Assignment-1-Part-2

October 24, 2018

1 ELEN 6885 Reinforcement Learning coding assignment

Your code should remain in the block marked by ############################ # YOUR CODE STARTS HERE # YOUR CODE ENDS HERE ############################ Please don't edit anything outside the block.

```
In [1]: import numpy as np
        import random
        import matplotlib.pyplot as plt
        import gym
```

1.1 1. Incremental Implementation of Average

We've finished the incremental implementation of average for you. Please call the function estimate with 1/step step size and fixed step size to compare the difference between this two on a simulated Bandit problem. (2 pts)

```
In [2]: from RLalgs.utils import estimate
        random.seed(6885)
        numTimeStep = 10000
        q_h = np.zeros(numTimeStep + 1) # Q Value estimate with 1/step step size
        q_f = np.zeros(numTimeStep + 1) # Q value estimate with fixed step size
        FixedStepSize = 0.5 #A large number to exaggerate the difference
        for step in range(1, numTimeStep + 1):
            if step < numTimeStep / 2:
                r = random.gauss(mu = 1, sigma = 0.1)
            else:
                r = random.gauss(mu = 3, sigma = 0.1)
            #TIPS: Call function estimate defined in ./RLalgs/utils.py
            ############################
            # YOUR CODE STARTS HERE
            q_f[step] = estimate(q_f[step-1], FixedStepSize, r)  # NewEstimate = OldEstimate +

            q_h[step] = estimate(q_h[step-1],1/step, r)

            # YOUR CODE ENDS HERE
            ############################
```

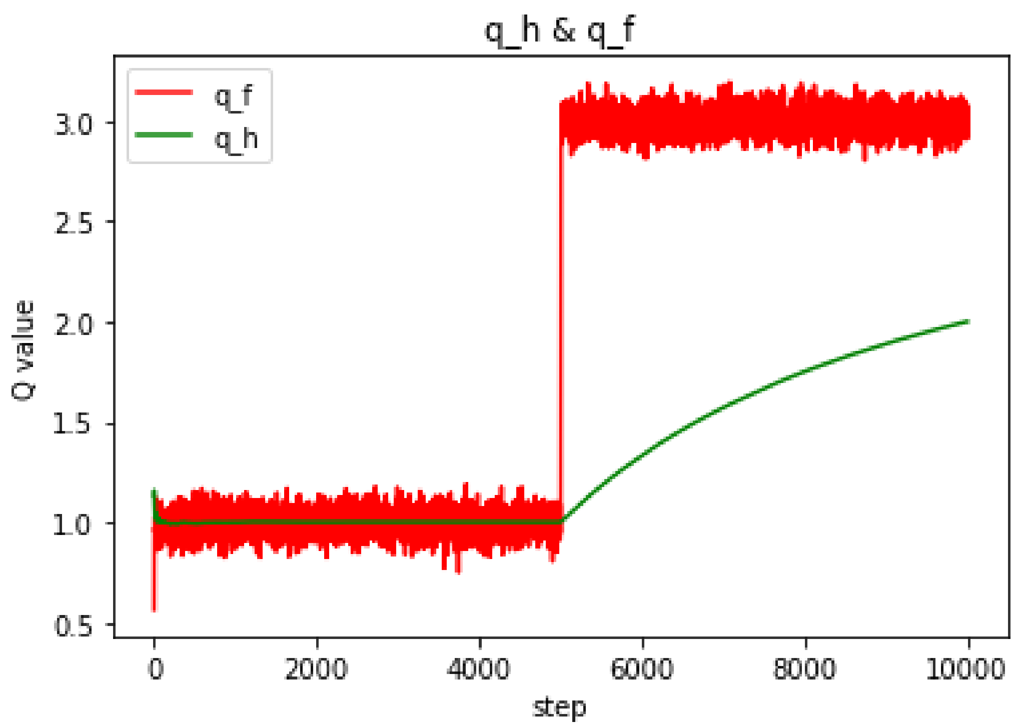
```
q_h = q_h[1:]
q_f = q_f[1:]
```

```
RLalgs is a package containing Reinforcement Learning algorithms Epsilon-Greedy, Policy Iterat
```

Plot the two Q value estimate (Please include a title, labels on both axes, and legends) (3 pts)

In [3]:
```
#############################
# YOUR CODE STARTS HERE
x = np.arange(numTimeStep)
y1 = np.array(q_f)
y2 = np.array(q_h)
plt.plot(x, y1, 'r', label = 'q_f')
plt.plot(x, y2, 'g', label = 'q_h')
plt.title("q_h & q_f")
plt.xlabel('step')
plt.ylabel('Q value')
plt.legend()
plt.show
# YOUR CODE ENDS HERE
#############################
```

Out[3]:
```
<function matplotlib.pyplot.show>
```



1.2 2. ϵ-Greedy for Exploration

In Reinforcement Learning, we are always faced with the dilemma of exploration and exploitation. ϵ-Greedy is a trade-off between them. You are gonna implement Greedy and ϵ-Greedy. We combine these two policies in one function by treating Greedy as ϵ-Greedy where $\epsilon = 0$. Edit the function epsilon_greedy in ./RLalgs/utils.py (5 pts)

```
In [4]: from RLalgs.utils import epsilon_greedy
        np.random.seed(6885) #Set the seed to cancel the randomness
        q = np.random.normal(0, 1, size = 5)
        ###########################
        # YOUR CODE STARTS HERE
        greedy_action = epsilon_greedy(q,0,6885) #Use epsilon = 0 for Greedy
        e_greedy_action = epsilon_greedy(q, 0.1 ,6885) #Use epsilon = 0.1 and pass the paramet
        # YOUR CODE ENDS HERE
        ###########################
        print('Values:')
        print(q)
        print('Greedy Choice =', greedy_action)
        print('Epsilon-Greedy Choice =', e_greedy_action)
```

```
Values:
[ 0.61264537  0.27923079 -0.84600857  0.05469574 -1.09990968]
Greedy Choice = 0
Epsilon-Greedy Choice = 0
```

You should get the following results. Values: [0.61264537 0.27923079 -0.84600857 0.05469574 -1.09990968] Greedy Choice = 0

1.3 3. Frozen Lake Environment

```
In [5]: env = gym.make('FrozenLake-v0')
```

1.3.1 3.1 Derive Q value from V value

Edit function action_evaluation in ./RLalgs/utils.py TIPS: $q(s,a) = \sum_{s',r} p(s',r|s,a)(r + \gamma v(s'))$ (5 pts)

```
In [6]: from RLalgs.utils import action_evaluation
        v = np.ones(16)
        q = action_evaluation(env = env.env, gamma = 1, v = v)
        print('Action values:')
        print(q)
```

```
Action values:
[[1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
```

```
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.33333333 1.33333333 1.33333333]
 [1.         1.         1.         1.        ]]
```

You should get Q values all equal to one except at State 14

Pseudo-codes of the following four algorithms can be found on Page 80, 83, 130, 131 of the Sutton's book

1.3.2 3.2 Model-based RL algorithms

```
In [7]: from RLalgs.utils import action_evaluation, action_selection, render
```

1.3.3 3.2.1 Policy Iteration

Edit the function policy_iteration and relevant functions in ./RLalgs/pi.py to implement the Policy Iteration Algorithm (15 pts)

```
In [8]: from RLalgs.pi import policy_iteration
        V, policy, numIterations = policy_iteration(env = env.env, gamma = 1, max_iteration = 5
        print('State values:')
        print(V)
        print('Number of iterations =', numIterations)
        print('policy=',policy)
```

```
State values:
[0.82352774 0.8235272  0.82352682 0.82352662 0.82352791 0.
 0.52941063 0.         0.82352817 0.82352851 0.76470509 0.
 0.         0.88235232 0.94117615 0.        ]
Number of iterations = 7
policy= [0 3 3 3 0 0 0 0 3 1 0 0 0 2 1 0]
```

```
In [9]: #Uncomment and run the following to evaluate your result, comment them when you genera
        #Q = action_evaluation(env = env.env, gamma = 1, v = V)
        #policy_estimate = action_selection(Q)
        #render(env, policy_estimate)
```

1.3.4 3.2.2 Value Iteration

Edit the function value_iteration and relevant functions in ./RLalgs/vi.py to implement the Value Iteration Algorithm (10 pts)

```
In [10]: from RLalgs.vi import value_iteration
         V, policy, numIterations = value_iteration(env = env.env, gamma = 1, max_iteration = 5
         print('State values:')
         print(V)
         print("policy=",policy)
         print('Number of iterations to converge =', numIterations)
```

```
State values:
[0.82352937 0.82352936 0.82352935 0.82352935 0.82352938 0.
 0.52941174 0.         0.82352938 0.82352939 0.76470586 0.
 0.         0.88235293 0.94117646 0.         ]
policy= [0 3 3 3 0 0 0 0 3 1 0 0 0 2 1 0]
Number of iterations to converge = 500
```

```
In [11]: #Uncomment and run the following to evaluate your result, comment them when you genera
         #Q = action_evaluation(env = env.env, gamma = 1, v = V)
         #policy_estimate = action_selection(Q)
         #render(env, policy_estimate)
```

1.3.5 3.3 Model free RL algorithms

1.3.6 3.3.1 Q-Learning

Edit the function QLearning in ./RLalgs/ql.py to implement the Q-Learning Algorithm (10 pts)

```
In [12]: from RLalgs.ql import QLearning
         Q = QLearning(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1, e = 0.1)
         print('Action values:')
         print(Q)
```

```
Action values:
[[6.10895640e-04 2.29458529e-02 3.91062816e-04 2.49042429e-03]
 [5.45491449e-03 0.00000000e+00 2.20310930e-02 6.98618191e-03]
 [4.38825675e-02 1.51222461e-02 1.59740274e-02 1.91570255e-03]
 [1.98886788e-02 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [2.67950614e-02 4.78051419e-03 0.00000000e+00 4.06382555e-03]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [4.21315633e-02 7.63872383e-02 5.95143231e-03 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [2.28528662e-04 2.83367451e-02 1.24973363e-03 0.00000000e+00]
 [1.20297389e-01 7.36709703e-02 8.22597921e-02 2.06216529e-02]
 [2.06687013e-01 8.85889802e-03 5.67941247e-02 2.07456919e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
```

```
 [1.31356405e-02 2.23795585e-01 6.99429619e-02 5.69381967e-02]
 [1.06482335e-01 1.38108548e-01 5.37628427e-01 1.38212541e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

```
In [13]: #Uncomment the following to evaluate your result, comment them when you generate the p
         #policy_estimate = action_selection(Q)
         #render(env, policy_estimate)
```

1.3.7 3.3.2 SARSA

Edit the function SARSA in ./RLalgs/sarsa.py to implement the SARSA Algorithm. (10 pts)

```
In [16]: from RLalgs.sarsa import SARSA
         Q = SARSA(env = env.env, num_episodes = 2000, gamma = 1, lr = 0.1, e = 0.1)
         print('Action values:')
         print(Q)
```

```
Action values:
[[0.19283373 0.14601927 0.15075225 0.13176006]
 [0.03524757 0.05637272 0.04663125 0.15656155]
 [0.154889   0.05541737 0.07252648 0.05926315]
 [0.05424932 0.03222373 0.00936301 0.01550972]
 [0.20630603 0.16537498 0.09368401 0.10780817]
 [0.         0.         0.         0.        ]
 [0.20751778 0.07660365 0.07512893 0.02749119]
 [0.         0.         0.         0.        ]
 [0.07431331 0.10325834 0.14378254 0.26714383]
 [0.15602657 0.28779959 0.22637807 0.11348312]
 [0.34755293 0.25485939 0.21197539 0.10551269]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.11976931 0.2196419  0.46745267 0.23700168]
 [0.35447934 0.68133434 0.55132055 0.56614678]
 [0.         0.         0.         0.        ]]
```

```
In [17]: #Uncomment the following to evaluate your result, comment them when you generate the p
         #policy_estimate = action_selection(Q)
         #render(env, policy_estimate)
```

1.3.8 3.3.1 Human

You can play this game if you are interested. See if you can get the frisbee either with or without the model.

```
In [18]: from RLalgs.utils import human_play
         #Uncomment and run the following to play the game, comment it when you generate the pd
         #human_play(env)
```

1.4 4. Exploration VS. Exploitation

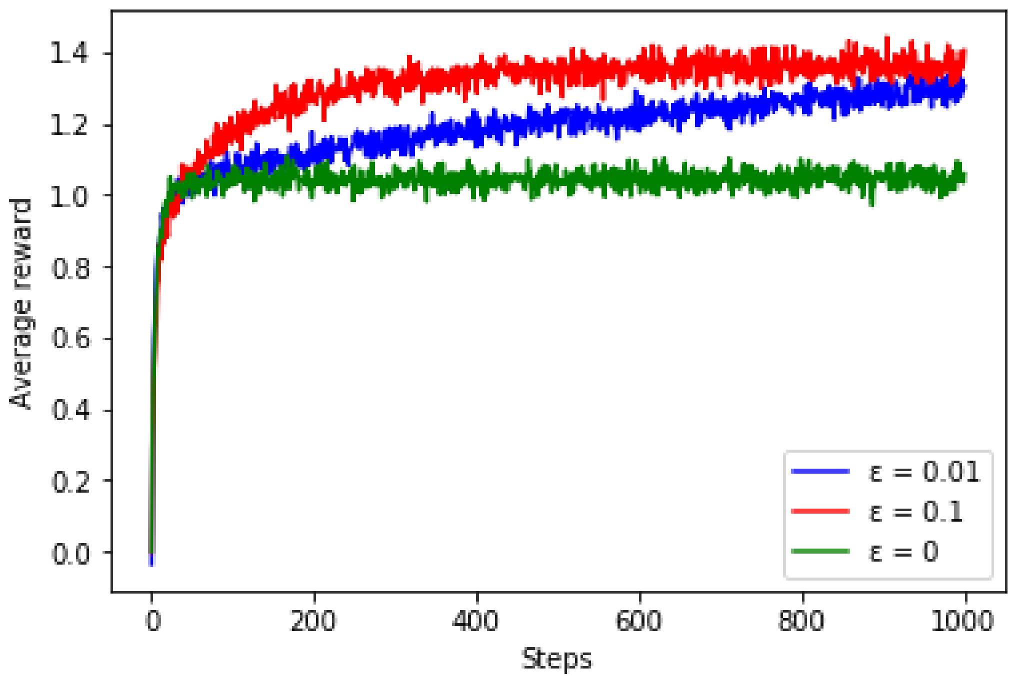
Try to reproduce Figure 2.2 (the upper one is enough) of the Sutton's book based on the experiment described in Chapter 2.3 Extra credit (3 pts)

In [19]:
```
# # Do the experiment and record average reward acquired in each time step
# ############################
# # YOUR CODE STARTS HERE
# Returns the action-value for each action at the current time step
def Qt(actions):
    results = [0.0 if actions[i][1] == 0 else actions[i][0] / float(actions[i][1]) fo
    return results

# The reward for selecting an action
def get_reward(true_values, action_index):
    estimated = np.random.normal(true_values[action_index], size=1)[0]
    return estimated

def epoch_greedy(k, epsilon, iterations):
    true_values = np.random.normal(size=k)
    # actions[i] is the ith action
    # actions[i][0] is the sum of rewards for action i
    # actions[i][1] is the no. of times action i has been taken
    actions = [[0.0, 0] for j in range(k)]
    rewards = []
    for it in range(iterations):
        prob =  np.random.rand(1)  #random.random()
        if prob > epsilon:
            action_index = np.argmax(Qt(actions))
        else:
            action_index = np.random.randint(0, k)
        reward = get_reward(true_values, action_index)
        # Update
        rewards.append(reward)
        action = actions[action_index]
        action[0] += reward
        action[1] += 1
    return rewards
# Returns the mean reward for each iteration across
# epochs executions
def run_experiment(k, epsilon, iters, epochs):
    rewards = []
    for i in range(epochs):
        rewards.append(epoch_greedy(k, epsilon, iters))
    # Compute the mean reward for each iteration
    means = np.mean(np.array(rewards), axis=0)
    return means
# # YOUR CODE ENDS HERE
# ############################
```

```
In [20]: # Plot the average reward
         ############################
         # YOUR CODE STARTS HERE
         e_0_01 = run_experiment(10, 0.01, 1000, 2000)
         e_0_1 = run_experiment(10, 0.1, 1000, 2000)
         e_0 = run_experiment(10, 0, 1000, 2000)
         x_axis = range(1, 1001)
         plt.plot(x_axis, e_0_01, c='blue', label=' = 0.01')
         plt.plot(x_axis, e_0_1, c='red', label=' = 0.1')
         plt.plot(x_axis, e_0, c='green', label=' = 0')
         plt.xlabel('Steps')
         plt.ylabel('Average reward')
         plt.legend()
         plt.show()
         # YOUR CODE ENDS HERE
         ############################
```



You should get a result that Greedy behaves well at the beginning, but then surpassed by ϵ-Greedy with $\epsilon = 0.1$