基于蒙特卡洛搜索的自学习五子棋

第 28 组　　员龙飞　　唐小龙　　戴劲

摘要

很长一段时间以来，学术界普遍认为机器在围棋上达到人类高手的水平是不现实的。令人惊讶的是，2016 **年** 3 **月，谷歌** DeepMind **发明的一种名为** AlphaGo **的算法在围棋比赛中以** 4 **比** 1 **击败了韩国世界冠军李世石，证明了怀疑论者都是错的。大约一年后，**Alpha Go **的下一代** Alpha Go Zero **以** 100 **比** 0 **的比分击败了它的前辈，这标志着该算法已经超越人类能够达到的极限。**

Alpha Go/Zero **系统是将几种方法组合成一个伟大的工程作品。**Alpha Go/Zero **的核心组成部分是**:

- **蒙特卡洛树搜索** (Monte Carlo Tree Search)
- **残差卷积神经网络** (Residual Convolutional Neural Networks): **策略和价值网络用于游戏评估和移动的先验概率估计**
- **强化学习** (Reinforcement learning)**：通过自我对弈** (self-play) **训练神经网络。**

在本次大作业中，我们首先完成了 AIGAMING **上面的四子棋项目，并且实现了登顶和** 100% **胜率。然后我们利用神经网络实现了一个五子棋的** AlphaZero **版本，它在经过** 3000 **局左右的训练后可以对基于朴素的蒙特卡洛树搜索的五子棋** bot **达到** 90% **的胜率。**

1. 屏风四子棋

我们首先用 $\alpha - \beta$ 剪枝算法完成了 AIGaming 上面的 four in a row 游戏。

1.1. 游戏规则

- 棋盘是 6×7 的棋盘格。
- 双方必须轮流把一枚己方棋子投入开口，让棋子因为重力下落在棋盘底部或者其它棋子上。
- 当己方棋子以纵、横、斜方向连成一线时获胜。
- 棋盘满棋且没有任何连成四子的情形时达成平手。

1.2. 算法:alpha-beta Pruning

```
function ALPHA-BETA-SEARCH(state) returns an action
    v <- MAX-VALUE(state, -inf, +inf)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, alpha, beta) returns a
     utility value
    if TERMINAL-TEST(state) then return UTILITY(
         state)
    v <- -inf
    for each a in ACTIONS(state) do
        v <- MAX(v, MIN-VALUE(RESULT(s, a), alpha,
             beta))
        if v >= beta then return v
        alpha <- MAX(alpha, v)
    return v

function MIN-VALUE(state, alpha, beta) returns a
     utility value
    if TERMINAL-TEST(state) then return UTILITY(
         state)
    v <- +inf
    for each a in ACTIONS(state) do
        v <- MIN(v, MAX-VALUE(RESULT(s, a), alpha,
             beta))
        if v <= alpha then return v
        beta <- MIN(beta, v)
    return v
```

1.3. 使用评估函数改进

$Min-Max$ 算法生成整个博弈的搜索空间，而 $\alpha-\beta$ 剪枝算法允许我们剪掉其中的一大部分。但是朴素的 $\alpha-\beta$ 剪枝算法仍然要搜索到叶子节点，从算法的复杂度 $O(b^m)$ 来看显然是不现实的。所以我们应该尽早截断搜索，将启发式评估函数用于对非叶子节点进行评估，另一方面我们还需要截断测试来判断什么时候运用启发式函数:

H-MINIMAX(s, d) =

$$\begin{cases} \text{EVAL}(s) & \text{如果 CUTOFF-TEST}(s,d)\text{为真} \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d{+}1) & s \text{ 为 MAX 结点} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d{+}1) & s \text{ 为 MIN 结点} \end{cases}$$

图 1.

我们设计评估函数时基于如下思路:

- $EVAL(s) = \sum_{i=1}^{4} w_i \times f_i(s)$。
- 随着连子的增加，权重 w_i 逐渐增加。
- 当连子两边没有敌方棋子或边界时 $f_i(s)$ 增加。
- 由于重力，所以我们可以定义当前一列最上方的棋子到连子的距离，如果距离小，$f_i(s)$ 较大。

1.4. 效果展示

我们用写好的算法与网站上的 bot 进行了一百局对战，胜率达到了 100%.

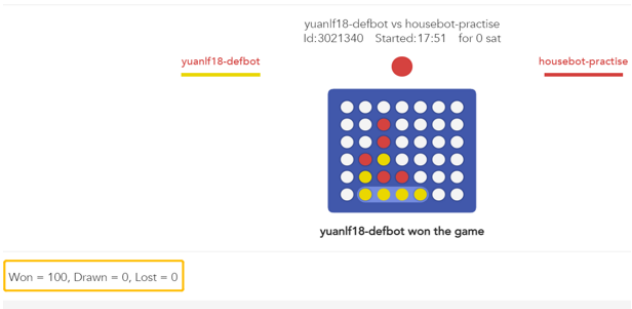

图 2. 测试结果：胜率 100%.

我们在 AIGaming 上的排名为第一名。

图 3. 排名为第一名

现在我们实现了基于 $\alpha-\beta$ 剪枝的智能体，并且运行结果证明它的性能十分优秀。接下来我们希望实现一个基于残差卷积神经网络训练的蒙特卡洛树搜索的智能体。

在第二部分，我们先简述蒙特卡洛树搜索和神经网络学习的原理，然后分析它的性能。

2. 基于 UCB 的 Monte-Carlo Tree Search

2.1. 基本思想

蒙特卡洛树搜索（简称 MCTS）是 Rémi Coulom 在 2006 年在它的围棋人机对战引擎「Crazy Stone」中首次发明并使用的的，并且取得了很好的效果。

MCTS 合并了两个重要的思想:

- 通过 Rollouts 完成游戏.
- Selective search.

图 5. Selective search: there is a well-known exploitation-exploration dilemma.
Exploitation: focus on more promising moves.
Exploration: focus on more uncertain node.

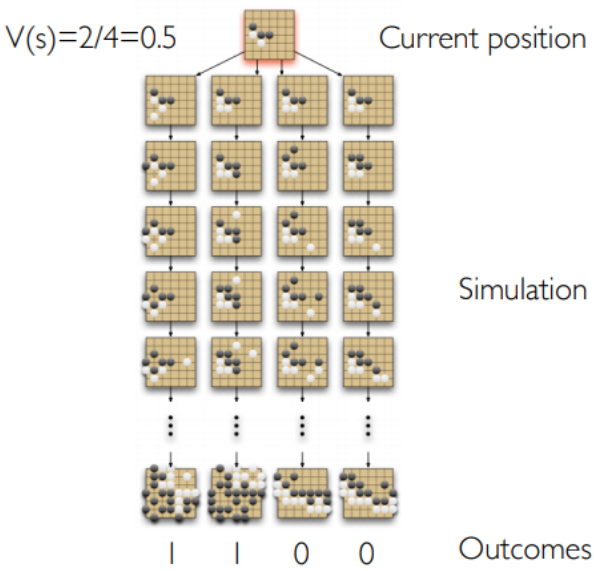

图 4. MCTS 通过 rollouts 评估当前局面。Do N rollouts from each child of the root, record fraction of wins, and then pick the move that gives the best outcome by this metric.

2.2. Upper Confidence Bound

UCB 是解决上述 exploitation-exploration dilemma 的一种有效的算法。

UCBI formula:

$$v_i + C \times \sqrt{\frac{N}{n_i}} \tag{1}$$

- v_i: value estimation.
- C: tunable parameter.
- N: total number of trials

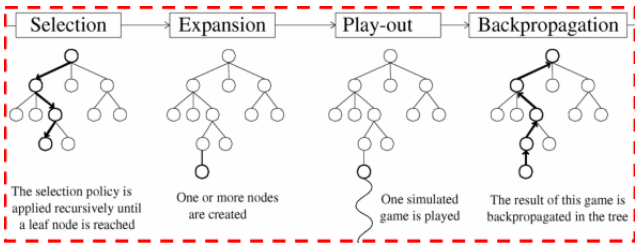

图 6. Monte-Carlo 算法的步骤 1。

- n_i: number trials for selection i.

从公式中我们可以看出 exploration 和 exploitation 之间的折衷:

1. Exploit: v_i prefers higer payoff selection.
2. Explore: prefer less played selection.

2.3. 算法流程

```
function Monte-Carlo-Tree-Search(state) returns an
    action
    tree = Node(state)
    while Is-Time-Remaining() do
        MCTS-sample(tree)
    return the move in Actions(state) whose node has
        highest numbers of playouts.
```

MCTS 可以具体分为以下四步:

1. Selection: 从之前没有选择过的节点中选择 UCB 值最大的节点。
2. Expansion: 当到达搜索树的边界节点时增加一个新节点。
3. Simulation: 在搜索边界外使用随机策略进行游戏。
4. Backpropagation: 到达终端节点后, 在选择和扩展中对状态进行更新值和访问扩展。

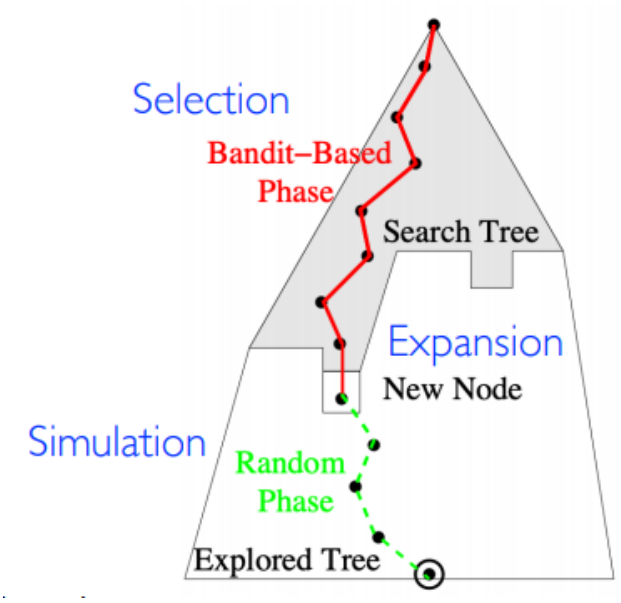

图 7. Monte-Carlo 算法的步骤 2。

2.4. 代码实现

```
function MCTS-sample(state)
    if all children of state expanded:
        next-state = UCB-sample(state)
        winner = MCTS-sample(next-state)
    else:
        next-state = expand(random unexpanded child)
        winner = random-playout(next-state)
        update-value(next-state, winner)
    update-value(state, winner)
    return winner
```

Selection

递归地应用选择策略，直到到达未完全展开的节点为止：

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{logN(Parent(n))}N(n) \quad (2)$$

1. U(n): the total utility of rollouts(e.g.wins) that went through node n.
2. N(n): the number of playouts through node n.

```
function UCB-sample(state):
    weights = []
    for child of state:
        w = child.value / child.visits + C * sqrt(
            log(state.visits) / child.visits)
        weights.append(w)
    distribution = [w / sum(weights) for w in
        weights]
    return child sampled according to distribution
```

Expansion

随机选择一个未访问的子位置，并向统计树添加一个新的记录节点。

```
function MCTS-sample(state)
    if all children of state expanded:
        next-state = UCB-sample(state)
        winner = MCTS-sample(next-state)
    else:
        next-state = expand(random unexpected child)
        winner = random-playout(next-state)
        update-value(next-state, winner)
    update-value(state, winner)
    return winner

function expand(state):
    state.visists = 0
    state.value = 0
```

Simulation

经典的蒙特卡洛模拟博弈方法：模拟真正的博弈过程是十分困难的，我们就随机地执行每一步完成游戏吧！

```
function random-playout(state):
    if is-terminal(state):
        return winner
    else:
        return random-playout(random-move(state))
```

Backpropagation

将游戏的结果反向传播，更新 U(n)(i.e. state.value) 和 N(n)(i.e. state.visits).

```
function update-value(state, winner):
    state.visits++
    if winner == state.turn:
        state.value += 1
```

Devision

返回结果：访问最频繁的路径（i.e. highest N(n))。为什么不是胜率最高的路径？因为有的节点可能会出现只访问了一两次，这样胜率很有可能可以达到 100%.

```
function Monte-Carlo-Tree-Search(state) returns an
    action
    tree = Node(state)
    while Is-Time-Remaining() do
        MCTS-sample(tree)
    return the move in Actions(state) whose node has
        highest number of playouts
```

我们的实现位于 mcts_pure.py 这个文件中。

2.5. MCTS 的性能分析

Pros

1. 平衡了 exploration 和 exploitation 的之间的关系，搜索树的生长不对称。
2. 不受分支因子的影响。
3. 很容易改编代码使之适应新的游戏。
4. 不需要设计启发函数，但可以使用启发函数使之更快到达叶子节点。

Cons

1. 不能处理极端的树深度。
2. 需要易于模拟和大量的计算资源。
3. 需要随机行棋之间弱相关。
4. 如果要适应新的游戏，需要专业知识去调参。
5. 理论性质还没有完全被理解。

3. 基于 AlphaGo Zero 的强化学习

通过使用经过训练可以复制人类专家决策的有监督学习系统，人工智能取得了很大进展。然而，专家数据集往往昂贵、不可靠或根本不可用。即使有可靠的数据集，它们也可能对以这种方式训练的系统的性能施加一个上限。

相比之下，强化学习系统是从自身的经验中训练出来的，原则上允许它们超越人类的能力，并在人类专业知识缺乏的领域运作。AlphaZero 用深度神经网络和对棋盘的强化学习算法取代了传统游戏程序中基于专家知识实现的增强。

3.1. 基本思想

我们使用了两个神经网络：

- Policy network: Output move(action) probabilities.
- Value network: Output a position evaluation.

价值网络根据策略网络的结果预测可能的获胜者，这些网络和 MCTS 合并在一起提供了前向搜索的方法：

- 使用策略网络把搜索空间缩小到可能性更高的 moves.
- 使用价值网络对搜索树中的节点进行评估。

AlphaZero 版本的蒙特卡洛智能体有如下的特点：

- 仅仅通过 self-play 强化学习训练，不需要任何监督或者使用专家知识。
- 只使用棋盘上的黑白棋作为输入特征。
- 基于神经网络实现更简单的树搜索，不需要执行 rollouts.

3.2. 使用的强化学习算法

我们使用的神经网络表示为 f_θ, 其中参数是 θ. 该神经网络以棋盘 s 作为输入, 输出为行棋的概率 $\vec{p}$ 和值 v.

$$f_\theta = (\vec{p}, v) \tag{3}$$

其中 $\vec{p}$ 表示在棋盘 s 的状态下选择 $p_a = Pr(a|s)$, v 表示当前的 player 从棋盘 s 获胜的概率。

如图 9 所示，我们 Self-play 的强化学习可以描述如下：

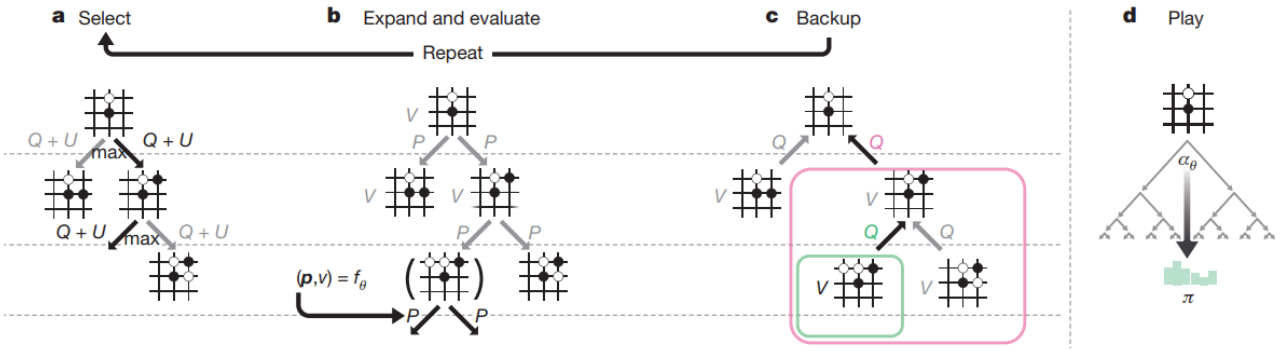

图 8. MCTS in alphaGo Zero.

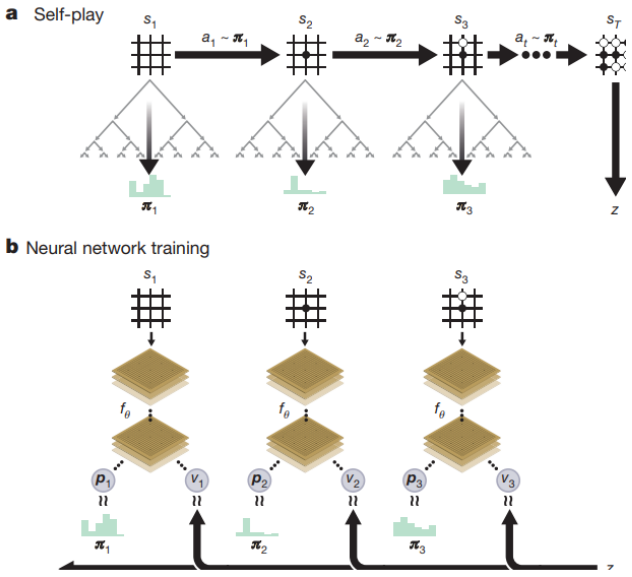

图 9. Self-play reinforcement learning in AlphaGo Zero.

MCTS 使用神经网络 f_θ 替代原来的 simulations. 搜索树中的每一条边存储先验概率 P(s, a), 访问次数 N(s, a) 和一个动作值 Q(s, a). 每一次 simulation 从一个根节点开始迭代地选择使上限置信区间 $Q(s,a)+U(s,a)$ 达到最大的动作, 其中 $U(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$, 直到达到一个叶子节点 s'。然后对叶节点进行扩展并使用神经网络进行评价:

$$f_\theta(s') = (P(s',), V(s')) \tag{4}$$

对 simulation 中的每条边增加计数 N(s, a), 更新 Q(s, a):

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s'|s,a->s'} V(s') \tag{5}$$

Algorithm 1 self-play reinforcement learning

1: The program plays a game $s_1, s_2, .., s_T$ against itself. In each position s_t, an MCTS α_θ is executed using the latest neural network f_θ.
2: Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z.
3: Neural network training in AlphaGo Zero. The neural network takes the raw board position s_t as its input, passes it through many convolutional layers with parameters θ, and outputs both a vector $\vec{p_t}$, and a scalar value v_t.
4: The neural network parameters θ are updated to maximize the similarity of the policy vector p_t to the search probabilites π_t and to minimize the error between the predicted winner v_t and the game winner z.
5: The new parameters are used in the next iteration of self-play.

MCTS 可以看作在给定神经网络的参数 θ 和根节点 s 的情况下，计算出一个可能移动的概率向量 $\alpha_\theta(\vec{s})$. 在时刻 t, $\alpha_{\theta_{i-1}}(s_t) = \pi_t$ 使用上一次迭代得到的参数 θ_{i-1} 计算得到。

和之前提到的 exploration-exploitation dilemma 类似，我们必须保证 self-play 生成的数据具有多样性，根据动作概率的向量值，我们加

入了一个 0.25 加权，$\eta = 0.3$ 的 Dirichlet 分布扰动，增加了动作取样的多样性。

当游戏结束、搜索值小于阈值、搜索树深度大于最大深度时，返回一个 reward r. 时刻 t 的数据可以表示为 (s_t, π_t, z_t), 其中 $z = \pm r$, 表示从当前玩家的视角观察到的 reward.

调整新的神经网络参数使得：

- 最小化预测值 v 和 self-play 的获胜者 z 之间的差异.
- 最大化移动概率 p 和搜索概率 π 之间的相似性.

具体而言，θ 通过对 MSE 损失函数和交叉熵损失函数之和进行梯度下降法：

$$l = (z - v)^2 - \pi^T log(\vec{p}) + c||\theta||^2$$

其中 c 是用来控制 L2 惩罚项的超参数。

3.3. 对于训练数据的补充

我们注意到棋盘具有旋转对称性和翻转对称性。为了减少训练局数，我们对每一局 self-play 都进行旋转和翻转，将等价的局面存起来扩充训练数据。

3.4. 网络结构

由于五子棋并不是非常复杂，所以我们也尽量使用了简单的神经网络可以更快速的训练参数。我们一开始使用了三层的卷积神经网络，分别使用了 32、64、128 个通道、3×3 的卷积核和 ReLU 激活函数。

然后我们利用卷积层，分成策略和价值两个输出。policy 先使用通道为 4 的 1×1 卷积核进行降维，然后连接一个全连接层，最后使用 log_softmax 函数返回落子概率。value 使用通道为 2 的卷积神经网络进行降维，然后连接两个全连接层，最后使用 tanh 函数返回局面的 score.

3.5. 成果展示

最小化损失函数 根据前面介绍的强化学习算法，我们训练的目标是让策略网络输出的概率 $\vec{p}$ 更加接近 MCTS 输出的概率 π, 让价值网络输出的 score v 更接近对弈结果 z。所以从优化的角度讲，我们应该不断最小化损失函数：

$$l = (z - v)^2 - \pi^T log(\vec{p}) + c||\theta||^2$$

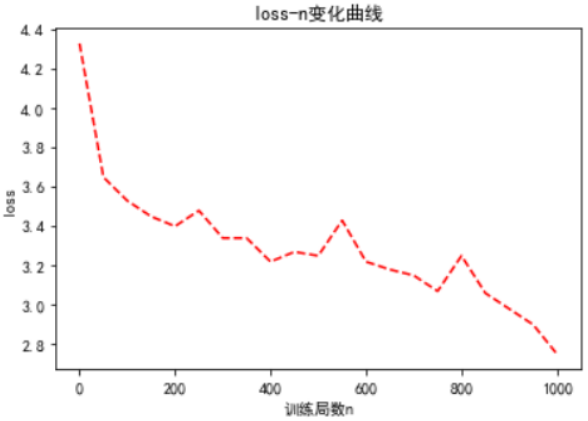

图 10. loss 随训练局数增加逐渐降低 (仅展示前 1000 局的结果)

和朴素的蒙特卡洛智能体对弈 我们首先采用的是训练 1000 局的朴素蒙特卡洛智能体，然后每次训练到 AlphaGo Zero 智能体以 10 : 0 战胜该朴素的蒙特卡洛智能体时，朴素蒙特卡洛智能体的训练局数加 1000。结果如下：

- 经过 450 局，Alpha Zero 智能体以 10:0 战胜训练 1000 局的 MCTS 智能体。
- 经过 1150 局，Alpha Zero 智能体以 10:0 战胜训练 2000 局的 MCTS 智能体。
- 经过 1600 局，Alpha Zero 智能体以 10:0 战胜训练 3000 局的 MCTS 智能体。
- 经过 2650 局，Alpha Zero 智能体以 10:0 战胜训练 4000 局的 MCTS 智能体。

上述结果证明，我们实现的 Alpha Zero 智能体比起蒙特卡洛智能体具有更加优秀的性能。

4. 小组分工

1. 屏风四子棋代码：贠龙飞、唐小龙讨论共同完成。戴劲完成平局判断函数 isDraw。
2. 展示 PPT、视频：贠龙飞、唐小龙、戴劲。

3. 基于 AlphaZero 的五子棋代码：负龙飞、唐小龙讨论共同完成。
4. 大作业文档：负龙飞。