

Interactively Evolved Modular Neural Networks for Agent Control

Jessica C. Sparks^a, Roberto Miguez^b,
John Reeder^c and Michael Georgiopoulos^d

^aComputer Engineering, Purdue University, West Lafayette, Indiana;

^bElectrical and Computer Engineering, University of Central Florida, Orlando, Florida;

^cComputer Engineering, University of Central Florida, Orlando, Florida;

^dElectrical Engineering, University of Central Florida, Orlando, Florida

ABSTRACT

As the realism in games continues to increase, through improvements in graphics and 3D engines, more focus is placed on the behavior of the simulated agents that inhabit the simulated worlds. The agents in modern video games must become more life like in order to seem to belong in the environments they are portrayed in. Many modern AI's achieve a high level of realism but this is accomplished through significant developer time spent scripting the behaviors of the Non-Playable Characters or NPC's. These agents will behave in a believable fashion in the scenarios they have been programmed for but do not have the ability to adapt to new situations. In this paper we introduce a modularized, real-time, co-evolution training technique to evolve adaptable agents with life like behaviors. Experiments conducted produced very promising results regarding efficiency of the technique, and demonstrate potential avenues for future research.

Keywords: artificial intelligence, interactive evolutionary computation, machine learning, neuroevolution, reinforcement learning

1. INTRODUCTION

Developing realistic and adaptable agent behavior is an important problem in Artificial Intelligence. A particular application is found in video games. In contemporary video game AI, behavior is scripted. A poorly scripted AI often leads to predictable and easily exploited agent behavior. This can lead to decreased entertainment, replay value, and gameplay bugs or errors. A well scripted AI takes significant time to develop, causing product delay and excessive expenditure of resources. Even if an AI is well scripted, it is still very difficult or impossible to have a scripted AI generalize and adapt to new situations. Machine Learning techniques can be utilized to provide the adaptability needed to produce convincing artificial intelligence in games.

A particular area of machine learning that shows promise in game agent control is neuroevolution. Neuroevolution uses genetic algorithms to evolve artificial neural networks, which can then be used to solve reinforcement learning problems.¹ The concept of implementing neuroevolution in a game was pioneered by Dr. Kenneth Stanley and his group with NeuroEvolving Robotic Operatives (NERO).² NeuroEvolution of Augmenting Topologies, or NEAT,¹ was used to evolve the behavior of the robots in the game. The object of the game is to evolve the most adaptable agents by providing positive and negative reinforcement for specific actions taken by the robots. This was the beginning of a new genre of games, called Machine Learning Games.²

This paper expands on the strategy introduced by NERO in certain key aspects. The problem of agent control is broken into subproblems by introducing a modular approach. This approach is taken to attempt to increase the convergence time and efficacy of evolution.

In order to introduce a finer grain of control for agent development, the paradigms of reinforcement learning and interactive evolution have also begun to be integrated. To advance this effort, a novel approach was taken where every agent is comprised of its own population. In this manner, different populations within the game develop different action policies and thus behave differently for similar states. By being privy to particular state-action pairs and understanding their linkage, human interactivity and reinforcement learning can be made to play an increased role in agent development. It is hoped that by taking a low-level look at states and corresponding

agent action, and adjusting the pairs as one sees fit, that quicker convergence rates and more realistic agent behavior will be realized.

2. BACKGROUND

This section contains the necessary background on the many building blocks of our project. Here reinforcement learning, NEAT and its modifications, interactive evolutionary computation, the tools and code bases used in our project, as well as terms used throughout the rest of the report are described.

2.1 XNA and Net Rumble

XNA⁴ is a set of tools for game developers designed to take the tedium out of game developing. Designed by Microsoft, XNA contains a comprehensive list of libraries to promote code reuse through all levels of game development. XNA also provides a community⁵ in which games can be reviewed by fellow developers. The games created with XNA can be distributed to either XBOX or Windows machines.

Net Rumble⁶ is a 2D game in which the player operates a space ship and tries to shoot other space ships in a free-for-all style battle (see Figure 1). The player can move, shoot, and lay mines. The objective of the game is to be the first to attain a specified number of points. Points are earned by killing another space ship in the game. However, points can be subtracted if the player causes their own destruction (such as by flying into an asteroid).



Figure 1. A screen shot of the Net Rumble game.

2.2 Reinforcement Learning

*Reinforcement Learning*⁷ is a type of machine learning approach in which an agent takes actions that will maximize its reward or *reinforcement*. The reward given is based on the agent's action and the current state of the environment. No other information about how to solve the problem is given to the agent. It must learn through trial and error. This introduces the trade off between *exploration* and *exploitation*. In exploration an agent will take an action that it has no information about. When the agent does not know much about its environment, exploration allows it to gather information about what actions and environment states lead to reinforcement. Exploitation, on the other hand, is when the agent uses the information from previous actions to choose which action to take in the current situation. If an agent always explores then it will never maximize reward, and if it never explores it may not find actions that lead to big payoffs. There is a delicate balance between how much exploration and how much exploitation to use and the optimal ratio is different for each problem.

There are different *reward models* that can be used to maximize an agent’s reward. A reward model is a model of how an agent should take the future into account in the decisions that it is making now. One such model is the *finite-horizon*⁷ model in which reward is maximized over a given number of steps. This model is represented by the expression

$$E\left(\sum_{t=0}^h r_t\right) \tag{2.1}$$

where r_t represents the reward given for time step t and h is the number of time steps in which reward should be optimized. The symbol E in the above expression represents the expected value (average value). The *infinite-horizon*⁷ model is a model in which the reward is to be maximized for all time. This model is represented by the expression

$$E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \tag{2.2}$$

where γ is the discount factor by which future rewards are geometrically discounted and $0 \leq \gamma \leq 1$. The *average-reward*⁷ model is similar to the infinite-horizon model as γ approaches 1. This model is represented by

$$\lim_{h \rightarrow \infty} E\left(\frac{1}{h} \sum_{t=0}^h r_t\right). \tag{2.3}$$

An agent using this model is optimizing its long-run average reward. The problem with this method is that there is no way to distinguish between very high initial rewards and very high late rewards.⁷

One particular reinforcement learning algorithm that is discussed later in the section 2.8 is *Q-learning*.⁷ This algorithm is a type of *temporal difference (TD)* algorithm that keeps track of the estimated reward Q for each state-action pair in the problem space through the function $Q(s, a)$ where s is the environment state and a is the action taken. The agent can use any learning strategy (the Q-learning algorithm is *exploration insensitive*⁷ meaning that the learning strategy taken in the early stages of learning has little effect on the overall performance) until the algorithm converges to the optimal Q values. In Q-learning the agent takes greedy actions, meaning that it takes the actions with the highest estimated Q . Q-learning, however, may be slow to converge on problems with large state or action spaces. This problem is addressed in a different method, described in section 2.8.

2.3 NeuroEvolution of Augmenting Topologies

*NeuroEvolution of Augmenting Topologies (NEAT)*¹ is a neuroevolution method that overcomes some common problems with the neuroevolution methods that came before it.¹ NEAT evolves both the weights and connections of a neural network to find a solution (see Figure 2 for a representation of an artificial neural network). One of the problems neuroevolution methods face is the *competing conventions* problem. The problem is that when two neural networks provide a solution to a problem but have different encodings, these different encodings cause important information to be lost during crossover. To remedy this problem NEAT introduced *innovation numbers*.¹ Innovation numbers are a system of historical markings that ensure that all genes in a neural network are encoded the same way. If a gene was derived from the same historical origins as another gene then it will have the same innovation number. When crossover occurs only genes with the same innovation number are crossed. Innovation numbers that are not common to both parents (*disjoint* and *excess genes*) are inherited from the more fit parent.

Another problem that NEAT solves is the problem of how to prevent new structures from dying off before they have a chance to optimize. NEAT’s solution to this issue is to separate the population of neural networks into *species* of similar structures. Each species then competes among its own members. Each member of the species must share its fitness with other members of the species. This creates *fitness peaks*¹ and each species is limited in size by the size of their peak. This way no one species is allowed to take over the entire population. This prevents innovations from dying out too quickly and keeps the population of neural networks diverse.

NEAT also uses the design principle of starting off neural network topologies minimally. Some neuroevolution methods before NEAT started topologies randomly.¹ This led to nonfunctional structures being implemented

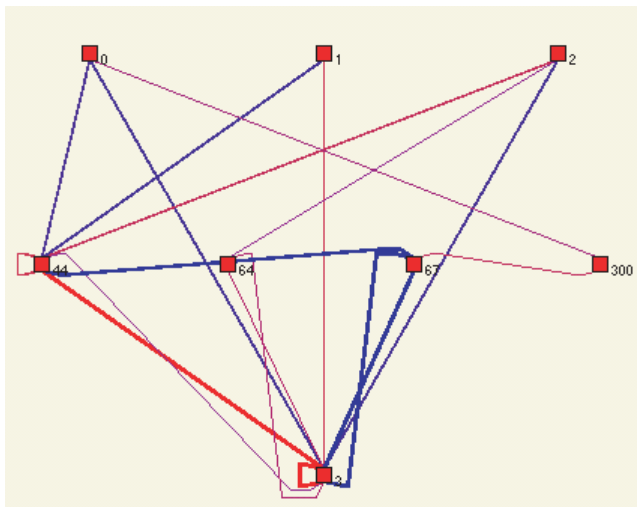


Figure 2. An example of a neural network representation. The input nodes are shown at the top labeled 0-2. Output node, labeled 3, is at the bottom. There are 4 hidden nodes between these. The color and thickness of the lines connecting the nodes represent the weight of that particular connection. Taken from the SharpNEAT GUI. Source code is available at: <http://sharpneat.sourceforge.net/index.htm>.

into the neural networks that could never be removed. However, this was necessary for these methods because otherwise innovations would not survive. Since NEAT has solved the problem of preserving innovations through speciation, it can start off networks minimally with no hidden nodes. Therefore NEAT can add structure only as it is functional which minimizes the solution throughout the entire evolutionary process.

2.4 Real-Time NeuroEvolution of Augmenting Topologies

In a real-time situation, such as a game, NEAT has some drawbacks. If an entire population of game agents is changing at the same time, a human user is likely to notice the difference, which leads to an unrealistic or even disorienting game experience. In order to combat this problem *real-time NEAT (rtNEAT)*² was created.

The idea behind rtNEAT is that the population be evolved gradually so the change in population is largely unnoticed by the user. This is done by replacing only one member of the population at a time. In each loop of the algorithm the fitness of each individual is calculated, the two most fit individuals are crossed over to form an offspring, and the least fit individual that has had enough time to optimize is replaced by this offspring. This provides a more gradual evolution that is more suited to gameplay.

2.5 NeuroEvolving Robotic Operatives

NeuroEvolving Robotic Operatives or *NERO*² is a game that uses rtNEAT to evolve “robotic” game agents that are suited to fighting in different situations. NERO is part of a new genre of game known as a Machine Learning Game (MLG)² in which Machine Learning is used to train the agents in the game. In MLGs the user’s role is to define the best fitness function for the game agents. The game starts in training mode where the user trains an army of robots. The user specifies the strategy the robots should use though a reinforcement interface in the lower right corner of the screen (see Figure 3). The user can place enemies, walls, and flags and specify how much positive or negative reinforcement should be given for certain actions taken (approach flag, approach enemy, avoid fire, stand ground, stick together, etc.). The user can also interact with their team by using converge, inhabit, smite, and milestone options. Converge allows the player to converge the entire team to one robot’s neural network; inhabit allows the player to control a robot’s movement so the team can learn by example; smite eliminates a poorly behaving neural network from the population; and milestone saves a team after the user feels they have learned a certain task according to their own judgement so that they can be recombined with the team later. *Milestoning* is a topic covered later in the background section of this paper. When the robots have trained to the user’s satisfaction they can be pitted against another player’s army in battle.



Figure 3. The reinforcement interface for providing feedback to the agents in NERO. Starting from the left the icons represent: approaching the enemy, hitting the enemy, avoiding fire, approaching a flag, sticking together, and standing ground.

2.6 Modular NeuroEvolution of Augmenting Topologies

Problems with a large search space can be hard to solve for NEAT.⁸ If there are too many inputs and outputs to search through, NEAT can take a long time to converge to a good solution. *Modular NEAT*⁸ provides a solution to this problem. Modular NEAT breaks down problems into subproblems or *modules*, and evolves each of these problems separately. Through this method, using a coarser search leads to a more robust solution that is converged upon more quickly.

Modular NEAT starts by evolving individual modules that can have input and output nodes as many as the maximum number of input and output nodes for the problem at hand. These modules are evolved using the NEAT algorithm with the addition that input and output nodes can be added through mutation as long as their numbers never exceed that of the total solution. The modules are then *bound* to the overall solution through *blueprints* or mappings of modules to the solution’s inputs and outputs. A representation of this is shown in Figure 4. Blueprints are given more generations to evolve than the modules to give them a chance to optimize to the current modules. The ideal ratio of module generations to blueprint generations is about 1:5.⁸

Fitness is assigned to the blueprints and this fitness is shared with the modules that the blueprints use. Module fitness is determined by the number of times the module is used in the blueprint population. If a module is used more often, a lower fitness is assigned to it. This gives rare and new modules the chance to optimize. This is yet another innovation saving strategy of the NEAT framework.

2.7 Milestoning with NeuroEvolution of Augmenting Topologies

*Milestoning*⁹ is a technique that can be used with NEAT to remember learned behaviors that may conflict with behaviors that are currently being learned. Milestoning works by saving a few of the most fit individuals in a *milestone pool*.⁹ These individuals are crossed over with the current population periodically. This creates offspring that have newly learned behaviors along with past learned behaviors.

2.8 NEAT-Q

*NEAT-Q*³ is another modification of NEAT that uses a type of reinforcement learning *Q-learning* (see background of Q-learning in 2.2) to approximate a value function $Q(s, a)$. This value function now has a network associated with it and this network is evolved with NEAT to find the appropriate representation of the problem to be learned. This can be very useful if the problem at hand has non-discrete states and actions. The only change to NEAT that is needed to convert it to NEAT-Q is the way the outputs of the neural networks are interpreted. These outputs now represent the particular state-action pair’s long-term discounted value, and they are used to update the other state-action pairs’ estimations, in addition to being used to choose which action to take.³ The result of each action is then backpropagated to the desired target. This results in neural networks that are better able to learn a problem by finding a good representation of it.

2.9 Interactive Evolutionary Computation

*Interactive Evolutionary Computation (IEC)*¹⁰ uses evolutionary computation and subjective human input to produce results that don’t have discrete, well defined solutions. In IEC the user input becomes the fitness

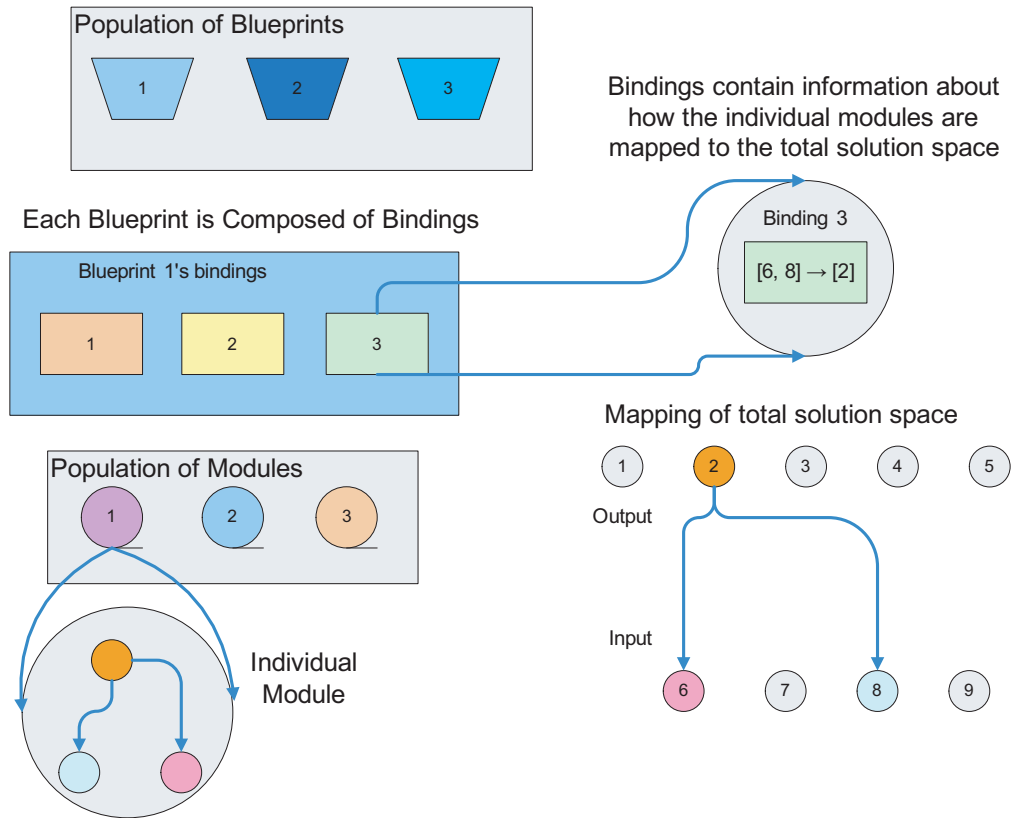


Figure 4. A visual representation of a module being bound to the total solution through a blueprint.

function to which the solution is being optimized. In this manner, the solutions produced can be based off of artistic preference, emotional understanding, impressions or biases that the user has. These types of solutions cannot easily be defined by equations or discrete states, so using a human evaluator is often the only way to quantify the goodness of solutions to these types of problems.¹⁰

3. APPROACH

It was decided early on that although our prototype would be a computer game, game development was not the priority. Since Microsoft's XNA framework⁴ abstracts many of the nuances of game development, it was used for the project. The Net Rumble starter kit⁶ was used as a starting point for the development of our prototype (Background 2.1).

Net Rumble, as the name implies, can be played over a network. Multiple players can play at once if they are connected via a network. The style of gameplay is a free-for-all, and points are scored for killing any ship and taken away for self-destruction. Since the goal was to create intelligent agents, the first task was to create non-playable characters, or bots, that would take part in a multiplayer game. Naturally, the bots needed to have the exact functionality of a normal network gamer.

3.1 Bot Integration

The logical approach would have been to derive a class Bot from the class NetworkGamer in the XNA framework, but XNA disallows this. Also, the collection of gamers that is constantly used by XNA, the AllNetworkGamers collection, is read-only and could not be modified. Due to this, a BotFactory and Bot class had to be created. The BotFactory would create Bots, and maintain them in a dynamic linked list. This allowed handling of bots to be fully abstracted away from the methods that handled network gamers.

A key design strategy was to emulate the Bot class after the NetworkGamer class in XNA. Also, since bots were going to use ships just as normal network gamers, every effort was made to use the same composed classes and method signatures as a gamer did in Net Rumble. This allowed for maximum code reuse, and allowed for quick and secure integration into the physics of the game. Following this procedure and making small adjustments, bot ships were introduced into the game which properly bumped into other objects while exhibiting effects of acceleration and drag.

Since Net Rumble is a networked multiplayer game, existing methods could not be used for bot ship death, damage, and score updating. Every machine that connects to Net Rumble only updates its own ship, and sends information to the other machines regarding the update. Every ship death occurs on the local machine and the local machine only. Information of this death is communicated via the network to other machines.

This fact then created two problems: the problem of properly updating bots, and the problem of properly transmitting this over the network. The solution was to handle updates of every bot damage, death, and score locally on the host, and then transmit this data to all of the other players. Packets were transmitted using the TCP protocol, and were transmitted asynchronously. The proper sequence of transmission was key here however; there were some initial problems with out-of-sequence transmission and different bot updates on different machines. At the end of this phase, there was a fully abstracted BotFactory that could create bots that had the full functionality of network gamers, both on local and networked machines.

One of the essential goals of this integration was to provide a clear interface for bot control. The interface was constructed such that all that was needed was a sequence of inputs to move the bot. This was done through *sensors* which gave the bots information about their environment.

3.2 Sensors

No intelligent decisions could be made without having knowledge of the current state of the environment. A sensor class was created to sense the environment. The sensor class is able to sense other ships, asteroids, projectiles, and power ups. It is only able to sense a given amount of any of these objects at any point in time, and a parameter is passed to specify this. For example, a sensor package can be created such that a bot can sense a maximum of 5 closest ships, 3 closest asteroids, and 1 closest power up at any point in time.

Ship, asteroid, and power up sensing is done by iterating through the position of each world object, and taking the Euclidean distance from the bot's position to the object's current position. If it is less than a given parameter, then it is considered as within *sight* of the bot, and is sensed. This is equivalent to every bot having a circle, of radius r , where if a particular object is within the circle, it is sensed (see Figure 5). If it is sensed, positional information is saved by the bot. This information is either relative Cartesian coordinates, or relative Polar coordinates. Either system provides translational invariance of input to the network, while relative polar coordinates provides rotational invariance. This is important since it minimizes the amount of inputs that the network must learn to provide output for.

Projectile sensing works differently than the sensing of other objects. Instead of the bots sensing the projectiles, the projectiles sense the bots. If a bot were to sense a projectile, it would need at least three degrees of freedom to properly classify it: 2 degrees for positional information, and one degree for its heading. That is, even if a projectile is within a bot's circle of vision, there is no way to tell if that projectile is heading directly towards the bot or if it is flying away from it. However, if a projectile senses a bot, then only two degrees of freedom are necessary to properly classify the projectile, two dimensions for its positional information. This is because a projectile will always be incoming if it detects a bot. By choosing this method, only two extra inputs need to be added to the network instead of three. The savings are fairly significant if a bot is sensing 4 or 5 projectiles at once. A projectile senses a bot by calculating the angle between its velocity vector and a bot's position, and calculating the distance between the two. If both the angle and the distance are less than certain thresholds, the bot is sensed, and positional information of the projectile is passed to the bot.

The relative positions are saved into a dynamic sorted list, where the keys are relative distances, and the values are positional information. This enables the bot to sense the ships most pertinent in accordance to its field of vision. For example, if only 3 ships can be sensed at once, and 5 ships are within r , the ship will sense only the 3 closest ships in its sense radius. Note, this also allows indiscriminate sensing of ships, there is no difference between network gamers and other bots.



Figure 5. A screenshot from the Net Rumble game that shows the sense radius of a non-playable character.

3.3 Intelligent Control

In order to create competent agents for game control, it is necessary to begin from an algorithm that is capable of learning from its environment in a reinforcement learning framework, as well as being able to generalize well to deal with unknown states. One such algorithm that has been shown to have these traits is NEAT or *NeuroEvolution of Augmenting Topologies*. For this reason it was chosen as the basis for our current approach. It is used in the context of a *Reinforcement Learning* problem, of which agent control can easily be interpreted. Elements of *Interactive Evolution* are also introduced to allow the human users to affect the direction the agents take in their learning.

Initially, NEAT was integrated into the Net Rumble code without any modification. The outputs of the bot sensors were fed as inputs to a particular network, forward propagation would occur, and then the outputs of the network were fed as inputs to bot control. Although this worked as expected, evolution was found to hinder gameplay. This is because each generation of NEAT alters the population, and immediate changes in every bot's behavior is easily noticed. This problem is one of the issues that rtNEAT (Section 2.4) addresses.

Our implementation was modified to follow the rtNEAT paradigm of evolving a static population in order to decrease the noticeable effect in generational changes. In rtNEAT only a single network, the weakest individual in the population, is replaced during evolution. As this occurs the bot routinely transitions between population members. This makes evolution fairly transparent to the user.

In order to facilitate an encapsulated agent design, it was also decided that each bot would maintain a unique population during gameplay. These populations represent a collection of policies that control the agent behavior. Net Rumble, being a free-for-all game, allows each of the bots to compete with each other to stay alive and accumulate points. This in turn means that each of the populations controlling the bots are competing against each other, giving the agents another source of information to learn from. This design choice will also facilitate the incorporation of techniques used in the NEAT-Q algorithm. These techniques will allow the agent to be reinforced on a state-action pair level, thus giving the users a finer grain of control on the agent's behavior through the interactive evolution interface. Each bot maintaining an independent population means that the total population now consists of a series of distinct and independent sub-populations, where each sub-population corresponds to a specific bot. This enables each population to evolve separately. This also allows each population to evolve a particular policy for the environment, where these policies may be distinct. Having each population of bots learning a certain policy is useful, since this will facilitate integration of both reinforcement learning and interactive evolution into the algorithm. Just as in the single population case, rtNEAT is applied to every sub-population.

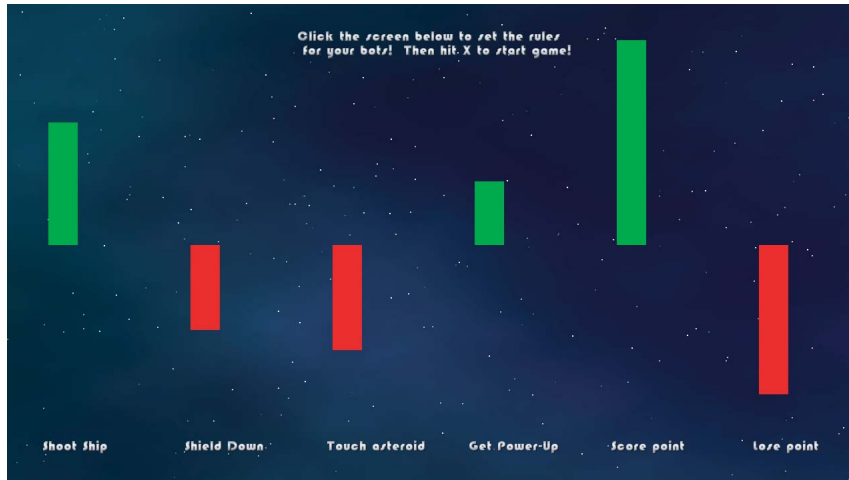


Figure 6. Reinforcement Interface of the Net Rumble game. The size of the red and green bars can be changed by clicking and dragging. Green bars correspond to positive reinforcement and red bars correspond to negative reinforcement.

To simulate a single population where each individual is constantly being evaluated, the bot switches between networks at regular intervals. This allows evaluation of every individual in the population, although only a single individual is in the game at any point in time. Since the ship itself is in direct contact with the environment, the ship accumulates fitness during gameplay. Negative actions, such as bumping into an asteroid or getting shot, are given negative reinforcement by decreasing fitness, while positive actions, such as scoring a kill or shooting another bot, are given positive reinforcement by increasing fitness. When a ship switches networks, it dumps its fitness to the network that was just controlling it, resets its fitness, and then takes on a new network to control it. In essence, the brains are switched in and out while the body remains the same. This abstraction allows us to replace or extend intelligent control without worrying about bot interaction with the environment.

Real world control problems can be difficult to learn because of high input/output count. Modular NEAT showed that breaking a problem down into subproblems and evolving networks to solve the subproblems can increase average fitness as well as convergence times. For this reason our agents are controlled by multiple networks designed to handle separate agent tasks. For example, two networks, one for movement and one for shooting, instead of a single network, are used to control a bot. One network can be removed independently of the other, hence the term modular. Note, that our modularization is not the same as Modular NEAT (Section 2.6) since we do not use blueprints to bind the networks to a total solution. Instead the domain knowledge of a human user is used to decide how the problem can be divided into subproblems. In future work, the game interface will allow the user to select how many, and in what fashion the networks are combined to control the agent.

In order to examine the real-time modular approach, mentioned above, a test case was designed in which one neural network handles moving, while another one handles shooting. Currently, both networks receive the same inputs, the outputs from the bot’s sensors. Each network provides 9 outputs. Outputs correspond to each direction a bot can move towards, that is the cardinal directions and their respective diagonals, and to the command of not moving at all. In order to choose the action of the bot, all of the outputs are polled and the output with the highest activation is selected. This is then mapped to an action of the agent. In order to keep the sensors and actions rotationally independent in the 2-D space of the game, the chosen action is transformed to be relative to the bot’s current reference frame.

3.4 Interactive Evolution

In order for the user to provide their input to the behavior of the bots, an interactive screen was added to the game. This screen would allow the user to provide positive and negative reinforcement values for specific actions in the game. A visual of our interface is shown in Figure 6. The height of the red and green bars correspond to the reinforcement given for particular actions and this information is fed into the neural networks. This screen can be accessed at any time during gameplay through the pause menu.

Human players are also able to have a direct role in the evolution of the bots by playing amongst them. This is a form of direct interactive evolution since the human player is providing a style of gameplay that the bots must learn to play against.

3.5 Storing and Loading Genomes

A database was created where one can both save and load bot populations that have been evolved. Not only can certain desirable populations be archived, but one can stop and then resume evolution at a later time. Also, since the bot populations can be stored on a centralized server, users can retrieve existing populations and evolve them. Due to this feature, the process of interactive evolution is extended from being confined to a single individual to a collaborative process, allowing multiple individuals to play a role in evolving the agent behaviors. These features allow the agents to benefit from greater exposure to training time than would be expected from a normal single users play time. This is important since evolutionary processes take advantage of large population sizes and high evaluation counts to effectively search for more robust solutions. This is a disadvantage that real-time applications often face, since they can not run in faster than real-time simulations, and a large amount of real-time is required to reach high evaluation counts.

Due to the existence of modular networks, shooting networks and moving networks from different populations can be combined to form a single population. To further extend modularity, one could also pick and choose genomes from different populations to form a single population that will control a bot. Perhaps a population whose policy is aggressive can be combined with a defensive population to evolve a hybrid.

4. RESULTS

4.1 Experimental Design

Experiments were run to test the validity of the modular network approach. The game was run with the same settings with only a single neural network controlling the bots and with modular neural networks controlling the bots. The average fitness of the bot (the ship itself) per evaluation was tracked, as well as the total fitness of the population driving the bot, for both approaches.

Five experiments were run for each setup, for a total of ten experiments. Due to the stochastic nature of the gameplay, the five experiments were run on five different machines, making for a total of 25 experiments per control setup. Each experiment was run for 200 evaluations, where each evaluation was defined by the completion of an episode for every network for every bot in the game. An episode was defined to be the entire interval of time that a network was present in the environment and its fitness was being determined, that is, the entire time a ship was being controlled by a particular network.

Each game had 10 bots present, with a population comprised of 20 networks. In the modular approach setup, there were 10 networks to control shooting and 10 networks to control moving. In the single network setup, all 20 networks controlled both actions. A bot switched between networks every 8 seconds, and evolution of its population occurred every 40 seconds. All networks had 18 inputs, with a sensing radius of 500 pixels. The single network had 18 outputs (9 options for shooting and moving respectively), while the modular networks had 9 outputs each, since each network controlled a single action.

For all experiments, fitness was defined by adding positive or negative scalars to a ship's fitness depending upon the actions it took. Positive reinforcement was given for a bot damaging other bots (+2), and for killing another bot (+2). Negative reinforcement was given for taking shield damage (-.25), for having shields completely down (-.5), for hitting an asteroid (-1), and for committing suicide (-1). Accuracy was tracked by keeping count of how many projectiles a bot shot, and how many connected. Unlike the other fitness changes which were immediate, accuracy was analyzed at the end of an episode. If accuracy was above (50%) then positive reinforcement was given (+5) and if it was below this threshold negative reinforcement was given (-5). This constant is significantly higher than the others since it only impacts fitness once per episode. It is important to note here that in the modular experiment setup, both the moving network and the shooting network shared fitness updates. This was done in order to have both networks evolve towards a similar goal. The idea was to co-evolve both networks so that although their function and composition are distinct, their behavior complements each other.

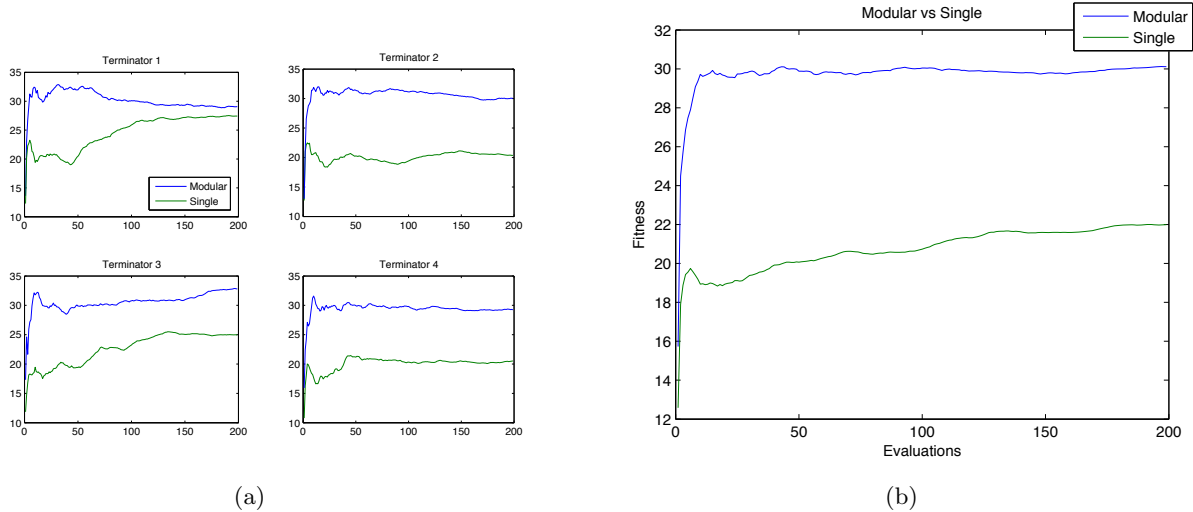


Figure 7. (a) Fitness Function values for Modular versus singular networks in individual NPCs (b) Fitness Function Values for Modular networks versus singular networks averaged over all bots.

4.2 Experimental Results

The results from the tests were averaged over each experiment to show the expected fitnesses as the agents progress. The results are summarized in Figures 8(a) and 8(b). Figure 8(a) shows average fitness of the first four individual bots averaged over every experiment, and Figure 8(b) shows the average fitness of all of the bots averaged over every experiment. It is important to note that the maximum fitness in this type of scenario is not readily available as the fitness happens in real-time from a non-deterministic world. These figures indicate the differences in achieved fitness between the modular and single network approach.

From Figure 8(b), it is evident that the modular approach outperforms the single network approach. This substantiates the hypothesis that the modular approach would reach a higher asymptotic fitness and reaches the same level of fitness as the singular case in fewer evolutionary steps. This result mirrors the findings of the Modular NEAT approach used in self similar board game domain.⁸ The speed to convergence is an important factor in a real-time scenario since each evaluation below optimal has a cost on the agents overall fitness level.

The initial noise found in the individual fitness curves in Figure 8(a) is also interesting, and likely due to the competing agent sub-populations. The figure shows the fitness of Terminator 1 jumping up very early and then settling to a lower level at later evaluations. This is an interesting result that demonstrates the increasing complexity of the problem as other agents learn and become more dangerous adversaries. In this way the whole population, consisting of all the agents individual populations, learns how to play efficiently. It is also important to realize that these results are dependent on the particular fitness function defined for the experiment as well as the size of the arena and number of bots in the game. In future testing the effects of competing against live human players will be tested. It is unlikely that the population would reach a steady state in that scenario since the input from the human would be non deterministic.

The results from these experiment indicate that the modular network approach would perform well in different scenarios, including against humans, as a result of its fast convergence. It is important to note here that the modular networks achieved their performance through co-evolution. This was achieved by making sure that the fitness that both of the modular neural networks were trying to maximize was linked. This avoided the potential problem of a bot knowing how to move, and knowing how to shoot, but not knowing how to both move and shoot for optimal results. Therefore, both networks evolved by learning how to work with each other, not independently.

5. CONCLUSIONS & FUTURE WORK

5.1 Conclusions

In this paper, the novel approach of user defined modular networks for game agent control was introduced. Agents within the 2D space fighter game *Net Rumble* were trained and controlled with the modular network approach, and tested against the singular network approach, to verify the effectiveness of the modular technique.

This project was built upon the XNA game frame work and sharpNEAT libraries to facilitate rapid development and prototyping. These frameworks provided a foundation and testing environment for the approach implementation, and saved significant development time. The XNA framework provides many avenues for AI development, in the form of freely available game code and development tutorials, that allow AI researchers to try their algorithms in simulated environments with a minimal amount of development effort.

The modular network approach was formulated by taking key aspects of prior work, namely NERO² and Modular NEAT,⁸ and combining them to achieve better performance, as was shown by the experimental results. This work shows that the modular neural network paradigm’s performance, as shown in Modular NEAT, transfers to a real-time reinforcement scenario. This modification of rtNEAT to follow the modular paradigm as well as the introduction of encapsulated agent populations is a novel approach in game agent control and shows significant promise for future research avenues.

The inclusion of human players in the game world, as well as the ability of the user to modify the fitness weights of the agents, allows the evolution of the agents to be affected by a human user. This offers the human the opportunity to guide the agent toward desired behaviors. The quick convergence of this modular approach will compliment the human interaction well, allowing the agents to quickly learn from the human participants.

The results of this project are very encouraging, and indicate that the modular approach is a fruitful design choice in real-time agent control. It is expected that additional testing and the pursuit of future work discussed below will lead to more effective agent control and more life like game AI.

5.2 Future Work

Although the modular network approach outperformed the single network approach, it still remains to be seen exactly how tightly knit the different modules need to be in terms of learning. In our experiment, both the moving and shooting networks shared the same fitness. However, it might be possible that improved performance could result from an independent evolution of separate networks, where each network would specialize in a respective action.

Further work integrating reinforcement learning techniques from NEAT-Q into real-time evolution will be done to increase agent effectiveness in the game environment. These techniques are desired since they will enable more finely tuned control over evolved behavior. By working at the level of state-action pairs, one possesses a finer grain of control over behavior in any given situation. This will also significantly enhance the ability to introduce real-time interactivity into evolution, since a human can modify single actions or value functions at the state level. It is hoped that combining techniques from these three paradigms (*Reinforcement Learning, Neuroevolution, and Interactive Evolution*) will provide a good level of control during evolution, and will lead into achieving satisfactory results faster.

ACKNOWLEDGMENTS

We thank the University of Central Florida and the Florida Institute of Technology for their support and providence. Especially the ML² Lab at UCF. Thanks also goes to the REU faculty mentor, Dr. Georgiopoulos, and the graduate mentor, John Reeder. Roberto also wishes to thank the UCF McNair Scholars Program, and his parents for their support.

This material is based upon work/research supported in part by the National Science Foundation under Grant No. 0647120 and Grant No. 0647018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

- [1] Stanley, K. O. and Miikkulainen, R., “Evolving neural networks through augmenting topologies,” *Evolutionary Computation* **Vol. 2**(No. 10), 99–127 (2002).
- [2] Stanley, K. O., Bryant, B. D., and Miikkulainen, R., “Real-time neuroevolution in the NERO video game,” *IEEE Transactions on Evolutionary Computation* **Vol. 9** (December 2005).
- [3] Whiteson, S. and Stone, P., “Evolutionary function approximation for reinforcement learning,” *Journal of Machine Learning Research* **Vol. 7**, 877–917 (May 2006).
- [4] “XNA.” <http://www.xna.com/> (2007).
- [5] “XNA Creators Club Online.” <http://creators.xna.com/> (2008).
- [6] “Net Rumble download page.” <http://creators.xna.com/en-us/starterkit/netrumble> (December 2007).
- [7] Kaelbling, L. P., Littman, M. L., and Moore, A. W., “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research* **Vol. 4**, 237–285 (May 1996).
- [8] Reisinger, J., Stanley, K. O., and Miikkulainen, R., “Evolving reusable neural modules,” *Springer-Verlag Berlin Heidelberg*, 69–81 (2004).
- [9] DSilva, T., Janik, R., Chrien, M., Stanley, K. O., and Miikkulainen, R., “Retaining learned behavior during real-time neuroevolution,” *American Association for Artificial Intelligence* (2005).
- [10] Takagi, H., “Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation,” *IEEE* (2001).