Problem Set 7 (Total points: 100)

Q1. Markov Decision Process [20 points]

Imagine an MDP corresponding to playing slot machines in a casino. Suppose you start with $20 cash to spend in the casino, and you decide to play until you lose all your money or until you double your money (i.e., increase your cash to at least \$40). There are two slot machines you can choose to play: 1) slot machine X costs $10 to play and will pay out \$20 with probability 0.05 and will pay $0 otherwise; and 2) slot machine Y costs \$20 to play and will pay out $30 with probability 0.01 and \$0 otherwise. As you are playing, you keep choosing machine X or Y at each turn.

Write down the MDP that corresponds to the above problem. Clearly specify the state space, action space, rewards and transition probabilities. Indicate which state(s) are terminal. Assume that the discount factor γ = 1.

Notes: There are several valid ways to specify the MDP, so you have some flexibility in your solution. For example, rewards can take many different forms, but overall you should get a higher reward for stopping when you double your money than when you lose all your money!

State space S = { 0 (terminal state), 10, 20, 30, 40 (goal state)}

Action Space A = {1, 2} - slot machines to play with

Reward R(s,a,s') = 1 if s' = 40 else 0

Transition probability = p(s,a,s')

$p(s_{-10}|s',1) = 0.95$

$p(s_{+10}|s',1) = 0.05$

$p(s_{-20}|s',2) = 0.99$

$p(s_{+20}|s',2) = 0.01$

Reinforcement Learning

In the remainder of the problem set you will implement the Q-Learning Algorithm to solve the "Frozen Lake" problem. We will use OpenAI’s gym package to develop our solution in Python.

OpenAI Gym Setup

Read the set-up instructions for Gym here. The instructions also give a good overview of the API for this package, so please read through it before proceeding.

Frozen Lake

Instead of using CartPole, we're going to be using the FrozenLake environment. Read through the code for this environment by following the Github link.

Winter is quickly approaching, and we have to worry about navigating frozen lakes. It's only early November, so the lakes haven't completely frozen and if you make the wrong step you may fall through. We'll need to learn how to get to our destination when stuck on the ice, without falling in. The lake we're going to consider is a square lake with spots to step on in the shape of a grid.

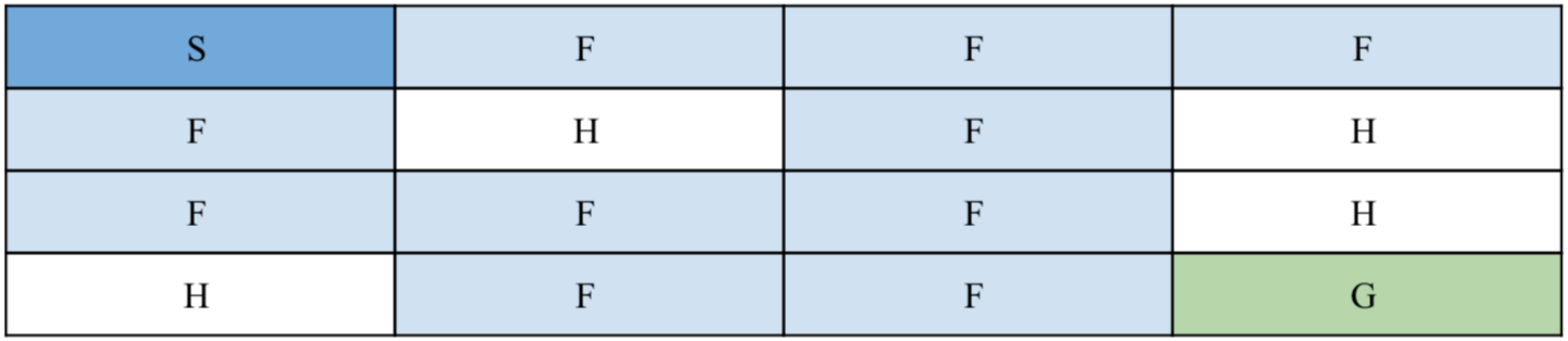
The surface is described using a 4x4 grid like the following

```
S F F F
F H F H
F F F H
H F F G
```

Each spot can have one of four states:

- S: starting point.
- G: goal point.
- F: frozen spot, where it's safe to walk.
- H: hole in the ice, where it's not safe to walk.

For example, consider the lake,



S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

There are four possible actions: UP, DOWN, LEFT, RIGHT. Although we can see the path we need to walk, the agent does not. We're going to train an agent to discover this via problem solving. However, walking on ice isn't so easy! Sometimes you slip and aren't able to take the step you intended.

The episode ends when you reach the goal or fall in a hole.

You receive a reward of 1 if you reach the goal, and zero otherwise.

Q2. Walking on the Frozen Lake [10 points]

Write a script that sets up the Frozen Lake environment and takes 10 walks through it, consisting of a maximum 10 randomly sampled actions during each walk. After each step, render the current state, and print out the reward and whether the walk is "done", i.e. in the terminal state, because the agent fell into a hole (stop if it is). In your own words, explain how this environment behaves.

With random actions, the agent keeps falling into the holes. However, it rarely makes it to the Goal and sometimes unable to finish the game in 10 steps.

In [16]:

```python
import gym

env = gym.make("FrozenLake-v1")
print('Initial State')
env.reset()
env.render()
for i in range(10):
    print('#############')
    print('Walk number: ', i+1)
    env.reset()
    for j in range(10):

        random_sample_action = env.action_space.sample()
        new_game_state, reward, done, info = env.step(random_sample_action)

        #env.render()
        if done:
            #print('Walk finished')
            print('Reward: ', reward)
            env.render()
            print('###########')

            break
```

```
Initial State

SFFF
FHFH
```

```
FFFH
HFFG
#############
Walk number:  1
Reward:  0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
###########
#############
Walk number:  2
Reward:  0.0
  (Up)
SFFF
FHFH
FFFH
HFFG
###########
#############
Walk number:  3
Reward:  0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
###########
#############
Walk number:  4
#############
Walk number:  5
Reward:  0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
###########
#############
Walk number:  6
Reward:  0.0
  (Left)
SFFF
FHFH
FFFH
HFFG
###########
#############
Walk number:  7
#############
Walk number:  8
```

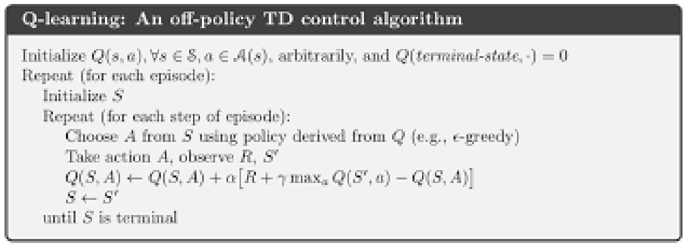
```
Reward:  0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
###########
#############
Walk number:  9
Reward:  0.0
  (Left)
SFFF
FHFH
FFFH
HFFG
###########
#############
Walk number:  10
Reward:  0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
###########
```

Q3. Q-Learning [50 points]

You will implement Q-learning to solve the problem. Assume that the environment has states S and actions A. Use the function signature provided below:

```
def q_learning(env, alpha=0.5, gamma=0.95, epsilon=0.1, num_episodes=500):
""" Performs Q-learning for the given environment.
Initialize Q to all zeros.
:param env: Unwrapped OpenAI gym environment.
:param alpha: Learning rate parameter.
:param gamma: Decay rate (future reward discount) parameter.
:param num_episodes: Number of episodes to use for learning.
:return: Q table, i.e. a table with the Q value for every <S, A> pair.
"""
```

The pseudocode for Q-Learning was described in lecture, but for your reference, we provide it here:



Q-learning: An off-policy TD control algorithm

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

In [271…

```python
def epsilon_greedy(Q, epsilon, num_actions):
    def policyFunction(state):
        action_prob = np.ones(num_actions, dtype = float) * epsilon / num_actions
        #print(Q[state])
        if Q[state,0] == Q[state,1] == Q[state,2] == Q[state,3]:
            best_action = np.random.randint(4,size = 1)
        else:
            best_action = np.argmax(Q[state])
        #print(best_action)
        action_prob[best_action] += (1.0 - epsilon)
        return action_prob
    return policyFunction


def q_learning(env, alpha=0.5, gamma=0.95, epsilon=0.1, num_episodes=10000):
    n_observations = env.observation_space.n
    n_actions = env.action_space.n
    Q = np.zeros((n_observations, n_actions))
    np.random.seed(42)
    policy = epsilon_greedy(Q, epsilon, env.action_space.n)
    for episode in range(num_episodes):
        current_state = env.reset()
        #done = False
        episode_reward = 0
        for i in range(100):
            action_prob = policy(current_state)
            #print(action_prob)

            action = np.random.choice(np.arange(len(action_prob)),p = action_prob)
            #print(action)
            next_state, reward, done, info = env.step(action)
            #print(next_state)
            #print(max(Q[next_state, :]))

            best_action = np.argmax(Q[next_state])
            Q[current_state, action] = (1-alpha) * Q[current_state,action] + alpha*(reward + gamma*(Q[next_state, best_action]))
            episode_reward += reward

            if done:
                #env.render()
                break
            current_state = next_state
        #env.render()
        #print(Q)
        #print(episode_reward)
    return Q
```

Q4. Main Function [20 points]

You also need to implement the main function to solve the entire FrozenLake-v0 problem, including setting up the Gym environment, calling the q-learning function as defined above, printing out the returned Q-table, etc.

You should use the ϵ-greedy algorithm (solving Exploration-Exploitation Dilemma) to generate actions for each state. Try `num_episodes` with different values, e.g. `num_episodes=500, 1000, 5000, 10000, 50000`. You should also try different initializations for the Q-table, i.e. Random Initialization and Zero Initialization.

Please do not change any default value of environment setting, such as `is_slippery`, if not mentioned above.

Provide the final Q-table of FrozenLake-v0 for **each <num_episode, init_method>** you have tried [10 points], and analyze what you observe [10 points].

If you are interested in testing your learned Q-learned, you can compute the win rate (the probability of agents to goal following the Q-table). Note that you should use `argmax` instead of epsilon greedy to choose the action at the current state. Win Rate is a good way to test whether you write your algorithm correctly.

Just for your reference, we test our implementation for 10000 trials after 50000-episode training using zero initialization and the win rate is 0.4985. You should achieve similar value under the same test.

Additional Instructions

If you're new to OpenAI's Gym, first go through OpenAI's full tutorial listed earlier and visit the Appendix to this homework before proceeding. Some additional rules:

- Only submit **original code** written entirely by you.
- **Permitted non-standard libraries**: gym, numpy.
- **Only use numpy for random sampling, and seed at the beginning of each function with**: np.random.seed(42)

- **Unwrap the OpenAI gym before providing them to these functions.**

  env = gym.make("FrozenLake-v0").unwrapped

In [277…

```
import gym
import numpy as np

#env = gym.make("FrozenLake-v1")
#Q_table = q_learning(env, num_episodes = 5000)
#print(Q_table)
def test():
    env = gym.make("FrozenLake-v1")
    Q_table500 = q_learning(env, num_episodes = 500)
    print("########################################")
    print("Q_table for num_episodes = 500: ")
    print(Q_table500)

    Q_table1000 = q_learning(env, num_episodes = 1000)
    print("########################################")
    print("Q_table for num_episodes = 1000: ")
    print(Q_table1000)

    Q_table5000 = q_learning(env, num_episodes = 5000)
    print("########################################")
    print("Q_table for num_episodes = 5000: ")
    print(Q_table5000)

    Q_table10000 = q_learning(env, num_episodes = 10000)
    print("########################################")
    print("Q_table for num_episodes = 10000: ")
    print(Q_table10000)

    Q_table50000 = q_learning(env, num_episodes = 50000)
    print("########################################")
    print("Q_table for num_episodes = 50000: ")
    print(Q_table50000)
    return(Q_table50000)
Q_table50000 = test()
```

```
########################################
Q_table for num_episodes = 500:
[[0.15077054 0.13636018 0.14721593 0.14278742]
 [0.08347643 0.08284517 0.02785152 0.13673039]
 [0.1318835  0.07830902 0.06627191 0.0821834 ]
 [0.0126798  0.04164205 0.05508008 0.07973382]
 [0.22540343 0.14317512 0.06841986 0.08905301]
 [0.         0.         0.         0.        ]
 [0.02850971 0.00188956 0.25859417 0.00774998]
 [0.         0.         0.         0.        ]
 [0.04893905 0.08508339 0.16219307 0.3179109 ]
 [0.25358963 0.34202583 0.16065538 0.16239624]
 [0.51643149 0.12514874 0.09245962 0.14748672]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.2116166  0.05995296 0.48635433 0.31624129]
 [0.49118074 0.5501288  0.83926946 0.60383824]
 [0.         0.         0.         0.        ]]
########################################
Q_table for num_episodes = 1000:
[[0.20561581 0.18185971 0.17087741 0.18771332]
 [0.04912142 0.0501324  0.06238729 0.13426355]
 [0.05523352 0.04555357 0.05534128 0.067123  ]
 [0.0210979  0.00772605 0.01464741 0.06564535]
 [0.26466979 0.17342956 0.09134907 0.12494567]
 [0.         0.         0.         0.        ]
 [0.00699176 0.00807202 0.17434409 0.00353495]
 [0.         0.         0.         0.        ]
 [0.21424508 0.21738334 0.0411381  0.29360033]
 [0.18340909 0.33096093 0.20004149 0.10735476]
 [0.48164173 0.15644032 0.00773013 0.20354244]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.24880093 0.25057095 0.49449939 0.46676513]
 [0.56871868 0.65691516 0.52882557 0.55787274]
 [0.         0.         0.         0.        ]]
########################################
Q_table for num_episodes = 5000:
[[1.19536145e-01 1.50257925e-01 1.19498254e-01 1.20551870e-01]
 [4.34084587e-02 7.80988538e-02 3.91309852e-02 7.78285761e-02]
 [6.81877991e-02 6.39059172e-02 6.58518717e-02 7.44757526e-02]
 [3.28987509e-02 4.73096015e-02 1.63268612e-02 7.33748640e-02]
 [2.66562793e-01 9.88989217e-02 1.70524158e-01 5.93092875e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [6.14849896e-02 2.07805951e-04 2.72197602e-02 1.22873600e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.68240444e-01 1.81223152e-01 2.01793747e-01 2.78175916e-01]
 [2.35395652e-01 5.15512709e-01 2.70620465e-01 2.36379002e-01]
 [1.70372899e-01 6.21310758e-01 1.43631228e-01 1.09562324e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [2.33758448e-01 1.54139084e-01 4.26614605e-01 1.18273990e-01]
 [4.70546096e-01 6.43369700e-01 4.12419156e-01 5.54321840e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
########################################
Q_table for num_episodes = 10000:
[[0.21222764 0.21094069 0.18852233 0.18505896]
 [0.10429224 0.08958777 0.02955794 0.14809458]
 [0.11279401 0.11200288 0.07639542 0.1014359 ]
 [0.07238097 0.08915613 0.01749123 0.12032012]
 [0.2211958  0.14776843 0.20374547 0.07889426]
 [0.         0.         0.         0.        ]
 [0.01133321 0.01184143 0.04817826 0.00733183]
 [0.         0.         0.         0.        ]
 [0.12660968 0.13772028 0.08887195 0.24276164]
 [0.19665547 0.45437291 0.23452585 0.35200007]
 [0.56630862 0.02783518 0.03383882 0.0871293 ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.19101712 0.37708072 0.53764188 0.06696474]
 [0.56863975 0.73881435 0.49396541 0.60574341]
 [0.         0.         0.         0.        ]]
########################################
Q_table for num_episodes = 50000:
[[0.35231867 0.16252721 0.3748051  0.17644232]
 [0.11004826 0.05919261 0.02825995 0.20153506]
 [0.22054773 0.03941685 0.07891465 0.04952346]
 [0.0304143  0.02454402 0.03780989 0.04763384]
 [0.45094803 0.28695926 0.03971802 0.06219748]
 [0.         0.         0.         0.        ]
 [0.23359052 0.01703528 0.13328323 0.00696655]
 [0.         0.         0.         0.        ]
 [0.07395916 0.09146009 0.08331465 0.4603538 ]
 [0.16715523 0.38928645 0.19173055 0.02498971]
 [0.11707152 0.47123332 0.15834314 0.2591235 ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.43799982 0.01286149 0.56870446 0.34680042]
 [0.5061679  0.84819269 0.40479586 0.5701664 ]
 [0.         0.         0.         0.        ]]
```

In [278…

```
def win_rate(env, Q, trials = 10000):
    rewardList = list()
    for episode in range(trials):
        current_state = env.reset()
        #env.render()
        episode_reward = 0
        for i in range(100):
            #print(Q[current_state])
            action = np.argmax(Q[current_state])
            #print(action)
            next_state, reward, done, info = env.step(action)
            #env.render()
            episode_reward += reward
            if done:
                #print("end")
                #env.render()
                break
            current_state = next_state
        rewardList.append(episode_reward)


    return np.mean(rewardList)
```

```
print(win_rate(env, Q_table50000))
```

```
0.41
```

Analyze: The more episodes the better winrate.

Appendix

This appendix includes references to APIs you may find useful. If the description sounds useful, check out the respective package's documentation for a much better description than we could provide.

Numpy

- np.zeros: N-dimensional tensor initialized to all 0s.
- np.ones: N-dimensional tensor initialized to all 1s.
- np.eye: N-dimensional tensor initialized to a diagonal matrix of 1s.
- np.random.choice: Randomly sample from a list, allowing you to specify weights.
- np.argmax: Index of the maximum element.
- np.abs: Absolute value.
- np.mean: Average across dimensions.
- np.sum: Sum across dimensions.

OpenAI Gym

- Environment (unwrapped):
  env.nS # Number of spaces.
  env.nA # Number of actions.
  env.P # Dynamics model.

```
In [ ]:
```