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

КУРСОВАЯ РАБОТА

по дисциплине «Обучение с подкреплением»

Тема: Реализация DQN для среды CartPole-v1

Студент гр. 0310 _____ Нагибин И.С.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2025

Содержание

Цель работы

Реализовать алгоритм DQN для обучения агента в среде CartPole.

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.

Выполнение работы

Алгоритм DQN реализован с использованием библиотеки TensorFlow на языке Python.

Полный код представлен в приложении (Приложение А).

Конфигурации архитектур задаются переменной LAYER_ARCHITECTURES:

```
LAYER_ARCHITECTURES = {
    "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),
        ],
}
```

Исследование происходило с слующими параметрами:"gamma": 0.99, "epsilon": 0.9, "epsilon_decay": 0.955.

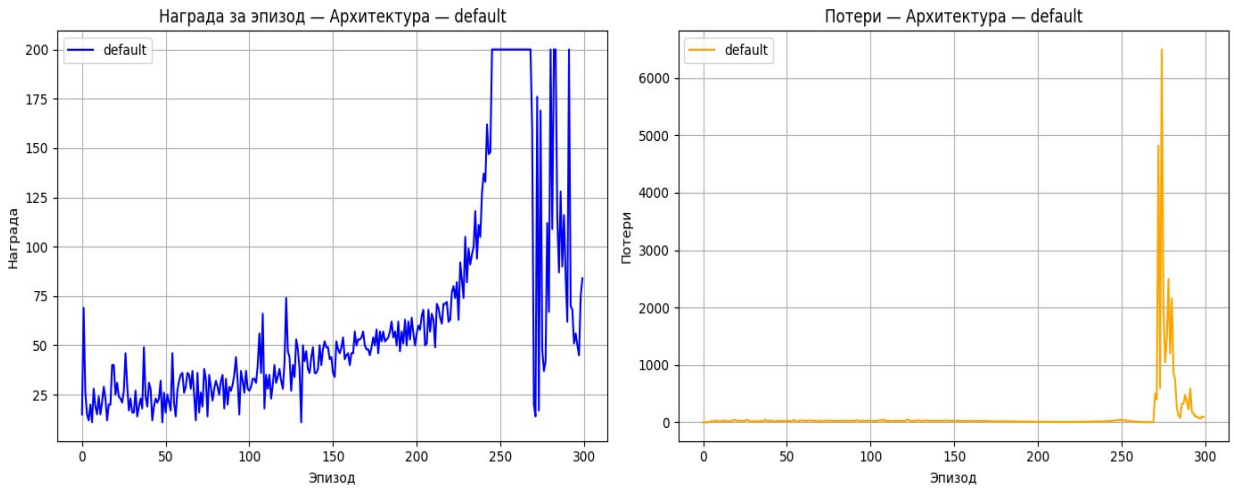

Рисунок 1 — График награды и потери (архитектура default)

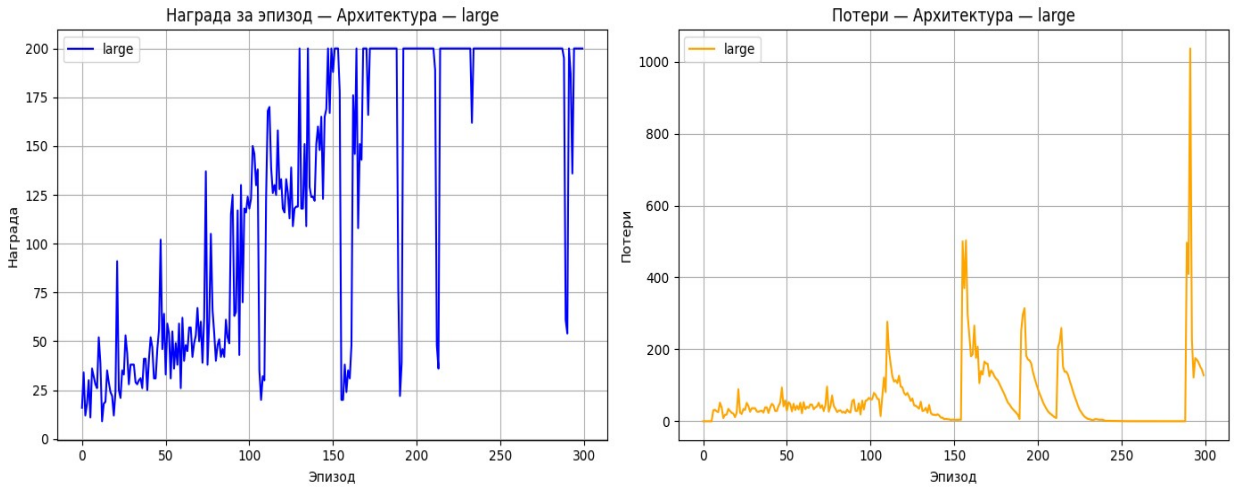

Рисунок 2 — График награды и потери (архитектура large)

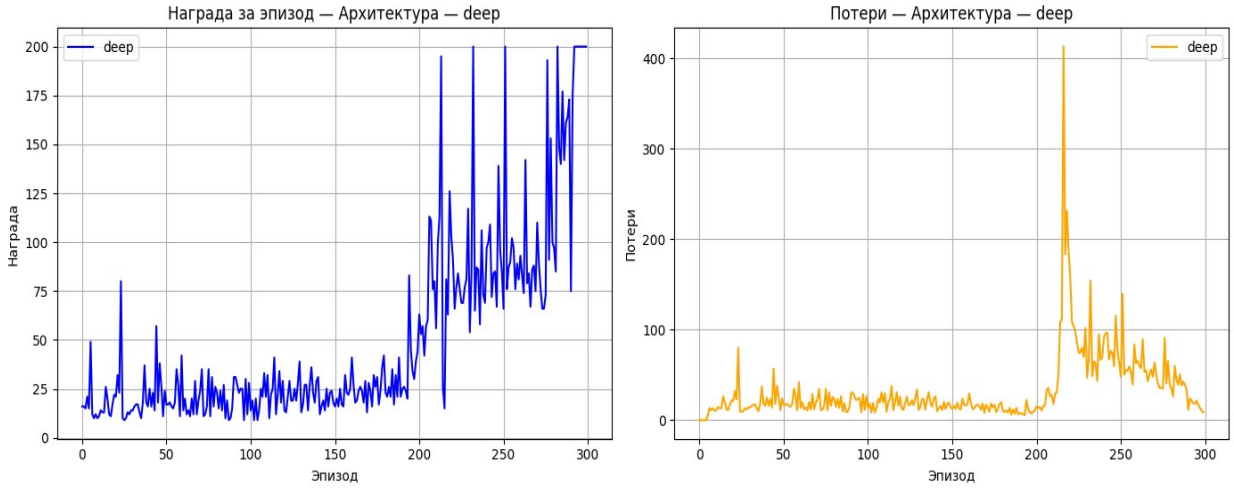

Рисунок 3 — График награды и потери (архитектура deep)

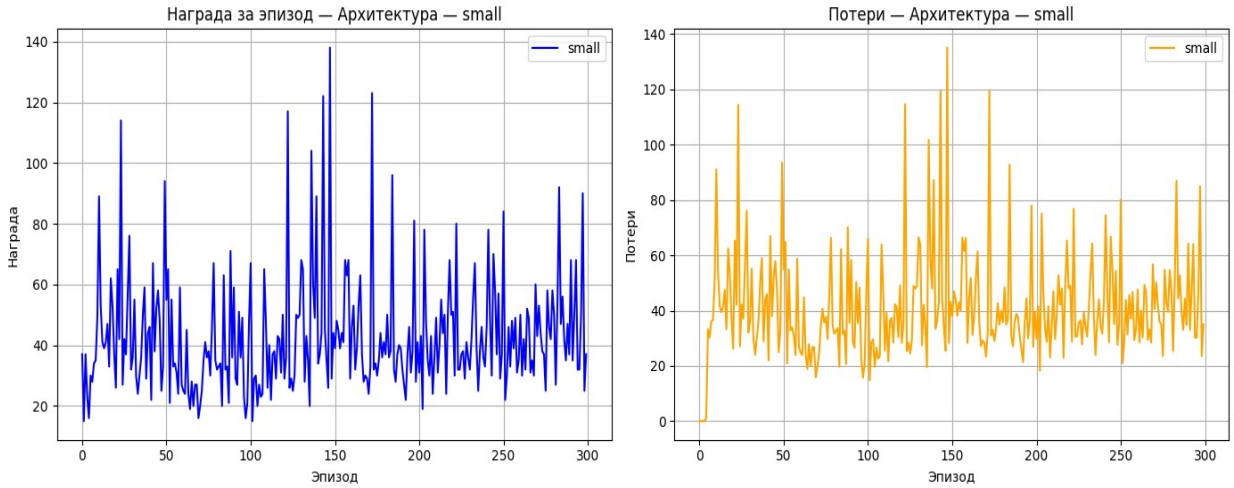

Рисунок 4 — График награды и потери (архитектура small)

Эксперименты с различными значениями gamma и decay происходили на архтектуре default.

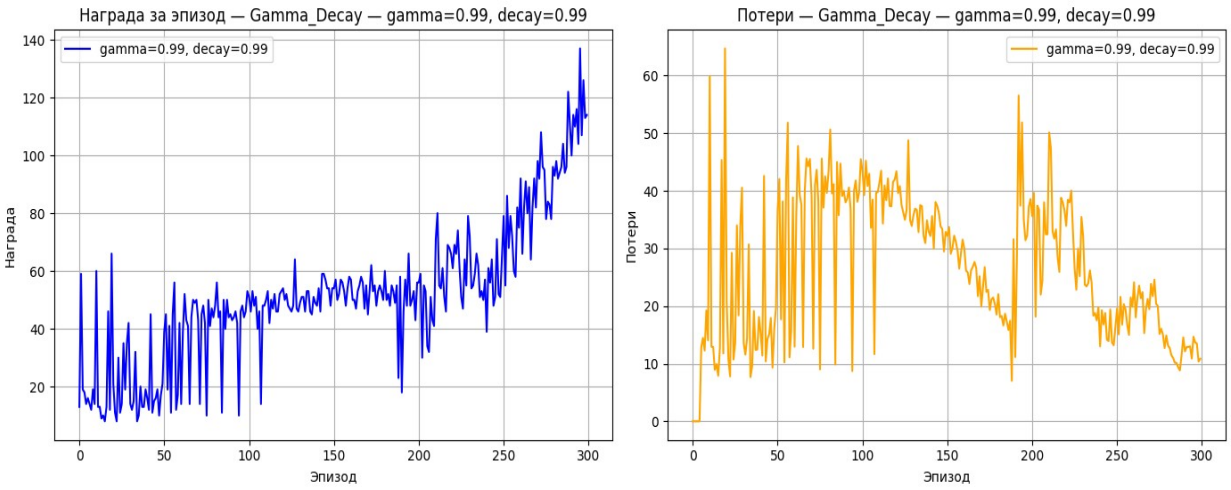

Рисунок 5 — График награды и потери (gamma = 0.99, decay = 0.99)

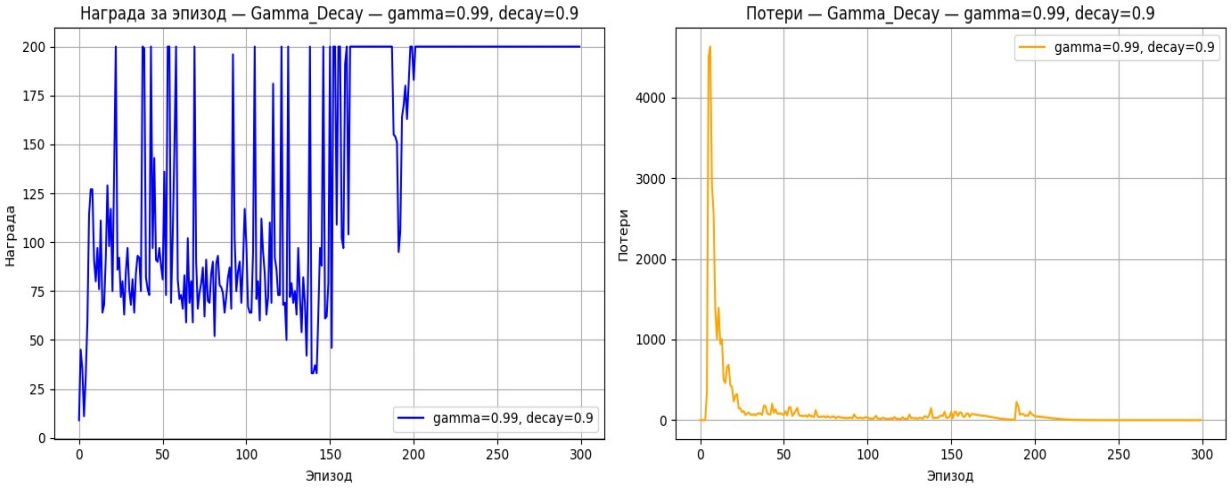

Рисунок 6 — График награды и потери (gamma = 0.99, decay = 0.9)

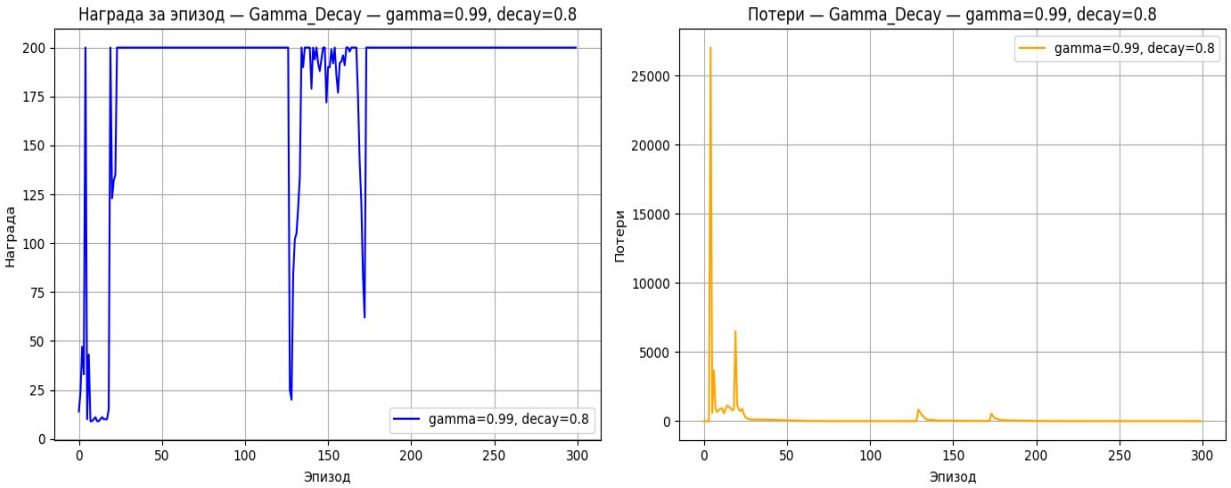

Рисунок 7 — График награды и потери (gamma = 0.99, decay = 0.8)

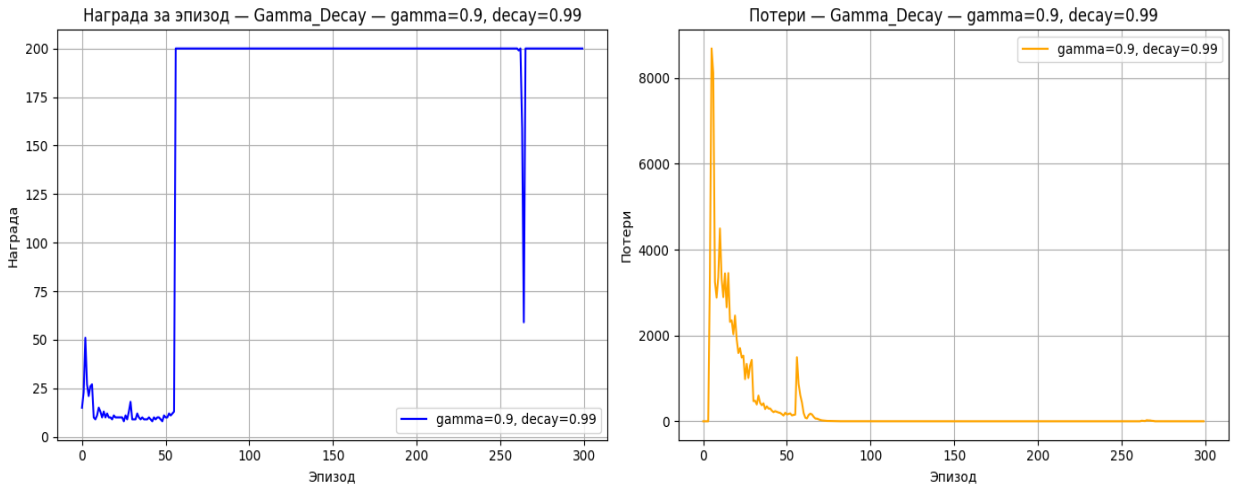

Рисунок 8 — График награды и потери (gamma = 0.9, decay = 0.99)

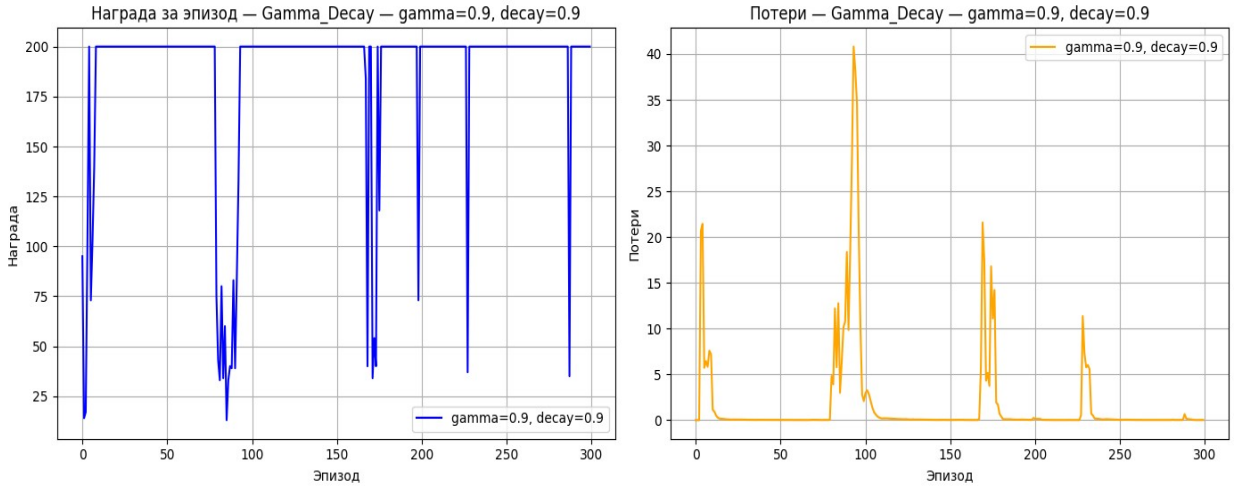

Рисунок 9 — График награды и потери (gamma = 0.9, decay = 0.9)

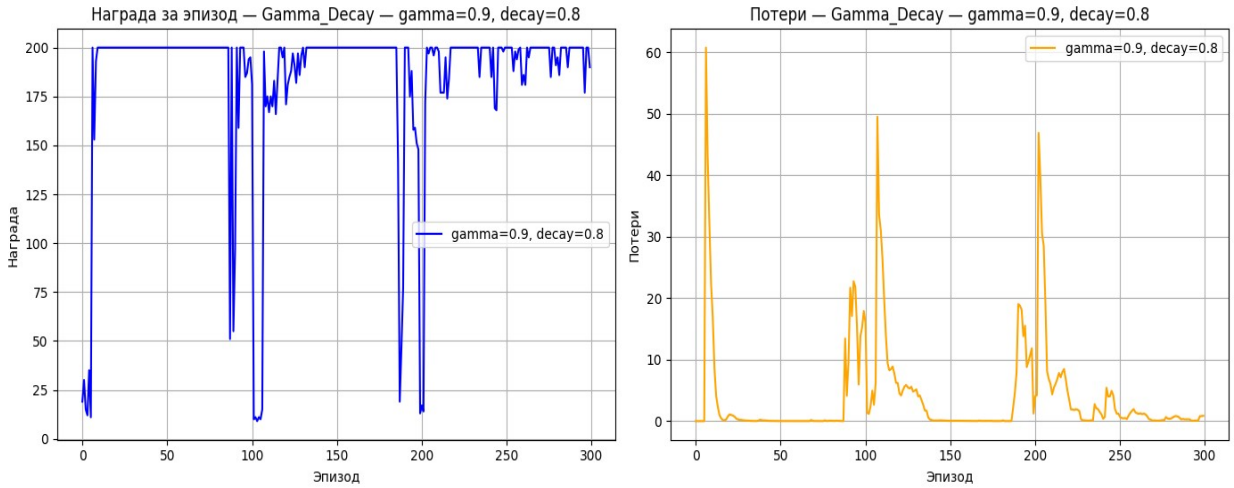

Рисунок 10 — График награды и потери (gamma = 0.9, decay = 0.8)

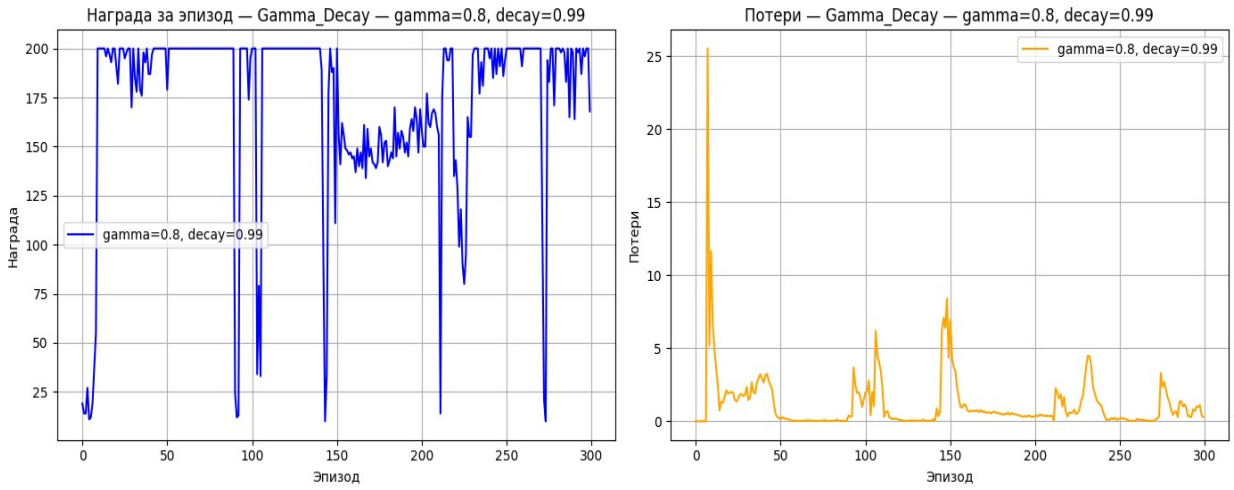

Рисунок 11 — График награды и потери (gamma = 0.8, decay = 0.99)

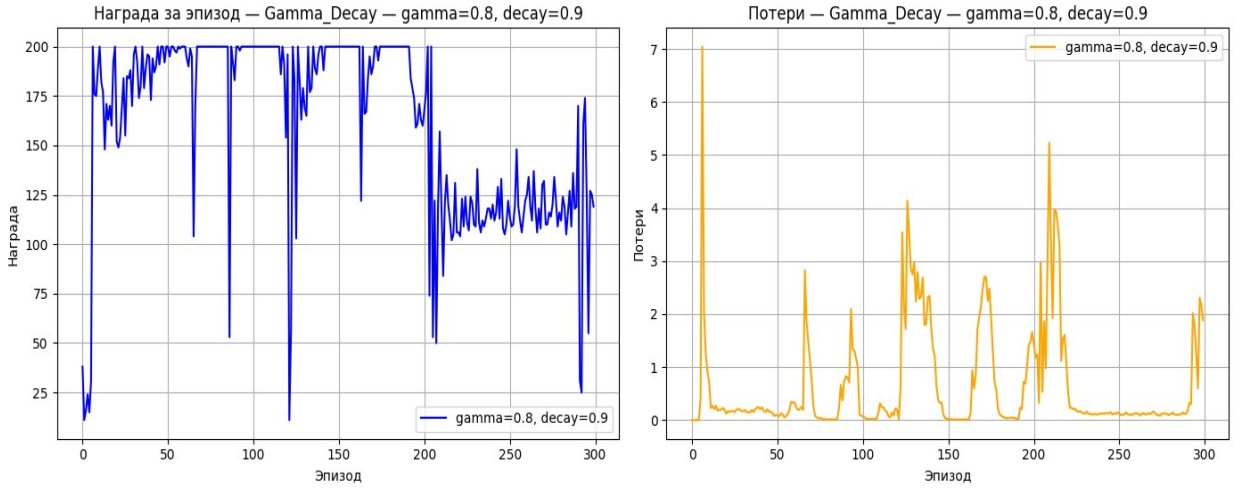

Рисунок 12 — График награды и потери (gamma = 0.8, decay = 0.9)

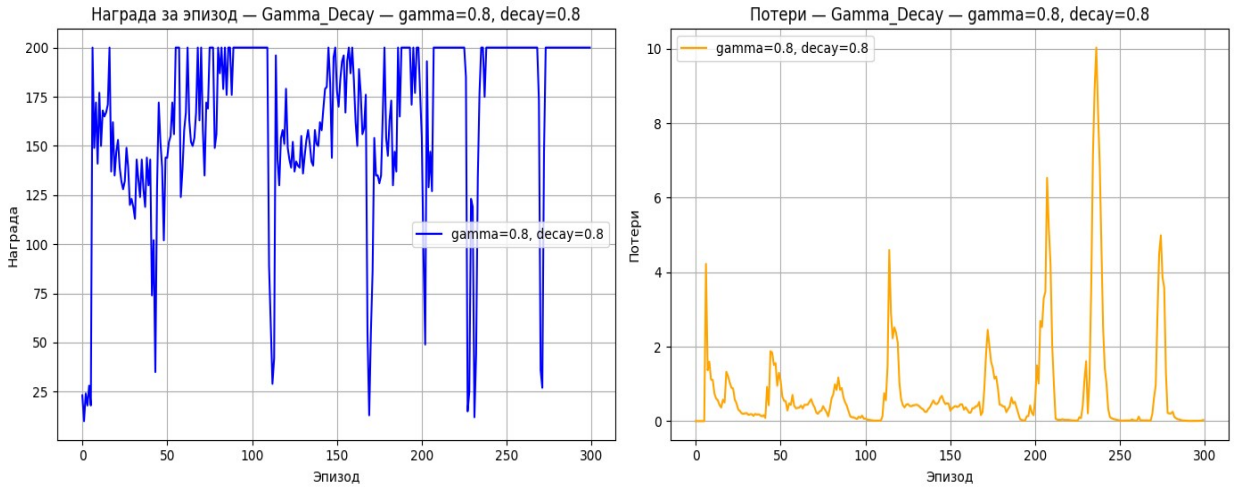

Рисунок 13 — График награды и потери (gamma = 0.8, decay = 0.8)

Эксперименты с различными значениями epsilon происходили на архтектуре default.

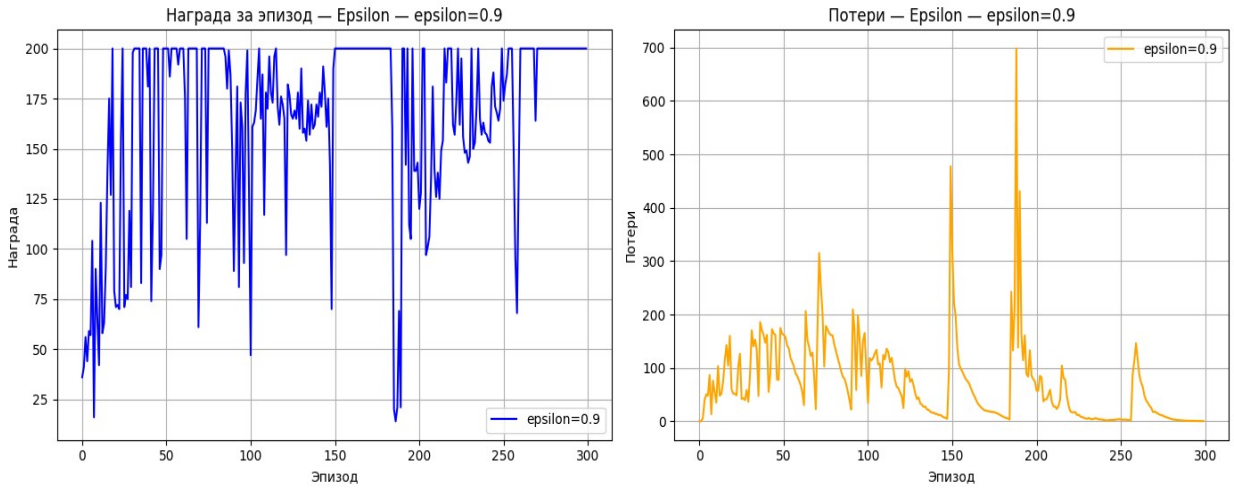

Рисунок 14 — График награды и потери (epsilon = 0.9)

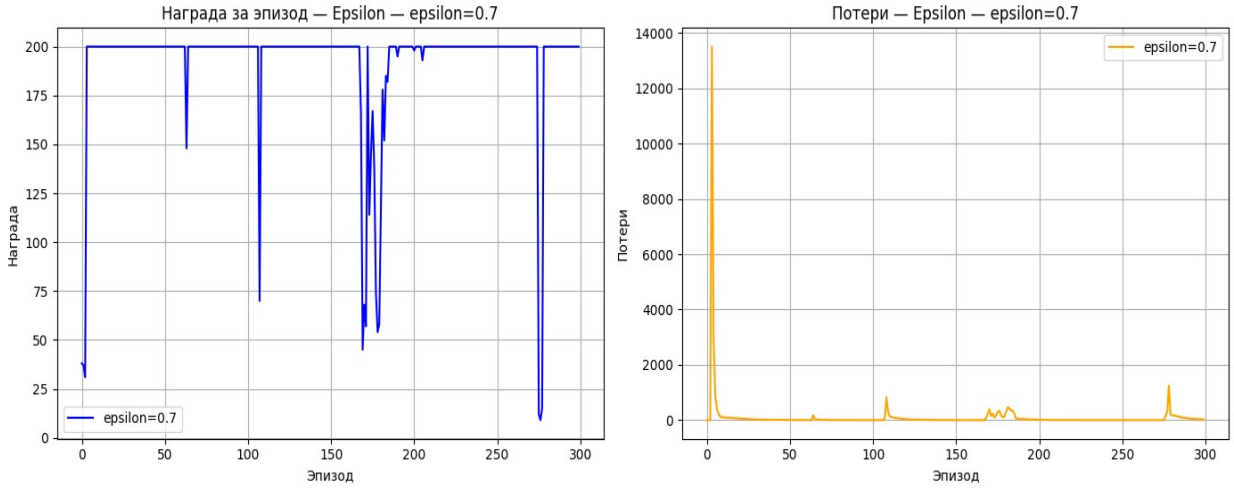

Рисунок 15 — График награды и потери (epsilon = 0.7)

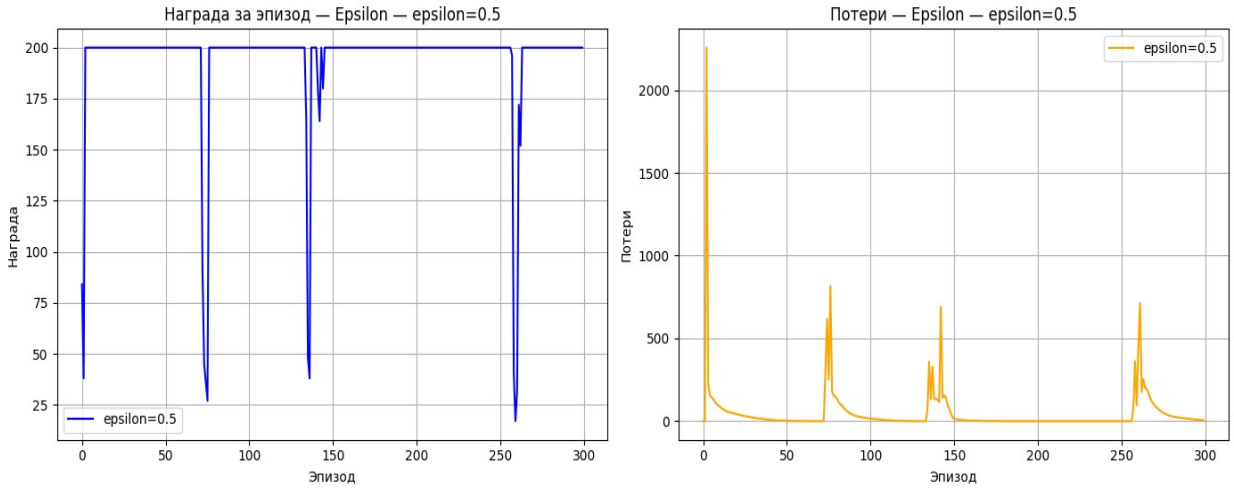

Рисунок 16 — График награды и потери (epsilon = 0.5)

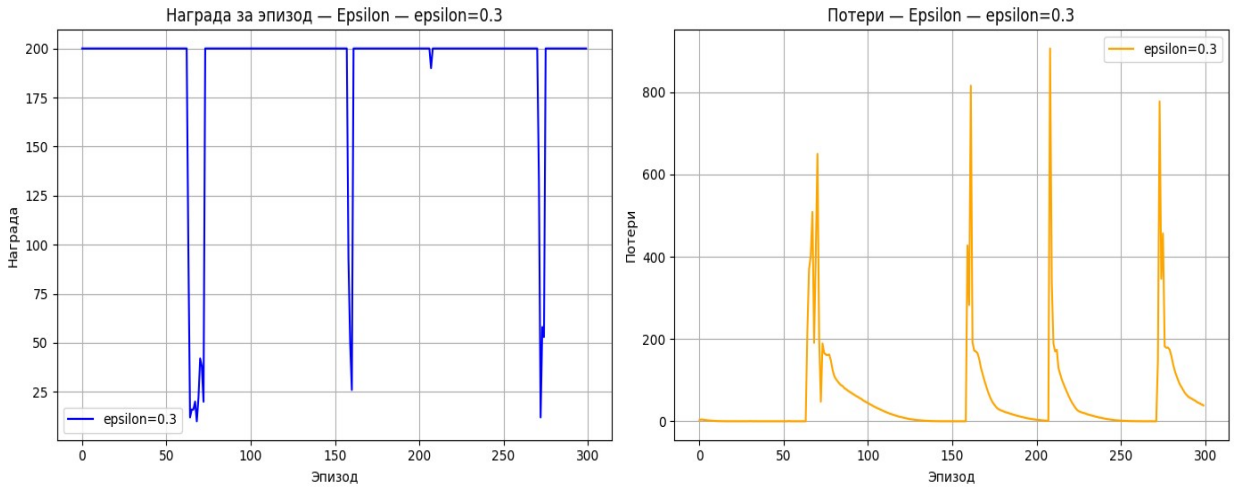

Рисунок 17 — График награды и потери (epsilon = 0.3)

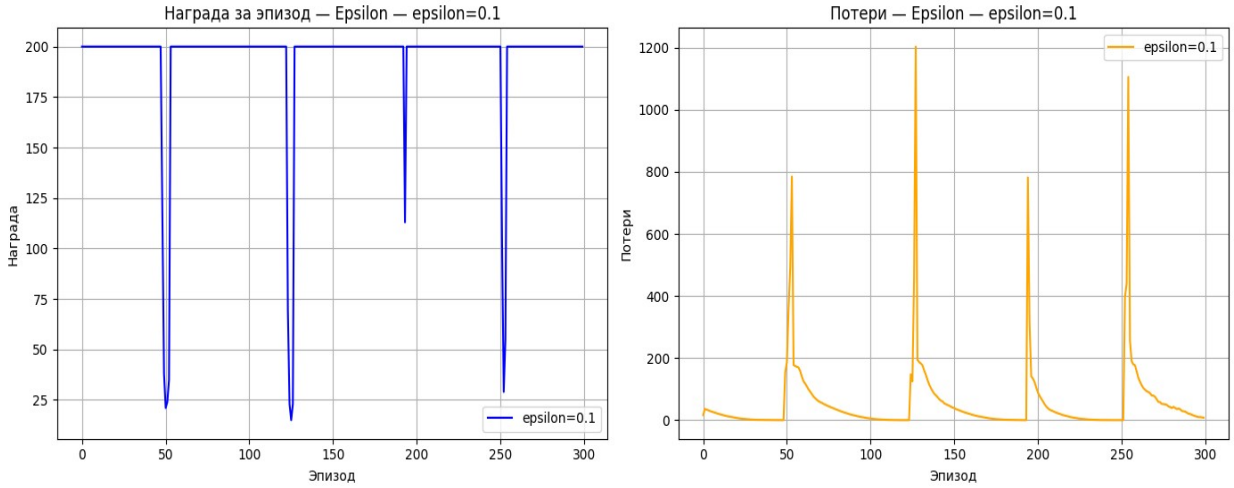

Рисунок 18 — График награды и потери (epsilon = 0.1)

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. Анализ архитектур**:**

- **Default**
    - График награды показывает стабильный рост до 200 (максимальной награды в CartPole-v1), что указывает на успешное обучение.
    - Потери стремятся к нулю, что говорит о стабильности обучения. Но в конце обучения потери резко возрастают. Скорее всего это связано с тем, что агент перестаёт исследовать и не успевает скорректировать свои оценки для редких состояний.
- **Large**
    - Награда достигает максимума быстрее, чем у default, но с большими колебаниями.
    - Потери больше чем у default, возможно, из-за избыточной сложности сети.
- **Small**
    - Награда и потери нестабильны и постоянно колеблются. Можно это связать с слишком узкими и малыми размерами сети.
- **Deep**
    - Награда долго не возрастает (примерно до 200 эпизода), но затем начинает резко расти до 200, после получения максимальной награды потери стабильно уменьшаются .

2.Коэффициент дисконтирования (gamma)

- **Gamma = 0.99** :
    - Агент акцентирует внимание на долгосрочной награде, что замедляет начальное обучение.
    - После достижения 200 баллов наблюдаются скачки потерь из-за редких состояний (например, почти падение шеста).
- **Gamma = 0.9** :
    - Агент быстрее достигает 200 баллов
- **Gamma = 0.8** :

- Агент быстрее всего достигает 200 баллов с минимальными колебаниями.

3. Скорость уменьшения исследования (epsilon_decay)

- **Epsilon_decay = 0.99** :
  - Долгое исследование среды, что позволяет агенту лучше адаптироваться к редким состояниям.
  - Потери снижаются плавно, но обучение занимает больше времени.
- **Epsilon_decay = 0.9** :
  - Баланс между исследованием и эксплуатацией.
  - Награда достигает 200 за 20–70 эпизодов, при gamma=0.8-0.9 потери стабильны.
- **Epsilon_decay = 0.8** :
  - Агент может "застрять" в локальных оптимумах, что вызывает колебания награды и потерь.

4. Начальное значение epsilon

- **Epsilon = 0.9** :
  - Активное исследование среды на начальных этапах.
  - Награда растёт медленно, без учета редких событий достигает 200.
  - Потери снижаются плавно.
- **Epsilon = 0.7–0.1** :
  - Быстрое достижение 200 баллов, но с риском недообучения.
  - Потери имеют колебания, что указывает на недостаточное исследование.
  - Агент не справляется с редкими событиями, из-за чего есть выбросы потерь.

ПРИЛОЖЕНИЕ А (Исходный код программы)

```
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 torch import nn, optim

torch.set_num_threads(os.cpu_count())
torch.set_num_interop_threads(os.cpu_count())

# --- Конфигурации ---
LAYER_ARCHITECTURES = {
    "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),
    ],
}

HYPERPARAMS = {
    "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,
    "target_update_freq": 10,  # Частота обновления целевой сети
}

# --- Буфер опыта ---
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.long),
            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)

# --- Q-сеть ---
class QNetwork(nn.Module):
    def __init__(self, obs_size, n_actions, layers:
list[nn.Module]):
        super().__init__()
        self.net = nn.Sequential(*layers)

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

```
# --- DQN Агент ---
class DQNAgent:
    def __init__(self, obs_size, n_actions, layers, **params):
        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.target_net = QNetwork(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=params["lr"])

        self.gamma = params["gamma"]
        self.epsilon = params["epsilon"]
        self.epsilon_decay = params["epsilon_decay"]
        self.epsilon_min = params["epsilon_min"]
        self.batch_size = params["batch_size"]
        self.target_update_freq = params["target_update_freq"]

        self.replay_buffer = ReplayBuffer(1000)
        self.loss = 0.0
        self.steps_done = 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

        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(self.batch_size)
        states = states.to(self.device)
        actions = actions.to(self.device)
        rewards = rewards.to(self.device)
        next_states = next_states.to(self.device)
        dones = dones.to(self.device)

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

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

        # Обновление epsilon
        self.epsilon = max(self.epsilon * self.epsilon_decay,
self.epsilon_min)
        self.steps_done += 1

        # Обновление целевой сети
        if self.steps_done % self.target_update_freq == 0:

self.target_net.load_state_dict(self.q_net.state_dict())

# --- Симуляция ---
```

```
class Simulation:
    def __init__(self, env, agent, num_episodes, num_steps):
        self.env = env
        self.agent = agent
        self.num_episodes = num_episodes
        self.num_steps = num_steps
        self.reward_history = []
        self.loss_history = []

    def run(self):
        for episode in range(self.num_episodes):
            state, _ = self.env.reset()
            episode_reward = 0
            episode_loss = 0
            for step in range(self.num_steps):
                action = self.agent.select_action(state)
                next_state, reward, done, truncated, _ =
self.env.step(action)
                self.agent.replay_buffer.push(state, action,
reward, next_state, done)
                state = next_state
                self.agent.train()
                episode_reward += reward
                episode_loss += self.agent.loss
                if done:
                    break
            self.reward_history.append(episode_reward)
            self.loss_history.append(episode_loss)

# --- Визуализация ---
def plot_results(results, title_suffix=""):
    os.makedirs("fig", exist_ok=True)
```

```
    for name, data in results.items():
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

        ax1.plot(data["reward"], label=name, color="blue")
        ax1.set_title(f"Награда за эпизод — {title_suffix} — {name}")
        ax1.set_xlabel("Эпизод")
        ax1.set_ylabel("Награда")
        ax1.grid(True)
        ax1.legend()

        ax2.plot(data["loss"], label=name, color="orange")
        ax2.set_title(f"Потери — {title_suffix} — {name}")
        ax2.set_xlabel("Эпизод")
        ax2.set_ylabel("Потери")
        ax2.grid(True)
        ax2.legend()

        plt.tight_layout()
        fig.savefig(f"fig/{title_suffix}_{name}.png")
        plt.close(fig)


# --- Эксперименты ---
def run_experiment(name, config, hyperparams):
    env = gym.make("CartPole-v1")
    agent = DQNAgent(obs_size=4, n_actions=2, layers=LAYER_ARCHITECTURES[config], **hyperparams)
    sim = Simulation(env, agent, hyperparams["num_episodes"], hyperparams["num_steps"])
    sim.run()
    return {"reward": sim.reward_history, "loss": sim.loss_history}
```

```
# --- Основная часть ---
if __name__ == "__main__":
    os.makedirs("fig", exist_ok=True)

    # Архитектура
     architecture_results = {}
     for arch in LAYER_ARCHITECTURES:
         print(f"Тестирование архитектуры: {arch}")
         result = run_experiment("architecture", arch,
HYPERPARAMS)
         architecture_results[arch] = result
     plot_results(architecture_results, "Архитектура")

    # Gamma и Epsilon Decay
    gamma_results = {}
    for gamma in [0.99, 0.9, 0.8]:
        for decay in [0.99, 0.9, 0.8]:
            print(f"Gamma={gamma}, Decay={decay}")
            params = HYPERPARAMS.copy()
            params["gamma"] = gamma
            params["epsilon_decay"] = decay
            result = run_experiment("gamma_decay", "default",
params)
            gamma_results[f"gamma={gamma}, decay={decay}"] =
result
    plot_results(gamma_results, "Gamma_Decay")

    # Epsilon
    epsilon_results = {}
    for eps in [0.9, 0.7, 0.5, 0.3, 0.1]:
        print(f"Epsilon={eps}")
        params = HYPERPARAMS.copy()
        params["epsilon"] = eps
```

```
        result = run_experiment("epsilon", "default", params)
        epsilon_results[f"epsilon={eps}"] = result
    plot_results(epsilon_results, "Epsilon")
```