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

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Обучение с подкреплением»
Тема: Реализация DQN для среды CartPole-v1

Студент гр. 0310 _____ Якушкин Д.А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025 г.

Цель работы.

Реализация DQN для среды CartPole-v1. Исследование влияния различных параметров: архитектура сети, значения gamma и epsilon_decay, влияние epsilon на скорость обучения

Задание.

1. Реализовать DQN для среды CartPole-v1
2. Изменить архитектуру нейросети.
3. Попробовать разные значения gamma и epsilon_decay.
4. Провести исследование как изначальное значение epsilon влияет на скорость обучения

Выполнение работы.

1. **Реализация DQN**

DQNAgent это класс, который реализует сам алгоритм DQN. Его задача принимать решения, которые приведут к наибольшей выгоде. Стандартные параметры для него будут следующими:

```
base_config = {
        "layers”: [64, 64],
        "gamma": 0.99,
        "epsilon_decay": 0.99,
        "epsilon_start": 1.0
    }
```

2. **Влияние архитектуры на обучение**

Для экспериментов были выбраны следующие структуры слоев:

```
layer_params = {
  "default": [64, 64],
  "deep":    [128, 128, 64],
  "wide":    [256, 128],
```

```
    "small":   [32, 32],
}
```

Были получены сводные графики loss и reward для каждого эксперимента. График loss можно увидеть на рисунке 1. График reward можно увидеть на рисунке 2.

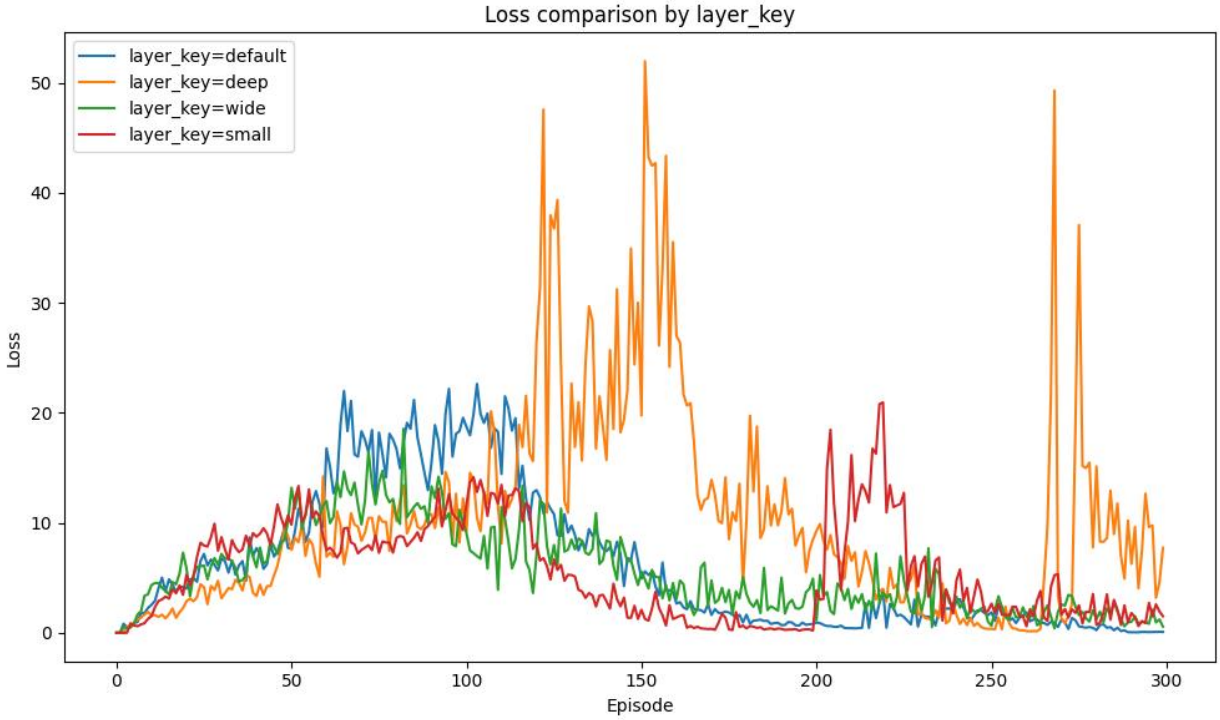

Рисунок 1 - Сравнение loss для различных конфигураций слоёв

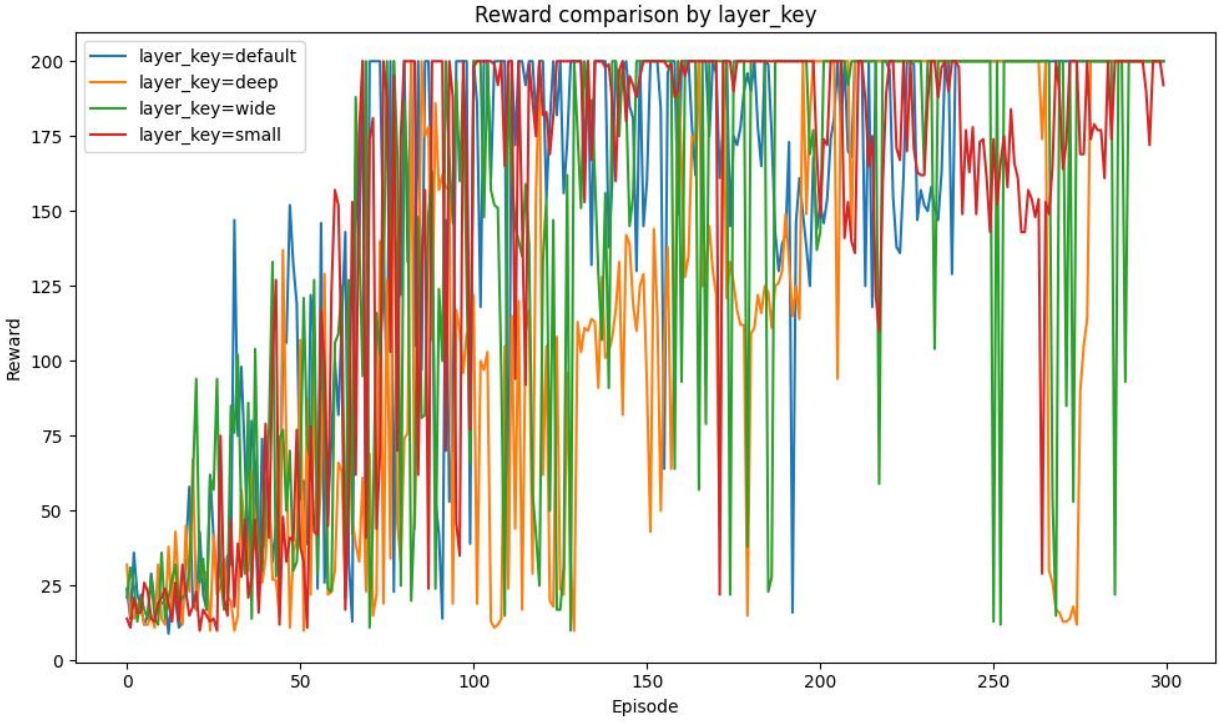

Рисунок 2 - Сравнение loss для различных конфигураций слоёв

Исходя из графиков можно судить что наилучший показатель нейросеть показывает при small и default конфигурациях слоев. Это может происходить из за того, что задача может быть слишком легкой для больших нейросетей и возможно переобучение, для больших нейросетей также может понадобиться большее количество эпизодов ввиду более медленной сходимости.

3. Влияния gamma на обучение

Gamma - это параметр дисконтирования. Он условно указывает то, насколько нейросеть учитывает будущие награды. Чем выше gamma, тем дольше мы можем ожидать большую выгоду.

Для эксперимента были выбраны значения 0.95 и 0.99. Результаты можно увидеть на рисунках 3 и 4.

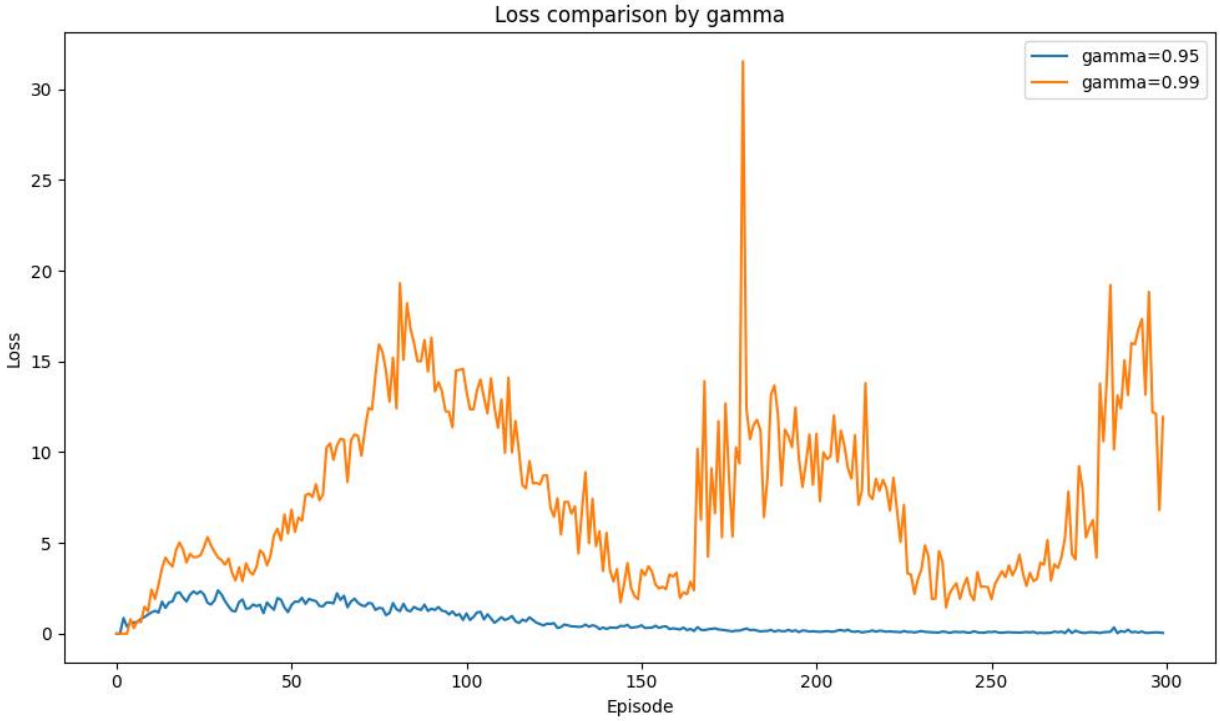

Рисунок 3 - Сравнение loss для различных параметров gamma

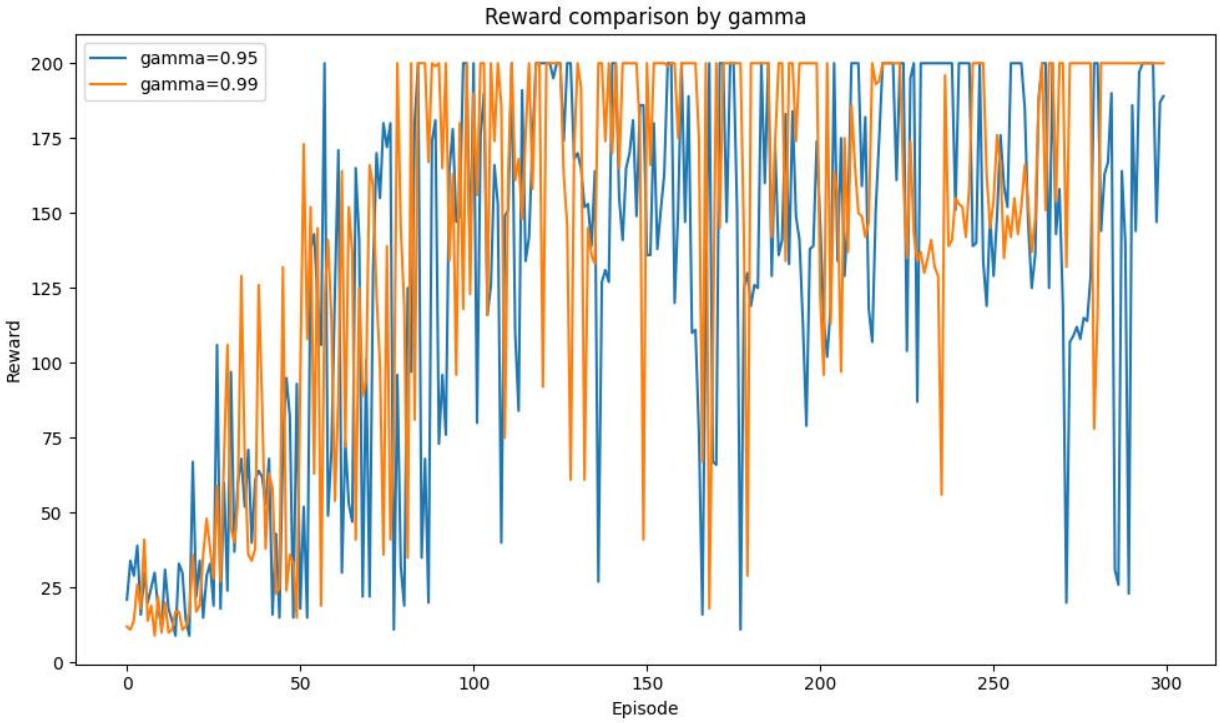

Рисунок 4 - Сравнение loss для различных параметров gamma

Исходя из графиков можно сделать вывод что gamma=0.95 даёт более быстрое достижение результата с меньшим количеством ошибок.

4. Влияние epsilon на обучение

Epsilon - это возможность случайного действия. Этот параметр задает то, как часто нейросеть будет предпринимать новые действия, вместо выполнения уже изученных. Для ее задания применяются два параметра: epsilon_start и epsilon_decay. Первый параметр это начальное значение, а второй это множитель который применяется к начальному параметру каждый эпизод.

Для эксперимента были выбраны значения множителя 0.95 и 0.99. Результаты можно увидеть на рисунках 5 и 6.

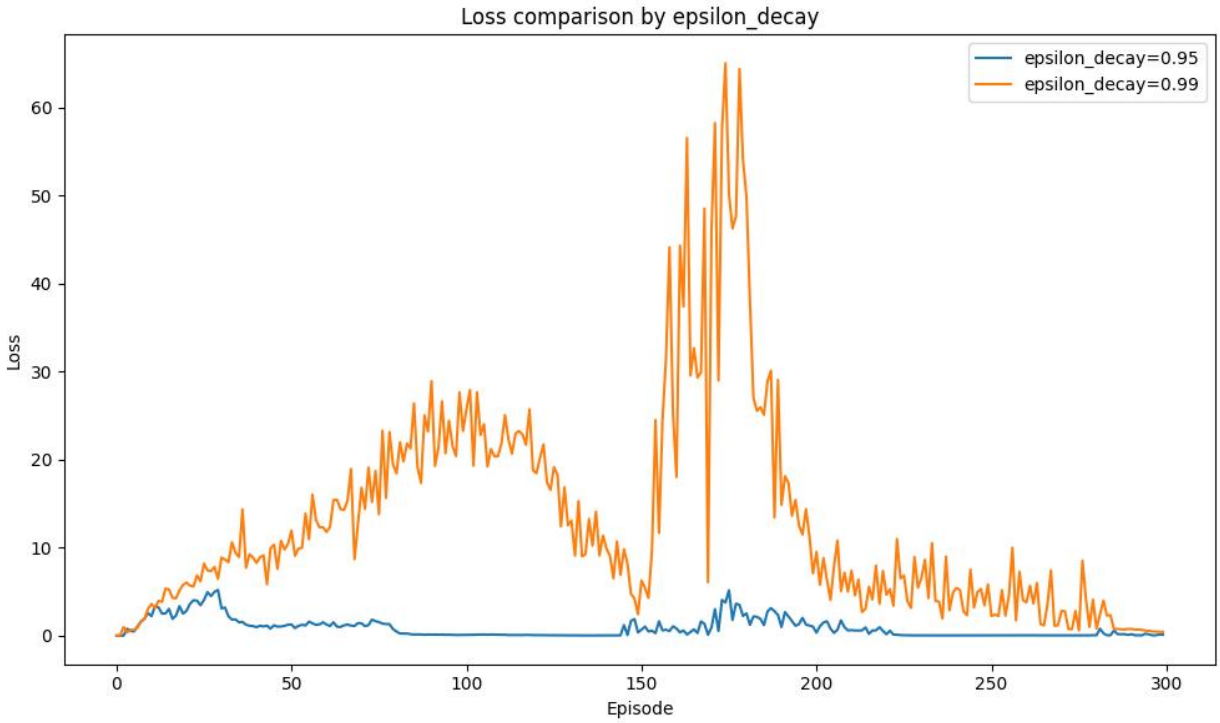

Рисунок 5 - Сравнение loss для различных параметров epsilon_decay

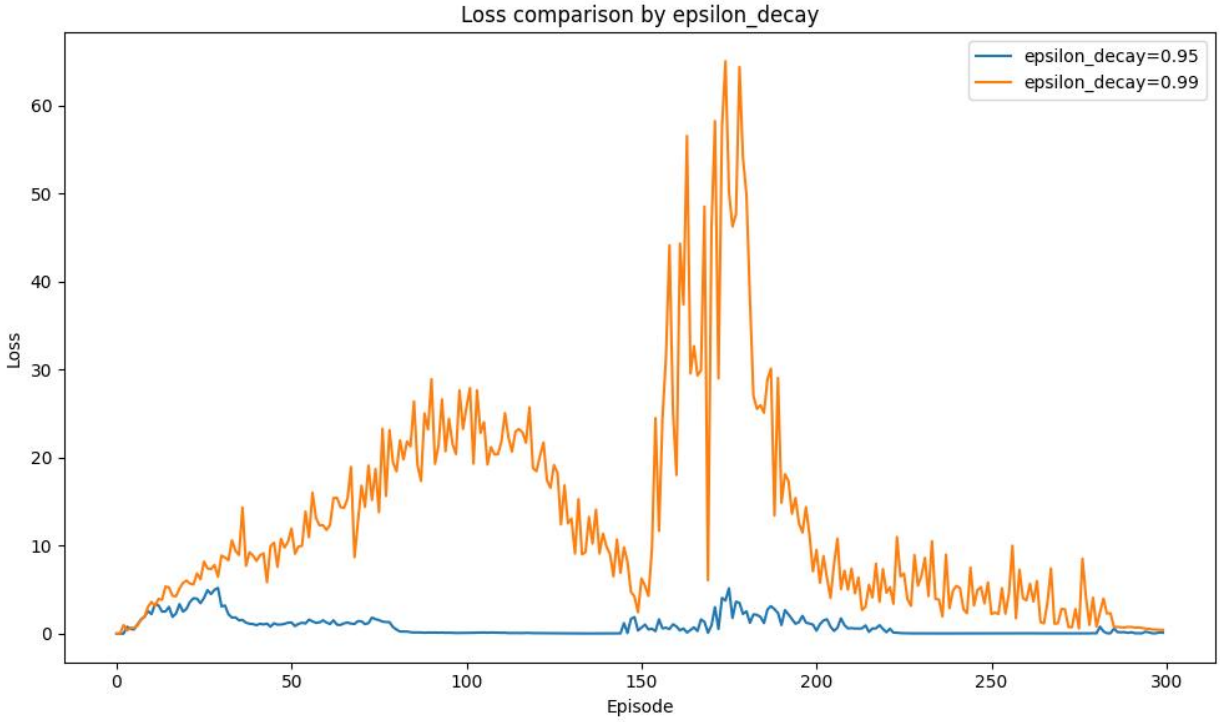

Рисунок 6 - Сравнение reward для различных параметров epsilon_decay

Можно сделать вывод что при более быстром уменьшении вероятности случаности действия мы получаем более быстрое достижение результата и меньшее количество loss.

На графиках 7 и 8 можно увидеть влияние параметра epsilon_start на

обучение.

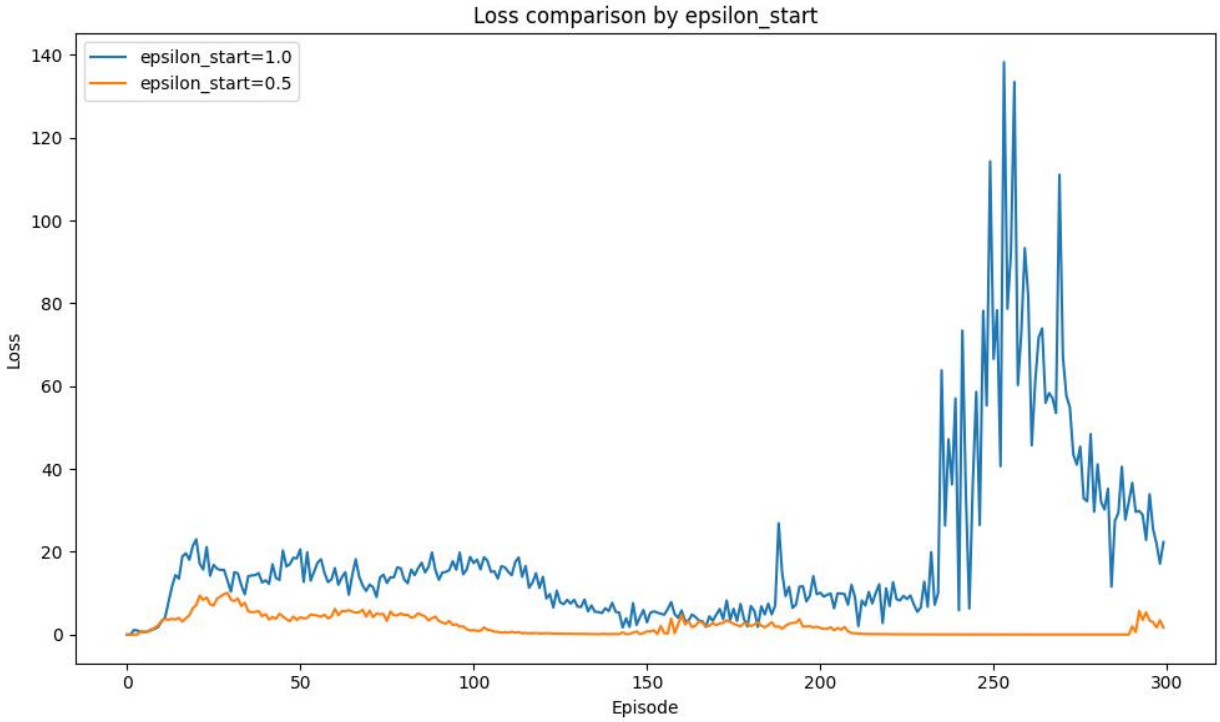

Рисунок 5 - Сравнение loss для различных параметров epsilon_start

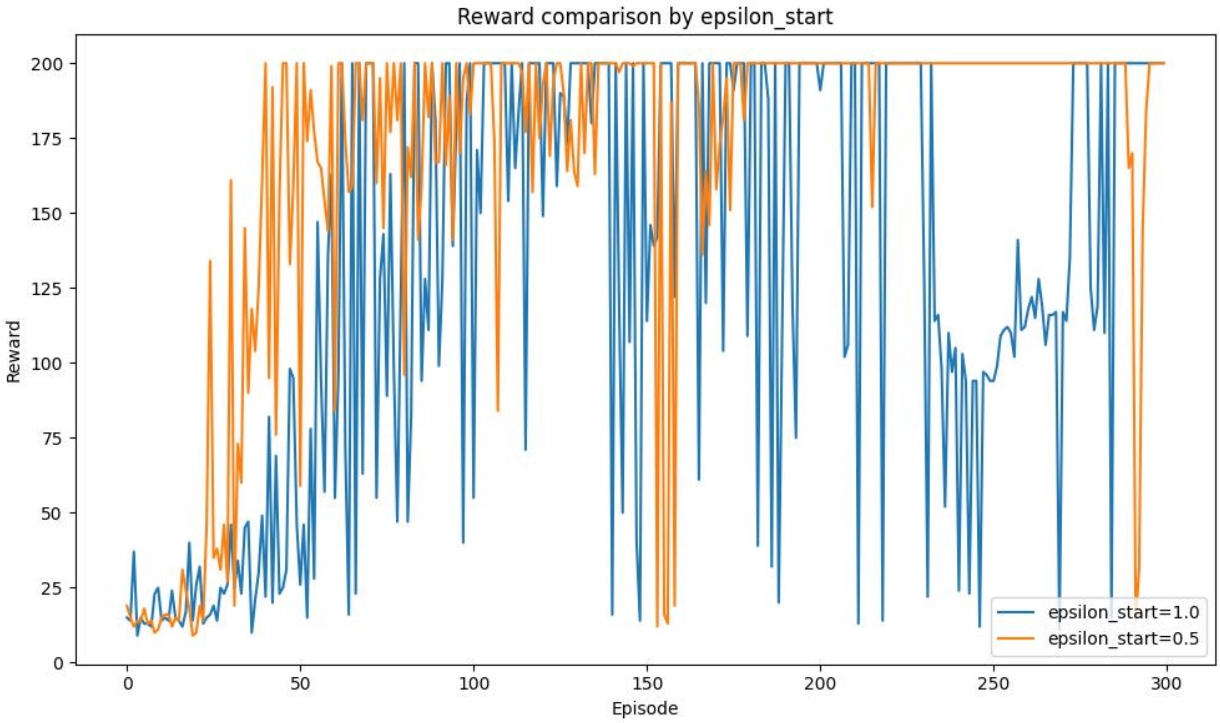

Рисунок 5 - Сравнение reward для различных параметров epsilon_start

Исходя из графиков можно заметить что более низкое значение параметра дает более быстрое обучение.

Выводы.

Был реализован DQN для среды CartPole-v1. Было проведено исследование влияния некоторых параметров сети на результат ее обучения.

ПРИЛОЖЕНИЕ А

```
import gymnasium as gym
import torch
import numpy as np
from collections import deque
from torch import nn, optim
import random
import matplotlib.pyplot as plt
import os
import time

# Параметры архитектуры
layer_params = {
    "default": [64, 64],
    "deep":    [128, 128, 64],
    "wide":    [256, 128],
    "small":   [32, 32],
}

def build_network(input_dim, output_dim, layer_sizes):
    layers = []
    prev_dim = input_dim
    for size in layer_sizes:
        layers.append(nn.Linear(prev_dim, size))
        layers.append(nn.ReLU())
        prev_dim = size
    layers.append(nn.Linear(prev_dim, output_dim))
    return nn.Sequential(*layers)

class ReplayBuffer:
    def __init__(self, capacity=10000):
        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):
        batch = random.sample(self.buffer, batch_size)
```

```
            state, action, reward, next_state, done = zip(*batch)
            return (
                torch.tensor(np.array(state), dtype=torch.float32),
                torch.tensor(action, dtype=torch.int64),
                torch.tensor(reward, dtype=torch.float32),
                torch.tensor(next_state, dtype=torch.float32),
                torch.tensor(done, dtype=torch.float32),
            )

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

    class DQNAgent:
        def __init__(self, obs_size, n_actions, layers, gamma,
epsilon_decay, epsilon_start):
            self.device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
            self.q_net = build_network(obs_size, n_actions,
layers).to(self.device)
            self.target_net = build_network(obs_size, n_actions,
layers).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
            self.batch_size = 64
            self.epsilon = epsilon_start
            self.epsilon_decay = epsilon_decay
            self.epsilon_min = 0.01

            self.replay_buffer = ReplayBuffer()

        def select_action(self, state):
            if random.random() < self.epsilon:
                return random.randint(0, 1)
            state_t = torch.tensor(state, dtype=torch.float32,
device=self.device)
            with torch.no_grad():
```

```
                q_vals = self.q_net(state_t)
            return int(q_vals.argmax())

        def train(self):
            if len(self.replay_buffer) < self.batch_size:
                return None
            state, action, reward, next_state, done =
self.replay_buffer.sample(self.batch_size)
            state = state.to(self.device)
            action = action.to(self.device)
            reward = reward.to(self.device)
            next_state = next_state.to(self.device)
            done = done.to(self.device)

            q_values = self.q_net(state).gather(1,
action.unsqueeze(1)).squeeze()
            next_q = self.target_net(next_state).max(1)[0]
            expected_q = reward + self.gamma * next_q * (1 - done)

            loss = nn.MSELoss()(q_values, expected_q)
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
            return loss.item()

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

    def run_experiment(layer_key, gamma, epsilon_decay, epsilon_start,
episodes=300):
        env = gym.make("CartPole-v1")
        agent = DQNAgent(obs_size=4, n_actions=2,
            layers=layer_params[layer_key],
            gamma=gamma,
            epsilon_decay=epsilon_decay,
            epsilon_start=epsilon_start)

        reward_history = []
        loss_history = []
```

```
        start_time = time.time()

        for episode in range(episodes):
            state, _ = env.reset()
            episode_reward = 0
            episode_losses = []

            for _ in range(200):
                action = agent.select_action(state)
                next_state, reward, done, _, _ = env.step(action)
                agent.replay_buffer.push(state,    action,    reward,
next_state, done)

                loss = agent.train()
                if loss is not None:
                    episode_losses.append(loss)

                state = next_state
                episode_reward += reward
                if done:
                    break

            agent.update_target()
            agent.epsilon = max(agent.epsilon * agent.epsilon_decay,
agent.epsilon_min)

            reward_history.append(episode_reward)
            avg_loss     =     float(np.mean(episode_losses))     if
episode_losses else 0.0
            loss_history.append(avg_loss)

        env.close()
        elapsed = time.time() - start_time
        return reward_history, loss_history, elapsed

    def run_all():
        os.makedirs("results", exist_ok=True)
        base_config = {
            "layer_key": "default",
```

```
    "gamma": 0.99,
    "epsilon_decay": 0.99,
    "epsilon_start": 1.0
}

param_variations = {
    "layer_key": list(layer_params.keys()),
    "gamma": [0.95, 0.99],
    "epsilon_decay": [0.95, 0.99],
    "epsilon_start": [1.0, 0.5],
}

param_rewards = {k: {} for k in param_variations.keys()}
param_losses = {k: {} for k in param_variations.keys()}

for param, values in param_variations.items():
    for value in values:
        config = base_config.copy()
        config[param] = value
        label = f"{param}_{value}"

        print(f"Running: {label}")
        rewards, losses, _ = run_experiment(
            config["layer_key"],
            config["gamma"],
            config["epsilon_decay"],
            config["epsilon_start"]
        )

        plt.figure()
        plt.plot(rewards)
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.title(f"Reward: {label}")
        plt.savefig(f"results/{label}_reward.png")
        plt.close()

        plt.figure()
        plt.plot(losses)
```

```
                plt.xlabel("Episode")
                plt.ylabel("Loss")
                plt.title(f"Loss: {label}")
                plt.savefig(f"results/{label}_loss.png")
                plt.close()

                param_rewards[param][str(value)] = rewards
                param_losses[param][str(value)] = losses

        for param in param_variations.keys():
            plt.figure(figsize=(10, 6))
            for val, rewards in param_rewards[param].items():
                plt.plot(rewards, label=f"{param}={val}")
            plt.xlabel("Episode")
            plt.ylabel("Reward")
            plt.title(f"Reward comparison by {param}")
            plt.legend()
            plt.tight_layout()
            plt.savefig(f"results/compare_{param}_reward.png")
            plt.close()

            plt.figure(figsize=(10, 6))
            for val, losses in param_losses[param].items():
                plt.plot(losses, label=f"{param}={val}")
            plt.xlabel("Episode")
            plt.ylabel("Loss")
            plt.title(f"Loss comparison by {param}")
            plt.legend()
            plt.tight_layout()
            plt.savefig(f"results/compare_{param}_loss.png")
            plt.close()

        print("All experiments finished. Summary plots saved in
'results/' folder.")

    if __name__ == "__main__":
        run_all()
```