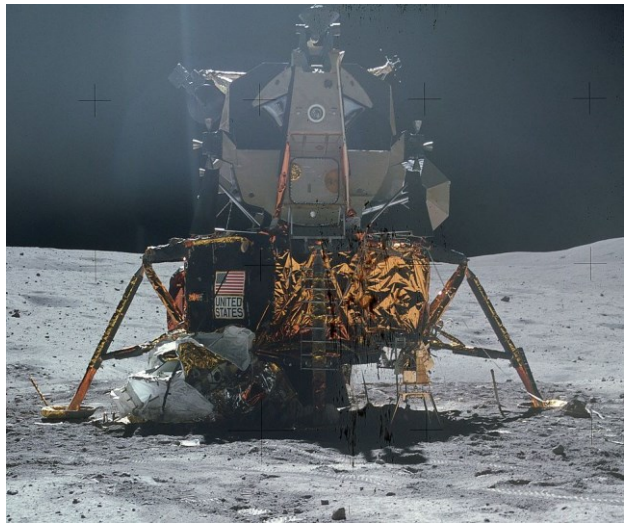
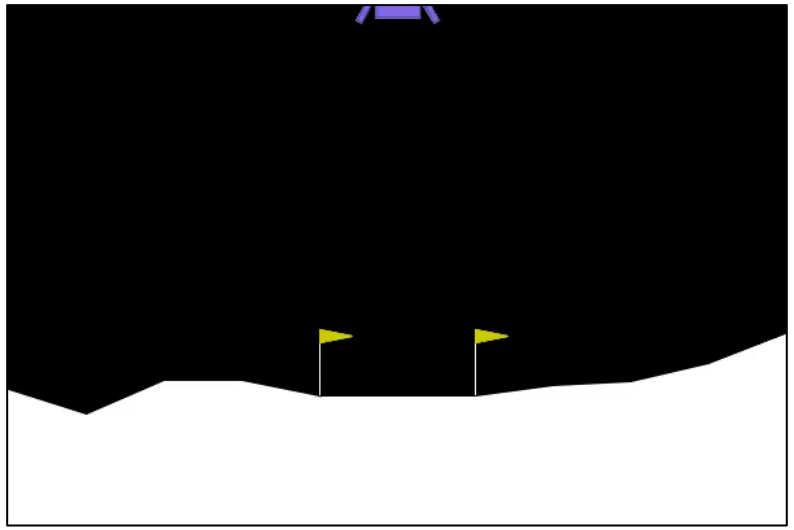
例：月球着陆器

- 受1969年美国阿波罗11号成功登月启发，**雅达利公司（Atari）**开发街机游戏《Lunar Lander》（月球着陆器），该游戏于1979年8月发布
- 玩家在游戏中控制月球登陆模块试图在月球表面平稳降落



阿波罗登月舱



月球着陆器游戏

Yunfeng Xin, Soham Gadgil, Chengzhe Xu, Solving the lunar lander problem under uncertainty using reinforcement learning, 2020 SoutheastCon, Raleigh, NC, USA, 28-29 March 2020

问题定义

- 使用强化学习方法解决OpenAI gym工具包中月球着陆器环境问题
- 环境模拟着陆器在低重力条件下需要降落到特定位置的情景，并实现定义明确的物理引擎
- 游戏主要目标：引导智能体尽可能轻柔且省油地降落到着陆垫上，状态空间如真实物理世界一样连续，但动作空间离散

月球着陆器框架

- gym，OpenAI开发的用于开发和比较强化学习算法的工具包，支持从Atari游戏到机器人技术等各种学习环境
- 使用的模拟器称为 Box2D，环境称为 LunarLander-v2

- **观测和状态空间**

观测空间决定着陆器的各种属性，状态空间有8个状态变量：

$$state \rightarrow \begin{pmatrix} \text{着陆器的}x\text{坐标} & \theta,\ \text{空间中的方向} \\ \text{着陆器的}y\text{坐标} & v_\theta,\ \text{角速度} \\ v_x,\ \text{水平速度} & \text{右腿是否触地（布尔值）} \\ v_y,\ \text{垂直速度} & \text{左腿是否触地（布尔值）} \end{pmatrix}$$

- **动作空间：**①不动作、②点燃左方向引擎、③点燃右方向引擎、④点燃主引擎
- **奖励：智能体需要在半空中保持良好姿势，并尽快到达着陆垫**

d_t–到着陆垫的距离　　v_t–智能体的速度

$$\text{Reward}(s_t) = -100 * (d_t - d_{t-1}) - 100 * (v_t - v_{t-1}) \\ -100 * (\omega_t - \omega_{t-1}) + \text{hasLanded}(s_t)$$

ω_t–智能体在时间t的角速度

hasLanded()–着陆奖励函数，表示智能体是否在着陆垫上轻柔着陆的布尔状态值的线性组合

通过奖励函数，鼓励智能体缩短到着陆垫的距离，降低速度以平稳着陆，将角速度保持在最低以防止翻滚，并且着陆后不再起飞

使用DQN训练月球着陆器步骤：

1 定义一个神经网络，输入：月球车的状态，输出：四个动作的Q值 // 使用全连接层，卷积层或其它类型的层构建网络结构
2 初始化记忆缓冲区 //用于存储月球车的经验，即状态、动作、奖励、下一个状态、是否结束的元组
3 初始化目标网络，复制神经网络的参数 // 用于计算目标Q值，避免训练过程中的不稳定
4 **repeat**
5 　重置环境，获取月球车的初始状态
6 　**repeat**
7 　　根据当前神经网络，以一定概率选择一个随机动作或一个最大Q值对应的动作
8 　　在环境中执行选择动作，观察月球车下一个状态、奖励、以及是否结束
9 　　将经验元组保存到记忆缓冲区中
10 　　如果记忆缓冲区中经验数量达到批次大小，从中随机抽取一批经验，进行学习更新
11 　　使用目标网络计算下一状态最大Q值，作为目标Q值的一部分
12 　　使用神经网络计算当前状态Q值，作为预测Q值
13 　　使用均方误差 (MSE) 作为损失函数，优化神经网络参数
14 　　使用软更新方法，将神经网络参数逐渐复制到目标网络中
15 　　记录每轮累积奖励，评估训练效果
16 　**until** 月球车着陆或坠毁
17 **until** 达到预设训练轮数或性能指标

月球着陆器

- 目标：训练智能体在 LunarLander 环境中成功着陆
- 环境与算法：OpenAI Gym 仿真环境 + 近端策略优化（PPO）算法

代码结构

文件名称	功能描述
`main.py`	训练主循环、环境交互逻辑
`ppo.py`	PPO 算法实现（策略 / 价值网络定义）

代码修改点（main.py）

- 核心参数调整：

```python
max_episodes = 50000  # 最大迭代次数
Lr = 0.002            # 学习率
update_timestep = 2000  # 策略更新间隔
eps_clip = 0.2        # PPO限幅参数
```

- 日志与保存：控制训练日志输出频率，设置模型保存条件（如奖励达标）

执行流程：

1.环境初始化 → 2. 行动选择 → 3. 反馈收集 → 4. 策略更新 → 5. 权重复制

环境初始化与状态获取

- 代码实现：

```python
env = gym.make("LunarLander-v2")  # 初始化环境
state = env.reset()               # 获取初始状态
```

- 关键参数：状态维度（8）、行动维度（4）

行动选择与执行

- 旧策略采样（main.py）：

```python
action = ppo.policy_old.act(state, memory)  # 基于旧策略选择行动
```

- 策略网络逻辑（ppo.py）：

```python
state = torch.from_numpy(state).float()  # 状态转张量
action_probs = self.action_layer(state)  # 计算行动概率
action = Categorical(action_probs).sample()  # 概率采样行动
```

环境反馈与记忆体更新

- 数据存储：

```python
memory.rewards.append(reward)      # 保存奖励
memory.is_terminals.append(done)   # 保存终止状态
```

- 隐式操作：act方法中自动保存状态与行动到记忆体

策略更新与梯度计算（PPO 核心）

- 数据预处理：提取记忆体数据并转换为张量
- 评估当前策略：计算新概率、价值、熵值
- 损失函数：

```python
ratios = torch.exp(logprobs - old_logprobs)  # 概率比率
advantages = rewards - state_values          # 优势函数
loss = -torch.min(ratios*advantages, clipped_advantages) + 价值损失 - 熵正则
```

梯度更新：反向传播更新策略网络权重

权重复制与策略迭代

- 代码实现：

```python
self.policy_old.load_state_dict(self.policy.state_dict())
```

- 作用：旧策略保留历史数据，用于下一轮迭代的稳定性控制