Robocode Documentation

Overview

During the development of the robot we went through several design iterations, as the topic of (Deep) Neural Networks and Machine Learning was new to all of us. We went through four major iterations, with the forth one being able to match more sophisticated bots (e.g. simple Wave Surfer Bots). However, it still struggles against bots like Diamond or DrussGT (Top 2 1v1 Rumble).

Version 1 - Deep Q Learning without EAs

Idea

At first we thought using a Neural Net to train a robot would be sufficient. The idea behind Deep Q Learning is that given a state (position, energy, …) the Neural Net calculates Q Values for each possible action (move right, shoot, …) and the action with the highest Q Value is chosen to be executed.

Execution

We found a robot that uses Deep Q Learning on Github (https://github.com/stevenpjg/QlearningRobocodeNN) and tried to rebuild it with more efficient inputs and more freedom in taking actions. We normalized inputs like position and energy to be of the same order of magnitude and added the option of shooting as a neural net output instead of having it hardcoded.

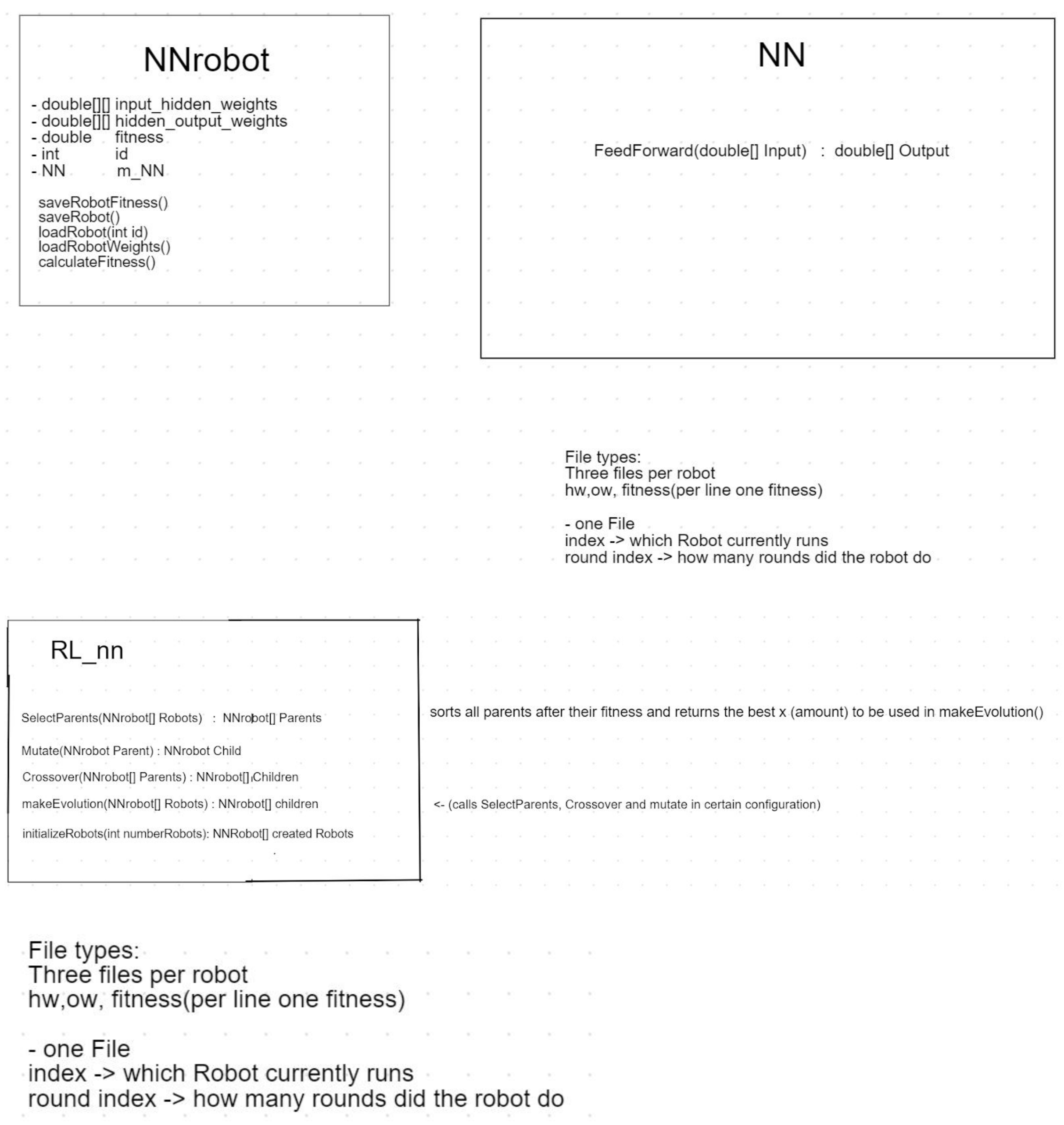
Analysis

We soon were told that we also need to implement Evolutionary Algorithms, so we adapted our robot before analyzing the Neural Net alone.

Version 2 - EAs for Neural Network Weights

Idea

When we searched for ways to combine Evolutionary Algorithms with Neural Nets, we read about optimizing the agents by using EAs on the NNs weights.



NNrobot

```
- double[][] input_hidden_weights
- double[][] hidden_output_weights
- double     fitness
- int        id
- NN         m_NN

saveRobotFitness()
saveRobot()
loadRobot(int id)
loadRobotWeights()
calculateFitness()
```

NN

```
FeedForward(double[] Input)  :  double[] Output
```

File types:
Three files per robot
hw,ow, fitness(per line one fitness)

- one File
index -> which Robot currently runs
round index -> how many rounds did the robot do

RL_nn

```
SelectParents(NNrobot[] Robots)  :  NNrobot[] Parents
Mutate(NNrobot Parent) : NNrobot Child
Crossover(NNrobot[] Parents) : NNrobot[] Children
makeEvolution(NNrobot[] Robots) : NNrobot[] children
initializeRobots(int numberRobots): NNRobot[] created Robots
```

sorts all parents after their fitness and returns the best x (amount) to be used in makeEvolution()

<- (calls SelectParents, Crossover and mutate in certain configuration)

File types:
Three files per robot
hw,ow, fitness(per line one fitness)

- one File
index -> which Robot currently runs
round index -> how many rounds did the robot do

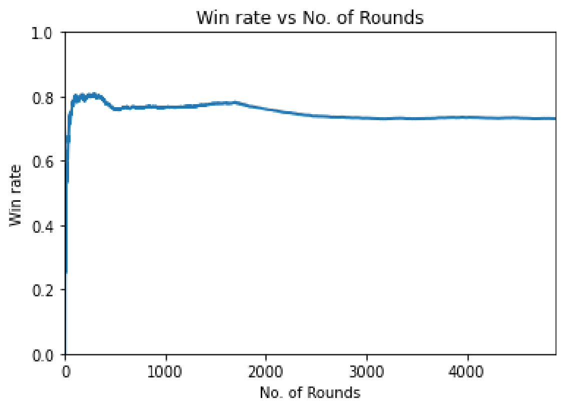
Execution

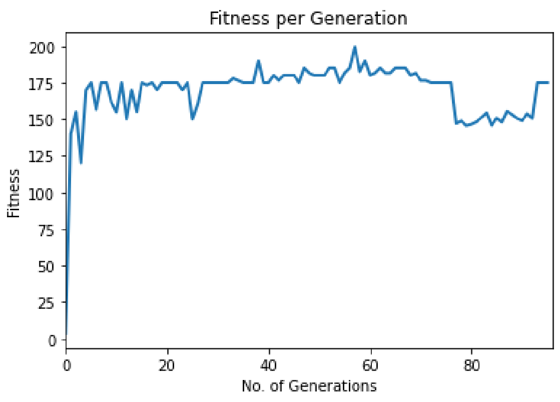
We generated weight files for each robot in a population and updated them after each generation using Selection, Crossover and Mutation. We did not train the Neural Net at all in this approach, instead we let the evolution optimize our weights. The fitness was determined by the sum of the rewards the robot gained during the battles. These rewards were hardcoded, examples would be a positive reward for hitting the enemy robot, negative reward for getting hit or driving into a wall.

Analysis

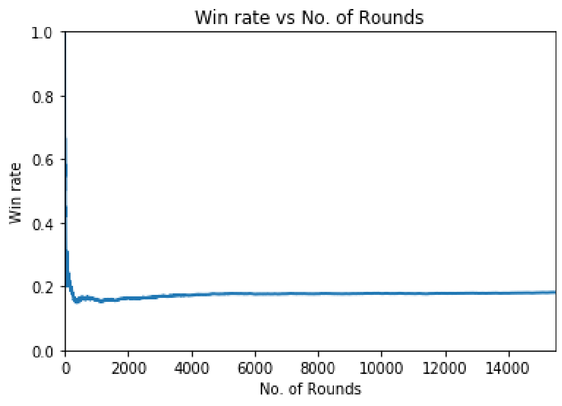
Again we learned that our method was not ideal, since it should be the NNs hyperparameters that are optimized by the Evolutionary Step and not the weights. However, for the simplest sample Robots (such as Trackfire) this approach seemed to work. Against better robots there was no chance.

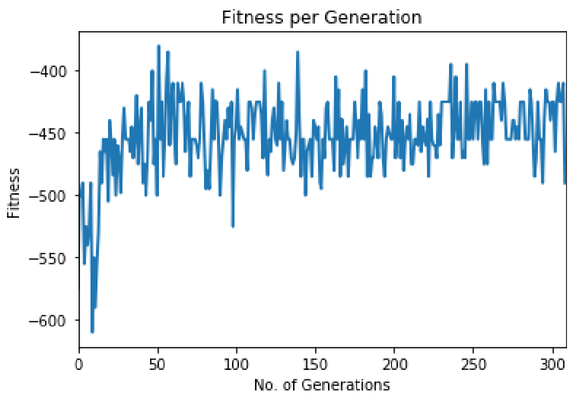
Vs.: Trackfire:





Vs.: Locutus:





Version 3 - EAs for Hyperparameter, Deep Q Learning for Weights

Idea

We switched back to using Deep Q Learning but this time we also generated hyperparameter files for each robot in a generation.

Python

generate Rewards
initialize Weights
make Evolution
Start Robocode Applications

Java

NN (feedforward + backpropagation)
Run Method
Get Battlescore

Files:
Pro Roboter:
Weight Files
Battlescore als Fitness
Hyperparameter
Win Rate

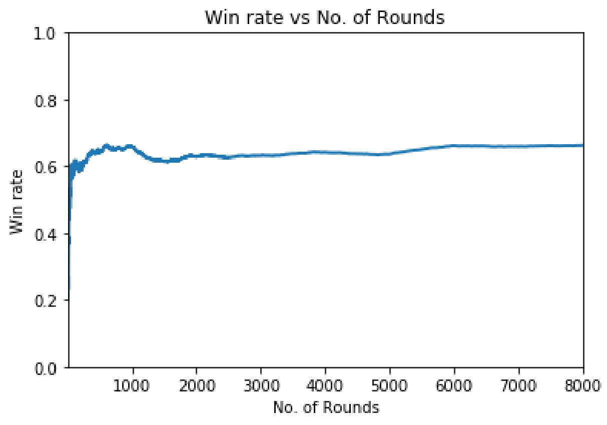
Execution

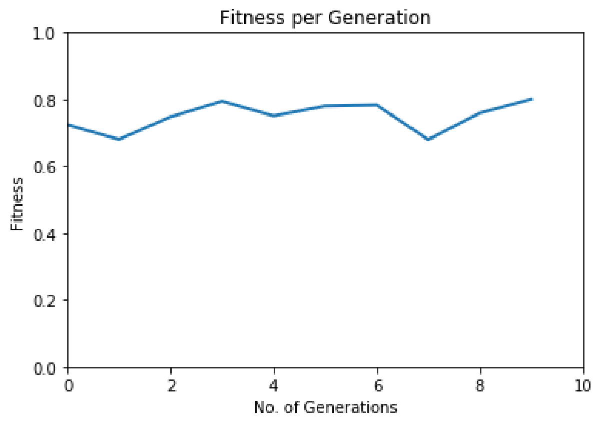
The Java files were now only responsible for running the Neural Net and executing the robots actions. The file handling got outsourced to a Python script that generates weights and hyperparameters and runs the robocode application. Each generation has a training phase and an evaluation phase. The training phase was used to train the weights of the NN, and lasted ~5000 rounds. Evaluation was done in 50 - 100 rounds, and using always the best robot.

Analysis

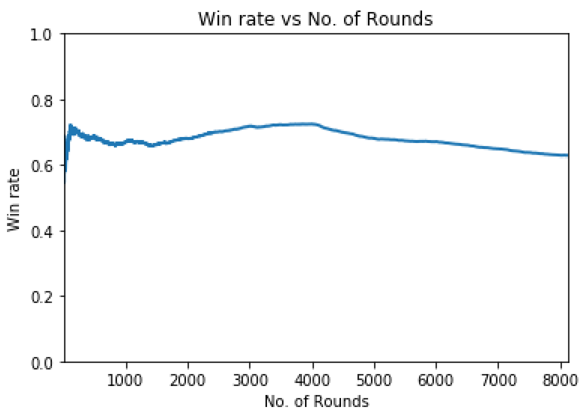
After lots of testing we came to the conclusion that our own robot may not be really improve through deep learning. While we had some success against the Sample Robots, for harder bots the actions were still too limited and a lot was hardcoded, also we possibly did not offer enough input for the neural net to make impactful decisions. The fitness was calculated from the battle result (own score / (own score + enemy score)).

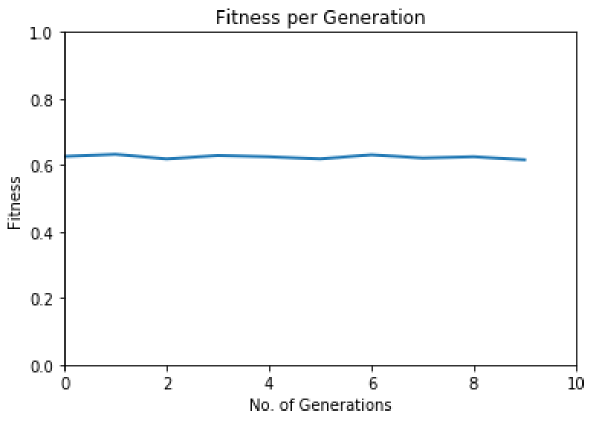
Vs. Sample.Crazy:



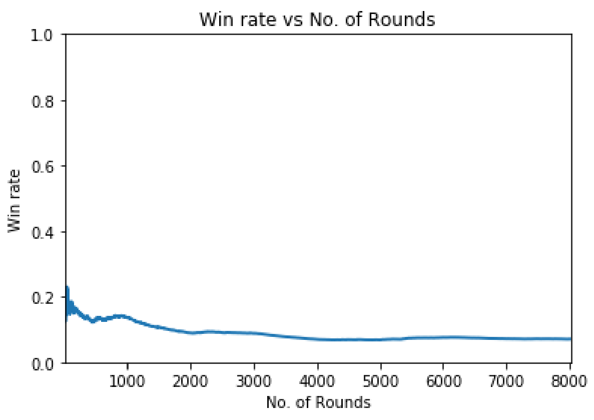


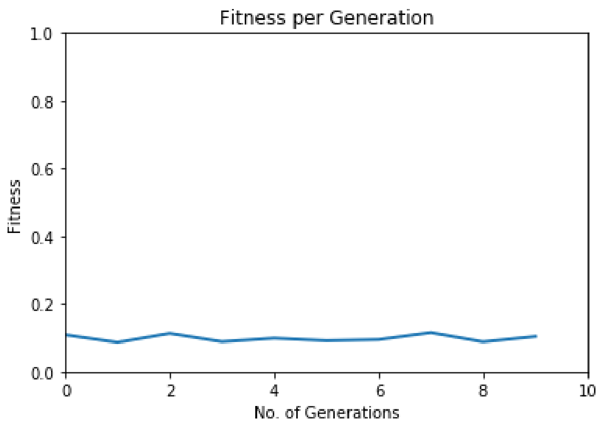
Vs. Trackfire.:





Vs. Locutus.:





Version 4 - EAs for Hyperparameter, Roboneural Library for Network

Idea

After spending weeks on the previous versions of our robot, we recently decided to take the code from robowiki's https://robowiki.net/wiki/NeuralMinimumRiskBot that uses the RoboNeural library. Since this robot was suggested in the moodle forum, we thought that it would give the best results to use the given robot and adapt its hyperparameters.

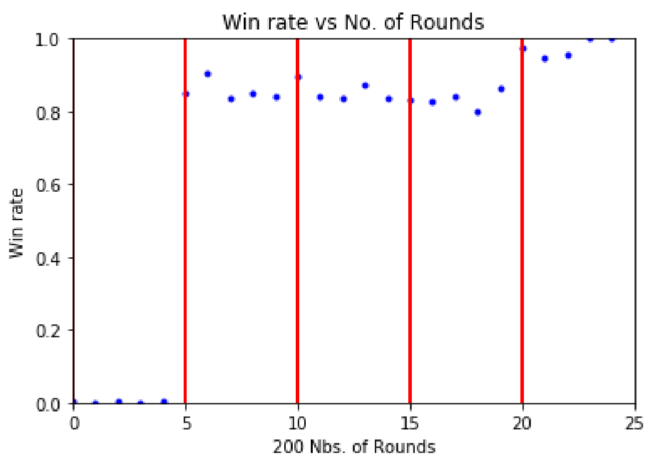
Execution

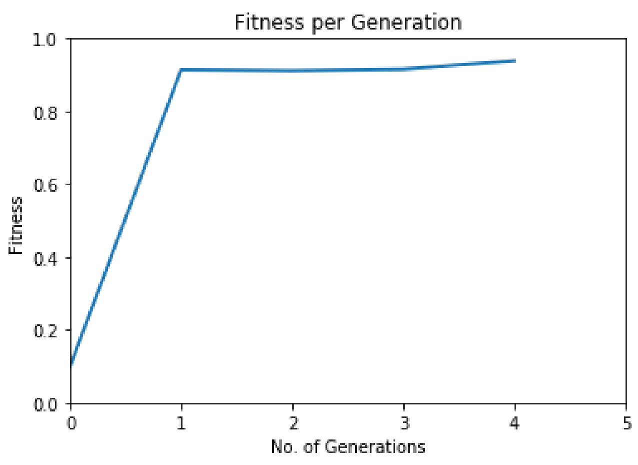
We modified the code from robowiki to add a shooting method that's hardcoded, since most fights ended in the enemy robot being disabled but our robot not winning cause he just moves and couldn't finish off the enemy. Also added were some methods for saving and loading files to read in the hyperparameters and saving the battlescore. The python script was modified to fit the new hyperparameters that are the layer numbers and sizes, the activation functions, the learning rate, the batch size and the rewards for calculating the NNs errors.

Analysis

In hindsight, choosing the battlescore as the fitness value was not the best idea. Since the weights are generated randomly each generation, the performance of the robot mostly depends on how lucky it was with its initial weights. So it seems that sometimes the fittest robot actually decreases in winrate because it could have wrongly signed rewards. A better approach may have been to take the difference in winrate between the first 1000 rounds and the last 1000 rounds of a battle as fitness value.

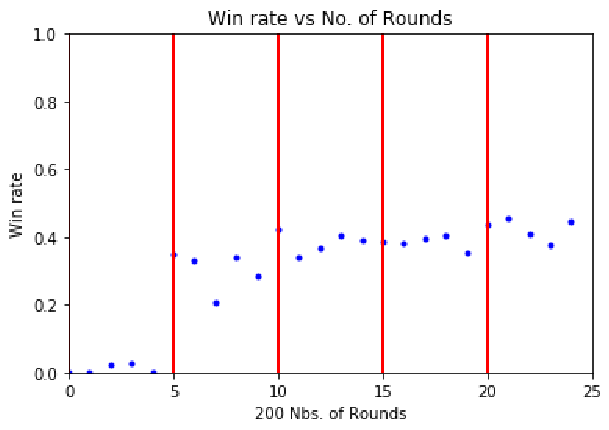
Vs. Crazy (Battlefield Size: 1000 x 1000, 1000 rounds per generation):

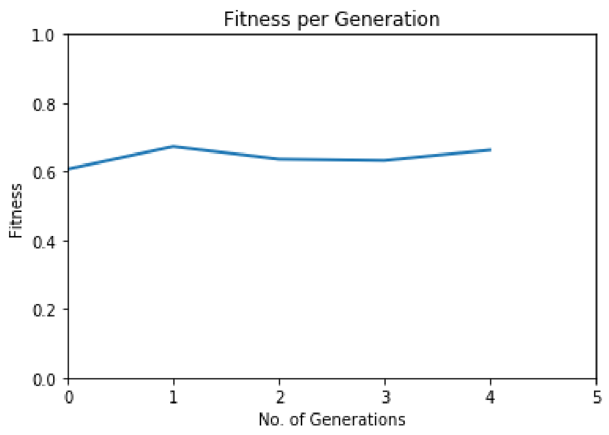




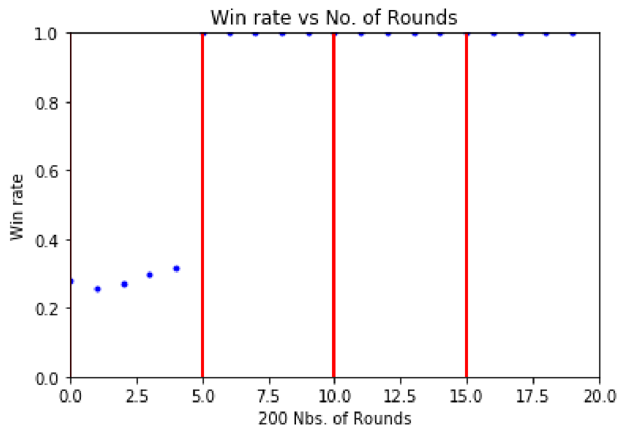
red lines indicate generations

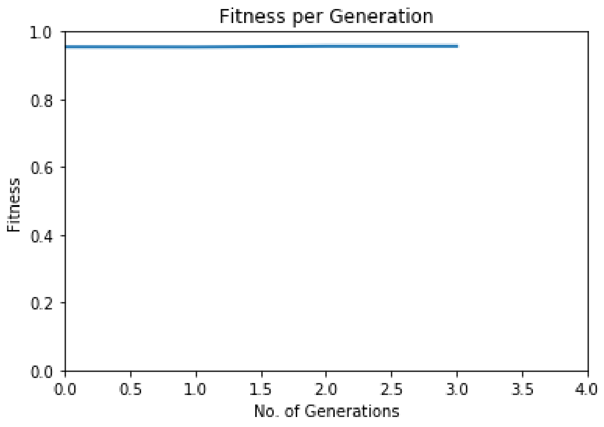
Vs. Crazy (Battlefield Size: 2000 x 2000, 1000 rounds per generation):



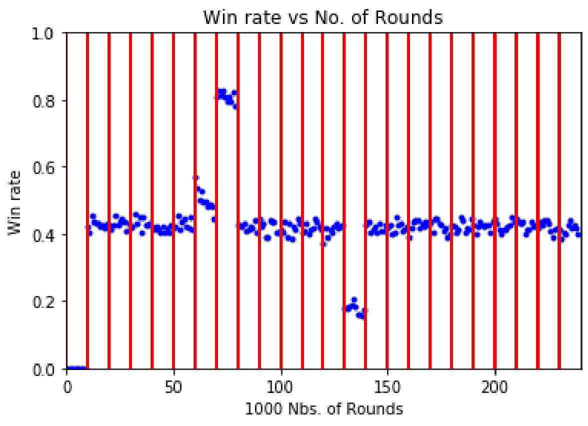


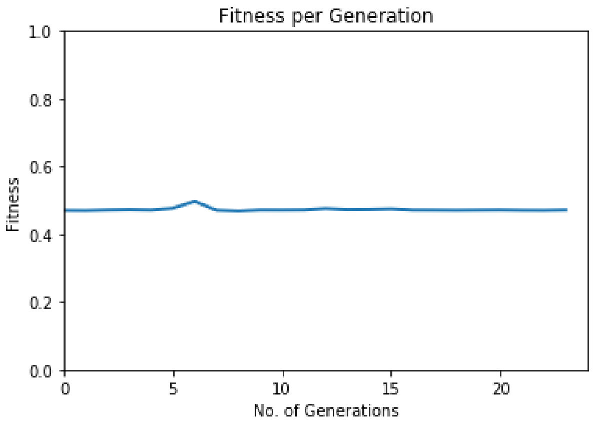
Vs. Crazy (Battlefield Size: 400 x 400, 1000 rounds per generation):





Vs. Locutus.: (Battlefield Size: 1000 x 1000, 10000 rounds per generation)





red lines indicate generations

This robot was trained for 25 Generations with a population size of 20 and 10000 rounds per Generation. This graph shows, that the bot should be capable of beating Locutus (which happened most likely due to very lucky initial weights). However, during a normal learning process, these optimal weights seem to not be reached with our current robot configuration, possibly due to the reasons explained before.

Class Description

NN class (used in Version 1, 2 & 3)

This class contains all the variables for the neural network itself. It has an array of the hidden and output weights, the amount of Input/output nodes aswell as how many nodes the hidden layer contains.
The Neural Net takes the current State of the Robot as Input and outputs the Q values for each possible action. As an activation function for the hidden layer we chose the sigmoid function.
Backpropagation is still implemented for Deep Q learning, but since we switched to Evolutionary Algorithms, it is not used anymore.

NNRobot class (used in Version 1, 2 & 3)

This class represents a single robot. It has a fitness value an ID and a Neural Network (NN) which keeps track of all the previously mentioned variables. Apart from some standard getters and setters it also has a saveRobot() function, which saves all the weights into 2 seperate files. Said files can also be read in this class and saved into the weight arrays of its Neural Network.

When the weight files are read inside of the run method, the amount of rows and columns has to be correct in order for the weight arrays in the Neural Network to be initialized correctly. Starting the robot with an Input/Ouput/HiddenLayer-node amount that doesn't match with the ones in the .txt files leads to errors. Therefor the NNRobot also has a method to initialize the weight files correctly.

Rl_nn class (used in Version 1, 2 & 3)

This class contains the main functionality for the Robot Optimization. Most of the settings can be changed by altering its variables like the hidden Layer neurons, inputNeurons or the Random weights standard deviation for example. In the run method the robot behavior is controlled.

At first X amount of robots are initialized with the initializeRobots() method. It creates an array of NNRobots and fills the weights of its NN with random values. The values follow a normal distribution which can be modified with the global variable "randomWeightsStandardDeviation".

ArlBot class (used in final Version)

The Evolutionary Step

Improving the robots is the responsibility of the makeEvolution() method. It takes an NNRobot array as an input and returns a new array with slightly different robots.
At first the method finds the parents to create the next evolution from.

This is done with another method called selectParents(). It also takes an NNRobot array as an input and returns a certain percentage of those robots again. If the global variable “topParentPercent” is set to 0.9 for example, 90% of the parents will be returned again.
The robots to be returned are not selected randomly however. The method starts by finding the robots with the best fitness. It then calculates which robot has the biggest diversity to the robot with the best fitness and puts it in the first slot of the output array. The rest of this array contains the remaining robots sorted by their fitness in descending order.

Once the parents for the next generation are found, the makeEvolution function generates the next generation through crossovers and mutations with the same size as the parent population. The best performing robot and the most diverse robot stay untouched, while the rest of the population has a random chance to mutate oder get selected for crossing over with another parent. The variable mutationNumber indicates how many children should be produced through mutation, the rest (populationSize - mutationNumber - 2) results from crossovers.
The crossover function takes two parents as input and randomly distributes the values into two new sets. Since the only hyperparameters that have a connections are the layers and the activation functions, those two are handled as one value. For the i input neurons, h hidden neurons and o output neurons, h splits are made for the weights between input layer and hidden layer and o splits are made between hidden layer and output layer.

If a parent is chosen to mutate, each is mutated with a chance that’s set with the variable mutationChance. For example a value of 0.1 means that each hyperparameter hast the chance of 10% to mutate.

Evaluation Graph Jupyter Notebook

To better evaluate robots, we created a jupyter notebook, that reads data from the robots (e.g.: winrate, fitness) and displays them in an easy readable format. This notebook can be found in the ArlBot.data directory.