|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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

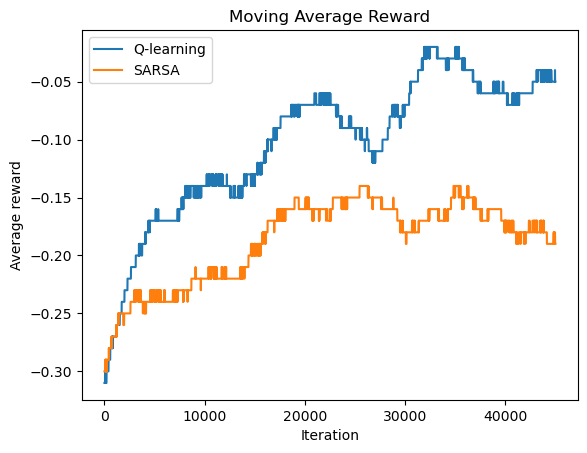
rewards\_current\_episode += reward

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

