**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.

|  |
| --- |
| 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:

**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 лаборатоной 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'))