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 _____ Аксенов И.В.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025

СОДЕРЖАНИЕ

Цель работы

Реализовать алгоритм DQN для обучения агента в среде CartPole.

Задание

1. Реализовать алгоритм DQN;
2. Измените архитектуру нейросети (например, добавьте слои);
3. Попробуйте разные значения γ и ε_{decay};
4. Проведите исследование как изначальное значение ε влияет на скорость обучения.

Выполнение работы

3.1. Реализовать алгоритм DQN

Алгоритм DQN реализован с использованием библиотеки TensorFlow на языке Python.

Полный код представлен в приложении (Приложение А)

Ниже представлены стандартные параметры, которые использовались для обучения агентов.

```
default_params = {
    "gamma": 0.99,
    "epsilon": 0.9,
    "epsilon_decay": 0.955,
    "epsilon_min": 0.05,
    "batch_size": 128,
    "num_steps": 200,
    "num_episodes": 300,
    "lr": 1e-4,
}
```

На таких исходных данных получились результаты, представленные на

рисунке (Рисунок 3.1).

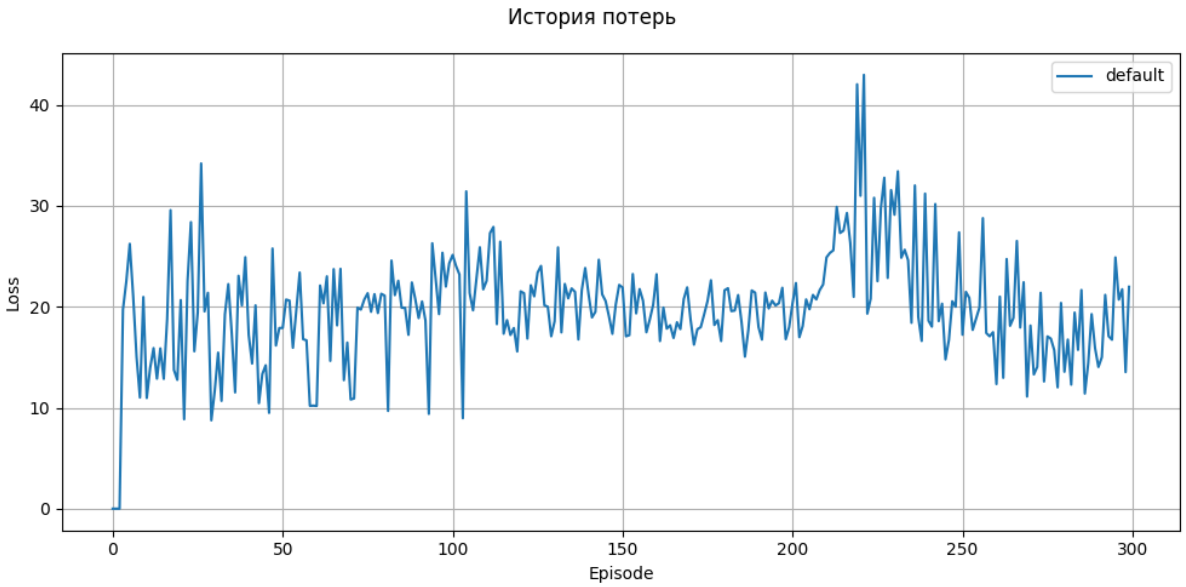

Рисунок 3.1 – Результаты обучения агента на стандартных параметрах

3.2. Измените архитектуру нейросети (например, добавьте слои)

Всего использовалось 4 различных набора слоев: ”default”, ”large”, ”small”, ”deep”.

```
"default": [
    nn.Linear(4, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 2),
],
```

```
"large": [
    nn.Linear(4, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 2),
],
```

```
"small": [
    nn.Linear(4, 32),
    nn.ReLU(),
    nn.Linear(32, 2),
],
```

```
"deep": [
    nn.Linear(4, 64),
    nn.ReLU(),
    nn.Linear(64, 64),
    nn.ReLU(),
    nn.Linear(64, 64),
    nn.ReLU(),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 2),
],
```

Результат обучения на каждом наборе представлен на рисунке (Рисунок 3.2)

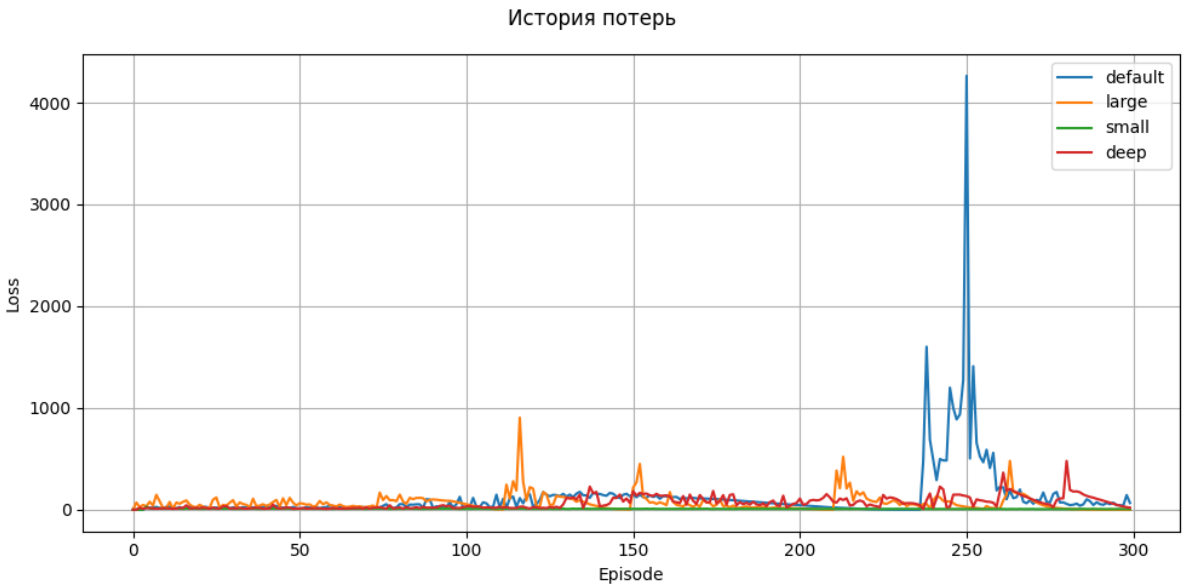

Рисунок 3.2 – Результат обучения на наборе слоев из группы ”default”

По полученным данным можно сделать следующие выводы:

- Сильной разницы между слоями в общем плане не наблюдается;
- Слои из группы ”small” в результате дали наиболее стабильный результат;

- Стандартные слои из группы ”default” в определенный момент начали давать высокие потери, что может говорить о том, что во время обучения мог произойти ряд ”выбросов” и в результате точность упала. Однако далее агент адаптировался к новым данным и стал снова выдавать стабильный результат;
- Слои ”deep” и ”large” в целом ведут себя одинаково. Обладают рядом скачков, однако в общем плане ведут себя стабильно.

3.3. Попробуйте разные значения γ и ε_{decay}

В качестве значений для γ выступал набор значений:

$$\gamma = \{0.99, 0.9, 0.85, 0.8, 0.75, 0.70\}$$

В качестве значений для ε_{decay} выступал набор значений:

$$\varepsilon_{decay} = \{0.995, 0.95, 0.9, 0.85, 0.80, 0.70\}$$

На рисунке (Рисунок 3.3) представлены графики с разными значениями γ и ε_{decay}.

Сверху вниз указаны графики для значений параметра в таком же порядке (верхний рисунок отображает результаты для $\gamma = 0.99$, последний для $\gamma = 0.70$).

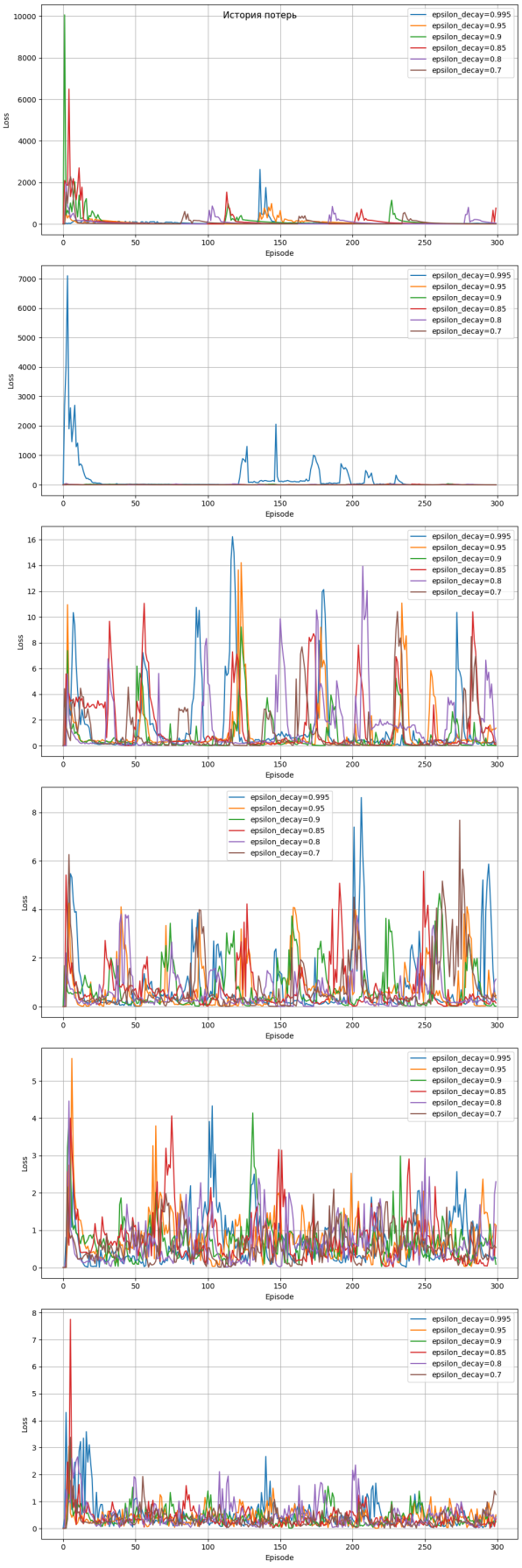

Рисунок 3.3 – Результаты измерения потерь от разных значений γ и ε_{decay}

В результате можно сделать следующие выводы:

- Слишком большое значение ε_{decay} приводит к возможным ”выбросам” и увеличению потерь;
- Уменьшение параметра γ приводит к ухудшеню точностей моделей.

3.4. Проведите исследование как изначальное значение ε влияет на скорость обучения

Параметры для данного исследования:

$$\varepsilon = \{0.9, 0.7, 0.5, 0.3, 0.2, 0.1, 0.05, 0.01, 0.05\}$$

На рисунке (Рисунок 3.4) представлены результаты эксперимента.

Рисунок 3.4 – Результаты эксперимента по изменению параметра ε

На основе полученных результатов можно сделать следующие выводы:

- Более стабильные модели получались при большем значении параметра ε;
- Остальные результаты в целом схожи по своей динамике и только в случае $\varepsilon = 0.9$ не было резких скачков и в целом отличается от остальных результатов.

Выводы

В ходе выполнения лабораторной работы был реализован алгоритм DQN и исследовано влияние различных параметров и архитектур нейросети на процесс обучения агента в среде CartPole.

1. Реализация алгоритма показала, что агент способен обучаться стабильно, однако в некоторых случаях наблюдаются скачки потерь, которые связаны с выбросами или особенностями обучения;
2. Наиболее стабильные результаты продемонстрировала архитектура ”small”, тогда как ”default” и ”deep” оказались менее устойчивыми;
3. Эксперименты с параметрами γ и ε_{decay} показали, что слишком высокое значение ε_{decay} вызывает выбросы, а низкое значение γ ухудшает точность модели;
4. Исследование начального значения ε показало, что большие значения ($\varepsilon = 0.9$) обеспечивают более стабильное обучение, тогда как малые значения приводят к резким скачкам потерь.

Таким образом правильный подбор гипермараметров и параметров для обучения очень важны для получения более точных моделей.

ПРИЛОЖЕНИЕ А

```
import os
import random
from collections import deque
from datetime import datetime

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

layers_pack = {
    "default": [
        nn.Linear(4, 128),
        nn.ReLU(),
        nn.Linear(128, 64),
        nn.ReLU(),
        nn.Linear(64, 2),
    ],
    "large": [
        nn.Linear(4, 256),
        nn.ReLU(),
        nn.Linear(256, 128),
        nn.ReLU(),
        nn.Linear(128, 64),
        nn.ReLU(),
        nn.Linear(64, 2),
    ],
    "small": [
        nn.Linear(4, 32),
        nn.ReLU(),
        nn.Linear(32, 2),
    ],
    "deep": [
        nn.Linear(4, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Linear(32, 2),
    ],
```

```
}

# Default parameters
default_params = {
    "gamma": 0.99,
    "epsilon": 0.9,
    "epsilon_decay": 0.955,
    "epsilon_min": 0.05,
    "batch_size": 128,
    "num_steps": 200,
    "num_episodes": 300,
    "lr": 1e-4,
}


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):
        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, layers: list[nn.Module]):
        super(QNetwork, self).__init__()
        self.net = nn.Sequential(*layers)

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


class DQNAgent:
    def __init__(
        self,
```

```
        obs_size: int,
        n_actions: int,
        layers: list[nn.Module],
        gamma: float = 0.99,
        epsilon: float = 1.0,
        epsilon_decay: float = 0.955,
        epsilon_min: float = 0.01,
        batch_size: int = 64,
    ):
        self.device = torch.device("cuda" if torch.cuda.is_available() else
        ↪  "cpu")

        self.q_net = QNetwork(obs_size, n_actions, layers).to(self.device)
        self.net_target = QNetwork(obs_size, n_actions,
        ↪  layers).to(self.device)
        self.net_target.load_state_dict(self.q_net.state_dict())

        self.optimizer = optim.Adam(self.q_net.parameters(),
        ↪  lr=default_params["lr"])

        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.batch_size = batch_size

        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)
        rewards = torch.tensor(reward, dtype=torch.float32,
        ↪  device=self.device)
        next_states = torch.tensor(next_state, dtype=torch.float32,
        ↪  device=self.device)
        dones = torch.tensor(done, dtype=torch.float32, device=self.device)

        """ dqn update """

        q_values = self.q_net(states).gather(1,
        ↪  actions.unsqueeze(1)).squeeze(1)
        next_q_values = self.net_target(next_states).max(1)[0]
        target_q_values = rewards + self.gamma * next_q_values * (1 - dones)

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

        self.loss = loss.item()

        self.epsilon = max(self.epsilon * self.epsilon_decay,
        ↪  self.epsilon_min)

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


class Simulation:
    def __init__(
        self,
        env: Env,
        agent: DQNAgent,
        time: datetime,
        sim_name: str = "simulation",
        num_episodes: int = 1000,
        num_steps: int = 200,
    ):
        self.env = env

        self.agent = agent
        self.sim_name = sim_name
        self.time = time

        self.num_episodes = num_episodes
        self.num_steps = num_steps
```

```
        self.reward_history = []
        self.loss_history = []

    def train(self):
        for episode in tqdm(range(self.num_episodes), desc="Эпизод"):
            state, _ = self.env.reset()
            episode_reward: float = 0.0
            episode_loss: float = 0.0

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

                state = next_state
                episode_reward += float(reward)

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

                if done:
                    break

            self.reward_history.append(episode_reward)
            self.loss_history.append(episode_loss)

        self.agent.update_target()

    def run(self):
        self.train()


def plot_fig_multiple(sim_variations: dict, name, rows, cols, width, height):
    fig1, ax1 = plt.subplots(rows, cols, figsize=(width * cols, height *
    ↪  rows))
    fig2, ax2 = plt.subplots(rows, cols, figsize=(width * cols, height *
    ↪  rows))
    ax1 = ax1.ravel()
    ax2 = ax2.ravel()
    fig1.suptitle("История наград")
    fig2.suptitle("История потерь")

    i = 0
    for sim_variation, sim_results in sim_variations.items():
        ax1_small = ax1[i]
        ax2_small = ax2[i]
```

```
        for sim_name, sim_results in sim_results.items():
            print(sim_name)
            print(sim_results)

            ax1_small.plot(sim_results["reward"], label=f"{sim_name}")
            ax1_small.set_xlabel("Episode")
            ax1_small.set_ylabel("Reward")

            ax2_small.plot(sim_results["loss"], label=f"{sim_name}")
            ax2_small.set_xlabel("Episode")
            ax2_small.set_ylabel("Loss")

        ax1_small.legend()
        ax1_small.grid()
        ax2_small.legend()
        ax2_small.grid()

        i += 1

    fig1.tight_layout()
    fig2.tight_layout()
    os.makedirs("fig", exist_ok=True)

    fig1.savefig(f"fig/{name}_reward_history.png")
    fig2.savefig(f"fig/{name}_loss_history.png")

    # plt.show()


def plot_fig_single(sim_results, name, width=6, height=6):
    fig1, ax1 = plt.subplots(figsize=(width, height))
    fig2, ax2 = plt.subplots(figsize=(width, height))
    fig1.suptitle("История наград")
    fig2.suptitle("История потерь")

    for sim_name, sim_results in sim_results.items():
        print(sim_name)
        print(sim_results)

        ax1.plot(sim_results["reward"], label=f"{sim_name}")
        ax1.set_xlabel("Episode")
        ax1.set_ylabel("Reward")

        ax2.plot(sim_results["loss"], label=f"{sim_name}")
        ax2.set_xlabel("Episode")
        ax2.set_ylabel("Loss")

    ax1.legend()
    ax1.grid()
```

```
    ax2.legend()
    ax2.grid()

    fig1.tight_layout()
    fig2.tight_layout()
    fig1.savefig(f"fig/{name}_reward_history.png")
    fig2.savefig(f"fig/{name}_loss_history.png")

    # plt.show()


def plot_fig(sim_results, rows, cols, name, width=6, height=6):
    fig1, ax1 = plt.subplots(rows, cols, figsize=(width, height))
    fig2, ax2 = plt.subplots(rows, cols, figsize=(width, height))
    fig1.suptitle("История наград")
    fig2.suptitle("История потерь")

    # Делаем обращение по одному индексу
    ax1 = ax1.ravel()
    ax2 = ax2.ravel()

    i = 0
    for sim_name, sim_results in sim_results.items():
        print(sim_name)
        print(sim_results)

        ax1[i].plot(sim_results["reward"], "C0", label=f"{sim_name}")
        ax1[i].set_xlabel("Episode")
        ax1[i].set_ylabel("Reward")
        ax1[i].legend()
        ax1[i].grid()

        ax2[i].plot(
            sim_results["loss"],
            "C1",
            label=f"{sim_name}",
        )
        ax2[i].set_xlabel("Episode")
        ax2[i].set_ylabel("Loss")
        ax2[i].legend()
        ax2[i].grid()

        i += 1

    fig1.tight_layout()
    fig2.tight_layout()
    fig1.savefig(f"fig/{name}_reward_history.png")
    fig2.savefig(f"fig/{name}_loss_history.png")
```

```
    # plt.show()


def default_experiment():
    env = gym.make("CartPole-v1", render_mode="rgb_array")

    simulation_results = {}

    print("Default experiment")
    agent = DQNAgent(
        obs_size=4,
        n_actions=2,
        layers=layers_pack["default"],
        gamma=default_params["gamma"],
        epsilon=default_params["epsilon"],
        epsilon_decay=default_params["epsilon_decay"],
    )

    simulation = Simulation(
        env=env,
        agent=agent,
        sim_name="epsilon_experiment",
        time=datetime.now(),
        num_steps=default_params["num_steps"],
        num_episodes=default_params["num_episodes"],
    )

    simulation.run()

    simulation_results["default"] = {
        "reward": simulation.reward_history,
        "loss": simulation.loss_history,
    }

    plot_fig_single(simulation_results, name="default_experiment", width=10,
    ↪  height=5)


def epsilon_experiment():
    epsilons = [0.9, 0.7, 0.5, 0.3, 0.2, 0.1, 0.05, 0.01, 0.05]
    env = gym.make("CartPole-v1", render_mode="rgb_array")

    simulation_results = {}
    total_experiments = len(epsilons)
    current_experiment = 1

    for eps in epsilons:
        print(
```

```
            f"Epsilon experiment: epsilon={eps}
            ↪  [{current_experiment}/{total_experiments}]"
        )
        agent = DQNAgent(
            obs_size=4,
            n_actions=2,
            layers=layers_pack["default"],
            gamma=default_params["gamma"],
            epsilon=eps,
            epsilon_decay=default_params["epsilon_decay"],
        )

        simulation = Simulation(
            env=env,
            agent=agent,
            sim_name="epsilon_experiment",
            time=datetime.now(),
            num_steps=default_params["num_steps"],
            num_episodes=default_params["num_episodes"],
        )
        simulation.run()

        simulation_results[f"epsilon={eps}"] = {
            "reward": simulation.reward_history,
            "loss": simulation.loss_history,
        }

        current_experiment += 1

    # plot_fig(simulation_results, 2, 3, name="epsilon_experiment")
    plot_fig_single(simulation_results, name="epsilon_experiment", width=10,
    ↪  height=5)


def gamma_decay_experiment():
    gammas = [0.99, 0.9, 0.85, 0.8, 0.75, 0.70]
    epsilon_decays = [0.995, 0.95, 0.9, 0.85, 0.80, 0.70]
    env = gym.make("CartPole-v1", render_mode="rgb_array")

    simulation_results_multiple = {}
    total_experiments = len(gammas) * len(epsilon_decays)
    current_experiment = 1

    for gamma in gammas:
        simulation_results = {}

        for epsilon_decay in epsilon_decays:
            print(
```

```
                f"Gamma decay experiment: gamma={gamma},
                ↪  epsilon_decay={epsilon_decay}
                ↪  [{current_experiment}/{total_experiments}]"
            )
            agent = DQNAgent(
                obs_size=4,
                n_actions=2,
                layers=layers_pack["default"],
                gamma=gamma,
                epsilon=default_params["epsilon"],
                epsilon_decay=epsilon_decay,
            )

            simulation = Simulation(
                env=env,
                agent=agent,
                sim_name="gamma_decay_experiment",
                time=datetime.now(),
                num_steps=default_params["num_steps"],
                num_episodes=default_params["num_episodes"],
            )

            simulation.run()

            simulation_results[f"epsilon_decay={epsilon_decay}"] = {
                "reward": simulation.reward_history,
                "loss": simulation.loss_history,
            }
            current_experiment += 1

        simulation_results_multiple[f"gamma_decay_experiment_gamma={gamma}"] =
        ↪  (
            simulation_results
        )

    plot_fig_multiple(
        simulation_results_multiple, "gamma_decay_experiment", 6, 1, 10, 5
    )


def architecture_experiment():
    env = gym.make("CartPole-v1", render_mode="rgb_array")
    simulation_results = {}

    current_experiment = 1
    total_experiments = len(layers_pack)

    for layers_name, layers in layers_pack.items():
        print(
```

```
            f"Architecture experiment: {layers_name}
            ↪  [{current_experiment}/{total_experiments}]"
        )

        agent = DQNAgent(
            obs_size=4,
            n_actions=2,
            layers=layers,
            gamma=default_params["gamma"],
            epsilon=default_params["epsilon"],
            epsilon_decay=default_params["epsilon_decay"],
        )

        simulation = Simulation(
            env=env,
            agent=agent,
            sim_name=layers_name,
            time=datetime.now(),
            num_steps=default_params["num_steps"],
            num_episodes=default_params["num_episodes"],
        )
        simulation.run()

        simulation_results[layers_name] = {
            "reward": simulation.reward_history,
            "loss": simulation.loss_history,
        }

        current_experiment += 1

    plot_fig_single(
        simulation_results, name="architecture_experiment", width=10, height=5
    )


if __name__ == "__main__":
    experiments = [
        default_experiment,
        architecture_experiment,
        gamma_decay_experiment,
        epsilon_experiment,
    ]

    for experiment in experiments:
        experiment()
```