Evolving Visibly Intelligent Behavior for Embedded Game Agents

Bobby D. Bryant

Technical Report AI-06-334 August 2006

bdbryant@cs.utexas.edu
http://nn.cs.utexas.edu/

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

The Dissertation Committee for Bobby Don Bryant
certifies that this is the approved version of the following dissertation:

Evolving Visibly Intelligent Behavior for Embedded Game Agents

Committee:

Risto Miikkulainen, Supervisor

Joydeep Ghosh

Benjamin Kuipers

Raymond Mooney

Peter Stone

Evolving Visibly Intelligent Behavior for Embedded Game Agents

by

Bobby Don Bryant, B.A., M.S.C.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

August 2006

This work is dedicated to all the wonderful teachers who have shaped my life and intellect.

Acknowledgments

This work was supported in part by NSF grant IIS-0083776, Texas Higher Education Coordinating Board grant ARP-003658-476-2001, and a fellowship from the Digital Media Collaboratory at the IC^2 Institute at the University of Texas at Austin. Most of the CPU time for the experiments was made possible by NSF grant EIA-0303609, for the Mastodon cluster at UT-Austin.

I would like to thank my friends and colleagues in the UTCS Neural Networks Research Group, the members of my committee, and the many wonderful people in the international AI-in-games research community, for their comments, suggestions, and probing questions during the evolution of this work. Additional thanks go to the UTCS staff for their friendship and technical support, and to our Graduate Program Coordinator Gloria Ramirez for her patient, professional, and cheerful help while guiding me through the system.

Special thanks go to Aliza Gold, Aaron Thibault, and Alex Cavalli of the Digital Media Collaboratory, for their role in funding this research and allowing me to follow it wherever it lead, and to Ken Stanley and all the volunteers on the *NERO* project for allowing me to find my natural niche in that effort.

Finally I would like to thank the FOSS development community for providing the software that was used exclusively in this research and the production of the dissertation, and for supporting it with occasional bug fixes and answers to my questions.

Civilization is a registered trademark of Infogrames Interactive, Inc.; *Pac-Man* and *Ms. Pac-Man* are trademarks of Namco Ltd.; *Pengo* is a trademark of Sega Corporation.; and *Warcraft* is a trademark of Blizzard Entertainment, Inc. *Freeciv* and *The Battle for Wesnoth* are FOSS strategy games. Images from *Freeciv* were used in the *Legion II* game described in chapter 3.

BOBBY D. BRYANT

The University of Texas at Austin
August 2006

Evolving Visibly Intelligent Behavior for Embedded Game Agents

Publication No. _____

Bobby Don Bryant, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Risto Miikkulainen

Machine learning has proven useful for producing solutions to various problems, including the creation of controllers for autonomous intelligent agents. However, the control requirements for an intelligent agent sometimes go beyond the simple ability to complete a task, or even to complete it efficiently: An agent must sometimes complete a task *in style*. For example, if an autonomous intelligent agent is embedded in a game where it is visible to human observers, and plays a role that evokes human intuitions about how that role should be fulfilled, then the agent must fulfill that role in a manner that does not dispel the illusion of intelligence for the observers. Such *visibly intelligent behavior* is a subset of general intelligent behavior: a subset that we must be able to provide if our methods are to be adopted by the developers of games and simulators.

This dissertation continues the tradition of using neuroevolution to train artificial neural networks as controllers for agents embedded in strategy games or simulators, expanding that work to address selected issues of visibly intelligent behavior. A test environment is created and used to demonstrate that modified methods can create desirable behavioral traits such as flexibility, consistency, and adherence to a doctrine, and suppress undesirable traits such as seemingly erratic behavior and excessive predictability. These methods are designed to expand a program of work leading toward adoption of neuroevolution by the commercial gaming industry, increasing player satisfaction with their products, and perhaps helping to set AI forward as *The Next Big Thing* in that industry. As the capabilities of research-grade machine learning converge with the needs of the commercial gaming industry, work of this sort can be expected to expand into a broad and productive area of research into the nature of intelligence and the behavior of autonomous agents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Autonomous intelligent agents provide a fertile field for research. They have utility in applications, they challenge the capabilities of our approaches to machine intelligence, and they allow investigation into the nature of intelligence and behavior as abstractions apart from biological instances.

The video computer game industry is business on a grand scale. For a generation it has driven improvements in the speed of desktop computers, and for almost that long it has driven improvements in graphics technology. It may soon become a driver for progress in machine intelligence as well, and with new application areas come new opportunities for research.

Thus there is a convergence of interest between research topics and commercial applications. If we wish to export machine intelligence technologies from the academic research lab to the entertainment marketplace we will be faced with new research challenges, and our research must provide new technologies to address those challenges.

Recent work has made progress at using artificial neural networks for agent controllers in the rich state/action environments of strategy games. This dissertation extends that work to the challenge of producing *visibly intelligent behavior* in autonomous agents controlled by neural networks and embedded in such rich game environments.

1.1 Motivation

For many problems the manner of solution is irrelevant: like sausage, you only want to see the end result. That applies to the work of software agents as well as humans. For instance, we have agents to search the Web and process our mail, and may soon have Cyrano-bots to compose our love letters. But the details of the workings of those systems are only of interest to their designers and maintainers; users of such systems only need the end result, and if tempted by curiosity to watch its creation, would have no intuitions about what they should expect to see, and likely as not would not understand what they saw when they did look.

However, some software agents are meant to be seen, and meant to be seen operating in ways

that are comprehensible to users of the system. For example, when intelligent agents are placed in the artificial environment of a videogame or training simulator to interact with a human player or trainee, they are often meant to be *seen* to interact, and are expected to display some comprehensible behavior that makes sense to human players and trainees, who can then create intelligent responses of their own.

In such cases the sausage-making of the software agents must be as good as the end product. It is no longer sufficient for them to score well at a game or succeed at their goals in a simulation; they must put on a good show along the way. Indeed, performing well by the ordinary metrics of success may be contraindicated: games and simulators often have difficulty-level settings to help beginners, and at the easier levels any computer-controlled rivals are *intended* to be less effective at task performance. Yet they must still be seen to be intelligent while pursuing some less-effective strategy – as if they were beginners as well.

Blind search and optimization methods such as genetic algorithms have been shown to be effective at solving a large number of problems, including the challenge of training controllers for autonomous intelligent agents. Sometimes these methods produce unexpected results, having blindly discovered a solution that would never have occurred to a human expert. For many applications such surprises can be seen as delightful quirks, providing a bonus of amusement in addition to an effective solution to the problem.

However, when the unexpected result is an unmotivated activity on the part of an autonomous intelligent agent, what might otherwise be a delightful quirk becomes a spoiler of the illusion of intelligence. It does not matter how well a simulated Elf performs at a simulated archery contest, if it exhibits behaviors that violate human intuitions regarding how Elves should behave while obtaining that level of performance. The challenge of creating a simulated Elf that scores well at archery is supplemented by the new, equally important challenge of providing Elf-like behavior to be displayed during the bout – by whatever notion of "Elf-like" the game designer wishes to pursue – and suppressing erratic, unmotivated behavior, even if it does not actually degrade the Elf's score in the bout or the broader context of the game.

Thus for environments where intelligent agents' actions are subject to observation, and human observers hold strong intuitions about what sorts of behavior are and are not appropriate for the agents, blind search and optimization for the agents' behavior must be tempered by considerations beyond raw optimization of task performance. The agents must be programmed or trained to produce *visibly intelligent behavior*.

Visibly intelligent behavior, like intelligence in general, is difficult to define (see section 2.1). It can be seen as a subset of intelligent behavior in general, i.e. the subset that is observable and evokes favorable judgements of appropriateness in the observers. Since the need for visibly intelligent behavior is motivated by the need to appease observers' intuitions, the concept is itself to a great extent inherently intuitive. Moreover, what constitutes visibly intelligent behavior varies with the context: behavior that is seen as intelligent in an Elvish archer may seem malapropos

in a Dwarvish berserker, and vice versa. Thus for present purposes visibly intelligent behavior is conceived as a list of desirable characteristics for a broad class of agents operating in observable environments.

One such desideratum is self-consistency. A rigorous and rigid self-consistency is obtainable with even the simplest of control scripts. However, the appearance of intelligence will sometimes require a *relative* self-consistency. That is, an agent must be seen to "do the same thing", mutatis mutandis, as the context changes. Moreover, self-consistency must be tempered by flexibility. For individuals, a lack of flexibility can convert a single failure at a task into a endless stream of failures; additionally, excessive predictability can often be exploited by a sufficiently intelligent rival agent, such as a human opponent in a game. For teams, a lack of flexibility in individual members can result in a gross lack of adaptivity at the team level: some of the team's needs may remain unsatisfied while the members uniformly and inflexibly service other requirements. Thus for many types of agent, visibly intelligent behavior will require flexible self-consistency, an adaptive trade-off between predictable and unpredictable behavior.

Another challenge for producing visibly intelligent behavior with machine learning methods is the suppression of unmotivated actions that will be seen by observers as merely erratic. For rich environments the possibilities for such behaviors are vast, and which of those possibilities might actually be produced by a given run of a given machine learning method is unknowable. A cycle of detecting and suppressing such behaviors by training and testing followed by tweaking the learning parameters and/or programmatic constraints for a re-run is expensive in terms of both time and human expertise. It will therefore be useful to develop machine learning methods that suppress erratic behavior implicitly, by training agents to do the right thing, without requiring excessive specification of what is and is not desirable.

This dissertation examines methods for encouraging or coercing the desirable characteristics of visibly intelligent behavior while suppressing undesirable behavioral traits, with a focus on the use of artificial neural networks as controllers for agents embedded in the rich environments of strategy games or simulators. The work supplements similar efforts using different technologies in different application areas.

1.2 Research Goals

The scripted intelligence of the agents embedded in the current generation of video games is generally unsatisfactory. The agents' behavior tends to be single-minded and brittle; they focus on the details that they were programmed to attend, often ignoring other important facets of the context, which is difficult to capture in a script. They show little regard for teamwork, and little ability to adapt to a changing situation.

There is a notion afoot in the AI-in-games research community that we can provide better embedded intelligences by means of machine learning than the industry currently does by script-

ing. Machine learning has the potential of capturing contexts and optimizing details of behavior that are difficult to capture or optimize in hand-written scripts. All the more so, we hope, as the simulated environments in games become richer and the number of possible contexts, the number of opportunities for an agent to exploit, grows exponentially.

However, it can already be demonstrated that machine learning methods do not automatically produce controllers that exhibit visibly intelligent behavior. Reinforcement methods such as neuroevolution can produce interesting behavior in embedded agents, but some of that behavior is interesting only for its malapropos nature. In principle it might be possible to constrain the results of reinforcement learning by properly and exhaustively specifying the rewards for all possible behaviors in all possible contexts, but for increasingly rich environments such specifications are likely to become as difficult as direct scripting is. We must, therefore, devise machine learning methods that produce effective behavior in embedded agents, but also limit solutions to those that can provide a visible display of intelligence while the agents are in action.

The goal of this research, therefore, is to create and examine methods for inducing visibly intelligent behavior in agents trained by machine learning. Consideration is limited to autonomous intelligent agents embedded in the rich environment of a strategy game or simulator, and controlled by artificial neural networks that map their sensory inputs onto responses. The work is founded on the popular methodology of using a genetic algorithm – neuroevolution – to train the controller networks by reinforcement based on task performance. It builds on that foundation by examining supplementary methods that produce various characteristics of visibly intelligent behavior in the agents it produces.

1.3 Approach

The challenge of creating agents that display visibly intelligent behavior is approached here by identifying important aspects of such behavior, creating a game where agents can be trained and observed for the qualities of their behavior, and using the game as a test-bed for the ability to bring out the desired behaviors by means of machine learning methods.

The selected target qualities are flexibility, adaptability, discipline, self-consistency, and a lack of excessive predictability. Some of these qualities are in tension with one another, though not mutually exclusive: all are needed for a convincing show of general intelligence in rich environments.

The test environment is a strategy game where agents can be embedded for training and testing. The game requires learning agents to cooperate with one another while competing with pre-programmed opponents. It was parameterized to require flexibility of behavior in the individual agents and adaptability in the way they organize those behaviors for teamwork. A slight modification to an off-the-shelf machine learning method was then used to induce those aspects of individual and team behavior.

However, success in that initial experiment revealed challenges regarding the other desired qualities of behavior. The blind mechanism of machine learning brought out the high-level behaviors required for optimizing game scores, and even supported generalization of those behaviors to modifications of the agents' environment. Yet lower-level details of the learned behavior, when not critical for optimizing the game score, proved to be erratic to the eye of human observers. They lacked the appearance of discipline and self-consistently while – paradoxically – being fully deterministic.

Therefore additional learning methods were applied to the same problem to modify the quality of behavior. Human-generated examples of play were used to improve the show of discipline and self-consistency in the trained agents, and a method was devised to allow a small amount of randomness to be introduced into the agents' behavior while minimizing the negative impact on their task performance.

The result of the study is a suite of machine-learning techniques useful for training autonomous intelligent agents for use in environments such as strategy games, where they are readily observable in action, and human observers often have strong intuitions about what sorts of behavior are and are not intelligent. The techniques under study induced the desired sort of behaviors in the agents; the research shows that machine learning methods can create agents that display the visible characteristics of intelligent behavior.

1.4 Overview of Dissertation

This dissertation divides naturally into three sections: introductory material (**chapters 1-2**), methods and experiments (**chapters 3-8**), and summary material (**chapters 9-10**).

The **current chapter** has introduced the dissertation. **Chapter 2** broadens the introduction by providing background material that sets up a context for the new work, and explicates some of the concepts necessary for understanding it.

The six chapters in the section on methods and experiments divide naturally into three subsections. In the first, **Chapter 3** introduces a strategy game called *Legion II*, which has been used for all the following experiments, and explains its mechanics in detail. Then **Chapter 4** provides the details of the experimental discipline used for applying machine learning methods to the *Legion II* game.

The two chapters in the next sub-section provide the key experiments for the dissertation. **Chapter 5** introduces a multi-agent control architecture that allows agents to adapt their behavior in response to their context, without requiring additional training. Then **Chapter 6** shows how example-guided evolution can be used to refine the details of the agents' behavior in ways that are intended to satisfy the viewer, rather than to optimize game performance *per se*. These two chapters illustrate how existing techniques can be used to induce flexibility and coherence – two key aspects of visibly intelligent behavior – into the behavior of agents trained by machine learning.

The two chapters in the third sub-section on methods and experiments introduce new techniques for further refinements of the agents' behavior. **Chapter 7** examines a way to exploit any symmetries in an agent, its sensors, and its environment when training examples are available, in order to improve its task performance and self-consistency by the use of artificially generated training examples. Then **Chapter 8** introduces a training mechanism called *stochastic sharpening*, which generates agent controllers that allow a controlled amount of randomization to be introduced into the agents' behavior, with minimal degradation of their task performance. These two chapters address two competing requirements – self-consistency and a lack of excessive predictability – that arise when intelligent agents are observed in a game context. The methods are also useful for improving raw task performance under some training conditions.

The third major section of the dissertation analyzes and summarizes the experimental work. **Chapter 9** examines the implications of the experiments and outlines ways in which the effects can be exploited or improved in future work. Finally, **Chapter 10** recaps the dissertation's contributions and draws a high-level conclusion about the work it contains.

Chapter 2

Foundations

This chapter provides background material necessary for comprehending the rest of the dissertation, and establishes the broad context for the new work that the dissertation reports. Section 2.1 offers some conceptual background on the way AI research is done in practice, to show how the goals of the dissertation fit in to that field. Then sections 2.2 and 2.3 provide background on autonomous intelligent agents and prior research into game applications of AI. Sections 2.4-2.5 give technical details about two technologies used in this dissertation, artificial neural networks and a mechanism for training them known as neuroevolution. Finally, section 2.6 provides a typology of the ways human biases can be used to guide the output of genetic algorithms, which will help with understanding how the work in chapter 6 relates to other potential methods of doing the same thing.

2.1 Artificial Intelligence

This dissertation is concerned with a topic that falls under the broad and ill-defined concept of Artificial Intelligence (AI). The "artificial" part is simple: the field is concerned with human artifacts rather than naturally occurring intelligence. But the "intelligence" part is conceptually difficult. Though some AI researchers with a philosophical bent offer and dispute definitions (e.g., Brooks 1991; Brooks et al. 1998; Fogel 1999), and textbook writers dutifully offer strawman definitions and call attention to the difficulty of producing a definition that is both rigorous and satisfactory (e.g., Rich and Knight 1991; Dean et al. 1995), for the most part the entire field of AI research, broadly defined, operates under a tacit *I know it when I see it* charter. Thus the Turing test defines intelligence operationally, by an appeal to human perception of intelligence, rather than offering a definition by or an appeal to some formalism (Turing 1950).

For example, we tacitly assume that planning is an intelligent activity, so if we create a system that can produce plans – however limited the context – we refer to it as an intelligent system. Similarly, if a machine learning system can train a controller to balance a real or simulated inverted pendulum, we tacitly assume that it is intelligent; if another system can train the controller faster,

or train it better by some metric such as the power consumption required by the control policy, we tacitly assume that it is *more* intelligent.

The point being made is that the research pursued in the various sub-fields that fall under the broad umbrella of Artificial Intelligence – ALife, robotics, multi-agent systems, machine learning, search, planning, etc. – is not for the most part driven by some formal or theoretical notion of intelligence. Rather, we focus our research on things that are generally agreed to be interesting problems. This fact is not offered as a critique, but as a simple observation: *for the most part, our criteria for success and progress are arbitrary*.

That observation is necessary groundwork for understanding the goals of this dissertation. Much of the work reported here does follow the conventions described above. Namely, a problem has been selected that is interesting, difficult, and presumed to be relevant to intelligence, and methods are devised for optimizing the scores obtained by an obvious metric for the problem. However, in certain places – particularly chapters 6 and 8 – the dissertation also emphasizes a new perspective on the problem, namely that *poorer scores on an obvious performance metric can actually be "better" results, when the focus shifts from the raw results of task performance to the observable behavior of an artificially intelligent system in operation*. That is, intelligence of the *I know it when I see it* sort can involve a *show* of intelligence while in operation, as legitimately as it can involve the production of a metrically superior outcome. In the absence of a rigorous and universally accepted definition of 'intelligence', intelligence lies to a great extent in the eye of the beholder.

Having called attention to this perspective, the remainder of this chapter introduces the fields and technologies that establish important background for the new work reported in the following chapters.

2.2 Autonomous Intelligent Agents

Agents are entities with sensors, actuators, and a controller that maps senses onto actions according to some policy. An agent is considered to be *autonomous* if the controller operates at its own behest rather than under the control of some other agency. It is often convenient to embed an agent's controller in a game or simulation engine for invocation as a callable routine at appropriate times, and the resulting *simulated* autonomous agent is normally treated as a genuinely autonomous agent without qualification.

Autonomous agents are categorized as either *reactive* or *deliberative* (Sycara 1998). The controller for a reactive agent behaves as a function, in the mathematical sense of the term, mapping senses onto actions in a one-to-one or many-to-one fashion. Various authors list varying desiderata for deliberative agents, but the minimal requirement is that a deliberative agent must maintain an internal state and consult it during its choice of actions. Thus the controllers for deliberative agents implement relations rather than functions. The internal state consulted by a deliberative agent can be properly viewed as a *memory*; the controller can update the state as a side effect of its choices of

actions.

Multi-Agent Systems (MASs) arise when multiple autonomous agents operate in a shared environment. The agents sense the common environment, though perhaps from different points of view, and their activities change the state of the common environment. The agents in a MAS can be cooperative, competitive, or a mixture of the two. For example, in the popular RoboCup competition (Kitano et al. 1997) each agent will cooperate with its teammates while competing against the opposing team.

Stone and Veloso (2000a) provide a general typology for MASs. The typology arranges MASs along two primary axes: homogeneous vs. heterogeneous, and communicative vs. non-communicative. For domains that require a division of labor among the team members it has become common to create customized agents for each required task and then deploy a customized mixture of such agents to satisfy the requirements of the task presented by the domain under study (e.g., Balch 1997, 1998; Haynes and Sen 1997; Yong and Miikkulainen 2001). Bryant and Miikkulainen (2003) suggested a reversion to homogeneity with the *Adaptive Team of Agents* MAS architecture; that architecture is re-presented and explored more thoroughly in chapter 5 of this dissertation.

When an agent's controller operates a mechanical device such as a robot, the agent can be said to be *embodied*, and we generally think of the entire software-hardware combination as the agent. When a software-only autonomous agent operates in the virtual environment of an Artificial Life (ALife) system or the simulated reality of a game, it can be said to be *embedded* in that environment. If all of an embodied agents sensors are physically located on its body, or if a game or simulator generates the sensory input an embedded agent as if its sensors were similarly localized, the agent is said to be *egocentric* – its view of its world depends on its position within that world. Egocentricity is often associated with perceptual limitations: just as an agent embodied as a robot cannot see around corners, egocentric software agents embedded in a game or simulator are often provided with only limited information about the state of the system.

Games are an important class of applications for embedded agents, as described in the following section.

2.3 Artificial Intelligence in Games

There is a long history of using games as a platform for AI research. Claude Shannon wrote on the possible use of computers to play chess as early as 1950 (Shannon 1950), and by 1959, Samuel was actually applying machine learning to train a checkers player (Samuel 1959). Since then interest has been continuous and expansive, with notable milestones including the establishment of the ongoing *RoboCup* competitions, the high-profile chess match between Garry Kasparov and *Deep Blue* (Hamilton and Garber 1997), and more recently, the creation of two conferences dedicated to the study of AI in games.

A rationale for the use of interactive computer games in AI research has been given by

Laird and van Lent (2000). They list as motivations the richness of the simulated environments in modern computer games and the associated breadth of competence required for the associated AI, the relative cost effectiveness of using games rather than real-world hardware or custom-built research environments, the economic need for AI solutions in the game industry, and the opportunity to move academic AI out of the lab and into the hands of consumers. To these may be added the safety benefit of working with games rather than systems that put health or wealth at hazard, and – very importantly from the researcher's perspective – the easily scalable complexity of the AI challenge offered by games. Regarding the latter, it is useful to be able to simplify a system in order to clarify a problem when progress is not being made, and useful to extend a system in scale or complexity when progress *is* being made, rather than having to jump to a completely new problem.

In the history of game applications of AI, two concepts have emerged that underly the work presented in this dissertation: the idea of the *embedded game agent*, i.e. autonomous intelligent agents that "live in" the virtual environment provided by a game engine, and the idea of applying machine learning to the problem of creating an AI system for a game.

Important early examples of AI for embedded game agents are *ROG-O-MATIC* (Maudlin et al. 1984) and *Pengi* (Agre and Chapman 1987). *ROG-O-MATIC* was an expert system for controlling an explorer in the seminal *Rogue* dungeon-exploration game. Though *Rogue* agents are embedded, they are not egocentric; the game provides a "gods eye" view of the playing area. It had minimal learning capabilities, limited to filling out tables with the attributes of the monsters it encountered. In comparison to 15 expert players on multiple games, it ranked first in median game score and seventh in highest game score. The authors critiqued it for its "single-minded" inflexibility: it took a greedy approach when faced with multiple opportunities or multiple threats, which is often not a good approach to *Rogue* play.

Pengi was an agent for playing *Pengo*™ , an arcade style game where bees chase a penguin in an icehouse-like representation of Antarctica. Agre and Chapman (1987) presented their approach as a sort of rebellion against the sort of planning systems that were then in vogue in AI research. The game was too difficult for the planning methods available at the time; the authors cite complexity, uncertainty, and the need for real-time decision making. So they replaced planning based on the complete game state with immediate reactive responses to an egocentric view of the game. The AI still bound game objects to schema variables, but the schemas were all relative to the penguin agent rather than absolute. The 'theory' in their title is a hypothesis for cognitive science: "Before and beneath any activity of plan-following, life is a continual improvisation, a matter of deciding what to do *now* based on how the world is *now*." The result was a solution that played the game "badly, in near real time"; it is nevertheless a landmark in the history of game applications of AI. The authors' novel approach is still in vogue: the embedded agents described in this dissertation also have an egocentric view of the world, and build up their higher-level behavior from a sequence of decisions that are local in both space and time.

Much other work has been done with embedded game agents. Among arcade games there

have been treatments of *Pac-Man*™ (Gallagher and Ryan 2003) and *Ms. Pac-Man*™ (Lucas 2005). Among sports games much work has been done with the agents embedded in the *RoboCup* simulator (or embodied in actual robots, to similar effect) (Kitano et al. 1997). Additional work has been done with simulators of the "dueling robots" genre (Stanley et al. 2003; Lucas 2004) and strategy games (Agogino et al. 2000; Bryant and Miikkulainen 2003; Stanley et al. 2005a,c). Much of this work follows the paradigm established by Agre and Chapman (1987) for their *Pengi* agent, though there have been important innovations as well, such as the use of simulated sensors rather than schemas, and provision for memory in the agents' controllers.

Samuel's work started a tradition of applying machine learning to boardgames. Popular targets include tic-tac-toe (Gardner 1962), checkers (English 2001; Fogel 2001), chess (Kendall and Whitwell 2001; Fogel et al. 2005), backgammon (Tesauro and Sejnowski 1987; Tesauro 1994; Pollack et al. 1996), Othello (Moriarty and Miikkulainen 1995b; Chong et al. 2003), and Go (Richards et al. 1998; Lubberts and Miikkulainen 2001; Kendall et al. 2004; Stanley and Miikkulainen 2004b). A survey of applications of machine learning to games through the year 2000 can be found in Fürnkranz (2001).

More recently it has become common to apply machine learning to strategy games and other genres that feature embedded agents; much of the work cited for those genres (above) consisted of machine learning approaches to game AI. The work reported in this dissertation lies at that intersection: an application of machine learning to the problem of controlling agents embedded in a strategy game. The details are given in the following chapters.

It should be noted that the AI provide with commercial strategy games is almost entirely a matter of scripting; developers do not seem to think that the AI stemming from academic research is predictable enough for commercial use. They do occasionally use standard algorithms, such as A* for pathfinding. But even then they tend to rely on simplifications, e.g. what is displayed to the player is a continuous map is a cosmetic overlay over a set of predefined waypoints; AI-controlled agents calculate their routes over a simple graph rather than a complex map. The net effect of commercial style AI programming is that researchers disdain it as "not real AI" and many game players disdain it as "not very intelligent", if not an outright cheat. Therein lies both challenge and opportunity for academic AI researchers: can research-quality AI techniques scale up to the scope and complexity required for commercial strategy games, and produce behaviors in the embedded agents that players will reckon to be genuinely intelligent?

The remaining sections of this chapter introduce technologies that have been found useful in preliminary experiments with machine learning for embedded game agents, and which are used for the new experiments reported in this dissertation.

2.4 Artificial Neural Networks

The study of Artificial Neural Networks (ANNs) includes simulations of the behavior of biological neurons and networks, but more commonly presents itself as merely a biologically inspired model for computation. The most commonly encountered models for networks are aggregates of simple models of biological neurons called *perceptrons* (Rosenblatt 1958). An overview of the operation of biological neurons, and a general treatment of artificial neural networks and the methods used to train them, can be found in standard textbooks on the topic such as Haykin (1994).

Biological neurons are non-linear computing devices. They are stimulated on the input side by the release of neurotransmitters from other neurons, and they release neurotransmitters of their own on the output side. The flow of neurotransmitters is not continuous; a neuron's output is normally quiescent. Over time the neurotransmitters received at the neuron's inputs build up the stimulation of an accumulator. When that stimulation reaches a critical threshold the neuron "fires", discharging the accumulator and emitting a brief pulse of neurotransmitters at its own output. Thereafter its output returns to quiescence until sufficient additional stimulation is received at the inputs, and even in the case of continuous stimulation there is a refractory period that may delay the neuron's ability to fire again. The refractory period introduces the non-linearity into the neuron's behavior: when the rate of receipt of neurotransmitters at its inputs is increased by a factor of n, the rate of release of neurotransmitters at its output cannot always increase by the same factor.

The perceptron model does not operate with the "spiking" behavior of a real neuron. Instead, it uses scalar inputs and outputs that can be loosely interpreted as the average firing rate of the neuron of some period of time. As with a biological neuron, a perceptron can have more than one input, and the inputs do not all contribute equally to the activation of the neuron even when subjected to the same stimulus. This phenomenon is modelled in a perceptron by a vector of scalars that weight the influence of each input. In operation the stimulation at the perceptron's accumulator is calculated as the dot product of the input signals (treated as a vector) and the vector of weights.

Non-linearity is introduced into the perceptron's operation by applying a *squashing function* to the scalar value of that dot product. The term "squashing" is a metaphor that arises from a plot of the function; the most common squashing functions are approximately linear over some range of inputs, but limited by upper and lower bounds for inputs outside that range (figure 2.1). Various squashing functions are appropriate, but the logistic function is the most commonly used:

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{2.1}$$

The value calculated by applying the squashing function to the dot product of the input and weight vectors is the perceptron's scalar output.

A perceptron is usually provided with a *bias*, which modifies the firing threshold of the neuron it models. The bias is implemented as a scalar that is added to the dot product of the inputs and weights before applying the squashing function; its effect, viewed graphically, is to shift the

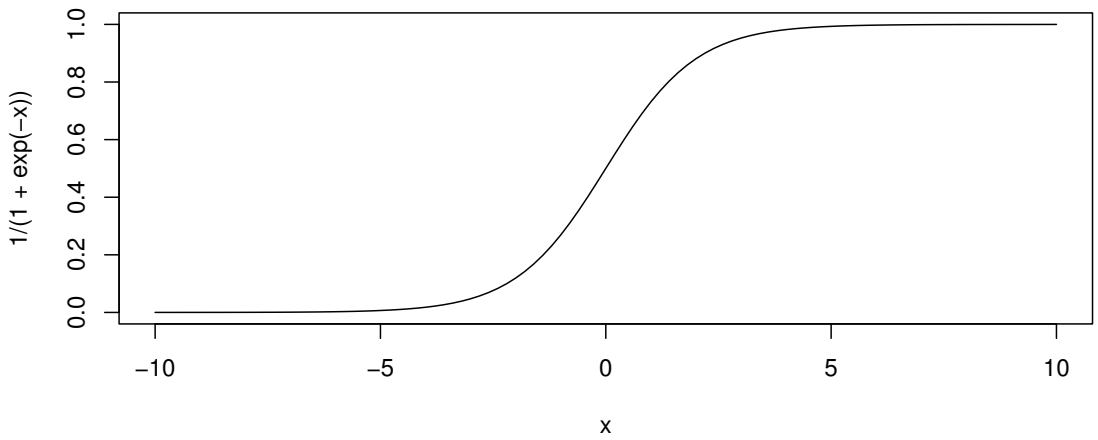


Figure 2.1: **The logistic function.** The logistic function (equation 2.1) is perhaps the most commonly used squashing function for artificial neural networks. It is approximately linear around $x = 0$, but is "squashed" so that higher and lower values of x are constrained to the range $(0, 1)$.

plot of the squashing left or right on the graph, changing the output associated with any given input activation. For pragmatic reasons the bias is usually implemented as a weight feeding an auxiliary input with a fixed value of 1 or -1 into the accumulator with the external inputs.

Despite the nonlinearity of its computations, a perceptron has very limited computational power. Its response is limited by a phenomenon called the *linear separability* of its inputs, and in fact it cannot compute the XOR of two binary inputs (Minsky and Papert 1988).

However, when perceptrons are organized into networks that feed the outputs of some into the inputs of others, the aggregate does not suffer the same limitations of computational power. It can be shown that a simple network of two "layers" of perceptrons, with the external inputs fed into all the perceptrons in the first layer and the outputs of those perceptrons used as the inputs for the perceptrons in the second layer, can compute any continuous multivariate function to within any desired accuracy (Cybenko 1989).

Such networks are called *feed-forward networks*, and their constituent perceptrons are called *neurons*. They can be arranged in any number of layers. The final layer of neurons, whose output activations give the value computed by the network, is called the *output layer*. All other layers are called *hidden layers*. By an entrenched misnomer, the receptors for the external inputs are often called the *input layer*, though there is not actually a layer of neurons there, and the first layer of neurons is called a hidden layer, even though it is directly exposed to the inputs (figure 2.2).

Feed-forward networks calculate a function of their inputs. A vector of input activations is used to calculate activations of the neurons in the first hidden layer, those activations are then used to calculate the activations of the next layer, and so on, until the signal has been *propagated* all the way through the output layer. The outputs of the neurons in that layer are a function, in the mathematical sense, of the network's inputs. If there is only a single neuron in the output layer then

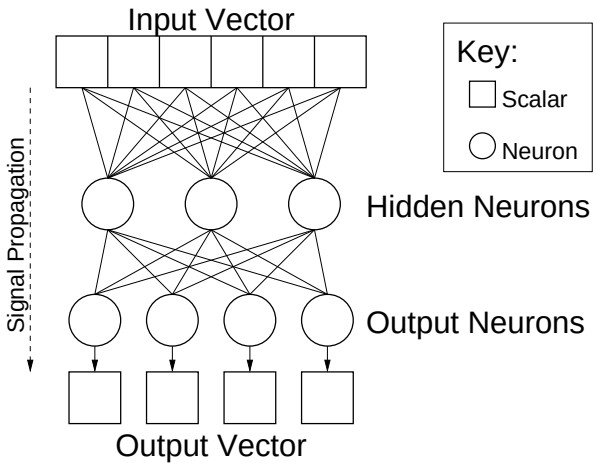


Figure 2.2: **Feed-forward neural network.** A feed-forward network calculates a function of an input vector by passing the input through a layer of neurons via weighted connections, and then passing the activation values of those neurons through another layer. The function calculated by the network depends on the weights on the connections.

the network calculates a scalar value; otherwise it calculates a vector value.

A feed-forward network can be used as the controller for some types of autonomous agents. If an agents sensory input can be represented as a vector of scalar values, and a network's output vector of scalar values can be interpreted as a control signal for the agent's activators, then an ANN can be used to map the agent's senses onto its choice of actions. The use of a feed-forward network results in a reactive agent. If a deliberative agent is required, a network can maintain internal state by means of recurrent connections (e.g., Elman 1990). Such state serves as a memory, and allows networks to be trained as controllers for non-Markovian control tasks (e.g., Gomez and Miikkulainen 1999).

Recurrency may also be useful for some tasks that do not obviously require memory, because it increases the computational power of a network. It has been shown that a recurrent network with rational numbers for its weights can be used to emulate a Universal Turing Machine, with the usual caveat regarding the need for infinite storage (Siegelman and Sontag 1991).

Artificial neural networks are frequently misunderstood to be learning devices. While it is true that they are almost always trained to some task rather than programmed by direct specification of their weights, and though there is interest in networks that actively learn from experience rather than being passively modified by a training algorithm (e.g., Stanley et al. 2003), they are properly understood to be computing devices. As a class they represent a paradigm for computation, just as do Turing machines, Von Neuman machines, and formal grammars.

However, training is an important issue when working with artificial neural networks. While it is a commonplace to assign a homework problem that requires specifying an ANN that computes

the XOR of two inputs, the specification of the weights needed for more complex tasks of pattern classification, motor control, or sequential decision tasks such as game play, is for all practical purposes impossible. Therefore a number of machine learning algorithms have been developed for training artificial networks to perform a designated task. One specific class of learning algorithms is reviewed in the next section.

Artificial neural networks are a popular implementation choice for agent controllers for several reasons: they are a universal computing architecture (Cybenko 1989; Siegelmann and Sontag 1994), they are fast, they can be trained to generalize well (Montana and Davis 1989), and they are tolerant of noise both during training (Bryant and Miikkulainen 2003; Gomez 2003) and during operation after having been trained (Gomez 2003).

2.5 Neuroevolution with Enforced Sub-Populations

For many agent control tasks the correct input-output mappings for the agents' controllers are not known, so it is not possible to program them or train them with supervised learning methods. However, controllers such as artificial neural networks can be evolved to perform a task in its actual context, discovering optimal mappings in the process. The use of a genetic algorithm to train an artificial neural network is called *neuroevolution*. Surveys of the field can be found in Schaffer et al. (1992) and Yao (1999).

A *genetic algorithm* (GA) is a biologically inspired reinforcement learning method that "breeds" a population of candidate solutions to find higher-quality solutions (Holland 1975). It works with a population of representations of solutions, often referred to metaphorically as DNA, genes, or chromosomes. Representations and their associated solutions are also distinguished by the biological terms *genotype* and *phenotype*. During evolution, representations are drawn from the population and instantiated into actual solutions, which are then evaluated by some criterion to obtain a scalar fitness score; the fitness score is used to determine whether the representation contributes to the creation of new "child" solutions (figure 2.3).

Most genetic algorithms operate in iterative *generations*. During a generation a fitness score is obtained for every representation, and then a new population is created from the current population, and the cycle is repeated. The details of implementation vary widely, but in general most or all of the population is replaced by new representations generated from the higher-scoring representations in the current population. Breeding mechanisms also vary widely, but most commonly they operate by applying biologically inspired operations such as crossover and mutations to a pair of solutions represented by vectors.

Neuroevolution is merely a special class of genetic algorithms. One distinguishing feature of this class is that the solution produced by the GA is not a static object such a solution to an instance of the Traveling Salesman Problem, but rather a software machine that is supposed to *do* something, such as classify patterns or control an agent. Therefore the fitness evaluations operate

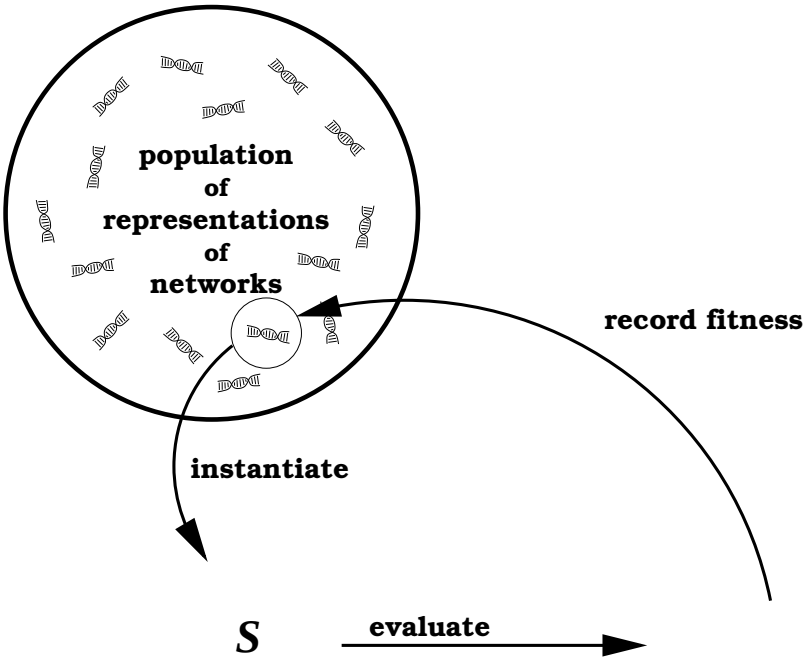


Figure 2.3: **Evolutionary life cycle.** Genetic algorithms operate by manipulating a population of representations of solutions to some target problem. The representations are evaluated on the basis of the quality of the solutions they specify. Those that score worst on that evaluation are usually discarded, and replaced by new representations bred form those that scored better.

by testing how well a candidate solution performs at that designated task. If the solution is the controller network for an autonomous intelligent agent, the fitness evaluation involves using the network to control an agent in its target environment through a number of sense-response cycles, and measuring how well the agent meets its goals in that environment (figure 2.4).

There are many different neuroevolutionary algorithms. Some only learn values for the weights for a predefined network structure, and others learn structures as well as weight values. The algorithm used for this dissertation, described below, learns weight values only.

When evolving weight values only, the simplest design for representations maps the network's weights onto a flat vector of floating point numbers via a canonical ordering. The design allows easy transduction between representations and networks (in both directions), and the vector representations can be directly subjected to breeding operations such as crossover. Mutations most commonly take the form of small deltas added to the existing value for a weight, but a number of other creative operations have been examined (Montana and Davis 1989).

One of the most empirically effective neuroevolutionary algorithms yet devised is *neuroevolution with enforced sub-populations* (NE-ESP, or usually just ESP) (Gomez and Miikkulainen 1999; Gomez 2003). The basic concept behind ESP is that a each representation specifies only

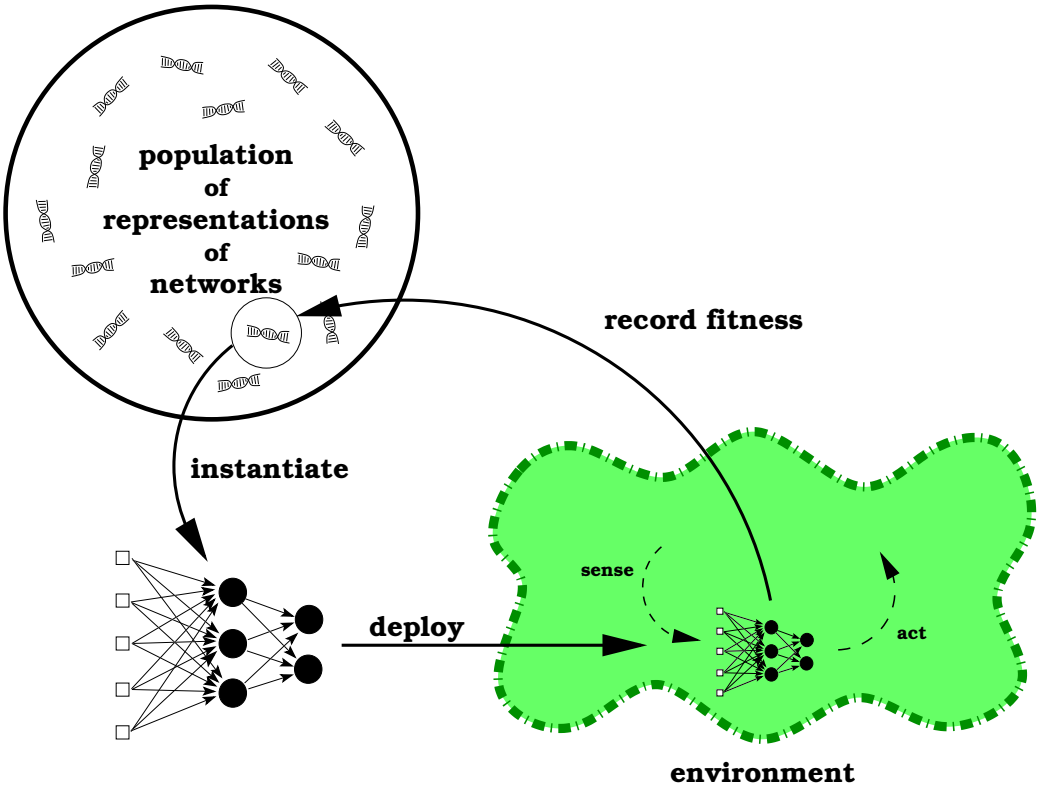


Figure 2.4: **Neuroevolutionary life cycle.** Neuroevolution is a special type of genetic algorithm that produces a trained artificial neural network as its output. That fact has implications for the way the candidate solutions must be evaluated. For example, if the network is to be used as the controller for an agent in some environment, the fitness evaluations are made by letting the candidate solutions actually control the agent in the target environment long enough to measure how well it works.

a single neuron rather than an entire network, and a separate breeding population is maintained for each neuron in the network.

Evaluations cannot be made on neurons in isolation, so the evaluations in ESP are done by drawing one neuron at random from each sub-population, assembling them into a complete network, evaluating the network as for any other neuroevolutionary algorithm, and ascribing that network's fitness score back to each of the individual neurons used to create it. When all the neurons in all the populations have been evaluated, selection and breeding is done independently within each sub-population. The fitness of an individual neuron depends not on its properties in isolation, but on how well it works together with neurons from the other populations. Thus the neurons in the sub-populations are subjected to cooperative coevolution (Potter and De Jong 1995; Moriarty 1997), and as evolution progresses they converge as symbiotic species into functional niches that work together in a network as a good solution to the target problem.

ESP has to date been used only for evolving the weight values for a network of pre-specified architecture. That is how it is used in all the experiments reported here, as described in section 4.2.2.

2.6 Human-Biased Genetic Algorithms

The design of every genetic algorithm incorporates some human biases, such as the choice of representation, the specification of the fitness function, and the parameterization of the randomization for the initial population of representations. If the solution to be discovered or optimized by the GA is a controller there are usually additional human-generated biases, such as specification of the controller's inputs and outputs. If the solution is an artificial neural network there is a biasing choice between the use of an algorithm that searches the weight space only vs. an algorithm that searches the structure space as well, and in the former case the specific choice of network topologies adds a further bias. Indeed, the basic decision to use a genetic algorithm will have some effect on the solution obtained, in comparison to the use of some other machine learning algorithm.

Thus some amount of human-induced bias is unavoidable when using genetic algorithms. However, it is useful to consider the class of GAs that deliberately invoke mechanisms for biasing the learning process with human knowledge or intuitions in order to help them find a solution, or steer them toward solutions with some specifically desired traits. We may refer to GAs that use such mechanisms as *human-biased genetic algorithms*.

Human-biased genetic algorithms can be classified according to where the human biases are inserted into the flow of the genetic algorithm. The resulting typology will be useful for clarifying thinking on the topic, and may reveal opportunities for useful new biasing mechanisms.

The typology offered here divides the biasing mechanisms into three classes: *population tampering*, *solution tampering*, and *fitness tampering* (figure 2.5). Population tampering involves inserting or removing representations (genotypes) into/from the population by mechanisms other than the conventional mechanisms necessary for all genetic algorithms. Solution tampering involves modifying a solution (phenotype) or its associated representation, and allowing the modification to replace or supplement the original in the ordinary flow of the evolutionary life cycle, so that a representation reflecting the tampering enters the gene pool. Fitness tampering involves mechanisms for ascribing a fitness to a solution other than what it would have obtained under ordinary fitness evaluations, to promote some types of solutions over others. Examples of each class are readily found, and are described in the following sections.

2.6.1 Population Tampering

A very direct method of population tampering is the Case-Injected Genetic Algorithm (CIGAR), which inserts the representations of solutions directly into the evolutionary population, possibly replacing low-fitness solutions already in the population (Louis and Miles 2005). A solution is considered to be a "case"; for human biasing the cases are human-generated solutions, from which

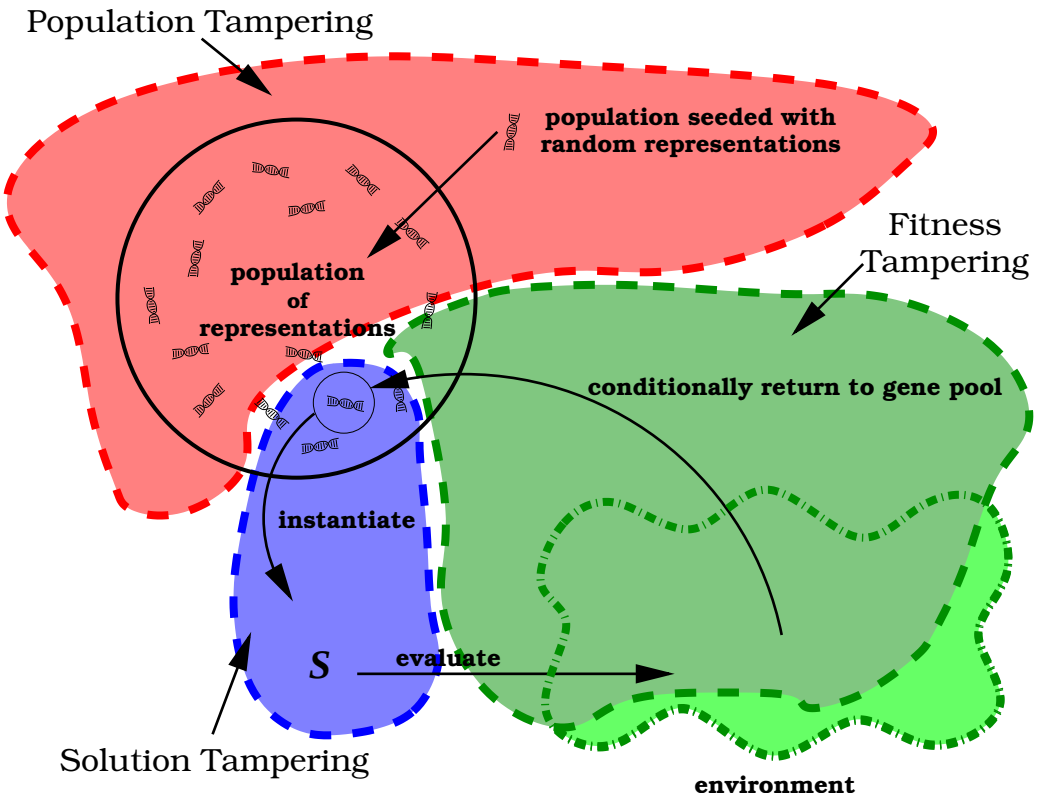


Figure 2.5: **Typology of human-biased genetic algorithms.** There are three basic classes of mechanisms for using human biases to guide genetic algorithms: population tampering, solution tampering, and fitness tampering.

representations are reverse engineered algorithmically for insertion into the population, or in principle a human could construct the representations directly. If the human-generated solutions are good, in the sense of being rated with high fitness by the GA, genetic material from their representations will spread throughout the population with high probability. Their use for human-generated biasing is necessarily limited to applications where solutions or their representations are simple enough to be constructed or discovered by humans. CIGARs have been used to create flight plans for simulated ground-strike operations; the cases provided by human design are resource allocation tables, from which the flight plans are derived automatically. The human bias allows the CIGAR to learn more efficient flight plans, and even to avoid plans that are likely to result in an ambush (Louis and Miles 2005).

Case injection is not strictly limited to the direct insertion of human-generated solutions into an evolutionary population. For example, Louis and Miles (2005) also used a side archive of good quality previously evolved solutions, from which some of the solutions were intermittently returned to the breeding population. The mechanism did not inject human bias, because the deci-

sions to archive and release solutions were made algorithmically. However, the same mechanism was invented independently by D'Silva et al. (2005) under the name *milestoning*, with the important difference that the decisions to archive or release a solution were made directly by a human rather than algorithmically. Thus a human observer could select evolved solutions with desirable properties for archiving – including solutions not easily constructed by humans – and return them to the breeding population later to re-introduce genetic material that had been bred out of the population, or to prevent it from being bred out. D'Silva et al. (2005) used the milestoning mechanism when evolving controllers for the *NERO* machine learning game, in order to ensure that desirable behaviors discovered by the *NERO* agents early in training were not lost when new behaviors were learned later.

The *NERO* game also provided another population-tampering mechanism called *convergence*, which upon demand replaces an entire evolutionary breeding population with clones of an evolutionary solution selected by a human player. Thus when the *NERO* agents have evolved to the point where some are starting to show desirable behavior, the player can select the agent showing the most desirable behavior and invoke convergence to immediately weed all other behaviors out of the population. This convergence operation is an as-yet unpublished feature of the *NERO* game (Stanley et al. 2005a; Gold 2005; Stanley et al. 2005c).

A final, obvious way of biasing evolutionary search is to start with a human-selected evolutionary population rather than a randomly generated one. The population can be directly constructed by humans, in which case this mechanism can be seen as mass case-injection, or it can be selected by humans from the output of other learning algorithms, in which case the GA re-trains or re-optimizes the solutions from the initial state established by the humans (e.g., Schultz and Grefenstette 1990, 1992; Ramsey and Grefenstette 1993).

2.6.2 Solution Tampering

Human decisions can also bias the result of a genetic algorithm by modifying individual solutions rather than moving solutions into and out of the evolutionary population. This mechanism is called solution tampering.

Lamarckian evolution (Lamarck 1809) is no longer thought to be an important mechanism in biology (Wilkins 2006), but can easily be made to work in simulated systems (e.g, Grefenstette 1991; Schultz and Grefenstette 1992). The mechanism allows adaptations in a solution (phenotype) to be reverse engineered back into its representation (genotype), which is returned to the breeding population with the fitness rating of the adapted solution. (The difficulty in biology is the lack of a general mechanism for reverse engineering the adaptations back into an organism's DNA.) Lamarckian evolution allows the insertion of human bias into a genetic algorithm by allowing human influence on the adaptation process. For example, in chapter 6 below, Lamarckian neuroevolution is used to incorporate the human bias by means of backpropagation on human-generated training examples for the mechanism of phenotypic adaptation.

Another mechanism for solution tampering is known as *advice-taking*, or simply *advice* (Maclin and Shavlik 1996; Yong et al. 2005). Advice mechanisms construct partial solutions on the basis of human-specified rules, integrate the partial solutions into one or more of the complete but inferior solutions in the evolutionary population, and allow the modified solution(s) and their associated representations to continue through the normal flow of the genetic algorithm. For example, the advice system of Yong et al. (2005), derived from the earlier RATLE system (Maclin and Shavlik 1996), allows human players to specify simple rules of behavior for the agents in the *NERO* game, such as "if there is a wall close in front of you, turn 90° to the right". The rule is converted via KBANN (Towell and Shavlik 1994) into an artificial neural network that maps the relevant sensory inputs onto the relevant motor outputs with the specified logic, and that network is integrated with an existing solution from the evolutionary population as an appliqué. The mechanism allows the *NERO* agents to learn the easily specified parts of their task quickly and in accord with human notions of how the task should be managed, and if the advice is apropos the modification rapidly spreads throughout the population. Notably, the system allows humans to inject expert knowledge directly into genetic algorithms when a complete solution would be too difficult to specify.

2.6.3 Fitness Tampering

Fitness tampering methods are the third class of mechanisms for injecting human knowledge or preferences into the evolutionary life cycle. It involves manipulating the relationship between a solution and the fitness that is ascribed to it.

Baldwinian evolution (Baldwin 1896; Hinton and Nowlan 1987) is a mechanism somewhat like Lamarckian evolution in that it involves adaptation in an organism (phenotype); the difference is that the adaptations are not reverse engineered back into the organism's DNA (genotype). Instead, the adaptation – usually conceived of as learning – gives the organism a fitness advantage in its own lifetime, increasing the probability that it will feed its genetic heritage back into the gene pool. The evolutionary guidance it provides comes about because the adaptations in the phenotype leverage partial adaptations in the genotype, increasing whatever fitness advantage they might offer. The long-term effect is to reinforce those partial genotypic adaptations that can be augmented by phenotypic adaptations, ultimately channeling evolution in the direction of the reinforcement.

For the purposes of genetic algorithms, Baldwinian evolution is like the Lamarckain evolution described above, except that the adaptations a solution (phenotype) undergoes are not reverse engineered back into its representation (genotype). The solution is instantiated from its representation normally, and trained (adapted) as in a Lamarckian genetic algorithm, but after the fitness evaluation the modified solution is discarded and the unmodified representation is fed back into the gene pool – albeit with the fitness score obtained by the modified solution. Thus the net effect is alter the actual fitness of the solution; it is therefore a form of fitness tampering rather than solution tampering.

The most common form of fitness tampering – probably the most common form of human

biasing in any of the tampering classes – is *shaping*. Shaping refers to training procedures that guide the trainee toward some target behavior through a series of successive approximations. The term and concept were introduced into the scientific literature by Skinner (1938), but the concept has long been used in animal training – presumably since prehistoric times. The notion behind shaping is to find solutions that are "close" to the desired solution in some conception of the solution space, and then refine training to find new solutions that are closer yet. A typology of approaches to shaping has been offered by Peterson et al. (2001), which can be stated in slightly modified form as: modify the task, modify the environment, or modify the reward structure.[1]

Modifying the task usually involves learning a target behavior by stepwise approximation, e.g. by modifying the relative lengths of the poles on the dual pole-balancing problem, which is easy when the lengths are different but becomes progressively more difficult as the lengths become more similar. Task shaping allows the difficult task to be learned in a series of small steps starting from the easy task (Gomez and Miikkulainen 1997). Another example of modifying the task is given by Perkins and Hayes (1996), who suggest that a complex task such as getting a robot to dock at a battery charger is most easily learned by first training the robot to succeed from an easy starting position and orientation and then iteratively retraining it from progressively more difficult starting positions. Modification of the task is also an integral part of the *NERO* machine learning game (Stanley et al. 2005c). The target task for the *NERO* agents is not even specified beyond the implicit, abstract task of "learn effective behavior"; the player must define the tasks to be learned in more detail, and shaping by task modification is explicitly recommended as a strategy of play.

Perkins and Hayes (1996) observe that modifying the task is the method that most closely resembles the use of the term *shaping* in psychology. They also interpret modification of the task as a matter of adapting previously learned policies. Taken broadly, that interpretation would include all instances of re-training, re-tuning, and behavior transfer as instances of shaping. However, the term is usually limited to situations where the intermediate steps are deliberately designed to lead learners to a difficult solution not easily reached otherwise.[2]

Modifying the environment allows discovery of solutions that work in a simple or helpful context, after which more precise or sophisticated solutions can be obtained by further learning in a less obliging environment. One example is changing the height of the mountain in the mountain car task (Randløv 2000); another is the *NERO* training environment, where enemies and obstacles are added or relocated as training progresses, in order to help the agents acquire the desired tactical skills (Stanley et al. 2005c) (figure 2.6).

[1] Peterson et al. (2001) offer "change the physics of the task" rather than "modify the environment", which is offered here as an umbrella case. Their narrower formulation leaves gaps in the typology, and still does not solve the problematic nature of the typology mentioned in the text.

[2] But see Dorigo and Colombetti (1993) for an exception to that convention. Their method is very far removed from the original notion described by the term "shaping": they build up behaviors by programming or training modules and then combining them, a mechanism similar to layered learning (Stone and Veloso 2000b), basis behaviors (Matarić 1995), and mixture-of-expert models (Jacobs et al. 1991; Jordan and Jacobs 1992) – none of which are labeled as "shaping" by their own authors.

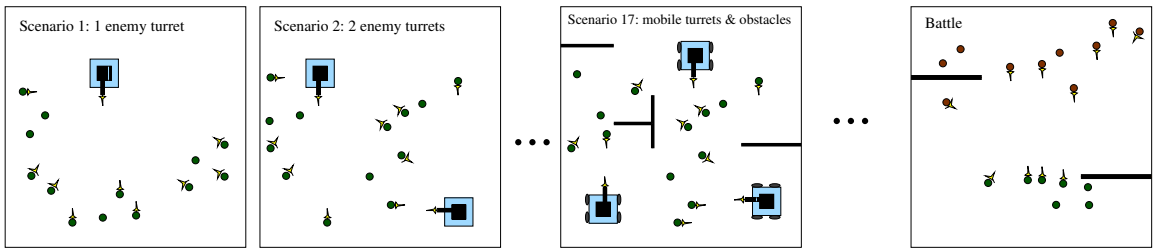


Figure 2.6: **Environment shaping in the *NERO* game.** The recommended method of training the simulated combat robots in the *NERO* machine learning game is to progressively enrich the environment, in order to encourage the acquisition of ever more sophisticated behavior.

Modifying the reward structure is the third class of shaping mechanisms. As with modifying the task or environment, the goal is to guide discovery of difficult solutions via successive approximations. In this case, solutions that are relatively far from the desired solution can be rewarded out of proportion to their success until discovered, and then the reward can be adjusted to require a more precise approximation to the desired solution. For genetic algorithms the reward structure is the fitness function. The *NERO* game offers a notable example of modifying the fitness function, which is manipulated very explicitly and in real time while observing the agents at their learning task.

Interestingly, the *NERO* game utilizes all three classes of shaping: the task is modified as the agents are sequentially trained toward increasingly sophisticated behaviors, the environment is modified as objects are added to or moved around in the sandbox to encourage flexibility in the learners, and the reward structure is modified by direct manipulation of the coefficients of the terms in the fitness function via the "sliders" on the control panel (Stanley et al. 2005a,c).

The typology for shaping mechanisms is slightly problematic. For example, when Randløv (2000) changes the height of the mountain for the mountain car task, is that a modification of the task or a modification of the environment? The distinction seems to be more semantic than typological. But in other cases the two categories are clearly distinct: in the *NERO* game the player explicitly modifies both the target task and the environment where the task is learned, independently, as play progresses.

A final important mechanism for fitness tampering is the interactive genetic algorithm, wherein a human directly selects which candidate solutions should be bred to produce the next generation of the evolutionary population. This mechanism is most commonly used for digital art and music, where unquantifiable aesthetic considerations are paramount (e.g., Rowbottom 1998; Biles 1994). The *GenJam* system for producing jazz solos has been successful enough that it is still being developed and used for public performances, over a decade after its introduction (Biles 2006).

2.6.4 Summary

This section has offered a high-level typology for mechanisms of inserting human biases into genetic algorithms, and offers examples of each. Examination of the typology may reveal opportunities for new mechanisms that can be exploited to good effect.

It should be noted that the three classes defined by the typology – and indeed most of the specific mechanisms surveyed – are independent in their operation. Combinations are possible; for example, the basic design of the *NERO* game is built around shaping (fitness tampering), but Yong et al. (2005) adds advice-taking (solution tampering) to the system, without modifying the use of shaping inherent in the design. It may be advantageous to explore other combinations of mechanisms for incorporating human biases into genetic algorithms.

Chapter 3

The *Legion II* Game

Legion II is a discrete-state strategy game designed as a test bed for studying the behavior of embedded game agents. It requires a group of legions to defend a province of the Roman Empire against the pillage of a steady influx of barbarians. The legions are the agents under study; they are trained by the various machine learning methods described in chapters 4-8. The barbarians act according to a preprogrammed policy, to serve as a foil for the legions.

The features and rules of the game are described in the next section. The sensors and controllers for the barbarians are described in section 3.2, and the sensors and controllers for the legions are described in section 3.3. The methods used to train the legions and the learning experiments done in the *Legion II* environment are explained in chapters 4-8.

3.1 Game Objects and Rules of Play

Legion II provides challenges similar to those provided by other games and simulators currently used for multi-agent learning research, and is expandable to provide more complex learning challenges as research progresses. In its current incarnation it is incrementally more complex than a multi-predator / multi-prey game. It is conceptually similar to the pseudo-*Warcraft*™ simulator used by Agogino et al. (2000), differing primarily in its focus on the predators rather than the prey, and consequently in the details of scoring games. The correspondences between *Legion II* , pseudo-*Warcraft*, and a multi-predator/multi-prey system are shown in table 3.1.

3.1.1 The Map

Legion II is played on a planar map. The map is tiled with hexagonal cells in order to quantize the location and movement of objects in the game; in gaming jargon such cells are called *hexes* (singular *hex*). For the experiments reported here the map itself is a large hexagonal area, with 21 cells along its major diagonals, giving a total of 331 cells in the playing area. Several randomly selected map

Legion II	$\sim$	pseudo-*Warcraft*	$\sim$	predator-prey
legion	$\equiv$	guard	$\equiv$	predator
barbarian	$\equiv$	peon	$\equiv$	prey
city	$\approx$	mine		N/A

Table 3.1: **Approximate correspondence between experimental domains.** The *Legion II* game is conceptually similar to domains that have been used in various other studies of multi-agent learning, including the pseudo-*Warcraft* game of Agogino et al. (2000) and multi-agent instances of the predator-prey domain.

cells are distinguished as *cities*, and the remainder are considered to be *farmland* (figure 3.1).

The hex tiling imposes a six-fold radial symmetry on the map grid, defining the cardinal directions NE, E, SE, SW, W, and NW. This six-fold symmetry, along with the quantization of location imposed by the tiling, means that an agent's atomic moves are restricted to a discrete choice of jumps to one of the six cells adjacent to the agent's current location, and that the atomic move is always in one of the six cardinal directions. The motor control laws that play an important role in many simulators are here abstracted away into that discrete choice between the discrete options. This map structure has important consequences for the design of the sensors and controllers for the agents in the game, which are described in detail below.

3.1.2 Units

There are two types of autonomous agents that can be placed on the map in a *Legion II* game: *legions* and *barbarians*. In accordance with gaming jargon these mobile agents are called *units* (singular *unit*) when no distinction needs to be made between the types.

Each unit is considered to be "in" some specific map cell at any time. A unit may move according to the rules described below, but its moves occur as an atomic jump from the cell it currently occupies to an adjacent one, not as continuous movement in Euclidean space.

Unless the graphic display has been disabled to speed up training or testing, the current position of each unit is shown by a sprite on the game map. In accordance with the jargon of the *Civilization* game genre, the sizes of the units are ambiguous. Thus the unit type called "a legion" represents a body of legionnaires, but is shown graphically as only a pair of men behind large rectangular shields. Similarly, the unit type called "a barbarian" represents a body of barbarians operating as a warband, but is shown graphically as only a single individual with axe and shield (figure 3.1).

The legions start the game already on the map, in randomly selected map cells. There are no barbarians in play at the start of the game. Instead, barbarians enter at the rate of one per turn, an a randomly selected unoccupied map cell.

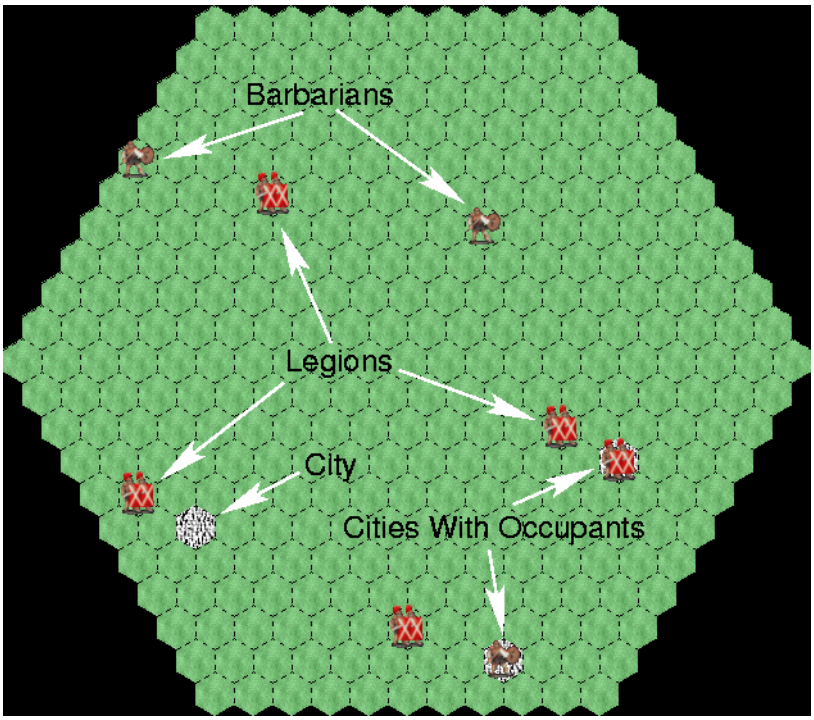


Figure 3.1: **The *Legion II* game.** A large hexagonal playing area is tiled with smaller hexagons in order to quantize the positions of the game objects. Legions are shown iconically as close pairs of men ranked behind large rectangular shields, and barbarians as individuals bearing an axe and a smaller round shield. Each icon represents a large body of men, i.e. a legion or a warband. Cities are shown in white, with any occupant superimposed. All non-city map cells are farmland, shown with a mottled pattern. The game is a test bed for multi-agent learning methods, whereby the legions must learn to contest possession of the playing area with the barbarians. (Animations of the *Legion II* game can be viewed at `http://nn.cs.utexas.edu/keyword?ATA`.)

3.1.3 Game Play

Legion II is played in turns. At the beginning of each turn a new barbarian is placed at a random location on the map. If the randomly generated location is already occupied by a unit, a new location is generated. The search continues until a free location for the barbarian is found.

Thereafter, each legion is allowed to make a move, and then each barbarian in play makes a move. A unit's move can be an atomic jump to one of the six adjacent map cells, or it can elect to remain stationary for the current turn. When all the units have made their moves the turn is complete, and a new turn begins with the placement of another new barbarian. Play continues for a pre-specified number of turns, 200 in all the experiments reported here.

All the units are autonomous; there is no virtual player that manipulates them as passive objects. Whenever it is a unit's turn to move, the game engine calculates the activation values for

that unit's egocentric sensors, presents them to the unit's controller, and implements the choice of move signaled at the output of the unit's controller. The sensors and controllers are described in full detail below.

There are some restrictions on whether the move requested by a unit is actually allowed, and the restrictions vary slightly between the legions and the barbarians. The general restrictions are that only one unit may occupy any given map cell at a time, and no unit may ever move off the edge of the playing area defined by the tiling.

If a legion requests moving into a map cell occupied by another legion, or requests a move off the edge of the map, the game engine leaves the legion stationary for that turn instead. If the legion requests moving into a map cell occupied by a barbarian, the game engine immediately removes the barbarian from play and then moves the legion as requested. If the legion requests a move into an unoccupied map cell, or requests to remain stationary for the current turn, the request is immediately implemented by the game engine.

If a barbarian requests a move into a map cell occupied by either a legion or another barbarian, the game engine leaves it stationary for the current turn. (Notice that that does not allow a barbarian to eliminate a legion from play the way a legion can eliminate a barbarian.) If the barbarian requests a move off the edge of the map, the game engine consults the barbarian's controller to see what its second choice would have been. If that second choice is also off-map then the game engine leaves the barbarian stationary for the current turn; otherwise, the secondary preference is implemented. If the barbarian requests a move that is neither off-map nor into an occupied map cell, or requests to remain stationary, the request is immediately implemented by the game engine.

Barbarians are given the second choice when they request a move off the map, because their programming is very simple (as described below), and it is not desirable to leave them 'stuck' at the edge of the map during a game. Legions do not get the second-chance benefit; they are expected to learn to request useful moves.

3.1.4 Scoring the Game

The game score is computed as follows. The barbarians accumulate points for any pillaging they are able to do, and the legions excel by minimizing the amount of points that the barbarians accumulate. At the end of every game turn, each barbarian in play receives 100 points for pillage if it is in a city, or only a single point otherwise. The points are totaled for all the barbarians each turn, and accumulated over the course of the game. When a barbarian is eliminated no points are forfeited, but that barbarian cannot contribute any further points to the total thereafter.

This scoring scheme was designed in order to meet one of the two major goals for this research, namely forcing the legions to learn two distinct classes of behavior in order to minimize the barbarian's score, as described in chapter 5 below. Due to the expensive point value for the cities, the legions must keep the barbarians out of them, which they can easily do by garrisoning them. However, further optimization requires the legions assigned to garrison duty to actively pursue

and destroy the barbarians in the countryside. If they fail to do so, a large number of barbarians will accumulate in the countryside, and though each only scores one point of pillage per turn, their cumulative aggregate is very damaging to the legions' goal of minimizing the barbarian's score.

In principle the legions might be able to minimize the pillage by neglecting to garrison the cities and utilizing every legion to try to chase down the barbarians, but the random placement of the incoming barbarians means that they can appear behind the legions, near any ungarrisoned cities, and inflict several turns of the very expensive city pillaging before a legion arrives to clear them out. The barbarian arrival rate was by design set high enough to ensure that the legions cannot mop them up fast enough to risk leaving any cities ungarrisoned. Thus the legions *must* garrison the cities in order to score well, and any improvement beyond what can be obtained by garrisoning the cities can only come at the cost of learning a second mode of behavior, pursuing the barbarians.

For the purposes of reporting game scores the pillage points collected by the barbarians are normalized to a scale of $[0, 100]$, by calculating the maximum possible points and scaling the actual points down according to the formula:

$$Score_{\text{reported}} = 100 \times Points_{\text{actual}}/Points_{\text{possible}} \tag{3.1}$$

The result can be interpreted as a pillage rate, stated as a percentage of the expected amount of pillaging that would have been done in the absence of any legions to contest the barbarians' activities. Notice that according to this metric, *lower* scores are better, from the legion's point of view.

In practice the legions are never able to drive the score to zero. This fact is due in part to the vagaries of the starting conditions: if the random set-up places all the legions very distant from a city and a barbarian is placed very near that city on the first turn, there is nothing the legions can do to beat that barbarian into the city, no matter how well trained they are. However, a factor that weighs in more heavily than that is the rapid rate of appearance of the barbarians vs. the finite speed of the legions. Since the legions and barbarians move at the same speed, it is difficult for the legions to chase down the barbarians that appear at arbitrary distances away. Moreover, as the legions thin out the barbarians on the map the average distance between the remaining barbarians increases, and it takes the legions longer to chase any additional barbarians down. Thus even for well-trained legions the game settles down into a dynamic equilibrium between the rate of new barbarian arrivals and the speed of the legions, yielding a steady-state density of barbarians on the map, and thus a steady-state accumulation of pillage counts after the equilibrium is achieved.

3.2 Barbarian Sensors and Controllers

The legions are the only learning agents in the game, so the barbarians can use any simple pre-programmed logic that poses a suitable threat to the legions' interests. The barbarians' basic design calls for them to be attracted toward cities and repulsed from legions, with the attraction slightly stronger than the repulsion, so that the barbarians will take some risks when an opportunity for

pillaging a city presents itself. This behavior is implemented by algebraically combining two "motivation" vectors, one for the attraction and one for the repulsion:

$$\mathcal{M}_{\text{final}} = \mathcal{M}_{\text{cities}} + 0.9\mathcal{M}_{\text{legions}} \tag{3.2}$$

Each vector consists of six floating point numbers, indicating the strength of the "desire" to move in each of the six cardinal directions. The 0.9 factor is what makes the motivation to flee the legions slightly weaker than the motivation to approach the cities. After the combination, the peak value in $\mathcal{M}_{\text{final}}$ indicates which direction the barbarian most "wants" to move. In situations where a second choice must be considered, the second-highest value in $\mathcal{M}_{\text{final}}$ is used to select the direction instead.

The values in the two arrays are derived, directly or indirectly, from the activation values in a simple sensor system. The barbarian's sensor system consists of two sensor arrays, one that detects cities and another that detects legions. Each array divides the world into six 60° non-overlapping egocentric fields of view. The value sensed for each field is:

$$s = \sum_i \frac{1}{d_i} \quad , \tag{3.3}$$

where d_i is the distance to an object i of the correct type within that field of view. The distances are measured in the hex-field equivalent of Manhattan distance, i.e. the length of the shortest path of map cells from the viewer to the object, not counting the cell that the viewer itself is in (figure 3.2).

Due to the relatively small map, no limit is placed on the range of the sensors. If an object is exactly on the boundary between two fields of view, the sensors report it as being in the field clockwise from the boundary, from the perspective of the observing agent.

Notice that this sensor architecture obscures a great deal of detail about the environment. It does not give specific object counts, distances, or directions, but rather only a general indication of how much opportunity or threat the relevant class of objects presents in each of the six fields of view.

Once these values have been calculated and loaded into the sensor arrays, the activations in the array that senses cities can be used directly for the $\mathcal{M}_{\text{cities}}$ vector in equation 3.2. $\mathcal{M}_{\text{legions}}$ can be derived from the values in the array that senses legions by permuting the values in the array to reverse their directional senses, i.e. the sensor activation for legions to the west can be used as the motivation value for a move to the east, and similarly for the other five cardinal directions. After the conversions from sensor activations to motivation vectors, $\mathcal{M}_{\text{final}}$ can be calculated and its peak value identified to determine the requested direction for the current move.

There is no explicit mechanism to allow a barbarian to request remaining stationary for the current turn. For simplicity the game engine examines a barbarian's location at the start of its move and leaves the barbarian stationary if it is already in a city. Otherwise the game engine calculates the values to be loaded into the barbarian's sensors, performs the numerical manipulations described above, and implements the resulting move request if it is not prohibited by the rules described in section 3.1.3.

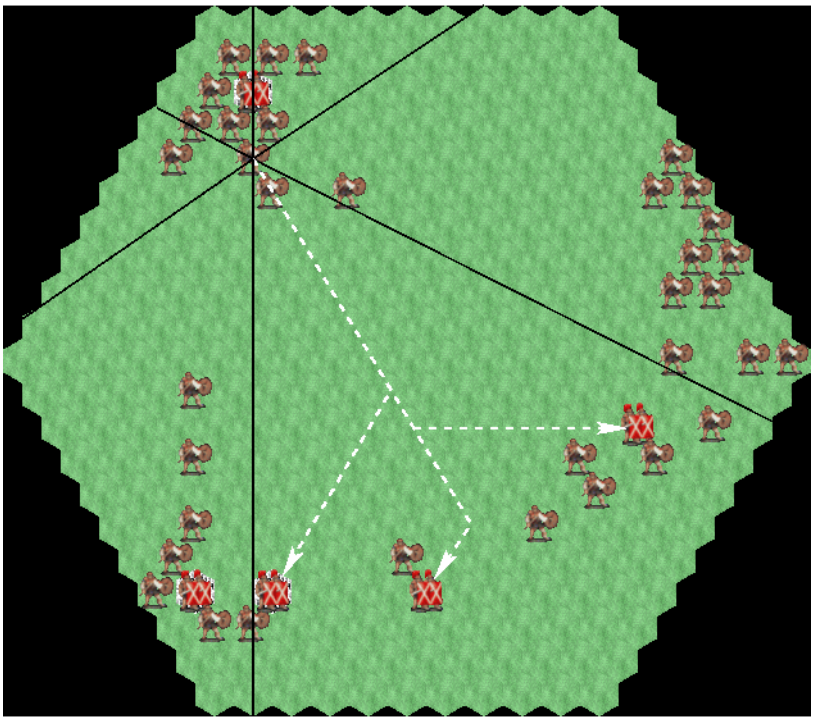


Figure 3.2: **The barbarian's sensor geometry.** The solid black lines show the boundaries of the six sensory fields of view for one barbarian near the northwest corner of the map. The boundaries emanate from the center of the map cell occupied by the barbarian and out through its six corners. For simplicity, an object in a cell split by a boundary is reckoned to be in the field to the clockwise of that boundary. For example, the city straight to the north of the sensing barbarian is reported by the barbarian's NE sensor, and ignored by its NW sensor. The dashed white lines show the hexagonal Manhattan distances to the three legions in the SE field of view of the sensing barbarian. These lines are traced from the center to center along a path of map cells, and thus emanate through the sides of the hexagons rather than through the corners as the field boundaries do.

The resulting behavior, although simple, has the desired effect in the game. As suggested by figure 3.2, barbarians will stream toward the cities to occupy them, or congregate around them if the city is already occupied. Other barbarians will flee any roving legions, sometimes congregating in clusters on the periphery of the map. The barbarians are quick to exploit any city that the legions leave unguarded. They do, however, tend to get in each other's way when a legion approaches a crowd and they need to flee, resulting in many casualties, but that is perhaps an appropriate simulation of the behavior of undisciplined barbarians on a pillaging raid.

3.3 Legion Sensors and Controllers

Unlike the barbarians, the legions are required to learn an appropriate set of behaviors for their gameplay. They are therefore provided with a more sophisticated, trainable control system. The design includes a sensor system that provides more detail about the game state than the barbarians' sensors do, plus an artificial neural network "brain" to map the sensor inputs onto a choice of actions.

3.3.1 The Legions' Sensors

The legions are equipped with sensor systems that are conceptually similar to the barbarians', but enhanced in several ways. Unlike the barbarians, the legions have a sensor array for each type of object in the game: cities, barbarians, and other legions. Also unlike the barbarians, each of those sensor arrays are compound arrays consisting of two six-element sub-arrays plus one additional element, rather than of only six elements total as for the barbarians.

An array's two six-element sub-arrays are similar to the barbarians' sensor arrays, except that one only detects objects in adjacent map cells and the other only detects objects at greater distances. For the first sub-array the game engine sets the array elements to 1.0 if there is an object of the appropriate type in the adjacent map cell in the appropriate direction, and to 0.0 otherwise.

For the second sub-array the game engine assigns values to the elements slightly differently from the way it assigns values to the barbarian's sensors. First, it ignores objects at $d = 1$, since those are detected by the short-range array described above. Second, since the distances used in the calculations for this array are as a result always greater than one, it deducts one from the distances used in the calculations, in order to increase the signal strength. That is, equation 3.3 used for the barbarians becomes:

$$s = \sum_i \frac{1}{d_i - 1} \quad , \tag{3.4}$$

for objects i of the correct type *and* at a distance greater than one, within that field of view.

A third difference is that sensed objects near a boundary between two sensor fields are not arbitrarily assigned to the field to the clockwise of the boundary. Instead, objects within 10° of the boundary have their signal split between the two fields. The effective result is a set of 40° arcs of unsplit signals alternating with 20° arcs of split signals, though the aggregations result in an array of only six activation values. As with the barbarians' sensors, there is no range limit on this long-range sub-array.

The additional independent sensor element in a compound array detects objects in the legion's own map cell: if an object of the appropriate type is present the game engine sets the value of this sensor to 1.0; otherwise it sets it to 0.0. However, a legion does not detect itself, and since the rules prevent multiple units from occupying the same map cell, the only time this detect-local sensor is activated in practice is when the legion occupies a city. In principle the detect-local sensor could have been eliminated from the sensor arrays used to detect legions and barbarians, but

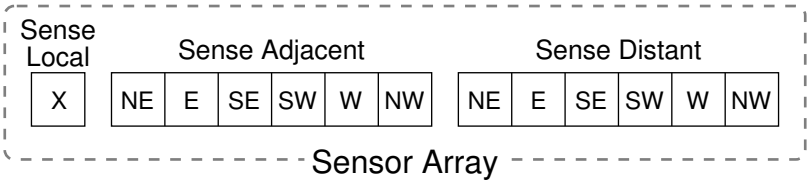


Figure 3.3: **The legions' compound sensor architecture.** The legions have three sensor arrays, one each for cities, barbarians, and other legions. Each of those three arrays consists of three sub-arrays as shown above. A single-element sub-array (left) detects objects colocated in the map cell that the legion occupies. Two six-element sub-arrays detect objects in the six radial fields of view; one only detects adjacent objects, and the other only detects objects farther away. These 13 elements of each of the three compound arrays are concatenated to serve as a 39-element input activation for an artificial neural network that controls the legion's behavior (figure 3.4).

identical arrays were used for all object types in order to simplify the implementation, and to make allowance for future game modifications that would allow "stacking" multiple units within a single map cell.

The architecture of the compound sensors is shown in figure 3.3. The two sub-arrays contain six elements each, corresponding to the six cardinal directions. Thus together with the additional independent element, each array reports 13 floating point values ≥ 0.0 whenever a sense is collected from the environment. Since there is one compound sensor for each of the three types of game object, a legion's perception of the game state is represented by 39 floating point numbers.

3.3.2 The Legions' Controller Network

A legion's behavior is controlled by a feed-forward neural network. The network maps the legion's egocentric perception of the game state onto a choice of moves. Whenever it is a legion's turn to move, the game engine calculates the sensor values for the legion's view of the current game state and presents the resulting 39 floating point numbers to the input of the controller network. The values are propagated through the network, and the activation pattern at the network's output is decoded to determine the legion's choice of move for the current turn (figure 3.4).

The output layer of the networks consist of seven neurons, corresponding to the seven discrete actions available to the legions. When the input activations have been propagated through the network the activation pattern in the output layer is interpreted as an *action unit coding*, i.e. the action corresponding to the output neuron with the highest activation level is taken to be the network's choice of action for the current turn.[1]

For all the experiments reported here, the controller network's hidden layer consisted of 10 neurons. That size was determined experimentally, and found to work well with all the training methods studied.

[1] A variant of this decoding scheme is introduced in chapter 8.

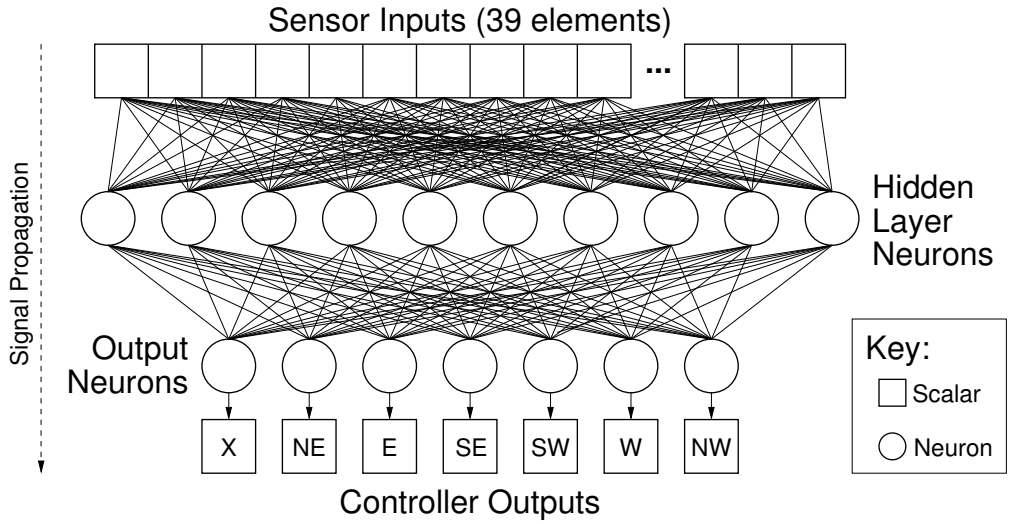


Figure 3.4: **The legions' controller networks.** During play the values obtained by a legion's sensors are propagated through an artificial neural network to create an activation pattern at the network's output. This pattern is then interpreted as a choice of one of the discrete actions available to the legion. When properly trained, the network serves as the controller for the legion as an autonomous intelligent agent.

In addition to the input signal, each neuron in the controller networks is fed by a bias unit with a fixed activation of $+1.0$ and a trainable weight to propagate the value into the neuron's accumulator. Recent authors commonly use -1.0, but the only difference is the sign that must be learned for the afferent weights, and using the positive number allows a minor optimization of the implementation.

3.3.3 Properties of the Legions' Control Architecture

There are a number of important consequences of the adoption of this sensor/controller architecture for the legions, which the reader may wish to keep in mind while reading about the methodologies and experiments:

- *The sensor readings are egocentric.* For a given state of the game, each of the legions in play will perceive the map differently, depending on their individual locations on the map.
- *The sensors provide a lossy view of the map.* The legions have a great deal of information about their immediate neighborhood, but that is reduced to a fuzzy "feel" for the presence of more distant objects.
- *The legions must work with uninterpreted inputs.* There is a semantic structure to the sensor arrays, but that structure is not known to the legions: the sense values appear as a flat vector of floating point numbers in their controller networks' input layers. The significance of any individual input or set of inputs, or of any correlations between inputs, is something the legions must obtain via the learning process.

- *There is no explicit representation of goals.* None of the network inputs, nor any other part of the controller logic, provide a legion with any sort of objective. Coherent higher-level behavior must be learned as a response to a sequence of inputs that vary over time.
- *The legions do not have memory.* The feed-forward controller networks do not allow for any saved state, so the legions are not able to learn an internal representation for goals to make up for the lack of externally specified goals. All actions are immediate reactive responses to the environment.

These various requirements and restrictions conspire to present the legions with a very difficult challenge if they are to learn to behave intelligently. However, experience shows that properly trained artificial neural networks excel at producing the appearance of purposeful intelligent behavior (e.g., Moriarty and Miikkulainen 1995a,b, 1996; Richards et al. 1997; Gomez and Miikkulainen 1998; Agogino et al. 2000).

Chapter 4

Experimental Methodology

This chapter describes the details of the methods used for training and testing in the various experiments reported in chapters 5-8 below. Additional details that are only relevant to the experiments reported in a given chapter are described in that chapter.

Section 4.1 provides details on the use of pseudo-random numbers in the experiments reported in the following chapters, with a special view to their use in creating games with randomized properties. Then chapters 4.2 and 4.3 explain important details about how training and testing were done in the experiments described later. Finally, section 4.4 states the requirements for statistical significance tests, and explains why they have been provided for some of the dissertation's experiments but not others.

4.1 Random Number Management

Random decisions based on the output of pseudo-random number generators were used for both gameplay and the learning algorithms. This section describes issues raised by the use of a pseudo-random number generator in the learning experiments, and explains how those issues were addressed.

4.1.1 Repeatable Streams of Pseudo-Random Numbers

Repeatability is a desirable property for experimental computer science, so for randomized algorithms it is common to use a pseudo-random number generator rather than a genuine source of randomness. That approach was used for all the experiments reported here. For simplicity, the "pseudo-" will usually be dropped when discussing random numbers or random events, but it should be understood that all numbers and events were generated with a pseudo-random number generator.

Repeated invocations of a random number generator over a single run of a program will generate a stream of random numbers. Since the stream of numbers produced by a given generator

starting in a given state is actually deterministic, it is repeatable. E.g., if a given implementation of a random number generator is initialized with a seed of, say, 37, then it will always produce the same stream of numbers thereafter. It is therefore useful to think of the initialization seed and any other keys as a name for that deterministic stream of numbers. I.e., in the example above the '37' can be though of as a handle for that stream of numbers.

A genetic algorithm will consume very many random numbers if it runs for many generations and/or the breeding population is large. Due to the finite storage used to maintain a random number generator's internal state, any stream of pseudo-random numbers will eventually repeat. To protect against any artifacts of the cyclic stream of numbers, the Mersene Twister (Matsumoto and Nishimura 1998) was used as the random number generator for all the experiments reported here; the Mersene Twister was designed to provide an extremely long cycle time. The Mersene Twister allows a set of numeric keys to be used for initialization in addition to the seed. However, for all the work reported here the default value of those keys was retained.

Since a random number generator is just a state variable and an algorithm for producing a new number from a given state, it is possible for a program to manage independent streams of random numbers by using separate variables to initialize and preserve their state. This mechanism was used to generate repeatable games for the *Legion II* experiments, as described in the next section.

4.1.2 Repeatable Gameplay

When training or testing by means of dynamic gameplay rather than static examples, it is useful to have a definition for the concept of "the same game", e.g. to make comparative evaluations of the performance of embedded game agents. However, games that are genuinely identical with respect to the course of play are, in general, impossible to generate, if they involve embedded game agents that learn: as the agents learn, their behavior will change, and the changed behavior will cause the course of the game to vary from earlier plays. For example, if the legions in *Legion II* fail to garrison the cities during the early stages of training, the barbarians will occupy the cities. But later during training, when the legions learned to garrison the cities, the details of the barbarians' behavior must also change in response – i.e., the city will not be pillaged as before – even if there has been no change to the starting state and the barbarians' control policy.

It is therefore useful to have a pragmatic definition of "the same game" for experimental work. Thus for *Legion II* two games are identified as "the same game" if they use the same starting position for the cities and legions, and the same schedule for barbarian arrivals. The schedule for arrivals includes both the time and the randomly selected position on the map. For all the games reported here the barbarian arrivals were fixed at one per turn, so only their placement mattered for identifying two games as being the same.

However, the randomized placement of the barbarians is not always repeatable: as described in section 3.1.3, if the position selected for placing a new barbarian on the map is occupied, an alternative randomly selected position is used instead, and the re-tries continue until an empty map

cell is found. But as described above, changes to the legions' behavior will result in different game states at a given time in various instances of "the same game", so a barbarian positioning during one play of the game may not be possible in another run using a different controller for the legions. Therefore, for pragmatic reasons, "the same game" is defined for *Legion II* to consider *only the first try* for the positioning of arriving barbarians; the additional tries triggered by the unavoidable divergences of the game start are not considered to make two games different.

In principle it is possible to store a given game by recording the start positions for the cities and legions and the first-try placement positions for the barbarians. In practice it is easier to simply regenerate a given game upon demand by reproducing the stream of random numbers used for all the randomized placement decisions. All that is required is a state variable for the random number generator, as distinct from any other generators in use for other purposes, and variables for storing the generator's internal state, e.g. the starting state for a given game that needs to be re-created repeatedly.

However, it is essential to ensure that the same number of random numbers is consumed each time a game or set of games is generated. Otherwise the repeated use of the stream will fall out of synch with the original, and all following decisions based on the random numbers will be different than before. Therefore the generator used to generate the stream of numbers that define the course of a game was not used for any other purpose; separate generators (i.e., state variables) were provided for other randomized activities such as the decisions that are made by the evolutionary algorithm. A separate stream of numbers was also required for generating the re-try positions during barbarian placement, since that mechanism does not consume the same number of random numbers when a given game is replayed with different controllers for the legions.

The concept of "the same game" was used to create sets of games that were used repeatedly during training and testing, as described below.

4.2 Training

The legions' controllers were trained by two different base methods, neuroevolution and backpropagation. The details of the application of those methods are described in this section.

4.2.1 Stopping Criterion

Randomized learning algorithms such as neuroevolution do not always produce their best solution at the end of a fixed-length run; the random modifications to the representations in an evolutionary population can make the solutions worse as well as better. Likewise, supervised learning methods such as backpropagation do not always produce their best solution at a fixed-length run, as overtraining may result in memorization of the training examples rather than a good general solution. Therefore it is useful to have a mechanism for returning the best solution obtained at any time in the course of a run.

The commonly used mechanism is to evaluate candidate solutions periodically during training, and, if the top performer is better than any previously encountered during the run, to save that top performer as the potential output of the learning algorithm. At the end of the run, the most recently saved top performer is returned as the solution produced by the algorithm. The learning algorithm still *runs* for some fixed number of iterations that the experimenter deems sufficient to find a good solution, but that solution may be discovered at any time during the run.

The periodic evaluation is performed against a *validation set*. When generalization is desired, the validation set must be independent of the training data; otherwise the algorithm will return a solution that is biased toward the training data. For supervised learning, the validation set normally takes the form of a reserved subset of the available training examples. However, when annotated examples are not available, such as when using reinforcement learning to learn a motor control task or a controller for an embedded game agent, the validation set can be a standardized set of example problems. The definition of "the same game" in *Legion II* and an effective method for generating multiple instances of a given game allows construction of a distinctive set of games to serve as the validation set for *Legion II* learning tasks, and that is the mechanism used in all the experiments reported here.

Therefore stopping was handled in the experiments by running the learning algorithms for a period deemed to be "long enough", and using the validation set mechanism to control which candidate was actually returned as the result of a run. The validation set for *Legion II* was a set of ten games, created and re-created by means of an dedicated random number generator as described in the previous section. A set of ten games with independently generated starting positions and barbarian placement positions was judged to be a sufficient evaluation for generalization; larger sets adversely affect the run time of the evolutionary algorithm. The score for a controller's validation evaluation was defined as the average of the game scores obtained by play against the ten games of the validation set.

For consistency, validation by means of gameplay was used for both neuroevolution and backpropagation. Though backpropagation requires a set of annotated training examples for learning, when the trained network is to be used as the controller for an embedded game agent it can be evaluated by gameplay. As a side benefit, the number of available training examples did not have to be reduced by the reservation of a validation set.

When training with neuroevolution, an evaluation was made against the validation set at the end of each generation. When training with backpropagation, the learning network was evaluated against the validation set only after each ten iterations, so that the evaluation time would not dominate the run time of the algorithm and extend it excessively.

For a given run of a training program, the same validation set was used at each evaluation period. However, the validation set was created independently for each run. The idea is that each run should represent an independent sample of the space of all possible runs, for a given parameterization of the learning algorithm. And though the random number generator initialization seeds

are *in fact* passed to a randomized learning algorithm as parameters, they are only used to ensure that independent runs get independent streams of numbers; the specific numbers that appear in the stream are a model for a stream of genuinely random numbers, which are not considered to be parameters of an algorithm.

Some of the experiments described in chapters 6-8 evaluate solutions on the basis of behavioral properties other than their performance as measured by game scores. However, the performance-based evaluations against the validation set were used to control the learning algorithm's output even when other properties were being studied; solutions were not biased toward those properties by the validation mechanism.

4.2.2 Neuroevolution

Whenever the legions were trained by neuroevolution, the Enforced Sub-Populations (ESP) mechanism (Gomez and Miikkulainen 1999; Gomez 2003) was used. (See section 2.5.) Both the application and the details of the ESP implementation were novel; ESP has previously been used only to train fully recurrent networks for use as continuous-state controllers, e.g. for the inverted pendulum problem and the similar application to a finless rocket (Gomez and Miikkulainen 2003). In *Legion II* it is used for learning to make a discrete choice among the legions' possible atomic actions.

For the *Legion II* experiments reported here, the controller networks were non-recurrent feed-forward networks with a single hidden layer, as described in section 3.3.2 (figure 3.4). A sub-population was used for each neuron, regardless of which layer it was in; the representations in the populations held only the weights on the input side of the neurons (figure 4.1). In principle it is not necessary to provide separate neurons for the output layer; an architecture more similar to previous uses of ESP would have dispensed with those neurons and stored both the input and output weights in the representations of the hidden-layer neurons. That is in fact the mechanism used in the original *Legion I* experiments (Bryant and Miikkulainen 2003). However, the introduction of new sub-populations for the output layer contributed to the improved scores in the experiments reported here.

Fitness evaluations were obtained by playing the learners against randomly generated game setups; the set of possible game setups is so large that none ever have to be reused. A different sequence of training games was used for each independent run with a given parameterization in a given experiment. For fair evaluations within a single run, every neuron was evaluated against the same game (as defined above) before moving on to the next game.

When the ESP mechanism is used, the actual fitness of a network is ascribed to each neuron used to construct it. As a result, the ascribed fitness is only an estimate of a neuron's "true" fitness; the "true" fitness is in fact ill-defined, since the neurons are only useful when associated with other neurons in the other populations. However, a reasonable estimate of the fitness of a neuron *given that it will be used in a network with other neurons in the current populations* can be obtained by evaluating the neuron repeatedly, in networks comprised of one independently randomly selected

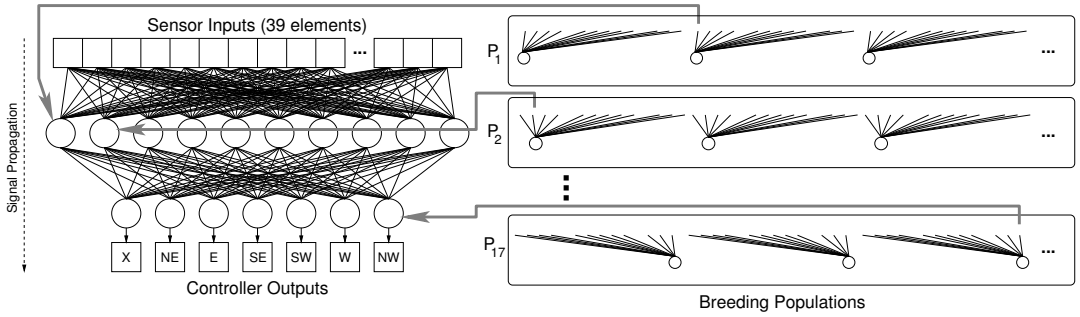


Figure 4.1: **Enforced Sub-Populations for the legions' controllers.** To apply the ESP method of neuroevolution for training the legions' controllers, a separate breeding population was maintained for each of the 17 neurons used in the controller network.

neuron from each sub-population.

Thus for all the neuroevolutionary learning experiments described here each neuron was evaluated on three different games per generation, and the three resulting fitness ratings are averaged to estimate the neuron's fitness.. The associations of the neurons into networks were re-randomized before each of the three games so that the averaged fitness ratings would reflect the quality of a given neuron *per se* more than the quality of the other neurons it happened to be associated with in the network. Each of the three evaluations used a different game setup, and all of the neurons were evaluated on the same three games during the generation.

Since the training game setups differed continually from generation to generation, learning progressed somewhat noisily: a neuron that performed well on the training games in one generation might not perform well on the new training games of the following generation. However, neuroevolution with ESP is robust even when evaluations are somewhat noisy, and the use of three games per generation helped smooth the noise of the evaluations. The continually changing stream of training games from generation to generation required candidate solutions to generalize to novel game setups, or else risk having their constituent neurons be weeded out of the breeding population; if a network performed poorly on the game setup used during a given generation it received a poor fitness score, regardless of how well it had performed during previous generations.

As described in section 4.2.1, an evaluation against the validation set was done at the end of every generation. For ordinary evolutionary mechanisms the network that performed best on that generation's fitness evaluations would have been chosen for the run against the validation set. However, the notion of "best network in the population" is ill-defined when the ESP mechanism is used, so for the *Legion II* experiments a *nominal best network* was defined as the network composed by taking the most fit neuron from each sub-population. It was that nominal best network that was evaluated against the validation set at the end of each generation.

Breeding was done by a probabilistic method that strongly favored the most fit solutions, but also allowed less fit solutions to contribute to the next generation with low probability. The

mechanism was as follows. When all the training evaluations for a generation were complete, the storage for the representations of the solutions in each sub-population was sorted from most fit to least fit, so that the most fit had the lowest index. Then each representation was replaced one at a time, starting from the highest index. Two parents were selected with uniform probability over the indices less than or equal to the index of the representation currently being replaced. I.e., that representation or any more fit representation could be chosen as a parent. The two selections were made independently, so that it was possible for the same representation to be used for both parents; in such cases the child would differ from the parent only by mutations. Notice that the mechanism *always* bred the most fit neuron with itself. Since the less fit representations were progressively eliminated from the effective breeding pool, the more fit solutions had more opportunities to contribute to the next population. Survey runs showed that this mechanism produced better results than an elitist mechanism.

Once a pair of parents were selected they were bred with either 1-point or 2-point crossover, with a 50% chance for each. Only one child was produced from the crossover; the remaining genetic material was discarded. Each weight in the representation was then subjected to a mutation at a 10% probability, independently determined. Mutations were implemented as a delta to the current weight chosen from the exponential distribution with $\lambda = 5.0$ (equation 4.1), inverted to the negative delta with a 50% chance.

$$f(x, \lambda) = \lambda e^{-\lambda x}, x \geq 0 \tag{4.1}$$

The choice of λ reduced the mean of the distribution, and was chosen on the basis of survey experiments. The deltas resulting from this distribution were small with high probability, but potentially very large with a low probability. That distribution allowed mutations to support both fine tuning of the weights and jumps to more distant regions of the solution space.

Training on the *Legion II* problem with neuroevolution appears to make progress asymptotically. For the experiments reported here, evolution was allowed to continue for 5,000 generations, well out on the flat of the learning curve, to ensure that comparisons and analyses were not made on undertrained solutions.

4.2.3 Backpropagation

The experiments reported in chapters 6 and 7 trained networks with backpropagation (Rumelhart et al. 1986b,a), either alone or in combination with neuroevolution. (See section 2.4.) When used alone it presented few complications; supervised learning on human-generated examples reduces the problem of training agent controllers to a Learning Classifier System problem, where each perceived game state must be mapped onto an appropriate choice between the possible moves. The only novelty in the application of backpropagation to the *Legion II* problem is the use of gameplay for the validation evaluations in lieu of a static list of annotated examples, as described in section 4.2.1.

As with neuroevolution, training on the *Legion II* problem by backpropagation on human-generated examples also appears to make asymptotic progress. Therefore when backpropagation

was used alone it was allowed to continue for 20,000 iterations, which preliminary survey runs had shown to put the solutions well out on the flat of the learning curve.

4.3 Testing

Each run of a learning algorithm returned a single neural network as its output. The networks were saved to files for later testing with a separate program. The test program spilled various run-time metrics to a file for analysis and plotting with the R statistical computing environment (R Development Core Team 2004).

Tests of current performance were also conducted during the course of training, for the production of learning curves. Tests are only made on those generations (for neuroevolution) or iterations (for backpropagation) where the evaluation on the validation set produced a new top performer, i.e. when measurable had been progress made. These were "side tests"; the learning algorithms ran the tests and spilled the results to a file for later analysis, but did not make any training decisions on the basis of the tests.

Whether during the learning run or afterward, tests were run on a set of games constructed as the validation set was, but independent of both the validation set and the training games. As with the validation evaluations, the evaluation score for this composite test was defined as the average of the scores obtained on the individual games of the test set.

Unlike the validation set, the same test set was used for every independent run of every learning algorithm, to ensure that any differences in the test metrics were the result of differences in the solutions being examined rather than differences in the difficulty of independently generated test sets. The training-time evaluations on the test set are not as frequent as the evaluations on the validation set, so a larger test set could be used without unduly extending the run times of the training algorithms. Also, it is essential that the test set be an accurate model of the set of possible games. Therefore a set of 31 games was used; the significance of that choice is explained in the next section.

The experiments reported in chapters 6, 7, and 8 had other goals that were best measured by metrics other than the game scores. Those metrics and the associated tests are described at an appropriate point in those chapters.

4.4 Note on Statistical Significance Tests

Parametric statistical tests such as the Student t-test require either that the samples being examined be drawn from the normal distribution, or else that sufficiently many samples (i.e. 30) be used that the sampling distribution of the mean of a sample is approximately normal (Pagano 1986). It is an established fact that the performances of solutions obtained by training artificial neural networks

do not distribute normally (Lawrence et al. 1997; Giles and Lawrence 1997). Therefore thirty independent runs are required for statistical significance tests to be done.

When possible, thirty-one independent runs were used rather than thirty, so that a well-defined median performer could be used for plotting as a "typical" outcome (e.g., figure 6.5). However, evolution using the play of complete games to compute fitness ratings is a very CPU-intensive task, and there was not always enough CPU time available to carry out the necessary number of runs using the desired number of generations and population size. Therefore statistical significance tests were only done for those experiments where the required number of runs were obtained. The number of runs differs between the experiments due to the differing availability of CPU time at the time the various experiments were conducted. Whenever a sufficient number of runs was obtained, the metrics of interest generated by the test program were evaluated for statistical significance at the 95% confidence level ($\alpha = 0.05$) using the Student t-test. That confidence level was chosen because it is the norm in the social sciences (Pagano 1986); we may as well adopt it for studying the behavior of embedded game agents as well.

Chapter 5

Adaptive Teams of Agents

This chapter examines the need for flexibility and adaptability in agents when they must cooperate as a team at a task that requires different activities by different individuals. A multi-agent architecture called the *Adaptive Team of Agents* (ATA) is proposed to meet that need, and its feasibility is examined experimentally. Section 5.1 motivates and introduces the ATA architecture, and then section 5.2 reports the experiments. Finally section 5.3 summarizes the findings regarding Adaptive Teams of Agents.

5.1 Introduction

Multi-agent systems are a commonplace in social, political, and economic enterprises. Each of these domains consists of multiple autonomous parties cooperating or competing at some task. Multi-agent systems are often formalized for entertainment as well, with instances ranging from team sports to computer games. Games have previously been identified as a possible "killer application" for artificial intelligence (Laird and van Lent 2000), and a game involving multiple autonomous agents is a suitable platform for research into multi-agent systems as well.

Multi-agent systems can be composed either of homogeneous agents or of heterogeneous agents. Heterogeneous teams are often used for complex tasks (e.g., Balch 1998; Haynes and Sen 1997; Yong and Miikkulainen 2001). However, heterogeneous teams of sub-task specialists are brittle: if one specialist fails then the whole team may fail at its task. Moreover, when the agents in a team are programmed or trained for a pre-specified division of labor the team may perform inefficiently if the size of the team changes – for example, if more agents are added to speed up the task – or if the scope of the task changes dynamically.

For example, suppose you owned a team of ten reactor cleaning robots, and the optimal division of labor for the cleaning task required two sprayers, seven scrubbers, and one pumper (figure 5.1). If the individual robots were programmed or trained as sub-task specialists the team would be brittle and lacking in flexibility. Brittle, because the failure of a single spraying robot

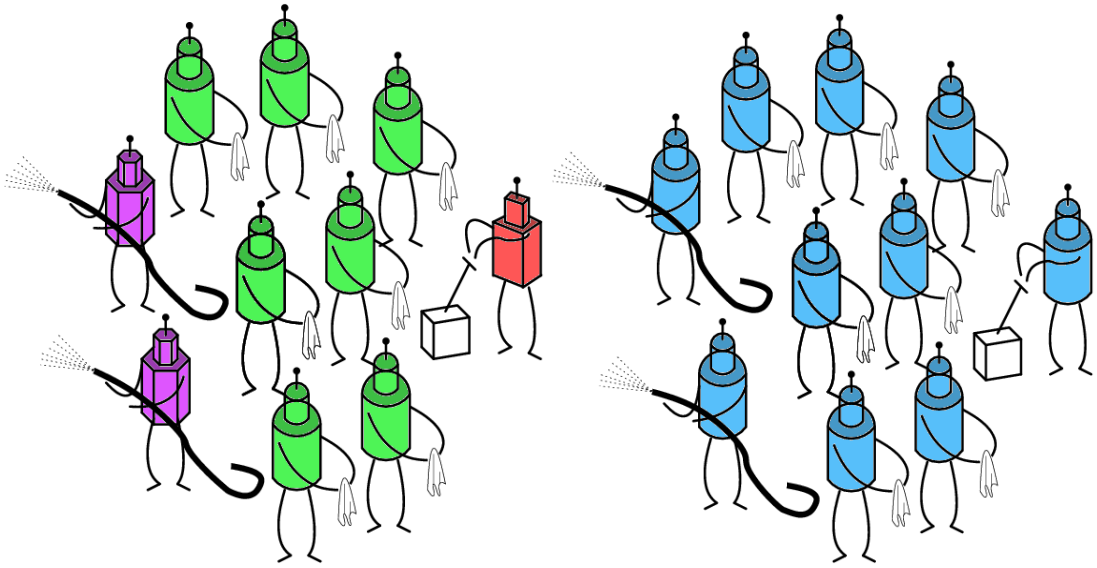


Figure 5.1: **The Adaptive Team of Agents.** *Left:* A heterogeneous team of cleaning robots is trained or programmed for sub-task specializations. Such a system is inflexible when the scope of the task changes, and brittle if key specialists are unavailable. *Right:* An Adaptive Team of Agents provides every agent with the capability of performing any of the necessary sub-tasks, and with a control policy that allows it to switch between tasks at need. The resulting team is more flexible and less brittle than the heterogeneous team.

would reduce the entire team to half speed at the cleaning task, or the loss of the pumper robot would cause the team to fail entirely. Inflexible, because if a client requested a 20% speed-up for the task you would not be able to simply send in two more robots; you would have to add $\lceil 20 \rceil \%$ more robots for each sub-task specialization, or four more robots rather than two.

An alternative approach is to use a team of *homogeneous* agents, each capable of adopting any role required by the team's task, and capable of switching roles to optimize the team's performance in its current context. Such a multi-agent architecture is called an *Adaptive Team of Agents* (ATA; Bryant and Miikkulainen 2003). An ATA is a homogeneous team that self-organizes a division of labor *in situ* so that it behaves as if it were a heterogeneous team. If composed of autonomous agents it must organize the division of labor without direction from a human operator, and be able to change the division dynamically as conditions change.

An ATA is robust because there are no critical task specialists that cannot be replaced by other members of the team; an ATA is flexible because individual agents can switch roles whenever they observe that a sub-task is not receiving sufficient attention. If necessary, an agent can alternate between roles continuously in order to ensure that sufficient progress is made on all sub-tasks. Thus for many kinds of task an ATA could be successful even if there were fewer agents than the number of roles demanded by the task.

Such adaptivity is often critical for autonomous agents embedded in games or simulators.

For example, games in the *Civilization*™ genre usually provide a "Settler" unit type that is capable of various construction tasks and founding new cities. Play of the game requires a division of labor among the settlers, and the details of the division vary with the number in play and the demands of the growing civilization – e.g. the choice between founding more cities vs. constructing roads to connect the existing cities. If the Settler units were heterogeneous, i.e. each recruited to perform only one specific task, there would be a great loss of flexibility and a risk of complete loss of support for a game strategy if all the Settlers of a given type were eliminated. But in fact the game offers homogeneous Settlers, and human players switch them between tasks as needed. For embedded autonomous Settlers that switching would have to be made by the Settlers themselves: an Adaptive Team of Agents is required.

This chapter explores the Adaptive Team of Agents experimentally, with a finding that it is possible to evolve an ATA with ANN controllers for a non-trivial strategy game. The game application was described in the previous chapter; the next section describes the use of the game in an ATA learning experiment, and examines the adaptive behavior of the evolved game agents.

5.2 Experimental Evaluation

ANN controllers for the legions in *Legion II* were trained using the base neuroevolutionary system described in chapter 4. The game parameters were set to require a division of labor to perform well: there were more legions than cities, the randomized placement of the barbarians and the 100:1 ratio of pillage between the cities and countryside made it essential to garrison the cities, and the large number of barbarians arriving over the course of the game made it essential to eliminate barbarians in the countryside as well, if pillage was to be minimized. With one barbarian arriving per turn, the count would ramp up from one to 200 over the course of a game in the absence of action by the legions, providing an average of $\sim$100 pillage points per turn. With three cities each subject to an additional 100 pillage points per turn, pillaging the countryside can amount to $\sim$1/4 of the worst possible score. Thus the legionary ATA must take actions beyond simply garrisoning the cities in order to minimize the pillage: a division of labor is required.

The following sections examine the results of the training experiment and the behavior produced in the legions.

5.2.1 Learning Performance

Hundreds of runs of neuroevolutionary learning on the *Legion II* problem, with a variety of learning parameters and a number of changes to the game rules and network architecture since the initial results reported in Bryant and Miikkulainen (2003), have consistently performed well, where "well" is defined fuzzily as "learns to bring the pillage rate substantially below the 25% threshold" obtainable by a policy of static garrisons and no division of labor to support additional activity by the spare legions. For the experiments reported here, eleven runs of the base learning method with the

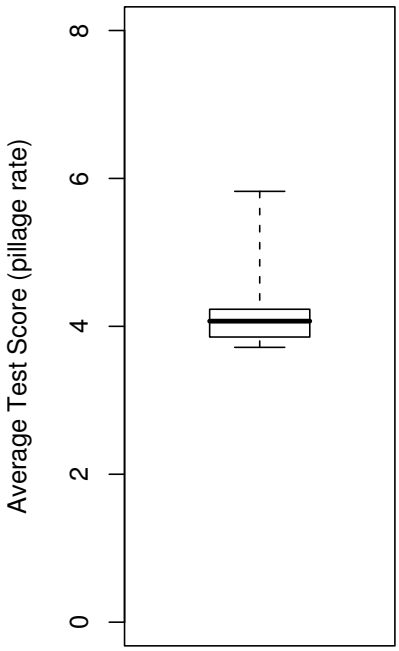


Figure 5.2: **Performance on the test set.** A boxplot shows the distribution of the performances of eleven trained networks on the test set. The boxplot shows the median and quartiles of the scores and whiskers show the extremes. Half the scores lie with the bounds of the box and a quarter lie within the range of each whisker. The asymmetry indicates the degree of skew to the set of scores. Lower scores are better, and the worst possible score is 100. A score of 25 is easily obtainable by learning to garrison the cities and take no further actions to eliminate barbarians; each of the eleven solutions learned to reduce the scores substantially below that threshold. Observation and additional metrics described below indicate that the improvement was obtained by learning to perform as an Adaptive Team of Agents.

parameters described in section 4.2.2 produced a mean test score performance of 4.316. The scores on the games in the test set show that all five runs produced controllers that allowed the legions to reduce pillaging well below the 25% rate obtainable by garrisoning the cities and taking no further actions against the barbarians (figure 5.2).

Notice that there is no *a priori* expectation that the legions will be able to completely suppress the pillaging. Since the warbands appear at random locations and move at the same speed as the legions, it becomes increasingly difficult for the legions to eliminate them as their density on the map decreases. Thus the performance of the legions is ultimately limited to an equilibrium condition between the number of legions in play, the rate of appearance of the warbands, and the size of the map.

As described in section 4.3, performance against the test set was also checked during training so that learning progress can be examined. A typical learning curve is shown in figure 5.3. The

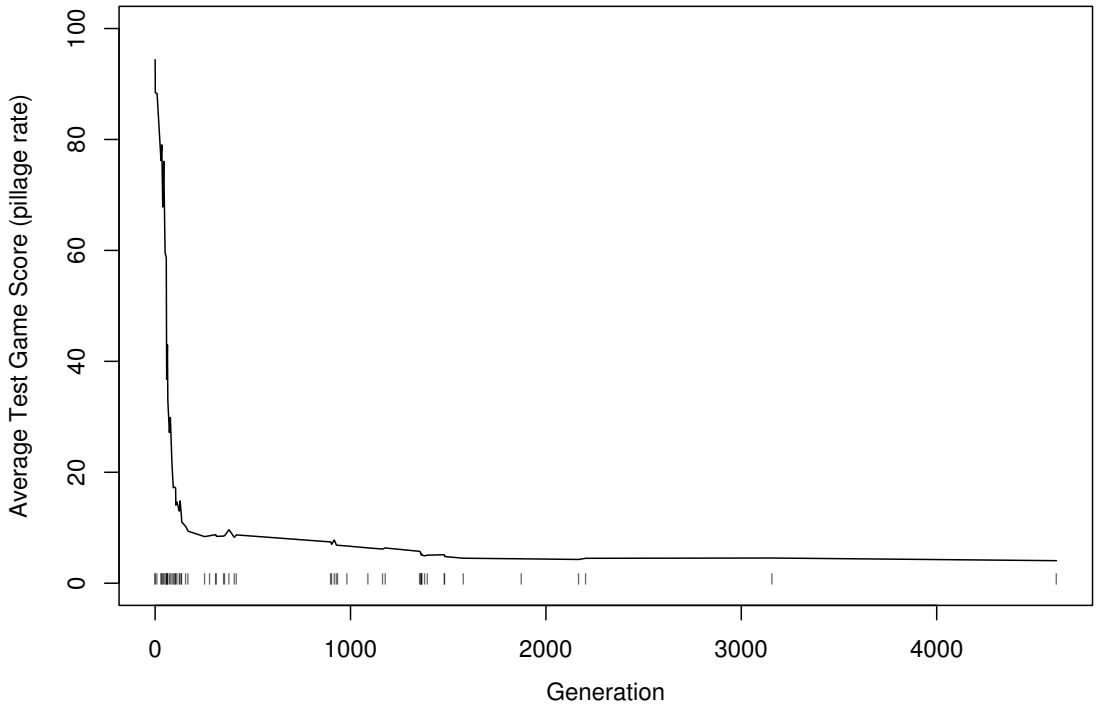


Figure 5.3: **A typical learning curve.** The plot shows progress against the test set for the median performer of the eleven training runs. At each generation when progress was made on a test against the validation set, the nominal best network was also evaluated against the test set. The hatch marks at the bottom of the plot identify those generations. The test scores for those generations are connected with lines to improve visibility. The plot is not strictly monotonic because progress on the validation set does not strictly imply progress on the test set.

learning curve familiar from many machine learning applications, though inverted because lower scores are better, with fast initial learning tapering off into slower steady progress. Experience shows that learning by neuroevolution on the *Legion II* problem appears to progress asymptotically. There is no stair-step pattern to suggest that the legions' two modes of behavior were learned sequentially; observations confirm that the legions begin chasing barbarians even before they have learned to garrison all the cities.

5.2.2 The Run-time Division of Labor

The behavior of the controller networks can be evaluated qualitatively by observing the real-time animations of game play. In every case that has been observed, the legions begin the game with a general rush toward the cities, but within a few turns negotiate a division of labor so that some of the legions enter the cities and remain near the cities as garrisons while the others begin to chase down barbarian warbands in the countryside. The only time the cities are not garrisoned promptly is when

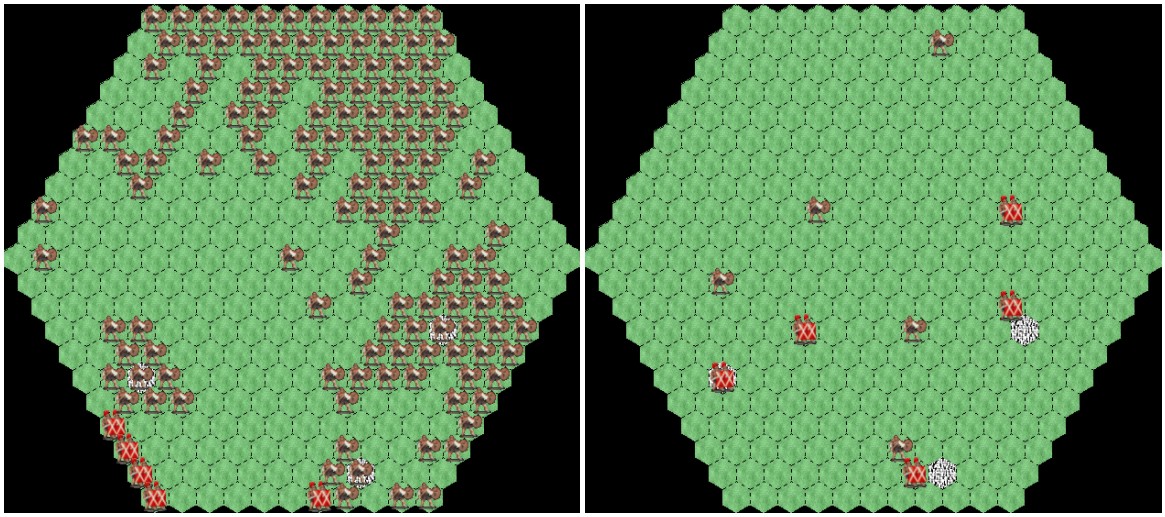


Figure 5.4: **Learning progress.** Two end-of-game screenshots show the legions' performance before and after training. *Left:* Before training the legions move haphazardly, drift to an edge of the map, or sit idle throughout the game, thereby failing to garrison the cities and allowing large concentrations of barbarians to accumulate in the countryside. *Right:* After training the legions have learned to split their behavior so that three defend the three cities while the other two move to destroy most of the barbarians pillaging the countryside. The desired adaptive behavior has been induced in the team.

two of them mask the third from the legions' low-resolution sensors. However, even in those cases the third city is garrisoned as soon as one of the roaming legions pursues a barbarian far enough to one side to have a clear view of the third city so that it can "notice" that it is ungarrisoned. A feel for these qualitative behaviors can be obtained by comparing end-of-game screenshots taken early and late during a training run, as shown in figure 5.4. An animation of the trained legions' behavior can be found at `http://nn.cs.utexas.edu/keyword?ATA`.

The legions' division of labor can can also be examined by the use of run-time metrics. The test program was instrumented to record, after each legion's move, how far away it was from the nearest city. The results for playing the median performer among the eleven networks against the first game in the test set are shown in figure 5.5. The plot clearly shows that after a brief period of re-deploying from their random starting positions three of the legions remain very near the cities at all times while two others rove freely. The rovers do approach the cities occasionally, since that is where the barbarians primarily gather, but for most of the game they remain some distance away.

When a rover does approach a city there is sometimes a role swap with the current garrison, but the 3:2 split is maintained even after such swaps. However, the legions show surprisingly persistent long-term behavior for memoryless agents: the plot shows that legion #1 acts as a rover for almost 3/4 of the game, and legion #2, after starting as a garrison and then swapping roles with a rover, spends the final 7/8 of the game in that new role.

Careful examination of figure 5.5 reveals that the average distance from the legions on gar-

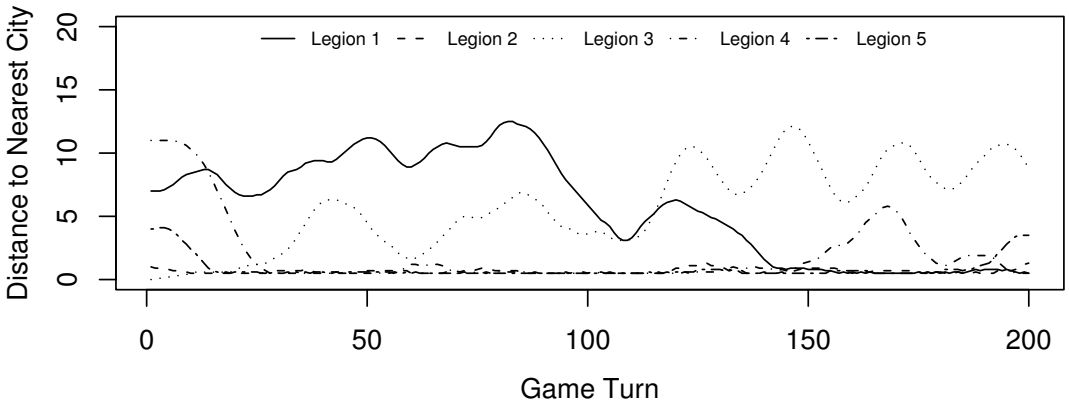


Figure 5.5: **Run-time division of labor.** The plot shows the distance from each legion to its nearest city over the course of a single game. The five legions start at random distances from the cities, but once some legion has had time to reach each of the three cities the team settles into an organizational split of three garrisons and two rovers. The legions sometimes swap roles when a rover approaches a garrisoned city, e.g. legions 3 and 4 just before turn 25. The numbering of the legions is arbitrary; they are identical except for the happenstance of their starting positions. The lines have been smoothed by plotting the average of the values measured over the previous ten turns.

rison duty to the nearest city is not actually zero as expected; in fact it hovers near 0.5. The movie reveals that that arises from the details of the garrisons' behavior: they oscillate in and out of the cities continually. The smoothing on the plot shows the 0/1 alternations as the non-zero average.

The oscillations sometimes allow a barbarian to slip into a city for a single turn, but also sometimes allow the elimination of barbarians that might have spent many turns pillaging the countryside otherwise. That trade-off between points lost and points gained was not biased enough to cause neuroevolution to settle on a different, more sophisticated solution. The issue of apparently erratic behavior in the details of the legions' behavior is deferred to chapter 6.

5.2.3 Mid-game Reorganizations

The training problems were parameterized to require the legions to organize a division of labor at the beginning of the game, and they successfully learned to do that. However, the motivation for the ATA multi-agent architecture calls for teams that can reorganize whenever a change in circumstances requires it. For example, if the pumper robot in the motivating example breaks down, one of the other robots should take over the task so that the team will not fail entirely. The legions in the *Legion II* game should also be able to reorganize at need.

That necessary ability was examined by modifying the test program to support the addition or removal of a city at the mid-point of the test games. When a city is added, the legions should reorganize into a team of four garrisons and one rover, or when a city is removed they should reorganize into a team of two garrisons and three rovers.

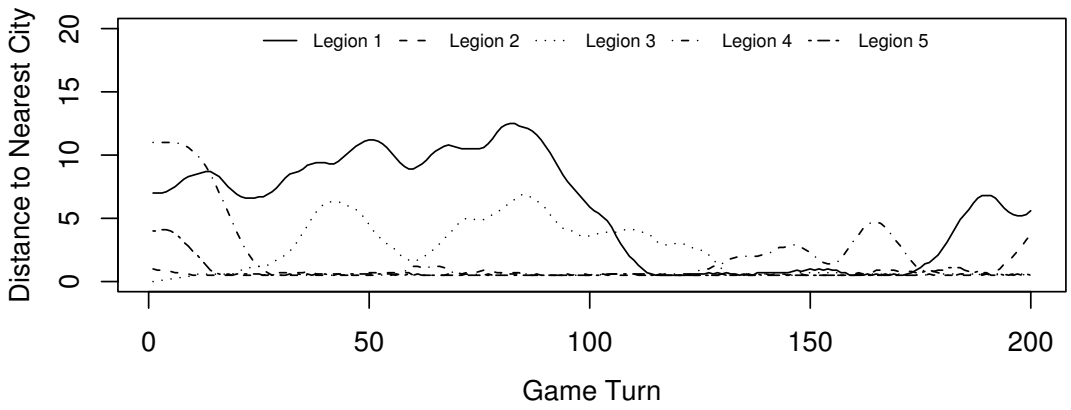


Figure 5.6: **Mid-game reorganization of the team (i).** In a second test, play follows the pattern of the previous test until turn 100, when a fourth city is added to the map. The team is forced to re-organize its division of labor so that there are now four garrisons and only a single rover. The role swaps continue, but they always leave four legions hovering near the cities. The lower distance from the rover to its nearest city after the city is added is an artifact of the increased density of cities on the map. Legion 2 erroneously abandons its city near the end of the game; see the text for the explanation.

The result of adding a city is shown in figure 5.6. The plot again shows the median-performing controller's behavior on the first game in the test set. Since the legions' behavior is deterministic and the game's stochastic decisions are repeated, as described in section 4.1.2, the game follows its original course exactly, up until the city is added at the mid-point of the game. Thereafter the team can no longer afford to have two rovers, and the plot shows the resulting re-organization. There are still role swaps in the second half of the game, but the swaps now always maintain four garrisons and a single rover.

The average distance from the rover to the cities is lower in the second half of the game. That is primarily an artifact of having more cities on the small map: regions that were once distant from all the cities no longer are, so even without any change in behavior the rover is expected to be nearer some city than before. A second cause is an indirect result of the change in the team's organization. With only one rover in the field, the legions are not able to eliminate the barbarians as quickly as before, so during the second half of the game the concentration of barbarians on the map builds up to a higher level than previously. Since they tend to crowd around the cities and the roving legions tend to chase down the barbarians wherever they mass, the roving legion now has more reason to operate close to the cities.

The plot shows legion #2 vacating the city it was garrisoning right at the end of the game. That is also an artifact of the increased density of the barbarians on the map. In ordinary play the trained legions are able to maintain a dynamic equilibrium between the rate of influx of the barbarians and the rate they are eliminated; the denser the barbarians are on the map, the easier it is for the rovers to catch some of them. However, when the add-city test causes one of the rovers to swap roles

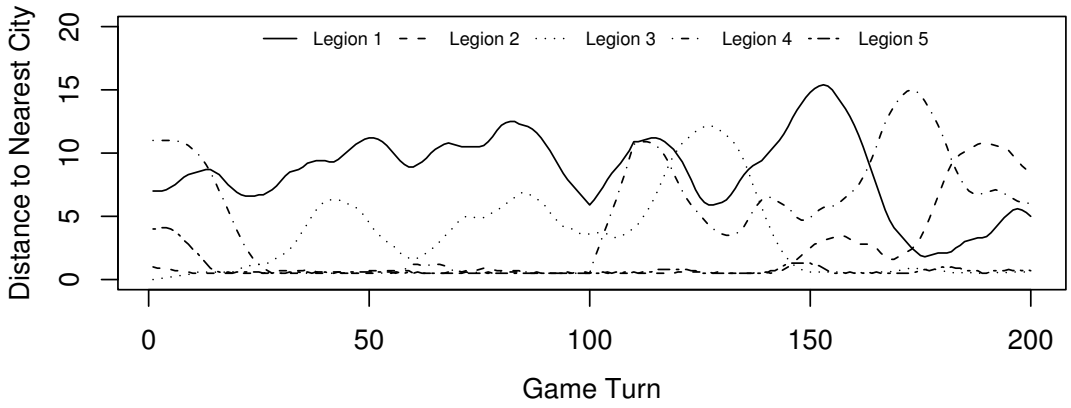


Figure 5.7: **Mid-game reorganization of the team (ii).** The mid-game reorganization experiment is repeated, except this time a city is *removed* from the map. That city had been garrisoned by legion 4, which now finds itself far from the nearest city, and immediately adopts the role of a rover. (The steep ramp-up of its nearest-city distance on the plot is an artifact of the smoothing; the change is actually instantaneous when the city is suddenly removed from under the legion.) There is a role swap just before turn 150, but the team consistently keeps two legions very near the two remaining cities, with the other three roving at various distances.

to garrison duty, that equilibrium can no longer be maintained by the single remaining rover, and the number of barbarians in play starts ramping up after the city has been added. Eventually their number oversaturates the legions' sensors – they have not seen such densities since early during training – and the legions begin to behave erratically. However, until their sensory input diverges far from what they were trained for, the legions exhibit the desired behavior.

The result of removing a city at the mid-point of a game is shown in figure 5.7. The play proceeds as before until the city is removed at turn 100. At that point the legion formerly garrisoning that city finds itself far away from any, but it adopts the roving behavior rather than sitting idle or trying to crowd into one of the other cities, and it maintains that behavior for the remainder of the game. There is a role swap between two of the other legions later, but the team is always left with two legions hovering very near the cities on garrison duty, while the other three range over various distances in pursuit of the barbarians.

5.3 Findings

The experiments show that the Adaptive Team of Agents is a feasible architecture for multi-agent systems, and that ATAs can be created by neuroevolutionary methods. The legions learned the desired variety of behavior, and the ability to organize a division of labor by individually adopting an appropriate choice of behaviors. They also learned to swap roles without disrupting the required organization of the team, both in the ordinary course of events and in response to a change in the

scope of their task.

As described in section 5.1, such capabilities are essential for autonomous embedded game agents. Games often provide multiple agents of some generic type – e.g. the Elvish Archer type in the open source game *The Battle for Wesnoth* – which must as individuals pursue differing activities that contribute to the success of the team rather than the individual. And those agents must be adaptive in their choice of activities, taking account of the observed game state, including the choices being made by their peers. Yet the scripted behavior of agents in commercial and open source games commonly fail at that requirement, making decisions that appear to take little account of context. For example, in strategy games, even when the agents are not autonomous and a higher-level AI is able to manipulate them according to some plan, individual agents are frequently observed to make piecemeal attacks that are suicidal due to a lack of supporting actions by their peers. To a human observer, such agents simply do not seem very intelligent. Appropriate machine learning methods should be able to take game intelligence beyond the brittleness, inflexibility, and narrowness of scripted activity, and for many games or simulators the context-awareness and adaptivity of the agents in an ATA will be a necessary part of any successful solution.

The *Legion II* ATA experiments also reveal a special challenge for the application of machine learning methods to agent behavior problems. The goal as understood by the machine learning algorithm – e.g. minimizing pillage in the *Legion II* game – may be met with little or no regard for the appearance of details of the learned behavior that make little difference to the goal implicit in the training regimen. For example, the legions in the *Legion II* experiment learn to display visibly intelligent behavior at a high level by choosing appropriate roles on the basis of context, but some of the details of their behavior are not equally satisfactory. The garrisons' "mindless" oscillations in and out of their cities would likely be the subject of ridicule if seen in the behavior of the agents in a commercial game. In principle such details of behavior could be addressed by careful specification of the goals of the training regimen, such as an evolutionary reward function that penalizes undesirable behavior, but for applications as complex as a commercial game it may be as difficult to specify an appropriate reward function as it would be to write a script that covers all situations adequately. Therefore chapter 6 will explore an alternative, general approach to refining the behavior of agents with evolved controllers.

Chapter 6

Example-Guided Evolution

This chapter examines the feasibility of using human-generated examples of gameplay to guide evolutionary learning to produce agents that behave in a manner similar to the way they were operated by the human player. Section 6.1 motivates and explains the concept, and section 6.2 addresses details of methodology specific to the use of example-guided evolution, beyond what has already been addressed in chapter 4. Then sections 6.3 and 6.4 report sets of experiments using different sets of human-generated examples reflecting two distinct behavioral doctrines. Finally, section 6.5 summarizes the findings regarding example-guided evolution.

6.1 Introduction

Chapter 5 illustrated training an Adaptive Team of Agents with neuroevolution; it also illustrated the arbitrary nature of the solutions genetic algorithms can produce. In the solutions to the *Legion II* problem obtained by evolution, the legions assigned to garrison duty oscillate in and out of their cities "mindlessly", i.e. there is no apparent motivation for the oscillating behavior. But since there is no direct cost for that behavior, and little detriment to the game scores, evolution of the controller does not weed the behavior out. Yet to a human observer the behavior does not seem appropriate for a legion garrisoning a city. So the new challenge is: how do we inform the evolutionary process with human notions of appropriateness?

In principle a more apropos behavior could be obtained by modifications to the evolutionary fitness function, e.g. by charging a fitness cost for each move a legion makes during the course of a game. However, when it is necessary to learn complex behavior it may be as difficult and time-consuming to specify and tune the necessary collection of cost factors as it would be to directly specify the desired behavior with a script or a collection of rules.

Therefore it is useful to consider other approaches, such as example-guided evolution. Example-guided evolution retains the benefits of evolutionary search and optimization – delegating the construction of a solution to machines rather than to human analysts – while inducing the

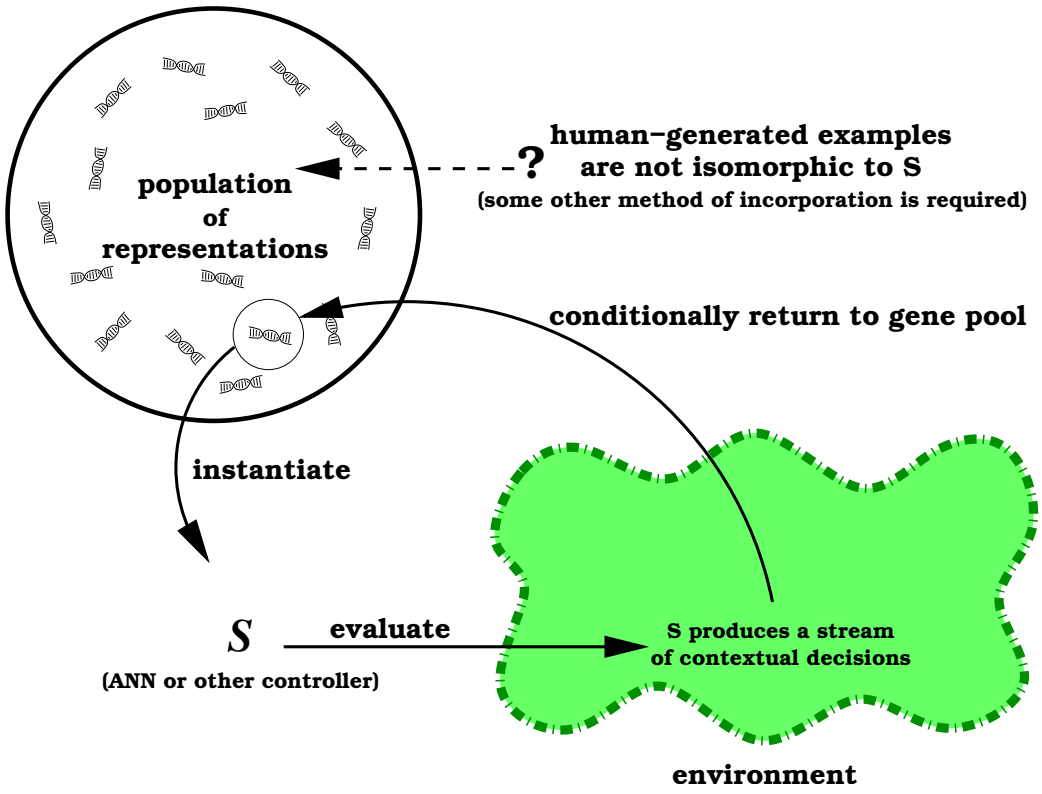


Figure 6.1: **Indirect specification of a solution.** If solution S is a controller that is too complex for direct specification by a human analyst, it can be specified indirectly in terms of the sort of contextual decisions it should make.

evolutionary result to be similar to a human-like solution specified relatively easily by means of examples.

When the solution produced by a genetic algorithm takes the form of a complex object not easily constructed by humans, such as a non-trivial artificial neural network, the genetic algorithm cannot rely on case injection with directly specified solutions. However, if the complex solution is an agent's controller then it can be specified, partially and indirectly, by means of examples of the behavior that it should produce (figure 6.1). If the agent is embedded in a game then such examples can be created relatively easily by adding game controls that allow a human to operate the agent, and a system for observing and recording the game state and the human's decisions. Then each time the human player uses the game controls to signal a decision, the observer records the game state and the decision as a $<$*state,action*$>$ tuple, which serves a a single example of play. The training examples collected over the course of one or more games can then be used to guide evolution toward a solution that behaves similarly (figure 6.2).

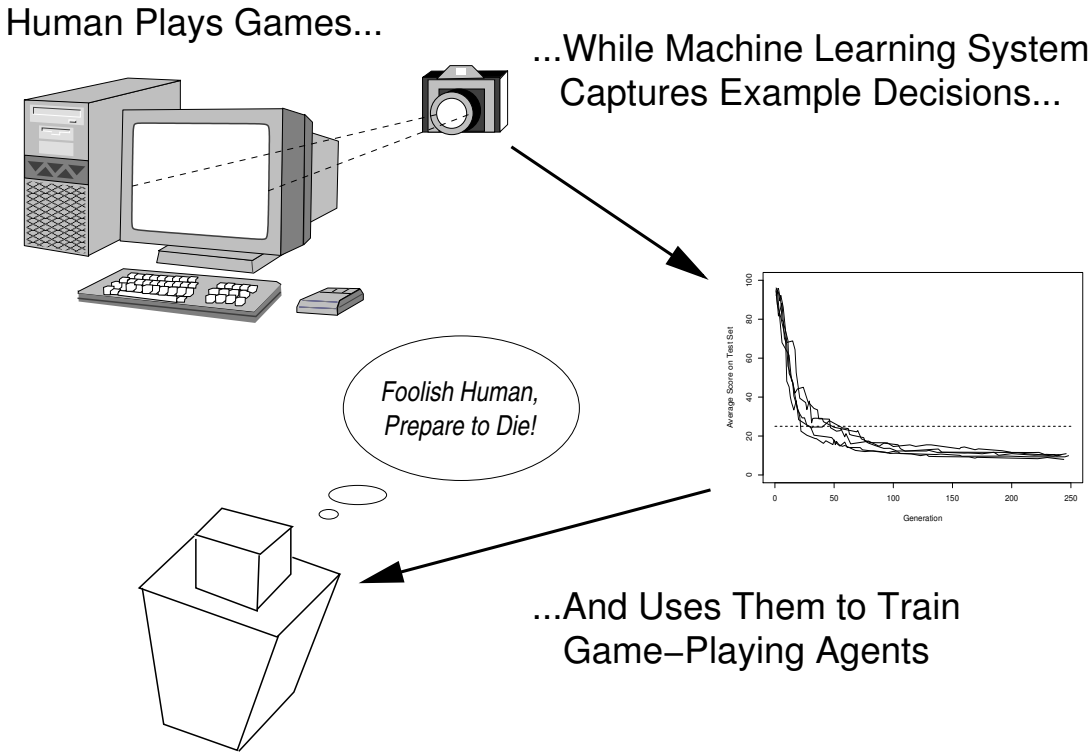


Figure 6.2: **Learning by observing in a game context.** Human examples of play can be captured by writing an observer into the game engine, and examples collected by the observer can be used to help train a controller for a game agent.

When the controller is an artificial neural network that maps sensory inputs onto behavioral outputs, then examples of the input-output mapping can be used to implement Lamarckian neuroevolution. Lamarckian evolution allows adaptation in a candidate solution (phenotype) during its lifetime, i.e. before or during its fitness evaluation, reverse-engineers a modified representation (genotype) from the adapted candidate solution, and replaces the original representation in the gene pool with the modified representation, which is credited with the fitness resulting from the adapted solution's evaluation (figure 6.3). When examples of the desired input-output mapping are available, Lamarckian neuroevolution can be implemented by using a supervised learning method such as backpropagation (Rumelhart et al. 1986b,a) for adaptation in the candidate solutions.

This chapter examines the application of that strategy of guided evolution to the *Legion II* game. Lamarckian neuroevolution with adaptation via backpropagation is evaluated experimentally, and is found to be capable of suppressing the garrisons' undesired oscillating behavior. Additional observations and metrics show that example-guided evolution produces agents that behave according to distinctive doctrines, and that specific rules of behavior used to define those doctrines – only implicit in the human-generated examples – are induced into the evolved controllers.

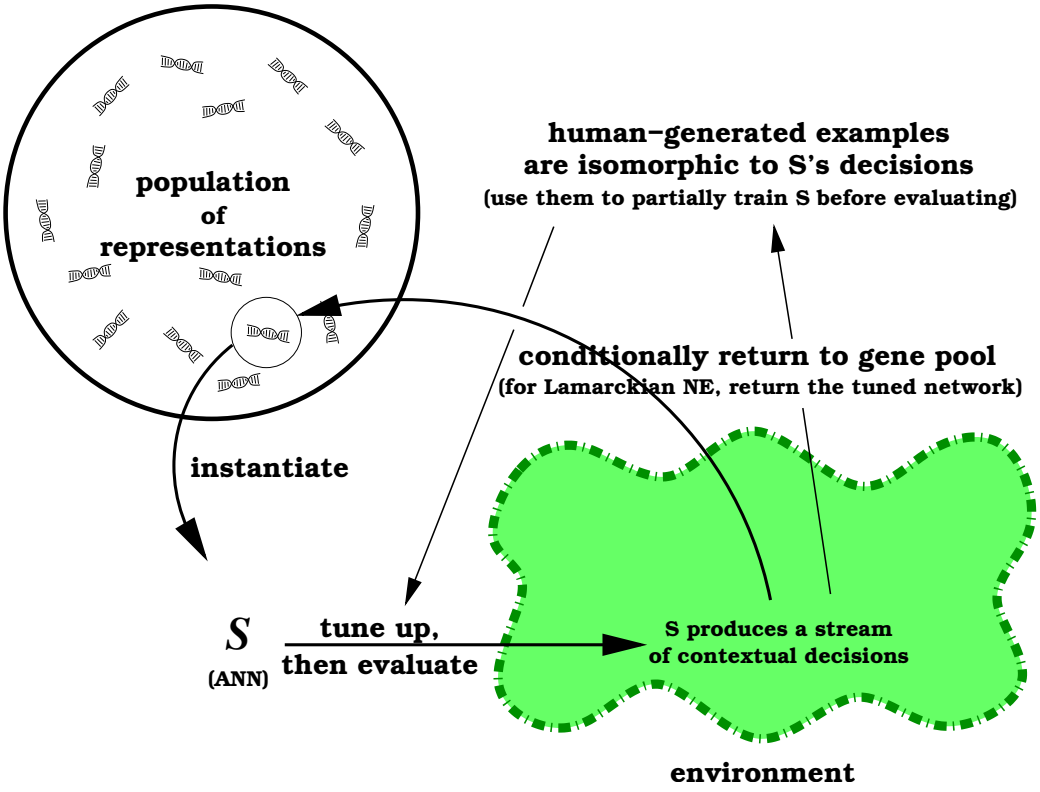


Figure 6.3: **Lamarckian neuroevolution.** Lamarckian neuroevolution uses examples of desirable behavior to tune a controller before its fitness evaluation, and returns a representation of the *modified* controller to the gene pool. As a result, human notions of desirable behavior are incorporated into the output of the genetic algorithm.

6.2 Methods

Examples of human play for the *Legion II* game were collected as described above, i.e. a user interface was added to the game and the game engine was modified to record each human use of the controls as an example of appropriate behavior. Since the legions' controller networks must learn to make decisions on the basis of their sensory inputs, the examples were recorded in terms of the game state as it was seen via the legions' sensors. Thus the $<$*state,action*$>$ tuples used a 39-element floating point vector of egocentric sensory activations for the *state*, and a symbol for one of the legion's seven discrete choices – to remain stationary or move to one of the six adjacent map cells – for the *action* (figure 6.4).

The author generated two separate sets of examples, each using a different doctrine to constrain the choice of actions for the legions. The doctrines are named *Level Zero* (L_0) and and *Level One* (L_1), where doctrine L_i requires a garrison to remain within distance i of the city it is guard-

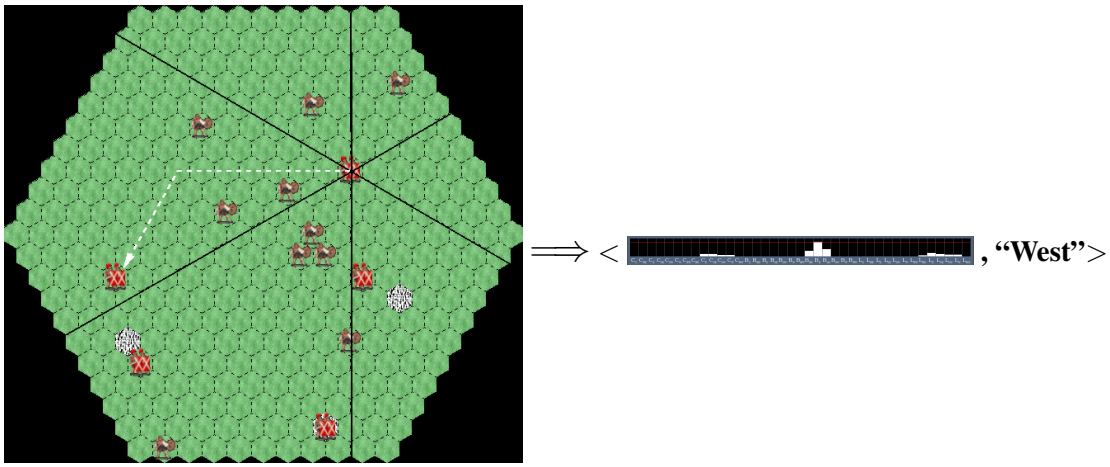


Figure 6.4: **Training examples for the *Legion II* game.** To train embedded game agents for the *Legion II* game, the captured <*state,action*> tuples consist of the legion's egocentric sensory perception of the game state and the human's choice of discrete actions for the legion. For example, if the human chooses to move the legion at the upper right to the west when the map is in the game state shown, the built-in observer records the 39-element pattern of the legion's sensor activations and the discrete symbolic choice “West”.

ing. In addition, the doctrines only allow a garrison to leave its city to eliminate a barbarian within range i; it must remain in the city if no such barbarian is present. The doctrines were formulated to allow $i > 1$ as well, but due to the fuzziness of the legions' sensors at distances greater than 1, no experiments have been conducted with $i > 1$.

Since the barbarians move after the legions each turn, any barbarian that is as close to a city as the garrison after that garrison's move will be able to beat the garrison to the city and inflict at least one turn of pillage on it. Thus in addition to the distance constraint the doctrines impose a safety condition on the garrisons: regardless of i, a garrison must move directly back to its city unless it can end its turn strictly nearer to the city than any barbarian.

Thus the L_0 doctrine requires a garrison to remain fixed in its city, but the L_1 doctrine allows a garrison to move to any of the map cells adjacent to its city under appropriate circumstances. Notice that all other constraints are trivially satisfied when a legion conforms to the distance limit of the L_0 doctrine, but not for doctrines with $i > 0$. An implicit constraint that arises from combining the L_1 doctrine with the safety condition is that a legion following the L_1 doctrine can move out of the city to eliminate a single adjacent barbarian, but is locked in the city when there are no adjacent barbarians, or more than one.

Twelve example-generating games were played using each doctrine. The pseudo-random numbers used by the game engine were marshaled to ensure that the same twelve game setups were used when generating the two sets of examples. With five legions in play and 200 turns per game, each game produced 1,000 examples. Since the legions were expected to behave as an Adaptive Team of Agents, every example generated during a game was equally relevant for every legion,

and thus for a given doctrine the examples collected from all five legions could be combined into a single pool. The net result was two sets of 12,000 human-generated examples of play, with some variety in the locations of the cities on the map within each set, and a single behavioral doctrine used throughout each set.

One game's examples from each set were reserved for testing, and the remaining 11,000 examples were used for Lamarckian neuroevolution, with backpropagation as the mechanism of adaptation in the phenotypes. The network architecture and evolutionary training parameters from the ATA learning experiment described in chapter 5 remained unchanged.

Use of the Lamarckian mechanism with neuroevolution is orthogonal to the use of the ESP mechanism, so both were used together. With 5,000 generations of evolution and three fitness evaluations per generation, there were 15,000 exposures to the training set for each member of the population. Due to that large amount of training a relatively low backpropagation learning rate of $\eta = 0.001$ was used.

For a constant-time mechanisms of phenotypic adaptation such as backpropagation, the overall training time increases approximately linearly with the number of training examples used. Moreover, there was concern that excessive training with backpropagation would disrupt convergence in the co-evolution between the populations used by the ESP algorithm: since the neurons are put into networks consisting of a random selection of peers, the gradient descent used by backpropagation might pull the neuron's weights toward very different solutions when the neuron participates in the different networks. Therefore the training time and number of backpropagations were reduced by training on only a subset of the available examples each generation, and the results of using different subset sizes were examined experimentally.

Subsets of sizes ranging from 5 to 5,000 examples were used. The full set of 11,000 training examples for a given doctrine was subsetted by sampling. A random sample of the desired size was selected and used to drive adaptation in all the members of the evolutionary population as they were evaluated; the same subset was used once for each network to be evaluated. Thus since the populations were evaluated three times per generation for ESP fitness averaging, three different random subsets of the training examples were used each generation. The samples were selected independently each time a new subset needed to be constructed.

Eleven training runs were started for each of ten subset sizes, for each of the two training sets generated using the different behavioral doctrines. In a few cases only nine or ten of the eleven runs ran to completion, due to circumstances unrelated to the learning algorithm. Since learning appears to be asymptotic for applications of neuroevolution to the *Legion II* problem, evolution was allowed to run for 5,000 generations in order to ensure that the training was well out on the flat of the learning curve when it terminated (figure 6.5).

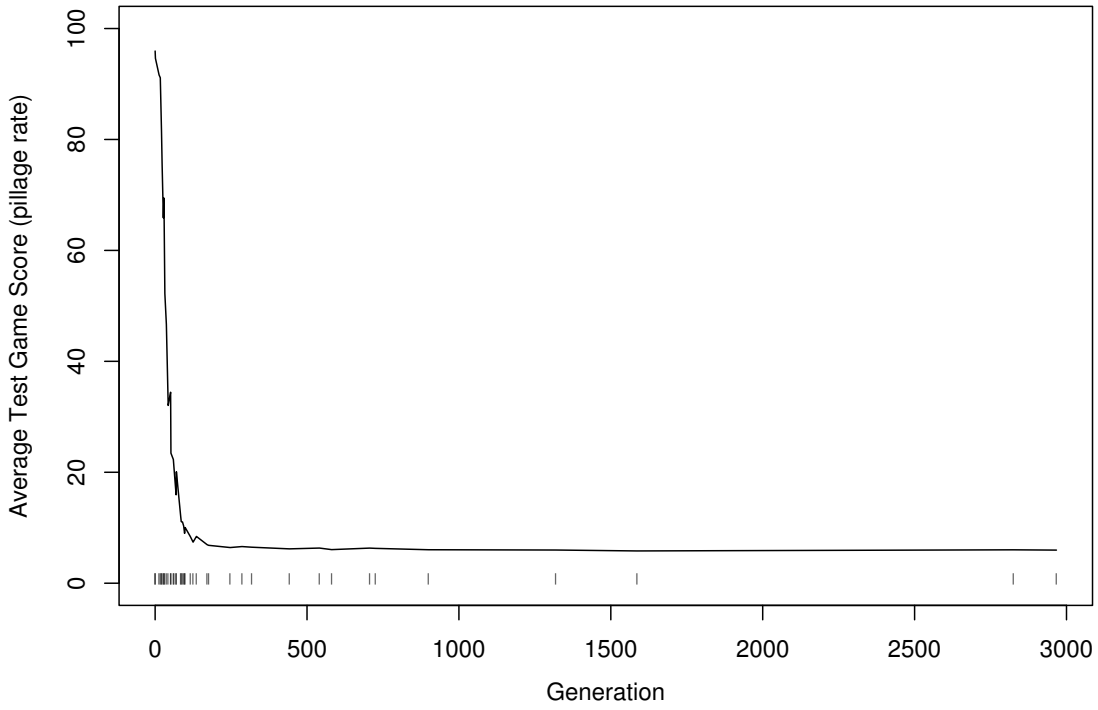


Figure 6.5: **Typical learning curve for Lamarckian neuroevolution.** The solid line shows how game scores varied by generation for Lamarckian neuroevolution using 5,000-example subsets of the 11,000 L_0 examples for adaptations. (Lower scores are better.) Points are plotted to show the score on the test set obtained on those generations where progress was made on the validation set, and the points are connected with straight lines to improve visibility. The ticks below the curve identify those generations. The curve stops at generation 2,886, as no progress was made after that on the 5,000 generation run. The dotted line shows progress on the coarse behavioral metric described in section 6.3.1, below.

6.3 Results Using the L_0 Training Examples

6.3.1 General Results

The application of Lamarckian neuroevolution to the *Legion II* problem using the L_0 training examples can be evaluated by the performance of the resulting controller networks on the test set of standard games, as was done for unguided evolution in chapter 5. The results for the various example sample sizes are shown with box-and-whisker diagrams in figure 6.6. The salient feature of the results is the discontinuity when the example sample size reached a few hundred: for the smaller sample sizes the median test score hovered around four, but for the larger sample sizes it worsened to around six.

However, our current focus is on behavior rather than on game performance *per se*. A behavioral fidelity metric can be constructed by comparing how the trained controllers respond to a

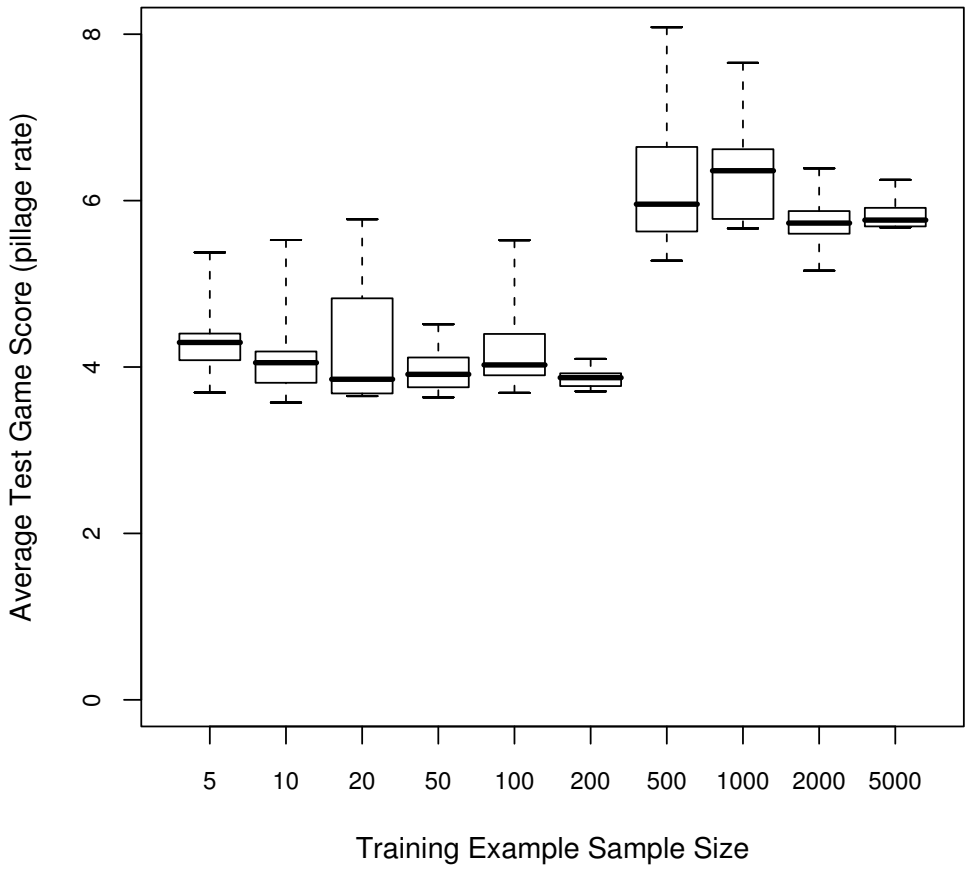


Figure 6.6: **Game performance vs. L_0 training examples used per generation.** (Lower scores are better.) When the number of randomly selected training examples used per generation of Lamarckian neuroevolution is increased, the effect on the performance of the trained networks is not continuous. Rather, there is a sudden jump to worse performance when more than a few hundred training examples are used per generation.

given situation with the way a human player using the same strategy actually did. For this purpose the 1,000 human-generated examples of play reserved for testing as described in section 6.2 can be treated as a classification problem: for a given game state and behavioral doctrine, what is the correct action among the set of seven discrete possibilities? If the actions taken by the human player while using a specific doctrine to generate a test set are considered "correct", controllers with the lowest classification error rates on that test set can be reckoned as being the most faithful to the doctrine followed by the human player.

The results of testing the networks trained with various example sizes against this behavioral similarity metric are shown with box-and-whisker diagrams in figure 6.7. Again, the salient feature of the results is a sudden discontinuity when the sample size reached a few hundred. This time, however, the jump was an improvement: for the smaller sample sizes the median classification error

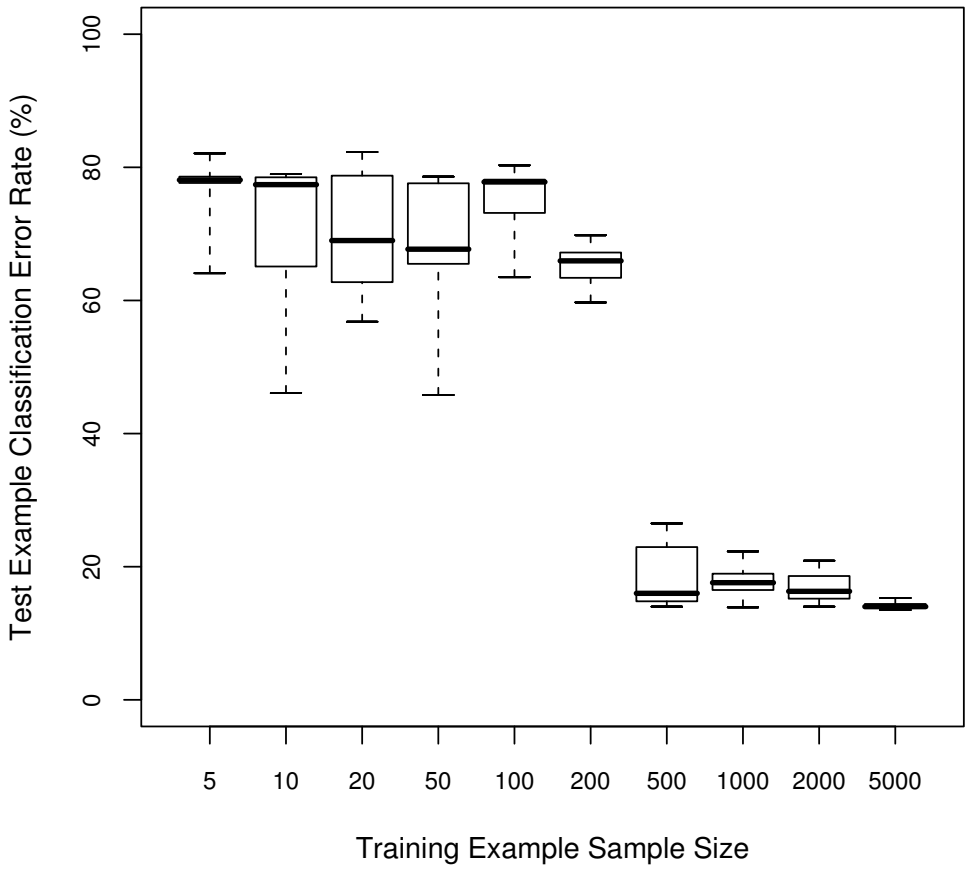


Figure 6.7: **Behavioral similarity vs. L_0 training examples used per generation.** (Lower rates are better.) As with the measured game scores, there is a sudden discontinuity in the behavior metric when more than a few hundred training examples are used per generation. In this case the jump is to a far better class of solutions.

rate fell in the range 65-80%, but for the larger sample sizes the rate fell to 14-19%.

The meaning of the common discontinuity in the two metrics – game scores and behavioral similarity – can be understood when the metrics are plotted one against the other in a phase diagram (figure 6.8). The locations in the phase space of the solutions obtained using various example sample sizes are given by symbols at their mean values on the two axes. The locations for solutions obtained by human play using the L_0 doctrine, unguided evolution, and 20,000 iterations of simple backpropagation on the full set of 11,000 L_0 training examples are also shown for comparison.

The phase diagram reveals the discontinuities seen in figures 6.6 and 6.7 as symptoms of a jump from solutions similar to those obtained by unguided evolution (i.e., nearby in the phase space) to solutions similar to human play using the L_0 doctrine. An immediate conclusion is that Lamarckian neuroevolution using backpropagation for adaptations does not meet the goals of example-

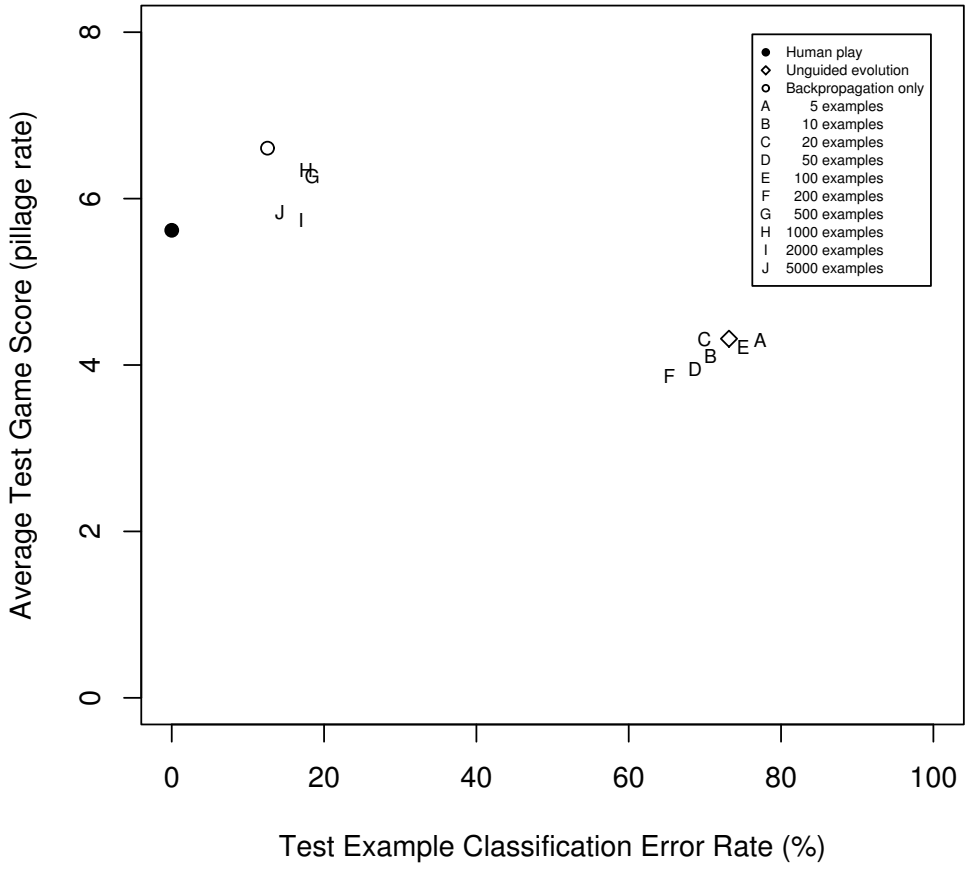


Figure 6.8: **Performance-behavior phase diagram for L_0 examples.** When the mean game scores are plotted against the mean values of the behavior metric, the mixed message of figures 6.6 and 6.7 is explained. When too few training examples are used per generation (A-F), Lamarckian neuroevolution produces controllers that behave and perform similarly to the controllers produced by unguided neuroevolution. When sufficient examples are used (G-J), the resulting controllers behave and perform more similarly to the human player. The game scores suffer with increasing behavioral fidelity because the L_0 strategy simply is not as effective as the strategy acquired by unguided evolution. The result of using backpropagation without evolution is shown for comparison; it performs slightly better than the 5000-example Lamarckian method (J) on the behavior metric, and a bit worse on the game scores.

guided evolution when an insufficient number of training examples is used. When too small an example sample size is used for training, 200 or fewer for the *Legion II* application (A-F on the phase plot), there are minor absolute improvements in the game scores – a move *away* from the scores obtained by the human player – and very minor improvements in behavioral similarity. Both trends generally increase with an increasing sample size, though not monotonically.

However, when a sufficiently large example size is used for training, 500 or more for the *Legion II* application (G-J on the phase plot), the solutions not only operate nearer the human's score

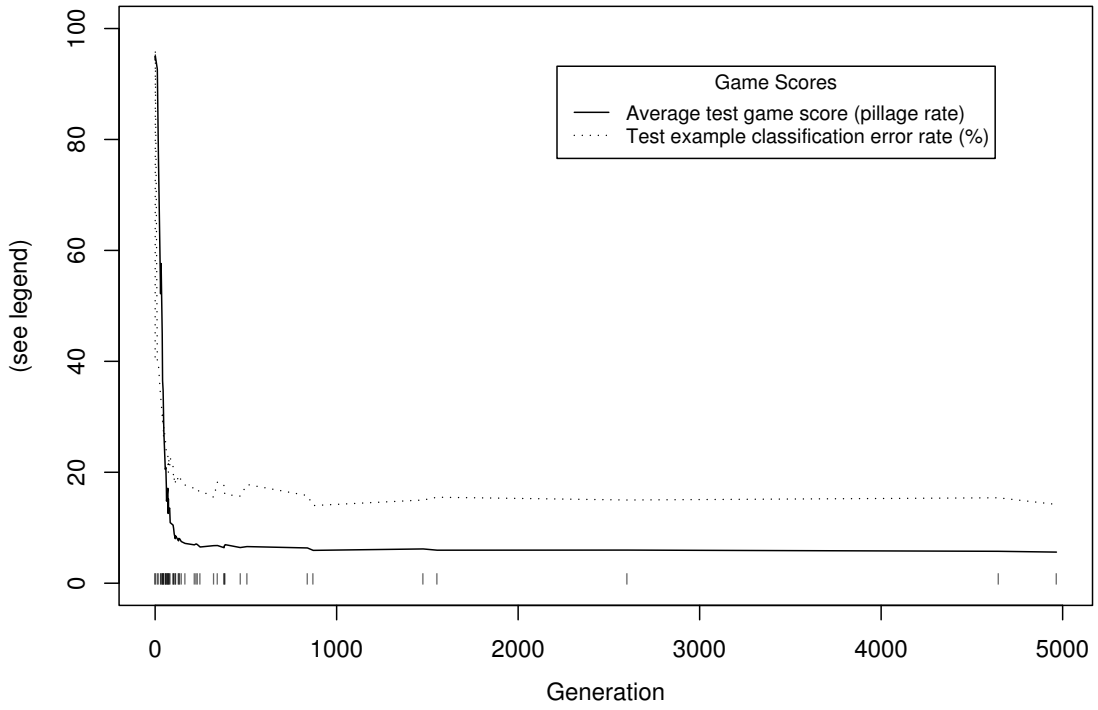


Figure 6.9: **Non-monotonic progress at example classification.** The dotted line shows progress on the classification-based behavioral metric as Lamarckian neuroevolution proceeds. (Lower is better.) Unlike the relatively smooth progress on the run shown in figure 6.5, this run using a different seed for the random number generator suffered from substantial and long-enduring deviations from monotonic improvement.

and behavior in the phase space, but further increases in the sample size generally move the solution toward the human results on both metrics, albeit still somewhat noisily. It might be conjectured that a sufficiently large number of examples would close the gap to within some arbitrarily small error on each dimension.

However, the behavioral metric shows a fairly high mean classification error rate of 14.2% even when the example sample size is 5,000 and evolution is allowed to continue for 5,000 generations. Nor could that error rate be substantially reduced by extended training on the full set of 11,000 training examples with backpropagation alone. Moreover, on some runs the improvements on that metric diverge considerably from the expected noisy approximation to monotonicity (figure 6.9; cf. the more expected pattern of progress shown in figure 6.5).

There are several reasons for the difficulty of completely eliminating the classification errors in the similarity metric. A count of overt errors in play resulting from habituation indicated that about 1% of the human-generated examples were not actually correct according to the L_0 doctrine. Such errors not only introduce contradictions that make learning more difficult; they also virtually guarantee a classification error whenever encountered in the test set.

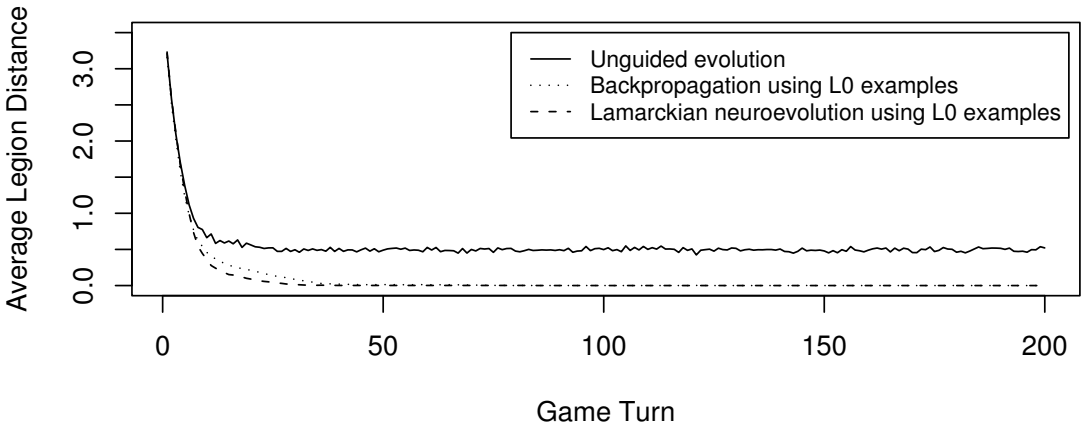
However, that 1% only accounts for a small fraction of the 14.2% error rate. Additional difficulties arise from the nature of the discrete tiling of the map with hexagons. When moving between two points by jumping along adjacent tiles on the map there is an obvious and unique shortest path only when those points are aligned on the grain of the tiling. But when the end points are not aligned with the grain there can be several paths of equal length by the Manhattan distance, and the human play was not constrained to follow any specific doctrine for choosing between the equally efficient paths. When a zig-zag path is taken between the two points, the sequences LRLRL... and RLRLR... may be equally appropriate, and if the controller chooses one during the example classifications when in fact the human chose the other, the entire sequence of classifications will be counted as erroneous.

Another factor, and a very important consideration when generating training examples for embedded game agents, is that the human player's view of the game state differed from the legions' views, and the transformation from the one to the other when recording the examples may have introduced aliases that left the controllers incapable of distinguishing why one move is appropriate in one instance of the alias while another move is more appropriate in another instance. For that reason, a deliberate attempt was made to avoid making decisions on the basis of privileged knowledge during the creation of the examples. For example, as the legions have no sense of the game time, the human player did not use end-game solutions that required knowledge of the number of turns remaining. However, due to the nature of human cognition it is almost certain that the human player used intermediate-term plans during play, such as "go over there and break up that crowd of barbarians", whereas the implementation if the game restricts the legions to decisions that are local in time as well as space; higher-level behavior arises strictly from the concatenation of sequences of localized decisions. The examples did not record any plans or intentions, and also reduced the human's full knowledge of the game state to the fuzzy view provided by the legions' egocentric sensors.

Thus while the similarity metric may be useful for quantifying behavior at a coarse scale – e.g. to detect the large gap in behavioral similarity between the use of too few vs. sufficiently many examples – more precise measurements of the details of behavior are also desirable. In the next section more detailed metrics are used to measure conformity to the specific rules that define the L_0 doctrine. Since those rules are mostly specified in terms of local conditions, the bigger problems with the behavioral similarity metric are avoided.

6.3.2 Acquisition of Rules of Behavior

A movie of the *Legion II* game being played by legions controlled by networks trained with Lamarckian neuroevolution using the L_0 examples is available on line at the *Legion II* research archive, `http://nn.cs.utexas.edu/keyword?ATA`. The movie shows that the legions learned to behave as an Adaptive Team of Agents by organizing themselves to garrison the three cities and use the two remaining legions as rovers. It also shows that the legions behave according to the L_0 doc-



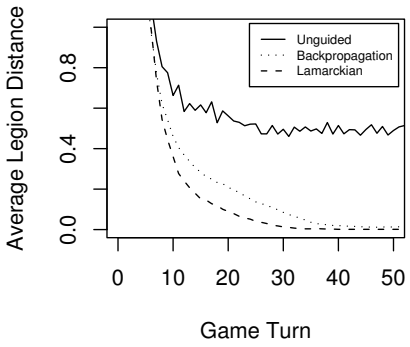


Figure 6.10: **L_0 behavior metric – garrison distance.** *Above:* Evolution guided by the L_0 examples results in a controller that locks the garrisons into their respective cities, resulting in an average distance from city to garrison of zero. The oscillating behavior of the legions trained by unguided evolution is completely suppressed. Backpropagation without evolution produces a similar result, but produces legions that are slower to conform to the doctrine. (See detail at right.)

trine with few errors; in particular, the garrisons' "mindless" oscillations in and out of their cities has been suppressed. In this section that qualitative assessment is supported by quantitative metrics. The analysis is based on the solutions obtained with the 5,000-example sample size.

In chapter 5 the distance from a city to the nearest legion was used to show that legions did in fact approach the cities and remain nearby after the ATA division of labor. For the solution obtained by unguided evolution the in-and-out oscillations caused the average distance to hover a bit over 0.5. However, if the correct division of labor is made and the garrisons observe the constraints of the L_0 doctrine, the average distance should be zero after the division of labor is made and the garrisons have had time to reach their respective cities. Therefore the distance measurements of chapter 5 were repeated using legions controlled by a network trained by Lamarckian neuroevolution using the L_0 examples for adaptation. The results are shown in figure 6.10, along with the original measurements from the unguided solution for comparison. The plot shows that the distance, averaged over the three cities, is brought down to zero early in the game and remains uniformly zero thereafter, as required by the L_0 doctrine. Backpropagation alone produces a similar result, but is slower to bring the average down to zero, as shown in the detail of the plot.

Violations of doctrine L_i can be reckoned as the number of times a city has no legion within

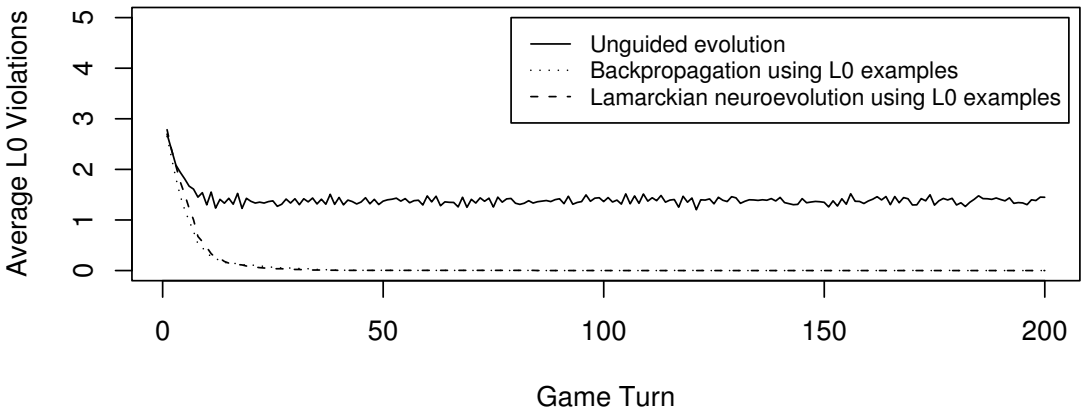
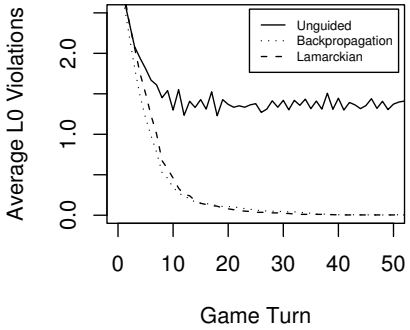


Figure 6.11: **L_0 behavior metric – doctrine violations.** *Above:* The L_0 doctrine is violated whenever there is no garrison actually in a city. The plot shows the average number of violations per turn; as expected for a controller that has learned the L_0 doctrine, the rate is brought down to zero after the garrisons have time to reach their cities. Backpropagation without evolution produces a similar result, and in fact performs slightly better than example-guided evolution by this metric, early in the game. (See detail at right.)

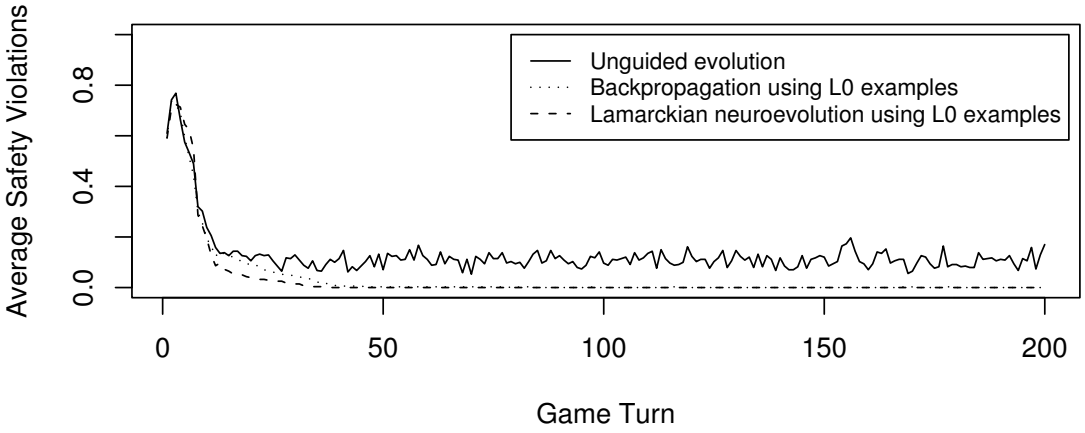


distance i. The average number of violations per turn for the L_0 doctrine is shown in figure 6.11. The L_0 result is similar to the plot of the average distance metric in figure 6.10, since both should be uniformly zero after the garrisons have moved into position. The successful acquisition of the L_0 doctrine does in fact drive both to zero: the plots differ only in the units for the y-axis.

The average number of L_0 violations accrued by the unguided solutions is shown for comparison. It hovers around 1.5 because the in-and-out oscillations mean that, on average, half of the three cities are unoccupied at any given time. Learning the L_0 doctrine requires suppression of the oscillating behavior, after which both the average distance and the L_0 doctrine violation count drop to zero. Backpropagation produces a very similar result, bringing the violation count down toward zero slightly faster during the first ten turns of the game.

Violations of the safety condition can be counted as well. Figure 6.12 shows the average number of safety condition violations per turn. For garrisons conforming to the L_0 doctrine, safety violations are not possible after the garrisons have reached their cities, so the rate becomes trivially zero as well.

The average number of safety violations accrued by the controllers trained by unguided evolution is shown for comparison. It is non-zero, but still low because violations only occur when



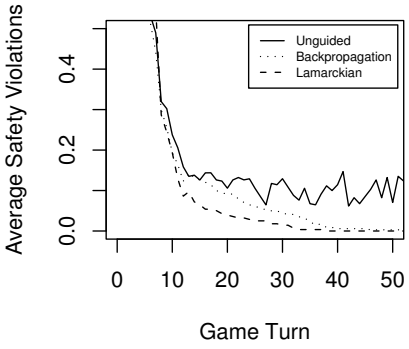


Figure 6.12: **L_0 behavior metric – safety condition violations.** *Above:* The average number of safety condition violations, as defined in section 6.2, is plotted vs. game time. Safety violations are impossible when a garrison remains in its city, so successful learning of the L_0 doctrine drives the count to zero. There is a slight rise at the beginning of the game because the randomly placed legions are still far from the cities and some of the new randomly placed barbarians enter at points nearer to the cities. The effect is eliminated as the garrisons approach their cities. Backpropagation without evolution produces a similar result, but the resulting legions are slower to eliminate the violations by garrisoning the cities. (See detail at right.)

there is a barbarian nearby when a garrison leaves a city. The garrisons leave their cities about half the time, but there is not always a barbarian nearby to threaten immediate pillage. Simple backpropagation produces results similar to the guided evolution, though somewhat slower to bring the violation rate down to zero, as shown in the detail of the plot.

For the legions trained by example-guided evolution using the L_0 training examples, the three metrics each reflect the consequences of learning to suppress the "mindless" oscillating behavior. The metrics confirm the qualitative assessment obtained by watching the movie: the method successfully learns to suppress the oscillating behavior, and since the legions also still learn the proper division of labor, ensuring that all the cities are garrisoned, the metrics are driven to the values expected for the L_0 behavior as a trivial consequence.

The next section analyses the results obtained by using the L_1 training examples. The L_1 behavior is more sophisticated, resulting in improved game scores, and the values for the metrics no longer follow trivially.

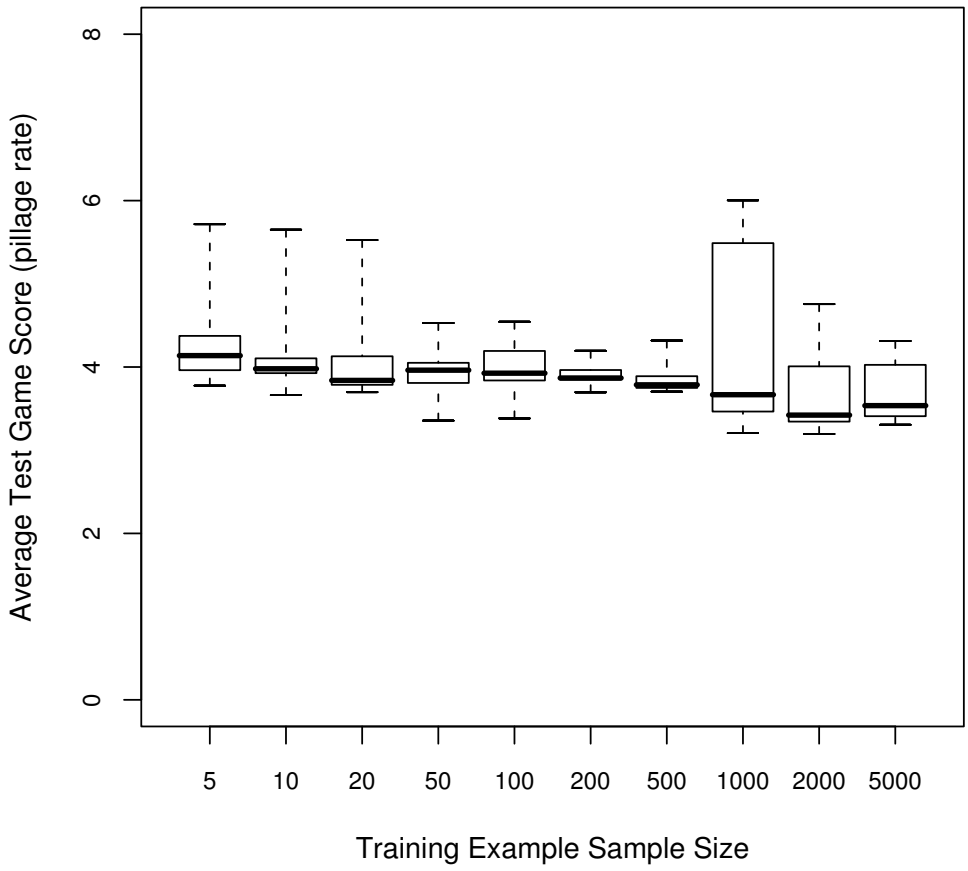


Figure 6.13: **Game performance vs. L_1 training examples used per generation.** In contrast to the use of the L_0 training examples, there was no great discontinuity in the game scores as the example sample size increased. (Cf. figure 6.6.)

6.4 Results Using the L_1 Training Examples

6.4.1 General Results

The learning experiments described in section 6.3 were repeated using the L_1 training examples. The results, as measured by game scores, are shown in figure 6.13. In contrast with the experiments using the L_0 examples, there is no sudden discontinuity in the scores between the smaller and larger example sample sizes, though there is a slight improvement in the scores when the larger sample sizes are used.

When analyzed in terms of the coarse behavior metric there is no discontinuity, though there is a strong shift in the measured value as the number of examples increases from 200 to 1000, and slow continued improvement with increasing sample size thereafter (figure 6.14).

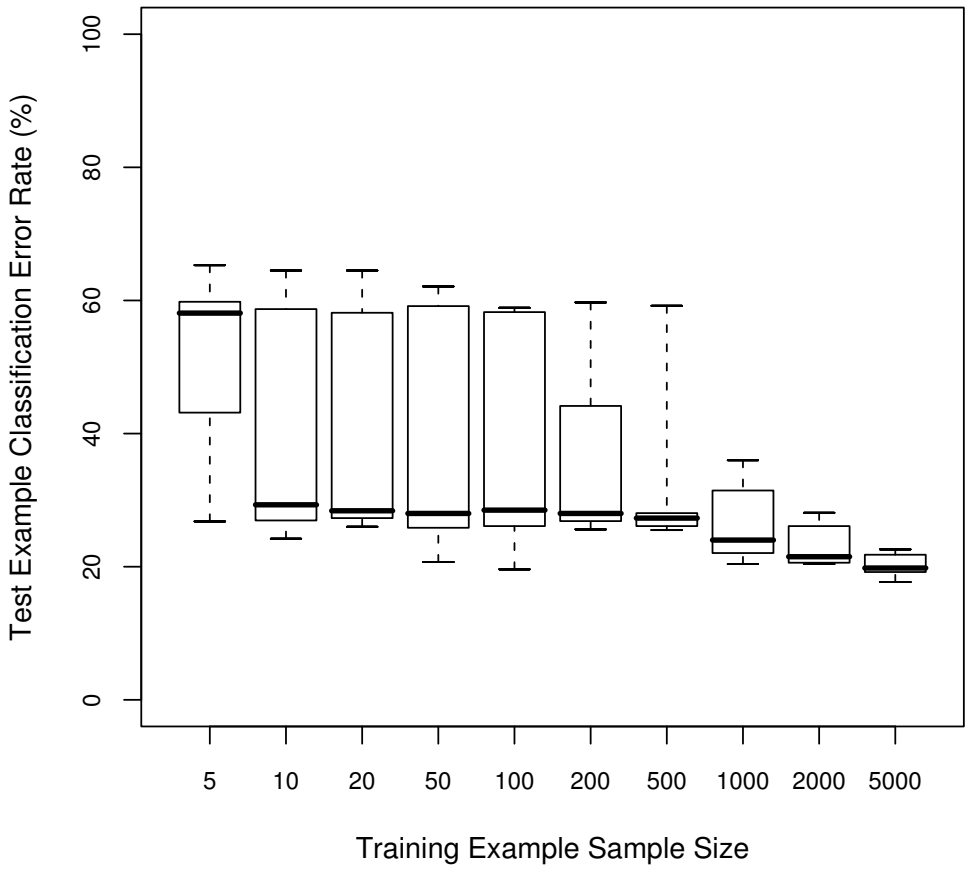


Figure 6.14: **Behavioral similarity vs. L_1 training examples used per generation.** As with training on the L_0 examples, there is a substantial discontinuity in the metric between the use of several hundred or more training examples per generation, vs. the use of fewer examples. (Cf. figure 6.7.)

Thus on the phase diagram there is no longer a distinctive gap between the points plotted for the lower vs. higher sample sizes; increasing the example sample size results in an almost linear progression in the general direction of the point plotted for human play (figure 6.15; cf. figure 6.8). The point plotted for simple backpropagation lies well to the side of that progression, due to its poor performance on the game score metric.

The weakened distinction between the use of an insufficient number of training examples per generation (A-E on the phase plot) and a sufficient number (F-J) is not entirely a matter of reduced learning effectiveness. In comparison to the L_0 doctrine, the deliberate in-and-out behavior of garrisons operating under the L_1 doctrine is more similar to the behavior acquired by unguided evolution. Thus the unguided solutions show only a mean 54.8% classification error rate on the L_1 test set, as opposed to the 77.0% error rate on the L_0 test set. In both cases the group of similar solutions obtained when too few training examples are used are spread over a range of about 14% on

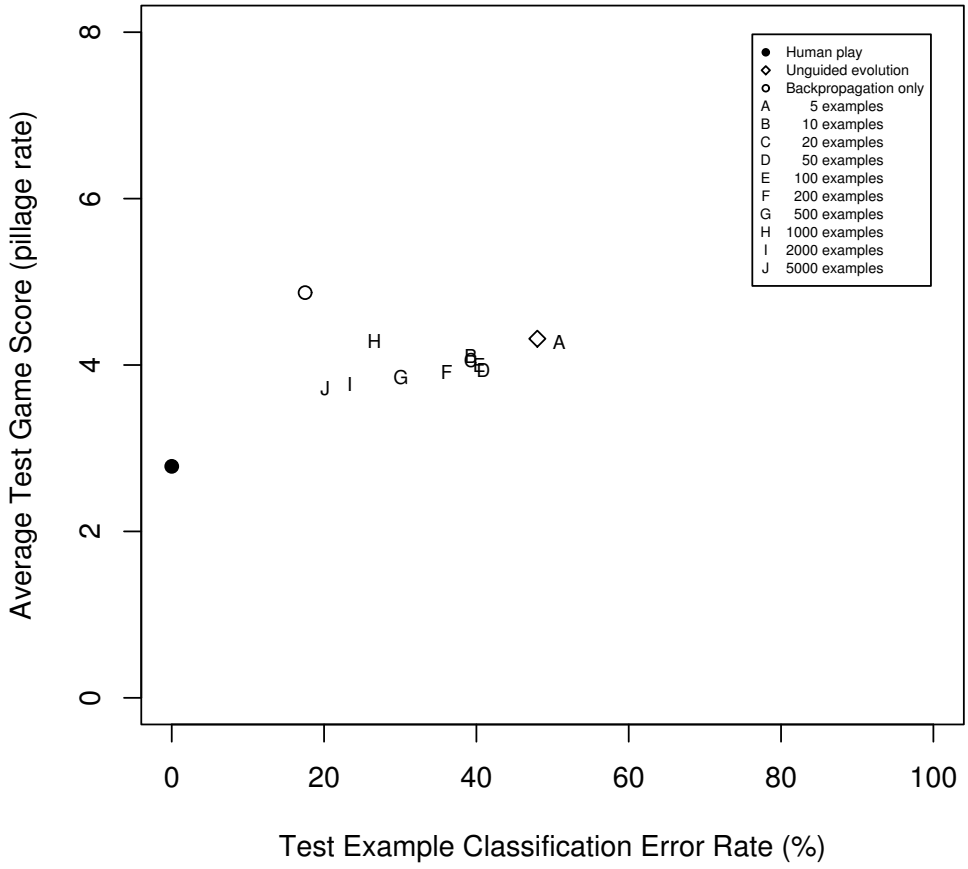


Figure 6.15: **Performance-behavior phase diagram for L_1 examples.** As for evolution guided by the L_0 examples, solutions cluster in the phase space according to whether they use a clearly insufficient number of training examples per generation (A-E) vs. a greater number (F-J). In contrast to the L_0 solutions, greater fidelity brings improved game scores, since the L_1 doctrine is in fact a superior strategy. (Cf. figure 6.8.) The result of using backpropagation without evolution is shown for comparison; it performs slightly better than the 5000-example Lamarckian method (J) on the behavior metric, but somewhat worse on the game scores.

the behavioral metric, ranging downward from a point near the rate obtained by unguided evolution, so there is a general leftward shift of those points on the phase diagram.

However, when sufficient examples are used the behavioral similarity error metric is only reduced to the range 20-27%, rather than the 14-19% obtained when comparing the L_0 learners with the L_0 test examples. But the general trend with increasing examples is in the right direction, and it should be noted that the similarity metrics are made with comparisons to different test sets. Moreover, the arguments for the coarse nature of the similarity metric stated in section 6.3.1 still apply, so the acquisition of L_1 behavior will be examined by the more detailed behavioral metrics in the next section.

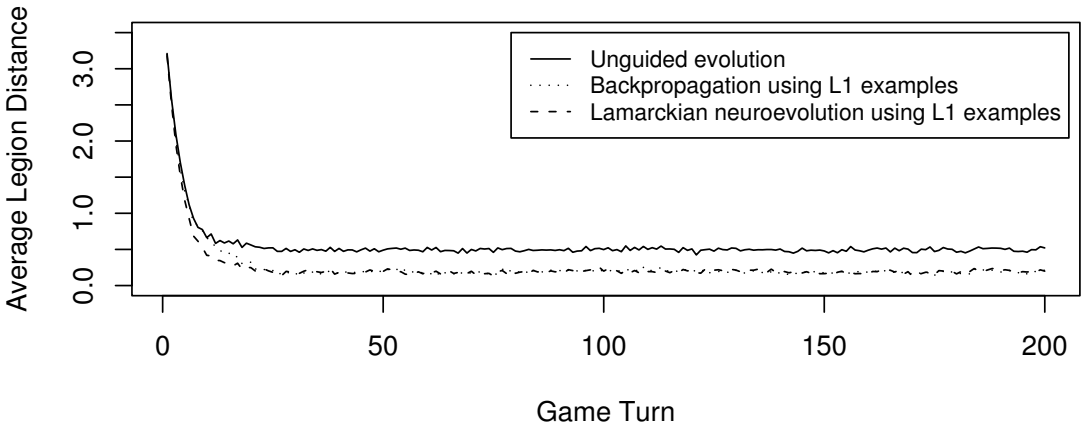
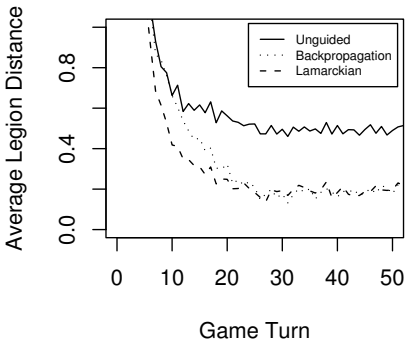


Figure 6.16: **L_1 behavior metric – garrison distance.** *Above:* The L_1 behavior reduces the average distance from cities to their garrisons due to the suppression of the unmotivated in-and-out oscillations, but does not reduce it to zero because the L_1 doctrine still allows the garrisons to leave the cities under appropriate circumstances. Backpropagation without evolution produces a similar result, but produces legions that are slower to conform to the doctrine. (See detail at right.)

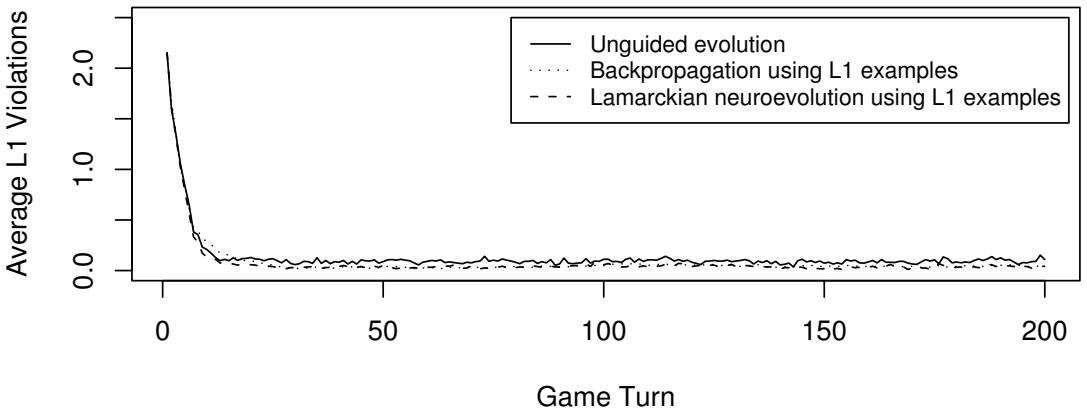


Meanwhile, the similarities between the L_0 and L_1 learning results suggest a conclusion that learning suffers when an insufficient number of training examples is used, and that even after a sufficient number is used to cause a jump from an area in the phase space near the results of unguided evolution to an area nearer the target behavior, further increases in the number of examples used tend to improve performance on the similarity metric.

6.4.2 Acquisition of Rules of Behavior

The controllers trained by Lamarckian neuroevolution guided by backpropagation on the L_1 training examples were tested on the detailed behavior metrics used for the L_0 training in section 6.3.2, and the results are given in this section. The plots are based on the solutions obtained with the 5,000-example sample size.

The results of the distance-to-garrison metric are shown in figure 6.16. Unlike the L_0 doctrine, the L_1 doctrine is not expected to reduce the distance to zero, because the garrisons are allowed to leave their cities under certain circumstances. The plot shows that the average distance is in fact reduced well below the average for controllers trained by unguided evolution, from somewhat above 0.5 to around 0.2, after the division of labor and garrisoning of the cities. The number is reduced



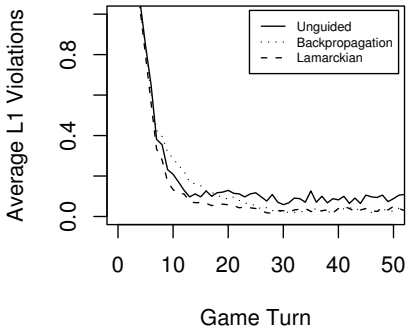


Figure 6.17: **L_1 behavior metric – doctrine violations.** *Above:* Most of the L_1 doctrine violations in the unguided solutions are eliminated by evolution guided by the L_1 examples. Ideally they should be eliminated altogether; they are in fact reduced to a very low rate. Backpropagation without evolution produces a similar result, but produces legions that are slower to conform to the doctrine. (See detail at right.)

because the L_1 doctrine only allows the garrisons to leave their cities to eliminate a barbarian, so much of the in-and-out of the unguided solution is eliminated. Simple backpropagation provides results very similar to example-guided evolution, though as shown in the detail of the plot, it is somewhat slower at bringing the metric down.

The rate of L_1 doctrine violations is shown in figure 6.17. L_1 doctrine violations occur whenever no legion is in or adjacent to a city. The rate is fairly low even for unguided evolution (solid line), since in those solutions the garrisons oscillate in and out of their cities but rarely move further away than an adjacent map cell. Evolution guided by the L_1 training examples reduces the violation rate still further. Though violations are not completely eliminated, after the division of labor is accomplished and the garrisons have had time to reach their cities the average rate never rises above 0.09 violations per turn (0.03 violations per city per turn). Simple backpropagation again produces similar results, though it is also again a bit slower to pull the violation count down, as shown by the detail of the plot.

The rate of safety condition violations for the controllers trained on the L_1 examples is shown in figure 6.18. Evolution guided by the L_1 training examples reduces safety violations in comparison to the results of unguided evolution because the legions learn not to leave their cities

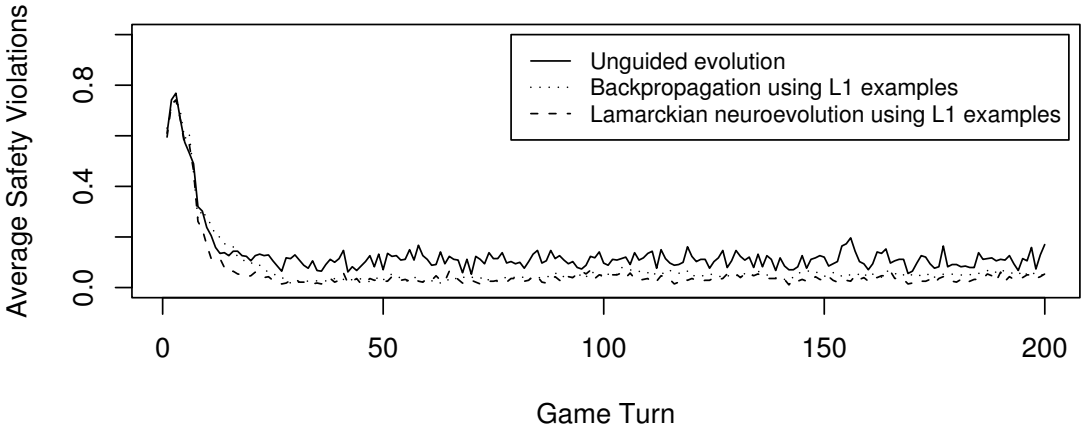
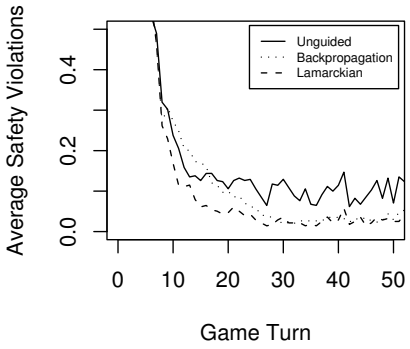


Figure 6.18: **L_1 behavior metric – safety condition violations.** *Above:* Solutions obtained by guidance with the L_1 training examples reduce the safety condition violations considerably in comparison to the solutions obtained by unguided evolution. Ideally they should be eliminated altogether; they are in fact reduced to a very low rate. Backpropagation without evolution produces a similar result, but again produces legions that are slower to eliminate the violations. (See detail at right.)



in the presence of certain barbarian threats. After the initial division of labor and garrisoning of the cities the error rate never exceeds 0.09 violations per turn (0.03 violations per city per turn). Backpropagation again produces controllers that are slower to pull the violation count down.

A movie of the *Legion II* game being played by legions controlled by networks trained with evolution guided by the L_1 examples is also available on line at the *Legion II* research archive, `http://nn.cs.utexas.edu/keyword?ATA`. Careful examination of the movie shows good conformance to the L_1 doctrine and the restrictions of the safety condition, e.g. garrisons can be seen to remain locked into a city when there is more than one adjacent barbarian. (See section 6.2 for the effect of combining the L_1 doctrine with the safety condition.) Significantly, such refined details of behavior are only implicit in the training examples, but are acquired by the controllers trained with example-guided evolution.

6.5 Findings

The experiments demonstrated that Lamarckian neuroevolution using backpropagation for adaptation is a feasible and effective mechanism for example-guided evolution. The use of human-

generated examples guided evolution to solutions that behave visibly and quantitatively more like the human player than the solutions produced by unguided evolution did. The guidance was effective to the extent of allowing the agents to learn rules of behavior that were only implicit in the examples, and even the behavior required by a combination of the rules, which was only implicit in the rules themselves.

The use of examples allowed suppression of undesirable behavior in the trained agents, such as the "mindless" oscillating in and out of the cities by the garrisons trained by unguided evolution. The use of different classes of examples allowed suppressing the undesired behavior in two different ways: in one case the movement was suppressed altogether, but in the other it was suppressed only in cases where there is no visible motivation.

The use of the different classes of examples also allowed the creation of agents that behave with greater or lesser effectiveness when evaluated in terms of game scores. Importantly, the agents obtaining the lower scores did not do so by abandoning visibly intelligent behavior, such as reverting to the "mindless" oscillations or adopting other erratic traits; rather, they simply follow a less effective strategy, and their behavior still makes sense to human observers while doing so.

Backpropagation, when used alone, produced behavioral results more similar to the results obtained by example-guided evolution than to those obtained by unguided evolution. Generally, after turn 50 the plots of the metrics are indistinguishable from the plots obtained for example-guided evolution, though on most of the metrics the controllers trained by backpropagation were slower to bring the metric down to the matching value. Moreover, the game scores obtained by backpropagation were worse than those obtained by guided evolution – worse in the absolute sense, and also worse on account of being farther away from the human score.

Chapter 7

Exploiting Sensor Symmetries

This chapter examines a method for exploiting the existence of symmetries in the structure of an embedded agent and its sensors. Once training examples have been introduced into the process of training agent controllers (chapter 6), symmetrical transformations of those examples can be used to create new artificial examples, and those examples can be used to improve learning. That idea is motivated and explained in detail in section 7.1. The section 7.2 reports on an experimental evaluation of the concept, and section 7.3 summarizes the findings regarding exploiting the symmetries.

7.1 Introduction

Intelligent agents in games and simulators often operate in geometric environments subject to reflections and rotations. For example, a two dimensional map can be reflected across an explorer agent or rotated about it, providing new and different but still reasonable maps. Similarly, the visible universe can be reflected or rotated on any of the three axes of a robotic construction worker in deep space. A well trained general purpose agent for deployment in such environments should be able to operate equally well in a given environment and its symmetric transformations. In general it is desirable for intelligent agents to exhibit symmetrical behavior as well. That is, if the optimal action in a given environment is to move to the left, then the optimal action in a mirror image of that environment would be to move to the right.

Symmetry of behavior is desirable for two reasons. First, if a correct or optimal move can be defined for a given context, failing to choose the symmetrical move in the symmetrical context will be sub-optimal behavior, and will degrade an agent's overall performance if it ever encounters such a context. Second, if the agent operates in an environment observable by humans, such as a game or a simulator, the humans will expect to see visibly intelligent behavior, i.e., they will expect the agent to always do the right thing because it is smart, rather than intermittently doing the right thing because it has been programmed or trained to manage only certain cases.

If an agent's controller operates by mapping sensory inputs onto behavioral responses, the

desired symmetries can be identified by analyzing the structure of the agent and its sensors. For example, if the agent and its sensors are both bilaterally symmetrical then it will be desirable for the agent's responses to be bilaterally symmetrical as well. However, if they are not symmetrical – e.g. for a construction robot with a grip on one side and a tool on the other – then its optimal behavior is asymmetrical. Thus the desired symmetry of behavior depends critically on the symmetry of the agent and its sensors.

When an agent's controller is directly programmed it is a straightforward task to ensure that its behavior observes the desired symmetries. However, when a controller is trained by machine learning there is in general no guarantee that it will learn symmetrical behavior. Thus the *Legion II* controllers must learn their behavior with respect to each cardinal direction independently. Therefore it is useful to devise machine learning methods that encourage behavioral invariants across relevant symmetries in agents trained by those methods.

The *Legion II* game is a useful platform for testing the use of training examples generated from symmetries, because it is a challenging learning task that offers multiple symmetries in the structure of the learning agents, their sensors, and their environment. This chapter reports initial results toward solving the symmetry challenge with supervised learning. A controller is trained for the legions, which operate in an environment that supports multiple symmetries of reflection and rotation. Human-generated examples of appropriate contextual behavior are used for the training, and artificial examples are generated from the human examples to expose the learner to symmetrical contexts and the appropriate symmetrical moves. This training mechanism addresses both of the motivations for symmetrical behavioral invariants, i.e. it improves the agents' task performance and also provides measurable improvements in the symmetry of their behavior with respect to their environment, even in environments not seen during training.

The methods used to exploit symmetries experimentally are described in the following section, then the experimental results are explained in section 7.2. Findings are summarized in section 7.3.

7.2 Experimental Evaluation

7.2.1 Methods

Experiments were designed to test two hypotheses: first, that exploiting symmetric transformations can broaden the range of experience made available to the agents during training, and thus result in improved performance at their task; and second, that exploiting symmetric transformations during training can make the agents' response to environments not seen during training measurably more consistent. These hypotheses were tested by training the legions' controller networks with human-generated examples, with or without supplementary examples created by reflecting and/or rotating them, and then applying appropriate metrics to the trained legions' performance and behavior during runs against the test-set games.

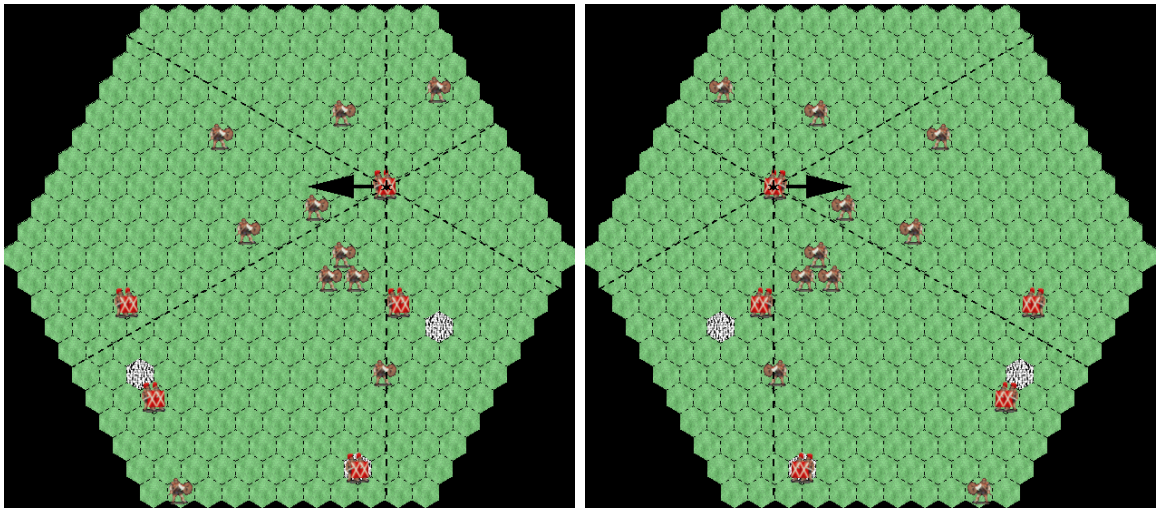


Figure 7.1: **Sensor symmetries in the *Legion II* game..** The bilateral and radial symmetries of the legions' sensors and the map grid make several invariant transformations possible. For example, if a legion senses a certain game state and chooses to move west, then if it ever senses a game state that is an east-to-west reflection of that state, it should choose to move east.

The *Legion II* sensor architecture allows reflections and rotations of the world about a legion's egocentric viewpoint (figure 7.1). The transformations can be represented by permutations of the values in the sensors. For example, a north-south reflection can be implemented by swapping the northwest (NW) sensor values with the southwest (SW), and the NE with the SE. Similarly, a 60° clockwise rotation can be implemented by moving the sensor values for the eastern (E) sector to the southeastern (SE) sensor, for the SE to the SW, etc., all the way around the legion. The legions' choices of action for a reflected or rotated sensory input can be reflected or rotated by the same sort of swapping. For example, a 60° clockwise rotation would convert the choice of a NE move to an E move. The option to remain stationary is not affected by reflections or rotations: if a legion correctly chooses to remain stationary with a given sensory input, it should also remain stationary for any reflection or rotation of that input.

The examples of human play using the L_0 doctrine that were generated for the example-guided evolution experiments (chapter 6) were re-used for the symmetry experiments. The legions are still expected to behave as an Adaptive Team of Agents, so they are still coerced to have identical control policies (chapter 5). Such uniform control means that all the examples recorded for the various legions during play can be pooled into a single set for training a controller network: what is correct for one legion is correct for all of them.

The supplementary artificial training examples were created by the training program at run time, by permuting the fields in the human-generated examples as described above. Since the legions in *Legion II* have no distinguished orientation, all the reflections were generated by flipping the

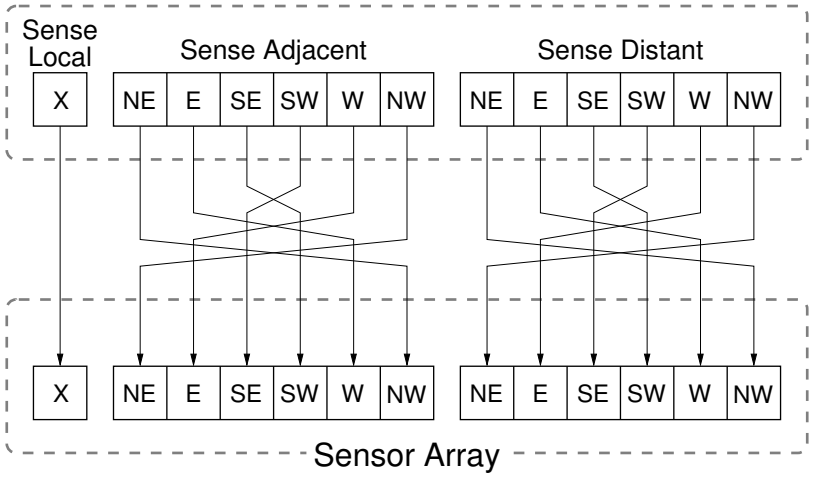


Figure 7.2: **Generating artificial examples with sensor permutations.** A reflection or rotation of the environment about a legion has the same effect as a permutation of its sensor inputs. Thus permutations provide a simple mechanism for creating artificial examples that are as correct as the originals.

sensory input and choice of move from north to south. When both reflections and rotations were used, the N-S reflection was applied to each rotation, to expand each original to a full set of twelve isomorphic examples.

The four possibilities of using vs. not using reflections and/or rotations define four sets of training examples. The choice between these sets defines four training methods for comparison. The four methods were used to train the standard *Legion II* controller network with backpropagation (Rumelhart et al. 1986a) on the examples. Training was repeated with from one to twelve games' worth of examples for each method. Since gameplay was used for the validation and test sets, all of the human-generated examples were available for use in the training set. On-line backpropagation was applied for 20,000 iterations over the training set, to ensure that none of the networks were undertrained, and the presentation order of the examples was reshuffled between each iteration. An evaluation against the validation set was done every ten iterations. Due to the relatively large number of examples available, the learning rate η was set to the relatively low value of 0.001.

Each differently parameterized training regime – method $\times$ number of games' examples used – was repeated 31 times with a different seeds for the random number generator, producing a set of 31 networks trained by each parameterization. The 31 independent runs satisfy the requirement of a sample size of at least 30 when using parametric statistical significance tests (Pagano 1986), so statistical significance results are reported in this chapter.

The next two sections examine the effects of generated examples on learning, per the two hypotheses set forward in section 7.1.

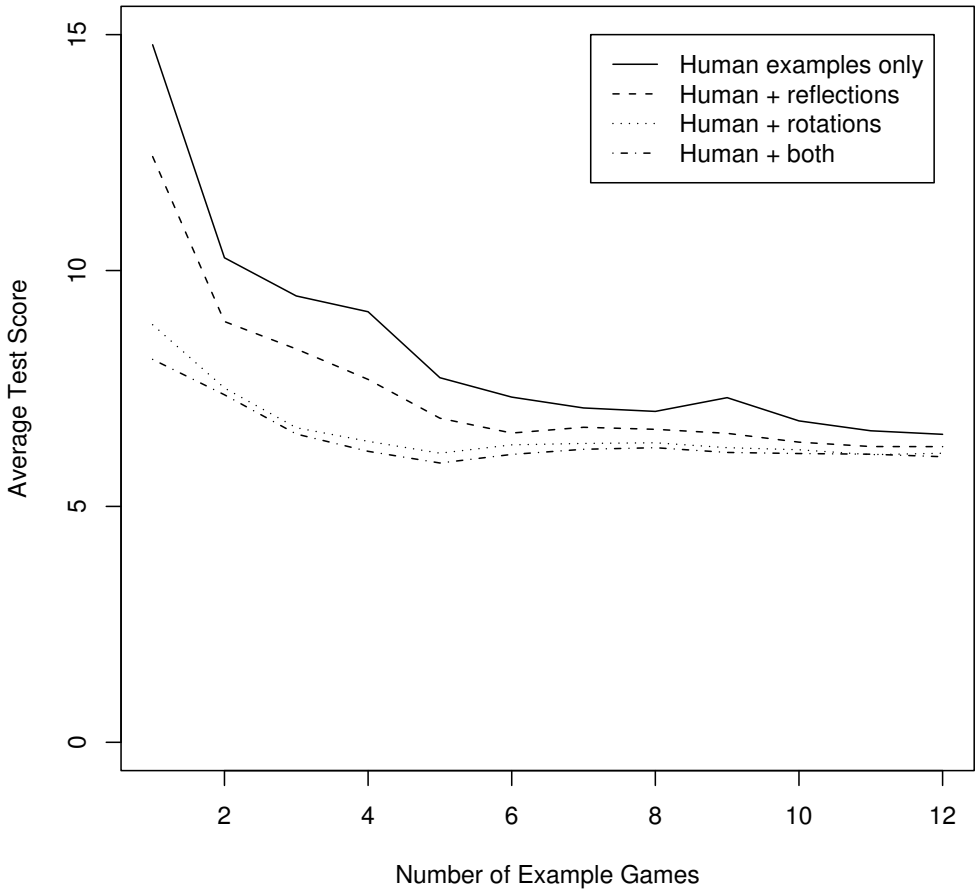


Figure 7.3: **Effect of generated examples on performance.** Lines show the average test score for 31 runs of each method vs. the number of example games used for training. (Lower scores are better.) Each game provided 1,000 human-generated examples; reflections increased the number of examples to 2,000 per game, rotations to 6,000, and both together to 12,000. All three symmetry-exploiting methods provided significant improvement over the base method throughout the range of available examples, albeit by a diminishing amount as more human examples were made available.

7.2.2 Effect on Performance

The first experiment illustrates the effect of adding artificially generated training examples on the performance of the controller networks. Networks were trained by each of the four methods, on from one to twelve games' worth of examples. As described in chapter 6, each game provided 1,000 human-generated examples; however, he reflections and rotations greatly increased that number.

The results of the experiment, summarized in figure 7.3, show that an increase in the number of example games generally improved learning when the human-generated examples alone were used for training, although without strict monotonicity and with decreasing returns as more games were used. The three methods using the artificially generated examples improved learning over the

use of human examples alone, regardless of the number of games used; each of the three provided statistically significant improvement at the 95% confidence level everywhere. The improvement was very substantial when only a few example games were available, and the best performance obtained anywhere was when both reflections and rotations were used with only five games' worth of examples.

Rotations alone provided almost as much improvement as reflections and rotations together (figure 7.3), and at only half the training time, since it only increased the number of examples per game to 6,000 rather than 12,000. Thus in some circumstances using rotations alone may be an optimal trade-off between performance and training time. Reflections alone increased training only to 2000 examples per game, $1/3$ of what rotations alone provided, but with substantially less improvement in performance when fewer than six example games were available (figure 7.3).

It is worthwhile to understand how much of the improved performance resulted from the increased number of training examples provided by the reflections and rotations, vs. how much resulted from the fact that the additional examples were reflections and rotations *per se*. A second experiment examined this distinction by normalizing the number of examples used by each method. For example, when a single example game was used in the first experiment, the human-example-only method had access to 1,000 examples, but the method using both reflections and rotations had access to $12 \times 1{,}000$ examples. For this second experiment the various methods were only allowed access to the same number of examples, regardless of how many could be created by reflections and rotations.

It was also necessary to control for the structural variety of the examples. Such variety arises from the fact that each training game is played with a different random set-up – most importantly, with randomized locations for the cities. In some games the cities are scattered, while in other games they are placed near one another. This sort of variety is very beneficial to generalization: the games in the test set may not be similar to any of the individual games in the human-generated training set, but agents exposed to a greater variety of set-ups during training learn to manage previously unseen situations better. Thus if the training example count is normalized to 12,000 by using the 12,000 human-generated examples from the twelve example games, to be compared against training with the 12,000 examples generated by applying reflections and rotations to the 1,000 human-generated examples from a single example game, the latter method will have less structural variety in its training examples, and its generalization will suffer.

So the second experiment controlled for both count and structural variety by selecting examples at random, without replacement, from the full set of 12,000 human-generated examples available for use. When the method of using human-generated examples alone selected n examples at random, the method using reflections selected $n/2$ examples and doubled the count by reflecting, the method using rotations selected $\lfloor n/6 \rfloor$ examples and multiplied the count by six by rotating, and the method using both reflections and rotations selected $\lfloor n/12 \rfloor$ examples and multiplied the count by twelve by reflecting and rotating. Since each method drew its randomly selected examples

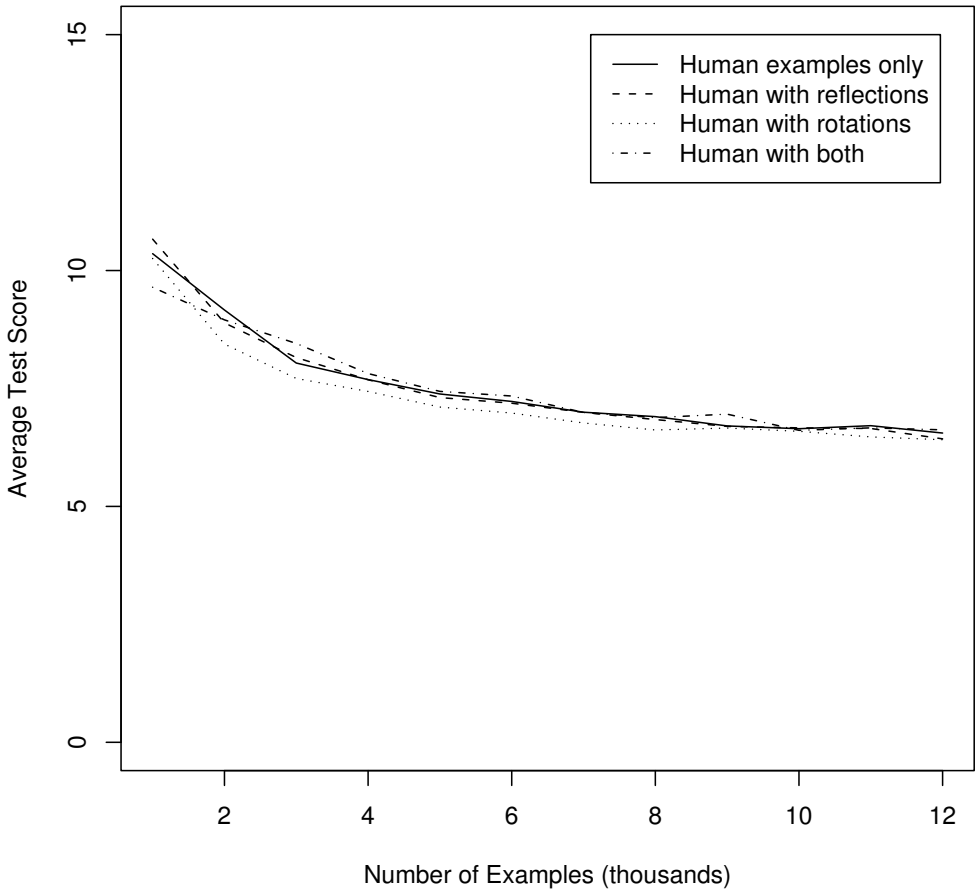


Figure 7.4: **Effect of reflections and rotations on performance.** Lines show the average test score for 31 runs of each method vs. the number of examples used for training, when the examples used by the four methods were controlled for count and structural variety. The tight clustering of the performance curves shows that most of the improvements obtained by using reflections and rotations in the first experiment (figure 7.3) were the result of the increased number of training examples they provided, rather than being the result of the use of reflections and rotations *per se*.

from the full set of the 12,000 available human-generated examples, each sampled the full structural variety of the examples. The reduced sample sizes for the methods that generate more artificial examples normalized the counts, to within rounding errors.

The results of the experiment, shown in figure 7.4, show little difference in the performance of the four methods when their training examples are controlled for count and structural variety. Thus most of the improvements obtained in the first experiment were the result of the increased number of training examples generated by the reflections and rotations, rather than by the fact that the additional examples were reflections or rotations *per se*.

7.2.3 Effect on Behavioral Consistency

Another pair of experiments revealed the effect of artificially generated examples on the detailed behavior of the legions. As described in section 7.1, a perfectly trained legion will show behavior that is invariant with respect to reflections and rotations. That is, if its sensory view of the world is reflected and/or rotated, then its response will be reflected and/or rotated the same way.

Although legion controllers trained by machine learning techniques are not guaranteed to provide perfect play, training them with reflected and/or rotated examples should make them behave more consistently with respect to reflections and rotations of their sensory input. This consistency can be measured when playing against the test set by generating reflections and rotations of the sensory patterns actually encountered during the test games, and making a side test of how the legions respond to those patterns. The responses are discarded after that test, so that they have no effect on play. For each move in a test game a count is made of how many of the twelve possible reflections and rotations result in a move that does not conform to the desired behavioral invariant. Each such failure is counted as a *consistency error*, and at the end of the test a consistency error rate can be calculated.

Since the perfect move for a legion is not actually known, the consistency errors are counted by deviation from a majority vote. That is, for each reflection and/or rotation of a sensory input, a move is obtained from the network and then un-reflected and/or un-rotated to produce an "absolute" move. The 12 absolute moves are counted as votes, and the winner of the vote is treated as the "correct" move for the current game state (Bryant 2006). When the response to the reflections and rotations does not correspond to the same reflection or rotation of the "correct" move, it is counted as a consistency error.

All of the networks already produced by the performance experiments described in the previous section were tested to examine the effect of the various training parameterizations on behavioral consistency. The results, summarized in figure 7.5, show that the three methods using reflections and rotations reduce consistency errors substantially in comparison to the base method of using only the human-generated examples, regardless of how many example games are used. In every case the improvements were statistically significant at the 95% confidence level. The best consistency was obtained when both reflections and rotations were used, and as with the performance experiment (figure 7.3), a local minimum was obtained when relatively few training games were used (six in this case). The consistency error rate for this optimal method is approximately flat thereafter.

Again, it is worthwhile to understand how much of the reduced consistency error rate resulted from the increased number of training examples provided by the reflections and rotations, vs. how much resulted from the fact that the additional examples were reflections and rotations *per se*. Thus the networks from the normalization experiment were also tested for behavioral consistency. The results, shown in figure 7.6, show the familiar trend of improvement as the number of training examples increases, but also show very substantial differences between the four methods, even when their training examples are controlled for count and structural variety. Thus much of

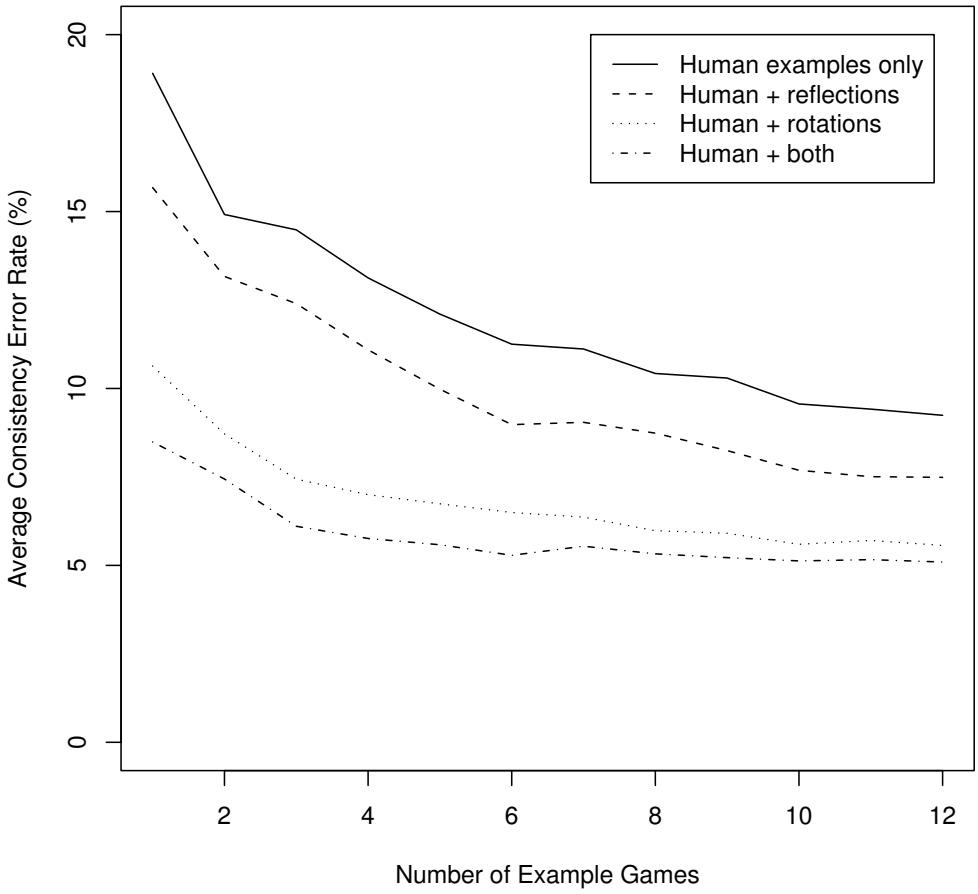


Figure 7.5: **Effect of generated examples on consistency.** Lines show the average consistency error rates for 31 runs of each method, vs. the number of example games used for training. (Lower rates are better.) All three symmetry-exploiting methods provided a significant improvement in consistency over the base method, throughout the range of available examples.

the improvement in the consistency error rate in the uncontrolled experiment (figure 7.5) can be attributed to the fact that the generated examples used reflections and/or rotations *per se*, rather than simply resulting from the increased number of training examples.

7.3 Findings

The experiments show that training intelligent agents for games and simulators can benefit from the extra examples artificially generated from reflections and rotations of available human-generated examples. In accord with the hypotheses stated in section 7.2.1, the technique results in agents that score better in the game and behave more consistently.

The improved performance scores were shown to result primarily from the increased num-

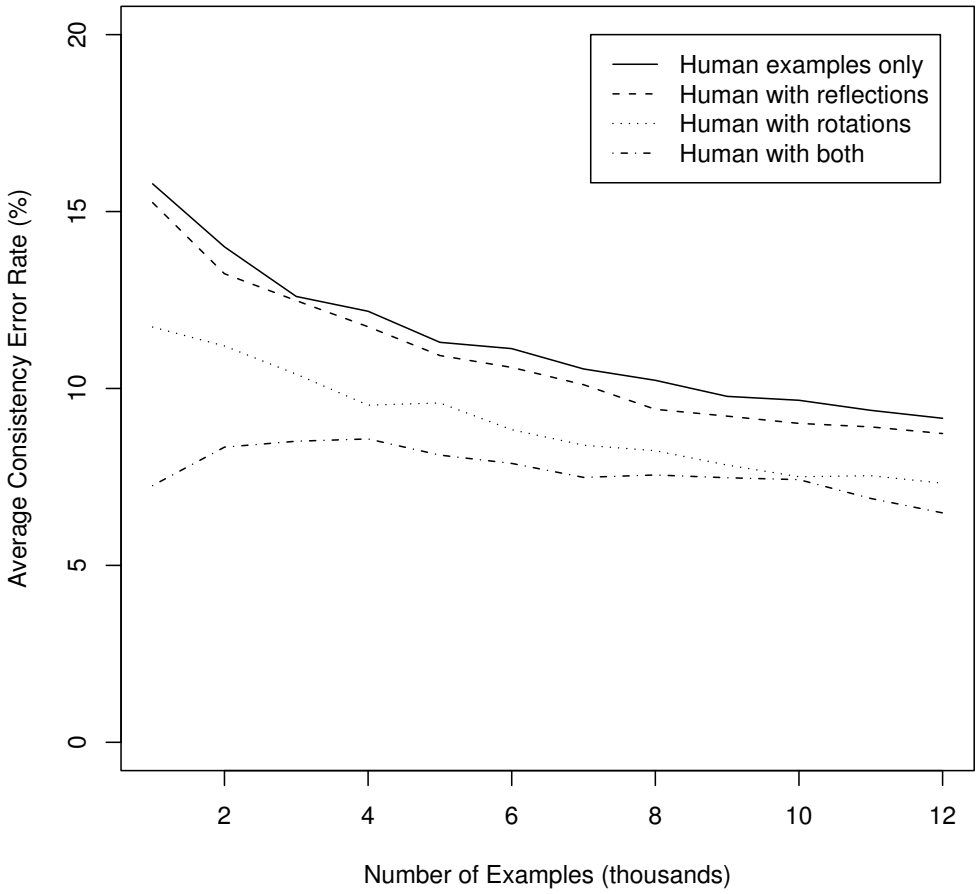


Figure 7.6: **Effect of reflections and rotations on consistency.** Lines show the average consistency error rates for 31 runs of each method vs. the number of examples used for training, when the examples used by the four methods were controlled for count and structural variety. The substantial gaps between the lines show that much of the improvement obtained by using reflections and rotations in the the third experiment (figure 7.5) were the result of the use of reflections and rotations *per se*, rather than merely being a result of the increased number of training examples they provided.

ber of training examples provided by the reflections and rotations, but the improved behavioral consistency resulted largely from the fact that those examples were reflections and rotations *per se*. In principle, reflections and rotations *per se* could improve performance scores as well, by providing a more systematic coverage of the input space. However, in *Legion II* , the base method already provided over 80% consistency with respect to reflections and rotations when only a single game's examples were used for training (figure 7.5). The agents quickly learned the symmetries necessary for the situations that had the most impact on their game scores. Further improvements in behavioral consistency provided polish, but had no substantial impact on the scores. In other domains the base coverage may be more idiosyncratic, so that reflections and rotations *per se* would significantly

improve performance.

The combination of both reflections and rotations provided the best results throughout. That method obtained its best performance and consistency when relatively few example games were made available for training, five and six games respectively. This is a very promising result, because it suggests that good training can be obtained without excessive human effort at generating examples. Rapid learning from relatively few examples will be important for training agents in a Machine Learning Game (Stanley et al. 2005c) if the player trains game agents by example during the course of the game, and for simulations where agents must be re-trained to adapt to changed environments, doctrines, or opponent strategies. Future work will thus investigate whether rapid learning from relatively few examples is seen in other applications, and also whether a greatly increased number of human-generated examples will ever converge to the same optimum provided by the artificial examples in these experiments.

The number of examples that can be generated from symmetries depends critically on the sensor geometry of the agent being trained. The number of radial sensors may vary with the application and implementation, providing a greater or lesser number of rotations. For example, the *Cellz* game (Lucas 2004) uses an eight-way radial symmetry. But if radial sensors do not all encompass equal arcs then rotations may not be possible at all. For example, the agents in the NERO video game (Stanley et al. 2005c) also use "pie slice" sensors, but with narrower arcs to the front than to the rear, in order to improve their frontal resolution. There is therefore no suitable invariant for the rotation of the NERO agents' sensors.

However, the NERO agents, and probably most mechanical robots as well, have a bilateral symmetry that allows applying the behavioral invariant for reflections across their longitudinal axis. The results presented in this chapter show that artificially generated examples provide significant training benefits even when only reflections are used, especially when relatively few human-generated examples are available (figures 7.3 and 7.5). Thus the methods examined here should prove useful even in situations with far more restrictive symmetries than in the *Legion II* game. On the other hand, agents operating in three-dimensional environments, such as under water or in outer space, may have a greater number of symmetries to be exploited, offering even greater advantage for these methods. Future work should also investigate the effect of exploiting symmetries in boardgames with symmetrical boards, such as *Go*, where an external player-agent manipulates passive playing pieces on the board.

Supervised learning is not always the best way to train agents for environments such as the *Legion II* game; the methods described in previous chapters produced controller networks that perform somewhat better than those produced by backpropagation in these experiments. However, reflections and rotations are feasible when example-guided evolution is used, and work in progress will evaluate the utility of doing so.

Chapter 8

Managed Randomization of Behavior

This chapter introduces a method for introducing a controlled amount of randomness into an autonomous agent's behavior, with a modified training mechanism that allows a bias to the randomness so that the negative effect of randomness on task performance is minimized. Section 8.1 motivates and explains the ideas. Then section 8.2 reports their experimental evaluation. Finally, section 8.3 summarizes the findings regarding managed randomization of behavior.

8.1 Introduction

A very common problem with the behavior of computerized game opponents – the so-called game AI – is that they are brittle because they are too predictable. With repeated play, players learn to predict the behavior of the AI and take advantage of it. Such predictability takes much of the fun out of the game; it becomes, as the saying goes, too much like shooting fish in a barrel.

A similar concern arises with simulations that are used for training humans, or as interactive tools for investigating phenomena such as traffic or water management. If the AI-controlled agents in such systems are fully predictable, the human trainees or investigators may simply learn to beat the system rather than solve the intended problem. It is therefore desirable to have controllers for embedded intelligent agents that allow them to behave stochastically, yet still intelligently, in both games and simulators.

Neuroevolution with fitness determined by game play has been found useful in training artificial neural networks as agent controllers in strategy games and simulators (Bryant and Miikkulainen 2003; Lucas 2004; Stanley and Miikkulainen 2004a; Stanley et al. 2005c). The designs of these systems provide egocentric embedded agents, that is, agents that decide on their own actions based on their own localized knowledge, rather than being moved around as passive game objects by a simulated player. For egocentric agent behavior, an artificial neural network can be used to map the agent's sensory inputs onto a set of controller outputs, which are interpreted by the game engine or simulator as the agent's choice of action for the current time step.

Figure 8.1: **Output activations as confidence values.** In these screenshots, a network's output activations are shown graphically, with a vertical white bar showing each neuron's activation value on a scale of $[0, 1]$ and a symbolic code for the action-unit encoded interpretation beneath. *Left:* An output activation pattern plausibly interpretable as confidence values, with a high confidence for output option W, substantially less confidence for option SW, and no discernible level of confidence for any of the other options. *Right:* An output activation pattern only dubiously interpretable as confidence values, since over half the options are fully or near-fully activated. The network that produced these outputs was trained with deterministic winner-take-all decoding, and did not develop activation behaviors suitable for interpretation as confidence values.

For discrete-state games and simulators the choice between the actions available to an agent can be implemented in an artificial neural network with *action unit coding*. That is, with a localist encoding that associates each of the network's outputs with a choice of one of the possible discrete actions (figure 8.1). Whenever a decision must be made, the agent's sensory inputs are propagated through the network and the output unit with the highest resulting activation is taken to be that agent's decision for which action to take. This deterministic winner-take-all decoding of the network's outputs results in fully deterministic behavior for the agent.

The question then arises, can such a system be modified to provide stochastic behavior in an agent, without degrading its performance excessively? A simple and intuitive solution is to interpret the network's outputs stochastically, treating the relative activation levels of the various outputs as confidence values, i.e. indications of how strongly the network evaluates the corresponding action as being appropriate for the current context (Renals and Morgan 1992). If an action is chosen with confidence-weighted decoding, i.e. with probability proportional to the corresponding action unit's activation level, the network will provide stochastic behavior and the stochastic choices will be biased toward the optimal, potentially minimizing any degradation of the agent's performance.

However, it turns out that evolved networks do not automatically learn to represent confidence values in their outputs. When training with the commonly used winner-take-all decoding method described above, there is no penalty for secondary activations at inappropriate action units, so evolution does not learn to suppress them. The network learns only what is useful in solving the problem, and thus its behavior is constrained only by the specific algorithm used to detect the peak output (figure 8.1). As a result, decoding the network's activations stochastically, as if they were confidence levels, severely degrades the agent's performance (figure 8.2).

In this chapter a method is proposed for evolving networks to produce *utility confidence values* in their outputs. That is, networks learn to produce patterns in their outputs such that the relative activation levels of the various action units reflect the networks' estimates of the relative utility of the associated actions. The training method, called *stochastic sharpening*, discourages

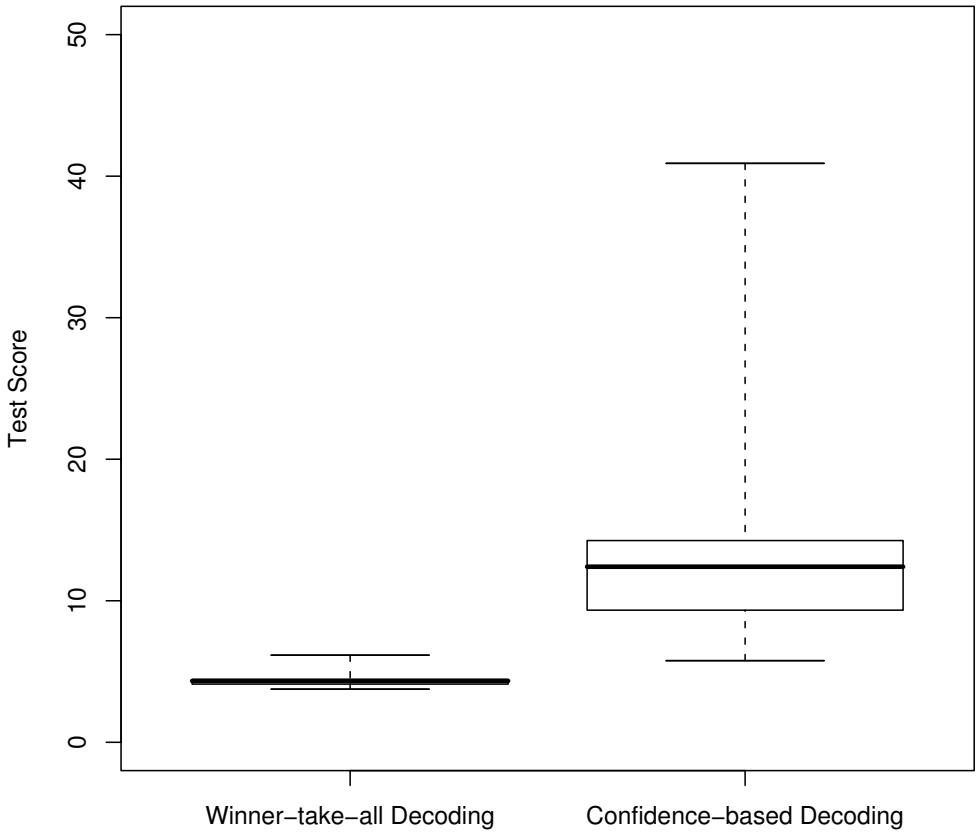


Figure 8.2: **Effect of stochastic decoding during testing.** Box-and-whisker plots show the median, quartile, and extremum values obtained by testing 31 networks using deterministic winner-take-all decoding, vs. testing the same networks using confidence-weighted stochastic decoding. (Lower test scores are better.) The stochastic decoding greatly degrades the networks' performance, suggesting that the patterns of output activations do not accurately reflect confidence or expected utility for the various options.

spurious output activations by interpreting the network's activation patterns as utility confidence values whenever it is evaluated during training. Networks with inappropriate activations will thus choose inappropriate actions more frequently than others, and consequently perform poorly on the fitness evaluations during training. As a result they receive lower fitness scores, and are eventually bred out of the population.

The rest of this chapter evaluates the concept of stochastic sharpening experimentally, and examines how sharpened networks can be used to introduce a controlled amount of randomness into the behavior of embedded game agents.

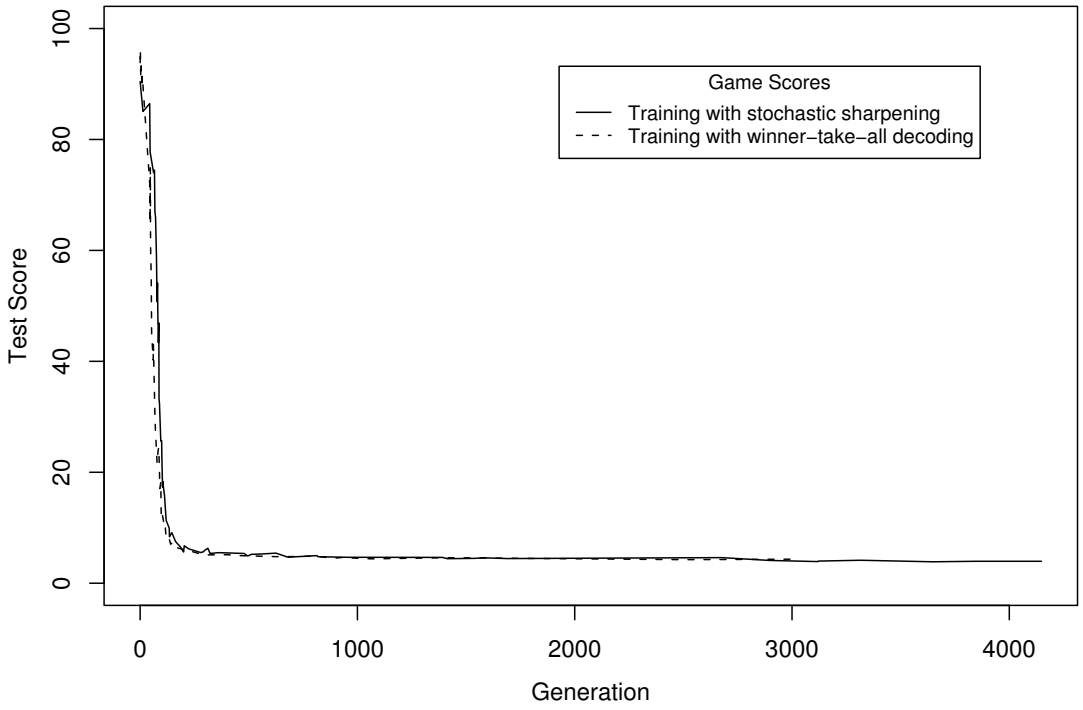


Figure 8.3: **Typical learning curves for neuroevolution with ordinary deterministic training and stochastic sharpening.** Learning progress is shown for the median performers of 31 runs of training with deterministic winner-take-all decoding and with stochastic sharpening. The plotted points are the average scores on the test set for the nominal best network at various times during training. Since learning is noisy, points are plotted only for the generations in which progress was made on the validation set, and then connected to make the plot easier to interpret. Other than some noisiness early on, learning with stochastic sharpening progresses similarly to learning with the deterministic winner-take-all method.

8.2 Experimental Evaluation

For evaluating stochastic sharpening experimentally two issues need to be studied: (a) its effect on learning performance, and (b) its effect on interpreting output patterns as utility confidence values. These requirements were covered by training several networks with and without stochastic sharpening and applying appropriate metrics to their performance during testing.

8.2.1 Learning with Stochastic Sharpening

Stochastic sharpening is implemented in neuroevolution by using confidence-weighted decoding during training. That is, at each move during the training games a legion's sensory inputs are propagated through the controller network being evaluated to obtain a pattern of activations at the seven action-unit outputs. Those activations are normalized so that their sum is 1.0, and the normalized

value of each is treated as the probability that its associated action should be selected for the legion's current move. Thus the behavior of the legion – and ultimately the game score – depends on the *pattern* of activations that the controller network produces, rather than on the peak activation alone. Since a network's evolutionary fitness is derived from the game score, evolution ultimately rewards networks that produce "good" patterns and punishes networks that produce "bad" patterns, where good patterns assign high probabilities to contextually appropriate moves and bad patterns assign high probabilities to contextually inappropriate moves.

When stochastic sharpening is used with neuroevolution the fitness values are initially more random due to the stochastic decoding of the poorly trained networks during evaluations, so learning initially progressed with slightly more variation. However, neuroevolution learns well under noisy fitness evaluations, and in the experiments training with stochastic sharpening rapidly converged onto a learning curve very similar to what was seen for deterministic winner-take-all decoding (figure 8.3).

The networks produced with stochastic sharpening ultimately converged to a performance that was better by a small but statistically significant amount in comparison to those trained with the base method. The 31 networks trained with deterministic winner-take-all decoding gave a mean score of 4.439 on the test set; those trained with stochastic sharpening gave a mean score of 4.086 when tested with stochastic decoding and 4.092 when tested with deterministic winner-take all decoding (figure 8.4). In both cases the improvement over the deterministic training method was statistically significant at the 95% confidence level ($p = 0.002$ and $p = 0.003$, respectively). The minimal variety in the performance of the sharpened networks under the two decoding methods suggests that stochastic sharpening greatly suppressed the secondary output activations, so that confidence-weighted stochastic decoding almost always picks the same action that the deterministic decoding does.

Stochastic sharpening also reduced the variance in the performance of the networks by an order of magnitude (figure 8.4). The performance of the 31 networks trained with deterministic winner-take-all decoding had a test score variance of 0.378; those trained with stochastic sharpening had a variance of 0.039 when tested with stochastic decoding and 0.051 when tested with deterministic winner-take-all decoding. Reduced variance in the result of a learning algorithm is useful because it increases the probability that a single run will perform near the expected value. For commercial application to difficult problems, a large number of independent runs may not be deemed feasible.

The *randomness* of a network tested with stochastic decoding can be defined as the percentage of the time that an output other than the peak activation is chosen. For networks trained with stochastic sharpening and tested with confidence-weighted decoding, the randomness was continually reduced as training continued, even beyond the point where progress at learning the task had flattened out (figure 8.5). This fact also suggests that stochastic sharpening suppresses secondary activations in the networks, at least for the current application.

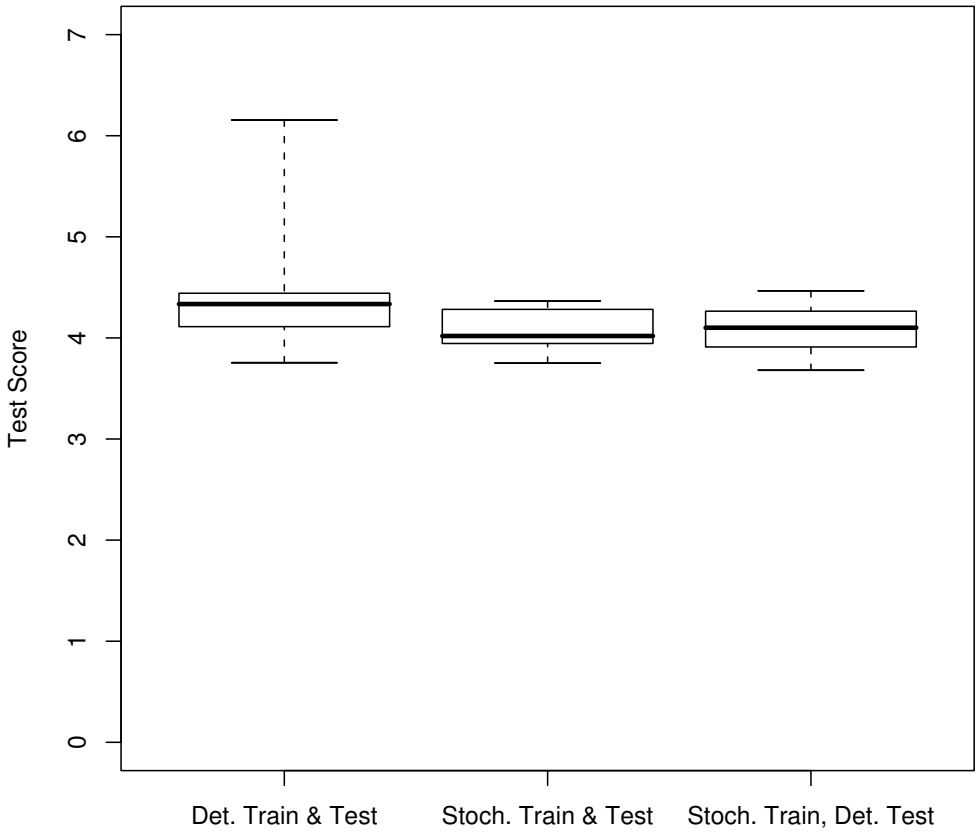


Figure 8.4: **Effect of stochastic sharpening on performance.** Box-and-whisker plots show the performance when using deterministic winner-take-all decoding for both training and testing (left) vs. using stochastic sharpening for training and either stochastic or deterministic decoding during testing (center and right, respectively). Stochastic sharpening improved the performance of the resulting networks and reduced the variability of the result as well. It also caused the networks to suppress most secondary activations, so there is little difference in the networks' behavior whether stochastic or deterministic decoding is used during testing (cf. figure 8.2; note the different scale on the y-axis).

8.2.2 Inducing Stochastic Behavior

An ϵ-*greedy* method (Sutton and Barto 1998) was used as a baseline method for comparison. Under ϵ-*greedy* control, a parameter ϵ allows direct specification of the amount of randomness in an agent's behavior. With probability $1 - \epsilon$ the controller's output is decoded and applied in the ordinary deterministic fashion; otherwise one of the other actions is chosen instead, selected randomly and with equal probability for each possibility. Since the secondary activations of the deterministically trained *Legion II* controller networks are not a useful guide for choosing among them (figure 8.2), the unbiased random selection is a reasonable approach.

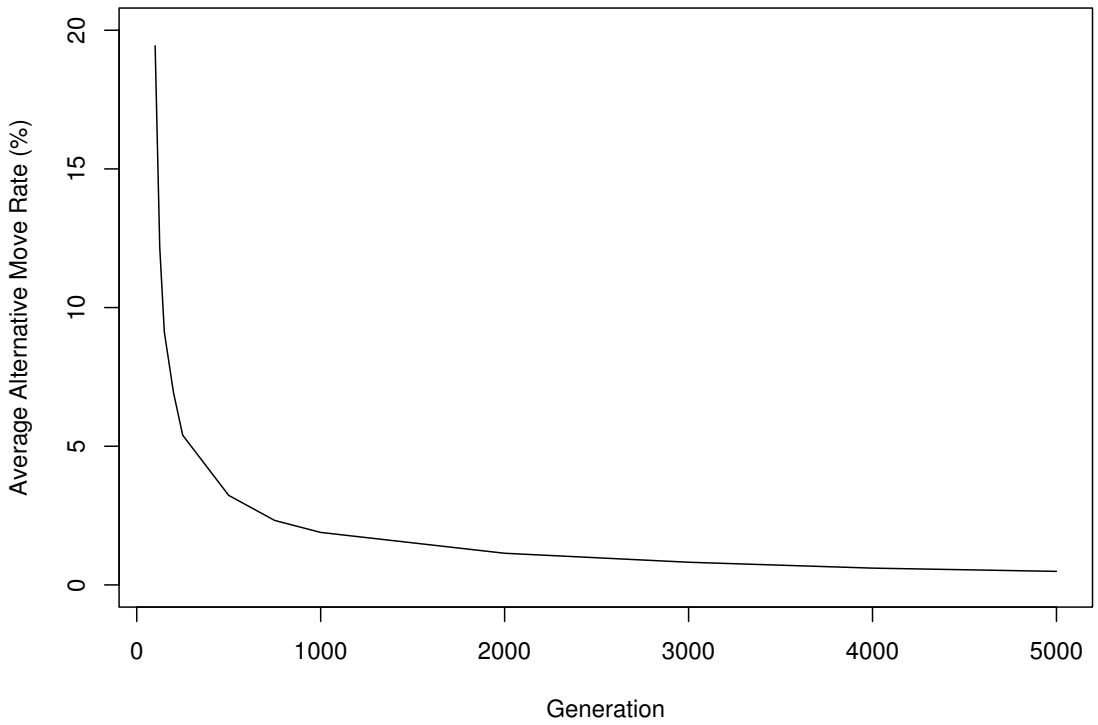


Figure 8.5: **Effect of training time on randomness.** A behavioral randomness metric, averaged over the 31 training runs with stochastic sharpening, is plotted against training time. The randomness with confidence-weighted stochastic decoding decreases as training progresses. The effect continues even after task-learning has stalled out (cf. figure 8.3).

Testing the networks that had been trained with deterministic winner-take-all decoding, using ϵ-*greedy* decoding for the tests, revealed that task performance degraded at an approximately constant rate as ϵ was increased over the range $0.00 \leq \epsilon \leq 0.15$. When the same method was used for testing the networks trained with stochastic sharpening the same pattern was seen, though the improved learning obtained by stochastic sharpening (figure 8.4) provided a constantly better performance for all ϵ (figure 8.6).

However, the hypothesis that stochastic sharpening produces networks with useful utility confidence values in their output activations suggests an alternative method of obtaining random behavior. This method works like ϵ-*greedy* method, except that whenever an alternative action must be selected it is chosen based on a confidence-weighted interpretation of the secondary output activations. That is, the activation levels other than the peak are normalized so that they total to 1.0, and then one is selected with probability proportional to the normalized values. When using this method task performance degraded at only about half the rate of the ϵ-*greedy* method, as randomness was increased. The method allows significantly more randomness to be introduced into the agents' behavior before their game performance suffers excessively (figure 8.6). An observer no-

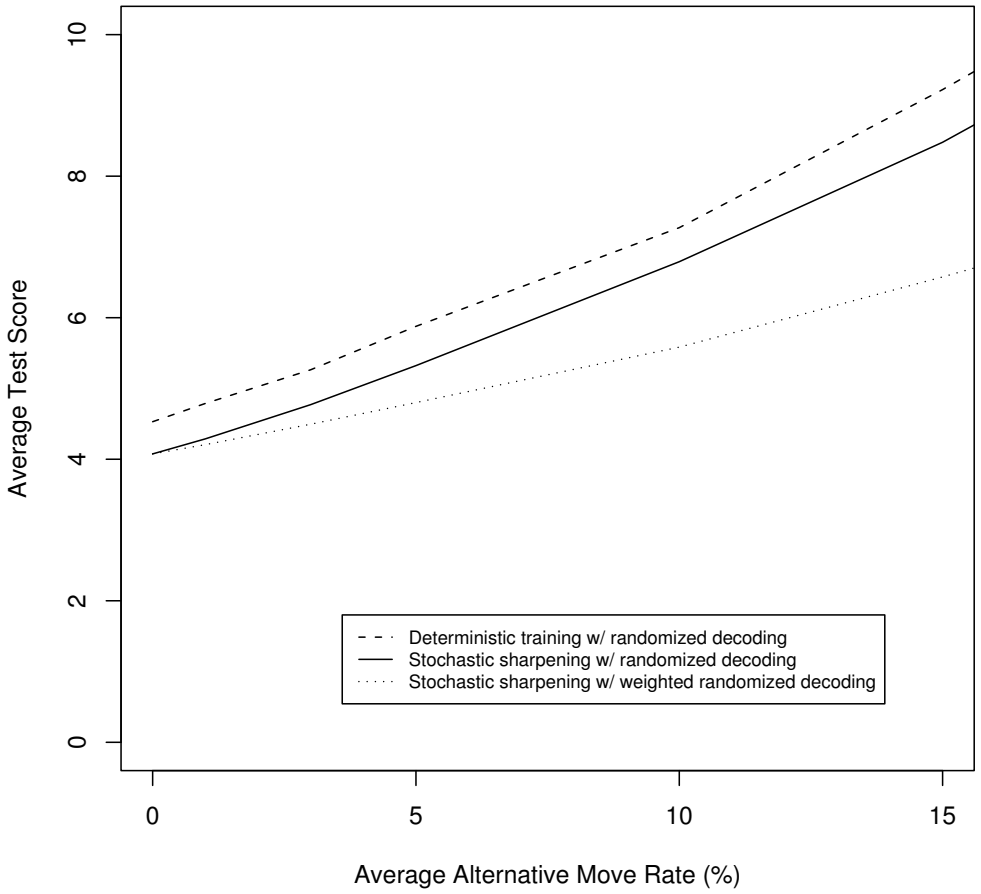


Figure 8.6: **Tradeoff between randomness and task performance.** Coerced randomness using ϵ-*greedy* decoding degrades game scores almost linearly as the amount of randomness increases over the range examined, regardless of which training method was used (solid and dashed lines). However, for networks trained with stochastic sharpening, if the alternative moves are selected by a confidence-weighted choice between the possibilities, the task performance score degrades far more slowly (dotted line).

tices the legions making sub-optimal moves slightly more often as the level of induced randomness is increased, but there is no qualitative change in their behavior. (Animations of the randomized behavior can be viewed at `http://nn.cs.utexas.edu/keyword?ATA`.)

8.3 Findings

The experiments show that it is possible to introduce stochastic behavior into game agents without degrading their task performance excessively. Moreover, the stochastic sharpening used to train the agents for stochastic behavior provided an absolute improvement in learning performance over the conventional deterministic method; stochastic sharpening may be a useful technique even when

training networks for deterministic use. The combination of stochastic sharpening with managed randomization allows agents in the *Legion II* game to choose a non-optimal move about 5% of the time and still perform as well as the fully deterministic agents trained by the conventional method. Greater or lesser amounts of randomness can be obtained by a direct trade-off against the agents' ability to perform their task well (figure 8.6).

The coerced-randomness method, when used with stochastic sharpening and confidence weighted choice among alternative moves, provides the best trade-off options of the methods examined here. It is also flexible, since the randomness parameter is set at run time; no re-training is required for changing the amount of randomness displayed by the agents. Indeed, the amount of randomness can be adjusted during the course of a game or simulation, by simply changing the current value of ϵ.

The improved learning performance for stochastic sharpening was an unexpected side benefit. There are two reasons for the improvement. First, when stochastic sharpening is used, even the peak output activations are somewhat low, whereas the deterministic method tends to produce networks that saturate their outputs. Saturated outputs tend to generate a race condition among a network's weights during training, which is generally detrimental to learning. Stochastic sharpening pushes the network's activations back away from the saturation point, and thus avoids the race condition in tuning the weights. Second, the stochastic decoding used during training serves as a self-guiding shaping mechanism (Skinner 1938; Peterson et al. 2001). That is, a partially trained network does not have to maximize the correct output in order to perform well; it merely needs to activate it enough to be selected with some probability. A partially trained network that does the right thing *sometimes* may show a higher evolutionary fitness than a partially trained network that stakes everything on being able to pick the right choice *always*.

It should be noted that stochastic sharpening does not depend on the choice of neuroevolutionary algorithm; it relies only on the use of action-unit output codings. Thus it should be possible to deploy stochastic sharpening as an add-on to neuroevolutionary methods other than ESP, such as CMA-ES (Igel 2003) and NEAT (Stanley and Miikkulainen 2002). Indeed, it may be applicable even apart from the use of artificial neural networks, so long as the decision engine can represent a choice between discrete options with scalar utility confidence values.

In the future, methods must be developed for inducing stochastic behavior in games and simulators that operate in continuous space and time, such as the NERO machine-learning strategy game (Stanley et al. 2005c). Since controllers for those environments generally select by degree – what angle or how fast – rather than selecting from among distinct options, operations in those environments will provide a different sort of challenge for randomizing behavior intelligently.

Chapter 9

Discussion and Future Directions

The previous chapters introduced and motivated the concept of visibly intelligent behavior, and demonstrated mechanisms for inducing various aspects of such behavior by machine learning in a test environment. This chapter offers a discussion of those experimental findings and their implications for our ability to generate visibly intelligent behavior in embedded game agents, and examines the possibilities for follow-up work. Closely related work not covered in chapter 2 is also examined here for contrasts with the new work.

9.1 Defining Visibly Intelligent Behavior

Section 1.1 defined visibly intelligent behavior as a subset of all intelligent behavior. A question naturally arises: is fakery sufficient where visibly intelligent behavior is required, and should the definition be modified to allow it? After all, if the motivation is to please the viewer, what difference does what is *really* happening make?

To some extent the question is moot. As described in section 2.1, the notion of intelligence is not easily defined, and research into artificial intelligence usually proceeds on an "I know it when I see it" basis. In the absence of a formal definition for intelligence, it is difficult to draw a line between "real" intelligence and fakery; sufficiently sophisticated fakery is indistinguishable from genuine intelligence. Thus the quest for visibly intelligent behavior could be seen as an attempt to pass a visual Turing test on a limited problem domain. That is, visible intelligence is achieved when human observers cannot distinguish between an embedded autonomous artificial intelligence and a passive playing piece that is moved by an unseen human player.

Thus in future work it may be useful to turn to observers for expectations and/or evaluations of attempts to produce visibly intelligent behavior. For example, internet forums could be mined for complaints about specific behavioral traits of the agents in published games, and the more common complaints could be taken as signs of what observers expect to see in the behavior of intelligent agents of similar types in similar environments. A human subjects experiment could then use a

problem-specific visual Turing test to measure progress at producing visibly intelligent behavior; a suite of such tests using varying problems would test for progress on machine learning methods designed to produce such behavior.

Alternatively, visibly intelligent behavior could be assumed to correlate with a temptation in observers to anthropomorphize what they see in an agent's behavior. Under that assumption, a human subjects test could expose observers to animations of autonomous agents in action, prompt the observers to answer questions or provide a free-form description of what they saw, and examine those responses for the frequency of anthropomorphizing terminology.

However, a researcher may prefer to pursue "real" intelligence rather than fakery, for philosophical or aesthetic reasons. Under such personal motivations the distinction between intelligence and fakery must also become a personal decision. For example, a programmed control that locked the *Legion II* garrisons into their cities, except when allowed out by the rules of their tactical doctrine, would produce the intended effect, but might not be satisfying as "real" intelligence – especially for an approach that otherwise relies on machine learning to produce the controllers.

Moreover, as we attempt to produce more sophisticated behavior in more complex environments with artificial intelligence, the number of such behavioral patches needed to produce visibly intelligent behavior is likely to grow large, and may become as difficult to specify completely and correctly as scripted behavior is. The negative reactions of gamers to the observed behavior of artificially intelligent opponents in commercial games indicates that scripting has not been able to provide a satisfactory degree of visible intelligence in practice, and there is no reason to suppose that applying an arbitrary number of behavioral patches to some other control mechanism will ultimately yield a more satisfactory result. The promise of machine learning is the ability to train agents to appropriate behaviors without requiring excessively complex specifications. The "mindless" oscillating behavior produced in the ATA experiment (chapter 5) indicates that completely unbiased learning mechanism will not always suffice, though it can be hoped that general learning mechanisms such as example-guided evolution (chapter 6) can provide the necessary bias, and eliminate the requirement for detailed specifications of behavior that are not portable between agents and applications.

Additionally, example-guided learning uses human intuitions to address the problem of human intuitions: the examples are generated using human intuitions about how the agents should behave, and if that behavior is properly induced by the learning system then the resulting agents will meet humans' intuitive expectations, to the extent that they are in mutual agreement. By mapping intuitions onto intuitions via the learning system, the labor-intensive and error-prone process of specifying the details of behavior required for all possible contexts can be eliminated. Thus further work must examine the challenge of learning such mappings for arbitrarily complex behaviors in arbitrarily complex environments, and games will continue to be a suitable platform for that research.

9.2 Adaptive Teams of Agents

The experiments reported in chapter 5 are an important demonstration of the capacity for flexibility in individuals and adaptivity at the team level, without any requirement for additional training. Balch (1998) has previously called attention to the robustness advantage of a homogeneous team (no individual is ever "the" critical component), but found that the agents in a homogeneous team interfere with one another in a foraging task. He found that homogeneous teams performed better than heterogeneous teams at the foraging task, but his best robo-soccer an cooperative movement teams were heterogeneous, and other researchers have found heterogeneous teams superior at other tasks as well, e.g. Yong and Miikkulainen (2001) on the predator-prey task. Balch concludes that the homogeneous team is superior on the foraging problem because it is essentially a single-agent problem with multiple agents present and pursuing the same task. However, an adaptive team allows retention of the advantages of a homogeneous team even on tasks that explicitly require a diversity of behavior.

The *Legion II* agents learn *in situ* diversity despite their static homogeneity, because the reward structure of the game demands it. Presumably any reinforcement method could learn such adaptivity, though it would require an appropriate reward structure along with the sensory and representational capabilities necessary for the task. Non-reinforcement methods should be able to learn run-time adaptivity as well, e.g. by use of examples. (The backpropagation-only experiments reported in Chapter 6 learned it quite well.)

Goldberg and Matarić (1997) addressed inter-robot interference on the foraging task by imposing a dominance hierarchy on an otherwise homogeneous team, and using *pack arbitration* to allow a dominant robot to preempt any interfering operation by a subordinate robot. The scheme requires hand-coding of the arbitration, though it may be possible to code it in a reusable application-independent way. However, it does not appear that pack arbitration will support adaptive diversity of behavior beyond resolution of direct inter-agent interference. For the *Legion II* application, pack arbitration could prevent two legions from attempting to garrison the same city if the game engine did not already prevent it, but it would not cause the subordinate legion to adopt the role of a rover.

A more general solution to the problem of coordination in a team of homogeneous agents is recursive agent modelling (Durfee 1995). Recursive agent modelling treats other agents as predictable elements of the environment, so that optimal actions can be selected on the basis of their effect on cooperating agents. Presumably the agents in *Legion II* implicitly learn to predict the actions of their teammates given the current game state, to the extent that it helps them optimize their behavior, but that conjecture has not yet been verified.

The primary issue for follow-up work on the ATA architecture is scaling; for application in the current generation of commercial computer games, ATAs would need to scale to richer environments, more choices of action, and larger maps. The richer environments will often involve the existence of multiple ATAs on a single side. For example, section 5.1 mentioned that games in the *Civilization* genre generally include a "Settler" unit type, and all the Settlers in play must operate

as an adaptive team. However, they also include other unit types, such as legions, which are in play at the same time and must also operate as adaptive teams – there is a need for garrisons and rovers in those games too. Many of the unit types must cooperate closely, e.g. infantry and artillery or caravans and naval transport, so it will not be possible to train them in isolation. The most promising option seems to be coevolution, with all the unit types in play together and learning to cooperate adaptively with their own kind while cooperating with other types. Coevolution is a much studied topic, but coevolving multiple ATAs will be an unprecedented challenge.

Richer environments will also demand more sophisticated sensor systems for the agents. The ATAs in *Legion II* learned their task with only a very fuzzy representation of their environment, but other domains can be expected to require more detailed representations. Most game environments will also require more sensor classes, so that the agents can distinguish the different unit types among their allies and opponents. Thus it can be expected that scaling up to richer environments will require the use of larger input representations, with the resulting increased demands on the learning methods. For example, if neural networks are used for the controllers, the number of weights to be learned for the input layer will grow linearly with the number of sensor elements. If that results in a need for a larger hidden layer, the growth will be superlinear.

Moreover, some domains will require representations for recursive symbolic relationships such as "in" and "on". Processing such relationships may require drawing on techniques developed for natural language processing in artificial neural networks (Elman 1990; Miikkulainen 1996), or inventing entirely new ones.

Scaling to more choices of action for the agents will have the same effect: larger, more complex controllers, and more input-output relations to be learned during training. Further challenges will arise for applications to continuous-state games and simulators. For most of those applications the agents will need to learn to produce continuous-valued outputs for motor control, rather than a simple choice between action units.

Even such a seemingly simple matter as scaling to larger maps will introduce new challenges. The legions in *Legion II* disperse themselves across the map suitably by self-organization, but the map is relatively small. For very large maps it is not clear that simple self-organization will provide the necessary dispersal if localized concentrations are required. Thus it may be necessary to impose hierarchical control atop the adaptive team, with generals marshaling legions for the necessary global pattern of dispersal, after which the localized dispersal and division of labor could proceed as before.

It can therefore be expected that scaling the ATA architecture to the needs of commercial applications will require and stimulate the invention of powerful new techniques for inducing sophisticated behavior in teams of autonomous intelligent agents.

9.3 Example-Guided Evolution

Artificial Lamarckian evolution has been in use for some time, e.g. by Grefenstette (1991), Whitley et al. (1993), and Ku et al. (2000). The distinction for the current work is the application: it is used here to guide evolution to discover controllers that exhibit specifically desired behavioral traits, rather than as a way of implementing life-time learning or simply helping evolution solve some difficult task. The current work is also the first combination of Lamarckian neuroevolution with the ESP mechanism.

There are two areas of recent work that are closely related to example-guided evolution, and it is worth understanding the similarities and differences between them.

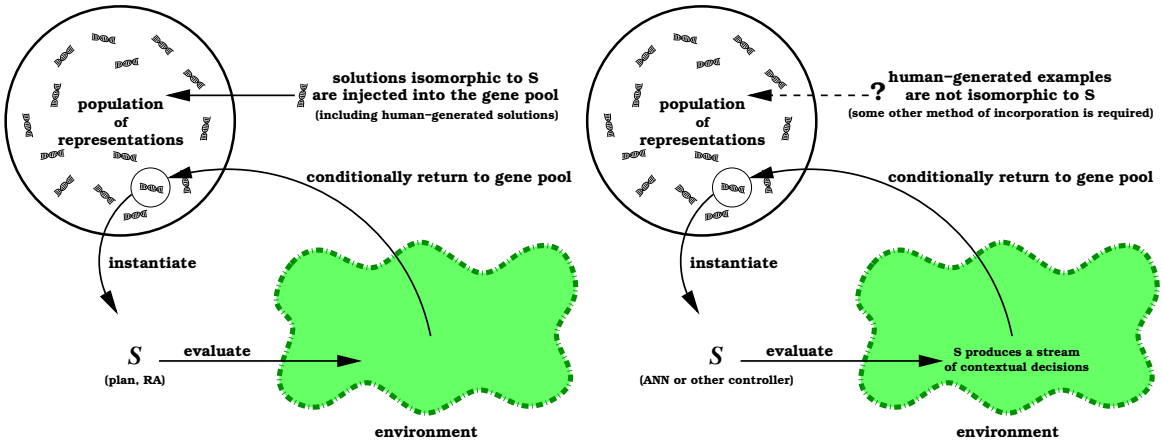


Figure 9.1: **CIGARs and hard-to-specify solutions.** When solutions are of a form that can be constructed by humans without too much difficulty, such as a static plan or resource allocation table, CIGARs can incorporate human-generated "cases" (representations of solutions) directly into the evolutionary population (left). But when the solutions are excessively complex, such as an artificial neural network or other agent controller, some indirect method of incorporating them must be used instead (right).

The most closely related work in terms of goals is the Case-Injected Genetic Algorithm (CIGAR), a population-tampering mechanism described in section 2.6.1. The "cases" are in fact examples; however, they are examples of complete solutions rather than examples of behavior. They are therefore very convenient and direct mechanisms for steering evolution toward some desired type of solution, but it is not feasible to use them when the necessary solution is a difficult-to-specify object such as an ANN agent controller (figure 9.1). As a result, CIGARs have been used to incorporate human guidance when creating relatively simple static objects (Louis and Miles 2005), but not for complex agent controllers.

The advice-taking system of Yong et al. (2005), also described in section 2.6.1, is very similar to the current work in its application and goals, though different in mechanism. It is used to insert human solutions into an evolutionary population of game agents, and like example-guided evolution it is a solution-tampering mechanism. The distinction is that it inserts explicit rules of

behavior rather than guiding toward a complete set of behaviors by induction. In principle, advice-taking systems could build up a complete set of behaviors rather efficiently by repeated application. However, if a complete set of rules is easy to specify, a script or rule-based system would likely be used rather than a genetic algorithm. Thus advice-taking is useful when machine learning is needed for producing a complex, underspecified controller, but some clearly expressible details of the controller's target behavior are known. Notice that the use of advice is orthogonal to the use of example-guided evolution, and there will likely be advantages to employing them together as we apply evolutionary learning to increasingly complex games.

Advice has also been used to directly modify the value function approximator for on-line reinforcement learning (Kuhlmann et al. 2004). In this system a rule-based *advice unit* modifies the function approximator in real time to guide the agent's learning. The advice unit tracks human-specified rules of behavior, and whenever one of the rules "fires" the advice unit modifies an action's value up or down, depending on whether the rule recommended for or against that action. Appropriate advice speeds learning by adjusting the value function approximator to its ideal form more quickly than can be done by exploration.

Once a decision has been made to use examples for agent training and the examples have been collected, new options arise. A discrete-option controller like the one used to implement the legions' control policy can be seen as a simple pattern classifier. That opens up the possibility of bringing the full weight of the field of learning classifier systems to bear on the problem. As an example, Sammut et al. (1992) use *behavioral cloning* (Bain and Sammut 1999) to induce a decision tree for controlling an airplane in a flight simulator, by applying C4.5 (Quinlan 1993) to 90,000 $<$*state,action*$>$ tuples extracted from traces of human control. The same task is addressed by van Lent and Laird (2001) using their rule-based *KnoMic* "Knowledge Mimic" system, which applies a modified version of the *Find-S* algorithm (Mitchell 1997) to a trace of $<$*state,action*$>$ tuples. Both sets of authors found it necessary for a human expert to split the traces into segments according to changing goals. It is not clear whether that requirement arose from an inherent difficulty of the problem or from weaknesses in the power of the methods used. However, since both operate on the same sort of data required by example-guided evolution, future comparisons will be straightforward.

Ng and Russell (2000) take an oblique route to the problem of instilling behavior. Their *inverse reinforcement learning* methods induce the reward function implicit in example behavior. From the reward function ordinary reinforcement learning methods can then be used for agent training, including methods that are not normally amenable to influence by examples. That opens the possibility for including such methods in empirical comparisons. Discovering an implicit reward function may also prove useful as a method for analyzing the tacit motivations for the example behavior.

An extended overview of methods for learning from demonstrations in the robotics domain is given in Nicolescu (2003).

The *Legion II* strategy game used for the experiments in this dissertation models only the

coarsest aspects of behavior, e.g. whether to move and in which direction. A very different consideration arises when creating artificial individuals for use as interactive characters or performers in animations. Such individuals must show nuances of emotion, including appropriate prosodic variations to verbal utterances, in order to be convincing to interacting or observing humans. Such requirements have given rise to the field of *believable agents* which attempts to model such nuances (e.g., Bates 1997; Hayes-Roth et al. 1999; Gratch and Marsella 2004, 2006; Johnson et al. 2002). The requirements for strategic or tactical behavior and emotional behavior converge in the popular genre of *first person shooter* games and similar training tools, so it will be useful to investigate inducing coarse and nuanced behavior together. For sufficiently rich environments it may be difficult to fully specify a mapping between game states and appropriate emotional responses, just as it is for mapping between game states and strategic or tactical decisions, so at some point it may be useful to rely on learning biased by human-generated examples to create emotionally believable agents.

Preliminary experiments have shown that for both backpropagation and example-guided evolution the relation between game scores and example-classification errors is not linear as learning progresses. Example classifiers are ordinarily applied to static sets of examples, and their error rate does not depend on the order of presentation of the examples. In contrast, discrete-state agent controllers are "embedded classifiers" – that is, each classification generates a move that changes the game state, and the changed game state influences what further inputs are presented for classification. Thus it is possible that the learning requirements for embedded classifiers differ fundamentally from the requirements for ordinary static classifiers. That hypothesis needs to be tested empirically, and if there are in fact important differences, the quest for visibly intelligent behavior may open up an entirely new side field of research.

Regardless of the learning technology used, deliberative agents will pose new challenges for example-based training. It is straightforward for a game engine to capture the <*state,action*> pairs of human moves, but it is not possible to capture the human's internal state that influenced the choice. Thus methods such as the *implicit imitation* of Price and Boutilier (2003), which assume visibility of a mentor's internal states rather than its actions, cannot be directly applied to human-generated examples. Previously mentioned work such as Sammut et al. (1992) and van Lent and Laird (2001) address the challenge of unrecorded internal states in the trainer by working with traces of examples rather than sets of discrete examples. In principle traces will address the problem for guided evolution as well, but for Lamarckian neuroevolution some method will need to be devised to allow for the unrecorded internal states during the adaptation phase. It may be as simple as using *backpropagation through time* (Werbos 1990), starting from a neutral internal state at the beginning of each trace (e.g., with all the activations in a recurrent layer set to zero). Such methods will need to be evaluated empirically.

Example-guided evolution was used in chapter 6 to impose discipline on the trained agents and to demonstrate the ability to train them to distinctive doctrines, for better or worse. The importance of that second capability cannot be overstated. In the traditions of artificial intelligence

research, improved task performance is almost universally taken to be a sign of more intelligence, and therefore "better" (cf. section 2.1). However, for some real-world applications of embedded game agents, worse is "better". For example, games with AI opponents usually support several difficulty levels. If the AI opponents are implemented as embedded autonomous agents, the requirement for varying difficulties will remain in effect: it will be necessary to provide agents that operate at different levels of task performance. The experiments in sections 6.3 and 6.4 show that embedded agents can still display visibly intelligent behavior while pursuing an inferior plan or operating under an inferior doctrine; that is precisely what is needed for games with embedded agents and a selection of difficulty levels. Examples of play can be generated using doctrines that are appropriate for the various difficulty levels, and separate controllers trained from each set. Then when a player starts the game and selects a difficulty level, the game engine can select the appropriate "brain" for use as the controller for the embedded agents.

There exists another potentially useful application of example-guided evolution: opponent modelling. If an agent can collect examples of its opponents' play, it can use example-guided evolution to train models of the opponents – including human players! A deliberative agent could generate such a model as a side-computation, and intermittently test it by comparing its predictions to the opponents' actual moves. When the model is deemed to be sufficiently accurate the deliberative agent could use it to optimize its own behavior by predicting opponents' responses to its own various choices of action.

9.4 Exploiting Sensor Symmetries

Chapter 7 offered a method for exploiting symmetries in an agent's structure and environment to generate artificial training examples, and demonstrated by experiment that the method is useful. Yet many questions immediately arise, and further experiments are needed to explore the extent of that utility.

For example, by both of the metrics examined in chapter 7 the optimal result was obtained when relatively few example games were used. Will that result obtain in other applications as well? Will the advantages of using the artificial examples disappear when only a few more example games become available, or will they always offer some advantage over using the original examples alone? Will the artificial examples provide a similar benefit when used with example-guided evolution rather than supervised learning, or will the noise inherent in evolutionary learning dominate any advantages to using them? Only further experimentation can answer these questions.

Broader questions also arise. Are there other ways that symmetries of structure and environment be exploited? Are there other useful mechanisms for creating artificial examples?

Regarding the former question, a structural approach to ensuring symmetry of behavior is suggested by the work of Togelius and Lucas (2005). In that work the authors studied the use of neuroevolution to train controllers for the game *Cellz* (Lucas 2004). *Cellz* is a sort of predator-

prey or "dueling robot" game, where agents representing unicellular biological organisms compete for resources and attempt to dominate the environment. *Cellz* is a continuous-state game, but is otherwise somewhat like the *Legion II* game. In particular, the agents are equipped with radially and rotationally symmetrical sensors, they see the world a fuzzy sense based on a $\sum_i \frac{1}{d_i}$ sensing system, and they are controlled by propagating the sensor values through an ANN controller.

The authors took note of the rotational symmetry of the sensor arrays, and exploited that symmetry to reduce the number of input weights to be trained for the controller network. Their system used convolutions of the sensory values fed into a reduced-size input layer; the convolutions allowed the sensor values to be "stacked" at the smaller input without rendering the rotations of a game state indistinguishable. Though their focus was on the opportunity for modularizing the controller network to improve learning, their system has a side effect of preserving rotational symmetries in the activation values of the controller network's hidden layer. That is, if the neurons in the hidden layer are conceived as a ring rather than a line, a rotation of the environment produces a rotation of the activation pattern in the hidden layer. It may be possible to exploit that effect to help coerce rotationally symmetric behavior in an agent, even without the use of examples. Notice, however, that that mechanism is completely orthogonal to the mechanism described in chapter 7, so it is possible to use the two mechanisms together, if such proves to be useful. Thus further research should examine the possibility of coercing behavioral symmetries architecturally, and determine whether there is any benefit to be had from combining the two independent methods.

Symmetries have also been exploited by very different mechanisms. Fox and Long (1999) used symmetry detection to trim the search space for a planner when applied to the *Gripper* and Traveling Salesman problems. In a very different context, Ellman (1993) created abstract symmetries (and "approximate symmetries") to allow pruning of large subsets of the solution space in constraint satisfaction problems; such symmetries are not necessarily geometric. Zinkevich and Balch (2001) used a more intuitive notion of symmetry, such as the reflection of a soccer field along its midline, to speed reinforcement learning for Markov decision processes (MDPs); the authors showed that a symmetric MDP necessarily also has a symmetric optimal value function, a symmetric Q function, symmetric optimal actions, and a symmetric optimal policy.

These methods effectively reduce the number of states in some space of interest (solution space or environment state space) by identifying two or more symmetric states as a single state. This is the opposite of the operation that exploited symmetries in Chapter 7, which effectively split a given state into a set of symmetric states. Various parties have suggested pursuing the former operation to reduce the size of the problem state space in *Legion II* , e.g. by rotating a legion's egocentric sensor activations to a canonical orientation before selecting an action, but no effective mechanism has yet been discovered. However, such a scheme has the potential of reducing the size of the policy to be learned by the legions by a factor of twelve, so the challenge merits further consideration.

Regarding the second question, whether other useful mechanisms for creating artificial ex-

amples exist, more work also needs to be done. Artificially generated examples have recently become a topic of interest in the field of learning classifier systems. In that field they are used to increase the diversity of opinion in ensemble-based classifiers; as a result, the artificial examples do not strictly have to be "correct" by some absolute standard. Thus they are usually estimated from statistical models of the distribution of features in the training set, or chosen on an information-theoretic basis. A brief survey of the use of such examples is given in Melville and Mooney (2004). The use of such examples has not yet been investigated for its simple utility in expanding the number of examples available for training, as was seen for the artificially generated examples in chapter 7 (Melville 2006). The question of the utility of such "estimated" examples for training agent controllers offers another avenue of research.

On a related note, the vote-counting mechanism used for identifying consistency errors in chapter 7 suggests a method for using a single neural network as a "virtual" ensemble, by reflecting and/or rotating its inputs and then applying the reverse transformation to its output. That mechanism will guarantee absolute symmetry of input-output responses: for a given game state and any of its reflections or rotations, the same 12 votes will be counted. However, it moves the extra processing from the training epoch to the deployment epoch, and thus may be unsuitable for applications where computation is a bottleneck. Investigation of the properties of this mechanism, called *virtual bagging* because it constitutes bagging (Breiman 1994) with a virtual ensemble, is already underway (Bryant 2006).

9.5 Managed Randomization of Behavior

Chapter 8 argued that some amount of stochasticity is essential for agents embedded in games or simulators as competitors, or else game players and simulation users will be tempted to learn to exploit the agents' determinism instead of learning to address whatever problem the game or simulator was designed to offer. An example mechanism was demonstrated, with stochastically decoded neural networks providing a parameterized degree of randomness, and a modified mechanism for training unit-action coded networks so that the randomized choices were optimized for minimal degradation of task performance.

It was also pointed out in that chapter that the mechanism is not strictly limited to use with artificial neural networks. Any controller that can specify utility confidence values in its output can be decoded stochastically for a similar effect. If such a controller is trained by evolution, or probably any mechanism of reinforcement learning, the method of stochastic sharpening can still be used for the training. This immediately establishes an area for comparative experiments: other controller technologies and other learning mechanisms can be compared for the effect on task performance when they are used to introduce managed randomization into an agent's controller.

Coerced randomization is used in on-line reinforcement learning as a way of ensuring continued exploration as an agent converges on a control policy; the ϵ-*greedy* method described in

section 8.2.2 is a classic mechanism for doing this. An alternative approach is using *softmax* selection, where a stochastic selection is made among all options, with a bias in favor of "preferred" options. See Sutton and Barto (1998) for an overview of ϵ-*greedy* and *softmax* mechanisms.

The decoding method used during stochastic sharpening is a *softmax* mechanism, i.e. the decoding rule selects among the actions with probabilities proportional to the associated output neuron activation levels, i.e. to the utility confidence values (section 8.2.1). However, the method has little utility as mechanism for coerced randomization, since in a well-trained network the sharpening was so extreme that behavior varied very little from winner-take-all decoding applied to the same network. In principle the amount of randomness provided by this mechanism can be controlled by limiting the amount of sharpening, i.e. by reducing the amount of training. However, that strategy introduces the Goldilocks problem of determining the "just right" amount of training to be used; as with other *softmax* strategies such as decoding according to the Gibbs distribution, the strategy replaces the intuitive and transparent ϵ parameter of the ϵ-*greedy* method with a less transparent parameter, in this case the number of generations of training (Sutton and Barto 1998). Moreover, when training was reduced enough to provide useful amounts of randomness, task performance scores deteriorated far faster than the baseline ϵ-*greedy* method.

The method introduced in chapter 8 can be seen as a combination of ϵ-*greedy* and *softmax* methods. With probability $1 - \epsilon$ the peak activation is selected, just as for the ϵ-*greedy* strategy; otherwise a biased *softmax*-style choice is made, but limited to a choice among the non-peak activations. The ϵ mechanism compensates for the possibility of extreme sharpening, and the *softmax*-style mechanism biases the choice among the remaining alternatives.

The experiment in chapter 8 introduced stochastic behavior by randomized decoding of the controller's output. Entirely different mechanisms are also conceivable. For example, noise could be added at a controller's inputs (or hidden layer, if a neural network), and then the randomization at the output would be the result of a principled amount of uncertainty about the state of the system on the agent's part. For deliberative agents, randomness could be added to the internal state instead.

An alternative approach would be to provide an agent with multiple independently trained controllers that would not produce identical outputs for a given input, and select between the available controllers at random. That mechanism would reflect an agent's random choice between competing behavioral policies, each of which should specify a reasonable response to a given sensory input, and thus might produce more nearly optimal responses than the stochastic decoding used in chapter 8. Such a system has been used by Whiteson and Stone (2006) to encourage exploration in a reinforcement learning system, using ϵ-*greedy* and *softmax* methods to select among the current members of an evolutionary population during training. The method could be used to select between distinct but fully trained controllers as well.

Since the agents in the *Legion II* experiments use reactive controllers the live in a sort of "eternal present", and so to some extent it makes sense to introduce randomness to their behavior on a continual basis. However, when deliberative agents are used, their moment-to-moment decisions

are derived from a history-dependent internal state as well as the present sensory inputs. That internal state is conventionally interpreted as a memory, but it can also be interpreted as a plan – however abstractly specified – because rewinding a game and changing the agent's internal state at some given time would likely result in a different sequence of decisions thereafter. Likewise, the policy-switching system described above can be conceived as a plan-switching system, whether the agents are deliberative or reactive. For such plan-based behavior systems it is no longer necessary (and perhaps not desirable) to introduce the behavioral stochasticity on a moment-to-moment basis. Instead, it would be possible to modulate behavior on an intermittent basis, by switching between "plans" at random points in time. Such a mechanism might well yield more coherent, self-consistent behavior on the agent's part, but still provide a useful degree of unpredictability if a sufficient number of sufficiently different "plans" are available.

It is a given that some amount of unpredictability is desirable for some applications of intelligent agents, and if the application calls for visibly intelligent agents that unpredictability needs to be obtained at minimal cost to other desirable traits such as discipline and self-consistency. Thus the suggestions above, and others, need to be evaluated for their relative utility and analyzed for the trade-offs they require. The challenge of producing useful amounts of principled randomness into the behavior of autonomous agents may well evolve into an entire field of research.

It should be noted that neuroevolution with stochastic sharpening improved the task performance of the controller networks even in the absence of stochastic decoding when deployed. Stochastic sharpening is a simple mechanism, and its only requirement is the use of scalar utility confidence outputs. Its use is orthogonal to most other details of the training mechanism; for example, it could be used with NEAT or rtNEAT (Stanley and Miikkulainen 2002; Stanley et al. 2005b) rather than the ESP mechanism. If the performance advantage it provided in the chapter 8 experiments is seen with other learning methods and/or in other applications, stochastic sharpening will find utility as a general purpose "piggyback method" in its own right.

9.6 Neuroevolution with ESP

The basic technologies used for creating visibly intelligent behavior in the preceding chapters also merit further investigation, and any discoveries will immediate benefit other application areas as well. In particular, two questions have arisen regarding the enforced sub-populations mechanism for neuroevolution.

As described in section 4.2.2, it is conventional to smooth the noise of evaluations when using ESP by evaluating each neuron several times per generation, associated in a network with a different random selection of peers each time, and average the fitness obtained by the tests. However, fitness evaluations are always noisy when training networks for the *Legion II* game, because different training games are used each generation. Some of the games may be more difficult than others in an absolute sense, or different games may be difficult for different sets of partially trained

neurons by pure luck. Yet neuroevolution with ESP learns solutions to the *Legion II* problem anyway, suggesting that it is very tolerant of noise. So the question arises, are the repeated evaluations per generation actually necessary, or advantageous? Preliminary experiments show that a single evaluation is also effective, if allowed to run for more generations. Further research should investigate the possibility of an optimal trade-off between evaluations per generation and the number of generations used, and whether such a trade-off would be universal or application-specific.

The second question regarding neuroevolution with ESP is whether its power derives from the fact that each population evolves a single neuron as a functional unit, as commonly assumed, or whether the mere fact that sub-populations are used provides the power. The temporary association of neurons into a network can be seen as a "trial crossover"; unlike the crossovers used in conventional genetic algorithms, the trial crossovers are defeasible. If the power of ESP derives from the use of sub-populations *per se* rather than the fact that the sub-populations evolve individual neurons, then it is possible that the optimal number of sub-populations does not match the number of neurons in the network. Such possibilities need to be investigated, partly for a better understanding of how ESP works, and partly for the possibility of discovering new optimization mechanisms.

It may be further observed that the mechanism of enforced sub-populations is a general concept, independent of many of the other details of a genetic algorithm. Thus it may prove possible to use it in conjunction with NEAT and rtNEAT (Stanley and Miikkulainen 2002; Stanley et al. 2005b) or CMA-ES (Igel 2003), which are ESP's major rivals for the currently most effective mechanism for neuroevolution. It is possible that using enforced sub-populations with one of the other methods will produce an even more powerful neuroevolutionary algorithm. Moreover, if it turns out that the power of ESP lies in the use of sub-populations *per se* rather than the fact that the sub-populations represent single neurons, then it is likely that ESP will prove useful to genetic algorithms in general, not limited to neuroevolution.

9.7 Summary

This dissertation has focused on the concept of visibly intelligent behavior and the requirements it imposes on certain types of agent applications, and has provided example solutions to a select set of challenges that the concept offers when neural networks are used to control agents embedded in a strategy game or simulator. However, the work has also raised a number of questions not yet answered, and the example methods it has provided are subject to competition from alternative mechanisms, whether existing or yet to be invented. The dissertation should therefore be seen as a supplement to a broad and deep field of research, interesting in its own right, utile for commercial applications, a challenge for our methods of learning and representation, and ripe with implications for our understanding of natural and artificial intelligence.

Chapter 10

Conclusion

The goal of this dissertation has been to lay a foundation for applications of neuroevolution to the creation of autonomous intelligent agents, for contexts where the agents' behavior is observable and observers have strong intuitions about what sorts of behavior do and do not make sense for an intelligent agent. That goal has been pursued experimentally – and successfully – by identifying problems and opportunities of interest and finding or inventing methods that solve the problems and exploit the opportunities.

This chapter offers a brief high-level summary of the work presented in the dissertation. Section 10.1 itemizes the dissertation's major contributions to our capabilities and understanding of artificial intelligence and its application to the challenge of visibly intelligent behavior in embedded game agents. Then section 10.2 completes the dissertation with a final statement of the importance of this work.

10.1 Contributions

Visibly Intelligent Behavior

The dissertation calls attention to the importance of visibly intelligent behavior in embedded intelligent agents for applications where their actions are observable and viewers have intuitions about how those actions should be carried out. Recent work has made neuroevolution increasingly viable as a technology for training agents to operate in the rich environments of strategy games and simulators, and this dissertation supplements that work with methods for inducing visibly intelligent behavior in such agents.

The ability to produce embedded agents that display visibly intelligent behavior may open the door for applications of research-grade artificial intelligence in commercial games, providing favorable PR for AI research and an economic boost to the game industry. AI may well become *The Next Big Thing* for that industry, spurring additional public interest in AI research.

The concept of visibly intelligent behavior has shaped the rest of the work presented in the

dissertation, which supplements similarly motivated work by other researchers working with other AI technologies and/or application areas.

Adaptive Teams of Agents

Flexibility is an important part of intelligence; the ability of an agent to adapt to circumstances is perhaps the holy grail of AI research. Such adaptivity is an important competence for embedded game agents, because the rich environment of a modern computer game presents the agents with multiple competing demands and opportunities that change continually over the course of a game.

The Adaptive Team of Agents is a multi-agent architecture that approaches the challenge of flexibility in a team by providing individual agents that are pre-trained to be adaptive. A team responsible for multiple tasks does not have to be tailored for each possible distribution of those tasks: an adaptive team can tailor itself *in situ* by exploiting the flexibility in the individual agents' behavior. In observable environments such as games, an adaptive team can continue to meet the team's goals as the fortunes of the game turn, and will convince viewers that the individual agents are behaving intelligently rather than programmatically.

Example-Guided Evolution

Genetic algorithms have proven to be a powerful method for solving problems of various types, including training controllers for intelligent agents. However, genetic algorithms optimize blindly, and sometimes produce odd results; that can be a problem when the oddness lies in the observable behavior of an embedded agent.

It has previously been established that example-guided evolution can help solve a problem better according to some conventional performance metric. This dissertation shows that it can also improve performance according to other considerations, such as eliminating erratic behavior in embedded agents trained by evolution, or causing such agents to show discipline in their behavior – to act according to a recognizable doctrine with an appearance of intelligence, whether that doctrine is more effective or less so.

Exploiting Sensor Symmetries

When machine learning is used to train the mapping between sense and response in an agent, the responses to various reasonable transformations of the sensory inputs must often be learned independently. Such independence can result in inconsistencies in an agent's behavior. For example, an agent may learn to respond optimally to a threat or opportunity to its left, but sub-optimally to the same threat or opportunity to its right.

When examples are available to assist with an agent's training, simple transformations of the examples can be used to generate artificial examples that are every bit as legitimate as the originals. Such artificial examples can improve the agent's training by the mere fact of providing more training examples, and can also expose the agent-in-training to the transformational regularities of its target environment, resulting in learned behaviors that are more consistent across those transforma-

tions. For embedded game agents such consistencies will be seen as signs of intelligence; observers will not conclude that the agents respond properly only to specific situations that they have been programmed or trained for.

Managed Randomization of Behavior

While self-consistency is a desideratum for most intelligent agents, excessive predictability in embedded game agents can be seen as a defect. Game players soon learn to exploit excessive predictability in AI-controlled opponents, and thereafter gameplay becomes a dull, mechanistic optimization problem – the illusion of competing against an intelligent opponent is dispelled.

When an agent's higher-order behavior is built up from the concatenation of atomic actions, some degree of unpredictability can be introduced by making the choice of atomic actions stochastic. Ordinarily that might be expected to result in poor performance by the agent. However, training methods such as stochastic sharpening can produce controllers that generate a utility confidence rating for the various available actions; stochastic behavior can then be distributed according to those ratings, resulting in less degradation of performance. For embedded game agents the reduced predictability – reduced exploitability – of such stochastic behavior may provide advantages that offset the divergence from the notionally optimal behavior, and the agents will no longer be seen as mindless automatons.

Typology of Human-Biased Genetic Algorithms

A typology can clarify our thinking on a topic, reveal non-obvious similarities and differences, and expose gaps in coverage. This dissertation provides a typology of mechanisms for introducing human biases into the outcome of genetic algorithms. The typology can be expected to have a positive influence on researchers who explore such mechanisms, by clarifying the context of their work and inducing more regularity into their terminology. It can also be hoped that the typology will reveal biasing opportunities that have not yet been exploited, inspiring new avenues of research and providing powerful new mechanisms that can be applied to difficult problems – such as training embedded game agents to exhibit visibly intelligent behavior.

10.2 Conclusion

The programmed artificial intelligence in modern computer games often leaves observers unsatisfied; the AI-in-games community thinks it will soon be able to produce better results with machine learning. However, blind optimization will not always produce the desired result. When a game features embedded intelligent agents whose actions are observable, players will expect them to display behavior that is flexible, adaptive, disciplined, and self-consistent, though not excessively predictable: such are the visible signs of intelligence. This dissertation lays a foundation for training agents that display such characteristics when using neuroevolution and related technologies for applications to strategy games or simulators. As the capabilities of research-grade artificial intelli-

gence converge with the needs of the commercial gaming industry, this foundation can be expected to expand into a broad and productive area of research that supports a commercially important genre of real-world applications.

Bibliography

Agogino, A., Stanley, K., and Miikkulainen, R. (2000). Online interactive neuro-evolution. *Neural Processing Letters*, 11:29–38.

Agre, P. E., and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In Forbus, K., and Shrobe, H., editors, *Proceedings of the Sixth National Conference on Artificial Intelligence*. San Francisco, CA: Morgan Kaufmann.

Bain, M., and Sammut, C. (1999). A framework for behavioural cloning. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence 15: Intelligent Agents*, chapter 6, 103–129. Oxford University Press.

Balch, T. (1997). Learning roles: Behavioral diversity in robot teams. In Sen, S., editor, *Multiagent Learning: Papers from the AAAI Workshop. Technical Report WS-97-03*, 7–12. Menlo Park, CA: American Association for Artificial Intelligence.

Balch, T. (1998). *Behavioral Diversity in Learning Robot Teams*. PhD thesis, Georgia Institute of Technology. Technical Report GIT-CC-98-25.

Baldwin, J. M. (1896). A new factor in evolution. *The American Naturalist*, 30:441–451, 536–553.

Bates, J. (1997). The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125.

Biles, J. A. (1994). GenJam: A genetic algorithm for generating jazz solos. In *Proceedings of the International Computer Music Conference*, 131–137. Aarhaus, Denmark.

Biles, J. A. (2006). *GenJam*. http://www.it.rit.edu/∼jab/GenJam.html. Retrieved June 21, 2006.

Breiman, L. (1994). Bagging predictors. Technical report, University of California, Berkeley, CA. Department of Statistics Technical Report No. 421.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

Brooks, R. A., Breazeal (Ferrell), C., Irie, R., Kemp, C. C., Marjanović, M., Scassellati, B., and Williamson, M. M. (1998). Alternative essences of intelligence. In *Proceedings of the 15th National Conference on Artificial Intelligence*, 961–976. Menlo Park, CA: AAAI Press.

Bryant, B. D. (2006). Virtual bagging for an evolved agent controller. Manuscript in preparation.

Bryant, B. D., and Miikkulainen, R. (2003). Neuroevolution for adaptive teams. In *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, vol. 3, 2194–2201. Piscataway, NJ: IEEE.

Bryant, B. D., and Miikkulainen, R. (2006a). Evolving stochastic controller networks for intelligent game agents. In *Proceedings of the 2006 Congress on Evolutionary Computation (CEC 2006)*, 3752–3759. Piscataway, NJ: IEEE.

Bryant, B. D., and Miikkulainen, R. (2006b). Exploiting sensor symmetries in example-based training for intelligent agents. In Louis, S. J., and Kendall, G., editors, *Proceeedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG'06)*, 90–97. Piscataway, NJ: IEEE.

Chong, S. Y., Ku, D. C., Lim, H. S., Tan, M. K., and White, J. D. (2003). Evolved neural networks learning Othello strategies. In *Proceedings of the 2003 Congress on Evolutionary Computation*, vol. 3, 2222–2229. Piscataway, NJ: IEEE.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314.

Dean, T., Allen, J., and Aloimonds, Y. (1995). *Artificial Intelligence: Theory and Practice*. Redwood City, California: Benjamin/Cummings.

Dorigo, M., and Colombetti, M. (1993). Robot Shaping: Developing Situated Agents through Learning. Technical report, International Computer Science Institute, Berkeley, CA. Technical Report TR-92-040 Revised.

D'Silva, T., Janik, R., Chrien, M., Stanley, K. O., and Miikkulainen, R. (2005). Retaining learned behavior during real-time neuroevolution. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005)*. American Association for Artificial Intelligence.

Durfee, E. H. (1995). Blissful ignorance: Knowing just enough to coordinate well. In *Proceedings of the First International Conference on Multi-Agent Systems*, 406–413.

Ellman, T. (1993). Abstraction via approximate symmetry. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 916–921.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

English, T. M. (2001). Learning to focus selectively on possible lines of play in checkers. In *Proceedings of the 2001 Congress on Evolutionary Computation*, vol. 2, 1019–1024. Piscataway, NJ: IEEE.

Fogel, D. B. (1999). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Piscataway, NJ: IEEE. Second edition.

Fogel, D. B. (2001). *Blondie24: Playing at the Edge of AI*. San Francisco, CA: Morgan Kaufmann.

Fogel, D. B., Hays, T. J., Hahn, S. L., and Quon, J. (2005). Further evolution of a self-learning chess program. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. Piscataway, NJ: IEEE.

Fox, M., and Long, D. (1999). The detection and exploitation of symmetry in planning problems. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 956–961.

Fürnkranz, J. (2001). Machine learning in games: A survey. In Fürnkranz, J., and Kubat, M., editors, *Machines That Learn to Play Games*, chapter 2, 11–59. Huntington, NY: Nova Science Publishers.

Gallagher, M., and Ryan, A. (2003). Learning to play Pac-Man: An evolutionary, rule-based approach. In *Proceedings of the 2003 Congress on Evolutionary Computation*, vol. 4, 2462–2469. Piscataway, NJ: IEEE.

Gardner, M. (1962). How to build a game-learning machine and then teach it to play and to win. *Scientific American*, 206(3):138–144.

Giles, C. L., and Lawrence, S. (1997). Presenting and analyzing the results of AI experiments: Data averaging and data snooping. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 362–367. Menlo Park, California: AAAI Press.

Gold, A. (2005). Academic AI and video games: A case study of incorporating innovative academic research into a video game prototype. In Kendall, G., and Lucas, S., editors, *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 141–148. Piscataway, NJ: IEEE.

Goldberg, D., and Matarić, M. J. (1997). Interference as a tool for designing and evaluating multi-robot controllers. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 637–642.

Gomez, F. (2003). *Robust Non-Linear Control Through Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin.

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

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

Gomez, F., and Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1356–1361. San Francisco, CA: Morgan Kaufmann.

Gomez, F., and Miikkulainen, R. (2003). Active guidance for a finless rocket using neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2084–2095. San Francisco, CA: Morgan Kaufmann.

Gratch, J., and Marsella, S. (2004). A domain independent framework for modeling emotion. *Journal of Cognitive Systems Research*, 5(4):269–306.

Gratch, J., and Marsella, S. (2006). Evaluating a computational model of emotion. *Journal of Autonomous Agents and Multiagent Systems*, 11(1):23–43.

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

Hamilton, S., and Garber, L. (1997). Deep blue's hardware-software synergy. *Computer*, 30(10):29–35.

Hayes-Roth, B., Johnson, V., van Gent, R., and Wescourt, K. (1999). Staffing the web with interactive characters. *Communications of the ACM*, 42(3):103–105.

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

Haynes, T. D., and Sen, S. (1997). Co-adaptation in a team. *International Journal of Computational Intelligence and Organizations*, 1:231–233.

Hinton, G. E., and Nowlan, S. J. (1987). How learning can guide evolution. *Complex Systems*, 1:495–502.

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.

Igel, C. (2003). Neuroevolution for reinforcement learning using evolution strategies. In *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, vol. 4, 2588–2595. Piscataway, NJ: IEEE.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3:79–87.

Johnson, W. L., Narayanan, S., Whitney, R., Das, R., Bulut, M., and Labore, C. (2002). Limited domain synthesis of expressive military speech for animated characters. In *Proceedings of 2002 IEEE Workshop on Speech Synthesis*, 163–166.

Jordan, M. I., and Jacobs, R. A. (1992). Hierarchies of adaptive experts. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems 4*, 985–992. San Francisco, CA: Morgan Kaufmann.

Kendall, G., and Whitwell, G. (2001). An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation*, vol. 2, 995–1002. Piscataway, NJ: IEEE.

Kendall, G., Yaakob, R., and Hingston, P. (2004). An investigation of an evolutionary approach to the opening of Go. In *Proceedings of the 2004 Congress on Evolutionary Computation*, vol. 2, 2052–2059. Piscataway, NJ: IEEE.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). RoboCup: The robot world cup initiative. In *Proceedings of The First International Conference on Autonomous Agents*. ACM Press.

Ku, K. W. C., Mak, M. W., and Siu, W. C. (2000). A study of the Lamarckian evolution of recurrent neural networks. *IEEE Transactions on Evolutionary Computation*, 4:31–42.

Kuhlmann, G., Stone, P., Mooney, R., and Shavlik, J. (2004). Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. In *The AAAI-2004 Workshop on Supervisory Control of Learning and Adaptive Systems*.

Laird, J. E., and van Lent, M. (2000). Human-level AI's killer application: Interactive computer games. In *Proceedings of the 17th National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Lamarck, J.-B. (1809). *Philosophie zoologique*. Paris.

Lawrence, S., Back, A. D., Tsoi, A. C., and Giles, C. L. (1997). On the distribution of performance from multiple neural-network trials. *IEEE Transactions on Neural Networks*, 8:1507–1517.

Louis, S. J., and Miles, C. (2005). Playing to learn: Case-injected genetic algorithms for learning to play computer games. *IEEE Transactions on Evolutionary Computation*, 9(6):669–681.

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

Lucas, S. M. (2004). Cellz: a simple dynamic game for testing evolutionary algorithms. In *Proceeedings of the 2004 Congress on Evolutionary Computation (CEC 2004)*, 1007–1014. Piscataway, NJ: IEEE.

Lucas, S. M. (2005). Evolving a neural network location evaluator to play ms. pac-man. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. Piscataway, NJ: IEEE.

Maclin, R., and Shavlik, J. W. (1996). Creating advice-taking reinforcement learners. *Machine Learning*, 22(1-3):251–281.

Matarić, M. J. (1995). Designing and understanding adaptive group behavior.

Matsumoto, M., and Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30. Retrieved August 4, 2002.

Maudlin, M. L., Jacobson, G., Appel, A., and Hamey, L. (1984). ROG-O-MATIC: A belligerent expert system. In *Proceedings of the Fifth National Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI-84)*.

Melville, P. (2006). Personal communication.

Melville, P., and Mooney, R. J. (2004). Creating diversity in ensembles using artificial data. *Information Fusion*, 6(1):99–111.

Miikkulainen, R. (1996). Subsymbolic case-role analysis of sentences with embedded clauses. *Cognitive Science*, 20:47–73.

Minsky, M., and Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press. Expanded edition.

Mitchell, T. M. (1997). *Machine Learning*. New York, NY: McGraw-Hill.

Montana, D. J., and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 762–767. San Francisco, CA: Morgan Kaufmann.

Moriarty, D., and Miikkulainen, R. (1995a). Learning sequential decision tasks. Technical Report AI95-229, Department of Computer Sciences, The University of Texas at Austin.

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

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

Moriarty, D. E., and Miikkulainen, R. (1996). Evolving obstacle avoidance behavior in a robot arm. In Maes, P., Mataric, M. J., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, 468–475. Cambridge, MA: MIT Press.

Ng, A. Y., and Russell, S. (2000). Algorithms for inverse reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, 663–670. San Francisco: Morgan Kaufmann.

Nicolescu, M. N. (2003). *A Fremework for Learning From Demonstration, Generalization and Practiced in Human-Robot Domains*. PhD thesis, University of Southern California.

Pagano, R. R. (1986). *Understanding Statistics in the Behavioral Sciences*. St. Paul, MN: West Publishing. Second edition.

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

Peterson, T. S., Owens, N. E., and Carroll, J. L. (2001). Towards automatic shaping in robot navigation. In *Proc. of International Conference on Robotics and Automation*.

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

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

Price, B., and Boutilier, C. (2003). Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research*, (19):569–629.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. San Francisco, CA: Morgan Kaufmann.

R Development Core Team (2004). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria.

Ramsey, C. L., and Grefenstette, J. J. (1993). Case-based initialization of genetic algorithms. In Forrest, S., editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, 84–91. San Mateo, CA: Morgan Kaufmann.

Randløv, J. (2000). Shaping in reinforcement learning by changing the physics of the problem. In *Proc. 17th International Conf. on Machine Learning*, 767–774. Morgan Kaufmann, San Francisco, CA.

Renals, S., and Morgan, N. (1992). Connectionist probability estimation in HMM speech recognition. Technical Report TR-92-081, International Computer Science Institute, Berkeley, CA.

Rich, E., and Knight, K. (1991). *Artificial Intelligence*. New York: McGraw-Hill. Second edition.

Richards, N., Moriarty, D., McQuesten, P., and Miikkulainen, R. (1997). Evolving neural networks to play Go. In Bäck, T., editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA-97, East Lansing, MI)*, 768–775. San Francisco, CA: Morgan Kaufmann.

Richards, N., Moriarty, D., and Miikkulainen, R. (1998). Evolving neural networks to play Go. *Applied Intelligence*, 8:85–96.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408. Reprinted in **?**.

Rowbottom, A. (1998). *Organic, Genetic, and Evolutionary Art*. http://snaffle.users.netlink.co.uk/form/evolutio.html. Retrieved June 21, 2006.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). 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.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning representations by back-propagating errors. *Nature*, 323:533–536.

Sammut, C., Hurst, S., Kedzier, D., and Michie, D. (1992). Learning to fly. In *Proceedings of the Ninth International Conference on Machine Learning*, 385–393. Aberdeen, UK: Morgan Kaufmann.

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

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 Whitley, D., and Schaffer, J., editors, *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1–37. Los Alamitos, CA: IEEE Computer Society Press.

Schultz, A. C., and Grefenstette, J. J. (1990). Improving tactical plans with genetic algorithms. In *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence*, 328–334. IEEE Computer Society Press.

Schultz, A. C., and Grefenstette, J. J. (1992). Using a genetic algorithm to learn behaviors for autonomous vehicles. In *Proceedings of the of the 1992 AIAA Guidance, Navigation and Control Conference*, 739–749.

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

Siegelman, H., and Sontag, E. D. (1991). On the computational power of neural nets. Technical Report SYCON-91-08, Rutgers Center for Systems and Control, Rutgers University, New Brunswick, NJ.

Siegelmann, H. T., and Sontag, E. D. (1994). Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360.

Skinner, B. F. (1938). *The Behavior of Organisms: An Experimental Analysis*. New York: Appleton-Century-Crofts.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2003). Evolving adaptive neural networks with and without adaptive synapses. In *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, vol. 4, 2557–2564. Piscataway, NJ: IEEE.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005a). Evolving neural network agents in the NERO video game. In Kendall, G., and Lucas, S., editors, *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*, 182–189. Piscataway, NJ: IEEE.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005b). Evolving neural network agents in the NERO video game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005c). Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668.

Stanley, K. O., and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127.

Stanley, K. O., and Miikkulainen, R. (2004a). Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100.

Stanley, K. O., and Miikkulainen, R. (2004b). Evolving a roving eye for Go. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*. Berlin: Springer Verlag.

Stone, P., and Veloso, M. (2000a). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robotics*, 8(3):345–383.

Stone, P., and Veloso, M. M. (2000b). Layered learning. In *Machine Learning: ECML 2000, 11th European Conference on Machine Learning, Barcelona, Catalonia, Spain, May 31 - June 2, 2000, Proceedings*, vol. 1810, 369–381. Springer, Berlin.

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

Sycara, K. (1998). Multiagent systems. *AI Magazine*, 19(2):79–92.

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program achieves master-level play. *Neural Computation*, 6:215–219.

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

Togelius, J., and Lucas, S. M. (2005). Forcing neurocontrollers to exploit sensory symmetry through hard-wired modlarity in the game of Cellz. In Kendall, G., and Lucas, S., editors, *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 37–43. Piscataway, NJ: IEEE.

Towell, G. G., and Shavlik, J. W. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70:119–165.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, LIX(236):433–460.

van Lent, M., and Laird, J. E. (2001). Learning procedural knowledge through observation. In *Proceedings of the International Conference on Knowledge Capture*, 179–186. New York: ACM.

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. In *Proceedings of the IEEE*, vol. 78, 1550–1560. Piscataway, NJ: IEEE.

Whiteson, S., and Stone, P. (2006). Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917.

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

Wilkins, J. S. (2006). Personal communication.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

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

Yong, C. H., Stanley, K. O., and Miikkulainen, R. (2005). Incorporating advice into evolution of neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005): Late Breaking Papers*.

Zinkevich, M., and Balch, T. R. (2001). Symmetry in markov decision processes and its implications for single agent and multiagent learning. In *Machine Learning: Proceedings of the 18th Annual Conference*.

Vita

Bobby Don Bryant was born to George Crosby Bryant and Iantha Omeda Bryant at Searcy, Arkansas, on July 23, 1954. He graduated from Brazoswood High School at Clute, Texas in 1972, and pursued undergraduate work at Abilene Christian College and Temple University before obtaining an Associate of Applied Science (AAS) in Computer Technology at Brazosport College in 1988, having also worked as a construction worker and a programmer/analyst in the interim. In 1993 he resumed studies at the University of Houston, and graduated with a Bachelor of Arts *summa cum laude* in Classical Studies in August of 1995. He then obtained a Master of Science in Computer Science (MSCS) at The University of Texas at Austin in May of 1999.

While pursuing the Doctor of Philosophy, Bobby worked as a Graduate Research Assistant, Teaching Assistant, and Assistant Instructor at the Department of Computer Sciences at UT-Austin, and consulted on the *NERO* and *Digital Warrior* projects at the Digital Media Collaboratory at the IC^2 Institute at UT-Austin. He has presented papers at the Congress on Evolutionary Computing and at the IEEE Symposium on Computational Intelligence and Games, where he won the *Best Student Paper* award in 2006. He has participated as a co-author for additional conference and journal papers regarding the *NERO* project, including the paper that won the *Best Paper* award at the 2005 IEEE Symposium on Computational Intelligence and Games. He has also appeared as an invited speaker at the Digital Media Collaboratory's *GameDev* conference, and at the University of Nevada, Reno.

Permanent Address: 104 Sage, Lake Jackson, Texas 77566 USA
bdbryant@cs.utexas.edu
http://nn.cs.utexas.edu/keyword?bdbryant

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

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American

Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.