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

ОТЧЕТ

по практической работе №1

по дисциплине «Обучение с подкреплением»

Тема: Реализация DQN для среды CartPole-v1

Студент гр. 0306 Кумаритов А.О.

Преподаватель Глазунов. С.А.

Санкт-Петербург

2025

Задание:

Реализовать DQN для среды CartPole-v1.

Задания для эксперимента:

Измените архитектуру нейросети (например, добавьте слои).

Попробуйте разные значения gamma и epsilon_decay.

Проведите исследование как изначальное значение epsilon влияет на скорость обучения

Описание среды:

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:

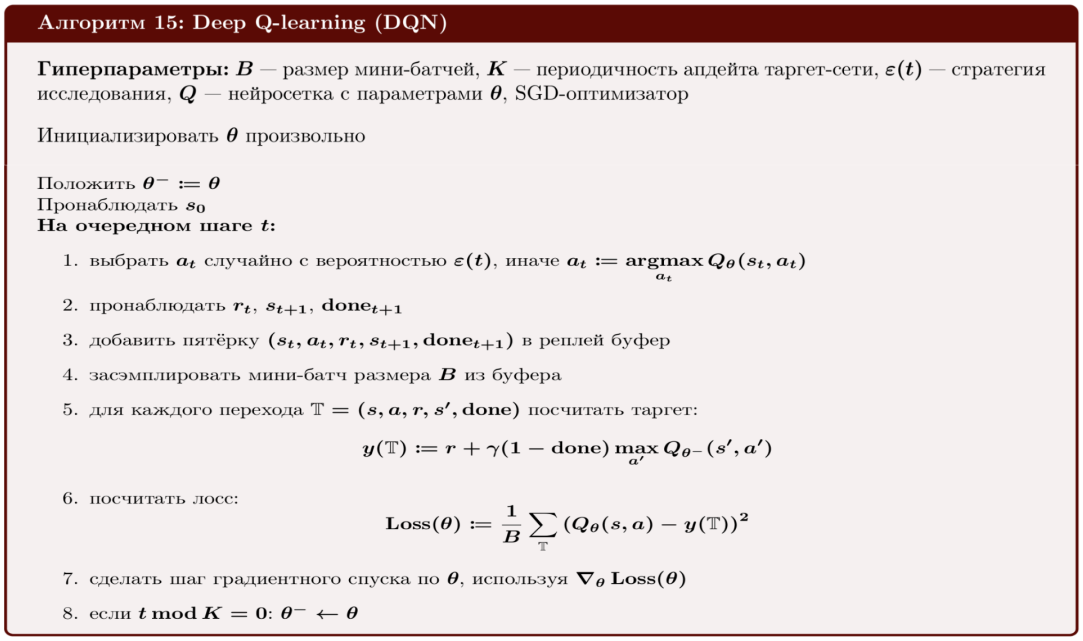
Pole angle меньше -12 градусов или больше 12 градусов

Cart position меньше -2.4 или больше 2.4

Номер эпизода больше 500.

Описание алгоритма:

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 $a_t := \operatorname{argmax}_{a_t} Q_\theta(s_t, a_t)$
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)

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

В реализации используем два класса:

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.

Под Q-функцией, выбирающей для агента лучшее действие, используется конфигурация нейронной сети, представленная на рисунке 2:

task1_0306_Kumaritov > dqn.py > DQN

```
1   from torch import nn
2
3
4   class DQN(nn.Module):
5
6       def __init__(self, n_observations, n_actions, hidden_size):
7           super(DQN, self).__init__()
8           self.model = nn.Sequential(
9               nn.Linear(n_observations, hidden_size),
10              nn.ReLU(),
11              nn.Linear(hidden_size, hidden_size // 2),
12              nn.ReLU(),
13              nn.Linear(hidden_size // 2, n_actions)
14          )
15
16      def forward(self, x):
17          return self.model(x)
18
```

Рис. 2 - конфигурация нейронной сети

При реализации алгоритма использовался ресурс https://docs.pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Выполнение экспериментов.

Изменение архитектуры нейросети:

В рамках эксперимента по сравнению влияния различной архитектуры было три запуска с hidden_size у нейронной сети равной 64, 128, 256. Этот параметр определяет ёмкость нейронной сети. Результат представлен на рисунке 3:

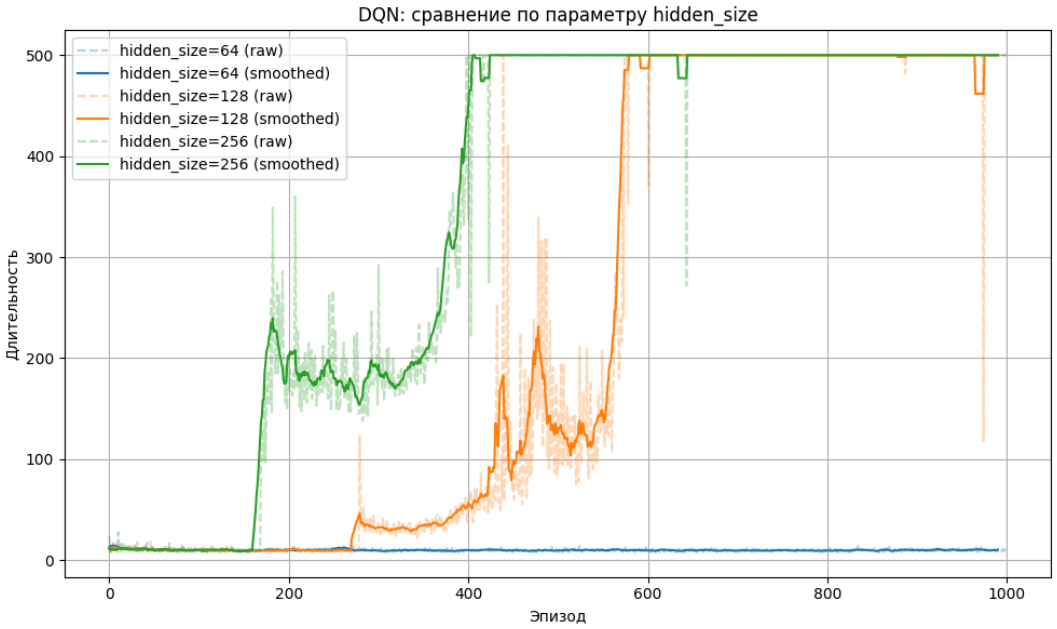


Рис. 3 - влияние различной архитектуры сети (hidden_size)

Лучший результат обучения показал размер скрытого слоя нейронной сети равный 256. Обучение происходило к приблизительно 400-му эпизоду. Из-за ограниченной емкости нейросети с размером скрытого слоя 64 был получен худший результат для аппроксимации Q-функции. Среднее значение в 128 показало средний результат, обучение которого происходило в районе 575 эпизода. В данных условиях использование большего количества скрытых слоев улучшает процесс обучения.

Разные значения gamma:

В рамках эксперимента по сравнению влияния различных значений gamma было три запуска с gamma равной 0.8, 0.9, 0.99. Этот

гиперпараметр, коэффициент дисконтирования, отвечает за то, насколько агент учитывает будущие или немедленные награды. Если значение ближе к 1, то агент оптимизирует сумму наград, если ближе к 0, то он учитывает только немедленные награды. Результат представлен на рисунке 4:

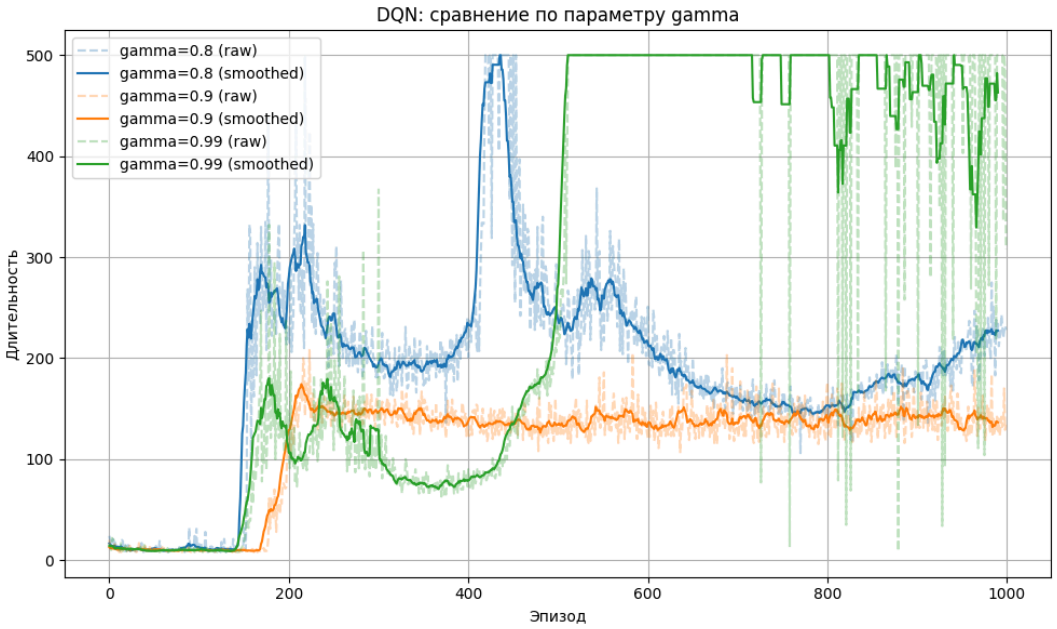


Рис. 4 - влияние различных значений gamma

Лучший результат обучения показало значение gamma равное 0.99. Среднее значение в 0.9 показало худший, но стабильный результат по длительности эпизода: наблюдался рост производительности к 200 эпизоду. Запуск с gamma равной 0.9 показал средний и нестабильный результат: достигнув максимума продолжительности к 425 эпизоду, далее продолжительность начала снижаться. Это говорит о важности оптимизации будущих наград в данной среде.

Разные значения epsilon_decay:

В рамках эксперимента по сравнению влияния различных значений eps_decay было три запуска с eps_decay равной 250, 500, 750. Этот гиперпараметр отвечает за то, насколько быстро уменьшается значение от

eps_start до eps_end. Чем больше значение, тем дольше происходит процесс уменьшения. Результат представлен на рисунке 5:

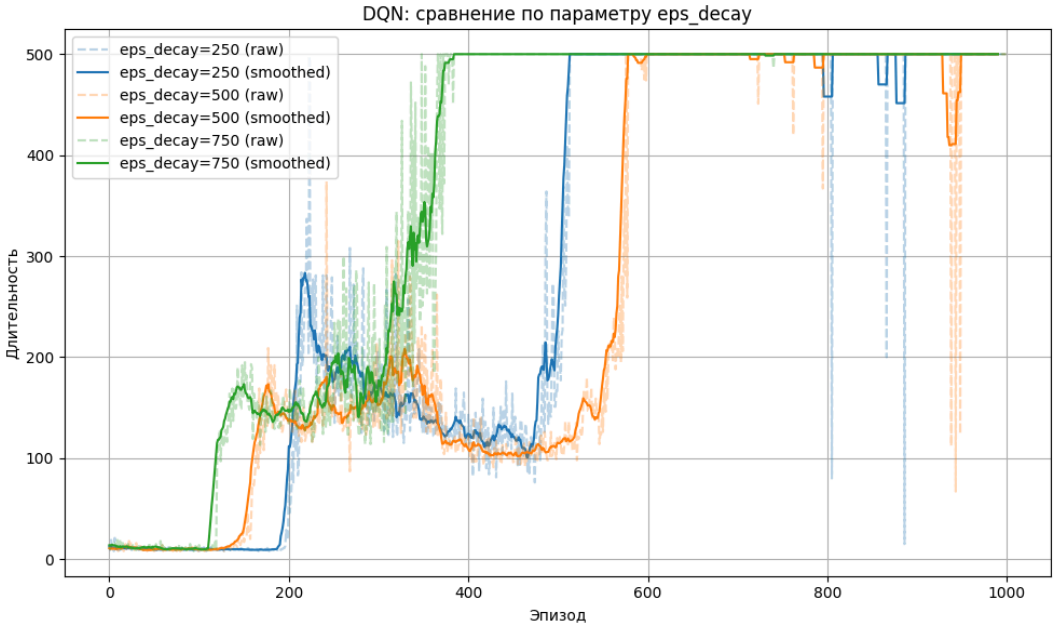


Рис. 5 - влияние различных значений eps_decay

Лучший результат обучения показало среднее значение eps_decay равное 750. Обучение происходило приблизительно после 350-го эпизода. Следующим по скорости обучения был запуск со значением eps_decay равным 250: обучение происходило к 500 эпизоду, а запуск со значением eps_decay равных 500 показали худший результат: обучение происходило к 575 эпизоду.

Влияние изначального значения epsilon на скорость обучения:

В рамках эксперимента по сравнению влияния различных значений eps_start было три запуска с eps_start равной 0.25, 0.5, 0.75. Этот гиперпараметр определяет начальное значение вероятности случайного действия в начале обучения. Результат представлен на рисунке 6:

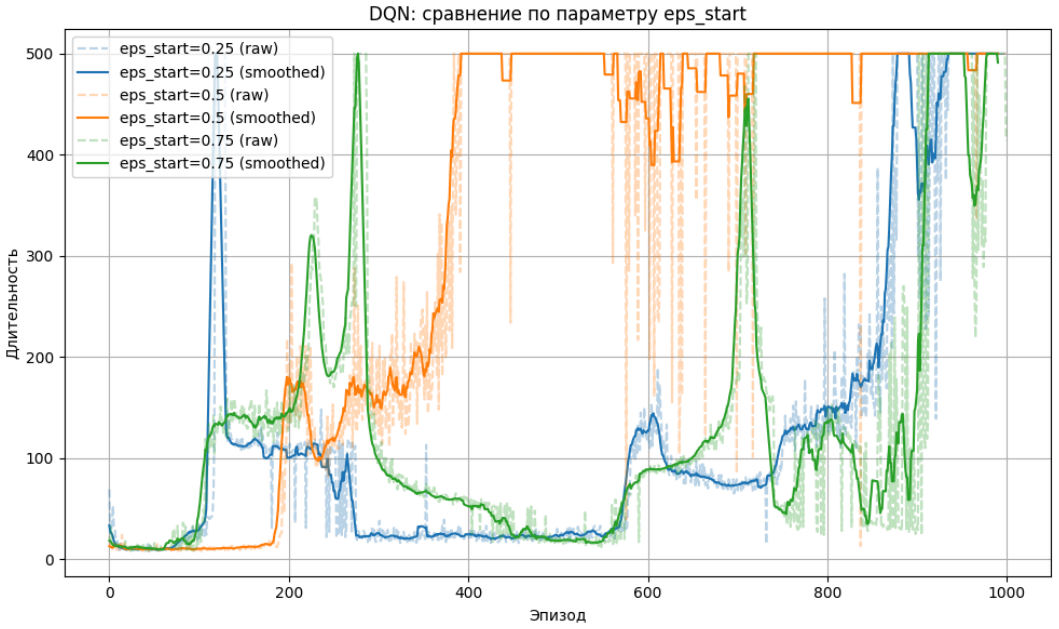


Рис. 6 - влияние различных значений eps_start

Лучший результат обучения показало среднее значение eps_start равное 0.5. Обучение происходило приблизительно к 400-му эпизоду. Два других значения eps_start в 0.25 и 0.75 показали схожий нестабильный результат. Обучение происходило приблизительно после 850-го эпизода. Результат показывает важность баланса между исследованием и использованием в данной среде.

Выводы.

Был реализован DQN для среды CartPole-v1. Были проведены исследования при различных значениях hidden_size, gamma, eps_decay, eps_start. Оптимальным для высоких результатов обучения, исходя из полученных данных, является запуск с hidden_size = 256, gamma = 0.99, eps_decay = 750, eps_start = 0.5.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ.

Исходный код main.py

```
import gymnasium as gym
import math
import matplotlib.pyplot as plt
import numpy as np
import random
import torch
from itertools import count
from torch import nn
from torch import optim

from dqn import DQN
from replay_memory import ReplayMemory
from transition import Transition

BATCH_SIZE = 128
GAMMA = 0.99
EPS_START = 0.5
EPS_END = 0.05
EPS_DECAY = 500
TAU = 0.005
LR = 1e-4
num_episodes = 1000

device = "cuda" if torch.cuda.is_available() else "cpu"
env = gym.make("CartPole-v1")
```

```
def optimize_model(memory, policy_net, target_net, optimizer, gamma):
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)

    batch = Transition(*zip(*transitions))

    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)

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

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
                                        next_state_values[non_final_mask] =
target_net(non_final_next_states).max(1).values
        expected_state_action_values = (next_state_values * gamma) +
reward_batch

    criterion = nn.SmoothL1Loss()
                                   loss = criterion(state_action_values,
expected_state_action_values.unsqueeze(1))
```

```
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
        optimizer.step()


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


    def train_dqn(hidden_size=256, gamma=GAMMA,
eps_start=EPS_START, eps_decay=EPS_DECAY):
        steps_done = 0
        n_actions = env.action_space.n
        state, _ = env.reset()
        n_observations = len(state)

        policy_net = DQN(n_observations, n_actions, hidden_size).to(device)
        target_net = DQN(n_observations, n_actions, hidden_size).to(device)
```

```
        target_net.load_state_dict(policy_net.state_dict())

            optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)
        memory = ReplayMemory(10000)
        episode_durations = []

        for _ in range(num_episodes):
            state, _ = env.reset()
                        state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
            for t in count():
                action, steps_done = select_action(state, policy_net, steps_done, eps_start, eps_decay)
                        observation, reward, terminated, truncated, _ = env.step(action.item())
                reward = torch.tensor([reward], device=device)
                done = terminated or truncated

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

                memory.push(state, action, next_state, reward)

                state = next_state
```

```
                optimize_model(memory, policy_net, target_net, optimizer,
gamma)

            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:
                episode_durations.append(t + 1)
                break

        return episode_durations


    def run_and_plot(param_values, param_name, train_kwargs,
filename_prefix):
        plt.figure(figsize=(10, 6))

        color_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']

        for i, val in enumerate(param_values):
            print(f"Обучение при {param_name} = {val}")
            steps_done = 0
            kwargs = train_kwargs(val)
            durations = train_dqn(**kwargs)
```

```
            color = color_cycle[i % len(color_cycle)]

            plt.plot(range(len(durations)), durations, linestyle='--', alpha=0.3,
                     color=color, label=f"{param_name}={val} (raw)")

            smoothed = np.convolve(durations, np.ones(10) / 10, mode='valid')
            plt.plot(range(len(smoothed)), smoothed, linestyle='-', color=color,
                     label=f"{param_name}={val} (smoothed)")

        plt.title(f"DQN: сравнение по параметру {param_name}")
        plt.xlabel("Эпизод")
        plt.ylabel("Длительность")
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.savefig(f"{filename_prefix}_{param_name}.png")


    def different_hidden_size():
        hidden_sizes = [64, 128, 256]
         run_and_plot(hidden_sizes, "hidden_size", lambda hz: {"hidden_size":
hz}, "results")


    def different_gamma():
        gammas = [0.8, 0.9, 0.99]
        run_and_plot(gammas, "gamma", lambda g: {"gamma": g}, "results")
```

```
def different_epsilon_decay():
  decays = [250, 500, 750]
      run_and_plot(decays, "eps_decay", lambda d: {"eps_decay": d},
"results")


def different_epsilon_start():
  starts = [0.25, 0.5, 0.75]
  run_and_plot(starts, "eps_start", lambda s: {"eps_start": s}, "results")


def main(seed=42):
  torch.manual_seed(seed)
  random.seed(seed)
  env.reset(seed=seed)
  np.random.seed(seed)
  different_hidden_size()
  different_gamma()
  different_epsilon_decay()
  different_epsilon_start()


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

Исходный код dqn.py

```
from torch import nn
```

```
class DQN(nn.Module):

  def __init__(self, n_observations, n_actions, hidden_size):
    super(DQN, self).__init__()
    self.model = nn.Sequential(
      nn.Linear(n_observations, hidden_size),
      nn.ReLU(),
      nn.Linear(hidden_size, hidden_size // 2),
      nn.ReLU(),
      nn.Linear(hidden_size // 2, n_actions)
    )

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

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)

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

Исходный код transition.py

```
from collections import namedtuple

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