Modern AI for games: RoboCode

Jon Lau Nielsen (jlni@itu.dk), Benjamin Fedder Jensen (bfje@itu.dk)

Abstract— **The study concerns the use of neuroevolution, neural networks and reinforcement learning in the creation of a virtual tank in the AI programming game Robocode. The control of a turret of such a robot was obtained through the use of historical information given to a feedforward neural network for aiming, and best projectile power by Q-Learning. The control of movement was achieved through use of neuroevolution through augmenting topologies. Results show that such machine learning methods can create behaviour that is able to satisfactorily compete against other Robocode tanks.**

I. INTRODUCTION

This study concerns the use of machine learning based agents in the guise of virtual robot tanks. Robocode [1] is an open source programming game, in which these robot tanks compete against each other in a simple two dimensional environment. The game is roborealistic in the sense that limited information about the environment are given to robots, and requires robots to actively scan the environment to gain new information. In the study such agents has been made with machine learning techniques, which are composed of two controllers; a turret controller and a movement controller.

The movement controller use NeuroEvolution through Augmenting Topologies [4] (NEAT), and the turret controller uses artificial neural network [7] (ANN) and reinforcement learning (RL). The reason for two techniques in the turret controller, is that aiming with the turret gun itself and loading a projectile of a certain power are divided into two subcontrollers. The neural network is a fully connected feedforward network using backpropagation for error gradient correction for controling the aim of the turret. The reinforcement learning is modified Q-Learning utilizing a Q-table. It controls the power of the projectiles to be fired.

A. Background

Evolutionary algorithms have been widely used to evolve behavior in Robocode agents.

In a study by J. Hong and S. Cho the use of genetic algorithms (GA) was investigated [2]. They separated the behavior of their agents into five subbehaviors, namely movement, aiming, bullet power selection, radar search and target selection. Their encoding of a genome consists of a gene that represents each of these subbehaviors. A finite set of actions to represent genes was made, and evolution was done through natural selection, mutation and crossover breeding. Training was performed against single opponents, and evolved specialized behavior. They were able to produce numerous genomes of distinct behavior capable of defeating simple opponents. The emergence of behavior was however restricted by the finite set of actions available.

Another study by Y. Shichel, E. Ziserman and M. Sipper investigated the use of tree-based genetic programming (GP) in Robocode agents [3]. Here the genomes are of varying size and consists of genes that refers to functions and terminals organized in a tree structure that relay information from the robocode environment and allows computation. They divided their agents behavior into three subhaviours; forward/backward velocity, left/right velocity and aiming. Evolution was done through tournament selection, subtree mutation and subtree crossover breeding, as well as putting upper bounds on the depth of the genetic trees. Training was done against several opponents per fitness evaluation, and was able to evolve generalized behavior. They used the scoring function that robocode uses as their fitness function, and was able to claim a third place in a Robocode tournament named the Haikubot league. Agents that participate in this tournament are only allowed four lines of code, which was suitable to the long lines of nested functions and terminals generated by genetic trees.

Neural networks has been used in RoboCode agents since May 2002, first utilized by Qohnil's (alias) agent. It was only a year later the Robocode community started to take interest in neural networks, when Albert (alias) experimented with different inputs in his tank "ScruchiPu" - but without approaching the accuracy of other top targeting methods. Finally in May, 2006, Wcsv's (alias) neural targeting agent named "Engineer" reached a RoboCode rating of 2000+ [14]. The most modern neural network for RoboCode is currently Gaff's (alias) targeting which is currently one of the best guns to predict the future positions of wave surfers [15]. Pattern matching and a technique termed "wave surfing" can be used for agents to model incoming projectiles as waves which they then atttempt to surf [16][17].

B. Platform

RoboCode is a real-time tank simulation game, where two or more agents compete for victory in points. Agents can participate in an online tournament called Roborumble [9] and climb the ranks. RoboCode was originally started by Matthew A. Nelson in late 2000, and was in 2005 brought to Sourceforge as an open source project [1].

Robocode are built on roborealism principles, and thus hides information about opponent positions, projectile positions et cetera. In order for an agent to gather the needed information to make informed decisions, it must use a radar controller. The radar gather information about opponents within its field of view, but not their projectiles. This information consists of opponents directional heading, current position and energy. Temporal information such as opponent velocity can be calculated from this information with respect to time.

The game is used as a teaching aid for aspiring JAVA, .NET and AI programmers given its simple interface and simple ingame JAVA compiler as well as a community in which ideas, code, algorithms and strategies are shared.

The mechanics of the game are such that agents receive sensory input at the beginning of each turn and is able to activate various actuators such as acceleration and projectile firing. These actions are either on a turn basis, or temporal. An action such as `MOVEAHEAD 1000 PIXELS` is temporal and will halt the program until it has finished, whereas a non-temporal action such as `ACCELERATE 20 PIXELS` causes no halting. Temporal actions are incapable of adapting to new situations, and are therefore used only for very primitive agents. At every round the robot tanks spawn at random positions with 100 energy.

Energy is used for firing projectiles, which can be fired with a power ranging from 0.1 to 3.0. Power does not determine only damage on impact however, but also the velocity of the projectile; high powered projectiles move slower than low powered projectiles. Damage is calculated from the power of the projectile and gives a bonus if high powered, thus making the high powered projectiles the most lethal in average. Additionally, when damaging an opponent, the agent recieves triple the amount of damage done as energy. Energy is also lost when the agent drives into walls, and when opponents collide with them, which is termed *ramming*. Ramming causes no damage to the agent who apply forward velocity against the obstacle opponent, and is thus also useful for offense. Should an agent reach 0 energy, by getting hit, driving into walls or drained by shooting, the agent will loose.

The radar controller has already been perfected through a technique termed *Narrow Lock* [12] from the Robocode community. It keeps an opponent within the field of the view constantly. Thus this technique is used for the radar throughout the study.

II. Turret Controller

The turret controller is split into two parts; aiming and projectile power.

For aiming an artificial neural network (ANN) [7] with one hidden layer estimates the future position of the opponent. For implementation details, the ANN uses 0.9 momentum rate and 0.1 learning rate.

The input of the ANN for aiming consists of a value describing how much power is being put into the projectile, and a sequence of last known positions of the opponent spread over a range of radar scans. All inputs are normalized to be in range $[0;1]$. The last known positions are drawn as *dots* following the opponent in the presentation video, and we will therefore refer to them as green *dots*.

While exploring different possible inputs, we found it unneeded to include velocity and heading of the enemy, as it is indirectly included through the previous positions.

As for the output, one relative coordinate (two XY-outputs) is used to estimate where the enemy will be in the future. The relative coordinates are used such that placement on the battlefield wont matter for the neural network.

When firing a projectile, the input data which finds the estimated future position of the enemy is saved. For each turn a method checks if any projectile has flown further than expected, if so the projectile missed the target and the current relative position of the opponent is marked. Backpropagation happens the turn when the projectile hits a wall, using the stored input and exact location of the opponent for when the projectile should have hit the opponent, relative to the position the projectile was fired from.

Before being able to estimate the future position of an opponent, the most suitable degree of power for the projectile has to be determined. The more power put into a projectile the more damage it deals, but the slower it moves and therefore (all things equal) it is harder to hit an opponent with high powered projectiles, given it has more time to alter its moving pattern to avoid it.

To pick the best degree of power, a Q-learning module is made to divide the battlefield into states surrounding the agent, as seen in Figure 1. A state consist of only a start range and end range.

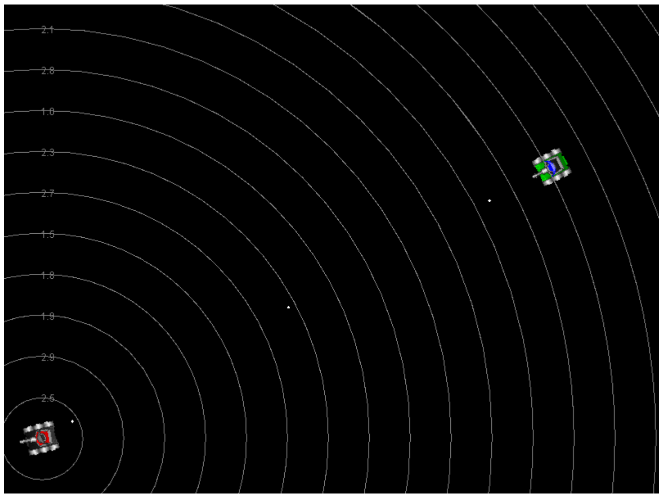

Fig. 1. The agent with surrounding rings, each representing a state in the RL.

In the Q-table each state is represented by itself and its actions $Q(s_t, a_t)$. Here the actions are the different degrees of power that a projectile can be fired with. Since the best shot is determined only by the maximum Q-value of the possible actions (shots) in a given state, and not by a discounted maximum future value, the equation for updating Q is rewritten as can be seen in Equation (1).

$$Q(s_t, a_t) = Q(s_t, a_t)(1 - \alpha_t(s_t, a_t)) + \alpha_t(s_t, a_t)R(s_t) \quad (1)$$

Where the Q value of state s_t applied action a_t is updated with part of the old Q value and presented with a part of the reward, $R(s_t)$, given by the newly fired projectile. To allow for small changes the reward and old Q value is affected by the learning rate $\alpha_t(s_t, a_t)$ which in our application has been set to 0.1 by default.

For $R(s_t)$ the energy-penalty and rewards given by Robocode is used, in form of damage dealt (positive), gun heat generated (negative), energy drained (negative) and energy returned (positive) [1]. These are determined by the power of the projectile fired - a shot powered by 3 will drain the agent of 3 energy, generate 1.6 turret heat, but cause 12 damage to the opponent it hits, with additional 4 bonus damage and 9 energy returned to the agent. This will cause a reward of $-3-0.8+12+4+9 = 21.2$ but missing a high powered shot will create a bigger penalty than a smaller shot. The reward is given upon the event of hitting an opponent, hitting a wall (missing) or hitting another projectile. The full reward function can be seen in Equation (2), where the event PHP means a projectile hits the enemy projectile.

$$R(s_t) = \begin{cases} -Drain - Heat + HitDamage & \\ +EnergyBack + Bonus, & \text{if hit;} \\ -Drain - Heat, & \text{if miss;} \\ -Drain - Heat, & \text{if PHP.} \end{cases} \tag{2}$$

A. ANN: Input experiments

The purpose of this ANN experiment is to see the effect of using the past positions of the opponent. How much information is needed to provide the best estimation of the future position? The best input will be used from this test onward.

To train the ANN, only the turret is active and only the opponent is allowed to move throughout the training. The opponent "sample.MyFirstJuniorRobot" is used, which uses simple movement patterns and do not alter the pattern based on shots fired towards it.

For each turn of the rounds the ANN is provided with the necessary inputs of the battlefield and asked for an output. As mentioned in before the inputs used is the shot and an amount of history from where the opponent has been prior to this point in time (dots). To have a useful ANN which can be used afterwards, the power-input is randomized for each shot.

While decreasing the amounts of *dots* in size, the amount of hidden neurons is also decreased with a 1 : 2 ratio. As an example: ten dots needs 20 numbers (X/Y) and one input is used for energy of the shot, the total input size is 21 and therefore 42 neurons is used on the hidden layer. The reason for decreasing the hidden neurons, is to allow for less error corrections as the patterns are getting less complex.

Each input is tested over 1000 sessions of 50 games each. For each session the current hit percentage is saved and shown in the graph. Figure 2 shows the average of these percentages. Since these are averages, slow converge pulls down the average therefore the average of the last 20 sessions (1000 rounds) are shown in table I.

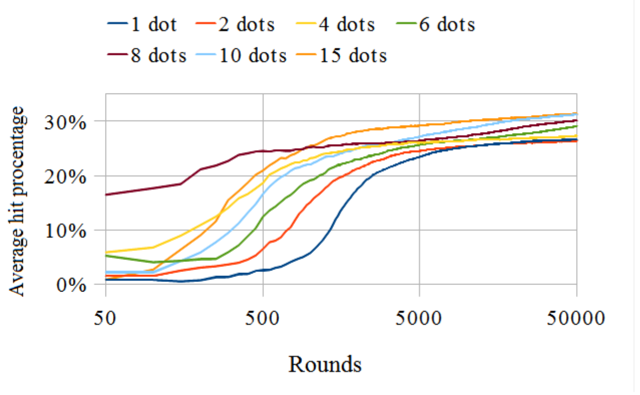

Fig. 2. Input test for ANN using relative coordinates.

TABLE I
HIT% OVER THE LAST 20 SESSIONS (1000 ROUNDS)

Dots	Input size	Hidden neurons	Hit%
15	31	62	32.69
10	21	42	32.08
8	17	34	31.38
6	13	26	30.96
4	9	18	27.81
2	5	10	26.22
1	3	6	27.60

B. RL: Range experiment

This test allows us to show the benefit different ranges provides. The smaller the ranges, the more circles are placed around the agent, as seen in Figure 1 which represents the states.

The benefit from dividing the states into smaller ranges, allows the turret to find which shot is the most rewarding in each state. As high powered shots moves slow, we expect the RL-module to figure out a balance between risk and reward. The RL-module also has the benefit of tailoring the needs of the ANN. Should the ANN start missing in one state the RL can lower the energy put in the projectile.

For this test, the ANN from the previous test is used using 15 previous positions (dots), 31 inputs and 62 hidden neurons. Throughout the test, the neural network is frozen such that no weight updates can be performed. For 32 shots (about one fast round) the agent shoots random powered shots in range $[0.1; 3]$, for this period the RL is allowed to update the Q-table. For the next 32 shots the RL stops updating the Q-table and utilizes the gained knowledge. The damage dealt over the utilized 32 shots is recorded and the process is repeated 15.000 times (close to 50.000 rounds).

Figure 3 shows the average reward given for each range throughout the test.

Damage done is a big factor in the reward function, as can be seen in Equation (2), and important in taking down the opponent. Therefore the test were done twice allowing us to show the effect of average damage done as seen in Figure 4.

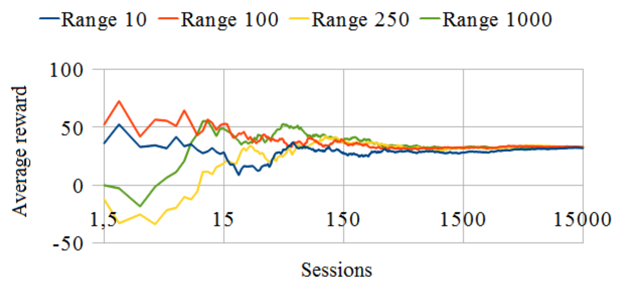

Fig. 3. Reward test using RL-module with frozen ANN with the enemy's 15 previous positions as input.

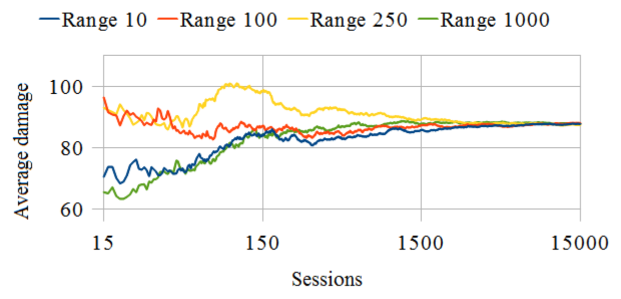

Fig. 4. Damage test for RL using ANN with the enemy's 15 previous positions as input.

C. Result discussion

In testing different inputs for the ANN, the usage of more historical information (or dots) clearly has a big impact on how well the ANN can predict. From a single dot (meaning the current position of the enemy) it was able to hit an average of 27.60%, while giving it more information such as 10 and 15 dots resulted in an average of 32.08% and an average 32.69% as seen in Table I.

While further increasing dots, the advantage of knowing more became smaller, as seen with ten dots and 15 dots. Further dot increase could possible move it to 33%, but this would also hinder the agents performance on another field; when a new round starts, the turret waits for all inputs to be gathered before shooting, meaning slowing down shooting.

The RL range test allowed us to show the impact of both reward recieved and damage dealt as the RL-module altered its Q-table, as seen in Figure 3 and Figure 4.

As we were trying to explain the experimental results, we noticed if everything can be calculated in time t, as can be seen in Equation (1), and since the RL is not dependent on other future Q values of other states, there's a better solution than RL. The solution is simple, if the hit percentage for each state is known by the ANN, the expected reward can be calculated for each degree of power (action), as can be seen in Equation (3), and as such a near optimal solution can be precalculated into a lookup-table. The lookup-table has been precalculated and attached on the CD.

$$\begin{aligned} ExpectedReward(s_t) = &(Hit\%)(-Drain - Heat \\ &+ HitDamage + EnergyBack \\ &+ Bonus - Drain - Heat) \\ &+ (1 - Hit\%)(-Drain - Heat) \end{aligned} \tag{3}$$

While this is a better solution, the RL is far from a bad solution - it just takes time to converge.

Having the near optimal policy helps us explain why the test-results ended as they did:

In Figure 3 and 4 the average reward and average damage can be seen to meet at the same value, for all ranges. In Figure 3 the reward seems to converge at 33 and at Figure 4 the average damage is 87. As we are using a ANN that averages a hit percentage of 32.69% over all states the optimal expected reward is about 3.5 points (according to the near optimal policy) using the optimal shot with a power of three. As the experiment doesn't show $3.5 * 32 = 112$, it's a strong indication that the RL is far from fully converged.

Before starting the range experiment, it was assumed range ten would outperform range 1000, as knowing how well the turret hit in one state, allowed for a tailored shot for that range. At close range a high hit percentage might be present and a high powered projectile should be used for optimal reward. But as can be seen in the optimal values going for the high powered shot is generally the best option given our turret is shooting with about 32.69% hit percentage.

Should we later meet a more evasive opponent, or should the ANN start missing at long range, range ten would show much better results. The reason range ten would show better results, is its ability to isolate the areas where it's hitting well and use the highest powered projectile, while using a less powered projectile in areas where the hit percentage is low - on the other side range 1000 wouldn't be able to adjust to maximize the reward and minimizing the penalty of missing.

III. Movement Controller

The movement controller is composed of sensors and actuators. A multilayered recursive neural network processes the sensor information and actuate through synapses and neurons, which feeds information through the network by neural activation [7]. This network is created through neuroevolution by the use of NEAT. In evoluationary terms, the neural network is the phenotype of a genome, which has been evolved through several generations of mutation and selection. NEAT evolves the topology of a neural networked genome, through the creation of additional neurons and synapses, initially starting with a minimal structure consisting of input and output neurons and full connectivity [4]. As with other genetic computational methods, the best genomes are selected for cross breeding and are allowed to survive for future generations. However, NEAT features speciation which divides dissimilar genomes into species and allows all species to exists for a number of generations independent of their immediate suitability to the environment [4].

This allows evolutionary innovations to occur that gives no immediate benefit, but may serve as the base of offspring that may prove strong in future generations. Species does however go extinct if they have shown no improvement over several generations. Additionally, the history of each synapse in the genome is recorded, such that offspring throughout the species share a genetic history [4]. This allows crossover between dissimilar genomes without duplication of synapses, as only synapses with the same genetic history are crossed over. The strength of the parents determine the topology should be inherited, whereas the synapses of the identical structure are randomly chosen.

NEAT has been implemented in JAVA as a library for the purpose of this study. The library is based on *NEAT C++ for Microsoft Windows* [6] and provides similar features such as sigmoid activation response mutation and recursive synapses. There are several additions however. It is possible to load the neural networks of phenotypes and use them as it was an ordinary multilayered neural network, without loading the NEAT module. This network also allows recursion not only of synapses that connects the same neuron, but also topology that causes recursion. This is done through method similar to breadth first search that allows neurons to be visited twice, rather than once. This allows memorization of the previous time step as well as small cycles during processing. Recursion is an optional feature, but the cycles are a sideeffect of the innovations of new synapses that may cause cycles. Another approach would be to restrict innovations such that cycles does not occur. Another additional feature is the ability to evolve based on a genome rather than minimal structure, which allows training through several phases. Saving and loading of both genomes and phenotypes are supported.

A. *Sensors*

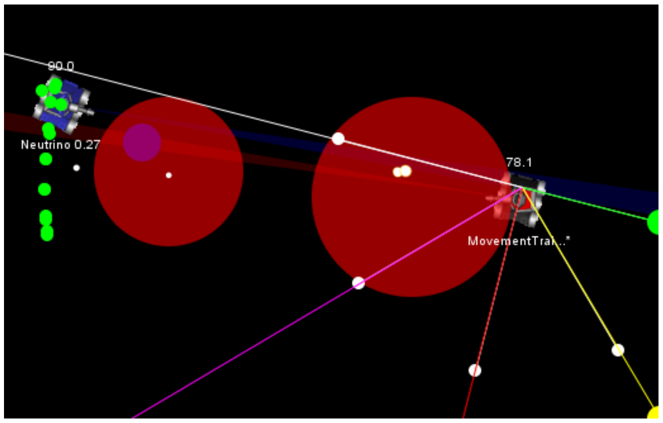

Fig. 5. Agent in-game displaying its sensors. The agent is black and red, and the opponent blue.

The input for the movement controller consists of eight continous values supplied from eight sensors as can be seen in Figure 5. Six of the sensors are distance sensors, where five detect the distance to the walls of the environment, and the last the distance to the opponent. The sensors who measure distance to walls do so by casting rays at angles that cover the forward field of view of the agent. The environment are modeled as the four lines that the square of the environment consists of, and are then checked for intersections with the rays. The resulting intersection points are then used to calculate the distance from the ray origins to the environment boundary [8]. To fit within the range [0;1], only distances within an upper bound are perceived. The two remaining sensors are the angle to the opponent, which together with the distance gives the relative polar coordinates to opponents, and a projectile danger field sensor. This last sensor, the projectile danger field, is the amount of danger of projectile impacts within the environment, which is determined by an attempt to track all projectiles as they travel from an opponent. However, it is impossible to detect at what angle projectiles has been fired by opponents. It thus arbitrarily assumes that the range is toward this agent, but this is imperfect as better opponents fire at angles that intercept the agent at future turns. In order to make better predictions of the flight path of enemy projectiles, a prediction method must be used. This has not been done in this study.

Together, these sensors provide enough information for the agent to percieve its environment and the opponent within. The movement controller reacts to the environment through forward/backward acceleration and left/right acceleration actuation.

B. *Training*

The training of the movement controller is divided into four phases of increasing difficulty. First it must learn to move around the battlefield at the highest possible rate. Then it must learn to chase an opponent, and attempt to keep a specific distance to it. Then it must learn to avoid getting hit by an opponent, and lastly it must learn to position itself optimally for the turret controller. All the phases has the limitation that the agent must never touch a wall; if it does so, it will receive a fitness score of zero. There are consequences of such a strictness; innovations that later could turn out strong, may be discarded early because of inability to move correctly. Preliminary tests however showed that allowing the agent to touch walls with a lesser penalty could spawn behavior that optimized around hitting walls, which is unacceptable.

Dividing training into phases does cause loss of knowledge, but in the context of the training being done this is acceptable and necessary. The idea is that instead of optimizing from a minimal structure which is not adapted to the environment, each phase after the first can optimise from a structure that is already suboptimal within certain aspects that is deemed useful. To learn how to chase for example, it is an aid that it already knows how to move, albeit the new movement after training will of course be very different than the base of which it originated from. Tests have shown however that in order to preserve previous knowledge, the fitness function must constrain the training in such a way that old behavior is required as well as the new desired behavior.

Thus the fitness function grow in complexity as the phases increase.

1) Phase 1: Mobility: The fitness function for the first phase is given in Equation (4). The variable w is the amount of wall hits, $\overline{v}$ is the average forward/backward velocity of a round and $\overline{h}$ is the absolute average left/right velocity of a round with a minimal bound of 0.1. A maximum score of 100 is possible if $\overline{v}$ is 1 and $\overline{h}$ is 0.1. A fraction is used to enable the fitness to depend on two variables. Preliminary testing showed that simply addition or multiplication can cause training to get stuck in local optima that only optimizes one of the variables. An optimal behavior to this function should thus move forward at highest possible velocity constantly, avoid any walls, and keep turning to a minimum. The reason for a limitation on turning, is that preliminary tests without this variable had a tendency of evolving behavior that turned in circles in the center of the environment thus fulfilling w and $\overline{v}$.

$$f_1(s_i) = \begin{cases} 0, & \text{if } w > 0; \\ \left(\frac{\overline{v}}{\overline{h}}\right)^2 \overline{v}, & \text{otherwise.} \end{cases} \tag{4}$$

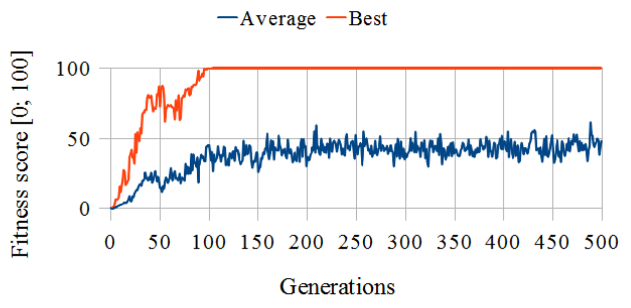

Fig. 6. Training with $f_1(s_i)$: 500 generations with a population of 100, 1 game per specimen, no opponent.

As can be seen in Figure 6 the agent quickly evolved structure which optimize the fitness function, with the resulting genome having a value of 100. The training spanned 500 generations with a population of 100 specimen, each of which had one round to optimize their fitness. Since there are no opponents in this phase, the environment is static and unchanging, which allows a fairly precise fitness calculation as chance are mostly avoided. Behaviorally the agent has a preference of turning right, and does drive in cycles. However, the cycles are so wide, that it must react when nearing walls, which also disrupts the cycles and cause it to travel around most of the environment.

2) Phase 2: Chasing: The fitness function for the second phase is given in Equation (5). It depends on the same variables as in the previous phase, with several additions to determine its ability to chase an opponent. The variable $\overline{d}$ is the average distance when within sensory range (the sixth distance sensor). Within the numerator of the fraction, this variable is part of a triangular equation where 0.5 is the top and 0 and 1 is the bottom. This forces it to stay within the middle of the sensory range of the sixth distance sensor. The fraction has a maximum of 100 as with the previous phase. An additional term is added, with the variable c which is the amount of turns within sensory range, and t which is the amount of turns in total. This has the effect, that the agent should stay within sensory range for at least half of the round in order to fully benefit from the fraction. Since start locations are random, it takes time for the agent to get within sensory range of an opponent, and this ensures that it is not penalized for this, which would cause initial random location to affect the fitness. In effect, the agent is forced to stay within the middle of the sensory range for at least half of the round.

$$f_2(s_i) = \begin{cases} 0, & \text{if } w > 0; \\ \left(\frac{(1-[\overline{d}-0.5]*2)}{\overline{h}}\right)^2 \overline{v}\frac{c}{(\frac{t}{2})}, & \text{otherwise.} \end{cases} \tag{5}$$

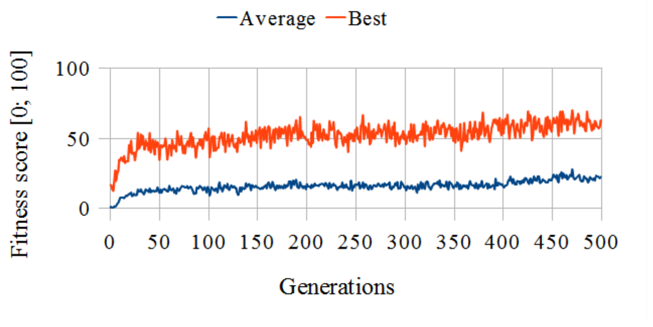

Fig. 7. Training with $f_2(s_i)$: 500 generations with a population of 100, 5 games per specimen, random moving opponent.

The results can be seen in Figure 7. The resulting genome have a value of 63.07. The training spanned 500 generations with a population of 100 specimen, where each specimen had five rounds to optimize. The opponent was “sample.crazy” [1], a random moving opponent that does not attack. The reason for the additional turns, is that there now is an opponent which causes the environment to become dynamic and now includes chance. There is a clear sign of improvement as generations increase, but it has been unable to fully optimize for the fitness function. This can also be seen in the behavior, that has inherited the circular movement of the first phase. It clearly gets within sensory range, but it continuously gets too close or too far, as it circles either around or away from the opponent. A curious tendency, is that if the agent is pitted against opponents made to circle around their prey, they will drive around in a perfect circle, where both attempts to get an optimal distance but negates each other.

3) Phase 3: Avoidance: The third phase was attempted with two different fitness functions, Equation (6) and Equation (7), as it was unclear on which variables the training should depend on. The variable $\overline{k}_1$ is the average danger sensor score, and $\overline{k}_2$ is the average projectiles that hit the agent out of the projectiles that was fired by the opponent. The difference lies in that the sensor is an abstraction of possible projectile dangers, whereas $\overline{k}_2$ is the actual occurrences in the environment. Part of the fitness function of phase 2 is included, such that it should attempt to maintain the desired distance to the opponent while at the same time avoiding being hit. There is no constraint on velocity however. The danger variables are inverted, such that avoidance is rewarded. In effect it should avoid projectiles while staying within optimal range, some that may seem like a paradox as closer range means a statistically higher change of getting hit. However, since movement should benefit the turret controller, it must stay within range such that the statistical chance of hitting the opponent is high for the agent as well.

$$f_{3,1}(s_i) = \begin{cases} 0, & \text{if } w > 0; \\ (1-\overline{k}_1)\left(\frac{(1-[\overline{d}-0.5]*2)}{\overline{h}}\right)^2 \frac{c}{(\frac{t}{2})}, & \text{otherwise.} \end{cases} \tag{6}$$

$$f_{3,2}(s_i) = \begin{cases} 0, & \text{if } w > 0; \\ (1-\overline{k}_2)\left(\frac{(1-[\overline{d}-0.5]*2)}{\overline{h}}\right)^2 \frac{c}{(\frac{t}{2})}, & \text{otherwise.} \end{cases} \tag{7}$$

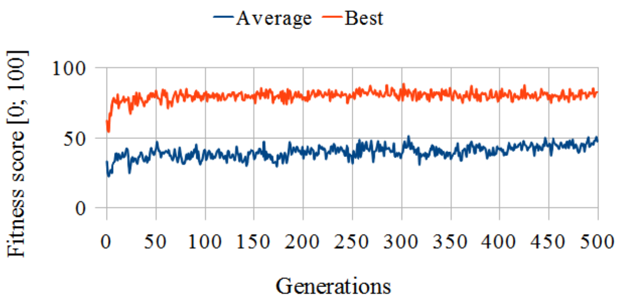

Fig. 8. Training with $f_{3,2}(s_i)$: 500 generations with a population of 100, 5 game per specimen, direct targeting opponent.

TABLE II
COMPARISON OF THE TRAINED GENOMES OF THE FITNESS FUNCTIONS.

Fitness function	Avoidance percentage
$f_2(s_i)$	73.53%
$f_{3,1}(s_i)$	78.70%
$f_{3,2}(s_i)$	78.84%

As can be seen in Figure 8 the agent had difficulties evolving a structure that optimizes the function. The resulting genome have a value of 82.49. The training parameters is identical to the previous phase, but with an attacking opponent named "rz.HawkOnFire 0.1" which uses direct targeting. Direct targetting was chosen because the danger sensor is only able to predict projectiles fired by such a strategy. It starts quite high as the training from the previous phase receives a fairly good score from the part of the function that they share. Innovations does however increase the success of the agent as generations increase, but it is unable to achieve total avoidance. By keeping part of the previous fitness function, it maintains the previous behavior while adding new behavior that in this case attempts to avoid getting hit by the opponent.

The efficiency of the two fitness functions can be seen in Table II. The second fitness function, Equation (7), proved best, while both proved to be an improvement from the structure of the previous phase. There are however only a very small difference in the efficiency of training from using an abstraction or real occurrences.

4) Phase 4: Combat: Two fitness functions was investigated for the combat phase, as can be seen in Equation (8) and Equation (9). Variable u is the number of projectiles fired by the agent that hit the opponent, and r is the number of projectiles fired by the opponent that hit the agent.

Equation (8) simply rewards hitting the opponent and avoid being hit through a fraction. This is an abstraction of how combat works; the robot that hits its opponent the most will achieve victory. However, bullet power is not mirrored in the fitness function as that is out of scope for this controller. It is assumed that the turret controller is itself near optimal, such that this abstraction is true. Tests have shown that it evolves cyclic behavior, and appear to loose much of the previous training.

Equation (9) forces the agent to evolve a behavior that still fulfill the task of chasing. The performance of both are to be evaluated in the result section.

$$f_{4,1}(s_i) = \begin{cases} 0, & \text{if } w > 0; \\ \frac{u}{r}, & \text{otherwise.} \end{cases} \tag{8}$$

$$f_{4,2}(s_i) = \begin{cases} 0, & \text{if } w > 0; \\ \frac{u}{r}\left(\frac{(1-[\overline{d}-0.5]*2)}{\overline{h}}\right)^2 \frac{c}{(\frac{t}{2})}, & \text{otherwise.} \end{cases} \tag{9}$$

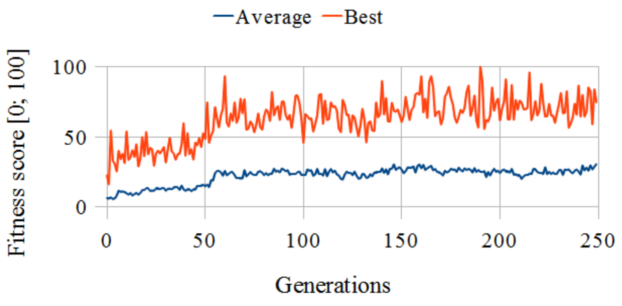

Fig. 9. Training with $f_{4,1}(s_i)$: 250 generations with a population of 100, 5 games per specimen, predictive targeting opponent.

Since the training for the turret was only done using the "sample.MyFirstJuniorRobot" [1] opponent, it does not have the accuracy needed for real opponents. Thus "demetrix.nano.Neutrino 0.27" which ranks 397 in Roborumble [9] was used to train the turret through 10000 rounds before the training of the movement controller begun. Unlike in the chasing phase, this opponent does not use direct targeting, but use prediction. This will force the agent to learn how to avoid these projectiles as well. Overfitting will occur,

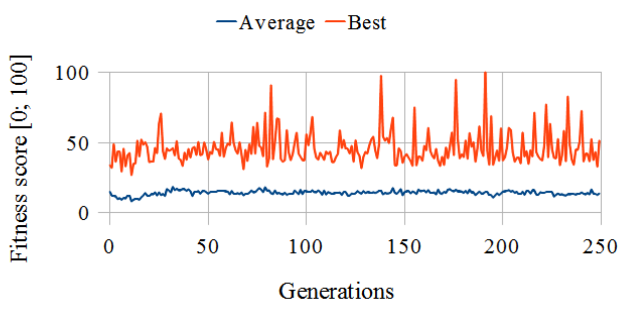

Fig. 10. Training with $f_{4,2}(s_i)$: 250 generations with a population of 100, 5 games per specimen, predictive targeting opponent.

in that it will optimize for this opponent and this opponent only. The best solution would be to train against several opponents of varying strategies that generalize the tactics used throughout the whole population of robots. However the application does not allow swapping of opponents, so this is unfeasible without altering the freely available source code. This has not been done, and the training has been made within the constraints of the officially released client. Harder opponents was not chosen, as preliminary tests showed that the turret had problems matching their complex moving patterns, and that no movement improvements could be found that increased the agents overall success.

The training can be seen in Figure (9) and Figure (10). Training with the fitness function $f_{4,1}(s_i)$ resulted in a genome with a value of 74.67 and function $f_{4,2}(s_i)$ resulted in a genome with a value of 51.46. The training spanned 250 generations with a population of 100, and five rounds per specimen. Thus the simpler fitness function was easier for the agent to fit, whereas it had difficulties in fitting the second function. The resulting structure of the genomes are minimal, such that the genome of the simple function contain only five hidden neurons and 15 new synapses, and the second function contain only four hidden neurons and 13 new synapses.

The behavior is as expected; the agent with a phenotype derived from the genome of $f_{4,1}(s_i)$ moves in fast irregular cycles, and the agent of $f_{4,2}(s_i)$ moves in almost constant motion, circles around the opponent and keeps within range. Both is capable of avoiding projectiles fired by opponents.

IV. Results

To determine the performance of the combination of the movement and turret controllers, a test is made against robots from the official Roborumble tournament [9]. The "Sample.MyFirstJunior" robot serves as the base, and then three robots of decreasing ranks in the official Roborumble tournament [9]. These are "matt.UnderDark3 2.4.34" with a rank of 691, "ar.TheoryOfEverything 1.2.1" with a rank of 411 and "jk.mega.DrussGT 1.9.0b" with a rank of 1. Two agents will be measured to determine the difference by the two movement controller variations found in phase 4 of the training. Each opponent is engaged in combat with these agents ten times for 1000 rounds.

TABLE III
Performance of agent with movement variant A

Opponent	Mean	Standard Deviation
"Sample.MyFirstJunior"	72.5%	0.92%
"matt.UnderDark3 2.4.34"	48%	7.75%
"ar.TheoryOfEverything 1.2.1"	20.5%	0.67%
"jk.mega.DrussGT 1.9.0b"	1%	0%

TABLE IV
Performance of agent with movement variant B

Opponent	Mean	Standard Deviation
"Sample.MyFirstJunior"	74.8%	0.75%
"matt.UnderDark3 2.4.34"	26.2%	0.87%
"ar.TheoryOfEverything 1.2.1"	21.1%	0.7%
"jk.mega.DrussGT 1.9.0b"	2%	0%

The scoring used is that of Robocode [10], which is a sum of survival score, bullet damage and ram damage, and additional bonuses. This scoring scheme rewards not only victory, but also the ability to damage an opponent. It does thus not reward agents that wears opponents down through avoidance only, but requires offensive actions. Ramming is the action of driving into an opponent, which causes damage to the opponent only.

The agent with movement control trained by Equation (8) will be referred to as movement variant A, and the movement control trained by Equation (9) will be referred to as movement variant B. The performance of these agents can be seen in Table III and Table IV.

Both movement variants performed well against "Sample.MyFirstJunior" which is the base line. There is little variance, as both has a standard deviation (SD) below 1%. The agents perform above chance, and is able to fire and move in such a way that they can defeat an offensive opponent. Against harder opponents, the agents have difficulties winning. Against "ar.TheoryOfEverything 1.2.1" and especially "jk.mega.DrussGT 1.9.0b" they perform equally poor, and decidedly so with a SD below 1%. The opponent "jk.mega.DrussGT 1.9.0b" is the current champion of Roborumble [9], and the agents are incapable of hitting it, nor avoid getting hit by it. They are however able to both hit and avoid the projectiles of "ar.TheoryOfEverything 1.2.1" , but at a too low frequency to claim victory.

There is a big difference in the performance of the agents against "matt.UnderDark3 2.4.34". Movement variant B performs almost as poor as with "ar.TheoryOfEverything 1.2.1", with a mean of 26.2% and a SD of 0.87%. Movement variant A however had an almost even match with a mean of 48% and a SD of 7.75. It does however have slightly less success against the other opponents than variant B. Overall the agent trained with the simple fitness function in Equation (8) has the best performance. This means that the complex fitness function in Equation (9) creates a too large search space, or constrains the search space in such a way, that local optima is harder to find.

The problem with both the turret controller and the movement controller is that of overfitting as well as the scope of patterns that must be approximated. There are many different opponents that have highly varying behaviors, and learning one of these puts the controller at a disadvantage against others. Some of this can be slightly hindered by training against a multitude of opponents, allowing the turret controller and movement controller to find an approximation that generalizes their behavior. As mentioned in phase four of the movement training, this was not possible because of software constraints. It would be possible for the movement controller to converge at a good generalized strategy given enough generations, and a suitable fitness function. The turret controller however faces problems with its neural network that utilizes backpropagation. While possible, it is much more prone to be stuck in undesirable local optimas, valleys in the search space.

The other problem is sensory input of both the controllers. Better input increases the chance of convergence in higher local optimas. Certainly there are enough information in the input given to the controllers, as can be seen on the performance against the base opponent and for movement variant A against the opponent "matt.UnderDark3 2.4.34". But a higher understanding of the environment and the game will allow input that are more relevant to the controllers. There exists techniques for training feature selection [13] as well, allowing the controllers to learn what inputs proves essential to their success and what is unnecessary or impeding.

V. Conclusion

We have proven that artificial neural networks can successfully be used for aiming with the turret of a Robocode agent using historical information of opponents in Section II. An average hit percentage of 32.69% was achieved using this method against a simple opponent. Using this network it was shown that reinforcement learning can be used for determining the power of a projectile, albeit the difference in power across different distances varied little due to high hit percentage. It was determined that high powered projectiles are often most favorable, but low powered projectiles were also favorable in a few circumstances.

It was also proven that neuroevolution can successfully be used for evolving relevant behavior of movement in Section III. It also shows that training can be divided into several phases, such that the size of the search space can be reduced into several smaller partitions. The terms of the fitness functions of each phase was determined to be of utmost importance to avoid knowledge loss.

The results in Section IV showed that an agent composed of controllers trained by these machine learning techniques are capable of defeating easier opponents, but was unable to defeat the leading robots in the Robocode tournament. Overfitting was a problem that could not be tackled in this study. Possible improvements was determined to be better sensory inputs for the networks in both controllers.

References

[1] F. N. Larsend, *Robocode*. [online] Available: http://robocode.sourceforge.net [accessed 2. December 2010]

[2] J. Hong and S. Cho, "Evolution of emergent behaviors for shooting game characters in Robocode," *Evolutionary Computation*, pp. 634–638, 2004.

[3] Y. Shichel, E. Ziserman and M. Sipper, "GP-Robocode: Using Genetic Programming to Evolve Robocode Players, "*Genetic Programming*. Springer Berlin / Heidelberg, pp. 143-143, 2005.

[4] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, pp. 99–127, 2002.

[5] D. M. Bourg and G. Seemann, *AI for Game Developers*. Sebastopol, CA: O'Reilly Media, pp. 269–347, 2004.

[6] M. Buckland, *NEAT C++ for Microsoft Windows*. [online] Available: http://nn.cs.utexas.edu/soft-view.php?SoftID=6 [accessed 2. December 2010]

[7] P. Ross, *Neural Networks: an introduction*. AI Applications Institute, University of Edinburgh, pp. pp. 4:1-23, 1999.

[8] J. M. V. Verth, L. M. Bishop, *Essential Mathematics for Games & Interactive Applications: A Programmers Guide*. Burlington, MA: Morgan Kaufman, pp. 541–599, 2008.

[9] F. N. Larsend, *Roborumble*. [online] Available: http://darkcanuck.net/rumble/Rankings?version=1&game=roborumble [accessed 2. December 2010]

[10] F. N. Larsend, *Robocode Scoring*. [online] Available: http://robowiki.net/wiki/Robocode/Scoring [accessed 11. December 2010]

[11] F. N. Larsend, *Robocode Scoring*. [online] Available: http://robowiki.net/wiki/Robocode/Scoring [accessed 11. December 2010]

[12] F. N. Larsend, *Narrow Lock*. [online] Available: http://robowiki.net/wiki/One_on_One_Radar [accessed 11. December 2010]

[13] K. O. Stanley et al., "Automatic Feature Selection in Neuroevolution," *Proceedings of the Genetic and Evolutionary Computation Conference*, 2005.

[14] F. N. Larsend, *Neural targeting*. [online] Available: http://robowiki.net/wiki/Neural_Targeting [accessed 12. December 2010]

[15] F. N. Larsend, *Targeting*. [online] Available: http://robowiki.net/wiki/Gaff/Targeting [accessed 12. December 2010]

[16] F. N. Larsend, *Wave surfing*. [online] Available: http://robowiki.net/wiki/Wave_surfing [accessed 12. December 2010]

[17] F. N. Larsend, *Pattern matching*. [online] Available: http://robowiki.net/wiki/Pattern_matchin [accessed 12. December 2010]