МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

им. Н.Э. Баумана

Факультет «Информатика и системы управления»

Кафедра «Систем обработки информации и управления»

ОТЧЕТ

**Лабораторная работа №\_\_6\_\_**

по дисциплине«Методы машинного обучения»

Тема: «Обучение на основе глубоких Q-сетей»

ИСПОЛНИТЕЛЬ: Лу Сяои

ФИО

группа ИУ5И-22М \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись

"10" Июнь 2023 г.

ПРЕПОДАВАТЕЛЬ: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ФИО

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись

"\_\_"\_\_\_\_\_\_\_\_\_2023 г.

Москва - 2023

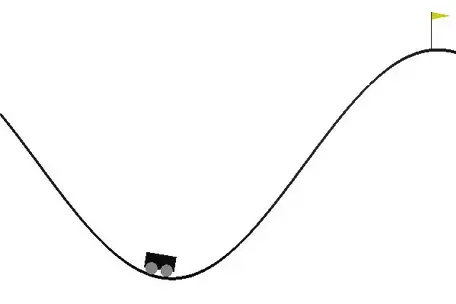
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **описание задания**

На основе рассмотренного на лекции примера реализуйте алгоритм **Policy Iteration** для любой среды обучения с подкреплением (кроме рассмотренной на лекции среды Toy Text / Frozen Lake) из библиотеки Gym (или аналогичной библиотеки).

## **текст программы и экранные формы с примерами выполнения программы.**

Я выбрала среду **MountainCar-v0**.



## Observation

Type: Box(2)

| **Num** | **Observation** | **Min** | **Max** |
| --- | --- | --- | --- |
| 0 | Car Position | -1.2 | 0.6 |
| 1 | Car Velocity | -0.07 | 0.07 |

Note that velocity has been constrained to facilitate exploration, but this constraint might be relaxed in a more challenging version.

## Actions

Type: Box(1)

| **Num** | **Action** |
| --- | --- |
| 0 | Push car to the left (negative value) or to the right (positive value) |

## Reward

Reward is 100 for reaching the target of the hill on the right hand side, minus the squared sum of actions from start to goal.

This reward function raises an exploration challenge, because if the agent does not reach the target soon enough, it will figure out that it is better not to move, and won't find the target anymore.

import gym

import random

import numpy as np

import copy

import matplotlib.pyplot as plt

from collections import namedtuple, deque

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

import warnings

warnings.filterwarnings("ignore")

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

env = gym.make('MountainCar-v0')

n\_actions = env.action\_space.n

n\_states = env.observation\_space.shape[0]

Transition = namedtuple('Transition', ('state', 'action', 'next\_state', 'reward', 'done'))

#### Буфер для experience replay

class ReplayMemory(object):

    def \_\_init\_\_(self, capacity):

        self.capacity = capacity

        self.memory = []

        self.position = 0

    def push(self, \*args):

        """Saves a transition."""

        if len(self.memory) < self.capacity:

            self.memory.append(None)

        self.memory[self.position] = Transition(\*args)

        self.position = (self.position + 1) % self.capacity

    def sample(self, batch\_size):

        return random.sample(self.memory, batch\_size)

    def \_\_len\_\_(self):

        return len(self.memory)

#### Класс для DQN

class DQN:

    def \_\_init\_\_(self, layers, lr=0.0005, optim\_method=optim.Adam):

        self.layers = layers

        self.lr = lr

        self.loss = F.mse\_loss

        self.optim\_method = optim\_method

        self.TargetNetwork = None

        self.EstimateNetwork = None

        self.optimizer = None

        self.build\_model()

    def build\_model(self):

        def init\_weights(layer):

            if type(layer) == nn.Linear:

                nn.init.xavier\_normal\_(layer.weight)

        self.EstimateNetwork = nn.Sequential(\*self.layers)

        self.EstimateNetwork.apply(init\_weights)

        layers\_for\_target = copy.deepcopy(self.layers)

        self.TargetNetwork = nn.Sequential(\*layers\_for\_target)

        self.TargetNetwork.load\_state\_dict(self.EstimateNetwork.state\_dict())

        self.optimizer = self.optim\_method(self.EstimateNetwork.parameters(), lr=self.lr)

    def Q\_target(self, inp):

        return self.TargetNetwork(inp)

    def Q\_estimate(self, inp):

        return self.EstimateNetwork(inp)

    def update\_target(self):

        self.TargetNetwork.load\_state\_dict(self.EstimateNetwork.state\_dict())

    def update\_parameters(self, estimated, targets):

        loss = self.loss(estimated, targets.unsqueeze(1))

        self.optimizer.zero\_grad()

        loss.backward()

        for param in self.EstimateNetwork.parameters():

            param.grad.data.clamp\_(-1, 1)

        self.optimizer.step()

    def save(self, name):

        torch.save(self.EstimateNetwork, name)

        print('------ Model saved ------')

## Класс для агента

class Agent:

    def \_\_init\_\_(self, env, Model, n\_actions, goal, min\_score, \

                 eps\_start=1, eps\_end=0.001, eps\_decay=0.9, gamma=0.99, \

                 batch\_size=64, memory\_size=100000, max\_episode=2000, upd\_rate=1):

        self.env = env

        self.n\_actions = n\_actions # number of possible actions

        self.goal = goal # the score to reach during learning

        self.min\_score = min\_score # min score to complete the episode

        self.eps\_start = eps\_start

        self.eps = eps\_start

        self.eps\_end = eps\_end

        self.eps\_decay = eps\_decay

        self.gamma = gamma

        self.batch\_size = batch\_size

        self.target\_update\_rate = upd\_rate # how often we update our target network

        self.Model = Model # DQN instance

        self.max\_episode = max\_episode # how long we train our agent

        self.memory = ReplayMemory(memory\_size) # Replay buffer

    def act(self, state, eps): # epsilon greedy policy

        if random.random() < eps:

            return torch.tensor([[random.randrange(self.n\_actions)]], device=device, dtype=torch.long)

        else:

            with torch.no\_grad():

                result = self.Model.Q\_estimate(state).max(1)[1]

                return result.view(1, 1)

    def optimize(self): # experience replay

        if len(self.memory) < self.batch\_size:

            return

        transitions = self.memory.sample(self.batch\_size)

        batch = Transition(\*zip(\*transitions))

        next\_state\_batch = torch.cat(batch.next\_state)

        state\_batch = torch.cat(batch.state)

        action\_batch = torch.cat(batch.action)

        reward\_batch = torch.cat(batch.reward)

        done\_batch = torch.cat(batch.done)

        estimate\_value = self.Model.Q\_estimate(state\_batch).gather(1, action\_batch)

        Q\_value\_next = torch.zeros(self.batch\_size, device=device)

        with torch.no\_grad():

            Q\_value\_next[~done\_batch] = self.Model.Q\_target(next\_state\_batch).max(1)[0].detach()[~done\_batch]

        target\_value = (Q\_value\_next \* self.gamma) + reward\_batch

        self.Model.update\_parameters(estimate\_value, target\_value)

    def train(self): # learning procedure

        all\_scores = []

        successful\_sequences = 0

        for ep in range(1, self.max\_episode + 1):

            state = self.env.reset()

            state = torch.tensor(state).to(device).float().unsqueeze(0)

            done = False

            episode\_reward = 0

            while not done:

                action = self.act(state, self.eps)

                action = torch.tensor(action).to(device)

                next\_state, reward, done, info = self.env.step(action.item())

                episode\_reward += reward

                modified\_reward = reward + 300 \* (self.gamma \* abs(next\_state[1]) - abs(state[0][1]))

                next\_state = torch.tensor(next\_state).to(device).float().unsqueeze(0)

                modified\_reward = torch.tensor(modified\_reward).to(device).float().unsqueeze(0)

                done = torch.tensor(done).to(device).unsqueeze(0)

                self.memory.push(state, action, next\_state, modified\_reward, done)

                state = next\_state

                self.optimize() # experience replay

            if ep % self.target\_update\_rate == 0:

                self.Model.update\_target()

            self.eps = max(self.eps\_end, self.eps \* self.eps\_decay)

            all\_scores.append(episode\_reward)

            if ep % 100 == 0:

                print('episode', ep, ':', np.mean(all\_scores[:-100:-1]), 'average score')

            if np.mean(all\_scores[:-100:-1]) >= self.goal:

                successful\_sequences += 1

                if successful\_sequences == 5:

                    print('success at episode', ep)

                    return all\_scores

            else:

                successful\_sequences = 0

        return all\_scores

    def test(self, episodes=50, render=False): #test trained agent

        state = self.env.reset()

        state = torch.tensor(state).to(device).float().unsqueeze(0)

        ep\_count = 0

        current\_episode\_reward = 0

        scores = []

        while ep\_count < episodes:

            if render:

                env.render()

            action = self.act(state, 0)

            state, reward, done, \_ = self.env.step(action.item())

            state = torch.tensor(state).to(device).float().unsqueeze(0)

            current\_episode\_reward += reward

            if done:

                ep\_count += 1

                scores.append(current\_episode\_reward)

                current\_episode\_reward = 0

                state = self.env.reset()

                state = torch.tensor(state).to(device).float().unsqueeze(0)

        print('average score:', sum(scores) / len(scores))

        print('max reward:', max(scores))

        print('-----')

        print()

    def save(self, name='agent.pkl'): # save policy network

        self.Model.save(name)

#### Инициализация агента и обучение

Используется трехслойная нейронная сеть с прямой связью, состоящая из двух полностью связанных слоев и одного выходного слоя.

Первый слой - входной слой, содержащий n\_состояний входных узлов, второй и третий слои - скрытые слои, содержащие по 256 нейронов, и последний слой - выходной слой, содержащий n\_действий выходных узлов.

layers = (

        nn.Linear(n\_states, 256),

        nn.ReLU(),

        nn.Linear(256, 256),

        nn.ReLU(),

        nn.Linear(256, n\_actions),

)

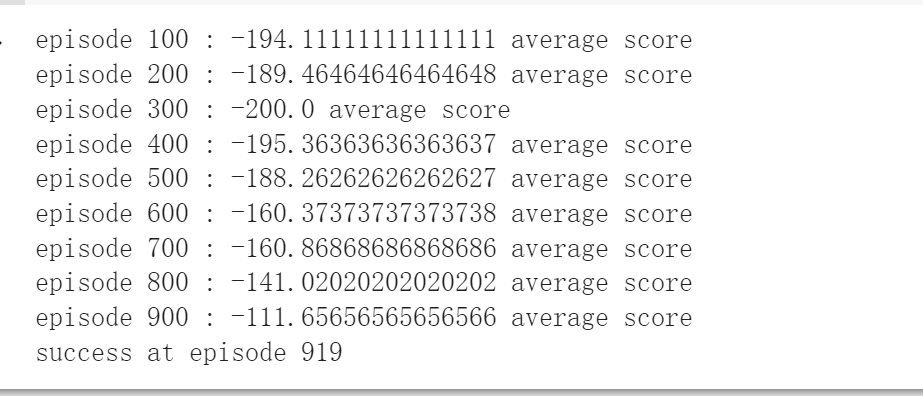
Model = DQN(layers, lr=0.0001, optim\_method=optim.Adam)

MountainCarAgent = Agent(env, Model, n\_actions, goal=-110, min\_score=-200, \

                         eps\_start=1, eps\_end=0.001, eps\_decay=0.9, gamma=0.99, \

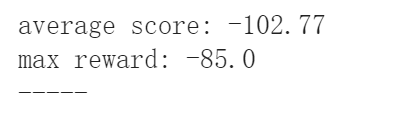
                         batch\_size=64, memory\_size=100000, max\_episode=2000)

scores = MountainCarAgent.train()



##### Протестируем агента и выведем средний скор за 100 эпизодов

MountainCarAgent.test(episodes=100)



##### История обучения

episodes = range(len(scores))

plt.plot(episodes, scores)

plt.xlabel('episodes')

plt.ylabel('scores')

plt.show()

