Interactively Evolved Modular Neural Networks for Game Agent Control

John Reeder, Roberto Miguez, Jessica Sparks, Michael Georgiopoulos, and Georgios Anagnostopoulos

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 artificial intelligence approaches 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 evolution training technique to evolve adaptable agents with life-like behaviors. Online performance during evolution is also improved by using selection mechanisms found in temporal difference learning methods to appropriately balance the exploration and exploitation of control policies. These methods are implemented and tested using the XNA framework producing very promising results regarding efficiency of techniques, and demonstrating many potential avenues for further research.**

I. 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 games 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 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 capable agents to compete against other teams 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 methods introduced by NERO in certain key aspects. In particular, the agent control problem is broken down into sub problems, and a modular neural network architecture is used to increase convergence rates and efficacy of evolution. Furthermore, we implemented techniques from *on-line evolutionary computation*, as introduced in [3] to increase on-line performance.

In our work, in order to introduce a finer grain of control for agent development, the paradigms of reinforcement learning and interactive evolution have also been integrated. To facilitate this effort, an approach was taken where every agent is composed of an individualized population. In this manner, individualized populations within the game develop different policies and thus behave differently for similar states. This enables the use of an ϵ-greedy selection mechanism to search for an optimal policy. It also allows promising solutions to be further investigated while improving online performance during evolution. Furthermore, by involving human interaction it is hoped that evolution will be led to more promising paths earlier in the evolutionary cycle. The integration of reinforcement learning from the environment and human interaction is expected to achieve more desirable agent behaviors in a more efficient manner.

II. BACKGROUND

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

A. XNA and Net Rumble

XNA [4] 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 community [5] in which games can be reviewed by fellow developers. The games created with XNA can be distributed to either XBOX or Windows machines.

Net Rumble [6] 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

John Reeder (*jreeder@mail.ucf.edu*), Roberto Miguez and Michael Georgiopoulos are with the University of Central Florida. Jessica Sparks is with Purdue University. Georgios Anagnostopoulos is with the Florida Institute of Technology.

978-1-4244-2974-5/08/$25.00 ©2008 IEEE

be subtracted if the player causes their own destruction (such as by flying into an asteroid).

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

B. 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. A policy in *Reinforcement Learning* is a complete mapping of states to actions. NEAT can be viewed as an example of *policy search* reinforcement learning. NEAT evolves networks that map the states to the action an agent should take, and thus comprise a policy for an agent.

In *Reinforcement Learning* there is always a tradeoff 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 the 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.

C. On-Line Evolutionary Computation

In order to better balance exploration and exploitation in real time evolutionary systems, the concept of *on-line evolutionary computation* was introduced in [3]. *On-line evolutionary computation* borrows selection mechanisms from *Temporal Difference* (TD) learning methods in order to achieve better performance during on-line evaluation.

In this paper, ϵ-greedy selection is used when choosing a network to evaluate. Selection through ϵ-greedy works by choosing the best policy from a set of policies with a probability of $(1 - \epsilon)$ (exploitation), and a random policy with a probability of ϵ (exploration).

D. NeuroEvolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies (NEAT) [1] is a neuroevolution method that overcomes some common problems with the neuroevolution methods that preceded it. 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*. 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* 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. This led to nonfunctional structures being implemented 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 necessary,which minimizes the solution and solution search throughout the entire evolutionary process.

E. 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

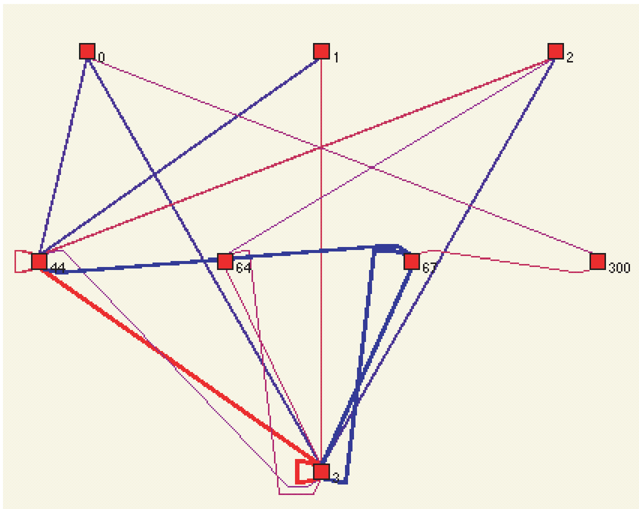


Fig. 2. An example of a neural network representation. The input nodes are shown at the top labeled 0-2. Output node, labeled 3, is at the bottom. There are 4 hidden nodes between these. The color and thickness of the lines connecting the nodes represent the weight of that particular connection. Taken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

game experience. In order to combat this problem *real-time NEAT (rtNEAT)* [2] was created.

The idea behind rtNEAT is that the population should be evolved gradually so the change in population is largely unnoticed by the user. This is done by adapting the NEAT techniques to a steady state genetic algorithm 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.

F. 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 through a reinforcement interface in the lower right corner of the screen When the robots have trained to the user's satisfaction they can be pitted against another player's army in battle. In our paper we used key ingredients of the work conducted in *NERO* [2].

G. Modular NeuroEvolution of Augmenting Topologies

Problems with a large search space can be difficult for NEAT to solve. 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 counter to this problem. Modular NEAT breaks down problems into subproblems by evolving reusable NEAT *modules*. By evolving reusable modules, modular neat abstracts the search space allowing more complex solutions to be searched more efficiently. However, this abstraction leads to a coarser search, which can cause some solutions to be missed.

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 evolved mappings of modules to the solution's inputs and outputs. The modules and blueprints are co-evolved until a solution is reached.

H. Interactive Evolutionary Computation

Interactive Evolutionary Computation (IEC) [9] 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 function to which the solution is being optimized. In this manner, the solutions produced can be based on 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.

III. Methodology

The XNA Starter Kit *Net Rumble* [6] was used to provide a framework and testing ground for experimentation. The libraries provided by the XNA framework reduced the amount of time needed to begin working with a fully functional environment and saved considerable development time and effort.

Net Rumble also provides networked game play allowing multiple players to be involved in the game simultaneously. This also allows multiple human players to be involved in the interactive evolutionary process at the same time.

A. Bot Integration

The Net Rumble Starter Kit only provides functionality for human players. In order to implement our methods, computer controlled ships, or bots, needed to be added into the game. A Bot Factory was developed to handle all facets of bot creation. The bots were necessarily identical to human controlled ships in all respects, except of course, their method of control.

In order to allow for more than one human individual to play alongside the bots and take part in their evolution (Section II-H), networking functionality was maintained. This caused a shift in the game's network architecture to a client-server model, where the host is in charge of bot creation and transmitting bot updates to all other machines.

B. 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 amount. 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 3). 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 will be important later on, since it minimizes the amount of inputs that the network must learn to provide output for.

Projectile sensing is handled differently from other game objects. In normal game play scenarios there are many projectiles in space at any given time, and sensing these in the same way as other objects would overload the sensors without providing very useful information because large numbers of projectiles are likely to pass through the sensor radius of our bots while flying through the game environment. In the case of projectiles we are only really concerned with the ones that are on a collision course with our bots. Since our bots are able to change direction at any time, and projectiles have a fixed heading, it is easier to calculate which projectiles are heading for the bots from the projectiles perspective. For each projectile a cone of influence is calculated and each bot that falls inside that cone of influence is alerted to the position of that projectile.

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 most pertinent ships, 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, that this also allows indiscriminate sensing of ships, that is, there is no difference between human gamers and other bots.

C. 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. 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.

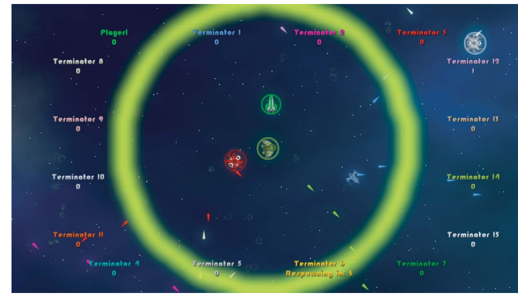


Fig. 3. A screenshot from the Net Rumble game that shows the sense radius of a non-playable character.

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 occurred, 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 II-E) 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 and incorporate intelligent selection mechanisms, 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.

Since each bot maintains an independent population, this means that the total population consists of a series of distinct and independent sub-populations, where each sub-population corresponds to a specific bot. This 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.

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 II-G) 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, 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.

D. Policy Selection

To simulate a single population where each individual is constantly being evaluated, the bot switches between networks, or policies, at regular intervals.

Two selection methods were evaluated. The first method simply iterates through the population in a linear fashion choosing each network to be evaluated in turn. This mechanism would be similar to selecting a policy at random to evaluate since the agents will be at arbitrary positions when the networks are put in control. Also each network gets evaluated an equal number of times throughout the agents lifetime. The second method uses ϵ-greedy selection, where a random policy is chosen for exploration with probability ϵ, and the best policy is exploited with probability $(1 - \epsilon)$. This method is expected to increase the overall performance of the agent, since the more proficient networks are given control of the agent more frequently. This however will mean that some of the networks will have few evaluations than better performing networks, and thus fewer chances to increase their fitness. Both methods are compared in the results section.

Although selection of the network for evaluation differs, the evaluation of any given network is the same. Since the ship itself is in direct contact with the environment, the ship accumulates fitness during gameplay. Negative actions,

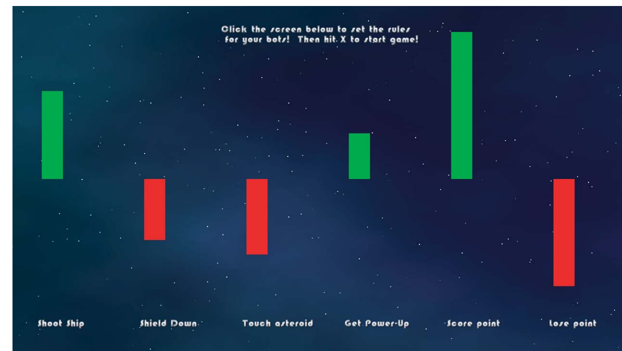


Fig. 4. Reinforcement Interface of the Net Rumble game. The size of the red and green bars can be changed by clicking and dragging. Green bars correspond to positive reinforcement and red bars correspond to negative reinforcement.

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.

E. Interactive Evolution

An interactive screen was added to the game to allow the user to provide their input to the behavior of the bots. 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 4. 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 can take the form of a single individual playing against bots, or a group of individuals doing so. There are no constraints on human to bot ratios, thus equal or more humans than bots can play at any given round and affect evolution collectively. In this manner, evolution is directed by both the bots and the humans simultaneously. By altering the bot to human ratio, either co-evolution or interactive evolution can be emphasized.

F. 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 faster than real-time simulations, and a large amount of 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.

IV. Results

In this section we detail the initial experiments conducted using the *Net Rumble* game environment. Experiments were run to test the validity of the modular network approach, and the $\epsilon - greedy$ selection approach. Experiments to validate the human interaction, and the agent performance against humans and scripted agents will be carried out in future work.

A. Experimental Design

To test the modular design, 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, in both approaches. The same experiment was conducted for both simple iterative selection, and $\epsilon - greedy$ selection to compare the two. Only the selection mechanism was changed from one experiment to the other.

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 composed 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.

B. 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 5(a) and 5(b). Figure 5(a) shows average fitness of the first four individual bots averaged over every experiment, and Figure 5(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 5(b), it is evident that the modular approach outperforms the single network approach. This gives credence to the hypothesis that the modular approach would reach a higher asymptotic fitness and that it would reach 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 fluctuations found in the individual fitness curves in Figure 5(a) is also an interesting phenomenon, 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. 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 examined.

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 attained 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 at the same time. Therefore, both networks evolved by learning how to work with each other, not independently.

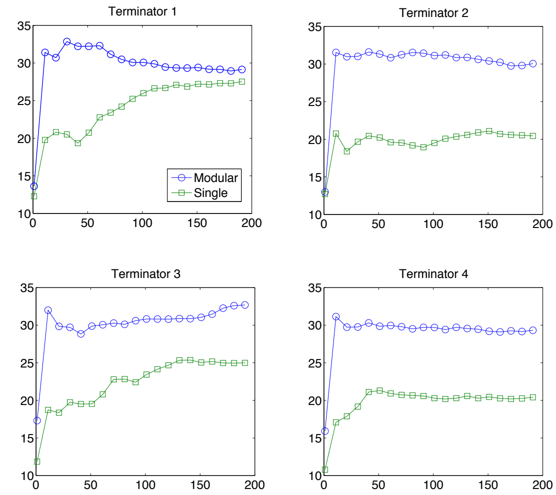
The experiments carried out using ϵ-greedy selection indicate that the overall performance of both the single and multi-networked cases are improved. This result matches our expectation that exploiting the best performing networks of a population some of the time increases the on-line performance of the population without negatively affecting evolutionary process. Figure 6 shows that the ϵ-greedy selection method increases the maximum fitness attained for both architectures. In the multi network case, ϵ-greedy selection improves maximum fitness to level 36.76 compared to level 30.05 attained with iterative selection, while in the singular network case the improvement is more profound increasing fitness to level 33.5 from level 22.01, attained with iterative selection.

V. Conclusions

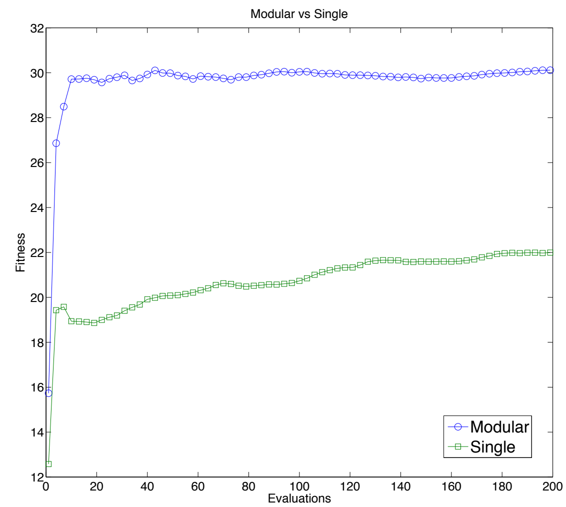
In this paper, the novel approach of user defined modular networks for game agent control was introduced. Agents within the 2D space fighter game *Net Rumble* were trained and controlled with the modular network approach, and tested against the singular network approach, to verify the effectiveness of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

This application 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 NERO [2] 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



(a)



(b)

Fig. 5. (a) Fitness Function values for Modular versus singular networks in individual NPCs (Terminators 1-4) (b) Fitness Function Values for Modular networks versus singular networks averaged over all bots. (Evaluations are for the NPC not individual networks)

game agent control and shows significant promise for future research avenues.

By using intelligent selection mechanisms such as ϵ-greedy, exploration of policies can be appropriately balanced with exploitation. This prevents on-line performance from degrading needlessly, and also enables a deeper search into current optimal policies. The combination of real-time NEAT and on-line evolutionary computation prevents the evolution of game agents from being obvious or even apparent to a user, and thus provides a more realistic gaming experience.

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

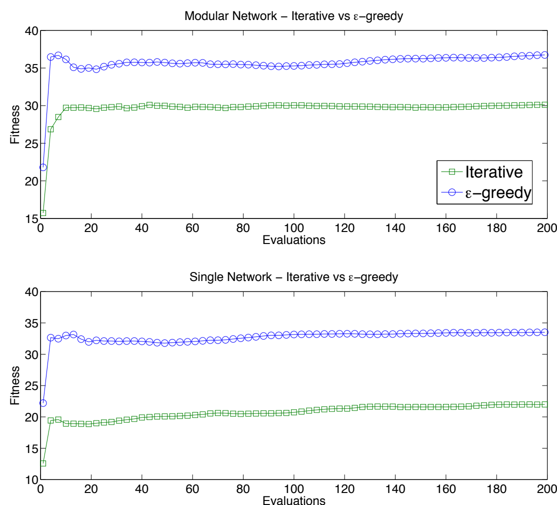


Fig. 6. The fitness values for ϵ-greedy vs. iterative selection for Multi and Singular networks

guide the agent toward desired behaviors. The quick convergence of this modular approach will complement the human interaction well, allowing the agents to quickly learn from the human participants. Finally, the promising results given by ϵ-greedy selection demonstrates that on-line evolutionary computation for game agent control is not only possible, but can prove to be more effective.

The results of this application are very encouraging, and indicate that the modular approach combined with intelligent selection mechanisms 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.

A. Future Work

Future work on this project will include more detailed experiments to show how well the agents perform compared to scripted agents, as well as performance against human players. Also experiments will be carried out to show the benefits of the human interaction, and the value of online storage allowing many people to be involved in the evolutionary process.

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 is planned 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

This work is a result of the AMALTHEA NSF REU program. This program is a 10 week research experience for undergraduates interested in Machine Learning. For more information visit http://cygnus.fit.edu/amalthea. This work was supported in part by the NSF grants: 0647018, 0717674, 0717680, 0647120, 0525429, and 0806931

REFERENCES

[1] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. Vol. 2, no. No. 10, pp. 99–127, 2002.
[2] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the NERO video game," *IEEE Transactions on Evolutionary Computation*, vol. Vol. 9, no. No. 6, December 2005.
[3] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," *Journal of Machine Learning Research*, vol. Vol. 7, pp. 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] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. Vol. 4, pp. 237–285, May 1996.
[8] J. Reisinger, K. O. Stanley, and R. Miikkulainen, "Evolving reusable neural modules," *Springer-Verlag Berlin Heidelberg*, pp. 69–81, 2004.
[9] H. Takagi, "Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation," *IEEE*, 2001.