Reinforcement Learning Coursework

Mohammad Raihanul Bashar, Rayhan Sardar Tipu, 1320454@iub.edu.bd, 1330418@iub.edu.bd
Department of Computer Science, School of Engineering & Computer Science, Independent University, Bangladesh

I. Introduction (*Heading 1*)

In this article, we analyze the basic reinforcement learning algorithms in three discrete models. We implement value and policy iteration algorithms. We compare performance between SARSA and Q-Leaning.

II. Questions

A. *Value Iteration (Heading 2)*

Value iteration is a method of computing an optimal MDP policy and its value. In reinforcement learning, value iteration is about finding the optimal policy π using an iterative approach of the Bellman optimality backup. Value iteration starts at the "end" and then works backward, refining an estimate of either Q^* or V^*. There is really no end, so it uses an arbitrary end point. Let V_k be the value function assuming there are k stages to go, and let Q_k be the Q-function assuming there are k stages to go. These can be defined recursively. Value iteration starts with an arbitrary function V_0 and uses the following equations to get the functions for $k+1$ stages to go from the functions for k stages to go:

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V_k(s')) \quad \text{for } k \geq 0 \qquad (1)$$

$$V_k(s) = \max_a Q_k(s,a) \quad \text{for } k>0. \qquad (2)$$

We concentrate on implementing the value iteration in the gridworld model. It can either save the V[S] array or the Q(S, A) array. Saving the V array results in less storage, but it is more difficult to determine an optimal action, and one more iteration is needed to determine which action results in the greatest value. After an infinite number of iterations value iterations converge to V^{π}. When the change in value function is less than a very small positive number we terminate the algorithm. Unlike policy iteration, there is no explicit policy in value iteration. In our algorithm, policy is evaluated in every iteration of the episode as shown by the MATLAB code below.

```
function [v, pi] = valueIteration(model, maxit)
% initialize the value function
v = zeros(model.stateCount, 1);
v1 = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
epsilon = 1.0000e-22;

for i = 1:maxit,
% initialize the policy and the new value function
    policy = ones(model.stateCount, 1);
    v_ = zeros(model.stateCount, 1);

% perform the Bellman update for each state
    for s = 1:model.stateCount,
% COMPUTE THE VALUE FUNCTION AND POLICY
% YOU CAN ALSO COMPUTE THE POLICY ONLY AT THE END
        P = reshape(model.P(s,:,:), model.stateCount, 4);
        [v_(s,:), action] = max(model.R(s,:) +
                        (model.gamma * P' * v)');
        policy(s,:) = action;
    end


    v1 = v;
    v = v_;
    pi = policy;

    % exit early
    if v - v1 <= epsilon
        % CHANGE THE IF-STATEMENT
        fprintf('Function converged after %d
        episode\n',i);
        break;
    end
end
```

1) *Experiment Results*

The plot shown in figure 1 is of the gridworld MDP. For smallworld MDP we also obtain the same result as shown in coursework question. We calculate the number of iterations it took to converge. We used 1000 iterations to guarantee convergence of value functions in value iteration. Value function can be considered as a goodness measure of how good it is for an agent to be in a state. At each state, it is a measure of the expectation of cumulative rewards over the time steps. In figure 1, regions completely dark denote that it is bad for the agent to be in that state, on the other hand regions marked white shows maximum goodness. The lighter the regions, the better it is for the agent to be in those states.

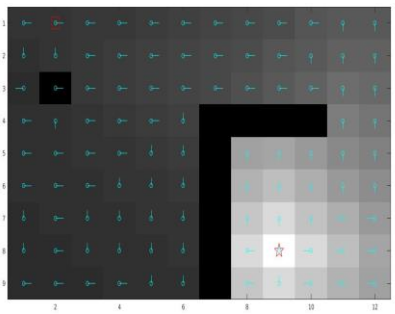

Figure 1. Value function and policy for 1000 iterations in gridworld MDP.

2) *Discussion of Results*

Figures 1 shows the value function and the policy in the gridworld MDP. Value iterations converges after 120 iterations in gridworld and took 49 for the smallworld. It indicates that the relation between state complexity and convergence iteration is proportional. Number of iterations also depends upon the threshold value in large state space. Using an infinite number of iterations, value iteration is guaranteed to converge to an optimal V^{π}. The directions of arrows in figure 1 means those are the optimal actions to take at every state.

B. *Policy Iteration*

The policy iteration algorithm manipulates the policy directly, rather than finding it indirectly via the optimal value function. The value function of a policy is just the expected infinite discounted reward that will be gained, at each state, by executing that policy. It can be determined by solving a set of linear equations. Once we know the value of each state under the current policy, we consider whether the value could be improved by changing the first action taken. If it can, we change the policy to take the new action whenever it is in that situation. This step is guaranteed to strictly improve the performance of the policy. When no improvements are possible, then the policy is guaranteed to be optimal. Once a policy, π, has been improved using V^{π} to yield a better policy, π', we can then compute $V^{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*;$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement. Policy evaluation step evaluates the value function for a policy π using the Bellman expectation backup. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.
The MATLAB code is added below.

```
function [v, pi] = policyIteration(model, maxit)
% initialize the value function
v = zeros(model.stateCount, 1);
epsilon = 0.001;

for i = 1:maxit,
% initialize the policy and the new value function
    pi = ones(model.stateCount, 1);
    v_ = zeros(model.stateCount, 1);

    for s = 1:model.stateCount,
        TranProb = reshape(model.P(s,:,:),
        model.stateCount, 4);
        [~, action] = max(model.R(s,:) +
        (model.gamma * TranProb' * v)');
        v_(s,:) = model.R(s, action) + (model.gamma
        * TranProb(:, action)' * v)' ;
        pi(s,:) = action;
    end

    v1 = v;
    v = v_;

    if v - v1 <= epsilon
        fprintf('Function converged after %d
        episode\n',i);
        break;
    end

end
```

1) *Experimental Results*

We first use an iterative policy evaluation method to estimate V^{π} and then use a greedy policy improvement step to improve the policy. Like value iteration, policy iteration also always converges to the optimal value function V^{π} from which the optimal π^* can be derived. Surprisingly policy iteration often converges in few iterations. We found that policy iteration converges after 46 iterations, when running with 1000 iterations whereas value iterations took 120 iterations.

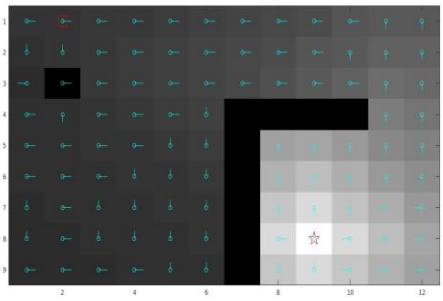

Figure 2. Policy iteration for 1000 iterations in gridworld MDP.

We can converge faster to the optimal policy π. Value iteration took longer to converge because the iterations of value iteration are independent of the actual policy, and algorithm runs even when the policy is not changing. In RL the goal is to find the optimal policy and value functions. So, it is better to evaluate the policy directly and calculate convergence by directly computing the policy. That's why policy iteration provide a good measure of when value functions have converged. When there are no more improvements in the policy, the value function is guaranteed to converge. Compared to this, value iteration has no measure of the policy and therefore finding the optimal value function takes large number of iterations than policy iteration.

2) Discussion of Results

Figure 1 and 2 both are same. This is because both yields some sort of the same result. As mentioned before value iteration and policy iteration is similar, except policy iteration compute policy evaluation in every step. That's why when both of this algorithm converges for same V^{π} they generate same π^{*}. So, if value function and actions are same at convergence in a specific MDP there can only be one optimal policy.

C. SARSA Algorithm

SARSA (so called because it uses state-action-reward-state-action experiences to update the Q-values) is an on-policy reinforcement learning algorithm that estimates the value of the policy being followed. SARSA is a form of Temporal Difference (TD) learning method. SARSA learns directly from episodes of experience and learns from incomplete episodes by bootstrapping unlike Monte Carlo methods which require complete episodes. This means using SARSA, we can learn before knowing the final outcome, can learn online after every step from incomplete episodes and works in continuing environments. An experience in SARSA is of the form ⟨s, a, r, s', a'⟩, which means that the agent was in state s, did action a, received reward r, and ended up in state s', from which it decided to do action a'. This provides a new experience to update Q(s, a). The new value that this experience provides is r + γQ(s', a').

$$Q(s, a) = Q(s, a) + \alpha[R + \gamma Q(s', a') - Q(s, a)] \quad (3)$$

We implement one step return SARSA algorithm, which is guaranteed to converge to the optimal action-value function with appropriately chosen step size. For the epsilon-greedy policy improvement step, we use the following epsilon-greedy approach:

$$\text{action} = \arg\max_{a0} Q(s0, a0) \quad (4)$$

The convergence of the SARSA algorithm depends on the Q function. SARSA algorithm is guaranteed to converge to an optimal Q and therefore optimal policy as long as all the state-action pairs in the environment are visited an infinite number of times. This depends on the exploration-exploitation tradeoff. By considering a greedy policy with a carefully chosen Ɛ parameter, we can balance the exploration-exploitation, such that the states action pairs are visited large number of times. The MATLAB code for SARSA is given below:

```
function [v, pi, c_Rew] = sarsa(model, maxit,
maxeps)

% initialize the value function
Q = zeros(model.stateCount, 4);
pi = ones(model.stateCount, 1);
policy = ones(model.stateCount, 1);
c_Rew = zeros(length(maxeps), 1);

for i = 1:maxeps,
% every time we reset the episode, start at the
given startState
    s = model.startState;
    a = 1;
%   a = epsilon_greedy_policy(Q(s, :), epsilon);
    Rew = 0;
     for j = 1:maxit,
% PICK AN ACTION
%         a = 1;
        p = 0;
        r = rand;

        for s_ = 1:model.stateCount,
            p = p + model.P(s, s_, a);
            if r <= p,
                break;
            end
        end
% s_ should now be the next sampled state.
% IMPLEMENT THE UPDATE RULE FOR Q HERE.

        R = model.R(s,a);
        Rew = Rew + R;
        a_ = epsilon_greedy_policy(Q(s, :), j);
         alpha = 1/j;
          Q(s,a) = Q(s,a) + alpha * [R + model.gamma
*        Q(s_ , a_) - Q(s , a)];
        s = s_;
         a = a_;
        [~, idx] = max(Q(s,:));
        policy(s,:) = idx;
        Q2 = Q(:, idx);
% SHOULD WE BREAK OUT OF THE LOOP?
         if s == model.goalState
%             if R == model.R(model.goalState, a)
%                 break;
%             end
         break;
         end
    end
    c_Rew(i) = Rew;
end

% REPLACE THESE
pi = policy;
v = Q2;

end
```

The MATLAB code for the epsilon greedy policy is given below:

```
function action = epsilon_greedy_policy(Q, j)
    Total_a = [1 2 3 4];
%     e = 0.01;
    e_prob = rand();
    epsilon = 1/j;
    if e_prob < (1 - epsilon)
        [~, action] = max(Q);
    else
        action = Total_a(randi(length(Total_a)));
    end

end
```

1) Experimental Results

Here, we use maxit = 1000 iterations for an episode, and maxeps = 1000 episodes to calculate the Q function. In our algorithm, we initialize the action as a = 1, instead of that we can use an epsilon-greedy action at the beginning of each episode. Which results in a significant difference as we can see from figure 3 and figure 4. Figure 3 represents SARSA on smallworld MDP with decaying value of alpha and epsilon whereas figure 4 represents fixed alpha and epsilon SARSA.

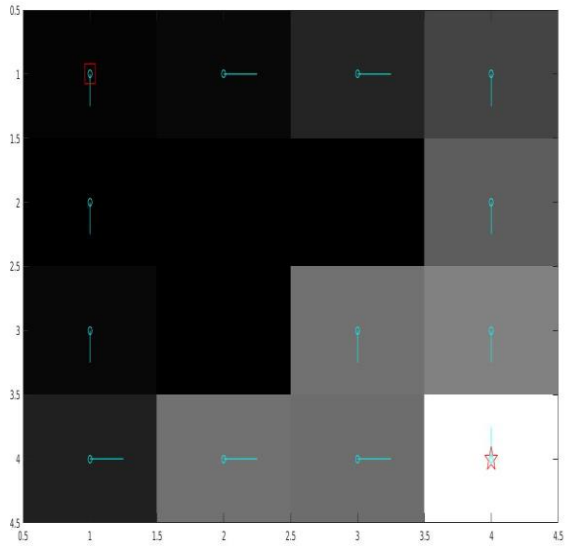

Figure 3. SARSA Algorithm on smallworld MDP with Decaying ε and α parameters.

Figure 4 represent the change in policy and value functions at every state changes with fixed alpha and epsilon parameters, instead of decaying values.

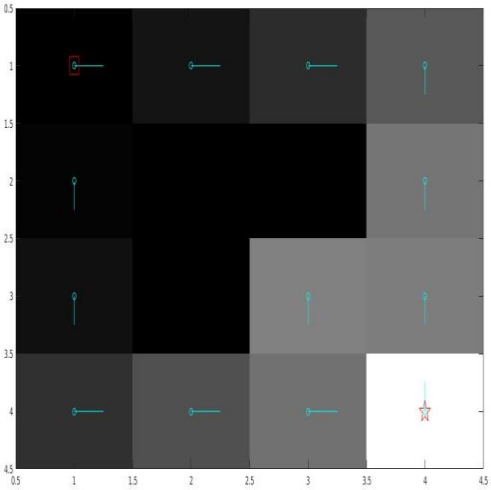

Figure 4. SARSA Algorithm on smallworld Model, ε = 0.02, α = 0.1

2) Discussion of Results

In figure 3 we are using a decaying epsilon, in which the ε parameter decays with the number of steps in each episode. The α is also decaying like the epsilon as α = 1/j. At the beginning of each episode, the agent is encouraged to explore more of the environment since it has not visited all the states yet. A large value of ε encourages more exploration since the probability of choosing an action from a uniform distribution is higher, as given by the conditions for action selection given above. However, at later stages of a given episode, ie, later steps within an episode, the agent has already explored the environment, and now should be encouraged to exploit more of the given information. This is achieved by smaller ε values such that the exploitation can be done by choosing an action that maximizes the Q function. Similarly, a high α at the start of an episode ensures faster to achieve convergence of the Q function. The Q function is updated at every step in the SARSA algorithm. However, at later stages of a given episode, smaller values of α ensures that the optimal Q function can be reached.

Figure 4 represents results when choosing a fixed ε and α parameters. A fixed ε means that we are encouraging the agent to explore as much in later stages of an episode even after having visited most of the states. Such effect is more apparent in small state spaces such as in the smallworld MDP. In a small world MDP with small state space compared to gridworld, exploration at later stages of an episode should be less encouraged since the agent must have visited all the states already. A fixed α value also does not necessarily ensure convergence to optimal Q function since a small alpha should be required to update Q at later steps in the episode to ensure good convergence properties. Comparing results of SARSA on figure 3 and 4, we can see that a fixed ε and α should be more acceptable.

D. Q-Learning Algorithm

Q-Learning is an Off-Policy algorithm for Temporal Difference learning. It can be proven that given sufficient training under any ε -soft policy, the algorithm converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy. Q-Learning learns the optimal policy even when actions are selected according to a more exploratory or even random policy. The behavior policy is determined by epsilon-greedily ensuring sufficient exploration, while the target policy is obtained by arg $\max_{a0}$ Q(st+1, a0). The MATLAB code for Q-learning is given below:

```
function [v, pi, c_Rew] =
qLearning(model, maxit, maxeps)
% initialize the value function
Q = zeros(model.stateCount, 4);
pi = ones(model.stateCount, 1);
policy = ones(model.stateCount, 1);
c_Rew = zeros(length(maxeps), 1);
```

```
for i = 1:maxeps,
% every time we reset the episode, start at the
given startState
    s = model.startState;
    a = 1;
%   a = epsilon_greedy_policy(Q(s,:), epsilon);
    Rew = 0;

    for j = 1:maxit,
        % PICK AN ACTION
%         a = 1;
        p = 0;
        r = rand;

        for s_ = 1:model.stateCount,
            p = p + model.P(s, s_, a);
            if r <= p,
                break;
            end
        end

% s_ should now be the next sampled state.
        % IMPLEMENT THE UPDATE RULE FOR Q HERE.

        R = model.R(s,a);

        Rew = Rew + R;

        a_ = epsilon_greedy_policy(Q(s,:), j);
        alpha = 1/j;

        Q(s,a) = Q(s,a) + alpha * ( R + model.gamma
       * max(Q(s_, :)) - Q(s,a) );
        s = s_;
        a = a_;

        [~, idx] = max(Q(s,:));
        policy(s,:) = idx;
        Q1 = Q(:, idx);

        % SHOULD WE BREAK OUT OF THE LOOP?
         if s == model.goalState
             if R == model.R(model.goalState, a)
                 break;
             end
         break;
         end

    end
    c_Rew(i) = Rew;
end

% REPLACE THESE
pi = policy;
v = Q1;

end
```

1) Experimental Results

As SARSA, we also used decaying ε = 1/j and α = 1/j to implement Q-learning algorithm also maxit and maxeps value was same as before. Figure 5 represents the value function and policy obtained in the smallworld MDP by off-policy Q-learning.

2) Discussion of Results

In Q-learning, in order to convergence the idea is that all state action pairs need to be continually updated, since the learned Q function directly approximates to Q^{π}. Q-learning learns an optimal policy no matter which policy the agent is actually following (i.e., which action a it selects for any state s) as long as there is no bound on the number of times it tries an action in any state (i.e., it does not always do the same subset of actions in a state). Because it learns an optimal policy no matter which policy it is carrying out, it is called an off-policy method.

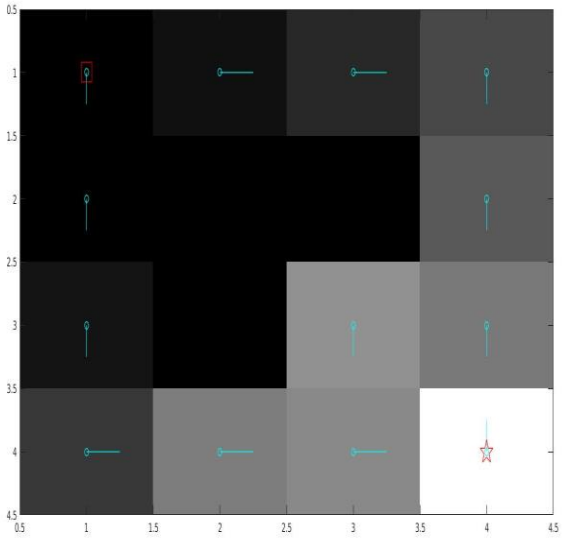

Figure 5. Q-Learning on smallworld MDP.

E. SARSA vs Q-Learning

In order differentiate between SARSA and Q-Learning we compare the cumulative rewards obtained from each episode for both of the algorithms. The cumulative reward is obtained by calculating the rewards with every (s, a) pair visited by the agent. The convergence of the cumulative reward function shows that using SARSA and Q-Learning, we can also converge to an optimal Q function. When cumulative reward becomes constant or steady, it indicates that the optimal policy has been reached, with more exploration, no other better policy can be gained.

The MATLAB code below shows calculation of discounted cumulative reward :

```
Rew = Rew + model.gamma * R;
c_Rew(i) = Rew;
```

Balancing the ratio between exploration and exploitation is one of the most challenging tasks in reinforcement learning with great impact on the agent's learning performance. On the one hand, too much exploration prevents the agent from maximizing short-term reward because selected exploration actions may yield negative reward from the environment. On the other hand, exploiting uncertain environment knowledge prevents from maximizing long-term reward since selected actions may remain suboptimal. This problem is well known as the dilemma of exploration and exploitation.A straightforward—and often very successful—approach is to balance exploration/exploitation by the ε-greedy method. With this method, the amount of exploration is globally controlled by a parameter, ε, that determines the randomness in action selections. In contrast to others, one advantage of ε-greedy is the fact that no memorization of exploration specific data is required, such as counters or confidence bounds, which makes the method particularly interesting for very large or even continuous state-spaces.

Convergence of optimal policy depends on the convergence of Q function and Q* depends upon alpha parameter. Due to higher state space in cliffworld we need to find tune the ε and α parameter in order to get good performance. Q-Learning's or SARSA's cumulative reward is not always smooth because of updates of the Q functions depend on greedy action and maximization of Q(s,a). Because it is unclear that which setting of ε leads to good results for a given learning problem.

In order to subsidize of the above issues, we ran our experiments with different combinations of ε and α parameters. For both of the algorithm, we ran the same experiment with the same number of episodes and iterations and then took the average cumulative reward for cliffworld MDP. Below we also showed our plot of how the ε parameter determines the amount of exploration and exploitation in the environment. A very small value of epsilon leads to more exploitation since $a = \arg\max_a Q(s0, a0)$ with probability 1 - ε is chosen with a higher probability. Compared to that, if ε values are too high, then this leads to more exploration of the state space and less exploitation of the states that have already been visited.

1) Experimental Results

The result below in figure 6 is obtained using an α = 0.2 and ε = 0.2 and experiments averaged over 150 iterations to smooth out. We used 500 episodes and 500 iterations within each episode for both of the algorithm. The figures show the discounted cumulative reward with the number of episodes.

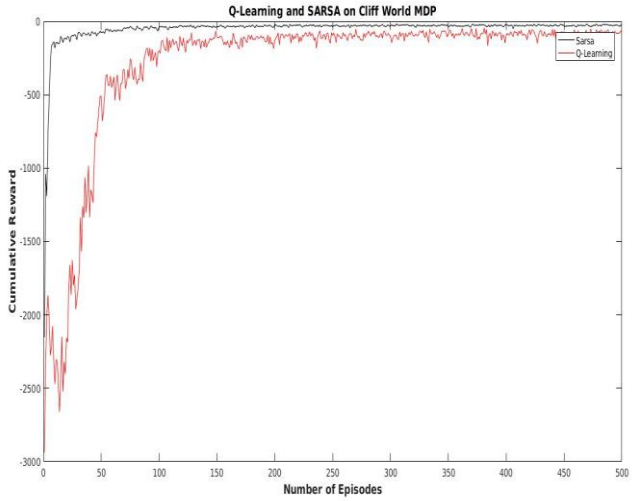

Figure 6. Cumulative Rewards: Q-Learning and SARA on cliffworld MDP

The lower part of the figure shows the performance of the SARSA and Q-learning methods with ε-greedy action selection. After an initialization, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection. SARSA, on the other hand, takes the action selection seriously and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of SARSA. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.

We also experiment on our Q-learning and SARSA algorithms with varying ε and α compared the convergence of the Q function's dependency upon cumulative reward. Figure 9 below shows results with decaying ε and α parameters.

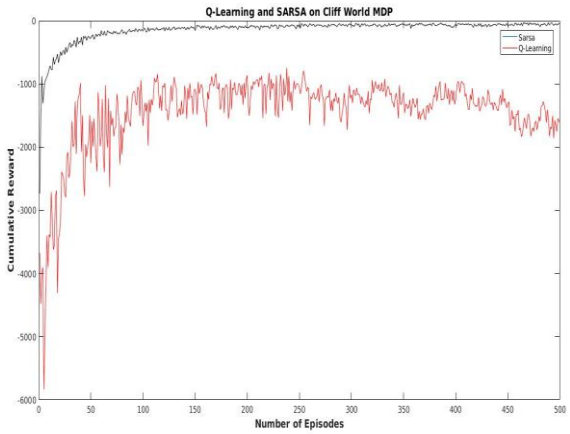

Figure 7. Cumulative Rewards: Q-Learning and SARSA on cliffworld MDP with Decaying ε and α

In figure 9 we present the results of how the performance of the agent for finding an optimal Q and optimal policy, in both Q-Learning and SARSA is dependent on the value of step size α and exploration parameter ε. In figure 8 and 9, an epsilon value of 0.4 is used with varying α parameters. We varied the same number of alpha, and notice significant differences in how the ε parameter has a higher effect during Q-Learning compared to SARSA algorithm.

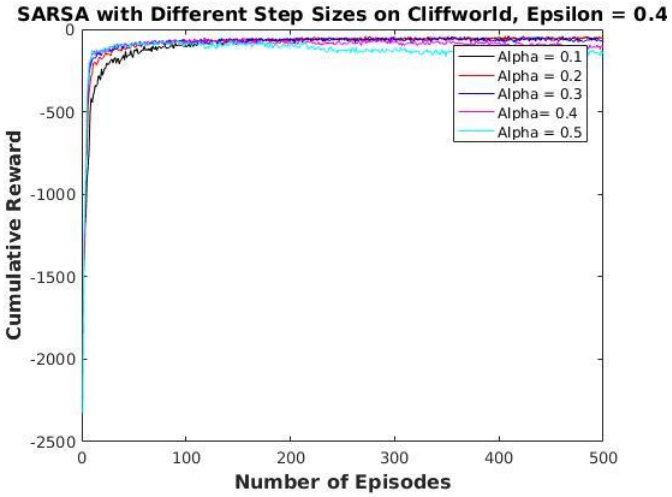

Figure 8. SARSA with ε = 0.4 with varying step sizes on cliffworld MDP

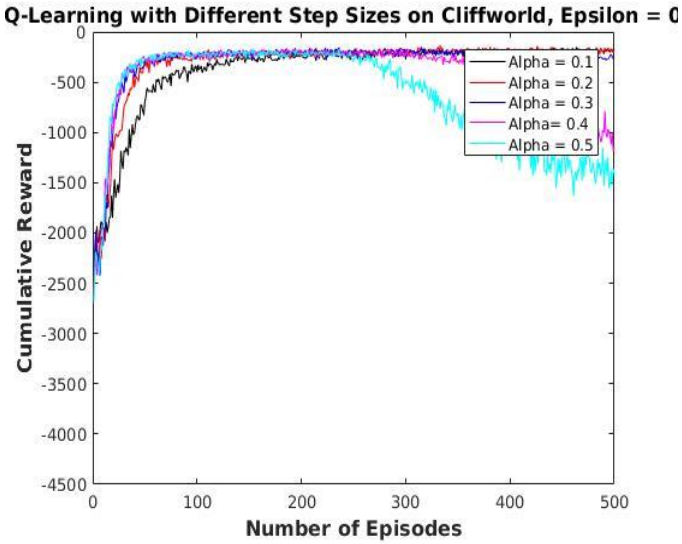

Figure 9. Q-Learning with ε = 0.4 with varying step sizes on cliffworld MDP

At last we present the results with an ε parameter of 0.6 for both the algorithm. A small value of epsilon encourages more exploitation of the state space. Figures 10 and 11 shows how the exploration-exploitation trade off plays a more significance role in SARSA and Q-Learning.

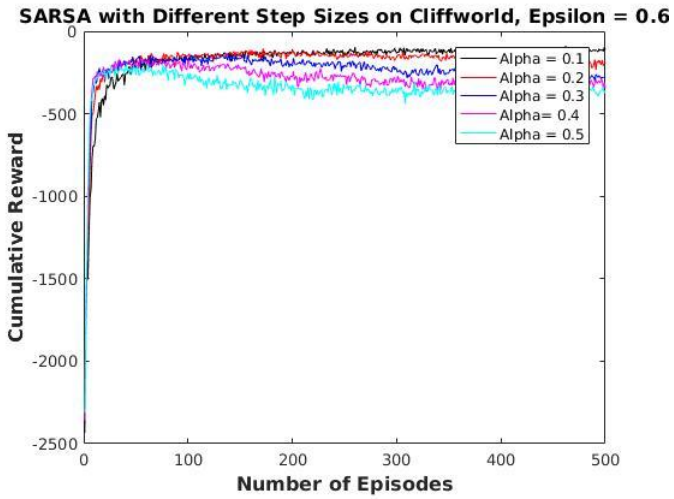

Figure 10. SARSA with ε = 0.6 with varying step sizes on cliffworld MDP

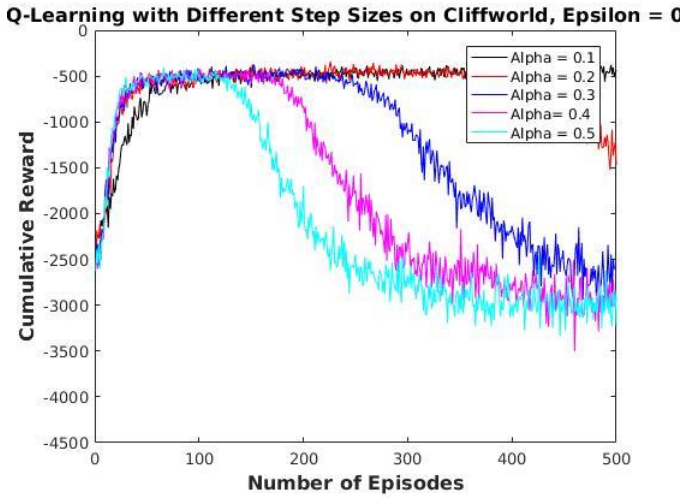

Figure 11. Q-Learning with ε = 0.6 with varying step sizes on cliffworld MDP

2) Discussion of Results

Figure 6 shows how the learning performance of the agent, and the cumulative reward obtained and how they varies depending on whether the action-value functions are updated using off-policy or on-policy. It represents that SARSA learns better and is good at avoiding the bad states compared to Q-learning. This is because, each step of SARSA update is dependent on the Q function obtained by the states and actions visited. Compared to that, in Q-learning, the Q function is updated is made by maximizing the actions from off-policy.

Compared to Q-learning, SARSA learns directly from episodes of experience and can learn from incomplete episodes, and can learn before knowing the final outcome. This on-policy method further makes SARSA more suitable to avoid bad states with low or negative rewards in the environment.

From figure 6 we can say that Q-learning is more interested with the exploration of the environment than the reward. During learning, Q-learning tries to update Q depending on the exploratory actions from the behavior off-policy, which makes it more prone to get into bad states while it is learning. But, once Q-learning learns well to avoid bad states, it reaches a maximal cumulative reward close to SARSA. Both the algorithms are guaranteed to converge to an optimal Q^{π} under sufficient conditions.

Figure 7 represents how the Q-learning and SARSA algorithms are dependent on the exploration parameter ε. Our hypothesis is that the ε parameter has a greater effect in Q-Learning compared to SARSA. It shows that Q-learning is more heavily affected if the ε is not well-tuned. This is because since the Q-function updates in Q-Learning is more dependent on the exploratory off-policy, therefore the amount of exploration-exploitation has more effect in Q-Learning. Form figure 7 we can say that learning with decaying exploratory parameter, such that exploration is more encouraged initially, while exploitation is more encouraged towards the end of learning episodes.

This hypothesis can be further evaluated by the figures 8, 9, 10 and 11. Comparing figure 11 and 13, figure 11 shows that even if the α parameter is carefully fine-tuned, the large ε parameters makes the agent more prone to going into bad stages, compared to figure 9 and 10 when the ε parameter is chosen to be low. This further evaluates the fact that Q-learning takes less account of bad states while it is learning.

Finally, those figures prove that on-policy learning such as SARSA is more effective to avoid bad states in the MDP, and reaches a better cumulative reward while learning. Compared to that, Q-Learning performs well once it learns completely, although due to the higher dependency on the exploration-exploitation trade-off, it is more prone to falling into the bad states in the environment. We have also shown that the ε exploration parameter has a greater significance in off-policy learning since the Q function updates is done by exploratory actions in Q-learning.