作业5报告

xxx

(南京大学 xxx, 南京 210093)

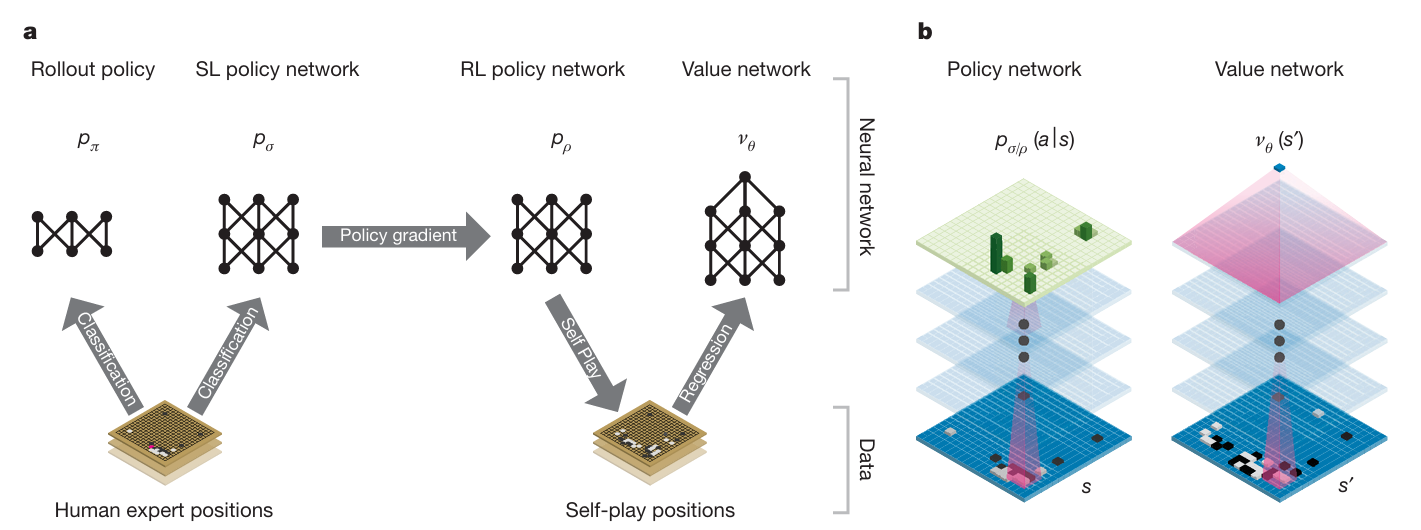
# 对AlphaGo原理的一点心得体会

**简介**

AlphaGo 是一种使用深度神经网络和树搜索技术的人工智能系统，通过阅读论文 *"Mastering the Game of Go with Deep Neural Networks and Tree Search"*和*"Mastering the Game of Go without Human Knowledge"*，我们可以了解到相比一般的棋类游戏如Othello，围棋的组合方式和可能的棋局变化极为庞大，远远超出了普通计算机能够处理的范围。而AlphaGo的成功，正是基于它使用了一些非常聪明的策略，使得它能在围棋这项游戏中能够在有限时间内给出决策，并进而击败人类顶级选手。

### 1.1. ****神经网络的作用：****

AlphaGo用了两个主要的神经网络：



* **策略网络(**rollout policy nets**)**的作用是决定每一步棋应该下在哪里。它会给出每个可能的棋步一个分数，分数越高的地方，表示下棋的机会越好。这个网络通常是通过大量的人类历史围棋棋谱进行训练，从中学习哪些棋步比较好。但是本次mini\_go项目中使用的rollout policy nets是浅层RL与Uniform Random对弈得到的策略网络，因此在实际测试过程中会出现对战测试结果不稳定的现象。
* **价值网络(**policy nets**)**的作用是评估当前棋局的局势好坏。它的目标是告诉AlphaGo当前棋局到底是谁占优势。这个网络的训练则是通过对围棋局面的结果（胜或负）进行学习。本次mini\_go中GoEnv.py中的reward函数价值判断也很简单粗暴, 对局胜利就返回1，否则返回0。这些数据会由深层RL与Uniform Random对弈得到的策略网络来处理并预测胜者。

### 1.2. 训练的过程：

根据*"Mastering the Game of Go with Deep Neural Networks and Tree Search"*，AlphaGo的训练过程分为两个阶段。

* 在**监督学习**阶段(见Supervised learning of policy networks部分)，AlphaGo从大量的人类对弈棋谱中学习，模仿人类玩家的棋风。通过这些数据，它训练出了一个非常强大的策略网络。
* 但是仅仅模仿人类是不够的，AlphaGo还需要提升自己的棋力。于是就进入了**自我对弈**阶段(见Reinforcement learning of policy networks部分)，通过和自己对弈，来进一步学习和改进策略。在每一场自我对弈中，AlphaGo都会对自己的表现进行评估，自己指导自己变得更强。它没有仅仅依赖于人类的知识，而是通过自己的对弈经验不断进化。

### 1.3. ****树搜索和蒙特卡罗树搜索（MCTS）：****

虽然神经网络非常强大，但围棋的复杂度依然让它无法通过单纯的神经网络进行完美的决策。为了弥补这个不足，AlphaGo引入了**蒙特卡罗树搜索**的算法。

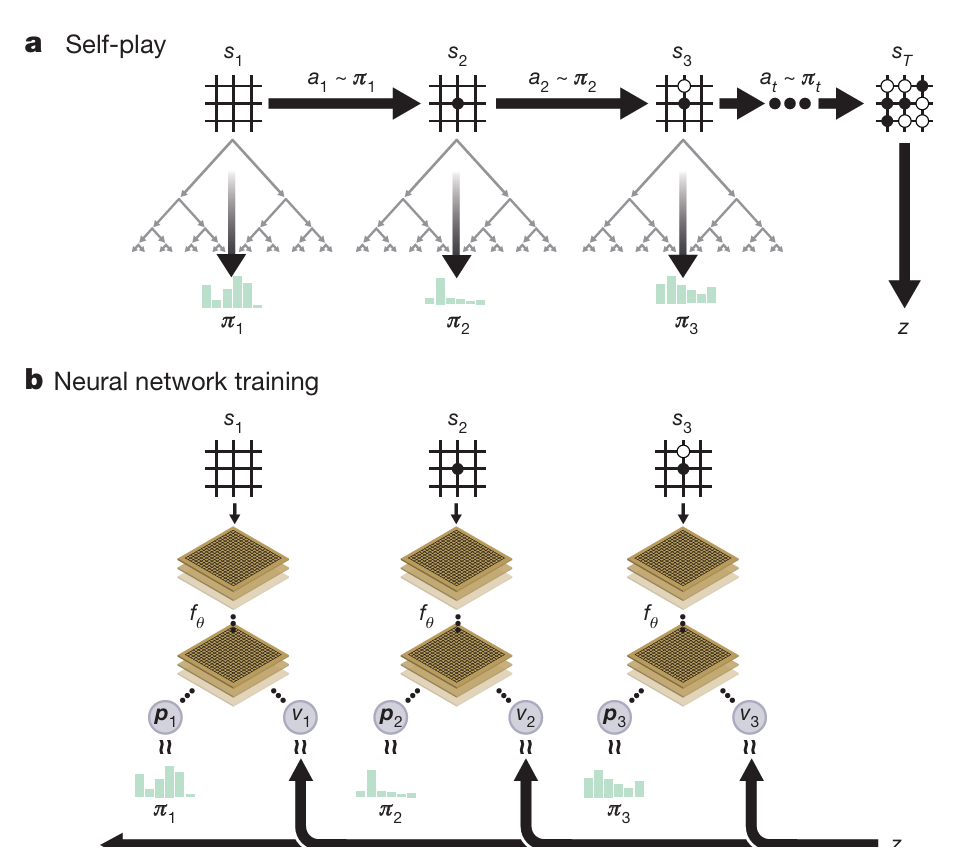
MCTS的基本思想是：在决策过程中，AlphaGo会先模拟很多种可能的棋局，然后通过多次的模拟对比，选择最优的棋步。它就像是在走棋之前先“预演”了好几遍，最终决定哪条路最有可能带来胜利。

具体来说，AlphaGo会使MCTS的策略网络预测每一步可能的棋步，并通过价值网络评估这些棋步的效果。然后，MCTS会在这些预测的基础上，模拟多次对弈，最终选出最有可能带来胜利的棋步。

### 1.4. ****AlphaGo Zero的突破：****

AlphaGo Zero是AlphaGo的升级版，它的一个重要创新是，**完全不依赖人类棋谱**。也就是说，AlphaGo Zero不再从人类的对弈中学习，而是从零开始，通过自我对弈的方式自己学习规则，自己找到最强的策略。这种方式让它比AlphaGo更加强大。

根据*"Mastering the Game of Go without Human Knowledge"*，AlphaGo Zero与AlphaGo采用了完全不同的方式训练学习：即**它完全只使用强化学习训练；只使用棋盘上黑白棋作为输入特征；只使用单一的神经网络。不使用分开的策略网络和值网络；只使用简单的树搜索，且树搜索只依赖于上述单一的神经网络，同时进行棋面局势评估和走棋动作选择。抛弃了蒙特卡罗中的Rollout Policy，只将蒙特卡罗树搜索树纳入训练阶段。**通过不断地和自己下棋，AlphaGo Zero逐渐找到了围棋的最优解，并最终超过之前的AlphaGo模型，由于不是本次实验的内容，所以在有余力的情况下我会酌情实现。



# 围棋环境中实现MCTS方法

在阅读MiniGo代码时，我常常因为对着标准的MCTS板子写代码时因为忘记了有什么环境函数导致卡壳。痛定思痛之后，我决定重新开始，边写代码边在实验报告中记录需要或者可能需要调用的环境API接口，具体见下：

# **2.1 环境（主要是GoEnv.py，Go.py中主要是围棋规则）中的数据交互方式**

# 2.1.1 TimeStep 类:

TimeStep 类用于表示围棋环境中的一个时间步，包含以下属性：

* observations：包含每个玩家观察值的字典列表。
* rewards：包含每个玩家奖励的列表。
* discounts：包含每个玩家折扣因子的列表。
* step\_type：表示时间步类型的枚举值（FIRST、MID 或 LAST）。

以及以下方法：

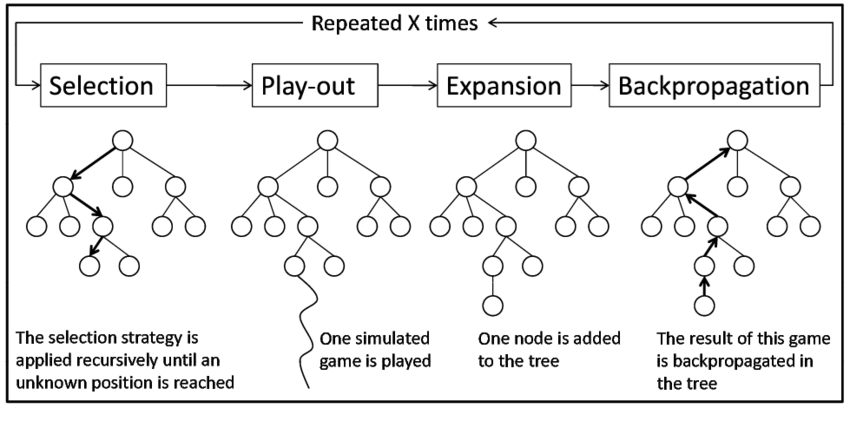
* first()：判断时间步是否为序列的第一个时间步。
* mid()：判断时间步是否为序列的中间时间步。
* last()：判断时间步是否为序列的最后一个时间步。
* current\_player()：返回当前玩家的 ID。

# 2.1.2 Go 类:

Go 类表示围棋环境，包含以下核心方法：

* step(self, action)：执行一个动作，并返回下一个时间步的 TimeStep 对象。
* reset(self)：重置围棋环境，并返回初始时间步的 TimeStep 对象。
* get\_all\_legal\_moves(self)：返回所有合法的动作。
* get\_current\_board(self)：返回当前棋盘的状态。

根据上述的接口，参考algorithm/dqn.py和dqn\_vs\_random\_demo.py，我们可以实现基本的mcts方法（algorithm/mcts.py）。



我实现的MCTS分为选择、拓展、模拟和回传4个部分（如上图所示），核心功能见下：

（1） 选择阶段的目标是从根节点开始，根据 UCB（上置信界）算法选择一个最优的子节点，并逐步深入到树的叶子节点， 可见best\_child函数。

（2） 拓展阶段是将当前节点下所有合法的动作作为子节点展开。每个子节点代表从当前状态执行某个动作后的结果, 可见expand函数。

（3）模拟阶段是从当前节点开始，通过某种策略（如随机选择动作或使用回合策略网络）进行模拟，直到达到游戏的终局。模拟结束后返回奖励。

1. **def** rollout(self, rollout\_policy\_net=None):
2. current\_time\_step = copy.deepcopy(self.time\_step)
3. current\_env = copy.deepcopy(self.env)
4. **while** **not** current\_time\_step.last():
5. possible\_moves = current\_time\_step.observations['legal\_actions'][self.player\_id]
6. **if** rollout\_policy\_net:
7. action = self.policy\_rollout(rollout\_policy\_net, current\_time\_step)
8. **else**:  # 没有给定的rollout\_policy\_net，主要用于前期测试
9. action = self.rollout\_policy(possible\_moves)
10. current\_time\_step = current\_env.step(action)
11. **return** current\_time\_step.rewards[self.player\_id]

（4）回传阶段的目标是将模拟阶段获得的奖励反馈到树的每一个节点上，更新节点的访问次数和累计价值。可见backpropagate函数。同时损失函数我只是选取了较为简单的均方差函数。

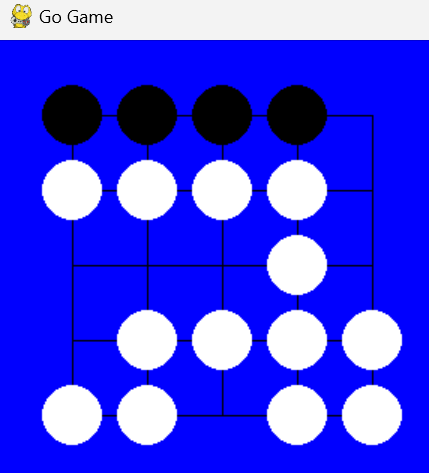
在mcts\_vs\_random\_demo.py中我们在创建两个独立的计算图之后，使用两个独立的tf.Session()创建对话并且调用已经训练好的DQN模型(谁说DQN不是RL算法):

1. rollout\_hidden\_layers\_sizes = [128, 128]
2. rollout\_hidden\_layers\_sizes\_kwargs = {
3. "replay\_buffer\_capacity": int(5e4),
4. "epsilon\_decay\_duration": int(0.6 \* 10000),
5. "epsilon\_start": 0.8,
6. "epsilon\_end": 0.001,
7. "learning\_rate": 1e-3,
8. "learn\_every": 128,
9. "batch\_size": 128,
10. "max\_global\_gradient\_norm": 10,
11. }
13. # ...
15. # Create a separate graph for each network
16. rollout\_graph = tf.Graph()
17. value\_graph = tf.Graph()
19. # Load fast-rollout network
20. with tf.Session(graph=rollout\_graph) as sess:
21. with rollout\_graph.as\_default():
22. rollout\_agent = DQN(sess, 0, env.state\_size, env.action\_size, rollout\_hidden\_layers\_sizes, \*\*rollout\_hidden\_layers\_sizes\_kwargs)
23. sess.run(tf.global\_variables\_initializer())
24. rollout\_agent.restore("saved\_model/10000\_fast")
26. # Load value network
27. with tf.Session(graph=value\_graph) as sess:
28. # ...

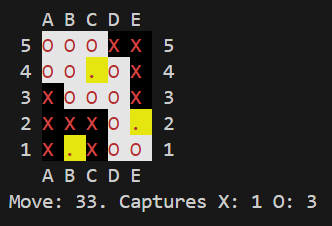
最后我们保存模型：

1. **def** save(self, checkpoint\_root, checkpoint\_name):
2. save\_prefix = os.path.join(checkpoint\_root, checkpoint\_name)
3. **if** self.policy\_net:
4. self.policy\_net.save(checkpoint\_root, checkpoint\_name + '\_policy\_net')
5. **if** self.rollout\_policy\_net:
6. self.rollout\_policy\_net.save(checkpoint\_root, checkpoint\_name + '\_rollout\_policy\_net')
7. with open(save\_prefix + '\_version.txt', 'w') as f:
8. f.write(str(self.version))

顺手做了一个pygame可视化……主要是print(env.get\_current\_board())在训练的时候打印过快颜色和符号过于难以区分导致训练效果其实并不太清晰，通过mcts\_vs\_random\_demo.py中的flag布尔变量visualize控制开关，添加之后的贴图如下：



print(env.get\_current\_board())的效果如下(一言难尽啊)：



# 实现对手池方法

为了实现对手池，我们需要在之前创建的algorimths/mcts.py和mcts\_vs\_random\_demo.py文件中添加对手池的类及其调用方法。

具体代码见algorimths/opponent.py和mcts.py和mcts\_vs\_random\_demo.py文件，根据我的理解和网络搜索结果，对手池应该实现以下功能：维护一个固定大小的对手池；支持动态添加新对手并随机移除旧对手；提供随机采样对手的机制；支持大部分对手池的持久化存储和加载；以及通过版本号追踪对手的演化历史。我们给MCTS类增加一个version属性，通过version来区别历史迭代版本。同时在OpponentPool中实现add\_opponent, sample\_opponent, save\_pool, load\_pool的方法即可。

在mcts\_vs\_random\_demo中我们使用已经定义好的接口接入OpponentPool:

1. # ...
2. with tf.Session() as sess:
3. # 初始化对手池
4. opponent\_pool = OpponentPool(sess, env, FLAGS.pool\_size)
6. # 加载已有的对手池（如果存在）
7. **if** os.path.exists(FLAGS.pool\_directory):
8. opponent\_pool.load\_pool(FLAGS.pool\_directory)
10. # ...
11. # 训练循环(下面的代码应该在sess之外的,多缩进了请见谅)
12. **for** ep **in** range(FLAGS.num\_train\_episodes):
13. # 决定是使用对手池中的对手还是随机对手
14. **if** random.random() < FLAGS.pool\_sampling\_rate **and** opponent\_pool.pool:
15. opponent, version = opponent\_pool.sample\_opponent()
16. agents = [mcts\_agent, opponent]
17. # ...
18. **if** (ep + 1) % FLAGS.save\_every == 0:
19. # 保存当前模型到对手池
20. save\_path = os.path.join(FLAGS.pool\_directory,
21. f'opponent\_{ep+1}')
22. rollout\_agent.save(
23. checkpoint\_root=FLAGS.pool\_directory,
24. checkpoint\_name=f'opponent\_{ep+1}'
25. )
26. opponent\_pool.add\_opponent(save\_path, ep+1)
28. # 保存整个对手池
29. opponent\_pool.save\_pool(FLAGS.pool\_directory)

# 含有对手池的MCTS vs 均匀随机的MCTS

具体代码见mcts\_vs\_mcts.py，训练后的模型保存在save\_model\_mcts中，相比mcts\_vs\_random\_demo.py没有什么变化，通过构建自动化脚本autorun.py(懒得贴代码了)完成100，1000次对局并且自动统计结果(由于环境配置原因可能无法在别的环境下复现)，直接上结果分析（roll\_out网络深度始终为128）：

#### **100局对战结果**

| **价值网络深度** | **训练后的AI胜率（%）** | **MCTS AI胜率（%）** |
| --- | --- | --- |
| **128** | 54 | 46 |
| **256** | 75 | 25 |
| **512** | 55 | 45 |

#### **1000局对战结果**

| **价值网络深度** | **训练后的AI胜率（%）** | **MCTS AI胜率（%）** |
| --- | --- | --- |
| **128** | 66 | 34 |
| **256** | 82 | 18 |
| **512** | 33 | 67 |

可以看到使用了对手池之后，在局数为1000局的时候使用对手池的胜率比较高，但是在对局网络层数增加到512时胜率反而下降，这说明含有对手池的MCTS出现了较为严重的过拟合现象。价值网络层数设置成256最为合适。在不考虑数据较少的情况以及网络过拟合的情况，我们发现整体而言使用了RL预制网络，并且经过对手池数次迭代之后的MCTS搜索在性能上远远优于随机搜索的MCTS。

# 尝试调整超参数

为了优化 Mini AlphaGo ，我们可以像这样调整超参数：

（1）首先我们可以调整 MCTS 的 rollout 次数, 增加或减少 MCTS 的模拟次数（n\_simulations）, 调整 MCTS 的最大深度（max\_depth）。

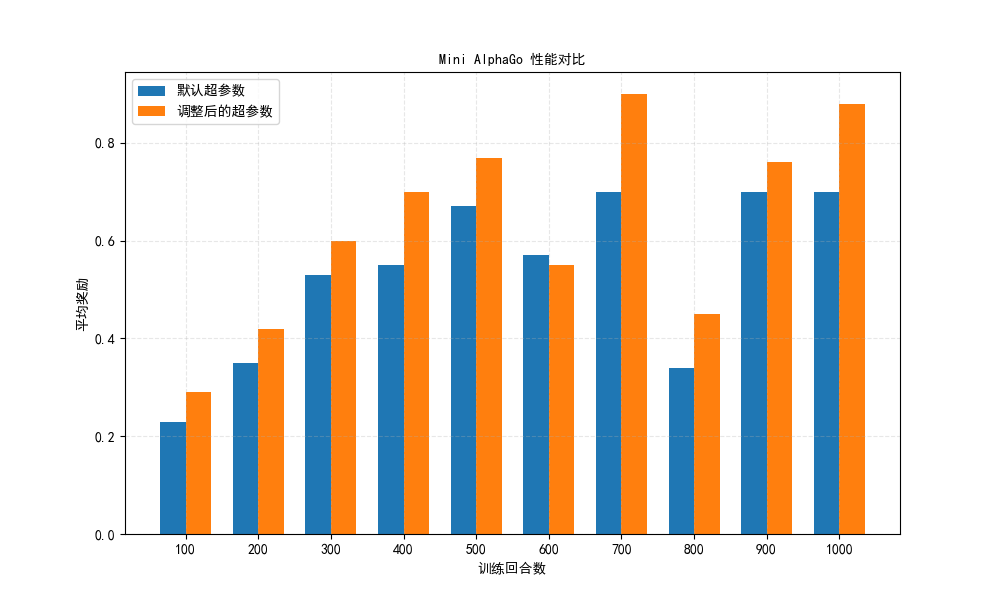
1. flags.DEFINE\_integer("n\_simulations", 100, "Number of simulations for MCTS.")
2. flags.DEFINE\_integer("max\_depth", 20, "Maximum depth for MCTS.")

（2）其次我们可以调整DQN强化学习算法的超参数， 如调整 DQN 的学习率（

epsilon 参数，replay buffer 容量和batch size（batch\_size）。

1. rollout\_hidden\_layers\_sizes\_kwargs = {
2. "replay\_buffer\_capacity": int(1e5),  # 增加 replay buffer 容量
3. "epsilon\_decay\_duration": int(0.8 \* FLAGS.num\_train\_episodes),  # 调整 epsilon 衰减持续时间
4. "epsilon\_start": 1.0,  # 增加 epsilon 起始值
5. "epsilon\_end": 0.01,  # 减少 epsilon 结束值
6. "learning\_rate": 5e-4,  # 调整学习率
7. "learn\_every": 64,  # 减少学习频率
8. "batch\_size": 256,  # 增加 batch size
9. "max\_global\_gradient\_norm": 10,
10. }

两者最后得到的平均reward结果绘制如下图所示，分析之后我得到如下推断：增加模型的复杂度往往会导致不同程度的过拟合，但是模型的局面推断能力能够得到一定提升；而增大搜索能力则能够一定程度上降低这种现象，但是也变相增长了训练总时间。这两方面在Mini AlphaGo的总体性能提升上需要有所权衡取舍。



References:

[1] [Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature **529**, 484–489 (2016).](https://www.nature.com/articles/nature16961)

[2] [Silver, D., Schrittwieser, J., Simonyan, K. et al. Mastering the game of Go without human knowledge. Nature 550, 354–359 (2017).](https://www.nature.com/articles/nature24270)

[3] [Schadd, Maarten. Selective search in games of different complexity.](https://www.researchgate.net/publication/254849243_Selective_search_in_games_of_different_complexity)

[4] [面向初学者的蒙特卡洛树搜索MCTS详解及其实现\_mcts算法-CSDN博客](https://blog.csdn.net/caozixuan98724/article/details/103213795)

[5] [AlphaGo - Google DeepMind](https://deepmind.google/research/breakthroughs/alphago/)

[6] [联盟训练在游戏AI多样化的探索\_联盟学习 模型多样性csdn-CSDN博客](https://blog.csdn.net/wangchewen/article/details/120904962)]