Assignment 3

Yuan Sun
48500904

***Question (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.***

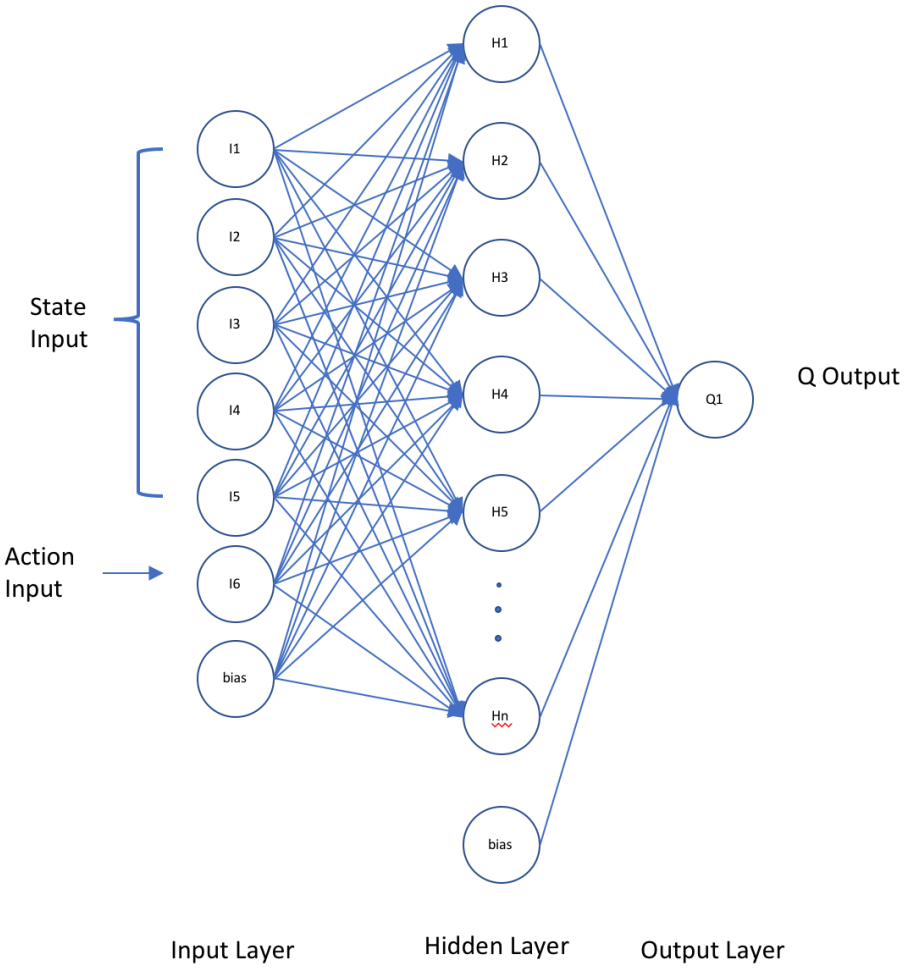


Figure 1 The architecture of neural network

The neural network is a single network with one single output and the actions are put in the one of the input neurons. The first input layer has 7 neurons, the first 5 are the state inputs and the $6^{th}$ neuron is the action input and the $7^{th}$ neuron is the bias. The states are number of

heading(4 possible values), number of target distance(10 possible values), number of target bearing(4 possible values), robot's X position(8 values), robot's Y position(6 possible values). The actions have 7 possible values, which are ahead, back, ahead left, ahead right, fire one, fire two, fire three. The inputs are zero centered and normalized to -1 to 1 and the output is also zero centered and normalized to -1 to 1. The number of hidden neurons are about 4 to 100, which I will also discuss in the next question.

***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.***

Table 1. RMS of different number of hidden neurons, learning rate = 0.01, momentum=0.3

| number of hidden neurons | 4 | 14 | 23 | 30 | 100 |
|---|---|---|---|---|---|
| RMS | 0.1278 | 0.1298 | 0.1220 | 0.1312 | 0.1347 |

Table 2. RMS of different learning rate, number of hidden neurons = 23, momentum=0.3

| learning rate | 0.002 | 0.005 | 0.01 | 0.08 | 0.2 |
|---|---|---|---|---|---|
| RMS | 0.1215 | 0.1219 | 0.1220 | 0.1342 | 0.1454 |

Finding the best paremeters:
In terms of hidden neurons, it's not the more hidden neurons, the better the RMS. From my experiment, there exists some optimal number of neurons that can represent our model well, such as 23 hidden neurons. Too few or too many hidden neurons will both result in a bit high RMS.

In terms of learning rate, it's true that the bigger the learning rate, the faster the convergence, but setting it too high might lead us to miss the optimal solution. What's more, if the learning rate is too small, it's too time consuming to train. As a result, we need to balance between speed and accuracy, so I chose a learning rate of 0.01.

In terms of momentum, I found that the differences between different values don't vary too much, so after several trails, I chose the momentum to be 0.3 .

The parameters that best learn my LUT data: number of input is 6, number of hidden neurons is 23, learning rate is 0.01, momentum is 0.3. The optimal RME is 0.1220. And the graph is shown below. You can see that the RMS has been gradually decreased using back propagation network in assignment1. After 10000 epochs, the RMS no longer decreases, so it has converged to the optimal solution.

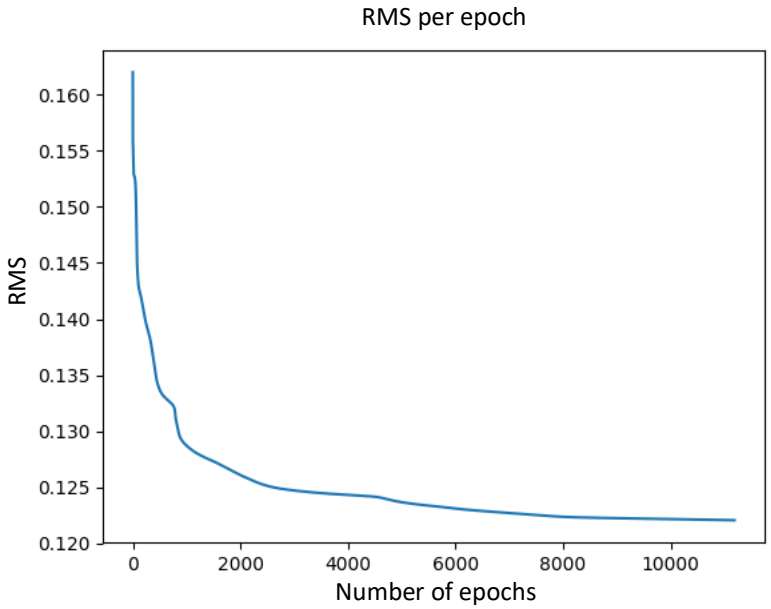


Figure 2 The decrease of RMS during each epoch
(parameters: number of input is 6, number of hidden neurons is 23,
learning rate is 0.01, momentum is 0.3)

***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.***

Table 3. Comparison of input representation

| | | LUT inputs | Neural net inputs |
|---|---|---|---|
| states | heading | [0,1,2,3], quantize the heading(0~360) to 4 | -1~1, zero center and normalize 0~360 double number |
| | targetDistance | [0,1,2,3,4,5,6,7,8,9], quantize the distance(0~1000) to 10 | -1~1, zero center and normalize 0~1000 double number |
| | targetBearing | [0,1,2,3], quantize the heading(-180~180) to 4 | -1~1, zero center and normalize -180~180 double number |
| | XPosition | [0,1,2,3,4,5,6,7], quantize the distance(0~800) to 8 | -1~1, zero center and normalize 0~800 double number |
| | YPosition | [0,1,2,3,4,5,], quantize the distance(0~1000) to 6 | -1~1, zero center and normalize 0~600 double number |
| actions | 7 actions | [0,1,2,3,4,5,6], 7 actions in total | [0,1,2,3,4,5,6], 7 actions in total |

Table 3 shows the difference of the input of my LUT and Neural net. In my robot in Part2, the input of LUT all have been quantized to reduce the dimension of the LUT. But in Part3, with the neural net, I used the original values shift to zero center and normalized to -1 to 1. So basically, the inputs for NN are numerical numbers(exception for the 7 discrete actions.)

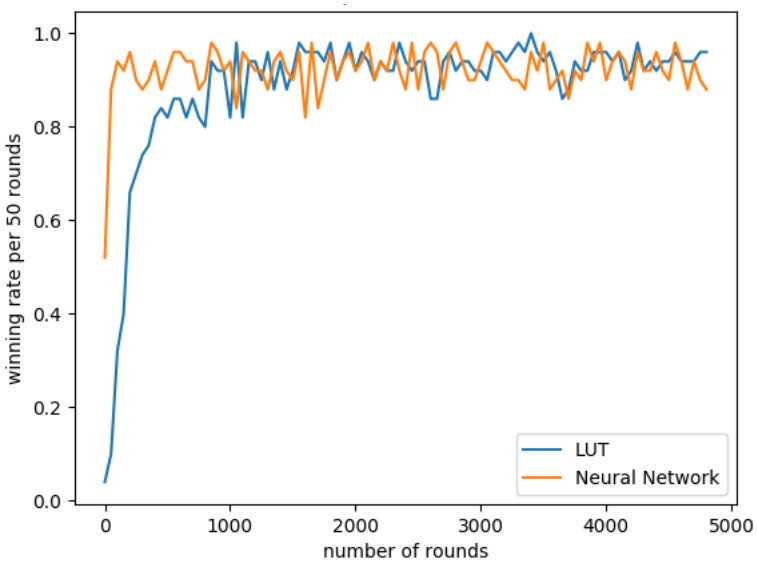


Figure 3 Performance comparison between LUT and NN

Figure 3 compares the result of my robot from Part2(LUT with state space reduction) and neural net based robot using no state space reduction. The opponent is the example Fire Robot. From the graph, we can see that both LUT and Neural Net can help increase the winning rate of the robot. In my case, Neural Net can improve the performance of the robot even faster. This might be because the number of weights of a neural net is less than that of a LUT, so it actually takes longer for the LUT to be updated to the final stable one, while the training of a neural net can take relatively less time. However, this is not a fixed conclusion, because the convergence time also depend on your opponent and the action you set up.

***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.***

The difference between a look up table and a neural network is that the former uses a table to store all the corresponding Q values, while the latter uses a neural network to compute the Q values.

A look up table stores all the states and actions pair's Q values, so the total amount of storage is the number of states times the number of actions. So this makes it hard for us to use numerical states, which will consume too much storage space. Besides, it will also make the look up table too big in dimension to learn the expected Q value of every state. So we need to conduct state space reduction.

A neural network, on the other hand, only requires us to store the weights, which will be a finite number of parameters and will be much less compared with numerical input state values. As a result, state space reduction will not be necessary.

***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.***

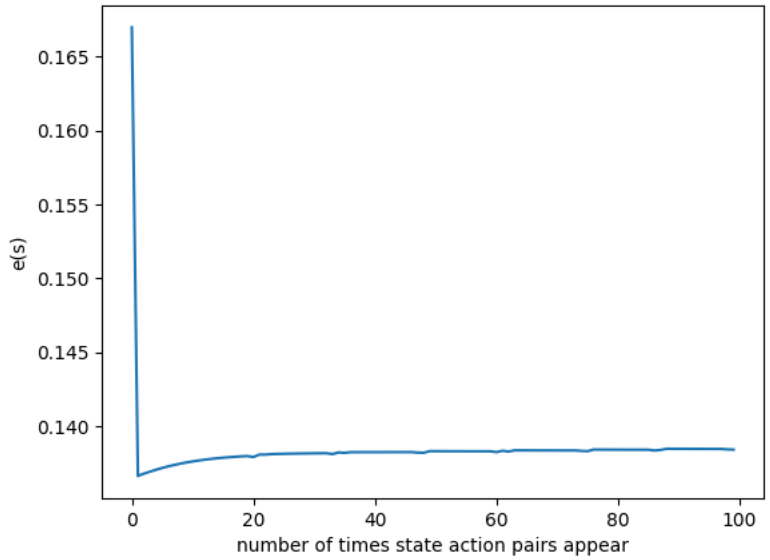


Figure 4 e(s) during the beginning of the battle

The best win rate observed for my tank is 95% against Fire, 72% against Tracker(more difficult to battle with). Figure 4 shows the e(s) between the state input [-0.1~0.1, -0.1~0.1, -0.1~0.1, -0.1~0.1, -0.1~0.1 ] and action 6(fire with the power of 3). Since I didn't use dimensionality reduction in my Neural Net, the state is a small range of inputs. In the graph, I only recorded e(s) when the state action pair occurs, and since the value will fluctuate along the battle, I chose the beginning part of data. And we can see that the error e(s) is getting smaller(although slightly getting bigger later but remains to be small compared to the initial value) with the battle going on, which means that the robot is learning.

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

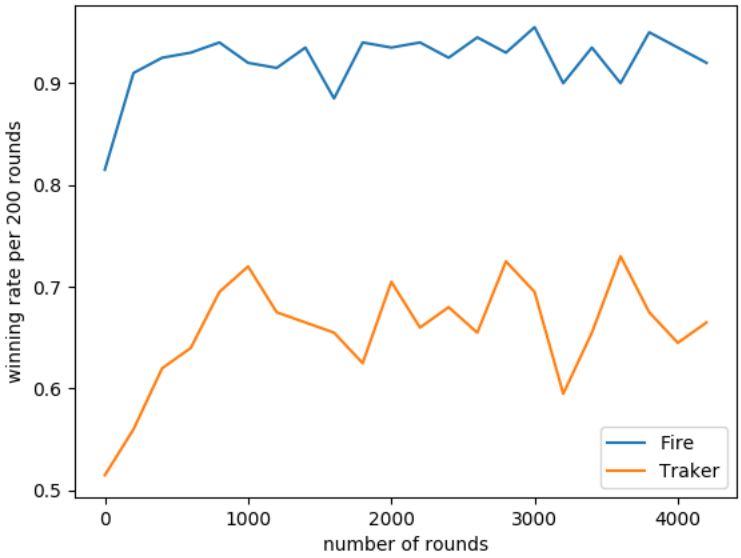


Figure 5 Win rate vs number of rounds(with two opponents: Fire and Tracker)

Figure 5 depicts the winning rate of the robot using NN, against two opponents, Fire and Tracker. Both of the two lines improves asymptotically, with a low win rate in the beginning and gradually increase in the end. The performance for Fire is better and increases faster compared to that of Tracker. This is due to the fact that Fire is relatively easier to battle with than Tracker. Therefore it might happen that when facing a difficult opponent, the win rate is hard to increase beyond a certain percentage.

***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.***

From the lecture slides, the Bellman equation is expressed as:

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

Where $V(S_t)$ and $V(S_{t+1})$are the value functions at time $t$ and $t+1$, respectively. $r_t$ is the reward at time $t$ and $\gamma$ is the discount factor.

We denote the optimal value function at time $t$ as $V^*(S_t)$, so the error in the value function is:

$$e(S_t) = V(S_t) - V^*(S_t) \quad (2)$$

Similarly,

$$e(S_{t+1}) = V(S_{t+1}) - V^*(S_{t+1}) \quad (3)$$

And,

$$V^*(S_t) = r_t + \gamma V^*(S_{t+1}) \quad (4)$$

Combining equation (1)(2) (3)(4), we get:

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

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

Therefore, the error in successive states are related. If $e(S_T)$ denotes the terminal error, $e(S_t) = \gamma^n\ e(S_T)$, $n$ is the number of difference time states of the terminal state to the current state $t$. Eventually, $e(S_T)$ will be zero, which means that $e(S_t)$ will be zero as well, so the TD learning algorithm converges.

However, when the Q-function is approximated. We introduced some approximation error, so

$$e(S_t) + e_{aprox} = V(S_t) - V^*(S_t).$$

Therefore,

$$\begin{aligned} e(S_t) &= r_t + \gamma V(S_{t+1}) - e_{aprox\_t} - \left(r_t + \gamma V^*(S_{t+1})\right) \\ &= \gamma(V(S_{t+1}) - V^*(S_{t+1}))\ - e_{aprox\_t} \\ &= \gamma\ (e(S_{t+1}) - e_{aprox\_t+1}) - e_{aprox\_t} \end{aligned}$$

Where $e_{aprox\_t}$ denotes the approximation error at time $t$.
Therefore, when the terminal error $e(S_T)$ is zeros, the state error $e(S_t)$ will not be zero. Hence convergence is not guaranteed.

***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: Readup on experience replay.***

In the case of using the NN for online training of the Q-function, we can introduce the experience replay method and calculate the error for each mini batch of experience. To illustrate, this is done by first storing the robot's states and action input set and the corresponding Q value in a replay memory. Each time the robot conducts the best action($A_t$) using the highest Q value in the NN, and gets the reward($R_{t+1}$) and enters into a new state($S_{t+1}$). We add the tuple ($S_t,\ A_t, R_{t+1}, S_{t+1}$) into the replay memory[1]. And each time, we use a mini batch of the replay memory to update multiple Q-values and also compute the error of the mini batch. In this way, we can obtain the error for our NN.

***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?***

Practical issues surrounding the application of RL & BP to my problem: Both RL & BP can help improve the performance of the robot. But due to the limitation of number of actions, when facing difficult opponent, the win rate is hard to reach very high level. What's more, due to the always changing environment, the win rate is fluctuating and maybe some other controlling methods can be applied.

To improve the performance of my robot, I think I can make more actions. For example, we can set more ahead distance or turn degrees to allow the robot to conduct more actions. Right now, the robot seems to perform the basic actions, but hopefully when having more possible actions, it can learn, on its own, some fancy combination of simple actions, like swirl and fire or fire and duck.

Convergence: Convergence will happen eventually if the parameters are suitable. Therefore it is crucial to do off line training and adjust the parameters with experiments. What's more, the rewards can also have an effect on convergence. To make it converge, we can check to see our rewards are appropriate.

Practical applications: First of all, checking your code and doing unit code test is important. Second, finding a good way of measuring whether your network is working or not is also important. Third, set good reward and set good parameters, since they are critical in a RL model.

***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.***

Concerns:

1) We can't afford negative results. A control system is different from that of a robocode game. The immediate negative feedback in each step might do harm to a patient's body, what's more we can't afford the terminal negative feedback, which might cause someone's life. Hence the level of importance is different in this case and requires us to be extra careful.
2) The control system is more complex. A control system is not as straight-forward as the rules in a game. Sometimes, the combination of several of the patient's feedback might have a certain meaning and requires an action to be done. What's more, a surgery environment is dynamic with unexpected happening a lot of times. Therefore it's much harder to come up with a reward scheme like the robocode to decide which we need to give positive reward to and which to give negative reward to. Sometimes, the scheme relies on doctors' experience and knowledge. Other than that, we can use big data to train the model offline but that leads to my third concern.
3) Simply using the medical big data is not enough. Unlike robocode that we trained a neural net offline using the LUT, a medical system requires more than that. On the one hand, it's hard to obtain very comprehensive and reliable data on previous patients' cases. On the other hand, even if we managed to get the data, simply train the data offline and then using the network is too risky. The data are just numbers and can't see the nature of problems.

Variations:

1) We can first gather as many data as we can, and train the offline network. Next, we do some simulation surgeries, and ask expert doctor to give opinion on the performance of the control system. This monitoring and evaluation process is necessary and essential, in that it add experts' opinion into the model and the simulation surgeries can improve the reliability of the system.
2) Add more risk control into the model, e.g. add some measurements that monitors the risk of each action. Therefore besides each negative rewards, we add risk of each action as negative rewards to closely pay attention to any possible risks.

***Reference:***

[1] Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto. Second Edition, MIT Press, Cambridge, MA, 2018

Appendix:

**updateRobotNN .java**

```
package a2;
import java.awt.*;
import java.awt.geom.*;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Random;


import robocode.*;

public class updateRobotNN extends AdvancedRobot
{
	public static final double PI = Math.PI;
	private Target target;
	private LUT table;
	private NeuralNet NN;
	private Learner learner;
	private boolean found=false;
	private double reward = 0.0;
	private double firePower;
	private int direction = 1;
	private int isHitWall = 0;
	private int isHitByBullet = 0;
	private int winFlag = 0;
	private static int countForWin=0;
	double rewardForWin=100;//100
	double rewardForDeath=-20;//-20
	double accumuReward=0.0;
```

```
private static int count=0;
private int lastAction=4;
private double[] NN_last_states = new double[5];
private boolean NNFlag=true;
private int action;
private static final double NN_alpha = 0.3;
private double NN_last_action;
private double NN_lambda=0.9;
private double NN_epsilon=0;
double [] NN_current_states;

public void run()
{
	target = new Target();
	target.distance = 1000;
	if(NNFlag==false){
		table = new LUT();
		loadData();
		learner = new Learner(table);
	}
	else{
		NN=new NeuralNet(6, 23, 0.02, 0.2, -1, 1);
		System.out.println("NNNNNNNNNNNNNNNNN6, 12, 0.001, 0.8, -1, 1");
		//NN.initializeWeights();
		loadWeights();
		action=selectActionNN(getState_numerical(), reward);
	}

	setColors(Color.green, Color.white, Color.green);
	setAdjustGunForRobotTurn(true);
	setAdjustRadarForGunTurn(true);
	setAdjustRadarForRobotTurn(true);
	turnRadarRightRadians(2 * PI);
	while (true)
	{
		if(getRoundNum()<200)//
			learner.explorationRate=0.0;
		else
			learner.explorationRate=0.0;
		robotMovement();
		execute();
	}
}
```

```
private void robotMovement()
{
	int state = getState();
	reward = 0.0;
	if(NNFlag==false){
		action = learner.selectAction(state);
		learner.learn(state, action, reward);
	}
	else {
		action=selectActionNN(getState_numerical(), reward);
	}

	switch (action)
	{
	case Action.ahead:
		setAhead(Action.aheadDistance);
		break;
	case Action.back:
		setBack(Action.aheadDistance);
		break;
	case Action.aheadLeft:
		setAhead(Action.aheadDistance);
		setTurnLeft(Action.turnDegree);
		break;
	case Action.aheadRight:
		setAhead(Action.aheadDistance);
		setTurnRight(Action.turnDegree);
		break;
	case Action.fireOne:
		findTargetFire();
		if (getGunHeat() == 0) {
			setFire(1);
		}
		break;
	case Action.fireTwo:
		findTargetFire();
		if (getGunHeat() == 0) {
			setFire(2);
		}
		break;
	case Action.fireThree:
		findTargetFire();
		if (getGunHeat() == 0) {
```

```
                        setFire(3);
                    }
                    break;
                }
    }
    private void findTargetFire() {
        found=false;
        while(!found) {
            setTurnRadarLeft(360);
            execute();
        }
        double gunOffset=(getGunHeading()-getHeading())/360*2*PI-target.bearing;
        setTurnGunLeftRadians(NormaliseBearing(gunOffset));
        execute();
    }


    private int getState()
    {
        int heading = State.getHeading(getHeading());
        int XPosition = State.getXPosition(getX());
        int YPosition = State.getYPosition(getY());
        int targetDistance = State.getTargetDistance(target.distance);
        int targetBearing = State.getTargetBearing(target.bearing);
        int energy = State.getEnergy(getEnergy());

        int state =
State.Mapping[heading][targetDistance][targetBearing][XPosition][YPosition];   //[isHitWall]
        return state;
    }
    private double[] getState_numerical()
    {
        double heading = getHeading();
        double XPosition = getX();
        double YPosition = getY();
        double targetDistance = target.distance;
        double targetBearing = target.bearing;
        double[] state= {heading,targetDistance,targetBearing,XPosition,YPosition,};

        return state;
    }

    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);
                if(NNFlag==false) {
                        int state = getState();
                        int action = learner.selectAction(state);
                        learner.learn(state, action, change);
                }
                else {
                        action=selectActionNN(getState_numerical(), reward);
                }
        }
}


public void onBulletMissed(BulletMissedEvent e)
{
        double change = -e.getBullet().getPower();
        out.println("Bullet Missed: " + change);
        if(NNFlag==false) {
                int state = getState();
                int action = learner.selectAction(state);
                learner.learn(state, action, change);
        }
        else {
                action=selectActionNN(getState_numerical(), reward);
        }
}

public void onHitByBullet(HitByBulletEvent e)
{
        double power = e.getBullet().getPower();
```

```
		double change = -(4 * power + 2 * (power - 1));
		out.println("Hit By Bullet: " + change);
		if(NNFlag==false) {
			int state = getState();
			int action = learner.selectAction(state);
			learner.learn(state, action, change);
		}
		else {
			action=selectActionNN(getState_numerical(), reward);
		}
}

public void onHitRobot(HitRobotEvent e)
{
		double change = -6.0;
		out.println("Hit Robot: " + change);
		if(NNFlag==false) {
			int state = getState();
			int action = learner.selectAction(state);
			learner.learn(state, action, change);
		}
		else {
			action=selectActionNN(getState_numerical(), reward);
		}
}

public void onHitWall(HitWallEvent e)
{
		double change = -(Math.abs(getVelocity()) * 0.5 );
		out.println("Hit Wall: " + change);
		if(NNFlag==false) {
			int state = getState();
			int action = learner.selectAction(state);
			learner.learn(state, action, change);
		}
		else {
			action=selectActionNN(getState_numerical(), reward);
		}
}

public void onScannedRobot(ScannedRobotEvent e)
{
		if ((e.getDistance() < target.distance)||(target.name == e.getName()))
		{
```

```
                found=true;
                //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)
        {
            winFlag=1;
            File file = getDataFile("accumReward1.dat");
            if(NNFlag==false) {
                int state = getState();
                int action = learner.selectAction(state);
                learner.learn(state, action, rewardForWin);
                saveData();
            }
            else {
```

```
                action=selectActionNN(getState_numerical(), rewardForWin);
                saveWeights();
            }
            saveResult2(winFlag);
        }

        public void saveResult2(int winFlag) {
            if(true) {
                File file = getDataFile("saveResult1.dat");
                PrintStream w = null;
                try
                {
                    w = new PrintStream(new
RobocodeFileOutputStream(file.getAbsolutePath(), true));
                    w.println(winFlag);
                    if (w.checkError())
                        System.out.println("Could not save the data!");
//setTurnLeft(180 - (target.bearing + 90 - 30));
                    w.close();
                }

                catch (IOException e1)
                {
                    System.out.println("IOException trying to write: " + e1);
                }
                finally
                {
                    try
                    {
                        if (w != null)
                            w.close();
                    }
                    catch (Exception e2)
                    {
                        System.out.println("Exception trying to close witer: " + e2);
                    }
                }
            }
        }
        public void onDeath(DeathEvent event)
        {
            winFlag=0;
            accumuReward+=rewardForDeath;
            count++;
```

```
        if(NNFlag==false) {
                int state = getState();
                int action = learner.selectAction(state);
                learner.learn(state, action, rewardForDeath);
                saveData();
        }
        else {
                action=selectActionNN(getState_numerical(), rewardForDeath);
                saveWeights();
        }
        saveResult2(winFlag);
}

public void loadData()
{
        try
        {
                table.load(getDataFile("movement1.dat"));
        }
        catch (Exception e)
        {
                out.println("Exception trying to load: " + e);
        }
}

public void saveData()
{
        try
        {
                table.save(getDataFile("movement1.dat"));
        }
        catch (Exception e)
        {
                out.println("Exception trying to write: " + e);
        }
}
public void loadWeights()
{
        try
        {
                NN.loadWeights(getDataFile("weight.dat"));
        }
        catch (Exception e)
        {
```

```
                out.println("Exception trying to load: " + e);
                NN.initializeWeights();
            }
        }

        public void saveWeights()
        {
            try
            {
                NN.saveWeights(getDataFile("weight.dat"));
                out.println("saaaaaaaaaaaaaaaaaaavingWeights");

            }
            catch (Exception e)
            {
                out.println("Exception trying to write: " + e);
            }
        }
        private void initialNN(){
            double ErrorBound=0.05,totalError,RMS=0.0;
            double []saveTotalError = null;
            double[][] x=
{{0,1,2,3},{0,1,2,3,4,5,6,7,8,9},{0,1,2,3},{0,1,2,3,4,5,6,7},{0,1,2,3,4,5},{0,1,2,3,4,5,6}};
            double [][] x_input= getInput(x);
            double [][] X_input_NN=normalize(x);

        //loadData("/Users/anna321321/E/CPEN502/hw/a2/bin/a2/updateRobot.data/movem
ent1.dat");
            double mean=85;
            double max=150;
            double min=-20;


            //File writename=new File("output.bat");
            //boolean createNewFile =writename.createNewFile();
            //BufferedWriter write=new BufferedWriter(new FileWriter(writename));
            int avgEpoch=0;
            for(int trials=0;trials<1;trials++) {
                //write.write("Start trial="+String.valueOf(trials)+":\r\n");
                NN=new NeuralNet(6, 12, 0.001, 0.8, -1, 1);
                NN.initializeWeights();
                int epoch=0;
                for(epoch=0;epoch<1;epoch++) {
                    totalError=0;
```

```
                                for(int i=0;i<x_input.length;i++) {
                                        for(int j=0;j<NN.argNumInputs;j++) {

        //totalError+=NN.train(xBinary[i],yBinary[i],"binary");
                                                int state =State.Mapping[(int)
x_input[i][0]][(int)x_input[i][1]][(int)x_input[i][2]][(int)x_input[i][3]][(int)x_input[i][4]]; //
x_input[i][1] x_input[i][2] ;//;
                                                int action=(int)x_input[i][x.length-1];

        totalError+=NN.train(X_input_NN[i],(table.table[state][action]-mean)*2/(max-
min),"bipolar");
                                        }
                                        RMS=Math.sqrt(totalError/x_input.length);
                                }
                                //System.out.println("epoch="+epoch+", error="+RMS);
                                //write.write(String.valueOf(RMS)+"\r\n");
//                              if(totalError<ErrorBound||epoch>40000) {
//                                      System.out.println("error="+totalError);

//                                      break;
//                              }
                                //write.flush();
                        }
                        //write.write("End trail="+trials+".\r\n");
                        avgEpoch+=epoch;
                }
                avgEpoch=avgEpoch/1000;
                System.out.println("avgEpoch="+avgEpoch);
//              write.write("avgEpoch="+avgEpoch);
//              write.flush();
//              write.close();
        }
        private static double [][] getInput(double [][] x){
                double [][] table=new double[53760][6]; //=
                int i=0;
                for (int a=0;a<x[0].length;a++) {
                        for(int b=0;b<x[1].length;b++) {
                                for(int c=0;c<x[2].length;c++) {
                                        for(int d=0;d<x[3].length;d++) {
                                                for(int e=0;e<x[4].length;e++) {
                                                        for(int f=0;f<x[5].length;f++) {
                                                                table[i]= new double[]
{x[0][a],x[1][b],x[2][c],x[3][d],x[4][e],x[5][f]};
                                                                i++;
```

```
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                            return table;
                    }
                    private static double [][] normalize(double [][] x){
                            double [][] table=new double[53760][6]; //=
                            int i=0;
                            for (int a=0;a<x[0].length;a++) {
                                    for(int b=0;b<x[1].length;b++) {
                                            for(int c=0;c<x[2].length;c++) {
                                                    for(int d=0;d<x[3].length;d++) {
                                                            for(int e=0;e<x[4].length;e++) {
                                                                    for(int f=0;f<x[5].length;f++) {
                                                                            table[i]= new double[] {(x[0][a]-
1.5)/1.5,(x[1][b]-4.5)/4.5,(x[2][c]-1.5)/1.5,(x[3][d]-3.5)/3.5,(x[4][e]-2.5)/2.5,(x[5][f]-3)/3};
                                                                            i++;
                                                                    }
                                                            }
                                                    }
                                            }
                                    }
                            }
                            return table;
                    }
                    public void loadData(String path)
                    {
                            File file=new File(path);
                            try
                            {
                                    table.load(file);
                            }
                            catch (Exception e)
                            {
                            }
                    }
                    public int selectActionNN(double [] state,double reward){
                            state=normalizeState(state);
                            int action=4;
                            double qmax=Double.NEGATIVE_INFINITY;
                            for(int i=0;i<7;i++){
```

```
                        double q=NN.forward_with_action(state,(double)i,"bipolar");
                        if (q>qmax){
                                qmax=q;
                                action=i;
                        }
                }
                double
NN_Q_new=NN.forward_with_action(state,(double)action,"bipolar");//myNet[action].outputFo
r(NN_current_states);
        double error_signal = 0;
        double
old_Q=NN.forward_with_action(NN_last_states,(double)NN_last_action,"bipolar");
        error_signal = NN_alpha*((reward-85)/170 + NN_lambda * NN_Q_new - old_Q);
//myNet[NN_last_action].outputFor(NN_last_states));


                //newRobot1.total_error_in_one_round += error_signal*error_signal/2;
                double correct_old_Q = old_Q + error_signal;
                if((action==6)&&(state[1]<0.1&&state[1]>-0.1)&&(state[0]<0.1&&state[0]>-
0.1)&&(state[2]<0.1&&state[2]>-0.1)) {
                        saveActionStatePairError(error_signal);
                }
                NN.train_with_action(NN_last_states,(double)NN_last_action,(correct_old_Q-
85)/170,"bipolar");//myNet[NN_last_action].train(NN_last_states, correct_old_Q);
                if((Math.random() < NN_epsilon) && (getRoundNum()<1000) )
                {
                  action = new Random().nextInt(Action.numActions);
                }
                for(int i=0; i<5; i++)
                {
                  NN_last_states[i] = state[i];
                }
                NN_last_action=action;
                //out.println("action is
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSs"+action);
                return action;
        }
        public double [] normalizeState(double[] state){
                state= new double[] {(state[0]-180)/180,(state[1]-
500)/500,(state[2])/180,(state[3]-400)/400,(state[4]-300)/300};

                return state;
        }
        public void saveActionStatePairError(double error) {
```

```
                if(true) {
                        File file = getDataFile("ActionStatePairError.dat");
                        PrintStream w = null;
                        try
                        {
                                w = new PrintStream(new
RobocodeFileOutputStream(file.getAbsolutePath(), true));
                                w.println(error);
                                if (w.checkError())
                                        System.out.println("Could not save the data!");
//setTurnLeft(180 - (target.bearing + 90 - 30));
                                w.close();
                        }

                        catch (IOException e1)
                        {
                                System.out.println("IOException trying to write: " + e1);
                        }
                        finally
                        {
                                try
                                {
                                        if (w != null)
                                                w.close();
                                }
                                catch (Exception e2)
                                {
                                        System.out.println("Exception trying to close witer: " + e2);
                                }
                        }
                }
        }
}
```

**trainNN.java**

```
package a2;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
```

```
import java.util.Arrays;
//import org.jfree.chart.JFreeChart;

public class trainNN {
	private  static LUT table=new LUT();
	public static void main(String[]args) throws IOException{
		double ErrorBound=0.05,totalError,RMS=0.0;
//		int table[][]=new int [4][2];
//		int[] yBinary= {0,1,1,0};
//		int[] yBipolar= {-1,1,1,-1};
		double []saveTotalError = null;
//		double[][] xBinary= {{0,0},{0,1},{1,0},{1,1}};
//		double[][] xBipolar= {{-1,-1},{-1,1},{1,-1},{1,1}};
		double[][] x=
{{0,1,2,3},{0,1,2,3,4,5,6,7,8,9},{0,1,2,3},{0,1,2,3,4,5,6,7},{0,1,2,3,4,5},{0,1,2,3,4,5,6}};//{-1,1},{-
1,1},{-1,1},{-1,1},{-1,1},{-1,1},{-1,1}
		double [][] x_input= getInput(x);
		double [][] X_input_NN=normalize(x);

	loadData("/Users/anna321321/E/CPEN502/hw/backup/updateRobot.data/movement1.
dat");
		double mean=85;
		double max=150;
		double min=-20;



		int epochmin=10000;
		int minEpoch=10000,maxEpoch=0;

		File writename=new File("output.txt");
		boolean createNewFile =writename.createNewFile();
		BufferedWriter write=new BufferedWriter(new FileWriter(writename));


		int avgEpoch=0;
		for(int trials=0;trials<1;trials++) {
			write.write("Start trial="+String.valueOf(trials)+":\r\n");
			NeuralNet NN=new NeuralNet(6, 4, 0.01, 0.1, -1, 1);		//, 23, 0.01,
0.3, -1, 1
			NN.initializeWeights();
			int epoch=0;
			for(epoch=0;;epoch++) {
				totalError=0;
```

```
                                for(int i=0;i<x_input.length;i++) {
                                        for(int j=0;j<NN.argNumInputs;j++) {

        //totalError+=NN.train(xBinary[i],yBinary[i],"binary");
                                                int state =State.Mapping[(int)
x_input[i][0]][(int)x_input[i][1]][(int)x_input[i][2]][(int)x_input[i][3]][(int)x_input[i][4]]; //
x_input[i][1] x_input[i][2] ;//;
                                                int action=(int)x_input[i][x.length-1];

        totalError+=NN.train(X_input_NN[i],(table.table[state][action]-mean)*2/(max-
min),"bipolar");
                                        }
                                        RMS=Math.sqrt(totalError/x_input.length);
                                }
                                System.out.println("epoch="+epoch+", error="+RMS);
                                File weightname=new File("weight.txt");
                                boolean createNewWeightFile =weightname.createNewFile();
                                BufferedWriter weight=new BufferedWriter(new
FileWriter(weightname));
                                for(int i=0;i<NN.weight.length;i++)
                                        for(int j=0;j<NN.weight[i].length;j++)
                                                for(int k=0;k<NN.weight[i][j].length;k++){

        weight.write(String.valueOf(NN.weight[i][j][k])+"\r\n");
                                        }
                                write.write(String.valueOf(RMS)+"\r\n");

                                if(totalError<ErrorBound||epoch>20000) {
                                        System.out.println("error="+totalError);

                                        break;
                                }
                                write.flush();
                                weight.flush();
                                weight.close();
                        }
                        write.write("End trail="+trials+".\r\n");
                        avgEpoch+=epoch;
                }
                avgEpoch=avgEpoch/1000;
                System.out.println("avgEpoch="+avgEpoch);
                write.write("avgEpoch="+avgEpoch);
                //totalError=0;
                /*
```

```
                    if (epoch >= 10000) {
          ++counter;
        } else {
          counter2 += epoch;
          if (epoch < countermin) {
            countermin = epoch;
          }
          if (epoch > countermax) {
            countermax = epoch;
          }
        }*/
                    write.flush();
                    write.close();
          }
//        private int getState()
//        {
//                  int heading = State.getHeading(getHeading());
//                  int XPosition = State.getXPosition(getX());
//                  int YPosition = State.getYPosition(getY());
//                  int targetDistance = State.getTargetDistance(target.distance);
//                  int targetBearing = State.getTargetBearing(target.bearing);
//                  int energy = State.getEnergy(getEnergy());
//
//                  int state =
State.Mapping[heading][targetDistance][targetBearing][XPosition][YPosition];   //[isHitWall]
//                  return state;
//        }
          private static double [][] getInput(double [][] x){
                    double [][] table=new double[53760][6]; //=
                    int i=0;
                    for (int a=0;a<x[0].length;a++) {
                              for(int b=0;b<x[1].length;b++) {
                                        for(int c=0;c<x[2].length;c++) {
                                                  for(int d=0;d<x[3].length;d++) {
                                                            for(int e=0;e<x[4].length;e++) {
                                                                      for(int f=0;f<x[5].length;f++) {
                                                                                table[i]= new double[]
{x[0][a],x[1][b],x[2][c],x[3][d],x[4][e],x[5][f]};
                                                                                i++;
                                                                      }
                                                            }
                                                  }
                                        }
                              }
```

```
		}
		return table;
	}
	private static double [][] normalize(double [][] x){
		double [][] table=new double[53760][6]; //=
		int i=0;
		for (int a=0;a<x[0].length;a++) {
			for(int b=0;b<x[1].length;b++) {
				for(int c=0;c<x[2].length;c++) {
					for(int d=0;d<x[3].length;d++) {
						for(int e=0;e<x[4].length;e++) {
							for(int f=0;f<x[5].length;f++) {
								table[i]= new double[] {(x[0][a]-
1.5)/1.5,(x[1][b]-4.5)/4.5,(x[2][c]-1.5)/1.5,(x[3][d]-3.5)/3.5,(x[4][e]-2.5)/2.5,(x[5][f]-3)};
								i++;
							}
						}
					}
				}
			}
		}
		return table;
	}

	public static void loadData(String path)
	{
		File file=new File(path);
//	if(!file.exists()||file.isDirectory())
//	  throw new FileNotFoundException();
//	BufferedReader br=new BufferedReader(new FileReader(file));
		try
		{
			table.load(file);
		}
		catch (Exception e)
		{
		}
	}
}
```

**NeuralNet.java**

```
package a2;

import java.io.BufferedReader;
```

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Arrays;
import java.util.Random;

import robocode.RobocodeFileOutputStream;

public class NeuralNet implements NeuralNetInterface{
	int argNumInputs;
	int argNumHidden;
	double argLearningRate;
	double argMomentumTerm;
	double argA;
	double argB;
	double[][][]weight;
	double [][][] weightChange;
	double[] input;
	double []h;
	double []delta;

	public NeuralNet (
			int argNumInputs,
			int argNumHidden,
			double argLearningRate,
			double argMomentumTerm,
			double argA,
			double argB ) {
		this.argNumInputs=argNumInputs;
		this.argNumHidden=argNumHidden;
		this.argLearningRate=argLearningRate;
		this.argMomentumTerm=argMomentumTerm;
		this.argA=argA;
		this.argB=argB;
		this.weight=new double[2][][];
		this.weight[0]=new double[argNumInputs+1][argNumHidden+1];
		this.weight[1]=new double[argNumHidden+1][1];
		this.weightChange=new double[2][][];
		this.weightChange[0]=new double[argNumInputs+1][argNumHidden+1];
		this.weightChange[1]=new double[argNumHidden+1][1];

		this.h =new double[argNumHidden+1];
		this.delta=new double[argNumHidden+1];
```

```
	}

	public double outputFor(double[] X) {
		return 0;
	}

	public double train(double[] X, double argValue,String type) {
		double yhat=forward( X,type);
		if(type=="binary") {
			//yhat=sigmoid(sum);
			for(int i=0;i<argNumHidden+1;i++) {
				delta[i]=yhat*(1-yhat)*(argValue-yhat);

	weight[1][i][0]+=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChange[1][i]
[0];

	weightChange[1][i][0]=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChang
e[1][i][0];
				System.out.println(weightChange[1][i][0]);
			}
			for(int i=0;i<argNumInputs+1;i++) {
				for(int j=0;j<argNumHidden+1;j++) {
					weight[0][i][j]+=argLearningRate*h[j]*(1-
h[j])*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];
					weightChange[0][i][j]=argLearningRate*h[j]*(1-
h[j])*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];
				}
			}
		}
		else {
			//yhat=customSigmoid(sum);//customSigmoid
			for(int i=0;i<argNumHidden+1;i++) {
				delta[i]=(argB-argA)*0.25*(1-Math.pow(yhat,2))*(argValue-yhat);
//0.5*(1-Math.pow(yhat,2))*(argValue-yhat);
				//System.out.println("delta[ "+i+"]:"+delta[i]);

	weight[1][i][0]+=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChange[1][i]
[0];

	weightChange[1][i][0]=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChang
e[1][i][0];
			}
			for(int i=0;i<argNumInputs+1;i++) {
				for(int j=0;j<argNumHidden;j++) {
```

```
                                        weight[0][i][j]+=argLearningRate*(argB-argA)*0.25*(1-
Math.pow(h[j],2))*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];

        //System.out.println("weight[0]["+i+"]["+j+"]:"+argLearningRate*0.5*(1-
Math.pow(h[j],2))*delta[j]*weight[1][j][0]*input[i]);
                                        weightChange[0][i][j]=argLearningRate*(argB-
argA)*0.25*(1-
Math.pow(h[j],2))*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];

        //System.out.println("weightChange[0]["+i+"]["+j+"]:"+argLearningRate*0.5*(1-
Math.pow(h[j],2))*delta[j]*weight[1][j][0]*input[i]);
                                }
                        }
                }
                /*
                for(int i=0;i<argNumInputs;i++) {
                        System.out.println(Arrays.toString(weightChange[0][i]));
                }
                for(int i=0;i<argNumHidden;i++) {
                        System.out.println(Arrays.toString(weightChange[1][i]));
                }*/
                return Math.pow(argValue-yhat, 2);//0.5*
        }

        public double forward(double[] X, String type){
                //double []h =new double[argNumHidden+1],delta=new
double[argNumHidden+1];

                double yhat=0,error=0;
                input = Arrays.copyOf(X, X.length+1);//数组扩容
                input[input.length-1]=bias;
                for(int j=0;j<this.argNumHidden;j++) {
                        for(int i=0;i<this.argNumInputs+1;i++) {
                                h[j]+=input[i]*weight[0][i][j];
                        }
                        if(type=="binary")
                                h[j]=sigmoid(h[j]);
                        else
                                h[j]=customSigmoid(h[j]);//customSigmoid
                }
                h[argNumHidden]=bias;
                double sum=0;
                for(int i=0;i<argNumHidden+1;i++) {
                        sum+=h[i]*weight[1][i][0];
```

```
		}
		return customSigmoid(sum);
	}

	public void save(File argFile) {
		// TODO Auto-generated method stub
	}


	public void load(String argFileName) throws IOException {
		// TODO Auto-generated method stub
	}


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

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

	public void initializeWeights() {
		for(int i=0;i<=this.argNumInputs;i++) {
			for(int j=0;j<=this.argNumHidden;j++) {
				weight[0][i][j]=Math.random()*2-1;
			}
		}
		for(int i=0;i<=this.argNumHidden;i++) {
			weight[1][i][0]=Math.random()*2-1;//Math.random()-0.5;
		}
		System.out.println("INNNNNNNNNNNNNNNNitalizing weights");
	}
	public void loadWeights(File file) {
		if(!file.exists())
			initializeWeights();

		BufferedReader read = null;
```

```
		try{
			read = new BufferedReader(new FileReader(file));
			for(int i=0;i<weight.length;i++)
				for(int j=0;j<weight[i].length;j++)
					for(int k=0;k<weight[i][j].length;k++){
						weight[i][j][k] =
Double.parseDouble(read.readLine());
					}
		}
		catch (IOException e){
			System.out.println("loadWeights. IOException trying to open reader: " +
e);
		}
		catch (NumberFormatException e){
		}
		finally{
			try{
				if (read != null)
					read.close();
			}
			catch (IOException e)
			{
				 System.out.println("IOException trying to close reader: " + e);
			}
		}
	}
	public void saveWeights(File file) {
		int i,j=0,count=0;
		PrintStream write = null;
		try{
			write = new PrintStream(new RobocodeFileOutputStream(file));
			count++;
			System.out.println("count is: "+count);
			for ( i = 0; i < weight.length; i++)
				for (j = 0; j < weight[i].length; j++)
					for(int k=0;k<weight[i][j].length;k++){
						write.println(weight[i][j][k]); //write.println(new
Double(table[i][j]));
					}
			System.out.println("i is"+i+"j is "+j);
			if (write.checkError())
				System.out.println("Could not save the data!");
			write.close();
		}
```

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

    public void zeroWeights() {
            // TODO Auto-generated method stub
    }

    double forward_with_action(double[] state, double i, String string) {
            double[] input = Arrays.copyOf(state, state.length+1);//数组扩容
            input[input.length-1]=(i-3)/3;
            return forward(input,string);
    }
    double train_with_action(double[] state, double i,double argY, String string) {
            double[] input = Arrays.copyOf(state, state.length+1);//数组扩容
            input[input.length-1]=i;
            return train(input,argY,string);
    }
    //getHeading()/180-1,target.distance/500-1,target.bearing/180-1, reward);
}
```