|  |  |
| --- | --- |
| 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 №6 «Обучение на основе глубоких Q-сетей»**

**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 рассмотренных на лекции примеров реализуйте алгоритм DQN.

В качестве среды можно использовать классические среды (в этом случае используется полносвязная архитектура нейронной сети).

В качестве среды можно использовать игры Atari (в этом случае используется сверточная архитектура нейронной сети).

В случае реализации среды на основе сверточной архитектуры нейронной сети +1 балл за экзамен.

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

class DeepQNetwork:

def \_\_init\_\_(

self,

n\_actions,

n\_features,

learning\_rate=0.01,

reward\_decay=0.9,

e\_greedy=0.9,

replace\_target\_iter=300,

memory\_size=500,

batch\_size=32,

e\_greedy\_increment=None,

output\_graph=False,

):

self.n\_actions = n\_actions

self.n\_features = n\_features

self.lr = learning\_rate

self.gamma = reward\_decay

self.epsilon\_max = e\_greedy

self.replace\_target\_iter = replace\_target\_iter

self.memory\_size = memory\_size

self.batch\_size = batch\_size

self.epsilon\_increment = e\_greedy\_increment

self.epsilon = 0 if e\_greedy\_increment is not None else self.epsilon\_max

# total learning step

self.learn\_step\_counter = 0

# initialize zero memory [s, a, r, s\_]

self.memory = np.zeros((self.memory\_size, n\_features \* 2 + 2))

# consist of [target\_net, evaluate\_net]

self.\_build\_net()

t\_params = tf.get\_collection('target\_net\_params')

e\_params = tf.get\_collection('eval\_net\_params')

self.replace\_target\_op = [tf.assign(t, e) for t, e in zip(t\_params, e\_params)]

self.sess = tf.Session()

if output\_graph:

# $ tensorboard --logdir=logs

# tf.train.SummaryWriter soon be deprecated, use following

tf.summary.FileWriter("logs/", self.sess.graph)

self.sess.run(tf.global\_variables\_initializer())

self.cost\_his = []

def \_build\_net(self):

# ------------------ build evaluate\_net ------------------

self.s = tf.placeholder(tf.float32, [None, self.n\_features], name='s') # input

self.q\_target = tf.placeholder(tf.float32, [None, self.n\_actions], name='Q\_target') # for calculating loss

with tf.variable\_scope('eval\_net'):

# c\_names(collections\_names) are the collections to store variables

c\_names, n\_l1, w\_initializer, b\_initializer = \

['eval\_net\_params', tf.GraphKeys.GLOBAL\_VARIABLES], 10, \

tf.random\_normal\_initializer(0., 0.3), tf.constant\_initializer(0.1) # config of layers

# first layer. collections is used later when assign to target net

with tf.variable\_scope('l1'):

w1 = tf.get\_variable('w1', [self.n\_features, n\_l1], initializer=w\_initializer, collections=c\_names)

b1 = tf.get\_variable('b1', [1, n\_l1], initializer=b\_initializer, collections=c\_names)

l1 = tf.nn.relu(tf.matmul(self.s, w1) + b1)

# second layer. collections is used later when assign to target net

with tf.variable\_scope('l2'):

w2 = tf.get\_variable('w2', [n\_l1, self.n\_actions], initializer=w\_initializer, collections=c\_names)

b2 = tf.get\_variable('b2', [1, self.n\_actions], initializer=b\_initializer, collections=c\_names)

self.q\_eval = tf.matmul(l1, w2) + b2

with tf.variable\_scope('loss'):

self.loss = tf.reduce\_mean(tf.squared\_difference(self.q\_target, self.q\_eval))

with tf.variable\_scope('train'):

self.\_train\_op = tf.train.RMSPropOptimizer(self.lr).minimize(self.loss)

# ------------------ build target\_net ------------------

self.s\_ = tf.placeholder(tf.float32, [None, self.n\_features], name='s\_') # input

with tf.variable\_scope('target\_net'):

# c\_names(collections\_names) are the collections to store variables

c\_names = ['target\_net\_params', tf.GraphKeys.GLOBAL\_VARIABLES]

# first layer. collections is used later when assign to target net

with tf.variable\_scope('l1'):

w1 = tf.get\_variable('w1', [self.n\_features, n\_l1], initializer=w\_initializer, collections=c\_names)

b1 = tf.get\_variable('b1', [1, n\_l1], initializer=b\_initializer, collections=c\_names)

l1 = tf.nn.relu(tf.matmul(self.s\_, w1) + b1)

# second layer. collections is used later when assign to target net

with tf.variable\_scope('l2'):

w2 = tf.get\_variable('w2', [n\_l1, self.n\_actions], initializer=w\_initializer, collections=c\_names)

b2 = tf.get\_variable('b2', [1, self.n\_actions], initializer=b\_initializer, collections=c\_names)

self.q\_next = tf.matmul(l1, w2) + b2

def store\_transition(self, s, a, r, s\_):

if not hasattr(self, 'memory\_counter'):

self.memory\_counter = 0

transition = np.hstack((s, [a, r], s\_))

# replace the old memory with new memory

index = self.memory\_counter % self.memory\_size

self.memory[index, :] = transition

self.memory\_counter += 1

def choose\_action(self, observation):

# to have batch dimension when feed into tf placeholder

observation = observation[np.newaxis, :]

if np.random.uniform() < self.epsilon:

# forward feed the observation and get q value for every actions

actions\_value = self.sess.run(self.q\_eval, feed\_dict={self.s: observation})

action = np.argmax(actions\_value)

else:

action = np.random.randint(0, self.n\_actions)

return action

def learn(self):

# check to replace target parameters

if self.learn\_step\_counter % self.replace\_target\_iter == 0:

self.sess.run(self.replace\_target\_op)

print('\ntarget\_params\_replaced\n')

# sample batch memory from all memory

if self.memory\_counter > self.memory\_size:

sample\_index = np.random.choice(self.memory\_size, size=self.batch\_size)

else:

sample\_index = np.random.choice(self.memory\_counter, size=self.batch\_size)

batch\_memory = self.memory[sample\_index, :]

q\_next, q\_eval = self.sess.run(

[self.q\_next, self.q\_eval],

feed\_dict={

self.s\_: batch\_memory[:, -self.n\_features:], # fixed params

self.s: batch\_memory[:, :self.n\_features], # newest params

})

# change q\_target w.r.t q\_eval's action

q\_target = q\_eval.copy()

batch\_index = np.arange(self.batch\_size, dtype=np.int32)

eval\_act\_index = batch\_memory[:, self.n\_features].astype(int)

reward = batch\_memory[:, self.n\_features + 1]

q\_target[batch\_index, eval\_act\_index] = reward + self.gamma \* np.max(q\_next, axis=1)

# train eval network

\_, self.cost = self.sess.run([self.\_train\_op, self.loss],

feed\_dict={self.s: batch\_memory[:, :self.n\_features],

self.q\_target: q\_target})

self.cost\_his.append(self.cost)

# increasing epsilon

self.epsilon = self.epsilon + self.epsilon\_increment if self.epsilon < self.epsilon\_max else self.epsilon\_max

self.learn\_step\_counter += 1

def plot\_cost(self):

import matplotlib.pyplot as plt

plt.plot(np.arange(len(self.cost\_his)), self.cost\_his)

plt.ylabel('Cost')

plt.xlabel('training steps')

plt.show()

UNIT = 40 # pixels

MAZE\_H = 4 # grid height

MAZE\_W = 4 # grid width

class Maze(tk.Tk, object):

def \_\_init\_\_(self):

super(Maze, self).\_\_init\_\_()

self.action\_space = ['u', 'd', 'l', 'r']

self.n\_actions = len(self.action\_space)

self.n\_features = 2

self.title('maze')

self.geometry('{0}x{1}'.format(MAZE\_W \* UNIT, MAZE\_H \* UNIT))

self.\_build\_maze()

def \_build\_maze(self):

self.canvas = tk.Canvas(self, bg='white',

height=MAZE\_H \* UNIT,

width=MAZE\_W \* UNIT)

# create grids

for c in range(0, MAZE\_W \* UNIT, UNIT):

x0, y0, x1, y1 = c, 0, c, MAZE\_H \* UNIT

self.canvas.create\_line(x0, y0, x1, y1)

for r in range(0, MAZE\_H \* UNIT, UNIT):

x0, y0, x1, y1 = 0, r, MAZE\_W \* UNIT, r

self.canvas.create\_line(x0, y0, x1, y1)

# create origin

origin = np.array([20, 20])

# hell

hell1\_center = origin + np.array([UNIT \* 2, UNIT])

self.hell1 = self.canvas.create\_rectangle(

hell1\_center[0] - 15, hell1\_center[1] - 15,

hell1\_center[0] + 15, hell1\_center[1] + 15,

fill='black')

# hell

# hell2\_center = origin + np.array([UNIT, UNIT \* 2])

# self.hell2 = self.canvas.create\_rectangle(

# hell2\_center[0] - 15, hell2\_center[1] - 15,

# hell2\_center[0] + 15, hell2\_center[1] + 15,

# fill='black')

# create oval

oval\_center = origin + UNIT \* 2

self.oval = self.canvas.create\_oval(

oval\_center[0] - 15, oval\_center[1] - 15,

oval\_center[0] + 15, oval\_center[1] + 15,

fill='yellow')

# create red rect

self.rect = self.canvas.create\_rectangle(

origin[0] - 15, origin[1] - 15,

origin[0] + 15, origin[1] + 15,

fill='red')

# pack all

self.canvas.pack()

def reset(self):

self.update()

time.sleep(0.1)

self.canvas.delete(self.rect)

origin = np.array([20, 20])

self.rect = self.canvas.create\_rectangle(

origin[0] - 15, origin[1] - 15,

origin[0] + 15, origin[1] + 15,

fill='red')

# return observation

return (np.array(self.canvas.coords(self.rect)[:2]) - np.array(self.canvas.coords(self.oval)[:2]))/(MAZE\_H\*UNIT)

def step(self, action):

s = self.canvas.coords(self.rect)

base\_action = np.array([0, 0])

if action == 0: # up

if s[1] > UNIT:

base\_action[1] -= UNIT

elif action == 1: # down

if s[1] < (MAZE\_H - 1) \* UNIT:

base\_action[1] += UNIT

elif action == 2: # right

if s[0] < (MAZE\_W - 1) \* UNIT:

base\_action[0] += UNIT

elif action == 3: # left

if s[0] > UNIT:

base\_action[0] -= UNIT

self.canvas.move(self.rect, base\_action[0], base\_action[1]) # move agent

next\_coords = self.canvas.coords(self.rect) # next state

# reward function

if next\_coords == self.canvas.coords(self.oval):

reward = 1

done = True

elif next\_coords in [self.canvas.coords(self.hell1)]:

reward = -1

done = True

else:

reward = 0

done = False

s\_ = (np.array(next\_coords[:2]) - np.array(self.canvas.coords(self.oval)[:2]))/(MAZE\_H\*UNIT)

return s\_, reward, done

def render(self):

# time.sleep(0.01)

self.update()

def run\_maze():

step = 0

for episode in range(300):

# initial observation

observation = env.reset()

while True:

# fresh env

env.render()

# RL choose action based on observation

action = RL.choose\_action(observation)

# RL take action and get next observation and reward

observation\_, reward, done = env.step(action)

RL.store\_transition(observation, action, reward, observation\_)

if (step > 200) and (step % 5 == 0):

RL.learn()

# swap observation

observation = observation\_

# break while loop when end of this episode

if done:

break

step += 1

# end of game

print('game over')

env.destroy()

if \_\_name\_\_ == "\_\_main\_\_":

# maze game

env = Maze()

RL = DeepQNetwork(env.n\_actions, env.n\_features,

learning\_rate=0.01,

reward\_decay=0.9,

e\_greedy=0.9,

replace\_target\_iter=200,

memory\_size=2000,

# output\_graph=True

)

env.after(100, run\_maze)

env.mainloop()

RL.plot\_cost()

