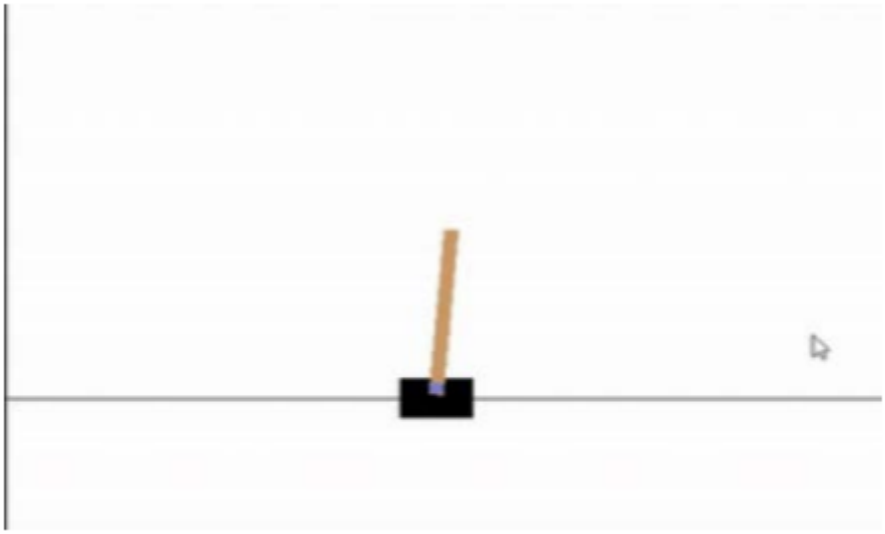
1. 作业说明

实现倒立摆（CartPole）小游戏，游戏里面有一个小车，上有竖着一根杆子，每次重置后的初始状态会有所不同。小车需要左右移动来保持杆子竖直，为了保证游戏继续进行需要满足以下两个条件：

1. 杆子倾斜的角度θ不能大于15°
2. 小车移动的位置x需保持在一定范围（中间到两边各2.4个单位长度）



2. 评分标准

本次RL小实验设计主要为了学习状态，动作，价值，回报的设计，所以侧重于通过完成报告阐述对该算法了解，因此各分数设计为：

- PDF报告描述该强化学习场景，简要叙述state，action，value，reward设计过程（3）
- 给出CartPole算法伪代码（1）
- 代码、运行结果和截图（1）

3. 设计过程

3.1. 环境描述

使用环境是OpenAi Gym中的CartPole-v1环境，但是根据作业要求做了自定义

Episode End

The episode ends if any one of the following occurs:

1. Termination: Pole Angle is greater than ±12°
2. Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than 500 (200 for v0)

作业要求的角度是15°，因此在MyModel文件中对环境进行集成做了自定义

3.2. 强化学习场景描述

这是一个经典的倒立摆问题。在这个场景中，一个杆子通过一个不可控制的关节连接到一个小车上，这个小车可以沿着无摩擦的轨道移动。杆子被放置在小车上并竖直向上。智能体的目标是通过向小车施加左右方向的力，保持杆子竖直不倒。

3.3. 详细设计

State:

Observation Space

这个环境的状态（state）由一个长度为 4 的向量构成，包含了以下信息：

1. Cart Position（小车位置）
2. Cart Velocity（小车速度）
3. Pole Angle（杆子角度）
4. Pole Angular Velocity（杆子角速度）

Action:

Action Space

供选择的动作是离散的，只有两种：0 或 1。这表示对小车施加的力的方向。

- `0`: 向左推小车
- `1`: 向右推小车

Value:

Value Function

在这里，价值（value）函数是由 DQN（Deep Q-Network）网络来近似估计的状态动作值函数。它表示每个状态动作对的预期累积奖励（就是采取这个动作从长期来看收益如何

Reward:

可以简单理解为每多撑一轮奖励就会加一

在 CartPole环境中，每一步操作都会获得 +1 的奖励。目标是尽可能地保持杆子竖直，所以只要杆子没有超出倾斜角度或小车位置超出范围，就会得到奖励 +1。而在该版本的环境中，完成任务所需的奖励阈值为 500。

也就是超过500就会被截断，不过这个参数也是可以改的

4. 伪代码

```
初始化环境
注册定制的CartPole环境，修改角度偏差值
创建经验回放池
定义DQN网络结构

for 循环进行多个episode:
    初始化环境并获取初始状态
    while 循环进行每个time step:
```

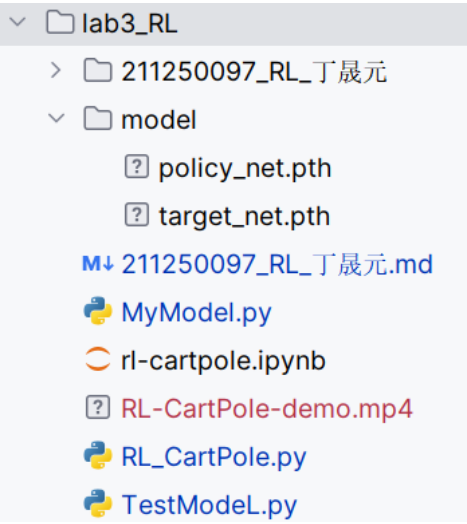
```
            通过epsilon-greedy策略选择action（使用DQN网络或随机选择）
            执行action，获取新状态和奖励
            存储经验（状态，动作，新状态，奖励）到经验回放池
            将初始状态更新为新状态
            从经验回放池中抽取部分经验用于训练DQN网络（通过计算loss并反向传播）
                如果当前时间步不是最终时间步，则：
                    通过DQN网络估计下一个状态的Q值
                    计算损失（实际的Q值和目标Q值之间的差异）然后优化网络参数
            更新target network的权重
            如果达到终止条件，则退出while循环
        绘制和可视化训练进程

保存训练好的模型
关闭环境
```

5. 代码

主要是两个py文件，一个是MyModel.py，一个是RL_CartPole.py

测试时可以用TestModel.py和model下的pth参数（权重和偏置）文件



5.1. MyModel.py

```python
import math
from typing import Optional

from gymnasium.envs.classic_control import CartPoleEnv


# 创建一个新的 CartPole 环境类，继承自原始的 CartPoleEnv
class CustomCartPoleEnv(CartPoleEnv):
    metadata = {
        "render_modes": ["human", "rgb_array"],
        "render_fps": 50,
    }

    def __init__(self, render_mode: Optional[str] = None):
```

```
        super(CustomCartPoleEnv, self).__init__(render_mode=render_mode)

        # 修改 theta_threshold_radians 参数为 ±15度
        self.theta_threshold_radians = 15 * 2 * math.pi / 360

# # 注册新的 CartPole 环境
# gym.envs.register(
#     id='CustomCartPole-v1',
#     entry_point='xxxxx:CustomCartPoleEnv',
#     max_episode_steps=500,  # 可以根据需要修改
#     reward_threshold=475.0,  # 可以根据需要修改
# )
#
# # 创建新的环境实例
# custom_env = gym.make('CustomCartPole-v1')

```

5.2. RL_CartPole.py

```
import os

import gymnasium as gym
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# 注册新的 CartPole 环境
gym.envs.register(
    id='CustomCartPole-v1',
    entry_point='MyModel:CustomCartPoleEnv',
    max_episode_steps=600,  # 可以根据需要修改
)

# 创建新的环境实例
env = gym.make('CustomCartPole-v1', render_mode="rgb_array")

# Transition是一个命名元组，用于表示一个转换，包含四个属性：state, action, next_state, reward
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))


# ReplayMemory是一个有限长度的存储器，用于存储Agent的经验
# 也叫作经验回放池（Experience Replay Pool）
class ReplayMemory(object):

    def __init__(self, capacity):
```

```
        # deque是一个双端队列，可以从头尾两端添加和删除元素
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        # *args是一个可变参数，可以接受任意多个参数
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        # 选取batch_size个转换
        return random.sample(self.memory, batch_size)

    def __len__(self):
        # 自定义实现len函数
        return len(self.memory)


# Q Network是一个简单的全连接神经网络
# 输入是状态，输出是每个动作的Q值
class DQN(nn.Module):

    def __init__(self, n_observations_, n_actions_):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations_, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions_)

    # forward函数定义了前向传播的运算
    # 返回值是每个动作的Q值
    # x可能包含多个样本，每个样本是一个状态，返回的是每个样本对应两个动作的Q值
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)


# IPython环境是指在Jupyter Notebook或者Jupyter QtConsole中运行Python代码
# 如果是IPython环境，那么is_ipython为True，调用display模块中的display函数显示动画
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

# 启用交互模式，可以在动画进行中更新图像
plt.ion()

# 如果有GPU，使用GPU，否则使用CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# BATCH_SIZE是一个批次的大小，即每次从ReplayMemory中随机选取多少个转换进行训练
BATCH_SIZE = 128
# GAMMA是折扣因子，用于计算折扣回报
# 折扣回报是指未来的奖励的折扣累加和，越靠近1，越重视未来的奖励
GAMMA = 0.99
# EPS_START是起始的探索率， EPS_END是最终的探索率， EPS_DECAY是探索率的衰减率
EPS_START = 0.9
EPS_END = 0.05
```

```
EPS_DECAY = 1000
# TAU是目标网络的更新率，LR是优化器的学习率
TAU = 0.005
LR = 1e-4

# n_actions是动作的数量，其实在CartPole的例子中就是2
n_actions = env.action_space.n
# Get the number of state observations
state, info = env.reset()
n_observations = len(state)

# 打印state 和 info
print("state:")
print(state)
print("info:")
print(info)
print("n_observations:")
print(n_observations)

# policy_net是当前的Q网络，target_net是目标Q网络
# 前者用于选择动作，后者用于计算目标Q值
policy_net = DQN(n_observations, n_actions).to(device)
target_net = DQN(n_observations, n_actions).to(device)

# 一开始target_net初始化为policy_net的参数
target_net.load_state_dict(policy_net.state_dict())

# 优化器使用AdamW，学习率为LR，amsgrad=True表示使用amsgrad算法
optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)
# 经验回放池的容量为10000
memory = ReplayMemory(10000)

# steps_done是一个计数器，用于记录Agent总共与环境交互了多少次
steps_done = 0


# select_action函数用于根据当前状态选择动作
# 依据epsilon-greedy策略选择动作，即以epsilon的概率随机选择动作，以1-epsilon的概率选择Q值最大的动作
def select_action(state_):
    global steps_done

    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
                    math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1

    # 以如果sample大于eps_threshold，那么以当前Q网络选择动作
    if sample > eps_threshold:
        # t.no_grad()表示不需要计算梯度，因为我们不需要对Q网络进行训练，只是用于选择动作
        with torch.no_grad():
            # 由于我们的输入是一个批次的状态，所以返回的是每个状态对应的两个动作的Q值
            # 因此我们要用t.max(1)[1]来获取状态对应的两个动作中Q值最大的那个动作的索引
            # view(1, 1)是将其变形为一个行向量
```

```
            return policy_net(state_).max(1)[1].view(1, 1)
    else:
        # 否则随机选择动作，调用env.action_space.sample()函数
        return torch.tensor([[env.action_space.sample()]], device=device,
dtype=torch.long)


episode_durations = []


def plot_durations(show_result=False):
    plt.figure(1)
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    if show_result:
        plt.title('Result')
    else:
        plt.clf()
        plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated
    if is_ipython:
        if not show_result:
            display.display(plt.gcf())
            display.clear_output(wait=True)
        else:
            display.display(plt.gcf())


# new
# 用来显示CartPole的状态
def plot_cartpole_state(screen_):
    plt.figure(2)
    plt.clf()
    plt.imshow(screen_, interpolation='none')
    plt.title('CartPole State')
    plt.axis('off')
    plt.pause(0.001)
    if is_ipython:
        display.clear_output(wait=True)
        display.display(plt.gcf())


# training loop
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)

```

```
    # 将一个批次的转换解压为一个批次的状态，一个批次的动作，一个批次的下一个状态，一个批次的奖励
    batch = Transition(*zip(*transitions))

    # 首先计算一个批次的非终止状态的掩码，即next_state不为None就为True，否则为False
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)), device=device,
                                  dtype=torch.bool)
    # 提取了这个批次中所有next_state不为None的转换，然后将其拼接成一个tensor
    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)

    # 将当前状态输入policy_net，得到每个状态对应的两个动作的Q值
    # 再使用gather函数，将action_batch中的动作对应的Q值提取出来
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # 计算下一个状态的Q值，首先初始化为0
    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)[0]
    # 预期的值是下一个状态的Q值乘以折扣因子再加上奖励 r + gamma * max_a' Q(s', a')
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # 计算损失，使用smooth_l1_loss函数，即Huber Loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # 优化模型，首先将梯度置为0，然后进行反向传播，最后进行梯度裁剪
    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    # 使用优化器来更新模型参数，使得梯度下降
    optimizer.step()


if torch.cuda.is_available():
    num_episodes = 1000
else:
    num_episodes = 600

for i_episode in range(num_episodes):
    # 初始化环境并获取初始状态
    # 按照官方文档的说法可以设置初始状态的生成范围
    state, info = env.reset()
    # unsqueeze(0) 将原始状态数据变成了一个形状为 (1, ...) 的张量，其中 1 表示批次大小
    # 便于后续的模型输入
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)

```

```
    # 这是一种无限循环的写法，直到任务终止才会退出循环
    for t in count():
        # 根据状态选择动作
        action = select_action(state)
        # step函数执行动作，返回新的状态、奖励、是否终止、是否被截断等信息
        print(action)
        print(type(action))
        observation, reward, terminated, truncated, _ =
env.step(action.item())
        reward = torch.tensor([reward], device=device)
        # done是终止或者被截断的标志
        done = terminated or truncated

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

        # 存储状态转换信息到经验回放缓冲区
        memory.push(state, action, next_state, reward)

        # 切换到下一个状态
        state = next_state

        # 执行一次模型优化（policy网络的训练）
        # 取样一个批次的转换，然后进行模型优化
        optimize_model()

        # new
        # 获取当前环境的屏幕状态
        screen = env.render()
        plot_cartpole_state(screen)

        # 软更新目标网络的权重
        # 这里的软更新是指将目标网络的权重向策略网络的权重靠近一点
        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)
            plot_durations()
            break

print('Complete')
plot_durations(show_result=True)
plt.ioff()
plt.show()
# 结束后关闭环境
env.close()

# 保存模型
```

```
print("Saving model...")
# 创建文件夹名为model
if not os.path.exists('./model'):
    os.mkdir('./model')
# 保存模型参数
torch.save(policy_net.state_dict(), './model/policy_net.pth')
torch.save(target_net.state_dict(), './model/target_net.pth')
print("Model saved!")
```

6. 结果

运行TestModel.py可以直接看到动画

也可以看本文件夹下的video

7. 参考链接

1. 依旧是官网的demo

   Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.1.0+cu121 documentation

2. gym官网文档说明

   https://gymnasium.farama.org/

   https://github.com/Farama-Foundation/Gymnasium/blob/main/gymnasium/envs/classic_control/cartpole.py

   CartPole

   Cart Pole - Gymnasium Documentation (farama.org)