中山大學
SUN YAT-SEN UNIVERSITY

《人工智能》

期末实验报告

学 院 名 称 ： 数据科学与计算机学院

专 业 ： 计算机科学与技术

组 员 姓 名 ： 钟怡宁 朱春陶

学 号 ： 17341217 17341220

时 间 ： 2019 年 12 月 25 日

模型一 ：MCTS黑白棋

一、实验原理

1、蒙特卡罗树搜索（MCTS）

简述：MCTS是一类树搜索算法的统称，用于解决具有巨大搜索空间的问题。其基本思想在于采样，采样的越多，则越接近最优解，采样的方式就是通过计算机模拟。网上有一个例子很容易理解，比如说你有100个苹果，你需要找到这100个苹果中最大的那个，你每次从苹果中随机选择一个，与下一个随机选择的苹果做比较，保留更大的那个（无放回），如果你进行选择的次数越多，则选到最大苹果的概率很明显越大。直到你模拟了100次，最后保留的苹果一定是最大的。但在实际问题中，比如说围棋，你没有办法把所有的情况都模拟出来，树的规模将激增，开销也会很大。所以让MCTS算法采样所有从根节点到游戏结束状态的路径访问情况，是不科学的。这其中还有一些问题需要解决，比如如何选择分支节点，因为不选择有效的分支来模拟会产生大量的无谓搜索，即宽度优化；以及评估节点是否一定要走到游戏结束状态才能进行选择，即深度优化。基本的MCTS算法主要包括4个步骤，Selection(选择),Expansion(扩展),Simulation(模拟),backpropagation(回溯)下面将根据MCTS的四个基本步骤进行更详细的介绍。

UCB算法:在MCTS的经典实现UCT(Upper Confidence Bounds for Trees)中还运用到了UCB算法。算法概括为如下公式:

$$arg\,max\ (\frac{Q(v')}{N(v')}+c*\sqrt{\frac{2lnN(v)}{N(v')}})$$

其中v'表示当前树节点，v表示父节点，Q表示这个树节点的累计quality值，N表示这个树节点的访问次数，c是一个常量参数。这个公式是用来求当前节点的后面一个选择，这个值由两个部分组成，分别是左边的平均收益值，和右边的父节点访问次数与子节访问次数的比值（这个比值越大表示子节点的访问次数越少，则越值得选择，因为访问次数少的很有可能是更优的解）。c常量通常会选择$\frac{1}{\sqrt{2}}$，这是Kocsis、Szepesvari提出的经验值。接下来就可以具体说明UCB在MCTS的应用以及MCTS算法四个步骤的具体过程。

MCTS算法原理:

①Selection(选择)

对于当前节点，目标是寻找一个最值得探索的点进行探索。这个最值得怎么评估，一般策略是选择未被探索过的子节点，如果都探索过了，那就选择UCB值最大的节点进行探索。

②Expansion(扩展)

对于选择的子节点，随机创建一个新的子节点（新子节点不能与之前的节点重复）。

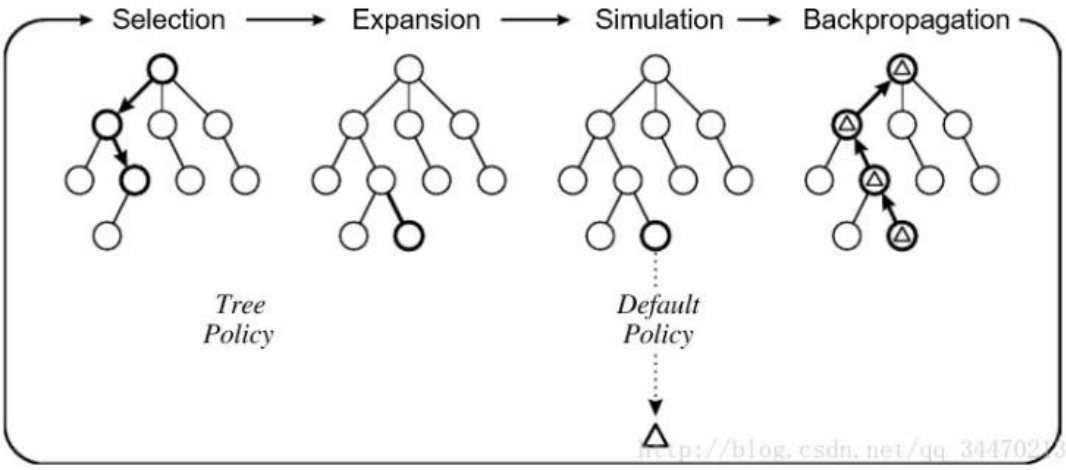
③Simulation(模拟)

扩展出子节点后，就可以根据扩展出来的节点模拟游戏交替play，直至达到游戏结束状态，将胜负信息来计算节点的得分。胜利reward为1，失败reward为0

④backpropagation(回溯)

将扩展节点经过游戏模拟后计算的reward回溯给节点的所有祖先，其中访问次数都加1，即UCB公式中的$N(v)$都加1，而$Q(v)$都加上reward。

该四个过程用一张图总结如下：



图片截自网络

MCTS 黑白棋：

传统黑白棋策略包括，a）贪心策略，每次下棋只找使得下棋之后己方棋子最多的点，即局部最优解。b）确定子策略，某些棋子一旦落子之后就不会被翻转，这样的棋子称为确定子，因此每一步都选择使得己方确定子最多的点。c）位置优先策略，考虑到对角的重要性，给棋盘上每一个点都赋予一个优先级，而每次选择优先级最高的点落子。d）机动性策略，黑白棋每一步可落子的点都是有限的，因此每一选择使得对方可落子点最少的点，这样往往容易抢占先机。而在 MCTS 中，我们也可以利用上这些策略来优先选择更有效益的点来扩展模拟。本次实验采用的就是位置优先策略。

在 8×8 的棋盘中，我们给剩下的 60 个可落子点分配不同的优先级，优先级表如下所示

1	5	3	3	3	3	5	1
5	5	4	4	4	4	5	5
3	4	2	2	2	2	4	3
3	4	2			2	4	3
3	4	2			2	4	3
3	4	2	2	2	2	4	3
5	5	4	4	4	4	5	5
1	5	3	3	3	3	5	1

这个优先级表由 Roxanne 提出，结合了多种策略，同时了结合了机动性的特性，可以因此提高自己的机动性而限制对方的可走步数。

二、算法描述

伪代码的描述。主要摘自参考论文 **A Survey of Monte Carlo Tree Search Methods**
http://pubs.doc.ic.ac.uk/survey-mcts-methods/survey-mcts-methods.pdf

Algorithm '

function UctSearch(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l =$ TreePolicy(v_0)
    $delta =$ DefaultPolicy($s(v_l)$) // $s(v_l)$表示节点的状态
    $BackUp(v_l, delta)$
  return $a(BestChild(v_0, 0))$

function TreePolicy(s_0)
  while v is not teminal **do**
    if v not fully expanded then
      return Expand(v)
    else
      $v = BestChild(v_0, C_p)$ // C_p是 UCB 算法中的 c 常数

function Expand(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BestChild(v,c)
  return $arg\,max\ (\frac{Q(v')}{N(v')} + c * \sqrt{\frac{2lnN(v)}{N(v')}})$ //v' 是v的孩子节点

function DefaultPolicy(s)

while s is not teminal **do**
choose $a \in A(s)$ uniformly at random
$s = f(s,a)$
return reward for state s

function BackUp(v,$delta$)
while v is not null **do**
$N(v) = N(v) + 1$
$Q(v) = Q(v) + delta$
$v = parent\ of\ v$

三、实现过程

说明：本实验可以实现一个利用 MCTS 搜索策略来下黑白棋的应用程序，代码实现时使用了 MVC(Model-View-Controller，模型-视图-控制)框架，该模型将应用程序划分成 3 个独立的组件：模型层、视图层、控制层以此来尽可能降低它们之间的依赖。

模型层是本次实验的核心，即 MCTS 搜索树的实现，包含了模型的数据类型定义以及模型功能实现，主要以处理控制层的请求。

视图层就负责渲染可视化，即把棋盘显示出来，并向控制层发送请求后执行相应的渲染。

控制层就是负责搭建视图层与模型层之间的桥梁，负责接受所有的用户事件，向模型层发送待处理的数据，并将结果返回给视图层以更新显示。

而本实验报告只着重介绍关于模型部分的实现过程，其余部分不是本课程的重点。

搜索树类（MCSearchTree）

——初始化

```
1.  def __init__(self, board, player, **kwargs): #kwargs 表示参数中成对键值对
2.      """
3.      #初始化类
4.      '''
5.      self.player = player
6.          #ai 角色 0（black），1（white）
7.      self.time_limit = timedelta(seconds=kwargs.get("time_limit", 15))
8.          #timedalte 是 datetime 中的一个对象，该对象表示两个时间的差值
9.          #两个 datetime.datetime 类型相减 或两个 datetime.date 类型相减 的结果就是
    datetime.timedelta 类型
10.     self.max_moves = kwargs.get('max_moves', 60)
11.         #最多可以下的棋子数
12.     self.Cp = kwargs.get('Cp', 1.414)
13.         #UCB 中常量 c
14.     self.root = Node(board, player, None, None)
15.         #根节点
16.     self.start_time = None
17.         #开始时间，用于控制搜索深度
18.     #self.definite_count = [0, 0]
```

```
        self.moves = 4
        #棋局中初始化时已经下的棋子数
        self.priority_table = deepcopy(roxanne_table)
        #优先级表
```

说明：初始化搜索树时需要区别黑棋或者白棋两种情况。

——MCTS 搜索的主函数（uct_search）

```
def uct_search(self):
    """
    #搜索函数
    :return: a step [x, y]
    """
    self.start_time = datetime.utcnow()
    self.do_simulations(self.root)
    if len(self.root.children) == 0:
        return None
    max_win_percent, chosen_child = self.best_child(self.root, 0)
    print("Winning percentage", max_win_percent)
    return chosen_child.pre_move
```

说明：该函数调用了 do_simulations 函数去执行搜索，搜索结束后，每个子节点都有了相应的 quality 以及 visit 值，调用 best_child 后计算各自的 UCB 值，从中选择最大的 UCB 值作为搜索结果。

——模拟函数（do_simulations）

```
def do_simulations(self, node):
    """
    #多线程模拟
    :param node: root node
    :return: count
    """
    count = 0
    self.time_limit = timedelta(seconds=min(single_time_limit, 62 - fabs(34 - self.moves) * 2))
    while datetime.utcnow() - self.start_time < self.time_limit:
        vl = self.tree_policy(node)
        delta = self.default_policy(vl)
        self.backup(vl, delta)
        count += 1
        if node.is_fully_expanded():
            break
    if len(node.children) == 0:
```

```
            return count
        # use multi-thread policy, refering github
        pool = ThreadPool(len(node.children))
        counts = pool.map(self.simulation, node.children)
        pool.close()
        pool.join()
        count += sum(counts)
        return count
```

说明：利用多线程去执行搜索过程，节省搜索时间。

——选择和扩展（tree_policy）

```
def tree_policy(self, v):
    """
    tree policy 过程
    :param v: Node
    :return: Node
    """
    while not v.is_terminal():
        if v.is_fully_expanded():
            value, v = self.best_child(v, self.Cp)
        else:
            return self.expand(v)
    return v
```

说明：该函数主要实现 mcts 的前两个步骤，selection 以及 expansion。

——模拟（default_policy）

```
def default_policy(self, node):
        """
        #default policy 过程
        :param node: Node
        :return: reward node.player： 0-输了，1-赢了
        """
        cur_player = node.player
        state = deepcopy(node.board)

        num_moves = 0
        while num_moves + self.moves < 64:
            if self.moves + num_moves < 56:
                valid_set = valid.get_priority_valid_moves(state.matrix, self.priority_table, c
ur_player)
                if len(valid_set) == 0:
```

```
                num_moves += 1
                cur_player = 1 - cur_player
                continue
            (cx, cy) = choice(valid_set)
        else:
            valid_set = valid.get_valid_list(state.matrix, cur_player)
            if len(valid_set) == 0:
                cur_player = 1 - cur_player  # toggle player
                num_moves += 1
                continue
            (cx, cy) = choice(valid_set)  # choose one randomly
        state = move(state, cx, cy, cur_player)  # update board
        cur_player = 1 - cur_player  # toggle player
        num_moves += 1
    return dumb_score(state.matrix, self.player) > 0 #0-输了，1-赢了
```

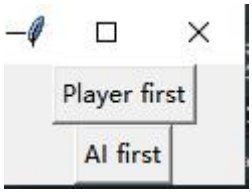
说明：该函数主要实现了 MCTS 的第三个步骤，模拟游戏对战的过程。其中模拟下棋不是随机选择落子点，而是根据优先级选择落子点，并返回最终的模拟结果。

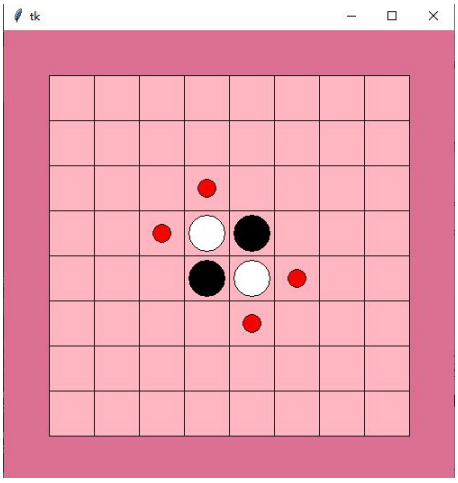
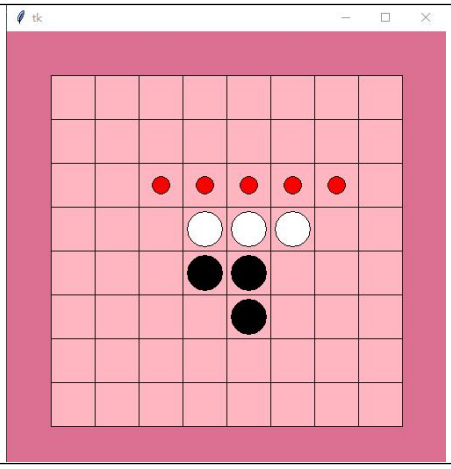
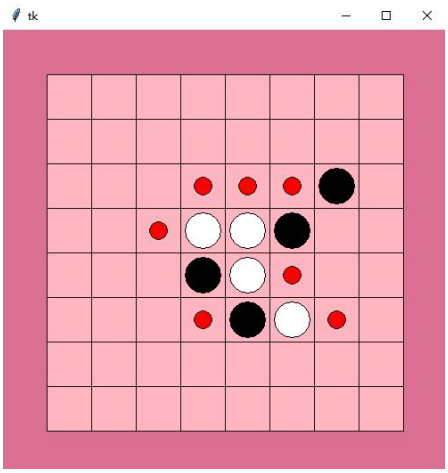
——回溯（back_up）

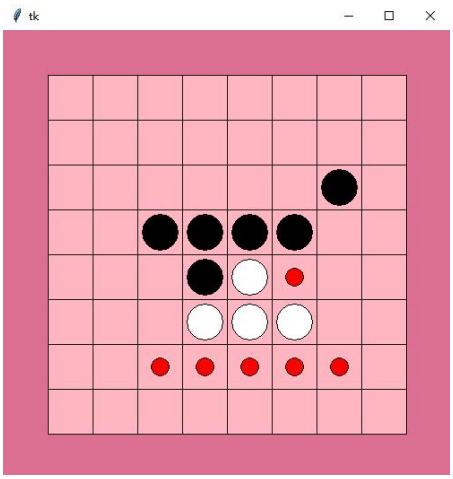
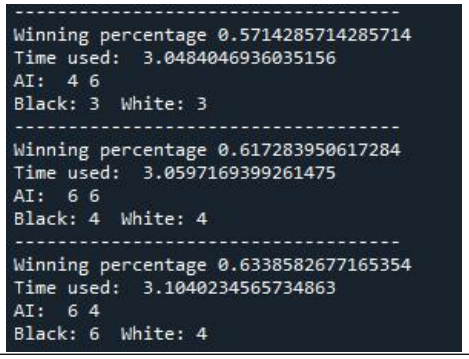
```
def backup(self, node, reward):
    """
    #回溯
    :param node: Node
    :param reward: reward:1 or 0
    :return:
    """
    while node is not None:
        node.n += 1
        if node.player == self.player:
            node.q += reward
        else:
            node.q += 1 - reward
        node = node.parent
```

说明：该函数就实现了 MCTS 的最后一个步骤，回溯游戏模拟结果。将 reward 传递给节点的祖先并且 visit 次数都要加 1。

四、实验结果

游戏开始	

游戏进行（以玩家先手为例）	
第一回合	
第二回合	

第三回合		
控制面板信息		每次 AI 下棋都会输出当前获胜概率,搜索花费的时间,AI 下的节点以及当前局面中黑棋和白棋的相应数量。

模型二 ：DQN+MCTS黑白棋

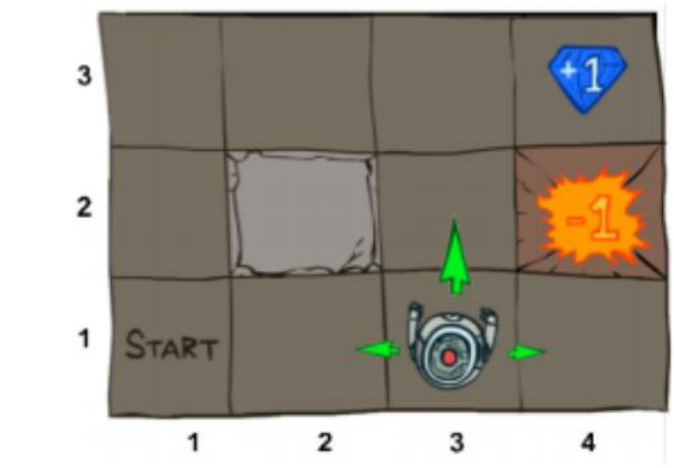
一：实验原理

强化学习的本质是 decision making 问题，即自动进行决策，并且可以做连续决策
有四个元素：agent、state、action、reward。
策略（policy）：state 到 action 的映射关系
评估方法（value function）：一个状态的价值就是从这个状态开始，期望在未来获得的奖赏，一种长期目标。
强化学习没有标签，通过反馈调节，它是在尝试动作后才能获得结果，不断调整策略，从而学习到在什么样的状态下选择什么样的动作可以获得最好的结果。强化学习中的策略相当于有监督学习中的分类或者是回归问题，强化学习的输入总在变化，每个状态下的动作影响下一个状态，从而强化学习的结果反馈有延时。

1. Qlearning 算法详解

QLearning 是强化学习算法中，值迭代的算法，Q 即为 Q（s,a）：在某一时刻的 s 状态下(s ∈ S)，采取 a (a ∈ A)动作能够获得收益的期望，环境会根据 agent 的动作反馈相应的回报 reward r，所以算法的主要思想就是将 State 与 Action 构建成一张 Q-table 来存储 Q 值，然后根据 Q 值来选取动作获得较大的收益。

公式推导引入：



（图 1.1 图片来源于:
https://blog.csdn.net/qq_30615903/article/details/80739243 ）

游戏从起点出发到达终点为胜利，掉进陷阱为失败。智能体（Agent）、环境状态(environment)、奖励（reward）、动作（action）可以将问题抽象成一个马尔科夫决策过程， 我们在每个格子都算是一个状态 S_t， $\pi(a|s)$ 为在 s 状态下采取动作 a 策略，$p(s'|s,a)$ 为在 s 状态下选择 a 动作转换到下一个状态 s’的概率。$R(s'|s,a)$

表示在 s 状态下采取 a 动作转移到 s’的奖励 reward，目的是找到一条能够到达终点获得最大奖赏的策略。

所以目标就是求出累计奖励最大的策略的期望：

$\max_{\pi} E[\sum_{t=0}^{H} \gamma^t R(S_t, A_t, S_{t+1}) | \pi]$

Qlearning 的主要优势就是使用了时间差分法 TD（融合了蒙特卡洛和动态规划）能够进行离线学习，使用 bellman 方程可以对马尔科夫过程求解最优策略。

贝尔曼方程：

通过贝尔曼方程求解决策过程中的最佳决策序列，状态值函数$V_{\pi}(s)$可以评价当前状态的好坏，每个状态的值不仅由当前状态决定，还要由后面的状态决定，所以状态的累计奖励求期望就可以得出当前 s 状态的状态值函数$V(s)$。

贝尔曼方程如下：

$V_{\pi}(s) = E(U_t | S_t = s)$

$V_{\pi}(s) = E(R_{t+1} + \gamma[R_{t+2} + \gamma[...]] | S_t = s)$

$V_{\pi}(s) = E(R_{t+1} + \gamma V(s') | S_t = s)$

其中，立刻回报为：R_{t+1}，后续状态的折扣价值函数为$\gamma V(s_{t+1})$即$\gamma V(s')$。

如果用 s’表示 s 状态下一时刻任一可能的状态，那么 Bellman 方程可以写成：

$V(s) = R_s + \gamma \sum_{s' \in S} p_{ss'} V(s')$

S 为所有可能的状态的集合，$P_{ss'}$为下一状态为s'的概率。

在该样例游戏中我们令 Q(s,a)为状态的动作值函数，则有

$Q_{\pi}(s,a) = E_{\pi}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+2} + ... | A_t = a, S_t = s]$

即：$Q_{\pi}(s,a) = E_{\pi}[G_t | A_t = a, S_t = s]$

其中G_t是 t 时刻开始的总折扣奖励， γ衰变值对 Q 函数的影响：γ越接近于 1 代表它越有远见，会着重考虑后续状态的的价值，当γ接近 0 的时候就会只考虑当前的利益的影响，所以从 0 到 1，算法就会越来越会考虑后续回报的影响。

在此基础山，我们求解最佳的价值动作函数：

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

$Q^*(s,a) = \sum_{s'} P(s' | s,a)(R(s,a,s') + \gamma \max_{a'} Q^*(s',a'))$

从而我们可以得出初步的 Q(s,a)函数的迭代更新公式:

$$Q_{k+1}^{*}(s,a) \leftarrow \sum_{s'} P(s' \mid s,a)(R(s,a,s') + \gamma \max_{a'} Q_{k}^{*}(s',a'))$$

时间差分法:

时间差分方法结合了蒙特卡罗的采样方法和动态规划方法的 bootstrapping(利用后继状态的值函数估计当前值函数)使得他可以适用于 model-free 的算法，单步更新，速度更快。值函数计算方式如下:

$$V(s) \leftarrow V(s) + \alpha(R_{t+1} + \gamma V(s') - V(s))$$

其中，$R_{t+1} + \gamma V(s')$ 被称为 TD 目标，$R_{t+1} + \gamma V(s') - V(s)$ 称为 TD 偏差。

使用时间差分法来更新我们的状态值函数，则有:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

可以利用该公式进行 Q(s,a)等迭代更新，从而得出需要的最佳的策略。

2. DQN 详解

在传统的强化学习中，Q_learning 需要一张由状态 S 以及行为 A 组成的 Q表,行为的种类一般较少，比如常见的前进后退两种或上下左右四种等,Q 表的列的数量在可控范围之内，但是状态，在一些条件下是非常多的，如围棋，也就是 Q 表的行数过多，导致的结果就是难以维护如此大的一张 Q 表。

现在假设有一个函数 f（x）

如果输入状态 S 就可以得到每个行为的 Q 值即 Q(S)= f（S）

只要知道这个函数，就不必再去维护那张很大的 Q 表。

函数模型的确定可以依靠神经网络，由于这是深度学习（DL）和强化学习（RL）的结合，所以也叫作 DRL，而 DQN 就是 DRL 的一种，其是神经网络和 Q_learning 的一种结合，

其有两个关键技术:

冻结 targetNet，experience replay（经验池）

冻结 targetNet 就是说每隔一段时间将训练好的神经网络参数保存下来,这里实际就是定义两个完全相同的神经网络 Net1 和 Net2，然后 Net1 作为实时训练的网络，里面包括 loss 以及 optimizer，而 Net2 有着与 Net1 相同的模型结果，但没有 loss

以及 optimizer，每隔一段时间，就将 Net1 中训练到目前为止的权值和偏置值保存到 Net2。

experience replay 它存储了带标签的一个个数据样本，然后神经网络通过随机抽样解决了相关性及非静态分布问题。训练神经网络是需要带标签的样本的，experience replay 是通过将（ S，A，R，S'）（当前状态，基于当前状态采取的下一行动，采取 A 后的奖励值，采取 A 后到达的下一步状态）作为一个带标签的样本。

根据 Qlearning 的更新公式：

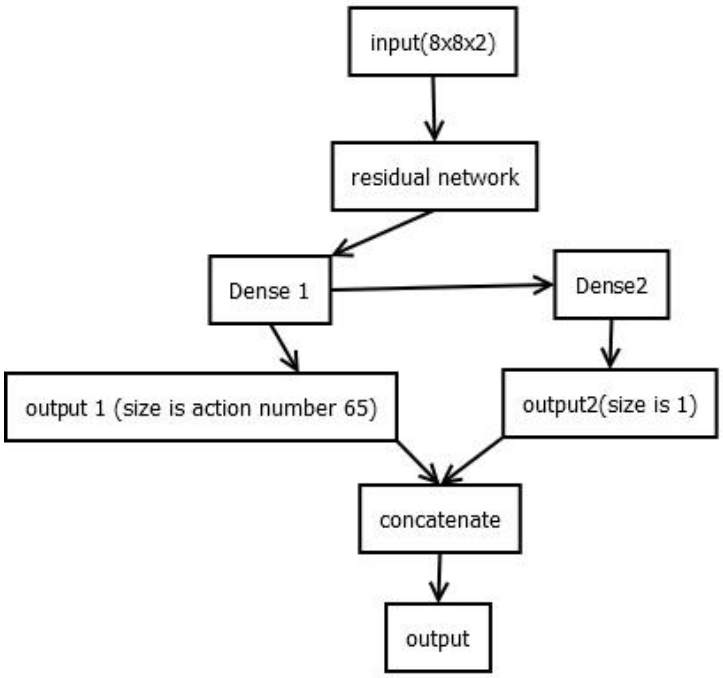
$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

这里将 $r + \gamma \max_{a'} Q(s',a')$ 作为每个样本的 label,将 $\max_a Q(s,a)$ 作为 S 输入后得到的预测值。Loss 的设置可以取交叉熵，也可以使用 MSE。

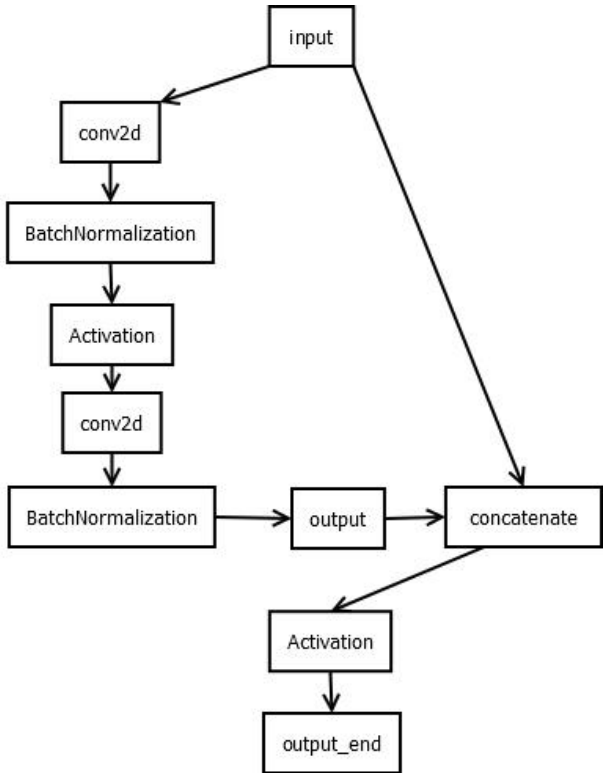
二：DQN 的实现

网络搭建：选用卷积残差神经网络，网络搭建相对简单

网络结构图如下：



其中残差神经网络的单个神经单元的结构定义为：



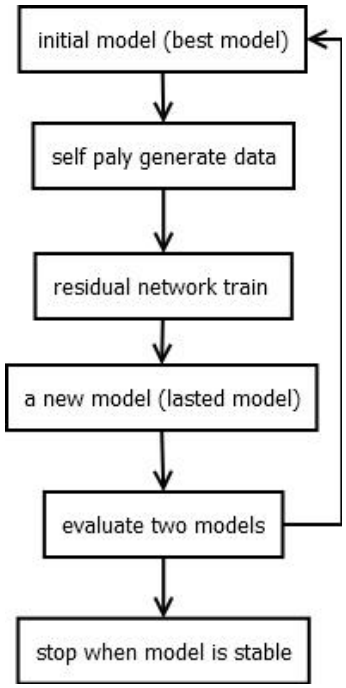
模型参数更新方法:

通过两个模型对比分析的方法实现模型参数的更新取优,利用翻转棋执棋双方进行博弈的思想，同时 load 进 2 个模型，一个为最新训练出来的 latest model,另外一个为从多轮评测中得到的最优的 best model,然后让其一方为先手，一方为后手，进行对弈，胜利，失败或者是平局都有其对应的得分，最后求出 latest model 在设定的对弈场数如 10 局中得分的均值，如果均值大于我们设定的平局的得分阈值，说明 latest model 是要比 best model 的性能要好的，就更新 latest model 为 best model，舍弃原来的 best model。

模型训练及数据喂取方式:

这部分也是 DQN 的精华之所在了吧，我觉得。首先，言语表达的过程为：首先会有一个最初始的 best model,这个 model 为经过训练，相当于一个初始化的状态，然后，我们有一个 self_play 环节，这个环节的主要目的是训练数据的产生和获取，在产生训练集数据的过程中，使用了 MCTS 算法，利用 best model,来获取每一步的得分，利用该得分当作每一个 action 的权重，来随机选取下一个状态，每次自动更换先手后手，计算每一个当前状态，动作对应的奖励和下一个状态，记录下来，然后喂给我们的神经网络,就会进行训练生成一个 latest model,此时我们会进行模型的评估工作,即将 best

model 和 latest model 进行对弈选择，选择出当前的 best model，然后再次利用 best model 进行 self_play，产生新的一轮训练数据，接着喂给神经网络，从而会再次产生一个 latest model，至此循环迭代进行操作，直到 model 趋于稳定结束训练。图示如下：



模型 mcts select 部分借鉴：

模型中使用的 mcts 和一般的 mcts 是有区别的，参考论文为 AlphaZero 论文《Mastering the game of Go without human knowledge》中，定义了 PUCT 算法，其中

$$a_t = \arg\max_a (Q(s_t,a) + U(s_t,a))$$

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

如果想要进行选择，我们必须直到当前节点的 Q 值和 U 值，他们的默认值为 0，我们默认每个节点的 Q 值和 U 值都是存在的，所以我们可以直接根据公式选择最优节点。详细算法请看第一部分的 MCTS 模型实现。

代码展示请看附件。

伪代码：

初始化 $Q(s,a)$，$\forall s \in S, a \in A(s)$,任意的数值，并且 Q（$ter\min al - state,.) = 0$

重复（对每一节 episode）

初始化状态 S

重复（对 episode 中的每一步）：

使用某一个 policy,根据状态 S 选取一个动作执行

执行玩动作后，观察 reward 和新的状态 S'

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

$$S \leftarrow S'$$

循环直到 S 终止

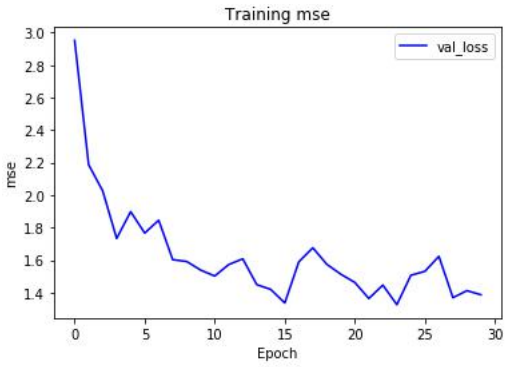
三：结果分析展示

1.训练相关数据展示

由于硬件需求无法满足，我们只能在可以满足的条件下进行模型的训练更新，所以训练的次数可能并不足够，这里会进行可达范围内的结果分析。

训练中 loss 的走势

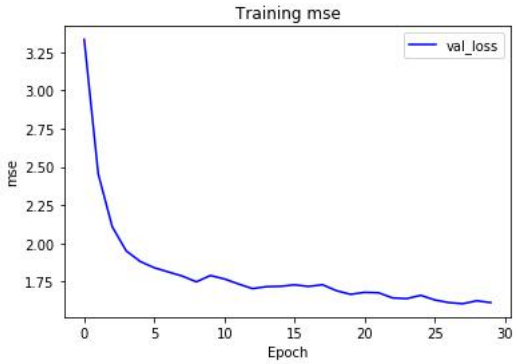
在初始 model 的情况下，loss 的走势如下：



这时候和 latest model 进行 evaluate,结果为：

```
Evaluate 10/10
AveragePoint 0.6
Change BestPlayer
```

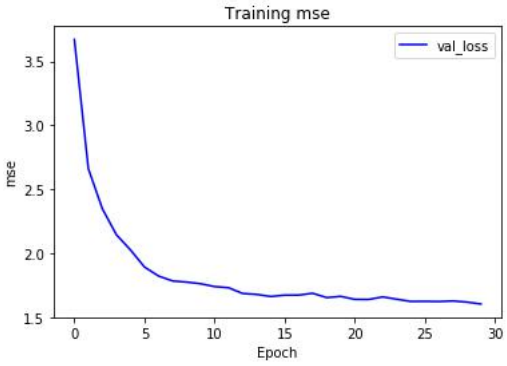
结果超过了我们设置的平局时的阈值 0.5，所以我们更换最佳模型的选择，然后再次进行训练数据的生成，再次喂给神经网络，观察 loss 的变化情况以及模型评估情况。



模型评估结果为：

```
Evaluate 10/10
AveragePoint 0.75
Change BestPlayer
```

再次进行训练数据的生成，再次喂给神经网络，观察 loss 的变化情况以及模型评估情况。



Evaluate 结果为：

```
Evaluate 10/10
AveragePoint 0.5
```

结果分析：上述为 3 轮数据投喂的结果，初始的 model 是我训练了 12 小时的 best model,该 model，数据投喂循环了 6 次，每次 50 个回合的 self_play,100 个 epoch 训练，10 次 evaluate。

从结果图可以看出，loss 最终趋于收敛，收敛结果在 1.5 左右，并且有继续下降的趋势，因为不可以再次跑 12 个小时来跑出来一张图，所以在跑图的时候，只训练了 30 次，self_play 的自我对战局数也下降到了 20。

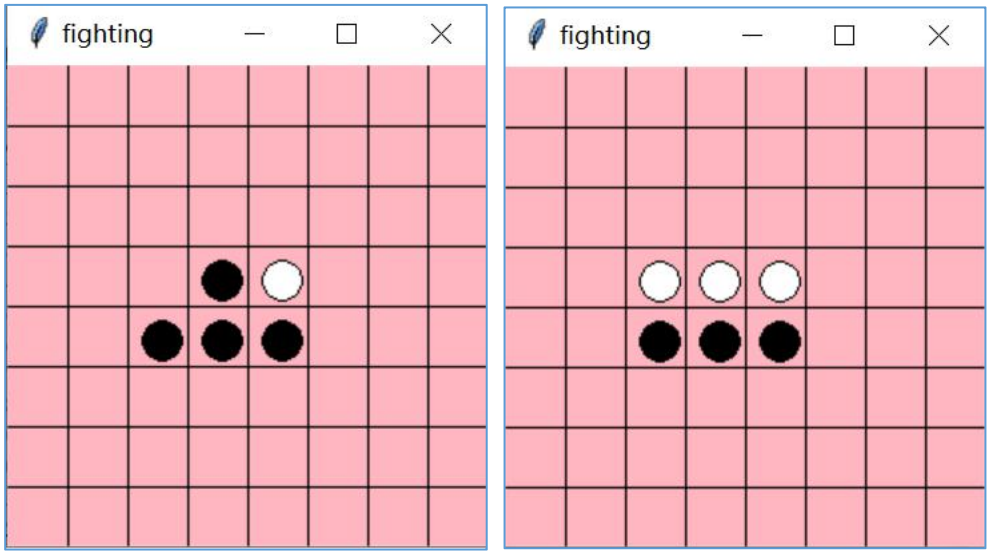
然后，又进行了 3 次数据投喂，得到了我们训练的 loss 的的走向趋势，可以看出第

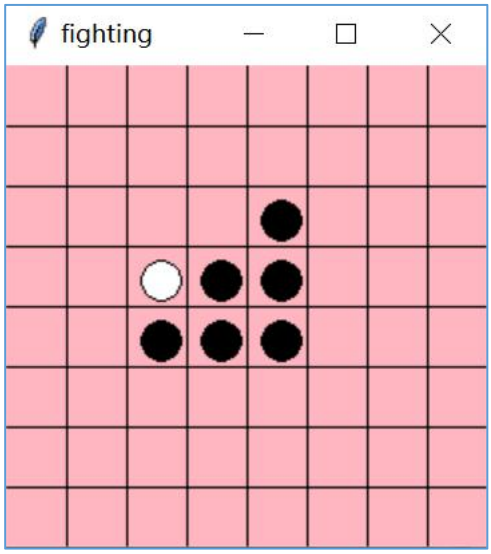
一个图 loss 的收敛有波动，第二个和第三个图的 loss 就趋于了稳定和收敛，然后再模型的对比评估上，我们设置的是胜利积分为 1，失败积分为 0，平局积分为 0.5，所以如果再两个模型对战 10 局之后，latest model 的平均得分大于 0.5，说明当前模型的性能比 best model 的性能是要好的，所以就可以进行 change 操作，更新 best model。我们进行的 3 次数据投喂中有两次都发生了模型的更新，可知当前模型并没有稳定，并没有达到最优，而且再数据投喂，迭代训练的过程中，模型是逐渐趋于稳定的，平均得分会逐渐趋于 0.5，并达到稳定状态，从而得到一个最优的模型。再结果分析中，个人认为是需要画一个 model 在 evaluate 的过程中的平均得分图的，看平均得分是否会趋于并收敛到 0.5，但是由于硬件条件有限，这里画起来有些不现实。所以只在理论上估计猜测，模型在得到了足够的训练和评估更新之后，模型的平均得分会收敛到 0.5，达到最优。

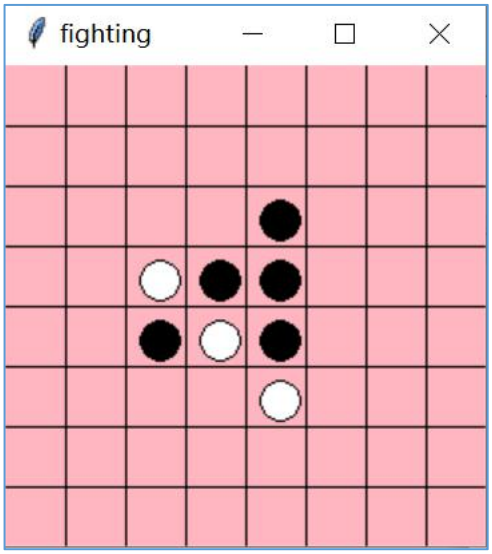
2.人机对战结果展示（5 个回合）

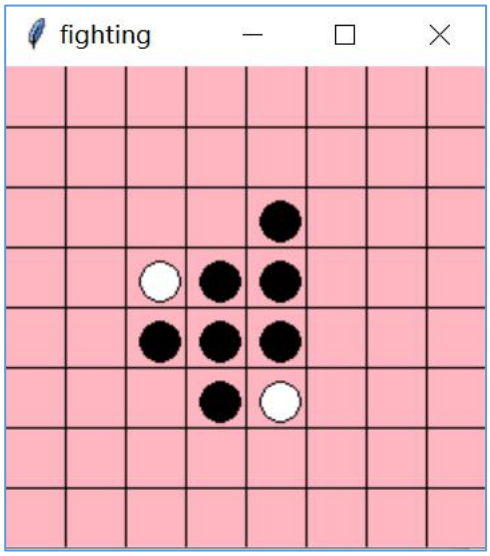
使用的模型并非最优模型，最优模型需要大量的训练，这台电脑能力有限，做不到啊！

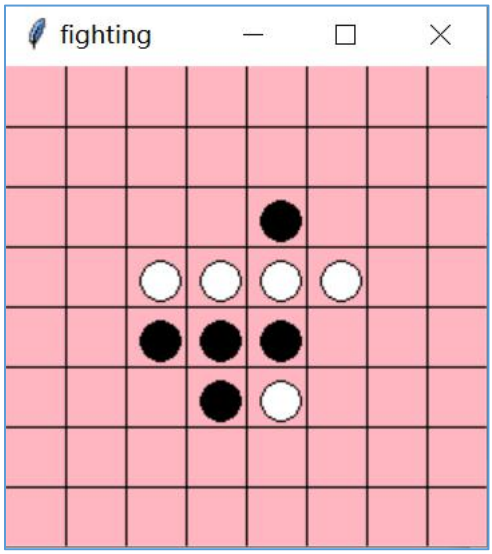
人类先手：flag=False



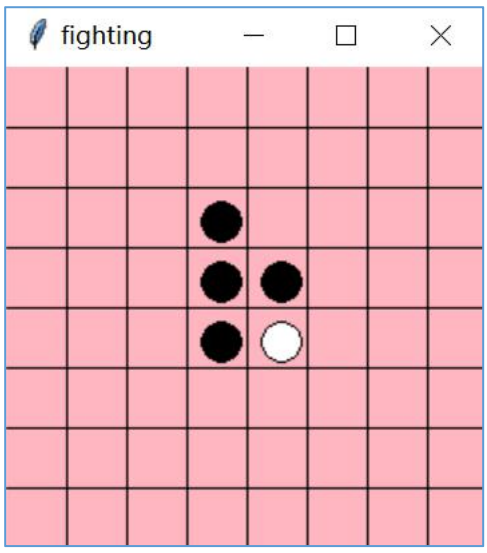


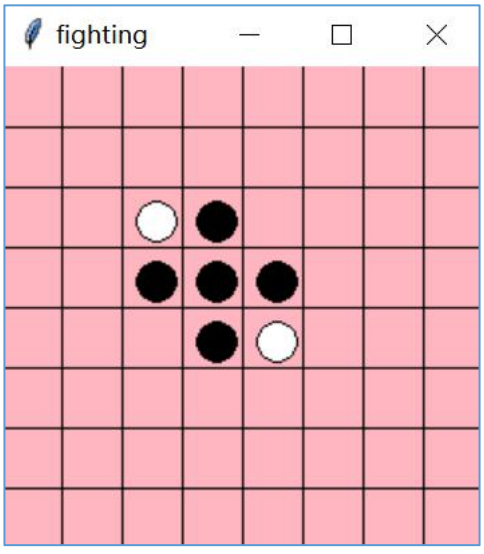


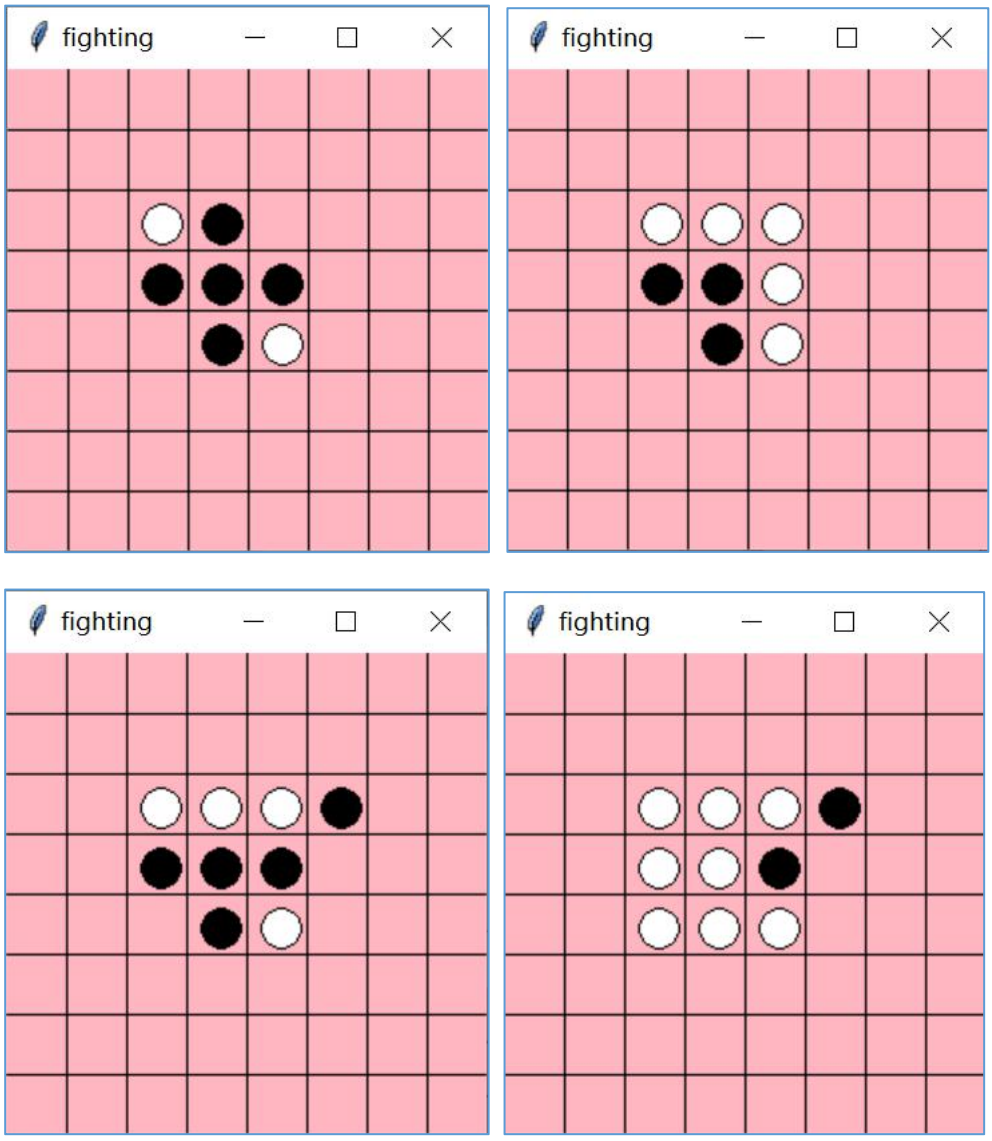




AI 先手：flag=True







至此，结果展示完毕。该 AI 可以正常和人类下棋，只是模型有待优化。

参考资料:

Github：

https://github.com/kaychintam/MiniAlphaGo

https://github.com/avartia/miniAlphaGo-for-Reversi

蒙特卡洛树搜索 MCTS：

https://zhuanlan.zhihu.com/p/25345778

https://zhuanlan.zhihu.com/p/26335999

https://zhuanlan.zhihu.com/p/34990220

TKinter:

http://effbot.org/tkinterbook/

https://www.cnblogs.com/collectionne/p/6885066.html

https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html

http://www.runoob.com/python/python-gui-tkinter.html

https://docs.python.org/2/library/tkinter.html

DQN

https://www.cnblogs.com/webRobot/p/10062267.html

https://github.com/sasaco/tf-dqn-reversi

https://www.jianshu.com/p/142072151161

https://blog.csdn.net/weixin_42001089/article/details/81448677

https://blog.csdn.net/qq_30615903/article/details/80744083

https://www.jianshu.com/p/b92dac7a4225

小组分工说明:

模型一代码实现及报告: 钟怡宁

模型二代码实现及报告: 朱春陶