

实验指导书

任务1 基于动态规划的路径规划

冰湖路径规划

目 录

[一、实验信息 1](#_Toc144218420)

[二、实验准备 1](#_Toc144218421)

[1. 实验环境 1](#_Toc144218422)

[2. 实验数据 4](#_Toc144218423)

[三、样例代码 4](#_Toc144218424)

[1. 导入相关包并初始化环境 4](#_Toc144218425)

[2. 策略评估算法的实现 4](#_Toc144218426)

[3. 策略迭代算法的实现 5](#_Toc144218427)

[4. 价值迭代算法的实现 5](#_Toc144218428)

[四、实验内容 5](#_Toc144218429)

[五、思考题 5](#_Toc144218430)

一、实验信息

|  |  |
| --- | --- |
| 实验名称 | 基于动态规划的路径规划 |
| 实验等级 | 中阶 |
| 实验时长 | 2课时 |
| 实验内容 | 1. 尝试运行并理解样例代码 2. 在样例代码的基础上修改，至少实现一种异步动态规划算法，并且和原算法比较策略收敛需要的迭代次数。（样例代码已经实现了原位动态规划，需要实现优先级动态规划或者实时动态规划） |
| 实验目标 | 1. 理解动态规划的基本思想  2. 掌握利用贝尔曼方程评估价值函数  3. 掌握策略迭代算法  4. 掌握价值迭代算法  5. 掌握异步动态规划算法 |

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 |

#### 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 |

这里需要注意gym的版本，本实验中所使用的版本为0.24.1，其他版本的环境可能会导致代码运行出错。

**【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)**

**旧版的 gym 交互格式中没有truncated字段。**

#### 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。



图1-1 冰湖环境

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() |

我们可以通过如下代码获取该环境的状态转移函数与奖励函数：

|  |
| --- |
| env = env.unwrapped  P = env.P |

P为一个字典，对于每一个状态动作对（state, action），P[state][action]返回一个形如（probability, next\_state, reward, terminal）的元组，其中：

* + probability为在状态“state”采取动作“action”转移到状态“next\_state”的概率。
  + next\_state为转移到的下一个状态。
  + reward为发生状态转移所获得的奖励。
  + terminal用于指示next\_state是否为终止状态。

### 实验数据

无。

三、样例代码

### 1. 导入相关包并初始化环境

|  |
| --- |
| import gym # openAi gym  import numpy as np  import warnings  warnings.filterwarnings('ignore')  env = gym.make("FrozenLake-v1")  env.reset() |

### 2. 策略评估算法的实现

策略评估过程用于计算一个策略的状态价值函数。回顾之前学习的贝尔曼期望方程:

其中， 是策略 在状态 下采取动作 的概率。可以看到，当知道奖励函数和状态转移函数时，我们可以根据下一个状态的价值来计算当前状态的价值。

为了评估策略，得到状态价值函数，我们首先设定任意初始值 ，并使用贝尔曼期望方程进行迭代更新：

而根据贝尔曼期望方程的性质，可以得知 是以上更新公式的一个不动点（fixed point）。事实上，可以证明当 时，序列 会收敛到 ，所以可以据此来计算得到一个策略的状态价值函数。在实际的实现过程中，如果某一轮 的值非常小了，就可以提前结束策略评估。

具体而言，我们在本次任务需要实现policy\_evaluation函数，用于计算获得马尔可夫决策过程中任意策略 的状态价值函数 。

|  |
| --- |
| def policy\_evaluation(policy, env, gamma=1.0, theta=0.00001):  """  实现策略评估算法，给定策略与环境模型，计算该策略对应的价值函数。  参数：  policy：维度为[S, A]的矩阵，用于表示策略。  env：gym环境，其env.P表示了环境的转移概率。  env.P[s][a]为一个列表，其每个元素为一个表示转移概率以及奖励函数的元组(prob, next\_state, reward, done)  env.observation\_space.n表示环境的状态数。  env.action\_space.n表示环境的动作数。  gamma：折扣因子。  theta：用于判定评估是否停止的阈值。    返回值：长度为env.observation\_space.n的数组，用于表示各状态的价值。  """    nS = env.observation\_space.n  nA = env.action\_space.n  # 初始化价值函数  V = np.zeros(nS)  while True:  delta = 0  for s in range(nS):  v\_new = 0  for a in range(nA):  for prob, next\_state, reward, done in env.P[s][a]:  # TODO: 计算用于更新 V[s] 的 v\_new    delta = max(delta, np.abs(V[s]-v\_new))  V[s] = v\_new  # 误差小于阈值时终止计算  if delta < theta:  break    return np.array(V) |

### 3. 策略迭代算法的实现

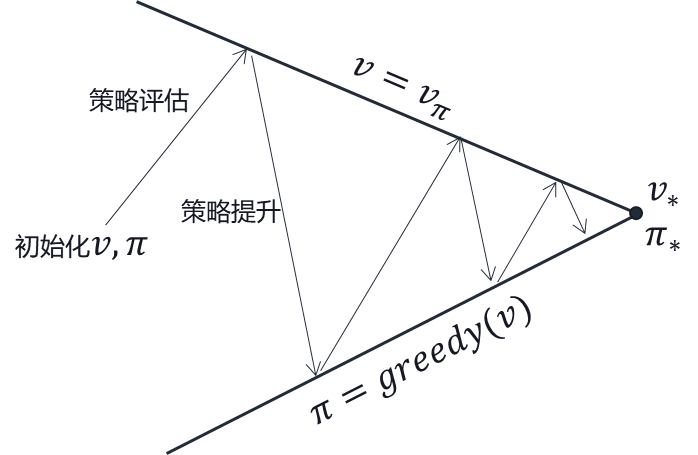
使用策略评估计算得到当前策略的状态价值函数之后，我们可以据此来改进该策略。假设此时对于策略 ，我们已经知道其价值 ，也就是知道了在策略下从每一个状态出发最终得到的期望回报。我们要如何改变策略来获得在状态 下更高的期望回报呢？假设智能体在状态 下采取动作 ，之后的动作依旧遵循策略 ，此时得到的期望回报其实就是策略 下的动作价值 。如果此时满足 ，则说明在状态 下采取动作 会比原来的策略 得到更高的期望回报。以上假设只是针对一个状态，现在假设存在一个确定性策略 ，在任意一个状态下，都满足:

于是在任意状态下，我们有,

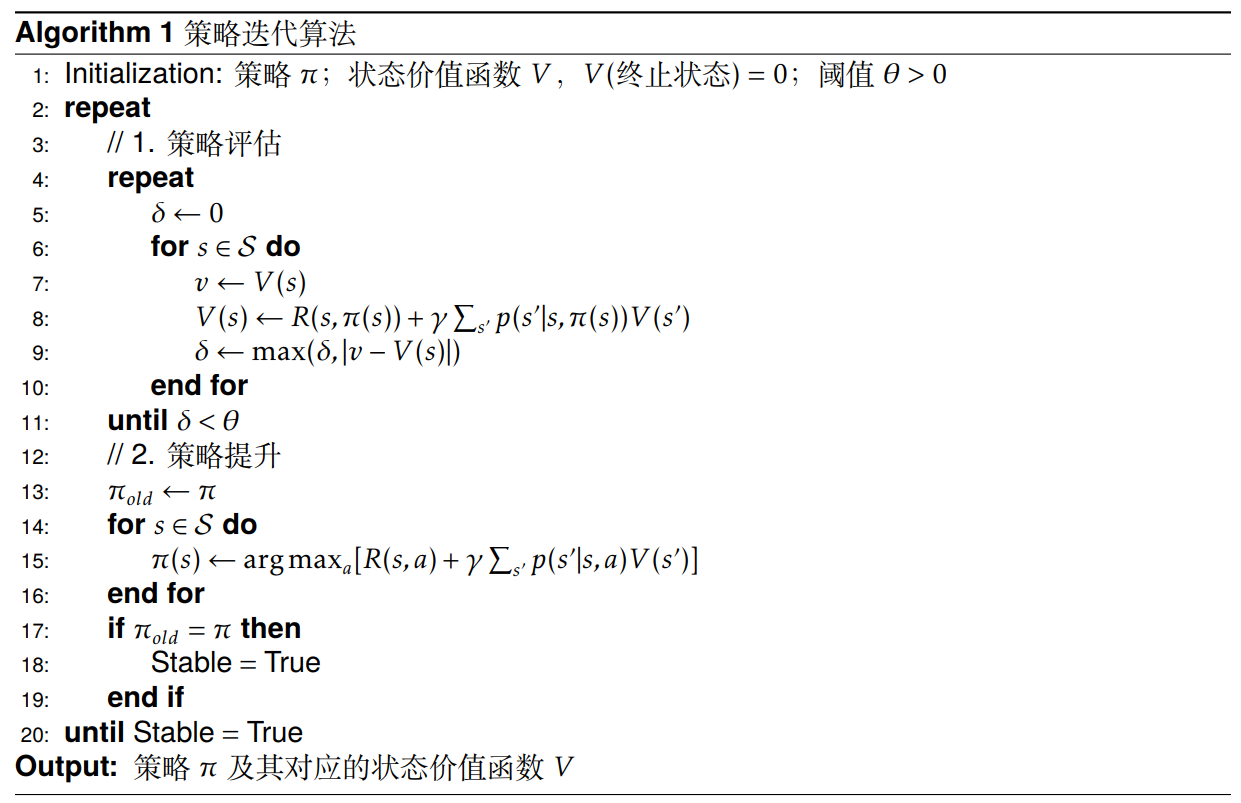
因此我们可以直接贪心地在每一个状态选择动作价值最大的动作，获得一个新的策略，也就是，

我们可以知道，策略 π′ 比策略 π 更好或至少一样好。

总体来说，策略迭代算法的过程如下：对当前的策略进行策略评估，得到其状态价值函数，然后根据该状态价值函数进行策略提升以得到一个更好的新策略，接着继续评估新策略、提升策略……直至最后收敛到最优策略：



结合策略评估与策略提升，我们可以得到策略迭代算法：



下面我们实现策略迭代算法，如下所示。

|  |
| --- |
| def policy\_iteration(env, policy\_eval\_fn=policy\_evaluation, gamma=1.0):  """  实现策略提升算法，迭代地评估并提升策略，直到收敛至最优策略。  参数：  env：gym环境。  policy\_eval\_fn：策略评估函数。  gamma：折扣因子。  返回值：  (policy, V)  policy为最优策略，由维度为[S, A]的矩阵进行表示。  V为最优策略对应的价值函数。  """  nS = env.observation\_space.n  nA = env.action\_space.n  def one\_step\_lookahead(state, V):  """  对于给定状态，计算各个动作对应的价值。    参数：  state：给定的状态 (int)。  V：状态价值，长度为env.observation\_space.n的数组。    返回值：  每个动作对应的期望价值，长度为env.action\_space.n的数组。  """  A = np.zeros(nA)  for a in range(nA):  for prob, next\_state, reward, done in env.P[state][a]:  # TODO: 计算动作 a 的价值 A[a]  return A  # 初始化为随机策略  policy = np.ones([nS, nA]) / nA    num\_iterations = 0  while True:  num\_iterations += 1    V = policy\_eval\_fn(policy, env, gamma)  policy\_stable = True    for s in range(nS):  old\_action = np.argmax(policy[s])  q\_values = one\_step\_lookahead(s, V)  new\_action = np.argmax(q\_values)  if old\_action != new\_action:  policy\_stable = False    policy[s] = np.zeros([nA])  policy[s][new\_action] = 1  if policy\_stable:  print(num\_iterations)  return policy, V |

简单进行测试。

|  |
| --- |
| env.reset()  policyPI,valuePI = policy\_iteration(env, gamma=0.95)  print(policyPI)  print(valuePI) |

### 4. 价值迭代算法的实现

策略迭代中的策略评估需要进行很多轮才能收敛得到某一策略的状态函数，这需要很大的计算量，尤其是在状态和动作空间比较大的情况下。我们是否必须要完全等到策略评估完成后再进行策略提升呢？试想一下，可能出现这样的情况：虽然状态价值函数还没有收敛，但是不论接下来怎么更新状态价值，策略提升得到的都是同一个策略。可见，策略迭代算法进行了一些无谓的计算，还有进一步优化的空间。策略迭代算法如果只在策略评估中进行一轮价值更新，然后直接根据更新后的价值进行策略提升，这样是否可以呢？答案是肯定的，这其实就是本节将要讲解的价值迭代算法，它可以被认为是一种策略评估只进行了一轮更新的策略迭代算法。需要注意的是，价值迭代中不存在显式的策略，我们只维护一个状态价值函数。

下面我们实现价值迭代算法。

|  |
| --- |
| def value\_iteration(env, theta=0.0001, gamma=1.0):  """  实现价值迭代算法。    参数：  env：gym环境，其env.P表示了环境的转移概率。  env.P[s][a]为一个列表，其每个元素为一个表示转移概率以及奖励函数的元组(prob, next\_state, reward, done)  env.observation\_space.n表示环境的状态数。  env.action\_space.n表示环境的动作数。  gamma：折扣因子。  theta：用于判定评估是否停止的阈值。    返回值：  (policy, V)  policy为最优策略，由维度为[S, A]的矩阵进行表示。  V为最优策略对应的价值函数。  """  nS = env.observation\_space.n  nA = env.action\_space.n    def one\_step\_lookahead(state, V):  """  对于给定状态，计算各个动作对应的价值。    参数：  state：给定的状态 (int)。  V：状态价值，长度为env.observation\_space.n的数组。    返回值：  每个动作对应的期望价值，长度为env.action\_space.n的数组。  """  A = np.zeros(nA)  for a in range(nA):  for prob, next\_state, reward, done in env.P[state][a]:  # TODO: 计算动作 a 的价值 A[a]  return A    V = np.zeros(nS)    num\_iterations = 0    while True:  num\_iterations += 1  delta = 0    for s in range(nS):  q\_values = one\_step\_lookahead(s, V)  new\_value = np.max(q\_values)    delta = max(delta, np.abs(new\_value - V[s]))  V[s] = new\_value    if delta < theta:  break    policy = np.zeros([nS, nA])  for s in range(nS):  q\_values = one\_step\_lookahead(s,V)    new\_action = np.argmax(q\_values)  policy[s][new\_action] = 1    print(num\_iterations)  return policy, V |

简单进行测试。

|  |
| --- |
| env.reset()  policyVI,valueVI = value\_iteration(env, gamma=0.95)  print(policyVI)  print(valueVI) |

我们可以通过运行如下代码，查看策略迭代算法与价值迭代算法的最终策略差异。

|  |
| --- |
| nS = env.observation\_space.n  nA = env.action\_space.n  same\_policy = True  for s in range(nS):  if same\_policy == False:  break  for a in range(nA):  if policyPI[s][a] != policyVI[s][a]:  same\_policy=False  break  if same\_policy:  print("策略迭代算法与价值迭代算法的最终策略一致。")  else:  print("策略迭代算法与价值迭代算法的最终策略不一致。") |

四、实验内容

样例代码中实现了值迭代和策略迭代动态规划算法的框架，学生需要补全代码中的空缺，在正确补全的基础上记录实验效果，按照实验报告提供的指引完成实验报告。

五、思考题

问题1. 如果数据流不具备马尔可夫性质怎么办？应该如何处理？

参考答案：

如果不具备马尔可夫性，即下一个状态与之前的状态也有关，若仅用当前的状态来求解决策过程，则难以作出最优决策。为了解决这个问题，可以重新设计状态特征，将历史状态信息纳入当前状态。也可以利用循环神经网络对历史信息建模，获得包含历史信息的状态表征，表征过程也可以使用注意力机制等手段，最后在表征状态空间求解马尔可夫决策过程问题。