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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Обучение с подкреплением»

Тема: Реализация DQN для среды CartPole-v1

Студент гр. 0310 _____ Афанасьев Н. С.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025

Цель работы.

Реализация DQN для среды CartPole-v1

Задание.

Окружение: Cartpole

Задания для эксперимента:

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

Теоретические положения.

DQN – это алгоритм глубокого обучения с подкреплением, сочетающий Q-обучение с глубокой нейронной сетью для аппроксимации Q-функции. Он решает проблему масштабируемости классического Q-обучения, позволяя работать с высокоразмерными пространствами состояний.

Основные параметры:

Gamma (γ): Коэффициент дисконтирования (обычно 0.99). Определяет, насколько агент учитывает будущие награды.

Epsilon (ε): Параметр ε-жадной стратегии. Начинается с ε_start = 1.0 (полностью случайные действия), затем уменьшается по epsilon_decay (например, 0.9995 за шаг) до ε_min (например, 0.01).

Сначала инициализируются две идентичные нейронные сети – основная (Q) и целевая (Q') — и буфер воспроизведения для хранения переходов (состояние, действие, награда, следующее состояние, флаг завершения). На каждом шаге агент выбирает действие с использованием ε-жадной стратегии: с вероятностью ε действие выбирается случайно, иначе – жадно, как argmax Q(s, a). После выполнения действия переход сохраняется в буфер, из которого затем случайно выбирается мини-батч для обучения: целевые Q-значения вычисляются как r + γ * max Q'(s', a'), а основная сеть оптимизируется методом

градиентного спуска. Периодически веса целевой сети обновляются весами основной, что стабилизирует обучение. Этот процесс повторяется, пока агент не достигнет требуемой производительности, используя experience replay для декорреляции данных и target network для устойчивости целевых значений.

Выполнение работы.

Реализация алгоритма

Основные компоненты кода включают класс ReplayBuffer, который реализует буфер воспроизведения для хранения переходов (state, action, reward, next_state, done) и их выборки для обучения. Класс QNetwork определяет архитектуру нейронной сети с заданными слоями и функцией активации ReLU. Класс DQNAgent объединяет все компоненты алгоритма DQN: две нейронные сети (основную и целевую), оптимизатор Adam, буфер воспроизведения и методы для выбора действия, обучения и обновления параметров.

Функция train реализует основной цикл обучения, в котором агент взаимодействует со средой, сохраняет переходы в буфер, обучает нейронную сеть и обновляет значения epsilon и целевую сеть. Функции plot_results и plot_results_grouped используются для визуализации результатов обучения в виде графиков награды и потерь.

Изменение архитектуры нейросети

Алгоритм был запущен для разных архитектур нейронной сети (рис. 1).

Сравниваемые архитектуры:

- *Default* – [64, 64] – Два скрытых слоя по 64 нейрона каждый
- *Large* – [256, 128, 64] – Три скрытых слоя с уменьшающейся размерностью: 256 → 128 → 64
- *Small* – [32] – Всего один скрытый слой из 32 нейронов
- *Deep* – [64, 64, 64, 32] – Четыре скрытых слоя: три по 64 нейрона и один с 32 нейронами.

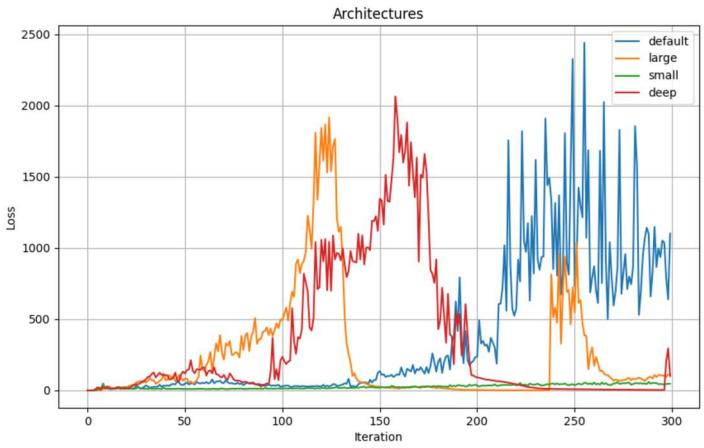

Рисунок 1 – Потери для разных архитектур сети

По результатам можно видеть, что значения ошибок варьируются от 0 до 2000, что что указывает на нестабильность обучения в некоторых архитектурах.

Наилучший результат у архитектуры small – обучение стабильное, без скачков.

Архитектуры deep и large имеют большие скачки, но за 300 итераций достигают схожего результата.

Архитектура default показывает наихудший результат – ближе к концу начинаются сильные скачки, количество итераций недостаточно для получения должного результата.

Изменение gamma и epsilon_decay

Алгоритм был запущен для разных значений gamma и epsilon_decay (рис. 2).

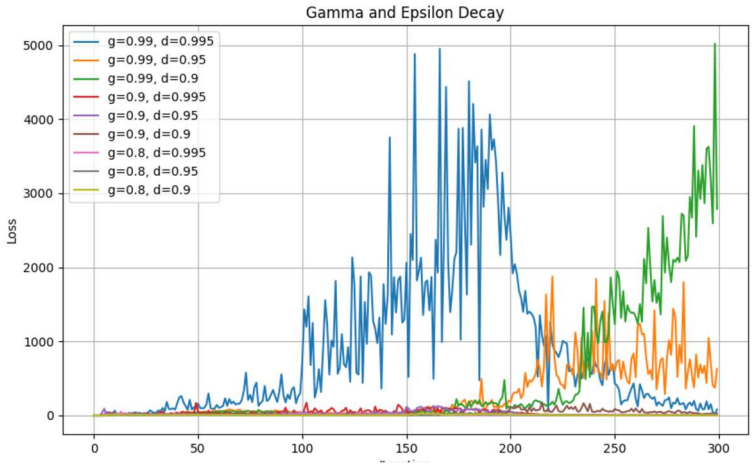

Рисунок 2 – Потери для разных gamma и epsilon_decay

По результатам можно видеть, что почти во всех случаях результат по завершении 300 итераций положительный.

При высоком значении gamma (g=0.99) можно видеть большие скачки, достигающих значения 5000. Более высокие значения epsilon_decay, однако, помогают быстрее преодолеть этот промежуток, так как агент начинает меньше полагаться на случайные действия.

При низких значениях gamma агент будет недооценивать будущие награды, делая обучение менее качественным, поэтому сильно опускать это значение тоже не стоит.

Изменение начального значение epsilon

Алгоритм был запущен для разных значений epsilon (рис. 3).

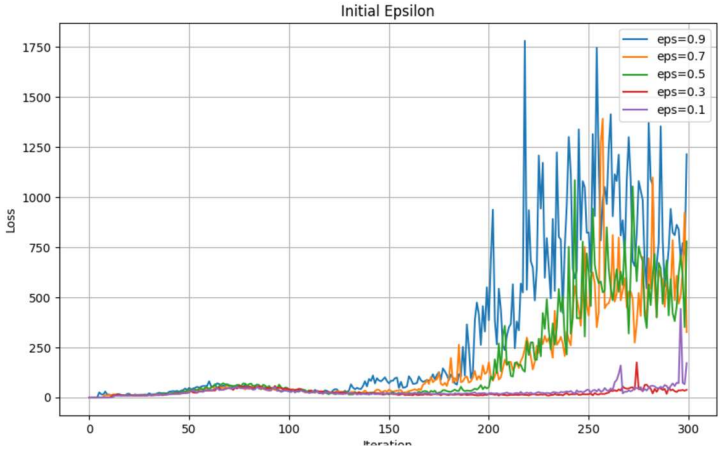

Рисунок 3 – Потери для разных epsilon

По результатам можно видеть, что при более низких значениях epsilon скачки ошибок становится меньше, так как агент быстрее начинает использовать выученную стратегию вместо случайного выбора.

Визуально можно поделить все значения на три категории: при eps=0.9 скачки наиболее интенсивные и продолжительные, при eps=0.5 и eps=0.7 скачки становятся меньше, при eps=0.1 и eps=0.3 скачков почти нету.

Разработанный программный код см. в приложении А.

Выводы.

Был изучен на практике алгоритм DQN. Алгоритм был реализован для среды CartPole-v1, также был проведён анализ работы алгоритма взависимости от различных входных параметров.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py:

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

os.makedirs("plots", exist_ok=True)

class ReplayBuffer:
    def __init__(self, capacity=10000):
        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(np.array(state), dtype=torch.float32),
            torch.tensor(action, dtype=torch.long),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(np.array(next_state), dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32),
        )

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

class QNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, layers):
        super().__init__()
        net = []
        last_dim = input_dim
        for l in layers:
            net.append(nn.Linear(last_dim, l))
            net.append(nn.ReLU())
            last_dim = l
        net.append(nn.Linear(last_dim, output_dim))
        self.model = nn.Sequential(*net)

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

class DQNAgent:
    def __init__(self, state_dim, action_dim, layer_cfg, gamma, epsilon,
epsilon_decay):
        self.q_net = QNetwork(state_dim, action_dim, layer_cfg)
        self.target_net = QNetwork(state_dim, action_dim, layer_cfg)
        self.target_net.load_state_dict(self.q_net.state_dict())
        self.optimizer = optim.Adam(self.q_net.parameters(), lr=1e-4)
```

```
        self.buffer = ReplayBuffer()
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        self.q_net.to(self.device)
        self.target_net.to(self.device)

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, 1)
        state_tensor = torch.tensor(state,
dtype=torch.float32).to(self.device)
        with torch.no_grad():
            return self.q_net(state_tensor).argmax().item()

    def train_step(self):
        if len(self.buffer) < 128:
            return 0
        s, a, r, s2, d = self.buffer.sample(128)
        s, a, r, s2, d = s.to(self.device), a.to(self.device),
r.to(self.device), s2.to(self.device), d.to(self.device)

        q_vals = self.q_net(s).gather(1, a.unsqueeze(1)).squeeze(1)
        with torch.no_grad():
            target = r + self.gamma * self.target_net(s2).max(1)[0] * (1 - d)
        loss = nn.MSELoss()(q_vals, target)

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

    def update_epsilon(self):
        self.epsilon = max(self.epsilon * self.epsilon_decay, 0.05)

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

def train(agent, env, episodes, steps):
    reward_history = []
    loss_history = []
    for ep in trange(episodes, desc="Эпизоды"):
        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 or truncated:
                break
        reward_history.append(total_reward)
        loss_history.append(total_loss)
        agent.update_epsilon()
        agent.update_target()
    return reward_history, loss_history
```

```
def plot_results(results, title, filename):
    for metric, idx in [('reward', 0), ('loss', 1)]:
        plt.figure(figsize=(10, 6))
        for label, data in results.items():
            plt.plot(data[idx], label=label)
        plt.title(f"{title}")
        plt.xlabel("Iteration")
        plt.ylabel(metric.capitalize())
        plt.legend()
        plt.grid()
        plt.savefig(f"plots/experiment_{filename}_{metric}.png")
        plt.close()

params = {
    "gamma": 0.99,
    "epsilon": 0.9,
    "epsilon_decay": 0.955,
    "epsilon_min": 0.05,
    "num_steps": 200,
    "num_episodes": 300,
}

def experiment_architectures():
    env = gym.make("CartPole-v1")
    results = {}
    for name, layers in {
        "default": [64, 64],
        "large": [256, 128, 64],
        "small": [32],
        "deep": [64, 64, 64, 32]
    }.items():
        agent = DQNAgent(4, 2, layers, params["gamma"], params["epsilon"],
params["epsilon_decay"])
        rewards, losses = train(agent, env, params["num_episodes"],
params["num_steps"])
        results[name] = (rewards, losses)
    return results

def experiment_gamma_decay():
    env = gym.make("CartPole-v1")
    results = {}
    for gamma in [0.99, 0.9, 0.8]:
        for decay in [0.995, 0.95, 0.9]:
            label = f"g={gamma}, d={decay}"
            agent = DQNAgent(4, 2, [64, 64], gamma, params["epsilon"], decay)
            rewards, losses = train(agent, env, params["num_episodes"],
params["num_steps"])
            results[label] = (rewards, losses)
    return results

def experiment_epsilons():
    env = gym.make("CartPole-v1")
    results = {}
    for eps in [0.9, 0.7, 0.5, 0.3, 0.1]:
        agent = DQNAgent(4, 2, [64, 64], params["gamma"], eps,
params["epsilon_decay"])
        rewards, losses = train(agent, env, params["num_episodes"],
params["num_steps"])
        results[f"eps={eps}"] = (rewards, losses)
    return results

if __name__ == "__main__":
    print("Running architecture experiment...")
```

```
r_arch = experiment_architectures()
plot_results(r_arch, "Architectures", "architectures")

print("Running gamma & decay experiment...")
r_gd = experiment_gamma_decay()
plot_results(r_gd, "Gamma and Epsilon Decay", "gamma_decay")

print("Running epsilon init experiment...")
r_eps = experiment_epsilons()
plot_results(r_eps, "Initial Epsilon", "epsilon_init")
```