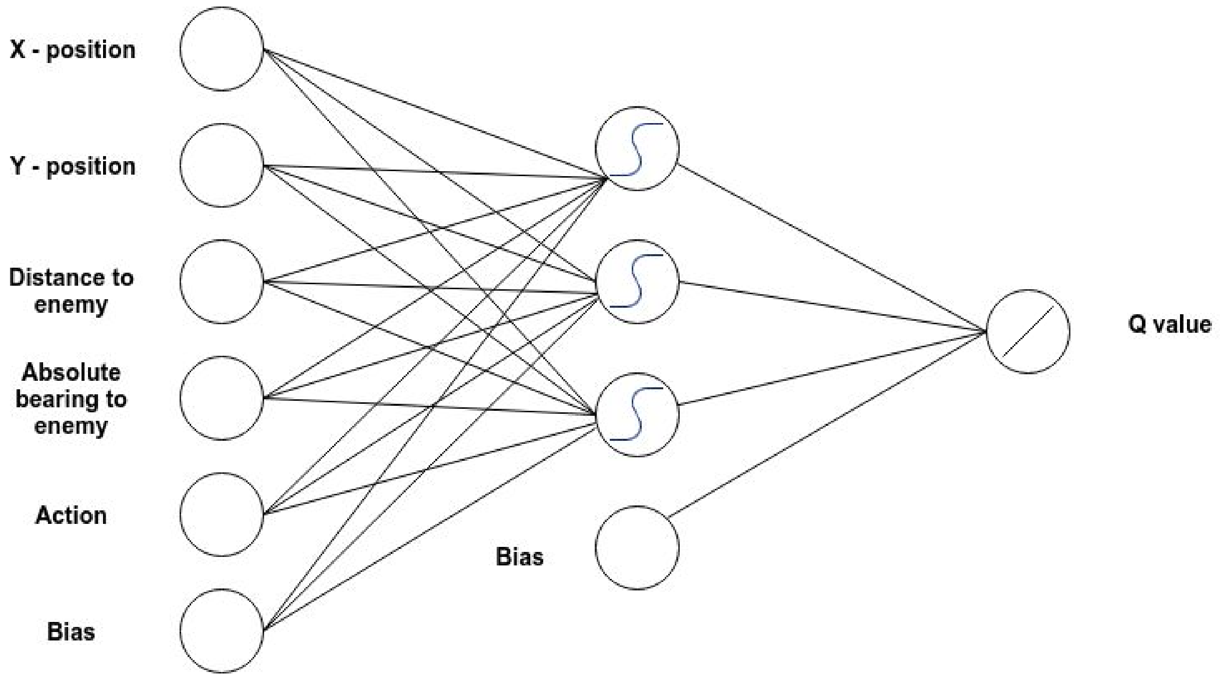
Part 3 – Reinforcement Learning with Backpropagation

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

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.



Architecture of neural network

The Architecture of the neural network is shown above. 3 hidden neurons are considered based on the graphs of training and validation sets (mentioned in question 4.b). The hidden neurons use the sigmoid activation function and the output neuron uses the linear activation function in order to perform linear regression on the input data.

Based on the offline training of the LUT, the following input and output representation is considered:

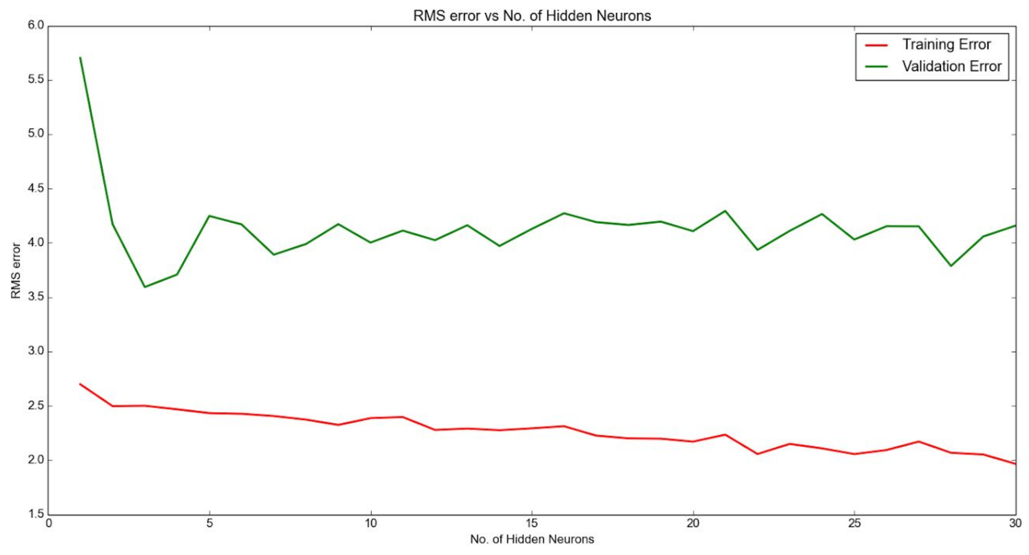
Input No	Input	Non Quantized values	Inputs to NN
1	X position	0-800 pixels	{0,0.01....7.99,8}
2	Y position	0-600 pixels	{0,0.01....5.99,6}
3	Distance to enemy	0-1000 pixels	{0,0.01....9.99,10}
4	Absolute bearing to enemy	0-360 degrees	{0,0.01....3.99,4}
5	Action	{1,2,3,4}	{1,2,3,4}

Output: Q value of the Q learning algorithm.

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 hyperparameter 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.

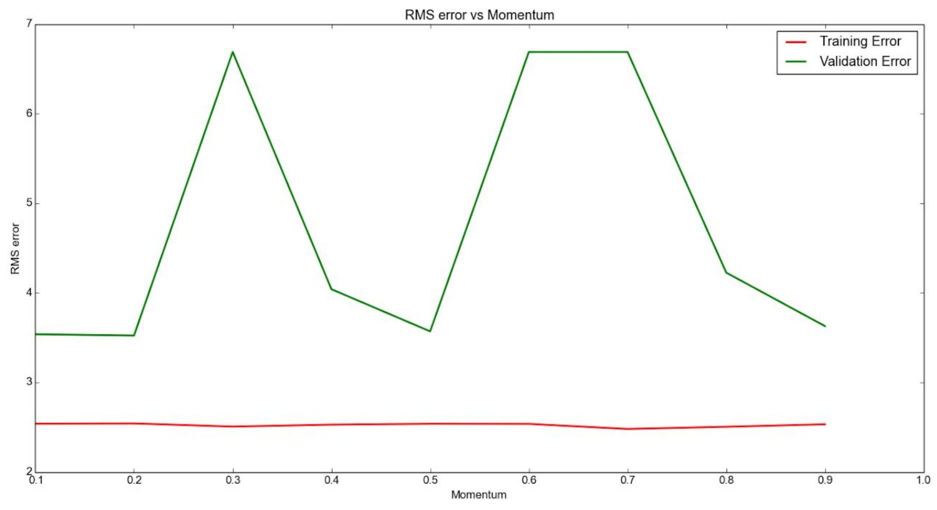
The LUT got from part is used to train the neural network (ϵ = 0.6 is considered as it gave the best results for part 2). This data is divided into training and validation sets. Training set consists of 826 samples and the validation set consists of 440 samples. The Root Mean Square (RMS) error is plotted for both the training and the validation sets across number of hidden neurons.

As can be seen from the following graph, the validation error decreases until **n=3** and is is minimum for the same, after which it diverges again. Here we make the tradeoff between bias and variance in our implementation. We chose the model with less variance and possibly greater bias. This is because we don't want our model to overfit to our training data and perform poorly on new samples as seen from the graph when *n* increases. The rationale is that with just 5 inputs (excluding bias), given sufficient time, the neural network should be able to approximate the Q values using three hidden neurons. We also note that the **least RMS error achieved is ~3.5 for validation set and ~2.5 for the training set**, which is not satisfactory, but stick to this to avoid overfitting.



Training and Validation RMS error

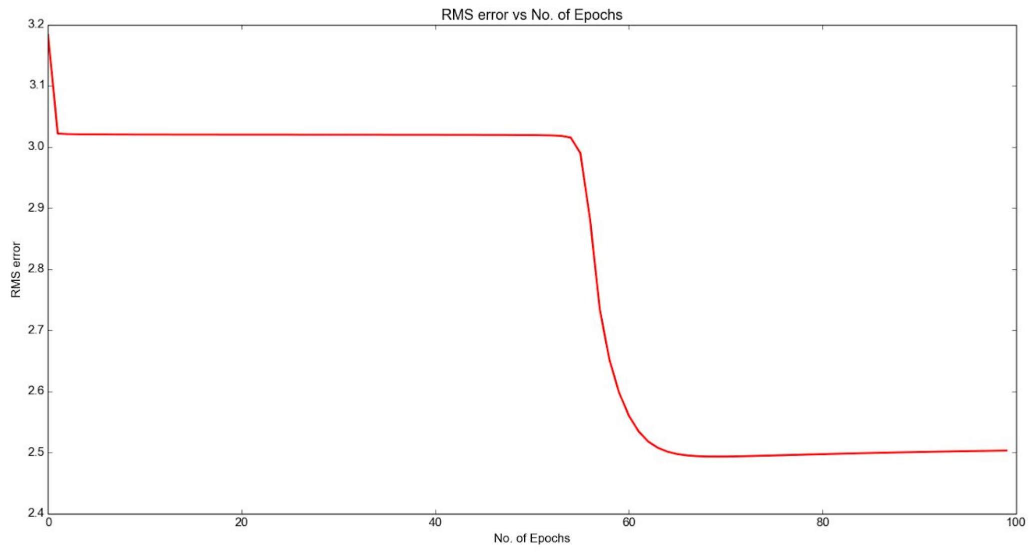
Once the number of hidden neurons is obtained, the momentum value is changed and results are observed.



Momentum change

It can be observed that the training error doesn't change much with change in momentum values (Similar results are obtained by changing learning rate) and the validation error shows no particular trend. Therefore, supported by the graph, **momentum value of 0.9** is chosen (giving importance to previous weight change values). Also, in order to trade off between the training time and RMS error value obtained, **learning rate (α) of 0.1** is used.

Using these values, RMS error is plotted vs number of epochs. It can be seen from the graph that the NN saturates at an RMS error of 2.5 very early and doesn't improve significantly after. This could be due to the batch training from the LUT. The NN performs much better when trained online, as will be shown in *5.(b)*.



RMS error

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.

Comparison of input representation of Part 2 (LUT) and Part 3 (NN):

Inputs	**LUT**	**Non Quantized values**	**Inputs to NN**
X position	{1,2,3,4,5,6,7,8}	0-800 pixels	{0,0.01....7.99,8}
Y position	{1,2,3,4,5,6}	0-600 pixels	{0,0.01....5.99,6}
Distance to enemy	{1,2,3,4}	0-1000 pixels	{0,0.01....9.99,10}
Absolute bearing to enemy	{1,2,3,4}	0-360 degrees	{0,0.01....3.99,4}
Action	{1,2,3,4}	{1,2,3,4}	{1,2,3,4}

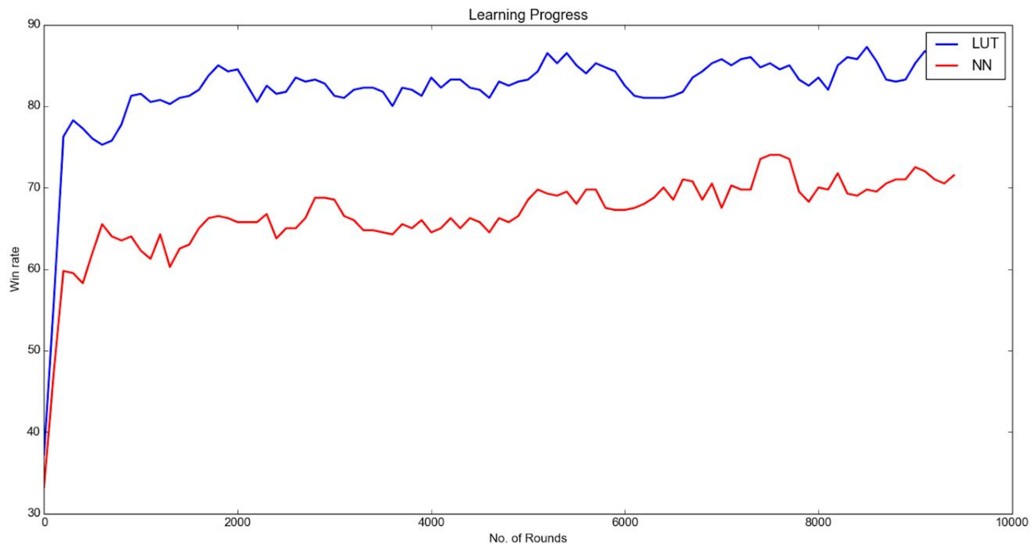
Output: The Q value is taken to be the output. For the LUT, its the column of Q values. For the NN, it's the values learnt by regression.

A Sample input/output vector representation is shown below.

Implementation	**Input Vector**	**Output**
LUT	[1, 2, 1, 2, 2]	1.5613317420010555
NN	[7.82, 1.24, 2.36, 3.14, 4]	2.4783255893201003

The graph of learning progress for the LUT and NN across number of rounds is shown below. In order to track learning progress, win rate is plotted vs number of rounds (same as in part 2). Win rate is the number of wins for every 100 rounds. It is seen that the LUT implementation from part 2 learns quickly and reaches asymptotic performance within 8K rounds, whereas the NN learns slowly and doesn't reach maximum performance even at 10K rounds. This is expected as the LUT values, even though bootstrapped, converge fast because there's no approximation involved and the robot learns better actions quickly. But the NN is still approximating in the early stages and it takes time to learn the weights that approximate better. It should be noted that the **NN is learning completely online, where the weights are initialized randomly.**

As will be shown in *5.(b)*, the NN does start approximating better as the number of rounds increase and the weight values learnt are much better.



LUT vs NN

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 lookup table.

The entire state space needs to be discretized and stored in a LUT with each discrete state having a Q value. With a stochastic environment with large number of factors that could impact the decisions, discretizing all of them and storing them and their Q values in a LUT would take up a lot of memory space.

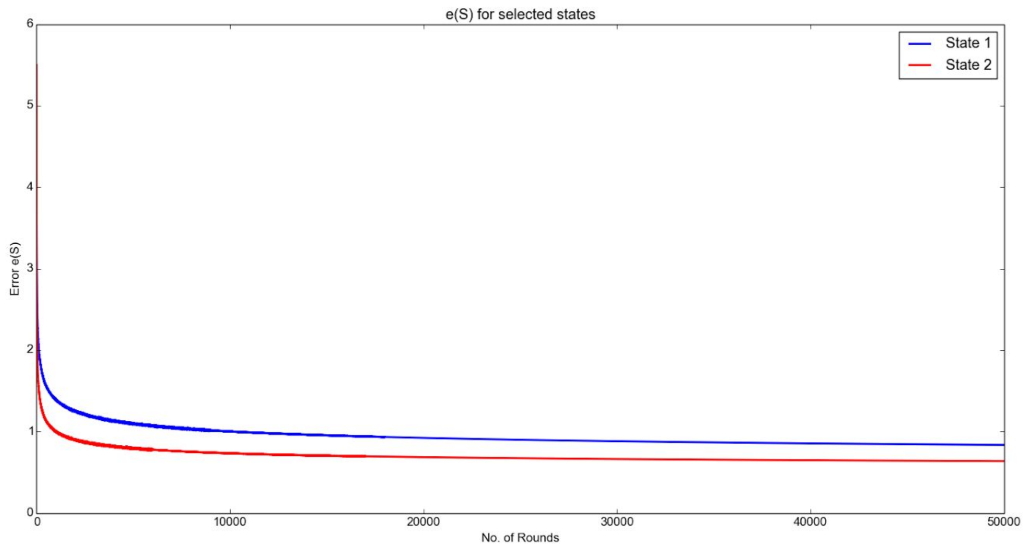
A neural network on the other hand doesn't need to discretize the states and store them. It can compute the Q values online given a particular state by just feeding to the learnt layer of weights. The only memory space required would be to store the weights and this scales with the size of the network and not the stochastic environment.

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

a) 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 win rate is calculated as the number of wins for every 100 rounds and this value peaked at **87%** during the observed time of 50K rounds. Here the Neural network is trained completely online with random initialization of weights and the value function is approximated on the fly. Enemy chosen is **Tracker**. The graph for the win rate is included in the answer to *5.(b)*.

State 1: [7.82, 1.24, 2.36, 3.14, 4]
State 2: [4.76, 3.48, 1.15, 2.42, 2]

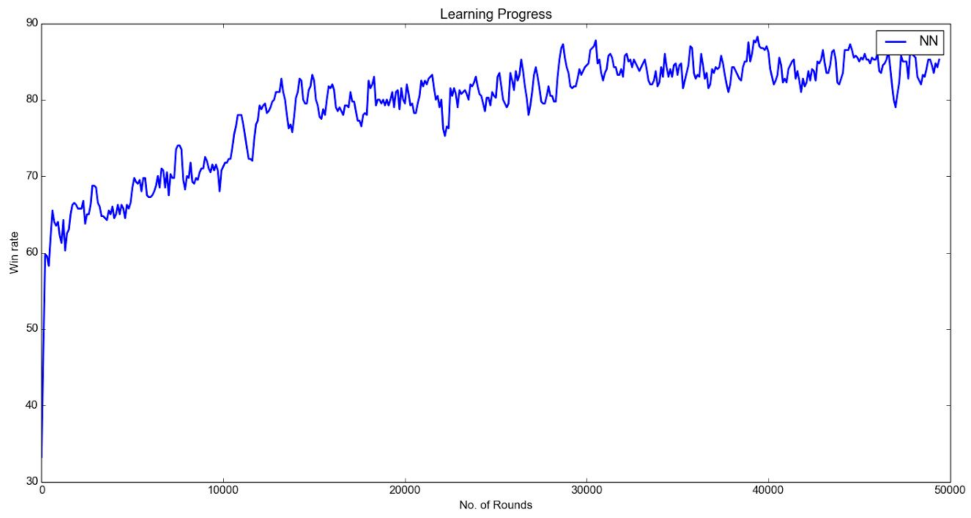


e(S) over iterations

The error $e(S)$ is plotted vs number of rounds and it can be seen that $e(S)$ approaches 0.8 for state 1 and 0.6 for state 2 as the number of iterations increases. This tells us that our robot is learning to approximate the Q values but not too perfectly as the error is not equal to zero. This can be attributed to the nonlinear nature of the neural network hypothesis and nonconvex nature of the loss function. This is elaborated in part *5.(c)*.

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

The win rate (number of wins for every 100 rounds) is plotted vs number of rounds. As can be observed from the graph, the win rate increases steadily with increasing number of rounds and improves asymptotically with number of rounds approaching 50K.



Learning progress

c) Theory question: With a lookup 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 given by

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

Where $V(S_t)$ and $V(S_{t+1})$ are the value functions at time t and $t+1$ respectively. r_t is the reward at time t and γ is the discount factor.
The Bellman equation for the optimal value function is given by

$$V^*(S_t) = r_t + \gamma V^*(S_{t+1}) \tag{2}$$

Relating the value function and the optimal value function at time t we have

$$V(S_t) = e(S_t) + \gamma V^*(S_t) \tag{3}$$

Also similarly

$$V(S_{t+1}) = e(S_{t+1}) + \gamma V^*(S_{t+1}) \tag{4}$$

Substituting these values of $V(S_t)$ and $V(S_{t+1})$ in the equation (1) we have

$$e(S_t) + V^*(S_t) = r_t + \gamma(e(S_{t+1}) + V^*(S_{t+1})) \tag{5}$$

$$e(S_t) + V^*(S_t) = r_t + \gamma V^*(S_{t+1}) + \gamma e(S_{t+1}) \tag{6}$$

Replace the first part of RHS using equation (2)

$$e(S_t) + V^*(S_t) = V^*(S_t) + \gamma e(S_{t+1}) \tag{7}$$

This leaves us with

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

Now we know that for a terminal state whose reward is known, the error would be zero.

$$e(S_T) = 0$$

Where S_T is the terminal state.
Therefore with enough number of iterations, the error in the preceding states also becomes zero and the value function equals the optimal value function. One has to note that this is only when the value function is not approximated.

When functional approximation is carried out using a neural network or some other learning algorithm, an additional approximation error is introduced in the value function.

$$e(S_t) = V(S_t) - V^*(S_t) + e_{approx}$$

This approximation error can go to zero if a linear combination of the basis functions is used (which would lead to a convex loss function) as shown in [3]. But if we use non linear hypothesis (neural network) this would lead to the loss function that is neither convex nor concave and hence global minima cannot be guaranteed [1]. This implies that **the value function may get stuck in a local minima arbitrarily far from the optimal value function we desire if we use non linear hypothesis for function approximation.**

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: Read Up on experience replay.

Some of the methods that can be used to monitor learning performance of the neural net are:

1. As having the Q values for the entire state space is not possible and beats the purpose of using a neural network, it could be noted initially which states are most visited and these and their Q values could be stored in memory. Later random mini batches from this set could be used to train the network instead of feeding it with new samples each time. This would not only help us keep track of the training error but also keep the NN out of local minima due to correlation between successive samples during online training.
2. A crude way to measure learning performance of the robot would be to keep track of the cumulative reward in each round. The robot would initially receive low cumulative rewards and as it learns, the cumulative reward in each round would keep growing before saturating at a certain stage.
3. Tracking the win rate is also a good way to monitor performance. During the initial stages the win rate will be suboptimal, but one can see the win rate improving as the robot learns in further rounds. This implementation uses the win rate to track the learning performance of the robot.

6) Overall Conclusions

a) This question is open-ended and offers you an opportunity to reflect on what you have learned overall through this project. For example, what insights are you able to offer with regard to the practical issues surrounding the application of RL & BP to *your problem? E.g. What could you do to improve the performance of your robot? How would you suggest convergence problems be addressed? What advice would you give when applying RL with neural network based function approximation to other practical applications?*

Insights with regard to the practical issues surrounding the application of RL & BP to the problem

1. The loss function of a neural network is in general neither convex nor concave. This is because if one were to permute the nodes in the hidden layer and likewise change the associated weights, then one would still end up with another minimum. This poses a problem for us during training of our network. It should be noted that the best RMS error obtained is around 2.5 which is not particularly satisfying. This could be because of the neural network getting stuck at a local minima.
2. The number of hidden layers and number of neurons in each hidden layer is still a practical issue to deal with. In this implementation we simplify the problem by considering only one hidden layer. But performing search techniques like grid search to get the best values may be computationally intensive.
3. The neural network is prone to overfitting or underfitting and the tradeoff has to be made. This reduces with increasing number of samples (increasing number of rounds) if the proper choice of hidden neurons is made.
4. Functional approximated RL may not converge to the optimal Q values, leaving us with suboptimal results.
5. RL learning depends on the stochastic environment it experiences. This environment changes with the chosen enemy. Therefore this trained robot won't perform well against all enemies. This can be mitigated by keeping a track of all the possible actions of different robots and then including those actions in the state space.

What could you do to improve the performance of your robot?

1. Perform search techniques like grid search to chose the best value of number of hidden layers and neurons.
2. By using a deep neural network coupled with appropriate regularization in the loss function and dropouts, overfitting can be mitigated along with further reduction in the RMS error.

3. Implement varying learning rate and momentum so that the neural network doesn't get stuck in a local minimum.
4. As mentioned previously, experience replay could be used to mitigate correlated samples and local minima as well.

How would you suggest convergence problems be addressed?

1. Implement varying learning rate and momentum.
2. Experience replay implementation.
3. With increasing popularity of ReLU and LReLU transfer functions as they solve the issue of vanishing gradients, these along with dropout implementation could help the neural network converge faster.
4. [1] presents the Bridge algorithm for RL with better error bounds which won't be covered here. The reader can refer to mentioned paper for mathematical details.

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

1. It is better to go with a deep neural network implementation with regularization and dropouts. This would ensure low RMS error while avoiding overfitting.
2. It's imperative that you document everything about the stochastic environment beforehand. If one is looking at a general purpose implementation, then the salient characteristics of the environment have to be observed over a period of time. The RL robot can learn stochastically. But it can't drive a car in the city if you train it on open roads, even though the two tasks are relatively similar in nature.
3. Training of Generative Adversarial Networks with Reinforcement Learning could be tried as a method of self correcting and improvising.

b) Theory question: Imagine a closed-loop control system for automatically delivering anesthetic to a patient undergoing surgery. You intend to train the controller using the approach used in this project. Discuss any concerns with this and identify one potential variation that could alleviate those concerns.

1. An anesthetist would need to be aware of all the possible types of patients and different types of anesthetic. A general purpose RL is difficult to train.
2. Considering an RL controller would be tricky for this task as the RL algorithm needs to learn from its own mistakes by exploring at first and this would put the patient's life at risk.
3. Image recognition (or any other sensory input drawn) would need to be top notch with a high awareness of the surrounding environment.
4. The surgery room is dynamic and things can change in a matter of seconds. It is difficult to tell whether a RL controller can be trusted with the job of working alongside humans with behaviour patterns that are highly unpredictable.

One potential variation could be carried out by training the controller in a simulated environment with no real risk, like the Model Predictive Control used in [2]. This comes with its own concerns of how accurately the surgery environment can be modelled and whether the controller will be capable of working with real world issues mentioned previously.

References

1. Vassilis A. Papavassiliou and Stuart Russell, Convergence of Reinforcement Learning with general function approximators, IJCAI'99 Proceedings of the 16th international joint conference on Artificial intelligence - Volume 2, 1999
2. Gaweda AE, Muezzinoglu MK, Jacobs AA, Aronoff GR, Brier ME, Model predictive control with reinforcement learning for drug delivery in renal anemia management, Conf. Prc. IEEE Eng. Med. Biol. Sci., 2006
3. John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDSP-2322, Laboratory for Information and Decision Systems,Massachusetts Institute of Technology, 1996.

Appendix

```
package sample.Kevin.Try2.NN1; //change it into your package name

import static robocode.util.Utils.normalRelativeAngleDegrees;
import java.awt.Color;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import com.sun.javafx.geom.Point2D;

import robocode.AdvancedRobot;
import robocode.BattleEndedEvent;
import robocode.BulletHitEvent;
import robocode.DeathEvent;
import robocode.HitByBulletEvent;
import robocode.HitRobotEvent;
import robocode.HitWallEvent;
import robocode.RobocodeFileOutputStream;
import robocode.RoundEndedEvent;
import robocode.ScannedRobotEvent;
import robocode.WinEvent;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;

@SuppressWarnings("unused")
public class Rl_nn extends AdvancedRobot {

        // declare variables
        final double epsilon = 0.6;
        final double gamma = 0.9;
        final double alpha = 0.1;

    //LUT table initialization
    int[] action=new int[4];
    int[] actionsAll=new int[4];
    double[] Xtrain=new double[5];
        double[] Xtrain_next=new double[5];
        double[] Ytrain=new double[1];
        static String [][] weight = new String[3*6 + 3 + 1][3];
        Map<String, Object> multiValues = new HashMap<String, Object>();

    //standard robocode parameters
    double turnGunValue;
        double bearing;
    double absbearing=0;
    double distance=0;
    private double getVelocity;
        private double getBearing;

    //quantized parameters
    double quantX=0;
    double quantY=0;
    double quantDist=0;
    double quantBearingAbs=0;

    //initialize reward related variables
    double reward=0;
    double presentQValue;
    int actionEpsRandom=0;
    int chosenAction = 0;
    double NextQValue;
    double latestQValue;
        int actionQGreedy=0;
        int[] matchActions=new int[actionsAll.length];
        double[] optionsQ=new double[actionsAll.length];
        double[] q_possible=new double[actionsAll.length];

        //counting variables
        int count = 0;
        static int [] winsRate = new int[10000];


public void run(){

        //if(getRoundNum() != 0)
        //{
                try {
          weightUpdate();}
          catch(IOException e) {
                e.printStackTrace();
          }
        //}

        count+=1;

        //set colour
       setColors(null, new Color(200,0,192), new Color(0,192,192), Color.black, new
Color(0, 0, 0));
   setBodyColor(new java.awt.Color(192,100,100,100));

   while(true){

        //draw a random value for E greedy
        Random rand = new Random();
        double epsilonCheck = rand.nextDouble();


   //turn gun to scan
        turnGunRight(360);

        //random action with prob = epsilon
        if (epsilonCheck <= epsilon) {
                actionEpsRandom=intRandom(1,actionsAll.length);

                Xtrain[0]=quantX;
                Xtrain[1]=quantY;
                Xtrain[2]=quantDist;
        Xtrain[3]=quantBearingAbs;
        Xtrain[4]=actionEpsRandom;

        System.out.println("Hi");

        try {
        if (getRoundNum() == 0)
        multiValues = NeuralNet.start(Xtrain,Ytrain,false,false,weight);
        else
        multiValues = NeuralNet.start(Xtrain,Ytrain,false,false,weight);
        }
        catch (IOException e) {
                e.printStackTrace();
                }


        presentQValue = (double) multiValues.get("value");
        weight = (String[][]) multiValues.get("array");


}

//greedy with prob = 1 - epsilon
else if (epsilonCheck > epsilon) {
        try {
                presentQValue = Qpolicy();
        }
        catch (IOException e) {
                e.printStackTrace();
                }
} // back outside

/*-------------common code-------------*/

 //reset reward to 0
 reward=0;

 /*---------common code ends------------*/
```

```
        //take random action
        if (epsilonCheck <= epsilon) {
        actionSet(actionEpsRandom);
        chosenAction  = actionEpsRandom;
        }

        //take greedy action
        else if (epsilonCheck > epsilon) {
        actionSet(actionQGreedy);
        chosenAction = actionQGreedy;
        }

        /*----------------common code----------------*/

        //scan again
        turnGunRight(360);

        //off-policy Q learning (update state with greedy policy) (This part changes for
Sarsa)
        try {
        NextQValue = Qpolicy();
        }
        catch (IOException e) {
                e.printStackTrace();
                }

        //update the present Q value according to Q learning equation

presentQValue=presentQValue+alpha*(reward+gamma*NextQValue-presentQValue);
        Ytrain[0] = presentQValue;
        //-----update Q value in NN
        try {
        multiValues = NeuralNet.start(Xtrain,Ytrain,false,true,weight);
        }
        catch (IOException e) {
                e.printStackTrace();
        }
        latestQValue = (double) multiValues.get("value");
        weight = (String[][]) multiValues.get("array");
```

```
            weightSave();



}//while loop ends

}//run function ends

//function to support greedy policy
public double Qpolicy() throws IOException
{   /*
        for (int k=0;k<weight.length;k++) {
                System.out.println(weight[k][0]+" "+weight[k][1]+"    "+weight[k][2]);
        }*/

        for(int j=1;j<=actionsAll.length;j++)
        {
                Xtrain[0]=quantX;
                Xtrain[1]=quantY;
                Xtrain[2]=quantDist;
                Xtrain[3]=quantBearingAbs;
                Xtrain[4]=j;

                multiValues =NeuralNet.start(Xtrain,Ytrain,false,false,weight);

                q_possible[j-1] = (double) multiValues.get("value");
                weight = (String[][]) multiValues.get("array");
        }
        /*
        weightSave();
        try {
                weightUpdate();}
                catch (IOException e) {
                        e.printStackTrace();
                }
        */

        //finding action that produces maximum q
        actionQGreedy=getMax(q_possible)+1;


        return q_possible[actionQGreedy-1];

}

//Intermediate reward functions: (turn this off for terminal rewards)
public void onHitRobot(HitRobotEvent event){reward-=2;}
public void onBulletHit(BulletHitEvent event){reward+=3;}
public void onHitByBullet(HitByBulletEvent event){reward-=3;}

//function to chose random action
public static int intRandom(int min, int max) {
  Random rand = new Random();
  int randomNum = rand.nextInt((max - min) + 1) + min;
  return randomNum;
}

//function to get the maximum Q value
public static int getMax(double[] array){

        int indc = 0;
        double maxValue = array[0];
        for (int i = 1; i < array.length; i++) {
          if ( array[i] >= maxValue ) {
              maxValue = array[i];
              indc = i;
           }
        }
        return indc;
}

//function to quantize distance to 4 values
private double quantDistance (double dist) {
        quantDist = dist/100;
        return quantDist;
}
```

```
//function that return values on scanning the enemy
public void onScannedRobot(ScannedRobotEvent e)
        {
        double getVelocity=e.getVelocity();
        this.getVelocity=getVelocity;
        double getBearing=e.getBearing();
        this.getBearing=getBearing;

        //got from trial and error
        this.turnGunValue = normalRelativeAngleDegrees(e.getBearing() + getHeading()
- getGunHeading() -15);

        //distance to enemy
        distance = e.getDistance(); //distance to the enemy
        quantDist=quantDistance (distance);

        //fire depending on the quantized distance
        if(quantDist<=2.50){fire(3);}
        if(quantDist>2.50&&quantDist<5.00){fire(2);}
        if(quantDist>5.00&&quantDist<7.50){fire(1);}

        //My robot
        quantX=quantPos(getX());
        quantY=quantPos(getY());

        //Calculate the coordinates of the robot
        double enemyAngle   = e.getBearing();
        double angle = Math.toRadians((getHeading() + enemyAngle   % 360));
        double oppoX = (getX() + Math.sin(angle) * e.getDistance());
        double oppoY = (getY() + Math.cos(angle) * e.getDistance());

        //absolute angle to enemy
        absbearing=getAbsBearing((float)    getX(),(float)    getY(),(float)    oppoX,(float)
oppoY);
        quantBearingAbs=quantAngle(absbearing); //state number 4

}

//find absolute bearing (borrowed from robocode wiki)
```

```
double getAbsBearing(float xPos, float yPos, float oppX, float oppY) {
        double xo = oppX-xPos;
        double yo = oppY-yPos;
        double hyp = Point2D.distance(xPos, yPos, oppX, oppY);
        double arcSin = Math.toDegrees(Math.asin(xo / hyp));
        double bearing = 0;

        if (xo > 0 && yo > 0) { // both pos: lower-Left
                bearing = arcSin;
        } else if (xo < 0 && yo > 0) { // x neg, y pos: lower-right
                bearing = 360 + arcSin; // arcsin is negative here, actuall 360 - ang
        } else if (xo > 0 && yo < 0) { // x pos, y neg: upper-left
                bearing = 180 - arcSin;
        } else if (xo < 0 && yo < 0) { // both neg: upper-right
                bearing = 180 - arcSin; // arcsin is negative here, actually 180 + ang
        }

        return bearing;
}

//function to quantize bearing to 4 values
private double quantAngle(double bearAbs) {

        return bearAbs/90;
}

//function to chose from 4 set of actions
public void actionSet(int x)
                        {
        switch(x){
                case 1:
                        int moveDirection=+1;
                        setTurnRight(getBearing + 90);
                        setAhead(150 * moveDirection);
                        break;
                case 2:
                        int moveDirection1=-1;
                        setTurnRight(getBearing + 90);
                        setAhead(150 * moveDirection1);
```

```
                    break;
                case 3:
                    setTurnGunRight(turnGunValue);
                    turnRight(getBearing-25);
                    ahead(150);
                    break;
                case 4:
                    setTurnGunRight(turnGunValue);
                    turnRight(getBearing-25);
                    back(150);
                    break;
        }
}

//function to quantize position into 8 values (X) or 6 values (Y)
private double quantPos(double pos) {

        return pos/100;
}

public void weightSave()
{
        PrintStream S = null;
        try {
        S = new PrintStream(new
RobocodeFileOutputStream(getDataFile("weight.txt")));
        for (int k=0;k<weight.length;k++) {
          S.println(weight[k][0]+" "+weight[k][1]+"    "+weight[k][2]);
        }
        }
        catch (IOException e) {
                e.printStackTrace();
        }finally {
                S.flush();
                S.close();
        }
}

public void weightUpdate() throws IOException
```

```
{
        BufferedReader        fileReader        =        new        BufferedReader(new
FileReader(getDataFile("weight.txt")));
        String rowLine = fileReader.readLine();
        try {
        int u=0;
        while (rowLine != null) {
         String[] splitLine = rowLine.split("    ");
         //System.out.println(u);
         String[] splitFurther = splitLine[0].split(" ");
         weight[u][0]=splitFurther[0];
         weight[u][1]=splitFurther[1];
         weight[u][2]=splitLine[1];
         u=u+1;
         rowLine= fileReader.readLine();
        }
        } catch (IOException e) {
          e.printStackTrace();
          }finally {
        fileReader.close();
        }
}

//save win rate
public void saveWinRate()
{
  PrintStream w = null;
  try
  {
   w = new PrintStream(new RobocodeFileOutputStream(getDataFile("winsRate.txt")));
   for(int i=0; i<winsRate.length; i++)
          w.println(winsRate[i]);
  }
  catch (IOException e) {
                e.printStackTrace();
        }finally {
                w.flush();
                w.close();
        }
}


public void onBattleEnded(BattleEndedEvent e) {
	saveWinRate();
}

public void onDeath(DeathEvent event)
{
	reward += -5;
	winsRate[getRoundNum()] = 0;
}

public void onWin(WinEvent event)
{
	reward += 5;
	winsRate[getRoundNum()] = 1;
}

}//Rl_check class
```