MPhil in Machine Learning, Speech and Language Technology
2015-2016

MODULE COURSEWORK FEEDBACK

Student Name: Riashat Islam

Module Title: Reinforcement Learning and Decision Making

CRSiD: ri258

Module Code: MLSALT7

College: St John's

Coursework Number: 1

I confirm that this piece of work is my own unaided effort and adheres to the Department of Engineering's guidelines on plagiarism

---

Date Marked:

Marker's Name(s):

Marker's Comments:

This piece of work has been completed to the following standard *(Please circle as appropriate)*:

	Distinction			**Pass**			**Fail (C+ - marginal fail)**		
Overall assessment (circle grade)	**Outstandi** ☐	**A+** ☐	**A** ☐	**A-** ☐	**B+** ☐	**B** ☐	**C+** ☐	**C** ☐	**Unsatisfactor** ☐
Guideline mark (%)	90-100	80-89	75-79	70-74	65-69	60-64	**55-59**	**50-54**	**0-49**
Penalties	**10% of mark for each day late (Sunday excluded)**								

The assignment grades are given **for information only**; results are provisional and are subject to confirmation at the Final Examiners Meeting and by the Department of Engineering Degree Committee.

Reinforcement Learning Coursework

Riashat Islam RI258@CAM.AC.UK
University of Cambridge

1. Introduction

In this work, we consider analysis of the basic reinforcement learning algorithms on three different models (MDPs). We consider value and policy iteration and discuss the proof of convergence for these algorithms. We then consider the difference in performance between SARSA and Q-Leaning on benchmark RL tasks such as the cliffworld model. In each section, we first include a brief explanation of the algorithm, present code used for implementation and then include a discussion of results.

1.1. Experimental Setup

Models/Environments: We consider experiments over three different benchmark RL tasks. For all the environments, the agent is allowed 4 discrete actions in a discrete state space. In the Small World MDP (model), the goal states (state with maximal reward) is at the coordinates $(4, 4)$. We consider a constant reward of -1 in all the states, reward of 10 in the goal state and a negative reward (cost) of -6 in the bad state. The smallworld MDP has 17 different states. Similarly, the gridworld MDP has 109 different states, with the goal state at 93 and pre-defined start state. We also consider a cliffworld MDP with the usual actions of up, down, left and right, with a reward of -1 for all transitions, except for a reward of -100 if the agent enters the cliff, and a reward of 10 in the goal state with co-ordinates $(5, 9)$. We always consider discounted MDPs with $\gamma = 0.9$. Our work also considers careful fine-tuning of the ϵ parameter for greedy action selection as discussed later. We always consider stochastic discrete policies in a model-based environment with given transition dynamics.

2. Question A: Value Iteration

In reinforcement learning, value iteration concerns with finding the optimal policy π using an iterative application of the Bellman optimality backup. At each iteration, value iteration uses synchronous updates for all the states to update the value function, ie, update $V_{k+1}(s)$ from $V_k(s')$. In our work, we consider using value iteration over the gridworld model. In order for value iteration to converge, it requires an infinite number of iterations to converge to $V*$. We use the stopping criterion for value iteration that when there are no further improvements to the value function, or when the change in value function is less than a very small positive number in a given sweep, we terminate the algorithm. Value iteration is guaranteed to converge to an optimal policy for finite MDPs with a discounted reward. Unlike policy iteration, there is no explicit policy in value iteration. In our algorithm, we evaluate the policy at each step of the episode as shown by the code below.

$$V_{k+1}(s) = \max_{a \in A}(R_s^a + \gamma(\sum_{s' \in S} P_{ss'}^a V_k(s')) \tag{1}$$

The **MATLAB code** for value iteration is given below:

```
% run VI on GridWorld
gridworld;
[v, pi] = valueIteration(model, 1000)
plotVP(v,pi, paramSet)

%value iteration algorithm
function [v, pi] = valueIteration(model, maxit)
% initialize the value function
v = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
old_v = zeros(model.stateCount, 1);
threshold =  1.0000e-22;

for iterations = 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 transition probability
    P = reshape(model.P(s,:,:), model.stateCount, 4);
    %update value function
    [v_(s,:), action] = max(model.R(s,:) +
        (model.gamma * P' * v)');
    %policy evaluated every step
    policy(s,:) = action;
    end
old_v = v;
v = v_;
pi = policy;
%break condition
%to check convergence of VI algorithm
if v - old_v <= threshold
    fprintf('Value function converged
    after \%d iterations\n',iterations);
        break;
end

end
end
```

Explanation of the Code: We used 1000 iterations of value iteration, ie, considering 1000 episodes. For each state in a given episode, the value function is updated by the following, with the policy $\pi(a|s)$ being evaluated by the action that maximizes the expression. Experimental results are given in section 2.1 below.

```
[v_(s,:), action] = max(model.R(s,:) +
(model.gamma * P' * v)');
```

To check for convergence, we use a **threshold value** of $1e^{-22}$ to ensure convergence of the value function, such that the value iteration algorithm stops when there are no more improvements in the value function. In other words, when we converge to an optimal value function $V*$.

```
if v - old_v <= threshold
fprintf('Value function converged after
\%d iterations\n',iterations);
break;
end
```

2.1. Experiment Results

The result in figure 1 is shown for the gridworld MDP. Our algorithm shows the same result for small world as given in the testing result. In our code, we also show the number of iterations it took for the value iteration algorithm 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 given state. At each given state, it is a measure of the expectation of cumulative rewards over the time steps. This is shown by the figure 1. In the figure 1, the colours specify the goodness measure. Regions completely dark denote that it is bad for the agent to be in that state, whereas regions marked white shows maximal goodness (goal state). The lighter the regions, the better it is for the agent to be in those states.

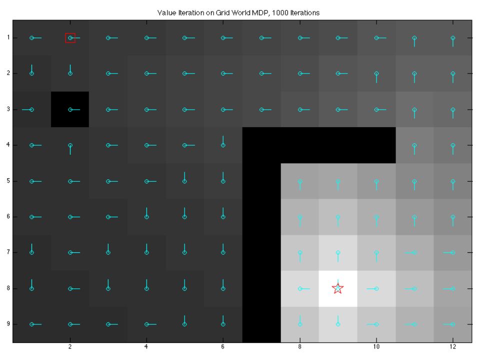

Figure 1. Value Iteration - Value Function and Policy for Grid World MDP

2.2. Discussion of Results

Figures 1 shows the value function and the policy (shown by the actions taken at every state) on the smallworld and gridworld MDPs. For the grid world, the value function converges after 120 iterations, whereas for the smallworld, it converges after 49 iterations. Certainly, higher number of iterations for convergence of value iteration are required in larger state spaces. The number of iterations required for convergence is also dependent on the threshold parameter used which terminates the algorithm when the improvement in value function at every iteration is less than the threshold. In practice, such conditions are required for termination of value iteration algorithms. Using an infinite number of iterations, value iteration is guaranteed to converge to an optimal $V*$. The directions of arrows after convergence of value function shown in figure 1 are therefore the optimal actions to take at every state.

3. Question B: Policy Iteration

We then consider policy iteration instead of value iteration on the gridworld model. Policy iteration uses an iterative policy evaluation step to estimate V^π and a policy improvement step to generate $\pi' > \pi$, where π' is obtained by a greedy policy improvement step. The policy evaluation step evaluates the value function for a given policy π using an iterative Bellman expectation backup. The policy improvement step is then the action that maximizes the value function, using a greedy policy improvement step. In other words, policy iteration obtains a sequence of continually improving policies and value functions, where we do policy evaluation and improvement separately, while every policy improvement is guaranteed to be an improvement.

The **MATLAB** code is included below.

```
function [v, pi] = policyIteration(model, maxit)

% initialize the value function
v = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
% old_v = zeros(model.stateCount, 1);
policy = ones(model.stateCount, 1);
tol = 0.0000000000000000000001;

%run this extra loop
% to check for convergence
%in policy evaluation and
%policy improvement step
for iterations = 1:maxit,

    % Policy Evaluation Step
    %with same number of episodes
    for i = 1:maxit,
    v_ = zeros(model.stateCount, 1);
    % perform the Bellman update for each state
    for s = 1:model.stateCount
    v_(s) = model.R(s, policy(s))
    + (model.gamma*model.P(s,:,policy(s))*v)';
    end
    delta = norm(v - v_);
    v = v_;

%check for convergence
if delta <= tol
fprintf('Value function
converged after $\%$ d iterations\n',i);
break;
end
end

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

    policy(s) = action;
end
end
pi = policy;
v = v_;

end
```

Explanation of Code: In the code above, there are two stages of policy iteration. First we evaluate the value function given the current policy, and then the policy improvement step is to take the action that maximizes the value function using a greedy improvement step as below:

```
[~, action] = max(model.R(s,:)+
(model.gamma  * P' * v)' );
```

To check for convergence of policy iteration, we again use a stopping criteria to check if there are no more improvements in the value function. When there are no improvements in V^π, it also suggests that $\pi' = \pi$, ie, there are no more improvements in the policy from the greedy policy improvement step.

Sample Output of Code: In our code, we used two loops to check for convergence of policy iteration. The policy evaluation step to estimate V^π is done with same number of iterations as episodes (second loop), followed by which we do policy improvement. The first loop further checks when policy improvement step has converged as well, such that our output looks as below:

```
Value function converged after 338 iterations
Value function converged after 281 iterations
Value function converged after 261 iterations
Value function converged after 171 iterations
Value function converged after 104 iterations
Value function converged after 88 iterations
Value function converged after 1 iterations
Value function converged after 1 iterations
Value function converged after 1 iterations
```

From the above we see that, after 88 iterations, the policy iteration algorithm converges, and there are no further improvements in the greedy policy improvement step after convergence.

3.1. Experimental Results

Our code above shows the two stages in policy iteration compared to value iteration. In policy iteration, we first use an iterative policy evaluation method to estimate V_π and then use a greedy policy improvement step to improve the policy. Similar to value iteration, the process of policy iteration also always converges to the optimal value function V^* from which the optimal π^* can be derived.

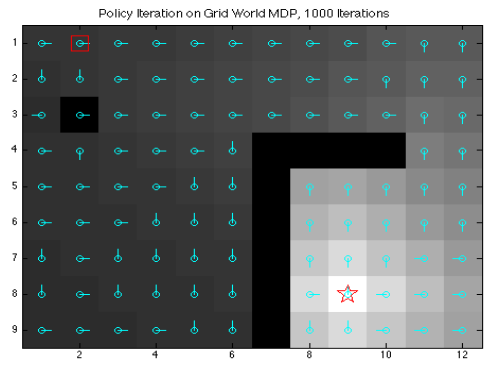

Figure 2. Policy Iteration on Grid World Model

From our experimental results, we find that the policy iteration algorithm converges after 88 iterations, when we are using 1000 iterations for the algorithm, compared to 120 iterations of value iteration algorithm. In section 3.2.1, we include a comparision of convergence of policy and value iteration and discuss why the convergence to the optimal value function differs for these algorithms.

3.2. Discussion of Results

Note that both figures 1 and 2 for the GridWorld MDP should be the same. This is because, as explained earlier, policy iteration is same as value iteration, except that policy iteration also considers policy evaluation in each of the iterations. Therefore, at convergence of V^*, both policy and value iteration should yield the same optimal policy π^*. The value function (denoted by the colour) and the actions (arrows) should therefore be the same at convergence, since for a given MDP, there can only be one optimal policy.

3.2.1. Comparing Convergence of Value and Policy Iteration

The value iteration algorithm converges after 120 iterations. Compared to that, the policy iteration algorithm converges after 88 iterations. This further validates that using policy iteration, we can converge faster to the optimal policy π^*. The reason is further explained below: A drawback of using value iteration is that it can take longer for value iteration algorithms to converge in some state spaces. This is because the iterations of value iteration is independent of the actual policy, and so the algorithm runs even when the policy is not changing. Since in RL, the goal is to find the optimal policy, and value functions at each state provides a tool to find the optimal policy, it is indeed better to evaluate the policy directly and check for convergence based directly computing the policy. Policy iteration algorithms therefore provide a good measure of when value functions have converged, since we can directly compute the policy. When there are no more improvements in the actual policy, the value function is guaranteed to converge. Compared to this, value iteration uses no measure of the actual policy and hence finding the optimal value function takes larger number of iterations than policy iteration. Policy iteration algorithms converges faster.

4. Question C: Convergence of Policy and Value Iteration

This section considers the proof of convergence of the policy iteration algorithm. We denote our policy as $\pi(s)$ and the Bellman operator as T. As discussed earlier, policy iteration algorithm involves two steps: policy improvement followed by policy evaluation.

First we show that by acting greedily means policy improvement $\pi'(s) > \pi(s)$ for every greedy action. Since the value function is given as $V_{\pi(s)} = R_s + \gamma P^{\pi} V$, greedy action means:

$$\pi'(s) = \arg\max_a [R_s + P^{\pi} V] \tag{2}$$

therefore, a greedy action leads to an improvement in the value function:

$$V_{\pi'(s)} \geq V_{\pi(s)} \tag{3}$$

In other words, since $Q_\pi(s, \pi(s)) = V_\pi(s)$, this means

$$Q_\pi(s, \pi'(s)) \geq Q_\pi(s, \pi(s)) \tag{4}$$

We can therefore prove that a policy improvement step in the policy iteration algorithm leads to $V_{\pi'}(s) \geq V_\pi(s)$ as follows:

$$\mathrm{V}_\pi(s) = \leq Q_\pi(s, \pi'(s))$$
$$= \mathrm{E}_\pi[R_{t+1} + \gamma V_\pi(s_{t+1}) | s_t]$$
$$= \leq E_{\pi'}[R_{t+1} + \gamma Q_\pi(s_{t+1}, \pi'(s_{t+1})) | s_t]$$
$$= \leq E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 Q_\pi(s_{t+2}, \pi'(s_{t+2})) | s_t$$
$$= \leq E_{\pi'}[R_{t+1} + \gamma R_{t+2} + ... | s_t]$$

$= \mathrm{V}_{\pi'}(s)$

(5)

The policy improvement step in policy iteration therefore guarantees that:

$$V^{\pi_{k+1}} \geq V^{\pi_k} \tag{6}$$

Using the Bellman operator this therefore means:

$\mathrm{V}^{\pi_k} = T^{\pi_k} V^{\pi_k}$

$\mathrm{V}^{\pi_k} =\leq TV^{\pi_k}$

$\mathrm{V}^{\pi_k} = T^{\pi_{k+1}} V^{\pi_k}$

(7)

where π_{k+1} is an improved policy following greedy policy improvement. We want to show that following each step of policy improvement, there is an improvement in the value function given by the Bellman operator as follows:

$$V^{\pi_k} \leq T^{\pi_{k+1}} V^{\pi_k} \tag{8}$$

Hence, following the Bellman operator, the improvement of policy from π_k to π_{k+1} shows an improvement in the value function as given by:

$$V^{\pi_k} \leq T^{\pi_{k+1}} V^{\pi_k} \leq (T^{\pi_{k+1}})^2 V^{\pi_k} \leq ... \tag{9}$$

Hence, we can show that:

$$V^{\pi_k} \leq lim_{n->\inf} (T^{\pi_{k+1}})^n V^{\pi+k} = V^{\pi_{k+1}} \tag{10}$$

Following iterations in policy iteration algorithm, when the policy improvement stops, ie $\pi'(s) = \pi(s)$, or alternatively we can write $Q_\pi(s, \pi'(s)) = Q_\pi(s, \pi(s)) = V_\pi(s)$. When an optimal policy is reached following greedy policy improvement steps, it will satisfy the Bellman optimality equation:

$$V^{\pi_k} = V^* \tag{11}$$

So the algorithm stops after finite steps k, when there is no more improvement in policy improvement step, and hence the policy evaluation step. **This means convergence of the policy iteration algorithm**. Policy iteration stops when there are no more improvements by taking a greedy action, and the Bellman optimality equation is satisfied as given by equation 11.

There exists only a finite number of policies in a given MDP. However, there exists only one optimal deterministic policy for any setting. Since the number of policies is finite, so the policy iteration algorithm steps must also converge after a finite k steps as:

$$V^{\pi_q} = V^{\pi_{q+1}} \tag{12}$$

V^{π_q} is a fixed point of T, and since T has a unique fixed point, we can therefore also deduce that:

$$V^{\pi_q} = V^* \tag{13}$$

and hence π_q is an optimal policy.

5. Question D: SARSA Algorithm

In this section, we consider the on-policy SARSA algorithm for learning the action value function. SARSA is a form of Temporal Difference (TD) learning method where Q function is estimated using the Q evaluated at the next state and action. 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 and can learn online after every step from incomplete episodes. In the on-policy SARSA method, we estimate Q^π for the policy π and improve π greedily with respect toQ^π using an epsilon greedy approach.

In SARSA, at every time step, the policy is evaluated which is then followed by a epsilon greedy policy improvement step, where the Q function is updated as:

$$Q(s,a) = Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)] \tag{14}$$

We consider the one step return SARSA algorithm, which is also guranteed 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:

$$action = \begin{cases} \arg\max_{a'} Q(s', a') - - - w.p 1 - \epsilon \\ Uniform(A) - - - otherwise \end{cases}$$

We also consider the convergence properties of SARSA algorithm later in section 7, by considering the convergence of the cumulative reward objective function. The convergence of the SARSA algorithm depends on the Q function. SARSA algorithm is guaranteed to converge to an optimal Q and hence optimal policy as long as all the state-action pairs in the environment are visited an infinite number of times. This however depends on the exploration-exploitation tradeoff. By considering a greedy policy with a carefully fine tuned ϵ parameter, we can balance the exploration-exploitation, such that the states action pairs are visited large number of times. In section 5.1 below, we show results of the SARSA algorithm on the small-world MDP. We here show results for both fixed and decaying α step size and greedy exploration ϵ parameters.

The **MATLAB** code for SARSA is given below:

```
% sarsa on SmallWorld
smallworld;
[v, pi, ~] = sarsa(model, 1000, 1000);
plotVP(v,pi, paramSet)


%SARSA Algorithm
function [v, pi, Cum_Rwd] =
sarsa(model, maxit, maxeps)

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

for i = 1:maxeps,
%every time we reset the episode,
start at the given startState
%get Start State
s = model.startState;
%OR INITIALIZE ACTION ARBITRARILY
a = 1;
%% initialize the first action
%%greedily as well
%%a = epsilon_greedy_policy(Q(s,:));

%FOR EACH STEP OF EPISODE
for iter = 1:maxit,
    p = 0;
    r = rand;
    for next_state = 1:model.stateCount,
        p = p + model.P(s, next_state, a);
        if r <= p,
            break;
        end
    end

%TAKE ACTION, OBSERVE S' AND R
s_ = next_state;

%get R with given a
reward = model.R(s,a);
 %taking discounted cum rewards
Cum_Rwd(i) = Cum_Rwd(i) + model.gamma * reward;

%CHOOSE A' FROM S' USING GREEDY POLICY
a_ = epsilon_greedy_policy(Q(s_,:), iter);
alpha = 1/iter;

% IMPLEMENT THE UPDATE RULE FOR Q HERE.
Q(s,a) = Q(s,a) + alpha *
[reward + model.gamma * Q(s_, a_) - Q(s,a)];
s = s_;
a = a_;

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

    if s == model.goalState
        break;
    end
end
end

pi = policy;
v = q;
end
```

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

```
function action = epsilon_greedy_policy(Q, iter)
all_actions = [1 2 3 4];
epsilon = 1/iter;
probability = rand();

if probability < (1 - epsilon)
    [~, action] = max(Q);
else
    action = all_actions(randi(length(all_actions)));
end
end
```

Explanation of Code: In the SARSA code above, we use an adaptive learning rate with $\alpha = 1/iter$, with SARSA run for 1000 iterations and 1000 steps within each episode. The function *epsilon greedy policy* is the function to take either the random exploratory action given by $action = all_actions(randi(length(all_actions)))$; or the exploitation action given by $max(Q)$, the choice of which depends on whether the probability is less than $1 - \epsilon$. The variable $CumRwd(i)$ keeps count of the total discounted sum of rewards, which is further used to plot how the cumulative reward varies with the number of episodes.

5.1. Experimental Results

For the SARSA algorithm, we use $Maxit = 1000$ steps within each an episode, and use $MaxEps = 1000$ episodes in total for ensuring convergence of the Q function. Also note that, in our algorithm, instead of initializing the action arbitrarily with $a = 1$, we can use an epsilon-greedy action at the beginning of each episode as well. This ensures that in subsequent episodes, the first action is also chosen greedily in the first step of each episode.

Figure 3 shows SARSA algorithm on the SmallWorld MDP with decaying values of step sizes and epsilon parameters. Figure 4 further uses fixed step sizes and epsilon parameters. The significance of these two choices, and its importance is further discussed in section 5.2 below.

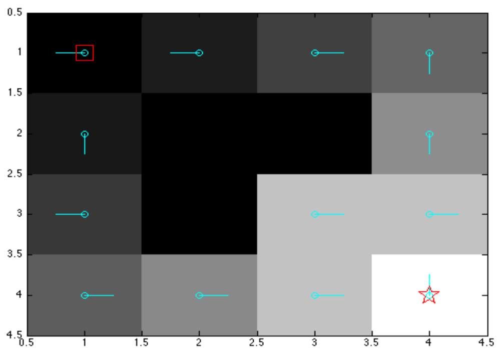

Figure 3. SARSA Algorithm on Small World Model, Decaying $\epsilon = 1/iter$ and $\alpha = 1/iter$ parameters

Figure 4 further shows how the value functions and policy at every state changes with fixed step size and epsilon parameters, instead of decaying values.

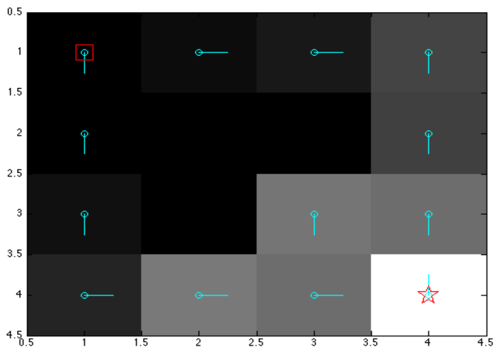

Figure 4. SARSA Algorithm on Small World Model, $\epsilon = 0.4$, $\alpha = 0.2$

5.2. Discussion of Results

The results in figure 3 are using an decaying epsilon greedy parameter $\epsilon = 1/iter$, in which the ϵ parameter decays with the number of steps in each episode. The α parameter is also chosen to be decaying as $\alpha = 1/iter$. This is practical since it means that 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 encoured 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 step size 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 (similar to doing stochastic gradient ascent). However, at later stages of a given episode, smaller values of α ensures that the optimal Q function can be reached. **Note:** In RL, we always consider trying to achieve the highest local optima of Q function (or $J(\theta)$ in case of policy gradient methods), since achieving the globally optimal policy is quite difficult due to the non-convex functions.

Figure 4 further shows results when choosing a constant ϵ and α parameters. A constant ϵ value means that we are encouring 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 small world MDP. In a small world MDP with small state space compared to grid world, exploration at later stages of an episode should be less encouraged since the agent must have visited all the states already (we have also chosen 1000 steps within an episode with a breaking condition of ending the episode once the goal state has been reached). A fixed α value also does not necessarily ensure convergence to optimal Q function since a small step size 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 therefore see that a decaying ϵ and α parameter should be more feasible. Figure 4 shows lower value functions even near the goal state (darker regions) compared to figure 3. Figure 3 shows higher value functions near the goal states (lighter regions near goal state) which suggests better convergence of Q function with decaying parameter values.

5.3. Extension: 2-Step SARSA Algorithm

In this section, we include an extension from the SARSA algorithm (not part of the coursework), in which we consider the 2-step SARSA algorithm. Unlike 1-step SARSA, where we consider $R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ for the Q function update, here we consider taking two steps forward such that the Q function is updated with respect to $R_{t+1} + \gamma R_{t+2} \gamma^2 Q(s_{t+2}, a_{t+2})$ such that the Q function update becomes:

$$Q(s,a) = Q(s,a) + \alpha[R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(s_{t+2}, a_{t+2}) - Q(s,a)] \tag{16}$$

Here, we consider showing results of the 2-step SARSA algorithm on the GridWorld MDP (to analyze SARSA on larger state space), using decaying ϵ and α parameters, with 1000 episodes, and 1000 steps within an episode. In later section, we will also compare how the cumulative reward changes with the number of episodes for the 2-step SARSA algorithm. Below, we present the results for the 2-step SARSA on the grid world model.

Compared to SARSA, the following snippets of code are the only difference for the 2-STEP SARSA algorithm.

```
s_ = next_state;
```

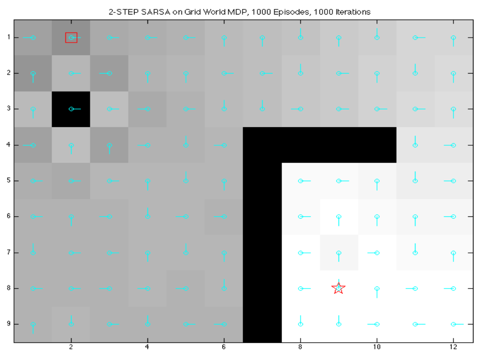

Figure 5. Extension: 2-Step SARSA Algorithm on Grid World Model

```
reward = model.R(s,a);
a_ = epsilon_greedy_policy(Q(s_,:), maxit);

        % 2nd step of SARSA
for two_step_state = 1:model.stateCount,
p_2 = p_2 + model.P(s_, two_step_state, a_);
    if r_q <= p_2,
        break;
    end
end

s_2 = two_step_state;
reward_2 = model.R(s_2,a_);
a_2 = epsilon_greedy_policy(Q(s_2,:), maxit);

Q(s,a) = Q(s,a) +
alpha/iter * [reward + model.gamma* reward_2
+ model.gamma.^2 * Q(s_2, a_2) - Q(s,a)];
```

5.4. Extension 2: SARSA(λ) Algorithm

We then considered another extension to implement the SARSA(λ) algorithm which considers combining all the n-step returns and uses a weight of $(1-\lambda)\lambda^{n-1}$ for the Forward-View Sarsa (λ) algorithm. The λ parameter in SARSA (λ) is another parameter to fin-tune (not considered here). Results of SARSA (λ) on the gridworld MDP is shown in figure 6 below.

In SARSA (λ), an eligibility tree is considered to keep count of the states and actions that have been visited in an episode. The Q function is then updated with respect to the eligibility tree as well as shown by the code snippet below. Details not discussed here due to limit considerations.

```
Delta = reward +
(model.gamma * Q(s_, a_) - Q(s,a));
```

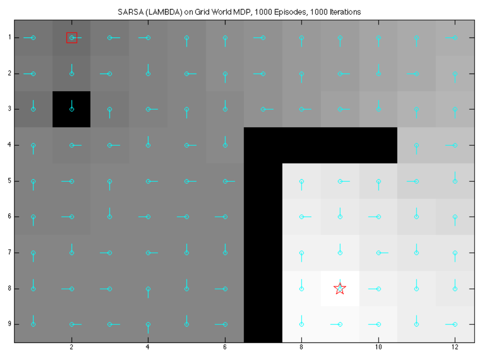

Figure 6. Extension: Sarsa (λ) on Grid World Model

```
eligibility(s,a) = eligibility(s,a) + 1;

for m = 1:model.stateCount
    for n = 1:4
Q(m,n) = Q(m,n) +
alpha/iter * Delta * eligibility(m,n);

eligibility(m,n) = model.gamma *
lambda * eligibility(m,n);
    end
end
```

6. Question E: Q-Learning Algorithm

We then consider off-policy learning of action-values Q(s,a) where Q(s,a) direclty approximates Q^* independent of the policy being valued. This means that even though the policy π determines which states and actions are visited, the Q function is updated directly towards a value of alternative action. The next action is chosen using a behaviour off-policy while we consider an alternative successofr action and update the Q(s,a) towards value of the alternative action. The behaviour policy still follows an epsilon greedy policy for exploration. However, the target policy is now greedy with respect to Q(s,a). In other words, the behaviour policy is determined by epsilon-greedily ensuring sufficient exploration, while the target policy is obtained by $\arg\max_{a'} Q(s_{t+1}, a')$.

The Q learning target therefore simplifies as:

$$TargetQ = R_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a')) \tag{17}$$

which therefore further means:

$$TargetQ = R_{t+1} + \max_{a'} \gamma Q(s_{t+1}, a') \tag{18}$$

The **MATLAB** code for our implementation of Q-learning is given below:

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

for i = 1:maxeps,
% every time we reset the episode,
%start at the given startState
s = model.startState;
%initialize a arbitrarily
a=1;

%repeat for each step of episode
for j = 1:maxit
    p = 0;
    r = rand;

for s_ = 1:model.stateCount,
    p = p + model.P(s, s_, a);
        if r <= p,
            break;
        end
    end
%get action from behaviour policy
%- epsilon_greedy wrt Q(s,a)

%action from behaviour policy
a_ = epsilon_greedy_policy(Q(s, :), j);
%take action, observe r
Reward = model.R(s,a);
Cum_Rwd(i) = Cum_Rwd(i)
+ model.gamma * Reward;

TargetQ = Reward +
model.gamma * max(Q(s_, :));

alpha = 1/j;
Q(s,a) = Q(s,a) +
alpha * ( TargetQ - Q(s,a) );

s = s_;
a = a_;
```

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

% SHOULD WE BREAK OUT OF THE LOOP?
    if s == model.goalState
    break;
    end
end

end

pi = policy;
v = q;
end
```

Explanation of the Code: The Q-learning code uses an off-policy to explore the environment, and the Q function update is taken with respect to maximizing the off-policy action as given in the code above below. Note again we use a learning rate for the Q funciton update which depends on the number of iterations. This is to ensure better convergence of the Q function, as the equation below is same as doing gradient ascent, but for Q functions in tabular form. The update of Q function being the same as gradient ascent, and hence the requirement to choose learning rates properly, is more apparent when considering Q function approximation with parameters w, instead of tabular forms.

```
TargetQ = Reward +
model.gamma * max(Q(s_, :));

alpha = 1/j;
Q(s,a) = Q(s,a) +
alpha * ( TargetQ - Q(s,a) );
```

6.1. Experimental Results

Below we present the results for learning the value function and policy using Q-learning on both small-world MDP. For our experiments, we again used decaying $= 1/iter$ and $\alpha = 1/iter$ parameters for reasons explained earlier. Figure 7 shows the value function and policy obtained in the smallworld MDP by off-policy Q-learning. For our experiments, we again used 1000 episodes and 1000 steps within each episode.

6.2. Discussion of Results

In Q-learning, one key issue of convergence is that all state action pairs need to be continually updated, since the learned Q function directly approximates to Q^*.

In section 7 below, we compare the performance of our Q learning and SARSA algorithm on the cliffworld MDP, by analyzing how the cumulative rewards obtained for each episode varies for the on-policy and off-policy algorithms.

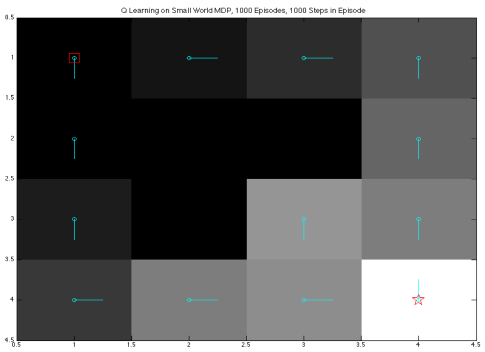

Figure 7. Q-Learning on Small World MDP

In section 7 we further argue that the ϵ exploration parameter has a greater influence in Q-learning compared to SARSA methods when we compare cumulative rewards over number of episodes. We show that exploration-exploitation plays a more significant role in off-policy Q-learning compared to on-policy SARSA methods. This is because the target Q function in Q function update involves maximizing over the off-policy actions, and hence more exploration of the unknown environment has significance in convergence of Q function to an optimal policy.

7. Question F: SARSA vs Q Learning

In this section, we compare the cumulative rewards obtained from each episode for both the SARSA and the Q-Learning 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 $J(\pi)$ shows that using SARSA and Q-Learning, we can also converge to an optimal Q function. When $J(\pi)$ becomes constant or steady, it indicates that the optimal policy has been achived, such that with more episodes, no other better policy can be achieved.

The MATLAB code below shows the how the discounted cumulative reward is calculated with number of episodes (also shown previously in code for SARSA and Q-Learning).

```
Cum_Rwd(i) = Cum_Rwd(i)
+ model.gamma * reward;
```

Notice that the epsilon parameters determines the amount of exploration and exploitation in the unknown environment. Additionally, the step size α in the update of the action-value function determines how quickly we Q function converges to an optimal Q^*. Convergence of the Q function determines how quickly we can converge to an optimal policy. Hence fine tuning of the ϵ and α parameters is required to achieve good performance on the cliffworld, due to the higher state space in this model. Additionally, since the updates of the Q function depends on the random greedy action and a maximization over the Q function, the cumulative reward achieved with SARSA or Q-learning is not always smoothed out.

In order to take account of the above issues, we ran our experiments with different combinations of ϵ and α parameters. For each experiment, we ran the same experiment with the same number of episodes and iterations within each episode for upto 50 times and then took the average cumulative reward to smooth out the results on the cliffworld.

Below we also present our results 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 $a = \arg\max_a Q(s', a')$ with probability $1 - \epsilon$ is chosen with a higher probability. Compared to that, if ϵ values are too high, then this leads to more exploraation of the state space and less exploitation of the states that have already been visited.

7.1. Experimental Results

First, we present the results in figure 8 comparing Q-learning and SARSA algorithm on the cliffworld MDP given by figure 8. The result below in figure 8 is obtained using an $\alpha = 0.2$ and $\epsilon = 0.4$ and experiments averaged over 50 iterations to smooth out. We used 500 episodes and 500 iterations or steps within each episode for both SARSA and Q-Learning for all the 50 iterations. The $\epsilon = 0.4$ ensures a balanced trade-off between exploration and exploitation, and this is validated further by the results below. Notice that, unlike before, we used 500 iterations since the cumulative reward scale depends on the number of iterations within each episode (reward of -1 for every transition). The figures show the discounted cumulative reward with the number of episodes.

The lower part of the figure shows the performance of the Sarsa and Q-learning methods with ε-greedy action selection. After an initial transient, Q-learning

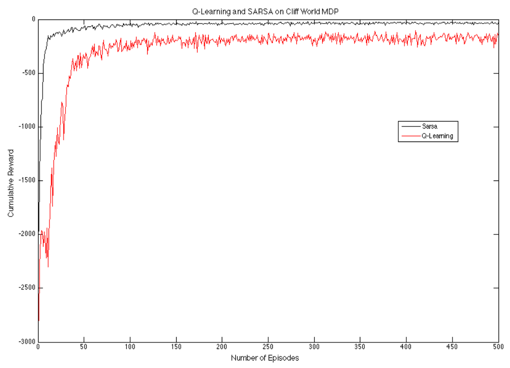

Figure 8. Cumulative Rewards: Q-Learning and SARA on CliffWorld MDP

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 into account 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, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.

We also evaluated our Q-learning and SARSA algorithms with varying ϵ and α parameters and compared how the convergence of the Q function and hence cumulative reward values depend on it. Figure 9 below shows results with decaying ϵ and α parameters.

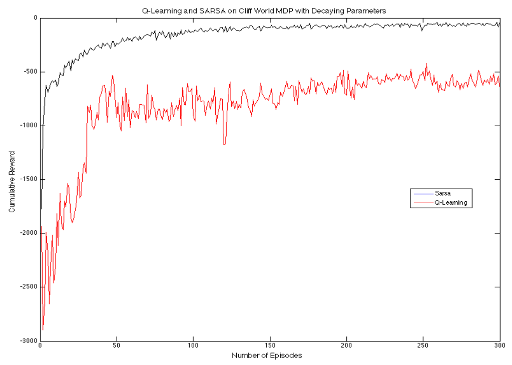

Figure 9. Cumulative Rewards: Q-Learning and SARA on CliffWorld MDP with Decaying ϵ and α

We then present the results of how the performance of the agent for finding an optimal Q and hence optimal policy, in both Q-learning and SARSA is dependent on the value of step size α and exploration parameter ϵ. In figure 10 and 10, an epsilon value of 0.6 is used with varying step sizes α parameters. We varied the same number of step sizes, and notice significant differences in how the ϵ parameter has a higher effect during Q-learning compared to SARSA algorithm.

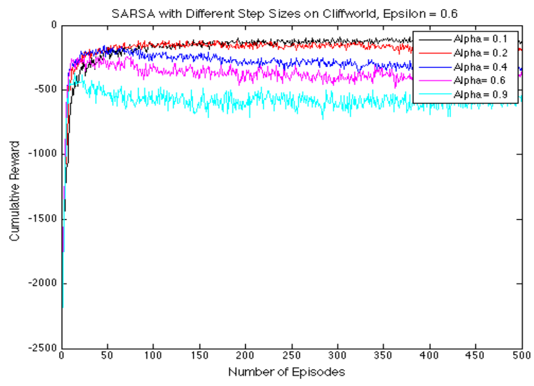

Figure 10. SARSA with $\epsilon = 0.6$ with varying step sizes on CliffWorld MDP

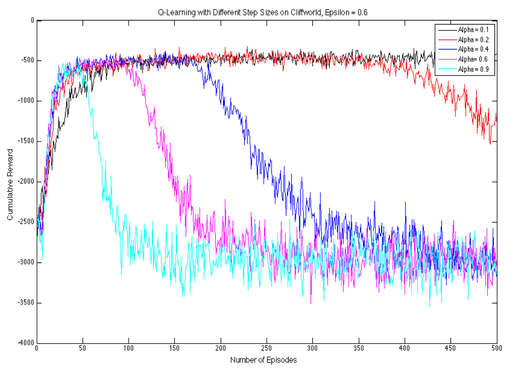

Figure 11. Q-Learning with $\epsilon = 0.6$ with varying step sizes on CliffWorld MDP

Finally we present the results with an ϵ parameter of 0.1 for both the SARSA and Q-Learning. A small value of epsilon encourages more exploitation of the state space. Figures 12 and 13 shows how the exploration-exploitation tradeoff plays a more significance role in off-policy Q-learning.

7.1.1. **Extension**

We briefly include the cumulative rewards obtained from the different types of SARSA algorithms we implemented before. Result in figure 14 shows comparison of the SARSA algorithms. Since the cliffworld is a relatively small MDP, significant differences cannot be observed. We expect that on a larger and higher dimensional state space, the different types of SARSA algorithms will have a more significant effect in convergence to an optimal Q function.

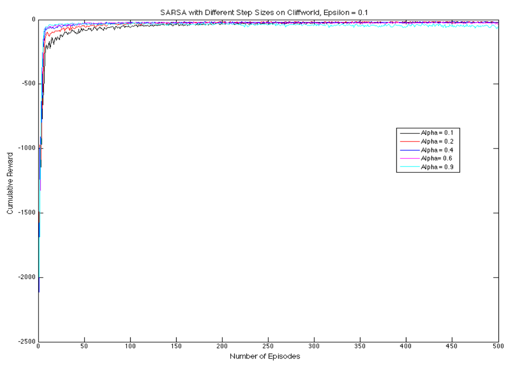

Figure 12. SARSA with $\epsilon = 0.1$ with varying step sizes on CliffWorld MDP

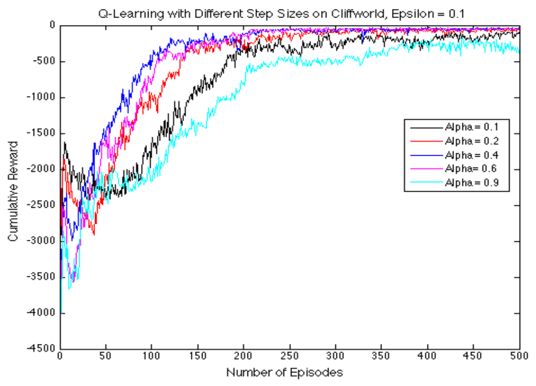

Figure 13. Q-Learning with $\epsilon = 0.1$ with varying step sizes on CliffWorld MDP

7.2. Discussion of Results

The experimental result in figure 8 shows how the learning performance of the agent, and hence the cumulative reward obtained, varies depending on whether the action-value functions are updated using on-policy or off-policy. Figure 8 shows that SARSA learns better and is good at avoiding the bad states compared to Q-learning. This is because the Q function in SARSA is updated w.r.t to the on-policy that maximizes the Q function. In other words, 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 w.r.t an off-policy without using actions

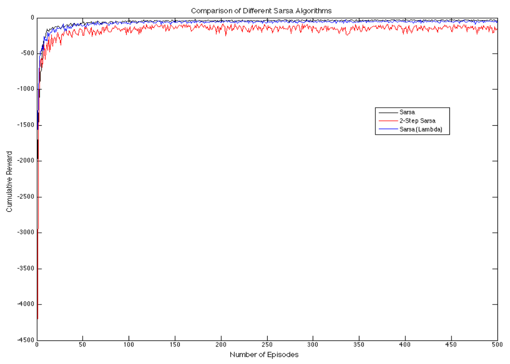

Figure 14. Comparison of 1-Step SARSA, 2-Step SARSA and SARSA(λ) Algorithms

selected by the exploratory policy. In other words, the Q function update is made by maximizing the actions from off-policy, ie, maximizing over the action that depends on the exploration-exploitation trade-off.

Compared to Q-learning, SARSA learns directly from episodes of experience and can learn from incomplete episodes, and hence we 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.

Figure 8 therefore justifires that Q-learning is more concerned with the exploration of the environment, and it does not care about the rewards it obtains while it is learning. During learning, Q-learning tries to update Q depending on the exploratory actions from the behaviour off-policy, which further makes it more prone to get into bad states while it is learning - hence achieving lower cumulative reward while it is learning. However, 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^* under sufficient conditions.

Also, note that the cumulative reward scale starts from very large negative values. This is because at the beginning when the agent has not learned about the environment, it is more likely for it to fall down the cliff which has a reward of -100. The larger the number of iterations used within each episode, due to exploratory actions, the agent is often likely to fall off the cliff. However, as the learned Q function improves, the agent starts avoiding falling off the cliff.

Figure 9 further shows how the Q-learning and SARA 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. Figure 9 shows that Q-learning is more heavily affected if the ϵ parameter is not well-tuned. This is because since the Q-function updates in Q-learning is more dependent on the exploratory off-policy, hence the amount of exploration-exploitation has more effect in Q-learning. Figure 9 shows 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 10, 11, 12 and 13. Comparing figure 11 and 13, figure 11 shows that even if the α step-size parameter is carefully fine-tuned, the large ϵ parameters makes the agent more prone to going into bad stages, compared to figure 13 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. Compared to that, figure 10 and **??** always shows that with proper step-sizes, the cumulative reward is less affected by the ϵ parameters.

Our results therfore proves 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 higheer dependence on the exploration-exploitation tradeoff, it is more prone to falling into the bad states in the environment. We have also shown that the ϵ exploration paramter has a greater significance in off-policy learning since the Q function updates is done w.r.t exploratory actions in Q-learning. Our results have also examined the influence of well-tuned α and ϵ parameters to achieve good learning performance.

8. Summary

We therefore examined the convergence and performance of policy and value iteration algorithms, and discuss how the convergence of these algorithms to the optimal value function depends on the number of iterations used. Furthermore, we evaluated the difference between on-policy SARSA and off-policy Q-learning algorithms and showed how the performance of these algorithms depends on the exploration-exploitation tradeoff, and on learning rates. Our experiments were evaluted on benchmark reinforcement learning tasks such as a smallworld, gridworld and a cliffworld MDP to analyze the performance of our algorithms.