Reinforcement Learning with Functional Approximation Using a Neural Network

Steven Spielberg Pon Kumar, Dr. Sarbjit Sarkaria, *University of British Columbia.*

***Abstract*—This work focusses on implementation of reinforcement learning on a robot tank in Robocode frame-work. Q-learning algorithm was used and the state-action pairs and the corresponding Q-values were learnt using Neural Networks. The Neural Network is pretrained with the values from the static Look up Table. The first part contains the problem formulations, which introduces the notion of states, actions, rewards considered in the system. Then later a bit of system architecture, followed by simulation results and future works.**

Keywords*—Reinforcement Learning, Neural Networks, Q-learning, Robocode.

I. Introduction

Reinforcement learning(RL) involves teaching a machine (agent) to think for itself by using a system of rewards. For example, a robot can be trained by giving it a series of rewards: for every correct step taken by the robot a reward is made and for every incorrect move a reward is taken away. Essentially, you teach the robot that the reward is good for it. The robot learns, reinforced by its several rewards. The fundamental classes of methods for solving these problems are Dynamic Programming(DP), Monte Carlo Methods(MCM) and Temporal Difference(TD). To apply DP, it is necessary to observe and know the frequencies with which competing actions occur beforehand. This is known as a model. In practice, such information is typically not available. TD and MC are model-free and are easier to apply. In TD, learning takes place as the episode unfolds, after each action is taken. Whereas in MC, all learning is performed at the end of each episode. In this work TD learning algorithm was applied on a robot in a robot tank. The Q-values obtained for the state-action pair was approximated using a Neural Network model. The detailed simulation results, comparison with discrete Look Up Table (LUT) reinforcement learning, and the detailed description of the architecture of the work are discussed.

A. Theoretical Background

Reinforcement learning (RL) is an approach to automating goal-directed learning and decision-making. This approach comes to solve problems in which an agent interacts with an environment. The agent's actions receive a reward from the environment (usually represented as a real number). RL algorithms aim to find a policy that maximizes the cumulative reward for the agent's actions. The basic RL model consists of:

1) Set of environment states.
2) Set of possible agent actions
3) Set of real scalar rewards.

In a given time the agent perceives its state and a set of possible actions. It decides an action and receives a reward and a new state. Based on this interaction with the environment the agent should develop a policy that maps states into actions, such that the cumulative reward for the agent's actions is maximal. RL has two main classes of methods for RL: methods that search in the space of value functions and methods that search in the space of policies. The evolutionary algorithms approach is example for the second class, where the policy is modified as a whole. Temporal difference (TD) is an example for the first class, in which the attempt is to learn the value function (that determines the rewards). The function is learnt through trial and error and exploring the environment.

B. Methods and Approaches in RL

RL has two main classes of methods,

1) Methods that search in the policy-space, such as Evolutionary Algorithms, perform the learning after a complete execution. These methods don't refer to reward given in every single step during the run, but examine only the cumulative reward after completion.
2) Methods that search in value function space, such as Temporal Differences, consider the reward received in each of the steps. Every approach has its own advantages and disadvantages, so choosing an approach should be based on the specific problem itself. Moreover, some methods consider both step reward and final cumulative reward, so there's no strict distinction.

II. Project Design

The RoboCode platform supplies programmers with an environment of tanks that battles each other on a plain, objects-less screen. The programmer writes only the code that defines the behavior of the tank. The platform provides different tank sensors and controllers. The sensors are represented as a set of get functions (e.g. getEnergy(), getHeading()) and a set of events (e.g. BulletHitEvent, HitByBulletEvent). The programmer use this inputs to determine an action and then outputs it to the game using action functions such as fire(), turnLeft() etc. It is designed by IBM.

III. Problem Formulation

RL aims to learn and optimize a mapping from states to actions by trial and error interactions with the environment. In order to implement RL the following should be formulated:

Dr. Sarbjit is with the Department of Electrical and Computer Engineering, University of British Columbia, Canada

- States
- Actions
- Reward
- Q-function

States
The following states were considered for implementing reinforcement learning in the robocode.

- State 1: x-position of the RL robot
- State 2: y-position of the RL robot
- State 3: distance between RL robot and Enemy Robot
- State 4: Absolute Bearing Angle between RL robot and Enemy Robot

Actions

- Action 1: circling clockwise direction
- Action 2: circling anticlockwise direction
- Action 3: Moving towards enemy
- Action 4: Moving away from enemy

Reward

- RL-robot being hit by enemy robot: reward=-2
- One of RL-robot hits enemy robot: reward=+3
- RL-robot being hit by enemy bullet: reward=-3
- RL-robot hits wall: reward=-2

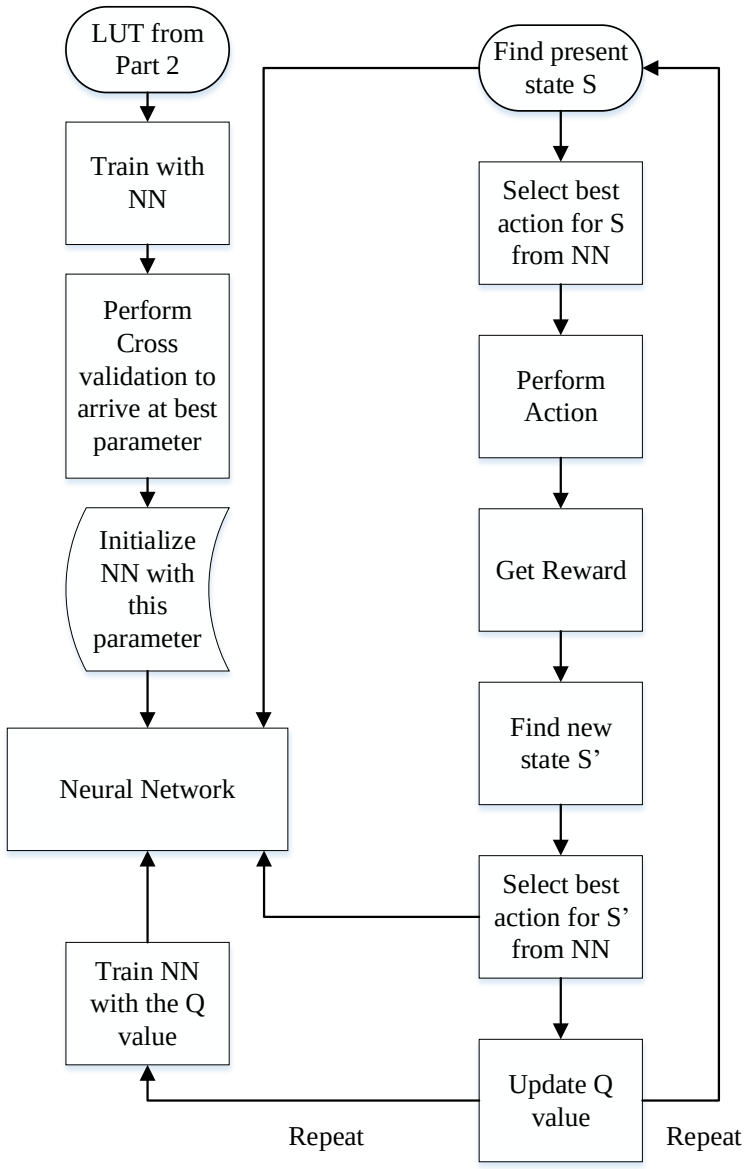
Enemy Tank: Spin Bot.

IV. Results from Part 2

The Reinforcement learning algorithm was implemented on a robot in a robocode with Q-learning. The discretized version of RL was implemented. For that the states in the robocode were quantized. Then, Q-learning framework was used to train the RL robot. After training, the state-action pairs and Q-values were stored in the look up table. The disadvantage of using this version of reinforcement learning is that it requires storing of large state-action pairs. Next, since states are quantized the accuracy is quite less. To overcome this, the non-quantized version of reinforcement learning was implemented using functional approximation with neural network.

V. System Architecture

1) RL Robot: The reinforcement learning robot uses the following algorithm

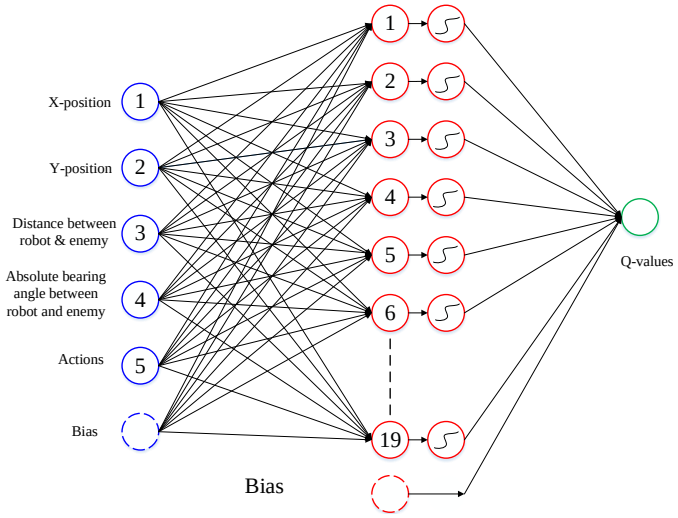


The RL-robot was trained against spinbot (given in sample of robots in robocode) for a total of 100,000 rounds with 90% exploration using the quantized version of qlearning algorithm with look up table. The look up table was trained with Neural network to arrive at the best hyperparameters. The neural network to be used as a functional approximation is initialized with those parameters. Then the neural network was made to learn online by making it to play against spin bot. The q-value for the neural network was updated at each iteration and the RL robot was with neural network.

2) Spin bot: The spin bot is a robot developed by IBM. It spins in the battle field and shot at the enemy on scanning. To make the opponent more competitive wall smoothing was added.

VI. Answers to Question 1

A. Architecture of the Neural Network



1) Neural Network Setting: The typical neural network used for performing linear regression is used. In order to make the neural network to perform Linear regression, the sigmoid activation function should be added to the hidden neurons(not to the output neurons). A neural network with one hidden layer and 4 inputs are used for training the static Look Up Table.

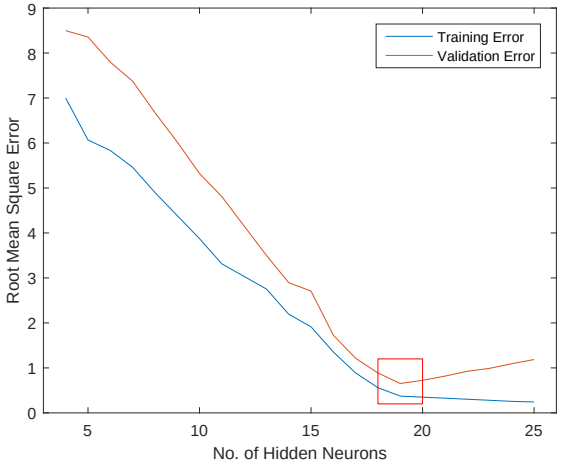
2) Input & Output Representation used in Neural Network:

Input #	Input Representation	Non-Quanitized	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 between robot and enemy	0-1000 pixels	{0,0.01,...9.99,10}
4	Absolute bearing angle between robot and enemy	0-360°	{0,0.01,...3.99,4}
5	Actions	{1,2,3,4}	{1,2,3,4}

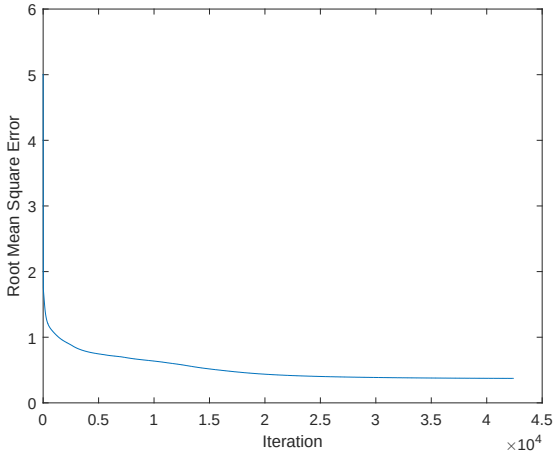
- Output: Q-value from Q-learning algorithm were used as such since the above neural network setting performs linear regression for any ranges of input.

B. Training of Neural Network

1) Tuning of hyper-parameters: The offline LUT table in part 2 was taken to train the neural network. To find the best hyper parameters for neural networks 2-fold cross validation was used. The static LUT table was split into training and cross validation sets. The neural network was trained with the training set with different number of hidden neurons from 4 to 25 and the model parameters were tested on the cross-validation sets. Then the training Root Mean Square and validation Root Mean Square(RMS) were computed. The following plot shows Training and Validation root mean squares for various hidden neurons.



From the above plot, the number of hidden neurons were chosen considering the bias-variance trade-off in Machine Learning. Else as the hidden neurons increases the neural network would overfit the data. Hence it is not a good model. It can be seen that the training and validation error is minimum when number of hidden neurons is 19. So the number of hidden neurons considered for functional approximation of Q-function is 19. The process is also done with various values of step-size and momentum values.



2) Parameters of Neural Network: By Cross validation, the optimum number of hidden neurons were found to be 19. The Training RMS for 19 hidden neurons was found to be **0.3733** and the validation RMS was found to be **0.6536**.

- Number of hidden Neurons: 19
- Step Size: 0.00001
- Momentum: 0.9

- Activation function: bipolar sigmoid
- The weights arrived by this 19 neurons is stored and used for initializing the Reinforcement learning robot.

C. Comparison of Part 2 and Part 3

1) Input Representation in Part 2 and Part 3:

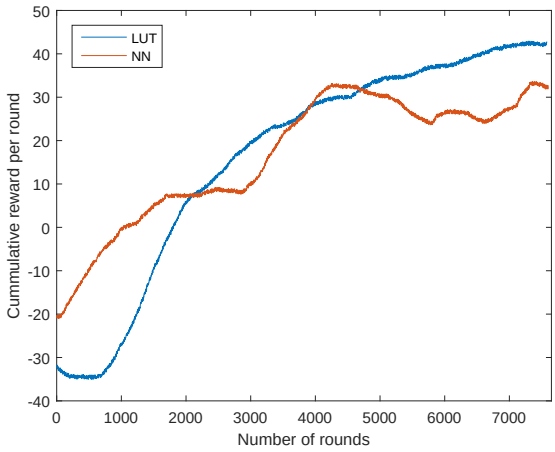
Input Representation	**Part-2 LUT**	**Part-3 (non-quantized)**	**Inputs to NN**
x-position	{1,2,3,...8}	0-800 pixels	0,0.01,....7.99,8
y-position	{1,2,3...6}	0-600 pixels	0,0.01,....5.99,6
distance between robot and enemy	{1,2,3,4}	0-1000 pixels	0,0.01,...9.99,10
Absolute bearing angle between robot and enemy	{1,2,3,4}	0-360°	0,0.01,...3.99,4
Actions	{1,2,3,4}	{1,2,3,4}	{1,2,3,4}

Output Representation	**Part-2 LUT**	**Part-3 (non-quantized)**	**Inputs to NN**
Q-value	{-19.34 to -0.12}	Q-value computed online	Q-value computed online is used as the current NN performs direct regression

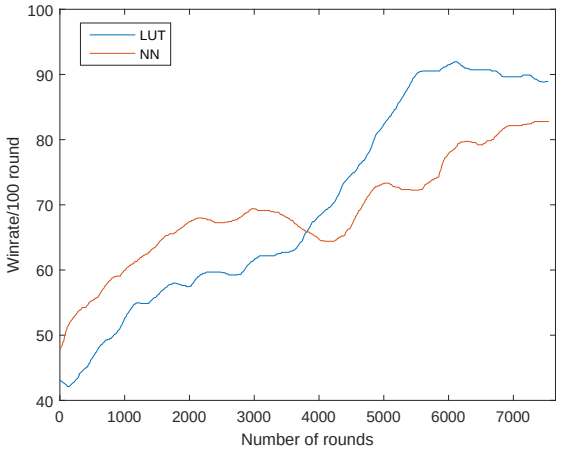
2) Sample Input-Output Vector:

- Input vector: [3.21 5.32 6.78 1.74 2]
- Output vector: [-4.611291003115964]

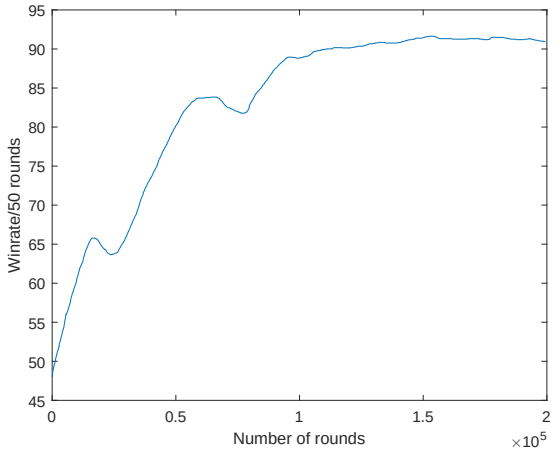
3) Graph Comparison of Part-2 and Part-3: The cumulative reward per round was used to compare the Discrete LUT version and non-quantized version of the Reinforcement Learning robot. The plot of cumulative reward per round vs number of rounds is as shown:



The win rate per 100 rounds were also used to compare part 2 and part 3 and the plot is shown below:



4) Explanation: It can be seen that Neural network Reinforcement Learning robot learns really fast when compared to static discrete LUT reinforcement learning robot. LUT are usually converged at lesser iteration (shown above in the cumulative reward plot) whereas neural networks takes long time to converge as it is learning online. The following plot shows the justification for it, it can be seen that the neural network converges at around 150000 rounds (plot shown below)



D. Neural Network on State Space Reduction

The neural network would not necessarily need the same level of state space reduction as a look up table because reinforcement learning using a look up table is a discretized version. In look up table all the state-action pairs and their corresponding Q-values have to be stored. In practical, it is not feasible to store all the state-action pairs. For example in course work part 2, the state action pairs were not quantized it would have been $800 \times 600 \times 1000 \times 360 = 1.7289 \times 10^{11}$ states. It is very difficult to store all the entries in the look up table with these many states and it takes a long time to learn.

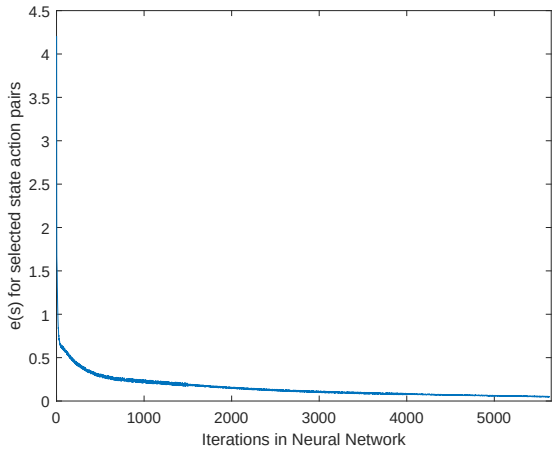
Whereas, Neural network model tries to approximate the state-action pairs and its q-values with its functions, hence we do not have to store state-action pairs separately, we can learn the state-action pairs and its corresponding q-values online using neural network. The cost of storing is just the weights in the neural network which is very less compared to entries encountered in look up table.

VII. Answers to Question 2

A. Win Rate and e(s) for selected state action pair

1) Best Win Rate: The best win rate observed for the reinforcement learning robot with functional approximation using neural network against spin bot was found to be **91%**.

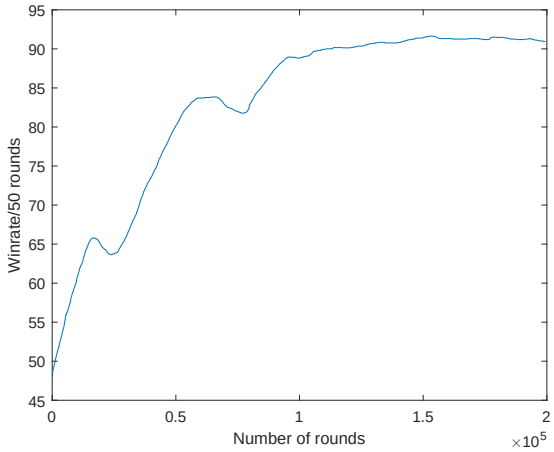
2) Plot of e(s): The e(s)=Q(s',a')-Q(s,a) is computed for the state action pair [4.81 6.43 5.97 0.74 2] and the value of e(s) vs iterations in the neural network is plotted.



It can be seen that the e(s) decreases as the iteration proceeds, this shows that the neural network is learning. It is also noticed that the e(s) is bit noisy in the beginning and the noise reduces with increase in iteration. The reason for noise is that neural network is not learning from a static data it is learning from a dynamic data.

B. Winning Rate Against Number of Battles

The winrate per 50 battles vs number of rounds is plotted as shown



It can be seen that as training proceeds the win rate increases asymptotically but takes a long time to converge (around 150000 rounds)

C. Theory Question:Comment on Convergence

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 function at time t and t+1, r_t is the reward and γ is the discount factor
The bellman equation for optimal value function is written as

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

The value function can also be written with respect to optimal value function and error as

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

From (3),

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

Substituting $V(s_t)$ from equation (3) and $V(s_{t+1})$ in equation (4) in equation (1)

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

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

From(2),

$$e(s_t) + \cancel{V^*(s_t)} = \cancel{V^*(s_t)} + \gamma e(s_{t+1}) \tag{7}$$

$$e(s_t) = \gamma e(s_{t+1}) \tag{8}$$

The error at the terminal for a decision process where reward is known precisely, there is not error in the reinforcement learning is

$$e(s_T) = 0 \tag{9}$$

where s_t is the terminal state
Also from (8) the error at the current state is gamma times the future state and since the terminal error is zero after enough

iterations the error in the preceding states also becomes zero from equation (8) and the value function becomes equal to optimal value function. However this case holds only for discrete case with Look up table approach

1) Approach 1: When the value functions are approximated with neural networks this introduces additional error term in the value function. The error term now becomes,

$$e(s_T) = V(s_t) - V^*(s_t) + e_{fun} \tag{10}$$

where e_{fun} is the error caused by functional approximation. This error becomes biased estimate after many number of iterations provided the gradient descent method for neural network finds a minima. However, there is no fomal proof that the loss functions of neural networks are convex and hence it does not guarantee to find a global minima. This makes the error biased when it finds a minima, else it keeps oscillating(becomes unbiased gradually) if the minmima is not found. Thus the extra error term does not guarantee the convergence of reinforcement learning by functional approximation. Hence value function cannot be equal to optimal value function because of the truncation error introduced by functional approximation.

2) Approach 2:: The gradient descent for the functional approximation interms of value function is given by

$$\begin{aligned}\vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}_t}\big[V^*(s_t) - V_t(s_t)\big]^2 \\ &= \vec{\theta}_t + \alpha\big[V^*(s_t) - V_t(s_t)\big]\nabla_{\vec{\theta}_t}V_t(s_t)\end{aligned} \tag{11}$$

where α is a positive step-size parameter and $\nabla_{\vec{\theta}_t} f(\vec{\theta}_t)$ for any function denotes the vector of partial derivatives, $\left(\frac{\partial f(\vec{\theta}_t}{\theta_t(1)}, \frac{\partial f(\vec{\theta}_t}{\theta_t(2)}, ..., \frac{\partial f(\vec{\theta}_t}{\theta_t(2)}\right)^T$
This derivative vector is the gradient of f with respect to $\vec{\theta}_t$. We turn now to the case in which the target output, v_t of the *t*th training example is not the true value $V^*(s_t)$, but some approximation of it. For example, v_t might be a noise-corrupted version of $V^*(s_t)$ or it might be one of the backed-up values mentioned in the previous section. In such cases we cannot perform the exact update because $V^*(s_t)$ is unknown, but we can approximate it by substituting v_t in place of $V^*(s_t)$. This yields the general gradient-descent method for state-value prediction as

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\big[v_t - V_t(s_t)\big]\nabla_{\vec{\theta}_t}V_t(s_t) \tag{12}$$

If v_t is an unbiased estimate, that is if $Ev_t = V^*(s_t)$ for each *t*, then $\vec{\theta}$ is guaranteed to converge to local optimum under stochastic gradient descent.
Let R_t denotes the return following each state, s_t, because the true value of a state is the expected value of the return following it. In our case it is Temporal difference algorithm, The gradient descent form of $TD(\lambda)$ uses the λ-return, $v_t = R_t^\lambda$ as its approximation to $V^*(s_t)$, yielding the forward-view update:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\big[R_t^\lambda - V_t(s_t)\big]\nabla_{\vec{\theta}_t}V_t(s_t) \tag{13}$$

Unfortunately, our approach is TD(0) and λ is less than 1($\lambda = 0$). Therefore R_t^λ is not an unbiased estimate of $V^* s_t$ and thus this does not converge to local optimum.

3) Approach 3: Vasilis et. al (see Reference [1]) in their article on 'Convergence of reinforcement learning with general function approximation' have proved the non-convergence of Temporal Difference with non-linear functional approximation.

D. Monitor Learning Performance of Neural Net

1) Approach 1: If the robot is learning, it falls under one of the two cases:

- If the specific state action pair leads to increase in the reward then the Qvalue at t+1 should be greater than Qvalue at t. then $e(s) = Q(s_{t+1}) - Q(s_t)$ is positive. If the Neural network is learning then the plot of e(s) vs iterations should decrease slowly and it should approach towards zero, when the number of iteration is large. If the robot is not learning then the plot of e(s) vs iterations in neural network remains noisy and oscillating.
- If the specific state action pair leads to decrease in the reward then the Qvalue at t+1 should be lesser than Qvalue at t. then $e(s) = Q(s_{t+1}) - Q(s_t)$ is negative. If the Neural network is learning then the plot of e(s) vs iterations should increase from negative direction slowly and it should approach towards zero, when the number of iteration is large. If the robot is not learning then the plot of e(s) vs iterations in neural network remains noisy and oscillating.
- The above two cases can be used to check the learning of the robot. If there is no specific trend as mentioned above, it means that the robot is not learning.

2) Approach 2: Another way to check the performance is to find the optimal value of all the state-action pair offline by iterating it over many times and compute Root mean square of neural network while it is learning with the value function it approximates and the offline value function we have.

VIII. Answers to Question 3

A. 3a

1) Practical Issues surrounding the application of RL & BP:

- The reinforcement learning can perform well only within the operating range of how it is learnt. For example: if the reinforcement learning robot is learnt against spin bot. while testing, it is not guaranteed that it will perform well versus other robots. So, the reinforcement learning robot has to be separately trained against each and every robot. So no generalized trained robot can be achieved by reinforcement learning.
- In this case the static version of the Look up table was trained to find the best parameter of the neural network and the neural network of reinforcement learning robot was initialized with those hyper parameters. If the states are high, then storing of LUT would have been a difficult task and off line training of neural network would have been a difficult task. In such case the neural network

has to be initialized with random weights and with other hyper parameters, tuning of this kind of neural network would be a difficult task and there is no closed loop solutions for that.
- If the robot has to perform well we will have to have more number of states and actions and the error while approximating the larger state action pairs Q values would be very high and neural networks with higher number of hidden neurons or hidden layers have to be introduced which eventually lead to higher training time.
- The convergence of BP to global optimum is not guaranteed.
- The learning of reinforcement learning with neural networks takes a long time.

2) Improve the performance of the robot:
- The lowest Root mean square error during cross validation with cross validation test set was approximately around 0.5. It would have been much more efficient if it is near zero. Neural network with higher number of hidden layers (Deep Neural Network) can be used for functional approximation. This learning method is efficient because it involves many nodes and parameters and can fit the data very well.
- Here the training of neural network is done after each update of Q, that is here only one value of training samples are shown to neural network at a time. The training performance of the neural network can be improved by showing historical batches of training sets to neural network while training. This can improve the efficiency of the model.
- Instead of constant step size variable step size can be used in the back propagation algorithm.
- Separate neural networks for each actions will reduce the overall root mean square

3) Addressing Convergence Problem:
- As discussed earlier, instead of using constant step size, variable step size can be used
- While selecting actions, it should be noticed that each actions are unique and differ widely in their characteristics with respect to others, this make the Q-value to differ in a wide range when compared to other values and the neural network can approximate such value functions easily.
- The activation function used in the neural network is bipolar sigmoid function, any other activation function like tanh function can be used.
- The bridge algorithm given in [1] (see Reference [1]) can be applied

4) Advice while applying RL with Neural Network for other practical applications: It is good to use pre-trained neural network (as used in this project)in the reinforcement learning as the rate of convergence will be faster. If the neural network were to be initialized with random weights the network will either diverge or will take a long time to converge. If the practical application can be modeled it will be easier and safer to perform training of reinforcement learning. It is recommended to perform training in a simulator rather than training and testing on line. While considering states and actions, it should be noted that if the state is observable (as observability in control theory) and similarly it should be noticed that if the actions are controllable(as controllability in control theory). Usually practical problem involves many correlated states, in such case only independent states need to be selected. While selecting actions, select the actions that affects the output to a maximum extent.

B. 3b-Theory Question

1) Concerns: The main concern is that Reinforcement learning cannot be trained by delivering anesthetic to a patient under going surgery because at the beginning the RL control tries to explore and learning takes place gradually. In such case it would be a serious problem for the patient if RL is exploring. Moreover training of RL requires lot of training samples (and it requires more number of patients to do that). This is dangerous to the patient as well as leads to loss of resources (anesthetic).

2) Potential variation that could alleviate the concerns: If the anesthetic delivery system can be simulated then reinforcement learning can be applied on the simulator and the optimal control moves can be learnt. Controllers like Model Predictive Control(MPC) are used in drug delivery to the patients and Reinforcement learning algorithm can be applied on the simulated MPC to learn the optimal control moves. (see reference [2])

IX. Future Work

In this work Neural Network with one hidden layer was used for function approximation. Later, deep leep neural network with 3 hidden layers will be applied to improve the overall efficiency of root mean square error. Since my current master's project focuses on process control & optimization, learning controller which can learn when a controller is controlling will be developed. For learning, model predictive controller(MPC) has been planned to use. The controller will learn while MPC is controlling the plant with the help of reinforcement learning and deep neural network.

X. Conclusion

Thus Reinforcement learning has been applied on the robot tank and the results were discussed.

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] Richard Sutton, Andrew G. Barto *Reinforcement Learning: An Introduction* MIT Press, Cambridge, MA, 1998

[4] Christos Stergiou, Dimitrios Siganos, *Introduction to Neural Networks*, 1996

[5] IBM Robocode (http://robocode.sourceforge.net/)