The Incremental Evolution of Attack Agents in Xpilot

Gary B. Parker, *Computer Science, Connecticut College,* parker@conncoll.edu
Matt Parker, *Computer Science, Indiana University,* matparke@indiana.edu

Abstract— **In the research presented in this paper, we use incremental evolution to learn multifaceted neural network (NN) controllers for agents operating in the space game Xpilot. Behavioral components specific to the accomplishment of specific tasks, such as bullet-dodging, shooting, and closing on an enemy, are learned in the first increment. These behavioral components are used in the second increment to evolve a NN that prioritizes the output of a two-layer NN depending on that agent's current situation.**

I. Introduction

In previous work we introduced Xpilot as an environment for testing systems designed for learning control programs for autonomous agents. In that paper, we presented initial experiments where a genetic algorithm (GA) was used to learn the connection weights for a single-layer artificial neural network (NN) controller. The tests demonstrated the use of the Xpilot environment, but the controllers evolved lacked the sophistication required to compete with a human opponent. In this paper, we report research conducted to take the next step in our progression toward evolving a truly competitive Xpilot agent. We evolve single-layer NNs that are each designed to handle a specific task. These NNs are then used as part of a larger NN with their outputs combined to produce multifaceted agent behavior.

Xpilot is an open-source 2-dimensional space combat simulator which is playable over the internet. Multiple players can connect to a central Xpilot server and compete in many varieties of game play, such as free-for-all combat, capture-the-flag, or team combat. Each player controls a space-ship that can turn, thrust, and shoot. There is often a variety of weapons and ship upgrades available on the particular map in which they play. The game uses synchronized client/server networking to allow for solid network play.

There are a number of researchers who have used GAs to evolve game playing agents. Most of this work has been in that area of thought games where the agent is competing with a single opponent (board games, etc.). Konidaris, Shell, and Oren worked to evolve a NN to capture in Go [1]. Hingston and Kendall used evolution in the iterated prisoner's dilemma problem [2], and Fogel researched learning in checkers [3]. Some research has been done in the area of action computer games. Funes and Pollack evolved controllers for light-cycles against human opponents in their online Java Tron applet [4], Yannakakis and Hallam evolved interesting ghost opponents for the game Pac-Man [5], and Cole, Louis, and Miles evolved agent parameters for the multiplayer first person shooter, Counter-Strike [6].

In previous work, we used evolutionary computation to learn controllers for agents operating in Xpilot [7]. A single-layer NN with 22 inputs and three outputs (thrust, shoot, and turn) was developed and a GA was used to learn the connection weights. The evolved controllers learned to survive and fight against an enemy bot in a simple square arena. Instead of learning the best behavior from a particular starting location, each generation switched starting locations to make the agent more skilled at general combat. Because the hostility of the starting locations varied, graphing improvement in fitness was difficult, yet it could clearly be seen by looking at the average fitness of the populations that the agents had successful evolved against the particular enemy bot used in the evolution.

The results were promising in that the Xpilot environment showed significant potential for future work, and the evolved controllers showed progress as they gained survival fitness over training time. However, the resultant agents lacked many skills needed for successful combat. In this paper, we report the results of our research in which we used incremental evolution to learn controllers for Xpilot agents. In the first increment we used specific training environments to learn specific facets of control. These were then used in the second increment to evolve a two layer NN that used a separate NN to control its second layer connection weights.

II. Modifications to Xpilot

In previous work, we described the modifications made to Xpilot to create a system for testing artificial intelligence (AI) agent learning systems. In this section those modifications will be reviewed and new changes discussed. The Xpilot client, which a player uses to join with an Xpilot server, is mainly used to display relevant information about the game world to the player and to capture keyboard and mouse input that it sends it to the server. Between each frame, the client receives from the server information that is needed to display the player's ship and the surroundings. We intercept this information and convert it into variables relevant for use with an AI agent. We also simulate keyboard strokes and mouse movement to control the ship. Xpilot was originally coded in C, which is a difficult language for AI modules because it is necessary to recompile the entire program after making even small

changes. We have now added a Scheme interface to make writing AI agents for Xpilot much more convenient. This allows us to modify our controllers without compiling and even alter agent behaviors while Xpilot is running.

III. Evolution of Behavioral Components

The first increment in the development of combat Xpilot agents was to evolve behavioral components. The Xpilot agent operates in a complex combat environment. In order to successfully engage the enemy; the AI agent needs to be able to locate, move towards, and track the opponent; fire bullets at it; and dodge bullets from it. Each of these behavioral components takes specific skills. In order to equip our agent with these skills, specialized NNs for control and training environments for learning were developed. The NN connection weights were learned to develop controllers appropriate for the specific tasks. The three possible outputs are thrust (on/off), shoot (on/off), and turn (between -15° and 15°).

A. Bullet Dodging

We determined that a key skill in agent survival is its ability to dodge bullets. This is a very difficult task that advanced human players accomplish through experience. We wanted our Xpilot agent to learn through training in an environment where bullet dodging was isolated as the key to survival. We used a single-layer NN that had the inputs that we considered to be of greatest importance. The inputs used were changed as necessary through and an iteration of tests until a good compromise between required inputs and chromosome length was found. The inputs included the agent's velocity, the difference in the agent's heading from its track, shot-alert (a function that computed the danger of bullets in the vicinity), and the difference in the agent's track in comparison to the most dangerous bullet's track. The outputs were turn and thrust. Shoot was not included since this action is not needed for bullet dodging.

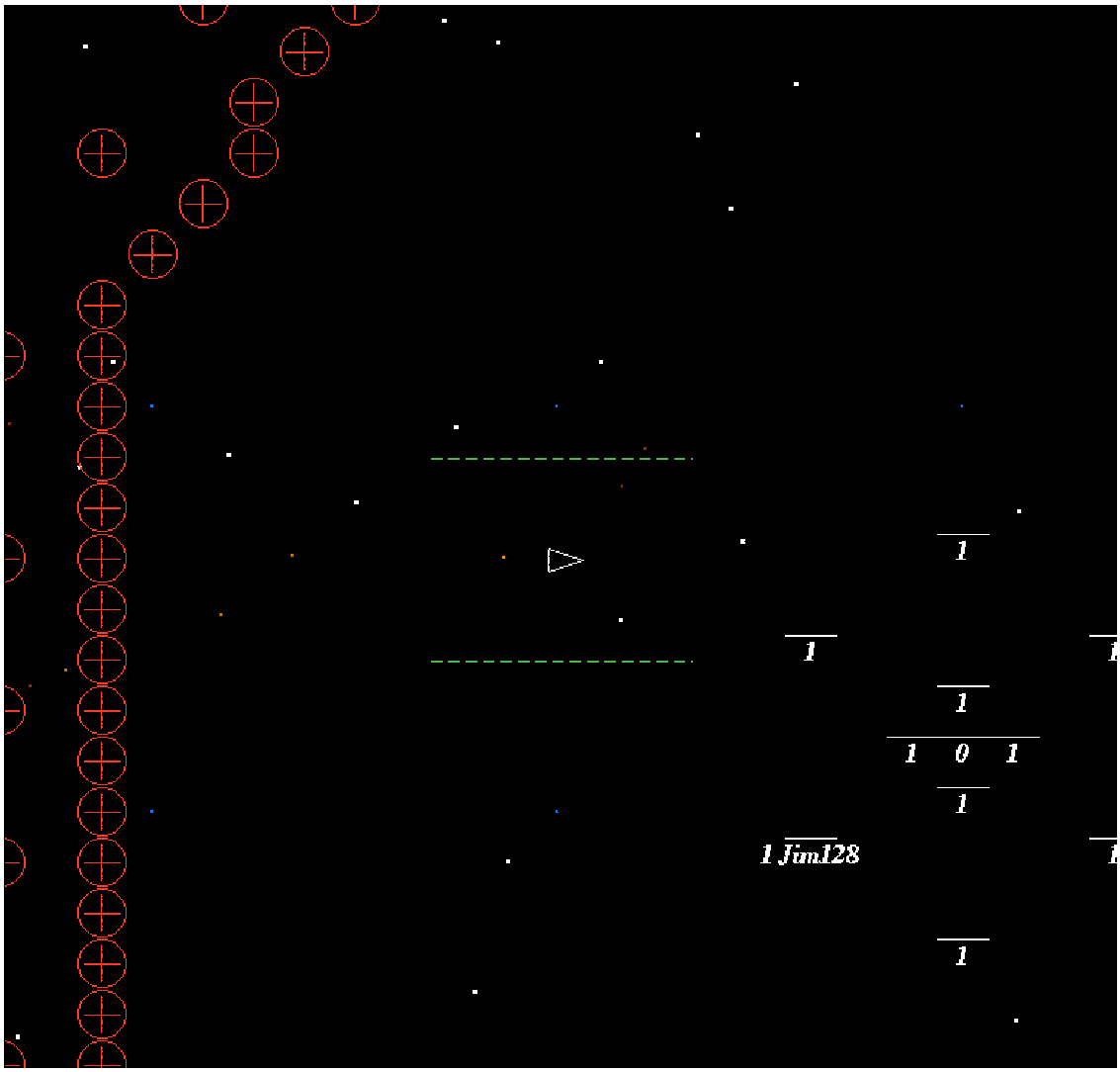

Fig. 1. Training arena for the bullet dodging behavior component. This is a screenshot of the map. Jim128 is navigating through a barrage of bullets (larger white dots). The label "1 Jim128"shows the agents starting base, which is randomly selected from the team 1 starting basis. The agent is shown as a triangle; in this case it is currently heading east. The "+" circles are repellers, which constrain the agent to the center of the map.

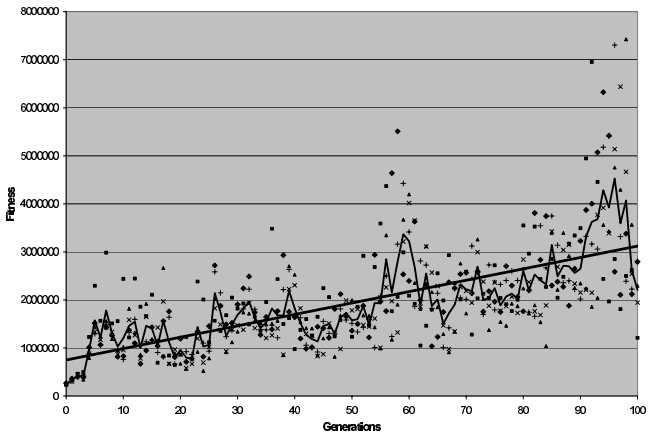

Fig. 2. Graph of learning curve for bullet dodging. The points are the average fitness of each population at each generation. The line shows the mean of the five runs. The bold line shows linear least squares trend line.

The agent was trained in an environment (Figure 1) that was focused on bullet dodging with minimal need for wall avoidance and interaction with enemy ships. The agent was placed in the center area of this arena with a perimeter of repeller nodes (circles marked with a +). These nodes repel ships in their vicinity. This perimeter was designed to keep the agent isolated in the center of the arena. A second circle of repellers surrounded the first, and enemy ships were placed outside of the second circle. They were programmed to shoot freely at the AI agent. The two layers of repellers kept, for the most part, the enemy ships separate from the AI agent's ship while a flurry of enemy bullets peppered the area where the AI agent could operate. The AI agent's fitness was determined by how many frames it survived. In order to lessen the effects of luck, three tests were done for each individual at each generation to determine its fitness.

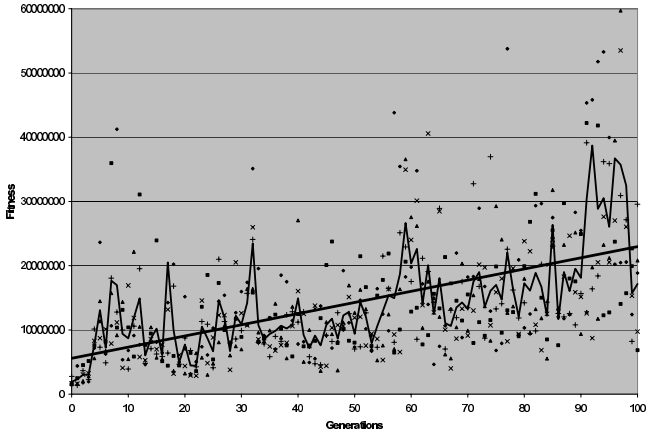

Fig. 3. Graph of learning curve for bullet dodging. The points are the best individual fitness from each population at each generation. The line shows the mean of the five runs. The bold line shows linear least squares trend line.

Figures 2 and 3 show the fitness growth as five tests were run for 100 generations each. The graph shows points for the five runs plus an average line showing the average at each generation. As can be observed, there was a high degree of variation throughout the learning process. This is primarily due to the fact that luck plays a large part in bullet dodging. However, steady improvement can be seen as the agents learn the specific task. A least squares line is provided to help the reader visualize the improvement. Observations of the agents in the arena also confirmed that they learned techniques for bullet evasion. They would thrust away from bullets heading in their direction and would turn if their heading was pointed in the direction of (or directly opposite to) the approaching bullet.

B. Shooting

Another important skill for an Xpilot agent is to be able to destroy the enemy. Since the only means in our simulation for it to destroy the enemy without destroying itself was to shoot it, we set up a training environment for this purpose (Figure 4). We did not want the agent to be concerned with avoiding walls, so contact with walls does not damage the ship. In addition, we made collisions with the enemy harmless to both ships so that our agent would disregard potential collisions with the enemy. Both the AI agent and the enemy were placed within the arena with the agent placed in the center and the enemy placed at any of the number of bases near the perimeter. The enemy could not shoot and stayed in place until a bullet approached. It would take action to avoid the bullet and then continue moving in an attempt to survive. The AI agent had no thrust so its task was to use turn and shoot to destroy the enemy.

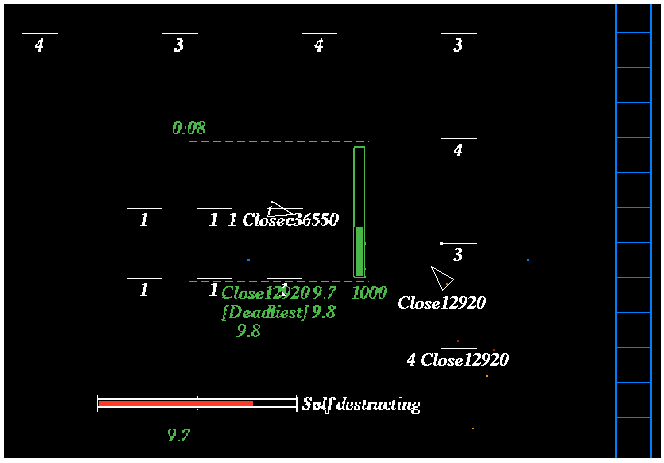

Fig. 4. Training arena for the shooting behavior component. The agent (Closec36550) is shown firing at enemy ship (Close12920). The arena is not much larger than the viewable area in this screenshot. The agent starts at a randomly selected 1 base. The enemy switches between the 3 and 4 bases. Its base for this run was one of the 4s located directly below it. The bar to the right of the agent shows its fuel remaining. However, it will never last long enough to burn the fuel since self-destruct has been activated. Self destruct is used as a means to enforce the time limit.

The input the AI agent used was the difference of its heading to the direction of the enemy, the history of this difference in the previous frame, the difference of agent's track to the direction of the enemy, the distance to the enemy, the difference between its own track and heading, its velocity, and a node always set to one to act as the threshold. These seven values are all inputs to a single-layer NN that

has turn and shoot as outputs. The output value of turn was multiplied by 15 yielding a resultant turn (heading change) between -15° and 15° (15° is the maximum turn rate). We impose a maximum turn rate of 15° on our agents to better simulate reasonable human play. The shoot node caused the agent to shoot if its resulting value was zero or above. The weights for this NN were evolved using a GA. Each weight could have a value between -1 and 1, which was converted from an 8 bit number; 14 of these made up the chromosome.

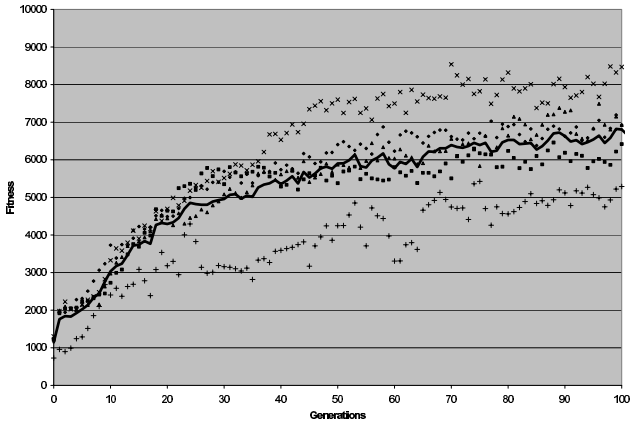

Fig. 5. Graph of learning curve for shooting. The points are the average fitness of each population at each generation. The line shows the mean of the five runs.

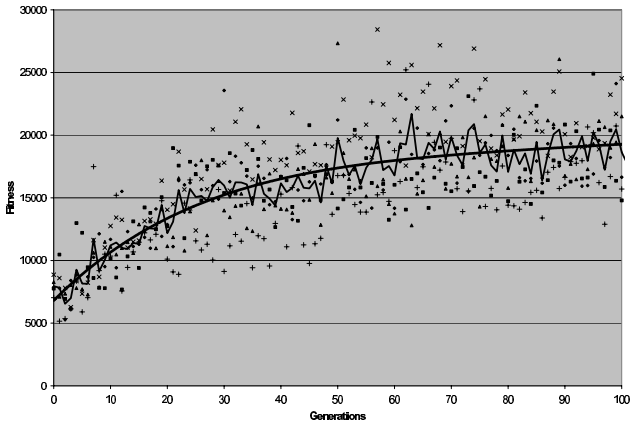

Fig. 6. Graph of learning curve for shooting. The points are the best individual fitness from each population at each generation. The line shows the mean of the five runs. The bold line shows a 6th order polynomial least squares trend line.

The tests yielded favorable results. Five tests were run with the average and best fitnesses at each generation shown in Figures 5 and 6. The fitness was determined by how close the bullets approached the enemy with a bonus added for a kill (each individual was run 3 times). The graphs show increased fitness growth through 100 generations. A mean line is displayed on the graph showing the average fitness. There is much more variation from generation to generation for the best of each population. This graph shows a means line and a least squares 6th order polynomial is also shown. Observation of the agents revealed that they had learned to turn toward the enemy and shoot. Some individuals appeared to be using the difference in heading to the ship from the previous frame (one of the input nodes) to predict the future position of the enemy and lead it with a shot. The shooting agents were successful in their task.

C. Close

The final task for the agent to learn involved two skills: the agent was to close on an enemy to attack and avoid flying into the lethal walls while navigating toward the ship. The arena for this is shown in Figure 7. The AI agent always started someplace in the center of the arena and the stationery enemy was placed alternately at the top or bottom of the arena.

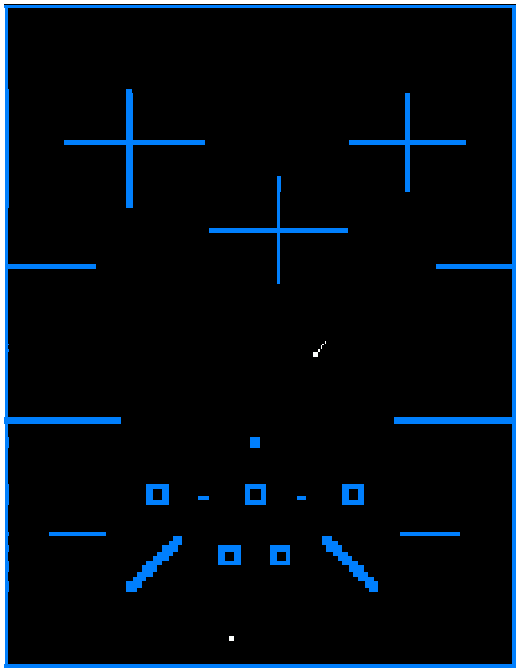

Fig 7: Training arena for the close behavior component. The agent, represented as a white dot with a heading indicator, is located in center. It must attempt to reach the enemy ship, the white dot in the southern extreme of the map. The enemy ship cycles between two random locations, one at the extreme north and the other at the extreme south of the map.

The fitness of the agent was determined by its continual closure on the enemy with a bonus for each frame it stayed within 35 pixels of it (two runs were done for each individual). The NN had 16 inputs going to two outputs (thrust and turn). The inputs dealt with wall locations relative to 15° left and right of the agent's track and the enemy ship's bearing relative to the agent's heading and track. Figures 8 and 9 show the learning curve through 100 generations. The GA learned the weights needed for the agent to find its way to the target going both in the north and south directions. Observations of the agent revealed a peculiar behavior; it would spin as it advanced toward the enemy. At first, it seemed to make little sense because spinning required extra coordination since thrusts had to be timed to occur when the heading was facing in the correct direction. After some consideration, it was speculated that

the spinning helped in avoiding death through wall collisions. In the default Xpilot settings, a ship is more sensitive to wall collisions on its bow than on its stern. The spinning reduced the probability that initial contact with a wall would be in the front.

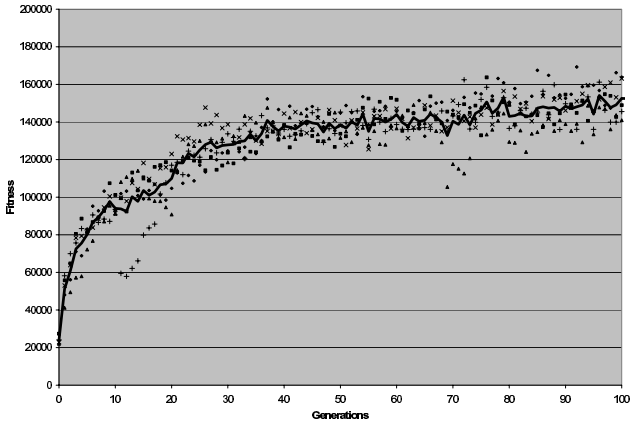

Fig. 8. Graph of learning curve for close. The points are the average fitness of each population at each generation. The line shows the mean of the five runs.

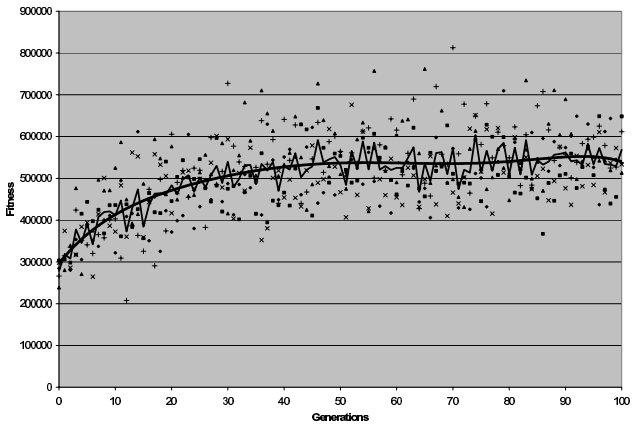

Fig. 9. Graph of learning curve for close. The points are the best individual fitness from each population at each generation. The line shows the mean of the five runs. The bold line shows a 6th order polynomial least squares trend line.

IV. Evolution of Control from the Components

The second increment in the evolution was to use the components evolved in the first increment to evolve a complete controller. We had three good components evolved to handle three tasks: dodge bullets, shoot the enemy, and close on the enemy while avoiding walls. Joining these together was the next step. The obvious method would be to use the best controllers from each of the three components as the first layer of a NN and evolve a second layer. The outputs of the three components were turn and thrust for bullet dodging, turn and shoot for shooting, and turn and thrust for close. This would give us three middle layer turn nodes and two middle layer thrust nodes. The shoot node (from shooting) could go directly to the output. The problem with this method had to do with the combining of middle layer nodes. For example, combining the best turn for dodging, with the best turn for shooting, with the best turn for closing, was very unlikely to result in the desired behavior without some other input.

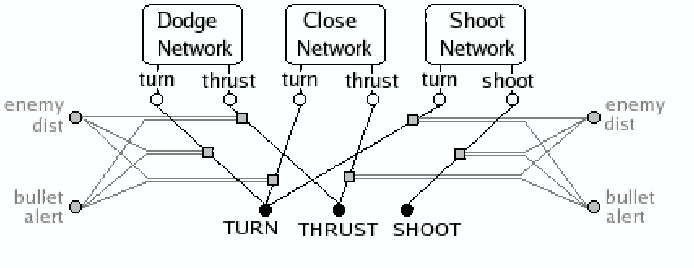

Fig. 10. Diagram of entire incremental network. The enemy distance and bullet alert are inputs to a network whose outputs are the weights between the specialized networks' outputs and the three resultant outputs.

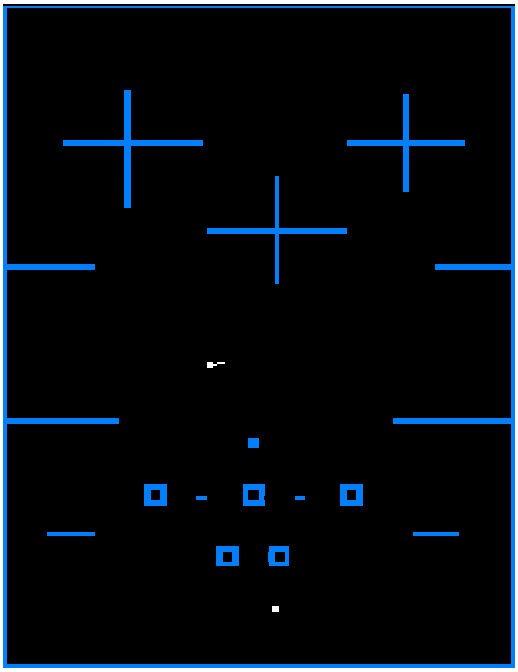

Fig. 11. Training arena for the final increment of evolution, in which the priority network was evolved. The map is very similar to Close, but it has fewer obstacles and the enemy only starts in the south.

We reasoned that what was really needed was a NN that determined the priority of these three components and combine them with that in mind. This meant that we needed a separate NN to set the weights for the second layer of our two-layer NN. In other words, the weights of the connections of the second layer needed to change depending on the environment. The two primary conditions to be considered were the risk of being killed by a bullet and the distance from the enemy ship. Bullet risk should give dodging the priority, a close ship should give shooting the priority, and a far ship should give close the priority. We used a formula to determine the risk of the bullets; this included their distance, their closing velocity, and collision probability assuming that the agent maintained its current track and speed. Using these two inputs (bullet risk and

enemy distance) as input nodes, we used a GA to learn the weights of a separate NN that set the weights between the middle and output layer of the control NN.

A drawing of the NN is shown in Figure 10. These two inputs, through the separate NN, set the weights for the two-layer control NN. This separate network required that twelve weights be learned. Each of these had a value between -1.0 and 1.0 and was represented by eight bits. In addition, the GA learned the threshold weights for the three output notes (thrust, turn, & shoot) making a chromosome that consisted of fifteen eight bit numbers.

The fitness function rewarded aggressive behavior by giving points for being closer to the enemy and for killing the enemy. Survival was not directly rewarded, although the run ended when the ship was destroyed, so no further near / kill enemy points could be accumulated. The tests were done in an arena similar to one used for the close test (Figure 11). Except that some wall obstacles were removed and the enemy always started in the lower section.

For these tests, the enemy was a hand-coded attack bot that remained stationary until there were no walls between it and the learning agent, at which time it would attack. Figures 12 and 13 show the results of five tests run for 100 generations (3 run for each individual). As can be seen, the GA continually improved the control program, which resulted in good agent behavior.

Observations of the resultant agents revealed that they had all improved in their effectiveness, but varied in their strategies. In most cases they spun as they closed on the enemy, firing sporadically. The spin often continued as they approached the enemy, but the firing rate increased. Some individuals would tend more toward shifting into a shoot mode when close to the enemy, turning and firing without using thrust. None of the learned bullet dodging behavior was observed. Although in some cases the agent would have an irregular spin pattern when a bullet approached. This resulted in less likelihood of bullet contact, but the behavior appeared to be random as opposed to motions appropriate for the situation. In all cases the tests resulted in aggressive agents who engaged the enemy and attempted to destroy it.

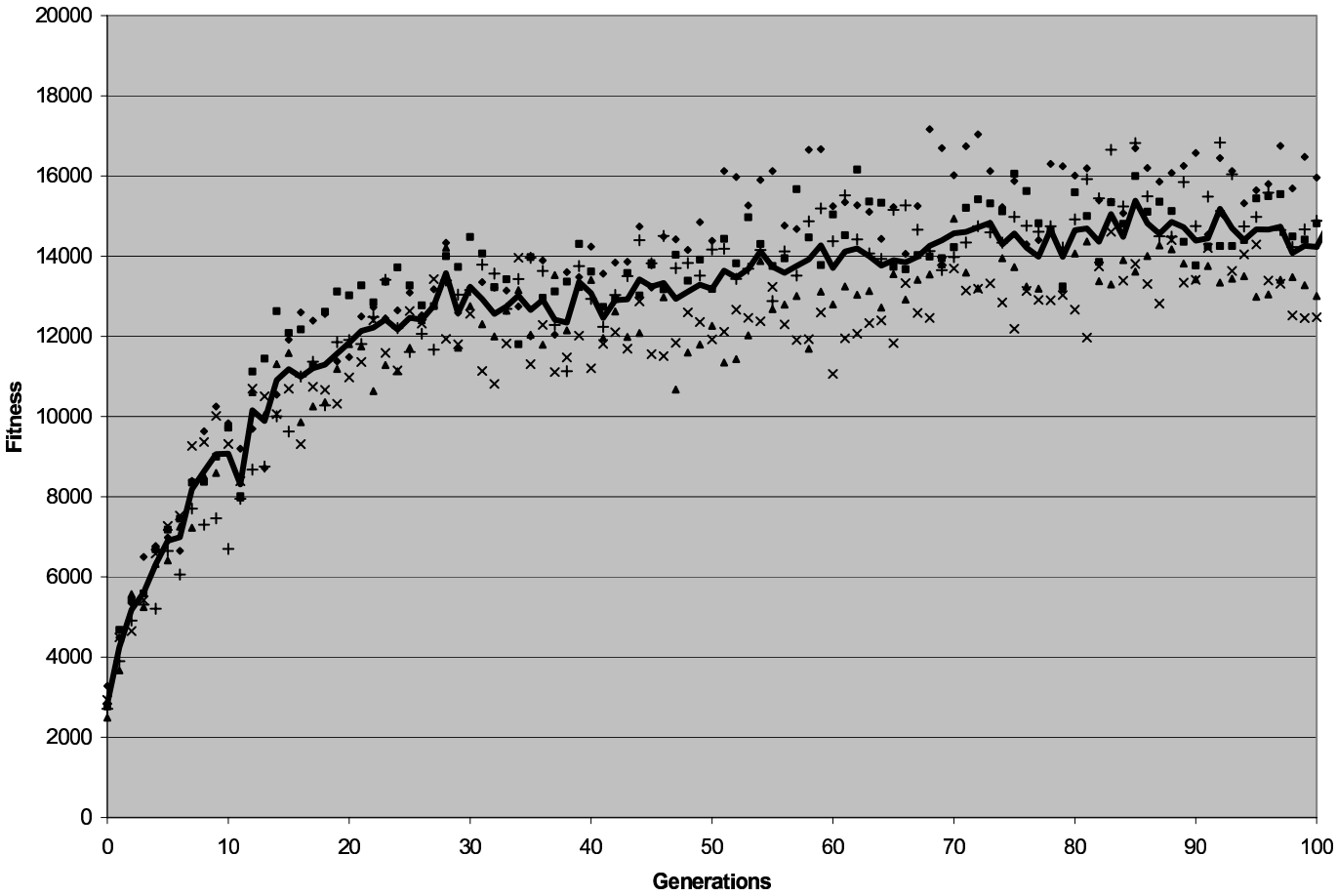

Fig. 12. Graph of learning curve for the second increment NN. The points are the average fitness of each population at each generation. The line shows the mean of the five runs.

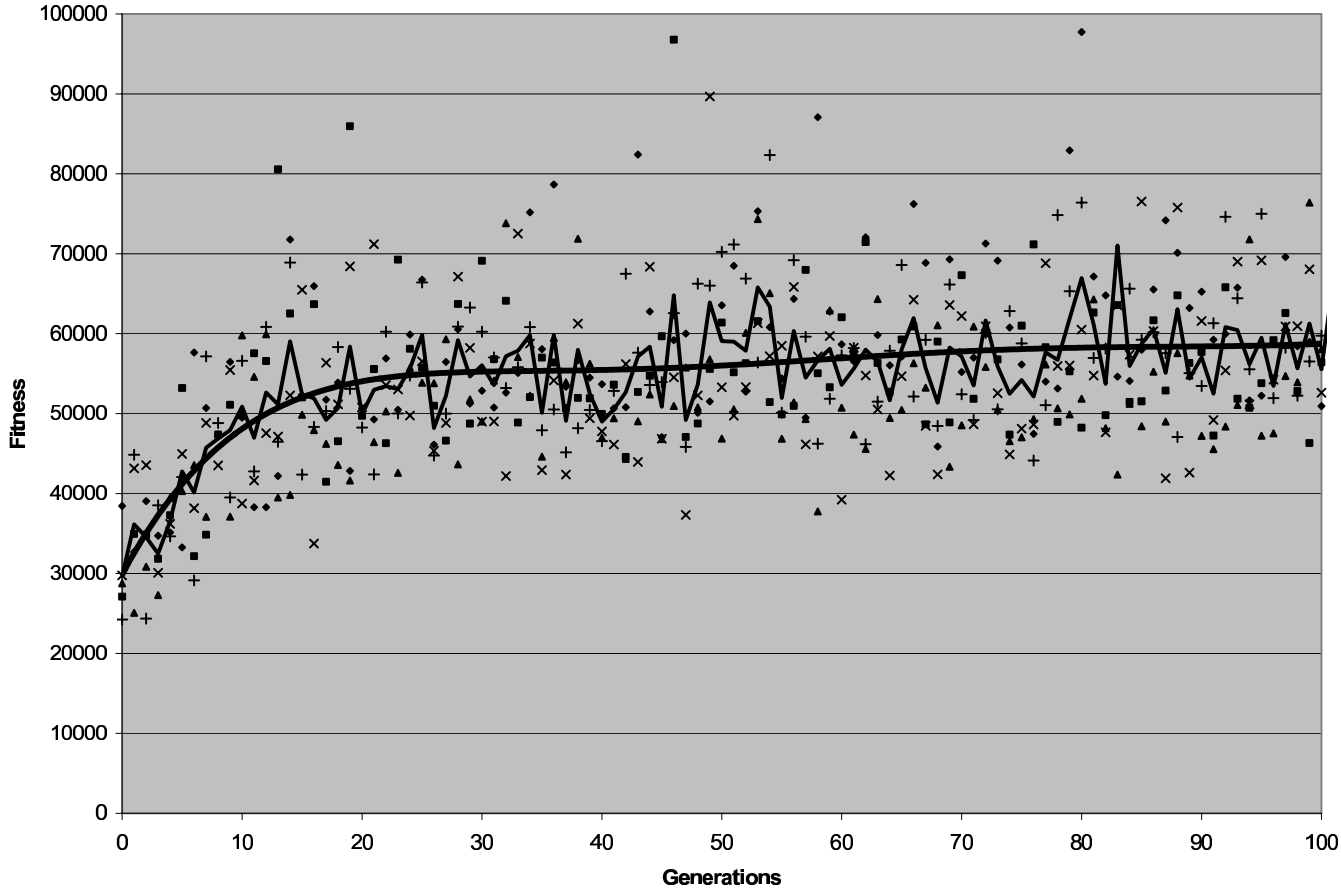

Fig. 13. Graph of learning curve for the second increment NN. The points are the best individual fitness from each population at each generation. The line shows the mean of the five runs. The bold line shows a 6th order polynomial least squares trend line.

V. Conclusions

This research uses incremental evolution to continue to advance our work in evolving NN controllers for agents operating in the Xpilot combat environment. The first increment was to learn NN controllers that were evolved to be proficient in specific tasks. The second increment used the outputs of these controllers, plus a separate NN to assign them varying degrees of priority to create a two-layer NN that was responsive to changes in the agent's combat conditions. This controller produced an output that took all three learned responses into consideration, giving priority to the one most appropriate for the current situation.

In future work, we will continue to use incremental learning to evolve behaviors appropriate for specific situations pertinent to space combat. These will be used to increase the complexity of controllers for Xpilot agents. For example, adding a behavioral component which had learned the proper behavior for moving away from an enemy would allow the agent to retreat after a failed attack or in situations where multiple enemies were present. In addition, components could be learned that help the agents participate in coordinated efforts such as teams of agents working together to complete an assigned task.

References

[1] Konidaris, G., Shell, D., and Oren, N. "Evolving Neural Networks for the Capture Game," Proceedings of the SAICSIT Postgraduate Symposium, Port Elizabeth, South Africa, September 2002.

[2] Hingston, P. and Kendall, G. "Learning versus Evolution in Iterated Prisoner's Dilemma," Proceedings of the International Congress on Evolutionary Computation 2004 (CEC'04), Portland, Oregon, 20-23 June 2004, pp 364-372.

[3] Fogel, D. Blondie24: Playing at the Edge of AI, Morgan Kaufmann Publishers, Inc., San Francisco, CA., 2002.

[4] Funes, P. and Pollack, J. "Measuring Progress in Coevolutionary Competition," From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior. 2000, pp 450-459.

[5] Yannakakis, G. and Hallam, J. "Evolving Opponents for Interesting Interactive Computer Games," Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04); From Animals to Animats 8, 2004, pp 499-508.

[6] Cole, N., Louis, S., and Miles, C. "Using a Genetic Algorithm to Tune First-Person Shooter Bots," Proceedings of the International Congress on Evolutionary Computation 2004 (CEC'04), Portland, Oregon, 2004, pp 139–145.

[7] Parker, G., Parker, M., and Johnson, S. "Evolving Autonomous Agent Control in the Xpilot Environment," Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005), Edinburgh, UK., September 2005.