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 ИУ5 «Системы обработки информации и управления»

Отчет по лабораторной работе №6

по дисциплине «Методы машинного обучения»

по теме «Обучение на основе DQN»

Выполнил:
студент группы № ИУ5-24М
Винников С.С.
подпись, дата

Проверил:

подпись, дата

2024 г.

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.**

Текст программы

SetUp.py

```
from collections import namedtuple
import torch


# Название среды
CONST_ENV_NAME = 'Acrobot-v1'

# Использование GPU
CONST_DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Элемент ReplayMemory в форме именованного кортежа
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

ReplayMemory.py

```
import random
from collections import deque

from SetUp import Transition

# Реализация техники Replay Memory
class ReplayMemory(object):
 def   init  (self, capacity):
 self.memory = deque([], maxlen=capacity)

 def push(self, *args):
 '''
 Сохранение данных в ReplayMemory
 '''

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

 def sample(self, batch_size):
 '''
 Выборка случайных элементов размера batch_size
 '''

 return random.sample(self.memory, batch_size)

 def   len  (self):
 return len(self.memory)
```

DQN_Model.py

```
import torch.nn as nn
import torch.nn.functional as F

class DQN Model(nn.Module):
 def __init__(self, n_observations, n_actions):
 '''
 Инициализация топологии нейронной сети
 '''

 super(DQN_Model, self).__init__()
```

```
 self.layer1 = nn.Linear(n observations, 128)
 self.layer2 = nn.Linear(128, 64)
 self.layer3 = nn.Linear(64, n_actions)

 def forward(self, x):
 '''
 Прямой проход
 Вызывается для одного элемента, чтобы определить следующее действие  Или
для batch во время процедуры оптимизации
 '''

 x = F.relu(self.layer1(x))
 x = F.relu(self.layer2(x))
 return self.layer3(x)
```

DQN_Agent.py

```
import gymnasium as gym
import math
import random
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from DQN Model import DQN Model
from ReplayMemory import ReplayMemory
from SetUp import CONST_DEVICE, CONST_ENV_NAME, Transition


class DQN Agent:
 def  _init__(
 self,
 env,
 BATCH SIZE = 128,
 GAMMA = 0.99,
 EPS START = 0.1,
 EPS END = 0.5,
 EPS DECAY = 1000,
 TAU = 0.005,
 LR = 0.0001,
 ):
 # Среда
 self.env = env
 # Размерности Q-модели
 self.n actions = env.action_space.n
 state,   = self.env.reset()
 self.n observations = len(state)
 # Коэффициенты
 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
 self.LR = LR

 # Модели
 # Основная модель
 self.policy_net = DQN_Model(self.n_observations, self.n_actions).to(CONST_DEVICE)

 # Вспомогательная модель, используется для стабилизации алгоритма  #
Обновление контролируется гиперпараметром TAU
 # Используется подход Double DQN
 self.target net = DQN Model(self.n observations, self.n actions).to(CONST_DEVICE)
self.target_net.load_state_dict(self.policy_net.state_dict())

 # Оптимизатор
 self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=self.LR, amsgrad=True)

 # Replay Memory
 self.memory = ReplayMemory(10000)

 # Количество шагов
 self.steps_done = 0

 # Длительность эпизодов
 self.episode_durations = []

 def select_action(self, state):
```

```
        '''
        Выбор действия
        '''

        sample = random.random()
        eps = 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:
        with torch.no_grad():
        # Если вероятность больше eps
        # то выбирается действие, соответствующее максимальному Q-значению  # t.max(1)
возвращает максимальное значение колонки для каждой строки  # [1] возвращает
индекс максимального элемента
        return self.policy_net(state).max(1)[1].view(1, 1)
        else:
        # Если вероятность меньше eps
        # то выбирается случайное действие
        return torch.tensor([[self.env.action_space.sample()]], 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('Результат')
        else:
        plt.clf()
        plt.title('Обучение')
        plt.xlabel('Эпизод')
        plt.ylabel('Количество шагов в эпизоде')
        plt.plot(durations_t.numpy())
        plt.pause(0.001)  # пауза

    def optimize_model(self):
        '''
        Оптимизация модели
        '''

        if len(self.memory) < self.BATCH_SIZE:
        return

        transitions = self.memory.sample(self.BATCH_SIZE)
        # Транспонирование batch'а
        # Конвертация batch-массива из Transition
        # в Transition batch-массивов.
        batch = Transition(*zip(*transitions))

        # Вычисление маски нефинальных состояний и конкатенация элементов batch'а  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)

        # Вычисление Q(s_t, a)
        state_action_values = self.policy_net(state_batch).gather(1, action_batch)

        # Вычисление V(s_{t+1}) для всех следующих состояний
        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]  #
Вычисление ожидаемых значений Q

        expected_state_action_values = (next_state_values * self.GAMMA) + reward_batch

        # Вычисление Huber loss
        criterion = nn.SmoothL1Loss()
        loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

        # Оптимизация модели
        self.optimizer.zero_grad()
        loss.backward()

        # gradient clipping
        torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 100)
self.optimizer.step()
```

```
 def play_agent(self):
 '''
 Проигрывание сессии для обученного агента
 '''

 env2 = gym.make(CONST_ENV_NAME, render_mode='human')
 state = env2.reset()[0]
 state = torch.tensor(state, dtype=torch.float32, device=CONST_DEVICE).unsqueeze(0)  res =
[]

 terminated = False
 truncated = False

 while not terminated and not truncated:
 action = self.select action(state)
 action = action.item()
 observation, reward, terminated, truncated, _ = env2.step(action)
env2.render()
 res.append((action, reward))

 state = torch.tensor(observation, dtype=torch.float32,
device=CONST_DEVICE).unsqueeze(0)

 print('done!')
 print('Данные об эпизоде: ', res)

 def train(self):
 '''
 Обучение агента
 '''

 if torch.cuda.is available():
 num episodes = 600
 else:
 num_episodes = 50

 for i episode in range(num_episodes):
 # Инициализация среды
 state, info = self.env.reset()
 state = torch.tensor(state, dtype=torch.float32, device=CONST_DEVICE).unsqueeze(0)

 terminated = False
 truncated = False

 iters = 0
 while not terminated and not truncated:
 action = self.select action(state)
 observation, reward, terminated, truncated, _ = self.env.step(action.item())  reward =
torch.tensor([reward], device=CONST_DEVICE)

 if terminated:
 next state = None
 else:
 next state = torch.tensor(observation, dtype=torch.float32,
device=CONST DEVICE).unsqueeze(0)
 # Сохранение данных в Replay Memory
 self.memory.push(state, action, next_state, reward)

 # Переход к следующему состоянию
 state = next_state

 # Выполнение одного шага оптимизации модели
 self.optimize_model()

 # Обновление весов target-сети
 # θ' ←τ θ + (1 − τ )θ'
 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)  iters +=
1

 self.episode durations.append(iters)
 self.plot_durations()
```

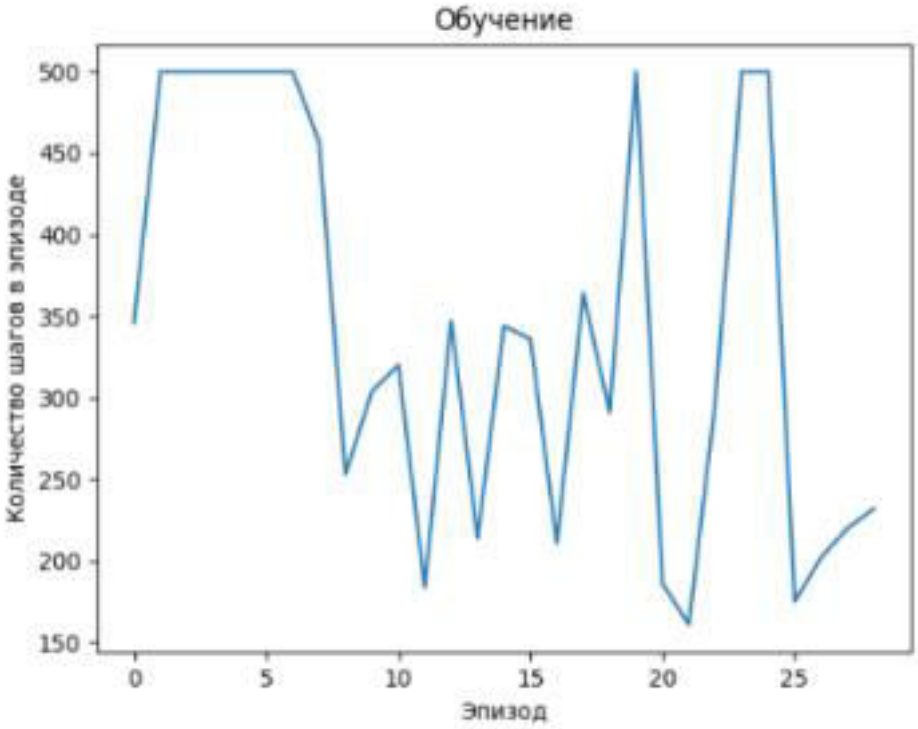
main.py

```
import gymnasium as gym
from DQN_Agent import DQN_Agent

import os
os.environ['SDL_VIDEODRIVER']='dummy'
import pygame
pygame.display.set mode((640,480))
from SetUp import CONST_ENV_NAME
def main():
 env = gym.make(CONST ENV_NAME)
 agent = DQN Agent(env)
 agent.train()
 agent.play_agent()

if   name__ == '__main__':
 main()
```

Экранные формы



done!

Данные об эпизоде: [(2, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (1, -1.0), (1, - 1.0), (2, -1.0), (2, -1.0), (1, -1.0), (0, -1.0), (2, -1.0), (2, -1.0), (2, -1.0), (0, -1.0), (0, -1.0), (1, - 1.0), (1, -1.0), (1, -1.0), (0, -1.0), (2, -1.0), (0, -1.0), (1, -1.0), (2, -1.0), (1, -1.0), (0, -1.0), (0, - 1.0), (1, -1.0), (1, -1.0), (1, -1.0), (2, -1.0), (1, -1.0), (1, -1.0), (2, -1.0), (2, -1.0), (0, -1.0), (1, - 1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (1, -1.0), (0, -1.0), (1, -1.0), (1, -1.0), (2, -1.0), (0, - 1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (1, -1.0), (2, -1.0), (2, -1.0), (2, - 1.0), (0, -1.0), (0, -1.0), (1, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (1, -1.0), (0, -1.0), (0, -1.0), (1, - 1.0), (2, -1.0), (0, -1.0), (1, -1.0), (2, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, - 1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, - 1.0), (0, -1.0), (0, -1.0), (1, -1.0), (2, -1.0), (2, -1.0), (2, -1.0), (0, -1.0), (1, -1.0), (2, - 1.0), (2, -1.0), (2, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (2, -1.0), (1, -1.0), (2, -1.0), (2, -1.0), (2, -1.0), (2, -1.0), (0, -1.0), (0, -1.0), (2, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (1, - 1.0), (0, -1.0), (0, -1.0), (1, -1.0), (2, -1.0), (2, -1.0), (2, -1.0), (0, -1.0), (1, -1.0), (0, -1.0), (2, - 1.0), (2, -1.0), (2, -1.0), (1, -1.0), (0, -1.0), (2, -1.0), (0, -1.0), (2, -

1.0), (1, -1.0), (0, -1.0), (1, - 1.0), (2, -1.0), (2, -1.0), (0, -1.0), (1, -1.0), (0, -1.0), (2, -1.0), (1, -1.0), (1, -1.0), (1, -1.0), (0, - 1.0), (2, -1.0), (0, -1.0), (2, -1.0), (1, -1.0), (2, -1.0), (0, -1.0), (1, -1.0), (2, -1.0), (2, -1.0), (1, - 1.0), (2, -1.0), (2, -1.0), (2, -1.0), (2, -1.0), (1, -1.0), (0, -1.0), (0, -1.0), (1, -1.0), (1, -1.0), (1, - 1.0), (2, -1.0), (2, -1.0), (2, -1.0), (2, -1.0), (1, -1.0), (2, -1.0), (1, -1.0), (2, -1.0), (0, -1.0), (1, - 1.0), (0, -1.0), (0, -1.0), (0, -1.0), (0, -1.0), (2, -1.0), (0, -1.0), (0, -1.0), (1, -1.0), (0, -1.0), (0, - 1.0), (0, -1.0), (0, 0.0)]