# Temporal_Difference

June 4, 2020

## 1 Temporal-Difference Methods

In this notebook, you will write your own implementations of many Temporal-Difference (TD) methods.

While we have provided some starter code, you are welcome to erase these hints and write your code from scratch.

---

#### 1.0.1 Part 0: Explore CliffWalkingEnv

We begin by importing the necessary packages.

```
In [1]: import sys
        import gym
        import numpy as np
        from collections import defaultdict, deque
        import matplotlib.pyplot as plt
        %matplotlib inline

        import check_test
        from plot_utils import plot_values
```

Use the code cell below to create an instance of the CliffWalking environment.

```
In [2]: env = gym.make('CliffWalking-v0')
```

The agent moves through a $4 \times 12$ gridworld, with states numbered as follows:

```
[[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
 [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],
 [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
 [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47]]
```

At the start of any episode, state `36` is the initial state. State `47` is the only terminal state, and the cliff corresponds to states `37` through `46`.

The agent has 4 potential actions:

```
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3
```

Thus, $\mathcal{S}^+ = \{0, 1, \ldots, 47\}$, and $\mathcal{A} = \{0, 1, 2, 3\}$. Verify this by running the code cell below.

```
In [3]: print(env.action_space)
        print(env.observation_space)
```

```
Discrete(4)
Discrete(48)
```

In this mini-project, we will build towards finding the optimal policy for the CliffWalking environment. The optimal state-value function is visualized below. Please take the time now to make sure that you understand *why* this is the optimal state-value function.

```
In [4]: # define the optimal state-value function
        V_opt = np.zeros((4,12))
        V_opt[0:13][0] = -np.arange(3, 15)[::-1]
        V_opt[0:13][1] = -np.arange(3, 15)[::-1] + 1
        V_opt[0:13][2] = -np.arange(3, 15)[::-1] + 2
        V_opt[3][0] = -13

        plot_values(V_opt)
```


### 1.0.2 Part 1: TD Control: Sarsa

In this section, you will write your own implementation of the Sarsa control algorithm.

Your algorithm has four arguments: - `env`: This is an instance of an OpenAI Gym environment. - `num_episodes`: This is the number of episodes that are generated through agent-environment interaction. - `alpha`: This is the step-size parameter for the update step. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as output: - `Q`: This is a dictionary (of one-dimensional arrays) where `Q[s][a]` is the estimated action value corresponding to state `s` and action `a`.

Please complete the function in the code cell below.

(*Feel free to define additional functions to help you to organize your code.*)

```
In [13]: env.step(1)
```

```
Out[13]: (36, -100, False, {'prob': 1.0})
```

```
In [81]: def action_epsilon_greedy(state,Q,nA,i_episodes,eps=None):
             epsilon = 1.0/i_episodes
             if eps is not None :
                 epsilon = eps
             prob = np.ones(nA) * (epsilon/nA)
             greedy = np.argmax(Q[state])
             prob[greedy] = 1-epsilon + epsilon/nA
             action = np.random.choice(np.arange(nA),p=prob)
             return action
```

```
In [83]: def sarsa(env, num_episodes, alpha, gamma=1.0,esp=0,eps_decay=0.99,esp_min=0.05):
             # initialize action-value function (empty dictionary of arrays)
             nA = env.action_space.n
             Q = defaultdict(lambda: np.ones(nA)*10)
             # initialize performance monitor
             # loop over episodes
             epsilon = esp
             for i_episode in range(1, num_episodes+1):
                 # monitor progress
                 epsilon = max(epsilon*eps_decay,esp_min)
                 state = env.reset()
                 action = action_epsilon_greedy(state,Q,nA,i_episode)
                 ## Looping though one episode
                 for t in range(200):
                     next_state,reward,done,info = env.step(action)
                     if not done:
                         next_action = action_epsilon_greedy(next_state,Q,nA,i_episode)
                         Q[state][action] = Q[state][action] + (alpha*(reward+ (gamma*Q[next_sta
                         state = next_state
                         action = next_action
                     if done:
                         Q[state][action] = Q[state][action] + alpha*(reward-Q[state][action])
                         break
                 if i_episode % 100 == 0:
                     print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
                     sys.stdout.flush()

                 ## TODO: complete the function

             return Q
```

In [ ]:

Use the next code cell to visualize the ***estimated*** optimal policy and the corresponding state-value function.

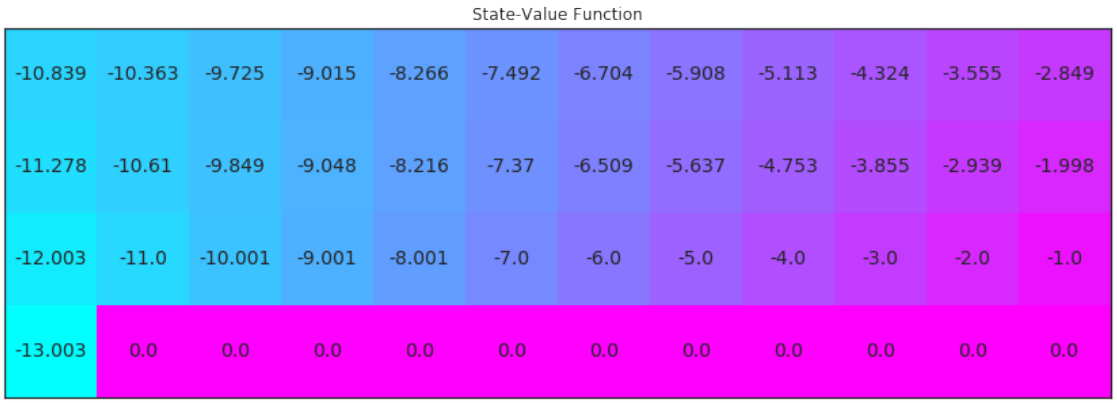
If the code cell returns **PASSED**, then you have implemented the function correctly! Feel free to change the `num_episodes` and `alpha` parameters that are supplied to the function. However, if you'd like to ensure the accuracy of the unit test, please do not change the value of `gamma` from the default.

In [85]:
```
# obtain the estimated optimal policy and corresponding action-value function
Q_sarsa = sarsa(env, 5000, .01)

# print the estimated optimal policy
policy_sarsa = np.array([np.argmax(Q_sarsa[key]) if key in Q_sarsa else -1 for key in n
check_test.run_check('td_control_check', policy_sarsa)
print("\nEstimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):")
print(policy_sarsa)

# plot the estimated optimal state-value function
V_sarsa = ([np.max(Q_sarsa[key]) if key in Q_sarsa else 0 for key in np.arange(48)])
plot_values(V_sarsa)
```

```
Episode 5000/5000
```

**PASSED**

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 2  1  0  1  1  1  1  1  2  1  1  2]
 [ 1  1  3  1  2  1  1  1  1  1  1  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```



### 1.0.3 Part 2: TD Control: Q-learning

In this section, you will write your own implementation of the Q-learning control algorithm.

Your algorithm has four arguments: - `env`: This is an instance of an OpenAI Gym environment. - `num_episodes`: This is the number of episodes that are generated through agent-environment interaction. - `alpha`: This is the step-size parameter for the update step. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as output: - `Q`: This is a dictionary (of one-dimensional arrays) where `Q[s][a]` is the estimated action value corresponding to state `s` and action `a`.

Please complete the function in the code cell below.

(*Feel free to define additional functions to help you to organize your code.*)

```
In [90]: def q_learning(env, num_episodes, alpha, gamma=1.0):
             # initialize empty dictionary of arrays
             Q = defaultdict(lambda: np.zeros(env.nA))
             # loop over episodes
             for i_episode in range(1, num_episodes+1):
                 state = env.reset()
                 for i in range(200):
                     action = action_epsilon_greedy(state,Q,env.nA,i_episode)
                     next_state,reward,done,info = env.step(action)
                     if not done:
                         Q[state][action] += alpha*(reward+gamma*max(Q[next_state]) - Q[state][a
                         state = next_state
                         action = action_epsilon_greedy(state,Q,env.nA,i_episode)
                     else :
                         Q[state][action] += alpha*(reward-Q[state][action])
                         break

                 # monitor progress
                 if i_episode % 100 == 0:
                     print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
                     sys.stdout.flush()

                 ## TODO: complete the function

             return Q
```

Use the next code cell to visualize the ***estimated*** optimal policy and the corresponding state-value function.

If the code cell returns **PASSED**, then you have implemented the function correctly! Feel free to change the `num_episodes` and `alpha` parameters that are supplied to the function. However, if you'd like to ensure the accuracy of the unit test, please do not change the value of `gamma` from the default.

```
In [91]: # obtain the estimated optimal policy and corresponding action-value function
         Q_sarsamax = q_learning(env, 5000, .01)

         # print the estimated optimal policy
```

```
        policy_sarsamax = np.array([np.argmax(Q_sarsamax[key]) if key in Q_sarsamax else -1 for
        check_test.run_check('td_control_check', policy_sarsamax)
        print("\nEstimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):")
        print(policy_sarsamax)

        # plot the estimated optimal state-value function
        plot_values([np.max(Q_sarsamax[key]) if key in Q_sarsamax else 0 for key in np.arange(4
```

```
Episode 5000/5000
```

**PASSED**

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 1  1  2  2  1  1  1  1  2  0  1  2]
 [ 1  1  2  1  1  1  1  1  1  1  2  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```


### 1.0.4 Part 3: TD Control: Expected Sarsa

In this section, you will write your own implementation of the Expected Sarsa control algorithm.

Your algorithm has four arguments: - `env`: This is an instance of an OpenAI Gym environment. - `num_episodes`: This is the number of episodes that are generated through agent-environment interaction. - `alpha`: This is the step-size parameter for the update step. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as output: - `Q`: This is a dictionary (of one-dimensional arrays) where `Q[s][a]` is the estimated action value corresponding to state `s` and action `a`.

Please complete the function in the code cell below.

(*Feel free to define additional functions to help you to organize your code.*)

```
In [92]: def next_state_value(state,Q,nA,i_episodes,eps=None):
             epsilon = 1.0/i_episodes
```

```
            if eps is not None :
                epsilon = eps
            prob = np.ones(nA) * (epsilon/nA)
            greedy = np.argmax(Q[state])
            prob[greedy] = 1-epsilon + epsilon/nA
            value  = sum(prob*Q[state])
            return value
```

```
In [95]: def expected_sarsa(env, num_episodes, alpha, gamma=1.0):
            # initialize empty dictionary of arrays
            Q = defaultdict(lambda: np.zeros(env.nA))
            # loop over episodes
            for i_episode in range(1, num_episodes+1):
                # monitor progress
                state = env.reset()
                for i in range(200):
                    action = action_epsilon_greedy(state,Q,env.nA,i_episode)
                    next_state,reward,done,info = env.step(action)
                    if not done:
                        Q[state][action] += alpha*(reward+gamma*next_state_value(next_state,Q,e
                        state = next_state
                        action = action_epsilon_greedy(state,Q,env.nA,i_episode)
                    else:
                        Q[state][action] += alpha*(reward-Q[state][action])
                        break
                if i_episode % 100 == 0:
                    print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
                    sys.stdout.flush()

                ## TODO: complete the function

            return Q
```

Use the next code cell to visualize the ***estimated*** optimal policy and the corresponding state-value function.

If the code cell returns **PASSED**, then you have implemented the function correctly! Feel free to change the `num_episodes` and `alpha` parameters that are supplied to the function. However, if you'd like to ensure the accuracy of the unit test, please do not change the value of `gamma` from the default.

```
In [98]: # obtain the estimated optimal policy and corresponding action-value function
         Q_expsarsa = expected_sarsa(env, 10000, 0.01)

         # print the estimated optimal policy
         policy_expsarsa = np.array([np.argmax(Q_expsarsa[key]) if key in Q_expsarsa else -1 for
         check_test.run_check('td_control_check', policy_expsarsa)
         print("\nEstimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):")
         print(policy_expsarsa)
```

```
        # plot the estimated optimal state-value function
        plot_values([np.max(Q_expsarsa[key]) if key in Q_expsarsa else 0 for key in np.arange(4
```

```
Episode 10000/10000
```

**PASSED**

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 3  2  1  2  1  0  3  1  0  2  2  1]
 [ 0  2  1  2  1  1  2  1  1  0  2  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```