# 作业 5: Mini AlphaGo

季千焜（221300066、qkjiai@smail,nju.edu.cn）

(南京大学 人工智能学院，南京 210093)

**摘 要:** 本次作业的目的是结合 MCTS、强化学习和 Self Play 等技术，实现一个简单的围棋 AI 程序。在本次作业中，需要阅读程序，参考其中使用的强化学习算法，并结合 MCTS/ Self Play 方法，完成 AlphaGo / AlphaGo Zero 框架。

**关键词:** 强化学习、MCTS 算法、监督学习、AlphaGo

注：能力有限，仅完成了 AlphaGo 原文和本次作业代码阅读，并有了实现 MCTS 方法的思路，没有具体的代码实现，助教哥哥不用看代码了，希望实验报告给个感情分。

## 1 任务一

阅读 AlphaGo 原文(PDF)，阅读本次作业代码，熟悉与围棋环境的数据交互方式及 RL 算法在提供的围棋环境下训练的方法，策略网络、值函数网络模型保存的方式。并且在围棋环境中实现 MCTS 方法。

### 1.1 与围棋环境的数据交互方式：

作业代码提供了一个名为 GoEnv 的围棋环境的类，它继承了 gym.Env 类，实现了围棋游戏的规则和逻辑。

GoEnv 类的主要属性和方法有：

board：一个二维数组，表示棋盘上的棋子，0 表示空，1 表示黑子，-1 表示白子。

player_color：一个整数，表示当前轮到的玩家的颜色，1 表示黑，-1 表示白。

done：一个布尔值，表示游戏是否结束。

winner：一个整数，表示游戏的胜者，1 表示黑胜，-1 表示白胜，0 表示平局，None 表示未分出胜负。

reset()：重置棋盘和玩家颜色，返回初始状态。

step(action)：执行一个动作，更新棋盘和玩家颜色，返回新的状态，奖励，是否结束，和额外信息。

render()：绘制棋盘，显示当前局面。

get_legal_actions()：返回一个列表，包含所有合法的动作。

get_winner()：根据棋盘上的子数，判断游戏的胜者，更新 winner 属性。

与围棋环境的数据交互方式是通过 reset()，step()，render()等方法，输入或输出状态，动作，奖励等信息，实现智能体和环境的交互。

### 1.2 RL 算法在提供的围棋环境下训练的方法

RL 算法在提供的围棋环境下训练的方法大致如下：

使用两个神经网络，一个是策略网络（policy network），一个是值函数网络（value network）。

策略网络的输入是棋盘状态，输出是每个动作的概率，用于逼近最优策略 $\pi*(a|s)$。

值函数网络的输入是棋盘状态，输出是当前状态的价值，用于逼近最优值函数 $V*(s)$。

使用监督学习的方法，从人类棋谱中训练策略网络，使其能够模仿人类的下棋方式。

使用强化学习的方法，利用自我对弈的数据，对策略网络和值函数网络进行更新，使其能够提高自身的水平。

使用 MCTS 的方法，结合策略网络和值函数网络，对每个可能的动作进行搜索和评估，选择最优的动作。

使用 Self Play 的方法，让当前的 AI 与历史版本的 AI 进行对弈，不断生成新的数据，提升 AI 的能力。

### 1.3 策略网络、值函数网络模型保存的方式

本次作业使用 PyTorch 框架来实现神经网络，因此可以使用 PyTorch 提供的方法来保存和加载模型。

1.3.1 一种方法是直接保存整个模型，包括网络结构和参数，这种方法比较简单，但是不太灵活，需要使用相同的代码来加载模型，而且可能会保存一些不必要的信息。保存和加载的代码如下：

```
# 保存模型
torch.save(policy_net, 'policy_net.pth')
torch.save(value_net, 'value_net.pth')
# 加载模型
policy_net = torch.load('policy_net.pth')
value_net = torch.load('value_net.pth')
```

1.3.2 另一种方法是只保存模型的参数，即 state_dict，这种方法比较灵活，可以在不同的代码中加载模型，而且只保存了必要的信息。保存和加载的代码如下：

```
# 保存模型
torch.save({'policy_net': policy_net.state_dict()}, 'policy_net.pt')
torch.save({'value_net': value_net.state_dict()}, 'value_net.pt')
# 加载模型
policy_net = PolicyNet() # 创建一个新的策略网络对象
value_net = ValueNet() # 创建一个新的值函数网络对象
policy_net.load_state_dict(torch.load('policy_net.pt')['policy_net']) # 加载参数
value_net.load_state_dict(torch.load('value_net.pt')['value_net']) # 加载参数
```

### 1.4 实现 MCTS 方法的思路

**一个没有具体实现的思路代码如下：**

```
# 定义节点类，包含状态、父节点、子节点、访问次数、累积价值等属性
class Node:
    def __init__(self, state, parent=None):
        self.state = state # 当前的游戏状态
        self.parent = parent # 父节点
        self.children = [] # 子节点列表
        self.N = 0 # 访问次数
        self.W = 0 # 累积价值
        self.Q = 0 # 平均价值
        self.P = 0 # 先验概率

# 定义 MCTS 类，包含根节点、模拟次数、探索常数等属性
class MCTS:
    def __init__(self, root, num_simulations, c_puct):
        self.root = root # 根节点
        self.num_simulations = num_simulations # 模拟次数
        self.c_puct = c_puct # 探索常数
```

```
# 定义选择函数，根据 UCB 公式选择最优的子节点
def select(self, node):
    # 如果节点是叶子节点，直接返回
    if len(node.children) == 0:
        return node
    # 否则，计算所有子节点的 UCB 值，选择最大的一个
    max_ucb = -float('inf')
    best_child = None
    for child in node.children:
        # UCB = Q + c_puct * P * sqrt(N) / (1 + N)
        ucb = child.Q + self.c_puct * child.P * math.sqrt(node.N) / (1 + child.N)
        if ucb > max_ucb:
            max_ucb = ucb
            best_child = child
    # 递归地选择下一个子节点
    return self.select(best_child)

# 定义扩展函数，根据策略网络输出添加新的子节点
def expand(self, node):
    # 获取当前状态下的所有可能的行动
    actions = get_legal_actions(node.state)
    # 获取策略网络输出的行动概率分布
    action_probs = policy_network.predict(node.state)
    # 为每个可能的行动创建一个子节点
    for action in actions:
        # 获取行动对应的概率
        prob = action_probs[action]
        # 获取执行行动后的新状态
        new_state = step(node.state, action)
        # 创建子节点并添加到子节点列表
        child = Node(new_state, node)
        child.P = prob
        node.children.append(child)

# 定义模拟函数，根据快速走子策略进行模拟，返回模拟结果
def simulate(self, node):
    # 获取当前的状态
    state = node.state
    # 循环直到游戏结束
    while not is_terminal(state):
        # 获取当前状态下的所有可能的行动
        actions = get_legal_actions(state)
        # 获取快速走子策略输出的行动概率分布
```

```
            action_probs = fast_policy.predict(state)
            # 按照概率分布随机选择一个行动
            action = np.random.choice(actions, p=action_probs)
            # 获取执行行动后的新状态
            state = step(state, action)
        # 返回最终的游戏结果
        return get_result(state)

    # 定义反向传播函数，根据模拟结果更新节点的统计信息
    def backpropagate(self, node, result):
        # 如果节点不是空的，更新节点的访问次数和累积价值
        if node is not None:
            node.N += 1
            node.W += result
            node.Q = node.W / node.N
            # 递归地更新父节点
            self.backpropagate(node.parent, -result) # 反转结果，因为父节点是对手的回合

    # 定义执行函数，执行一次 MCTS 搜索
    def run(self):
        # 循环执行指定次数的模拟
        for i in range(self.num_simulations):
            # 从根节点开始，选择最优的叶子节点
            node = self.select(self.root)
            # 如果叶子节点不是终止状态，扩展出新的子节点
            if not is_terminal(node.state):
                self.expand(node)
                # 从新的子节点中随机选择一个
                node = random.choice(node.children)
            # 从选中的节点开始，进行一次模拟，得到模拟结果
            result = self.simulate(node)
            # 根据模拟结果，从选中的节点向上反向传播，更新统计信息
            self.backpropagate(node, result)
        # 返回访问次数最多的子节点作为最优的行动
        best_child = max(self.root.children, key=lambda x: x.N)
        return best_child
```

**References**:

[1] https://blog.csdn.net/newlw/article/details/125313094?spm=1001.2014.3001.55066.