基于蒙特卡洛树搜索与策略价值网络的五子棋 AI 算法设计

姓名：王明宇
学号：2023080911015

完成日期：2023 年 12 月 6 日

摘要

随着人工智能的发展，深度学习与强化学习等算法被广泛运用于解决各类博弈问题。而棋类游戏博弈问题的研究，始终是人工智能领域研究的热点。从上世纪中期至今，棋类游戏 AI 经历了巨大的进步。过去的棋类游戏 AI，有的依靠传统博弈树搜索来穷举大量数据，而另一些依靠机器学习技术，但依然需要大量的样本数据进行训练学习。在棋类游戏博弈算法的设计中，如何兼顾时间与博弈能力，是一个值得讨论的问题。在本次课程设计中，我们基于 DeepMind 的 AlphaZero 围棋博弈算法，使用蒙特卡洛树搜索与深度学习神经网络设计了一种从零开始学习五子棋博弈的五子棋 AI 算法。其中神经网络是经过设计的策略价值网络；蒙特卡洛树搜索可根据多次模拟博弈的结果预测最优的移动方案。将五子棋规则与蒙特卡洛树搜索和策略价值网络相结合，蒙特卡洛树搜索使用策略价值网络评估落子位置和选择移动，增强树的搜索强度，提高落子质量，优化自对弈迭代。通过蒙特卡洛树搜索进行自对弈，训练一个神经网络来预测落子选择以及游戏的赢家。最后，测试结果表明此算法比较其它算法具有更低的资源占用与更高的博弈准确度。

关键词：强化学习；蒙特卡落树搜索；五子棋；深度学习

目录

第一章 设计目的

棋类游戏 AI 的发展历程是计算机科学和人工智能领域的一个重要章节，经历了从简单算法到高级机器学习技术的转变。1950 年代末，人们开始尝试开发可以下棋的计算机程序。这些程序主要基于规则，计算力有限。1960 年代，国际象棋程序开始出现。最初的版本比较原始，但它们奠定了计算机棋类游戏的基础。在 1970 年代，国际象棋程序开始采用更先进的搜索算法，如迭代深化搜索和 α－β 剪枝，这显著提高了它们的实力。在 1970 年代和 1980 年代，贝尔实验室开发的程序在国际象棋计算机锦标赛中表现出色。1996 年，IBM 的深蓝在一场历史性的比赛中首次击败了当时的世界冠军加里·卡斯帕罗夫。深蓝能够每秒评估约 2 亿个棋局位置，展示了原始计算力在棋类游戏中的重要性。2016 年，AlphaGo 使用深度学习和蒙特卡洛树搜索，首次在比赛中击败了世界级围棋选手李世石。AlphaGo Zero 和 AlphaZero 使用了强化学习，通过自我对弈无需依赖人类棋谱就能达到超高水平。棋类游戏 AI 的发展反映了人工智能技术的整体进步。从早期的基于规则的程序到利用深度学习和强化学习的高级算法，AI 在棋类游戏中的应用不断推动了其在更广泛领域的发展和应用。棋类游戏 AI 的成功不仅在于它们在游戏中的表现，还在于它们为解决复杂问题提供的方法和思路，对人工智能领域产生了深远影响。

五子棋，又称为连珠、五子连线、五子棋、五目並べ（日语）、gomoku（英语），是一种两人对弈的纯策略型棋类游戏。它简单易学，但却拥有深刻的策略和技巧，是世界流行的棋类游戏之一。五子棋游戏具有很大的分支因素，因此在使用暴力搜索算法时，较深的搜索深度将导致大量的计算资源占用，导致搜索效率下降，乃至计算资源无法满足。而使用蒙特卡洛树搜索办法可以在极少分支的情况下达到极深的搜索深度，较快找到最优落子策略。五子棋游戏也具有灵活变化性，传统算法通过穷举的方式，难以识别复杂棋型，传统的机器学习算法也需要大量的数据进行训练。而深度学习算法可以通过自我对弈来学习与提升，它能够识别出有效的棋型和潜在的胜利策略，甚至能发现传统棋手未曾注意的新策略，与依赖人类棋谱的其它算法不同，深度学习算法可以在没有任何人类先验知识的情况下自主学习游戏。

本课程设计将探究基于蒙特卡洛树搜索与深度学习算法的五子棋 AI 的实现过程。本算法可实现完全由自我对弈训练，从随机情况开局，无任何人为干预。本算法使用深度学习神经网络与蒙特卡洛树搜索综合评估最优落子位置。

第二章 相关理论

2.1 蒙特卡洛树搜索算法

2.1.1 蒙特卡洛树搜索算法的前世今生

蒙特卡洛树搜索（Monte Carlo Tree Search, MCTS）算法是一种在各类决策过程中广泛应用的搜索算法，特别是在复杂的游戏领域如围棋、象棋等。它的核心思想是通过随机抽样的方式，在有限的时间内估计出最有可能达到胜利的走法。

MCTS 算法的历史可以追溯到 20 世纪 40 年代，当时的统计学家们开始尝试使用随机样本来解决复杂的统计问题，这也是“蒙特卡洛方法”的由来。直到 21 世纪初，这种方法开始被应用于计算机游戏和人工智能领域。2006 年，荷兰的一位研究员 Remi Coulom 发展了 MCTS 的现代形式，并将其应用于围棋游戏中，开启了算法在游戏领域的新纪元。

MCTS 算法的核心包括四个阶段：选择（Selection）、扩展（Expansion）、模拟（Simulation）和回溯（Backpropagation）。首先，在选择阶段，算法从根节点开始，通过评估下一步可能的各个选项，选择出最有可能胜利的路径。接着，在扩展阶段，它会在当前路径的基础上添加一个新的节点。然后，在模拟阶段，算法通过随机模拟来预测从这个新节点开始可能出现的游戏结果。最后，在回溯阶段，算法将模拟的结果反馈到之前的各个节点，用以更新这些节点的统计数据。

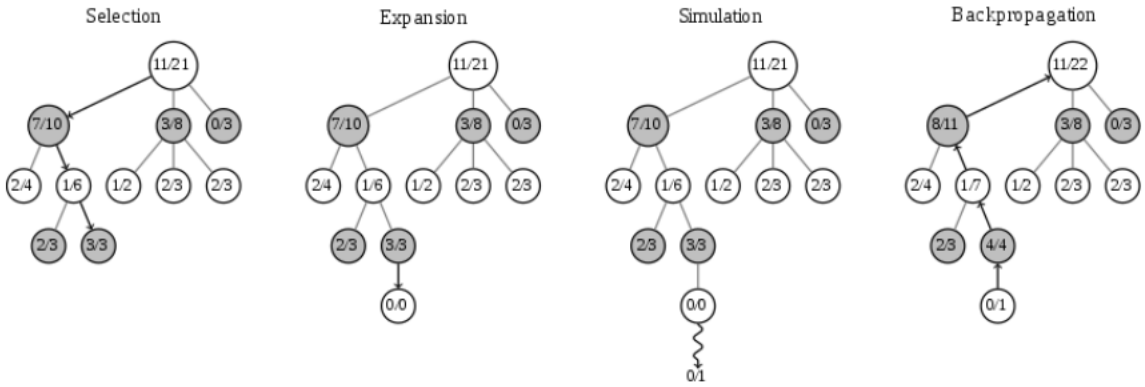
MCTS 的一个关键优势是它不需要对问题域有深入的了解。它通过重复的模拟和统计来逐渐构建出一个决策树，每个节点代表一个可能的游戏状态。通过这种方式，MCTS 能够在相对较短的时间内找到优秀的策略，尤其是在那些规则复杂、可能的状态数量巨大的游戏中。

此外，MCTS 算法的另一个显著特点是它的灵活性。这个算法可以很容易地和其他类型的算法（如深度学习）结合，进一步提升搜索的效果。最著名的例子是 AlphaGo，它结合了深度学习和 MCTS，创造了一个在围棋上能够战胜世界冠军的人工智能系统。

随着技术的不断进步，MCTS 算法在各种领域的应用也在不断扩展，从游戏智能发展到复杂的决策制定和问题解决场景。它的发展历程不仅展现了计算机科学领域的飞速发展，也体现了跨学科研究在解决复杂问题中的巨大潜力。

2.1.2 蒙特卡洛树搜索算法的原理

蒙特卡洛树搜索算法的每次迭代分为四步，**选择**，**拓展**，**模拟**和**反向传播**



2.1.2.1 选择（Select）

在这一阶段，算法从根节点开始，根据一定的策略（如 UCT 公式）沿树向下移动，直到达到一个“有前途”的节点。这个过程是基于已有的模拟结果来决定的，目的是平衡探索（尝试不太熟悉的走法）和利用（选择已知的好走法）。

最终，这一步将到达博弈树底部的一个节点，即“叶节点”

2.1.2.2 拓展（Expansion）

当到达一个还有未探索的子节点的节点时，算法会选择其中之一并添加到树中。这个新节点代表了一种新的、尚未评估的游戏状态。

即当到达一个叶节点时，算法将下一步可能的状态作为新的节点添加到树中

2.1.2.3 模拟（Simulation）

从新添加的节点开始，算法进行快速、随机的模拟游戏，也称为“下棋”阶段。这个模拟可以是简化的，不需要非常智能，目的是快速得出一个游戏结果。

例如在五子棋游戏中，算法沿树向下移动直到达到一个叶节点，这时将下一次所有可能的落子添加至树中，这些新添加的节点的“价值”是尚未评估的，算法将从这些新节点开始，依据某种策略（如随机落子）模拟游戏进程直至游戏终局，从而根据模拟结果为新节点赋予价值（如获胜则价值为 1，平局则为 0，失败则为-1）

2.1.2.4 回传（Backpropagation）

模拟阶段结束后，根据当前节点的状态更新从根节点到新节点之间的所有节点的统计状态（如价值、访问次数等）

2.1.3 最大置信上界算法（Upper Confidence bounds applied to Trees, UCT）

2.1.3.1 探索和利用（Exploration and Exploitation）

探索（Exploration）和利用（Exploitation）是强化学习中的两个核心概念，它们描述了一个学习代理(如人工智能算法)如何在未知环境中做出决策的策略。

探索（Exploration）：

探索是指智能体尝试新行为，以了解这些行为可能带来的奖励。探索对于发现更优策略和了解环境是必要的，尤其是在环境信息不完全或动态变化的情况下通过探索，智能体可以发现之前未尝试过的策略，这些策略可能会带来比当前策略更大的奖励。但过度探索可能导致智能体在低奖励的行为上浪费时间，从而降低整体效率。

示例： ε-贪婪（随机选择探索或利用）、汤普森采样（基于概率模型探索）、熵正则化（增加策略的随机性）。

利用（Exploitatiom）：

利用是指智能体选择它认为当前最优的行为，基于过去的经验和学习。利用使得智能体能够最大化基于当前知识的奖励，确保在已知的好策略上获得稳定回报。过度利用可能导致智能体陷入局部最优，错过更好的策略。

示例：贪婪策略（总是选择最好的已知行为）、值函数最大化（基于预测的最大奖励选择行为）。

强化学习中最大的挑战在于如何在探索新策略和利用已知策略之间找到合适的平衡。这个平衡点取决于多个因素，如环境的动态性、任务的复杂度和智能体的

学习速度。

2.1.3.2 UCT 算法

最大置信上界（Upper Confidence bounds applied to Trees，UCT）算法是一种特别的蒙特卡洛树搜索（MCTS）算法，主要用于解决需要平衡探索与利用的复杂决策问题，如棋类游戏。UCT 通过结合上置信界限（Upper Confidence Bound，UCB）算法来优化 MCTS 的性能。

$$UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$$

MCTS 算法在遍历博弈树时会优先选择 UCT 值更高的节点，这一算法包含两个组件

$$\frac{Q(v_i)}{N(v_i)}$$

这一部分称为利用（Exploitation）组件。其值为子节点v_i的总价值除以其总访问次数，即该子节点的胜率估计。Exploitation 组件使得 MCTS 在搜索过程中总是趋向于胜率更高的节点，但一味的追求最大胜率可能会导致陷入局部最优的困境，无法发现更多潜在的更优策略。UCT 算法中引入第二个组件，探索（Exploration）组件

$$c\sqrt{\frac{\log(N(v))}{N(v_i)}}$$

Exploration 组件提高了那些访问次数较少的节点被选中的概率，随着节点的访问次数的增加，Exploration 的值也会减少。其中 $N(v)$，$N(v)_i$分别为某一节点及其子节点的访问次数。这一组件指导 MCTS 算法进行更充分的探索。此处的 c 为算法超参数，用于控制 Exploration 对 UCT 值的影响程度。

2.2 深度学习算法

2.2.1 深度学习算法概述

深度学习算法是一种先进的机器学习方法，基于人工神经网络的架构，专门设计来模拟人类大脑的信息处理方式。这种算法通过构建多层的神经网络来学习复杂的数据模式，其中每一层由多个神经元或节点组成，负责接收、处理和传递信息。

深度学习的关键在于其多层结构，每一层都能从输入数据中提取特征，并将这些特征传递给下一层，这样层层叠加，使得网络能够学习到数据中的高级抽象特征。这些层分为不同的类型，包括输入层，负责接收原始数据；隐藏层，其中每一层都进行特定的数据处理和特征提取；以及输出层，产生最终的预测或分类结果。深度学习算法的核心是通过激活函数激活神经元，这些函数决定了是否以及如何将信息传递到网络的下一层。常见的激活函数包括 ReLU（线性整流单元）、Sigmoid 和 Tanh，它们帮助网络捕捉输入数据中的非线性关系。训练深度学习模型涉及到大量的数据和复杂的数学运算，特别是反向传播算法，它是一种有效的方式来计算网络中每个权重对最终输出误差的影响，从而可以调整权重以最小化这个误差。优化算法，如随机梯度下降（SGD）或 Adam，随后用于根据这些梯度来调整权重，以优化网络性能。

深度学习算法在多个领域都取得了显著成就，特别是在图像和声音识别、自然语言处理和复杂策略学习等方面。卷积神经网络（CNNs）是一种特别设计用于处理图像的深度学习架构，通过模拟视觉皮层的工作原理来有效地识别和分类图像中的对象。另一方面，循环神经网络（RNNs）和长短期记忆网络（LSTMs）则特别适用于处理序列数据，如时间序列分析或语言建模。近年来，变换器模型，如 BERT 和 GPT 系列，在自然语言处理领域引起了巨大变革，它们通过创新的注意力机制和大规模预训练，显著提高了机器对自然语言的理解能力。总的来说，深度学习的强大之处在于它能够从原始数据中自动学习复杂的表示，而无需人工设计特征，这使得它在许多领域都成为了解决复杂问题的关键技术。

2.2.2 策略价值网络

基于策略价值网络的深度学习算法是一种革命性的方法，代表了人工智能领域的一个重大突破，尤其是在复杂的决策和策略游戏领域。这种算法的核心是将深度神经网络应用于同时学习和优化游戏的策略（即行动的选择）和价值（即行动的潜在价值或长期回报）。这种方法最著名的应用是 DeepMind 的 AlphaZero，它在国际象棋、围棋和将棋等游戏中展示了卓越的能力。

在这种算法中，策略网络和价值网络是两个主要组成部分，但它们被集成在一个统一的架构中。策略部分的目标是预测每个可能行动的概率，基本上告诉系统在当前游戏状态下哪些移动可能是最佳的。价值网络则估计给定游戏状态的胜率，

基本上评估当前位置的优势或劣势。这种组合允许系统不仅理解哪些动作在当前情境下是最佳的，而且还能理解这些动作对游戏结果的长期影响。

这种算法的一个关键特点是它的自我学习能力。通过与自己的早期版本进行成千上万次的自我对弈，系统能够在没有任何外部指导或预先编程的知识的情况下学习游戏的微妙之处。这种自我对弈过程产生的大量数据用于不断训练和调整神经网络，使其越来越精确地预测策略和价值。这样，算法能够从最基本的规则出发，逐步发展出高度复杂和精细的游戏策略。

蒙特卡洛树搜索（MCTS）在这一过程中扮演了关键角色。结合策略和价值网络的输出，MCTS 能够有效地探索可能的走法，同时评估每一步的潜在价值。这种结合深度学习和树搜索的方法使得基于策略价值网络的算法能够在不确定性很高的环境中做出非常复杂的决策。

第三章 算法设计

3.1 蒙特卡洛树搜索算法设计

与传统的蒙特卡洛树搜索算法不同，AlphaZero 算法的蒙特卡洛树搜索算法优化了模拟与回传步骤，用神经网络指导的自我对弈过程代替传统的 rollout 随机落子过程

将神经网络记作f_θ，则对于任一游戏状态s_t，神经网络将生成相应的落子概率分布与价值分布

$$p_{s_t}, v_{s_t} = f_\theta(s_t)$$

3.1.1 选择阶段

以五子棋棋盘的当前状态s_0作为根节点，若全部下一步可能落子未添加至树中，则根据当前状态s_0生成下一步落子的先验概率与价值p_0, v_0，然后选择一个 Q 与 U 最大的落子动作 d 进行落子

$$d = argmax(Q(s,d) + U(s,d))$$

$$U(s,d) = c_{puct} P(s,d) \frac{\sqrt{\sum C(s,d)}}{1 + C(s,d)}$$

$Q(s,d)$	s 局面下 d 落子动作的平均行动价值
$P(s,d)$	s 局面下 d 落子动作的先验概率
$\sum C(s,d)$	s 局面下所有落子动作的访问次数之和（即根节点的访问次数）
$C(s,d)$	s 局面下 d 落子动作的访问次数
c_{puct}	算法超参数，用于平衡搜索过程中 Exploration 的影响，当此参数较大时，算法将趋向于探索那些访问次数较小的节点，此参数较小时，算法将快速收敛

这种搜索算法最初将趋向于先验概率较大，访问次数较小的动作，之后将渐渐趋向于平均行动价值较大的动作。

其中$P(s,d)$由神经网络直接评估得到，而$Q(s,d)$则在评估阶段不断更新

代码实现如下：

```
//展开
```

```
void expand() {
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            if (policy[i][j] != 0) {
                createNode(j,i,0,value[i][j],0,policy[i][j],0,(target->type==B_BLACK?B_WHITE:B_BLACK));
            }
        }
    }
}

//根据 UCT 算法选择节点
void chooseNode() {
    double max;
    for (int i = 0; i < childrenNum; ++i) {
        if (target -> children[i] != NULL) {
            max = max > target->children[i]->P_real ? max : target->children[i]->P_real;
        } else {
            break;
        }
    }
    for (int j = 0; j < childrenNum; ++j) {
        if (target->children[j] != NULL) {
            if (target->children[j]->P_real == max) {
                target = target->children[j];
            }
        } else {
            break;
        }
    }
}
void UCT() {
    for (int i = 0; i < childrenNum; ++i) {
        if(target -> children[i] != NULL) {
            target -> children[i] -> P_real = C_UCT * target->children[i]->P *
( sqrt((double)root()->N)/(1+(double)target->children[i]->N) );
            target -> children[i] -> P_real += target -> children[i] -> Q;
        } else {
            break;
        }
    }
    chooseNode();
}
```

3.1.2 评估阶段

从选中的下一步行动节点s_i开始，以自我对弈的方式模拟评估游戏进程以至最大迭代次数或游戏终局

每一次自我对弈时，将当前状态输入神经网络得到下一步走子的先验概率与价值，并用 UCT 算法选择最佳节点并落子，每次落子时将根据当前节点的先验价值更新节点s_i的平均行动价值

$$Q(s,d) = Q(s,d) + \frac{(v - Q(s,d))}{\mathrm{C}(s,d)}$$

或

$$Q(s,d) = \frac{(v + (\mathrm{C}(s,d) - 1)Q(s,d))}{\mathrm{C}(s,d)}$$

$Q(s,d)$	s 局面下 d 落子动作的平均行动价值
v	当前节点的价值
$\mathrm{C}(s,d)$	s 局面下 d 落子动作的访问次数

若自我对弈中遭遇游戏终局，则直接设定当前节点的价值为 1（胜），0（平）或者-1（负），并进入下一轮评估

代码实现如下：

```
//评估与回传
void EstimateAndBack(std::string model_name,std::string model_save_name) {
    //初始化神经网络
    GobangCNN gobangCNN;
    ValuePolicyLoss lossFunc;
    if (!model_name.empty()) {
        torch::serialize::InputArchive input_archive;
        input_archive.load_from(model_name);
        gobangCNN.load(input_archive);
    }
    torch::optim::Adam optimizer(gobangCNN.parameters(), torch::optim::AdamOptions(1e-4));
    data_trans((target->type == B_BLACK ? B_WHITE : B_BLACK),REAL);
    auto input = convertBoardDataToTensor(NN_input);
    //第一次神经网络评估，此时应保存第一次的结果供后续损失函数使用
    auto [v, p] = gobangCNN.forward(input);
    auto orin_v = v.detach();
    auto orin_p = p.detach();
    trans_policy(v);
    trans_value(p);
    //第一次展开，生成所有可能的 move;
```

```
expand();
for (int i = 0; i < E_times; ++i) {
    printf("%d\n",i);
    //重置虚拟棋盘与虚拟历史落子数据
    resetBoard(VIRTUAL);
    reset_HCD(VIRTUAL);
    //选择一个 move
    back_to_root();
    target -> N++;
    UCT();
    target->N++;
    //存储下一步落子颜色
    int type = target -> type;
    //落子
    chess(target->X,target->Y,target->type,VIRTUAL);
    data_trans((target->type == B_BLACK ? B_WHITE : B_BLACK),VIRTUAL);
    //开始评估
    for (int j = 0; j < MCTS_times; ++j) {
        //神经网络评估
        auto input1 = convertBoardDataToTensor(NN_input);
        auto [v1, p1] = gobangCNN.forward(input1);
        trans_value(v1);
        trans_policy(p1);
        //拓展
        expand();
        //选择
        UCT();
        //落子
        chess(target->X,target->Y,target->type,VIRTUAL);
        data_trans((target->type == B_BLACK ? B_WHITE : B_BLACK),VIRTUAL);
        //判断游戏是否到达终局
        game_terminate(VIRTUAL);
        if (win != 0) {
            if (win==3) {
                target->v = 0;
            } else {
                target->v = win == type ? 1 : -1;
            }
            back_value();
            break;
        }
        //回传
        back_value();
    }
```

```
            //释放内存，清除用无节点
            while (!is_next_move()) {
                target = target->father;
            }
            for (int j = 0; j < childrenNum; ++j) {
                if (target->children[j] != NULL) {
                    freeTree(target->children[j]);
                    target->children[j] = NULL;
                }
            }
        }
        //评估结束，返回根节点
        back_to_root();
        //生成落子概率与价值矩阵
        double P_mcts[SIZE][SIZE];
        double V_mcts[SIZE][SIZE];
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                P_mcts[i][j] = 0;
                V_mcts[i][j] = 0;
            }
        }
        for (int i = 0; i < childrenNum; ++i) {
            if (target->children[i] != NULL) {
                P_mcts[target->children[i]->Y][target->children[i]->X] = ((double)target->children[i]->N) /
sum_N();
                V_mcts[target->children[i]->Y][target->children[i]->X] = target->children[i]->Q;
            }
        }
        //反向传播更新神经网络
        auto p_mcts = convertMCTSResultToTensor(P_mcts);
        auto v_mcts = convertMCTSResultToTensor(V_mcts);
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                if (orin_p[0][i][j].item<float>() == 0) {
                    orin_p[0][i][j] = 1;
                }
            }
        }
        auto loss = lossFunc.forward(gobangCNN,orin_p,orin_v, p_mcts, v_mcts);
        optimizer.zero_grad();  // 清除之前的梯度
        loss.backward();
        optimizer.step();
        torch::serialize::OutputArchive output_archive;
```

```
    gobangCNN.save(output_archive);
    output_archive.save_to(model_save_name);
}
```

3.1.3 执行阶段

每步走子进行若干次（本文中设置为 500 次）蒙特卡洛树搜索后，所有数据将存储在树的边 $\mathrm{r}(s,d)$中，落子概率从这些数据中得到，其值为：

$$\pi(s,d) = \frac{C(s,d)}{\sum C(s,d)}$$

此时根据每个落子动作的落子概率选择最佳落子，以选中落子动作节点作为新树的根节点，重复进行下一轮蒙特卡洛树搜索

代码实现如下：

```
/*
* 根据拓展、评估、回传阶段的结果选择最佳 move 作为下一步的根节点
*/
void inheritRoot() {
    int max_N = 0;
    for (int i = 0; i < childrenNum; ++i) {
        if (target->children[i]!=NULL) {
            max_N = max_N>target->children[i]->N ? max_N : target->children[i]->N;
        } else {
            break;
        }
    }
    int j;
    for (j = 0; j < childrenNum; ++j) {
        if (target->children[j]->N==max_N) {
            target = target->children[j];
            break;
        }
    }
    for (int i = 0; i < childrenNum; ++i) {
        if (i!=j) {
            free(target->father->children[i]);
            target->father->children[i] = NULL;
        }
    }
    free(target->father);
    target->father = NULL;
    target->N = 0;
```

```
    chess(target->X,target->Y,target->type,REAL);
}
```

3.2 神经网络设计

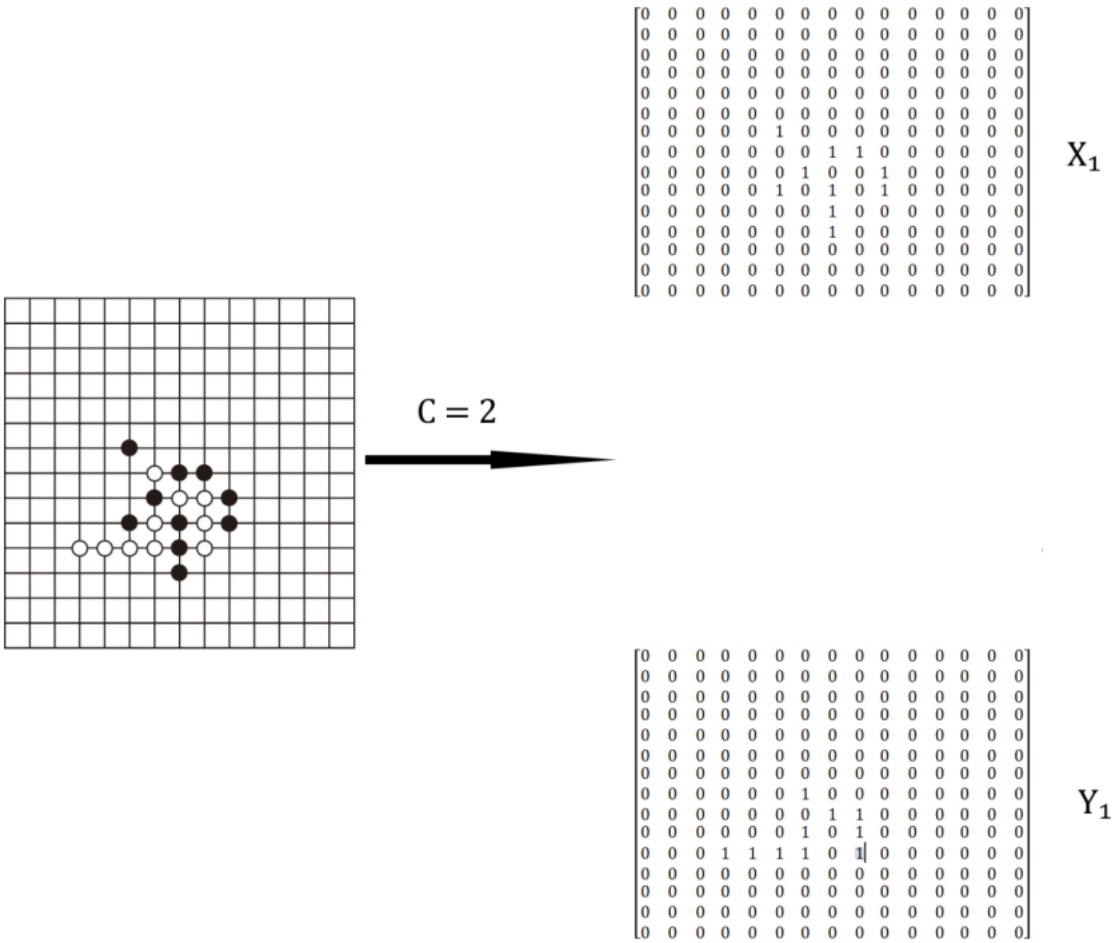
原版 AlphaZero 算法中使用了极其庞大的神经网络，首先将棋盘状态输入进 20/40 个带有卷积层的残差网络模块中，而后再经过 2/3 层网络结构得到策略、价值输出。本设计中将网络结构极大简化以减少计算资源的使用

3.2.1 神经网络结构设计

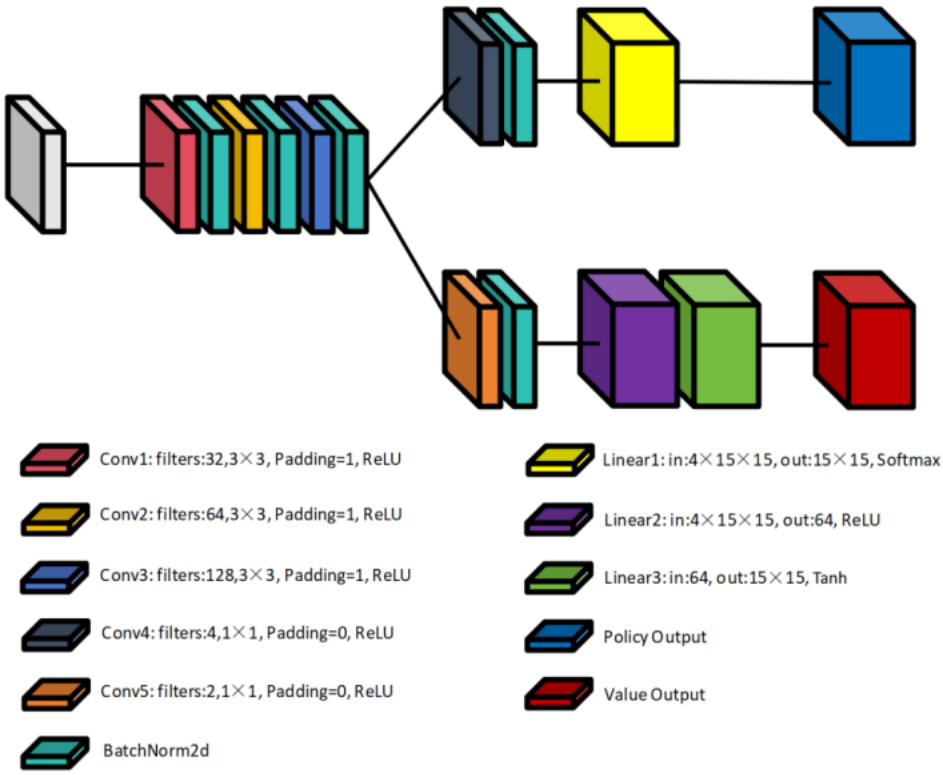
网络的输入为棋盘的历史落子状态，其数据包含了双方过去八手的落子以及一个颜色标记通道，数据尺寸为 17×15×15

$$s_t = \{X_1, Y_1, X_2, Y_2, \ldots\ldots, X_8, Y_8, C\}$$

s_t	当前棋盘状态
X_i	己方过去第 i 手落子，己方落子表示为 1，其它位置均为 0
Y_i	对方过去第 i 手落子，对方落子表示为 1，其它位置均为 0
C	颜色标记通道，标记己方所持棋子颜色，在本文中黑子为 1，白子为 2



输入的数据将先经过三层公共的卷积层网络，分别使用 32、64、128 个 3×3 尺寸的卷积核，均使用 ReLU 作为激活函数，且每个卷积层后都有一个正则化层，然后将网络分为 Policy 与 Value 两个输出端。Policy 输出端先经过 4 个 1×1 的卷积核进行降维，使用 ReLu 作为激活函数，再连接一个正则化层和使用 Softmax 作为激活函数的全连接层，输出棋盘上每个位置的落子概率。Value 输出端先经过 2 个 1×1 的卷积核进行降维，使用 ReLU 作为激活函数，其后连接然后一个正则化层，最后经过一个 64 神经元，使用 ReLU 激活函数的全连接层和和一个使用 tanh 激活函数的全连接层输出局面评分。整个网络的层数简化至 9/10 层，极大的减少了资源使用，提高了训练效率



代码实现如下：

```
// GobangCNN 类的构造函数的定义
GobangCNN::GobangCNN() {
    // 定义和初始化网络层
    publicConv = torch::nn::Sequential(
                torch::nn::Conv2d(torch::nn::Conv2dOptions(17, 32,3).padding(1)),
                torch::nn::BatchNorm2d(32),
                torch::nn::ReLU(),
                torch::nn::Conv2d(torch::nn::Conv2dOptions(32, 64, 3).padding(1)),
                torch::nn::BatchNorm2d(64),
                torch::nn::ReLU(),
                torch::nn::Conv2d(torch::nn::Conv2dOptions(64, 128, 3).padding(1)),
                torch::nn::BatchNorm2d(128),
                torch::nn::ReLU()
    );

    value = torch::nn::Sequential(
                torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 2, 1)),
                torch::nn::BatchNorm2d(2),
                torch::nn::ReLU(),
                torch::nn::Flatten(),
                torch::nn::Linear(2 * 15 * 15, 64),
                torch::nn::ReLU(),
```

```
                torch::nn::Linear(64, 15 * 15),
                torch::nn::Tanh()
    );

    policy = torch::nn::Sequential(
                torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 4, 1)),
                torch::nn::BatchNorm2d(4),
                torch::nn::ReLU(),
                torch::nn::Flatten(),
                torch::nn::Linear(4 * 15 * 15, 15 * 15),
                torch::nn::Softmax(/*dim=*/1)
    );

    // 注册模块
    register_module("publicConv", publicConv);
    register_module("value", value);
    register_module("policy", policy);
}

// GobangCNN 类的 forward 方法的定义
std::tuple<torch::Tensor, torch::Tensor> GobangCNN::forward(torch::Tensor x) {
    auto y = publicConv->forward(x);
    auto v = value->forward(y);
    auto p = policy->forward(y);

    v = unflatten(v, 15, 15);
    p = unflatten(p, 15, 15);

    // 遍历并更新 v 和 p Tensor 的值
    for (int i = 0; i < 15; ++i) {
        for (int j = 0; j < 15; ++j) {
            if (x[0][0][i][j].item<float>() != 0) {
                v[0][i][j] = 0;
                p[0][i][j] = 0;
            }
            if (x[0][1][i][j].item<float>() != 0) {
                v[0][i][j] = 0;
                p[0][i][j] = 0;
            }
        }
    }
    return std::make_tuple(v, p);
}
```

3.2.2 损失函数与优化器

3.2.2.1 损失函数设计

神经网络输入当前局面情况，输出下一步落子的落子概率分布 P 与价值分布 V，通过蒙特卡洛树搜索过程得到新的落子概率分布 π 与价值分布 Q。本设计中令神经网络输出的策略价值分布接近蒙特卡洛树搜索输出的策略价值分布以达到训练的目的，对于价值分布采用均方损失，策略分布采用交叉熵损失，并引入 L2 正则化以避免过拟合

$$Loss_{value} = (Q - V)^2$$

$$Loss_{policy} = -\pi^T logP$$

$$\rho_{L2} = \sum f_\theta.param$$

$$Loss = (Q - V)^2 - \pi^T logP + \lambda_{L2}\rho_{L2}$$

$f_\theta.param$	神经网络的参数
ρ_{L2}	L2 正则化值
λ_{L2}	L2 正则化系数，本设计中取 0.01

代码实现如下：

```
// ValuePolicyLoss 类的构造函数和 forward 方法的定义
ValuePolicyLoss::ValuePolicyLoss(float lambda_reg) : lambda_reg(lambda_reg) {
}

torch::Tensor ValuePolicyLoss::forward(torch::nn::Module &model, torch::Tensor p_cnn, torch::Tensor
v_cnn,
                                       torch::Tensor p_mcts, torch::Tensor v_mcts) {
    torch::Tensor l2_reg = torch::tensor(0.0);

    // 计算 L2 正则化项
    for (const auto& param : model.parameters()) {
        l2_reg = l2_reg + param.norm(2);
    }

    // 展平输入张量
    auto p_cnn_tensor = torch::flatten(p_cnn);
    auto p_mcts_tensor = torch::flatten(p_mcts);
    auto v_cnn_tensor = torch::flatten(v_cnn);
    auto v_mcts_tensor = torch::flatten(v_mcts);
```

```
    // 计算损失
    auto loss = torch::norm(v_mcts_tensor - v_cnn_tensor, 2).pow(2) -
                torch::dot(p_mcts_tensor, torch::log(p_cnn_tensor)) +
                lambda_reg * l2_reg;

    return loss;
}
```

3.2.2.2 优化器

为避免模型的过拟合，本设计中采用了 Adam 自适应学习率（Adaptive Moment Estimation）算法作为优化器，此优化器的运用在深度学习算法中极为常见 Adam 算法结合了动量法（Momentum）和 RMSprop（Root Mean Square Propagation）两种梯度下降算法的优点。Adam 算法在计算学习率时考虑了第一时刻（均值）与第二时刻（未平方的方差）的梯度，从而适应的调整每个参数的学习率

$$g_t = \nabla_\theta Loss(\theta)$$

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2$$

$$\overline{m_t} = \frac{m_t}{1 - \beta_1^t}$$

$$\overline{v_t} = \frac{v_t}{1 - \beta_2^t}$$

$$\theta = \theta - \alpha * \frac{\overline{m_t}}{\sqrt{\overline{v_t}} + \varepsilon}$$

$Loss(\theta)$	损失函数
g_t	当前参数的梯度
λ_{L2}	L2 正则化系数，本设计中取 0.01
m_t	第 t 次迭代的一阶矩估计
v_t	第 t 次迭代的二阶矩估计
β_1	算法超参数，用于控制一阶矩估计中过去梯度与当前梯度的影响
β_2	算法超参数，用于控制二阶矩估计中过去梯度与当前梯度的影响
$\overline{m_t}$	偏差修正一阶矩估计
$\overline{v_t}$	偏差修正二阶矩估计

t	当前迭代次数
α	学习率
ε	极小量，防止 0 除数

此算法在程序中的实现较为简单，相关代码如下：

```
torch::optim::Adam optimizer(gobangCNN.parameters(), torch::optim::AdamOptions(1e-4));
```

3.2.3 神经网络训练过程

对于每一次落子，先将当前局面数据输入神经网络得到下一步行动的价值策略分布，然后进行蒙特卡洛树搜索得到新的价值策略分布，计算损失函数更新神经网络参数

对于整个游戏过程，初始时随机创建两个模型 model1 与 model2 进行对弈，采取三局两胜制，获胜一方将继续与新随机创建的模型进行对弈，以此类推，迭代多次后得到训练充分的模型

相关代码如下：

```
#include "board_paint.h"

#define train_times 20

void createNewModel(std::string name) {
    GobangCNN model;
    torch::serialize::OutputArchive output;
    model.save(output);
    output.save_to(name);
}

int main() {
    createNewModel("model1.pth");
    createNewModel("model2.pth");
    int win1 = 0;
    int win2 = 0;
    for (int k = 0; k < train_times; ++k) {
        for (int i = 0; i < 3; ++i) {
            reset_HCD(REAL);
            resetBoard(REAL);
            resetBoard(VIRTUAL);
            reset_HCD(VIRTUAL);
            int now = 2;
```

```
                chess(7,7,B_BLACK,REAL);
                drawBoard();
                creatRoot(7,7,B_BLACK);
                drawBoard();
                while (game_terminate(REAL) == 0) {
                    if (now == 1) {
                        EstimateAndBack("model1.pth","model1.pth");
                        inheritRoot();
                        drawBoard();
                        now = 2;
                    } else if (now == 2) {
                        EstimateAndBack("model2.pth","model2.pth");
                        inheritRoot();
                        drawBoard();
                        now = 1;
                    }
                }
                if (game_terminate(REAL) == 1) {
                    win1++;
                }
                if (game_terminate(REAL) == 2) {
                    win2++;
                }
                if (game_terminate(REAL) == 3) {
                    i--;
                }
            }
            if (win1 < win2) {
                torch::serialize::InputArchive input;
                GobangCNN model;
                input.load_from("model2.pth");
                model.load(input);
                torch::serialize::OutputArchive output;
                model.save(output);
                output.save_to("model1.pth");
                createNewModel("model2.pth");
            } else if (win1 > win2) {
                createNewModel("model2.pth");
            }
        }
}
```

第四章 算法运行测试

本实验运行平台如下：

系统	Windows 10 专业版 21H2
CPU	AMD Ryzen 9 7940H w/Radeon 780M Graphics 八核
内存	16GB DDR5 4800MHZ
GPU	NVIDIA GeForce RTX 4060 Laptop GPU 8GB
硬盘	Fanxiang S790 4TB(PCIe 4.0 x4)

代码采用 C/C++编写，神经网络采用 Libtorch 框架

完整代码见此处(https://github.com/Mgepahmge/c2023-a)

经过数十小时的训练，我们得到了若干个神经网络模型，从中随机挑选十个模型，编号为模型 1、模型 2 ..... 模型 10，这十个模型经历的迭代次数依次增加，令这十个模型两两一组进行比赛，最终得到比分结果如下：

模型	1	2	3	4	5	6	7	8	9	10	总分
1		-2	0	-2	-2	-1	-2	-2	-1	-2	-14
2	1		-2	0	-1	0	0	-2	-2	-1	-7
3	0	1		0	1	-2	-2	0	-1	-2	-5
4	2	0	0		1	-1	-1	-2	0	-2	-3
5	0	2	-1	-1		2	0	1	-1	-1	1
6	1	0	2	1	-2		0	-2	0	-2	-2
7	2	0	2	1	1	0		-1	0	-2	3
8	2	2	0	0	-1	2	1		0	0	6
9	1	2	1	2	1	0	1	0		1	9
10	2	1	2	2	1	2	2	0	1		13

从表格可以看出，随着训练迭代次数的增加，训练时间较长的模型要优于训练时间较短的模型。

传统的智能五子棋算法通常基于极大极小值搜索和 Alpha-Beta 剪枝算法来构建，这些算法足以战胜绝大多数业余棋手。在本实验中，我们将上述 10 个训练出的深度神经网络模型与基于极大极小值搜索和 Alpha-Beta 剪枝的智能五子棋算法进行对比，采用五子棋对弈的形式来评估它们的效能。

在标准的五子棋比赛规则下，通常采用五局三胜制来决定胜负，即在五局比赛中，谁先赢得三局，则该方获得最终的胜利。基于这一规则，我们对前述的三种算法进行了比较分析。具体的比较结果如下所示：

网络模型	极大极小值搜索	Alpha-Beta 剪枝算法
1	1:3	0:3
2	0:3	1:3
3	2:3	1:3
4	1:3	2:3
5	3:2	3:1
6	3:1	3:2
7	3:0	3:1
8	3:0	3:0
9	3:0	3:0
10	3:0	3:0

由结果可看出本设计的算法相较传统算法取得了更好的成绩，十个模型均取得了比赛的胜利

基于以上测试，可认为本设计的算法已经具有较好的博弈能力

第五章 实验设计中的其它问题

5.1 蒙特卡洛树搜索算法的实现

5.1.1 博弈树的实现

5.1.1.1 节点结构

```
//节点结构
typedef struct node {
    //父节点
    struct node* father;
    //子节点
    struct node* children[childrenNum];
节点其它属性.....
} Node;
```

每个节点结构指针代表树中的一个节点，每个节点存储有它们的父节点与子节点的信息

5.1.1.2 节点移动

定义一个指针 target，作为当前访问节点

```
extern Node* target;
```

对于 target 可以调用其父节点与子节点实现沿树向上或者向下移动

```
//向上移动
target = target -> father;
//向下移动
target = target -> children[i];
```

5.1.1.3 节点创建与销毁

根节点为博弈树中的最上层节点，因此在创建根节点时，创建一个父节点为空指针的节点

```
//创建根节点
void creatRoot(int x,int y,int type) {
    target = (Node*) malloc(sizeof(Node));
    target -> father = NULL;
    for (int j = 0; j < childrenNum; ++j) {
        target -> children[j] = NULL;
    }
初始化其它属性.....
}
```

其余节点在创建时将以当前节点为父节点来创建子节点，程序首先从头遍历当前节点的子节点数组，寻找第一个空位来创建一个新的节点，从而实现博弈树的向下拓展

```
//为当前节点创建一个子节点
void createNode(int x,int y,double Q,double v,int N,double P,double P_real,int type) {
    for (int i = 0; i < childrenNum; ++i) {
        if (target->children[i] == NULL) {
            Node* newNode = (Node*) malloc(sizeof(Node));
            newNode -> father = target;
            for (int j = 0; j < childrenNum; ++j) {
                newNode -> children[j] = NULL;
            }
初始化其它属性……
            target -> children[i] = newNode;
            break;
        }
    }
}
```

棋盘状态每一次更新都将导致博弈树的改变，并且在自我对弈过程中也会产生大量的无用节点，这些节点将占用大量资源且对程序的正常运行产生影响（例如若新的根节点来源于之前的树，若其父节点未被抛弃则无法判断此节点是否为根节点），因此销毁节点是程序的一个重要环节，本设计中采用一个递归函数来销毁无用节点

```
//释放内存
void freeTree(Node* input) {

    if (input == NULL) return;

    for (int i = 0; i < childrenNum; i++) {
        if (input->children[i] != NULL) {
            freeTree(input->children[i]);
        }
    }
    free(input);
}
```

此函数将从当前节点开始向下遍历所有节点并释放它们的内存，达到一次性销毁一个分支的效果，但需要注意的是在释放内存后需要把已销毁节点的最上层节点设置为空指针

5.1.2 传统 MCTS 的实现

```
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include "MCTS.h"

//UCT 算法超参数
#define c 0.5

//当前访问节点
Node* target;

//设定访问节点
void setTarget(Node* input) {
    target = input;
}

//为当前节点创建一个子节点，可自主设定价值与访问次数
void createNode(int value,int S_times) {
    for (int i = 0; i < childrenNum; ++i) {
        if (target->children[i] == NULL) {
            Node* newNode = (Node*) malloc(sizeof(Node));
            newNode -> father = target;
            newNode -> value = value;
            newNode -> S_times = S_times;
            for (int j = 0; j < childrenNum; ++j) {
                newNode -> children[j] = NULL;
            }
            target -> children[i] = newNode;
            break;
        }
    }
}

//判断当前节点是否是叶节点
int is_leaf() {
    for (int i = 0; i < childrenNum; ++i) {
        if (target->children[i] != NULL) {
            return 0;
        }
    }
    return 1;
}
```

```
//根据UCT 算法结果返回UCT 算法计算值最大的子节点的索引
int max_mark(double input[childrenNum]) {
    double max = 0;
    for (int i = 0; i < childrenNum; ++i) {
        max = max > input[i] ? max : input[i];
    }
    for (int i = 0; i < childrenNum; ++i) {
        if (max == input[i]) {
            return i;
        }
    }
    return 0;
}

//判断当前节点是否为根节点
int is_root() {
    if (target->father == NULL) {
        return 1;
    } else {
        return 0;
    }
}

//反向传播更新访问次数
void back_times() {
    while (!is_root()) {
        target->S_times++;
        target = target->father;
    }
}

//反向传播更新价值
void back_value() {
    while (!is_root()) {
        Node* now = target;
        target = target->father;
        target -> value += now->value;
    }
}

/*
* 最大置信上界算法搜索(Upper Confidence Bound Apply to Tree, UCT)
* 每一轮搜索从当前节点开始，向下搜索 n 层(若 n=-1，则搜索至叶节点结束)
```

```
* 搜索开始时先反向传播以更新当前节点以前节点的访问次数
* 搜索时每搜索至一个节点时增加该节点的访问次数
* 采用以上更新方式以避免初次开始搜索时根节点访问次数为0 导致计算错误
*/
void UCT(int n) {
    //存储 UCT 算法的计算结果
    double result[childrenNum];
    if (n==-1) {
        //初次更新访问次数
        if (is_root()) {
            target->S_times++;
        } else {
            //反向传播更新访问次数
            Node * now = target;
            back_times();
            target = now;
        }
        //搜索至叶节点结束
        while (!is_leaf()) {
            //每次初始化 result 数组
            for (int i = 0; i < childrenNum; ++i) {
                result[i] = 0;
            }
            //UCT 算法计算
            for (int i = 0; i < childrenNum; ++i) {
                if (target->children[i] != NULL) {
                    //若访问次数为0，则使其值为极大值，代表必定访问
                    if (target->children[i]->S_times == 0) {
                        result[i] = DBL_MAX;
                    } else {
                        result[i] = (double)target->children[i]->value/target->children[i]->S_times+c*
sqrt((double) log((double)target->S_times)/target->children[i]->S_times);
                    }
                } else {
                    break;
                }
            }
            setTarget(target->children[max_mark(result)]);
            //正向更新访问次数
            target->S_times++;
        }
    } else {
        //初次更新访问次数
        if (is_root()) {
```

```
                target->S_times++;
            } else {
                //反向传播更新访问次数
                Node * now = target;
                back_times();
                target = now;
            }
            //向下搜索 n 层
            for (int i = 0; i < n; ++i) {
                //每次初始化 result 数组
                for (int j = 0; j < childrenNum; ++j) {
                    result[j] = 0;
                }
                //UCT 算法计算
                for (int j = 0; j < childrenNum; ++j) {
                    if (target->children[j] != NULL) {
                        //若访问次数为0，则使其值为极大值，代表必定访问
                        if (target->children[j]->S_times == 0) {
                            result[j] = DBL_MAX;
                        } else {
                            result[j] = (double)target->children[j]->value / target->children[j]->S_times + c
* sqrt((double) log((double)target->S_times) / target->children[j]->S_times);
                        }
                    } else {
                        break;
                    }
                }
                setTarget(target->children[max_mark(result)]);
                //正向更新访问次数
                target->S_times++;
            }
        }
}

//输出 target 向量以供其它文件访问
Node* outputTarget() {
    return target;
}
```

上述代码展示了传统的蒙特卡洛树搜索算法的实现（除去模拟步骤），而本设计中采用的 MCTS 算法与上述算法不同，但上述算法可检验本设计中采用的博弈树结构的可行性

5.2 神经网络设计原理

5.2.1 基于价值的强化学习原理

5.2.1.1 有模型学习

通常，强化学习问题被描述为一个马尔可夫决策过程（Markov Decision Process, MDP）。在此框架中，代理（机器）被置于一个环境中，其中每个状态是对环境感知的描述。代理通过执行动作来影响环境，导致环境根据某种概率分布转移到新的状态。与状态转移并行的是环境通过某种机制提供的反馈奖赏。强化学习的基础构成包括四个主要元素：状态（S）、动作（A）、转移概率（P）以及奖赏函数（R）。状态是代理对环境的感知，其集合构成了状态空间。动作是代理所采取的干预措施，其集合构成了动作空间。转移概率是在特定动作执行后，从当前状态转移到另一状态的概率值。奖赏函数是在状态转移期间环境对代理提供的反馈机制，通常体现为状态-价值映射函数。相关公式如下：

T 步累计奖赏	执行该策略 T 步所得奖赏平均值的期望。	$E\left[\frac{1}{T}\sum_{t=1}^{T} r_t\right] = \frac{1}{T}(r_1 + r_2 + r_3 + \cdots + r_T)$
折扣累计奖赏	策略一直执行的加权累计奖赏期望，且越往后奖赏权重越低。	$E\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right] = r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \cdots$
状态价值函数	$V^{\pi}(s)$表示从状态 s 出发，使用策略带来的累计奖赏。	$V_T^{\pi} = E\left[\frac{1}{T}\sum_{t=1}^{T} r_t \mid s_0 = s\right]$
		$V_{\gamma}^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s\right]$
状态-动作价值函数	$Q^{\pi}(s，a)$表示从状态s出发。执行动作a后再使用策略π带来的累计奖赏。	$Q_T^{\pi} = E\left[\frac{1}{T}\sum_{t=1}^{T} r_t \mid s_0 = s, a_0 = a\right]$
		$Q_{\gamma}^{\pi} = E\left[\frac{1}{T}\sum_{t=0}^{\infty} \gamma_t r_{t+1} \mid s_0 = s, a_0 = a\right]$

这里引入强化学习中的一个重要概念——贝尔曼方程，贝尔曼方程利用了马尔可夫决策过程的递归特性，即未来的奖赏可以表示为当前的奖赏加上对未来期望奖

赏的折现。

贝尔曼期望方程：

$$V(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) V(s')$$

$V(s)$	状态 s 的价值
$R(s)$	在状态 s 获得的即时奖赏
γ	折扣因子
$P(s' \mid s, a)$	从状态 s 通过动作 a 转移到状态 s'的概率
$V(s')$	后续状态 s'的价值

利用贝尔曼方程可对一个 MDP 过程使用动态规划的方法更新价值状态函数。

贝尔曼最优性方程：

优化目标$\pi^*(s)$可以记为

$$\pi^*(s) = \underset{\pi}{\operatorname{argmax}} V^{\pi}(s)$$

分别记$\pi^*(s)$对应的状态价值函数和状态-动作价值函数为$V^*(s)$和$Q^*(s,a)$，于是有：

$$V^*(s) = \max_{a} Q^*(s,a)$$

$$V^*(s) = \max_{a} E\left[r(s' \mid s,a) + \gamma V^{\pi}(s') \mid s_0 = s\right]$$

$$= \max_{a \epsilon A(S)} \sum_{s' \epsilon S} p(s' \mid s,a) \left[r(s' \mid s, \pi(a)) + \gamma V^{\pi}(s')\right]$$

$$Q^*(s,a) = E\left[r(s' \mid s,a) + \gamma \max_{a'} Q^*(s',a') \mid s_0 = s, a_0 = a\right]$$

$$= \sum_{s' \epsilon S} p(s' \mid s, \pi(s)) \left[r(s' \mid s, \pi(s)) + \gamma \max_{a' \epsilon A(S)} Q^*(s',a')\right]$$

利用贝尔曼方程递归计算状态价值函数：

$$V'(s) = \sum_{s' \epsilon S} p(s' \mid s, \pi(s)) \left[r(s' \mid s,a) + \gamma V(s')\right]$$

利用上述公式更新价值状态函数，然后利用新的价值状态函数更新策略，反复迭代以收敛至最优策略：

$$\pi'(s) = \underset{a \epsilon A}{\operatorname{argmax}} Q^{\pi}(s,a)$$

$$if\ \pi'(s) = \pi(s)\ break\ else\ \pi(s) = \pi'(s)$$

在策略迭代过程中，每次策略优化后都需要进行策略评估，这一步骤往往是时间消耗较大的环节。考虑到策略的优化与价值函数的增强在本质上是相同的，策略优化可以被重构为价值函数的提升。基于此理念，值迭代算法被提出：

$$V'(s) = \max_{a \epsilon A(S)} \sum_{s' \epsilon S} p\left(s' \middle| s, \pi(s)\right)\left[r\left(s' \middle| s, \pi(s)\right) + \gamma V(s')\right]$$

5.2.1.2 免模型学习

在实际应用中，许多马尔可夫决策过程（MDP）的问题面临着缺乏完整模型的挑战，即环境的转移概率、奖赏函数不明确，甚至环境中的状态总数也无法准确知晓。在这种情况下，依赖于环境模型的学习算法（如动态规划）变得不切实际。相应地，不依赖于环境模型的学习方法被称为免模型学习，这种方法在处理实际问题时通常更为困难。

蒙特卡洛方法是一种免模型学习技术，它通过采样完成的序列（或称为情节）来估计值函数。在免模型学习中，蒙特卡洛方法通过探索来获取状态转移的基本信息。执行某策略 T 步可以得到：

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, \cdots, r_{T-1}, s_T$$

在处理马尔可夫决策过程（MDP）时，动态规划是一种有效的方法来更新状态价值函数，这要求对环境的完整模型有准确的了解。然而，在免模型学习的情境中，由于缺乏关于环境的全面信息，动态规划方法无法直接应用。相反，免模型学习依赖于从实际交互中采样得到的数据来估计状态价值函数。在这种情况下，主要的采样估计方法包括蒙特卡洛策略评估和时序差分方法。

蒙特卡洛策略评估法：使用G_t作为蒙特卡洛探索的完全折扣奖赏

$$G_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_T$$

$$V^{\pi}(s) = E[G_t | S_t = s]$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

时序差分法：使用$r_{t+1} + \gamma V(S_{t+1})$作为完全折扣奖赏的估计。

$$V(S_t) \leftarrow V(S_t) + \alpha(r_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

价值函数的学习方法可以根据环境条件被划分为两大类，即全备份（Full Backup）（即动态规划，Dynamic Programming）和采样备份（Sample Backup）（即时序

差分，Temporal Difference）。此外，基于探索与优化时所采用动作的一致性，价值函数学习进一步分为同策略（on-policy）和异策略（off-policy）两种方式，这导致了多种强化学习算法的产生。除动态规划方法如策略评估、策略迭代和值迭代外，还发展了基于采样更新的时序差分（TD）算法、Sarsa 算法和 Q 学习算法等。这些算法各自的递推公式如下：

全备份	
迭代策略评估	$V(s) \leftarrow E\left[r\left(s^{'}\middle\|s,a\right)+\gamma V\left(s^{'}\right)\middle\|s\right]$
Q-策略迭代	$Q(s,a) \leftarrow E\left[r\left(s^{'}\middle\|s,a\right)+\gamma V\left(s^{'}\right)\middle\|s,a\right]$
Q-价值迭代	$Q(s,a) \leftarrow E\left[r\left(s^{'}\middle\|s,a\right)+\gamma \max_{a \epsilon A(s)} Q^{*}(s^{'},a^{'})\middle\|s,a\right]$

采样备份	
时序差分学习	$V(s) \overset{\alpha}{\leftarrow} r\left(s^{'}\middle\|s,a\right)+\gamma V\left(s^{'}\right)$
Sarsa	$Q(s,a) \overset{\alpha}{\leftarrow} r\left(s^{'}\middle\|s,a\right)+\gamma Q\left(s^{'},a^{'}\right)$
Q-学习	$Q(s,a) \overset{\alpha}{\leftarrow} r\left(s^{'}\middle\|s,a\right)+\gamma \max_{a \epsilon A(s)} Q^{*}(s^{'},a^{'})$

5.2.1.3 五子棋博弈的价值状态函数

基于前述分析，文章中选用神经网络作为状态价值函数的近似工具，并利用蒙特卡洛策略评估方法作为更新机制。该神经网络的目标是最小化估计值与奖励之间的均方误差。

$$z = G_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_T$$

$$l_v = (z - v)^2$$

从而得到神经网络的损失函数：

$$Loss = (Q - V)^2 - \pi^T log P + \lambda_{L2} \rho_{L2}$$

5.2.2 神经网络的实现与相关问题

5.2.2.1 神经网络输入数据的数据结构

如前文所述，本设计采用双方棋手过去八手落子的落子数据作为神经网络的输入，

因此在程序中定义了两个数组用于存储过去八次对弈的棋盘状态

```
//历史棋盘数据(真实)
int Board_History_Real[8][SIZE][SIZE];

//历史棋盘数据(虚拟)
int Board_History_Virtual[8][SIZE][SIZE];
```

其中真实棋盘数据用于存储实际棋盘的落子情况，虚拟棋盘数据用于存储模拟过程中的落子状况并可能随时重置到与真实数据相同的状态

存储棋盘数据时，将最早一次的棋盘数据弹出，其余数据依次向后移位，同时写入新一次数据

```
//记录历史落子数据
void write_chessboard_data(int D_type) {
    if (D_type == REAL) {
        for (int i = 7; i > 0; --i) {
            for (int j = 0; j < SIZE; ++j) {
                for (int k = 0; k < SIZE; ++k) {
                    Board_History_Real[i][j][k] = Board_History_Real[i-1][j][k];
                }
            }
        }
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                Board_History_Real[0][i][j] = board_real[i][j];
            }
        }
    } else {
        for (int i = 7; i > 0; --i) {
            for (int j = 0; j < SIZE; ++j) {
                for (int k = 0; k < SIZE; ++k) {
                    Board_History_Virtual[i][j][k] = Board_History_Virtual[i-1][j][k];
                }
            }
        }
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                Board_History_Virtual[0][i][j] = board_virtual[i][j];
            }
        }
    }
}
```

将数据输入神经网络之前需对数据进行转换，转换后的数据结构见第三章

相关代码如下：

```
//数据转录
void data_trans(int B_type,int D_type) {
    if (D_type == REAL) {
        for (int k = 0; k < 8; ++k) {
            for (int i = 0; i < SIZE; ++i) {
                for (int j = 0; j < SIZE; ++j) {
                    NN_input[2*k][i][j] = Board_History_Real[k][i][j] == B_type ? 1: 0;
                    NN_input[2*k+1][i][j] = Board_History_Real[k][i][j] == (B_type == B_BLACK ?
B_WHITE : B_BLACK) ? 1 : 0;
                }
            }
        }
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                NN_input[16][i][j] = B_type;
            }
        }
    } else {
        for (int k = 0; k < 8; ++k) {
            for (int i = 0; i < SIZE; ++i) {
                for (int j = 0; j < SIZE; ++j) {
                    NN_input[2 * k][i][j] = Board_History_Virtual[k][i][j] == B_type ? 1 : 0;
                    NN_input[2 * k + 1][i][j] =
                            Board_History_Virtual[k][i][j] == (B_type == B_BLACK ? B_WHITE :
B_BLACK) ? 1 : 0;
                }
            }
        }
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                NN_input[16][i][j] = B_type;
            }
        }
    }
}
```

5.2.2.2 神经网络结构的实现

本设计的早期代码使用 Python 中的 Pytorch 框架来实现神经网络，其代码如下：

```
import torch
from torch import nn
```

```
class GobangCNN(nn.Module):
    def __init__(self):
        super(GobangCNN, self).__init__()
        self.publicConv = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=17, out_channels=32, kernel_size=(3, 3), padding=1, stride=1),
            torch.nn.BatchNorm2d(32),
            torch.nn.ReLU(),
            torch.nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(3, 3), padding=1, stride=1),
            torch.nn.BatchNorm2d(64),
            torch.nn.ReLU(),
            torch.nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3, 3), padding=1,
stride=1),
            torch.nn.BatchNorm2d(128),
            torch.nn.ReLU()
        )
        self.value = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=128, out_channels=2, kernel_size=(1, 1), padding=0, stride=1),
            torch.nn.BatchNorm2d(2),
            torch.nn.ReLU(),
            torch.nn.Flatten(),
            torch.nn.Linear(2 * 15 * 15, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 15 * 15),
            torch.nn.Tanh(),
            torch.nn.Unflatten(1, (15, 15))
        )
        self.policy = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=128, out_channels=4, kernel_size=(1, 1), padding=0, stride=1),
            torch.nn.BatchNorm2d(4),
            torch.nn.ReLU(),
            torch.nn.Flatten(),
            torch.nn.Linear(4 * 15 * 15, 15 * 15),
            torch.nn.Softmax(),
            torch.nn.Unflatten(1, (15, 15))
        )

    def forward(self, x):
        y = self.publicConv(x)
        v = self.value(y)
        p = self.policy(y)
        return v, p


class ValuePolicyLoss(nn.Module):
```

```
    def __init__(self, model, p_cnn, v_cnn, p_mcts, v_mcts, lambda_reg=0.01):
        super(ValuePolicyLoss, self).__init__()
        self.model = model
        self.p_cnn = p_cnn
        self.v_cnn = v_cnn
        self.p_mcts = p_mcts
        self.v_mcts = v_mcts
        self.lambda_reg = lambda_reg

    def forward(self):
        l2_reg = None
        for param in self.model.parameters():
            if l2_reg is None:
                l2_reg = param.norm(2)
            else:
                l2_reg = l2_reg + param.norm(2)
        p_cnn_tensor = torch.Tensor(self.p_cnn)
        p_mcts_tensor = torch.Tensor(self.p_mcts)
        v_cnn_tensor = torch.Tensor(self.v_cnn)
        v_mcts_tensor = torch.Tensor(self.v_mcts)
        loss = torch.norm(v_mcts_tensor - v_cnn_tensor, 2) ** 2 - torch.dot(p_mcts_tensor,
                                                                            torch.log(

p_cnn_tensor)) + self.lambda_reg * l2_reg
        return loss
```

本设计的早期程序尝试使用 Python C API 将使用 Python 编写的神经网络模块与使用 C 语言编写的 MCTS 模块进行链接

Python 的 C API 是一套功能强大的接口，它允许开发者在 C 或 C++中直接与 Python 解释器进行交互。这个 API 提供了广泛的功能，使得程序员可以创建或扩展 Python 模块，从而增强 Python 的功能。通过这个 API，可以实现从简单的数据类型转换到复杂的对象操作。它还允许开发者在更低的级别上管理 Python 对象的内存和生命周期。这一接口的使用使得 Python 能够与其他语言编写的程序或库紧密集成，大大提升了其灵活性和性能。因此，Python 的 C API 是连接 Python 世界和 C 语言生态系统的关键桥梁，特别适用于性能要求较高的应用场景

以下是一段示例程序：

```
#include <Python.h>
```

```
// 定义 greet 函数
static PyObject* py_greet(PyObject* self, PyObject* args) {
    const char* name;

    // 解析传入的 Python 字符串参数
    if (!PyArg_ParseTuple(args, "s", &name)) {
        return NULL;
    }

    // 创建返回的字符串
    char greeting[50];
    sprintf(greeting, "Hello, %s!", name);

    // 返回 Python 字符串
    return Py_BuildValue("s", greeting);
}

// 定义模块的方法列表
static PyMethodDef MyMethods[] = {
    {"greet", py_greet, METH_VARARGS, "Greet someone."},
    {NULL, NULL, 0, NULL}   // 哨兵，表示方法结束
};

// 定义模块
static struct PyModuleDef mymodule = {
    PyModuleDef_HEAD_INIT,
    "mymodule",   // 模块名
    NULL,         // 模块文档
    -1,           // 模块保持状态的大小，-1 表示不保存状态
```

```
    MyMethods
};

// 初始化模块
PyMODINIT_FUNC PyInit_mymodule(void) {
    return PyModule_Create(&mymodule);
}
```

但碍于 Python C API 相关的资料极度匮乏，且使用 Python C API 时，需采用 C 语言的方式对 Python 程序进行内存管理，从而失去了 Python 自动化管理内存的特性，在应对诸如神经网络的较为复杂且使用了第三方库的程序时，出现了难以捕捉的未知问题，从而导致本设计中将 Python 中的神经网络程序接入 C 语言中的 MCTS 程序的计划搁置，因此本设计最终采用 C++中的 Libtorch 库实现神经网络 LibTorch 是 PyTorch 的 C++库版本，它提供了 PyTorch 的核心功能，包括张量操作、自动微分、模型定义等

模型定义代码如下：

```
#include "CNN.h"

// GobangCNN 类的构造函数的定义
GobangCNN::GobangCNN() {
    // 定义和初始化网络层
    publicConv = torch::nn::Sequential(
            torch::nn::Conv2d(torch::nn::Conv2dOptions(17, 32, 3).padding(1)),
            torch::nn::BatchNorm2d(32),
            torch::nn::ReLU(),
            torch::nn::Conv2d(torch::nn::Conv2dOptions(32, 64, 3).padding(1)),
            torch::nn::BatchNorm2d(64),
            torch::nn::ReLU(),
            torch::nn::Conv2d(torch::nn::Conv2dOptions(64, 128, 3).padding(1)),
            torch::nn::BatchNorm2d(128),
            torch::nn::ReLU()
    );

    value = torch::nn::Sequential(
            torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 2, 1)),
            torch::nn::BatchNorm2d(2),
            torch::nn::ReLU(),
```

```
                torch::nn::Flatten(),
                torch::nn::Linear(2 * 15 * 15, 64),
                torch::nn::ReLU(),
                torch::nn::Linear(64, 15 * 15),
                torch::nn::Tanh()
    );

    policy = torch::nn::Sequential(
                torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 4, 1)),
                torch::nn::BatchNorm2d(4),
                torch::nn::ReLU(),
                torch::nn::Flatten(),
                torch::nn::Linear(4 * 15 * 15, 15 * 15),
                torch::nn::Softmax(1)
    );

    // 注册模块
    register_module("publicConv", publicConv);
    register_module("value", value);
    register_module("policy", policy);
}

// GobangCNN 类的 forward 方法的定义
std::tuple<torch::Tensor, torch::Tensor> GobangCNN::forward(torch::Tensor x) {
    auto y = publicConv->forward(x);
    auto v = value->forward(y);
    auto p = policy->forward(y);

    v = unflatten(v, 15, 15);
    p = unflatten(p, 15, 15);

    // 遍历并更新 v 和 p Tensor 的值
    for (int i = 0; i < 15; ++i) {
        for (int j = 0; j < 15; ++j) {
            if (x[0][0][i][j].item<float>() != 0) {
                v[0][i][j] = 0;
                p[0][i][j] = 0;
            }
            if (x[0][1][i][j].item<float>() != 0) {
                v[0][i][j] = 0;
                p[0][i][j] = 0;
            }
        }
    }
```

```
    return std::make_tuple(v, p);
}
```

以上代码定义了神经网络的基本结构，并且对输出数据进行了调整（将棋盘非空位置的数据设置为 0）

Libtorch 中并未内置 unflatten 方法，因此需要使用 view 方法手动实现此功能

```
// 手动 Unflatten
torch::Tensor unflatten(torch::Tensor input, int64_t height, int64_t width) {
    // 获取 batch_size
    int64_t batch_size = input.size(0);

    // 使用 view 方法重新塑形 Tensor
    return input.view({batch_size, height, width});
}
```

相应的损失函数的实现：

```
// ValuePolicyLoss 类的构造函数和 forward 方法的定义
ValuePolicyLoss::ValuePolicyLoss(float lambda_reg) : lambda_reg(lambda_reg) {
}

torch::Tensor ValuePolicyLoss::forward(torch::nn::Module &model, torch::Tensor p_cnn, torch::Tensor
v_cnn,
                                       torch::Tensor p_mcts, torch::Tensor v_mcts) {
    torch::Tensor l2_reg = torch::tensor(0.0);

    // 计算 L2 正则化项
    for (const auto& param : model.parameters()) {
        l2_reg = l2_reg + param.norm(2);
    }

    // 展平输入张量
    auto p_cnn_tensor = torch::flatten(p_cnn);
    auto p_mcts_tensor = torch::flatten(p_mcts);
    auto v_cnn_tensor = torch::flatten(v_cnn);
    auto v_mcts_tensor = torch::flatten(v_mcts);

    // 计算损失
    auto loss = torch::norm(v_mcts_tensor - v_cnn_tensor, 2).pow(2) -
                torch::dot(p_mcts_tensor, torch::log(p_cnn_tensor)) +
                lambda_reg * l2_reg;

    return loss;
}
```

C++程序与 C 语言程序具有较好的兼容性，因此此神经网络程序可与 MCTS 程序流畅衔接

第六章 总结与展望

6.1 总结

在本课程设计中，我们成功实现了一个基于 AlphaZero 框架的五子棋 AI。这个 AI 结合了策略价值网络和蒙特卡洛树搜索（MCTS），能够通过自我博弈不断提升其博弈水平。尽管已经展现出了学习和掌握五子棋技巧的能力，但在实验过程中，我们发现还有进一步提升算法性能的空间。主要优化方向包括：

1. 增加自我博弈的迭代次数：通过增加自我博弈的迭代次数，AI 能够在更复杂和多样的博弈情景中学习，从而提高其整体实力。

2. 提升 MCTS 的模拟次数：增加 MCTS 过程中的模拟次数可以使 AI 在决策时考虑更多可能的走法，从而提升决策质量。

3. 采用更复杂的网络结构：当前使用的网络较为简单。考虑到 AlphaZero 采用的是更复杂的网络，我们可以通过增加卷积层和全连接层来增强 AI 的学习和预测能力。同时，需要注意避免过拟合。

4. 调整学习率：合理的学习率对网络的有效学习至关重要，适当调整学习率可以帮助 AI 更快地达到最优性能。

5. 实现并行化的 MCTS：目前使用的 MCTS 基于串行结构，限制了其并行计算能力。改进 MCTS 为分布式并行计算模式，可以提高搜索效率，使 AI 在相同时间内达到更高的博弈水平。

6.2 展望与感想

展望未来，这个基于 AlphaZero 框架的五子棋 AI 项目为我提供了宝贵的学习经验和技术洞见。我对人工智能和机器学习的能力有了更深的认识，特别是在如何让

机器通过自我学习和探索来掌握复杂技能的领域。在项目的进展中，我意识到尽管我们的 AI 已经取得了显著的成绩，但人工智能的发展潜力远未被完全挖掘。

未来，我希望能进一步提升这个五子棋 AI 的性能，使其能够在更复杂的博弈环境中做出更加精准和高级的决策。此外，探索 AI 在不同领域的应用，如图像识别、自然语言处理或预测分析，将是我接下来的研究方向。这些领域的探索不仅能够扩展我的技术视野，也有助于我更好地理解和运用人工智能技术。

从个人感想来看，这个项目不仅加深了我对人工智能和机器学习的理解，也激发了我对这一领域未来可能性的好奇心。看到 AI 从无到有，逐渐学会并精通五子棋的过程，让我对技术的力量和创新的价值有了更加深刻的感悟。我期待着将这些知识和经验应用到未来的项目中，继续在这一激动人心的技术前沿领域探索和成长。

参考文献

[1] McCulloch.W.S and Pitts.W.A logical calculus of the ideas immanent in nervous activity[J].The bulletin of mathematical biophysics, 1943, 5(4): 115-133F.

[2] Rosenblatt.Perceptron Simulation Experiments[J].Proceedings of the Ire, 1960.48(3):301-309.

[3] 孙志军，薛雷，许阳明，王正，深度学习研究综述[J].计算机应用研究，2012, 29(8):2806-2810.

[4] 刘建伟，刘媛，罗雄麟，深度学习研究进展[J].计算机应用研究，2014，31（7）：1921-1942.

[5] G.E.Hinton，S.Osindero，Y.W Teh.A Fast Learning Algorithm for Deep Belief Nets[J].Neural Computation, 2006, 18(7): 1527-1554.

[6] M. Ranzato, Y. Boureau, S. Chopra, and Y. LeCun. A unified energy-based framework for unsupervised learning[J]. Proc. Conference on Al and Statistics (Al-Stats), 2007.

[7] Wang F Y, Zhang H, and Liu D. Adaptive dynamic programming: an introduction[J]. IEEE Computational Intelligence Magazine, 2009, 4(2):39-47.

[8] Werbos PJ. Using ADP to Understand and Replicate Brain Intelligence: the Next Level DesignfCI//IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning. IEEE, 2007:209-216

[9] Sutton R S . Learning to Predict by the Methods of Temporal Differences[J].Machine Learning, 1988, 3(1):9-44.

[10] Silver D, Tesauro G. Monte-Carlo simulation balancing[C//International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June.DBLP, 2009:945-952

[11] Mnih V, Kavukcuoglu K, Silver D, et al. Playing Atari with Deep Reinforcement Learning[J]. Computer Science, 2013.

[12] Silver D, Lever G, Heess N, et al. Deterministic Policy Gradient Algorithms[C]//International Conference on International Conference on Machine Learning. JMLR. org,2014:387-395

[13] Hasselt H V, Guez A, Silver D. Deep Reinforcement Learning with Double Q-learning[J]. Computer Science, 2015(4):28-34.

[14] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree

search."Nature 529.7587 (2016): 484-489.

[15] Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of Go without human knowledgeJ]. Nature, 2017, 550(7676):354-359.

[16] Huang S C, Coulom R, Lin S S. Time Management for Monte-Carlo Tree Search Applied to the Game of Go[C]//International Conference on Technologies and Applications of Artificial Intelligence. IEEE, 2010:462-466.

[17] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search[C//Proc of the International Conference on Computer & Games. 2006.

[18] Allis L V. Searching for solutions in games and artificial intelligence [D]. Netherlands: Maastricht University, 1994: 164-165.

[19] Wágner J, Virág I. Solving renju [J]. Icga J, 2001, 24(1): 30-34.

[20] 张明亮，吴俊，李凡长．五子棋机器博弈系统评估函数的设计[J]．计算机应用，2012，32(7): 1969-1972．

[21] 毛丽民，朱培逸，卢振利，等．基于 LabVIEW 的五子棋博弈算法[J]．计算机应用，2016，36(6): 1630-1633．

[22] 程宇，雷小锋．五子棋中 Alpha-Beta 搜索算法的研究与改进[J]．计算机工程，2012，38(17): 186-188．

[23] 张效见．五子棋计算机博弈系统的研究与设计[D]．合肥：安徽大学，2017: 62-67．

[24] 王杨．基于计算机博弈的五子棋算法研究[D]．沈阳：沈阳理工大学，2016: 56-63．

[25] Knuth D E , Moore R w. An analysis of alpha-beta pruning[J]. Artificial Intelligence, 1975, 6(4):293-326.

[26] Shannon, C. E. Programming a computer for playing chess[J]. Philosophical Magazine, 1950, 41(314):256-275.

[27] Knuth D E, Moore R W. An analysis of alpha-beta pruning[J]. Artificial intelligence, 1975, 6(4): 293-326.

[28] 李学俊，王小龙，吴蕾，刘慧婷，六子棋中基于局部"路"扫描方式的博弈树生成算法[J]. 智能系统学报，2015，10(2)267-272.

[29] 张利群，五道棋计算机博程的设计与现[Jit 机工程，2010，36(10)：221-22.

[30] Vamvoudakis K G , Lewis F L. Online Solution of Nonlinear Two-Player Zero-Sum Games Using Synchronous Policy Iteration[C]//Decision and Control (CDC), 2010 49th IEEE Conference

on. IEEE, 2010.

[31] Buffet O , Sigaud O . Markov Decision Processes in Artificial Intelligence[M].ISTE, 2010.

[32] Mausam, Kolobov A. Planning with Markov Decision Processes: An Al Perspective[M//Planning with Markov Decision Processes: An Al Perspective. Morgan & Claypool Publishers, 2012.

[33] Bertsekas DP. Dynamic programming and optimal control[M]. Athena Scientific.2000.

[34] Sutton R S . Learning to predict by the methods of temporal difference[J].Machine Learning, 1988, 3.

[35] Radford A , Metz L, Chintala S. Unsupervised Representation Learning withDeep Convolutional Generative Adversarial Networks[J]. Computer Science, 2015.