МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЁТ

по лабораторной работе № 1 по дисциплине

«Обучение с подкреплением»

Тема: Реализация DQN для среды CartPole-v1

Студентка гр. 0310 Шкода М. А.

Преподаватель Глазунов С.А.

Санкт-Петербург
2025

Цель работы

Реализовать DQN для среды CartPole-v1 и изучить влияние различных параметров на обучение.

Задание

1. Измените архитектуру нейросети (например, добавьте слои).
2. Попробуйте разные значения gamma и epsilon_decay.
3. Проведите исследование как изначальное значение epsilon влияет на скорость обучения

Ход работы

Для выполнения задания были реализованы классы: ReplayBuffer – буфер, который хранит состояние, действие (передвинуть влево или вправо), награду, следующее состояние и ключ выполнения. Код класса представлен ниже. Весь код представлен в приложении А.

```
class ReplayBuffer:
    def __init__(self, capacity=1000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size): #нередко нужно взять не весь буфер
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (
            torch.tensor(np.array(states), dtype=torch.float32),
            torch.tensor(np.array(actions), dtype=torch.float32),
            torch.tensor(np.array(rewards), dtype=torch.float32),
            torch.tensor(np.array(next_states), dtype=torch.float32),
            torch.tensor(np.array(dones), dtype=torch.float32),
        )

    def __len__(self):
        return len(self.buffer)
```

Класс QNetwork, который содержит в себе три варианта архитектуры. Они отличаются количеством слоев и нейронов. Первая архитектура содержит 1 скрытый слой и 128 нейронов. Вторая архитектура уже содержит

4 скрытых слоя, а третья является самой простой и содержит всего 32 нейрона. Различие в архитектурах влияет на скорость обучения, как в плане времени выполнения, так и в сравнении по эпизодам, что можно будет увидеть далее. Код вариантов архитектур в классе QNetwork представлен ниже.

```
if layer_num == 0:
            self.net = nn.Sequential(
                nn.Linear(obs_size, 128),
                nn.ReLU(),
                nn.Linear(128, n_actions)
            )
        elif layer_num == 1:
          self.net = nn.Sequential(
            nn.Linear(obs_size, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.Sigmoid(),
            nn.Linear(256, 128),
            nn.Sigmoid(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )
        else:
          self.net = nn.Sequential(
             nn.Linear(obs_size, 32),
             nn.ReLU(),
             nn.Linear(32, n_actions)
        )
```

С использованием прошлых двух классов был реализован класс DQNAgent. В качестве параметров класса используются параметры: gamma (коэффициент дисконтирования от 0 до 1). Он определяет, как будущее зависит от прошлого. При равном 0 агент близок к жадному алгоритму, стремится к краткосрочной награде, при 1 рассчёт идёт на долгосрочную награду. Параметр epsilon влияет вероятность случайности выбранных действий, а epsilon_decay на скорость уменьшения epsilon.

На рисунках 1.1 – 1.2 показаны графики наград и потерь для первой архитектуры и значения коэффициентов, при которых были достигнуты результаты.

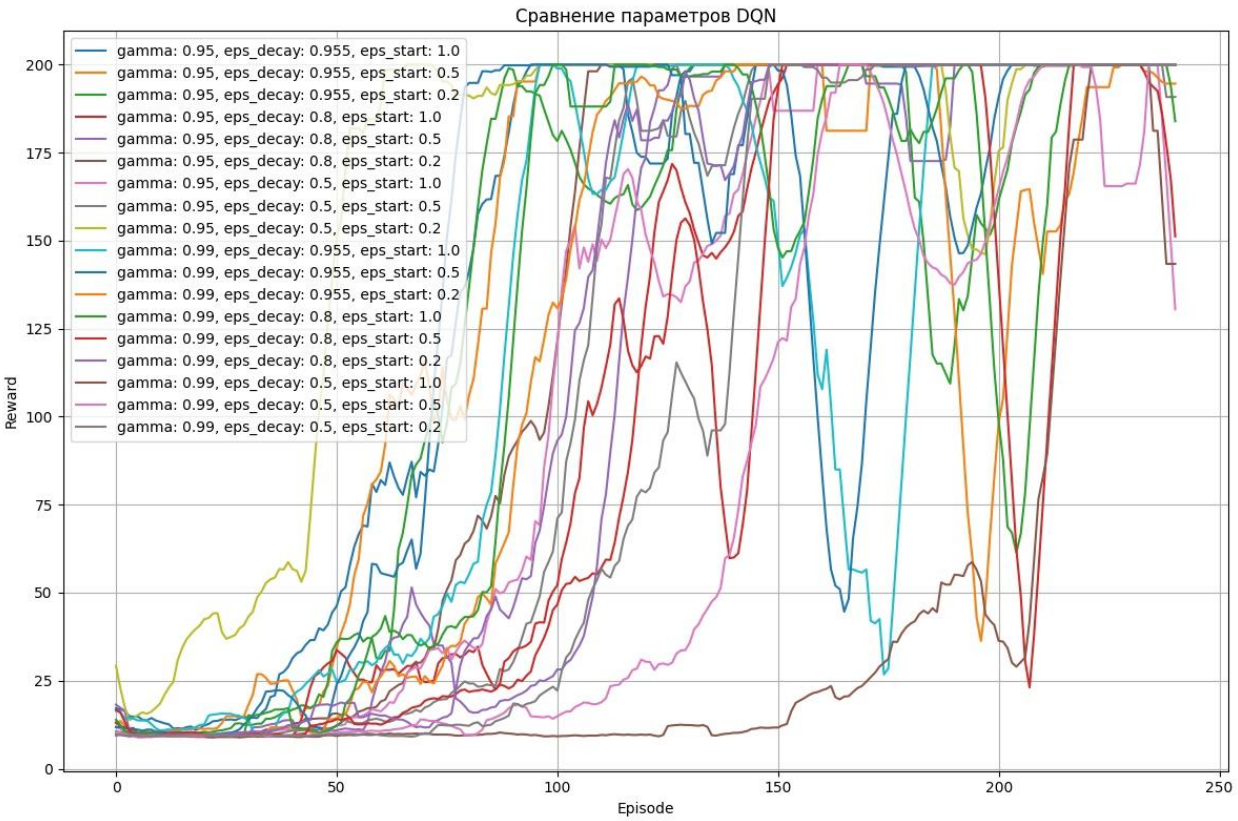

Рисунок 1.1 – График наград для первой архитектуры

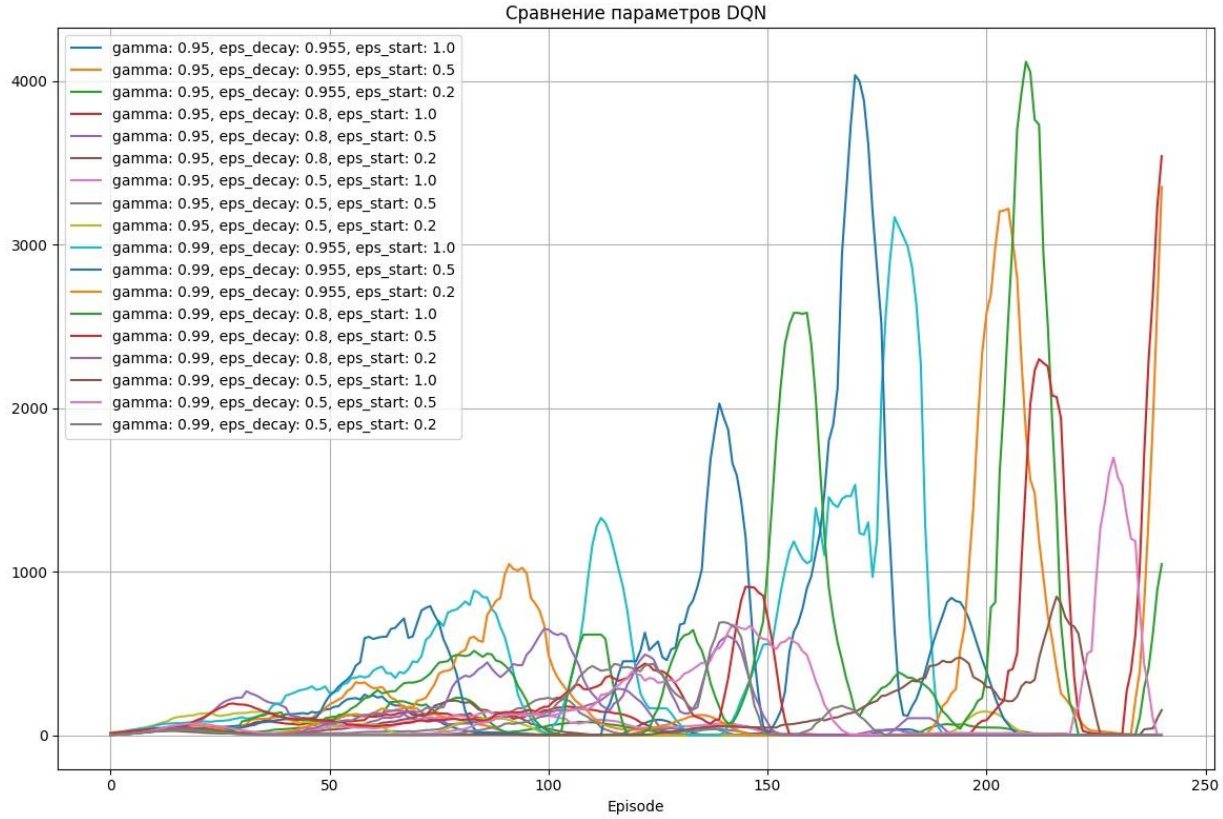

Рисунок 1.2 – График потерь для первой архитектуры

Для данной архитектуры лучшими можно назвать комбинации gamma = 0.95, eps_decay = 0.955, eps_start = 1.0 и gamma: 0.95, eps_decay: 0.5, eps_start: 0.2. «Салатовая» функция уже к 70 эпизоду приближается к максимальной награде, притом, что потери данной функции стремятся к нулю. При этом функция с параметрами 0.99, 0.5, 1.0 даже на 200 эпизоде имела значение награды 38, что говорит о медленной скорости обучения, что можно связать с высоким значения gamma, так как почти все функции с данным значением приближались к высоким значениям награды не раньше 150 эпизода.

На рисунках 2.1 – 2.2 показаны графики наград и потерь для второй архитектуры. Это самая сложная архитектура в работе, что сказалось на времени выполнения. Как видно на графике, функции со значением gamma = 0.95 быстрее достигают больших значений наград, при этом eps_start = 0.5 и 1.0. При этом многие функции имеют достаточно большие колебания, что видно и на графике потерь.

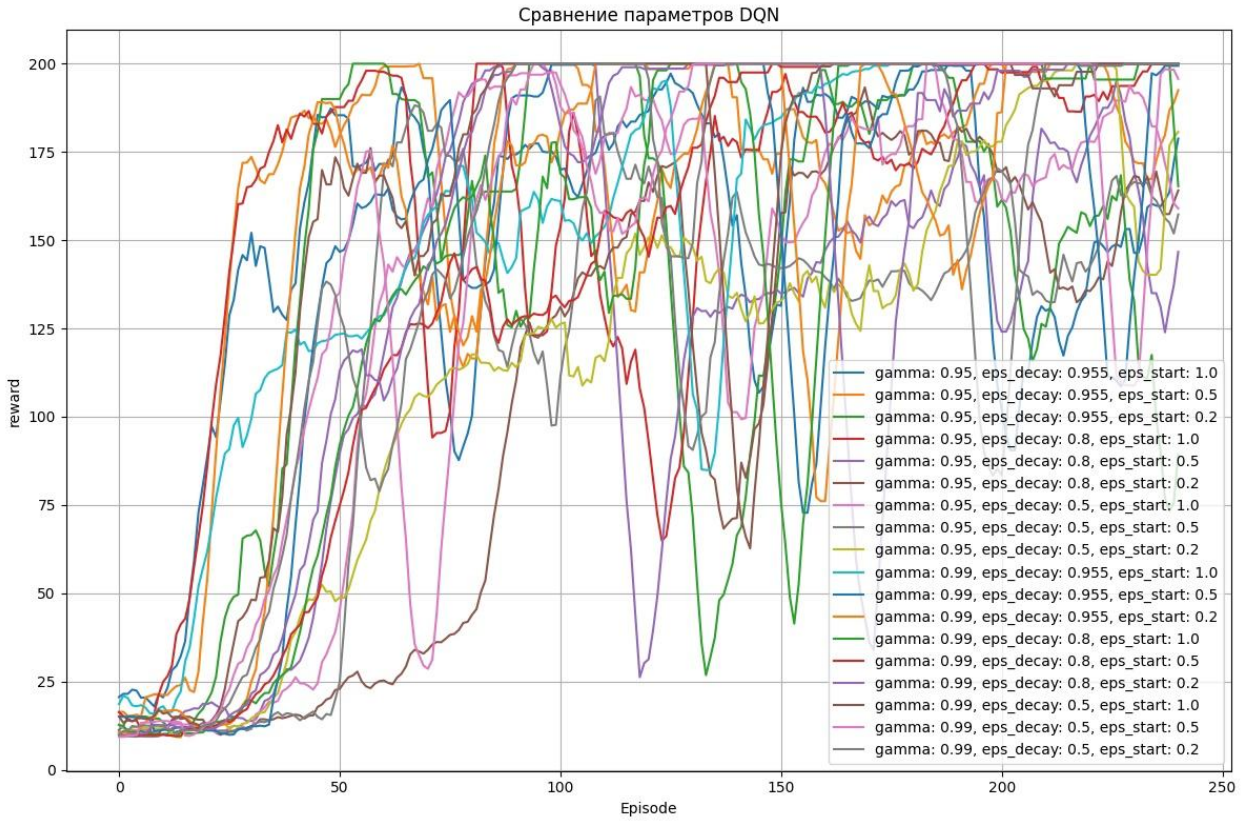

Рисунок 2.1 – График наград для второй архитектуры

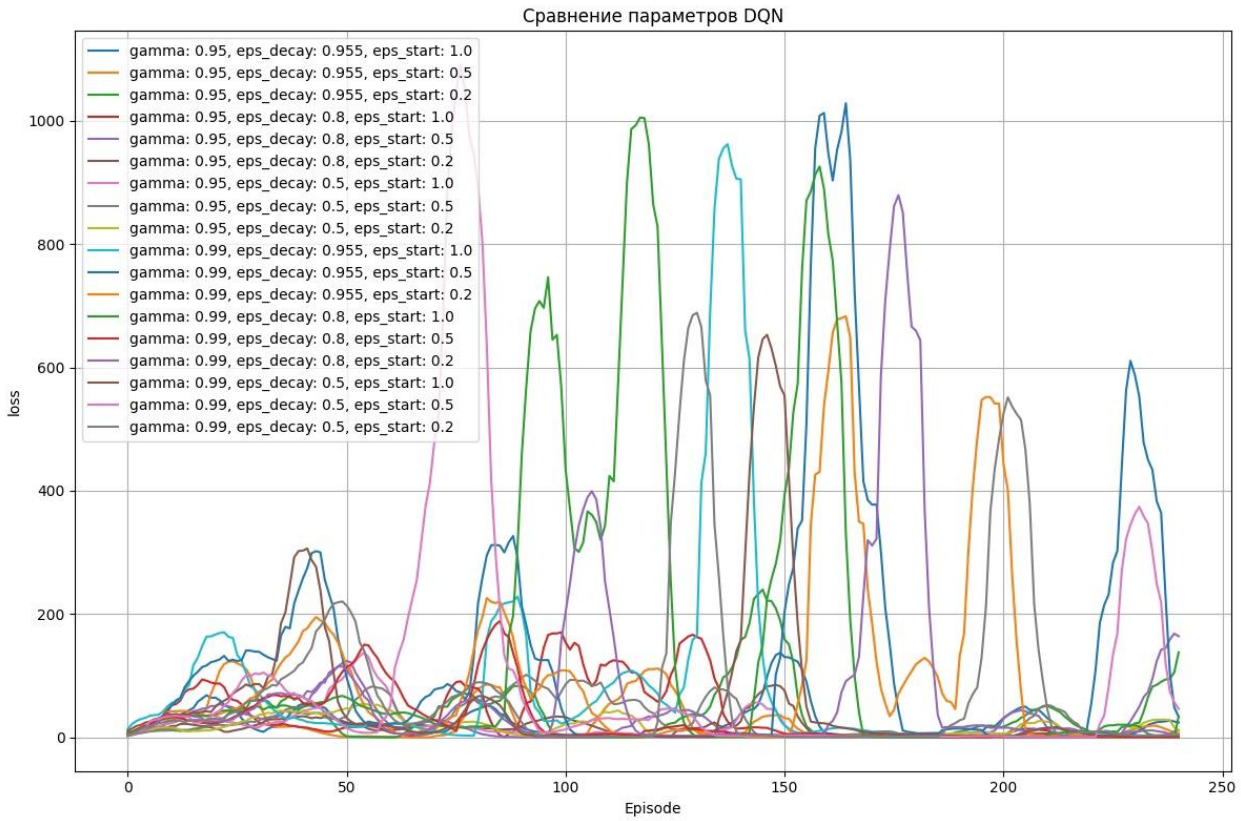

Рисунок 2.2 – График потерь для второй архитектуры

На рисунках 3.1 – 3.2 показаны графики наград и потерь для третьей архитектуры. Это самая простая архитектура, поэтому результаты были получены достаточно быстро. При этом это повлияло и на время обучения. Хорошие значения наград были достигнуты только к концу эпизодов. Однако обучение происходило достаточно равномерно, по сравнению с прошлыми графиками потерь, потери для этой архитектуры достаточно монотонны и лишь для некоторых значений коэффициентов имеются скачки.

Немного лучше остальных показали себя результаты для gamma 0.99, eps_decay 0.955 и eps_start 1.0 и gamma 0.99, eps_decay 0.8 и eps_start 0.5.

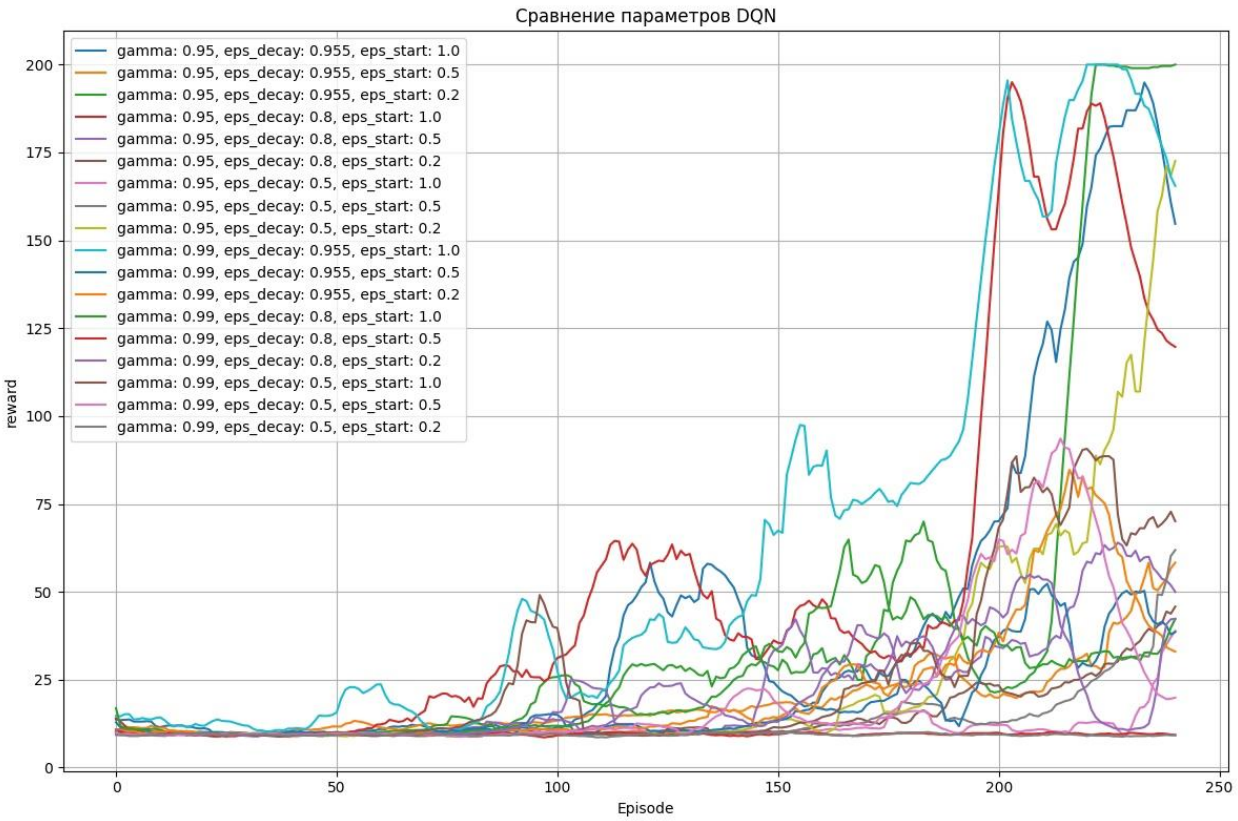

Рисунок 3.1 – График наград для третьей архитектуры

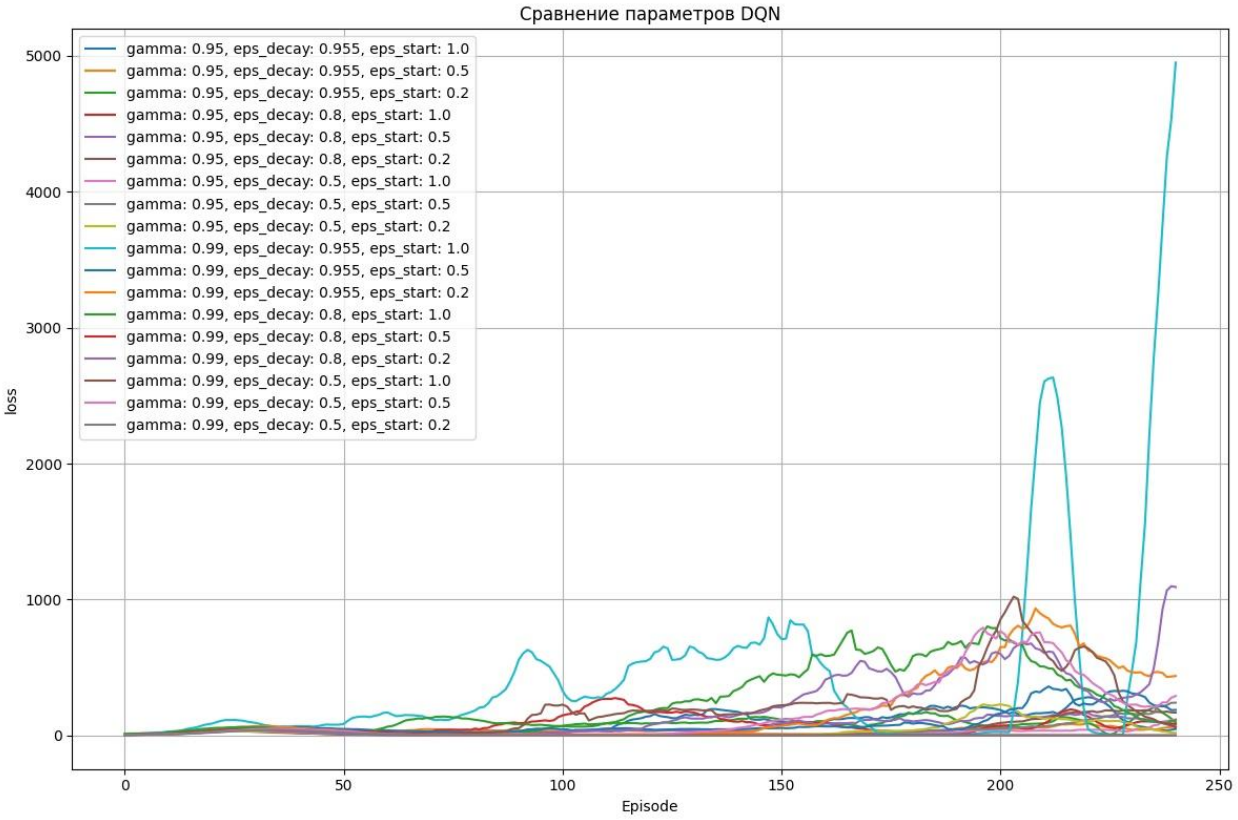

Рисунок 3.2 – График потерь для третьей архитектуры

Заключение

В результате данной работы был реализован алгоритм DQN для среды CartPole-v1. Было проанализировано влияние различных архитектур и параметров на скорость и качество обучения. Для разных архитектур были получены следующие значения: gamma = 0.95, eps_decay = 0.955, eps_start = 1.0; gamma: 0.95, eps_decay: 0.5, eps_start: 0.2; gamma = 0.95 eps_decay = 0.955, eps_start = 0.5 и 1.0. Можно сделать вывод, что значение коэффициента дисконтирования в районе 0.95 является оптимальным для данного окружения, независимо от архитектуры. Явное влияние epsilon сложно отследить, для задачи будет достаточно значений в диапазоне 0.5 – 1.0 для получения хороших результатов.

При этом простая архитектура дает достаточно медленное и стабильное обучение, но хорошую скорость вычислений, что подтверждает гипотезу влияния количества нейронов. Сложная архитектура дала некоторые колебания, что видно на функции потерь, что можно связать с переобучением, так как усложнение может быть избыточно для такой простой среды. Кроме того, результаты агента со сложной архитектурой ощутимо дольше вычислялись. Первая архитектура показала неплохую скорость обучения и с рядом коэффициентов также имела минимальные потери, из чего можно сделать вывод, что 128 нейронов является оптимальным для выполнения данной задачи.

Приложение А

```
import random
from collections import deque
import gymnasium as gym
import matplotlib.pyplot as plt
import numpy as np
import torch
from gymnasium.core import Env
from torch import nn, optim
from tqdm import tqdm

class ReplayBuffer:
    def __init__(self, capacity=1000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state,
done))

    def sample(self, batch_size): #нередко нужно взять не весь
буфер
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones =
zip(*batch)
        return (
            torch.tensor(np.array(states), dtype=torch.float32),
            torch.tensor(np.array(actions),
dtype=torch.float32),
            torch.tensor(np.array(rewards),
dtype=torch.float32),
            torch.tensor(np.array(next_states),
dtype=torch.float32),
            torch.tensor(np.array(dones), dtype=torch.float32),
        )
```

```
    def __len__(self):
        return len(self.buffer)

class QNetwork(nn.Module):
    def __init__(self, obs_size, n_actions, layer_num ):
#obs_size = 4, n_actions = 2 (или 0, или 1)
        super(QNetwork, self).__init__()
        if layer_num == 0:
            self.net = nn.Sequential(
                nn.Linear(obs_size, 128),
                nn.ReLU(),
                nn.Linear(128, n_actions)
            )
        elif layer_num == 1:
          self.net = nn.Sequential(
            nn.Linear(obs_size, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.Sigmoid(),
            nn.Linear(256, 128),
            nn.Sigmoid(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )
        else:
          self.net = nn.Sequential(
             nn.Linear(obs_size, 32),
             nn.ReLU(),
             nn.Linear(32, n_actions)
        )

    def forward(self, x): #обязаткльный метод
        return self.net(x)

class DQNAgent:
```

```
    def __init__(self, obs_size, n_actions, layer = 0, gamma =
0.99, epsilon_decay = 0.955, epsilon = 1):
        self.device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
        #важно, чтобы q_net и target в начале были одинаковые
        self.q_net = QNetwork(obs_size, n_actions,
layer).to(self.device)
        self.target_net = QNetwork(obs_size, n_actions,
layer).to(self.device)
        self.target_net.load_state_dict(self.q_net.state_dict())

        self.optimizer = optim.Adam(self.q_net.parameters(),
lr=1e-3)

        self.gamma = gamma #как будущее зависит от прошлого 0.99
        self.batch_size = 64
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay #на какой коэффициент
уменьшаем 0.955
        self.epsilon_min = 0.01

        self.replay_buffer = ReplayBuffer(1000)
        self.loss: float = 0.0

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, 1)

        with torch.no_grad():
            state_tensor = torch.tensor(state,
dtype=torch.float32, device=self.device)
            q_values = self.q_net(state_tensor)
            return torch.argmax(q_values).item()

    def train(self):
        if len(self.replay_buffer) < self.batch_size:
```

```
            return

        self.replay_buffer.sample(self.batch_size)

        state, action, reward, next_state, done =
self.replay_buffer.sample(self.batch_size)

        states = torch.tensor(state, dtype=torch.float32,
device=self.device)
        actions = torch.tensor(action, dtype=torch.long,
device=self.device)
        reward = torch.tensor(reward, dtype=torch.float32,
device=self.device)
        next_states = torch.tensor(next_state,
dtype=torch.float32, device=self.device)
        done = torch.tensor(done, dtype=torch.float32,
device=self.device)

        """ dqn update магия """

        q_values = self.q_net(states).gather(1,
actions.unsqueeze(1)).squeeze(1)
        next_q_values = self.target_net(next_states).max(1)[0]
        target_q_values = reward + self.gamma * next_q_values *
(1 - done)

        loss = nn.MSELoss()(q_values, target_q_values)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        self.loss = loss.item()

    def update_target(self):
        self.target_net.load_state_dict(self.q_net.state_dict())
```

```
def calc(lay, gamma, eps_decay, eps_start):
  agent = DQNAgent(4, 2, layer = lay, gamma = gamma,
epsilon_decay = eps_decay, epsilon = eps_start)
  num_episodes = 250
  env = gym.make("CartPole-v1", render_mode="rgb_array")
  done = False
  reward_history = []
  loss_history = []
  num_steps = 200
  key = f"gamma: {gamma}, eps_decay: {eps_decay}, eps_start:
{eps_start}"
  for episode in tqdm(range(num_episodes), desc=key):
      state, _ = env.reset()
      episode_reward = 0.0
      episode_loss = 0.0

      for step in range(num_steps):
          action = agent.select_action(state)
          next_state, reward, done, truncated, info =
env.step(action)
          agent.replay_buffer.push(state, action, reward,
next_state, done)

          state = next_state
          episode_reward += reward

          agent.train()
          episode_loss += float(agent.loss)

          if done:
              break
      agent.update_target()
      agent.epsilon = max(agent.epsilon * agent.epsilon_decay,
agent.epsilon_min)
      reward_history.append(episode_reward)
      loss_history.append(episode_loss)
```

```
      if episode % 10 == 0:
          print(f"Episode: {episode}, Reward: {episode_reward},
Loss: {episode_loss}")
  return reward_history, loss_history, key

def run_experiment(lay, gammas, epsilon_decas, epsilons):
    reward_results = {}
    loss_results = {}
    for gamma in gammas:
        for eps_decay in epsilon_decas:
            for eps_start in epsilons:
                  reward_history, loss_history, key = calc(lay,
gamma, eps_decay, eps_start)
                  reward_results[key] = reward_history
                  loss_results[key] = loss_history

    return reward_results, loss_results

def plot_results(results, name, smooth_window=10):
    plt.figure(figsize=(12, 8))
    for label, rewards in results.items():
        smoothed = np.convolve(rewards,
np.ones(smooth_window)/smooth_window, mode='valid')
        plt.plot(smoothed, label=label)

    plt.xlabel('Episode')
    plt.ylabel(name)
    plt.title('Сравнение параметров DQN')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()



gamma_values = [0.95, 0.99]
epsilon_decay_values = [0.955, 0.8, 0.5]
```

```
initial_epsilons = [1.0, 0.5, 0.2]

reward, loss = run_experiment(0, gamma_values,
epsilon_decay_values, initial_epsilons)
plot_results(reward, 'reward')
plot_results(loss, 'loss')

reward, loss = run_experiment(1, gamma_values,
epsilon_decay_values, initial_epsilons)

def plot_results(results, name, smooth_window=10):
    plt.figure(figsize=(12, 8))
    for label, rewards in results.items():
        smoothed = np.convolve(rewards,
np.ones(smooth_window)/smooth_window, mode='valid')
        plt.plot(smoothed, label=label)

    plt.xlabel('Episode')
    plt.ylabel(name)
    plt.title('Сравнение параметров DQN')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

plot_results(reward, 'reward')
plot_results(loss, 'loss')

reward, loss = run_experiment(2, gamma_values,
epsilon_decay_values, initial_epsilons)
plot_results(reward, 'reward')
plot_results(loss, 'loss')
```