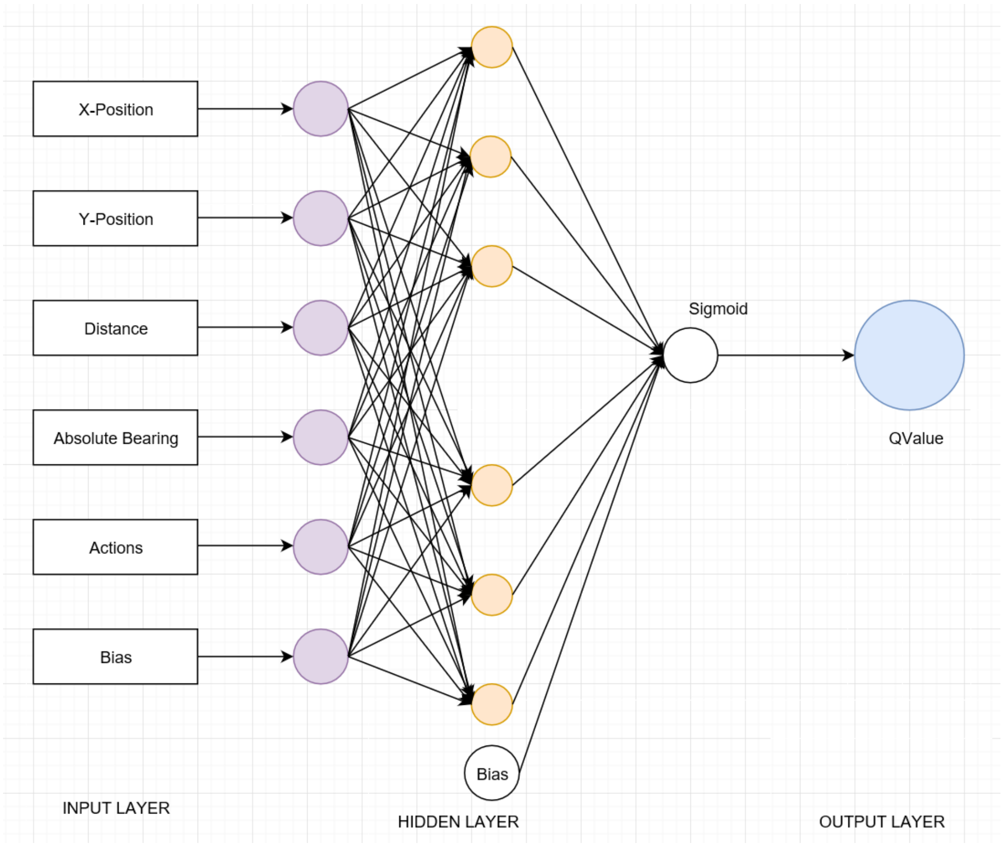
Student Name: Ravneet Kaur Student Number: 41580994

CPEN-502 101

Part 3 - Reinforcement Learning with Backpropagation

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

4.1 Describe the architecture of your neural network and how the training set captured from Part 2 was used to "offline" train it mentioning any input representations that you may have considered. Note that you have 3 different options for the high level architecture. A net with a single Q output, a net with a Q output for each action, separate nets each with a single output for each action. Draw a diagram for your neural net labeling the inputs and outputs.



Figure(a): The Neural Network

The Input Layer of the Neural Net consists of 6 inputs, 1 of which is a Bias. And the single output value is used i.e. the QValue which is resized according to the minQValue or the maxQValue. The Sigmoid activation function is used with the hidden layer. The inputs and the outputs for training of the Neural net is done using the state action values obtained from the LUT in part 2. And the bipolar activation function is used for this neural network, and the normalized values scaled between -1,1.

The input representations considered to train the robot from the part 2 of the assignment are specified as the states and the actions for the input layer which are given as follows:

States:

1. X-Position of the Robot
2. Y-Position of the Robot
3. Distance Between my robot and the enemy robot
4. Absolute Bearing angle between my robot and the enemy robot

Actions:

0. Moving in the direction of the enemy
1. Moving in the opposite direction to the enemy
2. Move opposite to enemy and circle in clockwise direction
3. Move opposite to enemy and circle in anticlockwise direction
4. Moving towards enemy and circle in clockwise direction
5. Moving towards enemy and circle in anticlockwise direction

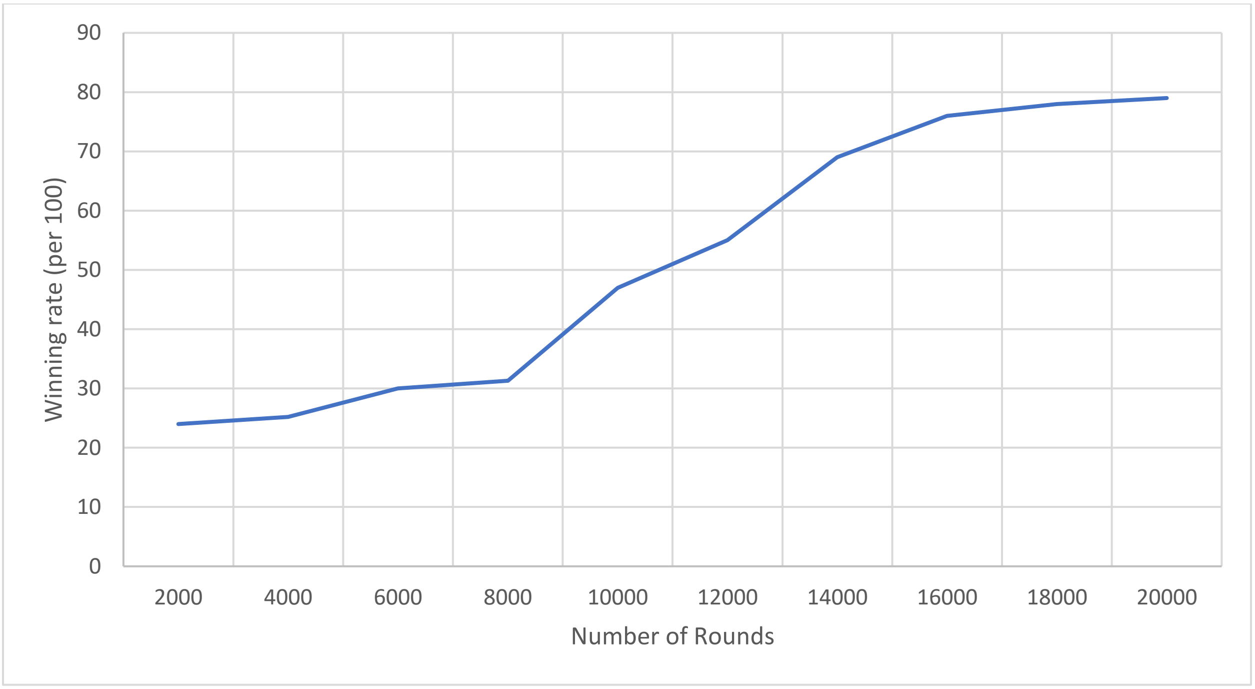
<u>Input Representation</u>

	Input	Non Quantized Value	NN
1.	X-Position	0-800 pixels	0,0.01,………,7.99,8
2.	Y-Position	0-600 pixels	0,0.01,………,5.99,6
3.	Distance between my robot and enemy robot	0-1000 pixels	0,0.01,0.02,…….9.99,10.00
4.	Absolute bearing angle between my robot and enemy robot	0-360^o	0,0.01,………3.99,4.00
5.	Actions	0,1,2,3,4,5	0,1,2,3,4,5

Table(a): Input Representation

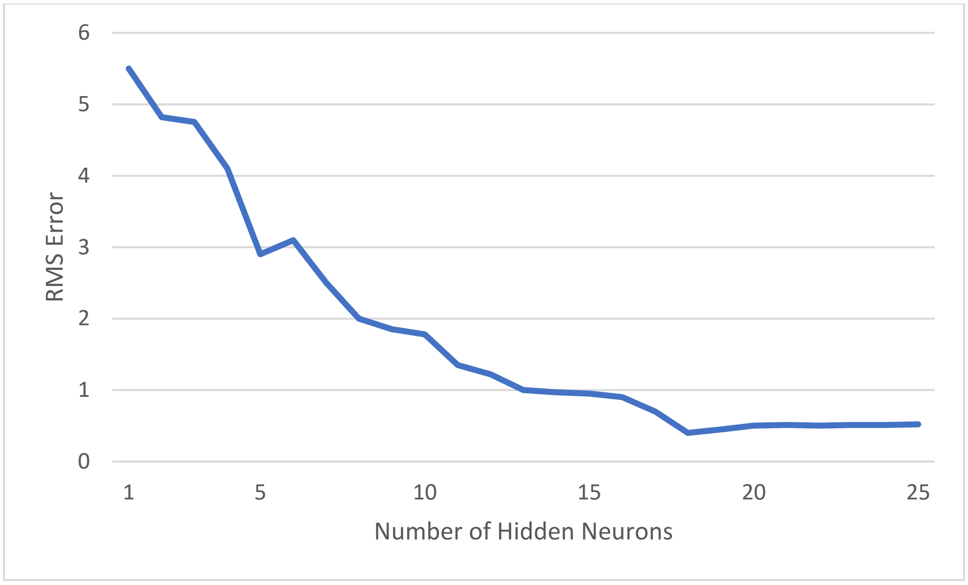
4.2 Show (as a graph) the results of training your neural network using the contents of the LUT from Part 2. Include how you arrived at the parameters that best learned your LUT data. You may have attempted learning using different hyper-parameter values (i.e. momentum, learning rate, number of hidden neurons). Include graphs showing which parameters best learned your LUT data. Compute the RMS error for your best results.

In the part-2 assignment of the course, the winning rate observed is approximately 80% after training the robot for nearly 25000 battles.



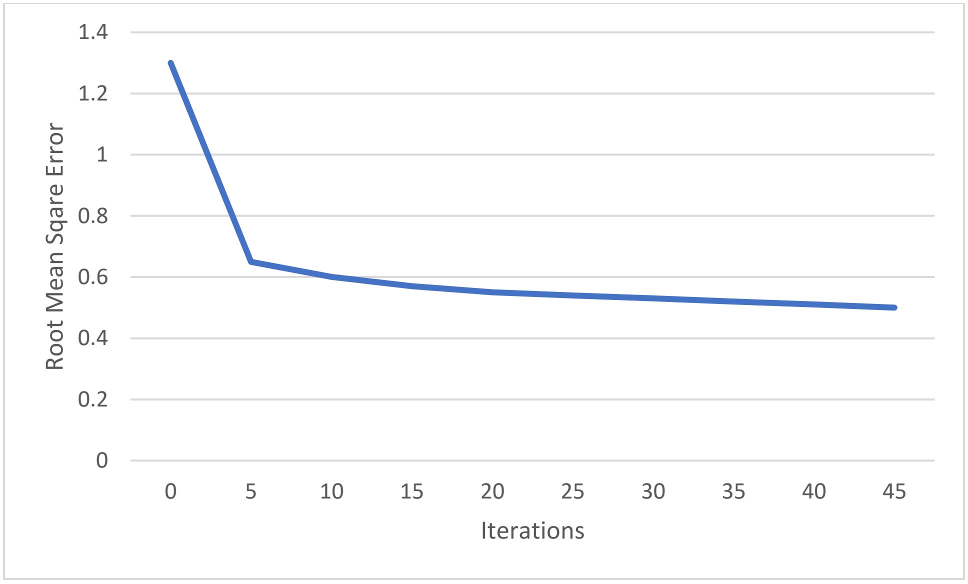
Graph(b): Winning Rate vs Number of rounds

The LUT obtained in the part-2 assignment of the course is used to train the neural net by varying the hyper-parameters. Firstly, the best number of hidden number of neurons is determined by changing the number of hidden neurons from 2-25 by keeping the value of momentum and learning rate constant 0.9 and 0.5 respectively. Then the RMS(root mean square) is computed for varying number of hidden neurons.



Graph(c): RMS error vs Number of Hidden Neurons

Based on the graph above, the RMS error with 18 number of hidden neurons is minimum. Then by keeping the number of hidden neurons as 18, momentum is varied to determine the best results by changing the values.



Graph(d): RMS vs Number of Iterations

The graph shows the root mean square error and the error decreases as the number of iterations increase (iterations* 10^3) with the 18 number of hidden neurons. The graph shows that the error is still decreasing with the increasing number of iterations.

Momentum: 0.9

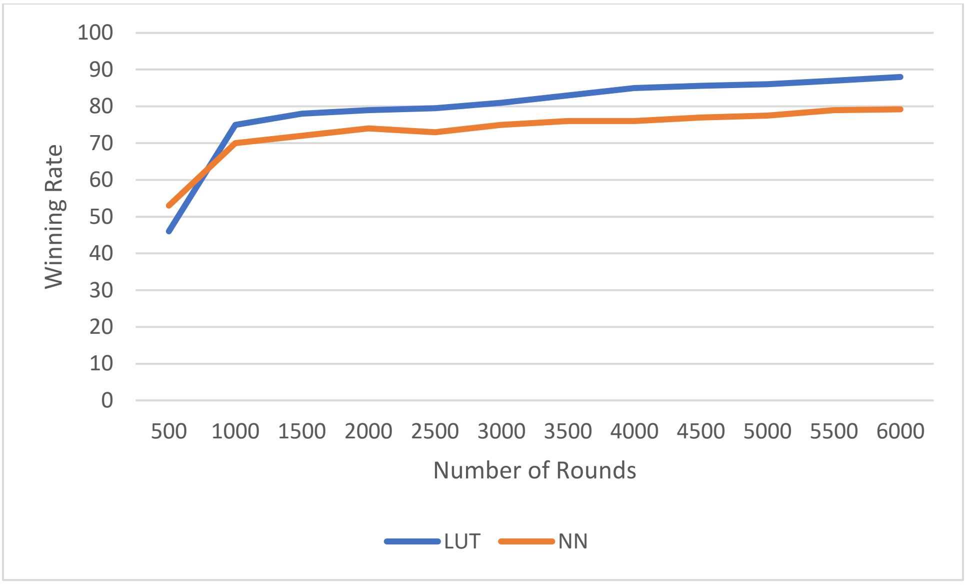
Learning Rate: 0.5

RMS Error= 0.541

4.3 Try mitigating or even removing any quantization or dimensionality reduction (henceforth referred to as state space reduction) that you may have used in part 2. A side-by-side comparison of the input representation used in Part 2 with that used by the neural net in Part 3 should be provided. (Provide an example of a sample input/output vector). Compare using graphs, the results of your robot from Part 2 (LUT with state space reduction) and your neural net based robot using less or no state space reduction. Show your results and offer an explanation.

INPUT-OUTPUT REPRESENTATION

Input	Using LUT	Non-Quantized	Using NN
X-Position	1,2,…….,8	0-800 Pixels	0,0.01,………,7.99,8
Y-Position	1,2,…..,6	0-600 Pixels	0,0.01,………,5.99,6
Distance between my robot and enemy robot	1,2,…….,4	0-1000 Pixels	0,0.01,0.02,…….9.99,10.00
Absolute bearing angle between my robot and enemy robot	1,2,3,4	0-360^{o}	0,0.01,………3.99,4.00
Actions	0,1,2,3,4,5	0,1,2,3,4,5	0,1,2,3,4,5



Graph(e): Winning rate of LUT vs NN

The graph shows that the NN takes longer time to learn than LUT.

4.4 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.

For any state-action pair, considering its energy, bearing, x-position, y-position, distance, actions, without the quantization the state space for one scan of tank can be as follows:

$$(energy \times heading \times x \times y \times velocity \times gun\ heading \times radar\ heading)$$

$$100 \times 360 \times 800 \times 600 \times 8 \times 360 \times 360 = 1.79 * 10^{16}$$

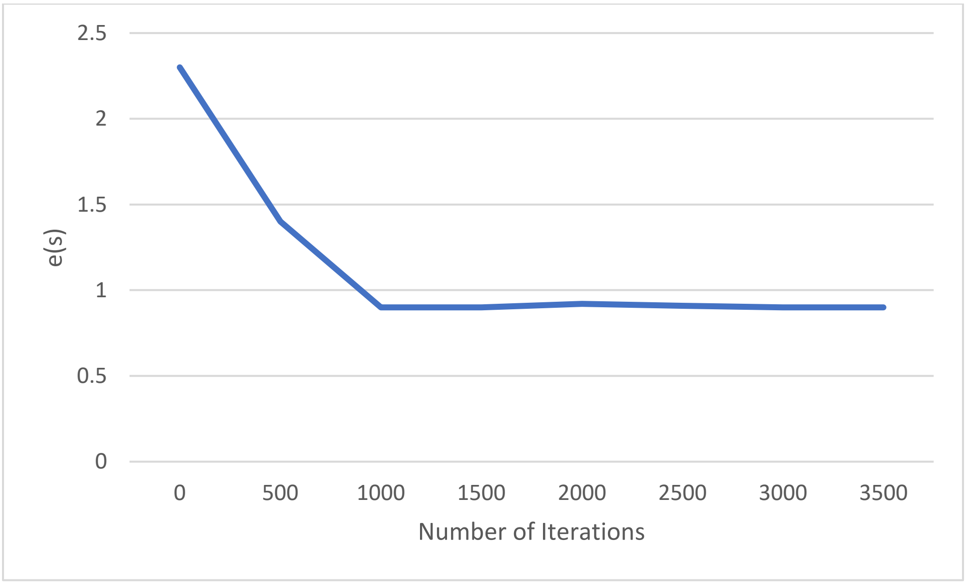
[Reference:[1]]

It can be very difficult to storage of these many states as this would generate a very large table and thus can result in longer time in learning. Whereas in the case of NN, the state action pair and the corresponding QValue are sent as the training data, instead of saving it in a large table. This is the reason why NN does not require state-space reduction as we only save the weights that approximate the value function.

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.

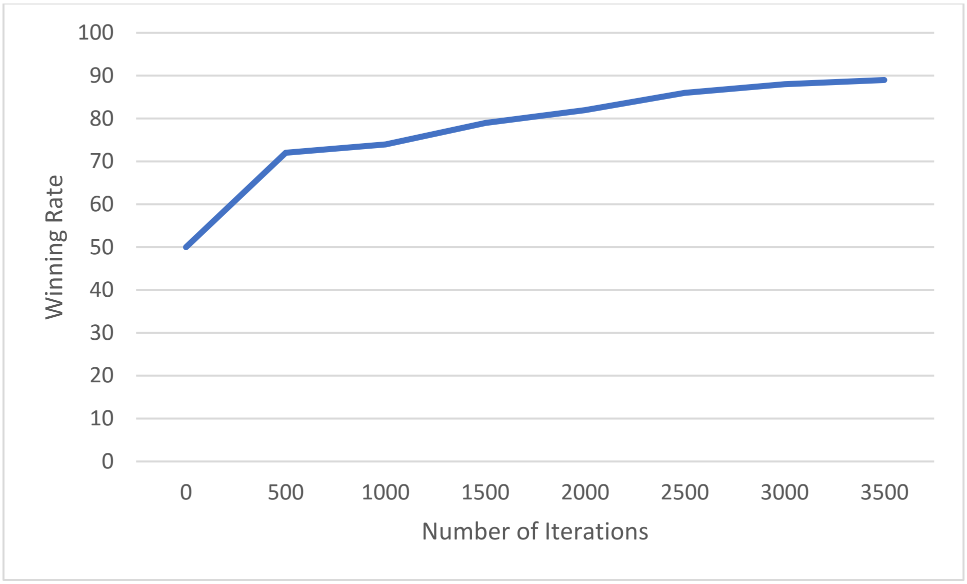
5.1 What was the best win rate observed for your tank? Describe clearly how your results were obtained? Measure and plot e(s) (compute as Q(s',a')-Q(s,a)) for some selected state-action pairs. Your answer should provide graphs to support your results. Remember here you are measuring the performance of your robot online. I.e. during battle.

The best win rate observed for my Reinforcement Learning robot with the Neural Net is 89% using functional approximation. The following graph illustrates that the e(s) decreases gradually with the increasing number of iterations. Initially, oscillations are noticed and as the oscillations reduce, the robot starts learning. As the number of iterations increases the learning of the Neural net becomes better.



Graph(f): e(s) vs Number of iterations

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



Graph(g): Winning Rate vs Number of Iterations

As the number of iterations increase, the winning rate increases asymptotically, and it takes longer time to converge. The best winning rate achieved is 89%.

5.3 *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.*

The Bellman equation is

V(st) = rt + γV(st+1) (1)

Where V(st) and V(st+1) are value functions at time t and time t+1

rt is the reward

γ is the discount factor

The Bellman e`quation for the optimal value function is written as:

V∗(st) = rt + γV∗(st+1) (2)

The value function can also be written with respect to the optimal value function and error as given below:

$V(st) = e(st) + V*(st)$ (3)

From (3)

$V(st+1) = e(st+1) + V*(st+1)$ (4)

Substituting V(st) from equation (3) and V(st+1) in equation (4) in equation (1)

$e(st) + V*(st) = rt + \gamma(e(st+1) + V*(st+1))$ (5)

$e(st) + V*(st) = rt + \gamma V*(st+1) + \gamma e(st+1)$ (6)

From equation (2)

$e(st) = \gamma e(st+1)$ (7)

which specifies that the errors of the successive states are related to each other the same way the value functions of the successive states are related to each other.

In the MDP, the terminal state T is observed as a result of the chain of states. For the terminal state, we must know the reward beforehand, so that there is no error resulting at the terminal state associated with the reward[2].

In robocode, for e(st)=0, we are propagating the future reward backwards, one state at a time and this is applied to the error as well which is also propagated backwards. Thus, e(st) will eventually become zero. This is how Bellman's value function converges to the stable state over time as dictated by optimal value function.

But when the Q-Function is approximated, backpropagation and the gradient descent is used to update the weights which allows the value of the input vector to reach the desired value. The gradient descent does not gaurantee the global minima, thus NN might settle at the local minima. Thus, e in the function approximation might not converge or might not become zero.

5.4 When using a neural network for supervised learning, performance of training is typically measured by computing a total error over the training set. When using the NN for online learning of the Q-function in robocode this is not possible since there is no a-priori training set to work with. Suggest how you might monitor learning performance of the neural net now. Hint: Readup on experience replay

By learning performance, we can figure out whether or not our Neural Net is learning. The learning performance in online training of the neural net can be measured in various ways:

When we compute the value of e(s)=Q(st+1)-Q(st), we can see if the Neural Net is learning or not. This is a good measure because we can come to know whether the output of the NN for input x converges or diverges. If it converges, the value of E(x) approaches zero over time else otherwise. Also, If there is decrease in the reward for any specific state-action pair then the QValue at t+1 should be less than QValue at t, then e(s) becomes negative. If the NN is showing some learning, then the value of e(s) vs number of rounds should ultimately approach zero from negative value. The above approach can be used to see whether the robot is learning or not. If there is no such trend, it means the robot is not showing any learning.

6) Overall Conclusions

6.1 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?

<u>Practical Issues Surrounding the Application of RL and BP</u>

There were certain issues faced during the application of RL and BP, as for the robot to perform better, a large number of state-action pairs were required, which had more error while approximating the QValues. To address this, number of hidden neurons would have to be increased which results in longer training time. Also, RL with the NN, takes a lot of time. During the implementation of RL, the robot took a long time to learn and this also required training the robot for a lot of iterations overnight.

Also, my robot is trained against TrackFire and performs quite well after learning. But if my robot is used against some other enemy robot, the performance is not that high. So, generalization is not there.

<u>Improving the Performance of the Robot</u>

The performance of the robot can be enhanced by changing various hyperparameters as we have already seen in the graph above. Also, the number of hidden neurons affect the way the robot performs as the more number of hidden neurons can be used for function approximation, which improved the learning of the robot. Also increasing the generalization of the NN would improve the accuracy of QValue approximation for any input values. Generalizing depends upon various factors like number of hidden neurons etc.

In the BP algorithm, a variable step size would be a better choice than a constant step size. Lastly, separating the NN for each action will also help in improving the performance.

<u>Addressing the Convergence Problems</u>

As mentioned above, the convergence problem might be addressed my using variable step size instead of constant step size in BP. And varying learning rate and momentum values, different initial weights might also prove effective to deal with convergence problems. Also, the activation function used for my robot is Bipolar, and other activation function might be better like RELU [2].

<u>Advice</u>

If I were to give any advice when applying RL with NN based function approximation, I would recommend training the robot using the training data from the LUT as the way we did for the project. As, the path for the NN using RL is better this way and gives better insight for an intermediate programmer to understand the NN step-by-step.

6.2 *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.*

The way the RL performs is through learning. Learning is required for a robot using a lot of training samples and gradually learns and this may take some time. So, this can be fatal for the patient. Though, RL can be used in simulators, which can be used to train the simulators. And based on the data collected over time, this data can be used to treat patients to provide anesthetic during surgery.

References

[1] S. Sarkaria, "Preprocessing_c : EECE592 Course Notes," October 2016.

[2] S. Sarkaria, "Reinforcement Learning Part 2: EECE592 Course Notes," October 2016.

Appendix

CommonInterface.java

```
import java.io.File;
import java.io.IOException;

public interface CommonInterface {

        public double outputFor(double [] X);

        /**
        * This method will tell the NN or the LUT the output
        * value that should be mapped to the given input vector. I.e.
        * the desired correct output value for an input.
        * @param X The input vector
        * @param argValue The new value to learn
        * @return The error in the output for that input vector
        */
        public double train(double [] X, double argValue);

        /**
        * A method to write either a LUT or weights of an neural net to a file.
        * @param argFile of type File.
        */
        public void save(File argFile);

        /**
        * Loads the LUT or neural net weights from file. The load must of course
        * have knowledge of how the data was written out by the save method.
        * You should raise an error in the case that an attempt is being
        * made to load data into an LUT or neural net whose structure does not match
        * the data in the file. (e.g. wrong number of hidden neurons).
        * @throws IOException
        */
        public void load(File argFileName) throws IOException;


        }
```

LUTInterface.java

```
public interface LUTInterface extends CommonInterface {

 /**
 * Constructor. (You will need to define one in your implementation)
 * @param argNumInputs The number of inputs in your input vector
 * @param argVariableFloor An array specifying the lowest value of each variable in the input
ctor.
 * @param argVariableCeiling An array specifying the highest value of each of the variables in
e input vector.
 * The order must match the order as referred to in argVariableFloor.
*
public LUT (
int argNumInputs,
int [] argVariableFloor,
int [] argVariableCeiling );
*/

/**
* Initialise the look up table to all zeros.
*/
public void initialiseLUT();

/**
* A helper method that translates a vector being used to index the look up table
* into an ordinal that can then be used to access the associated look up table element.
* @param X The state action vector used to index the LUT
* @return The index where this vector maps to
*/
public int indexFor(double [] X);


} // End of public interface LUT
```

MyNeuralNet.java

```
        import java.io.BufferedWriter;
        import java.io.File;
        import java.io.FileInputStream;
        import java.io.FileWriter;
        import java.io.IOException;
        import java.io.PrintStream;
        import java.util.Scanner;
        import robocode.RobocodeFileOutputStream;

        public class MyNeuralNet implements NeuralNetInterface {


                public static void main() {
                        System.out.println("numHidden: " + numHidden);
                        MyNeuralNet bp = new MyNeuralNet();
                        try {
                                bp.load(inputFile);
                        }
                        catch (IOException e) {
                                e.printStackTrace();
                        }
                        bp.initWeights();

                        do {
                                totalError = 0;
                                epoch++;
                                for (int i = 0; i < numVectors; i++) {
                                        bp.train(inputValues[i], targetValues[i]);
                                        totalError += getOneVectorError(tmpOut, targetValues[i]);
```

```
			}


			System.out.println("epoch: " + epoch + ", total error: " + totalError);
			if (epoch > 1000)
				break;
		} while (totalError > acceptableError /*xxx-->0*/);

	}

	private static double getOneVectorError(double tmpOut, double target) {
		double onceError = 0.5 * Math.pow((tmpOut - target), 2);
		return onceError;

	}

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

	@Override
	public double train(double[] X, double target) {
		calculateHidVal(X);
		calculateOutVal();

		calculateOutErr(target);
		calculateHidErr();


		updateWeightInpToHid(X);
		updateWeightHidToOut();
		return tmpOut;
	}

	private void calculateHidVal(double[] X) {
		for (int i = 0; i < numHidden; i++) {
			hiddenValues[i] = 0;
			for (int j = 0; j < numInputs; j++) {
				hiddenValues[i] += X[j] * weightInputToHidden[i][j];
			}
			hiddenValues[i] += weightInputToHidden[i][numInputs] * 1;

			hiddenValues[i] = sigmoid(hiddenValues[i]);
		}
	}

	private void calculateOutVal() {
		tmpOut = 0;
		for (int i = 0; i < numHidden; i++) {
			tmpOut += hiddenValues[i] * weightHiddenToOutput[i];
		}
		tmpOut += weightHiddenToOutput[numHidden] * 1;
		tmpOut = sigmoid(tmpOut);
	}
```

```
private void calculateOutErr(double target) {

        outputError = 0.5 * (1 +tmpOut) * (1 - tmpOut) * (target - tmpOut);


}

private void calculateHidErr() {

        for (int i = 0; i < numHidden; i++) {
        hiddenError[i] = 0.5 * (1 +hiddenValues[i]) * (1 - hiddenValues[i])
                        * outputError * weightHiddenToOutput[i];

        }
}

private void updateWeightInpToHid(double[] X) {

        for (int i = 0; i < numHidden; i++) {
        System.arraycopy(weightInputToHidden[i], 0, tempWeightInputToHidden[i], 0,
                        weightInputToHidden[i].length);
        }


        for (int i = 0; i < numHidden; i++) {
                for (int j = 0; j < numInputs; j++) {
                        weightInputToHidden[i][j] += learningRate
                                        * hiddenError[i] * X[j] + momentumTerm
                                        *  getPrevToHidDeltaWt(i, j);
                }
                weightInputToHidden[i][numInputs] += learningRate
                                * hiddenError[i] + momentumTerm
                                *  getPrevToHidDeltaWt(i, numInputs);
        }

        for (int i = 0; i < numHidden; i++) {
        System.arraycopy(tempWeightInputToHidden[i], 0, previousWeightInputToHidden[i], 0,
                        weightInputToHidden[i].length);
        }

}

private void updateWeightHidToOut() {
        System.arraycopy(weightHiddenToOutput, 0, tempWeightHiddenToOutput, 0,
                        weightHiddenToOutput.length);

        for (int i = 0; i < numHidden; i++) {
                weightHiddenToOutput[i] += learningRate * outputError
                                * hiddenValues[i] + momentumTerm
                                * getPrevToOutDeltaWt(i);
        }
        weightHiddenToOutput[numHidden] += learningRate * outputError * 1
                        + momentumTerm * getPrevToOutDeltaWt(numHidden);
        System.arraycopy(tempWeightHiddenToOutput, 0,
                        previousWeightHiddenToOutput, 0,
                        tempWeightHiddenToOutput.length);
```

```
	}

	private double getPrevToOutDeltaWt(int i) {
		if (previousWeightHiddenToOutput[i] != 0)
			return weightHiddenToOutput[i] - previousWeightHiddenToOutput[i];
		else
			return 0;
	}

	private double getPrevToHidDeltaWt(int i, int j) {
		if (previousWeightInputToHidden[i][j] != 0)
			return weightInputToHidden[i][j]
					- previousWeightInputToHidden[i][j];
		else
			return 0;
	}

	@Override
	public double sigmoid(double x) {

		return 2 / (1 + Math.pow(Math.E, -x)) - 1;
	}

	@Override
	public double customSigmoid(double x) {
		return (B - A) / (1 + Math.pow(Math.E, -x)) - A;
	}

	@Override
	public void initWeights() {
		for (int i = 0; i < numHidden; i++) {
			for (int j = 0; j < numInputs + 1; j++) {
				weightInputToHidden[i][j] = Math.random() - 0.5;

			}
		}

		for (int j = 0; j < numHidden + 1; j++) {
			weightHiddenToOutput[j] = Math.random() - 0.5;
		}
	}

	@Override
	public void zeroWeights() {
		for (int i = 0; i < numHidden; i++) {
			for (int j = 0; j < numInputs + 1; j++) {
				weightInputToHidden[i][j] = 0;
			}
		}

		for (int j = 0; j < numHidden + 1; j++) {
			weightHiddenToOutput[j] = 0;
		}
	}

	public void saveErrEpoch(File argFile, int epoch, double totalError) {
		BufferedWriter writer = null;
		try {
			writer = new BufferedWriter(new FileWriter(argFile, true));
			writer.write(epoch + "\t" + totalError + "\n");

		} catch (IOException e) {
		} finally {
			try {
				if (writer != null)
					writer.close();
			} catch (IOException e) {
			}
		}

	}
	@Override
	public void save(File argFile) {
		PrintStream saveFile = null;

		try {
			saveFile = new PrintStream(new RobocodeFileOutputStream(argFile));
		} catch (IOException e) {
			System.out
					.println("*** Could not create output stream for NN save
file.");
		}

		saveFile.println(numInputs);
		saveFile.println(numHidden);
		for (int i = 0; i < numHidden; i++) {
			for (int j = 0; j < numInputs; j++) {
				saveFile.println(weightInputToHidden[i][j]);
			}
			saveFile.println(weightInputToHidden[i][numInputs]);
		}

		for (int i = 0; i < numHidden; i++) {
			saveFile.println(weightHiddenToOutput[i]);
		}
		saveFile.println(weightHiddenToOutput[numHidden]);
		saveFile.close();
	}


	public void load(String inputFile) throws IOException {
		try {
			FileInputStream fileInputStream = new FileInputStream(inputFile);
			Scanner inputScanner = new Scanner(fileInputStream);

			for (int i = 0; i < numVectors; i++) {
				for (int j = 0; j < numInputs; j++) {
					inputValues[i][j] = inputScanner.nextDouble();
				}
				inputValues[i][numInputs] = bias;
				targetValues[i] = inputScanner.nextDouble();
			}
```

```
				} catch (IOException e) {
					System.out.print(e.getMessage());
				}
		}

		static int epoch = 1;
		static int numInputs = 4;
		static public int numHidden = 18;
		static int numOutputs = 1;
		static int numVectors = 42;
		static double learningRate = 0.5;
		static double momentumTerm = 0.9;
		static double bias = 1.0;
		double A;
		double B;
		static double totalError;
		static double acceptableError = 0.05;

		static double[][] weightInputToHidden = new double[numHidden][numInputs + 1];
		static double[][] previousWeightInputToHidden = new double[numHidden][numInputs + 1];


		static double[] weightHiddenToOutput = new double[numHidden + 1];
		static double[] previousWeightHiddenToOutput = new double[numHidden + 1];

		static double[][] tempWeightInputToHidden = new double[numHidden][numInputs + 1];
		static double[] tempWeightHiddenToOutput = new double[numHidden + 1];
		static double[] hiddenValues = new double[numHidden];
		static double[][] inputValues = new double[numVectors][numInputs + 1];
		static double[] targetValues = new double[numVectors];
		static double tmpOut = 0;
		static double outputError = 0;
		static double[] hiddenError = new double[numHidden];

		File paraFile = new File("C:/robocode/New folder/param.txt");
		static File ErrorEpochFile = new File("C:/robocode/New folder/Finalepoch.txt");
		static String inputFile = new String("C:/robocode/New folder/XOR.txt");
		static String LUT = new String ("C:/robocode/New folder/myLUT.txt");
		static String savetest = new String ("C:/robocode/New folder/saveFile.txt");
		@Override
		public void load(File argFileName) throws IOException {

		}

		public double getQValue(double[] sVector) {
			calculateHidVal(sVector);
			calculateOutVal();

			return tmpOut;
		}
	}
```

NeuralNetInterface.java

```
public interface NeuralNetInterface extends CommonInterface{
	final double bias = 1.0; // The input for each neurons bias weight

	/**
	* Constructor. (Cannot be declared in an interface, but your implementation will need one)
	* @param argNumInputs The number of inputs in your input vector
	* @param argNumHidden The number of hidden neurons in your hidden layer. Only a single hidden layer
is supported
	* @param argLearningRate The learning rate coefficient
	* @param argMomentumTerm The momentum coefficient
	 * @param argA Integer lower bound of sigmoid used by the output neuron only.
	 * @param arbB Integer upper bound of sigmoid used by the output neuron only.

	 public abstract NeuralNet (
	 int argNumInputs,
	 int argNumHidden,
	 double argLearningRate,
	 double argMomentumTerm,
	 double argA,
	 double argB );
	 */

	 /**
	 * Return a bipolar sigmoid of the input X
	 * @param x The input
	 * @return f(x) = 2 / (1+e(-x)) - 1
	 */
	 public double sigmoid(double x);

	 /**
	 * This method implements a general sigmoid with asymptotes bounded by (a,b)
	 * @param x The input
	 * @return f(x) = b_minus_a / (1 + e(-x)) - minus_a
	 */
	 public double customSigmoid(double x);

	 /**
	 * Initialize the weights to random values.
	 * For say 2 inputs, the input vector is [0] & [1]. We add [2] for the bias.
	 * Like wise for hidden units. For say 2 hidden units which are stored in an array.
	 * [0] & [1] are the hidden & [2] the bias.
	 * We also initialise the last weight change arrays. This is to implement the alpha term.
	 */
	 public void initWeights();

	 /**
	 * Initialize the weights to 0.
	 */
	 public void zeroWeights();


	 } // End of public interface NeuralNetInterface
```

NeuralTest.java

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class NeuralTest {
	static String numNeuronEpoch = new String ("C:/robocode/New folder/NeuronEpoch.txt");

	public static void main(String[] args) {
		MyNeuralNet NNTest = new MyNeuralNet();
		NNTest.main();
	}

	private static void saveConvergeEpoch(int n, int epoch) {
		BufferedWriter writer = null;
		try {
			writer = new BufferedWriter(new FileWriter(numNeuronEpoch, true));
			writer.write(n + "\t" + epoch + "\n");
		} catch (IOException e) {
			e.printStackTrace();
		} finally {
			try {
				if (writer != null)
					writer.close();
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
	}
}
```

RLTest.java

```
import java.awt.geom.Point2D;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

import robocode.AdvancedRobot;
import robocode.BulletHitBulletEvent;
import robocode.BulletHitEvent;
import robocode.BulletMissedEvent;
import robocode.DeathEvent;
import robocode.HitByBulletEvent;
import robocode.HitRobotEvent;
import robocode.HitWallEvent;
import robocode.ScannedRobotEvent;
import robocode.WinEvent;

public class RLTest extends AdvancedRobot implements LUTInterface {
	static File savetest = new File("C:/robocode/New folder/saveFile.txt");
	static File convergeCheck = new File("C:/robocode/New folder/convCheck.txt");
	static File rewardCheck = new File("C:/robocode/New folder/rewCheck.txt");
	static int battleCount = 1;
	public static double reward = 0;
```

```
        public static double finalReward = 0;
        public static final int numHeading = 4;
        public static final int numEnemyBearing = 4;
        public static final int numEnemyDistance = 2;
        public static final int numMyEnergy = 10;
        public static final int numEnemyEnergy = 10;
        public static int stateIndex[][][] = new int[numHeading][numEnemyBearing][numEnemyDistance];

        private static final int numState = numHeading * numEnemyBearing
                        * numEnemyDistance;
        private static final int numAction = 6;
        static double[][] qtable = new double[numState][numAction];
        static int[] actionIndex = new int[6];
        public static final int goAhead = 0;
        public static final int goBack = 1;
        public static final int goAheadTurnLeft = 2;
        public static final int goAheadTurnRight = 3;
        public static final int goBackTurnLeft = 4;
        public static final int goBackTurnRight = 5;
        public static final int numActions = 6;
        private boolean first = true;
        public static final double LearningRate = 0.5;
        public static final double DiscountRate = 0.9;
        public static final double ExplorationRate = 1;
        public static final double ExploitationRate = 0.5;
        private int lastStateIndex;
        private int lastAction;
        public static final double RobotMoveDistance = 300.0;
        public static final double RobotTurnDegree = 45.0;
        private double firePower = 1;

        String nameEnemy;
        public double bearingEnemy;
        public double headingEnemy;
        public double spotTimeEnemy;
        public double speedEnemy;
        public double XPositionEnemy;
        public double YPositionEnemy;
        public double distanceEnemy = 10000;
        public double constantHeadingEnemy;
        public double energyEnemy;

    public void run() {
                initialiseLUT();
                setAdjustGunForRobotTurn(true);
                setAdjustRadarForGunTurn(true);

                while (true) {
                        move();
                        firePower = 400 / distanceEnemy;
                        radarMove();
                        gunMove();
                        if (getGunHeat() == 0) {
                                setFire(firePower);
                        }
                        execute();
```

```
		}
	}

	private void gunMove() {
		long time;
		long nextTime;
		Point2D.Double p;
		p = new Point2D.Double(XPositionEnemy, YPositionEnemy);
		for (int i = 0; i < 20; i++) {
			nextTime = (int) Math
					.round((getrange(getX(), getY(), p.x, p.y) / (20 - (3 * firePower))));
			time = getTime() + nextTime - 10;
			p = futurePosition(time);
		}
		double gunOffset = getGunHeadingRadians()
				- (Math.PI / 2 - Math.atan2(p.y - getY(), p.x - getX()));
		setTurnGunLeftRadians(nomalizeDegree(gunOffset));
	}

	public Point2D.Double futurePosition(long futureTime) {
		double duration = futureTime - spotTimeEnemy;
		double futureX;
		double futureY;
		if (Math.abs(constantHeadingEnemy) > 0.000001) {
			double radius = speedEnemy / constantHeadingEnemy;
			double tothead = duration * constantHeadingEnemy;
			futureY = YPositionEnemy
					+ (Math.sin(headingEnemy + tothead) * radius)
					- (Math.sin(headingEnemy) * radius);
			futureX = XPositionEnemy + (Math.cos(headingEnemy) * radius)
					- (Math.cos(headingEnemy + tothead) * radius);
		} else {
			futureY = YPositionEnemy + Math.cos(headingEnemy) * speedEnemy
					* duration;
			futureX = XPositionEnemy + Math.sin(headingEnemy) * speedEnemy
					* duration;
		}
		return new Point2D.Double(futureX, futureY);
	}

	private double getrange(double x1, double y1, double x2, double y2) {
		double xo = x2 - x1;
		double yo = y2 - y1;
		double h = Math.sqrt(xo * xo + yo * yo);
		return h;
	}

	private void radarMove() {
		double turnDegree;
		if (getTime() - spotTimeEnemy > 4) {
			turnDegree = Math.PI * 4;
		} else {
			turnDegree = getRadarHeadingRadians()
					- (Math.PI / 2 - Math.atan2(YPositionEnemy - getY(),
							XPositionEnemy - getX()));
```

```
                turnDegree = nomalizeDegree(turnDegree);
                if (turnDegree < 0)
                    turnDegree -= Math.PI / 10;
                else
                    turnDegree += Math.PI / 10;
            }

            setTurnRadarLeftRadians(turnDegree);
        }

        private double nomalizeDegree(double ang) {
            if (ang > Math.PI)
                ang -= 2 * Math.PI;
            if (ang < -Math.PI)
                ang += 2 * Math.PI;
            return ang;
        }

        private void move() {
            double tmpHeading = getHeading();
            tmpHeading += 45;
            if (tmpHeading > 360)
                tmpHeading -= 360;
            int heading = (int) (tmpHeading / 90);
            if (heading > numHeading - 1)
                heading = numHeading - 1;
            if (bearingEnemy < 0)
                bearingEnemy += 360;
            double tmpBearingEnemy = bearingEnemy + 45;
            if (tmpBearingEnemy > 360)
                tmpBearingEnemy -= 360;
            int enemyBearing = (int) (tmpBearingEnemy / 90);
            if (enemyBearing > numEnemyBearing - 1)
                enemyBearing = numEnemyBearing - 1;
            int enemyDistance = (int) (distanceEnemy / 200);
            if (enemyDistance > numEnemyDistance - 1)
                enemyDistance = numEnemyDistance - 1;
            int myEnegy = (int) (getEnergy() / 10);
            if (myEnegy > numMyEnergy - 1)
                myEnegy = numMyEnergy - 1;

            int enemyEnergy = (int) (energyEnemy / 10);
            if (enemyEnergy > numEnemyEnergy - 1)
                enemyEnergy = numEnemyEnergy - 1;

            double[] currentState = { heading, enemyBearing, enemyDistance };
            double crrentAction = train(currentState, reward);
            int intAction = (int) crrentAction;

            reward = 0;
            executeAction(intAction);
        }
        private void executeAction(int action) {
            switch (action) {
            case goAhead:
                setAhead(RobotMoveDistance);
                break;

            case goBack:
                setBack(RobotMoveDistance);
                break;

            case goAheadTurnLeft:
                setAhead(RobotMoveDistance);
                setTurnLeft(goAhead);
                break;

            case goAheadTurnRight:
                setAhead(RobotMoveDistance);
                setTurnRight(RobotTurnDegree);
                break;

            case goBackTurnLeft:
                setBack(RobotMoveDistance);
                setTurnRight(RobotTurnDegree);
                break;

            case goBackTurnRight:
                setBack(RobotMoveDistance);
                setTurnRight(RobotTurnDegree);
                break;
            }
    }

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

    @Override
    public double train(double[] X, double reward) {
            int stateIndexNow = indexFor(X);
            double maxQ = maxQValueFor(stateIndexNow);
            int actionNow = selectAction(stateIndexNow);
            double diffQValue = 0;

            System.out.println("Reward: " + reward);
            if (first)
                first = false;
            else {


                // for Q-Learning
                double oldQValue = qtable[lastStateIndex][lastAction];
                double newQValue = (1 - LearningRate) * oldQValue + LearningRate
                                * (reward + DiscountRate * maxQ);
                diffQValue = newQValue - oldQValue;
                qtable[lastStateIndex][lastAction] = newQValue;
            }
            lastStateIndex = stateIndexNow;
            lastAction = actionNow;
            if (stateIndexNow == 1199 && actionNow == 0)
				saveDiffQValue(stateIndexNow, actionNow, diffQValue, convergeCheck);


			return actionNow;
		}

		private int selectAction(int stateIndexNow) {
			int action = actionMaxQValue(stateIndexNow);
			if (Math.random() > ExplorationRate) {
				return action;
			} else {
				int tmpAction = (int)(Math.random() * 10) % 6;
				return tmpAction;
			}
		}

		private void saveDiffQValue(int stateIndexNow, double action,
				double diffQValue, File convergeCheck) {
			BufferedWriter writer = null;
			try {
				writer = new BufferedWriter(new FileWriter(convergeCheck, true));
				writer.write(battleCount + "\t" + stateIndexNow + "\t" + action
						+ "\t" + diffQValue + "\n");

			} catch (IOException e) {
			} finally {
				try {
					if (writer != null)
						writer.close();
				} catch (IOException e) {
				}
			}
		}

		@Override
		public void save(File argFile) {
			BufferedWriter writer = null;
			try {
				writer = new BufferedWriter(new FileWriter(argFile));
				for (int i = 0; i < numState; i++)
					for (int j = 0; j < numAction; j++) {
						int saveDistance = i % 2;
						int saveHeading = i / 8;
						int saveBearing = (i - saveDistance * 2 - saveHeading * 4) / 4;

						if (qtable[i][j] != 0)
						writer.write(/*i + "\t" +*/ saveHeading + "\t" + saveBearing + "\t" +
saveDistance + "\t" + j + "\t" + qtable[i][j]
								+ "\n");
					}

			} catch (IOException e) {
			} finally {
				try {
					if (writer != null)
						writer.close();
				} catch (IOException e) {
				}
		}

	}

	@Override
	public void load(File inputFile) throws IOException {
		try {
			FileInputStream fileInputStream = new FileInputStream(inputFile);
			Scanner inputScanner = new Scanner(fileInputStream);

			for (int i = 0; i < numState; i++)
				for (int j = 0; j < numAction; j++) {
					double statee = inputScanner.nextDouble();
					double actionn = inputScanner.nextDouble();
					if ((int) statee == i && (int) actionn == j)
						qtable[i][j] = inputScanner.nextDouble();
				}

		} catch (IOException e) {
			System.out.print(e.getMessage());
		}
	}

	@Override
	public void initialiseLUT() {
		int count = 0;
		for (int a = 0; a < numHeading; a++)
			for (int b = 0; b < numEnemyBearing; b++)
				for (int c = 0; c < numEnemyDistance; c++)
					stateIndex[a][b][c] = count++;

		for (int x = 0; x < numState; x++)
			for (int y = 0; y < numAction; y++)
				qtable[x][y] = 0;
	}

	@Override
	public int indexFor(double[] X) {
		int index = ((int) X[0] + 1) * 4 + ((int) X[1] + 1) * 4 + ((int) X[2] + 1) * 2 -1;
		return index;
	}

	public double maxQValueFor(int index) {
		double max = 0;
		for (int i = 0; i < numAction; i++) {
			if (qtable[index][i] > max)
				max = qtable[index][i];
		}
		return max;
	}

	public int actionMaxQValue(int stateIndexNow) {
		double max = 0;
		int action = 0;
		MyNeuralNet nn = new MyNeuralNet();
		for (int i = 0; i < numAction; i++) {

			double saveDistance = stateIndexNow % 2;
			double saveHeading = stateIndexNow / 8;
			double saveBearing = (stateIndexNow - saveDistance * 2 - saveHeading * 4) / 4;
			double[] sVector = new double[]{saveHeading, saveBearing, saveDistance, i};
			double tmpMax = nn.getQValue(sVector);
			if (tmpMax > max) {
				max = tmpMax;
				action = i;
			}

		}
		return action;
	}
	public void onBulletHit(BulletHitEvent e) {
		double change = e.getBullet().getPower() * 9;
		reward += change;
		finalReward += change;
	}

	public void onBulletHitBullet(BulletHitBulletEvent e) {

	}

	public void onHitByBullet(HitByBulletEvent e) {

		double change = -5 * e.getBullet().getPower();
		reward += change;
		finalReward += change;
	}

	public void onBulletMissed(BulletMissedEvent e) {
		double change = -e.getBullet().getPower();
		reward += change;
		finalReward += change;
	}

	public void onHitRobot(HitRobotEvent e) {

		double change = -6.0;
		reward += change;
		finalReward += change;
	}

	public void onHitWall(HitWallEvent e) {

		double change = -(Math.abs(getVelocity()) * 0.5 - 1);
		reward += change;
		finalReward += change;
	}

}
```