Part 3 – Reinforcement Learning with Back-propagation

Yuwei Luo 14152342

I. Introduction

In the third and last part you are to replace the LUT used by the Reinforcement Learning component with a neural net. This will be easy to do in your software if your class headers for the LUT and the neural net match. However this is a little tricky to get working since there are many parameters that can be adjusted. Furthermore, the training data is dynamic as it is generated by the Robocode environment. I.e. it is not an a-priori static set of training data. This means that it is not possible to compute a "total error" during training, which also means that there is no conventional way to judge if learning is happening or not.

II. Questions part 1

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 options for the architecture of your neural network. Describe and draw all three options and state which you selected and why.

The first model is the Neural Net for Single Q Output Model. This Model's inputs includes the states as well as the actions, the neural net just calculate one action Q value each time the Model architecture is showing in Fig. 1

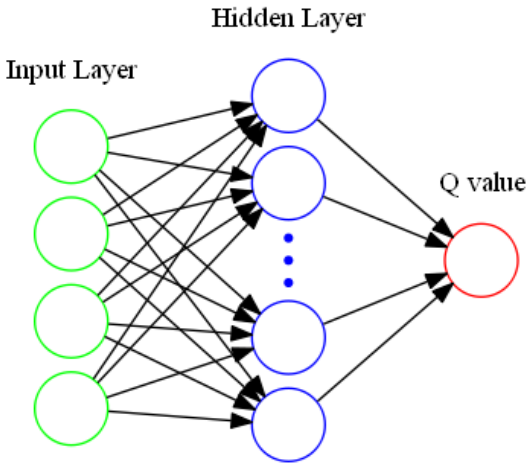

Fig. 1. Neural Net for Single Output Model

The second model is the Neural Net with Q output for each Action Model. Unlike Neural Net for Single Q Model, this model's inputs exclude actions. However, the output receive Q Values for all possible actions at-once. The model architecture is showing in Fig. 2.

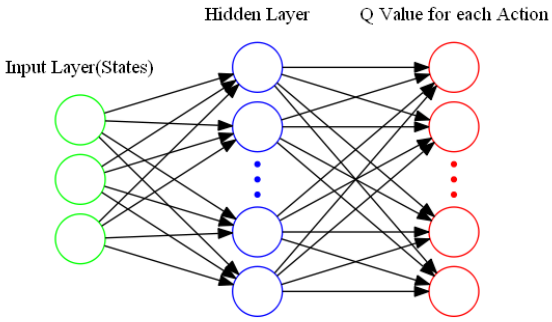

Fig. 2. Neural Net with Q output for each Action

The third model combine the first model and second model. This model separates nets each with a single output for each action. The inputs of this model just includes states. And the output for this model just includes the one specific action. However this model also includes several(number of actions) parallel neural net to calculate the each actions Q value. The model architecture is showing in Fig. 3

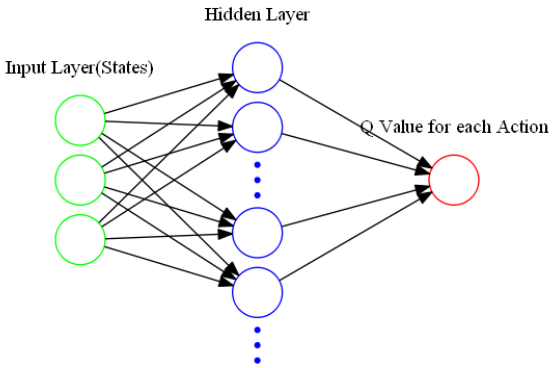

Fig. 3. Separate Nets each with a single output for each action.

I choose the third option, I tried to implement three different neural network models. Then I found that using the third neural network model can have a better result. Since for every action, it uses the isolated neural networks which are separated and will not share network weights and the output. Therefore it would be more accurate in terms of the approximate of the Q-learning value function. This modal also reduce the correlation between consecutive samples. This will increase the learning speed and reduce the RMS.

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.

By using the Look up table contents in part 2, I use the data in the look up table as the training set. I also normalize the outputs as well as inputs range between -1 to 1 of the neural network and set bipolar as the activation function before the training. Then the neural network calculate the RMS by using the equation: $RMS = \sqrt{\frac{1}{2}\sum_p (y^p - C^p)^2}$

The process for choosing hyper-parameters are showing below:

Firstly, I try to choose the momentum value. I tested 4 different momentum value and observed the RMS value for each momentum after 10000 epoch. The results are showing in table below:

TABLE I
RMS VALUE FOR DIFFERENT MOMENTUM AFTER 10000 EPOCH

Momentum	0.1	0.3	0.6	0.9
RMS	0.65	0.68	0.55	0.43

Based on the results above, the momentum and RMS value inversely related. With momentum increasing, the RMS value decrease. Therefore, I set momentum value to 0.9.

Secondly, I try to select the learning rate. I also tested 4 different learning rate with 0.9 momentum and observed the RMS value for each learning rate after 10000 epoch. The results are showing below:

TABLE II
RMS VALUE FOR DIFFERENT LEARNING RATE AFTER 10000 EPOCH

Learning rate	0.1	0.3	0.6	0.9
RMS	0.45	0.49	0.47	0.49

Based on the results above, I set the neural network learning rate as 0.1, since this learning rate lead to the lowest RMS value.

Finally, I try to decide number hidden layer neurons I tested 4 different number of neurons with 0.9 momentum and 0.1 learning rate. Then I observed the RMS value for each number of neurons. The results are showing below:

TABLE III
RMS VALUE FOR DIFFERENT NUMBER OF NEURONS AFTER 10000 EPOCH

Number neurons	5	10	15	20
RMS	0.5	0.42	0.45	0.47

Based on the results, I set the number of neurons to 10, since the 10 neurons is the balance point which lead the lowest RMS value. Continuing increase the number of neurons the RMS value increase, since large amount of neurons may cause the overfitting problem.

In a nutshell, the neural network hyper-parameters are setting as follow: Momentum = 0.9, learning rate = 0.1, number of neurons = 10. The Fig. 4 below show the neural network training result by using those hyper-parameters above.

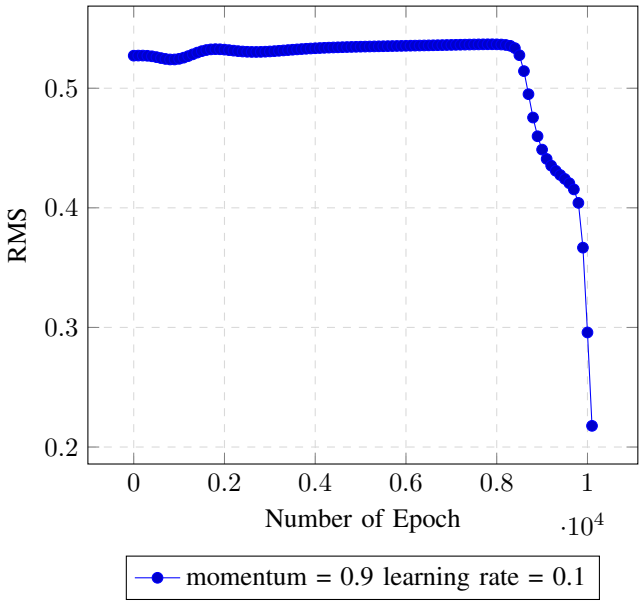

Fig. 4. The Training Result of NN

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.

In the assignment 2, we use the look up table to store all the state-action pairs Q value information. Besides the look up table of the assignment 2 stores the discrete Q value, since we quantize the the inputs states. In assignment 2, I defined 4 states and 8 actions, so there will have $3 \times 3 \times 2 \times 2 \times 8 = 288$ discrete state-action pairs corresponds Q value in the look up table. For improving the robot performance, we need add more state-action pairs for the robot. When the state-action pairs become large, it will cost large amount of memory to store Q value information. For example, If all the state-action pairs are to be stored in the look up table for assignment 2, it will take $100 \times 1000 \times 2 \times 2 \times = 3200000$ memory spaces to store those Q value and this only consider the integer states, let alone continuous states. With such huge input states, large amount of memory resources are required and learning speed will be greatly decreased when running the code with save and load the data.

However the neural network does not face memory problem, since we can use original value directly as the inputs. Besides the neural network does not need load and store Q value from the look up table. The neural network only need to store the different weights between each layers which require obviously

less memory than the look up table.

III. Question part 2

(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 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.

For measuring the performance of my robot with online training, I set the Spin bot as my enemy robot. I choose two metrics to evaluate the performance of my robot. The first is the robot winning rate and the another one is the error of q value between the current state and previous state.

Winning rate metrics:

I trained the neural network for 20000 rounds battles (exploration rate = 0.3) and rest 10000 battles (exploration rate = 0) to show The results of the training. Calculating the wining rate for every 1000 rounds battles. The graph are showing in Fig.5. Based on the results during the learning process the winning rate is low. However after the training, the rest of battles winning rates are dramatically increase. Therefore, the robot did learn during the learning process.

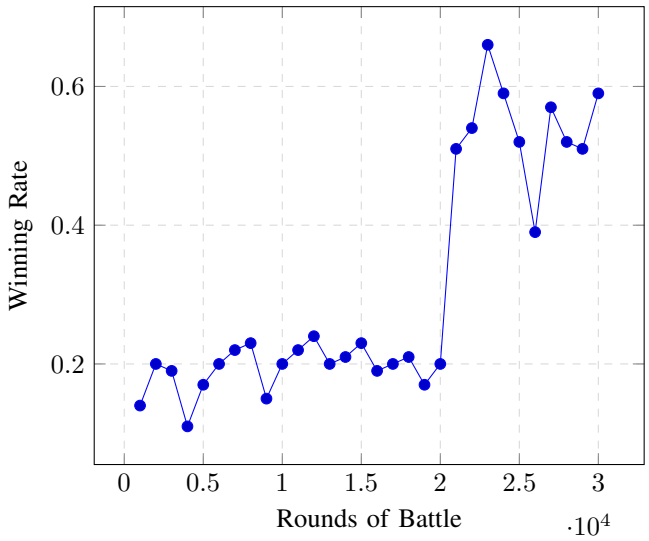

Fig. 5. The NN robot Winning rate

Error of Q value metrics: I trained the neural network for 20000 rounds battles (exploration rate = 0.3) and rest 10000 battles (exploration rate = 0) to show The results of the training. Calculating the $error = Q(S_{t+1}) - Q(S_t)$ The graph are showing in Fig.6. Based of the result of the error of the Q value, the trend of the error is decreasing. Therefore the robot did learn during the online training.

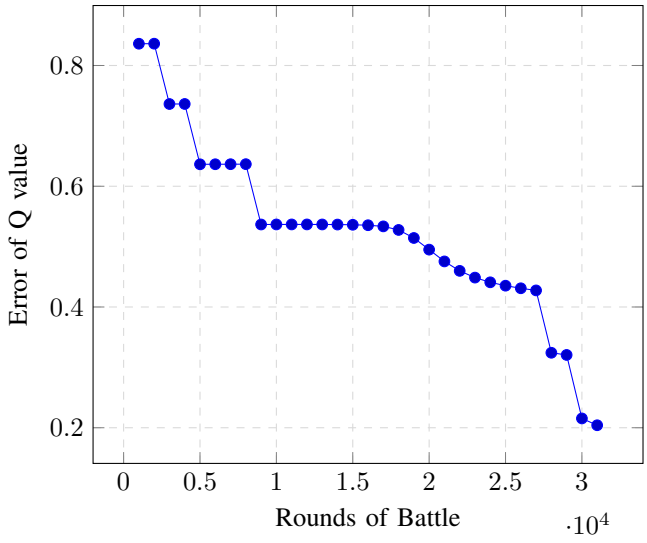

Fig. 6. The NN robot Q value error

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?

I expect the higher γ value is better. Since discount factor is applied to the total accumulated reward represented by $V(S_{t+1})$. It is a number in the range 0 to 1, typically close to 1 and is used to basically weight future rewards. When γ is 0, only the immediate rewards are used in the determination of the next action/state as any future rewards are ignored. When γ is 1, future rewards are as significant as immediate ones. Therefore combine immediate rewards as well as future rewards will help the neural network learning speed. I tested 4 different γ value. Then I trained the neural network for 20000 rounds battles and rest 10000 battles to show The results of the training.The graph of different γ value result are showing in Fig. 7.

The results shows that with γ increasing, the wining rate after learning process also increase. Therefore, the higher γ value will help the neural network learning speed.

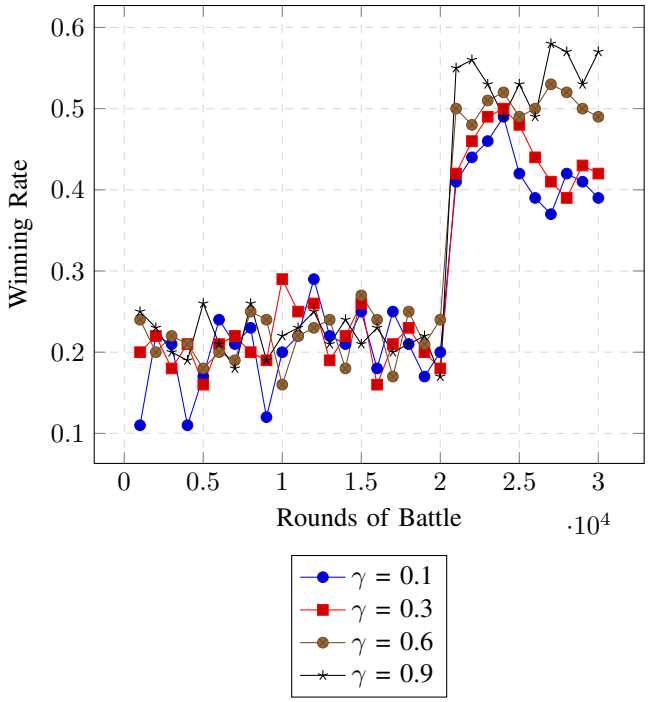

Fig. 7. The Winning rate with different γ

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.

The Explanation process are showing below:

$$V(S_t) = r_t + \gamma V(S_{t+1}) \tag{1}$$

The equation (1) shows the Bellman equation. $V(S_t)$ and $V(S_{t+1})$ represent the value function for the state at adjacent time periods. r_t is the immediate reward at time t. γ is the discount factor.

$$V^*(S_t) = r_t + \gamma V^*(S_{t+1}) \tag{2}$$

The equation (2) shows the optimal value Bellman equation. $V^*(S_t)$ and $V^*(S_{t+1})$ represent the optimal value function at adjacent time periods.

$$V^*(S_t) = e(S_t) + V^*(S_t) \tag{3}$$

$$V^*(S_{t+1}) = e(S_{t+1}) + V^*(S_{t+1}) \tag{4}$$

The equation (3) and (4) shows the error in the value function at any given state be represented by e_s and e_{t+1}.

$$e(S_t) = r_t + \gamma V(S_{t+1}) - V^*(S_t) \tag{5}$$

$$e(S_t) + V^*(S_t) = r_t + \gamma(e(S_{t+1}) + V^*(S_{t+1})) \tag{6}$$

$$e(S_t) + V^*(S_t) = \gamma e(S_{t+1}) + V^*(S_t) \tag{7}$$

The equation (5) (6) and (7) shows that using the error equation (3) (4) replace the $V(S_t)$ and $V(S_{t+1})$ in Bellman equation.

$$e(S_t) = \gamma e(S_{t+1}) \tag{8}$$

From the equation (8), concluding that the error in successive states is related. But we know that. already, since we know that the future rewards $V(s)$ are themselves are related. By implementing the Markov Chain, decision process where a reward is available at a terminal state, we know that the reward is known precisely. Assuming $e(S_t) = 0$ when it is trained for long enough iterations with Q-learning and it finally converges to the optimal value function.

$$e(S_t) + e_{error} = V(S_t) - V^*(S_t) \tag{9}$$

From the equation (9) shows that when implement the neural network model, the approximate of the value function may lead to some other additional errors. the e_{error} represents the additional errors cause by the approximate of the value function. Unlike look up table, the neural network model error may not decrease due to additional errors. Therefore, the value function of $V(S_T)$ may not finally converge to the optimal value function $V^*(S_t)$ by implementing the neural network model.

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.

Since there is no a-priori training set to work with, there still several ways to monitor learning performance of the the neural network, the method are showing below:

Firstly, since we implement the neural network model on the robocode, the most straightforward way is to observe the wining rate of the robot. Measuring the winning rate every 50 rounds of battles can observe the trend of the wining rate.if the neural network is learning, The wining rate is unstable at beginning learning process since the experience replay. After several rounds the wining rate trend will keep increase until converge, and stay at the stable and high level of wining rate.

Secondly, as I mentioned above since we implement the neural network model on the robocode, observing the robot survive time can also indicate whether the neural network is learning. If the neural network is learning, the survive time at beginning learning process is short since the robot Will be easily defeated by other robots. After several rounds the survive become longer. When it converge, the survive time become shorter since At this time it can easily defeat other robots. The survive time trend will increase at beginning, after reaching some break-even point, the survive time will reduce until it converge.

Thirdly, Since we implement the memory replay for the neural network model. Memory replay list can also indicate whether it is learning. At beginning learning process the state-action pairs are random. After several rounds of the learning, the list of memory replay will become more and more fixed until it converge, since the neural net is learning and it will pick the most state-action pairs to defeat the enemy robot.

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.

The experience replay stores the agent's experiences at each time step in a data set and represents the agent's experience at time t as e_t. At time t the agent's experience e_t is defined as this tuple: $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. Each tuple includes the current state s_t, the action take for current state a_t, the rewards at time t+1 r_{t+1} given to the agent at time (s_t, a_t) as a result of the previous state-action pair, And the next state s_{t+1}.

The main purpose for the experience replay is to break the correlation between consecutive samples. If the network learned only from consecutive samples of experience as they occurred sequentially in the environment, the samples would be highly correlated and would therefore lead to inefficient learning. Taking random samples from replay memory breaks this correlation.

I tested four different n and compared the effects of different n on the learning process and results.I trained the neural network for 20000 rounds battles and rest 10000 battles to show The results of the training.The graph are showing in Fig. 8.(on the right)

Based on the results with the size of N increasing, the wining rate is little bit increasing after the learning process since we increase the efficient of the learning and break the correlation between consecutive samples. However, the trade-off is increasing the time of learning process.

IV. Conclusion

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?

Practical Issues:

1. During the training process of the neural network, the selection of the hyper-parameters including learning rate, momentum factor, the structure of neural networks should be carefully considered. Some combination of those factors may lead to some bad training result such as increase of the learning time to converge and even not converge.

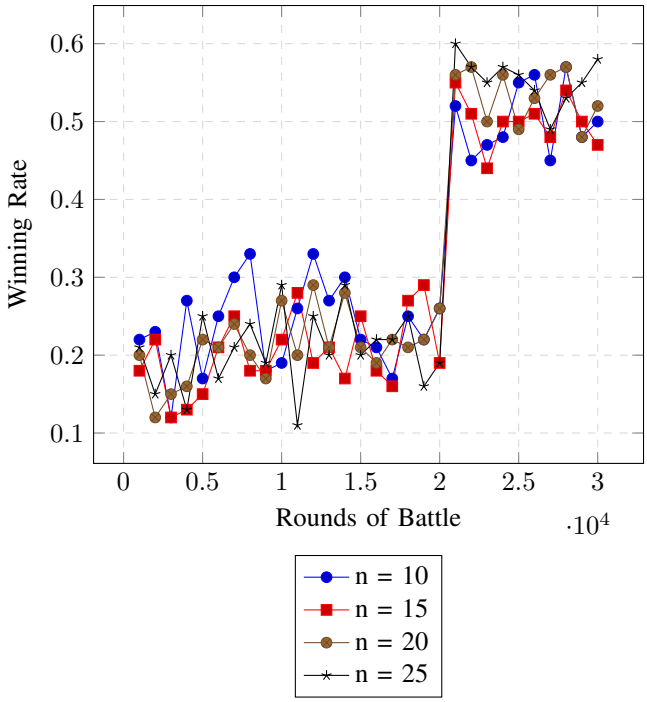

Fig. 8. The Memory replay with different N size Results

2.During the training process of neural network, the selection of replay memory size should also be carefully considered. The small size of the replay memory may lead to inefficient learning and cannot eliminate the correlation between two states. Therefor, the result of learning might be biased. The large size of the replay memory may increase the time of learning process, since sample size is large. Therefore, find the balance point is essential and hard.

3.For improving the performance of the robot, the states and actions should be clearer purpose. So the neural net suppose to add more inputs due to the encoding format. This will lead to more complicated structure of neural net should be applied. More inputs means more hidden layer units and longer learning process. Sometimes the large amount of the hidden layer units even result in over-fitting problem.

Improve the performance:

1.Increasing appropriate states and actions diversity is one essential way to improve the performance of the robot. Small size of states and actions will confine learning process especially facing some strong enemy robots. Due to the small size of states and actions the neural net system just postpone the fail instead of winning. Large size of states and actions will increase the time of learning process as well as the structure of the neural network. And some states and actions cannot be chosen since the size is too large. Therefore find the

appropriate states and actions is a essential way to improve the performance.

2.The exploration rate also has the essential effect on the performance. Setting the exploration rate range between 0.7 to 0.9 during the learning process will help to improve the robot performance. The high exploration rate will force robot to try the different actions to react the enemy robot's attack and save the best action. On the other hand, the high exploration rate will also help the neural network to decrease the learning time. Therefore, appropriate exploration rate will help to improve the robot performance.

convergence problems:

1. Normalizing the inputs as well as outputs will help the neural network convergence. The inputs do not quantize in the assignment 3, therefore, the inputs may vary and has big difference. Some inputs value are very large such as distance between two robots. The results calculating by neural network may reach the threshold of the activation function which will hinder the neural network from learning. Without normalizing may lead to the neural network even can not convergence. The normalizing inputs and outputs will help the neural net system convergence.

2. Using the some other activation function instead of bipolar sigmoid function may get the better result. The other activation function may have larger range which could corresponding to more domain. In other wards, the difference of each state will be magnified and the saturation problem may be relieved. The activation function such as Hyperbolic tangent and Rectified linear units may even eliminate the saturation problem and get better performance for the neural network convergence.

3.Setting the appropriate hyper-parameters will also help to address the convergence problem. The step size parameter is close related to convergence. The small step size may drag the neural network system speed to find the global minimum point for error. The large step size may lead to the neural network system jump gap to large and can not find the global minimum point for error. Therefore, picking the correct and appropriate hyper-parameters are also essential.

advice for other practical application:

To do function approximation for other applications, one important thing is that the states and actions should be carefully selected considering that they should be independent and can best represent the actually state of the application. Another advice is that before doing the online training, do offline training just like training the neural net with the lookup table. It will greatly increase the speed of training and converging comparing with using random weights in the neural network. What is more, choosing the proper parameters of the network, using the proper structure and applying proper activation functions will help the reinforcement learning to get better result.

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.

When applying the neural network reinforcement learning technology to the closed-loop control system for automatically delivering anesthetic there are several concerns need to be considered. Some concerns are essential.

Concerns:

1, by applying the offline training of the neural network to the closed-loop control system need massive amount of surgery information which can make the neural network system work to be as precise as possible. However, the surgery information is very based on the different patient especially for the anesthetic step in the surgery . Therefore, the surgery information is rarer which will catalyze the training set problem.

2, by applying the online training of the neural network to the closed-loop control system may cause the serious medical accident. During the training process of the neural network system, the error is inevitable especially at beginning of the learning, although it will converge at the end. However, in in reality it is not allowed to make errors during the surgery. The minor mistake may lead to surgery fail and even affect the health of the patient.

3, in reality patient's health condition and the area and scope of anesthesia will make this problem more complicated. Therefore, the current neural network system needs to be updated in time to adapt new conditions. If this system cannot be updated in time, this may lead to bad irreversible consequences for the patient

variation:

The variation is the development of information collection tools in the future. By using this kind of tools, it will provide sufficient surgery information to the neural network system which will help the neural network system finish the offline training and be implemented in real world.

APPENDIX

```
public class Action
{
  public static final int RobotAhead = 0;
  public static final int RobotBack = 1;
  public static final int RobotAheadTurnLeft = 2;
  public static final int RobotAheadTurnRight= 3;
  public static final int RobotBackTurnLeft = 4;
  public static final int RobotBackTurnRight= 5;

  public static final int NumRobotActions = 6;

  public static final double RobotMoveDistance = 300.0;
  public static final double RobotTurnDegree =  30.0;
}

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

import robocode.DeathEvent;
import robocode.RobocodeFileOutputStream;
import robocode.*;

import robocode.AdvancedRobot;

public class BpNetWork implements NeuralNetInterface {

    int argNumInputs;
    int argNumHidden;
    double argLearningRate;
    double argMomentumTerm;
    double argA;
    double argB ;
    public double[][] layer;
    public double[][] layError;
    public double[][][] layerWeight;
    public double[][][] layerWDelta;

    //double inputValue[]=new double[argNumInputs+1];
    BpNetWork(){
        argNumInputs=5;
        argNumHidden=15;
```

```
    argLearningRate=0.1;
    argMomentumTerm=0.9;
    argA=0;
    argB=1;
    layer = new double[3][];
    layError = new double[3][];
    layerWeight = new double[2][][];
    layerWDelta = new double[2][][];
    /*for(int i=0;i<inputValue.length-1;i++){
        inputValue[i]=0;
    }
    inputValue[inputValue.length-1]=1;*/
}
BpNetWork(int[] layernum, double rate, double mobp,double A,double B)
{
    argNumInputs=layernum[0];
    argNumHidden=layernum[1];
    argA=A;
    argB=B;
        this.argMomentumTerm = mobp;
        this.argLearningRate = rate;
        layer = new double[layernum.length][];
        layError = new double[layernum.length][];
        layerWeight = new double[layernum.length-1][][];
        layerWDelta = new double[layernum.length-1][][];
}

 int NumInputWBias=argNumInputs+1;
 int NumHidWBias=argNumHidden+1;


 /**
     * 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){
     return 2/(1+Math.pow(Math.E, -x))-1;
 };


 public double customSigmoid(double x){
     return (argB-argA)/(1+Math.pow(Math.E, -x))+argA;
 };
```

```
    public void initializeWeights(){
        int[] layerNum=new int[3];
        layerNum[0]=this.argNumInputs;
        layerNum[1]=this.argNumHidden;
        layerNum[2]=1;
        for(int l=0;l<layerNum.length;l++){
              layer[l]=new double[layerNum[l]];
              layError[l]=new double[layerNum[l]];
              if(l+1<layerNum.length){
                  layerWeight[l]=new double[layerNum[l]+1][layerNum[l+1]];
                  layerWDelta[l]=new double[layerNum[l]+1][layerNum[l+1]];
                  for(int j=0;j<layerNum[l]+1;j++)
                      for(int i=0;i<layerNum[l+1];i++)
                          layerWeight[l][j][i]=Math.random()-0.5;
              }
        }
    }


    /**
    * Initialize the weights to 0.
    */

    public void zeroWeights(){
   int[] layerNum=new int[3];
    layerNum[0]=this.argNumInputs;
    layerNum[1]=this.argNumHidden;
    layerNum[2]=1;

    for(int l=0;l<layerNum.length;l++){
          layer[l]=new double[layerNum[l]];
          layError[l]=new double[layerNum[l]];
          if(l+1<layerNum.length){
              layerWeight[l]=new double[layerNum[l]+1][layerNum[l+1]];
              layerWDelta[l]=new double[layerNum[l]+1][layerNum[l+1]];
              for(int j=0;j<layerNum[l]+1;j++)
                  for(int i=0;i<layerNum[l+1];i++)
                      layerWeight[l][j][i]=0;
          }
      }
}
```

```
////commone interface part
/**
 * @param X The input vector. An array of doubles.
 * @return The value returned by th LUT or NN for this input vector
 */
 public double outputFor(double [] X){
      for(int l=1;l<layer.length;l++){
          for(int j=0;j<layer[l].length;j++){
              double z=layerWeight[l-1][layer[l-1].length][j];
              for(int i=0;i<layer[l-1].length;i++){
                  layer[l-1][i]=l==1?X[i]:layer[l-1][i];
                  z+=layerWeight[l-1][i][j]*layer[l-1][i];
              }
              layer[l][j]=this.customSigmoid(z);
          }
       }

      return layer[layer.length-1][0];
 }



* @return The error in the output for that input vector
*/
public double train(double [] X, double argValue){
double out=outputFor(X);
double[] val=new double[1];
val[0]=argValue;

 int l=layer.length-1;
// System.out.print(l);
 for(int j=0;j<layError[2].length;j++)

layError[2][j]=(val[j]-out)*1/(this.argB-this.argA)*(out-argA)*(argB-out);

/* l--;

 for(;l>=0;l--){
     for(int j=0;j<layError[l].length;j++){
         double z = 0.0;
         for(int i=0;i<layError[l+1].length;i++){
             z=z+l>0?layError[l+1][i]*layerWeight[l][j][i]:0;
```

```
                layerWDelta[l][j][i]=
this.argMomentumTerm*layerWDelta[l][j][i]+this.argLearningRate*layError[l+1
][i]*layer[l][j];//隐含层动量调整
                layerWeight[l][j][i]+=layerWDelta[l][j][i];
                if(j==layError[l].length-1){
                    layerWDelta[l][j+1][i]=
this.argMomentumTerm*layerWDelta[l][j+1][i]+this.argLearningRate*layError[l
+1][i];//截距动量调整
                    layerWeight[l][j+1][i]+=layerWDelta[l][j+1][i];
                }
            }

layError[l][j]=z*1/(this.argB-this.argA)*(layer[l][j]-argA)*(argB-layer[l][
j]);//记录误差
         }

     }*/

     //hid-output
     for(int j=0;j<this.argNumHidden;j++){
     double preW=layerWeight[1][j][0];

layerWeight[1][j][0]+=this.argMomentumTerm*layerWDelta[1][j][0]+this.argLea
rningRate*layError[2][0]*layer[1][j];
     layerWDelta[1][j][0]=layerWeight[1][j][0]-preW;
     }

layerWDelta[1][this.argNumHidden][0]=this.argMomentumTerm*layerWDelta[1][th
is.argNumHidden][0]+this.argLearningRate*layError[2][0];

layerWeight[1][this.argNumHidden][0]+=layerWDelta[1][this.argNumHidden][0];


     //input-hidden
     for(int j=0;j<this.argNumHidden;j++){

layError[1][j]=layError[2][0]*1/(this.argB-this.argA)*(layer[1][j]-argA)*(a
rgB-layer[1][j])*layerWeight[1][j][0];

     for(int i=0;i<this.argNumInputs;i++){

layerWDelta[0][i][j]=this.argMomentumTerm*layerWDelta[0][i][j]+this.argLear
ningRate*layError[1][j]*layer[0][i];
         layerWeight[0][i][j]+=layerWDelta[0][i][j];
```

```
      }

layerWDelta[0][this.argNumInputs][j]=this.argMomentumTerm*layerWDelta[0][th
is.argNumInputs][j]+this.argLearningRate*layError[1][j];

layerWeight[0][this.argNumInputs][j]+=layerWDelta[0][this.argNumInputs][j];

      }

      //System.out.print(layer_weight);
     return val[0]-out;
}


     public void save(File argFile) throws IOException {
            //File weightsfile = new File("weightsfile.lqn.txt");
            if(argFile.exists())
            {
                FileWriter erasor = new FileWriter(argFile);
                erasor.write(new String());
                erasor.close();
            }
            else
            {
                argFile.createNewFile();
            }

            int count=0;
            //for(int i = 0; i<this.layerWeight.length; i++)
            //{
                int i=0;
                ArrayList<double[]> weight_vector = new
ArrayList<double[]>(0);
                //System.out.println(this.layerWeight[i].length);

                for(int j=0;j<this.layerWeight[i].length;j++){
                    double[] buffer=new
double[this.layerWeight[i][j].length];
                    //System.out.println(this.layerWeight[i][j].length);
                    //System.out.println(this.layerWeight[i][j][0]);
                    for(int k=0;k<this.layerWeight[i][j].length;k++)
                        {
                        buffer[k]=layerWeight[i][j][k];
                        double[] bufferN=new double[1];
```

```
                    bufferN[0]=buffer[k];
                    weight_vector.add(count++, bufferN);
                    }


                }
                //if(this.writeWeights(weight_vector,argFile) == 0)
                    //System.out.println("Writing hidden nodes fails");

                i++;
                for(int j=0;j<this.layerWeight[i].length;j++){
                    double[] buffer=new double[1];
                    //System.out.println(this.layerWeight[i][j].length);
                    //System.out.println(this.layerWeight[i][j][0]);
                    buffer[0]=layerWeight[i][j][0];

                    weight_vector.add(count++, buffer);
                }
                if(this.writeWeights(weight_vector,argFile) == 0)
                    System.out.println("Writing output nodes fails");
            //}

        }

    public int writeWeights(ArrayList<double[]> content, File file) throws
IOException
        {
            BufferedWriter bw = new BufferedWriter (new FileWriter(file,true));
            for(int i=0; i<content.size(); i++)
            {
                for(int j = 0; j<content.get(i).length;j++)
                {
                    bw.write(content.get(i)[j]+"\t" );
                    bw.flush();
                }
                bw.newLine();
                bw.flush();
            }
            bw.write("=========================================");
            bw.newLine();
            bw.flush();
            bw.close();
            content.clear();
            System.out.println("Writing successed");
```

```
            return 1;
        }

        public void loadWeight ( String argFileName ) throws IOException {

            FileInputStream inputFile = new FileInputStream( argFileName );
            BufferedReader inputReader = new BufferedReader(new
InputStreamReader( inputFile ));


            // First load the weights from the input to hidden neurons (one line
per weight)
            //System.out.println(this.layerWeight[0].length+"\n");
            for ( int i=0; i<this.layerWeight[0].length; i++) {
                for(int j=0;j<this.layerWeight[0][i].length;j++)

    this.layerWeight[0][i][j]=Double.valueOf( inputReader.readLine() );

            }
            for(int i=0;i<this.layerWeight[1].length;i++){
                this.layerWeight[1][i][0] =
Double.valueOf( inputReader.readLine() );
            }
            inputReader.close();

        }



      public void load(String argFileName) throws IOException{
         FileInputStream fin;
         try{
             fin = new FileInputStream(argFileName);
         }catch(FileNotFoundException exc){
             System.out.println("file not found");
             return;
         }

         int[] layernum=new int[3];
         layernum[0]=this.argNumInputs;
         layernum[1]=this.argNumHidden;
         layernum[2]=1;

         for(int l=0;l<layernum.length;l++){
```

```
                layer[l]=new double[layernum[l]];
                layError[l]=new double[layernum[l]];
                if(l+1<layernum.length){
                    layerWeight[l]=new double[layernum[l]+1][layernum[l+1]];
                    layerWDelta[l]=new double[layernum[l]+1][layernum[l+1]];
                    for(int j=0;j<layernum[l]+1;j++)
                        for(int i=0;i<layernum[l+1];i++)
                            layerWeight[l][j][i]=fin.read();
                }
            }
            fin.close();
        }


}package rlRobotNew;



import java.io.File;
import java.io.FileNotFoundException;
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 th 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 an neural net to a file.
 * @param argFile of type File.
 * @throws FileNotFoundException
 */
 public void save(File argFile) throws IOException;

 /**
 * Loads the LUT or neural net weights from file. The load must of course
 * have knowledge of how the data was written out by the save method.
 * You should raise an error in the case that an attempt is being
 * made to load data into an LUT or neural net whose structure does not match
 * the data in the file. (e.g. wrong number of hidden neurons).
 * @throws IOException
 */
 public void load(String argFileName) throws IOException;
}

import java.util.Random;

public class Qlearning {
    private double alpha = 0.1;
    private double gamma = 0.9;
    public double explorationRate = 0;
    private int currentState;
    private int currentAction;
    private boolean isFirstState = true;
    private LUT lut;

    public Qlearning(LUT lut) {
        this.lut = lut;
    }
    //On policy method
    public void SARSLearn(int nextState, int nextAction, double reward) {
        double lastQVal;
        double newQVal;
        if(isFirstState) {
            isFirstState = false;
        }
        else {
            lastQVal = lut.getQvalue(currentState, currentAction);
```

```
            newQVal = lastQVal + alpha*(reward + gamma * lut.getQvalue(nextState,
nextAction) - lastQVal);
                lut.setQvalue(currentState, currentAction, newQVal);
            }

            currentState = nextState;
            currentAction = nextAction;
        }
        //Off policy method
        public void  offPolicy(int nextState, int nextAction, double reward) {
            double lastQ;
            double newQ;
            if(isFirstState) {
                isFirstState = false;
            }else {
                lastQ = lut.getQvalue(currentState, currentAction);
                newQ  = lastQ + alpha*(reward + gamma *
lut.getMaxQvalue(nextState)-lastQ);
                lut.setQvalue(currentState, currentAction, newQ);
            }
            currentState =  nextState;
            currentAction = nextAction;
}

        public int nextAction(int state) {
            double probability = Math.random();
            int action = 0;
            if(probability < explorationRate) {
                Random rand = new Random();
                action = rand.nextInt(Actions.numActions);
            }else {

                action = lut.getBestAction(state);
            }
            return action;

        }




}
package rlRobotNew;
```

```
public interface NeuralNetInterface extends CommonInterface{

 final double bias = 1.0; // The input for each neurons bias weight

/**
 * 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 arbB Integer upper bound of sigmoid used by the output neuron only.

 public abstract NeuralNet (
 int argNumInputs,
 int argNumHidden,
 double argLearningRate,
 double argMomentumTerm,
 double argA,
 double argB );
 */

 /**
 * 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
 */
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();


 } // End of public interface NeuralNetInterface

import java.awt.*;
import java.awt.geom.*;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Random;
import java.util.Vector;

import NNRobot.Action;
import robocode.*;

import robocode.AdvancedRobot;

    public class newRobot1 extends AdvancedRobot
    {
      public static final double PI = Math.PI;
      private Target target=new Target();
      private QTable table;
      private Learner learner;
      private double reward = 0.0;
      private double firePower;
      private int direction = 1;
      private int isHitWall = 0;
      private int isHitByBullet = 0;
      private boolean NNFlag=true;


        private static final double NN_alpha = 0.1; //learning rate for NN Q
learning
    private static final double NN_lambda = 0.9; //Discount rate for NN Q learning
```

```
    private static double NN_epsilon = 0.0;
    private int NN_last_action = 0;
    private double[] NN_last_states = new double[5];
    //e(s) statistics
        private static double total_error_in_one_round = 0.0;
        private static Vector<Double> error_statistics = new Vector<Double>(0);

        //Q-value learning history
        private static Vector<Double> Q_value_history = new Vector<Double>(0);
        private static final double test_heading = 100;
    private static final double test_target_distance = 100;
    private static final double test_target_bearing = 70;
    private static final int test_action = 0;
    private static final int test_hitWall = 0;
    private static final int test_hitByBullet = 0;
    private static double NN_test_heading = test_heading/180 -1;
    private static double NN_test_target_distance = test_target_distance/500
-1;
    private static double NN_test_target_bearing = test_target_bearing/180-1;
    private static final double NN_test_hitWall = 0;
    private static final double NN_test_hitByBullet = 0;
    private static int NN_test_action = test_action;

      double rewardForWin=100;
      double rewardForDeath=-10;
      double accumuReward=0.0;
      private static int count=0;
      private static int countForWin=0;

      BpNetWork[] myNet=new BpNetWork[Action.NumRobotActions];

      public void run()
      {
        if(NNFlag==false){
        table = new QTable();
        loadData();
        learner = new Learner(table);
        target = new Target();
        target.distance = 1000;
        }
        else
        {

        int[] layernum=new int[3];
```

```
            layernum[0]=5;
            layernum[1]=15;
            layernum[2]=1;
            double mobp=0.9;
            double rate=0.1;
            double[] maxminQ=new double[2];
            maxminQ[0]=1.0;
            maxminQ[1]=0.0;
        //System.out.println(Action.NumRobotActions);
        for(int i=0;i<Action.NumRobotActions;i++){
            myNet[i]=new
BpNetWork(layernum,rate,mobp,maxminQ[1],maxminQ[0]);
            myNet[i].initializeWeights();
            System.out.println("robotlayerweightis
"+myNet[i].layerWeight.length+"\t"+myNet[i].layerWeight[0].length+"\t"+myNe
t[i].layerWeight[0][0].length+"\t");
            try {

    myNet[i].loadWeight("C:/robocode/robots/rlRobotNew/newRobot1.data/NN_we
ights_from_LUT"+i+".txt");
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
        }
        System.out.println(myNet[0].layerWeight[1][15][0]);
        }


        setColors(Color.green, Color.white, Color.green);
        setAdjustGunForRobotTurn(true);
        setAdjustRadarForGunTurn(true);
        turnRadarRightRadians(2 * PI);
       /* if(getRoundNum()>500)
        {
          learner.explorationRate=0.3;
         }
        */
        while (true)
        {
          robotMovement();
          firePower = 3000/target.distance;
          if (firePower > 3)
            firePower = 3;
```

```
        radarMovement();
        gunMovement();
        if (getGunHeat() == 0) {
          setFire(firePower);
        }
        execute();

      }
    }


     private void robotMovement()
     {
    int action;
    if(NNFlag==false){
        int state = getState();
        action = learner.selectAction(state);

        //learner.learn(state, action, reward);
        learner.learnSARSA(state, action, reward);
        accumuReward+=reward;
    }
    else{
        System.out.println(target.distance+"\t");
        //System.out.println(target.bearing+"\t");
        //System.out.println(isHitWall+"\t");
        //System.out.println(reward+"\t");
        //System.out.println(isHitByBullet+"\t");

    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
    }

        reward = 0.0;
        isHitWall = 0;
        isHitByBullet = 0;

        switch (action)
        {
          case Action.RobotAhead:
            setAhead(Action.RobotMoveDistance);
            break;
          case Action.RobotBack:
            setBack(Action.RobotMoveDistance);
```

```
            break;
          case Action.RobotAheadTurnLeft:
            setAhead(Action.RobotMoveDistance);
            setTurnLeft(Action.RobotTurnDegree);
            break;
          case Action.RobotAheadTurnRight:
            setAhead(Action.RobotMoveDistance);
            setTurnRight(Action.RobotTurnDegree);
            break;
          case Action.RobotBackTurnLeft:
            setAhead(Action.RobotMoveDistance);
            setTurnRight(Action.RobotTurnDegree);
            break;
          case Action.RobotBackTurnRight:
            setAhead(target.bearing);
            setTurnLeft(Action.RobotTurnDegree);
            break;
        }
      }

      public int NeuralNetforAction(double heading, double targetDistance,
double targetBearing, double isHitWall, double isHitByBullet, double reward)
      {
      int action = 0;
      double[]  NN_current_states= new double[5];
      NN_current_states[4]=heading;
      NN_current_states[3]=targetDistance;
      NN_current_states[2]=targetBearing;
      NN_current_states[1]=isHitWall;
      NN_current_states[0]=isHitByBullet;

System.out.println(NN_current_states[0]+"\t"+NN_current_states[1]+"\t"+NN_c
urrent_states[2]+"\t"+NN_current_states[3]+"\t"+NN_current_states[4]+"\t");
      //get the best action
      for(int i=0; i<Action.NumRobotActions; i++)
      {

if(myNet[i].outputFor(NN_current_states)>myNet[action].outputFor(NN_current
_states))
          {
              action = i;
          }
      }
```

```
      //update weights
      double NN_Q_new=myNet[action].outputFor(NN_current_states);
      double error_signal = 0;
      //if(NN_last_states[0]==0.0||NN_last_states[1]==0.0);
     // else
      //{
          error_signal = NN_alpha*(reward + NN_lambda * NN_Q_new -
myNet[NN_last_action].outputFor(NN_last_states));
      //}

      newRobot1.total_error_in_one_round += error_signal*error_signal/2;
      double correct_old_Q = myNet[NN_last_action].outputFor(NN_last_states)
+ error_signal;
      myNet[NN_last_action].train(NN_last_states, correct_old_Q);
      if(Math.random() < NN_epsilon)
      {
          action = new Random().nextInt(Action.NumRobotActions);
      }

      for(int i=0; i<5; i++)
      {
          NN_last_states[i] = NN_current_states[i];
      }

      NN_last_action=action;
      System.out.println(target.distance+"\t");
      return action;

      }

    private int getState()
      {
        int heading = State.getHeading(getHeading());
        int targetDistance = State.getTargetDistance(target.distance);
        int targetBearing = State.getTargetBearing(target.bearing);
        out.println("Ststе(" + heading + ", " + targetDistance + ", " +
targetBearing + ", " + isHitWall + ", " + isHitByBullet + ")");
        int state =
State.Mapping[heading][targetDistance][targetBearing][isHitWall][isHitByBul
let];
        return state;
      }

      private void radarMovement()
```

```
      {
        double radarOffset;
        if (getTime() - target.ctime > 4) { //if we haven't seen anybody for a
bit....
          radarOffset = 4*PI;              //rotate the radar to find a target
        } else {

          //next is the amount we need to rotate the radar by to scan where the
target is now
          radarOffset = getRadarHeadingRadians() - (Math.PI/2 -
Math.atan2(target.y - getY(),target.x - getX()));
          //this adds or subtracts small amounts from the bearing for the radar
to produce the wobbling
          //and make sure we don't lose the target
          radarOffset = NormaliseBearing(radarOffset);
          if (radarOffset < 0)
            radarOffset -= PI/10;
          else
            radarOffset += PI/10;

        }
        //turn the radar
        setTurnRadarLeftRadians(radarOffset);
      }

      private void gunMovement()
      {
        long time;
        long nextTime;
        Point2D.Double p;
        p = new Point2D.Double(target.x, target.y);
        for (int i = 0; i < 20; i++)
        {
          nextTime =
(int)Math.round((getrange(getX(),getY(),p.x,p.y)/(20-(3*firePower))));
          time = getTime() + nextTime - 10;
          p = target.guessPosition(time);
        }
        //offsets the gun by the angle to the next shot based on linear targeting
provided by the enemy class
        double gunOffset = getGunHeadingRadians() - (Math.PI/2 - Math.atan2(p.y
- getY(),p.x -  getX()));
        setTurnGunLeftRadians(NormaliseBearing(gunOffset));
      }
```

```
//bearing is within the -pi to pi range
double NormaliseBearing(double ang){

  if (ang > PI)
    ang -= 2*PI;
  if (ang < -PI)
    ang += 2*PI;
  return ang;
}

//heading within the 0 to 2pi range
double NormaliseHeading(double ang) {
  if (ang > 2*PI)
    ang -= 2*PI;
  if (ang < 0)
    ang += 2*PI;
  return ang;
}

//returns the distance between two x,y coordinates
public double getrange( double x1,double y1, double x2,double y2 )
{
  double xo = x2-x1;
  double yo = y2-y1;
  double h = Math.sqrt( xo*xo + yo*yo );
  return h;
}

//gets the absolute bearing between to x,y coordinates
public double absbearing( double x1,double y1, double x2,double y2 )
{
  double xo = x2-x1;
  double yo = y2-y1;
  double h = getrange( x1,y1, x2,y2 );
  if( xo > 0 && yo > 0 )
  {
    return Math.asin( xo / h );
  }
  if( xo > 0 && yo < 0 )
  {
    return Math.PI - Math.asin( xo / h );
  }
  if( xo < 0 && yo < 0 )
```

```
        {
          return Math.PI + Math.asin( -xo / h );
        }
        if( xo < 0 && yo > 0 )
        {
          return 2.0*Math.PI - Math.asin( -xo / h );
        }
        return 0;
      }



      public void onBulletHit(BulletHitEvent e)
      {

        if (target.name == e.getName())
        {
          double change = e.getBullet().getPower() * 9;
          out.println("Bullet Hit: " + change);
          accumuReward += change;
          //int state = getState();
          //int action = learner.selectAction(state);
         // learner.learn(state, action, change);
          //learner.learnSARSA(state, action, change);
          int action;
          if(NNFlag==false){
              int state = getState();
              action = learner.selectAction(state);

              //learner.learn(state, action, reward);
              learner.learnSARSA(state, action, reward);
              accumuReward+=reward;
             }
             else{

    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
             }
        }
      }


      public void onBulletMissed(BulletMissedEvent e)
      {
```

```
        double change = -e.getBullet().getPower();
        out.println("Bullet Missed: " + change);

        accumuReward += change;
        //int state = getState();
       // int action = learner.selectAction(state);
        //learner.learn(state, action, change);
        //learner.learnSARSA(state, action, change);
        int action;
        if(NNFlag==false){
            int state = getState();
            action = learner.selectAction(state);

            //learner.learn(state, action, reward);
            learner.learnSARSA(state, action, reward);
            accumuReward+=reward;
        }
        else{

    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
        }
      }

      public void onHitByBullet(HitByBulletEvent e)
      {
        if (target.name == e.getName())
        {
          double power = e.getBullet().getPower();
          double change = -(4 * power + 2 * (power - 1));
          out.println("Hit By Bullet: " + change);
          accumuReward += change;
          //int state = getState();
          int action;
          if(NNFlag==false){
              int state = getState();
              action = learner.selectAction(state);

              //learner.learn(state, action, reward);
              learner.learnSARSA(state, action, reward);
              accumuReward+=reward;
            }
            else{
```

```
    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
            }
          //int action = learner.selectAction(state);
          //learner.learn(state, action, change);
          //learner.learnSARSA(state, action, change);
        }
        isHitByBullet = 1;
      }

      public void onHitRobot(HitRobotEvent e)
      {
        if (target.name == e.getName())
        {
          double change = -6.0;
          out.println("Hit Robot: " + change);
          accumuReward += change;
          //int state = getState();
          //int action = learner.selectAction(state);
          //learner.learn(state, action, change);
          //learner.learnSARSA(state, action, change);
          int action;
          if(NNFlag==false){
              int state = getState();
              action = learner.selectAction(state);

              //learner.learn(state, action, reward);
              learner.learnSARSA(state, action, reward);
              accumuReward+=reward;
             }
             else{

    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
              }
        }
      }

      public void onHitWall(HitWallEvent e)
      {

        double change = -(Math.abs(getVelocity()) * 0.5 );
        out.println("Hit Wall: " + change);
        accumuReward += change;
```

```
        //int state = getState();
        //int action = learner.selectAction(state);
        //learner.learn(state, action, change);
        //learner.learnSARSA(state, action, change);
        int action;
        if(NNFlag==false){
            int state = getState();
            action = learner.selectAction(state);

            //learner.learn(state, action, reward);
            learner.learnSARSA(state, action, reward);
            accumuReward+=reward;
        }
        else{

    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
        }
        isHitWall = 1;
      }


      public void onScannedRobot(ScannedRobotEvent e)
      {
        if ((e.getDistance() < target.distance)||(target.name == e.getName()))
        {
          //the next line gets the absolute bearing to the point where the bot
is
          double absbearing_rad =
(getHeadingRadians()+e.getBearingRadians())%(2*PI);
          //this section sets all the information about our target
          target.name = e.getName();
          double h = NormaliseBearing(e.getHeadingRadians() - target.head);
          h = h/(getTime() - target.ctime);
          target.changehead = h;
          target.x = getX()+Math.sin(absbearing_rad)*e.getDistance(); //works
out the x coordinate of where the target is
          target.y = getY()+Math.cos(absbearing_rad)*e.getDistance(); //works
out the y coordinate of where the target is
          target.bearing = e.getBearingRadians();
          target.head = e.getHeadingRadians();
          target.ctime = getTime();             //game time at which this scan
was produced
          target.speed = e.getVelocity();
```

```
            target.distance = e.getDistance();
            target.energy = e.getEnergy();
          }
        }

        public void onRobotDeath(RobotDeathEvent e)
        {

          if (e.getName() == target.name)
          {
          target.distance = 1000;
          }

        }

        public void onWin(WinEvent event)
        {
        File file0 = getDataFile("NN_weights_from_LUT0.dat");
        File file1 = getDataFile("NN_weights_from_LUT0.dat");
        File file2 = getDataFile("NN_weights_from_LUT0.dat");
        File file3 = getDataFile("NN_weights_from_LUT0.dat");
        File file4 = getDataFile("NN_weights_from_LUT0.dat");
        try {
              save(file0,myNet[0]);
              save(file1,myNet[1]);
              save(file2,myNet[2]);
              save(file3,myNet[3]);
              save(file4,myNet[4]);
          } catch (IOException e1) {
              // TODO Auto-generated catch block
              e1.printStackTrace();
          }
        /*for(int j=0;j<Action.NumRobotActions;j++){
              try {
                    myNet[j].save(new
File("C:/robocode/robots/rlRobotNew/newRobot1.data/NN_weights_from_LUT"+j+"
.txt"));
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
              }*/
    File file = getDataFile("accumReward.dat");
    accumuReward+=rewardForWin;
```

```
    int action;
    if(NNFlag==false){
        int state = getState();
        action = learner.selectAction(state);

        //learner.learn(state, action, reward);
        learner.learnSARSA(state, action, reward);
        accumuReward+=reward;
    }
    else{

    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
    }
    //int state = getState();
    //int action = learner.selectAction(state);
    //learner.learn(state, action, rewardForWin);
    //learner.learnSARSA(state, action, rewardForWin);
        robotMovement();
        saveData();
         int winningFlag=7;
         countForWin++;
         count++;
         PrintStream w = null;
            try
            {
              w = new PrintStream(new
RobocodeFileOutputStream(file.getAbsolutePath(), true));
             if(count==50){
                    count=0;
             w.println(accumuReward+" "+countForWin*2+"\t"+winningFlag+" "+"
"+learner.explorationRate);
             accumuReward=0;
                countForWin=0;
             if (w.checkError())
               System.out.println("Could not save the data!");
//setTurnLeft(180 - (target.bearing + 90 - 30));
             w.close();
             }
            }
            catch (IOException e)
            {
              System.out.println("IOException trying to write: " + e);
            }
```

```
            finally
            {
              try
              {
                if (w != null)
                  w.close();
              }
              catch (Exception e)
              {
                System.out.println("Exception trying to close witer: " + e);
              }

            }
      }
      public void save(File file,BpNetWork myNet) throws IOException {

         PrintStream w = null;
            try
            {
              w = new PrintStream(new
RobocodeFileOutputStream(file.getAbsolutePath(), true));

            //for(int i = 0; i<this.layerWeight.length; i++)
            //{
                int i=0;

                //System.out.println(this.layerWeight[i].length);

                for(int j=0;j<myNet.layerWeight[i].length;j++){
                    //System.out.println(this.layerWeight[i][j].length);
                    //System.out.println(this.layerWeight[i][j][0]);
                    for(int k=0;k<myNet.layerWeight[i][j].length;k++)
                        {
                        w.println(myNet.layerWeight[i][j][k]);

                        }


                }
                //if(this.writeWeights(weight_vector,argFile) == 0)
                    //System.out.println("Writing hidden nodes fails");

                i++;
                for(int j=0;j<myNet.layerWeight[i].length;j++){
```

```
                //System.out.println(this.layerWeight[i][j].length);
                //System.out.println(this.layerWeight[i][j][0]);
                w.println(myNet.layerWeight[i][j][0]);


            }
             if (w.checkError())
                   System.out.println("Could not save the data!");
                w.close();

        //}

        }
        catch (IOException e)
        {
          System.out.println("IOException trying to write: " + e);
        }
        finally
        {
          try
          {
            if (w != null)
              w.close();
          }
          catch (Exception e)
          {
            System.out.println("Exception trying to close witer: " + e);
          }

        }



  }
public void onDeath(DeathEvent event)
{
File file0 = getDataFile("NN_weights_from_LUT0.dat");
File file1 = getDataFile("NN_weights_from_LUT1.dat");
File file2 = getDataFile("NN_weights_from_LUT2.dat");
File file3 = getDataFile("NN_weights_from_LUT3.dat");
File file4 = getDataFile("NN_weights_from_LUT4.dat");
try {
        save(file0,myNet[0]);
        save(file1,myNet[1]);
```

```
            save(file2,myNet[2]);
            save(file3,myNet[3]);
            save(file4,myNet[4]);
        } catch (IOException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
      accumuReward+=rewardForDeath;
      int action;
        if(NNFlag==false){
            int state = getState();
            action = learner.selectAction(state);

            //learner.learn(state, action, reward);
            learner.learnSARSA(state, action, reward);
            accumuReward+=reward;
        }
        else{

    action=this.NeuralNetforAction(getHeading()/180-1,target.distance/500-1
,target.bearing/180-1,isHitWall*2-1,isHitByBullet*2-1, reward);
        }
      //int state = getState();
          //int action = learner.selectAction(state);
         // learner.learn(state, action, rewardForDeath);
          //learner.learnSARSA(state, action, rewardForDeath);
          count++;

        saveData();
        File file = getDataFile("accumReward.dat");
        int losingFlag=5;
         PrintStream w = null;
            try
            {
              w = new PrintStream(new
RobocodeFileOutputStream(file.getAbsolutePath(), true));
              if(count==50){
                count=0;
              w.println(accumuReward+" "+countForWin*2+"\t"+losingFlag+" "+"
"+learner.explorationRate);
              accumuReward=0;
              countForWin=0;
              if (w.checkError())
                System.out.println("Could not save the data!");
```

```
        w.close();
        }
      }
      catch (IOException e)
      {
        System.out.println("IOException trying to write: " + e);
      }
      finally
      {
        try
        {
          if (w != null)
            w.close();
        }
        catch (Exception e)
        {
          System.out.println("Exception trying to close witer: " + e);
        }

      }
}

public void loadData()
{
  try
  {
    table.loadData(getDataFile("movement.dat"));
  }
  catch (Exception e)
  {
  }
}

public void saveData()
{
  try
  {
    table.saveData(getDataFile("movement.dat"));
  }
  catch (Exception e)
  {
    out.println("Exception trying to write: " + e);
  }
}
```

```
    }

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

import NNRobot.Action;



public class NNRL {
     public static final int NumHeading = 4;
     public static final int NumTargetDistance = 10;
     public static final int NumTargetBearing = 4;
     public static final int NumHitWall = 2;
     public static final int NumHitByBullet = 2;

     public static final int NumActions = 6;

     public static final int
NumSpace=NumHeading*NumTargetDistance*NumTargetBearing*NumHitWall*NumHitByB
ullet*NumActions;

     //Load look-up table
     private static double[] table_loaded = new double[NumSpace];

     //Process the look-up table
     private static double[] table_processed = new double[NumSpace];

     //learning rate and momentum
     public static final double learningRate = 0.1;

     public static final double momentum = 0.9;

     //threshold
     public static final double threshold = 0.01;

     //Initialization
     private void initialize()
     {
         for(int i=0; i< NumSpace; i++)
         {
```

```
            table_loaded[i]=0;
            table_processed[i]=0;
        }
    }
    public static void main(String[] args) throws IOException{
        int[] layernum=new int[3];
        layernum[0]=5;
        layernum[1]=15;
        layernum[2]=1;
        double mobp=0.9;
        double rate=0.01;

//loadData("C:/robocode/robots/rlRobotNew/newRobot1.data/movement.dat");

load_table("C:/robocode/robots/rlRobot/newRobot1.data/movement.dat");
        process_table();//set 1 for the action with highest Q value for each
state

        double[][] stateInput = new double [NumSpace][5];
        int StateNum=0;
           for (int a = 0; a < NumHeading; a++)
               for (int b = 0; b < NumTargetDistance; b++)
                 for (int c = 0; c < NumTargetBearing; c++)
                   for (int d = 0; d < NumHitWall; d++)
                     for (int e = 0; e < NumHitByBullet; e++) {
                      /*
                          stateInput[StateNum][0]=(e+1)/2;
                          stateInput[StateNum][1]=(d+1)/2;
                          stateInput[StateNum][2]=(c+1)/4;
                          stateInput[StateNum][3]=(b+1)/20;
                          stateInput[StateNum][4]=(a+1)/4;*/
                      stateInput[StateNum][0]=e;
                      stateInput[StateNum][0]/=2.0;
                      stateInput[StateNum][1]=d;
                      stateInput[StateNum][1]/=2.0;
                      stateInput[StateNum][2]=c;
                      stateInput[StateNum][2]/=4.0;
                      stateInput[StateNum][3]=b;
                      stateInput[StateNum][3]/=10.0;
                      stateInput[StateNum][4]=a;
                      stateInput[StateNum][4]/=4.0;
                          StateNum=StateNum+1;
                      }
        //build network with multiple nets.
```

```
        BpNetWork[] myNet=new BpNetWork[NumActions];
        double[] maxminQ=new double[2];
        maxminQ=findMaxMinQ(table_processed);
        //maxminQ=findMaxMinQ(table_loaded);
        //System.out.println("maxQ:"+maxminQ[0]+"minQ:"+maxminQ[1]+"\n");
        for(int i=0;i<NumActions;i++){
            myNet[i]=new
BpNetWork(layernum,rate,mobp,maxminQ[1],maxminQ[0]);
            myNet[i].initializeWeights();
        }

        final int epoch=10000;

       //double[] inp=new double[]{1,2,3,4,5,6};
       //double out=1;
       //myNet.train(inp, out);
       //Training
        double total_error[] = new double[]{0,0,0,0,0,0};
        double error = 0;
        double max_error = 0.0;
        int iteration=0;
        //System.out.println(NumSpace/NumActions);

        for(;iteration<epoch;){
            max_error=0.0;
            double totalErr=0;
            for(int j=0;j<NumActions;j++){
                total_error[j]=0;
            for(int i=0; i<NumSpace/NumActions;i++)//NumSpace/NumActions;
i++)
            {

                    double[] inputArray;
                    inputArray=new double[5];
                    //for(int k=0;k<5;k++){
                        inputArray[0]=stateInput[i*NumActions][0];
                        inputArray[1]=stateInput[i*NumActions][1];
                        inputArray[2]=stateInput[i*NumActions][2];
                        inputArray[3]=stateInput[i*NumActions][3];
                        inputArray[4]=stateInput[i*NumActions][4];
                    //
System.out.println(inputArray[0]+"\t"+inputArray[1]+"\t"+inputArray[2]+"\t"
+inputArray[3]+"\t"+inputArray[4]+"\t");
```

```
//inputArray=generateInputVector(generateInputandActionFromTable(i*NumActio
ns));
                    double outputExp=generateCorrectOutput(i*NumActions,j);
                    //System.out.println(outputExp+"\n");
                    error = Math.pow(myNet[j].train(inputArray,outputExp),
2);
                    total_error[j] += error;
                    if(max_error<Math.abs(error))
                    max_error=Math.abs(error);
                }
            //System.out.println("total_error["+j+"]:"+
total_error[j]+"\n");
            }

            iteration++;
            for(int m=0;m<6;m++){
                total_error[m]=Math.sqrt(total_error[m]/NumSpace);
                totalErr+=total_error[m];

            }
            System.out.println(iteration+" total_error "+
totalErr+'\t'+total_error[0]+'\t'+total_error[1]+'\t'+total_error[2]+'\t'+t
otal_error[3]+'\t'+total_error[4]+'\n');

            if(max_error < threshold) break;
        }


        for(int j=0;j<NumActions;j++){
        myNet[j].save(new
File("C:/robocode/robots/rlRobotNew/newRobot1.data/NN_weights_from_LUT"+j+"
.txt"));
        }
        /*for(int i=0;i<Action.NumRobotActions;i++){
           //myNet[i]=new BpNetWork();
           try {

    myNet[i].loadWeight("C:/robocode/robots/rlRobotNew/newRobot1.data/NN_we
ights_from_LUT"+i+".txt");
               } catch (IOException e) {
                   // TODO Auto-generated catch block
                   e.printStackTrace();
               }
```

```
        }*/


//System.out.println(myNet[0].layerWeight[0][0][0]+"\t"+myNet[0].layerWeigh
t[1][0][0]+"\t"+myNet[0].layerWeight[1][15][0]);

         /*
         //build one network

         BpNetWork myNet=new BpNetWork(layernum,rate,mobp,0,1);
         double[] maxminQ=new double[2];
         maxminQ=findMaxMinQ(table_loaded);
         System.out.println("maxQ:"+maxminQ[0]+"minQ:"+maxminQ[1]+"\n");

         final int epoch=10;
        //double[] inp=new double[]{1,2,3,4,5,6};
        //double out=1;
        //myNet.train(inp, out);
        //Training
         double total_error = 0;
         double error = 0;
         double max_error = 0.0;
         int iteration=0;
         System.out.println(NumSpace/NumActions);
         for(;iteration<epoch;){
             max_error=0.0;

             for(int i=0; i<NumSpace; i++)
             {

                     double[] inputArray=new double[5];
                     inputArray=generateInputandActionFromTable(i);
                     double outputExp=generateCorrectOutput(i,i%NumActions);
                     error = Math.pow(myNet.train(inputArray,outputExp), 2)/2;
                     total_error += error;
                     if(max_error<Math.abs(error))
                     max_error=Math.abs(error);
                 }
             //System.out.println("total_error["+j+"]:"+
total_error[j]+"\n");

             iteration++;
             System.out.println(" max_error "+ max_error+" iteration
"+iteration);
```

```
            if(max_error < threshold)
                break;
        }


        }
    */

}


        private static double[] findMaxMinQ(double[] table) {
        double[] maxminQ=new double[2];
        maxminQ[0]=Integer.MIN_VALUE;
        maxminQ[1]=Integer.MAX_VALUE;
            for(int i=0;i<table.length;i++){
            if(table[i]>maxminQ[0])
                maxminQ[0]=table[i];
            if(table[i]<maxminQ[1])
                maxminQ[1]=table[i];
            //if(table[i]!=0)
            //System.out.println(table[i]+"\n");
        }
            //System.out.println(table.length+"\n");
            return maxminQ;
    }
        //compute error
         public static double computeError(double[] errorFromOutput)
         {
             double total_error = 0;
             for(int i = 0; i < errorFromOutput.length; i++)
             {
                 total_error += errorFromOutput[i];
             }

             return total_error;
         }

         //load table
         public static void load_table(String filename) throws IOException
         {
             File readFile = new File(filename);
             BufferedReader br = new BufferedReader(new FileReader(readFile));
             String str;
```

```
    int count = 0;

    while((str = br.readLine())!= null)
    {
        if(count < NumSpace)
        {
            table_loaded[count]=Double.parseDouble(str);
            count++;
        }
        else
        {
            break;
        }
    }

    br.close();
}

public static void process_table()
{
    for(int i=0; i<NumSpace; i=i+NumActions)
    {
        int max = 0;

        for(int j=0; j<NumActions; j++)
        {
            if(table_loaded[i+j]<table_loaded[i+max])
            {
                table_processed[i+j] = 0.0;
            }
            else
            {
                table_processed[i+max] = 0.0;
                table_processed[i+j] = 1.0;
                max = j;
            }
        }
    }
}

//There is an action within return array
public static double[] generateInputandActionFromTable(int index)
{
    int heading =
```

```
index/(NumTargetDistance*NumTargetBearing*NumHitWall*NumHitByBullet*NumActi
ons);
            int left = index %
(NumTargetDistance*NumTargetBearing*NumHitWall*NumHitByBullet*NumActions);
            int targetDistances =
left/(NumTargetBearing*NumHitWall*NumHitByBullet*NumActions);
            left = left %
(NumTargetBearing*NumHitWall*NumHitByBullet*NumActions);
            int targetBearing = left/(NumHitWall*NumHitByBullet*NumActions);
            left = left % (NumHitWall*NumHitByBullet*NumActions);
            int hitWall = left/(NumHitByBullet*NumActions);
            left = left % (NumHitByBullet*NumActions);
            int hitByBullet = left % NumActions;
            int action = left;  //This action might be wrong if index is not
times of Action

            double[] return_array = new double[6];

            return_array[0]=heading;
            return_array[1]=targetDistances;
            return_array[2]=targetBearing;
            return_array[3]=hitWall;
            return_array[4]=hitByBullet;
            return_array[5]=action;

            return return_array;
        }

        public static double[] generateInputVector(double [] table_array)
        {
            if(table_array.length !=6)
            {
                System.out.println("Wrong Array Length");
            }

            double[] return_array = new double[5];
            return_array[0] = table_array[0];
            return_array[1] = table_array[1];
            return_array[2] = table_array[2];
            return_array[3] = table_array[3];
            return_array[4] = table_array[4];

            return return_array;
```

```
        }

        public static double generateCorrectOutput(int index,int j)
        //for all combination of action and state as index (input of NN), get
the
        //correctOutput as whether this state+action is chosen or not with the
table _processed (==1 means the Q is the highest and it is the chosen one)
        {
            int state_index = index - index%NumActions;
            double correctOutput;
            //for(int i=0; i<NumActions; i++)

           correctOutput=table_processed[state_index + j];
           //correctOutput=table_loaded[state_index + j];

            return correctOutput;
        }




}
public class State
{
  public static final int NumHeading = 4;
  public static final int NumTargetDistance = 10;
  public static final int NumTargetBearing = 4;
  public static final int NumHitWall = 2;
  public static final int NumHitByBullet = 2;
  public static final int NumStates;
  public static final int Mapping[][][][][][];

  static
  {
    Mapping = new
int[NumHeading][NumTargetDistance][NumTargetBearing][NumHitWall][NumHitByBu
llet];
    int count = 0;
    for (int a = 0; a < NumHeading; a++)
      for (int b = 0; b < NumTargetDistance; b++)
        for (int c = 0; c < NumTargetBearing; c++)
          for (int d = 0; d < NumHitWall; d++)
```

```
            for (int e = 0; e < NumHitByBullet; e++)
          Mapping[a][b][c][d][e] = count++;

    NumStates = count;
  }

  public static int getHeading(double heading)
  {
    double angle = 360 / NumHeading;
    double newHeading = heading + angle / 2;
    if (newHeading > 360.0)
      newHeading -= 360.0;
    return (int)(newHeading / angle);
  }

  public static int getTargetDistance(double value)
  {
    int distance = (int)(value / 30.0);
    if (distance > NumTargetDistance - 1)
      distance = NumTargetDistance - 1;
    return distance;
  }

  public static int getTargetBearing(double bearing)
  {
    double PIx2 = Math.PI * 2;
    if (bearing < 0)
      bearing = PIx2 + bearing;
    double angle = PIx2 / NumTargetBearing;
    double newBearing = bearing + angle / 2;
    if (newBearing > PIx2)
      newBearing -= PIx2;
    return (int)(newBearing / angle);
  }

}
```