Temporal_Difference_Solution

August 14, 2018

1 Mini Project: 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

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

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

The agent moves through a 4×12 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]: import numpy as np
        from plot_utils import plot_values

        # 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)
```

```
<matplotlib.figure.Figure at 0x7f1cd09fe8d0>
```

1.0.2 Part 1: TD Prediction: State Values

In this section, you will write your own implementation of TD prediction (for estimating the state-value function).

We will begin by investigating a policy where the agent moves: - `RIGHT` in states `0` through `10`, inclusive,
- `DOWN` in states `11`, `23`, and `35`, and - `UP` in states `12` through `22`, inclusive, states `24` through `34`, inclusive, and state `36`.

The policy is specified and printed below. Note that states where the agent does not choose an action have been marked with `-1`.

```
In [5]: policy = np.hstack([1*np.ones(11), 2, 0, np.zeros(10), 2, 0, np.zeros(10), 2, 0, -1*np.o
        print("\nPolicy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):")
        print(policy.reshape(4,12))
```

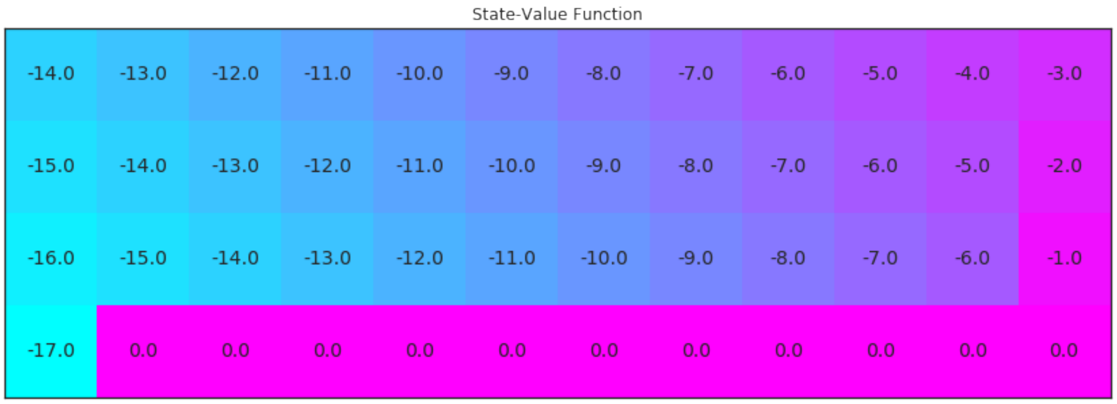
```
Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
```

Run the next cell to visualize the state-value function that corresponds to this policy. Make sure that you take the time to understand why this is the corresponding value function!

```
In [6]: V_true = np.zeros((4,12))
        for i in range(3):
            V_true[0:12][i] = -np.arange(3, 15)[::-1] - i
        V_true[1][11] = -2
```

```
V_true[2][11] = -1
V_true[3][0] = -17

plot_values(V_true)
```



The above figure is what you will try to approximate through the TD prediction algorithm.

Your algorithm for TD prediction has five 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. - `policy`: This is a 1D numpy array with `policy.shape` equal to the number of states (`env.nS`). `policy[s]` returns the action that the agent chooses when in state `s`. - `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: - `V`: This is a dictionary where `V[s]` is the estimated value of state `s`.

Please complete the function in the code cell below.

```
In [7]: from collections import defaultdict, deque
        import sys

        def td_prediction(env, num_episodes, policy, alpha, gamma=1.0):
            # initialize empty dictionaries of floats
            V = defaultdict(float)
            # loop over episodes
            for i_episode in range(1, num_episodes+1):
                # monitor progress
                if i_episode % 100 == 0:
                    print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
                    sys.stdout.flush()
                # begin an episode, observe S
                state = env.reset()
                while True:
                    # choose action A
                    action = policy[state]
                    # take action A, observe R, S'
                next_state, reward, done, info = env.step(action)
                # perform updates
                V[state] = V[state] + (alpha * (reward + (gamma * V[next_state]) - V[state])
                # S <- S'
                state = next_state
                # end episode if reached terminal state
                if done:
                    break
        return V
```

Run the code cell below to test your implementation and visualize the estimated 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.

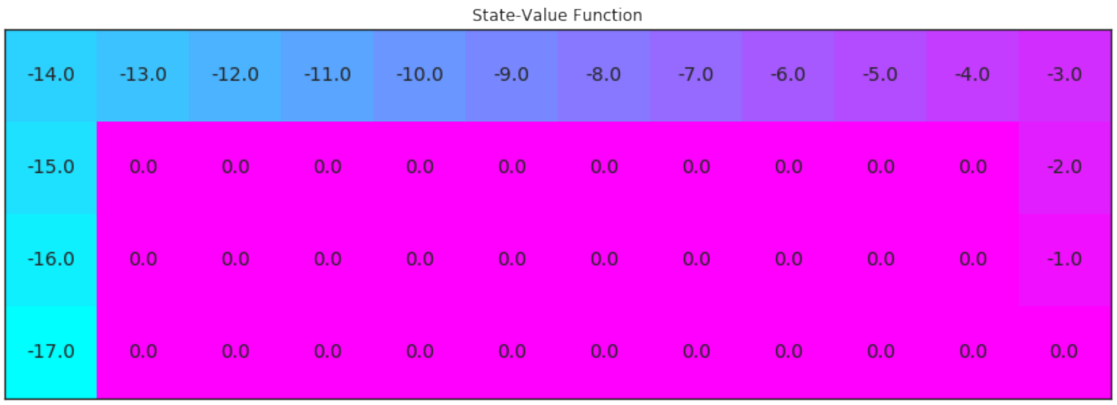
```
In [8]: import check_test

        # evaluate the policy and reshape the state-value function
        V_pred = td_prediction(env, 500000, policy, .01)

        # please do not change the code below this line
        V_pred_plot = np.reshape([V_pred[key] if key in V_pred else 0 for key in np.arange(48)],
        check_test.run_check('td_prediction_check', V_pred_plot)
        plot_values(V_pred_plot)
```

```
Episode 500000/500000
```

PASSED



How close is your estimated state-value function to the true state-value function corresponding to the policy?

You might notice that some of the state values are not estimated by the agent. This is because under this policy, the agent will not visit all of the states. In the TD prediction algorithm, the agent can only estimate the values corresponding to states that are visited.

1.0.3 Part 2: 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 [8]: def update_Q(Qsa, Qsa_next, reward, alpha, gamma):
            """ updates the action-value function estimate using the most recent time step """
            return Qsa + (alpha * (reward + (gamma * Qsa_next) - Qsa))

        def epsilon_greedy_probs(env, Q_s, i_episode, eps=None):
            """ obtains the action probabilities corresponding to epsilon-greedy policy """
            epsilon = 1.0 / i_episode
            if eps is not None:
                epsilon = eps
            policy_s = np.ones(env.nA) * epsilon / env.nA
            policy_s[np.argmax(Q_s)] = 1 - epsilon + (epsilon / env.nA)
            return policy_s
```

```
In [9]: import matplotlib.pyplot as plt
        %matplotlib inline

        def sarsa(env, num_episodes, alpha, gamma=1.0):
            # initialize action-value function (empty dictionary of arrays)
            Q = defaultdict(lambda: np.zeros(env.nA))
            # initialize performance monitor
            plot_every = 100
            tmp_scores = deque(maxlen=plot_every)
            scores = deque(maxlen=num_episodes)
            # loop over episodes
            for i_episode in range(1, num_episodes+1):
                # monitor progress
                if i_episode % 100 == 0:
                    print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
                    sys.stdout.flush()
                # initialize score
                score = 0
                # begin an episode, observe S
                state = env.reset()
                # get epsilon-greedy action probabilities
                policy_s = epsilon_greedy_probs(env, Q[state], i_episode)
                # pick action A
                action = np.random.choice(np.arange(env.nA), p=policy_s)
        # limit number of time steps per episode
        for t_step in np.arange(300):
            # take action A, observe R, S'
            next_state, reward, done, info = env.step(action)
            # add reward to score
            score += reward
            if not done:
                # get epsilon-greedy action probabilities
                policy_s = epsilon_greedy_probs(env, Q[next_state], i_episode)
                # pick next action A'
                next_action = np.random.choice(np.arange(env.nA), p=policy_s)
                # update TD estimate of Q
                Q[state][action] = update_Q(Q[state][action], Q[next_state][next_action]
                                            reward, alpha, gamma)
                # S <- S'
                state = next_state
                # A <- A'
                action = next_action
            if done:
                # update TD estimate of Q
                Q[state][action] = update_Q(Q[state][action], 0, reward, alpha, gamma)
                # append score
                tmp_scores.append(score)
                break
        if (i_episode % plot_every == 0):
            scores.append(np.mean(tmp_scores))
    # plot performance
    plt.plot(np.linspace(0,num_episodes,len(scores),endpoint=False),np.asarray(scores))
    plt.xlabel('Episode Number')
    plt.ylabel('Average Reward (Over Next %d Episodes)' % plot_every)
    plt.show()
    # print best 100-episode performance
    print(('Best Average Reward over %d Episodes: ' % plot_every), np.max(scores))
    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.

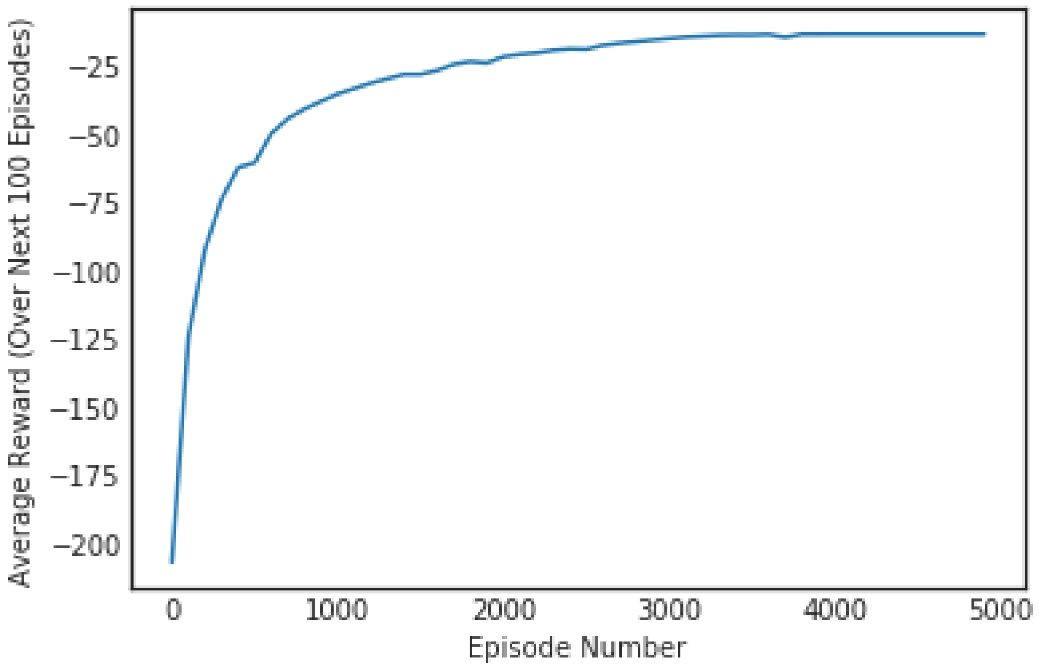
```
In [10]: # obtain the estimated optimal policy and corresponding action-value function
         Q_sarsa = sarsa(env, 5000, .01)

         # 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)
```

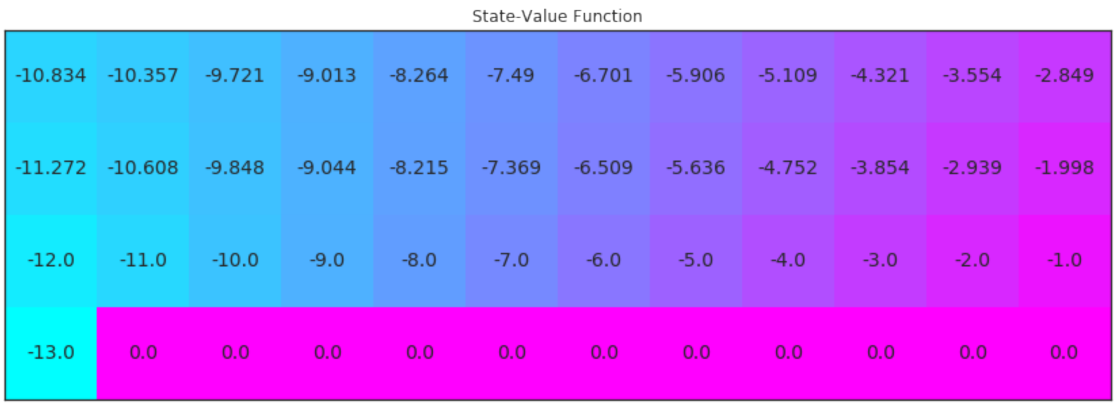
```
Episode 5000/5000
```



```
Best Average Reward over 100 Episodes:  -13.0
```

PASSED

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 3  1  0  1  1  1  1  2  0  1  1  2]
 [ 2  1  2  2  1  0  1  0  1  3  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: 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 [11]: def q_learning(env, num_episodes, alpha, gamma=1.0):
             # initialize action-value function (empty dictionary of arrays)
             Q = defaultdict(lambda: np.zeros(env.nA))
             # initialize performance monitor
             plot_every = 100
             tmp_scores = deque(maxlen=plot_every)
             scores = deque(maxlen=num_episodes)
             # loop over episodes
             for i_episode in range(1, num_episodes+1):
                 # monitor progress
                 if i_episode % 100 == 0:
                     print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
                     sys.stdout.flush()
                 # initialize score
                 score = 0
                 # begin an episode, observe S
                 state = env.reset()
                 while True:
                     # get epsilon-greedy action probabilities
                     policy_s = epsilon_greedy_probs(env, Q[state], i_episode)
                     # pick next action A
```

```
                action = np.random.choice(np.arange(env.nA), p=policy_s)
                # take action A, observe R, S'
                next_state, reward, done, info = env.step(action)
                # add reward to score
                score += reward
                # update Q
                Q[state][action] = update_Q(Q[state][action], np.max(Q[next_state]), \
                                                       reward, alpha, gamma)
                # S <- S'
                state = next_state
                # until S is terminal
                if done:
                    # append score
                    tmp_scores.append(score)
                    break
            if (i_episode % plot_every == 0):
                scores.append(np.mean(tmp_scores))
        # plot performance
        plt.plot(np.linspace(0,num_episodes,len(scores),endpoint=False),np.asarray(scores))
        plt.xlabel('Episode Number')
        plt.ylabel('Average Reward (Over Next %d Episodes)' % plot_every)
        plt.show()
        # print best 100-episode performance
        print(('Best Average Reward over %d Episodes: ' % plot_every), np.max(scores))
        return Q
```

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

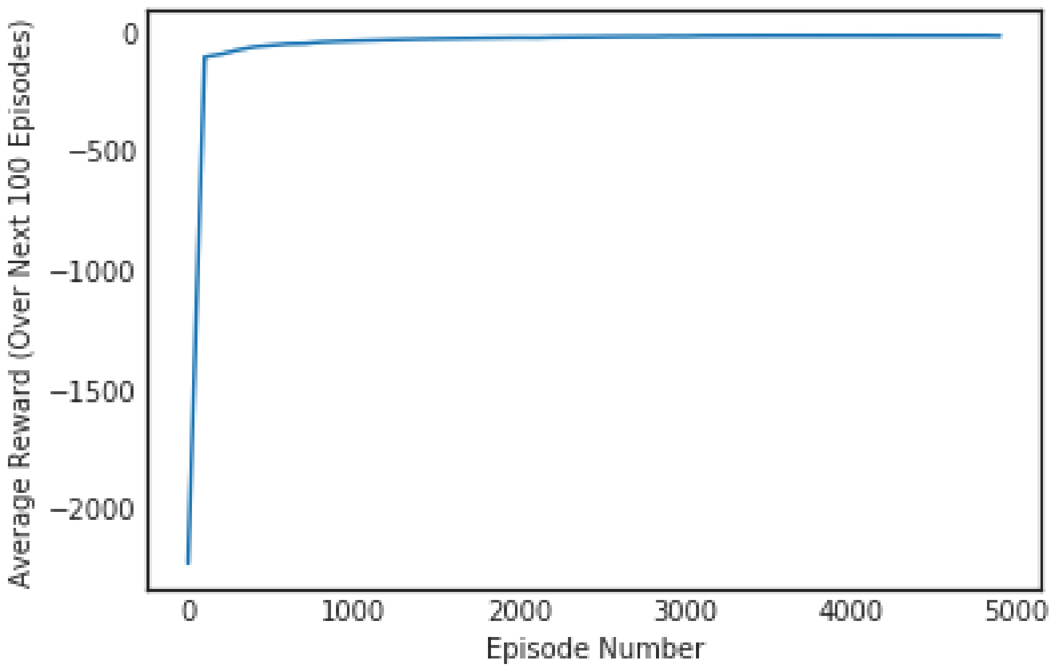
If the code cell returns **PASSED**, then you have implemented the function correctly! Feel free to change the `num_episodes` and `alpha` parameters that are supplied to the function. However, if you'd like to ensure the accuracy of the unit test, please do not change the value of `gamma` from the default.

```
In [12]: # 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
```

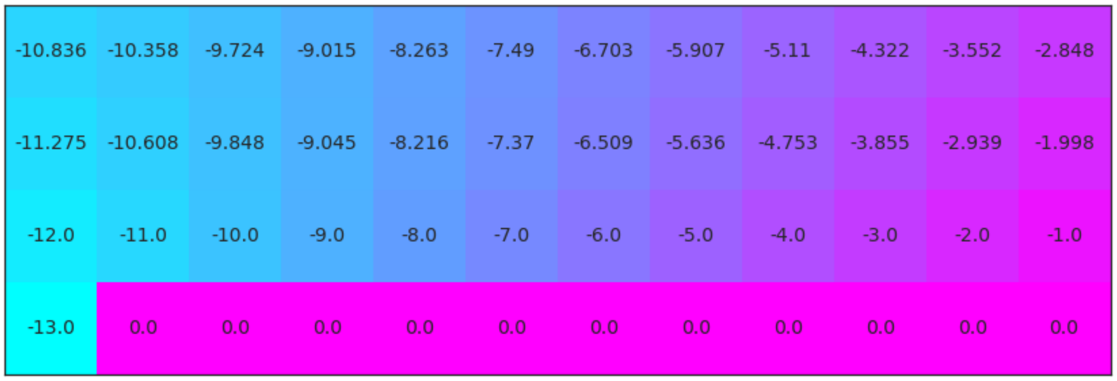


```
Best Average Reward over 100 Episodes:  -13.0
```

PASSED

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

State-Value Function



-10.836	-10.358	-9.724	-9.015	-8.263	-7.49	-6.703	-5.907	-5.11	-4.322	-3.552	-2.848
-11.275	-10.608	-9.848	-9.045	-8.216	-7.37	-6.509	-5.636	-4.753	-3.855	-2.939	-1.998
-12.0	-11.0	-10.0	-9.0	-8.0	-7.0	-6.0	-5.0	-4.0	-3.0	-2.0	-1.0
-13.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

1.0.5 Part 4: 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 [13]: def expected_sarsa(env, num_episodes, alpha, gamma=1.0):
             # initialize action-value function (empty dictionary of arrays)
             Q = defaultdict(lambda: np.zeros(env.nA))
             # initialize performance monitor
             plot_every = 100
             tmp_scores = deque(maxlen=plot_every)
             scores = deque(maxlen=num_episodes)
             # loop over episodes
             for i_episode in range(1, num_episodes+1):
                 # monitor progress
                 if i_episode % 100 == 0:
                     print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
                     sys.stdout.flush()
                 # initialize score
                 score = 0
                 # begin an episode
                 state = env.reset()
                 # get epsilon-greedy action probabilities
                 policy_s = epsilon_greedy_probs(env, Q[state], i_episode, 0.005)
                 while True:
                     # pick next action
                     action = np.random.choice(np.arange(env.nA), p=policy_s)
                     # take action A, observe R, S'
                     next_state, reward, done, info = env.step(action)
                     # add reward to score
                     score += reward
                     # get epsilon-greedy action probabilities (for S')
                     policy_s = epsilon_greedy_probs(env, Q[next_state], i_episode, 0.005)
                     # update Q
                     Q[state][action] = update_Q(Q[state][action], np.dot(Q[next_state], policy_
                                                    reward, alpha, gamma)
                     # S <- S'
                     state = next_state
                     # until S is terminal
                     if done:
                         # append score
```

```
                    tmp_scores.append(score)
                    break
            if (i_episode % plot_every == 0):
                scores.append(np.mean(tmp_scores))
        # plot performance
        plt.plot(np.linspace(0,num_episodes,len(scores),endpoint=False),np.asarray(scores))
        plt.xlabel('Episode Number')
        plt.ylabel('Average Reward (Over Next %d Episodes)' % plot_every)
        plt.show()
        # print best 100-episode performance
        print(('Best Average Reward over %d Episodes: ' % plot_every), np.max(scores))
        return Q
```

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

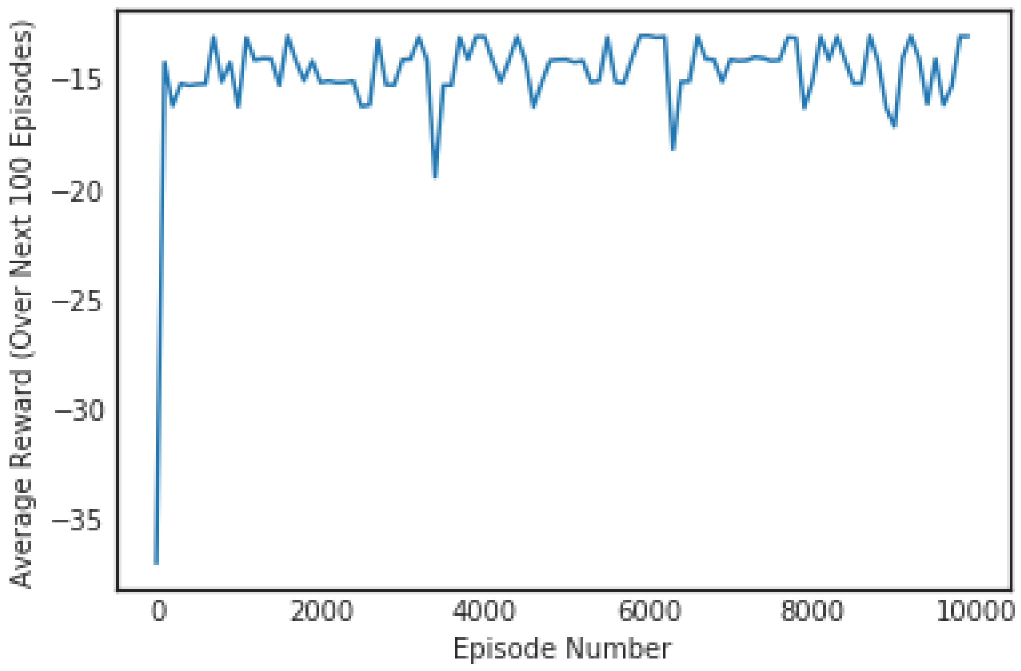
If the code cell returns **PASSED**, then you have implemented the function correctly! Feel free to change the `num_episodes` and `alpha` parameters that are supplied to the function. However, if you'd like to ensure the accuracy of the unit test, please do not change the value of `gamma` from the default.

```
In [14]: # obtain the estimated optimal policy and corresponding action-value function
         Q_expsarsa = expected_sarsa(env, 10000, 1)

         # 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
```

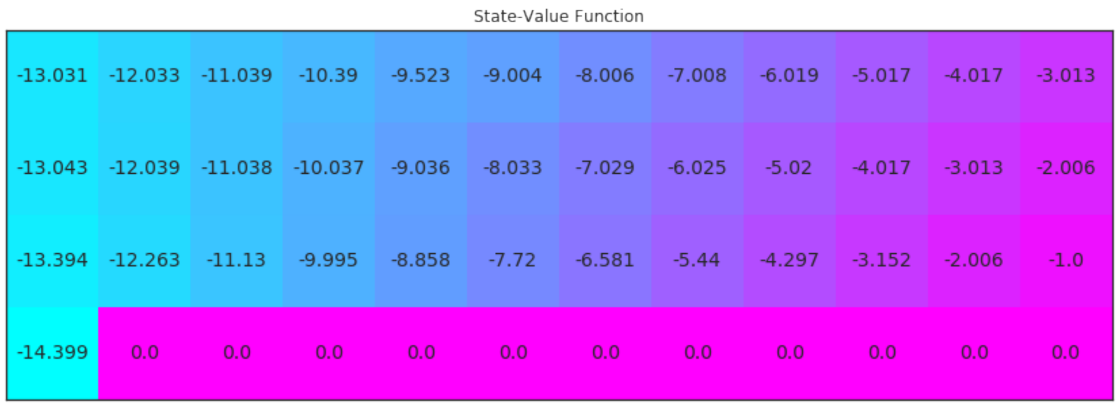
```
Episode 10000/10000
```



```
Best Average Reward over 100 Episodes:  -13.03
```

PASSED

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


State-Value Function

-13.031	-12.033	-11.039	-10.39	-9.523	-9.004	-8.006	-7.008	-6.019	-5.017	-4.017	-3.013
-13.043	-12.039	-11.038	-10.037	-9.036	-8.033	-7.029	-6.025	-5.02	-4.017	-3.013	-2.006
-13.394	-12.263	-11.13	-9.995	-8.858	-7.72	-6.581	-5.44	-4.297	-3.152	-2.006	-1.0
-14.399	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0