Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ И УПРАВЛЕНИЯ

Отчёт к лабораторным работам по курсу

«Методы машинного обучения»

Лабораторная работа №5 «Обучение на основе временны’х различий»

Выполнил:

студент(ка) группы ИУ5И-21М Лю Бэйбэй

подпись, дата

Проверил:

к.т.н., доц., Виноградовой М.В.

подпись, дата

Москва, 2022 г.

1. описание задания

На основе рассмотренного на лекции примера реализуйте следующие алгоритмы:
SARSA
Q-обучение
Двойное Q-обучение
для любой среды обучения с подкреплением (кроме рассмотренной на лекции среды Toy Text / Frozen Lake) из библиотеки Gym (или аналогичной библиотеки).

2. Текст программы и экранные формы с примерами выполнения программы.

```
env = gym.make('Blackjack-v1')
action_space_size = env.action_space.n
num_episodes = 50000
learning_rate = 0.1
discount_rate = 0.95
epsilon_start = 1
epsilon_end = 0.001
decay_factor = (epsilon_end/epsilon_start)**(1/num_episodes)

def choose_action(q_table, state: tuple[int, int, bool]) -> int:
    if random.uniform(0,1) > epsilon:
        return (int(np.argmax(q_table[state])))
    else:
        return (env.action_space.sample())
```

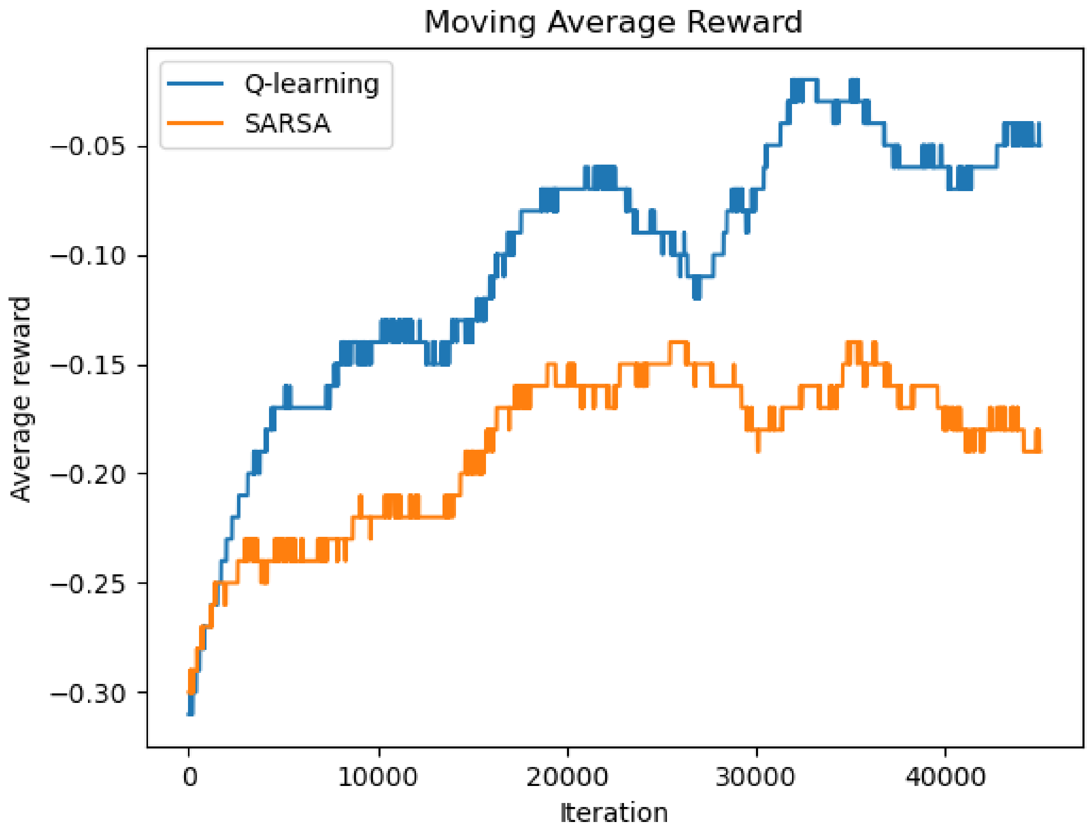
Sarsa

```
q_table_S = defaultdict(lambda: np.zeros(action_space_size))
rewards_list_S = []
epsilon = epsilon_start

for episode in range(num_episodes):
    state, info = env.reset()
    done = False
    rewards_current_episode = 0
    action = choose_action(q_table_S, state)

    while not done:
        new_state, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated

        next_action = choose_action(q_table_S, new_state)

        q_table_S[state][action] = q_table_S[state][action] + learning_rate*(reward
+ discount_rate*q_table_S[new_state][next_action] - q_table_S[state][action])
        state = new_state
        action = next_action
```

```
            rewards_current_episode += reward

        epsilon = epsilon*decay_factor
        rewards_list_S.append(rewards_current_episode)
env.close()
```

Q-learning

```
q_table_Q = defaultdict(lambda: np.zeros(action_space_size))
rewards_list_Q = []
epsilon = epsilon_start

for episode in range(num_episodes):
        state, info = env.reset()
        done = False
        rewards_current_episode = 0

        while not done:
            action = choose_action(q_table_Q, state)

            new_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated

            q_table_Q[state][action] = q_table_Q[state][action] + learning_rate*(reward
+ discount_rate*int(np.argmax(q_table_Q[new_state])) - q_table_Q[state][action])
            state = new_state
            rewards_current_episode += reward

            if done: break

        epsilon = epsilon*decay_factor
        rewards_list_Q.append(rewards_current_episode)
env.close()
```



Double-learning:

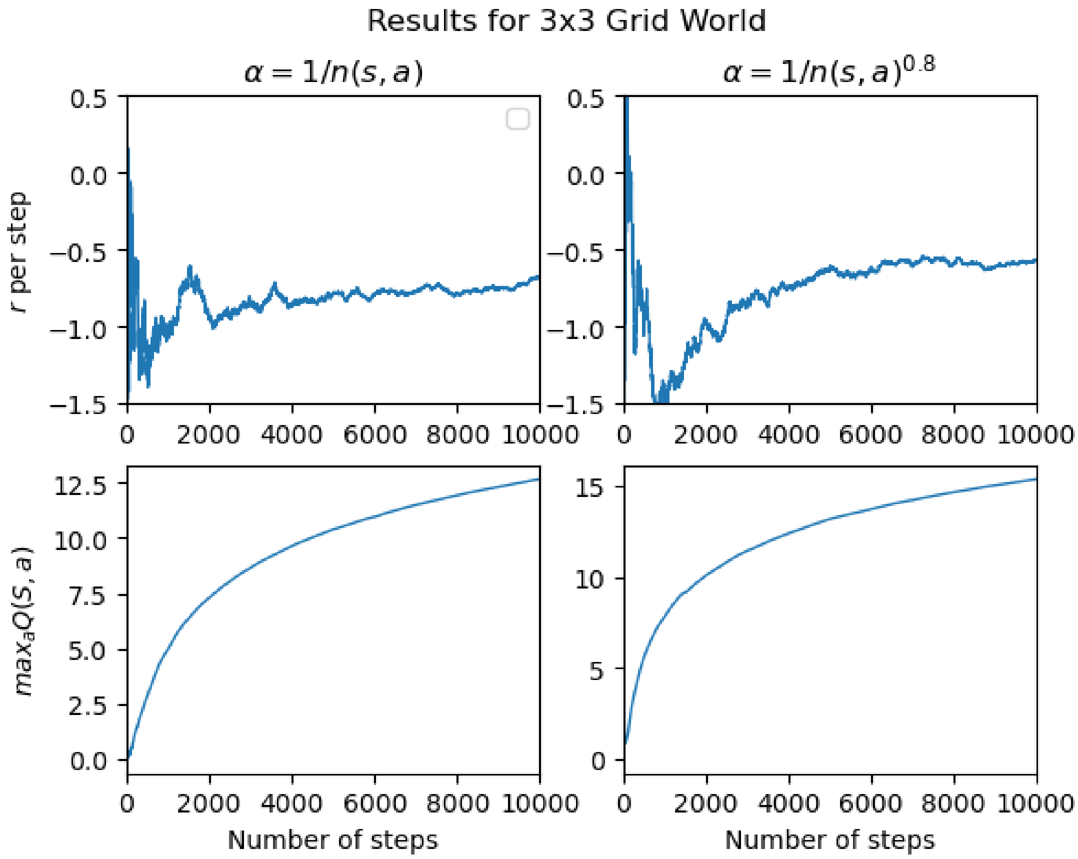
```
class GridWorldEnv(Env):

    UP = 0
    LEFT = 1
    RIGHT = 2
    DOWN = 3

    def __init__(self, height, width):
        self.action_space = Discrete(4)
        self.height = height
        self.width = width
        self.observation_space = Discrete(self.height * self.width)
        # initial state
        self.state = 0
        self.end_state = self.height * self.width - 1

    def step(self, action):
        '''
        Function applying the action to current state.
        If the action makes the agent go out of bounds, the agent remains in the same place.
```

```
        '''

        terminated = False

        # sample reward randomly
        if random.binomial(n=1, p=0.5, size=1) == 0:
            reward = -12
        else:
            reward = 10

        # check if currently at target
        if self.state == self.end_state:
            reward = 5
            terminated = True

        # otherwise, check action and apply if valid
        elif action == self.UP:
            if self.state + self.width <= self.end_state:
                self.state += self.width

        elif action == self.LEFT:
            if self.state % self.width != 0:
                self.state -= 1

        elif action == self.RIGHT:
            if self.state % self.width != self.width - 1:
                self.state += 1

        elif action == self.DOWN:
            if self.state >= self.width:
                self.state -= self.width

        information = {}

        return self.state, reward, terminated, information

    def reset(self):
        '''
        Function that resets the environment after termination.
        '''
        self.state = 0
        information = {}

        return self.state, information
```

```
    def render(self):
        pass

GAMMA = 0.95
MAX_STEPS = 10000

def Double_Q_learning(env, n_episodes, mode):
    '''
    Implementation of the Double Q-learning algorithm.
    '''

    # create Q_a, Q_b tables
    Q_a = np.zeros((env.observation_space.n, env.action_space.n))
    Q_b = np.zeros((env.observation_space.n, env.action_space.n))

    # variables to keep track of reward running average for first MAX_STEPS
number of steps
    running_avg_per_step = []
    running_avg = 0
    total_steps_taken = 0

    # variables to keep track of max Q(S,a) for the starting state over all episodes
    running_avg_max_Q_per_step = []
    running_avg_max_Q = 0

    for episode in tqdm(range(n_episodes)):

        # for double Q-learning, as the original paper states, the variable used for
the learning rate is:
        # n(s,a) = n_a(s,a) if Q_a is updated
        # otherwise, n(s,a) = n_b(s,a)
        n_s_a_Qa = np.zeros((env.observation_space.n, env.action_space.n))
        n_s_a_Qb = np.zeros((env.observation_space.n, env.action_space.n))

        # keep track of number of times a state is visited (used to calculate epislon)
        n_s = np.zeros(env.observation_space.n)

        # keep track of number of times a state-action pair is visited (used to
calculate alpha)
        n_s_a = np.zeros((env.observation_space.n, env.action_space.n))

        # reset env
```

```
        state, _ = env.reset()

        # update n_s for initial state
        n_s[state] += 1

        # for starting state: get maximal Q value, get running average
        max_starting_Q_a = np.max(Q_a[state])
        max_starting_Q_b = np.max(Q_a[state])
        max_starting_Q = max_starting_Q_a if max_starting_Q_a >
max_starting_Q_b else max_starting_Q_b
        running_avg_max_Q = (total_steps_taken * running_avg_max_Q +
max_starting_Q) / (total_steps_taken + 1)
        running_avg_max_Q_per_step.append(running_avg_max_Q)

        terminated = False

        while not terminated:

            # epsilon-greedy exploration
            epsilon = get_epsilon(n_s[state])

            if np.random.rand() < epsilon:
                action = env.action_space.sample()
            else:
                # get action with maximal Q value per Double Q-learning
                Q = Q_a[state] + Q_b[state]
                max = np.where(np.max(Q) == Q)[0]
                action = np.random.choice(max)

            # apply action, get reward
            next_state, reward, terminated, _ = env.step(action)

            # update n_s for new state
            n_s[next_state] += 1

            # update n_s_a for (state, action)
            n_s_a[state, action] += 1

            # update A or B with equal probability
            if np.random.rand() < 0.5:
                # update Q_a
                # increment count for (state, action) pair matrix for Qa
                n_s_a_Qa[state, action] = n_s_a_Qa[state, action] + 1
                # get alpha
```

```
                alpha = get_lr(n_s_a_Qa[state, action], mode)
                c = alpha * (reward + GAMMA * Q_b[next_state,
np.argmax(Q_a[next_state])] - Q_a[state, action])
                Q_a[state, action] = Q_a[state, action] + c
            else:
                # update Q_b
                # increment count for (state, action) pair matrix for Qb
                n_s_a_Qb[state, action] = n_s_a_Qb[state, action] + 1
                # get alpha
                alpha = get_lr(n_s_a_Qb[state, action], mode)
                c = alpha * (reward + GAMMA * Q_a[next_state,
np.argmax(Q_b[next_state])] - Q_b[state, action])
                Q_b[state, action] = Q_b[state, action] + c

            # update state to be next state
            state = next_state

            # calculate running average of rewards for first MAX_STEPS steps
            if total_steps_taken < MAX_STEPS:
                running_avg = (total_steps_taken * running_avg + reward) /
(total_steps_taken + 1)
                running_avg_per_step.append(running_avg)

            total_steps_taken += 1

    return running_avg_per_step, running_avg_max_Q_per_step
```


Results for 3x3 Grid World
$\alpha = 1/n(s,a)$
$\alpha = 1/n(s,a)^{0.8}$
r per step
$max_a Q(S,a)$
0.5
0.0
−0.5
−1.0
−1.5
0
2000
4000
6000
8000
10000
12.5
10.0
7.5
5.0
2.5
15
10
5
Number of steps
Number of steps