Symbiotic Evolution of Neural Networks in Sequential Decision Tasks

David Eric Moriarty

Report AI97-257 January 1997

moriarty@cs.utexas.edu
http://www.cs.utexas.edu/users/moriarty/

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

Symbiotic Evolution of Neural Networks in Sequential Decision Tasks

by

David Eric Moriarty, M.S., B.S.

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

May 1997

Symbiotic Evolution of Neural Networks in Sequential Decision Tasks

To my mother, Dr. Gwen Childs Jones, whose unquestioning support has allowed me to accomplish all of my goals.

Acknowledgments

There have been many people who have contributed to the success of my dissertation, some unwittingly, some overtly. I can only hope to reflect some measure of my gratitude to those individuals by acknowledging their contributions in this section.

First and foremost, I must convey the largest debt of gratitude to my wife, Tara Estlin. Tara's encouragement gave me the confidence I needed, when I doubted my own ability to complete a PhD. She knew when to encourage and when to lay low. Her encouragement was only matched by her understanding of the trials and tribulations of PhD work. I appreciate the many discussions about our work, but also the times when we left the work at school and just enjoyed our home lives.

I give special thanks to my advisor, Risto Miikkulainen, for allowing me the freedom to pursue my research topic, which at the time was quite different from any of his other students. I hope that this new research direction fosters many new opportunities and discoveries for both you and your students. Risto taught me not only how to excel in research, but also how excellent research should be written so that it does not go unnoticed. On the first drafts of my early papers, there was at least a two to one ratio between Risto's comments and my original text. Of course, I quickly learned to expand the margins to accomodate all of the red ink. The net result was much better papers and a better understanding on my part of effective writing style. In addition to Risto, other faculty that have had a direct impact on my graduate career include Ray Mooney, Bruce Porter, and Ben Kuipers. Thank you all.

To my four great graduate school friends: Paul Baffes, Dan Clancy, Jeff Mahoney, and John Zelle, I thank you for the always intriguing and amusing conversations in Taylor 5.152. By the way what *is* the difference between apple juice and apple cider? I appreciate the much needed distractions your office brought and, not to be overshadowed, our numerous golf outings. Oh, how I will miss the freedom of graduate school.

There have been many other graduate students who have been both colleages and friends. Thank you to Joe Sirosh for suggesting the term "symbiotic", to Charles Calloway for the juggling lessons, to Mark Johnstone for giving me two new hobbies, and to Suzanne Buchele for putting up with a buch of AI students in one office. Other graduate students I'd like to thank include Yoonsuck Choe, Andrea Haessley, Marty Mayberry, Paul McQuesten, Sowmya Ramachandran, Cindi Thompson, and Lance Tokuda.

I'd also like to thank other colleagues around the world who have both inspired me and influenced my research. Thank you to John Grefenstette, Pat Langley, Mitch Potter, Alan Schultz, Jude Shavlik, and Rich Sutton.

And finally, I must give thanks to the people paying my bills. This research was supported in part by the National Science Foundation under grant #IRI-9504317.

David Eric Moriarty

The University of Texas at Austin
May 1997

Symbiotic Evolution of Neural Networks in Sequential Decision Tasks

Technical Report AI97-257

David Eric Moriarty, Ph.D.
The University of Texas at Austin, 1997

Supervisor: Risto Miikkulainen

Sequential decision tasks appear in many practical situations ranging from robot navigation to stock market trading. Because of the complexity of such tasks, it is often difficult to perceive the direct consequences of individual decisions and even harder to generate examples of correct behavior. Consequently, difficult decision problems such as routing traffic, autonomous control, and resource allocation are often unautomated or are only semi-automated using "rule-of-thumb" strategies or simple heuristics. This dissertation proposes a general methodology for automating these tasks using techniques from machine learning. Specifically, this research studies the combination of evolutionary algorithms and artificial neural networks to learn and perform difficult decision tasks. Evolutionary algorithms provide an efficient search engine for building decision strategies and require only minimal reinforcement or direction from the environment. Neural networks provide an efficient storage mechanism for the decision policy and can generalize experiences from one situation to another. The learning system developed in this dissertation called SANE contains an evolutionary algorithm specifically tailored to sequential decision learning. Populations evolve faster than previous methods and rarely converge on suboptimal solutions. SANE is extensively evaluated and compared to existing decision learning systems and other evolutionary algorithms. SANE is shown to be significantly faster, more robust, and more adaptive in almost every situation. Moreover, SANE's efficient searches return more profitable decision strategies. The flexibility and scope of SANE is demonstrated in two real-world applications. First, SANE significantly improves the play of a world champion Othello program. Second, SANE successfully forms neural networks that guide a robot arm to target objects while avoiding randomly placed obstacles. The contributions of this research are twofold: a novel integration of evolutionary algorithms and neural networks and an efficient system for learning decision strategies in complex problems.

Contents

Chapter 1

Introduction

A chess player makes hundreds of moves during a single game. It is not until the final move, however, that he learns whether his moves have been successful in winning the game. Which of his moves are most responsible for the win? Which moves had no effect on the outcome? How can the player apply the single reward from his entire sequence of moves to learn good individual moves? These questions have intrigued artificial intelligence researchers since the pioneering work of Arthur Samuel (1959). The crux of the problem is how to apportion credit to individual decisions based on an evaluation of a sequence of decisions and has been termed the *credit assignment problem* (Minsky 1963).

Game playing is just one example of the genre of problems that have been termed sequential decision tasks (Barto et al. 1990; Grefenstette et al. 1990; Littman 1996). Put simply, a sequential decision task is any task where a sequence of decisions must be made before their net effect can be measured. Some examples of real world sequential decision problems include directing automobile or air traffic, controlling the flow of chemicals in a chemical reactor, and routing information on the Internet. In each of these domains, the effect of a single decision is often not realized until some time in the future, and even then it is often unclear which decisions were most responsible for the outcome.

The fact that decisions often have both immediate and future consequences only contributes to the arduous nature of sequential decision tasks. Often the best strategy is not to maximize each immediate payoff, because some actions that produce high immediate payoffs may enter states from which high future payoffs are impossible. In chess, a piece capture achieves an immediate payoff by decreasing the opponent's piece count. However, the same capture could generate a negative future payoff if the capturing piece leaves a key defensive position unguarded. The decision strategy must consider both immediate and future payoffs of actions to optimize the total payoff.

The problem complexity has caused many decision tasks in real world situations to remain unautomated or to be only partially automated through simple, "rule of thumb" strategies. These policies are normally problem-general and do not take advantage of problem-specific knowledge inherent in each domain. For example, in a communication network, packet routing is normally decided by a shortest path strategy (Tanenbaum 1989), which is a problem-general policy. However, Littman and Boyan (1993) showed that better routing policies can be achieved using more domain-specific knowledge such as the specific network topology and traffic patterns.

A learning mechanism to automatically generate decision strategies and tailor it to the specific problem could greatly benefit a wide range of decision tasks. Learning such strategies, however, is often very difficult for standard, supervised machine learning approaches that require explicit examples of correct behavior. In many decision tasks, this knowledge is very costly to obtain or simply not available. Robot navigation on a distant planet, dispatching elevators, and game playing are all examples of decision tasks where correct behavior is not known *a priori*. A more flexible learning algorithm, capable of learning from simple and sparse reinforcements is necessary.

This dissertation presents a new approach that combines evolutionary algorithms and neural networks[1] to form a general learning mechanism for sequential decision tasks. Neural networks have proven very effective in pattern recognition and pattern association tasks and have been shown to generalize well to unseen situations. Generalization is very important in sequential decision tasks, because in large problems the learning system will be unable to explore every state of the system. Instead, it must generalize decision strategies from observed states to unobserved states. Neural networks provide an efficient mechanism for generalizing such decisions.

Evolutionary algorithms provide a general training tool in which few assumptions about the domain are necessary. Since evolutionary algorithms only require a single fitness evaluation over the entire (possibly multi-step) task, they are able to learn in domains with very sparse reinforcement, which makes them particularly well-suited for evaluating performance in sequential decision tasks. No examples of correct behavior are necessary. The evolutionary algorithm searches for the most productive decision strategies using only the infrequent rewards returned by the underlying system. Together evolutionary algorithms and neural networks offer a promising approach for learning and applying effective decision strategies in many different situations.

A novel neuro-evolution mechanism called SANE (Symbiotic, Adaptive Neuro-evolution) is presented that is specifically designed for efficient sequential decision learning. Unlike most approaches, which operate on a population of neural networks, SANE applies genetic operators to a *population of neurons*. Each neuron's task involves establishing connections with other neurons in the population to form a functioning neural network (figure 1). Since no one neuron can perform well alone, they must *specialize* or optimize one aspect of the neural network and connect with other neurons that optimize other aspects. SANE thus decomposes the search space, which creates a much more efficient genetic search. Moreover, because of the inherent diversity in the neuron population, SANE can quickly revise its decision policy in response to shifts in the domain.

The contributions of this dissertation are in two areas: sequential decision making and neuro-evolution. A system that can automatically construct effective domain-specific decision policies is significant not only in the field of computer science, but also in fields such as business operations management, military science, and engineering control systems. The goal is thus to create a powerful system for optimizing many types of decision tasks. SANE also presents a novel evolutionary mechanism that could spawn additional empirical and theoretical research both as a new neuro-evolutionary tool and as a new machine learning paradigm.

The body of this dissertation is organized as follows. The remainder of the introduction gives a more formal definition of the sequential decision task and describes some features that contribute to the complexity. Chapter 2 describes two different approaches for learning decision tasks through reinforcements, temporal difference learning and evolutionary reinforcement learning.

[1]Such combinations are often called neuro-evolutionary approaches or evolutionary neural networks.

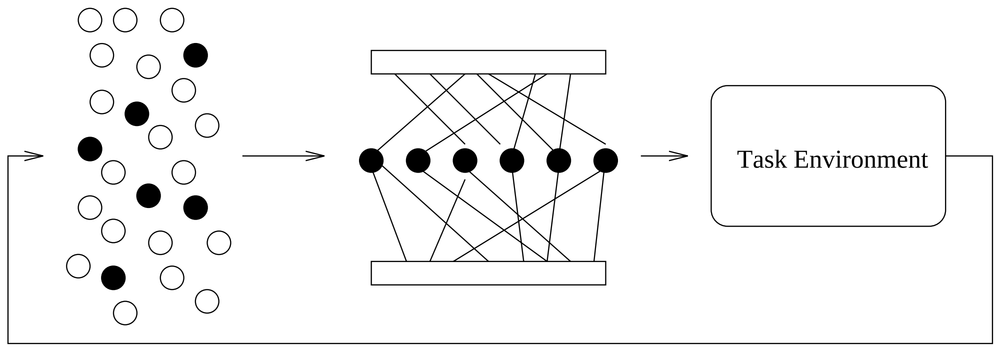


Figure 1.1: An overview of SANE's neuron-level evolution. Subpopulations of neurons are selected and used to build a neural network. The neural network is then evaluated in the task.

The similarities and differences between the two methods are discussed in the context of general issues in reinforcement learning. In chapter 3, the SANE decision learning system is motivated and described in detail. Chapter 4 presents an evaluation and in-depth experimental analysis of SANE's symbiotic search mechanism through comparisons with more standard methods in neuro-evolution. In chapter 5, SANE is compared to existing methods for sequential decision learning in two benchmarks: pole balancing and mobile robot control. Chapter 6 presents successful applications of SANE in two real world decision problems: game playing and robot arm control. Chapter 7 discusses work related to SANE including other evolutionary reinforcement learning methods and co-adaptive evolutionary algorithms. Chapter 8 describes other projects spawned by the research in this dissertation and outlines important future research directions. The final section summarizes the conclusions and underlines the significance of this research.

1.1 Sequential Decision Tasks

To understand the motivation for the learning methods presented in this dissertation, it is important to understand the difficulty and scope of the problems to be solved. Barto et al. (1990) and Grefenstette et al. (1990) coined the term sequential decision task in reference to a common genre of problems often encountered in artificial intelligence. This section gives a similar definition as well as several examples demonstrating the ubiquitous nature of these tasks.

1.1.1 Examples of Sequential Decision Tasks

Elevator Dispatching. All modern office buildings contain several elevators that transport individuals to and from different floors. As anyone who has ridden in elevators can attest, the basic movement strategy consists of movement in the same direction (either up or down), while stopping at floors that need servicing in that direction. If no floor buttons are active, the elevator normally remains in its current position.

If the goal of an elevator dispatching system is to minimize the average wait and ride time for an individual, there are clearly more efficient control strategies. For example, if it is the beginning of the day, elevators should not wait at the top of the building but should return to the bottom, since more people are likely to arrive than leave the building. Elevators should also skip floors with pressed call buttons if they have several stops to make and there is an elevator with a lighter load nearby.

An intelligent elevator dispatching strategy should consider the time of day, the position of the elevators, and the specific floors that need servicing before moving or stopping an elevator. A specific elevator movement may produce no obvious benefits, and in fact may lengthen the ride for several customers. The long term benefit, however, is to minimize the average wait and ride time for *all* customers. Such benefits cannot normally be measured after a single elevator movement, but must delay until several control decisions have been made.

Autonomous Robots. For decades researchers have studied the problems associated with building fully autonomous robots. A robot capable of maneuvering in unfamiliar environments and performing arbitrary tasks must master a rich collection of sequential decision tasks at many different levels. For example, mapping the visual and sensory input into appropriate motor control is often very difficult to specify by hand. Learning such behaviors is also difficult because examples of correct behavior may not be easily attainable. In an unfamiliar situation, the robot must actively explore its environment to discover the best course of behavior based on its input signals. Such exploration involves several trial and error sequences that may jointly receive a performance reward from the environment. Distributing credit from the single reward to individual behaviors is the key element of sequential decision learning that makes robust, adaptive robot behaviors difficult to learn.

Traffic Control. Rush hour traffic in most large cities has become nothing short of an adventure. Traffic congestion often occurs in different areas and at different times each day. Traffic signals that control the flow of traffic onto the highways and within the city can greatly affect the average commute time for a motorist. An intelligent traffic control system must make effective signal decisions based on the current traffic patterns and expected traffic patterns. Each control decision may have long reaching effects on the traffic pattern that are not realized until several time steps in the future.

Each of the above tasks is an instance of a sequential decision task. There are two common threads that run between each. First, examples of correct behavior are difficult to attain and second, it is difficult to perceive the direct consequences of individual decisions.

1.1.2 Problem Definition

Russell and Norvig (1994) argue that all artificial intelligence problems can be formulated from the perspective of an agent operating in an environment. Following their approach, I will define a sequential decision task from the perspective of a decision making agent. The agent in this case is the system responsible for making the decisions. It is the elevator controller, the autonomous robot controller, or the traffic controller.

In a sequential decision task, an agent interacts with a dynamic system by making *decisions* at specific time steps. The agent's decisions map directly or indirectly to *actions* that are performed in the domain. Put simply, an action is an interpretation of a decision. In most approaches to sequential decision learning the decision and action are synonymous, however, future systems may sharpen the distinction by using the agent's decision as a basis to compute the appropriate action.

The agent's decision is based on the values returned by its *sensors*. Sensors provide a view of the agent's current state and the state of the world. The input is normally a simplified representation of the environment and contains information pertinent to the future behavior of the system. Once an action is performed, the system enters a new state and the agent must make another decision. The state transition may be determined solely by the current state and the agent's action or may also involve stochastic processes. The specific decision-selection strategy employed by the agent is termed the agent's *decision policy* or simply its *policy*.

The agent receives reinforcement from the system in the form of *payoffs* that provide some measure of performance. The objective is to select the sequence of decisions that returns the highest total or cumulative payoff. Often the best strategy may not be to maximize each individual payoff, because some actions may produce high immediate payoffs but may enter states from which high later payoffs are impossible. The decision policy must take into account immediate and future payoffs of decisions to optimize the total payoff.

1.1.3 Properties of Sequential Decision Tasks

The above definition is a very general description and defines a family of sequential decision tasks. Each specific task contains additional properties that further classify it. It is important to understand these distinctions, because many decision policy learning methods are limited to only a particular class of the general sequential decision tasks. Several task properties defined by Littman (1996) are outlined below.

Finite vs. Continuous State Space. Is the number of possible states of the environment finite? In a finite state space problem, there exists an enumerable collection of states in which the environment can be. Chess is a finite state space problem, since the number of possible states is bounded by the number of possible board configurations. In a continuous state space problem, the number of possible states is infinite. Most control tasks that use real sensors to determine the state of the environment are continuous state space problems. For example, the number of possible sonar signals a mobile robot may receive to help avoid obstacles is, for all practical purposes, infinite.[2] This dissertation will consider both finite and continuous state space tasks.

Finite vs. Continuous Decision Space. Are the agent's range of actions infinite? Similar to the state space property, the agent may select decisions from a discrete set or a continuous range. Elevator control is a finite decision space problem, since the agent selects from a set of three actions (up, down, stop) for each elevator. Generating joint rotations in a robot arm is an example of a continuous decision space problem, since there is an infinite number of possible of rotations for each joint. This dissertation will consider both finite and continuous decision space tasks.

Accessible vs. Inaccessible. Are the agent's observations sufficient to describe the state of the environment? In an accessible problem, the input to the agent represents all of the features relevant to the current state. In an inaccessible problem, the agent must make decisions based upon only a partial view of the state. Both accessible and inaccessible tasks are explored in this dissertation. The Khepera task presented in chapter 4 and the OSCAR robot arm task presented in chapter 6

[2]Since computers only have finite memory, the resolution of the sensory signals is limited and thus the number of possible signals is actually finite. However, even with limited resolution, the number of possible states is so large that it can be considered infinite.

are examples of inaccessible tasks. Inaccessible tasks are also referred to as partially observable, since the agent only receives partial state information.

Markovian vs. Non-Markovian Tasks. Does the path to the current state influence future system behavior? The Markov assumption states that anything about a system's history is present in the current description of its state. In other words, in a Markovian task the current state description is sufficient to determine the future behavior of the system and the path to the current state is irrelevant. In non-Markovian tasks, properties external to the state description influence future state transitions. All of the tasks presented in this dissertation will be Markovian, although they may appear non-Markovian to the agent because they are inaccessible.

Stationary vs. Non-stationary Environments. Can the agent's environment change? In a stationary environment the state transition rules are always the same. They may be probabilistic, but the probabilities remain constant. In a Non-Stationary environment, the transition rules may change in response to time or possibly an agent's action. For example, if a robot's wheel falls off its movement decisions will no longer produce the same state transitions, since movement is much more restricted. In chapter 4, the Khepera robot task is modified to make it non-stationary.

I must make one more distinction in the type of problems to be studied in this dissertation. This research focuses on problems with little existing **domain knowledge**. That is, there is no domain model to analyze mathematically, nor is there a domain expert to provide examples of correct behavior. Often in real world decision tasks, such domain information is very costly or simply impossible to obtain. Consider the traffic control problem. An accurate domain model would require enormous effort to collect and analyze all of the data pertaining to the traffic flows and highway patterns. Once a model is built, even more effort is required to maintain the model to ensure that it continues to reflect the real traffic situation. Thus, a learning system that does not require a domain model, but instead forms effective decision policies through direct interactions with the actual domain reduces the implementation effort considerably. Moreover, in many tasks such a learning system may be the only feasible option. It is unlikely that an autonomous robot on a distant planet will have access to an accurate model of the planetary environment and landscape. To be effective, it must learn and adapt directly from it's experiences in the actual environment.

1.2 Concluding Remarks

This chapter has both motivated and defined the types of problems to be studied in this dissertation. Sequential decision tasks are quite arduous in nature because the environment typically provides only infrequent and many times very general feedback. The only concrete feedback a chess player receives is the final outcome of the game. The central issue in solving sequential decision tasks is how to apportion credit to individual decisions based on an evaluation of a sequence of decisions. The next chapter describes two different and somewhat opposing methods for accomplishing this goal.

Chapter 2

Learning Decision Strategies from Reinforcements

As described in the previous chapter, the hallmark of difficult decision problems is the credit assignment problem coupled with minimal domain information. To learn effective decision strategies in such tasks, a learning system must be capable of learning under very general and often infrequent reinforcements. This type of learning has become known as *reinforcement learning* (Sutton 1988). Kaelbling et al. (1996) identified two main branches of research in reinforcement learning: methods that search the space of behaviors and methods that search the space of value functions that assess the utility of behaviors. The former is representative of an evolutionary algorithm (EA) approach, and the latter a temporal difference (TD) approach.[1] The method of choice in this dissertation is an evolutionary algorithm.

The purpose of this chapter is to describe both methods of reinforcement learning and outline their similarities and differences. Because the temporal difference methods are much more visible in the literature, it is important to understand how the EA methods relate. Additionally, since EA approaches to reinforcement learning in the past can best be characterized as "every man for himself", it is important to draw a single thread that characterizes the general EA approach. In this chapter, I hope to provide the much needed generalization of the EA reinforcement learning approach and define the term *evolutionary reinforcement learning* (ERL).

The chapter begins with a characterization of reinforcement learning and contrasts it with the more common *supervised learning* methods. The two prominent reinforcement learning methods are then described in detail. The TD and ERL methods are compared and contrasted in the context of general issues in reinforcement learning. The goal of this chapter is to familiarize the reader with the somewhat opposing methods of reinforcement learning and to motivate the adoption of the evolutionary algorithm approach.

[1]Excluded from this section are the methods of dynamic programming such as value iteration and policy iteration (Bellman 1957; Bertsekas 1987; Puterman 1994). Dynamic programming is a model-based method that requires complete domain information to form decision policies. Since extensive domain information is normally absent from difficult decision problems, this section focuses on reinforcement learning methods that are model-free.

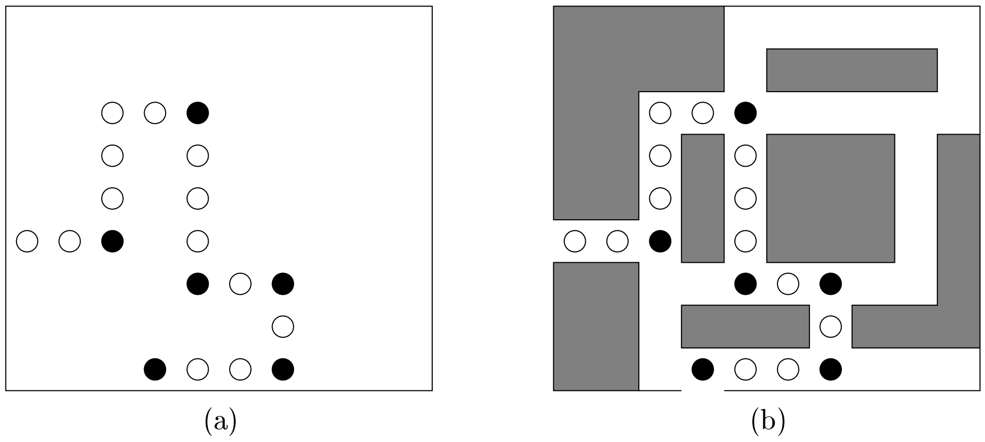


Figure 2.1: A characterization of the domain information necessary for reinforcement learning (a) and for supervised learning (b). The goal is to find the shortest path through the maze. The dots represent a possible path and the black dots represent decision points. To find the shortest path through the maze, a supervised learning method requires knowledge of all of the walls to decide which decisions were good and which were bad. A reinforcement learning system learns using only the total length of the path as a reward.

2.1 Reinforcement Learning vs. Supervised Learning

In reinforcement learning, the only response a decision making agent receives from the domain is in the form of reinforcement signals. Such signals provide only a general measure of proficiency in the task and do not explicitly direct the agent towards any course of action. This environment differs greatly from those found in applications of the more common *supervised learning* methods (Quinlan 1986). In supervised learning, the agent has access to examples of correct behavior and learns from errors between its decisions and the correct decisions. In reinforcement learning, the correct course of action is not known. The agent must learn good behavior through trial and error by directly interacting with the domain.

Figure 2.1 illustrates the difference between little and complete domain information. Imagine a robot learning the shortest path through a maze. Figure 2.1(a) characterizes the domain information from a reinforcement learning perspective. The robot makes a series of movement decisions and receives the total length of its path as its only reinforcement. Figure 2.1(b) characterizes the domain information required by a supervised learning method. The robot computes errors for each movement decision and must know the correct move at each decision point. Such domain information requires a priori knowledge of the positions of all of the walls in the maze. The problems discussed in this dissertation reflect the first maze, because in many interesting and important decision tasks very little domain information is available to the decision making agent.

2.2 Temporal Difference Reinforcement Learning

This section describes the temporal difference (TD) approach to reinforcement learning (Sutton 1988). TD learning is by far the most popular reinforcement learning method and has therefore become a standard for which alternative methods must be compared. The general idea of temporal difference learning is first described followed by a description of its two most prominent implementations: the Adaptive Heuristic Critic (Barto et al. 1983) and Q-learning (Watkins 1989; Watkins and Dayan 1992).

2.2.1 Learning Through Temporal Differences

In temporal difference learning (Sutton 1988), an evaluation or *value* function maintains *predictions* of current and future rewards. More specifically, the value function predicts the expected return from the environment given the current state of the world and the current decision policy. If the value function is accurate, the agent can base all of its decisions on its predicted values of subsequent states of the world. In other words, when selecting the next decision, the agent considers the effect of that decision by examining the expected value of the state transition caused by the decision.

The optimal value function is achieved using a version of the TD(λ) learning algorithm. TD(λ) uses observations of *prediction differences* from consecutive states to learn correct value predictions. Suppose that two consecutive states i and j return payoff prediction values of 5 and 2, respectively. The difference suggests that the payoff from state i may be overestimated and should be reduced to agree with predictions from state j. Updates to the value function V are achieved using the following update rule:

$$V(i) = V(i) + \alpha((V(j) - V(i)) + R(i)) \tag{2.1}$$

where α represents the learning rate and R any immediate reward. Thus, the difference in predictions $(V(j) - V(i))$ from consecutive states is used as a measure of prediction error. One can imagine a long chain of value predictions $V(0)..V(n)$ from consecutive state transitions with the last state $V(n)$ containing the true payoff from the environment. The values of each state are adjusted so that they agree with their successors and eventually the true payoff in $V(n)$. In other words, the true payoff is propagated backwards through the chain of value predictions. The net result is an accurate value function that can be used to assess the utility of decisions by comparing values from subsequent state transitions.

2.2.2 The Adaptive Heuristic Critic

One of the earliest reinforcement learning methods that used TD learning is the Adaptive Heuristic Critic (AHC). In the AHC, a TD value function called the "critic" is trained to predict the performance of a second agent that is responsible for generating the decisions. Figure 2.2 gives a block diagram of the interacting critic and decision agent. The critic uses equation 2.1 to learn the accurate predictions given the state transitions caused by the decision agent. The decision agent simultaneously updates its decision policy to maximize the value received from the critic. For example, if the decision agent receives a low value from the critic after making a decision, it should reduce the likelihood of making that decision in the same situation. Since the decision agent receives constant feedback from the critic, policy modifications can be made through a number of different hill-climbing search methods. The most common approach is to use a variant of the backpropagation (Rumelhart et al. 1986) method.

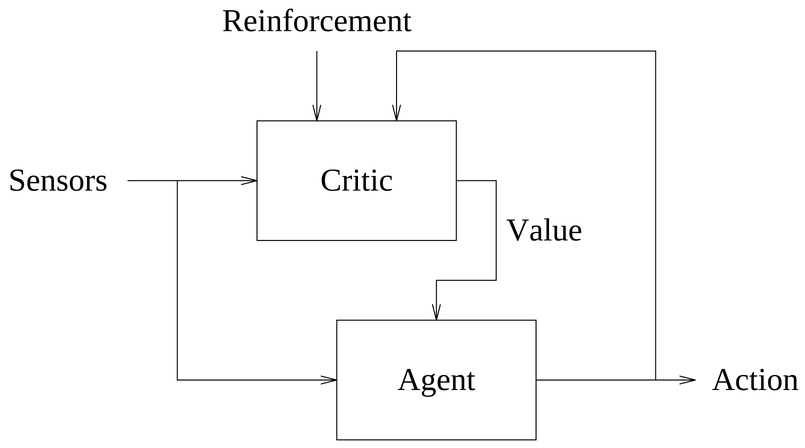


Figure 2.2: An overview of the Adaptive Heuristic Critic. The critic learns to provide error signals from which the action agent is trained.

2.2.3 Q-learning

Q-learning (Watkins 1989; Watkins and Dayan 1992) is closely related to the AHC and is currently the most widely-studied reinforcement learning approach. Q-learning combines the critic and decision agents into a single function called the Q-function. The Q-function maps decisions and states of the world into an expected reward estimate. In other words, the Q-function $Q(d,i)$ represents the utility of making a specific decision d in state i. Given accurate Q-function values, called Q values, an optimal policy is one which selects for each state the decision with the highest associated Q value (expected payoff).

The Q-function is learned through the following TD update equation:

$$Q(d,i) = Q(d,i) + \alpha(R(i) + \max_{d'} Q(d',i') - Q(d,i)) \tag{2.2}$$

where d' is the next decision and i' the next state. Essentially, this equation updates $Q(d,i)$ based on the current reward and the predicted reward if all future decisions are selected optimally. Watkins and Dayan (1992) proved that if updates are performed in this fashion, the Q-function will asymptotically converge to the optimal Q values. A reinforcement learning system can thus use the Q values to evaluate each decision that is possible from a given state. The decision that returns the highest Q-value is the optimal choice.

Q-learning has been shown to perform comparably to the AHC approach in terms of training time in toy domains (Lin 1992) and preliminary research has been done to scale up Q-learning to practical tasks (Lin 1993; Littman and Boyan 1993). However, there are several research issues like perceptual aliasing and generalization that must be resolved before Q-learning can be considered effective for large-scale tasks. Section 2.4 will outline some of these issues and relate them to the evolutionary reinforcement learning method described in this dissertation.

2.3 Evolutionary Reinforcement Learning

Evolutionary reinforcement learning provides an alternative to the temporal difference methods and overcomes many of TD's limitations. This section describes the evolutionary algorithm approach and gives some examples of how decision policies could be represented and operated on within an EA. ERL methods are not restricted to a specific evolutionary algorithm. Methods from genetic algorithms (Holland 1975; Goldberg 1989), evolutionary programming (Fogel et al. 1966), genetic programming (Koza 1992), or evolutionary strategies (Rechenberg 1964) could all be used in this framework to form effective decision-making agents. However, I do restrict ERL methods to those implementations of evolutionary algorithms that solve reinforcement learning problems.

2.3.1 Overview

Evolutionary algorithms are global search techniques patterned after Darwin's theory of natural evolution. Numerous potential solutions are encoded in structures called *chromosomes*. During each iteration, the EA evaluates solutions and generates offspring based on the fitness of each solution in the task. Substructures, or *genes*, of the solutions are then modified through genetic operators such as mutation and recombination. The idea is that structures that led to good solutions in previous evaluations can be mutated or combined to form even better solutions in subsequent evaluations.

In evolutionary reinforcement learning (ERL), the solutions take the form of decision making agents that operate in dynamic environments. Agents are placed in the world where they make decisions in response to environmental conditions. The EA selects agents based on their performance in the task, and applies genetic operators to generate new types of agents. Since evolutionary algorithms only require a single fitness evaluation over the agent's entire (normally multi-step) task, they find effective solutions in domains that return only occasional reinforcement over a sequence of actions. The only feedback that is required from the environment is a general measure of proficiency for each agent.

Like TD methods, ERL is a model-free approach, because it does not require a simulation model or knowledge of the state transition rules to form its policies. Many policy generation methods from the operations research community require precise models of the state transition probabilities in order to iterate through different policies using dynamic programming. ERL methods can learn "online" through direct interaction with the underlying system. Online learning frees the implementor from extracting expensive domain information to build an accurate model. Moreover, online learning is more adaptive since the agent observes changes immediately, rather than after the model is updated. However, since many explorative control strategies move the system in undesirable states, online learning can also be very costly and even dangerous. For this reason, researchers in both ERL and TD learning often train in simulation and then apply the control policy to the real system.

2.3.2 Policy Representations

In ERL, the decision policy normally maps sensor values directly to a decision. For example, a decision policy in a mobile robot may map sonar sensors values to wheel rotations. This representation is quite different from TD methods. In TD learning, the decision policy represents a value function that maps sensor and decision pairs to a utility measure. For example, a TD policy may

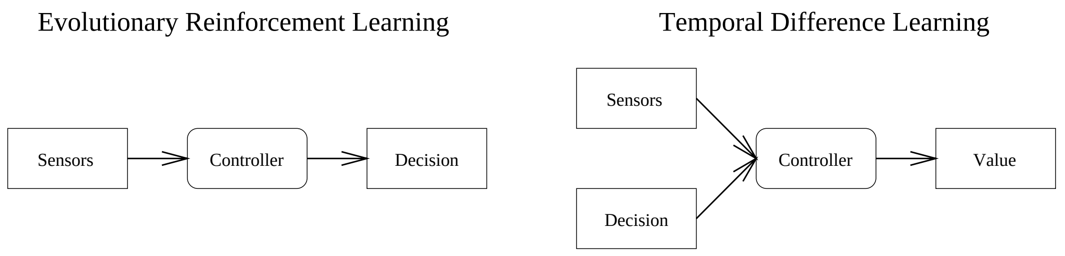

Evolutionary Reinforcement Learning
Sensors
Controller
Decision
Temporal Difference Learning
Sensors
Decision
Controller
Value

Figure 2.3: The input to output mapping of an evolutionary reinforcement learning policy compared to the mapping of a temporal difference policy or Q function.

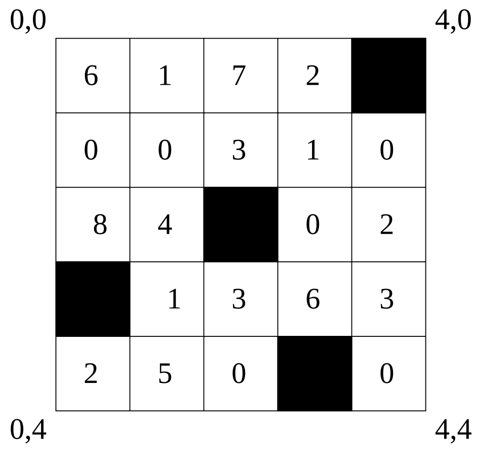


Figure 2.4: A simple grid world sequential decision task. Agents move from one box to another by selecting between four moves (up, down, left, or right). The number in each box represents the reward the agent receives when it enters a box, and the agent's sensors can detect the actual box that it is in. The goal of the agent it to maximize its rewards by moving into the best boxes.

map the sonar sensors and wheel rotations to an expected payoff estimate. Figure 2.3 illustrates the difference. Whereas ERL methods search the space of behaviors directly and learn to generate good decisions in response to specific state features, TD methods approach the problem indirectly by learning to predict the value of decisions from states of the system.

To better understand the different policy representations, consider the grid world shown in figure 2.4. The task of the agent is to move from box to box by selecting among four decisions (left, right, up, down) and collect the most rewards. Figure 2.5 shows a possible value function representation of a decision policy for this task. This table-based representation associates an estimate of current and future rewards with each possible state of the system (i.e. each box). The agent adjusts those estimates by experiencing different states and rewards. Figure 2.6 shows a possible policy representation using an evolutionary algorithm. Rather than value estimates, EA policies associate specific decisions with each state.

It's important to note that ERL methods could employ a value function for policies. However, value functions are problematic for several reasons. First, only a finite number of options can be considered at each decision point. The decision space itself may be continuous, but because a decision must be evaluated to be chosen, the number of options from any state is finite. In other words, each option must be passed through the value function before it can be chosen. Conse-

State:	0,0	1,1	2,0	3,0	0,1	1,1	2,1	3,1	4,1	0,2	1,2	3,2	4,2	1,3	2,3	3,3	4,3	0,4	1,4	2,4	4,4
Policy	12	11	4	11	15	9	17	16	20	25	24	15	8	19	14	20	21	12	11	14	15

Figure 2.5: A table-based value function representation of a decision policy for the grid world. A value is estimated with each possible state (box) the agent can be in. The value represents the expected return from that state and future states that will be reached from that state using the current policy. The agents moves in the direction of the highest estimated values. For example, if the agent is in box 0,0, it will move to state 0,1, because it has a higher estimated value than state 1,1. The values shown are not the optimal Q values.

Population of Policies

State:	0,0	1,1	2,0	3,0	0,1	1,1	2,1	3,1	4,1	0,2	1,2	3,2	4,2	1,3	2,3	3,3	4,3	0,4	1,4	2,4	4,4	Fitness
Policy	D	R	R	R	D	L	R	U	L	L	U	U	D	R	D	R	U	R	U	L	U	16
Policy	D	R	R	R	D	L	R	U	L	L	U	U	D	R	D	R	U	R	U	R	U	11
Policy	R	D	U	R	D	U	R	U	R	U	U	U	L	D	R	R	U	R	U	U	D	19
Policy	D	R	R	R	D	L	R	U	L	L	U	U	D	R	R	R	U	U	U	L	U	20
Policy	D	R	R	R	D	L	R	U	L	L	U	U	D	R	D	R	U	R	U	L	U	12

Figure 2.6: A simple policy representation for an evolutionary algorithm. A population of five policies are shown. In each policy, the decision is specified for each state of the world. A fitness value is shown for each policy and was computed by placing the agent initially in the lower left corner (box 0,4) and allowing 5 moves.

quently, the number of possible decisions is bounded by the number of options passed through the function, which is finite. If a mobile robot is controlled by generating rotations to its wheels, only a finite number of rotations can be considered at every step. In contrast, a direct mapping from sensors to decisions does not restrict the number of options, and can consider an infinite number of rotations for the robot's wheels. Many ERL implementations of direct policy mappings, however, do restrict the decision selections to finite sets, but this is a product of the policy implementation not the policy mapping.

A second potential disadvantage of the value function is that agents must expend more CPU time than a direct policy mapping to compute their decision. A direct policy representation requires only a single mapping from sensors to output to produce the appropriate decision. A value function, however, requires an input to output mapping for each decision that is being considered. If the decision choices are numerous, the CPU time required to cycle through each option could become a liability to the agent.

2.4 Issues in Reinforcement Learning

Evolutionary algorithms are traditionally viewed as function optimizers and have been analyzed and described using specific terminology developed within the EA communities. However, it is important to ground their application for reinforcement learning using terminology common in the existing reinforcement learning literature. While the temporal difference community has developed well-defined features and terminology among all TD approaches, the ERL approach has largely been "every man for himself." This section endeavors to bridge the gap between ERL methods and TD methods and draw some common threads between the two. Specifically, the section characterizes the similarities and differences in ERL and TD methods by how they address six issues in reinforcement learning: exploration vs. exploitation, the credit assignment problem, perceptual aliasing, generalization, policy updates, adaptation, and memory.

The goals of this section are twofold. First, it familiarizes the reader with important learning-related issues common to any reinforcement learning system. Second, it motivates evolutionary algorithms as effective search methods for reinforcement learning by relating their features to and describing advantages over the more common temporal difference approaches.

2.4.1 Exploration vs. Exploitation

In a sequential decision task, the sequence of policy decisions affects the states of the world that are visited. To form an effective global policy, the agents must actively explore the environment to discover the best states of the system. However, since the goal is to make decisions that return the highest payoffs, there is also pressure to exploit sequences of decisions that return the best known reward. Thus, there is a fundamental tradeoff between exploration and exploitation of the state space. Too much exploration could result in lower average rewards and too little could prevent the learning system from discovering the optimal policy.

In TD learning, there is normally a single agent with a single decision policy. TD methods exploit the state space by choosing decisions that return the highest reward from the value function. Exploration is normally achieved by interleaving random decisions with policy decisions. Typically, the agent can control the frequency of random decisions through it's value function. For example, a decision with a low Q-value may be more often replaced by a random decision than a decision with a higher associated Q-value. Another popular strategy is to choose more decisions randomly early in training to encourage initial exploration.

ERL methods blend exploration and exploitation of the solution space through the population of different strategies. Since multiple decision policies are represented, ERL methods automatically sample various policy decisions and thus random decisions are unnecessary. Each agent exploits it's own strategy, while the population as a whole explores alternative strategies. If diversity is maintained within the population, the agents' decision policies will differ, allowing the evolutionary algorithm to explore and compare different areas of the state space. How to easily maintain effective levels of diversity to promote exploration, however, is currently an open research issue and is the primary motivation of the SANE system described in chapter 3.

2.4.2 The Credit Assignment Problem

In a reinforcement learning problem, rewards often reflect the goodness of a sequence of decisions rather than each individual decision. A reward may be received after each decision, but it often derives from several of the agent's previous decisions. For example, a robot may receive a very high reward after a movement that places it in a "goal" position within a room. The robot's reward, however, reflects many of its previous movements leading it to that point. A difficult credit assignment problem therefore exists in reinforcement learning in how to apportion the rewards of a sequence of decisions to each individual decision.

Both ERL and TD methods address the credit assignment problem, but in very different ways. The difference can best be described as explicit vs. implicit credit assignment to the individual decisions. In TD approaches, credit from the reward signal is explicitly propagated to each decision made by the agent. Credit is assigned based on the difference in the predicted reward from each visited state and the actual final reward. In this manner, rewards are distributed and associated with each individual state and decision pair.

In ERL, rewards are normally only associated with sequences of states and decisions, not with individual decisions. Credit assignment for each individual decision is made implicitly, since poor agents generally select poor individual decisions. Thus, which individual decisions are most responsible for a good or poor decision policy is irrelevant to the evolutionary algorithm, because by selecting against poor policies, evolution automatically selects against poor policy decisions.[2]

The implications of the different credit assignment strategies are currently unclear. However, the choice of strategy does affect how each method addresses other important issues in reinforcement learning. The next section discusses one such issue: perceptual aliasing.

2.4.3 Perceptual Aliasing

Often in real world situations, the decision-making agent will not have access to complete information on the state of its world. Its sensors are more likely to provide only a partial view that does not disambiguate between many states. Consequently, the agent will often be unable to completely distinguish its current state. This problem has been termed *perceptual aliasing* or *hidden state*. Using the terminology from the first chapter, a domain with hidden states is *inaccessible*.

Unfortunately, both ERL and TD methods operate under the assumption that the agent's sensors accurately identify the state of its world. If this is not the case, both methods degrade and can produce suboptimal policies. Consider the inaccessible situation in figure 2.7, where an agent must act without complete state information. Circles represent the states of the world, and the colors represent the sensor information the agent receives in the state. Square nodes represent goal states with the corresponding reward shown inside. In each state, the agent has a choice of two actions (L or R). For this example, assume that the transitions are deterministic, and the agent is equally likely to start in either the state with the red or green sensors.

In this example, there are two different states that return a sensor reading of *blue*, and the agent is unable to distinguish between them. Moreover, the actions for each *blue* state return very different rewards. A Q function applied to this problem treats the sensor reading of *blue* as

[2]Some ERL implementations such as SAMUEL (section 7.1.1) actually do use local credit assignment for individual states or decisions. However, this credit assignment is normally used for small refinements, while the EA remains the primary search engine.

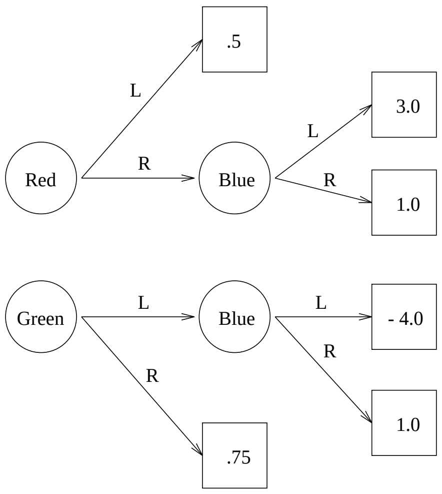


Figure 2.7: An environment with incomplete state information. The circles represent the states of the world and the colors represent the agent's sensory input.

one observable state, and the rewards for each action are averaged over both *blue* states. Thus, $Q(blue, L)$ and $Q(blue, R)$ will converge to -0.5 and 1, respectively. Since the reward from $Q(blue, R)$ is higher than the alternatives from observable states *red* and *green*, the agent's policy will choose to enter observable state *blue* each time. The final decision policy is shown in table 2.1. This table also shows the optimal policy with respect to the agent's limited (Markovian) view of its world. In other words, the policy reflects the optimal choices if the agent cannot distinguish the two blue states.

	Value Function Policy	Markovian Optimal Policy
Red	R	R
Green	L	R
Blue	R	L
Expected Reward	1.0	1.875

Table 2.1: The policy and expected reward returned by a converged Q function compared to the optimal policy given the same sensory information.

By associating values with individual observable states, the TD methods are very vulnerable to hidden state problems. In this example, the ambiguous state information misleads the TD method, and it mistakenly combines the rewards from two different states of the system (the two *blue* states). By mixing information from multiple states, TD methods fail to recognize the advantages of actions from specific states. In this example, the TD method does not recognize that choosing action L from the top *blue* state achieves a very high reward.

While ERL methods are also vulnerable to hidden state problems, their credit assignment strategy makes them more robust than the TD approaches. Since ERL methods associate values with entire state and decision sequences, they can implicitly disambiguate sensor information by considering the path the agent takes to each observable state. In this example, the evolutionary algorithm can recognize the disparity in rewards from the different *blue* states and evolve agents that enter the good *blue* state and avoid the bad one. The agent itself remains unable to distinguish the two *blue* states, but the evolutionary algorithm recognizes the distinction and compensates by redirecting the agent.

An ERL method can achieve the optimal policy in the previous example given the existing state information. Agents that start in the *red* state and choose the action sequence R,L achieve the highest levels of fitness. Such agents are selected for reproduction by the EA, and the next generation contains more agents that generate this sequence from the *red* state. If these agents are placed in the *green* state and select action L, they receive the lowest fitness score, since their subsequent action, L from the *blue* sensors, returns a negative reward. Thus, many of the individuals that achieve high fitness when started in the *red* state are selected against when they choose L from the *green* state. Since the EA will continue to promote individuals that select R,L from the *red* state, to survive individuals will learn to select R from the *green* state to minimize their fitness penalty.

Kaelbling et al. (1996) describe several solutions to the hidden state problem, where additional features such as the agent's previous decisions and observations are automatically generated and included in the agent's sensory information (Chrisman 1992; Lin and Mitchell 1992; McCallum 1995; Ring 1994). These methods have been effective at disambiguating states in initial work. An important future research question is whether similar methods can resolve all of the hidden states in a large, real-world application.

It is important to note that solutions to the hidden state problem are not exclusive to TD approaches, but should benefit ERL as well. While ERL appears more robust in the presence of hidden states, their performance is nevertheless affected. In the previous example, the ERL agents should achieve a better average reward than the TD method, but they are unable to procure both the 3.0 and 1.0 rewards from the two *blue* states. These rewards could be realized, however, if the agent could separate the two *blue* states. Thus, any method that generates additional features to disambiguate states presents an important asset to ERL. Such combinations suggest an important bridge between the two research communities.

2.4.4 Generalization

As reinforcement learning scales up, the number of possible states of the world normally grows exponentially with the size of the task. In large state spaces, agents cannot observe every state and must apply action decisions learned in observed states to unobserved states. Researchers in both ERL and TD learning have thus turned to generalization tools such as artificial neural networks[3] and rule bases to represent the decision policy. Such methods do not represent each state explicitly, but rather represent the decision policy as a function that maps sets of states to decisions. For example, in the grid world example from figure 2.4, a very general rule that affects many situations is the following: "if the agent is in a column greater than 2, move left." This rule is very useful because most of the high rewards are located on the left side of the world. Instead of learning the left action for each right side box, the controller learns one rule to cover them all. Thus by learning the mapping from features of states to decisions, the agent generalizes it's control policy from a few observed states to the numerous unobserved states.

In TD learning, the use of generalization tools is often called function approximation, since the value function or Q function is approximated rather than represented explicitly in a table of

[3]A background on neural networks is presented in appendix A.

values. The most common form of function approximation is a neural network that represents a Q function. Rather than representing each table value explicitly, the values are represented in a distributed fashion in the weights of the neural network. Thus, policy updates no longer modify single table values, but instead modify all of the weights in the neural network. Consequently, an update from a single state observation influences all other policy decisions and effectively generalizes actions in observed states to unobserved states. Network weights are normally updated using a gradient descent algorithm such as backpropagation (Rumelhart et al. 1986).

Similarly, ERL methods often employ techniques such as neural networks to generalize the control policy. One difference, however, is that ERL methods do not use gradient descent algorithms, but rather allow the evolutionary algorithm to search for useful weights. In addition to neural networks, ERL decision policies have been successfully represented in symbolic rule sets (Grefenstette et al. 1990) and as Lisp S-expressions (Koza 1992). Each of these representations represents the decision policy as a function that maps features of states to actions, and updates based on single state observations influences many state and action mappings.

2.4.5 Policy Revisions

As agents experience different state and decision sequences, they must revise their current decision policy to account for the rewards returned by the environment. The question of how much reinforcement or how many rewards are necessary to revise the policy is an important one in reinforcement learning and one where the TD and ERL methods differ.

In a TD approach, the policy is normally revised after every decision. The revisions are small, since they are based only on a single reinforcement signal. An ERL approach normally makes revisions only after one or several agents have been completely evaluated over a sequence of decisions. The revisions are typically larger, since they are based on all of the reinforcement collected over the sequence of actions or only based on a single reward over the entire task.

Frequent policy updates based on local information can adversely affect generalization. Boyan and Moore (1995) describe experiments where the error in a function approximator applied to Q-learning actually diverges and becomes arbitrarily large. Boyan and Moore speculate that the updates based on a single state and decision pair in one situation can cause state and decision pairs in other situations to be under or overestimated. Such behavior occurs because updates to a function approximator normally affect policy decisions for many other states. If a single update is based only on a single observation, that observation can bias the global decision policy. In several examples, Boyan and Moore show how this phenomena can actually lead to convergence on the least optimal solution.

In ERL, policy changes are normally based on more global information. By updating less frequently and from information collected from several states and actions, it is more difficult for a reward from a single decision to significantly bias the decision policy. On the other hand, less frequent updates may cause the system to adapt slower to changes in the agent's environment. Adaptation is an important reinforcement learning issue in itself and is explored in the next section.

2.4.6 Adaptation

In many non-stationary environments, an agent must adapt its current decision policy in response to changes that occur in its world. Faulty sensors and effectors, new obstructions, and the presence of other adaptive agents are all examples of situations where the agent must revise many of its policy decisions. Policy revisions should occur quickly to avoid the costly effects of an outdated control policy. For example, if an agent makes packet routing decisions in a local area network and a network node goes down, packets that are routed through the down node will be delayed or lost. A new routing policy should be promptly established that avoids the faulty node. Neither field of reinforcement learning has directed significant attention on adaptation issues, however, it will likely become a central concern in future real world applications.

Because TD learning makes constant updates to the decision policy in response to each environmental signal, it should recognize domain changes as soon as they occur. Since ERL approaches do not normally realize the change in environment until an individual or a population of individuals have been completely evaluated over several actions, the adaptive policy revisions are delayed. However, since TD methods normally employ small learning rates to ensure convergence, it is unclear whether the immediate, but small responses will complete the adaptive revisions faster than a delayed, but larger changes.

In some situations, however, the revision delay in ERL methods may incur unacceptable costs or allow the system to enter states from which recovery is difficult. For example, in automobile traffic control an accident at a busy interchange necessitates immediate rerouting of traffic to prevent the problem from snowballing into an unmanageable mess. For ERL methods to handle such problems, efficient mechanisms for adaptive behavior based on immediate rewards need to be developed.

Several researchers have investigated the use of local learning after each action in an evolutionary algorithm (Grefenstette 1991; Gruau and Whitley 1993; Littman 1995; Nolfi and Parisi 1995). Such approaches have been termed Lamarckian if the local revisions are written back to the genetic chromosomes or Baldwinian if the revisions are not persevered. Local learning affords the evolutionary algorithm a quicker response to environmental shifts, however, it has not yet been applied to difficult sequential decision tasks.

2.4.7 Memory

The abstract goal of a reinforcement learning agent is to explore its world, gather statistics about states and decisions, and build an effective decision policy. So far, all of the described research issues have dealt with how the agent builds a decision policy. This section contrasts how the methods collect and maintain statistics about the agent's environment. Specifically, the memory or record of observed states and actions differs greatly between TD and ERL methods.

Temporal difference methods normally maintain a constant record of the reinforcement received from every state and decision pair. As states are revisited, the new reinforcement is combined with the previous value. New values thus supplement previous values, and the information content of the agent's reinforcement model increases during exploration. In this manner, TD methods sustain knowledge of both good and bad state and decision pairs.

In ERL, normally only the statistics of good states and decisions are maintained. The record is kept implicitly in the population of different policies. Knowledge of bad decisions is not preserved, since agents that make such decisions are removed from the population. Thus, a snapshot of the population reveals an implicit distribution of the profitable decisions from state descriptions. The most profitable decision from a given state description can be found by polling each policy in the population and returning the most popular choice. The second most profitable decision is the second most popular choice. More concretely, the selection rate of a decision from a given state description across a population of policies is proportional to its profitability.

In sharp contrast to the TD methods, the information content of ERL populations actually *decreases* as the population evolves. Evolutionary algorithms normally converge genes in the population to their presumably optimal values. In a sequential decision task, this often reflects convergence on a specific decision from a specific state description. For example in chess, the EA may recognize that it is very good to capture the opponent's queen whenever possible. Through selection and recombination, the EA will propagate this knowledge to each policy in the population and eventually every policy will take the queen whenever possible. In other words, as the EA finds the perceived best decision from a state, it often loses the statistics on other decisions from that state, because it removes individuals that make those decisions. In the chess example, since all policies capture the queen, nothing is maintained about the other legal moves in those situations. The statistics that were present before, in the form of a distribution of policy decisions from the queen capture state, are lost.

This memory degradation in ERL methods can produce poor performance if unexpected state transitions place the agent in a less frequented state. For example, in a game playing situation the opponent may make an unexpected move that transfers the game into an unfamiliar state. Since ERL methods converge policy decisions around expected state transitions, statistics on actions from less-frequented states are often lost. This places a heavier burden on the assumption that the agent's training environment accurately reflects the target environment. Since TD methods record and store all observances, they should retain knowledge of all experienced states and therefore be more robust in less frequented states. However, a function approximator, such as a neural network, applied to a TD method also suffers from memory lapses over rare states. As described in section 2.4.4, updates to a neural network representation of a decision policy affect many policy decisions. Since less frequented states produce fewer policy updates, their policy decisions may be overwritten or "averaged out" by the updates from the frequented states.

The practical implications of memory loss are largely unclear and are almost certainly an empirical question for each domain. While a complete memory method may ensure better behavior in less frequented states, the complexity costs of maintaining and searching every observance may outweigh any robustness gain. Future work is needed to examine this tradeoff and possibly discover a practical middle ground between robustness and efficiency.

2.5 Concluding Remarks

Researchers in ERL and TD learning are studying very different methods for solving sequential decision tasks. Both methods assume limited knowledge of the underlying system and both learn decision strategies by experimenting with different state and decision sequences. Both methods can learn from small amounts of reinforcement that provide some measure of proficiency in the task.

Neither require a precise mathematical model of the domain, and both may learn through direct interactions with the actual system.

An important advantage of ERL methods over TD methods, however, is a more robust credit assignment strategy. In situations where the sensory information does not completely describe the state of the system, ERL methods can implicitly disambiguate states. Since decisions are evaluated in a sequence, the EA can discover differences in states by considering the path to each state. TD methods evaluate decisions independently of the path and thus are more often misled by ambiguous states. This phenomena will be demonstrated in chapter 5 in a mobile robot task where knowledge of the path is essential to generate good behavior. Two ERL methods form effective decision policies, while a TD method fails.

Despite the different learning strategies, ERL and TD methods share many learning-related concerns. Specifically, state space generalization and hidden state identification are viewed by both communities as essential for large-scale applications. In this regard, practical tools for addressing these concerns should not be isolated to one method, but should benefit both. By pointing out the shared goals and concerns, I hope to motivate evolutionary algorithms for sequential decision learning in this dissertation as well as future work in ERL. Moreover, I hope these comparisons will promote further collaboration between the two communities.

Chapter 3

Symbiotic, Adaptive Neuro-Evolution

This chapter presents a new reinforcement learning method called SANE (Symbiotic, Adaptive Neuro-Evolution) that uses an evolutionary algorithm as its primary search engine. The chapter begins with motivation for a neural network representation of a decision policy followed by a description of how evolutionary algorithms and neural networks can be integrated to create a powerful approach to sequential decision learning. Several important research issues are outlined and subsequently addressed in the description of the SANE method. The goal of this chapter is to familiarize the reader with neuro-evolution and to present the neuro-evolution system developed in this dissertation to solve difficult sequential decision tasks. Portions of this chapter are taken from (Moriarty and Miikkulainen 1996a, 1996c).

3.1 Neural Computation for Sequential Decision Tasks

Research in artificial neural networks (ANN) and their applications has exploded in the past decade. The scope of ANNs has extended far beyond artificial intelligence or even computer science. Researchers in psychology, physics, and business finance, to name a few, have turned to ANN methods to solve problems and/or provide new models of behavior. Despite the overwhelming support for neural networks in many different problems and in many different fields, their adoption nonetheless must be well motivated. Specific to this dissertation, it is important to understand why neural networks present an attractive mechanism for representing a decision policy. The neural representation is motivated through three important attributes: compact and constant storage, constant computational time, and generalization.

The first advantage neural networks present is an efficient storage mechanism for the decision policy. Since the neural network performs the sensory to decision mapping, the actual decision policy is represented in a distributed fashion in the network's connections and weights. As more states and actions are observed, the new information is distributed and effectively merged with the old information in the weights. The size of the decision policy therefore does not grow unmanageably large with the agent's experiences, but remains constant. The neural network thus provides compact storage for a decision policy that may cover a very large space of input situations.

In addition to constant storage, neural networks provide constant computational time. The computational complexity is bounded by the number of neurons and connections within the network.

Since these components remain constant[1], the computation time also remains constant. This feature is very important in real time decision tasks when time spent generating the decision can decrease the performance of the system. Also, since neural networks are made up of many separate computational elements, they can be easily parallelized to further speed up the computation.

Perhaps the most important advantage of a neural network representation is effective generalization of the decision policy. Since policy decisions are globally distributed among the weights, single weight changes affect many subsequent policy decisions. Thus, a policy modification in one situation will automatically generalize to other situations. Moreover, since the storage space is finite the neural network must consolidate space by forming general policies based on features or ranges in the input space rather than exact input values. Generalization is important in large state spaces where the agent cannot realistically expect to experience every domain situation during training. By generalizing the decision policy, the neural network can apply decisions to unexperienced states based on common features with experienced states.

While the advantages are attractive, it is also important to consider the disadvantages of a neural network decision policy. First, there are numerous parameters that must be set *a priori* to ensure good behavior. The network architecture, the number of neurons, and the activation function, are three examples of parameters for which optimal values are not well understood. Implementation often involves several trial and error experiments to generate effective parameter settings. Second, since the decision policy is represented in the connections and weights, it is very difficult to extract the policy in a more readable and understandable form. A lucid representation may be necessary because it is to be implemented in some other manner or it is to be used to better understand the underlying system. Traditionally, implementors of neural networks must accept satisfaction solely from the outward behavior of the neural network, since the reasoning behind it is indecipherable. However, it is the position of this dissertation that the advantages of neural networks far outweigh the disadvantages.

3.2 Evolving Neural Networks

Recently there has been much interest in combining evolutionary algorithms and artificial neural networks (Kitano 1990; Koza and Rice 1991; Liu and Yao 1996; Moriarty and Miikkulainen 1994b; Schaffer et al. 1992; Whitley et al. 1990; Yao and Liu 1996). GA/NN combinations offer many important advantages over the more traditional neural network learning methods like backpropagation (Rumelhart et al. 1986) and cascade correlation (Fahlman and Lebiere 1990). This section describes and motivates the evolutionary approach for forming effective neural networks. The section also describes several important research issues in neuro-evolution that are addressed in this dissertation.

3.2.1 Overview

In neuro-evolution, an evolutionary algorithm is used to search for effective connections and/or connection weights within a neural network. The dynamic components (i.e. weights and/or archi-

[1]There are many neural network learning algorithms that do increase the number of neurons and connections during training and thus do not maintain constant complexity. The neural networks in this dissertation, however, have a fixed number of neurons and connections.

tecture) of the neural networks are encoded in structures that form the genetic chromosomes. A basic encoding method is to simply concatenate all of the connection weights on a single string. The string of weights becomes the genetic description of the neural network and is essentially the working material for evolution.

In most applications of neuro-evolution, a *population* of neural networks is maintained and evolved. Each neural network is evaluated and assigned a fitness score based on its performance in the task. The evolutionary algorithm then recombines and/or mutates structures of the best neural networks in the population to create new and hopefully better neural networks.

3.2.2 Why Evolve Neural Networks?

Evolutionary algorithms offer a much more flexible alternative to the traditional hill-climbing search methods. EAs are applicable to a much broader range of network architectures and do not require examples of correct behavior. Most current learning methods calculate errors in network output to serve as gradients for a hill-climbing search (e.g. backpropagation). Such methods require smooth, continuous activation functions from which gradient information can be easily derived. Since evolutionary algorithms do not use derivatives for credit assignment, other activation functions such as linear thresholds, splines, or product units may be used just as easily. In more complex architectures such as networks with recurrent connections, computing the gradient information necessary for hill-climbing is very costly (Williams and Zipser 1989). Since evolutionary algorithms do not rely on backpropagating error signals, evolving recurrent networks requires no extra computation over networks with no recurrent connections.

The primary motivation for neuro-evolution over the more standard techniques like backpropagation, however, is the ability to train under sparse reinforcement. In many domains, computing gradient information after every network output is infeasible because reinforcement from the system is infrequent. For example, in sequential decision tasks, a reinforcement signal may be given only after a sequence of decisions has been made. Training a neural network using backpropagation or other supervised learning methods in such a task requires credit assignment to each individual network output. As described in chapter 1, such assignments are very difficult to make based on an overall performance metric, because it is not always obvious how individual decisions affect the outcome. Since evolutionary algorithms do not require explicit credit assignment to individual network outputs, they can tackle a much wider range of problems including sparse reinforcement problems.

Recent research has shown neuro-evolution to be competitive in training time with standard gradient descent training of neural networks. Montana and Davis (1989) ran several experiments comparing evolutionary neural networks to standard backpropagation networks in the task of classifying passive sonar data from arrays of underwater acoustic receivers. Two different implementations of neuro-evolution were used: one where the weights of a feed-forward network were evolved using an evolutionary algorithm alone, and one where an evolutionary algorithm was implemented together with backpropagation in a hybrid approach. Montana and Davis report no clear advantage using the hybrid approach over the evolutionary algorithm alone and report superior results in the number of training iterations using the evolutionary algorithm over backpropagation.

Potter (1992) used an evolutionary algorithm in place of the quickprop learning method (Fahlman 1988) in the cascade correlation architecture (Fahlman and Lebiere 1990), which is one of the fastest known neural network training algorithms. The genetic form of cascade correlation required fewer epochs in the two-spirals benchmark and slightly more epochs in the eight-bit parity problem than the standard quickprop cascade correlation. Potter points out that the results are encouraging since the genetic cascade correlation performed comparably in a domain where gradient information was readily available. In domains with more sparse reinforcement, genetic cascade correlation should continue to form good networks, whereas standard cascade correlation may be unable to form effective targets from the sparse reinforcement signals.

3.2.3 Research Issues

The results discussed above are very encouraging and motivate further applications of neuro-evolution. However, several improvements can be made in current neuro-evolution techniques to allow them to scale well to much harder problems. Two important issues are outlined here that are directly addressed in the SANE neuro-evolution system.

Network Encoding

The first issue concerns the representation or encoding of the networks in chromosomes. Most neuro-evolution approaches fix the architecture to be evolved and the chromosome merely reflects the concatenation of the network's weights (Belew et al. 1991; Jefferson et al. 1991; Werner and Dyer 1991; Whitley et al. 1990). Fixing the architecture and forcing weights to correspond directly to their chromosome location or *locus* inhibits much of the flexibility of the evolutionary algorithm approach. For example, it is very difficult to build structures of highly-fit weights if the weights are located in distant regions of the chromosome. By fixing the weights to a specific locus, a bias is introduced regarding which weights will be combined in useful building blocks: the weights that lie in close proximity to each other. Such a bias restricts the freedom of the evolutionary algorithm to explore many different useful building blocks and can significantly increase the search time.

Convergence

The second research issue in neuro-evolution concerns convergence of the population. Because evolutionary algorithms continually select and breed the best individuals in the population, populations normally lose diversity and eventually converge around a single "type" of individual (Goldberg 1989). Such convergence is undesirable for two reasons: (1) populations often converge on suboptimal peaks and (2) converged populations cannot adapt well to changes in the task environment. While these two problems may have limited effect in standard function optimization, they have very large consequences when evolving decision strategies in complex and dynamic domains.

The importance of population diversity throughout evolution cannot be overemphasized. Diversity promotes quick exploration of the solution space. An evolutionary algorithm in a converged population can normally only proceed by randomly mutating the single solution representation, which produces a very slow and inefficient search. A genetic search with a diverse population, however, can continue to utilize recombination to generate new structures and make larger traversals of the solution space in shorter periods of time. In many complex problems, search efficiency is paramount for generating effective decision policies in a timely manner. Such an advantage is equally important when changes occur in the domain. Often in decision tasks in dynamic or unstable environments, policies must be quickly revised to avoid costly effects. A diverse population can more adeptly make the modifications necessary to compensate for the domain changes.

3.2.4 Maintaining Population Diversity

It is clear that a convergent evolutionary algorithm is the wrong kind of search strategy for reinforcement learning. Maintaining diverse populations, however, is very difficult and remains an open research issue in the evolutionary algorithms community. The most common method for maintaining diversity is to use a less aggressive genetic selection strategy or a high rate of mutation. Weak selection strategies do not ensure diversity, but rather slow evolution and delay the convergence of the population. Slower evolution can help prevent premature convergence, but often at the expense of slower searches. Chapter 4 will present experiments that demonstrate this phenomenon. The second strategy, increasing the mutation rate, only artificially injects diversity into the population through noise. Despite their obvious disadvantages, these two methods generally produce better search behavior than an aggressive, convergent EA, and their adoption has unfortunately become commonplace. The system developed in this dissertation will demonstrate that aggressive selection and recombination strategies can work well if tempered with effective diversity pressures.

Several more intelligent methods have been developed to enforce population diversity, including fitness sharing (Goldberg and Richardson 1987), crowding (De Jong 1975), and local mating (Collins and Jefferson 1991). Each of these techniques relies on external genetic functions that prevent convergence of the genetic material. The diversity assurances, however, are achieved through very expensive operations. For example, in Goldberg's fitness sharing model, similar individuals are forced to share a large portion of a single fitness value from the shared solution point. Sharing decreases the fitness of similar individuals and causes evolution to select against individuals in overpopulated niches. While fitness sharing is effective at maintaining diversity, it incurs a heavy computational expense. Sharing requires $O(n^2)$ similarity comparisons each generation, where n is the size of the population. In large populations with large chromosomes, comparison-based diversity methods such as sharing, crowding, and local mating are simply not practical (Smith et al. 1993).

A more recent technique for ensuring diversity has been termed *implicit fitness sharing* (Horn et al. 1994; Smith et al. 1993). In implicit fitness sharing, no comparisons are made between individuals. Instead, diversity pressures are built into the task through cooperative behavior among the individuals in the population. The important feature of implicit fitness sharing is that individuals no longer represent complete solutions to the problem. Individuals represent only partial solutions and must cooperate with other individuals to form a full solution. By reducing the capacity of individuals and *coevolving* them together, evolution searches for several different types of individuals that together solve the problem. Implicit fitness sharing also presents a very nice side effect. While evolution searches for individuals that optimize different aspects of the problem, it effectively performs several parallel searches in decompositions of the solution space, which can greatly speed up evolution.

The core evolutionary strategy of the SANE neuro-evolution system, presented in the next section, borrows many ideas from the implicit fitness sharing model. Specifically, the notion of cooperating individuals that represent only partial solutions is incorporated in SANE to promote

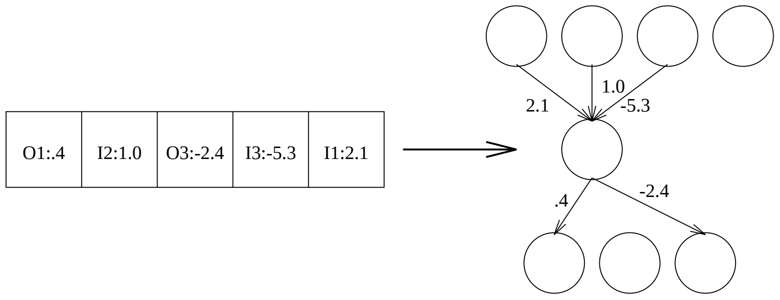


Figure 3.1: An individual in SANE's population and the hidden neuron it defines. Hidden neurons are defined by a series of weighted connections to be made from the input layer or to the output layer. For example, the first gene specifies a connection to the first output unit with a weight of 0.4. The encoding shown is a simplified form of SANE's actual neuron encoding, which is described in section 3.3.3.

diversity and search efficiency. Several changes, however, were necessary to tailor implicit fitness sharing to neuro-evolution. These changes are highlighted in the next section and in the related work chapter.

3.3 The SANE Approach

The endeavor of this dissertation is to study evolutionary algorithms and neural networks as methods for learning and performing difficult decision tasks. This section presents the main experimental vehicle called SANE developed for this purpose. SANE is designed as a sophisticated methodology for evolving decision-making neural networks in difficult problems. The description of SANE is divided into two main sections. Since SANE is quite different from most approaches to neuro-evolution, an overview is first presented along with proper motivation. The second section provides a more detailed description of the specific neural and genetic mechanisms.

3.3.1 Evolving Symbiotic Neurons

SANE's greatest departure from standard neuro-evolution is perhaps its most important contribution. In almost all approaches to neuro-evolution, each individual in the genetic population represents a complete neural network that is evaluated independently of other networks in the population (Belew et al. 1991; Koza and Rice 1991; Nolfi and Parisi 1992; Whitley et al. 1993). As described in section 3.2.3, by treating each member as a separate, full solution, the evolutionary algorithm focuses the search towards a single dominant individual. Such concentration can greatly impede search progress in both complex and dynamic tasks. In contrast, the SANE method restricts the scope of each individual to a single neuron. More specifically, each individual represents a hidden neuron in a 2-layer neural network (figure 3.1). In SANE, complete neural networks are built by combining several neurons. Figure 3.2 illustrates the difference between standard neuro-evolution and the neuro-evolution performed in SANE.

Since no single neuron can perform the whole task alone, the neurons must optimize one aspect of the neural network and connect with other neurons that optimize other aspects. Evolu-

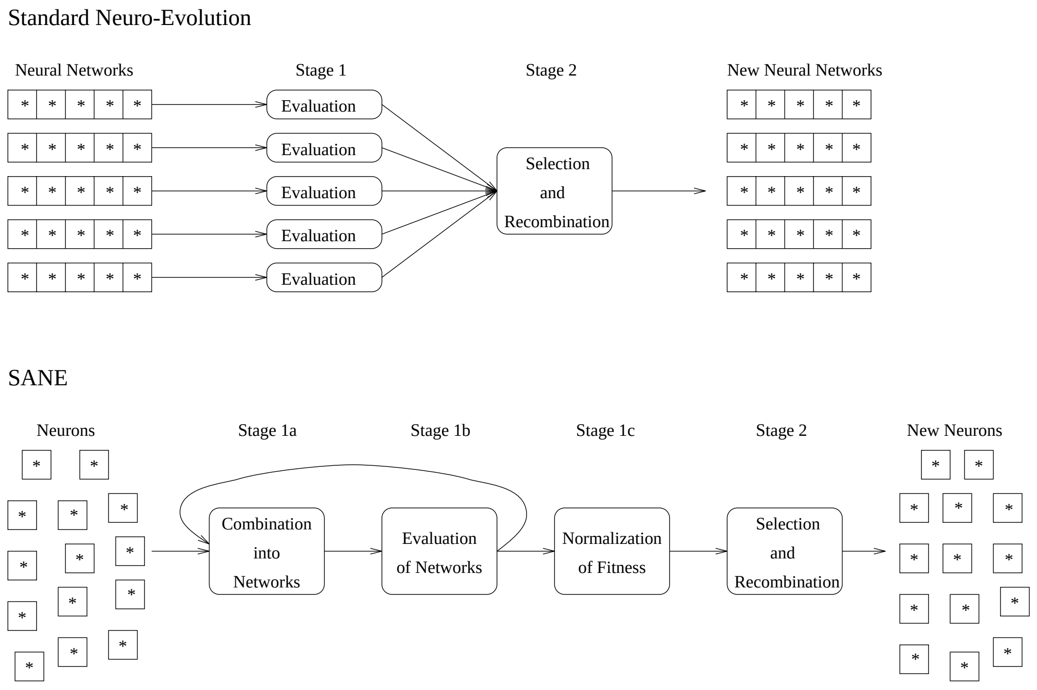


Figure 3.2: An illustration of the neuro-evolution performed in SANE compared to the standard approach to neuro-evolution. The standard approach maintains a population of neural networks and evaluates each independently. SANE maintains a population of neurons and evaluates each in conjunction with other neurons. Step 1 (the evaluation step) in SANE is broken into three substeps. Neurons are continually combined with each other and the resulting networks are evaluated in the task. Each neuron receives a normalized fitness based on the performance the networks in which it participates.

tionary pressures therefore exist to evolve several different types or *specializations* of neurons.[2] In this way, the neurons will form a *symbiotic* relationship. It follows that the evolution performed in SANE can be characterized as *symbiotic evolution*. I define symbiotic evolution as a type of co-evolution where individuals explicitly cooperate with each other and rely on the presence of other individuals for survival. Symbiotic evolution is distinct from most coevolutionary methods, where individuals compete rather than cooperate to survive. A more detailed discussion of the relationship between symbiotic and competitive coevolution is presented in the related work chapter.

The advantages of symbiotic evolution are twofold. First, the neuron specializations ensure diversity which discourages convergence of the population. A single neuron cannot “take over” a population since to achieve high fitness values, there must be other specializations present. If a specialization becomes too prevalent, its members will not always be combined with other specializations in the population. Thus, redundant partial solutions do not always receive the benefit of other specializations and will incur lower fitness evaluations. Evolutionary pressures are therefore present to select against members of dominant specializations. This is quite different from standard evolutionary approaches, which always converge the population, hopefully at the global optimum, but often at a local one. In symbiotic evolution, solutions are found in diverse, *unconverged* popula-

[2]I chose the term specialization rather than species since each neuron does not represent a full solution to the problem.

tions. By maintaining diverse populations, SANE can continue to use its recombination operators to build effective neural structures.

In addition to maintaining diverse populations, evolution at the neuron level more accurately evaluates the genetic building blocks. In a network-level evolution, each neuron is implemented only with the other neurons encoded on the same chromosome (e.g. figure 3.2). With such a representation, a very good neuron may exist on a chromosome but be subsequently lost because the other neurons on the chromosome are poor. In a neuron-level evolution, neurons are continually recombined with many different neurons in the population, which produces a more accurate evaluation of the neural network building blocks.

Essentially, a neuron-level evolution takes advantage of the *a priori* knowledge that individual neurons constitute basic components of neural networks. A neuron-level evolution *explicitly* promotes genetic building blocks in the population that may be useful in building other networks. A network-level evolution does so only *implicitly*, along with various other sub- and superstructures (Goldberg 1989). In other words, by evolving at the neuron level the evolutionary algorithm is no longer relied upon to identify neurons as important building blocks, since neurons are the object of evolution.

3.3.2 Maintaining Effective Neuron Collections

The neuron evolution alone, however, is not sufficiently powerful to generate the complex networks necessary in difficult tasks. Knowledge of the useful combinations of neurons must be maintained and exploited. Combining neurons without such intelligent direction is undesirable for two reasons. First, the neurons may not be combined with neurons that work well together. Thus, a very good neuron may be lost, because it was ineffectively combined during a generation. The second problem is that the quality of the networks varies greatly throughout evolution. In early generations this works as an advantage, since the search produces many different types of networks to find the most effective neurons. However, in later generations, when the search should focus on the best networks, the inconsistent networks often stall the search and prevent the global optima from being located.

An outer loop mechanism is necessary to maintain knowledge of the good neuron combinations. Many different approaches could perform the necessary record keeping ranging from maintaining complex tables which record the fitness of each neuron when combined with other neurons to simply remembering the top neuron combinations of the previous generation. Clearly, the memory requirements of the first method make it impractical. For example, if 8 neurons were used to build a network from a population of 800 neurons, a complete record of all neuron combinations would contain 800^8 entries. Conversely, the second method maintains very little information of the history of each neuron and may provide only limited benefit. A solution in between the two seems appropriate.

The current method of maintaining useful neuron combinations in SANE is to evolve a layer of neural network records or *blueprints* on top of the neuron evolution. The blueprint population maintains a record of the most effective neuron combinations found with the current population of neurons and uses this knowledge as the basis to form the neural networks in the next generation. Figure 3.3 shows the relationship between the blueprint and neuron populations. Each blueprint specifies a collection of neurons that have performed well together. SANE uses genetic selection and recombination to exploit this knowledge and hopefully form even better collections of neurons

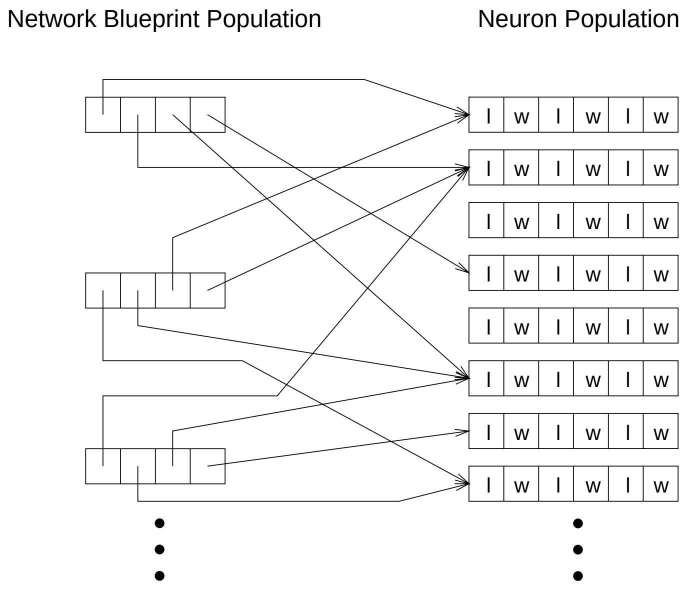


Figure 3.3: An overview of the network blueprint population in relation to the neuron population. Each member of the neuron population specifies a series of connections (labels and weights) to be made from the input layer or to the output layer within a neural network. Each member of the blueprint population specifies a series of pointers to specific neurons which are used to build a neural network. The neuron population searches for effective partial networks, while the blueprint population searches for effective combinations of partial networks.

in subsequent generations.

Maintaining network blueprints produces more accurate neuron evaluations and concentrates the search on the best neural networks. Since neurons are connected systematically based on past performance, they are more consistently combined with other neurons that perform well together. Additionally, better-performing neurons garner more pointers from the blueprint population and thus participate in a greater number of networks. Biasing the neuron participation towards the historically better-performing neurons provides more accurate evaluations of the top neurons. The sacrifice, however, is that newer neurons may not receive enough trials to be accurately evaluated. In practice, allocating more trials to the top neurons produces a significant improvement over uniform neuron participation.

The primary advantage of evolving network blueprints, however, is the exploitation of the best networks found during evolution. SANE maintains the proficient collections of neurons in the blueprint chromosomes and ensures that the best networks are reconstructed. By *evolving* the blueprint population, the best neuron combinations are also recombined to form new, potentially better, collections of neurons. The blueprint level evolution thus provides a very exploitive search that can build upon the best networks found during evolution and focus the search in later generations.

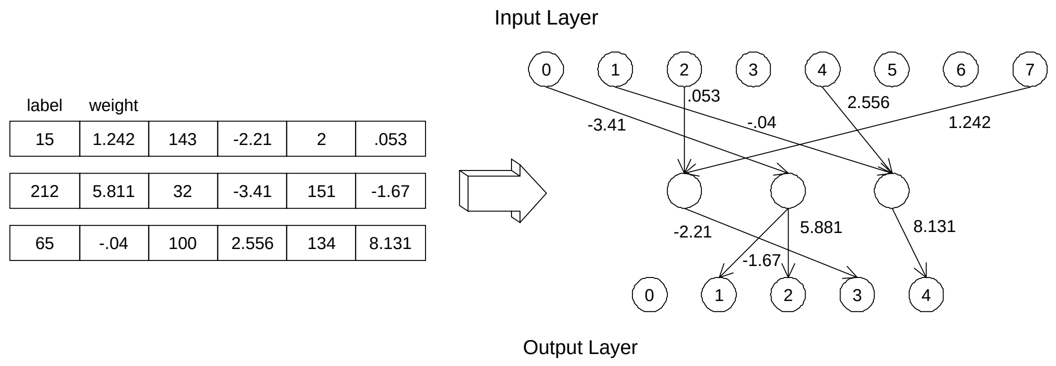


Figure 3.4: Forming an 8 input, 3 hidden, 5 output unit neural network from three hidden neuron definitions. The chromosomes of the hidden neurons are shown to the left and the corresponding network to the right. In this example, each hidden neuron has 3 connections.

3.3.3 SANE Implementation

This section provides the implementation details of the SANE system.[3] SANE maintains and evolves two populations: a population of neurons and a population of network blueprints. Each individual in the neuron population specifies a set of connections to be made within a neural network. Each individual in the network blueprint population specifies a set of neurons to include in a neural network. Conjunctively, the neuron evolution searches for effective partial networks, while the blueprint evolution searches for effective combinations of the partial networks.

Each individual in the neuron population represents a hidden neuron in a 2-layer feedforward network. Neurons are defined in bitwise chromosomes that encode a series of connection definitions, each consisting of an 8-bit label field and a 16-bit weight field. The absolute value of the label determines where the connection is to be made. The neurons only connect to the input and the output layer. If the decimal value of the label, D, is greater than 127, then the connection is made to output unit D mod O, where O is the total number of output units. Similarly, if D is less than or equal to 127, then the connection is made to input unit D mod I, where I is the total number of input units. The weight field encodes a floating point weight for the connection. Figure 3.4 shows how a neural network is formed from three sample hidden neuron definitions.

Each individual in the blueprint population contains a series of neuron pointers. More specifically, a blueprint chromosome is an array, of size ζ, of address pointers to neuron structures. Figure 3.3 illustrates how the blueprint population is integrated with the neuron population. Initially, the chromosome pointers are randomly assigned to neurons in the neuron population. During the neuron evaluation stage, subpopulations of neurons are selected based on each blueprint's array of pointers.

Since SANE operates on bit strings and uses both mutation and recombination, its search strategy fall under the *genetic algorithm* (Holland 1975; Goldberg 1989) method of evolutionary computation. The basic steps in one generation of SANE are as follows (listed table 3.1): during the evaluation stage, each blueprint is used to select neuron subpopulations of size ζ to form a neural network. Each blueprint receives the fitness evaluation of the resulting network and each neuron

[3]The source code can be obtained from the UTCS Neural Networks' home page: http://www.cs.utexas.edu/users/nn/

1.	Clear fitness value of each neuron and blueprint.
2.	Select ζ neurons from the population using a blueprint.
3.	Create a neural network from the selected neurons.
4.	Evaluate the network in the given task.
5.	Assign the blueprint the evaluation of the network as its fitness.
5.	Repeat steps 2-4 for each individual in the blueprint population.
7.	Assign each neuron the evaluation of the best 5 networks in which it participated.
8.	Perform crossover and mutation operations on the both populations.

Table 3.1: The basic steps in one generation of SANE.

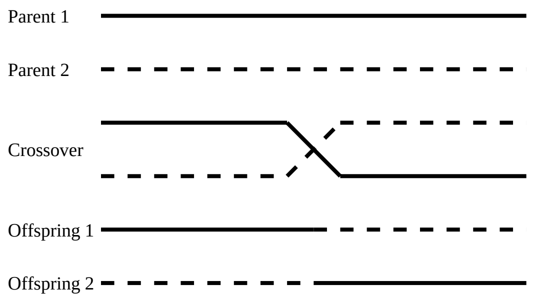


Figure 3.5: A one-point crossover operation. In SANE, only one of the children (chosen at random) enter the population. The other is thrown away.

receives the summed fitness evaluations of the *best five* networks in which it participated. Calculating fitness from the best five networks, as opposed to all of the neuron's networks, discourages selection against neurons that are crucial in the best networks, but ineffective in poor networks. For example in a robot arm manipulation task, a neuron that specializes in small movements near the target would be effective in networks that position the arm close to the target, but useless in networks that do not get anywhere near the target. Such neurons are very important to the population and should not be penalized for the poor networks that they cannot help.

After the evaluation stage, the neuron population is ranked based on the fitness values. For each neuron in the top 25% of the population, a mate is selected randomly among the top 25%. Each mating operation creates two offspring: a child created through a one-point crossover operation and a copy of one of the parent chromosomes. Figure 3.5 illustrates how the one-point crossover operation recombines structures from two parent chromosomes to create two offspring chromosomes. In SANE, one of the offspring produced by crossover is chosen at random to enter the population. Copying one of the parents as the second offspring reduces the effect of adverse neuron mutation on the blueprint-level evolution. This effect will be further explained in the context of the blueprint evolution later in this section. The two offspring replace the worst-performing neurons (according to rank) in the population. Mutation at the rate of 0.1% per bit position is performed on the entire population as the last step in each generation. Such an aggressive, elitist breeding strategy is not normally used in evolutionary applications, since it leads to quick convergence of the population. SANE's neuron evolution, however, performs quite well with such an aggressive selection strategy, since it contains strong evolutionary pressures against convergence.

In the blueprint population, since the chromosomes are made up of address pointers instead of bits, crossover only occurs between pointers. The new offspring receive the same address pointers that the parent chromosomes contained. In other words, if a parent chromosome contains a pointer to a specific neuron, one of its offspring will point to that same neuron (barring mutation). The current evolutionary algorithm on the blueprint level is identical to the aggressive strategy used at the neuron level, however the similarity is not essential and a more-standard evolutionary algorithm or other methods of evolutionary computation could be used. Empirically, the aggressive strategy at the blueprint level coupled with the strong mutation strategy described below, has outperformed many of the more-standard evolutionary algorithms.

To avoid convergence problems at the blueprint level, a two-component mutation strategy is employed. First, a pointer in each offspring blueprint is randomly reassigned to another member of the neuron population at a rate of 1%. This strategy promotes participation of neurons other than the top neurons in subsequent networks. Thus, a neuron that does not participate in any networks can acquire a pointer and participate in the next generation. Since the mutation only occurs in the blueprint offspring, the neuron pointers in the top blueprints are always preserved.

The second mutation component is a selective strategy designed to take advantage of the new structures created by the neuron evolution. Recall that a breeding neuron produces two offspring: a copy of itself and the result of a crossover operation with another breeding neuron. Each neuron offspring is thus similar to and potentially better than its parent neurons. The blueprint evolution can use this knowledge by occasionally replacing pointers to breeding neurons with pointers to offspring neurons. In the experiments described in this paper, pointers are switched from breeding neurons to one of their offspring with a 50% probability. Again, this mutation is only performed in the offspring blueprints, and the pointers in the top blueprints are preserved.

The selective mutation mechanism described above has two advantages. First, because pointers are reassigned to neuron offspring that are the result of crossover, the blueprint evolution can explore new neuron structures. Second, because pointers are also reassigned to offspring that were formed by copying the parent, the blueprints become more resilient to adverse mutation in the neuron evolution. If pointers were not reassigned to copies, many blueprints would point to the same exact neuron, and any mutation to that neuron would affect every blueprint pointing to it. When pointers are occasionally reassigned to copies, however, such mutation is limited to only a few blueprints. The effect is similar to schema promotion in standard evolutionary algorithms. As the population evolves, highly fit schema (i.e. neurons in this case) become more prevalent in the population, and mutations to one copy of the schema do not affect other copies in the population. The advantages of selective mutation will be demonstrated in section 4.4.

3.4 Concluding Remarks

The SANE neuro-evolution system is specifically designed to tackle difficult sequential decision tasks. SANE contains a very aggressive evolutionary algorithm that forms neural network representations of decision policies efficiently and with little domain information. SANE produces effective diversity pressure and searches decompositions of the neural network space, which further reduces training time. In summary, SANE is an efficient decision learning method applicable to many difficult problems. The remainder of this dissertation will extensively test this hypothesis through analysis of SANE's evolution, benchmark comparisons, and application to previously unoptimized decision problems.

Chapter 4

Evaluation and Analysis

The evolutionary search in SANE is quite different from those of standard neuro-evolution, and it is important to understand how the individual neurons work together to form a global solution. This chapter examines the internal mechanisms of SANE through comparisons with more standard neuro-evolutionary approaches and empirical analysis of the developing neuron populations. The first experiments show some of the advantages of SANE's symbiotic search strategy in terms of search efficiency, diversity, and adaptability. The second experiments illustrate the specializations within SANE's population and uncover some of the different roles that the neurons assume in the networks.

4.1 The Khepera Robot Simulator

The domain chosen for initial evaluation of the SANE method was mobile robotics, or more specifically, controlling the Khepera mobile robot (Mondada et al.1993).[1] Robotics is a natural application for SANE. Many robot tasks such as navigation, sensory mapping, and kinematics approximation are very difficult to learn because the domain knowledge is normally not sufficient to provide targets for each action. By casting these problems as sequential decision tasks, SANE can form neural networks to control a wide range of robot behaviors.

A full discussion of SANE's contribution to robotics is deferred to chapter 7, where an application of SANE to the OSCAR-6 robot arm is presented. Similarly, comparisons of related work and alternate methods of reinforcement learning in the Khepera task are postponed until chapter 6. For the purposes of this chapter, the Khepera robot task is strictly viewed as a tool to understand SANE's symbiotic search mechanism.

Michel (1995) has developed a simulator of the Khepera robot, which contains useful X window utilities for visualizing neural network controllers. Network architectures and activations can be viewed during the simulation along with the activation of the robot's sensors and motors. These utilities, along with the real world sensory input and motor output, make the Khepera simulator an excellent utility to evaluate many of the features and components of SANE

Khepera is a tiny robot (5 cm diameter) designed for research and teaching purposes. Khepera contains both infrared and light sensors positioned around its circumference. Figure 4.1 shows

[1]Information on Khepera as well as the Khepera simulator can be found at http://wwwi3s.unice.fr/ om/khep-sim.html.

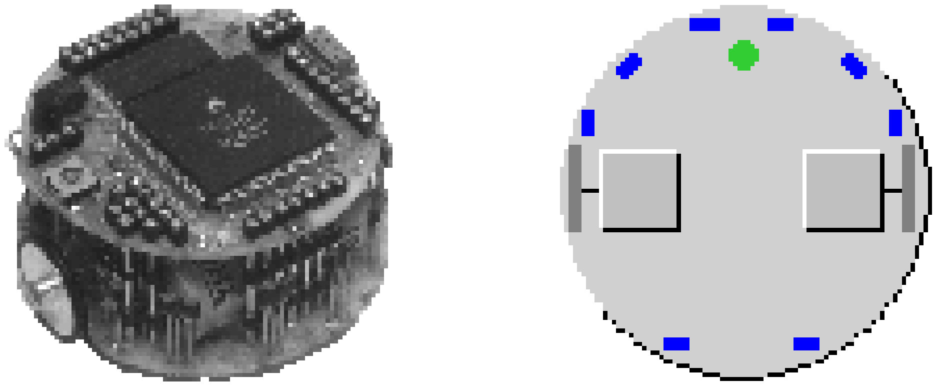


Figure 4.1: A picture of the actual Khepera robot (left) and Khepera simulation. The pictures were reproduced with permission from Olivier Michel.

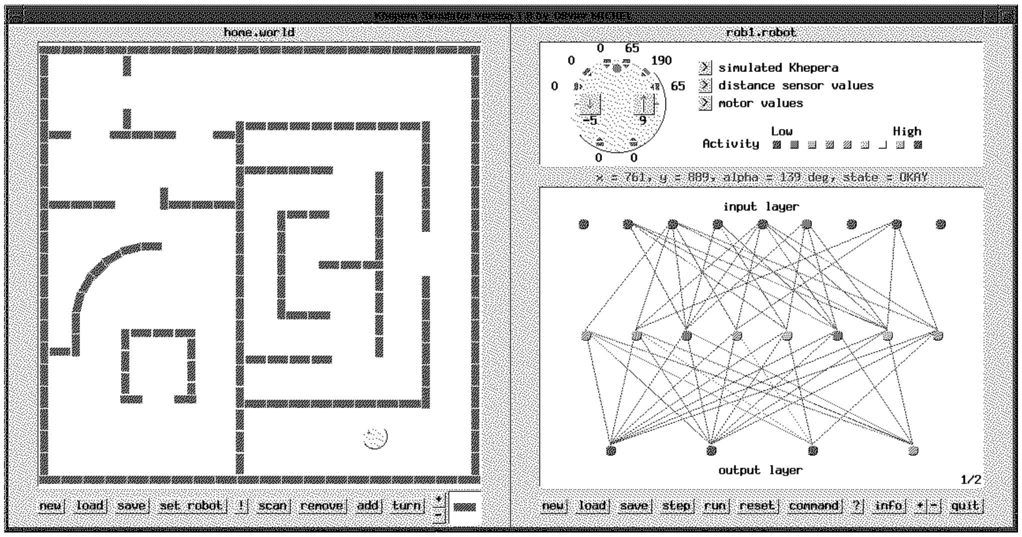
a picture of an actual Khepera robot and the simulated robot. Despite its size, the robot is not easy to control. Khepera provides real world sensory information and requires a strong grounding to the motor outputs to effectively maneuver the robot (Mondada et al. 1993).

The I/O resources of the simulator were designed to accurately reflect those of the real robot. The eight infrared sensors detect the proximity of objects by light reflection and return values between 0 and 1023 depending on the color level. A value of 0 indicates that no object is sensed, and a value of 1023 indicates an object almost touching the sensor. Khepera's light sensors measure the amount of ambient light around the object, however, these values were not used in the experiments. Khepera has two wheels, controlled by two separate motors, which can receive speed ranges from -10 to 10. If the two motors output equal speeds, the robot will move in a straight line. Otherwise, the robot will rotate.

Figure 4.2 shows a snapshot of the simulator window and the layout of Khepera's world.[2] The Khepera robot was placed in the world with the following goal: within a specific allotted time, move as far away (in Euclidean distance) from your starting position as possible without colliding with obstacles. Thus, an effective controller must accurately translate the infrared sensory information into specific motor outputs to both move the robot forward (or backward) and maneuver the robot around obstacles. The controller must balance aggressive acceleration with careful sensory processing. Driving the robot's motors in high gear may achieve great distances on wide open straightaways, but will perform poorly in tight enclosers. Conversely, a controller too conservative may make the tight corners, but will not generate enough speed to achieve a great distance in the allotted time. The best controllers must vary their speed depending on the value of the sensors.

The Khepera robot and task provide an effective domain for initial evaluation and analysis of SANE. The sensors and actuators reflect real world resources, and the task requires a complex balance of control. The remainder of this chapter will focus on the specific evaluation experiments performed with Khepera to foster a deeper understanding of SANE and its search mechanism.

[2]The specific world that was used was the lab0.world from the Khepera 1.0 simulator package.


Khepera Simulator version 1.0 by Olivier MICHEL
home.world
rob1.robot
simulated Khepera
distance sensor values
motor values
Low
High
Activity
input layer
output layer
1/2
new load save set robot ! scan remove add turn
new load save step run reset command ? info +|- quit

```
x = 761, y = 889, alpha = 139 deg, state = OKAY
```

Figure 4.2: The interface to the Khepera 1.0 simulator (Michel 1995). The window shows a view of Khepera and the configuration of its artificial world.

4.2 Evaluation of Symbiotic Evolution

The first experiments were designed to evaluate the merits of symbiotic evolution. Comparisons were run against more standard neuro-evolution techniques using the same neural network encoding strategy as SANE. The experiments were designed to test the following hypotheses:

- SANE's symbiotic search is more efficient than standard genetic algorithms in neuro-evolution.
- SANE maintains diverse populations throughout evolution.
- SANE is more adaptive than standard neuro-evolution.

4.2.1 Experimental Setup

Four evolutionary approaches were tested in the Khepera simulator: SANE, a standard neuro-evolution approach using the same aggressive selection strategy as SANE, a standard neuro-evolution approach using a less aggressive tournament selection strategy, and a version of SANE without the network blueprint population.

The standard neuro-evolution approaches evolve a population of neural networks. Each individual's chromosome consisted of 8 concatenated neurons, which are encoded in the same fashion as SANE's neurons. The difference in the two standard approaches is in the underlying genetic selection strategies. The first approach, which will be called *Standard Elite*, uses the same aggressive selection strategy used in SANE (described in section 3.3.3). Thus, the only difference between SANE and standard elite is the level of evolution. SANE performs evolution on the neuron level and the standard elite approach on the network level. The neuron encoding and genetic algorithm are identical. Comparisons of SANE to the standard elite approach demonstrate how the diversity pressures in the neuron evolution allow for very aggressive searches that perform very poorly in normal genetic algorithms because of premature convergence.

	SANE	Neuron SANE	Standard Elite	Standard Tournament
Neuron Population	800	200	-	-
Network Population	100	-	100	100

Table 4.1: Implementation parameters for each method.

The second standard neuro-evolution approach uses a less aggressive tournament selection strategy and will be called *Standard Tournament*. In a binary tournament selection, two random random members are selected from the population, and the member with a higher fitness is used for recombination. The standard tournament approach uses a binary tournament to select each parent in the population for recombination. Tournament selection creates a selection bias towards the top individuals but does not preclude recombination of poor individuals. Contrasted with SANE's elitist approach, where the top 10% of the population participates in over half of the recombination operations, binary tournament selection is much less aggressive. Recent research has shown tournament selection to be the preferred method of genetic selection in terms of its growth ratios for discouraging premature convergence (Goldberg and Deb 1991). Comparisons of SANE to the standard tournament neuro-evolution approach demonstrate the performance of the symbiotic search relative to a more "state of the art" genetic search strategy.

The fourth evolutionary approach is a symbiotic search without the higher-level blueprint evolution. This approach is called *Neuron SANE*. Instead of using a population of network blueprints to form the neural networks, Neuron SANE forms networks by randomly selecting subpopulations of neurons. Comparisons of SANE to Neuron SANE can thus effectively gauge the contribution of the blueprint-level evolution.

To focus the comparison on the different strategies of neuro-evolution, rather than the choice of parameter settings, several preliminary experiments were run to discover effective parameter values for each approach. Table 4.1 summarizes the parameter choices for each method. With the standard approaches, a population size of 100 networks was found to be more effective than populations of 50 or 200. Keeping the number of network evaluations per generation constant across each approach, a neuron population size of 800 for SANE and 200 for Neuron SANE performed well. A 0.1% mutation rate was used for all four approaches.

Neuron SANE requires a smaller population than SANE because its neurons are evaluated through random combinations. The population must be small enough to allow neurons to participate in several networks per generation. For example, randomly selecting 8 neurons for 100 networks in a 200 neuron population gives each neuron an expected network participation rate of 4 networks per generation. In SANE, the neuron population is not as restricted, since neuron participation is dictated by the network blueprints. SANE skews the participation rate towards the best neurons and leaves many neurons unevaluated in each generation. An unevaluated neuron is normally garbage, since no blueprint uses it to build a network.

Three experiments were run to compare the four approaches: a performance analysis, a diversity analysis, and an adaptive analysis. The results of each of the experiments were averaged over 20 simulations.

	> 100	> 150	> 200	> 250	> 300
SANE	1	6	14	26	41
Neuron SANE	1	8	14	34	64
Standard Elite	3	13	37	65	-
Standard Tournament	2	10	21	40	79

Table 4.2: The average number of generations to reach the desired level of distance over the 50 position test set. For example, SANE required 26 generations on average to generate a network that averaged over 250 cm of distance on the test set. SANE's evolution was the most efficient requiring half of the evaluations of the standard approaches to reach the top level of performance.

4.2.2 Performance Analysis

The first experiment tested the learning speed and solution quality of each evolutionary approach. Populations were evolved in the Khepera simulator for 80 generations. During each network evaluation, the robot was placed in a random position in the Khepera world, and the network was allowed to move the robot until it hit an obstacle or the maximum number of moves, 200, was exhausted. The fitness of each network was the maximum Euclidean distance the robot moved from its starting position.

Since each network was evaluated over a single trial, the evaluation may be inherently noisy. A good network may receive a poor evaluation simply because the robot started in a tight encloser. On the other hand, allocating more trials per evaluation delays recombination and mutation operations which slows the search. In practice, I have found that it is more efficient to allocate fewer trials per evaluation and thereby increase the frequency of this genetic operations. This methodology is adopted for all of the simulations presented in this dissertation.

To generate a learning curve, the best network of each generation (according to fitness) was tested on a 50 start-position test set. The average distance on the test set defines the performance of that generation. For each generation, the learning curve plots the best performance found at or before that generation. The learning curve thus shows the quality of solution that can be expected by each generation.

Figure 4.3 shows the learning curve from the performance analysis and table 4.2 shows the average number of generations to reach the specific levels of distance over the test set. As expected, the standard approach with the elite, aggressive selection strategy performed poorly. The search was inconsistent; it either found very good networks or stalled with very poor networks, which is characteristic of an aggressive, convergent search. If the search converged on a good network, it could often tweak it with mutation into a great network. Otherwise, it remained stuck with a suboptimal solution. These experiments confirmed that for a network level evolution, tournament selection is a better selection strategy.

SANE performed the most efficient search, finding networks that averaged 300 cm of distance in half as many generations as the standard tournament search and less than half of the generations of the standard elite search. Unlike the standard elite approach, SANE's aggressive searches were very consistent. Only one of the SANE simulations out of the 20 returned a final network that averaged less than 300 cm. SANE's neuron-based searches, thus, do not appear as susceptible to premature convergence as an aggressive evolutionary algorithm operating on a population of neural networks.

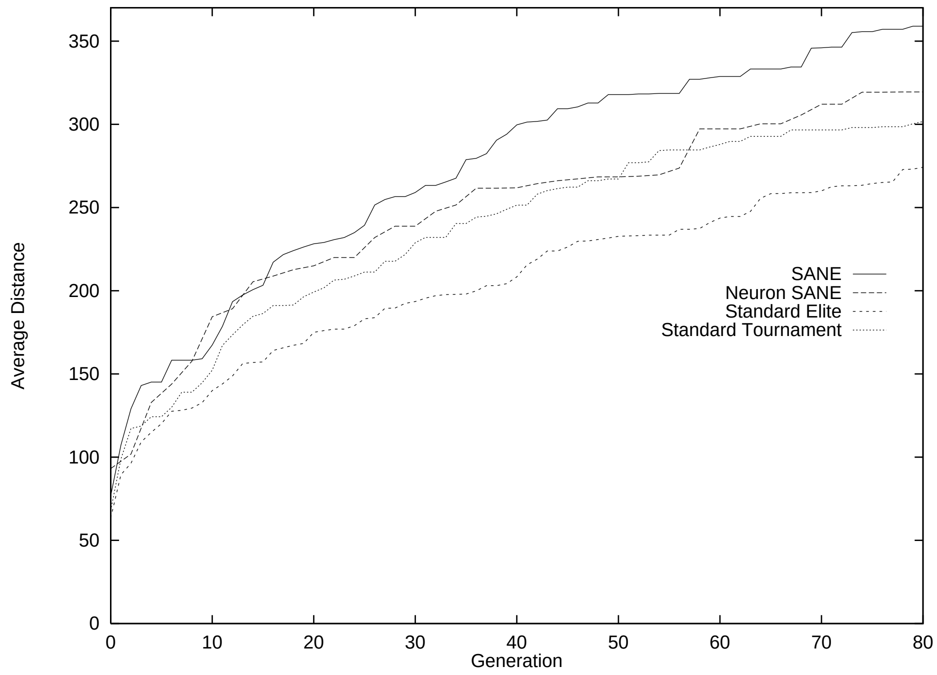


Figure 4.3: Comparison of the learning speeds of the different evolutionary approaches. The distance refers to the average distance over a 50 position test set for the best network found at or before each generation. The distances are averaged over 20 simulations. Simulations were run for 80 generations, because at that point the standard approaches began to plateau. The learning curve demonstrates the efficiency gain from the neuron and blueprint populations.

The neuron-only version of SANE was as efficient as SANE in early generations, but was unable to maintain the same efficiency in later generations. Without a mechanism to propagate knowledge of the good networks that are formed, it is difficult for Neuron SANE to build upon the best networks. The poor late performance gives strong evidence to this problem and effectively shows the contribution of the network blueprint evolution. Neuron SANE performed comparably to the standard tournament approach.

4.2.3 Diversity Analysis

The second experiment tested the diversity level of the populations throughout evolution. Populations were evolved for 80 generations as in the first experiment, but after each generation, the population diversity was measured. A diversity metric, Φ, can be generated by taking the average Hamming distance between every two chromosomes, divided by the length of the chromosome:

$$\Phi = \frac{2\sum_{i=1}^{n}\sum_{j=i+1}^{n} H_{i,j}}{n(n-1)l},$$

where n is the population size, l is the length of each chromosome, and $H_{i,j}$ is the Hamming distance between chromosomes i and j. The value Φ represents the probability that a given bit at a specific position on one chromosome is different from a bit at the same position on a different chromosome. Thus, a random population would have $\Phi = 0.5$ since there is a 50% probability that any two bits in the same position differ.

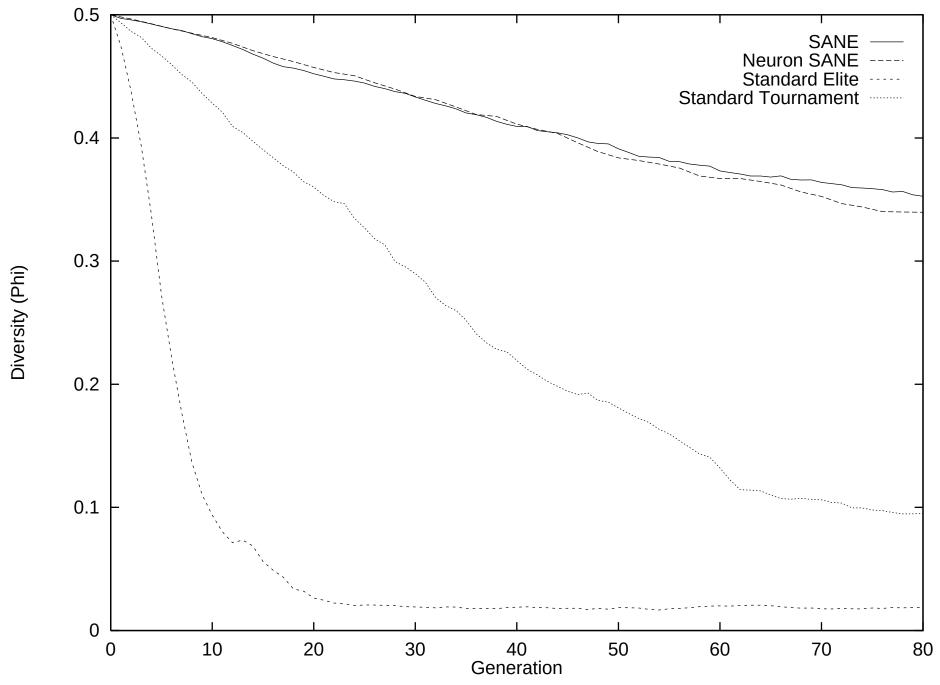


Figure 4.4: The population diversity for each simulation. The neuron-based approaches maintain very high levels of diversity, while the network-based approaches converge to a single solution.

Figure 4.4 shows the average diversity levels at each generation. The convergence of the standard elite approach is quite dramatic. Within 10 generations, 95% of the bit positions were identical. It is this phenomenon that leads most evolutionary algorithm implementors away from aggressive selection strategies and towards more conservative approaches like tournament selection. The tournament standard approach did converge much slower, but after 60 generations 90% of its bit positions were identical as well. Both SANE and Neuron SANE maintained very diverse populations throughout evolution, which confirms my hypothesis that SANE can perform a very aggressive search while maintaining a high level of diversity. Aggression balanced with diversity is the core of SANE's search strategy and is what sets it apart from current neuro-evolution approaches. Diversity allows SANE to improve its networks in later generations and, as demonstrated in the next experiment, adapt in changing environments.

4.2.4 Adaptive Analysis

The third experiment tested the ability of each approach to adapt to changes in the domain. Populations were evolved for 80 generations as in the first experiment. After 80 generations, the right back sensor of the khepera robot was "damaged", and the populations were evolved for 40 more generations. The sensor was set to a constant value of 1.0, which gives the illusion of an immediate obstacle to the rear of the robot. To adapt, the populations must learn to ignore the malfunctioning sensor and rely on the other back sensor. This experiment was designed to give a realistic situation for which adaptive behavior is necessary.

Figure 4.5 plots a learning curve for the simulations in the adaptive analysis. The y axis represents the difference in performance relative to the performance level achieved before the mal-

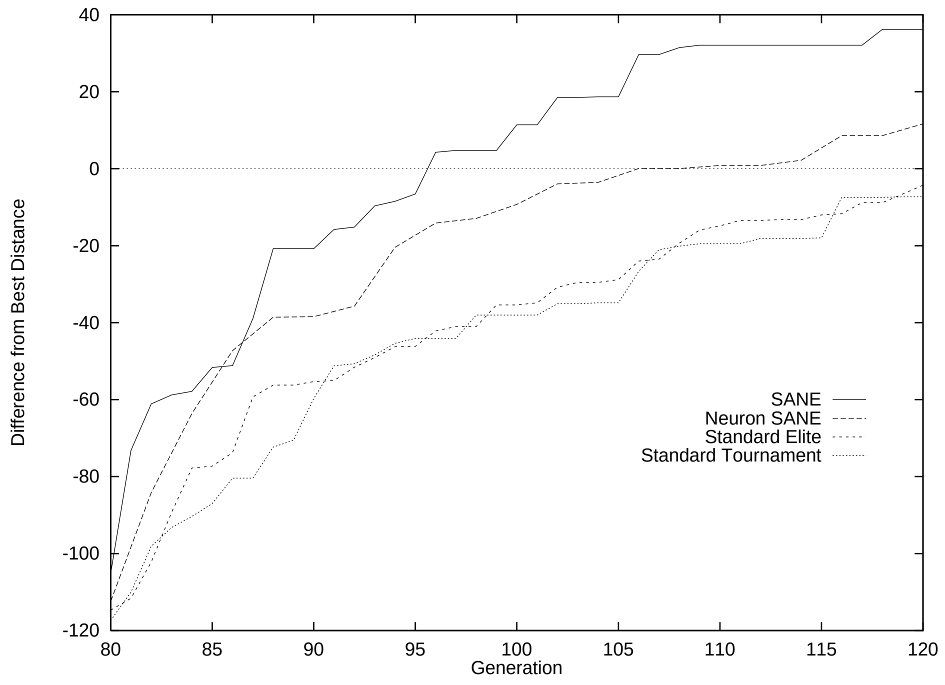


Figure 4.5: Adaptive comparisons. After evolving for 80 generations, a back sensor is fixed at 1.0. The graph plots the difference in performance from the best performance in the previous 80 generations.

functioning sensor. The horizontal line at 0 on the y axis represents the point where the search achieves the same performance as before the domain change. The relative distance is plotted rather than the absolute distance because each approach achieves a different level of performance after 80 generations.

As expected, the converged populations were much less adaptive than the diverse populations. After the sensor malfunctioned, each of the approaches lost about the same amount of performance, approximately 120 cm of distance on average. SANE quickly matched and surpassed its previous performance in an average of 15 generations. Neuron SANE required 25 generations to fully adapt. On average, neither of the standard approaches were able to achieve the performance level they had reached with the correct sensor within 40 generations. The performance of SANE and Neuron SANE demonstrates the importance of diversity in adapting populations. In addition, the performance difference between SANE and Neuron SANE further illustrates the contribution of the blueprint level evolution. SANE's ability to focus on the best neuron combinations allows it to continue to improve even in later generations.

The experiments in this section effectively demonstrate the advantages of SANE's symbiotic evolution. SANE found better solutions in less time than the more standard network-based approaches. Additionally, SANE maintained a high level of population diversity, which allowed it to adapt much easier to domain fluctuations. SANE's efficient search and adaptability make it more effective than standard neuro-evolution for solving difficult sequential decision tasks.

4.3 Analysis of Symbiotic Neurons

The next battery of tests were designed to study the different specializations that are formed within SANE's populations. In chapter 3, I hypothesized that SANE's neurons will not converge to a single type, but will instead form several subpopulations that each fill an important role in a neural network. The experiments described in this section illustrate this process and, in the context of the Khepera simulator, describe how and why certain neuron specializations emerge.

4.3.1 Principal Component Analysis

Principal component analysis (PCA) (see e.g. (Jolliffe 1986)) is a useful tool for visualizing the relative distances between high-dimensional vectors. PCA performs a coordinate transformation on a set of data points. The first dimension is chosen along the direction with the most variance in the data. The second is chosen perpendicular to the first, accounting for as much data variance as possible. Each new dimension or *principal component* is chosen in a similar fashion. The net result is a new coordinate system that is ordered in the directions of maximum data variance.

PCA can be used to perform a *dimensionality reduction* on high-dimensional data. To reduce the data points to M dimensions, PCA is run to determine the dimensions of maximum variance. The data points are then plotted along the first M coordinates. Since the coordinates are ordered, the resulting plot accounts for as much of the data variance as possible in M dimensions.

PCA can reduce the high-dimensional genetic chromosomes into two or three dimensions, which are easily plotted. Through PCA, similar chromosomes will reduce to similar two-dimensional representations and appear close together in the corresponding PCA plot. At first glance, this approach appears effective for studying the neuron specializations that are believed to evolve. There are several problems, however, in directly transforming the 240-bit chromosomes into 2-dimensional representations. The first problem is that the variance within the 240-dimensional space is too large for a 2-dimensional PCA reduction to accurately capture. In other words, in any dimensionality reduction there is a loss of information. When transforming from 240 dimensions to 2 dimensions the loss of information is so great that vectors with similar 2-dimensional representations may still vary greatly. In PCA reductions on several 240-dimensional vector populations, the first two principal components were only able to capture 60% of the data variance. More components are needed to accurately represent all of the data variance, but anything over 3 dimensions cannot be plotted.

The second problem is that a PCA plot of the raw genetic chromosomes may not reflect the true functionality dispersement of the population. This method assumes that similar genotypes (chromosomes) produce similar phenotypes (neural network hidden units). While this may be true in many cases, it is not an absolute. For example, two neurons may have identical bits in their weight alleles, but a few differing bit positions in their label alleles can create vastly different network architectures. A plot based on the chromosomes of these two neurons would place them close to each other, when they actually function quite differently. Thus, a plot based on the genotype does not ensure accurate representation of the relative function of each neuron.

A more effective strategy is to compute a *function vector* for each neuron in the population that describes its role in a neural network. The function vector can then be reduced using a PCA and plotted, resulting in a more accurate visualization of the different neuron roles. To generate

```
1.  Initialize the function vector to nil.
2.  Build a neural network with the neuron as the only hidden unit.
3.  For each input unit i:
    a. Set input i to 1.0 and all others to 0.0
    b. Propagate the activation through the output layer
    c. Append the output layer activations to the function vector
```

Table 4.3: The steps to compute a functional representation of a neuron. The function vector represents the actual function the neuron performs within the neural neural network.

such a vector, a neuron is implemented as the only hidden unit in a network and the network's output layer activations are recorded. The steps to compute a function vector for a given neuron are shown in table 4.3.
By cycling through the input units, the function vector captures the neuron's responses to each input unit activation. Thus, unlike the basic chromosome vector, the function vector encodes the direct behavior of each neuron when implemented in a neural network.

The function vector can be postprocessed to produce an even closer functional representation by interpreting the output layer activations for the appropriate task. For example, in the Khepera task the output layer activations are interpreted through the corresponding motor activations. Using the output layer interpretation described in section 4.1, output layer activations of $[0.4, 1.0, 0.6, 0.2]$ can be translated into motor activations of $[2, -8]$[3]. Postprocessing the function vector in this way creates a more accurate representation of the effect each neuron has on the robot. The final function vector consists of only 18 dimensions which can be reduced to 2 dimensions while preserving 95% of the data variance.

Figures 4.6 plots the two-dimensional functional representations of the neuron populations from a simulation in the Khepera task. PCA plots of two other simulations are presented in appendix B. Snapshots of the populations were taken at generations 0, 10, 20, 40, and 80. Generation 0 shows a fairly uniform distribution of the neurons reflective of the random populations. As the populations evolve, neurons begin to clump together and form subpopulations or specializations. In the final generation, the specializations become very distinct. The last graph plots the neurons that were included in the top three networks of the last generation. The graphs show that the best networks utilized neurons from several different subpopulations. This diversity demonstrates that each of the specializations plays an active role in the best neural networks.

In addition to the subpopulations, each plot contains several examples of neurons that are in between the clumps and isolated from other neurons. These neurons are created by interbreeding two members of different specializations and thus contain some of the functionality of each. The isolated neurons in the PCA, demonstrates how SANE can build new neuron roles through inter-breeding existing roles. In other words, these neurons are the explorers or pioneers that search out different neuron roles. If an effective role is found, a new subpopulation of neurons will form around the isolated neuron. An example of this phenomenon can be seen in figure 4.6. In generation 10, a single neuron exists around the point 27,2. As the population evolves, more

[3]For each motor (left and right) take the difference in the positive and negative direction and multiply by the maximum motor activation: $(0.6 - 0.4) \times 10 = 2$; $(0.2 - 1.0) \times 10 = -8$

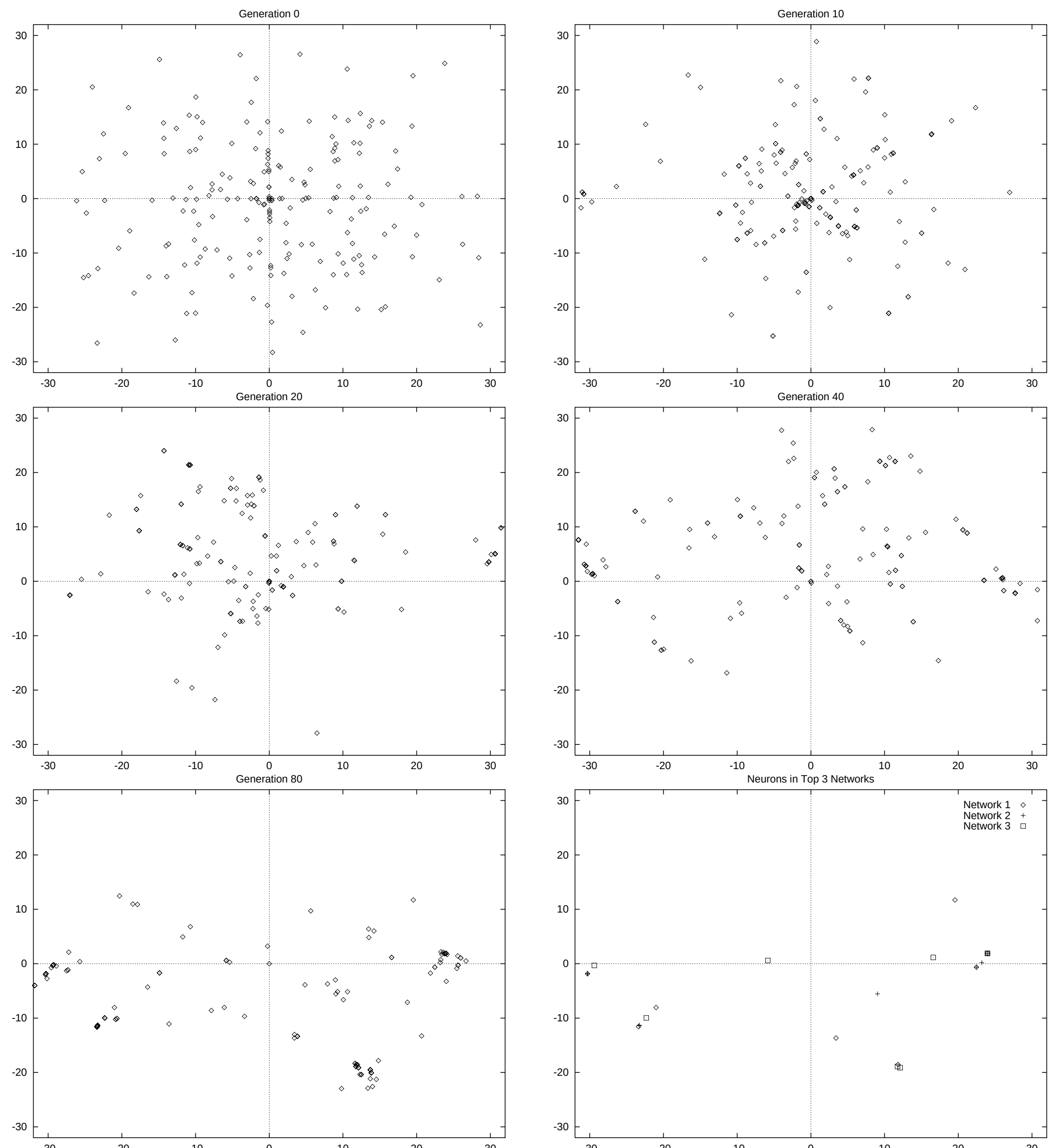


Figure 4.6: PCA of simulation 1.

and more neurons begin to clump around this area, which leads to the conclusion that the original neuron discovered a valuable neuron role. It should be noted, that the position on one PCA graph does not necessarily correspond to the same position on another PCA graph because dimensions are determined separately in each plot. However in these experiments, this has in general held true. Thus, the neurons around point 25,2 in generation 10 are similar in function to the neurons around point 25,2 in generation 80.

The PCA analysis confirms the hypothesis that SANE evolves several different types of neurons in a single population. As the populations evolve, neurons merge into several different specializations that optimize different aspects of the neural networks. Moreover, the specializations provide sufficient diversity to form new neural structures by inter-breeding between specializations. This phenomena is quite unique in evolutionary algorithms, since most approaches to co-evolution evolve separate species in segregated subpopulations or islands. In contrast, SANE forms its specializations naturally in a single population and takes advantage of the inherent diversity to produce a more explorative search.

4.3.2 Lesion Studies

While the emergence of the specializations is clear from the PCA studies, the function of each and the overall division of labor is not. To better understand the role of each specialization in the neural networks, simulated lesion studies were conducted. Lesions are used in biological neural networks to identify functions or functional areas of the brain. If a lesion results in a loss of a brain function, it can be concluded that the lesioned area was involved in carrying out that function. Similar experiments can be performed in artificial neural networks, by removing neurons and observing the behavior of the modified network.

Figure 4.7 shows a close up of the final PCA of the population for the first simulation. The specializations that are represented in the best network of the final generation are labeled A through E. While other groupings of neurons are also present, the current lesion experiments are only interested in the specializations included in the top neural network.

Two types of lesion experiments were conducted for a specific neuron or specialization: a neuron capability test and a neuron necessity test. In the capability test, a neuron is implemented alone and thus forms the entire hidden layer. Such experiments are designed to show how functional each neuron is without aid from other neurons in the population. The capability test will effectively demonstrate that a single neuron can not perform the entire task on its own. The second test is a necessity test, where a neuron in a functioning neural network is removed. The necessity test shows how crucial a specific neuron is in the performance of a neural network.

Table 4.4 shows the neurons included in the final network (numbered by fitness rank), their corresponding specializations, and the performance in the two lesion tests. The network contains two neurons from specialization B and two from E. Neuron number 1, which is represented by the single point at (19,11) in figure 4.7, does not appear to be a member of a well-defined specialization and is therefore not given a specialization label.

As expected, no neuron formed an effective hidden layer by itself. In the capability test, the highest scoring neuron only achieved a performance level of 64.5 (average Euclidean distance in the Khepera task), which is 1/6 of the performance of the entire network of neurons. The neurons generally spun the robot rapidly in either direction. Neurons 48 and 40, however, did not spin the

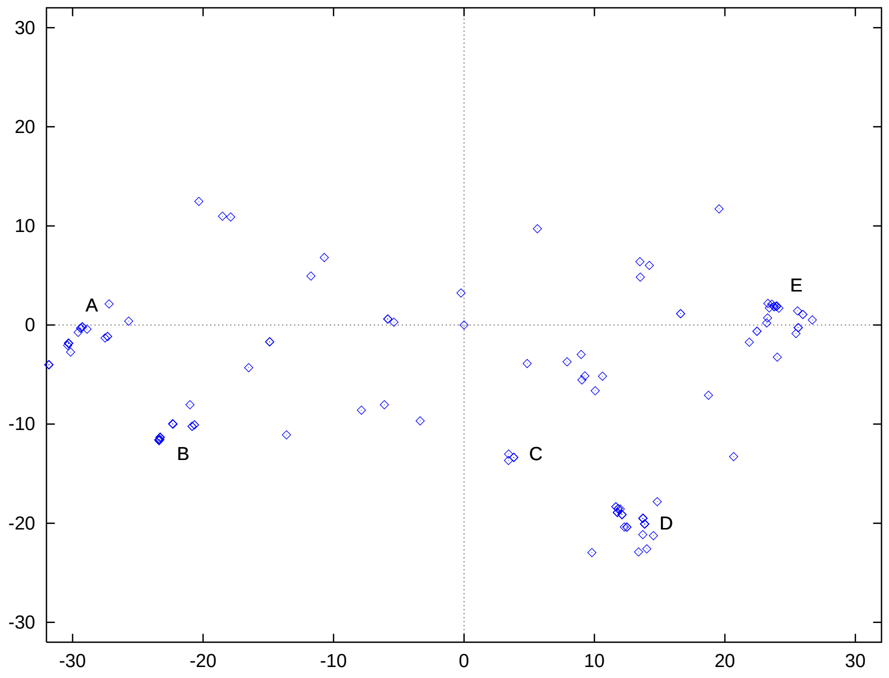


Figure 4.7: The principle components analysis (PCA) of the neuron population in the final generation. The specializations that are included in the top network are labeled A through E.

robot, but rather moved in a straight line. Unfortunately, these two neurons also moved straight into the nearest wall. The capability test numbers clearly illustrate the symbiotic nature of SANE's neuron population. The specializations that each represent cannot function without the presence of the other neurons in the hidden layer.

The necessity tests show the importance of each neuron to the network. Each neuron lesion caused a performance drop, in some cases as much as 25%. This consistent drop demonstrates that each neuron serves an important role. Interestingly, neurons of similar specializations did not exhibit similar performance degradations when lesioned. For example, neurons 11 and 38 are both members of specialization E. When lesioned from the network, the resulting performances are 402.6 without neuron 11 and 293.0 without neuron 38.

An explanation for this disparity is that neuron 39 encompasses neuron 11's function and that neuron 11 is a weak member of specialization E. To confirm this hypothesis, experiments were conducted where entire specializations were lesioned from the network. Figure 4.5 summarizes the performance numbers of the specialization lesion experiments. When both members of specialization E were lesioned, the network experienced a dramatic performance drop. The difference between the performance when both members are lesioned (145.3) and the performance when only neuron 39 is lesioned (293), confirms that neuron 11 does provide some of the function of specialization E. However, given the results of the previous necessity test, the function is not enough to compensate for the loss of neuron 39. Thus, neuron 11 does appear to be a weak member of specialization E.

To better understand the actual function of each neuron, the motor outputs, in response to specific sensory input, were analyzed. Table 4.6 shows the motor responses of each neuron given sensor activations on the left, in front, on the right, and behind the robot. The motor outputs

Neuron Rank	Specialization	Capability	Necessity
48	C	64.5	382.9
16	A	6.0	305.7
11	E	11.8	402.6
46	B	12.3	341.6
1	-	10.8	294.6
38	E	12.3	326.0
40	D	44.2	381.4
39	B	13.3	293.0

Table 4.4: Lesion results from simulation 1. Each of the neurons in the top network are implemented alone (Capability) and lesioned from the network (Necessity). The resulting network is tested in the Khepera task. The complete (non-lesioned) network achieved a performance level of 409.5. The capability test shows that no neuron can perform the entire task alone. The necessity test shows that while the network is quite robust to damage, each neuron plays an important role.

Lesioned Specialization	Performance
A	305.7
B	207.0
C	382.9
D	381.4
E	145.3

Table 4.5: Specialization Lesions. All members of a specialization are removed from the network. The resulting network is tested in the Khepera task. The complete (non-lesioned) network achieved a performance level of 409.5. Each specialization is crucial to achieve the network's top performance.

given no activation are also shown. The two output numbers represent the activation of each of the robot's motors.

Each neuron appears to give attention to a single sensor direction and change its output only when that particular sensor becomes active. For example, neuron 48 produces a consistent motor activation of +3 for both the left and right motors, except when the left sensors are active. Such output changes are very illustrative of the neuron's function, because they demonstrate the sensors to which the neuron responds and the type of response that is elicited.

As described in the capability test, neurons 40 and 48 cause the robot to move in a straight line when implemented alone. This behavior is represented in their motor output activations in table 4.6. Both neurons provide consistent positive activations to both motors, which causes the robot to move forward. The difference between the two neurons, which constitutes the difference between specialization C and D, is that they are sensitive to different sensory input. Neuron 40 reduces its positive motor activations when a left sensor is activated and neuron 48 does so when a forward sensor is activated. The elicited response of both neurons is to slow the robot when the sensor is activated. These two neurons provide the thrust of the robot on long straightaways and then slow the robot when obstacles are present. Neither of the neurons provide actual obstacle avoidance behavior; they are merely responsible for slowing the robot. Another specialization performs the turning.

Neurons 11 and 38, members of specialization E, and neuron 1 provide strong positive

		Motor Activation from Specific Sensors				
Neuron Rank	Specialization	None	Left	Forward	Right	Rear
48	C	+3,+3	0,0	+3,+3	+3,+3	+3,+3
16	A	-10,+8	-4,+2	-10,+8	-10,+8	-10,+8
11	E	+9,-5	+9,-5	+5,-3	+1,-1	+9,-5
46	B	-5,+8	-5,-6	-5,+8	-5,+8	-5,+8
1	-	+4,-7	+4,-9	+4,-9	+1,-4	+4,-8
38	E	+9,-5	+9,-5	+9,-5	0,-1	+9,-5
40	D	+9,+4	+9,+4	0,0	+5,+2	+9,+4
39	B	-5,+8	-3,+3	-5,+8	-5,+9	-5,+9

Table 4.6: Individual neuron responses to specific sensory inputs. The output numbers refer to the speed at which the neuron drives the left and right motors. For example, when neuron 11 receives input from its front sensors, it generates motor outputs of +5 and -3.

impulses to the left motor and strong negative impulses to the right motor. Naturally, these activations cause the robot to spin very fast in a clockwise direction. Specialization E, however, has evolved a symbiotic relationship with specializations A, B, and D, which spin the robot in a counterclockwise direction. The net result is a robot that does not spin. The balance that is attained between all of the motor activations explains the performance drop off when any one neuron is excluded.

Specialization E responds to activations of the front and right sensors by reducing the left motor and increasing the right motor. Such activations reduce the clockwise spin impulse and the robot begins to turn in a counterclockwise direction. The effect of specialization E is thus to veer the robot to the left to avoid objects in front of and to the right. Specialization A is a mirror of specialization E, veering the robot to the right to avoid objects on the left.

Neuron 1 was not placed in a specific specialization, because in the PCA graph it was not plotted in a distinct subgroup. Its proximity to specialization E on the PCA, however, is exhibited in its motor responses. Like specialization E, it reduces its normal clockwise spin impulse when the right sensors are active. The normal spin impulse and modified spin impulse of neuron 1, however, are not nearly as strong as specialization E. Neuron 1 was likely a mutant of specialization E that has increased the spin impulses just enough to make the robot make right turns in response to objects on the left. As shown in the necessity test, the network performs poorly when neuron 1 is removed.

Specialization B has a similar relationship to specialization A as neuron 1 does to specialization E. Like specialization A, B neurons spin the robot in a counter clockwise direction and reduce the spin impulse when a left sensor is active. The magnitude of the spin reduction, however, is much less than specialization A and appears to be more of a fine tuning of the right turn.
In summary, the roles of the neurons in the top network are:

48	Provides forward activations and slows down when object is sensed on the left.
16	Veers the robot right to avoid objects to the left.
11	Veers the robot left to avoid objects to the right.
46	Provides small right impulse when objects are sensed to the left.
1	Provides additional spin impulses for specialization E.
38	Veers the robot left to avoid objects to the right.
40	Provides forward activations and balances the spin caused by the other neurons. Slows robot when an object is sensed in front and to right.
39	Veers the robot right to avoid objects to the left.

Lesion studies conducted in other Khepera simulations produced very similar results. In almost all cases, separate specializations evolved to control the forward thrust, right turns, and left turns. At least one simulation evolved a neuron specialization that produced a 0,0 motor activation across all sensory inputs. These "no-op" neurons have no effect on the network behavior and may have evolved as fillers, since the existing specializations were sufficient for solving the task.

It's also important to note that neuron roles were often redundant across several specializations and within the neural networks. For example, specializations A and B both maneuver around objects to the left of the robot, and members of both are included in the final network. This suggests that SANE does not develop a minimal set of behaviors, but instead distributes important functions across several neurons and multiple specializations. This produces a much more robust controller, since damage to a single neuron can be overcome by the redundant function of the other neurons. As shown in table 4.4, the loss of one neuron does not result in a dramatic drop in performance.

The lesion studies demonstrate the symbiotic nature of SANE's neurons. No neuron could perform well alone, and every neuron was necessary to achieve the network's top performance level. The studies also confirm the hypothesis that SANE performs a parallel search in several different subgroups of its neuron population. Analysis of the roles of the different specializations show a clear division of labor with the largest division between forward motor neurons, left turn neurons, and right turn neurons. Several subpopulations within specializations appear to be refinements of the major roles, which allow SANE to continue to improve each specialization. It is this parallel search that sets SANE apart from existing neuro-evolution approaches and should allow SANE to more efficiently solve difficult problems.

4.4 Analysis of the Blueprint Population

The analysis of the blueprint population is less complicated than the neuron population, because it uses the standard evolutionary algorithm strategy of searching for a single fit individual to solve the task. Thus, while specialization may emerge in the blueprint population, it is not essential because we are only interested in a single type of individual: the one that can perform the entire task. The task in this case is to include a group of neurons that together form the necessary input/output

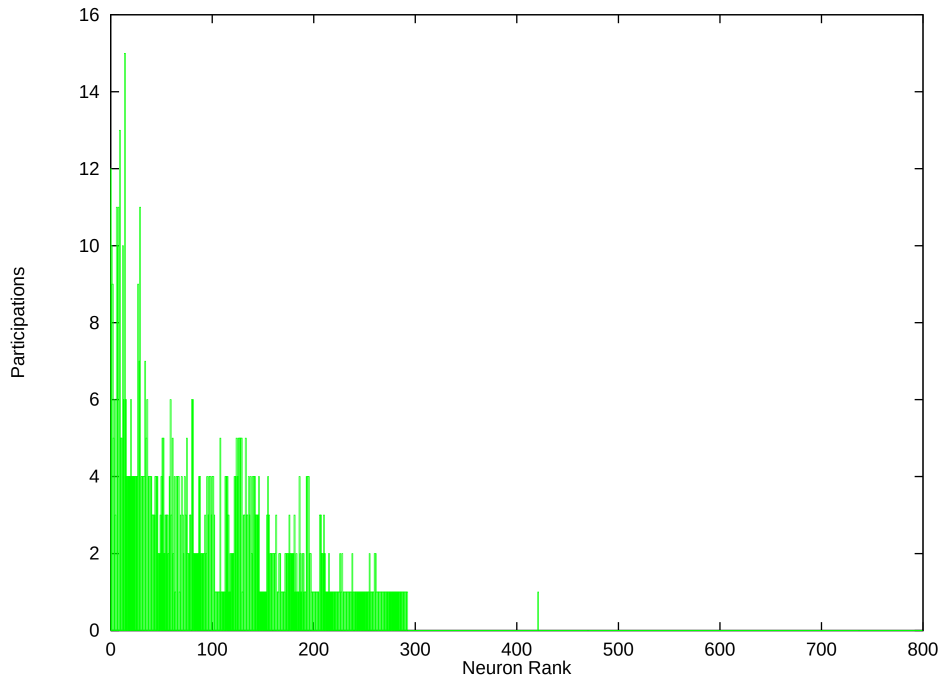


Figure 4.8: The number of network participations for each neuron in the neuron population. The neurons are ranked based on their fitness. The blueprint population biases the rate of participation towards the best neurons.

connections to produce a highly fit neural network. However, it is still important to examine the blueprint chromosomes to determine the rate of participation of the neurons and to understand the dispersement of neurons within the top blueprints.

As described in chapter 3, one of the advantages of the blueprint evolution is that neuron participation is biased towards the top performing neurons. In other words, the best neurons will participate in more networks than the newer or poorer neurons. Figure 4.8 shows the typical rate of participation in the neuron population in the last generation of the Khepera simulations. The data was collected from a single simulation, however, other simulations exhibited very similar behavior. Of the 800 opportunities for participation,[4] 40% were filled by neurons ranked in the top 6% of the population and 78% by neurons in the top 25%. Neurons ranked from 300 to 800 did not participate in any networks. This participation bias causes the better-performing neurons to be more extensively evaluated, resulting in more accurate fitness estimates. As shown in the comparisons with Neuron SANE in section 4.2.2, the biased participation produces more efficient search behavior in later generations than the uniform participation.

Figure 4.9 shows the neurons of the top 20 network blueprints during the last generation of a single simulation. The neuron pointer numbers refer to the rank of that neuron in the population *after* the fitness had been distributed. In other words, the rankings reflect that the neurons participated in these top blueprints. For example, the top blueprint included the 48th ranked neuron, the 16th ranked neuron, the top ranked neuron, and so on. The first conclusion from the table is that the blueprint population is quite diverse. Blueprints 16 and 19 are the most similar, but they

[4]100 networks are formed per generation, each with 8 neurons.

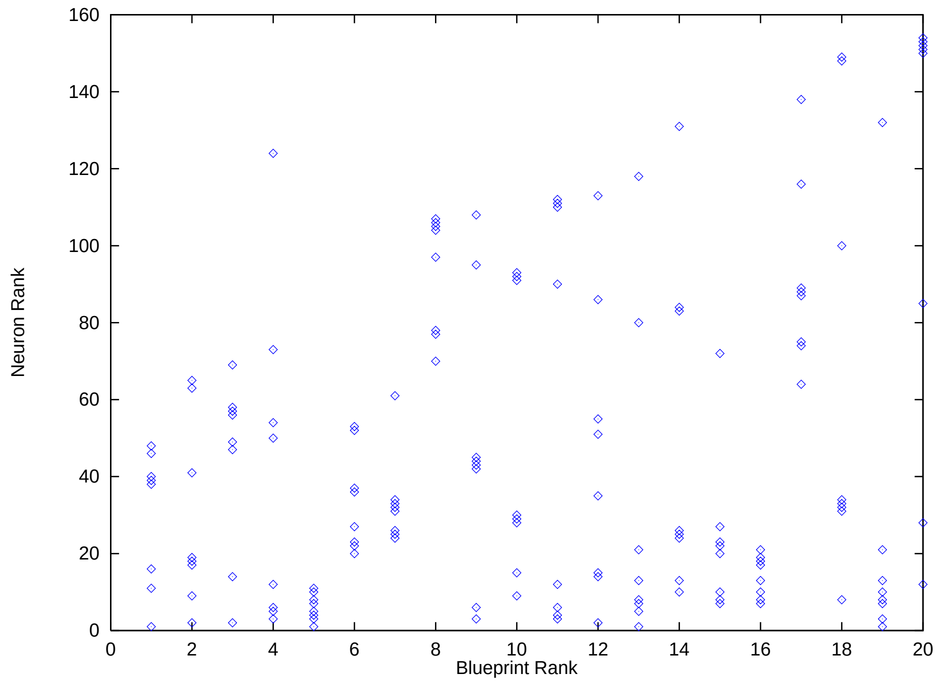


Figure 4.9: The neurons in the top 20 network blueprints in the last generation of a simulation. The neuron numbers refer to their rank in the population after their fitness was calculated. The figure shows a very diverse collection of neurons and several examples of offspring neurons present in the best networks.

only share 5 out of the 8 neuron pointers. Thus the mutation strategy presented in section 3.3.3 appears to provide sufficient diversity, allowing the blueprint population to sample many different combinations of neuron pointers.

Figure 4.9 also shows that many neurons ranked in the 100-150 range are included in the top networks. Interestingly, these neurons only participate in 1 or 2 networks per generation (figure 4.8). Closer inspection reveals that the majority of these neurons are new offspring neurons generated during the previous generation. From the prevalent use of these neurons in the top blueprints, it can be concluded that the blueprint evolution is making effective use of the new neural structures created by the neuron evolution. This behavior illustrates how the blueprint population immediately capitalizes on the new solutions created in the neuron population and demonstrates the effective integration of the two genetic searches.

Recall that the blueprint evolution uses a selective mutation strategy to switch pointers from breeding neurons to their offspring. To evaluate this strategy ten simulations were run with this feature removed from SANE. Figure 4.10 shows the performance of the searches with and without the selective mutation strategy. The searches are comparable in early generations, but diverge after generation 20. Without selective mutation, the blueprints converge, and there is little variance in the types of networks that are created. Consequently, the neuron evolution cannot evaluate new neuron combinations and the search stalls. With selective mutation, the blueprint population remains diverse (figure 4.9), offspring neurons participate in new networks, and the search continues to improve.

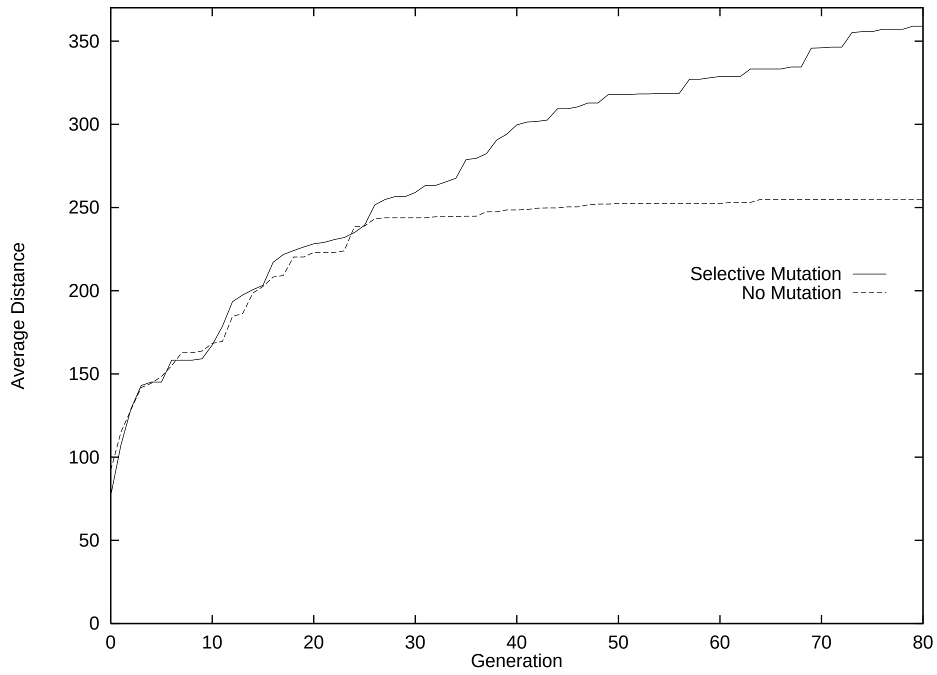


Figure 4.10: An evaluation of the selective mutation strategy of the blueprint evolution. When pointers are not switched from breeding neurons to offspring, the search stalls in later generations. The results are averaged over 10 simulations in the Khepera task.

4.5 Evaluation of Neuron Encoding

The SANE system makes two contributions to neuro-evolution. The first, symbiotic evolution, was illustrated in the previous experiments. The second contribution of SANE is a more effective neural encoding strategy. As described in chapter 3, the straightforward network encoding most researchers use is problematic. Simply concatenating the weights of a fixed-architecture network is not conducive to an evolutionary algorithm.

To demonstrate the merits of SANE's encoding, an experiment was conducted to compare SANE with a version of SANE that used the concatenation of weights strategy (Belew et al. 1991; Whitley et al. 1993). Such strategies fix the architecture of the neural network (normally feed-forward and fully connected) and decode the weights sequentially from the chromosome. In the Khepera task, the chromosome of a neuron using the concatenation strategy consists of 13 16-bit weight genes, which equals the number of input and output units. The total number of bits in the concatenated encoding is thus 208, compared to 240 bits for SANE's normal encoding.

The experimental setup was identical to the performance experiment in section 4.2.2. Populations were evolved for 80 generations with fully functioning sensors. Figure 4.11 plots the learning curve averaged over the 20 simulations. The curve shows a clear advantage to the SANE encoding. Using the concatenation approach, SANE could only achieve a 325 cm performance level on average compared to 358 cm with its normal encoding.

A key advantage of SANE's encoding is the freedom to modify the architecture of the network along with the weights. This flexibility may allow SANE to master more difficult tasks than the more rigid, fixed-architecture approaches. For example, if some of the network inputs are

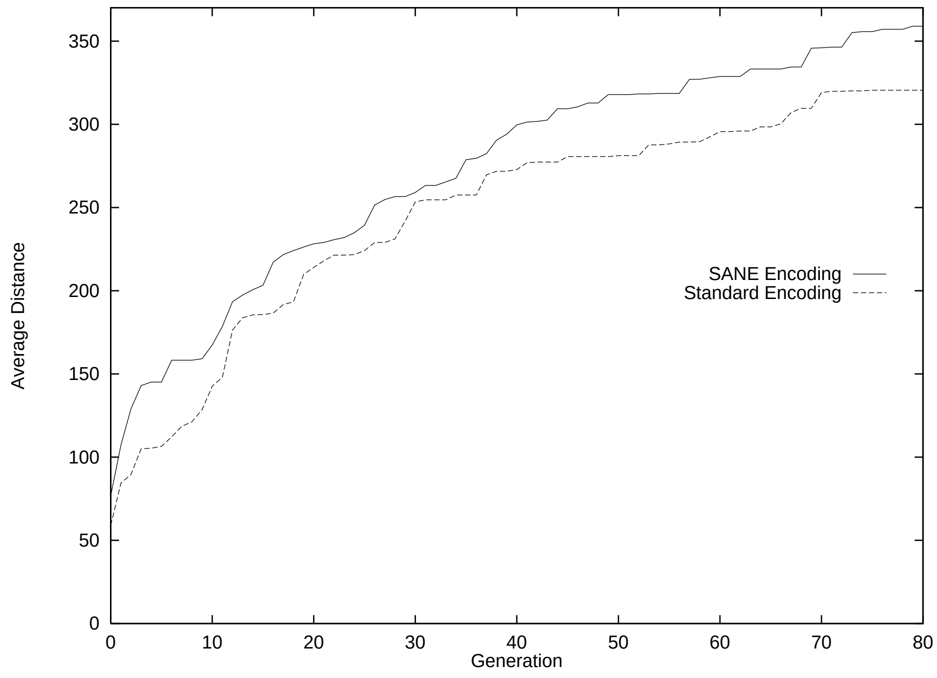


Figure 4.11: Encoding Comparison. SANE was implemented with its normal encoding (SANE Encoding) and the more standard concatenation encoding. The graph illustrates the search efficiency gain from SANE's encoding.

irrelevant and only create noise in the network, SANE can easily adjust the architecture so that no hidden units form connections to them. A fixed, fully-connected architecture, however, is forced to connect to all units and would have to set all of the weights to 0 to ignore input from these units. Since evolutionary algorithms do not normally make small systematic weight changes, it is very difficult to set a weight to such an exact value.

Freedom to modify the architecture, however, comes with a price: a much larger search space. The disparity in search spaces between SANE with and without a fixed architecture only makes the differences presented in figure 4.11 more significant. SANE with its normal encoding finds better solutions faster, while searching a larger space of solutions. As described in section 3.2.3, the missing attribute in the concatenation strategy is position independence of the genes. In the SANE encoding, the effective network connections are not restricted to specific positions, but may form anywhere on the chromosome. This freedom allows SANE to realize good connections wherever they initially appear, resulting in a faster manifestation of useful neural structures.

4.6 Concluding Remarks

The experiments in this chapter illustrate how SANE works by comparing its search behavior to other methods of neuro-evolution and empirically analyzing its solutions. Compared to two standard neuro-evolutionary approaches, SANE formed solutions faster, maintained a much higher level of population diversity, and was more adaptive in domain shifts. Additionally, SANE's blueprint evolution was shown effective in maintaining and exploiting good combinations of neurons. The principal component analysis and lesion studies illustrated the unique dynamics in SANE's population. As hypothesized, SANE's neurons specialize within the single population and fill several different roles. This specialization is the heart of SANE's search efficiency. By reducing the solution space for each individual, SANE searches in parallel decompositions of the complete neural network space.

Another contribution of SANE is demonstrated through comparisons of its neuron encoding strategy to the very common "concatenation" encoding. The position-independent feature of SANE's genes allows the evolutionary algorithm more freedom to realize and build useful structures, which increases the overall search efficiency. In summary, this chapter has shown the contribution of the primary features of SANE's neuro-evolutionary search. Conjunctively, these features produce an efficient approach to both neuro-evolution and reinforcement learning.

Chapter 5

Performance Comparisons

The preceding chapters have described the SANE method of neuro-evolution and evaluated many of the components that contribute to its efficiency. The purpose of this chapter is to compare the performance SANE directly to existing approaches for sequential decision learning. Comparisons were carried out with four other methods, 1 and 2 layer AHC, Q-learning, and GENITOR in two benchmarks. The first benchmark of balancing a pole on a cart has become a standard benchmark within the reinforcement learning community. The second benchmark uses the Khepera simulator and the task described in the previous chapter. The comparisons were designed to test the following hypotheses:

1. SANE is more effective in terms of network evaluations and CPU time required to both standard neural network reinforcement learning techniques and more aggressive neuro-evolution approaches.

2. SANE outperforms existing approaches with noisy evaluations and sparse reinforcement.

3. The generalization of networks formed through SANE is comparable to the generalization of networks formed through other reinforcement learning techniques.

4. SANE can form higher quality solutions in difficult problems.

5. SANE is more robust than temporal difference approaches in problems with hidden state information.

5.1 Pole Balancing Benchmark

The inverted pendulum or pole-balancing problem is a classic control problem that has received much attention in the reinforcement learning literature (Anderson 1989; Barto et al. 1983; Michie and Chambers 1968; Whitley et al. 1993). Since almost every RL method to date has been evaluated in pole balancing, there exists a rich collection of methods to benchmark against. Moreover, many RL researchers have publicly distributed the simulation code of their methods solving this problem. Pole balancing thus contains several methods that can be taken “off the shelf” and run without implementation bias. The goal of these comparisons is to evaluate SANE with respect to existing methods in a well known problem. The specific experiments compare learning speed, generalization,

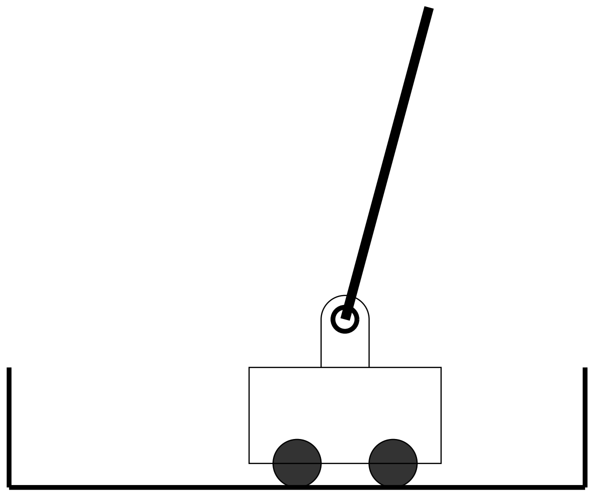


Figure 5.1: The cart-and-pole system in the inverted pendulum problem. The cart is pushed either left or right until it reaches a track boundary or the pole falls below 12 degrees.

and resilience to evaluation noise during. All learning-related parameters were selected by the corresponding authors and in most cases each method was run from the original code written by the authors.

5.1.1 The Pole Balancing Task

In the pole balancing problem, a single pole is centered on a cart (figure 5.1), which may move left or right on a horizontal track. Naturally, any movements to the cart tend to unbalance the pole. The objective is to push the cart either left or right with a fixed-magnitude force such that the pole remains balanced and the track boundaries are avoided. The controller receives reinforcement only after the pole has fallen, which makes this task a challenging credit assignment problem for a reinforcement learning system.

The controller is afforded the following state information: the position of the cart (ρ), the velocity of the cart ($\dot{\rho}$), the angle of the pole (θ), and the angular velocity of the pole ($\dot{\theta}$). At each time step, the controller must resolve which direction the cart is to be pushed. The cart and pole system can be described with the following second order equations of motion:

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

where

ρ:	The position of the cart.
$\dot{\rho}$:	The velocity of the cart.
θ:	The angle of the pole.
$\dot{\theta}$:	The angular velocity of the pole.
l:	The length of the pole = 0.5 m.
m_p:	The mass of the pole = 0.1 kg.
m:	The mass of the pole and cart = 1.1 kg.
F:	The magnitude of force = 10 N.
g:	The acceleration due to gravity = 9.8.

Through Euler's method of numerical approximation, the cart and pole system can be simulated using discrete-time equations of the form $\theta(t+1) = \theta(t) + \tau\dot{\theta}(t)$, with the discrete time step τ normally set at 0.02 seconds. Once the pole falls past 12 degrees or the cart reaches the boundary of the 4.8 meter track, the trial ends and a reinforcement signal is generated. The performance of the controller is measured by the number of time steps in which the pole remains balanced. The above parameters are identical to those used by Barto et al. (1983), Anderson (1987), and Whitley et al. (1993) in this problem, and presumably the architectures were optimized to work well under these parameters.

5.1.2 Controller Implementations

Five different reinforcement learning methods were implemented to form a control strategy for the pole-balancing problem: SANE, the single-layer Adaptive Heuristic Critic (AHC) of Barto et al. (1983), the two-layer AHC of Anderson (1987), Anderson (1989), the Q-learning method of Watkins and Dayan (1992), and the GENITOR system of Whitley et al. (1993). The original programs written by Sutton and Anderson were used for the AHC implementations, and the simulation code developed by Pendrith (1994) was used for the Q-learning implementation. For GENITOR, the system was reimplemented as described in (Whitley et al. 1993). A control strategy was deemed successful if it could balance a pole for 120,000 time steps.

SANE

SANE was implemented to evolve a 2-layer network with 5 input, 8 hidden, and 2 output units. Each hidden neuron specified 5 connections giving each network a total of 40 connections. The number of hidden neurons was chosen so that the total number of connections was compatible with the 2-layer network implementations of the AHC and GENITOR. Several trial simulations were run to discover effective population sizes for SANE. Since there is a high solution density in the pole balancing problem and solutions are found often within 1000 network evaluations, small population sizes of 100 and 50 for the neuron and network levels were the most effective. Each network evaluation consisted of a single balance attempt where a sequence of control decisions were made until the pole fell or the track boundaries were reached. The fitness was determined by the number of steps the pole remained balanced. The input to the network consisted of the 4 state variables $(\theta, \dot{\theta}, \rho, \dot{\rho})$, normalized between 0 and 1 over the following ranges:

	1-AHC	2-AHC	QL		GENITOR	SANE
Action Learning Rate (α):	1.0	1.0		Population Size:	100	100,50
Critic Learning Rate (β):	0.5	0.2	0.2	Mutation Rate:	Adaptive	0.1%
TD Discount Factor (γ):	0.95	0.9	0.95	Chrom. Length:	35 (floats)	120 (bits)
Decay Rate (λ):	0.9	0		Subpopulation (ζ):		8

Table 5.1: Implementation parameters for each method.

$$\begin{array}{rl} \rho: & (-2.4, 2.4) \\ \dot{\rho}: & (-1.5, 1.5) \\ \theta: & (-12^\circ, 12^\circ) \\ \dot{\theta}: & (-60^\circ, 60^\circ) \end{array}$$

To make the network input compatible with the implementations of Whitley et al. (1993) and Anderson (1987), an additional bias input unit that is always set to 0.5 was included.

Each of the two output units corresponded directly with a control choice (left or right). The output unit with the greatest activation determined which control action was to be performed. The output layer, thus, represented a ranking of the possible choices. This approach differs from most neuro-control architectures like those in these comparisons, where the activation of an output unit represents a probability of that action being performed (Anderson, 1987; Barto et al., 1983; Whitley et al., 1993). For example, a decision of "move right" with activation 0.9 would move right only 90% of the time. Probabilistic output units allow the network to visit more of the state space during training, and thus incorporate a more global view of the problem into the control policy (Whitley et al. 1993). In the SANE implementation, however, randomness is unnecessary in the decision process since there is a large amount of state space sampling through multiple combinations of neurons.

AHC

Two different AHC implementations were tested: A single-layer version (Barto et al. 1983) and a two-layer version (Anderson 1987). Table 5.1 lists the parameters for each method. Both implementations were run directly from simulation code written by Sutton and Anderson, respectively. The learning parameters, network architectures, and control strategy were thus chosen by Sutton and Anderson and presumably reflect parameters that have been found effective.

Since the state evaluation function to be learned is non-monotonic (Anderson 1989) and single-layer networks can only learn linearly-separable tasks, Barto et al. (1983) discretized the input space into 162 nonoverlapping regions or "boxes" for the single-layer AHC. This approach was first introduced by Michie and Chambers (1968), and it allows the state evaluation to be a linear function of the input, which allows single layer networks to be used. Both the value and action network consist of one unit with a single weight connected to each input box. The output of the unit is the inner product of the input vector and the unit's weight vector, however, since only one input box will be active at one time, the output reduces to the weight corresponding to the active input box.

In the two-layer AHC, discretization of the input space is not necessary since additional hidden units allow the network to represent any non-linear discriminant function. Therefore, the same continuous input that was used for SANE was also used for the two-layer AHC. Each network (evaluation and action) in Anderson's implementation consists of 5 input units (4 input variables and one bias unit set at 0.5), 5 hidden units, and one output unit. Each input unit is connected to every hidden unit and to the single output unit. The two-layer networks are trained using a variant of backpropagation (Anderson 1989). The output of the action network is interpreted as the probability of choosing that action (push left or right) in both the single and two-layer AHC implementations.

Q-learning

The Q-learning simulations were run using the simulation code developed by Pendrith (1994), which employs one-step updates as described by Watkins and Dayan (1992). In this implementation, the Q-function is a look-up table that receives the same discretized input that Barto et al. created for the single-layer AHC. Actions on even-numbered steps are determined using the stochastic action selector described by Lin (1992). The action on odd-numbered steps is chosen deterministically according to the highest associated Q-value. Pendrith found that such interleaved exploration and exploitation greatly improves Q-learning in the pole-balancing domain. My experiences confirmed this result: when interleaving was disabled, Q-learning was incapable of learning the pole-balancing task.

GENITOR

GENITOR is an advanced evolutionary algorithm method that includes external functions for ensuring population diversity. Diversity is maintained through *adaptive mutation*, which raises the mutation rate as the population converges (section 7.1). The motivation for comparing SANE to GENITOR is twofold. Comparisons between GENITOR's and SANE's search efficiency thus test the hypothesis that the symbiotic evolution produces an efficient search without reliance on additional randomness. Since GENITOR has been shown to be effective in evolving neural networks for the inverted pendulum problem (Whitley et al. 1993), it also provides a state-of-the-art neuro-evolution comparison.

GENITOR was implemented as detailed by Whitley et al. (1993) to evolve the weights in a fully-connected 2-layer network, with additional connections from each input unit to the output layer. The network architecture is identical to the two-layer AHC with 5 input units, 5 hidden units and 1 output unit. The input to the network consists of the same normalized state variables as in SANE, and the activation of the output unit is interpreted as a probabilistic choice as in the AHC.

5.1.3 Learning-Speed Comparisons

The first experiments compared the time required by each algorithm to develop a successful network (one that can balance the pole for 120,000 time steps). Both the number of pole-balance attempts required and the CPU time expended were measured and averaged over 50 simulations. The number of balance attempts reflects the number of training episodes required. The CPU time was included because the number of balance attempts does not describe the amount of overhead each algorithm incurs. The CPU times should be treated as ballpark estimates because they are sensitive to the implementation details. However, the CPU time differences found in these experiments are large

	Pole Balance Attempts				CPU Time				
Method	Mean	Best	Worst	SD	Mean	Best	Worst	SD	Failures
1-layer AHC	430	80	7373	1071	49.4	14	250	52.6	3
2-layer AHC	12513	3458	45922	9338	83.8	13	311	61.6	14
Q-learning	2402	426	10056	1903	12.2	4	41	7.8	0
GENITOR	2578	415	12964	2092	9.8	4	54	7.9	0
SANE	900	101	2502	598	7.7	4	17	2.9	0

Table 5.2: The number of pole balance attempts and CPU time required to find a successful network. The number of pole balance attempts refers to the number of training episodes or "starts" necessary. The numbers are computed over 50 simulations for each method. A training failure was said to occur if no successful network was found after 50,000 attempts.

enough to indicate real differences in training time among the algorithms. Each implementation was written in C and compiled using the cc compiler on an IBM RS6000 25T workstation with the -O2 optimization flag. Otherwise, no special effort was made to optimize any of the implementations for speed.

The comparison (table 5.2), was based on the random start state approach of Anderson (1987), Anderson (1989) and Whitley et al. (1993). The cart and pole were both started from random positions with random initial velocity. The positions and velocities were selected from the same ranges that were used to normalize the input variables, and could specify a state from which pole balancing was impossible. Following Anderson (1987) and Whitley et al. (1993), a network was considered successful if it could balance the pole from any single start state.

Results

The results show the AHCs to require significantly more CPU time than the other approaches to discover effective solutions. While the single-layer AHC needed the lowest number of balance attempts on average, its long CPU times overshadow its efficient learning. Typically, it took over two minutes for the single-layer AHC to find a successful network. This overhead is particularly large when compared to the evolutionary algorithm approaches, which took only five to ten seconds. The two-layer AHC performed the poorest, exhausting large amounts of CPU time and requiring at least 5, but often 10 to 20, times more balance attempts on average than the other approaches.

The experiments confirmed Whitley's observation that the AHC trains inconsistently when started from random initial states. Out of the 50 simulations, the single-layer AHC failed to train in 3 and the two-layer AHC failed in 14. Each unsuccessful simulation was allowed to train for 50,000 pole balance attempts before it was declared a failure. The results presented for the AHC in table 5.2 are averaged over the successful simulations only, excluding the failures.

Q-learning and GENITOR were comparable across both tests in terms of mean CPU time and average number of balance attempts required. The differences in CPU times between the two approaches are not statistically significant and thus are not large enough to discount implementation details. Both Q-learning and GENITOR were close to an order of magnitude faster than the AHCs and incurred no training failures.

SANE required 1/3 as many balance attempts as Q-learning and GENITOR on average and 1/12 as many as the two-layer AHC. Like Q-learning and GENITOR, SANE found solutions in every simulation. The additional overhead of SANE's two populations, compared to GENITOR's one, caused the CPU disparity to be less than the difference in evaluations, however, SANE still found solutions in 80% of the time required by GENITOR. Furthermore, the time required to learn the task varied the least in the SANE simulations. 90% of the CPU times (in seconds) fall in the following ranges:

1-layer AHC:	$[17, 136]$
2-layer AHC:	$[17, 124]$
Q-learning:	$[4, 20]$
GENITOR:	$[4, 17]$
SANE:	$[4, 10]$

Thus, while the AHC can vary as much as 2 minutes among simulations and Q-learning and GENITOR about 15 seconds, SANE consistently finds solutions in 4 to 10 seconds of CPU time, making it the fastest and most consistent of the learning methods tested in this task. The results therefore confirm the hypothesis that in the standard reinforcement benchmark, SANE is faster and more efficient than existing methods.

Discussion

The large CPU times of the AHC are caused by the many weight updates that they must perform after every action. Both the single and two-layer AHCs adjust every weight in the neural networks after each activation. Since there are thousands of activations per balance attempt, the time required for the weight updates can be substantial. The Q-learning implementation reduces this overhead considerably by only updating a single table entry after every step, however, these continuous updates still consume costly CPU cycles. Neither SANE nor GENITOR require weight updates after each activation, and do not incur these high overhead costs.

Note that the Q-function can be represented efficiently as a look-up table only when the state space is small. In a real-world application, the enormous state space would make explicit representation of each state impossible. Larger applications of Q-learning are likely to use neural networks (Lin 1992), which can learn from continuous input values in an infinite state space. Instead of representing each state explicitly, neural networks form internal representations of the state space through their connections and weights, which allows them to generalize well to unobserved states. Like the AHC, a neural network implementation of Q-learning would require continuous updates of all neural network weights, which would exhaust considerably more CPU time than the table look-up implementation. Recent research has shown that that a neural network implementation of Q-learning performs comparably to the two-layer AHC (Lin 1992; Pendrith 1994).

Both the single-layer AHC and Q-learning had the benefit of a presegmented input space, while the two-layer AHC, GENITOR, and SANE methods received only undifferentiated real values of the state variables. Barto et al. (1983) selected the input boxes according to prior knowledge of the "useful regions" of the input variables and their compatibility with the single-layer AHC. This information allowed the single-layer AHC to learn the task in the least number of balance attempts. The input partitioning, however, did not extend well to the Q-learner, which required as many pole-balance attempts as the methods receiving real-valued inputs.

Interestingly, the results achieved with GENITOR were better than those reported by Whitley et al. (1993). This disparity is probably caused by the way the input variables were normalized.

	Pole Balance Attempts				CPU Time				
Method	Mean	Best	Worst	SD	Mean	Best	Worst	SD	Failures
Neuron SANE	1691	46	4461	984	5.2	4	9	1.1	0
SANE	900	101	2502	598	7.7	4	17	2.9	0

Table 5.3: A comparison of SANE with an early version of SANE (called Neuron SANE) that did not use the blueprint population to maintain knowledge of effective neuron combinations. SANE's second population creates more overhead, but reduces the number of training evaluations considerably. Even without the blueprint population, Neuron SANE solves the problem faster than the other approaches.

Since it was unclear what ranges Whitley et al. (1993) used for normalization, the input vectors could be quite different. On average, my implementation of GENITOR required only half of the attempts, which suggests that Whitley et al. may have normalized over an overly broad range.

The comparison between SANE and GENITOR confirms the hypothesis that symbiotic evolution can perform an efficient genetic search without relying on high mutation rates. It appears that in GENITOR, the high mutation rates brought on through adaptive mutation may be causing many disruptions in highly-fit schemata (genetic building blocks), resulting in many more network evaluations required to learn the task.

Previous results have shown that a version of SANE without the blueprint population is quite effective in the pole balancing task (Moriarty and Miikkulainen 1996a). The neuron-only version of SANE, however, was ineffective when scaled up to more difficult tasks, and the blueprint population was added to focus the search on the best combination of neurons. Table 5.3 shows the performance of SANE with and without a blueprint population.[1] The addition of the blueprint population creates more overhead (higher CPU time), but cuts the number of evaluations necessary almost in half. In larger tasks, where evaluations are very expensive the efficiency gain allows SANE to scale much better than Neuron SANE.

5.1.4 Generalization Comparisons

The second test explores the generalization ability of the neural networks formed with each method. Networks that generalize well can transfer concepts learned in a subset of the state space to the rest of the space. Such behavior is of great benefit in real-world tasks where the enormous state spaces make explicit exploration of all states infeasible. In the pole balancing task, networks were trained until a network could balance the pole from a single start state. How well these networks could balance from other start states demonstrates their ability to generalize.

One hundred random start states were created as a test set for the final network of each method. The network was said to successfully balance a start state if the pole did not fall below 12° within 1000 time steps. Table 5.4 shows the generalization performance over 50 simulations. Since some initial states contained situations from which pole balancing was impossible, the best networks were successful only 80% of the time.

Generalization was comparable across the AHCs and the genetic algorithm approaches. The mean generalization of the Q-learning implementation, however, was significantly lower than those of the single-layer AHC, GENITOR, and SANE. This difference is likely due to the look-up table

[1]The method of neuron fitness is slightly different between the two approaches and is described in section 4.2.1

Method	Mean	Best	Worst	SD
1-layer AHC	50	76	2	16
2-layer AHC	44	76	5	20
Q-learning	41	61	13	11
GENITOR	48	81	2	23
SANE	48	82	1	24

Table 5.4: The generalization ability of the networks formed with each method. The numbers show the percentage of random start states balanced for 1000 time steps by a fully-trained network. Fifty networks were formed with each method. There is a statistically significant difference ($p < .01$) between the mean generalizations of Q-learning and those of the single-layer AHC, GENITOR, and SANE. The other differences are not significant.

employed by the Q-learner. In the single-layer AHC, which uses the same presegmented input space as the Q-learner, all weights are updated after visiting a single state, allowing it to learn a smoother approximation of the control function. In Q-learning, only the weight (i.e. the table value) of the currently visited state is updated, preventing interpolation to unvisited states.

Whitley et al. (1993) speculated that an inverse relationship exists between learning speed and generalization. In their experiments, solutions that were found in early generations tended to have poorer performance on novel inputs. Sammut and Cribb (1990) also found that programs that learn faster often result in very specific strategies that do not generalize well. This phenomenon, however, was not observed in the SANE simulations. Figure 5.2 plots the number of network evaluations incurred before a solution was found against its generalization ability for each of the 50 SANE simulations. As seen by the graph, no correlation appears to exist between learning speed and generalization. These results suggest that further optimizations to SANE will not restrict generalization.

5.1.5 Noisy Evaluation Comparisons

The final pole balancing experiment tested the ability of each approach to learn with increased noise in the fitness or evaluation function. Noisy evaluation often occurs because the number of evaluations does not provide sufficient coverage of the problem space. For example, in the inverted pendulum problem the evaluation function consists of a single attempt to balance the pole from a random start state. However, if the pole was in a difficult start state the evaluation score might not reflect the actual ability of the network. Thus, noise exists between the evaluated fitness and the true fitness of the network. Since evaluations in the large problem spaces of real world tasks will often be very noisy, it is important to study the performance of each approach as the noise level increases.

To increase the evaluation noise in the pole balancing problem the length of the pole was extended beyond the standard 1.0 meters. With a longer pole, the angular acceleration of the pole $\ddot{\theta}$ is increased, because the pole has more mass and the pole's center of mass is farther away from the cart. As a result, some states that were previously recoverable no longer are. The controllers experience more states from which pole balancing is impossible, and consequently require more balance attempts to form an effective control policy.

Simulations were run using pole lengths of 1, 2, 3, 4, and 5 meters. Figure 5.3 plots the number of successful simulations for each approach at each pole length. One of the more glaring

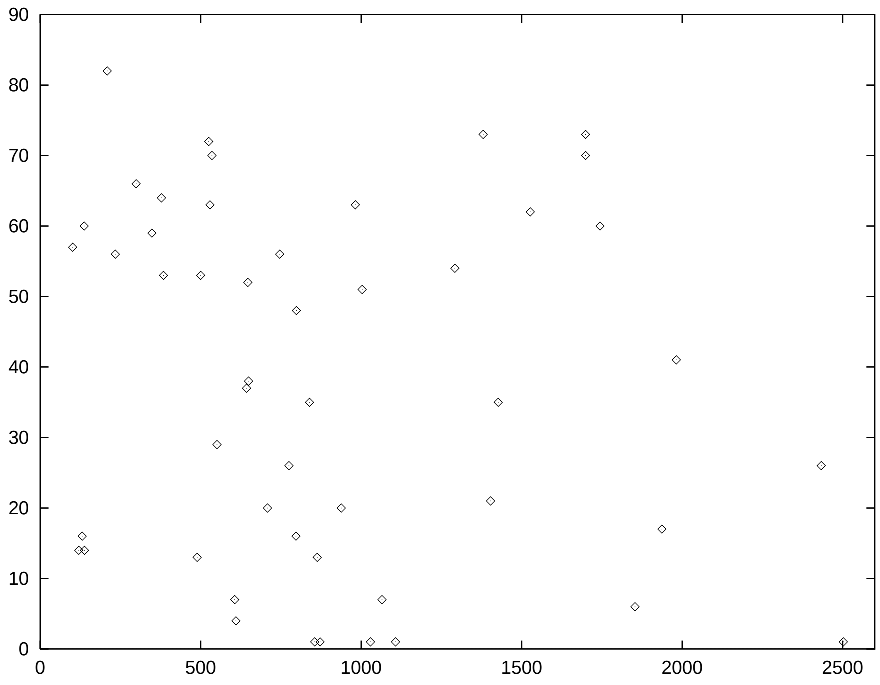


Figure 5.2: A plot of the learning speed versus generalization for the final networks formed in the 50 SANE simulations. The learning speed is measured by the number of network evaluations (balance attempts) during the evolution of the network, and the generalization is measured by the percentage of random start states balanced for 1000 time steps. The coefficient of correlation is 0.2, which indicates very little correlation between learning speed and generalization.

results of these experiments is the performance of AHC implementations under noisy evaluation. While they performed adequately under the normal problem setting, when the pole length was extended, their performance decreased dramatically. With a 2 meter pole, both approaches failed in over half of their simulations. The two-layer AHC failed in every simulation after the pole length reached 3 meters. The one-layer AHC fell to a success rate less than 20% after 2 meters and failed in every simulation at 5 meters.

Q-learning scaled significantly better than the AHCs, but significantly worse than the evolutionary algorithm approaches. Interestingly, the two table-based approaches, Q-learning and the single-layer AHC, both performed better with a 3 meter pole than a 2 meter pole. Since they are using the same "boxes" input space, it appears that the input division, tuned for a 1.0 meter pole, does not scale proportionally with the problem difficulty.

SANE was the most resilient to fitness noise, finding solutions in all 50 simulations for pole lengths up to 4 meters. At 5 meters, SANE was successful in 46 out of 50 simulations. SANE's performance advantage is due to the multiple evaluations each neuron receives. Each neuron is evaluated in several networks per generation, which results in a greater sampling of the solution space for each and averages out the fitness evaluation noise.

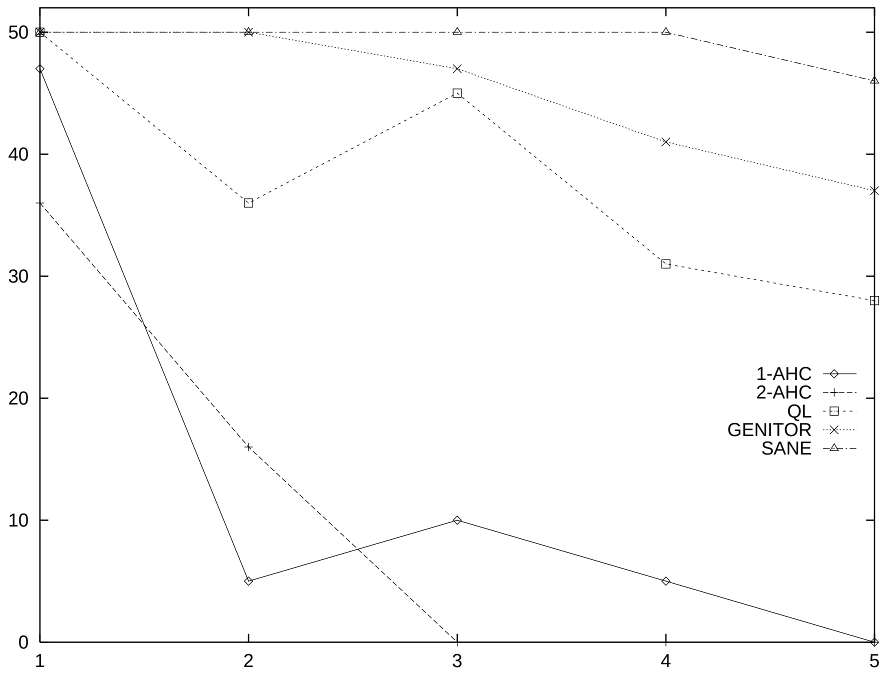


Figure 5.3: The number of successful simulations out of 50 as the pole length is increased. For a simulation to be successful, it must find a network that can balance the pole for 120,000 time steps from some start state within 50,000 evaluations.

5.1.6 Concluding Remarks

SANE was evaluated using the common pole balancing in three main areas: speed of learning, generalization, and resilience to evaluation noise. SANE required less than half of the training episodes of Q-learning and GENITOR and only one tenth of the two-layer Adaptive Heuristic Critic. The single layer adaptive heuristic critic required fewer training episodes than SANE, but had the benefit of a pre-segmented, discretized input space. In CPU time, SANE was an order of magnitude faster than both AHC approaches. Furthermore, SANE's quick solutions do not lack generalization as suggested by Whitley et al. (1993). SANE's performance is significant because pole balancing is a problem designed by the temporal difference community and set up to show the merits of their methods. The next section will present a different benchmark that is not specifically tailored to the TD methods. The benchmark is also more difficult than pole balancing which, while historically interesting, has become relatively easy for modern methods.

5.2 Khepera Benchmark

The pole balancing comparisons illustrate SANE's efficient learning in a common benchmark from the reinforcement learning literature. However, while SANE performed well, some of the advantages of SANE and neuro-evolution in general were not exhibited. Specifically, the quality of solutions that could be achieved and the robustness to perceptual aliasing, described in chapter 2, was not demonstrated. To examine these two issues, SANE, GENITOR, and the two-layer AHC were implemented in the Khepera simulator. Q-learning and the single-layer AHC were not implemented, because unlike pole balancing there is no obvious discretization of the input variables for the Khepera robot.

5.2.1 Solution Quality and Robustness to Hidden States

As described in chapter 4, Khepera provides an effective domain for evaluating reinforcement learning methods. The sensors and actuators reflect real world resources, and the task requires a complex balance of control. Unlike pole balancing where solutions are either successful or unsuccessful,[2] solution quality in the Khepera domain can be measured on a continuum. Neural networks are evaluated by how far on average they can move the robot from its starting position. Thus, a graded evaluation scale exists based on average Euclidean distance for each solution. A network that learns to maneuver in tight corners and accelerate on straightaways will outperform a network that simply drives at high speed everywhere.

Khepera also provides a domain where the robustness of learning methods to perceptual aliasing or problems with hidden state can be studied. The neural network controller receives a limited view of the world and cannot disambiguate its complete state. More concretely, the controller receives sensory information reflecting the immediate presence of obstacles (walls) but has no information about its position within the maze. The Khepera simulator can thus test how each reinforcement learning method performs without complete state information.

The experimental setup is identical to the performance analysis experiment discussed in section 4.2.2: using only local sensors, move as far away from your starting position as possible. Networks are trained (or evolved) for 8000 evaluations using the same learning parameters as in the pole balancing benchmark. During each network evaluation, the robot is placed in a random position in the Khepera world (figure 4.2) and the network is allowed to move the robot until it hits an obstacle or the maximum number of moves (200) is exhausted. The fitness of each network is the maximum distance the robot moved from its initial starting position. A learning curve is generated by testing the best network of each generation (according to fitness) on a 50 robot position test set. For the AHC, since there are no "generations", networks are tested after every 100 evaluations.

5.2.2 Results

Figure 5.4 plots the learning curves for the three approaches. SANE returned far better networks than both GENITOR and the AHC. Compared to GENITOR, SANE's networks averaged twice as much distance from their starting location. GENITOR's search appeared to move quickly initially but then stalled after 1500 evaluations. It is likely that the high mutation rates in GENITOR, which are used to promote diversity, prevented the search from focusing on and possibly converging to the global optimum. SANE's search was efficient throughout evolution, allowing it to continue to generate higher quality networks.

Despite numerous attempts with several different parameters, the two-layer AHC was unable to learn this task. Several reinforcement signals were generated to aid the AHC's learning:

- The distance from the last robot position, given after every action.
- The best distance attained during the trial, given after every action.

[2]In pole balancing, solutions typically either drop the pole within 100 steps or balance it forever.

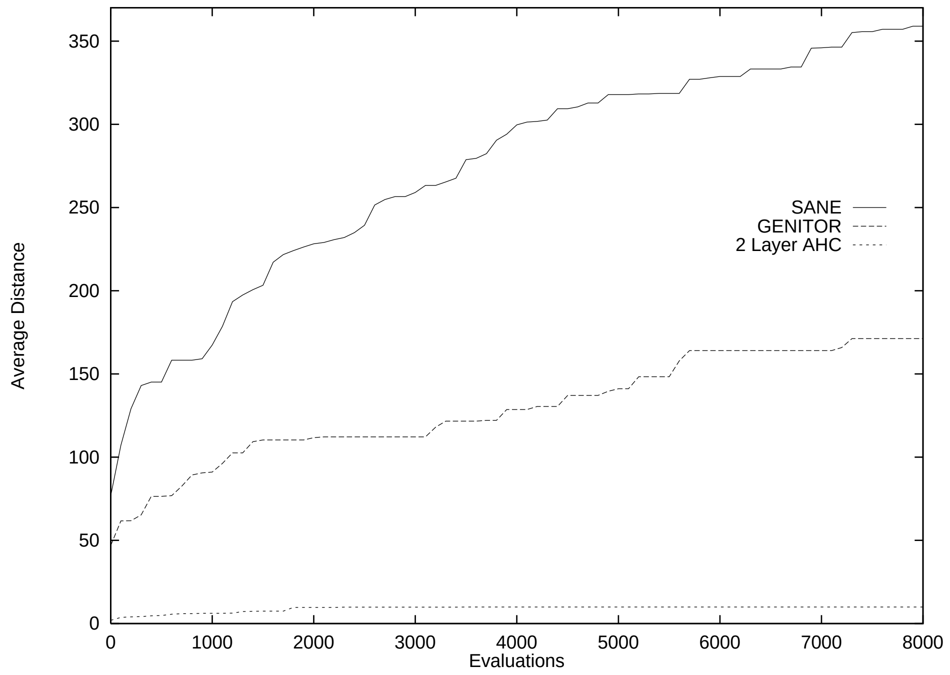


Figure 5.4: Comparison of the learning rates. The distance refers to the average distance over a 50 position test set for the best network found at or before each evaluation. The distances are averaged over 20 simulations. SANE found the most proficient solutions, while the missing state information caused the AHC to fail.

- The final best distance, given after a collision.

These extra signals were designed to boost the AHC's performance by rewarding solutions for primitive behavior. The first signal rewards the robot for moving straight and not spinning. The second and third signals reward the robot for moving further than its previous best distance. None of the aforementioned reinforcement signals had any effect on the AHC's performance, nor did any learning parameter adjustments.

The AHC's performance in this task demonstrates the weakness of the temporal difference approaches in problems with limited state information and conversely the robustness of evolutionary algorithm methods. Methods such as Q-learning and the AHC learn value functions that define the expected reward from the current state. The value function assumes that the input is sufficient to uniquely identify the state of the world (see section 2.4.3 for more explanation), which in real problems is not feasible. In the Khepera task, the sensory inputs do not adequately describe the robot's state because they give no information about the position of the robot in its world nor do they provide knowledge of the number of moves left that the robot can make.

From figure 4.2 in section 4.1, it is obvious that to accurately evaluate the expected distance (ie. the "value") from any point, a value function must know 1) the position of the robot, 2) how far the robot is from its starting point, and 3) how many moves it can make. If the robot is positioned in the tight corners of the maze, has made little progress, and has only a few moves left, its expected return is very low. Conversely, if the robot is on a long straightaway, is very far from its starting position, and has many moves left the expected return is very high. The AHC, however, makes predictions of expected returns based only on sensory information of the immediate presence of walls. A characterization of such a prediction could be: "If there are no walls in front of the robot, the expected return is 400." Clearly, there is not enough information here to make such a prediction. What if the robot is still at the starting point, has been spinning for 199 action steps, and only has one move left? Ignorant performance predictions do not take into account important factors that determine the true performance and are essentially meaningless. Consequently, the temporal difference approach makes no headway into the Khepera problem.

In contrast, evolutionary algorithm approaches do not associate sensors with values. As described in chapter 2, they learn direct mappings from sensors to actions and thus associate sensors with actions, not values. For example, a characterization of an evoluationary algorithm control decision could be: "If there is a wall in front of the robot, reverse the motors." Clearly, such action decisions can be made based on limited sensory information. By associating the *reverse* action with the forward sensor activation, an EA will develop a policy that avoids frontal collisions. Likewise, the *left* and *right* actions become associated with the right and left sensor activations, respectively. Such actions do not require positional information within the robot's world, and the EA can easily solve the task given the limited state information.

5.3 Concluding Remarks

SANE's performance in the two benchmarks demonstrates the advantages of both SANE as a tool for sequential decision learning and symbiotic evolution in general. More specifically, the benchmarks confirm the hypotheses presented in the beginning of the chapter: 1) The pole balancing simulations showed SANE to be more efficient and faster than both current approaches to reinforcement learning and a state of the art neuro-evolution system. 2) SANE's learning was shown to be more robust than the other methods when significant noise was introduced in the fitness function. 3) The generalization of SANE's solutions in pole balancing were comparable to current RL approaches. 4) The Khepera benchmark demonstrated how SANE's efficient search can evolve better solutions than GENITOR and the two-layer AHC in difficult tasks. 5) The Khepera benchmark demonstrated the robustness of SANE and conversely the weakness of the temporal difference methods without complete state information. These assets suggest that SANE could be effective in challenging and novel real world sequential decision tasks. Two such applications are described in the next chapter.

Chapter 6

Applications of SANE

One of the primary goals of this research was to build a system that can scale well and apply to decision tasks in the real world. This chapter presents implementations of SANE to two such situations. In the first task, SANE was used to filter, or focus, a minimax game-tree search. By discarding misinformation in the search tree, SANE significantly improved the play of a world champion Othello program. In the second task, SANE was applied to a difficult problem in the field of robotics. SANE successfully formed neural networks that guided a robot arm to target objects while avoiding randomly placed obstacles. Such behavior was previously only possible through sophisticated path planning algorithms. The first task in this chapter is an example of a discrete state and action space problem. The second demonstrates SANE's performance in a continuous state and action space problem. Conjunctively, the two applications demonstrate both the flexibility and scope of the SANE decision learning system. Portions of this chapter are taken from (Moriarty and Miikkulainen 1994b, 1996b).

6.1 Focusing Minimax Search

Almost all current game programs rely on the minimax search algorithm (Shannon 1950) to return the best move. Minimax operates by searching from the current game situation through all possible moves. In most games to find the best move, minimax must search search through several turns (player and opponent moves), which can be characterized as search levels. Because of time and space constraints, searching to the end of the game is not feasible for most games. Heuristic evaluation functions, therefore, are used to approximate the payoff of a state. Unfortunately, heuristics create errors that propagate up the search tree, and can greatly diminish the effectiveness of minimax (Korf 1988). In other words, minimax is only as strong as the state evaluation function, since it always assumes that the heuristic estimates are accurate.

A second drawback is that minimax also assumes that the opponent will always make the best move. It does not promote risk taking. Often in losing situations the best move may not be towards the highest min/max value, especially if it will still result in a loss. Knowledge of move probabilities could guide a search towards a more aggressive approach and take advantage of possible mistakes by the opponent.

Recently, several algorithms have emerged that are more selective than the standard fixed-depth minimax search (Korf and Chickering 1994; McAllester 1988; Rivest 1987). These algorithms allow moves that appear more promising to be explored deeper than others, creating nonuniform-depth trees. While these techniques have lead to better play, they still allow minimax to evaluate every unexplored board and are therefore vulnerable to errors in the evaluation function.

Most game programs overcome weak evaluation functions by searching deeper in the tree. Presumably, as the search frontier gets closer to the goal states, the heuristic evaluations become more accurate. While this may be true, there is no guarantee that deeper searches will provide frontier nodes closer to the goal states. Hansson and Mayer (1989) have shown that without a sound inference mechanism, deeper searches can actually cause more error in the frontier nodes. A more directed search, therefore, seems necessary.

6.1.1 Creating a Focus Window

Neural networks can be integrated with a minimax search to create a novel approach to game tree searching that can overcome deficiencies in the evaluation function and promote risk taking. At each level of minimax search, the network sees the updated game situation and evaluates each move. Only those moves that are better than a threshold value will be further explored. This subset of moves can be seen as a window to the search tree returned by the focus network. The search continues until a fixed depth bound is reached. A static evaluation function is applied to the leaf states, and the values are propagated up the tree using the standard minimax method. The α-β pruning algorithm (Edwards and Hart 1963; Knuth and Moore 1975) is used as in a full-width search to prune irrelevant states.

To illustrate how such control of minimax might be beneficial, consider the following situation. Two moves, A and B, are considered in the current board configuration. Although move A returns, through minimax search, a higher evaluation value than move B, both moves appear to lead to losing situations. Move B, however, can result in a win if the opponent makes a mistake. By assuming that the opponent will always make the best move, minimax would choose A over B resulting in a sure loss. Focus networks, however, could learn that a win can sometimes be achieved by selecting move B, and they would thus not include A in their search window.

More generally, restricting the number of moves explored has two advantages: (1) the branching factor is reduced which greatly speeds up the search. As a result, searches can proceed deeper on more promising paths. (2) The focus networks are forced to decide which moves the minimax search should evaluate, and in order to play well, they must develop an understanding of the minimax algorithm. It is possible that they will also discover limitations of minimax and the evaluation function, and learn to compensate by not allowing minimax to see certain moves.

Figures 6.1 and 6.2 illustrate the focused search process. The current player has a choice of 5 moves (a through e). Figure 6.1 shows a basic minimax search with a depth bound of 2. The leaf states are evaluated according to a static evaluation function. The actual payoff value of each leaf is shown below the depth bound. The difference between these values is the error or misinformation generated by the evaluation function. The best move is e, as it will generate a payoff of at least 11. Because of the misinformation, however, full-width minimax would choose move b. Figure 6.2 shows the same search tree but with the addition of a focus window. Only the nodes in the window are evaluated. By focusing the search away from the poor information, the best move (e) would be selected.

While it is clear how a neural network can benefit a minimax search by directing it away

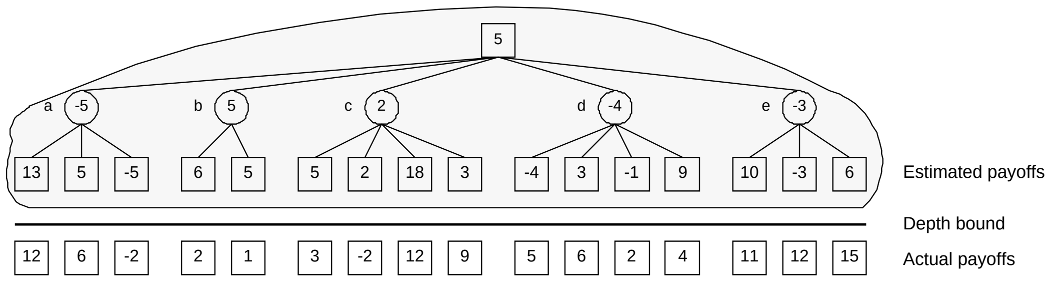


Figure 6.1: A full-width minimax search to level 2. All nodes in the shaded area are evaluated. The actual payoff values of the leaf states are listed below the depth bound. Their heuristic estimates are shown inside the leaf nodes. Min (circles) selects the lowest payoff and max (squares) the highest of min's choices. As a result, move *b* is selected for the root.

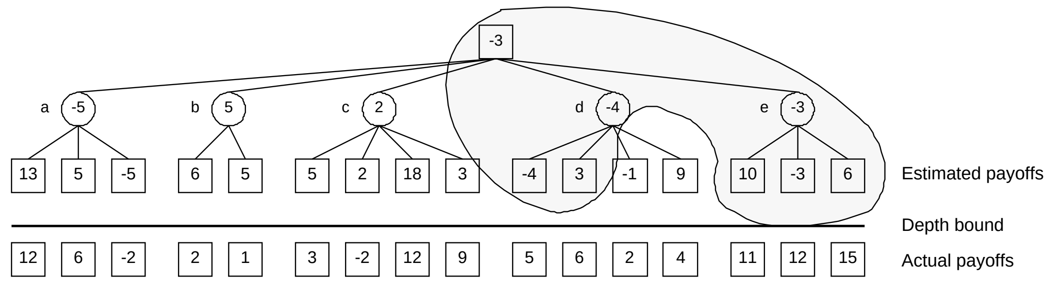


Figure 6.2: A focused minimax search. Only the states in the focus window (the shaded region) are evaluated. As a result, move *e* appears to be max's best choice.

from misinformation, forming such a network is quite difficult. The credit assignment problem exists here in grand form. The only reward returned by the domain is the final outcome of the game: a win or loss. The score or piece differential may provide additional information, but this reward must still be distributed among thousands or possibly millions of search level decisions made during the game. Since there is no obvious dispersement of game rewards, a supervised learning method is impractical. The single reward attributed to all of the search level decisions makes this an extremely challenging and important problem for reinforcement learning. A system capable of forming effective focus neural networks presents a unique and quite novel application to game-tree search and could benefit a wide range of high-level game programs.

6.1.2 Implementation of SANE in Othello

To test the effectiveness of SANE in a challenging real-world application, SANE was implemented to evolve focus networks in the game of Othello. Othello is a board game played on an 8×8 grid (figure 6.3). Each piece has one white and one black side. Players ("white" and "black") take turns placing pieces on the board with their own color facing up until there are no further moves. For a move to be legal, it must cause one or more of the opponent's pieces to be surrounded by the new piece and another of the player's pieces. All surrounded pieces are subsequently flipped to become the player's pieces. Several world championship-level Othello programs have been created using full-width minimax search (Lee and Mahajan 1990; Rosenbloom 1982). Like most advanced game programs, they achieve high performance through examining millions of positions per move.

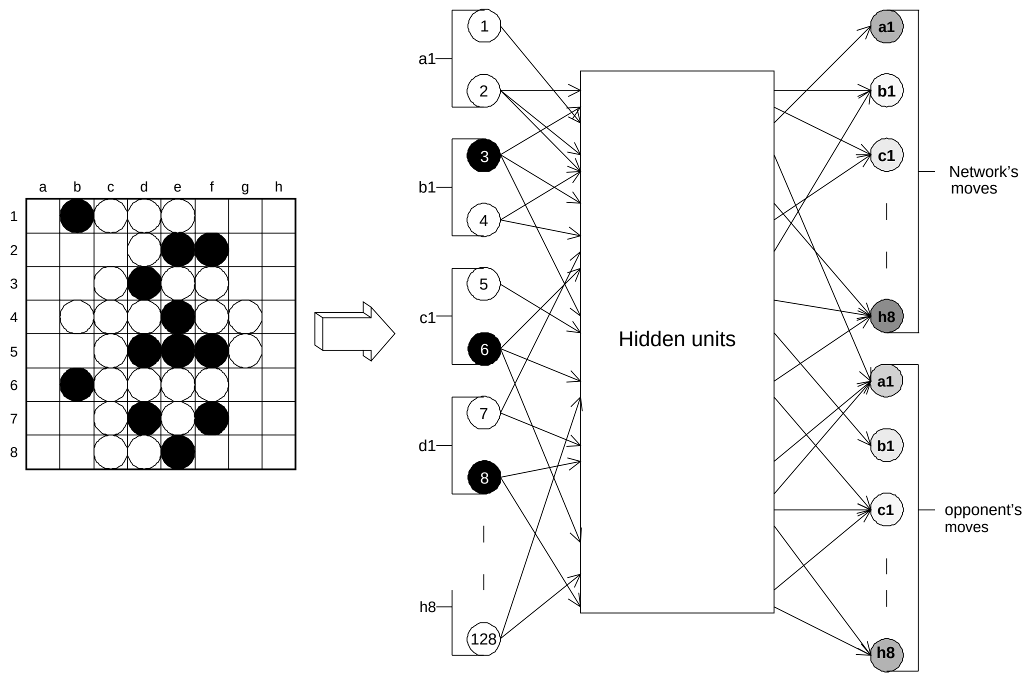


Figure 6.3: The architecture of the focus networks for Othello. Two inputs are used to encode each position on the board. The encoding of the first four spaces (a1, b1, c1, d1) for the given board with the network playing black are shown in the input layer. Both input nodes 1 and 2 are off since a1 is empty. Node 3 is on (i.e. dark) since b1 has the network's piece in it, and nodes 6 and 8 are on since the opponent has pieces in c1 and d1 (both nodes for the same position are never on simultaneously). The activation of the output layer is shown by the shading. The corners (such as a1 and h8) have high activations since corners are almost always good moves.

Two input units were used to represent the type of piece in each board space. If the space contains the network's piece, the first input unit is turned on (value = 1). If the space contains the opponent's piece, the second input unit is turned on. If the space is empty, neither input unit is activated. The two input units are never both on.

Each output unit corresponded directly to a space on the board. The activation of an output unit determined whether a move was to be considered or not. If the activation was greater than or equal to 0, the move was included in the focus window. Separate output units were used for the two players. Thus, the ranking for the network's moves may differ from the ranking of the opponent's moves. This distinction is beneficial since an aggressive player should not assume his opponent is equally aggressive and should take a more conservative approach when predicting his opponent's moves. Similarly, a defensive player should not presume defensive play from his opponents. The separation of player and opponent's output units allows offensive and defensive strategies to develop. Figure 6.3 shows an example game situation, the input the network receives, and an activation in the network's output layer.

To evaluate a network, it was inserted into an α-β search program and played against a full-width, fixed-depth minimax-α-β search. The number of wins over ten games played determined the network's fitness. To create different games, an initial state was selected randomly among the 244 possible board positions after four moves. Both players were allowed to search through the second level and used the same evaluation function. To optimize α-β pruning, node ordering was implemented based on the values of the evaluation function (Pearl 1984).

The evaluation function implemented was the Bayes-optimized function used in Bill (Lee and Mahajan 1990), which is composed of enormous lookup tables gathered from expert games. Bill was at one time the world-champion program and is still believed to be one of the best in the world. Any improvement over the current Bill evaluation function would thus be a significant result.

SANE's dual populations contained 200 blueprints and 2000 neurons. Each blueprint pointed to 25 neurons and each neuron specified 12 network connections. The number of hidden neurons and connections is small relative to the input and output layer, because SANE's hidden neurons are mainly responsible for deselecting a move from consideration. The hidden layer deselects a move by creating a negative activation in the corresponding output unit. If the output unit receives no activation, the move is still included in the focus window. Since there is most likely a larger proportion of selected moves than deselected moves, the hidden layer can be small.

The evaluation was performed over three main stages: the training stage, the validation stage, and the testing stage. In the training stage, the SANE neurons and blueprints were evolved for 200 generations, which took about 11 hours of CPU time on an IBM RS6000 25T. After each generation, the network with the best score over its ten games played was saved for testing in the validation stage. In the validation stage, the collection of best-of-the-generation networks were further tested to obtain the final network. The networks were evaluated over the complete 244 starting positions against another full-width search, however, this time each player was allowed to search through level 3. The search level was extended during the validation stage to better evaluate networks that could generalize well. In other words, each of the networks were known to perform well at search level 2 from the training stage. The validation stage narrows the field of networks by testing the top networks using a more powerful (ie. deeper) search. The top performing network over all 244 starting positions is taken as the final network to be used testing stage. In the testing stage, the final network is tested on the 244 starting positions using 1 through 6 level searches.

6.1.3 Focused Search Results

Figure 6.4 shows the best focus network's performance during the testing stage over various search levels against the full-width opponent. The results show that the minimax search with the focus network was playing a stronger game than the unfocused search. SANE's winning percentage is statistically significant ($p < .05$) at every search level, except levels 1 and 5. At level 5, SANE's winning percentage is statistically significant with $p < .09$. Of all of the games played over all levels, SANE wins 54.8% of the time, which is statistically significant ($p < .01$).

What is most remarkable about the focus network's play is that it is able to win while looking at only a subset of the states of the full-width search. Of all available moves to level 6, only 77% were included in the focus network's window. Since the full-width search is looking at the same moves as the focused search plus additional moves, there must be some misinformation in the additional moves that are causing it to select poor moves. Since the focused search employs the same evaluation function to the same depth and yet is selecting better moves, the focus network must be shielding the root from this misinformation.

To better understand how the stronger play was achieved, the moves included in the focus

Level	1	2	3	4	5	6
% of games won by SANE	52	59	57	54	53	54
Statistical Significance ($p < n$)	0.2	0.00004	0.001	0.03	0.09	0.03
Nodes searched with SANE	198	931	5808	30964	166911	939999
Nodes searched with full-width	207	977	6305	35932	212285	1217801

Figure 6.4: The winning percentage of SANE and the average number of moves searched per level. The percentage reflects the number of games won out of 488 games played. The statistical significance is shown beneath the percentage. For example at level 3, SANE's winning percentage is statistically significant with ($p < .001$).

window were further analyzed. One hundred test games were played against a full-width search using the same evaluation function. At each board position the moves in the focus window were compared with the move a full-width minimax search would return at the same position. Figure 6.5 shows the percentage of full-width minimax's moves that were included in the focus network's window. The graph thus reflects how often the focus network agrees with full-width minimax. The results show that the focus network is effectively looking far ahead. The moves in the network's window are similar to moves that a deep-searching, full-width minimax would return (black triangles in figure 6.5). However, since the network has only been evolved against a shallow-searching opponent, its predictions of the opponent's moves become less accurate as the opponent searches deeper (white circles in figure 6.5). The focus network's moves are strong because they are not tied to the moves that a full-width minimax search would choose. Instead, they reflect moves that have led to wins. It is this strong offense that allows the networks to scale with the search level. It is conceivable that eventually the network's diminishing defense will leave it vulnerable to a powerful opponent, however that was never observed in these experiments.

In the implementation described here, focus networks searched only through uniform–depth trees. Focus networks could also be implemented with algorithms such as best–first minimax (Korf and Chickering 1994), where the tree is grown in non-uniform depths allowing more promising moves to be searched deeper. Whereas the standard best–first minimax considers all unexplored board positions in the decision of where to explore next, a selective window of the most important positions could be maintained to focus the search.

6.1.4 Concluding Remarks

The results indicate that SANE can evolve better and more efficient game play through more selective search. Much like humans, focus networks selectively dismiss moves that have previously led to adverse situations. Whereas full-width minimax is very sensitive to inconsistencies in the evaluation function, focused searches can actually discover and discard unreliable information. The approach will be most useful in improving performance in domains where it is difficult to discover effective evaluation functions. SANE's decision policies can tailor the minimax search to make the best use out of the information the evaluation function provides to produce stronger overall play.

More generally, the simulations demonstrate SANE's ability to form effective decision policies in novel decision tasks. Whereas most research has improved game-playing through optimization of the evaluation function (Hansson and Mayer 1990; Lee and Mahajan 1990) or altering the minimax algorithm (Korf and Chickering 1994; McAllester 1988; Rivest 1987), SANE attacks

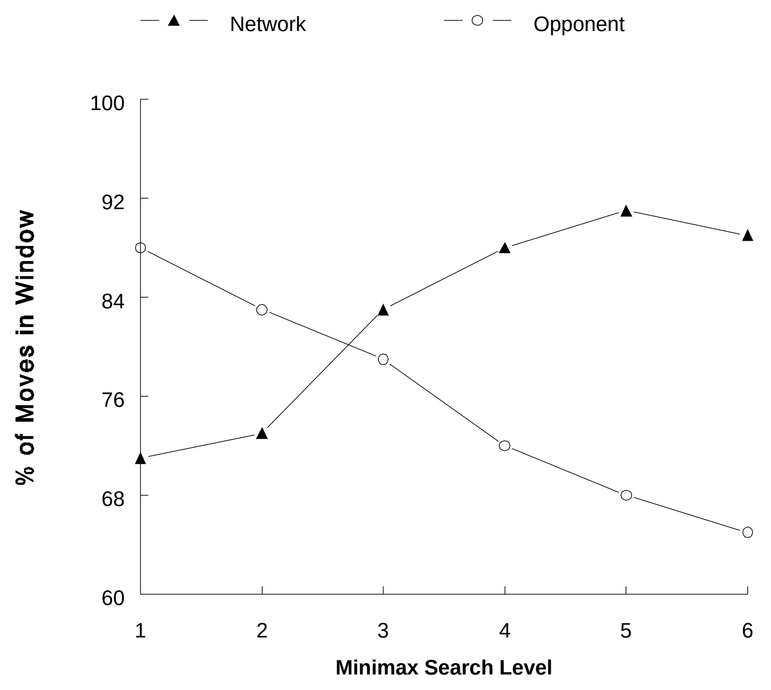


Figure 6.5: The percentage of moves returned by minimax as its choice that the focus network considers.

misinformation by making search-level decisions pertinent to the particular game and evaluation function. Such a novel approach to game-tree search, which is one of the most-studied fields in artificial intelligence, illustrates how SANE, through its generality and ability to learn with sparse reinforcement, may uncover previously unrealizable decision tasks.

6.2 Controlling a Robot Arm

In the minimax task, both the state and action spaces were finite. The state space consisted of the number of possible board configurations, and the action space consisted of the 64 possible moves. However, states and actions in many real world control tasks in fields such as robotics are not easily discretizable. A mobile robot that receives sensory input over a continuous range and must generate real-valued rotations to its motors, operates in a continuous state and action space. This section presents an application of SANE to an important real-world control problem in robotics, where both the input and output spaces are continous.

6.2.1 Robot Hand Eye Coordination

Many industrial tasks such as assembly, packaging, and processing rely heavily on the manipulation and transportation of small components. Robot arms can automate many of these processes and improve the cost efficiency of the operation. To be as effective as their human counterparts, robot arms normally use vision systems based on one or more cameras to identify and locate the target objects (Feddema and Lee 1990; Papanikolopoulos and Khosla 1993; van der Smagt 1995; Weiss et al. 1987; Wijesoma et al. 1993).

Vision-based robot arm control is a very complex task that requires mapping the information of the target and obstacle locations to the joint rotations that position the hand near the target. This task in commonly known as hand-eye coordination. Because it is difficult to specify such a mapping by hand, many researchers have applied machine learning techniques to learn the control strategy. The resulting policies are often much more robust than manually-designed, fixed policies.

Several methods have been proposed for learning robot arm control in neural networks (Kawato 1990; Kuperstein 1991; Miller 1989; van der Smagt 1995; Walter et al. 1991). Each of these methods is based on supervised learning from a corpus of input/output examples that demonstrate correct behavior. During training, the network is presented inputs from the database and its output is compared to the desired output. Errors are calculated according to the differences, and modifications are made to the network's weights based on some variant of the backpropagation algorithm. As in any supervised learning application, it is crucial that the training corpus contains a good representative sample of the desired behavior.

The most common approach for generating training examples is to flail the arm and record the resulting joint and hand positions (Werbos 1992). For example, if the joints are initially in position $\vec{J}$ and a random rotation $\vec{R}$ results in hand position $\vec{H}$, a training example of the form $Input : \vec{J}, \vec{H}; Output : \vec{R}$ can be constructed. This example reflects the correct rotation to reach target position $\vec{H}$ from joint position $\vec{J}$. Given a sufficient database of such examples, a neural network can learn to approximate the inverse kinematics necessary to translate between the camera-based visual and joint spaces.

A major limitation of generating training examples by "flailing" is that it only applies to situations where the target can be reached in a *single* rotation ($\vec{R}$) applied to the joints. It cannot demonstrate more general behavior such as reaching while simultaneously avoiding obstacles, where a sequence of rotations ($\vec{R}, \vec{R}, \vec{R}, \ldots$) are necessary. For example, when an obstacle is placed between the arm and the target, the arm cannot take a direct path, but must instead make several moves around the obstacle. Random arm movement would never produce a sufficient training example for this situation, since there is no single rotation $\vec{R}$ of the joints that can reach the target. To produce such behavior using a supervised learning approach, training examples must demonstrate movement to intermediate arm positions (e.g. above the block). It is unclear how such examples could be generated without a path-planning algorithm (Lumelsky 1987). Path-planning is an analytical approach performed offline in a complete mathematical model of the robot and its environment. Thus path-planning requires significant domain knowledge and computational resources, which may not be available in many situations.

Learning obstacle avoidance behavior without explicitly generating training examples is thus an important problem in robot arm control. SANE presents a solution to this problem by evolving the neuro-control networks. SANE offers two important advantages to robotic arm control over the standard supervised methods. First, it operates using only the overall performance of the neural network controllers as a guide. If avoiding obstacles is a necessary component of good performance, the evolutionary algorithm will select for networks that can avoid obstacles. No input/output examples are necessary, and thus neuro-evolution is not constrained by the inability to generate training examples. Second, neuro-evolution can be applied with very little *a priori* information. Knowledge of the robot arm dynamics, the arm's environment, or the components of the visual system is not necessary as they are in path-planning algorithms. Neuro-evolution evolves this

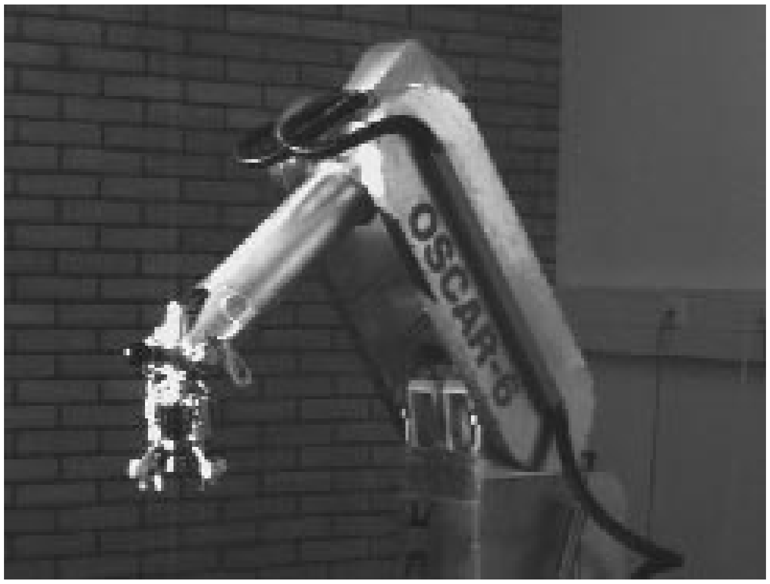


Figure 6.6: The OSCAR-6 robot arm. OSCAR is designed for pick-and-place tasks and has been applied in several industrial settings. The arm contains 6 total joints and a camera mounted in the end effector. This OSCAR-6 robot is owned by the Autonomous Systems Group at The University of Amsterdam. Printed with permission.

knowledge through experience and tailors it's control policy to meet the specific demands of the domain.

6.2.2 Evolving Obstacle Avoidance Behavior

To demonstrate how SANE can integrate obstacle avoidance into a robot arm control policy, SANE was implemented in a sophisticated robot arm simulation of the OSCAR-6 anthromoporphic robot. Figure 6.6 shows a picture of an OSCAR robot.

The network controller receives both camera-based visual and infrared (IR) sensory input representing the location of the target and the distance from obstacles, and must resolve a series of joint rotations to position the hand at a target location. The hardware specifications and algorithms used to generate the input are independent of the learning system and are considered given. For descriptions of camera-based vision and IR sensors in robot manipulators, see (Lumelsky 1987; Papanikolopoulos and Khosla 1993; Sanderson and Weiss 1983; van der Smagt 1995; Wijesoma et al. 1993). The focus of this section is how SANE can automatically integrate the sensory information into an effective control policy.

Primary and Secondary Control Networks

The task of reaching a target can be seen as a composite of two basic movements. First, the robot arm must make several large joint rotations to get within a certain proximity of the target object. Such rotations involve detecting and avoiding obstacles in the arm's path. Second, the robot arm must make smaller, more precise movements to position the end effector within grasping distance of the target object. This observation leads to an efficient design of a neuro-evolution system, where control is divided between two networks. The first, called the *primary network*, positions the arm near the target, while the *secondary network* makes the smaller movements to reach the target object.

When a task is initiated, the primary network moves the arm until it specifies that the arm should be stopped or it exceeds a prespecified maximum number of joint rotations. There is no fixed proximity boundary for the primary network; its goal is to "get as close as it can". It is possible to evolve primary networks to complete the entire task, however, because of the diminishing returns of late evolution, it is often more advantageous to accept a certain level of proficiency and begin a new search for secondary networks. Since the secondary networks start close to the target, there is normally little need for an obstacle avoidance strategy. Thus, any of the existing supervised approaches can generate effective secondary networks. Secondary networks are evolved here to demonstrate that neuro-evolution can solve the entire task.

Network Architectures

Each neural network controller (primary and secondary) contains 9 input, 16 hidden, and 7 output units (figure 6.7). The input units correspond to the x, y, and z relative distances of the hand to the target and six directional proximity sensors located on the hand that sense obstacles in the negative and positive x, y, and z directions.[1] In other words, they sense obstacles in back of, in front of, to the left of, to the right of, above, and below the end effector. They have a 10 cm range and return the absolute distance to the obstacle. If no obstacle is currently within the sensor range, the activation is 10.0.

Each joint's rotation is determined by two unique output units. The first unit is linear; the sign of its total activation specifies the direction of rotation. The second output unit is sigmoidal and specifies the magnitude of rotation. Dividing the output function this way makes it easier for hidden units to control a specific function, such as the direction of rotation of a particular joint. In the primary network, the magnitude output units are normalized between 0.0 and 5.0, limiting each joint rotation to [-5,+5] degrees. This forces the primary network to make several small joint rotations to reach the target, which allows it to more effectively sense and avoid obstacles in the arm's path. In the secondary network, the rotation is normalized between 0.0 and 1.0 to allow for finer movements near the target.

When the joint rotations are small, it is not necessary to take into account a large magnitude of distance from the target object. Such information can only interfere with the next local movement. Thus, it is very useful to "cap" the camera-based visual input units at 10.0 cm, such that if the target object is further away than 10.0 cm in any direction, the corresponding input unit receives an activation of only +/- 10.0.

A final threshold output unit is included as an override unit that can prevent movement regardless of the activations of the other output units. If the activation of the override unit is positive, the arm is not moved. If it is negative, the joint rotations are made based on the other output units. Without the override unit, stopping the arm would require setting the activation of the three sigmoidal units to exactly 0.0. Since genetic algorithms do not make systematic, small

[1]The OSCAR-6 at the University of Amsterdam (figure 6.6) does not actually have IR sensors on its hand. These experiments, however, use a simulation of OSCAR, which was modified to include IR sensors.

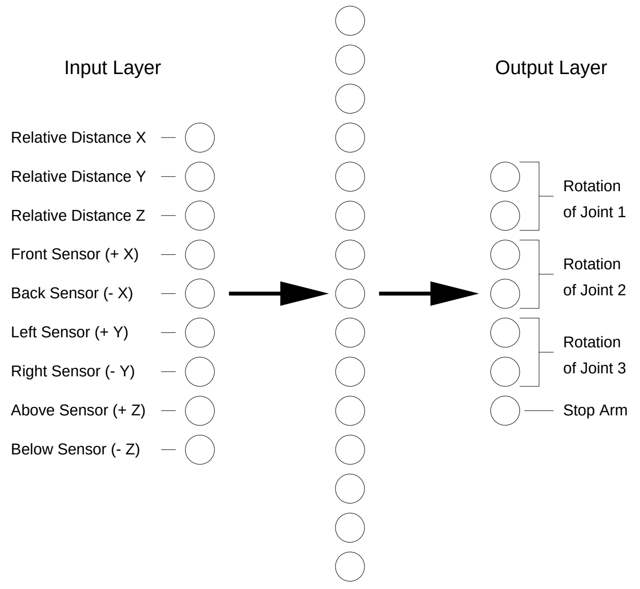


Figure 6.7: The architecture of the neural network controller for the OSCAR-6 robot. The dark arrows indicate activation propagation from the input to hidden layer and hidden layer to output layer. The connections and weights between the hidden layer and the input and output layers are evolved by the genetic algorithm.

weight changes, it is very difficult to evolve neurons to compute such exact values. The override unit allows networks to easily stop the arm when it is sufficiently close to the target.

6.2.3 Evaluation

Experimental Setup

The Simderella 2.0 package written by van der Smagt (1994) was used as the robot arm simulator in these experiments. Simderella is a simulation of the OSCAR-6 anthromoporphic arm, and van der Smagt (1995) has reported that controllers that perform well in Simderella exhibit very similar performance when applied to OSCAR. Figure 6.8 illustrates the Simderella robot. Since neither the OSCAR-6 robot at the University of Amsterdam nor the Simderella simulator use IR sensors, the simulator was modified to reflect sensor placement on the robot's hand.

Obstacles were introduced in the Simderella environment in the form of "boxes". Figure 6.9 shows the twelve different obstacle placements used in the simulations. During each trial, which consists of a sequence of moves to reach a target, one of the boxes is occupied by an obstacle. If the end effector moves into an occupied box, the trial ends, and the last position before the collision is used as the final position for fitness evaluation. This obstacle scheme is very simple, since obstacles always have the same size and there is no check if the rest of the arm (besides the hand) violates an occupied box. However, the task is still quite difficult and, to my knowledge, no existing supervised learning approach can learn the intermediate joint rotations without significant *a priori* information about the size, shape, and location of the obstacles. Thus, this task presents an important first step in learning obstacle avoidance behaviors.

Each neural network evaluation begins with random, but legal, joint positions and a random

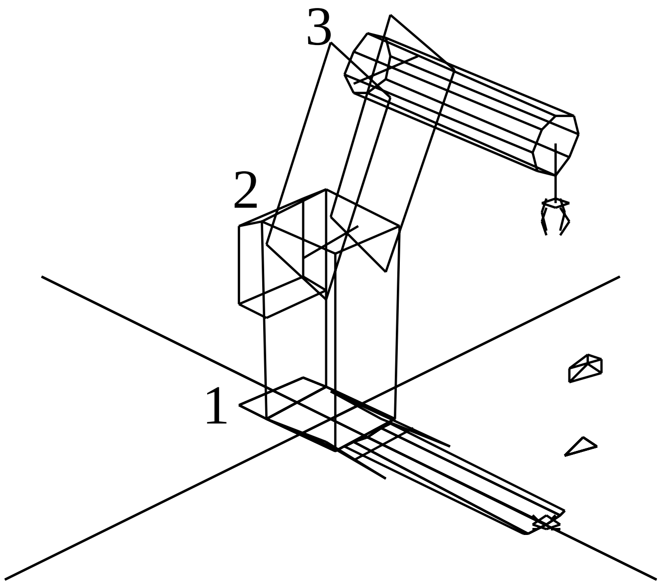


Figure 6.8: The Simderella simulation of the OSCAR robot. The numbers indicate the joints which are to be controlled. The Simderella software was developed by Patrick van der Smagt and is supported by the Dutch Foundation for Neural Networks.

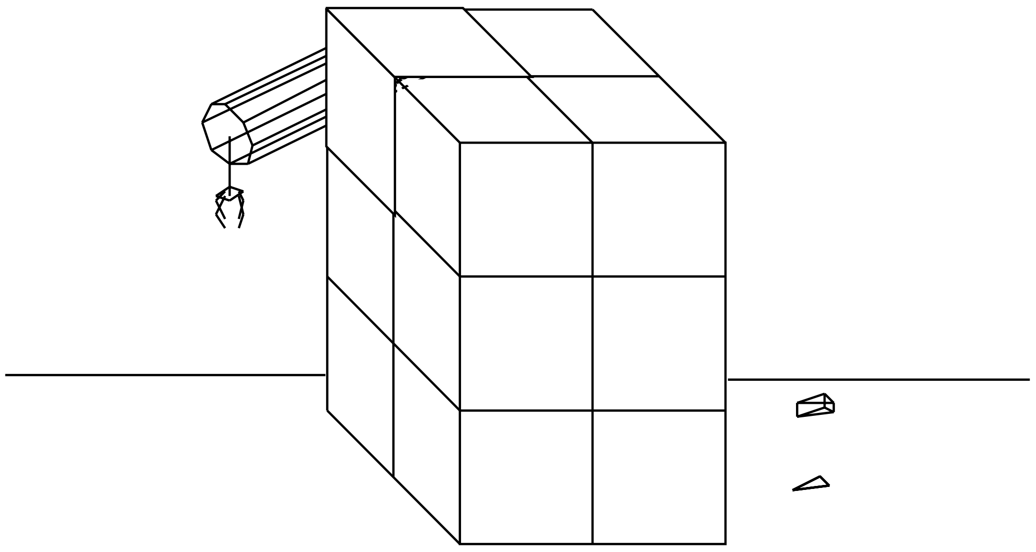


Figure 6.9: The 12 obstacle placements in the robot simulator. Each box is $30 \times 30 \times 30$ centimeters. During any trial, one box is filled with an obstacle.

target position. The target and hand are never started within an obstacle. The same reach space used by van der Smagt (1995) is employed, where targets are placed within a 180 degree rotation of the first joint. A total of 450 target positions were created and separated into a 400 position training set and a 50 position test set. During evolution, a target position is randomly selected from the training set before each network evaluation. During each trial, a network is allowed to move the arm until one of the following conditions occur:

1. The network stops the arm.
2. The network places the arm in an illegal position (e.g. hits the floor).
3. The network hits an obstacle.
4. The number of moves exceeds 40.

The score for each trial is computed as the percentage of distance that the arm covered from its initial starting point to the target position. For example, if the arm starts 120 cm from the target and its final position is 20 cm from the target, the network receives a score of $(120 - 20)/120 = 0.83$. The percentage of distance covered, instead of the absolute final distance, provides a fairer comparison between a network that receives a close target and a network that receives a distant target. Each network is evaluated over a single randomly selected target.

A population of 1600 neurons and 200 network blueprints are evolved by SANE. The first stage of evolution consists of only primary networks. The population is evolved for 200 generations, and the best network of each generation is tested over the 50 target test set. The overall best network is then fixed as the primary network, and the secondary network evolution begins from random populations of neurons and blueprints. In other words, the secondary network evolution occurs after the primary network evolution and uses a fixed primary network to make the initial joint rotations. An interesting future research question is whether primary and secondary networks can be evolved simultaneously.

Results

Figure 6.10 shows the performance of the primary networks per generation averaged over 10 simulations. The graph plots the average final distance of the best neural network found at or before each generation. On average, a network capable of moving the arm within an average of 10 cm was found in 41 generations.

Figure 6.11 shows the performance of the secondary networks per generation. Again, the graph presents an average of 10 simulations. In each simulation, the primary network was taken from a different one of the 10 primary evolutions. Within 80 generations, the secondary networks were able to position the arm with an error of only 1 cm, which is considered acceptable for most industrial applications (van der Smagt 1995). Thus, in this task, the combination of the primary and secondary networks can effectively control the robot arm to within industry standards.

It is difficult to measure how efficiently a network avoids obstacles as a function of each generation, since early networks do not hit many obstacles simply because they do not move the arm very far. Thus, counting the number of hits is a poor measure of obstacle avoidance. A better metric is the percentage of trials in which the primary network positions the arm within 10 cm

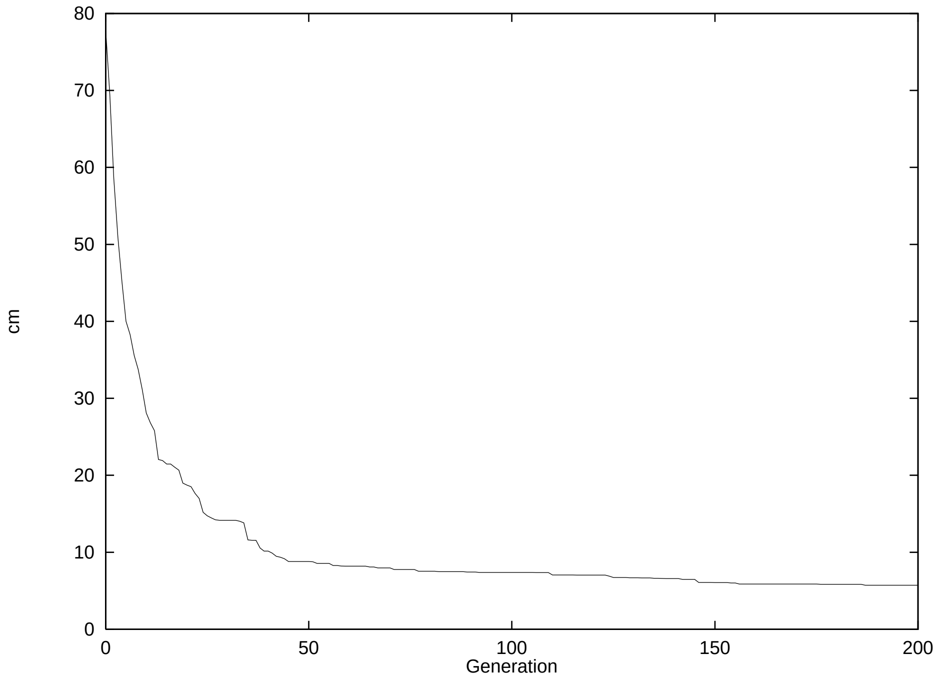


Figure 6.10: Evolution of primary networks. The average distance from the targets is plotted for the best network found so far at each generation. The curve is an average over 10 simulations. In each simulation, the distances were averaged over 50 randomly placed targets. The networks achieve the desired performance level (10 cm) after an average of 41 generations.

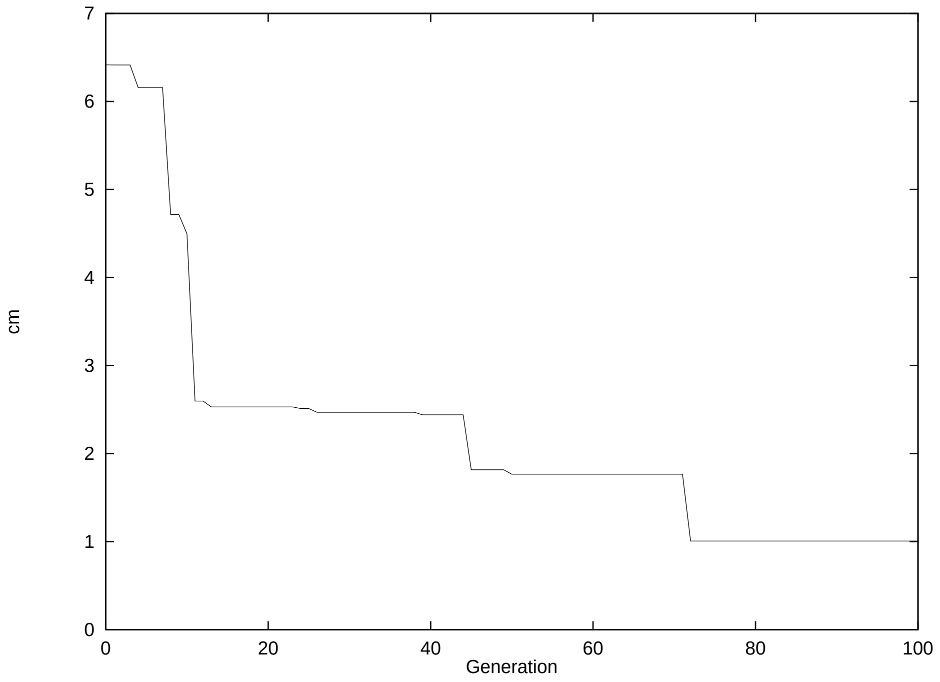


Figure 6.11: Evolution of secondary networks. The distance per generation is plotted, averaged over 50 trials and 10 simulations. Each simulation uses a different fixed primary network return from the 10 primary network evolutions.

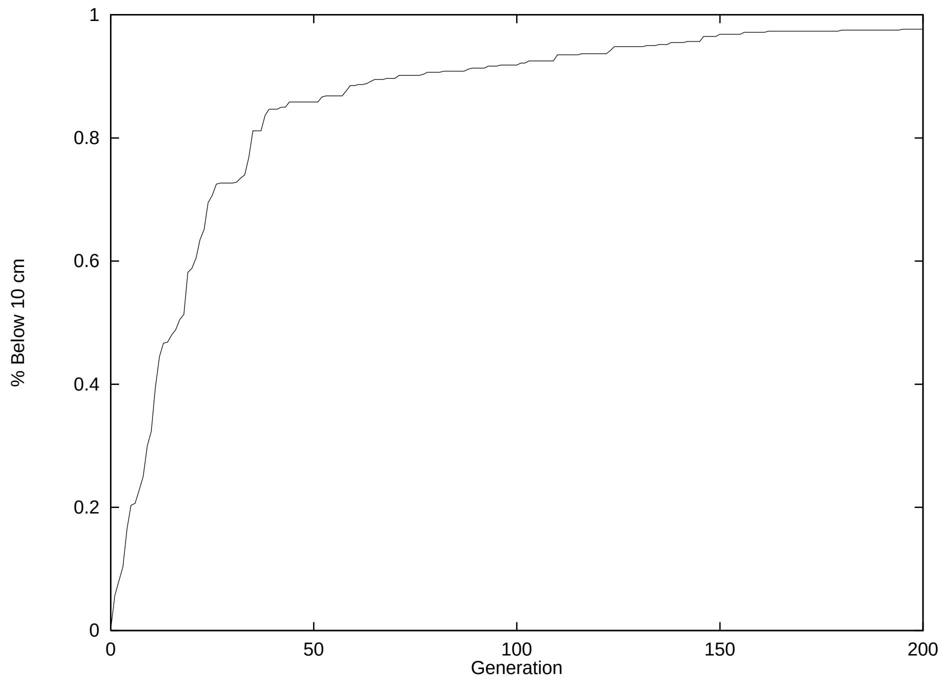


Figure 6.12: Evolution of obstacle avoidance behavior. The percentage of trials within 10 cm per generation is plotted for the primary network evolution.

of the target object. To achieve a high percentage, the network must contain a strong avoidance strategy coupled with an effective target reaching ability. Figure 6.12 plots this percentage for the best primary networks at each generation. On average, the best primary networks moved within 10 cm of the target objects 98% of the time. When collisions did occur, it was often not due to poor control decisions. With only 6 proximity sensors in these simulations, blind spots are inevitable, and occasionally a nearby obstacle can not be detected. Thus given more sensors, the primary networks should encounter even fewer obstacles.

In order to discover how often it was necessary to avoid obstacles, a manually-designed inverse-kinematics controller[2] was tested on the same 50 position test set and obstacle configurations. The fixed controller takes the most direct path towards the target and does not contain any obstacle avoidance strategies. On average, the fixed controller, hit obstacles in 11% of the trials. Since the best networks found in these simulations hit obstacles in only 2% of the trials, it follows that that significant avoidance strategies have been evolved.

6.2.4 Related Work

Davidor (1991) used a genetic algorithm to generate the intermediate joint rotations necessary to keep the end effector of a robot arm on a straight path. Davidor's work differs from the work presented in this chapter in several ways. First, Davidor used a two-dimensional robot arm simulation that was not based on a real arm. Second, Davidor evolved specific trajectories for a single, fixed path. The evolved trajectories are unique to the specific path and do not provide generalization for other arm paths. Thus, a new set of trajectories must be evolved for every new robot movement.

[2]The controller is included as part of Simderella simulator.

The method presented here evolves a single control policy for all robot movements. The final difference from Davidor's work is that Davidor's intermediate joint positions were computed to maintain a straight line of movement for the end effector. SANE's networks have the exact opposite effect: Straight lines are broken when obstacles are sensed in the arm's path. Davidor's trajectories do not contain obstacle avoidance strategies.

6.2.5 Concluding Remarks

In many industrial settings, it is crucial for a robot arm to detect and avoid obstacles in the its path. Existing methods for learning robot arm control, however, cannot learn the intermediate joint rotations necessary to move around an obstacle. By evolving neuro-controllers, such rotations can be learned since performance is evaluated over multiple control steps. Experiments in a sophisticated simulation of the OSCAR-6 robot arm showed that SANE can effectively integrate both target reaching and obstacle avoidance into a single control policy.

More generally, the experiments demonstrate that SANE can be applied to sequential decision problems with continous sensor and action spaces. Because SANE generates a direct mapping from sensors to actions (figure 2.3) and does not have to evaluate each possible action choice, it can consider an infinite number of actions for each sensor reading. These results coupled with SANE's performance in the minimax domain demonstrate the flexibility and scope of SANE for learning decision strategies.

Chapter 7

Related Work

The work in this dissertation makes novel contributions in two main areas: an efficient reinforcement learning method for sequential decision tasks and a novel coevolutionary mechanism to foster population diversity and improve search efficiency. Work related to this dissertation is separated along these two fronts. First, other evolutionary algorithm approaches to reinforcement learning are discussed. Second, other methods of coevolution are presented and compared to the symbiotic evolution in SANE.

7.1 Evolutionary Reinforcement Learning Methods

Several systems have been built or proposed for sequential decision learning through evolutionary algorithms, including both symbolic and connectionist approaches. This section highlights three of the most well-known systems.

7.1.1 SAMUEL

The SAMUEL system uses evolutionary algorithms to form a production system to solve sequential decision problems (Grefenstette et al. 1990). SAMUEL consists of three major components: a problem-specific module, a performance module, and a learning module. The problem-specific module consists of the environment and its interfaces. The performance module is a production system made up of several if-then rules that represent the decision policy. This set of reactive rules is called the *tactical plan.* Like traditional rule-based systems, the production system performs matching and conflict resolution to select the appropriate rule to fire. Additionally, the production system adjusts rule strengths based on the infrequent feedback from the environment. The learning module uses an evolutionary algorithm to develop new tactical plans. Each plan is evaluated by testing its performance in the task. New plans are created using genetic operators such as selection, crossover, and mutation. SAMUEL has been applied to several small problems including the evasive maneuvers problem (Grefenstette et al.1990) and the game of cat-and-mouse (Grefenstette 1992). In more recent work, SAMUEL has been extended to the task of mobile robot navigation (Grefenstette and Schultz 1994).

The key difference between SAMUEL and SANE is the choice of representation. SAMUEL uses a rule-based production system, while SANE uses neural networks. Which representation is more appropriate is a matter of some debate in the artificial intelligence community and beyond the scope of this dissertation. Neural networks do not require matching or conflict resolution to decide the appropriate output. On the other hand, it is easier to incorporate preexisting domain knowledge into a set of decision rules than accross the weights of a neural network. The bottom line, however, is that both provide effective generalization of the decision policy.

7.1.2 GENITOR

GENITOR (Whitley and Kauth 1988; Whitley 1989) is an aggressive search genetic algorithm that has been shown effective in reinforcement-learning problems. Whitley et al. (1993) demonstrated how GENITOR can efficiently evolve decision-making neural networks using only limited reinforcement from the domain. Since GENITOR uses neural networks to represent the decision policy, it achieves effective generalization of the state space. GENITOR relies solely on its evolutionary algorithm to adjust the weights in the neural networks and thus belongs to the ERL class of methods.

In the GENITOR reinforcement learning method, a neural network is represented in the population as a sequence of connection weights. The weights are concatenated in a real-valued chromosome along with an allele (chromosome position) that represents a crossover probability. The crossover allele determines whether the network is to be mutated (randomly perturbed) or whether a crossover operation (recombination with another network) is to be performed. The crossover allele is modified and passed to the offspring based on the offspring's performance compared to the parent. If the offspring outperforms the parent, the crossover probability is decreased. Otherwise, it is increased. Whitley et. al. refer to this technique as *adaptive mutation* which tends to increase the mutation rate as populations converge. Essentially, this method promotes diversity within the population to encourage continual exploration of the solution space.

GENITOR is considered a "steady-state" genetic algorithm (Syswerda 1991), which differs considerably from the traditional function-optimization GA. In traditional GA's, genetic operators are applied after the entire population of individuals have been evaluated. In a steady-state GA, rather than following the synchronous generation model, genetic operators are applied asynchronously often after each solution is evaluated. For example in GENITOR, after a network is evaluated, it is immediately mutated or recombined to form a new neural network. A steady-state GA may prove more adaptive than a generational GA because revisions are more frequent.

In addition to the steady-state feature, GENITOR makes three other important modifications to the canonical genetic algorithm for reinforcement learning. First, each chromosome position is a single real value rather than a binary bit. By associating chromosome positions directly with weights of the neural network, GENITOR always recombines solutions between weight definitions. Thus, GENITOR reduces the haphazard destruction of neural network weights that would result if crossover operations occured in the middle of a weight definition. The second modification is a very high mutation rate. GENITOR relies heavily on mutation to maintain diversity and promote rapid exploration of the solution space. Finally, GENITOR uses unusually small populations of neural networks. Small populations are used to discourage different, competing neural network "species" from forming within the population. Whitley et al. (1993) argue that speciation leads to competing conventions and produces poor offspring when two dissimilar networks are recombined.

Whitley et al. (1993) compared GENITOR to the Adaptive Heuristic Critic (Anderson 1989,

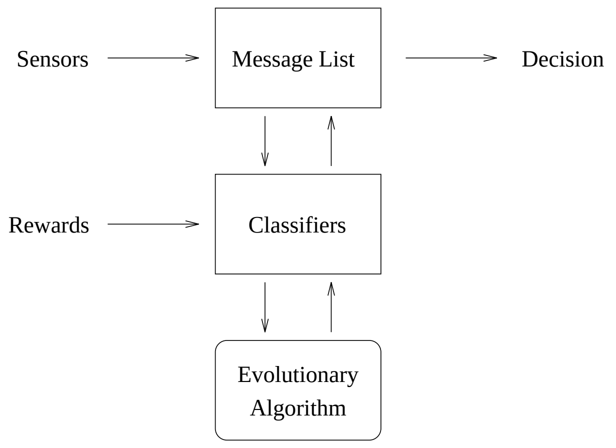


Figure 7.1: Holland's Learning Classifier System.

AHC), which uses the TD method of reinforcement learning. In several different versions of the common benchmark of balancing an pole on a cart, GENITOR was found to be comparable to the AHC in both learning rate and generalization. The only difference Whitley et. al. found was that the AHC was less consistent in solving the problem. In 5 out of 50 simulations, the AHC failed to return a successful solution.

There are several key differences between GENITOR and SANE. The first difference is that in GENITOR, each network is only evaluated once. While this can reduce the number of network evaluations, it is very susceptible to sample error often referred to as noisy fitness evaluation (Grefenstette et al. 1990). In SANE, each neuron is reevaluated in several networks per generation and in subsequent generations. Such variance results in a greater sampling of the solution space for each neuron and averages out the fitness evaluation noise. Another difference between SANE and Whitley's approach lies in the network architectures. In the current implementation of GENITOR (Whitley et al. 1993), the network architecture is fixed and only the weights are evolved. The topology of the network must be resolved a priori by the implementor. In SANE, the topology of the network evolves together with the weights, granting more freedom to the genetic algorithm to manifest useful neural structures. Perhaps the major difference between SANE and GENITOR is that SANE does not require any extra randomness to maintain diverse populations. GENITOR achieves diversity through unusually high mutation rates that produce a random point within a specific radius of the parent network (Whitley et al. 1993). Reliance on high randomness will create diversity, however, at the expense of many disruptions to important genetic building blocks.

7.1.3 Learning Classifier Systems

Learning Classifier Systems (LCS) enjoy the longest history of the ERL methods (Holland and Reitman 1978; Holland 1987; Wilson 1994). An LCS uses an evolutionary algorithm to evolve symbolic if-then rules called *classifiers* that map sensory input to an appropriate decision. Figure 7.1 outlines Holland's LCS framework (Holland 1986). When sensory input is received, it is posted on the *message list*. If the left hand side of a classifier matches a message on the message list, its right hand side is posted on the message list. These new messages may subsequently trigger other classifiers to post messages or invoke a decision from the LCS.

Because of memory and complexity issues, normally only a subset of the matching classifiers are allowed to post their messages to the message list. Determining which classifiers should activate is typically handled by a bidding system known as Holland's *bucket brigade algorithm* (Holland 1986). In the bucket brigade algorithm, classifiers maintain a strength measure and use their strength to bid against other classifiers to post their messages. Classifiers are thus in direct competition with each other to post their messages. Bids are subtracted from winning classifiers and passed back to the classifiers that posted the message that triggered them. Classifier strengths are thus reinforced if the classifier posts a message that triggers another classifier. The classifier that invokes a decision from the LCS receives a strength reinforcement directly from the environment.

Sutton (1988) noted that the bucket brigade bid passing strategy bears a strong resemblance to the theory of temporal differences. Both follow from Samuel's observation that steps in a sequence should be evaluated and adjusted based on near successors rather than on the final outcome. The bucket brigade updates a given classifier's strength based on the strength of the classifiers that fire as a direct result of it's activation. The TD methods differ slightly in this respect because they assign credit based strictly on temporal succession and do not take into account causal relations of steps. It remains unclear which is more appropriate for distributing credit.

While the bucket brigade distributes credit among classifiers within the population, it does not provide a means to generate new types of classifiers. An evolutionary algorithm is the main reinforcement learning engine that modifies the base of classifiers. The strength of the classifier is normally used as the fitness function for the EA. The EA selects, mutates, and recombines classifiers that accrue the most credit through the bucket brigade. Such classifiers are the most responsible for the good behavior of the LCS and should be selected for by evolution. Recent work, however, has suggested that a separation of classifier strength and fitness may be more appropriate to build smaller, but important niches (Wilson 1995).

Unfortunately, the progress of the LCS has been somewhat disappointing. Despite the long history of work in LCS, there are very few successful applications. Wilson and Goldberg (1989) give an excellent historical perspective of the LCS and discuss many problems that have prevented its practical implementations. Despite the disappointing progress, the LCS is a very intriguing framework because it applies ideas from both ERL and TD learning. Future work may resolve many of the difficulties in the LCS and allow it to reap the combined benefits of the two approaches. Future work may also uncover more efficient combinations of ERL and TD methods.

The LCS differs from SANE in many ways. Most notably, the LCS uses temporal differences for explicit credit assignment to individual decisions. SANE uses an *implicit* credit assignment strategy (see section 2.4.2) by evaluating and promoting entire sequences of decisions. Section 2.4.3 demonstrated how explicit credit assignment can be mislead by ambiguous state information. The implicit strategies of SANE, SAMUEL, and GENITOR are much more robust under problems of perceptual aliasing. It is likely no coincidence that the three implicit methods have achieved greater applications than the LCS.

7.2 Co-Evolutionary Genetic Algorithms

7.2.1 Co-Adaptive Genetic Algorithms

Symbiotic evolution is somewhat similar to implicit fitness sharing or co-adaptive genetic algorithms (Smith et al. 1993; Smith and Gray 1993). In their immune system model, Smith and Gray (1993) evolved artificial antibodies to recognize or match artificial antigens. Since each antibody can only match one antigen, a diverse population of antibodies is necessary to guard against a variety of antigens.

The co-adaptive genetic algorithm model is based more on competition than cooperation. Each antibody must compete for survival with other antibodies in the subpopulation to recognize the given antigen. The fitness of each individual reflects how well it matches its opposing antigen, not how well it cooperates with other individuals. The antibodies are thus not dependent on other antibodies for recognition of an antigen and only interact implicitly through competition. Horn et al. (1994) characterize this difference as weak cooperation (co-adaptive GA) vs. strong cooperation (symbiotic evolution). Since both approaches appear to have similar effects in terms of population diversity and speciation, further research is necessary to discover the relative strengths and weaknesses of each method.

Smith and Cribbs (1994) have proposed a method where a learning classifier system (LCS) can be mapped to a neural network. Each hidden node represents a classifier rule that must compete with other hidden nodes in a winner-take-all competition. Like SANE, the evolution in the LCS/NN is performed on the neuron level instead of at the network level. Unlike SANE, the LCS/NN is a pure "Michigan" approach where the entire population of neurons represents the final solution. In SANE, subpopulations represent the solution.

The LCS/NN implementation uses a variant of the cascade correlation algorithm (Fahlman and Lebiere 1990) to compute fitness levels for each neuron. Neuron fitness levels are increased if their activations correlate with correct output from the neural network. However, by basing credit assignment on the known correct behavior, the current LCS/NN implementation cannot be used for reinforcement learning. In most difficult sequential decision tasks, correct behavior is unknown.

7.2.2 Cooperative Coevolutionary Genetic Algorithms

Potter and De Jong have developed a symbiotic evolutionary strategy called Cooperative Coevolutionary Genetic Algorithms (CCGA) and have applied it to both neural network and rule-based systems (Potter and De Jong 1995; Potter et al. 1995). The CCGA evolves partial solutions much like SANE, but distributes the individuals differently. Whereas SANE keeps all individuals in a single population, the CCGA evolves specializations in distinct subpopulations or *islands*. Members of different subpopulations do not interbreed across subpopulations, which eliminates haphazard, destructive recombination between dominant specializations, but also removes information-sharing between specializations.

Evolving in distinct subpopulations places a heavier burden on *a priori* knowledge of the number of specializations necessary to form an effective complete solution. In SANE, the number and distribution of the specializations is determined implicitly throughout evolution. For example, a network may be given eight hidden neurons but may only require four *types* of hidden neurons. SANE would evolve four different specializations and redundantly select two from each for the final network. While two subpopulations in the CCGA could represent the same specialization, they cannot share information and therefore are forced to find the redundant specialization independently. Potter and De Jong (1995) have proposed a method that automatically determines the number of partial solutions necessary by incrementally adding random subpopulations. This approach appears promising, and motivates further research comparing the single population and incremental subpopulation approaches.

Chapter 8

Discussion and Future Directions

This chapter describes several research projects that have been spawned by the work in the dissertation and outlines several future research directions. The chapter is divided into three major sections. First, additional applications of SANE and systems related to SANE are presented. Second, several enhancements and proposed improvements to the SANE decision learning system are outlined. The final section discusses how the concept of symbiotic individuals can be transferred to other machine learning systems.

8.1 Other Applications of SANE

SANE has not been limited to the minimax and robot arm control applications, but has been successfully applied in other domains as well. This section describes two such applications that have been completed during my tenure at Texas.

8.1.1 Value Ordering in Constraint Satisfaction Problems

One of the first applications of SANE was for search control in a difficult class of artificial intelligence problems called constraint satisfaction problems (Moriarty and Miikkulainen 1994a). Constraint satisfaction problems are common in many areas of computer science such as machine vision, scheduling, and planning. A CSP generally consists of a set of variables and a set of possible values for them. The variables must be bound such that none of the constraints in the problem are violated. Most CSP solution methods are based on depth-first search with backtracking. When variables are instantiated, constraints are propagated forward, which either constrains the possible values for other variables or produces a contradiction. If a contradiction is found, the search backtracks and alternative variable bindings are tried. Since the order in which variables and values are bound affects when a contradiction is found, choosing variable and value binding wisely can have a significant impact on the time required to find a solution.

SANE was implemented to direct a depth-first search in the car sequencing problem (Van Hentenryck et al. 1992; Parrello et al. 1986), which is known to be NP-Complete. At each level of the search, SANE's networks decided the order in which values should be applied to variables. More specifically, SANE decided the order that classes of cars should be applied to an assembly line.

SANE used only the number of backtracks over an entire depth-first search as the fitness for each network. After 100 generations, the SANE networks required 1/30 of the backtracks of random value ordering and 1/3 of the backtracks of the commonly-used maximization-of-future-options heuristic. SANE's performance in this applications further demonstrates its applicability to important real-world problems.

8.1.2 Controlling Chaos

Weeks and Burgess (1996) applied SANE to the difficult task of controlling chaos in unstable systems. Chaos is dynamical behavior that is unpredictable over long periods of time, but obeys simple laws. Chaos can be controlled by applying small perturbations to system variables to achieve stability around a fixed point. Once the chaotic behavior is controlled, future system behavior can be more easily and accurately predicted. Weeks and Burgess demonstrated how SANE can efficiently evolve neural networks to control chaos in several unstable systems. Unlike existing methods that require knowledge of the underlying system dynamics, SANE's formed networks simply through trial and error experimentation using only the relative stability of the system as feedback. Moreover, SANE is the only method that has been shown to stabilize a system that is far from its stable state. The results show that SANE is quite effective at controlling chaotic behavior and should be applicable to many unstable systems including systems that are poorly understood.

8.2 Learning Enhancements

There are several learning enhancements that can be made within SANE and its applications to increase search efficiency and scalability in more difficult problems. Many of these enhancements are actively being explored by members of the neural networks research group at The University of Texas. The primary enhancements include seeding the population with good initial behaviors, learning complex behaviors by incrementally increasing the problem complexity, implementing adaptive online learning, and incorporating short term memory into control policies.

8.2.1 Population Seeding

SANE performs a *tabula rasa* form of learning, since searches are started from random populations which contain no domain specific knowledge. Such learning requires little a priori information or implementation effort. As reinforcement learning methods are scaled up, however, it is becoming increasingly clear that they will need aid from existing domain knowledge (Kaelbling et al. 1996). SANE has no direct mechanism to incorporate such knowledge into its initial population. Currently, the most effective way to encode domain information in SANE is through more descriptive input units or a better tuned fitness evaluation function. An important future research direction is how to seed SANE's population with existing domain knowledge. One possible method is to extract some examples of correct behavior from the domain and train a neural network using backpropagation over the training set. The network's weights could then be encoded in a chromosome and mutated to create differing individuals in a population. This seeding could help jump start SANE's search by placing it in a good region of the solution space.

8.2.2 Incremental learning

In many difficult sequential decision tasks, the problem may be too complex to evolve the desired behavior all at once. Simple behaviors may evolve that give some fitness benefits, but may not be essential to the optimal solutions. Far worse, these behaviors may in fact be detrimental to the optimal solutions. Gomez and Miikkulainen (1996) refer to these behaviors as *mechanical* behaviors. Once mechanical behaviors emerge, it may be difficult to redirect the population towards the (often opposing) desired behaviors.

An example of detrimental mechanical behaviors occurs in the game of Othello (section 6.1.2). A very simple strategy that gives immediate fitness benefits over random play is to select the move that maximizes the number of your pieces on the board. However, the desired behavior of a championship Othello program is just the opposite. The best Othello players keep their piece count quite low during most of the game to maximize the number of move options (Billman and Shaman 1990). If populations of game-playing individuals are evolved against a top Othello program from the start, they will likely never experience a winning game and will not evolve the complex strategies necessary for winning. The individuals will likely succumb to the piece maximization strategy to minimize the overall damage at the end of the game.

Previously, I showed that complex Othello strategies could be evolved in an incremental fashion (Moriarty and Miikkulainen 1995). Populations were first evolved against a simple opponent to learn some good overall winning strategies. The opponent's skill level was then increased, and the population adapted this strategy into the difficult to master mobility strategy. Similarly, incremental approaches may be necessary in other difficult tasks to avoid convergence on less desirable behaviors. Gomez and Miikkulainen (1996) explored the advantages of incremental evolution in several difficult problems, including enemy avoidance, catching a prey, and balancing multiple poles. They found that problems that could not be solved through direct evolution could often be solved using an incremental approach.

8.2.3 Online or Local Learning

All of the neural controllers presented in this dissertation can be characterized as *fixed adaptive controllers* (Werbos 1992). That is, once the controller is evolved and placed in its task it does not change. Evolution may create new controllers, but these merely replace the existing controller. Since evolution can often take too long to modify control policies, it is necessary to develop malleable controllers that can make small adaptations during the task. Such behavior can be characterized as online or *local learning.* Essentially, local learning is an inner loop learning mechanism inside of the outer loop evolution. Local learning can more quickly adapt control policies to the specific domain and may greatly speed up evolution. One can imagine the outer loop evolution searching for good areas of solution space and the inner loop learning mechanism making the smaller movements to reach the global optimum within that space. These smaller refinements relieve the outer loop evolution from the task of locating the exact global optima, which is often difficult for an evolutionary algorithm.

Since the domains of interest are sequential decision tasks, the local learning mechanism must be capable of adjusting policies under general and infrequent reinforcements. One possibility is to use a temporal difference method to form value functions that critique the decision policies created by the outer loop evolution. While the value functions may be unable to solve the task alone due to perceptual aliasing and function approximation issues, their feedback may be sufficient to make small yet important refinements to the neural network controller. This system would essentially be the Adaptive Heuristic Critic inside of SANE. SANE would generate the new neural network controllers and perform the primary search movements. The AHC would make smaller refinements to the controllers and optimize the specific area of the solution space. This approach suggests a very interesting combination of temporal difference and evolutionary algorithm approaches.

Another solution to local learning is to evolve neural networks that can provide accurate training signals for themselves. Nolfi and Parisi (1995) have developed a method where evolved neural networks compute both the desired action and training signals for each of the output units. The training signals are used to compute errors from which standard supervised learning methods such as backpropagation can be used to adjust the network's weights. Nolfi and Parisi have shown this to work well in small problems, and it will be interesting to see if it can be scaled up to larger applications.

8.2.4 Short Term Memory

All of the tasks described in this dissertation have been *Markovian.* The Markov assumption states that future behavior of the system is only dependent on the current state and future inputs and is independent of the past states. While many interesting problems can be formulated to satisfy the Markov property, in many real-world applications factors outside the current observable state or the path on which the current state was achieved may also influence the behavior of the system. For example, in chess move decisions are often based not only on the current board configuration, but also on the opponent's apparent strategy up to that point. Furthermore, while many tasks are Markovian in nature, they may appear non-Markovian to the decision making agent because it does not have access to all of the state information. As described in section 1.1.2, such tasks are termed inaccessible. If a decision making agent can remember several state sequences, however, it can more easily disambiguate the true state of the system as well as predict future system behavior. The incorporation of short term memory in SANE's neural networks is thus an important enhancement that could greatly increase the performance of the evolved controllers.

The most obvious way to create short term memory in neural networks is to use recurrent connections within the network to propagate values from previous network activations. Recurrent neural networks can maintain an internal state representation in their hidden layer activations which can be used as input in subsequent activations. This internal state represents the network's memory since it is generated and maintained over several previous activations. Since SANE uses an evolutionary algorithm to search for neural network controllers, evolving recurrent neural networks is very straightforward. In contrast, forming a recurrent network using temporal difference methods may not be feasible, because training recurrent connections from gradient information (i.e. backpropagation) requires many error propagations through various network states, which is very costly (Williams and Zipser 1989). Neuro-evolution does not require error propagation procedures and can thus form recurrent neural networks without additional overhead to the learning algorithm.

Recurrent networks, however, do bring up several interesting issues within SANE's symbiotic populations. Currently, each neuron only connects to the input layer or the output layer. Thus, the neuron connects the same way and performs the same function no matter what other hidden neurons are connected with it. With recurrent connections, however, this is no longer true. Since a neuron may receive input from another hidden neuron or an output unit influenced by another hidden neuron, its function is highly dependent on the hidden neurons that are implemented with it. It is therefore essential for a neuron to achieve some degree of reliance or expectation of the activations propagated by other neurons. Since SANE operates by continually combining neurons with different types of neurons, such expectations may be difficult to achieve. Gomez and Miikkulainen (1996) developed a neuro-evolution system called ESP (Enforced Sub-Populations) that is based on SANE, but designed to evolve recurrent networks. The important change ESP makes is to enforce segregated subpopulations within SANE's single population so that neuron combinations and connections are more consistent. Neurons should therefore evolve with a certain level of expectation of the activations of other neurons and hopefully utilize such activations as short term memory. Gomez and Miikkulainen (1996) demonstrate their approach in a pursuit and evasion task that requires knowledge of previous activations to perform well.

8.3 Symbiotic Evolution in Machine Learning

Symbiotic evolution is not unique to connectionist systems, but may provide useful insight in other areas of machine learning as well. One application is to employ symbiotic evolution to evolve a rule base for multi-category classification. Current machine learning techniques do not directly induce shared intermediate concepts between multiple categories, but instead typically re-invent intermediate states for each category. For example, in an animal classification domain, the mammal concept is normally not shared between zebra and giraffe, but is learned separately for each specific mammal. Shared concepts, however, are advantageous because they can increase the classification accuracy for each category by applying general knowledge attained about one category to a related, but possibly more unfamiliar category (Ourston and Mooney 1994).

Symbiotic evolution, however, is capable of forming shared intermediate concepts by simultaneously evolving rules which are used to classify multiple categories. From an initially random rule base, subpopulations of rules could be selected to form a domain theory. The domain theory could then be evaluated through theory refinement (Ourston and Mooney 1994) which measures both the accuracy of the domain theory and the amount of refinement necessary. The evaluation score of the domain theory would be given to each participating rule and the process of selecting and evaluating random subpopulations would repeat. Once each rule has an average utility measure, crossover and mutation operators would be applied to form a new rule base. Since sharing intermediate states generally requires less theory refinement and can produce more accurate classifiers, evolutionary pressures will select cooperative rules which connect together and form shared intermediate concepts.

While I believe that symbiotic evolution is a general principle, applicable not only to neural networks but to other representations as well, not all representations may be compatible with this approach. Symbiosis emerges naturally in the current representation of neural networks as collections of hidden neurons, but preliminary experiments with other types of encodings, such as populations of individual network connections, have been unsuccessful (Steetskamp 1995). An important facet of SANE's neurons is that they form complete input to output mappings, which makes every neuron a primitive solution in its own right. SANE can thus form subsumptiontype architectures (Brooks, 1991), where certain neurons provide very crude solutions and other neurons perform higher-level functions that fix problems in the crude solutions. Preliminary studies in simple classification tasks have uncovered some subsumptive behavior among SANE's neurons. An important focus for future research will be to further analyze the functions of evolved hidden neurons and to study other symbiotic-conducive representations.

Chapter 9

Summary and Conclusions

This dissertation has presented a novel approach for generating effective decision strategies in complex problems using evolutionary algorithms. Evolutionary approaches to reinforcement learning are quite different from the more standard temporal difference approaches. While there are many subtle contrasts, the primary difference exists in the level of credit assignment to individual decisions. Whereas temporal difference approaches explicitly distribute and assign credit to each individual decision, evolutionary algorithms do so only implicitly by selecting against poor strategies. Chapter 2 demonstrated how the implicit credit assignment of evolutionary algorithms can avoid some of the pitfalls that temporal difference methods face when the sensory input does not adequately cover all of the features of a state.

The decision learning system developed in this dissertation called SANE combines evolutionary algorithms with artificial neural networks. SANE's decision policies are represented by neural networks, which provide constant storage, constant computation time, and generalize decisions from one situation to another. SANE uses an evolutionary algorithm to adjust the neural networks, which allows it to learn tasks with only minimal direction from the environment. SANE's evolutionary algorithm is unique in that it maintains a diverse collection of individuals and effectively decomposes the search for complete solutions into a search for partial solutions. Consequently, SANE's search is very efficient and less susceptible to premature convergence.

Simulations using the Khepera mobile robot simulator confirmed the efficiency SANE's evolutionary algorithm. SANE was compared to three other neuro-evolutionary approaches: an aggressive standard neuro-evolution, a less-aggressive standard neuro-evolution, and a version of SANE without the outer loop blueprint evolution. In different experiments, SANE was shown to learn faster, maintain higher levels of diversity, and adapt quicker to changes in the environment. Further experiments in the robot simulator using a principal component analysis and artificial lesion studies showed both the emergence of specializations within SANE's population and the different roles the neurons assume. These results confirm the initial hypothesis that SANE maintains a diverse collection of neurons in several subpopulations that define different roles in the neural network. The emergence of specializations allows the evolutionary algorithm to search several decompositions of the neural network space in parallel.

SANE's performance relative to existing methods for sequential decision learning was demonstrated in chapter 5. SANE was compared to several temporal difference approaches and GENITOR, a fast neuro-evolution system. Using the common reinforcement learning benchmark of pole balancing, SANE outperformed each approach in average learning speeds, learning consistency, and resilience to noisy evaluation. Moreover, SANE's quick learning did not sacrifice generalization to unfamiliar problem states. In the Khepera mobile robot simulator, the advantages of the evolutionary algorithm approaches in problems with hidden state information were more clearly illustrated. While both SANE and GENITOR found effective solutions in all simulations, the two-layer Adaptive Heuristic Critic could not make any headway. Compared to GENITOR in the Khepera task, SANE formed more profitable solutions in less time.

The scope and scalability of SANE was demonstrated in two applications: game playing and robot arm manipulation. In a minimax search, SANE was implemented to evolve neural networks that made search level decisions to avoid misinformation in the search tree. The SANE-directed searches were trained and evaluated using the powerful former world champion program Bill. By directing the minimax search towards the most promising paths, SANE formed networks that significantly improved the performance of Bill.

The second application demonstrated SANE in a continuous input and decision space task. SANE was implemented to manipulate a robot arm to reach target objects, while avoiding obstacles in the arm's path. Previously, this task was only possible through mathematically complex path-planning algorithms that require complete knowledge of the arm and the obstacle environment. SANE requires no such knowledge and learned to reach objects and avoid obstacles simply through trial and error movements in the environment.

The goals of this dissertation were twofold: to provide a novel and effective methodology for learning decision strategies in complex problems and to develop symbiotic evolution as a new and powerful evolutionary paradigm. Towards the first goal, SANE was shown to outperform current approaches and scale well to difficult problems. Moreover, SANE requires very little knowledge from the implementor about the underlying environment. SANE thus presents a powerful and domain-independent methodology that should extend well to many challenging and novel real-world decision tasks.

The symbiotic search strategy in SANE should also spawn additional research in evolutionary algorithms and machine learning. By reducing the functional capacity of individuals and evaluating performance in conjunction with several individuals, symbiotic evolution encourages co-operating problem specializations. Thus, the search for a single solution is broken up into a parallel search for several cooperative components of the solution. This search paradigm may extend well beyond neural networks and provide important insights into intelligent behavior in other multi-agent systems such as nodes on the Internet or cars on a highway. I believe that this research has opened up many new and exciting research directions.

Appendix A

Artificial Neural Network Background

This appendix provides a general background on artificial neural networks. Since neural networks are many things to many people, the intention of this chapter is to provide a description that best suits the work in this dissertation. Readers interested in a more comprehensive view are encouraged to read the excellent foundation text by Haykin (1994).

A loosely-put definition of an artificial neural network[1] is a simulation of the mechanisms of the brain on a computer. Neural network research is motivated by the complex and entirely different computation performed in the brain as compared to the computation in the traditional digital computer. Researchers theorize that by simulating the basic computational processes of the brain, true artificial intelligence will be realized. This dissertation takes a more moderate position that can be characterized by the following statemement: a neural network provides an efficient tool for building, executing, and generalizing decision policies.

A neural network is made up of several small computational elements called *neurons*. Neurons are interconnected with communication links such that the output of one neuron can serve as an input to another neuron. The specific interconnection of neurons is termed the *neural network architecture*. An effective and very common architecture for decision making agents is a layered approach where neurons are organized in sequential layers or groups (figure A.1). The output from one layer of neurons propagates as input to the next layer. Network architectures are termed *feedforward* if the neuron activations propagate in a single direction and *recurrent* if the activation can loop back to previous neuron layers. All of the network architectures in this dissertation are feedforward, although recurrent networks may offer some advantages and will be explored in post-dissertation work.

Figure A.1 shows a common decision-making neural architecture known as a two-layer feedforward neural network. The network contains three layers of neurons and two layers of connections.[2] The first layer of neurons is commonly referred to as the *input layer*. The neurons in the input layer, called the *input units*, receive direct stimulation from the environment. In the context of this dissertation, stimulation represents the sensory input that the decision-making agent perceives. The input send their output to the second layer of neurons known as the *hidden layer*. The hidden layer neurons perform a computation based on their input and pass their output to the

[1]For brevity, the term artificial is not used in this dissertation. All neural networks in this dissertation are artificial.

[2]There has been some discrepency in the literature as to whether the number of layers neurons or layers of connections determines the number of layers in the neural network. This dissertation uses the latter as its convention.

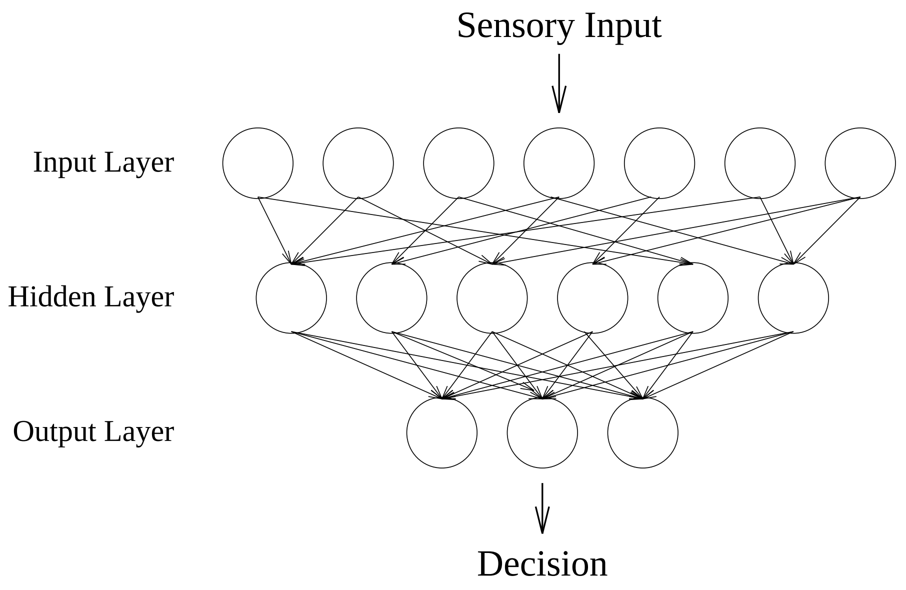


Figure A.1: A two-layer, feedforward neural network.

final layer called the *output layer.* The final computation of the neural network is normally taken directly from the neurons in the output layer.

The computation within a neuron can vary greatly among implementations, however, the most common method is to sum up all inputs received from other neurons and/or the environment and pass the sum through an *activation function.* The output of the activation function represents the neuron's output. The most popular activation function is the sigmoid function because it is simple mathematically and is biologically plausible (Haykin 1994). A common sigmoid function is defined by

$$F(v) = \frac{1}{1 + e^{-v}}$$

where v is the sum of the neuron's input.

The communication between neurons is amplified or reduced by weighting the neural connections. The output from a neuron is typically multiplied by the value of the weight and passed to the connecting neuron. Thus, the influence of each neuron on other neurons can be altered by "tuning" the weights on the connections. The connection weights are the most dynamic elements of the neural network and define how the network behaves. Consequently, most neural network learning methods focus on developing good combinations of connection weights to solve a particular task. If the weights are set properly, a 2-layer, fully-connected, feedforward neural network can approximate any continous function.

Appendix B

Additional PCA and Lesion Results

This appendix contains the additional PCA and lesion data from the Khepera simulations discussed in chapter 5. The format for the lesion data follows the data format for the first simulation, which was presented in the chapter. For explanation of the specific lesion experiments, please refer to section 4.3.2.

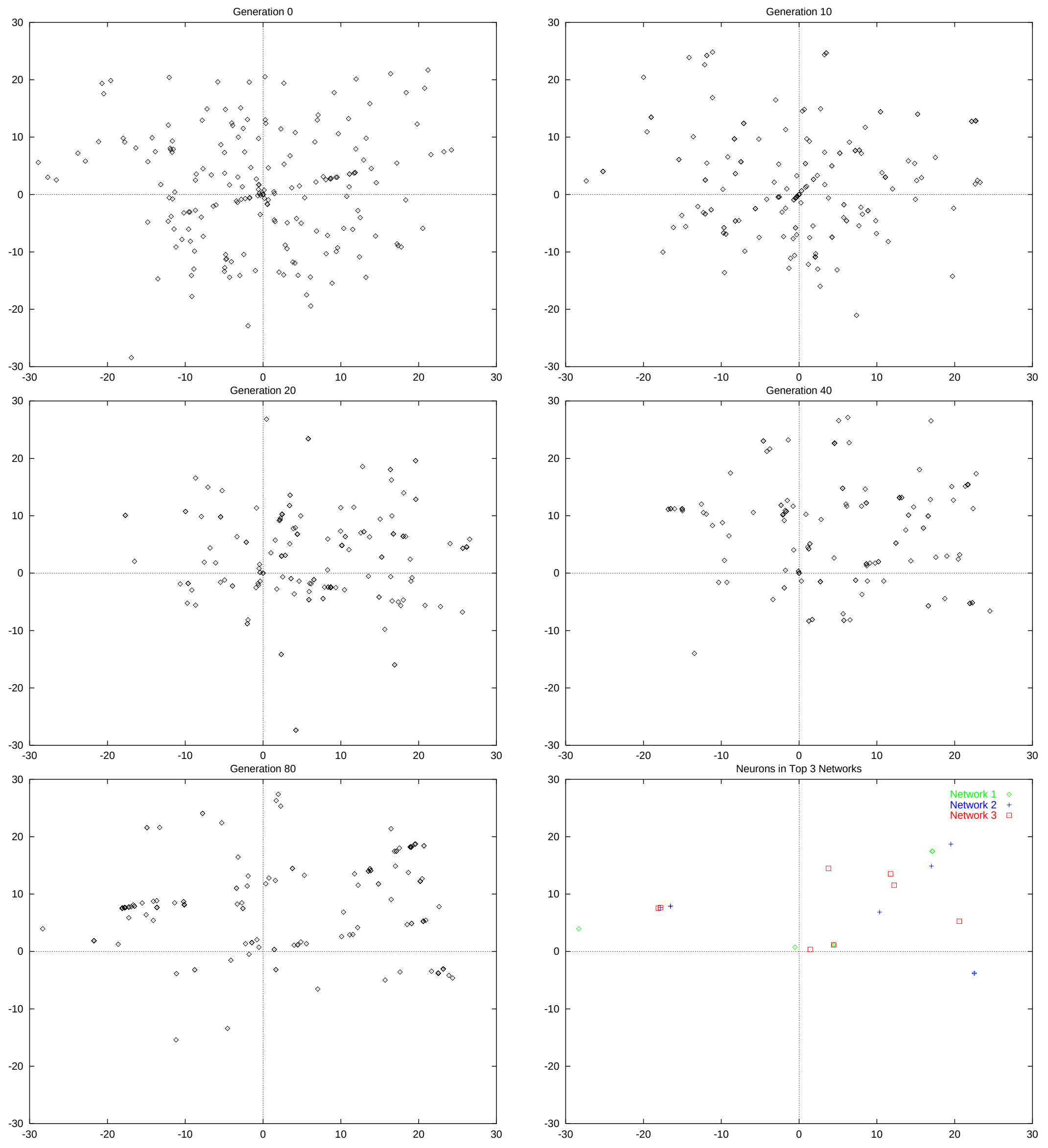


Figure B.1: PCA of simulation 2.

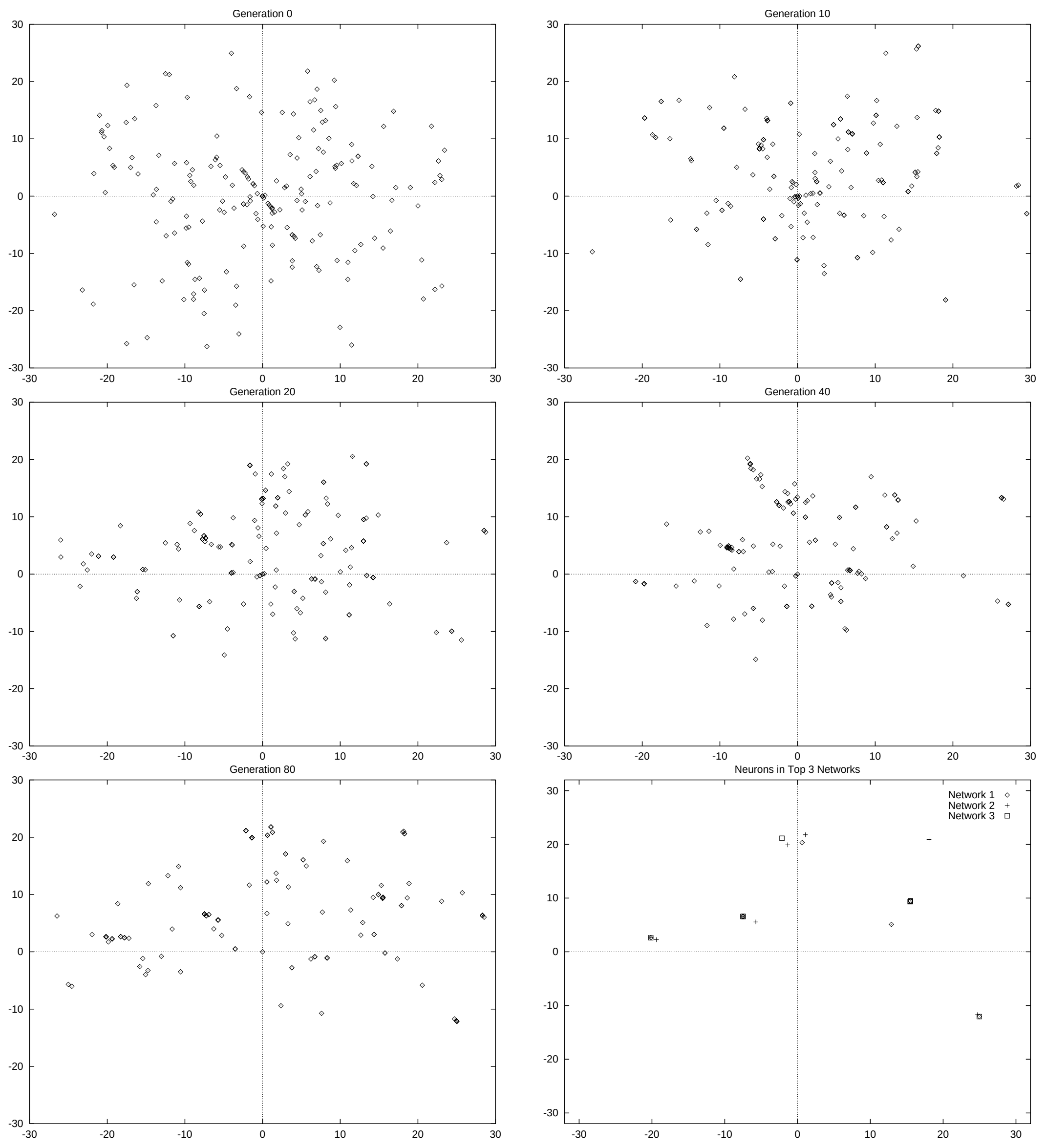


Figure B.2: PCA of simulation 3.

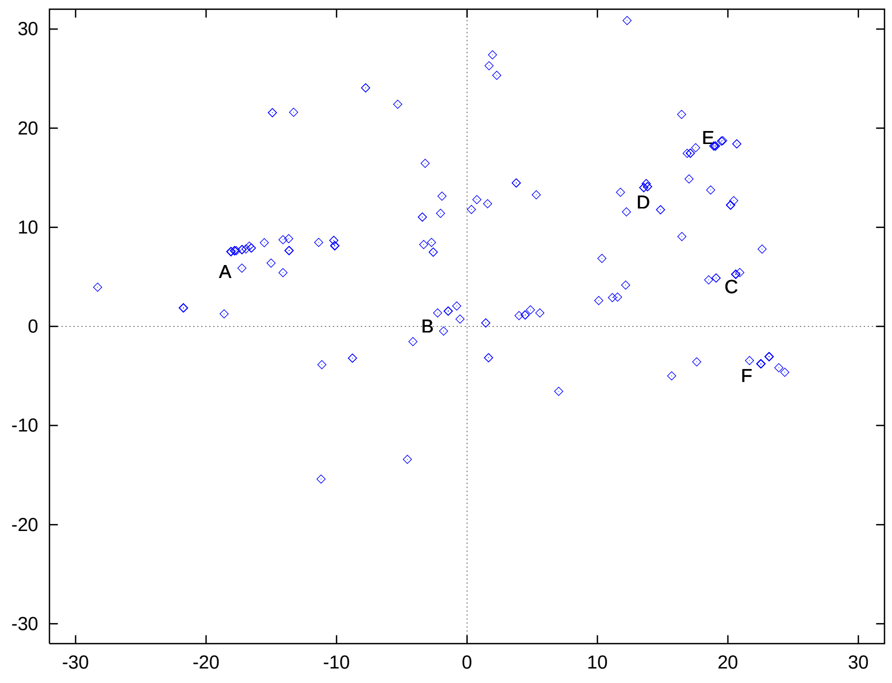


Figure B.3: PCA with labelled specializations for simulation 2.

Neuron Rank	Specialization	Capability	Necessity
73	A	6.9	293.0
2	C	20.4	332.5
72	B	35.7	307.5
23	A	14.8	304.1
1	F	14.9	193.2
14	E	44.0	195.4
16	C	19.2	311.6
15	D	50.4	328.9

Table B.1: Lesion results from simulation 2. The complete (non-lesioned) network achieved a performance level of 363.7

Lesioned Specialization	Performance
A	166.6
B	307.5
C	304.7
D	328.9
E	195.4
F	193.4

Table B.2: Specialization Lesions for simulation 2. All members of a specialization are removed from the network.

		Motor Activation from Specific Sensors				
Neuron Rank	Specialization	None	Left	Forward	Right	Rear
73	A	-6,+4	-6,+4	-6,+4	-6,+4	-6,+4
2	C	+9,-1	+9,-1	0,-1	0,-1	+9,-1
72	B	-2,0	-2,0	-2,0	-2,0	-2,0
23	A	-5,+4	-6,+4	-6,+4	-6,+4	-6,+4
1	F	+9,-5	+9,-5	+1,-1	0,-1	+8,-4
14	E	+9,+4	+9,+4	+9,+4	0,0	+9,+4
16	C	+9,-1	+9,-1	0,-1	0,-1	+9,-1
15	D	+7,+4	+9,+4	0,0	0,0	+7,+4

Table B.3: Individual neuron responses to specific sensory inputs. The output numbers refer to the speed at which the neuron drives the left and right motors.

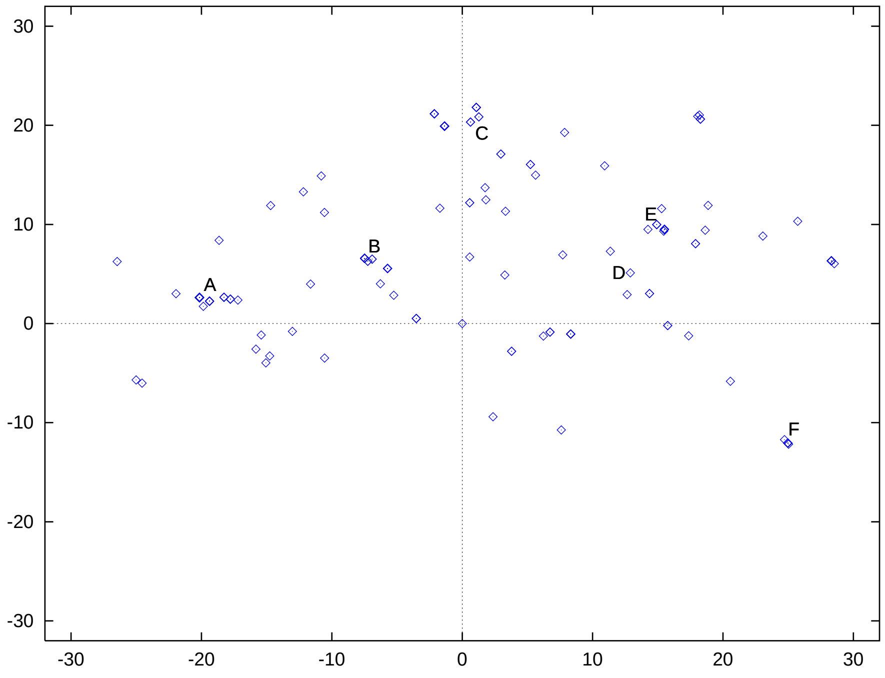


Figure B.4: PCA with labelled specializations for simulation 3.

Neuron Rank	Specialization	Capability	Necessity
3	E	26.1	296.2
2	E	29.0	276.0
25	B	18.7	310.2
1	F	12.0	216.9
26	C	37.5	371.2
24	B	18.7	319.5
32	A	7.5	62.0
33	D	25.6	407.1

Table B.4: Lesion results from simulation 3. The complete (non-lesioned) network achieved a performance level of 363.7

Lesioned Specialization	Performance
A	62.0
B	65.9
C	371.2
D	407.1
E	75.4
F	216.9

Table B.5: Specialization Lesions for simulation 3. All members of a specialization are removed from the network.

		Motor Activation from Specific Sensors				
Neuron Rank	Specialization	None	Left	Forward	Right	Rear
3	E	+8,0	+9,0	0,0	+9,0	+9,0
2	E	+8,0	+9,0	0,0	+9,0	+9,0
25	B	-1,+4	0,0	0,0	-2,+5	-1,+4
1	F	+5,-9	+5,-6	+5,-9	+5,-9	+5,-9
26	C	+5,+6	0,0	0,0	+5,+7	+6,+9
24	B	-1,+4	0,0	0,0	-2,+5	-1,+4
32	A	-6,+6	0,0	0,0	-7,+7	-6,+6
33	D	+5,-1	+5,-1	+5,-1	0,0	+5,-1

Table B.6: Individual neuron responses to specific sensory inputs for simulation 3. The output numbers refer to the speed at which the neuron drives the left and right motors.

Bibliography

Anderson, C. W. (1987). Strategy learning with multilayer connectionist representations. Technical Report TR87-509.3, GTE Labs, Waltham, MA.

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

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

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1990). Learning and sequential decision making. In Gabriel, M., and Moore, J. W., editors, *Learning and Computational Neuroscience.* Cambridge, MA: MIT Press.

Belew, R. K., McInerney, J., and Schraudolph, N. N. (1991). Evolving networks: Using genetic algorithm with connectionist learning. In Farmer, J. D., Langton, C., Rasmussen, S., and Taylor, C., editors, *Artificial Life II.* Reading, MA: Addison-Wesley.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton, NJ: Princeton University Press.

Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models.* Englewood Cliffs, NJ: Prentice-Hall.

Billman, D., and Shaman, D. (1990). Strategy knowledge and strategy change in skilled performance: A study of the game othello. *American Journal of Psychology*, 103:145–166.

Boyan, J. A., and Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7.* Cambridge, MA: MIT Press.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 183–188. San Jose, CA.

Collins, R. J., and Jefferson, D. R. (1991). Selection in massively parallel genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 249–256. San Mateo, CA: Morgan Kaufmann.

Davidor, Y. (1991). *Genetic Algorithms and Robotics.* Teaneck, NJ: World Scientific.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, The University of Michigan, Ann Arbor, MI.

Edwards, D., and Hart, T. (1963). The alpha-beta heuristic. Technical Report 30, MIT.

Fahlman, S. E. (1988). An empirical study of learning speed in backpropagation networks. Technical Report CMU-CS-88-162, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

Fahlman, S. E., and Lebiere, C. (1990). The cascade-correlation learning architecture. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, 524–532. San Mateo, CA: Morgan Kaufmann.

Feddema, J. T., and Lee, G. C. S. (1990). Adaptive image feature prediction and control for visual tracking with a hand-eye coordinated camera. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(5).

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution.* New York: Wiley Publishing.

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

Goldberg, D. E., and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. In Rawlins, G., editor, *Foundations of Genetic Algorithms*, 69–93. Morgan-Kaufmann.

Goldberg, D. E., and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, 148–154. San Mateo, CA: Morgan Kaufmann.

Gomez, F., and Miikkulainen, R. (1996). Incremental evolution of complex general behavior. Technical Report AI96-248, Department of Computer Sciences, The University of Texas at Austin.

Grefenstette, J., and Schultz, A. (1994). An evolutionary approach to learning in robots. In *Proceedings of the Machine Learning Workshop on Robot Learning, Eleventh International Conference on Machine Learning.* New Brunswick, NJ.

Grefenstette, J. J. (1991). Lamarckian learning in multi-agent environments. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 303–310. San Diego, CA: Morgan Kaufman.

Grefenstette, J. J. (1992). An approach to anytime learning. In *Proceedings of the Ninth International Conference on Machine Learning (ML92)*, 189–195.

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

Gruau, F., and Whitley, D. (1993). Adding learning to the cellular development of neural networks. *Evolutionary Computation*, 1(3):213–233.

Hansson, O., and Mayer, A. (1989). Heuristic search as evidential reasoning. In *Proceedings of the Fifth Workshop on Uncertainty in AI.*

Hansson, O., and Mayer, A. (1990). Probabilistic heuristic estimates. *Annals of Mathematics and Artificial Intelligence*, 2:209–220.

Haykin, S. (1994). *Neural Networks a Comprehensive Foundation.* New York: Macmillan College Publishing Company.

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. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach*, vol. 2. Los Altos, CA: Morgan Kaufmann.

Holland, J. H. (1987). Genetic algorithms and classifier systems: Foundations and future directions. In *Proceedings of the Second International Conference on Genetic Algorithms*, 82–89. Hillsdale, New Jersey.

Holland, J. H., and Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In *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.

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 Farmer, J. D., Langton, C., Rasmussen, S., and Taylor, C., editors, *Artificial Life II.* Reading, MA: Addison-Wesley.

Jolliffe, I. T. (1986). *Principal Component Analysis.* New York, NY: Springer-Verlag.

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

Kawato, M. (1990). Computational schemes and neural network models for formation and control of multijoint arm trajectory. In *Neural Networks for Control.* Cambridge, MA: MIT Press.

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

Knuth, D. E., and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326.

Korf, R. E. (1988). Search: A survey of recent results. In Shrobe, H. E., editor, *Exploring Artificial Intelligence*. San Mateo, California: Morgan Kaufmann.

Korf, R. E., and Chickering, D. M. (1994). Best-first minimax search: Othello results. In *AAAI-94*.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

Koza, J. R., and Rice, J. P. (1991). Genetic generalization of both the weights and architecture for a neural network. In *International Joint Conference on Neural Networks*, vol. 2, 397–404. New York, NY: IEEE.

Kuperstein, M. (1991). INFANT neural controller for adaptive sensory-motor coordination. *Neural Networks*, 4(2).

Lee, K.-F., and Mahajan, S. (1990). The development of a world class Othello program. *Artificial Intelligence*, 43:21–36.

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). Scaling up reinforcement learning for robot control. In *Proceedings of the Tenth International Conference on Machine Learning (ML93)*, 182–189. Amherst, MA.

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.

Littman, M. L. (1995). Simulations combining evolution and learning. In *Adaptive Individuals in Evolving Populations: Models and Algorithms: Santa Fe Institute Studies in the Sciences of Complexity*, vol. XXVI, 465–477. Reading, MA: Addison-Wesley.

Littman, M. L. (1996). *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University.

Littman, M. L., and Boyan, J. A. (1993). A distributed reinforcement learning scheme for network routing. Technical Report CMU-CS-93-165, School of Computer Science, Carnegie Mellon University.

Liu, Y., and Yao, X. (1996). A population-based learning algorithm which learns both architectures and weights of neural networks. *Chinese Journal of Advanced Software Research*, 3(1).

Lumelsky, V. J. (1987). Algorithmic and complexity issues of robot motion in an uncertain environment. *Journal of Complexity*, 3:146–182.

McAllester, D. A. (1988). Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, The University of Rochester.

Michel, O. (1995). Khepera simulator version 1.0 user manual. http://wwwi3s.unice.fr/ om/khep-sim.html.

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.

Miller, W. T. (1989). Real-time application of neural networks for sensor-based control of robots with vision. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4):825–831.

Minsky, M. (1963). Steps toward artificial intelligence. In Feigenbaum, E. A., and Feldman, J. A., editors, *Computers and Thought*, 406–450. New York: McGraw-Hill.

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. Kyoto, Japan.

Montana, D. J., and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Aritificial Intelligence*, 762–767. San Mateo, CA: Morgan Kaufmann.

Moriarty, D. E., and Miikkulainen, R. (1994a). Evolutionary neural networks for value ordering in constraint satisfaction problems. Technical Report AI94-218, Department of Computer Sciences, The University of Texas at Austin.

Moriarty, D. E., and Miikkulainen, R. (1994b). Evolving neural networks to focus minimax search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1371–1377. Seattle, WA: MIT Press.

Moriarty, D. E., and Miikkulainen, R. (1995). Discovering complex Othello strategies through evolutionary neural networks. *Connection Science*, 7(3):195–209.

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 neuro-controllers for hand-eye coordination and obstacle avoidance in a robot arm. In *From Animals to Animats: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB-96).* Cape Cod, MA.

Moriarty, D. E., and Miikkulainen, R. (1996c). Hierarchical evolution of neural networks. Technical Report AI96-242, Department of Computer Science, The University of Texas at Austin.

Nolfi, S., and Parisi, D. (1992). Growing neural networks. In *Artificial Life III.* Reading, MA: Addison-Wesley.

Nolfi, S., and Parisi, D. (1995). Learning to adapt to changing environments in evolving neural networks. Technical Report 95-15, Department of Neural Systems and Artificial Life, Institute of Psychology, CNR - Rome.

Ourston, D., and Mooney, R. J. (1994). Theory refinement combining analytical and emprical methods. *Artificial Intelligence*, 66:311–344.

Papanikolopoulos, N. P., and Khosla, P. K. (1993). Adaptive robotic visual tracking: Theory and experiments. *IEEE Transactions on Automatic Control*, 38(3):429–444.

Parrello, B. D., Kabat, W. C., and Wos, L. (1986). Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated Reasoning*, 2:1–42.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Reading, MA: Addison-Wesley.

Pendrith, M. (1994). On reinforcement learning of control actions in noisy and non-markovian domans. Technical Report UNSW-CSE-TR-9410, School of Computer Science and Engineering, The University of New South Wales.

Potter, M. A. (1992). A genetic cascade-correlation learning algorithm. In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, 123–133. Baltimore, MD.

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

Potter, M. A., De Jong, K. A., and Grefenstette, J. (1995). A coevolutionary approach to learning sequential decision rules. In Eshelman, L., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*. Pittsburgh, PA.

Puterman, M. L. (1994). *Markov Decision Processes - Discrete Stochastic Dynamic Programming.* New York, NY: John Wiley and Sons Inc.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

Rechenberg, I. (1964). Cybernetic solution path of an experimental problem. In *Library Translation 1122*. Farnborough, Hants, Aug. 1965: Royal Aircraft Establishment.

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

Rivest, R. L. (1987). Game tree searching by min/max approximation. *Artificial Intelligence*, 34:77–96.

Rosenbloom, P. (1982). A world championship-level Othello program. *Artificial Intelligence*, 19:279–320.

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.

Russell, S. J., and Norvig, P. (1994). *Artificial Intelligence: A Modern Approach.* Englewood Cliffs, NJ: Prentice-Hall.

Sammut, C., and Cribb, J. (1990). Is learning rate a good performance criterion for learning? In *Proceedings of the Seventh International Conference on Machine Learning*, 170–178. Morgan Kaufmann.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal*, 3:210–229.

Sanderson, A. C., and Weiss, L. E. (1983). Adaptive visual servo control of robots. In Pugh, A., editor, *Robot Vision*, 107–116. New York: Springer-Verlag.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92).* Baltimore, MD.

Shannon, C. E. (1950). Programming a computer for playing chess. *Philisophical Magazine*, 41:256–275.

Smith, R. E., and Cribbs, H. B. (1994). Is a learning classifier system a type of neural network? *Evolutionary Computation*, 2(1).

Smith, R. E., Forrest, S., and Perelson, A. S. (1993). Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1(2):127–149.

Smith, R. E., and Gray, B. (1993). Co-adaptive genetic algorithms: An example in othello strategy. Technical Report TCGA 94002, Department of Engineering Science and Mechanics, The University of Alabama.

Steetskamp, R. (1995). Explorations in symbiotic neuro-evolution search spaces. Masters Stage Report, Department of Computer Science, University of Twente, The Netherlands.

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

Syswerda, G. (1991). A study of reproduction in generational and steady-state genetic algorithms. In Rawlings, G., editor, *Foundations of Genetic Algorithms*, 94–101. San Mateo, CA: Morgan-Kaufmann.

Tanenbaum, A. (1989). *Computer Networks.* Prentice-Hall. second edition.

van der Smagt, P. (1994). Simderella: A robot simulator for neuro-controller design. *Neurocomputing*, 6(2).

van der Smagt, P. (1995). *Visual Robot Arm Guidance using Neural Networks.* PhD thesis, The University of Amsterdam, Amsterdam, The Netherlands.

Van Hentenryck, P., Simonis, H., and Dincbas, M. (1992). Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113.

Walter, J. A., Martinez, T. M., and Schulten, K. J. (1991). Industrial robot learns visuo-motor coordination by means of neural-gas network. In Kohonen, T., editor, *Artfificial Neural Networks*, vol. 1. Amsterdam.

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.

Weeks, E. R., and Burgess, J. M. (1996). Evolving artificial neural networks to control chaos. *Phys. Rev. E.* (submitted).

Weiss, L. E., Sanderson, A. C., and Neumann, C. P. (1987). Dynamic sensor-based control of robots with visual feedback. *Journal of Robotics and Automation*, RA-3.

Werbos, P. J. (1992). Neurocontrol and supervised learning: An overview and evaluation. In *Handbook of Intelligent Control*, 65–89. New York: Van Nostrand Reinhold.

Werner, G. M., and Dyer, M. G. (1991). Evolution of communication in artificial organisms. In Farmer, J. D., Langton, C., Rasmussen, S., and Taylor, C., editors, *Artificial Life II.* Reading, MA: Addison-Wesley.

Whitley, D. (1989). The GENITOR algorithm and selective pressure. In *Proceedings of the Third International Conference on Genetic Algorithms*, 116–121. San Mateo, CA: Morgan Kaufman.

Whitley, D., Dominic, S., Das, R., and Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284.

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

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

Wijesoma, S. W., Wolfe, D. F. H., and Richards, R. J. (1993). Eye-to-hand coordination for vision-guided robot control applications. *The International Journal of Robotics Research*, 12(1):65–78.

Williams, R. J., and Zipser, D. (1989). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1:87–111.

Wilson, S. W. (1994). Zcs: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18.

Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2).

Wilson, S. W., and Goldberg, D. E. (1989). A critical review of classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann.

Yao, X., and Liu, Y. (1996). Evolving artificial neural networks through evolutionary programming. In *Proceedings of the Fifth Annual Conference on Evolutionary Programming.* San Diega, CA.