**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.

**Алгоритм 24: Soft Actor-Critic (SAC)**

**Гиперпараметры:** $B$ — размер мини-батчей, $\beta$ — параметр экспоненциального сглаживания таргет-сети, $\alpha$ — температура, $\pi_\theta(a \mid s) \coloneqq \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)^2 I)$ — гауссова стратегия с параметрами $\theta$, $Q_{\omega_1}(s,a), Q_{\omega_2}(s,a)$ — две нейросетки с параметрами $\omega_1$ и $\omega_2$, $V_\psi(s)$ — нейросетка с параметрами $\psi$, SGD-оптимизаторы.

Инициализировать $\theta, \omega_1, \omega_2, \psi$ произвольно
Инициализировать таргет-сеть $\psi^- \coloneqq \psi$
Пронаблюдать $s_0$
**На очередном шаге $t$:**

1. выбрать $a_t \sim \pi_\theta(a_t \mid s_t)$
2. пронаблюдать $r_t$, $s_{t+1}$, $\text{done}_{t+1}$
3. добавить пятёрку $(s_t, a_t, r_t, s_{t+1}, \text{done}_{t+1})$ в реплей буфер
4. засэмплировать мини-батч размера $B$ из буфера
5. для каждого $s$ из батча засэмплировать шума $\varepsilon(s) \sim \mathcal{N}(0, I)$ и посчитать $\mu(s,\theta), \sigma(s,\theta)$ стратегии $\pi_\theta$
6. посчитать оценку градиента по параметрам стратегии:

$$\nabla_\theta \coloneqq \frac{1}{B} \sum_{s \in B} \nabla_\theta \left[ \alpha \sum_{i=1}^{A} \log \sigma_i(s,\theta) + \min_{i=1,2} Q_{\omega_i}(s, \mu_\theta(s) + \sigma_\theta(s) \odot \varepsilon(s)) \right]$$

7. делаем шаг градиентного подъёма по $\theta$, используя $\nabla_\theta$
8. для каждого перехода $\mathbb{T} \coloneqq (s, a, r, s', \text{done})$ засэмплировать $a_\pi \sim \pi_\theta(a_\pi \mid s)$ и сохранить вероятности $\pi_\theta(a_\pi \mid s)$
9. посчитать таргеты:

$$y_V(\mathbb{T}) \coloneqq \min_{i=1,2} Q_{\omega_i}(s, a_\pi) - \alpha \log \pi_\theta(a_\pi \mid s)$$

$$y_Q(\mathbb{T}) \coloneqq r + \gamma V_{\psi^-}(s')$$

10. посчитать лоссы:

$$\text{Loss}_V(\psi) \coloneqq \frac{1}{B} \sum_{\mathbb{T}} \left(V_\psi(s') - y_V(\mathbb{T})\right)^2$$

$$\text{Loss}_{Q1}(\omega_1) \coloneqq \frac{1}{B} \sum_{\mathbb{T}} \left(Q_{\omega_1}(s,a) - y_Q(\mathbb{T})\right)^2$$

$$\text{Loss}_{Q2}(\omega_2) \coloneqq \frac{1}{B} \sum_{\mathbb{T}} \left(Q_{\omega_2}(s,a) - y_Q(\mathbb{T})\right)^2$$

11. делаем шаг градиентного спуска по $\psi$, $\omega_1$ и $\omega_2$, используя $\nabla_\psi \text{Loss}_V(\psi)$, $\nabla_\omega \text{Loss}_{Q1}(\omega_1)$ и $\nabla_{\omega_2} \text{Loss}_{Q2}(\omega_2)$ соответственно
12. обновляем таргет-сеть: $\psi^- \leftarrow (1-\beta)\psi^- + \beta\psi$

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.

```
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)
```

Рис. 2 - конфигурация нейронной сети Actor

```python
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)
```

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:

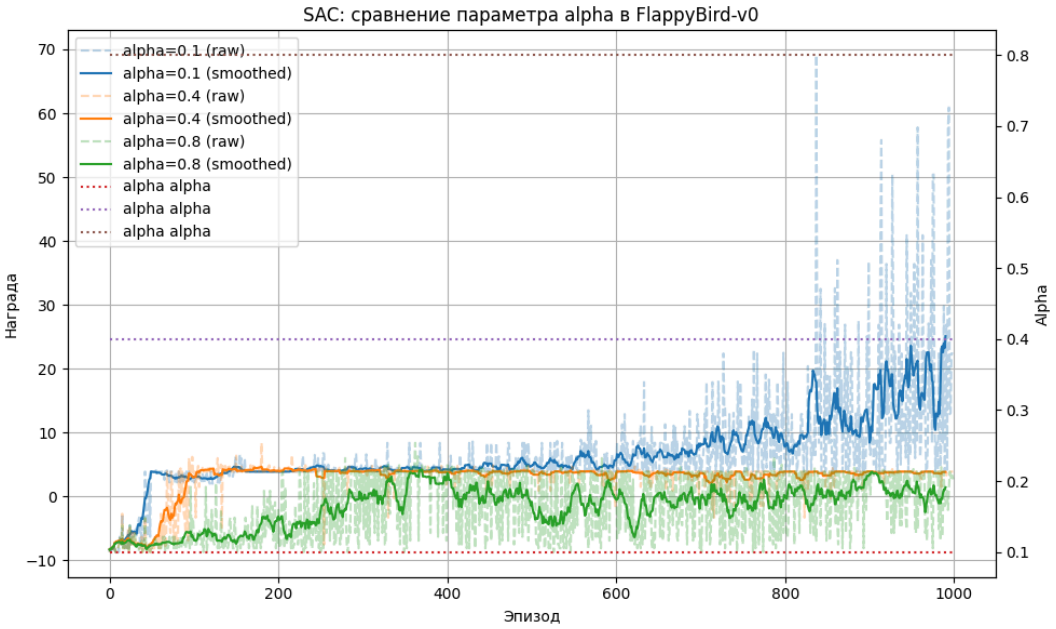


Рис. 4 - влияние различного значения alpha

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:

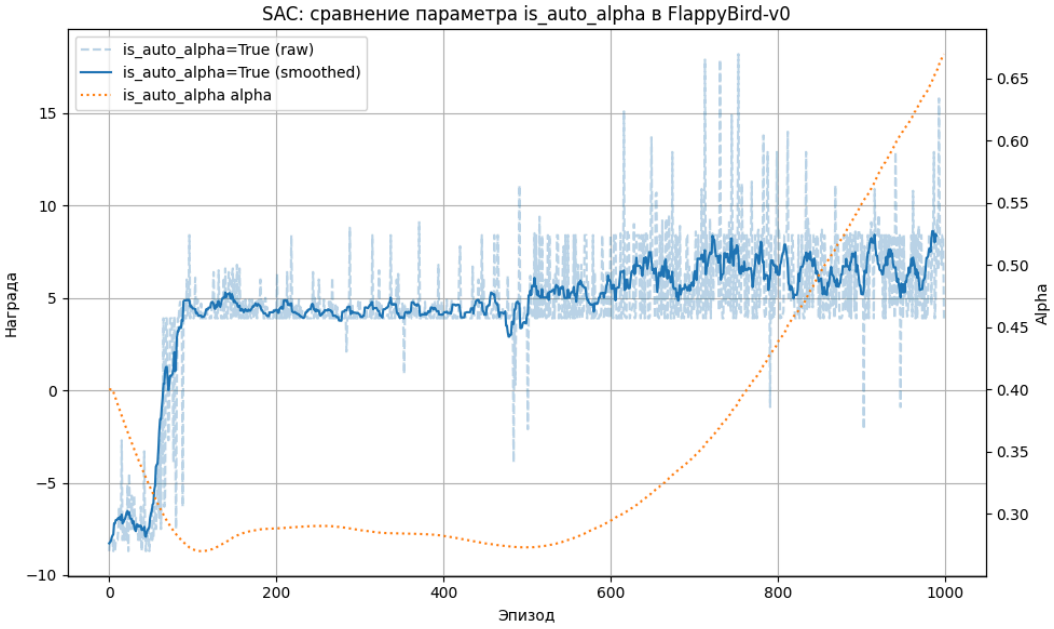


Рис. 5 - влияние различных значений clip_ratio

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'))
```