CPEN 502
Assignment 2 Report

Table of Contents

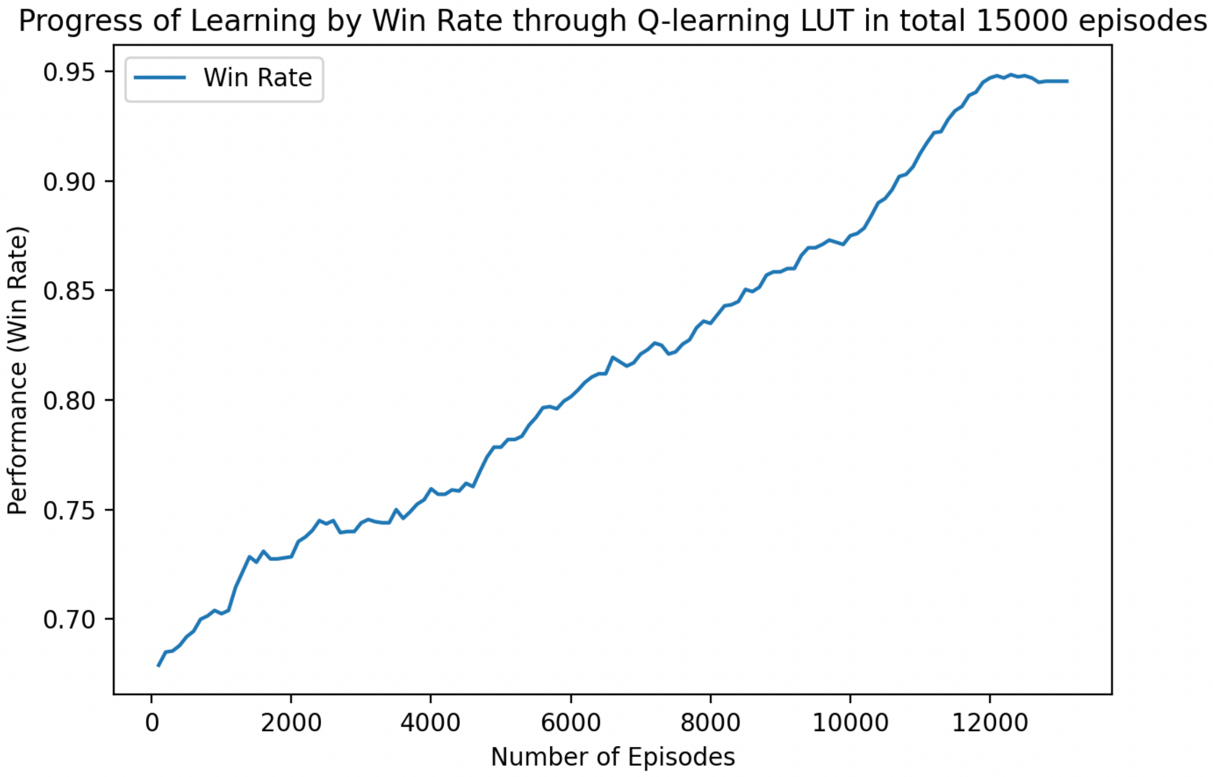
Question 2a

(2) Once you have your robot working, measure its learning performance as follows:

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

I chose win rate (number of wining rounds in every 100 episodes) to indicate the progress of learning. As the graph below shows, the win rate increases as the number of episodes increases from 0 to 15000. At around 12000 episodes, the win rate performance seems to converge to around 0.95.

Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is likely the reason why the x slightly shifted.

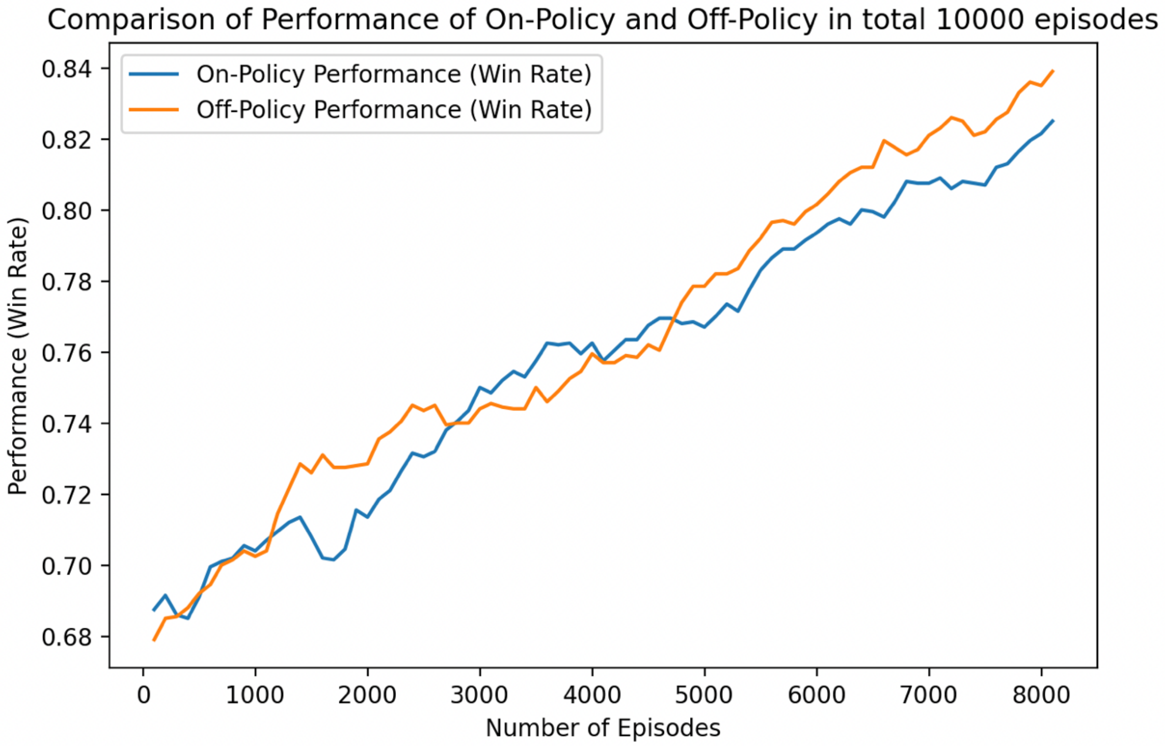


Question 2b

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

I compared the performance (using win rate in every 100 episodes) using on-policy learning vs off-policy learning in 10000 episodes. As the graph below shows, the off-policy learning seems ultimately better than on-policy training when the number of episodes is relatively large.

Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is likely the reason why the x slightly shifted.

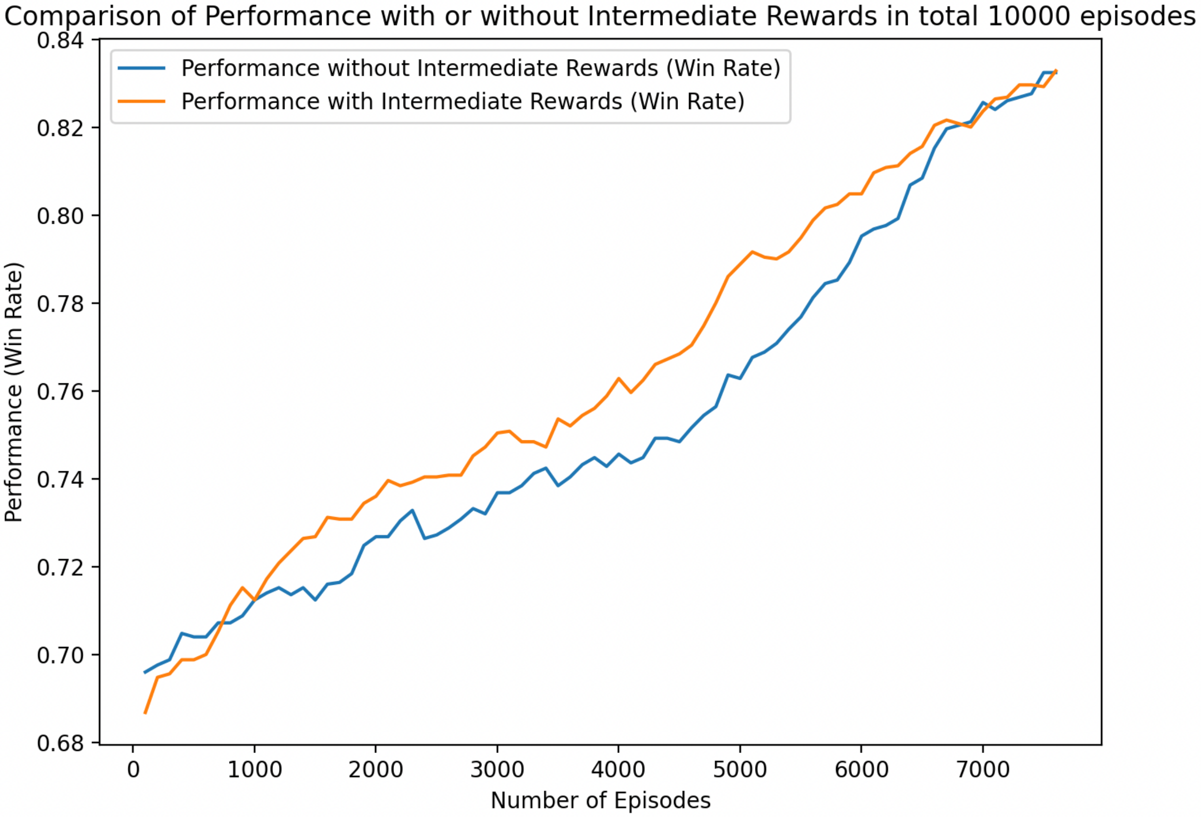


Question 2c

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

I compared the performance (using win rate in every 100 episodes) using only terminal rewards vs using both terminal and intermediate rewards in 10000 episodes. As the graph below shows, the one using both terminal and intermediate rewards seems to make progress faster. However they don't seem to have significant difference performance once the number of episodes is relatively large.

Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is likely the reason why the x slightly shifted.



Question 3a

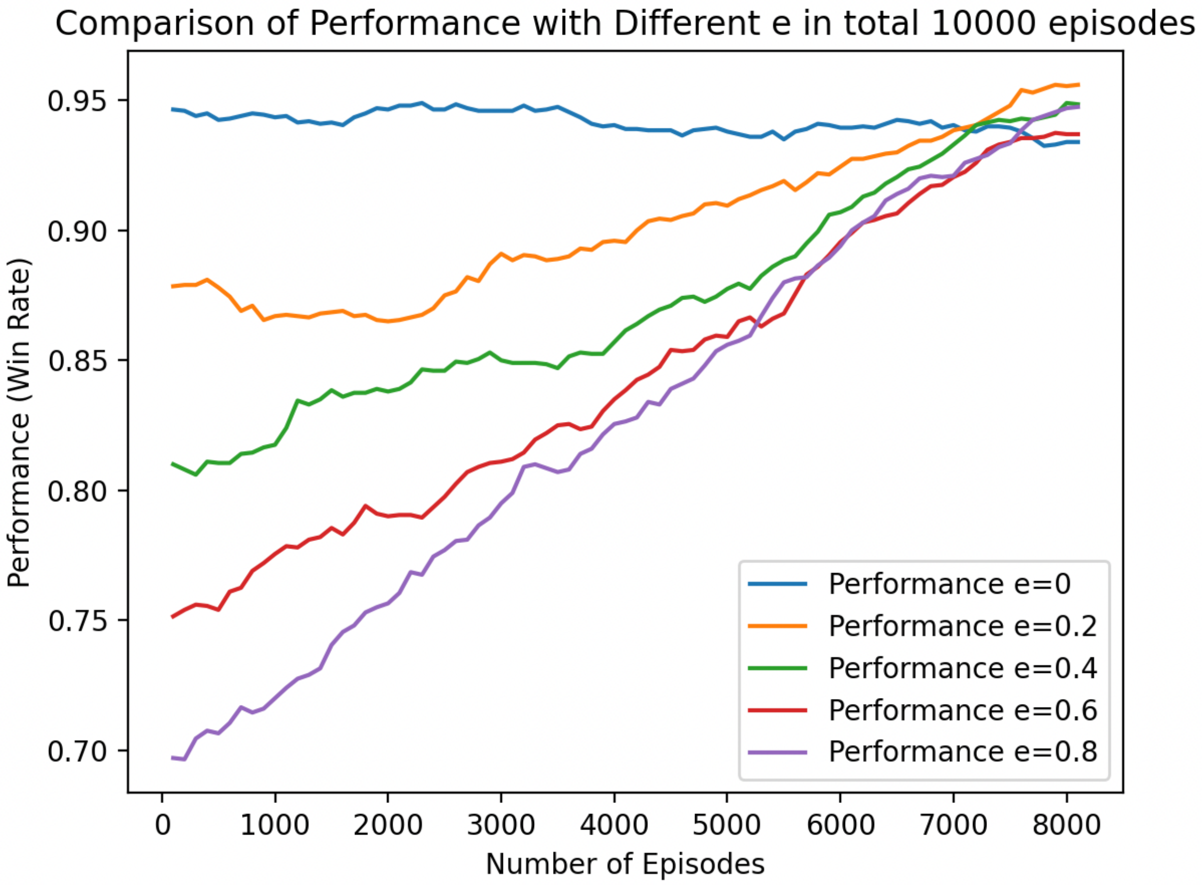
(3) This part is about exploration. While training via RL, the next move is selected randomly with probability e and greedily with probability 1-e.

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

I compared the performance (using win rate in every 100 episodes) using different epsilon e in 10000 episodes. In every case, e is gradually decaying to 0 in the first 80% episodes, and staying at 0 in the last 20% episodes in order to be compared fairly.

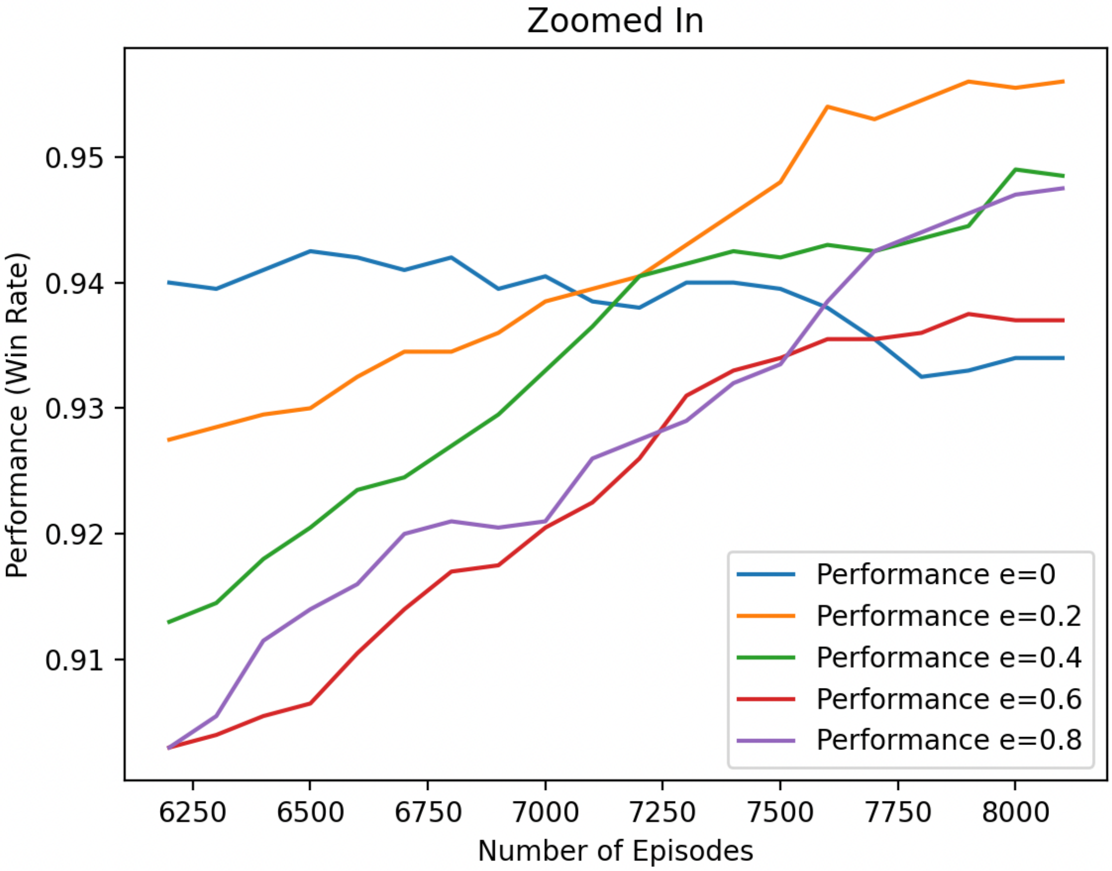
As the graph below shows, during the e decaying stage, learning processes using different e are all improving except e=0. During the measuring stage, we can see using e>0 would have better performance than e=0. (see a zoomed-in graph in the next page)

Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is likely the reason why the x slightly shifted.



This graph below is a zoomed-in portion of the graph above, to provide a close look at the measuring stage in the last 20% episodes. In which we can clearly see that using e>0 would have better performance than e=0. In this specific case, e=0.2 has the best performance, followed with e=0.4, e=0.8, e=0.6, and e=0 has the worst.

Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is likely the reason why the x slightly shifted.



Appendix for source code

```java
package Robot.My502Robot;

import LUT.LUT;
import Robot.Action;
import Robot.State;
import robocode.*;

import java.awt.*;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Arrays;
import java.util.Date;

import static robocode.util.Utils.normalRelativeAngleDegrees;


public class MyLUTRobot extends AdvancedRobot {
    private enumOperationalMode operationalMode = enumOperationalMode.scan;
//    private String weightsFileName = getClass().getSimpleName() + "-weights.txt";
    private String logFileName = getClass().getSimpleName() + "-" + "qValues" +new Date().toString() + ".dat";

    private String logFileNameWinRate = getClass().getSimpleName() + "-" + "winRate" + new Date().toString() + ".dat";

    private String logFileNameEpsilonList = getClass().getSimpleName() + "-" + "epsilonList" + new Date().toString() + ".dat";
    static private LUT lut = new LUT();
//    private String[] outputLog;
    private double curR;
    private double goodTerminalReward = 1;
    private double badTerminalReward = -1;
    private double totalR = 0;




    static double alpha = 0.2;

    static double gamma = 1;
    static int totalNumRounds = 0;
    static int numRoundTo100 = 0;
    static int numWins = 0;
    // for question2(a), used 15000, other times used 10000
    static int desiredTotalRounds = 300;
    static double[] winRatePer100 = new double[desiredTotalRounds/100];

    // for question3(a), try e = 0, 0.2, 0.4, 0.6 and 0.8(default)
    static double epsilon = 0;
    static double epsilon_init = 0;
    static double[] epsilonList = new double[winRatePer100.length];


    static int numOfRoundsToDecayE = (int) (desiredTotalRounds * 0.8);
    static double decayEStepSize = epsilon_init/numOfRoundsToDecayE;

    public enum enumOperationalMode {scan, performAction};
    private double oE;
    private double oD;
    private double oV;
    private double oB;
    private double eH;

    private State curS, preS;
    private Action curA, preA;

    public void run() {
        initialize();
        setColor();

        while (true) {
            switch (operationalMode) {
                case scan: {
                    turnRadarLeft(360);
                    curR = 0; // reset curR to 0 when scan again
                    break;
                }
                case performAction: {
                    if (Math.random() <= epsilon)
                        curA = selectRandomAction();
                    else
                        curA = bestAction(curS);

                    switch (curA) {
                        case ATTACK: {
                            turnGunRight(normalRelativeAngleDegrees(getHeading() - getGunHeading() + oB));
                            fire(2);
                            execute();
                            break;
                        }
                        case RUN_AWAY: {
                            turnRight(normalRelativeAngleDegrees(90 - (getHeading() - eH)));
                            ahead(50);
                            execute();
                            break;
                        }
                        // deleted this action because it does not seem to be useful
//                        case CHASE: {
//                            turnRight(oB);
////                            setVelocityRate(3);
//                            ahead(50);
//                            execute();
//                            break;
//                        }

                    }

                    // update Q value for (s,a)
                    lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));

//                    //TODO: for assignment 3
//                    replayMemory.add(new Experience(pres, preA, curR, curS));
//                    replayExperience(replayMemory);

                    this.operationalMode = enumOperationalMode.scan;
                }
            }
        }
    }

    private void setColor() {
        setBodyColor(Color.yellow);
        setGunColor(Color.black);
        setRadarColor(Color.red);
        setBulletColor(Color.white);
        setScanColor(Color.white);
    }

    private void initialize() {
        curS = new State(getEnergy(), 100, getX(), getY(), 100, getVelocity(), 0.2);
        curA = Action.values()[0];

        preS = curS;
        preA = curA;
    }

    private Action bestAction(State curS) {
        double bestQ = -Double.MAX_VALUE;
        int bestAindex = 0;
        double[] X = curS.transformToX();
        double[] newX = Arrays.copyOf(X, X.length + 1);
        for (int i = 0; i < Action.values().length; i++) {
            newX[X.length] = i;
            double q = lut.outputFor(newX);
            if (q > bestQ) {
                // bestQ = q;
                bestAindex = i;
            }
        }
        Action bestA = Action.values()[bestAindex];
        return bestA;
    }

    private double bestActionQ(State curS) {
        double bestQ = 0;
        int bestAindex = 0;
        double[] X = curS.transformToX();
        double[] newX = Arrays.copyOf(X, X.length + 1);
        for (int i = 0; i < Action.values().length; i++) {
            newX[X.length] = i;
            double q = lut.outputFor(newX);
            if (q > bestQ) {
                bestQ = q;
                bestAindex = i;
            }
        }
        Action bestA = Action.values()[bestAindex];
        return bestQ;
    }


    private double computeQ(State preS, State curS, double r) {
        // off-policy, q learning
        // take action, observe r, s' (find the best a' and update Q(s,a))
        // Q(s,a) = Q(s, a) + alpha(r + gamma * max(Q(s', a'))-Q(s,a))
        double oldQ = lut.outputFor(preS.transformToX(preA));
        double maxNextQ = bestActionQ(curS);
        return oldQ + alpha * (r + gamma * maxNextQ - oldQ);

//        // on-policy
//        double oldQ = lut.outputFor(preS.transformToX(preA));
//        double curQ = lut.outputFor(curS.transformToX(curA));
//        return oldQ + alpha * (r + gamma * curQ - oldQ);
    }

    private Action selectRandomAction() {
        int numOfChoice = Action.values().length;
        return Action.values()[(int) (Math.random() * numOfChoice)];
    }

//    public void replayExperience(ReplayMemory rm){
//        int ms = rm.sizeOf();
//        int requestedSs = (ms < MAX)
//    }

    public void onScannedRobot(ScannedRobotEvent e) {
        // update preS, preA; update curS
        preS = curS;
        preA = curA;
        curS = new State(getEnergy(), e.getEnergy(), getX(), getY(), e.getDistance(), getVelocity(), e.getVelocity());
        oB = e.getBearing();
        eH = e.getHeading();


        this.operationalMode = enumOperationalMode.performAction;
    }

    public void onWin(WinEvent e) {
        System.out.println("I win!!!!!!!!!!!");
        numWins++;

        curR = goodTerminalReward;
        totalR += curR;

        lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));
        //TODO: can add stat
    }

    public void onDeath(DeathEvent e) {
        System.out.println("I lose.");
        curR = badTerminalReward;
        totalR += curR;

        lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));
        //TODO: can add stat
    }
//////////////////////intermediate rewards start///////////
    public void onBulletHit(BulletHitEvent e) {
        curR = +0.4;
        totalR += curR;

        //lut.train(preS.transformToX(), computeQ(preS, curS, curR));
    }

    public void onBulletMissed(BulletMissedEvent e) {
        curR = -0.01;
        totalR += curR;
    }

    public void onHitByBullet(HitByBulletEvent event) {
        curR = -0.2;
        totalR += curR;
    }

    public void onHitWall(HitWallEvent event) {
        curR = -0.01;
        totalR += curR;
    }
//////////////////////intermediate rewards end///////////
    public void onRoundEnded(RoundEndedEvent event) {

        if(totalNumRounds < numOfRoundsToDecayE){
            if(epsilon > 0 & epsilon >decayEStepSize){
                epsilon -= decayEStepSize; // so e decaying to 0 in the first 80% round
            }
        }else{
            epsilon=0;
        }

        totalNumRounds++;
        if(totalNumRounds % 100 == 0){
            int index = totalNumRounds / 100 - 1;
            winRatePer100[index] = numWins;
            epsilonList[index] = epsilon;

            out.println("The round has ended and the winRatePer100[] updated");
            out.println("totalNumRounds"+ totalNumRounds);
            out.println("winRatePer100" + winRatePer100[index]);
            out.println("numWins" + numWins);
            numWins = 0; // reset
            out.println("numWins set to 0 again");

        }
        System.out.println("round ended");

    }


    public void onBattleEnded(BattleEndedEvent e)
    {
        finalWriteQ();
        finalWriteWins();
        finalWriteEpsilonList();
    }

    private void finalWriteEpsilonList() {
        PrintStream w = null;
        try {
            w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileNameEpsilonList)));

            for(double e: epsilonList){
                w.println(e);
            }

            if (w.checkError()) {
                out.println("I could not write the finalWriteEpsilonList!");
            }
        } catch (IOException e) {
            out.println("IOException trying to write: ");
            e.printStackTrace(out);
        } finally {
            if (w != null) {
                w.close();
            }
        }
    }

    private void finalWriteWins() {
        PrintStream w = null;
        try {
            w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileNameWinRate)));

            for(double winR: winRatePer100){
                w.println(winR);
            }

            if (w.checkError()) {
                out.println("I could not write the winRatePer100!");
            }
        } catch (IOException e) {
            out.println("IOException trying to write: ");
            e.printStackTrace(out);
        } finally {
            if (w != null) {
                w.close();
            }
        }
    }



    private void finalWriteQ(){
        PrintStream w = null;
        try {
            w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileName)));

            double[] qs = lut.getQValues();
            for(double q: qs){
                w.println(q);
            }            // PrintStreams don't throw IOExceptions during prints, they simply set a flag.... so check it here.
            if (w.checkError()) {
                out.println("I could not finalWriteQ!");
            }
        } catch (IOException e) {
            out.println("IOException trying to write: ");
            e.printStackTrace(out);
        } finally {
            if (w != null) {
                w.close();
            }
        }
    }

}
```

```java
package LUT;

import Interface.LUTInterface;
import Robot.Action;
import Robot.State;

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

public class LUT implements LUTInterface {
    private double[] qValues;
    private int numOfStates;
    private int numOfActions;
//    /**
//     * Constructor. (You will need to define one in your implementation)
//     * @param argNumInputs The number of inputs in your input vector
//     * @param argVariableFloor An array specifying the lowest value of each variable in the input vector.
//     * @param argVariableCeiling An array specifying the highest value of each of the variables in the input vector.
//     * The order must match the order as referred to in argVariableFloor. *
//     */
////    public LUT( int argNumInputs, int [] argVariableFloor, int [] argVariableCeiling ){
    public LUT(){
        numOfStates = State.possibleStates;
        numOfActions = Action.values().length;
//        int totalStates = 0;
//
//        for(int i = 0; i < argNumInputs-1; i++){
//            totalStates += (argVariableCeiling[i] - argVariableFloor[i]);
//        }
//        this.numOfStates = totalStates;
//        this.numOfActions = argVariableCeiling[argNumInputs] - argVariableFloor[argNumInputs];
        initialiseLUT();
    }

    /**
     * Initialise the look up table to all zeros.
     */
    @Override
    public void initialiseLUT() {
        this.qValues = new double[numOfStates * numOfActions];
    }


    /**
     * 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.LUT
     * @return The index where this vector maps to
     */
//    @Override
//    public int indexFor(double[] X) {
//        // X = state + action
//        // form the stateVec from copying first n-1 element from X
//        // so that it can be used as a parameter to form a State object
//        // so that we can use getIndex function in State.class to index
//        double[] stateVec = new double[X.length-1];
//        for (int i = 0; i < stateVec.length; i++) {
//            stateVec[i] = X[i];
//        }
//        State state = new State(stateVec[0], stateVec[1], stateVec[2], stateVec[3], stateVec[4], stateVec[5], stateVec[6]);
//        int index = state.getIndex((int) X[X.length]);
//
//        // we can form the action from X, but it is useless
//        // Action action = Action.values()[(int)X[X.length]];
//
//        return index;
//    }

    @Override
    public int indexFor(double[] X) {
        // X = state(size=5) + action (size=1) -> X length = 6
        //
        State state = new State((int)X[0], (int)X[1], (int)X[2], (int)X[3], (int)X[4]);
        int actionIndex = (int) X[5];
        int index = state.getIndex(actionIndex);


        return index;
    }

    @Override
    public double outputFor(double[] X) {
        double output = 0.0;
        try {
            output =qValues[indexFor(X)];
            //return output;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: " + e.getMessage());
            for(double x: X){
                System.out.println(x);
            }
        }
        return output;
        // return qValues[indexFor(X)];
    }
    public double[] getQValues(){
        return qValues;
    }

    @Override
    public double train(double[] X, double argValue) {
        qValues[indexFor(X)] = argValue;
        return 0;
    }

    @Override
    public void save(File argFile) {

    }

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

    }

}

```

```
package Robot;

public class State {
    static final int numOfLevelForDistance = 5;
    static final  int disForTooCloseToWall = 100;
    static final  int numOfLevelForEnergy = 5;

    public static final int possibleStates = numOfLevelForDistance * numOfLevelForEnergy * numOfLevelForEnergy *2 *2;
    private int disL;
    private int isCloseToW;
    private int myEL;
    private int oEL;
    private int isFaster;
    public State(double myE, double oE, double myX, double myY, double oD, double myV, double oV){
        // disL: distance level between the enemy and our robot: low(1), high(numOfLevelForDistance)
        // closeToW: if it is too close to wall: yes(1), no(-2)
        // elMy: my energy level: low(1), high(numOfLevelForEnergy)
        // elO: enemy's energy level: low(1), high(numOfLevelForEnergy)
        // total possible state:numOfLevelForDistance * numOfLevelForEnergy * numOfLevelForEnergy *2 *2
        this.myEL = computeEnergyLevel(myE);
        this.oEL = computeEnergyLevel(oE);
        this.disL = computeDistanceLevel(oD);
        this.isCloseToW = computeTooCloseToWall(myX, myY);
        this.isFaster = computeIsFaster(oV, myV);

    }

    public State(int myE, int oEL, int disL, int isCloseToW, int isFaster){
        this.myEL = myE;
        this.oEL = oEL;
        this.disL = disL;
        this.isCloseToW = isCloseToW;
        this.isFaster = isFaster;

    }

    /*
    return the index for this state (among all possible states)
     */
    public int getIndex(int actionIndex){
        int tempForisCloseToW = 0;
        int tempForisFaster = 0;
        if(this.isCloseToW == -1){
            tempForisCloseToW = 1;
        }else{
            tempForisCloseToW = 2;
        }

        if(this.isFaster == -1){
            tempForisFaster = 1;
        }else{
            tempForisFaster = 2;
        }
//        //int actionIndex = a.ordinal();
//        int numOfActions = Action.values().length;
//        return this.myEL*this.oEL*this.disL*tempForisCloseToW*tempForisFaster*numOfActions + actionIndex;

        int NUM_ACTIONS = Action.values().length;
        return (myEL-1) * (numOfLevelForEnergy * numOfLevelForDistance * 2 * 2 * NUM_ACTIONS)
                + (oEL-1) * (numOfLevelForDistance * 2 * 2 * NUM_ACTIONS)
                + (disL-1) * (2 * 2 * NUM_ACTIONS)
                + (tempForisCloseToW-1) * (2 * NUM_ACTIONS)
                + (tempForisFaster-1) * NUM_ACTIONS
                + actionIndex;
    }

    public double[] transformToX(){
        double[] X = new double[5];
        X[0] = this.myEL;
        X[1] = this.oEL;
        X[2] = this.disL;
        X[3] = this.isCloseToW;
        X[4] = this.isFaster;
        return X;
    }

    public double[] transformToX(Action a){
        double[] X = new double[6];
        X[0] = this.myEL;
        X[1] = this.oEL;
        X[2] = this.disL;
        X[3] = this.isCloseToW;
        X[4] = this.isFaster;
        X[5] = a.ordinal();
        return X;
    }

    private int computeIsFaster(double oV, double myV) {
        if (oV < myV){
            return 1; // faster than the opponent
        }else{
            return -1;
        }
    }

    private int computeEnergyLevel(double e) {
        double ratio = e / 100.0;
        int output = (int) Math.ceil(1 + ratio * (numOfLevelForEnergy-1));
        return output >5? 5: output; //energy can actually go beyond 100
        //return (int) Math.ceil(1 + ratio * (numOfLevelForEnergy-1));
        //return numOfLevelForEnergy - (int) Math.round(myE / numOfLevelForEnergy);
    }

    private int computeTooCloseToWall(double myX, double myY) {
        double YtoWall = Math.min(myY, 600-myY);
        double XtoWall = Math.min(myX, 800-myX);
        double toWall = Math.min(YtoWall, XtoWall);
        if (toWall < disForTooCloseToWall){
            return 1; // to close to wall!
        }else{
            return -1;
        }
    }

    private int computeDistanceLevel(double oD) {
        double ratio = oD / 1000.0;
        return (int) Math.ceil(1 + ratio * (numOfLevelForDistance-1));
        //return numOfLevelForDistance- (int) Math.round(oD/numOfLevelForDistance);
    }

}
```

Action.java

```java
package Robot;

public enum Action {
//    MOVE_UP,
//    MOVE_DOWN,
    RUN_AWAY,
    ATTACK;
//    CHASE;

}

```

LUTInterface.java

```java
package Interface;

public interface LUTInterface extends CommonInterface {


    /**
     * Initialise the look up table to all zeros.
     */
    public void initialiseLUT();



    /**
     * 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.LUT
     * @return The index where this vector maps to
    */
    public int indexFor(double [] X);
}

```