CPEN 502
Assignment 3 Report

Table of Contents

Question 4

(4) The use of a neural network to replace the look-up table and approximate the Q-function has some disadvantages and advantages.

Question 4a

a) There are 3 options for the architecture of your neural network. Describe and draw all three options and state which you selected and why. (3pts)

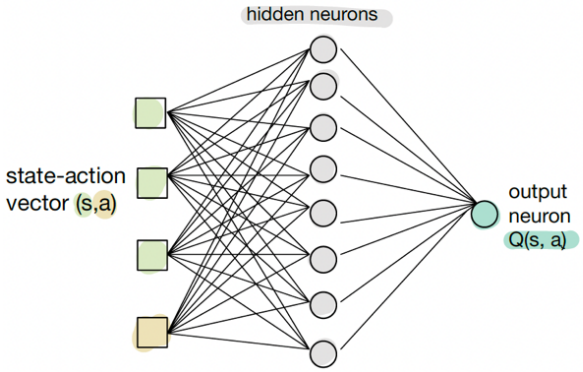
1) Inputs: a vector of the state and action *(s, a),* output: *Q(s, a),* single layer

2) Inputs: a vector of state *s,* outputs: *Q(s, a_1), Q(s, a_2) …* for all actions, multiple layers

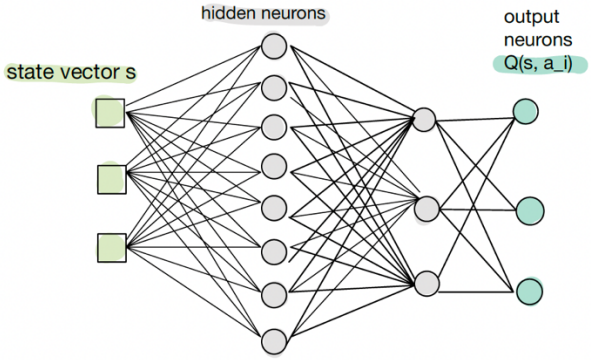
3) Inputs: two vectors (*s* is for state and *a* is for action) which are not fully connected to the first hidden layer, output: *Q(s, a),* multiple layers

Note that all of three options could potentially use single layer or multiple layers.

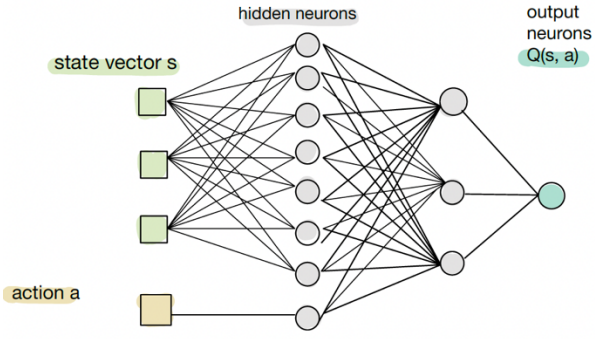
I selected option 1, because it aligns with the structure of the neural network that we wrote in assignment 1. It also aligns the structure of the look up table that we used where Q values are stored in assignment 2. **In summary, I chose option 1 because of the compatibility of the previous structure and I believe it is sufficient for this relatively small state-action space.**



Graph of Option 1



Graph of Option 2



Graph of Option 3

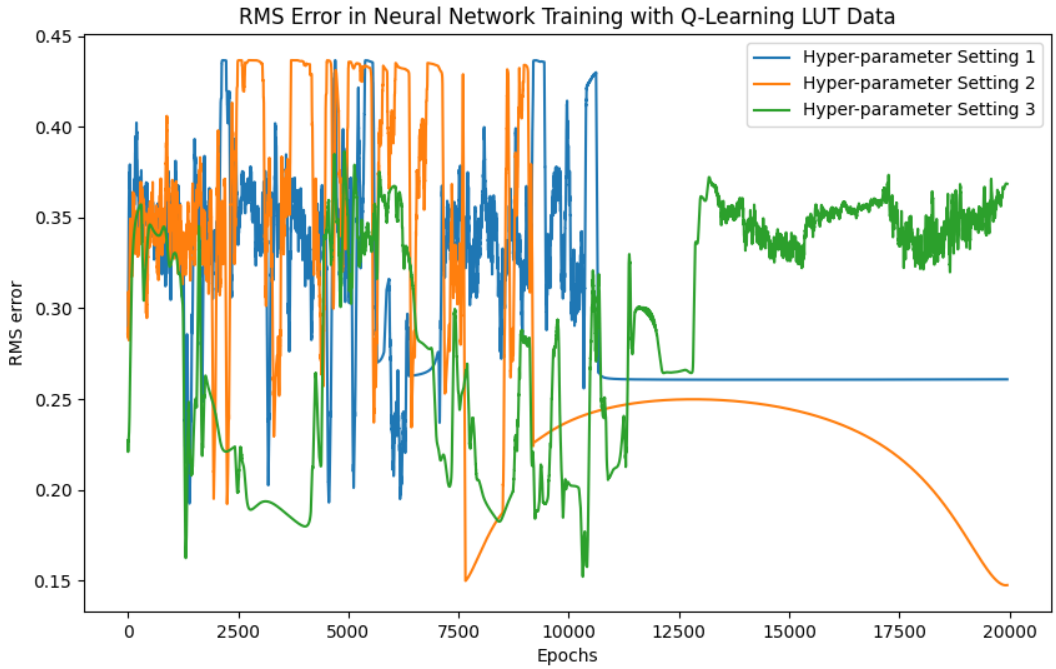
Question 4b

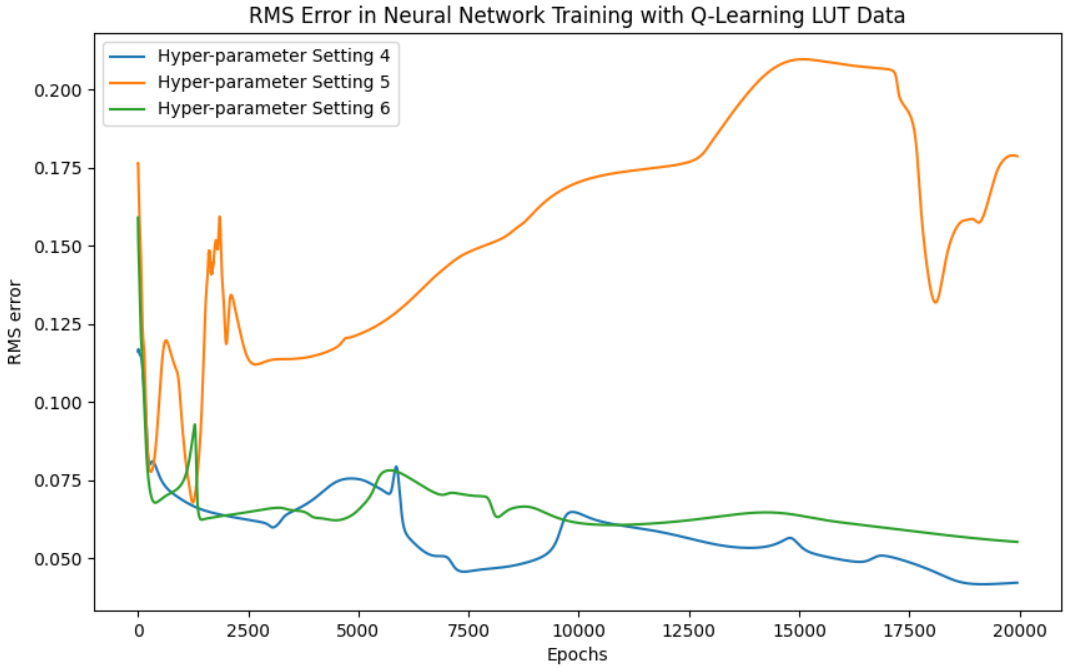
b) Show (as a graph) the results of training your neural network using the contents of the LUT from Part 2. Your answer should describe how you found the hyper-parameters which worked best for you (i.e. momentum, learning rate, number of hidden neurons). Provide graphs to backup your selection process. Compute the RMS error for your best results. (5 pts)

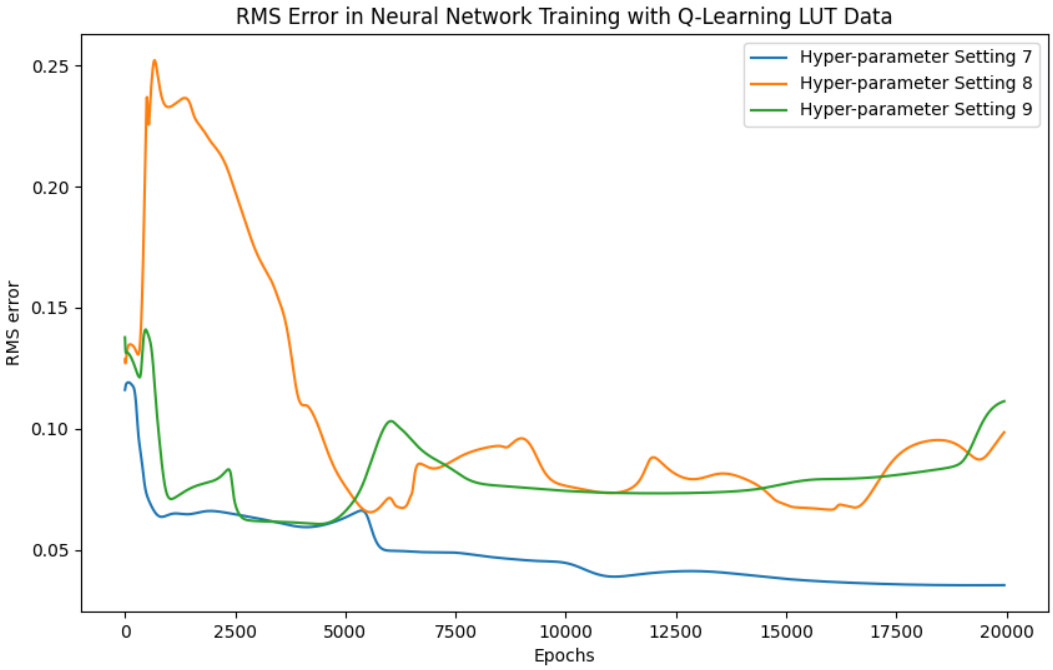
I trained the neural network with 18 different settings of the hyper-parameters (number of hidden neurons: 8, 10, learning rate: 0.2, 0.4, 0.6, momentum:0.4, 0.6, 0.8), 20000 epochs for each setting.

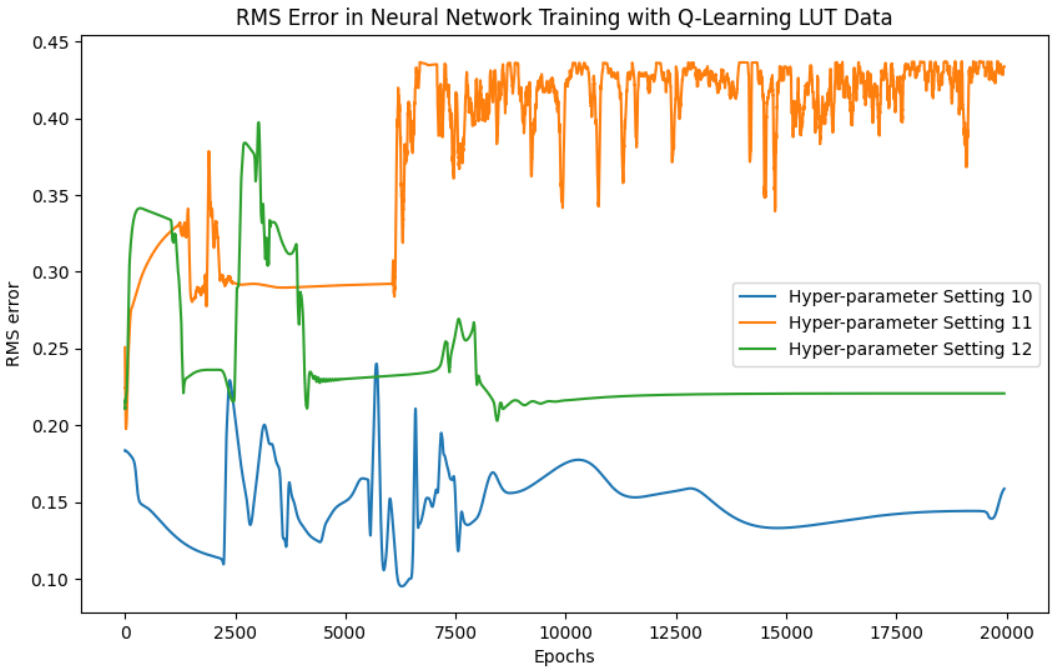
Hyper-parameter Setting Number	number of hidden neurons	learning rate	momentum	Hyper-parameter Setting Number	number of hidden neurons	learning rate	momentum
1	8	0.2	0.8	**10**	10	0.2	0.8
2	8	0.6	0.8	**11**	10	0.6	0.8
3	8	0.4	0.8	**12**	10	0.4	0.8
4	8	0.2	0.6	**13**	10	0.2	0.6
5	8	0.6	0.6	**14**	10	0.6	0.6
6	8	0.4	0.6	**15**	10	0.4	0.6
7	8	0.2	0.4	**16**	10	0.2	0.4
8	8	0.6	0.4	**17**	10	0.6	0.4
9	8	0.4	0.4	**18**	10	0.4	0.4

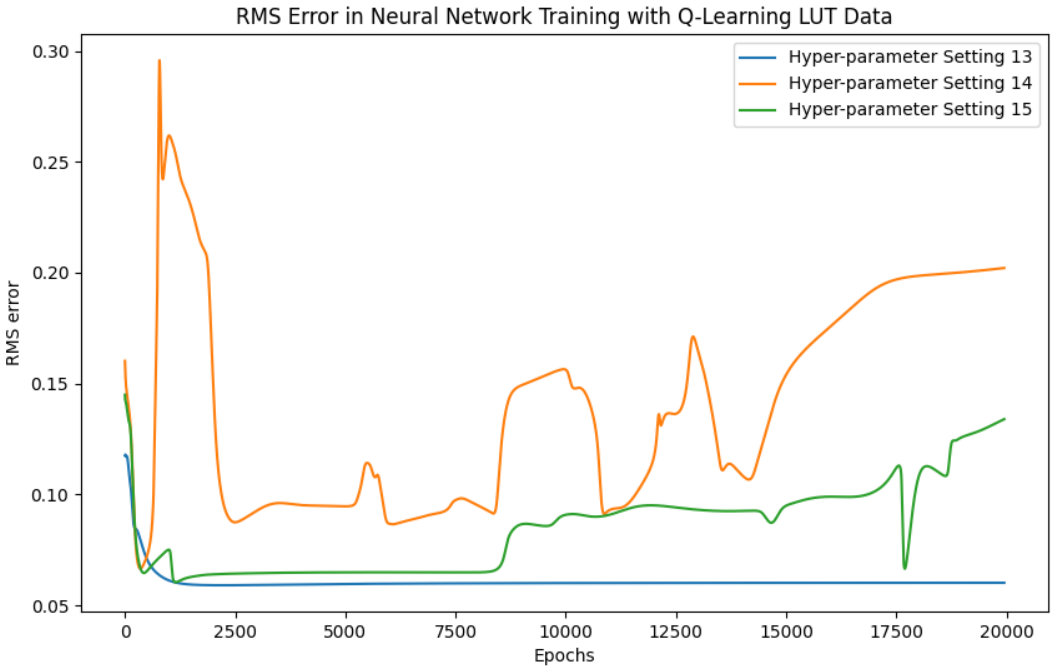
By analysing the graphs of the errors for all 18 settings, many settings are eliminated because of weird behavior of RMS such as noisy (e.g., setting 1), multiple peaks (e.g., setting 5), and not converging (e.g., 15). Among all settings, I finally choose the hyper-parameters setting 16 (**10 hidden neurons, learning rate = 0.2, momentum = 0.4**), because this setting seems to converge fast with a reasonably RMS pattern, as well as achieving a best RMS error around **0.04**.

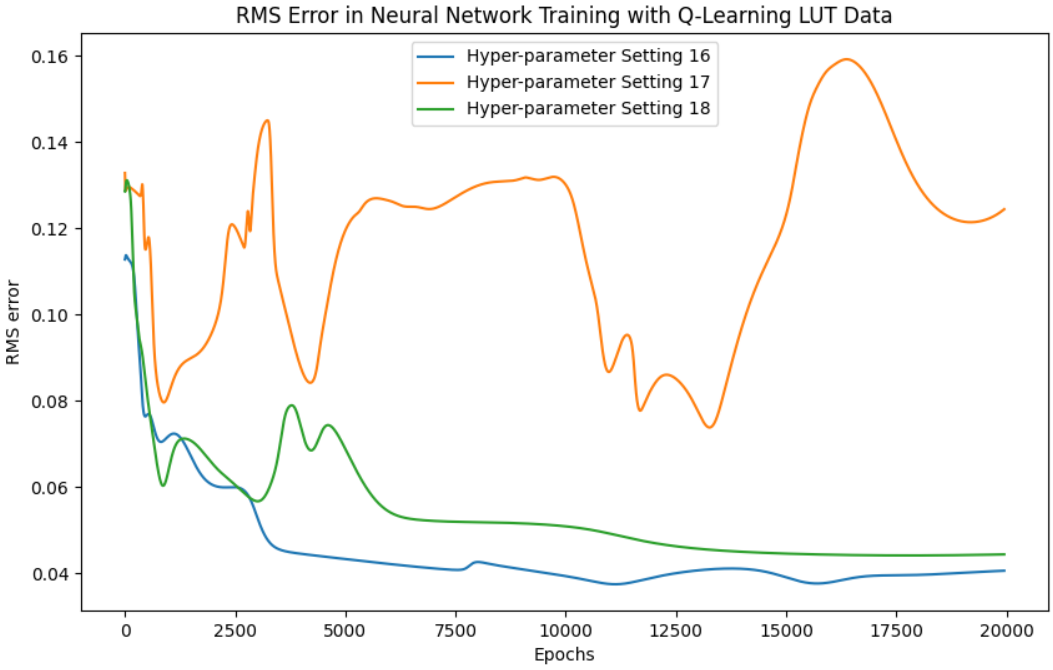





RMS Error in Neural Network Training with Q-Learning LUT Data
Hyper-parameter Setting 7
Hyper-parameter Setting 8
Hyper-parameter Setting 9
RMS error
Epochs
0.25
0.20
0.15
0.10
0.05
0
2500
5000
7500
10000
12500
15000
17500
20000


RMS Error in Neural Network Training with Q-Learning LUT Data
Hyper-parameter Setting 10
Hyper-parameter Setting 11
Hyper-parameter Setting 12
RMS error
Epochs
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
2500
5000
7500
10000
12500
15000
17500
20000


RMS Error in Neural Network Training with Q-Learning LUT Data
Hyper-parameter Setting 13
Hyper-parameter Setting 14
Hyper-parameter Setting 15
RMS error
Epochs
0.30
0.25
0.20
0.15
0.10
0.05
0
2500
5000
7500
10000
12500
15000
17500
20000


RMS Error in Neural Network Training with Q-Learning LUT Data
Hyper-parameter Setting 16
Hyper-parameter Setting 17
Hyper-parameter Setting 18
RMS error
Epochs
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0
2500
5000
7500
10000
12500
15000
17500
20000

Question 4c

c) Comment on why theoretically a neural network (or any other approach to Q-function approximation) would not necessarily need the same level of state space reduction as a look up table. (2 pts)

When applying Q-learning using a look up table, state space reduction is necessary because a look up table stores the state-action and the Q values accordingly, but it is impossible to store all possible states, especially if there are continuous variables.

However, Q-function approximation tries to find out the **relationship** between the input and output, which are the state-action pair and the Q value. It is trying to find a relationship or a function, so it is **generalizable**. It needs to learn the pattern from the inputs and outputs, but it does **not need to store** all the inputs, thus it does not necessarily need the same level of state space reduction.

Question 5

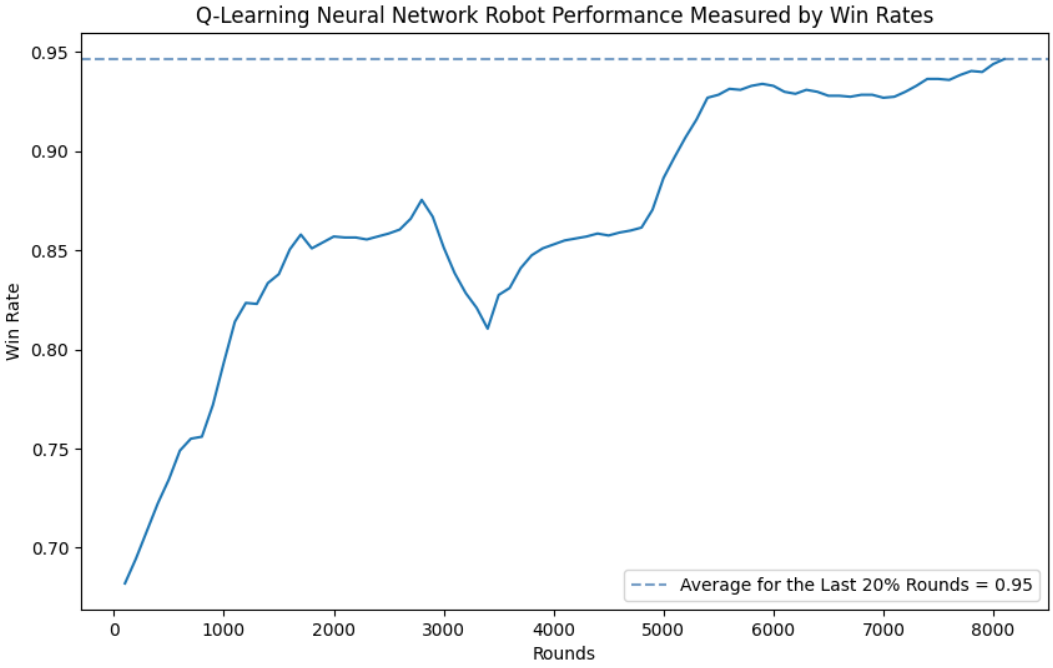
(5) Hopefully you were able to train your robot to find at least one movement pattern that results in defeat of your chosen enemy tank, most of the time.

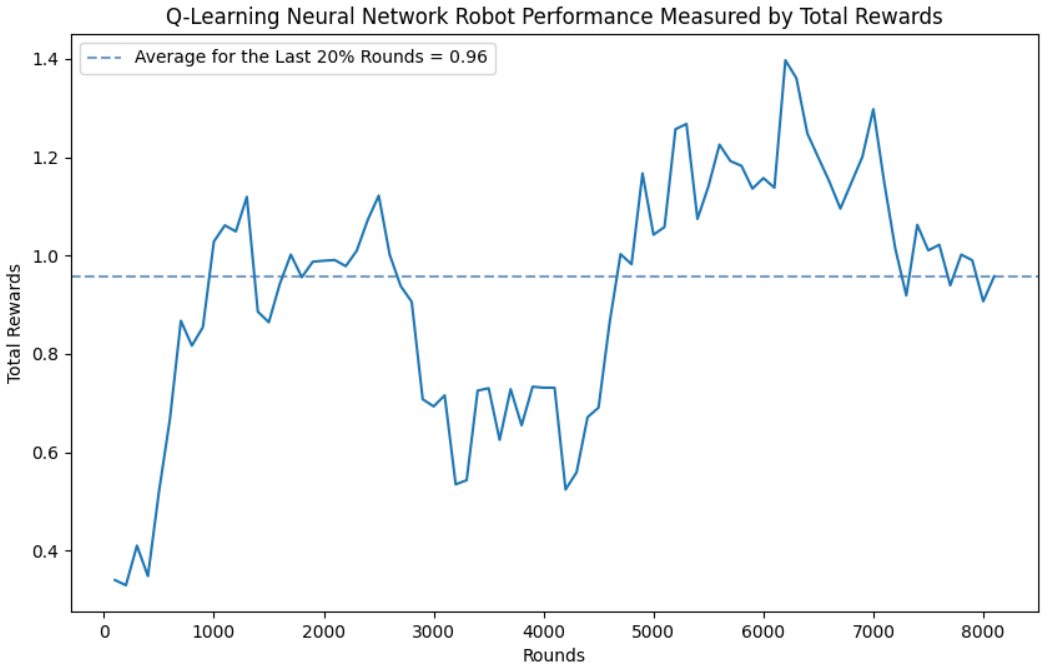
Question 5a

a) Identify two metrics and use them to measure the performance of your robot with online training. I.e. during battle. Describe how the results were obtained, particularly with regard to exploration? Your answer should provide graphs to support your results. (5 pts)

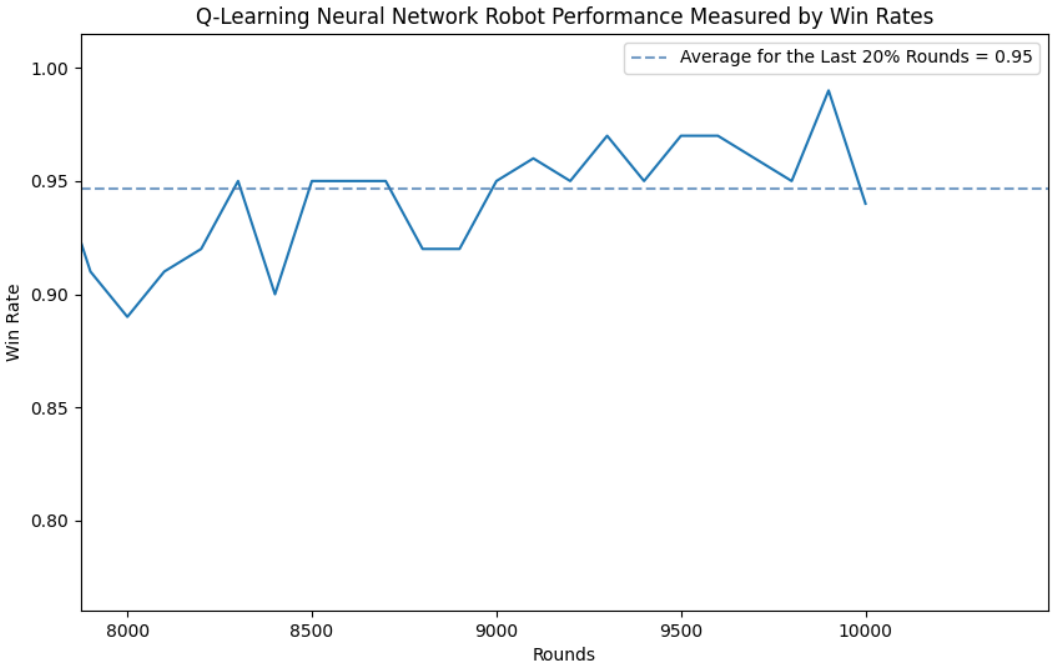
I am using two metrics (1) win rates in every 100 rounds and (2) total rewards in every 100 rounds to measure the performance of my robot. In the total 10000 rounds, exploration decay is used in the first 80% rounds, the true performance after learning is measure in the last 2000 rounds with e=0. By calculating the average of the last 20% rounds (performance measurement state), we conclude that our robot achieves a **win rate 0.95**, and **total rewards 0.96** after learning. **The graphs are in the next page**.

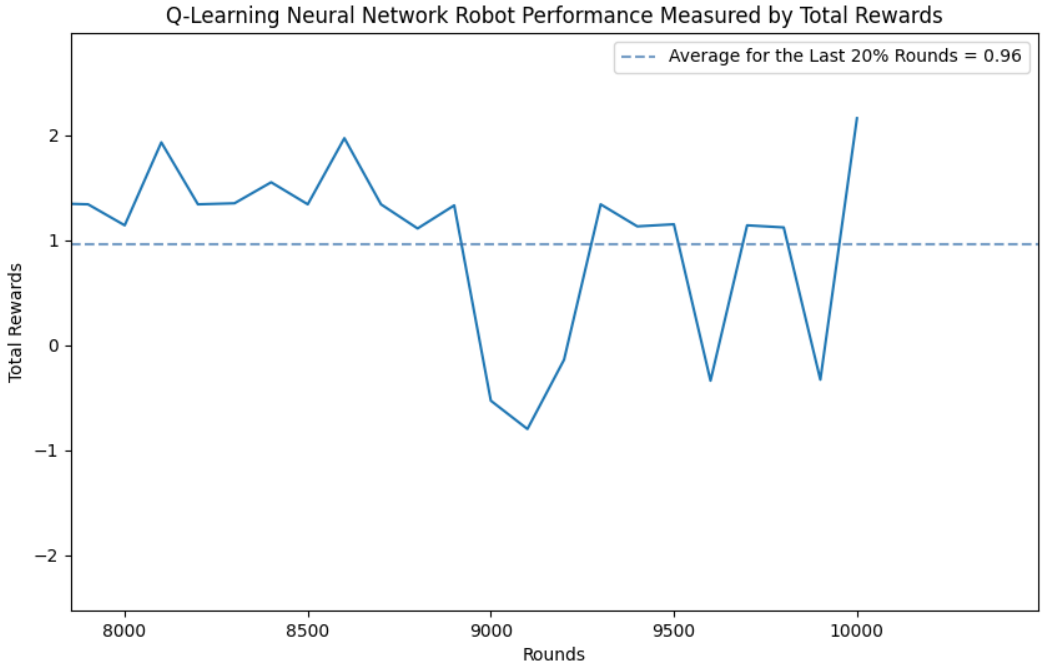
Performance measured by win rate and total rewards in 10000 rounds. Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is the reason why the x slightly shifted.





Zoomed-In version: Performance measured by win rate and total rewards in the last 20% rounds.





Question 5b

b) The discount factor can be used to modify influence of future reward. Measure the performance of your robot for different values of and plot your results. Would you expect higher or lower values to be better and why? (3 pts)

Setting of the experiment:

I tested four different discount factors (1, 0.9, 0.4, 0.1) in neural network training. In the total 10000 rounds, exploration decay is used in the first 80% rounds, the true performance after learning is measure in the last 2000 rounds with e=0. Robot performance is measured by (1) win rate in every 100 rounds and (2) total rewards in every 100 rounds.

Findings: (Please find the graphs in the next two pages)

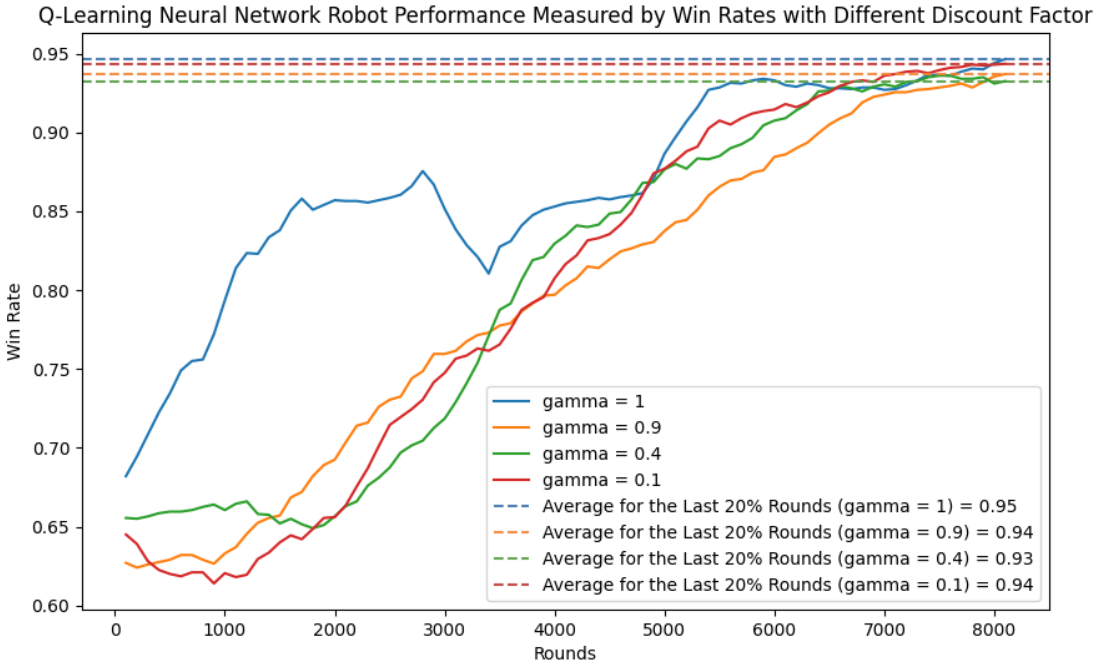
By analyzing the data, we found that different gammas have different effects in the performance when using different measurement. If we measure the performance of the robot by win rate, we notice that all different gammas reach a similar win rate 0.94 eventually. However, if we measure the performance of the robot by total rewards, the neural network with gamma=1 reaches the highest total rewards 0.96, while the lower gammas 0.9, 0.4, 0.1 all give total rewards around 0.62.

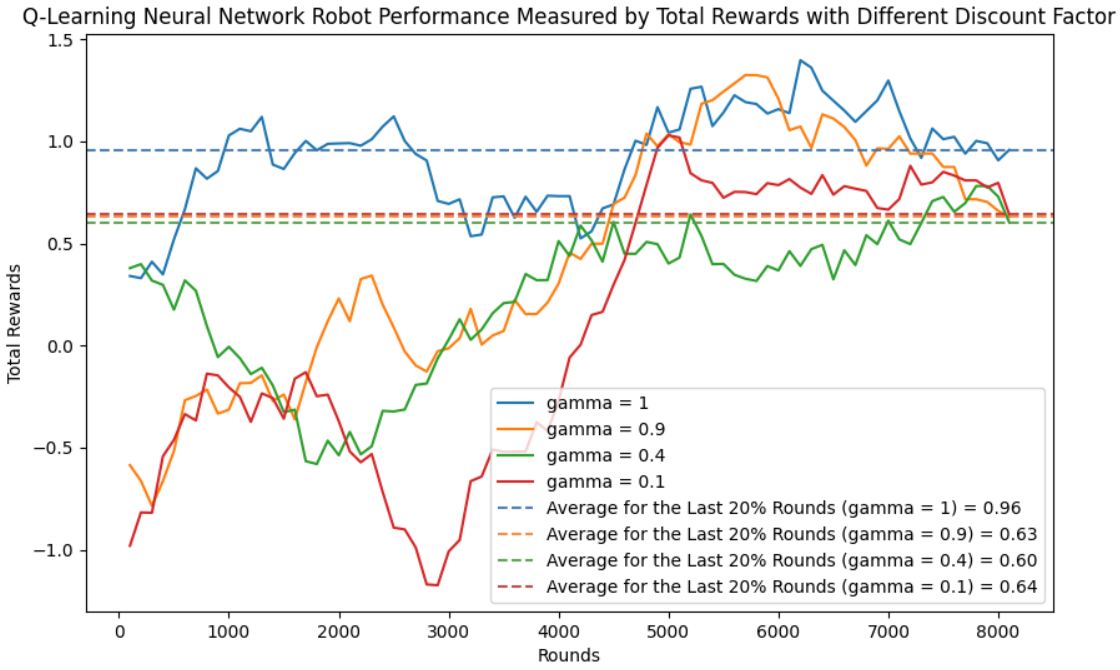
Discussion:

The **Initially expectation** was that the higher discount factor would give higher win rate and higher total rewards. The rational was that prioritize the future rewards would, to some extent, mean that it prioritizes the terminal rewards more. To reach the terminal rewards, we need to win the game (and the strategy that the robot discovered is to fire, not to hide) by accumulate immediate rewards, consequently both the win rate and the total rewards would go high.

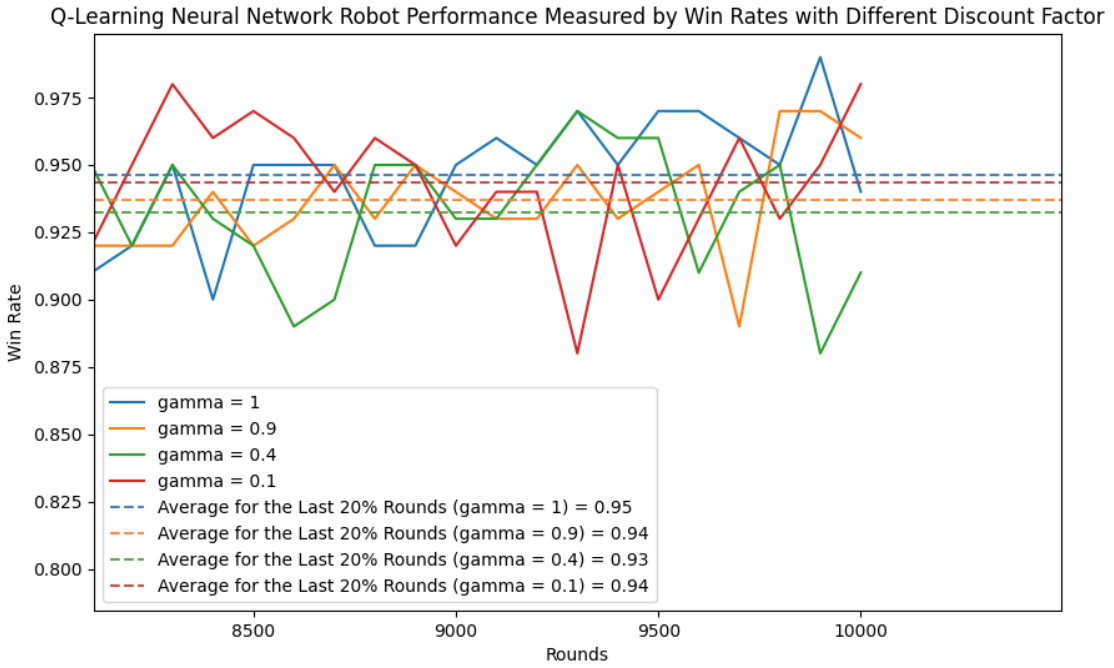
After examining the results, I found that the effect is could be more complex because of the complex nature of the robot. We found that the discount factors would reach a high win rate after sufficient training, but it does not necessarily guarantee the similar total rewards because they may have different decision-making. Only the gamma=1 shows a significantly high total rewards, which is a possible indicator of better optimized resource utilization. It is interesting to notice that gamma=0.9 and gamma=0.1 have similar low total rewards 0.62, this could be just because in our specific dynamics of the robocode setting (the nature of the battle, the opponent, the way that we set up the terminal and immediate rewards), an extreme high discount factor would be significantly better, and any deviation ones could result similar low total rewards.

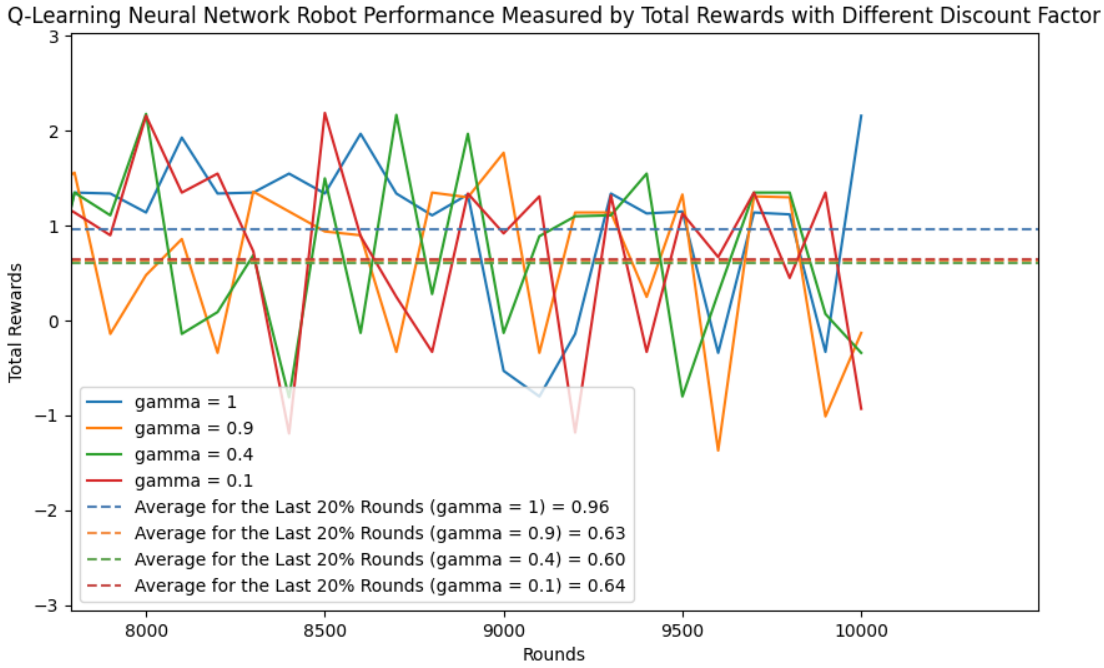
Performance measured by win rate and total rewards in 10000 rounds, with 4 different gammas. Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is the reason why the x slightly shifted.





Zoomed-In version: Performance measured by win rate and total rewards in the last 20% rounds, with 4 different gammas.





Question 5c

c) Theory question: With a look-up table, the TD learning algorithm is proven to converge – i.e. will arrive at a stable set of -values for all visited states. This is not so when the Q-function is approximated. Explain this convergence in terms of the Bellman equation and also why when using approximation, convergence is no longer guaranteed. (3 pts)

From the Bellman equation $V(s\_\text{t}) = r_t + \gamma * V(s\_t')$, we can derive that error $e(s\_t) = \gamma * e(s\_t')$. The reward at the terminal is known as zero, therefore after enough learning, we will have $e(s_t) = 0$ for all t, which leads to an optimal value function $V(s\_\text{t}) = r_t + \gamma * V'(s\_t)$. $V(s\_t)$ approaches $V'(s\_t)$, the optimal value.

When using neural network to do Q function approximation, convergence is not guaranteed because ultimately it is an approximation of the value function. In more details, a possible reason is that when we use iteration in a look up table, other unrelated states' value does not get influenced by the update of a specific state. However, in function approximation through neural network, we approximate value function by updating all the weights, which will indeed influence all states' values to minimize the error.

Question 5d

d) When using a neural network for supervised learning, performance of training is typically measured by computing a total error over the training set. When using the NN for online learning of the -function in robocode this is not possible since there is no a-priori training set to work with. Suggest how you might monitor learning performance of the neural net now. (3 pts)

We can monitor the learning performance by the **performance of the robot**, such as win rate, total rewards, etc. We can also monitor it by evaluate the actions or **observe the behaviours** of the robot before and after training, and try to interpret if the decision-making process reflects some learning. What's more, we can have a **baseline** where we evaluate the performance of the robot using the same weights of the neural network (usually generated randomly) before any backpropagation or learning.

What's more, we can compare the speed or the efficiency of learning performance by the **convergence speed** of the matrices mentioned above (such as win rate, total rewards, action choices), or the **convergence speed** of the weights of the neural network.

Question 5e

e) At each time step, the neural net in your robot performs a back propagation using a single training vector provided by the RL agent. Modify your code so that it keeps an array of the last say n *training vectors and at each time step performs* n *back propagations. Using graphs compare the performance of your robot for different values of* n*. (4 pts)*

Setting of the experiment:

I tested four different replay memory size (3, 8, 10, 15, 20) in neural network training. In the total 15000 rounds, exploration decay is used in the first 80% rounds, the true performance after learning is measure in the last 2000 rounds with e=0. Robot performance is measured by (1) win rate in every 100 rounds and (2) total rewards in every 100 rounds.

Findings: (Please find the graphs in the next two pages)

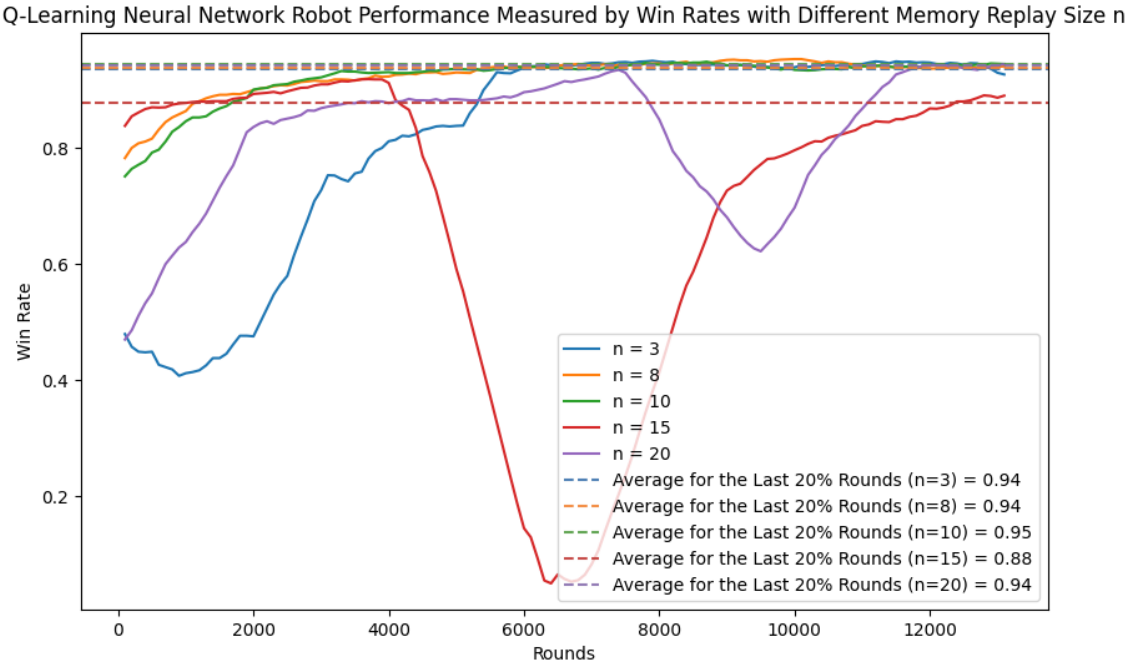
By analyzing the data, we found that different replay memory sizes have different effects in the performance when using different measurement.

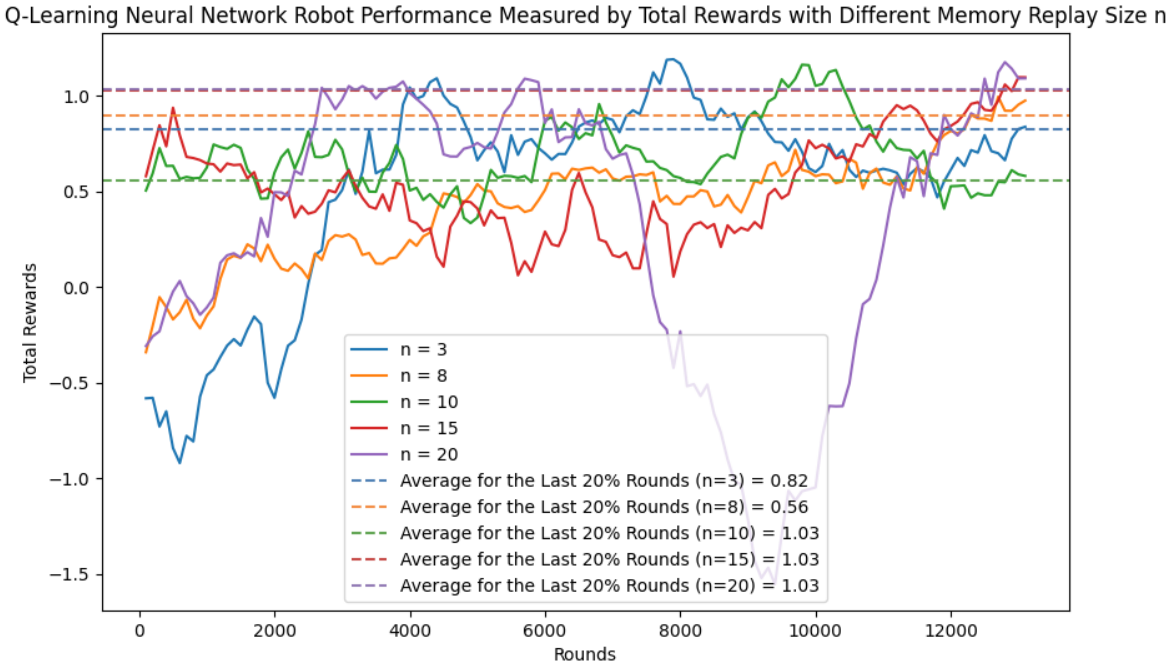
If we measure the performance of the robot by **win rate**, we notice that n=15 reaches a win rate 0.88, while all other sizes n=3, 8, 10, 20 reach a higher win rate 0.94 eventually.

If we measure the performance of the robot by **total rewards**, the neural network with n=3 reaches a total rewards 0.82, n=8 reaches a total rewards 0.56, while all higher sizes n=10, 15, 20 give total rewards around 1.03.

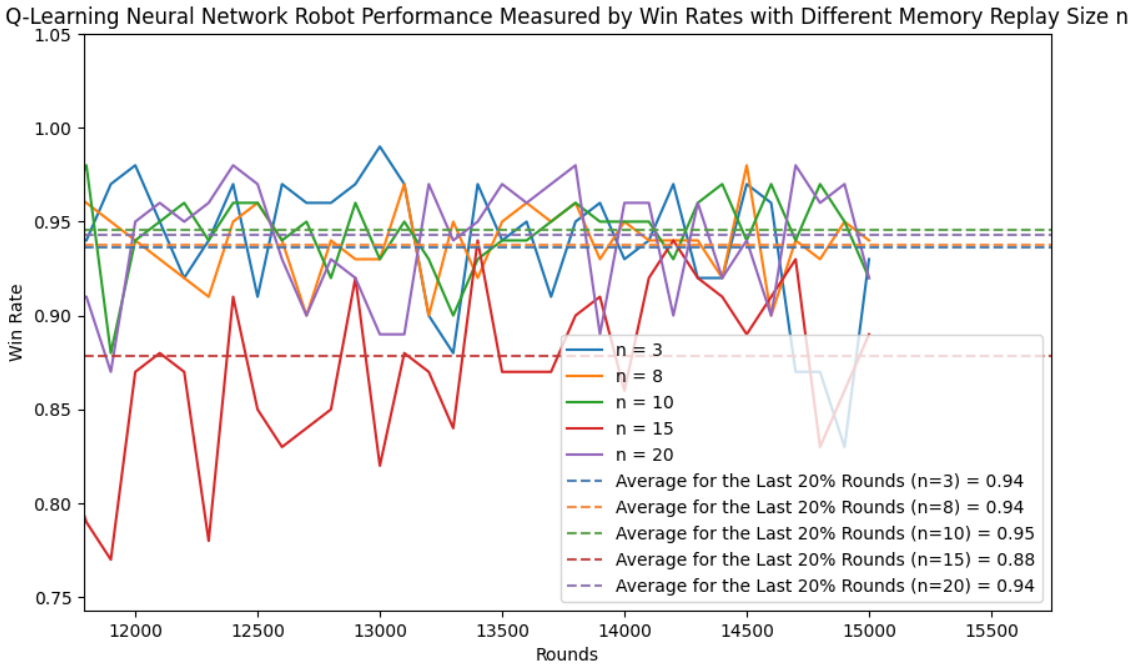
In summary, I found that larger replay memory sizes generally tend to lead better performance of the robot. This seems make sense because one major advantage of reply memory is that it can break the correlation between consecutive samples thus lead to faster and better learning. However, the replay memory size takes extra memory to store the experiences, so there is a trade-off in application. What's more, I notice that n=15 and n=20 have some sudden drop of performance during training process, which could be because the time span among the experiences is too extensive so it causes significant changes in the environment.

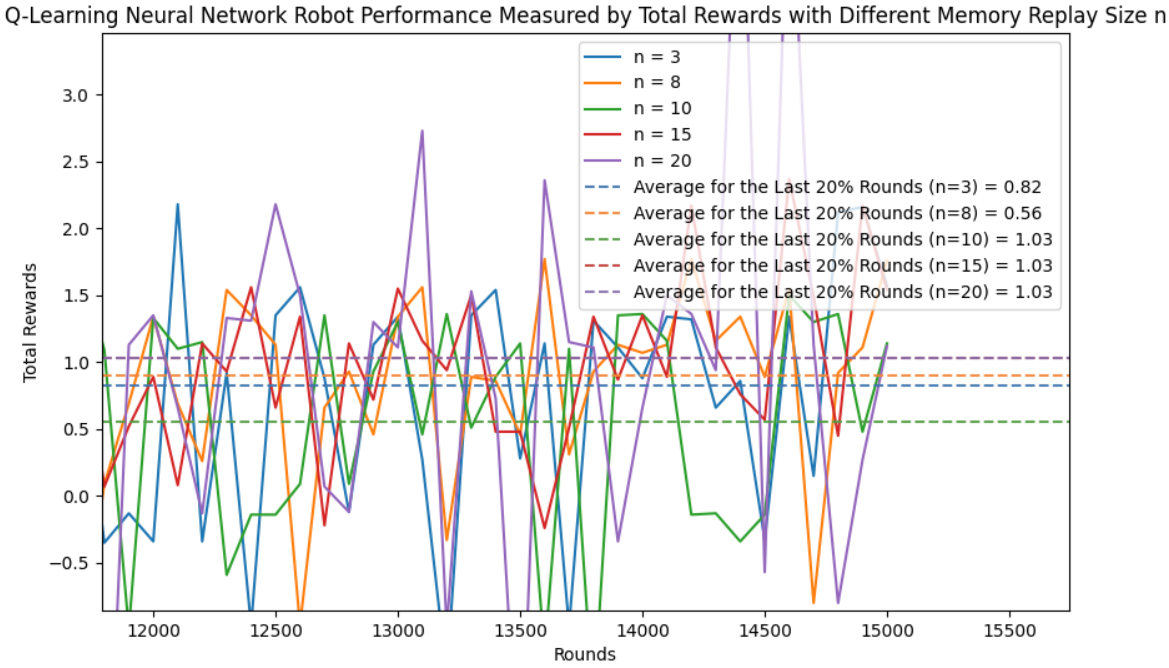
Performance measured by win rate and total rewards in the last 20% rounds, with 5 different replay memory size. Note: In order to better show the trend, moving average is used to smooth out fluctuations in data, which is the reason why the x slightly shifted.





Zoomed-In version: Performance measured by win rate and total rewards in the last 20% rounds, with 5 different replay memory size.





Question 6

(6) Overall Conclusions

Question 6a

a) This question is open-ended and offers you an opportunity to reflect on what you have learned overall through this project. For example, what insights are you able to offer with regard to the practical issues surrounding the application of RL & BP to your problem? E.g. What could you do to improve the performance of your robot? How would you suggest convergence problems be addressed? What advice would you give when applying RL with neural network based function approximation to other practical applications? (4 pts)

From the project, one of the things that I learned is that **hyper-parameter tunning** is crucial yet challenging. I had 3 main hyper-parameter (learning rate, number of hidden neurons, momentum) in this assignment, and there are potentially more to consider, such as discount factor, number of hidden layers. I learned that the hyper-parameter has strong and somewhat unpredictable effect on the performance of learning, thus careful tunning process is important. Specifically for this project, I think using other architecture such as **multiple hidden layers** may help improve the performance too.

One thing I would improve is to adjust **exploration control** more in order to gain insights how it affects the performance of learning and the robot. I tried exploration decay (decay from initial exploration rate to 0 in the first 80% rounds and stay 0 in the last 20% rounds to measure the performance) and exploration windows (initial exploration rate for 10% rounds and no exploration for the next 10% rounds, and repeat). However, I did not have the chance to carefully compare the results in these two different or possibly more ways of controlling exploration rate (also possibly for more choices of exploration rates), and measure the performance. I expect that using the exploration window method could help evaluate the **convergence speed** better in order to measure the actual **performance of the learning**.

Another thing that I reflect on is about **generalization**. Using different opponents could be a good idea to generalize the overall performance of the robot. Although I expect it could be hard and require lots of more training and hyper-parameter tunning.

Question 6b

b) Theory question: Imagine a closed-loop control system for automatically delivering anesthetic to a patient under going surgery. You intend to train the controller using the approach used in this project. Discuss any concerns with this and identify one potential variation that could alleviate those concerns. (3 pts)

Concerns:

- My first concern would be **safety**. In the robot project we did, it is fine to experience bad performance even after hyper-parameter training. However, it would be damaging if our controller delivering anesthetic does not perform well in practice.
- Another concern is that it could be hard to have enough high-quality **training data**, because the controller is designed to perform a sensitive task. Furthermore, how to perform **exploration** will be a problem.

Variations:

- A possible way to handle safety concern is to use **pre-defined rules** as well in the system. Besides reinforcement learning and neural network function approximation, we can combine experts' knowledge with the model to better ensure the safety.
- In terms of lack of training data, **synthetic data** in simulated training environments could be helpful.

Appendix for source code

```java
package Robot.My502Robot;

import NN.NeuralNet;
import ReplayMemory.ReplayMemory;
import Robot.Action;
import Robot.Experience;
import Robot.StateNN;
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 MyNNRobot extends AdvancedRobot {
//    static double alpha = 0.2;
    //    private String weightsFileName = getClass().getSimpleName() + "-weights.txt";
    static final int MAX_SAMPLE_SIZE = 15;
    static ReplayMemory<Experience> replayMemory = new ReplayMemory<Experience>(MAX_SAMPLE_SIZE);
    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 = 15000;
    static double[] winRatePer100 = new double[desiredTotalRounds / 100];
    static double[] epsilonList = new double[winRatePer100.length];
    static double[] totalRewardsPer100 = new double[desiredTotalRounds / 100];
    // for question3(a), try e = 0, 0.2, 0.4, 0.6 and 0.8(default)
    static double epsilon = 0.8;
    static double epsilon_init = 0.8;
    static double[][] actionCountsPer100 = new double[desiredTotalRounds / 100][3];
//    static ArrayList<Integer> actionList = new ArrayList<>();
    static int numOfRoundsToDecayE = (int) (desiredTotalRounds * 0.8);
    static double decayEStepSize = epsilon_init / numOfRoundsToDecayE;
    //    static private LUT lut = new LUT();
    static private int argA = -1;
    static private int argB = 1;
    static private NeuralNet NN = new NeuralNet(4, 8, 0.2, 0.4, argA, argB, 2);
    private enumOperationalMode operationalMode = enumOperationalMode.scan;
    private String logFileName = getClass().getSimpleName() + "-" + "qValues" + new Date().toString() + ".dat";
    private String logFileName1 = getClass().getSimpleName() + "-" + "qValuesBeforeTurningOffE" + new Date().toString() + ".dat";
    private String logFileNameWinRate = getClass().getSimpleName() + "-" + "winRate" + new Date().toString() + ".dat";
    private String logFileNameTotalRewards = getClass().getSimpleName() + "-" + "totalRewards" + new Date().toString() + ".dat";
    private String logFileNameEpsilonList = getClass().getSimpleName() + "-" + "epsilonList" + new Date().toString() + ".dat";
    private String logFileNameActions = getClass().getSimpleName() + "-" + "actionList" + new Date().toString() + ".dat";
    //    private String[] outputLog;
    private double curR;
    private double goodTerminalReward = 1;
    private double badTerminalReward = -1;
    //    private double[] totalR = new double[100];
    private double totalR = 0;
    private double oE;

    ;
    private double oD;
    private double oV;
    private double oB;
    private double eH;
    private StateNN curS, preS;
    private Action curA, preA;

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

        for(int i = 0; i <NN.hiddenNeurons.length; i++){
            System.out.println("weights for : " + i + "neuron");
            printArrayCheck(NN.hiddenNeurons[i].getWeights());
        }
        System.out.println("weights for : " + "output neuron");
        printArrayCheck(NN.outputNeuron.getWeights());

        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);
                    }

//                    actionList.add(curA.ordinal());

                    actionCountsPer100[totalNumRounds/100][curA.ordinal()] = actionCountsPer100[totalNumRounds/100][curA.ordinal()] +1;
```

```
95
96                      switch (curA) {
97                          case ATTACK: {
98                              turnGunRight(normalRelativeAngleDegrees(getHeading() - getGunHeading() + oB));
99                              fire(3);
100                             execute();
101                             break;
102                         }
103                         case ATTACK_MINOR: {
104                             turnGunRight(normalRelativeAngleDegrees(getHeading() - getGunHeading() + oB));
105                             fire(1);
106                             execute();
107                             break;
108                         }
109 //                        case RUN_AWAY: {
110 //                            turnRight(normalRelativeAngleDegrees(90 - (getHeading() - eH)));
111 //                            ahead(30);
112 //                            execute();
113 //                            break;
114 //                        }
115                         // deleted this action because it does not seem to be useful
116                         case MOVE_LEFT: {
117                             turnLeft(90);
118                             ahead(15);
119                             execute();
120                             break;
121                         }
122 //                        case MOVE_RIGHT: {
123 //                            setTurnRight(90);
124 ////                            setVelocityRate(3);
125 //                            setAhead(50);
126 //                            execute();
127 //                            break;
128 //                        }
129
130                     }
131
132                     // update Q value for (s,a)
133 //                    lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));
134 //                    NN.train(preS.transformToX(preA), computeQ(preS, curS, curR));
135                     NN.train(preS.transformToX(preA), computeTargetForNN(preS, curS, curR));
136                     System.out.println("training done");
137                     System.out.println("cur A is " + curA.name());
138 //                    for(int i = 0; i <NN.hiddenNeurons.length; i++){
139 //                        System.out.println("weights for : " + i + "neuron");
140 //                        printArrayCheck(NN.hiddenNeurons[i].getWeights());
141 //                    }
142 //                    System.out.println("weights for : " + "output neuron");
143 //                    printArrayCheck(NN.outputNeuron.getWeights());
144
145
146                     replayMemory.add(new Experience(preS, preA, curR, curS));
147                     replayExperience(replayMemory);
148
149                     this.operationalMode = enumOperationalMode.scan;
150                 }
151             }
152         }
153     }
154
155     private void setColor() {
156         setBodyColor(Color.yellow);
157         setGunColor(Color.black);
158         setRadarColor(Color.red);
159         setBulletColor(Color.white);
160         setScanColor(Color.white);
161     }
162
163     private void initialize() {
164         curS = new StateNN(getEnergy(), 100, 100);
165         curA = Action.values()[0];
166
167         preS = curS;
168         preA = curA;
169     }
170
171
172     private Action bestAction(StateNN curS) {
173         double bestQ = -Double.MAX_VALUE;
174         int bestAindex = 0;
175         double[] X = curS.transformToX();
176         double[] newX = Arrays.copyOf(X, X.length + 1);
177         for (int i = 0; i < Action.values().length; i++) {
178             System.out.println(Action.values().length);
179             switch (i){
180                 case 0: newX[X.length]=-1;
181                     break;
182                 case 1: newX[X.length]=0;
183                     break;
184                 case 2: newX[X.length]=1;
185                     break;
186             }
187             //newX[X.length] = i;
188 //            double q = lut.outputFor(newX);
189
190             double q = NN.outputFor(newX);
191             System.out.println("newX is: ");
```

```
            printArrayCheck(newX);
            System.out.println("NN.outputFor(newX) is: ");
            System.out.println(q);

            if (q > bestQ) {
                bestQ = q;
                bestAindex = i;
            }
        }
        Action bestA = Action.values()[bestAindex];
        return bestA;
    }

    private void printArrayCheck(double[] newX) {
        for( double x : newX){
            System.out.println(x);
        }
    }

    private double bestActionQ(StateNN 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++) {
            switch (i){
                case 0: newX[X.length]=-1;
                    break;
                case 1: newX[X.length]=0;
                    break;
                case 2: newX[X.length]=1;
                    break;
            }
//            newX[X.length] = i;
//            double q = lut.outputFor(newX);
            double q = NN.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 oldQ = NN.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 double computeTargetForNN(StateNN preS, StateNN 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 oldQ = NN.outputFor(preS.transformToX(preA));
        double maxNextQ = bestActionQ(curS);
        return (r + gamma * maxNextQ);

//        // 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 onScannedRobot(ScannedRobotEvent e) {
        // update preS, preA; update curS
        preS = curS;
        preA = curA;
        curS = new StateNN(getEnergy(), e.getEnergy(), e.getDistance());
        oB = e.getBearing();
        eH = e.getHeading();


        this.operationalMode = enumOperationalMode.performAction;
    }

    public void replayExperience(ReplayMemory rm){
        int memorySize = rm.sizeOf();
        int requestedSampleSize = (memorySize < MAX_SAMPLE_SIZE) ? memorySize : MAX_SAMPLE_SIZE;

```

```
289         Object [] sample = rm.randomSample(requestedSampleSize);
290         for(Object item: sample){
291             Experience exp = (Experience) item;
292 
293 //            double[] x = new double[] {
294 //                    exp.preS.myEl,
295 //                    exp.preS.oEl,
296 //                    exp.preS.disL,
297 //                    exp.
298 //            }
299             NN.train(exp.preS.transformToX(exp.preA), computeTargetForNN(exp.preS, exp.curS, exp.curR));
300         }
301     }
302 
303     public void onWin(WinEvent e) {
304         System.out.println("I win!!!!!!!!!!!!");
305         numWins++;
306 
307         curR = goodTerminalReward;
308         totalR += curR;
309 
310 //        lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));
311 //        NN.train(preS.transformToX(preA), computeQ(preS, curS, curR));
312         NN.train(preS.transformToX(preA), computeTargetForNN(preS, curS, curR));
313         //TODO: can add stat
314         System.out.println("totalR =" + totalR);
315     }
316 
317     public void onDeath(DeathEvent e) {
318         System.out.println("I lose.");
319         curR = badTerminalReward;
320         totalR += curR;
321 
322 //        lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));
323 //        NN.train(preS.transformToX(preA), computeQ(preS, curS, curR));
324         NN.train(preS.transformToX(preA), computeTargetForNN(preS, curS, curR));
325 
326         System.out.println("totalR =" + totalR);
327     }
328 
329     ////////////////////////intermediate rewards start//////////
330     public void onBulletHit(BulletHitEvent e) {
331         curR = +0.4;
332         totalR += curR;
333 
334     }
335 
336     public void onBulletMissed(BulletMissedEvent e) {
337         curR = -0.01;
338         totalR += curR;
339     }
340 
341     public void onHitByBullet(HitByBulletEvent event) {
342         curR = -0.2;
343         totalR += curR;
344     }
345 
346     public void onHitWall(HitWallEvent event) {
347         curR = -0.01;
348         totalR += curR;
349     }
350 
351     ////////////////////////intermediate rewards end//////////
352     public void onRoundEnded(RoundEndedEvent event) {
353 
354 //        actionList.add(99);
355         if (totalNumRounds == numOfRoundsToDecayE) {
356             //WriteQ();
357         }
358 //         method 1: decay E to 0 in the first 80% rounds, stay 0 in the last 20% rounds
359         if (totalNumRounds < numOfRoundsToDecayE) {
360             if(epsilon > 0 & epsilon >decayEStepSize){
361                 epsilon -= decayEStepSize; // so e decaying to 0 in the first 80% round
362             }
363         } else {
364             epsilon = 0;
365         }
366 
367         // method 2: constant E for 10% rounds, stay 0 for 10% rounds, repeat 5 times
368 //        if (totalNumRounds > (0.1 * desiredTotalRounds) && totalNumRounds < (0.2 * desiredTotalRounds)) {
369 //
370 //
371 //            epsilon = 0;
372 //        }else if(totalNumRounds > (0.3 * desiredTotalRounds) && totalNumRounds < (0.4 * desiredTotalRounds)){
373 //            epsilon = 0;
374 //        }else if(totalNumRounds > (0.5 * desiredTotalRounds) && totalNumRounds < (0.6 * desiredTotalRounds)){
375 //            epsilon = 0;
376 //        }else if(totalNumRounds > (0.7 * desiredTotalRounds) && totalNumRounds < (0.8 * desiredTotalRounds)){
377 //            epsilon = 0;
378 //        }else if(totalNumRounds > (0.9 * desiredTotalRounds) && totalNumRounds < (1 * desiredTotalRounds)){
379 //            epsilon = 0;
380 //        }else{
381 //            epsilon=epsilon_init;
382 //        }
383 
384         totalNumRounds++;
385         if (totalNumRounds % 100 == 0) {
```

```
            int index = totalNumRounds / 100 - 1;
            winRatePer100[index] = numWins;
            totalRewardsPer100[index] = totalR;
            epsilonList[index] = epsilon;

            out.println("The round has ended and the winRatePer100[] updated");
            out.println("totalNumRounds" + totalNumRounds);
            out.println("winRatePer100" + winRatePer100[index]);
            out.println("totalRewardsPer100" + totalRewardsPer100[index]);
            out.println("numWins" + numWins);
            numWins = 0; // reset
            totalR = 0;
            out.println("numWins set to 0 again");

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

    }

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


    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 finalWriteRewardss() {
        PrintStream w = null;
        try {
            w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileNameTotalRewards)));

            for (double R : totalRewardsPer100) {
                w.println(R);
            }

            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 finalWriteActionList() {
        PrintStream w = null;
```

```
        try {
            w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileNameActions)));

            for (double[] row : actionCountsPer100) {
                for(double actionCount: row) {
                    w.println(actionCount);
                }
            }

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



    public enum enumOperationalMode {scan, performAction}
}

```

```java
package Robot;
// this class is different from State in minor way:
// StateNN is state that we use for NN, with argA, argB be the domain of the data
// this is different than quantizied data used in LUT
public class StateNN {
    static final int argA = -1;
    static final int argB = 1;
//    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;
//    public static final int possibleStates = numOfLevelForDistance * numOfLevelForEnergy * numOfLevelForEnergy;
    private double disL;
    //    private int isCloseToW;
    private double myEL;
    private double oEL;

    //    private int isFaster;
    public StateNN(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(-1)
        // 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 = convertEnergyLevel(myE);
        this.oEL = convertEnergyLevel(oE);
        this.disL = convertDistance(oD);
//        this.isCloseToW = computeTooCloseToWall(myX, myY);
//        this.isFaster = computeIsFaster(oV, myV);

    }

//    public StateNN(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 * NUM_ACTIONS)
//                + (oEL - 1) * (numOfLevelForDistance * NUM_ACTIONS)
//                + (disL - 1) * (NUM_ACTIONS)
////                + (tempForisCloseToW-1) * (2 * NUM_ACTIONS)
////                + (tempForisFaster-1) * NUM_ACTIONS
//                + actionIndex;
//    }

    public double[] transformToX() {
//        double[] X = new double[5];
        double[] X = new double[3];
        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[4];
        X[0] = this.myEL;
        X[1] = this.oEL;
```

```
95          X[2] = this.disL;
96  //        X[3] = this.isCloseToW;
97  //        X[4] = this.isFaster;
98          switch (a.ordinal()){
99              case 0: X[3]=-1;
100                 break;
101             case 1: X[3]=0;
102                 break;
103             case 2: X[3]=1;
104                 break;
105         }
106
107 //        X[3] = a.ordinal();
108 //        X[5] = a.ordinal();
109         return X;
110     }
111
112 //    private int computeIsFaster(double oV, double myV) {
113 //        if (oV < myV){
114 //            return 1; // faster than the opponent
115 //        }else{
116 //            return -1;
117 //        }
118 //    }
119
120     // convert energy (around 0 to 105) to argA and argB
121     private double convertEnergyLevel(double e) {
122         double scaledValue = scaleValue(e, 0, 110, argA, argB);
123         return scaledValue;
124     }
125
126     private double convertDistance(double oD) {
127         double scaledValue = scaleValue(oD, 0, 1000, argA, argB);
128         return scaledValue;
129     }
130     public static double scaleValue(double originalValue, double minOriginal, double maxOriginal, double minTarget, double maxTarget) {
131         // Linear scaling formula
132         return minTarget + (originalValue - minOriginal) * (maxTarget - minTarget) / (maxOriginal - minOriginal);
133     }
134
135 }
136
```

Action.java

```java
package Robot;

public enum Action {
    MOVE_LEFT,

    ATTACK_MINOR,

    ATTACK;
//    RUN_AWAY;
//    CHASE;

}

```

```java
import NN.NeuralNet;

import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

public class Main {

    static ArrayList<Double> printValues = new ArrayList<>();
    static final double[][] XORInputBinary = {{0.0, 0.0}, {0.0, 1.0}, {1.0, 0.0}, {1.0, 1.0}};
    static final double[] XOROutputBinary = {0.0, 1.0, 1.0, 0.0};

    static final double[][] XORInputBipolar = {{-1.0, -1.0}, {-1.0, 1.0}, {1.0, -1.0}, {1.0, 1.0}};
    static final double[] XOROutputBipolar = {-1.0, 1.0, 1.0, -1.0};

    public static void main(String[] args) {
//        Q1a();
//        Q1b();
//        Q1c();
        Q4();
    }

    public static void Q1a() {
        int totalTrials = 1000;
        int numOfConverge = 0, numOfFail = 0;
        int totalEpochsToConverge = 0;

        for (int trial = 0; trial < totalTrials; trial++) {
            NeuralNet XORNNBinary = new NeuralNet(2, 4, 0.2, 0.0, 0, 1, 2);

            double errs[] = new double[4];
            int numEpoch = 0;
            double totalErr;
            do {
                totalErr = 0;
                // train all pattern once
                for (int i = 0; i < 4; i++) {
                    errs[i] = XORNNBinary.train(XORInputBinary[i], XOROutputBinary[i]);
//                  totalErr += errs[i];
                }

                //after one epoch, calculate the total error
                for (int i = 0; i < 4; i++) {
                    double output = XORNNBinary.outputFor(XORInputBinary[i]);
                    totalErr += Math.pow((output - XOROutputBinary[i]), 2);
                }
                totalErr *= 0.5;
                System.out.println("Total error " + totalErr + " at " + numEpoch + " epochs.");
                numEpoch++;

                if (totalErr < 0.05) {
                    numOfConverge++;
                    totalEpochsToConverge += numEpoch;
                    System.out.println("Desired error reached at " + numEpoch + " epochs.");
                    break;
                }

            } while (numEpoch < 10000);

            if (numEpoch >= 10000) {
                System.out.println("Fail within " + numEpoch + " epochs.");
                numOfFail++;
            }
        }
        System.out.println("==========SUMMARY==========");
        System.out.println("Total trials: " + totalTrials);
        System.out.println("Success to reach desired error " + numOfConverge + " times");
        System.out.println("Fail to reach desired error (within 10000 epochs) " + numOfFail + " times");
        System.out.println("Average epochs to reach desired error: " + totalEpochsToConverge / numOfConverge);

    }

    public static void Q1b() {
        int totalTrials = 1000;
        int numOfConverge = 0, numOfFail = 0;
        int totalEpochsToConverge = 0;

        for (int trial = 0; trial < totalTrials; trial++) {
            NeuralNet XORNNBinary = new NeuralNet(2, 4, 0.2, 0.0, -1, 1, 2);

            double errs[] = new double[4];
```

```
            int numEpoch = 0;
            double totalErr;
            do {
                totalErr = 0;
                // train all pattern once
                for (int i = 0; i < 4; i++) {
                    errs[i] = XORNNBinary.train(XORInputBipolar[i], XOROutputBipolar[i]);
//                  totalErr += errs[i];
                }

                //after one epoch, calculate the total error
                for (int i = 0; i < 4; i++) {
                    double output = XORNNBinary.outputFor(XORInputBipolar[i]);
                    totalErr += Math.pow((output - XOROutputBipolar[i]), 2);
                }
                totalErr *= 0.5;
//                System.out.println("Total error " + totalErr + " at " + numEpoch + " epochs.");
                numEpoch++;

                if (totalErr < 0.05) {
                    numOfConverge++;
                    totalEpochsToConverge += numEpoch;
                    System.out.println("Desired error reached at " + numEpoch + " epochs.");
                    break;
                }

            } while (numEpoch < 10000);

            if (numEpoch >= 10000) {
                System.out.println("Fail within " + numEpoch + " epochs.");
                numOfFail++;
            }
        }
        System.out.println("==========SUMMARY==========");
        System.out.println("Total trials: " + totalTrials);
        System.out.println("Success to reach desired error " + numOfConverge + " times");
        System.out.println("Fail to reach desired error (within 10000 epochs) " + numOfFail + " times");
        System.out.println("Average epochs to reach desired error: " + totalEpochsToConverge / numOfConverge);

    }

    public static void Q1c() {
        int totalTrials = 1000;
        int numOfConverge = 0, numOfFail = 0;
        int totalEpochsToConverge = 0;

        for (int trial = 0; trial < totalTrials; trial++) {
            NeuralNet XORNNBinary = new NeuralNet(2, 4, 0.2, 0.9, -1, 1, 2);

            double errs[] = new double[4];
            int numEpoch = 0;
            double totalErr;
            do {
                totalErr = 0;
                // train all pattern once
                for (int i = 0; i < 4; i++) {
                    errs[i] = XORNNBinary.train(XORInputBipolar[i], XOROutputBipolar[i]);
//                  totalErr += errs[i];
                }

                //after one epoch, calculate the total error
                for (int i = 0; i < 4; i++) {
                    double output = XORNNBinary.outputFor(XORInputBipolar[i]);
                    totalErr += Math.pow((output - XOROutputBipolar[i]), 2);
                }
                totalErr *= 0.5;
//                System.out.println("Total error " + totalErr + " at " + numEpoch + " epochs.");
                numEpoch++;

                if (totalErr < 0.05) {
                    numOfConverge++;
                    totalEpochsToConverge += numEpoch;
                    System.out.println("Desired error reached at " + numEpoch + " epochs.");
                    break;
                }

            } while (numEpoch < 10000);

            if (numEpoch >= 10000) {
                System.out.println("Fail within " + numEpoch + " epochs.");
                numOfFail++;
            }
        }
        System.out.println("==========SUMMARY==========");
```

```
166         System.out.println("Total trials: " + totalTrials);
167         System.out.println("Success to reach desired error " + numOfConverge + " times");
168         System.out.println("Fail to reach desired error (within 10000 epochs) " + numOfFail + " times");
169         System.out.println("Average epochs to reach desired error: " + totalEpochsToConverge / numOfConverge);
170     }
171 
172 
173     // use LUT from Q2 to train NN, to pick up hyper-parameter:
174     // numhidden, learningrate, momenteum
175     public static void Q4() {
176 //        double[][] LUTInputBipolar = {{20,20,200,1},{20,20,200,2},{20,20,200,3}};
177         double[][] input = generateLUTInputBipolar();
178 //        double[][] LUTInputBipolar = transformToBipolar(input);
179         double[][] LUTInputBipolar = input;
180         printArray(LUTInputBipolar);
181 
182         double[] q = loadQfromLUT();
183 //        double[] LUTOutputBipolar = transformQtoBipolar(q);
184 //        double[] LUTOutputBipolar = bipolarTransform(q);
185         double[] LUTOutputBipolar = transformToRange(q);
186 //        double[] LUTOutputBipolar=q;
187 //        Assertions.assertEquals(LUTOutputBipolar.length, LUTInputBipolar.length, "Arrays have different lengths");
188 
189         if (LUTOutputBipolar.length == LUTInputBipolar.length) {
190             System.out.println("Arrays have the same length");
191         } else {
192             System.out.println("!!!!Arrays have different lengths!!!!");
193         }
194         for (double v : LUTOutputBipolar) {
195             System.out.println(v);
196         }
197         //writeQ(LUTOutputBipolar);
198 
199 
200         int totalTrials = 1;
201         int numOfConverge = 0, numOfFail = 0;
202         int totalEpochsToConverge = 0;
203 
204 
205         int argA = 1;
206         int argB = 2;
207 
208         System.out.println(argA);
209         System.out.println(argB);
210         for (int trial = 0; trial < totalTrials; trial++) {
211             NeuralNet LUTNNBipolar = new NeuralNet(4, 8, 0.2, 0.4, argA, argB, 2);
212 
213             // num of patterns (inputs): 5*5*5*numofaction(3)=375
214             int numOfPatterns = 375;
215             double errs[] = new double[numOfPatterns];
216             int numEpoch = 0;
217             double totalErr;
218             do {
219                 totalErr = 0;
220                 // train all pattern once
221                 for (int i = 0; i < numOfPatterns; i++) {
222                     errs[i] = LUTNNBipolar.train(LUTInputBipolar[i], LUTOutputBipolar[i]);
223 //                  totalErr += errs[i];
224                 }
225 
226                 //after one epoch, calculate the total error
227                 for (int i = 0; i < numOfPatterns; i++) {
228                     double output = LUTNNBipolar.outputFor(LUTInputBipolar[i]);
229 //                    System.out.println(output);
230 //                    System.out.println(LUTOutputBipolar[i]);
231                     totalErr += Math.pow((output - LUTOutputBipolar[i]), 2);
232                 }
233 //                totalErr *= 0.5;
234 //                System.out.println(totalErr);
235 //                totalErr = totalErr*(1/numOfPatterns);
236 //                System.out.println(totalErr);
237 //                totalErr = Math.pow(totalErr, 0.5);
238 //                System.out.println(totalErr);
239                 System.out.println(totalErr);
240                 printValues.add(totalErr);
241 
242 //                System.out.println("Total error " + totalErr + " at " + numEpoch + " epochs.");
243                 numEpoch++;
244 
245 //                if (totalErr < 0.05) {
246 //                    numOfConverge++;
247 //                    totalEpochsToConverge += numEpoch;
248 //                    System.out.println("Desired error reached at " + numEpoch + " epochs.");
249 //                    break;
```

```
//                }

            } while (numEpoch < 20000);
            String filePath = "output.txt";
            saveArrayListToTxt(printValues, filePath);
            System.out.println("Array saved to: " + filePath);

            if (numEpoch >= 20000) {
                System.out.println("Fail within " + numEpoch + " epochs.");
                numOfFail++;
            }
        }
        System.out.println("==========SUMMARY==========");
        System.out.println("Total trials: " + totalTrials);
        System.out.println("Success to reach desired error " + numOfConverge + " times");
        System.out.println("Fail to reach desired error (within 10000 epochs) " + numOfFail + " times");
        System.out.println("Average epochs to reach desired error: " + totalEpochsToConverge / numOfConverge);
    }

    private static void saveArrayListToTxt(ArrayList<Double> arrayList, String filePath) {
        try (PrintWriter writer = new PrintWriter(new FileWriter(new File(filePath)))) {
            for (Double value : arrayList) {
                writer.println(value);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void printArray(double[][] array) {
        for (double[] row : array) {
            for (double value : row) {
                System.out.print(value + "\t");
            }
            System.out.println();
        }
    }

    private static double[] transformToRange(double[] originalArray) {
        double[] transformedArray = new double[originalArray.length];

        // Find the maximum absolute value in the original array
        double maxAbsValue = 0;
        for (double value : originalArray) {
            maxAbsValue = Math.max(maxAbsValue, Math.abs(value));
        }

        // Define the target range [1, 2] and the center (1.5)
        double targetMin = 1.0;
        double targetMax = 2.0;
        double center = 1.5;

        // Perform the transformation
        for (int i = 0; i < originalArray.length; i++) {
            if (originalArray[i] > 0) {
                transformedArray[i] = ((originalArray[i] / maxAbsValue) * (targetMax - center)) + center;
            } else {
                transformedArray[i] = ((originalArray[i] / maxAbsValue) * (center - targetMin)) + center;
            }
        }

        return transformedArray;
    }

    private static double[] bipolarTransform(double[] originalArray) {
        double minValue = findMinValue(originalArray);
        double maxValue = findMaxValue(originalArray);

        double[] bipolarArray = new double[originalArray.length];

        for (int i = 0; i < originalArray.length; i++) {
            bipolarArray[i] = ((originalArray[i] - minValue) / (maxValue - minValue)) * 2 - 1;
        }

        return bipolarArray;
    }

    private static double findMinValue(double[] array) {
        double minValue = Double.MAX_VALUE;
        for (double value : array) {
            minValue = Math.min(minValue, value);
        }
        return minValue;
    }
```

```java

    private static double findMaxValue(double[] array) {
        double maxValue = Double.MIN_VALUE;
        for (double value : array) {
            maxValue = Math.max(maxValue, value);
        }
        return maxValue;
    }

    private static double[][] generateLUTInputBipolar() {
        // Define the ranges for each dimension
//        int[] xValues = {20, 40, 60, 80, 100};
//        int[] yValues = {20, 40, 60, 80, 100};
//        int[] zValues = {20, 40, 60, 80, 100};
//        int[] aValues = {10, 20, 30};

        double[] xValues = {1, 1.25, 1.5, 1.75, 2};
        double[] yValues = {1, 1.25, 1.5, 1.75, 2};
        double[] zValues = {1, 1.25, 1.5, 1.75, 2};
        double[] aValues = {1, 1.5, 2};
        // Calculate the total number of combinations
        int totalCombinations = xValues.length * yValues.length * zValues.length * aValues.length;

        // Create the LUTInputBipolar array
        double[][] LUTInputBipolar = new double[totalCombinations][4];
        int index = 0;

        // Generate all combinations
        for (double x : xValues) {
            for (double y : yValues) {
                for (double z : zValues) {
                    for (double a : aValues) {
                        LUTInputBipolar[index][0] = x;
                        LUTInputBipolar[index][1] = y;
                        LUTInputBipolar[index][2] = z;
                        LUTInputBipolar[index][3] = a;
                        index++;
                    }
                }
            }
        }
        return LUTInputBipolar;
    }



    public static double[] loadQfromLUT() {
        // Provide the path to your text file
        String filePath = "/Users/zhangxiyu/IdeaProjects/MyRobot/src/q.txt";
        // Create an ArrayList to store the values
        ArrayList<Double> values = new ArrayList<>();
        try {
            // Create a File object representing the text file
            File file = new File(filePath);

            // Create a Scanner to read from the file
            Scanner scanner = new Scanner(file);


            // Read values from the file and add them to the ArrayList
            while (scanner.hasNextDouble()) {
                double value = scanner.nextDouble();
                values.add(value);
            }


            // Close the scanner
            scanner.close();

        } catch (FileNotFoundException e) {
            // Handle file not found exception
            e.printStackTrace();
        }
        // Convert the ArrayList to an array if needed
        double[] valuesArray = new double[values.size()];
        for (int i = 0; i < values.size(); i++) {
            valuesArray[i] = values.get(i);
        }

        // Now you can use the valuesArray as an array in your code
        return valuesArray;
    }


```

```
}

```

CircularQueue.java

```java
package ReplayMemory;
import java.util.LinkedList;

public class CircularQueue<T> extends LinkedList<T> {
    private int capacity = 10;

    public CircularQueue(int capacity){
        this.capacity = capacity;
    }

    public boolean add(T e) {
        if(size() >= capacity)
            removeFirst();
        return super.add(e);
    }

    public Object [] toArray() {
        return super.toArray();
    }
}
```

Experience.java

```java
package Robot;

public class Experience {
    public StateNN preS;
    public Action preA;
    public double curR;
    public StateNN curS;

    public Experience(StateNN preS, Action preA, double curR, StateNN curS){
        this.preS = preS;
        this.preA = preA;
        this.curR = curR;
        this.curS = curS;

    }
}

```

ReplayMemory.java

```java
package ReplayMemory;
import java.util.Arrays;
import java.util.Collections;

/**
 * Author: Sarbjit Sarkaria
 * Date  : 07th January 2021
 * This class implements a replay memory for any type T.
 * The capacity of the memory must be specified upon construction.
 * The memory will discard the oldest items that do not fit.
 * @param <T> Type to be managed by the ReplayMemory
 */

public class ReplayMemory<T> {

    private CircularQueue<T> memory;
    private Object[] EMPTYARRAY = {};

    // Constructor
    public ReplayMemory(int size) {
        memory = new CircularQueue<T>(size);
    }

    // Add an item to the memory
    public void add(T experience) {
        memory.add(experience);
    }

    // Retrieve a sample of n most recently added items from the memory and return it as an array
    public Object[] sample(int n) {
        if (memory.isEmpty())
            return EMPTYARRAY;
        else {
            // I don't have a way of returning T[], so instead I return Object[]
            // See the unit tests to see how to use this tyle - no additional effort necessary
            // .... see here for more : https://stackoverflow.com/questions/1115230/casting-object-array-to-integer-array-error
            int size = memory.size();
            Object[] objectArray = memory.toArray();
            Object[] sampleObjectArray = Arrays.copyOfRange(objectArray, size-n, size);
            return sampleObjectArray;
        }
    }

    // Retrieve a random sample of n items from the memory and return it as an array
    public Object[] randomSample(int n) {
        if (memory.isEmpty())
            return EMPTYARRAY;
        else {
            // I don't have a way of returning T[], so instead I return Object[]
            // See the unit tests to see how to use this tyle - no additional effort necessary
            // .... see here for more : https://stackoverflow.com/questions/1115230/casting-object-array-to-integer-array-error
            int size = memory.size();
            CircularQueue shuffledMemory = new CircularQueue<T>(size);
            shuffledMemory.addAll(memory); // This does a deep copy
            Collections.shuffle(shuffledMemory);
            Object[] objectArray = shuffledMemory.toArray();
            Object[] sampleObjectArray = Arrays.copyOfRange(objectArray, size-n, size);
            return sampleObjectArray;
        }
    }

    // Returns the current size of the replay memory. Use for test/debug purposes
    public int sizeOf() {
        if (memory.isEmpty())
            return 0;
        else
            return memory.size();
    }
}

```

```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.ArrayList;
import java.util.Arrays;
import java.util.Date;

import static robocode.util.Utils.normalRelativeAngleDegrees;


public class MyLUTRobot extends AdvancedRobot {
    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 = 3000;
    static double[] winRatePer100 = new double[desiredTotalRounds / 100];
    static double[] epsilonList = new double[winRatePer100.length];
    // for question3(a), try e = 0, 0.2, 0.4, 0.6 and 0.8(default)
    static double epsilon = 0.8;
    static double epsilon_init = 0.8;
    //static double[] actionList = new double[desiredTotalRounds];
    static ArrayList<Integer> actionList = new ArrayList<>();
    static int numOfRoundsToDecayE = (int) (desiredTotalRounds * 0.8);
    static double decayEStepSize = epsilon_init / numOfRoundsToDecayE;
    static private LUT lut = new LUT();
    private enumOperationalMode operationalMode = enumOperationalMode.scan;
    //    private String weightsFileName = getClass().getSimpleName() + "-weights.txt";
    private String logFileName = getClass().getSimpleName() + "-" + "qValues" + new Date().toString() + ".dat";
    private String logFileName1 = getClass().getSimpleName() + "-" + "qValuesBeforeTurningOffE" + new Date().toString() + ".dat";
    private String logFileNameWinRate = getClass().getSimpleName() + "-" + "winRate" + new Date().toString() + ".dat";
    private String logFileNameEpsilonList = getClass().getSimpleName() + "-" + "epsilonList" + new Date().toString() + ".dat";
    private String logFileNameActions = getClass().getSimpleName() + "-" + "actionList" + new Date().toString() + ".dat";
    //    private String[] outputLog;
    private double curR;
    private double goodTerminalReward = 1;
    private double badTerminalReward = -1;
    private double totalR = 0;
    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);
                    }

                    actionList.add(curA.ordinal());

                    switch (curA) {
                        case ATTACK: {
                            turnGunRight(normalRelativeAngleDegrees(getHeading() - getGunHeading() + oB));
                            fire(3);
                            execute();
                            break;
                        }
                        case ATTACK_MINOR: {
                            turnGunRight(normalRelativeAngleDegrees(getHeading() - getGunHeading() + oB));
                            fire(1);
                            execute();
                            break;
```

```
                        }
//                        case RUN_AWAY: {
//                            turnRight(normalRelativeAngleDegrees(90 - (getHeading() - eH)));
//                            ahead(30);
//                            execute();
//                            break;
//                        }
                        // deleted this action because it does not seem to be useful
                        case MOVE_LEFT: {
                            turnLeft(90);
                            ahead(15);
                            execute();
                            break;
                        }
//                        case MOVE_RIGHT: {
//                            setTurnRight(90);
////                              setVelocityRate(3);
//                            setAhead(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, 100);
        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 = -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 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);
```

```
183         return oldQ + alpha * (r + gamma * maxNextQ - oldQ);
184
185 //        // on-policy
186 //        double oldQ = lut.outputFor(preS.transformToX(preA));
187 //        double curQ = lut.outputFor(curS.transformToX(curA));
188 //        return oldQ + alpha * (r + gamma * curQ - oldQ);
189     }
190
191     private Action selectRandomAction() {
192         int numOfChoice = Action.values().length;
193         return Action.values()[(int) (Math.random() * numOfChoice)];
194     }
195
196     public void onScannedRobot(ScannedRobotEvent e) {
197         // update preS, preA; update curS
198         preS = curS;
199         preA = curA;
200         curS = new State(getEnergy(), e.getEnergy(), e.getDistance());
201         oB = e.getBearing();
202         eH = e.getHeading();
203
204
205         this.operationalMode = enumOperationalMode.performAction;
206     }
207
208 //    public void replayExperience(ReplayMemory rm){
209 //        int ms = rm.sizeOf();
210 //        int requestedSs = (ms < MAX)
211 //    }
212
213     public void onWin(WinEvent e) {
214         System.out.println("I win!!!!!!!!!!!!");
215         numWins++;
216
217         curR = goodTerminalReward;
218         totalR += curR;
219
220         lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));
221         //TODO: can add stat
222     }
223
224     public void onDeath(DeathEvent e) {
225         System.out.println("I lose.");
226         curR = badTerminalReward;
227         totalR += curR;
228
229         lut.train(preS.transformToX(preA), computeQ(preS, curS, curR));
230         //TODO: can add stat
231     }
232
233     ////////////////////////intermediate rewards start//////////
234     public void onBulletHit(BulletHitEvent e) {
235         curR = +0.4;
236         totalR += curR;
237
238         //lut.train(preS.transformToX(), computeQ(preS, curS, curR));
239     }
240
241     public void onBulletMissed(BulletMissedEvent e) {
242         curR = -0.01;
243         totalR += curR;
244     }
245
246     public void onHitByBullet(HitByBulletEvent event) {
247         curR = -0.2;
248         totalR += curR;
249     }
250
251     public void onHitWall(HitWallEvent event) {
252         curR = -0.01;
253         totalR += curR;
254     }
255
256     ////////////////////////intermediate rewards end//////////
257     public void onRoundEnded(RoundEndedEvent event) {
258
259         actionList.add(99);
260         if (totalNumRounds == numOfRoundsToDecayE) {
261             WriteQ();
262         }
263         if (totalNumRounds < numOfRoundsToDecayE) {
264 //            if(epsilon > 0 & epsilon >decayEStepSize){
265 //                epsilon -= decayEStepSize; // so e decaying to 0 in the first 80% round
266 //            }
267         } else {
268             epsilon = 0;
269         }
270
271         totalNumRounds++;
272         if (totalNumRounds % 100 == 0) {
273             int index = totalNumRounds / 100 - 1;
274             winRatePer100[index] = numWins;
275             epsilonList[index] = epsilon;
```

```
276
277             out.println("The round has ended and the winRatePer100[] updated");
278             out.println("totalNumRounds" + totalNumRounds);
279             out.println("winRatePer100" + winRatePer100[index]);
280             out.println("numWins" + numWins);
281             numWins = 0; // reset
282             out.println("numWins set to 0 again");
283
284         }
285         System.out.println("round ended");
286
287     }
288
289     private void WriteQ() {
290         PrintStream w = null;
291         try {
292             w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileName1)));
293
294             double[] qs = lut.getQValues();
295             for (double q : qs) {
296                 w.println(q);
297             }            // PrintStreams don't throw IOExceptions during prints, they simply set a flag.... so check it here.
298             if (w.checkError()) {
299                 out.println("I could not finalWriteQ!");
300             }
301         } catch (IOException e) {
302             out.println("IOException trying to write: ");
303             e.printStackTrace(out);
304         } finally {
305             if (w != null) {
306                 w.close();
307             }
308         }
309     }
310
311     public void onBattleEnded(BattleEndedEvent e) {
312         finalWriteQ();
313         finalWriteWins();
314         finalWriteEpsilonList();
315 //        finalWriteActionList();
316     }
317
318     private void finalWriteEpsilonList() {
319         PrintStream w = null;
320         try {
321             w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileNameEpsilonList)));
322
323             for (double e : epsilonList) {
324                 w.println(e);
325             }
326
327             if (w.checkError()) {
328                 out.println("I could not write the finalWriteEpsilonList!");
329             }
330         } catch (IOException e) {
331             out.println("IOException trying to write: ");
332             e.printStackTrace(out);
333         } finally {
334             if (w != null) {
335                 w.close();
336             }
337         }
338     }
339
340     private void finalWriteWins() {
341         PrintStream w = null;
342         try {
343             w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileNameWinRate)));
344
345             for (double winR : winRatePer100) {
346                 w.println(winR);
347             }
348
349             if (w.checkError()) {
350                 out.println("I could not write the winRatePer100!");
351             }
352         } catch (IOException e) {
353             out.println("IOException trying to write: ");
354             e.printStackTrace(out);
355         } finally {
356             if (w != null) {
357                 w.close();
358             }
359         }
360     }
361
362     private void finalWriteQ() {
363         PrintStream w = null;
364         try {
365             w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileName)));
366
367             double[] qs = lut.getQValues();
368             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();
            }
        }
    }

    private void finalWriteActionList() {
        PrintStream w = null;
        try {
            w = new PrintStream(new RobocodeFileOutputStream(getDataFile(logFileNameActions)));

            for (int action : actionList) {
                w.println(action);
            }

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

    public enum enumOperationalMode {scan, performAction}
}
```

State.java

```java
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;
```

```
82          X[3] = this.isCloseToW;
83          X[4] = this.isFaster;
84          X[5] = a.ordinal();
85          return X;
86      }
87
88      private int computeIsFaster(double oV, double myV) {
89          if (oV < myV){
90              return 1; // faster than the opponent
91          }else{
92              return -1;
93          }
94      }
95
96      private int computeEnergyLevel(double e) {
97          double ratio = e / 100.0;
98          int output = (int) Math.ceil(1 + ratio * (numOfLevelForEnergy-1));
99          return output >5? 5: output; //energy can actually go beyond 100
100         //return (int) Math.ceil(1 + ratio * (numOfLevelForEnergy-1));
101         //return numOfLevelForEnergy - (int) Math.round(myE / numOfLevelForEnergy);
102     }
103
104     private int computeTooCloseToWall(double myX, double myY) {
105         double YtoWall = Math.min(myY, 600-myY);
106         double XtoWall = Math.min(myX, 800-myX);
107         double toWall = Math.min(YtoWall, XtoWall);
108         if (toWall < disForTooCloseToWall){
109             return 1; // to close to wall!
110         }else{
111             return -1;
112         }
113     }
114
115     private int computeDistanceLevel(double oD) {
116         double ratio = oD / 1000.0;
117         return (int) Math.ceil(1 + ratio * (numOfLevelForDistance-1));
118         //return numOfLevelForDistance- (int) Math.round(oD/numOfLevelForDistance);
119     }
120
121 }
122
```

```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;
```

```
88          } catch (ArrayIndexOutOfBoundsException e) {
89              System.out.println("Error: " + e.getMessage());
90              for(double x: X){
91                  System.out.println(x);
92              }
93          }
94          return output;
95          // return qValues[indexFor(X)];
96      }
97      public double[] getQValues(){
98          return qValues;
99      }
100
101     @Override
102     public double train(double[] X, double argValue) {
103         qValues[indexFor(X)] = argValue;
104         return 0;
105     }
106
107     @Override
108     public void save(File argFile) {
109
110     }
111
112     @Override
113     public void load(String argFileName) throws IOException {
114
115     }
116
117 }
118
```

Neuron.java

```java
import java.util.Random;
public class Neuron {
    public double argA;
    public double argB;
    public double[] weights;

    public double[] lastWeightsChanges;
    public double output;

    public double weightedSum;

    // error signal
    public double delta;

    final double bias = 1.0;



    public Neuron(int argNumInputs, double argA, double argB){
        // initialize the weights for the neuron (including the bias)
        this.weights = new double[argNumInputs+1];
        this.lastWeightsChanges = new double[weights.length];
        this.argA = argA;
        this.argB = argB;
    }

    public void initializeWeight(){
        for(int i=0; i< weights.length; i++){
            Random random = new Random();
            weights[i] = -0.5 + random.nextDouble();
        }
    }

    public double[] getLastWeightsChanges(){
        return this.lastWeightsChanges;
    }



    public double[] getWeights(){
        return this.weights;
    }
    public void setWeights(double[] newWeights){
        double[] oldWeights = this.weights;
        double[] changes = new double[oldWeights.length];
        if(newWeights.length != this.weights.length){
            System.out.println("newWeights length does not match the weights for this neuron!");
        }
        for(int i = 0; i < oldWeights.length; i ++){
            changes[i] = newWeights[i] - oldWeights[i];
        }
        this.weights = newWeights;
        this.lastWeightsChanges = changes;
    }

    public double getWS(){
        return this.weightedSum;
    }
    public void updateOutput(double[] inputs){
        if(inputs.length != this.weights.length-1){
            System.out.println("inputs length don't match the weights for this neuron!");
            System.out.println("inputs length" + inputs.length + " and weights length " + this.weights.length);
        }
        double sum = bias * weights[weights.length-1];
        for (int i = 0; i < inputs.length; i++) {
            sum += inputs[i] * weights[i];
        }
        this.output = customSigmoid(sum);
        this.weightedSum = sum;
    }

    public double getOutput(){
        return this.output;
```

```
    }


    public double sigmoid(double x) {
        return 2 / (1+Math.exp(-x)) - 1;
    }


    public double customSigmoid(double x) {
        return (argB-argA) / (1+Math.exp(-x)) + argA;
    }
    public double getDelta(){
        return this.delta;
    }

    public void setDelta(double delta){
        this.delta = delta;
    }


}
```

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

public class NeuralNet implements NerualNetInterface{
    public double argA;
    public double argB;
    public Neuron[] hiddenNeurons;
    public Neuron outputNeuron;
    public double learningRate;
    public double momentumTerm;
    public int backwardPropagationVersion;


    /**
     * Constructor. (Cannot be declared in an interface, but your implementation will need one)
     *
     * @param argNumInputs    The number of inputs in your input vector
     * @param argNumHidden    The number of hidden neurons in your hidden layer. Only a single hidden layer is supported
     * @param argLearningRate The learning rate coefficient
     * @param argMomentumTerm The momentum coefficient
     * @param argA            Integer lower bound of sigmoid used by the output neuron only.
     * @param argB            Integer upper bound of sigmoid used by the output neuron only.
     **/
    public  NeuralNet(
            int argNumInputs,
            int argNumHidden,
            double argLearningRate,
            double argMomentumTerm,
            double argA,
            double argB,
            int backwardPropagationVersion){
        // initialize variables for the NN
        this.backwardPropagationVersion = backwardPropagationVersion;
        this.argA=argA;
        this.argB=argB;
        this.learningRate = argLearningRate;
        this.momentumTerm = argMomentumTerm;
        // initialize hidden cells
        this.hiddenNeurons = new Neuron[argNumHidden];
        for(int i = 0; i < hiddenNeurons.length; i++){
           hiddenNeurons[i] = new Neuron(argNumInputs, argA, argB);
        }
        // initialize output cell
        this.outputNeuron = new Neuron(hiddenNeurons.length, argA, argB);

        initializeWeights();
        };

    @Override
    public double sigmoid(double x) {
        return 2 / (1+Math.exp(-x)) - 1;
    }

    @Override
    public double customSigmoid(double x) {
        return (argB-argA) / (1+Math.exp(-x)) + argA;
    }

    public double deri(double x) {
        return (argB-argA) * Math.pow((1+Math.exp(-x)), -2) * Math.exp(-x);
    }

    @Override
    public void initializeWeights() {
        for(int i = 0; i < hiddenNeurons.length; i++){
            hiddenNeurons[i].initializeWeight();
        }
        outputNeuron.initializeWeight();
    }

    @Override
    public void zeroWeights() {

    }

    @Override
    public double outputFor(double[] X) {
        double[] hiddenLayerOutputs = new double[this.hiddenNeurons.length];
        for(int i = 0; i<hiddenLayerOutputs.length; i++){
            hiddenNeurons[i].updateOutput(X);
            hiddenLayerOutputs[i] = hiddenNeurons[i].getOutput();
        }
        outputNeuron.updateOutput(hiddenLayerOutputs);
        return outputNeuron.getOutput();
    }

    /**
     * Given X, update outputs for hidden neurons and output neuron
     * @param X input vector
```

```
     */
    public void forward(double[] X){
        double[] hiddenLayerOutputs = new double[this.hiddenNeurons.length];
        for(int i = 0; i<hiddenNeurons.length; i++){
            hiddenNeurons[i].updateOutput(X);
            hiddenLayerOutputs[i] = hiddenNeurons[i].getOutput();
        }
        outputNeuron.updateOutput(hiddenLayerOutputs);
    }

    /**
     * backward propagation: update delta for output neuron and hidden neurons
     * and update weights for output neuron and hidden neurons
     * (also update the lastWeightsChanges)
     *
     * version 1 and version 2 use different orders
     */

    public void backwardPropagationVersion1(double[] X, double argValue){
        updateErrorSignalOutputN(argValue);
        updateErrorSignalHiddenNs();
        updateWeightsOutputN();
        updateWeightsHiddenNs(X);
    }

    /**
     * backward propagation: update delta for output neuron and hidden neurons
     * and update weights for output neuron and hidden neurons
     * (also update the lastWeightsChanges)
     *
     * version 1 and version 2 use different orders
     */
    public void backwardPropagationVersion2(double[] X, double argValue){
        updateErrorSignalOutputN(argValue);
        updateWeightsOutputN();
        updateErrorSignalHiddenNs();
        updateWeightsHiddenNs(X);
    }

    private void updateErrorSignalOutputN(double argValue) {
        // update and set up the delta for output cell
        double y_o = outputNeuron.getOutput();
        double ws_o = outputNeuron.getWS();
        double deri_o = deri(ws_o);
        double δ_o = (argValue-y_o) * (deri_o);
        outputNeuron.setDelta(δ_o);
    }
    private void updateErrorSignalHiddenNs() {
        // update and set up the delta for hidden cells
        for(int i = 0; i < hiddenNeurons.length; i++){
            double y_hi = hiddenNeurons[i].getOutput();
            double ws_hi = hiddenNeurons[i].getWS();
            double deri_hi = deri(ws_hi);
            double δ_hi = outputNeuron.weights[i] * outputNeuron.getDelta() * (deri_hi);
            hiddenNeurons[i].setDelta(δ_hi);
        }

    }
    private void updateWeightsOutputN() {
        double δ_o = outputNeuron.getDelta();
        // update weights for output cell
//        w[i]* = w[i] + ϱ * δ_o * y_hi + α * Δw[i];
        double[] oldW_o = outputNeuron.getWeights();
        double[] newW_o = new double[oldW_o.length];
        double[] Δw_o = outputNeuron.getLastWeightsChanges();
        // update everything but the weight for bias
        for(int i = 0; i < newW_o.length-1; i++){
            double y_hi = hiddenNeurons[i].getOutput();
            newW_o[i] = oldW_o[i] + this.learningRate * δ_o * y_hi + this.momentumTerm * Δw_o[i];
        }
        // update weight for bias
        int indexForBias = newW_o.length-1;
        newW_o[indexForBias] = oldW_o[indexForBias] + this.learningRate * δ_o * bias + this.momentumTerm * Δw_o[indexForBias];
        // set up the weights back
        outputNeuron.setWeights(newW_o);

    }
    private void updateWeightsHiddenNs(double[] X) {
        // for each, update weights for hidden cells
        for(int i = 0; i < hiddenNeurons.length; i++){
            double[] oldW_hi = hiddenNeurons[i].getWeights();
            double[] newW_hi = new double[oldW_hi.length];
            double[] Δw_hi = hiddenNeurons[i].getLastWeightsChanges();
            double δ_hi = hiddenNeurons[i].getDelta();
            // update everything but the weight for bias
            for(int j = 0; j < newW_hi.length-1; j++){
                newW_hi[j] = oldW_hi[j] + this.learningRate * δ_hi * X[j] + this.momentumTerm * Δw_hi[j];
            }
            // update weight for bias
            int indexForBias = newW_hi.length-1;
            newW_hi[indexForBias] = oldW_hi[indexForBias] + this.learningRate * δ_hi * bias + this.momentumTerm * Δw_hi[indexForBias];
            // set up the weights back
            hiddenNeurons[i].setWeights(newW_hi);
```

```
        }
    }

    @Override
    public double train(double[] X, double argValue) {
        /*
         * For each pattern:
         *  using train data and weights to do forward calculation
         *  get error signals through backward propagation
         *  update weights for each cell
         *
         * After training all pattern, check if total error is acceptable,
         * if yes, stop
         * otherwise, repeat training
         */
        forward(X);
        switch(this.backwardPropagationVersion){
            case 1:
                backwardPropagationVersion1(X, argValue);
                break;
            case 2:
                backwardPropagationVersion2(X, argValue);
                break;
        }
        return (0.5 * Math.pow((outputFor(X)- argValue), 2));
    }

    @Override
    public void save(File argFile) {

    }

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

    }

    /**
     * helper function to print the NN
     */
    public void printNN(){
//        System.out.println(outputNeuron.getWeights().length); //5
//        System.out.println(hiddenNeurons.length); // 4
        System.out.println("-------------print NN----------------");
        System.out.println("output Neuron weights:");
        int i = 0;
        while(i < 5){
            System.out.println(outputNeuron.getWeights()[i]);
            i++;
        }

        i = 0;
        while(i < 4){
            System.out.println("hidden Neuron "+ i + " weights:");
            int j = 0;
            while(j<3){
                System.out.println(hiddenNeurons[i].getWeights()[j]);
                j++;
            }
            i++;
        }
        System.out.println("-------------end----------------");

    }
}
```

CommonInterface.java

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

/**
 * This interface is common to both the Neural Net and LUT interfaces.
 * The idea is that you should be able to easily switch the LUT
 * for the Neural Net since the interfaces are identical
 * @date 20 June 2012
 * @author sarbjit
 */
public interface CommonInterface {
    /**
     * @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(double [] X);

    /**
     * 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(double [] X, double argValue);

    /**
     * A method to write either a LUT or weights of a neural net to a File
     * @param argFile of type File
     */
    public void save(File argFile);

    /**
     * Loads the LUT or NN weights from file. The load must of course
     * have knowledge of how the 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 nerual net whose structure does not match
     * the data in the file. (e.g., wrong number of hidden neurons)
     * @param argFileName
     * @throws IOException
     */
    public void load(String argFileName) throws IOException;
}
```

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);
}

```

NerualNetInterface.java

```java
public interface NerualNetInterface extends CommonInterface{
    final double bias = 1.0; // The input for each neurons bias weight

    /**
     * Return a bipolar sigmoid of the input X
     * @param x The input
     * @return f(x) = 2 / (1+e(-x)) - 1
     */
    public double sigmoid(double x);

    /**
     * This method implements a general sigmoid with asymptotes bounded by (a,b)
     * @param x The input
     * @return f(x) = b_minus_a / (1+e(-x)) - minus_a
     */
    public double customSigmoid(double x);

    /**
     * Initialize the weights to random values.
     * For say 2 inputs, the input vector is [0] & [1]. We add [2] for the bias.
     * Like wise for hidden units. For say 2 hidden units which are stored in an array.
     * [0] & [1] are the hidden & [2] the bias.
     * We also initialise the last weight change arrays. This is to implement the alpha term.
     */
    public void initializeWeights();

    /**
     * Initialize the weights to 0.
     */
    public void zeroWeights();
}
```