,1.0}$ is described for each stage of the evolution schedule.

Capturing an immobile prey ($e_{0,0.0}$)

This initial stage serves to bootstrap the entire incremental evolution process with a task that can be evolved from an initial random population. When $e_{0,0.0}$ is used as the initial evaluation task, there is sufficient variation in the performance of networks to direct the genetic search. No memory is needed to accomplish $e_{0,0.0}$ so the predator only needs to concern itself with the state of its sensory array, and no pursuit is involved: in other words, the predator only needs to behave reactively.

In this easier environment, some networks are able to survive significantly longer than others. Importantly, they survive longer by performing a basic skill that is also required to solve the goal task. Some predator may do well capturing prey from the east, another from the west, while another from the north or south. Over the course of evolution the genetic recombination of these skills eventually produces a well-adapted individual that can capture the prey from all directions.

Increasing initial prey moves ($e_{0,0.0} \rightarrow e_{2,0.0} \rightarrow e_{3,0.0} \rightarrow e_{4,0.0}$)

As the number of initial prey moves m is incremented, the ability of the predator to remember the position of the prey becomes increasingly important. When the evaluation task is $e_{2,0.0}$, the prey will often move out of sensor range. However, because of its probabilistic policy, the prey will also sometimes remain within the sensor range after m moves. As m is increased, the probability of the prey moving out of sensor range, and its distance from the predator, increases.

Because situations that demand memory are introduced gradually, a predator can still capture the prey most of the time even if it does not have the ability to always remember the prey's position. If the tasks were rapidly transitioned from $e_{0,0.0}$ to $e_{4,0.0}$, a predator would have to possess a general memory right away. When $e_{4,0.0}$ has been completed, the best network can capture the prey regardless of what direction it disappeared, and how far (within 4 moves).

Increasing prey speed ($e_{4,0.0} \rightarrow e_{4,0.3} \rightarrow e_{4,0.6} \rightarrow e_{4,0.8} \rightarrow e_{4,1.0}$)

After evolving a predator that can reliably remember the prey's position and capture it, the prey is made mobile. Until now, the prey's position has been encoded in the predator's recurrent network. When the prey moves, however, the predator's sensory inputs do not match its internal representation of the situation, and it does not perform well. Initially, the prey moves only about one third of the time. At this stage it is still sometimes possible

for the prey to be caught as it is unlikely to make many moves during the time it takes the predator to capture it. Those predators that can follow the prey for even one move will have an advantage and will be selected for reproduction. Over several task transitions, the prey gradually becomes faster, and evolution favors networks that pay more attention to the current sensory input in determining the prey's location. Eventually networks emerge that can pursue and capture the prey even when it is moving at every time step, solving the goal task.

Throughout incremental evolution, therefore, the changes made in the task are small enough so that the networks formed from the previous population can occasionally perform well. This makes it possible for evolution to discriminate between good and bad genotypes, and make progress towards the goal task.

7.4 Experimental Analysis

Given that general prey capture behavior was evolved, what do the solutions look like? That is, what kind of networks resulted, and what kind of behaviors do they exhibit? This issue is examined in the following subsections.

7.4.1 Prey Capture Behavior

Figure 7.5 shows a sequence of "snapshots" that illustrate a typical prey capture scenario in the goal task. In the first frame, the predator (denoted by the letter "A") is in its initial position, and the prey ("P") has been placed in a random position just within sensor range. At this point, the predator can see the prey. Frame 2 is taken four prey moves later. The prey is now outside the predator's sensory array, and the predator has not yet moved. In Frame 3, the predator has made four moves. The first move was selected while the prey was still in the SE sector. The next three moves, however, had to rely on a recollection of where the predator last saw the prey.

As the predator approaches the prey, it may not see it for several moves as the prey begins to flee. By move 16 (Frame 4), the predator has re-acquired the prey in its sensory array, and can begin to bear down on it. Since the prey will move every time-step, the predator can only capture it by trapping it against a wall. This behavior can be seen in Frame 5: The predator pursues the prey towards the wall, where its moves are limited and it is captured (Frame 6).

Similar prey capture behavior evolved in all simulations. Although behavior is easy to describe, it involves sophisticated components: remembering the likely location of the

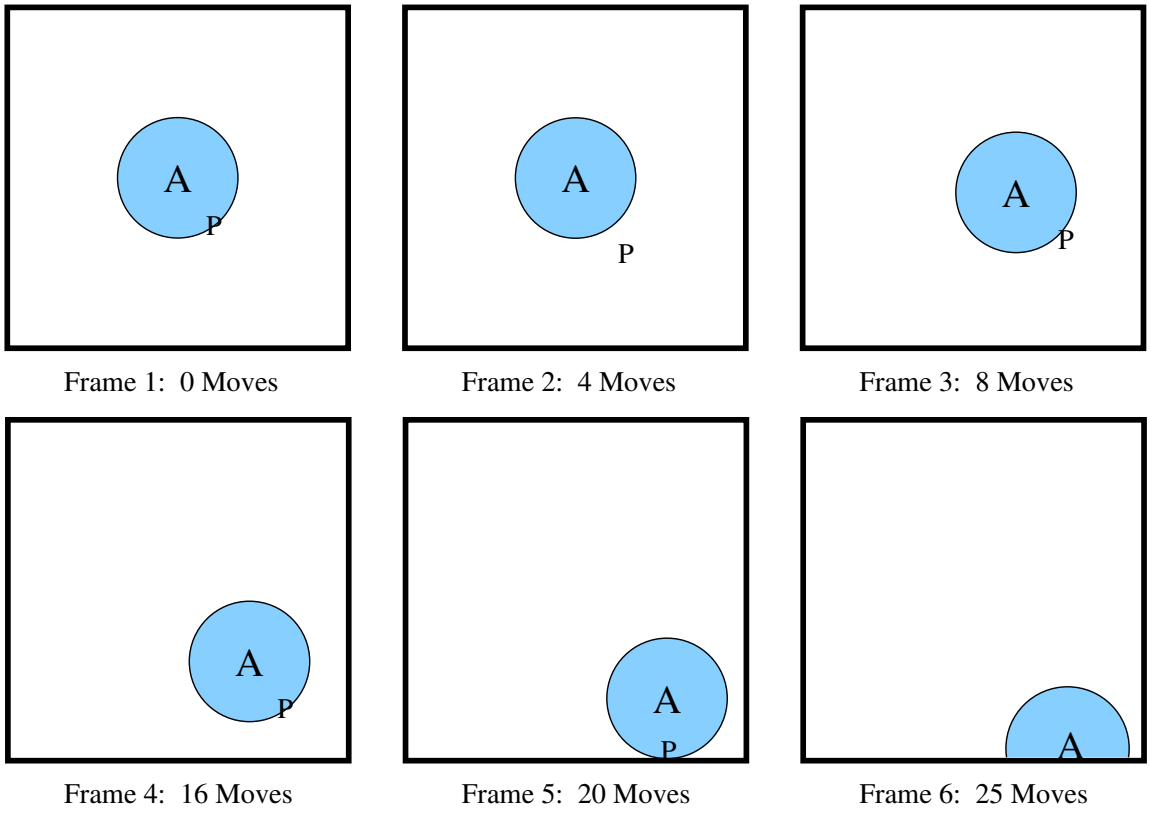


Figure 7.5: **An example of prey capture behavior.** The prey gets a head start of 4 moves and moves outside the sensory array. In 12 moves, however, the predator catches sight of it again, relying on its memory of where it last saw the prey. Eventually the predator pins the prey down against the wall and captures it. Similar scenarios occur from virtually all initial states, although the individual moves vary due to the stochastic nature of the prey.

prey for several time steps, driving the prey towards a wall, and capturing it against the wall. What is most important, though, is that the successful predators can perform this strategy from all different initial states, and with a prey that behaves stochastically. In this sense, the predators display believable and complex general behavior.

7.4.2 Network Analysis

What do the successful prey capture networks look like, that is, how do the different specializations contribute to and interact in prey capture? One way to analyze the contributions of individual neurons is to perform a lesion study: remove one of the neurons from the network and observe the effects on the network's behavior. The prey capture networks,

Lesioned Neuron	1	2	3	4	5
% Network Performance	63.2	21.9	47.7	48.1	25.1

Table 7.1: **Prey capture performance of a lesioned network.** One of the successful networks was systematically lesioned by removing the input weights of each of its neurons in turn. The lesioned network was tested in the prey capture task and its performance was compared to that of the original network. For example, when the first neuron of this network was lesioned, it was still able to capture the prey 63.2% as many times as the complete network in a single trial. The results are averages over 100 trials. Similar results were obtained for all networks tested.

however, are fully recurrent, and 4 out of the 5 units also serve as output units. Such units cannot be completely removed from the network. Removing for example the "north" output unit would only have the obvious and uninteresting effect of preventing the predator from moving north. Instead, a unit can be lesioned by disabling only its input connections (i.e. the connection from the sensory array), while still allowing it to receive recurrent signals from the other neurons. The functional role of the lesioned neuron may then be inferred by observing the behavior of the damaged network in prey capture.

The main result of the lesion study is that the networks are quite robust (table 7.1). When a any single neuron is lesioned, the behavior does not completely break down: the predator's tendency to pursue the prey is preserved to large extent, and it is still able to perform significant prey capture. When two neurons are lesioned simultaneously there is a corresponding double degradation in performance (varying between 38.1% for neurons 1 and 4 and 5% for 2 and 5).

It is difficult to attribute a particular behavior to any particular neuron. The coding of behavior seems to be distributed across the network. These results are in line with those of feedforward SANE networks for controlling a mobile robot (Moriarty 1997), where elementary behaviors such as advancing and turning and stopping in front of obstacles were also found to be distributed across multiple units. Recurrency apparently makes the behaviors even more distributed. Very few of the recurrent weights of a successful network are close to zero, which means that each neuron modulates the behavior of all other neurons. As a result, the functions are distributed across the whole network, and the system is very robust against degradations such as lesions, noise, and inaccurate weights values.

7.5 Discussion

Although the prey capture task takes place in discrete rather that continuous world, it is still interesting and relevant due of the predator's limited perception. Evolving controllers

that can operate in situations where sensory information may be interrupted is important in many real-time tasks. For example, consider a collision detection system for a car. If the sensors are temporarily obstructed or corrupted by say heavy rain or snow, a system that possesses short term memory will still be able to predict the location of objects that are no longer visible, and continue to make good decisions until sensory input is restored. In general, recurrent networks can make control systems more robust to temporary faults and noise by allowing actions to depend on more than just the immediate sensory input.

Chapter 8

Dynamic Resource Allocation for a Chip Multiprocessor

As computer chip design moves to architectures with more and more CPUs on a single chip, on-board controllers will be required to manage the various resources shared among the processors. In particular, to make efficient use of the memory cache and maximize the performance of the chip, some mechanism will be needed to assign cache banks dynamically to the processors according to their individual memory requirements.

In this chapter, ESP is used to evolve such a controller. Like the prey capture task in the previous chapter, the task requires the use of memory to predict future states of the environment. However, this task represents a significant scale-up in complexity due to the high-dimensionality of the state space and interdependence of the state variables. The experiments compare the performance of the evolved controllers to a static assignment of the cache resources, and show a significant improvement the instruction throughput of the chip over a broad range of operating conditions.

8.1 Background and Motivation

For decades, the performance of single processor chips has improved at a rate of 50 to 60% per year through an increase in both the clock rate and the number of transistors on the chip. More transistors means that more *instruction level parallelism* (ILP) can be exploited to increase performance by executing multiple, non-dependent instructions simultaneously. Unfortunately, this trend cannot continue much longer. As transistors get smaller and faster, the wires that connect them are becoming much slower, thereby limiting the number of transistors that can be reached in a clock cycle (Agarwal et al. 2000). Furthermore, since

the amount of ILP in any given instruction stream is finite, there is a diminishing return in performance with each increase in transistor count.

In order to sustain performance growth at its historical level, chip designers will have to focus less on ILP and more on parallelism at the *thread* or even *process* level. New microarchitectures will be partitioned into independent physical regions each containing its own CPU core and memory cache (Hammond et al. 1997; Sohi et al. 1998). Each region will occupy only a small portion of the entire chip so that wire lengths are shortened along critical paths allowing the design of each core to be simplified and optimized for speed. While sacrificing some ILP, these chip multiprocessors, or CMPs, can execute multiple independent instruction streams simultaneously. As parallelizing compilers improve and become more widespread, and parallel programming techniques become more accepted, CMPs promise to scale chip performance well into the future.

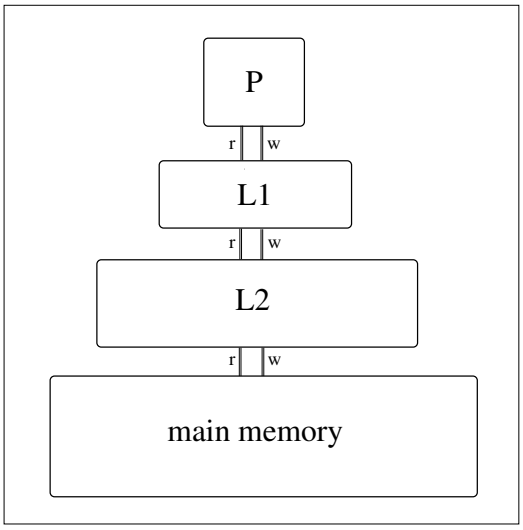


Figure 8.1: **Memory hierarchy.** The diagram shows a simplified view of the typical computer memory hierarchy. The processor (P) has read/write channels to access data in the caches (L1,L2) and in main memory. A data access involves checking each level of the hierarchy from the L1 down, until the data is found.

Commercial CMPs have already started to emerge including the IBM Power4 processor (Diefendorff 1999), which has two processing cores per die, the eight-core Compaq Piranha (Barroso et al. 2000), and many other designs are currently being studied in academia, e.g. Stanford's Hydra (Hammond et al. 2000) and RAPTOR (Lee et al. 1999) at Korea University.

An open question is how the memory hierarchy of CMPs will be designed as the number of cores on a chip increases from two to eventually hundreds. All modern computer architectures use memory hierarchies to minimize the frequency with which data accesses to main memory occur. The typical configuration consists of two levels of cache, level-one (L1) and level-two (L2), that intervene between the CPU and main memory (figure 8.1). Whenever the CPU needs to access a piece of data, each level of the hierarchy is checked in succession until the data is found or "hit." The lower the level of the hit in the hierarchy, the longer the delay, and potentially the longer the CPU has to wait to resume execution.

In the ideal case, all of the data required during the execution of an instruction

stream (i.e. a program) is loaded into the cache once, and the CPU never has to wait for data from main memory. However, since the exact memory access behavior cannot be known in advance and the cache space is finite, cache misses are inevitable. In order to optimize the memory design for a single processor machine, the L1 needs to be relatively small so that it is fast, and the L2 must be large so that fewer misses occur, but not so large that it is slow and significantly increases the penalty for a miss in the L1.

When there are multiple processors on the same chip, designing the memory hierarchy is potentially more complicated because some levels may be shared. It is likely that each core in future CMPs will have a private L1 cache, which will be small and tightly coupled to its processing core. The L2 caches, however, will consume much of the die to reduce the frequency with which processors must go off the chip for data. These L2 caches will total tens (and eventually hundreds) of megabytes in size, and will be divided into hundreds of physical banks.

In the simplest configuration, each core is assigned its dedicated memory resources (i.e. some number of L2 cache banks) at design time. However, such a static assignment is suboptimal, as different workloads have different memory requirements that vary over time. For instance, if job A is only using a small fraction of its L2 cache, and job B is memory bound, then performance would be improved by dynamically assigning some of the cache banks from job A's processor to job B.

A more flexible and potentially powerful solution is to allow the partitioning of the L2 resources to be determined by an on-board controller that dynamically allocates cache banks to cores. The controller would be responsible for managing the resources adaptively in response to the changing needs of the jobs running on the individual cores. For such a controller to be practical it must have an efficient implementation so that decisions can be made within the tight time constraints imposed by the operation of the chip. A sufficiently low-overhead implementation could significantly improve the performance of the CMP by allocating resources to where they contribute most to maximizing some desired measure of overall chip performance, e.g. the number of instructions executed per clock cycle (IPC). In the experiments below, ESP is used to evolve a recurrent neural network controller to manage the L2 cache resources. The next section describes the task in more detail and examines the challenges inherent in this problem.

8.2 Design Challenges

Figure 8.2 gives a high level view the control scheme for the hypothetical CMP used in this study. The CMP has four processing cores surrounded by a number of L2 cache banks. At

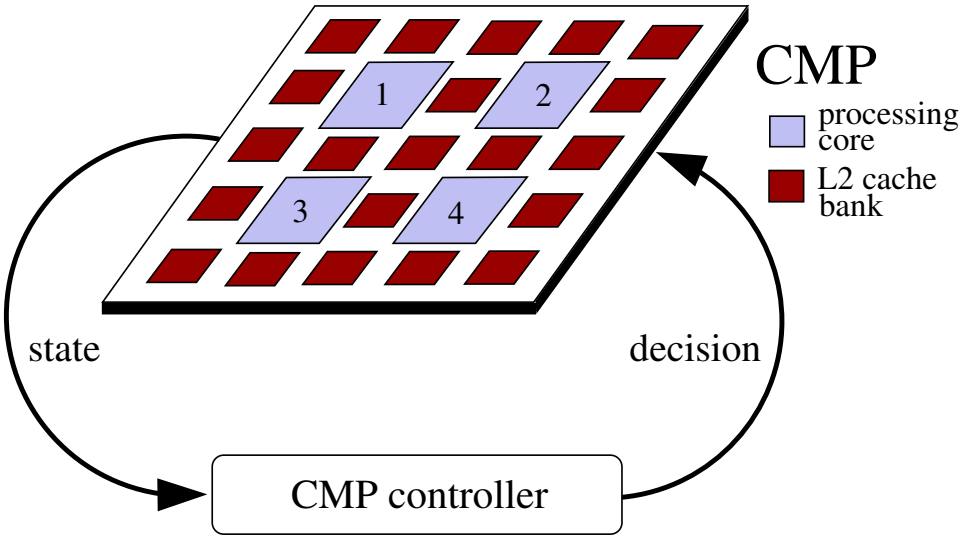


Figure 8.2: **Controlling a Chip Multiprocessor.** The CMP consists of a number of cores (in this case four) surrounded by L2 cache banks. Each core has its own private L1 cache situated at close proximity (not shown). The controller receives measurements from the chip at regular intervals and outputs a decision to optimize some measure of performance (e.g. IPC) by assigning each L2 cache bank to a core.

regular intervals, the controller receives information about the state of the CMP, and uses this information to assign a number of cache banks to each core.

The following three properties of the CMP make *designing* such a controller difficult.

1. High dimensionality: with n cores the input and output dimensionality of the controller is $O(n)$, and future CMPs are expected to contain hundreds of cores.
2. Non-Markov task: making effective decisions requires more information than just the current state of the chip.
3. Highly variable operating conditions. The controller must optimize performance for all combinations of possible jobs and their characteristic memory access behaviors.

Together these three properties make it impossible to accurately predict the behavior of the CMP in a timely manner. In general it is not known in advance what affect an action will have on future states of the CMP. Therefore, the problem can be viewed as one of delayed reward where the relative merit of a particular sequence of cache allocation decisions is determined by the performance of the CMP over the long term with respect to some cost function—a reinforcement learning problem.

As we have seen in the comparisons of chapter 4, conventional reinforcement learning methods are not suited to this kind of high-dimensional, non-Markov environment. In contrast, ESP can cope with such environments. Moreover, using ESP guarantees the resulting controller will be efficient since it can be realized in hardware as a dedicated parallel processor. The more important issue for neuroevolution is property 3: the CMP is expected to operate effectively over the entire range of likely workloads. Any practical evaluation regime can only sample a small subset of these conditions, and therefore the fitness measurement could be potentially very noisy.

8.3 CMP Controller Experiments

8.3.1 Simulation Environment

Controllers were evolved in an approximation to the CMP environment that relies on traces collected from the SimpleScalar processor simulator (Burger and Austin 1997). A trace is a sequence of measurements of processor variables sampled at fixed intervals, and is a common way to capture various characteristics of processor behavior (Prete et al. 1995). A set of traces was generated for each of the following SPEC2000 benchmarks: `art`, `equake`, `gcc`, `gzip`, `parser`, `perlbmk`, and `vpr`, using their respective reference working-sets. Each benchmark's trace set consists of one trace for each possible L2 cache size $s \in S = \{64K, 128K, 256K,$
$\ldots, L2_{tot}\}$, for a total of $7 \times |S|$ traces. The parameter $L2_{tot}\}$ is the total amount of L2 cache available on the CMP. For convenience, traces are identified by the naming scheme: <benchmark name><cache size>. For example, `gcc256` is the trace for the `gcc` benchmark for a processor with 256K of L2 cache. All traces recorded instructions per cycle (IPC), L1 cache miss rate (L1m), and L2 cache miss rate (L2m) of the simulated processor every 10,000 instructions using the DEC Alpha 21264 processor configuration. These three variables constitute the state of the chip that is observable to the controller, and were chosen intuitively to be the measurements most relevant to L2 cache resizing.

The traces provide a substitute for the actual CMP for which a full simulator is not currently available. By combining n traces, a CMP with n processing cores can be approximated. Taking the recorded values (IPC,L1m,L2m) from the k-th entry in each of the n traces gives the state that the CMP would be in after $10,000 \times k$ instructions have been executed on each of the cores.

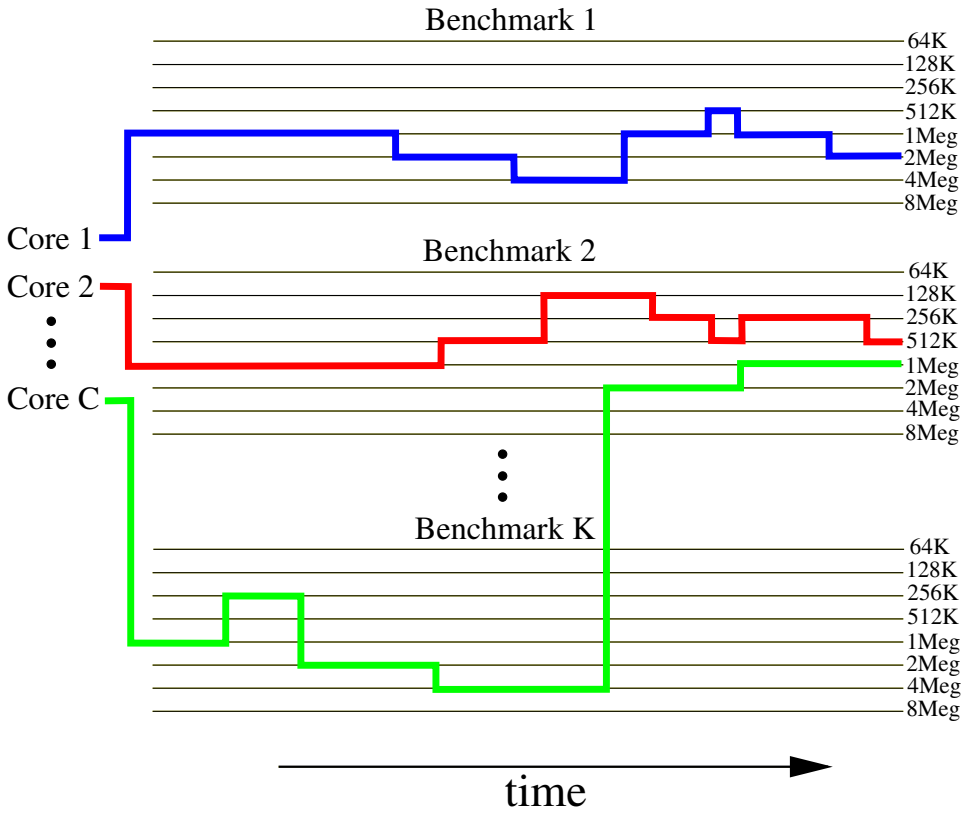


Figure 8.3: **Trace environment.** The CMP is approximated by several sets of traces, one for each benchmark. There are as many traces active during a network evaluation as there are cores. There is one trace in each set for each of the cache sizes. When the controller changes a core's cache size the trace environment switches over to the appropriate trace.

8.3.2 Control Architecture

Figure 8.4 shows the representation used in the experiments. The input layer receives the IPC, L1m, and L2m of each core $\{c_i\}_{i=1}^{C}$, for a total of $3C$ input units. Because the networks are recurrent, each neuron also receives the hidden layer activation from the previous time-step. There is one output unit per core, and all hidden and output units are sigmoidal.

At each decision point, the network outputs a vector $\mathbf{o} \in \Re^C$ that is normalized and quantized to produce a vector $\mathbf{u} \in \Re^C$ of cache size assignments

$$u_i = f\left(\frac{o_i}{\sum_j^C o_j}\right), \qquad f : [0,1] \rightarrow S \tag{8.1}$$

where f is the quantizing function and $\sum u_i \leq L2_{tot}$. This post-processing ensures that

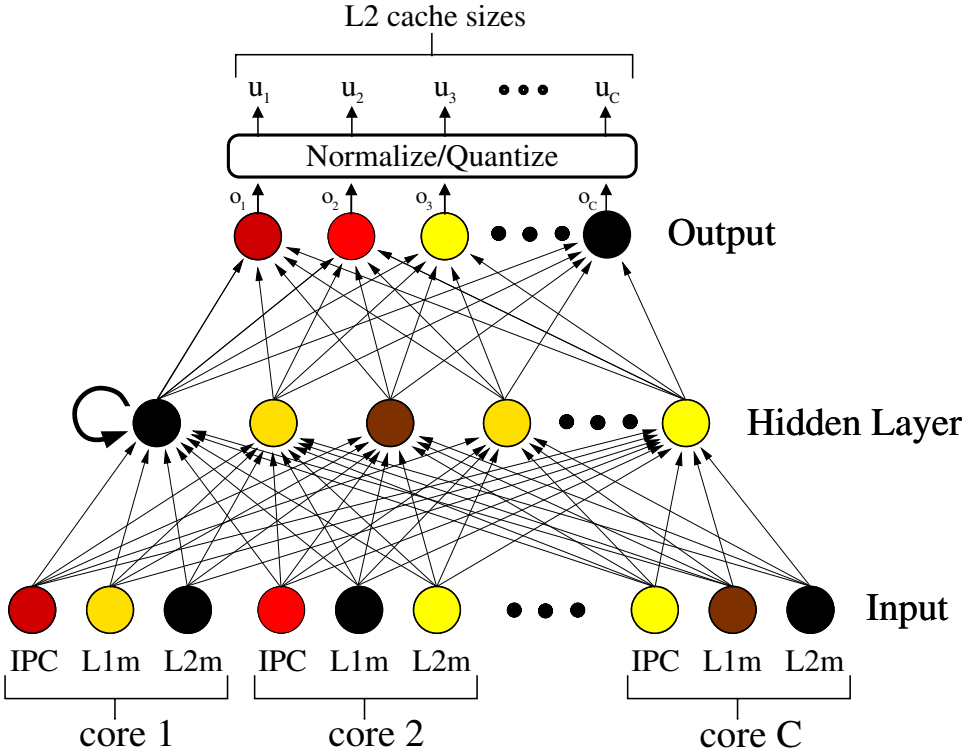


Figure 8.4: **CMP control network.** The network has a set of input units for each core, one unit for each of the three performance measurements (IPC,L1m,L2m). There is one output unit per core. The activation of the output unit for core i, o_i, (indicated by gray-scale coding) is normalized and quantized to produce a cache size request, u_i, for that core.

the total amount of cache requested never exceeds the total cache available, $L2_{tot}$, and partitions each output unit's continuous output into $|S|$ levels, i.e. the predefined number of possible cache sizes.

8.3.3 Experimental Setup

Controllers were evaluated by having them interact with the trace-based environment for some fixed number of control decisions. At the beginning of a network evaluation the environment is initialized by selecting a set of C benchmarks at random, and allocating an equal amount of L2 cache to each core ($L2_{tot}/C$). Once initialized, the network starts controlling the CMP by receiving the state of the chip at time t from the traces corresponding to caches of size $L2_{tot}/C$. The network then outputs its cache allocation decision which affects the configuration of the chip from t until the next decision point at time $t+1$, 10,000

instructions later. The next state at $t + 1$ then becomes the new input to the controller and the cycle is repeated.

In a real CMP, the reassignment of a cache bank from core A to core B would cause the entire caches of A and B to be unavailable for a significant number of cycles while their data is being written back to memory. In these experiments, this overhead is ignored and the chip is simply reconfigured by switching to the trace corresponding to the new cache size (figure 8.3). So, for instance, if the controller decides that core c_1, which is currently executing the `gzip` benchmark with a 256K cache, should have 512K, then the trace for c_1 will switch from $gcc256$ to $gcc512$, and the controller will receive values from $gcc512$ at the next decision point. The new trace is started at the same point as the old one (i.e. the same number of instructions into the computation). When a trace runs out, the environment switches to the trace of a different, randomly selected benchmark at the same cache size (see trajectory of core C in figure 8.3). The evaluation ends after some predefined number of cycles.

Parameter	Value
Environment	
no. of cores (C)	4
total L2 cache ($L2_{tot}$)	4M
caches sizes ($\|S\|$)	7 (64k..4M)
ESP	
no. of subpops	10
size of subpops	100
burst criteria	20
mutation rate	30%

Figure 8.5: **Task parameters.** The values for ESP are a compromise between performance (i.e. the quality of the solution) and the CPU time required for the simulation. Larger values produced similar results with a linear increase in CPU time.

All of the experiments presented here were conducted using the parameter settings in figure 8.5. With 7 possible cache sizes available to each core and 7 benchmarks, a total of 49 traces were used to implement the environment. Each network was evaluated for 1 billion instructions (i.e. 100,000 decisions). The fitness of a network was the average IPC of the chip averaged over the duration of the trial.

Although using this trace-based approach simplifies the CMP environment somewhat, it provides a good first approximation with which to evaluate the feasibility of actually applying ESP to a full-scale version of the simulator, once it becomes available. Furthermore, if the CMP is reconfigured using a much more efficient implementation where the number of cache banks assigned to a core is changed by increasing or decreasing the associativity of its total cache, then the trace-based model more closely approximates the true behavior of the chip.

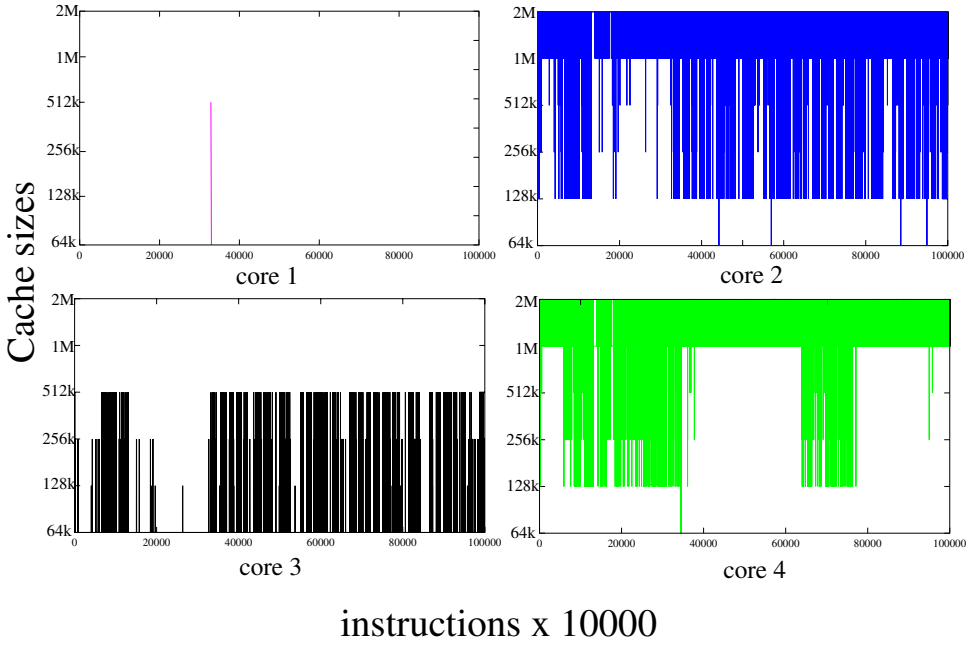


Figure 8.6: **Control behavior.** Each plot shows the cache size of each of the 4 cores executing one the benchmarks for 1 billion instructions. (expand, more analysis)

8.4 Results

Five simulations were run on a 14-processor Sun Ultra Enterprise 5500 for approximately 1000 generations each. At the end of each simulation the fitness of the best network was compared to a baseline performance value that is the average IPC of the chip obtained when the total amount of L2 cache on the chip is divided equally among the cores. The networks showed an average improvement of 16% over the baseline.

However, since this result is measured from a single trial, it only indicates the networks performance under the specific conditions experienced during that trial. To measure how well the networks perform the general task despite their limited exposure to the environment, the best network from each of the 5 simulations was submitted to a generalization test. The test consists of 1000 trials where the network controls the chip for 1 billion instructions under random initial conditions. In each trial, the baseline performance was also measured. Once all the trials were completed, the network performance was compared to the baseline across all trials. The result of this test showed that the networks still retained a 13% average performance advantage over the baseline, and, perhaps more importantly, the networks performed better than the baseline on every trial. These tests show that although the networks had very limited exposure to the task during evolution, they were able

to extract general competence to perform well under novel circumstances.

Figure 8.6 shows the behavior of one of the best networks over the course of 100,000 decisions. It is clear that the different cores that are running different benchmarks are being managed differently. Core c_1 stays almost entirely at $64K$ while the others oscillate rapidly within characteristic ranges. This oscillation is an artifact of not imposing an overhead on cache re-sizing.

8.5 Discussion

The experiments in this chapter represent a first step toward solving the complex resource management problem that will be critical as large-scale chip multiprocessors become widespread. The results indicate that ESP could potentially provide an effective mechanism for developing a CMP cache resource manager.

The trace-based model currently ignores the following characteristics of the CMP microarchitecture: (1) As mentioned in the previous section, since there is no penalty associated with reconfiguring the chip, the controller may change the size of the caches more frequently than would be optimal for the actual CMP, (2) it treats all cache banks as if they are equidistant from each of the cores without accounting for the variability in cache access latencies that exist due the physical layout of the chip—some banks are necessarily further from a core and require more cycles to access, and (3) to reduce the number of traces in the model, the cache sizes available to each core grow in powers of 2 ($2^n \times 64K$), instead of linearly ($n \times 64K$). This limits control to a relatively coarse partitioning of the cache.

A model that incorporates these complexities will force ESP to evolve controllers that are more reserved in their resource management regime, favor cache banks that reside at close proximity to the cores, and are able to control resources at a finer granularity. It is important to note that unlike physical dynamical systems, a CMP simulator can be built which perfectly replicates the behavior of the actual CMP (Pai et al. 1997; Ikodinovic et al. 1999). Therefore, transfer in this task is not an issue. However, characterizing and simulating realistic workloads will still be important to ensure that controllers are evaluated under conditions that are representative of what can be expected in the real world.

Chapter 9

Active Guidance of a Finless Rocket

Finless rockets are more efficient than conventional finned designs, but are too unstable to fly unassisted. These rockets require an active guidance system to control their orientation during flight and keep them from tumbling. However, because the dynamics of these rockets are highly non-linear, designing such a guidance system can be prohibitively costly.

In this chapter, ESP is used to evolve an active guidance system for a finless version of Interorbital Systems RSX-2 rocket. This is the most challenging control problem attempted in this dissertation. Not only is the environment continous, high-dimensional, and non-linear, but the difficulty of the task changes during the course of each evaluation. In contrast to the optimization task in the previous chapter... The experimental results show that the evolved guidance system can increase the final altitude of the finless rocket far beyond that of the unguided full-finned version.

Figure 9.1: **The Interorbital Systems RSX-2 rocket.** The RSX-2 is capable of lifting a 5 pound payload into the upper atmosphere using four liquid-fueled thrusters. It is the only liquid-fueled sounding rocket in production. Such rockets are desirable because of their low acceleration rates and non-corrosive exhaust products.

9.1 Background and Motivation

Sounding rockets carry a payload for making scientific measurements to the Earth's upper atmosphere, and then return the payload to the ground by parachute. These rockets serve an invaluable role in many areas of scientific research including high-G-force testing, meteorology, radio-astronomy, environmental sampling, and micro-gravity experimentation (Corliss 1971; Seibert 2001). They have been used for more than 40 years; they were instrumental e.g., in discovering the first evidence of X-ray sources outside the solar system (Giacconi et al. 1962). Today, sounding rockets are much in demand as the most cost-effective platform for conducting experiments in the upper atmosphere.

Figure 9.1 shows one such sounding rocket, the Interorbital Systems RSX-2. The RSX-2 is a low-cost, state of the art design that uses four liquid-fueled engines. Like all sounding rockets and most rockets in general, the RSX-2 is equipped with fins to keep the rocket on a relatively straight path and maintain stability. While fins are an effective "passive" guidance system, they increase both mass and drag on the rocket which lowers the final altitude or *apogee* that can be reached with a given amount of fuel. A rocket with smaller fins or no fins at all can potentially fly much higher than a full-finned version. Unfortunately, such a design is unstable, requiring some kind of *active* attitude control or guidance to keep the rocket from tumbling. In the case of the RSX-2, active guidance could be implemented by controlling the amount of thrust from each of the four engines.

Finless designs have been used for decades in expensive, large-scale launch vehicles such as the USAF Titan family, the Russian Proton, and the Japanese H-IIA. The guidance systems on these rockets are based on classical feedback control such as Proportional-Integral-Differential (PID) methods to adjust the thrust angle (i.e. thrust vectoring) of the engines. Because rocket flight dynamics are highly non-linear, engineers must make simplifying assumptions in order to apply these linear methods, and take great care to ensure that these assumptions are not violated during operation. Such an undertaking requires detailed knowledge of the rocket's dynamics that can be very costly to acquire. Recently, non-linear approaches such as neural networks have been explored primarily for use in guided missiles (see (Liang Lin and Wen Su 2000) for an overview of neural network control architectures). Neural networks can make control greatly more accurate and robust, but, unfortunately, still require significant domain knowledge to train.

For these reasons, the cost of developing finless versions of small-scale rockets, like the RSX-2, has been prohibitive. However, using ESP it may be possible to solve the guidance problem without the need for analytical modeling of the rocket's dynamics or prior knowledge of the appropriate kind of control strategy to employ. All that is required

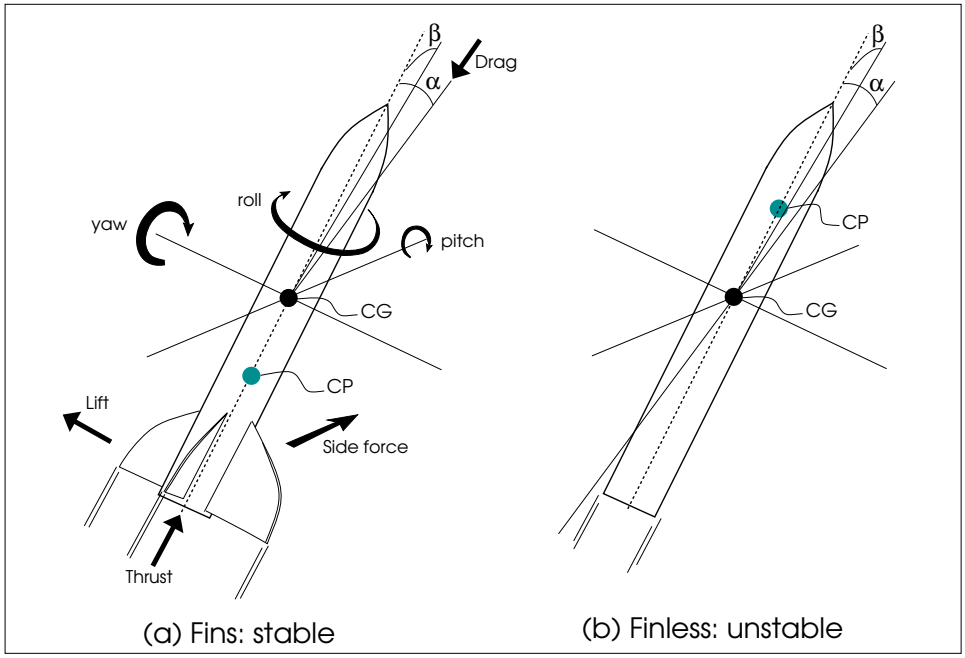


Figure 9.2: **Rocket dynamics.** The rocket (a) is stable because the fins increase drag in the rear of the rocket moving the center of pressure (CP) behind the center of gravity (CG). As a result, any small angles α and β are automatically corrected. In contrast, the finless rocket (b) is unstable because the CP stays well ahead of the CG. To keep α and β from increasing, i.e to keep the rocket from tumbling, active guidance is needed to counteract the destabilizing torque produced by drag.

is a sufficiently accurate simulator and a quantitative measure of the guidance system's performance, i.e. a fitness function.

9.2 Stabilizing the Finless RSX-2 Rocket

The rocket guidance domain is similar to pole-balancing in that both involve stabilizing an inherently unstable system. Figure 9.2 gives a basic overview of rocket dynamics. The motion of a rocket is defined by the translation of its center of gravity (CG), and the rotation of the body about the CG in the pitch, yaw, and roll axes. Four forces act upon a rocket in flight: (1) the thrust of the engines which propel the rocket, (2) the drag of the atmosphere exerted at the *center of pressure* (CP) in roughly the opposite direction to the thrust, (3) the *lift force* generated by the fins along the yaw axis, and (4) the *side force* generated by the fins along the pitch axis. The angle between the direction the rocket is flying and the longitudinal axis of the rocket in the yaw-roll plane is known as the *angle of attack* or α, the corresponding angle in the pitch-roll plane is known as the *sideslip angle* or β. When either

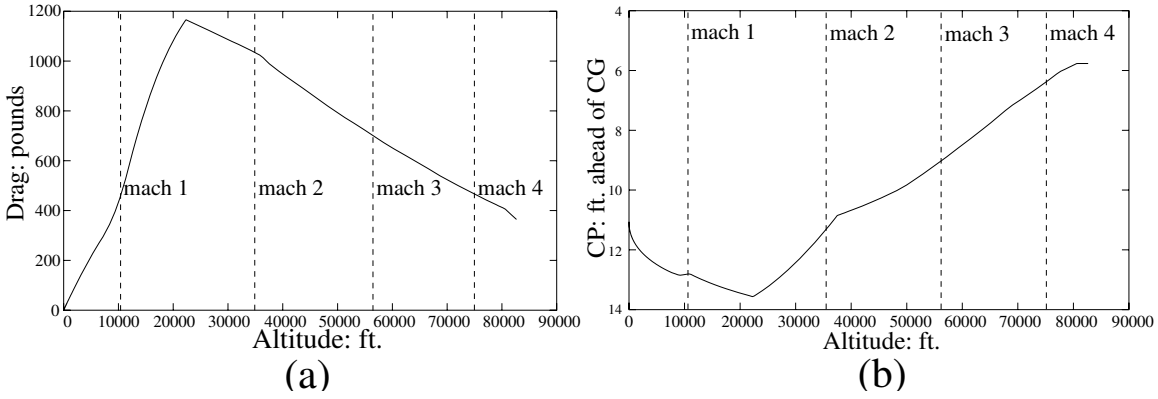


Figure 9.3: **The time-varying difficulty of the guidance task.** Plot (a) shows the amount of drag force acting on the finless rocket as it ascends through the atmosphere. More drag means that it takes more differential thrust to control the rocket. Plot (b) shows the position of the center of pressure in terms of how many feet ahead it is of the center of gravity. From 0 to about 22,000ft the control task becomes more difficult due to the rapid increase in drag and the movement of the CG away from the nose of the rocket. At approximately 22,000ft, drag peaks and begins a decline as the air gets thinner, and the CP starts a steady migration towards the CG. As it ascends further, the rocket becomes progressively easier to control as the density of the atmosphere decreases.

α or β is greater than 0 degrees the drag exerts a torque on the rocket that can cause the rocket to tumble if it is not stable. The arm through which this torque acts is the distance between the CP and the CG.

In figure 9.2a, the finned rocket is stable because the CP is behind the rocket's CG. When α (β) is non-zero, a torque is generated by the lift (side) force of the fins that counteracts the drag torque, and tends to minimize α (β). This situation corresponds to a pendulum hanging down from its hinge; the pendulum will return to this stable equilibrium point if it is disturbed. When the rocket does not have fins, as in figure 9.2b, the CP is ahead of the CG causing the rocket to be unstable. A non-zero α or β will tend to grow causing the rocket to eventually tumble. This situation corresponds to a pendulum at its unstable upright equilibrium point where any disturbance will cause it to diverge away from this state.

Although the rocket domain is similar to the inverted pendulum, the rocket guidance problem is significantly more difficult for two reasons: the interactions between the rocket and the atmosphere are highly non-linear and complex, and the rocket's behavior continuously changes throughout the course of a flight due to system variables that are either not under control or not directly observable (e.g. air density, fuel load, drag, etc.).

Figure 9.3 shows how the difficulty of stabilization varies over the course of a suc-

cessful flight for the finless rocket. In figure 9.3a, drag is plotted against altitude. From 0ft to about 22,000ft, the rocket approaches the sound barrier (Mach 1) and drag rises sharply. This drag increases the torque exerted on the rocket in the yaw and pitch axes for a given α and β, making it more difficult to control its attitude. In figure 9.3b, we see that also during this period the distance between the CG and CP increases because the consumption of fuel causes the CG to move back, making the rocket increasingly unstable. After 22,000ft, drag starts to decrease as the air becomes less dense, and the CP steadily migrates back towards the CG, so that the rocket becomes easier to control.

For ESP, this means that the fitness function automatically scales the difficulty of the task in response to the performance of the population. At the beginning of evolution the task is relatively easy. As the population improves and individuals are able to control the rocket to higher altitudes, the task becomes progressively harder. Although figure 9.3 indicates that above 22,000ft the task again becomes easier, progress in evolution continues to be difficult because the controller is constantly entering an unfamiliar part of the state space. A fitness function that gradually increases in difficulty is usually desirable because it allows for sufficient selective pressure at the beginning of evolution to direct the search into a favorable region of the solution space. However, the rocket control task is already too hard in the beginning—all members of the initial population perform so poorly that the evolution stalls and converges to a local maxima. In other words, direct evolution does not even get started on this very challenging task. Therefore, the controller must be evolved incrementally, first using a more stable finned version of the rocket, and then transitioning to the goal task of stabilizing the finless version.

The following section describes the simulation environment used to evolve the controller, the details of how a guidance controller interacts with the simulator, and the experimental setup for evolving a neural network controller for this task.

9.3 Rocket Control Experiments

9.3.1 Simulation Environment

The sounding rocket environment was simulated using the JSBSim Flight Dynamics Model[1] adapted for the RSX-2 rocket by Eric Gullichsen of Interorbital Systems. JSBSim is an open source, object-oriented flight dynamics simulator with the ability to specify a flight control system of any complexity. JSBSim provides a realistic simulation of the complex

[1]More information about the free JSBSim software package is available at: `http://jsbsim.sourceforge.net/`

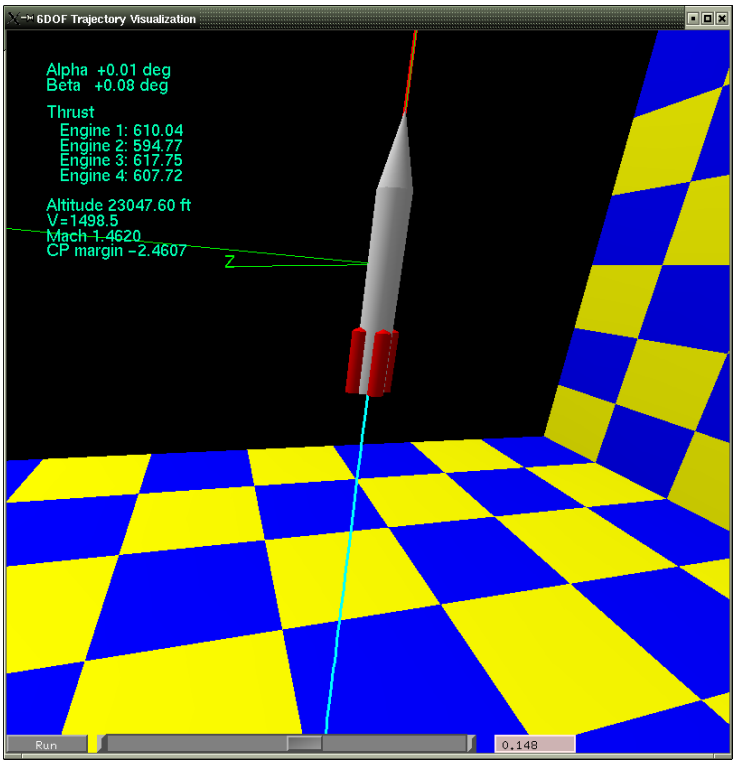


Figure 9.4: **RSX-2 rocket simulator.** The picture shows a 3D visualization of the JSBSim rocket simulation used to evolve the RSX-2 guidance controllers. The simulator provides a realistic environment for designing and verifying aircraft dynamics and guidance systems.

dynamic interaction between the airframe, propulsion system, fuel tanks, atmosphere, and flight controls. The aerodynamic forces and moments on the rocket were calculated using a detailed geometric model of the RSX-2.

Four versions of the rocket with different fin configurations were used: full fins, half fins (smaller fins), quarter fins (smaller still), and no fins, i.e. the actual finless rocket (figure 9.5). This set of rockets allows the behavior of the RSX2 to be observed at different levels of instability, and provides a sequence of increasingly difficult evaluation tasks with which to evolve incrementally. All simulations used Adams-Bashforth 4th-order integration with a time step of 0.0025 seconds.

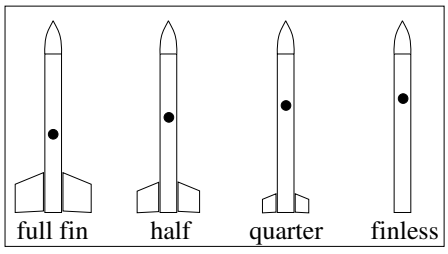


Figure 9.5: **Fin configurations.** Four versions of the RSX-2 were used in the experiments, each with a different fin size and CG location (black circles).

9.3.2 Control Architecture

The rocket controller is represented by a feedforward neural network with one hidden layer (figure 9.6). Every 0.05 seconds (i.e. the control time-step) the controller receives a vector of readings from the rocket's on-board sensors that provide information about the current orientation (pitch, yaw, roll), rate of orientation change, angle of attack α, sideslip angle β, current throttle position of the four thrusters, altitude, and velocity in the direction of flight. This input vector is propagated through the sigmoidal hidden and output units of the network to produce a new throttle position for each engine determined by:

$$u_i = 1.0 - o_i/\delta,\ i = 1..4 \qquad (9.1)$$

where u_i is the throttle position of thruster i, o_i is the value of network output unit i, $0 \leq u_i, o_i \leq 1$, and $\delta \geq 1.0$. A value of ω for u_i means that the controller wants thruster i to generate $\omega \times 100\%$ of maximum thrust. The parameter δ controls how far the controller is permitted to "throttle back" an engine from 100% thrust.

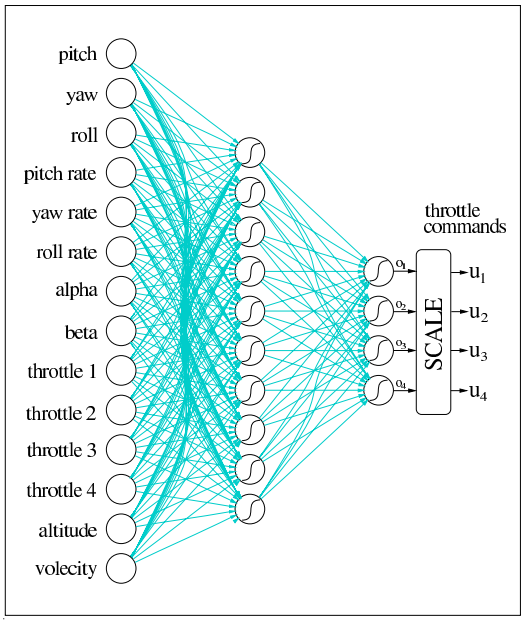


Figure 9.6: **Neural network guidance.** The control network receives the state of the rocket every time step through its input layer. The input consists of the rocket's orientation, the rate of change in orientation, α, β, the current throttle position of each engine, the altitude, and the velocity of the rocket in the direction of flight. These values are propagated through the network to produce a new throttle command (the amount of thrust) for each engine.

9.3.3 Experimental Setup

Each ESP network was evaluated in a single trial that consisted of the following four phases:

1. At time t_0, the rocket is attached to a launch rail that will guide it on a straight path for the first 50 feet of flight. The fuel tanks are full and the engines are ignited.

2. At time $t_1 > t_0$, the rocket begins its ascent as the engines are powered to full thrust.

3. At time $t_2 > t_1$, the rocket leaves the launch rail and the controller begins to modulate the thrust as described in section 9.3.2 according to equation 9.1.

4. While controlling the rocket one of two events occurs at time $t_f > t_2$:

   (a) α or β exceeds ± 5 degrees, in which case the rocket is about to tumble, and the controller has failed.

   (b) the rocket reaches burnout, in which case the controller has succeeded.

   In either case, the trial is over and the altitude of the rocket at t_f becomes the fitness score for the network.

In a real launch, the rocket would continue after burnout and coast to apogee. Since we are only concerned with the control phase, for efficiency the trials were limited to reaching burnout. This fitness measure is all that is needed to encourage evolutionary progress. However, there is a large locally maximal region in the network weight space corresponding to the policy $o_i = 1.0,\ i = 1..4$; the policy of keeping all four engines at full throttle. Since it is very easy to randomly generate networks that saturate their outputs, this policy will be present in the first generations. Such a policy clearly does not solve the task, but because the rocket is so unstable, no better policy is likely to be present in the initial population. Therefore, it will quickly dominate the population and halt progress toward a solution. To avoid this problem, all controllers that exhibited this policy were penalized by setting their fitness to zero. This procedure ensured that the controller was not rewarded for doing nothing.

Parameter	Value
Environment	
output scale (δ)	10
ESP	
no. of subpops	10
size of subpops	500
burst criteria	50
mutation rate	80%

Figure 9.7: **Task parameters.** ESP evaluated 5,000 networks per generation. The parameter δ is used in equation 9.1 to determine thrust of each engine.

All simulations used the parameter settings in figure 9.7. The parameter δ was set to 10 so the network could only control the thrust in the range between 90% and 100% for each engine. It was determined in early testing that this range produced sufficient differential thrust to counteract side forces, and solve the task.

As was discussed in section 9.2, evolving a controller directly for the finless rocket was too difficult and an incremental evolution method was used instead. First a controller for the quarter-finned rocket was evolved. Once a solution to this easier task was found, the evolution was transitioned to the more difficult finless rocket.

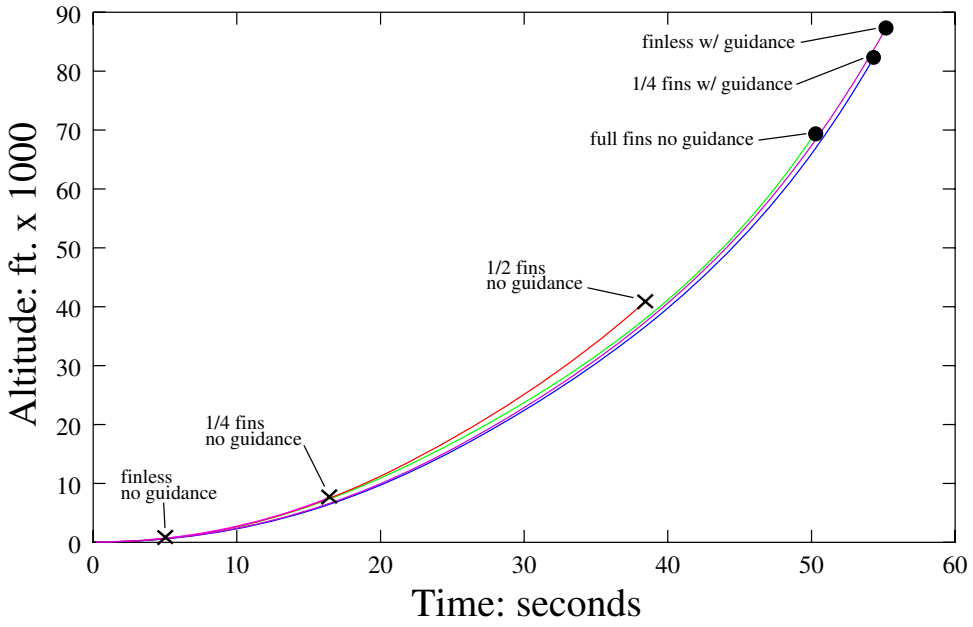


Figure 9.8: **Burnout altitudes for different fin-size rockets with and without guidance.** The crosses indicate the altitude at which a particular rocket becomes unstable (i.e. either α or $\beta > \pm 5$ degrees). The circles indicate the altitude of a successful rocket that remained stable all the way to burnout. The guided quarter-finned and finless rockets fly significantly higher than the unguided full-finned rocket.

9.4 Results

ESP solved the task of controlling the quarter-finned rocket in approximately 600,000 evaluations. Another 50,000 evaluations were required to successfully transition to the finless rocket. Figure 9.8 compares the altitudes that the various rockets reach in simulation. Without guidance, the full-finned rocket reaches burnout at approximately 70,000ft, whereas the finless, quarter-finned, and half-finned rockets all fail before reaching burnout. However, with neural network guidance the quarter-finned and finless rockets do reach burnout and exceed the full-finned rocket's altitude by 10,000ft and 15,000ft, respectively. After burnout, the rocket will begin to coast at a higher velocity in a less dense part of the atmosphere; the higher burnout altitude and the aerodynamically more efficient design of the finless rocket translates into an apogee that is about 20 miles higher than that of the finned rocket (figure 9.9).

Figure 9.10a shows the behavior of the four engines during a guided flight for the finless rocket. The controller makes smooth changes to the thrust of the engines throughout the flight. This very fine control is required because any abrupt changes in thrust at speeds

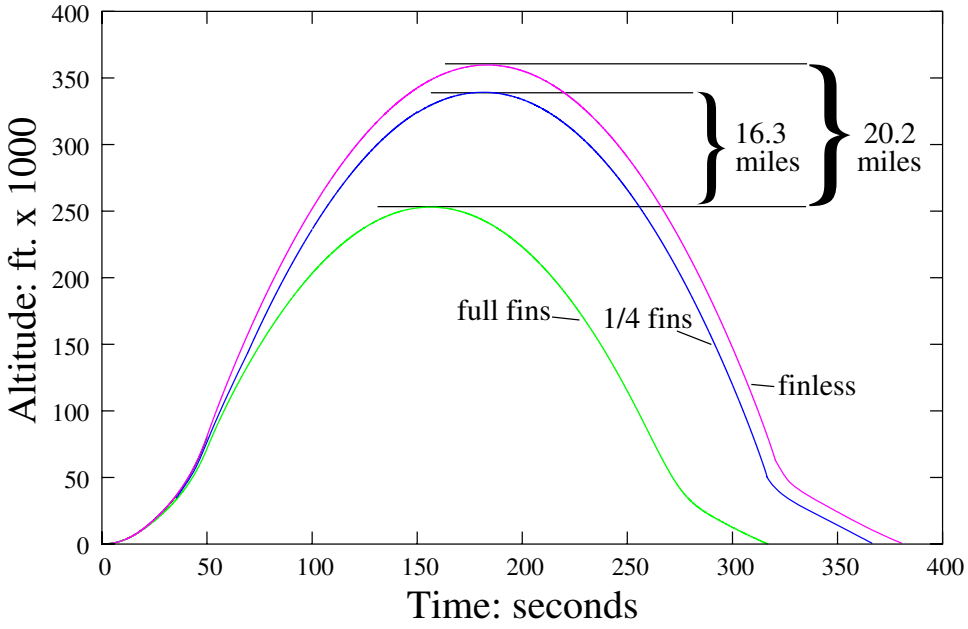


Figure 9.9: **Final altitudes for the unguided full-finned, guided quarter-finned, and finless rockets.** After a sounding rocket reaches burnout, it continues to coast up to apogee where it makes its scientific measurements. Because the quarter-finned and finless rockets begin to coast at higher altitudes (where the atmosphere is less dense), with greater velocity and less drag than the full-finned version, they can fly up to 20 miles higher with the same amount of fuel.

of up to Mach 4 can quickly cause failure. Figure 9.10b shows α and β for the various rockets with and without guidance. Without guidance, the quarter-finned and even the half-finned rocket start to tumble as soon as α or β start to diverge from 0 degrees. Using guidance, both the quarter-finned and finless rockets keep α and β at very small values up to burnout. Note that although the finless controller was produced by further evolving the quarter-finned controller, the finless controller not only solves a more difficult task, but is also better able to minimize α and β.

9.5 Discussion

The rocket control task is representative of many real world problems such as manufacturing, process control, and robotics that are characterized by complex non-linear interactions between system components. The critical advantage of using ESP over traditional engineering approaches is that it can produce a controller for these systems without requiring formal knowledge of system behavior or prior knowledge of correct control behavior. To

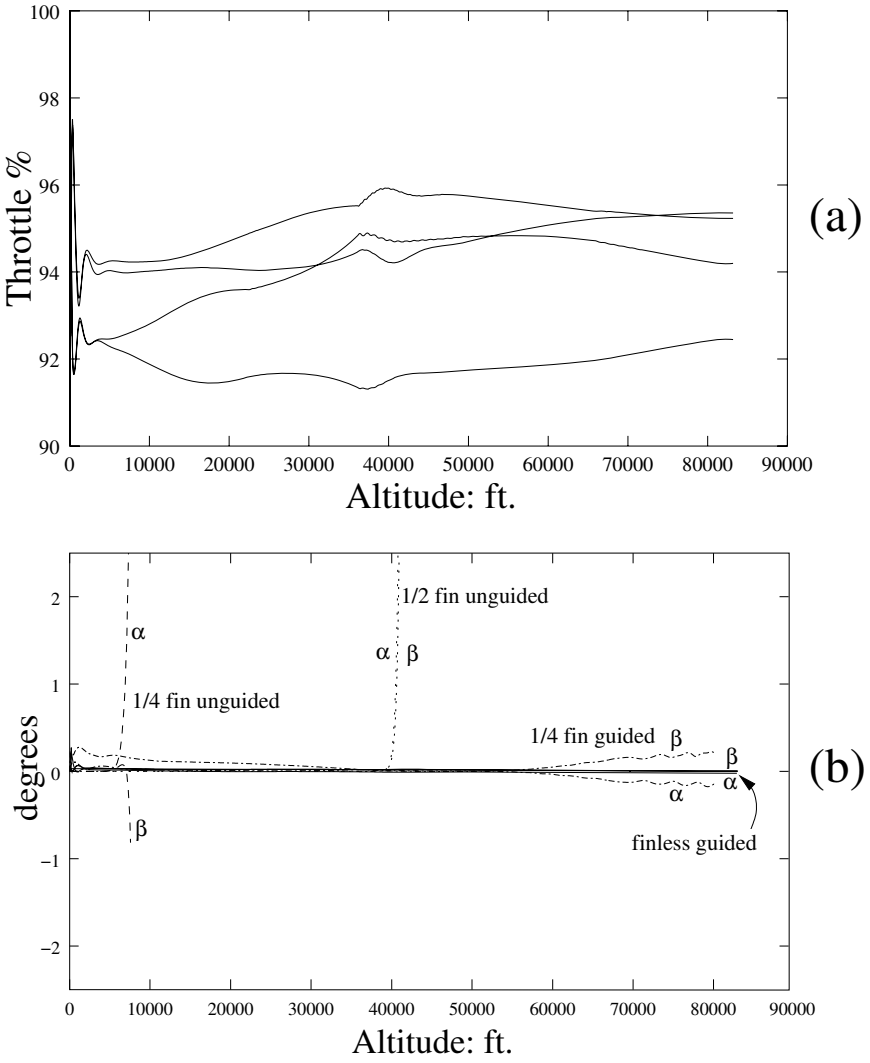


Figure 9.10: **Controller performance for the finless rocket.** Plot (a) shows the policy implemented by the controller. Each curve corresponds to the percent thrust of one of the four rocket engines over the course of a successful flight. After some initial oscillation the control becomes very fine, changing less than 2% of total thrust for any given engine. Plot (b) compares the values α and β for various rocket configurations, and illustrates how well the neural guidance system is able to minimize α and β. The unguided quarter-finned and half-finned rockets maintain low α and β for a while, but as soon as either starts to grow the rocket tumbles. In contrast, the guidance system for the quarter-finned and finless rockets is able to contain α and β all the way up to burnout. The finless controller, although evolved from the quarter-finned controller, is more optimal.

evolve a successful controller, ESP only had to measure the final altitude of each guidance attempt.

Of equal importance is the result that the differential thrust approach for the finless version of the current RSX-2 rocket is feasible. Prior to these experiments, it was not known whether the current performance specifications of the RSX-2's engines would provide responsive enough control to utilize this control scheme. Also, having a controller that can complete the sounding rocket mission in simulation has provided valuable information about the behavior of the rocket that would not otherwise be available.

Future work with the RSX-2 is discussed in the next chapter.

Chapter 10

Discussion and Future Directions

In this chapter, I discuss the three components of the methodology, and offer some directions for the future improvement of each. Several promising opportunities for applying the method to new domains are also presented, and a plan is outlined for continuing work in the rocket guidance problem.

10.1 ESP

10.1.1 Strengths and Limitations

ESP provides an efficient means to solve complex non-linear control tasks. By coevolving separate neuron subpopulations, ESP is able to form specializations rapidly, and reliably discover fruitful regions of the search space. The strength of the algorithm is further enhanced by burst mutation. As long as ESP can discover individuals in a neighborhood around a solution, it can still solve the task even after convergence by burst mutating and refocusing the population in that neighborhood. If ESP does not converge in or near such a neighborhood, burst mutation may not be able to escape the local minima. In situations where this problem occurs consistently, ESP can be run with larger subpopulations or, if possible, an easier configuration can be chosen as the initial task.

In chapter 4, ESP was shown to be significantly faster than the other methods 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. This result is important because despite its superior performance on these tasks, neuroevolution continues to receive much less attention from the machine learning community than conventional reinforcement methods. A key thrust of the comparisons is to raise the profile of NE and inspire broader research

interest in the field.

Although ESP's performance was roughly equal to that of NEAT in the pole balancing domain, I believe that the simplicity of ESP could be an overriding advantage when applying neuroevolution to more difficult tasks. NEAT, with its 23 user parameters, may prove more difficult to configure than ESP when applied to tasks that require larger networks, i.e. tasks that make more demanding use of NEAT's topology growing mechanism. Further study will be required to determine the parameter sensitivity of NEAT, and assess which types of problems benefit most from either of the two methods.

An interesting area for future work is to explore possible combinations of the two algorithms that would evolve both at the neuron and network level. For example, an ESP-like neuron level could provide the "raw material" for a NEAT-like network level. The network level evolves topologies or network *templates* that are instantiated with neurons selected from the neuron level. This idea is similar to the blueprints in SANE except that instead of specifying which neurons should be combined, the templates describe *how* different types of neurons should be combined. The challenge would be to organize the neurons such that they can adapt to perform useful functions in the different emerging topologies. Such a system could potentially harness the efficiency of neuron coevolution while also optimizing topology.

10.1.2 Probabilistic Subpopulations using Local Information

Currently, the number of subpopulations in ESP and their size is specified by the user at the outset of evolution. In addition, neurons are never permitted move to other subpopulations or mate with members of other subpopulations. While this restriction allows the neurons to specialize rapidly, an algorithm with softer boundaries between subpopulations might be able to sustain diversity for a longer period by allowing for some degree of neuron migration. This could be accomplished by making subpopulation membership probabilistic.

Each neuron maintains a probability mass function that it learns by sampling its own fitness in different network positions. Initially, all probabilities are uniform for all neurons (i.e. no preference for any particular hidden unit position). After the first generation each neuron develops a preference for some network positions over others as it obtains local information about which positions produce higher fitness. The probabilities would also be used to determine mating so that a neuron's preference becomes increasingly greedy throughout its lifetime, until the algorithm eventually approaches something close to ESP. Evolution starts with neurons that are positioned in networks at random (as in SANE), and gradually transitions toward disjoint subpopulations (ESP). The total number of neurons

would still be fixed, but the subpopulation sizes would change, assigning more neurons to network positions where they are most needed. This annealing process could help avert premature convergence giving evolution more time to discover a good region of the search space.

10.1.3 Probabilistic Subpopulations using Global Information

A more global approach than the one above could improve the autonomy of the algorithm even further by allowing the number of overlapping clusters of neurons to be determined automatically. For instance, if the population were modeled as a high-dimensional Gaussian Mixture Model (GMM), the number of clusters could be determined using the Expectation-Maximization (EM) algorithm. Initially, the neurons are distributed at random throughout the weight space and the EM is used to find the centers and variances of some predefined number of Gaussians (i.e. the number of hidden units in the networks) that fit the distribution. Then networks are formed by selecting one neuron from each cluster probabilistically, according to the model. The best neurons within each cluster are mated, and EM is run again on the new population, but now without specifying the number of clusters.

This way, not only would cluster membership be determined automatically, but the size of networks would be set by the changing distribution of neurons throughout the weight space. It is possible that such an algorithm would act to distribute neurons so that larger or smaller networks would be formed in response to the requirements of the task. Taking this idea one step further, the GMM representation could be used directly to generate new individuals in addition to those created through crossover. The approach could provide an additional source of diversity, and be generalized to other non-Gaussian kernel functions that may turn out to be better suited for neuroevolution.

Other extensions to ESP have already been implemented by other researchers, including combining ESP with rule-based systems (Fan et al. 2003), and adding another level to the system by using multiple conventional ESP processes to coevolve modules in hierarchical neural networks (Yong and Miikkulainen 2001).

10.1.4 Large-Scale Parallel Implementation of ESP

While improvements to the ESP algorithm should make it a more reliable method for a given population size, re-implementing the algorithm for parallel execution will greatly increase the population sizes the are feasible, and expand the range of tasks that can be solved efficiently. ESP, like most evolutionary systems, can be sped-up at a rate that is

linear in the number of processors used because each network evaluation in a generation can be performed independently.

To allow for the maximum number of processors to be utilized, I propose a parallel implementation of ESP that is based on a client-server model where the server and clients communicate primarily through a shared file system. At the beginning of a run the server is started on one machine and waits until it receives a signal from some number of clients each of which is running on a different remote machine. The server then generates random subpopulations of neurons, writes them to a file, and broadcasts the filename to the clients. Each client reads the file, evaluates some number of networks, and writes their fitnesses to another file that the server reads when all of the clients have completed their evaluations. The next generation begins after the server recombines the neurons, writes the new subpopulations to the file system, and, once again, signals the clients to start the next round of evaluations.

Since the CPU time of ESP is dominated by network evaluations (over 98% of the total CPU time for the tasks examined in this dissertation) and the clients only need to synchronize at generation boundaries, performing file I/O once per generation incurs negligible overhead and provides a simple and portable architecture. In chapter 8, a multi-threaded implementation was used to parallelize ESP for a specific shared-memory multiprocessor machine, the 14-processor Sun Enterprise. The implementation described here is more general, allowing a single ESP run to be distributed over an arbitrary number of processors, potentially harnessing the entire computational resources of e.g. local academic computing networks which commonly have one to several hundred machines. A speed increase of two orders of magnitude means that much larger simulations can be run, and, of equal importance, that more simulations can be run to study the task and determine useful environmental parameters. For instance, the large simulations that are planned for the RSX-2, which will require roughly a week of CPU time using the current implementation, could be reduced to a couple of hours.

10.2 Incremental Evolution

Although shaping is not a new concept, my work in this area constitutes some the first in applying the idea to evolution. Incremental evolution is a simple mechanism that allows the user to incorporate task knowledge to guide the evolutionary process at a coarse level of granularity. The technique should be applicable to all problems that can be naturally decomposed into a sequence of increasingly complex tasks. However, the user has to determine the free parameters that will reliably afford successful transitions for a population

of a given size. This may not always be easy to do: there may not always be a clear way to simplify a task without decorrelating it completely from the goal task, and it may not always be easy to determine the relative difficulty of asks.

However, the experiments in chapters 5, 7, and 9 show that, in practice, a reasonably low level of domain knowledge required to make effective use of incremental evolution. In the prey capture task, the problem was decomposed in a very intuitive manner that did not require special knowledge of the prey's policy. Likewise, in the rocket guidance domain, the problem of controlling the finless rocket was made possible by adding small fins to incrementally increase the rocket's stability—a solution that required only superficial knowledge of basic rocket dynamics. Furthermore, at least for Artificial Life and robot control, the task sequences are usually easy to come by because the goal task often subsumes natural layers of behavior (Brooks 1986).

10.2.1 Task Prediction

Incremental evolution relies on the user's intuition about the structure of the configuration space. To traverse the configuration space effectively requires some knowledge of the relative evolvability of different evaluation tasks. Currently, the incremental method does not exploit any task knowledge that may become available after some number of task transitions have taken place.

If the sequence of evaluation tasks $T : \{t_1, t_2, \ldots, t_n\}$ are related (as they should be), the solution of a subsequence of T, say $\tilde{T} : \{t_1, t_2, \ldots, t_{n-k}\}$ could provide some knowledge about t_{n-k+1}. It is very possible that $\tilde{T}$ represents a directed path in the weight space that can be used to predict where in the configuration space to start looking for the next solution network. For instance, a linear regression could be performed on the weights of the sequence of networks in $\tilde{T}$ to extrapolate a likely location of the next network. Burst mutating around this predicted point space could then lead to the solution of an evaluation task closer to the goal than would otherwise be attempted.

This approach could make incremental evolution more automatic, efficient, and capable of dealing with larger task transitions.

10.3 Controller Transfer

Transfer is potentially the most important stage in developing a neuroevolved controller. No matter how well the controller performs in the simulation environment, if it cannot

transfer successfully to the target environment, it is of no practical use. In some domains such as game-playing, transfer is not an issue, but for physical systems it must be ensured.

The experiments in chapter 6 revealed that a relatively accurate model combined with sensor noise 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 improved the transferred controllers significantly. The controllers were able to cope with a wide variety of conditions in the target environment that they had not experienced in the simulation environment.

This dissertation contributes the first examination of transfer in problems involving unstable systems, and demonstrates how such transfer can be achieved in principle. More research is needed to learn how to apply trajectory noise more optimally in order to maximize the amount of model error that can be tolerated. For example, different levels of trajectory noise could be applied to each state variable so that high levels of noise are used to widen the trajectory envelope in those dimensions where it is most needed, while maintaining the same level of evolvability (i.e. the number of evaluations required to solve the task) by reducing noise in less critical dimensions.

Instead of broadly sampling the state space to generate a training set for the model, future experiments should use data derived from an existing controller. This is important because in many real applications training data can only be obtained from the region of the state space that is visited while the system is being controlled. It would be interesting to see how well the controller can cope with regions of the target state space that were not explicitly modeled.

The next logical step is to apply the principles to a small-scale, but real, transfer experiment. On possibility is to use a physical apparatus such a pole balancer or similar unstable mechanical system. A model would be trained using data sampled directly from encoders mounted on the apparatus. Such an experiment would help validate the results in chapter 6, and provide more concrete information about the transfer process in a safe and inexpensive environment.

Together, ESP, incremental evolution, and transfer with trajectory noise provide a comprehensive base on which to build new and more powerful neuroevolution systems, and establish neuroevolution as a practical and effective design tool.

10.4 Applications

Beyond measuring performance on benchmark tasks, the real test of ESP lies in the applications. Too often learning methods are tested on limited problems and never fully evaluated

on full scale applications. The CMP resource allocation task (chapter 8) and the rocket guidance task (chapter 9) show that ESP can scale to the full complexity of the real world. These two applications exhibit all of the characteristics that must be dealt with to solve difficult control problems: non-linearity, high-dimensionality, continuous state and action spaces, partial observability, and stochasticity.

The ability to cope with various types of complexity in the environment suggests that ESP should be widely applicable outside the class of tasks treated in this dissertation. Already the algorithm has been applied to such diverse problems as discovering strategies in the game of Go (Perez-Bergquist 2001; Lubberts and Miikkulainen 2001), optimizing an aluminum recycling plant (Greer et al. 2002), developing cooperative strategies in robotic soccer (Whiteson et al. 2003), and evolving adaptive agent teams (Bryant and Miikkulainen 2003).

The RSX-2 will be the primary focus of ongoing research. The domain not only offers an interesting control problem, but also provides a rare opportunity to take the controller development process to completion. Future work will involve first changing the control scheme so that instead of generating a continuous control signal, the network will output a binary vector indicating whether or not each engine should "throttle back" to a preset low throttle position. This modification will simplify the control hardware on the real rocket by not requiring the engines to maintain arbitrary throttle settings.

Once a controller is evolved using this new scheme, work will focus on making the controller more robust by following the procedure developed in chapter 6. The target environment for this problem is characterized by variable and unpredictable conditions. Air density and wind intensity change at different rates through the atmosphere depending on the weather (which itself is not constant). In order to produce a sounding rocket that can be used in as wide a range of launch conditions as possible, new simulations must incorporate wind, and noise to force the controller to adopt a robust policy.

Incremental evolution will play an important role in this process. Starting with a controller that can stabilize the finless rocket under complete calm, wind will be introduced in stages, incrementally increasing its intensity until a level of approximately 5 knots is attained. Then trajectory noise will be introduced to compensate for the inevitable discrepancies between the simulated and real RSX-2. As with the robustness experiments in chapter 6, it is likely that resistance to wind (i.e. external disturbances) can be achieved by using trajectory noise alone, limiting the need to use wind explicitly in the network evaluations. Extensive testing will be conducted in the simulator to determine the performance envelope of each successfully evolved controller. This phase will be critical to the ultimate goal of transferring the controller to the RSX-2 and testing it in an actual rocket launch.

It may turn out that utilizing a single, monolithic controller for all reasonable flight conditions is not feasible. If so, it should be possible to partition the configuration space into qualitative regions, and evolve a controller for each separately. For example, a different controller could be evolved for low, medium, and high winds. The appropriate controller for a particular launch could either be selected prior to lift-off or chosen automatically by a discrete controller to build a hybrid control system for the rocket.

10.5 Conclusion

ESP in combination with burst mutation and incremental evolution has been successful in several domains. The current method is limited by requiring the user to estimate the initial network size, subpopulation size, and task schedule for incremental evolution. The extensions proposed in this chapter should go a long way toward making the method more self-sufficient by reducing the number of user parameters in ESP, and facilitating incremental evolution by using knowledge acquired from solved tasks. Further study of transfer will make these improvements worthwhile by providing a deeper understanding of the process that will ultimately determine the utility of the method and of neuroevolution in general. In the future, broader application of the method should be possible, establishing neuroevolution as a viable technique, and eventually leading to its regular use in industry.

Chapter 11

Conclusion

The goal of this dissertation was to provide a complete methodology for applying neuroevolution to real world control problems. In order for a neuroevolution system to be useful it must be powerful enough to evolve controllers in simulation that are robust enough to transfer to the real world. I have developed a complete approach designed to achieve this goal that consists of three components: the Enforced SubPopulations algorithm, incremental evolution, and controller transfer

This chapter summarizes the contributions of this dissertation, and then concludes with a reflection on the significance of this research and the prospects for the future.

11.1 Contributions

ENFORCED SUBPOPULATIONS

ESP represents a significant advancement in neuroevolution. The algorithm that ESP is built upon, SANE, has been demonstrated successfully on many problems, but cannot reliably evolve recurrent networks. ESP addresses this problem by using multiple subpopulations instead of a single population of neurons. This architecture makes neuron evaluations more consistent, allowing ESP to evolve recurrent networks, and therefore, solve tasks that require short-term memory.

The comparisons in chapter 4 showed that using subpopulations also results in greater efficiency by accelerating the specialization of neurons into useful network subfunctions. With the exception of NEAT, ESP was the the most efficient method in terms of both number of evaluations and CPU time. The performance of ESP and NEAT was statistically even on the two most difficult tasks, but the lower complexity of ESP suggests

that is may be more widely applicable.

INCREMENTAL EVOLUTION

Incremental evolution is a general technique that can be combined with any evolutionary algorithm. The experiments in chapter 5 showed that tasks that cannot be solved directly, using a given population size, can be solved efficiently and reliably by evolving on a sequence of increasingly difficult tasks.

The prey capture experiments demonstrated how two different faculties, memory and motor coordination, can be acquired efficiently by using intuitive knowledge about the task to structure the evolution. Incremental evolution was critical to the success of ESP in the RSX-2 domain. Evolving a controller for the finless rocket directly would have required much greater computational resources and allowed for much less experimentation in the domain.

CONTROLLER TRANSFER

The experiments in chapter 6 contribute the first study into the factors that influence the transfer of neuroevolved controllers in an unstable domain. This kind of study is necessary because it is often not safe or practical to experiment with real unstable systems. Therefore, it is important to first investigate transfer systematically in a safe and controlled setting before transferring to the real world.

The experiments showed that for unstable environments sensory noise is not beneficial. In contrast, trajectory noise produced robust controllers that could overcome model error to transfer successfully, and demonstrate general behavior in the target environment.

EMPIRICAL EVALUATION OF REINFORCEMENT LEARNING METHODS

Although, the primary thrust of the pole balancing comparisons was to demonstrate the efficiency of ESP, I believe that this study is a significant contribution in its own right. A total of nine different methods were compared on four tasks making it, to my knowledge, the most extensive comparison of both single-agent and evolutionary reinforcement learning methods to date.

The evolutionary methods consistently outperformed the single agent methods by a wide margin. Such a large difference in performance suggests that NE may be better suited to continuous reinforcement learning tasks than single-agent methods.

APPLICATIONS

ESP and incremental evolution were applied successfully to to three very different

applications: a pursuit-evasion contest (prey capture), chip-multiprocessor resource allocation, and the stabilization of a finless rocket.

The prey capture task, while smaller in scale and more abstract than the other two applications, is important because it requires short-term memory, and because it is related to a large class of autonomous agent tasks such as point-to-point navigation an object tracking.

The experiments with the chip-multiprocessor are a first step in trying to address the emerging need for effective on-chip resource management controllers. The task demonstrates ESP's ability to cope with high dimensional state spaces and unpredictable operating conditions. The evolved controller was able to allocate cache banks to the processors better than an equipartition of the cache, and generalize well to novel workloads.

The rocket guidance task is the most interesting, challenging, and promising application in the dissertation. The RSX-2 domain represents a full scale-up to the complexity of real-world non-linear control tasks. The task is difficult because it requires precise control and because the dynamics of the rocket change throughout the flight. ESP successfully evolved a controller that could stabilize the finless rocket to burnout, improving the final altitude of the rocket by over 40%. This achievement confirmed the feasibility of using differential thrust as a stabilizing mechanism for the RSX-2, and is an encouraging step toward an actual rocket launch.

11.2 Conclusion

Because real world control tasks are non-linear, there are no tractable and general mathematical solutions to these problems. Neuroevolution can be a means to solve such tasks if it can be made sufficiently efficient, and the resulting controllers are robust enough to transfer successfully. This research should bring neuroevolution closer to becoming a practical tool by addressing both issues. ESP has been shown to be the state of the art method in the difficult pole balancing benchmarks, and, combined with incremental evolution, it was able to scale up to the complexity of the RSX-2 domain. By studying the transfer process, this work lays the foundation for further investigation of how to make transfer reliable, and ultimately enable the industrial use of neuroevolution.

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 = 0.5m l_2 = 0.05m
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 4.3. 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	1b	2a	2b
no. of neurons	100	100	200	400
no. of blueprints	50	50	100	100
evals per generation	200	200	400	1000
size of networks	5	5	7	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	5
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.

Appendix C

The prey movement algorithm

The prey's actions are chosen stochastically. On each step, $(1 - v)$% of the time (where $v \in [0, 1]$ is user-defined) the prey will not move, and v% of the time it will choose one of the four actions, A_0 (north), A_1 (south), A_2 (east), and A_3 (north), according to the following distribution:

$$prob(A_i) = P_i/(P_0 + P_1 + P_2 + P_3),$$

where

$$\begin{aligned}
P_i &= exp(0.33 \cdot W(angle) \cdot T(dist)) \\
angle &= \text{angle between the direction of action } A_i \text{ and the direction from the prey to the agent,} \\
dist &= \text{distance between the prey and the agent,} \\
W(angle) &= (180 - |angle|)/180, \\
T(dist) &= \begin{cases} 15 - dist & \text{if } dist \leq 4, \\ 9 - dist/2 & \text{if } dist \leq 15, \\ 1 & \text{otherwise.} \end{cases}
\end{aligned}$$

Bibliography

Agarwal, V., Murukkathampoondi, H., Keckler, S., and Burger, D. (2000). Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.

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.

Bagnell, D., and Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *Internatinal Conference on Robotics and Automation*.

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

Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B. (2000). Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 282–293.

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., and Booker, L. B., editors (1991). *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann.

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.

Breitner, M., Pesch, H., and Grimm, W. (1993). Complex differential games of pursuit-evasion type with state constraints, part 1: Necessary conditions for optimal open-loop strategies. *Journal of Optimization Theory and Applications*, 78:419–442.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(10).

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.

Bryant, B., and Miikkulainen, R. (2003). Neuroevolution of adaptive teams: Learning heterogeneous behavior in homogeneous multi-agent systems. In *Congress in Evolutionary Computation, Canberra, Australia*.

Burger, D., and Austin, T. M. (1997). The simplescalar tool set version 2.0. Technical Report Technical Report 1342, Computer Sciences Department, University of Wisconsin.

Buskey, G., Roberts, J., and Wyeth, G. (2002). Online learning of autonomous helicopter control. In *Proceedings of Australasian Conference on Robotics and Automation*.

Carpenter, G. A., and Grossberg, S. (1987). ART 2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26:4919–4930.

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.

Colombetti, M., and Dorigo, M. (1992). Robot shaping: Developing situated agents through learning. Technical Report TR-92-040, International Computer Science Institute, Berkeley, CA.

Corliss, W. R. (1971). NASA sounding rockets, 1958-1968: A historical summary. Technical Report NASA SP-4401, National Aeronautics and Space Administration, Washington, D.C.

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

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314.

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

Diefendorff, K. (1999). Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13).

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

Dorigo, M., and Colombetti, M. (1998). *Robot Shaping: An Experiment in Behavior Engineering*, vol. 2 of *Intelligent Robotics and Autonomous Agents series*. MIT Press.

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

Eck, C., Chapuis, J., and Geering, H. P. (2001). Inexpensive autopilots for small unmanned helicopters. In *Proceedings of the Micro Mini Aerial Vehicles Conference, MAV*. Brussels, Belgium.

Elman, J. L. (1991). Incremental learning, or The importance of starting small. In *Proceedings of the 13th Annual Conference of the Cognitive Science Society*, 443–448. Hillsdale, NJ: Erlbaum.

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*.

Fan, J., Lau, R., and Miikkulainen, R. (2003). Utilizing domain knowlegde in neuroevolution. Unpublished manuscript (submitted to ICML-03).

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.

Floreano, D., and Nolfi, S. (1997). Adaptive behavior in competing co-evolving species. In Husbands, P., and Harvey, I., editors, *Fourth European Conference on Artificial Life*, 378–387. Cambridge, MA: MIT Press.

Floreano, D., Nolfi, S., and Mondada, F. (1998). Competitive co-evolutionary robotics: From theory to practice. In Pfeifer, R., editor, *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior*.

Giacconi, R., Gursky, H., Paolini, F., and Rossi, B. (1962). Evidence for X-rays from sources outside the solar system. *Physical Review Letters*, 9(11):439–444.

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

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

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

Greer, B., Hakonen, H., Lahdelma, R., and Miikkulainen, R. (2002). Numerical optimization with neuroevolution. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002)*.

Großmann, A. (2001). *Continual learning for mobile robots*. PhD thesis, School of Computer Science, The University of Birmingham, Birmingham, UK.

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.

Hammond, L., Hubbert, B., Siu, M., Prabhu, M., Chen, M., and Olukotun, K. (2000). The stanford HYDRA chip. In *IEEE MICRO Magazine*.

Hammond, L., Nayfeh, B., and Olukotun, K. (1997). A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85.

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.

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. New York: Macmillan.

Haynes, T., and Sen, S. (1995). Evolving behavioral strategies in predators and prey. In Sen, S., editor, *IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems*, 32–37. Montreal, Quebec, Canada: Morgan Kaufmann.

Higdon, D. (1963). *Automatic Control of Inherently Unstable Systems with Bounded Control Inputs*. PhD thesis, Department of Aeronautics and Astronautics, Standford University.

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

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.

Ikodinovic, I., Magdic, D., Milenkovic, A., and Milutinovic, V. (1999). Limes: A multiprocessor simulation environment for pc platforms. In *Third International Conference on Parallel Processing and Applied Mathematics (PPAM)*. Kazimierz Dolny, Poland.

Isaacs, R. (1965). *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*. Dover Publications.

Jacobs, R. A., Jordan, M. I., and Barto, A. G. (1991). Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive Science*, 15:219–250.

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).

Kaelbling, L. P., Littman, M., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence*, 4:237–285.

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., editor (1988). *Artificial Life I*. SFI Studies in the Sciences of Complexity. Addison-Wesley.

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.

Lee, S.-W., Hahn, W.-J., Oh, H.-C., Song, Y.-S., and Kim, S.-W. (1999). RAPTOR: A single chip multiprocessor. In *The First IEEE Asia Pacific Conference on ASICs*, 217–220.

Liang Lin, C., and Wen Su, H. (2000). Intelligent control theory in guidance and control system design: an overview. *Proc. Natl. Sci, Counc. ROC(A)*, 24(1):15–30.

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.

Lubberts, A., and Miikkulainen, R. (2001). Co-evolving a go-playing neural network. In *Coevolution: Turning Adaptive Algorithms Upon Themselves, Birds-of-a-Feather Workshop, Genetic and Evolutionary Computation Conference (GECCO-2001)*.

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.

Miller, W. T., Sutton, R. S., and Werbos, P. J., editors (1990). *Neural Networks for Control*. Cambridge, MA: MIT Press.

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.

Pai, V. S., Ranganathan, P., and Adve, S. V. (1997). RSIM: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*.

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.

Perez-Bergquist, A. S. (2001). Applying ESP and region specialists to neuro-evolution for Go. Technical Report CSTR01-24, Department of Computer Sciences, The University of Texas at Austin.

Perkins, S., and Hayes, G. (1996). Robot shaping–principles, methods, and architectures. Technical Report 795, Department of Artifical Intelligence, University of Edinburgh.

Pesch, H. J. (1992). Solving optimal control and pursuit-evasion game problems of high complexity. In Bulirsch, R., and Kraft, D., editors, *Proceedings of the 9th IFAC Workshop on Control Applications of Optimization*.

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.

Prete, C. A., Prina, G., and Ricciardi, L. (1995). A trace-driven simulator for performance evaluation of cache-based multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):915–929.

Reynolds, C. W. (1994a). Competition, coevolution and the game of tag. In Brooks, R., and Maes, P., editors, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life IV)*, 59–69. Cambridge, MA: MIT Press.

Reynolds, C. W. (1994b). 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.

Ring, M. B. (1994). *Continual Learning in Reinforcement Environments*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.

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

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E., and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, 318–362. Cambridge, MA: MIT Press.

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*.

Schultz, A. C. (1991). Adapting the evaluation space to improve global learning. In (Belew and Booker 1991), 158–164.

Seibert, G. (2001). A world without gravity. Technical Report SP-1251, European Space Agency.

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339.

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

Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. (1998). Multiscalar processors. In *25 Years ISCA: Retrospectives and Reprints*, 521–532.

Stanley, K. O., and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2). In press.

Strens, M. J. A., and Moore, A. W. (2002). Policy search using paired comparisons. *Journal of Machine Learning Research*, 3:921–950.

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.

Suykens, J. A. K., Vandewalle, J. P. L., and Moor, B. L. R. D. (1996). *Artificial Neural Networks for Modelling and Control of Non-Linear Systems*. Kluwer Academic Publishers.

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.

Thrun, S. (1996). *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer.

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*.

Whiteson, S., Kohl, N., Miikkulainen, R., and Stone, P. (2003). Evolving keepaway soccer players through task decomposition. In *Proceedings of the Genetic Evolutionary Computation Conference(GECCO-03)*.

Whitley, D., Mathias, K., and Fitzhorn, P. (1991). Delta-Coding: An iterative search strategy for genetic algorithms. In (Belew and Booker 1991), 77–84.

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.

Yao, X. (1993). A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, 4:203–222.

Yong, C. H., and Miikkulainen, R. (2001). Cooperative coevolution of multi-agent systems. Technical Report AI01-287, Department of Computer Sciences, The University of Texas at Austin.

Vita

Faustino John Gomez was born in Ft. Sill, Oklahoma on August 5th, 1969, the son of Asuncion Gomez and Faustino Gomez. After completing his work at Keystone School, San Antonio, Texas, in 1987, he entered Clark University in Worcester, Massachusetts where he earned the degree of Bachelor of Arts in Geography in 1991. In 1993, he moved to Austin, Texas, and in the fall of 1994 he entered the Graduate School of Computer Sciences at the University of Texas at Austin.

Faustino attended and presented papers at ICANN-98, IJCAI-99, IJCNN-01, won the best paper award in real world applications at GECCO-03, and published a paper in the Adaptive Behavior Journal.

Permanent Address: 407 West 18th Street #304
Austin, Texas 78701 USA
inaki@cs.utexas.edu
http://www.cs.utexas.edu/users/inaki/

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.