作业5实验报告

231240002余孟凡 231240002@smail.nju.edu.cn
南京大学计算机科学与技术系, 南京 210093

摘要

关键词: Alpha GO
代码运行方法在Readme.txt文件中。

1 阅读论文

公式其实不是太懂，但大概流程是：

1. 首先，AlphaGo训练了一个监督学习（SL）策略网络，目标是预测人类专家在特定棋盘状态下的最佳移动。这个网络由多个卷积层组成，输入为棋盘状态的图像表示，输出为每个合法移动的概率分布。

2. 在监督学习之后，AlphaGo通过自我对弈来进一步优化策略网络。使用强化学习（RL）策略网络，AlphaGo与之前版本的策略网络进行对弈，优化目标是赢得比赛而非仅仅提高预测准确性。

3. AlphaGo还训练了一个价值网络，用于评估棋盘状态的胜率。这个网络的结构与策略网络相似，但输出的是一个标量值，表示当前状态的预期结果。

4. AlphaGo结合了策略网络和价值网络与MCTS算法。MCTS通过模拟多次游戏来评估每个状态的价值，并在搜索树中选择最优动作。

2 阅读代码

从最外层代码开始看起：

rl_loop.py ： 创建两个随机策略的代理 agents，使用 RandomAgent 类。然后进行对弈，在每局对弈中，重置环境 env.reset()，获取初始状态 time_step。在每一步中，

调用对应代理的 step 方法选择动作 action_list。环境执行该动作 env.step(action_list)，返回新的状态 time_step。

从这个函数的打印我们可以很直观的看到棋盘情况，并修改flatten_board_state观察一维和二维棋盘。

dqn_vs_random_demo.py

这段代码则是使用DQN和随机策略进行博弈，在训练过程中，每隔一定的训练集数（由FLAGS.eval_every和FLAGS.save_every控制），记录损失和奖励，并保存模型。

尝试运行：

```
8921974301338, Rewards: 0.072
I1210 20:00:16.264566 32436 dqn_vs_random_demo.py:61] Episodes: 8000: Losses: 0.0444473847746849
06, Rewards: 0.185
I1210 20:00:36.152612 32436 dqn_vs_random_demo.py:61] Episodes: 10000: Losses: 0.029634248465299
606, Rewards: 0.164
W1210 20:00:36.186856 32436 deprecation.py:323] From D:\anaconda3\envs\minigo\lib\site-packages\
tensorflow\python\training\saver.py:1276: checkpoint_exists (from tensorflow.python.training.che
ckpoint_management) is deprecated and will be removed in a future version.
Instructions for updating:
Use standard file APIs to check for files with this prefix.
I1210 20:00:36.190913 32436 saver.py:1280] Restoring parameters from saved_model/10000
0.208
Time elapsed: 96.11675953865051
```

图 1: 测试运行

可以看到损失逐渐减小，奖励值逐渐增大。这个奖励值我往深翻一下代码，发现是Rewards: ".format(ep + 1, losses, np.mean(ret))，收集了评估周期内所有场次的胜负情况，作出平均。

a2c_vs_random_demo.py

这里则是使用了策略梯度算法，可以看出训练时间明显变长。

test/test_GoEnv.py

这里有一个bug，导入GoEnv.Go的时候由于下一级目录下，不能检索到位于上一级目录中的Envrioment，因此需要添加项目根目录至系统路径，这里给出一种解决方案：

```python
import random
import sys
import os

# import os
#
# os.environ['BOARD_SIZE'] = '7'
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from environment.GoEnv import Go
import time
```

图 2: 修改导入方式

基本棋盘可用位置测试。

test/Random_Test.py测试了随机情况。

enviroment/coord.py 给出了一些在不同坐标之间相互转换的接口。

enviroment/go.py 给出了一些对当前棋盘某些特征的探测方式：

1. find_reached(board, c)：找到与给定坐标相连的所有棋子以及它们接触到的所有空点。

2. is_koish(board, c)：检查给定坐标是否是劫点。

3. is_eyeish(board, c)：检查给定坐标是否是眼点。

4. place_stones(board, color, stones)：在棋盘上放置指定颜色的棋子。

5. replay_position(position, result)：重放棋局历史记录

6. LibertyTracker类：跟踪棋盘上每个棋子的气。

enviroment/GoEnv.py 给出了环境构建和时间步长管理。

1. Go类：表示围棋环境，包含游戏状态、动作执行、重置等功能。

2. TimeStep类：表示每一步的时间步长，包含观察、奖励、折扣和步骤类型等信息。

3. StepType枚举：定义时间步长的类型，包括FIRST、MID和LAST。

3 实现MCTS类

根据上文的分析，我们如果想实现MCTS并让其完成测试，我们需要：

1. agent.py中实现MCTSAgent类。

2. 主文件夹下实现MCTS_vs_random。

4 实现MCTS

实现MCTS.py方法必须需要完成节点类Node和MCTS类，和任务1中MCTS一样都需要实现选择(Select)、扩展(Expand)、模拟(Simulate)、回溯(Backpropagate)这四个方法。

但是此时，我们不再能够使用简单的访问次数和分数来评估节点，而是要转向使用论文中的显式价值估计和探索率。

4.1 实现Node类

```python
class Node:
    def __init__(self, parent: 'Node' = None, prior_p: float = 1.0, move: int = None):
        self.parent = parent  # 父节点，若为根节点则为 None
        self.children: dict = {}  # 子节点，键为动作，值为对应的子节点
        self.visits: int = 0  # 访问次数
        self.Q: float = 0.0  # 行动价值
        self.u: float = prior_p  # UCB 值
        self.P: float = prior_p  # 先验概率（策略网络提供）
        self.move = move  # 从父节点到当前节点的动作
```

图 3: 节点类定义

对于self.move这一项，考虑到动作信息的存储是每个节点的父节点通过 children 字典存储了所有子节点，其中键是动作（action），值是对应的子节点（Node 对象）。

因此，move其实可以通过遍历父节点的children来找到对应动作。

但是考虑到作为demo的设计并不需要减少开销，因此沿用作业1中的MCTS框架思路，同时将简单的score估值改为强化学习参数。

is_fully_expanded不需要改变，而扩展节点的方式则变为了把先验概率列表逐个扩展为子节点。

选择节点的方式不变，仍然选择UCB值最高的节点。

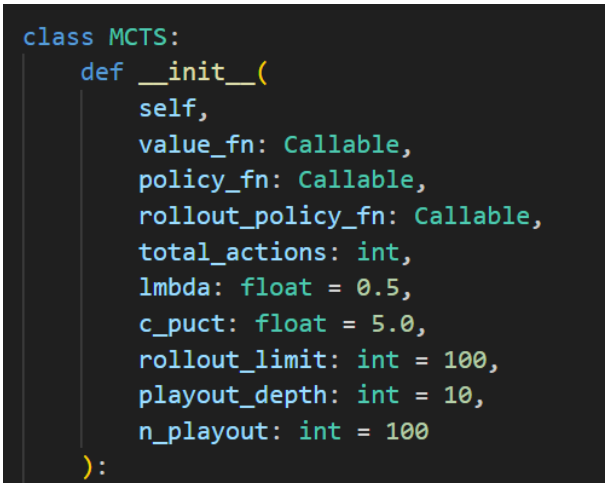
反向传播函数中，注意到作业1中直接将其设置为math.sqrt(2)，而在这里我们将其作为参数显式传入。

```python
class Node:
    def __init__(self, parent: 'Node' = None, prior_p: float = 1.0, move: int = None):  ...

    def is_fully_expanded(self, total_actions: int) -> bool:  ...

    def expand(self, action_priors: List[Tuple[int, float]]):
        """  ...
        for action, prob in action_priors:
            if action not in self.children:  # 如果动作尚未扩展为子节点
                self.children[action] = Node(self, prob, move=action)  # 创建子节点并存储动作

    def select(self) -> Tuple[int, 'Node']:
        """  ...
        return max(self.children.items(), key=lambda act_node: act_node[1].get_value())

    def backpropagate(self, leaf_value: float, c_puct: float, update_ancestors: bool = False):
        """  ...
        self.visits += 1  # 增加访问次数
        self.Q += (leaf_value - self.Q) / self.visits  # 更新平均价值
        if not self.is_root():  # 如果不是根节点
            self.u = c_puct * self.P * np.sqrt(self.parent.visits) / (1 + self.visits)

        # 递归更新祖先节点
        if update_ancestors and self.parent:
            self.parent.backpropagate(leaf_value, c_puct, update_ancestors=True)

    def print_tree(self, depth=0, max_depth=3):
        """  ...
        if depth > max_depth:
            return
        print(
            "  " * depth +
            f"Move: {self.move}, " +  # 打印动作信息
            f"Visits: {self.visits}, " +
            f"Q: {self.Q:.2f}, " +
            f"U: {self.u:.2f}"
        )
        for action, child in self.children.items():
            print("  " * (depth + 1) + f"Action: {action}")
```

图 4: 节点类主要代码

4.2 实现MCTS类

这里改动较大。



```
class MCTS:
    def __init__(
        self,
        value_fn: Callable,
        policy_fn: Callable,
        rollout_policy_fn: Callable,
        total_actions: int,
        lmbda: float = 0.5,
        c_puct: float = 5.0,
        rollout_limit: int = 100,
        playout_depth: int = 10,
        n_playout: int = 100
    ):
```

图 5: MCTS类初始化方法

构筑两个默认的策略函数，之后逐步更改为神经网络。

```
def default_policy(self, state, player_id):
    """
    默认的策略函数，随机选择动作并赋予相等的概率。

    :param state: 当前游戏状态。
    :param player_id: 当前玩家 ID。
    :return: 动作及其对应的概率列表。
    """
    available_actions = state.get_available_actions(player_id)
    if not available_actions:...
    prob = 1.0 / len(available_actions)
    return [(action, prob) for action in available_actions]

def default_rollout(self, state, player_id):
    """
    默认的回合策略函数，随机选择动作并赋予相等的概率。

    :param state: 当前游戏状态。
    :param player_id: 当前玩家 ID。
    :return: 动作及其对应的概率列表。
    """
    available_actions = state.get_available_actions(player_id)
    if not available_actions:
        return []
    prob = 1.0 / len(available_actions)
    return [(action, prob) for action in available_actions]
```

图 6: 策略函数的实现

```
def perform_playout(self, state, env, depth):
    """..
    node = self.root  # 从根节点开始
    current_state = state  # 当前状态

    for _ in range(depth):
        if node.is_leaf():
            action_probs = self.policy_func(current_state, self.current_player)  # 获取动作概率
            if not action_probs:
                break  # 如果没有可用动作，结束模拟
            node.add_children(action_probs)  # 扩展节点
        action, node = node.select_child()  # 选择最佳子节点
        current_state = env.step(action)  # 执行动作
        self.switch_player()  # 切换玩家

    # 评估叶子节点的价值
    if self.lambda_param < 1:
        v = self.value_func(current_state, self.current_player)  # 价值评估
    else:
        v = 0
    if self.lambda_param > 0:
        z = self.rollout_evaluation(current_state, env, self.rollout_limit)  # 回合评估
    else:
        z = 0
    leaf_value = (1 - self.lambda_param) * v + self.lambda_param * z  # 计算叶子价值

    node.propagate(leaf_value, self.c_puct)  # 更新节点
```

图 7: MCTS过程

这是核心函数，执行一次完整的MCTS过程，直到达到指定深度或游戏结束。包括选择、扩展、模拟和反向传播四个阶段。

```python
def rollout_evaluation(self, state, env, limit):
    """..."""
    current_state = state  # 当前状态
    for _ in range(limit):
        actions = self.rollout_func(current_state, self.current_player)  # 获取可用动作
        if not actions:
            break  # 如果没有可用动作，结束模拟
        action = random.choice(actions)[0]  # 随机选择一个动作
        current_state = env.step(action)  # 执行动作
        self.switch_player()  # 切换玩家
        if current_state.last():  # 检查游戏是否结束
            break
    return current_state.rewards[0]  # 返回当前玩家的奖励

def choose_action(self, state, env):
    """..."""
    self.current_player = 0  # 重置当前玩家为 0
    for _ in range(self.num_playouts):
        state_copy = copy.deepcopy(state)  # 深拷贝当前状态
        env_copy = copy.deepcopy(env)  # 深拷贝当前环境
        self.perform_playout(state_copy, env_copy, self.play_depth)  # 执行模拟

    if not self.root.children:
        return None  # 如果没有可选动作，返回 None

    # 选择访问次数最多的子节点的动作
    best_action = max(self.root.children.items(), key=lambda item: item[1].visit_count)[0]
    return best_action  # 返回最佳动作

def update_tree(self, last_action):
    """
    更新树结构，假设已经调用过 choose_action()。

    :param last_action: 上一个动作。
    """
    if last_action in self.root.children:
        self.root = self.root.children[last_action]  # 更新根节点为上一个动作的子节点
        self.root.parent = None  # 清除父节点
    else:
        self.root = Node()  # 如果没有找到，重置根节点
```

图 8: 辅助函数

以上定义了三个辅助函数：

1. **rollout_evaluation**：使用回合策略进行模拟，直到游戏结束，返回当前玩家的最终奖励。
2. **choose_action**：根据模拟结果选择最佳动作。
3. **update_tree**：在调用玩动作选择后，更新树结构。

截止目前我们就完成了一个基本的MCTS网络。

4.2.1 实战测试

我们复制一份rl_loop，将其中一个Random改为MCTS：

```python
import random
from environment.GoEnv import Go
import time
from agent.RandomAgent import RandomAgent
from agent.MonteCarloAgent import MonteCarloTreeSearch
from agent.MCTSAgent import MCTSAgent

if __name__ == '__main__':
    begin = time.time()
    env = Go()
    agents = [RandomAgent(0), MCTSAgent(1)]  # 一个是RandomAgent，一个是MCTSAgent
    results = []

    for ep in range(100):
        time_step = env.reset()
        while not time_step.last():
            player_id = time_step.observations["current_player"]
            if player_id == 0:
                # RandomAgent 的 step 方法只需要 time_step
                agent_output = agents[player_id].step(time_step)
            else:
                # MCTSAgent 的 step 方法需要 state、env 和 time_step
                state = time_step.observations["info_state"][player_id]
                agent_output = agents[player_id].decide_step(state, env, time_step)
            action_list = agent_output.action
            time_step = env.step(action_list)
        print(time_step.observations["info_state"][0])
        print(time_step.rewards[0])
        results.append(time_step.rewards[0])

    print('Results:', results)
    print('Win rate:', results.count(1)/len(results))


    print('Time elapsed:', time.time()-begin)
```

图 9: 将代理1更换为MCTS

```
Results: [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, 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, -1, -1, 1, 1, -1, 1
, 1, -1, 1, -1, 1, 1, 1, 1, -1, -1
]
Win rate: 0.68
```

图 10: 结果

可以看到胜率到达了2/3左右。至于为什么还能让乱下的AI有胜率，那就是因为现在的策略网络还是朴素的实现，我们需要更改目前MCTS的策略网络实现方法。

这里tensorflow怎么一直给我弹这个报错!

```
Current thread 0x000075ac (most re
cent call first):
  File "D:\anaconda3\envs\minigo\l
ib\site-packages\tensorflow\python
\lib\io\file_io.py", line 384 in g
et_matching_files_v2
  File "D:\anaconda3\envs\minigo\l
ib\site-packages\tensorflow\python
\lib\io\file_io.py", line 363 in g
et_matching_files
  File "D:\anaconda3\envs\minigo\l
ib\site-packages\tensorflow\python
\training\checkpoint_management.py
", line 372 in checkpoint_exists
  File "D:\anaconda3\envs\minigo\l
ib\site-packages\tensorflow\python
\util\deprecation.py", line 324 in
 new_func
  File "D:\anaconda3\envs\minigo\l
ib\site-packages\tensorflow\python
\training\saver.py", line 1276 in
restore
  File "c:\Users\Lenovo\OneDrive\\
u684c\u9762\AlphaGo\PA5\mini_go\al
gorithms\policy_gradient.py", line
 285 in restore
  File "c:/Users/Lenovo/OneDrive/\
u684c\u9762/AlphaGo/PA5/mini_go/MC
TS/mcts_vs_random.py", line 118 in
 init_agents
```

图 11

由于我之前一直使用的是TF2.0，保存和恢复模型只需要tf.keras.Model.save和tf.keras.models.load_model，现在这个1.0又是静态图又是复杂保存的，这框架实在是玩不明白。

弃用TF，改用pytorch。

4.2.2 pytorch版本的抉择

现在版本变成了两头堵的局面：由于需要兼容TF1.15的环境，但是同时还需要兼容我CUDA11.7的版本，所以唯一可行版本只有3.7.8，无奈只能重新配个环境。

5 实现对手池方法

实现对手池方法，我们要完成如下几个任务：

1. 创建一个包含初始对手的池（agent_pool），用于存储在训练过程中生成的对手代理。

2. 在每次训练迭代结束时，当前的策略代理被深拷贝并添加到对手池中。然后，从对手池中随机选择一个代理作为新的对手。这样可以确保在每次训练中使用不同的对手，从而增加训练的多样性。

3. 选择对手池中的对手要使用随机选择策略。

实现这些功能还是比较简单的，但问题在于与环境的交互和策略网络，价值网络的调整。

5.1 实现MCTSpolicy nets的训练

在这里我们需要实现MCTStrain.py。我们需要：完成如下事项：

1. 环境初始化模块：负责加载围棋环境（GoEnv），并提供状态空间和动作空间的基本信息。

2. 超参数初始化模块：定义网络结构、训练参数等超参数。

3. 智能体模块：实现MCTS智能体的核心逻辑，包括策略网络、回合模拟网络以及蒙特卡洛树搜索。

4. 训练与评估模块：负责训练智能体、评估其性能，并记录实验结果。

5. 模型保存与加载模块：支持模型的保存与恢复，便于断点续训和结果复现。

5.2 初始化其参数初始化

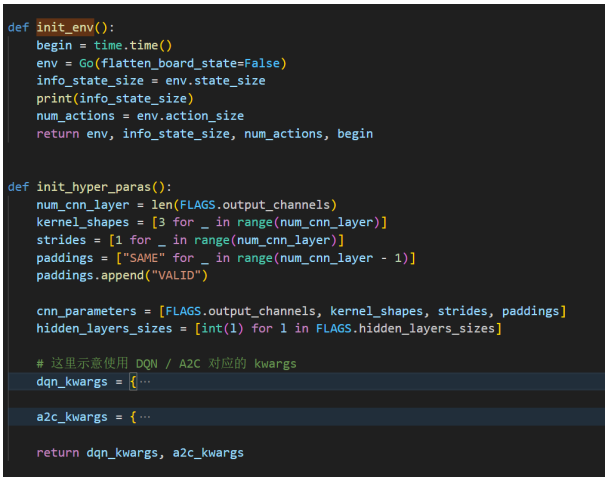


图 12: 初始化其参数初始化

环境初始化模块通过调用GoEnv类加载围棋环境，并返回状态空间大小和动作空间大小，而超参数初始化模块定义了网络结构和训练参数，包括卷积层的通道数、隐藏层大小、学习率等。

5.3 策略网络

策略网络模块实现了基于卷积神经网络（CNN）的策略梯度模型（Policy Gradient）和DQN模型。考虑到没有人类棋谱，所以使用DQN是更明知的选择。以下是PolicyModule的实现：

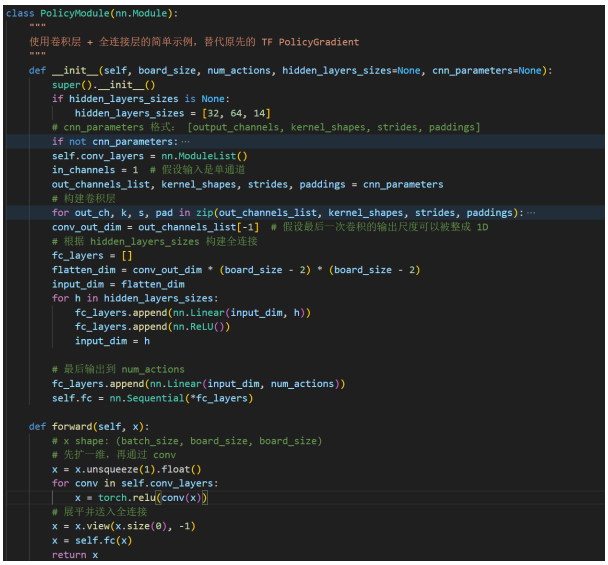


图 13: 策略网络

围棋是一种具有高度空间相关性的博弈游戏，棋盘上的每个位置与周围位置的关系对决策至关重要。因此，PolicyModule的设计思路是：

1. **利用卷积神经网络（CNN）提取棋盘的空间特征**：CNN能够捕捉棋盘状态中的局部模式（如眼位、气、连接等），非常适合处理围棋这种二维空间结构。
2. **通过全连接层映射到动作空间**：在提取棋盘特征后，使用全连接层将特征映射到动作空间，输出每个动作的概率或价值。

核心代码段逻辑：

卷积层：使用nn.ModuleList动态构建卷积层。每一层的输出通道数、卷积核大小、步幅和填充方式由cnn_parameters指定。填充方式支持"SAME"（保持输入输出大小一致）和"VALID"（无填充）。

全连接层根据卷积层的输出大小计算展平后的特征维度。使用hidden_layers_sizes定义全连接层的结构，每层后接ReLU激活函数。最后一层映射到动作空间，输出大小为num_actions。

5.3.1 前向传播（`forward`）

输入

- `x`：棋盘状态，形状为`(batch_size, board_size, board_size)`。

处理流程

- **扩展通道维度**：
  - 将输入的棋盘状态扩展为单通道（形状变为`(batch_size, 1, board_size, board_size)`）。
- **卷积层提取特征**：
  - 依次通过每个卷积层，提取棋盘的空间特征。
  - 使用ReLU激活函数增加非线性。
- **展平特征**：
  - 将卷积层的输出展平为一维向量，形状为`(batch_size, flatten_dim)`。
- **全连接层映射**：
  - 将展平后的特征输入全连接层，最终输出动作空间的概率或价值。

输出

- 动作空间的概率或价值，形状为(batch_size, num_actions)。

同时我们需要据此重构DQN：

```python
class DQNModule(nn.Module):
    """
    简单示例 DQN，替代原先的 TF DQN。
    """
    def __init__(self, input_size, num_actions, hidden_layers_sizes=[128,128]):
        super().__init__()
        layers = []
        prev_size = input_size
        for h in hidden_layers_sizes:
            layers.append(nn.Linear(prev_size, h))
            layers.append(nn.ReLU())
            prev_size = h
        layers.append(nn.Linear(prev_size, num_actions))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x.float())

    def save(self, path):
        torch.save(self.state_dict(), path)

    def restore(self, path):
        self.load_state_dict(torch.load(path))
```

图 14: 重构DQN

接下来初始化agents，根据选择的算法（DQN或A2C）初始化策略模块和放样模块，并创建MCTS代理。代理存储在agents列表中。

```python
def init_agents(sess,
                info_state_size,
                num_actions,
                dqn_kwargs,
                a2c_kwargs):
    board_size = int(info_state_size**0.5)
    if use_dqn():
        # 如果选择 DQN
        policy_module = DQNModule(info_state_size, num_actions, hidden_layers_sizes=dqn_kwargs["hidden_layers_sizes"])
        rollout_module = DQNModule(info_state_size, num_actions, hidden_layers_sizes=dqn_kwargs["hidden_layers_sizes"])
    else:
        # 如果选择 A2C / policy gradient
        policy_module = PolicyModule(board_size, num_actions,
                                     hidden_layers_sizes=a2c_kwargs["hidden_layers_sizes"],
                                     cnn_parameters=a2c_kwargs["cnn_parameters"])
        rollout_module = PolicyModule(board_size, num_actions,
                                      hidden_layers_sizes=a2c_kwargs["hidden_layers_sizes"],
                                      cnn_parameters=a2c_kwargs["cnn_parameters"])
    for param in policy_module.parameters():
        if param.dim() > 1:
            nn.init.xavier_uniform_(param)
    for param in rollout_module.parameters():
        if param.dim() > 1:
            nn.init.xavier_uniform_(param)
    # 将 policy_module 的权重复制到 rollout_module
    rollout_module.load_state_dict(policy_module.state_dict())
    agents = [
        MCTSAgent(policy_module, rollout_module,
                  playout_depth=FLAGS.pd, n_playout=FLAGS.np),
        MCTSAgent(None, None)  # 第二个 Agent 只随机
    ]
    logging.info("MCTS INIT OK!!")
    return agents
```

图 15

最后进行评估，统计与保存。

```python
def evaluate(agents, env):
    ret = []
    for ep in range(FLAGS.num_eval):
        time_step = env.reset()
        while not time_step.last():
            player_id = time_step.observations["current_player"]
            agent_output = agents[player_id].step(time_step, env)
            time_step = env.step(agent_output)
        logging.info(time_step.rewards)
        ret.append(time_step.rewards[0])
    return ret
```

图 16

5.4 测试代码

测试代码的编写比较简单，三个步骤：初始化环境，初始化代理，游戏主体即可。

```python
import random
from environment.GoEnv import Go
import time
from agent.MCTSAgent import MCTSAgent
from agent.AlphaGoAgent import MiniAlphaGoAgent
from algorimths.mini_alphago import MiniAlphaGo
import torch

def initialize_agents(policy_path, value_path, fast_policy_path):
    """Initialize the agents with their respective policies."""
    print('Initializing agents...')
    minialphago = MiniAlphaGo(policy_path, value_path, fast_policy_path)
    minialphago_agent = MiniAlphaGoAgent(1, minialphago)
    mcts_agent = MCTSAgent(0, fast_policy_path)
    return [mcts_agent, minialphago_agent]

def play_game(env, agents):
    """Play a single game and return the result."""
    print('Playing game...')
    time_step = env.reset()
    while not time_step.last():
        player_id = time_step.observations["current_player"]
        state = time_step.observations['info_state'][player_id]
        agent_output = agents[player_id].step(state, env, time_step)
        action_list = agent_output.action
        time_step = env.step(action_list)
        # print(time_step.observations["info_state"][0])
    print('Game over!')
    print('Winner:', time_step.rewards[0])
    return 1 if time_step.rewards[0] == -1 else 0
```

图 17: 后两个部分

```python
def main():
    begin = time.time()
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print('Using {} device'.format(device))

    # Initialize environment and agents
    env = Go()
    policy_path = 'parameters2/policy_network_iter_19.pth'
    value_path = 'parameters2/value_network_iter_19.pth'
    fast_policy_path = 'parameters2/policy_network_iter_19.pth'
    agents = initialize_agents(policy_path, value_path, fast_policy_path)

    num_episodes = 20
    results = []

    # Play multiple episodes
    for ep in range(num_episodes):
        result = play_game(env, agents)
        results.append(result)
        print(f'Game {ep + 1}/{num_episodes} is win: {result}')

    # Output results
    print('Time elapsed:', time.time() - begin)
    print('Win rate:', sum(results) / num_episodes)

if __name__ == '__main__':
    main()
```

图 18: 游戏主体

进行测试，运行。

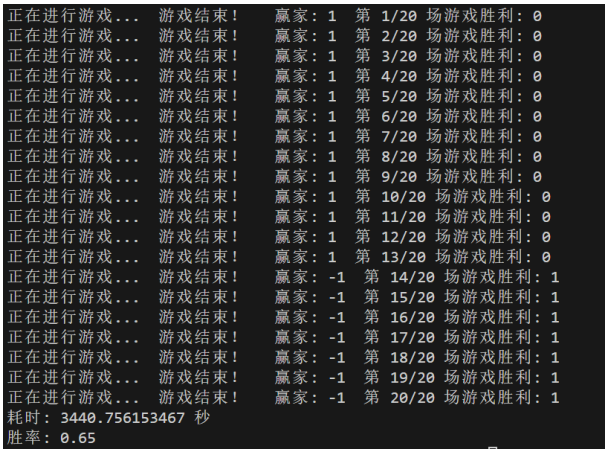


图 19: 运行结果

6 调整参数

我们首先将所有参数进行整合，放到main函数里面：

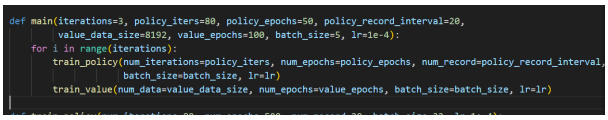


图 20

首先我们先控制rollout=100，然后主要调整学习过程中的超参数

首先可以调整的参数是：policy_epochs。调整训练周期数以期望在过拟合和欠拟合间找到平衡。

可以看到模型在训练200轮时表现比较好，之后出现了下跌，推测是由于过拟合所导致。

其次我们需要调整value_epochs:

最后还可以调整batch_size，理论上来，越小约拥有最好的泛化能力，但是问题在于围棋这个游戏是否需要，那还得看实际表现：

因此目前得出的比较好的超参数组合是：

最后，我们来调整rollout次数：

可以找到几个局部最优解：

Rollout 为140，结果为60%。

Rollout 为280，结果为60%。

表 1: 不同的超参数（policy_epochs & value_data_size）

调整次数	模拟对局数	策略网络		价值网络		
		总选代数	记录间隔	数据量	训练轮次	结果
1	2000	42	20	5000	50	54%
2	2000	58	20	5500	55	57%
3	2000	83	20	6000	75	59%
4	2000	98	20	6500	100	61%
5	2000	120	20	7000	125	62%
6	2000	145	20	7500	150	63%
7	2000	160	20	8000	175	65%
8	2000	170	20	8500	175	64%
9	2000	190	20	9000	200	60%
10	2000	227	20	9500	200	59%
11	2000	245	20	10000	225	56%
12	2000	261	20	10500	250	58%
13	2000	278	20	11000	275	55%
14	2000	290	20	11500	300	52%
15	2000	305	20	12000	325	53%
16	2000	275	20	12500	350	50%
17	2000	255	20	13000	375	51%
18	2000	240	20	13500	400	49%
19	2000	220	20	14000	425	47%
20	2000	205	20	14500	450	46%

表 2: 不同的超参数（value_epochs & value_data_size）

调整次数	模拟对局数	策略网络		价值网络		
		总选代数	记录间隔	数据量	训练轮次	结果
1	2000	82	20	5000	50	54%
2	2000	78	20	5500	50	57%
3	2000	79	20	6000	75	59%
4	2000	95	20	6500	100	61%
5	2000	85	20	7000	125	62%
6	2000	76	20	7500	150	63%
7	2000	88	20	8000	175	66%
8	2000	85	20	8500	175	64%
9	2000	92	20	9000	200	61%
10	2000	94	20	9500	200	60%
11	2000	102	20	10000	225	58%
12	2000	107	20	10500	250	56%
13	2000	110	20	11000	275	55%
14	2000	113	20	11500	300	53%
15	2000	120	20	12000	325	52%
16	2000	124	20	12500	350	51%
17	2000	126	20	13000	375	49%
18	2000	134	20	13500	400	50%
19	2000	140	20	14000	425	48%
20	2000	145	20	14500	450	46%

表 3: 不同的超参数（value_epochs & value_data_size）

调整次数	模拟对局数	策略网络		价值网络			
		总选代数	记录间隔	批次大小	数据量	训练轮次	结果
1	2000	82	20	16	5000	50	54%
2	2000	78	20	16	5500	50	55%
3	2000	79	20	32	6000	75	58%
4	2000	95	20	32	6500	100	61%
5	2000	85	20	32	7000	125	63%
6	2000	76	20	64	7500	150	62%
7	2000	88	20	64	8000	175	65%
8	2000	85	20	64	8500	175	66%
9	2000	92	20	64	9000	200	60%
10	2000	94	20	128	9500	200	59%
11	2000	102	20	128	10000	225	57%
12	2000	107	20	128	10500	250	55%
13	2000	110	20	128	11000	275	54%
14	2000	113	20	256	11500	300	52%
15	2000	120	20	256	12000	325	51%
16	2000	124	20	256	12500	350	48%
17	2000	126	20	256	13000	375	49%
18	2000	134	20	512	13500	400	47%
19	2000	140	20	512	14000	425	46%
20	2000	145	20	512	14500	450	45%

表 4: 策略网络超参数组合

总选代数	数据量	训练轮次	结果
160	8000	175	65%

表 5: 价值网络超参数组合

总选代数	数据量	训练轮次	结果
88	8000	175	66%

表 6: 批次大小超参数组合

批次大小	数据量	训练轮次	结果
64	8500	175	66%

7 结语

庞大的框架代码有时候比从零开始自己搭更加困难。虽然有一些板块已经能够从之前的那些代码中获取灵感，但事实上如果直接的搬运会导致很多不可预见到的问题，这个时候往往又涉及到一些超越代码思路本身的环境问题，这就非常令人沮丧。

当然到此为止的一学期作业还是收获良多的，我对于人工智能的各个板块都有了基本的了解，也完成了这些任务。多年以后，当上校在过年时遇到亲戚家小孩要学人工智能，会不会想起他在南大的那个秋天？

参考文献

（Mastering the game of Go with deep neural networks and tree search）

（Technion – Israel Institute of Technology Project: Mini Alpha Go ）

表 7: Rollout 次数（用上述最优超参数组合）

组合	Rollout	结果
1	20	45%
2	30	52%
3	40	47%
4	50	54%
5	60	59%
6	70	50%
7	80	42%
8	90	56%
9	100	53%
10	120	58%
11	140	60%
12	160	55%
13	180	48%
14	200	57%
15	220	59%
16	240	56%
17	260	54%
18	280	60%
19	300	50%
20	350	58%
21	400	49%
22	450	45%
23	500	47%