# AI Project2

韩金汝 521021910982
王嘉睿 521021910937
王瑶瑶 521021910438

2024 年 1 月 14 日

# 目录

# 1 项目简介

## 1.1 棋型介绍

在五子棋中，当获胜条件为五子一线时，最常见的棋形有以下几种情况：

- **连五**：五颗同色棋子连在一起。
- **活四**：有两个连五点（即有两个点可以形成五），活四出现的时候，如果对方单纯过来防守的话，是已经无法阻止自己连五了。
- **冲四**：有一个连五点，对方只要跟着防守在那个唯一的连五点上，冲四就没法形成连五。
- **活三**：可以形成活四的三。活三棋型是我们进攻中最常见的一种，因为活三之后，如果对方不以理会，将可以下一手将活三变成活四，而我们知道活四是已经无法单纯防守住了。所以，当我们面对活三的时候，需要非常谨慎对待。在自己没有更好的进攻手段的情况下，需要对其进行防守，以防止其形成必胜的活四棋型。
- **眠三**：只能够形成冲四的三。眠三的棋型与活三的棋型相比，危险系数下降，因为眠三棋型即使不去防守，下一手它也只能形成冲四，而对于单纯的冲四棋型，我们可以通过堵住仅剩的连五点来防止获胜。
- **活二**：能够形成活三的二。活二棋型暂时没有威胁，因为他下一手棋才能形成活三，等形成活三，我们再防守也不迟。但其实活二棋型是非常重要的，尤其是在开局阶段，我们形成较多的活二棋型的话，当我们将活二变成活三时，才能够令自己的活三数量大大增加，让对手防不胜防。
- **眠二**：能够形成眠三的二。

了解五子棋的棋型在 alpha-beta 剪枝算法中对于非终止状态的评估有非常重要的作用，在2.1.2可以看到设计估值函数时主要结合棋型以及当前落子玩家的颜色计算当前状态对搜索根节点玩家的价值。

# 2 Alpha-beta Pruning

我们组 Alpha-beta 剪枝方法的实现在 alphabeta 目录下。

## 2.1 算法原理

在该方法中，我们根据当前的棋盘状态以及当前玩家，在一定深度限制 `depth_max` 以及一定分支数限制 k 内，使用极大极小算法搜索可能的棋盘状态，在搜索过程中应

用 Alpha-beta 剪枝进行优化，并根据棋盘状态以及玩家颜色使用评估函数计算当前局面的价值，寻找对当前玩家价值最高的 action 作为下一步的落子位置。

对于每一次落棋，基本步骤为：

1. 初始化 $\alpha, \beta$, best_value, best_move
2. 判断棋盘状态：如果棋盘上棋子的数量为 0 或 1，分别调用 `AlphaBetaSearch.first_move` 或 `AlphaBetaSearch.second_move` 来进行开局阶段的预设移动。这会在 2.1.3 节中详细说明。
3. 生成可能的移动：调用 `self.generate_k_moves` 生成当前局面下可能的移动。该函数将遍历当前棋局的所有可能移动 move，并使用评估函数评估每个棋局的分数，返回前分数最高的前 k 个移动的序列。
4. 迭代搜索：对于每一个可能的移动，生成新的棋盘状态，并进行递归搜索。调用 `self.minimax` 函数进行极小极大搜索同时应用 Alpha-beta 剪枝排除不必要的搜索。此外，使用 depth_max 限制递归搜索的深度，当发现取得胜利或者达到了最大深度限制后，结束递归，得到当前移动的得分 score。
5. 更新最佳移动和评分：如果 score 比当前最佳值更好，根据当前颜色 color 更新 best_value 和 best_move。
6. 返回最佳移动：选取最佳移动 best_move 作为本次落棋的策略。

### 2.1.1 剪枝原理

原始的 Min-Max 算法蕴含“穷举”的思想，即遍历能访问到的所有可能情况。由于本场景中棋盘大小为 $15 \times 15$，这样穷举会极大地影响搜索效率——由于每个状态都可能有超过 200 个子节点，因此时间复杂度会随搜索层数指数级上升。

由此引入 Alpha-beta 剪枝算法，通过“剪去”无效分支优化搜索效率。

该算法基于极小极大搜索算法，同时维护两个额外的值，即 alpha 和 beta。

- alpha：表示 Max 层当前找到的最好分数，即最大值。
- beta：表示 Min 层当前找到的最好分数，即最小值。

**Min-Max 搜索算法** 简述如下。

首先构建一颗博弈树，博弈树的节点层次交替为 Max 和 Min 层，Max 层次的节点表示轮到当前玩家进行决策的状态，而 Min 层次的节点表示对手的决策。

对于每一层的节点，需要获取其对应的分数 v，如果是叶节点则直接返回，否则需要遍历其所有子节点，取最大或最小的值 (取决于当前是 Max 还是 Min 节点)。

考虑 Min-Max-Min 的结构，对于第一层 Min，在遍历若干 Max 子节点获得目前最好分数 v 后，对于剩下的 Max 子节点，如果该 Max 节点的 Min 子节点有一个值比 v 大，则说明当前 Max 节点的值必然不可能是更好的 v，无需获取该 Max 节点的最终值，由此诞生了剪枝的方案——在对当前节点遍历其子节点求值时，时刻维护目前搜索

到的最好值 alpha 或 beta，然后遍历子节点时传递给子节点，再由子节点判断是否可以剪枝。

**剪枝条件**

- 对于 Max 层，依次遍历 Min 子树，如果发现有一颗子树的值 v 比父节点传下来的 beta 大时，则无需继续搜索剩余子树，因为其父节点取 Min{所有 max 子树}，当前 Max 层的值$\geq v > \alpha$ 不可能成为其父节点的更优值。
- 对于 Min 层，依次遍历 Max 子树，如果发现有一颗子树的值 v 比父节点传下来的 alpha 小时，则无需继续搜索剩余子树，因为其父节点取 Max{所有 min 子树}，当前 Min 层的值$\leq v < \beta$ 不可能成为其父节点的更优值。

**$\alpha$ 和 $\beta$ 的维护**

- 对于 Max 层，只能更新 alpha 值，即 alpha 取 max(v,alpha)
- 对于 Min 层，只能更新 beta 值，即 beta 取 min(v,beta)

以上算法描述的伪代码如下：

### 2.1.2 评估函数

在对当前棋盘状态进行评估时需根据当前棋盘已有状态以及当前状态下即将落子的玩家颜色来计算对某一玩家的价值。在具体的计算时，我们分为以下四步：

1. `evaluation_state` 函数：根据当前落子玩家的颜色，分别计算当前棋盘上白子黑子两种情况下对当前玩家的价值并求和。

2. `evaluation_color` 函数：遍历棋盘棋盘的行、列以及对角线，对于每一条 line 计算对于当前落子玩家、当前评估的棋子颜色在该条线上所能带来的价值。

3. `evaluation_line` 函数：统计当前线上连续棋子数量、两边的堵塞情况、当前是否为该子落子、以及连续棋子中是否有空位来改变状态以及在不同情况下调用计算分数。对应到1.1可以理解为此函数在计算棋型。

4. `calc` 函数：根据当前线上连续棋子数量、两边的堵塞情况、当前是否为该子落子、以及连续棋子中是否有空位来计算分数。即为在给定棋型下根据当前落子玩家颜色计算分数。

其中 `calc` 函数的计算规则有如下几条，其中前三条为确定情况，直接返回分数：

- 两侧均被堵住且连子数量小于 5: 直接返回 0
- 连五：返回 100000
- 活四和冲四：返回 8000

- 其他情况根据连续棋子的数目情况，在连子数量对应的基准评分上根据两侧的堵塞状态、是否有空位以及是否为当前玩家乘对应的折扣因子。其中连子的数量对应的分值如表1所示，其他折扣因子对应的系数如表2所示

| **连子数目** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| 对应分数 | 2 | 5 | 1000 | 10000 |

表 1: 连子数目对应的 consec_score 评分

| **连子数目** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| 只有一端堵塞 | 0.5 | 0.6 | 0.01 | 0.25 |
| 当前落子玩家非估值玩家 | 1 | 1 | 0.2 | 0.15 |
| 形成的连子中是否有空位 | 1 | 1.2 | 0.9 | 0.4 |

表 2: 根据连子数目的不同条件下的折扣因子

### 2.1.3 第一、二步的特殊处理

当棋局刚开始的时候，情况较为特殊，通过简单处理省略搜索过程，提高效率。

- 当棋盘中没有任何棋子时，直接找到棋盘中心的位置，然后从当前位置的上、中、下三个相邻位置中随机选择一个位置作为第一步的落子位置。
- 当棋盘中只有一枚棋子时，下一步落子位置只有直止和斜止两种，我们认为斜止更优，故只考虑该棋子左下、左上、右下、右上四个位置落子。考虑到要向空间更开阔的方位落子，我们以棋盘中心位置为原点将棋盘划分为左下、左上、右下、右上四个象限，判断该枚棋子位于棋盘的哪个象限，然后选择与所在象限相反的方位落子。

具体的代码实现如下：

```
1     @staticmethod
2     def first_move(board):
3         # 落子到中心位置
4         x = board.shape[0] // 2
5         return tuple(np.random.choice((x - 1, x, x + 1), 2))
6
7     @staticmethod
8     def second_move(board, last_drop):
9         # 斜止落子
10        i, j = last_drop
11        size = board.shape[0]
```

```
        i2 = i <= size // 2 and 1 or -1
        j2 = j <= size // 2 and 1 or -1
        return (i + i2, j + j2)
```

## 2.2 性能优化

### 2.2.1 引入棋盘状态哈希

在搜索过程中，我们意识到一些相同的棋盘状态如果不进行存储的话会在每次搜索到时重复对该状态进行搜索，因此为了提高速度，我们引入了 Zobrist 哈希算法。

Zobrist 哈希通过一种特殊的置换表, 也就是对棋盘上每一位置的各个可能状态赋予一个编码索引值, 来实现在极低冲突率的前提下在一个整型数据上对棋盘进行编码。

其编码的步骤说明如下：

1. 将棋盘分为最小单位, 求出每个单位上不同状态数。

   对于 15*15 的五子棋盘，最小单位即棋盘的交叉点，共有 225 个，每个交叉点对应三个状态：白棋、黑棋、空。

2. 为每个单位上的每种状态生成一个一定范围内 (取 32 位整数) 随机数。

3. 对于特定的棋局, 将每个单位上的状态对应的随机数作异或运算, 所得即为该棋局的哈希值。

   用 Zobrist 哈希为棋局状态编码有以下两个优点:

- 当随机数的范围足够大时, 不同的棋局产生哈希冲突的概率非常小, 在实际应用中通常可以忽略。
- 在棋局进行过程中, 不必每次重新开始计算棋局的哈希值, 只需计算棋局状态发生改变的部分，即: 棋局原始哈希值 $\bigoplus$ 落子处原始随机数 $\bigoplus$ 落子后该处随机数

由于五子棋只能落棋不能吃棋，因此每个单位的状态只会从空转变为白棋或黑棋，不会有转变成空的情况，因此可以进一步简化。默认空对应随机数为 0，仅对白棋和黑棋两种状态生成随机数，得到 (15，15，2) 大小的哈希表。每次棋局发生改变时只需将当前棋局的哈希值与落子处新的状态随机数作异或即可，如下代码所示：

```
hash_value ^= self.hash_table[i, j, piece_index]
```

Listing 1: Hash Calculate

# 3 Reinforcement Learning

## 3.1 DQN

DQN（Deep Q-Network）是一种基于深度学习和强化学习的算法，用于解决马尔可夫决策过程（MDP）中的控制问题。它由 Google DeepMind 团队在 2013 年提出，并在后续的研究中得到了广泛应用。

DQN 的核心思想是通过使用神经网络近似值函数，实现对状态-动作值函数（Q 函数）的估计和优化。DQN 使用深度卷积神经网络来处理具有高维输入的问题，比如本项目中的棋盘状态。

### 3.1.1 DQN 原理

1. 经验回放（Experience Replay）：DQN 使用经验回放缓存器来存储智能体在环境中的经验。这些经验包括状态、动作、奖励和下一个状态。在训练过程中，DQN 从回放缓存器中随机选择一批经验样本进行训练，这有助于减少样本间的相关性，提高训练效率和稳定性。

2. 目标网络（Target Network）：为了增加训练的稳定性，DQN 使用两个神经网络：一个是用于选择动作的主网络，另一个是用于估计目标值的目标网络。目标网络的参数定期更新为主网络的参数，这样可以减少目标值估计的波动性。

3. Q-Learning 更新：DQN 使用 Q-Learning 算法进行更新。通过选择使得 Q 值最大化的动作来更新 Q 网络的参数。这个过程可以通过最小化 Q 网络输出和目标值之间的均方误差来实现。

$$\omega^* = \arg\min_{\omega} \frac{1}{2N} \sum_{i=1}^{N} \left[ Q_\omega(s_i, a_i) - \left( r_i + \gamma \max_{a'} Q_\omega(s_i', a') \right) \right]^2$$

图 1: Q 网络损失函数

### 3.1.2 如何运行项目代码

**使用项目代码进行训练、AI 自弈**

- 更改 `DQN.py` 文件中的 `IS_TRAIN` 可以设置当前的运行模式为训练还是测试。测试模式为 AI 自弈。
- `CHECKPOINT` 参数表示当前测试使用的是训练了 `CHECKPOINT` 轮的网络参数存档进行测试。存档均在 checkpoints 文件夹中。
- 运行 `DQN.py`。输出结果为训练进度、AI 自弈结果。

**使用项目代码进行人机对弈**

- `CHECKPOINT` 参数表示当前使用的是训练了 `CHECKPOINT` 轮的网络参数存档。
- 运行 `main.py` 进行人机对弈。

### 3.1.3 实现细节

**网络结构**

1. 输入层：输入为棋盘状态信息，形状为 (None,$SIZE^2$)，其中 SIZE 为棋盘的宽度。
2. 卷积层：使用 5x5 大小的卷积核，16 个输出通道。激活函数为 ReLU。权重矩阵 W1 的形状为 [5, 5, 1, 16]，偏置向量 b1 的形状为 [16]。
3. 最大池化层：使用 3x3 大小的池化窗口进行最大池化操作。输出形状为 [5, 5, 16]。
4. 全连接层：将最大池化层的输出展平为一维向量，形状为 [5x5x16]。然后与表示当前玩家的 turn 向量进行拼接，形状为 [5x5x16+1]。权重矩阵 W2 的形状为 [5x5x16+1, 225]，偏置向量 b2 的形状为 [1, 225]。
5. 输出层：输出为 Q 值的估计值，形状为 (None,225)，其中 None 表示批量大小。

**奖励函数** .

本项目的 DQN 使用的奖励函数不同于常规奖励函数。

常规的奖励函数为：（数值不一定相同）

- 非法落子：-10000
- 常规落子：0
- 本局和棋：50
- 本局胜利：100
- 本局失败：-100

以上常规的 Q-learning 奖励函数简单易懂，给予胜利正奖励、失败负奖励。但是其问题在于训练时只有到达一局的结束才能获得奖励，而棋局中的落子因为距离结束较远，受到折扣因子的影响，几乎没有任何奖励的指导。所以训练效果较差，在 50000 轮训练之后基本还是随机落子。

本项目的奖励函数：

- 非法落子：-1000000
- 常规落子：score(p)+score(-p)*0.9，p 为当前玩家
- 本局和棋：100000
- 本局胜利：1000000
- 本局失败：-1000000

其中，score(p) 表示对于玩家 p 来说当前位置的分数：

$$score(p) = 10^{\left(\sum_{d=1}^{4}(C_d+1)\right)}$$

d 表示方向，$C_d$ 表示方向 d 上当前位置相邻有几个连续的己方棋子

该奖励函数是本项目的一个创新点。它能够对于一局中间的落子给出合适的奖励，使得网络能够更快地学习到五子棋的规律，而不是只有一局结束时才能获取到奖励。另外，该奖励函数能够将对手的需求考虑在内，使得 AI 能够具有一定的防守倾向。通过控制参数（目前为 0.9）能调整 AI 的攻防策略。

## 3.2 AlphaZero

使用 DQN 进行尝试后，在训练了 40000 代后与人类对弈的效果不佳，因此尝试学习使用 alphazero 算法。

### 3.2.1 蒙特卡洛树搜索

蒙特卡洛树搜索主要可以分为四步，即选择、拓展、模拟、回溯 (反向传播)。初始阶段，蒙特卡洛树只有一个节点，代表我们需要进行决策的局面。接着将构建搜索树，该搜索树的每个节点 i 包含三个基本信息：代表的局面，被访问的次数 $N_i$，累计评分 $Q_i$。

**选择** 在选择阶段，需要从根节点出发向下在搜索树的底端选择出一个 “最急迫” 需要被拓展的节点。依次检查每个节点，其可能情况有三种：

- 该节点所有可行动作都已经被拓展过：则利用 UCB 公式计算其所有子节点的 UCB 值，选择值最大的子节点继续检查，反复向下迭代。
- 该节点有可行动作还未被拓展过：选定该节点作为拓展节点 N，同时找出其一个未拓展的动作 A，结束选择阶段。
- 这个节点游戏结束 (已经连成五子的五子棋局面)：直接跳转到**回溯**阶段。

标准 UCB(Upper Confidence Bound，上限置信区间) 公式如下：

$$S = UCB(i) = \frac{W(i)}{N(i)} + c \cdot \sqrt{\frac{\ln(N(p))}{N(i)}} \tag{1}$$

其中：$S$ 为对应分数，$W(i)$ 是节点 $i$ 胜利次数，$N(i)$ 是节点 $i$ 的访问次数，$N(p)$ 是父节点 $p$ 的访问次数，$c$ 是设置的超参数，用于控制已知路径与未知路径平衡的常数。

在与 AlphaZero 结合后，实际代码中采用的公式为：

$$S = Q + C \cdot P \cdot \frac{\sqrt{N_p}}{1 + N_i} \tag{2}$$

其中，Q 为节点本身历史估值的平均，更新公式在**回溯**中介绍，P 为策略价值网络的预测结果，后文进一步介绍。该公式结合了节点的先验概率和实际的游戏结果，允许算法在已知良好动作和潜在未探索动作之间做出平衡，因此可以取得良好的结果。

**拓展** 选择阶段结束后，我们找到了一个拓展节点 N，以及其对应的未拓展动作 A，在搜索树中创建一个新节点 Nn 表示 N 对应局面执行完 A 操作后得到新局面。

**模拟** 拓展好新节点后，需要获取 Nn 对应的初始评分，通过模拟游戏的方式获得。即从 Nn 开始让游戏随机进行直到结束，以该结局的胜利、失败、平局对应 v = 1、-1、0。

**回溯** 模拟阶段结束后，根据模拟的结局更新根节点到 N 的路径上所有节点的累计评分 Q 和访问次数 N。对于从选择阶段直接跳转到回溯阶段的节点，通过选择阶段发现的游戏结局来更新。更新公式如下：

$$Q = \frac{N_i \cdot Q + v}{N_i'}, \quad N_i' = N_i + 1 \tag{3}$$

### 3.2.2 策略价值网络

策略价值网络即为在给定的局面下，使用神经网络计算返回当前局面下每一个可行动作的概率以及当前局面的评分。

**局面描述方式** 与原始的 alphazero 不同，我们使用了 4x15x15 的二值特征平面来描述当前棋盘状态并并作为神经网络的输入。第一、二个二值平面分别表述当前玩家的棋子状态和对手玩家的棋子状态（有子为 1，无子为 2）；第三个平面表示上一步落子的位置，即落子位置为 1，其余部分全 0；第四个平面表示当前玩家，若为全 1 则表示为当前问价，全 0 则相反。

**网络结构** 在查阅资料以及搜索 alphazero 具体代码实现后，我们参考 (1) 后，将网络分为特征提取部分、策略网络部分以及价值网络。其中特征提取网络为 1x1x64 的卷积层以及 19 个 residual blocks；策略网络由 1x1x2 的卷积层连接 2x225 的全连接层组成，返回棋盘上 225 个位置的概率；价值网络由 1x1x1 的卷积层、1x256 全连接层以及 256x1 全连接层得到当前状态的价值。激活函数使用 relu。

损失函数的定义为：

$$\mathcal{L} = (z - v)^2 - \pi^T \log P + c\theta^2$$

其中 $z$ 表示自我博弈时的胜者，$v$ 为当前局面的评分，$p$ 为为当前棋盘下不同位置对应的概率，$\pi$ 为自我对弈时蒙特卡洛输出的概率，即优化的目标为使策略网络预测的概率更接近蒙特卡洛得到的概率，让价值网络预测的局面评分更接近自我对弈时的胜方。

#### 3.2.3 AlphaZero 自我对弈

alphazero 可进行自我对弈从而训练网络。

**对弈数据库** 在自我对弈时，我们在迭代过程中收集更新训练数据。具体而言，按照回合交替，输入当前的棋盘参数，统计蒙特卡洛树中每个节点的访问次数并归一化计算概率，根据该概率随机选取下一步动作。在获得动作后，收集原先的棋盘状态、每个动作对应的概率；在游戏结束后将此局收集的状态以及动作对应的概率作为一份数据加入数据库；同时，数据库指定大小，即当数据量达到一定值时覆盖旧的数据。

# 4 结果分析

## 4.1 Alpha-beta Pruning 算法

在第 2 节中介绍过我们通过 depth_max 和 k 分别限制搜索的深度及分支数，通过调整这些参数进行下棋时间的探索。由于不同棋局需要的搜索时间会有波动，因此统一考虑平均时间。如需进行更严谨的实验，还需要对棋局严格控制。

#### 4.1.1 搜索深度的探索

| **max_depth 取值** | **2** | **3** | **4** |
|---|---|---|---|
| 一步棋平均时间/s | 0.613 | 73.826 | 213.561 |

表 3: 不同深度限制 max_depth 对下棋时间的影响

测试时控制 k=5，使用最优的哈希算法，得到结果如表 3所示。

基本可以看到每一步下棋时间随着 max_depth 的增加非常快速地成指数增长，极大地限制了算法的性能。

#### 4.1.2 生成状态数目的探索

| **k 取值** | **10** | **20** | **40** |
|---|---|---|---|
| 一步棋平均时间/s | 1.17 | 2.27 | 4.25 |

表 4: 不同分支数目 k 对下棋时间的影响

测试时控制最大搜索深度为 2，使用最优的哈希算法，改变每个节点生成可能移动的数目 k，测试时间得到如 4所示的结果。

可以看到，平均每步棋的时间与 k 取值差不多成线性关系。

### 4.1.3 使用哈希优化的探索

| 哈希方法 | 遍历计算哈希值 | 根据改变值计算哈希值 |
|---|---|---|
| 一步棋平均时间/s | 213.561 | 323.32 |

表 5: 不同哈希实现对下棋时间的影响

测试时控制最大搜索深度为 4，k=5，测试时间得到如 5所示。

鉴于棋局的不可控性，因此实验结果并不能严格保证严谨性，无法精确分析结果产生的原因，但是可以作为性能比较的依据，很明显可以看出每次不重新计算当前棋局的哈希值，而是通过计算改变值的方法对于下棋所需时间的降低有一定效果。

### 4.1.4 结果分析

**算法复杂度分析** 下面对时间和空间复杂度分别分析。Min-Max 算法执行深度有限搜索，如果考虑树的最大深度为 m, 每个点的可移动方法 move 有 k 个，则其时间复杂度为 $O(k^m)$, 对于一次性生成所有 move 的算法，空间复杂度为 $O(km)$，对于每次生成一个 move 的算法空间复杂度为 $O(m)$。

引入 Alpha-beta 剪枝算法后，最好的时间复杂度可以达到 $O(k^{\frac{m}{2}})$。

据此分析 4.1.2 得到的结果：

对于不同 k 的测试，由于 m=2, 因此算法的时间复杂度为 $k^{\frac{m}{2}} = O(k)$，与实验结果吻合。

**实验结果展示** 展示两张人机对弈的结果如下所示。

## 4.2 强化学习

### 4.2.1 DQN

**奖励函数的比较** 使用两种奖励函数训练 1000 轮后，进行自弈测试结果比较：

根据以上自弈结果来看，使用常规奖励函数时，AI 落子比较随机，不能掌握到如何在五子棋中取胜。而在使用了本项目中设计的奖励函数后，仅 1000 轮训练后，AI 的落子就明显变得更加集中了，并且也能够较快地取胜。

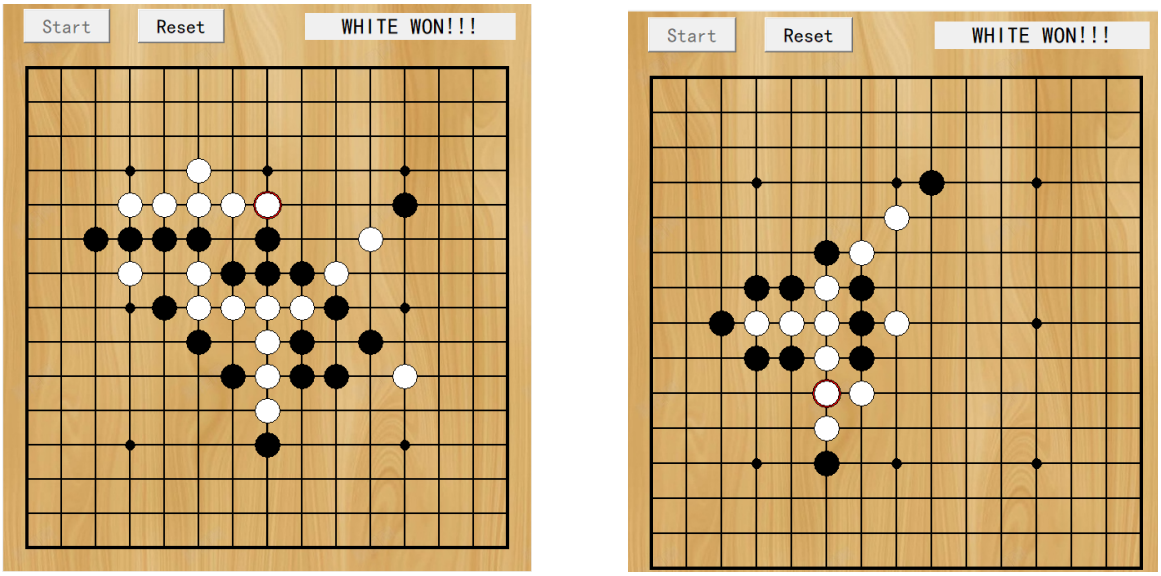


图 2: 人先手（左）和 AI 先手（右）的剪枝算法效果

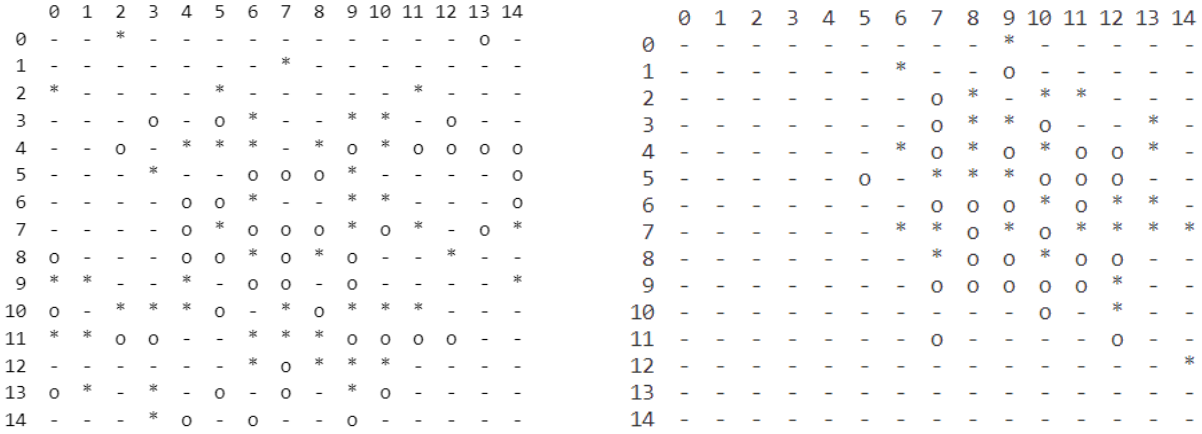

图 3: 常规（左）和本项目（右）奖励函数-自弈结果

**训练结果** 根据本项目的奖励函数来训练 DQN，训练了 40000 轮后，自弈结果如下图4所示。

可以发现，在 40000 轮训练后，AI 已经能够大概学习到如何来取胜了，并且落子相比之前更加集中。但是此时 AI 仍然无法在人机对弈中取胜。

通过比较发现，训练轮次增长时 AI 的表现也不断优化。所以经过分析，AI 在人机对弈中表现不足的原因应该是训练轮数仍然不够。15*15 的五子棋状态空间过于巨大，40000 轮的训练不能很好地覆盖五子棋的各种情况，导致 DQN 还没有很好地学习到奖励函数以及胜利方式的信息。然而因为 GPU 性能限制以及时间限制，没有训练更多轮次。同时参考了其他使用强化学习的五子棋任务，若想要在人机对弈中取胜需要进行更大量级的训练代数。因此我们猜测在训练轮次足够的情况下，DQN 应该能够学习到奖

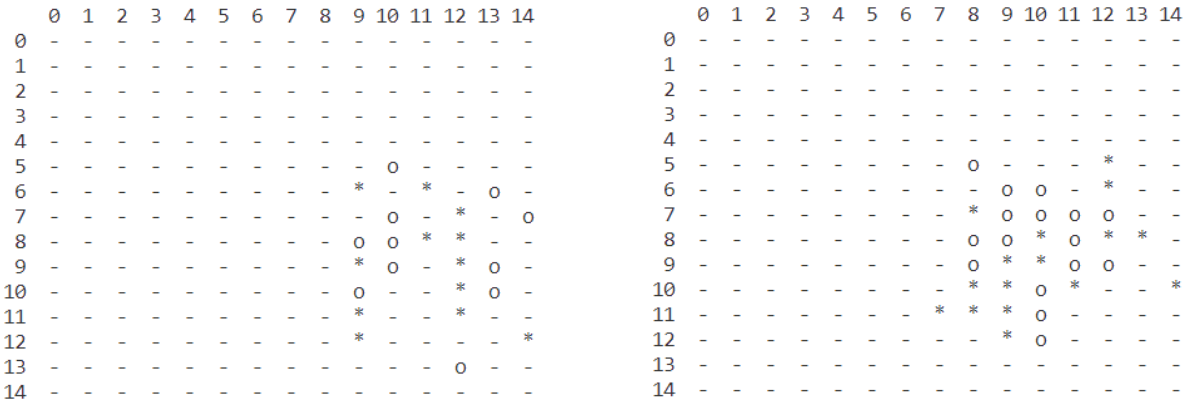

图 4: 40000 轮训练后自弈结果

励、胜利信息，并且至少能够表现出简单的五子棋攻防策略。

### 4.2.2 AlphaZero

同时我们还探索了 AlphaZero 的实现。在具体的实现过程中，我们参考 (1) 的代码并学习根据学习 AlphaZero(2) 的原理并实现蒙特卡洛搜索。但由于先前主要在探索 DQN 算法，以及源代码中网络架构为 tensorflow，我们只初步实现 pytorch 版本的网络但还未训练。因此我们提交的 AlphaZero 部分直接使用了其 tensorflow 的网络以及参数并进行了测试。测试的效果如下图5所示。

在没有 GPU 加速的情况下，单步运行时间较长约为 10s，但已经能与人对弈数局。根据该代码原作者表示，该网络对应的参数在 2 张 1080ti 的 GPU 训练了 100000 局花费 800h，这也反映了该任务的强化学习算法如需要达到人机对弈的水平需要较长的训练时间以及较大的数据量。

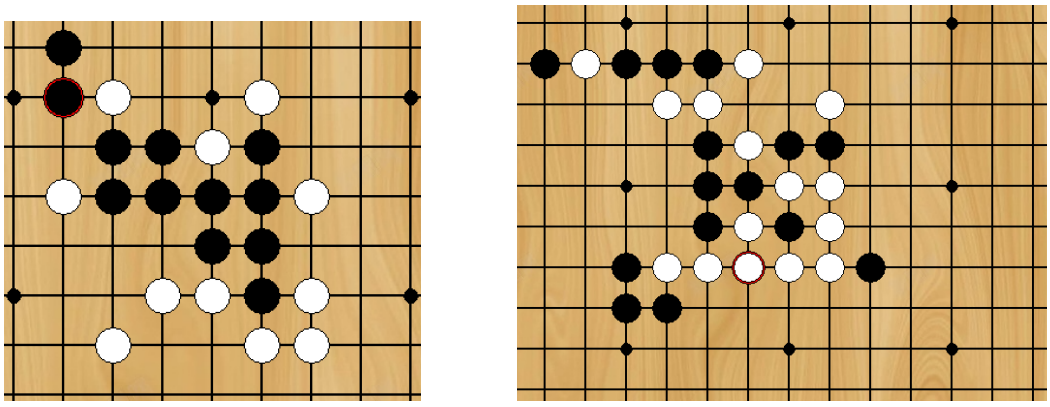

图 5: 人先手（左）和 AI 先手（右）的 alphazero 效果

# 5 Bibliography

[1] T. Y. Hongming Zhang, "Chapter15-alphazero," https://github.com/deep-reinforcement-learning-book/Chapter15-AlphaZero, 2019.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

# A 组内分工

- 王嘉睿：实现了 DQN 部分，并完成报告相关部分的撰写，占总工作量的 33.3%
- 韩金汝：实现 alphabeta 剪枝的基础算法，学习 alphazero 并实现部分代码同时测试，撰写对应内容的部分报告，占比 33.3%
- 王瑶瑶：alphabeta 部分的探索以及优化，alphabeta 以及 alphazero 算法对应的部分报告撰写，占比 33.3%

# B 代码结构

- alphabata-prunning：实现剪枝算法
- DQN：使用 DQN 实现强化学习
- alphazero：使用 alphazero 实现强化学习
- README.md
- report.pdf

具体的测试在相应的文件夹下运行 `main.py` 即可，详细代码文件说明见 README 文件。