3. Q-learning Algorithm

Extension Note: Project 5 due date has been extended by 1 **more** day to **September 6 23:59UTC** .

In this section, you will implement the Q-learning algorithm, which is a model-free algorithm used to learn an optimal Q-function. In the tabular setting, the algorithm maintains the Q-value for all possible state-action pairs. Starting from a random Q-function, the agent continuously collects experiences $(s, c, R(s,c), s')$ and updates its Q-function.

From now on, we will refer to $c = (a, b)$ as “an action" although it really is an action with an object.

Q-learning Algorithm

- The agent plays an action c at state s, getting a reward $R(s,c)$ and observing the next state s'.
- Update the single Q-value corresponding to each such transition:

$$Q(s,c) \leftarrow (1-\alpha)\,Q(s,c) + \alpha\left[R(s,c) + \gamma \max_{c' \in C} Q(s', c')\right]$$

Tip: We recommend you implement all functions from this tab and the next one before submitting your code online. Make sure you achieve reasonable performance on the *Home World* game

Single step update

1.0/1 point (graded)
Write a function `tabular_q_learning` that updates the single Q-value, given the transition date $(s, c, R(s,c), s')$.

Reminder: You should implement this function locally first. You can read through the next tab to understand the context in which this function is called

Available Functions: You have access to the NumPy python library as `np` . You should also use constants `ALPHA` and `GAMMA` in your code

```
    Returns:
        None
    """
    if not terminal:
        v_func = np.max(q_func[next_state_1, next_state_2])
    else:
        v_func = 0
    q_func[current_state_1, current_state_2, action_index, object_index] =\
        (1 - ALPHA) * q_func[current_state_1, current_state_2, action_index, object_index] +\
        ALPHA * (reward + GAMMA * v_func)

    return None  # This function shouldn't return anything
```

Press ESC then TAB or click outside of the code editor to exit

Correct

```
def tabular_q_learning(q_func, current_state_1, current_state_2, action_index,
                       object_index, reward, next_state_1, next_state_2,
                       terminal):
    """Update q_func for a given transition

    Args:
        q_func (np.ndarray): current Q-function
        current_state_1, current_state_2 (int, int): two indices describing the current state
        action_index (int): index of the current action
        object_index (int): index of the current object
        reward (float): the immediate reward the agent recieves from playing current command
        next_state_1, next_state_2 (int, int): two indices describing the next state
        terminal (bool): True if this epsiode is over

    Returns:
        None
    """
    if terminal:
        maxq_next_state = 0
    else:
        q_values_next_state = q_func[next_state_1, next_state_2, :, :]
        maxq_next_state = np.max(q_values_next_state)

    q_value = q_func[current_state_1, current_state_2, action_index,
                     object_index]
    q_func[current_state_1, current_state_2, action_index, object_index] = (
        1 - ALPHA) * q_value + ALPHA * (reward + GAMMA * maxq_next_state)
```

Test results

See full output

```
CORRECT
```

See full output

Submit You have used 3 of 25 attempts

Answers are displayed within the problem

Epsilon-greedy exploration

1.0/1 point (graded)

Note that the Q-learning algorithm does not specify how we should interact in the world so as to learn quickly. It merely updates the values based on the experience collected. If we explore randomly, i.e., always select actions at random, we would most likely not get anywhere. A better option is to exploit what we have already learned, as summarized by current Q-values. We can always act greedily with respect to the current estimates, i.e., take an action $\pi(s) = \arg\max_{c \in C} Q(s, c)$. Of course, early on, these are not necessarily very good actions. For this reason, a typical exploration strategy is to follow a so-called ε-greedy policy: with probability ε take a random action out of C with probability $1 - \varepsilon$ follow $\pi(s) = \arg\max_{c \in C} Q(s, c)$. The value of ε here balances exploration vs exploitation. A large value of ε means exploring more (randomly), not using much of what we have learned. A small ε, on the other hand, will generate experience consistent with the current estimates of Q-values.

Now you will write a function `epsilon_greedy` that implements the ε-greedy exploration policy using the current Q-function.

Reminder: You should implement this function locally first. You can read through the next tab to understand the context in which this function is called

Available Functions: You have access to the NumPy python library as `np`. Your code should also use constants `NUM_ACTIONS` and `NUM_OBJECTS`.

```
    Args:
        state_1, state_2 (int, int): two indices describing the current state
        q_func (np.ndarray): current Q-function
        epsilon (float): the probability of choosing a random command

    Returns:
        (int, int): the indices describing the action/object to take
    """
    if np.random.random() > epsilon:
        q_state = q_func[state_1, state_2]
        action_index, object_index = np.unravel_index(q_state.argmax(), q_state.shape)
    else:
        action_index = np.random.randint(NUM_ACTIONS)
        object_index = np.random.randint(NUM_OBJECTS)
    return (action_index, object_index)
```

Press ESC then TAB or click outside of the code editor to exit

Correct

```
def epsilon_greedy(state_1, state_2, q_func, epsilon):
    """Returns an action selected by an epsilon-Greedy exploration policy

    Args:
        state_1, state_2 (int, int): two indices describing the current state
        q_func (np.ndarray): current Q-function
        epsilon (float): the probability of choosing a random command

    Returns:
        (int, int): the indices describing the action/object to take
    """
    coin = np.random.random_sample()
    if coin < epsilon:
        action_index = np.random.randint(NUM_ACTIONS)
        object_index = np.random.randint(NUM_OBJECTS)
    else:
        q_values = q_func[state_1, state_2, :, :]
        (action_index,
         object_index) = np.unravel_index(np.argmax(q_values, axis=None),
                                          q_values.shape)

    return (action_index, object_index)
```

Test results

See full output

```
CORRECT
```

See full output

Submit You have used 10 of 25 attempts

Answers are displayed within the problem

Discussion

Show Discussion

Topic: Unit 5 Reinforcement Learning (2 weeks) :Project 5: Text-Based Game / 3. Q-learning Algorithm