МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Н.Э. Баумана

Факультет «Информатика и системы управления»
Кафедра «Систем обработки информации и управления»

ОТЧЕТ

Лабораторная работа №__7__
по дисциплине «Методы машинного обучения в автоматизированных системах»

Тема: «Алгоритмы Actor-Critic»

ИСПОЛНИТЕЛЬ: ___Калюта Н.И.___
ФИО
группа ИУ5-22М _____
подпись
"30"___05___2024 г.

ПРЕПОДАВАТЕЛЬ: ___ _____
ФИО
_____
подпись
"__"_____2024 г.

Москва - 2024

Цель лабораторной работы:

Ознакомление с базовыми методами обучения с подкреплением на основе алгоритмов Actor-Critic.

Задание:

Реализуйте любой алгоритм семейства Actor-Critic для произвольной среды.

Установка библиотек Python

```
[12] import sys
     import torch
     import gym
     import numpy as np
     import torch.nn as nn
     import torch.optim as optim
     import torch.nn.functional as F
     from torch.autograd import Variable
     import matplotlib.pyplot as plt
     import pandas as pd
     from IPython.display import Image
```

Определяем набор гиперпараметров и констант, которые будут использоваться в дальнейшем коде для реализации алгоритма обучения с подкреплением.

```
# Гиперпараметры
hidden_size = 256
learning_rate = 3e-4
entropy_coef = 0.01

# Константы
GAMMA = 0.99
num_steps = 250
max_episodes = 2500  # Увеличенное количество эпизодов
CONST_ENV_NAME = "CartPole-v1"
CONST_DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Определяем класс ActorCritic, который реализует модель Actor-Critic для обучения с подкреплением

```
[14] # Класс, реализующий модель Actor-Critic
     class ActorCritic(nn.Module):
         def __init__(self, num_inputs, num_actions, hidden_size, learning_rate=3e-4):
             super(ActorCritic, self).__init__()

             self.num_actions = num_actions
             self.critic_linear1 = nn.Linear(num_inputs, hidden_size)
             self.critic_linear2 = nn.Linear(hidden_size, 1)

             self.actor_linear1 = nn.Linear(num_inputs, hidden_size)
             self.actor_linear2 = nn.Linear(hidden_size, num_actions)

         def forward(self, state):
             value = F.relu(self.critic_linear1(state))
             value = self.critic_linear2(value)

             policy_dist = F.relu(self.actor_linear1(state))
             policy_dist = F.softmax(self.actor_linear2(policy_dist), dim=1)

             return value, policy_dist

         def select_action(self, state):
             _, policy_dist = self.forward(state)
             action = policy_dist.multinomial(num_samples=1).detach()
             return action[0]
```

Реализуем алгоритм Advantage Actor-Critic (A2C) для обучения агента в среде с ограниченным числом шагов. Он создает модель Actor-Critic, состоящую из актора, который выбирает действия, и критика, который оценивает их качество, обучает эту модель с использованием градиентного спуска, и визуализирует результаты обучения в виде графиков наград и длин эпизодов. Код также включает в себя логику для вычисления значений Q и преимуществ, необходимых для обновления весов модели.

```
# Функция для реализации алгоритма Advantage Actor-Critic (A2C)
def a2c(env):
    num_inputs = env.observation_space.shape[0]
    num_outputs = env.action_space.n

    actor_critic = ActorCritic(num_inputs, num_outputs, hidden_size)
    ac_optimizer = optim.Adam(actor_critic.parameters(), lr=learning_rate)

    all_lengths = []
    average_lengths = []
    all_rewards = []
    entropy_term = 0

    for episode in range(max_episodes):
        log_probs = []
        values = []
        rewards = []

        state = env.reset()
        if isinstance(state, tuple):
            state = state[0]
        state = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
```

```
        for steps in range(num_steps):
            value, policy_dist = actor_critic.forward(state)
            value = value.detach().numpy()[0, 0]
            dist = policy_dist.detach().numpy()

            action = np.random.choice(num_outputs, p=np.squeeze(dist))
            log_prob = torch.log(policy_dist.squeeze(0)[action])
            entropy = -np.sum(np.mean(dist) * np.log(dist))
            result = env.step(action)
            new_state = result[0]
            reward = result[1]
            done = result[2]

            if isinstance(new_state, tuple):
                new_state = new_state[0]
            new_state = torch.tensor(new_state, dtype=torch.float32).unsqueeze(0)

            rewards.append(reward)
            values.append(value)
            log_probs.append(log_prob)
            entropy_term += entropy
            state = new_state

            if done or steps == num_steps - 1:
                Qval, _ = actor_critic.forward(new_state)
                Qval = Qval.detach().numpy()[0, 0]
                all_rewards.append(np.sum(rewards))
                all_lengths.append(steps)
                average_lengths.append(np.mean(all_lengths[-10:]))
                if episode % 10 == 0:
                    sys.stdout.write("episode: {}, reward: {}, total length: {}, average length: {} \n".format(episode, np.sum(rewards), steps, average_lengths[-1]))
                break

        # Вычисление значений Q
        Qvals = np.zeros_like(values)
        for t in reversed(range(len(rewards))):
            Qval = rewards[t] + GAMMA * Qval
            Qvals[t] = Qval

        # Обновление модели actor-critic
        values = torch.FloatTensor(values)
        Qvals = torch.FloatTensor(Qvals)
        log_probs = torch.stack(log_probs)

        advantage = Qvals - values
        actor_loss = (-log_probs * advantage).mean()
        critic_loss = 0.5 * advantage.pow(2).mean()
        ac_loss = actor_loss + critic_loss + entropy_coef * entropy_term

        ac_optimizer.zero_grad()
        ac_loss.backward()
        ac_optimizer.step()
```

```
    # Построение графиков результатов
    smoothed_rewards = pd.Series(all_rewards).rolling(10).mean()
    plt.plot(all_rewards, label='Награды')
    plt.plot(smoothed_rewards, label='Сглаженные награды')
    plt.xlabel('Эпизод')
    plt.ylabel('Награда')
    plt.legend()
    plt.show()

    plt.plot(all_lengths, label='Длины эпизодов')
    plt.plot(average_lengths, label='Средние длины')
    plt.xlabel('Эпизод')
    plt.ylabel('Длина эпизода')
    plt.legend()
    plt.show()

    return actor_critic
```

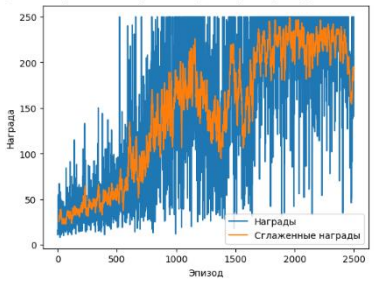
Определяем функцию play_agent, которая позволяет запустить обученного агента в среде и визуализировать его действия. Функция создает экземпляр среды, загружает обученную модель Actor-Critic, выбирает действия на основе текущего состояния среды, выполняет эти действия и визуализирует процесс. Она продолжает взаимодействие с средой до тех пор, пока не будет достигнуто завершение эпизода. Затем она выводит информацию о выбранных действиях и полученных наградах за весь эпизод.

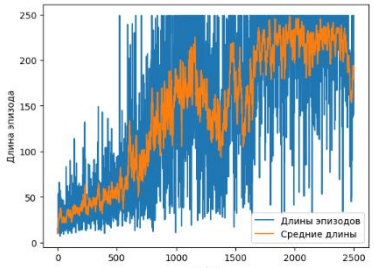
```
[16] # Функция для запуска агента в среде с использованием обученной модели
     def play_agent(actor_critic):
         '''
         Игра с обученным агентом
         '''
         env2 = gym.make(CONST_ENV_NAME, render_mode='human')
         state = env2.reset()
         if isinstance(state, tuple):
             state = state[0]
         state = torch.tensor(state, dtype=torch.float32, device=CONST_DEVICE).unsqueeze(0)
         done = False
         res = []
         while not done:
             action = actor_critic.select_action(state)
             action = action.item()
             result = env2.step(action)
             observation = result[0]
             reward = result[1]
             done = result[2]
             truncated = result[3]
             env2.render()

             res.append((action, reward))

             if done or truncated:
                 done = True
             else:
                 if isinstance(observation, tuple):
                     observation = observation[0]
                 state = torch.tensor(observation, dtype=torch.float32, device=CONST_DEVICE).unsqueeze(0)

         print('Данные эпизода: ', res)
```

Создаем экземпляр игровой среды, вызывает функцию a2c() для обучения модели Actor-Critic на этой среде, а затем вызывает функцию play_agent() для запуска обученного агента в той же среде и визуализации его поведения. Таким образом, этот код позволяет обучить агента с использованием алгоритма A2C и затем наблюдать за его действиями в игровой среде.

```
[17] if __name__ == "__main__":
         env = gym.make(CONST_ENV_NAME)
         trained_actor_critic = a2c(env)

         # Запуск сессии с обученным агентом
         play_agent(trained_actor_critic)
```

```
episode: 2210, reward: 236.0, total length: 235, average length: 222.0
episode: 2220, reward: 191.0, total length: 190, average length: 233.5
episode: 2230, reward: 250.0, total length: 249, average length: 234.5
episode: 2240, reward: 208.0, total length: 207, average length: 211.4
episode: 2250, reward: 141.0, total length: 140, average length: 226.7
episode: 2260, reward: 229.0, total length: 228, average length: 229.7
episode: 2270, reward: 250.0, total length: 249, average length: 225.0
episode: 2280, reward: 250.0, total length: 249, average length: 226.1
episode: 2290, reward: 250.0, total length: 249, average length: 233.6
episode: 2300, reward: 211.0, total length: 210, average length: 195.9
episode: 2310, reward: 234.0, total length: 233, average length: 232.7
episode: 2320, reward: 250.0, total length: 249, average length: 223.1
episode: 2330, reward: 247.0, total length: 246, average length: 222.9
episode: 2340, reward: 250.0, total length: 249, average length: 220.9
episode: 2350, reward: 239.0, total length: 238, average length: 219.2
episode: 2360, reward: 250.0, total length: 249, average length: 234.7
episode: 2370, reward: 212.0, total length: 211, average length: 231.5
episode: 2380, reward: 178.0, total length: 177, average length: 211.4
episode: 2390, reward: 250.0, total length: 249, average length: 230.4
episode: 2400, reward: 128.0, total length: 127, average length: 220.4
episode: 2410, reward: 241.0, total length: 240, average length: 209.9
episode: 2420, reward: 230.0, total length: 229, average length: 226.7
episode: 2430, reward: 183.0, total length: 182, average length: 222.8
episode: 2440, reward: 159.0, total length: 158, average length: 210.5
episode: 2450, reward: 209.0, total length: 208, average length: 191.3
episode: 2460, reward: 198.0, total length: 197, average length: 188.9
episode: 2470, reward: 155.0, total length: 154, average length: 164.3
episode: 2480, reward: 186.0, total length: 185, average length: 157.4
episode: 2490, reward: 138.0, total length: 137, average length: 179.1
```



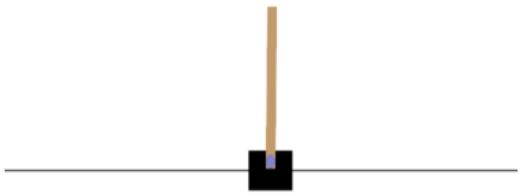



```
Данные эпизода: [(1, 1.0), (0, 1.0), (0, 1.0), (1, 1.0), (0, 1.0), (1, 1.0), (1, 1.0), (0, 1.0), (1, 1.0), (1, 1.0), (1, 1.0), (0, 1.0), (0, 1.0), (
```

```
[18] from google.colab import drive
     drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
[ ] image_path = "/content/drive/MyDrive/Временные файлы/a2c_result.png"
    display(Image(filename=image_path, width=500, height=500))
```



Вывод:

В ходе данной лабораторной работы был продемонстрирован процесс обучения агента с помощью алгоритма A2C (Advantage Actor-Critic) в игровой среде. Сначала была создана игровая среда с использованием библиотеки Gym. Затем была вызвана функция a2c(), которая обучала агента, используя алгоритм A2C. После обучения, агент был запущен в игровой среде, чтобы наблюдать за его поведением.