CPEN 502 Part3 Report

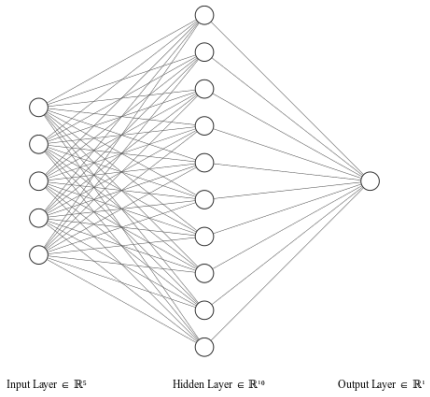
Shanny Lu - 57267783

In this file, I use green to highlight my answer to each question.

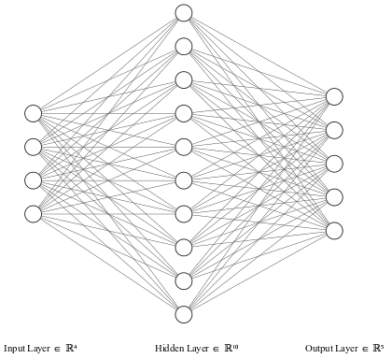
Part(4)

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

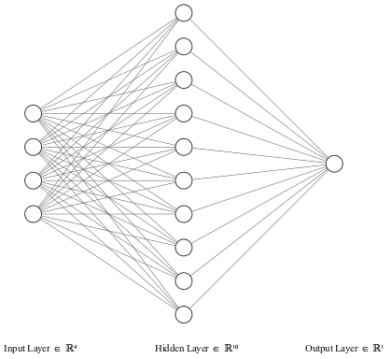
a) There are 3 options for the architecture of your neural network. Describe and draw all three options and state which you selected and why. (3pts)



Answer: In the first option, we use state-action pair as inputs, and then output a single Q value.



In the Second option, we only use the state as inputs, and then use a neural network to compute the Q value of all the possible actions, which is five in my case.



In the third option, we only use the state as input and then use a neural network to compute the single Q value for one action. And we can repeatedly five times for five actions Q value.

I decided to use option 1 in this project because it is relatively simpler compared to the other two. In the 2nd option, different actions might converge differently, which leads to unexpected behaviours. In the 3rd option, we need to repeat multiple times for each action, this is too time-consuming, even though this

option sounds accurate. However, there are also some drawbacks to using option 1. For example, considering both state and action together would lead to complications that diminish the model's precision.

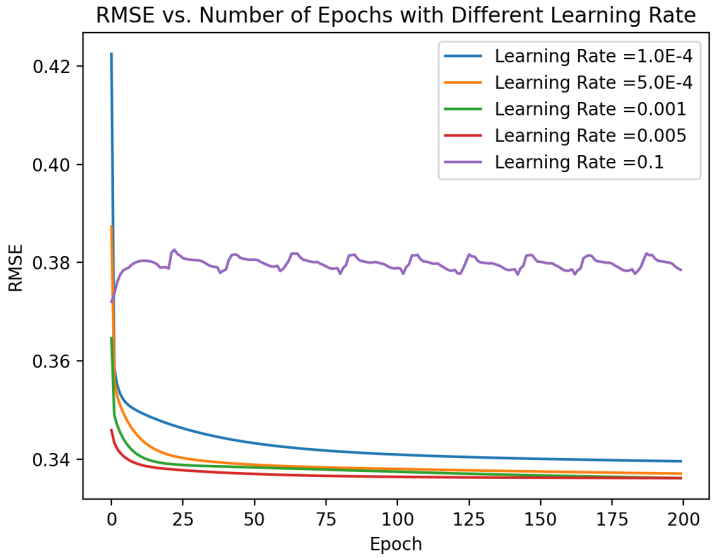
b) Show (as a graph) the results of training your neural network using the contents of the LUT from Part 2. Your answer should describe how you found the hyper-parameters which worked best for you (i.e. momentum, learning rate, number of hidden neurons). Provide graphs to backup your selection process. Compute the RMS error for your best results. (5 pts)

Answer:

Considering **learning rate** as variable, with momentum = 0.9, and Number of Hidden neuron = 20. We found when learning rate at 0.001 we get the lowest RMS error.

Learning rates	0.0001	0.0005	**0.001**	0.005	0.1
RMS Error	0.3395	0.3370	**0.336130**	0.336135	0.3785

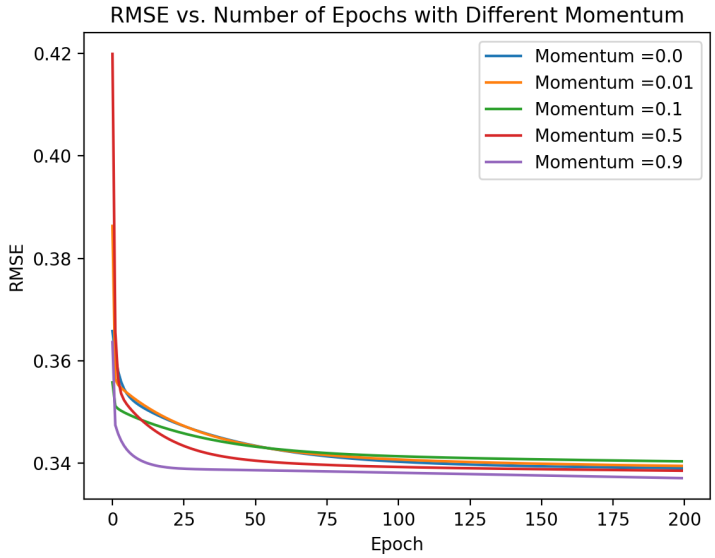
Table 1: Relationship between learning rates and RMS error.



Considering **momentum** as variable, with learning rate = 0.001, and Number of Hidden neuron = 20. We found that we get lowest RMS error when momentum at 0.9.

Momentum	0.0	0.01	0.1	0.5	**0.9**
RMS Error	0.3389	0.3394	0.3403	0.3384	**0.3370**

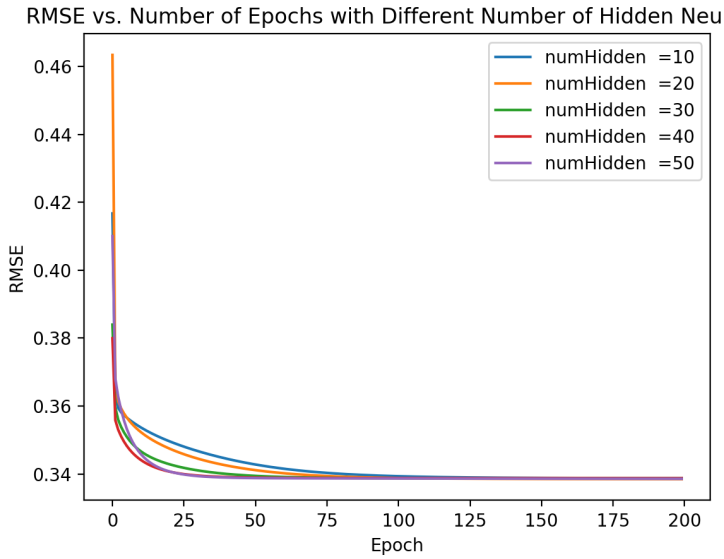
Table 2: Relationship between momentum and RMS error.



Considering **Number of Hidden Neurons** as variable, with learning rate = 0.001, and Momentum = 0.9. We get lowest RMS error when number of hidden neurons is 20.

Num of Hidden neurons	10	**20**	30	40	50
RMS Error	0.33873	**0.33847**	0.33867	0.33873	0.33856

Table 3: Relationship between NumOfHiddenNeurons and RMS error.



Thus, based on all comparisons above, I choose using learning rate = 0.001, momentum = 0.9 and number of hidden neurons = 20 as my hyper-parameters in this project.

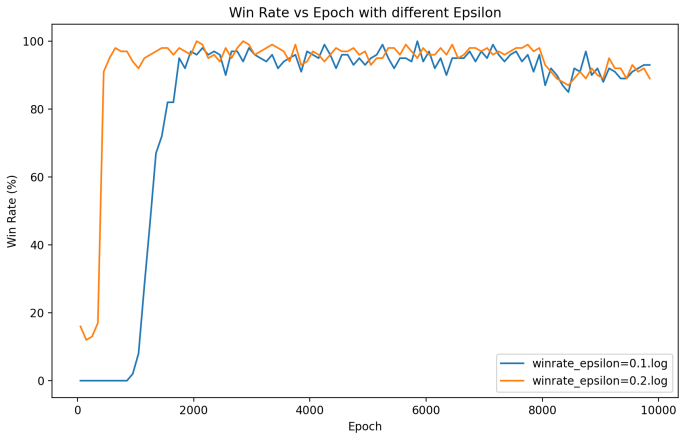
c) Comment on why theoretically a neural network (or any other approach to Q-function approximation) would not necessarily need the same level of state space reduction as a look up table. (2 pts)
Answer:

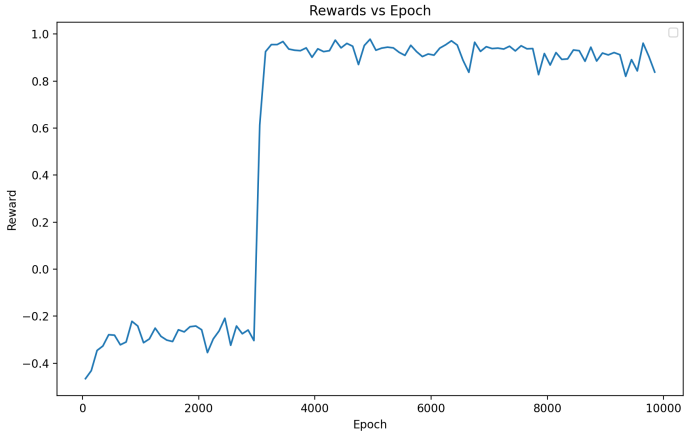
In cases when we have large state spaces, LUT becomes impractical since they need an individual entry for every state-action pair, which leads to overwhelming memory usage. Conversely, neural networks are highly effective in these high-dimensional state spaces. They can generalize from the states they have been trained on to new states, a feature particularly valuable in situations where encountering every possible state is unfeasible. Moreover, neural networks are more efficient in representing the Q-function, managing vast state spaces using fewer parameters than a lookup table would need for a complex state-action mapping. This efficiency significantly reduces the need for the level of state space compression required by lookup tables.

Part(5)

(5) Hopefully you were able to train your robot to find at least one movement pattern that results in defeat of your chosen enemy tank, most of the time.

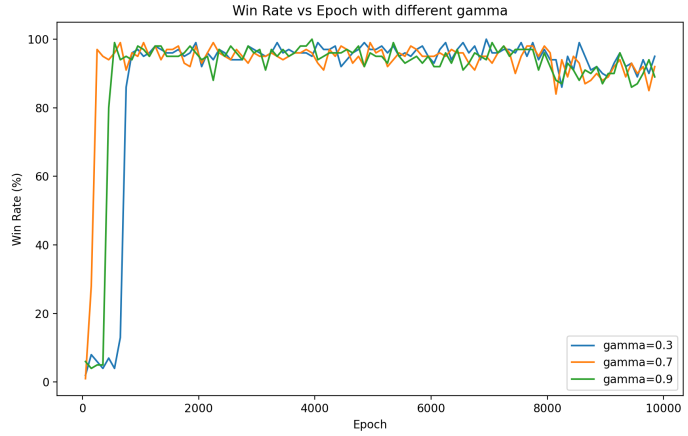
a) Identify two metrics and use them to measure the performance of your robot with online training. I.e. during battle. Describe how the results were obtained, particularly with regard to exploration? Your answer should provide graphs to support your results. (5 pts)
Answer:
We can use **win rate** and **total rewards** as two metrics to measure the performance of the robot with online learning.
Winning rates measure the percentage of wins by my implemented robot, while rewards measure the total rewards my robot received in each round battle. The robot will participate in 10000 rounds of battle with the selected robot (I choose fire robot in this project). We could obtain the win rate by dividing the total number of wins by the total number of battles, the robot could then determine its winning percentages. And we could obtain the value of the rewards by keeping track of how the value of the rewards ended up being at each round, either onWin or onDeath. I have visualized how the win rate or rewards varied vs the total epochs below.





In reinforcement learning, it is important to find a balance between exploration and exploitation. Initially, a robot will explore various actions, resulting in fluctuating winning rates and rewards. As it gradually shifts to exploiting successful strategies, we expect a more stable and rising winning rate, along with steadily increasing rewards, signaling improved performance.

b) The discount factor γ can be used to modify influence of future reward. Measure the performance of your robot for different values of γ and plot your results. Would you expect higher or lower values to be better and why? (3 pts)



Answer:
I visualized the win rate over epoch with different discount factors with 0.3, 0.7, and 0.9 as shown above. The discount factor indicates how much future rewards will be considered. A lower gamma value means the robot prioritizes immediate rewards, while a higher value leads the robot to consider future rewards more significantly.
In this robot battle cases, I expect that a higher discount factor would lead to better performance since it will consider more future, which could allow robots to consider the long-term consequences of their actions. However, too high γ value might also lead to overestimating the importance of distant future rewards, and too low γ value would focus too much on immediate rewards and miss the potential long-term outcome.
From the graph, we can tell that win rates for higher gamma values (when $\gamma = 0.7, 0.9$) increase and converge faster than lower values (when $\gamma = 0.3$), which matches with what I expected.

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. (3 pts)

Answer:

The key differences in convergence between TD learning with a lookup table(LUT) and using function approximation for the Q-function are based on their update mechanisms and the Bellman equation:

In TD learning with LUT, each state-action pair's Q-value is updated following the Bellman equation. With proper time and under certain conditions, such as sufficient exploration of all states and a decreasing learning rate, these Q-values will converge to their true values.

However, when using function approximation methods, like neural networks, the approach is to generalize across the entire state-action space, instead of updating discrete values. Here, the Bellman equation modifies the model's parameters, impacting numerous state-action pairs at once. This widespread effect of updates can cause instability or even divergence, because an adjustment made for one state-action pair can inadvertently affect the estimates for others, leading to a cycle of continuous, non-converging modifications.

To sum up, while a lookup table approach with direct updates can assure convergence under specific conditions, function approximation adds a layer of complexity and interdependence to these updates, resulting in a lack of guaranteed convergence.

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. (3 pts)

Answer:

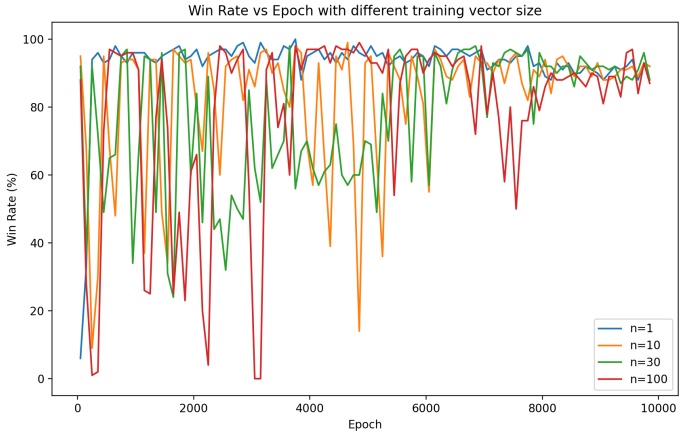
To assess a neural network's learning effectiveness, key metrics like the win rate, bullet hit accuracy, and longitudinal progress tracking can be utilized.

Firstly, the win rate serves as a vital measure. A consistent increase in the win rate over successive rounds indicates an enhancement in the model's performance.

Secondly, the frequency of successful bullet hits can reflect learning efficiency. Given that hitting more targets often correlates with winning, a higher rate of bullet hits suggests improved performance.

Finally, monitoring the neural network's development over time is an insightful approach. This involves real-time operation of the network while periodically documenting its performance on a series of test episodes. Charting these results over time provides a visual illustration of the network's gradual skill acquisition and progress.

e) At each time step, the neural net in your robot performs a back propagation using a single training vector provided by the RL agent. Modify your code so that it keeps an array of the last say n training vectors and at each time step performs n back propagations. Using graphs compare the performance of your robot for different values of n. (4 pts)



Answer:
In the graph provided, I explored the effect of varying the number of training vectors 'n', experimenting with values such as 1, 10, 20, and 100. The visual data suggests that employing a moderate to large batch of historical training vectors, particularly around 'n=30' or 'n=100', appears to enhance both the stability and the overall efficacy of the robot's performance in this case.

Part(6)

(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? (4 pts)
Answer:
In this project, I've deepened my understanding of using reinforcement learning (RL) and neural networks (NN), particularly focusing on backpropagation, for real-world applications. A crucial aspect of my experience was delving into the essential principles of both RL and NN while developing a robot leveraging these techniques.
For improving the robot's performance, exploring advanced RL strategies like deep Q-learning is advisable. These methods are better suited for handling extensive and complex state spaces. Moreover, integrating advanced function approximators, such as deep neural networks, could enhance the versatility and effectiveness of the Q-function representations.
To tackle convergence issues, employing regularization methods, including weight decay and dropout, is beneficial. These can help in preventing the neural network from overfitting to the training data.
When it comes to applying RL with neural network-based function approximation in other scenarios, experimenting with various approximators, like simpler neural networks and linear models, is valuable. This exploration aids in understanding their pros and cons. To further improve the policy's effectiveness and stability, it's useful to combine exploration techniques with regularization and adaptively tune learning rates and momentum.

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. (3 pts)

Answer:

A key concern with applying reinforcement learning in sensitive applications, like autonomous anesthesia administration in surgeries, is the ethical implication. The issue arises because reinforcement learning algorithms aim to optimize a given reward function, which may not fully align with the intricate goals of an anesthesia system. For instance, the algorithm might focus more on patient comfort, potentially overlooking other vital aspects like the patient's vital signs or the precise dosage of anesthesia.

To address these issues, a viable solution could be the adoption of a hybrid approach that merges reinforcement learning with other techniques like model-based control. Another option is to use these models as supplementary tools that provide additional insights to doctors, rather than making autonomous decisions. There are also many other concerns regarding this application, for example, we need high-quality, comprehensive data to train such a model. In medical scenarios, particularly with anesthesia, acquiring such data is difficult due to the variability between surgeries and patients, etc.

Appendix: Source Code Implemented

A State.java

```
    package main.java;

public class State {
    public double myEnergy;
    public double enemyEnergy;
    public double distToEnemy;
    public double distToCenter;

    // Constructor
    public State (double myEnergy, double enemyEnergy, double distToEnemy, double distToCenter)
        {
        this.myEnergy = myEnergy;
        this.enemyEnergy = enemyEnergy;
        this.distToEnemy = distToEnemy;
        this.distToCenter = distToCenter;
    }
}
```

B Experience.java

```
    package main.java;


public class Experience {
        public State prevState;
        public State currentState;
        public replayNNRobot.enumAction prevAction;
        public double currentReward;


        // Constructor
        public Experience(State prevState, replayNNRobot.enumAction prevAction, double
            currentReward, State currentState) {
            this.prevState = prevState;
            this.prevAction = prevAction;
            this.currentReward = currentReward;
            this.currentState = currentState;
        }
}
```

C main.java

```
package main.java;

import com.github.sh0nk.matplotlib4j.Plot;
import com.github.sh0nk.matplotlib4j.PythonExecutionException;
```

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import static main.java.Utils.IntToBiArray;

public class main {


    static ArrayList<Double> errorList = new ArrayList<Double>();
    static ArrayList<Integer> epochList = new ArrayList<Integer>();
    static LookUpTable lut= new LookUpTable(5, 5, 5, 5, 5);
    static Plot plt2 = Plot.create();

    public static void main(String[] args) throws PythonExecutionException, IOException {

        lut.load("out/production/part3/main/java/MyOwnRobot.data/LUT-fire.txt");

        // learning rate 0.001
        hiddenNeuronPlot(20,0.0001,0.9);
        hiddenNeuronPlot(20,0.0005,0.9);
        hiddenNeuronPlot(20,0.001,0.9);
        hiddenNeuronPlot(20,0.005,0.9);
        hiddenNeuronPlot(20,0.1,0.9);

//        // number of neuron 20
//        hiddenNeuronPlot(10,0.001,0.9);
//        hiddenNeuronPlot(20,0.001,0.9);
//        hiddenNeuronPlot(30,0.001,0.9);
//        hiddenNeuronPlot(50,0.001,0.9);

        // momentum
//        hiddenNeuronPlot(10,0.001,0.0);
//        hiddenNeuronPlot(20,0.001,0.01);
//        hiddenNeuronPlot(30,0.001,0.1);
//        hiddenNeuronPlot(30,0.001,0.5);
//        hiddenNeuronPlot(50,0.001,0.9);


        plt2.xlabel("Number of epochs");
        plt2.ylabel("RMS error");
        plt2.title("RMS error vs. number of epochs");
        plt2.legend();
        plt2.savefig("./figures/RMSE-neuron.png");
        plt2.show();

    }

    private static void hiddenNeuronPlot(int numOfNeuron, double lr, double momentum){
        // bipolar representation
        errorList.clear();
        epochList.clear();

        double errorSum;
        double avgErrorSum;
```

```
        int epoch;

        int EnergyLength = 5;
        int DistantLength = 5;
        int ActionSize = 5;


        NeuralNet nn = new NeuralNet(9, numOfNeuron, 1, lr, momentum, false);
        nn.initializeWeights();
        epoch = 0;
        errorList.clear();
        epochList.clear();

        do {
            int iteration = 0;
            errorSum = 0;
            for (int a = 0; a < EnergyLength; a++) {
                for (int b = 0; b < EnergyLength; b++) {
                    for (int c = 0; c < DistantLength; c++) {
                        for (int d = 0; d < DistantLength; d++) {
                            for (int e = 0; e < ActionSize; e++) {
                                double[] ExpandedX = IntToBiArray(e);
                                double[] input = new double[]{
                                        normalizeX.get(a),
                                        normalizeX.get(b),
                                        normalizeX.get(c),
                                        normalizeX.get(d),
                                        ExpandedX[0],
                                        ExpandedX[1],
                                        ExpandedX[2],
                                        ExpandedX[3],
                                        ExpandedX[4]
                                };

                                double qValue = lut.outputFor(new double[]{a,b,c,d,e});
//                                  double normalizedValue = 2 * (qValue - minMaxQ[0]) / (minMaxQ[1]
    - minMaxQ[0]) - 1;
                                double error = nn.train(input, qValue);
                                errorSum += Math.pow(error, 2); // E = SUM(C-y)^2
                                iteration++;
                            }
                        }
                    }
                }
            }
            epoch++;
            avgErrorSum = Math.sqrt(errorSum/iteration);
            errorList.add(avgErrorSum);
            epochList.add(epoch);
        } while (epoch < 1000);
        plt2.plot().add(epochList, errorList).label("NumOfNeurons = " + numOfNeuron);
    }


    static Map<Integer, Double> normalizeX = new HashMap<>(){{
        put(0, 0.2);
        put(1, 0.4);
```

```
        put(2, 0.6);
        put(3, 0.8);
        put(4, 1.0);
    }};
}
```

D replayNNRobot.java

```
    package main.java;

import robocode.*;

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

import static main.java.Utils.oneHotVectorFor;

public class replayNNRobot extends AdvancedRobot {

    //    public enum enumEnergy {zero, low, average, high, highest} // for myEnergy and
        enemyEnergy
//    public enum enumDistance {closest, close, medium, far, farthest} // for DistanceToEnemy
    and DistanceToCenter
    public enum enumAction {attack, forward, backward, left, right}
    public enum enumOperationMode {performScan, performAction}

    static int numInput = 9;
    static int numHidden = 20;
    static double lr = 0.001;
    static double momentum =0.9; //0.0
    static NeuralNet nn = new NeuralNet(numInput, numHidden,1, lr, momentum, false);

    // my state
    public double myX = 0.0;
    public double myY = 0.0;
    public double myEnergy = 0.0;

    // Enemy state
    public double enemyBearing = 0.0;
    public double enemyEnergy = 0.0;
    public double DistanceToEnemy = 0.0; //enemyDistance


    public double centerX = 0.0;
    public double centerY = 0.0;



    // RL learning parameters
    private double gamma = 0.9; // 0.2, 0.5, 0.7, 0.9
    private double alpha = 0.1; // learning rate: 0.1
    private double epsilon = 0.1; // exploration rate: 0.0, 0.1, 0.2, 0.5, 0.8
    private boolean offPolicy = true; // true for Q-leaning, false for Sarsa
```

```
// reward
private double currentReward = 0.0;
private double negativeReward = -0.1; // set to 0 when only consider terminal, -0.1
private double positiveReward = 0.5; // set to 0 when only consider terminal, 0.5
private double negativeTerminalRewards = -0.2; // -0.2
private double positiveTerminalRewards = 1; //1
static boolean NNinitialized = false;

// number of round
static int TotalRound = 0;
static int WinsPer100 = 0;
static int RoundsPer100 =0;
static double RewardsPer100 = 0.0;
static int[] winRate = new int[10000];

/**
 * Create replay memory to train more than 1 sample at a time step
 */
static int memSize = 10;
static ReplayMemory<Experience> rm = new ReplayMemory<>(memSize);

// CurrentState Initialization
public State currentState = new State(100.0, 100.0, 500.0, 500.0);
public enumAction currentAction = enumAction.forward;
public enumOperationMode operationMode = enumOperationMode.performScan;

// PreviousState Initialization
public State prevState = new State(100.0, 100.0, 500.0, 500.0);
public enumAction prevAction = enumAction.forward;


// Logging
static String logFilename = "replay_memSize=10.log";
static LogFile log = null;


@Override
public void run() {
    super.run();

    setGunColor(Color.blue);
    setBodyColor(Color.cyan);
    setBulletColor(Color.black);
    setRadarColor(Color.gray);
    setScanColor(Color.green);

    centerX = getBattleFieldWidth()/2;
    centerY = getBattleFieldHeight()/2;

    // Create log file
    if (log == null) {
        log = new LogFile(getDataFile(logFilename));
    }

    if(!NNinitialized){
        NNinitialized = true;
        nn.initializeWeights();
```

```
}

while(true){
    if(TotalRound > 8000) epsilon=0;

    switch (operationMode){
        case performScan:{
            currentReward = 0.0;
            turnRadarLeft(90);
            break;
        }
        case performAction:{
            if(Math.random() <= epsilon){
                currentAction = enumAction.values()[selectRandomAction()];
            } else {
                double DistanceToCenter = getDistFromCenter(myX,myY,centerX,centerY);

                currentAction = enumAction.values()[bestAction(
                        myEnergy,
                        enemyEnergy,
                        DistanceToEnemy,
                        DistanceToCenter
                )];
            }
            switch (currentAction){
                case attack:
                    setRadarColor(Color.red);
                    double amountToTurn = getHeading() - getGunHeading() + enemyBearing;
                    if(amountToTurn == 360.0 || amountToTurn == -360.0){
                        amountToTurn = 0.0;
                    }
                    turnGunRight(amountToTurn);
                    fire(5);
                    break;

                case forward:
                    setAhead(100);
                    execute();
                    break;

                case backward:
                    setBack(100);
                    execute();
                    break;

                case left:
                    setTurnLeft(30);
                    setAhead(100);
                    execute();
                    break;

                case right:
                    setTurnRight(30);
                    setAhead(100);
                    execute();
                    break;
            }
```

```
            }
            // Update previous Q
            double[] X = new double[]{
                    prevState.myEnergy,
                    prevState.enemyEnergy,
                    prevState.distToEnemy,
                    prevState.distToCenter,
                    prevAction.ordinal()
            };

            rm.add((new Experience(prevState,prevAction,currentReward,currentState)));
            replayExperience(rm);
            operationMode = enumOperationMode.performScan;
            execute();
        }
    }
}

public double getQValue(double currentReward, boolean offPolicy){

    // for sarsa on policy
    double currentQValue = nn.outputFor(oneHotVectorFor(new double[]{
            currentState.myEnergy,
            currentState.enemyEnergy,
            currentState.distToEnemy,
            currentState.distToCenter,
            currentAction.ordinal()})
    );

    int GreedyMove = bestAction(
            currentState.myEnergy,
            currentState.enemyEnergy,
            currentState.distToEnemy,
            currentState.distToCenter
    );

    // for q-learning off policy
    double maxQValue = nn.outputFor(oneHotVectorFor(new double[]{
            currentState.myEnergy,
            currentState.enemyEnergy,
            currentState.distToEnemy,
            currentState.distToCenter,
            GreedyMove})
    );

    double prevQValue = nn.outputFor(oneHotVectorFor(new double[]{
            prevState.myEnergy,
            prevState.enemyEnergy,
            prevState.distToEnemy,
            prevState.distToCenter,
            prevAction.ordinal()})
    );

    double newQValue;
    // Q-learning (off-policy)
    if(offPolicy){
        newQValue = prevQValue + alpha * (currentReward + gamma * maxQValue - prevQValue);
```

```
    }else {
        // Sarsa (on-policy)
        newQValue = prevQValue + alpha * (currentReward + gamma * currentQValue -
            prevQValue);
    }

    return newQValue;
}


public void replayExperience(ReplayMemory rm){
    int memorySize = rm.sizeOf();
    int requestedSampleSize = Math.min(memorySize, memSize);

    Object[] sample = rm.sample(requestedSampleSize);
    for(Object item:sample){
        Experience exp = (Experience) item;

        double[] x = new double[]{
                exp.prevState.myEnergy,
                exp.prevState.enemyEnergy,
                exp.prevState.distToEnemy,
                exp.prevState.distToCenter,
                exp.prevAction.ordinal()};

        double[] xScaledOneHotEncoded = oneHotVectorFor(x);
        nn.train(xScaledOneHotEncoded, getQValue(exp.currentReward, offPolicy));
    }
}

public double getDistFromCenter(double myX, double myY, double centerX, double centerY){
    double dist = Math.sqrt(Math.pow(myX - centerX, 2) + Math.pow(myY - centerY, 2));
    return dist;
}



@Override
public void onScannedRobot(ScannedRobotEvent e){
    super.onScannedRobot(e);

    myX = getX();
    myY = getY();
    enemyBearing = e.getBearing();
    DistanceToEnemy = e.getDistance();
    enemyEnergy = e.getEnergy();
    myEnergy = getEnergy();

    // Update previous state
    prevState.myEnergy = currentState.myEnergy;
    prevState.enemyEnergy = currentState.enemyEnergy;
    prevState.distToEnemy = currentState.distToEnemy;
    prevState.distToCenter = currentState.distToCenter;
    prevAction = currentAction;
    operationMode = enumOperationMode.performAction;

    //Update current state
```

```
    currentState.myEnergy = getEnergy();
    currentState.enemyEnergy = e.getEnergy();
    currentState.distToEnemy = e.getDistance();
    currentState.distToCenter = getDistFromCenter(myX,myY,centerX,centerY);
    operationMode = enumOperationMode.performAction;
}



@Override
public void onHitByBullet(HitByBulletEvent e) {
    currentReward += negativeReward;
}

@Override
public void onBulletHit(BulletHitEvent event) {
    currentReward += positiveReward;
}

@Override
public void onBulletMissed(BulletMissedEvent event) {
    currentReward += negativeReward;
}

@Override
public void onHitRobot(HitRobotEvent event) {
    currentReward += negativeReward;
    setBack(200);
    fire(3);
    setTurnRight(60);
    execute();
}

@Override
public void onHitWall(HitWallEvent event) {
    currentReward += negativeReward;
    setBack(200);
    setTurnRight(60);
    execute();
}


@Override
public void onWin(WinEvent event) {
    currentReward += positiveTerminalRewards;
    RewardsPer100 += currentReward;

    rm.add(new Experience(prevState, prevAction, currentReward, currentState));
    replayExperience(rm);


    if (RoundsPer100 < 100) {
        RoundsPer100++;
        TotalRound++;
        WinsPer100++;
    } else {
        // win rate
```

```
            log.stream.printf("%d - %d, %d\n", TotalRound - 100, TotalRound, 100*WinsPer100 /
                RoundsPer100);
            // rewards
//            log.stream.printf("%d - %d, %f\n", TotalRound - 100, TotalRound, RewardsPer100 /
    RoundsPer100);
            log.stream.flush();
            RoundsPer100 = 0;
            WinsPer100 = 0;
            RewardsPer100 = 0;
//            TotalRound++;
        }
    }

    @Override
    public void onDeath(DeathEvent event) {
        currentReward += negativeTerminalRewards;
        RewardsPer100 += currentReward;


        rm.add(new Experience(prevState, prevAction, currentReward, currentState));
        replayExperience(rm);

        if (RoundsPer100 < 100) {
            RoundsPer100++;
            TotalRound++;
        } else {
            // win rate
            log.stream.printf("%d - %d, %d\n", TotalRound - 100, TotalRound, 100*WinsPer100 /
                RoundsPer100);
            // rewards
//            log.stream.printf("%d - %d, %f\n", TotalRound - 100, TotalRound, RewardsPer100 /
    RoundsPer100);
            log.stream.flush();
            RoundsPer100 = 0;
            WinsPer100 = 0;
            RewardsPer100 = 0;
//            TotalRound++;
        }

    }

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

    public int bestAction(double myEnergy, double enemyEnergy, double distanceToEnemy, double
        distanceToCenter){
        double bestQ = -Double.MAX_VALUE;
        int bestAction = -Integer.MAX_VALUE;

        for(int i=0; i<enumAction.values().length; i++){
            double[] x = new double[]{myEnergy, enemyEnergy, distanceToEnemy, distanceToCenter,
                i};
            double[] xScaledOneHotEncoded = oneHotVectorFor(x);
            double predictedQ = nn.outputFor(xScaledOneHotEncoded);
```

```
            if(predictedQ > bestQ){
                bestQ = predictedQ;
                bestAction = i;
            }
        }
        return bestAction;
    }

}
```