A comparison of various reinforcement learning algorithms to solve racetrack problem

Vaibhav Mohan

June 20, 2014

Abstract

Reinforcement learning is an area of machine learning that is concerned with how an agent should take actions in an environment so that it can maximize its cumulative rewards. Unlike most machine learning techniques, the learner is not told which action it should take but instead must discover which action maximizes its cumulative reward by trying them. Trial-and-error search and delayed rewards are the two important distinguishing features of reinforcement learning.

The racetrack problem is a standard control problem and can be solved using various reinforcement learning techniques. In this paper, the implementation of value iteration algorithm, Q-learning algorithm, and a variant of Q-learning algorithm that uses function approximator is done in order to solve the racetrack problem. The algorithms are compared based on the score they obtain in solving the problem on various maps by varying various parameters such as convergence tolerance, number of iterations etc. The time taken for solving this problem by each algorithm on various maps corresponding to each setting is also compared in this paper. It is proven from the experiments that performance of all the algorithms is almost comparable to each other.

1 Introduction

The reinforcement learning is generally used when we need on-line performance which involves finding a balance between exploration of uncharted

space and exploitation of current knowledge. The exploration vs. exploitation trade-off in reinforcement learning has been most thoroughly studied through the multi-armed bandit problem and in finite Markov Decision Processes (MDPs)[7, 4, 8]. This problem arises in various fields ranging from operation research to control theory to simulation based optimizations. There are wide varieties of optimization problems in machine learning domain, all of which cannot be solved using one technique[11]. Therefore, for proving that the results which we are getting from one kind of technique is good enough for us makes it indispensable that we compare the results with other techniques for the given problem. Whilst doing this, we come across the various performance aspects of the algorithm i.e. where it would fail and where it can do remarkably well in solving the problem. Solving these kinds of problems involves making decision in stochastic environment[7, 6].

MDPs provide a mathematical framework for modeling these kind of problems. They are useful for studying a wide range of optimization problems that are solved using dynamic programming and reinforcement learning[2]. More precisely, a Markov Decision Process is a discrete time stochastic control process. The process is in some state s and the decision maker chooses any action a that is available in state s at each time step. The process then moves in a new state s' and gives reward $R_a(s, s')$[1].

One of the most popular technique for solving this kind of problem in reinforcement Learning domain is Value Iteration. It uses MDPs to mathematically model the problem. The value iteration algorithm is a simple iterative algorithm that finds an optimal policy for deciding the action that should be taken from each state. The value iteration algorithm converges to correct mapping between states in the world and their expected values. The problem in the algorithm is that it is not obvious when the value iteration algorithm must be terminated. One important result bounds the performance of the current greedy policy as a function of the Bellman residual of the current value function[10]. It states that if the maximum difference between two successive value functions is less than ϵ, then the value of the greedy policy differs from the value function of the optimal policy by no more than $2\epsilon\gamma/(1-\gamma)$ at any state. This provides an effective stopping criteria for the algorithm. Puterman discusses another stopping criterion based on the span semi-norm which may result in earlier termination[5].

The another technique used in solving this problem is using Q-learning algorithm. This technique learns an action-value function that gives the expected utility of taking a given action in a given state and following a fixed

policy thereafter. The advantage that Q-learning algorithm have over value iteration algorithm is that it can compare the expected utility of the available actions without requiring a model of the environment. The disadvantage with these kind of algorithms is that as the size of state space increases, the time to convergence and time per iteration increases rapidly. One way to handle this problem is to use function approximator[7, 4, 3, 8]. The Q-function is represented in form of a function rather than a lookup table and can be approximated using standard function approximating algorithms such as linear regression or neural networks[9]. We have implemented all the three variants in this paper. A comparison of all the three variants is also done in this paper.

The remainder of paper is organized as follows. Section 2 defines the racetrack problem and the various algorithms used in solving that problem. Section 3 comprises of details of the parameters chosen pertaining to implemented algorithms and experimental methods used with them. Section 4 presents the results obtained after performing the experiment. Section 5 talks about the interpretation of the data which we got from the experiment. Finally, Section 6 concludes this work.

2 The Racetrack Problem

The racetrack problem is a standard control problem. The goal is to control the movement of a race car along a pre-defined racetrack so that the racer can get to finish line from starting line in minimum amount of time. At each time step, the state of the racer is given by for variables viz. $\boldsymbol{X_t}$, $\boldsymbol{Y_t}$, $\boldsymbol{V_{xt}}$, and $\boldsymbol{V_{y_t}}$. $\boldsymbol{X_t}$ and $\boldsymbol{Y_t}$ are the $\boldsymbol{x}$ and $\boldsymbol{y}$ component of the displacement vector at time $\boldsymbol{t}$. Similarly $\boldsymbol{V_{xt}}$**, and** $\boldsymbol{V_{y_t}}$ are the $\boldsymbol{x}$ and $\boldsymbol{y}$ component of the velocity vector at time $\boldsymbol{t}$. The control variables are $\boldsymbol{a_x}$ and $\boldsymbol{a_y}$ that represent $\boldsymbol{x}$ and $\boldsymbol{y}$ component of the acceleration vector to be applied at current time step. We use a discrete, static world so that all the positions are integers. The

kinematics equations of this system is given by:

$$\boldsymbol{X_t} \equiv \text{x position} \tag{1}$$
$$\boldsymbol{Y_t} \equiv \text{y position} \tag{2}$$
$$\boldsymbol{V_{xt}} = \boldsymbol{X_t} - \boldsymbol{X_{t-1}} \equiv \text{x speed} \tag{3}$$
$$\boldsymbol{V_{y_t}} = \boldsymbol{Y_t} - \boldsymbol{Y_{t-1}} \equiv \text{y speed} \tag{4}$$
$$\boldsymbol{A_{xt}} = \boldsymbol{V_{xt}} - \boldsymbol{V_{xt-1}} \equiv \text{x acceleration} \tag{5}$$
$$\boldsymbol{A_{y_t}} = \boldsymbol{V_{y_t}} - \boldsymbol{V_{y_{t-1}}} \equiv \text{y acceleration} \tag{6}$$

The agent can only control $\boldsymbol{a_x}$ and $\boldsymbol{a_y}$ at any given time step and thus uses these control variable to influence its state. The values that acceleration variable can be assigned are $\mathbf{-1, 0}$ and $\mathbf{1}$. The maximum speed of a racer at any given time is $(V_{xt}, V_{y_t}) \in (\pm 5, \pm 5)$. If the agent tries to accelerate beyond these limits, the speed of the agent remains unchanged.

At each time step, velocity is updated first using the acceleration followed by update of position. This helps us in maintaining integer value for all our parameters at all times thereby allowing us to have a discrete state space.

Each time we try to accelerate, we have 10% probability that the attempt will fail thereby making this system non-deterministic. We also have an additional requirement that the agent must stay on the track all the times i.e. crashing into the wall is bad. We have two different variants of how bad a crash is. The first variant places the car to nearest position on the track to the place where it crashed while the second variant resets the car to start position making its velocity equal to zero. We have experimented with both the variants in this paper.

The various algorithms that are implemented in this paper in order to solve the given problem are discussed in the subsections given below.

2.1 Value Iteration

The value iteration algorithm is used for calculating an optimal policy so that the agent can choose actions based on that policy. This policy is computed by calculating the utility of each state and then using the state utilities to select an optimal action in each state.

The Bellman equation

The Bellman equation is used for calculating the utility of being in a state. The utility of state is the immediate award for that state plus the expected discounted utility of next state[7, 4] and is given by the equation:

$$U(s) = R(s) + \gamma.max_{a \in A(s)} \sum_{a'} P(s'|s,a).U(s')$$

where s is the present state, s' is the next state, a is the action which we choose from state s, γ is the discount factor, $R(s)$ is the reward function (which gives the reward an agent gets in each state), and $U(s)$ gives the utility of being in state s and $P(s'|s,a)$ is the probability of going into state s' from state s by choosing action a.

The Value Iteration Algorithm

The value iteration algorithm for calculating utilities of states is given in figure 1. The bellman equation is used in the value iteration algorithm for solving MDPs. We have one Bellman equation for each state. So if we have n states, we have n unknowns and thus we have n simultaneous equations. These equations are solved to obtain the utilities of the states.

We initialize utility of all state to zero. Then we update the utility value of each state according to the bellman equation in every iteration. We keep updating the utility values until we reach an equilibrium i.e. we do not see change in the utility values for two consecutive iterations. Once the algorithm converges, we use those utility values to choose an action in each state thereby making agent to reach finish line from starting line.

2.2 Q-Learning Algorithm

The Q-Learning algorithm comes under the category of active reinforcement learning and hence decides what action to take in each state. The difference between this algorithm and value iteration algorithm is that this algorithm compares the expected utility of the available actions and hence does not requires a model of the environment.

```
function VALUE-ITERATION(mdp, ϵ)
    inputs: mdp, an MDP with states S, actions A(s), transition model P(s'|s, a),
     rewards R(s), discount γ, ϵ the maximum error allowed in the utility of any state.

    local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

    repeat
        U = U'
        δ = 0
        for each state s ∈ S do
            U'[s] = R(s) + γ.max_{a∈A(s)} Σ_{s'} P(s'|s, a).U[s']
            if |U'[s] − U[s]| > δ then
                δ = |U'[s] − U[s]|
            end if
        end for

    until δ < ϵ.(1 − γ)/γ
return U
end function
```

Figure 1: The value iteration algorithm for calculating utilities of states [7].

Exploration

The problem with value iteration agent is that once it learns the policy, it then sticks to it and thus takes action based on that in the actual environment. The policy learned by agent is not true optimal policy. This is because once it finds the route, after experimenting with minor variations, it sticks to it. It never learns the utilities of other state and thus never finds the optimal route. This happens because the learned model is not the same as the true environment i.e. the policy which is optimal in learned model can be suboptimal in true environment. Since the agent does not knows the true environment, it cannot compute the optimal action for the true environment.

In order to overcome this problem, an agent must make a trade-off between exploitation to maximize its reward-as reflected in its current utility estimates-and exploration to maximize its long-term well-being[7, 4]. Pure exploitation might make agent to get stuck in rut while pure exploration to improve the knowledge of agent is of no use as it is never used in deciding the action.

This type of scheme must be greedy in the limit of infinite exploration (GLIE). A GLIE scheme must try each action in each state several times

so that it does not miss optimal action due to bad series of outcomes. In this implementation, we give some weight to the the actions that are not tried by agent while avoiding the action that has low utility value. We use exploration function in our implementation in order to accomplish this. This function determines the trade-off between greediness and curiosity of agent. The function used in this implementation is as follows:

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where u is the expected reward, n is the record of how frequently each action has been explored from each state,R^+ is an optimistic estimate of the best possible reward obtainable in any state and N_e is a fixed parameter. This forces agent to explore each action-state pair at least N_e times.

The Q-Learning Algorithm

The Q-learning algorithm learns an action-utility representation instead of learning utilities. The value of doing action $'a'$ in state $'s'$ is given by the Q-function, $Q(s,a)$. the relation between Q-values and utility values is given by:

$$U(s) = max_a Q(s,a)$$

Q-learning is a model-free method and hence it does not uses transition function. The update equation for Q-learning is given by:

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma(max_{a'} Q(s',a') - Q(s,a)))$$

where a' is the action actually taken in state s'. The Q-learning algorithm is given in figure 2.

The agent percepts the current state s' and reward r' and returns an action based on the Q-table that stores action values indexed by state and action. The update to the Q-values are done according to the aforementioned update equation. We keep updating the Q-values until we reach an equilibrium i.e. we do not see change in the Q-values for two consecutive iterations. Once the algorithm converges, we use those Q-values to choose an action in each state thereby making agent to reach finish line from starting line.

```
function Q-LEARNING-AGENT(percept)
    inputs: percept, a percept indicating the current state s' and reward signal r'

    persistent: Q, a table of action values indexed by state and action, initially zero
    N_{s,a}, a table of frequencies for state-action pairs, initially zero
    s, a, r, the previous state, action and reward, initially null

    if TERMINAL?(s) then
        Q[s, None] ← r'
    end if
    if s is not null then
        increment N_{s,a}[s, a]
        Q[s, a] ← Q[s, a] + α(r + γ.max_{a'} Q[s', a'] − Q[s, a])
    end if
    s, a, r ← s', argmax_{a'} f(Q[s', a'], N_{s,a}[s', a']), r'
return a
end function
```

Figure 2: The Q-Learning algorithm that learns the value $Q(s, a)$ of each action in each situation [7].

The Q-function Approximation Algorithm

The problem with this approach is that as the size of state space increases, the time to converge and time per iteration increases rapidly making this problem computationally intractable. In order to overcome this problem, we use function approximation algorithm in order to estimate the Q-function. The representation is approximate because it is not necessary that the true utility function or Q-function can be represented in the form which we have chosen.

A reinforcement learning algorithm learns values of various parameters such that the evaluation function $\hat{U}_\theta$ approximates the true utility function. This allows the learning agent to generalize from states it has visited to states it has not visited. One of the disadvantage of using function approximation is that the used function might fail to approximate the true utility function in the chosen hypothesis space.

In this implementation, we have represented Q-function as a weighted linear function of a set of features x, y, V_x, and V_y where x and y are x and y component of displacement vector and V_x and V_y are x and y component of velocity vector. The function used in this implementation is given below:

$$\hat{U_\theta}(x, y, V_x, V_y) = \theta_0 + \theta_1.x + \theta_2.y + \theta_3.V_x + \theta_4.V_y$$

Using collection of trials, we obtain a set of values of $\hat{U_\theta}(x, y, V_x, V_y)$ and minimize the least square error in finding the best fit using standard linear regression. We update the parameters after each trial using Widrow-Hoff rule for online least-squares. The update rules used for updating parameters in Q-function is as follows:

$$\theta_0 \leftarrow \theta_0 + \alpha[R(s) + \gamma(max_{a'}\hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)], \quad (7)$$

$$\theta_1 \leftarrow \theta_1 + \alpha[R(s) + \gamma(max_{a'}\hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)].x, \quad (8)$$

$$\theta_2 \leftarrow \theta_2 + \alpha[R(s) + \gamma(max_{a'}\hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)].y, \quad (9)$$

$$\theta_3 \leftarrow \theta_3 + \alpha[R(s) + \gamma(max_{a'}\hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)].V_x, \quad (10)$$

$$\theta_4 \leftarrow \theta_4 + \alpha[R(s) + \gamma(max_{a'}\hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)].V_y \quad (11)$$

3 Algorithms and Experimental Methods

Value Iteration Algorithm

There were various parameters to choose in case of value iteration algorithm like what should be the values of ϵ, discount factor etc. We chose the value of epsilon as 0.000000001 as more smaller the value of ϵ is, the more we are close to convergence. Also the value of discount factor was chosen to be 0.5. This is because it gives a good balance between rewards in the distant future and additive rewards. The starting position of an agent is chosen at random from one of the existing starting squares on map.

The results of comparison of algorithm is given in the next section. We compared the number of iterations and time taken to reach the convergence on various datasets. We also had a parameter for defining how bad a crash is i.e. hard crash resets the position of agent to starting line and makes its velocity zero while soft crash resets the agent to nearest cell where it collides with wall. We experiment with both the settings and their result is also compared. A comparison of time taken in training the algorithms on various datasets is also done.

Q-Learning Algorithm

There were various parameters tunable parameters such as ϵ, discount factor, minimum number of of times the agent will explore a given state-action pair before giving up on it, learning factor, whether to use function approximator or not etc. We chose the value of epsilon as 0.0001 as more smaller the value of ϵ is, the more we are close to convergence. Also the value of discount factor was chosen to be 0.99 as it favors the actions of agent to have more additive rewards. The maximum exploration count was set to 1 so that the agent tries each action at least once before giving up on it. This way we will not miss the optimal solution. The learning factor (α) was chosen to be 0.5. This is because if the learning factor is close to zero, it might get good results but it will take too long to converge. On the other hand if the learning factor is close to one, the agent might converge faster and thus might miss the optimal results. The starting position of an agent is chosen at random from one of the existing starting squares on map.

We compared the number of iterations and time taken to reach the convergence on various datasets. We also had a parameter for defining how bad

a crash is i.e. hard crash resets the position of agent to starting line and makes its velocity zero while soft crash resets the agent to nearest cell where it collides with wall. We experiment with both the settings and their result is also compared. We had an option whether to use function approximator for approximating Q-function or not. The experiment was done with each variant and their result was compared. We also compared how well an agent does as we increase the number of iterations to train the agent. A comparison of time taken in training the algorithms on various datasets is also done.

Data Sets

The datasets used in this problem are ASCII representations of racetracks. The first line in the data files contains number of rows and columns in a racetrack as a comma delimited pair. the rest of the file is a map with specified dimension with one character at each point. the characters present in the map are $'S'$, which represents that it is one of the start positions, $'F'$, which represents that it is one of the finish positions, $'.'$, which represents that it is open racetrack where if we apply an action, then there is 10% probability that the action will fail and there will be no change in the velocity of agent, $','$, which represents that it is open racetrack with slippery terrain where if we apply an action, then there is 60% probability that the action will fail and there will be no change in the velocity of agent, and finally $'\#'$, which represents that it is an obstacle.

We used four datasets of this kind to test our implementation. The racetracks taken were of O,L and R shape. All the tracks had only simple terrain except O-track. We had two variants of O-track. One had normal terrain while the other map contained slippery terrain in addition to normal terrain. The comparison of performance of all the algorithms were done on both type of datasets.

4 Results

The graph in figure 3 shows scores obtained on the various racetracks as we vary the discount factor (γ) in case of value iteration algorithm. The convergence tolerance was set to 0.0000000001. This value was chosen as having small value for convergence tolerance makes agent do well on given racetrack.

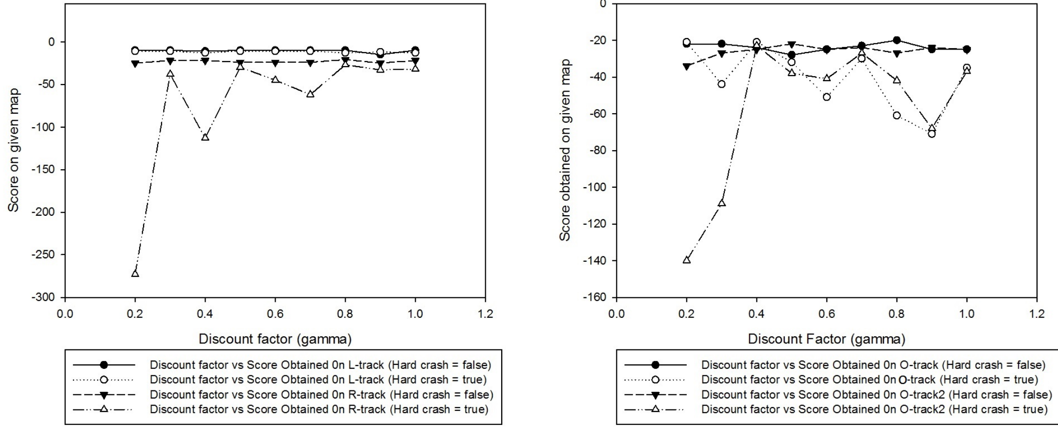

Figure 3: The Value Iteration Algorithm - Graph showing how score obtained on various tracks varies as we vary the discount factor (γ).

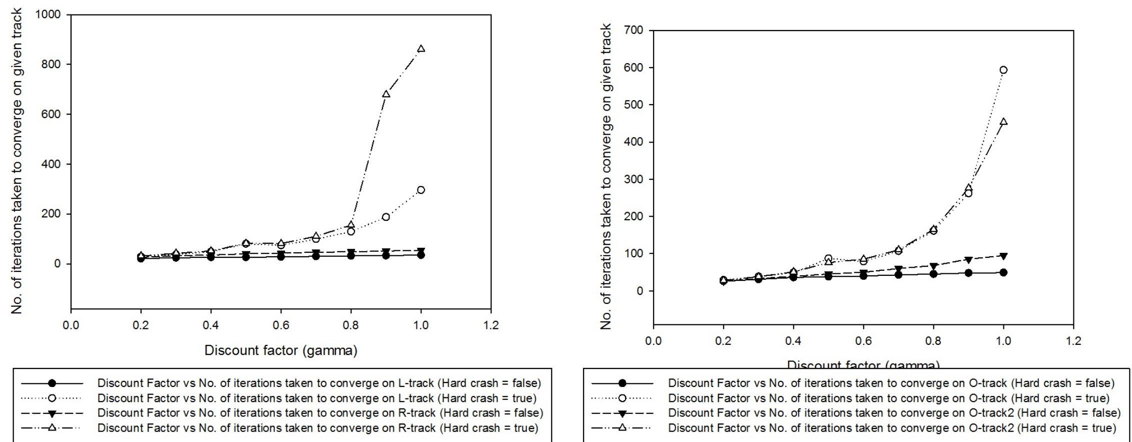

Figure 4: The Value Iteration Algorithm - Graph showing how number of iterations taken to converge on various tracks varies as we vary the discount factor (γ).

Data file	hard crashing?	No. of iter. taken to converge	Time	Score
L-track	no	35	33 sec.	-10
L-track	yes	80	175 sec	-13
R-track	no	54	45 sec	-21
R-track	yes	82	701 sec	-30
O-track	no	49	42 sec	-24
O-track	yes	87	593 sec	-32
O-track2	no	95	65 sec	-27
O-track2	yes	76	521 sec	-38

Table 1: The performance of Value Iteration Algorithm on various racetracks.

Figure 4 shows graph of number of iterations taken by algorithm to converge on a particular racetrack versus discount factor (γ) in case of value iteration algorithm. Here also the convergence tolerance was set to 0.0000000001.

Table 1 shows the performance of value iteration algorithm on various racetracks. We experimented with soft crash as well as hard crash and recorded number of iterations taken to converge, time taken in training the agent and score obtained by the agent on each racetrack.

The graph in figure 5 and figure 6 shows scores obtained on the various racetracks as we vary the number of iterations used to train the agent in case of Q-learning algorithm that uses lookup table and Q-learning agent that uses function approximator respectively. The convergence tolerance was set to 0.0000000001. This value was chosen as having small value for convergence tolerance makes agent do well on given racetracks. Also the value of discount factor was chosen to be 0.99 as it favors the actions of agent to have more additive rewards. The maximum exploration count was set to 1 so that the agent tries each action at least once before giving up on it. The learning factor (α) was chosen to be 0.5. This is because if the learning factor is close to zero, it might get good results but it will take too long to converge. On the other hand if the learning factor is close to one, the agent might converge faster and thus might miss the optimal results. The starting position of an agent is chosen at random from one of the existing starting squares on map.

Table 2 and table 3 shows results related to Q-learning algorithm. We had two variants of Q-learning algorithm. The first variant uses a lookup table for Q-values and its performance on various racetracks is shown in table 2 while

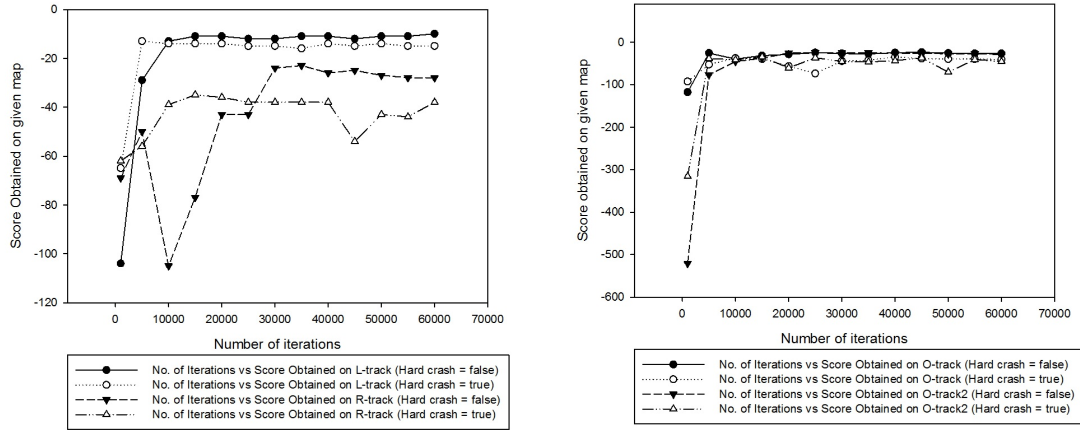

Figure 5: The Q-Learning algorithm without using function approximator - Graph showing scores obtained on various racetracks vs number of iterations

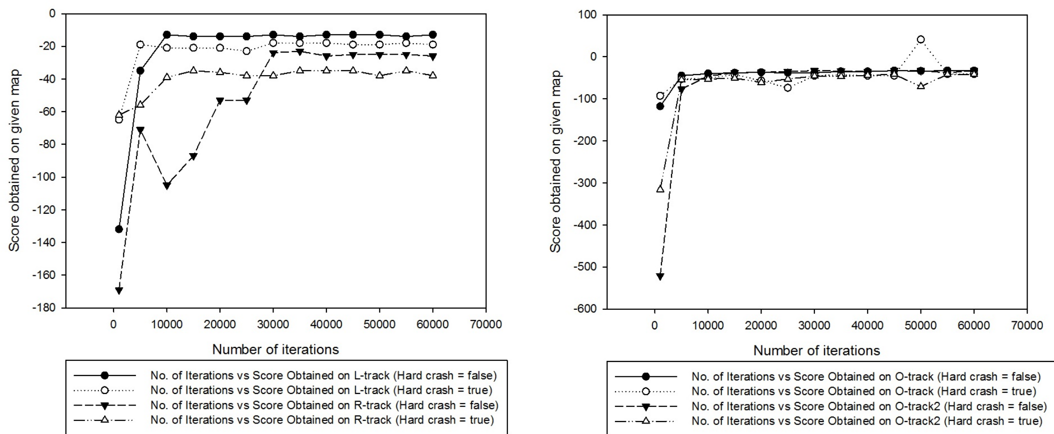

Figure 6: The Q-Learning algorithm using function approximator - Graph showing scores obtained on various racetracks vs number of iterations

Data file	hard crashing?	No. of iter. taken to converge	Time	Score
L-track	no	76911	2128 sec.	-11
L-track	yes	17625	2271 sec	-15
R-track	no	185784	4159 sec	-28
R-track	yes	159531	3603 sec	-38
O-track	no	218019	6769 sec	-25
O-track	yes	237413	7593 sec	-33
O-track2	no	167479	6512 sec	-35
O-track2	yes	209701	7521 sec	-33

Table 2: The performance of Q-Learning Algorithm using lookup table for Q-values on various racetracks.

Data file	hard crashing?	No. of iter. taken to converge	Time	Score
L-track	no	52541	1591 sec.	-13
L-track	yes	23769	2132 sec	-18
R-track	no	192783	3395 sec	-23
R-track	yes	115231	2819 sec	-35
O-track	no	233891	4563 sec	-32
O-track	yes	216567	6564 sec	-41
O-track2	no	143125	5593 sec	-33
O-track2	yes	255191	6147 sec	-42

Table 3: The performance of Q-Learning Algorithm using function approximator on various racetracks.

the second variant uses function approximator to approximate the values of Q-function weight and uses them to compute Q-value. The performance of second variant is shown in table 3. We experimented with soft crash as well as hard crash using both variants and recorded number of iterations taken to converge, time taken in training the agent and score obtained by the agent on each racetrack.

5 Discussion

L-track data file

Figure 3 shows the results of experimentation with value iteration algorithm by varying the discount factor(γ) on L-track. It can be seen from the figure that varying γ does not have any effect on the score obtained by agent on L-track. Even if we set use the hard crashing variant, we do not see any significant change in the score obtained. This might be happening because the map is easy to learn by the agent and agent converges to optimal path in both the cases.

In figure 4, we can see that as we increase *gamma*, the number of iterations taken to converge also increases in case of value iteration algorithm. Increase is not significant in case of soft crashing variant but hard crashing variant takes larger number of iterations to converge to optimal Q-values. It can also be seen from data given in table 1. Also the score is better in case of soft crashing variant. This happens because soft crash resets the car position to nearest cell where agent crashes while hard crash resets the car to start position.

Figure 5 and 6 shows the results of experimentation with Q-learning algorithm using lookup table for Q-values and Q-learning algorithm that uses function approximator to calculate Q-values on L-track. The graph shows scores obtained by both the algorithms using hard crashing as well as soft crashing as we increase the number of iterations. It can be seen from the obtained results that as we increase the number of iterations, the performance of agent increases and after approximately 50000 iterations, the score obtained on L-track becomes constant in case of both the algorithms. Here the soft crashing variant does better as compared to hard crashing variant in case of both the algorithms because soft crash resets the car position to nearest cell where agent crashes while hard crash resets the car to start position.

Table 2 and table 3 shows scores obtained by Q-learning algorithm using lookup table for Q-values and Q-learning algorithm that uses function approximator to calculate Q-values after it fully converges. It can be seen that the algorithm using lookup table performs better as compared to the algorithm using function approximator. This happens because the lookup table contains exact values for each state while the algorithm using function approximator approximates Q-values and hence would not be getting the optimal Q-values. It can also be seen that performance of hard crashing variant is bad as compared to the performance of soft crashing variant in case of both algorithms. It can also be seen that hard crashing version takes lesser number of iterations to converge as compared to soft crashing version.

R-track data file

Figure 3 shows the results of experimentation with value iteration algorithm by varying the discount factor(γ) on R-track. It can be seen from the figure that varying γ does not have any effect on the score obtained by agent using soft crashing variant on R-track. However if we use the hard crashing variant, we can see that the obtained score increases sometimes and then again decreases. The change is minimal if γ is set to 0.5.

In figure 4, we can see that as we increase *gamma*, the number of iterations taken to converge also increases in case of value iteration algorithm. Increase is not significant in case of soft crashing variant but hard crashing variant takes larger number of iterations to converge to optimal Q-values. It can also be seen from data given in table 1. Also the score is better in case of soft crashing variant. This happens because soft crash resets the car position to nearest cell where agent crashes while hard crash resets the car to start position.

Figure 5 and 6 shows the results of experimentation with Q-learning algorithm using lookup table for Q-values and Q-learning algorithm that uses function approximator to calculate Q-values on R-track. The graph shows scores obtained by both the algorithms using hard crashing as well as soft crashing as we increase the number of iterations. It can be seen from the obtained results that initially we get random values of scores but as we increase the number of iterations, we see an improvement in the score obtained in case of both the algorithms that corroborates the fact that we need to increase the number of iterations in order to obtain better score. Here the soft crashing variant does better as compared to hard crashing variant in case of

both the algorithms because soft crash resets the car position to nearest cell where agent crashes while hard crash resets the car to start position.

Table 2 and table 3 shows scores obtained by Q-learning algorithm using lookup table for Q-values and Q-learning algorithm that uses function approximator to calculate Q-values after it fully converges. The results obtained on this track is similar to the results obtained on L-track as it can be seen from both the tables.

O-track and O-track2 data file

Figure 3 shows the results of experimentation with value iteration algorithm by varying the discount factor(γ) on O-track and O-track2. It can be seen from the figure that as the value of γ increases, the score obtained becomes better. This happens on both tracks. It can also be seen that soft crashing variant does not show any significant change in scores obtained as we vary γ, but we can see a large variation in the scores obtained in case of hard crashing variants on both the tracks. However it can be seen that the changes are not too significant if $\gamma = 0.5$. Therefore we used this value of γ in our experiment.

In figure 4, we can see that as we increase *gamma*, the number of iterations taken to converge also increases in case of value iteration algorithm. Increase is not significant in case of soft crashing variant but hard crashing variant takes larger number of iterations to converge to optimal Q-values on both the tracks. It can also be seen from data given in table 1. Also the score is better in case of soft crashing variant. This happens because soft crash resets the car position to nearest cell where agent crashes while hard crash resets the car to start position.

Figure 5 and 6 shows the results of experimentation with Q-learning algorithm using lookup table for Q-values and Q-learning algorithm that uses function approximator to calculate Q-values on O-track as well as O-track2. The graph shows scores obtained by both the algorithms using hard crashing as well as soft crashing as we increase the number of iterations.It can be seen from the obtained results that as we increase the number of iterations, the performance of agent increases and after approximately 50000 iterations, the score obtained on L-track becomes constant in case of both the algorithms. Here the soft crashing variant does better as compared to hard crashing variant in case of both the algorithms because soft crash resets the car position to nearest cell where agent crashes while hard crash resets the car to start position.

Table 2 and table 3 shows scores obtained by Q-learning algorithm using lookup table for Q-values and Q-learning algorithm that uses function approximator to calculate Q-values after it fully converges on both the tracks. The results obtained on these two tracks are similar to the results obtained on L-track as it can be seen from both the tables.

Comparison of the algorithms

From table 1,2, and 3; it can be seen that value iteration algorithm takes really less time as compared to other two variants of Q-learning. It can also be seen that the version of Q-learning algorithm that uses lookup table takes less time to train as compared to the Q-learning algorithm that uses function approximator. This is because approximating the Q-function takes lots of time. However using Q-learning algorithm that uses function approximator makes more sense because if the size of state space will be increased, the version that uses lookup table might fail as we might run out of memory and hence would be unable to find the solution.

Table 1,2, and 3 also shows that value iteration algorithm takes very less number of iterations to find solution as compared to both the implementations of Q-learning algorithm. It can also be seen that the version of Q-learning algorithm that uses lookup table takes less number of iterations to converge as compared to the Q-learning algorithm that uses function approximator.

6 Conclusions

It have been shown that value iteration algorithm performs faster and better as compared to both the variants of Q-learning algorithm. However in more realistic settings, the value iteration might not be able to find the optimal solution while the Q-learning algorithm would perform better. It is also shown that the implementation of Q-learning algorithm that uses function approximator, though slow, would be better in long run as compared to the version that uses lookup table. This would happen because as the state space grows, the memory requirement for storing various states would also grow exponentially rendering the lookup table version useless.

The limitation of our approach was that we used linear classifier in order to approximate the Q-function. There might be a chance that the Q-function

actually would be a non-linear function in which case the performance of our algorithm would be terrible. In order to overcome this problem, neural networks can be used to approximate the Q-function which would yield good results.

References

[1] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 1957.

[2] R.E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] Lucian Busoniu and Robert Babuska. *Reinforcement Learning and Dynamic Programming using Function Approximators*. Taylor and Francis CRC Press, 2010.

[4] T. M. Mitchell. *Machine Learning*. McGraw hill, 1997.

[5] Martin L. Puterman. *Markov Decision Processes–Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[6] P. E. Hart R. O. Duda and D. G. Stork. *Pattern Classification, 2nd Ed.* Wiley interscience, 2001.

[7] S. Russell and P. Norvig. *Artificial Intelligence : A Modern Approach, 3rd Ed.* Prentice Hall, 2009.

[8] Christopher J.C.H. Watkins and Peter Dayan. Technical note: Q-learning. *Springer : Machine Learning*, 8(4):279–292, 1992.

[9] Ronald J. Williams. A class of gradient-estimating algorithms for reinforcement learning in neural networks. *Proceedings of the IEEE First International Conference on Neural Networks*, 1987.

[10] Ronald J. Williams and Leemon C. Baird. Tight performance bounds on greedy policies based on imperfect value functions. *Technical Report NU-CCS-93-14, Northeastern University, College of Computer Science, Boston, MA*, 1993.

[11] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.