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

ОТЧЕТ

по исследовательской работе

по дисциплине «Обучение с подкреплением»

Тема: DQN #1

Студент гр. 0310 _____ Афанасьев Н. С.

Студент гр. 0310 _____ Корсунов А. А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2025

Цель работы.

Реализовать и сравнить между собой различные версии алгоритма DQN.

Задание.

Необходимо реализовать и сравнить между собой следующие версии DQN:

- Double Q-learning
- Prioritized replay
- Dueling networks

В качестве окружения для тестирования:

- LunarLander-v3
- Mountain-Car

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 $y = r + \gamma \max_{a'} Q_{target}(s', a')$, 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.

Double DQN

На традиционное Q-обучение влияет смещение переоценки из-за шага максимизации, и это может навредить обучению. Двойное Q-обучение решает эту переоценку путем выделения в максимизации выбора действия из его оценки.

$$y = r + \gamma Q_{target}(s', \text{argmax}_{a'} Q(s', a'; \theta); \theta^-)$$

Prioritized Replay

DQN равномерно отбирает образцы из буфера воспроизведения. В идеале мы хотим чаще отбирать те переходы, из которых можно многому научиться. В качестве прокси для потенциала обучения, Prioritized Replay отбирает образцы переходов с вероятностью P(i) относительно последней обнаруженной абсолютной ошибки TD.

Приоритет перехода: $p_i = | \delta_i | + \epsilon$, где $\delta_i = y_i - Q(s_i, a_i)$

Вероятность выборки: $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$

Коррекция весов: $\omega_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$,

где α – гиперпараметр, контролирующий степень приоритезации, β – гиперпараметр для компенсации смещения (обычно растёт со временем)

Dueling networks

Dueling network – это архитектура нейронной сети, разработанная для RL на основе ценности. Она включает два потока вычислений: поток оценки состояния (V) и поток преимущества действий (A), совместно использующие сверточный кодер и объединенные специальным агрегатором.

$$Q(s,a;\theta) = V(s;\theta) + A(s,a;\theta) - \frac{1}{N_{actions}} \sum_{a'} A(s,a';\theta)$$

Выполнение работы.

Реализация алгоритмов

DQN

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.

Double DQN

Модифицируется train_step: сначала выбирается действие основной сетью и затем оно оценивается целевой сетью.

Prioritized Replay

Модифицируется ReplayBuffer: добавляется степень приоритезации (альфа), компенсация смещения (бета). Новым переходам даём максимальный приоритет. В конце шага приоритеты обновляются по значению абсолютной ошибки.

В train_step считаются значения абсолютной ошибки, и loss оценивается через взвешенный MSE.

Dueling networks

Модифицируется QNetwork: поток вычислений разделяется на два, затем объединяется специальным агрегатором (Q(s, a) = V(s) + A(s, a) - mean(A(s, a))).

Использованные параметры

Для всех тестов использовался один набор параметров:

- "gamma": 0.99,
- "epsilon": 0.7,
- "epsilon_decay": 0.955,
- "epsilon_min": 0.05,
- "layers": [128, 128],
- "num_steps": 200,
- "num_episodes": 300

Такие параметры выбраны отчасти экспериментально, отчасти по результатам выполнения первой практической работы ос сравнением работы в зависимости от изменения параметров.

Есть предположение, что с более глубокой и широкой сетью, а также большим числом эпизодов некоторые результаты стали бы лучше. Но предпочтение отдавалось экономии времени и ресурсов.

CartPole-v1

Было проведено сравнение алгоритмов в среде CartPole по показанием времени обучения (рис. 1), графика потерь (рис. 2) и наград (рис. 3).

```
300/300 [00:45<00:00,  6.56it/s]
300/300 [00:41<00:00,  7.29it/s]
300/300 [01:09<00:00,  4.31it/s]
300/300 [00:36<00:00,  8.13it/s]
```

Рисунок 1 – Время обучения (DQN→DDQN→ Prioritized Replay→ Dueling networks)

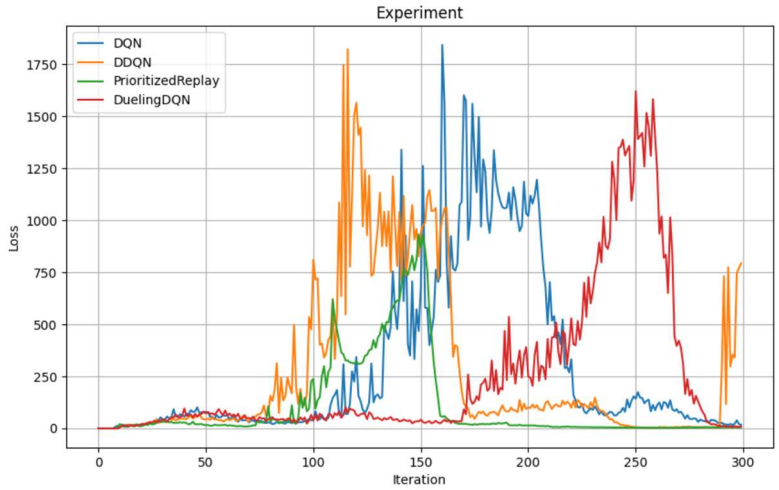

Рисунок 2 – График потерь CartPole-v1

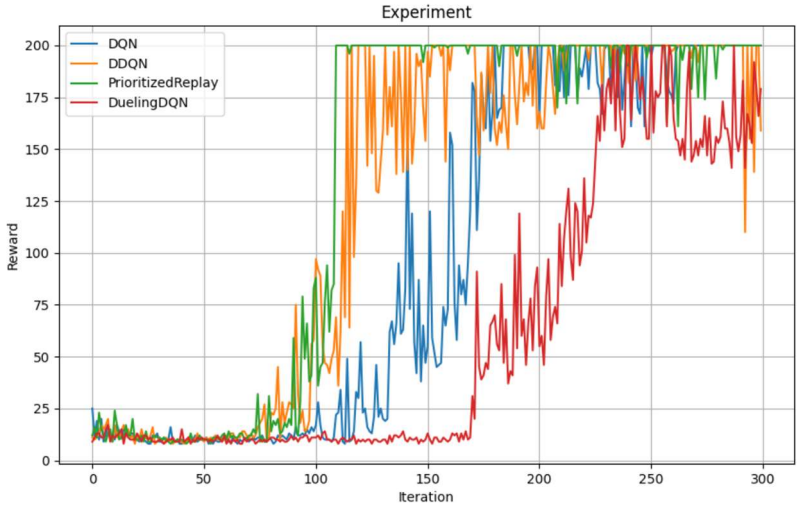

Рисунок 3 – График наград CartPole-v1

По результатам можно видеть, что все алгоритмы за 300 эпизодов смогли достигнуть максимальной награды, однако DuelingDQN достиг финальной отметки только на небольшом промежутке, а потом награды начали скакать вниз.

Быстрее всего процесс обучения прошли DDQN и DuelingDQN, медленнее всего – PrioritizedReplay. Однако PrioritizedReplay достиг финального результата быстрее всех, и он также оставался самым стабильным.

График потерь у всех алгоритмов примерно одинаковый, но с разным смещением, меньше всего амплитуда также у PrioritizedReplay.

LunarLander-v3

Было проведено сравнение алгоритмов в среде LunarLander по показанием времени обучения (рис. 4), графика потерь (рис. 5) и наград (рис. 6).

```
300/300 [01:14<00:00,  4.01it/s]
300/300 [01:22<00:00,  3.64it/s]
300/300 [02:01<00:00,  2.46it/s]
300/300 [01:46<00:00,  2.82it/s]
```

Рисунок 4 – Время обучения (DQN→DDQN→ Prioritized Replay→ Dueling networks)

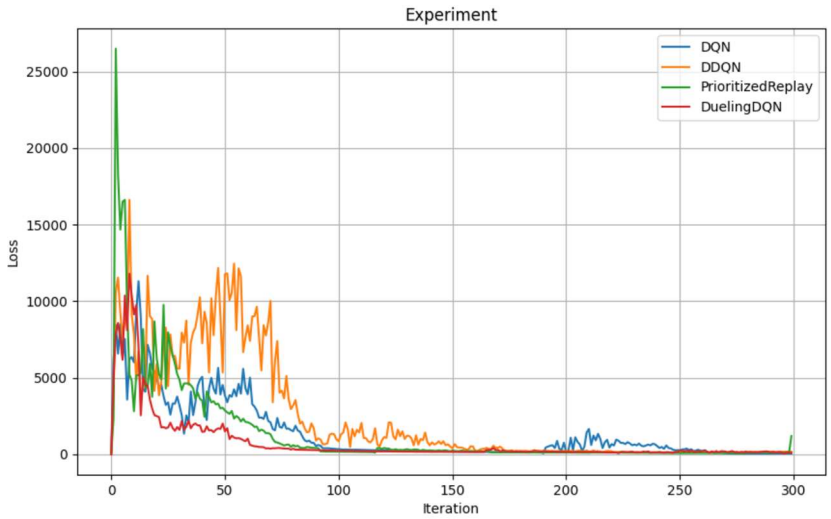

Рисунок 5 – График потерь LunarLander-v3

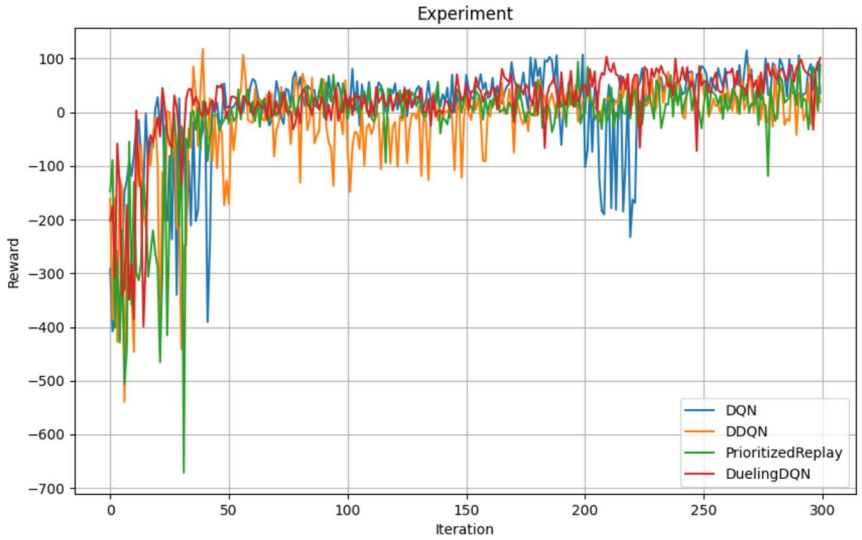

Рисунок 6 – График наград LunarLander-v3

По результатам можно видеть, что ни один из алгоритмов не достиг финального результата (200 очков), но иногда достигал 100 очков (скорее всего корабль был посажен). Можно, также, заметить, что агенты меньше врезаются в землю (-100 очков) в процессе обучения.

Быстрее всего отметки в 100 очков достиг DDQN, но он также был наиболее нестабильным, видны частые падения вниз.

Быстрее всего прошёл обучение стандартный DQN, медленнее всего – PrioritizedReplay.

Наиболее стабильными являются DuelingDQN и PrioritizedReplay.

MountainCar-v0

Было проведено сравнение алгоритмов в среде MountainCar по показанием времени обучения (рис. 7), графика потерь (рис. 8) и наград (рис. 9).

Изначально в среде даётся только пенальти за прошедшее время. Поэтому для получения каких-либо результатов была введена дополнительная система наград:

- Агент награждается за движение в сторону финиша
- Агент награждается за повышение скорости

- Агент наказывается за простой

```
300/300 [01:17<00:00,  3.85it/s]
300/300 [01:24<00:00,  3.53it/s]
300/300 [02:00<00:00,  2.49it/s]
300/300 [01:44<00:00,  2.86it/s]
```

Рисунок 7 – Время обучения (DQN→DDQN→ Prioritized Replay→ Dueling networks)

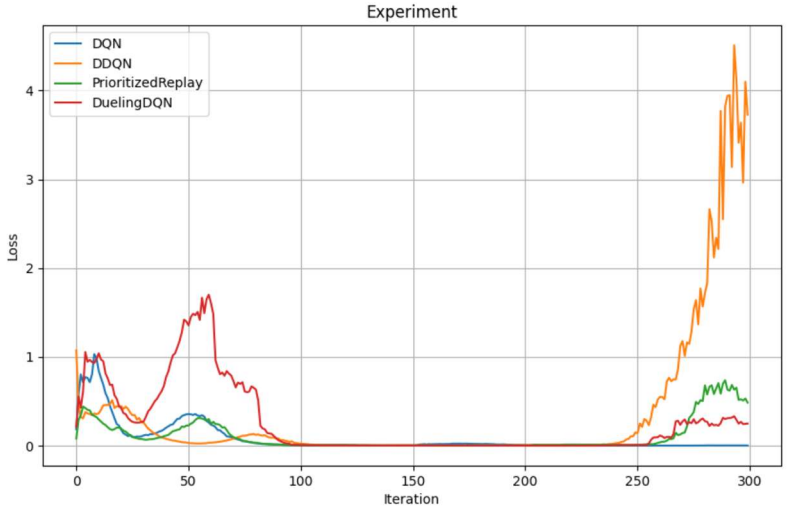

Рисунок 8 – График потерь MountainCar-v0

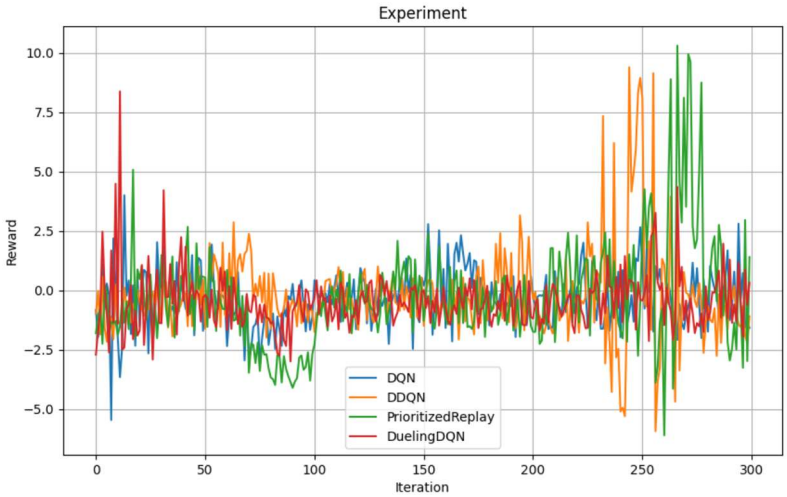

Рисунок 9 – График наград MountainCar-v0

По результатам можно видеть, что наибольшие награды у DDQN и Prioritized Replay. Так как в дополнительно указанных наградах не

указывалась награда за достижение финиша, трудно оценить кто смог ближе всего подобраться к финишу.

Быстрее всего прошёл обучение стандартный DQN, медленнее всего – PrioritizedReplay.

Наиболее стабильными являются DQN, у DDQN, наоборот, наблюдается большой скачок в потерях ближе к концу. Dueling networks показывает сильный скачок в потерях в начале, но после этого показывает хорошую стабильность.

Разработанный программный код см. в приложении А.

Выводы.

Были реализованы алгоритмы Double Q-learning, Prioritized replay, Dueling networks, и проведено их сравнение в окружениях LunarLander-v3 и Mountain-Car. Double DQN – улучшение DQN, уменьшающее переоценку Q-значений. Prioritized Replay – оптимизация буфера воспроизведения для более эффективного обучения. Dueling networks – архитектурное улучшение, разделяющее оценку состояния и преимущества действий.

В целом все алгоритмы показывают схожие результаты, можно выделить, что стандартный DQN выполняется быстрее всех, что неудивительно, так как остальные методы в какой-то мере усложняют процедуру. DDQN так же выполняется быстро, но в большинстве тестов показывает низкую стабильность. Prioritized Replay является самым медленным по результатам тестирования, но зачастую показывает более высокие и стабильные результаты. Dueling networks выполняется не так быстро, но в большей части случаев также показывает хорошие результаты и стабильность.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py:

```
import os
import gymnasium as gym
import matplotlib.pyplot as plt
from tqdm import trange

from DQN import DQNAgent
from DDQN import DDQNAgent
from PrioritizedReplay import PrioritizedReplayAgent
from DuelingDQN import DuelingDQNAgent

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

agents = [DQNAgent, DDQNAgent, PrioritizedReplayAgent, DuelingDQNAgent]
environments = [
    {"name": "CartPole-v1", "state_dim": 4, "action_dim": 2},
    {"name": "LunarLander-v3", "state_dim": 8, "action_dim": 4},
    {"name": "MountainCar-v0", "state_dim": 2, "action_dim": 3}
]


params = {
    "gamma": 0.99,
    "epsilon": 0.7,
    "epsilon_decay": 0.955,
    "epsilon_min": 0.05,
    "layers": [128, 128],
    "num_steps": 200,
    "num_episodes": 300,
}

def shaped_reward(state, action, original_reward, next_state):
    position = state[0]
    new_position = next_state[0]
    new_velocity = next_state[1]

    position_reward = 10 * (new_position - position) # Награда за движение в
сторону гола
    velocity_reward = 0.1 * abs(new_velocity) # Награда за высокую скорость
    action_penalty = -0.01 if action != 1 else 0 # Пенальти за отсутсвие действий

    return position_reward + velocity_reward + action_penalty

def train(agent, env, episodes, steps, reshape_reward=False):
    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)
            reward = shaped_reward(state, action, reward, next_state) if
reshape_reward else reward
```

```
            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()

def experiment(environment):
    env = gym.make(environment["name"])
    results = {}
    for agentClass in agents:
        agent = agentClass(environment["state_dim"], environment["action_dim"],
params["layers"], params["gamma"], params["epsilon"], params["epsilon_decay"])
        rewards,   losses   =   train(agent,   env,   params["num_episodes"],
params["num_steps"], environment["name"]=="MountainCar-v0")
        results[type(agent).__name__[:-5]] = (rewards, losses)
    return results

if __name__ == "__main__":
    print("Running CartPole experiment...")
    r_CartPole = experiment(environments[0])
    plot_results(r_CartPole, "Experiment", "CartPole")

    print("Running LunarLander experiment...")
    r_LunarLander = experiment(environments[1])
    plot_results(r_LunarLander, "Experiment", "LunarLander")

    print("Running MountainCar experiment...")
    r_MountainCar = experiment(environments[2])
    plot_results(r_MountainCar, "Experiment", "MountainCar")
```

Файл DQN.py:

```
import random
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque

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)
        self.action_dim = action_dim

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 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())
```

Файл DDQN.py:

```
import random
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque

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 DDQNAgent:
```

```
    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)
        self.action_dim = action_dim

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 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)

        # Double DQN: используем основную сеть для выбора действия, а target
сеть для оценки
        with torch.no_grad():
            next_actions = self.q_net(s2).argmax(1) # Выбираем действие
основной сетью
            target = r + self.gamma * self.target_net(s2).gather(1,
next_actions.unsqueeze(1)).squeeze(1) * (1 - d)

        q_vals = self.q_net(s).gather(1, a.unsqueeze(1)).squeeze(1)
        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())
```

Файл PrioritizedReplay.py:

```
import random
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque

class PrioritizedReplayBuffer:
```

```
    def __init__(self, capacity=10000, alpha=0.6, beta=0.4,
beta_increment=0.001):
        self.buffer = deque(maxlen=capacity)
        self.priorities = deque(maxlen=capacity)
        self.alpha = alpha  # степень приоритезации (0 = равномерно)
        self.beta = beta  # компенсация смещения
        self.beta_increment = beta_increment
        self.max_priority = 1.0  # начальный приоритет для новых переходов

    def push(self, *transition):
        self.buffer.append(transition)
        self.priorities.append(self.max_priority)    # новым переходам даём
максимальный приоритет

    def sample(self, batch_size):
        priorities = np.array(self.priorities)
        probs = priorities ** self.alpha
        probs /= probs.sum()

        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        samples = [self.buffer[i] for i in indices]

        # Importance sampling weights
        weights = (len(self.buffer) * probs[indices]) ** (-self.beta)
        weights /= weights.max()  # нормализация

        state, action, reward, next_state, done = zip(*samples)
        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),
            torch.tensor(indices, dtype=torch.long),
            torch.tensor(weights, dtype=torch.float32),
        )

    def update_priorities(self, indices, td_errors):
        for idx, error in zip(indices, td_errors):
            self.priorities[idx] = (abs(error) + 1e-5)  # обновляем приоритеты
        self.max_priority = max(self.priorities)   # обновляем максимальный
приоритет

    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 PrioritizedReplayAgent:
```

```
    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 = PrioritizedReplayBuffer() # <-- Заменяем на
PrioritizedReplayBuffer
        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)
        self.action_dim = action_dim

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 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, indices, weights = self.buffer.sample(128) # <--
Добавляем weights
        s, a, r, s2, d, weights = s.to(self.device), a.to(self.device),
r.to(self.device), s2.to(self.device), d.to(self.device),
weights.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)

        td_errors = (target - q_vals).abs().detach().cpu().numpy() # для
обновления приоритетов
        loss = (weights * (q_vals - target) ** 2).mean()  # взвешенная MSE

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

        self.buffer.update_priorities(indices, td_errors) # обновляем
приоритеты
        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())
```

Файл DuelingDQN:

```
import random
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque
```

```
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 DuelingQNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, layers):
        super().__init__()
        # Общая часть сети
        self.feature_net = nn.Sequential(
            nn.Linear(input_dim, layers[0]),
            nn.ReLU(),
        )
        # Ветка для V(s)
        self.value_stream = nn.Sequential(
            nn.Linear(layers[0], layers[1]),
            nn.ReLU(),
            nn.Linear(layers[1], 1),
        )
        # Ветка для A(s, a)
        self.advantage_stream = nn.Sequential(
            nn.Linear(layers[0], layers[1]),
            nn.ReLU(),
            nn.Linear(layers[1], output_dim),
        )

    def forward(self, x):
        features = self.feature_net(x)
        value = self.value_stream(features)
        advantages = self.advantage_stream(features)

        # Q(s, a) = V(s) + A(s, a) - mean(A(s, a))
        return value + (advantages - advantages.mean(dim=-1, keepdim=True))

class DuelingDQNAgent:
    def __init__(self, state_dim, action_dim, layer_cfg, gamma, epsilon,
epsilon_decay):
        self.q_net = DuelingQNetwork(state_dim, action_dim, layer_cfg)  # <--
Заменяем на DuelingQNetwork
        self.target_net = DuelingQNetwork(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)
        self.action_dim = action_dim

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 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())
```