DL – LAB 08 Answers – IT21161742

Git repo – "IT21226632/DL_Lab_08 (github.com)**"**

Markov Decision Process (MDP)

Modifications

```
[ ]  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 s0 in range(len(mdp.S)):
                 U[s0] = mdp.R[s0,policy] + mdp.discount*sum([mdp.T[s0,policy,s1]*U_old[s1] for s1 in
             U_old = copy.copy(U)
         return U

     numIterations = 5
     pl.figure(figsize=(15,3))
     pl.suptitle('Utilities', fontsize=15)
     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')
     pl.show()
     #print(np.round(U.reshape(10,10),3))
```

```
[ ]  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
         pi_kp1 = copy.copy(pi_k)
         for t in range(numIterations):
             #Policy evaluation: compute U_pi_k
               #type your code here
               U_pi_k_temp = copy.copy(U_pi_k)
               pi_k = copy.copy(pi_kp1)
               for s0 in range(len(mdp.S)):
                 U_pi_k[s0] = mdp.R[s0,pi_k[s0]] + mdp.discount*np.sum(mdp.T[s0,pi_k[s0],:]*U_pi_k_te
             #Policy improvement
               #type your code here
               for s0 in range(len(mdp.S)):
                 pi_kp1[s0] = np.argmax([mdp.R[s0,a] + mdp.discount*np.sum(mdp.T[s0,a,:]*U_pi_k[:]) f
         return U_pi_k, pi_kp1

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

```
[ ]  #Value iteration
     def valueIteration(mdp, numIterations=1):
         U = np.zeros(len(mdp.S))
         U_old = copy.copy(U)
         for t in range(numIterations):
             for s0 in range(len(mdp.S)):
                 U[s0] = max([mdp.R[s0,a] + mdp.discount*np.sum(mdp.T[s0,a,:]*U_old[:]) for a in np.a
             U_old = copy.copy(U)
         return U

     def policyExtration(mdp, U):
         policy = np.zeros(len(mdp.S))
         #type your code here
         for s0 in range(len(mdp.S)):
             policy[s0] = np.argmax([mdp.R[s0,a] + mdp.discount*np.sum(mdp.T[s0,a,:]*U[:]) for a in n
         return policy

     U = valueIteration(mdp, numIterations=2)
     policy = policyExtration(mdp, U=U)
     pl.figure(figsize=(3,3))
     mdp.gridPlot(ax=pl.gca(), im=U.reshape(10,10), title='Utility', cmap='jet')
     for s in range(100):
         x, y = mdp.s2xy(s)
         if policy[s] == 0:
             m='\u02C2'
         elif policy[s] == 1:
```

GridWorld

Modification

```
        def choose_action(self, available_actions):
            """Returns the optimal action from Q-Value table. If multiple optimal actions, chooses r
            Will make an exploratory random action dependent on epsilon."""
            # Epsilon-greedy action selection
            if random.uniform(0, 1) < self.epsilon:
                action = random.choice(available_actions)
            else:
                q_values = [self.q_table[self.environment.current_location][action] for action in av
                max_q_value = max(q_values)
                actions_with_max_q = [action for action, q_value in zip(available_actions, q_values)
                action = random.choice(actions_with_max_q)
            return action

        def learn(self, old_state, reward, new_state, action):
            """Updates the Q-value table using Q-learning"""
            old_value = self.q_table[old_state][action]
            next_max = max(self.q_table[new_state].values())
            new_value = (1 - self.alpha) * old_value + self.alpha * (reward + self.gamma * next_max)
            self.q_table[old_state][action] = new_value
```

Environment grid

```
class GridWorld:
    ## Initialise starting data
    def __init__(self):
        # Set information about the gridworld
        self.height = 16
        self.width = 16
        self.grid = np.zeros(( self.height, self.width)) - 1

        # Set random start location for the agent
        self.current_location = ( 4, np.random.randint(0,5))

        # Set locations for the bomb and the gold
        self.bomb_location = (1,3)
        self.gold_location = (0,3)
        self.terminal_states = [ self.bomb_location, self.gold_location]
```

02

Step 2: Explanation of Model-Based vs. Model-Free Algorithms

Model-Based Algorithms

- **Definition**: These algorithms assume a known model of the environment (i.e., transition probabilities and rewards) and use this model to compute the optimal policy.
- **Example**: **Value Iteration**, **Policy Iteration**.
- **Advantages**: Directly uses the model to plan, can be efficient in stationary environments.
- **Disadvantages**: Requires knowing or learning the model (transition and reward functions).

Model-Free Algorithms

- **Definition**: These algorithms do not assume a known model of the environment. Instead, they learn the policy directly from interactions with the environment (e.g., Q-learning).
- **Example**: **Q-Learning**, **SARSA**.
- **Advantages**: Can be used in environments where the model is not available or too complex to calculate.
- **Disadvantages**: Usually slower to converge because it relies on sampling to learn.