Part 2a: Reinforcement Learning using LUT

Dustin Johnson - 11338118

2016/10/26

Overview

For this assignment, Reinforcement Learning was used with a robotank, named kBot, to battle an opponent robotank in the RoboCode environment. For purposes of this report, we have narrowed our vision to the opponent "Tracker". Our objective was to construct a Reinforcement Learning agent that demonstrated learning behaviour over time, thereby earning the ability to destroy tracker, more often than not.The architecture for our Reinforcement Learning involved the following:

- five action variables (up, down, left, right, and fire)
- four quantized state variables (X position, Y position, health and distance to the enemy)

Two Reinforcement techniques will be considered in this report:

- Q-learning
- State-Action-Reward-State-Action (SARSA)

Q-learning is considered an off-policy technique. That is, Q-learning will update its Q-values using the Q-value of the next state s_{t+1} and the greedy action a_{t+1}, thereby always assuming an optimal policy. The iterative equation is provided below:

$$Q_{t+1}(s_t, a_t) \leftarrow (1-\epsilon)Q_t(s_t, a_t) + \epsilon[r_{t+1} + \gamma max_a Q_t(s_{t+1}, a)] \tag{1}$$

However, SARSA is an on-policy technique in that it updates its Q-values using the Q-value of the next state s_{t+1} and the current policy's action a, as shown below:

$$Q_{t+1}(s_t, a_t) \leftarrow (1-\epsilon)Q_t(s_t, a_t) + \epsilon[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1})] \tag{2}$$

The SARSA method will aim to determine the true value function of the dynamic system, whereas the Q-learning approach will not be restricted to such a policy. The ϵ for each equation is known as the epsilon-greedy strategy, which enables a balance between exploration and exploitation of the environment. Each technique will be outlined in this report and contain its corresponding performance measures in a series of 10,000 battles. After a comparison of these techniques, the balance between exploration and exploitation will be examined by varying ϵ.

Part 2: Performance Measures

convergence of Q-learning

We begin by graphing the ratio of wins to losses plotted over every 400 battles, out of a total of 10,000 battles. Figure 1 outlines the result performance measures for the off-policy Q-learning method. The epsilon value for this run was 0.001. It is clear that the volatility of the learning process is quite high, although the performance indeed improves at the beginning then oscillates at 0.55, which is greater than a probability of 0.5 of winning. This provides indication that learning is occurring, but rather slowly as the off-policy aspect may be widening its view from a true value function. To reach a level of convergence, more iterations will be required to determine whether or not the probability of each action and time in each state is approaching a steady state.

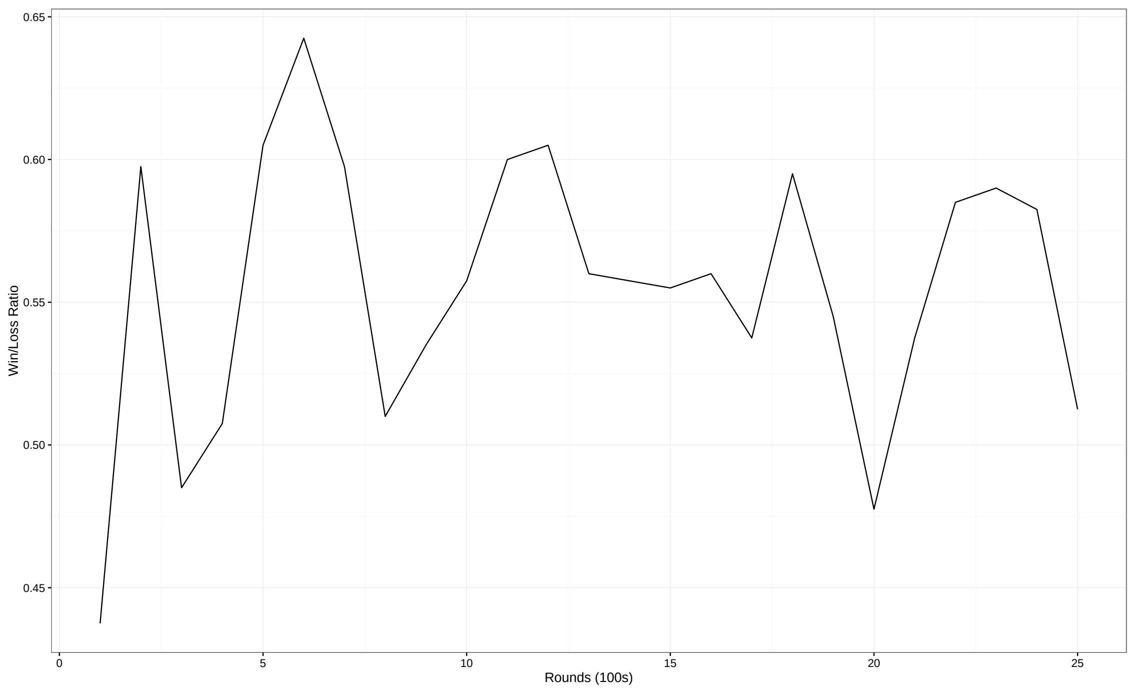

Figure 1: Win/Loss ratio for Q-learning over 10,000 iterations plotted ever 400 battles.

On-Policy vs Off-Policy

Figure two outlines the two techniques described in the overview section of this report. Clearly, the volatility of the win/loss ratio is lower for on-policy (SARSA) approach (denoted in magenta). SARSA is improved gradually then oscillates around a win/loss ration of 0.6 after 1200 rounds establishing a level of convergence far quicker then that of the off-policy approach (Q-learning). This is as expected - Sarsa will learn to be careful in an environment where exploration is costly, whereas Q-learning learns about the policy that doesn't explore and only takes optimal (as estimated) actions. For this reason, Sarsa's careful control over exploration has enabled it to converge faster (learn faster) in the current environment where the robotank Tracker likely has consistent moves to be taken advantage of.

Terminal Rewards Only vs Terminal and Intermediary Rewards

Figure 3 demonstrates two reward scenarios:

1. Terminal only rewards
2. Intermediary and terminal rewards

Quite noticeably, the plot with no terminal rewards performs worse and perhaps coincidentally, behaves in a reciprocal-type nature to that of the learner with intermediary rewards. This makes intuitive sense due to the nature of the game being played. As this tank may take a considerable amount of time and perform many actions in many states before the game ends (when a win or death occurs), then reaching a level of convergence could take a considerable amount of time. As denoted by the red plot, intermediary

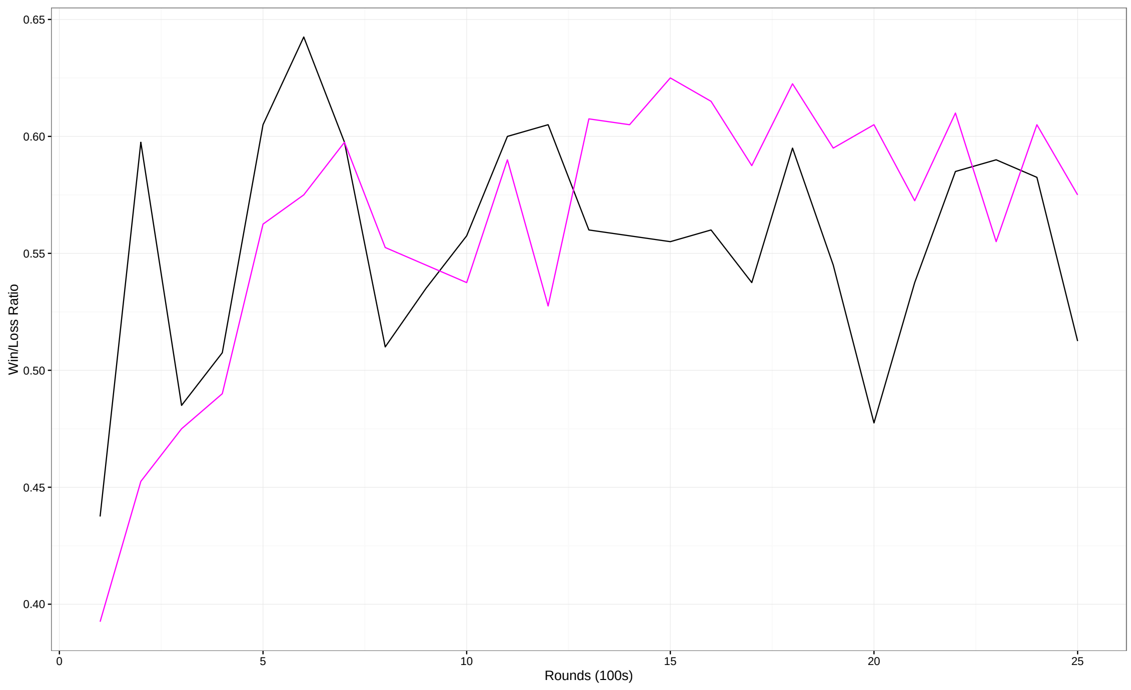

Figure 2: Win/Loss ratio for Q-learning (off-policy) denoted in black compared to SARSA (on-policy) denoted in magenta. 10,000 battles were run plotted ever 400 battles.

rewards keep the tank on track and provide incentive on how to act during the possibly lengthy game play. Using this approach, it is evident that the tank learns quicker and improves the progress of reaching its goal of destruction.

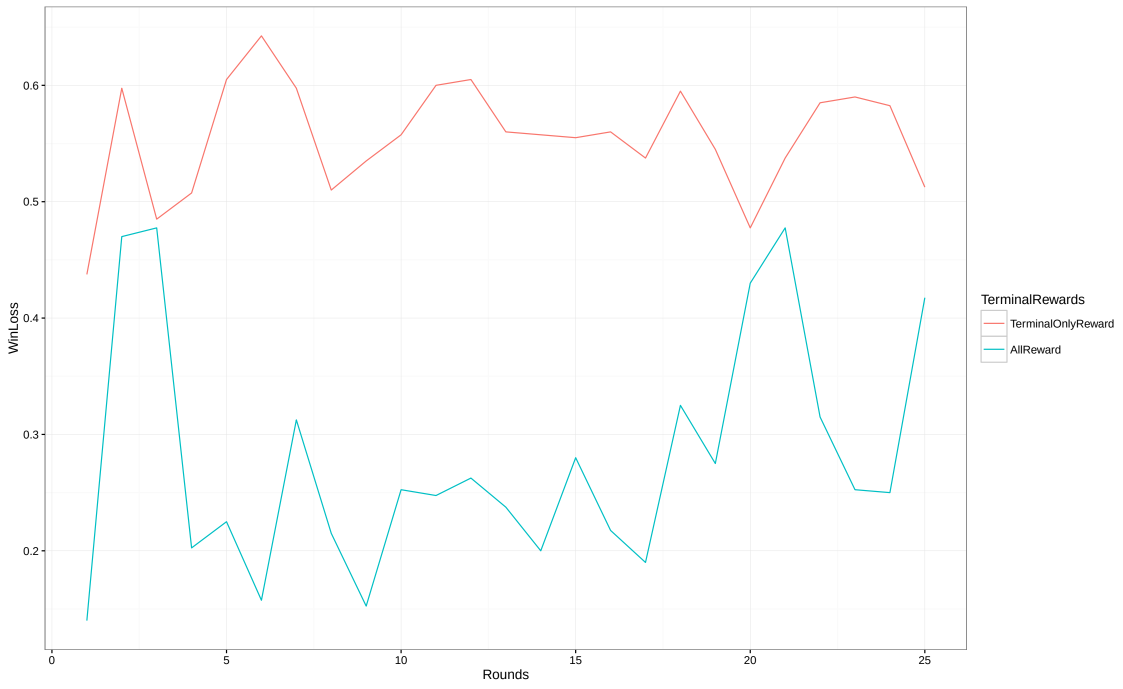

Figure 3: Win/Loss ratio for Q-learning over 10,000 iterations plotted ever 400 battles. Blue denotes learning with only terminal rewards, while the red denotes learning with all intermediary and terminal rewards.

Part 3: Exploration

ϵ determines the amount the Reinforcement Learner is willing to forgo to exploration as opposed to exploitation. The benefit of exploration is to enable the learner to try difference approaches that could lead to improved performance. On the other hand, too much exploration will cause the agent to converge very slowly to a true value function. If we decide to provide the learner no exploration at all, we have arrived at the equilibrium between Q-learning and SARSA - a deterministic Reinforcement learner. The question remains, what is the best ϵ? Figure 4 provides a plot comparing four different values of the greedy epsilon parameter.

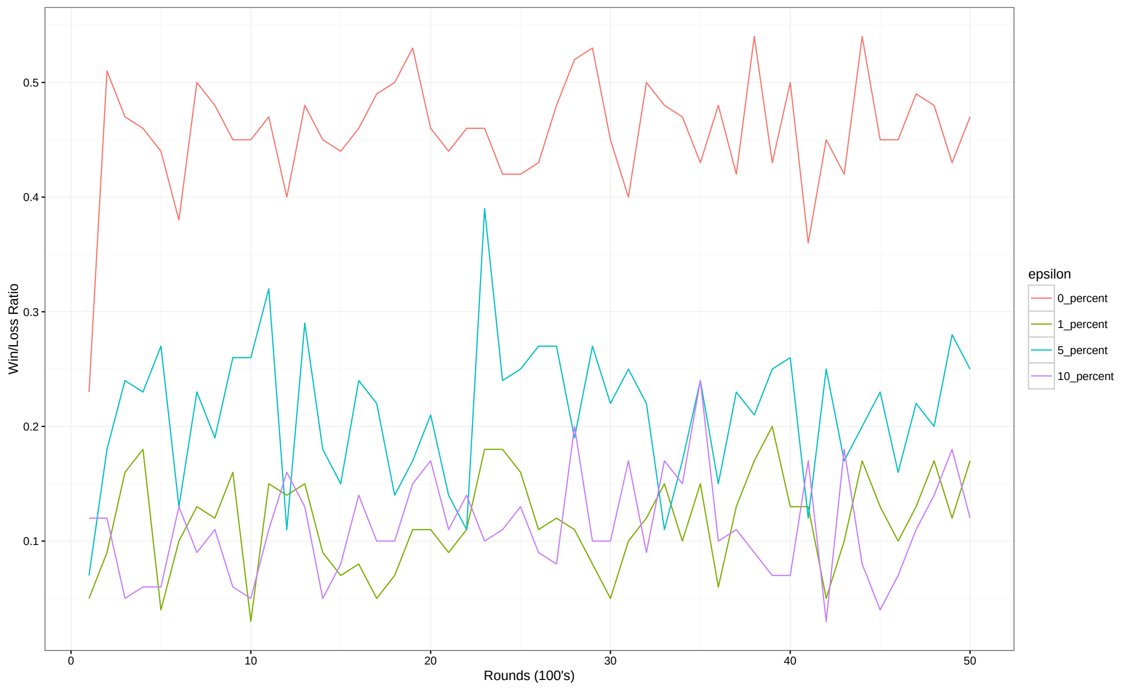

Figure 4: Win/Loss ratio for Q-learning with varying epsilon over 5000 battles plotted every 100 battles.

Clearly, the zero epsilon strategy outperforms with its win/loss ratio over the learning period. This reinforces our previous insight gained from 2(b) regarding on-policy and off-policy approaches. When the epsilon is zero, the Q-learning and SARSA are equivalent, thereby demonstrating similar performance to that of 2(b). Interestingly, the one percent and 10 percent behave vary similarly, while the 5 percent epsilon has the second best performance regarding its win/loss ratio.

Java Code

```
package kinz;

import robocode.AdvancedRobot;
import robocode.BattleEndedEvent;
import robocode.Bullet;
import robocode.BulletHitEvent;
import robocode.DeathEvent;
import robocode.HitByBulletEvent;
import robocode.HitRobotEvent;
import robocode.HitWallEvent;
import robocode.RobocodeFileWriter;
import robocode.RobotStatus;
import robocode.ScannedRobotEvent;
import robocode.StatusEvent;
import robocode.WinEvent;

import java.awt.*;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;

public class kBot extends AdvancedRobot {

static String wlFile = "winLossOut1.txt";

double reward = 0;

boolean sarsaGo = false;


static double alpha = 0.2;
static double gamma = 0.01;
//static double epsilon = 0.01;
static double epsilon = 0.01;

int moveDirection = 1;
int currentAction = 0;
boolean enemyLockedOn = false;
boolean actionComplete = false;
boolean actionInProgress = false;
boolean moveComplete = false;
boolean noShotsRemaining = false;
double enemyAbsBearing;
double gunAngleDifference;
double enemyDistance;
double health = 100;

int win;
int death;
```

```
static int xQuantize = 8;
static int yQuantize = 6;
int binBearing = 12;


double pX = 800, pY=600;

double bulletSuperWeak = 0.5;
double bulletWeak = 1;
double bulletMedium = 2;
double bulletStrong =3;
Bullet bullet;

int xBinned;
int yBinned;


int xQuantized;
int yQuantized;
int distanceQuantized;
int healthQuantized;
int binDistanceQuantized = 3;
int binHealthQuantized = 3;

double temp1;
double temp2;


ArrayList<Integer> state = new ArrayList<Integer>();
ArrayList<Integer> nextState = new ArrayList<Integer>();
ArrayList<Integer> prevState = new ArrayList<Integer>();
ArrayList<Integer> winLoss = new ArrayList<Integer>();

static Qtable q = new Qtable(4, 5, false);


public void onStatus(StatusEvent e)
{
if(actionComplete)
{
if(sarsaGo == false)
{
health = e.getStatus().getEnergy();

xQuantized = q.quantize(e.getStatus().getX(), 0.0, 800.0, xQuantize - 1);
yQuantized = q.quantize(e.getStatus().getY(), 0.0, 600.0, yQuantize - 1);
distanceQuantized = q.quantize(enemyDistance, 0, 1000, binDistanceQuantized - 1);
healthQuantized = q.quantize(health, 0, 100, binHealthQuantized - 1);


nextState.set(0, xQuantized);
nextState.set(1, yQuantized);
nextState.set(2, distanceQuantized);
nextState.set(3, healthQuantized);
```

```
temp1 = (1-alpha) * q.QTable[state.get(0)][state.get(1)][state.get(2)][state.get(3)][currentAction];

double max = -9999999;
for(int i = 0; i < 5; i++)
{
if(q.QTable[nextState.get(0)][nextState.get(1)][nextState.get(2)][nextState.get(3)][i] > max)
{
max = q.QTable[nextState.get(0)][nextState.get(1)][nextState.get(2)][nextState.get(3)][i];
}
}

temp2 = alpha * (reward + max);

q.QTable[state.get(0)][state.get(1)][state.get(2)][state.get(3)][currentAction] = temp1 + temp2;
reward = 0;
}

actionComplete = false;

}

RobotStatus status =  e.getStatus();

if(enemyLockedOn)
{

if(!actionComplete && !actionInProgress)
{
moveComplete = false;
health = e.getStatus().getEnergy();

xQuantized = q.quantize(e.getStatus().getX(), 0.0, 800.0, xQuantize - 1);
yQuantized = q.quantize(e.getStatus().getY(), 0.0, 600.0, yQuantize - 1);
distanceQuantized = q.quantize(enemyDistance, 0, 1000, binDistanceQuantized - 1);
healthQuantized = q.quantize(health, 0, 100, binHealthQuantized - 1);


state.set(0, xQuantized);
state.set(1, yQuantized);
state.set(2, distanceQuantized);
state.set(3, healthQuantized);


//actionInProgress = true;

Random r = new Random();

double rand = r.nextDouble();

int actionDecided =0;
if(rand <= 1-epsilon)
{
double max = -9999999;
for(int i = 0; i < 5; i++)
{
if(q.QTable[state.get(0)][state.get(1)][state.get(2)][state.get(3)][i] > max)
```

```
{
max = q.QTable[state.get(0)][state.get(1)][state.get(2)][state.get(3)][i];
actionDecided = i;
}
}
}
else
{
actionDecided = r.nextInt(5);
}

currentAction = actionDecided;


if(sarsaGo == true)
{
health = e.getStatus().getEnergy();

xQuantized = q.quantize(e.getStatus().getX(), 0.0, 800.0, xQuantize - 1);
yQuantized = q.quantize(e.getStatus().getY(), 0.0, 600.0, yQuantize - 1);
distanceQuantized = q.quantize(enemyDistance, 0, 1000, binDistanceQuantized - 1);
healthQuantized = q.quantize(health, 0, 100, binHealthQuantized - 1);


nextState.set(0, xQuantized);
nextState.set(1, yQuantized);
nextState.set(2, distanceQuantized);
nextState.set(3, healthQuantized);

temp1 = (1-alpha) * q.QTable[state.get(0)][state.get(1)][state.get(2)][state.get(3)][currentAction];


double max = q.QTable[nextState.get(0)][nextState.get(1)][nextState.get(2)][nextState.get(3)][curren

temp2 = alpha * (reward + max);

q.QTable[state.get(0)][state.get(1)][state.get(2)][state.get(3)][currentAction] = temp1 + temp2;
reward = 0;
}

switch(currentAction)
{

// move up one block
case 0:
double heading = status.getHeading();

if(heading <= 180)
{
turnLeft(heading);
}
else
{
turnRight(360 - heading);
}
actionInProgress = true;
```

```
break;

// move down one block
case 1:
heading = status.getHeading();

if(heading <= 180)
{
turnRight(180-heading);
}
else
{
turnLeft(heading-180);
}
actionInProgress = true;
break;

// move left one block
case 2:
heading = status.getHeading();

if(heading <= 90 && heading >= 0)
{
turnLeft(90 + heading);
}
else if(heading >= 270 && heading <= 359)
{
turnLeft(heading-270);
}
else if(heading < 270 && heading > 90)
{
turnRight(270 - heading);
}
actionInProgress = true;
break;

// move right one block
case 3:
heading = status.getHeading();

if(heading <= 90 && heading >= 0)
{
turnRight(90-heading);
}
else if(heading >= 270 && heading <= 359)
{
turnRight((360-heading)+90);
}
else if(heading < 270 && heading > 90)
{
turnLeft(heading-90);
}

actionInProgress = true;
break;
```

```
case 4:


if(getGunHeat() > 0)
{
noShotsRemaining = true;
actionInProgress = false;
actionComplete = true;
break;
}

fire(3.0);
//execute();


actionInProgress = false;
actionComplete = true;

break;


case 5:

turnGunRight(360.0/(double)binBearing);
actionInProgress = false;
actionComplete = true;

gunAngleDifference = robocode.util.Utils.normalRelativeAngle(enemyAbsBearing- getGunHeadingRadians()

if(Math.abs(gunAngleDifference) <= (binBearing*Math.PI/180))
{
//reward += 20;
reward = 20;
}
else
{
reward = -1;
}

break;

case 6:
turnGunLeft(360.0/(double)binBearing);
actionInProgress = false;
actionComplete = true;

gunAngleDifference =robocode.util.Utils.normalRelativeAngle(enemyAbsBearing- getGunHeadingRadians())

if(Math.abs(gunAngleDifference) <= (binBearing*Math.PI/180))
{
//reward += 20;
reward = 20;
}
else
```

```
{
reward = -1;
}
break;

}
}


if(!actionComplete && actionInProgress)
{
switch(currentAction)
{
case 0:
if(getTurnRemaining() == 0 && !moveComplete)
{
double upY = ((pY/yQuantize)*(yQuantized + 1)) + (pY/yQuantize)/2;
double dist = upY - status.getY();
setAhead(dist);
//execute();

moveComplete = true;
}



if(getDistanceRemaining() == 0)
{
actionInProgress = false;
actionComplete = true;

break;
}

break;


case 1:
if(getTurnRemaining() == 0 && !moveComplete)
{
double upY = ((pY/yQuantize)*(yQuantized - 1)) + (pY/yQuantize)/2;
double dist = upY - status.getY();
setAhead(dist);
//execute();

moveComplete = true;

}

if(getDistanceRemaining() == 0)
{
actionInProgress = false;
actionComplete = true;

break;
}
```

```
break;


case 2:

if(getTurnRemaining() == 0 && !moveComplete)
{
double upX = ((pX/xQuantize)*(xQuantized - 1)) + (pX/xQuantize)/2;
double dist = upX - status.getX();
setAhead(-dist);
//execute();

moveComplete = true;

}

if(getDistanceRemaining() == 0)
{
actionInProgress = false;
actionComplete = true;

break;
}

break;


case 3:

if(getTurnRemaining() == 0 && !moveComplete)
{
double upX = ((pX/xQuantize)*(xQuantized + 1)) + (pX/xQuantize)/2;
double dist = upX - status.getX();
setAhead(dist);
//execute();

moveComplete = true;

}

if(getDistanceRemaining() == 0)
{
actionInProgress = false;
actionComplete = true;

break;
}

break;


}
}
}
}
```

```
/**
 * run:  Tracker's main run function
 */
public void run() {

state.add(0);
state.add(0);
state.add(0);
state.add(0);
nextState.add(0);
nextState.add(0);
nextState.add(0);
nextState.add(0);

setAdjustRadarForRobotTurn(true); //keep the radar still while we turn
setBodyColor(new Color(128, 128, 50));
setGunColor(new Color(50, 50, 20));
setRadarColor(new Color(200, 200, 70));
setScanColor(Color.white);
setBulletColor(Color.red);
setAdjustGunForRobotTurn(true); // Keep the gun still when we turn
turnRadarRightRadians(Double.POSITIVE_INFINITY);//keep turning radar right
}

/**
 * onScannedRobot:  Here's the good stuff
 */
public void onScannedRobot(ScannedRobotEvent e) {
enemyLockedOn = true;
enemyDistance = e.getDistance();

double absBearing=e.getBearingRadians()+getHeadingRadians();//enemies absolute bearing
double latVel=e.getVelocity() * Math.sin(e.getHeadingRadians() -absBearing);//enemies later velocity
double gunTurnAmt;//amount to turn our gun
setTurnRadarLeftRadians(getRadarTurnRemainingRadians());//lock on the radar
gunTurnAmt = robocode.util.Utils.normalRelativeAngle(absBearing- getGunHeadingRadians()+latVel/22);/
setTurnGunRightRadians(gunTurnAmt); //turn our gun

}
public void onHitWall(HitWallEvent e){
moveDirection=-moveDirection;//reverse direction upon hitting a wall
}
/**
 * onWin:  Do a victory dance
 */
public void onWin(WinEvent e) {
/*for (int i = 0; i < 50; i++) {
turnRight(30);
turnLeft(30);
}*/

reward = 100;

double temp1 = q.QTable[nextState.get(0)][nextState.get(1)][nextState.get(2)][nextState.get(3)][curr
```

```
q.QTable[nextState.get(0)][nextState.get(1)][nextState.get(2)][nextState.get(3)][currentAction] = (1

//winLoss.add(1);

q.saveWins(wlFile, "1");

}

public void onHitRobot(HitRobotEvent event)
{
reward = -50;
}

public void onBulletHit(BulletHitEvent e)
{
reward = e.getBullet().getPower() * 50;
}

public void onBulletMissed(BulletHitEvent e)
{
reward = 0*-e.getBullet().getPower() * 10;
//System.out.format("Bullet missed\n");

}

public void onBulletHitBullet(BulletHitEvent e)
{
reward = -e.getBullet().getPower() * 2;
//System.out.format("Own bullet hit enemy bullet");
}

public void onHitByBullet(HitByBulletEvent e)
{
reward = -50;
//System.out.format("Hit by enemy bullet\n");
}

public void onDeath(DeathEvent e)
{
reward = -100;

double temp1 = q.QTable[nextState.get(0)][nextState.get(1)][nextState.get(2)][nextState.get(3)][curr
q.QTable[nextState.get(0)][nextState.get(1)][nextState.get(2)][nextState.get(3)][currentAction] = (1

//winLoss.add(0);
q.saveWins(wlFile, "0");

}

}

package kinz;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;

public class Qtable implements LUTInterface {

int xQuantized = 8;
int yQuantized = 6;
int distanceQuantized = 3;
int healthQuantized = 3;

double[][][][][] QTable;

int nStates;
int nActions;

int policy;
double quantizer;

@Override
public double outputFor(double[] X) {
// TODO Auto-generated method stub
return 0;
}

@Override
public double train(double[] X, double argValue) {
// TODO Auto-generated method stub
return 0;
}


//@Override
public void saveToFile(String wlFile, ArrayList<Integer> winLoss) {

try
{
FileWriter fileWriter = new FileWriter(wlFile, true);

for(int i=0; i < winLoss.size(); i++)
{

String s = String.format("%f,",winLoss.get(i));
fileWriter.write(s);

fileWriter.write("\n");
}
fileWriter.close();
}
catch (IOException e)
{
// TODO Auto-generated catch block
e.printStackTrace();
}

}

/**
@Override
public void load(String argFileName) throws IOException {

FileReader file = new FileReader(argFileName);

        Scanner rowScanner = new Scanner(file);
        Scanner colScanner;

        String line = null;

        int rows = 0;
        int cols = 0;

        while (rowScanner.hasNextLine())
        {

          line = rowScanner.nextLine();

          colScanner = new Scanner(line).useDelimiter(",");
          cols = 0;

          while (colScanner.hasNext())
          {
            double val =  Float.parseFloat(colScanner.next());
            QTable.get(rows).set(cols, val);

            cols++;

          }
          colScanner.close();
          rows++;
        }

        rowScanner.close();

        try
        {
file.close();
}
        catch (IOException e)
        {
// TODO Auto-generated catch block
e.printStackTrace();
}

}


@Override
public void initialiseLUT() {
```

```
Random r = new Random();
int i=0, j=0;

QTable = new ArrayList<ArrayList<Double>>();

// add rows to QTable (rows are states)
for(i = 0; i < nStates; i++)
{
QTable.add(new ArrayList<Double>());
}

for(i = 0; i < nStates; i++)
{
for(j = 0; j < nActions; j++)
{
QTable.get(i).add(0.0);
}
}

}

 **/


Qtable(int _dimStates, int _numActions, boolean loadFile)
{
 QTable = new double[xQuantized][yQuantized][distanceQuantized][healthQuantized][5];




/**
nStates = _dimStates;
nActions = _numActions;
initialiseLUT();
if(loadFile == true)
{
try
{
load("QTable.csv");
}
catch (IOException e)
{
// TODO Auto-generated catch block
e.printStackTrace();
}
}
**/
}

 public void saveWins(String f, String win)
 {
 try{

     File file =new File(f);
     //if file doesn't exists, then create it
     if(!file.exists()){
     file.createNewFile();
     }

     //true = append file
     FileWriter fileWriter = new FileWriter(f,true);
             BufferedWriter bufferWriter = new BufferedWriter(fileWriter);
             bufferWriter.write(String.format(win + "\n").toString());
             bufferWriter.close();

         System.out.format(win + "\n");

         fileWriter.close();

     }catch(IOException e){
     e.printStackTrace();
     }

 }


public void saveRewards(ArrayList <Double> r, String f) throws IOException
{
FileWriter fileWriter = new FileWriter(f, false);


for(int i=0; i<r.size(); i++)
{
String s = String.format("%d,%f\n",i,r.get(i));
fileWriter.write(s);
}

fileWriter.close();
}

@Override
public int indexFor(double[] X) {
// TODO Auto-generated method stub
return 0;
}

public int quantize(double nonQuantized, double minimum, double maximum, int q)
{
double qCreate = (nonQuantized - minimum)/(maximum - minimum);

return (int) (qCreate * q);
}

@Override
public void save(File argFile) {
// TODO Auto-generated method stub

}

@Override
```

```
public void load(String argFileName) throws IOException {
// TODO Auto-generated method stub

}

@Override
public void initialiseLUT() {
// TODO Auto-generated method stub

}
}

package kinz;

import java.io.File;
import java.io.IOException;

public interface CommonInterface {

public double outputFor(double [] X);

public double train(double [] X, double argValue);

public void save(File argFile);

public void load(String argFileName) throws IOException;

}
```