EECE 592 Coursework Part 3 RL with BP

Name: Weirui Kong Student Number: 95892162

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.

Solution:

The neural net used in my robocode tank has five input neurons, 10 hidden neurons and 5 output neurons, which means it's a net with a Q output for each action. The diagram, with S's denoting inputs and Q's denoting outputs, is as follows:

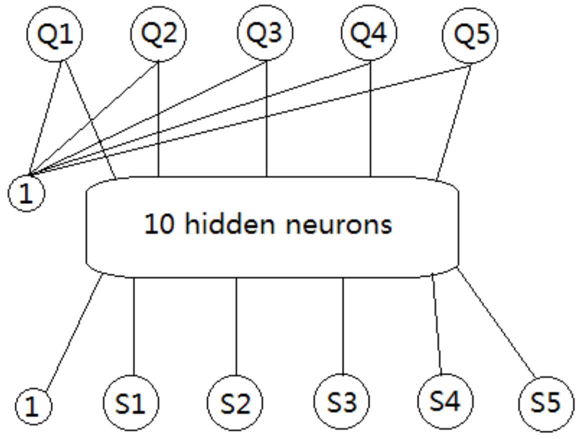

Figure 1

As for input representations, I don't use any encodings like one hot, but merely scale the real-valued inputs. The reason for scaling is that very positive/negative values saturate the sigmoid function, which kills the gradients, harming the BP process. The following chart shows how scaling effects.

Input	Original Range	After scaling
X	[-400, 400]	[-0.4, 0.4]
Y	[-300, 300]	[-0.3, 0.3]
DeltaEnergy	[-200, 200]	[-0.1, 0.1]
DeltaX	[-800, 800]	[-0.8, 0.8]
DeltaY	[-600, 600]	[-0.6, 0.6]

Chart 1 Input Scaling

On the other hand, in Part 2 I quantize the inputs in this way:

Input	Original Range	After quantization
X	[-400, 400]	-4, -3, -2, -1, 1, 2, 3, 4
Y	[-300, 300]	-3, -2, -1, 1, 2, 3
DeltaEnergy	[-200, 200]	-1, 1
DeltaX	[-800, 800]	-8, -7, …, -1, 1, … 7, 8
DeltaY	[-600, 600]	-6, -5, …, -1, 1, … 5, 6

Chart 2 Input Quantization in Part 2

To match the magnitude, the state values stored in the look up table are all divided by 10. Then the whole table is used to train the neural net, with states as inputs and Q-values of each actions as outputs.

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.

Solution:

Setting number of hidden neurons to 10 and momentum 0.9, Figure 2 shows how learning rate affects learning performance, with y-axis denoting the squared error and x-axis denoting the number of epochs. One epoch means a complete train through the whole 6048 entries in the look up table.

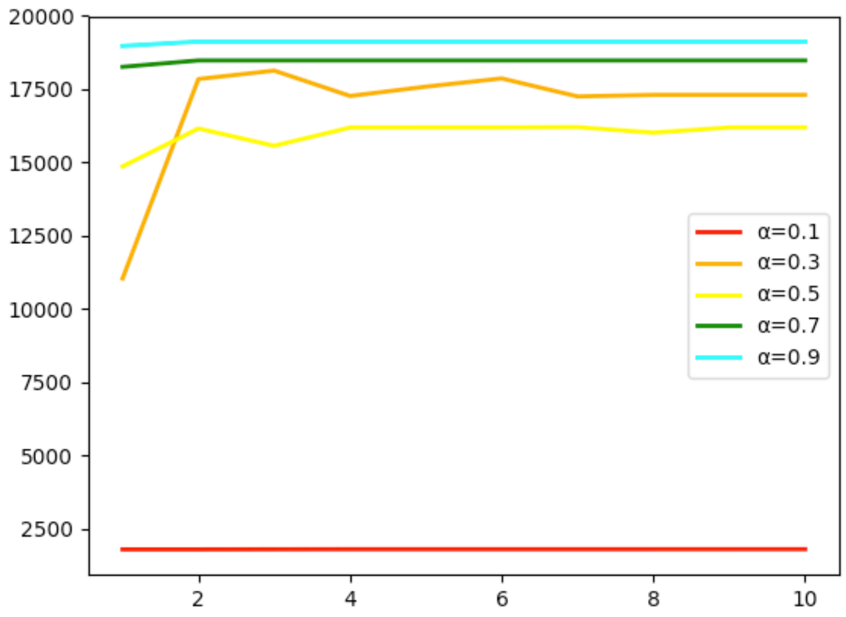

Figure 2

Figure 2 suggests that for most of the learning rate, the algorithm converges within 3 epochs and a learning rate of 0.1 achieves the lowest squared error.

To figure out how many hidden neurons lead to best generalization performance, I divide the look up table into training set (65%) and validation set (35%). Setting learning rate to 0.1 and momentum to 0.9, Figure 3 shows how different number of hidden neurons affects learning performance, with y-axis denoting validation error and x-axis denoting number of epochs.

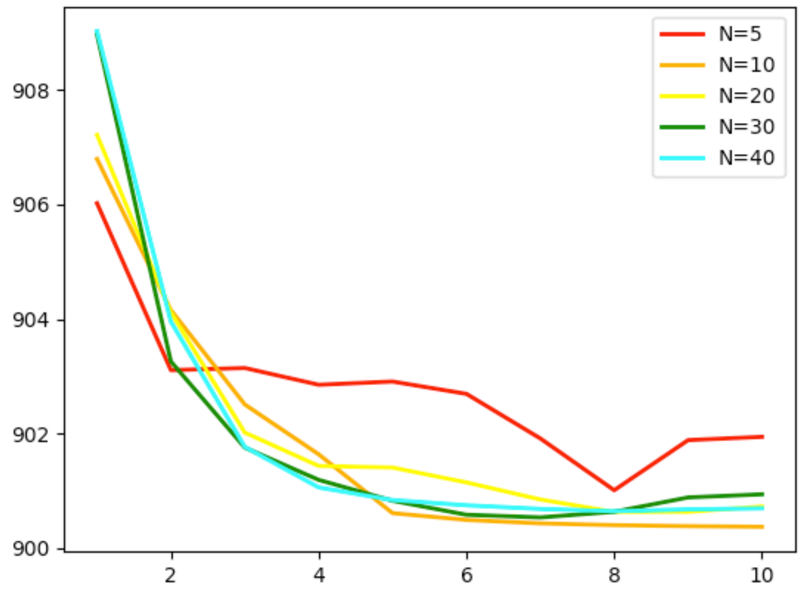

Figure 3

As Figure 3 indicates, 10 hidden neurons (orange line) achieves best validation error. Besides, despite that a large N can also achieve a relatively low validation error after converging, the validation error of first epoch keeps increasing as N getting larger. Therefore, I pick 10 to be the number of hidden neurons in my neural net.

Finally, we need to choose the value of momentum. Setting learning rate to 0.1 and number of hidden neurons to 10, Figure 4 shows how momentum value affects learning performance, with y-axis denoting training error and x-axis denoting number of epochs.

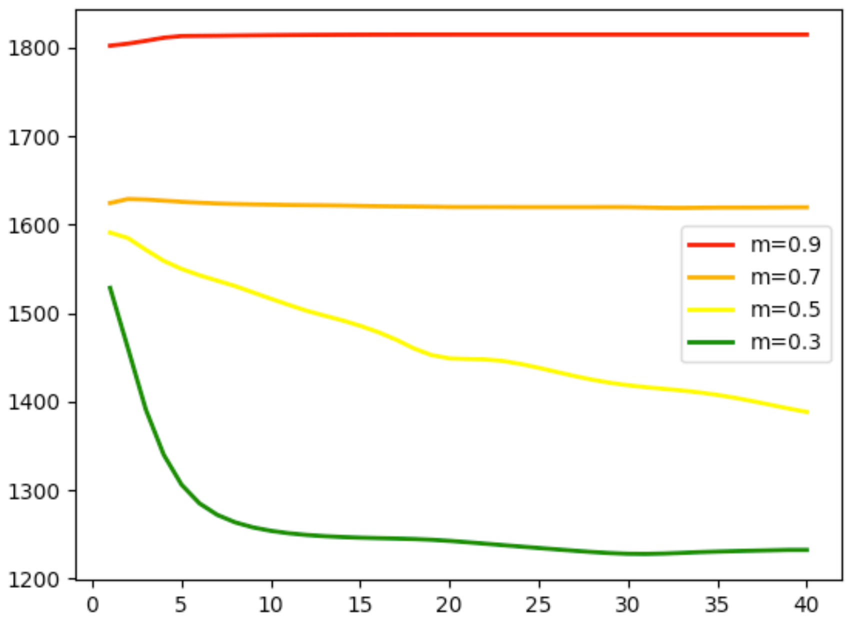

Figure 4

Although a small momentum decreases the training error, it also takes more epochs to converge. A value of 0.7 converges rapidly and achieves a good error, thus I pick 0.7 to be the momentum value in my neural net.

In conclusion, the values of hyper-parameters of the neural net are as follow: number of hidden neurons N = 10; learning rate α=0.1 and momentum m = 0.7. The RMS error is

$$\sqrt{\frac{1619}{6048}} = 0.517$$

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.

Solution:

In Part 2 I quantize the inputs in this way:

Input	Original Range	After quantization
X	[-400, 400]	-4, -3, -2, -1, 1, 2, 3, 4
Y	[-300, 300]	-3, -2, -1, 1, 2, 3
DeltaEnergy	[-200, 200]	-1, 1
DeltaX	[-800, 800]	-8, -7, …, -1, 1, … 7, 8
DeltaY	[-600, 600]	-6, -5, …, -1, 1, … 5, 6

And in Part 3 neither quantization nor dimensionality reduction is applied, but merely scaling the state as follows:

Input	Original Range	After scaling
X	[-400, 400]	[-0.4, 0.4]
Y	[-300, 300]	[-0.3, 0.3]
DeltaEnergy	[-200, 200]	[-0.1, 0.1]
DeltaX	[-800, 800]	[-0.8, 0.8]
DeltaY	[-600, 600]	[-0.6, 0.6]

For an input state like (X, Y, DeltaEnergy, DeltaX, DeltaY) = (-123.64, 258.23, -14.68, -110.10, 233.33), the representation for Part 2 is (-2, 3, -1, -2, 3), whose output is (-0.350, 0.131, 0, -0.455, 0). The representation for Part 3 is (-0.12364, 0.25823, -0.00743, 0.1101, 0.23333), and the output is something like (-0.350, 0.131, 0, -0.455, 0).

Now let's compare the performance of LUT based Q learning and Neural net based Q learning. RamFire is picked as the enemy tank. As Figure 5 shows, the y-axis denotes the wining rate (i.e. rounds that my robot won/total rounds played) and the x-axis is the rounds played (since I get a result every 500 rounds, an x value of 10 means there are 10*500=5000 rounds in total).

The first 5000 rounds (corresponds to x ranging from 0 to 10 in the graph) I set learning rate α of Q learning to zero, so there would be no learning and robots just took random actions. The next 5000 rounds (corresponds to x ranging from 10 to 20 in the graph) I set learning rate α to 0.7. The next 5000 rounds (corresponds to x ranging from 20 to 30 in the graph) I decrease the learning rate by a factor of 10, which is now 0.07. The next 5000 rounds (corresponds to x ranging from 30 to 40 in the graph) I decrease the learning rate again, by a factor of 10, which is now 0.007. Decreasing the learning rate is performed both on robot using LUT and robot using neural net, so it's a fair comparison.

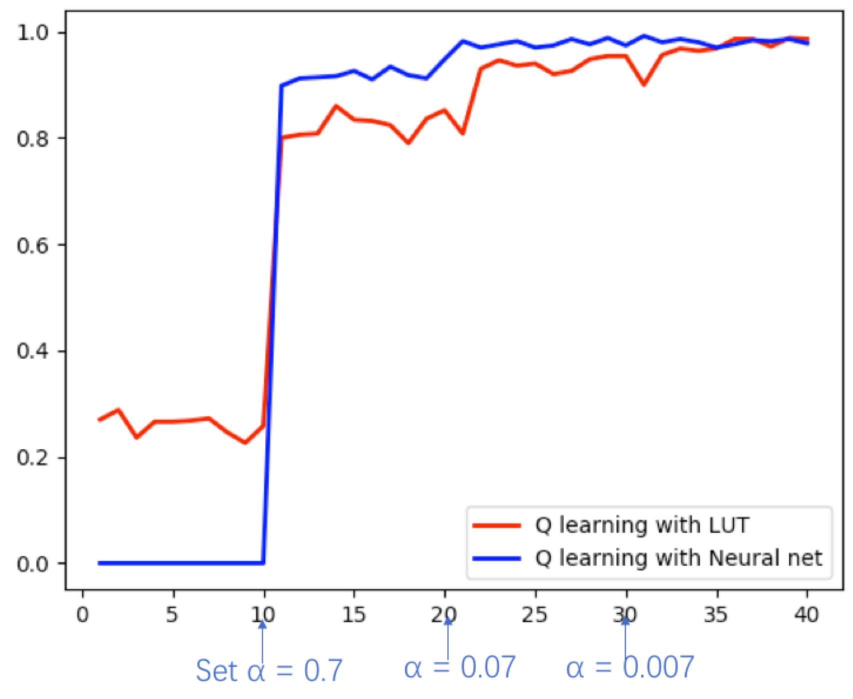

Figure 5

As Figure 5 shows, the wining rate of Neural net based Q learning is almost always higher than that of LUT based Q learning. Using a neural net to approximate Q value also makes the algorithm converge faster. The reason is that, without any quantization or dimensionality reduction, a neural net based Q learning is able to explore more states than the LUT based Q learning. It's more accurate with respect to the state space representation, which helps to make a better performance.

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.

Solution:

For real world problems where the state space is so large, it's impossible to store these values as a look up table. State space reduction solves the problem while sacrificing accuracy to some extent. However, large storage space is no longer a necessity for Q-function approximation. We can simply use raw states as inputs to the model (e.g. a neural net) and get an output of Q value. All we need to store is the weights of the neural net, which doesn't need much storage space. Thus, it would not necessarily need the same level of state space reduction as a look up table.

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.

Solution:

As Figure 7 shows, the best win rate observed is more than 98.5%. Here's the summary about my neural robotank:

For the neural net part, there are 5 inputs, 10 hidden neurons and 5 outputs. Learning rate is 0.1 and momentum is 0.7.

As for the Q learning part, I set epsilon to 0.1 and discount factor to 0.9. The learning rate starts with 0.7 and decreases by a factor of ten when algorithm converges. The five actions for robotank is ahead, back, left, right and fire. Rewards are generated when events like hit by bullet, fire miss etc. happen.

Figure 6 shows the e(s) plot. The graph on the left corresponds to the state action pair $(s, ahead)$, and graph on the right corresponds to $(s, fire)$. I pick a specific state s as the neural net input and get the values for each action at the end of every battle. The x-axis denotes the number of rounds played. As can be seen, the e(s) plot can be viewed as a measure for algorithm convergence.

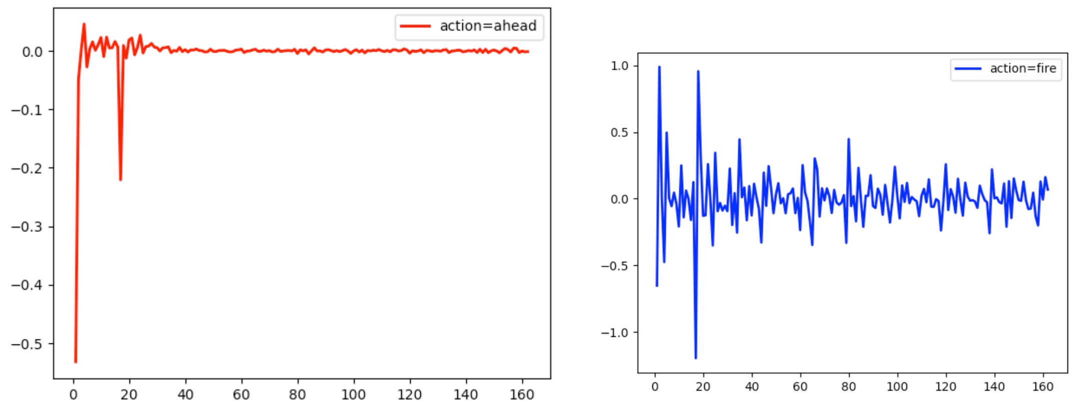

Figure 6

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

Solution:

The graph is Figure 7, with y-axis denoting the wining rate and the x-axis denoting the number of battles played. An x value of 1 means 5000 rounds played.

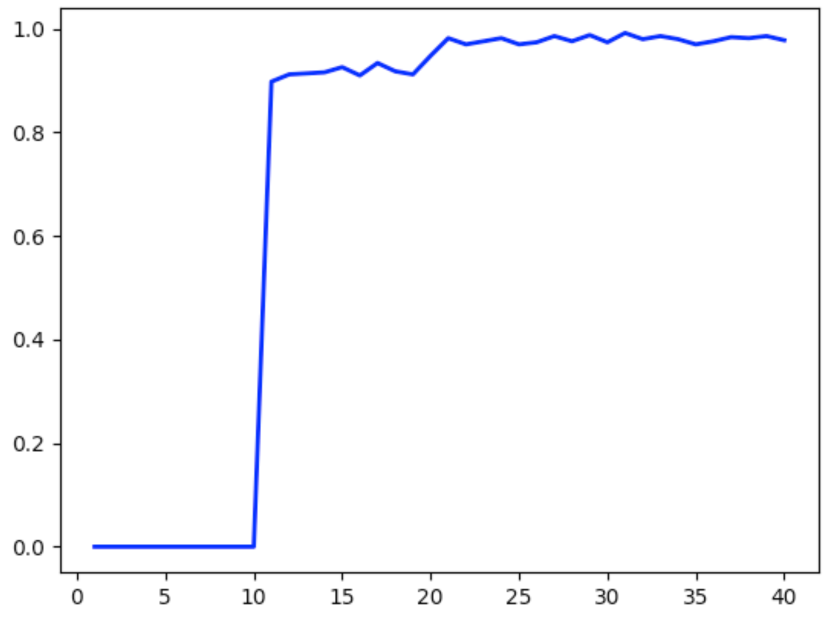

Figure 7

As the figure indicates, regardless of some small fluctuations, the winning rate improves asymptotically. It’s getting closer and closer towards a probability of 1.

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.

Solution:

Let the error in the value function at any given state be represented by $\mathrm{e}(s_t)$. Then $\mathrm{Q}(s_t, a) = \mathrm{e}(s_t) + Q^*(s_t, a)$ and $\mathrm{Q}(s_{t+1}, a) = \mathrm{e}(s_{t+1}) + Q^*(s_{t+1}, a)$.

Using the Bellman equation: $\mathrm{Q}(s_t, a) = r_t + \gamma\mathrm{Q}(s_{t+1}, a)$. We can derive that $\mathrm{e}(s_t) = \gamma\mathrm{e}(s_{t+1})$. When reaching the terminal state, we know that the reward is known precisely, i.e., there is no error in the reinforcement signal. Thus $\mathrm{e}(s_T) = 0$ where s_T is terminal. Given $\mathrm{e}(s_t) = \gamma\mathrm{e}(s_{t+1})$, we deduce that with enough sweeps through the state space, I.e. enough opportunities to learn, we will eventually reach the condition where $\mathrm{e}(s_t)$ is zero for all t. When this is the case, then we know we have the optimal value function: $\mathrm{Q}(s_t, a) = Q^*(s_t, a)$. This means the convergence of the algorithm.

On the other hand, function approximation brings another error in Q values. The error is common to any regression problem: the predicted value is never the same with the "true" value. It's this error that violates the convergence guarantee.

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.

Solution:

We can keep track of the learning performance by using experience replay. Specifically, every time the robot is in a new state, we store the previous state, reward, action and current state. We can implement it as a FIFO to control its capacity. These backups are the experiences of the robot and can be used to train the neural net as a mini batch. Also we can obtain a training error and by comparing these errors overtime, we can determine whether the neural net is making any progress.

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?

Solution:

As for BP performance, it's important to scale the inputs to the neural net. The reason for scaling is that very positive/negative values saturate the sigmoid function, which kills the gradients, harming the BP process. The initial learning rate α is also critical, because if α is too large, after a few updates the magnitude of weights would also become large. Hence the input to a hidden neuron would become very positive/negative. Similarly, it would lead to pretty bad learning performance. Besides we need a validation set to find the proper number of hidden neurons since the training error may keep decreasing with more hidden neurons, which is likely to overfit.

As for RL problem, it's OK to pick a relatively large learning rate so that we can get close to the optimal value rapidly. When reaching convergence, decrease the learning rate so that we can get closer and closer to the optimal value.

When applying neural net based RL to other applications, it's helpful to use the tabular approach first and then pre-train the neural net with the learned look up table. The reason why this is beneficial is that pre-train helps to set proper values to those hyper-parameters of the neural net, which is essential to the accuracy of approximation.

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.*

Solution:

The amount of anesthetic is vital to the patient. If the dose if insufficient, patients would suffer great pain during surgery. If overdosing, it may even endanger the lives of patients. We can't apply RL directly to the patient at the very beginning since it may bring pains to tens of thousands of patients before the algorithm converges. Just like using RL to make a self-driving helicopter, we can't afford the cost of multiple crashes. One possible solution is to build a simulator for the patient. If the simulator can reflect the conditions of patients after anesthetizing accurately, then we can apply RL to this simulator and get a controller probably suitable for real patients.

Appendix: Source Code

```
package weirui;

import java.awt.Color;
import java.awt.geom.Point2D;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

import robocode.AdvancedRobot;
import robocode.BulletHitEvent;
import robocode.BulletMissedEvent;
import robocode.DeathEvent;
import robocode.HitByBulletEvent;
import robocode.HitRobotEvent;
import robocode.HitWallEvent;
import robocode.Rules;
import robocode.ScannedRobotEvent;
import robocode.WinEvent;
import robocode.util.Utils;

public class BetaCat extends AdvancedRobot {
    NeuralNet neuralNet = new NeuralNet(5, 10, 5, 0.1, 0.9, -1, 1);

    static long round = 0;
    static long win = 0;
    static double accumulatedPosReward = 0;
    static double accumulatedNegReward = 0;

    boolean exploring = false;
    double epsilon = 0.9;

    double X;
    double Y;
    double deltaEnergy;
    double deltaX;
    double deltaY;

    double distance;
    double velocity;
    double enemyHeading;

    double[] previousState = new double[5];
    int previousAction;
    double reward = 0;
    double gamma = 0.9;
    double learningRate = 0.7;
    boolean getReward = false;

//  List<double[]> experience = new ArrayList<>();

    Random random = new Random();
    long fireTime;

    public void run() {
        // Set colors
```

```
setBodyColor(Color.red);
setGunColor(Color.orange);
setRadarColor(Color.blue);
setScanColor(Color.yellow);

setAdjustGunForRobotTurn(true);
setAdjustRadarForGunTurn(true);

fireTime = 0;

  try {
      neuralNet.load("weights.txt");
} catch (IOException e) {
      e.printStackTrace();
}
  double[] s;
  int A;
  double[] qt;
  double[] sPrime;
  double[] qtPlusOne;
// Loop forever
while (true) {
      if (getRadarTurnRemaining() == 0.0) {
              setTurnRadarRightRadians(Double.POSITIVE_INFINITY);
      }
      s = setState(X, Y, deltaEnergy, deltaX, deltaY);
      previousState = Arrays.copyOfRange(s, 0, s.length);
      s = scale(s);
      qt = neuralNet.outputsFor(s);
      if(exploring) {
        if(random.nextDouble()<epsilon) {
                A = getMax(qt);
        } else {
                A = getRandom();
        }
} else {
        A = getMax(qt);
}

switch (A) {
        case 0:
                setAhead(100);
                break;
        case 1:
                setBack(100);
                break;
        case 2:
                setTurnLeft(90);
                setAhead(100);
                break;
        case 3:
                setTurnRight(90);
                setAhead(100);
                break;
        case 4:
                doGun();
                break;
        default:
                break;
}
```

```
                previousAction = A;
                execute();
                sPrime = setState(X, Y, deltaEnergy, deltaX, deltaY);
                sPrime = scale(sPrime);
                qtPlusOne = neuralNet.outputsFor(sPrime);
                int a = getMax(qtPlusOne);
                if(!getReward) {
                    reward = 0;
                } else {
                    getReward = false;
                }
                double delta = learningRate * (reward+gamma*qtPlusOne[a]-qt[A]);
                qt[A] += delta;
                neuralNet.train(s, qt);

                // Add (st,a,r,st+1) to the experience data set
                double[] e = new double[s.length+1+1+sPrime.length];
                System.arraycopy(s, 0, e, 0, s.length);
                e[s.length] = A;
                e[s.length+1] = reward;
                System.arraycopy(sPrime, 0, e, s.length+2, sPrime.length);
//              experience.add(e);

                // Experience replay: learn from ten random prior steps
//              for(int i = 0; i < Math.min(experience.size(),9); i++) {
//                  double[] exper = experience.get(random.nextInt(experience.size()));
//                  s = Arrays.copyOfRange(exper, 0, s.length);
//                  A = (int)exper[s.length];
//                  reward = exper[s.length+1];
//                  sPrime = Arrays.copyOfRange(exper, s.length+2, exper.length);
//
//                      qt = neuralNet.outputsFor(s);
//                      qtPlusOne = neuralNet.outputsFor(sPrime);
//                      a = getMax(qtPlusOne);
//                      delta = learningRate * (reward+gamma*qtPlusOne[a]-qt[A]);
//                      qt[A] += delta;
//                      neuralNet.train(s, qt);
//                }
          }
     }

     public double[] setState(double X, double Y, double deltaEnergy, double deltaX,
double deltaY) {
          double[] s = {X,Y,deltaEnergy,deltaX,deltaY};
          return s;
     }

     public void doGun() {
          double power = Math.min(400/distance, 3);
           if (fireTime == getTime() && getGunTurnRemaining() == 0) {
                setFire(power);
           }
          double bulletSpeed = 20 - power*3;
          long time = (long)(distance/bulletSpeed);
          double futureX = getX()-deltaX+Math.sin(enemyHeading)*velocity*time;
          double futureY = getY()-deltaY+Math.cos(enemyHeading)*velocity*time;
          double absDeg = absoluteBearing(getX(), getY(), futureX, futureY);
          setTurnGunRight(normalizeBearing(absDeg-getGunHeading()));
           fireTime = getTime()+1;
     }
```

```
double absoluteBearing(double x1, double y1, double x2, double y2) {
    double xo = x2-x1;
    double yo = y2-y1;
    double hyp = Point2D.distance(x1, y1, x2, y2);
    double arcSin = Math.toDegrees(Math.asin(xo / hyp));
    double bearing = 0;

    if (xo > 0 && yo > 0) { // both pos: lower-Left
        bearing = arcSin;
    } else if (xo < 0 && yo > 0) { // x neg, y pos: lower-right
        bearing = 360 + arcSin; // arcsin is negative here, actuall 360 - ang
    } else if (xo > 0 && yo < 0) { // x pos, y neg: upper-left
        bearing = 180 - arcSin;
    } else if (xo < 0 && yo < 0) { // both neg: upper-right
        bearing = 180 - arcSin; // arcsin is negative here, actually 180 + ang
    }
    return bearing;
}

double normalizeBearing(double angle) {
    while (angle >  180) angle -= 360;
    while (angle < -180) angle += 360;
    return angle;
}

public double scale(double x, double a, double b, double r) {
    return r * (2*x-(a+b)) / (b-a);
}
```

```
public double[] scale(double[] x) {
    double[] scaledx = new double[x.length];
    scaledx[0] = scale(x[0]-400,-400,400,0.4);
    scaledx[1] = scale(x[1]-300,-300,300,0.3);
    scaledx[2] = scale(x[2],-200,200,0.1);
    scaledx[3] = scale(x[3],-800,800,0.8);
    scaledx[4] = scale(x[4],-600,600,0.6);
    return scaledx;
}

public double scaleReward(double x, double c, double r) {
    return x/c*r;
}

public int getMax(double[] x) {
    double max = -1000;
    int index = 0;
    for(int i = 0; i < x.length; i++) {
        if(x[i] > max) {
            max = x[i];
            index = i;
        }
    }
    return index;
}

public int getRandom() {
    return random.nextInt(5);
}
```

```
	/**
	 * onScannedRobot: width lock strategy
	 */
	public void onScannedRobot(ScannedRobotEvent e) {
		double angleToEnemy = getHeadingRadians() + e.getBearingRadians();
		double     radarTurn     =     Utils.normalRelativeAngle(angleToEnemy     -
getRadarHeadingRadians());
		double      extraTurn      =      Math.min(Math.atan(36.0/e.getDistance()),
Rules.RADAR_TURN_RATE_RADIANS);
		if (radarTurn < 0) {
			radarTurn -= extraTurn;
		}
		else {
			radarTurn += extraTurn;
		}
		setTurnRadarRightRadians(radarTurn);

		X = getX();
		Y = getY();
		distance = e.getDistance();
		velocity = e.getVelocity();
		enemyHeading = e.getHeadingRadians();

		double absBearingDeg = getHeading() + e.getBearing();
		if (absBearingDeg < 0) absBearingDeg += 360;
		deltaX = -distance*Math.sin(Math.toRadians(absBearingDeg));
		deltaY = -distance*Math.cos(Math.toRadians(absBearingDeg));
		deltaEnergy = getEnergy()-e.getEnergy();
	}
```

```
public void onBulletHit(BulletHitEvent event) {
	getReward = true;
	double power = event.getBullet().getPower();
	double bonus = power>1? (power-1)*2:0;
	reward = scaleReward(4*power+bonus,16,3);
	accumulatedPosReward += reward;
}

public void onBulletMissed(BulletMissedEvent event) {
	getReward = true;
	reward = -scaleReward(event.getBullet().getPower(),3,3);
	accumulatedNegReward += reward;
}

public void handleEnd(boolean wining) {
	round++;
	if(wining) {
		win++;
		reward = 6;
		accumulatedPosReward += reward;
	} else {
		reward = -6;
		accumulatedNegReward += reward;
	}

	double[] s = scale(previousState);
	double[] qt = neuralNet.outputsFor(s);
	double[] sPrime = {getX(), getY(), deltaEnergy, deltaX, deltaY};
	sPrime = scale(sPrime);
	double[] qtPlusOne = neuralNet.outputsFor(sPrime);
```

```
        int a = getMax(qtPlusOne);
        double delta = learningRate * (reward+gamma*qtPlusOne[a]-
qt[previousAction]);
        qt[previousAction] += delta;
        neuralNet.train(s, qt);

        File file = new File("weights.txt");
         if (!file.exists()) {
             try {
                 file.createNewFile();
             } catch (IOException e) {
                 e.printStackTrace();
             }
         }
         neuralNet.save(file);

         if(round > 99) {
             File result = new File("result.txt");
             if (!result.exists()) {
                 try {
                     result.createNewFile();
                 } catch (IOException e) {
                     e.printStackTrace();
                 }
             }
             try {
                     FileWriter writer = new FileWriter(result.getName(), true);
                     writer.write("Wining: "+win+" in "+round+" rounds. "+"Total pos
rewards: "+String.format("%.3f", accumulatedPosReward)
                     +" Total neg rewards: "+String.format("%.3f",
accumulatedNegReward)+"\n");
                        writer.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                win = 0;
                round = 0;
                accumulatedPosReward = 0;
                accumulatedNegReward = 0;
            }
        }

        public void onWin(WinEvent event) {
            handleEnd(true);
        }

        public void onDeath(DeathEvent event) {
            handleEnd(false);
        }

        public void onHitByBullet(HitByBulletEvent event) {
            getReward = true;
            double power = event.getPower();
            double bonus = power>1? (power-1)*2:0;
            reward = -scaleReward(4*power+bonus,16,3);
            accumulatedNegReward += reward;
        }

        public void onHitWall(HitWallEvent event) {
            getReward = true;
        reward = -0.5;
        accumulatedNegReward += reward;
    }

    public void onHitRobot(HitRobotEvent e) {
        if (e.isMyFault()) {
            getReward = true;
            reward = -0.5;
            accumulatedNegReward += reward;
        }
    }
}
```