****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 ИУ5И-21М

Фань Лицзе

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.**

1. **Текст программы**

import gym

import random

import torch

import torch.nn as nn

import torch.optim as optim

from collections import namedtuple, deque

from itertools import count

import math

import matplotlib.pyplot as plt

# 设置环境

env = gym.make('CartPole-v1')

# GPU或CPU设备

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

# 命名元组存储转移

Transition = namedtuple('Transition',

('state', 'action', 'next\_state', 'reward'))

# 经验回放

class ReplayMemory(object):

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

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

def push(self, \*args):

"""Save a transition"""

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

def sample(self, batch\_size):

"""Randomly sample a batch of transitions"""

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

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

return len(self.memory)

# DQN网络模型

class DQN(nn.Module):

def \_\_init\_\_(self, inputs, outputs):

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

self.fc1 = nn.Linear(inputs, 128)

self.fc2 = nn.Linear(128, 64)

self.fc3 = nn.Linear(64, outputs)

def forward(self, x):

x = torch.relu(self.fc1(x))

x = torch.relu(self.fc2(x))

x = self.fc3(x)

return x

# DQN Agent

class DQNAgent:

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

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

self.policy\_net = DQN(env.observation\_space.shape[0], self.n\_actions).to(device)

self.target\_net = DQN(env.observation\_space.shape[0], self.n\_actions).to(device)

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

self.target\_net.eval()

self.optimizer = optim.Adam(self.policy\_net.parameters())

self.memory = ReplayMemory(10000)

self.steps\_done = 0

self.EPS\_START = 0.9

self.EPS\_END = 0.05

self.EPS\_DECAY = 200

self.BATCH\_SIZE = 128

self.GAMMA = 0.999

def select\_action(self, state):

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 random.random() > 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=device, dtype=torch.long)

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=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=device)

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

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

loss = nn.SmoothL1Loss()(state\_action\_values, expected\_state\_action\_values.unsqueeze(1))

self.optimizer.zero\_grad()

loss.backward()

self.optimizer.step()

return loss.item()

# 主训练循环

def main():

num\_episodes = 50

agent = DQNAgent()

rewards = []

losses = []

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

state = env.reset()

state = torch.tensor([state], device=device, dtype=torch.float32)

total\_reward = 0

for t in count():

action = agent.select\_action(state)

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

total\_reward += reward

reward = torch.tensor([reward], device=device)

if not done:

next\_state = torch.tensor([next\_state], device=device, dtype=torch.float32)

else:

next\_state = None

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

state = next\_state

loss = agent.optimize\_model()

if loss is not None:

losses.append(loss)

if done:

break

rewards.append(total\_reward)

print(f"Episode {i\_episode+1}: Total reward = {total\_reward}")

plt.figure(figsize=(10,5))

plt.plot(rewards)

plt.title('Reward per Episode')

plt.xlabel('Episode')

plt.ylabel('Total Reward')

plt.show()

plt.figure(figsize=(10,5))

plt.plot(losses)

plt.title('Loss per Step')

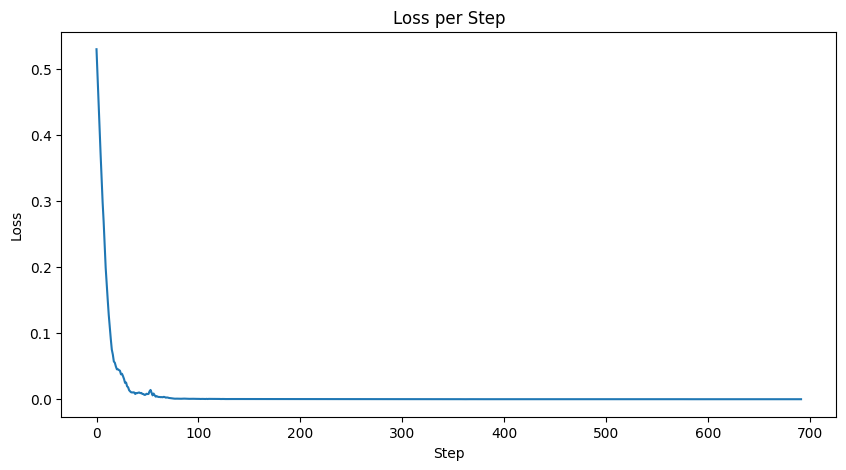
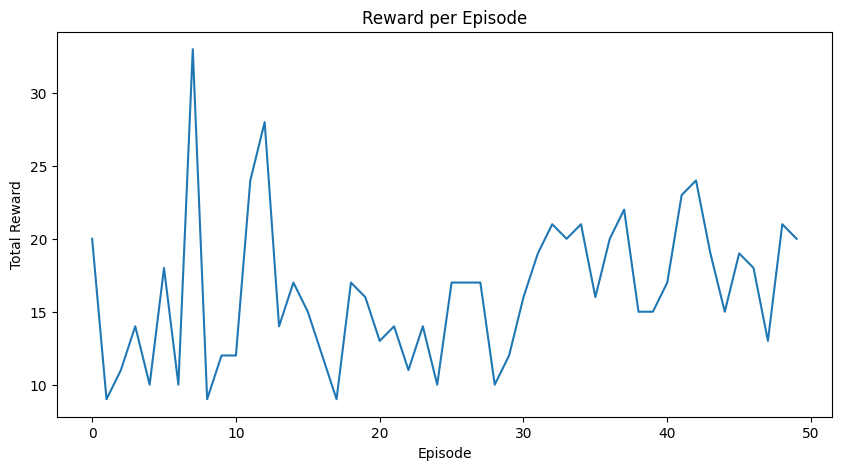
plt.xlabel('Step')

plt.ylabel('Loss')

plt.show()

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

main()



import gym

# 创建并初始化环境

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

env.reset()

# 执行一个动作 (0) 并获取反馈

action = 0 # 'NOOP'

observation, reward, done, info = env.step(action)

# 显示游戏屏幕的观察结果

import matplotlib.pyplot as plt

plt.imshow(env.render(mode='rgb\_array'))

plt.title("Breakout Game Screen")

plt.show()

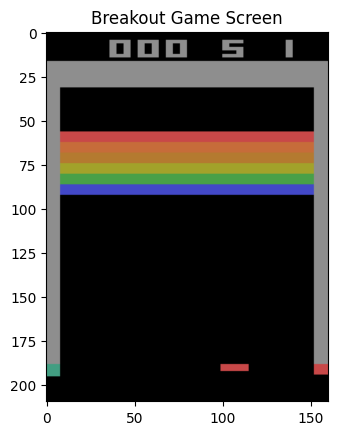
# 游戏结束时重置环境

if done:

env.reset()

# 关闭环境

env.close()



import gym

import numpy as np

import random

import torch

import torch.nn as nn

import torch.optim as optim

from collections import namedtuple, deque

from itertools import count

from torchvision import transforms

import math

import matplotlib.pyplot as plt

# 设置环境

env = gym.make('Breakout-v0').unwrapped

# GPU或CPU设备

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

# 命名元组存储转移

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

# 经验回放

class ReplayMemory(object):

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

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

def push(self, \*args):

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

def sample(self, batch\_size):

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

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

return len(self.memory)

# 预处理管道

transform = transforms.Compose([

transforms.ToPILImage(),

transforms.Grayscale(),

transforms.Resize((40, 40)),

transforms.ToTensor()

])

def get\_screen():

screen = env.render(mode='rgb\_array').transpose((2, 0, 1))

screen = np.ascontiguousarray(screen, dtype=np.float32) / 255

screen = torch.from\_numpy(screen)

return transform(screen).unsqueeze(0).to(device)

# DQN网络模型

class DQN(nn.Module):

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

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

self.conv1 = nn.Conv2d(1, 16, kernel\_size=5, stride=2)

self.bn1 = nn.BatchNorm2d(16)

self.conv2 = nn.Conv2d(16, 32, kernel\_size=5, stride=2)

self.bn2 = nn.BatchNorm2d(32)

self.conv3 = nn.Conv2d(32, 32, kernel\_size=5, stride=2)

self.bn3 = nn.BatchNorm2d(32)

def conv2d\_size\_out(size, kernel\_size=5, stride=2):

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

convw = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(w)))

convh = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(h)))

self.head = nn.Linear(convh \* convw \* 32, outputs)

def forward(self, x):

x = torch.relu(self.bn1(self.conv1(x)))

x = torch.relu(self.bn2(self.conv2(x)))

x = torch.relu(self.bn3(self.conv3(x)))

return self.head(x.view(x.size(0), -1))

# DQN Agent

class DQNAgent:

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

init\_screen = get\_screen()

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

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

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

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

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

self.target\_net.eval()

self.optimizer = optim.Adam(self.policy\_net.parameters())

self.memory = ReplayMemory(10000)

self.steps\_done = 0

self.EPS\_START = 0.9

self.EPS\_END = 0.05

self.EPS\_DECAY = 200

self.BATCH\_SIZE = 128

self.GAMMA = 0.99

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=device, dtype=torch.long)

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=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=device)

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

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

loss = nn.SmoothL1Loss()(state\_action\_values, expected\_state\_action\_values.unsqueeze(1))

self.optimizer.zero\_grad()

loss.backward()

self.optimizer.step()

# 主训练循环

def main():

num\_episodes = 50

episode\_rewards = [] # 用于记录每个episode的奖励

agent = DQNAgent()

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

env.reset()

total\_reward = 0

last\_screen = get\_screen()

current\_screen = get\_screen()

state = current\_screen - last\_screen

for t in count():

action = agent.select\_action(state)

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

reward = torch.tensor([reward], device=device)

total\_reward += reward.item()

last\_screen = current\_screen

current\_screen = get\_screen()

if not done:

next\_state = current\_screen - last\_screen

else:

next\_state = None

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

state = next\_state

agent.optimize\_model()

if done:

episode\_rewards.append(total\_reward)

break

plt.figure(figsize=(10,5))

plt.plot(episode\_rewards)

plt.title('Episode Rewards Over Time')

plt.xlabel('Episode')

plt.ylabel('Total Reward')

plt.show()

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

main()

