SE4050
Deep Learning
4th Year, 1st Semester

Lab 08

Submitted to

Sri Lanka Institute of Information Technology

IT21166488

In partial fulfillment of the requirements for the

Bachelor of Science Special Honors Degree in Information Technology

04/10/2024

Question 1:

Code Snippet Edit

Markov_Decision_Process File:

1. Policy evaluation

Computing the utility, U.

$$U_k^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_{k-1}^\pi(s')$$

```
def iterativePolicyEvaluation(mdp, policy, numIterations=10):
    U = np.zeros(len(mdp.S))
    U_old = copy.copy(U)
    for t in range(numIterations):
        #type your code here
        for s in mdp.S:  # Iterate over each state
            a = policy  # Use the fixed policy action
            U[s] = sum([mdp.T[s, a, s_prime] * (mdp.R[s, a] + 0.9 * U_old[s_prime])
                        for s_prime in mdp.S])
        U_old = copy.copy(U)  # Update old utility values after each iteration

    return U

numIterations = 5
pl.figure(figsize=(15,3))    "figsize": Unknown word.
pl.suptitle('Utilities', fontsize=15)    "suptitle": Unknown word.
for a in range(4):
    pl.subplot(1,4,a+1)
    U = iterativePolicyEvaluation(mdp=mdp, policy=a, numIterations=numIterations)
    mdp.gridPlot(ax=pl.gca(), im=U.reshape(10,10), title='a='+mdp.A[a], cmap='jet')    "cmap": Unknown word.
pl.show()
#print(np.round(U.reshape(10,10),3))
```

```
#Value iteration
def valueIteration(mdp, numIterations=1):
    U = np.zeros(len(mdp.S))
    U_old = copy.copy(U)
    for t in range(numIterations):
        #type your code here
        for s in mdp.S:  # Iterate over each state
            U[s] = max([sum([mdp.T[s, a, s_prime] * (mdp.R[s, a] + 0.9 * U_old[s_prime])
                             for s_prime in mdp.S]) for a in range(len(mdp.A))])
        U_old = copy.copy(U)  # Update old utility values
    return U

def policyExtration(mdp, U):    "Extration": Unknown word.
    policy = np.zeros(len(mdp.S))
    #type your code here
    for s in mdp.S:  # Iterate over each state
        action_values = [sum([mdp.T[s, a, s_prime] * (mdp.R[s, a] + 0.9 * U[s_prime])
                              for s_prime in mdp.S]) for a in range(len(mdp.A))]
        policy[s] = np.argmax(action_values)  # Select action with highest value    "argmax": Unknown word.

    return policy

U = valueIteration(mdp, numIterations=2)
policy = policyExtration(mdp, U=U)    "Extration": Unknown word.
pl.figure(figsize=(3,3))    "figsize": Unknown word.
mdp.gridPlot(ax=pl.gca(), im=U.reshape(10,10), title='Utility', cmap='jet')    "cmap": Unknown word.
for s in range(100):
    x, y = mdp.s2xy(s)
    if policy[s] == 0:
        m='\u02C2'
    elif policy[s] == 1:
        m='\u02C3'
    elif policy[s] == 2:
        m='\u02C4'
    elif policy[s] == 3:
        m='\u02C5'
    pl.text(x-0.5,y-1,m,color='k',size=20)
pl.show()
```

```python
def policyIteration(mdp, numIterations=1):
    U_pi_k = np.zeros(len(mdp.S)) #initial values
    pi_k = np.random.randint(low=0,high=4,size=len(mdp.S),dtype=int) #initial policy    "dtype": Unknown word.
    pi_kp1 = copy.copy(pi_k)
    for t in range(numIterations):
        #Policy evaluation: compute U_pi_k
        #type your code here

        for i in range(100):  # iterate over all states
            s = mdp.S[i]
            a = pi_k[s]  # action according to current policy
            U_pi_k[s] = sum([mdp.T[s, a, s_prime] * (mdp.R[s, a] + 0.9 * U_pi_k[s_prime])
                             for s_prime in mdp.S])

        #Policy improvement
         #type your code here

        for s in mdp.S:
            action_values = []
            for a in range(len(mdp.A)):  # evaluate all possible actions
                action_value = sum([mdp.T[s, a, s_prime] * (mdp.R[s, a] + 0.9 * U_pi_k[s_prime])
                                    for s_prime in mdp.S])
                action_values.append(action_value)
            pi_kp1[s] = np.argmax(action_values)  # choose action with highest value    "argmax": Unknown word.

        # Check for convergence (optional)
        if np.array_equal(pi_k, pi_kp1):
            break

        pi_k = copy.copy(pi_kp1)

    return U_pi_k, pi_kp1

U_pi_k, pi_kp1 = policyIteration(mdp, numIterations=2)
```

Gridworld File:

```python
qodo Gen: Options | Test this class
class Q_Agent():
    # Intialise    "Intialise": Unknown word.
    qodo Gen: Options | Test this method
    def __init__(self, environment, epsilon=0.05, alpha=0.1, gamma=1):
        self.environment = environment
        self.q_table = dict() # Store all Q-values in dictionary of dictionaries
        for x in range(environment.height): # Loop through all possible grid spaces, create sub-dictionary fo
            for y in range(environment.width):
                self.q_table[(x,y)] = {'UP':0, 'DOWN':0, 'LEFT':0, 'RIGHT':0} # Populate sub-dictionary with

        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma

    qodo Gen: Options | Test this method
    def choose_action(self, available_actions):
        """Returns the optimal action from Q-Value table. If multiple optimal actions, chooses random choice.
        Will make an exploratory random action dependent on epsilon."""
        if np.random.uniform(0,1) < self.epsilon:
            action = available_actions[np.random.randint(0, len(available_actions))]
        else:
            q_values_of_state = self.q_table[self.environment.current_location]
            maxValue = max(q_values_of_state.values())
            action = np.random.choice([k for k, v in q_values_of_state.items() if v == maxValue])

        return action

    qodo Gen: Options | Test this method
    def learn(self, old_state, reward, new_state, action):
        """Updates the Q-value table using Q-learning"""
        q_values_of_state = self.q_table[new_state]
        max_q_value_in_new_state = max(q_values_of_state.values())
        current_q_value = self.q_table[old_state][action]

        self.q_table[old_state][action] = (1 - self.alpha) * current_q_value + self.alpha * (reward + self.ga
```

Python