简易 alpha-go 的实现

丁豪（181220010、181220010@smail.nju.edu.cn）

(南京大学 计算机科学与技术系，南京 210093)

摘 要: alpha-go 是 google deepmind 团队于 2016 年完成的人工智能围棋程序，其基本原理使用到了蒙特卡洛树搜索（MCTS）、深度强化学习（DQN）、以及一系列减少搜索深度广度的方法。在此我们实现一个在 5x5 大小的棋盘上能够以较高水平对弈的简易 alpha-go，使用 python 与 tensorflow 作为基本工具。

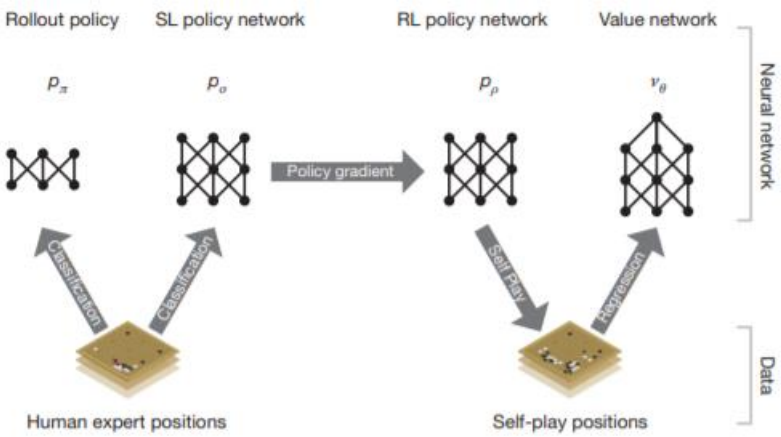
关键词: 蒙特卡洛树搜索、DQN、policy、rollout、value-network

中图法分类号: TP301 **文献标识码:** A

1 Alpha-go 原理初探

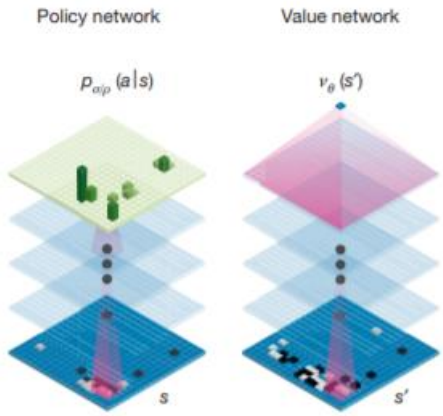
通过阅读论文，我们可以简单理解 alpha-go 的基本原理。大致分为四个版块，获得初始 rollout,policy，自我博弈获得更好的 policy network，根据已有数据训练 value network 用于非终止情况下的棋局评估，最终使用 MCTS 进行游戏。

首先，基于大量人类专家棋谱，使用监督学习(SL)训练得到 policy network，这一步的目的在于减少每一个状态下可行操作的数量，通过计算每一个可行操作的可能性排序来得到。此外，使用任意一种较为简单的能快速获得动作返回的算法，作为 rollout policy，用于进行之后的蒙特卡洛树搜索以及自我博弈学习。

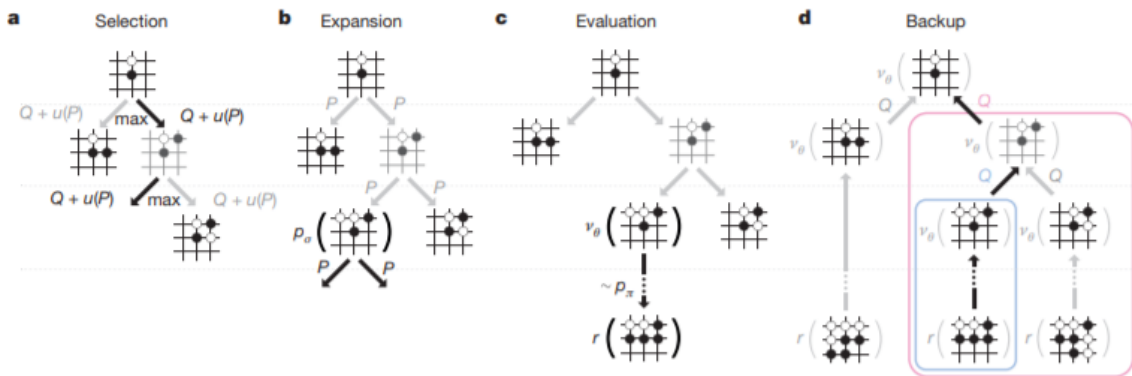


接着使用强化学习（RL）来提升 policy network 的强度。在自我对弈过程中，通过不断与自己的历史版本博弈，来最大化自身对于某历史版本的胜率，以此达到更好的效果。

在完成以上两个网络的训练后，额外训练一个 value network，用来基于当前棋局判断局势好坏情况。它的作用是降低蒙特卡洛树的搜索深度，这样在某些情况下可以不需要走到最终状态判断输赢，而是对当前状态直接用 value network 获得 value 值来进行判断为一次正尝试还是一次负尝试。



最终，通过蒙特卡洛树将上述三个网络加以结合使用。在每一步可行空间的选择上使用 policy-network 来缩减状态空间，在向后探索的过程中，以一定概率使用 rollout-network 来快速选择出一条走到终态的路径并计算胜负，或者以一定概率直接用 value-network 来近似判断胜负。如此不断展开蒙特卡洛树，与其它 agent 进行对弈。



2 实现 Alpha-go

最后一步使用蒙特卡洛树与随机模型对抗在文件 alpha_go/MCTS_with_trained_models 中实现，执行它将进行已有网络与随机模型的对抗实验。用于生成它所需的数个网络以及蒙特卡洛树的关键部分如下所述。

2.1 Rollout 与 policy 两个网络的初值

实现在/alpha_go/generate_policy_rollout.py。参考网页要求，我们分别使用两个不同参数的 DQN 模型来和随机 agent 进行对弈，其中较为复杂且胜率相应较高的一个作为我们的 policy-network 初值，而较为简单的一个作为今后的 rollout-policy。其分别存储在 saved_model/important/下的 policy_final 和 rollout_final 模型中。

```
max_len = 2000
rollout_hidden_layers_sizes = [32, 32]  # 用于rollout,蒙特卡洛树
policy_hidden_layers_sizes = [32, 64, 14]  # 用于之后的Q-learning
```

```
# evaluated the trained agent
agents[0].restore("saved_model/rollout10000")
agents[0].save(checkpoint_root='saved_model/important', checkpoint_name='rollout_final')
```

```
# evaluated the trained agent
agents[0].restore("./saved_model/policy_initial10000")
agents[0].save(checkpoint_root='./saved_model/important', checkpoint_name='policy_final')
```

2.2 用对手池方法训练policy-network

实现在/alpha_go/self_rival.py。在每一轮训练中首先将 self_与 rival 的初值设为上一次保存的模型，接着维持 rival 不变，将 self_优化对 rival 胜率训练，并把训练完成后的 self_保存，以备下一轮对手训练。

```
    self_.restore(rival_path)

agents = [self_, rival]
sess.run(tf.global_variables_initializer())
rival.restore(rival_path)
```

```
for ep in range(FLAGS.num_train_episodes):
    if (ep + 1) % FLAGS.eval_every == 0:
        prt_logs(global_ep + ep, agents, ret, begin)

    if (ep + 1) % FLAGS.save_every == 0:
        save_model(global_ep + ep, agents)

    time_step = env.reset()  # a go.Position object

    while not time_step.last():
        player_id = time_step.observations["current_player"]
        agent_output = agents[player_id].step(time_step, is_rival=(player_id == 0))
        action_list = agent_output.action
        print(player_id)
        print(action_list)
        time_step = env.step(action_list)


    for agent in agents:
        agent.step(time_step)
```

2.3 MCTS的实现

蒙特卡洛树的实现参考https://github.com/Rochester-NRT/RocAlphaGo中的 calss MCTS, 并对我们所需要的接口进行了重新封装得到基于蒙特卡洛树的 Agent 类 MCTSAgent,其实现代码在原 agent.py 后续。

蒙特卡洛 agent 接收一个 policy,一个 rollout，一个 value 三个网络用于蒙特卡洛过程，并会在 step 的过程中动态更新蒙特卡洛树节点保存的成功率与总次数等值。

```
# 蒙特卡洛树类
class TreeNode(object):...
class MCTS(object):...


# 采用蒙特卡洛方法的Agent
def random_policy_fn(time_step, player_id):...
def random_value_fn(time_step, player_id):...
class MCTSAgent(Agent):...
```

3 修改参数与性能分析

由于时间有限，暂时未进行参数调整。最终用 MCTS 于随机 Agent 对弈的结果如下。

```
α-go : random   [1, -1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [1, -1]
α-go : random   [1, -1]
α-go : random   [-1, 1]
α-go : random   [1, -1]
α-go : random   [-1, 1]
α-go : random   [1, -1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [1, -1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [1, -1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [1, -1]
α-go : random   [-1, 1]
α-go : random   [-1, 1]
α-go : random   [1, -1]
α-go : random   [-1, 1]
```

4 结束语

本次作业需要自主实现的部分量多而且难度较大，其中最有难度的是 tensorflow 的使用，几个算法的原理倒是其次的。此外，性能方面存在一些瓶颈，更改一次模型参数所需的训练时间过长，严重限制了我们深入分析参数与性能的关系。在进行 tensorflow 使用的过程中常出现一个错误是 windows fatal error：access violation, 在 stack overflow 以及 csdn 上提供的解决方法都是降级安装 tensorflow，本人并未尝试，故限制了部分代码的实现。

致谢 特别感谢李惟康同学与我的交流讨论，帮助我理解了许多框架代码，让我更好的理解了 alpha-go 的基本思想。特别感谢薛正海同学在此次作业中的领头作用，在他的指导下大家少走了许多弯路。