PART 3 - REINFORCEMENT LEARNING WITH BACKPROPAGATION:

Partner with Jay Fu, ID: 54675310

In the final part of this coursework, we are required to implement our reinforcement learning mechanism with Backpropagation which has been initially completed in the first part. Therefore, we need to replace the look-up-table with neural network.

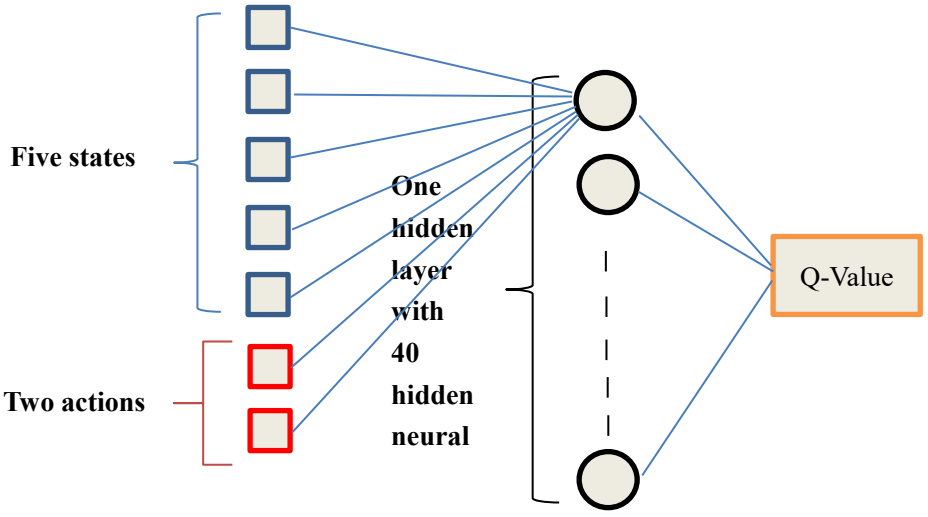
4(a) *Describe the architecture of your neural network and how the training set captured from Part 2 was used to "offline" train it mentioning any input representations that you may have considered. Note that you have 3 different options for the high level architecture. A net with a single Q output,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

Answer:

As instructed, there are three different options for high-level architectures, which are respectively are *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.*

The one used for this assignment is *a net with a Q output for each action*.

Here is the graph showing the basic structure of my neural network:



Graph.1 Basic architecture of Neural Network

States / Actions	Heading (4 sub-states)	Distance (4 sub-states)	Bearing (4 sub-states)	X-Position (4 sub-states)	Y-Position (4 sub-states)
Fire	Value1	Value2	Value3	Value4	Value5
Front-Left	……	……	……	……	……
Front-Right	……	……	……	……	……
Back-Left	……	……	……	……	……
Back-Right	……	……	……	……	……

Table.1 Actions and States (LUT)

As mentioned in the part2, there are five different type of states and five distinct types of actions were used in my reinforcement learning mechanism. However, in this assignment, those five single actions have been combined into two complex actions which are used as two inputs for NN. Back in the LUT, the general idea is to identify the action which yields the highest Q-value based on the current state and execute this action. In neural network, we are still going to choose one complex action based on the current state, which means there are ten inputs and one Q output for a single NN.

As we can see from the Graph.1, there are seven inputs in the neural network and five of them represent the states and the other five shows two complex actions. It is worthwhile to mention that one-hot encoding is applied in these actions in order to give one Q-value for one action. Only one action could be non-zero while the other one has to maintain zero. Therefore, the high-level idea is *a net with a Q output for each action.*

In order to use the static LUT to determine our appropriate hyperparameter, we firstly trained the robot with LUT to reach a high winning rate. With that LUT, we applied it into the neural network to find best Q-value by tuning the hyperparameter.

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

Answer:

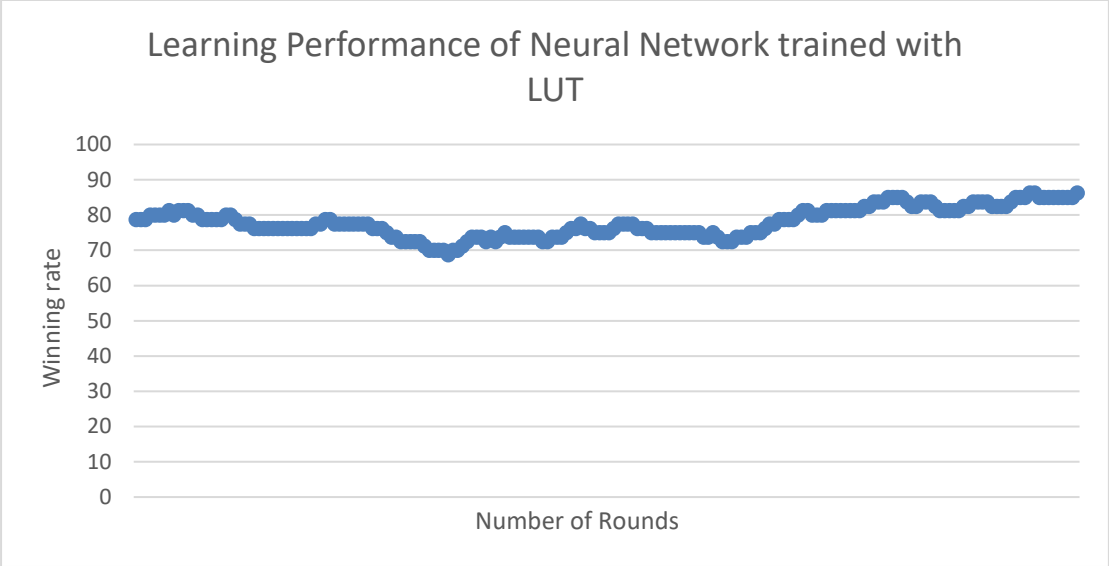
In this assignment, we have changed the activation function to the linear one rather than original sigmoid function. Therefore, the range of the output are no longer from -1 to 1 and the value of RMS would be larger than the previous one.

The results of my training neural network using the LUT is presented as the RMS value for different hyperparameter sets. Whenever the difference between the current epoch and previous one is smaller than 0.000005, we will say the learning process reaches the end, which means the results have converged already. Here is the table.2 showing all the hyperparameter sets that I used to evaluate results and RMS.

Number of hidden neural	Learning Rate	Momentum Term	Number of Epochs	Converged RMS
20	0.025	0.05	72	3.206853842372339
20	0.025	0.045	59	3.1964222008819556
20	0.025	0.055	74	3.214682089892597
20	0.025	0.06	89	3.2545535508844865
40	0.025	0.05	281	3.711153860775943
40	0.030	0.05	259	4.117525578660254
40	0.040	0.05	79	5.585588639344579
40	0.015	0.05	126	3.191757789439524
40	0.030	0.07	110	4.145222141168147
40	0.030	0.06	235	4.208977904960589
40	0.045	0.1	255	8.386637974643156
60	0.025	0.05	294	5.0274383154927165

Table.2 RMS representation of different sets of hyper-parameters

From the all of these sets of hyperparameters that I have tried, the one with best RMS value is [Number of hidden layer = 20; Learning rate = 0.025; Momentum Term = 0.045]. The RMS for it is 3.19. Here is the Graph.3 which shows the performance of NN with the best parameter.



Graph.2 Performance of NN trained by LUT

From the graph.2, we can clearly see that the winning rate is starting at a high point because of the training of LUT. Actually, it has reached its convergent field by tuning those hyperparameters based on RMS already.

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

Answer:

In this section, we are required to compare the performance of reinforcement learning with LUT with reinforcement learning with neural network. The former used the dimensionality reduction and the latter didn't or used less dimensionality reduction. Here is the example of a sample input/output vector in the LUT:

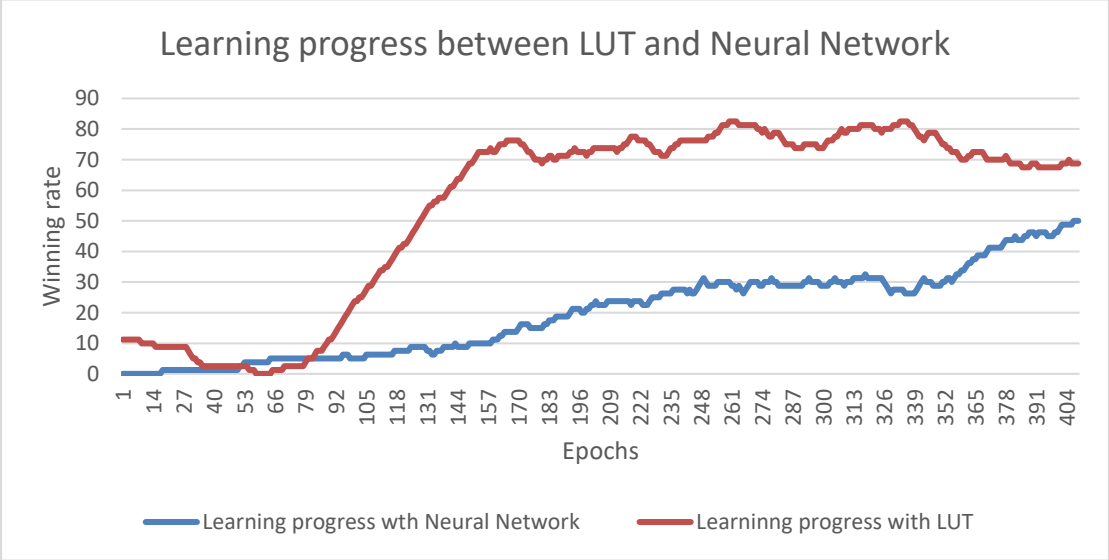
[0 1 3 5 6]

Here is the example of a sample input/output vector in the NN:

[0.1536 0.9532 1.2355 3.6642 0.8847]

In LUT, we have to discretize every action to decease its dimensionality. For example, the angle it turns only have four options therefore its dimensionality is really low.

However, in NN, angles could be successive therefore it can choose infinite number of value to turn. Hence dimension could be much higher than LUT.



Graph.3 Performance comparison between NN and LUT

From the above graph.3, apparently, the learning progress with NN converges more faster than the one with LUT although there is no dimensionality reduction.

In theory, it is clear that it is not that necessary to use dimensionality reduction for the neural network. Without it, it can also reach the same or even better performance for the game. NN's capability for handling or solving complex data is much more greater than simply updating LUT because of hidden layer and perceptron. For LUT, the model would like to become overfitting with too much input data without dimensionality reduction. However, for NN, layers have certain level of capability of

solving complex data.

What's more, the generalization capability if another crucial characteristic for the NN, which is also the reason why it would also give us a better results compared with the LUT.

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

Answer:

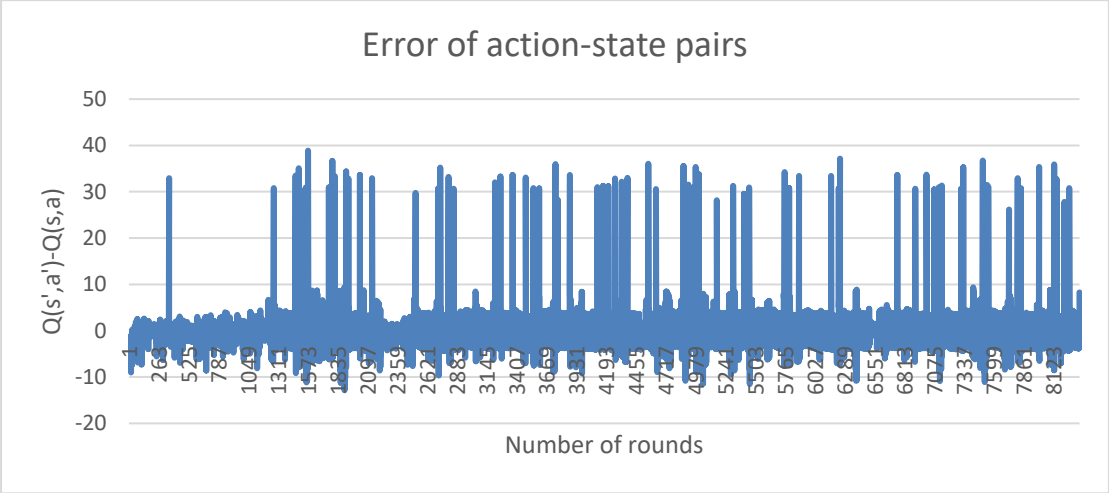
For the Look-Up-Table, it just simply store the Q-value with the given state and action, which means the table has to store all the Q-values for each different state-action pair. In the Robocode, there are dozens of choices of states and actions can be chosen as our own approach. Therefore, the number of state-action pairs has a high likelihood to become an enormous number, which could give a heavy load for the reinforcement learning mechanism while performing. Besides, the learning speed as well as the time consuming will be a big problem with huge LUT.

In terms of Neural Networks, it doesn't necessarily need to store all the Q-value of state-action pairs like the LUT. Therefore, the memory needed to implement this assignment is way more less and the efficiency would definitely a lot higher than before. This crucial advantage will be become more obvious while the dimensionality is getting higher. In the fact, the key thing for the neural network are weights among nodes and the suitable weights should be stored.

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

Answer:

From the graph.3, we can see the best winning rate my tank is approximately 83%. Base on the basic neural network structure of my learning model in 4(a). I firstly tried the NN trained by LUT to tune hyperparameters in order to find the weights.



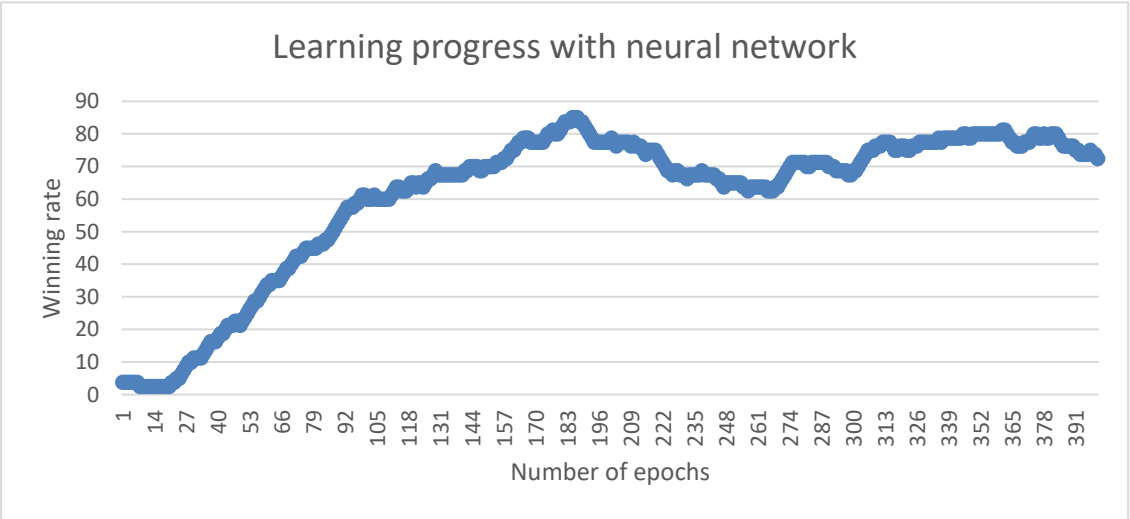
Graph.4 Error of action-state pairs

Indeed, there are some outliers in this graph. I still think these outliers won't affect the general performance of the learning process. The majority of the e(s) are still quite stable and nearly zero, which means these errors actually won't have a great influence on my NN's performance.

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

Answer:

Here is the graph.5 which shows us the win rate against number of battles.



Graph.5 Learning progress with neural network

From the Graph.5, we can see the win rate does improve asymptotically.

At the beginning, the winning rate nearly starts at zero and it ultimately reach to 85%. Although the convergent value is still vibrating a small range in the end, it is still clear that win rate is vibrating around a stable line, which means it improves asymptotically.

5(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.*

Answer:

Here is the Bellman equation:

$$\mathrm{V}(S_t) = r_t + \gamma * \mathrm{V}(S_{t+1})$$

From the lecture notes, $\mathrm{V}(\mathrm{S_t})$ represents the current state and the $\mathrm{V}(\mathrm{S_{t+1}})$ represents the successor state, r is the immediate reward, the discount factor applied to future rewards is γ. And the equation is recursive in nature and suggests that the values of successive states are related [1].

Now we use $\mathrm{V}^*(S_t)$ denotes the optimal value of current state and substitute $\mathrm{V}(\mathrm{S_t})$ by $\mathrm{V}^*(S_t)$.

$$\mathrm{V}^*(S_t) = r_t + \gamma * \mathrm{V}^*(S_{t+1})$$

$\mathrm{e}(S_t)$ is the error between original value and optimal value, which is

$$\mathrm{e}(S_t) = \mathrm{V}^*(S_t) - \mathrm{V}(S_t)$$

Therefore,

$$\mathrm{e}(S_t) = r_t + \gamma \mathrm{V}(S_{t+1}) - \mathrm{V}^*(S_t)$$

Eventually, we get

$$\mathrm{e}(S_t) = \gamma \mathrm{e}(S_{t+1})$$

In terms of the TD learning, the error would converge to zero in the end. Hence the convergence for TD learning algorithm is guaranteed to converge.

However, in the case of neural network with using approximation, the total error is the combination of $e(S_t)$ and e_{approx}, the latter one is generated by NN.

Therefore, convergence is no longer guaranteed when using approximation.

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

Answer:

There are serval ways to monitor learning performance of the neural net. As the hint suggested, the utilization of experience replay is a wonderful idea.

In my opinion, the key idea of experience replay is that use a memory to contain the pervious information and let it to guide or influence on the next move. In this case, we could use momentum which is a typical presentation of experience replay to monitor it. The converge efficiency greatly depend on the choice of step size. What's more, the step size could be adjusted by the momentum which is a utilization of experience replay. We could monitor the step size and momentum to the learning process.

Another instinct way of monitoring the learning performance is directly measure the winning rate of robot. Same as before, the increasing winning rate the ultimate goad what we will always keep an eye on. The final winning rate is supposed to converge to a high and stable value in the end. Hence, the process of increases of the winning rate could intuitively show us the learning process.

6(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?*

Answer:

i. *Practical issues surrounding the application of RL & BP to my problem*

- Modification of reward is the critical part to keep the reinforcement learning working properly. For instance, we set -50 as a failure terminal reward for the robot and 100 as winning terminal rewards, which gave us a really bad performance. It was consistently vibrating and converging slowly. However, the value of failure terminal reward had been adjusted to -10, the performance of robot improved a lot. Not only for the terminal rewards but also for the immediate rewards are needed to be considered as well.
- In terms of tuning the hyperparameter of the neural network, somehow, we cannot always use the same LUT in previous battle to determine hyperparameters no matter how well it performed in the last part. Because I can hardly find a set of universal hyperparameter for all the enemies. It just like “no free lunch” theory in machine learning field, you can never find the best model for all cases.

ii. *Methods of improving the performance of my robot*

- We can add more layers and hidden neural for NN to improve performance. Although it would be hard to see overfitting only for several layers we still need to bear that in mind.
- Adjustment of rewards sometimes will give you an outstanding performance.
- A comprehensive LUT would be helpful as a guidance to tune the hyperparameters for NN. We can use multiple enemies to formulate the

final LUT.

- Dimensionality reduction and normalization sometimes should be done before, which can lead to an more efficient convergence process.

iii. How would you suggest convergence problems be addressed

- Avoiding overfitting when apply the NN into the reinforcement learning.
- Using dimensionality reduction and normalization when using LUT.
- Set step-size and momentum ideally. Small momentum would not have a great effect on the step size and convergence rate while too large one would definitely cause non-convergence.

iv. What advice would you give when applying RL with neural network based function approximation to other practical application?

Some tiny and inevitable approximation errors could be generated because of approximation functions. Therefore, the choice of states would have a considerable influence on the performance of NN. Therefore, the relationship among chosen states is looser, the effect of function approximation would be smaller. For example, distance and bearing almost have nothing common in relationship while distance and X/Y position have more relationship. Whenever we choose states, we should try our best to choose independent states with function approximation.

What's more, hyperparameter is also should be tuned correctly before embedded NN into the reinforcement learning. Based on the given instruction, we are suggested to use static LUT to train NN and obtain appropriate hyperparameter like learning rate, exploration rate and so on. Hence, tuning suitable hyperparameter should be considered when applying RL with neural network based function approximation to other practical application.

6(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.*

Answer:

i. Concerns

In this scenario, the patient who is going to use anesthetic is under going surgery. Firstly, the accuracy of the type of anesthetic is the first priority, which has relationship to do with the life of patient. To ensure its accuracy, a huge amount of data should be fed into the database to choose the hyperparameter for 'offline training process'. With a reliable database, we can say the accuracy of this controller would be very high, even nearly 100%. However, the criteria of the system should be very high because its relevant to the medical issue.

Secondly, time is also very critical under this circumstance. We can imagine every second for a patient under going surgery is crucial. If we apply a deep neural network, how can we ensure there will never be a time delay for patients.

Thirdly, the limitation of the database will restrict the output of neural network.

If the number of the typical anesthetic is not obvious enough, there is a high probability that the control system will neglect this sample even the patient need this type.

ii. Potential Variation

There two potential variations that I think might alleviate those concerns.

Firstly, we could use the ensemble method in machine learning to enhance the accuracy of this control model. For example, we randomly divide the database into several part and use those divided section as input to generate the output. And then use averaging, bagging or boosting to greatly enhance the accuracy.

Secondly, we could apply thought of feature selection in terms of specific patient because each patient would probably has his own unique symptom. The output of learning algorithm should be more accurate with these customized features.

Reference

[1] S. Sarkaria, “Backpropagation,” *Architectures for Learning System*, pp. 6–6, Sep-2018.

Appendix

1. *States.java*

```
package JBot_NN;

public class States {
 public static int numHeading = 4;
 public static int numTargetDistance = 4;
 public static int numTargetBearing = 4;
 public static int numXPosition = 8;
 public static int numYPosition = 6;

 public static final int[][][][][] statesMap = new int[numHeading][numTargetDistance][numTargetBearing][numXPosition][numYPosition];
 public static int numStates = 0;
 public static void init(int[] f, int[] c) {
 //  System.out.println("Im init-ing States!!!");
  if (f.length != 5 || c.length != 5) {
   System.out.println("Wrong number of state parameters!!!!!");
   return;
  }
  numHeading = c[0] - f[0] + 1;
  numTargetDistance = c[1] - f[1] + 1;
  numTargetBearing = c[2] - f[2] + 1;
  numXPosition = c[3] - f[3] + 1;
  numYPosition = c[4] - f[4] + 1;
  int count = 0;
  for (int i = 0; i < numHeading; i++)
   for (int j = 0; j < numTargetDistance; j++)
    for (int k = 0; k < numTargetBearing; k++)
     for (int l = 0; l < numXPosition; l++)
      for (int m = 0; m < numYPosition; m++)
       statesMap[i][j][k][l][m] = count++;
  numStates = count;
 }

 public static double getHeading(double arg){
  // 0 ~ 4
  return arg / Math.PI * 2 / 4;
 }

 public static double getTargetDistance(double arg){
  // 0 ~ 10
```

```
        if (arg > 400) return 1;
        return arg / 400;
    }

    public static double getTargetBearing(double arg){
        // 0 ~ 4
        return (arg + 180) / 90 / 4;
    }

    public static double getXPosition(double arg) {
        // 0 ~ 8
        return arg / 100 / 8;
    }

    public static double getYPosition(double arg) {
        // 0 ~ 6
        return arg / 100 / 6;
    }

//    public static int getHeading(double arg){
//        //4
//        double unit = 2 * Math.PI / numHeading;
//        return (int)Math.floor(arg / unit);
//    }
//
//    public static int getTargetDistance(double arg){
//        //4 close, near, far, really far
//        int temp=(int)(arg/100);
//        if(temp > numTargetDistance - 1) temp = numTargetDistance - 1;
//        return temp;
//    }
//
//    public static int getTargetBearing(double arg){
//    //4
//        double unit = 360.0 / numTargetBearing;
//        return (int)Math.floor((arg + 180) / unit);
//    }
//
//    public static int getXPosition(double arg) {
//        double unit = 800 / numXPosition;
//        return (int)Math.floor(arg / unit);
//    }
//
//    public static int getYPosition(double arg) {
```

```
        //          double unit = 600 / numYPosition;
        //          return (int)Math.floor(arg / unit);
        //    }
    }
```

2. NeuralNetInterface.java

```
        package JBot_NN;

        /**
        * @date 20 June 2012
        * @author sarbjit
        *
        */
        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 initializeWeights();

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


} // End of public interface NeuralNetInterface
```

3. NeuralNet.java

```
    package JBot_NN;
    import java.lang.Math;

    import robocode.RobocodeFileOutputStream;

    import java.io.BufferedReader;
    import java.io.File;
    import java.io.FileReader;
    import java.io.IOException;
    import java.io.PrintStream;

    public class NeuralNet implements NeuralNetInterface
    {
     // ----------------------hyper-parameters----------------------
```

```
    private int argNumInputs;
    private int argNumHidden;
    private double argLearningRate;
    private double argMomentumTerm;
    private int argA;
    private int argB;
public static final double learningRate = 0.3;
public static final double discountFactor = 0.9;

    // ----------------------parameters----------------------
    private double[][] layer1;
    private double[] layer2;
    private double[] hiddenBias;
    private double outputBias;

    // ----------------------Global Variables----------------------
    private double[][] delta1;
    private double[] delta2;
    private double[] delataHiddenBias;
    private double deltaOutputBias;

    private double[] hiddenOutput;
    private double[] hiddenSum;
    private double outputSum;

    public NeuralNet(int argNumInputs,
                     int argNumHidden,
                     double argLearningRate,
                     double argMomentumTerm,
                     int argA,
                     int argB)
    {
        this.argNumInputs = argNumInputs;
        this.argNumHidden = argNumHidden;
        this.argLearningRate = argLearningRate;
        this.argMomentumTerm = argMomentumTerm;
        this.argA = argA;
        this.argB = argB;

        this.layer1 = new double[argNumInputs][argNumHidden];
        this.layer2 = new double[argNumHidden];
        this.hiddenBias = new double[argNumHidden];
        this.outputBias = 0;
        this.delta1 = new double[argNumInputs][argNumHidden];
```

```
            this.delta2 = new double[argNumHidden];
            this.delataHiddenBias = new double[argNumHidden];
            this.deltaOutputBias = 0;
            this.hiddenOutput = new double[argNumHidden];
            this.hiddenSum = new double[argNumHidden];
            this.outputSum = 0;

            this.initializeWeights();
        }

        public double sigmoid(double x)
        {
            return 2.0 / (1 + Math.exp(-x)) - 1;
        }

        public double customSigmoid(double x)
        {
            return (this.argB - this.argA) / (1 + Math.exp(-x)) + this.argA;
        }

        public double sigmoidDerivative(double x)
        {
            return 2.0 * Math.exp(-x) / ((1 + Math.exp(-x)) * (1 + Math.exp(-x)));
        }

        public double customSigmoidDerivative(double x)
        {
            return (this.argB - this.argA) * Math.exp(-x) / ((1 + Math.exp(-x)) * (1
+ Math.exp(-x)));
        }

        public void initializeWeights()
        {
            for (int j = 0; j < this.argNumHidden; j++)
            {
                this.layer2[j] = Math.random() - 0.5;
                this.delta2[j] = 0.0;
                this.hiddenBias[j] = Math.random() - 0.5;
                this.delataHiddenBias[j] = 0.0;
                this.hiddenOutput[j] = 0.0;
                this.hiddenSum[j] = 0.0;
                for (int i = 0; i < this.argNumInputs; i++)
                {
                    this.layer1[i][j] = Math.random() - 0.5;
```

```
                this.delta1[i][j] = 0.0;
            }
        }
        this.outputBias = Math.random() - 0.5;
        this.outputSum = 0;
        this.deltaOutputBias = 0;
    }

    public void zeroWeights()
    {
        for (int j = 0; j < this.argNumHidden; j++)
        {
            layer2[j] = 0.0;
            delta2[j] = 0.0;
            this.hiddenBias[j] = 0.0;
            this.delataHiddenBias[j] = 0.0;
            for (int i = 0; i < this.argNumInputs; i++)
            {
                layer1[i][j] = 0.0;
                delta1[i][j] = 0.0;
            }
        }
        this.outputBias = 0;
        this.deltaOutputBias = 0;
    }

    public void testWeights()   // 0.1
    {
        for (int j = 0; j < this.argNumHidden; j++)
        {
            layer2[j] = 0.1;
            delta2[j] = 0.0;
            this.hiddenBias[j] = 0.1;
            this.delataHiddenBias[j] = 0.0;
            for (int i = 0; i < this.argNumInputs; i++)
            {
                layer1[i][j] = 0.1;
                delta1[i][j] = 0.0;
            }
        }
        this.outputBias = 0.1;
        this.deltaOutputBias = 0.0;
    }
```

```
        private void hiddenOutputFor(double [] X)
        {
            for (int j = 0; j < this.argNumHidden; j++)
            {
                this.hiddenSum[j] = 0;
                for (int i = 0; i < this.argNumInputs; i++)
                {
                    this.hiddenSum[j] += X[i] * this.layer1[i][j];
                }
                this.hiddenSum[j] += this.hiddenBias[j] * 1.0;
                this.hiddenOutput[j] = customSigmoid(this.hiddenSum[j]);
            }
        }

        public double outputFor(double [] X)
        {
            this.hiddenOutputFor(X);
            this.outputSum = 0;

            for (int j = 0; j < this.argNumHidden; j++)
            {
                this.outputSum += this.hiddenOutput[j] * this.layer2[j];
            }

            this.outputSum += 1.0 * this.outputBias;

    //          return customSigmoid(this.outputSum);
            return this.outputSum;
        }

        public double train(double [] X, double argValue)
        {
            double predict = outputFor(X);
    //                  double outputError = (argValue - predict) *
customSigmoidDerivative(this.outputSum);
            double outputError = argValue - predict;
            updateLayer2(outputError);
            double[] hiddenErrors = new double[this.argNumHidden];
            for (int j = 0; j < this.argNumHidden; j++)
            {
                hiddenErrors[j] = outputError * this.layer2[j] *
customSigmoidDerivative(this.hiddenSum[j]);
            }
            updateLayer1(X, hiddenErrors);
```

```
            return argValue - predict;
        }

        private void updateLayer1(double [] X, double [] hiddenErrors)
        {
            double tempDelta;
            for (int j = 0; j < this.argNumHidden; j++)
            {
                tempDelta = this.argMomentumTerm * this.delataHiddenBias[j]
+ this.argLearningRate * hiddenErrors[j] * 1.0;
                this.hiddenBias[j] += tempDelta;
                this.delataHiddenBias[j] = tempDelta;

                for (int i = 0; i < this.argNumInputs; i++)
                {
                    tempDelta = this.argMomentumTerm * this.delta1[i][j] +
this.argLearningRate * hiddenErrors[j] * X[i];
                    this.layer1[i][j] += tempDelta;
                    this.delta1[i][j] = tempDelta;
                }
            }
        }

        private void updateLayer2(double outputError)
        {
            double tempDelta;
            tempDelta = this.argMomentumTerm * this.deltaOutputBias +
this.argLearningRate * outputError * 1.0;
            this.outputBias += tempDelta;
            // this.outputBias = 0;
            this.deltaOutputBias = tempDelta;
            for (int j = 0; j < this.argNumHidden; j++)
            {
                tempDelta = this.argMomentumTerm * this.delta2[j] +
this.argLearningRate * outputError * this.hiddenOutput[j];
                this.layer2[j] += tempDelta;
                this.delta2[j] = tempDelta;
            }
        }

        private double[] getXArray(double[] states, int action) {
        double[] X = new double[10];
        X[0] = states[0] * 2 - 1;
        X[1] = states[1];
```

```
            X[2] = states[2] * 2 - 1;
            X[3] = states[3] * 2 - 1;
            X[4] = states[4] * 2 - 1;

        //          X[0] = states[0];
        //          X[1] = states[1];
        //          X[2] = states[2];
        //          X[3] = states[3];
        //          X[4] = states[4];
            if (action != 4) {
                X[5] = action / 1.5 - 1;
                X[6] = 0;
            }
            else {
                X[5] = 0;
                X[6] = 1;
            }
            return X;
        }

            public double getQ(double[] states, int action) {
        //          return outputFor(getXArray(states, action)) * 50 + 30;   // corner
            return outputFor(getXArray(states, action)) + 30;   // corner linear
        //          return outputFor(getXArray(states, action)) * 25 + 2;   // fire
        //          return outputFor(getXArray(states, action)) + 2;   // fire linear
            }

            private double QtoY(double Q) {
        //          return (Q - 30) / 50;   // corner
            return (Q - 30);   // corner linear
        //          return (Q - 2) / 25;   // fire
        //          return (Q - 2);   // fire linear
            }

            public void updateNNOff(double[] states_before, double[] states_after, int
    action, double r) {
            double newQ = r + discountFactor * maxQFor(states_after);
            updateQ(states_before, action, newQ);
        }

            public void updateQ(double[] states, int action, double newQ) {
        //          System.out.print("old Q: ");
        //          System.out.print(getQ(states, action));
        //          System.out.print(", delta Q: ");
```

```
//          System.out.print(delta);
//          System.out.print(", new y: ");
//          System.out.println(QtoY(getQ(states, action) + delta));
        train(getXArray(states, action), QtoY(newQ));
//          train(getXArray(states, action), QtoY(getQ(states, action) + delta));
    }

//      public void updateNNOff(double[] states_before, double[] states_after,
int action, double r) {
//          if (r == 0) return;
//          double delta = learningRate * (r + discountFactor *
maxQFor(states_after) - getQ(states_before, action));
//          updateQ(states_before, action, delta);
//      }
//
//      public void updateQ(double[] states, int action, double delta) {
//          System.out.print("old Q: ");
//          System.out.print(getQ(states, action));
//          System.out.print(", delta Q: ");
//          System.out.print(delta);
//          System.out.print(", new y: ");
//          System.out.println(QtoY(getQ(states, action) + delta));
//          train(getXArray(states, action), QtoY(getQ(states, action) + delta));
////          train(getXArray(states, action), QtoY(getQ(states, action) + delta));
//      }

        public int pickAction(double[] states, boolean rand) {
        if (rand) return (int)Math.random() * Action.numActions;
        else {
            return pickBestAction(states);
        }
    }

        public int pickBestAction(double[] states) {
        double maxQ = getQ(states, 4);
        int bestAction = 4;
        for (int i = 0; i < 4; i++) {
            double Q = getQ(states, i);
            if (Q > maxQ) {
                maxQ = Q;
                bestAction = i;
            }
        }
        return bestAction;
```

```
}

    public double maxQFor(double[] states) {
    double maxQ = getQ(states, 4);
    for (int i = 0; i < 4; i++) {
        double Q = getQ(states, i);
        if (Q > maxQ) maxQ = Q;
    }
    return maxQ;
}

    public void printLayers(int mode)
    {
        if (mode == 0 || mode == 1)
        {
            for (int i = 0; i < this.argNumInputs; i++)
            {
                for (int j = 0; j < this.argNumHidden; j++)
                {
                    System.out.print(this.layer1[i][j]);
                    System.out.print(' ');
                }
                System.out.println();
            }
        }
        if (mode == 0 || mode == 2)
        {
            for (int j = 0; j < this.argNumHidden; j++)
            {
                System.out.print(this.layer2[j]);
                System.out.print(' ');
            }
            System.out.println();
        }
    }

    public void save(File argFile)
    {
    PrintStream write = null;
    try {
        write = new PrintStream(new RobocodeFileOutputStream(argFile));
        for (int i = 0; i < this.argNumInputs; i++)
            for (int j = 0; j < this.argNumHidden; j++)
                write.println(new Double(this.layer1[i][j]));
```

```
        for (int i = 0; i < this.argNumHidden; i++)
            write.println(new Double(this.layer2[i]));
        for (int i = 0; i < this.argNumHidden; i++)
            write.println(new Double(this.hiddenBias[i]));
        write.println(new Double(this.outputBias));
        if (write.checkError())
            System.out.println("Could not save the data!");
        write.close();
    }
    catch (IOException e) {
   //     System.out.println("IOException trying to write: " + e);
    }
    finally {
        try {
            if (write != null)
                write.close();
        }
        catch (Exception e) {
  //          System.out.println("Exception trying to close witer: " + e);
        }
    }
    }

    public void loadFile(File file) throws IOException {
    BufferedReader read = null;
    try {
        read = new BufferedReader(new FileReader(file));
        for (int i = 0; i < this.argNumInputs; i++)
            for (int j = 0; j < this.argNumHidden; j++)
                this.layer1[i][j] = Double.parseDouble(read.readLine());
        for (int i = 0; i < this.argNumHidden; i++)
            this.layer2[i] = Double.parseDouble(read.readLine());
        for (int i = 0; i < this.argNumHidden; i++)
            this.hiddenBias[i] = Double.parseDouble(read.readLine());
        this.outputBias = Double.parseDouble(read.readLine());
    }
    catch (IOException e) {
    //    System.out.println("IOException trying to open reader: " + e);
        initializeWeights();
    }
    catch (NumberFormatException e) {
        initializeWeights();
    }
    finally {
```

```
                try {
                    if (read != null)
                        read.close();
                }
                catch (IOException e) {
              //    System.out.println("IOException trying to close reader: " + e);
                }
            }
        }

            public void load(String argFileName) throws IOException
            {}
    }
```

4. Jbot_NN.java

```
package JBot_NN;

import java.awt.*;
import java.io.IOException;
import java.io.PrintStream;

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

public class JBot_NN extends AdvancedRobot {
 int action_old, action;
 double[] states_old = new double[5];
 double[] states = new double[5];
 double reward;
 double oppoDist, oppoBearing;
 boolean found = false;
 double epsilon = 0;    // when 1.0 always random
 boolean useInterReward = true;    // when false only collects terminal rewards
 boolean train = true;    // when false JBot does not learn, i.e. LUT does not update
//       double interRewardGood = 1.4;
//       double interRewardBad = 0.8;
//       double terminalRewardWin = 25;
```

```
//      double terminalRewardDeath = 15;

 // corner
//      NeuralNet myNet = new NeuralNet(10, 20, 0.2, 0.9, -1, 1);
//      NeuralNet myNet = new NeuralNet(10, 15, 0.045, 0.35, -1, 1);
//      NeuralNet myNet = new NeuralNet(10, 100, 0.005, 0.6, -1, 1);
//      NeuralNet myNet = new NeuralNet(10, 100, 0.2, 0.1, -1, 1);
//      NeuralNet myNet = new NeuralNet(10, 100, 0.002, 0.9, -1, 1);
//      NeuralNet myNet = new NeuralNet(7, 100, 0.02, 0.6, -1, 1);
//      NeuralNet myNet = new NeuralNet(7, 100, 0.005, 0.5, -1, 1);    // corner
//      NeuralNet myNet = new NeuralNet(7, 40, 0.005, 0.05, -1, 1);    // corner linear

 // fire
//      NeuralNet myNet = new NeuralNet(7, 100, 0.05, 0.1, -1, 1);
 NeuralNet myNet = new NeuralNet(7, 40, 0.05, 0.1, -1, 1);    // fire linear
 public void run() {    // off policy
        if (!train) epsilon = 0;
        loadData();
        setColors(Color.darkGray, Color.black, Color.lightGray);
        setAdjustGunForRobotTurn(false);    // when false gun is fixed to body
        setAdjustRadarForGunTurn(false);    // when false radar is fixed to body
        turnRadarRightRadians(2 * Math.PI);

        updateState();
        while (true) {
            updateOldState();
            action = myNet.pickAction(states, Math.random() < epsilon);
            reward = 0.0;
            switch (action)
                {
                    case Action.frontLeft:
                        setAhead(Action.forewardDist);
                        setTurnLeft(Action.turnDegree);
                        break;
                    case Action.frontRight:
                        setAhead(Action.forewardDist);
                        setTurnRight(Action.turnDegree);
                        break;
                    case Action.backLeft:
                        setBack(Action.backwardDist);
                        setTurnRight(Action.turnDegree);
                        break;
                    case Action.backRight:
                        setBack(Action.backwardDist);
```

```
                        setTurnLeft(Action.turnDegree);
                        break;
                    case Action.fire:
                        ahead(0);
                        turnLeft(0);
                        scanAndFire();
                        break;
                }
            execute();
            while (getDistanceRemaining() != 0 || getTurnRemaining() != 0) execute();
            turnRadarRightRadians(2 * Math.PI);
            updateState();
            if (train) myNet.updateNNOff(states_old, states, action, reward);
        }
    }

//      public void run() {    // on policy
//          if (!train) epsilon = 0;
//          loadData();
//          setColors(Color.darkGray, Color.black, Color.lightGray);
//          setAdjustGunForRobotTurn(false);    // when false gun is fixed to body
//          setAdjustRadarForGunTurn(false);    // when false radar is fixed to body
//
//          turnRadarRightRadians(2 * Math.PI);
//          updateState();
//          action = lut.pickAction(state, Math.random() < epsilon);
//          state_old = state;
//          action_old = action;
//          while (true) {
//              reward = 0.0;
//              switch (action)
//              {
//                  case Action.frontLeft:
//                      setAhead(Action.forewardDist);
//                      setTurnLeft(Action.turnDegree);
//                      break;
//                  case Action.frontRight:
//                      setAhead(Action.forewardDist);
//                      setTurnRight(Action.turnDegree);
//                      break;
//                  case Action.backLeft:
//                      setBack(Action.backwardDist);
//                      setTurnRight(Action.turnDegree);
//                      break;
```

```
//                      case Action.backRight:
//                          setBack(Action.backwardDist);
//                          setTurnLeft(Action.turnDegree);
//                          break;
//                      case Action.fire:
//                          ahead(0);
//                          turnLeft(0);
//                          scanAndFire();
//                          break;
//                  }
//                  execute();
//                  while (getDistanceRemaining() != 0 || getTurnRemaining() != 0) execute();
//                  turnRadarRightRadians(2 * Math.PI);
//                  updateState();
//                  action = lut.pickAction(state, Math.random() < epsilon);
//                  if (train) lut.updateLUTOn(state_old, state, action_old, action, reward);
//                  state_old = state;
//                  action_old = action;
//              }
//      }

    public void scanAndFire() {
        found = false;
        while (!found) {
            setTurnRadarLeft(360);
            execute();
        }
        turnGunLeft(getGunHeading() - getHeading() - oppoBearing);
        double currentOppoDist = oppoDist;
        if (currentOppoDist < 101) fire(6);
        else if (currentOppoDist < 201) fire(4);
        else if (currentOppoDist < 301) fire(2);
        else fire(1);
    }

    private void updateState() {
        this.states[0] = States.getHeading(getHeadingRadians());    // heading
        this.states[1] = States.getTargetDistance(oppoDist);    // targetDistance
        this.states[2] = States.getTargetBearing(oppoBearing);    // targetBearing
        this.states[3] = States.getXPosition(getX());    // xPosition
        this.states[4] = States.getYPosition(getY());    // yPosition
    }

    private void updateOldState() {
```

```
        for (int i = 0; i < this.states.length; i++) this.states_old[i] = this.states[i];
    }

    //------------------------Events---------------------------------
    public void onScannedRobot(ScannedRobotEvent e) {
        oppoDist = e.getDistance();
        oppoBearing = e.getBearing();
        found = true;
    }

    public void onBulletHit(BulletHitEvent e) {
        if (useInterReward) reward += 7.5 * e.getBullet().getPower();
    }

    public void onBulletMissed(BulletMissedEvent e) {
        if (useInterReward) reward -= 8 * e.getBullet().getPower();
    }

    public void onHitByBullet(HitByBulletEvent e) {
        if (useInterReward) reward -= e.getBullet().getPower() * 8.5;
    }

    public void onHitWall(HitWallEvent e) {
        if (useInterReward) reward -= 10;
    }

    public void onWin(WinEvent event) {
        if (train) saveData();
        reward += 100;
//            System.out.print("wwwwwwwwwww");
        if (train) saveBattleHist(1);
    }

    public void onDeath(DeathEvent event) {
        if (train) saveData();
        reward -= 75;
//            System.out.print("ddddddddddddd");
        if (train) saveBattleHist(0);
    }

    //------------------------File IO---------------------------------
    public void loadData() {
        try {
            myNet.loadFile(getDataFile("NN.dat"));
```

```
            }
            catch (Exception e) {
            }
        }

        public void saveData() {
            try {
                myNet.save(getDataFile("NN.dat"));
            }
            catch (Exception e) {
                out.println("Exception trying to write: " + e);
            }
        }

        public void saveBattleHist(int win) {
            PrintStream write = null;
            try {
                write = new PrintStream(new
    RobocodeFileOutputStream(getDataFile("battle_history_NN.dat").getAbsolutePath(), true));
                write.println(new Double(win));
                if (write.checkError())
                    System.out.println("Could not save the data!");
                write.close();
            }
            catch (IOException e) {
           //     System.out.println("IOException trying to write: " + e);
            }
            finally {
                try {
                    if (write != null)
                        write.close();
                }
                catch (Exception e) {
          //          System.out.println("Exception trying to close witer: " + e);
                }
            }
        }
    }
```

5. CommonInterface.java

```
package JBot_NN;

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

```
/**
 * This interface is common to both the Neural Net and LUT interfaces.
 * The idea is that you should be able to easily switch the LUT
 * for the Neural Net since the interfaces are identical.
 * @date 20 June 2012
 * @author sarbjit
 *
 */
public interface CommonInterface {
    /**
     * @param X The input vector. An array of doubles.
     * @return The value returned by th LUT or NN for this input vector
     */
    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(String argFileName) throws IOException;
}
```

6. *Action.java*

```
package JBot_NN;
```

```
public class Action {
    public static final int frontLeft = 0;
    public static final int frontRight = 1;
    public static final int backLeft = 2;
    public static final int backRight = 3;
    public static final int fire = 4;

    public static final int numActions = 5;

    public static final double forewardDist = 200.0;
    public static final double backwardDist = 150.0;
    public static final double turnDegree = 45.0;
}
```