15 AlphaZero

本章首先介绍组合博弈问题（如象棋、围棋等）的概念，然后以五子棋为例介绍 AlphaZero 算法。AlphaZero 算法作为棋类问题的通用算法，在许多挑战巨大的棋类游戏中都取得了超越人类的表现，例如围棋、国际象棋、日本将棋等。该算法结合蒙特卡罗树搜索和深度强化学习自博弈，是人工智能史上的标志性算法。本章分为三个部分：第一部分介绍组合博弈的概念；第二部分介绍蒙特卡罗树搜索算法；第三部分以五子棋为例，详细介绍 AlphaZero 算法。

15.1 简介

AlphaGo Zero (Silver et al., 2017b) 算法在围棋中取得了超越人类冠军的表现，AlphaZero (Silver et al., 2017a, 2018) 算法是 AlphaGo Zero 的通用版本。相比最初击败人类选手的 AlphaGo (Silver et al., 2016) 系列算法 AlphaGo Fan（击败 Fan Hui）、AlphaGo Lee（击败 Lee Sedol）和 AlphaGo Master（击败柯洁），AlphaZero 算法完全基于自博弈（Self-Play）的强化学习从零开始提升。它没有利用人类专家数据进行监督学习，而是直接从随机动作选择开始探索。AlphaZero 有两个关键部分：（1）在自博弈中使用蒙特卡罗树搜索来收集数据；（2）使用深度神经网络拟合数据，并在树搜索过程中用于动作概率和状态价值估计。该算法不仅适用于围棋，还在国际象棋和日本将棋中击败了世界冠军程序，证明了该算法的通用性。本章首先介绍组合博弈（包括围棋、国际象棋、五子棋等）的概念，并给出无禁手五子棋的代码；然后介绍蒙特卡罗树搜索的具体步骤；最后以五子棋为游戏环境演示 AlphaZero 算法的具体细节。为了帮助读者理解，我们提供了五子棋游戏和 AlphaZero 算法的代码链接见读者服务。

15.2 组合博弈

组合博弈理论（CGT）(Albert et al., 2007) 是数学和理论计算机科学的一个分支，通常研究具有完美信息（Perfect Information）的序列化游戏。这类游戏通常具有以下特点：

- 游戏通常包含两个玩家（如围棋、象棋）。有时只包含一个玩家的游戏（如数独、纸牌）也可以看成游戏设计者和玩家之间的组合博弈。包含两个以上玩家的游戏不被视为组合博弈问题，因为游戏中会出现合作等更加复杂的博弈问题 (Browne et al., 2012)。
- 游戏不包含任何影响游戏结果的随机性因素（Chance Factor），如骰子等。
- 游戏给玩家提供完美信息 (Muthoo et al., 1996)，这意味着每个玩家都完全了解之前发生的所有事件。
- 玩家以回合制的方式执行动作，且动作空间和状态空间都是有限的。
- 游戏会在有限步内结束，结果通常为输赢，有些游戏有平局情况。

许多组合博弈问题 (Albert et al., 2007)，包括数独和纸牌等单人游戏，以及如六连棋（Hex）、围棋（Go）和象棋（Chess）等双人游戏，都是计算机科学家需要解决的经典问题。自从 IBM 公司的深蓝系统 (Campbell et al., 2002; Hsu, 1999) 击败了国际象棋大师 Gary Kasparov 之后，围棋成为了人工智能的下一个桥头堡。除此之外，还有很多其他的游戏，如黑白棋（Othello）、亚马逊棋（Amazons）、日本将棋（Shogi）、跳棋（Chinese Checkers）、四子棋（Connect Four）、五子棋（Gomoku）等，吸引了一大批人用计算机来寻找解决方案。

介绍完组合博弈的特点之后，我们以五子棋为例给出一些代码细节。首先从一个空棋盘开始，当有五个相同颜色的棋子连成一条线（水平、垂直或斜线）时，即代表一名玩家获胜，否则为平局。五子棋有各种各样的规则，最常见的规则是无禁手（Freestyle Gomoku）规则或长连禁手（Standard Gomoku）规则。无禁手五子棋只需要有至少五个子连成一条线即可赢得比赛。而长连禁手五子棋需要恰好五个子才代表获胜，任何多于五个棋子都不算获胜。这里我们以无禁手五子棋作为示例。

这里我们进一步简化棋盘大小，使用 3×3 的棋盘作为示例。三个棋子连成一线即表示获胜（我们可以称之为“三子棋”或井字棋），图 15.1 展示了在该棋盘上的动作序列样例。

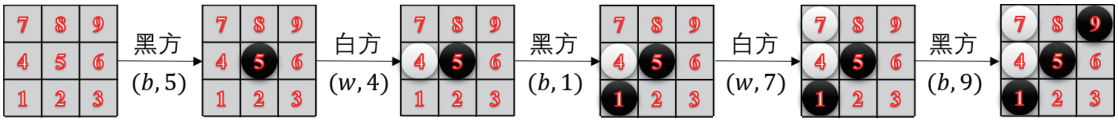


图 15.1 3×3 棋盘上的落子序列示例。“b”代表“黑方玩家”，“w”代表“白方玩家”。$(b, 5)$ 表示黑方玩家在位置 5 处落子。最终黑方玩家获得了游戏胜利（见彩插）

棋盘上的红色数字表示不同的位置，可用于表示每次动作的选择。白色和黑色圆圈是两个玩家的棋子。游戏过程可以表示为一个序列：$((b, 5), (w, 4), (b, 1), (w, 7), (b, 9))$，其中“b”代表“黑

方玩家”，“w”代表“白方玩家”。如图 15.1 最后一个棋盘状态所示，黑方玩家有三个棋子连成一线，这表明黑方赢得了比赛。回忆我们之前提到的定义，这个简化的五子棋（或“三子棋”）满足组合博弈问题的所有特征：游戏包含两个玩家；游戏不包含任何随机因素；游戏提供完美信息；玩家以回合制的方式执行动作；游戏在有限时间步内结束。

这里我们给出无禁手五子棋的代码示例。

定义游戏为 Board 类，并将游戏规则实现成一些函数。我们之前用简化的版本介绍了五子棋的规则，这里通过给变量 n_in_row 赋值为 5 来定义一个标准的五子棋。

```
class Board(object):
    # 定义游戏的类
    def __init__(self, width, height, n_in_row): ... # 初始化函数
    def move_to_location(self, move): ... # 位置表示转换函数
    def location_to_move(self, location): ... # 位置表示转换函数
    def do_move(self, move): ... # 更新每一步走子，并交换对手
    def has_a_winner(self): ... # 判断是否有玩家胜利
    def current_state(self): ... # 生成网络的状态输入
    ...
```

如图 15.1 所示，棋盘上的每个走子位置都用一个数字表示，这样方便在蒙特卡罗树搜索过程中建立树节点。但这种方式不便于辨认是否有五个棋子连成一线。所以我们定义了坐标和数字之间的转换函数，坐标用来判断玩家是否有五个棋子连成一线，数字用来在树搜索中建立树节点。

```
    def move_to_location(self, move):
        # 从数字转换到坐标表示
        # 例如 3 x 3 棋盘:
        # 6 7 8
        # 3 4 5
        # 0 1 2
        # 数字 5 的坐标表示为 (1,2)
        h = move // self.width
        w = move
        return [h, w]
    def location_to_move(self, location):
        # 从坐标转换到数字表示
        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
```

为了判断是否有玩家获胜，需要函数来判断一行或一列或对角线中是否有五个棋子连成一线。函数 has_a_winner() 如下所示：

```
    def has_a_winner(self):
        # 判断是否有玩家获胜，如果有，返回是哪个玩家
        width = self.width
        height = self.height
        states = self.states
        n = self.n_in_row
        # 棋盘上所有棋子的位置
        moved = list(set(range(width * height)) - set(self.availables))
        # 当前所有棋子数量不足以获胜
        if len(moved) < self.n_in_row + 2:
            return False, -1

        for m in moved:
            h, w = self.move_to_location(m)
            player = states[m]
            # 判断是否有水平线
            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
            # 判断是否有竖线
            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
            # 判断是否有斜线
            if (w in range(width - n + 1) 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
            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
```

15.3　蒙特卡罗树搜索

蒙特卡罗树搜索（MCTS）(Browne et al., 2012) 是一种通过动作采样，并根据结果建立搜索树，寻找在给定空间中最优决策的方法。这种方法在组合博弈和规划问题方面产生了革命性的影响，并将围棋等 AI 算法的性能推向了前所未有的高度。

蒙特卡罗树搜索主要包括两部分：树结构和搜索算法。树是一种数据结构（图 15.2），它包含由边连接的节点。一些重要的概念包括根节点、父节点与子节点、叶节点等。树结构最上方的节点称为根节点；一个节点对应的上一级节点称为其父节点，一个节点对应的下一级节点称为其子节点；没有子节点的节点称为叶节点。通常，除状态和动作外，搜索树中的节点还存有被访问次数的统计和奖励的估值。在 AlphaZero 算法中，节点还包含该状态对应的动作概率分布。

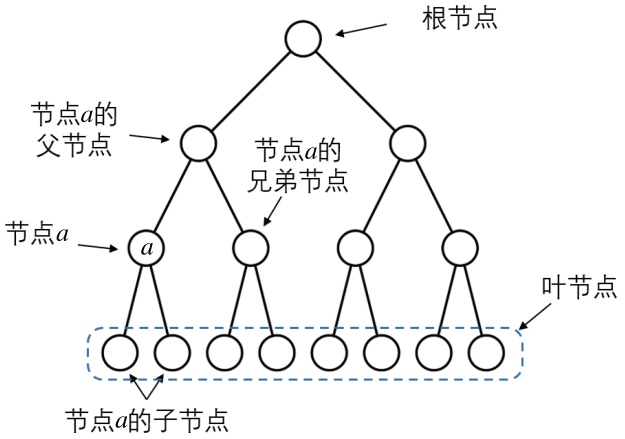


图 15.2　树结构示意图

综上，如图 15.3 所示，AlphaZero 算法的搜索树中，每个节点包含以下信息：

- A：到达该节点所需执行的上一个动作（用以索引其父节点）。
- N：节点被访问次数。初始值为 0，表示该节点未被访问过。
- W：节点的奖励值之和，用以计算平均奖励。初始值设为 0。
- Q：节点的平均奖励值，通过 $\frac{W}{N}$ 计算得到，代表该节点的值函数估计。初始值设为 0。

- P：动作 A 的选取概率。这个值由神经网络输入其父节点的状态得到，存储到该子节点便于索引和计算。

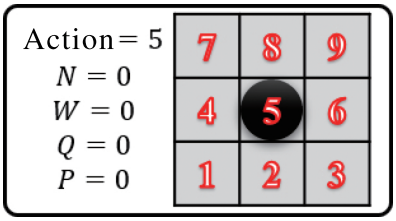


图 15.3 节点包含信息示例。其中位置 5 处有黑方玩家落子，这里动作可表示为 $(b, 5)$，代表黑方玩家（“b”）执行动作 5 并到达该状态。$N = 0$ 表示当前节点访问次数为 0，W 表示该状态的奖励值之和，Q 表示平均奖励，P 表示选择动作 $A = 5$ 的概率。由于当前节点还未被访问过，所有的初始值都设为 0

在继续介绍之前，我们先强调一个关键点。由于游戏中存在两个玩家，所以在建立搜索树时，一棵树里存在两个玩家的视角。节点上的信息要么从黑方玩家的视角进行更新，要么从白方玩家的视角进行更新。例如，在图 15.3 中，此节点表示的棋盘状态只有一个黑方的棋子，所以此时应该轮到白方玩家执行下一步。但是，需要注意的是，这个节点上的信息是从其父节点（即黑方玩家）的角度来存储的。由于该节点是父节点在扩展其子节点的过程中新产生的，因此该节点上的 A、N、W、Q、P 都是由黑方玩家初始化的，并用于黑方视角下的后续更新和使用。所以，只有黑方玩家选择动作 $A = 5$ 才到达该节点，同时初始化当前信息为 $N = 0$、$W = 0$、$Q = 0$、$P = 0$。对每个节点的视角有一个清晰的理解是非常重要的，否则在随后树搜索的过程中执行backup 步时，不易理解整个更新过程。

建立搜索树后，蒙特卡罗树搜索通过启发式的方法探索决策空间，用以估计根节点的动作价值函数 $Q^{\pi}(s, a)$。整个过程可以描述为，从根节点开始一直探索到叶节点，多次重复该过程使得每个动作的奖励估计逐渐精确，从而在搜索树中找到最优动作。不带折扣因子（Discount Factor）的动作价值函数可以表示为 (Couetoux et al., 2011)：

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[\sum_{h=0}^{T-1} P(S_{h+1}|S_h, A_h) R(S_{h+1}|S_h, A_h) | S_0 = s, A_0 = a, A_h = \pi(S_h)\right]. \tag{15.1}$$

其中 $Q^{\pi}(s, a)$ 表示动作价值函数，即在状态 s 执行动作 a 并依策略 π 选择动作，直到终止状态时获得的期望奖励。

通常，树搜索方法有四个步骤：选择（Select），扩展（Expand），模拟（Simulate），回溯（Backup），所有这些步骤都是在搜索树中执行的，真正的棋盘上没有落子。

- 选择：根据某个策略，从根节点开始选择动作，直到到达某个叶节点。
- 扩展：在当前叶节点之后添加子节点。

- 模拟：从当前节点开始，通过某种策略（如随机策略）模拟下棋直到游戏结束，得到结果：胜、负或平局。根据结果获得奖励，通常 $+1$ 代表胜，-1 代表负，0 代表平局。
- 回溯：回溯更新模拟得到的结果，依次回访本轮树搜索中经过的节点，并更新每个节点上的信息。

最常用的树搜索算法是 UCT（Upper Confidence Bound in Tree，树置信上界）算法 (Kocsis et al., 2006)，它很好地解决了树搜索过程中探索与利用（Exploration versus Exploitation）之间的平衡。UCT 算法是 UCB（Upper Confidence Bound，置信上界）算法 (Auer et al., 2002) 在树结构中的扩展。UCB 算法（详见 2.2.2 节）是解决多臂赌博机（Multi-Armed Bandit）问题的经典算法。在多臂赌博机问题中，智能体需要在每个时刻选择一个赌博机并得到对应奖励，其目标为最大化期望奖励。UCB 算法根据以下策略在 t 时刻选择动作：

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]. \tag{15.2}$$

其中 $Q_t(a)$ 是动作值估计，该项增加了优势动作被选到的可能性，即估值越大，动作越倾向被选取（即利用，Exploitation）。后一项平方根式中 $N_t(a)$ 表示动作 a 在前 t 次时间步内被选中的次数，该项增加了动作的探索度，即动作被选中的次数越少，该动作越倾向被选取（即探索，Exploration）。c 是一个正的实数，用来调节探索与利用之间的权重。UCB 算法还有一系列变体，如 UCB1、UCB1-NORMAL、UCB1-TUNED 和 UCB2 等 (Auer et al., 2002)。

UCT 算法是 UCB1 算法在树结构中的实现，该算法选择搜索树中最大 UCT 值对应的动作，UCT 值定义如下：

$$\mathrm{UCT} = \overline{X}_j + C_p\sqrt{\frac{2\ln n}{n_j}}. \tag{15.3}$$

这里，n 是当前节点的访问次数，n_j 是其子节点 j 的访问次数，$C_p > 0$ 是控制探索的权重参数，可以根据具体问题具体设置。平均奖励项 $\overline{X}_j$ 鼓励利用高奖励对应的动作，而平方根项 $\sqrt{\frac{2\ln n}{n_j}}$ 鼓励探索访问次数少的动作。

UCT 算法解决了树搜索中每个状态下对应动作的探索与利用的平衡，并颠覆了许多大规模的强化学习问题，例如六连棋（Hex）、围棋（Go）和雅达利游戏（Atari）等。Levente Kocsis 和 Csaba Szepesvári (Kocsis et al., 2006) 证明了：考虑一个有限状态马尔可夫决策过程（Finite-Horizon MDP），其中奖励在 $[0, 1]$ 之间，状态数为 D，每个状态的动作数为 K。考虑 UCT 算法，令 UCT 的根号项乘以 D，那么期望奖励 $\overline{X}_n$ 的估计偏差与 $O(\frac{\log n}{n})$ 同阶。此外，随着搜索次数的增加，根节点估计错误的概率以多项式速率收敛到零。这表明，随着搜索次数的增加，UCT 算法能够保证树搜索收敛到最优解。

AlphaZero 算法舍弃了模拟步骤，直接用深度神经网络预测结果。因此，AlphaZero 算法包含

三个关键步骤，如图 15.4 所示。

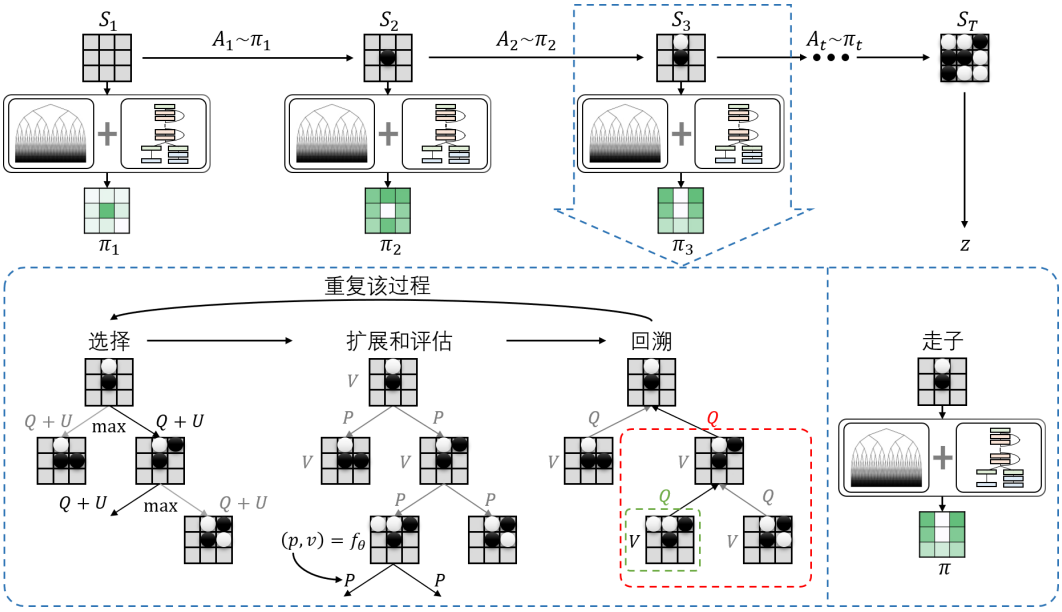


图 15.4 AlphaZero 中的蒙特卡罗树搜索。在每次真正落子之前，树搜索过程都会重复多次。它首先从根节点选择动作直到到达叶节点，然后扩展叶节点并对其估值，最后执行回溯步骤更新节点信息

- 选择: 根据某个策略，从根节点开始选择动作，直到到达某个叶节点。
- 扩展和评估：在当前叶节点之后添加子节点。同时每个动作的选取概率和状态值的估计直接通过策略网络和价值网络预测得到。为了节约资源，通常在不损失算法效力的前提下，会设置一个阈值来判断该节点是否需要扩展。我们的实现省略了这个阈值，每次到达叶节点都进行扩展和评估。
- 回溯：扩展和评估完成之后，回溯更新结果，依次回访本轮树搜索中经过的节点，并更新每个节点上的信息。如果叶节点不是游戏的终止状态，那么游戏无法返回胜负结果，转而由神经网络预测得到。如果叶节点已经到达游戏的终止状态，那么结果直接由游戏给出。

在选择步骤中，动作由公式 $a = \arg\max_a (Q(s,a) + U(s,a))$ 给出。其中 $Q(s,a) = \frac{W}{N}$ 鼓励利用高奖励值对应的动作，$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ 鼓励探索访问次数较少的动作，c_{puct} 平衡探索和利用的权重，在 AlphaZero 算法中该值设为 5。

在扩展和评估步骤中，策略网络输出当前状态下每个动作被选择的概率 $p(s,a)$，价值网络输出当前状态 s 的估值 v。$p(s,a)$ 用于在 select 步骤中计算 $U(s,a)$，其中 $U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$。$v$ 用于在回溯步骤中计算 W，其中 $W(s,a) = W(s,a) + v$。神经网络输出的动作概率和状态值估计开始时可能不准确，但在训练过程中会逐渐变准。

在回溯步骤中，每个节点上的信息被依次更新，其中 $N(s,a) = N(s,a)+1, W(s,a) = W(s,a)+v, Q(s,a) = \frac{W(s,a)}{N(s,a)}$。

部分核心代码如下：

蒙特卡罗树搜索过程定义为类 MCTS，它包含整个树结构和树搜索函数 _playout()：

```
class MCTS(object):
    # 蒙特卡罗树搜索类
    def__init__(self, policy_value_fn,action_fc,evaluation_fc, is_selfplay,c_puct,
        n_playout): ... # 初始化函数
    def _playout(self, state): ... # 树搜索过程
```

树中的节点定义为类 TreeNode，其中包括前述的三个关键步骤：选择，扩展和评估，回溯。

```
    class TreeNode(object):
        # 树节点类
        # 每个节点保存值估计，动作选择概率等相关参数
        def __init__(self, parent, prior_p): ... # 初始化函数
        def select(self, c_puct): ... # 选择动作
        def expand(self, action_priors, add_noise): ...# 扩展节点并评估当前状态和每个动作
        def update(self, move): ... # 回溯更新节点
        ...
```

函数 select() 对应选择步骤：

```
    def select(self, c_puct):
        # 选择最大 UCT 值对应的动作，返回动作和下一个节点
        return max(self._children.items(),
                   key=lambda act_node: act_node[1].get_value(c_puct))
```

函数 expand() 对应扩展和评估步骤。我们在每个节点都添加了狄利克雷噪声，增加随机探索：

```
    def expand(self, action_priors, add_noise):
        # 扩展新节点
        # action_priors 是策略网络输出的动作及其对应的概率值
        if add_noise:
            action_priors = list(action_priors)
            length = len(action_priors)
            dirichlet_noise = np.random.dirichlet(0.3 * np.ones(length))
```

```
            for i in range(length):
                if action_priors[i][0] not in self._children:
                    self._children[action_priors[i][0]] = TreeNode(self,
                        0.75 * action_priors[i][1] + 0.25 * dirichlet_noise[i])
        else:
            for action, prob in action_priors:
                if action not in self._children:
                    self._children[action] = TreeNode(self, prob)
```

函数 update_recursive() 对应回溯步骤：

```
    def update_recursive(self, leaf_value):
        # 递归更新所有节点
        # 若该节点不是根节点，则递归更新
        if self._parent:
            # 通过传递取反后的值来改变玩家的视角
            self._parent.update_recursive(-leaf_value)
        self.update(leaf_value)

    def update(self, leaf_value):
        # 更新节点信息
        self._n_visits += 1
        # 更新访问次数
        self._Q += 1.0 * (leaf_value - self._Q) / self._n_visits
        # 更新值估计：(v-Q)/(n+1)+Q = (v-Q+(n+1)*Q)/(n+1)=(v+n*Q)/(n+1)
```

蒙特卡罗树搜索类 MCTS 调用树搜索函数 _playout() 依次执行三个步骤：选择，扩展和评估，回溯。

```
    def _playout(self, state):
        # 执行一次树搜索过程
        node = self._root
        # 选择
        while(1):
            if node.is_leaf():
                break
            action, node = node.select(self._c_puct)
            state.do_move(action)
        # 扩展和评估
```

```
        action_probs, leaf_value =
            self._policy_value_fn(state,self._action_fc,self._evaluation_fc)
        end, winner = state.game_end()
        if not end:
            node.expand(action_probs,add_noise=self._is_selfplay)
        else:
            if winner == -1: # draw
                leaf_value = 0.0
            else:
                leaf_value = (
                    1.0 if winner == state.get_current_player() else -1.0
                )
        # 回溯
        node.update_recursive(-leaf_value)
```

15.4　AlphaZero：棋类游戏的通用算法

一般来说，AlphaZero 算法适用于各种组合博弈游戏，如围棋、国际象棋、日本将棋等。这里，我们以 15.2 节中提到的无禁手五子棋作为例子，介绍 AlphaZero 算法的细节。因为游戏本身不是重点，五子棋这样一个规则简单的回合制游戏非常适合作为例子。进一步，我们简化棋盘大小为 3×3，如前所述，三个棋子连成一线表示获胜。另外，由于 AlphaZero 算法是 AlphaGo Zero 算法的加强版，这两种算法非常相似。我们的实现同时参考了这两种算法。

为了让读者更好理解该算法，本节将演示 AlphaZero 算法的详细流程。整个算法可分为两部分：（1）采用蒙特卡罗树搜索的自博弈强化方法收集数据；（2）利用深度神经网络拟合数据并用于蒙特卡罗树搜索中。整个过程如图 15.5 所示。

首先，我们演示蒙特卡罗树搜索收集数据。为了用相对较短的篇幅演示树搜索过程直到游戏的终止状态，我们假定游戏从图 15.6 所示的状态开始（通常游戏是从一个空棋盘开始的）。此时，轮到白方玩家执行动作。

我们从这个状态开始构建树结构，依次执行前述三个树搜索步骤：*选择*，*扩展和评估*，*回溯*。此时树中只有一个节点，由于它在树的顶部，所以是根节点，又因为它没有子节点，所以它也是叶节点。这意味着我们已经到达了一个叶节点，相当于已经完成了*选择*步骤。因此，接下来执行第二个步骤：*扩展和评估*。图 15.7 展示了节点扩展的过程，该节点的所有子节点被展开，同时策略网络以该节点状态作为输入，给出了每个动作被选择的概率。

最后一步是*回溯*。由于当前节点是根节点，我们不需要回溯 W 和 Q（用于判断树搜索是否应该到达该节点），只需更新访问次数 N。将 $N = 0$ 更新为 $N = 1$，本次树搜索过程完成。

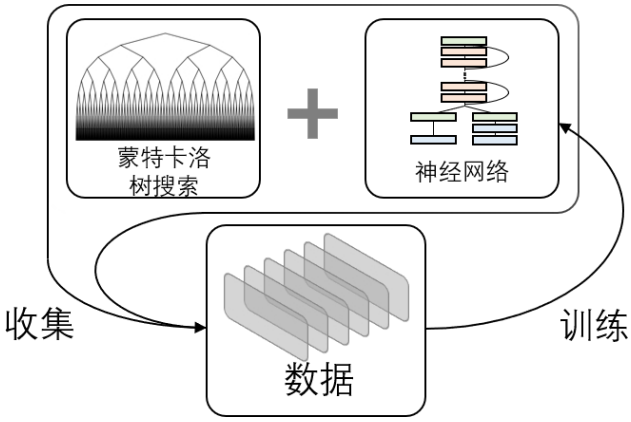


图 15.5 算法流程。在 AlphaZero 算法中，蒙特卡罗树搜索、数据及神经网络形成了一个循环。蒙特卡罗树搜索结合神经网络用于生成数据，生成的数据用于提升网络预测精度。网络预测越精确，蒙特卡罗树搜索生成的数据质量越高；数据质量越高，训练的网络预测越精确；整个过程形成良性循环

图 15.6 棋盘状态。棋盘大小为 3 × 3。在该状态下，轮到白方玩家执行动作

每次重新执行树搜索，我们都将从根节点开始。如图 15.8 所示，第二次树搜索过程也将从根节点开始。这一次，根节点下存在子节点，这意味该节点不是叶节点。动作由公式 $a = \arg\max_a(Q(s,a)+U(s,a))$, $Q(s,a)=\frac{W}{N}$, $U(s,a)=c_{\text{puct}}P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ 给出。这里白方玩家选择动作 $A=2(w,2)$，并到达新节点。这个新节点为叶节点，且此时，轮到黑方玩家选择动作。

我们对这个叶节点进行*扩展和评估*。与第一次相同：所有可行的动作都被扩展，每个动作的概率由策略网络给出。

现在轮到*回溯*操作了。此时树里有两个节点，我们首先更新当前节点，然后更新前一个节点。这两个节点的更新遵循相同的方式：$N(s,a)=N(s,a)+1, W(s,a)=W(s,a)+v(s), Q(s,a)=\frac{W(s,a)}{N(s,a)}$。值得注意的是，树中有两个视角：黑方视角和白方视角。我们需要注意更新的视角，并且总是以当前玩家的视角更新值。例如，在图 15.10 中，价值网络的估值 $v(s)=-0.1$，这是从黑方玩家的角度来看的。当更新属于白方玩家的信息时需要取反，即 $v(s)=-0.1$。所以，我们得到 $N=1, W=0.1, Q=0.1$。

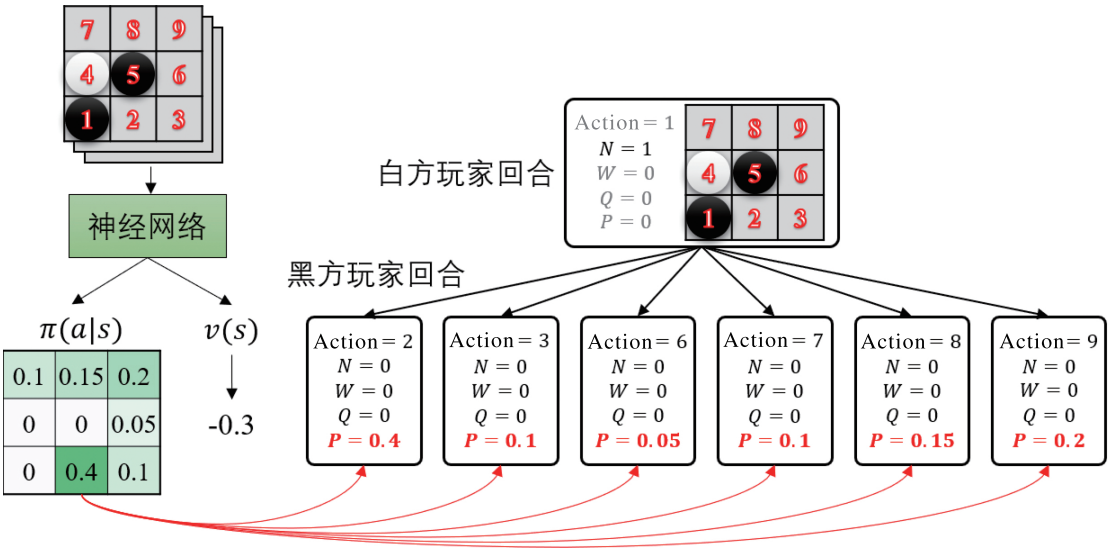
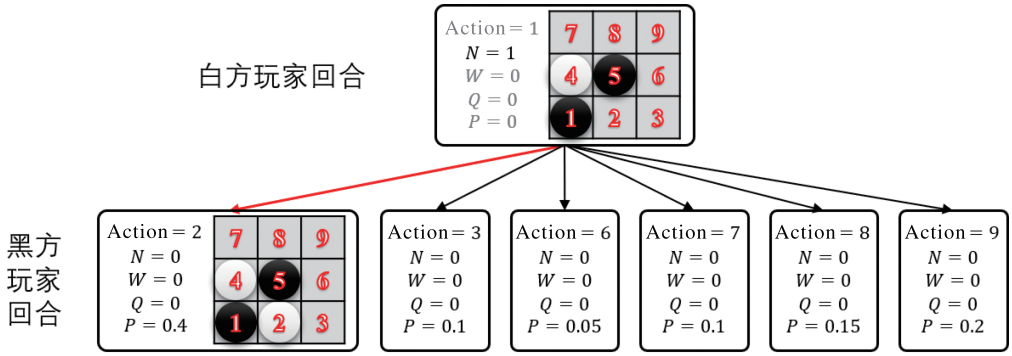


图 15.7　根节点处的扩展和评估。所有可行动作的节点都被扩展，神经网络给出相应的概率值 $\pi(a|s)$



$$a = \operatorname{argmax}_a(Q(s,a) + U(s,a))$$

$$\text{探索}: U(s,a) = c_{\text{puct}}\ P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

$$\text{利用}: Q(s,a) = \frac{W}{N}$$

图 15.8　根节点处的选择。白方玩家选择 $A = 2\ (w, 2)$ 并到达叶节点。此时轮到黑方玩家选择动作

然后我们返回到它的父节点。和之前一样，由于当前状态的节点是根节点，我们不需要回溯更新 W 和 Q，只需要更新访问次数 N。所以，令 $N = 2$，第二次树搜索过程完成。如图 15.11 所示。

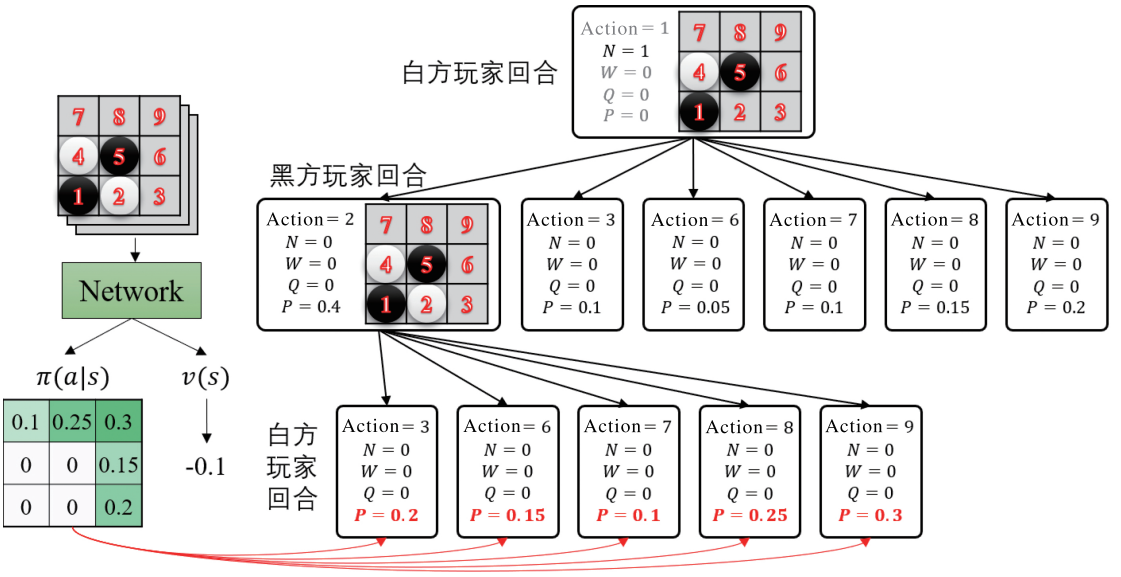


图 15.9 新节点处的扩展和评估。所有可行的动作都被扩展，神经网络给出相应的概率值 $\pi(a|s)$

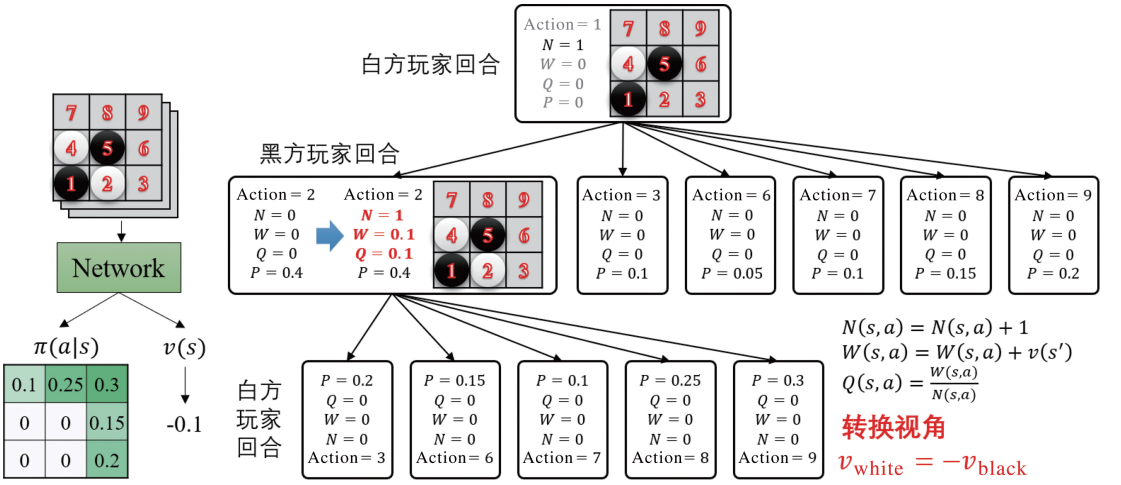


图 15.10 新节点处的回溯。当前节点的信息被更新，Q 值从白方视角进行更新：$N = 1, W = 0.1, Q = 0.1$

第三次树搜索过程也从根节点开始。根据公式 $a = \arg\max_a(Q(s,a) + U(s,a))$ 和当前树中的信息，白方玩家选择动作 2 $(w, 2)$，黑方玩家选择动作 9 $(b, 9)$。如图 15.12 所示，经过选择步骤后，游戏到达了终止状态。这次对于扩展和评估步骤，节点将不会被扩展，同时可以直接从游戏中获取价值 v。因此，价值网络不会用来估计状态价值，策略网络也不会输出动作概率。

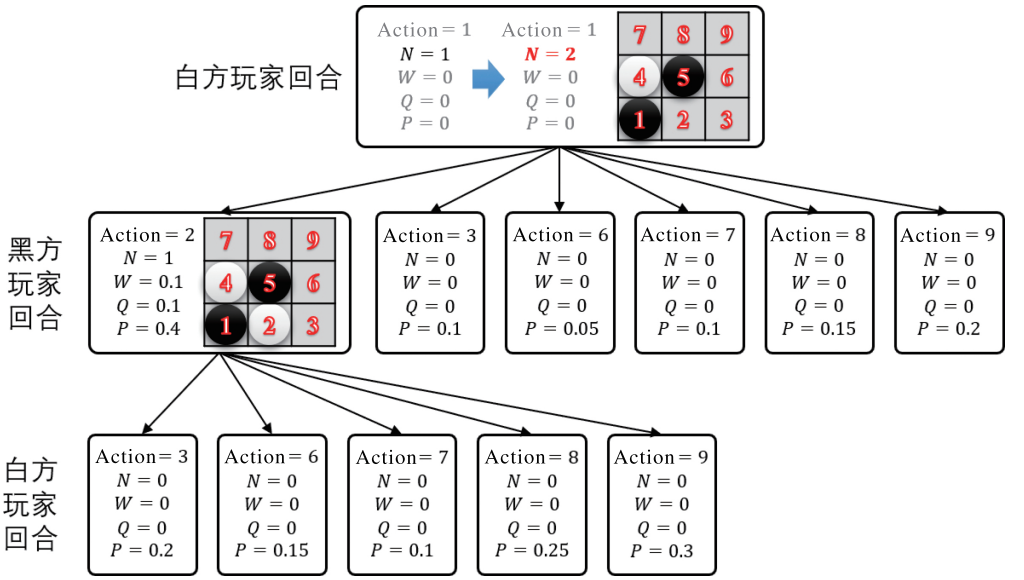


图 15.11　根节点处的回溯。N 更新为 2，W 和 Q 不需要更新

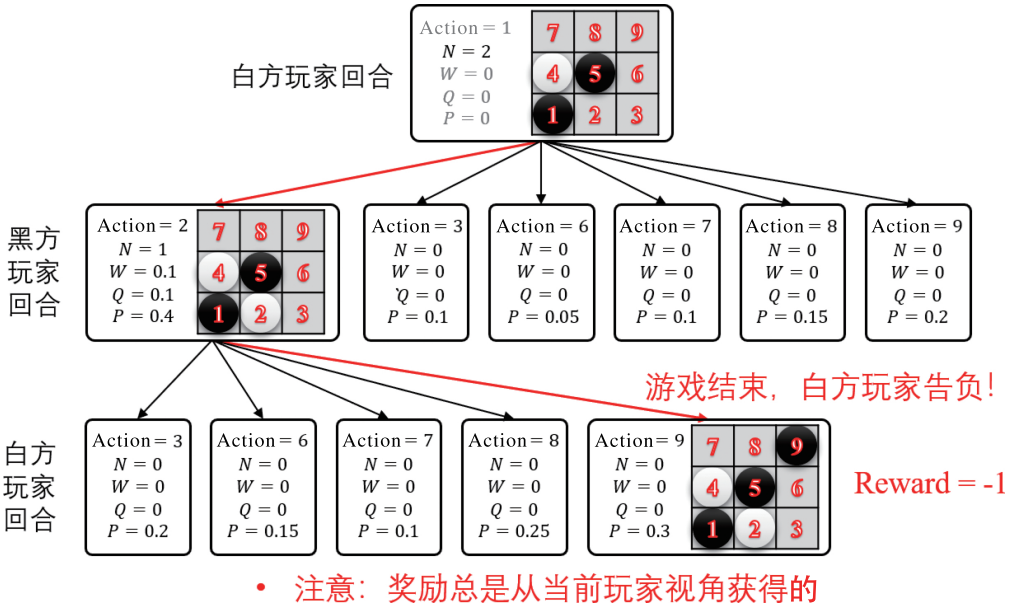


图 15.12　终节点处的扩展和评估。由于游戏在该节点结束，因此不会扩展任何节点，并且可以直接从游戏中获得奖励。所以，这里不会使用策略网络和价值网络

接下来是回溯，且轨迹上有三个节点。如前所述，节点从叶节点递归更新直到根节点，其中 $N(s,a) = N(s,a)+1, W(s,a) = W(s,a)+v(s), Q(s,a) = \frac{W(s,a)}{N(s,a)}$。此外，还应该切换每个节点的

视角，这意味着 $v_{\text{white}} = -v_{\text{black}}$。在这个游戏中，黑方玩家选择动作 9 $(b, 9)$ 并到达一个新节点。现在本该轮到白方玩家选择动作，但遗憾的是游戏在此结束了，白方玩家输掉了游戏。所以奖励值 reward $= -1$ 是从白方玩家的视角来看的，也就是说 $v_{\text{white}} = -1$。当我们更新这个节点上的信息时，如前所述，这些信息是被黑方玩家用来选择动作 $A = 9$ 并到达这个节点，所以这个节点的值应该是 $v_{\text{black}} = -v_{\text{white}} = 1$，其他信息同理，有 $N = 1, W = 1, Q = 1$。剩余两个节点的信息也以同样的方式更新。

在完成回溯步骤之后，树结构如图 15.13 所示。根节点已被访问三次，并且每个被访问过的节点信息都已更新。

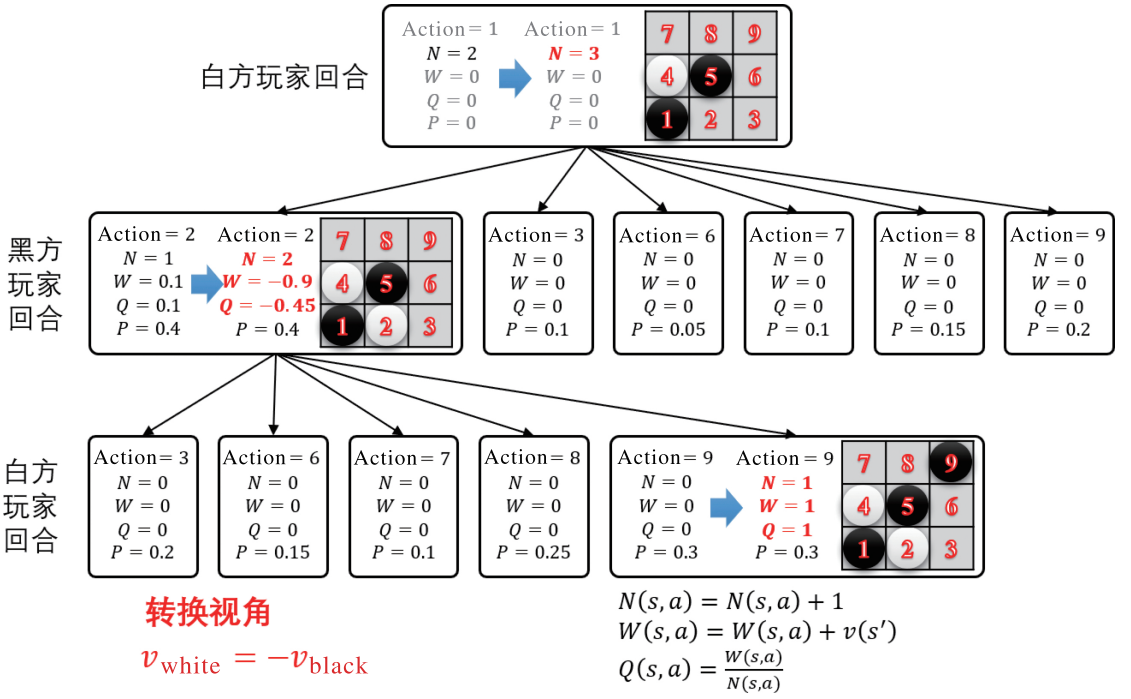


图 15.13 回溯步骤之后的树结构。在第三次树搜索过程中，回溯步骤递归地更新三个被访问节点的信息。由于两个玩家在同一树结构中，且 $v_{\text{white}} = -v_{\text{black}}$，所以需要注意从正确的视角更新信息

我们已经演示了三次蒙特卡罗树搜索的迭代过程。经过 400 次搜索后（在 AlphaGo Zero 算法中，搜索次数是 1600；在 AlphaZero 算法中，搜索次数是 800，如图 15.14 所示），树结构变得更大，且估值更加精确。

经过树搜索过程之后，可以在真正的棋盘上走子了。动作的选取通过计算每个动作的访问次数并归一化为概率进行选择，而不是直接通过策略网络输出动作概率：$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{N(s)^{1/\tau - 1}} = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$，其中 $\tau \to 0$ 是温度参数，$b \in A$ 表示状态 s 下的可行动作。这里选择的动作是 9

$(w, 9)$，如图 15.15 所示。

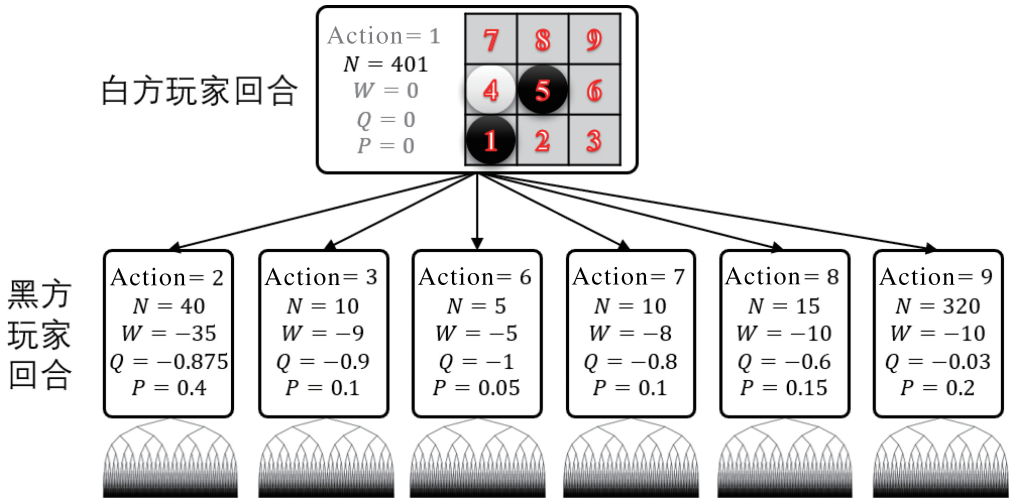


图 15.14　经过 400 次搜索的树结构。由于第一次搜索是从扩展和评估步骤开始的，并没有选择子节点，因此其子节点的访问次数之和为 400，根节点的访问次数为 401。这个细节并不影响算法思想

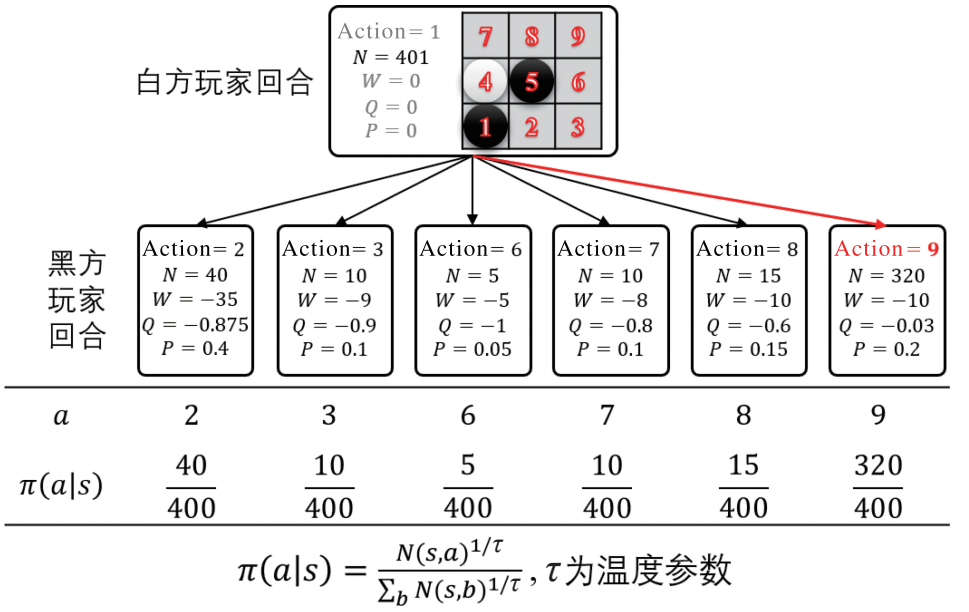


图 15.15　在真正的棋盘上走子。经过 400 次搜索后，根据 $\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$ 选择动作。这里白方玩家选择动作 9 $(w, 9)$

温度参数用来控制探索度。若 $\tau = 1$，动作的选择概率和访问次数成正比，则这种方式探索度高，可以确保数据收集的多样性。若 $\tau \to 0$，则探索度低，此时倾向于选择访问次数最大的动作。在 AlphaZero 和 AlphaGo Zero 算法中，当执行自博弈过程收集数据时，前 30 步（在我们的实现中为 12 步）的温度参数设为 $\tau = 1$，其余部分设置为 $\tau \to 0$。当与真正对手下棋时，温度参数始终设置为 $\tau \to 0$，即每次都选择最优动作。

至此，位置 9 处已经放置了白方棋子，因此树中的根节点将被更改。如图 15.16 所示，蒙特卡罗树搜索将从新的根节点继续。其他兄弟节点及其父节点将被剪枝丢弃以节省内存。

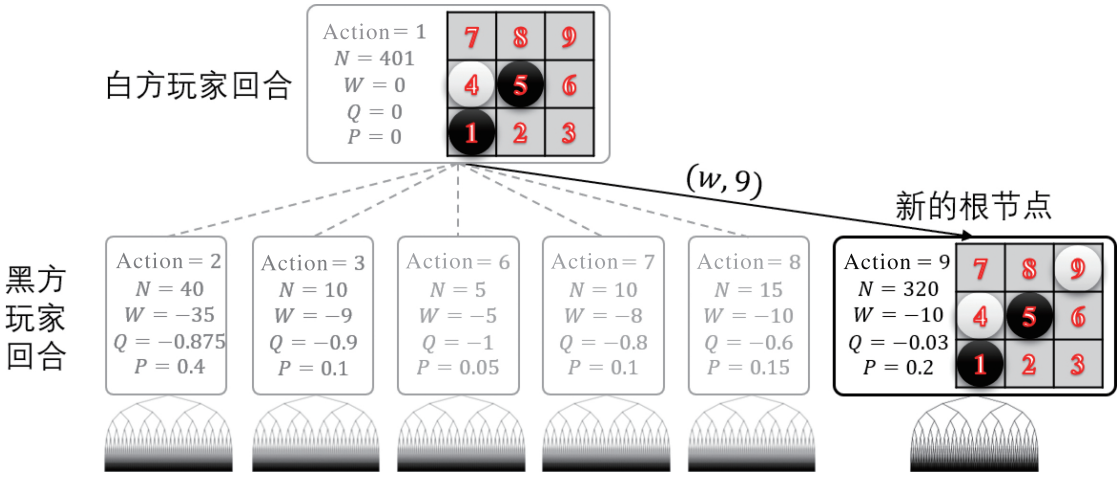


图 15.16　新的根节点。新根节点下的节点将被保留，其他节点将被丢弃

整个过程一直重复下去，直到一局游戏结束。我们得到数据和结果如图 15.17 所示。

每个动作的概率计算方式为：$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}, \tau = 1$。需要注意，这里的概率通过访问次数计算，这是蒙特卡罗树搜索自博弈过程和神经网络训练相结合的关键点。由于该局游戏的结果是平局，这里所有数据的标签都是 0（图 15.18）。

现在我们已经有了蒙特卡罗树搜索生成的数据，下一步就是利用深度神经网络进行训练。在训练过程中，首先将数据转换成堆叠的特征层。每个特征层只包含 0-1 值用以表示玩家的落子，其中一组特征层表示当前玩家的落子，另一组特征层表示对手的落子。这些特征层按照历史动作序列顺序堆叠。然后，我们用与 AlphaGo Zero 算法相同的数据增强方法对数据进行扩展：由于围棋和五子棋的规则都不受旋转和镜像翻转的影响，因此在训练之前，我们将数据做旋转和镜像翻转增强。在蒙特卡罗树搜索过程中，棋盘状态被随机旋转或镜像翻转，然后再用神经网络进行预测，从而可以在一定程度上减小方差。然而在 AlphaZero 算法中，由于某些游戏规则不具有旋转和镜像翻转不变性，因此 AlphaZero 没有使用该技巧。言归正传，随着收集的数据越来越多，不断训练的网络会得到更加精确的估计。

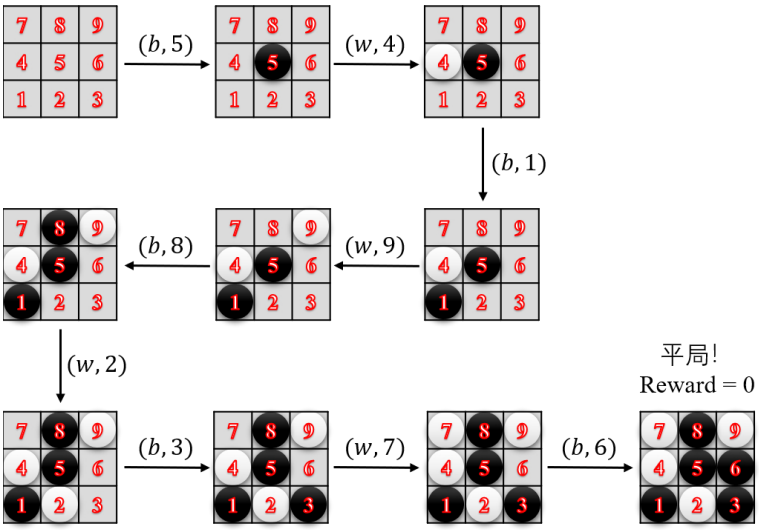


图 15.17　棋谱数据。该局游戏的所有状态都将被保存，并赋予动作概率 $\pi(a|s)$ 和状态值 $v(s)$

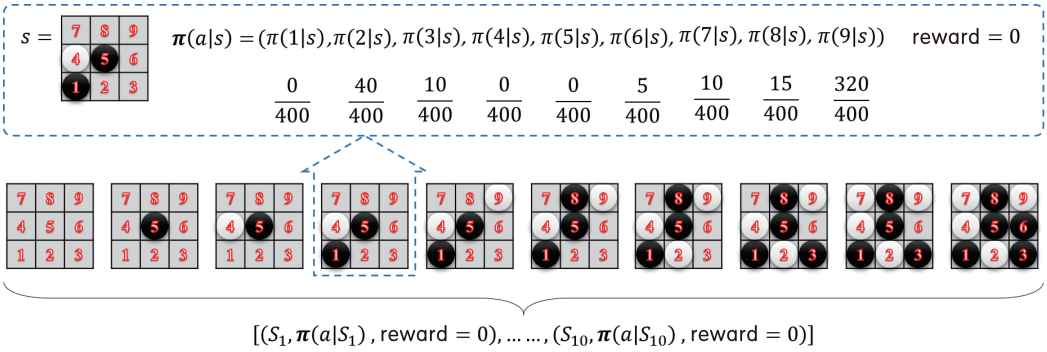


图 15.18　带标签的数据。动作的概率根据 $\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$ 计算，标签 $v(s)$ 来自游戏的结果：1 表示胜利，-1 表示失败，0 表示平局

我们使用 `ResNet` (He et al., 2016) 作为网络结构（图 15.19），这和 AlphaGo Zero 算法相同。网络的输入是前述构造的状态特征，输出是动作概率和状态值。网络可以表示为 $(\boldsymbol{p}, v) = f_\theta(s)$，数据为 $(s, \boldsymbol{\pi}, r)$，其中 $\boldsymbol{p}, \boldsymbol{\pi}$ 为列向量。损失函数 l 由动作分布的交叉熵损失、状态值的均方误差和参数的 L2 正则化组成。具体公式为 $l = (r - v)^2 - \boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p} + c\|\theta\|^2$，其中参数 c 调节正则化权重。

此外，我们介绍一些关于模型更新的细节。在 AlphaGo Zero 算法中，新模型将和当前的最优模型对打 400 局，如果新模型胜率超过 55%，那么它将替换掉之前的模型成为当前的最优模型，即对模型有一个评估的过程。相比之下，在 AlphaZero 算法的版本中，它不与之前的模型进行对打，而是直接不断更新模型参数，这些都是可行的方法。我们的版本和 AlphaGo Zero 的方式相

同，以使训练过程更加稳定。此外，如果想更快地训练模型，可以使用多进程并行收集数据，甚至采用原论文异步树搜索的方式。图 15.20 展示了并行的训练方式，多个进程同时从最优模型中源源不断生成自博弈数据，收集到最新的自博弈数据用来训练神经网络，最新训练的模型和最优模型进行不断评估（AlphaGo Zero 的方式），所有这些进程都并行执行。

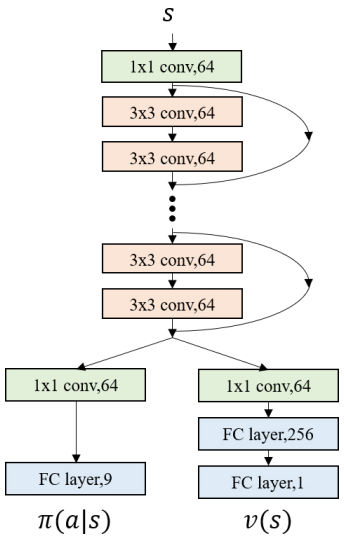


图 15.19 网络结构。结构与 AlphaGo Zero 算法相同。`ResNet` 作为主干，两个头分别输出概率分布和状态估值

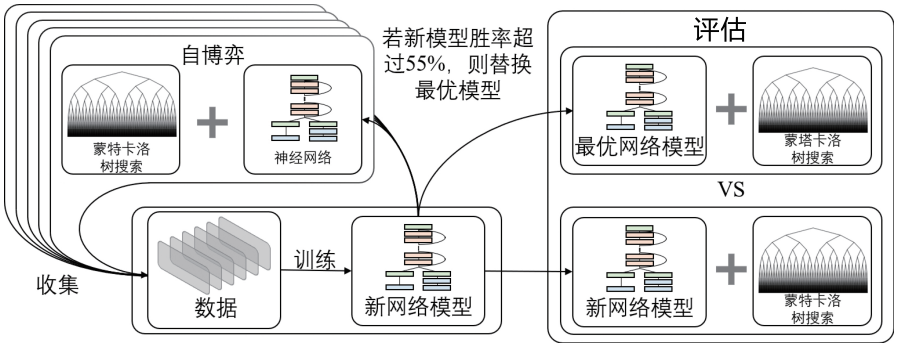


图 15.20 并行训练框架

随着新数据源源不断地生成，神经网络不断迭代训练，得到更准确的估值，一个强大的五子棋 AI 就生成了。

我们最终在 11×11 的棋盘上通过多进程并行的方式训练了无禁手规则的五子棋 AI，表 15.1 列出了一些具体参数。随后我们在 15×15 的棋盘上同样成功训练了一个模型，这表明了 AlphaZero

算法的通用性和稳定性。

表 15.1　参数对比

参数	五子棋	**AlphaGo Zero**	**AlphaZero**
c_{puct}	5	5	5
MCTS times	400	1600	800
residual blocks	19	19/39	19/39
batch size	512	2048	4096
learning rate	0.001	annealed	annealed
optimizer	Adam	SGD with momentum	SGD with momentum
Dirichlet noise	0.3	0.03	0.03
weight of noise	0.25	0.25	0.25
$\tau = 1$ for the first n moves	12	30	30

参考文献

ALBERT M, NOWAKOWSKI R, WOLFE D, 2007. Lessons in play: an introduction to combinatorial game theory[M]. CRC Press.

AUER P, CESA-BIANCHI N, FISCHER P, 2002. Finite-time analysis of the multiarmed bandit problem[J]. Machine learning, 47(2-3): 235-256.

BROWNE C B, POWLEY E, WHITEHOUSE D, et al., 2012. A survey of monte carlo tree search methods[J]. IEEE Transactions on Computational Intelligence & Ai in Games, 4(1): 1-43.

CAMPBELL M, HOANE JR A J, HSU F H, 2002. Deep blue[J]. Artificial intelligence.

COUETOUX A, MILONE M, BRENDEL M, et al., 2011. Continuous rapid action value estimates[C]// Asian Conference on Machine Learning. 19-31.

HE K, ZHANG X, REN S, et al., 2016. Deep residual learning for image recognition[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 770-778.

HSU F H, 1999. Ibm's deep blue chess grandmaster chips[J]. IEEE Micro, 19(2): 70-81.

KOCSIS L, SZEPESVÁRI C, 2006. Bandit based monte-carlo planning[C]//European conference on machine learning. Springer: 282-293.

MUTHOO A, OSBORNE M J, RUBINSTEIN A, 1996. A course in game theory.[J]. Economica, 63 (249): 164-165.

SILVER D, HUANG A, MADDISON C J, et al., 2016. Mastering the game of go with deep neural networks and tree search[J]. Nature.

SILVER D, HUBERT T, SCHRITTWIESER J, et al., 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm[J]. arXiv preprint arXiv:1712.01815.

SILVER D, SCHRITTWIESER J, SIMONYAN K, et al., 2017b. Mastering the game of go without human knowledge[J]. Nature, 550(7676): 354.

SILVER D, HUBERT T, SCHRITTWIESER J, et al., 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play[J]. Science, 362(6419): 1140-1144.