人工智能导论:作业五实验报告

薛正海(181220063、xuezh@smail.nju.edu.cn)

(南京大学 人工智能学院, 南京 210093)

1 基本代码组织

基本的训练代码在 MCTS/alg_vs_random.py 中.其可通过传入 use_dqn 命令行参数来决定使用 DQN 算法还是 A2C 算法,避免不必要的复制粘贴.框架代码的主要内容都在 main()函数中,较为臃肿,我将其重构为若干模块,重构后的 main 函数为:

```
def main(unused_argv):

    logging.info("Train on " + fmt_output_channels())

    env, info_state_size,num_actions, begin = init_env()
    cnn_parameters, hidden_layers_sizes, kwargs, ret, max_len = init_hyper_paras()

    with tf.Session() as sess:

        agents = init_agents(sess,info_state_size,num_actions, cnn_parameters, hidden_layers_sizes,**kwargs)

        train(agents,env,ret,max_len, begin)

        ret = evaluate(agents,env)

        stat(ret,begin)
```

初始化、训练、测试、统计等模块分工清晰,易于阅读、修改、扩展、重用.

2 实现 MCTS 方法

2.1 Policy Nets和Rollout Policy Nets的准备

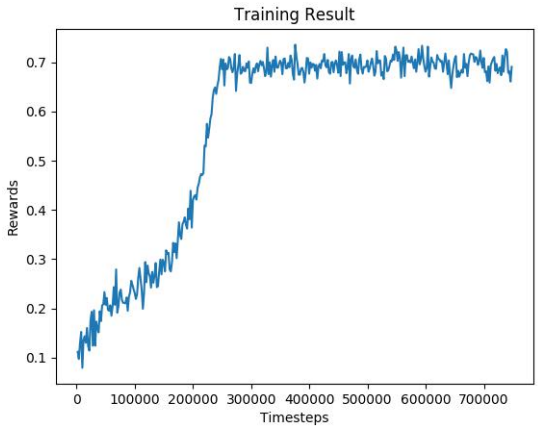
此模块的目的是使用 RL 算法和 Uniform Random 对手博弈,得到用纯强化学习训练的结果,对 MCTS 的采样网络和训练网络进行初始化.算法提供了基本的利用 MLP 网络的 DQN 算法,可以直接训练.训练无需对代码进行改动,过程较为顺利,使用较少的时间步和较简单的网络结构取得了略优于 RandomAgent 的性能.最终结果如下:

然而,AlphaGo 原文中使用的网络为卷积神经网络(CNN).其能捕捉一个方形区域内像素级别的特征,且具有平移不变性.这些特点与围棋的特性不谋而合,因此 Deepmind 使用 CNN 取得了不俗的成果.我试图在框架代码中加入 CNN 作为特征提取工具.修改后的代码在 algorithms/dqn_cnn.py 和 algorithms/policy_gradient.py.关键

代码如下:

```
self._q_network_cnn = snt.nets.ConvNet2D(output_channels = cnn_parameters[0],
                                         kernel_shapes = cnn_parameters[1],
                                         strides = cnn_parameters[2],
                                         paddings = cnn_parameters[3])

self._q_network_mlp = snt.nets.MLP(output_sizes=self._layer_sizes)
self._q_values = self._q_network_mlp(tf.layers.flatten(self._q_network_cnn((self._info_state_ph))))
```

其中 snt.nets.ConNet2D 是我通过阅读 sonnet 的文档 https://sonnet.readthedocs.io/en/latest/了解到的与原框架代码兼容的卷积神经网络的实现方法.cnn_parameters 通过 FLAGS 命令行参数传入,方便超参数调整.进过上述调整后就可以训练带 CNN 的策略模型了.训练过程详见第五章超参数调整部分.最优的训练结果如下:



2.2 MCTS方法的代码实现

基本 MCTS 的代码实现基于 https://github.com/Rochester-NRT/RocAlphaGo 中的 MCTS 实现,并作了多处改动.代码在 MCTS/mcts.py 中.其中改动之处有:1. 改变原代码中在环境中模拟的方式 state.step(action):因为 Go 环境的 state 没有提供 step 方法,执行 step 需结合环境返回的 env,所以我将上述代码改为 state = env.step(state,action),并传入相应的 env; 2. 加入 player_id 属性,指明当前玩家编号,符合策略网络等的输入要求,并在 env.step()时切换当前玩家; 3. 更新 rollout_limit, playout_depth 等参数,使其与环境设定相符.

为将基本 MCTS 的实现封装为可执行动作的 Agent 类,我在 MCTS/mcts_agent.py 中加入了实现相应功能的代码.具体代码如下:

```
class MCTSAgent():

    def __init__(self,policy_module, rollout_module, playout_depth = 10, n_playout = 100):

        if policy_module == None and rollout_module == None:

            self.policy_fn = self.rollout_policy_fn = random_policy_fn
            self.value_fn = random_value_fn

        else:

            self.value_fn = policy_module.value_fn
            self.policy_fn = policy_module.policy_fn
            self.rollout_policy_fn = rollout_module.policy_fn

        self.mcts = MCTS(value_fn = self.value_fn,
                         policy_fn = self.policy_fn,
                         rollout_policy_fn = self.rollout_policy_fn,
                         playout_depth=playout_depth,
                         n_playout = n_playout)

    def step(self,timestep,env):

        move = self.mcts.get_move(timestep,env)
        self.mcts.update_with_move(move)
        return move
```

其初始化时在 MCTS 类中注册策略模块的策略函数和值函数,以及模拟模块的 rollout 策略函数.其中 random_policy_fn 与 random_value_fn 为随机 rollout 的 MCTS 的策略函数和值函数,它们的实现如下:

```
def random_policy_fn(time_step, player_id):

    legal_actions = time_step.observations["legal_actions"][player_id]
    probs = np.zeros(NUM_ACTIONS)
    probs[legal_actions] = 1
    probs /= sum(probs)
    return [i for i in zip(range(len(probs)),probs)]

def random_value_fn(time_step, player_id):

    return normal(scale=0.3)
```

结合 MCTS 的要求,A2C 算法中天然包含策略函数(Actor)和值函数(Critic),适合与 MCTS 相整合,故 policy_module 为 A2C 的算法类.我在 A2C 代码中额外准备了 value_fn 和 policy_fn,作为接口提供给 MCTS.

2.3 MCTS方法的运行结果

MCTS 的运行结果就是整个算法的训练结果.请见第四节训练结果部分.

3 实现对手池方法

3.1 实现模型的加载与保存

3.1.1 模型的保存

模型保存的样例代码在 DQN 算法的中已给出,其要求在初始化 tf.train.Saver 时传入需要保存的变量信息.但 A2C 算法的框架代码中并未给出模型加载与保存的代码,原因可能是 A2C 算法中不同的网络数量过多,较为繁杂.但考虑到作业的要求,我将其一并实现了.代码在 algorithms/policy_gradient.py 中,具体代码如下:

```
if loss_class.__name__ == "BatchA2CLoss":
    value_head = snt.Linear(output_size=1, name="baseline")
    self._baseline = tf.squeeze(value_head(torso_out), axis=1)
else:
    # Add q-values head otherwise
    value_head = snt.Linear(output_size=self._num_actions, name="q_values_head")
    self._q_values = value_head(torso_out)
```

```
self.variable_list =  list(cnn_net.variables) + list(mlp_net.variables) + list(policy_head.variables) \
                        + list(value_head.variables) + critic_optimizer.variables() + pi_optimizer.variables()

self._saver = tf.train.Saver(var_list = self.variable_list)
```

另外,刘驭壬学长在之前组会上提到他在加载某个网络并继续训练时,发现网络性能的下降.他推测原因是优化器中的参数的丢失.因此我此处额外加上了两个优化器的变量集合.

3.1.2 模型的加载

代码中有两处需要加载模型:一是重启训练时,需要加载上次训练结束后保存的结果;二是训练中对手模型的加载.前者只需要加载一次模型,而后者需要加载两次(本模型和对手模型).但加载相同模型两次时 Tensorflow 报错,提示某些节点缺失.这可能是连续加载两次模型后计算图中的某些节点被共享了,导致第二次权重加载的异常.为此,我采取了一个替代方法:先加载一个模型,再用 tf.assign 批量更新另一个模型.代码在 rival_pool/rival_pool_train.py 中,具体代码如下:

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

restore_agent_op = tf.group([
            tf.assign(self_v, rival_v)
            for (self_v, rival_v) in zip(self_.variable_list,rival.variable_list)
        ])
sess.run(restore_agent_op)
```

为实现已保存模型的自动加载,我还在 utils.py 中实现了 get_max_idx()函数,自动查找目标文件夹中最新的模型进行加载.

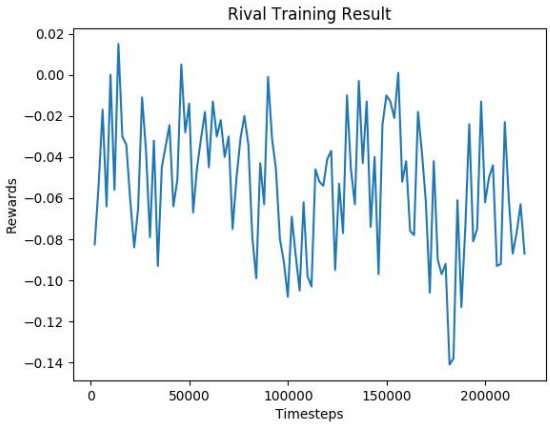
3.2 对手池方法的代码实现

对手为 A2CAgent 与对手为 RandomAgent 的代码差别不大,需要修改之处为对手的 step 函数:其应处于 evaluate 模式,且不考虑当前 player_id(框架代码中如果 player_id 与本模块 player_id 不符会返回空值).具体代码如下:

```
if is_rival:
    info_state = time_step.observations["info_state"][self.player_id]
    legal_actions = time_step.observations["legal_actions"][self.player_id]
    action, probs = self._act(info_state, legal_actions, is_evaluation=True)

    return StepOutput(action=action, probs=probs)
```

3.3 对手池方法的训练过程

使用第二节中提到的训练结果最好的网络对对手模型和本体模型进行初始化并训练的结果如下:



可见智能体“与过去的自己”对弈的结果较差,很难击败“过去的自己”.其原因可能是与 RandomAgent 对弈已经完全挖掘了当前网络结构的表示能力.考虑到训练对手是贪心选择了与 RandomAgent 对战效果最好的模型,可知这样的单步贪心不一定会带来全局最优.为此,我额外挑选了一些结构较复杂,潜在容量可能较大,但与 RandomAgent 对战效果并非最好的网络模型作为对手和本体的初始化模型,进行对手池训练,但实验效果皆不理想,可能是结构复杂的模型需要更长时间更强的算力进行训练,不是笔记本的 CPU 所能承受的.

4 训练结果

最终实验结果的评定由训练好的 MCTS 网络和随机 rollout 的 MCTS 网络的对弈给出.以下是它们对弈十

次的实验记录:

```
I1213 20:44:18.625975 140658621970240 mcts_vs_random.py:123] MCTS INIT OK!!
I1213 20:45:41.202489 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:45:53.816700 140658621970240 mcts_vs_random.py:199] [-1, 1]
I1213 20:48:06.140342 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:51:35.370108 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:53:46.548774 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:55:27.608590 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:57:39.386697 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:59:30.604049 140658621970240 mcts_vs_random.py:199] [-1, 1]
I1213 21:02:08.941816 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 21:02:22.559658 140658621970240 mcts_vs_random.py:199] [1, -1]
```

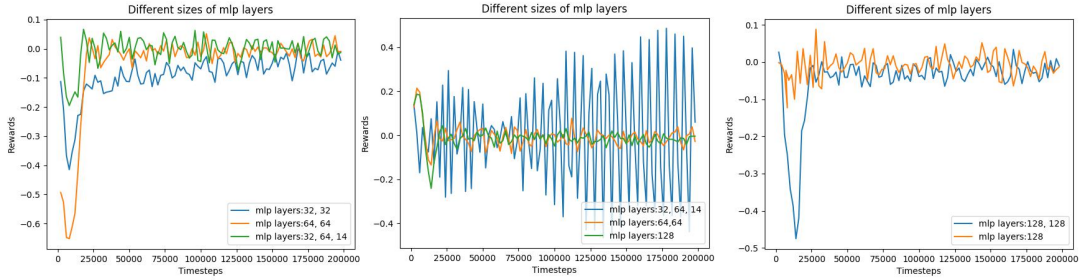
可见使用强化学习策略和价值网络的 MCTSAgent 的胜率达到了 80%.

5 超参数调整过程

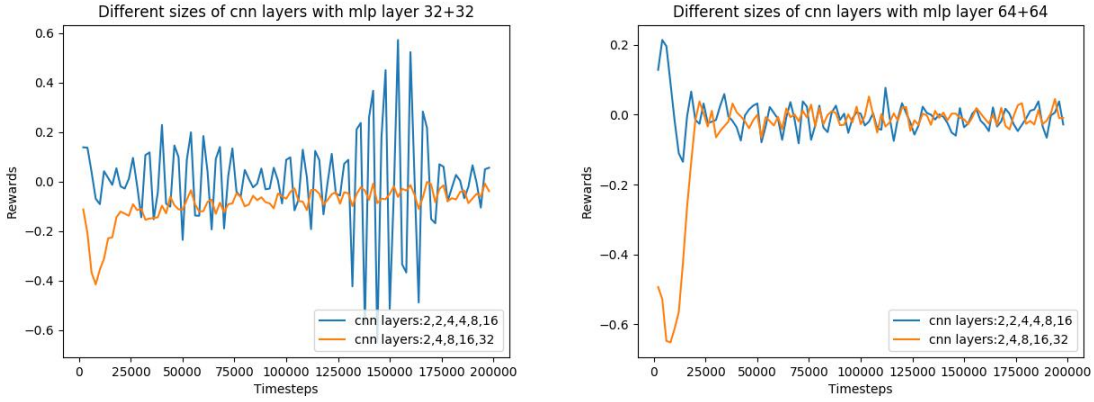
5.1 网络结构的调整

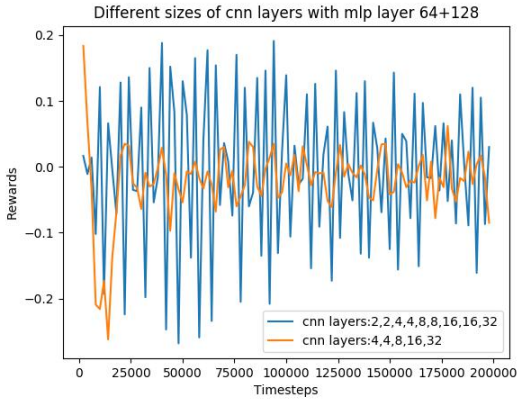
5.1.1 CNN 与 MLP 层数的调整

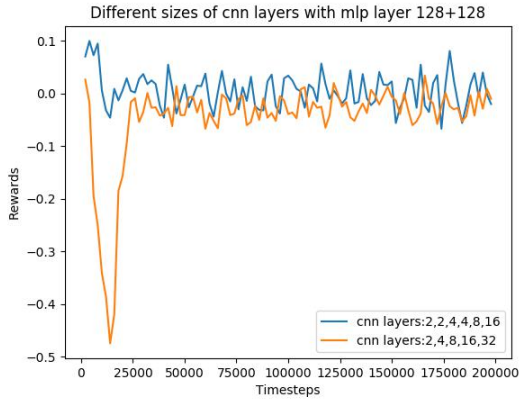
我先在 DQN 算法上进行了超参数调整.首先对比了给定卷积层参数,不同全连接层规模对性能的影响.实验我针对不同的卷积层参数,进行了三组对比实验.实验结果如下:



结果显示,32-64-14 个节点的全连接层效果较好,64-64 次之,32-32 和 128-128 节点的全连接层效果最差.其次我对比了给定全连接层参数,不同卷积层规模对性能的影响.实验结果如下:







结果显示,在 DQN 算法中,具有“2,2”或“4,4”这种相同 channel 数堆叠的卷积层的网络能取得更好的性能,且更深的网络最优性能更好,但也伴随更多震荡.

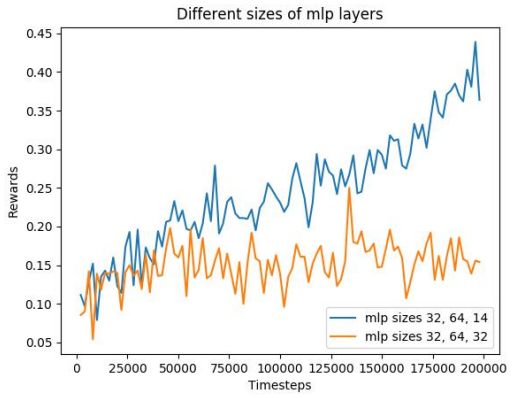
总体而言, DQN 算法没有取得较好较稳定的训练效果.

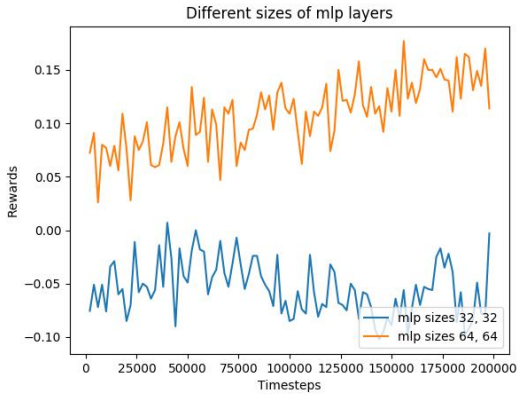
5.1.2 Batch Normalization 的加入

为解决上述训练不稳定的问题,我在网络中加入了 Batch Normalization(有无 Batch Normalization 应该也算超参数).具体代码如下:

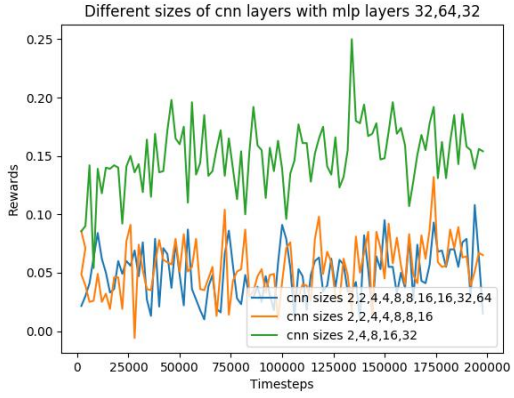
```
cnn_net = snt.nets.ConvNet2D(output_channels = cnn_parameters[0],
                             kernel_shapes = cnn_parameters[1],
                             strides = cnn_parameters[2],
                             paddings = cnn_parameters[3],
                             normalization_ctor=snt.BatchNormV2,
                             activate_final=True)

mlp_net = snt.nets.MLP(output_sizes=self._layer_sizes,activate_final=True)

policy_head = snt.Linear(output_size=self._num_actions, name="policy_head")



torso_out = mlp_net(tf.layers.flatten(cnn_net(self._info_state_ph,is_training=True)),is_training=True)
torso_out_eval = mlp_net(tf.layers.flatten(cnn_net(self._info_state_ph,is_training=False)),is_training=False)


self._policy_logits = policy_head(torso_out)
self._policy_logits_eval = policy_head(torso_out_eval)


self._policy_probs = tf.nn.softmax(self._policy_logits)
self._policy_probs_eval = tf.nn.softmax(self._policy_logits_eval)
```

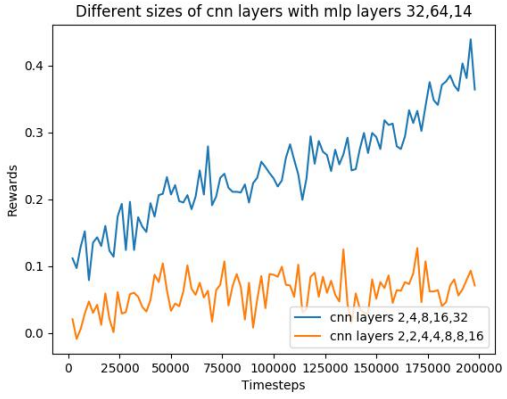
具体地,代码在 ConvNet2D 模块初始化时加入 Batch Normalization 相关参数,并在前向传播时根据是否处于训练状态定义相应量的 train 节点和 evaluation 节点以备调用,对应 Batch Normalization 的 train 状态和 evaluation 状态.此外,考虑到原论文和实际 MCTS 的要求,我将调整超参数所用的算法由 DQN 换为了 A2C.给定卷积层,对比全连接层的实验结果如下:





可见与 DQN 算法类似,32+64+14 的全连接层结构取得了最好的实验效果,32+32 的全连接层结构效果最差.给定全连接层,对比卷积层的实验结果如下:





与 DQN 算法相反,具有相同 channel 堆叠结构的卷积层反而效果最差,更简单的 2+4+8+16+32 的卷积层结构取得了最好的效果.

总的来说,使用 A2C 算法,加入 Batch Normalization 后训练效果较好,与普通 DQN 相比提升明显.

5.2 MCTS参数的调整

可调整的 MCTS 参数为开始 rollout 的深度和节点搜索展开的总次数.考虑到我具体实现 MCTS 时,action_policy 和 rollout_policy 选择了同样的网络和同样的参数(受制于上文提到的加载两个相同网络的 bug),开始 rollout 的深度对 MCTS 的性能没有影响.故实验比较了不同节点展开次数对 MCTS 性能的影响.注意调整此参数时,与之对弈的 RandomRolloutMCTSAgent 的参数并未修改.实验结果如下:

搜索展开 100 次:

```
I1213 20:45:41.202489 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:45:53.816700 140658621970240 mcts_vs_random.py:199] [-1, 1]
I1213 20:48:06.140342 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:51:35.370108 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:53:46.548774 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:55:27.608590 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:57:39.386697 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 20:59:30.604049 140658621970240 mcts_vs_random.py:199] [-1, 1]
I1213 21:02:08.941816 140658621970240 mcts_vs_random.py:199] [1, -1]
I1213 21:02:22.559658 140658621970240 mcts_vs_random.py:199] [1, -1]
```

搜索展开 150 次:

```
I1213 21:21:01.709067 140402031945536 mcts_vs_random.py:202] [1, -1]
I1213 21:25:01.408745 140402031945536 mcts_vs_random.py:202] [1, -1]
I1213 21:25:38.690265 140402031945536 mcts_vs_random.py:202] [-1, 1]
I1213 21:26:06.025836 140402031945536 mcts_vs_random.py:202] [-1, 1]
I1213 21:30:21.835044 140402031945536 mcts_vs_random.py:202] [1, -1]
I1213 21:34:39.226240 140402031945536 mcts_vs_random.py:202] [-1, 1]
I1213 21:37:42.265127 140402031945536 mcts_vs_random.py:202] [1, -1]
I1213 21:41:46.313240 140402031945536 mcts_vs_random.py:202] [-1, 1]
I1213 21:44:54.336240 140402031945536 mcts_vs_random.py:202] [1, -1]
I1213 21:50:06.697634 140402031945536 mcts_vs_random.py:202] [1, -1]
```

搜索展开 200 次:

```
I1213 21:34:19.326539 140441179322176 mcts_vs_random.py:202] [1, -1]
I1213 21:34:36.620157 140441179322176 mcts_vs_random.py:202] [-1, 1]
I1213 21:35:04.589100 140441179322176 mcts_vs_random.py:202] [1, -1]
I1213 21:35:32.129242 140441179322176 mcts_vs_random.py:202] [-1, 1]
I1213 21:35:45.408952 140441179322176 mcts_vs_random.py:202] [-1, 1]
I1213 21:39:22.099360 140441179322176 mcts_vs_random.py:202] [1, -1]
I1213 21:46:00.863716 140441179322176 mcts_vs_random.py:202] [-1, 1]
I1213 21:49:06.584528 140441179322176 mcts_vs_random.py:202] [1, -1]
I1213 21:55:03.208632 140441179322176 mcts_vs_random.py:202] [1, -1]
I1213 22:01:01.057631 140441179322176 mcts_vs_random.py:202] [1, -1]
```

可见增大搜索展开的次数并没有提升 MCTS 的性能,原因可能是 100 次搜索展开已经能使 Agent 较好评估当前状态下的每个动作,无需更多的搜索展开.此外,更多的搜索展开带来了明显更长的耗时.

6 不足与反思

6.1 特征提取

框架代码中提供的特征有当前棋盘信息和当前执子的 Player,其中当前棋盘的信息是较为原始的棋子排布的像素点,维度为 BOARD_SIZE*BOARD_SIZE*1. 而在 AlphaGo 的实现中,棋盘信息的维度为 BOARD_SIZE*BOARD_SIZE*48,其中包含了棋子在历史上的出现时间和一些棋局发展的信息.这些额外的信息融合了人类的先验知识,使得 CNN 的特征提取和策略网络的训练更加容易,带来模型性能的提升.因此,我的代码中的特征提取方法是一个不足之处.

6.2 并行计算

AlphaGo 是在 GPU 上进行网络计算,在 CPU 上进行模拟,其中搜索过程使用了数十个线程以加快速度,网络计算也高度并行.然而我的代码没有考虑到性能因素,完全是单核单线程版本,这使得计算资源被极大浪费了.我采取的一个应对措施是调整超参数时手动运行多个超参数检验任务,提高 CPU 使用率.更方便的做法可能是将任务写在 Makefile 里并用 make -jn 实现并行,但由于时间关系没有实现.

6.3 Tensorflow的使用

我对 Tensorflow 还不够熟悉,出现诸如多个相同模型加载时节点缺失的问题感到无从下手,代码的修改也不是随心所欲,需要反复查阅文档.

以上就是我本次实验报告的全部内容,感谢您的阅读!