****Московский государственный технический университет им. Н.Э. Баумана  
Кафедра «Системы обработки информации и управления»

Лабораторная работа №6  
по дисциплине  
«Методы машинного обучения»  
на тему

# «Обучение на основе глубоких Q-сетей»

Выполнил:  
студент группы ИУ5-23М  
Чжэн Сяохуэй

Москва — 2024 г.

**1. Цель лабораторной работы**

ознакомление с базовыми методами обучения с подкреплением на основе глубоких Q-сетей.

**2. Задание**

* 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.**

**3. текст программы**

Моя среда реализована на основе архитектуры сверточной нейронной сети.

import gym

import math

import random

import matplotlib

import matplotlib.pyplot as plt

from collections import namedtuple, deque

from itertools import count

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

# 名称环境

CONST\_ENV\_NAME = 'CartPole-v1'

# 使用GPU

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

# CONST\_DEVICE = torch.device("cpu")

# ReplayMemory元素定义

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

# 实现ReplayMemory技术

class ReplayMemory(object):

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

self.memory = deque([], maxlen=capacity)

def push(self, \*args):

'''保存数据到ReplayMemory'''

self.memory.append(Transition(\*args))

def sample(self, batch\_size):

'''随机选取batch\_size大小的样本'''

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

def \_\_len\_\_(self):

return len(self.memory)

# 基于卷积神经网络的DQN模型

class DQN\_Model(nn.Module):

def \_\_init\_\_(self, h, w, outputs):

super(DQN\_Model, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 32, kernel\_size=8, stride=4)

self.conv2 = nn.Conv2d(32, 64, kernel\_size=4, stride=2)

self.conv3 = nn.Conv2d(64, 64, kernel\_size=3, stride=1)

self.fc1 = nn.Linear(self.\_calculate\_conv\_output(h, w), 512)

self.fc2 = nn.Linear(512, outputs)

def \_calculate\_conv\_output(self, h, w):

# 计算卷积层输出大小的函数

def conv2d\_size\_out(size, kernel\_size=3, stride=1):

return (size - (kernel\_size - 1) - 1) // stride + 1

convw = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(w, 8, 4), 4, 2), 3, 1)

convh = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(h, 8, 4), 4, 2), 3, 1)

return convw \* convh \* 64

def forward(self, x):

x = F.relu(self.conv1(x))

x = F.relu(self.conv2(x))

x = F.relu(self.conv3(x))

x = F.relu(self.fc1(x.view(x.size(0), -1)))

return self.fc2(x)

class DQN\_Agent:

def \_\_init\_\_(self, env, BATCH\_SIZE=128, GAMMA=0.99, EPS\_START=0.9, EPS\_END=0.05, EPS\_DECAY=1000, TAU=0.005, LR=1e-4):

self.env = env

self.n\_actions = env.action\_space.n

state, \_ = self.env.reset()

init\_screen = self.get\_screen()

\_, \_, screen\_height, screen\_width = init\_screen.shape

self.policy\_net = DQN\_Model(screen\_height, screen\_width, self.n\_actions).to(CONST\_DEVICE)

self.target\_net = DQN\_Model(screen\_height, screen\_width, self.n\_actions).to(CONST\_DEVICE)

self.target\_net.load\_state\_dict(self.policy\_net.state\_dict())

self.optimizer = optim.AdamW(self.policy\_net.parameters(), lr=LR, amsgrad=True)

self.memory = ReplayMemory(10000)

self.steps\_done = 0

self.episode\_durations = []

self.BATCH\_SIZE = BATCH\_SIZE

self.GAMMA = GAMMA

self.EPS\_START = EPS\_START

self.EPS\_END = EPS\_END

self.EPS\_DECAY = EPS\_DECAY

self.TAU = TAU

def get\_screen(self):

# 从环境中获取当前状态

screen = self.env.render()

screen = torch.tensor(screen, dtype=torch.float32).permute(2, 0, 1) # 调整通道顺序

return screen.unsqueeze(0).to(CONST\_DEVICE)

def select\_action(self, state):

sample = random.random()

eps\_threshold = self.EPS\_END + (self.EPS\_START - self.EPS\_END) \* math.exp(-1. \* self.steps\_done / self.EPS\_DECAY)

self.steps\_done += 1

if sample > eps\_threshold:

with torch.no\_grad():

return self.policy\_net(state).max(1)[1].view(1, 1)

else:

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

def plot\_durations(self, show\_result=False):

plt.figure(1)

durations\_t = torch.tensor(self.episode\_durations, dtype=torch.float)

if show\_result:

plt.title('resoult')

else:

plt.clf()

plt.title('trining...')

plt.xlabel('Episode')

plt.ylabel('Duration')

plt.plot(durations\_t.numpy())

plt.pause(0.001)

def optimize\_model(self):

if len(self.memory) < self.BATCH\_SIZE:

return

transitions = self.memory.sample(self.BATCH\_SIZE)

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

non\_final\_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next\_state)), device=CONST\_DEVICE, dtype=torch.bool)

non\_final\_next\_states = torch.cat([s for s in batch.next\_state if s is not None])

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

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

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

state\_action\_values = self.policy\_net(state\_batch).gather(1, action\_batch)

next\_state\_values = torch.zeros(self.BATCH\_SIZE, device=CONST\_DEVICE)

with torch.no\_grad():

next\_state\_values[non\_final\_mask] = self.target\_net(non\_final\_next\_states).max(1)[0]

expected\_state\_action\_values = (next\_state\_values \* self.GAMMA) + reward\_batch

criterion = nn.SmoothL1Loss()

loss = criterion(state\_action\_values, expected\_state\_action\_values.unsqueeze(1))

self.optimizer.zero\_grad()

loss.backward()

torch.nn.utils.clip\_grad\_value\_(self.policy\_net.parameters(), 100)

self.optimizer.step()

def play\_agent(self):

env2 = gym.make(CONST\_ENV\_NAME, render\_mode='human')

state = env2.reset()[0]

state = self.get\_screen()

done = False

res = []

while not done:

action = self.select\_action(state)

action = action.item()

observation, reward, terminated, truncated, \_ = env2.step(action)

env2.render()

res.append((action, reward))

if terminated:

next\_state = None

else:

next\_state = self.get\_screen()

state = next\_state

if terminated or truncated:

done = True

print('Episode data: ', res)

def learn(self):

num\_episodes = 200 if torch.cuda.is\_available() else 50

for i\_episode in range(num\_episodes):

self.env.reset()

state = self.get\_screen()

for t in count():

action = self.select\_action(state)

observation, reward, terminated, truncated, \_ = self.env.step(action.item())

reward = torch.tensor([reward], device=CONST\_DEVICE)

done = terminated or truncated

next\_state = None if terminated else self.get\_screen()

self.memory.push(state, action, next\_state, reward)

state = next\_state

self.optimize\_model()

target\_net\_state\_dict = self.target\_net.state\_dict()

policy\_net\_state\_dict = self.policy\_net.state\_dict()

for key in policy\_net\_state\_dict:

target\_net\_state\_dict[key] = policy\_net\_state\_dict[key] \* self.TAU + target\_net\_state\_dict[key] \* (1 - self.TAU)

self.target\_net.load\_state\_dict(target\_net\_state\_dict)

if done:

self.episode\_durations.append(t + 1)

self.plot\_durations()

break

def main():

env = gym.make(CONST\_ENV\_NAME, render\_mode='rgb\_array')

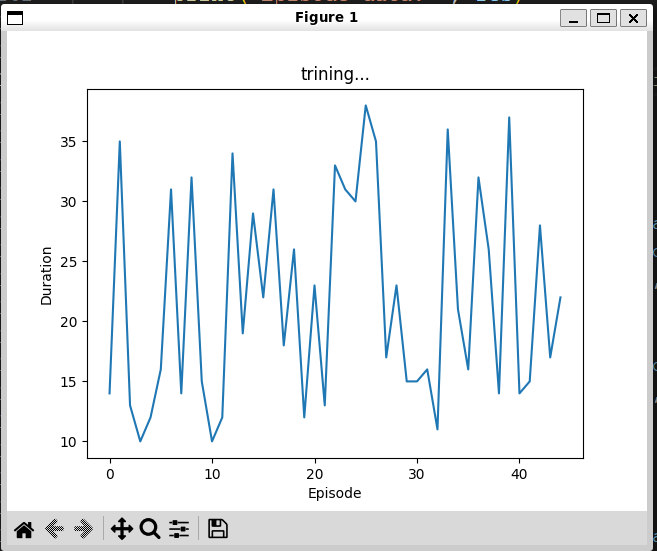
agent = DQN\_Agent(env)

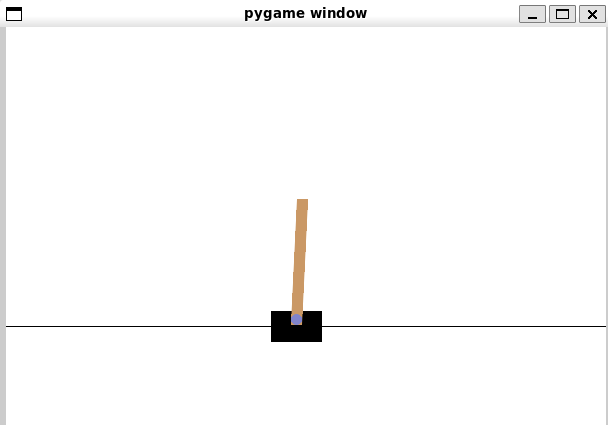
agent.learn()

agent.play\_agent()

if \_\_name\_\_ == '\_\_main\_\_':

main()

****4. экранные формы с примерами выполнения программы.

****

**Список литературы**

[1] Гапанюк Ю. Е. COURSE\_MMO\_SPRING\_2024// GitHub. –– 2024. –– Режим доступа: https://github.com/ugapanyuk/courses\_current/wiki/COURSE\_MMO\_SPRING\_2024

[2] <https://www.kaggle.com/datasets>