项目代码解释

代码是用 Python 实现的，用的深度学习框架是 TensorFlow，代码总共分了 6 个模块，（五子棋类、界面类、纯 MCTS 类、真 MCTS 类、策略价值网络类、训练类），代码总长度大概 1000 行+。

chessboard 类

主要功能

主要是完成了五子棋的规则及下棋相关方面的东西，定义五子棋棋盘的大小，并且完成落子、判断胜负、提供当前棋盘状态等功能，输出胜负情况等。

全局变量

1. base[]:数组，base[i]记录的是 3^i，是为了加速下面对棋盘的哈希运算，（为了对棋盘进行哈希表示，我将棋盘表示成一个 3 进制数，0 代表棋盘上该点为空，1 代表棋盘上为黑子，2 代表白子）

chessboard 类

1. def __init__(self, length = 15, n_in_rows = 5, current_player = 1, display = False)
   初始化 chessboard 类
   - length：棋盘的大小为 length*length
   - n_in_rows：当前的是几子棋
   - player：当前下棋的是哪个
   - board：numpy 数组，大小为 length*length，记录当前棋盘的状态，0 代表空，1 代表黑子，2 代表白子
   - availabels：set 类型，是当前没有被落子点的集合
   - direction_x：五子棋的棋盘是 8 联通的图，即上下左右、左上、左下、右上、右下，这个保存的是可供选择的 x 坐标变化量
   - direction_y：铜 direction_x，记录可供选择的 y 坐标变化量，和 direction_x 两两对应
   - hash_baord：在全局变量 base[]介绍中提到过，是当前棋盘状态对应的哈希值，范围为[0,3^(length*length)]
   - excuted_step：当前下棋的步数

- history1: 列表, history[i]记录的下完第 i 步棋时, 是一个二值、大小为(length, length)的 numpy 数组，记录的是棋手 1 的历史棋盘状态，0 代表空或者棋手 2 的落子，1 代表棋手 1 的落子
- history2：同 history1，记录的是棋手 2 的状态

2. def change_player(self)
   改变当前的棋手，如果原来 player 是 1，改成 2；如果原来 player 是 2，改成 1

3. def position_to_index(self, position)
   将棋盘上的一个坐标（x，y）映射成一个数字
   - position：一个坐标（x,y），代表棋盘上的一个位置

4. def index_to_position(self, index)
   将一个数字映射成棋盘上的一个坐标（x，y）
   - index：棋盘上坐标（x，y）对应的数字

5. def point_in_chessboard(self, x, y)
   检查坐标(x, y)是否是一个合法的坐标

6. def check_point(self, position)
   检查以坐标 position 为中心的点，是否构成大于等于规定的 n_in_rows 的同色合法棋，比如是否连成五子，是返回 True，否返回 False

7. def get_state(self, num_history)
   返回当前棋局的状态，供神经网络使用
   - num_hisotry：常数，定义返回往前几步的棋局状态

8. def excute_move(self, position)
   执行落子操作，返回棋局是否结束，如果结束返回获胜者，0 代表平局，1 代表先手获胜，2 代表后手获胜

9. def end_winner(self)
   不执行落子操作，单纯判断棋局状态，返回值同 excute_move

interface 类

主要功能

实现对棋盘的展示，即界面显示，主要是用了库 pygame

全局函数

1. def evaluate(gameboard)
   通过随机落子，判断当前局面的好坏，这是一个十分粗浅的判断方法
   - gameboard：是一个棋盘类实体

Interface 类

1. def start_play(self, player1, player2, start_player)
   主要显示两者对弈过程中，在项目中主要用来观测真 MCTS 和纯 MCTS 对弈过程，返回胜负状态，0 代表平局，1 代表先手胜，2 代表后手胜
   - player1：下棋者 1，是一个决策类，比如真 MCTS
   - player2：下棋者 2，是一个决策类，比如纯 MCTS
   - start_player：先手是 player1 还是 player2

2. def start_self_play(self, player)
   和 start_play 类似，不同的是这个函数是用于真 MCTS 在自我对弈时候的监测，并且返回落子数据，用于神经网络的训练

3. def run1(self)
   用于显示对弈，里面可以自己选择对弈双方的能力，比如模拟次数

4. def run2(self)
   用于人机对弈，用鼠标点击要落子位置可以于内置 AI 定义，AI 就是前面的决策类真 MCTS 和纯 MCTS

pure_mcts 类

主要功能

实现了一个基本的 MCTS 决策类，通过调大模拟次数，能在 8x8 的五子棋棋盘和 6x6 的四子棋棋盘上实现相当高的智能表现，主要是用来和真 MCTS 进行比较，测试真 MCTS 的智能水平。

EDGE 类

1. def __init__(self, prob)
   初始化 EDGE 类

- prob：这条边的访问概率，用于选择的时候使用，是边的一个性质
- visit_count：这条边的实际访问次数，这是在 mcts 的模拟过程中，这条边的访问次数，这说明 mcts 看好这条边
- Q：这条边访问的平均价值 = W / visit_count
- W：这条边访问的总价值

NODE 类

1. def __init__(self)
   初始化 NODE 类
   - child：一个映射，代表从当前 NODE 执行某个动作后会到达哪个 NODE
   - edge：一个映射，代表从当前 NODE 执行某个动作后会走那条边
   - visit_count：常数，记录当前节点被访问的次数
   - cpuct：常数，表示了当前选择节点公式中 cpuct 的值

2. def select(self)
   节点选择函数，根据公式，下一步应该选择哪个节点进行模拟

pure_mcts 类

1. def __init__(self, chess, simulation_times)
   初始化 pure_mcts 类
   - chess：棋盘类
   - states：字典，记录当前已经出现过的棋盘状态对应的 NODE 节点，方便进行映射
   - root：当前 pure_mcts 棋盘的根节点状态
   - simulation_times：进行一次模拟，搜索到叶子节点的次数

2. def simulation(self)
   进行一次搜索到叶子节点的模拟，并且扩展该叶子节点，更新路径上蒙特卡洛树的参数

3. def get_action(self)
   获得蒙特卡洛树得出的最优下一步

4. def update_action(self, position)
   当外面棋盘的状态发生变化时，需要更新蒙特卡洛树，以对之前已经进行过的搜索结果进行利用

real_mcts 类

主要功能

实现了一个神经网络版本的 MCTS 决策类，这也是我们用来进行自我对弈和探索-利用的真 MCTS 框架。

EDGE 类

1. def __init__(self, prob)
   初始化 EDGE 类
   - prob：这条边的访问概率，用于选择的时候使用，是边的一个性质
   - visit_count：这条边的实际访问次数，这是在 mcts 的模拟过程中，这条边的访问次数，这说明 mcts 看好这条边
   - Q：这条边访问的平均价值 = W / visit_count
   - W：这条边访问的总价值

NODE 类

1. def __init__(self)
   初始化 NODE 类
   - child：一个映射，代表从当前 NODE 执行某个动作后会到达哪个 NODE
   - edge：一个映射，代表从当前 NODE 执行某个动作后会走那条边
   - visit_count：常数，记录当前节点被访问的次数
   - cpuct：常数，表示了当前选择节点公式中 cpuct 的值

2. def select(self)
   节点选择函数，根据公式，下一步应该选择哪个节点进行模拟

real_mcts 类

1. def __init__(self, chess, simulation_times)
   初始化 pure_mcts 类
   - chess：棋盘类
   - policy：决策落子用的函数，现在就是神经网络
   - cpuct：常数，表示了当前选择节点公式中 cpuct 的值
   - simulation_times：进行一次模拟，搜索到叶子节点的次数

- temperature：AlphaGo 论文里面提到的在 MCTS 选择最终节点时用的公式的参数，控制对访问次数的利用情况
- num_history：对 chessboard 函数中 get_states 的一个控制，表示希望用之前多少步的状态来作为神经网络的输入
- is_selfplay：表示当前的 real_mcts 实体是否是用来
- states：字典，记录当前已经出现过的棋盘状态对应的 NODE 节点，方便进行映射
- root：当前 pure_mcts 棋盘的根节点状态
- random_steps：会被加上 dirichlet 噪声的步数

2. def simulation(self)
   进行一次搜索到叶子节点的模拟，并且扩展该叶子节点，更新路径上蒙特卡洛树的参数

3. trans_prob(self, prob, temperature)
   将数组进行映射，以得到最终选择的概率
   - prob：传入的初始概率，在文中的实现是节点的访问次数
   - temperature：温度参数，用于对访问次数的转换

4. def get_action(self)
   获得蒙特卡洛树得出的最优下一步

5. def update_action(self, position)
   当外面棋盘的状态发生变化时，需要更新蒙特卡洛树，以对之前已经进行过的搜索结果进行利用

PolicyValueNet 类

主要功能

用了 tensorflow，定义了策略价值网络的架构和训练方式，是整个算法最终希望能学习到的地方。

PolicyValueNet 类

1. def __init__(self, board_length, num_history, model_file = None)
   - board_length：棋盘宽度
   - num_history：神经网络接受的是包含几步的状态的输入
   - model_file：初始的 model，如果有的话，可以让神经网络在之前训练过的网络上继续训练，也可以让神经网络利用已经训练好的网络进行决策
   - input_states：神经网络的输入状态，也就是 chessboard 函数中的 get_state 函数返回出来的状态

- conv1：一层 32 个 3x3 filter 的卷积层，使用 ReLu 激活函数
- conv2：一层 64 个 3x3 filter 的卷积层，使用 ReLu 激活函数
- conv3：一层 128 个 3x3 filter 的卷积层，使用 ReLu 激活函数
- action_conv：一层 4 个 1x1 filter 的卷积层进行降维
- action_conv_flat：被摊成向量的 action_conv
- action_fc：一个 length*length 的全连接层，输出的就是落子的概率
- evaluation_conv：一层 2 个 1x1 filter 的卷积层进行降维
- evaluation_conv_flat：被摊成向量的 evaluation_conv
- evaluation_fc1：一层 64 个神经元的全连接层
- evaluation_fc2：一层 1 个神经元的全连接层，接一个 tanh 激活函数，作为对局面的价值判断
- labels：输入数据中的价值，也就是最后的结果 z
- value_loss：损失函数中价值部分
- mcts_prob：数据中，蒙特卡洛树给出的概率
- policy_loss：损失函数中策略部分
- l2_penalty_beta：对参数的惩罚项参数
- loss：损失函数
- optimizer：参数更新方式
- entropy：落子地点的熵，只是为了监测落子情况

2. def policy_value(self, states)
   对输入的 states 输出策略和价值，这就是策略价值网络的功能

3. def train_step(self, state_batch, mcts_probs, winner_batch, lr)
   更新神经网络的参数

4. def save_model(self, path)
   保存当前模型

5. def restore_model(self, model_path)
   还原训练过的模型

Train_Pipeline 类

这里主要是对整个模型进行训练的场所，基本上所有超参数都要在这里进行修改

全局变量

1. def start_play(player1, player2, start_player)
   同 interface 类中的同名函数，区别就是这里的不能进行可视化，这是为了供在可视化化和训练速度之间调节（可视化会在一定程度上影响训练速度，虽然影响很小）

2. def start_self_play(player)
   同上，对应 interface 类中的同名函数

TrainPipeline 类

这就是训练神经网络的主体类

- def __init__(self, init_model=None)
  初始化 TrainPipeline 类，也是很多超参数调节的地方
  - board_length：棋盘大小
  - n_in_row：定义了几子连线才算赢
  - num_history：定义了传给神经网络的状态包含了过去几手的情况
  - chess：一个 chessboard 类的实体
  - learn_rate：神经网络学习速率，实际上后来我直接用 Adam 更新参数了
  - lr_multiplier：学习率衰减程度
  - temperature：MCTS 节点选择时候公式中的参数之一
  - cpuct：MCTS 模拟选择节点的时候公式参数之一
  - buffer_size：双端队列的最大长度
  - batch_size：每次从双端队列中抽取的数据量，用于神经网络的参数更新
  - data_buffer：用于存储训练数据的双端队列
  - play_batch_size：一次自我对弈进行的盘数
  - epochs：一次神经网络训练，神经网络参数更新的轮数
  - kl_targ：kl divergence 变化量的限制
  - check_freq：进行多少次自我对弈后验证一下当前模型的好坏
  - best_win_ratio：历史最优模型对弈当前纯 MCTS 的胜率
  - game_batch_num：自我对弈盘数上限
  - loss_dict：保存了历史 loss_hold 盘的 loss 的一个字典
  - loss_hold：保存历史 loss 的数量，这是为了看 loss 相对于之前有没有在继续变小
  - real_mcts_simulation_times：真 MCTS 选择一步的模拟次数
  - pure_mcts_simulation_times：纯 MCTS 选择一部的模拟次数

- def get_equi_data(self, play_data)
  扩展训练数据

- def collect_selfplay_data(self, n_games = 1)
  自我对弈并且收集数据

- def policy_update(self)
  对神经网络参数进行更新，并且监测训练情况

- def policy_evaluate(self, n_games=10)
  将当前模型和纯 MCTS 进行比较

- def run(self)
  总的一些训练过程