CmpE58Y - Robot Learning - Temporal Difference Learning

March 17, 2018

```
In [1]: import numpy as np


        class Grid:
            def __init__(self, columns, rows, rewards, actions):
                self.columns = columns
                self.rows = rows
                self.rewards = rewards
                self.actions = actions
                self.i = 3
                self.j = 0

            def states(self):
                x_states=[i for i in range(self.rows)]
                y_states=[j for j in range(self.columns)]
                states = [(i,j) for i  in  x_states for j in y_states ]
                return states

            def move(self, action):
                if action in self.actions[self.i][self.j]:
                    if action == 'U':
                        self.i -= 1
                    elif action == 'D':
                        self.i += 1
                    elif action == 'R':
                        self.j += 1
                    elif action == 'L':
                        self.j -= 1
                return self.rewards[self.i][self.j]


            def pick_max_value_index(self,Q_state):
                highest = max(Q_state.values())
                return np.random.choice([k for k, v in Q_state.items() if v == highest])

            def epsilon_greedy_selection(self, Q, eps=0.01, start_point=False):
                if start_point:
```

```
                sx = 3
                sy = 0
            else:
                sx = self.i
                sy = self.j

            p = np.random.random()
            if p < (1 - eps):
                Q_state=Q[sx][sy]
                return self.pick_max_value_index(Q_state)
            else:
                possible_actions=self.actions[sx][sy]
                return np.random.choice(possible_actions)

        def maxQ(self,Q,state):
            max_valued_action=self.pick_max_value_index(Q[state[0]][state[1]])
            return Q[state[0]][state[1]][max_valued_action]
```

In [2]:

```
def initialize_grid():

        rewards = []
        for row in range(4):
            rewards.append([])
            for column in range(12):
                rewards[row].append(-1)
        for i in range(1,11):
            rewards[3][i] = -100
        rewards[3][0] = -1
        rewards[3][11] = 0

        actions = []
        for row in range(4):
            actions.append([])
            for column in range(12):
                actions[row].append('')

        for i in range(1,11):
            actions[0][i] = ('L', 'D', 'R')
            actions[1][i] = ('L', 'U', 'R', 'D')
            actions[2][i] = ('L', 'U', 'R', 'D')
            actions[3][i] = ('L', 'U', 'R')
        actions[0][0] = ('D', 'R')
        actions[1][0] = ('U', 'D','R')
        actions[2][0] = ('U', 'D','R')
        actions[3][0] = ('U', 'R')
        actions[0][11] = ('D', 'L')
        actions[1][11] = ('U', 'D','L')
        actions[2][11] = ('U', 'D','L')
```

```
        actions[3][11] = ('U', 'L')

        g = Grid(12, 4, rewards, actions)
        return g
```

```
In [3]: def initq(grid):
            Q = []
            for row in range(4):
                Q.append([])
                for column in range(12):
                    Q[row].append({})

            states_=grid.states()
            for s in states_:
                for action in grid.actions[s[0]][s[1]]:
                    Q[s[0]][s[1]][action]=0
            return Q
```

```
In [4]: def sarsa(grid, gamma, alpha):
            Q = initq(grid)
            epochs = 2000
            sarsa_path = []
            rewards = []
            for epoch in range(epochs):
                s = (3,0)
                curr_x, curr_y = s
                grid = initialize_grid()
                action = grid.epsilon_greedy_selection(Q, start_point=True)
                path = []
                sum_reward = 0
                goal = (3,11)
                while (curr_x,curr_y) != goal:
                    path.append([curr_x,curr_y])
                    reward = grid.move(action)
                    x = grid.i
                    y = grid.j


                    if x==3 and y in range(1,11):

                        Q[curr_x][curr_y][action] += alpha * (reward + gamma * 0 - Q[curr_x][cu
                        grid.i = 3
                        grid.j = 0
                        next_action = grid.epsilon_greedy_selection(Q, start_point=True)
                        print('cliff!')
                        curr_x=3
                        curr_y=0
```

```
                else:
                    next_action = grid.epsilon_greedy_selection(Q)
                    Q[curr_x][curr_y][action] += alpha * (reward + gamma * Q[x][y][next_ac

                    curr_x = x
                    curr_y = y
                action = next_action
                sum_reward += reward
            rewards.append(sum_reward)
            path.append(goal)
            sarsa_path.append(path)

        return path, rewards
```

In [37]:
```
grid = initialize_grid()
gamma = 1
alpha = 0.6
```

In [38]:
```
s_path, s_rewards=sarsa(grid, gamma, alpha)
```

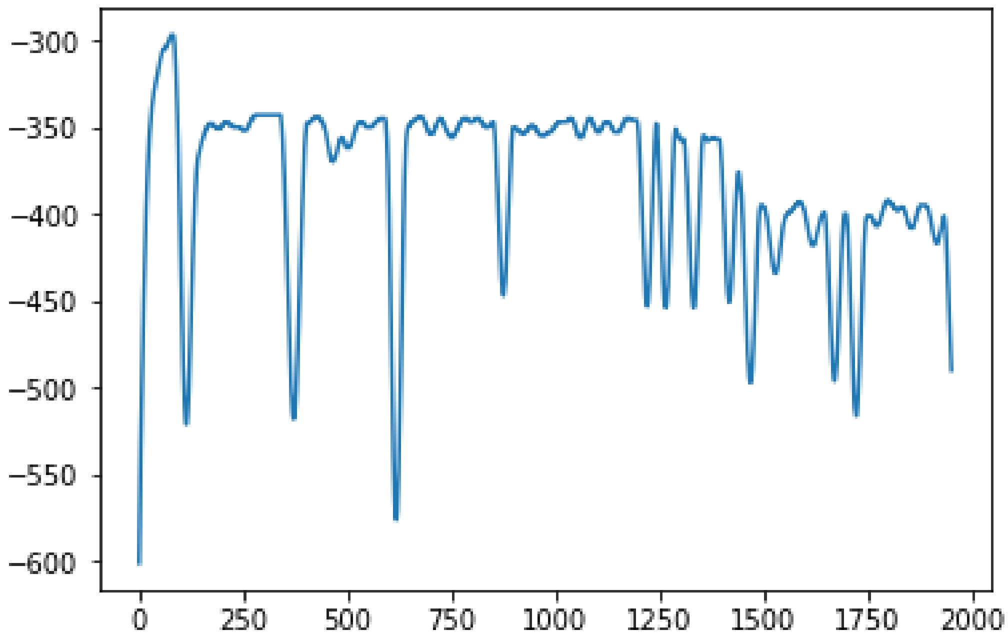
```
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
```

```
In [39]: def q_learning(grid, gamma, alpha):
             Q = initq(grid)
             epochs = 2000
             q_path = []
             rewards = []
             for epoch in range(epochs):
                 s = (3,0)
                 curr_x = s[0]
                 curr_y = s[1]
                 grid = initialize_grid()

                 path = []
                 sum_reward = 0
                 goal = (3,11)
                 while (curr_x,curr_y) != goal:
                     path.append([curr_x,curr_y])
                     action = grid.epsilon_greedy_selection(Q)
                     reward = grid.move(action)
                     x = grid.i
                     y = grid.j
                     if x==3 and y in range(1,11):

                         Q[curr_x][curr_y][action] += alpha * (reward + gamma * 0 - Q[curr_x][c
                         grid.i = 3
                         grid.j = 0
                         print('cliff!')
                         curr_x=3
                         curr_y=0

                     else:
                         Q[curr_x][curr_y][action] += alpha * (reward + gamma * grid.maxQ(Q, (x
                         curr_x = x
                         curr_y = y

                     sum_reward += reward

                 rewards.append(sum_reward)
                 path.append(goal)
                 q_path.append(path)

             return path, rewards


In [40]: q_grid = initialize_grid()
         gamma = 1
         alpha = 0.6

In [41]: Q_path, Q_rewards=q_learning(q_grid, gamma, alpha)
```

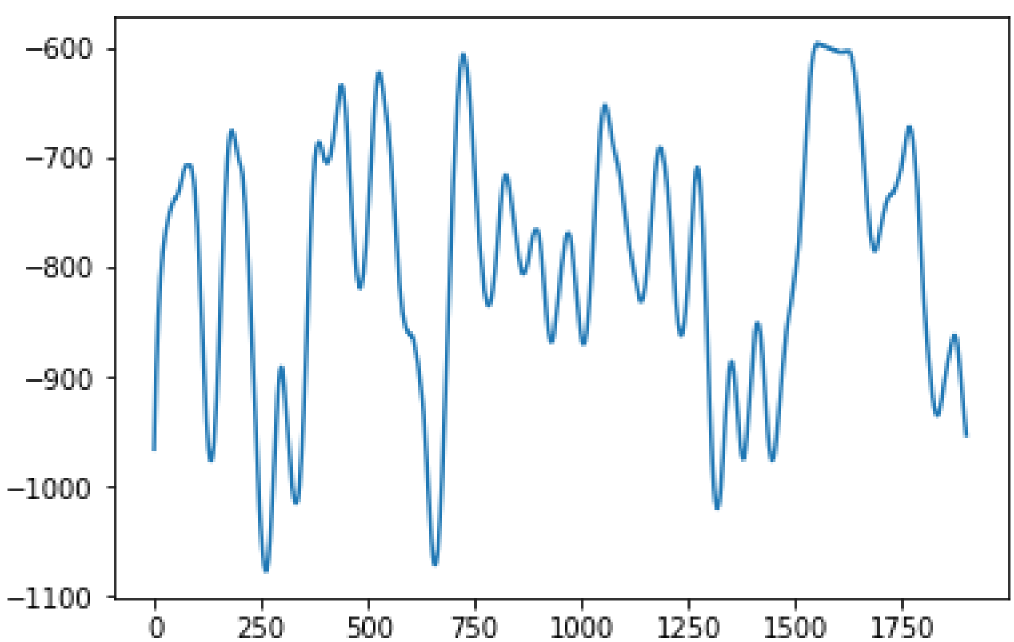
```
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
```

```
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
cliff!
```

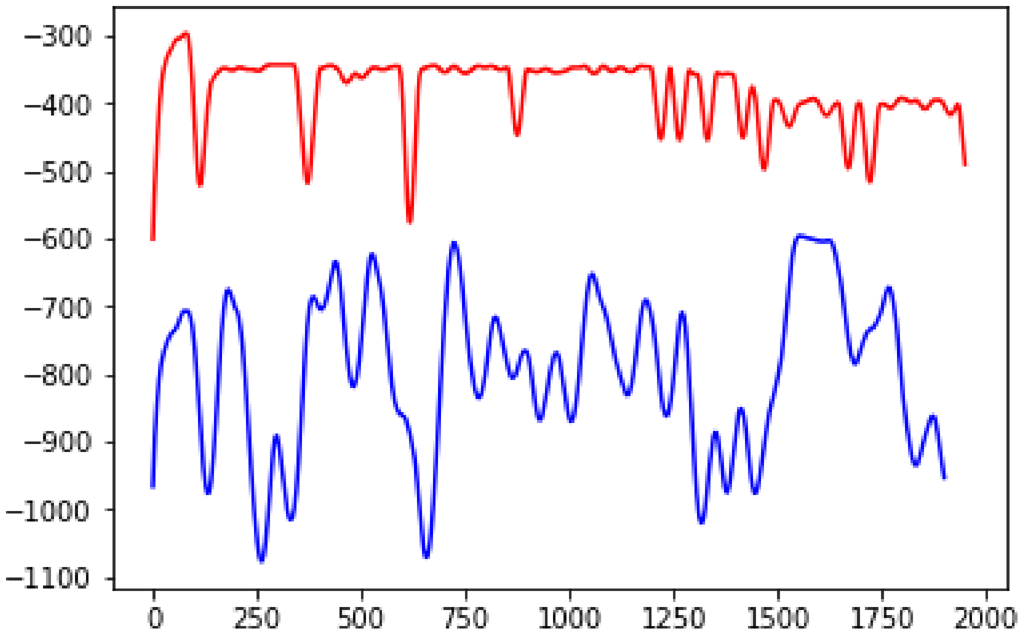
In [48]:
```python
from scipy import signal
import matplotlib.pyplot as plt
win = signal.hann(50)
sarsa_filtered = signal.convolve(s_rewards, win,method='fft', mode='valid')
plt.plot(sarsa_filtered)
plt.show()
```



In [49]:
```
win = signal.hann(100)
Q_filtered = signal.convolve(Q_rewards, win, method='fft',mode='valid')
plt.plot(Q_filtered)
plt.show()
```



In [50]:
```
plt.plot(sarsa_filtered,'r-')
plt.plot(Q_filtered,'b')
plt.show()
```



In [ ]:

In [ ]: