Computational Intelligence in Games[*]

Risto Miikkulainen, Bobby D. Bryant, Ryan Cornelius, Igor V. Karpov, Kenneth O. Stanley, and Chern Han Yong

Abstract

Video games provide an opportunity and challenge for the soft computational intelligence methods like the symbolic games did for "good old-fashioned artificial intelligence." This article reviews the achievements and future prospects of one particular approach, that of evolving neural networks, or neuroevolution. This approach can be used to construct adaptive characters in existing video games, and it can serve as a foundation for a new genre of games based on machine learning. Evolution can be guided by human knowledge, allowing the designer to control the kinds of solutions that emerge and encouraging behaviors that appear visibly intelligent to the human player. Such techniques may allow building video games that are more engaging and entertaining than current games, and those that can serve as training environments for people. Techniques developed in these games may also be widely applicable in other fields, such as robotics, resource optimization, and intelligent assistants.

1 Introduction

Games have long been a popular area of artificial intelligence (AI) research, and for a good reason. They are challenging yet easy to formalize, making it possible to develop new AI methods, measure how well they are working, and demonstrate that machines are capable of impressive behavior generally thought to require intelligence without putting human lives or property at risk.

Most of the research so far has focused on games that can be described in a compact form using symbolic representations, such as board and card games. The so-called "good old-fashioned artificial intelligence" (GOFAI; Haugeland 1985) techniques work well with them, and to a large extent, such techniques were developed for such games. They have led to remarkable successes, such as the checkers program Chinook (Schaeffer 1997) that became the world champion in 1994, and the chess program Deep Blue (Campbell et al. 2002) that defeated the world champion in 1997, gaining significant attention to AI.

Since the 1990s, the field of gaming has changed tremendously. Inexpensive yet powerful computer hardware has made it possible to simulate complex physical environments, resulting in an explosion of the video game industry. From modest beginnings in the 1960s (Baer 2005), the entertainment software sales have expanded to $25.4 billion worldwide in 2004 (Crandall and Sidak 2006). Video games have become a facet of many people's lives and the market continues to expand.

Curiously, this expansion has involved little AI research. Many video games utilize no AI techniques, and those that do are usually based on relatively standard, labor-intensive scripting and authoring methods. The reason is that video games are very different from the symbolic games. There are often many agents involved, embedded in a simulated physical environment where they interact through sensors and effectors

[*]To appear in Yen, G. Y. and Fogel, D. B. (editors), *Computational Intelligence: Principles and Practice*, IEEE Computational Intelligence Society, 2006.

that take on numerical rather than symbolic values. To be effective, the agents have to integrate noisy input from many sensors, and they have to react quickly and change their behavior during the game. The techniques that have been developed for and with symbolic games are not well suited for video games.

In contrast, soft computational intelligence (CI) techniques such as neural networks, evolutionary computing, and fuzzy systems are very well suited for video games. They excel in exactly the kinds of fast, noisy, numerical, statistical, and changing domains that today's video games provide. Therefore, video games constitute an opportunity similar to that of the symbolic games for GOFAI in 1980s and 1990s: an opportunity to develop and test CI techniques, and an opportunity to transfer the technology to industry.

Much research is already being done with CI in video games, as demonstrated by e.g. the CIG and AIIDE symposia, as well as recent gaming special sessions and issues in conferences and journals (Fogel et al. 2005; Kendall and Lucas 2005; Laird and van Lent 2006; Louis and Kendall 2006; Young and Laird 2005; see Lucas and Kendall 2006 for a review). The goal of this article is to review the achievements and future prospects of one particular approach, that of evolving neural networks, or neuroevolution (NE). Although NE methods were originally developed primarily for robotics and control tasks, and were first applied to symbolic games, the technique is particularly well suited for video games. It can be used to implement modifications to existing games, or be used as a foundation for a new genre of games where machine learning plays a central role. The main challenges are in guiding the evolution using human knowledge, and in achieving behavior that is not only successful, but visibly intelligent to a human observer. After a review of AI in video games and current neuroevolution technology, these opportunities and challenges will be discussed in this article, using several implemented systems as examples.

2 AI in Video Games

One of the main challenges for AI is to create intelligent agents that adapt, i.e. change their behavior based on interactions with the environment, becoming more proficient in their tasks over time, and adapting to new situations as they occur. Such ability is crucial for deploying robots in human environments, as well as for various software agents that live in the Internet or serve as human assistants or collaborators.

While general such systems are still beyond current technology, they are already possible in special cases. In particular, video games provide complex artificial environments that can be controlled, and carry perhaps the least risk to human life of any real-world application (Laird and van Lent 2000). On the other hand, such games are an important part of human activity, with millions of people spending countless hours on them. Machine learning can make video games more interesting and decrease their production costs (Fogel et al. 2004b). In the long run, such technology might also make it possible to train humans realistically in simulated adaptive environments. Video gaming is therefore an important application of AI on its own right, and an excellent platform for research in intelligent adaptive agents.

Current video games include a variety of high-realism simulations of human-level control tasks, such as navigation, combat, team and individual tactics and strategy. Some of these simulations involve traditional AI techniques such as scripts, rules, and planning (Agre and Chapman 1987; Maudlin et al. 1984). A large fraction of AI development in the industry is devoted to path-finding algorithms such as A*-search and simple behaviors built using finite state machines. The AI is used to control the behavior of the non-player-characters (NPCs), i.e. the autonomous computer-controlled agents in the game. Although such agents can exhibit impressive behaviors, they are often repetitive and inflexible. Indeed, a large part of the gameplay in many games is figuring out what the AI is programmed to do, and learning to defeat it.

More recently, machine learning techniques have begun to appear in video games. This trend follows a long history of learning in board games, originating from Samuel's (1959) checkers program that was based on a method similar to temporal difference learning (Sutton 1988), followed by various learning methods applied to tic-tac-toe (Gardner 1962; Michie 1961), backgammon (Pollack et al. 1996; Tesauro 1994; Tesauro and Sejnowski 1987), go (Richards et al. 1997; Stanley and Miikkulainen 2004b), othello (Moriarty 1997; Yoshioka et al. 1998), and checkers (Fogel 2001; Fogel et al. 2004a; see Fürnkranz 2001 for a survey). Many of these learning methods can be applied to video games as well (Fogel et al. 2004b; Laird and van Lent 2000). For example, Fogel et al. (2004b) trained teams of tanks and robots to fight each other using a competitive coevolution system, and Spronck (2005) trained agents in a computer role-playing game using dynamic scripting. Others have trained agents to fight in first and third-person shooter games (Cole et al. 2004; Geisler 2002; Hong and Cho 2004). ML techniques have also been applied to other video game genres from Pac-Man (Gallagher and Ryan 2003; Lucas 2005) to strategy games (Bryant and Miikkulainen 2003; Revello and McCartney 2002; Yannakakis et al. 2004).

Still, there is very little adaptation in current commercial video games. There may be several reasons for why this is the case. One is that high-end game applications have traditionally pushed hardware systems to their limits and simply did not have the resources to perform online learning. Hardware limitations are becoming less of a problem with the availability of cheap processing power and true parallelism. Second, methods developed in the academic machine learning research may not apply directly to gaming. Whereas the goal of an AI researcher is to build a system that solves difficult tasks (by whatever means necessary), the goal of a game developer is to build agents that act similarly to humans, i.e. that are visibly intelligent. A third problem is that with adaptation, the game content becomes unpredictable. Because correct behaviors are not known, learning must be done through exploration. As a result, agents sometimes learn idiosyncratic behaviors that make the gaming experience unsatisfying.

Therefore, in order to make adaptation viable, the learning method needs to be powerful, flexible, and reliable; it needs to discover solutions on its own (without targets), but also allow guiding it with human knowledge. For example, while traditional reinforcement learning (RL) techniques such as temporal differences and Q-learning have many of these properties in simple domains (Kaelbling et al. 1996; Santamaria et al. 1998; Sutton and Barto 1998; Watkins and Dayan 1992), they do not currently scale up well to the complexity of video games. In particular, video games have large state and action spaces and require diverse behaviors, consistent individual behaviors, fast adaptation, and memory of past states (Stanley et al. 2005a). These are difficult issues in the standard RL framework; however, they can be solved naturally in the neuroevolution approach.

3 Neuroevolution Methods

In neuroevolution, genetic or evolutionary algorithms are used to evolve neural network weights and structures. Neural networks perform statistical pattern transformation and generalization, and evolutionary adaptation allows learning the networks without explicit targets, even with extremely sparse reinforcement. The approach is particularly well-suited for video games: NE works well in high-dimensional spaces, diverse populations can be maintained, individual networks behave consistently, adaptation can take place in real time, and memory can be implemented through recurrency (Stanley et al. 2005a).

Several methods exist for evolving neural networks. The most straightforward way is to form the genetic encoding for the network by concatenating the numerical values for its weights (either binary or floating point), and to evolve a population of such encodings using crossover and mutation (Wieland 1991; Yao 1999). Although this approach is easy to implement and practical in many domains, more sophisticated methods can solve much harder problems. As background for the later sections, two such methods will be reviewed in this section. The first one, ESP, is based on the idea of subgoaling, i.e. evolving neurons instead of networks. The other, NEAT, is based on complexification, i.e. gradually adding more structure to the networks.

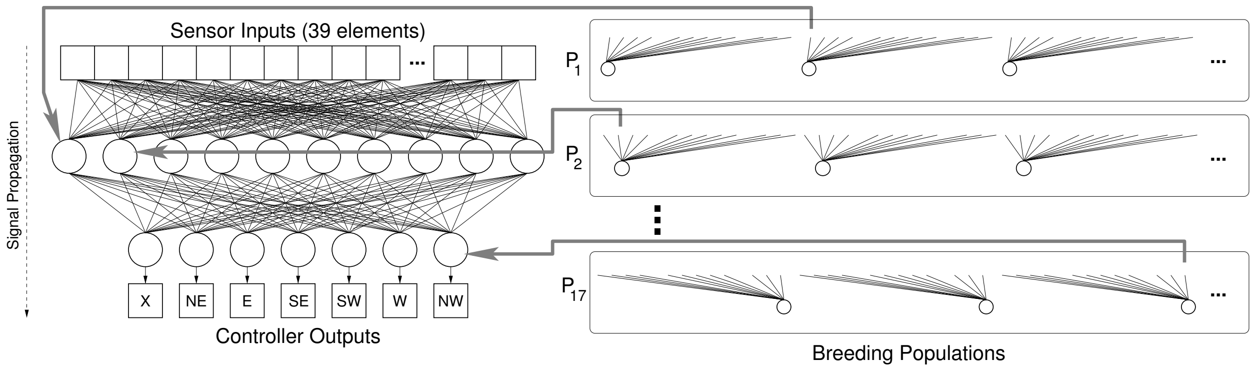


Figure 1: **Neuroevolution with ESP..** In ESP, a separate breeding population P_i is maintained for each neuron in a network, in this case $\{P_1, P_2, ..., P_{17}\}$. Networks are assembled by drawing one chromosome at random from the population associated with each position in the network and instantiating the neurons the chromosomes represent (gray arrows). The resulting network is tested in the environment and its fitness is ascribed back to each of the chromosomes that specified its neurons. The process is repeated until a fitness score has been determined for every chromosome in all of the sub-populations, at which time an evolutionary generation is complete. Each population is then updated by selective breeding with random crossovers and mutations, independently of the other populations. As generations pass, the sub-populations coevolve to produce chromosomes describing neurons that work well with the others in the network.

3.1 Enforced Sub-Populations (ESP)

The distinctive feature of the ESP neuroevolution algorithm (Gomez and Miikkulainen 1997, 1999) is that the “chromosome” representations manipulated by the genetic algorithm represent individual neurons rather than entire networks (see Agogino et al. 2005; Moriarty and Miikkulainen 1997; Potter and Jong 2000 for other implementations of this idea). Moreover, a separate breeding population is maintained for the position of each neuron in the complete network (figure 1). Breeding is only done within each population, so that each will evolve neurons specific to one position in the network. During training, the populations coevolve functionality complementary to one another; as a result the algorithm is able to converge quickly on solutions to problems that were formerly considered difficult (Gomez 2003).

For fitness evaluations, one chromosome is drawn at random from each population and the neurons represented by the selected chromosomes are assembled into a network. The network is then evaluated at its assigned task, such as controlling an agent during one complete game. The fitness value that is measured for the network—the game score—is recorded for each neuron that participated. This scoring method is somewhat noisy because the “real” fitness of a neuron can be brought down by the bad luck of being chosen to participate in a network with other incompetent neurons, or it can be brought up by being chosen for network with superior neurons. To minimize such evaluation noise, each neuron is tested several times in each generation, participating in a network with a different random selection of peers each time. These scores are then averaged to approximate the neuron’s unknown “real” fitness.

Within each sub-population, chromosomes are selected for breeding favoring those with the best fitness scores, but still allowing selection of less-fit chromosomes with low probability. Each neuronal chromosome lists a neuron’s input weights as floating point numbers in a flat array. During breeding, 1-point or 2-point crossover are used to create offspring, and point mutations (random weight changes whose probability decreases exponentially with change magnitude) are applied with a low probability at each position in the resulting chromosome. This method is used in the experiments described in this article; other selection methods and genetic operators are possible as well, as well as delta-coding to encourage diversity (Gomez

2003).

ESP has previously been used for training continuous-state controllers for pole balancing and other standard benchmark tasks. It has also been applied for controlling a finless rocket and learning effective behaviors for single predators and predator teams and for robotic keepaway soccer players (Gomez 2003; Gomez and Miikkulainen 1997, 2003; Whiteson et al. 2005a; Yong and Miikkulainen 2001). Its power is based on *subgoaling*, i.e. breaking up the task of finding a good solution network to several simpler problems of finding good neurons. The neurons that work well together get high fitness, and gradually the subpopulations begin to optimize neurons for different roles in the network. The networks can be densely connected and recurrent, making the approach especially strong in challenging nonlinear control tasks.

3.2 Neuroevolution of Augmenting Topologies (NEAT)

Whereas the work with ESP originally focused on controlling nonlinear systems, the NeuroEvolution of Augmenting Topologies (NEAT) method was originally developed for learning behavioral strategies (Stanley 2003; Stanley and Miikkulainen 2002, 2004a). Like ESP, the neural networks control agents that select actions based on their sensory inputs. However, while previous neuroevolution methods evolved either fixed topology networks (Gomez and Miikkulainen 1999; Moriarty and Miikkulainen 1996; Saravanan and Fogel 1995; Whitley et al. 1993; Wieland 1991), or arbitrary random-topology networks (Angeline et al. 1993; Bongard and Pfeifer 2001; Braun and Weisbrod 1993; Fullmer and Miikkulainen 1992; Gruau et al. 1996; Hornby and Pollack 2002; Opitz and Shavlik 1997; Pujol and Poli 1997; Yao and Liu 1996; Zhang and Muhlenbein 1993), NEAT is the first to begin evolution with a population of small, simple networks and *complexify* those networks over generations, leading to increasingly sophisticated behaviors. In prior work, NEAT has been successfully applied e.g. to pole balancing, competing simulated robots, automobile warning systems, and the game of Go (Stanley et al. 2005c; Stanley and Miikkulainen 2002, 2004a,b).

NEAT is based on three key ideas. First, in order to allow neural network structures to increase in complexity over generations, a method is needed to keep track of which gene is which. Otherwise, it is not clear in later generations which individual is compatible with which, or how their genes should be combined to produce offspring. NEAT solves this problem by assigning a unique historical marking to every new piece of network structure that appears through a structural mutation. The historical marking is a number assigned to each gene corresponding to its order of appearance over the course of evolution. The numbers are inherited during crossover unchanged, and allow NEAT to perform crossover without the need for expensive topological analysis. That way, genomes of different organizations and sizes stay compatible throughout evolution, solving the previously open problem of matching different topologies (Radcliffe 1993) in an evolving population.

Second, NEAT speciates the population, so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before competing with other niches in the population. NEAT uses the historical markings on genes to determine to which species different individuals belong.

Third, unlike other systems that evolve network topologies and weights NEAT begins with a uniform population of simple networks with no hidden nodes. New structure is introduced incrementally as structural mutations occur, and only those structures survive that are found to be useful through fitness evaluations. This way, NEAT searches through a minimal number of weight dimensions and finds the appropriate complexity level for the problem. This process of complexification has important implications for search: While it may not be practical to find a solution in a high-dimensional space by searching in that space directly, it may be possible to find it by first searching in lower dimensional spaces and complexifying the best solutions into the high-dimensional space.

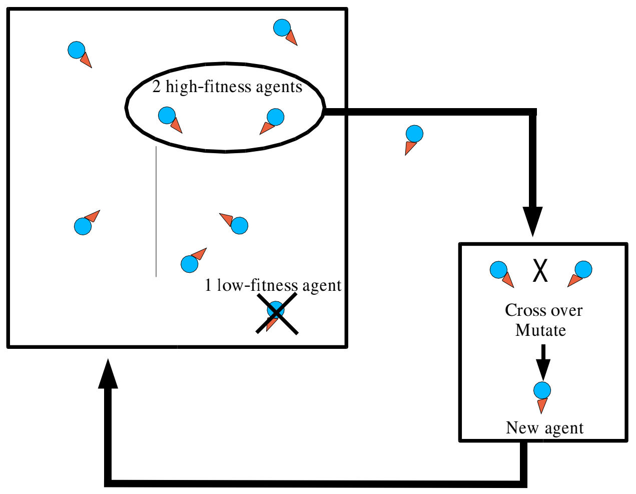


Figure 2: **The Main Replacement Cycle in rtNEAT.** Robot game agents (represented as small circles) are depicted playing a game in the large box. Every few ticks, two high-fitness robots are selected to produce an offspring that replaces one of the robots with low fitness. This cycle of replacement operates continually throughout the game, creating a constant turnover of new behaviors.

As is usual in evolutionary algorithms, the entire population is replaced at each generation in NEAT. However, in a real time game or a simulation, such a step would look incongruous since every agent's behavior would change at once. In addition, behaviors would remain static during the large gaps between generations. Therefore, in order to apply NEAT to video games, a real-time version of it called rtNEAT was created. In rtNEAT, a single individual is replaced every few game ticks. One of the worst individuals is removed and replaced with a child of parents chosen from among the best. This cycle of removal and replacement happens continually throughout the game and is largely invisible to the player (figure 2).

This replacement cycle is tricky to implement because the usual NEAT dynamics, i.e. protection of innovation through speciation and complexification, are based on equations that assume generational replacement. In rtNEAT, these equations are changed into probabilistic expressions that apply to a single reproduction event (Stanley et al. 2005a). The result is an algorithm that can evolve increasingly complex neural networks fast enough for a user to interact with evolution as it happens in real time.

3.3 Further Research on Neuroevolution Methods

The two methods described in this section, ESP and NEAT, have proven strong in various benchmark comparisons (Gomez et al. 2006). They have also been applied to a variety of domains (software packages for each these methods are available at `http://nn.cs.utexas.edu`). In many cases, they make applications possible that have not been even considered learning tasks before.

However, neuroevolution algorithms are an active area of research, and there are many areas in the future where progress can be made. For example, the idea of subgoaling can be taken one step further, to the level of individual weights (Gomez et al. 2006). Similarly, the complexification process can be extended include selection of input features (Whiteson et al. 2005b), or to utilize modular structures (Reisinger et al. 2004). Further, evolutionary strategies can be utilized in mutating the weights (Igel 2003), networks can be encoded indirectly and constructed in a developmental process (Angeline et al. 1993; Gruau et al. 1996; Mjolsness et al. 1989; Siddiqi and Lucas 1998; Stanley and Miikkulainen 2003; Yao 1999), and evolution can be combined with traditional neural network learning (Floreano and Urzelai 2000; Hinton and Nowlan

1987; McQuesten 2002; Nolfi et al. 1994; Stanley et al. 2003; Valsalam et al. 2005). Neuroevolution can be extended to multiple agents, using competitive or cooperative coevolution (Stanley and Miikkulainen 2004a; Whiteson et al. 2005a; Yong and Miikkulainen 2001), or through evolving the team directly (Bryant and Miikkulainen 2003; Yao 1999). Such extensions will be useful for many applications, including those in video games.

4 Adding NE to Existing Games

The most immediate opportunity for NE in video games is to build a "mod", or a new feature or extension, to an existing game. For example, a character that is scripted in the original game can be turned into an adapting agent, gradually learning and improving as the game goes on. Or, and entirely new dimension can be added to the game, such as an intelligent assistant or a tool, that changes as the player progresses through the game. Such mods can make the game more interesting and fun to play; on the other hand, they are easy and safe to implement since they do not change the original structure of the game. From the research point of view, mods allow testing ideas about embedded agents, adaptation, and interaction in a rich and realistic game environment.

In this section, progress in building a general game/learning system interface that would support implementing several different learning mods to several different games is reviewed. In particular, an NE module is developed for the TIELT interface framework, and tested in the Unreal Tournament (UT) video game.

4.1 General Game/Learning Interface

Many games allow mods as part of their design. In some cases the games are open source, such as many based on the Stratagus engine (Ponsen et al. 2005; `http://stratagus.sourceforge.net`). In others, such as UT, even though the source code for the game is not open, an interface has been defined through which the mod, and the learning system, can interact with it. In practice, the game and its mod interface have usually not been designed with learning in mind, and getting useful information from the engine for learning can be difficult. For example, sensor information may arrive with a delay, and the learning methods may have to be modified to take that into account. Also, different games may have different interfaces, making it difficult to run experiments across different game platforms.

One potential solution is to build a game/learning system interface, or middleware, that could be used across different games and different learning methods. For each game and each learning method, only one interface would need to be constructed to the middleware. Different learning methods could then be applied to different games, making it possible to compare methods and test how well they generalize across games.

One such system, TIELT (Testbed for Integration and Evaluation of Learning Techniques; Aha and Molineaux 2004; Molineaux and Aha 2005; `http://nrlsat.ittid.com`) is already under active development. TIELT is a Java application intended to connect a game engine to a decision system that learns about the game. The goal is to provide the researcher with a tool that simplifies the task of testing a general learner in several different environments.

TIELT consists of five modules that correspond to different areas of TIELT's functionality. The **Game Model** encapsulates the information about the game from a single player's perspective. The **Game Interface Model** describes how TIELT communicates with the game. The **Decision System Interface Model** describes interactions with the decision system. The **Agent Description** describes tasks performed by the system in order to play the game, and the **Experimental Methodology** module is used as an evaluation platform for particular games and learning techniques.

During integration, TIELT is connected with the game environment and with the learning system using the appropriate modules. The software allows the user to define the communication protocols between TIELT, the game engine and the learning system. TIELT also includes a visual scripting language that allows scripting of the game behavior and update rules. In this manner, TIELT provides the ability to connect environments and learners with arbitrary interfaces and rules into a cohesive learning system and to automate the evaluation of this system's performance.

However, TIELT was originally designed with relational learning systems in mind, not exploration-based systems like NE. An interesting question is, how well does it work with NE, and how such interface systems in general should be constructed to best support NE?

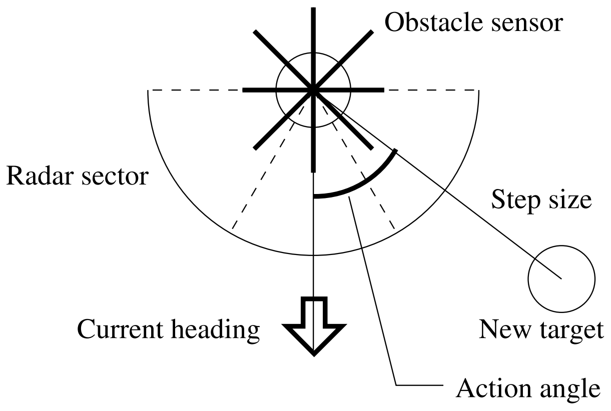
4.2 Implementing an NE Module for TIELT

In order to study this question experimentally, a NEAT NE module was developed for TIELT and tested with UT. At the highest level, the system consists of three parts: the UT game server, the TIELT integration platform, and the decision system based on NEAT. The game server simulates the environment for the learning agent. TIELT communicates with the environment and accumulates a state, which is then communicated to the decision system. The decision system selects an action in the environment and communicates it back through TIELT. The decision system continually adapts its behavior by evolving artificial neural network controllers to perform the task.

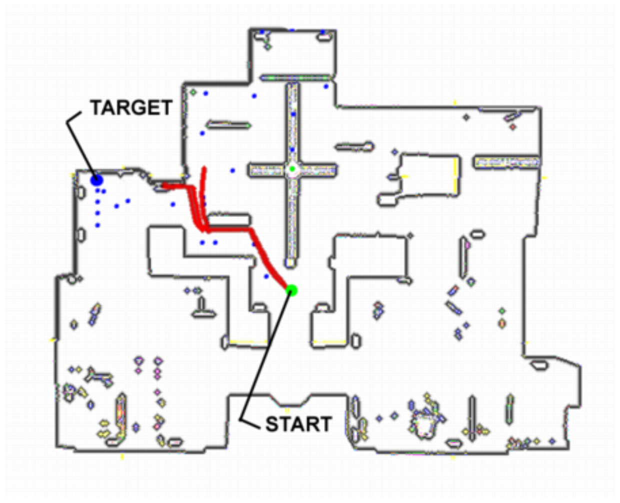
UT is a popular real-time first person shooter (FPS) computer game produced by Epic Games, Inc. in 1999 and winning a Game of the Year title for that year. This game was previously integrated with the TIELT system and other learning methods were tested on it (Molineaux 2004). In UT, a player navigates a three-dimensional space. The player can walk, run, jump, turn, shoot and interact with objects in the game such as armor, doors and health vials. The goal of one variant of the game (the tournament) is to be the first to destroy a certain number of opponents. Up to 16 players can play the game concurrently in a single level. A player can be controlled either by a human player connected over a network or an automated bot controlled by a built-in script.

The Gamebots API (Kaminka et al. 2002) modifies the original game to allow players to be controlled via sockets connected to other programs such as an adaptive decision system. The communication protocol consists of synchronous and asynchronous sensory messages sent from the server to all the clients and of commands sent from the clients back to the server. The server has all the information about player locations, interactions and status. The synchronous updates occur about every 100 milliseconds, and include updates to the player's view of the game state. Asynchronous events include collisions with objects, players and projectiles, results of queries by the player, and game over events.

The TIELT implementation of UT and NEAT consists of the following modules: **The Game Model** represents the knowledge that a player has about the game at any given time. It includes the locations and types of objects encountered in the game, the objects that are currently visible or reachable, the location and heading of players, health, armor and other player status indicators. Additionally, the game model holds an array of eight Boolean variables that correspond to whether the locations distributed at a fixed radius around the player's most recent position are reachable. This game model allows the information from synchronous and asynchronous updates to be combined into a single state that can then be used by the decision system to generate appropriate actions. Because TIELT scripting language did not implement the operations necessary to combine these values into useful sensory inputs, the final values were calculated in the NEAT decision system implementation. **The Game Interface Model** defines a subset of the GameBots protocol that is used to update the game model and trigger agent actions. **The Decision System Interface Model** uses Java Reflection to dynamically load and use libraries of Java classes. These classes implement the NEAT



(a) Agent sensors and actions



(b) Example of a path to target task

Figure 3: **Evolving Agents for Unreal Tournament.** (a) Each neural network has 11 sensory inputs: eight radial Boolean obstacle sensors and three forward-facing "radar" sensors, whose values are derived from the distance to navigation points visible in each of the three sectors shown on the figure. (b) Three traces of the best neural network navigating from the starting position to the ending position in an Unreal Tournament level. The network shown is the result of 33 generations of evolution and has the fitness score of 3518 out of 4000. Animations of learned behaviors can be seen at `http://nn.cs.utexas.edu/keyword?tieltne`.

learning system. **The Agent Description** is a script that sends sensor information from the Game Model to the Decision System and executes the resulting action on each synchronous update from the UT server. This process is performed many times to evaluate a single individual. **The Experimental Methodology** module allows the user to specify which Game Model, Decision System, and Agent Description are to be used in an experiment and how many repetitions of the experiment have to be run. With NEAT, a single TIELT experiment corresponds to the evaluation of a single individual's fitness, and must be repeated $e \times p$ times, where e is the number of epochs and p is the population size. The state of the NEAT algorithm is persisted in memory across TIELT experiments.

The output of the decision system is controlled by evolving neural networks. Each neural network in a population performs a number of decisions during a predefined lifetime of the individual in the UT game environment. The resulting behavior is analyzed to compute a fitness score (section 4.3). The UT world is then reset and a new network is used for decision making. The decision system keeps track of individuals, their fitnesses, evaluation times, populations and epochs. The resulting algorithm is thus simply a serial variant of the neuroevolution algorithm, evaluating each individual in turn.

Each neural network has 11 egocentric sensor inputs: eight boolean obstacle sensors $S_0..S_7$ with a small radius (in relative scale of the game they are roughly equivalent to stretching out the bot's arms in 8 directions) and three forward-directed 60-degree "radar" values $R_0..R_2$ (figure 3a). In addition to these 11 sensors, each network has a bias input (a constant 0.3 value found appropriate in previous NEAT experiments). Each radar value R_I is computed as $R_I = \sum_{x \in N_I} d(x)/C$, where N_I is the collection of navigation landmarks visible in sector I, $d(x)$ is the distance to each landmark, and C is a constant scaling factor. Thus, the R_I can be interpreted as the amount of free space in direction I, with higher activations corresponding to more visible navigation landmarks and to landmarks that are visible further away (figure 3a).

At each update, the decision system scales the single output of the network to a relative yaw angle $\Delta\alpha$

in the range of $[-\pi, \pi]$. TIELT then sends a command to the UT game server to move towards a point $(x + s\cos(\alpha + \Delta\alpha), y + s\sin(\alpha + \Delta\alpha), z)$ where s is the step size parameter.

4.3 Experiments with TIELT/NE

A number of validation experiments were conducted to verify that the learning system design is effective (for details on the setup, see Karpov et al. 2006). The task was to navigate through the environment to a static target within a given time (10 seconds; figure 3*b*). At the beginning of an evaluation, the bot is placed at the origin and performs actions for the duration of the evaluation. The minimal distance $d_{\min}$ to the target is measured over the synchronous updates received by the learning system. The fitness function is $C - d_{\min}$, which grows to a maximum value of C when the bot is able to reach the target (figure 3*b*).

Using the TIELT/NEAT system, controllers evolved reliably to solve this task. Starting from initially random behavior, the average record population fitness improved quickly from 3080 to 3350 in 20 generations, and to beyond 3500 in further evolution. This result means that the best agent was able to navigate reliably around obstacles to the the close neighborhood of the target.

4.4 Evaluation of Interface Systems

TIELT made the work of applying neuroevolution to UT simpler in several ways. Above all, the communication layer between UT and TIELT was already implemented and required only minor adjustments to integrate with the NEAT decision system. On the other hand, there are several ways in which TIELT could be modified or other similar integration platforms be built in the future to better support exploration-based learning.

First, the system should support multi-agent functionality explicitly. Currently TIELT does not provide a convenient way to parallelize evaluations of agents or to evolve multiple agents acting in the same environment. As a result, a large portion of the functionality that can be taken care of in the middleware framework had to be implemented in the decision system. This made the decision system specific to the UT application and reduced the advantages of the middleware.

Second, the system should supply the specific features needed in exploration-based learning. The modules of TIELT, while well-suited for rule-based learning, are not as useful with neuroevolution or online reinforcement learning. For NEAT as well as other exploration-based learning methods, the concepts of an evaluation episode, an individual agent, and a population are beneficial, and should be provided.

Third, the system should be implemented efficiently, minimizing unexpected delays. Because much of TIELT is implemented in Java, it runs slower than the game engine, and often incurs unpredictable garbage collection and indirection delays. These factors make the latencies of sensors and actions highly variable, which introduces new challenges into learning a task in a simulated real-time environment.

Fourth, the system should support scripted batch experiments. The interactive nature of TIELT makes it difficult to run long series of repeated experiments, especially when distributing the work to a cluster of machines (which TIELT already supports).

Fifth, learning agent benchmarking interfaces such as TIELT need to have their source open to the users. Doing so will greatly shorten the debugging cycle as well as allow researchers to have complete knowledge of the implementation of their experimental system.

The current results already show that a generic framework such as TIELT can be used to integrate and evaluate adaptive decision systems with rich computer game environments. With these extensions, it

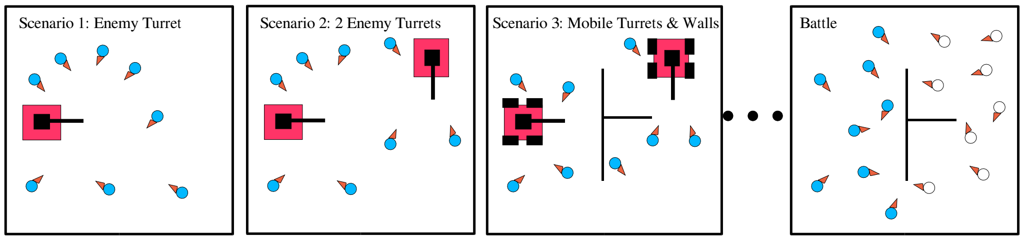

Scenario 1: Enemy Turret
Scenario 2: 2 Enemy Turrets
Scenario 3: Mobile Turrets & Walls
Battle

Figure 4: **A Sample Training Sequence in NERO.** The figure depicts a sequence of increasingly difficult training exercises in which the agents attempt to attack turrets without getting hit. In the first exercise there is only a single turret but more turrets are added by the player as the team improves. Eventually walls are added and the turrets are given wheels so they can move. Finally, after the team has mastered the hardest exercises, it is deployed in a real battle against another team. For animations of various training and battle scenarios, see `http://nerogame.org`.

may be possible to use sophisticated game playing domains in developing better exploration-based learning methods, as well as develop interesting mods for current and future games.

5 Building Machine Learning Games

With current NE techniques it is possible to take learning well beyond game mods, and develop entirely new game genres. One such genre is Machine Learning Games, where the player explicitly trains game agents to perform various tasks. The fun and the challenge of such games is to figure out how to take the agents through successive challenges so that in the end they perform well in the chosen tasks. Games such as Tamagotchi virtual pet and Black & White "god game" suggest that such interaction with artificial agents can make a viable and entertaining game. This section describes how advanced learning methods such as NE can be utilized to build complex machine learning games. As an example, the NERO game (Stanley et al. 2005a, 2006, 2005b) is built based on the rtNEAT method.

5.1 The NERO Game Design

The main idea of NERO is to put the player in the role of a trainer or a drill instructor who teaches a team of agents by designing a curriculum. The agents are simulated robots, and the goal is to train a team of these agents for military combat. The agents begin the game with no skills and only the ability to learn. In order to prepare for combat, the player must design a sequence of training exercises and goals. Ideally, the exercises are increasingly difficult so that the team can begin by learning basic skills and then gradually build on them (figure 4). When the player is satisfied that the team is well prepared, the team is deployed in a battle against another team trained by another player, allowing the player to see how his or her training strategy paid off. The challenge is to anticipate the kinds of skills that might be necessary for battle and build training exercises to hone those skills.

The player sets up training exercises by placing objects on the field and specifying goals through several sliders. The objects include static enemies, enemy turrets, rovers (i.e. turrets that move), flags, and walls. To the player, the sliders serve as an interface for describing ideal behavior. To rtNEAT, they represent coefficients for fitness components. For example, the sliders specify how much to reward or punish approaching enemies, hitting targets, getting hit, following friends, dispersing, etc. Each individual fitness component is normalized to a Z-score (i.e. the number of standard deviations from the mean) so that they are measured on the same scale. Fitness is computed as the sum of all these components multiplied by their slider levels,

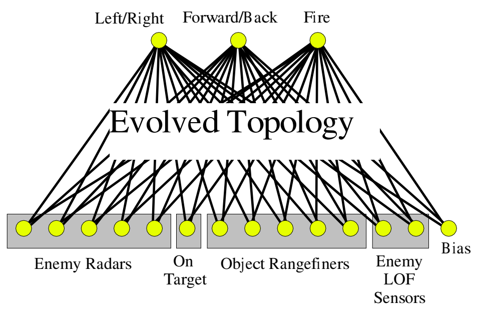


Figure 5: **NERO Input Sensors and Action Outputs.** Each NERO agent can see enemies, determine whether an enemy is currently in its line of fire, detect objects and walls, and see the direction the enemy is firing. Its outputs specify the direction of movement and whether or not to fire. This configuration has been used to evolve varied and complex behaviors; other variations work as well and the standard set of sensors can easily be changed in NERO.

which can be positive or negative. Thus, the player has a natural interface for setting up a training exercise and specifying desired behavior.

Agents have several types of sensors (figure 5). Although NERO programmers frequently experiment with new sensor configurations, the standard sensors include enemy radars, an "on target" sensor, object rangefinders, and line-of-fire sensors. To ensure consistent evaluations, the agents spawn from a designated area of the field called the factory. Each agent is allowed a limited time on the field during which its fitness is assessed. When their time on the field expires, agents are transported back to the factory, where they begin another evaluation.

Training begins by deploying 50 agents on the field. Each agent is controlled by a neural network with random connection weights and no hidden nodes, which is the usual starting configuration for NEAT. As the neural networks are replaced in real-time, behavior improves, and agents eventually learn to perform the task the player sets up. When the player decides that performance has reached a satisfactory level, he or she can save the team in a file. Saved teams can be reloaded for further training in different scenarios, or they can be loaded into battle mode.

In battle mode, the player discovers how well the training worked out. Each player assembles a battle team of 20 agents from as many different trained teams as desired, possibly combining teams with different skills. The battle begins with the two teams arrayed on opposite sides of the field. When one player presses a "go" button, the neural networks obtain control of their agents and perform according to their training. Unlike in training, where being shot does not lead to an agent body being damaged, the agents are actually destroyed after being shot several times (currently five) in battle. The battle ends when one team is completely eliminated. In some cases, the only surviving agents may insist on avoiding each other, in which case the winner is the team with the most agents left standing.

The game engine Torque, licensed from GarageGames (`http://www.garagegames.com/`), drives NERO's simulated physics and graphics. An important property of the Torque engine is that its physics is slightly nondeterministic, so that the same game is never played twice. In addition, Torque makes it possible for the player to take control of enemy robots using a joystick, an option that can be useful in training.

5.2 Playing NERO

Behavior can be evolved very quickly in NERO, fast enough so that the player can be watching and interacting with the system in real time. In this section, a number of basic and sophisticated behaviors are described, based on the experience of multiple players under a controlled tournament.

The most basic battle tactic is to aggressively seek the enemy and fire. To train for this tactic, a single static enemy was placed on the training field, and agents were rewarded for approaching the enemy. This training required agents to learn to run towards a target, which is difficult since agents start out in the factory facing in random directions. Starting from random neural networks, it takes on average 99.7 seconds for 90% of the agents on the field to learn to approach the enemy successfully (10 runs, $sd = 44.5$s).

Note that NERO differs from most applications of evolutionary algorithms in that the quality of evolution is judged from the player's perspective based on the performance of the *entire* population, instead of that of the population champion. However, even though the entire population must solve the task, it does not converge to the same solution. In seek training, some agents evolved a tendency to run slightly to the left of the target, while others run to the right. The population diverges because the 50 agents interact as they move simultaneously on the field at the same time. If all the agents chose exactly the same path, they would often crash into each other and slow each other down, so naturally some agents take slightly different paths to the goal. In other words, NERO is actually a massively parallel coevolving ecology in which the entire population is evaluated together.

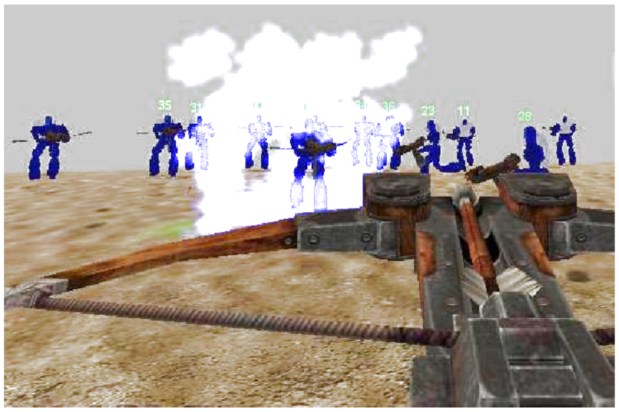
Agents were also trained to avoid the enemy. In fact, rtNEAT was flexible enough to *devolve* a population that had converged on seeking behavior into a completely opposite, avoidance, behavior. For avoidance training, players controlled an enemy robot with a joystick and ran it towards the agents on the field. The agents learned to back away in order to avoid being penalized for being too near the enemy. Interestingly, the agents preferred to run away from the enemy backwards, because that way they could still see and shoot at the enemy (figure $6a$).

By placing a turret on the field and asking agents to approach it without getting hit, agents were able to learn to avoid enemy fire. Agents evolved to run to the side that is opposite of the spray of bullets, and approach the turret from behind, a tactic that is promising for battle.

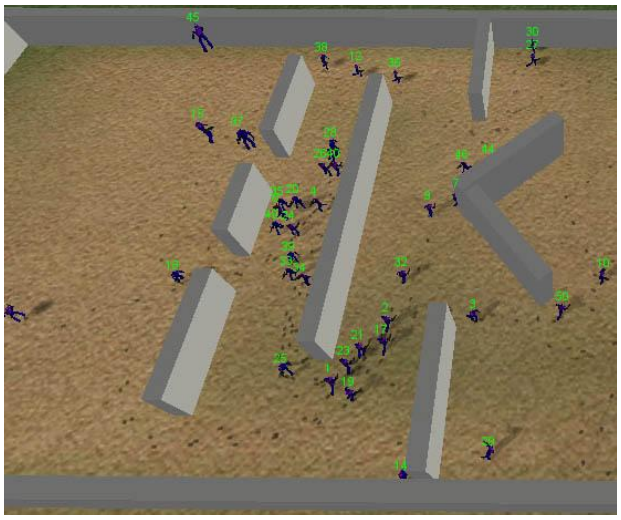
Other interesting behaviors were evolved to test the limits of rtNEAT, rather than specifically prepare the troops for battle. For example, agents were trained to run around walls in order to approach the enemy. As performance improved, players incrementally added more walls until the agents could navigate an entire maze (figure $6b$). This behavior is remarkable because it is successful without any path planning. The agents developed the general strategy of following any wall that stands between them and the enemy until they found an opening. Interestingly, different species evolved to take different paths through the maze, showing that topology and function are correlated in rtNEAT, and confirming the success of real-time speciation. The evolved strategies were also general enough to navigate significantly different mazes without further training.

In a powerful demonstration of real-time adaptation, agents that were trained to approach a designated location (marked by a flag) through a hallway were then attacked by an enemy controlled by the player. After two minutes, the agents learned to take an alternative path through an adjacent hallway in order to avoid the enemy's fire. While such training is used in NERO to prepare agents for battle, the same kind of adaptation could be used in any interactive game to make it more realistic and interesting.

In battle, some teams that were trained differently were nevertheless evenly matched, while some training types consistently prevailed against others. For example, an aggressive seeking team had only a slight advantage over an avoidant team, winning six out of ten battles. The avoidant team runs in a pack to a corner of the field's enclosing wall. Sometimes, if they make it to the corner and assemble fast enough, the aggressive team runs into an ambush and is obliterated. However, slightly more often the aggressive team gets a few shots in before the avoidant team can gather in the corner. In that case, the aggressive team traps the avoidant team with greater surviving numbers. The conclusion is that seeking and running away are



(a) Avoiding the enemy effectively



(b) Navigating a maze

Figure 6: **Behaviors Evolved in NERO.** (a) This training screenshot shows several agents running away backwards and shooting at the enemy, which is being controlled from a first-person perspective by a human trainer with a joystick. Agents discovered this behavior during avoidance training because it allows them to shoot as they flee. This result demonstrates how evolution can discover novel and effective behaviors in response to the tasks that the player sets up for them. (b) Incremental training on increasingly complex wall configurations produced agents that could navigate this complex maze to find the enemy. Remarkably, they had not seen this maze during training, suggesting that a general path-navigation ability was evolved. The agents spawn from the left side of the maze and proceed to an enemy at the right. Notice that some agents evolved to take the path through the top while others evolved to take the bottom path. This result suggests that protecting innovation in rtNEAT supports a range of diverse behaviors, each with its own network topology. Animations of these and other behaviors can be seen at `http://nerogame.org`.

fairly well-balanced tactics, neither providing a significant advantage over the other.

Strategies can be refined further by observing the behaviors in the battle, and setting up training exercises to improve them. For example, the aggressive team could eventually be made much more effective against the avoidant team by training them with a turret with its back against the wall. This team learned to hover near the turret and fire when it turned away, but back off quickly when it turned towards them. In this manner, rtNEAT can discover sophisticated tactics that dominate over simpler ones; the challenge for the player is to figure out how to set up the training curriculum so that they will emerge.

5.3 Evaluation of NERO

NERO was created over a period of about two years by a team of over 30 student volunteers (Gold 2005). It was first released in June of 2005 at `http://nerogame.org`, and has since then been downloaded over 60,000 times. It is under continuing development, currently focused on providing more interactive gameplay. In general, players agree that the game is engrossing and entertaining. Battles are exciting events, and players spent many hours honing behaviors and assembling teams with just the right combination of tactics. Remarkably, players who have little technical background often develop accurate intuitions about the underlying mechanics of machine learning. This experience is promising, suggesting that NERO and other machine learning games are viable as a genre.

NERO can also be used as a research platform for implementing novel machine learning techniques. An important issue for the future is how to assess results in a game in which behavior is largely subjective.

One possible approach is to train benchmark teams and measure the success of future training against those benchmarks. This idea and others will be employed as the project matures and standard strategies are identified.

Significant challenges for future research have also emerged from the NERO project. First, in order to make NE learning viable in commercial games, it is necessary to guide it with human knowledge. Instead of initializing the population with random behaviors, they should begin with a set of basic skills, and improve upon them; similarly, the human observer should be able to guide them towards desired behaviors during evolution. Second, even though the behaviors discovered through evolution may be effective, they do not always appear visibly intelligent. It should be possible to bias evolution towards behaviors that humans judge intelligent, e.g. those that are parsimonious and goal-directed. The next two sections discuss ways in which neuroevolution can be guided towards such behaviors.

6 Guiding NE with Human Knowledge

As was discussed in section 2, in most current video games, the non-player-characters (NPCs) are controlled with scripts, i.e. collections of preset behaviors for various situations foreseen by the programmer. They are often implemented as sets of rules or as finite-state machines (FSMs). Such behaviors are easy to program and they are reliable; on the other hand they tend to be repetitive and predictable. However, the video-game industry has learned to rely on them and has done quite well with this approach.

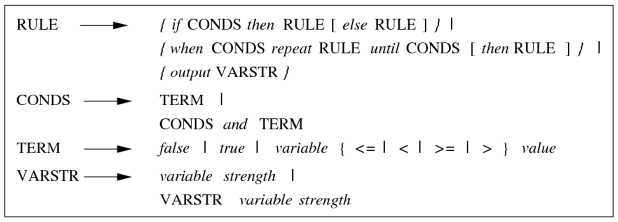
As described in the previous two sections, the technology now exists that allows adapting NPCs in real time, producing a variety of complex novel behaviors. However, there are times when evolution seems to take longer than necessary to discover a desired behavior, or when that behavior would be easier to specify verbally rather than through a training regimen. In such cases, it would be useful to be able to incorporate the desired behavior directly into the population initially or during evolution. Such human-injected knowledge is difficult to take into account in evolution in general, but the structure of the rtNEAT method makes it possible, as described in this section (for more details, see Cornelius et al. 2006; Yong et al. 2006).

6.1 Knowledge-based NEAT (KB-NEAT)

The knowledge-incorporation method called KB-NEAT consists of four components: First, the knowledge is specified according to a formal language. Second, it is converted into a neural-network structure. Third, it is added to the selected agents in the population by splicing the knowledge neural network onto the agent's neural network. Fourth, the knowledge is refined in further evolution in the domain.

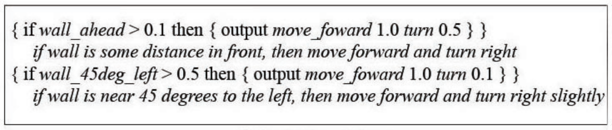
The knowledge language is specified as a grammar, given in figure 7a. There are three different types of rules: if-then rules, repeat rules, and output rules. The first two are self-explanatory; the third, output rule, states that an output variable should be either excited or inhibited. A wide range of knowledge can be expressed in this language, including state memories (using repeat loops) and nested statements (loops within loops).

For example, if the agents have been trained to go to a flag but have no experience with walls, they will run directly towards the flag even if there is a wall in the way. Through exploration, they will eventually learn to head away from the flag and go around the wall. Although such learning does not take very long in terms of game time—about one minute on average—it still seems like such a simple solution is not worth the time to be discovered automatically, since it is obvious to the human player what the agents should do. Such knowledge can be easily encoded as knowledge, as shown in figure 7b. It tells the agent how to go around the right side of a wall by turning right and moving along the wall. The next step is to convert it to



```
RULE    ——►  { if CONDS then RULE [ else RULE ] } |
             { when CONDS repeat RULE until CONDS [ then RULE ] } |
             { output VARSTR }
CONDS   ——►  TERM |
             CONDS and TERM
TERM    ——►  false | true | variable { <= | < | >= | > } value
VARSTR  ——►  variable strength |
             VARSTR variable strength
```

(a) Knowledge grammar



```
{ if wall_ahead > 0.1 then { output move_foward 1.0 turn 0.5 } }
   if wall is some distance in front, then move forward and turn right
{ if wall_45deg_left > 0.5 then { output move_foward 1.0 turn 0.1 } }
   if wall is near 45 degrees to the left, then move forward and turn right slightly
```

(b) Knowledge on how to go around a wall

Figure 7: **Expressing Knowledge in Rules.** (a) The grammar allows specifying if-then rules and nested loops, i.e. finite-state machines similar to those used to control non-player characters in many current video games. (b) This sample knowledge, expressed in the language specified in (a) tells the agent to move forward and to the right, going around the right side of the wall.

the neural network structure.

The knowledge is converted into an equivalent neural-network structure via a recursive algorithm similar to RATLE (Maclin and Shavlik 1996), which itself is based on KBANN (Towell and Shavlik 1994). The main difference is that learnable recurrent connections are used to maintain the state of the loop in repeat rules, instead of copies of previous activations. This algorithm builds one of three structures depending on whether the rule is an if-then, repeat, or output rule (figure 8):

For an if-then rule:

1. Create the CONDS structure.
2. Build the rule in the "then" action by calling the conversion algorithm recursively.
3. If there is an "else" clause, create the negation node, and build the rule in the "else" action by calling the conversion algorithm recursively.
4. Connect the nodes with the weight values given in the rule.

For a repeat rule:

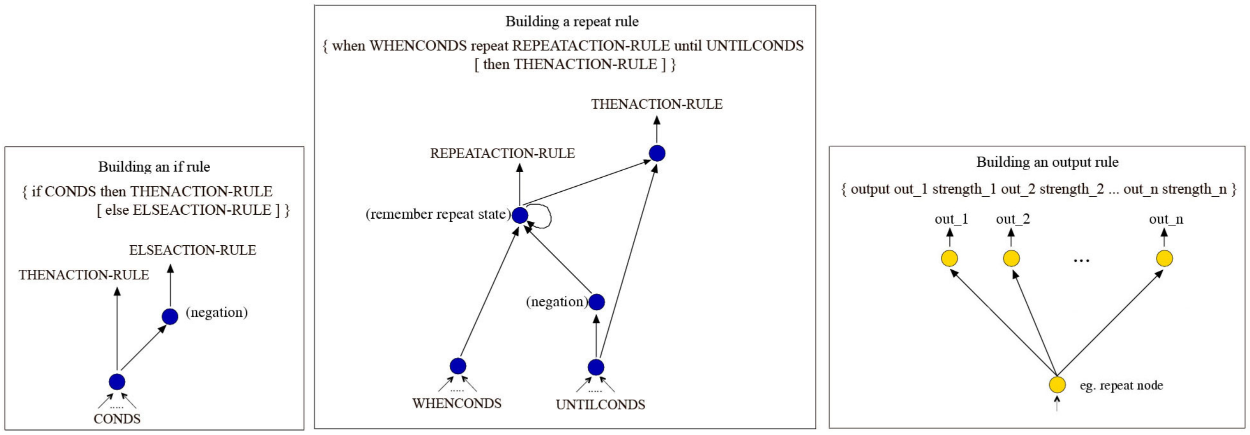
1. Create the WHEN-CONDS and UNTIL-CONDS structures.
2. Create the remember-repeat-state and negation nodes.
3. Build the rule in the "repeat" action by calling the conversion algorithm recursively.
4. If there is a "then" clause, create the node for it, and build the rule in the "then" action by calling the algorithm recursively.
5. Connect the nodes with the weight values given in the rule.

For an output rule:

1. Connect the current node to the output units with weight values specified in the rule.

Knowledge is incorporated into an agent by splicing the knowledge network structure onto the agent's neural network. This step is similar to RATLE, except connections with low weights are not included to make the resulting network fully connected; instead, the rtNEAT algorithm adds connections later as needed. Figure 9 shows the resulting network: The original network consists of light nodes and the added network structure implementing the knowledge consists of dark nodes.

The inserted knowledge is refined during further evolution by allowing the knowledge portion of the network to evolve along with the original network. Its weights can be mutated, new links can be added



(a) If-then rule (b) Repeat rule (c) Output rule

Figure 8: **Converting Knowledge into Neural-network Structures.** (a) An if-then rule is constructed by linking the conditions to the "then" action, and their negation to the "else" action. (b) A repeat rule is constructed by creating a neuron that remembers the repeat state through a recurrent connection, and linking the "when" conditions to activate this node and the "until" conditions to deactivate it. (c) An output rule is constructed by linking the rule node to each of the output nodes, weighted as specified in the rule. A dark node indicates a node to be added to the network, a light node indicates an existing node in the network.

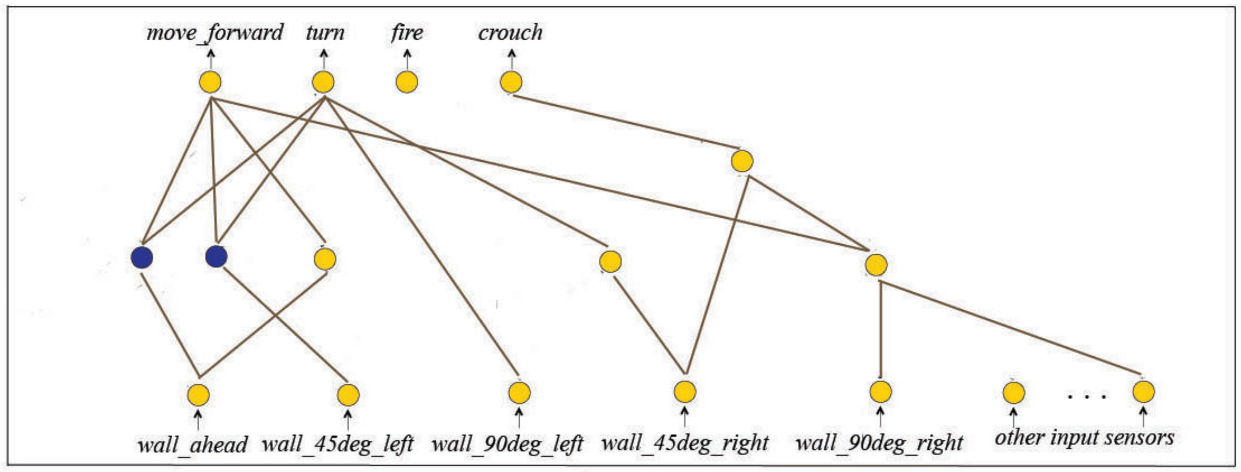
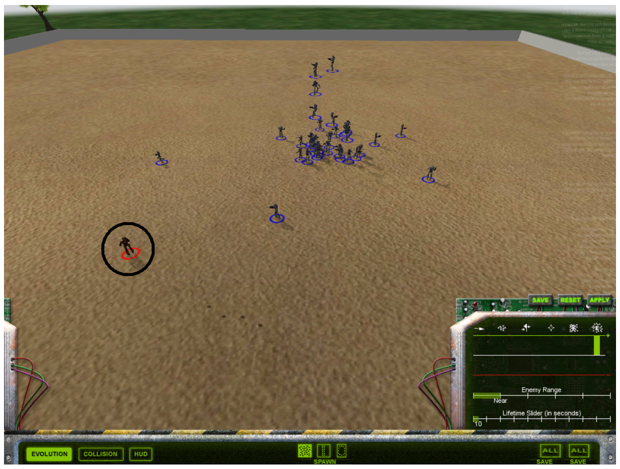


Figure 9: **Incorporating Knowledge into Neural Networks in NERO.** The network structure that implements the knowledge in figure 7b is shown as dark nodes. The thresholds and weights of each hidden unit is set so that they implement the two rules. This network structure is spliced into the existing network (shown with light nodes) by connecting the hidden nodes of the knowledge structure to the appropriate inputs and outputs.

between nodes, and new nodes can be added. Such knowledge can be introduced in the initial population, or given during evolution as advice, as will be described in the next two subsections.

6.2 Seeding NEAT with Initial Knowledge

The first experiment tested the ability of a team of 40 agents to approach a roving enemy in NERO (figure 10). This enemy was placed 50 distance units away from the middle of the factory at all times, and was made to run around it at a constant speed. As each agent spawned from the factory it was given ten seconds to get within five distance units of the enemy. At the end of an agent's evaluation period, its distance from


(a) Random initialization

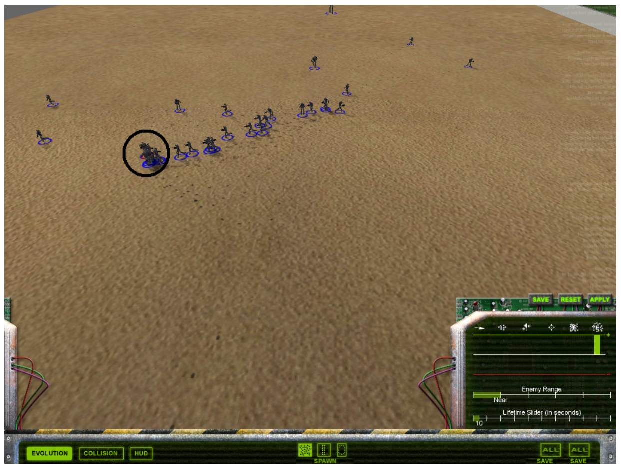

(b) Knowledge initialization

Figure 10: **Testing Initial Behaviors to Approach a Single Roving Enemy.** The environment contains a single roving enemy robot (circled). The rover circles the factory (cluster), from which the agents spawn. (a) With random neural network controllers, the agents run around in random fashion. (b) With controllers initialized with an FSM, they know how to approach the enemy right away. Alternatively, similar behavior can be discovered from the random behavior in (a) in about three minutes of rtNEAT evolution. Animations of these behaviors can be seen at `http://nn.cs.utexas.edu/keyword?kbneat`.

the enemy was used as its fitness, and the neural network controller was reused in another agent.

An FSM was designed to control the agents so that they would approach an enemy. It consists of 14 state transitions, described by the following rules:

1: EnemyRadarSensor_RightCenter > 0 and Forward' > 0.6 then output Forward 1.0
2: EnemyRadarSensor_LeftCenter > 0 and Forward' > 0.6 then output Forward 1.0
3: EnemyRadarSensor_RightFront > 0 and Forward' > 0.6 then output Right 1.0
4: EnemyRadarSensor_LeftFront > 0 and Forward' > 0.6 then output Left 1.0
5: EnemyRadarSensor_LeftFront > 0 and Left' > 0.6 then output Left 1.0
6: EnemyRadarSensor_RightCenter > 0 and Left' > 0.6 then output Forward 1.0
7: EnemyRadarSensor_LeftCenter > 0 and Right' > 0.6 then output Forward 1.0
8: EnemyRadarSensor_RightFront > 0 and Right' > 0.6 then output Right 1.0
9: EnemyRadarSensor_Backwards > 0 and Right' > 0.6 then output Right 1.0
10: EnemyRadarSensor_Backwards > 0 and Idle' > 0.6 then output Right 1.0
11: EnemyRadarSensor_LeftFront > 0 and Idle' > 0.6 then output Left 1.0
12: EnemyRadarSensor_RightFront > 0 and Idle' > 0.6 then output Right 1.0
13: EnemyRadarSensor_LeftCenter > 0 and Idle' > 0.6 then output Forward 1.0
14: EnemyRadarSensor_RightCenter > 0 and Idle' > 0.6 then output Forward 1.0

where apostrophy indicates the previous value of an output variable. Rules 1–2 state that if the agent is currently moving forward and the enemy is ahead of it, it should continue moving forward. Rules 6–7 and 13–14 state that if the agent has turned or is idle, and the enemy appears to be straight ahead, it should move forward. Rule 3–5, 8, and 10–12 tell the agent to turn left if there is an enemy to the left, or right if there is an enemy to the right. Rule 9 tells the agent to turn right if there is an enemy behind it. Together, these rules express the behavior of approaching an enemy in any situation.

A population was created by converting this FSM into a neural network and making 100 copies of it. Another population was created by initializing the weights of 100 networks with values chosen randomly from a uniform distribution within [0,1]. Each population was evolved until 3500 networks had been eval-

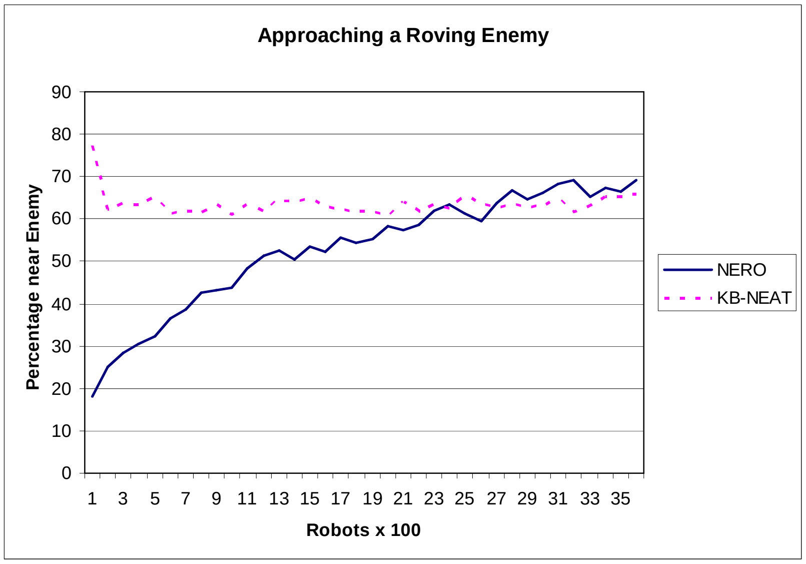


Figure 11: **The Percentage of Successful Agents with Random and Knowledge Initialization.** The agents initialized with the knowledge (KB-NEAT) understood how to approach the roving enemy from the beginning; it took nearly 1100 evaluations for evolution to discover this behavior from scratch (NERO). The graph is an average of ten runs and the differences are statistically significant up to 1100 evaluations.

uated. This experiment was run ten times for each population and the average success rate taken as the performance of the approach.

The population initialized in this manner was able to complete the task immediately (figure 10b). The overall success rate drops slightly at the beginning of evolution because it is always exploring new solutions, but the population as a whole continues to do well (figure 11, plot labeled KB-NEAT). In contrast, the randomly initialized population starts out by exploring random behaviors (figure 10a). After evaluating about 1500 individuals (which took about three minutes) they learn to approach the enemy and perform as well as the initialized population (figure 11, NERO).

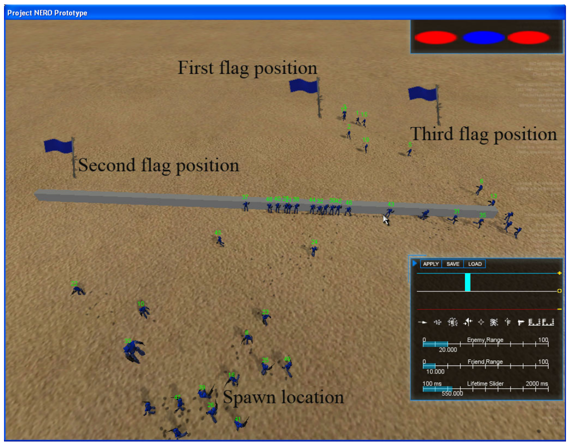
The conclusion from this experiment is that human knowledge can be used to create agents that exhibit basic skills even before any evolution has taken place, and that these skills persist as evolution continues. The next question is whether such knowledge can also be injected during evolution.

6.3 Advising NEAT During Evolution

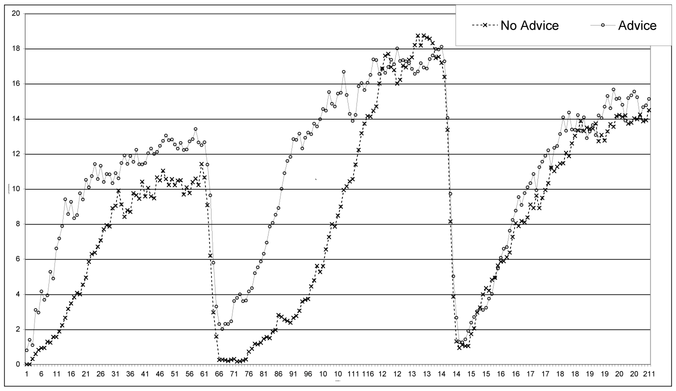
Experiments were conducted to test three hypotheses: (1) Giving advice helps the agents learn the task; (2) even if the task changes into one that conflicts with the advice, agents can modify the advice to solve the new task; and (3) the advice eventually becomes incorporated into the network the same way as behaviors discovered through exploration.

These hypotheses were tested in learning to move around a wall to get to a flag that is placed in various locations behind it. The advice used in all these experiments was the sample depicted in figure 7b. In NERO, the advice is added to the best individual in the top k species of the population. Each of these individuals is copied n times, slightly mutating their weights, replacing the worst kn agents in the population. In the current experiments, $k = 5$ and $n = 4$ for a total of 20 agents with the knowledge.

The performance of each approach was measured as the number of agents out of the population of 50



(a) The three phases of the experiment



(b) Performance over generations

Figure 12: **Experiment on Advising NEAT.** The agents in the NERO game were first (during generations 0 to 600) trained to go to the flag by going around the right side of the wall. The advised agents learned significantly faster and reached significantly higher performance. The task then changed (during generations 601 to 1400), requiring them to go to the left. Surprisingly, the difference is even greater: It was easier for evolution to discover mutations that changed the bad advice into behaviors that worked in the new situation, than change the behaviors it had discovered without advice. In the third phase (generations 1401 to 2100), the right-path behavior was required again. Both systems learned equally: The advice had become encoded the same way as the behaviors discovered without advice. The control window on the bottom right in (a) indicates that the agents are trained only on one objective: Going to the flag (indicated by the single light slider). The ovals on top right of (a) indicate that the snapshot is taken during the second (middle) task. The generations in (b) are listed as multiples of ten. An animation of this process can be seen at `http://nn.cs.utexas.edu/keyword?kbneat`.

that reached the flag at each timestep. The agents' lifetime was 5.5 seconds; depending on the position of the flag, the shortest path to it takes 3 to 5 seconds. Thus each agent had to behave nearly optimally to be successful. To smooth out the performance values, at each time step the windowed average of the performance of the past 30 time steps was taken, and the results were averaged over 3 runs.

Before the experiment, the agents were pretrained to go to the flag, but they did not have any experience with walls. In each run, they start in front of a wall with the flag placed behind it (figure 12a), and have to learn to circumvent the wall around the right side to reach the flag before their lifetime expires. The evolution continues until a stable, successful behavior has been discovered; this happened in 600 generations in all runs.

Figure 12b plots the average performance of each version over time. The left third of the graph (generations 0 to 600) corresponds to the first phase of the experiment. Without advice, the agents reached a performance level of 11 at around generation 450. With advice, the performance improved significantly: The agents learned faster and reached a higher level of performance (13 at generation 450). These results support the first hypothesis that giving pertinent advice makes learning the task easier.

After the agents had learned to go to the flag, it was moved close behind the wall on the left side (figure 12a). To reach the flag in their lifetime, agents now had to circumvent the wall on the left side; there is not enough time if they follow the advice and go to the right. Evolution with and without the advice was compared to determine whether evolution can recover from such bad advice and learn the new task. Evolution was run for 800 further generations to achieve the new task.

The middle third of the graph (generations 601 to 1400) corresponds to the second phase. Initially, the agents either tried to make a long detour around the right side of the wall, or go straight towards the flag and get stuck behind the wall, resulting in a drastic drop in performance. Gradually, they began to learn to circumvent the wall around its left side. Surprisingly, the agents without the advice were the slowest to learn this behavior, taking over 200 generations longer than the agents with advice. This result was unexpected because the advice is in direct conflict with the necessary behavior. Analysis of successful networks showed that the same advice portion of the network was still used, but evolution had discovered mutations that turned a right circumvention into a left one. In other words, the original advice was still useful, but the network had discovered a refinement that applied to the changed task. Hypothesis 2 was therefore confirmed: Evolution can modify advice to fit the task.

In the third phase, the flag was moved further back behind the wall to near where it originally was, but slightly to its right (figure 12*b*). The agents now had to circumvent the wall on its right side again. Evolution with and without the advice was compared to see if whether any differences remained, or whether the advice had been incorporated into the network like the behaviors learned by exploration.

The right third of the graph (generations 1401 to 2100) corresponds to the third phase. Since learning to go around a wall is not that difficult a task, it can be solved by exploration without the advice if given enough time. In this case, such learning had already taken place during the previous two phases, and the advised and non-advised agents learned equally well. Hypothesis 3 was therefore confirmed as well: Advice had become fully integrated into the network.

6.4 Evaluation of Knowledge-based Neuroevolution

The experiments presented in this section demonstrate how human-generated knowledge can be incorporated into neuroevolution. Such knowledge allows bypassing the initial phase where agents behave randomly. It can also be given dynamically during evolution, and it makes learning faster and easier. Even when such knowledge conflicts with the task, evolution may discover ways to utilize it to its advantage. The structures generated from the knowledge gradually become incorporated into the network like structures discovered by evolution.

The knowledge-based neuroevolution technique makes it possible to control the kinds of behaviors that evolution will discover. The initial NPC behaviors can be expressed as is currently done in most video games, i.e. through FSMs; Neuroevolution can then be invoked to develop these behaviors further. If the evolved behaviors turn out different from those envisioned, real-time advice can be given to guide evolution towards desired behaviors. Such techniques should make it easier and safer for the game designers in industry to adopt novel adaptation methods like neuroevolution. The software for KB-NEAT is available at `http://nn.cs.utexas.edu`.

7 Evolving Visibly Intelligent Behavior Through Examples

A second major challenge for neuroevolution in game playing is to develop agents that appear visibly intelligent. Evolution is very good at discovering effective behaviors that solve the task, as defined in the fitness function, even though such behaviors may not always make sense to a human observer. For example, the agents may learn to utilize loopholes in the game design, or simply move back and forth for no reason. Such solutions may be fine in optimization applications, but they are a problem in game playing. Part of the goal is to develop agents that make the game fun, and appearing intelligent is an important goal in itself. In this section, a new technique for encouraging visibly intelligent behavior will be reviewed: utilizing human examples. This technique is illustrated with examples in the Legion II game, developed as a research platform for visible intelligence in strategy games.

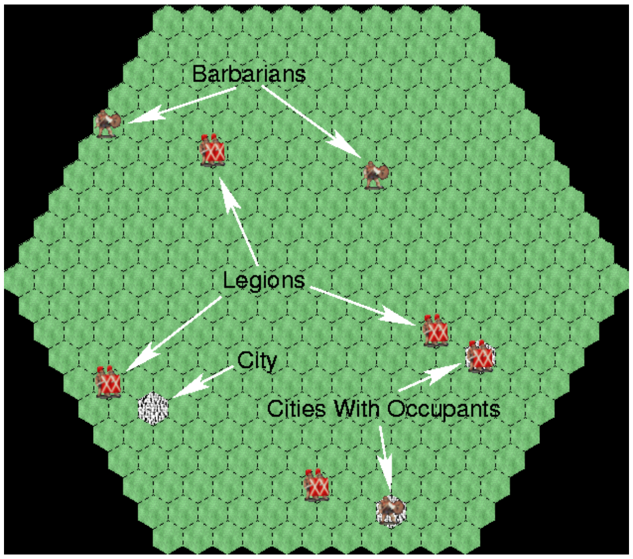


Figure 13: **The Legion II game..** A large hexagonal playing area is tiled with smaller hexagons in order to quantize the positions of the game objects. Legions are shown iconically as close pairs of men ranked behind large rectangular shields, and barbarians as individuals bearing an axe and a smaller 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 hexes are farmland, shown with a mottled pattern. The game is a test bed for multi-agent learning methods, wherein the legions must learn to contest possession of the playing area with the barbarians. For various animations of the Legion II game, see `http://nn.cs.utexas.edu/keyword?ata`.

7.1 The Legion II Game

Legion II is a discrete-state strategy game designed as a test bed for multi-agent learning problems, with legions controlled by artificial neural networks acting as the intelligent agents in the game (Bryant 2006; Bryant and Miikkulainen 2006b). The game is played on a map that represents a province of the Roman empire, complete with several cities and a handful of legions for its garrison (figure 13). Gameplay requires the legions to minimize the pillage inflicted on the province by a steady stream of randomly appearing barbarian warbands. The barbarians collect a small amount of pillage each turn they spend in the open countryside, but a great deal each turn they spend in one of the cities.

The game is parameterized to provide enough legions to garrison all the cities and have a few left over, which can be used to disperse any warbands they find prowling the countryside. The Legion II map is in the shape of a large hexagon, divided into small hexagonal cells to discretize the placement of game objects such as legions and cities (figure 13). Moves are taken in sequential turns. During a turn each legion makes a move, and then each barbarian makes a move. All moves are atomic, i.e. during a game agent's move it can either elect to remain stationary for that turn or else move into one of the six hexagons of the map tiling adjacent to its current position.

Only one agent, whether legion or barbarian, can occupy any map cell at a time. A legion can bump off a barbarian by moving into its cell as if it were a chess piece; the barbarian is then removed from play. Barbarians cannot bump off legions: They can only hurt the legions by running up the pillage score. Neither legions nor barbarians can move into a cell occupied by one of their own kind, nor can they move off the edge of the map.

A game is started with the legions and cities placed at random positions on the map; the combinatorics allow a vast number of distinct game setups. The barbarians enter play at random unoccupied locations, one per turn. If the roving legions do not eliminate them they will accumulate over time until the map is almost entirely filled with barbarians, costing the province a fortune in goods lost to pillage.

Play continues for 200 turns, with the losses to pillage accumulated from turn to turn. At the end of the game the legions' score is the amount of pillage lost to the barbarians, rescaled to the range $[0, 100]$ so that the worst possible score is 100. Lower scores are better for the legions, because they represent less pillage. Learning allows the legions to reduce the score to around 4 when tested on a random game setup never seen during training (i.e. to reduce pillage to about 4% of what the province would suffer if they had sat idle for the entire game).

The barbarians are programmed to follow a simple strategy of approaching cities and fleeing legions, with a slight preference for the approaching. The are not very bright, which suits the needs of the game and perhaps approximates the behavior of barbarians keen on pillage.

The legions must be trained to acquire appropriate behaviors. They are provided with sensors that divide the map up into six pie slices centered on their own location. All the relevant objects i in a pie slice are sensed as a single scalar value, calculated as $\sum_i 1/d_i$. This design provides only a fuzzy, alias-prone sense of what is in each sector of the legion's field of view, but it works well as a threat/opportunity indicator: A few barbarians nearby will be seen as a sensory signal similar to what would be seen of a large group of barbarians further away. There is a separate sensor array for each type of object in play: cities, barbarians, and other legions. There are sensors within each array to provide more detail about what is in the map cells adjacent to the sensing legion, or colocated in the legion's own cell.

The scalar sensor values, 39 in all, are fed into a feed-forward neural network with a single hidden layer of ten neurons and an output layer of seven neurons. A fixed-value bias unit is also fed into each of the neurons in the network. The size of the hidden layer was chosen by experimentation. The output neurons are associated with the seven possible actions a legion can take in its turn: remain stationary, or move into one of the six adjacent map cells. This localist action unit coding is decoded by selecting the action associated with the output neuron that has the highest activation level after the sensor signals have been propagated through the network.

7.2 Lamarckian Evolution with Examples

When a team of legions is trained in the game, they learn the necessary devision of labor and become quite effective in the game. However, their behaviors also illustrate the arbitrary nature of the solutions evolution can produce: 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. However, 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 challenge is: Is it possible to inform the evolutionary process with human notions of what is appropriate behavior?

In principle, better behavior could be obtained by modifying the evolutionary fitness function, e.g. by charging a fitness cost for each move a legion makes during the course of a game. Another alternative is to guide evolution away from such behavior by coding knowledge in rules and seeding or advising evolution with them, as described in the previous section. However, in many cases such knowledge is difficult to specify and formalize, especially when the behavior itself is not harmful for performance per se, only not visibly intelligent. In such cases, a more natural way is to demonstrate what the desired behavior would look like. The goal, therefore, is to utilize the human examples to guide evolution.

When the controller is an artificial neural network that maps sensory inputs onto behavioral outputs, human examples can be used to train networks through a supervised learning method such as backpropagation

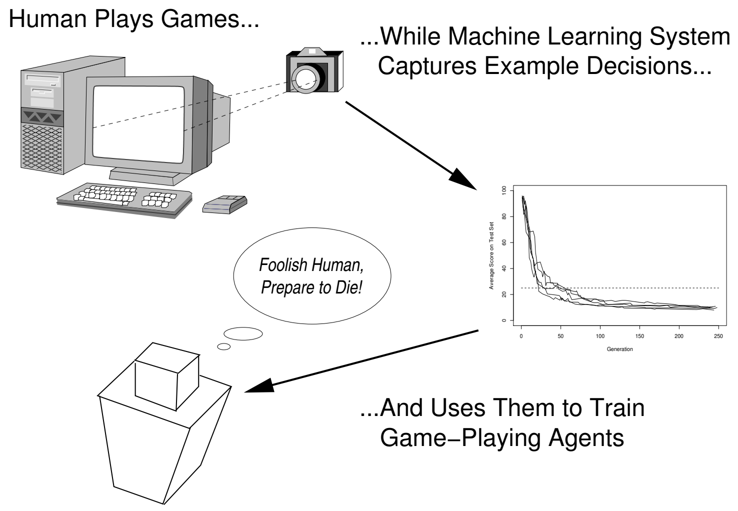

Human Plays Games...
...While Machine Learning System Captures Example Decisions...
Foolish Human, Prepare to Die!
...And Uses Them to Train Game−Playing Agents
Average Score on Test Set
Generation

Figure 14: **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.

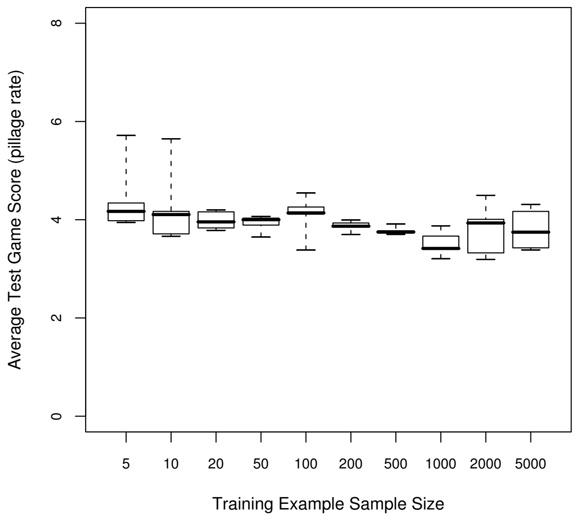
(Rumelhart et al. 1986). The changes can be reverse-engineered back to the chromosome, replacing the original chromosome in the gene pool. In this manner, learned behaviors become part of the genetic encoding, i.e. knowledge in the human examples is incorporated into evolution. This process is a form of Lamarckian evolution: Although such a process does not take place in biology, it is often useful in combining lifetime learning with evolution in artificial systems (D. Whitley 1994; Ku et al. 2000; Lamarck 1984).

The necessary 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. 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 as 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 14).

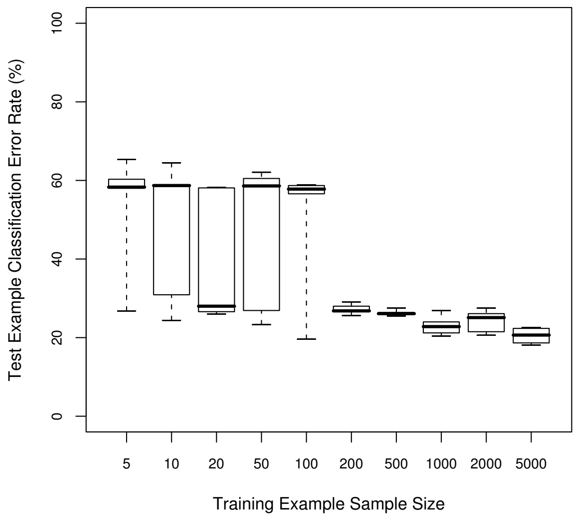
Examples of human play for the Legion II game were collected in this manner. The $<$*state,action*$>$ tuples consisted of a 39-element floating point vector of egocentric sensory activations for the state, and a symbol for one of the legion's seven discrete choices of action. The data was limited to situations where the legion needs to remain within one cell from the city it is guarding: in other words, if there are no barbarians nearby, it should stay in the city, if a single barbarian appears in a nearby cell, the legion should bump it off, and if more barbarians appear, the legion should again remain in the city (otherwise the extra barbarians can enter the city and pillage). Twelve games were played using this strategy, each generating 1000 examples. One of these games was reserved for testing, the others were available for training through random sampling.

The Lamarckian training had two main effects: First, the overall performance (i.e. the game score) improved slightly, as can be seen in figure 15a. Since the oscillations mostly happened when there were no barbarians around, they mostly did not affect the score. However, in a few cases, especially with more than one barbarian around, the legion would leave the city, allowing pillage to happen. Such events were rare after Lamarckian training, leading to the slightly improved score.

Second, the observed behavior was much more reasonable: Lamarckian training got rid of almost all of the oscillations. The only time they leave the cities is to bump off a lone nearby barbarian. This result can be seen qualitatively in figure 15b, where the behavior of the system is compared to human examples in the test set. Provided a sufficient number of training examples were used (200 or more), the behavior is



(a) Performance



(b) Behavior

Figure 15: **Effect of Human Examples on Game Score and Behavior.** (a) Game performance vs. number of training examples used per generation. Performance is measured as pillage rate, so lower is better. As more samples are used, the performance improves slightly: The legions no longer leave the city when there are more than one barbarian around. (b) Behavioral similarity vs. number of training examples used per generation. If enough examples are used (e.g. 200 or more), the difference between the agents' behavior and human examples is small. This result suggests that the agents have learned the behavioral "doctrine" demonstrated by the human through the examples.

close to the human examples. The remaining differences are mostly due to the fact that the human player uses extra information not as easily available to the agents, such as knowledge of barbarians outside the 1-cell radius. Overall, the conclusion is that the behavior is visibly intelligent in the same way as the human demonstration.

7.3 Evaluation of Visible Intelligence Through Examples

The experiments in this section demonstrate that human knowledge can be used also when it is difficult to formalize in rules. It is sufficient for the human to demonstrate the desired behavior: Through supervised learning and Lamarckian evolution, the "doctrine" expressed in the examples is adopted by evolution. Such knowledge is particularly useful in driving evolution towards visibly intelligent behavior.

There are several other, more automatic ways in which the behavior can be made more believable. First, agents may develop behaviors that are not symmetric, e.g. they may turn left by turning 270 degrees to the right. Such problems can be avoided by taking into account symmetries in the sensors, and explicitly generating training situations that include all symmetrically equivalent cases (Bryant and Miikkulainen 2006b). Also, as long as the networks are deterministic, their behavior may appear to be inflexible and predictable. As long as actions are selected by finding the most highly active unit in the evolved network's output, there is only one unit whose activation makes sense: There is no incentive for evolution to indicate any ranking of alternatives. Using a technique called stochastic sharpening (Bryant and Miikkulainen 2006a), it is possible to evolve output activations that represent confidence values for the different actions. Actions can then be chosen stochastically, without a significant loss in performance, making the agent more interesting in the game play.

8 Conclusion

Neuroevolution is a promising new technology that is particularly well suited for video game applications. Although NE methods are still being developed, the technology can already be used to make current games more challenging and interesting, or to implement entirely new game genres. The designer can control the kinds of solutions that emerge, giving evolution a head start, or making the behaviors more visibly intelligent. Such games, with adapting intelligent agents, are likely to be in high demand in the future. In addition, they may finally make it possible to build effective training games, i.e. those that adapt as the trainee gets better.

At the same time, video games provide interesting, concrete challenges for CI. For example, methods for control, coordination, decision making, and optimization, with severe uncertainty, material, and time constraints, can be studied systematically in such games. The techniques should be widely applicable in other fields, such as robotics, resource optimization, and intelligent assistants. Like traditional symbolic games for GOFAI, video gaming may thus serve as a catalyst for research in computational intelligence for decades to come.

Acknowledgments

Thanks to David Aha and Matthew Molineaux for providing the TIELT platform, for Thomas D'Silva and Craig Varrichio for help with the TIELT experiments, and for Aliza Gold and the entire NERO development team for their work on NERO. This work was supported in part by DARPA through an NRL grant #N00173041G025, by the Digital Media Collaboratory of the University of Texas at Austin, by the Texas Higher Education Coordinating Board through grant ARP-003658-476-2001, and by the National Science Foundation through grants EIA-0303609 and IIS-0083776.

The NERO game and demos can be downloaded at `http://nerogame.org` and the ESP and NEAT software and demos at `http://nn.cs.utexas.edu`.

References

Agogino, A., Tumer, K., and Miikkulainen, R. (2005). Efficient credit assignment through evaluation function decomposition. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings AAAI-87 Sixth National Conference on Artificial Intelligence*, volume 1, pages 268–272, Los Altos, CA. Morgan Kaufmann.

Aha, D. W. and Molineaux, M. (2004). Integrating learning in interactive gaming simulators. In *Challenges of Game AI: Proceedings of the AAAI'04 Workshop*, Menlo Park, CA. AAAI Press.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1993). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65.

Baer, R. H. (2005). *Videogames: In the Beginning*. Rolenta Press, Springfield, NJ.

Bongard, J. C. and Pfeifer, R. (2001). Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836. San Francisco: Kaufmann.

Braun, H. and Weisbrod, J. (1993). Evolving feedforward neural networks. In *Proceedings of ANNGA93, International Conference on Artificial Neural Networks and Genetic Algorithms*, Berlin. Springer.

Bryant, B. D. (2006). *Evolving Visibly Intelligent Behavior for Embedded Game Agents*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin.

Bryant, B. D. and Miikkulainen, R. (2003). Neuroevolution for adaptive teams. In *Proceedings of the 2003 Congress on Evolutionary Computation*.

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

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

Campbell, M., Hoane Jr., A. J., and hsiung Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, 134:57–83.

Cole, N., Louis, S., and Miles, C. (2004). Using a genetic algorithm to tune first-person shooter bots. In *Evolutionary Computation, 2004. CEC2004. Congress on Evolutionary Computation*, volume 1, pages 139–145, Piscataway, NJ. IEEE.

Cornelius, R., Stanley, K. O., and Miikkulainen, R. (2006). Constructing adaptive AI using knowledge-based neuroevolution. In Rabin, S., editor, *AI Game Programming Wisdom 3*. Charles River Media.

Crandall, R. W. and Sidak, J. G. (2006). Video games: Serious business for america's economy. Report, Entertainment Software Association.

D. Whitley, S. Gordon, K. M. (1994). Lamarckian evolution, the Baldwin effect and function optimization. In *Parallel Problem Solving from Nature - PPSN III*, pages 6–15.

Floreano, D. and Urzelai, J. (2000). Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, 13:431–4434.

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

Fogel, D. B., Blair, A., and Miikkulainen, R., editors (2005). Special issue on evolutionary computation and games. *IEEE Transactions on Evolutionary Computation*, 9(6).

Fogel, D. B., Hays, T. J., Hahn, S. L., and Quon, J. (2004a). A self-learning evolutionary chess program. *Proceeedings of the IEEE*, 92(12):1947–1954.

Fogel, D. B., Hays, T. J., and Johnson, D. R. (2004b). A platform for evolving characters in competitive games. In *Proceedings of 2004 Congress on Evolutionary Computation*, pages 1420–1426, Piscataway, NJ. IEEE Press.

Fullmer, B. and Miikkulainen, R. (1992). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In Varela, F. J. and Bourgine, P., editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 255–262. MIT Press, Cambridge, MA.

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

Gallagher, M. and Ryan, A. (2003). Learning to play Pac-Man: An evolutionary, rule-based approach. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on Evolutionary Computation*, volume 4, pages 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.

Geisler, B. (2002). An empirical study of machine learning algorithms applied to modeling player behavior in a 'first person shooter' video game. Master's thesis, Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI.

Gold, A. (2005). Academic AI and video games: A case study of incorporating innovative academic research into a video game prototype. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*. IEEE.

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. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1356–1361, San Francisco. Kaufmann.

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

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2006). Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning*.

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

Haugeland, J. (1985). *Artificial Intelligence: The Very Idea*. MIT Press, Cambridge, MA.

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

Hong, J.-H. and Cho, S.-B. (2004). Evolution of emergent behaviors for shooting game characters in robocode. In *Evolutionary Computation, 2004. CEC2004. Congress on Evolutionary Computation*, volume 1, pages 634–638, Piscataway, NJ. IEEE.

Hornby, G. S. and Pollack, J. B. (2002). Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3).

Igel, C. (2003). Neuroevolution for reinforcement learning using evolution strategies. In *Proceedings of the 2003 Congress on Evolutionary Computation*, pages 2588–2595.

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

Kaminka, G. A., Veloso, M. M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A. N., Scholer, A., and Tejada, S. (2002). Gamebots: A flexible test bed for multiagent team research. *Communications of the ACM*, 45:43–45.

Karpov, I., D'Silva, T., Varrichio, C., Stanley, K. O., and Miikkulainen, R. (2006). Integration and evaluation of exploration-based learning in games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, Piscataway, NJ. IEEE.

Kendall, G. and Lucas, S. M., editors (2005). *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, Piscataway, NJ. IEEE.

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.

Laird, J. and van Lent, M., editors (2006). *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*, Menlo Park, CA. AAAI Press.

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 and the 12th Annual Conference on Innovative Applications of Artificial Intelligence*, Menlo Park, CA. AAAI Press.

Lamarck, J. B. (1809/1984). *Zoological Philosophy: An Exposition with Regard to the Natural History of Animals*. University of Chicago Press, Chicago. Translated from the French *Philosophie Zoologique* by Hugh Elliot, 1914.

Louis, S. M. and Kendall, G., editors (2006). *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 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.

Lucas, S. M. and Kendall, G. (2006). Evolutionary computation and games. *IEEE Computational Intelligence Magazine*, 1:10–18.

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

Maudlin, M. L., Jacobson, G., Appel, A., and ard 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)*.

McQuesten, P. (2002). *Cultural Enhancement of Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX. Technical Report AI-02-295.

Michie, D. (1961). Trial and error. *Penguin Science Survey*, 2:129–145.

Mjolsness, E., Sharp, D. H., and Alpert, B. K. (1989). Scaling, machine learning, and genetic neural nets. *Advances in Applied Mathematics*, 10:137–163.

Molineaux, M. (2004). *TIELT (v0.5 Alpha) User's Manual*.

Molineaux, M. and Aha, D. W. (2005). TIELT: A testbed for gaming environments. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (Intelligent Systems Demonstrations)*, Menlo Park, CA. AAAI Press.

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. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32.

Moriarty, D. E. and Miikkulainen, R. (1997). Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5:373–399.

Nolfi, S., Elman, J. L., and Parisi, D. (1994). Learning and evolution in neural networks. *Adaptive Behavior*, 2:5–28.

Opitz, D. W. and Shavlik, J. W. (1997). Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research*, 6:177–209.

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.

Ponsen, M. J. V., Lee-Urban, S., Muoz-Avila, H., Aha, D. W., and Molineaux, M. (2005). Stratagus: An open-source game engine for research in real-time strategy games. Technical Report AIC-05-127, Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

Potter, M. A. and Jong, K. A. D. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8:1–29.

Pujol, J. C. F. and Poli, R. (1997). Evolution of the topology and the weights of neural networks using genetic programming with a dual representation. Technical Report CSRP-97-7, School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK.

Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimization. *Neural computing and applications*, 1(1):67–90.

Reisinger, J., Stanley, K. O., and Miikkulainen, R. (2004). Evolving reusable neural modules. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

Revello, T. and McCartney, R. (2002). Generating war game strategies using a genetic algorithm. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1086–1091, Piscataway, NJ. IEEE.

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)*, pages 768–775. San Francisco: Kaufmann.

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

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

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

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

Schaeffer, J. (1997). *One Jump Ahead*. Springer, Berlin.

Siddiqi, A. A. and Lucas, S. M. (1998). A comparison of matrix rewriting versus direct encoding for evolving neural networks. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 392–397, Piscataway, NJ. IEEE.

Spronck, P. (2005). *Adaptive Game AI*. PhD thesis, Maastricht University, the Netherlands.

Stanley, K. O. (2003). *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX.

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

Stanley, K. O., Bryant, B. D., Karpov, I., and Miikkulainen, R. (2006). Real-time evolution of neural networks in the NERO video game. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2003). Evolving adaptive neural networks with and without adaptive synapses. In *Proceedings of the 2003 Congress on Evolutionary Computation*, 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., Kohl, N., and Miikkulainen, R. (2005c). Neuroevolution of an automobile crash warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

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

Stanley, K. O. and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9:93–130.

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*.

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

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

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.

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

Valsalam, V., Bednar, J. A., and Miikkulainen, R. (2005). Constructing good learners using evolved pattern generators. In Beyer, H.-G. et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2005*, pages 11–18. New York: ACM.

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

Whiteson, S., Kohl, N., Miikkulainen, R., and Stone, P. (2005a). Evolving keepaway soccer players through task decomposition. *Machine Learning*, 59:5–30.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., and Kohl, N. (2005b). Automatic feature selection in neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

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

Wieland, A. (1991). Evolving neural network controllers for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks* (Seattle, WA), pages 667–673. Piscataway, NJ: IEEE.

Yannakakis, G., Levine, J., and Hallam, J. (2004). An evolutionary approach for interactive computer games. In *Evolutionary Computation, 2004. CEC2004. Congress on Evolutionary Computation*, volume 1, pages 986–993, Piscataway, NJ. IEEE.

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

Yao, X. and Liu, Y. (1996). Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90.

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., Miikkulainen, R., and Karpov, I. (2006). Incorporating advice into evolution of neural networks. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*.

Yoshioka, T., Ishii, S., and Ito, M. (1998). Strategy acquisition for the game Othello based on reinforcement learning. In Usui, S. and Omori, T., editors, *Proceedings of the Fifth International Conference on Neural Information Processing*, pages 841–844. Tokyo: IOS Press.

Young, M. and Laird, J., editors (2005). *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*, Menlo Park, CA. AAAI Press.

Zhang, B.-T. and Muhlenbein, H. (1993). Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7:199–220.