**Robocode** - **Reinforcement Learning**

**Robot – FizzBot Specifications:**

For Part 2 of the coursework, I have implemented Reinforcement Learning in my FizzBot. I have used values of ALPHA = 0.1 and GAMMA = 0.9. I did experiment with different values of ALPHA and GAMMA but the selected values work best for my implementation.

My state space consists of my robots:

* X position - Quantized to 8 values with 100 values in each level
* Y position - Quantized to 6 values with 100 values in each level
* Energy – Quantized to 3 values with 33 values in each level
* Relative bearing between robot headings - Quantized to 3 values with 60 degrees in each level
* Distance to the enemy - Quantized to 3 values [LOW, FAR, MED]

My robot’s actions consist of:

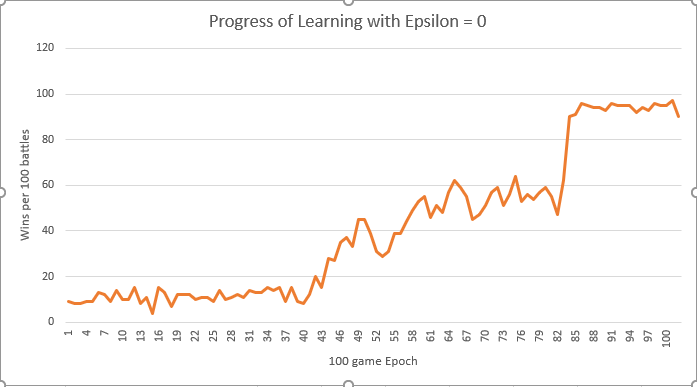
* Aim and Fire
* Move Forward
* Move Backward
* Diagonal Right
* Diagonal Left
* Change Direction

To help with learning and prevent my state-space from exploding, I have implemented a hard-code for dodging a bullet after an enemy bullet hits us.

My robot learns to position itself in a bearing between 60-120 degrees relative to the enemy heading followed by strafing before finally shooting (once it starts shooting it stops strafing and continually shoots). FizzBot achieved the highest win rate of 96% when trained with no exploration. Its second highest win rate is 80% which is done with 50% exploration. While different levels of average win rates at different explorations is being compared in question 3, I found that always taking the greedy move and bootstrapping based on the next greedy move (definition of no exploration, no random actions) resulted in the best performance of my robot. I have provided a brief intuitive to explain why this may be the case in section 3.

Opponent used throughout is TrackFire, it scans and tracks the opponent robot while firing continuously.

***2a) Draw a graph of a parameter that reflects a measure of progress of learning and comment on the convergence of your robot.***



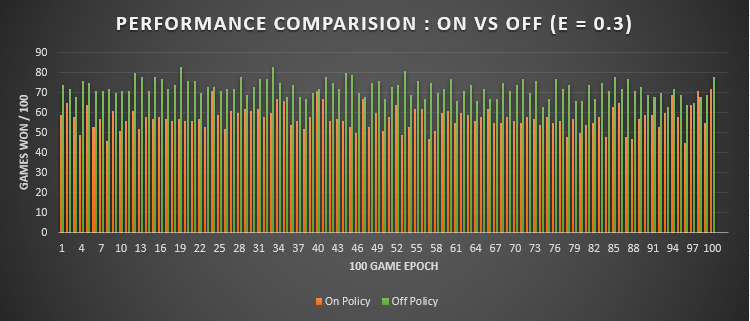
**2a) Figure 1: The number of games won out of 100 games over 10,000 games in total**

In this graph, I have sampled the battles won per 100 games of my robot while it is **being trained**. Here, I have no exploration (epsilon = 0) to observe the learning trend when only greedy moves are taken. Over 10,000 games we can observe that the robot has converged to a win rate of approximately 90% while learning; we can also observe that 8000 games are played before the robot starts to converge to this value.

Figure 12 shows the performance of the robot with the trained LUT. Here we can see that in 1000 games, the average rate of winning per 100 games is 96%.

In the following questions I will investigate the learning performance of on-policy training versus off-policy training and discuss how different values of epsilon (for exploration) affect the learning performance of my robot.

***2b) Using your robot, show a graph comparing the performance of your robot using on-policy vs off-policy learning.***

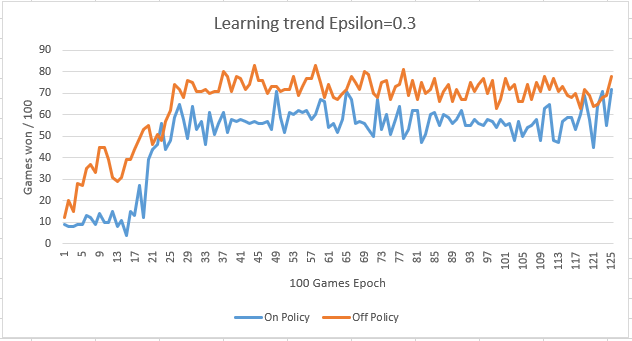


**2b) Figure 2: Performance of the TRAINED robot comparing on versus off policy training. On policy results in a 60% average win rate while off policy results in a 72% average win rate.**

In this version of RL, my robot performs exploratory moves 30% of the time. While the average win rates between on and off policy trained robots are close at 30% exploration, we can see that the off-policy training resulted in better performance than the on-policy trained robot. Off policy has an average win rate of 72% while on policy has a win rate of 63% over 10000 games.

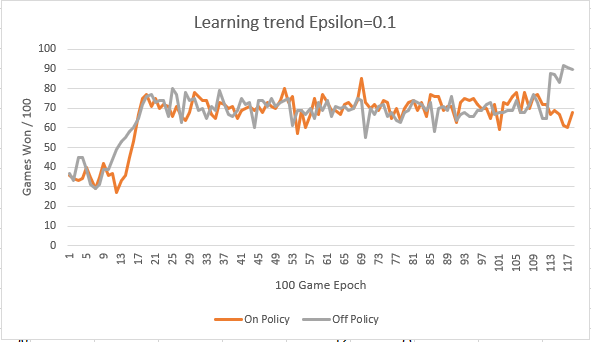
From the class we know that off policy learns faster because we are allowing the agent to sample different policies over time while making updates to its value function as if it were only making greedy moves. By increasing the robot’s exposure to other possible actions, we are exploring the solution space in a greater extent than with on policy or no exploration at all.

We can also look at the trend of the robot’s **learning while it is being trained** (to compare with Figure 1 in the previous question):



**2b) Figure 3: Learning trend of robot during training over 10,000 games for 30% exploration**

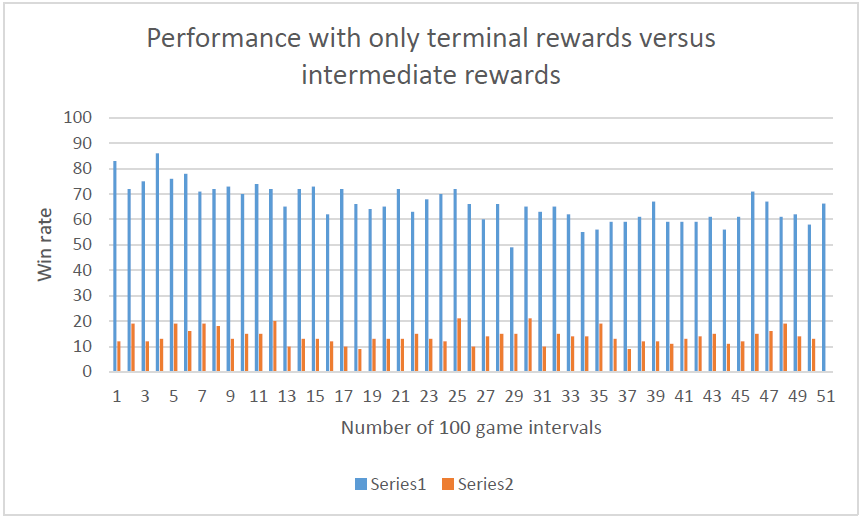
Notice that compared to figure 1, these trends are a lot more volatile even though there is an overall upward trend. This is due to the 30% random actions that are taken by the robot while it explores the solution space. Figure 4 shows the learning curve at 10% exploration – this curve has a more distinct upward trend and while there is volatility, it is less than what is seen at 30% exploration which makes sense.



**2b) Figure 4: Learning trend of robot during training over 10,000 games for 10% exploration**

***2c) Implement a version of your robot that assumes only terminal rewards and show & compare its behaviour with one having intermediate rewards.***

In this version of RL, I implemented learning using only terminal rewards and compared its performance with my earlier robot that used intermediate rewards. Here we can see that having only terminal rewards results in a deteriorated performance. This is because the robot is learning in a dynamic environment and terminal rewards are too late of a feedback signal. Because of the dynamic nature of the environment, the states-action-values of the robot are non-deterministic and so a terminal reward does not allow the agent to understand what actions resulted in this reward very well.



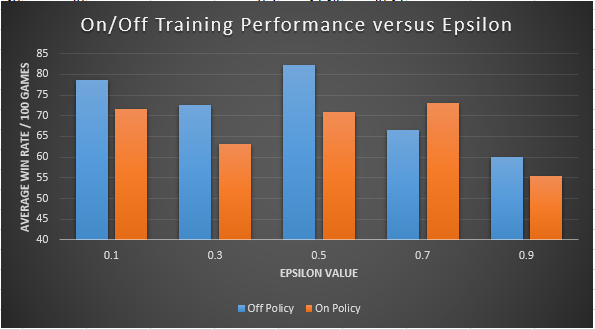
**2c) Figure 5: Performance [Trained] comparing terminal rewards only (series 2) versus intermediate rewards (series 1)**

***3) This part is about exploration. While training via RL, the next move is selected randomly with probability Ɛ and greedily with probability 1- Ɛ***

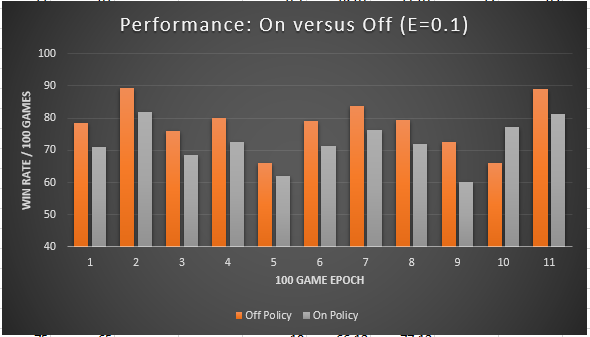
***3a) Compare training performance using different values of Ɛ including no exploration at all. Provide graphs of the measured performance of your tank vs Ɛ***

In the following graphs we investigate the performance of my robot for different values of epsilon. Figure 6 shows the average winning rate of FizzBot for on and off policy at different values of epsilon. We can observe that at epsilon=0.5, FizzBot achieves its highest performance with an off policy win rate at 82% on average (over 5000) games. In Figures 10 and 11 we can also observe that at too high exploration (0.7 then 0.9), the performance of the robot deteriorates, and on/off policy results are not too distinct anymore. We can imagine that too many exploratory moves do not allow the robot to converge to a good solution because there are too many stochastic actions taken during training (at higher epsilons).

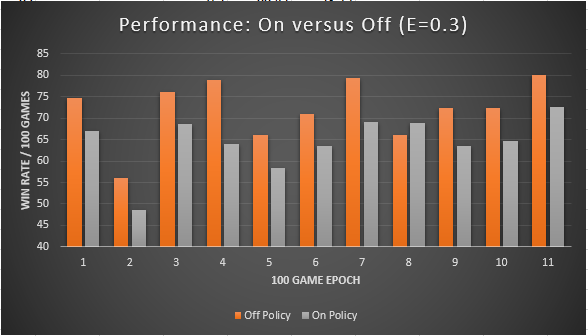
Figures 7-12 show individual graphs of on/off policy performance per epsilon value. Note the win rate is calculated per 100 games over 1000 total games after training has taken place over 50,000 games. Figure 12 is the performance of the robot **with no exploration**. Here the average win rate is 96% which is higher than exploration at 0.1- 0.9. It is possible that in my case training for 50,000 games is not enough for the robot to explore different solution spaces, resulting in a lower trained performance than 0 exploration. This could be investigated more in the next assignment.



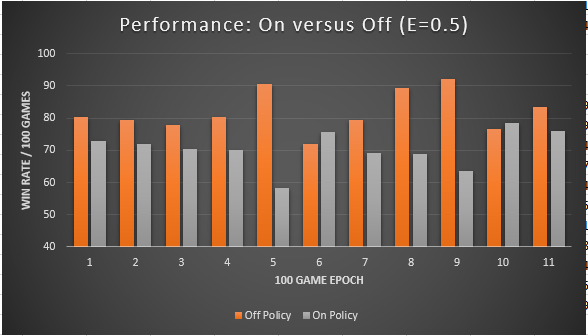
**3a) Figure 6: Performance (on policy and off policy) versus Epsilon. The win rate (per 100 games) was calculated over 5000 games.**



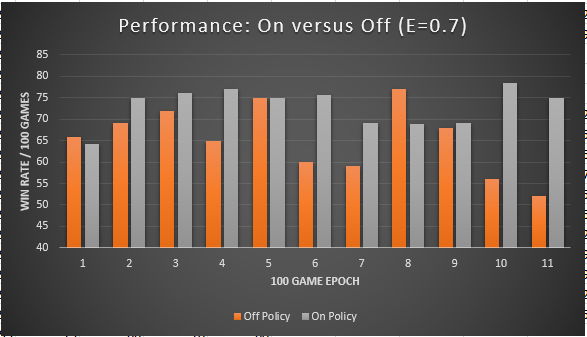
**Figure 7: With epsilon=0.1, off policy performance has an average 78%-win rate versus 72%-win rate for on policy**



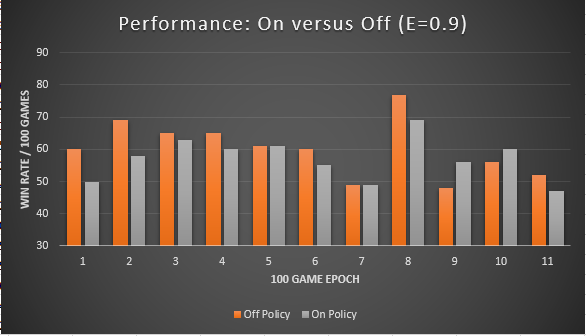
**Figure 8: With epsilon=0.3, off policy performance has an average 72%-win rate versus 63%-win rate for on policy**



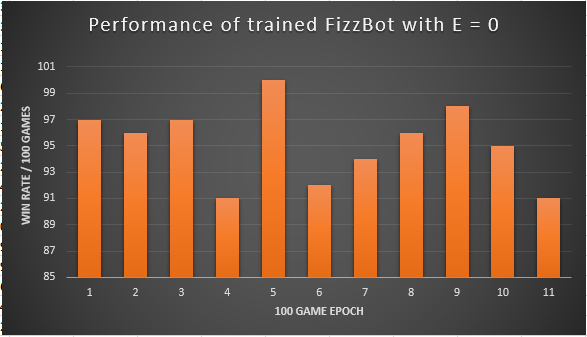
**Figure 9: With epsilon=0.5, off policy performance has an average 82%-win rate versus 72%-win rate for on policy**



**Figure 10: With epsilon=0.7, off policy performance has an average 66%-win rate versus 73%-win rate for on policy**



**Figure 11: With epsilon=0.9, off policy performance has an average 60%-win rate versus 57%-win rate for on policy**



**Figure 12: With epsilon=0.0, the robot win rate is on average 96%. Note: with no exploration there is no notion of off and on policy. The robot is only taking greedy moves.**

**APPENDIX**

**LUT Class**

**package** my.rl.robot;

**import** java.util.Random;

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

//import java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintStream;

//import java.io.Writer;

**import** java.nio.file.Files;

**import** java.nio.file.Paths;

**import** java.util.List;

**public** **class** MyLUT2{

// static declarations

**public** **double**[][][][][][] qTable;

**public** **static** File *game\_stats\_wr*;

**public** **static** File *trained\_lut\_wr*;

**int** lutSize = 1;

**int** featureVectorSize;

**int** [] featureNumValues;

**public** **boolean** trainingMode;

String path1 = "C:/Users/Faizan Mansuri/My Data/robocode\_game\_stats\_onpolicy.txt";

**static** String *path* = "C:/Users/Faizan Mansuri/My Data/trained\_lut.txt";

**public** MyLUT2(**boolean** initializeflag, **boolean** trainingMode,**int** featureVectorSize,**int** [] featureNumValues)

{

// determine size of lut

**this**.featureVectorSize = featureVectorSize;

**this**.featureNumValues = featureNumValues;

**for** (**int** i = 0; i < featureVectorSize; i++) {

lutSize = lutSize \* featureNumValues[i];

System.***out***.println(featureNumValues[i]);

}

qTable = **new** **double**[featureNumValues[5]][featureNumValues[4]][featureNumValues[3]][featureNumValues[2]][featureNumValues[1]][featureNumValues[0]];

**this**.trainingMode = trainingMode;

//System.out.println("I was called");

//if(initializeflag)

initialiseLUT(**this**.trainingMode,initializeflag);

}

/\*\* \* Initialize the look up table to all zeros. \*/

**public** **void** initialiseLUT(**boolean** trainingMode, **boolean** initialflag)

{

Random number = **new** Random();

**double** min = 0.5;

**double** max = 1;

**if** (trainingMode && initialflag)

{

**for** (**int** i = 0; i < featureNumValues[5]; i++)

{ // 5

**for** (**int** j = 0; j < featureNumValues[4]; j++)

{ // 4

**for** (**int** k = 0; k < featureNumValues[3]; k++)

{// 3

**for** (**int** l = 0; l < featureNumValues[2]; l++)

{ // 2

**for** (**int** m = 0; m < featureNumValues[1]; m++)

{ // 1

**for** (**int** n = 0; n < featureNumValues[0]; n++)

{ //0

qTable[i][j][k][l][m][n] = min + (max - min) \* number.nextDouble();

//System.out.println(qTable[i][j][k][l][m][n]);

}

}

}

}

}

}

}

**else**

{

**try** {

load(*path*);

}

**catch** (IOException e)

{ // **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

/\*\* \* A helper method that translates a vector being used to index the look up table

\* \* into an ordinal that can then be used to access the associated look up table element.

\* \* **@param** X The state action vector used to index the LUT

\* \* **@return** The index where this vector maps to - the first row actions for this feature vector

\* \*/

**public** **int** indexFor(**double** [] X)

{ // unused in my implementation of qTable

**return** 0;

}

/\*\* \* **@param** X The input vector. An array of doubles.

\* \* **@return** The value returned by the LUT or NN for this input vector

\* \*/

**public** **double** outputFor(**int** [] state)

{

**assert**(state.length == **this**.featureVectorSize);

**return** qTable[state[5]][state[4]][state[3]][state[2]][state[1]][state[0]];

}

/\*\*

\* \* Given a state vector, return the actionSet which is the corresponding Q values for that feature vector

\* \* **@param** state

\* \* **@return**

\* \*/

**public** **double** [] outputQvaluesForCurrentState( **int** [] state)

{

**assert**(state.length == **this**.featureVectorSize - 1);

System.***out***.println(state[4]+" "+state[3]+" "+state[2]+" "+state[1]+" "+state[0]);

**assert**(state[4] < featureNumValues[5]);

**assert**(state[3] < featureNumValues[4]);

**assert**(state[2] < featureNumValues[3]);

**assert**(state[1] < featureNumValues[2]);

**assert**(state[0] < featureNumValues[1]);

**double** [] actionSet = qTable[state[4]][state[3]][state[2]][state[1]][state[0]];

**return** actionSet;

}

/\*\*

\* \* This method will tell the NN or the LUT the output

\* \* value that should be mapped to the given input vector. I.e.

\* \* the desired correct output value for an input.

\* \* **@param** X The input vector

\* \* **@param** argValue The new value to learn

\* \* **@return** The error in the output for that input vector \*/

**public** **double** train(**int** [] state, **double** argValue)

{

**assert**(state.length == **this**.featureVectorSize);

qTable[state[5]][state[4]][state[3]][state[2]][state[1]][state[0]] = argValue;

**return** 0;

}

/\*\* \* \*/

**public** **void** saveGameStats(**int** numWinGamesPerHundred, **double** rewardsPerHundred)

{

PrintStream saveFile = **null**;

**try** {

saveFile = **new** PrintStream( **new** FileOutputStream( path1,**true** ));

}

**catch** (IOException e) {

System.***out***.println( "\*\*\* Could not create output stream for Reward save file.");

}

//saveFile.append(String.valueOf(numWinGamesPerHundred+" "+rewardsPerHundred) + System.getProperty("line.separator"));

saveFile.append(String.*valueOf*(numWinGamesPerHundred) + System.*getProperty*("line.separator"));

saveFile.close();

}

/\*\*

\* \* A method to write either a LUT or weights of an neural net to a file.

\* \* **@param** argFile of type File. \*/

**public** **void** save()

{

PrintStream saveFile = **null**;

**try** {

*trained\_lut\_wr* = **new** File(*path*);

*trained\_lut\_wr*.createNewFile();

saveFile = **new** PrintStream( **new** FileOutputStream( *trained\_lut\_wr* ));

}

**catch** (IOException e) {

System.***out***.println( "\*\*\* Could not create output stream for LUT save file.");

}

**for** (**int** i = 0; i < featureNumValues[5]; i++)

{ // 5

**for** (**int** j = 0; j < featureNumValues[4]; j++)

{ // 4

**for** (**int** k = 0; k < featureNumValues[3]; k++)

{ // 3

**for** (**int** l = 0; l < featureNumValues[2]; l++)

{ // 2

**for** (**int** m = 0; m < featureNumValues[1]; m++)

{ // 1

**for** (**int** n = 0; n < featureNumValues[0]; n++)

{ //0

saveFile.print(String.*valueOf*(qTable[i][j][k][l][m][n]) + System.*getProperty*("line.separator"));

}

}

}

}

}

}

saveFile.close();

}

/\*\*

\* \* Loads the LUT or neural net weights from file. The load must of course

\* \* have knowledge of how the data was written out by the save method.

\* \* You should raise an error in the case that an attempt is being

\* \* made to load data into an LUT or neural net whose structure does not match

\* \* the data in the file. (e.g. wrong number of hidden neurons).

\* \* **@throws** IOException \*/

**public** **void** load(String argFileName) **throws** IOException

{

FileInputStream inputFile = **new** FileInputStream( argFileName );

BufferedReader br = **new** BufferedReader( **new** InputStreamReader( inputFile ));

**int** count = 0;

List<String> lines = Files.*readAllLines*(Paths.*get*(argFileName));

**if** (lines.size() != lutSize)

{

System.***out***.println("Error loading file, incorrect size");

}

**for** (**int** i = 0; i < featureNumValues[5]; i++)

{ // 5

**for** (**int** j = 0; j < featureNumValues[4]; j++)

{ // 4

**for** (**int** k = 0; k < featureNumValues[3]; k++)

{ // 3

**for** (**int** l = 0; l < featureNumValues[2]; l++)

{ // 2

**for** (**int** m = 0; m < featureNumValues[1]; m++)

{ // 1

**for** (**int** n = 0; n < featureNumValues[0]; n++)

{ //0

qTable[i][j][k][l][m][n] = Double.*parseDouble*(lines.get(count));

count +=1;

}

}

}

}

}

}

br.close();

}

**public** **static** **void** main(String[] args) { // **TODO** Auto-generated method

**int** [] featureNumValues = {6, 3, 6, 8, 3, 3};

// initialized backwards!

**int** featureVectorSize = 6;

**boolean** trainingMode = **true**;

**boolean** initializeflag = **false**;

MyLUT2 lut = **new** MyLUT2(initializeflag,trainingMode, featureVectorSize, featureNumValues);

lut.save();

//String path = "C:\\Users\\Faizan Mansuri\\My Data\\trained\_lut.txt";

**try**

{

lut.load(*path*);

}

**catch** (IOException e)

{ // **TODO** Auto-generated catch block

e.printStackTrace();

}

**for** (**int** i = 0; i < featureNumValues[5]; i++)

{ // 5

**for** (**int** j = 0; j < featureNumValues[4]; j++)

{ // 4

**for** (**int** k = 0; k < featureNumValues[3]; k++)

{ // 3

**for** (**int** l = 0; l < featureNumValues[2]; l++)

{ // 2

**for** (**int** m = 0; m < featureNumValues[1]; m++)

{ // 1

**for** (**int** n = 0; n < featureNumValues[0]; n++)

{ //0

System.***out***.println(i+" "+j+" "+k+" "+l+" "+m+" "+n+": "+lut.qTable[i][j][k][l][m][n]);

}

}

}

}

}

}

}

}

**FizzBot Class**

**package** my.rl.robot;

**import** java.awt.Color;

**import** java.awt.geom.Point2D;

**import** java.util.Random;

**import** robocode.\*;

**import** robocode.util.Utils;

**public** **class** FizzBotRL **extends** AdvancedRobot {

//learning versus trained

**private** **static** **boolean** *Q\_LEARNING* = **false**; //Set this to true for learning

**private** **static** **boolean** *initializeflag* = **true**; //Set this true if training for first time

**private** **static** **double** *EPSILON\_INIT* = 0.0;

**private** **static** **double** *EPSILON* = *EPSILON\_INIT*;

**private** **static** **boolean** *EXPLORATION* = **true**;

**private** **static** **boolean** *ON\_POLICY* = **false**;

**private** **static** **int** *MAX\_GAMES* = 50000;

// actions

**enum** Action {***AIMANDFIRE***, ***FORWARD***, ***BACKWARD***, ***DIAGRIGHT***, ***DIAGLEFT***, ***CHANGEDIRECTION***}

// rewards

**private** **static** **final** **double** ***WALL\_COLLISION\_REWARD*** = -0.9;

**private** **static** **final** **double** ***AWAY\_FROM\_WALL*** = 0.4;

**private** **static** **final** **double** ***ROBOT\_COLLISION\_REWARD*** = -0.4;

**private** **static** **final** **double** ***HIT\_TARGET\_REWARD*** = 0.3;

**private** **static** **final** **double** ***I\_AM\_HIT\_REWARD*** = -1.2;

**private** **static** **final** **double** ***WIN\_GAME\_REWARD*** = 1;

**private** **static** **final** **double** ***LOSE\_GAME\_REWARD*** = -1;

**private** **static** **final** **double** ***LOW\_DIST\_TO\_ENEMY\_REWARD*** = -0.6;

**private** **static** **final** **double** ***MID\_DIST\_TO\_ENEMY\_REWARD*** = 0.7;

**private** **static** **final** **double** ***CLOSE\_BEARING\_REWARD*** = -0.5;

**private** **static** **final** **double** ***MID\_BEARING\_REWARD*** = 0.3;

// end rewards

**private** **static** **double** *ALPHA* = 0.1;

**private** **static** **double** *GAMMA* = 0.9;

**int** direction = 1;

**private** **int** [] prevState = **new** **int**[*featureVectorSize* - 1];

**private** **int** [] currState = **new** **int**[*featureVectorSize* - 1];

**private** **int** [] prevStateAction = **new** **int**[*featureVectorSize*];

**private** **int** [] currStateAction = **new** **int**[*featureVectorSize*];

**private** **int** [] currStateActionGreedy = **new** **int**[*featureVectorSize*];

//Terminal Reward

**public** **int** totalReward;

//Intermediate Reward

**public** **double** rewardPerTurn;

// stats to save on the enemy when I get to scan him

**double** enemyDistance = 0.0;

**double** enemyEnergy = 0.0;

**double** enemyHeading = 0.0;

**double** enemyBearing = 0.0;

// for auto aim

**double** enemyBearingRadians = 0.0;

**double** enemyVelocity = 0.0;

**double** enemyHeadingRadians = 0.0;

**long** lastScanTime = 0;

Action act;

//private double gunTurnAmt;

// static declarations /\* In order from MSB to LSB \*/

// energy: 0, 1, 2

// distance to enemy: 0, 1, 2

// x\_pos: 0, 1, 2, 3, 4, 5, 6, 7

// y\_pos: 0, 1, 2, 3, 4, 5

// bearing: 0, 1, 2

// action: 0 - 5

//set corresponding index in feature vector

**public** **static** **final** **int** ***MY\_ENERGY*** = 5;

**public** **static** **final** **int** ***DIST*** = 4;

**public** **static** **final** **int** ***X\_DIST*** = 3;

**public** **static** **final** **int** ***Y\_DIST*** = 2;

**public** **static** **final** **int** ***ENEMY\_BEARING*** = 1;

**public** **static** **final** **int** ***ACTION*** = 0;

**public** **static** **int** *featureVectorSize* = 6;

**public** **static** **int** [] *featureNumValues* = {6, 3, 6, 8, 3, 3};

// initialized backwards!

**public** **static** MyLUT2 *lut* = **new** MyLUT2(*initializeflag*,*Q\_LEARNING*, *featureVectorSize*, *featureNumValues*);

**public** **int** changecolor = 0;

// learning stats

**public** **static** **double** *rewardPerHundred* = 0.0;

**public** **static** **int** *numTotalGames* = 0;

**public** **static** **int** *numWinGamesPerHundred* = 0;

**public** **static** **int** *wallMargin* = 60;

**public** **void** run()

{

setColors(**null**, **new** Color(0,255,0), **new** Color(255,0,0), Color.***black***, **new** Color(0, 0, 255));

setBodyColor(**new** java.awt.Color(255-changecolor,192 - changecolor,150 - changecolor,100));

//lut = new MyLUT2(Q\_LEARNING, featureVectorSize, featureNumValues);

setAdjustRadarForRobotTurn(**true**);

// keep the radar still while we turn

setAdjustGunForRobotTurn(**true**);

setAdjustRadarForGunTurn(**false**);

//Fancy Implementation - Cosmetics of Robot

**if** (changecolor >= 255)

changecolor = 0;

**else**

changecolor += 5;

// keep the gun still while we turn

turnGunRight(10);

prevState = getState();

// Turn the radar if we have no more turn, starts it if it stops and at the start of round

//if ( getRadarTurnRemaining() == 0.0 )

setTurnRadarRightRadians( Double.***POSITIVE\_INFINITY*** );

scan();

/\*\*\* Q-Learning \*\*\*/

**if** (*Q\_LEARNING*)

{

**while** (**true**)

{

scan();

turnRadarRight(90);

}

}

**else**

{

// playing from trained table

prevState = getState();

**while**(**true**)

{

**if** (getTime() - lastScanTime > 5)

{

turnRadarRight(90);

}

prevStateAction = chooseAction(prevState);

// take action a

performAction((**int**)prevStateAction[0]);

rewardPerTurn = 0;

prevState = getState();

}

}

}

// all mapping for feature vector state occurs here (i.e. int indexes)

**public** **int** [] getState()

{

/\* In order from MSB to LSB \*/

// wall distance: 0, 1

// energy: 0, 1, 2

// distance to enemy: 0, 1, 2

// x\_dist: 0, 1, 2, 3, 4, 5, 6, 7

// y\_dist: 0, 1, 2, 3, 4, 5, 6,

// bearing: 0, 1, 2

// action: 0 - 5

**int** [] state = **new** **int**[*featureVectorSize* - 1];

// energy (low, medium high)

**double** energy = getEnergy();

**if** (energy <= 33)

{

state[***MY\_ENERGY*** - 1] = 0;

}

**else** **if** (energy > 33 && energy <= 66)

{

state[***MY\_ENERGY*** - 1] = 1;

}

**else**

{

state[***MY\_ENERGY*** - 1] = 2;

}

// distance to enemy (low, med, far)

**double** distance = enemyDistance;

**assert**(distance >=0 && distance <=1000);

**if** (distance <= 150)

{

state[***DIST*** - 1] = 0;

}

**else** **if** (distance > 100 && distance <= 333)

{

state[***DIST*** - 1] = 1;

}

**else**

{

state[***DIST*** - 1] = 2;

}

// x pos

state[***X\_DIST*** - 1] = (**int**) (getX() / 100);

// y pos

state[***Y\_DIST*** - 1] = (**int**) (getY() / 100);

// enemy bearing //double bearing = Math.abs(enemyBearing);

**double** bearing = Math.*abs*(normalizeBearing(getHeading() - enemyHeading));

**if** (bearing <= 60)

{

state[***ENEMY\_BEARING*** - 1] = 0;

}

**else** **if** (bearing > 60 && bearing <= 120)

{

state[***ENEMY\_BEARING*** - 1] = 1;

}

**else**

{

state[***ENEMY\_BEARING*** - 1] = 2;

}

**return** state;

}

**public** **int** [] chooseAction(**int**[] prevState2)

{

**double** qValue = 0.0;

**double** maxQValue = Double.***NEGATIVE\_INFINITY***;

**int** actionToTake = 0;

Random number = **new** Random();

**double** rand = number.nextDouble();

**int** [] fullFeatureVector = **new** **int**[*featureVectorSize*];

**assert**(prevState2.length == *featureVectorSize* - 1);

System.*arraycopy*(prevState2, 0, fullFeatureVector, 1, prevState2.length);

**double** [] actionSet = *lut*.outputQvaluesForCurrentState(prevState2);

**assert**(actionSet.length == *featureNumValues*[0]);

/\* determine random or greedily chosen action \*/

**if** (rand < *EPSILON* && *EXPLORATION*)

{

// choose random

actionToTake = (**int**) (0 + (actionSet.length - 0) \* number.nextDouble());

fullFeatureVector[***ACTION***] = actionToTake;

**if** (!*ON\_POLICY*)

{

// choose greedy for off policy update later

**for** (**int** i = 0; i < actionSet.length; i++)

{

qValue = actionSet[i];

**if** (qValue > maxQValue)

{

maxQValue = qValue;

actionToTake = i;

}

}

System.*arraycopy*(fullFeatureVector, 0, currStateActionGreedy, 0, fullFeatureVector.length);

currStateActionGreedy[***ACTION***] = actionToTake;

}

}

**else**

{

// choose greedy

**for** (**int** i = 0; i < actionSet.length; i++)

{

qValue = actionSet[i];

**if** (qValue > maxQValue)

{

maxQValue = qValue;

actionToTake = i;

}

}

fullFeatureVector[***ACTION***] = actionToTake;

System.*arraycopy*(fullFeatureVector, 0, currStateActionGreedy, 0, fullFeatureVector.length);

}

**return** fullFeatureVector;

}

**public** **void** onScannedRobot(ScannedRobotEvent e)

{

//System.out.println("Scanned enemy robot!");

lastScanTime = getTime();

// update state info on enemy

enemyDistance = e.getDistance();

enemyEnergy = e.getEnergy();

enemyHeading = e.getHeading();

enemyBearing = e.getBearing();

enemyBearingRadians = e.getBearingRadians();

enemyVelocity = e.getVelocity();

enemyHeadingRadians = e.getHeadingRadians();

**double** absBearing=e.getBearingRadians()+getHeadingRadians(); //enemies absolute bearing

// Absolute angle towards target

**double** angleToEnemy = getHeadingRadians() + e.getBearingRadians();

// Calculate exact location of the robot

**double** absoluteBearing = getHeading() + e.getBearing();

**double** bearingFromGun = Utils.*normalRelativeAngleDegrees*(absoluteBearing - getGunHeading());

**if** (Math.*abs*(bearingFromGun) <= 3) {

turnGunRight(bearingFromGun);

}**else** {

turnGunRight(bearingFromGun);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Subtract current radar heading to get the turn required to face the enemy, be sure it is normalized

**double** radarTurn = Utils.*normalRelativeAngle*( angleToEnemy - getRadarHeadingRadians() );

radarTurn =

// Absolute bearing to target

getHeadingRadians() + e.getBearingRadians()

// Subtract current radar heading to get turn required

- getRadarHeadingRadians();

setTurnRadarRightRadians(2.0\*Utils.*normalRelativeAngle*(radarTurn));

//Turn the radar

setTurnRadarRightRadians(radarTurn);

execute();

//fire

**double** qdistancetoenemy =0;

**if**((enemyDistance > 0) && (enemyDistance<=250)){

qdistancetoenemy=1;

}

**else** **if**((enemyDistance > 250) && (enemyDistance<=500)){

qdistancetoenemy=2;

}

**else** **if**((enemyDistance > 500) && (enemyDistance<=750)){

qdistancetoenemy=3;

}

**else** **if**((enemyDistance > 750) && (enemyDistance<=1000)){

qdistancetoenemy=4;

}

**if**(qdistancetoenemy==1) {

**if** (getGunHeat() == 0 && Math.*abs*(getGunTurnRemaining()) < 10)

fire(3);

}

**if**(qdistancetoenemy==2){

**if** (getGunHeat() == 0 && Math.*abs*(getGunTurnRemaining()) < 10)

// setFire(firePower);

fire(2);}

**if**(qdistancetoenemy==3){

**if** (getGunHeat() == 0 && Math.*abs*(getGunTurnRemaining()) < 10)

// setFire(firePower);

fire(1);}

//fire

**if** (*Q\_LEARNING*)

{

// choose a from s using policy

System.***out***.println("Choose Action");

prevStateAction = chooseAction(prevState);

// take action a

performAction((**int**)prevStateAction[0]);

// observe some rewards r

**if** (enemyDistance <= 250)

{

rewardPerTurn += ***LOW\_DIST\_TO\_ENEMY\_REWARD***;

}

**else** **if** (enemyDistance > 250 && enemyDistance <= 333)

{

rewardPerTurn += ***MID\_DIST\_TO\_ENEMY\_REWARD***;

}

System.***out***.println("Enemy bearing: "+enemyBearing);

absBearing = Math.*abs*(normalizeBearing(getHeading() - enemyHeading));

System.***out***.println("Angle between robot headings: "+absBearing);

**if** (absBearing <= 60)

{

rewardPerTurn += ***CLOSE\_BEARING\_REWARD***;

}

**else** **if** (absBearing > 60 && absBearing <= 120)

{

rewardPerTurn += ***MID\_BEARING\_REWARD***;

}

**else**

{

rewardPerTurn += ***CLOSE\_BEARING\_REWARD***;

}

**double** x, y;

x = getX();

y = getY();

**if** ( (x <= *wallMargin*) || (x >= (getBattleFieldWidth() - *wallMargin*)) || (y <= *wallMargin*) || (y >= (getBattleFieldHeight() - *wallMargin*)) )

{

rewardPerTurn += ***WALL\_COLLISION\_REWARD***;

}

**else**

{

rewardPerTurn += ***AWAY\_FROM\_WALL***;

}

// observe s'

currState = getState();

currStateAction = chooseAction(currState);

updateLUT(prevStateAction, currStateAction);

System.***out***.println("rewardperTurn: "+rewardPerTurn);

*rewardPerHundred* +=rewardPerTurn;

rewardPerTurn = 0; // reset reward for next state transition

//prevState = currState

System.*arraycopy*(currState, 0, prevState, 0, prevState.length);

}

// Generates another scan event if we see a robot.

// We only need to call this if the gun (and therefore radar)

// are not turning. Otherwise, scan is called automatically.

**if** (bearingFromGun == 0) {

scan();

}

}

**public** **void** updateLUT(**int**[] prevStateAction, **int**[] currStateAction)

{

**double** prevQValue = *lut*.outputFor(prevStateAction);

**double** currQValueTaken = *lut*.outputFor(currStateAction);

**if** (*ON\_POLICY*)

{

// make update based on the action that was performed

*lut*.train(prevStateAction, ((1-*ALPHA*)\*prevQValue + *ALPHA*\*(rewardPerTurn + *GAMMA*\*currQValueTaken)));

}

**else**

{

// make update based on the greedy action regardless of whether you took it

**double** currQValueGreedy = *lut*.outputFor(currStateActionGreedy);

*lut*.train(prevStateAction, ((1-*ALPHA*)\*prevQValue + *ALPHA*\*(rewardPerTurn + *GAMMA*\*currQValueGreedy)));

}

}

**public** **void** performAction(**int** action)

{

act = Action.*values*()[action];

**switch** (act)

{

**case** ***AIMANDFIRE***:

System.***out***.println("Firing");

**double** angleToEnemy = getHeadingRadians() + enemyBearingRadians;

**double** radarTurn = Utils.*normalRelativeAngle*( angleToEnemy - getRadarHeadingRadians() );

// The 36.0 is how many units from the center of the enemy robot it scans.

**double** extraTurn = Math.*min*( Math.*atan*( 36.0 / enemyDistance ), Rules.***RADAR\_TURN\_RATE\_RADIANS*** );

**if** (radarTurn < 0)

radarTurn -= extraTurn;

**else**

radarTurn += extraTurn;

radarTurn =

// Absolute bearing to target

getHeadingRadians() + enemyBearingRadians

// Subtract current radar heading to get turn required

- getRadarHeadingRadians();

setTurnRadarRightRadians(2.0\*Utils.*normalRelativeAngle*(radarTurn));

//Turn the radar

setTurnRadarRightRadians(radarTurn);

**double** absoluteBearing = getHeading() + enemyBearing;

**double** bearingFromGun = Utils.*normalRelativeAngleDegrees*(absoluteBearing - getGunHeading());

turnGunRight(bearingFromGun);

//setTurnGunRight(getHeading() - getGunHeading() + enemyBearing);

**if** (getGunHeat() == 0 && Math.*abs*(getGunTurnRemaining()) < 10)

setFire(Math.*min*(400 / enemyDistance, 3));

// if e farther away, use less fire power. Increase fire power if e is closer.

execute();

**break**;

**case** ***FORWARD***:

System.***out***.println("Going Forward");

setAhead(direction \* 100);

execute();

**break**;

**case** ***BACKWARD***:

System.***out***.println("Going Backward");

setBack(direction \* 100);

execute();

**break**;

**case** ***DIAGRIGHT***:

System.***out***.println("DiagRight");

setTurnRight(45);

setAhead(direction \* 100);

execute();

**break**;

**case** ***DIAGLEFT***:

System.***out***.println("DiagLeft");

setTurnLeft(45);

setAhead(direction \* 100);

execute();

**break**;

**case** ***CHANGEDIRECTION***:

System.***out***.println("ChangingDirection");

direction \*= -1;

setAhead(200 \* direction);

execute();

**break**;

**default**:

**break**;

}

**while**(getDistanceRemaining() != 0 || getTurnRemaining() != 0)

{

execute();

}

}

**public** **void** onHitWall(HitWallEvent e)

{

System.***out***.println("Robot hit a wall");

//rewardPerTurn += WALL\_COLLISION\_REWARD;

**double** xPos=**this**.getX();

**double** yPos=**this**.getY();

**double** width=**this**.getBattleFieldWidth();

**double** height=**this**.getBattleFieldHeight();

**if**(yPos<80)//too close to the bottom

{

turnLeft(getHeading() % 90);

//System.out.println("Get heading");

//System.out.println(getHeading());

**if**(getHeading()==0){turnLeft(0);}

**if**(getHeading()==90){turnLeft(90);}

**if**(getHeading()==180){turnLeft(180);}

**if**(getHeading()==270){turnRight(90);}

ahead(150);

//System.out.println("Too close to the bottom");

**if** ((**this**.getHeading()<180)&&(**this**.getHeading()>90))

{

**this**.setTurnLeft(90);

}

**else** **if**((**this**.getHeading()<270)&&(**this**.getHeading()>180))

{

**this**.setTurnRight(90);

}

}

**else** **if**(yPos>height-80){ //to close to the top

//System.out.println("Too close to the Top");

**if**((**this**.getHeading()<90)&&(**this**.getHeading()>0)){**this**.setTurnRight(90);}

**else** **if**((**this**.getHeading()<360)&&(**this**.getHeading()>270)){**this**.setTurnLeft(90);}

turnLeft(getHeading() % 90);

//System.out.println("Get heading");

//System.out.println(getHeading());

**if**(getHeading()==0){turnRight(180);}

**if**(getHeading()==90){turnRight(90);}

**if**(getHeading()==180){turnLeft(0);}

**if**(getHeading()==270){turnLeft(90);}

ahead(150);

}

**else** **if**(xPos<80){

turnLeft(getHeading() % 90);

//System.out.println("Get heading");

//System.out.println(getHeading());

**if**(getHeading()==0){turnRight(90);}

**if**(getHeading()==90){turnLeft(0);}

**if**(getHeading()==180){turnLeft(90);}

**if**(getHeading()==270){turnRight(180);}

ahead(150);

}

**else** **if**(xPos>width-80){

turnLeft(getHeading() % 90);

//System.out.println("Get heading");

//System.out.println(getHeading());

**if**(getHeading()==0){turnLeft(90);}

**if**(getHeading()==90){turnLeft(180);}

**if**(getHeading()==180){turnRight(90);}

**if**(getHeading()==270){turnRight(0);}

ahead(150);

}

}

/\*\* \* Occurs when my robot collides with another robot \*/

**public** **void** onHitRobot(HitRobotEvent e)

{

System.***out***.println("Robot hit another robot");

rewardPerTurn += ***ROBOT\_COLLISION\_REWARD***;

}

/\*\* \* One of my bullets hit the enemy robot \*/

**public** **void** onBulletHit(BulletHitEvent e)

{

System.***out***.println("Robot shot the enemy");

rewardPerTurn += ***HIT\_TARGET\_REWARD***;

}

/\*\* \* I am hit by a bullet \*/

**public** **void** onHitByBullet(HitByBulletEvent e)

{

System.***out***.println("I got shot");

rewardPerTurn += ***I\_AM\_HIT\_REWARD***;

/\*\*\*\*\*\* Dodge Code \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

setTurnRight(e.getBearing()+90-

30\*direction);

**if**(enemyDistance <= 250)

direction = -1;

**else**

direction = 1;

setAhead((enemyDistance/4+25)\*direction);

execute();

}

**public** **void** onWin(WinEvent e)

{

System.***out***.println("I won");

rewardPerTurn += ***WIN\_GAME\_REWARD***;

updateLUT(prevStateAction, currStateAction);

*numTotalGames*++;

System.***out***.println("Num total games "+*numTotalGames*);

*numWinGamesPerHundred*++;

writeToFile();

}

**public** **void** onDeath(DeathEvent e)

{

System.***out***.println("I died");

rewardPerTurn += ***LOSE\_GAME\_REWARD***;

updateLUT(prevStateAction, currStateAction);

*numTotalGames*++;

System.***out***.println("Num total games "+*numTotalGames*);

writeToFile();

}

**public** **void** writeToFile()

{

**if** ((*numTotalGames* % 100) == 0)

{

// output to file numWinGamesPerHundred

*lut*.saveGameStats(*numWinGamesPerHundred*, *rewardPerHundred*);

*numWinGamesPerHundred* = 0;

*rewardPerHundred* = 0;

// exponentially decay EPSILON //

//EPSILON = EPSILON\_INIT \* Math.exp(-numTotalGames/MAX\_GAMES);

}

**if** (*numTotalGames* == *MAX\_GAMES*)

{

*lut*.save();

}

}

/\*\* \* normalizes a bearing to between +180 and -180

\* \* **@param** angle

\***@return** \*/

**double** normalizeBearing(**double** angle)

{

**while** (angle > 180)

angle -= 360;

**while** (angle < -180)

angle += 360;

**return** angle;

}

//absolute bearing

**double** absoluteBearing(**float** x1, **float** y1, **float** x2, **float** y2) {

**double** xo = x2-x1;

**double** yo = y2-y1;

**double** hyp = Point2D.*distance*(x1, y1, x2, y2);

**double** arcSin = Math.*toDegrees*(Math.*asin*(xo / hyp));

**double** bearing = 0;

**if** (xo > 0 && yo > 0) { // both pos: lower-Left

bearing = arcSin;

} **else** **if** (xo < 0 && yo > 0) { // x neg, y pos: lower-right

bearing = 360 + arcSin; // arcsin is negative here, actuall 360 - ang

} **else** **if** (xo > 0 && yo < 0) { // x pos, y neg: upper-left

bearing = 180 - arcSin;

} **else** **if** (xo < 0 && yo < 0) { // both neg: upper-right

bearing = 180 - arcSin; // arcsin is negative here, actually 180 + ang

}

**return** bearing;

}

}