

实验指导书

任务3 基于时序差分的路径规划

冰湖路径规划

目 录

[一、实验信息 1](#_Toc133738938)

[二、实验准备 1](#_Toc133738939)

[1.实验环境 1](#_Toc133738940)

[2.实验数据 4](#_Toc133738941)

[三、样例代码 4](#_Toc133738942)

[1. 时序差分方法介绍 4](#_Toc133738943)

[2. Sarsa算法 5](#_Toc133738944)

[3.Q-Learning算法 8](#_Toc133738945)

[四、实验内容 11](#_Toc133738946)

[五、思考题 12](#_Toc133738947)

一、实验信息

|  |  |
| --- | --- |
| 实验名称 | 基于时序差分的路径规划 |
| 实验等级 | 中阶 |
| 实验时长 | 2课时 |
| 实验内容 | 1. 在样例代码的基础上，实现n-step TD算法，比较分析n的不同取值对策略收敛速度的影响。（由于随机性建议多次实验，可以选择sarsa或者Q-learning修改） 2. 在样例代码的基础上，实现TD(λ)算法，比较分析λ的不同取值对策略收敛速度的影响。（由于随机性建议多次实验，可以选择sarsa或者Q-learning修改） |
| 实验目标 | 1. 掌握时序差分方法的内涵  2. 掌握Sarsa算法的设计与实现  3. 掌握Q-Learning算法的设计与实现  4. 掌握n-step TD算法与TD(λ)算法的实现。理解超参数n与λ对算法效果的影响。 |

二、实验准备

### 1. 实验环境

|  |  |  |
| --- | --- | --- |
| 序列 | 名称 | 规格/版本 |
| 硬件 | PC计算机或服务器 | 1G以上内存，40G 以上硬盘，有英伟达GPU更佳 |
| 开发语言 | Python | 3.8.13 |
| 强化学习环境 | gym | 0.24.1 |
| 数值计算 | Numpy | 1.23.0 |

**【注意】较新的gym版本中，与环境交互方式会有不同导致代码报错：**

**obs, info = env.reset()**

**state, reward, done, truncated, info = env.step(action)**

#### 1.1 环境安装

本次实验将基于OpenAI的Gym环境。OpenAI是一家非营利性的人工智能研究公司，公布了非常多的学习资源以及算法资源。其之所以叫作OpenAI，是因为他们把所有开发的算法都进行了开源。OpenAI 的 Gym库是一个环境仿真库，里面包含很多现有的环境。针对不同的场景，我们可以选择不同的环境。

为了下载并安装 OpenAI Gym，我们可以使用以下方法：

|  |
| --- |
| pip install gym # minimal install  pip install gym[all] # full install  pip install gym[atari] # for Atari specific environment installation |

#### 1.2 环境介绍

在本实验中将会用到Gym中的冰湖环境（Frozen Lake），冰湖环境是一个网格环境，大小为 ，见图1-1。每一个方格对应了一个状态，智能体起点状态在左上角，目标状态在右下角，中间还有若干冰洞。在每一个状态都可以采取上、下、左、右 4 个动作。由于智能体在冰面行走，因此每次行走都不一定按预定目标方向移动，而是有一定的概率滑行到附近的其它状态。当掉入冰洞或到达目标状态时结束。每一步行走的奖励是 0，到达目标的奖励是 1。



我们可以通过如下代码查看一条随机轨迹：

|  |
| --- |
| import gym  env = gym.make("FrozenLake-v1") # 创建环境  env.reset()  for t in range(100):  env.render() # 渲染画面  a = env.action\_space.sample() # 随机采样动作  observation, reward, done, \_ = env.step(a) # 环境执行动作，获得转移后的状态、奖励以及环境是否终止的指示  if done:  break  env.render() |

### 实验数据

无。

三、样例代码

### 1. 时序差分方法介绍

时序差分是一种用来估计一个策略的价值函数的方法，它结合了蒙特卡洛和动态规划算法的思想。时序差分方法和蒙特卡洛的相似之处在于可以从样本数据中学习，不需要事先知道环境；和动态规划的相似之处在于根据贝尔曼方程的思想，利用后续状态的价值估计来更新当前状态的价值估计。回顾一下蒙特卡洛方法对价值函数的增量更新方式：

这里我们将 替换成了 ，表示对价值估计更新的步长。可以将 取为一个常数，此时更新方式不再像蒙特卡洛方法那样严格地取期望。蒙特卡洛方法必须要等整个序列结束之后才能计算得到这一次的回报 ，而时序差分方法只需要当前步结束即可进行计算。具体来说，时序差分算法用当前获得的奖励加上下一个状态的价值估计来作为在当前状态会获得的回报，即：

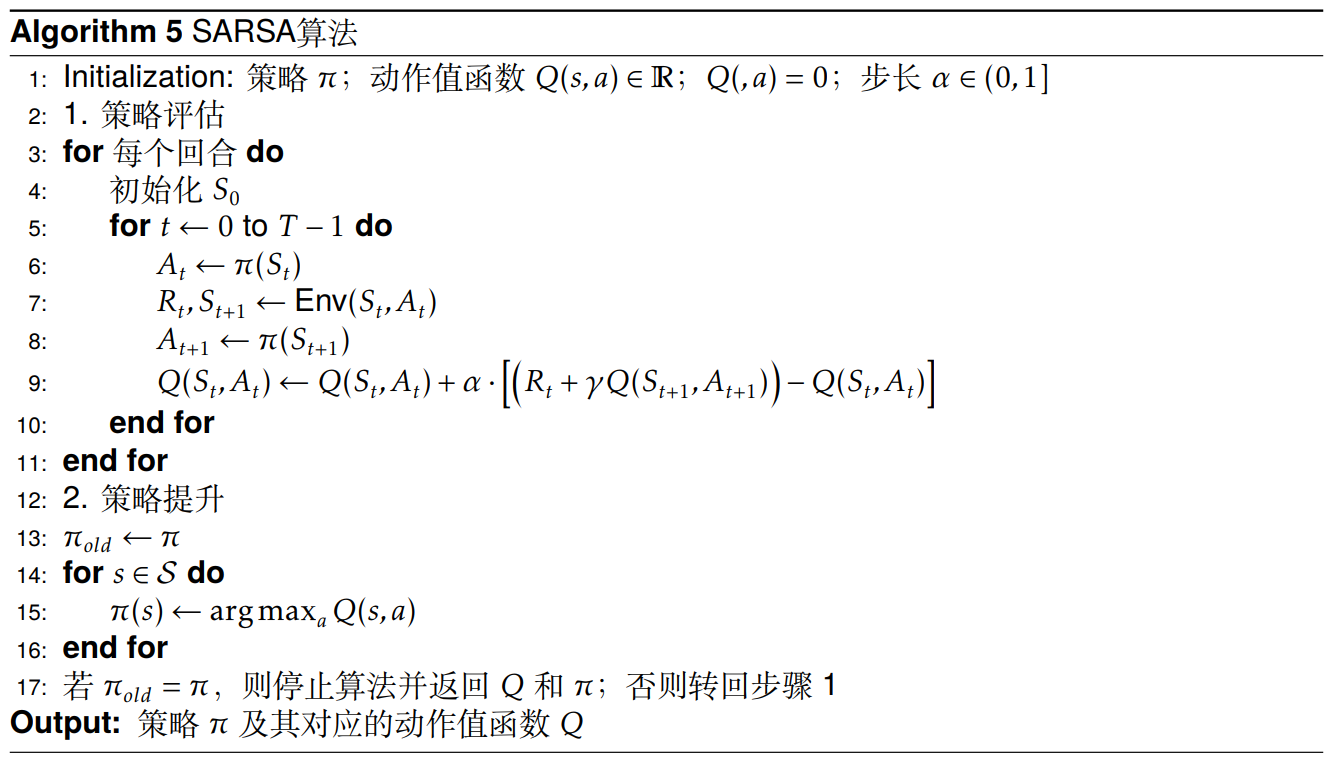
被称为TD error（temporal difference error），时序差分算法将其与步长的乘积作为状态价值的更新量。

### 2. Sarsa算法

在引入时序差分方法后，我们就可以利用其进行策略评估。与上一节实验类似，我们并不知道环境的转移概率与奖励函数，因此直接估计价值函数难以进行策略提升步骤。在这里依旧是对动作价值函数进行评估：

为了确保对环境进行充分的探索，这里也需要采取ε-贪心策略：

如果采用时序差分进行策略评估，利用ε-贪心策略提升就得到了Sarsa算法。下面给出Sarsa算法的伪代码，



#### 2.1 基于Sarsa路径规划算法实现：

|  |
| --- |
| import numpy as np  import gym  env = gym.make("FrozenLake-v1") # 创建环境  env.reset()  def sarsa(env, num\_episodes=1000, alpha=0.1, gamma=0.99, eps0=1, decay=0.001):  # 初始化Q表为0  Q = np.zeros((env.observation\_space.n, env.action\_space.n))  # 针对每个回合进行更新  for i\_episode in range(num\_episodes):  # 初始化状态  state = env.reset()  # 使用epsilon-greedy策略选择动作  epsilon = eps0 / (1 + decay \* i\_episode)  print(f"iteration: {i\_episode}, epsilon: {epsilon}")  action = epsilon\_greedy(Q, state, env.action\_space.n, epsilon)  # 针对每个时间步进行更新  while True:  # 执行选定的动作  next\_state, reward, done, \_ = env.step(action)  # 使用epsilon-greedy策略选择下一个动作  next\_action = epsilon\_greedy(Q, next\_state, env.action\_space.n, epsilon)  # 计算TD误差  td\_target = reward + gamma \* Q[next\_state, next\_action] \* (not done)  td\_error = td\_target-Q[state, action]  # 更新Q表  Q[state, action] += alpha \* td\_error  # 更新状态和动作  state = next\_state  action = next\_action  if done:  break  # 返回最终的Q表和策略  policy = np.argmax(Q, axis=1)  return Q, policy  def epsilon\_greedy(Q, state, num\_actions, epsilon):  if np.random.random() < epsilon:  # 随机选择动作  return np.random.choice(num\_actions)  else:  # 选择最优动作  return np.argmax(Q[state, :]) |

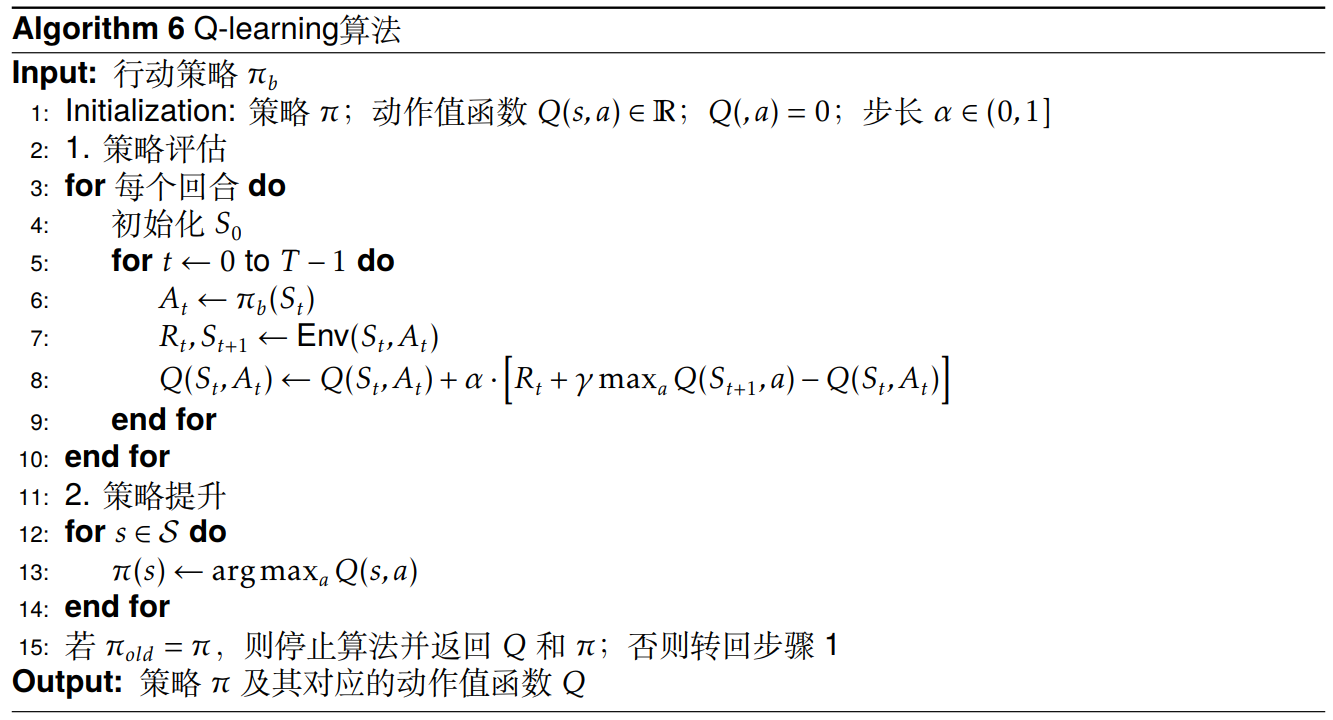
#### 2.2 策略的性能测试

|  |
| --- |
| def test\_pi(env, pi, num\_episodes=100):  """  测试策略。  参数：  env -- OpenAI Gym环境对象。  pi -- 需要测试的策略。  num\_episodes -- 进行测试的回合数。  返回值：  成功到达终点的频率。  """  count = 0  for e in range(num\_episodes):  ob = env.reset()  for t in range(100):  a = pi[ob]  ob, rew, done, \_ = env.step(a)  if done:  count += 1 if rew == 1 else 0  break  return count / num\_episodes  # Q, pi = sarsa(env, num\_episodes=10000)  # result = test\_pi(env, pi)  # print(result) |

### Q-Learning算法

在Sarsa算法中，用于采样的策略是由动作价值函数导出的ε-贪心策略，而所更新的也是该ε-贪心策略对应的动作价值函数。这种用于采样（行为策略）和被评估的策略（目标策略）为同一种策略的算法，我们称之为同策略算法。与Sarsa不同，Q-Learning算法是一种异策略算法。其行为策略为当前动作价值函数导出的ε-贪心策略，而目标策略为当前动作价值函数导出的贪心策略。

下面给出Q-Learning算法的伪代码，



#### 3.1 基于Q-Learning算法实现：

|  |
| --- |
| def q\_learning(env, num\_episodes=1000, alpha=0.05, gamma=0.99, eps0=1, decay=0.001):  # 初始化Q表为0  Q = np.zeros((env.observation\_space.n, env.action\_space.n))  # 针对每个回合进行更新  for i\_episode in range(num\_episodes):  # 初始化状态  state = env.reset()  # 使用epsilon-greedy策略选择动作  epsilon = eps0 / (1 + decay \* i\_episode)  # print(f"iteration: {i\_episode}, epsilon: {epsilon}")  # 针对每个时间步进行更新  while True:  # 使用epsilon-greedy策略选择下一个动作  action = epsilon\_greedy(Q, state, env.action\_space.n, epsilon)  # 执行选定的动作  next\_state, reward, done, \_ = env.step(action)  # 根据Q表选取最优动作，用于计算TD目标  next\_action = np.argmax(Q[next\_state, :])  # 计算TD误差  td\_target = reward + gamma \* Q[next\_state, next\_action] \* (not done)  td\_error = td\_target - Q[state, action]  # 更新Q表  # print(state, action, td\_error)  Q[state, action] += alpha \* td\_error  # 更新状态  state = next\_state  if done:  break  if i\_episode % 10000 == 0:  print(Q)  print(epsilon)  # 返回最终的Q表和策略  policy = np.argmax(Q, axis=1)  return Q, policy  def epsilon\_greedy(Q, state, num\_actions, epsilon):  if np.random.random() < epsilon:  # 随机选择动作  return np.random.choice(num\_actions)  else:  # 选择最优动作  return np.argmax(Q[state, :]) |

#### 3.2 性能测试

|  |
| --- |
| def test\_pi(env, pi, num\_episodes=100):  """  测试策略。  参数：  env -- OpenAI Gym环境对象。  pi -- 需要测试的策略。  num\_episodes -- 进行测试的回合数。  返回值：  成功到达终点的频率。  """  count = 0  for e in range(num\_episodes):  ob = env.reset()  for t in range(100):  a = pi[ob]  ob, rew, done, \_ = env.step(a)  if done:  count += 1 if rew == 1 else 0  break  return count / num\_episodes  # Q, pi = q\_learning(env, num\_episodes=10000)  # result = test\_pi(env, pi)  # print(result) |

四、实验内容

样例代码提供了sarsa算法与Q-learning算法的标准实现。这两种算法均仅考虑了同一个回合上相邻两个状态之间价值的关系。n-step TD是对TD算法的改进，通过多次使用Bellman方程展开状态价值，使得同一个回合上更靠前的状态的价值也能被用于计算更新当前状态的价值，而TD(λ)是将不同n值的n-step TD算法的更新值加权平均，用来更新当前状态的价值。这两种算法使得值能够更快地在状态之间传播，从而加快学习速率。

实验要求：

1. 在示例代码的基础上，尝试实现n-step TD算法，比较并分析n的不同取值对策略收敛速度的影响（由于随机性建议多次实验）。

2. 在示例代码的基础上，尝试实现TD(λ)算法，比较并分析λ的不同取值对策略收敛速度的影响（由于随机性建议多次实验）。

实验完成后需要填写实验报告，说明对代码的理解以及修改思路，列出不同配置下的实验结果并分析可能的原因。

五、实验思考

问题1. 蒙特卡洛方法和时序差分方法是无偏估计吗？另外谁的方差更大呢？为什么？

蒙特卡洛方法是无偏估计，时序差分方法是有偏估计；蒙特卡洛方法的方差较大，时序差分方法的方差较小，原因在于时序差分方法中使用了自举，减小了随机变量的数目，导致估计的价值函数的方差更小。