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

ОТЧЕТ

по исследовательскому проекту
по дисциплине «Обучение с подкреплением»
Тема: Реализация и сравнение версий DQN Prioritized replay, Dueling networks, Noisy Nets для сред LunarLander-v3 и Mountain-Car

Студент гр. 0306 Кумаритов А.О.

Преподаватель _____ Глазунов. С.А.

_____

Санкт-Петербург
2025

Задание:

Необходимо реализовать и сравнить между собой следующие версии DQN:

Prioritized replay

Dueling networks

Noisy Nets

Задания для эксперимента:

Сравнить версии в средах:

LunarLander-v3

MountainCar-v0

Описание среды LunarLander-v3:

Action space состоит из числа, принимающего 4 значения:

0 - ничего не делать

1 - запустить двигатель левой ориентации

2 - запустить основной двигатель

3 - запустить двигатель правой ориентации

Observation space состоит из 8 чисел:

Координаты модуля X и Y

Линейная скорость по X и Y

Угол наклона

Угловая скорость

И два флага, определяющие касание левой и правой части модуля поверхности

Rewards:

Для каждого шага

увеличивается/уменьшается, чем ближе/дальше посадочный модуль к посадочной площадке

увеличивается/уменьшается, чем медленнее/быстрее движется посадочный модуль

уменьшается, чем больше наклонен посадочный модуль

увеличивается на 10 очков за каждую ногу, которая находится в контакте с землей

уменьшается на 0,03 очка каждый кадр, когда запускается боковой двигатель

уменьшается на 0,3 очка каждый кадр, когда запускается основной двигатель

Эпизод получает дополнительную награду в размере -100 или +100 очков за падение или безопасную посадку соответственно

Эпизод считается решением, если он набирает не менее 200 очков.

Starting state - модуль стартует в верхнем центре кадра, к центру массы приложена случайная сила

Конец эпизода в трёх случаях:

Модуль разбился

Посадочный модуль выходит за пределы кадры

Посадочный модуль не управляется

Описание среды MountainCar-v0:

Action space состоит из числа, принимающего 3 значения:

0 - движение влево

1 - бездействие

2 - движение вправо

Observation space состоит из 2 чисел:

Car position - позиция машины по оси X, значения от -1.2 до 0.6

Car velocity - скорость машины, значение -0.07 до 0.07

Rewards негативный равен -1 за каждый шаг

Starting state - position присваивается случайное значение от -0.6 до -0.4, velocity присваивается 0.

Конец эпизода в двух случаях:

Car position больше или равен 0.5

Длительность эпизода равна 200

Также алгоритмы были запущены на среде CartPole-v1 из 1 лабораторной работы.

Описание среды CartPole-v1:

Action space состоит из числа, принимающего два значения:

0 - движение каретки налево

1 - движение каретки направо

Observation space состоит из 3 чисел:

Cart position - значения от -4.8 до 4.8

Cart velocity - значения от минус бесконечности до плюс бесконечности

Pole angle - от -0.418 rad (-24 градуса) до 0.418 rad (24 градуса)

Pole angular velocity - значения от минус бесконечности до плюс бесконечности

Rewards +1 за каждый шаг, включая терминальный.

Starting state - всем переменным из observation space присваивается значение от -0.05 до 0.05.

Терминальный шаг наступает в трёх случаях:

Pole angle меньше -12 градусов или больше 12 градусов

Cart position меньше -2.4 или больше 2.4

Номер эпизода больше 500.

Описание модификаций алгоритмов DQN:

Базовое описание алгоритма представлено на рисунке 1.

Алгоритм 15: Deep Q-learning (DQN)

Гиперпараметры: B — размер мини-батчей, K — периодичность апдейта таргет-сети, $\varepsilon(t)$ — стратегия исследования, Q — нейросетка с параметрами θ, SGD-оптимизатор

Инициализировать θ произвольно

Положить $\theta^- := \theta$
Пронаблюдать s_0
На очередном шаге t:

1. выбрать a_t случайно с вероятностью $\varepsilon(t)$, иначе $a_t := \underset{a_t}{\operatorname{argmax}}\, Q_\theta(s_t, a_t)$
2. пронаблюдать r_t, s_{t+1}, $\mathbf{done}_{t+1}$
3. добавить пятёрку $(s_t, a_t, r_t, s_{t+1}, \mathbf{done}_{t+1})$ в реплей буфер
4. засэмплировать мини-батч размера B из буфера
5. для каждого перехода $\mathbb{T} = (s, a, r, s', \mathbf{done})$ посчитать таргет:
$$y(\mathbb{T}) := r + \gamma(1 - \mathbf{done}) \max_{a'} Q_{\theta^-}(s', a')$$
6. посчитать лосс:
$$\mathbf{Loss}(\theta) := \frac{1}{B} \sum_{\mathbb{T}} (Q_\theta(s, a) - y(\mathbb{T}))^2$$
7. сделать шаг градиентного спуска по θ, используя $\nabla_\theta \mathbf{Loss}(\theta)$
8. если $t \bmod K = 0$: $\theta^- \leftarrow \theta$

Рис. 1 - алгоритм Deep Q-learning (DQN)

Prioritized replay

В этой модификации алгоритма вместо обычного буфера используется буфер с приоритетами ошибок. Приоритет при выборе из буфера отдаётся тем переходам, которые принесли наибольшую ошибку предсказания (td_error).

Dueling networks

В этой модификации для вычисления Q функции используется помимо feature модели ещё value и advantage. Первая оценивает состояние, а вторая насколько полезно каждое действие. И далее происходит объединение по формуле:

$$Q = Value + Advantage - Mean(Advantage.dim(1))$$

Noisy Nets

Вместо политики жадности (epsilon) используется вызываемый шум, регулируемый коэффициентом std_init - начальным количеством шума в реализации NoisyLinear слоя.

В качестве основы использовался алгоритм, реализованный в первой лабораторной работе. В реализации следующие параметры:

BATCH_SIZE - количество переходов, которые выбираются из памяти.

GAMMA - коэффициент дисконтирования.

EPS_START - начальное значение эпсилон.

EPS_END - конечное значение эпсилон.

EPS_DECAY - скорость экспоненциального затухания, чем выше, тем медленнее затухание.

TAU - скорость обновления целевой сети.

LR - скорость обучения оптимизатора.

num_episodes - количество эпизодов.

hidden_size - количество скрытых слоёв в нейронных сетях.

training_criteria - порог среднего значения для оценки early stopping.

training_criteria_length - количество значений для оценки early stopping.

std_init - количество шума в реализации NoisyDQN_Net.

alpha - степень приоритета в Prioritized replay DQN.

beta - коэффициент важности весов для оценки в Prioritized replay DQN.

beta_increment - инкримент коэффициента beta из прошлого пункта.

В реализации используем несколько классов:

Transition - именованный кортеж, хранящий переход в среде: соответствие состоянию и действию к следующему состоянию и награде.

ReplayMemory - буфер, хранящий в себе ограниченное количество наблюдаемых при взаимодействии со средой переходов. В нём реализован метод sample для выбора случайных BATCH_SIZE элементов.

PrioritizedReplayMemory - аналогичный буфер для Prioritized replay DQN.

DQN_Net - базовая конфигурация нейронной сети.

DuelingDQN_Net - конфигурация нейронной сети для Dueling networks DQN.

NoisyDQN_Net - конфигурация нейронной сети для Noisy Nets DQN.

DQN - класс, реализующий инициализацию и обучение.

DQN_ENV - Enum со средами (DQN_ENV.CART_POLE, DQN_ENV.MOUNTAIN_CAR, DQN_ENV.LUNAR_LANDER).

DQN_MODE - Enum с модификациями алгоритма DQN (DQN_MODE.BASE, DQN_MODE.PRIORITIZED, DQN_MODE.DUELING, DQN_MODE.NOISY).

Так же для всех алгоритмов и сред был добавлен механизм early stopping. Значения порога наград взяты из таблицы https://github.com/openai/gym/wiki/Table-of-environments.

Выполнение экспериментов.

Запуск алгоритмов на CartPole-v1:

В рамках эксперимента в данной среде был запущен базовый алгоритм DQN, а также 3 его модификации, такие как Prioritized replay, Dueling networks, Noisy Nets. Критерий обучения в этой среде равен 475.

Базовый алгоритм достиг критерия early stopping на 303 эпизоде.

Prioritized replay DQN - на 281 эпизоде.

Noisy Nets DQN - на 181 эпизоде.

Dueling networks DQN - на 163 эпизоде.

Подробные графики запусков показаны на рисунке 2:

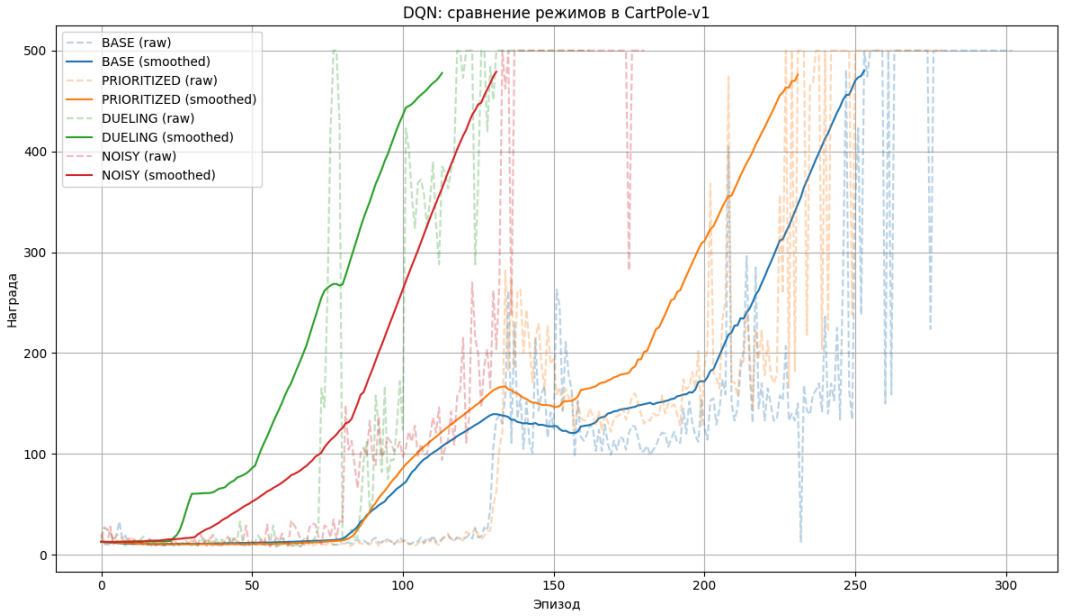

Рис. 2 - запуск реализованных модификаций на CartPole-v1

Все модификации достигли критерия обучения раньше базового алгоритма. Использование Dueling networks DQN и Noisy Nets DQN ускоряет обучения в эпизодах почти в два раза.

Запуск алгоритмов на MountainCar-v0:

В рамках эксперимента в данной среде был запущен базовый алгоритм DQN, а также 3 его модификации, такие как Prioritized replay, Dueling networks, Noisy Nets. Критерий обучения в этой среде равен -110.

Базовый алгоритм достиг критерия early stopping на 1026 эпизоде.

Dueling networks DQN - на 256 эпизоде.

Prioritized replay DQN и Noisy Nets DQN не достигли критерия за 1500 эпизодов.

Подробные графики запусков показаны на рисунке 3:

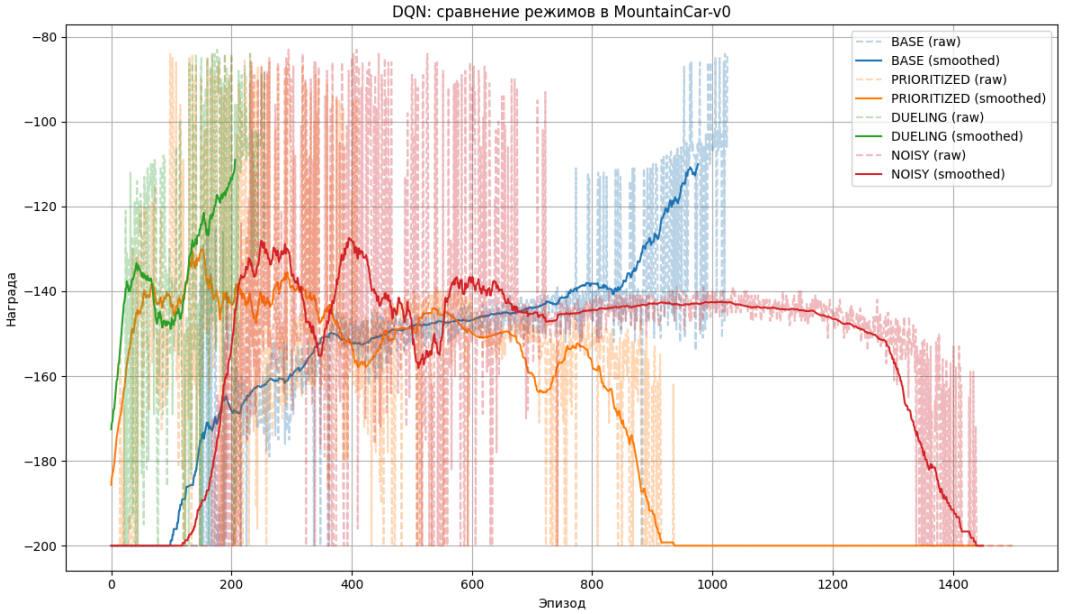

Рис. 3 - запуск реализованных модификаций на MountainCar-v0

С экспериментом справился базовый алгоритм и Dueling networks DQN, который дал ускорение в эпизодах в 5 раз. Модификации Prioritized replay DQN и Noisy Nets DQN показывали лучшую динамику по сравнению с базовым в начале обучения, но позже начали получать минимальную награду за эпизод, что может говорить о нестабильности обучения этих модификаций в данной среде.

Запуск алгоритмов на LunarLander-v3:

В рамках эксперимента в данной среде был запущен базовый алгоритм DQN, а также 3 его модификации, такие как Prioritized replay, Dueling networks, Noisy Nets. Критерий обучения в этой среде равен 200.

Базовый алгоритм достиг критерия early stopping на 204 эпизоде.

Dueling networks DQN - на 297 эпизоде.

Prioritized replay DQN - на 241 эпизоде.

Noisy Nets DQN - на 128 эпизоде.

Подробные графики запусков показаны на рисунке 4:

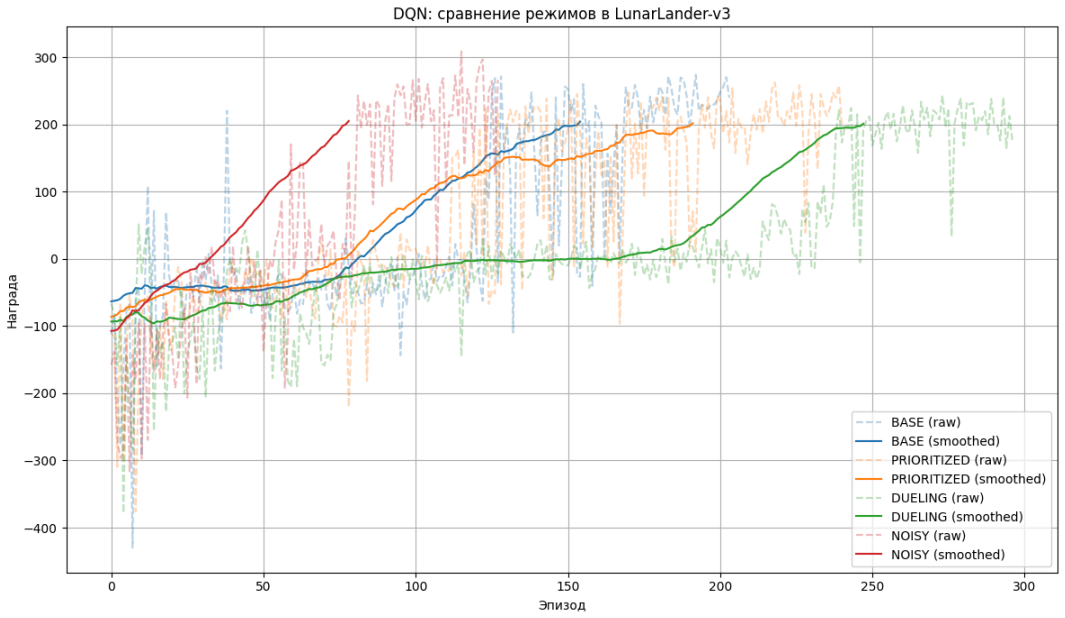

Рис. 4 - запуск реализованных модификаций на LunarLander-v3

Все алгоритмы достигли критерия сравнения. Модификация Noisy Nets DQN ускорило обучение по эпизодам почти в 2 раза по сравнению с базовым.

Выводы.

Был реализован базовый DQN и три его модификации: Prioritized replay, Dueling networks, Noisy Nets. Были проведены сравнения этих алгоритмов в средах: CartPole-v1, MountainCar-v0, LunarLander-v3. Базовый DQN и Dueling networks DQN справились с заданиями во всех средах. Noisy Nets DQN и Prioritized replay DQN показали ускорение, но достигли критерия обучения в среде MountainCar-v0.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ.

Исходный код main.py

```
import matplotlib.pyplot as plt
import numpy as np

from dqn import DQN, DQN_MODE, DQN_ENV



def run_and_plot(envs, modes, filename_prefix):
  plt.figure(figsize=(12, 7))
  color_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']

  for i, env in enumerate(envs):
    plt.clf()
    for j, mode in enumerate(modes):
      print(f"Обучение в {env.value}, режим {mode.name}")
      dqn = DQN(env_name=env, mode_name=mode)
      episode_rewards = dqn.train_dqn()
            print(f"Обучение в {env.value}, режим {mode.name} завершено")

      color = color_cycle[j % len(color_cycle)]
        plt.plot(range(len(episode_rewards)), episode_rewards, linestyle='--', alpha=0.3,
          color=color, label=f"{mode.name} (raw)")

        smoothed = np.convolve(episode_rewards, np.ones(50) / 50, mode='valid')
```

```
            plt.plot(range(len(smoothed)), smoothed, linestyle='-', color=color,
                     label=f"{mode.name} (smoothed)")

        plt.title(f"DQN: сравнение режимов в {env.value}")
        plt.xlabel("Эпизод")
        plt.ylabel("Награда")
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.savefig(f"{filename_prefix}_{env.value}.png")


    def run_all_envs():
        envs = [DQN_ENV.CART_POLE, DQN_ENV.MOUNTAIN_CAR, DQN_ENV.LUNAR_LANDER]
            modes = [DQN_MODE.BASE, DQN_MODE.PRIORITIZED, DQN_MODE.DUELING, DQN_MODE.NOISY]
        run_and_plot(envs, modes, "dqn_comparison")


    def main(seed=42):
      np.random.seed(seed)
      run_all_envs()


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

Исходный код dqn.py

```
import math
import random
from enum import Enum
from itertools import count

import gymnasium as gym
import numpy as np
import torch
from torch import nn
from torch import optim

from nets import DQN_Net, DuelingDQN_Net, NoisyDQN_Net
from prioritized_replay_memory import PrioritizedReplayMemory
from replay_memory import ReplayMemory
from transition import Transition



class DQN_ENV(Enum):
  CART_POLE = "CartPole-v1"
  LUNAR_LANDER = "LunarLander-v3"
  MOUNTAIN_CAR = "MountainCar-v0"



class DQN_MODE(Enum):
  BASE = 1
  PRIORITIZED = 2
  DUELING = 3
  NOISY = 4
```

```
class DQN:
    def __init__(self, env_name: DQN_ENV, mode_name: DQN_MODE, seed=42):
        self.env = gym.make(env_name.value)
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.mode = mode_name
        self.env_name = env_name
        self.batch_size = 128
        self.gamma = 0.99
        self.tau = 0.005
        self.lr = 1e-4
        self.hidden_size = 256
        self.state_min = torch.tensor(self.env.observation_space.low, device=self.device)
        self.state_max = torch.tensor(self.env.observation_space.high, device=self.device)
        self.training_criteria_length = 50

        if self.env_name == DQN_ENV.MOUNTAIN_CAR:
            self.std_init = 0.1
            self.eps_start = 0.95
            self.eps_end = 0.01
            self.eps_decay = 1500
            self.training_criteria = -110
            self.num_episodes = 1500
            self.replay_memory_size = 20000
```

```
            self.beta = 0.4
            self.beta_increment = (1 - self.beta) / (self.num_episodes * 200)
            self.goal_position = 0.5
                    self.mid_point = self.state_min[0] + (self.state_max[0] -
self.state_min[0]) / 2
        elif self.env_name == DQN_ENV.LUNAR_LANDER:
            self.std_init = 0.1
            self.eps_start = 0.9
            self.eps_end = 0.01
            self.eps_decay = 1500
            self.training_criteria = 200
            self.num_episodes = 500
            self.replay_memory_size = 30000
            self.beta = 0.4
            self.beta_increment = (1 - self.beta) / (self.num_episodes * 200)
        elif self.env_name == DQN_ENV.CART_POLE:
            self.std_init = 0.2
            self.eps_start = 0.5
            self.eps_end = 0.05
            self.eps_decay = 500
            self.training_criteria = 475
            self.num_episodes = 1000
            self.replay_memory_size = 10000
            self.beta = 0.4
            self.beta_increment = (1 - self.beta) / (self.num_episodes * 200)

        torch.manual_seed(seed)
        random.seed(seed)
```

```
        if self.mode == DQN_MODE.PRIORITIZED:
                                                    self.memory =
PrioritizedReplayMemory(self.replay_memory_size, beta=self.beta,
                                            beta_increment=self.beta_increment)
        else:
            self.memory = ReplayMemory(self.replay_memory_size)

        self.env.action_space.seed(seed)
        self.env.observation_space.seed(seed)
        n_actions = self.env.action_space.n
        state, _ = self.env.reset(seed=seed)
        n_observations = len(state)

        if self.mode == DQN_MODE.DUELING:
            net_class = DuelingDQN_Net
        elif self.mode == DQN_MODE.NOISY:
            net_class = NoisyDQN_Net
            self.lr = 1e-3
                elif self.mode == DQN_MODE.BASE or self.mode ==
DQN_MODE.PRIORITIZED:
            net_class = DQN_Net
        else:
            raise ValueError(f"Unknown mode: {self.mode}. Supported modes
are: {list(DQN_MODE)}")

        if self.mode == DQN_MODE.NOISY:
                    self.policy_net = net_class(n_observations, n_actions,
self.hidden_size, self.std_init).to(self.device)
```

```
                self.target_net = net_class(n_observations, n_actions, self.hidden_size, self.std_init).to(self.device)
            else:
                self.policy_net = net_class(n_observations, n_actions, self.hidden_size).to(self.device)
                self.target_net = net_class(n_observations, n_actions, self.hidden_size).to(self.device)
            self.target_net.load_state_dict(self.policy_net.state_dict())

            self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=self.lr, amsgrad=True)
            self.steps_done = 0

        def optimize_model(self):
            if len(self.memory) < self.batch_size:
                return

            transitions, indices, weights = self.memory.sample(self.batch_size)

            batch = Transition(*zip(*transitions))

            non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                                    batch.next_state)), device=self.device, dtype=torch.bool)
            non_final_next_states = torch.cat([s for s in batch.next_state
                                               if s is not None])
            state_batch = torch.cat(batch.state)
            action_batch = torch.cat(batch.action)
            reward_batch = torch.cat(batch.reward)
```

```
                    state_action_values = self.policy_net(state_batch).gather(1,
action_batch)

            next_state_values = torch.zeros(self.batch_size, device=self.device)
            if self.mode == DQN_MODE.NOISY:
                self.target_net.sample_noise()
            with torch.no_grad():
                                        next_state_values[non_final_mask] =
self.target_net(non_final_next_states).max(1).values
                expected_state_action_values = (next_state_values * self.gamma) +
reward_batch

            criterion = nn.SmoothL1Loss(reduction="none")
                            individual_losses = criterion(state_action_values,
expected_state_action_values.unsqueeze(1))

            if self.mode == DQN_MODE.PRIORITIZED:
                            weights = torch.tensor(weights, device=self.device,
dtype=torch.float32).unsqueeze(1)
                loss = (weights * individual_losses).mean()
            else:
                loss = individual_losses.mean()

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

```
            if self.mode == DQN_MODE.PRIORITIZED:
                with torch.no_grad():
                        td_errors = (expected_state_action_values.unsqueeze(1) -
state_action_values).squeeze().abs().tolist()
                    self.memory.update_td_errors(indices, td_errors)

        def select_action(self, state):
            if self.mode == DQN_MODE.NOISY:
                self.policy_net.sample_noise()
                with torch.no_grad():
                    return self.policy_net(state).max(1).indices.view(1, 1)
            sample = random.random()
            eps_threshold = self.eps_end + (self.eps_start - self.eps_end) * \
                        math.exp(-1. * self.steps_done / self.eps_decay)
            self.steps_done += 1
            if sample > eps_threshold:
                with torch.no_grad():
                    return self.policy_net(state).max(1).indices.view(1, 1)
            else:
                            return   torch.tensor([[self.env.action_space.sample()]],
device=self.device, dtype=torch.long)

        def normalize_state(self, state):
            if self.env_name == DQN_ENV.MOUNTAIN_CAR:
                state = (state - self.state_min) / (self.state_max - self.state_min)
            return state

        def get_reward_mountain_car(self, observation):
            if observation[0] >= self.goal_position:
```

```
            return 0
        if observation[0] > self.mid_point:
            return (-1 / (self.mid_point - self.goal_position)) * (observation[0]
- self.goal_position)
        return -1

    def early_stopping(self, episode_rewards):
        if len(episode_rewards) < self.training_criteria_length:
            return False
        recent_rewards = episode_rewards[-self.training_criteria_length:]
        return np.mean(recent_rewards) >= self.training_criteria

    def train_dqn(self):
        self.policy_net.train()
        self.target_net.train()

        episode_rewards = []
        log_interval = max(1, self.num_episodes // 20)
        for episode in range(1, self.num_episodes + 1):
            state, _ = self.env.reset()
            state = torch.tensor(state, dtype=torch.float32, device=self.device)
            state = self.normalize_state(state).unsqueeze(0)
            total_reward = 0
            for _ in count():
                action = self.select_action(state)
                                observation, reward, terminated, truncated, _ =
self.env.step(action.item())
                total_reward += reward
                if self.env_name == DQN_ENV.MOUNTAIN_CAR:
```

```
                reward = self.get_reward_mountain_car(observation)
            reward = torch.tensor([reward], device=self.device)
            done = terminated or truncated

            if done:
                next_state = None
            else:
                    next_state = torch.tensor(observation, dtype=torch.float32,
device=self.device)
                next_state = self.normalize_state(next_state).unsqueeze(0)

            self.memory.push(state, action, next_state, reward, 1.0)
            state = next_state
            self.optimize_model()

                        for target, policy in zip(self.target_net.parameters(),
self.policy_net.parameters()):
                    target.data.copy_(self.tau * policy.data + (1 - self.tau) *
target.data)

            if done:
                episode_rewards.append(total_reward)
                break

        if self.early_stopping(episode_rewards):
            print(
                f"Early stopping at episode {episode} with average reward
{np.mean(episode_rewards[-self.training_criteria_length:]):.2f}")
            break
```

```
            if episode % log_interval == 0:
                percent_done = (episode / self.num_episodes) * 100
                avg_reward = np.mean(episode_rewards[-log_interval:])
                print(
                                        f"Training progress: {percent_done:.0f}%
({episode}/{self.num_episodes} episodes), average reward {avg_reward:.2f}")

            self.env.close()
            self.memory.clear()
            return episode_rewards
```

Исходный код nets.py

```
import math

import torch
import torch.nn.functional as F
from torch import nn


class DQN_Net(nn.Module):
  def __init__(self, n_observations, n_actions, hidden_size):
      super(DQN_Net, self).__init__()
      self.model = nn.Sequential(
          nn.Linear(n_observations, hidden_size),
          nn.ReLU(),
          nn.Linear(hidden_size, hidden_size),
          nn.ReLU(),
          nn.Linear(hidden_size, n_actions)
```

```
        )

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


class DuelingDQN_Net(nn.Module):
    def __init__(self, n_observations, n_actions, hidden_size):
        super().__init__()
        self.feature = nn.Sequential(
            nn.Linear(n_observations, hidden_size),
            nn.ReLU(),
            nn.LayerNorm(hidden_size)
        )
        self.value_stream = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1)
        )
        self.advantage_stream = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, n_actions)
        )

    def forward(self, x):
        x = self.feature(x)
        value = self.value_stream(x)
        advantage = self.advantage_stream(x)
```

```
            return value + advantage - advantage.mean(dim=1, keepdim=True)


        class NoisyLinear(nn.Module):
          def __init__(self, in_features, out_features, std_init=0.1):
            super().__init__()

            self.in_features = in_features
            self.out_features = out_features
            self.std_init = std_init

                        self.weight_mu   =   nn.Parameter(torch.empty(out_features,
in_features))
                      self.weight_sigma   =   nn.Parameter(torch.empty(out_features,
in_features))
                    self.register_buffer('weight_epsilon',  torch.empty(out_features,
in_features))

            self.bias_mu = nn.Parameter(torch.empty(out_features))
            self.bias_sigma = nn.Parameter(torch.empty(out_features))
            self.register_buffer('bias_epsilon', torch.empty(out_features))

            self.reset_parameters()
            self.sample_noise()

          def reset_parameters(self):
            mu_range = 1 / math.sqrt(self.weight_mu.size(1))

            self.weight_mu.data.uniform_(-mu_range, mu_range)
```

```
                                        self.weight_sigma.data.fill_(self.std_init /
math.sqrt(self.weight_mu.size(1)))

            self.bias_mu.data.uniform_(-mu_range, mu_range)
                                            self.bias_sigma.data.fill_(self.std_init /
math.sqrt(self.bias_mu.size(0)))

        def _scale_noise(self, size):
            x = torch.rand(size)
            return x.sign() * x.abs().sqrt()

        def sample_noise(self):
            eps_in = self._scale_noise(self.in_features)
            eps_out = self._scale_noise(self.out_features)

            self.weight_epsilon.copy_(eps_out.ger(eps_in))
            self.bias_epsilon.copy_(self._scale_noise(self.out_features))

        def forward(self, x):
            if self.training:
                            weight = self.weight_mu + self.weight_sigma *
self.weight_epsilon
                bias = self.bias_mu + self.bias_sigma * self.bias_epsilon
            else:
                weight = self.weight_mu
                bias = self.bias_mu
            return F.linear(x, weight, bias)
```

```
class NoisyDQN_Net(nn.Module):
  def __init__(self, n_observations, n_actions, hidden_size, std_init):
    super(NoisyDQN_Net, self).__init__()

    self.linear = nn.Linear(n_observations, hidden_size)
    self.noisy1 = NoisyLinear(hidden_size, hidden_size, std_init)
    self.noisy2 = NoisyLinear(hidden_size, n_actions, std_init)

  def forward(self, x):
    x = F.relu(self.linear(x))
    x = F.relu(self.noisy1(x))
    return self.noisy2(x)

  def sample_noise(self):
    self.noisy1.sample_noise()
    self.noisy2.sample_noise()
```

Исходный код replay_memory.py

```
import random
from collections import deque

from transition import Transition


class ReplayMemory(object):
  def __init__(self, capacity):
    self.memory = deque([], maxlen=capacity)

  def push(self, *args):
```

```
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size), None, None

    def __len__(self):
        return len(self.memory)

    def clear(self):
        self.memory.clear()

    def update_td_errors(self, indices, td_errors):
        pass
```

Исходный код prioritized_replay_memory.py

```
from collections import deque

import numpy as np

from transition import Transition


class PrioritizedReplayMemory:
    def __init__(self, capacity, alpha=0.6, beta=0.4, beta_increment=0.001):
        self.alpha = alpha
        self.beta = beta
        self.beta_increment = beta_increment
        self.capacity = capacity
```

```
        self.memory = deque([], maxlen=capacity)
        self.max_priority = 1.0
        self.min_priority = 0.01

    def push(self, *args):

self.memory.append(Transition(*args)._replace(td_error=self.max_priority))

    def sample(self, batch_size):
        self.beta = min(1.0, self.beta + self.beta_increment)
         priorities = np.array([(abs(experience.td_error) + 1e-5) ** self.alpha
for experience in self.memory])
        probabilities = priorities / priorities.sum()
        actual_batch_size = min(batch_size, len(self.memory))
        sample_indices = np.random.choice(
            range(len(self.memory)), size=actual_batch_size, p=probabilities)
             weights = (len(self.memory) * probabilities[sample_indices]) **
(-self.beta)
        weights /= weights.max()
             return [self.memory[i] for i in sample_indices], sample_indices,
weights

    def update_td_errors(self, indices, td_errors):
        current_max = 0
        for idx, td_error in zip(indices, td_errors):
            td_error_abs = max(min(abs(td_error), 5.0), self.min_priority)
                                                    self.memory[idx]      =
self.memory[idx]._replace(td_error=td_error_abs)
            if td_error_abs > current_max:
```

```
            current_max = td_error_abs
        if current_max > self.max_priority:
            self.max_priority = current_max

    def __len__(self):
        return len(self.memory)

    def clear(self):
        self.memory.clear()
```

Исходный код transition.py

```
from collections import namedtuple

Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward', 'td_error'))
```