EECE 592 Project Part 3: Reinforcement Learning with Backpropagation (December 2010)

Alison Michan 79172995, UBC M.A.Sc. Student
EECE 592 Architectures for Learning Systems, Dr. Sarbjit Sarkaria

***Abstract*— A Robocode robot was trained using Q-learning and a neural network. As a first step, a neural network was trained using lookup table data obtained in part 2 of the project. Next, the neural network replaced the lookup table in Q-learning. Finally, state-space reduction was removed and the win-rate of the robot was observed. The robot was successful in learning with a maximum win rate of 92%.**

I. Introduction

A neural network was used in place of a lookup Q table to implement Q-learning for a robot in Robocode. In order to implement the neural network a first step was offline training using training data obtained from a Q-table, obtained in part 2. Next, the neural network was used to replace the lookup table and finally state-space reduction was removed.

II. Experimental Results, Question 6

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

A. Question 6a

6a) Describe the architecture of your neural network and describe how the training set captured from Part 2 was used to "offline" train it.

Lookup-up table values were loaded as a training set for a neural network. The look-up table contained Q values for 128 x 5 actions. The original table values were mapped into state-action pair vectors suitable for inputs to the neural net. The target values of the neural net for backpropagation were the associated Q-values for each state-action pair.

The state-action vector, with state-reduction, is made up of states and actions as follows (see Appendix A and B for more detail):

Inputs:

State: [heading][distance][bearing][isHitWall]
Actions: [aim-fire, chase, retreat, ahead, fire]
State-Action (s,a): [0-3, 0-3, 0-3, T/F, T/F, T/F, T/F, T/F, T/F]

Using backpropagation, different numbers of hidden neurons were used with adjusted learning rate and momentum values in order to find convergence. The activation function was also adjusted to correspond to maximum and minimum Q-values. The fastest convergence was found using larger number of neurons for this particular data set. Lower numbers of neurons did not converge or were observed to converge very slowly.

The final neural network architecture chosen was a 9-20-1 network with 0.4 learning rate and 0.9 momentum values.

B. Question 6b

6b) Show (as a graph) the results of training your neural network using the contents of the LUT from Part 2.

The error for convergence was chosen as <25% of the absolute maximum Q value. A result of the convergence is shown below.

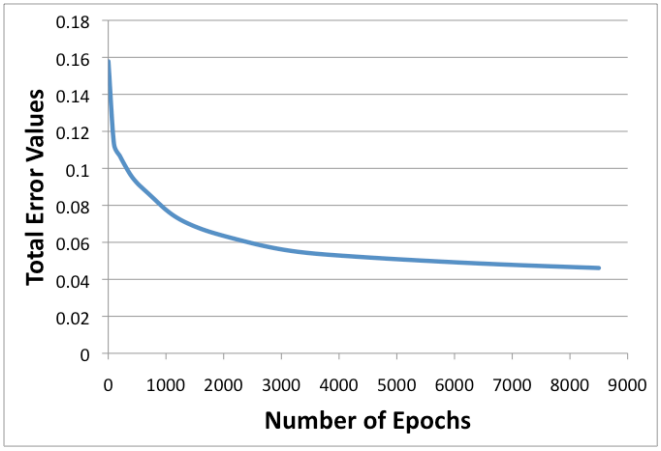

Figure 1: Neural network training convergence

C. Question 6c

6c) 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 comparison of the input representation used in Part 2 with that used by the neural net in Part 3 should be provided. Did you see any difference in the success of learning with and without this reduction? Here, just compare 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.

By applying the constants found for convergence 0.4 learning and 0.9 momentum, and using the optimal weights from part 2 of epsilon-greedy, discount, and learning rate, the neural net was implemented with the robot. Some small adjustments to the parameters were made, including reducing the neural net learning rate from 0.4 to 0.3 in order to improve performance. Also, the win rate was evaluated every 100 rounds and if the number of wins was not greater with learning, the previous neural net weight table was reloaded. By reloading the weight table, the learning did not decline to 0 and was relatively stable. Sample results are as follows.

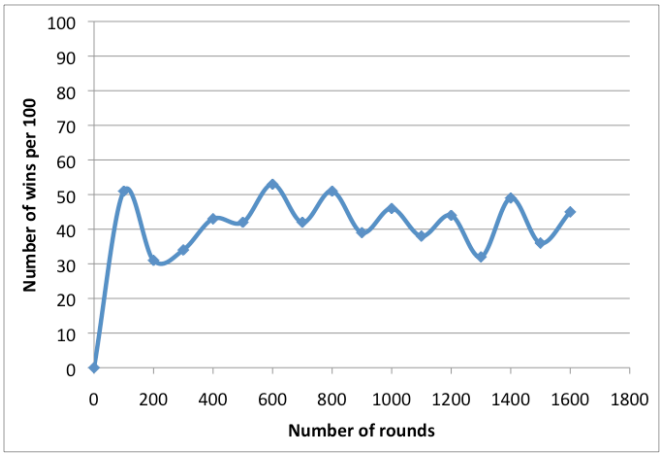

Figure 2: Sample results for 1600 rounds, using state reduction

Several adjustments to parameters were tried but it did not seem possible to improve the win rate beyond an average win rate of about 40-45% with state-reduction. Additionally, there were cases where the win rate did not seem to converge at all and the robot was observed to get into an action pattern that was not on track for winning. For example, there were cases where the robot would converge into a constant retreat that was unsuccessful in learning.

Removing state-space reduction changed the input vector to the neural net as follows:
Inputs:

State: [heading][distance][bearing][isHitWall]
Actions: [chase, fire, aim-fire, retreat, ahead]

Inputs with state reduction:

(s,a): [0-3, 0-3, 0-3, T/F, T/F, T/F, T/F, T/F, T/F]

Inputs without state reduction or normalization:
(s,a): [0…6.28, 0…1000, -3.14…3.14, T/F, T/F, T/F, T/F, T/F, T/F]
Inputs without state reduction with normalization
(s,a): [0…6.28, 0…10, -3.14…3.14, T/F, T/F, T/F, T/F, T/F, T/F]

Observing the state-action vector, the distance value did not typically rise any larger than 600 and by normalizing the value by a factor of 100, the typical value was 0-6 that was comparable to the heading and bearing input values. Removing the state-space reduction and running the algorithm showed an increase in performance of about 30% and a maximum win rate was above any win rate observed in part 2. The results below were found in the first run after increasing the number of states.

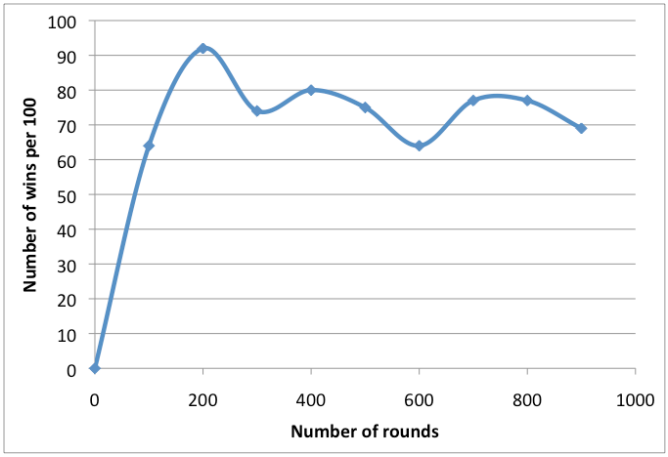

Figure 3: Sample results, no state space reduction

The increase in performance was due to the increase in more states. Since the states were not reduced to a small subset, the robot could learn more effectively to win against the opponent.

After running the program several times, it was noted that the learning was more sensitive to input parameters. Adjusting epsilon or the learning rate for example, reduced the win rate significantly. Based on this sensitivity, it was noted that it would have been very difficult to find all of the correct algorithm parameters without state-reduction and 'offline' training as a first step.

D. Question 6d

6d) 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.

A look-up table uses a 2-dimensional array to hold Q values and memory requirements scale with the state space. Neural nets do not scale in size with the size of state space. Specifically, they have a memory requirement associated with the number of inputs, hidden neurons, and outputs. State space reduction does not change the neural net implementation and does not benefit the implementation in the same way as the look-up table.

III. EXPERIMENTAL RESULTS, QUESTION 7

7) 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. Question 7a

7a) What was the best win rate observed for your tank? Describe clearly how your results were obtained? Also describe results due to any other parameters that you may have used to measure how well your system learned.

The best win rate observed was 92% against the Robocode sample myFirstRobot, which was an increase from the implementation of using the Q-table where a maximum win rate of 84% was observed. The win-rate was obtained using the neural net without state-reduction and the result is shown in Figure 3 and described in the previous question with parameters for the neural net of 0.3 learning rate, 0.9 momentum, and for the Q-learning epsilon greedy of 0.15, reinforcement learning rate of 0.1 and discount factor of 0.2.

Also the total number of turns was observed to increase with learning. The number of turns corresponded to the length of the battle and so the number of turns increased as learning progressed and the robot became more proficient at winning against the opponent and putting up a good fight when it did not win.

In addition to number of win rates, total reward and number of turns were monitored. An example is shown below where number of turns is increasing as learning occurs.

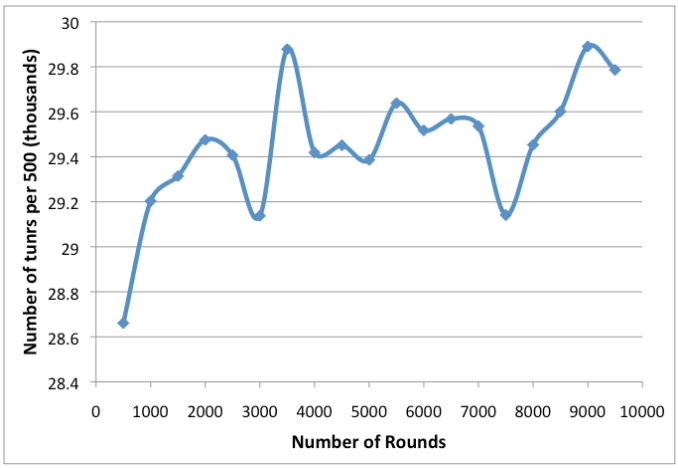

Figure 4: Number of turns increasing with learning

B. Question 7b

7b) Plot the win rate against number of battles. As training proceeds, does the win rate improve asymptotically? In cases where the win rate does not asymptotically converge, are you able to comment why that might be the case?

The win rate versus number of battles was previously shown in earlier question 6. Figure 5 is repeated for completeness.

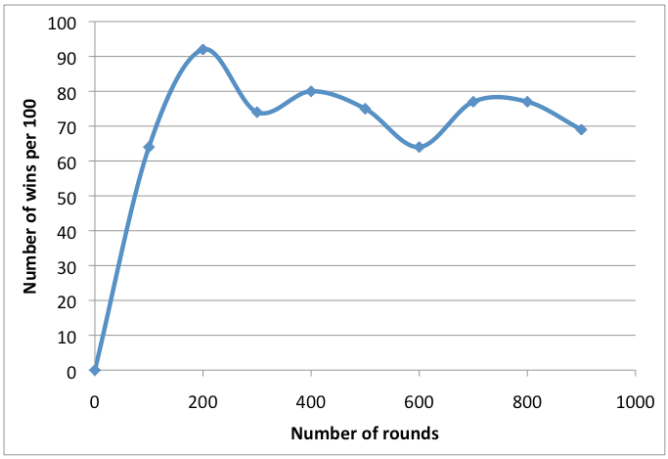

Figure 5: Sample results, no state space reduction

The win rate tended to oscillate as it converged to some win-rate average. In order to reduce oscillations from going to a zero win-rate, the previous weight table for the neural net was loaded if an increase in win-rate was not found every 100 rounds. In addition, experiments were completed with decaying the learning rate and epsilon-greedy to lower values as time progressed in order to reduce oscillations and increase stability. This was observed to create positive results in many cases. Additionally, there were test runs where the win rate did not rise above 10% and the learning was unsuccessful. The non-convergence and oscillating behavior was attributed to the neural network. Additional analysis on convergence of the neural net is provided in question 7c.

C. Question 7c

7c) With a look-up table, the TD learning algorithm is proven to converge – i.e. will arrive at a stable set of Q-values for all visited states. This is not so when the Q-function is approximated. Explain this convergence in terms of the Bellman equation and also why when using approximation, convergence is no longer guaranteed.

The Bellman equation describes the relationship between the current state and the next state and is described as follows (Sarkaria, class notes):

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

and it can be shown that the relationship of error from one state to the next is

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

The process can be modeled as a Markov chain with a terminal state where the error of the terminal state is 0 by definition. In order for the above equation to be satisfied, it must be that the remaining error also goes to 0 for the equation to be satisfied and V→V*.

In the case of Q-learning using a neural net approximation, a gradient descent method is applied to the error that is updated each iteration. The new target value depends on (s',a') and reward.

$$T\arg et = Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$$

The error may increase or decrease and the behavior outcome from applying gradient descent can have oscillations or non-convergent behavior. (Harmon, 2010).

D. Question 7d

4d) Do you think that your learning was converging or not? Show your results to back this up.

Learning converged to an oscillating value in the majority of cases and in some cases it did not converge at all. The weight tables were reloaded to help provide stability around a win rate. If the weight table was not reloaded, some results appeared to fall to zero after doing very well with learning. The learning was relatively stable reloading the states and lowering the epsilon-greedy and learning rate. Below is an example using state-space reduction, where the learning was relatively stable over 8000 iterations.

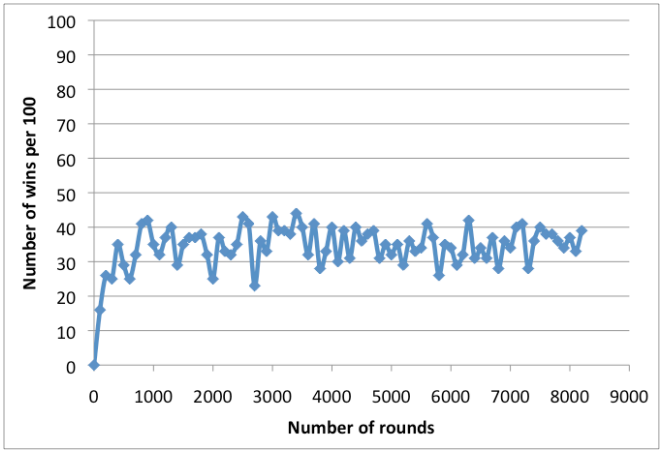

Figure 6: Sample results for 8000 rounds, using state reduction

In non-convergence, the total reward became worse and the robot had a consistent zero win rate. An example of the decreasing accumulated reward for a zero win rate is shown below.

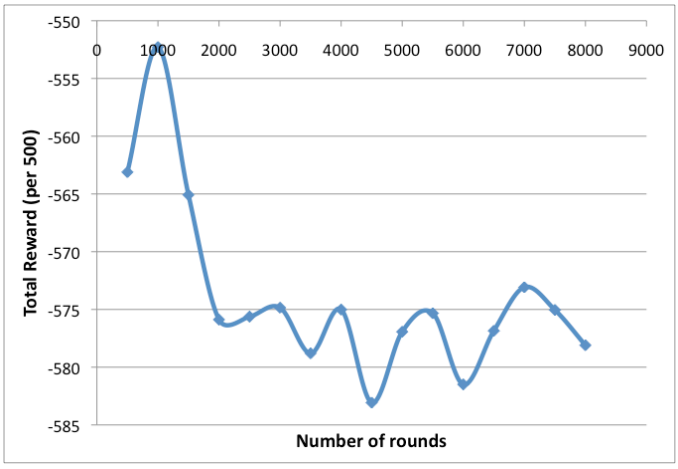

Figure 7: Example of total reward decreasing with non-convergence and zero win-rate

IV. EXPERIMENTAL RESULTS, QUESTION 8

A. Overall Conclusions

8a) Did your robot perform as you might have expected?

The robot behavior did perform relatively well and as expected was able to achieve a higher win-rate maximum than the Q-table as the neural network did not require state-reduction. It was found that the network was very sensitive to network parameters and stability of learning and predictability of win-rate was less than expected. With additional time for simulations, it may be possible to reconstruct a more stable network with more reproducible and predictable results. A first step could be to re-visit the neural network architecture and work on finding a more optimal number of hidden neurons. A second step could be to re-visit the reward system since the reward values were very dependant on bullet hits and providing more alternatives for rewards may have helped learning behavior converge more predictably.

B. Question 8b

8b) 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 converge problems be addressed? What advice would you give when applying RL with neural network based function approximation to other practical applications?

In order to improve robot performance, many parameters can be adjusted. Starting with a good foundation of states, actions, and rewards, was important in implementing the reinforcement-learning algorithm. Adjusting weights, states, and actions had a significant impact on potential win-rate of the robot.

In order to use a neural net to replace the look-up table, the approach of using the Q-table to find a convergent neural network architecture was extremely helpful. It would be difficult find a convergent behavior without taking this step since there are many parameters to adjust. The step-by-step approach saved time in trial and error of parameter adjustments.

The reduction in state-space was required in order to implement the Q-table and was helpful in finding the optimal parameters in the neural net implementation with the robot. After state-space reduction was removed, the behavior seemed very sensitive to parameter adjustments. With the state-space reduction removed, the win-rate was improved and the optimal implementation should not have state-space reduction in order to reach the maximum win rate. However, the neural network did introduce some unpredictability and was generally less stable in this implementation. It may be possible to find a more stable neural network solution, but the Q-table was found to be much easier to implement in this regard.

Finally, in order to increase stability, the neural net weight values were re-loaded if win-rate was not increased with

learning. This helped maintain the winning behavior of the robot once it was learned.

V. CONCLUSION

Reinforcement learning with a neural network was implemented in order to teach a robot in Robocode how to beat an opponent. In order to implement the neural net, the architecture was chosen by using 'offline' training with Q-values obtained from a lookup table in part 2. State-reduction was removed once the neural network was able to converge.

APPENDIX

Appendix A: State Vector Sample
Appendix B: State-Action Vector Sample
Appendix C: Source Code – Robot Class
Appendix D: Source Code – Backpropagation Driver Class
Appendix E: Source Code – Neural Net Class
Appendix F: Source Code – LUT Class

REFERENCES

[1] M.Harmon. "Reinforcement Learning: a Tutorial. Wright Laboratory. http://courses.ece.ubc.ca/592/rltutorial.pdf
[2] S. Sarkaria, course notes for EECE 592, 2010 http://courses.ece.ubc.ca/592/PDFfiles/Reinforcement_Learning2_c.pd
[3] R. Sutton, & A. Barto, "Reinforcement Learning: An Introduction, " MIT Press, Cambridge, MA, 1998.

Appendix A: State Vector Sample (128 states)

[0--3]	[0--3]	[0--3]	[0--1]	State #
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	0	2	0	4
0	0	2	1	5
0	0	3	0	6
0	0	3	1	7
0	1	0	0	8
0	1	0	1	9
0	1	1	0	10
0	1	1	1	11
0	1	2	0	12
0	1	2	1	13
0	1	3	0	14
0	1	3	1	15
0	2	0	0	16
0	2	0	1	17
0	2	1	0	18
0	2	1	1	19
0	2	2	0	20
0	2	2	1	21
0	2	3	0	22
0	2	3	1	23
0	3	0	0	24
etc	etc	etc	etc	etc
...	...	...	...	...

Appendix B: State-Action Vector Sample

```
1.0 3.0 2.0 1.0 0.0 -1.0 1.0 -1.0 -1.0 -1.0
1.0 3.0 2.0 2.0 0.0 1.0 -1.0 -1.0 -1.0 -1.0
1.0 3.0 2.0 2.0 0.0 1.0 -1.0 -1.0 -1.0 -1.0
1.0 2.0 2.0 2.0 0.0 1.0 -1.0 -1.0 -1.0 -1.0
1.0 2.0 2.0 2.0 0.0 -1.0 -1.0 1.0 -1.0 -1.0
1.0 2.0 2.0 2.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 2.0 2.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 2.0 2.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 2.0 3.0 0.0 -1.0 -1.0 -1.0 1.0 -1.0
1.0 2.0 2.0 3.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 2.0 3.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 2.0 3.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 2.0 2.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 2.0 2.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 0.0 0.0 0.0 -1.0 -1.0 1.0 -1.0 -1.0
1.0 2.0 1.0 1.0 0.0 -1.0 1.0 -1.0 -1.0 -1.0
1.0 2.0 1.0 1.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
1.0 2.0 0.0 1.0 0.0 -1.0 -1.0 -1.0 -1.0 1.0
```

Appendix C: Source Code – Robot Class

```
package ca.ubc.ece.lut;
import java.awt.Color;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;

import ca.ubc.ece.backprop.NeuralNet;

import robocode.AdvancedRobot;
import robocode.BattleEndedEvent;
import robocode.BulletHitEvent;
import robocode.DeathEvent;
import robocode.HitByBulletEvent;
import robocode.HitRobotEvent;
import robocode.HitWallEvent;
import robocode.RobocodeFileOutputStream;
import robocode.Robot;
import robocode.RoundEndedEvent;
import robocode.ScannedRobotEvent;
import robocode.WinEvent;

/**
 * Daphne Robot - created from Mathew Nelson's sample MyFirstRobot
 * EECE 592 Part 3 - Reinforcement Learning && Neural Net
 * Alison Michan, December 2010 - robot uses reinforcement learning with LUT class
 */
public class Daphne extends AdvancedRobot {
	//declarations
	static double learningRateOrigR=0.1;// 0.08/0.1 is good
	static double learningRateR=0.1;
	static double learningRateNN=0.3;//0.3/0,4 is good
	static double discount=0.2;
	static double momentum=0.9;//from 0.9
	static double epsilonGreedy=0.15;	//initialize epsilonGreedy
	static double epsilonGreedyOrig=0.15; // 0.15 TO 0.2

	//static LUT lookupQ = new LUT(2, 12, learningRate, 0.1,-1.0, 1.0); //constructor, STATIC
	//static LUT lookupQPrevious = new LUT(2, 12, 0.15, 0.2,-1.0, 1.0); //constructor, STATIC
	static NeuralNet Trial = new NeuralNet(9,20,learningRateNN, momentum); //constructor, STATIC

	static double []weightHPrevious=new double[21];
   static double [][]weightIHPrevious=new double[10][21];

	double [] stateVector = new double[5]; //hard code numStates
	double [] currentStateVector = new double[5]; //hardcode stateVector
	double [] stateAction = new double[10]; //hardcode stateVector

	//process variables
	double currBearing=0.0;
	double currHeading=0.0;
	double currDistance=0.0;
	double currRadar=0.0;
	double reward=0.0; //reward values hardcoded
	int isHitWall=0;
	double BReward=0.5;
	double SReward=0.02;
	double BNReward=-0.5;
	double SNReward=-0.02;
	int currentAction = 4;			//initialize action a'
	int previousAction = 4;			//initialize previous action a
	static boolean trainFlag=false;
	static int numTrainRounds=0;
	static int numTestRounds=0;
	static int numWonRounds=0;
	static int numWonRoundsPrev=0;
	static String saveWins="";
	static String saveWinsTrain="";
	static int total=0;
	double error=0.0;
	double  outputCurr=0.0;
	double Target=0;
	static String SaveTotalReward="";
```

```
static double TotalReward=0;
static String SaveTotalTurns="";
static int TotalTurns=0;
static boolean firstRunFlag=true;

public void run() {
	setBodyColor(Color.pink);	//set colors
	setGunColor(Color.gray);
	setRadarColor(Color.pink);
	currentStateVector[0]=1; //bias
	stateVector[0]=1; //bias
	for(int i=1; i<5; i++){
		currentStateVector[i]=0;
		stateVector[i]=0;
	}
	setAdjustGunForRobotTurn(false);
	setAdjustRadarForGunTurn(true);

	while (true) {
		double epsilonGreedyTemp = Math.random(); //epsilon greedy
		if(epsilonGreedyTemp<epsilonGreedy)
		{
			double tempE=Math.random();
		//	System.out.println(tempE);
			if(tempE<=0.2) {currentAction = 0;}
			if(tempE>0.2 && tempE<=0.4) {currentAction = 1;}
			if(tempE>0.4 && tempE<=0.6) {currentAction = 2;}
			if(tempE>0.6 && tempE<=0.8) {currentAction = 3;}
			if(tempE>0.8) {currentAction = 4;}
		}

		doAction(currentAction); //do currentAction a
		turnRadarRightRadians(6.28); //observe r and stateVector s in onScannedRobot
		stateAction=indexState(currentStateVector, currentAction);//returns (s,a) combined in a vector
		outputCurr=Trial.outputFor(stateAction); //returns Q(s,a)
		Target=outputCurr+learningRateR*(reward + discount * getMaxQ(stateVector)-outputCurr); //Target is newQ, stateVector is (s') and MaxQ found at a'
		if(trainFlag) error=Trial.train(stateAction, Target);//update NeuralNet weights
		//System.out.print("\nrewardMain:\t"+reward+"\n");
		TotalReward=TotalReward+reward;
		reward=0.0;//re-set reward
		isHitWall=0;//re-set isHitWall
		currentStateVector=stateVector; //update currentState s to s'
		currentAction=getMaxQAction(currentStateVector); //update a to a' from the maxQ action

		//output data for monitoring
		//for(int i=0; i<10; i++) System.out.print(stateAction[i]+" ");
		//System.out.print("\n");
		//System.out.print("\n");
		//System.out.print("\terrorCurr:\t");
		//System.out.print(error);
		//System.out.print("\toutputCurr:\t");
		//System.out.print(outputCurr);
		//System.out.print("\tTarget:\t");
		//System.out.println(Target);
	}
}
public void onScannedRobot(ScannedRobotEvent e) {
	stateVector[0]=1;//bias
	stateVector[1]=e.getHeadingRadians();//getHeading(e.getHeadingRadians());//getHeading(e.getHeadingRadians());
	currHeading=getHeadingRadians();
	stateVector[2]=e.getDistance()/100;//getTargetDistance(e.getDistance());////
	currDistance=e.getDistance();
	stateVector[3]=e.getBearingRadians();//getTargetBearing(e.getBearingRadians());	////
	currBearing=e.getBearingRadians();
	currRadar=getRadarHeadingRadians();
	//if(isHitWall==0) isHitWall=-1;
	stateVector[4]=isHitWall;
	//System.out.println(stateVector);
}


public void onHitByBullet(HitByBulletEvent e) {
	reward=reward+SNReward;
}
public void onBulletHit(BulletHitEvent e) {
```

```
                reward=reward+SReward;//*2 in this
                //System.out.print("\nreward:\t"+reward+"\n");
        }
        public void onHitWall(HitWallEvent e)
         {
           reward=reward+SNReward/2;
           isHitWall=1;
         }


        public void onDeath(DeathEvent e) {
                reward=reward+BNReward;
                outputCurr=Trial.outputFor(stateAction); //returns Q(s,a)
                Target=outputCurr+learningRateR*(reward + discount * getMaxQ(stateVector)-outputCurr);
                //Target is newQ, stateVector is (s') and MaxQ found at a'
                if(trainFlag) error=Trial.train(stateAction, Target);//update NeuralNet weights
                //System.out.print("\nreward:\t"+reward+"\n");

                TotalReward+=reward;
                reward=0;
                isHitWall=0;
        }
        public void onWin(WinEvent e) {
                reward=reward+BReward;
                outputCurr=Trial.outputFor(stateAction); //returns Q(s,a)
                Target=outputCurr+learningRateR*(reward + discount * getMaxQ(stateVector)-outputCurr);
                //Target is newQ, stateVector is (s') and MaxQ found at a'
                if(trainFlag) error=Trial.train(stateAction, Target);//update NeuralNet weights
                //System.out.print("\nreward:\t"+reward+"\n");
                TotalReward+=reward;
                reward=0;
                isHitWall=0;
                numWonRounds++;
        }

//---------Helper Functions-------------------

        public void assignWeightsPrev(){
                for(int i=0; i<=20; i++)
                        weightHPrevious[i]=Trial.getWeightHO(i);
           for(int j=0; j<10; j++)
                for(int k=0; k<20; k++)
                        weightIHPrevious[j][k]=Trial.getWeightIH(j,k);
        }
        public void updateWeightsPrev(){
                for(int i=0; i<=20; i++)
                        Trial.setWeightHO(i,weightHPrevious[i]);
           for(int j=0; j<10; j++)
                for(int k=0; k<20; k++)
                        Trial.setWeightIH(j,k,weightIHPrevious[j][k]);
        }


        public void doAction(int argAction)
        {
                TotalTurns=TotalTurns+1;
                if(argAction==0) //aim fire
           {
                        double temp = currBearing;
                        if(temp>0) turnLeftRadians(-1*temp);
                        if(temp<0) turnRightRadians(temp);
                        fire(1);
           }
                if(argAction==1) //chase
                {
                        double temp = currBearing;
                        if(temp>0) turnLeftRadians(-1*temp);
                        if(temp<0) turnRightRadians(temp);
                        ahead(currDistance/2) ;//15
                        fire(1);
                }

                if(argAction==2) //spin retreat fire
                {
```

```
                turnLeftRadians(3.14/2); //spin
                back(125);
                fire(1);

        }
        if(argAction==3) //ahead
        {
                ahead(50);

        }
        if(argAction==4) //fire random
        {
                fire(3);
        }
}

public double getMaxQ(double currentStateVector[])
{
        double maxQ=Double.NEGATIVE_INFINITY;
        for(int i=0; i<5; i++){
                double temp=Trial.outputFor(indexState(currentStateVector, i));
                if(temp>maxQ){
                        maxQ=temp;
                }
        }
        return maxQ;
}

public int getMaxQAction(double currentStateVector[]){
        double maxQ=Double.NEGATIVE_INFINITY;
        int action=0;
        for(int i=0; i<5; i++){
                double temp=Trial.outputFor(indexState(currentStateVector, i));
                if(temp>maxQ){
                        maxQ=temp;
                        action=i;
                }
        }
        return action;
}

/*
 * input is state vector and action number
 * returns state action vector for use in neural net
 */

public double [] indexState(double[]state, int action){

        double [] SA = new double[10];
        for(int i=0; i<10; i++)
                SA[i]=-1;
        for(int i=0; i<5; i++)
                SA[i]=state[i];
        if(action==0) SA[5]=1;
        if(action==1) SA[6]=1;
        if(action==2) SA[7]=1;
        if(action==3) SA[8]=1;
        if(action==4) SA[9]=1;

        return SA;

}
public int getHeading(double arg)
{

        //4
        int temp=0;
        if (arg>=0 && arg<(Math.PI/2)) temp=0;
        if (arg>=Math.PI/2 && arg<(Math.PI)) temp=1;
        if (arg>=Math.PI && arg<(Math.PI*3/2)) temp=2;
        if (arg>=(Math.PI*3/2)) temp =3;
        return temp;

}
public int getTargetDistance(double arg)
```

```
{		//4 close, near, far, really far
		int temp=(int)(arg/100);
		if(temp>3) temp=3;
		return (temp);
}
public int getTargetBearing(double arg)
{

		//4
		int temp=0;
		arg=arg+Math.PI;
		if (arg>=0 && arg<(Math.PI/2)) temp=0;
		if (arg>=Math.PI/2 && arg<(Math.PI)) temp=1;
		if (arg>=Math.PI && arg<(Math.PI*3/2)) temp=2;
		if (arg>=(Math.PI*3/2)) temp =3;
		return temp;


}

public void onRoundEnded(RoundEndedEvent e){

		System.out.println("trainFlag:"+trainFlag+" numTrainRounds:" + numTrainRounds + " numTestRounds:"
				+ numTestRounds +" numWon" +numWonRounds+ "Previous" +numWonRoundsPrev
				+ "Greedy" +epsilonGreedy + "totalRounds" +total+ "alpha" +learningRateR
				+"NN_Learning" +Trial.learningRate+"NN_M" +Trial.momentumWeight);
		System.out.print("\nSaveWinsTrain: "+saveWinsTrain);
		System.out.print("\nReward: "+SaveTotalReward);
		System.out.print("\nTurns: "+SaveTotalTurns);

		if(trainFlag) {
			if(numTrainRounds==500)
			{ //alternate between training and no training in multiples
				numTrainRounds=0;
				System.out.print("inside decision: numWon: "+numWonRounds+" numPrev: "+numWonRoundsPrev);
				//use if want to only accept 'improved' q-table
				if(numWonRoundsPrev>numWonRounds){
					//revert weights
					updateWeightsPrev();
					System.out.print("\ninside prev if \n");
					//do not update numWonRoundsPrev
					}
				else{
					//keep new weights in previous holder
					assignWeightsPrev();
					numWonRoundsPrev=numWonRounds; //update numMaxWins
					System.out.print("\ninside else\n");
					}
				saveWinsTrain=saveWinsTrain+Integer.toString(numWonRounds);
				saveWinsTrain=saveWinsTrain+"\t";
				SaveTotalReward=SaveTotalReward+Double.toString(TotalReward);
				SaveTotalReward=SaveTotalReward+"\t";
				SaveTotalTurns=SaveTotalTurns+Integer.toString(TotalTurns);
				SaveTotalTurns+="\t";
				TotalReward=0;
				TotalTurns=0;
				numWonRounds=0;
				total=total+100;
				trainFlag = false;
				System.out.print("\nTrial\t"+Trial.getWeightHO(3)+"\t"+Trial.getWeightHO(4)+"\t"+Trial.getWeightHO(5)+"\n");
				System.out.print("Previous\t"+weightHPrevious[3]+"\t"+weightHPrevious[4]+"\t"+weightHPrevious[5]+"\n");



			} else{
				numTrainRounds++;
				//System.out.print(TotalReward+"\t"+TotalTurns);
				System.out.print("\nTrial\t"+Trial.getWeightHO(3)+"\t"+Trial.getWeightHO(4)+"\t"+Trial.getWeightHO(5)+"\n");
				System.out.print("Previous\t"+weightHPrevious[3]+"\t"+weightHPrevious[4]+"\t"+weightHPrevious[5]+"\n");

				//use to exponentially decay epsilonGreedy and/or learning rate
				//if(epsilonGreedy>0.08) epsilonGreedy=epsilonGreedyOrig*Math.exp(-total/400);
				//if(learningRateR>0.05) learningRateR=learningRateOrigR*Math.exp(-total/800);//change from 400
```

```
                                        //if(learningRateR<0.05) learningRateR=0.05;
                                        //if(epsilonGreedy<0.08) epsilonGreedy=0.08;
                                }
                        } else {
                                if(numTestRounds==3) { //alternate between training and no training in multiples
                                        if(firstRunFlag){//baseline data
                                                saveWinsTrain=saveWinsTrain+Integer.toString(numWonRounds);
                                                saveWinsTrain=saveWinsTrain+"\t";
                                                SaveTotalReward=SaveTotalReward+Double.toString(TotalReward);
                                                SaveTotalReward=SaveTotalReward+"\t";
                                                SaveTotalTurns=SaveTotalTurns+Integer.toString(TotalTurns);
                                                SaveTotalTurns+="\t";
                                        }
                                        firstRunFlag=false;
                                        numTestRounds = 0;
                                        trainFlag = true;
                                } else{
                                        numTestRounds++;
                                        //total++;
                                }

                        }
                }

                /**
                 * At the end of a battle, write q-table to file as well as number of wins stats
                 */

                public void onBattleEnded(BattleEndedEvent e){
                        System.out.println("Battle ended!");
                        Trial.save(getDataFile("log.csv"));

                        PrintStream writer;
                        try {
                                writer = new PrintStream(new RobocodeFileOutputStream(getDataFile("wins.csv")));
                                writer.append(saveWins +"\n" + saveWinsTrain+ "\n"+ SaveTotalReward+"\n"+ SaveTotalTurns);
                           writer.close();

                        } catch (IOException e1) {
                                // TODO Auto-generated catch block
                                e1.printStackTrace();
                        }

                }
}
```

Appendix D: Source Code – Backpropagation Driver Class

```
package ca.ubc.ece.backprop;

import java.io.IOException;

public class BPDriver {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        for(int test=0; test<1; test++){
        NeuralNet Trial = new NeuralNet(9,20,0.4,0.9);
        //numInputs=2, numHidden=4, learningRate=0.2, momentum=0
        double error=0.0;
        double  outputCurr=0.0;
        double X[]=new double[10]; //vector input to neural net
        double Target=0;;
        int numEpochs = 0;
        double RMSError=0.0;

        try {
            NeuralNet.loadLUT("/Users/amichan/Documents/workspace/backprop/bin/ca/ubc/ece/lut/Daphne.data/TrainDataP.csv");
        } catch (IOException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }
        //NeuralNet.initializeTrainingDataBinary(); //load xor training data
        do{
            RMSError=0.0;
            for(int trainingPairNum=0; trainingPairNum<128; trainingPairNum++)
            {
                for(int a=0;a<5;a++){ //load training inputs
                    X=Trial.indexState(trainingPairNum,a);
                    //for(int i=0; i<10; i++){
                    //    System.out.print(X[i]+"\t");
                    //}
                    //System.out.print(trainingPairNum+"\t" + a +"\t");

                    Target=Double.valueOf((Trial.QTableNum[trainingPairNum][a]).trim()).doubleValue();
                    outputCurr=Trial.outputFor(X);
                    error=Trial.train(X, Target);
                    RMSError=RMSError+0.5*error*error;


                //System.out.print("\terrorCurr:\t");
                //System.out.print(error);
                //System.out.print("\toutputCurr:\t");
                //System.out.print(outputCurr);
                //System.out.print("\tTarget:\t");
                //System.out.println(Target);
                }
            }
            numEpochs++;
            if(numEpochs>990000)
            {
                RMSError=0.004;
            }
            if(numEpochs==2)System.out.print(numEpochs+"\t" +RMSError+ "\n");
            if(numEpochs%100==0) System.out.print(numEpochs+"\t"+RMSError+"\n");

        }while(RMSError>0.04608);//(RMSError>0.05);
        System.out.println("numEpochs:\t"+numEpochs);

        }
    }
}
```

Appendix E: Source Code – Neural Net Class

```
package ca.ubc.ece.backprop;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintStream;
import java.math.*;
import java.util.StringTokenizer;

import ca.ubc.ece.lut.LUT;

import robocode.RobocodeFileOutputStream;

/*
 * @author Michan
 */

public class NeuralNet {

/*
 *  Private members of this class
 */


/*
 * Public attributes of this class. Used to capture the largest Q
 */
    public static double[] hiddenNeurons; //hidden neuron vector
```

```
    public static double[] lastWeightChangeHiddenOutput; //momentum
    public static double [][] lastWeightChangeInputToHidden; //momentum
    public static double []weightHiddenOutput;//= {-0.1401,0.4919,-0.2913,-0.3979,0.3581};//textbook weights
    public static double [][]weightInputToHidden;//={ {-0.3378, 0.2771, 0.2895, -0.3329},
                {0.1970,0.3191,-0.1448, 0.3594},{0.3099, 0.1904, -0.0347, -0.4861} };//textbook weights
    public static double sigma_a=-.1; //sigmoid bound a
    public static double sigma_b=.1; //sigmoid bound b
    public static double gamma=sigma_b-sigma_a; //for custom sigmoid and prime
    public static double n=-sigma_a; //for custom sigmoid and prime
    public static int numHidden;
    public static int numInputs;
    public static double learningRate;
    public static double momentumWeight;

    public static int numPatterns;
    public static double[][] trainingDataInput;
    public static double[] trainingDataOutput;

    public static int patNum; //process variable
    public static double y_output;

    //this is not great way to do this but ok
    public static int numRows=128;
    public static int numCol=5;
    public static String [][] QTableNum = new String [numRows][numCol];

/*
 * Constructor.
 */
     public NeuralNet (int argNumInputs,
               int argNumHidden,
               double argLearningRate,
               double argMomentum)
     {
          numHidden=argNumHidden;
          numInputs=argNumInputs+1; //add 1 for bias
          learningRate=argLearningRate;
          momentumWeight=argMomentum;
          lastWeightChangeHiddenOutput=new double[numHidden+1]; //momentum
          lastWeightChangeInputToHidden=new double [numInputs][numHidden]; //momentum
          hiddenNeurons=new double[numHidden+1];//add 1 for bias of 1 at [0]
          numPatterns=128*5; //4xor
          trainingDataInput = new double[numPatterns][numInputs]; //xor
          trainingDataOutput= new double[numPatterns];
          y_output=0.0;
          weightHiddenOutput=new double[numHidden+1];
         weightInputToHidden=new double[numInputs][numHidden];
          initializeWeights();
          System.out.println(numHidden);
          System.out.println(numInputs);
     }

/*
 *   Return a sigmoid of the input X //make private
 */
    private double sigmoid( double x ) {
        double temp = Math.exp(-x);
        return (1/(1+temp));
    }
/*
 * This method implements a bipolar sigmoid
 */
    private double customSigmoid( double x ) {

        double temp=Math.exp(-x);
        temp=1/(1+temp);
        temp=gamma*temp-n;
        return temp;

    }

/*
 * Initialize the weights to random values.
 */
```

```
    private void initializeWeights() {
        double temp=0.0;

        for(int j=0;j<=numHidden;j++)
        {
            temp=(Math.random()-0.5);
            if((Math.abs(temp))<0.15){
                temp=temp*10;
            }
            weightHiddenOutput[j]=temp; //make random
        }
        for(int j=0;j<numHidden;j++)
        {
            for(int k=0;k<numInputs;k++)
            {
                temp=(Math.random()-0.5);
                if((Math.abs(temp))<0.15)
                {
                    temp=temp*10;
                }
                weightInputToHidden[k][j]=temp; //make random
            }
        }
    }

/*
 * Computes output of the NN without training. ie a forward pass
 * pass in training pair including bias
 */
    public double outputFor( double[] X)
    {
        hiddenNeurons[0]=1.0; //apply bias of 1 at hiddenNeurons Zo
        for (int j=1; j<=numHidden; j++) //count through hidden neurons not bias (4 in assign 1)
        {
            hiddenNeurons[j]=0.0;
            for(int i=0;i<numInputs;i++) //count through bias and inputs x1,x2
            {
                hiddenNeurons[j]=hiddenNeurons[j]+(X[i]*weightInputToHidden[i][j-1]);
            }
            hiddenNeurons[j]=customSigmoid(hiddenNeurons[j]);//bipolar
            //hiddenNeurons[j]=sigmoid(hiddenNeurons[j]);//binary
        }
        y_output=0.0;
        for(int j=0; j<=numHidden; j++)
        {
            y_output=y_output+hiddenNeurons[j]*weightHiddenOutput[j];
        }
        y_output=(customSigmoid(y_output));//bipolar
        return y_output;
        //return (sigmoid(y_output));//binary
    }

/*
 * This method is used to update the weights of the neural net.
 * Returns the error that is the difference of
 * the target input and the outputFor(inputStateVector)that was updated y_output
 * in call to outputFor just before
 */
    public double train( double[] argInputVector, double argTargetOutput)
    {
        double errorDelta=0.0;
        double outputError=0.0;
        double hiddenDelta =0.0;
        double[] weightChangeHiddenOutput = new double[numHidden+1]; //momentum
        double[][] weightChangeInputToHidden= new double [numInputs][numHidden]; //momentum

        //backpropagation to hidden layer
        outputError=(argTargetOutput-y_output); //bipolar
       errorDelta=(outputError*(1/gamma)*(n+y_output)*(gamma-n-y_output));//bipolar
    //outputError=argTargetOutput-(sigmoid(y_output));//binary/bipolar
        //errorDelta=outputError*sigmoid(y_output)*(1-sigmoid(y_output));//hard coded - switch to bipolar/binary
        for(int j=0; j<=numHidden; j++) //j=0 is bias, where hiddenNeurons[0]=1
        {
            //calculate weight correction, added momentum here
            weightChangeHiddenOutput[j]=(learningRate*errorDelta*hiddenNeurons[j])+(momentumWeight*lastWeightChangeHiddenOutput[j]);
```

```
                lastWeightChangeHiddenOutput[j]=weightChangeHiddenOutput[j];
            }
            //backpropagation to input layer
            for(int j=1; j<=numHidden; j++) //offset since bias=1 term at hiddenNeurons[0]
            {
                hiddenDelta=weightHiddenOutput[j]*errorDelta;
                hiddenDelta=(hiddenDelta*(1/gamma)*(n+hiddenNeurons[j])*(gamma-n-hiddenNeurons[j]));//bipolar
                //hiddenDelta=hiddenDelta*(hiddenNeurons[j])*(1-hiddenNeurons[j]); //binary, sigmoid already applied to hiddenNeuron values
                for(int i=0; i<numInputs; i++) //i=0, X[0] is bias=1
                {
                    //calculate weight correction, added momentum here
                    weightChangeInputToHidden[i][j-1]=(learningRate*hiddenDelta*argInputVector[i])+
                                    (momentumWeight*lastWeightChangeInputToHidden[i][j-1]);
                    lastWeightChangeInputToHidden[i][j-1]=weightChangeInputToHidden[i][j-1];
                }
            }
            //weight change updates Hidden Output
            for(int j=0; j<=numHidden; j++)
            {
                weightHiddenOutput[j]=(weightHiddenOutput[j]+weightChangeHiddenOutput[j]);
            }
            //weight change updates InputToHidden
            for(int j=1; j<=numHidden; j++)
            {
                for(int i=0; i<numInputs; i++)
                {
                    weightInputToHidden[i][j-1]=(weightInputToHidden[i][j-1]+weightChangeInputToHidden[i][j-1]);
                }
            }
        return outputError;
        }

    /*
     * saves the weight arrays to a file.
     */
        public void save( File argFile){
            PrintStream writer;
            try {
                 System.out.println(numHidden);
                 System.out.println(numInputs);
                writer = new PrintStream(new RobocodeFileOutputStream(argFile));
                String saveFile="weightInputToHidden\n";
                for(int j=0; j<numHidden; j++){
                    saveFile+="\n";
                    for(int i=0; i<numInputs; i++){
                        saveFile+=weightInputToHidden[i][j]+"\t";
                    }
                }
                String saveFile2="\n\nweightHiddenOutput";
                for(int a=0;a<=numHidden; a++){
                    saveFile2+="\n";
                    saveFile2+=weightHiddenOutput[a]+"\t";
                }
                writer.append(saveFile);
                writer.append(saveFile2);
               writer.close();
            } catch (IOException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
            }
        }




    /*
     * loads the LUT Q values from file. format of the file is expected to follow
     */

        public static void loadLUT( String argFileName) throws IOException{

            File file = new File(argFileName);
            BufferedReader bufRdr;
            try {
                bufRdr = new BufferedReader(new java.io.FileReader(file));
```

```
        String line = null;
        int row = 0;
        int col = 0;
        //read each line of text file
        while((line = bufRdr.readLine()) != null && row < numRows)
        {
            StringTokenizer st = new StringTokenizer(line,",");
            while (st.hasMoreTokens())
            {
                //get next token and store it in the array

                QTableNum[row][col] = st.nextToken();

                col++;
            }
            col = 0;
            row++;
        }
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }


    }

    public static void initializeTrainingDataBinary(){
        //binary

        trainingDataInput[0][1]=1;
        trainingDataInput[0][2]=0;
        trainingDataInput[0][0]=1;//bias
        trainingDataOutput[0]=1;

        trainingDataInput[1][1]=0;
        trainingDataInput[1][2]=1;
        trainingDataInput[1][0]=1;//bias
        trainingDataOutput[1]=1;

        trainingDataInput[2][1]=1;
        trainingDataInput[2][2]=1;
        trainingDataInput[2][0]=1;//bias
        trainingDataOutput[2]=0;

        trainingDataInput[3][1]=0;
        trainingDataInput[3][2]=0;
        trainingDataInput[3][0]=1;//bias
        trainingDataOutput[3]=0;
    }

    public static void initializeTrainingDataBipolar(){
        //bipolar
        trainingDataInput[0][1]=1;
        trainingDataInput[0][2]=-1;
        trainingDataInput[0][0]=1;//bias
        trainingDataOutput[0]=1;

        trainingDataInput[1][1]=-1;
        trainingDataInput[1][2]=1;
        trainingDataInput[1][0]=1;//bias
        trainingDataOutput[1]=1;

        trainingDataInput[2][1]=1;
        trainingDataInput[2][2]=1;
        trainingDataInput[2][0]=1;//bias
        trainingDataOutput[2]=-1;

        trainingDataInput[3][1]=-1;
        trainingDataInput[3][2]=-1;
        trainingDataInput[3][0]=1;//bias
        trainingDataOutput[3]=-1;
```

```
    }


    //helper function---------

    public double[] indexState(int argState, int argAction) {
            int length=10;
        double [] stateVector=new double[length];
        for(int i=0; i< length; i++) stateVector[i]=-1;

        stateVector[0]=1;//bias
        stateVector[1]=(argState/32)%4;
        stateVector[2]=(argState/8)%4;
        stateVector[3]=(argState/2)%4;
        stateVector[4]=(argState/1)%2;
        if(argAction==0) stateVector[5]=1;
        if(argAction==1) stateVector[6]=1;
        if(argAction==2) stateVector[7]=1;
        if(argAction==3) stateVector[8]=1;
        if(argAction==4) stateVector[9]=1;


            return stateVector;
    }

    public double getWeightHO(int index){
        return weightHiddenOutput[index];
    }
    public double getWeightIH(int index1, int index2){
        return weightInputToHidden[index1][index2];
    }
    public void setWeightHO(int index, double value){
        weightHiddenOutput[index]=value;
    }
    public void setWeightIH(int index1,int index2, double value){
        weightInputToHidden[index1][index2]=value;
    }
    public void printWeights(){
        for(int i=3; i<8; i++) System.out.print(weightHiddenOutput[i]+"\t");
    }

}//End of public class NeuralNet
```

Appendix F: Source Code – LUT Class

```
package ca.ubc.ece.lut;

import java.io.File;
import java.util.Random;
import java.io.IOException;
import java.math.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;

import robocode.RobocodeFileOutputStream;

/*
 * Lookup table class for reinforcement learning
 * ECE 592, Alison Michan, 2010
 */

public class LUT {
    /*
     * Public attributes of this class. Used to capture the largest Q
     */
    public static int stateMap [][][][];
    public static int numStates=0;
    public static int numActions = 5;
    public static final int NumTargetDistance = 4; //hardcoded in robot
    public static final int NumTargetBearing = 4;  //hardcoded in robot
```

```
public static final int NumHeading = 4; //hardcoded in robot
public static final int isHitWall=2; //hardcoded in robot 0 or 1
public double QTable[][]; // 2-d array for state table
double discountRate;
double learningRate;
double upperBoundQ;
double lowerBoundQ;

/*
 * Constructor.
 */
public LUT(int argNumInputs, int argNumHidden, double argLearningRate,
          double argAlpha, double argLowerQ,double argUpperQ) {
     //argNumInputs and argNumHidden are disregarded for reinforcement
     discountRate=argAlpha;
     learningRate=argLearningRate;
     upperBoundQ=argUpperQ;
     lowerBoundQ=argLowerQ;
     stateMap= new int[NumHeading][NumTargetDistance][NumTargetBearing][isHitWall];
     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 <isHitWall ; d++){
          stateMap[a][b][c][d] = count++;
          System.out.println(a + "\t" +b+ "\t"+c+"\t"+d);
           }
    numStates= count;
    QTable= new double[numStates][numActions];
     initializeLUT();
    System.out.println(numActions);
    System.out.println(numStates);
    System.out.println("exit constructor");
}

/**
 * Initialize the look-up table to all zeros.
 */
private void initializeLUT() {
     for(int a=0;a<numStates;a++)
          for(int b=0; b<numActions; b++)
               QTable[a][b]=0.0;
     System.out.println("exit initialize");
}

/**
 * Returns the QValue of a state/action pair
 */
public double getQValue(int state, int action)
{
     double temp=QTable[state][action];
     return temp;
}
/**
 * Returns the number of states
 */
public int getNumStates()
{
     return numStates;
}
/**
 * Returns the number of actions
 */
public int getNumActions()
{
     return numActions;
}
/**
 * Returns the max QValue of a state (not a state vector)
 */
public double getMaxQValue(int argState)
  {
    double Qtemp = Double.NEGATIVE_INFINITY;
    for (int i = 0; i < numActions; i++)
    {
```

```
        if (QTable[argState][i] > Qtemp)
          Qtemp = QTable[argState][i];
      }
      return Qtemp;
    }

    /**
     * A helper method that translates a vector being used to index the look up
     * table - returns state number from state vector input
     */
    public int indexFor(int[] argVector) {
        int [] stateVector=argVector;
        int state=0 ;
        state+=stateVector[0]*32;
        state+=stateVector[1]*8;
        state+=stateVector[2]*2;
        state+=stateVector[3]*1;

        return state;
    }

    /**
     * Retrieves the value stored in that location of the
     * look up table that output Q value?
     * Returns best action, for Qmax
     */
    public int outputFor(int[] argStateVector) {
        int stateTemp=indexFor(argStateVector);
        //find the Qmax value by finding all Q values of the action pairs
        int actionTemp=0;
        double Qtemp=Double.NEGATIVE_INFINITY;
        for(int i=0; i<numActions; i++){
            if(QTable[stateTemp][i]>Qtemp) {
                Qtemp=QTable[stateTemp][i];
                actionTemp=i;
            }
        }

        return actionTemp; //return best action (maxQ)
    }

    /**
     * Will replace the value currently stored in the
     * location of the look up table
     */
    public void train(int[] argCurrentStateVector, int argCurrentAction, double argReward) {
        int currentState=this.indexFor(argCurrentStateVector);
        double oldQ=this.getQValue(currentState,argCurrentAction);
        double temp = learningRate*(argReward + discountRate * this.getMaxQValue(currentState)-oldQ);
        temp=oldQ+temp;
        if(temp>upperBoundQ) temp=upperBoundQ;
        if(temp<lowerBoundQ) temp=lowerBoundQ;
        QTable[currentState][argCurrentAction]=temp;
    }

    /**
     * Will attempt to write only the 'visited' elements of the look up table
     */
    public void writeToFile(File fileHandle) {


        PrintStream saveFile = null;

        try
        {
            saveFile = new PrintStream( new RobocodeFileOutputStream( fileHandle ));
        }
        catch (IOException e)
        {
            System.out.println( "*** Could not create output stream for LUT save file.");
        }

        //saveFile.println( maxIndex );
        int numEntriesSaved = 0;
        for (int i=0; i<numStates; i++)
```

```
        {
            for(int j=0; j<numActions; j++)
            {
                saveFile.println( i +"\n"+ getQValue(i, j) );
                numEntriesSaved ++;
            }
        }
        saveFile.close();
        System.out.println ( "--+ Number of LUT table entries saved is " + numEntriesSaved );
    }

    public void replace(LUT argLookupQPrevious){
        for(int a=0;a<numStates;a++)
            for(int b=0; b<numActions; b++)
                QTable[a][b]=argLookupQPrevious.getQValue(a,b);
    }
}// End of public class NeuralNet
```