期末project

姓名	学号
欧阳浩岚	18340133
彭永昕	18340138

一、实验原理

本次实验的目的是利用强化学习的方法来完成黑白棋。

（一）黑白棋

在具体阐述实验原理之前，有必先简要要描述黑白棋的主要的游戏规则，对于设计游戏的路程是必要的，也能帮助我们更加有针对性地设计算法。

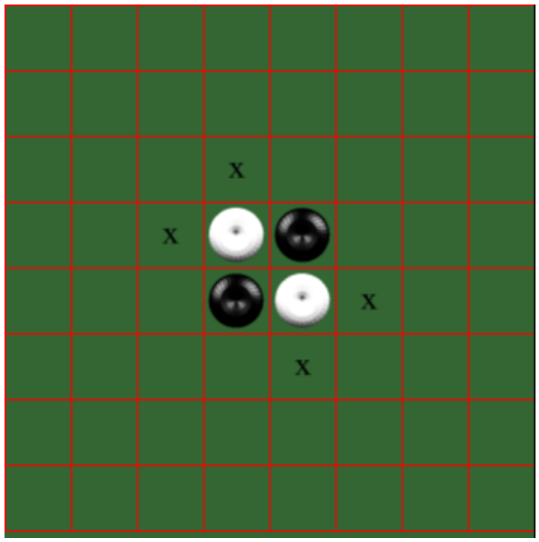


图-1 黑白棋示例

棋局开始时黑棋白棋各两颗交叉分布在棋局的中央，黑方先行，而后交替下棋。新落下的棋子与棋盘上已有的同色棋子间，对方被夹住的所有棋子都要翻转过来。横向、竖向、斜向皆可，夹住的位置需要全部是对手的棋子，不能留有空格。对于一方，落子必须能偶翻转对手至少一个棋子，否则就不能落子，对手接着下棋。棋局持续下去，直到棋盘填满或者双方都无合法棋步可下，最后统计棋子数多者获胜。

从上面的规则简介，以及实践尝试的经历，我们知道黑白棋某一个时刻的局面很可能不代表其最后结果，甚至最后一刻全面翻盘也有可能，这意味着我们设计的模型需要能够着眼于长远利益。此外，黑白棋的棋局是对称的，以90度及其倍数旋转、镜面翻转等操作下的棋局其本质上是一致。值得一提的是，黑白棋存在平局（32：32）。

（二）强化学习

在之前的学习中，我们主要把视线放在监督学习上。在监督学习中，我们需要准备一些带有标签的数据，用这些数据来训练模型使其达到我们所需要的功能。但是在实际中，这些数据的标签的获取可能需要很大的代价，或者标签本身也参杂噪声，有时一些人类的知识甚至会给模型的优化设定上界，带来限制。就我们本次的实验黑白棋来说，棋局数量可能性非常多，也很难给各种棋局下的正确动作“打标签”，不太适合仅使用监督学习来训练模型。

而强化学习（Reinforcement Learning , RL）是指一类从与环境的交互中不断学习的问题以及解决这类问题的方法。强化学习问题即智能体在与环境的交互中来学习，来实现特定的目标。并且强化学习的过程中，每一个动作也并不能直接获得监督信息，往往需要通过模型的最终监督信息来得到，具有一定的延时性。所以我们需要解决的问题是如何从这些直接得到的监督信息中，来在交互过程中的每一个状态下选出比较恰当的动作。

强化学习中主要有两个对象，分别是智能体（Agent）和环境(environment)。智能体能够感知环境的状态（state）以及获得的反馈（reward），能够将这些作为经验来学习与决策。智能体做出动作(action)也能够影响环境，使得环境改变状态，并反馈给智能体。智能体的决策功能是根据外界环境的状态来作出不同的动作，学习功能是指根据外界环境的奖励来调整自己策略。

为了描述上面这些定义与过程，有以下一些要素：

- 状态 s 是对环境的描述，并且属于状态空间中：$s \in \mathcal{S}$。
- 动作 a 是对智能体行为的描述，属于动作空间中：$a \in \mathcal{A}$。
- 策略$\pi(a|s)$是智能体根据环境状态s来决定下一步动作a的函数。
- 状态转移概率 $P(s'|s,a)$ 是在智能体在当前环境s做出动作a的情况下，转移到状态s'的概率。
- 奖励 $Q(s,a,s')$ 即只能体在当前环境s做出动作a的情况下，环境反馈的奖励，这个奖励也可能与下一个状态s'有关。、

（三）我们使用的方法

在本次实验中，我们组仔细阅读了Deep Mind在Nature上发表的论文 *Mastering the game of Go without human knowledge* 和它的后续版本 *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* 。两篇论文分别提出了基于围棋的 *AlphaGo Zero* 和通用版的 *Alpha Zero*, 该论文提出方法是：

- 使用自对弈的 `Policy Iteration` 策略
- 利用神经网络作为 `Policy Network` 和 `Value Network`，对当前局面进行评估
- 基于神经网络的评估使用`蒙特卡洛树搜索`

我们仔细学习了论文对这些方式的分析与实现，将这些主要的思想运用到我们的黑白棋的实现中。

1、蒙特卡洛树搜索（MCTS）

1.1 算法总览

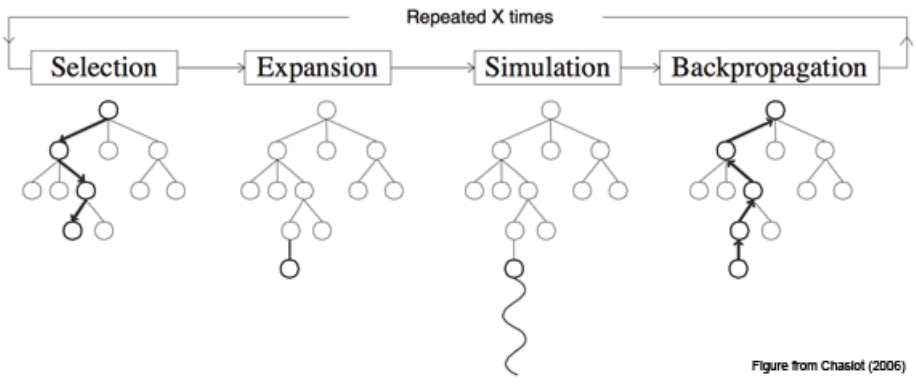


图-2 MCTS示例

我们将使用蒙特卡洛树搜索的方法。在一个典型的蒙特卡洛树搜索中，包含有`选择`、`拓展模拟`、`反向传播`三个阶段。

1. 选择阶段，从父节点开始向下选择一个最需要被扩展的结点。如果选择的结点时一个终止结点，那么就跳到第3步
2. 拓展结点N‘，从新的结点开始rollout。在rollout中，游戏将会随机地进行，得到游戏的结局，根据结局作为这一点的评分。**与这种方法不同的是，Alpha Zero并不进行rollout，而是直接使用神经**

网络的输出作为新节点的评分。

3. 反向传播，从N'到根节点，根据这次模拟的结构来修改自己的参数。

在黑白棋中，对于某一个棋面，需要根据游戏规则来判断哪些位置可下，而后通过这些位置以及其到达的新状态继续探索，经过蒙特卡洛树搜索后会对各个探索的结点进行估值，从而来做出选择。

1.2 结合神经网络

我们通过神经网络作为蒙特卡洛树搜索的估值函数。在蒙特卡洛搜索树中，每一个结点都代表一个棋局，每一条边代表一个状态在对应的一个动作下，会转换到另一个状态。当扩展搜索树，每遇到一个新的结点，该节点的值函数用神经网络来预测，这个值还会通过搜索路径向上传。这个过程，我们会为每条搜索边维护一些数据：

$$(Q(s,a),N(s,a),P(s,a))$$

其中，$Q(s,a)$为在状态 s 下做出动作 a 所预测的回报，可以理解为价值函数；$N(s,a)$记录的是蒙特卡洛树搜索在状态 s 做出动作 a 的次数；$P(s,a)$为在状态s下做出动作a的概率预测。

利用这些数据，我们计算每条边的置信上确界 $U(s,a)+Q(s,a)$

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

其中，c_{puct}是一个常数。$U(s,a)$ 与访问次数大小成反比，通过控制c_{puct}的大小，我们可以控制蒙特卡洛树搜索的方向：

- c_{puct}过大时，倾向于广度的搜索
- c_{puct}过小时，倾向于深度的搜索

我们使用该搜索树来对下一步棋进行判断。从根节点开始，每一次模拟都计算那些能够最大化置信上确界的动作。当到达的新状态s'已经在搜索树中，就在s'上递归地调用搜索树，如果新状态不在搜索树中，就添加进来，并且初始化这条边将要维护的数据，并且会将该状态输入神经网络得到值函数，该值函数将会沿着路径上传，每一条边在模拟过程中每经过一次，对应的$N(s,a)$都会累加1，并且会更新它值函数为多次模拟的均值：

$$Q(s,a)=\frac{1}{N(s,a)}\sum_{s'|s,a\to s'}V(s')$$

其中，$V(s^{'})$时神经网络的估计值。

当然，如果到达了叶子节点，也就是游戏结束的情况，直接得到实际的回报，有 $-1,0,1$ 三种情况。

在多次模拟后，根节点上连接的各边上的$N(s,a)$的值将会是一个更好的决策的近似。在自对弈学习中，MCTS将会使用优化后的神经网络。在最后返回的策略中，形式如下：

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

可以看到策略中各动作的概率与其在探索过程中的次数的占比相关。其中 τ 温度参数，控制着探索的程度。当 τ 比较大时，如取1，得到的决策将会利用动作累计次数的进行结果的采样。举一个例子，如果其中一个动作次数占比80%，只能说它有较大概率被采纳，因为还有另外的概率会采用别的动作。这就意味着，模型在自对弈中能够有一定的探索，尝试那些"不同寻常"的步伐，这可能会失去一些短期的利益，但是从长远来看的话也许能够获得一些比较新奇的状态。当 $\tau\to 0$时，这个指数将会变得比较大，那么原来占比比较大的动作会通过更快的增长速度而占据更大的比例，此时就限制了探索，专注于计算出来的最优的动作。

2. 神经网络

2.1 总的结构

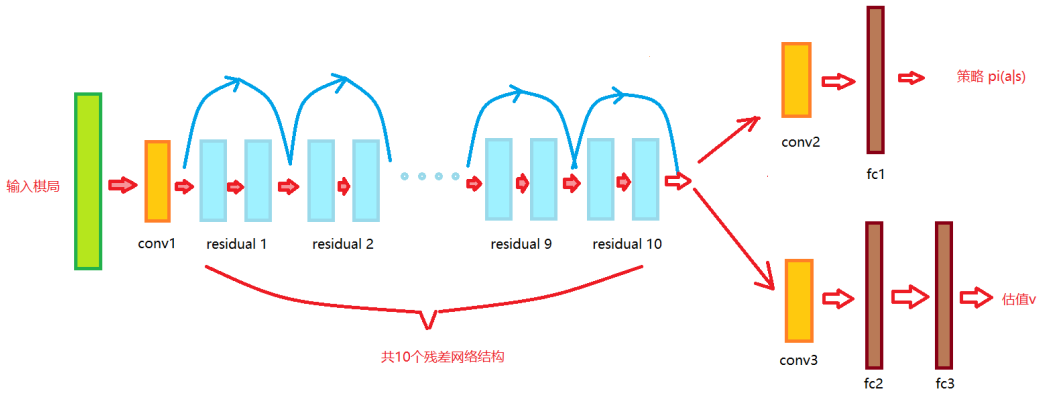


图-3 神经网络结构

神经网络的组成部分如下：

- 输入特征 $x \in R^{8*8}$
- CNN提取棋局特征
- 残差网络进一步提取特征
- 针对 Policy 的卷积神经网络
- 针对 Value 的卷积神经网络

从网络结构我们也可以看到，输出分为两个部分，一部分为对各动作预计的概率 $\pi \in R^{8*8+1}$（8*8代表下棋位置，1代表当前局面无法下），另一部分则是当前局面下预计的获胜者 $z \in R$。与围棋不同的是，黑白棋的下法不需要利用历史记录。因此，我把整个网络简化：假设当前棋手是白棋，评估当前白棋的局面。

2.2 合并值函数与策略函数

在我们的实现中，只使用了一个神经网络，该神经网络包含了 Value Network 与 Policy Network。记该神经网络为f_θ，θ为该网络的参数，以棋局的状态作为输入，有两个输出：第一个是从当前玩家视角下的状态的值函数，第二个则是一个概率向量，其内容为该状态下各动作的概率，即得到的策略。

当训练这一神经网络时，在每一次自对弈学习（self-play）结束的时候，神经网络将会被提供一些训练的样例，形式为$(s_t, \vec{\pi}_t, z_t)$。其中$\vec{\pi}_t$是对当前状态s_t下策略的估计，$z_t \in \{-1, 0, 1\}$是相对当前一方的最终结果，其中 -1 代表负，1 代表胜，0 代表平局。据此，神经网络训练的目标函数是：

$$l = \frac{1}{n}\sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}^T log\vec{p}$$

其中，n 是训练样本数，$v_\theta(s_t)$是对 t 时刻的状态 s_t 的估值，预测最终的得分，$\vec{p}$ 为网络输出的策略的向量。该函数包含估值函数的均方误差以及策略的交叉熵，优化目标是让网络输出的值函数和策略的估计尽可能接近样本的情况。

此外，在神经网络的目标函数中还包含有一个正则项 $c||\theta||^2$，这一项在我们使用的PyTorch库中的Adam优化器中已经包含。

2.3 残差结构神经网络

在神经网络中，我们学习AlphaGo Zero的实现，使用了带残差的神经网络。但也考虑到我们的实际要求，论文中的对象是围棋，而我们的是更为简单的黑白棋，所以我们的残差网络的层数也相应地有所减少，黑白棋的规模也更小，为8×8。 其结构如图所示：

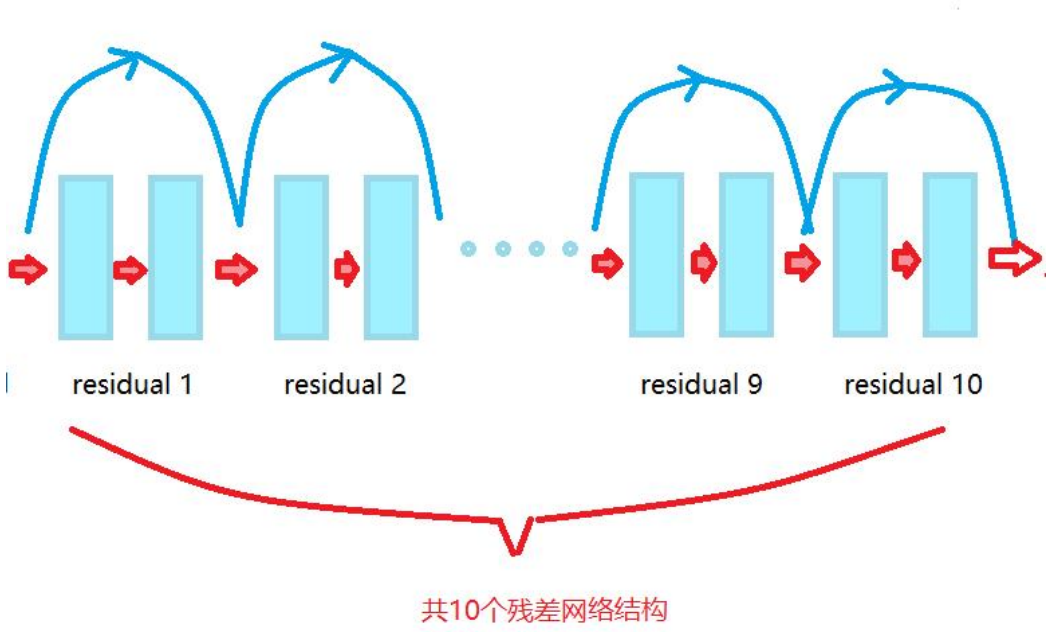


图-4 残差结构

残差结构的存在，能够解决深度网络训练的退化问题，在之前的期中项目中，我们也对ResNet结构有过尝试和实践。当增加网络层数后，网络可以进行更加复杂的特征模式的提取，所以当模型更深时理论上可以取得更好的结果，但退化问题可能会使其效果饱和甚至下降。而残差结构则可以通过"短路"连接的恒等映射来等设计来解决退化问题，使其在较深的网络中也能达到较好的训练效果。

二、实现过程

在掌握了上面的思想后，我们就可以结合黑白棋这一游戏的规则及特点来实现。我们将这些思想的实现分为若干的模块。

(一) 代码编写

1、神经网络

（1）思路

在有了之前CNN的实验基础后，调用python的PyTorch库编写一个不太复杂的神经网络是比较容易的，关键的部分在于如何设计这个神经网络。在论文中提供了一份针对围棋的神经网络的结构，其中使用了残差结构。在黑白棋中，我们也运用这一思想，但考虑到黑白棋的规则与难易程度，其规模仅为8×8, 其棋局类型规模也会比围棋少，所以我们仅使用10个残差块。输入为8×8规模的棋局状态，在残差块之后的输出，像前面神经网络图所展示的，分别进入两个卷积神经网络输出策略和值函数。

（2）超参数

- 卷积层通道数：256
- 残差块数量：10
- batch_size：256
- Epoch：10
- 优化函数：Adam

- 学习率： 1e-3

（3）关键代码展示

通过前向传播的模块来简要展示我们使用的网络模型。

```
    def forward(self, x):
        # x (batch_size, 1, n, n)
        x = x.view(-1, 1, self.args.n, self.args.n)
        batch_size = x.shape[0]

        # Input layer
        x = F.relu(self.bn1(self.conv1(x)))

        # Residual Tower
        for i in range(self.args.res_layer_num):
            x = self.residual_blocks[i](x)

        # Output layer
        # Policy network
        # policy_out (batch_size, 2 * n * n)
        policy_out = F.relu(self.bn2(self.conv2(x))).view(batch_size, -1)

        # policy_out (batch_size, n * n + 1)
        policy_out = self.fc1(policy_out)

        # Value network
        # value_out (batch_size, n, n)
        value_out = F.relu(self.bn3(self.conv3(x))).view(batch_size, -1)

        # value_out (batch_size, 256)
        value_out = self.dropout(F.relu(self.fc2(value_out)))

        # value_out (batch_size, 1)
        value_out = self.fc3(value_out)

        return policy_out, torch.tanh(value_out)
```

2、MTCS

（1）思路

代码的实现就遵循实验原理中对蒙特卡洛树搜索的描述。

首先维护三组数据 $W(s,a), N_s(s,a), P(s,a)$, 由于神经网络在策略上输出是关于各个动作的一个概率向量，所以在代码实现中，我们对这些数据的存储也以向量的形式存储。

其次在搜索树的建立和扩展过程中，情形也遵循实验原理中三种情况的描述：如果到达游戏结束的结点，我们就返回这一回报；扩展到新结点，其$P(s,a)$用神经网络获得，另外两组数据都初始化为0；如果扩展到的节点已经在树中，那么就选择一个能够最大化置信上确界的动作，达到新状态后再递归地搜索。

最后，我们还通过参数温度值τ来对搜索过程中探索的控制，原理中描述中得到的策略 $\pi(a|s_0) = N(s_0,a)^{\frac{1}{\tau}} / \sum_b N(s_0,b)^{\frac{1}{\tau}}$，其探索程度与$\tau$有关。在这里实现中，我们对$\tau$取两端的极值0和1。当$\tau$无限接近于0时，那么这个表达式中被访问次数最多的动作就会占有绝对的优势，最后策略的实行时会选择这个计算得到的最优动作。当τ取一个相对大一些的值如1，那么其策略返回各动作的概率

就是探索过程中各动作被访问过的次数，此时还能够留有探索一些新结点的可能。在我们的训练中，定义了一个改变τ的模拟过程的步数阈值（该值可以在参数中调整），当步数小于这一个阈值时，τ取1，此时探索程度相对较高，即不一定会按照计算的最优动作执行；当步数大于等于这个阈值时，就令τ取0，此时就会只按照最优动作执行而停止探索。

(2) 超参数

- 蒙特卡洛树搜索模拟次数：200
- 温度参数 $\tau = 1$ 控制时间：15步

(3) 关键代码展示

(i) 获取当前状态下各动作的概率

```
    def GetNextActionProb(self, board, player, tau=1):
        '''
        Assuming that temp
        input:
            board -- current board state
            player -- current player
            tau -- temperture parameters
        output:
            prob -- probability of each action
        '''
        n = self.args.n
        standard_board = Othello.GetStandardBoard(board, player)
        s = Othello.Board2String(standard_board)

        for i in range(self.args.num_of_mcts_sim):
            self.Search(standard_board)

        counts = self.N_s[s].numpy()

        # tau == 0
        # No extra exploration
        if tau == 0:
            best_actions = np.array(np.argwhere(counts ==
np.max(counts))).flatten()
            best_action = np.random.choice(best_actions)
            pi = np.zeros(n * n + 1)
            pi[best_action] = 1
            return pi

        # tau == 1
        # With the probability to exploration
        counts = counts ** (1. / tau)
        return counts / counts.sum()
```

(ii) 蒙特卡洛搜索过程

```
    def Search(self, standard_board):
        '''
        action size = n * n + 1 (1 for do nothing)
        input:
            standard_board -- alway for the white to move
        output:
            v -- current state value function
```

```
        '''
        n = self.args.n
        s = Othello.Board2String(standard_board)

        # 棋局终止状态
        if s not in self.end_situation:
            self.end_situation[s] = Othello.GetFinalReward(standard_board, 1)
        if self.end_situation[s] is not None:
            return -self.end_situation[s]

        # 拓展与模拟
        if self.P_s.get(s) is None:
            net_input =
torch.FloatTensor(standard_board.astype(np.float64)).to(self.args.cuda)
            with torch.no_grad():
                self.net.eval()
                self.P_s[s], v = self.net(net_input) #based on the neural
network
            self.P_s[s], v = F.softmax(self.P_s[s], dim=1).view(-1).cpu(),
v.cpu().item()

            valid_mask = Othello.GetValidMoves(standard_board, 1)
            self.P_s[s] = self.P_s[s] * valid_mask

            self.N_s[s] = torch.zeros([n * n + 1])
            self.W_s[s] = torch.zeros([n * n + 1])
            self.valid_set[s] = valid_mask
            return -v

        s_valid = self.valid_set[s]
        max_qu = -math.inf
        max_a = -1

        # 选择
        for a in range(n * n + 1):
            # Compute valid action
            if s_valid[a]:
                # PUCT algorithm
                Q_sa = self.W_s[s][a] / self.N_s[s][a] if self.N_s[s][a] !=0
else 0
                U_sa = self.args.c_puct * self.P_s[s][a] *
math.sqrt(self.N_s[s].sum()) / (1 + self.N_s[s][a])

                # Maximize the UCB(Upper considence bound)
                res = Q_sa + U_sa
                if res > max_qu:
                    max_qu = res
                    max_a = a

        next_s, next_player = Othello.GetNextState(standard_board, 1, max_a)
        next_s = Othello.GetStandardBoard(next_s, next_player)  # Get Standard
board for next search

        # 反向传播
        v = self.Search(next_s) #recursivly search
        self.W_s[s][max_a] += v # update
        self.N_s[s][max_a] += 1
```

```
        return -v
```

3、自对弈学习

（1）思路

在自对弈学习中，我们就可以充分利用上述实现的一些模块。在这里，我们的自对弈学习有三个主要的函数实现，功能分别为执行一轮具体的游戏过程（ ExecuteEpisode() ）、进行若干轮游戏并储存游戏数据(Learn())、将存储的游戏样本用于训练神经网络(Train())。

在函数ExecuteEpisode()中，我们基于当前的神经网络来进行决策，游戏过程中，每一步会调用MTCS模块中的GetNextActionProb()函数来决策，也正如前面所说，我们将会根据步数控制探索的温度。此外，由于黑白棋具有对称性，所以我们会将这些局面的对称的局面都加入训练的历史中去。这些训练的历史的保存形式为$(state, z, p)$, 其中$state$为某一个状态，z为最终的胜利者，p为策略。

在函数Learn()中，我们进行若干轮自对弈，并且会储存下来我们游戏的一些数据，如果储存数量大于上限将会删除掉最旧的。进行完后，就可以将这些游戏数据送入神经网络进行训练了。训练完成后将模型存储下来。在学习中我们调用了多进程来加快模型的训练。

在函数Train()中就调用pyTorch库进行神经网络的训练。其中的误差我们也根据原理中的描述，由估值的均方差和策略的交叉熵加和而成，正则项在Adam优化器中自动为我们加上。

（2）超参数

- 自对弈最大迭代次数：1000
- 每次迭代自对弈局数：100
- 训练集规模：200000

（3）关键代码展示

代码展示中，为了避免占用过大篇幅，将仅展示每个函数中较为关键的部分。

(i) ExecuteEpisode()

```
    def ExecuteEpisode(args, net):
        '''
        Self-play a match, starting with player Black
        input:
            net -- neural network
        output:
            history -- Training History
        '''
        n = args.n
        training_history = []
        game = Othello()
        mcts = MCTS(args, game, net)
        episode_step = 0
        while game.turn != 0:
            episode_step += 1

            # Get Training Example
            board = game.chessboard.chessboard
            s = Othello.GetStandardBoard(board, game.turn)
            temp = int(episode_step < args.temp_threshold) #Temperature
controlled by steps

            # Symmetries
            pi = mcts.GetNextActionProb(board, game.turn, temp)
```

```
            sym_list = Othello.GetSymmetries(s, pi)

            for b, p in sym_list:
            training_history.append([b, game.turn, p])

            # Self-play
            action = np.random.choice(len(pi), p=pi) #sample the action through
the policy
            move = (int(action / n), action % n)
            game.Play(move)

        final_board = game.chessboard.chessboard
        r = Othello.GetFinalReward(final_board, 1)
        return [(x[0], x[2], r * ((-1) ** (x[1] != 1))) for x in
training_history]
```

(ii) Learn()

下面主要展示我们使用多进程的训练过程。

```
    def Learn(self):
        '''
        Perform several self-play
        '''
        for i in range(self.curr_it, self.args.num_iters + 1):
            #Omitted Here

            # Parallel Self-play training
            manager = mp.Manager()
            return_list = manager.list()
            self.processes = []

            extra = self.args.num_episodes % self.num_cores
            for qk in range(1, self.num_cores + 1):
                iter_times = int(self.args.num_episodes / self.num_cores) +  1 *
(qk <= extra)
                self.processes.append(mp.Process(target=SelfPlay.Multi, args =
(qk, self.args, i, iter_times, return_list)))

            # Multi-Process stuff
            [process.start() for process in self.processes]
            [process.join() for process in self.processes]
            [process.terminate() for process in self.processes]
            [process.join() for process in self.processes]

            for qk in range(self.num_cores):
                examples_queue += return_list[qk]

            # save the iteration examples to the history
            # save procedure omitted here

            self.Train(train_loader) # train the net by the train_data
            self.SaveNet(i - 1)      # save the model
```

(iii) Train()

Train函数就是调用pyTorch库训练网络的一般模式，我们仅展示所使用的误差函数。

```
    def Train(self, train_loader):
        '''
        input:
            train_loader -- Training Dataset
        '''
        optimizer = optim.Adam(self.net.parameters())
        for epoch in range(self.args.epochs):
            # ... ommited here
            for data in tqdm(train_loader, desc = 'Traing Neural Net'):

                p, v = self.net(state.to(self.args.cuda))
                p, v = F.log_softmax(p, dim=1).cpu(), v.view(-1).cpu()

                #loss = (z,v)^2 + CE(pi,p)
                mse_loss = nn.MSELoss()
                ce_loss = -torch.sum(p * pi) / p.size()[0]
                loss = mse_loss(z, v) + ce_loss
                # ... ommited here
```

（二）具体过程与问题解决

（1）蒙特卡洛树搜索递归层数过大

这里解决方法是使用sys库，将递归次数调至3000。

```
import sys  # 导入sys模块

sys.setrecursionlimit(3000)  # 将默认的递归深度修改为3000
```

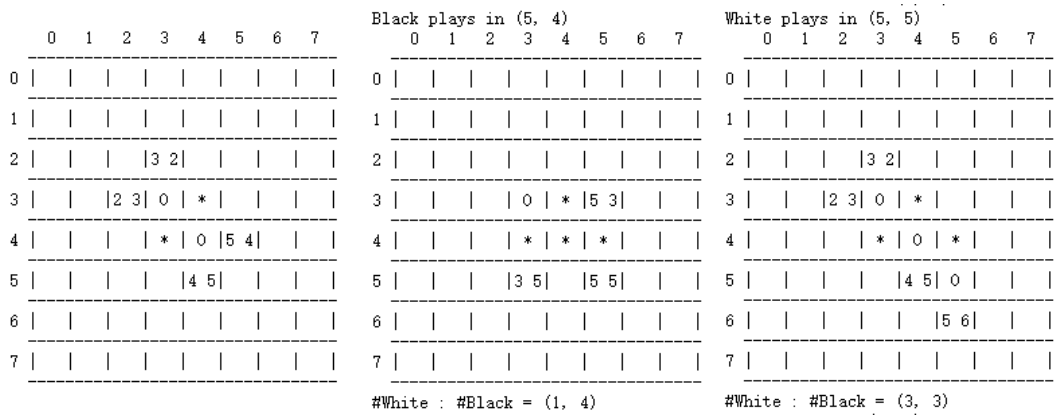
（2）自对弈时间过长

可能是因为我们的蒙特卡洛树搜索递归层数过大，导致模拟次数设为200时，100局自对弈需要3小时。值得注意的是，每一局自对弈都是独立的，互相不存在依赖，可以使用多进程进行加速。因此，我使用Python的`MultiProcessing`库来优化我们的程序，具体流程详见创新点。

三、实验结果分析

由于模型会在自对弈不断学习，为了有参考价值，以下实验均取自第26次迭代生成的模型

程序运行概览



程序运行截图

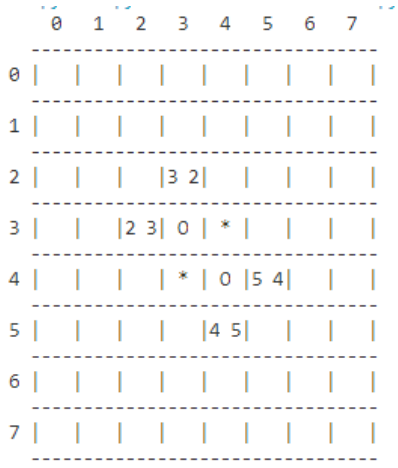
图中左边部分是程序的初始化状态，`*`代表黑棋，`o`代表白棋，数字是下一步棋可行位置的坐标，现在白棋在$(3,3)$ $(4,4)$而黑棋在$(3,4)(4,3)$处。中间部分是黑棋下在$(5,4)$处后的状态，此时白棋以$1:4$的比分落后。右图是白棋下完$(5,5)$后的局面。

自对弈过程分析

为了更加清晰地描述我们的蒙特卡洛树搜索算法，下面用两个例子进行细致的分析。

例子一

初始局面：

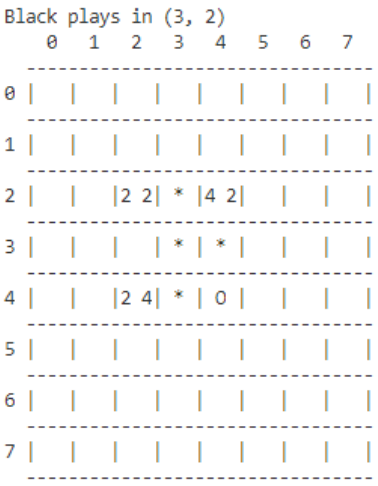


经过了200次MCTS模拟以后：根节点下四条可行边的参数如下表所示。其中，$N(s,a)$是MCTS搜索走过该条边的次数，$W(s,a)$是该条边所有子节点的收益和，而子节点的收益来自于神经网络的 Value Network。$Q(s,a)$是平均收益，$P(s,a)$是利用神经网络 Policy Network 输出的概率，$\pi(s,a)$是利用公式$\pi(a|s_0)=N(s_0,a)^{\frac{1}{\tau}}/\sum_b N(s_0,b)^{\frac{1}{\tau}}$ 得到的策略。

	(2,3)	(3,2)	(4,5)	(5,4)
$N(s,a)$	26	92	50	31
$W(s,a)$	-2.546	3.1530	0.72	-2.83
$Q(s,a)$	-0.098	0.0343	0.0144	-0.0913
$P(s,a)$	0.3073	0.1963	0.1796	0.3164
$\pi(s,a)$	0.1307	0.4623	0.2513	0.1558

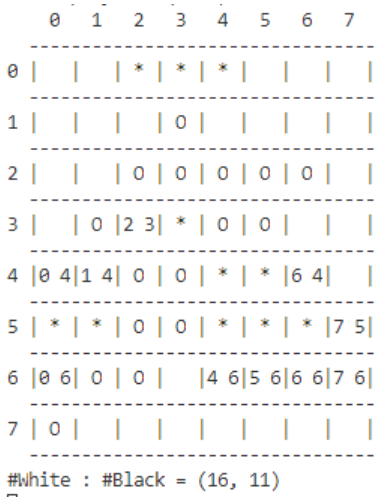
根据原理提到的算法，此时温度参数$\tau = 1$，我们将会利用$\pi(s,a)$作为概率进行可行动作的采样。

下出第一步：



例子二

搜索局面：



经过MCTS模拟以后的结果，由于已经超过了15步，模型直接执行概率最大的一步：

	(0,4)	(0,6)	(1,4)	(2,3)	(4,6)	(5,6)	(6,4)	(6,6)	(7,5)	(7,6)
$N(s,a)$	4	1	1	1	1	1	193	1	8	7
$W(s,a)$	-3.168	-0.995	-1	-1	-1	-1	-17	-1	-6.79	-5.88
$Q(s,a)$	-0.792	-0.995	-1	-1	-1	-1	-0.088	-1	-0.84875	-0.84
$P(s,a)$	0.0819	0.0	0.04	1e-5	0.0002	1e-4	0.506	0.0007	0.2	0.16
$\pi(s,a)$	0	0	0	0	0	0	1	0	0	0

经过数据比较可知，神经网络的 Policy Network 对于每个动作的估计值和经过MCTS搜索得到的价值函数分布相似，这一定程度上说明神经网络的学习是合理的。并且，本例子中大部分动作都因为模拟得到失败结果而得到一个很小的价值，反而 $(6,4)$ 因为胜率较高从而访问的次数明显多于其它几个动作，也说明我们的算法比较合理。

执行后结果：

```
White plays in (6, 4)
    0   1   2   3   4   5   6   7
  ---------------------------------
0 |   |   | * | * | * |   |   |   |
  ---------------------------------
1 |   |1 1|2 1| 0 |4 1|5 1|   |   |
  ---------------------------------
2 |   |   | 0 | 0 | 0 | 0 | 0 |   |
  ---------------------------------
3 |   | 0 |2 3| * | 0 | 0 |6 3|7 3|
  ---------------------------------
4 |0 4|   | 0 | 0 | 0 | 0 | 0 |   |
  ---------------------------------
5 | * | * | 0 | 0 | * | * | * |7 5|
  ---------------------------------
6 |   | 0 | 0 |3 6|   |   |   |   |
  ---------------------------------
7 | 0 |1 7|2 7|3 7|   |   |   |   |
  ---------------------------------
#White : #Black = (19, 9)
```

讨论

值得一提的是，温度参数的设置使得我们的决策变得不确定。这是因为，假如每一步都是确定的，那么自对弈1局和100局生成的局面都会是相同的，这样的学习是没有意义的。反而，在前15步将温度参数设为1，使得模型的走法变得多样化，而后面将温度参数设为0又让模型走的更加合理。

下面是温度参数带来的变化，在同样的神经网络参数下，多次自对弈5步，我们能够产生以下的局面：

```
Black plays in (4, 2)
    0   1   2   3   4   5   6   7
  ---------------------------------
0 |   |   |   |   |   |   |   |   |
  ---------------------------------
1 |   |   |   |3 1|   |5 1|   |   |
  ---------------------------------
2 |   |   |   |3 2| * |   |   |   |
  ---------------------------------
3 |   |   |   | 0 | * | 0 |   |   |
  ---------------------------------
4 |   |   |2 4| * | * | 0 |   |   |
  ---------------------------------
5 |   |   |   |3 5| * | 0 |   |   |
  ---------------------------------
6 |   |   |   |3 6|   |   |   |   |
  ---------------------------------
7 |   |   |   |   |   |   |   |   |
  ---------------------------------
#White : #Black = (4, 5)
```

```
Black plays in (6, 3)
    0   1   2   3   4   5   6   7
  ---------------------------------
0 |   |   |   |   |   |   |   |   |
  ---------------------------------
1 |   |1 1|   |   |   |   |   |   |
  ---------------------------------
2 |   |   | * |3 2|4 2|   |6 2|   |
  ---------------------------------
3 |   |   |   | * | * | * | * |   |
  ---------------------------------
4 |   |   |2 4| * | 0 |   |   |   |
  ---------------------------------
5 |   |   |   | 0 | * |5 5|   |   |
  ---------------------------------
6 |   |   |   |   |4 6|   |   |   |
  ---------------------------------
7 |   |   |   |   |   |   |   |   |
  ---------------------------------
#White : #Black = (2, 7)
```

```
Black plays in (4, 1)
    0   1   2   3   4   5   6   7
  ---------------------------------
0 |   |   |   |   |4 0|   |   |   |
  ---------------------------------
1 |   |   |   |3 1| * |   |   |   |
  ---------------------------------
2 |   |   |2 2| * | * |   | 0 |   |
  ---------------------------------
3 |   |   |2 3| * | * | 0 |   |   |
  ---------------------------------
4 |   |   |2 4| * | 0 |   |   |   |
  ---------------------------------
5 |   |   |   |   |   |   |   |   |
  ---------------------------------
6 |   |   |   |   |   |   |   |   |
  ---------------------------------
7 |   |   |   |   |   |   |   |   |
  ---------------------------------
#White : #Black = (3, 6)
```

```
Black plays in (3, 5)
    0   1   2   3   4   5   6   7
  ---------------------------------
0 |   |   |   |   |   |   |   |   |
  ---------------------------------
1 |   |   |   |   |4 1|   |   |   |
  ---------------------------------
2 |   |   | 0 | * |4 2|   |   |   |
  ---------------------------------
3 |   |   | 0 | * | * |5 3|   |   |
  ---------------------------------
4 |   |   | 0 | * | 0 |   |   |   |
  ---------------------------------
5 |   |   |   | * |4 5|   |   |   |
  ---------------------------------
6 |   |   |2 6|   |4 6|   |   |   |
  ---------------------------------
7 |   |   |   |   |   |   |   |   |
  ---------------------------------
#White : #Black = (4, 5)
```

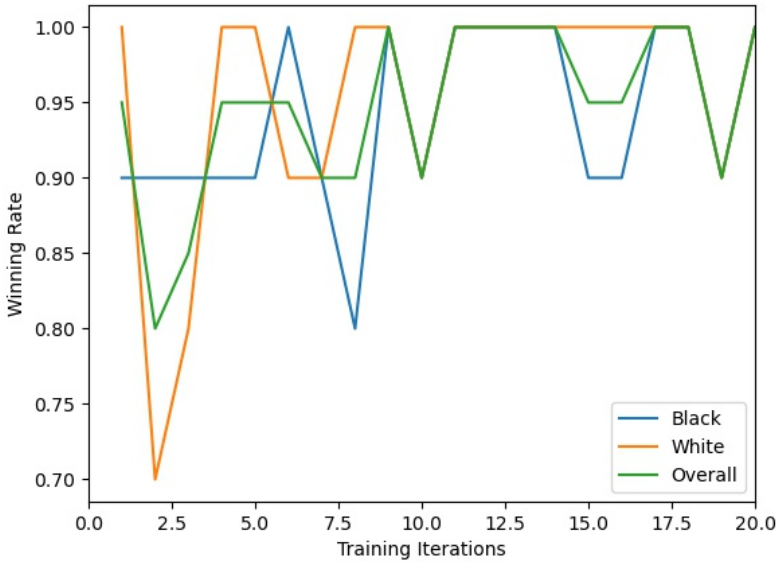
```
Black plays in (1, 4)
    0   1   2   3   4   5   6   7
  ---------------------------------
0 |   |   |   |   |   |   |   |   |
  ---------------------------------
1 |   |   |   |3 1|   |   |   |   |
  ---------------------------------
2 |   |   |2 2| * | 0 |   |   |   |
  ---------------------------------
3 |   |   |   | 0 | * |5 3|   |   |
  ---------------------------------
4 |   | * | * | * | * | * |   |   |
  ---------------------------------
5 |   |1 5|   |3 5|4 5|5 5|   |   |
  ---------------------------------
6 |   |   |   |   |   |   |   |   |
  ---------------------------------
7 |   |   |   |   |   |   |   |   |
  ---------------------------------
#White : #Black = (2, 7)
```

```
Black plays in (4, 5)
    0   1   2   3   4   5   6   7
  ---------------------------------
0 |   |   |   |   |   |   |   |   |
  ---------------------------------
1 |   |   |   |3 1|   |   |   |   |
  ---------------------------------
2 |   |   |   | * |   |   |   |   |
  ---------------------------------
3 |   |   |2 3| * | * | 0 |   |   |
  ---------------------------------
4 |   |   |   | * | * | * |   |   |
  ---------------------------------
5 |   |   |   | 0 | * |5 5|   |   |
  ---------------------------------
6 |   |   |   |   |   |   |   |   |
  ---------------------------------
7 |   |   |   |   |   |   |   |   |
  ---------------------------------
#White : #Black = (2, 7)
```

可以看出，虽然同样的参数同样走了5步，但是自对弈局面是多样化的，这就给我们神经网络学习提供了非常大量的数据。同时，由于Policy Iteration，自对弈的水平不断提高，神经网络的评估也会变得越来越强！

策略迭代结果

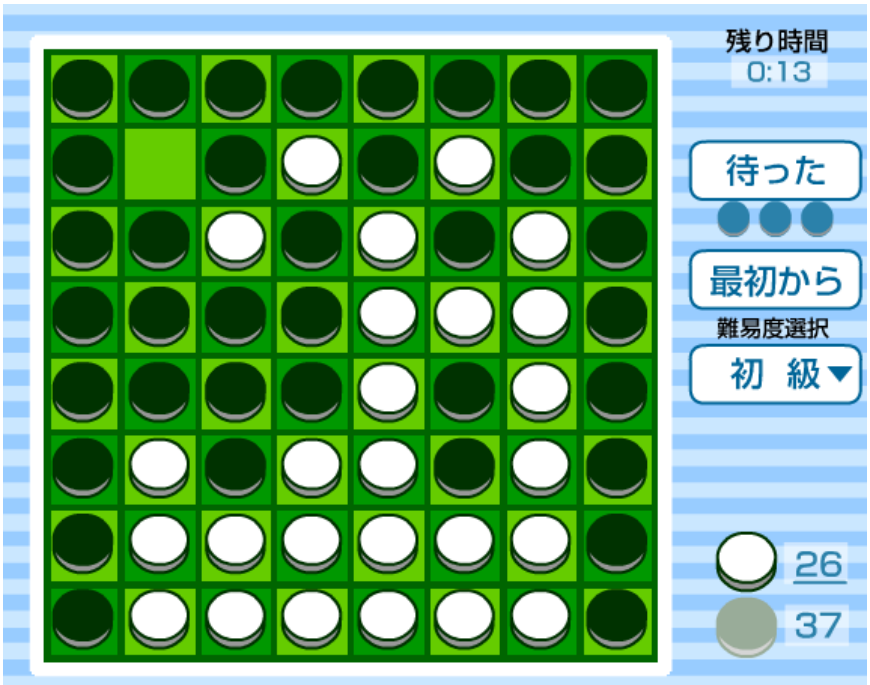
为了测试自对弈的效果，我使用随机棋手（对可用位置进行随机选择）与我们训练的模型进行PK，每个参数PK20次（10次先手，10次后手）。得到的结果：



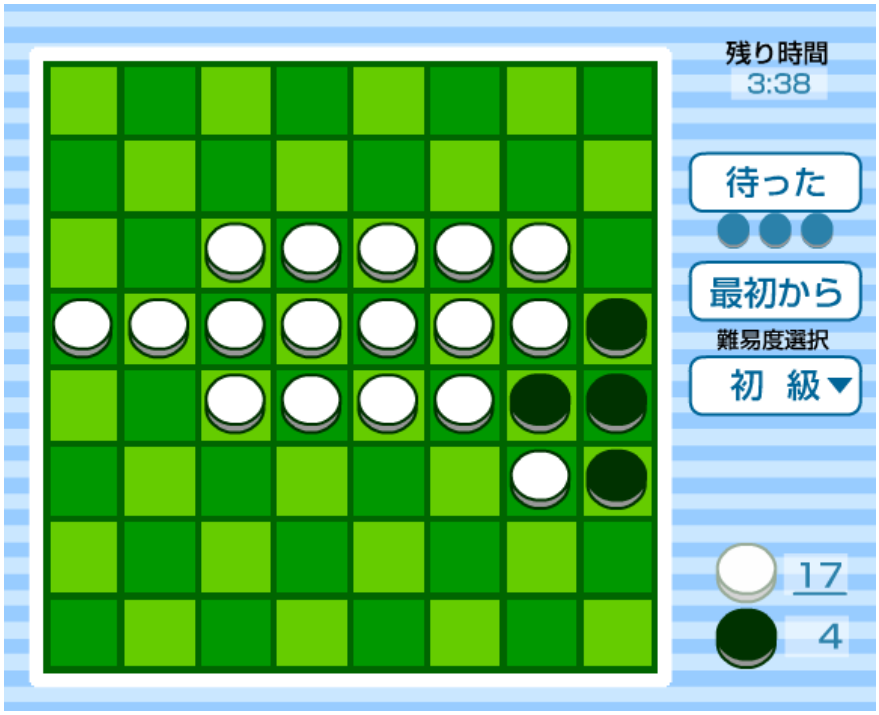
为了检验自对弈的能力，测试时我们保留温度参数。从上图可以看出，模型能力随着自对弈学习一直在提高，在11次迭代以后，模型胜率基本达到95%。

实际训练效果

除了与随机棋手PK，我还使用我们的模型与4399网站上的黑白棋游戏（http://www.4399.com/flash/66483_3.htm）进行pk。



图例1：对局实况，白棋是我们的模型



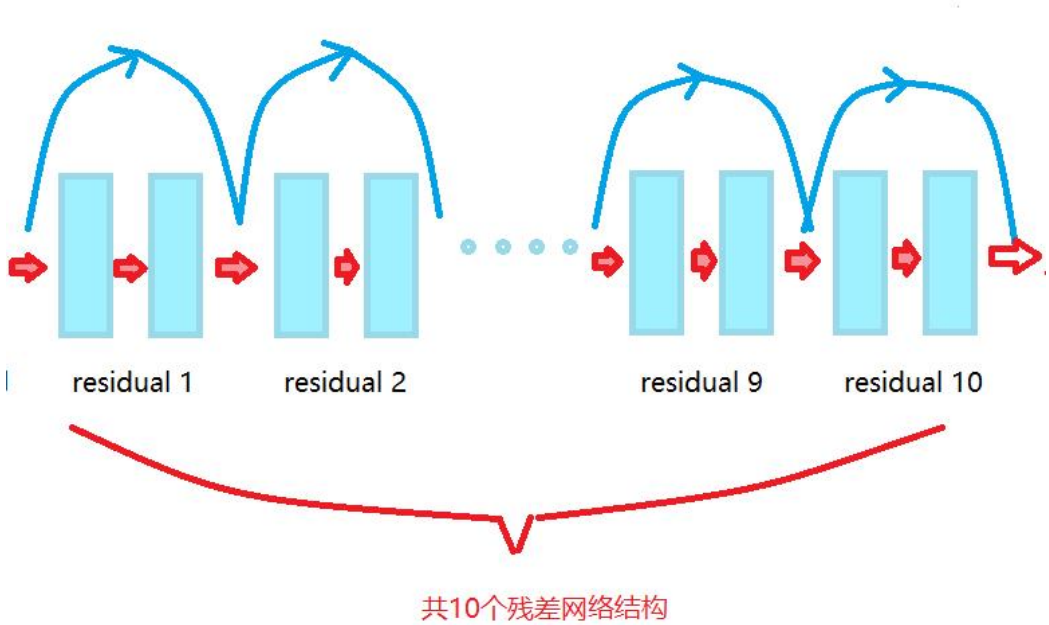
图例2：对局实况，白棋是我们的模型

可能是自对弈迭代次数太少的原因，我们与4399初级棋手多次PK中均落败了。尽管如此，我发现的一个奇怪的规律：不管是先手还是后手，我们的模型前期都下成是如图例2中大幅度领先黑棋的局面。这说明我们的模型崇尚的是进攻的风格，这与Alpha Zero原始论文的观察是一致的。

四、创新点

1、残差结构神经网络

相较于实验课学习的dqn，在神经网络中，我们学习了AlphaGo Zero的实现，使用带残差的神经网络。但考虑到我们的实际要求，论文中的对象是围棋，而我们的是更为简单的黑白棋，所以我们的残差网络的层数也相应地有所减少，黑白棋的规模也更小，为8×8。 其结构如图所示：



残差结构的存在，能够解决深度网络训练的退化问题，在之前的期中项目中，我们也对ResNet结构有过尝试和实践。当增加网络层数后，网络可以进行更加复杂的特征模式的提取，所以当模型更深时理论上可以取得更好的结果，但退化问题可能会使其效果饱和甚至下降。而残差结构则可以通过"短路"连接的恒等映射来等设计来解决退化问题，使其在较深的网络中也能达到较好的训练效果。

2、多进程加速自对弈过程

模型中的自对弈学习所运用的全部要点即为上述的各个部分。按照训练流程来描述一遍就是，首先以一种合理的方式初始化我们的神经网络，用该网络来提供值和策略的预测。随后基于这个网络，我们使用MTCS来进行若干轮的自对弈，该搜索过程能够最后接受到环境的实际回报，将提供一些更好的样本。随后这些样本可以供神经网络学习继续优化，接着再令MTCS基于这个网络进行自对弈..... 如此循环往复下去，达到一个比较好的模型。

此外，为了在比较有限的时间内训练尽量多的轮数，我们使用了多线程加速。

（1）流程

- 利用mp.Manager创建可以共享的变量，以便接收返回参数
- 根据电脑实际的核心数目，分配自对弈局数给每个进程
- 关闭多进程，继续进行下面的步骤

其中，因为神经网络的运行是在CUDA上，而CUDA原生不支持MultiProcessing库的多进程。这里，我去了讨巧的做法：将神经网络复制成7份，给每个进程分配一份。受限于电脑显存大小，最终我们是分成了7个进程，提速效果为3倍左右。

（2）关键代码

```
    # Parallel Self-play training
    manager = mp.Manager()
    return_list = manager.list()
    self.processes = []

    extra = self.args.num_episodes % self.num_cores
    for qk in range(1, self.num_cores + 1):
        iter_times = int(self.args.num_episodes / self.num_cores) +  1 * (qk <=
extra)
        self.processes.append(mp.Process(target=SelfPlay.Multi, args = (qk,
self.args, i, iter_times, return_list)))

    # Multi-Process stuff
    [process.start() for process in self.processes]
    [process.join() for process in self.processes]
    [process.terminate() for process in self.processes]
    [process.join() for process in self.processes]

    for qk in range(self.num_cores):
        examples_queue += return_list[qk]
```