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. 0306 _____ Парамонов В.В.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург
2025

Цель работы.

Ознакомиться с DQN и реализовать её с помощью библиотеки pytorch для решения задачи 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.

1) Реализация DQN.

Для начала кратко опишем среду, в которой происходит обучение модели (CartPole-v1). Состояние среды – 4 вещественных числа, которые отвечают за параметры каретки и столба, стоящего на каретке. Задача – удерживать равновесие столба на каретке с помощью 2-х видов действий (либо движение каретки влево, либо вправо). Среда входит в терминальное состояние, либо при достижении 500 шагов, либо в случае, если упал столб или каретка уехала слишком далеко от начальной позиции. За каждый прожитый шаг следует награда в 1, получаем, что чем дольше живет модель, тем большую награду она получит.

В качестве Q-функции в DQN используется нейронная сеть, базовый вид реализованной нейронной сети выглядит как показано на рисунке 1. Она состит всего из 3-х слоев и активацией в виде ReLU.

```python
# Определяем нейронную сеть для аппроксимации Q-функции.
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        return self.layer3(x)
```

Рисунок 1 – Базовая архитектура нейронной сети

Эта архитектура используется как для policy_net (последний вариант обученной модели для предсказания получаемой награды от переданного состояния и действия), так и для target_net (чуть отстающий вариант обученной модели, необходимый для обучения и обновления весов).

Учатся данные модели благодаря использованию *ReplayMemory* – циклического буфера фиксированной длины (в нашем случае 10000), который сохраняет соответствие пар (состояние, действие) -> (награда, следующее состояние), которые мы получаем в ходе взаимодействия со средой.

- Когда количество сохраненных состояний в *ReplayMemory* станет равно или больше размера батча обучения, то из данного буфера будут выбираться случайные данные, количество которых равно размеру батча.
- После чего пары (состояние, действие) будут отправлены в *policy_net* для получения оценки возможной награды.
- Затем с помощью *target_net* вычисляется оценка награды, которую получит агент от пары (следующее состояние, действие) для всех действий возможных в данной среде. Из полученных оценок выбирается максимальная.

- С помощью уравнения Беллмана вычисляется более близкая к реальности оценка возможной награды для пары (состояние, действие), далее вычисляется функция потерь Хубера от двух оценок наград.
- Происходит обратное распространение ошибки и обновляются веса *policy_net*.
- Обновление весов *target_net* происходит как замена части весов *target_net* на текущие веса *policy_net* (soft update).

Данные действия по обучению происходят на каждом шаге взаимодействия со средой. Когда среда достигает терминального состояния, она перезапускается для продолжения обучения (было выбрано значение числа перезапусков 600).

Базовые используемые значений гиперпараметров DQN представлены на рисунке 2:

```
# Гиперпараметры.
BATCH_SIZE = 128  # размер мини-батча.
GAMMA = 0.99  # коэффициент дисконтирования.
EPS_START = 0.9  # начальное значение ε для ε-жадной стратегии.
EPS_END = 0.05  # минимальное значение ε.
EPS_DECAY = (
    500  # скорость уменьшения ε (чем меньше, тем быстрее убывает, но должна быть > 0).
)
TAU = 0.005  # скорость обновления target нейронной сети.
LR = 1e-4  # скорость обучения.
```

Рисунок 2 – Базовые значения гиперпараметров DQN

2) Влияние изменения архитектуры нейронной сети.

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

Для анализа размера нейронной сети, который наиболее эффективно решит эту задачу были сделаны еще 2 архитектуры (с меньшим и большим числом нейронов). Данные архитектуры представлены на рисунках 3 и 4:

```python
# Нейронная сеть поменьше для аппроксимации Q-функции.
class DQN_small(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN_small, self).__init__()
        self.layer1 = nn.Linear(n_observations, 64)
        self.layer2 = nn.Linear(64, 32)
        self.layer3 = nn.Linear(32, n_actions)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        return self.layer3(x)
```

Рисунок 3 – Архитектура нейронной сети поменьше

```python
# Нейронная сеть побольше для аппроксимации Q-функции.
class DQN_big(nn.Module):
    def __init__(self, n_observations, n_actions, p=0.4):
        super(DQN_big, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 256)
        self.layer3 = nn.Linear(256, 512)
        self.layer4 = nn.Linear(512, 256)
        self.layer5 = nn.Linear(256, 128)
        self.layer6 = nn.Linear(128, n_actions)

        self.dropout = nn.Dropout(p)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = torch.relu(self.layer3(x))
        x = self.dropout(x)
        x = torch.relu(self.layer4(x))
        x = torch.relu(self.layer5(x))
        return self.layer6(x)
```

Рисунок 4 – Архитектура нейронной сети побольше

Результат запуска обучения каждой из трех моделей представлен на рисунке 5:

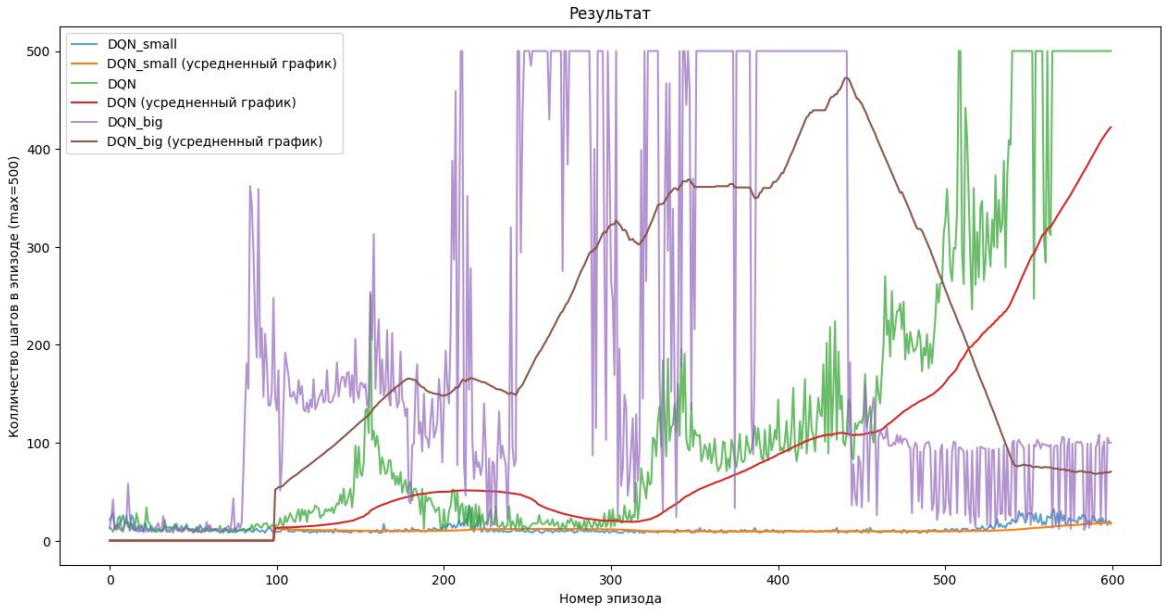

Рисунок 5 – Обучение для 3-х моделей

На графике представлены зависимости количества шагов в эпизоде от номера эпизода для каждой обучаемой модели в чистом и усредненном виде. Как видно, маленькая модель не смогла за 600 эпизодов обучиться совсем, средняя (базовая) обучилась где-то к 550 эпизоду, а большая модель обучилась к приблизительно 250 эпизоду.

Для данной задачи увеличение размера модели оказало положительное влияние на скорость обучения.

3) Влияние изменения параметра *gamma*.

Теперь поэкспериментируем с параметром *gamma,* который отвечает за коэффициент дисконтирования (чем он ближе к 1, тем модель больше заинтересована в дальней награде, чем ближе к 0, тем больше заинтересована в краткосрочной награде). В качестве эксперимента запустим обучение с параметрами *gamma*: 0.5, 0.9, 0.99, 1 (см. рисунок 6):

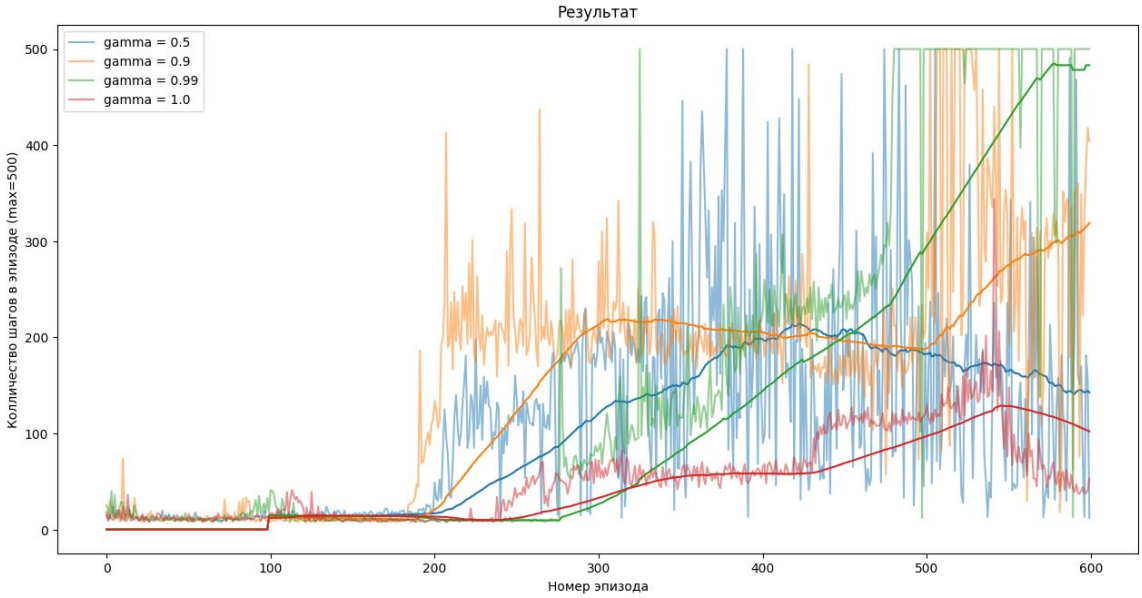

Рисунок 6 – Обучение с разными значениями *gamma*

Если судить по усредненным данным, то получаем, что базовое значение в 0.99 показало наилучший результат, что означает, что в данной задаче сбалансированная важность как краткосрочных, так и долгосрочных наград.

4) Влияние изменения параметра *epsilon_decay.*

epsilon_decay отвечает за то, насколько быстро будет уменьшаться изначальное значение *epsilon* (то есть насколько быстро будет уменьшаться процент выбора случайных действий и увеличится количество выборов действий обученной моделью). Чем ниже значение *epsilon_decay*, тем быстрее будет падать значение *epsilon* и наоборот. Запустим обучение со следующими значениями *epsilon_decay*: 100, 500, 1000, 2000 (см. рисунок 7):

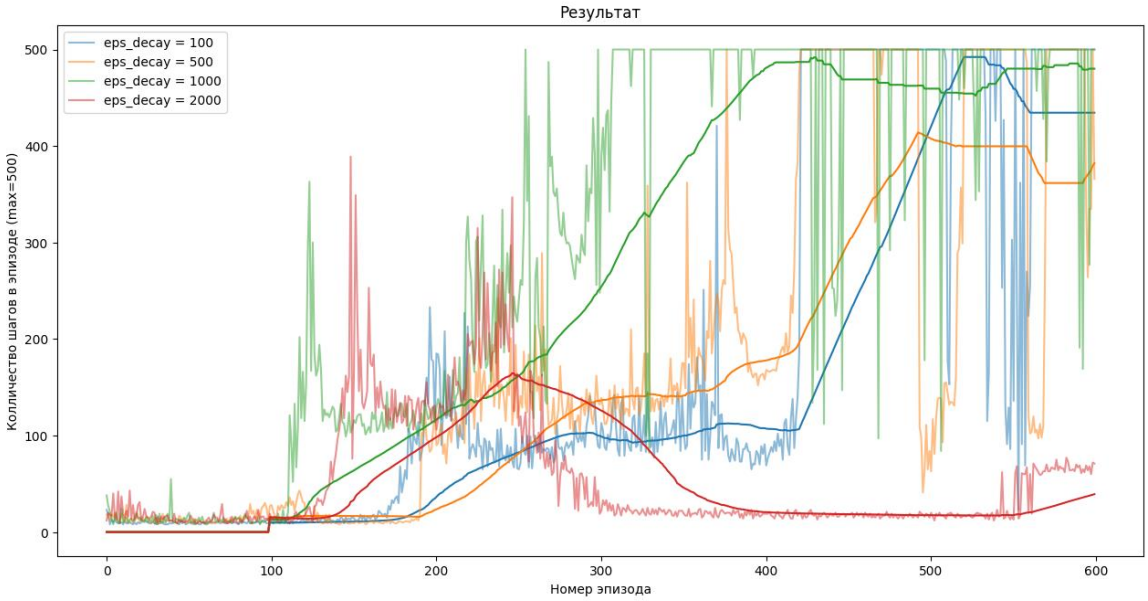

Рисунок 7 - Обучение с разными значениями *epsilon_decay*

Получаем, что при слишком большом значении (2000) действия слишком часто производятся случайно и модель не обучилась из-за этого, значение в 1000 оказалось лучшим, а значения 100 и 500, которые ведут почти к моментальному уменьшению *epsilon* до минимального значения, показывают средние результаты. Это означает, что в данной задаче случайное исследование имеет некоторую важность в начале, но даже без него возможно добиться обучения DQN.

5) Влияние изменения параметра *epsilon*.

epsilon влияет на изначальную частоту выбора случайных действий вместо действий модели. Было запущено обучение со следующими значениями для *epsilon*: 0.2, 0.5, 0.7 и 0.9 (см. рисунок 8):

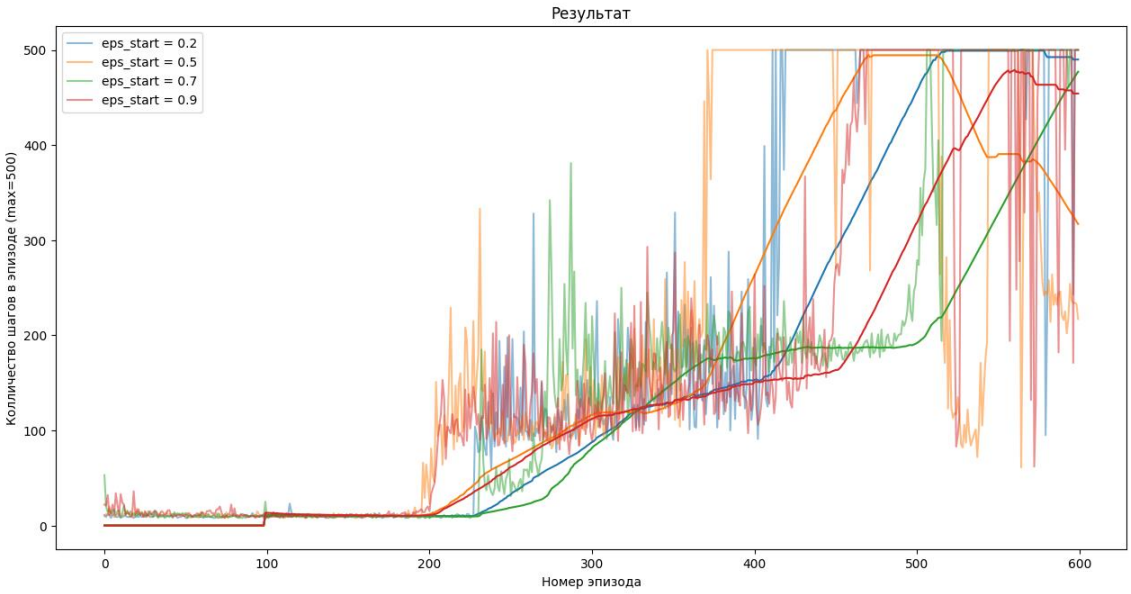

Рисунок 8 - Обучение с разными значениями *epsilon*

Полученные результаты согласуются с результатами, полученными при проверке разных значений *epsilon_decay*, так как быстрее всего обучение произошло при среднем значении выбора случайного действия 0.5.

Разработанный код, представлен в приложении А.

Заключение.

В ходе работы был изучен алгоритм обучения с подкреплением DQN и реализован с помощью библиотеки pytorch. Было исследовано влияние различных параметров на обучение DQN в конкретной среде CartPole-v1: большее количество слоев и нейронов в архитектуре модели повлияли положительно на скорость обучения DQN; анализ gamma показал, что для данной задачи важны как краткосрочные, так и долгосрочные результаты; исследование epsilon и epsilon_decay показало, что в данной задаче случайный выбор не так важен и достаточно среднего значения epsilon с довольно быстрой скоростью затухания, чтобы модель обучилась быстро.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import gym
import math
import random
from collections import namedtuple, deque
from itertools import count

import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Гиперпараметры.
BATCH_SIZE = 128  # размер мини-батча.
GAMMA = 0.99  # коэффициент дисконтирования.
EPS_START = 0.9  # начальное значение ε для ε-жадной стратегии.
EPS_END = 0.05  # минимальное значение ε.
EPS_DECAY = 1000  # скорость уменьшения ε.
TAU = 0.005
LR = 1e-4  # скорость обучения.

# Определяем структуру для хранения переходов (experience tuple).
Transition = namedtuple("Transition", ("state", "action", "next_state",
"reward"))


# Реализация буфера воспоминаний (replay memory).
class ReplayMemory:
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Сохраняем переход"""
        self.memory.append(Transition(*args))
```

```
    def sample(self, batch_size):
        """Случайным образом выбираем батч переходов"""
        return random.sample(self.memory, batch_size)

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


# Определяем нейронную сеть для аппроксимации Q-функции.
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        return self.layer3(x)


# Глобальная переменная для отслеживания числа шагов (для расчёта ε).
steps_done = 0


def select_action(state, policy_net, n_actions):
    """
    Выбирает действие с использованием ε-жадной стратегии.
    С вероятностью (1-ε) выбирается действие с максимальным Q,
    иначе – случайное действие.
    """
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(
        -1.0 * steps_done / EPS_DECAY
    )
    steps_done += 1

    if sample > eps_threshold:
        with torch.no_grad():
```

```
            # Выбираем действие с максимальным Q-значением.
            return policy_net(state).max(1).indices.view(1, 1)
    else:
        # Случайное действие.
        return torch.tensor(
            [[random.randrange(n_actions)]], device=device,
dtype=torch.long
        )


def optimize_model(policy_net, target_net, memory, optimizer):
    """
    Функция оптимизации модели на основе мини-батча из replay memory.
    """
    if len(memory) < BATCH_SIZE:
        return

    transitions = memory.sample(BATCH_SIZE)
    # Преобразуем список переходов в батч.
    batch = Transition(*zip(*transitions))

    # Создаем маску для тех переходов, где следующее состояние не None.
    non_final_mask = torch.tensor(
        tuple(map(lambda s: s is not None, batch.next_state)),
        device=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)

    # Вычисляем Q(s, a) для текущих состояний с помощью policy-сети.
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Вычисляем максимальные Q-значения для следующих состояний с исполь-
зованием target-сети.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
```

```
    with torch.no_grad():
        next_state_values[non_final_mask] = tar-
get_net(non_final_next_states).max(1)[0]

    # Вычисляем целевые значения Q: r + γ * max Q(next_state, a).
    expected_state_action_values = (next_state_values * GAMMA) + re-
ward_batch.squeeze()

    # Рассчитываем функцию потерь (MSE).
    loss = nn.SmoothL1Loss()(
        state_action_values.squeeze(), expected_state_action_values
    )

    # Обновляем веса сети.
    optimizer.zero_grad()
    loss.backward()
    # Ограничиваем градиенты для стабильности обучения.
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()


def plot_durations(episode_durations):
    plt.figure(1)

    durations_t = torch.tensor(episode_durations, dtype=torch.float)

    plt.title("Результат")
    plt.xlabel("Номер эпизода")
    plt.ylabel("Колличество шагов в эпизоде (max=500)")
    plt.plot(durations_t.numpy())

    # Усредняем по 100 эпизодам и тоже выводим.
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.show()
```

```
if __name__ == "__main__":
    env = gym.make("CartPole-v1")

    # Получаем размер наблюдения и количество действий из среды.
    initial_state, _ = env.reset()
    n_observations = len(initial_state)
    n_actions = env.action_space.n

    # Инициализируем policy-сеть и target-сеть.
    policy_net = DQN(n_observations, n_actions).to(device)
    target_net = DQN(n_observations, n_actions).to(device)
    target_net.load_state_dict(policy_net.state_dict())
    target_net.eval()

    optimizer = optim.Adam(policy_net.parameters(), lr=LR)
    memory = ReplayMemory(10000)

    num_episodes = 600  # количество эпизодов обучения.

    episode_durations = []

    for i_episode in range(num_episodes):
        # Инициализируем состояние среды.
        state, _ = env.reset()
        state = torch.tensor([state], device=device, dtype=torch.float32)

        for t in count():
            # Выбираем действие на основе текущего состояния.
            action = select_action(state, policy_net, n_actions)
            # Выполняем действие в среде.
            next_state, reward, done, truncated, _ =
env.step(action.item())
            reward = torch.tensor([reward], device=device)

            if done or truncated:
                next_state_tensor = None
            else:
                next_state_tensor = torch.tensor(
                    [next_state], device=device, dtype=torch.float32
                )
```

```
            # Сохраняем переход в replay memory.
            memory.push(state, action, next_state_tensor, reward)

            # Переходим к следующему состоянию.
            state = next_state_tensor if next_state_tensor is not None
else None

            optimize_model(policy_net, target_net, memory, optimizer)

            # Мягкое обновление весов target net.
            # θ' ← τ*θ + (1-τ)*θ'
            target_net_state_dict = target_net.state_dict()
            policy_net_state_dict = policy_net.state_dict()
            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[
                    key
                ] * TAU + target_net_state_dict[key] * (1 - TAU)
            target_net.load_state_dict(target_net_state_dict)

            if done or truncated:
                episode_durations.append(t + 1)
                print(f"Эпизод {i_episode} завершился за {t+1} шагов")
                break

    print("Обучение завершено")
    plot_durations(episode_durations)
    env.close()
```