**Технологии машинного обучения. Лабораторная работа №6.**

**Студент: Уляшин В.В.**

**Группа: ИУ5-23М**

**Обучение на основе глубоких Q-сетей.**

**Цель лабораторной работы:** ознакомление с базовыми методами обучения с подкреплением на основе глубоких Q-сетей.

**Требования к отчету:**

Отчет по лабораторной работе должен содержать:

1. титульный лист;
2. текст программы;
3. экранные формы с примерами выполнения программы.

**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:**

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

*# Название среды*  
CONST\_ENV\_NAME = 'LunarLander-v2'  
*# 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'))

*# 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)

**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, 256)  
 self.layer3 = nn.Linear(256, 128)  
 self.layer4 = nn.Linear(128, n\_actions)  
  
 **def** forward(self, x):  
 *'''*  
 *Прямой проход*  
 *Вызывается для одного элемента, чтобы определить следующее действие*  
 *Или для batch'а во время процедуры оптимизации*  
 *'''*  
 x = F.relu(self.layer1(x))  
 x = F.relu(self.layer2(x))  
 x = F.relu(self.layer3(x))  
 **return** self.layer4(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  
 ):  
 *# 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 Qзначению*  
 *# 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*  
 *# (https://stackoverflow.com/a/19343/3343043)*  
 *# 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)  
 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 = torch.tensor(observation, dtype=torch.float32,  
 device=CONST\_DEVICE).unsqueeze(0)  
 state = next\_state  
 **if** terminated **or** truncated:  
 done = True  
 print('Данные об эпизоде: ', res)  
  
 **def** learn(self):  
 *'''*  
 *Обучение агента*  
 *'''*  
 **if** torch.cuda.is\_available():  
 num\_episodes = 600  
 **else**:  
 num\_episodes = 20  
 **for** i\_episode **in** range(num\_episodes):  
 *# Инициализация среды*  
 print(f'Эпизод: {i\_episode}')  
 state, info = self.env.reset()  
 state = torch.tensor(state, dtype=torch.float32,  
 device=CONST\_DEVICE).unsqueeze(0)  
 **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  
 **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)  
 **if** done:  
 self.episode\_durations.append(t + 1)  
 self.plot\_durations()  
 **break**

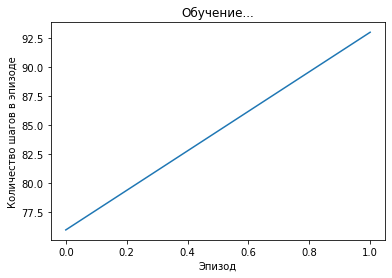
**def** main():  
 env = gym.make(CONST\_ENV\_NAME)  
 agent = DQN\_Agent(env)  
 agent.learn()  
 input()  
 **for** i **in** range(4):  
 agent.play\_agent()

main()

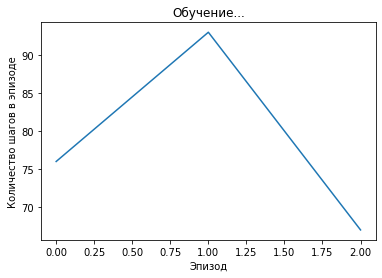
Эпизод: 0



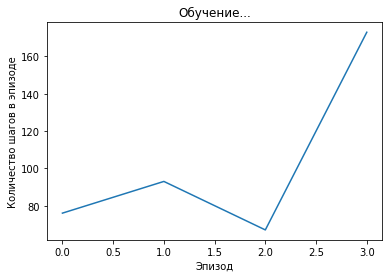
Эпизод: 1



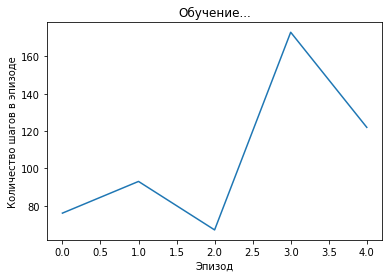
Эпизод: 2



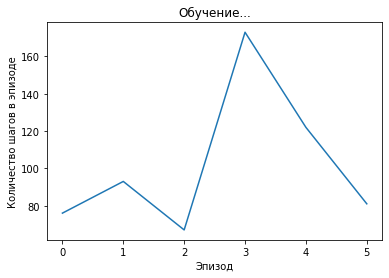
Эпизод: 3



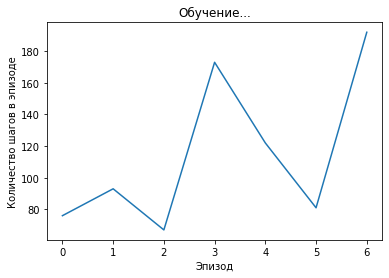
Эпизод: 4



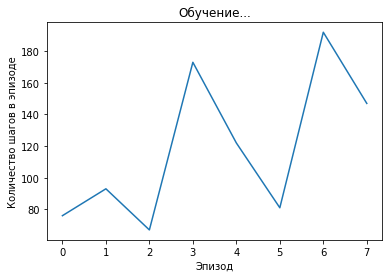
Эпизод: 5



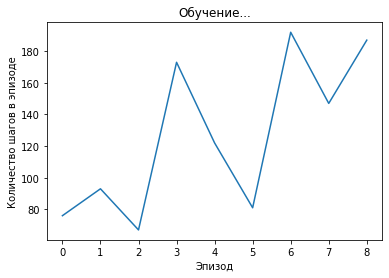
Эпизод: 6



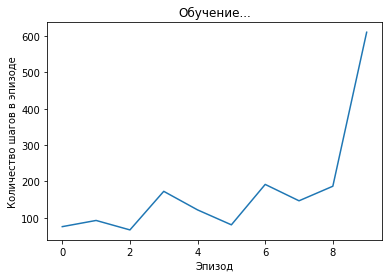
Эпизод: 7



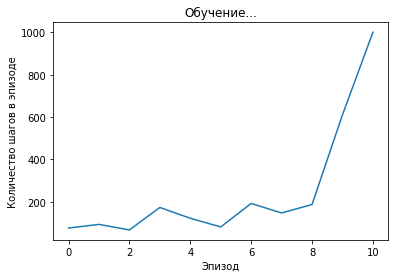
Эпизод: 8



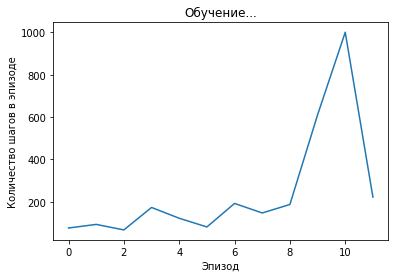
Эпизод: 9



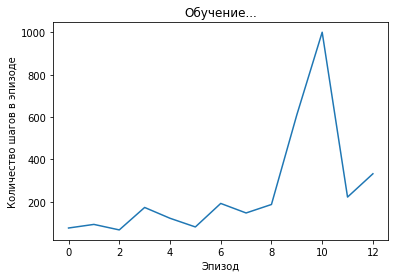
Эпизод: 10



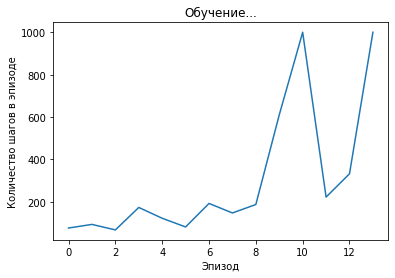
Эпизод: 11



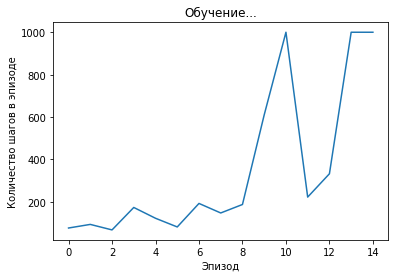
Эпизод: 12



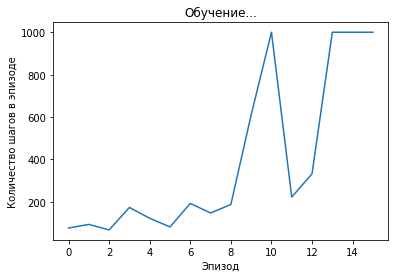
Эпизод: 13



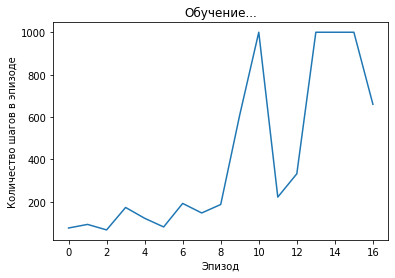
Эпизод: 14



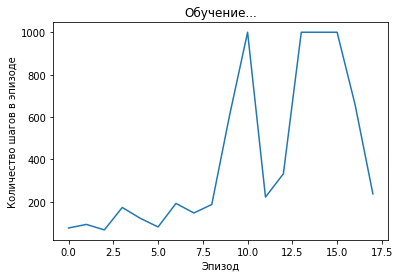
Эпизод: 15



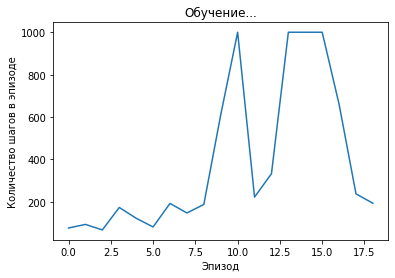
Эпизод: 16



Эпизод: 17



Эпизод: 18



Эпизод: 19

