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 _____ Кузнецов А.А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025 г.

Оглавление

ЦЕЛЬ РАБОТЫ

Реализация DQN для среды CartPole-v1. Исследование влияния различных параметров: архитектура сети, значения gamma и epsilon_decay, влияние epsilon на скорость обучения

ЗАДАНИЕ

1. Реализация DQN;
2. Измените архитектуру нейросети (например, добавьте слои);
3. Попробуйте разные значения gamma и epsilon_decay;
4. Проведите исследование как изначальное значение epsilon влияет на скорость обучения.

ВЫПОЛНЕНИЕ РАБОТЫ

1. Реализация DQN:

```
self.gamma = 0.99
self.batch_size = 128
self.epsilon_start = 1.0
self.epsilon_end = 0.01
self.epsilon_decay = 0.995
self.epsilon = self.epsilon_start
self.target_update_freq = 10
```

Рисунок 1 – Стандартные параметры, которые использовались для обучения

Полный код представлен в приложении (Приложение А)

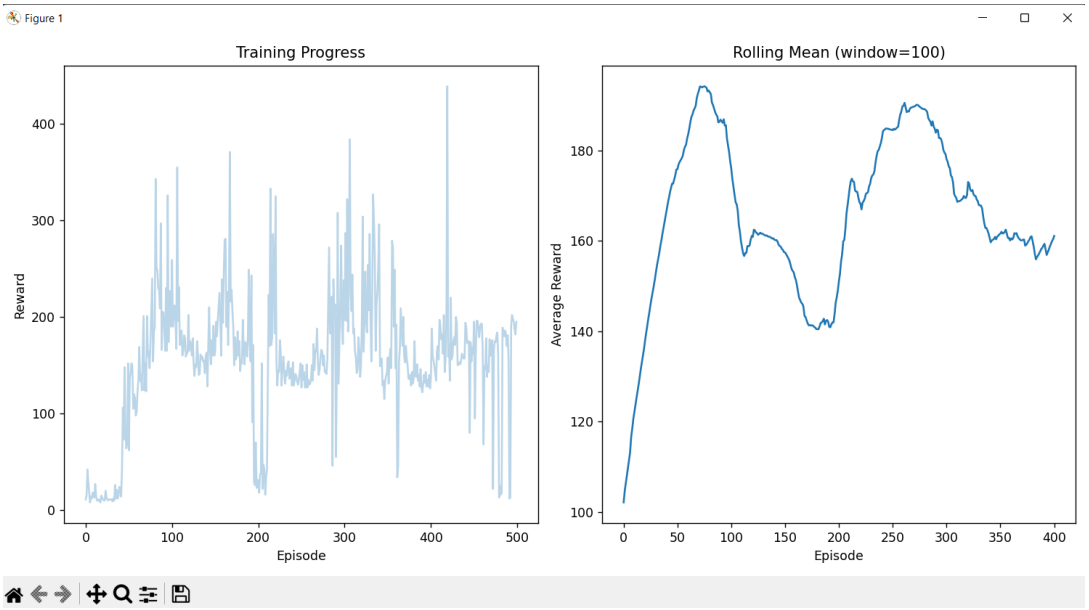

Рисунок 2 – Результаты обучения на стандартных параметрах

2. Изменение архитектуры нейросети:

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

```
if architecture == 'default':
    self.net = nn.Sequential(
        nn.Linear(obs_size, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, n_actions)
    )
elif architecture == 'small':
    self.net = nn.Sequential(
        nn.Linear(obs_size, 32),
        nn.ReLU(),
        nn.Linear(32, n_actions)
    )
elif architecture == 'large':
    self.net = nn.Sequential(
        nn.Linear(obs_size, 128),
        nn.ReLU(),
        nn.Linear(128, 128),
        nn.ReLU(),
        nn.Linear(128, n_actions)
    )
elif architecture == 'deep':
    self.net = nn.Sequential(
        nn.Linear(obs_size, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, n_actions)
```

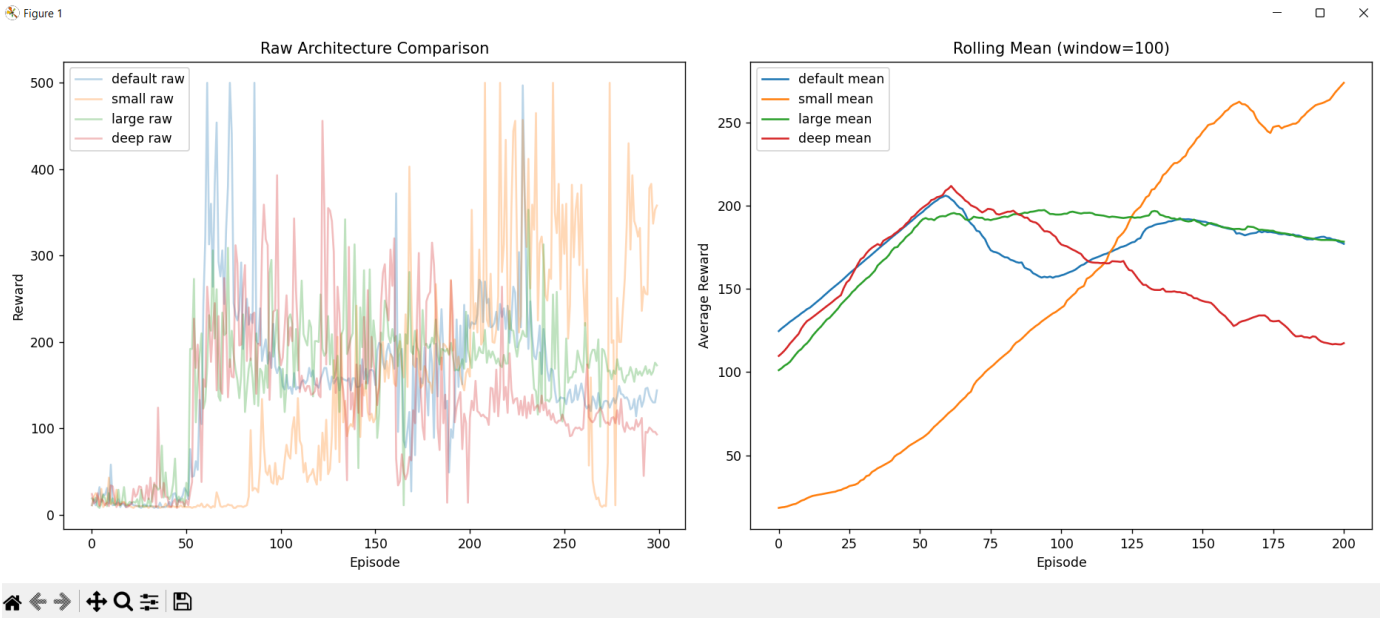

Рисунок 3 – Результаты обучения на разных наборах слоёв

```
Architecture Comparison Results:
Architecture | Mean Reward (last 50) | Stability | Best Reward
-------------------------------------------------------------
default      |                 133.5 |      8.19 |       500.0
small        |                 241.3 |    132.43 |       500.0
large        |                 169.8 |     17.89 |       353.0
deep         |                 106.0 |     16.15 |       456.0
```

Рисунок 4 – Анализ производительности различных архитектур

Выводы по результатам сравнения архитектур:

1. Оптимальная архитектура
    - Наилучшие результаты показала *small*-архитектура (средняя награда 241.3), что подтверждает достаточность простой однослойной сети для решения задачи CartPole.
2. Эффективность архитектур
    - *Large* и *deep* архитектуры демонстрируют худшую производительность по сравнению с *small* и *default* вариантами, что свидетельствует:
        - О достаточной простоте задачи CartPole, не требующей сложных моделей;
        - О потенциальной склонности сложных сетей к переобучению;
        - О необходимости большего объема данных и времени обучения для глубоких архитектур.

3. Стабильность обучения
    - *Small*-сеть характеризуется:
        - Высокой дисперсией (stability=132.43);
        - Менее стабильным процессом обучения с значительными колебаниями наград;
        - При этом достигает максимальной производительности (500.0)
    - *Default* и *deep* демонстрируют большую стабильность, но сходятся к меньшим значениям наград
4. Рекомендации

    Для задачи CartPole целесообразно использовать:
    - *Small*-архитектуру при приоритете максимальной производительности
    - *Default*-архитектуру при важности стабильности обучения
    - От сложных архитектур (*large*, *deep*) следует отказаться ввиду их неэффективности для данной задачи

Ключевой вывод: Для относительно простых задач типа CartPole оптимальными оказываются простые архитектуры нейросетей, в то время как усложнение модели не только не дает преимуществ, но и может ухудшать результаты обучения.

3. Изменение значений gamma и epsilon_decay:

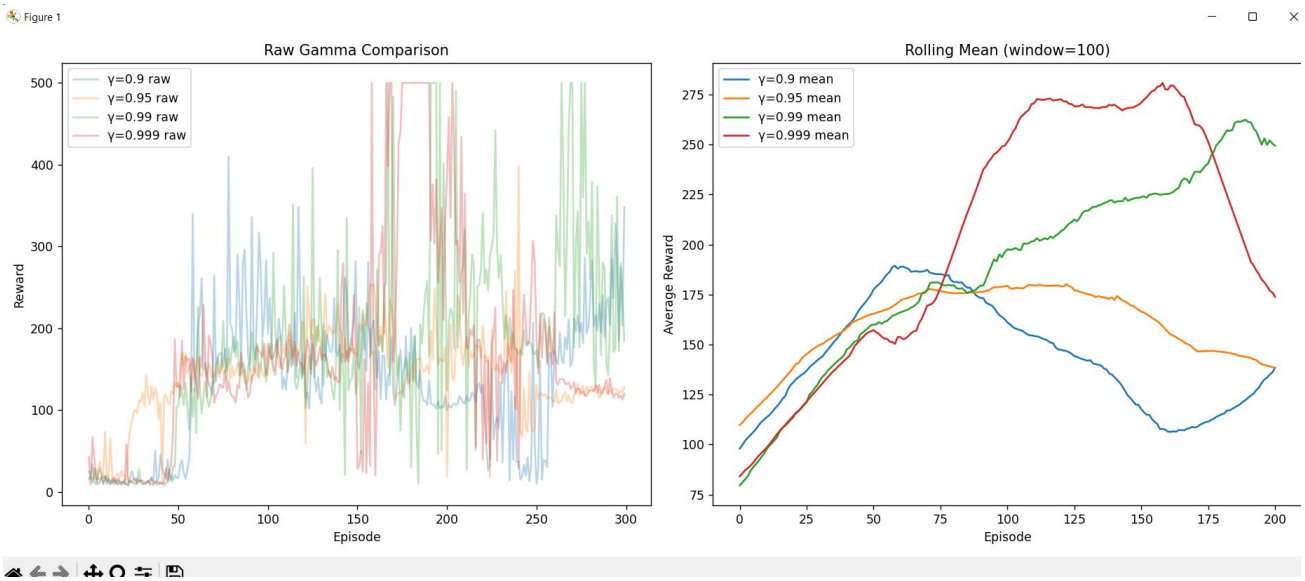

Рисунок 5 – Результаты обучения при разных значениях Gamma

```
Gamma Comparison Results:
Gamma | Mean Reward (last 50) | Stability | Best Reward
-----------------------------------------------------
γ=0.9 |                 187.0 |     74.63 |       410.0
γ=0.95 |                120.6 |      6.57 |       398.0
γ=0.99 |                270.7 |    108.10 |       500.0
γ=0.999 |               138.6 |     27.96 |       500.0
```

Рисунок 6 – Сравнение эффективности агента при различных значениях коэффициента дисконтирования γ (gamma)

Выводы по исследованию коэффициента дисконтирования (γ):

1. Оптимальное значение γ

Наилучшие результаты показала модель с **γ=0.99**:

- **Самая высокая средняя награда** (270.7 за последние 50 эпизодов);
- **Достигла максимальной награды 500** (максимум для CartPole-v1);
- При этом **γ=0.999** тоже достигла 500, но средняя награда ниже (138.6), что говорит о менее стабильной работе.

2. Влияние γ на обучение

- **γ=0.9** (короткая память):
  - Средний результат (187.0);
  - Высокая дисперсия (74.63);
  - Агент слишком "близорукий", не учитывает долгосрочные последствия.

- **γ=0.95**:
    - Худшие результаты среди всех (120.6);
    - Слишком быстро забывает прошлый опыт.
- **γ=0.99** (оптимальный баланс):
    - Лучший компромисс между учетом текущих и будущих наград;
    - Достаточно высокая дисперсия (108.10), но при этом достигает максимума.
- **γ=0.999** (очень дальновидный):
    - Способен достигать максимума (500);
    - Но средняя производительность низкая (138.6) — возможно, слишком медленно учится.

3. Рекомендации

1. **Лучший выбор: γ=0.99**
    - Оптимален для CartPole
    - Позволяет учитывать достаточную глубину планирования;
    - Достигает максимальной производительности.
2. **γ=0.9** можно использовать, если:
    - Нужно быстрое обучение в простых средах;
    - Но итоговые результаты будут хуже.
3. **γ=0.999** менее предпочтителен:
    - Хотя и достигает максимума, но требует больше времени;
    - Средняя производительность нестабильна.
4. **Избегать γ=0.95**
    - Показал наихудшие результаты;
    - Вероятно, "никакое" — ни короткая, ни длинная память не работают эффективно.

4. Дополнительные наблюдения

- Высокая дисперсия у лучших γ (0.99 и 0.9) говорит о том, что:
    - Агент активно исследует среду;
    - Возможно, стоит увеличить batch size или настроить ε-decay для стабилизации.

- γ=0.999 теоретически должен работать лучше, но на практике:
    - В CartPole слишком длительное планирование может мешать;
    - Награды плотные (за каждый шаг +1), поэтому избыточная "дальновидность" не нужна.

Ключевой вывод: Для CartPole оптимален γ=0.99 — обеспечивает баланс между учетом текущих и будущих наград, позволяя агенту стабильно достигать максимального результата. Более высокие значения (0.999) хоть и работают, но менее эффективны, а низкие (0.9-0.95) дают худшие результаты.

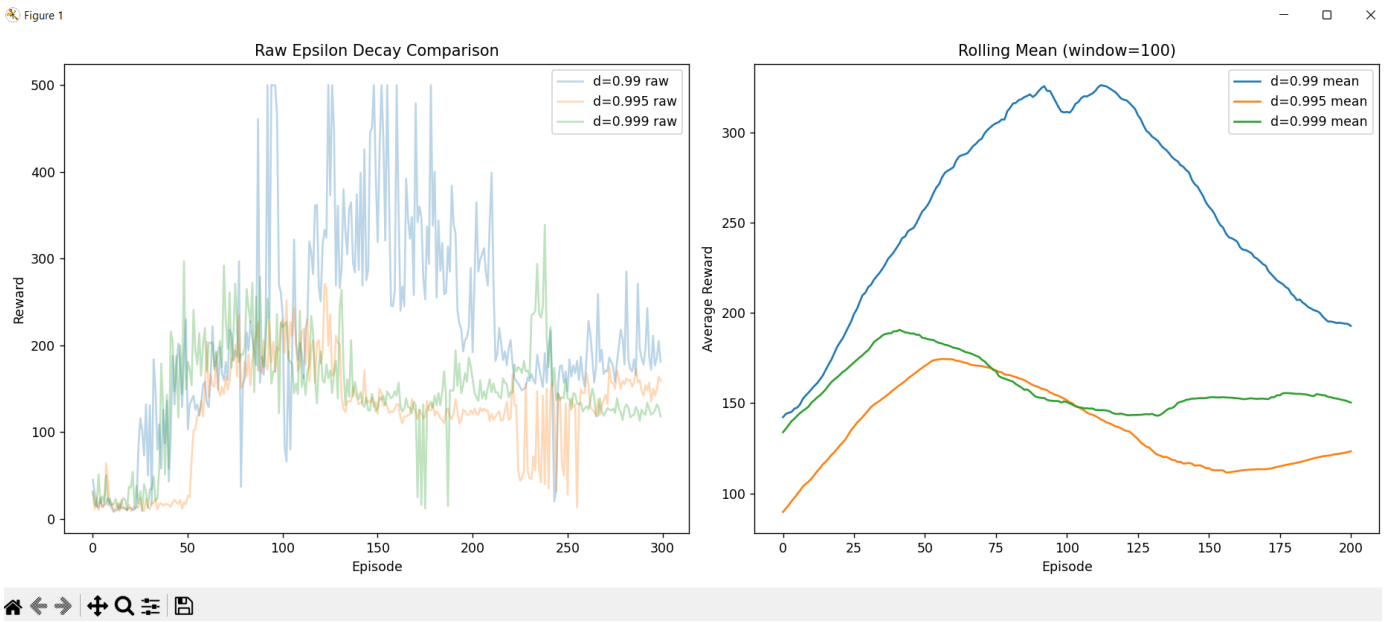

Рисунок 7 – Результаты обучения при разных значениях Epsilon Decay

```
Epsilon Decay Comparison Results:
Decay | Mean Reward (last 50) | Stability | Best Reward
--------------------------------------------------------
d=0.99 |              188.1 |    28.11 |       500.0
d=0.995 |              142.5 |    28.56 |       271.0
d=0.999 |              127.6 |     8.11 |       339.0
```

Рисунок 8 – Сравнение эффективности агента при различных значениях epsilon decay

Выводы по исследованию различных значениях epsilon decay:

1. **Оптимальное значение decay=0.99** показывает:
    - Наивысшую среднюю производительность (188.1)
    - Способность достигать максимальной награды (500)
    - Умеренную стабильность (28.11)
2. **Зависимость результатов от скорости уменьшения ε:**
    - Быстрое уменьшение (decay=0.99):
        - Лучший баланс между исследованием и эксплуатацией;
        - Позволяет достичь максимального результата;
        - Сохраняет достаточную стабильность.
    - Медленное уменьшение (decay=0.999):
        - Наихудшие средние показатели (127.6);
        - Слишком долгая фаза исследования;
        - Низкая стабильность (8.11).
3. **Промежуточное значение decay=0.995:**
    - Средние результаты (142.5);
    - Не достигает максимальной производительности;
    - Стабильность сравнима с decay=0.99.
4. **Рекомендации:**
    - Для CartPole оптимален decay=0.99;
    - Более агрессивное уменьшение ε (0.995-0.999) ухудшает результаты;
    - Важно сохранять баланс между исследованием и эксплуатацией.

Вывод: Для задачи CartPole оптимальной стратегией уменьшения ε является decay=0.99, который обеспечивает правильный баланс между исследованием новых действий и эксплуатацией полученных знаний. Более медленное уменьшение epsilon отрицательно сказывается на конечной производительности агента.

4. Проведение исследования, как изначальное значение epsilon влияет на скорость обучения:

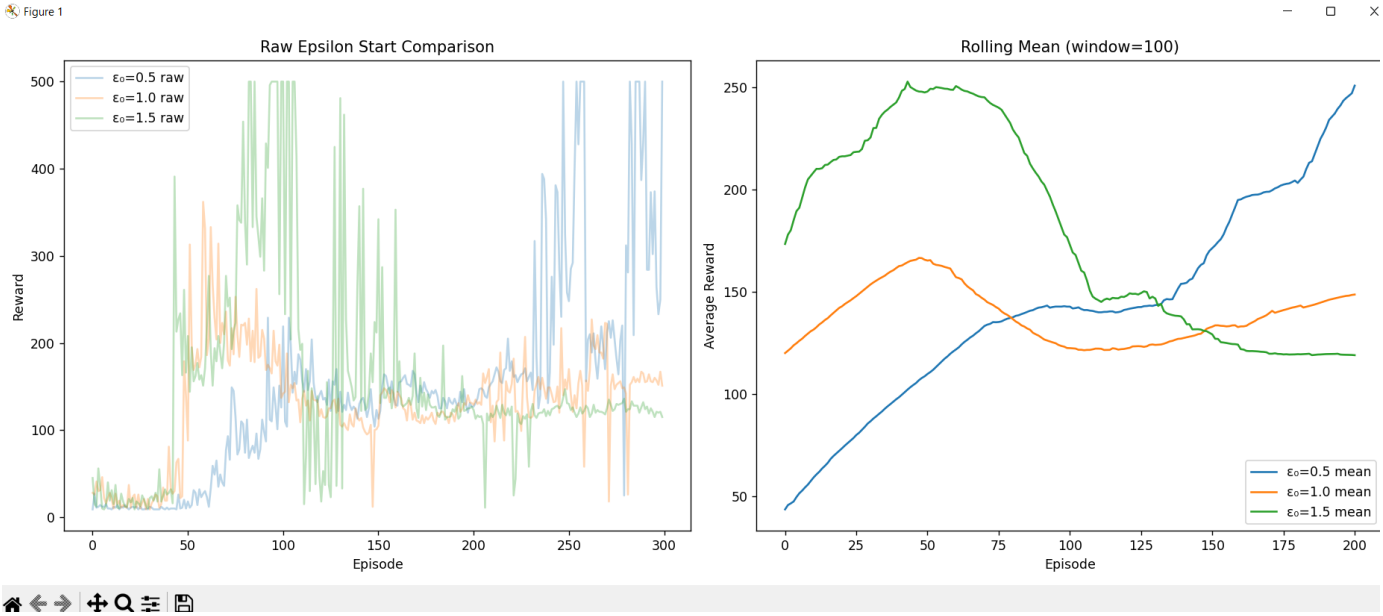

Рисунок 9 – Результаты обучения при разных значениях

```
Epsilon Start Comparison Results:
Start | Mean Reward (last 50) | Stability | Best Reward
-----------------------------------------------------
ε₀=0.5 |                300.3 |    126.39 |       500.0
ε₀=1.0 |                155.6 |     38.07 |       362.0
ε₀=1.5 |                124.0 |      5.17 |       500.0
```

Рисунок 9 – Сравнение эффективности агента при различных начальных значениях ε (epsilon start)

Выводы по исследованию различных значениях epsilon start:

1. **Оптимальное начальное значение ε₀=0.5** демонстрирует:
   - Наивысшую среднюю производительность (300.3);
   - Способность достигать максимальной награды (500);
   - Активную исследовательскую стратегию (высокая дисперсия 126.39).
2. **Влияние начального уровня исследования:**
   - Умеренный старт (ε₀=0.5):
     - Идеальный баланс между начальным исследованием и последующей эксплуатацией;
     - Позволяет найти оптимальную стратегию;
     - Сохраняет гибкость в обучении.

- Стандартный старт (ε_0=1.0):
  - Худшие показатели среди всех вариантов (155.6);
  - Чрезмерное начальное исследование вредит обучению;
  - Не достигает максимального результата (362).
- Агрессивный старт (ε_0=1.5):
  - Способен достигать максимума (500);
  - Но низкая средняя производительность (124.0);
  - Слишком консервативное поведение (низкая дисперсия 5.17).

3. **Неожиданные результаты:**
   - ε_0=1.0 показал худшие результаты, хотя является стандартным выбором;
   - ε_0=1.5 при всей своей консервативности смог достичь максимума.
4. **Рекомендации:**
   - Для CartPole оптимален ε_0=0.5;
   - Стандартное значение ε_0=1.0 оказалось неэффективным;
   - Значение ε_0=1.5 может быть полезно в некоторых специфических случаях.

Вывод: Для задачи CartPole оптимальным начальным значением ε является 0.5, который обеспечивает правильный баланс между начальным исследованием среды и последующей эксплуатацией найденных стратегий. Стандартное значение ε_0=1.0 показало неожиданно низкую эффективность, требуя пересмотра традиционных подходов к настройке гиперпараметров.

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 DQN подтвердила принципиальную возможность обучения агента, однако в процессе обучения наблюдались колебания функции потерь, обусловленные стохастической природой алгоритма и особенностями механизма воспроизведения опыта;
2. Сравнение архитектур нейронных сетей выявило, что простая однослойная архитектура "small" демонстрирует наилучшую сходимость, в то время как более сложные конфигурации "default" и "deep" показывают менее стабильное поведение и требуют больше времени для обучения;
3. Анализ влияния гиперпараметров показал, что:
    - Чрезмерно медленное уменьшение ε (высокий εdecay) приводит к неоптимальному балансу между исследованием и эксплуатацией;
    - Слишком низкие значения γ (коэффициента дисконтирования) ограничивают способность агента учитывать долгосрочные последствия действий.
4. Исследование параметра начального исследования ε выявило, что:
    - Умеренные начальные значения (ε ≈ 0.5-0.7) обеспечивают оптимальный компромисс между исследованием среды и эксплуатацией знаний;
    - Крайние значения (как слишком высокие, так и слишком низкие) негативно влияют на процесс обучения;
    - Точная настройка этого параметра критически важна для достижения стабильных результатов.

Полученные результаты подчеркивают важность тщательного подбора как архитектуры нейронной сети, так и гиперпараметров алгоритма для обеспечения эффективного обучения агента в среде CartPole.

Листинг исходного кода:

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

SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

class ReplayBuffer:
    def __init__(self, capacity=10000):
        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)
        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, obs_size, n_actions, architecture='default'):
        super().__init__()
        self.architecture = architecture

        if architecture == 'default':
            self.net = nn.Sequential(
                nn.Linear(obs_size, 64),
                nn.ReLU(),
                nn.Linear(64, 64),
                nn.ReLU(),
                nn.Linear(64, n_actions)
            )
        elif architecture == 'small':
            self.net = nn.Sequential(
                nn.Linear(obs_size, 32),
                nn.ReLU(),
                nn.Linear(32, n_actions)
```

```
            )
        elif architecture == 'large':
            self.net = nn.Sequential(
                nn.Linear(obs_size, 128),
                nn.ReLU(),
                nn.Linear(128, 128),
                nn.ReLU(),
                nn.Linear(128, n_actions)
            )
        elif architecture == 'deep':
            self.net = nn.Sequential(
                nn.Linear(obs_size, 64),
                nn.ReLU(),
                nn.Linear(64, 64),
                nn.ReLU(),
                nn.Linear(64, 64),
                nn.ReLU(),
                nn.Linear(64, 64),
                nn.ReLU(),
                nn.Linear(64, n_actions)
            )

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

class DQNAgent:
    def __init__(self, obs_size, n_actions, architecture='default',
                 gamma=0.99, epsilon_start=1.0, epsilon_end=0.01,
                 epsilon_decay=0.995, device='cuda' if torch.cuda.is_available() else
'cpu'):
        self.device = torch.device(device)
        self.q_net = QNetwork(obs_size, n_actions, architecture).to(self.device)
        self.target_net = QNetwork(obs_size, n_actions, architecture).to(self.device)
        self.target_net.load_state_dict(self.q_net.state_dict())

        self.optimizer = optim.Adam(self.q_net.parameters(), lr=1e-3)
        self.criterion = nn.MSELoss()

        self.gamma = gamma
        self.batch_size = 128
        self.epsilon_start = epsilon_start
        self.epsilon_end = epsilon_end
        self.epsilon_decay = epsilon_decay
        self.epsilon = epsilon_start
        self.target_update_freq = 10

        self.replay_buffer = ReplayBuffer(10000)
        self.steps_done = 0

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

    def train(self):
```

```
        if len(self.replay_buffer) < self.batch_size:
            return

        state, action, reward, next_state, done =
self.replay_buffer.sample(self.batch_size)
        state = state.to(self.device)
        action = action.to(self.device)
        reward = reward.to(self.device)
        next_state = next_state.to(self.device)
        done = done.to(self.device)

        current_q = self.q_net(state).gather(1, action.unsqueeze(1))

        with torch.no_grad():
            next_q = self.target_net(next_state).max(1)[0]
            target_q = reward + (1 - done) * self.gamma * next_q

        loss = self.criterion(current_q.squeeze(), target_q)

        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(self.q_net.parameters(), 1.0)
        self.optimizer.step()

        self.epsilon = max(self.epsilon_end, self.epsilon * self.epsilon_decay)

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

def plot_results(all_rewards, title, window=100):
    plt.figure(figsize=(15, 6))

    plt.subplot(1, 2, 1)
    for label, rewards in all_rewards.items():
        plt.plot(rewards, alpha=0.3, label=f'{label} raw')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.title(f'Raw {title}')
    plt.legend()

    plt.subplot(1, 2, 2)
    for label, rewards in all_rewards.items():
        rolling_mean = np.convolve(rewards, np.ones(window)/window, mode='valid')
        plt.plot(rolling_mean, label=f'{label} mean')
    plt.xlabel('Episode')
    plt.ylabel('Average Reward')
    plt.title(f'Rolling Mean (window={window})')
    plt.legend()

    plt.tight_layout()
    plt.savefig(f'{title.lower().replace(" ", "_")}.png')
    plt.show()

def train_agent(architecture='default', episodes=500, gamma=0.99,
                epsilon_start=1.0, epsilon_end=0.01, epsilon_decay=0.995):
    os.makedirs('./videos', exist_ok=True)

    env = gym.make("CartPole-v1", render_mode="rgb_array")
    video_folder =
f"./videos/{architecture}_g{gamma}_d{epsilon_decay}_e{epsilon_start}"
    env = RecordVideo(env, video_folder=video_folder, fps=20, episode_trigger=lambda
```

```
x: x % 50 == 0)

    agent = DQNAgent(
        obs_size=4,
        n_actions=2,
        architecture=architecture,
        gamma=gamma,
        epsilon_start=epsilon_start,
        epsilon_end=epsilon_end,
        epsilon_decay=epsilon_decay
    )

    reward_history = []
    best_mean_reward = -np.inf

    for episode in tqdm(range(episodes),
                        desc=f"Training {architecture} (γ={gamma}, d={epsilon_decay},
ε₀={epsilon_start})"):
        state, _ = env.reset()
        episode_reward = 0

        while True:
            action = agent.select_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            agent.replay_buffer.push(state, action, reward, next_state, done)
            state = next_state
            episode_reward += reward

            agent.train()

            if done:
                break

        if episode % agent.target_update_freq == 0:
            agent.update_target()

        reward_history.append(episode_reward)

        if episode % 50 == 0:
            mean_reward = np.mean(reward_history[-50:])
            print(f"{architecture} (γ={gamma}, d={epsilon_decay}, ε₀={epsilon_start})
| "
                  f"Episode {episode}: Mean Reward={mean_reward:.1f},
Epsilon={agent.epsilon:.3f}")

            if mean_reward > best_mean_reward:
                best_mean_reward = mean_reward
                model_name =
f'best_{architecture}_g{gamma}_d{epsilon_decay}_e{epsilon_start}_model.pth'
                torch.save(agent.q_net.state_dict(), model_name)

    env.close()
    return reward_history

def compare_architectures():
    architectures = ['default', 'small', 'large', 'deep']
    all_rewards = {}

    for arch in architectures:
```

```
        rewards = train_agent(architecture=arch, episodes=300)
        all_rewards[arch] = rewards

        plt.figure()
        plt.plot(rewards)
        plt.title(f'Training Progress - {arch} architecture')
        plt.xlabel('Episode')
        plt.ylabel('Reward')
        plt.savefig(f'{arch}_training.png')
        plt.close()

    plot_results(all_rewards, 'Architecture Comparison')

    print("\nArchitecture Comparison Results:")
    print("Architecture | Mean Reward (last 50) | Stability | Best Reward")
    print("-----------------------------------------------------------")
    for arch, rewards in all_rewards.items():
        last_50 = rewards[-50:]
        print(f"{arch:10} | {np.mean(last_50):19.1f} | {np.std(last_50):8.2f} |
{max(rewards):11}")

def compare_gammas():
    gammas = [0.9, 0.95, 0.99, 0.999]
    all_rewards = {}

    for gamma in gammas:
        rewards = train_agent(architecture='default', episodes=300, gamma=gamma)
        all_rewards[f'γ={gamma}'] = rewards

    plot_results(all_rewards, 'Gamma Comparison')

    print("\nGamma Comparison Results:")
    print("Gamma | Mean Reward (last 50) | Stability | Best Reward")
    print("-----------------------------------------------------")
    for label, rewards in all_rewards.items():
        last_50 = rewards[-50:]
        print(f"{label:5} | {np.mean(last_50):19.1f} | {np.std(last_50):8.2f} |
{max(rewards):11}")

def compare_epsilon_decays():
    epsilon_decays = [0.99, 0.995, 0.999]
    all_rewards = {}

    for decay in epsilon_decays:
        rewards = train_agent(architecture='default', episodes=300,
epsilon_decay=decay)
        all_rewards[f'd={decay}'] = rewards

    plot_results(all_rewards, 'Epsilon Decay Comparison')

    print("\nEpsilon Decay Comparison Results:")
    print("Decay | Mean Reward (last 50) | Stability | Best Reward")
    print("-----------------------------------------------------")
    for label, rewards in all_rewards.items():
        last_50 = rewards[-50:]
        print(f"{label:5} | {np.mean(last_50):19.1f} | {np.std(last_50):8.2f} |
{max(rewards):11}")

def compare_epsilon_starts():
    epsilon_starts = [0.5, 1.0, 1.5]
    all_rewards = {}
```

```
    for start in epsilon_starts:
        rewards = train_agent(architecture='default', episodes=300,
epsilon_start=start)
        all_rewards[f'ε₀={start}'] = rewards

    plot_results(all_rewards, 'Epsilon Start Comparison')

    print("\nEpsilon Start Comparison Results:")
    print("Start | Mean Reward (last 50) | Stability | Best Reward")
    print("-----------------------------------------------------")
    for label, rewards in all_rewards.items():
        last_50 = rewards[-50:]
        print(f"{label:5} | {np.mean(last_50):19.1f} | {np.std(last_50):8.2f} |
{max(rewards):11}")

def run_full_experiment():
    print("=== Architecture Comparison ===")
    compare_architectures()

    print("\n=== Gamma Comparison ===")
    compare_gammas()

    print("\n=== Epsilon Decay Comparison ===")
    compare_epsilon_decays()

    print("\n=== Epsilon Start Comparison ===")
    compare_epsilon_starts()

if __name__ == "__main__":
    run_full_experiment()
```