Reinforcement Learning：An Introduction (Second Edition)-中译本-学习笔记

前言：

　个人认为，在人工智能领域中强化学习是机器智能的钥匙。尤其是，在Alpha Go和Alpha Zero问世以后，让我更加坚信了这一观点，因此强化学习也会是我的近几年的学习重点。在齐老师的推荐下，找到了这本书《Reinforcement Learning：An Introduction (Second Edition)》，本来想看英文原版，但是在考虑到效率和自身能力等因素后，还是选择了今年9月份发行的中译本《强化学习（第2版）》。

　“思想总是走在行动前面，就好像闪电总是走在雷鸣之前。”

第1章：导论

　本书将以人工智能研究者或工程师的视角来探索学习机理。将讨论如何设计高效的机器来解决科学或经济领域的学习问题，并通过数学分析或计算机实验的方式来评估这些设计。这种探索的方法就称为“强化学习”。

1.1 强化学习

　强化学习就是学习“做什么（即如何把当前的情景映射成动作）才能使得数值化的收益信号最大化”。强化学习的两个最重要最显著的特征是**试错和延迟收益**。

- 试错：智能体必须通过自己不断的尝试去发现哪些动作会产生最丰厚的收益，这是一个试错的过程（就像人一样，不犯错永远也不会长进）。
- 延迟收益：智能体的动作往往影响的不仅仅是即时收益，也会影响下一个情景，从而影响随后的收益（可能和大家平时熟知的“蝴蝶效应”的原理类似）。

　　强化学习不依赖于每个样本所对应的正确行为（即标注），它的目的是通过一定的学习方法产生最大化的收益信号。因此它不同于机器学习的**有监督学习和无监督学习**，可以把它看做第三种机器学习范式。

　　如果不懂什么是有监督学习和无监督学习，这边建议在学习本书之前先学习一下机器学习的相关知识，推荐书籍周志华教授的西瓜书。

1.2 示例

- 国际象棋大师走每一步棋（动作）。这个选择是通过反复计算对手可能的策略和对特定局面位置及走棋动作（环境的状态）的直观判断做出的。
- 一个移动机器人决定它是进入一个新房间收集更多垃圾还是返回充电站充电（动作）。它的决定是基于当前电量以及它过去走到充电站的难易程度（环境的状态）。

　　以上两个例子都涉及明确的目标，智能体可以根据这一目标来判断它的进展。国际象棋选手知道他是否得胜，移动机器人知道它的电池何时耗尽。智能体可以利用其经验（reward）来改进性能。国际象棋选手改善他在估计落子位置时的选择，从而改进游戏策略，最后获胜；移动机器人通过修改它的选择，从而使得他在电量耗尽之间完成更多的工作，最后提高它的工作效率。

1.3 强化学习要素

除了智能体和环境之外，强化学习系统有四个核心要素：**策略、收益信号、价值函数**以及（可选的）对环境建立的**模型**。

- 策略定义了学习智能体在特定时间的行为方式，是环境状态到动作的映射，策略本身是可以决定行为的。
- 收益信号定义了强化学习问题中的目标。在每一步中，环境向强化学习智能体发送一个称为收益的标量数值，智能体的唯一目标是最大化长期总收益。因此收益信号是改变策略的主要基础。
- 价值函数表示了长远的角度看什么是好的，而收益信号则表明了在短时间内什么是好的。简单的说，一个状态的价值是一个智能体从这个状态开始，对将来累积的总收益的期望。

在制定和评估策略时，我们最关心的是价值。动作的选择是基于对价值的判断做出的。我们寻求能带来最高价值而不是最高收益的状态的动作，因为这些动作从长远来看会为我们带来最大的累积收益（换句话说，我们需要我们的智能体不能被眼前的蝇头小利诱惑，我们要的是最终的结果）。不幸的是，确实价值要比确定收益难的多，因此，价值评估方法是几乎所有强化学习算法中的最重要组成部分。

- 对环境建立的模型是一种对环境的反应模式的模拟，它具有对外部环境的行为进行推断的功能。例如，给定一个状态和动作，模型就可以预测外部环境的下一个状态和下一个收益。

当然，对于强化学习系统来说，也可以采用无模型的方法进行直接的试错。但是模型存在是必要以及有效的，现代强化学习已经从低级的、试错式的学习延展到了高级的、深思熟虑的规划。

1.4 局限性与适用范围

本书讨论的大多数强化学习方法建立在对价值函数的估计上。但这并不是解决强化学习问题的必由之路，比如进化算法。如果策略空间充分小，或者可以很好地结构化以找到好的策略，或者我们有充分的时间来搜索，那么进化算法是有效的。另外，进化算法在那些智能体不能精确感知环境状态的问题上具有优势。

1.5 扩展实例：井字棋（附源码）

井字棋的规则请自行百度

井字棋的代码（摘自GitHub）实现如下（逻辑清晰简洁，大家可自行下载并体会智能体从“什么都不会”到“你怎么也下不过”的成长过程）：

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 27 23:38:29 2019

@author: YUYU
"""

import numpy as np
import pickle

BOARD_ROWS = 3
BOARD_COLS = 3
BOARD_SIZE = BOARD_ROWS * BOARD_COLS


class State:
    def __init__(self):
        # the board is represented by an n * n array,
        # 1 represents a chessman of the player who moves first,
        # -1 represents a chessman of another player
        # 0 represents an empty position
        self.data = np.zeros((BOARD_ROWS, BOARD_COLS))
        self.winner = None
        self.hash_val = None
        self.end = None

    # compute the hash value for one state, it's unique
    def hash(self):
        if self.hash_val is None:
            self.hash_val = 0
            for i in np.nditer(self.data):
                self.hash_val = self.hash_val * 3 + i + 1
        return self.hash_val

    # check whether a player has won the game, or it's a tie
    def is_end(self):
        if self.end is not None:
            return self.end
        results = []
        # check row
        for i in range(BOARD_ROWS):
            results.append(np.sum(self.data[i, :]))
        # check columns
        for i in range(BOARD_COLS):
            results.append(np.sum(self.data[:, i]))

        # check diagonals
        trace = 0
        reverse_trace = 0
        for i in range(BOARD_ROWS):
            trace += self.data[i, i]
            reverse_trace += self.data[i, BOARD_ROWS - 1 - i]
        results.append(trace)
        results.append(reverse_trace)

        for result in results:
            if result == 3:
                self.winner = 1
                self.end = True
                return self.end
            if result == -3:
                self.winner = -1
                self.end = True
                return self.end

        # whether it's a tie
        sum_values = np.sum(np.abs(self.data))
        if sum_values == BOARD_SIZE:
            self.winner = 0
            self.end = True
            return self.end

        # game is still going on
        self.end = False
        return self.end

    # @symbol: 1 or -1
    # put chessman symbol in position (i, j)
    def next_state(self, i, j, symbol):
```

```python
        new_state = State()
        new_state.data = np.copy(self.data)
        new_state.data[i, j] = symbol
        return new_state

    # print the board
    def print_state(self):
        for i in range(BOARD_ROWS):
            print('-------------')
            out = '| '
            for j in range(BOARD_COLS):
                if self.data[i, j] == 1:
                    token = '*'
                elif self.data[i, j] == -1:
                    token = 'x'
                else:
                    token = '0'
                out += token + ' | '
            print(out)
        print('-------------')


def get_all_states_impl(current_state, current_symbol, all_states):
    for i in range(BOARD_ROWS):
        for j in range(BOARD_COLS):
            if current_state.data[i][j] == 0:
                new_state = current_state.next_state(i, j, current_symbol)
                new_hash = new_state.hash()
                if new_hash not in all_states:
                    is_end = new_state.is_end()
                    all_states[new_hash] = (new_state, is_end)
                    if not is_end:
                        get_all_states_impl(new_state, -current_symbol,
all_states)


def get_all_states():
    current_symbol = 1
    current_state = State()
    all_states = dict()
    all_states[current_state.hash()] = (current_state, current_state.is_end())
    get_all_states_impl(current_state, current_symbol, all_states)
    return all_states


# all possible board configurations
all_states = get_all_states()


class Judger:
    # @player1: the player who will move first, its chessman will be 1
    # @player2: another player with a chessman -1
    def __init__(self, player1, player2):
        self.p1 = player1
        self.p2 = player2
        self.current_player = None
        self.p1_symbol = 1
        self.p2_symbol = -1
        self.p1.set_symbol(self.p1_symbol)
        self.p2.set_symbol(self.p2_symbol)
        self.current_state = State()

    def reset(self):
        self.p1.reset()
        self.p2.reset()

    def alternate(self):
        while True:
            yield self.p1
            yield self.p2

    # @print_state: if True, print each board during the game
    def play(self, print_state=False):
        alternator = self.alternate()
        self.reset()
        current_state = State()
        self.p1.set_state(current_state)
        self.p2.set_state(current_state)
        if print_state:
            current_state.print_state()
        while True:
            player = next(alternator)
            i, j, symbol = player.act()
            next_state_hash = current_state.next_state(i, j, symbol).hash()
            current_state, is_end = all_states[next_state_hash]
            self.p1.set_state(current_state)
            self.p2.set_state(current_state)
            if print_state:
                current_state.print_state()
            if is_end:
                return current_state.winner


# AI player
class Player:
    # @step_size: the step size to update estimations
    # @epsilon: the probability to explore
    def __init__(self, step_size=0.1, epsilon=0.1):
        self.estimations = dict()
        self.step_size = step_size
        self.epsilon = epsilon
        self.states = []
        self.greedy = []
        self.symbol = 0

    def reset(self):
        self.states = []
        self.greedy = []

    def set_state(self, state):
        self.states.append(state)
        self.greedy.append(True)

    def set_symbol(self, symbol):
        self.symbol = symbol
        for hash_val in all_states:
            state, is_end = all_states[hash_val]
            if is_end:
                if state.winner == self.symbol:
                    self.estimations[hash_val] = 1.0
                elif state.winner == 0:
                    # we need to distinguish between a tie and a lose
                    self.estimations[hash_val] = 0.5
                else:
                    self.estimations[hash_val] = 0
            else:
                self.estimations[hash_val] = 0.5

    # update value estimation
    def backup(self):
        states = [state.hash() for state in self.states]

        for i in reversed(range(len(states) - 1)):
            state = states[i]
            td_error = self.greedy[i] * (
                self.estimations[states[i + 1]] - self.estimations[state]
            )
            self.estimations[state] += self.step_size * td_error

    # choose an action based on the state
    def act(self):
        state = self.states[-1]
        next_states = []
        next_positions = []
        for i in range(BOARD_ROWS):
            for j in range(BOARD_COLS):
                if state.data[i, j] == 0:
                    next_positions.append([i, j])
                    next_states.append(state.next_state(
                        i, j, self.symbol).hash())

        if np.random.rand() < self.epsilon:
            action = next_positions[np.random.randint(len(next_positions))]
            action.append(self.symbol)
            self.greedy[-1] = False
            return action

        values = []
        for hash_val, pos in zip(next_states, next_positions):
            values.append((self.estimations[hash_val], pos))
        # to select one of the actions of equal value at random due to Python's
sort is stable
        np.random.shuffle(values)
        values.sort(key=lambda x: x[0], reverse=True)
        action = values[0][1]
        action.append(self.symbol)
        return action

    def save_policy(self):
        with open('policy_%s.bin' % ('first' if self.symbol == 1 else 'second'),
'wb') as f:
            pickle.dump(self.estimations, f)

    def load_policy(self):
        with open('policy_%s.bin' % ('first' if self.symbol == 1 else 'second'),
'rb') as f:
            self.estimations = pickle.load(f)


# human interface
# input a number to put a chessman
# | q | w | e |
# | a | s | d |
# | z | x | c |
class HumanPlayer:
    def __init__(self, **kwargs):
        self.symbol = None
        self.keys = ['q', 'w', 'e', 'a', 's', 'd', 'z', 'x', 'c']
        self.state = None

    def reset(self):
        pass

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

    def set_symbol(self, symbol):
        self.symbol = symbol

    def act(self):
        self.state.print_state()
        key = input("Input your position:")
        data = self.keys.index(key)
        i = data // BOARD_COLS
        j = data % BOARD_COLS
        return i, j, self.symbol


def train(epochs, print_every_n=500):
    player1 = Player(epsilon=0.01)
    player2 = Player(epsilon=0.01)
    judger = Judger(player1, player2)
    player1_win = 0.0
    player2_win = 0.0
    for i in range(1, epochs + 1):
        winner = judger.play(print_state=False)
        if winner == 1:
            player1_win += 1
        if winner == -1:
            player2_win += 1
        if i % print_every_n == 0:
            print('Epoch %d, player 1 winrate: %.02f, player 2 winrate: %.02f' %
(i, player1_win / i, player2_win / i))
        player1.backup()
        player2.backup()
        judger.reset()
    player1.save_policy()
    player2.save_policy()


def compete(turns):
    player1 = Player(epsilon=0)
    player2 = Player(epsilon=0)
    judger = Judger(player1, player2)
    player1.load_policy()
    player2.load_policy()
    player1_win = 0.0
    player2_win = 0.0
    for _ in range(turns):
        winner = judger.play()
        if winner == 1:
            player1_win += 1
        if winner == -1:
            player2_win += 1
        judger.reset()
    print('%d turns, player 1 win %.02f, player 2 win %.02f' % (turns,
player1_win / turns, player2_win / turns))


# The game is a zero sum game. If both players are playing with an optimal
strategy, every game will end in a tie.
# So we test whether the AI can guarantee at least a tie if it goes second.
def play():
    while True:
        player1 = HumanPlayer()
        player2 = Player(epsilon=0)
        judger = Judger(player1, player2)
        player2.load_policy()
        winner = judger.play()
        if winner == player2.symbol:
            print("You lose!")
        elif winner == player1.symbol:
            print("You win!")
        else:
            print("It is a tie!")


if __name__ == '__main__':
    train(int(1e5))
    compete(int(1e3))
    play()
```

1.6 本章小结

强化学习是一种对目标导向的学习与决策问题进行理解和自动化处理的计算方法。它是第一个严格意义上的解决从环境互动中学习以达到长期目标这一计算问题的领域。

价值与价值函数的概念是我们本书中探讨的大多数强化学习方法的重要特征。