Mini AlphaGo 实验报告

宋磊（181220049、974534426@qq.com）

(南京大学 人工智能学院)

1 文件介绍

1.1 Agent定义

按照原本的文件结构，实现时将要求的 agent 的定义都放在了 agent/agent.py 中。包括使用 random rollout 方法的 Random_Rollout_MCTS_Agent 和使用值神经网络和策略神经网络的 Net_MCTS_Agent。

1.2 有关算法

需要的算法放在了 algorithm 中。

Dqn_cpy.py 为对原本 dqn 修改后的文件，增加了获取值_get_value()和策略_get_policy()的方法。

Random_rollout_mcts.py 为 random_rollout 的 MCTS 的实现，没有用到神经网络。

Net_mcts.py 为带网络的 MCTS 的实现，加载了黑棋和白棋方的策略网络和值网络。

MCTS 的实现过程参考了在 guthub 上基本的 MCTS 节点的定义和基本结构[1]，并进行了大量修改，适应作业中提供的环境及作业要求。

1.3 主函数

具体的对战的函数在最外面的 mini_go 文件夹中。

Net_mcts_vs_random.py 为带网络的 mcts 与 random agent 对战的实现，具体在后面介绍。

Net_mcts_vs_net_mcts.py 为对手池的实现，用最新的网络的 mcts 和历史上随机一个网络的 mcts 进行对战，进行自我改进。

Net_mcts_vs_random_rollout.py 为训练完成的 mcts 与 random rollout 的 mcts 的对战。

其他文件用于快速调试正确性(random_rollout_vs_random.py)，生成网络执白棋时的策略模型(random_vs_dqn.py)。

1.4 最终得到的模型

在 saved_model 文件夹中。

Dqn_vs_random 文件夹中为 dqn_vs_random_demo.py 生成的模型，random_vs_dqn 为 random_vs_dqn.py 生成的模型，这两个里面的模型是训练的基础，用于开始时的黑棋和白棋的策略网络的加载和值网络的加载。

Self_play 中为自我对战生成的模型，通过不断加载最好的模型和随机加载历史上一个模型进行训练，不断重复，最后得到训练好的模型，这个里面也是作业中要求的模型。

Net_mcts_vs_random 中为 net_mcts_vs_random.py 生成的模型。

2 MCTS 实现

在树节点 TreeNode 中定义了 MCTS 中的选择，扩展，更新的三个基本操作，分别对应 select()，expand()，

update_recuesive ()。将 rollout 的模拟过程放在了 MCTS 类的实现中，对应 rollout()函数。下面对 MCTS 的主要操作和一些重要步骤进行分析。

2.1 Select()实现

用于选取最优的节点，在每一次模拟时，都会调用 select()选择，直到到达一个当前的叶节点，之后进行扩展。

```
def select(self):
    return max(self._children.items(), key=lambda node: node[1].get_value())
```

2.2 Expand()实现

Expand()中，先通过环境获得当前状态，之后对叶节点增加所有合法的孩子，并且赋予相应的估值。

```
def expand(self, action_priors, time_step):
    # print(action_priors)
    actions = action_priors[0]
    priors = action_priors[1]
    cur_player = time_step.observations["current_player"]
    sensible_moves = time_step.observations["legal_actions"][cur_player]

    for action, prob in zip(actions, priors):
        if action not in self._children and action in sensible_moves and action < 25:
            self._children[action] = TreeNode(self, prob)
```

2.3 Rollout()实现

Rollout 时，先获取当前状态，然后传入策略函数，这里将 rollout 动作选择封装为函数，并且提供统一的 API，在基本的 MCTS 中，可以传入 random rollout 的函数，在用神经网络实现时，传入策略网络，之后选择最优的 action 之后不断执行，直到游戏结束，获得最终的游戏结束时的奖励值。

这里实现时遇到一个问题，使用策略网络时，可能会遇到这里陷入死循环的情况，所以用变量 i 控制循环的次数（即控制 rollout 的深度），到达一定次数后跳出，返回当前估值。

```
def rollout(self, time_step, env):
    env_cpy = copy.deepcopy(env)
    # use i to limit, I don't know why if I don't use i, the cycle will sometimes never end
    i = 0
    while not time_step.last() and i<100:
        # action_probs = self._rollout_fn(time_step)
        # print(action_probs)
        cur_player = time_step.observations["current_player"]
        info_state = time_step.observations["info_state"][cur_player]
        legal_actions = time_step.observations["legal_actions"][cur_player]

        action_probs = self._policy_fn[cur_player]._get_policy(info_state, legal_actions)
        # sensible_moves = time_step.observations["legal_actions"][cur_player]
        # print(sensible_moves)
        best_action = action_probs[0][np.argmax(action_probs[1])]
        # print(best_action)
        time_step = env_cpy.step(best_action)
        i += 1

    return time_step.rewards[0]
```

2.4 Update实现

Update_recuesive ()中，从叶节点开始向上更新，并且需要对 leaf_value 增加负号，因为上一层的为相对于这一层的节点的对手。估值应该是负的这一层的估值。这里只更新了 Q 值，u 值的更新在 get_value()函数

中，会在每次 select 时，对 u 值进行更新。

```
def update(self, leaf_value):
    self._n_visits += 1
    self._Q += 1.0*(leaf_value - self._Q)/self._n_visits

def update_recuesive(self, leaf_value):
    if self._parent:
        # the parent is another player, so add the subtract
        self._parent.update_recuesive(-leaf_value)
    self.update(leaf_value)
```

2.5 加载模型

加载的模型根据课程主页上所说，将黑白棋分开，所以加载了黑棋的策略网络模型和白棋的策略网络模型，以及一个值网络模型(本来考虑要对黑白棋分别加载值网络，但其实加载一个已经足够，例如对黑棋估值为 value，则对白棋取负值，即-value)

Tensorflow 加载多个模型时，遇到了一些问题，在网上查找相关资料后，发现因为 tensorflow 默认将模型都加载到默认图中，如果直接加载多个模型，会发生冲突，所以需要建立多个图，将多个模型加载到不同的图中，其中一个的实现如下，其他用同样的方法加载(见代码)：

```
# policy function when player_id = 0
graph_1 = tf.Graph()
self.sess_1 = tf.Session(graph=graph_1)
with graph_1.as_default():
    self._policy_fn.append(DQN(self.sess_1, 0, 25, 26, [128, 128], **kwargs))
    saver = tf.train.import_meta_graph(policy_fn[0] + '.meta', clear_devices=True)
    saver.restore(self.sess_1, policy_fn[0])
    self.sess_1.run(tf.global_variables_initializer())
```

2.6 对节点的估值

对节点估值时，也采用了论文中提到的使用 random rollout 和值函数同时估值并进行加权和的方法，加权比例为论文中建议的 lambda=0.5。(这里我还考虑了值网络求值另一种方法，不是简单对值网络得到的值进行求和，而是再加一个全连接层，让神经网络训练过程中自动学会求和各项的系数，但经过实践，其实直接求和已经有比较好的效果，所以最终实现的代码中，也是采用直接求和的方法)

```
# calculate the value
val_1 = self._value_fn._get_value(info_state, legal_actions)
# print(np.average(val_1), np.sum(val_1))
val_1 = np.sum(val_1)
val_2 = self.rollout(time_step, env_cpy)
leaf_value = (1-self.lambda_val)*val_1 + self.lambda_val*val_2

node.update_recuesive(leaf_value)
```

3 对手池的实现

对手池的实现在 net_mcts_vs_net_mcts.py 中，我选取了一种简单的方式，同时构建两个带网络的 mcts 的 agent，一个加载最好的模型(由于电脑性能的限制，我最多只训练了 100 轮，所以每次加载 100 轮时的网络)，另一个随机加载一个模型(我是每 20 轮保存一次，所以产生一个间隔为 20 的随机数，用于选择模型)，这样不断重复 run 这个程序，每次都会加载最好的模型和随机一个之前的模型进行对战，并且在训练过程中，会更新所有的模型，下一次调用时，都是调用最新的模型。最终达到和历史上的对手不断对战的效果，不断改进模型。

```
best_policy = ['saved_model/self_play/self_play_policy_fn_0_100',
               'saved_model/self_play/self_play_policy_fn_1_100']
best_value = 'saved_model/self_play/self_play_value_fn_100'

random_policy = ['saved_model/self_play/self_play_policy_fn_0_%d'%(random.randint(1, 5)*20),
               'saved_model/self_play/self_play_policy_fn_1_%d'%(random.randint(1, 5)*20)]
random_value = 'saved_model/self_play/self_play_value_fn_%d'%(random.randint(1, 5)*20)

agents = [agent.Net_MCTS_Agent(best_value, best_policy, n_playout=50),
    agent.Net_MCTS_Agent(random_value, random_policy, n_playout=50)]
```

4 性能测试以及超参数调整

先对 random rollout 的 mcts 和带网络的 mcts 性能进行测试，确保其性能优于 random agent，这样后面的 self play 训练和带网络的 mcts 与 random rollout 的对战性能测试才是有意义有价值的。

4.1 Random rollout vs random

```
0.59
Time elapsed: 1545.8695666790009
```

如图是经过 200 次 random rollout mcts 和 random 对战之后的结果，可以看出 mcts 的效果远好于 random agent，对战时每局有大约 80%的胜率。

4.2 Net mcts vs random

先加载 dqn 和 random 对战获得的基础模型(即 dqn 网络与 random agent 经过 10000 轮训练后得到的两个策略网络模型和一个值网络模型)，得到在未进行强化学习时的 net mcts 的效果。

```
0.87
[1, 1, 1, 1, 1, -1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, -1, 1, 1, 1
 1, -1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
Time elapsed: 4034.47483253479
```

如图是 200 次 net mcts 与 random 对战的结果(中间的-1 和 1 是对最终 ret 的打印，可以直接忽略)，可以看出，带网络的模型，即使只是加载未经过强化学习的模型，效果也远远好于 random agent，并且效果要好于只使用 random rollout 的 mcts agent，达到了大约 93%的胜率。但也可以看到，所需的时间也是大大增加，大约为 random rollout 的 2.7 倍。

可以看到，random rollout 和带网络的 mcts 的效果都是远优于 random agent 的，所以之后的 self play 的强化学习过程，以及带网络的 mcts 和 random rollout mcts 的对战结果才是有意义的。而不是两个完全没有效果的无用模型的对战。并且当前带网络的模型效果已经远好于 random rollout 的效果。

4.3 参数调整

训练过程的代码在 net_mcts_vs_net_mcts.py 中，即和历史上的随机一个对手对战，更新模型。

在训练完成后，对 rollout 的次数调整，每次 EVAL 次数为 100，得到的 reward 为 100 次对战的平均值，最终得到下面的表格。

Rollout 的次数	10	30	50
获得的平均 reward	-0.04	0.14	0.08

```
-0.04
[-1, 1, -1, 1, 1, 1, 1,
 -1, -1, -1, -1, 1, -1,
, -1, -1, -1, 1, 1, 1,
```

```
0.14
[-1, 1, 1, -1, -1, 1, -1, 1, -1,
, -1, 1, -1, -1, 1, 1, -1, -1, 1,
1, -1, 1, 1, 1, 1, 1, -1, 1]
Time elapsed: 466.47731280326843
```

```
0.08
[-1, 1, -1, 1, -1, 1, 1, 1, -1,
 1, 1, 1, -1, 1, -1, 1, -1, -1,
 1, 1, -1, 1, 1, 1, -1, -1, 1, -
Time elapsed: 745.6062693595886
```

上面的图分别为 10 次，30 次，50 次 rollout，可以看出，随着 rollout 的次数的增加，效果先变好再变差，但这其实是与我之前的想法违背的。我认为他应该是随着 rollout 的次数的增加，先增加，之后保持不变。因为开始时，少量的 rollout 次数使得叶节点的估值不准确，随着次数增加，估值逐渐准确，并且在达到一定次数后，估值已经接近真实的这个节点的值，所以保持不变，收敛到一个确定的值，即这个节点的真实值。

上面的情况发生可能是因为在 30 次 rollout 次数时，对这个节点估值已经碰巧比较准确，在之后的 rollout 中，因为 rollout 的次数还不够多，所以发生了估值曲线发生了波动，远离了真实值，导致 rollout 次数增多后，效果反而有一点下降。这也反映出 rollout 的次数还比较少，估值还没有完全收敛，所以可以考虑增多 rollout 的次数进行更多实验，但笔记本的性能受限，加载多个网络，并进行 50 次 rollout 时，已经需要过多时间，所以没有继续调整。

5 总结

在这次的作业中，可以明显感到有些吃力，对 MCTS 算法的不熟悉，虽然上课时已经对整个过程有了一定了解，但写代码时，却在各种方面遇到了困难，例如不知道 MCTS 中四个步骤应该怎么分配，哪些实现在叶节点中，哪些实现在 MCTS 中；对 tensorflow 的不熟悉，只是在很久之前略微学了一点 tensorflow，在用的时候遇到的很多 bug 都不知道是由于什么，例如在加载多个模型时，就遇到了很多问题，想要调用一些高级的方便的 api，也无从下手。

最后也感谢辛苦的助教，祝助教新年快乐。

References:

[1] GitHub: https://github.com/junxiaosong/AlphaZero_Gomoku/blob/master/mcts_alphaZero.py