The Evolution of Multi-Layer Neural Networks for the Control of Xpilot Agents

Matt Parker
Computer Science
Indiana University
Bloomington, IN, USA
matparker@cs.indiana.edu

Gary B. Parker
Computer Science
Connecticut College
New London, CT 06320
parker@conncoll.edu

Abstract— Learning controllers for the space combat game Xpilot is a difficult problem. Using evolutionary computation to evolve the weights for a neural network could create an effective/adaptive controller that does not require extensive programmer input. Previous attempts have been successful in that the controlled agents were transformed from aimless wanderers into interactive agents, but these methods have not resulted in controllers that are competitive with those learned using other methods. In this paper, we present a neural network learning method that uses a genetic algorithm to select the network inputs and node thresholds, along with connection weights, to evolve competitive Xpilot agents.

Keywords: Xpilot, Genetic Algorithm, Neural Network, Control, Autonomous Agent, Xpilot-AI

I. INTRODUCTION

In previous research, we used a genetic algorithm to evolve the weights for a neural network in the game Xpilot. In one experiment, we evolved a single layer perceptron network in a simple map against one opponent [1]. In the second experiment, we evolved three separate specialized networks; one to shoot, one to dodge bullets, and one to fly toward the enemy. These specialized networks were combined into a larger network in an attempt to create a good general combat robot [2]. The controllers evolved in these previous papers used a simple weight system and the inputs to the network were chosen by the researchers. In the research reported in this paper, we evolve a two-layer neural network without incrementally evolving specialized networks. The inputs to the neural network are selected by the GA from a large list of possibilities. In addition, we changed the weight system to include thresholds and inverted thresholds between every connected node, which allowed for more decisive behavior.

Most of the previous work done by other researchers in the area of evolving game-playing has been done with thought games, such as board games, where the agent competes against a single opponent. Work was done by Konidaris, Shell, and Oren to evolve a neural network to capture in the game Go [3]. Fogel conducted his famous research with Checkers [4]. Hingston and Kendall did research to solve the iterated prisoner's dilemma problem [5]. In the area of action computer games, Funes and Pollack created their Java Tron applet, which evolved controllers for light-cycles against human opponents [6]. Cole, Louis, and Miles evolved robot parameters for the 3D first person shooter game Counterstrike [7], Hallam and Yannakakis evolved "fun" ghost opponents for the game Pac-Man [8], Stanley, Bryant, and Miikkulainen evolved neural networks to control agents that could learn in real-time through a series of training exercises in the NERO video game [9], Priesterjahn, Kramer, Weimer, and Goebels evolved controllers for artificial players in the game Quake3 [10], and Miles and Louis evolved game playing strategies for opponents in a game that they created called Lagoon [11].

In previous work, we evolved controllers for Xpilot using a cyclic genetic algorithm (CGA) [12]. While these were our most successful controllers, they required a large amount of intelligent design on the part of the researcher, and lacked the variation of output of a neural network. In the research reported in this paper, we evolve a neural network that uses a new weight system and evolved inputs, which is comparable in skill to the CGA controller, but without any predefined behaviors and with a larger variety of possible actions.

II. XPILOT

Xpilot is a 2-dimensional multiplayer space combat game playable over a network and the internet. The player controls a space ship which can mainly either thrust, turn, or shoot. The game physics have a realistic feel with an accurate representation of acceleration, velocity, and momentum in a frictionless space environment. Though the few control keys are simple to learn, a good "feel" for the physics is required to skillfully pilot the ship, shoot opponents, and avoid their shots.

The standard versions of Xpilot include a server controlled robot with a respectably good artificially intelligent controller. There was no interface provided to allow people to reprogram the server robot, and few ever did. Recently, a group of researchers developed an easy-to-use interface for using AI to control a player's ship in Xpilot which is called Xpilot-AI (www.xpilot-ai.org). Because Xpilot is open-source, they were able to modify the Xpilot client, making new functions to control the ship and to read variables about the surrounding environment. Because it is a modification of the client, these AI controlled ships are able to connect to any Xpilot server and play along with other AIs and human players.

Xpilot is a game with few controls that requires complex behavior to successfully pilot the ship, so it is naturally a good environment to test neural network controllers, which usually take several inputs and produce a few outputs.

III. NEURAL NETWORK CONTROLLER

A. *Inputs*

Choosing inputs that are valuable to the neural network is difficult because we do not know what the neural network needs to produce the desired behavior. In past research, we chose whatever inputs we thought were required and then perhaps added a few more that might be valuable, usually creating a set of inputs more numerous than needed. Conversely, we might choose too few, not including an input because it did not seem necessary to us, and yet to a neural network it might actually be useful.

In this experiment, we chose to evolve which inputs to use in the network, rather than choosing them ourselves. We created a list of 64 possible inputs which covered a broad range of the possible inputs in the game; most of them seemingly useful, and some that did not seem to us to be particularly so, yet were included in the off chance that the network could use them. These were selected to be useful in combat against any Xpilot opponent as opposed to being modeled specifically for a known enemy. There were two main types of inputs, with about 32 of each type. The value range of the inputs was normalized by reducing it to a value between -1.0 and 1.0.

B. *Direct Input*

Variables that are directly read from the game environment are the "direct" inputs. For example, several of our direct inputs are: Self velocity, Enemy velocity, Enemy distance, Bullet intercept distance, Bullet intercept time, Wall distance directly in front of ship, etc., as well a few unchanging inputs, such as 1.0, 0.0, and -1.0.

C. *Comparison Input*

The comparison input compares two angles and returns their difference. The difference between the ship's heading (direction it is pointing), and another angle, perhaps the direction to the enemy ship, is the number of degrees that the ship should turn to be pointing at that angle. The difference between the ship's track (direction of velocity) and some other angle can reveal if the ship is flying in or close to that particular direction, which is helpful for flying towards or avoiding objects. We have several difference comparisons between the self ship's track or heading and other angles, such as the Enemy's target direction, the Bullet's predicted nearest intercept angle, the Enemy's track, the Bullet's track, and so on.

Each neural network input node was represented as a 6 bit gene, which was converted to a number between 0 and 63 and matched with the corresponding input in the list of inputs. We determined that 128 possible inputs were unnecessary and we did not want to increase the gene size to 7 bits, so we kept with 64, although a few more inputs could have been useful. For example, we wanted to have a "wall feeler" type input, which would detect walls within a certain range at an evolved angle from the ship. However, because we had only 64 possible inputs, we only included wall feelers at 6 different angles, with two different ranges each: at +10/-10 degrees and +30/-30 degrees from the ship's velocity direction, and at +0 and +180 degrees from the ship's heading.

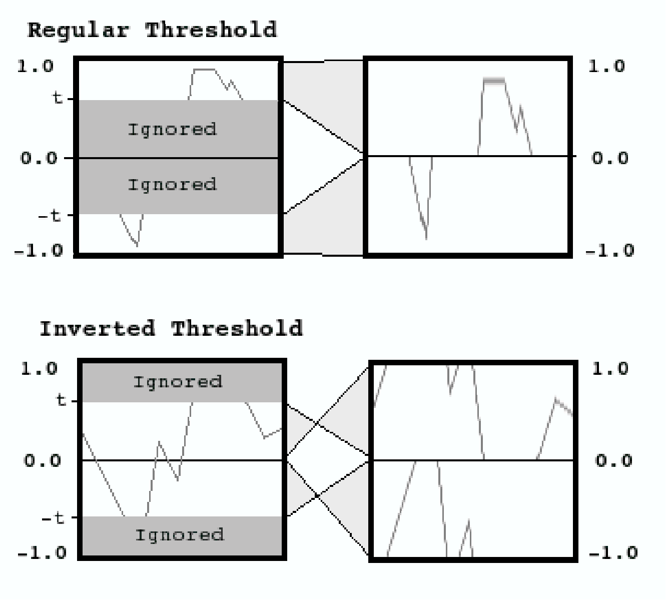

Fig. 1. A threshold is on every weight. If it is a regular threshold, every input value between -t and t is ignored, and the allowed values are amplified. If it is an inverted threshold, the values above t or below -t are ignored, and the allowed values are amplified and inverted.

D. *Weights*

In previous work [1,2], our neural networks had a simple system of weights with only one threshold per output node, where an input would simply be multiplied by a weight which was an evolved number between -1.0 and 1.0. There are a few problems with this method. One is that often times it may be beneficial for the controller to have neurons that perform no action when their input is below a certain level. For instance, if a bullet is far enough away, the ship might wish to completely ignore it, and yet it may need to perform drastic maneuvers when a bullet is too close. With our simple multiplied weight of the past, the input is linearly affected by the weight, so that a harmless bullet that is far away still influenced the behavior of the ship unnecessarily, especially if the neuron required a high output when the bullet was dangerous. Because all the neurons constantly affected the output of the network, the ships always developed a spinning behavior which was a blend of necessary movements, rather than separate behaviors which depended on different environmental situations.

Another problem with the old weights is that we always had to determine whether or not we should invert the input to the weight. For instance, the shorter the distance that a bullet is from the ship, the more action should be taken by the network. If we leave the input as just the distance to the bullet, when the bullet is nearest it is at 0.0, and farthest it is at 1.0. So, we would determine to invert the input before it goes into the weight. While this probably was a good idea for the bullet distance, for other inputs it's not so clear. Therefore we needed a way for the network to choose whether or not to invert it without our intervention.

We have solved both these problems with our current system of weights. Each weight is represented by two 6 bit genes. The first gene determines the threshold (Fig. 1), with the first bit determining if it is an inverted threshold and the last 5 bits determining the value of the threshold. Because the input to the weight can be anywhere between -1.0 and 1.0, the value of a threshold is a number converted from the 5 bits to a number between 0.0 and 1.0. If the value of a threshold is t and the first bit determines it is a regular threshold, then any input values between $-t$ and t are ignored, and the weight will output zero. If the input value is greater than t or less than $-t$, then that input value, V, is amplified: $newV = V * 1.0 / (1.0 - t)$

If it is an inverted threshold, then any values greater than t or less then $-t$ are ignored. Values between $-t$ and t are inverted and amplified: $newV = 1.0 / (V * 1.0 / t)$. We invert the values here because if not, the value of the weight would grow greater as the input approached the threshold, so that the greatest action would occur right before choosing to ignore the input. Normally, however, greater action should occur as the input gets further away from the threshold. For instance, as the distance of the bullet decreases, more action is required; but as it gets further away, perhaps nearer to the weight's threshold, less action is required.

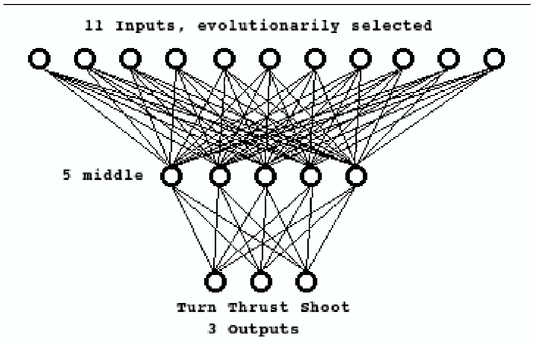

Fig. 2. The neural network is composed of 11 input, 5 middle, and three output nodes. The 11 input nodes are evolved and chosen from a list of 64 possible inputs. Between each of the connected nodes is a weight, which consists of a threshold and a multiplier.

The second gene of the weight is the multiplier, which is a number between -1.0 and 1.0. It is multiplied by whatever is output after the input is run through the threshold. This system of weights allows for more decisive behavior and control of the ship.

E. Network Structure

The neural network itself is made of 11 input nodes, 5 middle (hidden) layer neurons, and 3 outputs (Fig. 2). The inputs, as described above, are evolved and chosen from a list of 64 possible inputs. The 5 middle layer neurons exist to increase the logical ability and variation of the behavior of the ship. The three output neurons are turn, thrust, and shoot. To thrust or shoot, the corresponding output neuron must be greater than or equal to 0.0. For turning, the output is altered by an exponent of 0.15 and then multiplied by twenty degrees. The exponent of 0.15 was chosen by observing what made a good average turn speed for the initial random population.

There is a weight between each node of the network; each weight consists of two genes, 6 bits each. The entire network consisted of 140 genes, plus the 11 genes determining the inputs, for a total of 151 genes, 906 bits total.

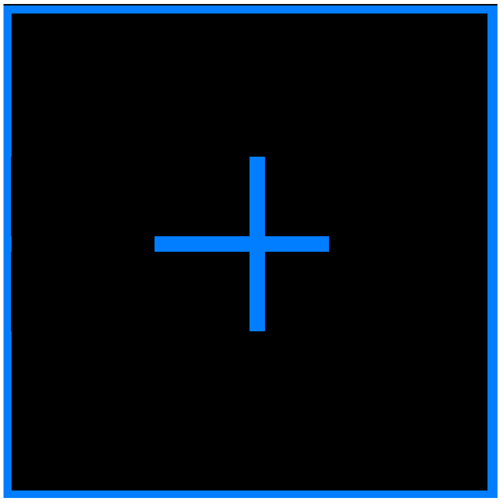

Fig. 3. Figure of the simple map used in this experiment. The starting bases are scattered about throughout the space. The map is 32x32 tiles; about 50 ship-lengths across.

IV. EVOLVING THE NEURAL NETWORK

A. Setting

We chose to evolve the neural network agent using a genetic algorithm in a setting similar to that of our previous experiments [1,12]. We used a simple square block map (Fig. 3) with an off-centered cross in the middle, and with many starting locations scattered around the map. We set inside the same opponent from our previous tests: Sel bot, who is our best hand-coded Xpilot agent. He has a good aiming function, bullet-dodging, wall avoidance, and the ability to chase enemies around walls. In previous experiments, we would reset Sel bot and the learning agent to their original locations after one of them died. This made sure that each evolving agent had the same opportunities, but added greater complexity to the evolution and neglected the importance of controlling the ship after killing an opponent, at which time the explosion from the opponent's ship can crash the agent into a wall. In this experiment, we do not make them both reset after one dies, but allow the other to keep floating around. The new agents appear at a new random starting location after every death. This is sometimes bad for the evolution, for example if Sel happens to be floating right by where a new agent appears. However, we give every agent three lives to display its fitness, so the fitter agents are still able to acquire a good fitness.

B. *Fitness*

Previously our fitness for the agents was based heavily upon staying alive [12]. This generally evolved defensive and passive agents, many who developed a constant spin and occasional thrust behavior, dodging Sel's bullets well, but not attempting to kill him. We tried with our previous neural networks to award a good fitness bonus for killing Sel, but this evolved bots that converged prematurely on a solution which just involved spinning slowly in place and shooting constantly. In this experiment we award the agent 200 points for killing Sel and 1/4 a point for each frame of game play it stays alive. While this fitness scheme would have worked poorly on our previous neural networks, probably because the neural networks themselves were so limited, it positively influences the evolution of the more complex neural network in this experiment. More aggressive agents are evolved that also attempt to stay alive.

The evolution was performed using a Queue Genetic Algorithm (QGA) [13], which essentially produces the same results as a regular genetic algorithm but is designed to be easily distributed among available computers to increase the speed of evolution. The population size was 256 individuals, but at the start was expanded to 1024 to increase diversity, then brought back down to 256. Individuals were selected stochastically (roulette wheel selection). The probability of crossover is 100% and there is a 1/300 chance of mutation per bit.

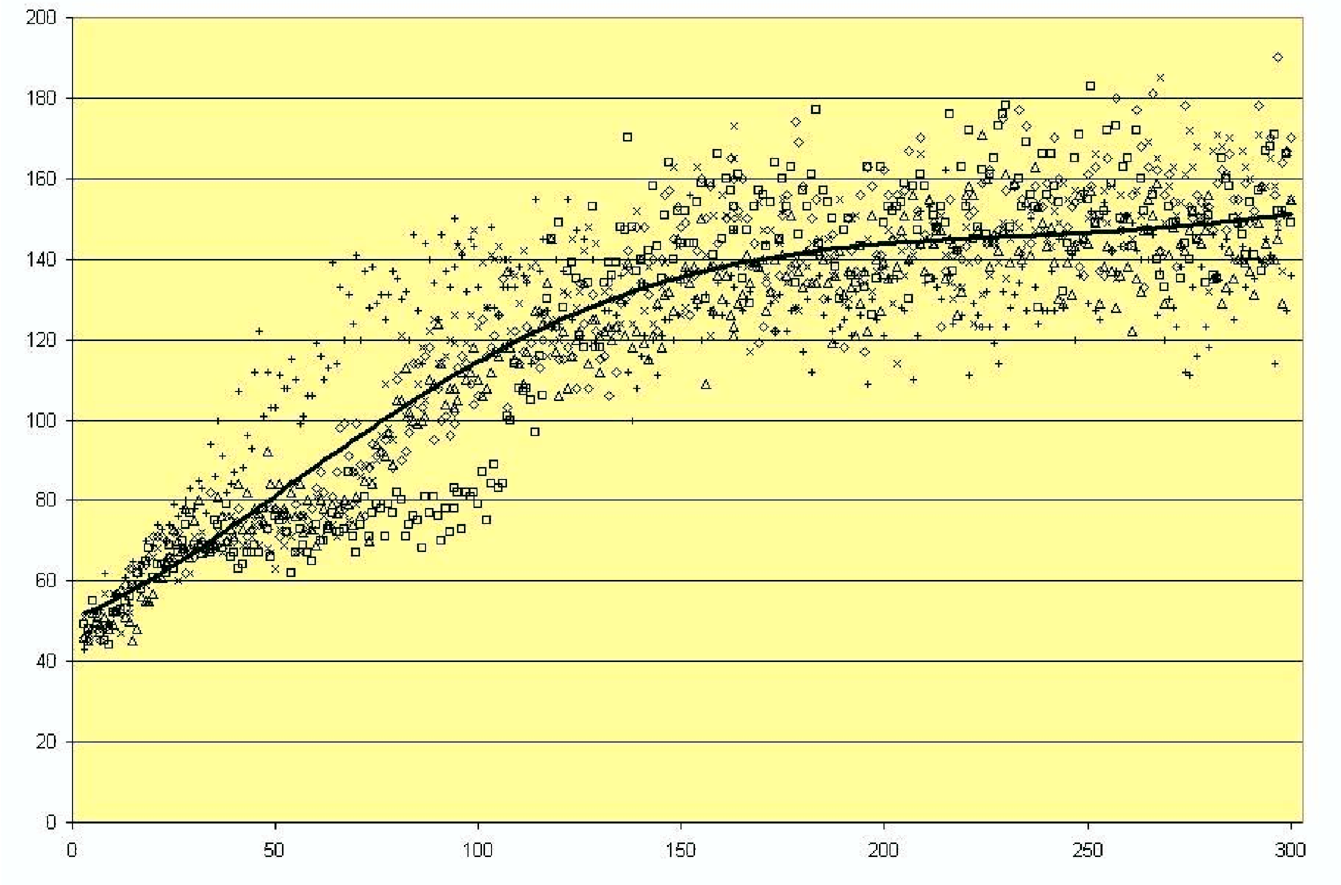

Fig. 4. Graph showing the average of the average fitnesses of the 5 tested populations over 300 generations. The line is fifth order polynomial a least squares fit.

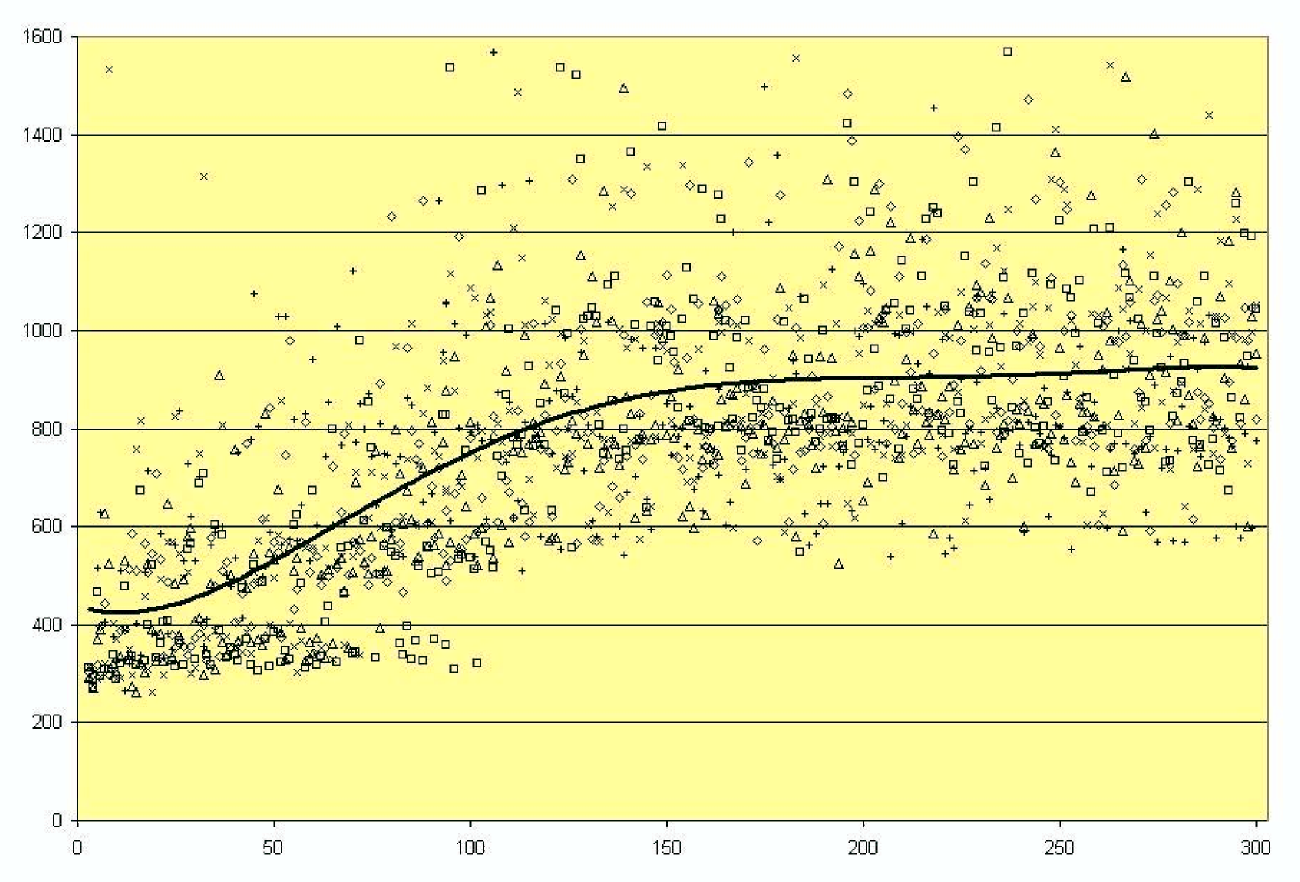

Fig. 5. Graph showing the average of the best fitnesses of the 5 tested populations over 300 generations. The line is a fifth order polynomial a least squares fit.

V. RESULTS

We ran 5 tests, each to 256 generations. We recorded the best individual for each generation as well as the average fitness of the population. The graph of the growth in average fitness over time for the 5 populations shows clear improvement (Fig. 4). The graph of the best fitnesses for the 5 populations has less obvious growth (Fig. 5), because often the best individual was merely the luckiest, and does not necessarily reflect the general health of the population or even of that individual.

The agents showed visually their improvement in behavior over time. At first they flew wildly about, smashing into walls, unresponsive to their speed or the location of the enemy. Over time, they learned not to thrust as much, and to spin around in circles, still not really aiming at the enemy, but at least not smashing into the walls. Eventually they learned to aim at the enemy, tracking Sel and shooting whichever direction he flew.

Because Sel attacked them, finding them wherever they were in the map, the agents did not need to be aggressive. Most learned to do nothing more than spin around shooting until Sel came, and then shoot at him. They had access to both the angle of the enemy ship and the "aimdir" angle, which is a calculated aiming function which takes into account both ships' velocities. The agents learned to use these angles, either singly or together, to such effect that it was more useful for them to constantly aim and shoot at Sel than to begin to learn to dodge Sel's bullets.

Some agents, though it was not a dominant trait, learned to thrust away from the walls in order to avoid crashing into them. These agents would shoot at Sel, and then give a puff of thrust as their back ends approached a wall (their shooting propelled them backwards). Some other agents learned to thrust towards Sel to attack him, which was actually a very good strategy because Sel could not so quickly dodge their faster bullets. However, these agents never combined their thrust-attack with useful wall-avoidance, and their fast attacks became their undoing, as they flew dangerously fast past Sel and into a wall.

The 5 evolved populations utilized less than half of their input nodes to perform this behavior. The most common inputs were the comparison between self's heading and the enemy's "aimdir", between self's heading and the direction of the enemy's current location, and a wall feeler or two, either +/-10 or +/-30 degrees from self's direction of velocity. Some tests had an input for some bullet attribute, and some had inputs that were constantly 1.0 or 0.0. The input nodes that we determined to not be in use by the neural network changed seemingly randomly from individual to individual in the population. Because the behavior remains similar for each individual, even with the varied inputs, those inputs were probably muted by the weights of the network, so that they made little difference to the output.

While the behavior was not as complex as we would like, it is by far the best we have evolved in Xpilot using a neural network. The underutilization of the input nodes shows the power of a network with this type of weight structure to evolve successful behavior with only a few inputs. The fitness function and simulation, which sent Sel bot quickly to attack and awarded mainly killing Sel, was not enough to warrant the network to evolve attacking, bullet dodging, and wall avoidance.

VI. CONCLUSION

Our previous research in Xpilot using neural networks [1,2] used a simple multiplicative weight system with no input thresholds, making it more difficult for the agent to evolve decisive behavior. We also guessed what inputs would be most useful to the network, often choosing too many unnecessary inputs, or not including less obvious inputs that may still be important. We manually inverted the inputs as seemed necessary to us, though the neural network may have found it less useful. We have addressed all of these problems by using a new weight structure with thresholds and inverted thresholds, allowing the network to choose to invert the inputs and to ignore the inputs after a certain point. We also evolved the inputs to the network, chosen from a large list of 64 possible inputs, so that the genetic algorithm could decide which were most important.

Our tests were run in a simple map against a single AI controlled bot. The tests showed substantial improvement in fitness over the 300 generations. The agents evolved to be well adapted to their environment, and became quite deadly to the enemy bot. Because Sel charged toward them, and because they could aim so effectively, most agents found it best to wait for Sel and shoot at him when he came. The networks seemed to utilize only about 4 or 5 of their input nodes on average, mainly using them for aiming, while muting the others.

In future research, we intend to use this same neural network with a different simulation and fitness function. One option is to put each agent through several fitness tests, such as bullet dodging, approaching the enemy, and general combat. We also intend to try this network with competitive co-evolution and in the Core, where all agents in the population simultaneously compete against one another, spreading their genes to those they kill [14]. Increasing the number of possible inputs would also be useful, especially adding more possible wall sensor angles. This particular weight structure, used with evolved inputs, has been very successful in this initial test and will most likely be the backbone of much of our future research involving neural networks.

REFERENCES

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

[2] Parker, G. and Parker, M. (2006). “The Incremental Evolution of Attack Agents in Xpilot,” Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Stanley, K., Bryant, B., and Miikkulainen, R. (2005). “Evolving Neural Network Agents in the NERO Video Game.” Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG 2005).

[10] Priesterjahn, S., Kramer, O., Weimer, A., and Goebels, A. (2006). “Evolution of Human-Competitive Agents in Modern Computer Games.” Proceedings of the 2006 IEEE Congress on Evolutionary Computation (ECE 2006), Vancouver, BC, Canada, July 2006.

[11] Miles, C. and Louis, S. (2006). “Towards the Co-Evolution of Influence Map Tree Based Strategy Games Players.” Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG 2006).

[12] Parker, G., Doherty, T., and Parker, M. (2006). “Generation of Unconstrained Looping Programs for Control of Xpilot Agents,” Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.

[13] Parker, M. and Parker, G. (2006). “Using a Queue Genetic Algorithm to Evolve Xpilot Control Strategies on a Distributed System,” Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.

[14] Parker, G. and Parker, M. (2006). “Learning Control for Xpilot Agents in the Core,” Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.