《博弈论与强化学习》课程设计报告

课程设计 1：策略迭代和值迭代方法实践

姓名	李昀哲		学号	20123101
学院	计算机工程与科学学院		专业	智能科学与技术
得分	代码得分(30)		总分	
	汇报得分(30)			
	报告得分(40)			

指导教师评语：

课程设计目的及内容：

强化学习强调如何基于环境而行动，来取得最大化的预期收益，广泛应用于复杂不确定的环境中，智能体通过与环境的互动学习从而使所获得的累计奖励最大化。强化学习智能体包含多个成分：策略（Policy）、价值函数（Value function）和模型（Model），分别用以表示智能体的行为、评判每个状态或动作的好坏、表征智能体对环境的理解。

其中又可以根据是否有模型（Model）对强化学习进行分类，分为基于模型（Model-Based）的强化学习和免模型（Model-Free）的强化学习；根据智能体的学习内容又可以分为基于值的智能体（Value-based agent）、基于策略的智能体（Policy-based agent）、演员评论家智能体（Actor-critic agent）。

马尔可夫决策过程（Markov Decision Process）描述了强化学习的框架，如图 1 所示。其中，策略评估是非常重要的，那如何快速评估一个策略，并且如何找到最佳策略就是值得关注的问题，分别被称为预测问题和控制问题。动态规划是一个比较通用的解决方案，MDP 又满足动态规划的最优子结构和重叠子问题的两个性质，因此可以用于解决策略评估和寻找

最优策略。

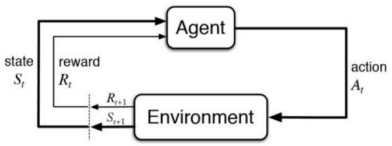

图 1 马尔可夫决策过程

对于 MDP 的控制问题即找到最优策略，除了穷举法，常通过值迭代（Value iteration）和策略迭代（Policy iteration）解决。

课堂中对两种方法进行了理论讲解，本实验将对两种方法进行算法实现从而对比两种方法在寻找最优价值函数的问题中的优劣，以帮助我们在未来遇到类似问题时，更合理的选择解决办法，更高效地解决实际问题。

课程设计过程中遇到的问题以及解决方法？

问题 1：贝尔曼方程中的 max...到底是对于什么而言的？不太理解 max 中的式子的含义。即使通过 Stanford 值迭代的可视化例子也难以理解。

通过课程设计，再次回归到贝尔曼方程的基础理论，max 中的并非在方格中找到最大的 value，而是对于方格中每个格子在每个状态下的所有可能的 action 中选择让 value 最大的方案，想通之后再看公式就变得迎刃而解，可视化例子也变得十分直观。因此，通过课程设计让我再一次强化了对理论的理解，才能更好地实践。

问题 2：策略迭代的实现过程为什么和值迭代有差异。

从迭代方式上就可以看出，策略迭代的每一步迭代过程并不求 max，只是通过迭代不断更新方格中的值，得到新的 value function，基于这个新的 value function 更新策略表，直到 value 收敛于某个值，所以是个反复 evaluate 和 improve 的过程；而值迭代会直接找出最佳的 value，所以会有 max。因此，对于状态空间比较小的情况，通过很少的策略评估就能获得最优的价值函数，所以比较快。通过实践过后的理解课程中的理论和两种迭代的差异，课程项目是不可或缺的。

问题 3：为什么在状态空间较小的情况下仍有可能出现策略迭代比值迭代更慢的情况？

运行的时间还可能与设置的 upper_bound 相关，当设置的收敛阈值较小时，所需的精度更高，迭代的次数和时间自然会需要的更长。

本次课程设计的体会

贝尔曼方程给我们了一个寻找最优值的理论基础，基于此分别进行策略迭代和值迭代，本课程设计根据两种方案分别进行算法验证，测试不同情况下算法的可行性和优劣性。首先参考理论推导，编写两种迭代算法的代码；再改变空间大小、收敛阈值等参数，对两种算法进行对比测试，观察影响因素，如表 1 所示。

表 1 策略迭代和值迭代算法测试

	Number/Time(s)					
upper_bound	1	0.1	0.01	1	0.1	0.01
matrix_width		3			9	
Policy_iteration	**0.005**	0.03054	0.05201	0.18425	0.27611	0.34269
Value_iteration	0.012	**0.02262**	**0.03738**	**0.09959**	**0.09993**	**0.20581**

测试发现，除了空间大小对运行时间和迭代次数有影响，收敛阈值同样起到关键作用。当阈值较大、空间较小时，策略迭代更好，但随着阈值减小即所需精度更高的情况，值迭代都表现出了更好的结果，且差异逐渐变大，如图 2 所示

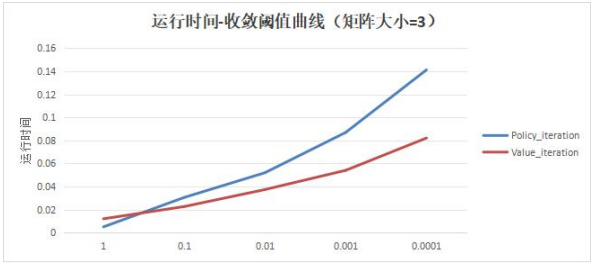

图 2 运行时间-收敛阈值曲线

运行结果部分截图如图 3 所示

```
===== Policy Evaluation =====
> iteration: 403 delta: 1.7424738864864787e-05
===== Policy Improve =====

===== Policy Evaluation =====
> iteration: 67 delta: 0.0009549992044970423
===== Policy Improve =====

===== Policy Evaluation =====
> iteration: 2 delta: 0.0007735493556424444
===== Policy Improve =====

===== Policy Evaluation =====
> iteration: 2 delta: 0.0006265749780709484
===== Policy Improve =====

===== Policy Evaluation =====
> iteration: 2 delta: 0.000507525732237113
===== Policy Improve =====

[time consumption]: 0.404832124710083 s
Evaluation: [reward] 1.0 [step] 4
```

```
===== Value Iteration =====
> iteration: 67 delta: 0.0009550049507973668
===== Policy Improve =====

[time consumption] 0.0661156177520752s
Evaluation: [reward] 1.0 [step] 4
```

图 3 运行结果部分截图

思考题：

思考题 1：动态规划中策略迭代和价值迭代的收敛速度比较

大多数情况下，策略迭代的次数更少，但值得注意的是，次数少并不意味着运行时间短。策略迭代包含策略评估和策略提升两个过程，反复迭代这两步直到策略收敛，因此所需花费的时间是非常昂贵的；相比之下值迭代的运行时间更少，值迭代主要通过寻找最优价值函数和策略提取，不需要反复策略迭代的步骤，只要找到最优的值函数，那策略一定是最优的。

因此，对于空间较小的马尔可夫决策过程而言，策略迭代迭代的次数更少，每一次迭代的时间也不长，因此策略迭代更合适；对于空间较大的马尔可夫决策过程，策略迭代在每一次迭代中需要花费大量的时间，因此值迭代更实用，效率更高。

代码附录

策略评估部分：

```
import numpy as np
from copy import copy


'''
Policy iteration: Evaluation + Improvement
env: environment;
values: current value;
policies: current policies;
upper_bound:
'''
def policy_eval(env, values, policies, upper_bound):
    print('\n===== Policy Evaluation =====')

    # Init delta and iternation number.
    delta = upper_bound
    iteration = 0

    while delta >= upper_bound:
        delta = 0.

        for state in env.states:
            value = values.get(state)
            env.set_state(state)

            action_index = policies.sample(state)
            action = env.actions[action_index]
            _, _, rewards, next_states = env.step(action)

            next_values = values.get(list(next_states))
            td_values = list(map(lambda x, y: x + env.gamma * y, rewards, next_values))

            exp_value = np.mean(td_values)
            values.update(state, exp_value)

            # update delta
            delta = max(delta, abs(value - exp_value))

        iteration += 1
        print('\r> iteration: {} delta: {}'.format(iteration, delta), flush=True, end="")
```

策略优化

```
'''
Policy iteration: Evaluation + Improvement
env: environment;
values: current value;
policies: current policies;
'''
def policy_improve(env, values, policies):
    print('\n===== Policy Improve =====')
    policy_stable = True

    for state in env.states:
        old_act = policies.sample(state)

        # calculate new policy execution
        actions = env.actions
        value = [0] * len(env.actions)

        for i, action in enumerate(actions):
            env.set_state(state)
            _, _, rewards, next_states = env.step(action)
            next_values = values.get(list(next_states))
            td_values = list(map(lambda x, y: x + env.gamma * y, rewards, next_values))
            prob = [1 / len(next_states)] * len(next_states)

            value[i] = sum(map(lambda x, y: x * y, prob, td_values))

        # action selection
        new_act = actions[np.argmax(value)]

        # greedy update policy
        new_policy = [0.] * env.action_space
        new_policy[new_act] = 1.
        policies.update(state, new_policy)

        if old_act != new_act:
            policy_stable = False

    return policy_stable
```

值迭代

```
'''
Value iteration
env: environment;
values: current value;
upper_bound:
'''
def value_iter(env, values, upper_bound):
    print('===== Value Iteration =====')
    delta = upper_bound + 1.
    states = copy(env.states)

    iteration = 0

    while delta >= upper_bound:
        delta = 0

        for s in states:
            v = values.get(s)

            # get new value
            actions = env.actions
            vs = [0] * len(actions)

            for i, action in enumerate(actions):
                env.set_state(s)
                _, _, rewards, next_states = env.step(action)
                td_values = list(map(lambda x, y: x + env.gamma * y, rewards, values.get(next_states)))

                vs[i] = np.mean(td_values)

            values.update(s, max(vs))
            delta = max(delta, abs(v - values.get(s)))

        iteration += 1
        print('\r> iteration: {} delta: {}'.format(iteration, delta), end="", flush=True)

    return
```

环境基类

```
'''
Env:
    states
    state
    actions
    gamma
'''
class Env:
    def __init__(self):
        self._states = set()
        self._state = None
        self._actions = []
        self._gamma = None

    @property
    def states(self):
        return self._states

    @property
    def state_space(self):
        return self._state_shape

    @property
    def actions(self):
        return self._actions

    @property
    def action_space(self):
        return len(self._actions)

    @property
    def gamma(self):
        return self._gamma

    def _world_init(self):
        raise NotImplementedError

    def reset(self):
        raise NotImplementedError

    def step(self, state, action):
        """Return distribution and next states"""
        raise NotImplementedError

    def set_state(self, state):
        self._state = state
```

矩阵环境类

```
class MatrixEnv(Env):
    def __init__(self, height=4, width=4):
        super().__init__()

        self._action_space = 4
        self._actions = list(range(4))

        self._state_shape = (2,)
        self._state_shape = (height, width)
        self._states = [(i, j) for i in range(height) for j in range(width)]

        self._gamma = 0.9
        self._height = height
        self._width = width

        self._world_init()

    @property
    def state(self):
        return self._state

    @property
    def gamma(self):
        return self._gamma

    def set_gamma(self, value):
        self._gamma = value

    def reset(self):
        self._state = self._start_point

    def _world_init(self):
        # start_point
        self._start_point = (0, 0)
        self._end_point = (self._height - 1, self._width - 1)

    def _state_switch(self, act):
        # 0: h - 1, 1: w + 1, 2: h + 1, 3: w - 1
        if act == 0:  # up
            self._state = (max(0, self._state[0] - 1), self._state[1])
        elif act == 1:  # right
            self._state = (self._state[0], min(self._width - 1, self._state[1] + 1))
        elif act == 2:  # down
            self._state = (min(self._height - 1, self._state[0] + 1), self._state[1])
        elif act == 3:  # left
            self._state = (self._state[0], max(0, self._state[1] - 1))
```

策略类

```
from collections import namedtuple


Pi = namedtuple('Pi', 'act, prob')


class Policies:
    def __init__(self, env: Env):
        self._actions = env.actions
        self._default_policy = [1 / env.action_space] * env.action_space
        self._policies = dict.fromkeys(env.states, Pi(self._actions, self._default_policy))

    def sample(self, state):
        if self._policies.get(state, None) is None:
            self._policies[state] = Pi(self._actions, self._default_policy)

        policy = self._policies[state]
        return np.random.choice(policy.act, p=policy.prob)

    def retrieve(self, state):
        return self._policies[state].prob

    def update(self, state, policy):
        self._policies[state] = self._policies[state]._replace(prob=policy)
```

策略迭代主函数

```
In [107]: set_width = 3
```

```
In [126]: import time

          env = MatrixEnv(width=set_width, height=set_width)
          policies = Policies(env)
          values = ValueTable(env)
          upper_bound = 1e-3
          stable = False

          start = time.time()
          while not stable:
              policy_eval(env, values, policies, upper_bound)
              stable = policy_improve(env, values, policies)
          end = time.time()

          print('\n[time consumption]: {} s'.format(end - start))

          done = False
          rewards = 0
          env.reset()
          step = 0

          while not done:
              act_index = policies.sample(env.state)
              _, done, reward, next_state = env.step(env.actions[act_index])
              rewards += sum(reward)
              step += 1

          print('Evaluation: [reward] {} [step] {}'.format(rewards, step))
```

值迭代主函数

```
env = MatrixEnv(width=set_width, height=set_width)
policies = Policies(env)
values = ValueTable(env)
upper_bound = 1e-3

start = time.time()
value_iter(env, values, upper_bound)
_ = policy_improve(env, values, policies)
end = time.time()

print('\n[time consumption] {}s'.format(end - start))
# print("===== Render =====")
env.reset()
done = False
rewards = 0
step = 0
while not done:
    act_index = policies.sample(env.state)
    _, done, reward, next_state = env.step(env.actions[act_index])
    rewards += sum(reward)
    step += 1

print('Evaluation: [reward] {} [step] {}'.format(rewards, step))
```