CPEN 502 Assignment 3 Report

Jiaqi Zhang (63174551)

Questions

(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)

1) The first model is Fig.1. shown below. Inputs consists of a state-action pair, where the state is represented by several attribute values. The output is just one single Q value associated with that input action.

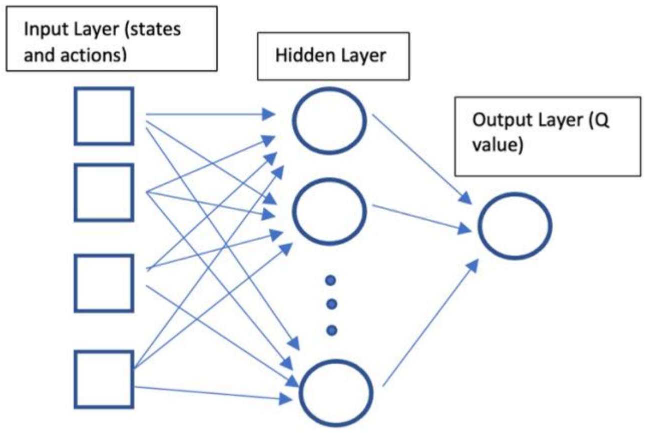


Fig.1. Neural Net for Single Q Value Outputs

2) The second model is modified from the first model where Q values are calculated for each action. The inputs of this model are only states, and the outputs are Q values per action.

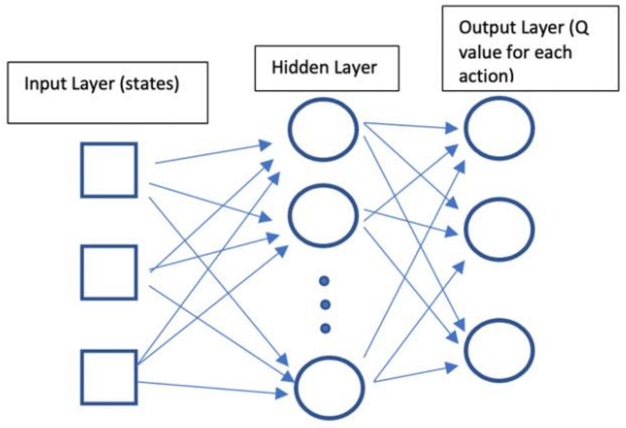


Fig.2. Neural Net for Q Values for each action

3) Different from the first two models, our third model constructs a list of Neural Nets instead of one single Neural Net. Each Neural Network corresponds to a different action. The input of each single Neural Net is the states, and the output is Q value.

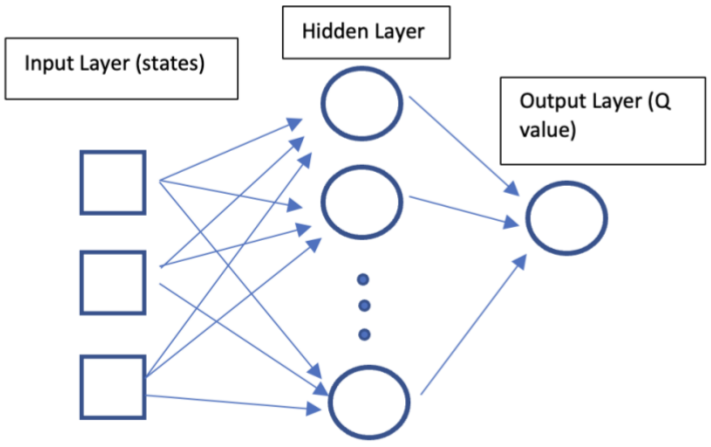


Fig.3. List of Neural Nets with Q value for certain action

Out of these 3 models, we selected the third one to implement. It has several advantages over the other two. The first model mixes states and weights in inputs, which could potentially reduce model accuracy. The second model, on the other hand, shares the same neural net among all actions. This adds a lot of correlations to actions. The third model is the most accurate because we train a separate neural net for each action, and actions will not share the weights and affect each other. This could help us train a better model.

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)

We implemented our neural network by using the contents of the LUT from Part 2 as inputs. We also calculated the RMS errors for Neural Nets with different hyper-parameters.

RMS error formula: $RMS_{error} = \sqrt{\frac{1}{n}\sum_{i=n}(y^i - C^i)^2}$

1) We try to find the best learning rate for our model first. We set four different values for the learning rate and calculate RMS error values after 1000 epochs. The results are shown in Table 1 below. Based on the results, we set our learning rates to 0.2 because RMS error is the smallest. This is as expected because if learning rate is too large, the model converges too quickly to a suboptimal result.

Table 1. RMS errors for different learning rates

Learning rates	0.2	0.4	0.6	0.8
RMS error	0.183	0.215	0.202	0.265

2) Next, we will find the best momentum value. We choose four momentum values to test and calculate RMS error values after 1000 epochs. The results are shown in Table 2 below. As we can see from the table, when the momentum increases, RMS errors decrease. RMS error and momentum are negatively correlated. Therefore, we can choose 0.9 for the momentum value which works best for our model.

Table 2. RMS errors for different momentums

Momentum	0.0	0.3	0.6	0.9
RMS error	0.327	0.246	0.206	0.183

3） Finally, we find the best number of hidden neurons. We set four different numbers for hidden neurons and calculate RMS error values after 1000 epochs. The results are shown in Table 3 below and we set the number of hidden neurons to 10.

Table 3. RMS errors for different numbers of hidden neurons

Numbers of hidden neurons	5	10	15	20
RMS error	0.230	0.182	0.217	0.266

In conclusion, the optimal hyper-parameter for our model is learning rate 0.2, momentum 0.9, and number of hidden neurons 10.

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)

In assignment 2, we implemented a look-up table to record all the states and actions. Without state space reduction, the spaced required is 360*1000*360*2*2*7 = 3628800000. If we perform space reduction, it will take only 4*10*4*2*2*7 = 4480 memory spaces. We can save much space by normalizing state values.

However, Neural Network does not face memory space problems because we set the state value as input values directly. Furthermore, for Neural Nets, we do not need to save the whole look-up table as we did in assignment 2. Instead, we only save all the weights connected to neurons which require less memory space.

(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)

The two metrics we chose are winning rates and RMS errors. The results were obtained by first training 20000 rounds of games (with exploration rate = 0.5), and then using our trained model to test another 10000 rounds (with exploration rate = 0.0).

1) Winning Rate

The winning rate is calculated by the proportion of winnings per 500 rounds. The result is Fig.4. shown below. As we can see from the chart, the winning rates of the first 20000 training rounds remain at a low level. However, for the test rounds, winning rates increased dramatically. Our robot did learn during the process.

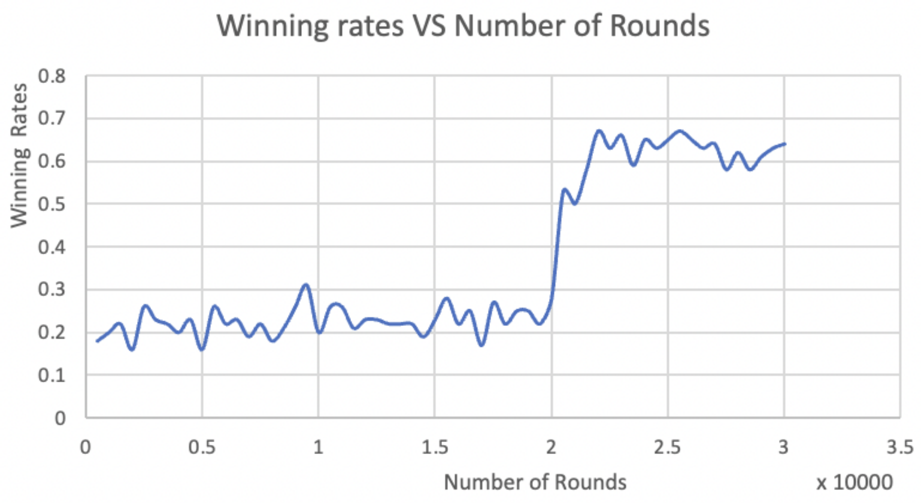


Fig.4. Winning rates VS Number of Rounds

2) RMS errors

The RMS errors are calculated per 500 rounds. The result is in the Fig.5. shown below. As we can see from the chart, RMS error decreases dramatically during the training. Therefore, the robot did learn during online training.

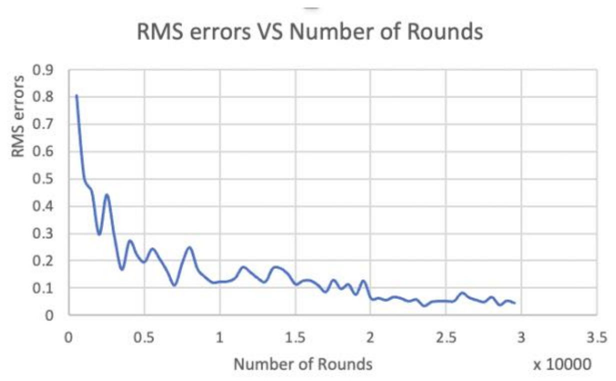


Fig.5. RMS errors VS Number of Rounds

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)

Our hypothesis is that higher γ value is better. If γ is set to 0, this implies that future states would not be taken into consideration in training. Instead, only immediate rewards will be used to predict the next action. This does not follow the Markov Chain rules. A higher γ value encourages long-term rewards while a lower value emphasizes short-term rewards. In the case of robocode, we desire more gains on the long run, so higher γ is expected to be better.

We choose three γ values to verify our hypothesis, which are 0.1, 0.5, and 0.9 respectively. We trained the first 20000 rounds and then tested another 10000 rounds of battles. The results are shown in Fig.6. below. As we can see from the graph, after training, the performance of larger γ value is better. This confirms our prediction.

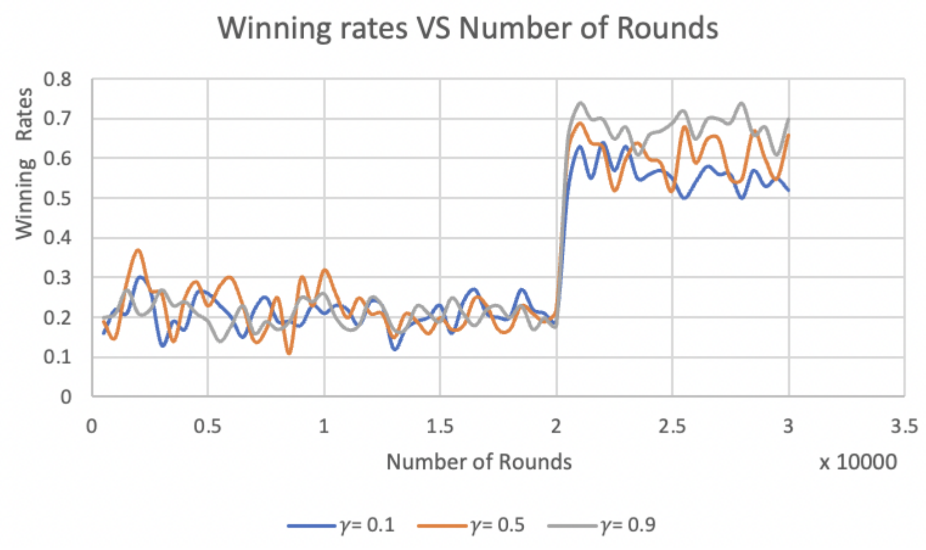


Fig.6. Winning rates VS Number of Rounds

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)

<u>TD learning algorithm converge</u>

Let $\mathrm{e}(\mathrm{s}_\mathrm{t})$ represent the error in the value function at any given state, $V(s_t)$ represent the value function for the state, snd $V*(s_t)$ represent the optimal value function for the state.
We have two equations:

$V(s_t) = \mathrm{e}(\mathrm{s}_\mathrm{t}) + V*(s_t)$ (1)

$V(s_{t+1}) = \mathrm{e}(\mathrm{s}_{\mathrm{t+1}}) + V*(s_{t+1})$ (2)

And Bellman equation is:

$V(s_t) = r_t + \gamma\, V(s_{t+1})$ (3)

Then we substitute $V(s_t)$ and $V*(s_{t+1})$ in equation (3) from equation (1) and (2), and we get the following equations:

$\mathrm{e}(\mathrm{s}_\mathrm{t}) + V*(s_t) = r_t + \gamma\,(\mathrm{e}(\mathrm{s}_{\mathrm{t+1}}) + V*(s_{t+1})\,))$ (4)

Also $V*(s_t) = r_t + \gamma V*(s_{t+1})$ (5)

Apply (4) – (5):

$\mathrm{e}(\mathrm{s}_\mathrm{t}) = \gamma\, \mathrm{e}(\mathrm{s}_{\mathrm{t+1}})$ (6)

From (6), we can draw a conclusion that the errors in the two adjacent states are related. Furthermore, the future rewards themselves are related and these rewards converge after a long training process. So, we can deduce that with sufficient state space and opportunities to learn, error could be zero for all t. Therefore, with a look-up table, the TD learning algorithm is proven to converge.

<u>Approximate Q-function does not guarantee converge</u>

When the value function is approximate, the approximation function causes errors. We label the approximate error as *an 'error'*. Then the optimal value function will be:

$V(s_t) = \mathrm{e}(\mathrm{s}_\mathrm{t}) + V*(s_t) + error$ (7)

Approximate errors may not converge to zero, so $V(s_t)$ may not be converged to $V*(s_t)$. Therefore, we cannot guarantee TD learning algorithm with approximated Q-function could be converged.

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)

Since there is no prior training set, it is hard to use the NN for training robots with online learning. However, there are some alternative ways to monitor the learning performance of the neural net. The possible methods could be winning rate, replay memory and bullet hit counts.

1) Winning rates

The winning rate is one of the most evident factors of learning performance. An increase in winning rate after thousands of rounds implies that robot performs better and does learn in the previous process.

2) Replay memory

Replay memory is also a way to indicate learning performance. In the beginning, the pairs of states and actions are random. After training thousands of rounds, the lists of replay memory will become more and more stable. This is because the learning agent can quickly choose the most suitable pairs of states and actions after learning.

3) Bullet Hit Counts

The Bullet Hit Counts might be another indicator of learning performance. Robot will have a bigger chance to win the game if it gets hit less frequently while hitting the enemy more. This indicates a better performance.

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)*

Replay memory could be implemented to store what the learning agent experiences at each timestamp. Let e_t be agent's experiences at time t and it can be defined as:
$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. This tuple contains the state of the environment, the action taken from the state, the reward given at time t+1, and the next state. The replay memory stores all the agent's experiences at each time step across all rounds. The aim of replay memory is to break the correlation between the samples by taking random samples from replay memory. If the network learned exclusively from successive samples of experience that occurred in sequential order, the samples would be highly correlated, and the learning would be inefficient.

We set three different values of n. We trained 20000 rounds of the game and tested another 10000 rounds of battle. The results are shown in Fig.7. below. According to the results from the line chart, there is no huge gap between the lines, but the winning rate did increase a little bit as the size N becomes bigger.

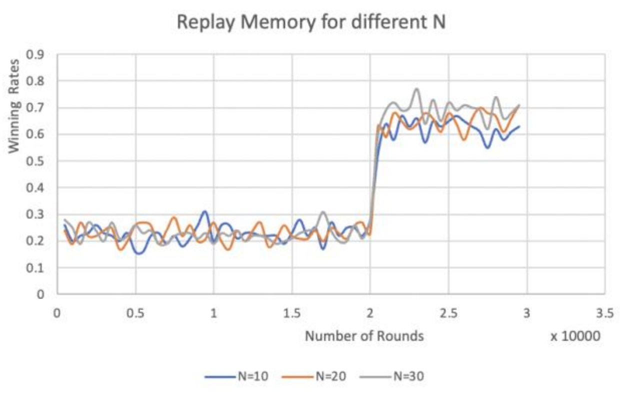


Fig.7. Replay Memory for different N

(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)

Improve the performance of robot:

There are multiple ways to improve the performance of the robot.

1) One way is to have a better choice of states and actions. States and actions choices are essential for the performance of robots. Too many states and actions may result in over-fitting, while too few may result in a bad performance. Some states may be irrelevant to the robot performance, while others can have a huge impact. Therefore, we can experiment more to possibly weigh different state values and improve the model.

2) In the phase of choosing hyper-parameters, my approach was to use 4 values for each hyper-parameter and choose independently. However, an improvement can be made to make use of grid search, I.e. experiment combinations of the three hyper-parameters and find the best result.

Convergence problem solving suggestions:

1) Setting appropriate hyper-parameters for the Neural Net. As we mentioned before, choosing proper hyper-parameters may decrease the RMS errors which also means they can help convergence as well. We can take the learning rate (step size) for example. Too big learning rates may cause the model to miss the expected value (the global minimum) and we may only get local minimum. Too small learning rates may result in slow convergence. Therefore, choosing proper hyper-parameters are important for convergence.

2) Rescaling the inputs could help with convergence as well. The input values are very imbalanced. Some input values are very small, such as isHitByBullets with only two separate values. However, some input values are very large, e.g. the distance between robot and the enemy can take 1000 different values. A huge difference in input value scales may slow down convergence. Therefore, input normalization is a key to convergence.

3) Proper activation functions could also help to solve convergence problems. For assignment 1, we found that Neural Net with bipolar activation function converges much faster than the one with binary activation function. For robocode, we could try various activation functions for our Neural Net such as Tanh activation function and Swish activation function.

Other practical application advice

When applying RL with neural networks to other practical applications, we could follow several rules. The first piece of advice is to choose appropriate states and actions. States and actions are essential for learning. The second piece of advice is to select appropriate hyper-parameters. The third piece of advice is to choose proper activation function. The last suggestion is to normalize the input and output values. This will help improve the results of reinforcement learning with neural nets.

b) Theory question: Imagine a closed-loop control system for automatically delivering anesthetic to a patient undergoing 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)

Concerns:

1) The health conditions of patients and the corresponding amounts of anesthetic are complicated. Various input values should be considered such as weights, age, blood pressure, and other biomedical factors. And it is also challenging to extract and normalize biological information from pathology reports.

2) Surgery accidents may occur when applying online training to the closed-loop control system. In early stages of learning process, we would expect to see big differences between predicted values and actual values. However, any errors or misbehaviors during surgery may cause surgery failure or even threaten patient's life. This brings some ethical concerns.

Variations:

The variation is to develop a surgery information tool that contains all the pathology information of patients. So, this powerful tool combined with Neural Net RL offline training Model can give doctors and patients advice about the type and amounts of anesthetic that should be injected into the patients. Doctors can double-check the analysis results and make some adjustments as well.

Appendix:

Action.java

```
package ece.assign3;

public class Action {
    public static final int ROBOT_UP = 0;
    public static final int ROBOT_UP_LONG = 1;
    public static final int ROBOT_DOWN = 2;
    public static final int ROBOT_DOWN_LONG = 3;
    public static final int ROBOT_LEFT = 4;
    public static final int ROBOT_RIGHT = 5;
    public static final int ROBOT_FIRE = 6;
    public static final int ROBOT_NUM_ACTIONS = 7;
    public static final double ROBOT_MOVE_SHORT_DISTANCE = 100.0;
    public static final double ROBOT_MOVE_LONG_DISTANCE = 300.0;
    public static final double ROBOT_TURN_DEGREE =  30.0;
}
```

Experience.java

```
package ece.assign3;

public class Experience {
    double[] currState;
    int action;
    double reward;
    double[] nextState;
}
```

CircularQueue.java

```
package ece.assign3;

import java.util.LinkedList;

/**
 * Author: Sarbjit Sarkaria
 * Date  : 07th January 2021
 * This class implements a replay memory for any type T.
 * The capacity of the memory must be specified upon construction.
 * The memory will discard the oldest items that do not fit.
 * @param <T> Type to be managed by the ReplayMemory
 */

public class CircularQueue<T> extends LinkedList<T> {
    private int capacity = 10;

    public CircularQueue(int capacity){
        this.capacity = capacity;
    }

    public boolean add(T e) {
        if(size() >= capacity)
            removeFirst();
        return super.add(e);
    }

    public Object [] toArray() {
        return super.toArray();
    }
}
```

ReplayMemory.java

```
package ece.assign3;

import java.util.Arrays;
import java.util.Collections;

/**
 * Author: Sarbjit Sarkaria
 * Date  : 07th January 2021
 * This class implements a replay memory for any type T.
 * The capacity of the memory must be specified upon construction.
 * The memory will discard the oldest items that do not fit.
 * @param <T> Type to be managed by the ReplayMemory
 */

public class ReplayMemory<T> {

    private CircularQueue<T> memory;
    private Object[] EMPTYARRAY = {};

    // Constructor
    public ReplayMemory(int size) {
        memory = new CircularQueue<T>(size);
    }

    // Add an item to the memory
    public void add(T experience) {
        memory.add(experience);
    }

    // Retrieve a sample of n most recently added items from the memory and
return it as an array
    public Object[] sample(int n) {
        if (memory.isEmpty())
            return EMPTYARRAY;
        else {
            // I don't have a way of returning T[], so instead I return
Object[]
            // See the unit tests to see how to use this tyle - no additional
effort necessary
            // .... see here for more :
https://stackoverflow.com/questions/1115230/casting-object-array-to-integer-
array-error
            int size = memory.size();
            Object[] objectArray = memory.toArray();
            Object[] sampleObjectArray = Arrays.copyOfRange(objectArray,
size-n, size);
            return sampleObjectArray;
        }
    }

    // Retrieve a random sample of n items from the memory and return it as
an array
    public Object[] randomSample(int n) {
        if (memory.isEmpty())
            return EMPTYARRAY;
        else {
            // I don't have a way of returning T[], so instead I return
Object[]
            // See the unit tests to see how to use this tyle - no additional
effort necessary
            // .... see here for more :
https://stackoverflow.com/questions/1115230/casting-object-array-to-integer-
array-error
            int size = memory.size();
            CircularQueue shuffledMemory = new CircularQueue<T>(size);
            shuffledMemory.addAll(memory); // This does a deep copy
            Collections.shuffle(shuffledMemory);
            Object[] objectArray = shuffledMemory.toArray();
            Object[] sampleObjectArray = Arrays.copyOfRange(objectArray,
size-n, size);
            return sampleObjectArray;
        }
    }

    // Returns the current size of the replay memory. Use for test/debug
purposes
    public int sizeOf() {
        if (memory.isEmpty())
            return 0;
        else
            return memory.size();
    }
}
```

Enemy.java
```java
package ece.assign3;

import java.awt.geom.Point2D;

public class Enemy {
    public String name;
    public double bearing;
    public double heading;
    public double changeHeading;
    public double x, y;
    public double distance, speed;
    public long ctime;

    public Enemy(String name) {
        this.name = name;
    }

    public Point2D.Double getNextPosition(long gaussTime) {
        double diff = gaussTime - ctime;
        double nextX = x + Math.sin(heading) * speed * diff;
        double nextY = y + Math.cos(heading) * speed * diff;
        return new Point2D.Double(nextX, nextY);
    }
}
```

Layer.java
```java
package ece.assign3;

import java.util.Arrays;

public class Layer {
    private boolean IS_BINARY = false;
    private static final String RANDOM = "random";
    private static final String ZERO = "zero";
    public static final String INPUT_LAYER = "input layer";
    public static final String HIDDEN_LAYER = "hidden layer";
    public static final String OUTPUT_LAYER = "output layer";
    private int size;
    private String type;
    private Layer prev;
    private Layer next;
    private double[] values;
    private double[] deltas;
    private double[][] currWeights;
    // only for output layer
    private double[] expectedOutputs;
    private double[][] deltaWeights;
    public Layer(int size, String type){
        this.size = size;
        this.type = type;
        this.deltas = new double[size];
        if(type.equals(OUTPUT_LAYER)) {
            this.values = new double[size];
        } else {
            this.values = new double[size+1];
            values[size] = 1.0; // bias neuron
        }
    }
    // set input data
    public void setInputs(double[] inputs) {
        if(!type.equals(INPUT_LAYER) || inputs.length != size) {
            return;
        }
        for(int i=0; i<size; i++) {
            values[i] = inputs[i];
        }
    }
    // set output data
    public void setOutputs(double[] outputs) {
        if(!type.equals(OUTPUT_LAYER) || outputs.length != size) {
            return;
        }
        expectedOutputs = new double[size];
        for(int i=0; i<size; i++) {
            expectedOutputs[i] = outputs[i];
        }
    }
    // connect current layer with prev layer and next layer
    public void connectLayer(Layer prevLayer, Layer nextLayer) {
        prev = prevLayer;
        next = nextLayer;
        // outputLayer
        if(type.equals(OUTPUT_LAYER)) {
            currWeights = null;
            deltaWeights = null;
            expectedOutputs = new double[size];
        } else {
            currWeights = new double[size+1][next.size];
            deltaWeights = new double[size+1][next.size];
            expectedOutputs = null;
        }
    }
    public double getRandom(double lowerBound, double upperBound) {
        return lowerBound + (upperBound-lowerBound)*Math.random();
    }
    public void setWeights(String setValue, double lowerBound, double
            upperBound) {
```

```
        if(type.equals(OUTPUT_LAYER)) {
            return;
        }
        for (int j = 0; j < next.size; j++) {
            for (int i = 0; i < size + 1; i++) {
                if(setValue == RANDOM) {
                    currWeights[i][j] = getRandom(lowerBound, upperBound);
                } else if(setValue == ZERO) {
                    currWeights[i][j] = 0.0;
                }
                deltaWeights[i][j] = 0.0;
            }
        }
    }
    public void initializeWeights(double lowerBound, double upperBound) {
        setWeights(RANDOM, lowerBound, upperBound);
    }
    public void zeroWeights() {
        setWeights(ZERO, 0, 0);
    }
    public double sigmoid(double x) {
        return 2/(1+Math.exp(-x))-1;
    }
    public double customSigmoid(double x) {
        if (IS_BINARY) {
            return 1/(1+Math.exp(-x));
        }
        return sigmoid(x);
    }
    public void forwardPropagation() {
        if(type.equals(OUTPUT_LAYER)) return;
        for (int j=0; j<next.size; j++) {
            next.values[j] = 0.0;
            // include current layer bias neuron to predict next layer values
            for(int i=0; i<size+1; i++) {
                next.values[j] += currWeights[i][j]*values[i];
            }
            next.values[j] = customSigmoid(next.values[j]);
        }
    }
    public double getDerivative(double value) {
        if(IS_BINARY) {
            return value*(1-value);
        }
        return (value+1)*(1-value)*0.5;
    }

    public void backwardPropagation(double learningRate, double momentum) {
        if(type.equals(INPUT_LAYER)) {
            return;
        }
        if(type.equals(OUTPUT_LAYER)) {
            for(int i=0; i<size; i++) {
                double error = expectedOutputs[i]-values[i];
                deltas[i] = getDerivative(values[i])*error;
            }
        } else if(type.equals(HIDDEN_LAYER)) {
```

```java
            for(int i=0; i<size; i++) {
                deltas[i] = 0;
                for(int j=0; j<next.size; j++) {
                    deltas[i] += currWeights[i][j]*next.deltas[j];
                }
                deltas[i] = getDerivative(values[i])*deltas[i];
            }
        }
        for(int j=0; j<size; j++) {
            for(int i=0; i <= prev.size; i++) {
                prev.deltaWeights[i][j] = momentum*prev.deltaWeights[i][j] +
learningRate*deltas[j]*prev.values[i];
                prev.currWeights[i][j] += prev.deltaWeights[i][j];
            }
        }
    }

    public Layer getNext() {
        return next;
    }
    public Layer getPrev() {
        return prev;
    }
    public double[] getValues() {
        return values;
    }
    public String saveCurrWeightsToString() {
        String str = "";
        for (int j = 0; j < next.size; j++) {
            for (int i = 0; i < size + 1; i++) {
                str += currWeights[i][j] + "-";
            }
        }
        return str;
    }

    public void updateCurrWeights(String str) {
        String[] weights = Arrays.stream(str.split("-")).filter(e ->
e.trim().length() > 0).toArray(String[]::new);
        int n = weights.length;
        int id = 0;
        if(n == next.size*(size+1)) {
            System.out.println("validation: success ..." );
        } else {
            System.out.println("validation: fails ...");
        }
        for (int j = 0; j < next.size; j++) {
            for (int i = 0; i < size + 1; i++) {
                currWeights[i][j] = Double.valueOf(weights[id]);
                System.out.print(currWeights[i][j] + "-");
                id++;
            }
        }
        System.out.println();
    }
}
```

NeuralNet.java

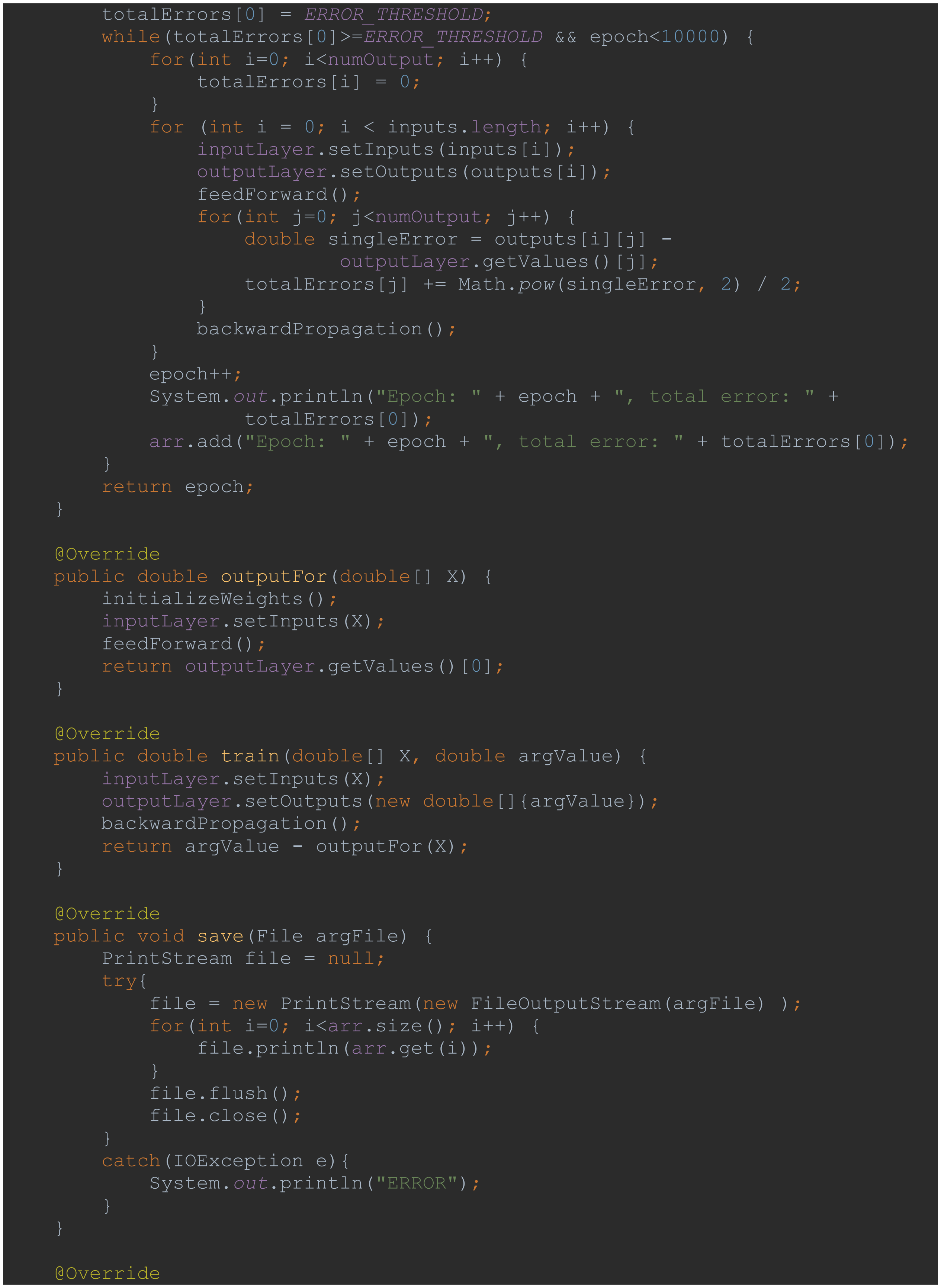
```java
package ece.assign3;

import robocode.RobocodeFileOutputStream;
import robocode.RobocodeFileWriter;

import java.io.*;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;

public class NeuralNet implements NeuralNetInterface {
    private static final double ERROR_THRESHOLD = 0.05;
    private Layer inputLayer;
    private Layer hiddenLayer;
    private Layer outputLayer;
    private double learningRate;
    private double momentum;
    private int numInput;
    private int numHidden;
    private int numOutput;
    private double lowerBound = -0.5;
    private double upperBound = 0.5;
    private ArrayList<String> arr = new ArrayList<>();
    public NeuralNet(int numInput, int numHidden, int numOutput, double
            learningRate, double momentum) {
        this.numInput = numInput;
        this.numHidden = numHidden;
        this.numOutput = numOutput;
        this.learningRate = learningRate;
        this.momentum = momentum;
        setUpNeuralNet();
    }
    public void setUpNeuralNet() {
        inputLayer = new Layer(numInput, Layer.INPUT_LAYER);
        hiddenLayer = new Layer(numHidden, Layer.HIDDEN_LAYER);
        outputLayer = new Layer(numOutput, Layer.OUTPUT_LAYER);
        inputLayer.connectLayer(null, hiddenLayer);
        hiddenLayer.connectLayer(inputLayer, outputLayer);
        outputLayer.connectLayer(hiddenLayer, null);
    }

    public int trainWholeProcess(double[][] inputs, double[][] outputs) {
        initializeWeights();
        double[] totalErrors = new double[numOutput];
        int epoch = 0;
```



```
        totalErrors[0] = ERROR_THRESHOLD;
        while(totalErrors[0]>=ERROR_THRESHOLD && epoch<10000) {
            for(int i=0; i<numOutput; i++) {
                totalErrors[i] = 0;
            }
            for (int i = 0; i < inputs.length; i++) {
                inputLayer.setInputs(inputs[i]);
                outputLayer.setOutputs(outputs[i]);
                feedForward();
                for(int j=0; j<numOutput; j++) {
                    double singleError = outputs[i][j] -
                            outputLayer.getValues()[j];
                    totalErrors[j] += Math.pow(singleError, 2) / 2;
                }
                backwardPropagation();
            }
            epoch++;
            System.out.println("Epoch: " + epoch + ", total error: " +
                    totalErrors[0]);
            arr.add("Epoch: " + epoch + ", total error: " + totalErrors[0]);
        }
        return epoch;
    }

    @Override
    public double outputFor(double[] X) {
        initializeWeights();
        inputLayer.setInputs(X);
        feedForward();
        return outputLayer.getValues()[0];
    }

    @Override
    public double train(double[] X, double argValue) {
        inputLayer.setInputs(X);
        outputLayer.setOutputs(new double[]{argValue});
        backwardPropagation();
        return argValue - outputFor(X);
    }

    @Override
    public void save(File argFile) {
        PrintStream file = null;
        try{
            file = new PrintStream(new FileOutputStream(argFile) );
            for(int i=0; i<arr.size(); i++) {
                file.println(arr.get(i));
            }
            file.flush();
            file.close();
        }
        catch(IOException e){
            System.out.println("ERROR");
        }
    }

    @Override
    public void load(String argFileName) throws IOException {
    }

    @Override
    public void initializeWeights() {
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
                currLayer.getNext()) {
            currLayer.initializeWeights(lowerBound, upperBound);
        }
    }

    @Override
    public void zeroWeights() {
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
                currLayer.getNext()) {
            currLayer.zeroWeights();
        }
    }

    public void feedForward() {
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
currLayer.getNext()) {
            currLayer.forwardPropagation();
        }
    }

    public void backwardPropagation() {
        for(Layer currLayer=outputLayer; currLayer != inputLayer; currLayer =
currLayer.getPrev()) {
            currLayer.backwardPropagation(learningRate, momentum);
        }
    }

    public void saveWeights(File file) {
        String[] strs = new String[2];
        //PrintStream w = null;
        int i = 0;
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
currLayer.getNext()) {
            strs[i] = currLayer.saveCurrWeightsToString();
            i++;
        }
        try{
            RobocodeFileWriter fileWriter = new
RobocodeFileWriter(file.getAbsolutePath(), false);
            fileWriter.write(strs[0] + "\r\n");
            fileWriter.write(strs[1] + "\r\n");
            fileWriter.close();
        }
        catch(Exception e){
            System.out.println(e);
        }
    }

    public void loadWeights(File file) {
        String[] strs = new String[2];
        try {
```

```java
            BufferedReader br = new BufferedReader(new
FileReader(file.getAbsoluteFile()));
            for(int i=0; i<2; i++) {
                try {
                    strs[i] = br.readLine();
                    System.out.println(strs[i]);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        int i = 0;
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
currLayer.getNext()) {
            currLayer.updateCurrWeights(strs[i]);
            i++;
        }
    }


}
```

NNRobot.java

```java
package ece.assign3;

import robocode.*;

import java.awt.*;
import java.awt.geom.Point2D;
import java.io.File;

public class NNRobot extends AdvancedRobot {
    private static final boolean RECORD_MEMORY = false;
    private static final double IMMEDIATE_REWARD = 5;
    private static final double TERMINAL_REWARD = 15;
    private static final double TERMINAL_FINE = -10;
    private static final double IMMEDIATE_FINE = -4;
    private static final double ENEMY_DEATH_REWARD = 6;
    private static final int NUM_INPUT_LAYERS = 5;
    private static final int NUM_HIDDEN_LAYERS = 20;
    private static final int NUM_OUTPUT_LAYERS = 1;
    private static final double NN_LEARNING_RATE = 0.2;
    private static final double NN_MOMENTUM = 0.6;
    private static final double NN_ALPHA = 0.1;
    private static final double NN_GAMMA = 0.9;
    private static final double NN_EXPLORATION_RATE = 0.5;
    private static final int PERIOD = 100;
    private static final int MEMORY_N = 10;
    private Enemy enemy;
    private double reward = 0.0;
    private double firePower = 1;
    private int isHitByBullet = 0;
    private int isHitWall = 0;
    int action;
    public static double enemyBearing;
    public static NeuralNet[] nn = new NeuralNet[Action.ROBOT_NUM_ACTIONS];
    public static ReplayMemory<Experience> memory = new
ReplayMemory<>(MEMORY_N);
    private double[] NNPrevState = new double[5];
    private int NNPrevAction = 0;
    private static int numRounds = 0;
    private static int winRounds = 0;
    private static String scoreListFile = "scoreList.txt";;
    private File weightFiles[] = new File[Action.ROBOT_NUM_ACTIONS];
    private double RMS_Error = 0.0d;
    private double totalError = 0.0d;

    public void run() {
        setUpNeuralNets();
//        if(numRounds > 0) {
//            loadNN_Weights();
//        }
        loadNN_Weights();
        enemy = new Enemy("enemy");
        enemy.distance = 10000;

        setAllColors(Color.red);
        setAdjustGunForRobotTurn(true);
        setAdjustRadarForGunTurn(true);

        while(true) {
            radarMovement();
            gunMovement();
            robotMovement();
            execute();
        }
    }

    public void setUpNeuralNets(){
        for(int i = 0; i< Action.ROBOT_NUM_ACTIONS; i++){
            nn[i]=new NeuralNet(NUM_INPUT_LAYERS, NUM_HIDDEN_LAYERS,
NUM_OUTPUT_LAYERS, NN_LEARNING_RATE, NN_MOMENTUM);
        }
    }

    public int getMaxAction(double[] currState) {
        int nextAction = 0;
        for(int i=0; i<Action.ROBOT_NUM_ACTIONS; i++) {
            if(nn[i].outputFor(currState) >
nn[nextAction].outputFor(currState)) {
                nextAction = i;
            }
        }
        return nextAction;
    }

    public int getNNAction(double heading, double distance, double bearing,
double reward) {
        double[] NNCurrStates = new double[5];
        NNCurrStates[0] = heading % 180;
        NNCurrStates[1] = distance % 10000;
        NNCurrStates[2] = bearing % Math.PI;
        NNCurrStates[3] = isHitByBullet;
        NNCurrStates[4] = isHitWall;
        int nextAction = getMaxAction(NNCurrStates);

        double NN_NewVal = nn[nextAction].outputFor(NNCurrStates);
        double NN_PrevVal = nn[NNPrevAction].outputFor(NNPrevState);
        double error = NN_ALPHA * (reward + NN_GAMMA * NN_NewVal -
NN_PrevVal);
        RMS_Error += error*error;
        //totalError += error*error/2;

        double NN_CorrectPrevVal = NN_PrevVal + error;
        nn[NNPrevAction].train(NNPrevState, NN_CorrectPrevVal);
        if(RECORD_MEMORY) {
            Experience e = new Experience();
            e.currState = NNPrevState;
            e.action = NNPrevAction;
            e.reward = reward;
            e.nextState = NNCurrStates;
            memory.add(e);
        }
        for(int i=0; i<NUM_INPUT_LAYERS; i++) {
            NNPrevState[i] = NNCurrStates[i];
        }
        NNPrevAction = nextAction;
        if(Math.random() < NN_EXPLORATION_RATE) {
            return (int)(Math.random() * Action.ROBOT_NUM_ACTIONS);
        }
        return nextAction;
    }

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

    public void robotMovement() {
        action = getNNAction(getHeading(), enemy.distance, enemy.bearing,
reward);
        reward = 0.0;
        isHitWall = 0;
        isHitByBullet = 0;
        switch (action) {
            case Action.ROBOT_UP:
                setAhead(ece.cpen502.Action.ROBOT_MOVE_SHORT_DISTANCE);
                break;
            case Action.ROBOT_UP_LONG:
                setAhead(ece.cpen502.Action.ROBOT_MOVE_LONG_DISTANCE);
```

```java
                break;
            case Action.ROBOT_DOWN:
                setBack(ece.cpen502.Action.ROBOT_MOVE_SHORT_DISTANCE);
                break;
            case Action.ROBOT_DOWN_LONG:
                setBack(ece.cpen502.Action.ROBOT_MOVE_LONG_DISTANCE);
                break;
            case Action.ROBOT_LEFT:
                setTurnLeft(ece.cpen502.Action.ROBOT_TURN_DEGREE);
                break;
            case Action.ROBOT_RIGHT:
                setTurnRight(ece.cpen502.Action.ROBOT_TURN_DEGREE);
                break;
            case Action.ROBOT_FIRE:
                //turnGunRight(getHeading() - getGunHeading() +
enemyBearing);
                setFire(firePower);
                break;
        }
    }

    public void onScannedRobot(ScannedRobotEvent e) {
        if ((enemy.name == e.getName()) || (e.getDistance() <
enemy.distance)) {
            enemy.name = e.getName();
            double bearingRadius = (getHeadingRadians() +
e.getBearingRadians()) % (2 * Math.PI);
            double heading = normaliseBearing(e.getHeadingRadians() -
enemy.heading);
            heading /= (getTime() - enemy.ctime);
            enemy.changeHeading = heading;
            enemy.distance = e.getDistance();
            enemy.x = Math.sin(bearingRadius) * enemy.distance + getX();
            enemy.y = Math.cos(bearingRadius) * enemy.distance + getY();
            enemy.ctime = getTime();
            enemy.speed = e.getVelocity();
            enemy.bearing = e.getBearingRadians();
            enemy.heading = e.getHeadingRadians();
            enemyBearing = e.getBearing();
        }
    }

    private void gunMovement() {
        long gaussTime, nextTime;
        double gunOffset;
        Point2D.Double p = new Point2D.Double(enemy.x, enemy.y);
        for (int i=0; i<20; i++) {
            nextTime = (int)Math.round((getEuDistance(getX(),getY(),p.x,p.y)
/ (20 - (3 * firePower))));
            gaussTime = getTime() + nextTime - 10;
            p = enemy.getNextPosition(gaussTime);
        }

        gunOffset = normaliseBearing(getGunHeadingRadians() -
                (Math.PI/2 - Math.atan2(p.y - getY(),p.x -  getX())));
        setTurnGunLeftRadians(gunOffset);
    }
    public double getEuDistance(double x1, double y1, double x2, double y2) {
        double x = x1 - x2;
        double y = y1 - y2;
        return Math.sqrt(x * x + y * y);
    }

    double normaliseBearing(double degree) {
        if (degree > Math.PI) {
            degree -= 2*Math.PI;
        }
        if (degree < -Math.PI) {
            degree += 2*Math.PI;
        }
        return degree;
    }

    public void onHitWall(HitWallEvent e){
        isHitWall = 1;
        reward += IMMEDIATE_FINE / 2;
        getNNAction(getHeading(), enemy.distance, enemy.bearing, reward);
        stepBack();
    }

    public void onBulletHit(BulletHitEvent e) {
        reward += IMMEDIATE_REWARD;
        getNNAction(getHeading(), enemy.distance, enemy.bearing, reward);
        //trainReplayMemory();
    }

    public void onHitByBullet(HitByBulletEvent e) {
        isHitByBullet = 1;
        reward += IMMEDIATE_FINE;
        getNNAction(getHeading(), enemy.distance, enemy.bearing, reward);
        //trainReplayMemory();
    }

    public void onBulletMissed(BulletMissedEvent e) {
        reward += IMMEDIATE_FINE / 2;
        getNNAction(getHeading(), enemy.distance, enemy.bearing, reward);
    }

    public void onRobotDeath(RobotDeathEvent e) {
        reward += ENEMY_DEATH_REWARD;
        getNNAction(getHeading(), enemy.distance, enemy.bearing, reward);
    }

    public void stepBack() {
        setBack(180);
        setTurnRight(60);
        execute();
    }

    public void onDeath(DeathEvent event) {
        numRounds++;
        Statistics.saveScore(0);
        getNNAction(getHeading(), enemy.distance, enemy.bearing,
TERMINAL_FINE);
        recordScores();
        saveNN_Weights();
    }

    public void onWin(WinEvent event) {
        numRounds++;
        winRounds++;
        Statistics.saveScore(1);
        getNNAction(getHeading(), enemy.distance, enemy.bearing,
TERMINAL_REWARD);
        recordScores();
        saveNN_Weights();
    }

    public void trainReplayMemory() {
        Object[] experiences = memory.randomSample(MEMORY_N/2);
        for(Object e: experiences) {
            Experience x = (Experience) e;
            int nextAction = getMaxAction(x.currState);
            double NN_NewVal = nn[nextAction].outputFor(x.nextState);
            double NN_PrevVal = nn[x.action].outputFor(x.currState);
            double error = NN_ALPHA * (x.reward + NN_GAMMA * NN_NewVal -
NN_PrevVal);
            double NN_CorrectPrevVal = NN_PrevVal + error;
            nn[action].train(x.currState, NN_CorrectPrevVal);
        }
    }


    public void recordScores() {
        if(numRounds % PERIOD == 0 && numRounds != 0) {
            double winRate = ((double)winRounds) / PERIOD;
            int round = numRounds / PERIOD;
            winRounds = 0;
            try{
                File file = getDataFile(scoreListFile);
                RobocodeFileWriter fileWriter = new
RobocodeFileWriter(file.getAbsolutePath(), true);
                RMS_Error = Math.sqrt(RMS_Error/numRounds);
                fileWriter.write(String.format("alpha:%s gamma:%s hidden:%s
round:%s score:%s RMS error: %s\n",
                        NN_LEARNING_RATE, NN_MOMENTUM, NUM_HIDDEN_LAYERS,
round, winRate, RMS_Error));
                fileWriter.close();
            } catch(Exception e) {
                System.out.println(e);
            }
        }
    }

    public void onBattleEnded(BattleEndedEvent event) {
        Statistics.printWinRates();
    }

    public void saveNN_Weights() {
        for(int i=0; i<Action.ROBOT_NUM_ACTIONS; i++) {
            String weightsPath = "weights" + i + ".txt";
            weightFiles[i] = getDataFile(weightsPath);
            nn[i].saveWeights(weightFiles[i]);
        }
    }

    public void loadNN_Weights() {
        for(int i=0; i<Action.ROBOT_NUM_ACTIONS; i++) {
            String weightsPath = "weights" + i + ".txt";
            weightFiles[i] = getDataFile(weightsPath);
            try {
                nn[i].loadWeights(weightFiles[i]);
            } catch(Exception e) {
                System.out.println(e);
            }
        }
    }
}
```

State.java

```
package ece.assign3;

public class State {
    public static final double TOTAL_ANGLE = 360.0;
    public static final double CIRCLE = Math.PI * 2;
    public static int NumStates;
    public static final int NUM_DISTANCE = 10;
    public static final int NUM_BREARING = 4;
    public static final int NUM_HEADING = 4;
    public static final int NUM_HIT_BY_BULLETS = 2;
    public static final int NUM_HIT_WALL = 2;
    public static final int NUM_ENERGY = 5;
    public static int states[][][][][];

    static {
        states = new
int[NUM_DISTANCE][NUM_BREARING][NUM_HEADING][NUM_HIT_BY_BULLETS][NUM_HIT_WALL
];
        int cnt = 0;
        for(int a=0; a<NUM_DISTANCE; a++) {
            for(int b=0; b<NUM_BREARING; b++) {
                for(int c=0; c<NUM_HEADING; c++) {
                    for(int d=0; d<NUM_HIT_BY_BULLETS; d++) {
                        for(int e=0; e<NUM_HIT_BY_BULLETS; e++) {
                            states[a][b][c][d][e] = cnt++;
                        }
                    }
                }
            }
        }
        NumStates = cnt;
    }
    public static int getDistance(double distance) {
        int res = (int)(distance / 100.0);
        return Math.min(NUM_DISTANCE-1, res);
    }

    public static int getHeading(double heading) {
        double angle = TOTAL_ANGLE / NUM_HEADING;
        double newHeading = heading+angle/2;
        while (newHeading > TOTAL_ANGLE) {
            newHeading-=TOTAL_ANGLE;
        }
        return (int)(newHeading/angle);
    }

    public static int getBearing(double bearing) {
        double angle=CIRCLE / NUM_BREARING;
        double newBearing = bearing;
        if(bearing < 0) {
            newBearing += CIRCLE;
        }
        newBearing += angle / 2;
        if(newBearing > CIRCLE) {
            newBearing = newBearing - CIRCLE;
        }
        return (int) (newBearing / angle);
    }

    public static int getEnergyLevel(double energy) {
        double levels = 100 / NUM_ENERGY;
        return Math.min((int)(energy/levels), NUM_ENERGY-1);
    }
}
```

LookUpTable.java

```
package ece.assign3;

public class LookUpTable {
    private double table[][];

    public LookUpTable() {
        this.table = new double[State.NumStates][Action.ROBOT_NUM_ACTIONS];
        initializeLUT();
    }

    public void initializeLUT() {
        for(int i = 0; i< State.NumStates; i++) {
            for(int j = 0; j< Action.ROBOT_NUM_ACTIONS; j++) {
                table[i][j] = Math.random();
            }
        }
    }

    public double getQValue(int state, int action) {
        return table[state][action];
    }

    public void setQValue(int state, int action, double value) {
        this.table[state][action] = value;
    }

    public double getMaxValue(int state) {
        double maxValue = -10;
        for(int i = 0; i< ece.cpen502.Action.ROBOT_NUM_ACTIONS; i++) {
            maxValue = Math.max(table[state][i], maxValue);
        }
        return maxValue;
    }

    public int getBestAction(int state) {
        double maxValue = -100000;
        int action = 0;
        for(int i = 0; i< Action.ROBOT_NUM_ACTIONS; i++) {
            if(table[state][i] > maxValue) {
                maxValue = table[state][i];
                action = i;
            }
        }
        return action;
    }
}
```