RL 上机实验 1

1120200822　07112002　郑子帆

1　Cliffwalking 问题

1.1　问题简介

1.1.1　游戏介绍及目标

Cliffwalking，即悬崖寻路问题，是强化学习的一个经典问题，如下图所示。该问题模型为一个 4*12 的格子图，需要智能体从左下角 S 出发，走到右下角 G 结束游戏。其中，S 和 G 中间有一个宽为 1，长为 10 的悬崖。具体规则如下：

1. 智能体每走一步有-1 分的惩罚。
2. 它不能移出网格，如果其想执行某个动作移出网格，则智能体不会移动，但仍然有-1 分的惩罚。
3. 如果智能体掉下悬崖，则有-100 分的惩罚，并从 S 位置继续游戏。
4. 当智能体到达目的地 G，游戏结束。

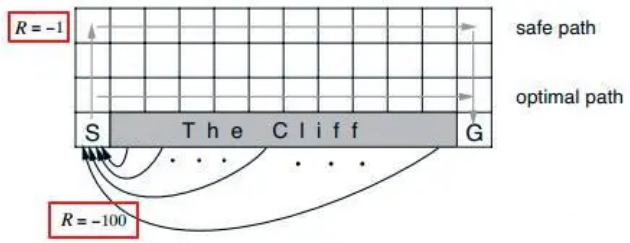

图 1: Cliffwalking 网格

游戏的目标为使智能体在不掉入悬崖的情况下尽可能快地从 S 走到 G。

1.1.2　实验环境介绍

本实验使用 gym 库中的 Cliffwalking-v0 环境。

环境中的状态表示为一维离散值 0 47，到每个格子的状态用一个数来表示，其中 S 为 36，G 为 47，编号 37-46 为悬崖。具体对应关系如下图。智能体的动作一共有四种，分别为向上、向

图 2: Cliffwalking 网格编号

右、向下、向左移动一步，分别对应 0，1，2，3。

在智能体做出一个动作后，若到达一个非悬崖位置，奖励-1；若到达悬崖奖励-100 并让智能体重新回到 S。当智能体到达 G 后，done=1，模拟结束。

1.2 算法简介

实验中一共使用了两种算法对该问题进行求解，分别为 Q-Learning 算法和 Sarsa 算法。两种算法均为表格型算法，即使用查找表 (look-up table) 的强化学习方法。这个表格称为 Q 表格，行数等于其状态数，列数等于其动作数。具体地，在 Cliffwalking 问题中 Q 表格是一个 48×4 的表格（不算状态和动作栏）。Q 表格中的每一个数对应一个 Q 函数值 $Q(s|a)$，即在状态 s 下，执行动作 a 所能获得的价值。在训练结束后，进行测试时，我们对于当前的每个状态选取 Q 表格中每行的最大值对应的动作 $(\arg\max_a Q(s,a))$。下面将分别介绍 Q-Learning 和 Sarsa 算法。

1.2.1 Q-Learning 算法

Q-Learning 算法是一个异策略 (off-policy) 算法，即生成样本的策略和更新 Q 值时使用的策略不同，生成样本时用的策略为行为策略 (behavior policy)，网络更新参数时使用的策略为目标策略 (target policy)。其中目标策略是我们最重要学习成的策略，它贪心的选择能使得当前状态下 Q 最大的动作，即 $\pi(s_t) = \arg\max_a Q(s_t, a)$。行为策略是探索环境的策略，是每次智能体下一步会走到的策略，它会探索很多可能的轨迹，采集数据并让目标策略学习，它可以是一个随机策略。在本实验中，我们采用 ϵ-贪心策略从状态 s 中选择动作 a 作为行为策略。ϵ-贪心策略，即我们设置一个随着训练轮数增加单调递减的 ϵ 值，有 $1-\epsilon$ 的概率贪心地选择让 Q 函数值最大的 a 作为当前的策略；另外有 ϵ 的概率随机决定策略。不过在进行增量式 (incremental learning) 方法更新 Q 值时，我们用目标策略对应的增量进行更新，具体公式如下：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

其中，α 为学习率，γ 为折扣因子，r 为当前状态下做出某动作的即时奖励。

1.2.2 Sarsa 算法

Sarsa 算法是一个同策略 (on-policy) 算法，即生成样本的策略和更新 Q 值时使用的策略相同。它与 1 步时序差分 (TD) 方法的区别在于它更新的是 Q 值而非 V 函数。它与 Q-learning 算法的区别在于它在更新 $Q(s_t, a_t)$ 值时利用的 Q_target 为 $Q(s_{t+1}, a_{t+1})$，具体公式如下：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

1.3 算法实现及源代码

考虑到 Q-Learning 和 Sarsa 算法较为相似，故我们写多个类以同时完成 Q-Learning 算法和 Sarsa 算法的编写。具体地，我们构建了 4 个类，分别为 Config 类，Train 类，Qlearning 类和 Sarsa 类：

1. Config 类封装了一些有关训练的超参数的值。
2. Train 类封装了 agent 的创建方法和针对不同算法（Q-Learning/Sarsa）的训练函数和测试函数。

3. Qlearning 类封装了 Q-Learning 算法下的 agent 的选择下一步动作和迭代更新 Q 表格的方法。
4. Sarsa 类封装了 Sarsa 算法下的 agent 的选择下一步动作和迭代更新 Q 表格的方法。

由于二者除了更新公式上其他并无太大的不同，故我们设置 Config 类将超参数存放于此，本实验中两个算法的超参数相同，具体如下：

超参数	含义	设定值
lr	学习率	0.05
gamma	价值的折扣因子	0.90
epsilon	e-贪心策略中的 epsilon	无
epsilon_start	e-贪心策略中的 epsilon 初始值	0.20
epsilon_end	e-贪心策略中的 epsilon 最终值	0.001
epsilon_decay	e-贪心策略中的 epsilon 衰减系数	100
Q_table	Q 表格	0
train_epi	训练轮数	1000
test_epi	测试轮数	1

在具体代码实现过程中，两个算法的区别在 train 函数中和各自类中的 update 成员函数中，下面将进行具体阐述。

1.3.1 Q-Learning 算法

在 train 函数中，Q-Learning 算法对于当前的状态用 ϵ-贪心策略选择一个动作作为行为策略，并用 $maxQ(s_{t+1}, a)$ 增量更新当前的 $Q(s_t, a_t)$。update 后要更新当前的状态和动作，即将 next_state 更新为 state。

具体训练代码如下：

```
        for i_ep in range(cfg.train_eps):
        tot_reward = 0  # 记录每一轮的奖励
        state = env.reset()  # 重置环境，开始新的一轮
        while True:
            action = agent.choose_action(state)  # 根据对应的算法选择一个动作
            next_state, reward, done, _ = env.step(action)  # 与环境进行一次交互
            agent.update(state, action, reward, next_state, done)  # 利用算法对应的公式进行更新
            state = next_state  # 将状态进行更新
            tot_reward += reward # 将当前获得的即时价值加进总奖励中
            if done:              # 如果游戏结束，则跳出循环结束这一轮
                break
        rewards.append(tot_reward)    # 将这一轮中智能体在游戏中获得的总奖励添加到列表中

```

具体 update 代码如下：

```
        def update(self, state, action, reward, next_state, done):  # 根据当前状态、动作、即时奖励、下一状态更新Q表
```

```
        Q_now = self.Q_table[str(state)][action] # Q(s, a)的值
        if done: # 终止状态
            Q_target = reward
        else:   # 非终止状态
            Q_target = reward + self.gamma * np.max(self.Q_table[str(next_state)])
    # Q_target为目标策略的Q值*折扣因子+即时奖励
        self.Q_table[str(state)][action] += self.lr * (Q_target - Q_now)     # 迭代
    更新

```

1.3.2 Sarsa 算法

在 train 函数中，Sarsa 算法先为利用 ϵ-贪心策略选择一个动作，再对于当前的状态用 ϵ-贪心策略选择一个动作作为行为策略，并用 $Q(s_{t+1}, a_{t+1})$ 增量更新当前的 $Q(s_t, a_t)$。update 后要更新当前的状态和动作，即将 next_state 和 next_action 更新为 state 和 action。

具体训练代码如下：

```
        for i_ep in range(cfg.train_eps):
        tot_reward = 0  # 记录每一轮的奖励
        state = env.reset()  # 重置环境，开始新的一轮
        action = agent.choose_action(state) # 根据对应的算法选择一个动作
        while True:
            next_state, reward, done, _ = env.step(action)  # 与环境进行一次交互
            if reward == -100:
                reward = -100
            next_action = agent.choose_action(next_state)  # 根据对应的算法选择一个
    动作
            agent.update(state, action, reward, next_state, next_action, done)  #
    利用算法对应的公式进行更新
            state = next_state  # 将状态进行更新
            action = next_action     # 更新当前动作
            tot_reward += reward # 将当前获得的即时价值加进总奖励中
            if done:             # 如果游戏结束，则跳出循环结束这一轮
                break
        rewards.append(tot_reward)    # 将这一轮中智能体在游戏中获得的总奖励添加到列
    表中

```

具体 update 代码如下：

```
        def update(self, state, action, reward, next_state, next_action, done):  #
    根据当前状态、动作、即时奖励、下一状态更新Q表
        Q_now = self.Q_table[str(state)][action] # Q(s, a)的值
        if done: # 终止状态
            Q_target = reward
        else:   # 非终止状态
            Q_target = reward + self.gamma * self.Q_table[str(next_state)][
    next_action] # Q_target为下一状态下一动作对应的Q值*折扣因子+即时奖励
```

```
7             self.Q_table[str(state)][action] += self.lr * (Q_target - Q_now)     # 迭代
    更新
8
```

1.4 实验结果

对于 Q-Learning 算法，我们绘制了每一轮的奖励随游戏轮数的关系图，具体如下。可以发现

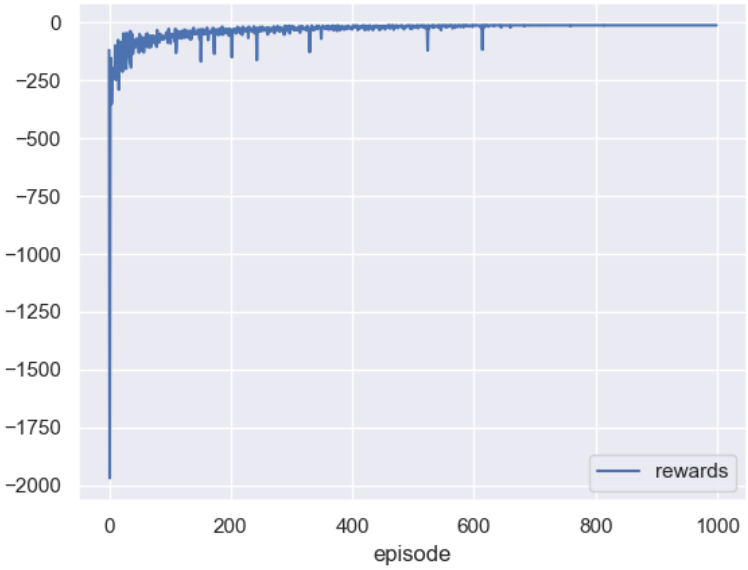

图 3: Q-Learning 奖励随轮数的关系

最终奖励收敛于-13，说明 Q-Learning 算法找到了悬崖寻路的最优解。在训练后的测试中，可通过 env.render() 显示出智能体的具体行进路线，这里没有使用 CliffwalkingWrapper 进行美化，部分截图如下。

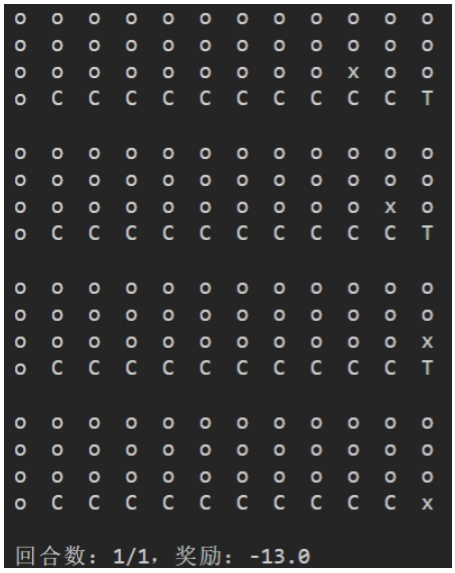

图 4: Q-Learning 智能体行为截图

对于 Sarsa 算法，我们绘制了每一轮的奖励随游戏轮数的关系图，具体如下。

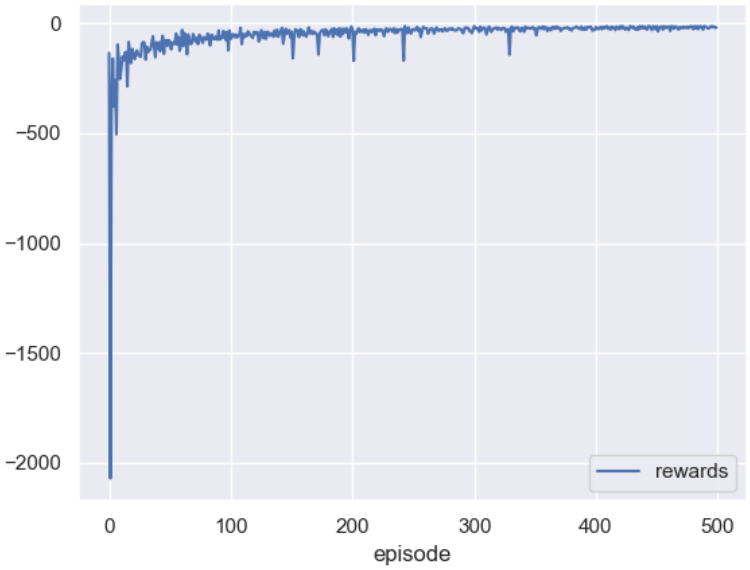

图 5: Sarsa 奖励随轮数的关系

可以发现最终奖励收敛于-13，说明 Sarsa 算法也找到了悬崖寻路的最优解。但是进行多次测试后有可能收敛在-15，不过大部分情况下收敛在-13。通过书上讲解和网上查询资料，为了体现 Sarsa 算法的保守性，我们将掉入悬崖的奖励改成-1000，即增加 10 倍，甚至增加 100 倍，再次训练智能体，发现最终的奖励一直收敛于-15，可视化智能体的动作可以发现，智能体避开了悬崖行走，即多往上走了一格，这样更保险保守，相比 Q-Learning，这也体现出了 Sarsa 算法更加保守。对应的具体关系图和过程截图如下。

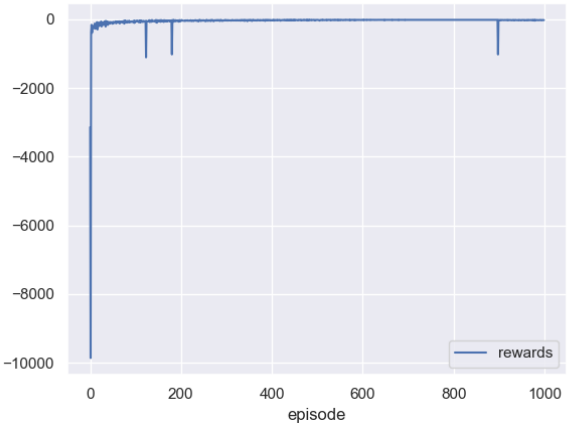

图 6: Sarsa 奖励随轮数的关系

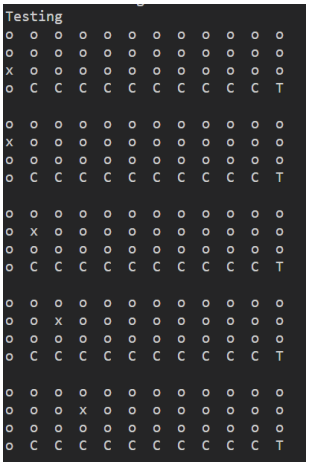

图 7: Sarsa 智能体行为截图 (掉入悬崖奖励-1000)

2 Cartpole-v0 问题

2.1 问题简介

2.1.1 游戏介绍及目标

Cartpole，即车杆游戏，是一个经典的控制类问题。模型由一辆车和车上的一个竖直的杆在 2 维平面构成，杆通过一个车上安装的转轴与车相连，可以向平面内的任意方向进行摆动。

游戏目标为让智能体控制小车的左右移动，使得杆保持直立的时间尽可能长。

2.1.2 实验环境介绍

本实验使用 gym 库中的 Cartpole-v0 环境。其中状态共有四维，具体值如下。

数值	观测	最小值	最大值
0	车位置	-2.4	2.4
1	车速	-inf	inf
2	杆角度	-12 度	12 度
3	杆顶端的速度	-inf	inf

动作有两种，分别为向左和向右，用 0 和 1 表示。

小车每走一步，包括终止步骤，奖励均为 1。

游戏满足如下条件之一则结束。

1. 杆角度超出范围。
2. 小车的中心位置超出边界。
3. 小车累计步数大于 200。

2.2 算法简介

本实验主要使用了 Sarsa 算法和 DQN(深度 Q 学习网络) 算法和多层感知机。

2.2.1 Sarsa 算法

Sarsa 算法的具体介绍详见 2.2.2。这里的 Sarsa 算法中和悬崖寻路问题中的不同点在于如何将状态离散化，在这里我们将 4 维的状态先各自离散化到 0-3 中，再通过简单的 hash 方法将状态表示成一个 0-255 之间的值。当然，这样做会使得小车在一些不同的状态但最后离散化成了一个值，丧失了一些它原本的特性，故会稍微影响学习效果。另外，这里利用 ϵ-贪心策略较为简单，为 $\epsilon(t+1) = \epsilon(t) \times 0.97$。具体地，通过多次的参数调整，超参数的设置值如下。

超参数	含义	设定值
lr	学习率	0.10
gamma	价值的折扣因子	0.90
Q_table	Q 表格	0
epsilon	epsilon 初始值	0.2
epsilon_decay	epsilon 衰减率	0.97
train_epi	训练轮数	2000
test_epi	测试轮数	10

2.2.2 DQN 算法

对于此问题，状态是连续的，且量较大，此时值函数无法用一张表格来表示，适合 DQN 算法，利用函数逼近的方法表示值函数。

DQN 算法是 Q-Learning 算法的进一步提升，主要在以下三个方面：

1. DQN 中不再是 Q 表格，而是“真正的”Q 函数，它的 Q 函数值利用神经网络（比如最基本的多层感知机 MLP）进行逼近求解。
2. 在原来的 Q-learning 算法中，每一个数据只会用来更新一次值。DQN 算法采用了经验回放（experience replay）方法，具体做法为维护一个回放缓冲区 (replay buffer)，将每次从环境中采样得到的五元组数据（状态、动作、奖励、下一状态、done）存储到回放缓冲区中，训练 Q 网络的时候再从回放缓冲区中随机采样若干数据来进行训练。这样可以使样本满足独立假设，并且经验回放池中的数据通常并不只来自一个策略，性质较为多样，保证了智能体更多的探索性，学习效果往往会更好。
3. DQN 利用了卷积神经网络，其更新方法是 SGD，因此值函数更新从 Q-Learning 中的增量更新实际上变成了监督学习的一次梯度更新的过程。

网络训练的过程遵循以下公式：

$$Q_\pi\left(s_t, a_t\right)=r_t+Q_\pi\left(s_{t+1}, \pi\left(s_{t+1}\right)\right)$$

如下图，在训练过程中我们希望等式左侧的结果逐渐向右侧逼近，但是如果等式两侧的结果出自同一个模型，每次迭代过程等式两侧的结果都在不断变化，会导致学习的不稳定。所以可以先固定住右侧，只更新左侧网络的参数，在一定次数后再将右侧网络的参数更新为左侧的。此时右侧网络就称为目标网络。

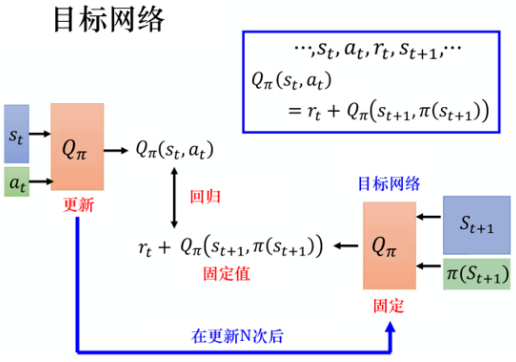

图 8: DQN 模型图

经过调参，具体参数值如下表。

超参数	含义	设定值
lr	MLP 学习率	0.01
gamma	价值的折扣因子	0.99
Q_table	Q 表格	0
epsilon	e-贪心策略中的 epsilon	无
epsilon_start	e-贪心策略中的 epsilon 初始值	0.10
epsilon_end	e-贪心策略中的 epsilon 最终值	0.001
epsilon_decay	e-贪心策略中的 epsilon 衰减系数	1000
batch_size	批量大小	64
hidden_dim	隐藏层神经元数	256
train_epi	训练轮数	500
test_epi	测试轮数	30

2.2.3 多层感知机

除了上述的强化学习算法和深度强化学习算法，我们还可以考虑直接用深度学习算法解决这一问题。这里我们考虑使用最为简单的多层感知机（Multilayer Perceptron，MLP）来完成实验。

多层感知机是最简单的神经网络，包括一个输入层、隐藏层和输出层。除输入层外，所有神经元都采用一个非线性激活函数。训练该模型需要使用反向传播（Backpropagation）算法进行监督学习。由于参数丰富，多层感知机具备比传统方法（如逻辑回归、支持向量机等等）更强的表达和学习能力。

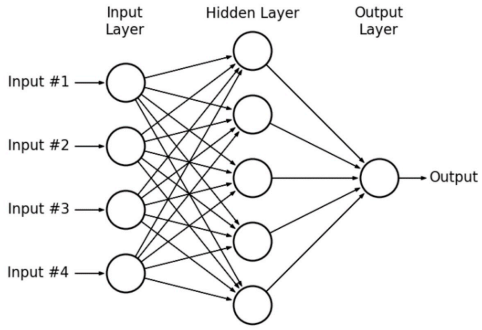

图 9: 多层感知机的模型示意图

但是书中曾经提到过，强化学习中基本不直接采用监督学习中的一些方法是因为智能体的决策还会收到和环境多轮交互的影响，即监督学习中只考虑当前状态会丢失很多有关前面的状态和动作的有价值的信息。不过在车杆问题中，我们从很直观的角度去想，杆如果要往左倒了那么车可能要向左挪，不然向右会加速杆倒；杆要往右倒同理。所以直观来看，当前小车所采取的动作和多轮之前的关系并不是很大，只不过是多轮之前的很多次小车的动作叠加到一起会影响小车和杆的当前状态。

这里我们假设各个状态和动作对之间都是独立的，（个人理解）即有所破坏马尔可夫过程（Markov Process，MP）。那么对于每一对状态-动作对，我们可以将 4 维的状态作为输入，动作 0/1 作为标签进行监督学习。至于如何得到较高质量的数据集，我们考虑先让小车随机移动，设定一个时间阈值，如果小车在一轮游戏中的时间超过这一阈值，则将这一轮的状态-动作对作为

训练数据。当得到足够多的训练样本时，则可训练 MLP 模型。

至于为什么上一个实验没有采用这一算法，因为上一实验的状态只有一维，过于简单，不易采用神经网络的方法。而车杆问题的状态维数较多，同时又都是连续变量而非离散变量，且动作空间是离散值，这让我们可以将当前问题转化成一个二分类（根据当前状态分往左还是往右）问题。

在本实验中，我们取小车游戏时间超过 120ms 的回合作为训练数据，总共取 100000 条训练数据。通过参数调整，其他参数设置如下：

超参数	含义	设定值
lr	MLP 学习率	0.05
train_epoch	训练迭代次数	5000
test_epoch	测试次数	100

在代码实现中，我们继承 nn.Module 类定义了 MLP 类，并通过筛选出的策略的状态-动作对作为训练集，编写 train 和 test 函数，部分代码和注释截图如下：

```python
def get_train_data(env, expected_score=100):
    '''
        生成N次游戏的训练数据，并选择 > 100 的数据作为训练集
    '''
    data_X, data_Y, scores = [], [], []
    for i in range(100000):
        X, Y, score = run_one_episode(env)  # 智能体随机动作进行一次游戏
        # 如果这轮游戏得分超过我们的预期得分，则作为训练数据
        if score > expected_score:
            data_X += X
            data_Y += Y
            scores.append(score)
    print('dataset size: {}'.format(len(data_X)))
    return np.array(data_X), np.array(data_Y)
```

图 10: get_train_data 函数

```python
def train(train_epoch, model, lr, data_X, data_Y):
    '''
        训练模型，梯度下降
    '''
    loss = []
    get_loss = nn.CrossEntropyLoss()        # loss函数选用交叉熵函数
    optimizer = optim.SGD(params = model.parameters(), lr = lr) # 优化器选用SGD优化器
    for idx in range(train_epoch):
        pred_y = model(data_X)      # MLP预测
        optimizer.zero_grad()       # 梯度清零
        tot_loss = get_loss(pred_y, data_Y)
        loss.append(tot_loss.sum().item())
        tot_loss.backward()     # loss反向传播
        optimizer.step()        # 优化器迭代更新参数
    return loss
```

图 11: train 函数

2.3 实验结果

2.3.1 Sarsa 算法

对于 Sarsa 算法，我们进行了 2000 轮的训练和 10 次的测试，具体小车得分如下图。

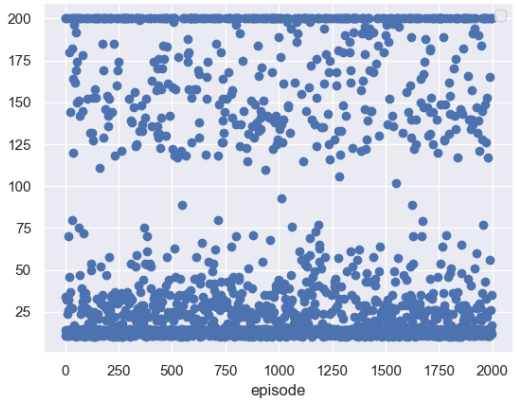

图 12: Sarsa 训练结果

```
Episode 0/10: 14.000000 time steps
Episode 1/10: 10.000000 time steps
Episode 2/10: 35.000000 time steps
Episode 3/10: 33.000000 time steps
Episode 4/10: 37.000000 time steps
Episode 5/10: 15.000000 time steps
Episode 6/10: 143.000000 time steps
Episode 7/10: 21.000000 time steps
Episode 8/10: 200.000000 time steps
Episode 9/10: 14.000000 time steps
```

图 13: Sarsa 测试结果

可以看到，小车训练所获得的奖励方差较大，效果并不是很好。在 10 次测试中仅有 1 次达到了胜利条件。不过再进行多次运行尝试后，有一次训练效果较好，测试结果如下图。究其具体原因，我认为是因为这一算法在这个问题上如果在刚开始有一个“错误”的决策，即 Qtable 的值偏离了“正确的”Qtable 时，则智能体 agent 会在一个当前状态下选一个 Q 值最大的动作，即之前探索过的动作，尽管从上帝视角这个动作不一定是最优的（个人理解），这样导致智能体**缺乏探索**，导致模型在某一个较低区间内波动，训练效果较差如上图。但是若刚开始的决策较为正确，则模型很快就能收敛，大概在第 20 轮左右就能稳定在 200ms，下图所展示的这次运行中模型在第 100 次就大概达到了 200ms 左右，收敛速度较快。

```
Episode 0/10: 200.000000 time steps
Episode 1/10: 182.000000 time steps
Episode 2/10: 200.000000 time steps
Episode 3/10: 152.000000 time steps
Episode 4/10: 124.000000 time steps
Episode 5/10: 155.000000 time steps
Episode 6/10: 143.000000 time steps
Episode 7/10: 200.000000 time steps
Episode 8/10: 199.000000 time steps
Episode 9/10: 154.000000 time steps
```

图 14: Sarsa 算法较好测试结果

进一步探究可以发现，大多数情况下当前 Sarsa 算法的学习效果仍然不是很好，在一个较低奖励的区间内波动。究其原因，因为不管当前游戏结束与否，智能体都会得到 1 的奖励，所以在不同的状态下评论员无法判断出向左或者向右这一动作的优劣，这样不能让智能体“有意识地”避开游戏结束这一状态。

为了解决这一问题，仿照 Cliffwalking 中调整智能体掉入悬崖的 reward，在这里我们将游戏结束的最后一步所获得的奖励设置为-1000。其他条件不变，进行 1000 轮游戏的学习，和之前训练的效果对比如下图。

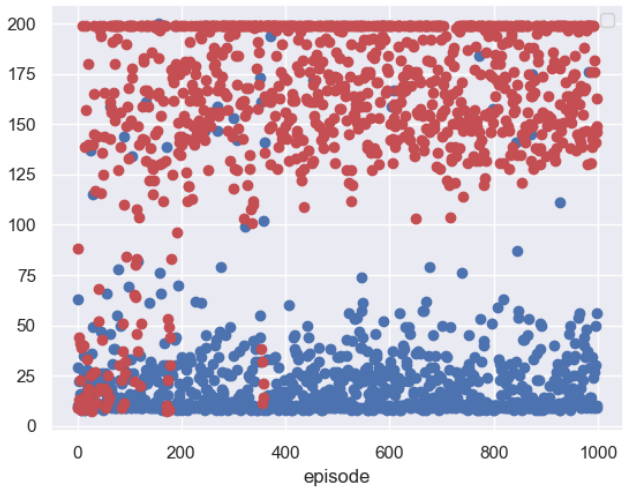

图 15: Sarsa 算法最后一步奖励 1 和-1000 的训练对比

其中红点是将最后一步奖励改为-1000 的每回合游戏总时长；蓝点是原本的每回合游戏总时长。从图中我们可以清楚的看到，原来奖励都为 1 时用 Sarsa 算法计算结果不够理想，只有个别轮能够保持 100ms 以上；但当把最后一步奖励设置的足够低后，智能体在 200 轮左右就能够学习到基本稳定保持 150ms 以上的方法，二者对比效果显著。可见对于 Sarsa 这种同策略算法，我们需要尽可能把我们不想出现的状态/结果所能获得奖励调的比其他状态低得多才能让智能体“意识到”导致这一状态的某些动作比较不好，从而在学习过程中尽可能“避开”这些状态，以获得更好的游戏结果和更高的奖励总和。

2.3.2 DQN 算法

2.3.2.1 基本结果展现

按照上述的参数设置，训练和测试结果如下。

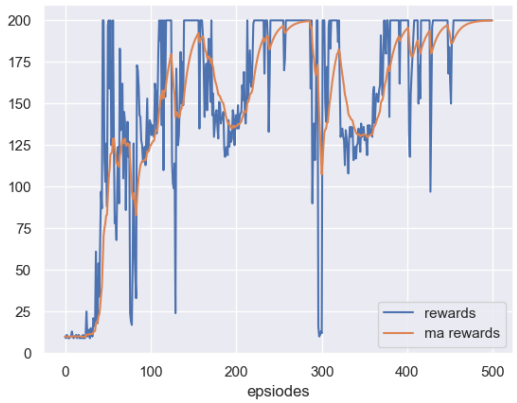

图 16: DQN 训练结果

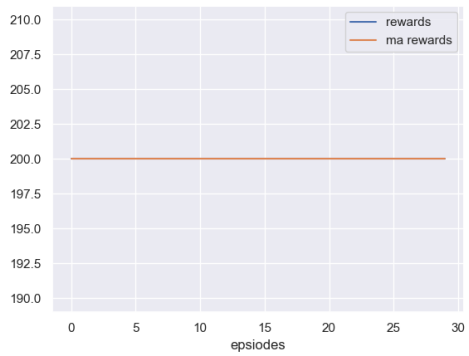

图 17: DQN 测试结果

可见随着迭代的进行奖励值逐渐向 200 收敛，测试效果也较好。说明训练后的模型已经能控制小车让杆子保持直立，达到了实验目的。

2.3.2.2 参数实验

虽然上述模型已经基本较好地完成了本项实验，但是我们往往可以通过参数实验，调整参数探究模型的收敛速度和学习效果，查看是否能够有进一步更深层的发现。

探究网络学习率对模型训练结果的影响。固定其他参数不变，调整 lr 的值从 0.001 逐渐增长至 0.3，观察模型测试时 30 轮的游戏得分（在这里为了缩短训练时间，我将模型的训练轮数改为 300 轮）。结果如下图：

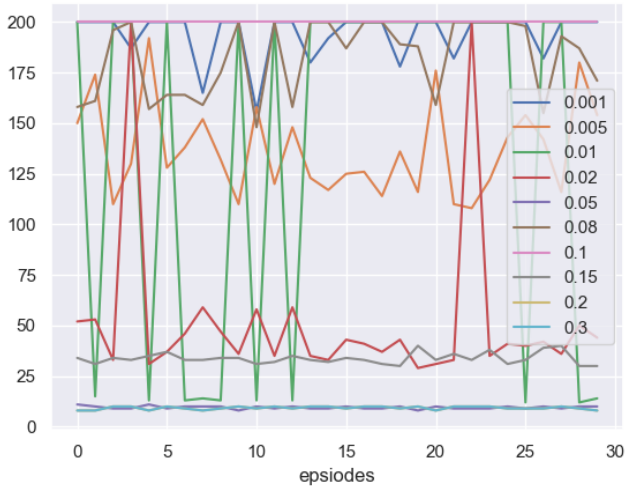

图 18: scores-lr 关系图

从整体来看，当学习率较小时学习效果更好一些，说明在这种类似“猫追老鼠”的更新方式时，不应调大学习率，因为之后根据更新周期 Q_target 网络也还会随着变化，如果学习率过大会使得“猫”的轨迹过于追随着“老鼠”走，即 Q_predict 过于趋同于 Q_target，这样变化波动较大，不够稳定。具体地，当 lr=0.1 时学习效果最好，故实验时我们取学习率为 0.01。

2.3.3 多层感知机

我们设置了 5000 轮的训练，每轮训练和当前对应的 loss 值如下图：

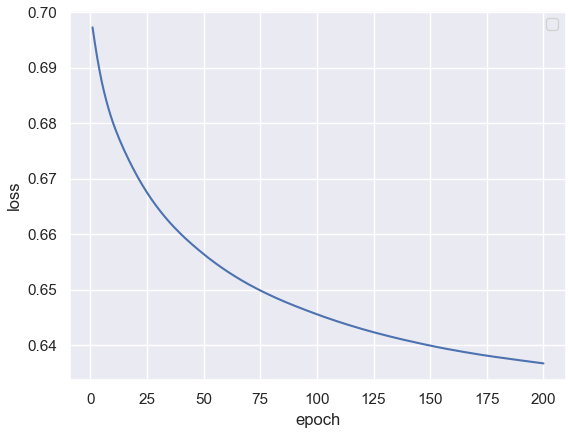

图 19: loss-epoch 关系图

可以看到 loss 随着训练轮数的增加有较为平滑的下降，说明训练效果都不错。

利用训练好的模型进行 10 次测试，结果如下：

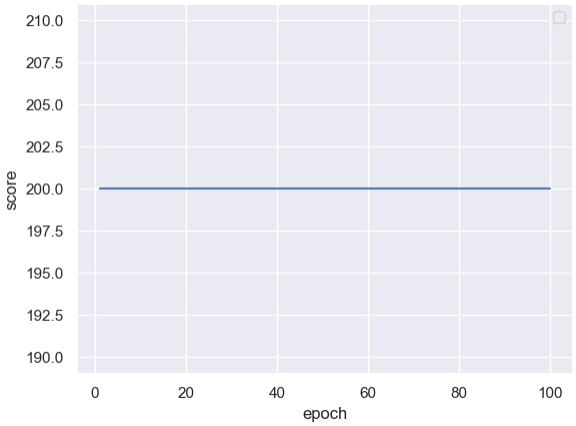

图 20: 模型测试

10 次测试智能体均能坚持 200ms，进一步验证了模型的效果较好。

3 FrozenLake 问题

3.1 问题简介

3.1.1 游戏介绍及目标

FrozenLake，即滑冰问题，是强化学习的一个经典控制类问题。该问题模型为一个 4*4 的格子图，需要智能体从左上角 S 出发，走到右下角 G 结束游戏。其中，中间有一些格子为窟窿（H），有些格子是安全的、能走的（F）。具体规则如下：

1. 智能体每走一步奖励都为 0；当智能体到 G 时的奖励为 1。
2. 它不能移出网格，如果其想执行某个动作移出网格，则智能体不会移动。
3. 智能体的移动方向部分依赖于它的决策动作，即这个游戏中考虑风的影响，例如智能体决策向上走，那么它也有一定概率走到左边或者右边的格子。
4. 当智能体到达目的地 G 或者掉下窟窿 H，游戏都结束。

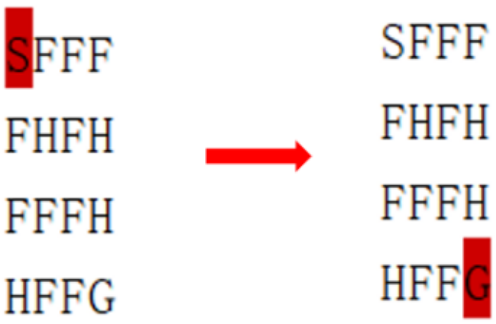

图 21: FrozenLake 游戏地图

游戏的目标为使智能体安全地从 S 走到 G。

3.1.2 实验环境介绍

本实验使用 gym 库中的 FrozenLake-v1 环境（在 gym 库中找不到 FrozenLake-v0 环境）。

环境中的状态表示为一维离散值 0 15，每个格子的状态用一个数来表示，其中 S 为 0，G 为 15，窟窿的状态是 $\{5, 7, 11, 12\}$。智能体的动作一共有四种，分别为向左、向下、向右、向上移动一步，分别对应 0，1，2，3。

在智能体做出一个动作后，若到达一个非 G 的位置，奖励 0；若到达 G，奖励 1。当智能体到达 H 或 G 后，done=1，模拟结束。

通过阅读 FrozenLake-v1 的源代码我们可以发现，程序中设置，当智能体做出某决策过程后，智能体实际会等概率地做出除了与决策动作完全相反的动作的动作。例如，智能体决策向左走，那么最终智能体各有 $\frac{1}{3}$ 的概率向左、向上、向下走。

3.2 算法简介

这个问题和 Cliffwalking 问题有很多相似的地方，我们固然可以用 Q-Learning 和 Sarsa 这些表格型方法去求解。但这里我们考虑用基于动态规划（dynamic programming，DP）的强化学习方法。基于动态规划的强化学习方法主要有两种，策略迭代和价值迭代，这里考虑使用策略迭代的方法。

在《Easy RL》书中第三章提到过，强化学习算法可以分为有模型和免模型。当我们知道状态转移概率时（比如这个问题和 Cliffwalking 问题），我们可以利用动态规划进行求解。策略迭代算法由两个主要部分组成：策略评估（policy evaluation）和策略提升（policy improvement）。其中，策略评估使用贝尔曼期望方程得到一个策略下的价值函数；策略提升通过计算 Q 函数值更新当前的策略，可以证明这样的策略对应的期望收益不劣于之前的。宏观可以将此算法理解为：先有一个策略 π_{old}，然后根据这个策略和贝尔曼期望方程得到这个策略下的价值函数 $V*$，然后再根据 $V*$ 计算 Q 函数得到每个状态下的最优动作，从而得到新的决策 π，以此往复最终策略收敛。具体的算法伪代码如下：

while $\Delta > \theta$ **do**：（策略评估循环）
  $\Delta \leftarrow 0$
  对于每一个状态 $s \in \mathcal{S}$:
    $v \leftarrow V(S)$
    $V(s) \leftarrow r(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s))V(s')$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
end while
$\pi_{\text{old}} \leftarrow \pi$
对于每一个状态 $s \in \mathcal{S}$:
  $\pi(s) \leftarrow \arg\max_a r(s, a) + \gamma \sum_{s'} P(s'|s, a)V(s')$
若$\pi_{\text{old}} = \pi$, 则停止算法并返回 V 和 π; 否则转到策略评估循环

图 22: 策略迭代伪代码

3.3 算法实现

按照上图中伪代码进行 policy_Iteration、policy_evaluation 和 policy_improvement 函数的编写，具体代码及注释截图如下：

```python
def policy_Iteration(env, theta):   # 策略迭代主函数
    theta = theta                   # 设置theta值
    obs_dim = env.observation_space.n
    act_dim = env.action_space.n
    policy = np.zeros(obs_dim)  # 初始化policy
    V = np.zeros(obs_dim)       # 初始化V函数
    iteration_epoch = 0
    while True:
        iteration_epoch += 1
        V = policy_evaluation(env = env, theta = theta, V = V, policy = policy)     # 策略评估
        new_policy, policy_stable = policy_improvement(env = env, V = V, old_policy = policy)

        if policy_stable:      # 如果相邻两次策略相同，说明收敛，结束
            return iteration_epoch, new_policy
        else :                 # 否则继续迭代
            policy = new_policy
```

图 23: policy_Iteration 函数

```python
def policy_evaluation(env, theta, V, policy):   #策略评估，并更新V函数
    obs_dim = env.observation_space.n
    act_dim = env.action_space.n
    while True:
        delta = 0
        old_V = np.copy(V)
        for s in range(obs_dim):                # 枚举状态
            action = policy[s]                  # 当前决策的动作
            v = 0
            for p, next_state, reward, done in env.env.P[s][action]:    # 枚举不同状态之间转移的概
                v += p * (reward + old_V[next_state])   # 计算当前状态在决策动作下获得的价值的期望
            V[s] = v                            # 更新V函数
            if delta < abs(V[s] - old_V[s]):
                delta = abs(V[s] - old_V[s])
        if delta < theta:
            return V
```

图 24: policy_evaluation 函数

```python
def policy_improvement(env, V, old_policy, policy_stable = True):   # 策略改进，根据Q值重选策略
    obs_dim = env.observation_space.n
    act_dim = env.action_space.n
    policy = np.zeros(obs_dim)
    for s in range(obs_dim):
        old_action = policy[s]
        q_value = np.zeros(act_dim)
        for action in range(act_dim):
            for p, next_state, reward, done in env.env.P[s][action]:
                q_value[action] += p * (reward + V[next_state]) # 利用更新完的V函数计算Q值
        policy[s] = np.argmax(q_value)  # 选出当前状态下Q值最大的动作作为新的策略
        if policy[s] != old_policy[s]:  # 如果策略收敛，退出
            policy_stable = False
```

图 25: policy_improvement 函数

3.4 实验结果

最终智能体学习到的在各状态下的策略表为 [0. 3. 3. 3. 0. 0. 0. 0. 3. 1. 0. 0. 0. 2. 1. 0.]，可视化策略如下（H 表示窟窿）：

<	^	<	^
<	H	<>	H
^	v	<	H
H	>	V	G

图 26: FrozenLake 可视化决策

注意到策略迭代为代码中有一个 θ 用于限制每次迭代 V 函数的更新，保证其能够收敛。下面探究不同的 θ 值会不会在本次实验中产生影响。分别将 θ 值设定为 1e-1 到 1e-15，探究训练后智能体在 100 轮中的平均得分和训练时的迭代次数如下图。

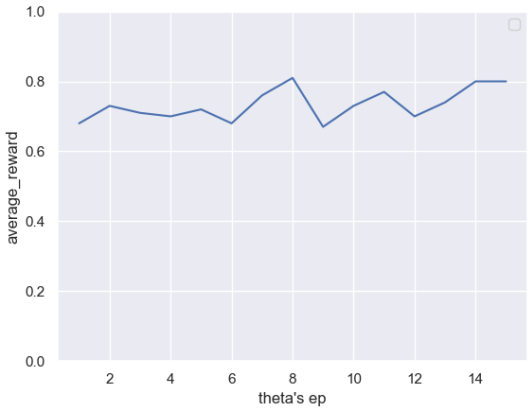

图 27: 智能体随 θ 值在每 100 轮中的得分

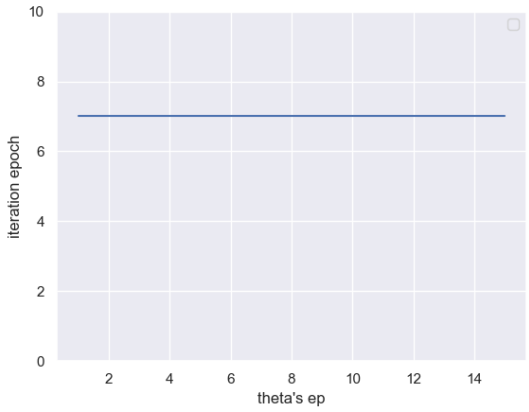

图 28: 智能体随 θ 值在训练中的迭代轮数

从上图中可以看出，上述 θ 取值对智能体的训练效果基本没有影响，并且所有迭代轮数均为 7。进一步分析，我认为这是因为每次 V 函数更新效果差距不大，所以最后得到了唯一的策略，这一策略是固定不变的，所以去除随机带来的影响，不同 θ 值训练得到的智能体的“学习效果”是一样好的。

4 说明与心得

4.1 实验环境

实验电脑为惠普暗影精灵 6，处理器 i7 10750H，Windows 10 家庭版，64 位操作系统，并搭配有 NVIDIA GeForce RTX 2080s 独立显卡。

本实验的开发环境为 Anaconda+vscode，在 vscode 中安装了相应的支持插件，并在 Anaconda 环境管理中新建了环境 gym，其中安装了本实验主要用到的 gym、pytorch、numpy、pyplot 等包。

4.2 心得（写于 10.6 课后）

在小组交流过程中，以及课上王九铮同学小组的汇报中都提到了可以直接用神经网络去解决这个问题。这让我也想到了不管是《Easy RL》书中还是老师一开课中的 ppt 都提到了强化学习是一个从强化学习方法走向深度强化学习的过程，这也使得我在课后进一步从纯深度学习的角度出发思考 CartPole 问题，并且用纯 MLP 方法进行了实验。

另外，在阅读《Easy RL》这本书的前 7 章的过程中，我发现它是按照有模型中的动态规划，包括策略迭代和价值迭代方法，再到免模型中的改进的蒙特卡洛、时序差分法，再到将 TD 中的 V 函数改成求 Q 表格的 Sarsa 法，再到将同策略该进成异策略的 Q-Learning 法，再到加入深度学习、将增量迭代改进成神经网络梯度下降的 DQN 算法这一由浅入深、由易到难的顺序进行了讲解和展示。这一清晰的思路也让我对强化学习基础内容有了一个更深、更清楚的认识，并在头脑中形成了一个清晰的思维导图。

于是我想到这次试验除了运用免模型的一些思路，我们还可以去实践一下有模型的思路，故了解到了 gym 库中的 FrozenLake 游戏，进行了动态规划法的实现求解。在完成后，我进行了进一步的思考，我认为 CliffWalking 游戏也可以用有模型方法去解决，因为它的状态转移概率是确定的，每一步都是 1。不过由于时间原因我没有调出这部分的代码，以后会继续完善并学习。

从小组展示和小组成员的交流中我也学习到了很多东西，比如一块探究 Sarsa 算法在 Cartpole 问题中的求解，罗熙同学能够想到将最后一步的 reward 调整成-100+steps 步，避免统一调整成-100 而在多轮训练后弱化了它的“警示”作用。再比如张远方同学通过查找资料学习了优先级经验回放这一技巧，结合了 sum tree 这一数据结构进一步提高了模型收敛的速度和学习效果。

总的来说，这次上机实验花了很多的时间，也有很大的收获。

4.3 其他说明

本实验中 Cliffwalking 的 Q-Learning 代码、Cartpole 的 DQN 代码均参考《EASY RL》并后续自己进行了编写。实验中的 Sarsa 代码均根据书中思路和相关的 Q-Leaning 代码自己完成编写。

在实验报告中进行了顺序的调换，因为本人认为 Cliffwalking 相对更容易更好实现所以先完成了这一实验，后进行了 Cartpole 实验，故实验报告先写 Cliffwalking 再写 Cartpole。

另外，在每个实验中的算法顺序是按照本人的实践编写顺序所写，而非按照难易程度递增顺序。