

# 我的五子棋 AI 果然有问题

韩梓辰 夏星晨 周贤玮 赵云龙 张坤龙

2020 年 12 月 10 日

## 摘要

近年来, AlphaGo 在棋坛上打遍天下无敌手, 甚至进军电子竞技行业, 人工智能在发展到今天, 人类在竞技体育领域可能越来越不是他们的对手。但是, 显然对胜利的渴求并不新颖, 因为人工智能现在越来越多的在各个领域聪明, 从以前的人工智障变成了人工智能, 在去年, 日本一个公司开发了一款人工智能, 号称史上最弱人工智能, 这个人工智能在几百万次的游戏对战中只获取了 1000 次的胜利, 无论人类如何放水, 这个人工智能反倒越来越弱。于是放弃原有的老套人工智能思路, 改为设计“人工智障”成为了一个全新的设计思路。

该五子棋“人工智障”将基于 Python 编程语言, 通过数学建模, 博弈树, 神经网络等算法实现。使用 pytorch 工具, CUDA 加速实现矩阵运算的优化, 更加优秀的卷积神经网络设计等方法对其进行进一步的优化。最后, 在通过大量的人机对战、机机对战、预设对战的数据的学习下, 该人工智障已具备一定的计算机科学技术上的智能水平, 具有了一定的研究与使用意义。

**关键词:** 人工智能, 五子棋, 神经网络, 人工智障, TensorFlow

## 目录

1 引言	3
2 相关工作	4
2.1 python 学习 . . . . .	4
2.2 对照算法 . . . . .	4
2.2.1 算法概述 . . . . .	4
2.2.2 算法实现 . . . . .	5
2.2.3 算法优点 . . . . .	7
2.3 人工智能实现 . . . . .	7
2.3.1 实现原理 . . . . .	7

2.3.2	策略价值网络训练 . . . . .	10
2.3.3	训练目标 . . . . .	12
2.4	人工智障算法思路 . . . . .	12
<b>3</b>	<b>程序代码</b>	<b>13</b>
3.1	主程序入口 . . . . .	13
3.2	UI 界面 . . . . .	13
3.3	AI . . . . .	18
<b>4</b>	<b>训练情况</b>	<b>39</b>
<b>5</b>	<b>结束语</b>	<b>40</b>
5.1	研究总结 . . . . .	40
5.2	工作展望 . . . . .	40
	<b>参考文献</b>	<b>41</b>

# 1 引言

在计算机科学高速发展的当代,人工智能的上限已经变成了一个未知数。人工智能之父图灵在 1950 年曾说过:下棋是很抽象的活动,是机器可以和人竞争的纯智能领域之一。[1] 自此之后,越来越多的学者开始研发超越人类的 AI,攻克那些曾让人类引以为傲的脑力项目。在 1997 年时,IBM 研发的 Deeper Blue 战胜了当年国际象棋世界冠军卡斯帕罗夫,成为人工智能挑战人类智慧发展的里程碑。[2] 而 2016 年 3 月,谷歌研发的人工智能-阿尔法狗与围棋世界冠军、职业九段棋手李世石进行围棋人机大战,以 4 比 1 的总比分获胜,震惊了棋坛;2016 年末 2017 年初,该程序在中国棋类网站上以“大师”(Master)为注册账号与中日韩数十位围棋高手进行快棋对决,连续 60 局无一败绩,当人们知晓的时候,无不对人工智能的力量感到佩服;2017 年 5 月,在中国乌镇围棋峰会上,它与排名世界第一的世界围棋冠军柯洁对战,以 3 比 0 的总比分获胜,取得了围棋界的王冠。围棋界公认阿尔法围棋的棋力已经超过人类职业围棋顶尖水平。[3] 人工智能在棋类方面令人诧异的表现将它推上了一个新的高度。时至

今日,棋类 AI 的算法技术趋向成熟,大量的优化算法,学习模型的构建被提出、完善,包括决策树,算杀, A\* 搜索等等。这让人工智能在棋类方面几乎变得无懈可击。[4]

就在去年,日本“AVILEN”AI 技术公司的首席技术官吉田拓真却反其道而行,研发出了一款“最弱 AI”。针对这个模型,他构建了五层神经网络,盘面信息为输入层,输出的是棋盘有利度,通过模仿 AlphaGO 的构建,以及使用的算法,他成功做出了这个号称“史上最弱”的人工智能。他在推特上发表了这款支持人机对战的黑白棋小程序,最终,这个黑白棋 AI 在上千名网友的挑战下只输了寥寥数次。这打破了原本“创造胜过人类的人工智能”的固有思维模式。然而,出于时间原因,吉田拓真仅制作了黑白棋的 AI 程序 [5],而目前,在其他棋类游戏方面的“人工智障”还是一片空白。

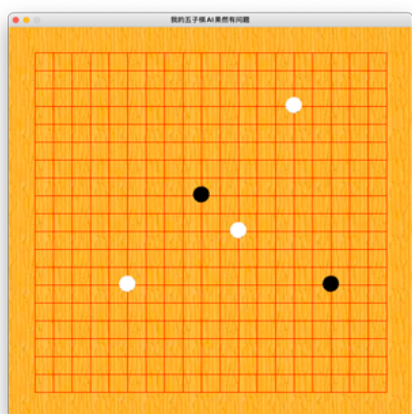
基于这个创意,本组决定转换方向,即通过反向思路实现,将人工智能彻底做成另一个新的方向,即“人工智障”。我们计划设计一款可以不断的被人类战胜的机器,无论人类如何放水都可以输掉整个比赛。本组决定以博弈树,极大极小值搜索,算杀等较为普遍的算法为基础,通过更

加优秀的数学建模，神经网络，ai 的算力的影响等问题，实现了第二版底层优化来实现本组预期制作的五子棋“人工智障”。并将以人与 AI，AI 与 AI 之间的棋局胜负为指标，来验证本组五子棋 AI 的优越性。

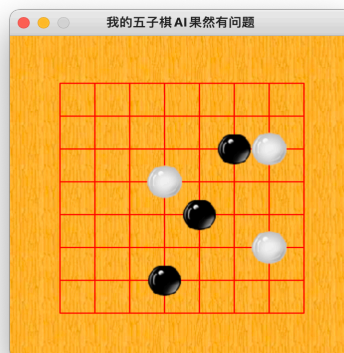
## 2 相关工作

### 2.1 python 学习

由于大家对 python 编程语言并不是很熟悉，所以在项目初期，我们五个人都进行了 python 的学习 [6]，通过 python 的短暂学习，大家均掌握了大部分的 python 语法，包括 pip 的安装库，for 的高级用法。通过在网站上的学习过程，我们逐渐的学习并熟练了 python 的过程，我们利用 pygame 对本次五子棋的图形界面进行了实现。



此版本游戏实现较为简陋，后期我们对棋子的图片进行了重制，并兼顾了



### 2.2 对照算法

在训练神经网络之前，我们需要一个标准算法对我们的模型进行训练，这里我们采用的是蒙特卡洛树搜索。

#### 2.2.1 算法概述

蒙特卡洛树搜索其实并非什么新型算法，早在上个世纪四十年代，为了满足原子能事业的发展，这个算法就已经被投入使用 [8]。而直到 2016 年 3 月，谷歌研发的人工智能—阿尔法狗与围棋世界冠军、职业九段棋手李世石进行围棋人机大战，最终以 4 比 1 的总比分获胜，它才引起了人们的注意，网上也开始出现各种博客、论文等详细解析这个算法。时至今日，蒙特卡洛树搜索已经在棋类游

戏 AI 中被广泛使用。

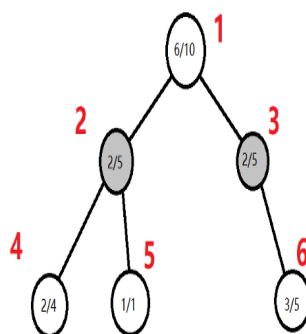
蒙特卡洛树搜索实际上是一种随机模拟与树的搜索的结合 [9]，它最大的优点它能权衡探索与利用，是一个在搜索空间巨大仍然比较有效的搜索算法。对于传统的树搜索算法来说，如果搜索层数较浅，我们可以照常穷举出所有的情况，得到每一个树节点输赢的概率，然后通过最大最小值搜索得到一个纳什均衡点。然而对于搜索层数比较深的情况（如一个  $10 \times 10$  棋盘的五子棋），若要遍历每一个棋局的每一种情况，所有可能的状态将近  $3^{100}$  个<sup>1</sup>，这是现有的计算机无法承受的。这个时候，我们就需要蒙特卡洛树搜索，来帮助我们进行抉择，随机抛弃一些节点，再进行搜索。这样虽然不能得到所有点的权重，但是可以在有限的时间内换取更多胜率更高的点，从而抛弃大量冗余的节点，节省下大量的时间。

### 2.2.2 算法实现

我们设一个节点  $i$  的价值为  $v_i$  (我们可以使用各种公式来决定函数  $v$ ，比如最简单的胜利局数/总局数，或者使用 Upper Confidence

Bounds (UCB) 公式<sup>2</sup>。大部分的蒙特卡洛树搜索包含一下四个阶段 [10]：

1. 选择 (Selection) 在这一步，我们会从树的根节点开始，每次都选择一个“最值得我们搜索的点”，即  $v_i$  最大的子节点，直到我们遇见一个存在未被扩展的子节点的节点。然后我们就对该节点进行扩展。如下图，假设我们遍历到了 5 号点。

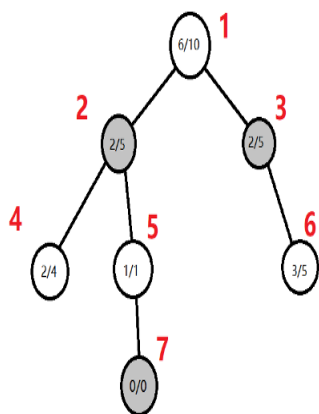


2. 扩展 (Expansion) 我们新建一个子节点 7 作为 5 号节点的扩展 (如图二)。值得注意的是，如果我们当前遍历到的点是白点 (这代表此时轮到白方执子)，子节点的建立必须是一个黑点 (因为此时已经轮到黑方执子)。接着进入第三步模拟。此外，我们的拓展需要有一定的随机性，而非仅仅是按照字典序排序，来保证蒙特

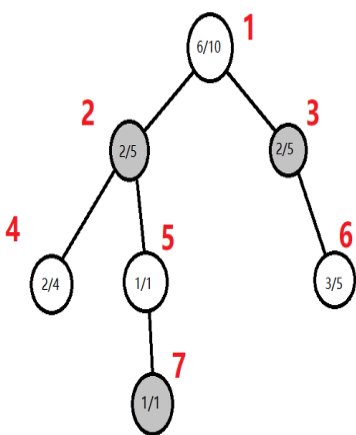
<sup>1</sup>对于棋盘上每个点有三种情况：白子，黑子，无子，一共有 100 个棋格，故大致估算为  $3^{100}$  种情况 (约为  $5.15 \times 10^{47}$  种情况)

<sup>2</sup>Upper Confidence Bounds (UCB) 公式:  $V_i + C * \sqrt{\frac{\ln N}{n_i}}$  其中  $V_i$  是节点的估计价值,  $n_i$  是节点被访问的次数, 而  $N$  则是其父节点已经被访问的总次数。C 是一个可调整参数, 类似于 Perception 中的 learning rate, 它决定了在我们进行蒙特卡洛树搜索时, 遍历的次数对该节点估值的影响大小。

卡洛树搜索能充分扩展到“最有搜索价值”的节点<sup>3</sup>。



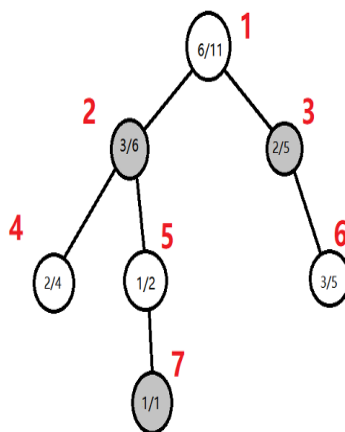
3. 模拟 (Simulation) 在这个步骤中, 我们通过某种特定的下子方法 (如仅仅是优秀的随机, 甚至于连杀, 连防<sup>4</sup>), 来快速走子, 获得最后的胜负情况, 更新到 7 节点上 (如图 3)。



<sup>3</sup>我们考虑这样一种情况: 如果在某一个节点  $i$  的胜率很高, 可能是 50/100, 而另一个节点  $j$  的胜率也有 40/80。那么此时, 我们会选择  $i$  节点进行扩展。当  $i$  节点胜利之后, 我们会再次搜索到  $i$  节点。如果这次的搜索失败了, 那么它的胜率依旧是 1/2。在我们不采用随机数的情况下, 根据字典序, 我们还是会选择节点  $i$  来扩展。这就导致了一个循环,  $i$  节点在不断被扩展, 而  $j$  节点一直不会被访问到——即使  $j$  的实际胜率可能远远高于  $i$  节点。

<sup>4</sup>连杀、连防: 这二者都是五子棋中一种最基础的下法, 它不需要大量的思考与计算, 但是却非常地接近最优解。所以对于新手而言, 连杀与连防是想要下好五子棋必须掌握的策略。具体来说, 连杀是指对于进攻方的每一次落子, 都能形成活三/眠四/三三/四三/四四, 从而逼迫对手进行连防。连防也是一个相似的概念, 它是指对于防守方的每一次落子, 都能将对手的活二/活三/眠四转变为眠二/眠三/死四。当防守方的防守做得足够优秀, 进攻方就无法产生足够大的压力, 此时就可以选择一转攻势, 通过连杀尝试将对手杀棋。

4. 反演 (Backpropagation) 从 7 号点出发, 将获得的结果回溯到父节点上, 更新每一个节点的胜负情况, 如图 4。需要注意的是, 白点代表白子, 灰点代表黑子, 所以更新的时候若黑点胜利则父白点应记为失败, 反之亦然。



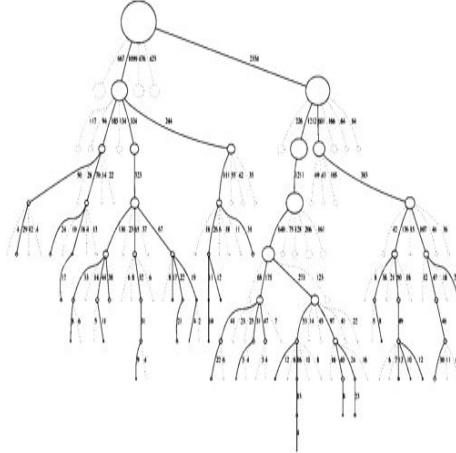
5. 重复步骤 1。蒙特卡洛树搜索可以在任何时候停止, 它的准确度随着搜索时间的增加而收敛。一般来说, 我们也会通过设定搜索深度的限制来防止节点任意地拓展, 减缓运行的速度。

### 2.2.3 算法优点

蒙特卡洛树搜索具有很多的优点 [11]:

1. 泛用性。蒙特卡洛树搜索并不要求有太多的专业知识,只要了解了基本的规则,就能很好地完成它的任务。这使得蒙特卡洛树搜索只要稍加更改就能用于另一个模型。

2. 非对称的扩展。[12]MCTS 执行一种非对称的树的适应搜索空间拓扑结构的生长。这个算法会更频繁地访问更加有趣的节点,并聚焦其搜索时间在更加相关的树的部分。非对称的增长这使得 MCTS 更加适合那些有着更大的分支因子的博弈游戏,比如说 19X19 的围棋。这么大的组合空间会给标准的基于深度或者宽度的搜索方法带来问题,所以 MCTS 的适应性说明它(最终)可以找到那些更加优化的行动,并将搜索的工作聚焦在这些部分。



3. 可被终止。算法可以在任何时候被终止,此时会返回目前所得的最优解。[13]

## 2.3 人工智能实现

基于上述的蒙特卡洛树的算法,我们以此来训练实现我们的人工智能模块。

### 2.3.1 实现原理

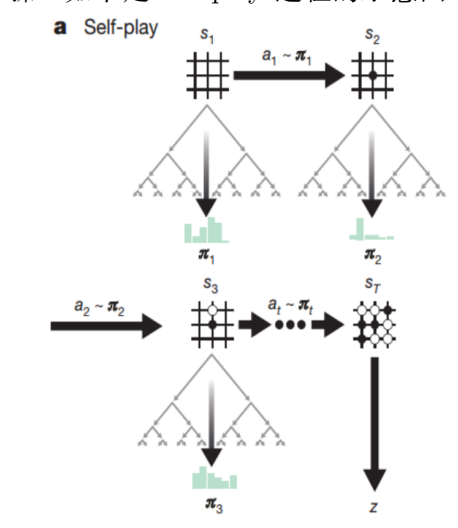
对于每一个节点上的状态  $s$ , 都调用卷积神经网络  $f_\theta$  进行价值评估和策略评估并使用评估结果作为蒙特卡洛树搜索算法(下称 MCTS)的参考。MCTS 经过模拟输出状态  $s$  下,在不同位置落子的概率  $\pi$ 。其中  $\pi$  是一个向量,向量值代表 MCTS 模拟的概率结果。如下所示的向量  $\pi_i$ ,位置 (1,3) 显示的概率值是 0.93,那么表示在该位置落子的可能性非常大。

$$\pi_i = (0.02, 0.03, 0.93, \dots)$$

经过 MCTS 模拟得到的策略  $i$  要比仅使用卷积神经网络  $F$  评估得到的落子概率  $P$  更准确, MCTS 通过探索加强了落子概率  $P$  的可靠性,这一过程称为策略改善 (policy improvement)。以经过 MCTS 增强的策略  $i$  作为落子依据进行模拟,把模拟最终的结果  $z$  作为价值函数的标准。

通过采用策略迭代算法去更新

卷积神经网络  $f_\theta$  的参数  $\theta$ , 优化的目标是令卷积神经网络的输出策略概率与价值评估更接近 MCTS 的模拟结果和博弈最终的胜负结果, 即令  $f_\theta(s) = (P, v) \rightarrow (\pi, z)$ 。并且, 在每一轮迭代中, 总是采用最新的网络参数进行预测和评估。[14] 本组通过该理论为基础, 基于 self-play 来进行“人工智障”的学习、进化。它不仅是整个训练过程中最耗费时间的步骤, 同时也是最关键的步骤。如下是 self-play 过程的示意图:



相较于用传统的自我对局, 本组通过在自我对局中的以下几个细节处理来实现了人工智能的优化:

1. 通过五子棋本身的性质来扩展训练数据的广度, 从而降低训练的重复度。

大部分的棋类游戏都具有一定几何上的性质。对于中国象棋来说,

它的棋盘在水平翻转和垂直翻转的情况下, 局面并不会发生任何的改变。而对于五子棋来说, 它的棋盘在经过  $90^\circ$  的旋转, 以及水平的翻转之后, 本质上棋局并不会发生改变。所以我们可以充分利用这一性质来进行我们自我对局的数据库扩展, 从而减少不必要的搜索, 提高人工智能训练的效率。那么它能够产生多少种相同的局面呢? 我们把图像按  $0^\circ$ ,  $90^\circ$ ,  $180^\circ$ ,  $270^\circ$ , 以棋盘中心为中心旋转, 得到 4 个图形, 每个图形再进行水平翻转, 则得到 8 个图形。当然可能会因为对称或者轴对称, 导致其中的一些相同, 最终只有 4 个或者 2 个甚至 1 个图形, 但是一般情况下我们还是认为这是 8 个图形。[15] 这样虽然会损失部分的空间利用率, 但是却能让代码实现的难度下降一些。从实践角度来说, 在每一次的自我对局结束后, 我们都会将这一局的棋盘的坐标进行翻转, 旋转, 从而得到 8 种相同的棋局, 并

记录在数据库中, 视作人工智能已训练过的棋局。这个优化能够在机器的算力较弱时, 以较快的速度获取更多模型的数据, 同时也能增加自我对局的数据的准确度与多样化。

2. 保证数据的统一性来减少每一个节点状态的维数。

在自我学习的算法执行过程中,



对于每一个局面，我们都会保存大量的数据。我们用  $(\pi_i, z_i)$  来表示当前局面  $i$  的状态，其中  $i$  是根据 MCTS 的根结点处，对于每一个根节点的访问次数的统计从而计算得到的概率，而  $z_i$  则是记录下了自我对局的结果。如果我们从两个棋手的视角去表示所得的状态，这实际上和仅仅计算一名棋手的局面是等价的<sup>5</sup>。从实现上来说，我们会通过使用两个矩阵来储存两个玩家的位置。假设我们把一个棋手的棋子用第一个矩阵表示，而另一个棋手的棋子用第二个矩阵来表示。在这种情况下，我们会通过  $i$  来判断谁是当前的玩家，与此同时，我们的矩阵将会交替表示两个棋手的棋子状态。这对于  $z$  来说也是同理，我们可以通过  $z_i = -1, 0, 1$  来代表  $i$  局面下的棋手失败，和局，以及  $i$  局面下的棋手胜利。只不过  $z$  必须要通过一个完整的对局结束之后才能被我们确定。

3. 通过使用最新模型而非最优模型从而避免“贪心假象”。

在 AlphaZero 的论文中，通过对比不断维护最优模型进行学习的人

工智能与仅仅优化最新模型的人工智能，前者花费了 72 小时所训练得出的人工智能被后者用仅仅 34 小时就超越了。[17] 在 AlphaGo Zero 模型中，作者通过不断地更新最优秀地模型用于训练当前的人工智能，获得最新的模型，从而希望保证它的准确性。然后每经过一段就通过对比历史模型和最新模型来更新当前最优模型。而到了 AlphaZero 模型中，我们并不会去更新历史最优模型，作为代替，我们只使用最新的模型作为自我对局的数据获取手段，同时这些数据被用于更新最新的模型。从常理上来说，前者通过不断维护最优秀的模型理应获得更加收敛的结果。而事实上恰恰相反，这种现象被称为“贪心假象”。为了验证这种情况，本组通过对两种方案的简化与训练，获得了与 AlphaZero 相似的结果：对于 3\*3 的井字棋的游戏中，如果我们不断更新最优模型，计算机运行了近半个小时才达到井字棋的最优解；而对于只用最新模型生成自我对局的情况下，只需要大约十分钟就能获得最优解。尽

<sup>5</sup> 五子棋符合博弈树的模型，而博弈树有以下特点，故保存一方的节点与保存双方的节点是等价的：

- (1) 当前的局面就是博弈的初始状态，即作为博弈的初始节点，其它节点是根据规则来扩展出来的局面。
- (2) 博弈树属于一颗与或树，己方掌握着主动权，一直选择最佳的落子点进行扩展，而对方则需要结合这个落子点进行推测落子，也就是说所有的节点的扩展都是以一方的立场进行扩展。
- (3) 博弈树是双方交替落子形成的，叶子节点可以是对方局面，也可以是一方局面，是由模拟的层数决定的。
- (4) 博弈的本原问题是博弈树的扩展是以己方利益最大化来构建的，能使己方获胜和消减对方的着法都需要考虑，反之，与此意愿相反的着法都是不可取的 [16]。

管这个测验的数据范围较少，训练时间较短，它依旧可以在一定程度上证明 AlphaZero 论文中得到的结论。根据本组的猜测，通过新模型产生的数据可能具有更大的多样性，从而导致在这种情况下能遍历到更为广泛的棋局情况。并且当前的最新模型与过去获得的最优模型相差并没有那么大，从而导致自我对局的质量得以保证。同时最新的模型能覆盖到更多特殊的数据，从而加快了人工智能的学习与收敛。

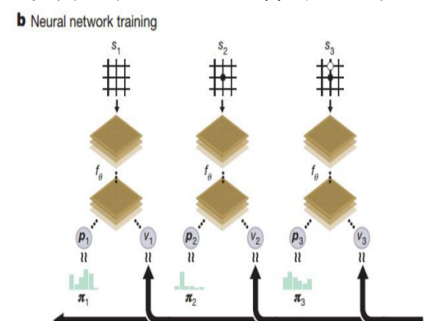
4. 本组通过对于数据的随机化处理来产生更加具有特殊性的数据，来保证人工智能在学习方面不会产生针对某一种局面的部分缺陷。

在 2017 年 6 月 21 日，中国棋手王昊洋六段击败了日本围棋程序 DeepZenGo，让柯洁惊呼“你是人类的希望”！事实上，在之前的对局中，王昊洋连连败退，直到这一局，他终于引发了 DeepZenGo 的 bug。这是由于这个 AI 的局部胜率判断错误，导致它在全局采取的策略出现极大的改变，最终败于王昊洋。局后 DeepZenGo 开发者加藤英树也承认，这是由于该局面的稀有性，导致 AI 在学习的过程中忽略了这一棋局状态。所以，只有在任何局面下都能正确评估当前局面的优劣以及之

后每一步落子的胜率，才能被称作一个优秀的策略价值模型。而对于我们来说，要得到这种策略价值模型，我们应该使用用最新的模型生成的自我对局数据，来帮助人工智能训练到每一个局面。当然，仅仅是这样的优化还不足以遍历到各种各样的局面，所以我们会通过设计一个特殊的拓展策略来扩展到每一种节点，这是不可或缺的。

### 2.3.2 策略价值网络训练

我们通过对整个游戏的价值网络训练，以此实现我们的“人工智能”目的：



策略价值网络训练示意图

在 19 路围棋(棋盘由  $19 \times 19$  个交叉点组成)中，将每一个交叉点编号，总共可以得到 361 个编号的交叉点；同时，可以将这 361 个交叉点映射到 361 类上，每一个交叉点看做是一类，那么当给定某一特定盘面时，该

盘面存在的一种解就是该盘面所属的那一类。这样可以得到运用多分类问题模型给出某一盘面的一个解或多个解的策略网络。策略网络主要解决的是围棋搜索空间太大的问题。每当给定一个特定的盘面,如果不将该盘面所有的落子点作为可能落子点,而是将落子概率最大的几个或十几个落子点作为可能落子点来计算,这样就可以极大的减少搜索空间,提高搜索的效率。[18]

和围棋类似,五子棋同样可以搭建一个相似的策略价值网络。这个网络可以在读入某一个局面情况时,返回在该局面下每一中落子的胜率以及当前局面的评估。我们就是采用之前自我对局所获得的数据来拓展更新策略价值网络。在这之后,为了获得更加优质的自我对局数据,更新完成的策略价值网络会被我们利用在蒙特卡洛树搜索之后进行的自我对局上。这二者就构成了策略价值网络的螺旋上升,让我们的策略价值网络更加准确和高效。

我们将从以下几个方面介绍策略价值网络:

1. 用二值特征平面表示某一个局面。

AlphaGo Zero 一共使用了 17 个 19\*19 的二值特征平面来描述当前局

面,其中前 16 个平面描述了最近 8 步对应的双方 player 的棋子位置,最后一个平面描述当前 player 对应的棋子颜色,也就是先后手。[19] 而本组在经过一定的考虑之后,简化用于表示棋盘的二值特征平面。本组仅仅将 4 个二值特征平面投入使用,其中第一个平面表示当前棋手的落子,而第二个平面表示对手的落子,分别使用 1/0 表示该点有/无子。根据下五子棋的传统经验,要保证每一步走子尽可能地优秀,一般我们会将子下载敌手上一步执子地附近。所以本组构建了第三个平面,在这个平面上,除了一个位置是 1,其他位置均为 0,它的意义是敌手最近一步执子的位置。同时,由于在五子棋游戏中,先后手的优势是巨大的,所以我们会采用最后一个平面来表示当前玩家的先后手情况,如果当前执子玩家是先手,则记作 1,否则记为 0。

## 2. 网络结构的构建。

在 AlphaGo Zero 中,输入局面首先通过了 20 或 40 个基于卷积的残差网络模块,然后再分别接上 2 层或 3 层网络得到策略和价值输出,整个网络的层数有 40 多或 80 多层,更新非常缓慢。[19] 基于时间消耗的考虑,我们同样简化了该网络结构。我们总共构筑了 3 层卷积网络,将 ReLu 作

为激活函数,使用 32、64、128 个 filter。同时,我们将输出分为 policy 和 value。在 value 端,我们会在降维之后链接全连接层,使用非线性函数 tanh 来给出对于当前局面的评分。而对于 policy 端,我们同样会在降维之后链接一个全连接层,然后使用 softmax 函数直接输出每个棋局上每一个节点的落子概率。

### 2.3.3 训练目标

在训练神经网络阶段,使用在蒙特卡洛模拟中生成的数据 (S,F,Z) 进行训练神经网络模型的参数,目的就是使得对于每个输入的局面 S,神经网络的输出策略 p 和价值 v 能够和训练样本中的 F、Z 的差距尽可能地减少,也就是令神经网络的损失函数尽可能地减少。函数如下:

$$l = (Z - v)^2 - F^l \log(p) + c||\theta||^2$$

此损失函数由三部分组成,第一个部分是均方误差损失函数,用于评估神经网络输出的胜负结果与真实对战结果的误差,第二部分采用的是交叉熵损失函数,用于评估神经网络输出的落子策略和 MCTS 输出的落子策略之间的差异,第三部分则是机器学习中传统的 L2 正则化项,可以起到防止过拟合的作用。训练的目

的就是在自我对战数据集上不断地调整参数,以达到损失函数减少地目的。[16]

通过全面地分析损失函数,我们发现它在一开始剧烈地下降,随后稳定地以一个较低的速率下降,这意味着我们的人工智能所下的子与真实对战越来越接近。同时,我们还会关注策略价值网络输出的落子概率分布的交叉熵的变化情况,在训练开始的一段时间,我们得到的落子概率都是随机的,这导致了交叉熵较大的情况。而随着时间的推移,交叉熵会逐渐变小,这是因为我们的策略网络逐渐学会了在不同的局面下哪些位置能有更大的执子概率。这种落子概率的偏向帮助了蒙特卡洛树搜索过程中能在更加“有趣”的位置进行扩展,得到更高的学习效率。

## 2.4 人工智障算法思路

基于人工智能的算法,我们认为人工智障就是对人工智能计算得到的概率矩阵找到其中获胜概率最低的点,然后不断在该点落子即可。

方法过程可以按照如下去理解,为了简化,我们以 2x2 矩阵为样例:

我们获得一个概率阵:

$$\begin{bmatrix} 0.2 & 0.5 \\ 1 & 0.6 \end{bmatrix}$$

我们对整个矩阵中的元素加以遍历,找到最小的值 min=0.2,坐标

为 (0,0) 则我们将棋子下在该位置, 以此循环遍历即可获得我们的人工智障

这样处理的思路有以下优点:

1. 可以对人工智能的数据直接进行利用, 理论上, 人工智能越聪明, 人工智障越傻

2. 运算量相同, 可以加速程序运行

通过以上的处理方法, 我们将人工智障的程序也顺利做了出来, 经过测试可以实现在人类显著放水的情况下其仍然能让人类胜利。

## 3 程序代码

### 3.1 主程序入口

这里可以通过修改 `Ai_or_notAI = 'AI'` 来选择是否和人工智能对战

```
#####
# author:happys
# coding:UTF8
# 程序目的: 程序进入的入口
#####
# from AI.train import *
# training_pipeline =
#     TrainPipeline()
# training_pipeline.run()
#
from AI.human_play import
*
Ai_or_notAI = 'AI' # 选择
```

智能还是智障

```
# Ai_or_notAI = 'notAI' # 选择智能还是智障
run(Ai_or_notAI)
```

### 3.2 UI 界面

UI\_Board.py

```
# code:utf-8
# author:zkl, 白炎
from time import sleep
import pygame
import tkinter
import tkinter.messagebox
from pygame.locals import *
from enum import Enum
# 颜色RGB
from UI import BoardData
class Color(Enum):
    WHITE = 0
    BLACK = 1
    RED = (255, 0, 0)
# 棋子类
class chess():
    # 初始化
    def __init__(self, screen, color, row, column):
        self.screen = screen
        self.image = pygame.image.load("AI/Images/chess" + str(color.value) + ".png")
        self.image = pygame.transform.scale(self.image, (Chessboard.UNIT * 2, Chessboard.UNIT * 2))
        self.pos = Chessboard.
```

```

        convertArrayToPos(row,
        column)
    self.rect = self.image.
    get_rect()
    self.rect = self.pos
    self.screen.blit ( self.image,
        self.rect)
# 棋盘类
class Chessboard:

    # 初始化棋盘
    def __init__(self, board_data,
        UNIT=35, width=50):
        self.ROW = board_data.row
        self.COLUMN = board_data.
        column
        Chessboard.UNIT = UNIT
        Chessboard.width = width

        # 绘制棋盘, 传入当前棋盘, 窗
        口, 边框宽度
    def drawmap(self, screen, width):
        # 行
        for row in range(self.COLUMN)
            :
            pygame.draw.line(screen, Color.
                RED.value,(width, width +
                row * self.UNIT),(width +
                self.UNIT * (self.ROW -
                1),width + row * self.UNIT
                ))
        # 列
        for column in range(self.ROW):
            pygame.draw.line(screen, Color.
                RED.value,(width +
                column * self.UNIT, width)
                ,(width + column * self.
                UNIT,width + self.UNIT *
                (self.COLUMN - 1)))
    @staticmethod
    def convertArrayToPos(row,
        column):
        return Chessboard.width + (
            row - 1) * Chessboard.
            UNIT, Chessboard.width +
            (column - 1) *
            Chessboard.UNIT
    @staticmethod
    def convertPosToArray(x, y):
        return (x - Chessboard.width
            // 2) // Chessboard.UNIT,
            (y - Chessboard.width //
            2) // Chessboard.UNIT

class bboard:
    def __init__(self,boardData):
        self.boardData=boardData

# 主程序
# 初始化界面
screen = 1

def set_human(human):
    global Human
    Human=human

def init_game():
    # 设置边框宽度

```

```

BOARD_WIDTH = 50
# 创建棋盘
global boardData
boardData = BoardData.
    BoardData(8, 8)
chessBoard = Chessboard(
    boardData)
pygame.init()

SIZE = (2 * BOARD_WIDTH + chessBoard.UNIT * (
    chessBoard.ROW - 1),
    2 * BOARD_WIDTH + chessBoard.UNIT * (
    chessBoard.COLUMN - 1)
)
# 创建窗口
global screen
screen = pygame.display.
    set_mode(SIZE)
# 设置窗口标题
pygame.display.set_caption("我
    的五子棋AI果然有问题")
# 设置背景
background = pygame.image.load
    ('AI/Images/bg.jpg')
background = pygame.transform.
    scale(background, SIZE)

screen.blit(background, (0, 0))
# 绘制棋盘
# Chessboard.drawmap(screen,
    BOARD_WIDTH)
chessBoard.drawmap(screen, 50)

pygame.display.update()

def updateBoard(board):
    if boardData:
        boardData.
            transportToBoardData(
                board)
        showChess()

# To solve mouse click event
def checkEvent(event):
    if event.type ==
        MOUSEBUTTONDOWN:
        row, column = Chessboard.
            convertPosToArray(event.
                pos[0], event.pos[1])
        putChessOnBoard(row, column
            )
        pygame.display.update()
    pass

def putChessOnBoard(row, column
    ):
    # pass
    print(row,column)
    if Human!=None:
        Human.give_input(boardData.
            row-1-column,row)
    # Human.give_input(row,column
        )
    # isWin = boardData.putChess(
        row, column)
    # if isWin != False:
    # chessNew = chess(screen,
        Color.WHITE if boardData.

```

```

        thisTurn else Color.BLACK,
        row, column)
# print(row,column)
# print(is Win)
# if is Win:
# tkinter.messagebox.
askokcancel( title ="Who Won
?", message="White won" if
boardData.thisTurn else "
Black won");
# init_game()
# return is Win

def showChess():
    for i in range(boardData.row):
        for j in range(boardData.
            column):
            if boardData.getBoard()[i][j]
                ==1:
                chessNew=chess(screen, Color
                    .WHITE , i, j)
            elif boardData.getBoard()[i][j]
                ==-1:
                chessNew = chess(screen,
                    Color.BLACK, i, j)
            pygame.display.update()

# 控制游戏进程
def open_UI():

    # init_game()
    while True:
        sleep (0.5)
        for event in pygame.event.get
            ():
            if event.type == QUIT:

        exit ()
    else:
        checkEvent(event)

if __name__=="__main__":
    open_UI()

BoardData.py

# author:白炎拉力

import numpy as np
from enum import Enum

class Chess(Enum):
    White = 1
    Black = -1

class BoardData:
    # in array board, 1 means white
    chess, -1 means black chess,
    0 means nothing
    thisTurn = False # this turn is
        true when it's white chess
        ready to put

    def __init__(self, row, column):
        self.row = row
        self.column = column
        print([row, column])
        self.board = np.zeros((row,
            column), dtype=int)

```



```

def getBoard(self):
    # print(self.board)
    return self.board

def transportToBoardData(self,
    boardFromGame):
    for location in
        boardFromGame.states:
x=self.row - 1 - location //
    self.row
y=location % self.row
    # temp=x
    # x=(self.row-1)-y
    # y=(self.row-1)-temp
    temp=x
x=y
y=temp
if boardFromGame.states[
    location] == 1:
    self.board[x][y] = 1
elif boardFromGame.states[
    location] == 2:
    self.board[x][y] = -1
else:
    self.board[x][y] = 0

def putChess(self, row, column):
    if row >= self.row or column
        >= self.column or row < 0
        or column < 0:
    return False
if self.board[row, column] ==
    0:
if self.thisTurn:
    self.board[row, column] = 1
else:
    self.board[row, column] =
        -1
if self.checkWin(row, column)
    :
    return True
self.thisTurn = not self.
    thisTurn
else:
    return False

def checkWinAll(self):
    for row in range(len(self.
        board)):
    for column in range(len(self.
        board[0])):
        if self.checkWin(row,
            column):
            return True
    return False

def checkWin(self, row, column):
    color = self.board[row, column
        ]
    directions = ((1, 0), (0, 1),
        (1, 1), (1, -1))
    if (color == 0): return False
    for direction in directions:
        count = 1
        step = 1
        while self.inBound(row +
            direction[0] * step, column
            + direction[1] * step) and
            color == self.board[
                row + direction[0] * step,
                column + direction[1] *
                step]:

```

```

        count += 1
        step += 1
        if count == 5: return True
    step = 1
    while self.inBound(row -
        direction[0] * step, column
        - direction[1] * step) and
        color == self.board[
    row - direction[0] * step,
        column - direction[1] *
        step]:
        count += 1
        step += 1
        if count == 5: return True

def inBound(self, row, column):
    if 0 <= row < len(self.board)
        and 0 <= column < len(
        self.board[0]):
    return True
    else:
    return False

if __name__ == '__main__':
    board = BoardData(5, 5)
    for i in range(5):
        board.putChess(0, i)
        print(board.checkWinAll())
        board.putChess(1, i)
        board.getBoard()

```

**3.3 AI**

policy\_value\_net\_pytorch.py  
训练模型

```

# -*- coding: utf-8 -*-
"""
An implementation of the
policyValueNet in PyTorch
Tested in PyTorch 0.2.0 and 0.3.0
"""

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import
    Variable
import numpy as np

def set_learning_rate(optimizer, lr
    ):
    """Sets the learning rate to the
        given value"""
    for param_group in optimizer.
        param_groups:
        param_group['lr'] = lr

class Net(nn.Module):
    """policy-value network module
        """
    def __init__(self, board_width,
        board_height):
        super(Net, self).__init__()

        self.board_width =
            board_width
        self.board_height =

```

```

        board_height
    # common layers
    self.conv1 = nn.Conv2d(4, 32,
        kernel_size=3, padding=1)
    self.conv2 = nn.Conv2d(32,
        64, kernel_size=3, padding
        =1)
    self.conv3 = nn.Conv2d(64,
        128, kernel_size=3,
        padding=1)
    # action policy layers
    self.act_conv1 = nn.Conv2d
        (128, 4, kernel_size=1)
    self.act_fc1 = nn.Linear(4*
        board_width*
        board_height,
        board_width*
        board_height)
    # state value layers
    self.val_conv1 = nn.Conv2d
        (128, 2, kernel_size=1)
    self.val_fc1 = nn.Linear(2*
        board_width*
        board_height, 64)
    self.val_fc2 = nn.Linear(64, 1)

def forward(self, state_input):
    # common layers
    x = F.relu(self.conv1(
        state_input))
    x = F.relu(self.conv2(x))
    x = F.relu(self.conv3(x))
    # action policy layers
    x_act = F.relu(self.act_conv1(
        x))
    x_act = x_act.view(-1, 4*self.

        board_width*self.
        board_height)
    x_act = F.log_softmax(self.
        act_fc1(x_act))
    # state value layers
    x_val = F.relu(self.val_conv1(
        x))
    x_val = x_val.view(-1, 2*self.
        board_width*self.
        board_height)
    x_val = F.relu(self.val_fc1(
        x_val))
    x_val = F.tanh(self.val_fc2(
        x_val))
    return x_act, x_val

class PolicyValueNet():
    """policy-value network """
    def __init__(self, board_width,
        board_height,
        model_file=None, use_gpu
            =False):
        self.use_gpu = use_gpu
        self.board_width =
            board_width
        self.board_height =
            board_height
        self.l2_const = 1e-4 # coef of
            l2 penalty
        # the policy value net module
        if self.use_gpu:
            self.policy_value_net = Net(
                board_width,
                board_height).cuda()
        else:

```

```

self.policy_value_net = Net(
    board_width,
    board_height)
self.optimizer = optim.Adam(
    self.policy_value_net.
    parameters(),
    weight_decay=self.
    l2_const)

if model_file:
    net_params = torch.load(
        model_file)
    self.policy_value_net.
        load_state_dict(
            net_params)

def policy_value(self,
    state_batch):
    """
    input: a batch of states
    output: a batch of action
            probabilities and state
            values
    """
    if self.use_gpu:
        state_batch = Variable(torch.
            FloatTensor(state_batch).
            cuda())
    log_act_probs, value = self.
        policy_value_net(
            state_batch)
    act_probs = np.exp(
        log_act_probs.data.cpu().
        numpy())
    return act_probs, value.data.
        cpu().numpy()

else:
    state_batch = Variable(torch.
        FloatTensor(state_batch))
    log_act_probs, value = self.
        policy_value_net(
            state_batch)
    act_probs = np.exp(
        log_act_probs.data.numpy
        ())
    return act_probs, value.data.
        numpy()

def policy_value_fn(self, board):
    """
    input: board
    output: a list of (action,
        probability) tuples for
        each available
        action and the score of the
        board state
    """
    legal_positions = board.
        availables
    current_state = np.
        ascontiguousarray(board.
            current_state()).reshape(
            -1, 4, self.board_width,
            self.board_height))
    if self.use_gpu:
        log_act_probs, value = self.
            policy_value_net(
                Variable(torch.
                    from_numpy(
                        current_state)).cuda().
                    float())
        act_probs = np.exp(

```

```

        log_act_probs.data.cpu().
        numpy().flatten())
    else:
        log_act_probs, value = self.
        policy_value_net(
            Variable(torch.
                from_numpy(
                    current_state)).float()
            )
        act_probs = np.exp(
            log_act_probs.data.numpy
            ().flatten())
        act_probs = zip(
            legal_positions, act_probs[
                legal_positions])
        value = value.data [0][0]
    return act_probs, value

def train_step(self, state_batch
    , mcts_probs, winner_batch,
    lr):
    """perform a training step"""
    # wrap in Variable
    if self.use_gpu:
        state_batch = Variable(torch.
            FloatTensor(state_batch).
            cuda())
        mcts_probs = Variable(torch.
            FloatTensor(mcts_probs).
            cuda())
        winner_batch = Variable(torch.
            FloatTensor(winner_batch
                ).cuda())
    else:
        state_batch = Variable(torch.
            FloatTensor(state_batch))
        mcts_probs = Variable(torch.
            FloatTensor(mcts_probs))
        winner_batch = Variable(torch.
            FloatTensor(winner_batch
                ))

    # zero the parameter gradients
    self.optimizer.zero_grad()
    # set learning rate
    set_learning_rate(self.
        optimizer, lr)

    # forward
    log_act_probs, value = self.
        policy_value_net(
            state_batch)
    # define the loss =  $(z - v)^2 - \pi^T * \log(p) + c/|\theta|$ 
    # Note: the L2 penalty is
    # incorporated in optimizer
    value_loss = F.mse_loss(value.
        view(-1), winner_batch)
    policy_loss = -torch.mean(
        torch.sum(mcts_probs*
            log_act_probs, 1))
    loss = value_loss +
        policy_loss
    # backward and optimize
    loss.backward()
    self.optimizer.step()
    # calc policy entropy, for
    # monitoring only
    entropy = -torch.mean(
        torch.sum(torch.exp(
            log_act_probs) *

```

```

        log_act_probs, 1)
    )
    # return loss.data[0], entropy.
    data[0]
    #for pytorch version >= 0.5
    please use the following
    line instead.
    return loss.item(), entropy.
    item()

def get_policy_param(self):
    net_params = self.
    policy_value_net.
    state_dict()
    return net_params

def save_model(self, model_file):
    """ save model params to file
    """
    net_params = self.
    get_policy_param() # get
    model params
    torch.save(net_params,
    model_file)

```

游戏进程 game.py

```

from __future__ import
    print_function
import numpy as np
from tkinter import messagebox
import UI.UI_Board
import sys
import threading

class Board(object):
    """board for the game"""

    def __init__(self, **kwargs):
        self.width = int(kwargs.get('
        width', 8))
        self.height = int(kwargs.get('
        height', 8))
        # board states stored as a dict
        ,
        # key: move as location on the
        board,
        # value: player as pieces type
        self.states = {}
        # need how many pieces in a
        row to win
        self.n_in_row = int(kwargs.
        get('n_in_row', 5))
        self.players = [1, 2] #
        player1 and player2

    def init_board(self, start_player
    =0):
        if self.width < self.n_in_row
        or self.height < self.
        n_in_row:
            raise Exception('board_width_
            and_height_can_not_be_
            ' less_than_{}'.format(
                self.n_in_row))
        self.current_player = self.
        players[start_player] #
        start player
        # keep available moves in a
        list

```

```

self . availables = list(range(
    self.width * self.height))
self . states = {}
self . last_move = -1

def move_to_location(self, move
    ):
    """
    3*3 board's moves like:
    6 7 8
    3 4 5
    0 1 2
    and move 5's location is (1,2)
    """
    h = move // self.width
    w = move % self.width
    return [h, w]

def location_to_move(self,
    location):
    if len(location) != 2:
        return -1
    h = location[0]
    w = location[1]
    move = h * self.width + w
    if move not in range(self.
        width * self.height):
        return -1
    return move

def current_state(self):
    """return the board state from
        the perspective of the
        current player.
    state shape: 4*width*height
    """
    square_state = np.zeros((4,
        self.width, self.height))
    if self . states :
        moves, players = np.array(list(
            zip(*self . states . items())))
        move_curr = moves[players ==
            self.current_player]
        move_oppo = moves[players !=
            self.current_player]
        square_state[0][move_curr //
            self.width,
                move_curr % self.height]
            = 1.0
        square_state[1][move_oppo //
            self.width,
                move_oppo % self.height]
            = 1.0
        # indicate the last move
        location
        square_state[2][ self . last_move
            // self.width,
                self . last_move % self.
                    height] = 1.0
        if len(self . states) % 2 == 0:
            square_state [3][:, :] = 1.0 #
                indicate the colour to
                    play
        return square_state[:, ::-1, :]

def do_move(self, move):
    if (move==None):return
    self . states [move] = self.
        current_player
    self . availables . remove(move)
    self . current_player = (

```

```

        self.players[0] if self.
            current_player == self.
            players[1]
        else self.players[1]
    )
    self.last_move = move

def has_a_winner(self):
    width = self.width
    height = self.height
    states = self.states
    n = self.n_in_row

    moved = list(set(range(width
        * height)) - set(self.
            availables))
    if len(moved) < self.n_in_row
        *2-1:
    return False, -1

    for m in moved:
        h = m // width
        w = m % width
        player = states[m]

        if (w in range(width - n +
            1) and
            len(set(states.get(i, -1)
                for i in range(m, m
                    + n))) == 1):
            return True, player

        if (h in range(height - n +
            1) and
            len(set(states.get(i, -1)
                for i in range(m, m
                    + n * width, width)))
            == 1):
            return True, player

        if (w in range(n - 1, width)
            and h in range(height -
                n + 1) and
            len(set(states.get(i, -1)
                for i in range(m, m
                    + n * (width - 1),
                        width - 1))) == 1):
            return True, player

    return False, -1

def game_end(self):
    """Check whether the game is
        ended or not"""
    win, winner = self.
        has_a_winner()
    if win:
        return True, winner
    elif not len(self.availables):
        return True, -1
    return False, -1

def get_current_player(self):

```



```

        return self.current_player

class Game(object):
    """game server"""

    def __init__(self, board, **
        kwargs):
        self.board = board # type:
            Board

    def showUI(self, board):
        UI.UI_Board.updateBoard(
            board)
        # globalVar.get_value("
            boardData").
            transportToBoardData(
                board)
        # UI.UI_Board.boardData.
            transportToBoardData(
                board)
        # UI.UI_Board.showChess()
        pass

    def graphic(self, board, player1,
        player2):
        """Draw the board and show
            game info"""

        self.showUI(board)

        width = board.width
        height = board.height

        print("Player", player1, "with
            _X".rjust(3))
        print("Player", player2, "with
            _O".rjust(3))
        print()
        for x in range(width):
            print("{0:8}".format(x), end=
                ")
            print('\r\n')
            for i in range(height - 1, -1,
                -1):
                print("{0:4d}".format(i), end
                    =")
                for j in range(width):
                    loc = i * width + j
                    p = board.states.get(loc,
                        -1)
                    if p == player1:
                        print('X'.center(8), end=
                            ')
                    elif p == player2:
                        print('O'.center(8), end=
                            ')
                    else:
                        print('_'.center(8), end=
                            ')
                print('\r\n\r\n')

    def start_play(self, player1,
        player2, start_player=0,
        is_shown=1):
        """start a game between two
            players"""

```

```

    if start_player not in (0, 1):
        raise Exception('start_player_
            should_be_either_0_(
                player1_first)_
                'or_1_(player2_first)')
    self.board.init_board(
        start_player)
    UI.UI_Board.set_human(None
    )
    UI.UI_Board.init_game()
    t2=threading.Thread(target=
        self.start_play1,args=(
            player1, player2,
            start_player, is_shown) )
    t2.start()
    target=threading.Thread(
        target=UI.UI_Board.
        open_UI())
    target.start()

def start_play1(self,player1,
    player2, start_player=0,
    is_shown=1):
    p1, p2 = self.board.players
    player1.set_player_ind(p1)
    player2.set_player_ind(p2)
    players = {p1: player1, p2:
        player2}
    if is_shown:
        self.graphic(self.board,
            player1.player, player2.
            player)
    while True:
        current_player = self.board.
            get_current_player()
        player_in_turn = players[
            current_player]
        if(type(player_in_turn).
            __name__=="Human"):
            UI.UI_Board.set_human(
                player_in_turn)
        #TODO: Change the action
        move = player_in_turn.
            get_action(self.board)
        if move==None:
            continue
        self.board.do_move(move)
        if is_shown:
            self.graphic(self.board,
                player1.player, player2.
                player)
        end, winner = self.board.
            game_end()
        if end:
            if is_shown:
                if winner != -1:
                    print("Game_end. Winner
                        _is", players[winner])
                    if(winner == 1):
                        messagebox.showinfo('
                            You_win', '恭喜您获
                            胜')
                    else:
                        messagebox.showinfo('
                            You_lose', '您输了')
                    else:
                        print("Game_end. Tie")
            return winner

def start_self_play(self, player,
    is_shown=0, temp=1e-3):
    """ start a self-play game

```

```

        using a MCTS player,
        reuse the search tree,
and store the self-play data: (
    state, mcts_probs, z) for
    training
"""
self.board.init_board()
p1, p2 = self.board.players
states, mcts_probs,
    current_players = [], [], []
while True:
    move, move_probs = player.
        get_action(self.board,
                    temp=temp,
                    return_prob=1)
# store the data
states.append(self.board.
    current_state())
mcts_probs.append(
    move_probs)
current_players.append(self.
    board.current_player)
# perform a move
self.board.do_move(move)
if is_shown:
    self.graphic(self.board, p1,
        p2)
end, winner = self.board.
    game_end()
if end:
    # winner from the perspective
    of the current player of
    each state
    winners_z = np.zeros(len(
        current_players))
    if winner != -1:
        winners_z[np.array(
            current_players) ==
            winner] = 1.0
        winners_z[np.array(
            current_players) !=
            winner] = -1.0
# reset MCTS root node
player.reset_player()
if is_shown:
    if winner != -1:
        print("Game_end. Winner
            is player:", winner)
    else:
        print("Game_end. Tie")
return winner, zip(states,
    mcts_probs, winners_z)

价值网络矩阵 policy_value_net_numpy.py

# -*- coding: utf-8 -*-
"""
Implement the policy value network
using numpy, so that we can
play with the
trained AI model without installing
any DL framework

"""

from __future__ import
    print_function
import numpy as np

# some utility functions
def softmax(x):
    probs = np.exp(x - np.max(x))

```

```

probs /= np.sum(probs)
return probs

def relu(X):
    out = np.maximum(X, 0)
    return out

def conv_forward(X, W, b, stride
                 =1, padding=1):
    n_filters, d_filter, h_filter,
        w_filter = W.shape
    # theano conv2d flips the filters
        (rotate 180 degree) first
    # while doing the calculation
    W = W[:, :, ::-1, ::-1]
    n_x, d_x, h_x, w_x = X.shape
    h_out = (h_x - h_filter + 2 *
              padding) / stride + 1
    w_out = (w_x - w_filter + 2 *
              padding) / stride + 1
    h_out, w_out = int(h_out), int
        (w_out)
    X_col = im2col_indices(X,
                           h_filter, w_filter,
                           padding=padding, stride=
                               stride)
    W_col = W.reshape(n_filters,
                      -1)
    out = (np.dot(W_col, X_col).T
            + b).T
    out = out.reshape(n_filters,
                      h_out, w_out, n_x)
    out = out.transpose(3, 0, 1, 2)
    return out

def fc_forward(X, W, b):
    out = np.dot(X, W) + b
    return out

def get_im2col_indices(x_shape,
                       field_height,
                       field_width, padding=1,
                       stride=1):
    # First figure out what the size
        of the output should be
    N, C, H, W = x_shape
    assert (H + 2 * padding -
            field_height) % stride == 0
    assert (W + 2 * padding -
            field_height) % stride == 0
    out_height = int((H + 2 *
                      padding - field_height) /
                     stride + 1)
    out_width = int((W + 2 *
                    padding - field_width) /
                   stride + 1)

    i0 = np.repeat(np.arange(
        field_height), field_width)
    i0 = np.tile(i0, C)
    i1 = stride * np.repeat(np.
        arange(out_height),
        out_width)
    j0 = np.tile(np.arange(
        field_width), field_height *
        C)
    j1 = stride * np.tile(np.arange(
        out_width), out_height)

```

```

i = i0.reshape(-1, 1) + i1.
    reshape(1, -1)
j = j0.reshape(-1, 1) + j1.
    reshape(1, -1)

k = np.repeat(np.arange(C),
    field_height * field_width).
    reshape(-1, 1)

return (k.astype(int), i.astype(
    int), j.astype(int))

def im2col_indices(x, field_height,
    field_width, padding=1,
    stride=1):
    """ An implementation of im2col
        based on some fancy
        indexing """
    # Zero-pad the input
    p = padding
    x_padded = np.pad(x, ((0, 0), (0,
        0), (p, p), (p, p)), mode=
        'constant')

    k, i, j = get_im2col_indices(x.
        shape, field_height,
        field_width, padding,
        stride)

    cols = x_padded[:, k, i, j]
    C = x.shape[1]
    cols = cols.transpose(1, 2, 0).
        reshape(field_height *
        field_width * C, -1)
return cols

class PolicyValueNetNumpy():
    """policy-value network in
        numpy """
    def __init__(self, board_width,
        board_height, net_params):
        self.board_width =
            board_width
        self.board_height =
            board_height
        self.params = net_params

    def policy_value_fn(self, board):
        """
        input: board
        output: a list of (action,
            probability) tuples for
            each available
            action and the score of the
            board state
        """
        legal_positions = board.
            availables
        current_state = board.
            current_state()

        X = current_state.reshape(-1,
            4, self.board_width, self.
            board_height)
        # first 3 conv layers with
            ReLu nonlinearity
        for i in [0, 2, 4]:
            X = relu(conv_forward(X, self.
                params[i], self.params[i
                +1]))

```

```

    # policy head
    X_p = relu(conv_forward(X,
        self.params[6], self.params
        [7], padding=0))
    X_p = fc_forward(X_p.flatten
        (), self.params[8], self.
        params[9])
    act_probs = softmax(X_p)
    # value head
    X_v = relu(conv_forward(X,
        self.params[10],
        self.params[11],
        padding=0))
    X_v = relu(fc_forward(X_v.
        flatten(), self.params[12],
        self.params[13]))
    value = np.tanh(fc_forward(
        X_v, self.params[14], self.
        params[15]))[0]
    act_probs = zip(
        legal_positions, act_probs.
        flatten()[legal_positions])
    return act_probs, value

MCTS 模型实现

    # -*- coding: utf-8 -*-
    """
    A pure implementation of the
    Monte Carlo Tree Search (
    MCTS)

    """

    import numpy as np
    import copy
    from operator import itemgetter

    def rollout_policy_fn(board):
        """a coarse, fast version of
        policy_fn used in the rollout
        phase."""
        # rollout randomly
        action_probs = np.random.rand(
            len(board.availables))
        return zip(board.availables,
            action_probs)

    def policy_value_fn(board):
        """a function that takes in a
        state and outputs a list of (
        action, probability)
        tuples and a score for the state
        """
        # return uniform probabilities
        and 0 score for pure MCTS
        action_probs = np.ones(len(
            board.availables))/len(board.
            availables)
        return zip(board.availables,
            action_probs), 0

    class TreeNode(object):
        """A node in the MCTS tree.
        Each node keeps track of its
        own value Q,
        prior probability P, and its
        visit-count-adjusted prior
        score u.
        """

```

```

def __init__(self, parent,
              prior_p):
    self._parent = parent
    self._children = {} # a map
                        from action to TreeNode
    self._n_visits = 0
    self._Q = 0
    self._u = 0
    self._P = prior_p

def expand(self, action_priors):
    """Expand tree by creating new
        children.
    action_priors: a list of tuples
                   of actions and their prior
                   probability
    according to the policy
        function.
    """
    for action, prob in
        action_priors:
    if action not in self._children:
        self._children[action] =
            TreeNode(self, prob)

def select(self, c_puct):
    """Select action among
        children that gives
        maximum action value  $Q$ 
        plus bonus  $u(P)$ .
    Return: A tuple of (action,
        next_node)
    """
    return max(self._children,
                items(),
                key=lambda act_node:
                    act_node[1].get_value(
                        c_puct))

def update(self, leaf_value):
    """Update node values from
        leaf evaluation.
    leaf_value: the value of
        subtree evaluation from
        the current player's
        perspective.
    """
    # Count visit.
    self._n_visits += 1
    # Update Q, a running average
    of values for all visits.
    self._Q += 1.0*(leaf_value -
                    self._Q) / self._n_visits

def update_recursive(self,
                    leaf_value):
    """Like a call to update(), but
        applied recursively for
        all ancestors.
    """
    # If it is not root, this node's
    parent should be updated
    first.
    if self._parent:
        self._parent.update_recursive(
            -leaf_value)
    self.update(leaf_value)

def get_value(self, c_puct):
    """Calculate and return the

```

```

        value for this node.
    It is a combination of leaf
        evaluations  $Q$ , and this
        node's prior
    adjusted for its visit count,  $u$ 
        .
    c_puct: a number in  $(0, \text{inf})$ 
        controlling the relative
        impact of
    value  $Q$ , and prior probability
         $P$ , on this node's score.
    """
    self._u = (c_puct * self._P *
        np.sqrt( self._parent.
            _n_visits) / (1 + self.
                _n_visits))
    return self._Q + self._u

def is_leaf( self ):
    """Check if leaf node (i.e. no
        nodes below this have been
        expanded).
    """
    return self._children == {}

def is_root( self ):
    return self._parent is None

class MCTS(object):
    """A simple implementation of
        Monte Carlo Tree Search."""

    def __init__(self,
        policy_value_fn, c_puct=5,
        n_playout=10000):
        """
        policy_value_fn: a function
            that takes in a board state
            and outputs
            a list of (action, probability)
            tuples and also a score in
             $[-1, 1]$ 
            (i.e. the expected value of the
            end game score from the
            current
            player's perspective ) for the
            current player.
        c_puct: a number in  $(0, \text{inf})$ 
            that controls how quickly
            exploration
            converges to the maximum-
            value policy. A higher value
            means
            relying on the prior more.
        """
        self._root = TreeNode(None,
            1.0)
        self._policy =
            policy_value_fn
        self._c_puct = c_puct
        self._n_playout = n_playout

    def _playout(self, state):
        """Run a single playout from
            the root to the leaf,
            getting a value at
            the leaf and propagating it
            back through its parents.
            State is modified in-place, so
            a copy must be provided.
        """

```



```

node = self._root
while(1):
    if node.is_leaf():

        break
    # Greedily select next move.
    action, node = node.select(self
        ._c_puct)
    state.do_move(action)

    action_probs, _ = self._policy
        (state)
    # Check for end of game
    end, winner = state.game_end
        ()
    if not end:
        node.expand(action_probs)
    # Evaluate the leaf node by
        random rollout
    leaf_value = self.
        _evaluate_rollout(state)
    # Update value and visit count
        of nodes in this traversal.
    node.update_recursive(-
        leaf_value)

def _evaluate_rollout(self, state
    , limit=1000):
    """Use the rollout policy to
        play until the end of the
        game,
        returning +1 if the current
        player wins, -1 if the
        opponent wins,
        and 0 if it is a tie.
    """

    player = state.
        get_current_player()
    for i in range(limit):
        end, winner = state.game_end
            ()
    if end:
        break
    action_probs =
        rollout_policy_fn(state)
    max_action = max(
        action_probs, key=
            itemgetter(1))[0]
    state.do_move(max_action)
    else:
        # If no break from the loop,
            issue a warning.
    print("WARNING: rollout_
        reached_move_limit")
    if winner == -1: # tie
    return 0
    else:
    return 1 if winner == player
        else -1

def get_move(self, state):
    """Runs all playouts
        sequentially and returns
        the most visited action.
    state: the current game state

    Return: the selected action
    """
    for n in range(self.
        _n_playout):
    state_copy = copy.deepcopy(
        state)

```

```

        self._playout(state_copy)
        return max(self._root.
            _children.items(),
            key=lambda act_node:
                act_node[1]._n_visits)
            [0]

def update_with_move(self,
    last_move):
    """Step forward in the tree,
        keeping everything we
        already know
        about the subtree.
    """
    if last_move in self._root.
        _children:
        self._root = self._root.
            _children[last_move]
        self._root._parent = None
    else:
        self._root = TreeNode(None,
            1.0)

def __str__(self):
    return "MCTS"

class MCTSPlayer(object):
    """AI player based on MCTS"""
    def __init__(self, c_puct=5,
        n_playout=2000):
        self.mcts = MCTS(
            policy_value_fn, c_puct,
            n_playout)

def set_player_ind(self, p):
        self.player = p

def reset_player(self):
    self.mcts.update_with_move
        (-1)

def get_action(self, board):
    sensible_moves = board.
        availables
    if len(sensible_moves) > 0:
        move = self.mcts.get_move(
            board)
        self.mcts.update_with_move
            (-1)
    return move
    else:
        print("WARNING: the board
            is full")

def __str__(self):
    return "MCTS{}".format(
        self.player)
    # -*- coding: utf-8 -*-
    """
    Monte Carlo Tree Search in
        AlphaGo Zero style, which uses
        a policy-value
        network to guide the tree search
        and evaluate the leaf nodes
    """

import numpy as np
import copy

```

```

def softmax(x):
    probs = np.exp(x - np.max(x))
    probs /= np.sum(probs)
    return probs

class TreeNode(object):
    """A node in the MCTS tree.

    Each node keeps track of its own
        value Q, prior probability P
        , and
    its visit-count-adjusted prior
        score u.
    """

    def __init__(self, parent,
        prior_p):
        self._parent = parent
        self._children = {} # a map
            from action to TreeNode
        self._n_visits = 0
        self._Q = 0
        self._u = 0
        self._P = prior_p

    def expand(self, action_priors):
        """Expand tree by creating new
            children.

        action_priors: a list of tuples
            of actions and their prior
            probability
        according to the policy
            function.
        """
        for action, prob in
            action_priors:
                if action not in self._children:
                    self._children[action] =
                        TreeNode(self, prob)

    def select(self, c_puct):
        """Select action among
            children that gives
            maximum action value Q
            plus bonus u(P).
        Return: A tuple of (action,
            next_node)
        """
        return max(self._children.
            items(),
            key=lambda act_node:
                act_node[1].get_value(
                    c_puct))

    def update(self, leaf_value):
        """Update node values from
            leaf evaluation.
        leaf_value: the value of
            subtree evaluation from
            the current player's
            perspective .
        """
        # Count visit.
        self._n_visits += 1
        # Update Q, a running average
            of values for all visits .
        self._Q += 1.0*(leaf_value -
            self._Q) / self._n_visits

    def update_recursive(self,

```

```

        leaf_value):
            """Like a call to update(), but
               applied recursively for
               all ancestors.
            """
            # If it is not root, this node's
            # parent should be updated
            # first.
            if self._parent:
                self._parent.update_recursive
                    (-leaf_value)
                self.update(leaf_value)

def get_value(self, c_puct):
    """Calculate and return the
       value for this node.
       It is a combination of leaf
       evaluations  $Q$ , and this
       node's prior
       adjusted for its visit count,  $u$ 
       .
        $c\_puct$ : a number in  $(0, \inf)$ 
           controlling the relative
           impact of
       value  $Q$ , and prior probability
            $P$ , on this node's score.
    """
    self._u = (c_puct * self._P *
               np.sqrt(self._parent.
                       _n_visits) / (1 + self.
                       _n_visits))
    return self._Q + self._u

def is_leaf(self):
    """Check if leaf node (i.e. no
       nodes below this have been
       expanded)."""
    return self._children == {}

def is_root(self):
    return self._parent is None

class MCTS(object):
    """An implementation of Monte
       Carlo Tree Search."""

    def __init__(self,
                 policy_value_fn, c_puct=5,
                 n_playout=10000):
        """
        policy_value_fn: a function
            that takes in a board state
            and outputs
            a list of (action, probability)
            tuples and also a score in
             $[-1, 1]$ 
            (i.e. the expected value of the
            end game score from the
            current
            player's perspective) for the
            current player.
        c_puct: a number in  $(0, \inf)$ 
            that controls how quickly
            exploration
            converges to the maximum-
            value policy. A higher value
            means
            relying on the prior more.
        """
        self._root = TreeNode(None,
                                1.0)

```

```

self._policy =
    policy_value_fn
self._c_puct = c_puct
self._n_playout = n_playout

def _playout(self, state):
    """Run a single playout from
        the root to the leaf,
        getting a value at
        the leaf and propagating it
        back through its parents.
        State is modified in-place, so
        a copy must be provided.
    """
    node = self._root
    while(1):
        if node.is_leaf():
            break
        # Greedily select next move.
        action, node = node.select(self
            ._c_puct)
        state.do_move(action)

        # Evaluate the leaf using a
            network which outputs a
            list of
        # (action, probability) tuples
            p and also a score v in
            [-1, 1]
        # for the current player.
        action_probs, leaf_value = self
            ._policy(state)
        # Check for end of game.
        end, winner = state.game_end
            ()
        if not end:
            node.expand(action_probs)
        else:
            # for end state, return the "
                true" leaf_value
            if winner == -1: # tie
                leaf_value = 0.0
            else:
                leaf_value = (
                    1.0 if winner == state.
                        get_current_player()
                    else -1.0
                )

            # Update value and visit count
                of nodes in this traversal.
            node.update_recursive(-
                leaf_value)

def get_move_probs(self, state,
    temp=1e-3):
    """Run all playouts
        sequentially and return the
        available actions and
        their corresponding
        probabilities.
    state: the current game state
    temp: temperature parameter in
        (0, 1] controls the level
        of exploration
    """
    for n in range(self.
        _n_playout):
        state_copy = copy.deepcopy(
            state)
        self._playout(state_copy)

```

```

        # calc the move probabilities
        based on visit counts at
        the root node
        act_visits = [(act, node.
            _n_visits)
            for act, node in self.
                _root._children.items
                    ()]
        acts, visits = zip(*act_visits)
        act_probs = softmax(1.0/temp
            * np.log(np.array(visits) +
                1e-10))

        return acts, act_probs

    def update_with_move(self,
        last_move):
        """Step forward in the tree,
        keeping everything we
        already know
        about the subtree.
        """

        if last_move in self._root.
            _children:
            self._root = self._root.
                _children[last_move]
            self._root._parent = None
        else:
            self._root = TreeNode(None,
                1.0)

    def __str__(self):
        return "MCTS"

class MCTSPlayer(object):
        """AI player based on MCTS"""

        def __init__(self,
            policy_value_function,
            c_puct=5, n_playout=2000,
            is_selfplay=0):
            self.mcts = MCTS(
                policy_value_function,
                c_puct, n_playout)
            self._is_selfplay = is_selfplay

        def set_player_ind(self, p):
            self.player = p

        def reset_player(self):
            self.mcts.update_with_move
                (-1)

        def get_action(self, board, temp
            =1e-3, return_prob=0):
            sensible_moves = board.
                availables
            # the pi vector returned by
            MCTS as in the alphaGo
            Zero paper
            move_probs = np.zeros(board.
                width*board.height)
            if len(sensible_moves) > 0:
                acts, probs = self.mcts.
                    get_move_probs(board,
                        temp)
                move_probs[list(acts)] = probs
            if self._is_selfplay:
                # add Dirichlet Noise for
                exploration (needed for
                self-play training)

```

```

move = np.random.choice(
    acts,
    p=0.75*probs + 0.25*np.
        random.dirichlet(0.3*
            np.ones(len(probs)))
)
# update the root node and
    reuse the search tree
self.mcts.
    update_with_move(
        move)
else:
    # with the default temp=1e
        -3, it is almost
        equivalent
    # to choosing the move with
        the highest prob
    move = np.random.choice(
        acts, p=probs)
    # reset the root node
    self.mcts.update_with_move
        (-1)
# location = board.
    move_to_location(move)
# print("AI move: %d,%d\n"
    % (location[0], location[1]))

if return_prob:
    return move, move_probs
else:
    return move
else:
    print("WARNING: the board
        is full")

def __str__(self):

```

```

return "MCTS_{}".format(
    self.player)

```

其他并不是本工程的主要部分的代码在此加以省略

## 4 训练情况

在训练的过程中，我们很高兴地看到我们地 AI 正在不断地“成长”：我们每训练 50 次就会进行若干次人机对战来确认我们人工智能地的进步。我们的测试员会以十种不同的开局（且这些开局在）与人工智能对战，从而得到当前人工智能的胜率。尽管由于样本数据过小，我们实验得到的数据并不是特别可靠，但它也具有一定的代表性。

- 在 AI 训练了 350 局后，人机首次与测试员胜率相同；
- 在 AI 训练了 600 局后，人机与测试员的胜负比首次达到 3：2
- 在 AI 训练了 1150 局后，人机与测试员的胜负比首次达到 4：1

.....

本次 AI 一共训练 1200 局，最后的胜负比稳定在 4：1 左右。由于人工测试的主观性，所以我们并没有找到相关的论文来证实我们的结果是否有偏差。但通过我们对已训练完成的策略价值网络的分析以及实战数据，

我们认为这一结果还是具有相当的可靠性的。

## 5 结束语

### 5.1 研究总结

1. 本组对棋类博弈人工智能的发展历程进行了了解,学习了博弈树、局面评估函数、卡罗蒙特树等相关算法,对当代的五子棋 AI 进行研究分析,从而确定了本组的研究目标与研究方向。

2. 本组开展了对自我学习算法的探索与实践,将蒙特卡洛树搜索与自我对局相结合,提高了训练数据的有效度。同时对博弈树进行了一定的剪枝,降低了博弈树搜索的时间复杂度,减少了不必要学习的局面。

3. 本组利用卷积神经网络进行模型的构筑,并且在此之上基于策略价值网络的设计,在一定程度上消除了传统方法带来的缺陷与不足。同时利用数据的扩充和反向传播算法的改进,实现了 AI 的自我学习能力。

### 5.2 工作展望

1. 由于本组未使用拥有足够强算力的计算机以及时间问题,本组的人工智能并未达到一个很高的层次,我们的人工智能还存在巨大的提升

空间,本组可能在将来继续进行对该人工智能的训练。

2. 本文对于剪枝算法和蒙特卡洛树的研究并不透彻,其中存在大量未被使用的剪枝算法可以用于提升人工智能的 AI,故在这一方面还需进行更进一步的研究。



## 参考文献

- [1] 李金洪 深度学习之 TensorFlow [M]. 北京: 机械工业出版社, 2018-3
- [2] 陈东焰, 陆畅. 从 AlphaGo 看机器学习 [J]. 科技创新导报, 2020, 17(13): 146-148.
- [3] 百度百科 AlphaGo
- [4] 计算机围棋 AlphaGo 算法对人类围棋算法的影响 [J]. 程思雨, 林锋. 中国科技信息. 2019(02)
- [5] naka. J., The weakest Othello, Takujin Yoshida. Thoroughly dig into the inside of the development! (2019-7-25) [2020-09-01] <https://ai-trend.jp/business-article/interview/othello-cto-interview>
- [6] 菜鸟教程 python 基础教程 <https://www.runoob.com/python/python-tutorial.html>
- [7] 董慧颖; 王杨. 多种搜索算法的五子棋博弈算法研究 [J]. 沈阳理工大学学报, 2017, 2
- [8] 沈大旺. 基于人工智能的五子棋搜索算法 [J]. 产业与科技论坛, 2020, 19(01): 73-74. <http://nooverfit.com/wp/>
- [9] 知乎; 匿名用户 蒙特卡罗树是什么算法? 2017-05-09 <https://www.zhihu.com/question/39916945>
- [10] 刘建平 Pinard 的博客强化学习 (十八) 基于模拟的搜索与蒙特卡罗树搜索 (MCTS) 2019-03-04 <https://www.cnblogs.com/pinard/p/10470571.html>
- [11] yif25 博客蒙特卡罗树搜索 (MCTS) 2018-01-17 16:07 <https://www.cnblogs.com/yifdu25/p/8303462.html>
- [12] 知乎; Xiaohu Zhu 蒙特卡罗树搜索简介 <https://zhuanlan.zhihu.com/p/30316076>
- [13] 基于蒙特卡罗树搜索的计算机围棋博弈研究 [D]. 于永波. 大连海事大学 2015
- [14] 林华. 基于 Self-Play 的五子棋智能博弈机器人 [D]. 浙江大学, 2019.
- [15] 知乎 叉色-xsir 五子棋的谱库设计与实现 <https://zhuanlan.zhihu.com/p/88598278>
- [16] 李昊. 五子棋人机博弈算法优化研究与实现 [D]. 大连海事大学, 2020.

- [17] David Silver, Aja Huang. Mastering the game of Go with deep neural networks and tree search[J]. Nature, 2016, 14(3)
- [18] 扶潇. 基于机器学习的围棋策略网络模型的数据优化 [D]. 北京邮电大学, 2017.
- [19] CSDN BBlue-Sky AlphaGo Zero 详解 [https://blog.csdn.net/qq\\_33813365/article/details/103344018](https://blog.csdn.net/qq_33813365/article/details/103344018)
- [20] 沈大旺. 基于人工智能的五子棋搜索算法 [J]. 产业与科技论坛, 2020, 19(01): 73-74.
- [21] 赵美勇, 宋思睿. 博弈论算法在 AI 中的应用 [J]. 计算机产品与流通, 2019(09): 278.
- [22] 郑健磊, 匡芳君. 基于极小极大值搜索和 Alpha Beta 剪枝算法的五子棋智能博弈算法研究与实现 [J]. 温州大学学报 (自然科学版), 2019, 40(03): 53-62.
- [23] 牛恺泽, 邓鑫. 五子棋人工智能研究与实践 [J]. 数字通信世界, 2019(01): 32-33.
- [24] 孙世文. 五子棋人工智能算法实现研究 [J]. 中国新通信, 2018, 20(23): 143.
- [25] 张效见. 五子棋计算机博弈系统的研究与设计 [D]. 安徽大学, 2017.
- [26] 董慧颖, 王杨. 多种搜索算法的五子棋博弈算法研究 [J]. 沈阳理工大学学报, 2017, 36(02): 39-43+83.
- [27] 邢森. 五子棋智能博弈的研究与设计 [J]. 电脑知识与技术, 2010, 6(13): 3497-3498.
- [28] 张明亮, 李凡长. 一种新的博弈树搜索方法 [J]. 山东大学学报 (工学版), 2009, 39(06): 1-7+23.
- [29] 马占欣, 李亚, 陆玉昌. 用遗传算法解决五子棋博弈问题 [J]. 信阳师范学院学报 (自然科学版), 2007(03): 356-359.
- [30] 游贵荣. 游戏搜索算法中估价函数的构造策略 [J]. 福建商业高等专科学校学报, 2005(06): 82-85+88.