UNIVERSITY OF CAMBRIDGE

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

MODULE COURSEWORK FEEDBACK

Student Name: Max Chamberlin

Module Title: Reinforcement Learning

CRSiD: MEC68

Module Code: MLSALT7

College: Clare

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

Maximilian Chamberlin

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.

Question a: Value Iteration

The algorithm is based on the pseudo-code given in the lectures, where we do an in place policy update. Below, I give the pseudo-code for clarity. The algorithm below implements this pseudo-code, with the different parts marked in the comments, e.g. (1a) or (1b).

1. Initialise v, π
2. Repeat until maxit:
   (a) For each state s:
      i. Compute the new value: $v(s) := (\mathcal{T} * v)_s = \max_{\mathrm{a}}[\mathrm{r(s,a)} + \gamma \mathrm{P}_{\mathrm{as}}^{\mathrm{T}} v]$, (by applying the optimal Bellman operator)
      ii. Do an in-place policy update $\pi(\mathrm{s}) = \underset{\mathrm{a}}{\mathrm{argmax}}[\mathrm{r(s,a)} + \gamma \mathrm{P}_{\mathrm{as}}^{\mathrm{T}} \mathrm{v}]$
   (b) If the value function has sufficiently converged [i.e norm(v-v_)< tolerance threshold] then exit early

Q1: Value Iteration

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

% (1) initialize the value function
v = zeros(model.stateCount, 1);
% initialise the set of actions
actions=1:4;
% initialize the policy and the previous value function v_
pi = ones(model.stateCount, 1);
v_ = zeros(model.stateCount, 1);

% (2) Main Loop
for i = 1:maxit,
    % (a) Loop over states
    for s = 1:model.stateCount,
        % (i & ii): COMPUTE THE VALUE FUNCTION AND POLICY
        % find the value of each action from the current state
        for a=actions
             rwd(a) = model.R(s,a)+model.gamma*sum(model.P(s,:,a)*v_);
        end
        % Perform the Bellman update for each state, by taking the best value action
        % Update value and policy of a given state.
        [v(s), pi(s)]=max(rwd);
    end

    % (b). EXIT EARLY
    % Sufficiently converged?
    if norm(v-v_)<0.000001
        display(['terminated at ',num2str(i)]);
        break;
    end
    % the old value is now the current value
    v_=v;

end
```

Below, I plot the actions and value functions for the gridWorld that my valueIteration algorithm produced.

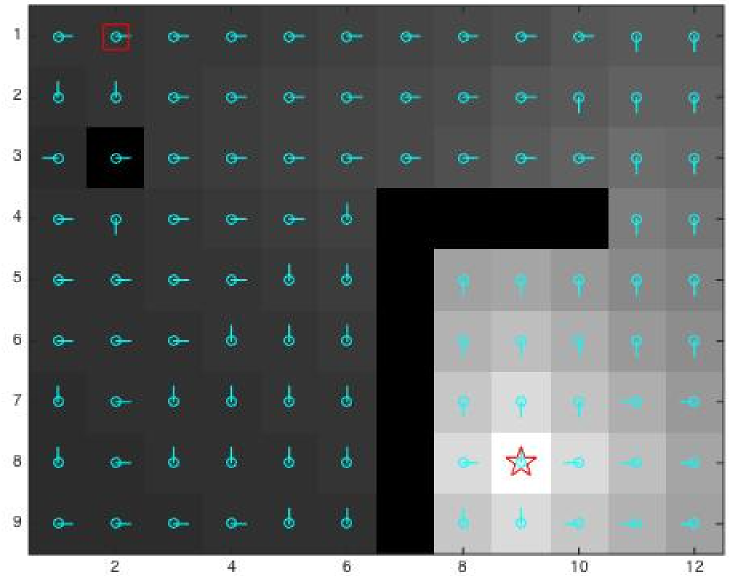

Figure 1: Plot of Values and Actions for the Value Iteration Algorithm

Question b: Policy Iteration

The algorithm is based on the pseudo-code given in Sutton, where the steps (a) and (b) below are reversed from the lectures. However, this makes little practical difference to the running of the algorithm, since the important aspect of the algorithm is that policy updates are interleaved with policy evaluations.

1. Initialise $v, \pi, \neq \pi\_$
2. While policies have not converged ($\pi \neq \pi\_$)
   (a) Evaluate the policy:
      i. This involves repeating: $v := \mathcal{T}^{\pi} v$, until sufficient convergence
   (b) For each state s:
      i. Update the policy $\pi(\mathrm{s}) = \underset{\mathrm{a}}{\mathrm{argmax}}[\mathrm{r(s,a)} + \gamma \mathrm{P}_{\mathrm{as}}^{\mathrm{T}} \mathrm{v}]$

Q2: Policy Iteration

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

% (1) initialize the value function, and the preceeding v_
v = zeros(model.stateCount, 1);v_ = zeros(model.stateCount, 1);
% initialize the policy, and the preceeding policy p_
pi = ones(model.stateCount, 1); pi_=nan(model.stateCount,1);
% The actions
actions=1:4;

% (2) Loop until Policy Convergence
while ~isequal(pi_,pi)
  % (a) Evaluate the Policy
  evalPolicy()

  % (b) Update the policy for each state s
  % The previous policy is now the current policy, since we are about to update
```

```
  pi_=pi;
  % The full update:
  for s = 1:model.stateCount,
        for a =actions
            rwd(a)= model.R(s,a)+model.gamma*sum(model.P(s,:,a)*v_);
        end
        [~, pi(s)]=max(rwd);
  end

end

% This function applies the Bellman operator for a policy, until the value converges
function[]= evalPolicy()
    for i = 1:maxit,
        for s1 = 1:model.stateCount
            v(s1)=model.R(s1,pi(s1))+model.gamma*sum(model.P(s1,:,pi(s1))*v_);
        end
        if norm(v-v_)<0.001 return; end;
        v_=v;
    end
end

end
```

Below, I plot the actions and value functions for the grid-world that my policyIteration algorithm produced. The general policy iteration and value iteration is very similar since the two figures are very alike, both in policy and value. This is unsurprising since both methods converge due to the Bellman operator.

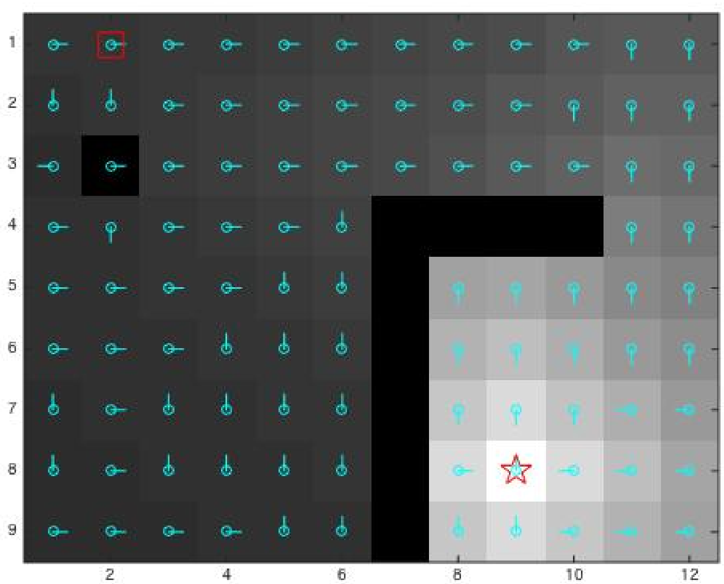

Figure 2: Policy Iteration: GridWorld

Question c: Mathematical Proof

Before I proceed to give the full proof, it is instructive to consider slightly different pseudo-code for policy iteration.

1. Start with policy π_0

2. at each k, given the current policy π_k
   (a) Evaluation step: we find the value of v^{π_k} by finding the unique fixed point of $v = \mathcal{T}^{\pi_k} v$
   (b) Policy update: we calculate the new policy $\pi_{k+1} = \text{argmax}_a \mathcal{T} v^{\pi_k}$
   (c) (NB: this means the policy π_{k+1} is greedy wrt the value function: $\mathcal{T}^{\pi_{k+1}} v^{\pi_k} = \mathcal{T} v^{\pi_k}$)

First, I will prove that $v^{\pi_{k+1}} \geq v^{\pi_k}$

From the pseudo code given, we can see:

$$\begin{aligned} v^{\pi_k} &= \mathcal{T}^{\pi_k} v^{\pi_k} && \text{By (2a) the evaluation step} \\ &\leq \mathcal{T} v^{\pi_k} && \text{By definition of optimal Belman operator} \\ &= \mathcal{T}^{\pi_{k+1}} v^{\pi_k} && \text{By greediness of policy updater (2b)} \end{aligned}$$

From this fact, and the monotonicity of the Bellman operator*,we can show:

$v^{\pi_k} \leq \mathcal{T}^{\pi_{k+1}} v^{\pi_k} \leq (\mathcal{T}^{\pi_{k+1}})^2 \ v^{\pi_k} \leq ... \leq (\mathcal{T}^{\pi_{k+1}})^n \ v^{\pi_k} \leq lim_{n\to\infty} (\mathcal{T}^{\pi_{k+1}})^n \ v^{\pi_k} = v^{\pi_{k+1}}$

QED

Second, since there are only a finite number of policies, and the value of v^{π_k} increases at each step, this is enough to prove convergence. However, we can go a step further. The algorithm terminates at step q with $v^{\pi_q} =$ (by fact policy is terminal)$= \mathcal{T}^{\pi_{q+1}} v^{\pi_q} =$(by 2a) $= \mathcal{T} v^{\pi_q}$. Thus, at termination v^{π_q} is the unique fixed point of the Bellman operator.

*The proof of monotonicity is given here: http://lhendricks.org/econ720/ih1/DP_SL.pdf

Question d: Sarsa Algorithm

The pseudo-code for Sarsa is based upon the implementation described on slides (3-4). The coded parts of the main body, steps (1-4) exactly match the main body of the Sarsa pseudo-code given in the lectures on slide 14. The only difference is that an early stopping rule is also included in case the value function has converged.

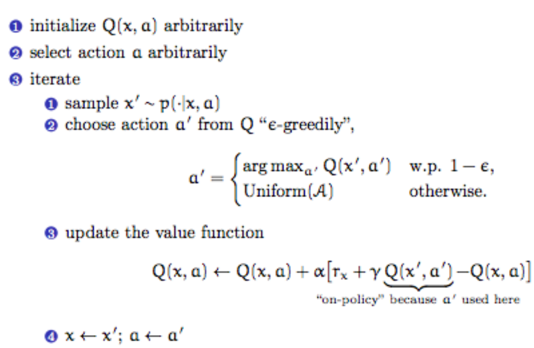

Figure 3: Sarsa Pseudo-Code from Lectures

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

% initialize the value function
Q = zeros(model.stateCount, 4);
% random
epsilon =0.4;
%epsilon=0.95;
alpha=0.2;

% REPLACE THESE
v = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
```

```
reward = zeros(maxeps,1);
Q      =zeros(model.stateCount,4);

for i = 1:maxeps

    s = model.startState;
    a =pickAction(s);



    % For each step of episode
    for j = 1:maxit,

        %1) sample s_ from p(.|s,a)
        cdf = cumsum(model.P(s,:,a));
        U = rand;
        s_ = sum(cdf<U)+1;

        %2) PICK AN ACTION arbitrarily (decided epsilon-greedily)
        a_=pickAction(s_);

        %Take action, Observe R, S1
        r=model.R(s,a);

        %3) Update Q(S,A)
        Q(s,a) = Q(s,a) + alpha*(r+model.gamma*Q(s_,a_)-Q(s,a));
        reward(i) = reward(i) + model.R(s,a);

        %3.5) Do in place policy and value updates
        pi(s)=a;
        v(s) = Q(s,a);

        %4) update state and action
        % reset a and s
        s = s_; a=a_;

        % SHOULD WE BREAK OUT OF THE LOOP?
        if s==model.goalState,
            display('Reached Goal State');
            break
        end;



    end
end


function [a]= pickAction(s)
    if rand<=epsilon,
    %if rand <=epsilon^i
        a = randi(4);
    else
        [~, a] = max(Q(s,:));
    end
end

end
```

As a note, I also tried a very simple based geometric cooling schedule for the epsilon. This is where we start epsilon at 0.97, and then as we move forward to later episodes k, we pick an action at random with probability ϵ^k. These changes are commented out in the code above. Below, I show the performance of Sarsa on small world.

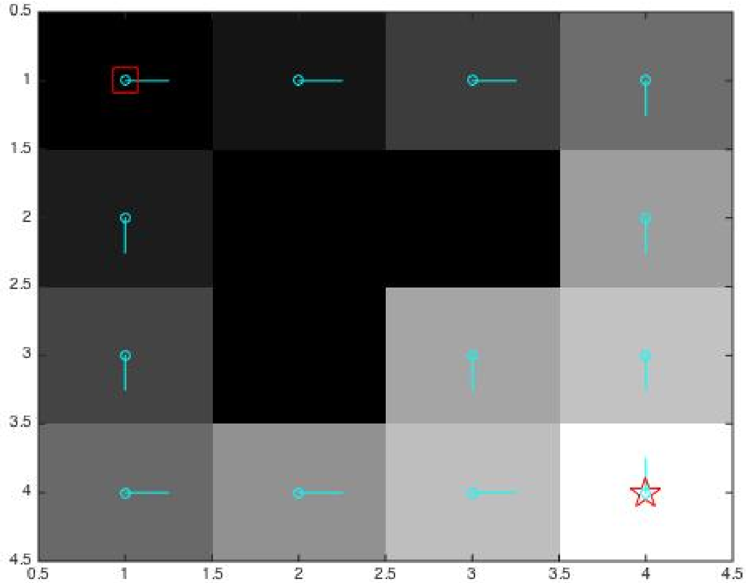

Figure 4: Action and Values for: Sarsa

Question e: qLearning

The pseudo-code for qLearning is based upon the implementation described in the slides. The coded parts of the main body. The only difference is that an early stopping rule is also included in case the value function has converged.

1. initialize $Q(x, a)$ arbitrarily
2. select action a arbitrarily
3. iterate
   1. sample $x' \sim p(\cdot|x, a)$
   2. choose action a' from Q "ϵ-greedily",
   3. update the value function

      $$Q(x,a) \leftarrow Q(x,a) + \alpha\big[r_x + \underbrace{\gamma \max_{a''} Q(x', a'')}_{\text{“off-policy” due to } a''} - Q(x,a)\big]$$

   4. $x \leftarrow x'$; $a \leftarrow a'$

Figure 5: qLearning Pseudo-Code from Lectures

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

% initialize the value function
Q = zeros(model.stateCount, 4);
% random
epsilon =0.4;
alpha=0.2;

% REPLACE THESE
v = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
reward = zeros(maxeps,1);
```

```
Q       =zeros(model.stateCount,4);

for i = 1:maxeps

    s = model.startState;
    a =pickAction(s);



    % For each step of episode
    for j = 1:maxit,

        %1) sample s_ from p(.|s,a)
        cdf = cumsum(model.P(s,:,a));
        U = rand;
        s_ = sum(cdf<U)+1;

        %2) PICK AN ACTION a epsilon-greedily
        a_=pickAction(s_);

        %Take action, Observe R, S1
        r=model.R(s,a);

        % 3) Update Q(S,A)
        Q(s,a) = Q(s,a) + alpha*(r+model.gamma*Q(s_,a_)-Q(s,a));
        reward(i) = reward(i) + model.R(s,a);

        % 3.5) Do in place policy updates
        pi(s)=a;
        v(s) = Q(s,a);

        %4) update state and action
        % reset a and s
        s = s_; a=a_;

        % SHOULD WE BREAK OUT OF THE LOOP?
        if s==model.goalState,
            display('Reached Goal State');
            break
        end;



    end
end


function [a]= pickAction(s)
    if rand<=epsilon,
        a = randi(4);
    else
        [~, a] = max(Q(s,:));
    end
end

end
```

Below, I show the performance of Sarsa on small world.

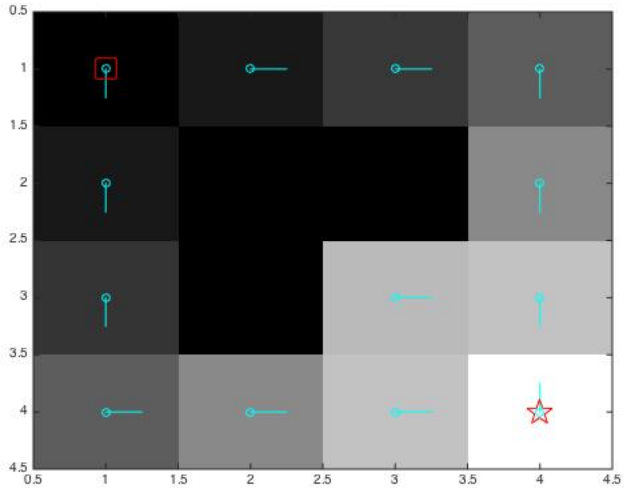

Figure 6: Actions and Values for QLearning on smallWorld

Question f: qLearning

The code given above for QLearning and Sarsa contains the modification made to store thew cumulative rewards. Below, I present two sets of cumulative reward graphs and action-value plots. The first set keeps the reward for falling off the cliff at -6, whereas the second set modifies this to -100. I did this so that the difference between the cumulative rewards for qLearning and Sarsa would become more apparent.

Reward -6:

As we can see from the figures below, qLearning is content to walk very close to the edge of the cliff, when finding a path to the goal state. However, Sarsa takes an aversion to walking by the cliff's edge. The cumulative rewards are roughly equal for the two algorithms are roughly the same because falling off the cliff's edge only diminishes rewards by a small amount in this first setting.

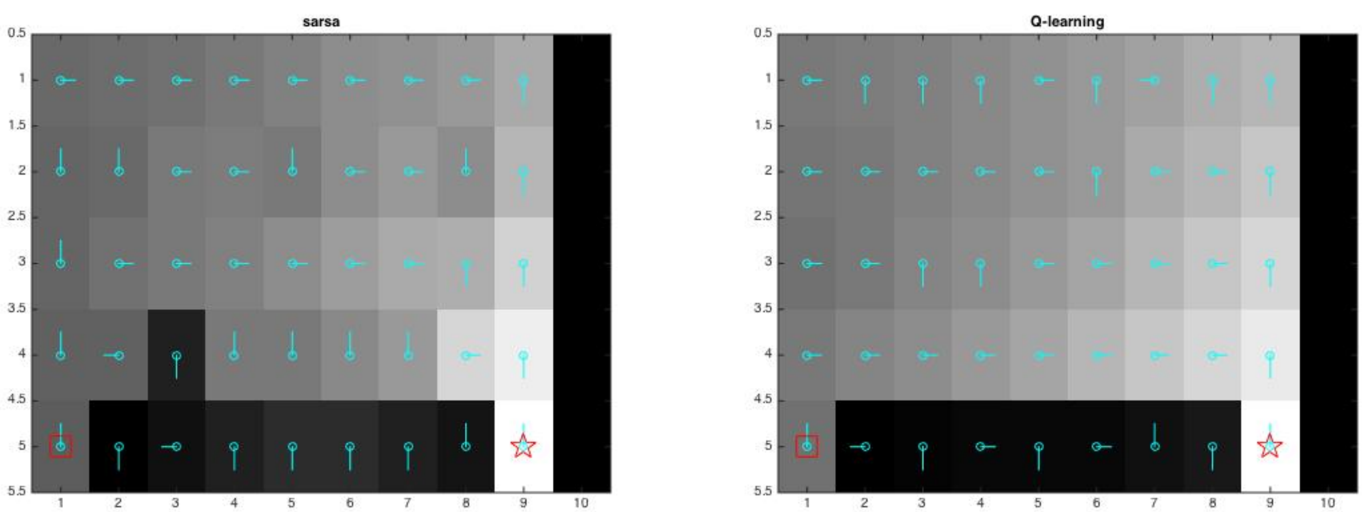

Figure 7: Sarsa

NB: To obtain the cumulative reward curves below, I smoothed over the initial rewards from each episode.

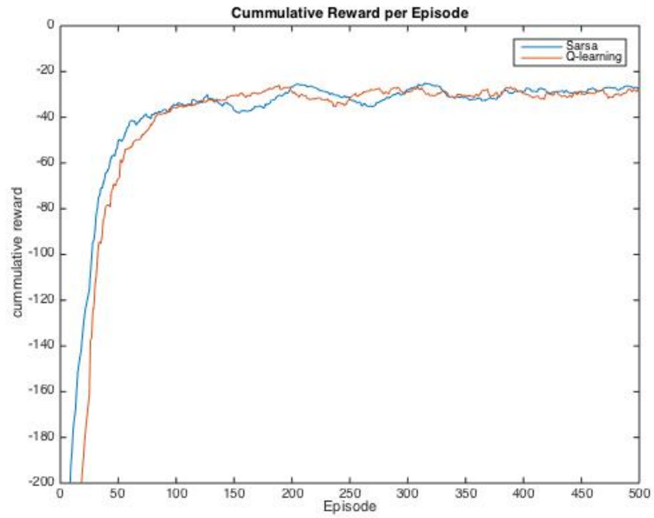

Figure 8: Sarsa

Reward -300:

In this second setting, I much ore heavily penalised walking off the cliff's edge. The strategies of the two algorithms remain broadly the same. However, now it can be seen from the cumulative reward curves that Sarsa's strategy outperforms that of qLearning.

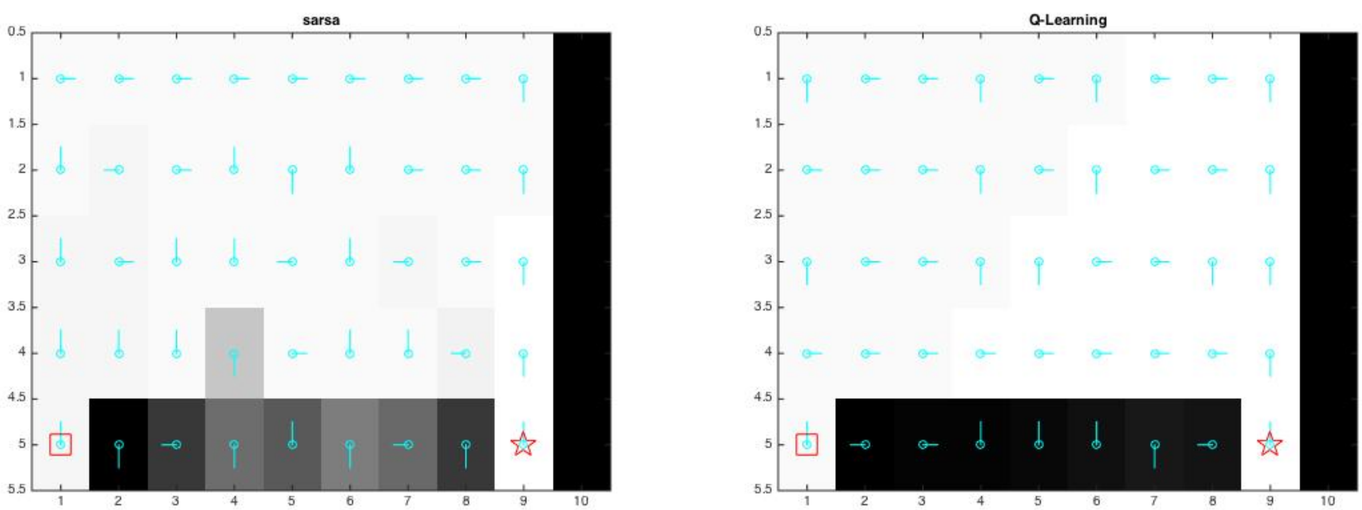

Figure 9: Sarsa

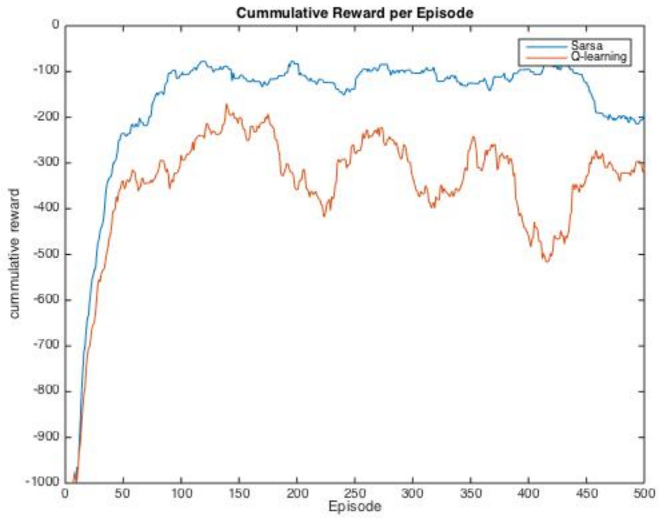

Figure 10: Sarsa

The reason for this behaviour is that Qlearning in on-policy, and so Q-learning may not necessarily care about rewards it gets while it's learning . So, if we choose bad actions due to the noisy exploration, this does not affect the value function learned by Q-learning, since we take the best action through the argmax. This means the random steps off the cliff's edge while learning will not necessarily affect the optimal policy learned.

However, Sarsa is off-policy, and so updates to Q(s,a) are performed using actions selected by the exploratory policy. And so the random steps we might take off the cliff will affect the value of the policy learned.