SE4050

Deep Learning

4th Year, 1st Semester

Lab 08

Registration Number: IT21347962

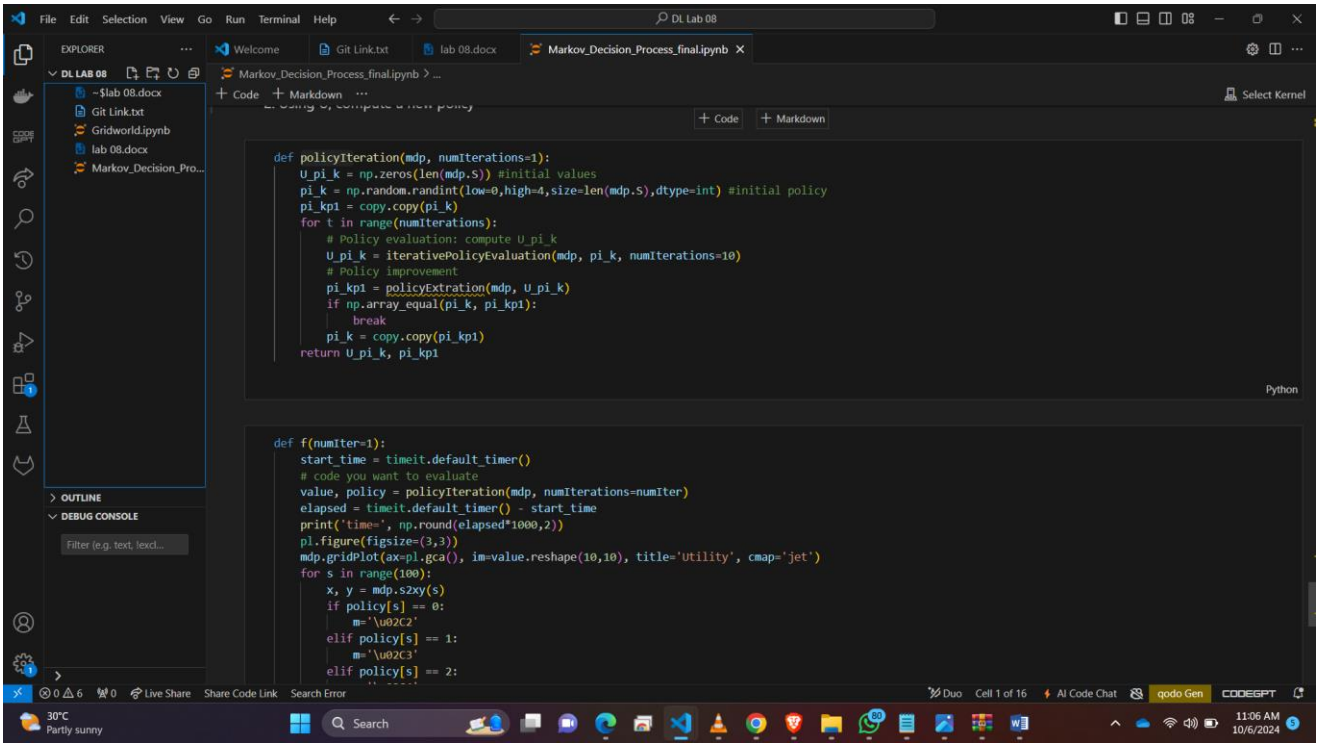
Name: Siribaddana K.

10/03/2024

Question 1

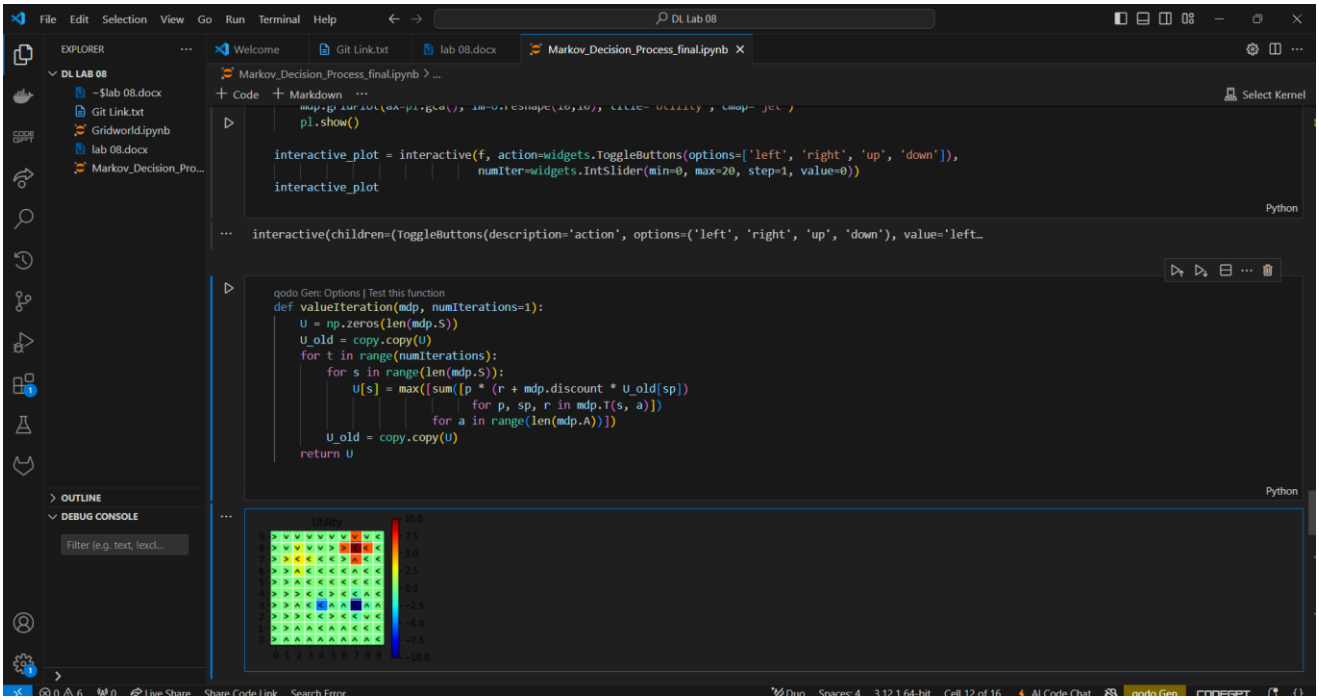
Task 3

Updated part in the Markov Decision Process file



```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
    pi_kp1 = copy.copy(pi_k)
    for t in range(numIterations):
        # Policy evaluation: compute U_pi_k
        U_pi_k = iterativePolicyEvaluation(mdp, pi_k, numIterations=10)
        # Policy improvement
        pi_kp1 = policyExtration(mdp, U_pi_k)
        if np.array_equal(pi_k, pi_kp1):
            break
        pi_k = copy.copy(pi_kp1)
    return U_pi_k, pi_kp1
```

```python
def f(numIter=1):
    start_time = timeit.default_timer()
    # code you want to evaluate
    value, policy = policyIteration(mdp, numIterations=numIter)
    elapsed = timeit.default_timer() - start_time
    print('time=', np.round(elapsed*1000,2))
    pl.figure(figsize=(3,3))
    mdp.gridPlot(ax=pl.gca(), im=value.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:
            m='\u02C3'
        elif policy[s] == 2:
```



```python
    pl.show()

interactive_plot = interactive(f, action=widgets.ToggleButtons(options=['left', 'right', 'up', 'down']),
                               numIter=widgets.IntSlider(min=0, max=20, step=1, value=0))
interactive_plot
```

```
interactive(children=(ToggleButtons(description='action', options=('left', 'right', 'up', 'down'), value='left…
```

```python
def valueIteration(mdp, numIterations=1):
    U = np.zeros(len(mdp.S))
    U_old = copy.copy(U)
    for t in range(numIterations):
        for s in range(len(mdp.S)):
            U[s] = max([sum([p * (r + mdp.discount * U_old[sp])
                             for p, sp, r in mdp.T(s, a)])
                        for a in range(len(mdp.A))])
        U_old = copy.copy(U)
    return U
```

Updpated part in the GridWorld (QLearning) notebook

```python
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."""
    #type your code here
    current_state = self.environment.current_location  # Get the current location from the environment

    if random.uniform(0, 1) < self.epsilon:  # Exploration: choose a random action
        return random.choice(available_actions)
    else:
        # Exploitation: choose the action with the highest Q-value
        q_values = self.q_table[current_state]  # Get Q-values for current state
        max_value = max([q_values[action] for action in available_actions])  # Get max Q-value among available actions
        # Find all actions with the max Q-value (in case of ties)
        best_actions = [action for action in available_actions if q_values[action] == max_value]
        return random.choice(best_actions)

def learn(self, old_state, reward, new_state, action):
    """Updates the Q-value table using Q-learning"""
    #type your code here
    # Get the current Q-value for the action taken in the old state
    current_q_value = self.q_table[old_state][action]

    # Get the maximum Q-value for the new state (future reward)
    max_future_q_value = max(self.q_table[new_state].values())

    # Q-learning update rule
    new_q_value = current_q_value + self.alpha * (reward + self.gamma * max_future_q_value - current_q_value)

    # Update the Q-value table with the new Q-value
    self.q_table[old_state][action] = new_q_value
```

Updated part in the GridWorld (QLearning) notebook Grid Size

```python
class GridWorld:
    ## Initialise starting data
    def __init__(self):
        # Set information about the gridworld
        self.height = 12
        self.width = 12
        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]

        # Set grid rewards for special cells
        self.grid[ self.bomb_location[0], self.bomb_location[1]] = -10
        self.grid[ self.gold_location[0], self.gold_location[1]] = 10

        # Set available actions
        self.actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']


    ## Put methods here:
    def get_available_actions(self):
        """Returns possible actions"""
        return self.actions

    def agent_on_map(self):
        """Prints out current location of the agent on the grid (used for debugging)"""
        grid = np.zeros(( self.height, self.width))
```

Question 2

Value Iteration was implemented to compute the optimal policy by updating the utility values for each state. The following code was added:

```
def valueIteration(mdp, numIterations=1):
    U = np.zeros(len(mdp.S))
    U_old = copy.copy(U)
    for t in range(numIterations):
        for s in range(len(mdp.S)):
            U[s] = max([sum([p * (r + mdp.discount * U_old[sp])
                            for p, sp, r in mdp.T(s, a)])
                        for a in range(len(mdp.A))])
        U_old = copy.copy(U)
    return U
```

Question 3

The policy extraction method was implemented to derive the policy from the utility values. The following code was added:

```
def policyExtration(mdp, U):
    policy = np.zeros(len(mdp.S))
    for s in range(len(mdp.S)):
        policy[s] = np.argmax([sum([p * (r + mdp.discount * U[sp])
                                for p, sp, r in mdp.T(s, a)])
                            for a in range(len(mdp.A))])
    return policy
```

Question 4

Policy Iteration was implemented to iteratively evaluate and improve the policy based on the utility values. The following code was added:

```
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):
        U_pi_k = iterativePolicyEvaluation(mdp, pi_k, numIterations=10)
        pi_kp1 = policyExtration(mdp, U_pi_k)
        if np.array_equal(pi_k, pi_kp1):
            break
        pi_k = copy.copy(pi_kp1)
    return U_pi_k, pi_kp1
```