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 _____ Бодунов П.А.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург

2025

Цель работы.

Реализовать алгоритм DQN с помощью библиотеки pytorch для решения задачи CartPole-v1.

Постановка задачи.

1) Реализовать базовую версию DQN для решения задачи CartPole-v1.
2) Проанализировать изменение в скорости обучения при изменении архитектуры нейронной сети.
3) Проанализировать влияние изменения параметров *gamma* и *epsilon_decay*.
4) Провести исследование как изначальное значение *epsilon_start* влияет на скорость обучения.

Выполнение задач.

1) **Реализация DQN.**

Среда CartPole-v1 представляет собой задачу балансировки. В ней участвуют каретка и стержень, который на ней установлен. Состояние среды описывается четырьмя вещественными числами: они кодируют положение каретки, её скорость, угол наклона стержня и скорость его изменения. Агент может совершать одно из двух действий: толкать каретку влево или вправо. Его цель — как можно дольше удерживать стержень в вертикальном положении. Эпизод заканчивается, если стержень падает, каретка выходит за допустимые границы или проходит 500 шагов. За каждый шаг агент получает награду +1. Таким образом, чем дольше система сохраняет равновесие, тем больше суммарное вознаграждение.

Класс ReplayBuffer хранит последние capacity переходов, которые состоят из текущего состояния, действия, награды, следующего состояния и флага done.

Методы:

- __init__(self, capacity) - задает буфер в виде очереди с максимальным размером capacity.
- push(self, *args) - добавление перехода в буффер.

- sample(self, batch_size) - сэмплирование (возврат из буфера batch_size рандомных переходов)
- __len__(self) - текущий размер буфера.

Было реализовано 3 архитектуры нейронной сети:

- QNetwork1 – состоит из 3 линейных слоев с функциями активации ReLU.
- QNetwork2 – состоит из 5 линейных слоев с функциями активации ReLU.
- QNetwork3 – состоит из 5 линейных слоев с 2 функциями активации ReLU и 2 функциями активации Sigmoid.

Также был реализован агент DQNAgent, содержаащий следующие методы:

- select_action(self, state) – выбор действия на основе состояния state.
- train_step(self) – шаг обучения. Пока в буффере не будет находится достаточно данных, агент накапливает опыт. Далее из буффера сэмплируются данные, вычисляются Q-функция батчей и более близкая к реальности оценка возможной награды при помощи уравнения Беллмана, обновляются веса q-сетки, где loss – MSE.
- update_epsilon(self) – обновление ε для ε -жадной политики.
- update_target(self) – обновление таргетной сетки новыми значениями q-сетки.

Обучение агента происходит в функции train(agent, env, n_epochs, steps), где agent – агент, env – среда в которой находится агент, n_epoch – количество эпох для обучения, steps – максимально количество шагов агента в одну эпоху. Агент выбирает действие, выполняет его, после чего в буффер кладутся данные перехода, которые сообщает окружение, происходит шаг обучения и вычисляется loss. После того, как агент попадет в терминальное состояние или выполнит максимальное количество шагов в эпоху, происходит обновление таргетной сетки.

2) Влияние изменения архитектуры нейронной сети.

Архитектура нейронной сети определяет её структуру: количество слоёв, типы нейронов, связи между ними и способ обработки данных. От неё зависит, как сеть обучается, насколько хорошо решает задачи и справляется с различными типами данных.

Графики награды от количества эпох и функции потери от количества эпох результатов обучения моделей представлены на рисунке 1:

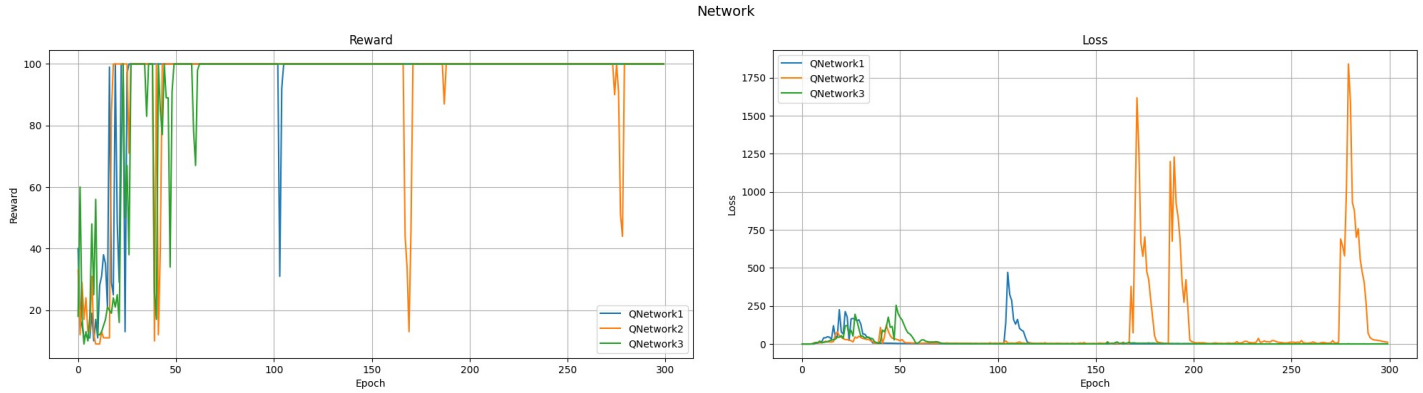

Рисунок 1 – Обучение разных архитектур

Исходя из построенных графиков, можно сделать вывод, что более сложная архитектура QNetwork3 после обучения стабильно набирает необходимую награду для завершения, также ее loss довольно быстро сходится к минимуму. QNetwork2 ведет себя хуже других моделей (на графике loss заметны большие скачки, также моделе приходится дообучаться), откуда можно сделать вывод, что усложнение архитектуры не всегда ведет к улучшению результата.

3) **Влияние изменения параметра *gamma*.**

Параметр *gamma* отвечает за коэффициент дисконтирования (чем он ближе к 1, тем модель больше заинтересована в дальней награде, чем ближе к 0, тем больше заинтересована в краткосрочной награде). Эксперимент проводился на значениях *gamma*: 0.6, 0.8, 0.99 (см. рисунок 2):

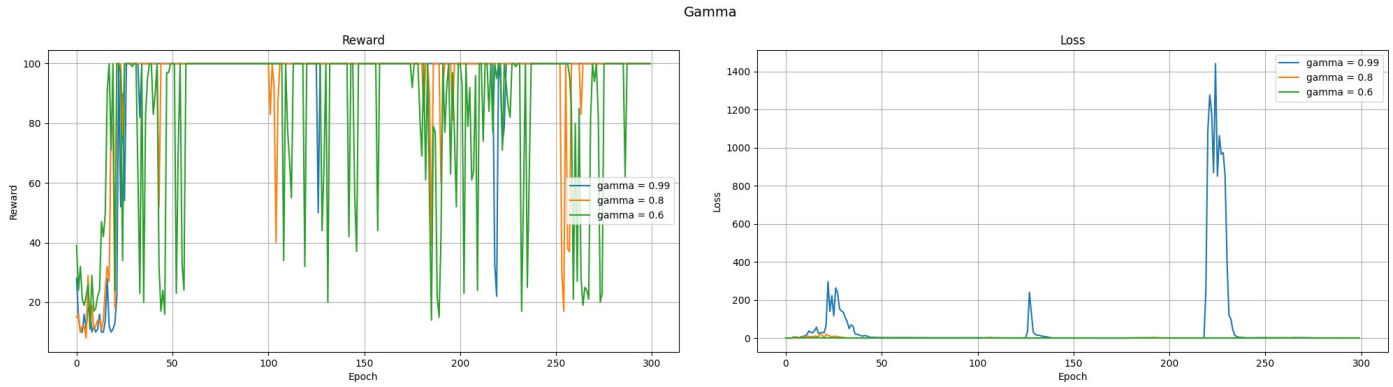

Рисунок 2 – Обучение с разными значениями *gamma*

Исходя из полученных результатов, можно сделать вывод, что при коэффициенте дисконтирования в 0.99 модель стабильней доходит до необходимой награды, в то время как при уменьшении этого параметра модель хуже обучается (увеличивается количество скачков). Таким образом в данной задаче модель больше заинтересована в дальней награде, что логично поскольку цель агента как можно дольше удержать стержень.

4) **Влияние изменения параметра *epsilon_decay.***

Параметр *epsilon_decay* отвечает за то, насколько быстро будет уменьшаться изначальное значение *epsilon_start*. Чем выше epsilon_decay, тем быстрее агент переходит от исследования (exploration) к эксплуатации (exploitation). Эксперимент проводился на значениях *epsilon_decay*: 0.6, 0.8, 0.95 (см. рисунок 3)

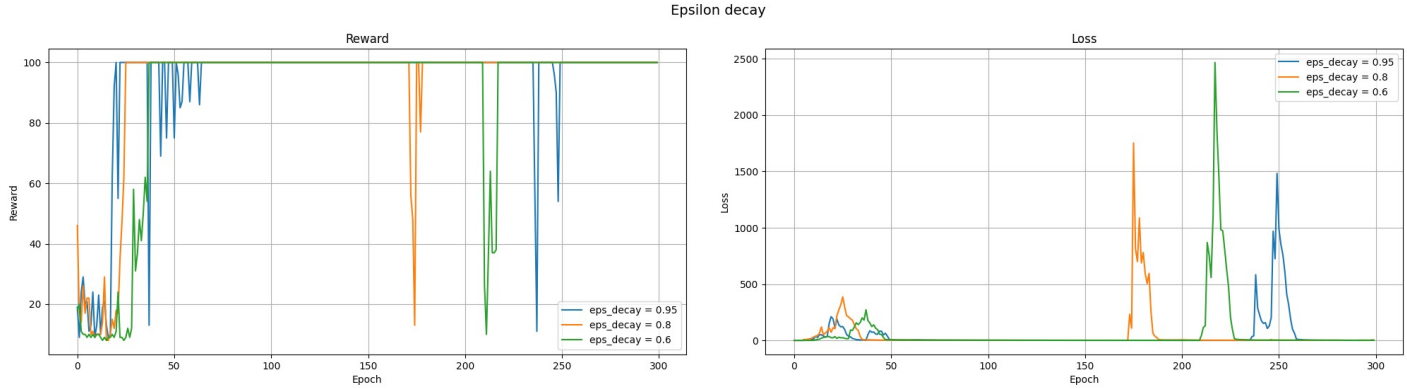

Рисунок 3 - Обучение с разными значениями *epsilon_decay*

Исходя из полученных графиков, можно сделать вывод, что при *epsilon_decay* = 0.95 модель быстрее вышло на необходимое количество

награды, но потом ей пришлось дообучаться (примерно с 40 по 70 эпоху). Это связано с тем, что модель запомнила действия в конкретных положениях. При *epsilon_decay* равном 0.6 модель обучилась лучше, а при 0.8 еще и быстрее, что указывает на то, что при *epsilon_decay* = 0.8 соблюден баланс от исследования к эксплуатации.

5) **Влияние изменения параметра *epsilon_start*.**

Параметр *epsilon_start* влияет на изначальную частоту выбора случайных действий (исследование окружения) вместо действий модели. Эксперимент проводился на значениях *epsilon_start*: 0.3, 0.6, 0.8 и 0.9 (см. рисунок 4):

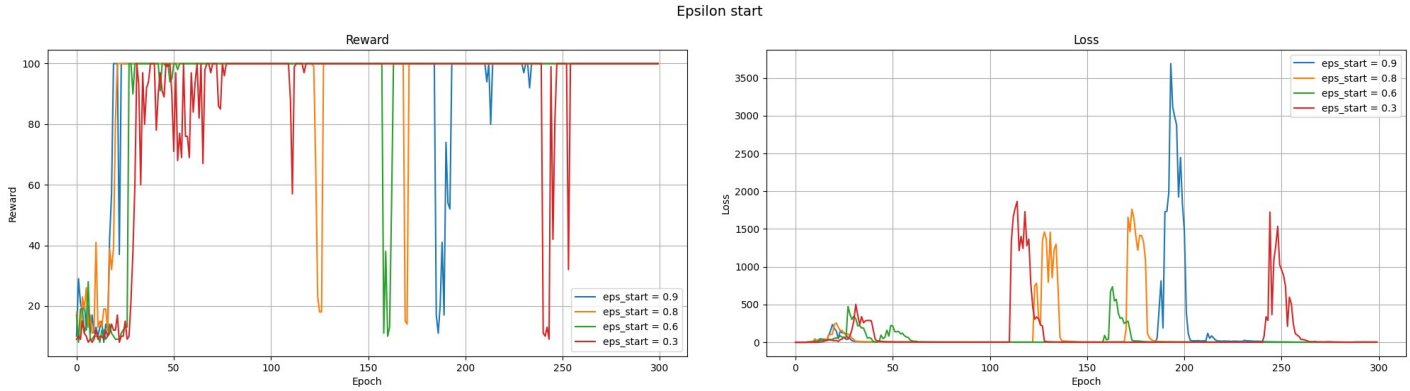

Рисунок 4 - Обучение с разными значениями *epsilon_start*

Исходя из полученных графиков, можно сделать вывод, что модель с большим *epsilon_start* быстрее обучается, поскольку агент больше изучает окружение, чем находит конкретный алгоритм выбора действий.

Выводы.

В ходе работы был изучен и реализован алгоритм обучения с подкреплением DQN. Было исследовано влияние различных параметров на обучение DQN в среде CartPole-v1. Усложнение архитектуры нейронной сети не всегда дает лучшие результаты, простая нейронная QNetwork1 сеть в данной задаче хорошо справляется с обучением. По результатам gamma в данной задаче модель больше заинтересована в дальней награде. Параметры epsilon_start и epsilon_decay влияют на изначальную частоту выбора случайных действий

(исследование окружения) вместо действий модели и скорость переходита агента от исследования (exploration) к эксплуатации (exploitation).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import math
import random
from collections import namedtuple, deque
from itertools import count
from typing import Type
from gymnasium.wrappers import RecordVideo
import gymnasium as gym
import torch
from tqdm import tqdm
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from dataclasses import dataclass

device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

@dataclass
class Params:
    batch_size: int = 128
    gamma: float = 0.99
    eps_start: float = 0.9
    eps_end: float = 0.05
    eps_decay: float = 0.95
    lr: float = 0.001

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

    def push(self, *transition):
        self.buffer.append(transition)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*batch)
```

```
        return (
            torch.tensor(state, dtype=torch.float32),
            torch.tensor(action, dtype=torch.long),
            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 QNetwork1(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(QNetwork1, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(n_observations, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )

    def forward(self, x):
        return self.net(x)


class QNetwork2(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(QNetwork2, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(n_observations, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
```

```
            nn.Linear(128, n_actions)
        )

    def forward(self, x):
        return self.net(x)

class QNetwork3(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(QNetwork3, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(n_observations, 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)
        )

    def forward(self, x):
        return self.net(x)

class DQNAgent:
    def __init__(self, model_type: Type[nn.Module], env: gym.Env,
params: Params = None, buff_size = 1000):
        initial_state, _ = env.reset()
        n_observations = len(initial_state)
        self.n_actions = env.action_space.n
        self.params = params if params is not None else Params()

        self.q_net = model_type(n_observations, self.n_actions)
        self.target_net = model_type(n_observations,
self.n_actions)
        self.target_net.load_state_dict(self.q_net.state_dict())

        self.optimizer = optim.Adam(self.q_net.parameters(),
lr=self.params.lr)
```

```
        self.buffer = ReplayBuffer(buff_size)
        self.eps = self.params.eps_start
        self.q_net.to(device)
        self.target_net.to(device)

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

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

    def train_step(self):
        if len(self.buffer) < self.params.batch_size:
            return 0
        state, action, reward, next_state, done =
self.buffer.sample(self.params.batch_size)
        state, action, reward, next_state, done =
state.to(device), \
            action.to(device), reward.to(device),
next_state.to(device), done.to(device)

        q_vals = self.q_net(state).gather(1,
action.unsqueeze(1)).squeeze(1)
        with torch.no_grad():
            target = reward + self.params.gamma *
self.target_net(next_state).max(1)[0] * (1 - done)
        loss = nn.MSELoss()(q_vals, target)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        return loss.item()

    def update_epsilon(self):
        self.eps = max(self.eps * self.params.eps_decay,
self.params.eps_end)
```

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


def train(agent, env, n_epochs, steps):
    reward_history = []
    loss_history = []
    epoch_tqdm = tqdm(range(n_epochs))
    for epoch in epoch_tqdm:
        state, _ = env.reset()
        total_reward = 0
        total_loss = 0
        for _ in range(steps):
            action = agent.select_action(state)
            next_state, reward, done, truncated, _ =
env.step(action)
            agent.buffer.push(state, action, reward, next_state,
float(done))
            loss = agent.train_step()
            state = next_state

            total_reward += reward
            total_loss += loss

            if done:
                break

        reward_history.append(total_reward)
        loss_history.append(total_loss)

        agent.update_epsilon()
        agent.update_target()
        epoch_tqdm.desc = f"Reward: {total_reward}, Loss:
{total_loss}"

    return reward_history, loss_history

def plot_graphics(results, labels, main_title):
    fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(21,
6))
```

```
    fig.tight_layout(pad=5.0)
    fig.suptitle(main_title, fontsize=14)

    ax1.set_title("Reward")
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Reward')
    ax1.grid(True)

    ax2.set_title("Loss")
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Loss')
    ax2.grid(True)

    for (rewards, losses), label in zip(results, labels):
        ax1.plot(rewards, label=label)
        ax2.plot(losses, label=label)

    ax1.legend()
    ax2.legend()

    plt.show()

def experiment_models(env):
    labels = ["QNetwork1", "QNetwork2", "QNetwork3"]
    agents = [DQNAgent(QNetwork1, env), DQNAgent(QNetwork2, env),
DQNAgent(QNetwork3, env)]
    results = []

    for agent, label in zip(agents, labels):
        print(f"{label} model train")
        results.append(train(agent, env, 300, 100))

    plot_graphics(results, labels, "Network")

def experiment_gamma_eps_decay(env):
    gammas = [0.99, 0.8, 0.6]
    labels = []
    results = []
    for gamma in gammas:
        label = f"gamma = {gamma}"
```

```
        labels.append(label)
        agent = DQNAgent(QNetwork1, env, Params(gamma=gamma))

        print(f"{label} model train")
        results.append(train(agent, env, 300, 100))

    plot_graphics(results, labels, "Gamma")


    eps_decays = [0.95, 0.8, 0.6]
    labels = []
    results = []
    for eps_decay in eps_decays:
        label = f"eps_decay = {eps_decay}"
        labels.append(label)
        agent = DQNAgent(QNetwork1, env,
Params(eps_decay=eps_decay))

        print(f"{label} model train")
        results.append(train(agent, env, 300, 100))

    plot_graphics(results, labels, "Epsilon decay")


def experiment_eps_start(env):
    eps_starts = [0.9, 0.8, 0.6, 0.3]
    labels = []
    results = []
    for eps_start in eps_starts:
        label = f"eps_start = {eps_start}"
        labels.append(label)
        agent = DQNAgent(QNetwork1, env,
Params(eps_start=eps_start))

        print(f"{label} model train")
        results.append(train(agent, env, 300, 100))

    plot_graphics(results, labels, "Epsilon start")

env = gym.make("CartPole-v1")
```

```
experiment_models(env)
experiment_gamma_eps_decay(env)
experiment_eps_start(env)

def visualize_agent_performance(agent, env_name, model_name):
    env = gym.make(env_name, render_mode="rgb_array")
    env = RecordVideo(env,
                      video_folder="./videos",
                      name_prefix=model_name,
                      video_length=7000,
                      episode_trigger=lambda x: True)

    state, _ = env.reset()
    done = False
    total_reward = 0

    while not done:
        action = agent.select_action(state)
        state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        total_reward += reward

        if total_reward > env.spec.reward_threshold:
           print(f"Total Reward: {total_reward}")
           break

    env.close()

agent = DQNAgent(QNetwork1, env)
res = train(agent, env, n_epochs=300, steps=200)
plot_graphics([res], ['DQN'], "CartPole-v1")
visualize_agent_performance(agent, 'CartPole-v1', 'dqn')
```