ВМИНОБРНАУКИ 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 _____ Хвостунов М. М.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург
2025

Цель работы.

Ознакомиться с методом Deep Q-Network и реализовать его на языке Python с использованием библиотеки PyTorch для среды CartPole-v1.

Постановка задачи.

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

Выполнение задач.

Среда CartPole-v1 представляет собой классическую задачу управления: удержание вертикального положения шеста, установленного на движущейся тележке. Агент получает 4 параметра состояния и может совершать два действия: двигать тележку влево или вправо. За каждый временной шаг, в течение которого шест удерживается вертикально, агент получает награду 1. Эпизод завершается при отклонении шеста или выходе тележки за границы.

Для оценки Q-функции используется нейросеть. Используется две модели: policy_net (основная сеть) и target_net (фиксированная копия для расчёта цели обучения). Оптимизация происходит с использованием буфера воспроизведения (ReplayBuffer) и стохастического градиентного спуска.

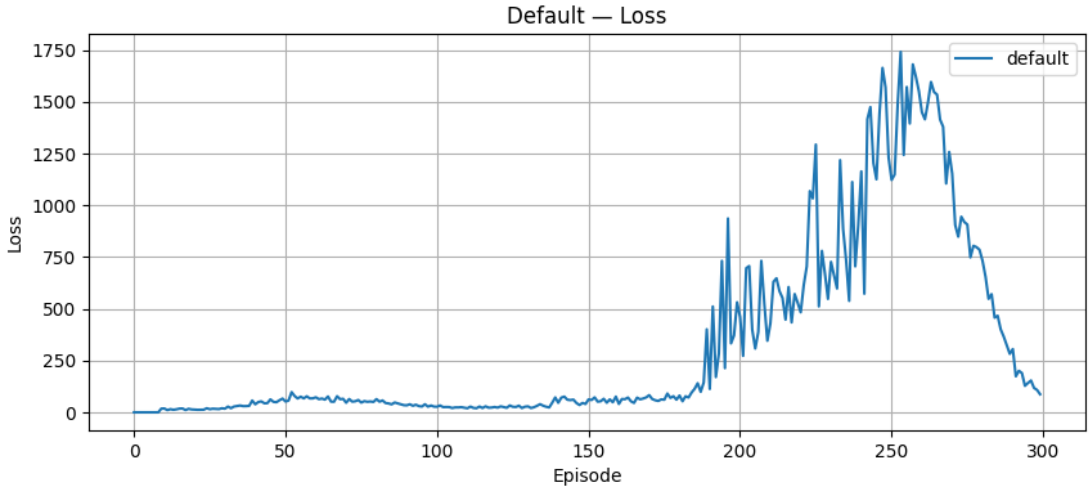

Рисунок 1 – Потери при обучении базовой архитектуры

Различные архитектуры.

Архитектура нейронной сети отвечает за то, насколько сложная аппроксимация функции, которая переводит начальные данные в конечные, может быть получена в ходе обучения нейронной сети.

Были протестированы 4 варианта архитектур нейросети:

- small: один скрытый слой на 32 нейрона,
- default: два слоя 128 и 64,
- large: три слоя 256–128–64,
- deep: четыре слоя по убыванию.

Это позволило оценить влияние глубины и ширины на устойчивость обучения.

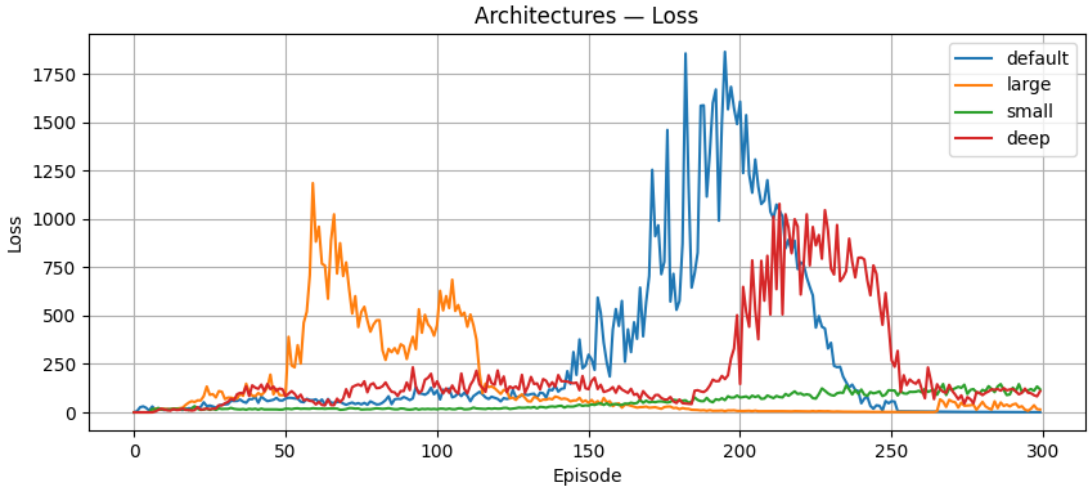

Рисунок 2 – Потери для различных архитектур

По результатам анализа графиков можно сделать несколько выводов:

- В общем и целом ближе к 300 эпизодам все модели показывают схожий результат в части ошибок
- На дефолтной и deep моделях можно заметить всплески графиков ошибок на отрезке 150-250 эпизодов, что может говорить о вероятных выбросах в процессе обучения, однако в дальнейшем они нивелировались, что говорит о полезности долгосрочного обучения любой модели
- Small архитектура дала самый стабильный результат в процессе обучения

Влияние изменения параметров *gamma и decay*.

Также интересно было провести эксперименты с различными комбинациями параметров gamma и epsilon_decay, чтобы узнать их влияние на результат обучения модели.

Было проведено несколько экспериментов - параметры gamma (0.7–0.99) и epsilon_decay (0.8–0.995) комбинировались, чтобы оценить, как долгосрочное планирование и скорость уменьшения случайности влияют на процесс обучения.

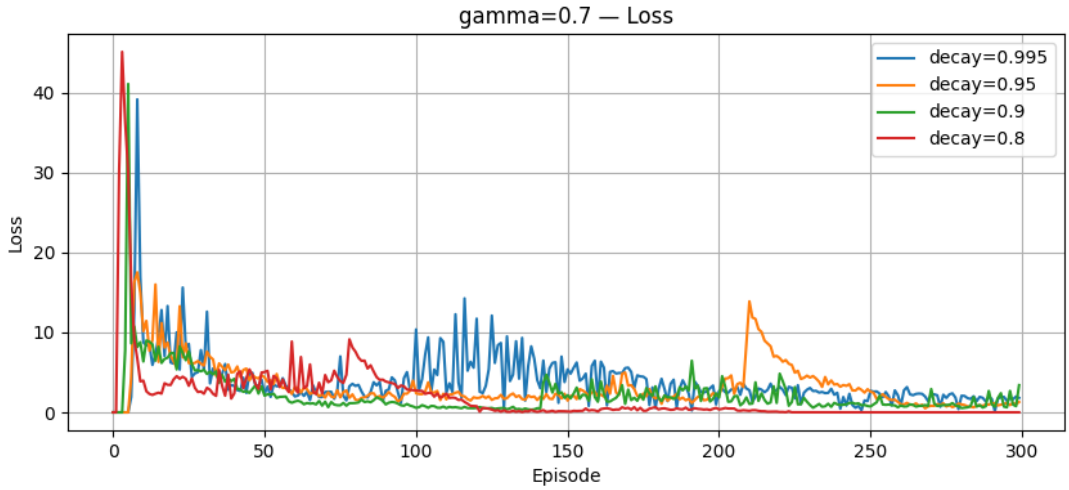

Рисунок 3 - Gamma = 0.7, различные epsilon_decay

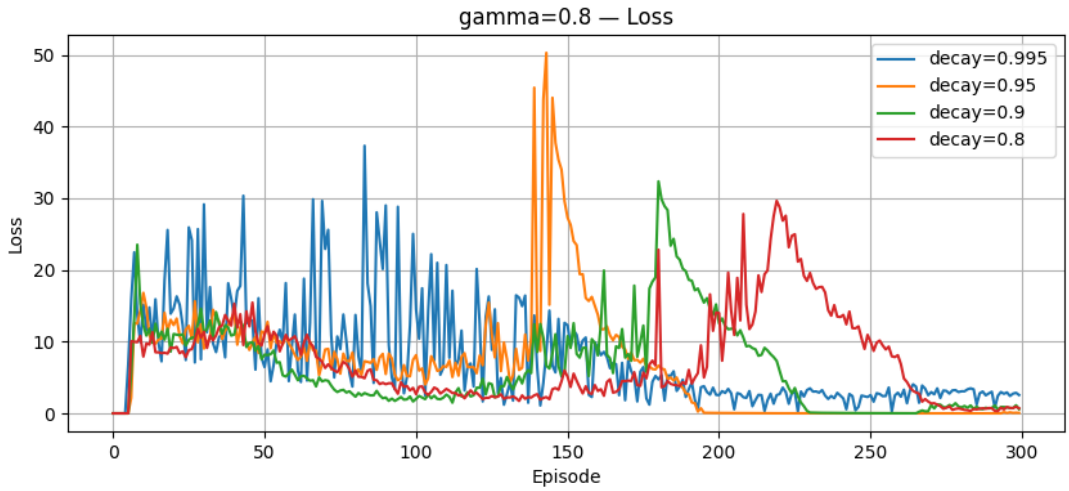

Рисунок 4 - Gamma = 0.8, различные epsilon_decay

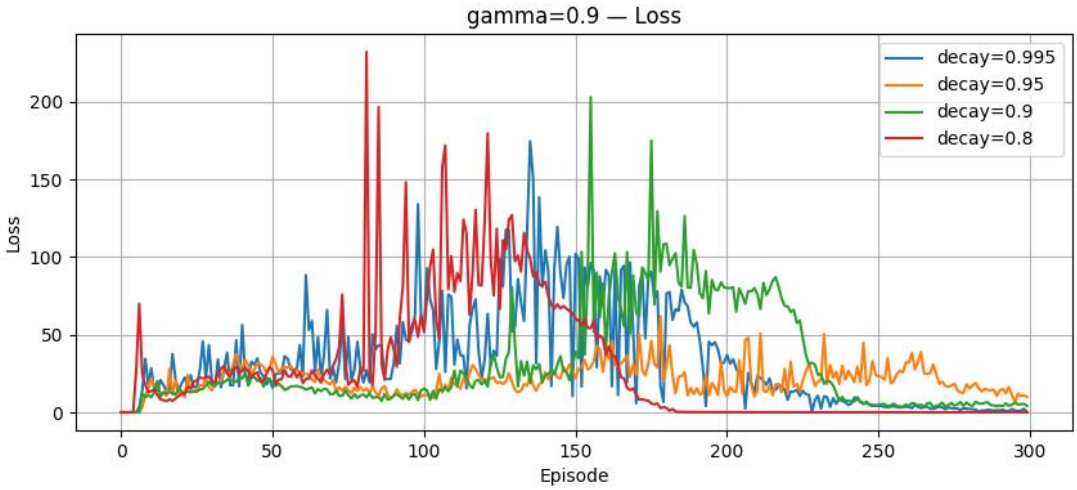

Рисунок 5 - Gamma = 0.9, различные epsilon_decay

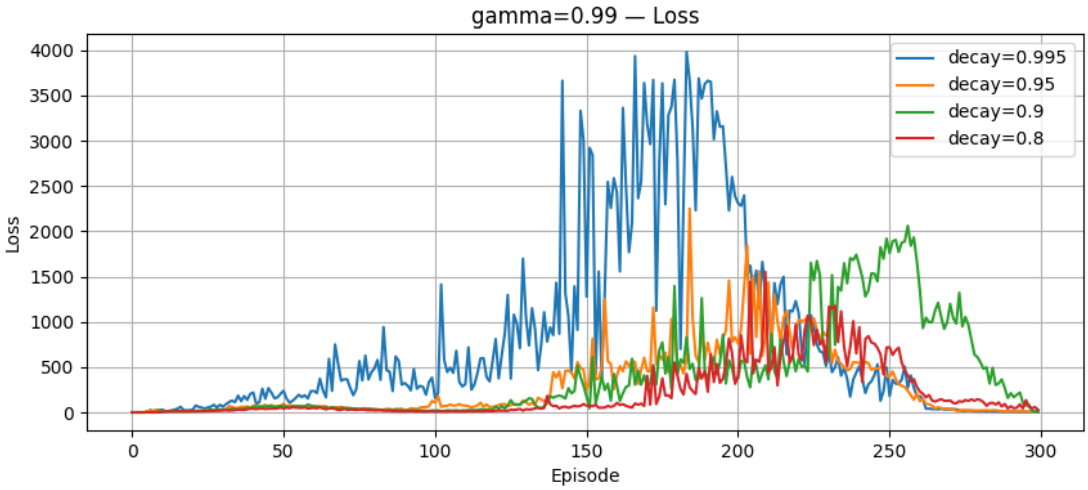

Рисунок 6 - Gamma = 0.99, различные epsilon_decay

По итогам анализа графиков можно сделать несколько выводов:

- Наилучшая сходимость достигается при значениях gamma = 0.8 и 0.9 с decay в диапазоне 0.9–0.95.
- При gamma = 0.7 обучение в целом стабильно, но агент недооценивает будущие награды, что ограничивает качество стратегии
- Повышение gamma до 0.99 приводит к сильной неустойчивости и резкому росту потерь, особенно при медленном затухании epsilon (decay = 0.995). Это объясняется тем, что агент становится слишком ориентированным на отдалённые награды, сохраняя при этом высокую случайность выбора действий, что затрудняет обучение.

Таким образом, оптимальными параметрами являются gamma = 0.9 и epsilon_decay = 0.9–0.95, обеспечивающие баланс между долговременным планированием и скоростью перехода к целенаправленным действиям.

Влияние изменения параметра *epsilon*.

Начальное значение epsilon варьировалось от 0.9 до 0.1, чтобы определить, как быстро агент начинает использовать выученную стратегию вместо случайного выбора.

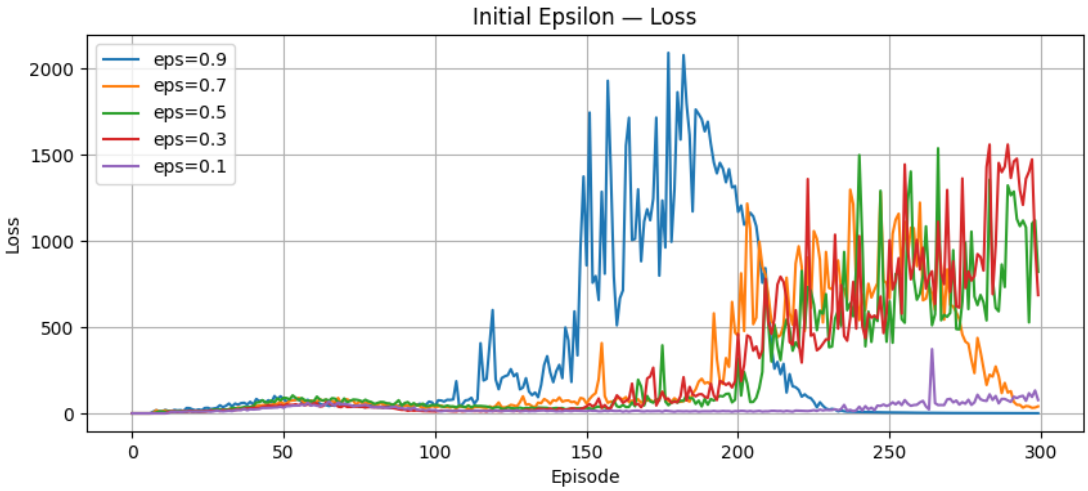

Рисунок 7 – Потери при различных начальных значениях epsilon

График потерь при различных начальных значениях параметра ε показывает, как скорость перехода от случайного поведения к использованию обученной стратегии влияет на стабильность и эффективность обучения. При ε = 0.9 агент слишком долго действует случайно, что приводит к высоким и нестабильным потерям. Значения ε = 0.7 и ε = 0.5 демонстрируют более стабильное обучение и постепенное снижение ошибки, что указывает на оптимальный баланс между исследованием среды и применением стратегии. При ε = 0.3 обучение начинается быстрее, но потери становятся менее стабильными, что говорит о преждевременном отказе от исследования. Наименьшее значение ε = 0.1 приводит к самым низким потерям, однако это может быть связано с тем, что агент рано перестал исследовать среду, попав в локальный минимум. Таким образом, оптимальными значениями ε являются 0.5 и 0.7.

Заключение.

В данной работе был реализован алгоритм DQN на PyTorch для среды CartPole-v1. Проведённые эксперименты показали, что увеличение архитектуры сети улучшает обучение, значение gamma = 0.9 и epsilon_decay около 0.9–0.95 обеспечивают наилучший баланс, а начальное значение epsilon в пределах 0.5–0.7 обеспечивает наиболее стабильное и быстрое обучение.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

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

# --- Настройки и архитектуры ---
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,
}

architectures = {
    "default": [128, 64],
    "large": [256, 128, 64],
    "small": [32],
    "deep": [64, 64, 64, 32]
}

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(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 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=params["lr"])
        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) < params["batch_size"]:
            return 0
        s, a, r, s2, d = self.buffer.sample(params["batch_size"])
        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, params["ep-
silon min"])

    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):
    plt.figure(figsize=(10, 4))
    for label, (rewards, _) in results.items():
        plt.plot(rewards, label=label)
    plt.title(f"{title} — Reward")
    plt.xlabel("Episode")
    plt.ylabel("Reward")
    plt.legend()
    plt.grid()
    os.makedirs("figs", exist_ok=True)
    plt.savefig(f"figs/{filename} reward.png")
    plt.close()

    plt.figure(figsize=(10, 4))
    for label, (_, losses) in results.items():
```

```
        plt.plot(losses, label=label)
    plt.title(f"{title} — Loss")
    plt.xlabel("Episode")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid()
    plt.savefig(f"figs/{filename}_loss.png")
    plt.close()

def experiment_default():
    env = gym.make("CartPole-v1")
    agent = DQNAgent(4, 2, architectures["default"], params["gamma"],
params["epsilon"], params["epsilon decay"])
    rewards, losses = train(agent, env, params["num episodes"],
params["num_steps"])
    return {"default": (rewards, losses)}

def experiment_architectures():
    env = gym.make("CartPole-v1")
    results = {}
    for name, layers in architectures.items():
        agent = DQNAgent(4, 2, layers, params["gamma"], params["epsi-
lon"], 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, 0.7]:
        for decay in [0.995, 0.95, 0.9, 0.8]:
            label = f"g={gamma}, d={decay}"
            agent = DQNAgent(4, 2, architectures["default"], 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, architectures["default"], 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

def experiment gamma decay grouped():
    env = gym.make("CartPole-v1")
    grouped_results = {}
    for gamma in [0.99, 0.9, 0.8, 0.7]:
        sub results = {}
        for decay in [0.995, 0.95, 0.9, 0.8]:
            label = f"decay={decay}"
            agent = DQNAgent(4, 2, architectures["default"], gamma,
params["epsilon"], decay)
            rewards, losses = train(agent, env, params["num_episodes"],
params["num_steps"])
            sub_results[label] = (rewards, losses)
        grouped_results[f"gamma={gamma}"] = sub_results
    return grouped_results

def plot_results_grouped(grouped_results, base_filename):
    os.makedirs("figs", exist_ok=True)
    for group_name, results in grouped_results.items():
        # Reward
        plt.figure(figsize=(10, 4))
        for label, (rewards,  ) in results.items():
            plt.plot(rewards, label=label)
        plt.title(f"{group_name} — Reward")
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.legend()
        plt.grid()
        plt.savefig(f"figs/{base_filename}_{group_name}_reward.png")
```

```
        plt.close()

        # Loss
        plt.figure(figsize=(10, 4))
        for label, (_, losses) in results.items():
            plt.plot(losses, label=label)
        plt.title(f"{group_name} — Loss")
        plt.xlabel("Episode")
        plt.ylabel("Loss")
        plt.legend()
        plt.grid()
        plt.savefig(f"figs/{base_filename}_{group_name}_loss.png")
        plt.close()

if __name__ == "__main__":
    print("Running default experiment...")
    r_default = experiment_default()
    plot_results(r_default, "Default", "default")

    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")

    print("Running improved gamma & decay experiment...")
    r_grouped = experiment_gamma_decay_grouped()
    plot_results_grouped(r_grouped, "gamma_decay_grouped")
```