中山大學
SUN YAT-SEN UNIVERSITY

AI 实验报告

期末大作业

姓名：刘斯宇
专业：计算机科学与技术
学号：17341110
姓名：何家栋
专业：计算机科学与技术
学号：17308054

Contents

1 DQN 算法 **3**
1.1 算法原理 . . . 3
1.1.1 强化学习 . . . 3
1.1.2 DQN 模型 . . . 3
1.1.3 Nature DQN 模型 . . . 4
1.2 实验伪代码 . . . 4
1.3 关键代码 . . . 5
1.3.1 Gym 环境定义 . . . 6
1.3.2 NDQN 类定义 . . . 9
1.4 实验结果展示 . . . 13

2 复现 AlphaZero **14**
2.1 什么是强化学习? . . . 14
2.2 Alpha-Othello Zero 概述 . . . 15

2.3 MCTS 16
2.3.1 步骤 16
2.3.2 参数介绍 16
2.3.3 Select 17
2.3.4 Expand 18
2.3.5 Backup 19
2.3.6 Play 20
2.4 神经网络 21
2.4.1 强化学习的过程 21
2.4.2 self-play 22
2.5 实验效果 24

3 分工 **25**

1 DQN 算法

1.1 算法原理

1.1.1 强化学习

基本情况

强化学习是和和监督学习、无监督学习并列的第三种机器学习方法。有监督学习中，人工智能通过学习已经给出来的样例以及标签，进行学习，通过寻找一个映射或者回归进行学习。而强化学习是通过给出来的奖励值。不断地纠正自己的行为以获得最大的奖励。同时强化学习中的数据之间并不是相互独立的，而是下一步会不断影响之后的状态，类似生物学中的负反馈调节，不断地更新每一步的价值矩阵，争取在每一步都找到最佳的策略。无监督学习则是通过没有给出标签的数据自己寻找里面的规律，同时每一个数据之间也是相互独立的。

基本要素

- 环境状态 S_t，表示 t 时刻下环境的状态，为其环境下状态集合中的一个子集。
- 个体动作 A，表示个体在某一个环境状态下的行动环境奖励 R，t 时刻下个体在 S_t 情况下采取的行动 A_t 的奖励会在下一个时刻得到即 R_{t+1}
- 采取的行动策略 Π，根据当前的情况判断个体应该如何行动
- 采取行动的价值 V，这里的价值不仅仅是由当前状态所决定，也是由后续状态的发展所决定的，如下

$$V_\pi(S_t) = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^n R_{t+n+1}$$

- 衰减系数 γ，在上述式子中的衰减系数，决定了后续的状态价值会对当前状态的价值造成多大的影响的权重

通常的选择策略一般是有一定的几率选择随机的合法节点进行探索，其他时候都选择预估价值最大的节点进行探索。

1.1.2 DQN 模型

实际上我们是采用一个神经网络来根据当前的状态来获得每一个动作对应的价值。然后根据当前获得的环境价值以及动作价值进行下一步动作的选择，需要注意的是，这里对神经网络并没有特别的要求，可以是任何形式的神经网络，如 CNN、DNN 和 RNN。需要注意的是，强化学习本身是没有任何的数据的，我们需要自己收集数据，这时候就涉及到我们自己设计一个经验池，每一次学习都从里面挑选 batch_size 大小的数据进行训练，我们通过不断地让后面状态的回报影响当前状态的动作价值，使得每一次预测的代价收敛得到最终的价值网络。正是因为后面的状态价值预测加入了当前代价中，因此一定会与原来的预测产生误差。需要注意的是，为了防止出现动作的单一化，每一步都有一定的概率进行合法位置的随机选择。算法流程：

- 初始化 Q 网络的所有参数，状态
- 进入迭代循环
  - 根据当前的状态进行动作的选择

- 实行当前动作 A，获得下一个状态 S_{t+1}、下一个状态的回报 R_{t+1} 以及是否终止将这一个五元组 $(SAR_{t+1}S_{t+1}done)$ 存入经验池 D
- $S = S_{t+1}$
- 如果经验池的长度大于训练一轮的样本长度时，从 D 中随机抽取 batch_size 个样本进行训练，即计算 y_i 值

$$y_i = \begin{cases} R_i, & \text{done} = \text{True} \\ R_i + \gamma \max Q\left(S_{i+1}\right), & \text{done} = \text{False} \end{cases}$$

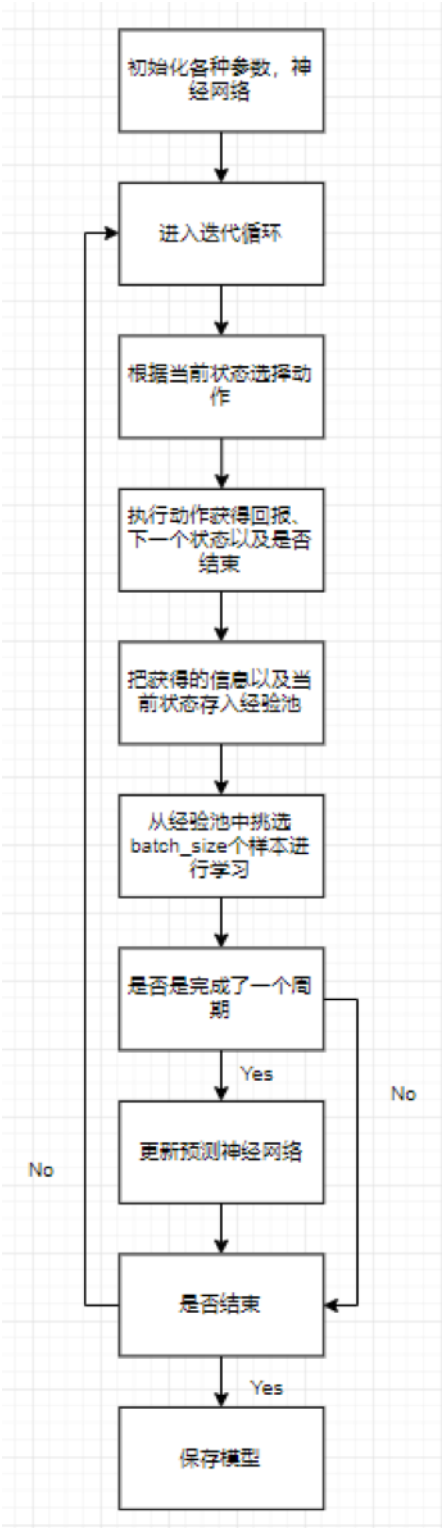
- 根据得到的 batch_size 个样本更新神经网络

1.1.3 Nature DQN 模型

在 DQN 模型中，我们通过获得的数据直接进行训练的话，会导致其对自己的前面的预测太过于敏感，loss 的波动会很大，不利于收敛，于是就有了 Nature DQN 模型，实际上就是把预测的神经网络与训练用的神经网络分开，维持预测神经网络在一定程度上的稳定性。迭代到一定周期之后，使用训练用的神经网络对预测用的神经网络进行参数更新即可。需要注意的是为了更加好地收敛，我们的探索率需要不断地缩小，但是最后也要维持在一定的水平。我这里直接实现了 Nature DQN。

1.2 实验伪代码

实际上这里的伪代码就是上述的算法流程的具体表达，由于我最终实现的是 Nature DQN，这里的流程图也采用 Nature DQN 的。



1.3 关键代码

这次的实验分成了 3 个部分，一个是 gym 的自定义环境文件，一个是 Nature DQN 的类声明以及实现文件，最后一个是主函数，包括了绘制函数以及 AI 的自我博弈过程。

1.3.1 Gym 环境定义

初始化部分

包括定义棋盘的大小，以及初始化每一个位置上的棋子（初始化状态），需要注意的是，还要在棋盘的中心放上黑白各两个棋子

```
    def __init__(self):
        #棋盘大小
        self.SIZE = 8
        #初始棋盘是0    -1表示黑棋子    1表示白棋子
        self.chessboard = [ [  0 for v in range(self.SIZE)  ] for v in range
        self.viewer = None
        self.step_count = 0
        self.chessboard[3][3] = 1
        self.chessboard[3][4] = -1
        self.chessboard[4][3] = -1
        self.chessboard[4][4] = 1
```

Reset 函数

将棋盘恢复到初始状态，在完成一轮训练之后，需要重新开始就会调用这个函数。

```
    def reset(self):
        self.chessboard = [
            [  0 for v in range(self.SIZE)  ] for v in range(self.SIZE) ]
        self.chessboard[3][3] = 1
        self.chessboard[3][4] = -1
        self.chessboard[4][3] = -1
        self.chessboard[4][4] = 1
        self.step_count = 0
        return self.chessboard
```

Step 函数

用于在当前状态下，执行传入的操作，并且根据传入的操作进行下一状态的胜负判断，并且返回一定的下一个状态带来的回报。如果胜利方为自己的颜色，返回 1000+ 自己比对面多的棋子的数量，如果是平局，返回-40，没有产生胜负则返回 0.

```
    def step(self, action):
    '''
␣␣␣␣#非法操作
␣␣␣␣if␣not␣self.is_valid_set_coord(action[0],action[1]):
␣␣␣␣␣␣␣␣return␣self.chessboard,-50,False,{}
␣␣␣␣'''

```

```
    #棋子

    self.judge_rule(action[0],action[1],action[2],True)
    self.chessboard[action[0]][action[1]] = action[2]

    self.step_count +=1

    #胜负判定
    color = action[2]

    win_reward = 1000
    common_reward = 0
    draw_reward = -1000

    win,a,b = self.is_win()
    if win == 2:
        return self.chessboard,common_reward,False,{}
    elif win == color:
        return self.chessboard,win_reward+10*(a-b),True,{}
    elif win == -1*color:
        return self.chessboard,draw_reward-10*(a-b),True,{}
    else:
        return self.chessboard,-40,True,{}
```

Is_win 函数

用于判断当前状态是否出现输赢的情况，并且对各个颜色的棋子进行计数，有利于调用函数对胜负情况进行更加仔细的评估。

```
    def is_win(self):
    black = 0
    white = 0
    isover = True
    count = 0

    for r in range(8):
        for c in range(8):

            if self.chessboard[r][c] == 1:
                white = white + 1
                count = count + 1
                continue
            if self.chessboard[r][c] == -1:
                black = black + 1
```

```
                count = count + 1
                continue

            if 0 < self.judge_rule(r,c,-1,False) or 0 < self.judge_rule(r,c,
                isover = False

    if isover == False:
        return 2,0,0

    if white < black:
        return -1,black,white
    elif black < white:
        return 1,white,black
    else:
        return 0,white,black
```

Judge_rule 函数

用于判断输入的位置是否会引起附近的异色棋子的变色，同时输入参数中还有一个是否允许吃子的参数，如果不能够吃子，则这个函数是用来判断是否存在变色的可能性，如果允许吃子，这个函数用于更新当前的状态。返回是可以吃或者已经被吃掉的棋子的个数，在外层函数中可以利用这个数量进行更加详细的处理判断。

```
    def judge_rule(self,r,c,role,eatchess):
        dir = [(1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1),

        tempX,tempY = r,c
        eatNum = 0
        if self.chessboard[tempX][tempY] != 0:
            return 0

        for i in range(8):
            tempX = tempX + dir[i][0]
            tempY = tempY + dir[i][1]
            if tempX < 8 and tempX >= 0 and tempY < 8 and tempY >= 0 and sel
                tempX += dir[i][0]
                tempY += dir[i][1]
                # // 继续判断下一个，向前走一步
                while (tempX < 8 and tempX >= 0 and tempY < 8 and tempY >= 0
                    if self.chessboard[tempX][tempY] == 0:
                        #  // 遇到空位跳出
                        break
                    if self.chessboard[tempX][tempY] == role :
                    # { // 找到自己的棋子，代表可以吃子
```

```
                        if eatchess == True :
                        # { // 确定吃子
                            self.chessboard[r][c] = role
                            # // 开始点标志为自己的棋子
                            tempX -= dir[i][0]
                            tempY -= dir[i][1]
                            #  // 后退一步
                            while (tempX != r) or (tempY != c):
                                # // 只要没有回到开始的位置就执行
                                self.chessboard[tempX][tempY] = role
                                #  // 标志为自己的棋子
                                tempX -= dir[i][0]
                                tempY -= dir[i][1]
                                # // 继续后退一步
                                eatNum = eatNum + 1
                                # // 累计

                        else:
                            #  { //不吃子，只是判断这个位置能不能吃子
                            tempX -= dir[i][0]
                            tempY -= dir[i][1]
                            # // 后退一步
                            while (tempX != r) or (tempY != c):
                                #  { // 只计算可以吃子的个数
                                tempX -= dir[i][0]
                                tempY -= dir[i][1]
                                # // 继续后退一步
                                eatNum = eatNum + 1
                            break
                    # // 跳出循环
                    #  // 没有找到自己的棋子，就向前走一步
                    tempX += dir[i][0]
                    tempY += dir[i][1]
                # // 向前走一步
            tempX,tempY = r,c
        return eatNum
```

1.3.2 NDQN 类定义

包括了构建网络、网络更新、网络加载、网络训练、构建经验池、贪婪行动策略以及非贪婪行动策略。这个文件主要是针对黑白棋的强化学习类。这里挑一些重点来说。

构建神经网络

由于棋盘是一个二维矩阵，我这里使用的是 CNN 网络，最后的映射到一个长度为 64 的一维向量，用于表示不同的位置，可以通过除法以及取余运算得到不同的棋盘位置。需要注意的是，target_q 网络需要和训练中的 q 网络可以实现更新，必须保证网络结构的一致性。

```
    def create_net(self):
        model = keras.Sequential()
        model.add(Conv2D(32,kernel_size=(3,3),activation='relu',kernel_initi
        model.add(Conv2D(32,kernel_size=(3,3),activation='relu',kernel_initi
        model.add(Conv2D(64,kernel_size=(3,3),activation='relu',kernel_initi
        model.add(Flatten())
        model.add(Dense(128,activation='relu',kernel_initializer='he_normal'
        model.add(Dense(64,activation='softmax',kernel_initializer='he_norma
        model.compile(loss='mse', optimizer=Adam(1e-3))
        model.summary()
        return model
```

Process_batch 函数

实际上就是根据经验池返回对应的训练使用的数据，x 值就是对应的状态值，y 值对应的是每一个位置的得分值，实际上就是更新对应的最佳位置的得分值，是当前状态的得分加上后续预测的得分，是一个得分的传递。

```
    def process_batch(self,batch):
    data = random.sample(self.memory_buffer,batch)
    states = np.array([d[0] for d in data])
    next_states = np.array([d[3] for d in data])

    y = self.target_net.predict(states)
    q = self.target_net.predict(next_states)

    for i,(_,action,reward,_,done) in enumerate(data):
        target = reward
        if not done:
            target += self.gamma * np.argmax(q[i])
        else:
            target = reward
        y[i][action[0]*action[1]] = target
    # self.memory_buffer.clear()
    return states, y
```

Egreed_action 函数

贪婪选择函数，每一次有一定的概率在合法的位置上随机选择，或者选择预测的动作价值最高的动作。这里的合法位置需要下了该位置可以实现至少一个异色棋子的反转，非合法位置将其的价值降到最低。

```
    def egreedy_action(self,state,cmp,env):
        state = [[cmp*x for x in v]  for v in state]
        state = np.reshape(state,(1,8,8,1))
        state = np.array(state)
        Q_value = self.model.predict(state)[0]

        min_v = -150000
        valid_action = []
        for i in range(len(Q_value)):
            if state[0][math.floor(i/8)][i%8][0]==0 and env.judge_rule(math.
                valid_action.append(i)
            else:
                Q_value[i] = min_v

        if len(valid_action) == 0:
            return -1

        if random.random() <= self.epsilon:
            return valid_action[random.randint(0,len(valid_action) - 1)]
            #return random.randint(0,self.action_dim - 1)
        else:
            return np.argmax(Q_value)
```

主函数文件

主函数主要是实现 AI 的自我博弈过程，并且在下了一定数量的对局之后，进行 target_q 的跟新以及测试，最后保存模型以及记录实验中的 loss 以及平均回报等数据。首先是自我训练部分。

```
    for episode in range(EPISODE):
    # initialize task
    state = env.reset()
    camp = -1
    # print('episode␣',episode)
    # Train
    for step in range(128):
        #自己下一步棋
        action = agent.egreedy_action(state,camp,env) # e-greedy action for
        if action==-1:
```

```
            if camp == 1:
                camp = -1
                continue
            camp = 1
            continue
        action = [math.floor(action/SIZE),action%SIZE,camp]
        next_state,reward,done,_ = env.step(action)
        if camp == -1:
            camp = 1
        else:
            camp = -1
        # Define reward for agent
        reward_agent = reward
        agent.remember(state,action,reward,next_state,done,camp)
        state = next_state
        if len(agent.memory_buffer) > agent.batch:
            count = count + 1
            x,y = agent.process_batch(agent.batch)
            loss += agent.model.train_on_batch(x,y)
            agent.update_epsilon()
        if done == True:
            # print('done step ',step)
            break
```

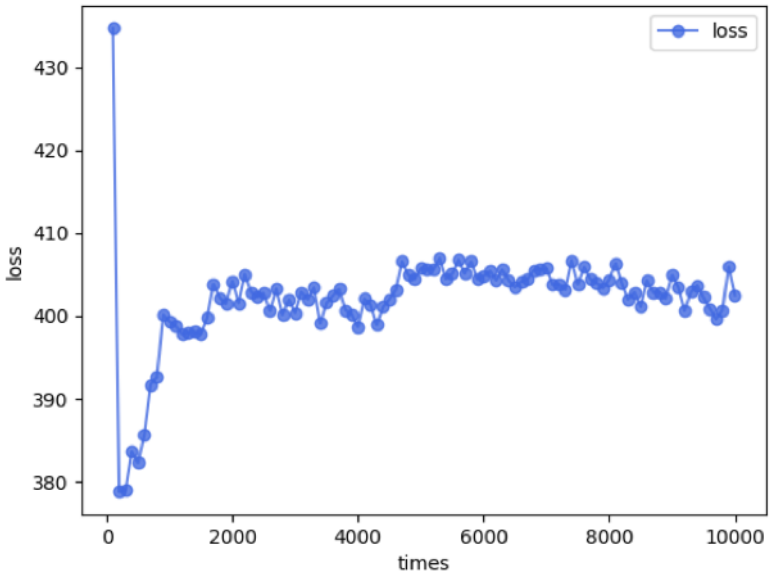
然后是测试部分，需要注意的是，这里的行动不需要随机行动，因此使用的只是 action 函数，以及一开始要实现网路的更新。

```
    if episode % 100 == 99:
    agent.update_net()
    # window = get_win()
    aver_loss.append((episode,loss/count))
    print('aver loss:',loss/count)
    count = 0
    loss = 0
    total_reward = 0
    for i in range(TEST):
        state = env.reset()

        # draw(window,env)
        camp = -1

        for j in range(STEP):
            action = agent.action(state,camp,env) # e-greedy action for trai

```

```
            if action==-1:
                if camp == 1:
                    camp = -1
                    continue
                camp = 1
                continue

            action = [math.floor(action/SIZE),action%SIZE,camp]
            state,reward,done,_ = env.step(action)
```

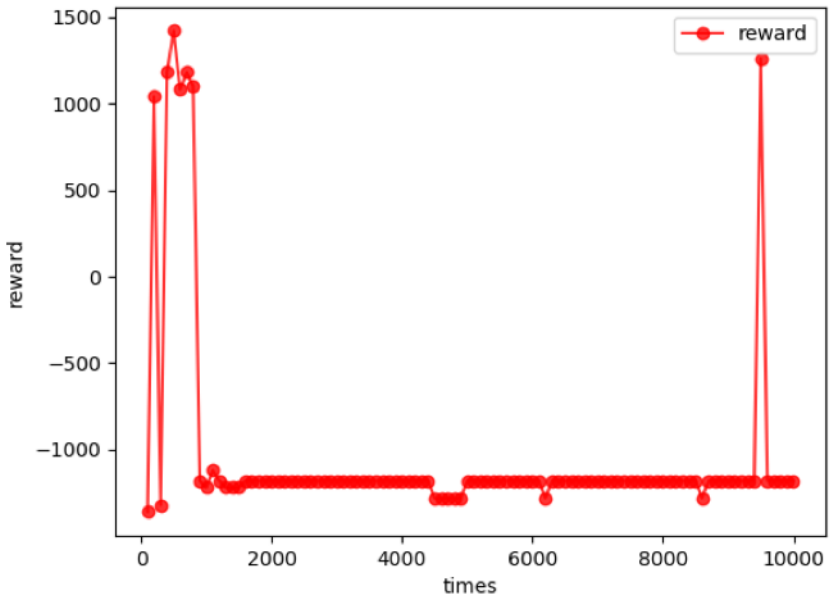
1.4 实验结果展示

首先展示的是损失函数的情况，可见刚开始的时候下降得很快但是到后面我们可以看到函数有个上升的趋势，但是越到后面反而有点继续往下走的感觉，由于训练的次数还不够多，这里



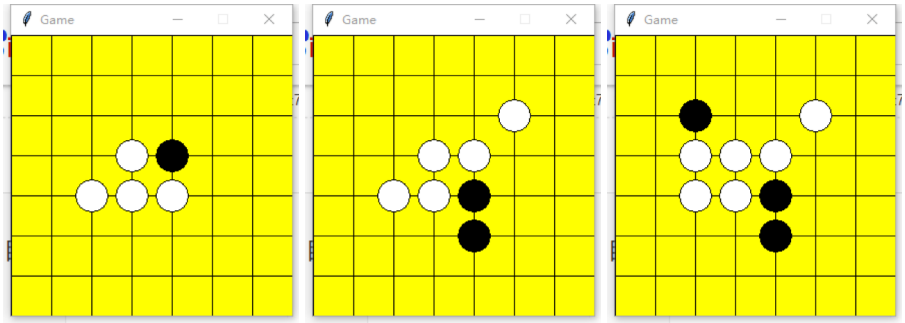
只能看出一个大概的趋势。

然后我们可以来看看 reward 回报函数的情况，我们可以看到 reward 一直都是保持一个相对稳

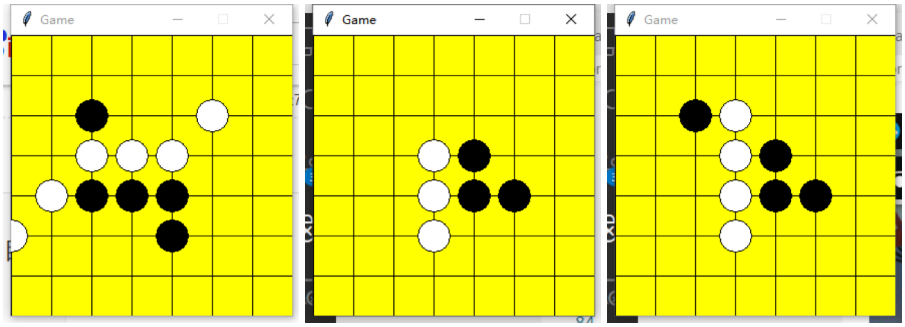


定的情况。

- 我们选择后手时：AI 执白色



- 我们选择先手时：AI 依旧执白棋



2 复现 AlphaZero

2.1 什么是强化学习?

强化学习是一类算法, 是让计算机实现从一开始什么都不懂, 通过不断地尝试, 从错误中学习, 最后找到规律, 学会了达到目的的方法. 这就是一个完整的强化学习过程. 这不就是我们小时候什么都不懂，然后一步步的摸索，然后学会东西的过程么？其实强化学习就是根据这个思路得到的。机器头一次在围棋场上战胜人类高手, 让计算机自己学着玩经典游戏 Atari。既然要让

计算机自己学, 那计算机通过什么来学习呢? 原来计算机也需要一位虚拟的老师, 这个老师比较吝啬, 他不会告诉你如何移动, 如何做决定, 他为你做的事只有给你的行为打分, 那我们应该以什么形式学习这些现有的资源, 或者说怎么样只从分数中学习到我应该怎样做决定呢? 很简单, 我只需要记住那些高分, 低分对应的行为, 下次用同样的行为拿高分, 并避免低分的行为. 可以看到在强化学习中，分数对于判断是非常重要的，所以强化学习具有分数导向性，其实这不就是监督学习里面的标签么，一样的思路呀。

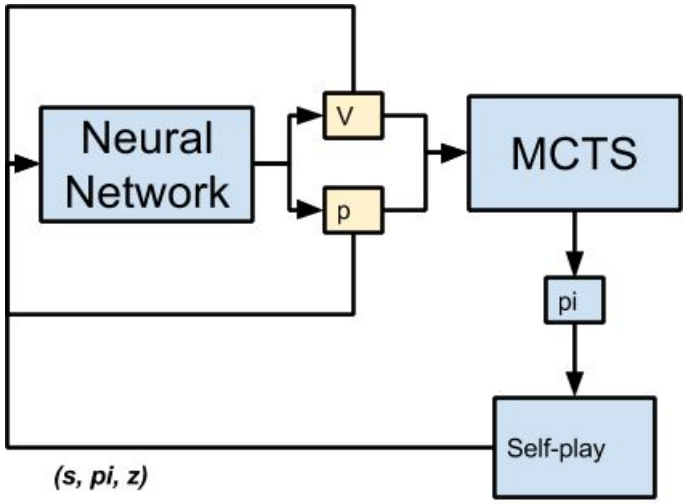
2.2 Alpha-Othello Zero 概述

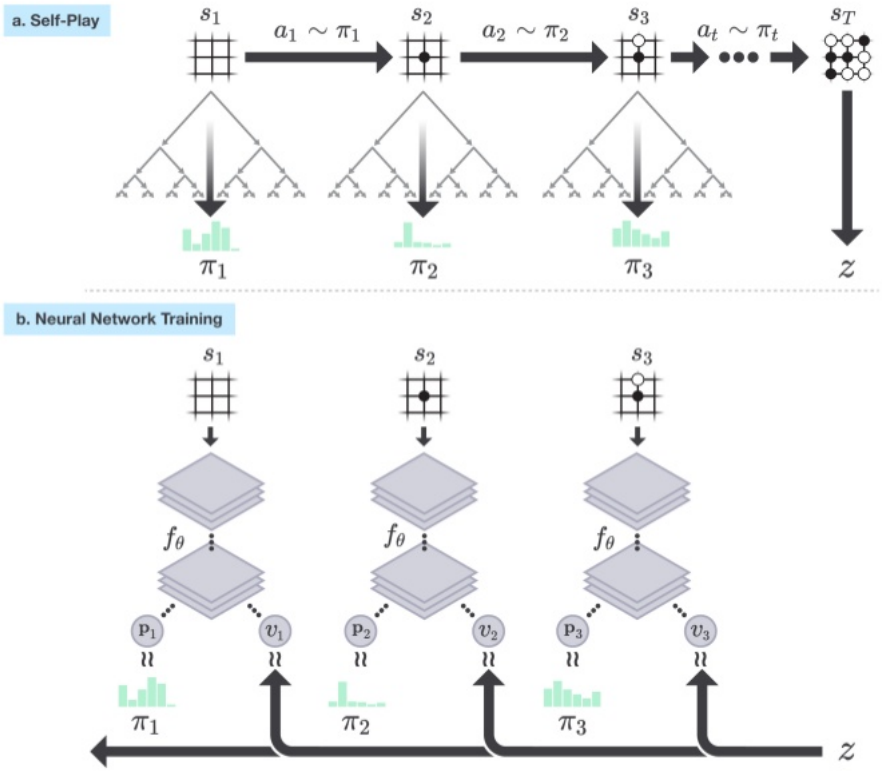
Alpha-Othello Zero 的核心思想是：MCTS 算法生成的对弈可以作为神经网络的训练数据。

总的来说，Alpha-Othello Zero 分为两个部分，一部分是 MCTS（蒙特卡洛树搜索），一部分是神经网络。

针对描述当前棋盘的一个状态（位置）s，执行一个由神经网络 f_θ 指导的 MCTS 搜索，MCTS 搜索输出每一步行为（在某个位置落子）的概率。MCTS 搜索给出的概率通常会选择那些比由神经网络 $s_\theta(s)$ 给出的执行某一行为的概率要更强大。从这个角度看，MCTS 搜索可以被认为是一个强大的策略优化工具。通过搜索来进行自我对弈——使用改善了的基于 MCTS 的策略来指导行为选择，然后使用棋局结果（哪一方获胜，用-1 和 1 分别表示白方和黑方获胜）来作为标签数据——可以被认为是一个强大的策略评估工具。

Alpha Zero 算法主体思想就是在策略迭代过程中重复使用上述两个工具：神经网络的参数得以更新，这样可以使得神经网络的输出 $(p, v) = f_\theta(s)$：移动概率和获胜奖励更接近与经过改善了的搜索得到的概率以及通过自我对弈得到的棋局结果，后者用 (π, z) 表示。得到的新参数可以在下一次自我对弈的迭代过程中让搜索变得更加强大。





2.3 MCTS

MCTS 就是用来自对弈生成棋谱的.

2.3.1 步骤

- 每次模拟通过选择具有最大行动价值 Q 的边加上取决于所存储的先验概率 P 和该边的访问计数 N（每次访问都被增加一次）的上限置信区间 U 来遍历树。
- 展开叶子节点，通过神经网络 $(P(s,), V(s) = f_\theta(s))$ 来评估局面 s；向量 P 的值存储在叶子结点扩展的边上。
- 更新行动价值 Q 等于在该行动下的子树中的所有评估值 V 的均值。
- 一旦 MCTS 搜索完成，返回局面 s 下的落子概率.

一般的，MTCS 包括 Select,Expand,Backup,play 四个步骤，我们下面对这四个步骤进行详细的讲解。

2.3.2 参数介绍

在 MCTS 里面，每一个边存储四个信息

- Q(s, a): 平均收益：这个就是前文提到的状态 - 动作值函数。
- N(s, a)：访问次数
- W(s,a): 总收益

- P(s,a): 出现状态 s 并且选择了动作 a 的先验概率。这个先验概率就是神经网络预测的落子概率。

2.3.3 Select

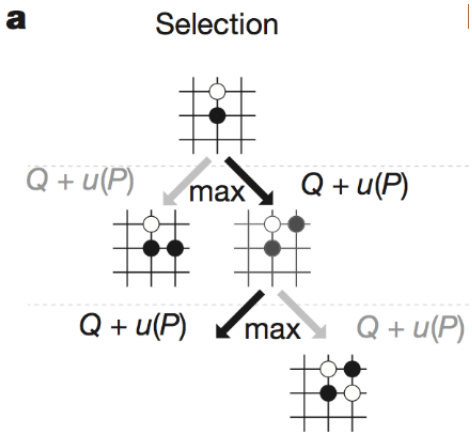
每次模拟的过程都一样，从父节点的局面开始，选择一个走子。比如开局的时候，所有合法的走子都是可能的选择，那么我该选哪个走子呢？这就是 select 要做的事情。每一次模拟的第一个阶段起自搜索树的根节点 s_0，在第 L 个时间步结束于搜索树的叶节点 s_L。对于其中的任意时间 $t < L$，根据搜索树内的统计数据来决定选择哪一个模拟行为。MCTS 选择 $Q(s,a) + U(s,a)$ 最大的那个 action。Q 的公式一会在 Backup 中描述。U 的公式如下：

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

c_{puct} 是一个决定探索水平的常数. 计算过后，我就知道当前局面下，哪个 action 的 Q+U 值最大，那这个 action 走子之后的局面就是第二次模拟的当前局面。

MCTS 在进行搜索的过程中，尝试重建状态动作值函数 Q。然后根据查表法类似的原理选择能使期望收益最大的动作。注意到这个增加了一个额外的选项 U(s,a)。为什么这个选项重要呢？有两个原因

- 即使我们 Q 的估计完全准确，如果我们每次都选最优的，那么算法很快会收敛到一个局部解跳不出来了，所以我们需要避免让算法老走一样的棋，增加一些探索其他选项的机会。这就跟小孩子学习一样，必须通过适当的犯错才能够学习到更多。
- 我们里的值函数只是一个估计值，不一定准确。



当 MCTS 第一次开始搜索时，由于所有 edge 的 $Q(s,a)$ 被初始化为 0，因此，此时行为的选择由 $U(s,a)$ 决定，而 $U(s,a)$ 又主要由 $P(s,a)$ 和 $N(s,a)$ 决定，所以在 MCTS 的搜索初期，$P(s,a)$ 越大，$U(s,a)$ 越小，那么行为 a 被选择的可能就会增大，保证了搜索算法的 exploration 性能。MCTS 的搜索后期，由于 $Q(s,a)$ 被持续更新趋向于真实值，因此，在后期，搜索主要由 $Q(s,a)$ 决定。

更通俗的解释：人在下棋过程中也会向前推演几步，人在推演的过程中所依据的就是直觉（类似于 Alpha-Othello Zero 网络输出的动作选择概率），比如在 root 节点处，人类感觉走哪

一步会比较好，就会在推演中走这一步，然后再自己分析对方可能会走在哪里，然后再自己根据直觉再走一步...，但是在推演的过程中，可能走了几步之后感觉局面变差了（Alpha-Othello Zero 网络输出的状态价值比较低），于是重头推演，重头推演的时候，动作的选择就不一样了，当走了几步之后发现局面变好了，那人类就会认为这样的下棋走法比较好，但是又怕自己推演的还不够全面，可能还有更好的走法，于是就推演很多种不同的走法（exploration），最后，人类会发现一种比较好的走法，但是在实际中，这样的推演对计算能力的要求太高，也就只有计算机能够完成，人类也只能推演少数走法。

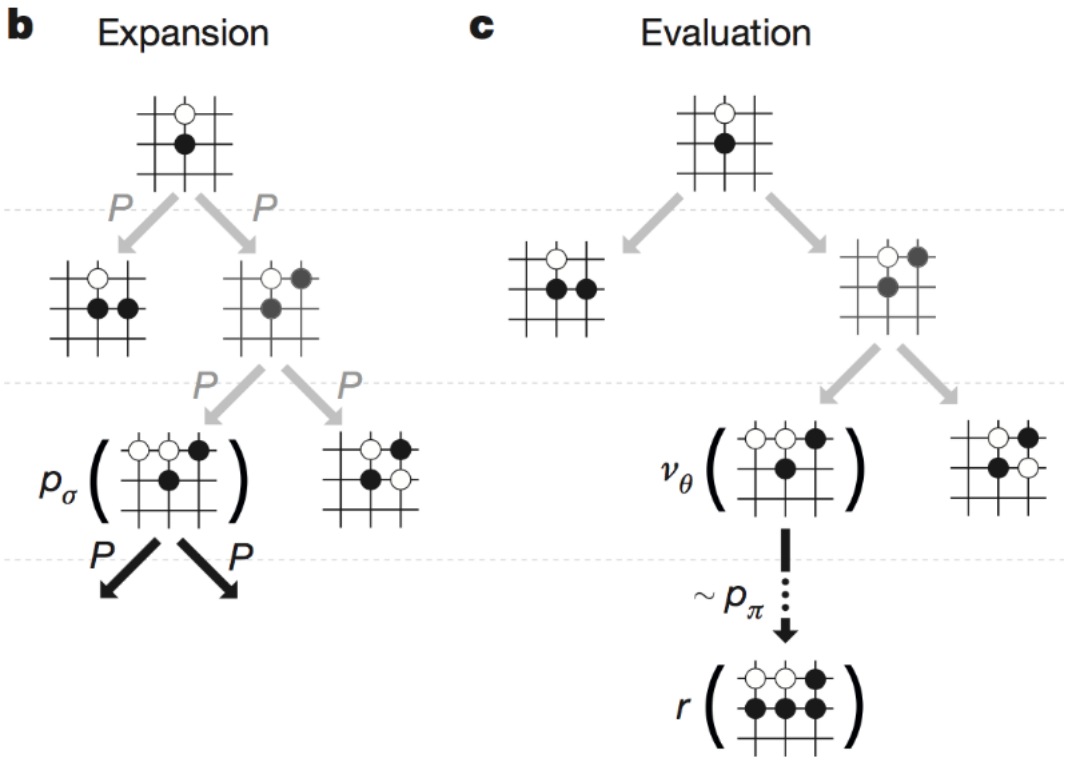
2.3.4 Expand

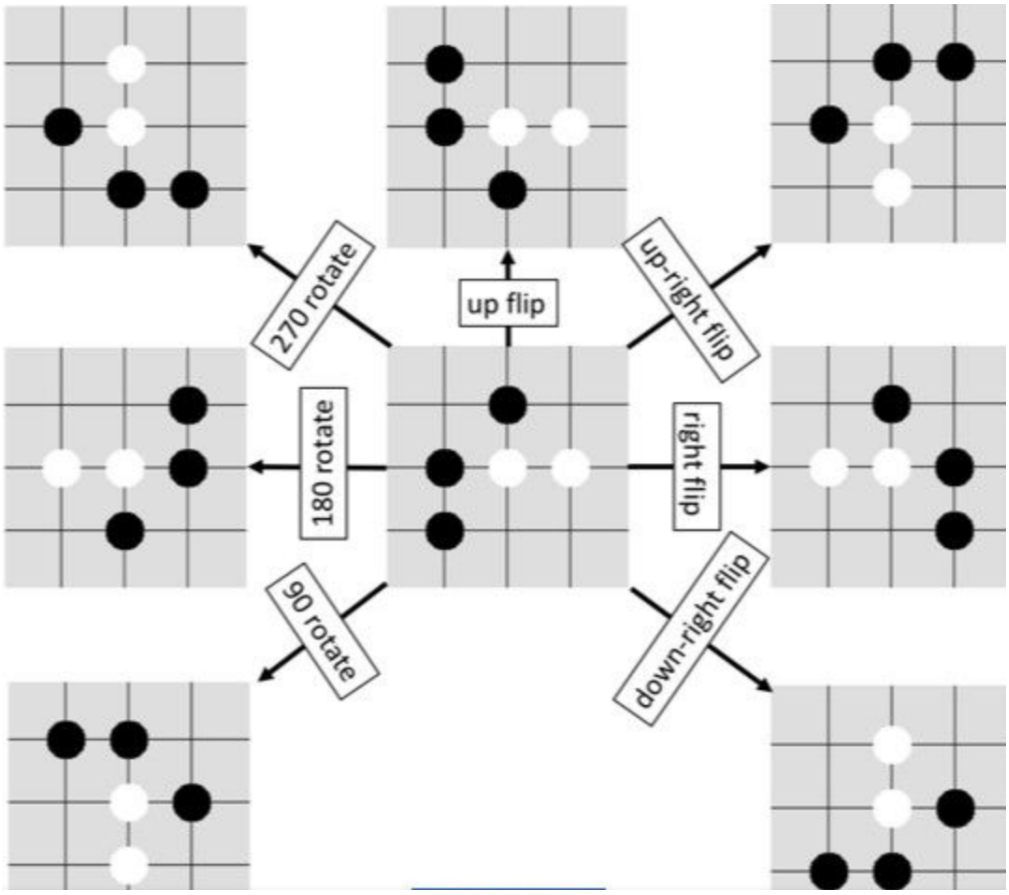
现在开始第二次模拟了，假如之前的 action 是当头炮，我们要接着这个局面选择 action，但是这个局面是个叶子节点。就是说当头炮之后可以选择哪些 action 不知道，这样就需要 expand 了，通过 expand 得到一系列可能的 action 节点。这样实际上就是在扩展这棵树，从只有根节点开始，一点一点的扩展。

叶节点 s_L 将会等待来自神经网络的评估 $(d_i(p), v) = f_\theta(d_i(s_L))$，其中 d_i 是一个 dihedral reflection 或 rotation(棋局旋转 90ř 的整数倍、棋局翻转不会影响棋局),$i \in [1\ldots,8]$。这些等待评估的状态会被送入一个队列，在神经网络评估队列里的状态时，搜索将被暂时锁定。当神经网络得到结果后，该叶节点会被展开，同时每一条可能的边 (s_L, a) 会以下面的数据进行初始化：

$$N(s,a) = 0, W(s,a) = 0, Q(s,a) = 0, P(s,a) = p_a$$

同时来自神经网络的对该叶节点的价值估计也会影响路径中每一个节点的统计数据 W。随后进行回溯过程。



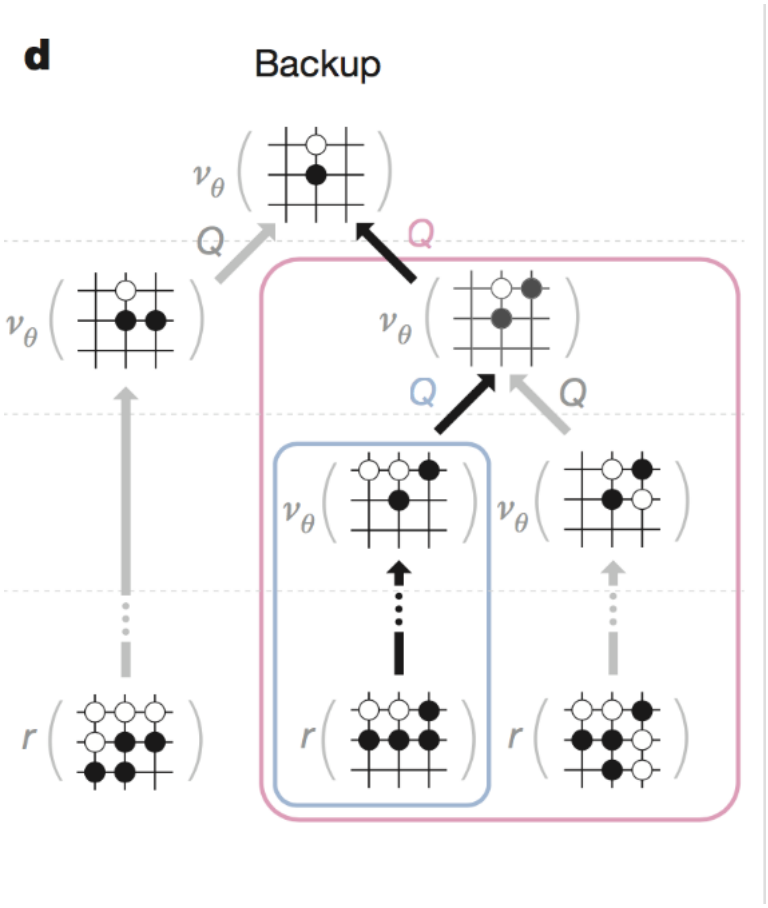


2.3.5 Backup

等一次推演结束后，Alpha-Othello zero 会根据推演的结果更新自己的知识，也就是值函数 $Q(s,a)$。

将上一步中计算出的叶子节点的价值向后传播，传播的结果就是从 root 节点 s_0 到叶子节点 s_L 的整个搜索路径上搜索到的节点都被更新了一遍，更新公示如下：

$$
\begin{aligned}
N\left(s_t,a_t\right) &= N\left(s_t,a_t\right)+1 \\
W\left(s_t,a_t\right) &= W\left(s_t,a_t\right)+v \\
Q\left(s_t,a_t\right) &= \tfrac{W(s_t,a_t)}{N(s_t,a_t)}
\end{aligned}
$$



2.3.6 Play

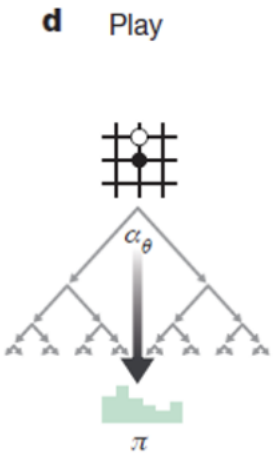
在上面的步骤执行了多次后之后，Alpha-Othello Zero 在 root 节点处选择真实的落子动作，动作选择的概率如下式所示：

$$\pi\left(a|s_0\right) = N\left(s_0, a\right)^{1/\tau} / \sum_b N\left(s_0, b\right)^{1/\tau}$$

参数 τ 用于控制动作选择的随机性，用于控制 exploration 的程度。

对该公式的理解就是，在之前的搜索模拟过程中，哪个动作被更多的采用，那么在真实的动作选择时，这个动作就更容易被选择到，因为在搜索的后期，动作的选择主要由 $Q(s,a)$ 决定，所以，在搜索过程中，$Q(s,a)$ 大的动作的 $N(s,a)$ 也会更大一些，那么在选择真实动作时，就会间接的更倾向于选择 $Q(s,a)$ 大的动作。

在执行完这真实的一步动作之后，当前的新节点变为 MCTS 的新 root 节点，重新回到前面的阶段，从新的 root 节点往下搜索。



2.4 神经网络

MCTS 算法生成的对弈可以作为神经网络的训练数据。还记得我们前面说过的深度学习最重要的部分吗？输入、输出、损失！随着 MCTS 的不断执行，下法概率及胜率会趋于稳定，而深度神经网络的输出也是下法概率和胜率，而两者之差即为损失。随着训练的不断进行，网络对于胜率的下法概率的估算将越来越准确。这意味着什么呢？这意味着，即便某个下法 AGZ 没有模拟过，但是通过神经网络依然可以达到蒙特卡洛的模拟效果！也就是说，我虽然没下过这手棋，但凭借我在神经网络中训练出的“棋感”，我可以估算出这么走的胜率是多少！

Alpha-Othello Zero 的对弈过程只需应用深度网络计算出的下法概率、胜率、MCTS 的置信区间等数据即可进行选点。

在 Alpha-Othello Zero 里使用的训练样本来自于自己和自己博弈的训练数据（self play）。每一步 Alpha Zero 都会根据 MCTS 来选择落子处，直到分出胜负。胜者的单步奖励（记为 z）是 + 1，负者的单步奖励是 - 1。Alpha-Othello Zero 里面的神经网络实际上是把 Alpha-Othello 里面的 Policy Network 和 Value Network 糅合在一起了，所以这个神经网络也有两个目标

- 让输出的落子概率 p 和 MCTS 的输出越接近越好
- 让预测的值函数 v 和实际的奖励 z 越接近越好

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

2.4.1 强化学习的过程

- 自我对弈过程 $s_1, \ldots, s_T$。在每个状态 s_t, 使用最近一次的网络 f，执行一次 MCTS, 下法根据 MCTS 计算的搜索概率而选择，$a_t - \pi_t$. 评价终止状态 s_T，根据游戏规则来计算胜利者 z。
- Alpha-Othello Zero 的神经网络训练。网络使用原始的棋盘状态 st 作为输入，通过数个卷积层，使用参数 ，输出有向量 pt, 表示下法的分布概率，以及一个标量 vt，表示当前玩家在 st 的胜率。网络参数 将自动更新，以最大化策略向量 pt 和搜索概率 t 的相似性，并最小化预测赢家 vt 与实际赢家 z 的误差。新参数将应用于下一次自我对弈 a 的迭代。

b: Alpha-Othello Zero 的神经网络训练。网络使用原始的棋盘状态 st 作为输入，通过数个卷积层，使用参数 ，输出有向量 pt, 表示下法的分布概率，以及一个标量 vt，表示当前玩家在 st 的胜率。网络参数 将自动更新，以最大化策略向量 pt 和搜索概率 t 的相似性，并最小化预测赢家 vt 与实际赢家 z 的误差。新参数将应用于下一次自我对弈 a 的迭代。

2.4.2 self-play

Self play 包括 3 部分，每一部分都使用多线程并行执行，如下：

1. optimisation：

neural network 的参数 θ_i 不断地使用最新的 self play 的数据来优化更新，使得网络输出的动作选择概率以及状态价值更可靠（类似于人的直觉，专业棋手的直觉相对于业余棋手对于棋局的判断的直觉更准确，这是因为专业棋手经过了更多的训练），AlphaGo 使用了 64 个 GPU 线程（workers），每个线程都有自己的网络参数，所以就有 64 个 θ_i ，每个 worker 的训练数据是从最近的 50 万盘 self play 的棋局中随机抽取的，其 batch_size 是 32。每训练 1000 步，保存一个 checkpoint。

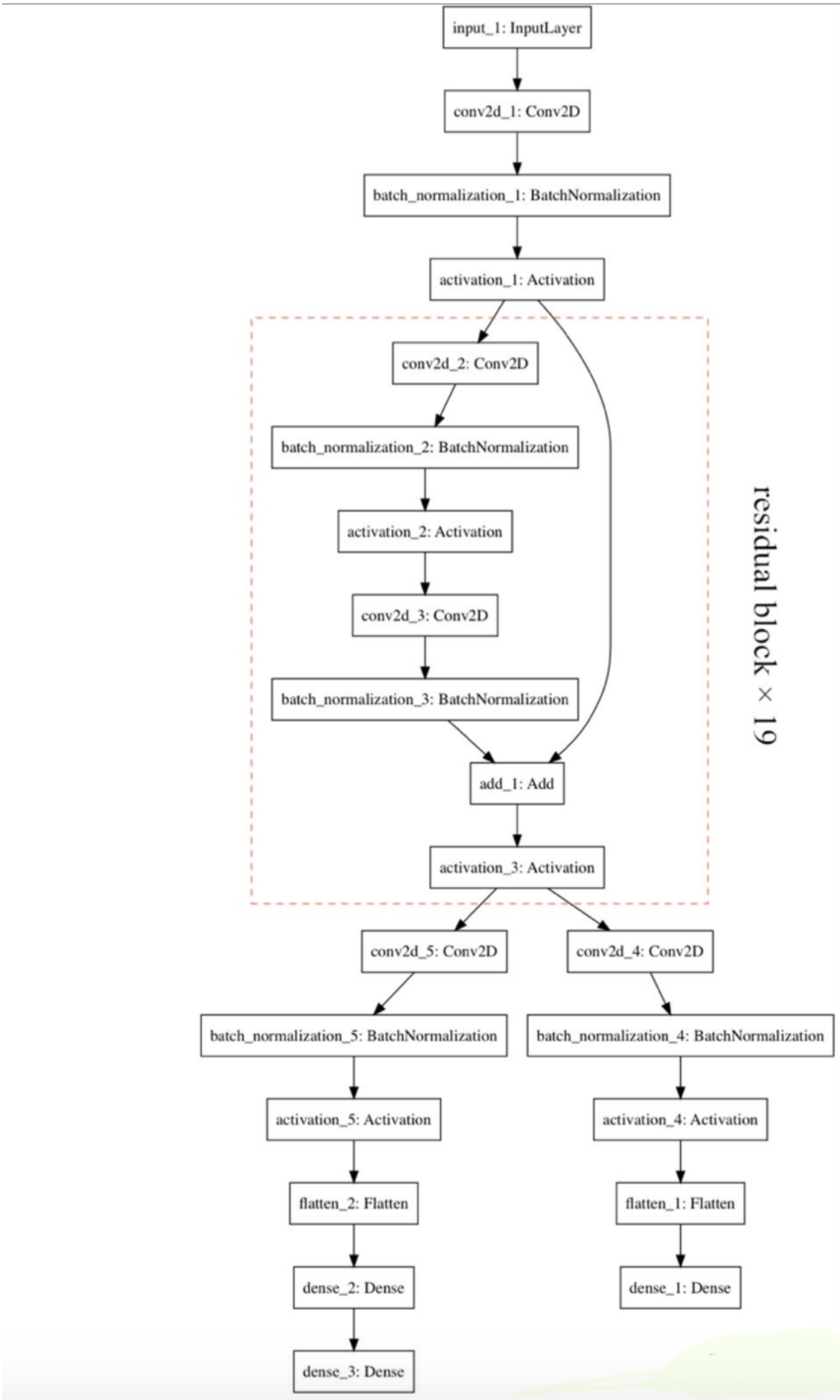
2. evaluator：

为了保证 self play 的棋局数据是目前最优的，我们需要评估每个 worker 的性能，假如某个 worker 优于现在最好的 worker，那么就让这个 worker 作为当前最好的 worker，并让这个最好的 worker 生成后续的 self play 的棋局数据。评估的办法就是让这两个 worker 对弈，如果参数为 θ_i 的 worker 对参数为 θ_* 的 worker 的取胜概率超过了 55%，那么就认为这个 θ_i 的 worker 优于 θ_* 的 worker。

3. self-play：

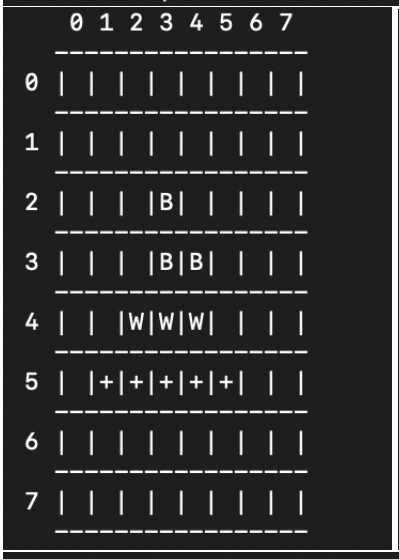
当前最好的 worker 用于生成 25000 局的 self-play data，在 self play 过程中，每一步都执行 1600 个 MCTS 来选择最终的真实动作，作者说这 1600 个 MCTS 仿真大约需要执行 0.4s，在每一局中的前 30 步，将 τ 设置为 1，之后将 τ 设置为一个很小的数，用于改变 exploration 的程度。另外，作者还通过给 root 节点的动作选择概率添加噪声的方法来增加 exploration。另外，作者为了降低计算量，会有 90% 的几率将肯定会输的棋局提前放弃。

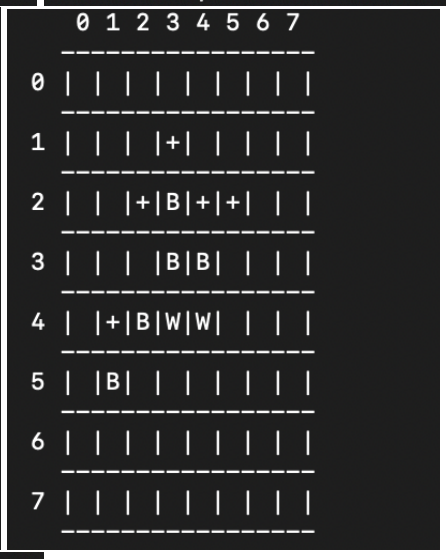
但是在我实际训练的过程中，因为硬件的限制，我并没能做到并发，我的实现是这样的：在每一次 self-play 的时候，多次采样进行训练。

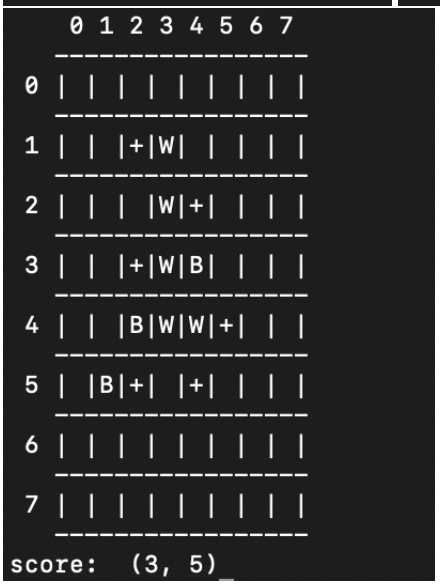

input_1: InputLayer
conv2d_1: Conv2D
batch_normalization_1: BatchNormalization
activation_1: Activation
conv2d_2: Conv2D
batch_normalization_2: BatchNormalization
activation_2: Activation
conv2d_3: Conv2D
batch_normalization_3: BatchNormalization
add_1: Add
activation_3: Activation
residual block × 19
conv2d_5: Conv2D
conv2d_4: Conv2D
batch_normalization_5: BatchNormalization
batch_normalization_4: BatchNormalization
activation_5: Activation
activation_4: Activation
flatten_2: Flatten
flatten_1: Flatten
dense_2: Dense
dense_1: Dense
dense_3: Dense

2.5 实验效果

这里我们只用一种情况来说明: 电脑后手。







但是因为硬件的原因，我们的效果并没有期待中的好，前期对战每次都会处于优势地位，

但是最后就失败了，可能是因为训练的太少了，有一个很有趣的现象是我的 Ai 不会占角，可能对于这种棋盘比较小的游戏来说，高层的神经网络并不太实用。

3 分工

何家栋：DQN 全部的代码和实验报告
刘斯宇：Alpha-Othello Zero 的所有实验报告和代码