In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*.
Piscataway, NJ: IEEE **Winner of the Best Paper Award at CIG'05**

Evolving Neural Network Agents in the NERO Video Game

Kenneth O. Stanley
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
kstanley@cs.utexas.edu

Bobby D. Bryant
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
bdbryant@cs.utexas.edu

Risto Miikkulainen
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
risto@cs.utexas.edu

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 during gameplay, 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 a 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 also demonstrates how multiple agents can evolve and adapt in video games like NERO in real time using rtNEAT. In the future, rtNEAT may allow new kinds of educational and training applications that adapt online as the user gains new skills.

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 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 (Laird and van Lent 2000). In fact, Fogel et al. (2004) argue that such techniques can potentially both increase the longevity of video games and decrease their production costs.

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 agents in current games is often repetitive and predictable. In most video games, 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, 2004). 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.ic2.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.

This paper describes rtNEAT and NERO, and reviews results from the first year of this ongoing project. The next section briefly reviews learning methods for games. NEAT is then described, including how it was enhanced to create rtNEAT. The last sections describe NERO and summarize the current status and performance of the game.

2 Background

This section reviews several machine learning techniques that can be used in games, and explains why *neuroevolution* (NE), i.e. the artificial evolution neural networks using a genetic algorithm, is the ideal method for real-time learning in NERO. Because agents in NERO 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(λ) 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) 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 significant challenges 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.
2. **Diverse behaviors**. Agents learning simultaneously should not all converge 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 see an individual agent periodically making inexplicable and idiosyncratic moves relative to its usual behavior.
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. Thus, choosing the right representation is difficult.
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.

While these properties make applying traditional RL techniques difficult, NE is an alternative RL technique that can meet each requirement: (1) NE works well in high-dimensional state spaces (Gomez and Miikkulainen 2003), 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 behaviors to be discovered quickly in the beginning and later complexified (Stanley and Miikkulainen 2004). (5) Recurrent neural networks can be evolved that utilize memory (Gomez and Miikkulainen 1999). Thus, NE is a good match for video games.

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 video game. 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 2004). 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 multiple evolving agents *while they are evolving*. This section first briefly reviews 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. See e.g. Stanley and Miikkulainen (2002a,b, 2004) for a complete description of NEAT.

NEAT is based on three key ideas. First, evolving network structure requires a flexible genetic encoding. Each genome includes a list of *connection genes*, each of which refers to two *node genes* being connected. Each connection gene specifies the in-node, the out-node, the connection weight, whether or not the connection gene is expressed (an enable bit), and an *innovation number*, which allows finding corresponding genes during crossover. Mutation 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 allow complexity to increase, either add a new connection or a new node to the network. Through mutation, genomes of varying sizes are created, sometimes with completely different connections specified at the same positions.

Each unique gene in the population is assigned a unique innovation number, and the numbers are inherited during crossover. Innovation numbers allow NEAT to perform crossover without the need for expensive topological analysis. Genomes of different organizations and sizes stay compatible throughout evolution, and the problem of matching different topologies (Radcliffe 1993) is essentially avoided.

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. The reproduction mechanism for NEAT is *explicit fitness sharing* (Goldberg and Richardson 1987), where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

Third, unlike other systems that evolve network topologies and weights (Gruau et al. 1996; Yao 1999) 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.

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.

3.1 Running NEAT in Real Time

In NEAT, the population is replaced at each generation. However, in real time, replacing the entire population together on each generation would look incongruous since everyone's behavior would change at once. In addition, behaviors would remain static during the large gaps between generations. Instead, in rtNEAT, a single individual is replaced every few game ticks (as in e.g. (μ,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 1). The challenge is to preserve the usual dynamics of NEAT, namely protection of innovation through speciation and complexification.

The main loop in rtNEAT works as follows. Let f_i be the fitness of individual i. Fitness sharing adjusts it to $\frac{f_i}{|S|}$, where $|S|$ is the number of individuals in the species. 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:

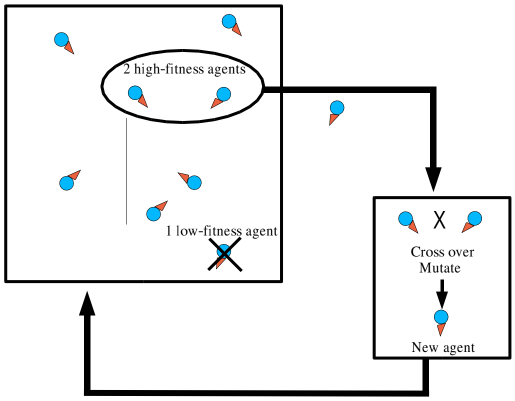


Figure 1: **The main replacement cycle in rtNEAT.** Robot game agents (represented as small circles) are depicted playing a game in the large box. Every few ticks, two high-fitness robots are selected to produce an offspring that replaces another of lower fitness. This cycle of replacement operates continually throughout the game, creating a constant turnover of new behaviors.

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 compatibility threshold 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.

3.1.1 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.

3.1.2 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 $\overline{F}$.

3.1.3 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.

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:

$$Pr(S_k) = \frac{\overline{F_k}}{\overline{F}_{\text{tot}}}. \tag{1}$$

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.

3.1.4 Step 4: Dynamic Compatibility Thresholding

Networks are placed into a species in original NEAT if they are compatible with a representative member of the species. rtNEAT attempts to keep the number of species constant by adjusting a threshold, C_t, that determines whether an individual is compatible with a species' representative. When there are too many species, C_t is increased to make species more inclusive; when there are too few, C_t is decreased to be stricter. An advantage of this kind of *dynamic compatibility thresholding* is that it keeps the number of species relatively stable.

In rtNEAT changing C_t alone cannot immediately affect the number of species 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.[1]

[1] 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.

3.1.5 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?

3.1.6 Determining 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):

$$I = \frac{m}{|P|n}. \tag{2}$$

According to Equation 2, the larger the population and the more time between replacements, the lower the fraction of ineligible agents. Based on the law, 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:

$$n = \frac{m}{|P|I}. \tag{3}$$

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 3 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.

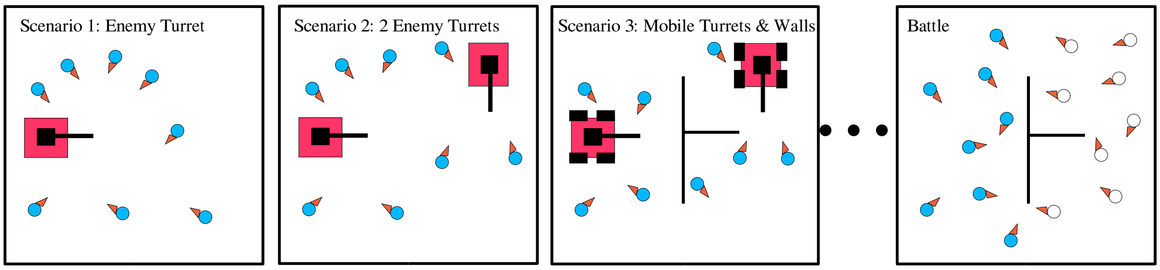


Figure 2: **A turret training sequence.** The figure depicts a sequence of increasingly difficult and complicated training exercises in which the agents attempt to attack turrets without getting hit. In the first exercise there is only a single turret but more turrets are added by the player as the team 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.

4 NeuroEvolving Robotic Operatives (NERO)

NERO is representative of a new 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.

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 specified with a set of sliders. Ideally, the exercises are increasingly difficult so that the team can begin by learning a foundation of basic skills and then gradually building on them (figure 2). 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 the 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.

4.1 Training Mode

The player sets up training exercises by placing objects on the field and specifying goals through several sliders (figure 3). 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. Each individual fitness component is normalized to a Z-score so that each fitness component is measured on the same scale. Fitness is computed as the sum of all these normalized components multiplied by their slider levels. 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 4 shows a neural network with the standard set of sensors and outputs. Several enemy radar sensors divide

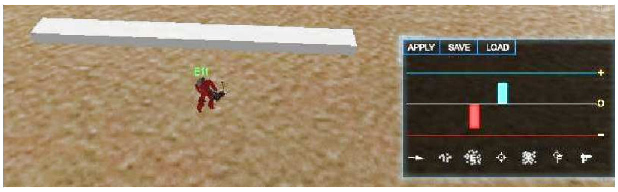


Figure 3: **Setting up training scenarios.** This screenshot shows items the player can place on the field and sliders used to control behavior. The robot is a stationary enemy turret that turns back and forth as it shoots repetitively. Behind the turret is a wall. The player can place turrets, other 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 descending bar above it 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.

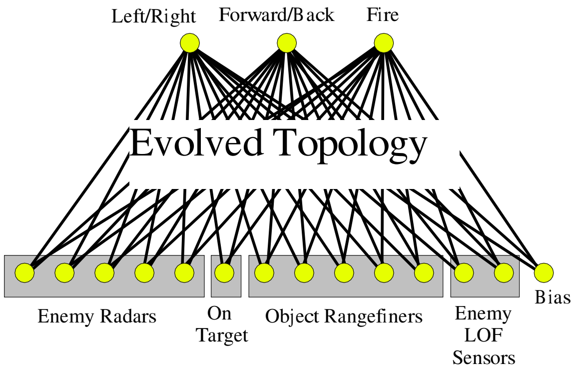


Figure 4: **NERO input sensors and action outputs.** Each NERO robot can see enemies, determine whether an enemy is currently in its line of fire, detect objects and walls, and see the direction the enemy is firing. Its outputs specify the direction of movement and whether or not to fire. This configuration has been used to evolve varied and complex behaviors; other variations work as well and the standard set of sensors can easily be changed.

the 360 degrees around the robot into slices. Each slice activates a sensor in proportion to how close an enemy is within that slice. 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. The on-target sensor returns full activation only if a ray projected along the front heading of the robot hits an enemy. 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. Robots can also be trained with friend radar sensors that allows them to see what each other are doing. The complete sensor set supplies robots with sufficient information to make intelligent tactical decisions.

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 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.

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. 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 can 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

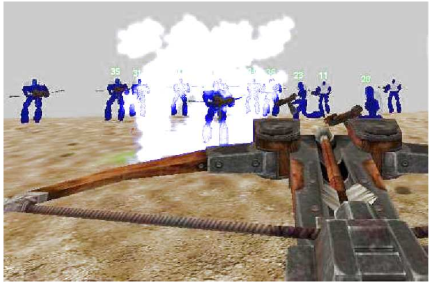


Figure 5: **Running away backwards.** This training screenshot shows several robots backed up against the wall after running backwards and shooting at the enemy, which is being controlled from a first-person perspective by a human trainer using a joystick. Robots learned to run away from the enemy backwards 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.

damaged, the robots are actually destroyed after being shot several times in battle. The battle ends when action ceases either because one team is completely eliminated, or because the remaining robots will not fight. 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. Players train their robots and assemble teams for the particular battlefield 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.

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.

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 can evolve 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).

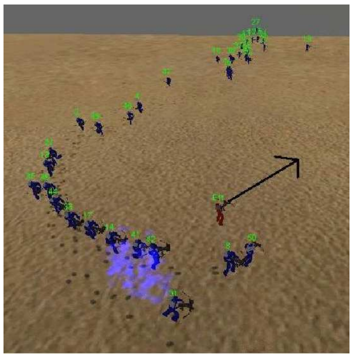


Figure 6: **Avoiding turret fire.** The arrow points in the current direction of the turret fire (the arrow is not part of the NERO display and is only added for illustration). Robots in training learn to run safely around the enemy's line of fire in order to attack. Notice how they loop around the back 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.

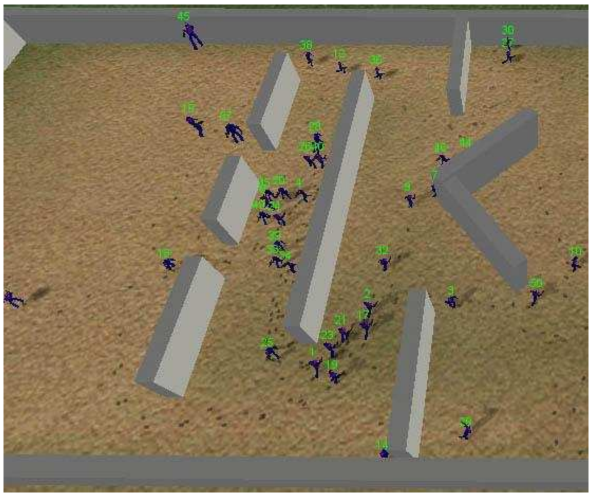


Figure 7: **Navigating a maze.** Incremental training on increasingly complex wall configurations produced robots that could navigate this maze to find the enemy. The robots spawn from the left side of the maze and proceed to an enemy at the right.

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 (figure 5).

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 6).

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 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 7)!

In a powerful demonstration of real-time adaptation with implications beyond NERO, 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 8). 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. Such fast strategic adjustment demonstrates that rtNEAT can be used in existing video game genres as well as in NERO.

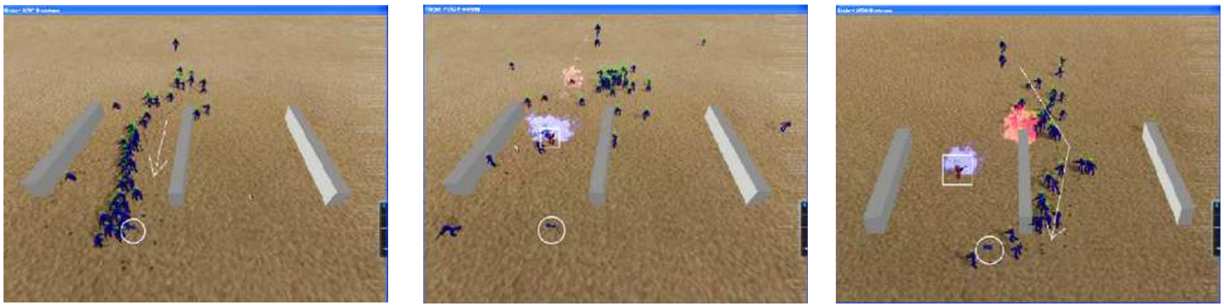
In battle, some teams that were trained differently were nevertheless evenly matched, while some training types consistently prevailed against others For example, an aggressive seeking team from the tournament had only a slight advantage over an avoidant team, winning six out of ten battles, losing three, and tying one. 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 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.

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 6), 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. The wall-based team won the first NERO tournament by using this strategy. 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 for all the participants, evoking plentiful clapping and cheering. Players spent hours honing behaviors and assembling teams with just the right combination of tactics.

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-



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

Figure 8: **Video game characters adapt to player's actions.** The robots in these screenshots spawn from the top of the screen and must approach the flag (circled) at the bottom left. White arrows point in the direction of mass motion. (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.

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.

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 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 (Lead Programmer as of Fall 2004), 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 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`.

Bibliography

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

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

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: Kaufmann.

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

Gomez, F. J., and Miikkulainen, R. (2003). Active guidance for a finless rocket through neuroevolution. In *Proc. of the Genetic and Evolutionary Computation Conf. (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.

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 *Proc. of the 17th Nat. Conf. on Artificial Intelligence and the 12th Annual Conference on Innovative Applications of Artificial Intelligence*. Menlo Park, CA: AAAI Press.

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

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

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.

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: Kaufmann.

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

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

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

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.

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

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