Applying and Comparing Evolutionary Algorithms for Robot Tanks

Richard Liang and Peng Zhao

Swarthmore College

Abstract

Robocode, an open source tank combat game, has become immensely popular based on both its educational as well as its "fun" value. Throughout the years, many interesting AIs have been developed for Robocode, most of which were based on finite-state machines and were generally "hard-coded". The effects of employing evolutionary algorithms and machine learning on Robocode, however, have not been so thoroughly explored. In his paper, we focus on two main methods and their effectiveness in evolving highly-intelligent robots which can effectively defeat good, existing hard-coded robots. A traditional Genetic Programming (GP) approach is first used as an intermediate goal due to its relative-ease as well as in order to familiarize ourselves with using machine learning on Robocode. We then employ Neuroevolution of Augmenting Topologies (NEAT), a method of evolving complex, unfixed topologies in Neural Networks since it has historically achieved success in evolving highly complex behavior. Although our results after experimentation were not we hoped, the trend in data leads us to believe that further modification of our experimental method can possibly help us attain the perfect battle bot.

I. Introduction

1. What is Robocode?

Robocode(short for "robot code" is an open-source game module which features tank robot battles in an arena. The module was written by Matthew Nelson while he was working as a technology evangelist for IBM Alphaworks. Conveniently, the environment of Robocode as well as Robot contestants are coded entirely in java. This means that Robocode can be run as an executable *JAR* file in practically any operating system which possess a Java Virtual Machine(JVM).

In Robocode, "players" are the programmer of a particular robot, and will have no direct influence on the game. Instead, the player must write the AI of the robot telling it how to behave and react on events occurring in the battle arena. Robots in this environment have two main goals:

- Intelligently and efficiently navigate the battlefield to avoid being shot by its opponent as well as running into walls(stalling).

- The tank must locate its adversary on its radar and shoot bullets at it. On hit, bullets deal damage to opponents. Upon taking sufficient damage, the opponent robot will explode and the the robot remaining is declared the winner of that particular round.

2. The Robots

Robots in Robocode are comprised of three parts: the body, radar, and gun. The body of a robot is its “actual mass", which also carries its radar and gun on top. Also, the body is what enables the robot to turn left/right and move forward/backward. The gun is mounted on the body and is used for firing energy bullets, and can rotate left or right. Finally, the radar is the other component mounted on the body and is used to scan for opponents. Like the robot’s gun, the radar is also allowed to turn left or right.

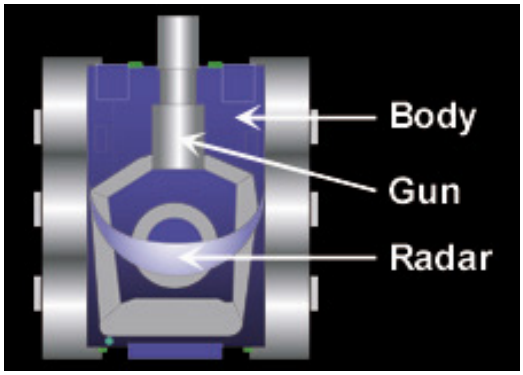

Figure 1: Anatomy of Robocode Robot agent

Robots are written as event-driven *JAVA* programs. That is, they contain usually contain a main loop which can be interrupted by event handler functions. Events can include: scanning an adversary on its radar, getting hit by an enemy’s bullet, and getting rammed into by its opponent. At the beginning of every battle, robots start with a fixed amount of energy. When a particular runs out of energy, it loses the battle. Several possible situations result in a robot losing energy, including shooting at an opponent, being hit by a bullet. However, there is only one possible way to gain energy, and that is by successfully hitting an opponent with a bullet. An interesting quality of Robocodes’ bullet-shooting mechanism is that robots can choose how much energy to expend per shot. Indeed, if a robot is confident that a bullet will score a hit on an enemy, it should expend more energy. Oh the other hand, if a robot is not so confident on its accuracy, it should expend less energy so that the risk of losing too much energy is minimized.

3. The Simulator

Robocode can be viewed as a simulator of real vehicular battles. Like most simulators, Robocode contains a mix of both realism and fantasy. Some “realistic" characteristics of

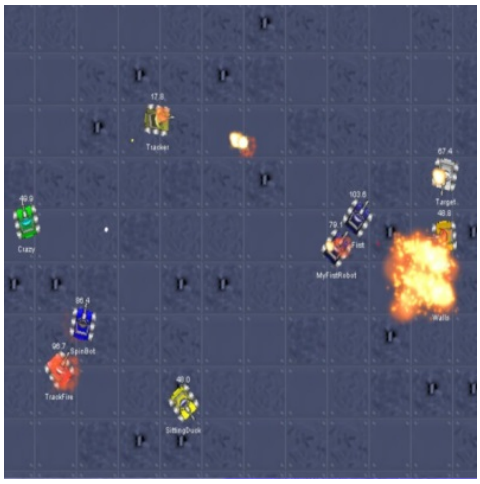

Figure 2: A Battle of 10 Robots

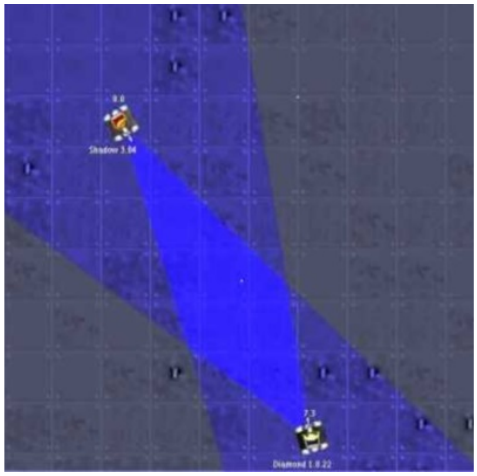

Figure 3: Radar Range of Two Robots

Robocode include:

- Gun/radar rotation taking time: while they are rotating, there is a chance that you get shot and take damage.
- Bullets take time to arrive at the target. So when shooting at a moving opponent, the robot must consider not the movement of the bullet, but the anticipated location of its enemy.
- The radar must actually pointed at enemies to see them. That is, the robots are not operating in a fully-observable environment.
- Guns have a heating/cooling period, meaning that one cannot send a constant barrage or "beam" of bullets.
- Ramming hurts: robots both deal and take damage while ramming into one another.

On the other hand, some “fantastical" elements of Robocode are the fact that:

- Sensors and radars are noiseless.
- Combat takes place in a two-dimensional, rectangular gridworld area.
- Radar information is too “informative." On scanned opponent, the radar detects the opponent’s bearing, velocity, "health" level, and velocity.

Nevertheless, Robocode can be very useful for employing and testing machine learning/evolutionary algorithms. The tank fighting environment provides the need to evolve complex and intelligent behavior in robots. More importantly, Robocode is such a popular framework among programmers that it is extremely easy to find pre-baked robots to train against in our experiments.

4. Goal

Our goal in this paper is to explore and compare the capabilities of various evolutionary machine learning algorithms when applied on Robocode. We focus on two main methodologies: Neuroevolution of Augmenting Topologies(NEAT), and Genetic Programming. Our goal with these two evolutionary algorithms is to evolve robots capable of *consistently* defeat other AIs as well as human players. In the next sections, we will describe relevant previous works regarding employing machine learning on Robocode, give a overview of our methods, display and discuss results, and finally discuss the future implications of our research.

II. Relevant Work

Due to the realistic and interesting nature of Robocode, plenty of research has already been completed in this field. Existing work on Robocode can be divided into two categories based on their use of algorithms, e. Most researchers choose the hard-coded approach, which uses static strategies which don't change over time. Others adopt the evolutionary approach, which starts off as a random controller with poor performance but gradually picks up knowledge and improves over time. In this section, we will review examples of robots implemented with both categories.

1. Hard-coded Approach

A popular and effective way to build a robot controller is to simply hard-code the behavior of the robot. Examples of this category include "RamFire" and "Diamond".

"RamFire" [1] is a simple robot that comes pre-baked into the Robocode package. At the start of battle, RamFire constantly turns its radar to the right in order to find targets. Once RamFire scans an enemy robot, it will lock the enemy down through appropriate radar movements, and approach that enemy at full speed. Once the two robots hit each other, both of them lose energy at the same rate due to the ramming effect. During this process, RamFire also shoots bullets carrying maximum energy to the enemy, which are guaranteed to hit because the two robots are right next to each other. This continues until one of the robot dies.

The RamFire Strategy is effective against robots with low mobility, which are easily "caught" by RamFire. Once the ramming starts, it all comes down to who deals damage faster. RamFire has advantage in this scenario, because it's programmed to shoot bullets with maximum energy, while other robots rarely do so (out of the consideration of saving energy). However, if the opponent has high mobility, then RamFire can be easily "kited" and destroyed before it catches the opponent, due to its predictable movement controller.

"Diamond"[2] is another example of hard-coded robot in Robocode. Unlike RamFire's strategy, the algorithm used by Diamond is very sophisticated and effective against most

types of opponents. The movement controller of Diamond calculates the risk of different movements using *Wave Surfing*, and then applies *Minimum Risk Movement* to choose the safest place to go. The gun uses Dynamic Clustering to find similar states of the opponent, and refers to the kernel density among those states to settle on a firing angle. As a result, Diamond emerges to be one of the top robots in both 1v1 and melee.

Clearly, hard-coded robots take advantage of the intuitive aspects of the game. For example, the radar has a 45° scope and gives information about the position & speed of the opponent on detection, it's relatively easy to write a "perfect" radar controller that locks down the scanned opponent. Other good strategies include firing many low energy bullets (harder to dodge), and moving at full speed all the time for high mobility. Currently, most of the top-ranked robots are hard-coded [3].

2. Evolutionary Approaches

Genetic Programming is the most popular algorithm for evolve robot tanks. The reason is that Robocode is an independent module that only takes the source code of robots, which means the Java source code of all robots needs to be generated and compiled before running a battle. In this case, it's hard to pass in complicated objects such as Neural Networks. On the other hand, Genetic Programming makes it easy because it naturally evolves the source code of robots.

In a paper [4] published in 2003, Jacob Eisenstein described his attempt to apply Genetic Programming to Robocode. Eisenstein observed that Java is not suitable for GP, because it's strongly typed and doesn't have a tree structure. As a solution, he designed a new, lisp-like programming language called TableRex, and used this language to evolve his robots. TableRex incorporates subsumption architecture, and can be easily encoded into fixed length genomes. Eisenstein trained his robots against hand-coded opponents, as well as other individuals in the population. The resulting robots can beat most hand-coded opponents, but are not good enough to win top ranks in Robot Rumble.

Following Eisenstein's work, Yehonatan Shichel and his co-workers published another paper on GP-Robocode. Instead of evolving the entire source file of a robot, they only focused on the actuators of a robot, such as `setForward()`, `turnRight()`, `turnGunRight()`, and `turnRadarRight()`. These functions are self-explanatory and controls the essential movements of a tank robot. In Shichel's experiment, every actuator is associated with a GP function that takes the robot's stats and any radar-scanned information as inputs, and gives the value to that actuator as output. The GP function includes common math constants and functions such as π, e, sin and cos. This incredibly simple technique, coupled with appropriate fitness functions and tournament matching, produced a GPBot that ranked third out of twenty-seven players in the HaikuBots contest. Moreover, their GPBot was the only entry not written by a human.

In the next section, we will introduce NEAT, the genetic algorithm that we used in this project. We will also explain how we overcame the difficulties described above, i.e.

combining Robocode with Neural Networks.

III. Methods

1. Genetic Programming (GP)

As an intermediate goal for our research, we wanted to reproduce Shichel's results on Robocode using GP. Unfortunately, we were unable to locate the exact source code of Shichel's experiment. As such, we were forced to implement our own version combining the Robocode Java API with our prior knowledge of the evolutionary method. In our algorithm, we trained a population evolving robot against 28 hard-coded robots, ranging from simple ones (e.g. `FireRobot`, which doesn't move and only fires) to highly competitive ones (e.g. Diamond, described in section II.1). The fitness function that we employed was quite simple: we simply divided our robot's score with the total score accumulated. A robot's fitness is determined precisely by its percentage of the total score. Other fitness functions we considered included total damage dealt to the opponent.

2. NeuroEvolution of Augmenting Topologies (NEAT)

In a paper [6] published in 2002, Kenneth O. Stanley proposed NeuroEvolution of Augmenting Topologies (NEAT), an evolving-topology Genetics Algorithm that outperforms most fixed-topology networks in competitive learning tasks. NEAT gives good performance because of the following features. First, it starts from simple topologies and gradually complexify them so that no easy solutions are missed. Second, it employs a principled method (innovation numbers) to keep track of evolution histories, which enables efficient and meaningful crossover of different topologies. Finally, it uses speciation to protect structural innovations, which means new structures have time to exploit their full potentials before being forced to compete with existing population. NEAT is particularly suitable for complicated learning tasks, in the sense that it evolves increasingly complicated network topologies over time.

3. Our Configuration of NEAT

As we have seen in the previous section, NEAT is not naturally compatible with Robocode. While NEAT evolves the network topology and weights between nodes, it's hard to serialize them into Java source code, which is what Robocode needs in order to run a battle. As a solution, we had to convert the NEAT network to an XML file and write it to disk. The source code of each robot will then read the corresponding file, parse the XML in it, and reconstruct its NEAT network. This process is illustrated by a block diagram in Figure 4.

NEAT has been implemented as Java Libraries by many researchers. The version we

used is called Another NEAT Java Implementation (ANJI), created by Derek James [7]. ANJI is flexible and well commented, which makes it ideal for our project. The inputs to ANJI network include bias the position, energy, speed, radar and gun positions of the robot itself, as well as the position, energy, speed, heading, and bearing of the enemy robot. If the opponent is not in radar range, all of its data will be 0. The output of ANJI are 5 motor values that will be fed to different motor functions: `setFire()`, `setForward()`, `turnRight()`, `turnGunRight()`, and `turnRadarRight()`. The meanings of these functions are self-evident. After the battle is finished, we calculate the fitness of each participant using the same method as described in GP (see section III.1).

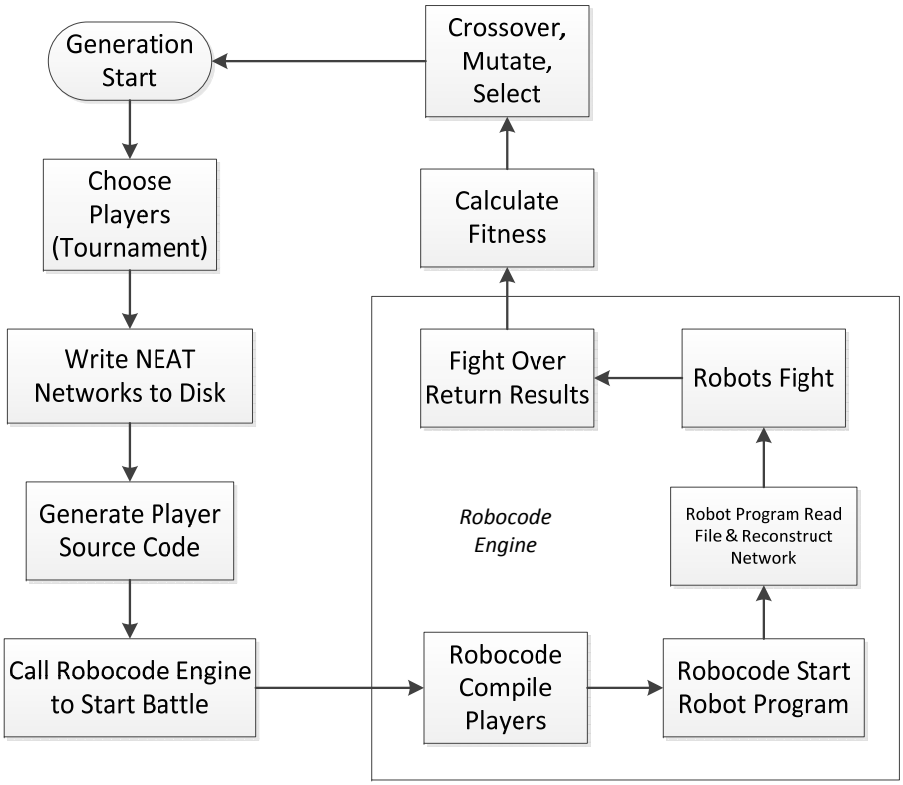

Figure 4: Block Diagram of One Generation

We did three experiments in our project. In the first experiment, we trained our population against the 28 hard-coded robots (see section III.1). In the second experiment, we trained the population against themselves. We let each robot fight against every other robot in the population, and averaged fitnesses over all battles to the get the final fitness for one individual. In the third experiment, we tried a mixture of the previous two methods. We ranked the robot by letting them fight against hard-coded robots first. Then we divided them into groups of 10, and run a tournament inside each group (i.e. the

best robot will fight against the 2nd, 3rd, $\cdots$, 10th, and so on). Then we reorder the rank based on the results of inner tournaments.

Notice the second experiment seems like a lot of work, because the number of battles in each generation is $O(n^2)$, where n is the population size. However, Robocode runs battles really fast. With animations hidden, the Robocode engine can run up to 15 battles per second on our testing laptop. This allows us to run the second experiment with a population size of 100 without worrying about waiting time.

IV. Results

As we stated in our goal, we want to compare the performance of different evolutionary algorithms, including GP and the 3 NEAT experiments. It seems that we only need to compare their fitness values achieved during evolution. However, this is not a very good criterion, because the fitness during evolution are calculated differently for each experiment. In GP and the first NEAT experiment, fitness was averaged over all battles against hard-coded robots. In the second NEAT experiment, fitness was averaged over all battles against other individuals in the population. The third NEAT experiment used hard-coded robots as well as "inner tournaments" in each 10-size group. As a result, it would be unfair to compare them directly using the fitness during evolution.

We need a new "evaluation" fitness function that's fair to every experiment. Since we care about how the evolved robots perform compared to the hard-coded ones, we decided to use the average fitness against the 28 hard-coded robots as our evaluation function. We ran all 4 experiments for 300 generations with a population size of 100, and we evaluated every experiment on every 5th generation. The following figure shows the maximum "evaluated" fitness for each experiment in each evaluation:

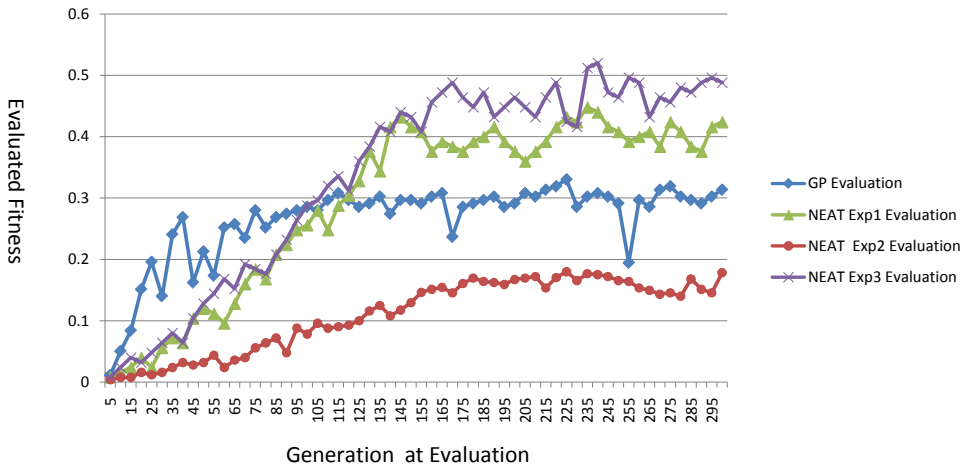

Figure 5: Evaluated Fitness for Each Experiment

The following table shows the maximum “evaluated” fitness for each experiment over these generations:

Table 1: Best Evaluated Results for each Experiment

Experiment	GP	NEAT Exp 1	NEAT Exp 2	NEAT Exp 3
Best Evaluated Fitness	0.3304	0.4482	0.1823	0.5229

V. Discussion

Unfortunately, our results did not meet our expectations. The GP experiment rises very fast but seems to run into a bottleneck very early, and ends up converging to a mediocre state. NEAT Exp1 (“normal" NEAT) manages to overcome the cumbersome bottleneck encountered by GP, but still falls short in terms of fitness. Sadly, NEAT Exp2 with the strict inner tournament module produces the worst results by far. Using the most-complex version of our evolution: version Exp3 combining training with hard-coded robots along with "inner tournaments," we have achieved the best results out of all of our methods. However, even our best evolved tank robot agent was unable to consistently defeat good hard-coded robots. In fact, all of our robots lost against “diamond" every single time. We believe that the main reason for this failure could be that the actions performed by robots in Robocode are far too complex to learn with Neural Networks. This could explain why all the best robot agents are essentially hard coded, finite-state machines. Another factor for our experiments’ mediocre results could be that we did not fully stretch the bounds of our experimentation. Future directions for our work could be incorporating more complex fitness functions, further modification to the NEAT parameters and inner tournament structure, and also incorporating Novelty Search to encourage diverse, unpredictable behaviors.

VI. Acknowledgements

We would like to thank Matthew Nelson for creating Robocode, as well as Lisa Meeden for her support and for teaching us the wonderful subject of Adaptive Robotics.

References

[1] Description and source code of RamFire can be found in this robot wiki website: http://robowiki.net/wiki/SuperRamFire.

[2] Descriptoin and source code of Diamond can be found in this robot wiki website: http://robowiki.net/wiki/Diamond.

[3] See Robot Rumble, the ranking system for Robocode robots around the world. http://literumble.appspot.com/Rankings?game=roborumble.

[4] Jacob Eisenstein. *Evolving Robocode Tank Fighters*, Computer Science and Artificial Intelligence Laboratory Technical Report, 2003.

[5] Yehonatan Shichel, Eran Ziserman, and Moshe Sipper. *GP-Robocode: Using Genetic Programming to Evolve Robocode Players*, Department of Computer Science, Ben-Gurion University, Israel.

[6] Kenneth O. Stanley and Risto Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. Evolutionary Computation Volume 10, Number 2.

[7] ANJI official website: http://anji.sourceforge.net/.