UBC-CPEN-502 Assignment 3

Reinforcement Learning with Backpropagation

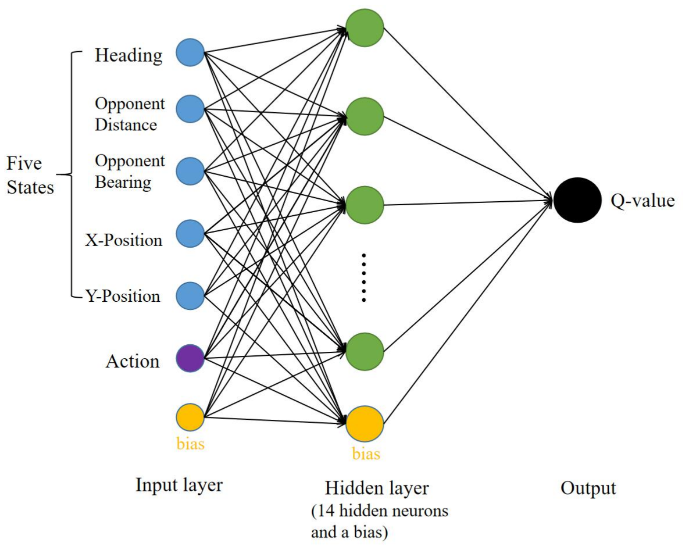
Name: Yue Shen Student #: 76006386

Questions

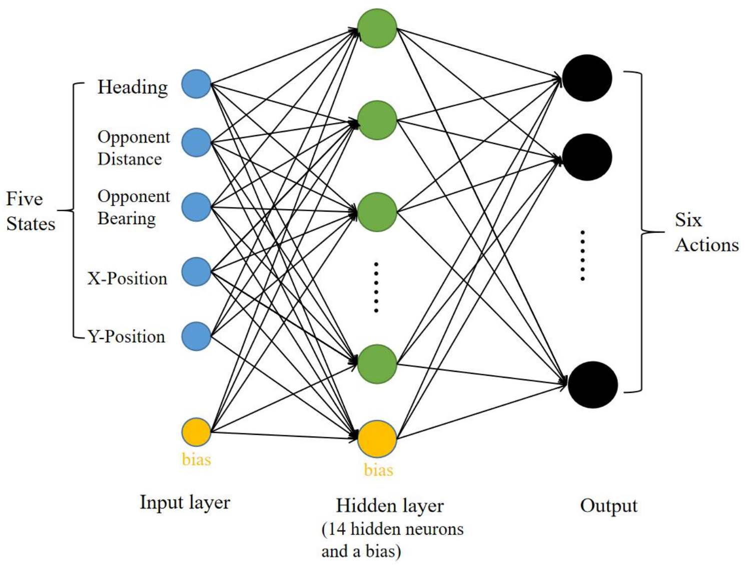
(4) The use of a neural network to replace the look-up table and approximate the Q-function has some disadvantages and advantages.

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)

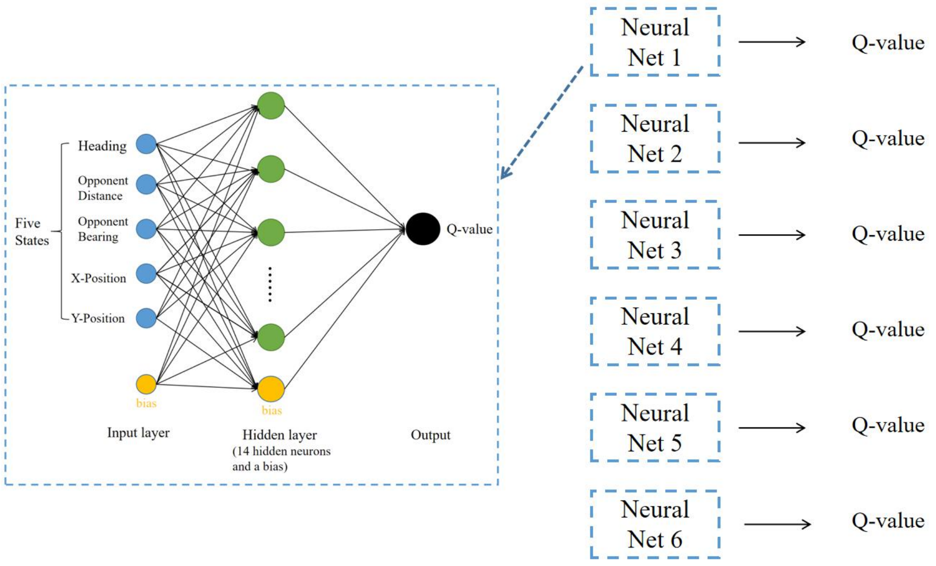
1. The first architecture has one neural net, taking the state-action data as inputs, and then outputs a single Q-value.



2. The second architecture has one neural net, taking all the states as inputs. It outputs a Q-value for every different action.



3. The third architecture has the same number of neural net as the number of actions. Internal structure of each neural net is show below, taking all the states as inputs, and outputs a Q-value for each action.



I choose the third architecture with 6 neural nets. The first reason is that the BP network created in Part 1 has only one output, which is not easy to be extended to several outputs, so we dismissed the second architecture . What's more, it's better to have separated networks for each action so that they won't share the weights and the same output. Final result could be accurate when approximating the Q-function.

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)

1. Active Type

In the training process, I normalize Q value between the range of -1 to 1 so that I use sigmoid function for Bipolar representation.

2. Find the best number of hidden neurons

In order to set the best number of hidden neurons, I tried many different number of hidden neurons (from 1 - 20) and calculate the RMS (Root Mean Square) respectively. Other settings of parameters are as Table-1 , the Figure-1 shows the RMS vs different number of hidden neurons after 2000 times training.

Table-1 Settings for best number of hidden neurons

Active Type	**Number of Hidden Neurons**	Learning rates	Momentum
Bipolar	1 - 20	0.1	0.9

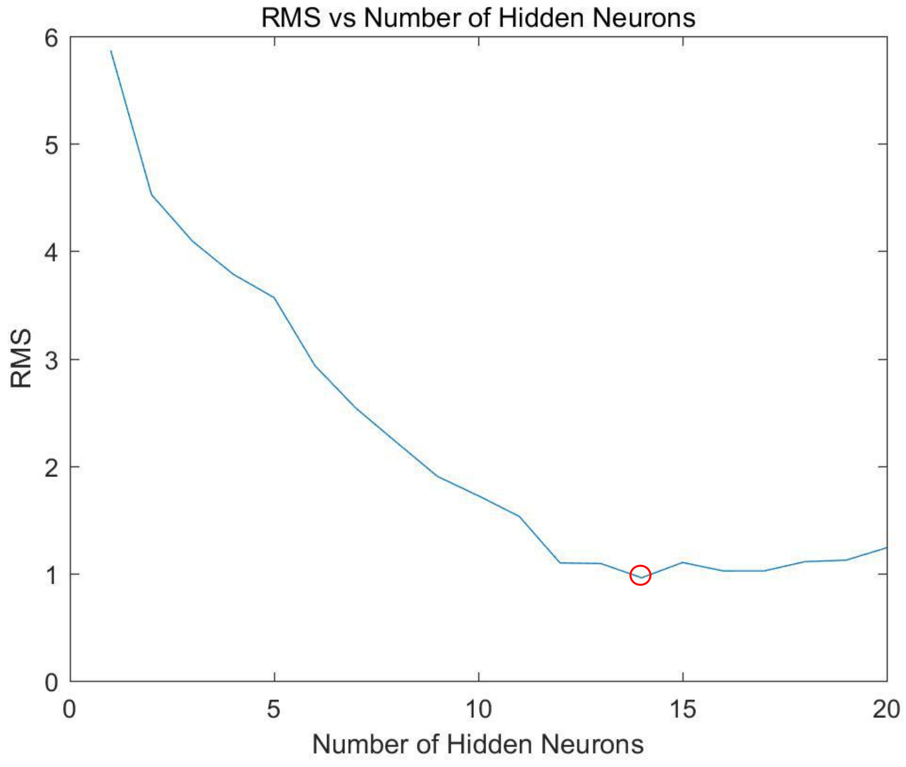


Figure-1 RMS vs Number of Hidden Neurons

As we can see from Figure-1, RMS is the lowest when the number of hidden neurons is 14 (red circle in the Figure-1) , hence we set 14 hidden neurons and the find the other hyper-parameters continuously.

3. Find the best Learning rates

Figure-2 show the RMS vs five different learning rates (from 0.1 to 0.9) after 2000 times training. Other settings are as Table-2.

Table-2 Settings for best learning rate

Active Type	Number of Hidden Neurons	**Learning rates**	Momentum
Bipolar	14	0.1 - 0.9	0.9

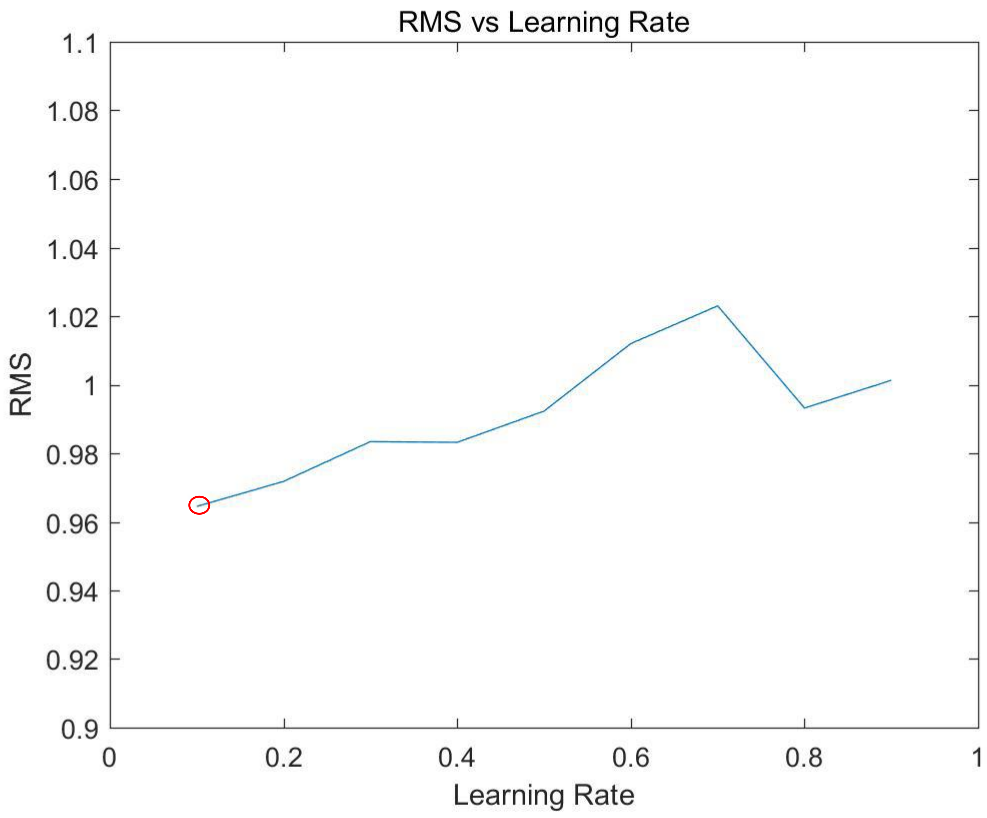


Figure-2 RMS vs Learning Rate

Hence we can determine the learning date to 0.1 since it outputs the minimum RMS (red circle in the Figure-2).

4. Momentum

When Momentum is 0.9, my robot increase its winning rates in the first 2500 iterations. If I set a smaller Momentum, my robot needs more iterations and more time. So I didn't make comparison between different momentum but set momentum always to be 0.9.

Final Best result

All in all, my settings for four hyper-parameters are as Table-3, the RMS decreasing trend during 5000 iterations are as Figure-3 followed.

Table-3 Final Settings

Active Type	Number of Hidden Neurons	Learning rates	Momentum
Bipolar	14	0.1	0.9

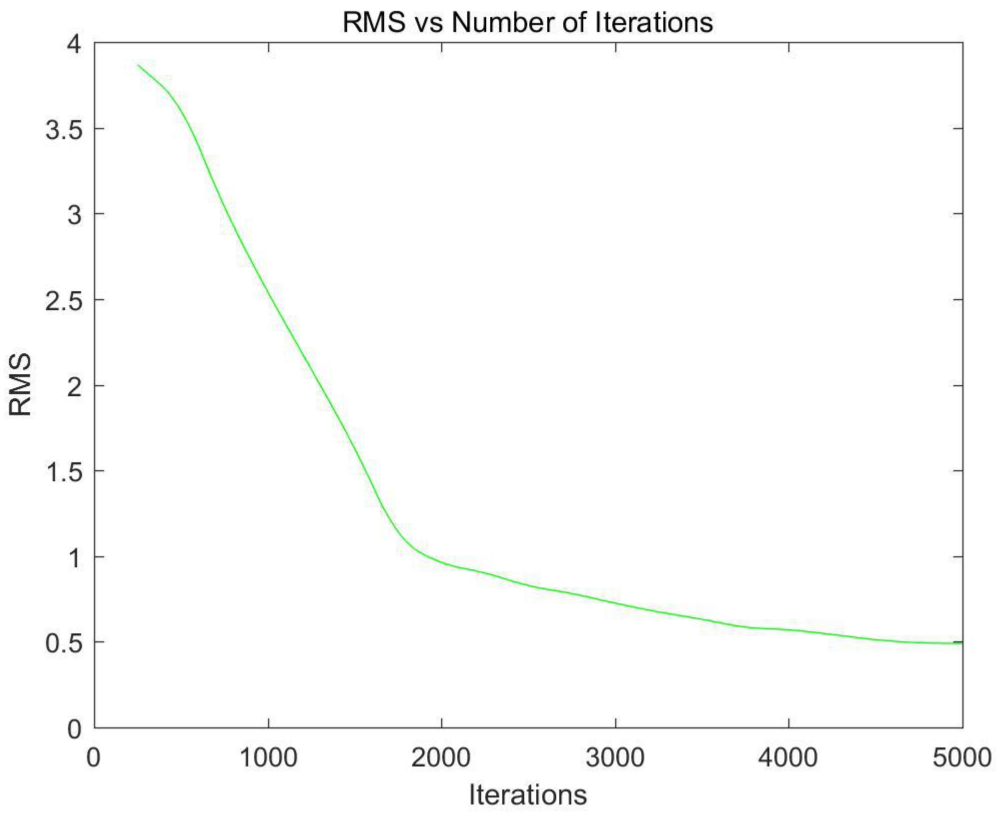


Figure-3 Final Result (RMS vs 5000 Iterations)

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)

For Look Up Table, if without quantization for the state, in the example of my robot states, there have 360 headings, 1000 distances, 360 bearings, 800 X-Positions and 600 Y-Position. Hence to store all the data for states, we need $360 \times 1000 \times 360 \times 800 \times 600 = 6.22 \times 10^{13}$ spaces, resulting in a huge Look Up Table. Hence it will take up too much memory and decrease the learning speed.

For Neural Network, however, we do not store the Q value to a table anymore since the different Q values of different states-actions pairs are just the training data of the neural network. Only we need to do is to store the weights between each layers, which results in smaller memory and better training.

(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.

a) Identify <u>two</u> 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)

As same as part 2, I choose robot *TrackFire* as my opponent.

1. Winning Rate

The first metric that I choose is winning Rate. The total number of rounds (or iterations) is 5000. With every 100 rounds, I calculate the winning rate. The increasing trend of winning rate vs 50 × 100 rounds is as follows:

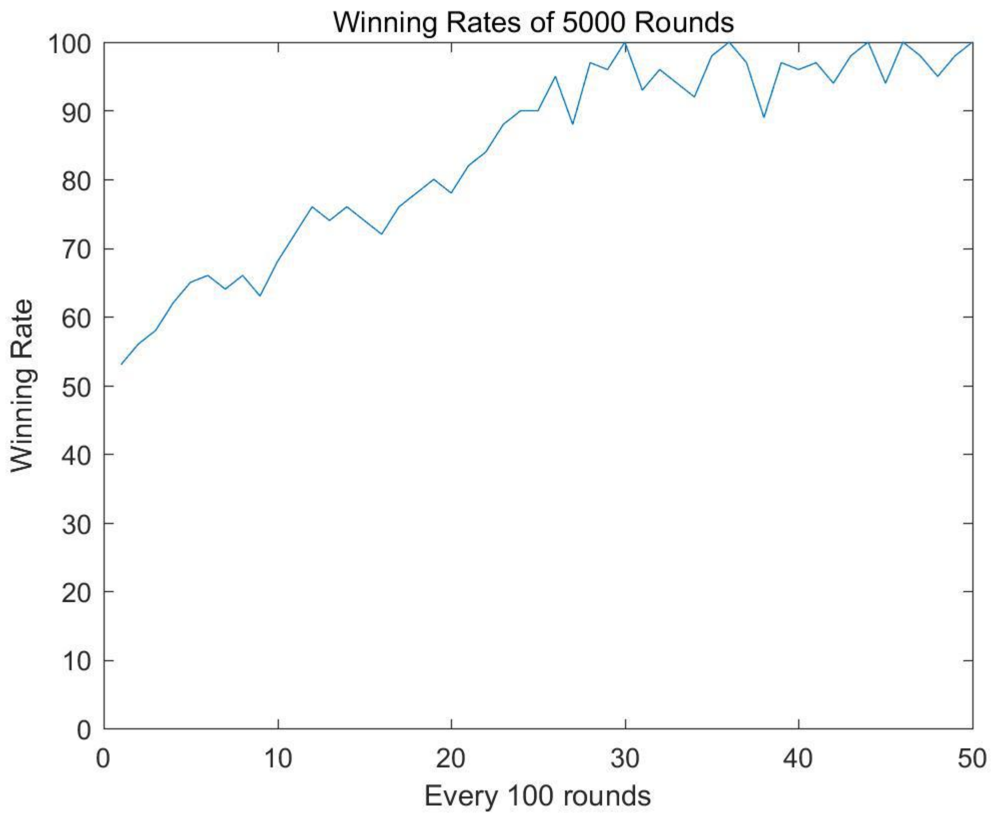


Figure-4 First Metric: Wining Rates

2. Error *e(s)*

For the two different Q values generated by an old state-action pair and a new state-action pair respectively, I calculate *e(s)=abs(Q(s', a')-Q(s, a)).*

With every 100 rounds, I calculate the average *e*. The trend of *e* versus increasing number of iterations is as Figure-5. We can find that the value of *e* decreases as the increasing rounds, which means my robot is learning by itself and performs better and better.

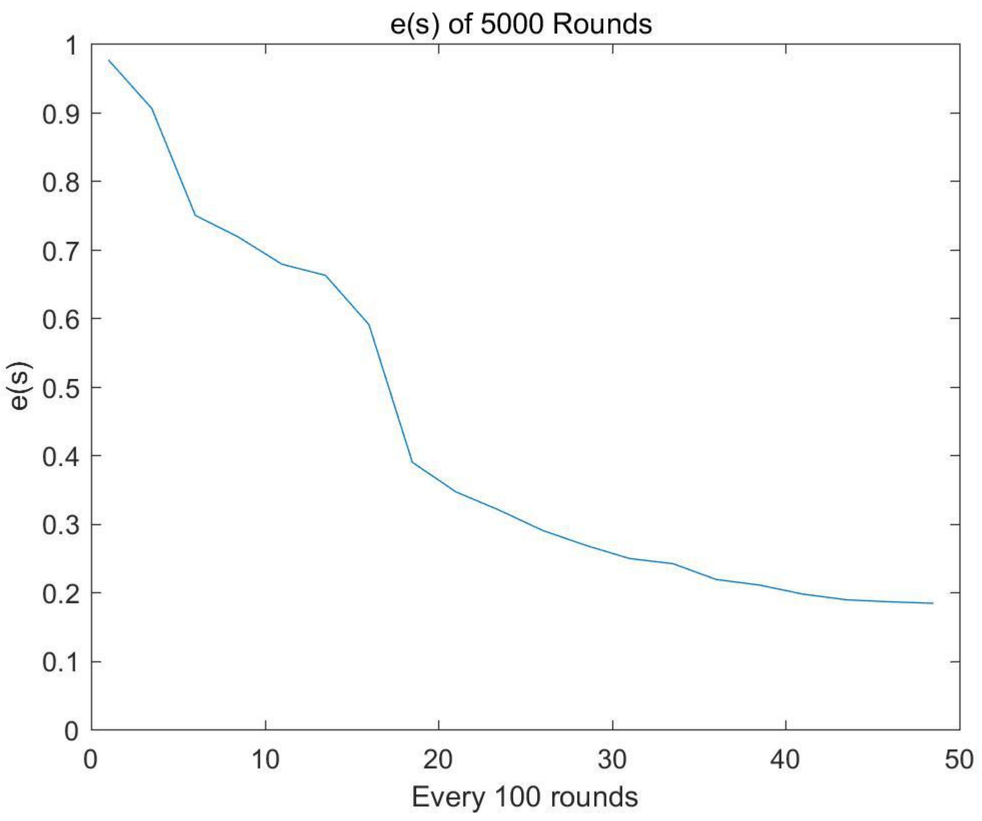


Figure-5   Second Metric: Error e(s)

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

The BLUE line shows the performance when discount factor $\gamma = 0.9$;
The RED line show the performance when discount factor $\gamma = 0.4$;
The PURPLE line show the performance when discount factor $\gamma = 0.0$;

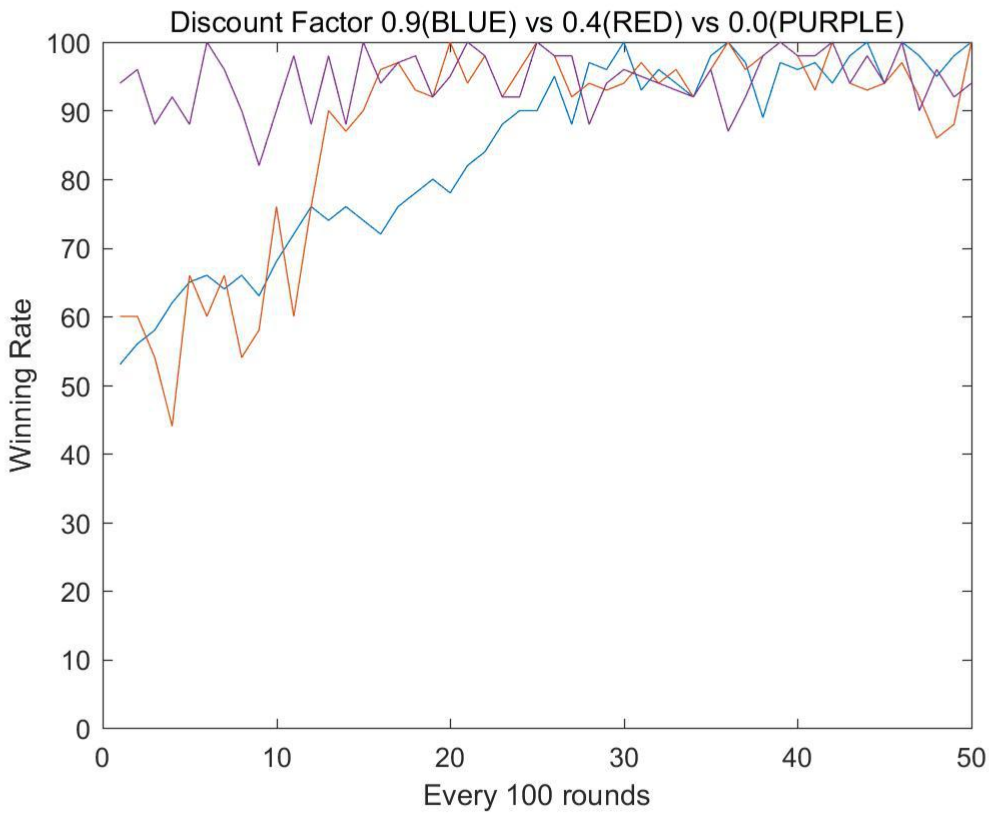


Figure-6 Wining Rates of Three Different Discount Factor

As we can see, when discount factor is 0.0, my robot doesn't learn by itself and the wining rate always fluctuates from 85% to 100%. When when discount factor is 0.4, my robot learns by itself only through first 1500 rounds, which is faster than 0.9 discount factor. However, the final wining rate reached in three cases are similar. My robot doesn't perform better due to the decrease of discount factor since this parameter only determines whether focus on the future rewards or past rewards more. However, it cannot change the final performance. There is another possibility that because of my robot can reach 100% winning rate in the end so that my result data cannot reflect the influence of discount factor.

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 Q-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)

We have Bellman equation as follows:

$$V(s_t) = r_t + \gamma V(s_{t+1})$$

Where $V(s_t)$ and $V(s_{t+1})$ are the value functions for the state at time *t* and *t*+1;

r_t is the rewards at time *t*; γ is the discount factor;

Let $V^*(s_t)$ denotes the optimal value function, thus we have:

$$V^*(s_t) = r_t + \gamma V^*(s_{t+1})$$

Let $e(s_t)$ denotes the error for the state at time *t*. $e(s_t)$ can be expressed as:

$$e(s_t) = V(s_t) - V^*(s_t)$$

$e(s_{t+1})$ can be expressed as: $e(s_{t+1}) = V(s_{t+1}) - V^*(s_{t+1})$

Thus we have: $e(s_t) + V^*(s_t) = V(s_t) = r_t + \gamma V(s_{t+1}) = r_t + \gamma\left(e(s_{t+1}) + V^*(s_{t+1})\right)$

We could minus $V^*(s_t)$ for both side of the equation above, then we have:

$$e(s_t) = \gamma e(s_{t+1})$$

So we know that the relationship of error between the next state and the current state are similar to the relationship of value function.

Under this circumstance, when we use TD learning (Q-learning) algorithm to train our robots, we could assume there is no error for the terminal state and then propagate the future value backwards. Hence after training enough times, the Q-value will converge to a stable value.

However, When Q-function is approximated, the $e(s_t)$ is no longer equal to $\gamma e(s_{t+1})$, and will bring in another error during the approximation. The process of backpropagation is aim to update weights in order to reach a good value and the neural network will not guarantee a local minima. Hence the value of error may not become zero in the end.

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 Q-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)

During the online learning, since there is no a-priori training set, there is no comparison between our own outputs and the ideal outputs, we could not calculate the total error. In this case we have to evaluate the performance during the online learning or we can say during the battles.

The first evaluation criterion is the state-action pair error *e(s)=abs(Q(s', a')-Q(s, a))*. To calculate the difference between the Q-value in the current state-action pair and previous state-action pair, we can see whether Q-value is converging or not. It it is converging, the value of error *e(s)* will be smaller and smaller and become zero in the end.

The second evaluation criterion is winning rate during the battles. For training, we may set a number of battles, so we can calculate the winning rate for every several battles with same number. If the winning rate is increasing, it means our robot is learning and performs better and better.

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)

The BLUE line shows the performance when $n = 1$;
The RED line show the performance when $n = 3$;
The PURPLE line show the performance $n = 7$;

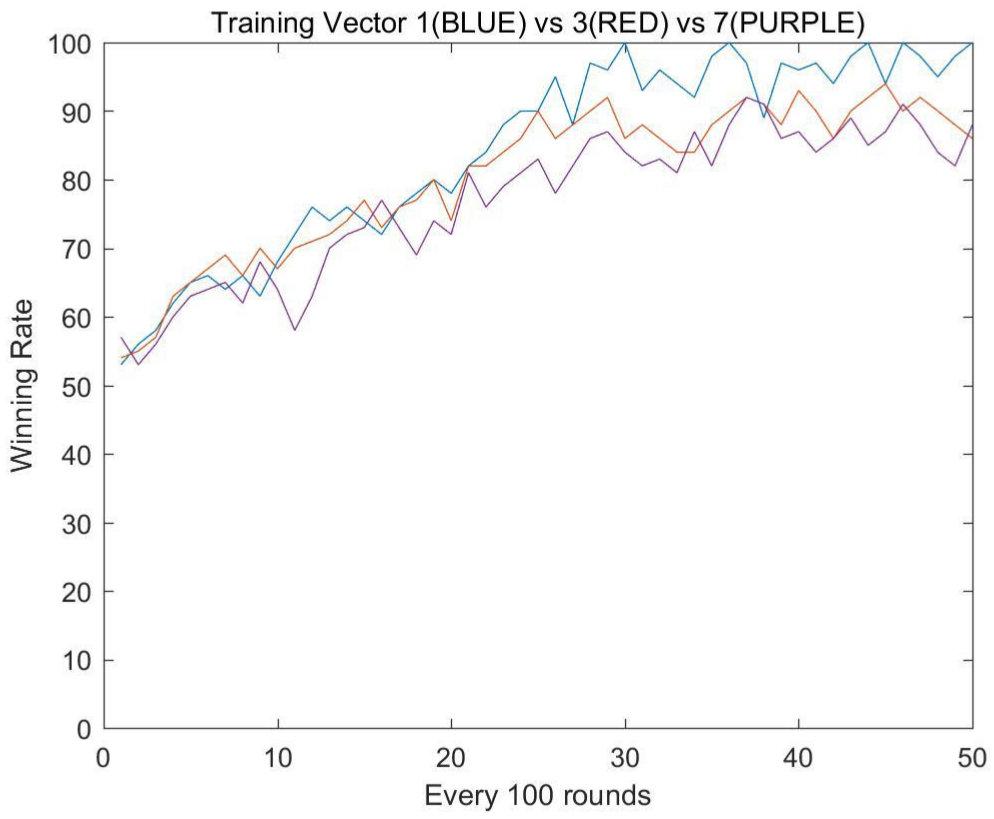


Figure-7 Wining Rates of Three Different Training Vector

When the number of training vector increases, my robot is still learning but doesn't perform well than the single vector. I think the increasing times of normalization of Q-function contributes to this phenomenon.

(6) Overall Conclusions

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)

1. Practical issues surrounding the application of RL & BP

In order to reach a better performance of robot, we need a larger kinds of states and actions so that more error will be caused in when Q-Function is approximated. Consequently, a more complicated neural network with larger number of hidden neurons will be needed, which results in the longer training time as well as more iterations needed, and even may cause overfitting.

One kind of Neural Network with RL & BP technically cannot be appropriate for beating all kinds of robots. Since I use the Look Up Table from Part 2 when battling with one robot to train the neural network, the NN I get in Part 3 have better performance only when my robot battle with the same sample robot.

The parameter setting (momentum, learning rate .etc) for the training is crucial and but difficult to find the optimum one. And I always spend lots of time trying different setting and train the NN for many times.

2. Methods to improve the performance of robot

Basically, different architecture of Neural Net may result in different performance of robot. It's better to have one Neural Net for each action, saying that the number of Neural Net is the number of actions. In this case, every action will not have the same weight and the same output. When approximating the Q-value, it may get more accurate value so that reduce the value of Root Mean Square.

As discussed before, using a better hyper-parameter setting leads to a beter performance of robot. The number of hidden neurons used has a great influence on performance with regard to the Q-function approximation.

3. Suggestions of convergence problems

The first one is to choose another activation function for backpropagation. Sigmoid function may result in the disappearance of gradient when the input is too large or too small and have low speed of convergence. In practice, Tanh function is better than sigmoid function or we can use ReLU with batch normalization to insure the inputs of each layer have similar distribution.

The second one is to use variable step size instead of constant step size. Also, try to find a better parameter setting and use different initial weight to train the neural net.

4. Advice on applying RL to other practical applications

When it comes to a practical application, we'd better choose the appropriate states and actions for the specific problem. They are independent factors and do not influence each other. Also, design appropriate neural net structure for different problems. Try as much as possible to find the optimum parameter settings.

What's more, it's better to apply the Look Up Table that we get when doing the offline training to the online training. Trough this way, the training process as well as the speed of convergence could be faster than simply using random weight. will be the result could be more accurate.

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)

There are few concerns that I can think of. The first one is the training data used in the process of training the neural net. Since the medical situations are various and cannot be concluded in several kinds, it may difficult to determine the states and actions.

Also, delivering anesthetic to a patient is the real life problem and tolerant nearly zero error. However, using reinforcement learning may cause false delivery even if it'll converge to a desired value. So there is full of risk using this approach.

To alleviate those concerns, we should improve the quality of training data at the same time strengthen the risk control. We could use other method in machine learning like Random Forest and Decision Tree to forecast the potential risk of disease then select the optimum choice.

Appendix

1. LearningNN.java

```java
package com.shenyue2;

import java.util.ArrayList;
import java.util.Random;

public class LearningNN {
    public static final double LearningRate = 0.1;
    public static final double discountRate = 0.9;
    public static double explorationRate = 0.0;
    public LUT LUTable;

   //Neural Net Setting
    public static int netNum = 6; // Number of actions.
    public static int inputsNum = 5; //Number of states
    public static int hiddenNum = 14;
    public static int outputsNum = 1;
    public static double LearningRate_NN = 0.2;
    public static double momentumTerm = 0.9;
    public static double argA = -1;
    public static double argB = 1;
    public double[] minQ = new double [LUT.actionsNum];
    public double[] maxQ = new double [LUT.actionsNum];
    public int[] currentState = new int [inputsNum];
    public int[]  newState = new int [inputsNum];
    public double currentAction [] = new double [netNum];
    public double newAction[] = new double [netNum];
    public double currentQValue[] = new double [netNum];
    public double newQValue[] =  new double [netNum];

    public ArrayList<NeuralNet> NetList = new ArrayList<NeuralNet>();
    public double error;

    public LearningNN(LUT LUTable) {
        this.LUTable = LUTable;
        for(int i = 0; i<LUT.actionsNum;i++) {
            maxQ[i] = 5+LUTNeuralNet.getMaxQ(getColumn(LUTable.getLUTable(),i));
            minQ[i] = 
-5+LUTNeuralNet.getMinQ(getColumn(LUTable.getLUTable(),i));
        }

        error=0.0;
        }


    public int selectAction_NN() {
        double random = Math.random();
        int action = 0;
        double [] input = LUTNeuralNet.normalizeInput(currentState);
        if(explorationRate > random) {
            for(NeuralNet NNet : NetList) {
                int id = NNet.getNetID();
                double output = NNet.outputFor(input);
                double QValue = LUTNeuralNet.inverseOutput(output, maxQ[id],
minQ[id], argA, argB);
                int nextId=NNet.getNetID();
                currentAction[nextId]=output;
                currentQValue[nextId]=QValue;
            }
            action = getMaxIndex(currentQValue);
        }else {
        Random rand = new Random();
        action = rand.nextInt(LUT.actionsNum);

        }
        return action;
    }

    public void QLearning_NN( int action, double reward) {
        double currentStateQValue = currentQValue[action] ;
        double [] newInput = new double[inputsNum];
        newInput = LUTNeuralNet.normalizeInput(newState);
        for(NeuralNet NNet: NetList) {
            int act = NNet.getNetID();
            double output = NNet.outputFor(newInput);
            double QValue = LUTNeuralNet.inverseOutput(output, maxQ[act],
minQ[act], argA, argB);
            newAction[NNet.getNetID()]=output;
            newQValue[NNet.getNetID()]=QValue;
        }

        int max = getMaxIndex(newQValue);
        double maxNewQValue = newQValue[max];
        double expectQValue = currentStateQValue + LearningRate*(reward +
discountRate *maxNewQValue -currentStateQValue);
        double  expectOutput;
        expectOutput = LUTNeuralNet.normalizeExpectOutput(expectQValue, 
maxQ[action], minQ[action], argA, argB);
        NeuralNet net = NetList.get(action);
        double [] currentInput= LUTNeuralNet.normalizeInput(currentState);
        net.train(currentInput, expectOutput);
        double output2 = net.outputFor(currentInput);
        double QValue2 = LUTNeuralNet.inverseOutput(output2, maxQ[action], 
minQ[action], argA, argB);
        error=Math.abs(currentStateQValue - QValue2);
    }

    public void initializeNetList(){
        for(int i = 0; i < LUT.actionsNum; i++) {
            NeuralNet NNet = new 
NeuralNet(inputsNum,hiddenNum,outputsNum,learningRate_NN,momentumTerm,argA,arg
B,"bipolar",i);
            NetList.add(NNet);
        }
    }

    public int getMaxIndex(double [] argValues) {
        double maxQValue = argValues[0];
        int maxIndex = 0;
        for(int i = 0; i < argValues.length; i++) {
            if(maxQValue < argValues[i]) {
                maxQValue = argValues[i];
                maxIndex = i;
            }
        }
        return maxIndex;
    }

    public double[] getColumn(double[][] array, int index){
        double[] column = new double[LUT.statesNum]; //
        for(int i=0; i<column.length; i++){
           column[i] = array[i][index];
        }
        return column;
    }
}
```

2. Learning.java

```java
package com.shenyue2;
import java.util.Random;

public class Learning {
    public static final double LearningRate = 0.1;
    public static final double discountRate = 0.9;
    public static double explorationRate = 0.0;
    private int previousState;
    private int previousAction;
    public LUT LUTable;

    public Learning(LUT LUTable) {
        this.LUTable = LUTable;
    }

    public void LUTlearning(int currentState, int currentAction, double reward,
boolean offPolicy) {
        double previousQ = LUTable.getQValue(previousState, previousAction);
        if (offPolicy) {
            double currentQ = (1 - LearningRate) * previousQ
                    + LearningRate * (reward + discountRate *
LUTable.getMaxQ(currentState));
            LUTable.setQValue(previousState, previousAction, currentQ);
        } else { // onPolicy
            double currentQ = (1 - LearningRate) * previousQ
                    + LearningRate * (reward + discountRate *
LUTable.getQValue(currentState, currentAction));
            LUTable.setQValue(previousState, previousAction, currentQ);
        }
        previousState = currentState;
        previousAction = currentAction;
    }

    public int selectAction(int state) {
        double random = Math.random();
        if (explorationRate > random) {
            Random ran = new Random();
            return ran.nextInt(((LUT.actionsNum - 1 - 0) + 1));
        } else { // Pure greedy
            return LUTable.getBestAction(state);
        }
    }
}
```

3. LUT.java

```
package com.shenyue2;
import java.util.Random;

public class Learning {
    public static final double LearningRate = 0.1;
    public static final double discountRate = 0.9;
    public static double explorationRate = 0.0;
    private int previousState;
    private int previousAction;
    public LUT LUTable;

    public Learning(LUT LUTable) {
        this.LUTable = LUTable;
    }

    public void LUTlearning(int currentState, int currentAction, double reward,
boolean offPolicy) {
        double previousQ = LUTable.getQValue(previousState, previousAction);
        if (offPolicy) {
            double currentQ = (1 - LearningRate) * previousQ
                    + LearningRate * (reward + discountRate *
LUTable.getMaxQ(currentState));
            LUTable.setQValue(previousState, previousAction, currentQ);
        } else { // onPolicy
            double currentQ = (1 - LearningRate) * previousQ
                    + LearningRate * (reward + discountRate *
LUTable.getQValue(currentState, currentAction));
            LUTable.setQValue(previousState, previousAction, currentQ);
        }
        previousState = currentState;
        previousAction = currentAction;
    }

    public int selectAction(int state) {
        double random = Math.random();
        if (explorationRate > random) {
            Random ran = new Random();
            return ran.nextInt(((LUT.actionsNum - 1 - 0) + 1));
        } else { // Pure greedy
            return LUTable.getBestAction(state);
        }
    }
}
```

4. LUTNeuralNet.java

```
package com.shenyue2;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
public class LUTNeuralNet {
    private static int netNum = 6; // Number of actions.
    private static int inputsNum = 5; //Kinds of states
    private static int hiddenNum = 14;
    private static int outputsNum = 1;
    private static double expectOutput[][];
    private static double learningRate_NN = 0.2;
    private static double momentumTerm = 0.9;
    private static double argA = -1.0;
    private static double argB = 1.0;
    private static double[] minQ = new double [LUT.actionsNum];
    private static double[] maxQ = new double [LUT.actionsNum];


    private static ArrayList<Double> epochError;
    private static ArrayList<NeuralNet> NetList;


    //Neuron testNeuron = new Neuron("test");

    public static void main(String[] args){
        LUT LUTable = new LUT();
        File file = new File("LUT.dat");
        LUTable.load(file);
        double inputArray[][] = new double [LUT.statesNum][inputsNum];
        double normExpectOutput[][] = new double [LUT.actionsNum][LUT.statesNum];
        expectOutput = LUTable.getLUTable();
        for(int i = 0; i<LUT.actionsNum;i++) {
            maxQ[i] = 5 + getMaxQ(getColumn(expectOutput,i));
            minQ[i] = -5 + getMinQ(getColumn(expectOutput,i));
        }
        for(int i = 0; i < LUT.statesNum; i++) {
            int[] state = LUT.getState(i);
            inputArray[i] = normalizeInput(state);
            for(int j = 0; j < LUT.actionsNum; j++) {
                normExpectOutput[j][i]
=normalizeExpectOutput(expectOutput[j][i],maxQ[i],minQ[i],argA,argB);
            }
        }
        NetList = new ArrayList<NeuralNet>();
        for(int i = 0; i < LUT.actionsNum; i++) {
            int average =
EpochAverage(i,inputArray,normExpectOutput,0.000005,10000,10);
            System.out.println(i+"The average of number of epoches to converge is:
"+average+"\n");
        }

        for(NeuralNet net : NetList) {
            try {
                    File weight = new File("weight.dat");
                    weight.createNewFile();
                    net.save(weight);
            }catch(IOException e) {
                System.out.println(e);
            }
        }


    }

    public static double [] normalizeInput(int [] states) {
        double [] normalizedStates = new double [5];
        for(int i = 0; i < 5; i++) {
            switch (i) {
            case 0:
                normalizedStates[0] = -1.0 +
((double)states[0])*2.0/((double)(LUT.headingNum-1));
                break;
            case 1:
                normalizedStates[1] = -1.0 +
((double)states[1])*2.0/((double)(LUT.OpponentDistanceNum-1));
                break;
            case 2:
                normalizedStates[2] = -1.0 +
((double)states[2])*2.0/((double)(LUT.OpponentBearingNum-1));
                break;
            case 3:
                normalizedStates[3] = -1.0 +
((double)states[3])*2.0/((double)(LUT.XPositionNum-1));
                break;
            case 4:
                normalizedStates[4] = -1.0 +
((double)states[4])*2.0/((double)(LUT.YPositionNum-1));
                break;
            default:
                System.out.println("Normalize Error!");
            }
        }
        return normalizedStates;
    }

    public static double normalizeExpectOutput(double expected, double max, double
min, double argA, double argB){
        double normalizedExpect = 0.0;
        if(expected > max) {
            expected = max;
        }else if(expected < min) {
            expected = min;
        }
        normalizedExpect = argA +(expected-min)*(argB-argA)/(max - min);
        return normalizedExpect;
    }

    public static double inverseOutput(double output, double maxQ, double minQ,
double argA, double argB) {
        double QValue = 0.0;
        if(output < -1.0) {
            output = -1.0;
        }else if(output > 1.0) {
            output = 1.0;
        }
        QValue = minQ + (output-argA)/(argB-argA)*(maxQ - minQ);
        return QValue;
    }

    public static int EpochAverage(int act,double[][] input, double[][]
output,double minError, int maxSteps, int trialsNum) {
        int epochNumber, failure,success;
        double average = 0;
        epochNumber = 0;
        failure = 0;
        success = 0;
        NeuralNet NNet = null;
        for(int i = 0; i < trialsNum; i++) {
            NNet = new
NeuralNet(inputsNum,hiddenNum,outputsNum,LearningRate_NN,momentumTerm,argA,arg
```

```java
B,"bipolar",i);
            tryConverge(NNet,input,output[i],maxSteps, minError);
            epochNumber = getErrorArray().size();
            if( epochNumber < maxSteps) {
                average = average +  epochNumber;
                success ++;
            }
            else {
                failure++;
            }
        }
        double convergeRate = 100*success/(success+failure);
        System.out.println("ConvergeRate: "+convergeRate);
        average = average/success;
        NetList.add(NNet);
        return (int)average;
    }
    public static void tryConverge(NeuralNet NNet, double[][] input, double []
output,int maxStep, double minError) {
        int i;
        double totalError = 1;
        double previousError = 1;
        double variation = 1;
        epochError = new ArrayList<>();
        for(i = 0; i < maxStep && variation > minError; i++) {
            previousError = totalError;
            totalError = 0.0;
            for(int j = 0; j < input.length; j++) {
                totalError += NNet.train(input[j],output[j]);
            }
            totalError = Math.sqrt(totalError/input.length);
            epochError.add(totalError);
            variation  = Math.abs(totalError - previousError);

        }
        System.out.println("Total Error: = " + totalError);
        System.out.println("Number of epoch: "+ i + "\n");
        if(i == maxStep) {
            System.out.println("Training Error!");
        }

    }

    public static ArrayList <Double> getErrorArray(){
        return epochError;
    }

    public static void setErrorArray(ArrayList<Double> errors) {
        epochError = errors;
    }
    public static double[] getColumn(double[][] array, int index){
        double[] column = new double[LUT.statesNum]; //
        for(int i=0; i<column.length; i++){
           column[i] = array[i][index];
        }
        return column;
    }
    public static double getMaxQ(double [] argValues) {
        double maxQValue = argValues[0];
        int maxIndex = 0;
        for(int i = 0; i < argValues.length; i++) {
            if(maxQValue < argValues[i]) {
                maxQValue = argValues[i];
                maxIndex = i;
            }
        }
        return maxQValue;
    }

    public static double getMinQ(double [] argValues) {
        double minQValue = argValues[0];
        int minIndex = 0;
        for(int i = 0; i < argValues.length; i++) {
            if(minQValue > argValues[i]) {
                minQValue = argValues[i];
                minIndex = i;
            }
        }
        return minQValue;
    }




}
```

5. syRobot.java

```java
package com.shenyue2;

import java.awt.*;
import java.awt.geom.*;
import java.io.IOException;
import java.io.PrintStream;

import robocode.*;

public class syRobot extends AdvancedRobot {
  // Action
  public static final int moveAhead = 0;
  public static final int moveBack = 1;
  public static final int turnLeft = 2;
  public static final int turnRight = 3;
  public static final int robotfire = 4;
  public static final double aheadDistance = 150.0;
  public static final double backDistance = 100.0;
  public static final double turnDegree = 20.0;
  public static final int actionsNum = 5;

  // Opponent
  public String opponentName;
  public double opponentSpeed;
  public double opponentBearing;
  public long opponentTime;
  public double opponentX;
  public double opponentY;
  public double opponentDistance;
  public double opponentHead;
  public double opponentChangehead;
  public double opponentEnergy;

  public Learning learning;
  private double firePower;
  public static int count;
  public static int winCount;
  public static double reward;
  public static double winningRates;

  public static boolean intermediate = true; // false for only terminal reward
  public static boolean offPolicy = true; // false for on-Policy
public double goodReward = 5;
public double badReward = -5;

public LearningNN learningNN;
public static boolean isOnline = true;


public void run() {
    if(isOnline) {
        learningNN = new LearningNN(new LUT());
        learningNN.initializeNetList();
        loadTable();
        opponentDistance = 1000;
          setColors(Color.white, Color.pink, Color.pink);
          setAdjustGunForRobotTurn(true);
          setAdjustRadarForGunTurn(true);
          turnRadarRightRadians(2 * Math.PI);
         while (true) {
                int state = getState();//Get Initial State
                 int action = learningNN.selectAction_NN();
                  learning.LUTlearning(state, action, reward, offPolicy);
                  reward = 0.0;
                  switch (action) {
                  case moveAhead:
                    setAhead(aheadDistance);
                    break;
                  case moveBack:
                    setBack(backDistance);
                    break;
                  case turnLeft:
                    setTurnLeft(turnDegree);
                    break;
                  case turnRight:
                    setTurnRight(turnDegree);
                    break;
                  case robotfire:
                    fire(1);
                    break;
                  }
                  radarAction();
                  gunAction(2);
                  execute();
    }
    }else {
```

```java
  learning = new Learning(new LUT());
  loadTable();
  opponentDistance = 1000; // Set Opponent distance to 'far'
  // Set my syRobot
  setColors(Color.white, Color.pink, Color.pink);
  setAdjustGunForRobotTurn(true);
  setAdjustRadarForGunTurn(true);
  turnRadarRightRadians(2 * Math.PI);

  while (true) {
    int state = getState();
    int action = learning.selectAction(state);
    learning.LUTlearning(state, action, reward, offPolicy);
    reward = 0.0;
    switch (action) {
    case moveAhead:
      setAhead(aheadDistance);
      break;
    case moveBack:
      setBack(backDistance);
      break;
    case turnLeft:
      setTurnLeft(turnDegree);
      break;
    case turnRight:
      setTurnRight(turnDegree);
      break;
    case robotfire:
      fire(1);
      break;
    }
    radarAction();
    gunAction(2);
    execute();
  }
    }

}

private int getState() {
  int heading = LUT.getHeading(getHeading());
  int oppoDistance = LUT.getOpponentDistance(opponentDistance);
  int oppoBearing = LUT.getOpponentBearing(opponentBearing);
  int x = LUT.getXPosition(getX());
```

```
    int y = LUT.getYPosition(getY());
    int state = LUT.States[heading][oppoDistance][oppoBearing][x][y];
    return state;
  }

  private void radarAction() {
    double radarRotate;
    if (getTime() - opponentTime > 4) {
      radarRotate = 4 * Math.PI; // Rotate radar to find an opponent
    } else {
      radarRotate = getRadarHeadingRadians() - (Math.PI / 2 - Math.atan2(opponentY
- getY(), opponentX - getX()));
      radarRotate = nomslizeBearing(radarRotate);
      if (radarRotate < 0)
        radarRotate -= Math.PI / 10;
      else
        radarRotate += Math.PI / 10;
    }
    setTurnRadarLeftRadians(radarRotate);
  }

  private void gunAction(double power) {
    long currentTime;
    long nextTime;
    Point2D.Double opponentPosition = new Point2D.Double(opponentX, opponentY);
    // Distance between my robot and opponent
    double distance = Math.sqrt((opponentPosition.x - getX()) *
(opponentPosition.x - getX())
        + (opponentPosition.y - getY()) * (opponentPosition.y - getY()));
    for (int i = 0; i < 20; i++) {
      // Calculate time to reach the opponent
      nextTime = (int) Math.round(distance / (20 - 3 * firePower));
      currentTime = getTime() + nextTime;
      opponentPosition = guessPosition(currentTime);
    }
    // Set off the gun
    double gunOffSet = getGunHeadingRadians()
        - (Math.PI / 2 - Math.atan2(opponentPosition.y - getY(), opponentPosition.x
- getX()));
    setTurnGunLeftRadians(nomslizeBearing(gunOffSet));
    if (getGunHeat() == 0) {
      setFire(power);
    }
  }
```

```java
  public void onScannedRobot(ScannedRobotEvent e) {
    if ((e.getDistance() < opponentDistance) || (opponentName == e.getName())) {
      double absBearing = (getHeadingRadians() + e.getBearingRadians()) % (2 *
Math.PI);
      opponentName = e.getName();
      double head = nomslizeBearing(e.getHeadingRadians() - opponentHead);
      head = head / (getTime() - opponentTime);
      opponentChangehead = head;
      opponentX = getX() + Math.sin(absBearing) * e.getDistance();
      opponentY = getY() + Math.cos(absBearing) * e.getDistance();
      opponentBearing = e.getBearingRadians();
      opponentHead = e.getHeadingRadians();
      opponentTime = getTime();
      opponentSpeed = e.getVelocity();
      opponentDistance = e.getDistance();
      opponentEnergy = e.getEnergy();
    }
  }

  public Point2D.Double guessPosition(long time) {
    double newX, newY;
    if (Math.abs(opponentChangehead) > 0.00001) {
      double radius = opponentSpeed / opponentChangehead;
      double totalHead = (time - opponentTime) * opponentChangehead;
      newX = opponentX + (Math.cos(opponentHead) * radius) - (Math.cos(opponentHead
+ totalHead) * radius);
      newY = opponentY + (Math.sin(opponentHead + totalHead) * radius) -
(Math.sin(opponentHead) * radius);
    } else {
      newX = opponentX + Math.sin(opponentHead) * opponentSpeed * (time -
opponentTime);
      newY = opponentY + Math.cos(opponentHead) * opponentSpeed * (time -
opponentTime);
    }
    return new Point2D.Double(newX, newY);
  }

  double nomslizeBearing(double argValue) {
    if (argValue > Math.PI)
      argValue -= 2 * Math.PI;
    if (argValue < -Math.PI)
      argValue += 2 * Math.PI;
    return argValue;
  }

  double nomalizeHeading(double argValue) {
    if (argValue > 2 * Math.PI)
      argValue -= 2 * Math.PI;
    if (argValue < 0)
      argValue += 2 * Math.PI;
    return argValue;
  }

  public void onBulletHit(BulletHitEvent e) {
    if (intermediate) {
      if (opponentName == e.getName()) {
        reward +=9 * e.getBullet().getPower() ;
      }
    }
  }

  public void onBulletMissed(BulletMissedEvent e) {
    if (intermediate) {
      reward -= e.getBullet().getPower();
    }

  }

  public void onHitByBullet(HitByBulletEvent e) {
    if (intermediate) {
      if (opponentName == e.getName()) {
        reward -= 5 * e.getBullet().getPower();
      }
    }
  }

  public void onHitWall(HitWallEvent e) {
    if (intermediate) {
      reward -= (Math.abs(getVelocity()) * 0.5 - 1);
    }
  }

  public void onRobotDeath(RobotDeathEvent e) {
    if (e.getName() == opponentName)
      opponentDistance = 1000;
  }
```

```
  // Terminal rewards
  public void onWin(WinEvent event) {
    saveTable();
    reward += goodReward;
    count += 1;
    winCount += 1;
    PrintStream file = null;
    try {
      file = new PrintStream(new
RobocodeFileOutputStream(getDataFile("winning-rates.dat").getAbsolutePath(),
true));
      if (count == 19) {
        winningRates = (double) (winCount) / 20;
        file.println(winningRates);
        reward = 0;
        winCount = 0;
        count = 0;
        if (file.checkError())
          System.out.println("Save Error!");
        file.close();
      }
    } catch (IOException e) {
      System.out.println(e);
    } finally {
      try {
        if (file != null)
          file.close();
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }

  public void onDeath(DeathEvent event) {
    saveTable();
    reward += badReward;
    count += 1;
    PrintStream file = null;
    try {
      file = new PrintStream(new
RobocodeFileOutputStream(getDataFile("winning-rates.dat").getAbsolutePath(),
true));
      if (count == 19) {
        winningRates = (double) (winCount) / 20;
        file.println(winningRates);
        reward = 0;
        winCount = 0;
        count = 0;
        if (file.checkError())
          System.out.println("Save Error!");
        file.close();
      }
    } catch (IOException e) {
      System.out.println(e);
    } finally {
      try {
        if (file != null)
          file.close();
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }

  public void loadTable() {
    try {
      learning.LUTable.load(getDataFile("LUT.dat"));
    } catch (Exception e) {
      out.println("Load Error!" + e);
    }
  }

  public void saveTable() {
    try {
      learning.LUTable.save(getDataFile("LUT.dat"));
    } catch (Exception e) {
      out.println("Save Error!" + e);
    }
  }
}
```

6. NeuralNet.java

```
package com.shenyue2;
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.util.Random;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.PrintStream;

public class NeuralNet implements NeuralNetInterface {


    static final int MAX_INPUTS_NUM = 20;
    static final int MAX_HIDDEN_NUM = 20;
    static final int MAX_OUTPUTS_NUM = 20;

    private int inputsNum;
    private int hiddenNum;
    private int outputsNum;
    private double learningRate;
    private double momentumTerm;
    private double argA;
    private double argB;
    private String activeType;
    private int netID;
    // Weights from input layers to hidden layers
    private double[][] weight1 = new double[MAX_INPUTS_NUM][MAX_HIDDEN_NUM];
    // Weights form hidden layers to output layers
    private double[][] weight2 = new double[MAX_HIDDEN_NUM][MAX_OUTPUTS_NUM];
    // Weights difference between updated weights and previous one.
    private double[][] weightChange1 = new
double[MAX_INPUTS_NUM][MAX_HIDDEN_NUM];
    private double[][] weightChange2 = new
double[MAX_HIDDEN_NUM][MAX_OUTPUTS_NUM];

    private double[] inputsNeuron = new double[MAX_INPUTS_NUM ];
    private double[] hiddenNeuron = new double[MAX_HIDDEN_NUM];
    private double[] outputsNeuron = new double[MAX_OUTPUTS_NUM];
    // Value of delta in the hidden layer
    private double[] deltaHidden = new double[MAX_HIDDEN_NUM];
    // Value of delta in the output layer
    private double[] deltaOutput = new double[MAX_OUTPUTS_NUM];
```

```java
private double error = 0;

public NeuralNet(int inputsNum, int hiddenNum, int outputsNum,
            double learningRate, double momentumTerm,
               double argA, double argB, String activeType, int id) {
    this.inputsNum = inputsNum;
    this.hiddenNum = hiddenNum;
    this.outputsNum = outputsNum;
    this.learningRate = learningRate;
    this.momentumTerm = momentumTerm;
    this.argA = argA;
    this.argB = argB;
    this.activeType = activeType;
    this.netID = id;
}



@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;
}



@Override
public void initializeWeights() {
    for (int i = 0; i < inputsNum + 1; i++) {
        // The last index represents the bias of input
        for (int h = 0; h < hiddenNum; h++) {
            weight1[i][h] = getRandomWeight(-0.5, 0.5);
            weightChange1[i][h] = 0.0;
        }
    }

    for (int h = 0; h < hiddenNum + 1; h++) {
        for (int j = 0; j < outputsNum; j++) {
                weight2[h][j] = getRandomWeight(-0.5, 0.5);
                weightChange2[h][j] = 0.0;
            }
        }
    }


    private double getRandomWeight(double minWeight, double maxWeight) {
        double random = new Random().nextDouble();
        return minWeight + (random * (maxWeight - minWeight));
    }


    @Override
    public void zeroWeights() {
        for (int i = 0; i < inputsNum + 1; i++) {
            for (int h = 0; h < hiddenNum; h++) {
                weight1[i][h] = 0.0;
            }
        }
        for (int h = 0; h < hiddenNum+1; h++) {
            for (int j = 0; j < outputsNum; j++) {
                weight2[h][j] = 0.0;
            }
        }
    }
    public int getNetID(){
         return this.netID;
    }


    /*Construct neural net and do feed forward process, calculating the final
output*/
    @Override
    public double outputFor(double[] X) {
        //Firstly, given the input X[], set up the neural net
        for(int i = 0;i < inputsNum; i++){
            inputsNeuron[i] = X[i];
        }
        //Add bias
        inputsNeuron[inputsNum] = 1;
        hiddenNeuron[hiddenNum] = 1;

        //Compute hidden layer
        for(int h = 0; h < hiddenNum; h++){
        hiddenNeuron[h] = 0;
            for(int i = 0;i < inputsNum + 1; i++){
                hiddenNeuron[h] += weight1[i][h] * inputsNeuron[i];
            }
            hiddenNeuron[h] = customSigmoid(hiddenNeuron[h]);
        }

        //Compute output layer
        for(int j = 0; j < outputsNum; j++){
        outputsNeuron[j] = 0;
            for(int h = 0;h < hiddenNum + 1; h++){
                outputsNeuron[j] += weight2[h][j] * hiddenNeuron[h];
            }
            outputsNeuron[j] = customSigmoid(outputsNeuron[j]);
        }
        return outputsNeuron[0];  //Single output
    }



    /*Backward process, computing the delta for each layer and then update weights*/
    private void updateWeight(double argValue){
        //Compute deltaOutput[] for output layer
        for(int j = 0; j < outputsNum; j++){
            if(activeType.equals("binary"))
            deltaOutput[j] = (argValue - outputsNeuron[j]) * (1 - outputsNeuron[j])
* outputsNeuron[j];
            else if(activeType.equals("bipolar")) {
             deltaOutput[j] = (argValue - outputsNeuron[j]) * 0.5 * (1 -
outputsNeuron[j]) * (1 + outputsNeuron[j]);
            }
        }
        //Update weights from output layer to hidden layer
        for(int j = 0; j < outputsNum; j++){
            for(int h = 0; h < hiddenNum + 1; h++){
                weight2[h][j] += momentumTerm * weightChange2[h][j] + learningRate
* deltaOutput[j] * hiddenNeuron[h];
                weightChange2[h][j] = momentumTerm * weightChange2[h][j] +
learningRate * deltaOutput[j] * hiddenNeuron[h];
            }
        }

        //Compute deltaHidden[] for hidden layer
        for(int h = 0; h < hiddenNum; h++){
            for(int j = 0; j < outputsNum; j++){
                deltaHidden[h] += deltaOutput[j] * weight2[h][j];
            }
            if(activeType.equals("binary"))
            deltaHidden[h] = (1 - hiddenNeuron[h]) * hiddenNeuron[h] *
deltaHidden[h];
            else if(activeType.equals("bipolar")) {
            deltaHidden[h] = 0.5 * (1 - hiddenNeuron[h]) * (1 + hiddenNeuron[h])
* deltaHidden[h];
            }
        }
        //Update weights from hidden layer to input layer
        for(int h = 0; h < hiddenNum; h++){
            for(int i = 0; i < inputsNum + 1; i++){
                weight1[i][h] += momentumTerm * weightChange1[i][h] + learningRate
* deltaHidden[h] * inputsNeuron[i];
                weightChange1[i][h] = momentumTerm * weightChange1[i][h] +
learningRate * deltaHidden[h] * inputsNeuron[i];
            }
        }
    }



    /*Train the network and return the error of each single neuron*/
    @Override
    public double train(double[] X, double argValue) {
    double output;
    try {
        output = outputFor(X);
        error = 0.5 * (argValue - output) * (argValue - output);
        updateWeight(argValue);
    }catch(Exception e) {
        System.out.println(e);
    }
        return error;
    }



   /*Save weights of a neural net*/
    @Override
    public void save(File argFile) {
        PrintStream saveWeight = null;
        try {
        saveWeight = new PrintStream(new FileOutputStream(argFile));
        }catch(Exception e) {
        System.out.println(e);
        }
        for(int h = 0; h < hiddenNum; h++){
        for(int i = 0;i < inputsNum + 1; i++) {
            saveWeight.println(weight1[i][h]);
        }
        }
        for(int j = 0; j < outputsNum; j ++) {
        for(int h = 0; h < hiddenNum+1; h++) {
            saveWeight.println(weight2[h][j]);
        }
        }
        saveWeight.close();
     }


    public void load(File argFile) throws IOException{}
    /*Load weights of a neural net from a file*/
    public void load(String argFileName) throws IOException {
     FileInputStream weightFile = new FileInputStream(argFileName);
         BufferedReader weightReader = new BufferedReader(new
InputStreamReader(weightFile));
         for(int h = 0; h < hiddenNum; h++){
             for(int i = 0; i < inputsNum + 1; i++) {
                 weight1[i][h] = Double.valueOf(weightReader.readLine());
             }
         }
         for(int j = 0; j < outputsNum; j++) {
             for(int h = 0; h< hiddenNum + 1; h++) {
                 weight2[h][j] = Double.valueOf(weightReader.readLine());
             }
         }
         weightReader.close();
    }
}
```