测试说明

1.创建并激活环境，安装对应依赖

```
conda create -n tf-gpu tensorflow-gpu=1.12.0 python=3.6.8
```

```
conda activate tf-gpu
```

```
pip install tensorlayer==1.10.1
```

2.进入/**playing**目录，将**zjr_fl**目录放在**player**目录下

3.进入**zjr_fl**目录，执行:

```
python setup.py build_ext --inplace
```

```
its\8.1\include\um" "-IC:\Program Files (x86)\Windows Kits\8.1\include\winrt" /Tcm
cts_alphaZero.c /Fobuild\temp.win-amd64-3.6\Release\mcts_alphaZero.obj
mcts_alphaZero.c
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\BIN\x86_amd64\link.exe /nol
ogo /INCREMENTAL:NO /LTCG /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO /LIBPATH:E:\an
aconda\envs\tf-gpu\libs /LIBPATH:E:\anaconda\envs\tf-gpu\PCbuild\amd64 "/LIBPATH:C
:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\LIB\amd64" "/LIBPATH:C:\Prog
ram Files (x86)\Windows Kits\10\lib\10.0.10240.0\ucrt\x64" "/LIBPATH:C:\Program Fi
les (x86)\Windows Kits\8.1\lib\winv6.3\um\x64" /EXPORT:PyInit_mcts_alphaZero build
\temp.win-amd64-3.6\Release\mcts_alphaZero.obj /OUT:build\lib.win-amd64-3.6\mcts_a
lphaZero.cp36-win_amd64.pyd /IMPLIB:build\temp.win-amd64-3.6\Release\mcts_alphaZer
o.cp36-win_amd64.lib
mcts_alphaZero.obj : warning LNK4197: export 'PyInit_mcts_alphaZero' specified mul
tiple times; using first specification
   Creating library build\temp.win-amd64-3.6\Release\mcts_alphaZero.cp36-win_amd64
.lib and object build\temp.win-amd64-3.6\Release\mcts_alphaZero.cp36-win_amd64.exp
Generating code
Finished generating code
copying build\lib.win-amd64-3.6\mcts_alphaZero.cp36-win_amd64.pyd ->
```

显示上图，生成完毕。

4.调用接口：修改**logic**目录下**control.py**:

（1）开头添加:

```
from player.zjr_fl.zjr_fl import ZJR_FL
```

（2）第27行添加**player**:

```
self.zjr_fl = ZJR_FL()
```

（3）结尾**'else'**之前增加:

```
elif player_name in ['zjr_fl', 'zjr_fl_copy']:
    if player_name == 'zjr_fl':
        pos_x, pos_y = self.zjr_fl.xiazi(self.playerjudger, color, step)
    else:
        try:
            pos_x, pos_y = self.zjr_fl_copy.xiazi(self.playerjudger, color,
step)
         except:
            self.zjr_fl_copy = ZJR_FL()
            pos_x, pos_y = self.zjr_fl_copy.xiazi(self.playerjudger, color,
step)

    print(player_name+'\'s decision:','x =',pos_x, 'y =',pos_y)
    return pos_y, pos_x
```

5.修改**main.py**第27行：

```
player_list = {'0':'pc',         #E.g. yourself control the pc
'1':'chenna',
'2':'easyai',
'3':'zjr_fl'}
```

算法原理

我们的算法主要采用蒙特卡洛树搜索(Monte Carlo Tree Search, MCTS)算法的强化学习和神经网络实现的。

定义：将状态-动作空间(s,a)，存储为$(N(s,a),W(s,a),Q(s,a),P(s,a))$。

其中，$N(s,a)$表示节点访问次数，$W(s,a)$表示动作的总价值，$Q(s,a)$表示动作的平均价值，$P(s,a)$表示在棋盘状态为s下选择动作a的先验概率。

（1）**模拟落子**，以预知每一种动作所带来的后果。当每次轮到我们下棋时，算法从MCTS的根节点s_0开始搜索，当到达时间步长 （搜索深度）为L 的叶子节点 s_L 时，对其进行扩展，并更新所有节点，完成一次搜索，具体如下：

首先，在每个时间步长$t<L$，根据$a_t=argmaxQ(s_t,a)+U(s_t,a)$,选择一个动作$a_t$。

其中，$U(s,a)=C(s)P(s,a)\sqrt{N(s)}/(1+N(s,a))$，$N(s)$是父亲节点的访问次数，$C(s)$是探索速率，它会随着搜索时间缓慢增长。当搜索到叶子节点 s_L时，会将它送入神经网络$(p,v)=f_\theta(s_L)$进行评估。其中，p表示网络输出执行该动作的概率，v表示网络输出执行该动作带来的价值。

然后，对 s_L进行扩展。扩展步骤为:

- 先将每个(s_L,a)初始化为$N(s_L,a)=0,W(s_L,a)=0,Q(s_L,a)=0,P(s_L,a)=p_a$。
- 再对访问次数和动作的价值在每一步$t\leq L$上做更新，即$N(s_t,a_t)=N(s_t,a_t)+1$，$W(s_t,a_t)=W(s_t,a_t)+v$，$Q(s_t,a_t)=W(s_t,a_t)/N(s_t,a_t)$。值得注意的，$W$是相对于我们作为玩家来说的，当搜索到的节点是对方玩家（e.g. ,easyai,chenna...），那么动作的价值需要取反。

最后，节点更新完毕，一次搜索结束。

以上搜索过程可以重复多次，重复次数越多，计算消耗时间越大，但同时决策越智能。

（2）**真正落子**，真正做出决策到棋盘上。

计算根节点s_0 的搜索概率，记为$\pi = \alpha\theta(s)$，它与每个动作的访问次数成正比，即$\pi \propto N(s,a)^{1/\tau}$，$\tau$为温度系数。当$\tau = 1$时，按照动作概率采样（偏于探索），当$\tau$越接近0，则越偏向选择当前最优动作。进而，通过$\pi_t$ 进行采样来选择当前的动作。

（3）**自我对弈（Self-play）**到游戏结束。当两个玩家有一方获胜或者游戏超过最大长度时，在步骤T终止，并对游戏进行评分，即记录最终奖励$r_T \in \{-1, 0, +1\}$，其中，-1代表对方获胜，+1代表我方获胜，0代表平局。

（4）**训练神经网络。**

首先，将每个时间步t的数据存储为$\{s_t, \pi_t, z_t\}$，其中$u_t = \pm r_T$，它表示在t步，处于当前玩家的角度来看的游戏赢家。

特别地，我们添加γ为当前环境的折扣率，则$z_t = u_{t+1} + \gamma u_{t+2} + \ldots + \gamma^{n-1} u_{t+n} + \gamma^n v_{t+n}$，以表现将来的奖励对现在的影响。

然后，根据(s, π, z)对新的网络参数θ进行训练。$(p, v) = f_\theta(s_L)$以最小化预测值v与自玩赢家获得奖励z之间的误差，并使神经网络动作概率 p 与搜索概率 π的相似性最大化。

最后，更新网络参数。网络参数θ通过以下损失函数L上的梯度下降进行更新，该损失函数分别为均方误差和交叉熵损失求和得到：

$$l = (z - v)^2 - \pi^T \log p + c||\theta||^2)$$

其中，c控制 L2 权重正则化水平的参数，以防止过拟合。

（5）多次重复以上对弈过程，对神经网络进行训练。

（6）最后，用训练好的网络，进行对战。注意：训练时，MCTS不加禁手规则，因为禁手规则的复杂性会给模型的学习带来困难，导致模型收敛过程不稳定。但是在真正对战时，MCTS增加了禁手规则。

其他细节描述：

（1）对于评估动作的神经网络，其结构如下：神经网络采取ResNet架构，包含body以及policy-head和value-head。body由一个整流的BN卷积层组成，后跟 19 个残差块。每个这样的块由两个具有shortcuts的BN卷积层组成。每个卷积应用 256 个大小为3×3的卷积核，步幅为 1。policy-head添加一个的BN卷积层，以及139个卷积核。value-head应用一个额外的BN卷积层，其中包含 1 个大小为1×1的过滤器，步幅为 1，后跟一个大小为 256 的Linear层和一个大小为 1 的 tanh 线性层。

（2）对于送入神经网络的棋盘状态 s_L，：其维度为$15 \times 15 \times 9$。15×15代表棋盘，9来自白棋和黑棋前4步历史状态，即$4 \times 2 = 8$，以及当前执棋的颜色，即1。

（3）保存最好模型的条件：当前训练的模型和历史最好的模型对弈十局（一黑一白交替），当十局得分超过5.5，则**保存最好模型。**其中，输为0，赢为1，平局为0.5分。

（4）为了给神经网络的训练提供充足的样本，棋盘的状态s在每次输入到神经网络时，会进行翻转，对应的，输出的动作也会相应翻转回去作为输出。这样操作的意义在于，相当于进行**数据增强**，通过形成更多不同的训练样本，使得网络探索的更充分。

设计过程

针对于上面的算法原理，对代码的关键实现步骤进行描述。

（1）在**zjr_fl.py**文件定义**ZJR_FL**类，包含：

- def init( ):

`policy_value_net = PolicyValueNet()`：加载最好的神经网络模型。其中，PolicyValueNet类中定义**policy_value_function**函数用于做**棋盘翻转**。

`self.mctsplayer`：生成MCTS树,其中**n_playout为**最大搜索深度。

`self.board_buffer`：获得 棋盘状态（$15 \times 15 \times 9$)

- def xiazi( ):

`move = self.mctsplayer.get_action(self.board_buffer, print_info=True)`：根据当前状态self.board_buffer，做出动作。此函数在**mcts_reinforcement.py**中。

（2）对于**mcts_reinforcement.py**中**get_action（）**函数：

`self.mcts.get_move_visits(board)`：得到动作，访问次数和最大搜索深度。具体如下：

将当前状态输入`self._playout(state_copy)`，目的是从根到叶子节点进行一次搜索，获得各个节点的值并进行回溯。

在该函数中，在MCST中搜索当前棋盘，`node.select`根据$argmaxQ(s_t,a)+U(s_t,a)$选择下一个动作，并更新状态，记录搜索深度deep。当搜索到叶子节点时，跳出MCTS搜索，通过`self._policy_value_fn`，获得将当前状态输入到神经网络后得到网络输出的p和v，分别对应代码的action_probs,和leaf_value。

然后，检查游戏状态是否结束，如果没有结束，则通过`node.expand(action_probs,add_noise=self._is_selfplay)`扩展MCTS，并对**动作概率action_probs增加噪声**，再记录至新增节点中。如果游戏结束，则根据胜负更新叶子节点的值。最后，返回MCTS当前搜索深度deep。

重复以上搜索过程，得到若干$\{s_t,\pi_t,z_t\}$以及MCTS的最大搜索深度 max_search_depth。

当达到最大搜索次数n_playout时，结束搜索。n_playout越大，搜索次数越多，时间花费越大，但网络决策更准确。其中，在训练过程中，n_playout设置为800，推理过程中，n_playout可以设置为8000。

实际上，训练完毕后的最好网络不需要搜索，即n_playout甚至是十分小的数，比如100，也可以轻松打败easy_ai以及chenna。

`probs = softmax(1.0 / temp * np.log(np.array(visits) + 1e-10)`：设置温度系数以控制探索概率，根据访问次数得到动作概率。**在self-play前8步时，温度系数设置为0.8，8-12步时，温度系数为0.6，后面趋向于无穷小，来选择最优的动作。注意：温度系数不能设置为1，即完全根据概率采样动作，这使得几乎所有棋局在第9~20步终止，无法学习后面的更远的棋局，因此设置如上，使得前面几步的动作具有随机性，后面的动作更加精确，从而使得棋局探索的更多样化。**

`move = np.random.choice(acts, p=probs)`：根据动作概率，对动作进行采样。

`eval_win = self.mcts._root._Q / 2 + 0.5`：记录这一局我方的胜率。

（3）设置算法最终打印结果：

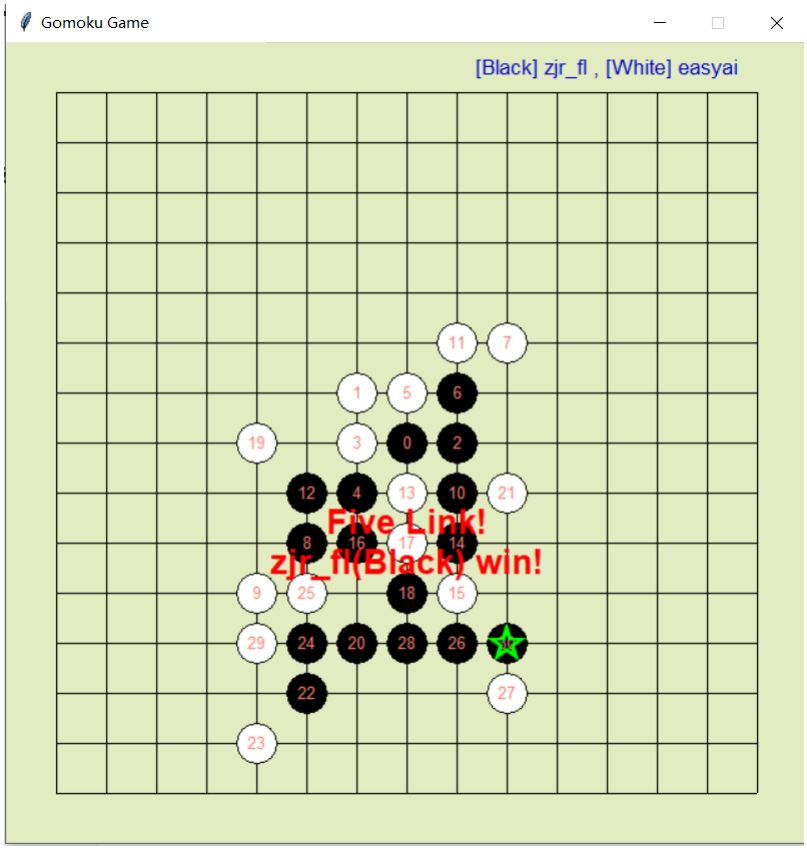
```
------------------------------   zjr_fl's turn begins !  ------------------------------

          step:                             10
          zjr_fl's decision:                x = 9 y = 6
          decision simulations:             1138
          total simulations:                1155
          max search depth:                 10
          zjr_fl's eval winning rate:       99.45%
          search time:                      7.6s

------------------------------   zjr_fl's turn ends !  ------------------------------
```

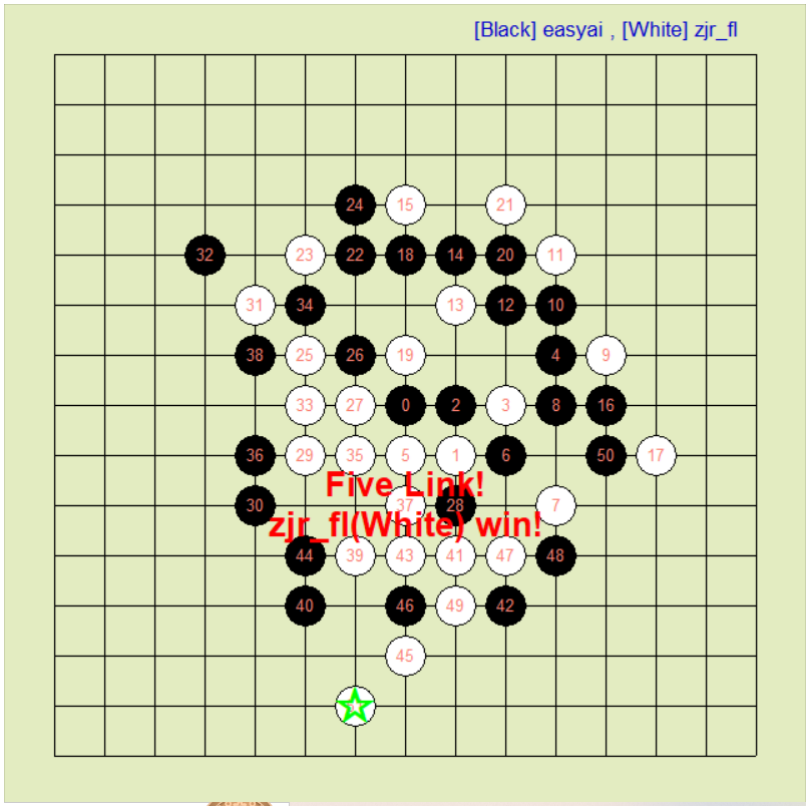
以上分别显示了：游戏步数；落子位置；执行这个动作的访问次数；根节点的访问次数；MCTS的最大搜索深度;胜率;以及搜索时长。（执行这个动作的访问次数与根节点的访问次数越接近，这个动作被选择的概率越高。）

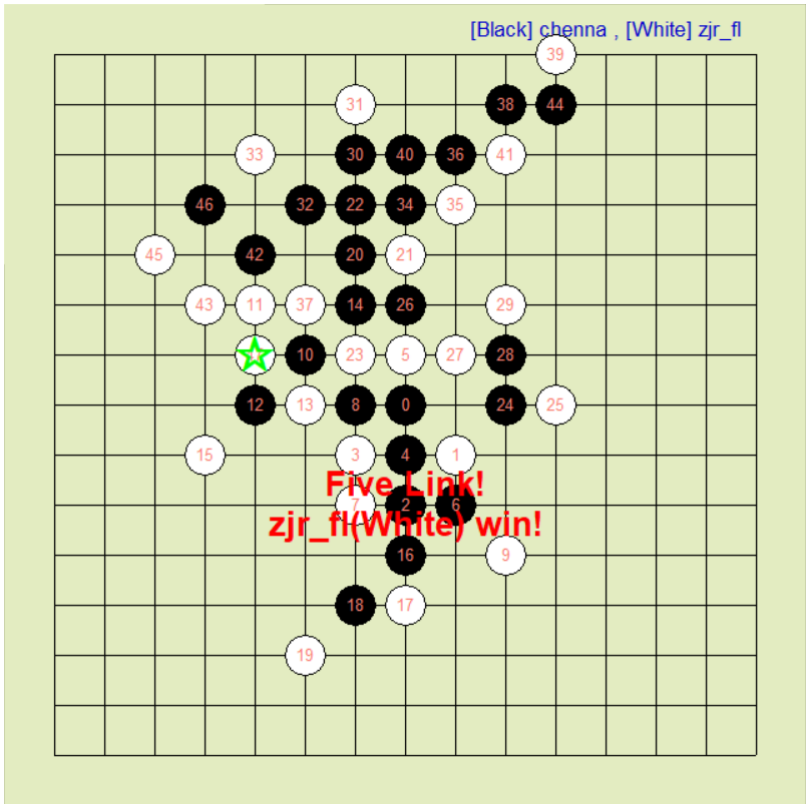
实验结果

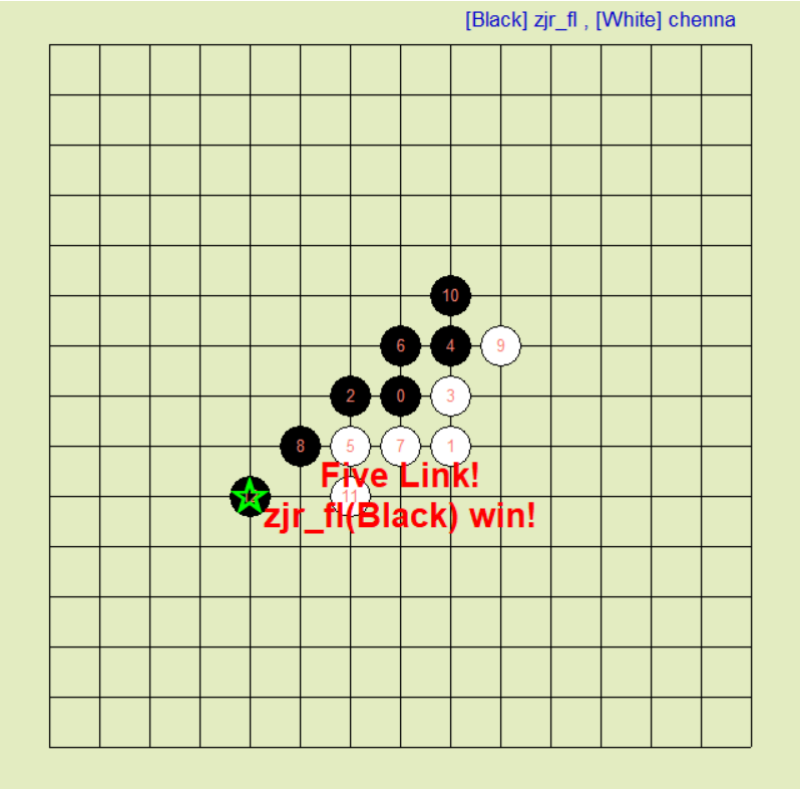
我们将self-play过程训练100,000局以上，网络训练1000个epoch,一共训练了720多个小时，占用35个self-play进程，1个训练进程，1个测试保存最好模型的进程，将训练好的模型保存至`./player/zjr_fl/policy_value_model`中。

最后，在超过50场对战中，无论对手是easy_ai还是chenna，且无论执黑还是执白，都取得100%的胜率。




[Black] easyai , [White] zjr_fl
Five Link!
zjr_fl(White) win!


[Black] chenna , [White] zjr_fl
Five Link!
zjr_fl(White) win!



创新点

（1）自我对弈，不依赖先验知识，完成学习。

（1）数据增强。在网络中增加policy_value_function函数，来对棋盘状态做翻转，使得每一次对峙的棋局都不同，从而增加训练数据，提高模型决策能力。

（2）*Cython*加速。提升速度与搜索次数正相关。搜索越多，提升越明显。例如，单步执行蒙特卡洛搜索一万次，时间从92.2s到48s.

```
-> physical GPU (device: 0, name: NVIDIA GeForce GTX 1650 Ti with Max-Q Design, pci bus i
d: 0000:01:00.0, compute capability: 7.5)
model loaded!
easyai's decision: x = 8 y = 8

------------------------------   zjr_fl's turn begins !  ------------------------------

            step:                                1
            zjr_fl's decision:                   x = 7 y = 8
            decision simulations:                4596
            total simulations:                   9999
            max search depth:                    17
            zjr_fl's eval winning rate:          None (will be available after 10 steps)
            search time:                         92.2s

------------------------------   zjr_fl's turn ends !  ------------------------------

zjr's decision: x = 7 y = 8
easyai's decision: x = 7 y = 7
```

```
------------------------------   zjr_fl's turn begins !   ------------------------------
          searching 0.0% ...          consumed time: 1.0s
          searching 9.1% ...          consumed time: 5.3s
          searching 18.2% ...          consumed time: 9.4s
          searching 27.3% ...          consumed time: 13.3s
          searching 36.4% ...          consumed time: 17.2s
          searching 45.5% ...          consumed time: 21.2s
          searching 54.5% ...          consumed time: 25.6s
          searching 63.6% ...          consumed time: 30.1s
          searching 72.7% ...          consumed time: 34.2s
          searching 81.8% ...          consumed time: 38.6s
          searching 90.9% ...          consumed time: 43.2s
          searching 100.0% ...          consumed time: 48.0s

          step:                              1
          zjr_fl's decision:                 x = 7 y = 7
          decision simulations:              5824
          total simulations:                 9999
          max search depth:                  20
          zjr_fl's eval winning rate:        None (will be available after 10 steps)
          search time:                       48.0s
```

（3）增加探索性。第一，在扩展节点的expand()函数中，增加了Dirichlet噪声到根节点的先验概率，来实现额外的探索，使得尽可能搜索到更多的情况。第二，在选择动作时加入了温度系数，而不是不直接选择访问次数最多的动作，这使得在开始的时候动作更多样化，学习更多的策略，而不只是概率最高的策略，并逐渐降低温度系数，使得在结束的时候尽可能保证胜率。

（4）避免陷入局部最优。整个训练过程只使用一个神经网络。每更新一个网络后，都要将迄今最好的网络对比，如果新的网络胜率超过 55%，才会用来取代以前最好的版本，防止自我对弈时陷入局部最优。

（5）引入γ折扣率，令往后的状态所反馈回来的 reward 乘上这个 γ，使得当下的 reward 比未来反馈的 reward 更重要，以尽可能在较少的步骤里得到尽可能多的奖励。