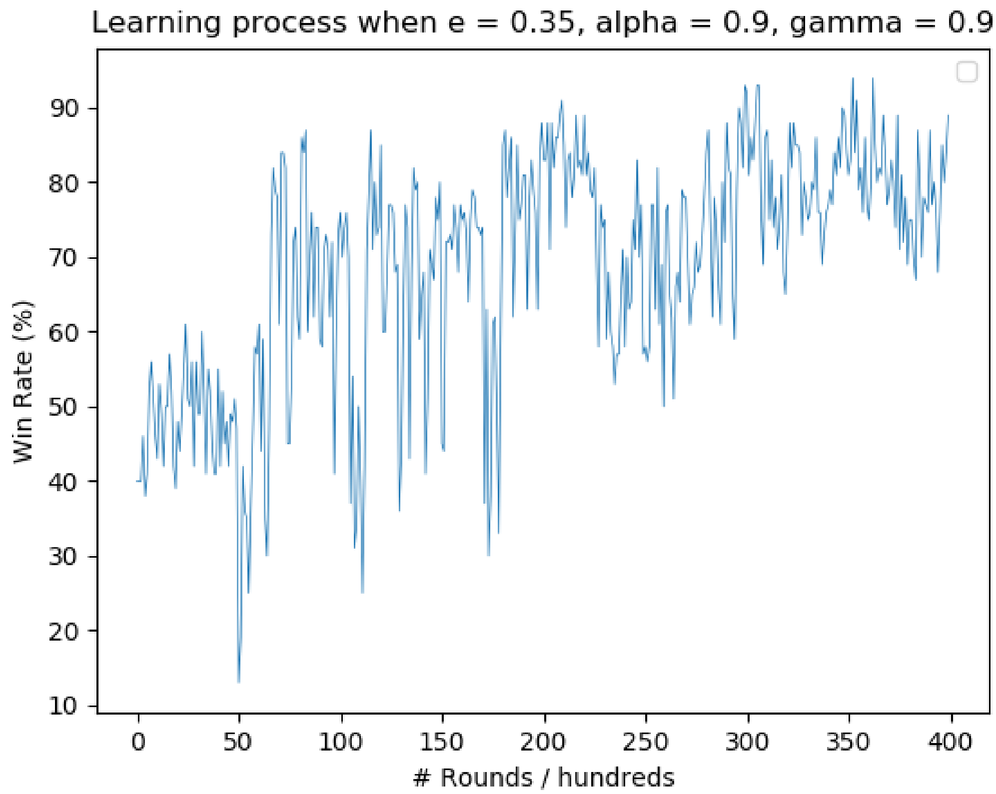
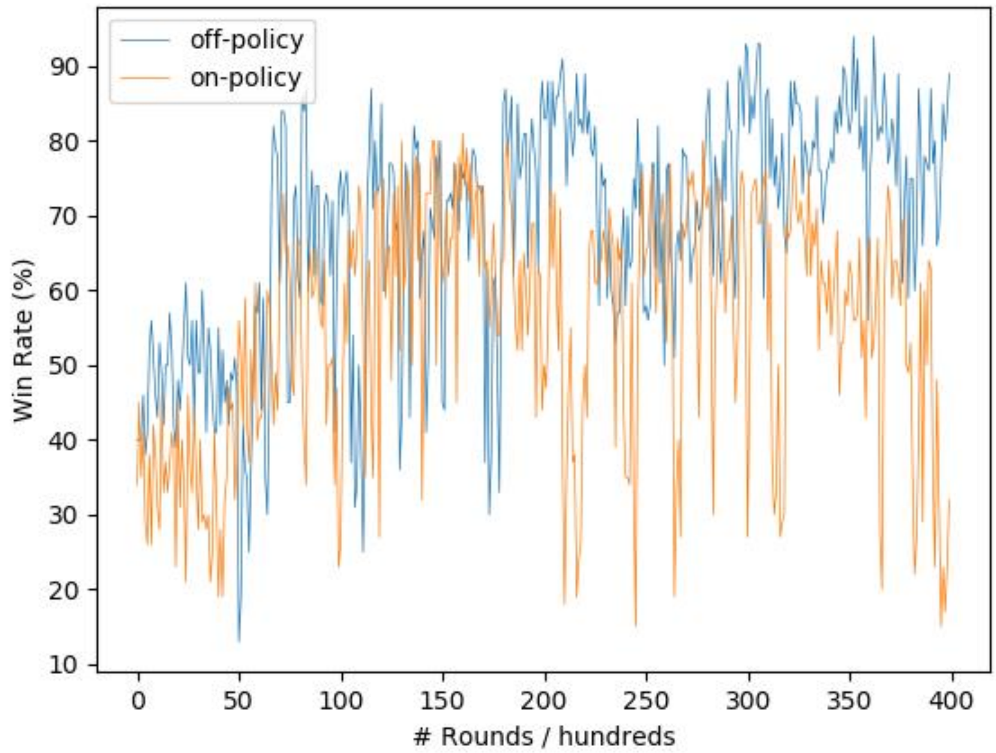
Assignment2 – Reinforcement learning (Look Up Table)

2-a.



This graph shows the learning process of my robot converging over time when e = 0.35, alpha =0.9, gamma =0.9. Initially the robot remains in the win rate of 50%, after 5000 rounds it starts to reach a win rate of 80% highest but can still goes down to 20% lowest. After 25000 rounds, the learning of the robot converges to around 80% and win rate fluctuation remains within 10%.

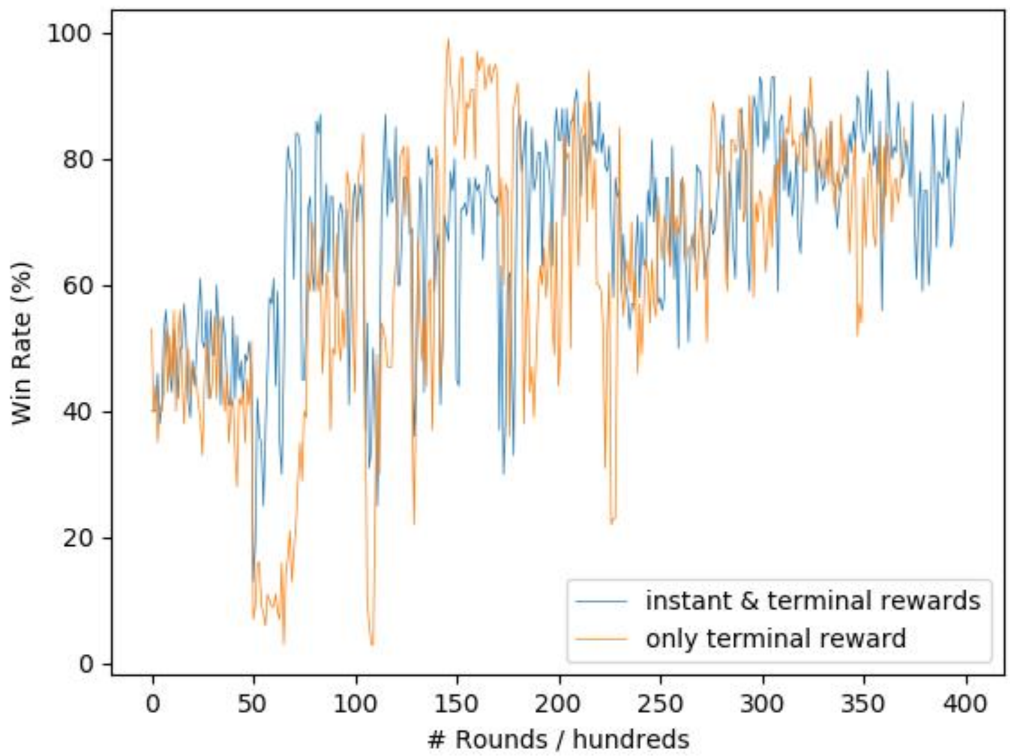
2-b.



This graph compares the learning efficiency of off-policy and on-policy, in which we can infer that on-policy learning has varying win rate and cannot converge even after 40000 rounds while off-policy learning coverges after 30000 rounds.

The reason is on-policy learning uses the same policy that is improved to select next actions, while off-policy learning uses an independent one. In this way, off-policy learning explores more than on-policy one so the learning outcome is more universal to more battle scenarios.

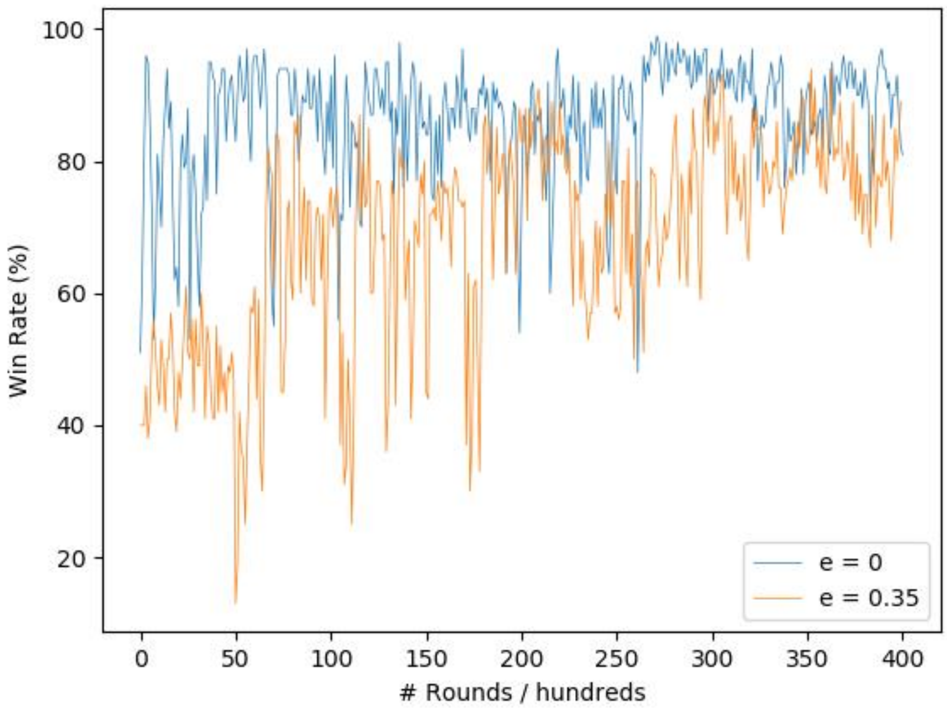
2-c.



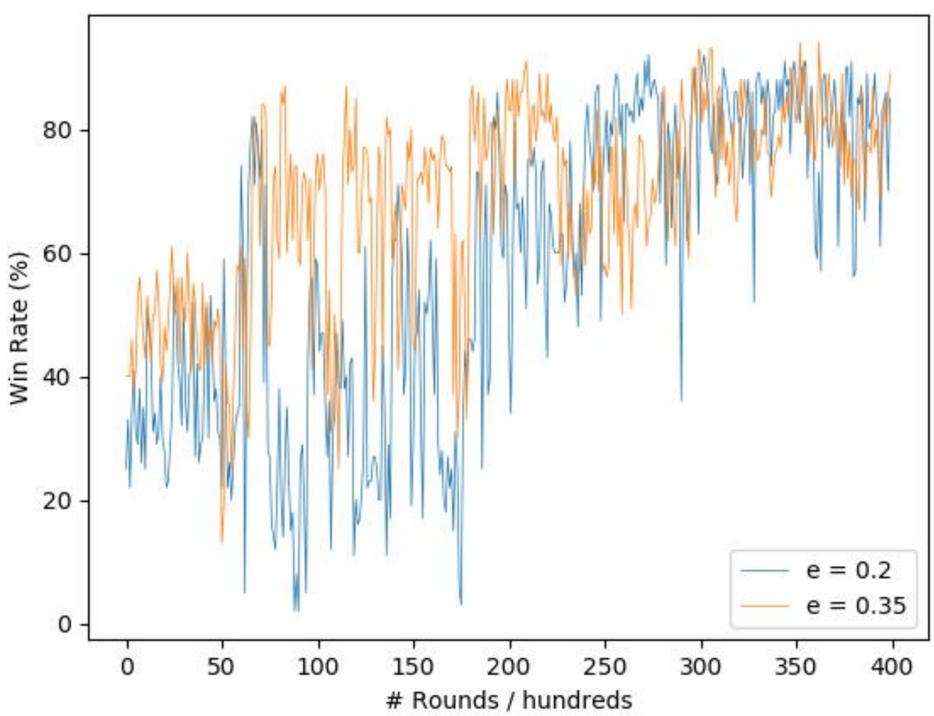
This graph shows the difference between two reward-giving strategies. The blue line indicates the scenario where intermediate rewards are given to the robot instantly if it hits or is hit by its opponent tank, while the yellow line indicates that the reward is given only when the robot wins or loses the battle.

From the graph we can see that when only terminal rewards are given, the win rate fluctuates more dramatically which means the performance of the robot is not stable. One of the possible reason of this behavior is that without intermediate rewards, the robot cannot learn more detailed information within the battle: winning(losing) a battle does not necessarily mean **every** action taken in this battle is correct(wrong), thus the convergence rate is slower than that of the learning method with intermediate rewards as the former takes more rounds to learn the same amount of knowledge than the latter.

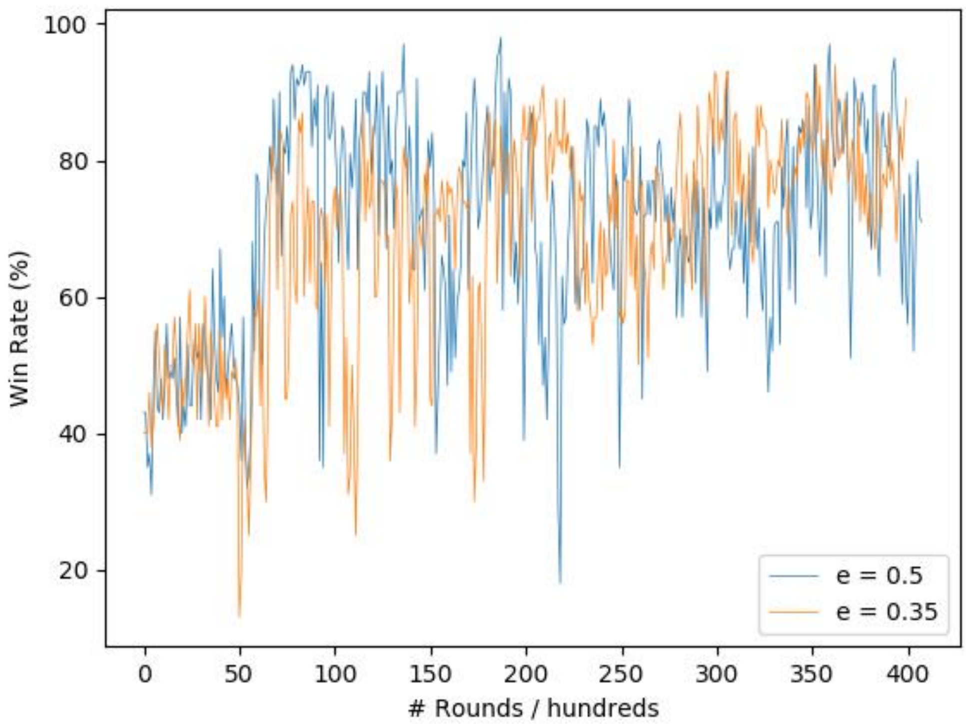
3-a.



Win Rate (%)
# Rounds / hundreds
e = 0
e = 0.35



Win Rate (%)
# Rounds / hundreds
e = 0.2
e = 0.35



The three graphs above shows the differences between probabilities e used to select random action.

When e = 0, there's no exploration at all, which means every action is greedy, this can be good for **this** battle but may not be useful to others if we change the opponent type and it's easy to come up with a local optimal solution.

When e = 0.2, the win rate remains at a lower level and still has a possibility of going down to 40% occasionally even after it converges; this happens when a certain but insufficient exploration rate is provided.

When e = 0.5, exploration rate goes higher than needed, which causes the learning process still cannot converge after 40000 rounds and the win rate varies dramatically over time.

Overall speaking, e = 0.35 is the rate that leads to the best learning performance.

Appendix

RLRobotLUT.java

```
import java.io.*;
import java.io.File;
import java.util.Arrays;
import java.util.Scanner;

public class RLRobotLUT implements LUTInterface {
    private int level1;
    private int level2;
    private int level3;
    private int level4;
    private int level5;
    private double[][][][][] lut;

    public RLRobotLUT(int level1, int level2, int level3, int level4,
int level5) {
        this.level1 = level1;
        this.level2 = level2;
        this.level3 = level3;
        this.level4 = level4;
        this.level5 = level5;

        lut = new double[level1][level2][level3][level4][level5];
    }

    @Override
    public void initialiseLUT() {
        for (int a = 0; a < level1; a++) {
            for (int b = 0; b < level2; b++) {
                for (int c = 0; c < level3; c++) {
                    for (int d = 0; d < level4; d++) {
                        Arrays.fill(lut[a][b][c][d], Math.random());
                    }
                }
            }
        }
    }

    @Override
    public int indexFor(double[] X) {
        return 0;
    }

    @Override
    public double outputFor(double[] x){
        return
lut[(int)x[0]][(int)x[1]][(int)x[2]][(int)x[3]][(int)x[4]];
    }

    @Override
    public double train(double[] x, double target){
        lut[(int)x[0]][(int)x[1]][(int)x[2]][(int)x[3]][(int)x[4]] =
target;
```

```
        return 0;
    }

    @Override
    public void save(File argFile) throws IOException {
        Gson gson = new Gson();
        String jsonModel = gson.toJson(this);
        if(!argFile.exists()){
            argFile.createNewFile();
        }
        FileWriter myWriter = new
FileWriter(argFile.getAbsolutePath());
        myWriter.write(jsonModel);
        myWriter.close();
        return;
    }

    @Override
    public void load(String argFileName) throws IOException {
        File modelFile = new File(argFileName);
        Scanner modelReader = new Scanner(modelFile);
        String model = modelReader.nextLine();
        modelReader.close();

        Gson gson = new Gson();
        RLRobotLUT loadedLUT = gson.fromJson(model, RLRobotLUT.class);

        this.lut = loadedLUT.getLUT();
        this.level1 = loadedLUT.getLevel1();
        this.level2 = loadedLUT.getLevel2();
        this.level3 = loadedLUT.getLevel3();
        this.level4 = loadedLUT.getLevel4();
        this.level5 = loadedLUT.getLevel5();
        return;
    }

    public double[][][][][] getLut() {
        return lut;
    }

    public int getLevel1() {
        return level1;
    }

    public int getLevel2() {
        return level2;
    }

    public int getLevel3() {
        return level3;
    }

    public int getLevel4() {
        return level4;
    }

    public int getLevel5() {
```

```
        return level5;
    }
}
```

RLRobot.java

```
import robocode.*;
import java.awt.*;
import java.util.Random;

import static robocode.util.Utils.normalRelativeAngleDegrees;

enum Energy {empty, almostEmpty, medium, half, high}
enum Action {retreat, advance, leaveCorner, fire, spin}
enum Distance {veryClose, close, middle, far, veryFar}

public class RLRobotCantWork extends AdvancedRobot {
    private final Energy energies[] = new Energy[]{
            Energy.empty,
            Energy.almostEmpty,
            Energy.medium,
            Energy.half,
            Energy.high
    };

    private final Distance distances[] = new Distance[]{
            Distance.veryClose,
            Distance.close,
            Distance.middle,
            Distance.far,
            Distance.veryFar
    };

    private final double hitReward = 1.0;
    private final double beHitReward = -0.25;
    private final double winReward = 2.0;
    private final double lossReward = -0.5;

    private final double gamma = 0.9;
    private final double alpha = 0.9;
    private final double epsilon = 0.35;

    private final boolean onPolicy = false;

    private RLRobotLUT RLLUT = new RLRobotLUT(
            Energy.values().length,
            Distance.values().length,
            Energy.values().length,
            Distance.values().length,
            Action.values().length
    );
    RLLUT.initialise();

    static int totalNumRounds = 0;
    static int numRoundsTo100 = 0;
```

```java
    static int numWins = 0;

    private Energy curMyEnergy = Energy.high;
    private Energy curEnemyEnergy = Energy.high;
    private Distance curDistanceToEnemy = Distance.middle;
    private Distance curDistanceleaveCorner = Distance.middle;
    private Action curAction = Action.spin;

    private Energy prevMyEnergy = Energy.high;
    private Energy prevEnemyEnergy = Energy.high;
    private Distance prevDistanceToEnemy = Distance.middle;
    private Distance prevDistanceleaveCorner = Distance.middle;
    private Action prevAction = Action.spin;

    // Initialize states
    double myX = 0.0;
    double myY = 0.0;
    double myEnergy = 0.0;
    double enemyBearing = 0.0;
    double enemyDistance = 0.0;
    double enemyEnergy = 0.0;

    double totalReward = 0.0;
    private double curReward = 0.0;
    int direction = 1;

    int xMid = 0;
    int yMid = 0;

    private void robotMovement() {
        if (Math.random() < epsilon)
            curAction = selectRandomAction();
        else
            curAction = selectBestAction(
                    myEnergy,
                    enemyDistance,
                    enemyEnergy,
                    distanceleaveCorner(myX, myY, xMid, yMid)
            );

        switch (curAction) {
            case spin: {
                setTurnRight(enemyBearing + 90);
                setAhead(50 * direction);
                break;
            }
            case fire: {
                turnGunRight(normalRelativeAngleDegrees(getHeading() -
getGunHeading() + enemyBearing));
                setFire(3);
                break;
            }
            case advance: {
                setTurnRight(enemyBearing);
                setAhead(100);
                break;
            }
```

```java
            case retreat: {
                setTurnRight(enemyBearing + 180);
                setAhead(100);
                break;
            }
            case leaveCorner: {
                double bearing = getBearingleaveCorner(getX(), getY(),
xMid, yMid, getHeadingRadians());
                setTurnRight(bearing);
                setAhead(100);
                break;
            }
        }
    }

    private void radarMovement() {
        setTurnRadarRightRadians(Double.POSITIVE_INFINITY);
    }

    private void trainLUT(){
        double[] index = new double[]{
                prevMyEnergy.ordinal(),
                prevDistanceToEnemy.ordinal(),
                prevEnemyEnergy.ordinal(),
                prevDistanceleaveCorner.ordinal(),
                prevAction.ordinal()};
        double Q = computeQ(curReward);

        RLLUT.train(index, Q);
    }

    public void run() {
        while (true) {
            robotMovement();
            radarMovement();

            if (getGunHeat() == 0)
                execute();

            trainLUT();
            execute();
        }
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent e) {
        myX = getX();
        myY = getY();
        enemyBearing = e.getBearing();
        enemyDistance = e.getDistance();
        enemyEnergy = e.getEnergy();
        myEnergy = getEnergy();

        prevMyEnergy = curMyEnergy;
        prevDistanceleaveCorner = curDistanceleaveCorner;
        prevDistanceToEnemy = curDistanceToEnemy;
        prevEnemyEnergy = curEnemyEnergy;
```

```
        prevAction = curAction;

        curMyEnergy = EnergyOf(getEnergy());
        curDistanceleaveCorner = DistanceOf(distanceleaveCorner(myX,
myY, xMid, yMid));
        curDistanceToEnemy = DistanceOf(e.getDistance());
        curEnemyEnergy = EnergyOf(e.getEnergy());
    }

    @Override
    public void onBulletHit(BulletHitEvent e) {
        totalReward += hitReward;
    }

    @Override
    public void onHitByBullet(HitByBulletEvent e) {
        totalReward += beHitReward;
    }

    private void updateBattleStats(){
        if (numRoundsTo100 < 100) {
            numRoundsTo100++;
            totalNumRounds++;
        } else {
            numRoundsTo100 = 0;
            numWins = 0;
        }
    }

    @Override
    public void onDeath(DeathEvent e) {
        curReward = lossReward;

        trainLUT();
        execute();

        updateBattleStats();
    }

    @Override
    public void onWin(WinEvent e) {
        curReward = winReward;

        trainLUT();
        execute();

        updateBattleStats();
    }

    @Override
    public void onHitWall(HitWallEvent e) {
        super.onHitWall(e);
        avoidWall();
    }

    @Override
    public void onHitRobot(HitRobotEvent e) {
```

```java
        super.onHitRobot(e);
        avoidWall();
    }

    public void avoidWall() {
        switch (curAction) {
            case spin: {
                direction = direction * -1;
                setAhead(50 * direction);
                break;
            }
            case advance: {
                setTurnRight(30);
                setBack(50);
                execute();
                break;
            }
            case retreat: {
                setTurnRight(30);
                setAhead(50);
                execute();
                break;
            }

        }
    }

    public double computeQ(double r) {
        Action candidateAction;
        if(onPolicy){
            if (Math.random() < epsilon)
                curAction = selectRandomAction();
            else {
                curAction = selectBestAction(
                        curMyEnergy.ordinal(),
                        curDistanceToEnemy.ordinal(),
                        curEnemyEnergy.ordinal(),
                        curDistanceleaveCorner.ordinal());
            }
            candidateAction = curAction;
        }
        else {
            candidateAction = selectBestAction(
                    curMyEnergy.ordinal(),
                    curDistanceToEnemy.ordinal(),
                    curEnemyEnergy.ordinal(),
                    curDistanceleaveCorner.ordinal());
        }

        double[] prevStateAction = new double[]{
                prevMyEnergy.ordinal(),
                prevDistanceToEnemy.ordinal(),
                prevEnemyEnergy.ordinal(),
                prevDistanceleaveCorner.ordinal(),
                prevAction.ordinal()};

        double[] curStateAction = new double[]{
```

```
                curMyEnergy.ordinal(),
                curDistanceToEnemy.ordinal(),
                curEnemyEnergy.ordinal(),
                curDistanceleaveCorner.ordinal(),
                candidateAction.ordinal()};

        double prevQ = RLLUT.outputFor(prevStateAction);
        double curQ = RLLUT.outputFor(curStateAction);

        double updatedQ = prevQ + alpha * (r + gamma * curQ - prevQ);
        return updatedQ;
    }

    public Action selectRandomAction() {
        Random rand = new Random();
        int r = rand.nextInt(Action.values().length);
        return Action.values()[r];
    }

    public Action selectBestAction(double e, double d, double e2,
double d2) {
        int energy = EnergyOf(e).ordinal();
        int distance = DistanceOf(d).ordinal();
        int enemyEnergy = EnergyOf(e2).ordinal();
        int distanceleaveCorner = DistanceOf(e2).ordinal();
        double bestQ = -Double.MAX_VALUE;
        Action bestAction = null;

        for (int a = Action.spin.ordinal(); a < Action.values().length;
a++) {
            double[] x = new double[]{energy, distance, enemyEnergy,
distanceleaveCorner, a};
            if (RLLUT.outputFor(x) > bestQ) {
                bestQ = RLLUT.outputFor(x);
                bestAction = Action.values()[a];
            }
        }
        return bestAction;
    }

    public Distance DistanceOf(double distance) {
        if (distance < 50) return Distance.veryClose;
        else return distances[(int) ((distance-50) / 200 + 1)];
    }

    public Energy EnergyOf(double energy) {
        if(energy == 0) return Energy.empty;
        return energies[(int) (energy / 20 + 1)];
    }

    public double distanceleaveCorner(double fromX, double fromY,
double toX, double toY) {
        double distance = Math.sqrt(Math.pow((fromX - toX), 2) +
Math.pow((fromY - toY), 2));
        return distance;
    }
}
```