

Assignment 1: Tabular Methods

Name: Nutchu Temiyasathit

ID: A53249944

This exercise requires you to solve a simple grid-world problem called 'FrozenLake-v0' in OpenAI Gym. We will solve the problem in two different ways. First we will solve the problem using dynamic programming, thus requiring a model of the system. Second we will do it using model-free temporal difference (Q-Learning). Finally, as a bonus you may also show it learning using a naive approach called hill-climbing.

Required for all

Set up environment

```
In [1]: %matplotlib inline
import gym
import numpy as np
from matplotlib import pyplot as plt
from collections import Counter
env = gym.make('FrozenLake-v0')
```

Pre. Test Policy Function

Write a function to test a policy. Return the average rate of successful episodes over 100 trials.

```
In [2]: def testPolicy(policy):
        result = 0
        for i in range(100):
            state = env.reset()
            done = False
            while not done:
                action = policy[state]
                state, r, done, _ = env.step(action)
                result += r

        percentSuccess = result/100

        return percentSuccess
```

Model-based Learning

1. Policy Iteration

Perform policy iteration on the Frozenlake example.

1.1 Find the system model

First, model $T(s, a, s')$ and $R(s, a, s')$ over 100000 random initializations.

```
In [3]: def learnModel():

    T = Counter()
    count = Counter()
    R = Counter()

    for i in range(100000):
        env.reset()
        env.env.s = np.random.choice(env.observation_space.n)
        state = env.env.s
        action = np.random.randint(0,4)
        next_state, r, done, _ = env.step(action)
        T[(state, action, next_state)] += 1
        count[(state, action)] += 1
        R[(state, action, next_state)] = r

    for i, (s, a, s_) in enumerate(T):
        T[(s, a, s_)] = T[(s, a, s_)]/count[(s,a)]

    return R,T

#first learn the model
[R,T] = learnModel()
```

1.2 What does the transition model tell you about the stochastic behavior of actions? What does it tell you about the stochasticity of the rewards? What would you expect an optimal agent's policy to do?

ans: The transition model states the probability of changing from one state to another by performing an action. The transition model consists of many zero entries, because one state can only translate into at most 4 other states. The reward model is deterministic here i.e. only when it reaches the goal then the reward is 1, 0 otherwise. Therefore there are only three transitions near the goal that will result in reward. The optimal agent's policy then should move to the direction that will get itself closer to the goal.

1.3 Write a function for Policy Evaluation

```
In [4]: #runPolicyEvaluation:
def runPolicyEvaluation(policy,V,R,T,discount_factor):
    Vnew = np.zeros(16)
    for i, v in enumerate(V):
        for j in range(16):
            Vnew[i] += T[(i,policy[i], j)]*(R[i, policy[i], j] + discount_factor*V[j])

    return Vnew
```

1.4 Run Policy iteration.

and show a bar graph of successful runs vs iteration on the policy. Use a discount factor of 0.98, and terminate policy after 40 iterations of policy updates. Plot the percentSuccesses at every iteration (i.e. the return of the testPolicy function).

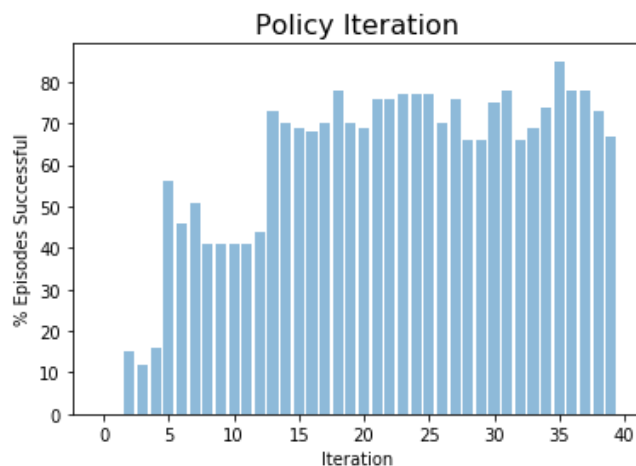
```
In [5]: policy = [np.random.randint(0,4) for i in range(16)]
V = np.zeros(16)
discount_factor = 0.98
percentSuccesses = np.zeros(40)

for i in range(40):
    V = runPolicyEvaluation(policy, V, R, T, discount_factor)
    for j in range(16):
        action_values = np.zeros(4)
        for a in range(4):
            for s in range(16):
                action_values[a] += T[(j, a, s)]*(R[j, a, s] + discount_factor*V[s])
            policy[j] = np.argmax(action_values)

    percentSuccesses[i] = testPolicy(policy)

# plot improvement over time
plt.figure()
plt.bar(np.arange(len(percentSuccesses)), 100*np.array(percentSuccesses), align='center',
        , alpha=0.5)
plt.ylabel('% Episodes Successful')
plt.xlabel('Iteration')
plt.title('Policy Iteration',fontsize=16)
print('Policy iteration policy:', policy)
```

Policy iteration policy: [0, 3, 0, 3, 0, 0, 0, 0, 3, 1, 0, 0, 0, 2, 1, 0]



Model-Free Learning

2 Q Value-Iteration (Q-Learning)

2.1 Set up a Q learning function

Set your exploration rate to $1 - \text{episode_num} / \text{total_num_of_episodes}$ for linear convergence from completely random action selection to a greedy policy. Return a set of policies (at 5%, 10%,...100% of the total number of episodes) so that in the later section you can perform policy evaluation on intermediate Q-tables and show progress.

```
In [6]: def runQLearning(learning_rate,discount_factor,num_of_episodes,Q,explore_type='linear'):
    Q_saved = []
    explore_rate = 0
    for i in range(num_of_episodes+1):
        i +=1
        if explore_type == 'linear':
            explore_rate = 1-i/num_of_episodes
        else:
            explore_rate = 1000/(1000+i)

        state = env.reset()
        done = False
        while not done:
            action = np.argmax(Q[state,:])
            if(np.random.random() <= explore_rate):
                action = np.random.randint(0,4)
            state_, r, done, _ = env.step(action)

            Q[state_,action]= Q[state_,action]+learning_rate*(r+ discount_factor* np.max(Q
[state_,:])-Q[state_,action])
            state = state_

        if i%(0.05*num_of_episodes) == 0:
            Q_saved.append(Q.copy()) #save last Q
    return Q_saved
```

2.2 Perform Q-learning.

Show policies during intermediate phases of Q-learning, at 0, 10%, 20%,...,100% of the total episodes during training. Set a learning rate of 0.98 and a discount factor of 0.95. Start with a zero-filled Q-table. Run 10000 episodes. Plot the bar graph of the success rate over time to visualize the improvements to success rate the new policies are providing.

```

In [7]: learning_rate = 0.02
discount_factor = 0.95
num_episodes = 10000

Q = np.zeros([16, 4])

Q_saved = runQLearning(learning_rate, discount_factor, num_episodes, Q)

policy = [np.random.randint(0,4) for i in range(16)]
percentSuccesses = []

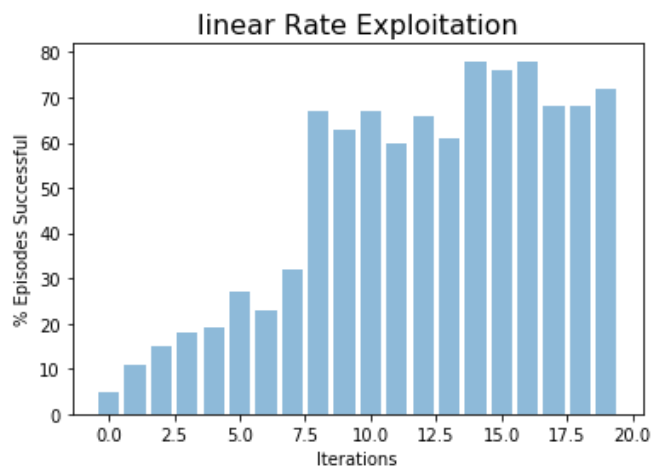
for q in Q_saved:
    for j in range(16):
        policy[j] = np.argmax(q[j, :])

    percentSuccesses.append(testPolicy(policy))

plt.figure()
plt.bar(np.arange(len(percentSuccesses)), 100*np.array(percentSuccesses), align='center',
        , alpha=0.5)
plt.ylabel('% Episodes Successful')
plt.xlabel('Iterations')
plt.title('linear Rate Exploitation', fontsize=16)
print('Q-learning (linear) policy:', policy)

```

Q-learning (linear) policy: [0, 3, 0, 3, 0, 0, 0, 0, 3, 1, 0, 0, 0, 2, 1, 0]



2.3 Log Rate Exploration

Run Q-learning for a log exploration rate, $\frac{1000}{1000+\text{episode_num}}$, for 10,000 episodes. Perform policy evaluation and plot the success rate over time. You may find setting Q0 to a random number initialization helps (set it to something very small, i.e. $0.000001 \cdot \text{rand}()$, since setting it to zero sets a fixed seed).

```

In [8]: learning_rate = 0.02
discount_factor = 0.95
num_episodes = 10000

Q = np.array([[0.000001*np.random.rand() for i in range(4)] for j in range(16)])

Q_saved = runQLearning(learning_rate, discount_factor, num_episodes, Q, 'log')

policy = [np.random.randint(0,4) for i in range(16)]
percentSuccesses = []

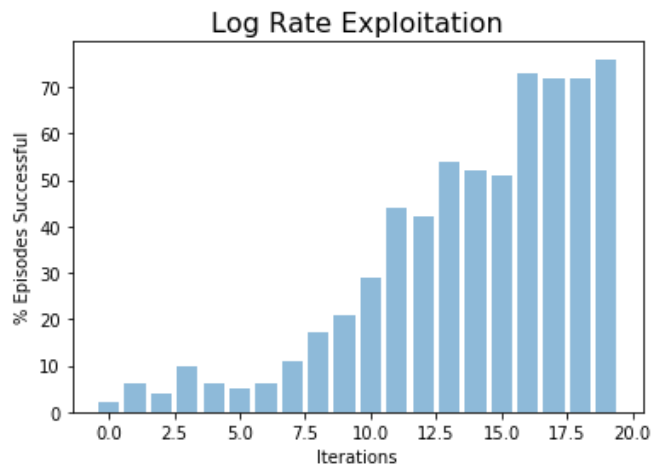
for q in Q_saved:
    for j in range(16):
        policy[j] = np.argmax(q[j, :])

    percentSuccesses.append(testPolicy(policy))

plt.figure()
plt.bar(np.arange(len(percentSuccesses)), 100*np.array(percentSuccesses), align='center',
        , alpha=0.5)
plt.ylabel('% Episodes Successful')
plt.xlabel('Iterations')
plt.title('Log Rate Exploitation', fontsize=16)
print('Q-learning (log) policy:', policy)

```

Q-learning (log) policy: [0, 3, 0, 0, 0, 3, 2, 0, 3, 1, 0, 0, 1, 2, 1, 3]



BONUS: Hill Climbing (25%, granted only if Parts 1 and 2 are complete)

Demonstrate hill climbing, where your Q values are chosen randomly, and you save improvements, with new Q values to try as

$$Q_{test} \leftarrow Q_{best} + rand(S, A)$$

Plot the a bar graph with x-axis labelling the iteration number when an improvement occurred, and y axis as the % of successful episodes. Run on N = 1000 iterations of hill climbing, with 100 episodes per iteration.

```

In [9]: Q_best = np.array([[np.random.rand() for i in range(4)] for j in range(16)])

improvementsIndex = []
percentSuccesses = []

policy = [np.argmax(Q_best[i, :]) for i in range(16)]
best_performance = testPolicy(policy)

for i in range(1000):
    Q_test = np.add(Q_best, np.array([[np.random.rand() for i in range(4)] for j in range(16)]))
    for j in range(16):
        policy[j] = np.argmax(Q_test[j, :])

    performance = testPolicy(policy)
    if performance > best_performance:
        Q_best = Q_test
        best_performance = performance
        improvementsIndex.append(i)
        percentSuccesses.append(best_performance)

plt.figure()
plt.bar(np.arange(len(improvementsIndex)), 100*np.array(percentSuccesses), align='center', alpha=0.5)
plt.xticks(np.arange(len(improvementsIndex)), improvementsIndex)
plt.ylabel('% Episodes Successful')
plt.xlabel('Iteration')
plt.title('Hill Climbing', fontsize=16)
print(policy, ': Hill-climbing policy')

```

[0, 0, 0, 2, 0, 3, 0, 2, 3, 3, 0, 2, 2, 3, 0, 3] : Hill-climbing policy

