EECE 592 Project Part 2: Reinforcement Learning using a Lookup Table (December 2010)

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

***Abstract*— A Q-learning algorithm was implemented in order to train a robot to fight an opposing robot. Using the iterative process, rewards were generated according to favorable and unfavorable outcomes and the robot adapted its behavior based on a finite number of state action pairs. The impact of adjustments to several parameters on learning convergence was observed. Overall performance was measured by tracking the percentage wins.**

I. Introduction

Reinforcement learning is a method that can be used to train a robot's behavior and includes mapping a set of robot state action pairs to reward. A Q-learning algorithm was implemented and robot behavior in Robocode was investigated.

II. Experimental Results, Question 3

3) Describe your application of RL (Q-learning) to Robocode.

A. Question 3a

3a) Describe in your own words the Q-learning algorithm.

The q-learning algorithm is a reinforcement learning algorithm. The algorithm maps state action pairs to reward outcomes in order to learn a behavior that maximizes reward (in this case, corresponding to a win). It uses an iterative process where a finite set of state action pairs and reward are found by exploration of states and actions. As learning progresses, the expected reward values are continuously updated for state action pairs. A two-dimensional array represents the expected rewards of state action pairs. At the beginning of the learning process, all of the reward Q values are initialized to 0. As the iterative process progresses, the Q values are updated to reflect actual outcomes and thus the robot learns to take actions associated with the best possible action at any given state. In Robocode, instantaneous and terminal rewards are given to teach the robot favorable actions to reach the ultimate goal of learning how to win against an opponent.

B. Question 3b

3b) Describe what task you tried to optimize using RL in terms of states and actions. I.e. What actions did you define? When and what rewards are generated? How large is the natural state and action space associated with your problem definition?

The goal of my optimization was to maximize percentage of wins against an opponent.

The states included robot heading, distance from opponent, bearing to opponent, and a true or false state of hitting a wall. The state vector is shown below:
[NumHeading][NumTargetDistance][NumTargetBearing][isHitWall]

Without reduction in dimensionality, the heading is a value between 0 and 2π, the target distance is a value depending on the arena size, up to a maximum of the diagonal of the arena, and the bearing is a value between $-\pi$ and π. The hit wall state is a Boolean value. Without reduction in number of possible states, the number of states is large. There would be thousands of states.

The number of actions represented included, aim-fire, chase, retreat, fire, and ahead. The complex movements aim-fire, chase, and retreat, were meant to simplify the learning while the more basic movements of fire and ahead were meant to give some independent flexibility to the movement adaptation of the robot.

Both instantaneous and final rewards were generated. Large rewards both positive and negative were generated on wins and losses while smaller rewards were generated on bullet hit or hit by bullet. The value of large rewards was +/- 0.5 and instantaneous rewards were +/-0.02.

C. Question 3c

3c) What steps did you take to reduce the size of the state-action space? How large is the resulting state and action space. (For Part 3 you are required to remove any such reduction, so keep this in mind).

In order to reduce the size of state-action space the states and actions were both quantized.

The number of headings, target distance, and bearing were all reduced to four states while the hit wall flag was either true or

false. The total number of states then is a combination of all of these states and is mapped using a simple base-2 representation thus the number of states is 64.

As a specific example, in the case of heading, the state was quantized using the following method:

[0, π/2) Heading State 0
[π/2, π) Heading State 1
[π, 3π/2) Heading State 2
[3π/2, 2π) Heading State 3

Overall, the state action space was represented as a two-dimensional array of number of states by number of actions and in this case was 64x5.

D. Question 3d

3d)What parameter(s) did you track to measure the progress of learning?

The parameter tracked to measure successful learning was the win rate. Progress of learning was also measured visually by observing change in behavior and an increase in battle time as the robot was a stronger match to the opponent.

E. Question 3e

3e) Draw a graph of a parameter as mentioned in (3d) against number of battles and comment on the convergence of learning

The number of wins converged to an average and oscillated around an average number of wins. Fewer oscillations were observed with a lower resolution. An example of convergence is shown below using epsilon of 0.1, discount rate of 0.2, and learning rate of 0.15. The red oscillation data is the number of wins observed every 100 rounds while the smoother plot is the same win data observed every 1000 rounds.

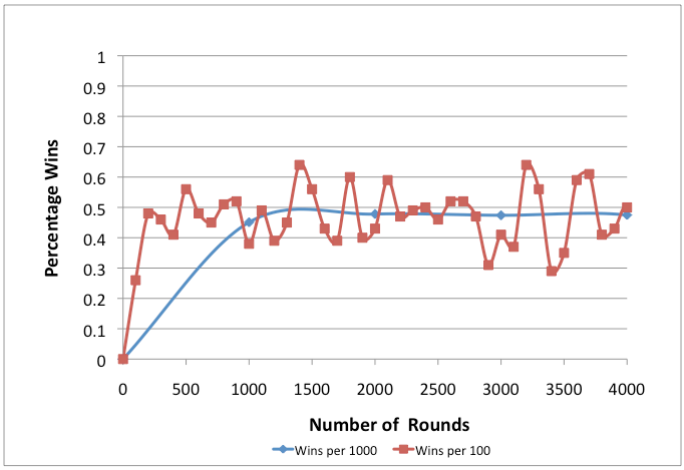

Figure 1: Learning convergence using epsilon 0.1, discount rate of 0.2, and learning rate 0.15 with no decay.

The number of wins started at 0 with no training and demonstrates how the number of wins increase as the robot learns how to win against the opponent. The 50% win rate in this case could be improved with different choices of algorithm parameters.

F. Question 3f

3f) What were the maximum and minimum Q values that were observed during training? How would you go about theoretically predicting what these values might be?

Maximum q-value observed was 0.0152 and minimum q-value observed was -0.078 for 4000 iterations using alpha of 0.15 and epsilon of 0.1. The majority of state action pairs represented in the q-table were visited.

The Q values are the reward values associated with state action pairs. According to Sutton and Barto (2.2), the Q value could be estimated by the sample average method.

$$Q(a) = \frac{r_1 + r_2 + r_3 + \ldots + r_{k_a}}{k_a} \qquad [1]$$

Q is initialized to 0 at ka=0 and as the number of times an action has been chosen ka becomes very large, Q approaches its true value. (Sutton & Barto).

G. Question 3g

3g) Discuss any other parameters that you investigated. E.g. discount factor, learning rate, values of rewards. Provide graphs where appropriate.

One set of experiments was completed to investigate the impact of decaying epsilon and learning rate. A plot showing a comparison of results with decay and no decay is shown below.

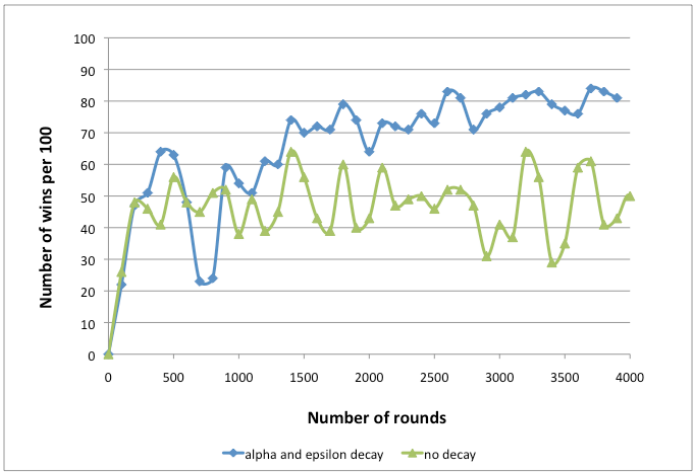

Figure 2: Decaying epsilon and learning rate

Decreasing the epsilon value and learning rate over time improved learning and enabled a smoother convergence to a higher optimal value. Additionally, a larger epsilon could be used in the beginning for increased exploration and then reduced as the number of rounds progressed.

A second set of experiments was completed to investigate the

impact of learning rate adjustments. It was found that a lower learning rate was more optimal than a very high learning rate in achieving the maximum average number of wins although it was slightly slower in converging. The impact of a higher learning rate of 0.4, with no decay, resulted in a lower number of wins compared to lower learning rates of 0.05 and 0.15. Sample results are shown below.

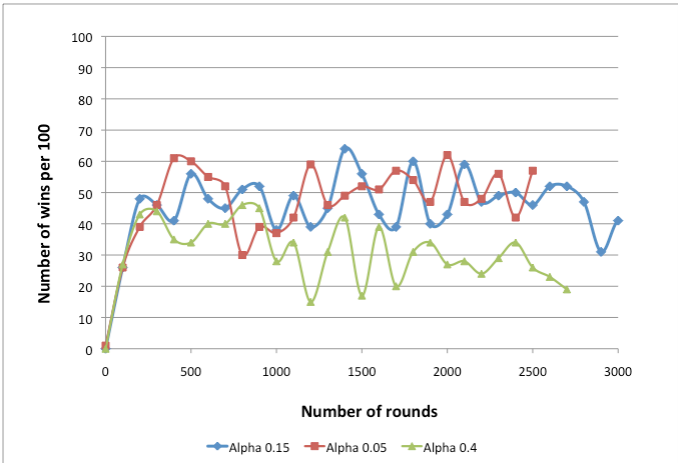

Figure 3: Learning rate investigation, number of wins per 100 rounds

The same results were analyzed, as number of wins per 500, and in the figure below it is easier to see the overall effect of increasing the size of learning rate.

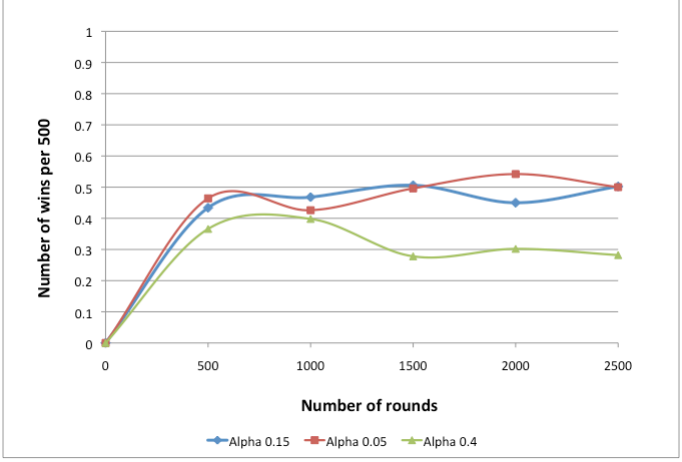

Figure 4: Learning rate investigation, number of wins per 500 rounds

Additionally, values of instantaneous rewards were changed slightly but did not seem have too much sensitivity with a discount rate of 0.2. For instance, changing an instantaneous reward from 0.05 to 0.02 did not have a very large impact compared to the impact of decaying. Discount value controls how much future rewards or instantaneous rewards are weighted. The algorithm reached a higher optimal performance with a lower discount value and 0.2 produced a reasonable result. A discount value of 0.4 converged at a lower win rate compared with 0.2.

H. Question 3h

3h) Describe your visual observations of how your robot's behavior improves against your chosen enemy. Does it always learn the same behavior or are there other alternative behaviors that are reached for different training sessions? What about behaviors against different enemies? (Your answer here needs only to be qualitative and can be based purely on visual observation).

The behavior adapted against a single robot was observed to converge to the same set of behaviors each time. For instance, training the robot against Ramfire always ended up in a behavior where the robot would end in a set of actions that was spin, back up, fire, repeat. In contrast, the behaviors developed against the sample robot Corners ended in a behavior where the robot would generally stay in the middle and fire at the corner continuously. These differences were captured in the final q values.

It was interesting to observe how the behavior of the robot adapted to the opponent with number of iterations. At first the robot actions appeared random and unsuccessful. Next, there was a phase where the robot would take successful actions in certain states. Finally, as the number of rounds progressed, the robot began putting together series of actions that were even more successful against the opponent.

I. Question 3i

3i) Do you think that learning would be easier/better without dimensionality reduction?

If reduction in dimensionality were not utilized, it would make the problem more difficult to solve. There would be a very large state table and the q values would be updated infrequently since the states and actions would not be generalized. As the number of state action pairs become very large, it is understood that additional techniques would need to be applied than a simple lookup table.

III. EXPERIMENTAL RESULTS, QUESTION 2

4) In RL, a policy is realized by a value function.

A. Question 4a

4a) Describe what the term policy in RL generally means?

According to Sutton and Barto, 1998, "A *policy* defines the learning agent's way of behaving at a given time." In other words, the policy is the rule for determining an action given a state.

B. Question 4b

4b) The value function implements a mapping from perceived states or state- action pairs to actions. Describe in your own words, what this value function actually represents. Describe in your own words, what TD learning is doing.

The value function maps states or state-action pairs to long-term reward. Each value represents expected future reward if an action is taken given a current state (Sarkaria, Sutton &

Barto). Temporal-difference (TD) is a method to continuously update values for states or state-action pairs. The method continuously updates an estimate of future reward by using a difference between the previous and current estimate to predict an improved estimate. Eventually this bootstrapping method allows convergence to the true value.

C. Question 4c

4c) What is the difference between continuous and episodic tasks? Also what is the difference between deterministic and non-deterministic tasks? Which of these does Robocode fall into?

As described by Sarkaria in class course notes, episodic tasks have an end state where a reward is generated at a defined terminal state. In contrast, continuous tasks can potentially go on for a very long time or forever and a reward generated at the terminal state is not effective. In continuous tasks it is necessary to generate instantaneous rewards, before the terminal state. Robocode is a continuous task.

D. Question 4d

4d) Describe the difference between Q-learning and "V-learning". This assignment asked you to implement Q-Learning, but do you think "V- learning" could be applied to Robocode instead?

V-learning can be thought of as a model of state transitions in an environment (Sarkaria). In Robocode, the next state is not known since it is a dynamic environment depending on the actions of other robots, unlike a game of tic-tac-toe. V-learning cannot be applied to Robocode because the next state is not known. Q-learning on the other hand maps state-action pairs to a long-term reward.

E. Question 4e

4e) In Robocode, what do you think would happen if you wait for rewards only in terminal states?

The instantaneous rewards help the robot to learn more optimal behaviors. Providing rewards only at the terminal state would not guarantee that the robot would reach its maximum potential or learn optimal behaviors. It would also increase learning time.

IV. Experimental Results, Question 5

While training via RL, the next move is selected randomly with probability ε and greedily with probability 1 − ε

A. Question 5a

5a) How well does your robot perform when moves are greedy only?

If greedy moves were always used, learning was unsuccessful in reaching an optimal performance and was observed to oscillate around a low win rate. A sample of observed behavior relating the number of wins versus number of rounds is shown below for an epsilon of 0.

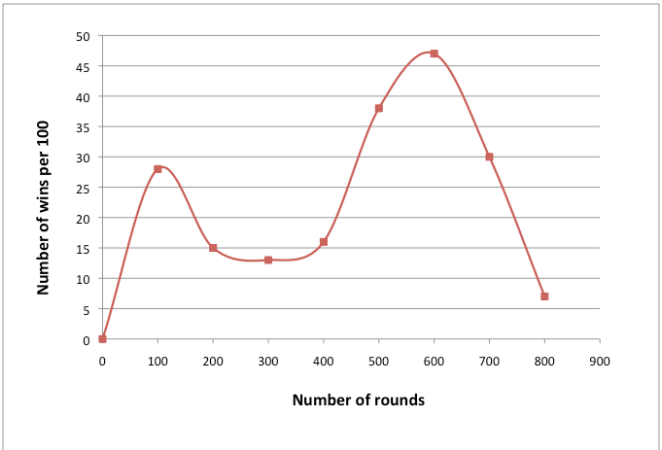

Figure 5: Sample of behavior with epsilon 0

B. Question 5b

5b) Explain why exploratory moves are necessary.

Exploratory moves are necessary for the robot to learn all the different rewards associated with state-action pairs; if there are no exploratory actions the robot will not necessarily visit all state-action pairs. The chance of becoming trapped in a local maximum win rate without ever reaching the global maximum win rate is increased without a sufficient rate of exploratory moves.

C. Question 5c

5c) What is the optimal value for ε? Provide a graph of the measured performance of your tank vs ε.

Several epsilon values were tested and the figure below summarizes the results over 2500 iterations. The lower epsilon values explored less and reached an optimal value at a slower rate. A higher epsilon value meant the robot explored more frequently and converged at a faster rate. An epsilon value of 0.1 resulted in a high win rate over thousands of rounds. In the case where the epsilon value was decayed over time, a larger initial epsilon value could be used that helped to learn at a fast rate and reach a higher optimal number of wins (Figure 2). When epsilon decay was used, a higher initial epsilon value of 0.2 was optimal. Performance using different epsilon values and keeping other parameters fixed is shown below.

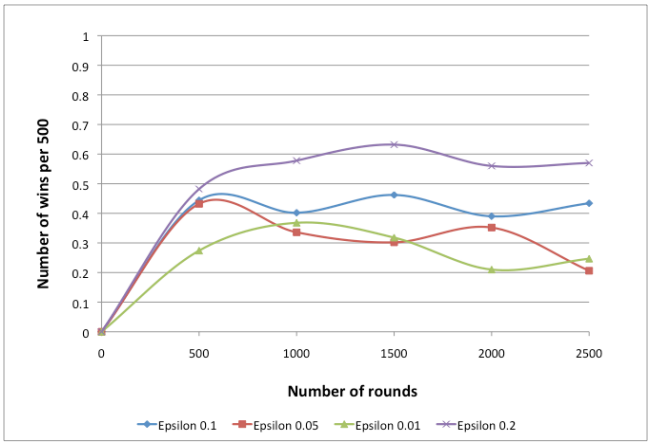

Figure 6: Peformance of robot with increasing values of epsilon

V. CONCLUSION

A Q-learning algorithm was implemented to relate robot state-action pairs to rewards so that the robot could learn how to win against an opponent. Behavior adapted differently depending on the opponent and reached a maximum potential, quantified in percentage wins, after a few thousand iterations. Epsilon-greedy exploratory moves were introduced to help with learning and different learning rates were tuned for optimal convergence.

APPENDIX

LUT and Robot Java code attached, Appendix A

REFERENCES

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

Appendix A: Source code, LUT class and Robot 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=1; //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++;
	   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=stateVector[3]; //just two states for wall make it first to sum
        for(int i=0; i<3; i++){
            int base=(int) Math.pow(2, i+1);
            int temp=base*stateVector[i];
            state=state+temp;
        }
        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
```

```
package ca.ubc.ece.lut;
import java.awt.Color;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
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 2 - Reinforcement Learning
 * Alison Michan, December 2010 - robot uses reinforcement learning with LUT class
 */

public class Daphne extends AdvancedRobot {
	//declarations
	static double learningRateOrig=0.15;
	static double learningRate=0.15;
	static LUT lookupQ = new LUT(2, 12, learningRate, 0.2,-1.0, 1.0); //constructor, STATIC
	//static LUT lookupQPrevious = new LUT(2, 12, 0.15, 0.2,-1.0, 1.0); //constructor, STATIC
	int [] stateVector = new int[4]; //hard code numStates
	int [] currentStateVector = new int[4]; //hardcode stateVector
	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 = 1;		//initialize action a'
	int previousAction = 1;		//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 double epsilonGreedy=0.1; //initialize epsilonGreedy
	static double epsilonGreedyOrig=0.1;
	static int total=0;

	public void run() {
		setBodyColor(Color.pink);	//set colors
		setGunColor(Color.gray);
		setRadarColor(Color.pink);
		currentStateVector[0]=0;//initialize stateVector
		currentStateVector[1]=0;
		currentStateVector[2]=0;
		currentStateVector[3]=0;
		stateVector[0]=0;	//initialize stateVector
		stateVector[1]=0;
		stateVector[2]=0;
		stateVector[3]=0;
		setAdjustGunForRobotTurn(false);
		setAdjustRadarForGunTurn(true);

		while (true) {
			double epsilonGreedyTemp = Math.random(); //epsilon greedy
			if(epsilonGreedyTemp<epsilonGreedy)
			 {
				double temp=Math.random();
				if(temp<=0.2) currentAction = 0;
				if(temp<=0.4) currentAction = 1;
				if(temp<=0.6) currentAction = 2;
				if(temp<=0.8) currentAction = 3;
				if(temp<=1.0) currentAction = 4;
			 }

			if(currentAction==0) //aim fire

			{
				double temp = currBearing;
				if(temp>0) turnLeftRadians(-1*temp);
				if(temp<0) turnRightRadians(temp);
				fire(1);
			}

			if(currentAction==1) //chase

			 {
```

```
                double temp = currBearing;
                if(temp>0) turnLeftRadians(-1*temp);
                if(temp<0) turnRightRadians(temp);
                ahead(currDistance + 15);
            }

            if(currentAction==2) //spin retreat fire
            {
                turnLeftRadians(3.14/2); //spin
                back(125);
                fire(1);
            }
            if(currentAction==3) //ahead
            {
                ahead(50);
            }

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

            turnRadarRightRadians(6.28);
            if(trainFlag) lookupQ.train(currentStateVector, currentAction, reward ); //calculate and update Q (train)
            reward=0.0;//re-set reward
            isHitWall=0;
            currentStateVector=stateVector; //update currentState s to s'
            previousAction=currentAction; //keep track of action a
            currentAction = lookupQ.outputFor(currentStateVector);//update currentAction a to a'
            //out.println("Action: " + currentAction);
            //out.println("State: " + currentStateVector[0]+" "+currentStateVector[1]+" "+currentStateVector[2]+" "+currentStateVector[3]);
        }
}

    public void onScannedRobot(ScannedRobotEvent e) {
        stateVector[0]=getHeading(e.getHeadingRadians());
        currHeading=getHeadingRadians();
        stateVector[1]=getTargetDistance(e.getDistance());
        currDistance=e.getDistance();
        stateVector[2]=getTargetBearing(e.getBearingRadians());
        currBearing=e.getBearingRadians();
        currRadar=getRadarHeadingRadians();
        stateVector[3]=isHitWall;
        //System.out.println(stateVector);
    }
    public void onHitByBullet(HitByBulletEvent e) {
        reward=reward+SNReward;
    }

    public void onBulletHit(BulletHitEvent e) {
        reward=reward+SReward;
    }

    public void onHitWall(HitWallEvent e)
     {
       reward=reward+SNReward/2;
       isHitWall=1;
     }

    public void onDeath(DeathEvent e) {
        reward=reward+BNReward;
        if(trainFlag) lookupQ.train(currentStateVector, currentAction, reward ); //calculate and update Q (train)
        reward=0;
        isHitWall=0;
    }

    public void onWin(WinEvent e) {
        reward=reward+BReward;
        if(trainFlag) lookupQ.train(currentStateVector, currentAction, reward ); //calculate and update Q (train)
        reward=0;
        isHitWall=0;
        numWonRounds++;
    }

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

    public int getHeading(double arg)
    {
        //4 states
        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 states
        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("Round ended!");
        System.out.println("assigned "+ lookupQ.getQValue(1, 0)+ " "+lookupQ.getQValue(1, 2));
        System.out.println("trainFlag:"+trainFlag+" numTrainRounds:" + numTrainRounds + " numTestRounds:"
                + numTestRounds +" numWon" +numWonRounds+ "Previous" +numWonRoundsPrev
                + "Greedy" +epsilonGreedy + "totalRounds" +total+ "alpha" +lookupQ.learningRate);

        if(trainFlag) {
            if(numTrainRounds==100) { //alternate between training and no training in multiples
                numTrainRounds=0;
                saveWinsTrain=saveWinsTrain+Integer.toString(numWonRounds);
                saveWinsTrain=saveWinsTrain+"\t";
                numWonRounds=0;
                trainFlag = false;

            } else{

                numTrainRounds++;

                //use to exponentially decay epsilonGreedy and/or learning rate
                if(epsilonGreedy>0.06) epsilonGreedy=epsilonGreedyOrig*Math.exp(-total/400);
                //lookupQ.learningRate=learningRateOrig*Math.exp(-total/400);
            }

        } else {

            if(numTestRounds==10) { //alternate between training and no training in multiples
                numTestRounds = 0;
                //use if want to only accept 'improved' q-table
                //if(numWonRoundsPrev>numWonRounds){
                //    lookupQ.replace(lookupQPrevious);
                //    }
                //else{
                //    lookupQPrevious.replace(lookupQ);
                //    numWonRoundsPrev=numWonRounds;
                //    }

                saveWins=saveWins+Integer.toString(numWonRounds);
                saveWins=saveWins+"\t";
                numWonRounds=0;
                trainFlag = true;

            } else{
                numTestRounds++;
                total++;
            }

        }

        System.out.println("assigned "+ lookupQ.getQValue(1, 0)+ " "+lookupQ.getQValue(1, 2));

    }
    /**
     * 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!");
        PrintStream writer;
        try {

            writer = new PrintStream(new RobocodeFileOutputStream(getDataFile("log.csv")));
            String saveFile="";
            for(int a=0;a<lookupQ.getNumStates();a++){
                saveFile+="\n";
                for(int b=0; b<lookupQ.getNumActions(); b++)
                    saveFile+=(lookupQ.getQValue(a,b)+ "\t");
            writer.append(saveFile);
           writer.close();
        } catch (IOException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        try {
            writer = new PrintStream(new RobocodeFileOutputStream(getDataFile("wins.csv")));
            writer.append(saveWins +"\n" + saveWinsTrain);
           writer.close();
        } catch (IOException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();

        }
    }
}
```