Part 3 – Reinforcement Learning with Backpropagation

Zhen Wang 98552169

Introduction

In the third and last part I am to replace the LUT used by the Reinforcement Learning component with a neural net.

First of all, I have to determine the structure and parameters of the neural network. I choose to use six neural networks each corresponding to an action and taking states as the input. By using the results from part 1 and do comparisons between hidden layer nodes, I determine the parameters of learning rate, momentum factor, the activation function and number of hidden nodes.

Then have to use the result LUT from the previous part, part 2 to pre-train the neural network so as to initialize the weight of neural net by load it into the robocode.

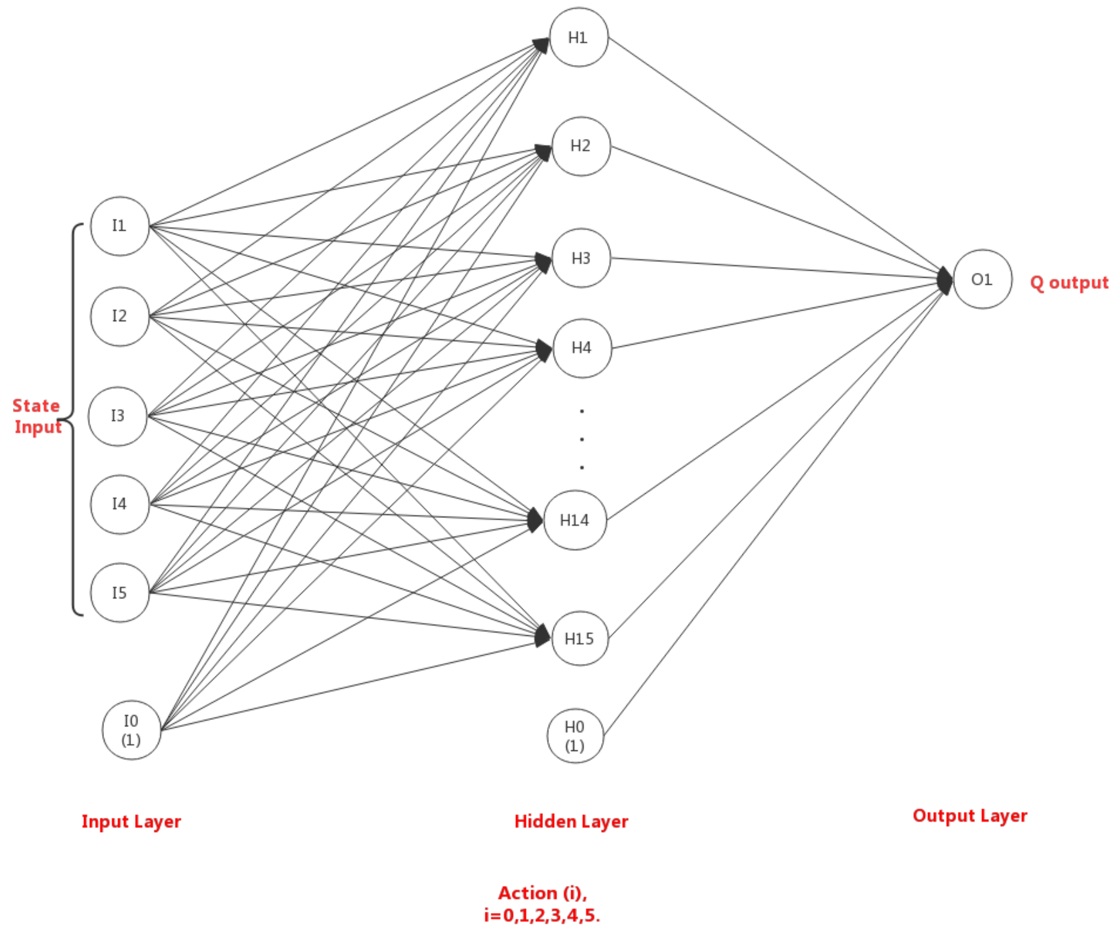
During the running of the robocode, the myNewRobot always fight against myFirstRobot which is also chosen in part 2. The actions are still selected based on Q-learning with the maximum Q-value generated by neural networks. And the neural networks will experience on-line training during this process.

Finally, winning rate and e(s) will be collected to be analyzed.

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) Describe the architecture of your neural network and how the training set captured from Part 2 was used to "offline" train it mentioning any input representations that you may have considered. Note that you have 3 different options for the high level architecture. A net with a single Q output, a net with a Q output for each action, separate nets each with a single output for each action. Draw a diagram for your neural net labeling the inputs and outputs.



Graph 1. Network architecture

In this part, I constructed six (the same number as actions) three-layer networks with the form above. Each of the network represents the training performance of one action from the Action set {ahead, back, ahead left, ahead right, back left, back right}. Each of the network has five inputs I1 to I5 which represent five states from the State set {heading, distance from the target, target bearing, hit by the wall or not, hit by bullet or not}. And one more input as the bias node. H1 to H15 represent 15 hidden layer nodes and also there is one more bias node H0 in this layer. O1 is the single output for a network which represents the Q value of the corresponding state - action combination. Both the State set and the Action set are the same as the ones used in part 2. With reduction of state space, the total state number is 640 with each having six actions, thus the total training set is of 3840 data.

b) Show (as a graph) the results of training your neural network using the contents of the LUT from Part 2. You may have attempted learning using different hyper-parameter values (i.e. momentum, learning rate, number of hidden neurons). Include graphs showing which parameters best learned your LUT data. Compute the RMS error for your best results.

To train the net with data in the training set and take bipolar sigmoid as the activation function, normalization for the Q value should be done by setting them to be in the range of -1 to +1.

RMS is computed with this equation: $RMS = \sqrt{\frac{1}{n}\sum_{i=1to6} TotalError * 2}$

The problem occurred when I maintained the number of the hidden layer nodes in previous part as 4, the convergence of RMS is not obvious and the value of RMS is always too big. To get better result I decided to try different numbers of nodes for the hidden layer to see which number is the best one.

Table 1. RMS of different numbers of hidden layer nodes after 1000 iterations

(learning rate=0.1 and momentum=0.9)

	4	10	15	20	25
RMS	5.6809	2.7751	0.9768	1.1223	1.1586

I also tried some different value of learning rates to choose the right one.

Table 2. RMS of different learning rates after 1000 iterations

(hidden layer nodes=15 and momentum=0.9)

	0.1	0.2	0.3	0.5	0.9
RMS	0.9768	0.9832	0.9782	1.0022	0.9968

Based on the table above, we can conclude that after 1,000 iterations when there are 15 hidden layer nodes and the learning rate is 0.1 (it seems the learning rate does not have a big influence), the RMS is the smallest. And actually when the number of hidden layer nodes continue to increase, the RMS will increase and even oscillate because of overfitting.

To sum up, the hyper-parameters are used as below,

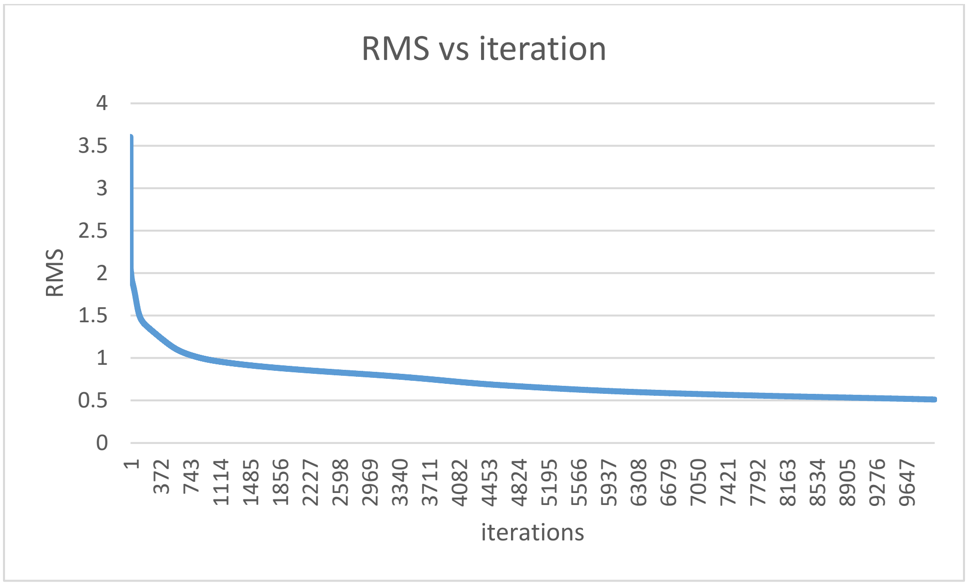
Number of hidden layer nodes: 15

Learning rate: 0.1

Momentum: 0.9

Activation function: Bipolar sigmoid

And in this condition, the convergence of RMS is shown below, after 10,000 iterations the RMS could be converged to about 0.5112.



Graph 2. RMS trend over 10,000 iterations

When finishing this part, the nets or specifically the weigh matrix of the nets were exported respectively to six .txt files. This is actually the preparation work for the robocode.

c) Try mitigating or even removing any quantization or dimensionality reduction (henceforth referred to as state space reduction) that you may have used in part 2. A side-by-side comparison of the input representation used in Part 2 with that used by the neural net in Part 3 should be provided. (Provide an example of a sample input/output vector). Compare using graphs, the results of your robot from Part 2 (LUT with state space reduction) and your neural net based robot using less or no state space reduction. Show your results and offer an explanation.

Both in part 2 using the Lookup Table and part 3 using neural net, I used the same five states as heading, distance from the target, target bearing, hit by the wall or not, hit by bullet or not. Considering the robocode environment, the state describing directions is of the range 0 to 360 degree and the distance is 0 to 1000 as the width is 600 and the length is 800.

In part 2, I quantified these states into discrete values. For example, using 1, 2, 3, 4 to show four directions of Up, Down, Left and Right and using number 1 to 10 to represent the range of 0 to 1000. Thus the whole huge state space is reduced to only 640 states.

When removing the reduction, I directly used the value received without quantification and normalized them all into the range -1 to 1 as the input to the neural network (the same thing was done when training the network).

The table below shows in detail about the comparison of input values.

Table 3. input representation

Input No.	Input representation	Value Without Quantification	Value With Reduction	Value when Reduction removed
I0	Heading	0~360	1, 2, 3, 4	-1~1
I1	Distance from target	0~1000	1~10	-1~1
I2	Target bearing	0~360	1, 2, 3, 4	-1~1
I3	Hit by the wall or not	0, 1	0, 1	-1, 1
I4	Hit by bullet or not	0, 1	0, 1	-1, 1

I chose the best result in part 2 to do the comparison in this part. That is,

Off-policy Q-learning

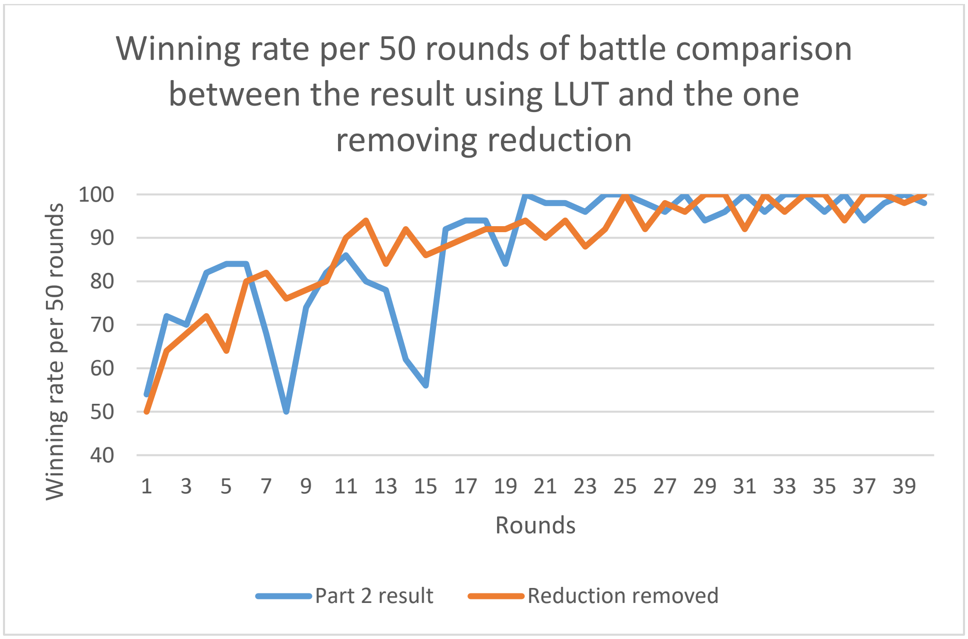
Exploration rate: 0

Learning rate: 0.1

Discount factor: 0.9

By trying many times, I found a good situation that using the look up table can result in nearly 100% winning rate. Then use it to train the neural network and import the weight array to robocode to do online training.

The graph below showed the trends of the winning rate in 2000 rounds of battles of the one with look up table and the one after reduction removed.



Graph 3. Winning rate comparison between RL with LUT and RL with NN

From it, at the very beginning, it seems that the look up table is faster than the NN, but the look up table-based RL has more oscillations. Finally, both of them converge to the same level of winning rate between 90% and 100%.

d) 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.

Look up table is a table, which means that it can only store discretized values. In part 2, we have to store the pairs each of which corresponds to one state and one action. When the state-action space is too huge, it is not a good idea to store all the data in the space. Just like the table in c) that if all the state-action pairs are to be stored in the look up table, it will take 360*1000*360*2*2*6=3.1104*10^9 spaces and this only considers that the integer states. With such huge input states, huge memory resources are required and the running speed and learning speed will be greatly decreased when running the code especially when save and load the data.

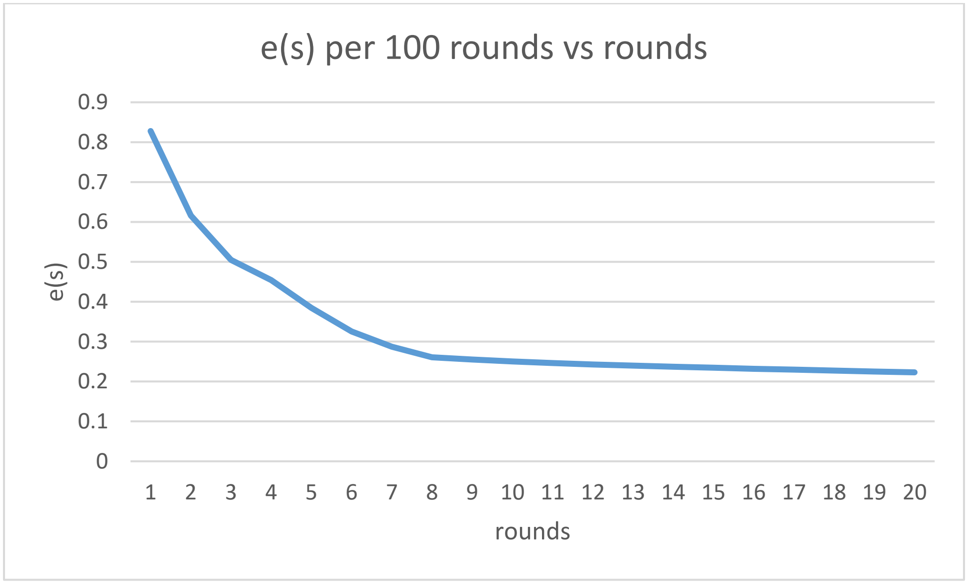
However, the neural network does not have such problem because we can directly use all kinds of values as the input of the neural net without storing the Q value to a table. Actually the well-trained neural net itself is the storage of different Q results of states and actions. It only needs to store the weights between different nodes which require obviously fewer memory resources than the look up table.

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) What was the best win rate observed for your tank? Describe clearly how your results were obtained? Measure and plot e(s) (compute as Q(s',a')-Q(s,a)) for some selected state-action pairs. Your answer should provide graphs to support your results. Remember here you are measuring the performance of your robot online. I.e. during battle.

After removing the reduction of state-action space and with the NN-reinforcement learning, the best winning rate of every 50 rounds could reach 100% when beating the same target robot as part 2 whose name is My First Robot. It happened a lot of time when iterating for over 1000 times.

Then $e(s) = Q(s', a') - Q(s, a)$ was computed and displayed in the below graph for the state-action labeled 0-0. To make the trend more explicit, calculate the average e per 100 rounds.

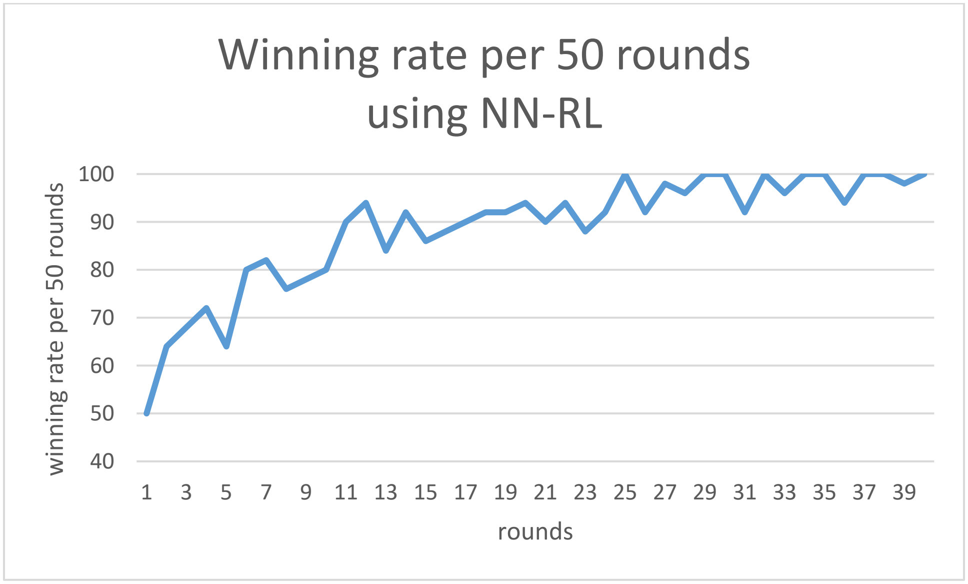


Graph 4. E(s) trend over 2,000 battle rounds

It is clear that the total trend of e is quickly decreasing from 0.83 at the very beginning and converge to nearly 0.2.

b) Plot the win rate against number of battles. As training proceeds, does the win rate improve asymptotically?

The graph below shows the winning rate of every 50 rounds of battles and there are totally 2000 rounds.



Graph 5. Winning rate of RL with NN over 2,000 battle rounds

The winning rate increased from around 50% to over 90% and kept stable. Sometimes there will appear some 100% winning rate of 50 rounds.

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 Qfunction is approximated. Explain this convergence in terms of the Bellman equation and also why when using approximation, convergence is no longer guaranteed.

From the lecture handout, we learned the Bellman equation that,

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

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 immediate reward at t. γ is the discount factor. This equation is recursive obviously and implies that the values of successive states have relationship with each other.

If use * to denote the optimal value function. We can have similar equations for the Bellman equation with optimal value function,

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

We define the error in the value function at any given state to be represented by $e(s_t)$. So it is clear that

$$e(s_t) = V(s_t) - V^*(s_t) \text{ and}$$

$$e(s_{t+1}) = V(s_{t+1}) - V^*(s_{t+1})$$

Then we replace the $V(s_t)$, $V(s_{t+1})$ and using Bellman equation and get the format below,

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

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

And as we know $V^*(s_t) = r_t + \gamma V^*(s_{t+1})$, the above equation can be transferred to

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

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

From this, we can know that the error in successive states is related. If $e(s_T)$ is used to denote the terminal error, we can assume $e(s_T) = 0$ when it is trained for long enough iterations with Q-learning and it finally converges to the optimal value function.

However, when it comes to the neural network, the approximate of the value function could lead to some other additional errors. Thus in this condition, we change the form of error equation as below,

$$e(s_t) + e_{additional} = V(s_t) - V^*(s_t)$$

Where $e_{additional}$ represents the additional error caused by the approximate of the value function.

The $e_{additional}$ is determined by the neural net, so when using approximate of value function in the NN it is not guaranteed to decrease and converge. Thus the value function of $V(s_t)$ cannot finally converge to the optimal value function $V^*(s_t)$ for this reason.

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. Hint: Read up on experience replay.

As there is no a-priori training set to work with, we cannot get the total error during the online learning process. Thus we have to think about other ways to show the learning performance.

Just as what is shown in the previous part, there are at least 2 ways to monitor the learning performance.

The first one is using the error between $Q(s', a')$ and $Q(s, a)$. We define $e(s) = abs(Q(s', a') - Q(s, a))$, where abs() computes the absolute value, so $e(s)$ is the absolute error. If the neural net is learning, $e(s)$ will decrease with the iteration proceeding and finally converge to around zero. If the neural net is not learning, such decrease will not be found.

The second method is using the winning rate. As it is mentioned before, we can use the winning rate of every 50 rounds of battles to see the trend of the winning rate. If the neural net is learning, the winning rate will keep increase until converge to a high level of winning rate.

Other methods may also works such as using the accumulated rewards.

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?

(1) The practical issues surrounding the application of RL & BP

1. There is no universal neural network working against all kinds of target robots. From the process of offline training the NN with LUT, we know that we have to use the LUT corresponding to specific robots in part 2 to train the NN. So one NN is only guaranteed to work against one kind of target robot. If you want to defeat another robot, you have to find another LUT and repeat the training again.
2. When trying to improve the performance of the robot, we have to try more input data presented as states and actions. This means that more complicated structure of neural net should be applied.

The increase of hidden layers and hidden neurons would lead to longer training time and sometimes even result in the problem of overfitting.

3. When training the NN, the selection of parameters including learning rate, momentum factor, the structure of neural networks should be carefully considered. The process of trying different combination of these parameters is quite long and the improper selection of the parameters may lead to a very long time of learning and converging, and sometimes it may even not converge.

(2) Ways to improve the robot performance

1. When thinking about ways to have better robot performance, I tried to apply different kind of neural net of using one network for all state-action data and using several networks with states as input, each corresponding to one action. I found that using several networks can have a better result. Since for every action, the networks are separated and will not share network weights and the output node. Thus it would be more accurate in terms of the approximate of the Q-learning value function. This can finally reduce the RMS
2. Change the structure of the network may also works. To get better approximate of function and fit the data better, proper increase of the complexity of the network can be done by using more hidden neurons and more hidden layers.

(3) Suggestions for addressing convergence problems

1. Using some other activation function other than bipolar sigmoid function may get better result. The drawback of sigmoid function is vital that when the input is too large or too small, the gradients for neurons may be killed, which leads to sigmoid saturation. Some other activation functions could have better performance. For example, Tanh which is could be treated as a different version of sigmoid or ReLu which is popular these years may works better.
2. In the network, the variable step size can be used to replace the constant step size.

(4) Advice for other applications of RL with neural network based functional approximation

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.

(1) Concerns:

There are a few concerns with the application of NN-reinforcement learning to the automatically delivering anesthetic patient under going surgery. Some of the concerns are vital.

Let us consider the process of using the NN based reinforcement learning. When we do the offline training of the neural network, huge amount of surgery information should be available so that to train the neural network to be as precise as possible. But the surgery information itself is not controllable and especially for that the more difficult the anesthetic step in the surgery is, the available data is rarer. Thus this concern of training set data is a big problem.

Second, when we training online the neural network, the most significant concern should be that it may lead to medical accidents. We know that the errors in the training process could not be avoided although it will converge. But the real cases to deliver anesthetic cannot bear any risk of the errors which may lead to the surgery unsuccessful and even be fatal to the patients' health.

Third, what makes it much difficult to realize is that the health condition on patients may change and in real cases there will be anesthetist doing real time monitoring on patient's condition, which means the current established neural net may not work for the new kind of conditions. If the neural network cannot get corrected in time and respond quickly to the change, irreparable consequences may occur.

(2) Variation

The potential variation that I can give to alleviate these concerns is that we can induce risk control into the process of applying reinforcement learning. Before we use this RL method, we have to evaluate the risk of failure. If it could reach the successful rate as the anesthetists do, it can be done automatically. Also, with the development of medicine, more valuable data will be available which means the training set could be more reliable. And the development of machine learning and control theories could provide better even perfect models which could finally result in the application of RL in automatically delivering anesthetic.

Appendix-code

```
package rlRobotNew;

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*Nu
mTargetBearing*NumHitWall*NumHitByBullet*Nu
mActions;
```

```
        //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/new
Robot1.data/movement.dat");


load_table("C:/robocode/robots/rlRobot/newRob
ot1.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],max
minQ[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/NumAc
tions; 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"+inp
utArray[4]+"\t");

//inputArray=generateInputVector(generateInputa
ndActionFromTable(i*NumActions));
                                        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]/NumSpa
ce);

totalErr+=total_error[m];

                        }

System.out.println(iteration+"    total_error    "+
totalErr+'\t'+total_error[0]+'\t'+total_error[1]+'\t'+
total_error[2]+'\t'+total_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_weights_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].layerWeight[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%NumAction
s);
                                                  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*NumActions);
			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_ind
ex + j];

            //correctOutput=table_loaded[state_index
+ j];

                        return correctOutput;
                }

}
package rlRobotNew;


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"+myNet[i].layerWeight[0][0].length+"\t");

		try {
```

```
            myNet[i].loadWeight("C:/robocode/robots
/rlRobotNew/newRobot1.data/NN_weights_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(getHeadin
g()/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_current_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("Stste(" + heading + ", " + targetDistance + ", " + targetBearing + ", " + isHitWall + ", " + isHitByBullet + ")");
      int state = State.Mapping[heading][targetDistance][targetBearing][isHitWall][isHitByBullet];
      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(getHeadin
g()/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(getHeadin
g()/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(getHeadin
g()/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(getHeadin
g()/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(getHeadin
g()/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(getHeadin
g()/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,argFil
e) == 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].l
ength);

	//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(getHeadin
g()/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);
    }
  }
 }
 package rlRobotNew;




 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;//神经网络各层节点
    public double[][] layError;//神经网络各节点误差
    public double[][][] layerWeight;//各层节点权重
    public double[][][] layerWDelta;//各层节点权重动量

        //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;
        };
```

```
	/**
	* This method implements a general sigmoid with asymptotes bounded by (a,b)
	* @param x The input
	* @return f(x) = b_minus_a / (1 + e(-x)) - minus_a
	*/
	public double customSigmoid(double x){
		return (argB-argA)/(1+Math.pow(Math.E, -x))+argA;
	};


	/**
	* 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(){
		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];
             }




             /**
             * 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){
            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.argLearningRate*layError[2][0]*layer[1][j];
        layerWDelta[1][j][0]=layerWeight[1][j][0]-preW;
    }
layerWDelta[1][this.argNumHidden][0]=this.argMomentumTerm*layerWDelta[1][this.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)*(argB-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.argLearningRate*layError[1][j]*layer[0][i];
layerWeight[0][i][j]+=layerWDelta[0][i][j];
        }
layerWDelta[0][this.argNumInputs][j]=this.argMomentumTerm*layerWDelta[0][this.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;
  }
```

```
	/**
	 * A method to write either a LUT or weights of an neural net to a file.
	 * @param argFile of type File.
	* @throws FileNotFoundException
	* @throws IOException
	 */
	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;
			}

	/**
			* Loads the weights from file. Format of the file is expected to follow
			* that specified in the "save" method specified elsewhere in this class.
			* @param argFileName the name of the file where the weights are to be found
			*/
		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();

		}

	/**
	 * 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{
		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.util.Random;



  public class Learner
  {
    public static final double LearningRate = 0.1;
    public static final double DiscountRate = 0.9;
    public static double explorationRate = 0;
    private int lastState;
    private int lastAction;
    private boolean first = true;
    private QTable table;


    public Learner(QTable table)
    {
      this.table = table;
    }


    public void learn(int state, int action, double
reward)
    {
      if (first)
        first = false;
      else
      {
        double oldQValue = table.getQValue(lastState,
lastAction);
        double newQValue = (1 - LearningRate) *
oldQValue + LearningRate * (reward + DiscountRate
* table.getMaxQValue(state));
        table.setQValue(lastState,          lastAction,
newQValue);
      }
      lastState = state;
      lastAction = action;
    }


    public void learnSARSA(int state, int action,
double reward) {
                  if (first)
                          first = false;
                  else {
                        double oldQValue = table.getQValue(lastState, lastAction);

                        double newQValue = (1 - LearningRate) * oldQValue
                                                 + LearningRate * (reward + DiscountRate * table.getQValue(state, action));

                        table.setQValue(lastState, lastAction, newQValue);
                }
                lastState = state;
                lastAction = action;
        }

    public int selectAction(int state)
    {

                double thres = Math.random();

                int actionIndex = 0;

                if (thres<explorationRate)
                {//randomly select one action from action(0,1,2)
                        Random ran = new Random();
                        actionIndex = ran.nextInt(((Action.NumRobotActions-1 - 0) + 1));
                }
                else
                {//e-greedy
                actionIndex=table.getBestAction(state);
                }
                return actionIndex;
        }

}

package NNRobot;

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][NumHitByBullet];
    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);
  }

}
package rlRobotNew;


import java.awt.geom.*;

class Target
{
  String name;
  public double bearing;
  public double head;
  public long ctime;
  public double speed;
  public double x, y;
  public double distance;
  public double changehead;
  public double energy;

  Target(){
          bearing=2*(Math.random()-0.5);
          distance=1000;

  }
  public Point2D.Double guessPosition(long when)
```

```
    {
      double diff = when - ctime;
      double newY, newX;

      /**if the change in heading is significant, use circular targeting**/
      if (Math.abs(changehead) > 0.00001)
      {
        double radius = speed/changehead;
        double tothead = diff * changehead;
        newY = y + (Math.sin(head + tothead) * radius) - (Math.sin(head) * radius);
        newX = x + (Math.cos(head) * radius) - (Math.cos(head + tothead) * radius);
      }
      /**If the change in heading is insignificant, use linear**/
      else {
        newY = y + Math.cos(head) * speed * diff;
        newX = x + Math.sin(head) * speed * diff;
      }
      return new Point2D.Double(newX, newY);
    }

    public double guessX(long when)
    {
      long diff = when - ctime;
      System.out.println(diff);
      return x+Math.sin(head)*speed*diff;
    }
    public double guessY(long when)
    {
      long diff = when - ctime;
      return y+Math.cos(head)*speed*diff;
    }
  }
package rlRobotNew;


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;
}
package rlRobotNew;


import java.io.*;
import robocode.*;

public class QTable
```

```
{
  private double[][] table;


  public QTable()
  {
    table = new
double[State.NumStates][Action.NumRobotAction
s];
    initialize();
  }


  private void initialize()
  {
    for (int i = 0; i < State.NumStates; i++)
      for (int j = 0; j < Action.NumRobotActions; j++)
        table[i][j] = 0.0;
  }


  public double getMaxQValue(int state)
  {
    double maxinum = Double.NEGATIVE_INFINITY;
    for (int i = 0; i < table[state].length; i++)
    {
      if (table[state][i] > maxinum)
        maxinum = table[state][i];
    }
    return maxinum;
  }


  public int getBestAction(int state)
  {
    double maxinum = Double.NEGATIVE_INFINITY;
    int bestAction = 0;
    for (int i = 0; i < table[state].length; i++)
    {
      double qValue = table[state][i];
      //System.out.println("Action " + i + ": " + qValue);
      if (qValue > maxinum)
      {
        maxinum = qValue;
        bestAction = i;
      }
    }
    return bestAction;
  }


  public double getQValue(int state, int action)
  {
    return table[state][action];
  }


  public void setQValue(int state, int action, double
value)
  {
    table[state][action] = value;
  }


  public void loadData(File file)
  {
    BufferedReader r = null;
    try
    {
      r = new BufferedReader(new FileReader(file));
      for (int i = 0; i < State.NumStates; i++)
        for (int j = 0; j < Action.NumRobotActions; j++)
          table[i][j] = Double.parseDouble(r.readLine());
    }
    catch (IOException e)
    {
      System.out.println("IOException trying to open reader: " + e);
      initialize();
    }
    catch (NumberFormatException e)
    {
      initialize();
    }
    finally
    {
      try
      {
        if (r != null)
          r.close();
      }
      catch (IOException e)
      {
        System.out.println("IOException trying to close reader: " + e);
      }
    }
  }

  public void saveData(File file)
  {
    PrintStream w = null;
    try
    {
      w = new PrintStream(new RobocodeFileOutputStream(file));
      for (int i = 0; i < State.NumStates; i++)
        for (int j = 0; j < Action.NumRobotActions; j++)
          w.println(new Double(table[i][j]));


      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);
        }
      }
    }
  }
```