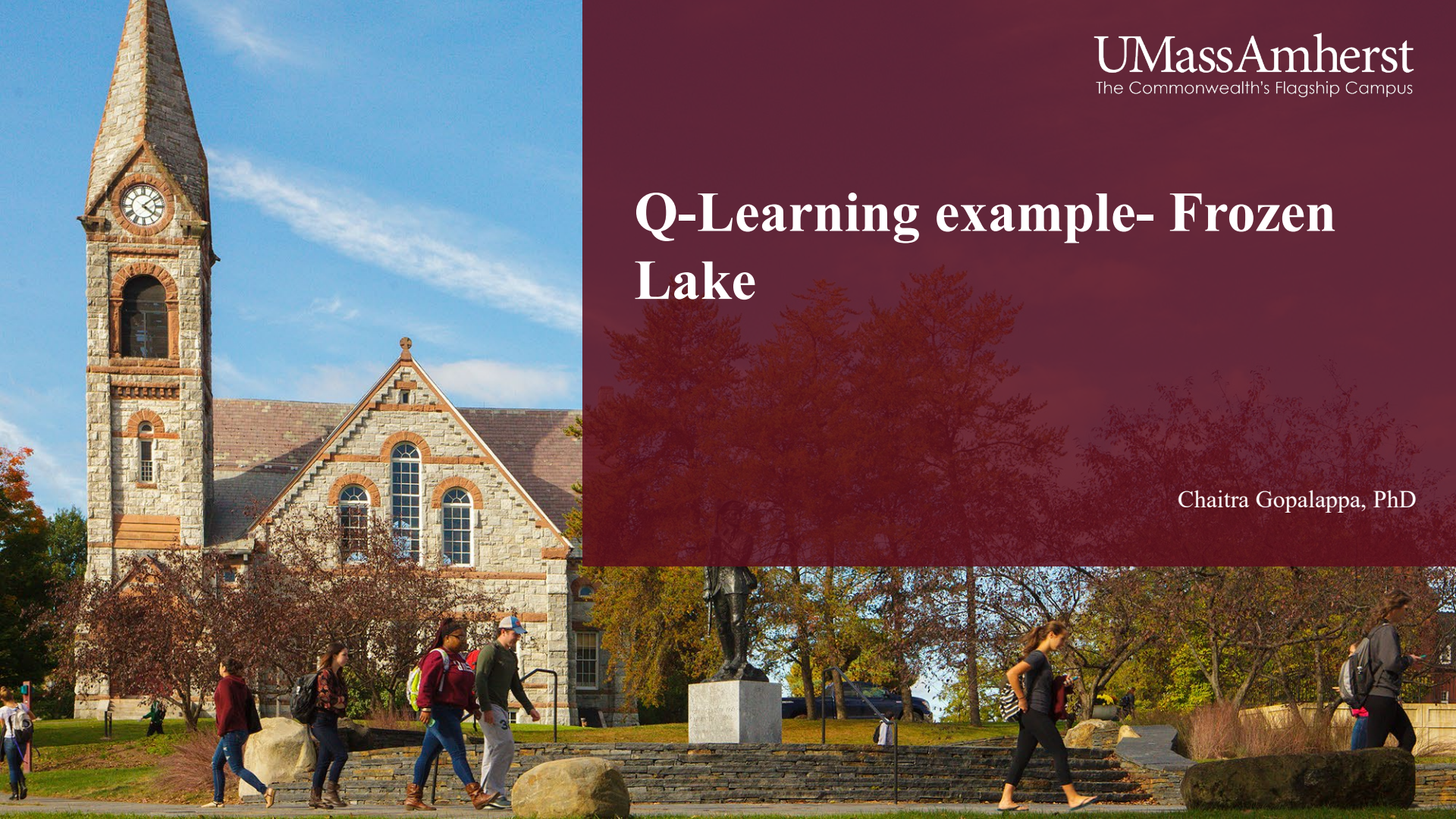

UMassAmherst
The Commonwealth's Flagship Campus

Q-Learning example- Frozen Lake

Chaitra Gopalappa, PhD

Frozen lake

MDP formulation

- Let X_t = location on grid at time t
- Let D_t = action to take at time t
- $\{X_t, D_t\}_t$ is MDP defined on tuple (Ω, P, A, R)
  - For 4X4 grid:
  - state space $\Omega = \{0,\ldots,15\}$
- $A = \{0$: Move left, 1: Move down, 2: Move right, 3: Move up$\}$
- $R_a(i,j) = \begin{cases} 1 \; if \; j = 15 \\ 0 \; o/w \end{cases}, \; \forall a$
- P =?

MDP formulation

- Let X_t = location on grid at time t
- Let D_t = action to take at time t
- $\{X_t, D_t\}_t$ is MDP defined on tuple (Ω, P, A, R)
  - For 4X4 grid:
  - state space Ω = {0,…,15}
- A = {0: Move left, 1: Move down, 2: Move right, 3: Move up}
- $R_a(i,j) = \begin{cases} 1 \; if \; j = 15 \\ 0 \; o/w \end{cases}, \; \forall a$
- $0 > p_a(i,j) > 1$ for some combinations of (i,j) because it is a slippery lake

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Sep 16 21:45:36 2023

@author: chaitrag
"""
import gym
import numpy as np
#from gym.utils.play import play
from tensorboardX import SummaryWriter
ENV_NAME = "FrozenLake-v1"
TEST_EPISODES = 50


'''Things to try
1. Epsilon greedy action (set exploration_prob=0.1) v. random action selection policy (set exploration_prob = 1)
2. Learning rate
'''
class QLearning:
    def __init__(self, num_states, num_actions, learning_rate=0.1, discount_factor=1, exploration_prob=1):
        self.num_states = num_states
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob
        self.q_table = np.zeros((num_states, num_actions))#initialize q-values

    def choose_action(self, state, random_action):
        if np.random.rand() < self.exploration_prob:
            return random_action # Exploration
        else:
            return np.argmax(self.q_table[state])  # Exploitation

    def update_q_table(self, state, action, next_state, reward):
        best_next_action = np.argmax(self.q_table[next_state, :])
        self.q_table[state, action] += self.learning_rate * (reward + self.discount_factor * self.q_table[next_state, best_next_action] - self.q

    def update_rates(self, iteration):
        self.exploration_prob = 1 / iteration

    def play_episode(self, env):
        total_reward = 0.0
        state = env.reset()
        while True:
            action = np.argmax(self.q_table[state])
            new_state, reward, is_done, _  = env.step(action)
            total_reward += reward
            if is_done:
                break
            state = new_state
        return total_reward

    def get_q_table(self):
        return (self.q_table)

if __name__ == "__main__":
```

Frozen lake

- Using inbuilt gym environment
- Basic functions in agent class

Frozen lake

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Sep 16 21:45:36 2023

@author: chaitrag
"""
import gym
import numpy as np
#from gym.utils.play import play
from tensorboardX import SummaryWriter
ENV_NAME = "FrozenLake-v1"
TEST_EPISODES = 50


'''Things to try
1. Epsilon greedy action (set exploration_prob=0.1) v. random action selection policy (set exploration_prob = 1)
2. Learning rate
'''
class QLearning:
    def __init__(self, num_states, num_actions, learning_rate=0.1, discount_factor=1, exploration_prob=1):
        self.num_states = num_states
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob
        self.q_table = np.zeros((num_states, num_actions))#initialize q-values

    def choose_action(self, state, random_action):
        if np.random.rand() < self.exploration_prob:
            return random_action # Exploration
        else:
            return np.argmax(self.q_table[state])  # Exploitation

    def update_q_table(self, state, action, next_state, reward):
        best_next_action = np.argmax(self.q_table[next_state, :])
        self.q_table[state, action] += self.learning_rate * (reward + self.discount_factor * self.q_table[next_state, best_next_action] - self.q

    def update_rates(self, iteration):
        self.exploration_prob = 1 / iteration

    def play_episode(self, env):
        total_reward = 0.0
        state = env.reset()
        while True:
            action = np.argmax(self.q_table[state])
            new_state, reward, is_done, _  = env.step(action)
            total_reward += reward
            if is_done:
                break
            state = new_state
        return total_reward

    def get_q_table(self):
        return (self.q_table)

if __name__ == "__main__":
```

Experiment with

- Exploration probability (epsilon in epsilon-greedy)
  - It works even if set to 1 and not decayed over time why?
- Discount factor

Frozen lake

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Sep 16 21:45:36 2023

@author: chaitrag
"""
import gym
import numpy as np
#from gym.utils.play import play
from tensorboardX import SummaryWriter
ENV_NAME = "FrozenLake-v1"
TEST_EPISODES = 50


'''Things to try
1. Epsilon greedy action (set exploration_prob=0.1) v. random action selection policy (set exploration_prob = 1)
2. Learning rate
'''
class QLearning:
    def __init__(self, num_states, num_actions, learning_rate=0.1, discount_factor=1, exploration_prob=1):
        self.num_states = num_states
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob
        self.q_table = np.zeros((num_states, num_actions))#initialize q-values

    def choose_action(self, state, random_action):
        if np.random.rand() < self.exploration_prob:
            return random_action # Exploration
        else:
            return np.argmax(self.q_table[state])  # Exploitation

    def update_q_table(self, state, action, next_state, reward):
        best_next_action = np.argmax(self.q_table[next_state, :])
        self.q_table[state, action] += self.learning_rate * (reward + self.discount_factor * self.q_table[next_state, best_next_action] - self.q

    def update_rates(self, iteration):
        self.exploration_prob = 1 / iteration

    def play_episode(self, env):
        total_reward = 0.0
        state = env.reset()
        while True:
            action = np.argmax(self.q_table[state])
            new_state, reward, is_done, _  = env.step(action)
            total_reward += reward
            if is_done:
                break
            state = new_state
        return total_reward

    def get_q_table(self):
        return (self.q_table)

if __name__ == "__main__":
```

Experiment with

- Exploration probability (epsilon in epsilon-greedy)
  - It works even if set to 1 and not decayed over time why? It is updating Q values for every (s,a) combination. Problem is small enough that all (s,a) combinations can be explored. Large state space and action space, exploring all state and action spaces may need more computes creating challenges with convergence. If some states have a very small chance of being visited, decaying exploration may speed up convergence. With the caveat that if some states are never visited their Q-values may never be updated.
- Discount factor

```
    def play_episode(self, env):
        total_reward = 0.0
        state = env.reset()
        while True:
            action = np.argmax(self.q_table[state])
            new_state, reward, is_done, _  = env.step(action)
            total_reward += reward
            if is_done:
                break
            state = new_state
        return total_reward

    def get_q_table(self):
        return (self.q_table)

if __name__ == "__main__":
    env = gym.make(ENV_NAME)
    ql_agent = QLearning(env.observation_space.n, env.action_space.n)
    #writer = SummaryWriter(comment="QL_frozen lake")
    NUM_EPISODES = 10000
    best_reward = 0.0
    for episode in range(NUM_EPISODES):
        state = env.reset()
        done = False

        while not done:

           action = ql_agent.choose_action(state,env.action_space.sample())#Choose an action
           next_state, reward, done, _  = env.step(action) # Take the chosen action and observe the next state and reward
           ql_agent.update_q_table(state, action, next_state, reward)#update Q-factors
           #ql_agent.update_rates(episode+1)#update epsilon
           state = next_state



        # Evaluate the learned Q-table
        total_reward = 0
        for _ in range(TEST_EPISODES):
            total_reward += ql_agent.play_episode(env)

        average_reward = total_reward / TEST_EPISODES
        #writer.add_scalar("reward", average_reward, episode)
        if average_reward > best_reward:
            print("Best reward updated %.3f -> %.3f" % (
                best_reward, average_reward))
            best_reward = average_reward
        if average_reward > 0.80:
            print("Solved in %d iterations!" % episode)
            break

    # Evaluate the learned Q-table
    #env = gym.wrappers.Monitor(env, "recording")
    q_table=ql_agent.get_q_table()
    #writer.close()
#tensorboard --logdir runs
```

Frozen lake

- Use additional application related metrics for evaluation of performance
- Here we know every successful episode will have returns (total reward per episode) of 1, and all others zero
- Considering randomness, taking average over a certain number of episodes

```
27
28  import gym
29  from gym.utils.play import play
30  import numpy as np
31
32  #ENV_NAME_1 = "LunarLander-v2"
33  ##ENV_NAME_2= "CartPole-v1"
34  #ENV_NAME_3 = "MountainCar-v0"
35  ENV_NAME_4 = "FrozenLake-v1"
36  q_table = np.array(
37        [[0.26092429, 0.24798012, 0.25145248, 0.2446796 ],
38         [0.15614233, 0.16296716, 0.17556336, 0.22719338],
39         [0.20532336, 0.17755031, 0.1660764 , 0.17034305],
40         [0.07799177, 0.10993306, 0.08465318, 0.14621668],
41         [0.26767313, 0.17209032, 0.1735222 , 0.19443704],
42         [0.        , 0.        , 0.        , 0.        ],
43         [0.21449847, 0.17372303, 0.14456769, 0.07848791],
44         [0.        , 0.        , 0.        , 0.        ],
45         [0.17042875, 0.19069884, 0.14923579, 0.30063623],
46         [0.23580664, 0.33822732, 0.19596721, 0.27093605],
47         [0.38922371, 0.34466193, 0.27502794, 0.19556056],
48         [0.        , 0.        , 0.        , 0.        ],
49         [0.        , 0.        , 0.        , 0.        ],
50         [0.18223229, 0.39927409, 0.43219388, 0.21861633],
51         [0.43178662, 0.64640394, 0.70177339, 0.5488019 ],
52         [0.        , 0.        , 0.        , 0.        ]])
53
54  env =gym.make(ENV_NAME_4, render_mode="human",is_slippery=False)#render_mode is how to display/vizualize the en
55
56  observation, info = env.reset(seed=42)
57  env.action_space.seed(42)#fixes the random seed for the action; during the learning phase there is a probabilit
58                           #useful for experimental analyses, e.g., when evaluating different rewards, we do not w
59
60  for _ in range(1000):
61      #action = env.action_space.sample() # take a random action
62      action = np.argmax(q_table[observation])
63      observation, reward, terminated, truncated, info = env.step(action)
64      #observation, reward, terminated, truncated, info = env.step(env.action_space.sample())
65
66      if terminated or truncated:
67        observation, info = env.reset()
68  env.close()
69
```

- Slippery =false makes it deterministic and q-table is no more optimal
  - Is it an MDP?

- Change the layout of the holes on the grid and the solution will no more work

UMassAmherst

The Commonwealth's Flagship Campus