# The NERO Real-time Video Game

**Kenneth O. Stanley (**kstanley@cs.utexas.edu**)**
**Bobby D. Bryant (**bdbryant@cs.utexas.edu**)**
**Risto Miikkulainen (**risto@cs.utexas.edu**)**
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712 USA
Technical Report UT-AI-TR-04-312
To appear in:
*IEEE Transactions on Evolutionary Computation*
*Special Issue on Evolutionary Computation and Games* (2005)

**Abstract**

In most modern video games, character behavior is scripted; no matter how many times the player exploits a weakness, that weakness is never repaired. Yet if game characters could learn through interacting with the player, behavior could improve as the game is played, keeping it interesting. This paper introduces the real-time NeuroEvolution of Augmenting Topologies (rtNEAT) method for evolving increasingly complex artificial neural networks in *real-time*, as a game is being played. The rtNEAT method allows agents to change and improve during the game. In fact, rtNEAT makes possible an entirely new genre of video games in which the player *teaches* a team of agents through a series of customized training exercises. In order to demonstrate this concept in the NeuroEvolving Robotic Operatives (NERO) game, the player trains a team of robots for combat. This paper describes results from this novel application of machine learning, and demonstrates that rtNEAT makes possible video games like NERO where agents evolve and adapt in real time. In the future, rtNEAT may allow new kinds of educational and training applications.

## 1 Introduction

The world video game market in 2002 was between $15 billion and $20 billion, larger than even that of Hollywood (Thurrott 2002). Video games have become a facet of many people's lives and the market continues to expand. Because there are millions of interactive players and because video games carry perhaps the least risk to human life of any real-world application, they make an excellent testbed for techniques in artificial intelligence and machine learning (ML). In fact, Laird and van Lent (2000) suggested that interactive video

games are a “killer application” for human-level AI.

One of the most compelling yet least exploited technologies in the video game industry is machine learning. Thus, there is an unexplored opportunity to make video games more interesting and realistic, and to build entirely new genres. Such enhancements may have applications in education and training as well, changing the way people interact with their computers.

In the video game industry, the term *Non-player-character* (NPC) refers to an autonomous computer-controlled agent in the game. This paper focuses on training NPCs as intelligent agents, and the standard AI term *agents* is therefore used to refer to them. The behavior of such agents in current games is often repetitive and predictable. In most video games, simple scripts cannot learn or adapt to control the agents: Opponents will always make the same moves and the game quickly becomes boring. Machine learning could potentially keep video games interesting by allowing agents to change and adapt. However, a major problem with learning in video games is that if behavior is allowed to change, the game content becomes unpredictable. Agents might learn idiosyncratic behaviors or even not learn at all, making the gaming experience unsatisfying. One way to avoid this problem is to train agents offline, and then freeze the results into the final game. However, if behaviors are frozen before the game is released, agents cannot adapt and change in response to the tactics of particular players.

If agents are to adapt and change in real-time, a powerful and reliable machine learning method is needed. This paper describes a novel game built around a real-time enhancement of the NeuroEvolution of Augmenting Topologies method (NEAT; Stanley and Miikkulainen 2002b, 2004a). NEAT evolves increasingly complex neural networks, i.e. it *complexifies*. Real-time NEAT (rtNEAT) is able to complexify neural networks *as the game is played*, making it possible for agents to evolve increasingly sophisticated behaviors in real time. Thus, agent behavior improves visibly during gameplay. The aim is to show that machine learning is indispensable for some kinds of video games to work, and to show how rtNEAT makes such an application possible.

In order to demonstrate the potential of rtNEAT, the Digital Media Collaboratory (DMC) at the University of Texas at Austin initiated the NeuroEvolving Robotic Operatives (NERO) project in October of 2003 (`http://dev.eltlabs.org/nero_public`). This project is based on a proposal for a game based on rtNEAT developed at the *2nd Annual Game Development Workshop on Artificial Intelligence, Interactivity, and Immersive Environments* in Austin, TX (presentation by Kenneth Stanley, 2003). The idea was to create a game in which learning is *indispensable*, in other words, without learning NERO could not exist as a game. In NERO, the player takes the role of a trainer, teaching skills to a set of intelligent agents

controlled by rtNEAT. Thus, NERO is a powerful demonstration of how machine learning can open up new possibilities in gaming and allow agents to adapt.

NERO opens up new opportunities for interactive machine learning in entertainment, education, and simulation. This paper describes rtNEAT and NERO, and reviews results from the first year of this ongoing project. The next section presents a brief taxonomy of games that use learning, placing NERO in broader context. NEAT is then described, including how it was enhanced to create rtNEAT. The last sections describe NERO and summarize of the current status and performance of the game.

## 2 Background

Early successes in applying machine learning (ML) to board games have motivated more recent work in live-action video games. For example, Samuel (1959) trained a computer to play checkers using a method similar to *temporal difference learning* (Sutton 1988) in the first application of machine learning (ML) to games. Since then, board games such as tic-tac-toe (Gardner 1962; Michie 1961), backgammon (Tesauro and Sejnowski 1987), Go (Richards et al. 1997), and Othello (Yoshioka et al. 1998) have remained popular applications of ML. A comprehensive survey of machine learning in board games was given by Fürnkranz (2001). Recently, interest has been growing in applying ML to video games (Laird and van Lent 2000). For example Geisler (2002) trained agents to run and shoot opponents using supervised learning techniques. This section examines how machine learning can be applied to video games.

From the human game player's perspective there are two types of learning in games. First, in *out-game learning* (OGL), game developers use ML techniques to pretrain agents that no longer learn after the game is shipped. Second, during *in-game learning* (IGL), agents learn as the player interacts with them in the game; the player can either purposefully direct the learning process or the agents can adapt autonomously to the player's behavior. Most applications of ML to games have used OGL, though the distinction may be blurred from the researcher's perspective when online learning methods are used for OGL. However, the difference between OGL and IGL is important to players and marketers, and ML researchers will frequently need to make a choice between the two.

In a *Machine Learning Game* (MLG), the player *explicitly* attempts to train agents as part of IGL. Since such explicit training requires powerful learning methods, MLGs have not been possible until recently, and thus represent a new genre of video games. Although some conventional game designs include a "training" phase during which the player accumulates resources or technologies in order to advance in levels, such

games are not MLGs because the agents are not actually adapting or learning.

Prior examples in the MLG genre include the *Tamagotchi* virtual pet[1] and the video "God game" *Black & White*[2]. In both games, the player shapes the behavior of game agents with positive or negative feedback. It is also possible to train agents by human example during the game, as van Lent and Laird (2001) described in their experiments with *Quake II*[3]. While these examples demonstrated that limited learning is possible in a game, NERO is an entirely new kind of MLG; it uses a reinforcement learning method (neuroevolution) to optimize a fitness function that is dynamically specified *by the player* while watching and interacting with the learning agents. Thus agent behavior continues to improve as long as the game is played.

A flexible and powerful ML method is needed to allow agents to adapt during gameplay. It is not enough to simply script several key agent behaviors because adaptation would then be limited to the foresight of the programmer who wrote the script, and agents would only be choosing from among a limited menu of options. Moreover, because agents need to learn online as the game is played, predetermined training targets are usually not available, ruling out supervised techniques such as backpropagation (Rumelhart et al. 1986) and decision tree learning (Utgoff 1989).

Traditional reinforcement learning (RL) techniques such as Q-Learning (Watkins and Dayan 1992) and Sarsa($\lambda$) with a Case-Based function approximator (SARSA-CABA; Santamaria et al. 1998) can adapt in domains with sparse feedback (Kaelbling et al. 1996; Sutton and Barto 1998; Watkins and Dayan 1992) and thus can be applied to video games as well. These techniques learn to predict the long-term reward for taking actions in different states by exploring the state space and keeping track of the results. However, video games have several properties that pose serious obstacles to traditional RL:

1. **Large state/action space**. Since games usually have several different types of objects and characters, and many different possible actions, the state/action space that RL must explore is high-dimensional. Not only does this pose the usual problem of encoding a high-dimensional space (Sutton and Barto 1998), but in a real-time game there is the additional challenge of checking the value of every possible action on every game tick for every agent in the game, which may overburden the CPU when other tasks such as screen updates must also be computed.

2. **Diverse behaviors**. Agents learning simultaneously in a simulated world should not all converge

[1] *Tamagotchi* is a trademark of Bandai Co., Ltd. of Tokyo, Japan.

[2] *Black & White* is a trademark of Lionhead Studios, Ltd. of Guildford, UK.

[3] *Quake II* is a trademark of Id Software, Inc.

to the same behavior because a homogeneous population would make the game boring. Yet since RL techniques are based on convergence guarantees and do not explicitly maintain diversity, such an outcome is likely.

3. **Consistent individual behaviors**. RL depends on occasionally taking a random action in order to explore new behaviors. While this strategy works well in offline learning, players do not want to constantly see individual agents periodically making inexplicable and idiosyncratic moves.

4. **Fast adaptation**. Players do not want to wait hours for agents to adapt. Yet a complex state/action representation can take a long time to learn. On the other hand, a simple representation would limit the ability to learn sophisticated behaviors.

5. **Memory of past states**. If agents remember past events, they can react more convincingly to the present situation. However, such memory requires keeping track of more than the current state, ruling out traditional Markovian methods.

It turns out there is an alternative RL technique called neuroevolution (NE), i.e. the artificial evolution of neural networks using a genetic algorithm, that can meet each of these requirements: (1) NE does not require enumerating the state/action space; it works well in high-dimensional state spaces (Gomez and Miikkulainen 2003b), and only produces a single requested action without checking the values of multiple actions. (2) Diverse populations can be explicitly maintained (Stanley and Miikkulainen 2002b). (3) The behavior of an individual during its lifetime does not change. (4) A *representation* of the solution can be evolved, allowing simple practical behaviors to be discovered quickly in the beginning and later complexified (Stanley and Miikkulainen 2004a). (5) Recurrent neural networks can be evolved that utilize memory (Gomez and Miikkulainen 1999). Thus, NE is a good match for video games. Neural networks also make good controllers for video game agents because they can compute arbitrarily complex functions, can both learn and perform in the presence of noisy inputs, and generalize their behavior to previously unseen inputs (Cybenko 1989; Siegelmann and Sontag 1994). NE has successfully evolved motor-control skills such as those necessary in continuous-state games in many challenging non-Markovian domains (Aharonov-Barki et al. 2001; Floriano and Mondada 1994; Fogel 2001; Gomez and Miikkulainen 1998, 1999, 2003a; Gruau et al. 1996; Harvey 1993; Moriarty and Miikkulainen 1996; Nolfi et al. 1994; Potter et al. 1995; Stanley and Miikkulainen 2004a; Whitley et al. 1993).

Our research group has been applying NE to gameplay for about a decade. Using this approach, we have applied several neuroevolutionary algorithms to board games (Moriarty and Miikkulainen 1993; Mo-

riarty 1997; Richards et al. 1997; Stanley and Miikkulainen 2004b). In *Othello*, NE discovered the *mobility strategy* only a few years after its invention by humans (Moriarty and Miikkulainen 1993). Recent work has focused on higher-level strategies and real-time adaptation, which are needed for success in both continuous and discrete multi-agent games (Agogino et al. 2000; Bryant and Miikkulainen 2003; Stanley and Miikkulainen 2004a). Relatively simple ANN controllers can be trained in games and game-like environments to produce convincing purposeful and intelligent behavior (Agogino et al. 2000; Gomez and Miikkulainen 1998; Moriarty and Miikkulainen 1995a,b, 1996; Richards et al. 1997; Stanley and Miikkulainen 2004a).

The current challenge is to achieve evolution in *real time*, as the game is played. If agents could be evolved in a smooth cycle of replacement, the player could interact with evolution during the game and the many benefits of NE would be available to the video gaming community. This paper introduces such a real-time NE technique, rtNEAT, which is applied to the NERO multi-agent continuous-state MLG. In NERO, agents must master both motor control and higher-level strategy to win the game. The player acts as a trainer, teaching a team of robots the skills they need to survive. The next section reviews the NEAT neuroevolution method, and how it can be enhanced to produce rtNEAT.

## 3 Real-time NeuroEvolution of Augmenting Topologies (rtNEAT)

The rtNEAT method is based on NEAT, a technique for evolving neural networks for complex reinforcement learning tasks using a genetic algorithm (GA). NEAT combines the usual search for the appropriate network weights with *complexification* of the network structure, allowing the behavior of evolved neural networks to become increasingly sophisticated over generations. This approach is highly effective: NEAT outperforms other neuroevolution (NE) methods e.g. on the benchmark double pole balancing task (Stanley and Miikkulainen 2002a,b). In addition, because NEAT starts with simple networks and expands the search space only when beneficial, it is able to find significantly more complex controllers than fixed-topology evolution, as demonstrated in a robotic strategy-learning domain (Stanley and Miikkulainen 2004a). These properties make NEAT an attractive method for evolving neural networks in complex tasks such as video games.

Like most GAs, NEAT was originally designed to run *offline*. Individuals are evaluated one or two at a time, and after the whole population has been evaluated, a new population is created to form the next generation. In other words, in a normal GA it is not possible for a human to interact with the evolving agents *while they are evolving*. This section first describes the original offline NEAT method, and then describes how it can be modified to make it possible for players to interact with evolving agents in real time.

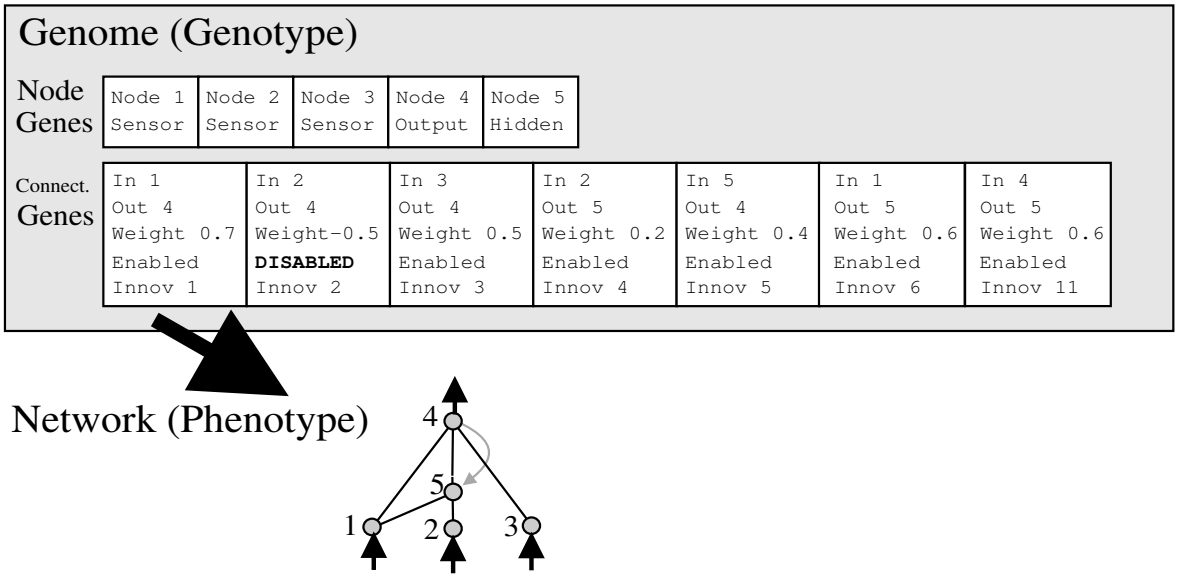



Figure 1: **A NEAT genotype to phenotype mapping example.** A genotype is depicted that produces the shown phenotype. There are 3 input nodes, one hidden, one output node, and seven connection definitions, one of which is recurrent. The second gene is disabled, so the connection that it specifies (between nodes 2 and 4) is not expressed in the phenotype. In order to allow complexification, genome length is unbounded.

The NEAT method consists of solutions to three fundamental challenges in evolving neural network topology: (1) What kind of genetic representation would allow disparate topologies to crossover in a meaningful way? The solution is to use historical markings to line up genes with the same origin. (2) How can topological innovation that needs a few generations to optimize be protected so that it does not disappear from the population prematurely? The solution is to separate each innovation into a different species. (3) How can topologies be minimized *throughout evolution* so the most efficient solutions will be discovered? The solution is to start from a minimal structure and add nodes and connections incrementally. This section explains how each of these solutions is implemented in NEAT.

The section begins by explaining the genetic encoding used in NEAT. Structural mutations are introduced, allowing genomes to grow in NEAT. Historical markings are applied whenever a genome grows; crossover uses historical markings as a way of addressing the competing conventions problem. NEAT's approach to speciation using fitness sharing is introduced as a way to protect innovation, allowing NEAT to grow networks from a minimal starting point. Finally, the last section explains how NEAT was modified to work in real time.

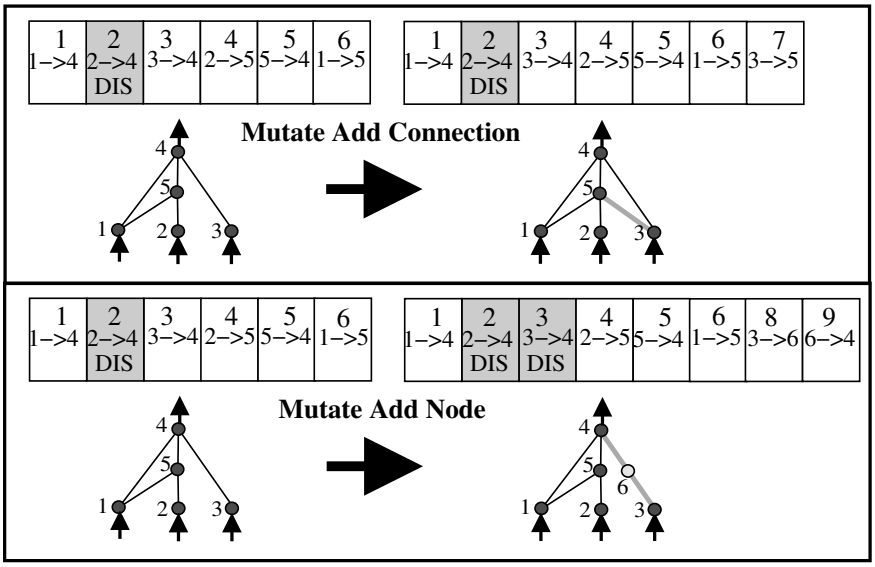



Figure 2: **The two types of structural mutation in NEAT.** Both types, adding a connection and adding a node, are illustrated with the genes above their phenotypes. The top number in each genome is the *innovation number* of that gene. The bottom two numbers denote the two nodes connected by that gene. The weight of the connection, also encoded in the gene, is not shown. The symbol DIS means that the gene is disabled, and therefore not expressed in the network. The figure shows how connection genes are appended to the genome when a new connection and a new node is added to the network. Assuming the depicted mutations occurred one after the other, the genes would be assigned increasing innovation numbers as the figure illustrates, thereby allowing NEAT to keep an implicit history of the origin of every gene in the population.

### 3.1 Genetic Encoding

Evolving structure requires a flexible genetic encoding. In order to allow structures to complexify, their representations must be dynamic and expandable. Each genome in NEAT includes a list of *connection genes*, each of which refers to two *node genes* being connected (Figure 1). Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an *innovation number*, which allows finding corresponding genes during crossover.

Mutation in NEAT can change both connection weights and network structures. Connection weights mutate as in any NE system, with each connection either perturbed or not. Structural mutations, which form the basis of complexification, occur in two ways (Figure 2). Each mutation expands the size of the genome by adding genes. In the *add connection* mutation, a single new connection gene is added connecting two previously unconnected nodes. In the *add node* mutation, an existing connection is split and the new node placed where the old connection used to be. The old connection is disabled and two new connections are added to the genome. The connection between the first node in the chain and the new node is given a weight of one, and the connection between the new node and the last node in the chain is given the

same weight as the connection being split. Splitting the connection in this way introduces a nonlinearity (i.e. sigmoid function) where there was none before. This nonlinearity changes the function only slightly, and the new node is immediately integrated into the network. Old behaviors encoded in the preexisting network structure are not destroyed and remain qualitatively the same, while the new structure provides an opportunity to elaborate on these original behaviors.

Through mutation, the genomes in NEAT will gradually get larger. Genomes of varying sizes will result, sometimes with different connections at the same positions. Crossover must be able to recombine networks with differing topologies, which can be difficult (Radcliffe 1993). The next section explains how NEAT addresses this problem.

### 3.2 Tracking Genes through Historical Markings

It turns out that the historical origin of each gene can be used to tell us exactly which genes match up between *any* individuals in the population. Two genes with the same historical origin represent the same structure (although possibly with different weights), since they were both derived from the same ancestral gene at some point in the past. Thus, all a system needs to do is to keep track of the historical origin of every gene in the system.

Tracking the historical origins requires very little computation. Whenever a new gene appears (through structural mutation), a *global innovation number* is incremented and assigned to that gene. The innovation numbers thus represent a chronology of every gene in the system. As an example, let us say the two mutations in Figure 2 occurred one after another in the system. The new connection gene created in the first mutation is assigned the number 7, and the two new connection genes added during the new node mutation are assigned the numbers $8$ and $9$. In the future, whenever these genomes cross over, the offspring will inherit the same innovation numbers on each gene. Thus, the historical origin of every gene in the system is known throughout evolution.

A possible problem is that the same structural innovation will receive different innovation numbers in the same generation if it occurs by chance more than once. However, by keeping a list of the innovations that occurred in the current generation, it is possible to ensure that when the same structure arises more than once through independent mutations in the same generation, each identical mutation is assigned the same innovation number. Extensive experimentation established that resetting the list every generation as opposed to keeping a growing list of mutations throughout evolution is sufficient to prevent innovation numbers from exploding.

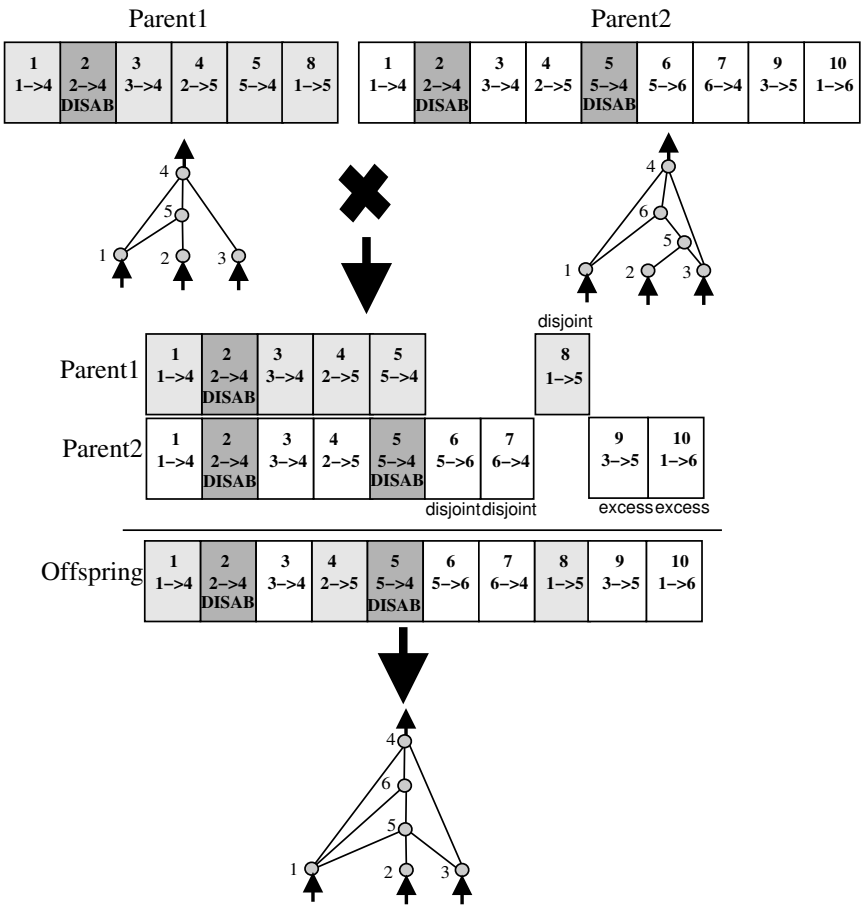



Figure 3: **Matching up genomes for different network topologies using innovation numbers.** Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us that several of their genes match up even without topological analysis. A new structure that combines the overlapping parts of the two parents as well as their different parts can be created in crossover. In this case, equal fitnesses are assumed, so each disjoint and excess gene is inherited from either parent randomly. Otherwise the genes would be inherited from the more fit parent. The disabled genes may become enabled again in future generations: There is a preset chance that an inherited gene is enabled if it is disabled in either parent.

Through innovation numbers, the system now knows exactly which genes match up with which (Figure 3). Genes that do not match are either *disjoint* or *excess*, depending on whether they occur within or outside the range of the other parent's innovation numbers. When crossing over, the genes with the same innovation numbers are lined up and crossed over in one of two ways. In the first method, matching genes are randomly chosen for the offspring genome; alternatively, the connection weights of matching genes can be averaged (Wright 1991). NEAT uses both types of crossover. Genes that do not match are inherited from the more fit parent, or if they are equally fit, each gene is inherited from either parent randomly. Disabled genes have a chance of being reenabled during crossover, allowing networks to make use of older genes once again.

Historical markings allow NEAT to perform crossover without analyzing topologies. Genomes of differ-

ent organizations and sizes stay compatible throughout evolution, and the variable-length genome problem is essentially avoided. This methodology allows NEAT to complexify structure while different networks still remain compatible. However, it turns out that it is difficult for a population of varying topologies to support new innovations that add structure to existing networks, Because smaller structures optimize faster than larger structures, and adding nodes and connections usually initially decreases the fitness of the network, recently augmented structures have little hope of surviving more than one generation even though the innovations they represent might be crucial towards solving the task in the long run. The solution is to protect innovation by speciating the population, as explained in the next section.

### 3.3 Protecting Innovation through Speciation

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 they have to compete with other niches in the population. In addition, speciation prevents bloating of genomes: Species with smaller genomes survive as long as their fitness is competitive, ensuring that small networks are not replaced by larger ones unnecessarily. Protecting innovation through speciation follows the philosophy that new ideas must be given time to reach their potential before they are eliminated.

Historical markings make it possible for the system to divide the population into species based on how similar they are topologically. We can measure the distance $\delta$ between two network encodings as a linear combination of the number of excess ($E$) and disjoint ($D$) genes, as well as the average weight differences of matching genes ($\overline{W}$):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}. \tag{1}$$

The coefficients $c_1$, $c_2$, and $c_3$ adjust the importance of the three factors, and the factor $N$, the number of genes in the larger genome, normalizes for genome size ($N$ can be set to 1 unless both genomes are excessively large). Genomes are tested one at a time; if a genome's distance to a randomly chosen member of the species is less than $\delta_t$, a compatibility threshold, it is placed into this species. Each genome is placed into the first species from the *previous generation* where this condition is satisfied, so that no genome is in more than one species. Keeping the same set of species from one generation to the next allows NEAT to remove stagnant species, i.e. species that have not improved for too many generations. If a genome is not compatible with any existing species, a new species is created. The problem of choosing the best value for $\delta_t$ can be avoided by making $\delta_t$ *dynamic*; that is, given a target number of species, the system can slightly raise $\delta_t$ if there are too many species, and lower $\delta_t$ if there are too few.

Let $P$ be the entire population. The algorithm for clustering genomes into species consists of two loops:

- The Genome Loop:
  - Take next genome $g$ from $P$
  - The Species Loop:
    - If all species in S have been checked, create new species $s_{new}$ and place $g$ in it
    - Else
      - get next species $s$ from $S$
      - If $g$ is compatible with $s$, add $g$ to $s$
    - If $g$ has not been placed, Species Loop
  - If not all genomes in $G$ have been placed, Genome Loop
  - Else STOP

As the reproduction mechanism, NEAT uses *explicit fitness sharing* (Goldberg and Richardson 1987), where organisms in the same species must share the fitness of their niche. Thus, a species cannot afford to become too big even if many of its organisms perform well. Therefore, any one species is unlikely to take over the entire population, which is crucial for speciated evolution to support a variety of topologies. The adjusted fitness $f'_i$ for organism $i$ is calculated according to its distance $\delta$ from every other organism $j$ in the population:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))}. \tag{2}$$

The sharing function $\mathrm{sh}$ is set to 0 when distance $\delta(i,j)$ is above the threshold $\delta_t$; otherwise, $\mathrm{sh}(\delta(i,j))$ is set to 1 (Spears 1995). Thus, $\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))$ reduces to the number of organisms in the same species as organism $i$. This reduction is natural since species are already clustered by compatibility using the threshold $\delta_t$. Every species is assigned a potentially different number of offspring in proportion to the sum of adjusted fitnesses $f'_i$ of its member organisms. The net effect of fitness sharing in NEAT can be summarized as follows. Let $\overline{F_k}$ be the average fitness of species $k$ and $|P|$ be the size of the population. Let $\overline{F}_{tot} = \sum_k \overline{F_k}$ be the total of all species fitness averages. The number of offspring $n_k$ allotted to species $k$ is:

$$n_k = \frac{\overline{F_k}}{\overline{F}_{tot}} |P|. \tag{3}$$

Species reproduce by first eliminating the lowest performing members from the population. The entire population is then replaced by the offspring of the remaining individuals in each species.

The net effect of speciating the population is that structural innovation is protected. The final goal of the system, then, is to perform the search for a solution as efficiently as possible. This goal is achieved through complexification from a simple starting structure, as detailed in the next section.

### 3.4 Minimizing Dimensionality through Complexification

Unlike other systems that evolve network topologies and weights (Angeline et al. 1993; Gruau et al. 1996; Yao 1999; Zhang and Muhlenbein 1993), NEAT begins with a uniform population of simple networks with no hidden nodes, differing only in their initial random weights. Speciation protects new innovations, allowing diverse topologies to gradually accumulate over evolution. Thus, because NEAT protects innovation using speciation, it can start in this manner, minimally, and grow new structure over generations.

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, significantly reducing the number of generations necessary to find a solution, and ensuring that networks become no more complex than necessary. This gradual increase in complexity over generations is *complexification*. In other words, NEAT searches for the optimal topology by incrementally complexifying existing structure.

In previous work, each of the three main components of NEAT (i.e. historical markings, speciation, and starting from minimal structure) were experimentally ablated in order to demonstrate how they contribute to performance (Stanley and Miikkulainen 2002b). The ablation study demonstrated that all three components are interdependent and necessary to make NEAT work. The next section explains how NEAT can be enhanced to work in real time.

### 3.5 Running NEAT in Real Time

NEAT is a powerful algorithm that can evolve increasingly complex structures. However, it evaluates one complete generation of individuals sequentially before creating the next generation. Real-time neuroevolution is based on the observation that in a video game, the entire population of agents plays *at the same time*. Therefore, unlike in offline genetic algorithms such as NEAT, agent fitness statistics are constantly collected as the game is played, and the agents are evolved continuously. The question is when the agents can be replaced with new ones so offspring can be evaluated.

Replacing the entire population together on each generation would look incongruous since everyone's

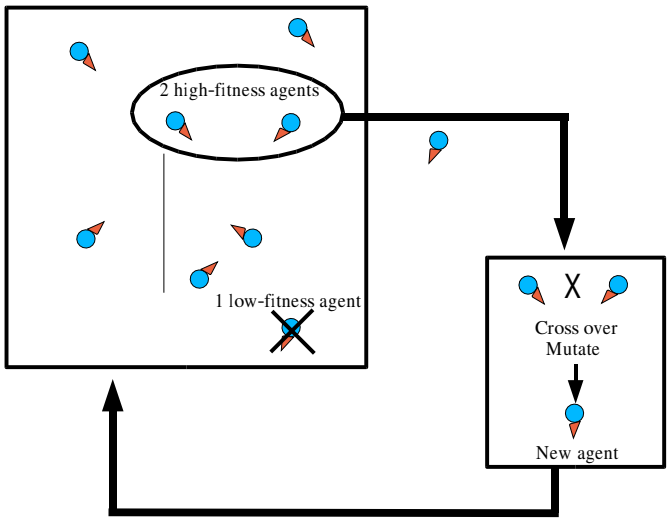



Figure 4: **The main replacement cycle in rtNEAT.** Robot game agents (represented as small circles) are depicted playing a game in the large box. Every few ticks, two high-fitness robots are selected to produce an offspring that replaces another of lower fitness. This cycle of replacement operates continually throughout the game, creating a constant turnover of new behaviors.

behavior would change at once. In addition, behaviors would remain static during the large gaps of time between generations. Instead, in rtNEAT, a single individual is replaced every few game ticks (as in e.g. (m,1)-ES; Beyer and Paul Schwefel 2002). 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 (figure 4).

Real-time evolution was first implemented using conventional neuroevolution (Agogino et al. 2000) before NEAT was developed. However, conventional neuroevolution is not sufficiently powerful to meet the demands of modern video games. In contrast, a real-time version of NEAT offers the advantages of NEAT: Agent neural networks can become increasingly sophisticated and complex during gameplay. The challenge is to preserve the usual dynamics of NEAT, namely protection of innovation through speciation and complexification. While original NEAT normally assigns offspring to species *en masse* for each new generation, rtNEAT cannot allocate space for an entire species at once since it only produces one new offspring at a time. Therefore, a new reproduction cycle must be introduced to allow rtNEAT to speciate in real-time with the same results.

The main loop in rtNEAT works as follows. Let $f_i$ be the fitness of individual $i$. Recall that fitness sharing adjusts it to $\frac{f_i}{|S|}$, where $|S|$ is the number of individuals in the species (Section 3.3). In other words,

fitness is reduced proportionally to the size of the species. This adjustment is important because selection in rtNEAT must be based on adjusted fitness rather than original fitness in order to maintain the same dynamics as NEAT. In addition, because the number of offspring assigned to a species in NEAT is based on its average fitness $\overline{F}$, this average must always be kept up-to-date. Thus, after every $n$ ticks of the game clock, rtNEAT performs the following operations:

1. Remove the agent with the worst *adjusted* fitness from the population assuming one has been alive sufficiently long so that it has been properly evaluated.
2. Re-estimate $\overline{F}$ for all species
3. Choose a parent species to create the new offspring
4. Adjust $C_t$ dynamically and *reassign* all agents to species
5. Place the new agent in the world

Each of these steps is discussed in more detail below.

### 3.5.1 Step 1: Removing the worst agent

The goal of this step is to remove a poorly performing agent from the game, hopefully to be replaced by something better. The agent must be chosen carefully to preserve speciation dynamics. If the agent with the worst *unadjusted* fitness were chosen, fitness sharing could no longer protect innovation because new topologies would be removed as soon as they appear. Thus, the agent with the worst *adjusted* fitness should be removed, since adjusted fitness takes into account species size, so that new smaller species are not removed as soon as they appear.

It is also important not to remove agents that are too young. In original NEAT, *age* is not considered since networks are generally all evaluated for the same amount of time. However, in rtNEAT, new agents are constantly being born, meaning different agents have been around for different lengths of time. It would be dangerous to remove agents that are too young because they have not played for long enough to accurately assess their fitness. Therefore, rtNEAT only removes agents who have played for more than the minimum amount of time $m$.

### 3.5.2 Step 2: Re-estimating $\overline{F}$

Assuming there was an agent old enough to be removed, its species now has one less member and therefore its average fitness $\overline{F}$ has likely changed. It is important to keep $\overline{F}$ up-to-date because $\overline{F}$ is used in choosing the parent species in the next step. Therefore, rtNEAT needs to re-estimate $\overline{F}$.

### 3.5.3 Step 3: Choosing the parent species

In original NEAT the number of offspring $n_k$ assigned to species $k$ is $\frac{\overline{F_k}}{\overline{F}_{\text{tot}}}|P|$, where $\overline{F_k}$ is the average fitness of species $k$, $\overline{F}_{\text{tot}}$ is the sum of all the average species' fitnesses, and $|P|$ is the population size (equation 3).

This behavior needs to be approximated in rtNEAT even though $n_k$ cannot be assigned explicitly (since only one offspring is created at a time). Given that $n_k$ is proportional to $\overline{F}$, the parent species can be chosen probabilistically using the same relationship:

$$Pr(S_k) = \frac{\overline{F_k}}{\overline{F}_{\text{tot}}}. \tag{4}$$

The probability of choosing a given parent species is proportional to its average fitness compared to the total of all species' average fitnesses. Thus, over the long run, the expected number of offspring for each species is proportional to $n_k$, preserving the speciation dynamics of original NEAT.

### 3.5.4 Step 4: Dynamic Compatibility Thresholding in Real time

Recall from section 3.3 that networks are placed into a species in original NEAT if their compatibility distance from the species' representative is less than the threshold $C_t$. Section 3.3 suggested that one way to avoid the burden of choosing the appropriate $C_t$ is to instead choose a target number of species and let NEAT adjust $C_t$ dynamically to reach the target. If there are too many species, $C_t$ can be raised to be more inclusive; if there are too few, $C_t$ can be lowered to be stricter.

An advantage of this kind of *dynamic compatibility thresholding* is that it keeps the number of species relatively stable. Such stability is particularly important in a real-time video game since the population may need to be small to accommodate CPU resources dedicated to graphical processing, and therefore a sudden explosion in the number of species would be undesirable.

In original NEAT, $C_t$ can be adjusted before the next generation is created, but in rtNEAT changing $C_t$ alone is not sufficient because most of the population simply remains where they are. Just changing a

variable does not cause anything to move to a different species. Therefore, after changing $C_t$ in rtNEAT, the entire population must be reassigned to the existing species based on the new $C_t$. As in original NEAT, if a network does not belong in any species a new species is created with that network as its representative.[4]

### 3.5.5 Step 5: Replacing the old agent with the new one

Since an individual was removed in step 1, the new offspring needs to replace it. How agents are replaced depends on the game. In some games, the neural network can be removed from a body and replaced without doing anything to the body. In others, the body may have died and need to be replaced as well. rtNEAT can work with any of these schemes as long as an old neural network gets replaced with a new one.

Step 5 concludes the steps necessary to approximate original NEAT in real-time. However, there is one remaining issue. The entire loop should be performed at regular intervals, every $n$ ticks: How should $n$ be chosen?

### 3.5.6 Determining the Number of Ticks Between Replacements

If agents are replaced too frequently, they do not live long enough to reach the minimum time $m$ to be evaluated. For example, imagine that it takes 100 ticks to obtain an accurate performance evaluation, but that an individual is replaced in a population of 50 on every tick. No one ever lives long enough to be evaluated and the population always consists of only new agents. On the other hand, if agents are replaced too infrequently, evolution slows down to a pace that the player no longer enjoys.

Interestingly, the appropriate frequency can be determined through a principled approach. Let $I$ be the fraction of the population that is too young and therefore cannot be replaced. As before, $n$ is the ticks between replacements, $m$ is the minimum time alive, and $|P|$ is the population size. A *law of eligibility* can be formulated that specifies what fraction of the population can be expected to be ineligible once evolution reaches a steady state (i.e. after the first few time steps when no one is eligible):

$$I = \frac{m}{|P|n}. \tag{5}$$

According to Equation 5, the larger the population and the more time between replacements, the lower the fraction of ineligible agents. This principle makes sense since in a larger population it takes more time

[4]Depending on the specific game, $C_t$ does not necessarily need to be adjusted and species reorganized as often as every replacement. The number of ticks between adjustments is chosen by the game designer.

to replace the entire population. Also, the more time passes between replacements, the more time the population has to age, and hence fewer are ineligible. On the other hand, the larger the minimum age, the more agents are ineligible because more time is necessary to become eligible.

It is also helpful to think of $\frac{m}{n}$ as the *number* of individuals that must be ineligible at any time; over the course of $m$ ticks, an agent is replaced every $n$ ticks, and all the new agents that appear over $m$ ticks will remain ineligible for that duration since they cannot have been around for over $m$ ticks. For example, if $|P|$ is $50$, $M$ is $500$, and $n$ is $20$, 50% of the population would be ineligible.

Based on the law of eligibility, rtNEAT can decide on its own how many ticks $n$ should lapse between replacements for a preferred level of ineligibility, specific population size, and minimum time between replacements:

$$n = \frac{m}{|P|I}. \tag{6}$$

It is best to let the user choose $I$ because in general it is most critical to performance; if too much of the population is ineligible at one time, the mating pool is not sufficiently large. Equation 6 allows rtNEAT to determine the correct number of ticks between replacements $n$ to maintain a desired eligibility level. In NERO, 50% of the population remains eligible using this technique.

By performing the right operations every $n$ ticks, choosing the right individual to replace and replacing it with an offspring of a carefully chosen species, rtNEAT is able to replicate the dynamics of NEAT in real-time. Thus, it is now possible to deploy NEAT in a real video game and interact with complexifying agents as they evolve. The next section describes such a game.

# 4 NeuroEvolving Robotic Operatives (NERO)

NERO is representative of a new MLG genre that is only possible through machine learning. The idea 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. Of course, for the player to be able to teach agents, the agents must be able to *learn*; rtNEAT is the learning algorithm that makes NERO possible.

In NERO, the learning agents are simulated robots, and the goal is to train a team of robots for military combat. The robots 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

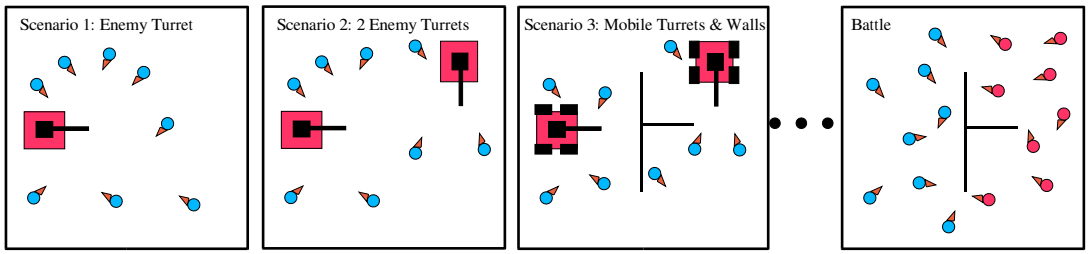



Figure 5: **A turret training sequence (*color figure*).** The figure depicts a sequence of increasingly difficult and complicated training exercises in which the agents attempt to attack turrets without getting hit. In the first exercise there is only a single turret but more turrets are added by the 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 exercise, it is deployed in a real battle against another team.


Figure 6: **Setting up training scenarios (*color figure*).** This screenshot shows items the player can place on the field and sliders used to control behavior. The red robot is a stationary enemy turret that turns back and forth as it shoots repetitively. Behind the turret is a wall. The player can place turrets, other kinds of enemies, and walls anywhere on the training field. On the right is the box containing slider controls. These sliders specify the player's preference for the behavior the team should try to optimize. For example the "E" icon means "approach enemy," and the red bar specifies that the player wants to punish robots that approach the enemy. The crosshair icon represents "hit target," which is being rewarded. The sliders represent fitness components that are used by rtNEAT. The value of the slider is used by rtNEAT as the coefficient of the corresponding fitness component. Through placing items on the field and setting sliders, the player creates training scenarios where learning takes place.

difficult so that the team can begin by learning a foundation of basic skills and then gradually building on them (figure 5). When the player is satisfied that the team is prepared, the team is deployed in a battle against another team trained by another player (possibly on internet), making for a captivating and exciting culmination of training. The challenge is to anticipate the kinds of skills that might be necessary for battle and build training exercises to hone those skills. The next two sections explain how the agents are trained in NERO and how they fight an opposing team in battle.

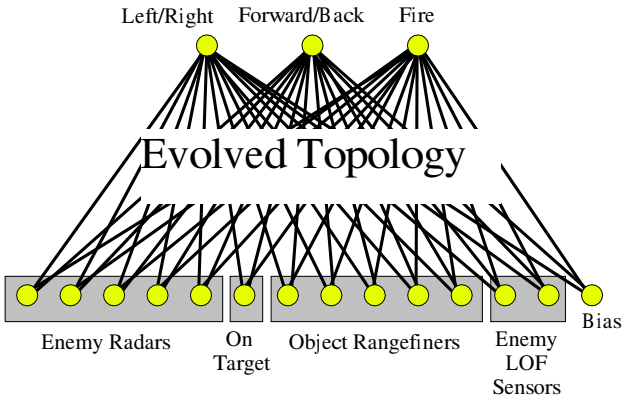


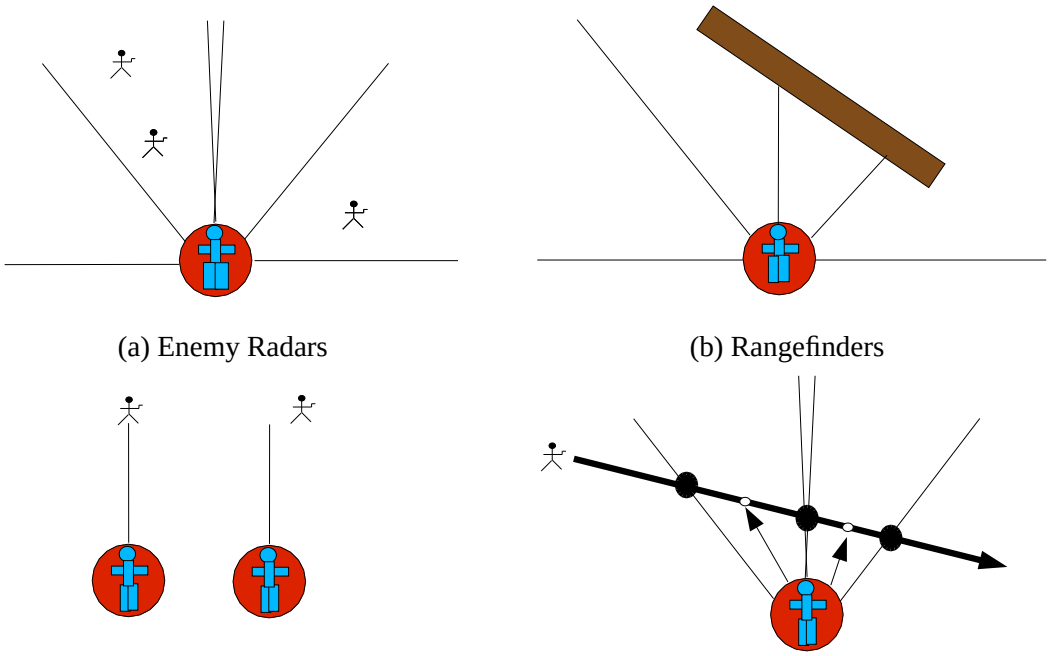

Figure 7: **NERO input sensors and action outputs (*color figure*).** Each NERO robot can see enemies, determine whether an enemy is currently in its line of fire, detect objects and walls, and see the direction the enemy is firing. Its outputs specify the direction of movement and whether or not to 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.

### 4.1 Training Mode

The player sets up training exercises by placing objects on the field and specifying goals through several sliders (figure 6). The objects include static enemies, enemy turrets, rovers (i.e. turrets that move), 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. Fitness is computed as the sum of all these components multiplied by their slider levels, which can be positive or negative. Thus, the player has a natural interface for setting up a training exercise and specifying desired behavior.

Robots have several types of sensors. 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. Figure 7 shows a neural network with the standard set of sensors and outputs, and figure 8 describes how the sensors function.

Training mode is designed to allow the player to set up a training scenario on the field where the robots can continually be evaluated while the worst robot's neural network is replaced every few ticks. Thus, training must provide a standard way for robots to appear on the field in such a way that every robot has an equal chance to prove its worth. To meet this goal, the robots spawn from a designated area of the field called the *factory*. Each robot is allowed a limited time on the field during which its fitness is assessed. When their time on the field expires, robots are transported back to the factory, where they begin another



(a) Enemy Radars

(b) Rangefinders

(c) On-Target Sensor

(d) Line-of-fire sensors

Figure 8: **NERO sensor design.** All NERO sensors are egocentric, i.e. they tell where the objects are from the robot's perspective. (a) Several enemy radar sensors divide the 360 degrees around the robot into slices. Each slice activates a sensor in proportion to how close an enemy is within that slice. If there is more than one enemy in a single slice, their activations are summed. (b) Rangefinders project rays at several angles from the robot. The distance the ray travels before it hits an object is returned as the value of the sensor. Rangefinders are useful for detecting long contiguous objects whereas radars are appropriate for relatively small, discrete objects. (c) The on-target sensor returns full activation only if a ray projected along the front heading of the robot hits an enemy. This sensor tells the robot whether it should attempt to shoot. (d) The line of fire sensors detect where a bullet stream from the closest enemy is heading. Thus, these sensors can be used to avoid fire. They work by computing where the line of fire intersects rays projecting from the robot, giving a sense of the bullet's path. These sensors provide sufficient information for robots to learn successful behaviors for battle.

evaluation. Neural networks are only replaced in robots that have been put back in the factory. The factory ensures that a new neural network cannot get lucky by appearing in a robot that happens to be standing in an advantageous position: All evaluations begin consistently in the factory. In addition, the fitness of robots that survive more than one deployment on the field is updated through a diminishing average that gradually forgets deployments from the distant past. Thus, older robots have more reliable fitness measures since they are averaged over more deployments than younger robots, but their fitness does not become out of date.

The diminishing average fitness is obtained by first computing an average over the first few trials and then maintaining a continuous leaky average. The fitness update rule is,

$$f_{t+1} = f_t + \frac{s_t - f_t}{r} \tag{7}$$

where $f_t$ is the current fitness, $s_t$ is the score from the current evaluation, and $r$ controls the rate of forgetting. The lower $r$ is set, the sooner recent evaluations are forgotten. This method ensures that fitness statistics do not become out of date even for older networks.

Training begins by deploying 50 robots on the field. Each robot is controlled by a neural network with random connection weights and no hidden nodes, as is the usual starting configuration for NEAT (see appendix A for a complete description of the rtNEAT parameters used in NERO). As the neural networks are replaced in real-time, behavior improves dramatically, and robots 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, they face off against teams trained by an opponent player, as will be described next.

## 4.2 Battle Mode

In battle mode, the player discovers how training paid off. A battle team of 20 robots is assembled from as many different training teams as desired. For example, perhaps some robots were trained for close combat while others were trained to stay far away and avoid fire. A player may choose to compose a heterogeneous team from both training sessions.

Battle mode is designed to run over a server so that two players can watch the battle from separate terminals on the internet. 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 robots and perform according to their training. Unlike in training, where being shot does not lead to a robot body being damaged, the robots are actually destroyed after being shot several times in battle. The battle ends when one team is completely eliminated. In some cases, the only surviving robots may insist on avoiding each other, in which case action ceases before one side is completely destroyed. In that case, the winner is the team with the most robots left standing.

The basic battlefield configuration is an empty pen surrounded by four bounding walls, although it is possible to compete on a more complex field, with walls or other obstacles (figure 9). Players train their robots and assemble teams for the particular battle field configuration on which they intend to play. In the experiments described in this chapter, the battlefield was the basic pen.

The next section gives examples of actual NERO training and battle sessions.

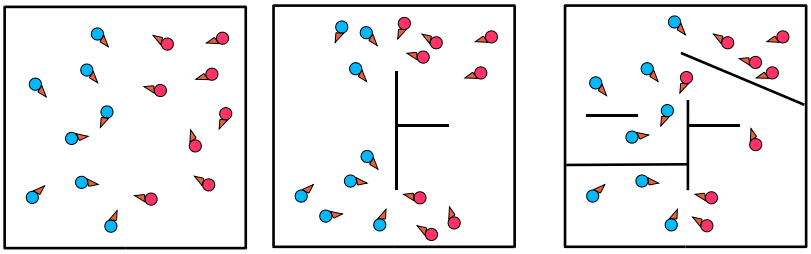


Figure 9: **Battlefield configurations (*color figure*).** The figure shows a range of possible configurations from an open pen to a maze-like environment. Players can construct their own battlefield configurations and train for them. The basic configuration, which is used in section 5, is the empty pen surrounded by four bounding walls.

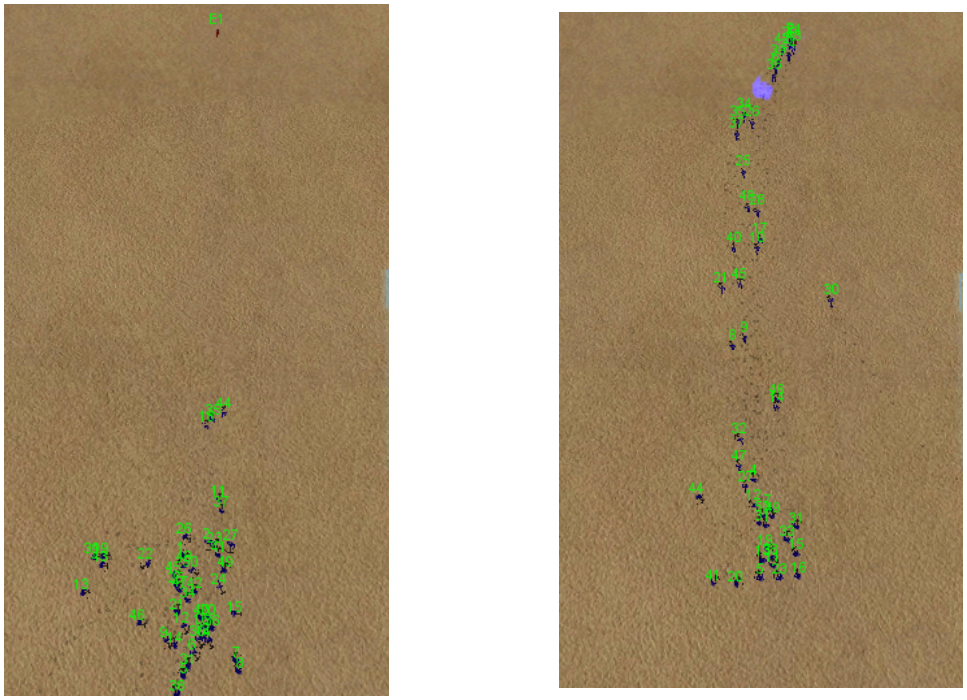
# 5 Playing NERO

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. 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 simulation 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.

The first playable version of NERO was completed in May of 2004. At that time, several NERO programmers trained their own teams and held a tournament. As examples of what is possible in NERO, this section outlines the behaviors evolved for the tournament, the resulting battles, and the real-time performance of NERO and rtNEAT.

NERO is capable of evolving behaviors very quickly in real-time. 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 robots were rewarded for approaching the enemy. This training required robots to learn to run towards a target, which is difficult since robots 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 robots on the field learn to approach the enemy successfully (10 runs, $sd = 44.5$s) It is important to note that the success criterion, i.e. that the team sufficiently learns to approach the enemy, is in part subjective since the player decides when training is complete by visually assessing the team's performance. Nevertheless, success in seeking is generally unambiguous as shown in figure 10.

NERO differs from most applications of GAs in that the quality of evolution is judged from the player's



(a) Five seconds: Mass confusion

(b) 100 seconds: Success

Figure 10: **Learning to approach the enemy (*color figure*).** These screenshots show the training field before and after the robots evolved seeking behavior. The factory is at the bottom of each panel and the enemy being sought is at the top. The numbers above the robots' heads are used to identify individual robots. (a) Five seconds after the training begins, the robots scatter haphazardly around the factory, unable to effectively seek the enemy. (b) After ninety seconds, the robots consistently seek the enemy. Some robots prefer swinging left, while others swing right. These pictures demonstrate that behavior improves dramatically in real-time over only 100 seconds.

perspective based on the performance of the *entire* population. On the other hand GA practitioners generally only look at the champions of a run. However, even though the entire population must solve the task, it does not converge to the same solution. In seek training, some robots evolve 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 robots chose exactly the same path, they would often crash into each other and slow each other down, so naturally some robots 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.

After the robots learned to seek the enemy, they were further trained to fire at the enemy. It is possible to train robots to aim by rewarding them for hitting a target, but that it is also aesthetically unpleasing to players to have to wait while robots fire haphazardly in all directions and slowly figure out how to aim. Therefore, the fire output of neural networks was connected to an aiming script that points the gun properly at the enemy closest to the robot's current heading within some fixed distance. Thus, robots quickly learn to

Figure 11: **Running away backwards (*color figure*).** This training screenshot shows several robots backed up against the wall after running backwards and shooting at the enemy, which is being controlled from a first-person perspective by a human trainer using a joystick. Robots learned to run away from the enemy backwards during avoidance training because that way they can shoot as they flee. Running away backwards is an example of evolution's ability to find novel and effective behaviors.

seek and attack the enemy.

Robots 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 robots on the field. The robots learned to back away in order to avoid being penalized for being too near the enemy. Interestingly, robots preferred to run away from the enemy backwards because that way they could still shoot the enemy. Also, most of their enemy radars are on their front half, giving them better resolution if they remain facing their target (figure 11).

By placing a turret on the field and asking robots to approach the turret without getting hit, robots were able to learn to avoid enemy fire (figure 12). The turret is programmed to periodically rotate back and forth spraying bullets. Robots evolved to run to the opposite side of the turret from the spray and approach it 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, robots were trained to run around walls in order to approach the enemy. As performance improved, players incrementally added more walls until the robots could navigate an entire maze without any path-planning (figure 13)! 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.

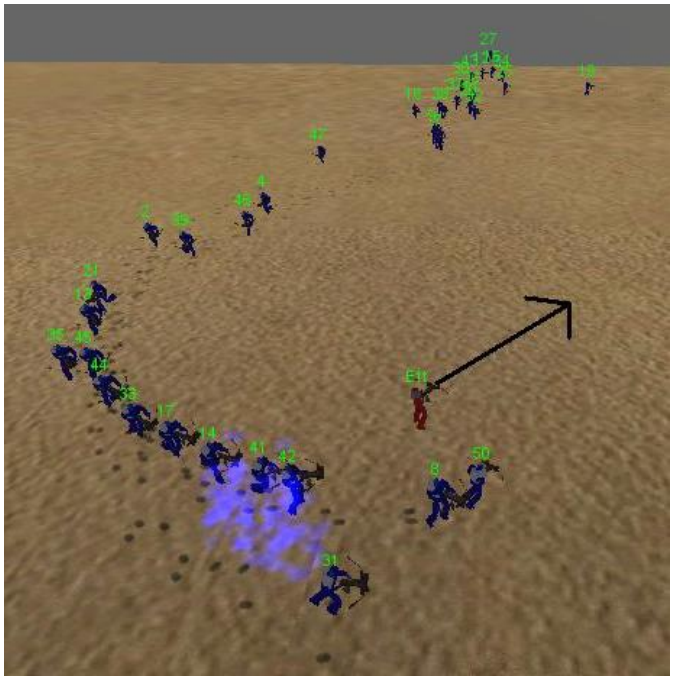
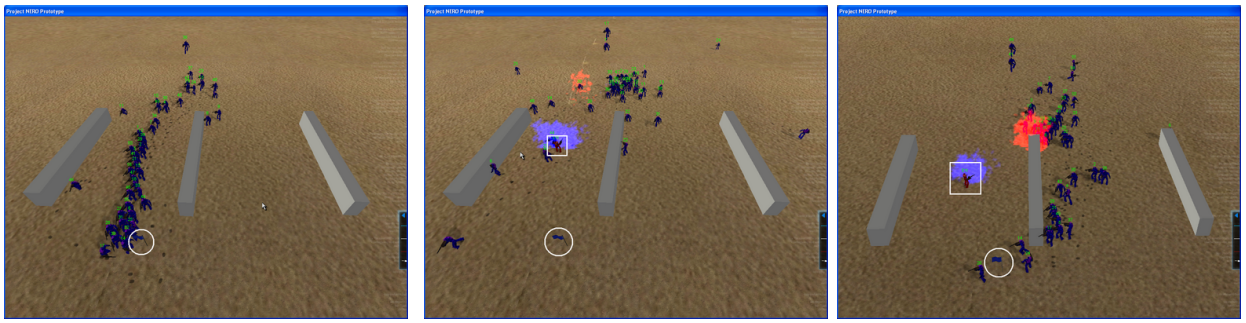


Figure 12: **Avoiding turret fire (*color figure*).** The black arrow points in the current direction of the turret fire (the arrow is not part of the NERO display and is only added for illustration). Robots in training learn to run safely around the enemy's line of fire in order to attack. Notice how they loop around the back of the turret and attack from behind. When the turret moves, the robots change their attack trajectory accordingly. Learning to avoid fire is an important battle skill. The conclusion is that rtNEAT was able to evolve sophisticated, nontrivial behavior in real time.

In a powerful demonstration of real-time adaptation, robots 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 (figure 14). After two minutes, the robots 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 robots 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 from the tournament had only a slight advantage over an avoidant team, winning six out of ten battles, losing three, and tying one (Table 1). The avoidant team runs in a pack to a corner of the field's enclosing wall (figure 15). 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 fairly well-balanced tactics, neither providing a significant advantage over the other. The interesting challenge of NERO is to conceive strategies that are clearly dominant over others.

Figure 13: **Navigating a maze.** Incremental training on increasingly complex wall configurations produced robots that could navigate this maze to find the enemy. The robots spawn from the factory at the top of the maze and proceed down to the enemy at the bottom. In this picture, the green numbers above the robots specify their species. Notice that species "4" evolved to take the path through the right side of the maze while other species take the the left path. This result suggests that protecting innovation in rtNEAT indeed supports a range of diverse behaviors, each with its own network topology.



(a) Robots approach flag (b) Player attacks on left (c) Robots learn new approach

Figure 14: **Video game characters adapt to player's actions (*color figure*).** The robots in these screenshots spawn from the top of the screen and must approach the flag (circled) at the bottom left. (a) The robots first learn to take the left hallway since it is the shortest path to the flag. (b) A human player (identified by a square) attacks inside the left hallway and decimates the robots. (c) Even though the left hallway is the shortest path to the flag, the robots learn that they can avoid the human enemy by taking the right hallway, which is protected from the human's fire by a wall. rtNEAT allows the robots to adapt in this way to the player's tactics in real time.

| Battle Number | Seekers | Avoiders |
|---|---|---|
| 1 | **6** | 0 |
| 2 | 4 | **7** |
| 3 | **8** | 0 |
| 4 | 7 | 7 |
| 5 | **8** | 3 |
| 6 | 6 | **10** |
| 7 | **5** | 4 |
| 8 | **5** | 2 |
| 9 | 3 | **7** |
| 10 | **8** | 0 |

Table 1: **Seekers vs. Avoiders.** Scores from 10 battles are shown between a team trained to aggressively seek and attack the enemy and another team taught to run away backwards and shoot at the same time. The seeking team wins six out of the 10 games, but its advantage is not significant, showing that when strategies contrast they can still be evenly matched. Results like this one can be unexpected, teaching players about the relative strengths and weakness of different tactics.

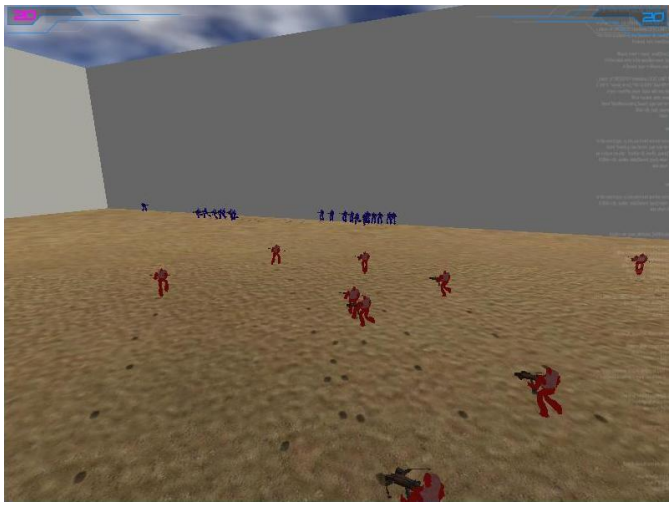


Figure 15: **Seekers chasing avoiders in battle (*color figure*).** In this battle screenshot, red robots trained to seek and attack the enemy pursue blue robots that have backed up against the wall. Teams trained for different tactics are clearly discernable in battle, demonstrating the ability of the training to evolve diverse tactics.

| Battle Number | Wall-fighters | Seekers |
|---|---|---|
| 1 | **7** | 2 |
| 2 | **9** | 0 |
| 3 | **4** | 3 |
| 4 | **7** | 2 |
| 5 | **10** | 0 |
| 6 | **8** | 2 |
| 7 | **12** | 2 |
| 8 | **7** | 2 |
| 9 | **4** | 2 |
| 10 | **9** | 1 |

Table 2: **Wall-fighters vs. Seekers.** The table shows final scores from 10 battles between a team trained to fight near walls and another trained to aggressively seek and attack the enemy. The wall-fighters win every battle because they know how to avoid fire near a wall, while the aggressive team runs directly into fire when fighting near a wall. The total superiority of the wall-fighters shows that the right tactical training indeed matters in battle, and that rtNEAT was able to evolve sophisticated fighting tactics.

One of the best teams was trained by observing a phenomenon that happened consistently in battle. Chases among robots from opposing teams frequently caused robots to eventually reach the field's bounding walls. Particularly for robots trained to avoid turret fire by attacking from behind (figure 12), enemies standing against the wall present a serious problem since it is not possible to go around them. Thus, training a team against a turret with its back against the wall, it was possible to familiarize robots with attacking enemies against a wall. This team learned to hover near the turret and fire when it turned away, but back off quickly when it turned towards them. This tactic works effectively when several friendly robots from the same team are nearby since an enemy can only be facing one direction at a time. In fact, the wall-based team won the first NERO tournament by using this strategy. Table 2 shows that the wall-trained team wins 100% of the time against the aggressive seeking team. Thus, it is possible to learn sophisticated tactics that dominate over simpler ones like seek or avoid.

# 6 Discussion

Participants in the first NERO tournament agreed that the game was engrossing and entertaining. Battles were exciting events for all the participants, evoking plentiful clapping and cheering. Players spent many hours honing behaviors and assembling teams with just the right combination of tactics.

An important point of this project is that NERO would not be possible without rtNEAT. rtNEAT was able to evolve interesting tactics quickly in real-time while players interacted with NERO, showing that neuroevolution can be deployed in a real game and work fast enough to provide entertaining results.

The success of the first NERO prototype suggests that the rtNEAT technology has immediate potential commercial applications in modern games. Any game in which agent behavior is repetitive and boring can be improved by allowing rtNEAT to at least partially modify tactics in real-time. Especially in persistent video games such as Massive Multiplayer Online Games (MMOGs) that last for months or years, the potential for rtNEAT to continually adapt and optimize agent behavior may permanently alter the gaming experience for millions of players around the world.

Since the first tournament took place, new features have been added to NERO, increasing its appeal and complexity. For example, robots can now duck behind walls and learn to run to a flag placed by the player to designate important areas of the field. The game continues to be developed and new features and sensors are constantly being added. The goal is to have a full network-playable version with an easy and intuitive user interface in the near future.

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 in testing as the project matures and standard strategies are identified. At present, the project's main contribution is to show that an entirely new genre of game is possible because of rtNEAT.

NERO is also being used as a common platform for quickly implementing complicated real-time neuroevolution experiments. While video games are intended mainly for entertainment, they are an excellent catalyst for improving machine learning technology. Because of the gaming industry's financial success and low physical risk, it makes sense to explore this area as a stepping stone to other more critical applications. With this new technology, it may finally be possible to use games for training as has long been envisioned. As humans improve in such training games, so could surrounding agents, keeping the simulation realistic for longer than has been possible in the past.

# 7 Conclusion

A real-time version of NEAT (rtNEAT) was developed to allow users to interact with evolving agents. In rtNEAT, an entire population is simultaneously and asynchronously evaluated as it evolves. Using this

method, it was possible to build an entirely new kind of video game, NERO, where the characters adapt in real time in response to the player's actions. In NERO, the player takes the role of a trainer and constructs training scenarios for a team of simulated robots. The rtNEAT technique can form the basis for other similar interactive learning applications in the future, and eventually even make it possible to use gaming as a method for training people in sophisticated tasks.

## Acknowledgments

Special thanks are due to Aaron Thibault and Alex Cavalli of the $\mathrm{IC}^2$ and DMC for supporting the NERO project. The entire NERO team deserves recognition for their contribution to this project including the NERO leads Aliza Gold (Producer), Philip Flesher (Lead Programmer), Jonathan Perry (Lead Artist), Matt Patterson (Lead Designer), and Brad Walls (Lead Programmer). We are grateful also to the original volunteer programming team Ryan Cornelius and Michael Chrien, and newer programmers Ben Fitch, Justin Larrabee, Trung Ngo, and Dustin Stewart-Silverman, and to our volunteer artists Bobby Bird, Brian Frank, Corey Hollins, Blake Lowry, Ian Minshill, and Mike Ward. This research was supported in part by the Digital Media Collaboratory (DMC) Laboratory at the $\mathrm{IC}^2$ Institute (`http://dmc.ic2.org/`), in part by the National Science Foundation under grant IIS-0083776, and in part by the Texas Higher Education Coordinating Board under grant ARP-003658-476-2001. NERO physics is controlled by the Torque Engine, which is licensed from GarageGames (`http://www.garagegames.com/`). NERO's official public website is `http://dmc.ic2.org/nero_public`, and the project's official email address is `nero@cs.utexas.edu`.

## A NERO System Parameters

The coefficients for measuring compatibility were $c_1 = 1.0$, $c_2 = 1.0$, and $c_3 = 0.4$. The initial compatibility distance was $\delta_t = 4.0$. The population was only 50 so that the CPU could accommodate all the agents being evaluated simultaneously. A target of 4 species was assigned. If the number of species grew above 4, $\delta_t$ was increased by $0.3$ to reduce the number of species. Conversely, if the number of species fell below 4, $\delta_t$ was decreased by $0.3$ to increase the number of species. The threshold $\delta_t$ was changed in real time (Section 3.5.4). The interspecies mating rate was 0.001. The probability of adding a new node was 0.05 and the probability of a new link mutation was 0.03. These parameter values were found experimentally but they do follow intuitively meaningful rules: Links need to be added significantly more often than nodes, and

weight differences are given low weight since the population is small. Performance is robust to moderate variations in these values: The dynamic compatibility distance measure caused speciation to remain stable.

The percentage of the population allowed to be ineligible at one time was 50%. The number of ticks between replacements is 20 and the minimum evaluation time is 500. The number of ticks between replacements can also be derived from equation 6.

## References


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

Aharonov-Barki, R., Beker, T., and Ruppin, E. (2001). Emergence of memory-Driven command neurons in evolved artificial agents. *Neural Computation*, 13(3):691–716.

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.

Beyer, H.-G., and Paul Schwefel, H. (2002). Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52.

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

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

Floriano, D., and Mondada, F. (1994). Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*.

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

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

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.

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

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

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

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

Gomez, F. J., and Miikkulainen, R. (2003b). Active guidance for a finless rocket through neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*. Berlin: Springer Verlag.

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*, 81–89. Cambridge, MA: MIT Press.

Harvey, I. (1993). *The Artificial Evolution of Adaptive Behavior*. PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, Sussex.

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

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*. Cambridge, MA: MIT Press.

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

Moriarty, D., and Miikkulainen, R. (1993). Evolving complex Othello strategies with marker-based encoding of neural networks. Technical Report AI93-206, Department of Computer Sciences, The University of Texas at Austin.

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

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

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

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

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

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

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

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

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Cambridge, MA: MIT Press.

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.

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

Spears, W. (1995). Speciation using tag bits. In *Handbook of Evolutionary Computation*. IOP Publishing Ltd. and Oxford University Press.

Stanley, K. O., and Miikkulainen, R. (2002a). Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*. San Francisco, CA: Morgan Kaufmann.

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

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

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

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*. Cambridge, MA: MIT Press.

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.

Thurrott, P. (2002). Top stories of 2001, #9: Expanding video-Game market brings microsoft home for the holidays. *Windows & .NET Magazine Network*.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning*, 4(2):161–186.

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

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

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

Wright, A. H. (1991). Genetic algorithms for real parameter optimization. In Rawlins, G. J. E., editor, *Foundations of Genetic Algorithms*, 205–218. San Francisco, CA: Morgan Kaufmann.

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

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*, 841–844. Tokyo: IOS Press.

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