南開大學
Nankai University

人工智能学院

强化学习实验报告

实验一　井字棋

姓名：石若川

学号：2111381

专业：智能科学与技术

2024 年 3 月 3 日

1 实验内容

阅读井字棋原代码 tic_tac_toe.py，修改为 4×4 的棋盘大小，并利用不同迭代次数的策略进行对弈，展示对弈结果。

2 代码理解

以下是代码中的主要的类与函数：

- State 类：表示井字棋游戏的状态，包括棋盘数据、胜者、哈希值和游戏是否结束等信息。其中 hash 函数将棋盘的状态利用三进制表示为一个整数。is_end 函数判断当前棋盘中是否在行、列、对角线、反对角线上出现了四子相连的情况。
- get_all_states_impl 和 get_all_states 函数：通过递归的方式，获取所有可能的游戏状态，并存储在“all_states ”变量中。
- Judger 类：判定游戏结果，负责玩家轮流进行游戏。play 函数中，通过不断调用玩家的 act 函数来进行游戏，直到游戏结束。
- Player 类：代表井字棋游戏中的玩家，使用 Q-learning 算法来学习最优策略。backup 函数使用 Q-learning 算法进行值函数的更新。act 函数根据当前状态选择动作。
- train 函数：训练两个玩家进行多次对战，通过 Q-learning 来优化玩家的策略。
- compete 函数：两个已经训练好的玩家进行对战，并输出结果。

3 代码修改

由于每次运行代码需要计算所有的状态，当棋盘大小增大为 4×4 后，状态空间急剧增大。为了节约计算时间，首先考虑将 all_states 变量保存为 pkl 文件。在 3×3 的棋盘大小上，该方法可行。但是由于 4×4 的状态空间过大，需要保存的文件过大，运行代码会出现 Memory Error 的错误。因此，使用 Jupyter Notebook 运行代码。

首先在棋盘大小改为 4×4：

```
BOARD_ROWS = 4
BOARD_COLS = 4
BOARD_SIZE = BOARD_ROWS * BOARD_COLS
```

修改 State 类中，将用于判断棋局胜利的 is_end 函数。每行列、对角线和反对角线中出现四子连线，则对应玩家胜利：

```
for result in results:
    if result == 4:
        self.winner = 1
        self.end = True
        return self.end
    if result == -4:
```

```
        self.winner = -1
        self.end = True
        return self.end
```

修改 Player 类的构造函数，增加一个成员变量 epochs，用来表示 Player 类实例化时的迭代次数大小：

```
class Player:
    def __init__(self, step_size=0.1, epsilon=0.1, epochs=int(1e5)):
        self.estimations = dict()
        self.step_size = step_size
        self.epsilon = epsilon
        self.states = []
        self.greedy = []
        self.symbol = 0
        self.epochs = epochs
```

修改 compete 函数，使其可以传入两个用来表示对弈双方迭代次数的形参 epochs1 和 epochs2：

```
def compete(turns, epochs1, epochs2):
    player1 = Player(epsilon=0, epochs=epochs1)
    player2 = Player(epsilon=0, epochs=epochs2)
    judger = Judger(player1, player2)
    player1.load_policy()
    player2.load_policy()
    player1_win = 0.0
    player2_win = 0.0
    for i in range(turns):
        print("------------- %s -------------" %("TURN "+str(i+1)))
        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, tie %.02f'
          % (turns, player1_win / turns, player2_win / turns, (turns - player1_win -
              player2_win) / turns))
```

最后，由于不需要人参与对弈，所以删去 HumanPlayer 类和 play 函数。

4　实验结果

实验中分别训练了 1000 轮、10000 轮和 100000 轮迭代次数下的策略并保存。部分的对弈过程如图4.1和4.2所示，完整的运行结果见附件。

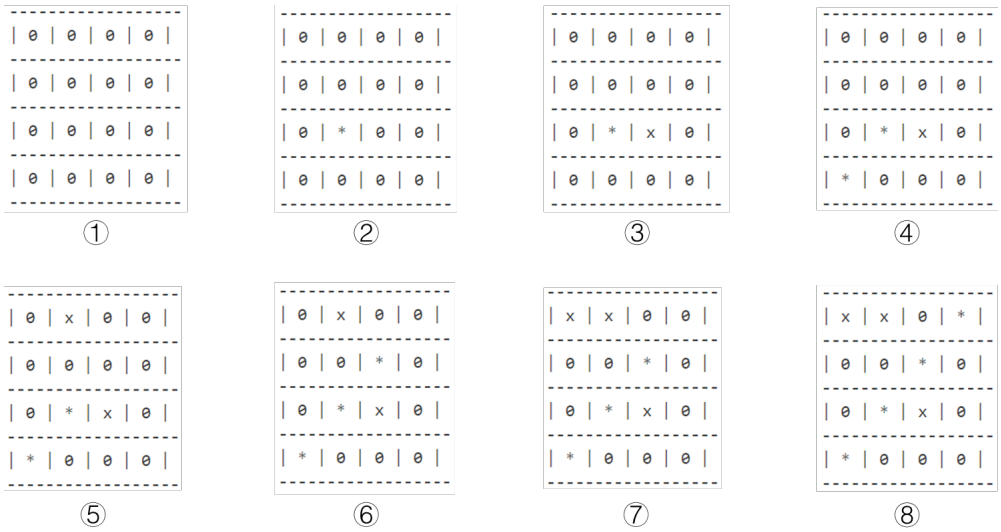

图 4.1: 10^5 vs 10^3 的一次对弈过程

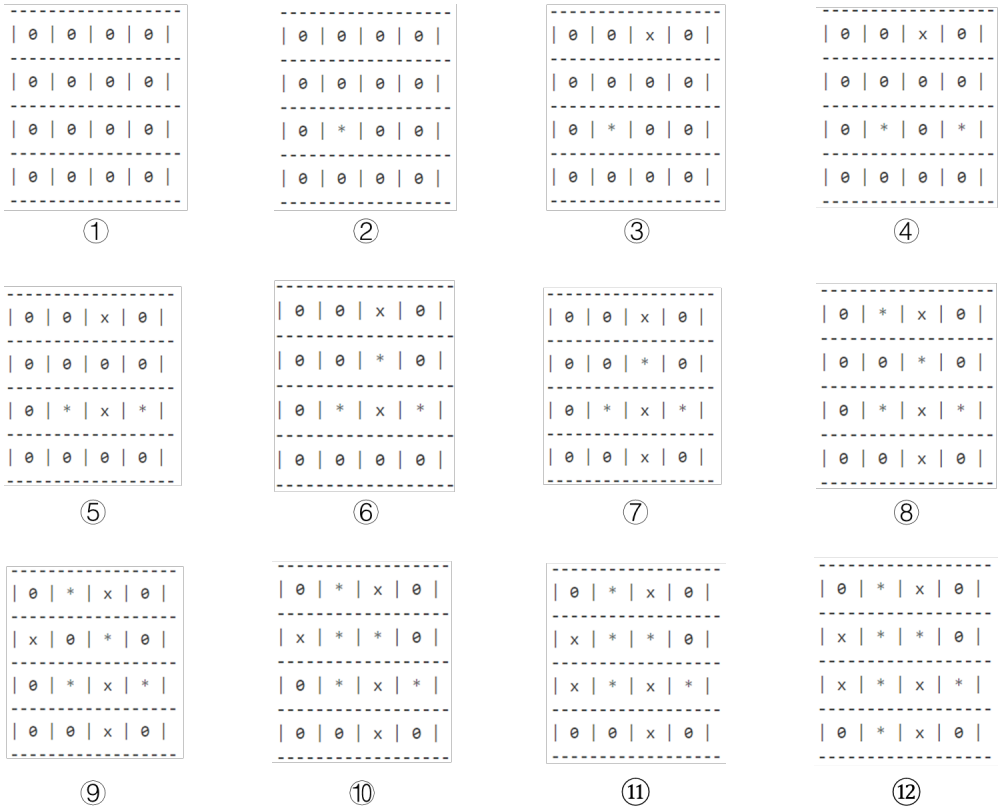

图 4.2: 10^5 vs 10^4 的一次对弈过程

将三个策略分先后手进行两两对弈，每两个策略之间共进行十次对弈，对弈情况如下表所示。

表 1: 各策略间的对弈结果

	10^3 vs 10^4	10^4 vs 10^3	10^3 vs 10^5	10^5 vs 10^3	10^4 vs 10^5	10^5 vs 10^4
先手胜	0.5	0.6	0.4	0.8	0.4	0.6
后手胜	0.3	0.3	0.3	0.1	0.1	0.2
平局	0.2	0.1	0.3	0.1	0.5	0.2

5 总结与分析

根据表1可以得到以下结论：

- 随着迭代次数增加，策略的获胜概率增加。这说明随迭代次数增加，可以训练得到更好的对弈策略。增加迭代次数意味着智能体有更多的机会学习并优化其策略。然而，这并不意味着能够无限制地提高性能，而是只能逼近最优策略。
- 当对弈策略不变，交换先后手顺序时，原先后手的一方，获胜概率增加。这说明先手具有一定先发优势。如果两名玩家都采用最佳的策略，那么如果双方都不犯错误，游戏将以平局结束。但在实际游戏中，如果双方没有采用最佳策略或者一方犯了错误，先发方可能会利用这些机会取得胜利。