# We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

**4,800**
Open access books available

**122,000**
International authors and editors

**135M**
Downloads

Our authors are among the

**154**
Countries delivered to

**TOP 1%**
most cited scientists

**12.2%**
Contributors from top 500 universities


WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

## Interested in publishing with us?
## Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

# 21

# Applications of Neural-Based Agents in Computer Game Design

Joseph Qualls and David J. Russomanno
*University of Memphis*
*United States*

## 1. Introduction

Most agents in computer games are designed using classical symbolic artificial intelligence (AI) techniques. The AI techniques include production rules for very large branching and conditional statements, as well as search techniques, including branch-and-bound, heuristic search, and A* (Russel & Norvig, 2003). Planning techniques, such as STRIPS (Stanford Research Institute Problem Solver) (Fikes & Nilsson, 1971) and hierarchical task network (HTN) (Erol, 1996) planning are common. Also, situational case-based reasoning, finite-state machines, classical expert systems, Bayesian networks, and other forms of logic, including predicate calculus and its derivatives, such as description logics (Baader et al., 2003), form the foundation of many game agents that leverage AI techniques.

The game agents are typically created with *a priori* knowledge bases of game states and state transitions, including mappings of the world environment and the game agent's reactions to the environment and vice versa (Dybsand, 2000; Zaroinski, 2001; Watt & Policarpo, 2001). Fig. 1. shows an editor for the open source game *Yo Frankie!* This game uses the engine by the Blender Institute, which provides the functionality to apply AI techniques to game engine design through an interactive editor (Lioret, 2008).

There are numerous other computer games that use classical AI techniques, including Star Craft, Unreal Tournament 3, and FEAR. In general, these games use agents as tactical enemies or support characters, such as partners that interact with other agents, humans, and the environment. Since these games execute in real-time, all of the agents must have extremely fast response times. AI techniques used in games include common-sense reasoning, speech processing, plan recognition, spatial and temporal reasoning, high-level perception, counter planning, teamwork, path planning, as well as other techniques (Larid & Lent, 2000). One specific example is FEAR and its use of Goal Oriented Action Planning or GOAP (Orkin, 2004; Orkin, 2005), which is a form of STRIPS. FEAR uses GOAP to create complex behaviors while relying on classical AI techniques. This approach allows for decoupling goals and actions, layering behaviors, and dynamic problem solving. Classical symbolic AI techniques have been used with varying degrees of success, but many challenges have risen as a result of increased demand on the game agents by human opponents and increasingly complex game environments, even when games, such as FEAR, leveraged classical AI techniques to a great extent.



Source: Evolutionary Computation, Book edited by: Wellington Pinheiro dos Santos,
ISBN 978-953-307-008-7, pp. 572, October 2009, I-Tech, Vienna, Austria

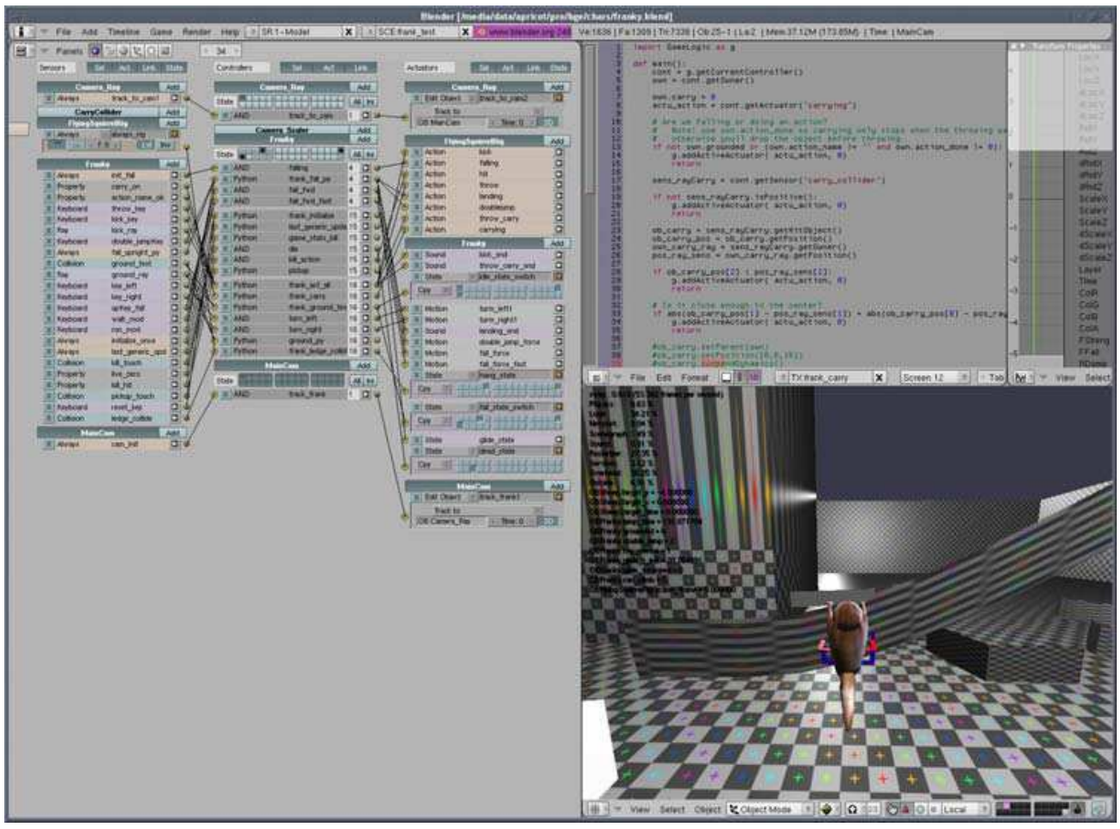

Fig. 1. Blender game engine editor from the open source game *Yo Frankie!*

Attempting to determine *a priori* every game state that the agent will face is a daunting task. Even for relatively simple games, over 20,000 possible states exist (Schaefer, 2002), which limits the applicability of some techniques. Classical AI techniques can become very complex to create, maintain, and scale as the possible game states become more complex (Larid & Lent, 2000). In many cases, since it is not feasible to plan for every event in a game, the agents have a very limited perception of the game. This limited perception of the game world creates two key problems. First, as the game environment or the human player's tactics change over time, the agents have difficulty adapting to the new environment or tactics. Without the ability to form new tactics to respond to major changes in the player's tactics or environment, the agent's performance will significantly degrade over time. Second, as the human players gain experience with the game, they can begin to predict the actions of the agents. Thus, the humans' knowledge about the game continues to improve through learning by playing, but the game agents do not learn through their experience of playing the game. These two shortcomings are common with the application of many of the classical symbolic AI techniques in game design today, which results in a game that loses challenge over time.

Neural networks have the ability to overcome some of the shortcomings associated with the application of many of the classical AI techniques in computer game agent design (Haykin, 1999). Neural networks have many advantages, including being self adaptive in that they adapt well to computer game environments that change in real-time. Neural networks can improve performance via off-line and on-line training. In other words, the game agents can

be trained once before being deployed in a game or the learning algorithm can be applied in real-time for continuous improvement while the game is played. Also, for neural-based agents, the corresponding source code tends to be small and generic allowing for code reuse through libraries since the essence of most neural networks consist of a series of inputs sent across an analog weight matrix that capture states and world environments to determine their outputs. Such designs allow easy computation and incorporation of data processing techniques, such as multi-threading and parallel processing, which is a requirement for today's high-performance games. With these benefits, neural-based agents gain the capability of adapting to changing tactics by humans or other game agents and may acquire the ability to learn and generate new tactics while playing the computer game, similar to the capability of many human players. Since the neural-based agents are adapting to the game conditions and generating new tactics, the game remains challenging for a longer time than games that use only classical AI techniques.

Incorporating neural networks in game designs also has problems, such as the difficulty in obtaining training data and unexpected emergent behavior in which the computer game does not function as intended by the designer. In general, obtaining training data is very critical to neural network development. In the computer game domain, there are two common approaches that can be used. First, data can be recorded as a human plays in the role of the game agent and this data can then be used to train the neural network. Another approach is to use an evolutionary process, such as genetic algorithms, to train the neural network by seeding the network and allowing for mutations and a cost function that terminates underperforming neural networks (Miikkulainen et al., 2006). Unexpected emergent behavior can be corrected by having a performance function or a teacher that evaluates the actions and corrects the agent. If the human player begins to lose consistently, the performance function can be adjusted so that the game agent performs at the appropriate level corresponding to the human player.

The remainder of this chapter will focus on four main topics. First, the background of various methods to apply neural networks in computer games will be explained along with several examples for commercial and academic computer games. Second, a strategy will be presented that will facilitate more efficient development of neural networks in computer games. Third, the complete development of a neural network for the Defend and Gather game will be described along with the network topologies used for the neural-based agents and the evaluation process used to analyze the overall performance of the agents. Although aspects of neural networks in the Defend and Gather game were previously described by Qualls et al., 2007, that work-in-progress conference paper does not contain the context and detail of this chapter. Finally, the future for neural networks in computer games will be explored along with recommendations for subsequent work.

## 2. Neural gaming background

### 2.1 Example game applications

Neural networks can be used in a variety of different ways in computer games. They can be used to control one game agent, several game agents, or several neural networks can be used to control a single game agent. A game agent can be a non-player character or it can be used to represent the game environment. With this in mind, a neural network can be used to control and represent just about any facet of a computer game. This section will discuss

several areas in which neural networks can be applied to computer games along with several examples of neural networks being used in computer games.

Path navigation is one of the most common uses of neural networks. A neural-based agent can adapt to an ever changing environment and more importantly, learn the human player's path to make the game more challenging over time. Neural networks can also be used to control ecology for the game. This neural-based ecology may be used to just populate an environment, or it could be used to control mutations based on a player's action, such as making animals more friendly or scared of the player. Animation is another promising area for neural networks. As computer games become more powerful, game designers demand more realistic animations. As the number of complex objects increase in games, attempting to create animations for every scenario can be impossible. For example, a neural network could be used to teach a neural-based agent some task, such as teaching a dog to walk. The neural-based agent could then learn to adapt to walking over a rocky environment or over an icy lake. Finally, advanced reasoning can be used by neural networks for dialog choices and strategies for defeating humans and other agent players, as well as other tasks. One example is two neural-based game agents may need to learn to work together to complete a game, which is the focus of Section 4 of this chapter.

### 2.2 Example games in academic and commercial markets

Neural networks have been around for quite some time but it has only been since the 1990s that there have been attempts at integrating neural networks within computer games. One of the first available commercial games that used neural networks was Creatures. The agents in Creatures used neural networks for learning and sensory motor control in conjunction with artificial biochemistries. Together, the neural network and artificial biochemistries are genetically specified to allow for evolution through reproduction. The neural networks are made up of 1000 neurons grouped into nine lobes interconnected with 5000 synapses. The neural-based agents learn by a reinforcement signal from the human player. The human player provides feedback by stroking or slapping the agent for positive or negative learning (Grand & Cliff, 1998).

In 2001, CodeMasters created the Colin McRae Rally 2.0 (CMR) racing game for the Sony PlayStation One. The CMR game used neural-based agents to control the opponent race cars as they drove around the track. The neural-based agents learned to drive all of the tracks in the game while learning how to maneuver around other race cars on the tracks. To obtain data to train the neural networks, the developers played the game and recorded their laps around the race tracks. This recorded data consisted of drivers' reactions to other cars and the average driving line around the track. The neural networks were then trained off-line with the recorded data and the resulting neural network was integrated within the agents to control the opponent race cars (Mathews, 2000).

Other games included Black and White and Democracy 2. Black and White Versions 1 and 2 used neural networks with reinforcement learning to allow a creature (neural-based agent) to learn from a player's actions. For instance, the creature may need to learn that it needs to destroy a structure before it sends in another army to destroy the opponent. Democracy 2, which is a political simulation game, uses a highly complex neural network that simulates desires, loyalties, and motivations of game agents that make up countries on a planet (Barnes, 2002).

There are many different examples of neural networks being used in computer games in the academic domain. The NERO (Neuro-Evolving Robotic Operatives) and GAR (Galactic

Arms Race) games, which use the NEAT (Neuroevolution of Augmenting topologies) algorithm, along with NeuroInvaders and Agogino's Game are discussed in this chapter. All of these example games use genetic algorithms or some form of evolution to train their neural networks (Agogino et al., 2000; Briggs, 2004; Stanley, 2005a; Stanley, 2005b; Stanley 2006; Hastings, 2007).

First, Agogino's Game was based on Warcraft II by Blizzard Entertainment. This game contains a human-controlled player and neural-based peons. The objective of the game is for the peons to find a resource point without being killed by the human player. The peons use genetic algorithms to evolve feed-forward networks in real-time. As time progresses, the peons become more adapt at finding resource points and avoiding the human player. The peons accomplish this task by evaluating the risk of choosing between a resource point that is near a human player or a resource point that is farther away, but unguarded (Agogino et al., 2000).

Second, the game NeuroInvaders allows human opponents and neural-based agents to play against each other in a death match. The agents must seek out and shoot the human player to win and vice versa. The agents use a spiking neural network containing eight neurons. For example, if the agent's FIRE neuron activates then the agent will fire its laser or if the RIGHT neuron fires then the agent will turn right, etc. The neural network of each game agent starts with random weights and delays. When the agent dies, a mutation created by adding random Gaussians to the weights and delays is added to the original weight matrix of the dead neural-based agent and integrated within a new neural-based agent that is still alive. All of these mutations occur in real-time using genetic algorithms while the game is being played (Briggs, 2004).

The NERO and GAR games both use the rtNEAT (real-time NEAT) algorithm and the cgNEAT (content generation NEAT) algorithm developed at the University of Texas. The NEAT algorithm is similar to other evolving neural networks but it has a key difference in that it can modify and increase the neural network's complexities along with its weights. The rtNEAT game is a real-time version of NEAT that gains a computation performance boost by putting a time limit on the life span of a neural-based agent and then evaluating its performance. If the neural-based agent is performing well then it is kept in the game. If its performance does not meet certain criteria, it is then discarded and it starts over. The cgNEAT algorithm is based on the idea that neural networks can generate more interesting content for the player, which results in an overall more enjoyable player experience (Stanley, 2005a; Stanley, 2005b; Stanley, 2006; Hastings, 2007).

The game NERO is essentially a real-time strategy game in which an agent must navigate an environment and defend itself against other game agents with the same goal. In NERO there are two phases of play for the human player. In the first phase, the human player acts as a teacher and creates exercises for the neural-based agents to go through and then the player can dictate fitness-based performance parameters by examining how well the agents move around walls or hit targets. In the second phase, the player can place its trained neural-based agents against other players' trained agents to determine which agent performs the best (Stanley, 2005a).

A second game based on NEAT called GAR is very different from NERO. The NERO game has the player actively involved in the neural network development, while GAR's neural network runs primarily behind the scenes. The GAR game uses cgNEAT to generate new types of weapons for the player to use in the game. These new weapons are generated by creating variants of weapons that a player is using within the game. In short, the neural

network in this case is generating interactive content for the player to use based on the player's preferences (Hastings, 2007). This last example shows that neural networks can be used in very interesting ways that may not necessarily fit the stereotypical functions of a game agent.

## 3. Strategies and techniques for neural networks in games

The most significant barrier to using neural networks in computer game design lies not with understanding how neural networks function but how to apply one of the various neural network techniques to a specific computer game. As seen from the previous examples, there are many different methods to develop and use neural networks in computer games. Decisions such as off-line/on-line training, neural network architecture, training algorithms, and most importantly, data acquisition, are all components of the neural network development process. This section of the chapter will discuss a simple process for incorporating a neural network inside a computer game and then follow up in Section 4 with the Defend and Gather game that uses the outlined development process in its implementation.

The first decision is to decide the scenarios in which the neural network will be used in the game. Will the neural network control a game agent, animation routine, or other function? The second decision revolves around training data. In other words, how and what data will be collected to train the neural network? This leads to questions concerning what constitutes data for the game system. Generally, the data is determined by the type of data inputted into the neural network and outputted by the neural network. The third decision that needs to be made is what type of neural network architecture is most appropriate based on the intended function of the agent. The neural network architecture can vary greatly depending on its given tasks. The fourth decision revolves around deciding between off-line/on-line training and the types of learning algorithms to be deployed. The main question is will the neural network continue to learn while the game is being played or will the neural network be trained and placed within the computer game with no further learning during game execution? There are advantages and disadvantages to both approaches. Allowing the neural network to continue to learn during game execution can provide it further refinement in playing the game but could lead to unforeseen actions that may result in the game not performing as desired. Even without further learning, anomalous game behavior can still be an issue. All of these important decisions are highly interdependent and are determined by the type of computer game being developed and the intended function of the agent.

As previously discussed, neural networks can be used to control various aspects of a computer game. For developing our simple process, we first create a hypothetical computer game in which an agent must navigate an environment to exit a simple maze. The game agent will have a sight range to allow for detection of walls and the goal of finding the exit. If the game agent gets close to a wall, it should move away from it. Fig. 2. shows a typical maze game.

For our first decision, the neural network will control all of the actions of the game agent. The neural network will determine how the agent moves around the environment. Therefore, the inputs to the agent are the directions of the walls from the perspective of the sight range of the agent and the outputs will be the direction the agent should follow. In short, we will have four inputs representing wall directions consisting of UP, DOWN, LEFT,

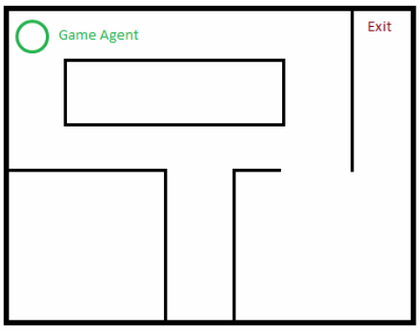


Fig. 2. Example maze game with the agent navigating the environment

and RIGHT and the outputs will be the direction the game agent travels: UP, DOWN, LEFT, and RIGHT. For the second decision, we will collect data by playing the game and recording the directions of the walls within the sight range of the game agent and the resulting direction we move the game agent. The sampling rate for data collection of the inputs and outputs was set as two samples per second. For most action games on the market today this rate will need to be increased due to the volume of information that a player may be exposed to within one second of game play. Table 1. shows an example recording of the game agent for two seconds.

| Maze Game Recording Output File | | | | | | | |
|---|---|---|---|---|---|---|---|
| Inputs: Wall Directions | | | | Outputs: Game Agent Direction | | | |
| UP | DOWN | LEFT | RIGHT | UP | DOWN | LEFT | RIGHT |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

Table 1. Agent recording for two seconds as human player moves around the maze

The third decision requires the specification of the neural network architecture. There are many types of architectures to choose from, such as recurrent, feed forward, and many more. If we look at the inputs and outputs of the game agent in this case, there are four inputs and four outputs, so we chose a feed-forward network, which is also one of the simplest architectures to implement. We also specify the number of hidden layers, as well as the number of nodes in each layer. A general rule of thumb is the number of nodes in the hidden layer should total to at least the sum of the output and input nodes to be adaptive to changing environments. Another rule of thumb is that a feed-forward network should have at least one and half times the number of input or output nodes in each hidden layer (Garzon, 2002). For the simple game in this case, there are four inputs and four outputs so there are two hidden layers with six nodes each. Fig. 3. shows the neural network architecture for our agent in the simple maze game.

The last decision involves deciding between training the neural network off-line then placing into the game or to allow the neural network to continue to learn while the game is being played. Also, the type of training algorithm must be specified. For this simple game, the neural network will be trained once off-line with back propagation. Larger and more complex games may need on-line training because the game itself may change over time, which is typical for multi-player on-line games. By answering these questions during the

development of a computer game, the process for adding a neural network to a computer game can become straightforward. The following section of the chapter will go through a more complex example detailing how a neural network was added to the Defend and Gather game.

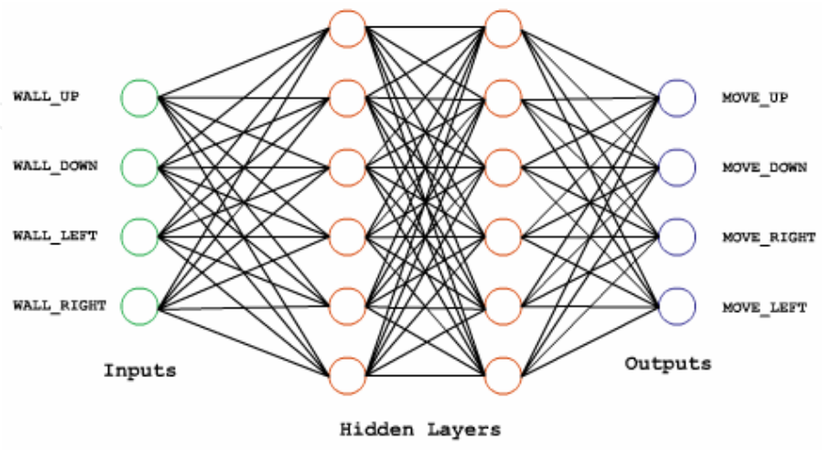


Fig. 3. Neural network architecture of the agent for the simple maze game

## 4. Example game with neural networks

### 4.1 Defend and gather

The Defend and Gather game will provide a better understanding of the benefits of neural networks in game design. In this game, neural-based agents play against classical symbolic AI agents in a contest to determine which agent will win the game. Defend and Gather was influenced by many of the games discussed in Section 2 of this chapter. One key difference is that instead of the neural-based agents facing off against human opponents they play against other game agents implemented with multiple classical AI techniques. The game agents consisting of classical AI techniques used techniques such as complex branching and conditional statements. These game agents will be referred to as BaC throughout the remainder of this chapter. Since the BaC agents use fairly simple AI techniques, it was decided that the neural-based agents would use off-line training techniques. Using off-line training also allows for better assessment of the neural-based agent's ability to cope with increasing difficulty with no additional learning while the game is being played. To facilitate the development of Defend and Gather an engine developed by Bruno Teixeira de Sousa was used (De Sousa, 2002). This game engine is a simple but robust two-dimensional open source engine that handles graphics, input/output, and simple physics computations. By choosing to use this game engine it allows other developers who may be interested in the implementation of neural-based agents to have a common platform to understand the process of developing their own neural-based agents for their respective game.

In Defend and Gather, the game agents have conflicting goals for winning the game. This was implemented to ensure that the neural-based agents will have confrontations with the BaC agents. The BaC agents have two goals to follow. First, they need to defend resource points and second they need to hunt and destroy the neural-based agents. There are two different neural-based agents in the game, each with a different goal. The first neural-based agent (Resource Collector) searches for and collects resource points while avoiding the BaC

agents. The second neural-based agent (Protector) actively protects the other neural-based agents and searches for and destroys the BaC agents. An additional constraint is that the protector agent cannot destroy the BaC agents unless the resource collector agents have collected resources. This implies that the neural-based agents must work together to survive and win the game. Defend and Gather ends when one of following three conditions is met: 1) all BaC agents are killed; 2) all neural-based agents are killed; or 3) all the energy is collected and exhausted by the neural-based agents. If conditions 1 or 3 are completed then the neural-based agents win the game, or if condition 2 is completed then the BaC agents win the game. Fig. 4. shows a sample screen shot of Defend and Gather showing several environmental components, including resource points (Triangles), resource collector (Smiley Face), BaC agent patrol (Orange Plus), and several walls (Squares), all of which will be explained in further detail.

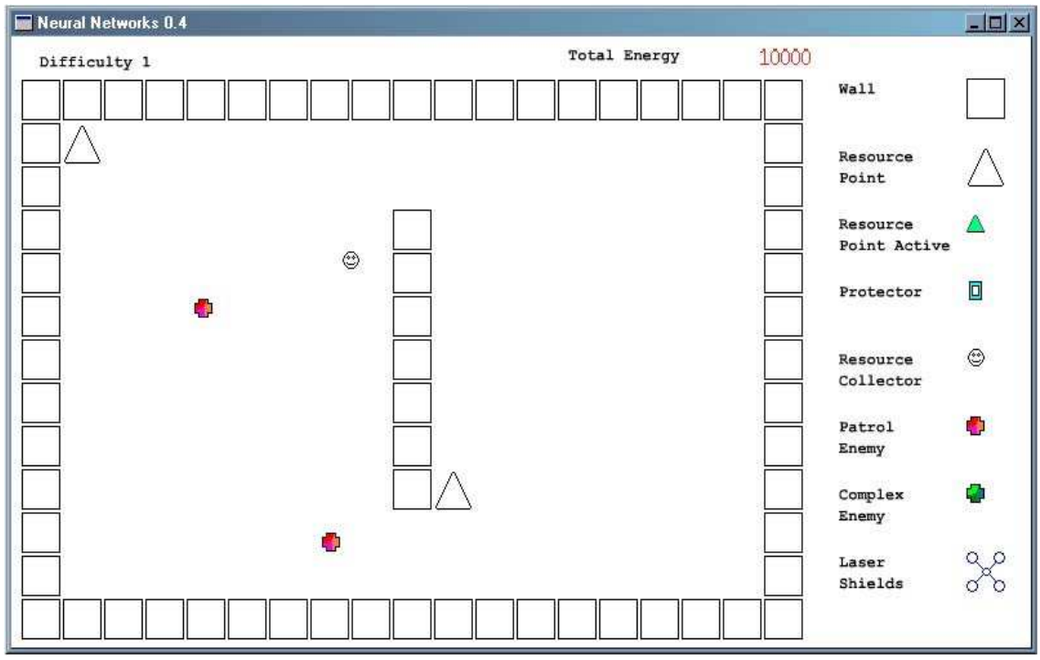


Fig. 4. Level 1 of Defend and Gather

There are four different agents in the game. The agents include two BaC agents called the patroller and the hunter and two neural-based agents called the protector and resource collector. All of these agents navigate the game environment in real-time. Mass-force vector calculations are used for movement. In other words, the agents control their thrust in four different directions for moving and slowing down. There are no limits on how fast any of the agents can move, so the agents will have to cope with not losing control. Also, if the agents run into a wall, then they will bounce off the wall based on the speed and trajectory they were traveling. So, if the agent moves too fast and hits a wall it can have the effect of bouncing around like a pin ball until control is regained. Each of the agents has a limited visibility range or field of view around them. The visibility range of each agent is a circle about three times their respective size.

The behaviors of each of the agents are dictated by their respective goals. First, the patrolling agent is a fairly simple BaC agent. It moves back and forth between two set points in the environment. If the patrolling agent detects a neural-based agent in its visibility range

it will accelerate and start to fire its laser shields at the neural-based agent. Once the neural-based agent is destroyed or out of its visibility range the patrolling agent will return to its normal patrolling route and stop firing its shields. The hunter agent is a more complex BaC agent. The hunter agent moves randomly across the environment in search of the neural-based agents. If the hunter agent detects a neural-based agent in its visibility range it will start to pursue the neural based-agent by increasing its speed while also firing its laser shield. The hunter agent will continue to pursue the neural-based agent until it is destroyed or it no longer detects the neural-based agent in its visibility range. Once a neural-based agent is no longer detected, it will stop firing its laser shield and return to randomly searching for another neural-based agent. The resource collector neural-based agent simply searches for resource points while avoiding the BaC agents. The resource collectors have no laser shield for protection. The only advantage they have is that they can accelerate slightly faster than all the other agents. Once the resource collector has discovered a resource point it will hover at that location until all the resources have been collected. If a BaC agent is detected in its visibility range the resource collector will flee and try to move toward the protector agent. Once the resource collector no longer detects the BaC agent it will return to collecting resources so that the protector agent will have energy to fire its laser shield. The protector actively searches for both types of BaC agents and tries to protect the resource collector agents. The protector agent tries to stay close to the resource collectors while also searching for the BaC agents. Once a BaC agent is detected in the protector's visibility range it will start to pursue the BaC agents by accelerating if it has energy to fire its laser shield. If the protector does not have energy to fire its laser shield it will not give chase. The protector agent will continue to pursue a BaC agent until it is destroyed, no longer detected in its visibility range, has energy for its laser shields, and the resource collectors are not being attacked. If a resource collector is being attacked the protector will start to move towards that resource collector.

The Defend and Gather game consists of a closed two-dimensional environment as seen in Fig. 4. The environment contains the game agents previously mentioned along with resource points and walls for the agents to navigate around. To test the viability of all the agents, four different environments were created that increase in difficulty and pose more difficult challenges for the game agents. Fig. 5. through Fig. 8. show the four levels increasing from easy to most difficult. All levels contain one protector neural-base agent and four resource collector neural-based agents. The amount of resource points and the number and types of BaC agents varies across the levels. Fig. 5. shows the easiest level in the game. This level consists of two resource points and a single wall to navigate around. The level also contains just two patrolling BaC agents, one protector and four resource collector neural-based agents. Fig. 6. shows the next level of the game. This level consists of two resource points, two patrolling BaC agents, and one hunter BaC agent. The resource points are located in two small rooms with one of the rooms having both entrances protected by patrolling BaC agents moving in opposite directions. Fig. 7. shows the third level of the game. There are three resource points in this level each of which is in a separate room. Two of the resource points are heavily guarded while the third is not guarded. The level also contains one hunter BaC agent and three patrolling BaC agents. The concept behind this level is to see if the resource collectors will go for the unguarded resource point first and then allow the protector to try to eliminate some of the BaC agents before attempting to

collect the other resource points. Fig. 8. shows the final level of the game, which is the most difficult level. This level has four resource points in separate rooms along with six patrolling BaC agents and two hunter BaC agents. All of the resource points are heavily guarded. The room itself is divided in half by a hallway that is designed as a chokepoint, which is common in many of today's games. The idea of a chokepoint is to force confrontation between players in the game.

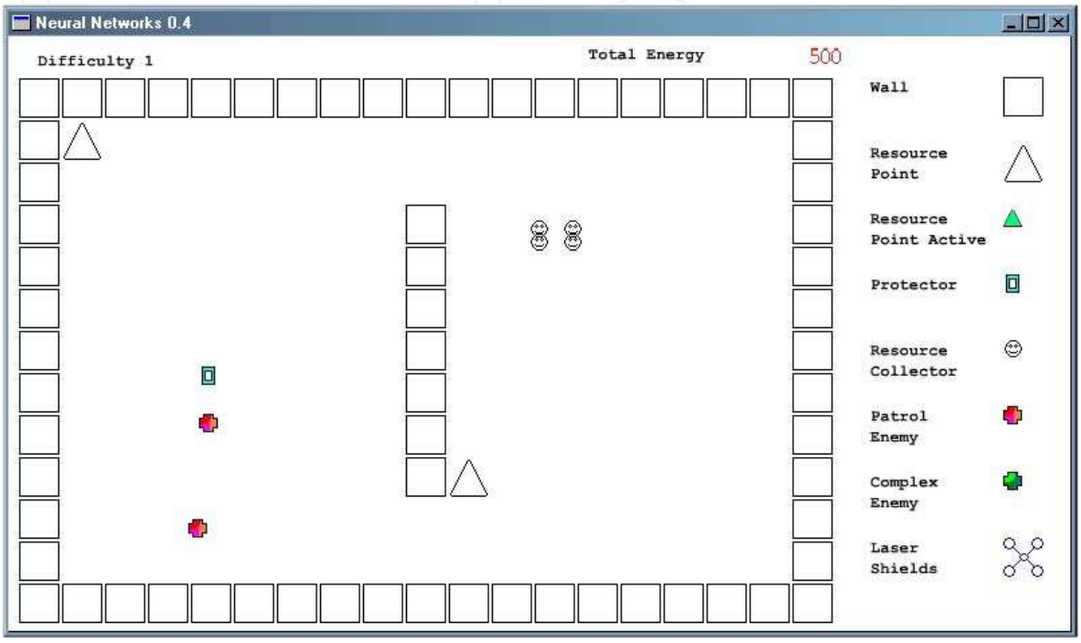


Fig. 5. Difficulty 1—Two resource points and a single wall to navigate around

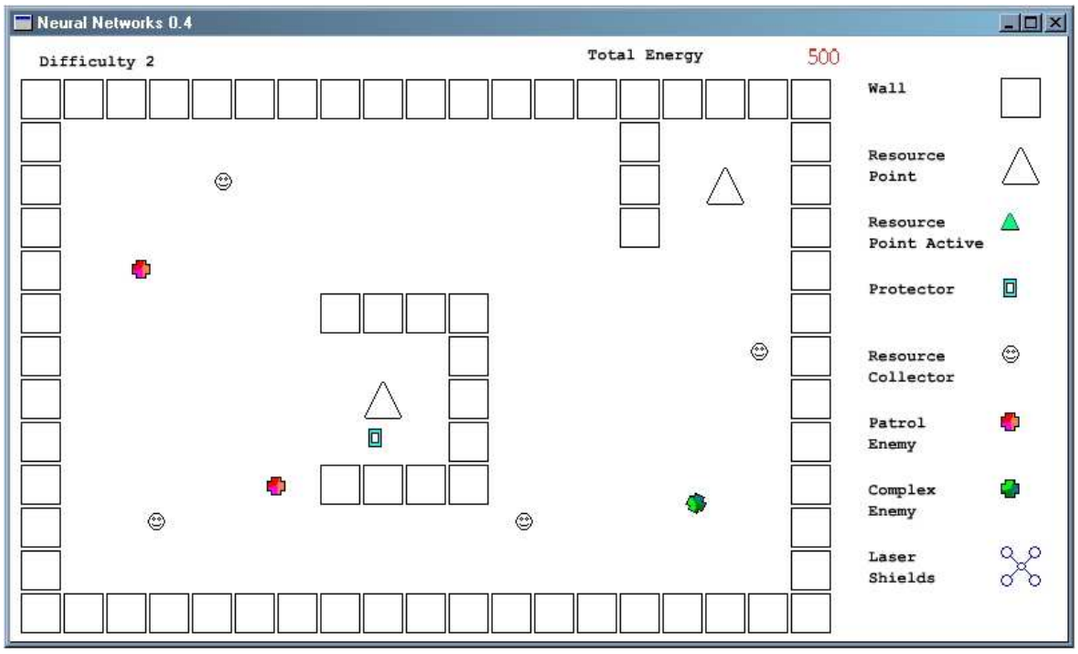


Fig. 6. Difficulty 2 —Two resource points each in a separate room

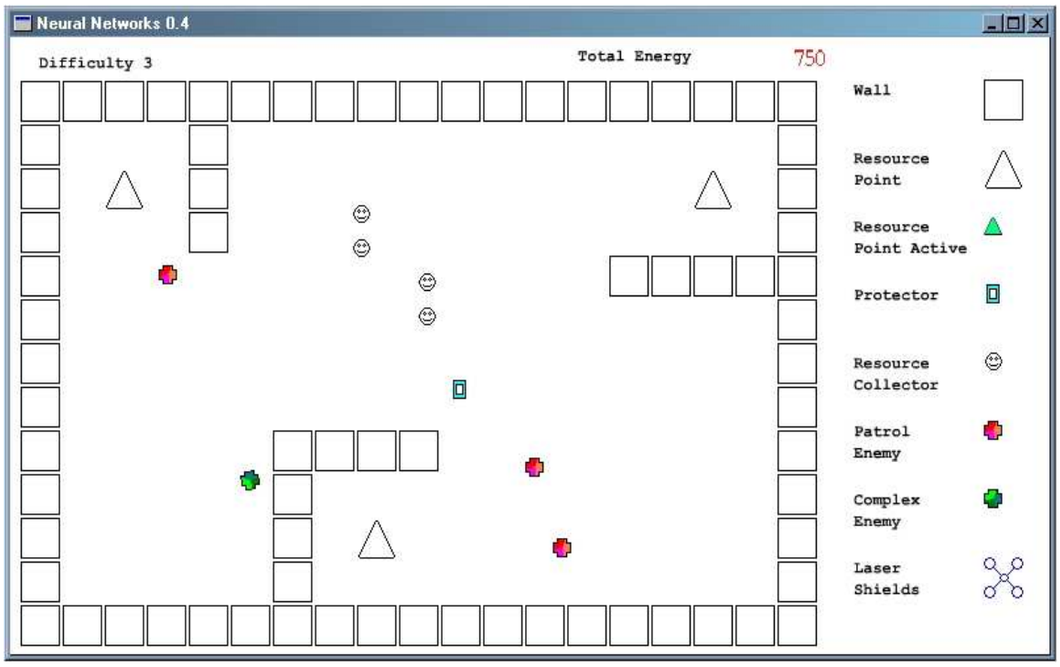


Fig. 7. Difficulty 3—Three resource points each in a separate room

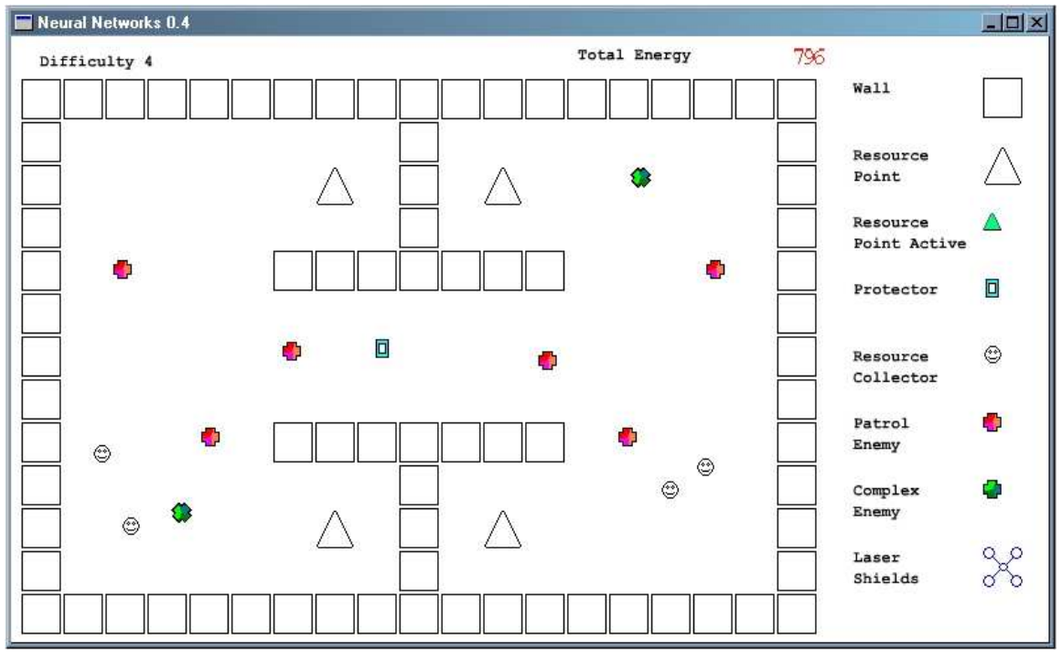


Fig. 8. Difficulty 4—Four resource points each in separate room with a choke point in the center of the level

### 4.2 Neural network development process

Now that the game world has been defined, we must determine how the neural network will operate in Defend and Gather. We have already defined the resource collectors and the protectors, which are the two types of agents that will use the neural networks in their

design. The neural-based agents must able to move around the environment and react to the surroundings within their visibility range. Therefore, the inputs are the surroundings in its visible range and the outputs are the resulting movement. That is, the inputs of the neural network are objects in the game while the outputs control the movement of the neural-based agents. The outputs are the four directions UP, DOWN, LEFT and RIGHT. These outputs inform the agents which direction to fire their thrusters. The agents are not limited to firing their thrusters in one direction at the same speed as they can fire in several directions and at different speeds, which increases their maneuverability. The inputs of the neural network consist of directions of other game objects in different directions. The neural network has eight inputs, with the top four inputs representing the directions of walls and the bottom four inputs representing the direction of BaC agents.

### 4.3 Network architecture

Since each neural-based agent is a separate entity within Defend and Gather, two neural network architectures were created. Since the BaC agents use relatively simple AI techniques, the neural-based agents were designed using feed-forward networks, which is the simplest type of neural network architecture. Feed-forward networks are also straightforward to implement in gaming software. Many other topologies could have been implemented; however, it was decided to implement the simplest design first, then, try more complex networks.

One of the more difficult aspects of implementing the network architecture is determining the number of hidden nodes in the network. As stated in Section 3, we used the convention of choosing the number of hidden nodes to be approximately 1.5 to 2.0 times the number of nodes in the input/output layer. This design criterion was used for both the protector and the resource collector neural-based agents. Since the number of inputs and outputs are different, eight and four respectively, we used two hidden layers. The design criterion resulted in a network architecture that consisted of eight input units, four output units, and two hidden layers with fifteen and six nodes, respectively. Fig. 9. shows the final neural network architecture for the resource collector along with several other details. This architecture is also the same for the protector neural-based agent.

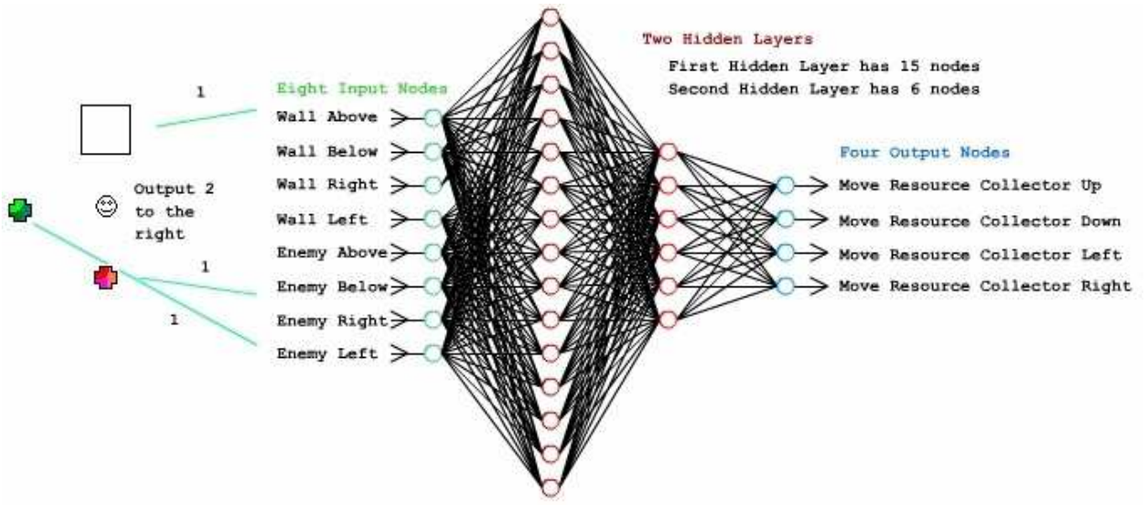


Fig. 9. Architecture of the resource collector with labeled inputs and outputs

As stated in Section 4.2, the eight inputs consist of two groups. The first four inputs in the network are used for detecting the direction of a wall and the last four inputs are used for detecting the BaC agent's direction. The four inputs for the walls and enemies operate in the same manner. These inputs remain at zero if no agent or wall is in the visible range of the neural-based agent. If detection occurs of a wall or enemy, then, the input will change to a one in the corresponding direction. For example, if there is a wall above the resource collector or protector, then, a one will be sent to the Wall_Above input in the neural network, Fig. 9.

Both the resource collector and the protector neural-based agents have the same outputs. The output determines the direction and speed of the agent with values zero, one, or two. For the resource collector, if all of the outputs are zero, then there is no detection of a wall or BaC agent in its field of view; therefore, the resource collector continues to search for resource points in the game environment. If any of the outputs are set to value one, then a wall has been detected and the resource collector will move in that direction at the same speed. If the value of any of the outputs is set to two, then a BaC agent has been detected and the resource collector will accelerate and move away from the BaC agent. Following Fig. 9., if the input value Enemy_Above for a resource collector is set to one, meaning a BaC agent has been detected above the resource collector, then the output value of Move_Down will be set to two resulting in the resource collector accelerating away from the BaC agent. If the input value Wall_Right is set to one then the output value Move_Left will be set to one to move the resource collector away from the wall. This example may appear simple but by supporting the ability for processing different multiple inputs, the resource collector can detect an enemy in different directions and detect walls. The resource collector can move in very complex directions, including arching curves. This capability is the same for the protector neural-based agent except for how it reacts to a BaC agent. If the protector detects a BaC agent, then it will accelerate toward it. For example, if the input value of Enemy_Up is set to one, then output value Move_Up will be set to two to accelerate the protector toward the BaC agent.

### 4.4 Training the neural networks

To obtain training data for the neural-based agents used in Defend and Gather, a similar approach that was taken in the PlayStation game Colin McRae 2.0 mentioned in Section 2 was implemented. In the game Colin McRae 2.0, human players were used to play as the opponent race car and their actions were recorded on given game states as exemplars in the training data. Based on this same approach, Defend and Gather also recorded data from humans playing the game. Training data was recorded by having humans play as the resource collector and the protector on the level two of the game Defend and Gather. Level two was chosen for play recording because both types of BaC agents, that is, hunters and patrollers, were present in this level. Essentially the inputs and outputs of the neural-based agents were recorded as the humans played the game and then the data was extracted to a file. Data was recorded at five times a second. For example, if a BaC agent or a wall was detected within the field of view of the neural-based agent while playing the game against a human, then, both the inputs of the human and agent would be recorded and logged in the output file. Table 2. shows a sample output as a human player moves down and to the left to move away from a wall to the right. The BaC agent is above the player.

| Defend and Gather Sample Recording File | | | | | | | |
|---|---|---|---|---|---|---|---|
| Inputs: Detection of Walls and BaC agents within visibility range in game environment | | | | | | | |
| Wall Direction | | | | BaC Direction | | | |
| ABOVE | BELOW | RIGHT | LEFT | ABOVE | BELOW | RIGHT | LEFT |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| | | | | | | | |
| Outputs: Player Movement Direction | | | | | | | |
| Acceleration Direction | | | | | | | |
| ABOVE | | BELOW | | RIGHT | | LEFT | |
| 0 | | 2 | | 1 | | 0 | |

Table 2. Sample recording of player playing as the resource collector

To obtain enough training data, the humans played level two for several rounds, with each round consisting of ten games. Data was recorded for both wins and losses. Losses were included in the training set with the objective of making the neural-based agents behave in a more human-like manner and to allow them to adapt to a changing game environment. One problem with the recorded data was the large numbers of zeros in both the input and output. This was a result of the absence of a sensed percept that resulted in an action, that is, 'nothing' being detected and the player not taking any action. These large areas of zeros were parsed out of the data. After the parsing, the recorded data was then split into two parts. The first part, which consisted of seventy percent of the data, was used for training the neural networks. The remaining thirty percent was used to test the neural networks to see how well the neural-based agents learned from the training data.

There are many different ways to train a neural network but to speed the process MATLAB was used to train the two neural networks. Werbos' back-propagation algorithm was chosen to train the two neural networks (Haykin, 1999). Back-propagation was selected to train the neural networks because it is considered one of the hallmarks of training algorithms for traditional neural networks. After training the neural networks with MATLAB, the mean squared error was obtained for training the resource collectors and the protectors (7e-7 and 2e-9) as shown in Fig. 10. and Fig. 11., respectively. The two neural networks yielded an error rate of less than three percent over a training set size of 4000 separate entries. The protector converged much faster than the resource collector. This may be due to the fact that as the humans played the game in the role of the protectors, they acted more aggressively in chasing down the BaC agents, whereas in the role of the resource collectors, the human players had to act far more cautiously to avoid the BaC agents to find the resource points.

### 4.5 Evaluation

Evaluating the neural networks centered on the agents' abilities to play Defend and Gather well enough to cope with increasingly difficult game environments and more complex BaC agents. To determine how well the neural-based agents play Defend and Gather twenty games were completed on each of the four difficulty levels. The number of wins and losses were recorded for each of the games. The number of wins and losses for the neural-based agents are shown in Fig. 12. The last column in Fig. 12. shows the total number of wins over all four difficulty levels. Further analysis of the neural-based agents focused on how well the agents interacted with the game environments. These interactions included navigation around walls to find resource points and how well the agents found or avoided the BaC agents in the levels.

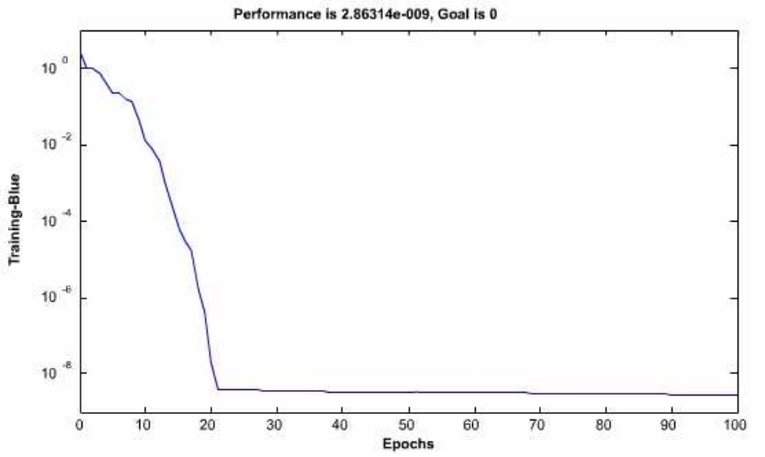


Fig. 10. Neural networks for the protector being trained in MATLAB

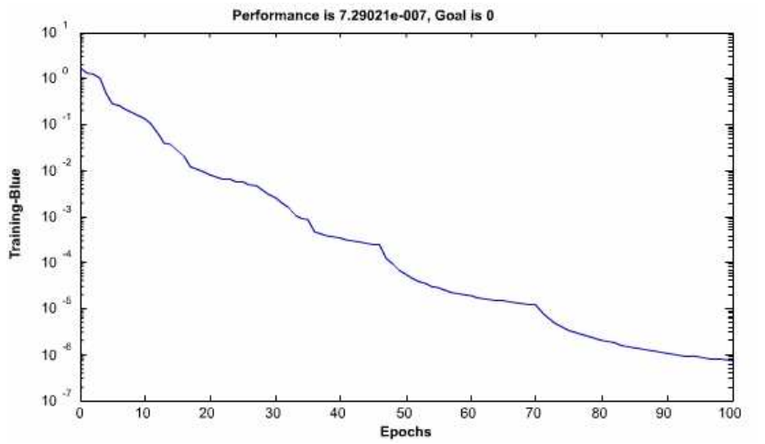


Fig. 11. Neural networks for the resource collector being trained in MATLAB

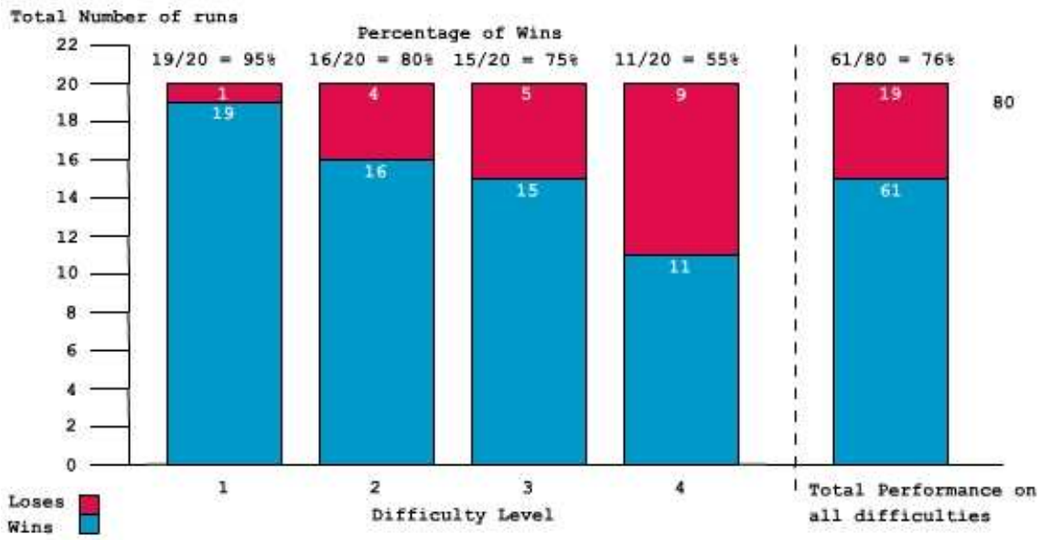


Fig. 12. Neural-based agents won 76% of the time over 80 plays of Defend and Gather

To ground the performance of the neural-based agents, a comparison is needed. While the humans played the game their resulting wins and losses were also recorded. Human players only won level two sixty percent of the time during the data recording. Human players were also asked to play other levels in the game and they won ninety five percent of the time on level one, thirty percent of the time for level three, and less than three percent for level four. The neural-based agents' performance was regarded as a success if they won more than seventy percent of the time for the total game. This benchmark was selected to ensure the agents performed better than the humans. The neural-based agents won seventy-five percent of all the games across the difficulties, thus exceeding our goal of seventy percent as shown in Fig. 12.

Next, observations were made of the neural-based agents' interactions with the game environments. In general, the neural-based agents were able to navigate the levels without too many problems. One problem scenario that consistently occurred was when a wall was placed directly between a resource point and the neural-based agent. The neural-based agents would sometimes stop at the wall and never attempt to navigate around the wall to the resource point. This scenario only happened about ten percent of the time, but it did contribute to losses incurred by the neural-based agents. The resource collectors did an excellent job of navigating around the BaC agents to find the resource points. The deaths of the resource collectors were generally the result of the agents accelerating too fast in one direction and running into the BaC agents because they could not change direction fast enough to avoid them. Other kills by the BaC agents involved trapping the resource collectors in a corner until they were destroyed. The protector also did an excellent job of hunting down the BaC agents to destroy them. Over half of the wins of the neural-based agent occurred because the protector was able to destroy all of the BaC agents in the game.

The neural-based agents were extremely effective at playing the game Defend and Gather. Their effectiveness of navigating the environment and interacting with the BaC agents led to winning over seventy percent of the games played. Even with the increasing difficulty, the neural-based agents were still able to win the game. On the most difficult level the neural-based agents were able to win half of the games, whereas the human players could only win three percent of the time.

### 4.6 Future work

As seen in Section 4.5, the neural-based agents designed for Defend and Gather learned to play the game quite effectively and they were able to win approximately seventy-six percent of the time. There is still opportunity for improvement in the capabilities of the neural-based agents. Various techniques could be applied to the agents to increase their performance. First, Defend and Gather used off-line learning so as the neural-based agents play the game they do not gain from their experience. The next step would be to include some form of on-line learning so that the neural-based agents can continue to learn while playing the game. Neural-based agents using on-line learning would be able to formulate better strategies during the game in real-time and over time rather than learning only once. This continuous learning may be particularly useful with dynamic obstacles and when other agents playing in the game are learning improving their performance over time through learning.

Only back-propagation was used for training in Defend and Gather. There are many other training techniques that could have also been used to increase performance. These various algorithms include genetic algorithms, dealing with mutations over time, simulated

annealing, and various other methods. Different network architectures other than feed-forward networks could be implemented for future work. Architectures, such as recurrent networks, which facilitate bi-directional data flow, a self-organizing map, to remember the location of certain enemies and the topology of the game environment, stochastic networks, which introduce randomness, and finally a modular network made up of many kinds of neural networks have applicability in game design (Haykin, 1999). From these improvements alone, there are many possibilities that could be applied to Defend and Gather to improve performance of the neural-based agents.

## 5. Conclusions

It is clear from the implementation and analysis of the performance of the game Defend and Gather and the many other examples discussed in this chapter that neural-based agents have the ability to overcome some of the shortcomings associated with implementing classical AI techniques in computer game design. Neural networks can be used in many diverse ways in computer games ranging from agent control, environmental evolution, to content generation. As outlined in Section 3 of this chapter, by following the neural network development process, adding a neural network to a computer game can be a very rewarding process. Neural networks have proven themselves viable for agent design, but there are still many unexplored avenues that could prove to benefit from neural networks in computer games. The area of content generation has only briefly been discussed in recent research. The potential is that neural networks could generate entire worlds or even entire computer games based on human players' preferences. Neural networks have great potential for designing computer games and technology that will entertain players in terms of newly generated content and increasing challenge as the players learn the game.

## 6. References

Agogino, A.; Stanley, K. & Miikkulainen, R. (2000). On-line Interactive Neuro-evolution, *Neural Processing Letters*, Vol. 11, No. 1, (February 2000) (29-38), 1370-4612.

Baader, F.; Calvanese, D.; McGuinness, D.L.; Nardi, D. & Patel-Schneider, P.F. (2003). *The Description Logic Handbook: Theory, Implementation, Applications*, 0-521-78176-0, Cambridge University Press, Cambridge, UK

Barnes, J. & Hutchens, J. (2002). Testing Undefined Behavior as a Result of Learning, In: *AI Game Programming Wisdom*, Rabin, S., (Ed.), Charles River Media, 13: 978-1-58450-077-3, MA

Briggs, F. (2004). Realtime Evolution of Agent Controllers in Games, In: *Generation 5*, Accessed 2009, Available Online at: http://www.generation5.org/content/2004/realtimeevolutionagents.asp

De Sousa, B. (2002). *Game Programming All in One*, Premier Press, 13: 978-1-93184-123-8, OR

Dybsand, E. (2000). A Finite-State Machine Class, In: *Game Programming Gems*, Deloura, M., (Ed.), Charles River Media, 13: 978-1-58450-049-0, MA

Erol, K.; Hendler, J. & Nau. D. (1996). Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, Vol. 19, No. 1, (1996) (pp. 69-93)

Fikes, R. & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, Vol. 2, (1971) (pp. 189-208)

Garzon, M. H.; Drumwright, E. & Rajaya, K. (2002). Training a Neurocontrol for Talking Heads, *Proceedings of the International Joint Conference on Neural Networks*, pp. 2449-12453, Honolulu HI, May 2002, IEEE Press, Piscataway, NJ

Grand, S. & Cliff, D. (1998). Creatures: Entertainment Software Agents with Artificial Life, *Autononomus Agents and Multi-Agent Systems*, Vol. 1, No. 1, (1998) (pp. 39-57)

Hastings, E. J.; Gutha, R. K. & Stanley, K. O. (2007). NEAT Particles: Design, Representation, and Animation of Particle Systems Effects, *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pp. 154-160, Honolulu Hawaii, 2007, IEEE Press, Piscataway NJ

Haykin, S. (1999). *Neural Networks A Comprehensive Foundation*, 2nd ed., Prentice Hall, 13: 978-0-13273-350-2, NJ

Larid, J. & Lent, M. V. (2000). Human-Level AI's Killer Application: Interactive Computer Games, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 1171-1178, 10: 0-262-51112-6, Austin Texas, August 2000, AAAI Press, Menlo Park, CA

Lioret, A. (2008). An Artifical Intelligence Nodes module inside Blender for expert modeling, texturing, animating, and rendering, *Proceedings of the Blender Conference*, Amsterdam Netherlands, October 2008.

Mathews, J. (2000). Colin McRea 2.0 (PlayStation), In: *Generation 5*, Accessed 2009, Available On-line at: http://www.generation5.org/content/2001/hannan.asp

Miikkulainen, R.; Bryant, B. D.; Cornelius, R.; Karpov, I. V.; Stanley, K. O. & Yong, C. H. (2006). Computational Intelligence in Games, In: *Computational Intelligence: Principles and Practice*, Yen, G. Y. & Fogel, D. B. (Ed.), (155-191), IEEE Computational Intelligence Society

Orkin, J. (2004). Symbolic Representation of Game World State: Toward Real-Time Planning in Games, *Proceedings of the AAAI Workshop on challenges in Game AI*, pp. 26-30, 13: 978-0-262-51183-4, San Jose, CA, July 2004, The MIT Press, Cambridge, MA

Orkin, J. (2005). Agent Architecture Considerations for Real-Time Planning in Games, *Proceedings of Artificial Intelligence and Interactive Digital Entertainment*, pp. 105-110, 13: 1-57735-235-1. Marina del Rey California, June 2005, AAI Press, Menlo Park, CA

Qualls, J.; Garzon, M. & Russomanno, D.J. (2007) "Neural-Based Agents Cooperate to Survive in the Defend and Gather Computer Game," *IEEE Congress on Evolutionary Computation*, IEEE Press, Singapore, pp. 1398-1402

Russel, S. & Norvig, P. (2003). *Artificial Intelligence a Modern Approach*, 2nd ed., Prentice Hall, 13: 978-0-13790-395-2, NJ

Schaefer, S. (2002). Tic-Tac-Toe (Naughts and Crosses, Cheese and Crackers, etc), In: *Mathematical Recreations*, Accessed 2007, Available On-line at: http://www.mathrec.org/old/2002jan/solutions.html

Stanley, K.; Bryant, B. D. & Miikkulainen, R. (2005a). Evolving Neural Network Agents in NERO Video Game, *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, pp. 182-189, Essex UK, April 2005a, IEEE Press, Piscataway, NJ

Stanley, K. O.; Bryant, B. D.; Karpov. I. & Miikkulainen, R. (2005b). Real-Time Neroevolution in the NERO Video Game, *IEEE Transactions on Evolutionary Computation*, Vol. 9, No. 6, (December 2005b) (653-668), 1089-778X

Stanley, K. O.; Bryant, B. D.; Karpov. I. & Miikkulainen, R. (2006). Real-Time Evolution of Neural Networks in the NERO Video Game, *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pp. 1671-1674, Boston, MA, July 2006, AAAI Press, Menlo Park, CA

Watt, A. & Policarpo, F. (2001). *3-D Games Real-Time Rendering and Software Technology*, Addison-Wesley, 13: 978-0-20161-921-8, NY

Zarozinski, M. (2001). Imploding Combinatorial Explosion in a Fuzzy system, In: *Game Programming Gems 2*, Deloura, M., (Ed.), Charles River Media, 13: 978-1-58450-054-4, MA


**Evolutionary Computation**

Edited by Wellington Pinheiro dos Santos

ISBN 978-953-307-008-7
Hard cover, 572 pages
**Publisher** InTech
**Published online** 01, October, 2009
**Published in print edition** October, 2009

This book presents several recent advances on Evolutionary Computation, specially evolution-based optimization methods and hybrid algorithms for several applications, from optimization and learning to pattern recognition and bioinformatics. This book also presents new algorithms based on several analogies and metafores, where one of them is based on philosophy, specifically on the philosophy of praxis and dialectics. In this book it is also presented interesting applications on bioinformatics, specially the use of particle swarms to discover gene expression patterns in DNA microarrays. Therefore, this book features representative work on the field of evolutionary computation and applied sciences. The intended audience is graduate, undergraduate, researchers, and anyone who wishes to become familiar with the latest research work on this field.

**How to reference**

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Joseph Qualls and David J. Russomanno (2009). Applications of Neural-Based Agents in Computer Game Design, Evolutionary Computation, Wellington Pinheiro dos Santos (Ed.), ISBN: 978-953-307-008-7, InTech, Available from: http://www.intechopen.com/books/evolutionary-computation/applications-of-neural-based-agents-in-computer-game-design

INTECH
open science | open minds

**InTech Europe**
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

**InTech China**
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen