**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МОЭВМ**

**ОТЧЕТ**

**по практической работе №3**

**по дисциплине «Обучение с подкреплением»**

**Тема: Реализация SAC для среды FlappyBird-v0**

| Студент гр. 0306 |  | Кумаритов А.О. |
| --- | --- | --- |
| Преподаватель |  | Глазунов. С.А. |

Санкт-Петербург

2025

**Задание:**

Реализовать SAC для среды FlappyBird-v0.

**Задания для эксперимента:**

Измените значение alpha для контроля энтропии.

Реализуйте автоматическую настройку alpha.

**Описание среды:**

Action space состоит из числа:

0 - ничего не делать

1 - взмахнуть крыльями

Observation space состоит из 12 чисел:

горизонтальное положение последней трубы

вертикальное положение последней верхней трубы

вертикальное положение последней нижней трубы

горизонтальное положение следующей трубы

вертикальное положение следующей верхней трубы

вертикальное положение следующей нижней трубы

горизонтальное положение следующей следующей трубы

вертикальное положение следующей следующей верхней трубы

вертикальное положение следующей следующей нижней трубы

вертикальное положение игрока

вертикальная скорость игрока

скорость вращения игрока

Rewards:

+0.1 - каждый успешный фрейм

+1.0 - успешный проход между труб

-1.0 - неудача

−0.5 - достижения верхней рамки экрана

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 0.45

Длительность эпизода равна 999

**Описание алгоритма:**

Базовое описание алгоритма представлено на рисунке 1.

|  |
| --- |
| Рис. 1 - алгоритм Soft Actor-Critic (SAC) |

В реализации следующие параметры:

alpha - коэффициент температуры, управляющий балансом между исследованием и обучением. Чем больше значение, тем больше агент предпочитает случайные действия, а при низких следует изученной стратегии.

is\_auto\_alpha - логическая переменная, включающая механизм динамического вычисления alpha.

В реализации используется несколько классов:

SAC - класс, реализующий инициализацию агента и все необходимые для обучения методы.

Actor, Critic\_DoubleQ - две нейросети. Actor рассчитывает действие (рис. 2), а Critic (рис. 3) оценивает состояние, в реализации SAC присутствуют две сети критика: основная critic и целевая critic\_target.

|  |
| --- |
| Рис. 2 - конфигурация нейронной сети Actor |

|  |
| --- |
| Рис. 3 - конфигурация нейронной сети Critic\_DoubleQ |

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.

**Выполнение экспериментов.**

**Изменение значения alpha для контроля энтропии:**

В рамках эксперимента по сравнению влияния различных значений параметра alpha на обучение алгоритма SAC было проведено три запуска с параметром alpha равным 0.1, 0.4, 0.8. Этот параметр определяет баланс между исследованием и использованием. Результат представлен на рисунке 4:

|  |
| --- |
| Рис. 4 - влияние различного значения alpha |

Лучший результат обучения на 1000 эпизодах показало значение alpha равное 0.1. Стабильность и рост награды достигалась к 25-ому эпизоду. Значение alpha равное 0.4 показало средний результат. Стабильность награды достигалась к 50-ому эпизоду. Значение alpha равное 0.8 показало худший и не стабильный результат. Такое высокое значение предполагает, что агент предпочитает случайные действия, что сказывается на стабильности получения награды на 1000 эпизодах.

**Реализация автоматической настройки alpha:**

В рамках эксперимента реализации автоматической настройки alpha и влияния этого механизма на обучение алгоритма SAC был проведён запуск с параметром is\_auto\_alpha равным True. Этот параметр включает механизм динамического изменения alpha. Результат представлен на рисунке 5:

|  |
| --- |
| Рис. 5 - влияние различных значений clip\_ratio |

Начальное значение alpha было равно 0.4. После достижения стабильного уровня наград к 100-ому эпизоду, значение alpha снизилось до 0.2, что сказалось на стабильности награды. К 500-ому эпизоду значение alpha начало расти, что отрицательно сказалось на стабильности, но положительно на увеличении награды за эпизод.

**Выводы.**

Был реализован SAC для среды FlappyBird-v0. Были проведены исследования при различных значениях alpha, а также при динамическом значении alpha. Лучшее значение из экспериментальных alpha на 1000 эпизодов - 0.1. Механизм динамической alpha показал возможность адаптации, но вызывал колебании награды.

**ПРИЛОЖЕНИЕ А.**

**ИСХОДНЫЙ КОД ПРОГРАММЫ.**

**Исходный код main.py**

import matplotlib.pyplot as plt

import numpy as np

from agent import SAC

def run\_and\_plot(param\_values, param\_name, train\_kwargs, filename\_prefix):

fig, ax1 = plt.subplots(figsize=(10, 6))

ax2 = ax1.twinx()

color\_cycle = plt.rcParams['axes.prop\_cycle'].by\_key()['color']

for i, val in enumerate(param\_values):

print(f"Обучение в FlappyBird-v0, режим {param\_name} {val}")

kwargs = train\_kwargs(val)

sac = SAC(\*\*kwargs, seed=42)

episode\_rewards, episode\_alphas = sac.train()

print(f"Обучение в FlappyBird-v0, режим {param\_name} {val} завершено")

color = color\_cycle[i % len(color\_cycle)]

episodes = np.arange(len(episode\_rewards))

ax1.plot(episodes, episode\_rewards, linestyle='--', alpha=0.3,

color=color, label=f"{param\_name}={val} (raw)")

smoothed = np.convolve(episode\_rewards, np.ones(10) / 10, mode='valid')

ax1.plot(episodes[:len(smoothed)], smoothed, linestyle='-',

color=color, label=f"{param\_name}={val} (smoothed)")

alpha\_color = color\_cycle[(i + len(param\_values)) % len(color\_cycle)]

ax2.plot(episodes, episode\_alphas, linestyle=':', color=alpha\_color,

label=f"{param\_name} alpha")

ax1.set\_title(f"SAC: сравнение параметра {param\_name} в FlappyBird-v0")

ax1.set\_xlabel("Эпизод")

ax1.set\_ylabel("Награда")

ax2.set\_ylabel("Alpha")

lines1, labels1 = ax1.get\_legend\_handles\_labels()

lines2, labels2 = ax2.get\_legend\_handles\_labels()

ax1.legend(lines1 + lines2, labels1 + labels2, loc="upper left")

ax1.grid(True)

fig.tight\_layout()

plt.savefig(f"{filename\_prefix}\_FlappyBird-v0.png")

plt.close()

def different\_alpha():

alpha = [0.1, 0.4, 0.8]

run\_and\_plot(alpha, "alpha", lambda a: {"alpha": a}, "alpha")

def calibrate\_alpha():

is\_auto\_alpha = [True]

run\_and\_plot(is\_auto\_alpha, "is\_auto\_alpha", lambda flag: {"is\_auto\_alpha": flag}, "is\_auto\_alpha")

def main():

different\_alpha()

calibrate\_alpha()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Исходный код agent.py**

from itertools import count

import flappy\_bird\_gymnasium

import gymnasium as gym

import numpy as np

import torch

import torch.nn.functional as F

import torch.optim as optim

from nets import Actor, Critic\_DoubleQ

from replay\_memory import ReplayMemory

class SAC():

def \_\_init\_\_(self, alpha=0.4, is\_auto\_alpha=False, seed=42):

torch.manual\_seed(seed)

np.random.seed(seed)

self.env = self.env = gym.make("FlappyBird-v0", use\_lidar=False)

self.env.action\_space.seed(seed)

self.env.observation\_space.seed(seed)

self.n\_actions = self.env.action\_space.n

state, \_ = self.env.reset(seed=seed)

self.n\_observations = len(state)

self.num\_episodes = 1000

self.tau = 0.005

self.gamma = 0.99

self.lr = 1e-4

self.hidden\_size = 256

self.alpha = alpha

self.is\_auto\_alpha = is\_auto\_alpha

self.replay\_memory\_size = 10000

self.memory = ReplayMemory(self.replay\_memory\_size)

self.batch\_size = 256

self.actor = Actor(self.n\_observations, self.n\_actions, self.hidden\_size)

self.actor\_optimizer = optim.AdamW(self.actor.parameters(), lr=self.lr, amsgrad=True)

self.critic = Critic\_DoubleQ(self.n\_observations, self.n\_actions, self.hidden\_size)

self.critic\_optimizer = optim.AdamW(self.critic.parameters(), lr=self.lr, amsgrad=True)

self.critic\_target = Critic\_DoubleQ(self.n\_observations, self.n\_actions, self.hidden\_size)

self.critic\_target.load\_state\_dict(self.critic.state\_dict())

if self.is\_auto\_alpha:

self.target\_entropy = 0.6 \* (-np.log(1 / self.n\_actions))

self.log\_alpha = torch.tensor(np.log(self.alpha), dtype=torch.float32, requires\_grad=True)

self.alpha\_optimizer = optim.AdamW([self.log\_alpha], lr=self.lr, amsgrad=True)

self.steps\_done = 0

def select\_action(self, state):

with torch.no\_grad():

state = torch.FloatTensor(state[np.newaxis, :])

probs = self.actor(state)

action = torch.multinomial(probs, num\_samples=1).item()

return action

def update\_critic(self):

if len(self.memory) < self.batch\_size:

return

transitions = self.memory.sample(self.batch\_size)

batch = list(zip(\*transitions))

states = torch.FloatTensor(np.array(batch[0]))

actions = torch.LongTensor(batch[1]).unsqueeze(1)

non\_final\_mask = torch.tensor([s is not None for s in batch[2]], dtype=torch.bool)

non\_final\_next\_states = [s for s in batch[2] if s is not None]

if non\_final\_next\_states:

next\_states = torch.FloatTensor(np.array(non\_final\_next\_states))

else:

next\_states = torch.empty((0, self.n\_observations), dtype=torch.float32)

rewards = torch.FloatTensor(batch[3]).unsqueeze(1)

with torch.no\_grad():

next\_probs = self.actor(next\_states)

next\_log\_probs = torch.log(next\_probs + 1e-8)

next\_q1, next\_q2 = self.critic\_target(next\_states)

min\_q = torch.min(next\_q1, next\_q2)

next\_v = (next\_probs \* (min\_q - self.alpha \* next\_log\_probs)).sum(dim=1, keepdim=True)

target\_q = rewards.clone()

target\_q[non\_final\_mask] += self.gamma \* next\_v

q1, q2 = self.critic(states)

q1\_pred = q1.gather(1, actions)

q2\_pred = q2.gather(1, actions)

critic\_loss = F.mse\_loss(q1\_pred, target\_q) + F.mse\_loss(q2\_pred, target\_q)

self.critic\_optimizer.zero\_grad()

critic\_loss.backward()

self.critic\_optimizer.step()

def update\_actor(self):

states = torch.FloatTensor(

np.array([t[0] for t in self.memory.sample(self.batch\_size)]))

probs = self.actor(states)

log\_probs = torch.log(probs + 1e-10)

q1, q2 = self.critic(states)

min\_q = torch.min(q1, q2)

actor\_loss = (probs \* (self.alpha \* log\_probs - min\_q)).sum(dim=1).mean()

self.actor\_optimizer.zero\_grad()

actor\_loss.backward()

self.actor\_optimizer.step()

if self.is\_auto\_alpha:

entropy = - (probs \* log\_probs).sum(dim=1).mean()

alpha\_loss = -(self.log\_alpha \* (self.target\_entropy - entropy).detach()).mean()

self.alpha\_optimizer.zero\_grad()

alpha\_loss.backward()

self.alpha\_optimizer.step()

self.alpha = min(max(self.log\_alpha.exp().item(), 0.001), 1.0)

def train(self):

self.actor.train()

self.critic.train()

self.critic\_target.train()

episode\_rewards = []

episode\_alphas = []

log\_interval = max(1, self.num\_episodes // 10)

for episode in range(self.num\_episodes):

state, \_ = self.env.reset()

total\_reward = 0

for \_ in count():

action = self.select\_action(state)

observation, reward, terminated, truncated, \_ = self.env.step(action)

total\_reward += reward

done = terminated or truncated

if done:

next\_state = None

else:

next\_state = observation

self.memory.push(state, action, next\_state, reward)

state = next\_state

if len(self.memory) >= self.batch\_size:

self.update\_critic()

self.update\_actor()

for target, policy in zip(self.critic\_target.parameters(), self.critic.parameters()):

target.data.copy\_(self.tau \* policy.data + (1 - self.tau) \* target.data)

if done:

episode\_alphas.append(self.alpha)

episode\_rewards.append(total\_reward)

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, episode\_alphas

**Исходный код nets.py**

import torch.nn as nn

import torch.nn.functional as F

class Actor(nn.Module):

def \_\_init\_\_(self, n\_observations, n\_actions, hidden\_size):

super(Actor, 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 F.softmax(self.model(x), dim=1)

class Critic\_DoubleQ(nn.Module):

def \_\_init\_\_(self, n\_observations, n\_actions, hidden\_size):

super(Critic\_DoubleQ, self).\_\_init\_\_()

self.q1 = nn.Sequential(

nn.Linear(n\_observations, hidden\_size),

nn.ReLU(),

nn.Linear(hidden\_size, hidden\_size),

nn.ReLU(),

nn.Linear(hidden\_size, n\_actions)

)

self.q2 = 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.q1(x), self.q2(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)

def clear(self):

self.memory.clear()

**Исходный код transition.py**

from collections import namedtuple

Transition = namedtuple('Transition',

('state', 'action', 'next\_state', 'reward'))