Robust Non-Linear Control through Neuroevolution

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

TR AI-2002-292

Abstract

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

1 Introduction

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

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

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

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

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

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

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

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

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

2 Background and Related Work

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

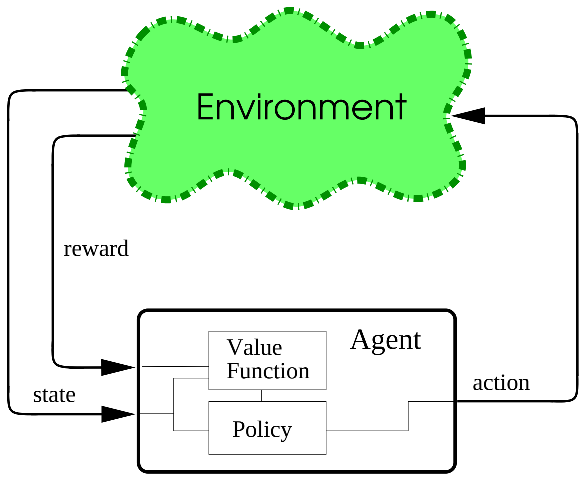

Figure 1: **The value function approach (color figure).** The agent is composed of a value-function and a policy. The value function tells the agent how much reward can be expected from each state if the best known policy is followed. The policy maps states to actions based on information from the value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

2.1 The Reinforcement Learning Problem

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

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

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

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

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

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

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

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

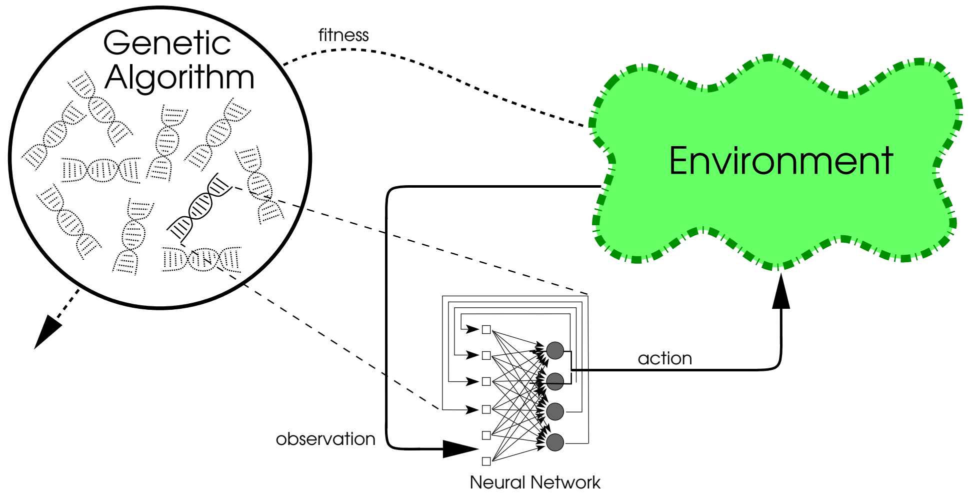

Figure 2: **Neuroevolution (color figure).** Each chromosome is transformed into a neural network phenotype and evaluated on the task. The agent receives input from the environment (observation) and propagates it through its neural network to compute an output signal (action) that affects the environment.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

2.2 Neuroevolution

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

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

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

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

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

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

2.3 Cooperative Coevolution

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

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

A very different kind of coevolutionary model emphasizes cooperation. Cooperative 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 Symbiotic, Adaptive Neuroevolution (SANE)

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

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

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

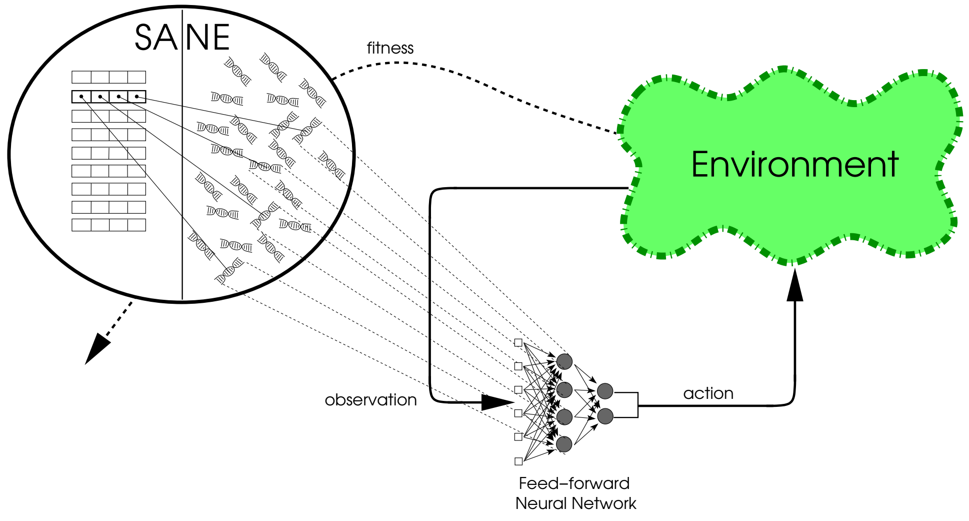

Figure 3: **Symbiotic, Adaptive Neuroevolution (color figure).** The algorithm maintains two distinct populations, one of network blueprints (left), and one of neurons (right). Networks are formed by combining neurons according to the blueprints. Networks are evaluated in the task, and the fitness is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

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

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

A key problem with SANE is that because it does not discriminate between the evolving specializations when it constructs networks and selects neurons for reproduction, evaluations can be very noisy. This limits its ability to evolve recurrent networks. A neuron's behavior in a recurrent network depends critically upon the neurons to which it is connected, and in SANE it cannot rely on being combined with similar neurons in any two trials. A neuron that behaves one way in one trial may behave very differently in another, and SANE cannot obtain accurate fitness information. Without the ability to evolve recurrent networks, SANE is restricted to reactive tasks where the agent can learn to select the optimal action in each state based solely on its immediate sensory input. This is a serious drawback since most interesting tasks require memory. The method presented in section 3, ESP, extends cooperative neuroevolution to tasks that make use of short-term memory.

2.5 Transfer

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

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

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

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

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

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

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

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

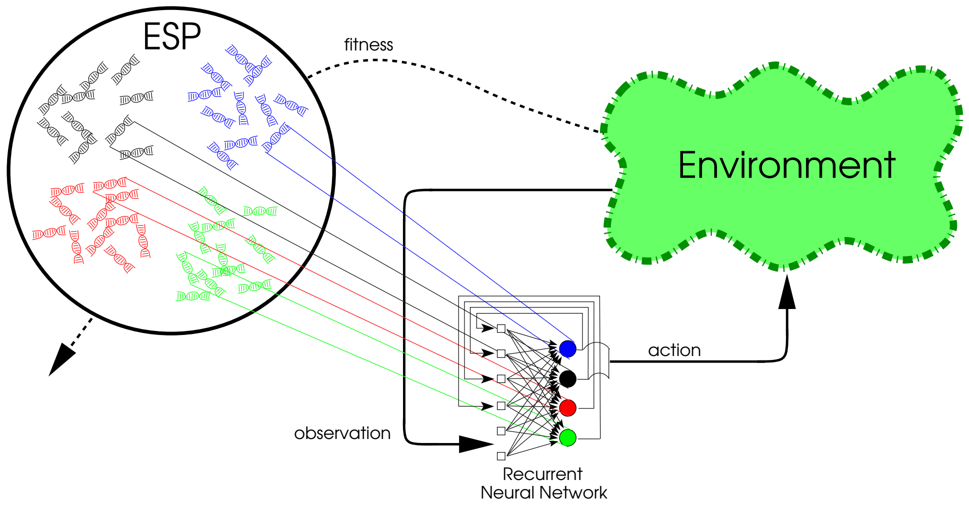

Figure 4: **The Enforced Subpopulations Method (ESP; color figure).** The population of neurons is segregated into subpopulations shown here in different colors. Networks are formed by randomly selecting one neuron from each subpopulation. As with SANE, a neuron accumulates a fitness score by adding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

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

3 Enforced Subpopulations (ESP)

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

Evolution in ESP proceeds as follows:

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

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

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

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

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

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

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

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

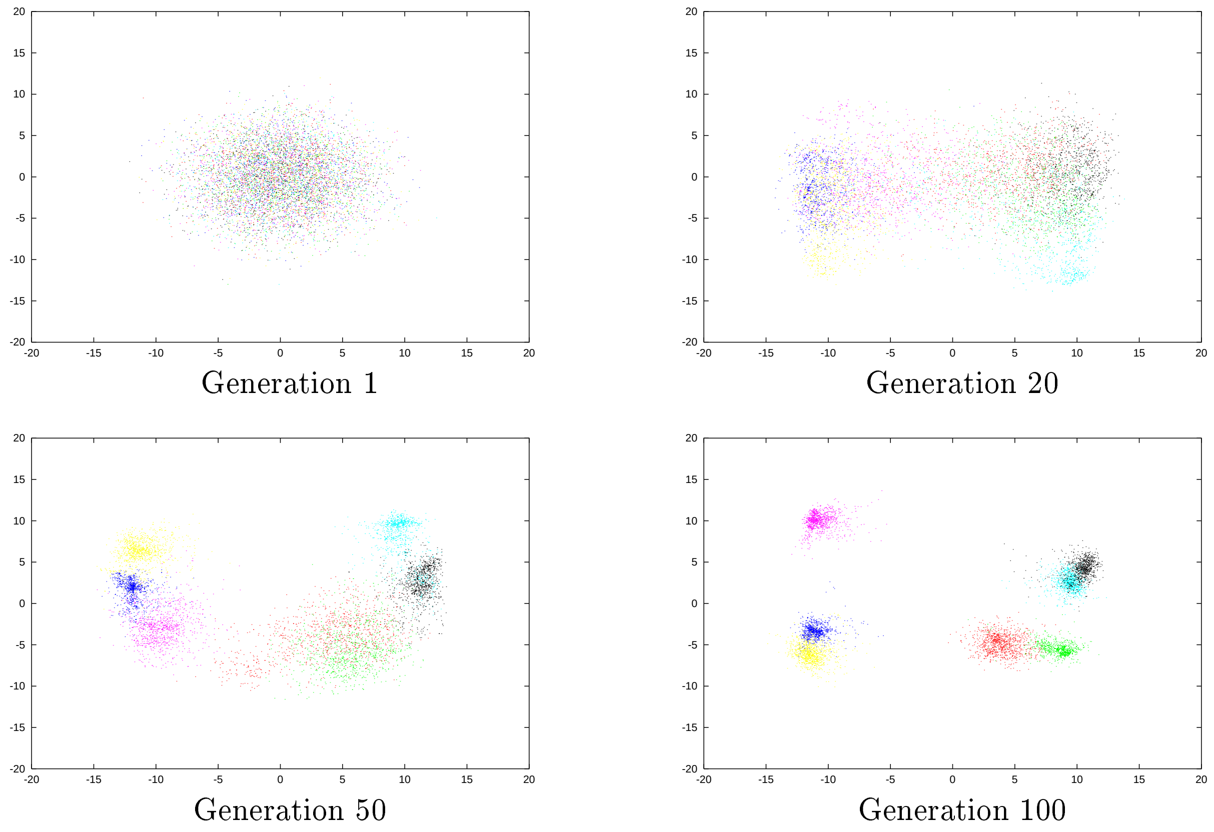

Figure 5: **Evolution of specializations in ESP (color figure).** The plots show a 2-D projection of the neuron weight vectors after Principal Component Analysis (PCA) transformation. Each subpopulation is shown in a different color. As evolution progresses, the subpopulations cluster into their own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

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

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

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

Burst mutation is similar to the Delta-Coding technique of Whitley et al. (1991) which was developed to improve the precision of genetic algorithms for numerical optimization problems. Because our goal is to maintain diversity, we do not reduce the range of the noise on successive applications of burst mutation and we use Cauchy rather that uniformly distributed noise.

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

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

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

4 Experimental Setup

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