

AI2603 强化学习期末项目报告

中国跳棋智能体设计与实现

组员一：张熙和 524030910219

组员二：邵开阳 524030910228

2026 年 1 月 5 日

目录

1 引言	2
2 问题分析与环境理解	2
2.1 游戏规则与状态空间	2
2.2 奖励函数分析	3
2.3 对手策略分析	3
3 Minimax 算法实现	3
3.1 算法设计思路	3
3.2 评估函数设计	4
3.3 Alpha-Beta 剪枝实现	5
3.4 Minimax 算法的局限性思考	6
4 强化学习智能体设计与实现	7
4.1 方法选择与初步尝试	7
4.2 对抗训练框架设计	7
4.3 动作掩码处理	8
4.4 奖励函数优化	9
4.5 从预训练模型开始训练	10
4.6 超参数调优	11
4.7 训练过程与结果	12
5 实验结果与分析	13
5.1 测试方法	13

5.2	Minimax 智能体测试结果	13
5.3	RL 智能体测试结果	13
5.4	测试截图	14
5.5	结果对比与分析	14
6	总结与展望	15
6.1	项目总结	15
6.2	遇到的主要困难	15
6.3	未来改进方向	15
6.4	个人感想	16
A	代码结构	16
B	运行说明	17
C	组员分工	17

1 引言

本项目的 GitHub 仓库地址:<https://github.com/xihe-820/AI2603-FinalProject.git>

本项目的目标是为中国跳棋游戏设计并实现智能体。中国跳棋是一种经典的多人策略棋类游戏，玩家需要将自己的所有棋子从起始区域移动到对角的目标区域。游戏中棋子可以进行普通移动（向相邻格子移动一步）或跳跃移动（跳过相邻的棋子到达更远的位置），而跳跃移动可以连续进行，这使得游戏策略变得非常丰富和复杂。

在本次项目中，我们需要实现两种类型的智能体：一种是基于传统搜索算法的 Minimax 智能体（带 Alpha-Beta 剪枝），另一种是基于深度强化学习的自定义智能体。这两种方法代表了人工智能领域的两大经典范式——符号主义和连接主义，通过这个项目可以深入理解和比较这两种方法在博弈问题上的表现。

说实话，在开始这个项目之前，我们对强化学习在博弈游戏中的应用还是比较陌生的。虽然之前在课上学过 PPO、DQN 这些算法的理论知识，但真正动手实现一个能够击败基线策略的智能体，还是让我们遇到了很多意想不到的困难。这份报告会详细记录我们在实现过程中的思考、尝试和最终的解决方案。

2 问题分析与环境理解

2.1 游戏规则与状态空间

在开始编码之前，我们首先花了一些时间理解项目提供的 PettingZoo 环境。这个环境实现了一个简化版的中国跳棋，支持 2 个玩家（player_0 和 player_3，分别位于棋盘的上方和下方）。棋盘使用六边形坐标系统，大小由 triangle_size 参数控制。在本项目中，我们主要使用 triangle_size=2 的配置进行实验，这意味着棋盘大小为 9×9 （即 $4 \times 2 + 1 = 9$ ）。

环境的观察空间是一个字典类型，包含两个部分：observation 和 action_mask。observation 是一个形状为 $(board_size, board_size, 4)$ 的张量，其中 4 个通道分别表示当前玩家的棋子位置、对手的棋子位置、跳跃起始位置和上次跳跃目标位置。action_mask 则是一个一维向量，标识当前所有合法动作。

动作空间的设计比较巧妙。每个动作由起始位置、移动方向和是否跳跃三个要素组成。具体来说，动作索引的计算公式为：

$$action = is_jump + 2 \times (direction + 6 \times ((r + 2n) + (4n + 1) \times (q + 2n)))$$

其中 q 和 r 是六边形坐标， $direction$ 是 6 个方向之一（右、右上、左上、左、左下、右下）， is_jump 表示是否跳跃。此外，还有一个特殊的 END_TURN 动作，用于在跳跃链中结束当前回合。

2.2 奖励函数分析

理解环境的奖励函数对于设计智能体至关重要。通过阅读环境代码，我们发现奖励函数主要由以下几个部分组成：

首先是胜负奖励，赢得游戏获得 100 分，输掉游戏获得 -100 分。其次是进度奖励，每次移动棋子向目标区域前进会获得正向奖励，后退则获得负向奖励。最后是超时惩罚，如果游戏超过最大步数限制仍未结束，双方都会受到惩罚。

这个奖励函数的设计是比较合理的，它既考虑了最终的游戏结果，又通过进度奖励引导智能体向正确的方向移动。不过在后续的训练过程中，我们发现进度奖励有时候会带来一些问题，这个我们在强化学习部分详细讨论。

2.3 对手策略分析

项目提供了两个基线对手：Greedy 策略和预训练的 RL Baseline。

Greedy 策略的逻辑比较简单，它会评估每个合法动作的即时收益，选择得分最高的动作。具体来说，Greedy 策略偏好向目标方向移动（DownLeft 和 DownRight 方向得分最高），同时给予跳跃动作额外的奖励。这种策略虽然简单，但在很多情况下表现还不错，因为它能够有效地利用跳跃来快速推进棋子。

RL Baseline 是一个使用 PPO 算法预训练的强化学习模型。通过测试，我们发现这个基线模型相当强大，它能够学习到一些 Greedy 策略无法捕捉的复杂模式，比如为后续的连跳创造条件、阻挡对手的前进路线等。要击败这个基线，我们的智能体需要学习到更高层次的策略。

3 Minimax 算法实现

3.1 算法设计思路

Minimax 算法是博弈论中的经典算法，其核心思想是假设对手会做出对我方最不利的选择，在这种最坏情况下寻找最优策略。对于中国跳棋这样的零和博弈游戏，Minimax 算法理论上可以找到最优解，但由于状态空间巨大，实际应用中必须结合剪枝和启发式评估函数。

在设计 Minimax 智能体时，我们面临的第一个问题是如何定义评估函数。一个好的评估函数应该能够准确估计当前局面的优劣，引导搜索向正确的方向进行。经过反复思考和实验，我们决定采用基于移动方向和跳跃的启发式评估函数。

这个设计背后的直觉是：在中国跳棋中，最重要的目标是尽快将棋子移动到对角的目标区域。因此，向目标方向移动的动作应该获得正向评分，而远离目标的动作应该获得负向评分。同时，跳跃动作能够一次移动两格，效率更高，应该获得额外奖励。

3.2 评估函数设计

我们的评估函数主要考虑以下几个因素：

对于移动方向，我们给 DownLeft 和 DownRight 方向（向目标区域移动）赋予最高分数 25 分，Left 和 Right 方向（横向移动）赋予较低的 8 分，而 UpLeft 和 UpRight 方向（远离目标）则扣除 20 分。这个权重设置反映了向目标前进的重要性。

对于跳跃动作，我们额外奖励 35 分。如果是向下方向的跳跃，再额外奖励 20 分。这样设计的原因是跳跃动作不仅移动距离更远，而且可以继续进行连跳，具有很高的战略价值。对于 END_TURN 动作，情况比较特殊。如果当前正在跳跃链中且还有其他合法动作可选，过早结束回合是很差的选择，我们给予 -100 分的惩罚。否则，END_TURN 获得 -5 分的轻微惩罚，因为通常继续移动比结束回合更好。

评估函数的代码实现如下：

```
1 def _evaluate_move(self, move, has_jump, num_legal_moves):
2     if move == Move.END_TURN:
3         if has_jump and num_legal_moves > 1:
4             return -100
5         return -5
6
7     score = 0.0
8
9     # 方向评估
10    if move.direction == Direction.DownLeft:
11        score += 25
12    elif move.direction == Direction.DownRight:
13        score += 25
14    elif move.direction == Direction.Left:
15        score += 8
16    elif move.direction == Direction.Right:
17        score += 8
18    elif move.direction == Direction.UpLeft:
19        score -= 20
20    elif move.direction == Direction.UpRight:
21        score -= 20
22
23     # 跳跃奖励
24    if move.is_jump:
25        score += 35
26        if move.direction in [Direction.DownLeft, Direction.DownRight]:
```

```
27     score += 20
28
29     return score
```

3.3 Alpha-Beta 剪枝实现

纯粹的 Minimax 算法时间复杂度是指数级的，对于中国跳棋这样的游戏来说根本无法在合理时间内完成搜索。Alpha-Beta 剪枝是一种经典的优化技术，它通过维护两个边界值（alpha 和 beta）来剪掉不可能影响最终决策的分支，从而大大减少搜索的节点数。

Alpha-Beta 剪枝的核心思想是：在 Max 节点，如果当前找到的值已经大于等于 beta（父节点的最小可接受值），那么后续的搜索都不可能改变父节点的决策，可以直接剪掉。类似地，在 Min 节点，如果当前找到的值已经小于等于 alpha，也可以剪枝。

我们的实现中，搜索深度设置为 3 层。说实话，这个深度并不算深，但考虑到中国跳棋每个状态可能有几十个合法动作，更深的搜索会导致响应时间过长。在实际测试中，3 层深度配合 Alpha-Beta 剪枝已经能够达到不错的效果。

```
1 def _minimax(self, legal_indices, obs, depth, alpha, beta,
2               is_maximizing):
3     has_jump = self._has_jump_in_progress(obs)
4     num_legal = len(legal_indices)
5
6     if depth == 0 or num_legal == 0:
7         return 0, legal_indices[0] if num_legal > 0 else self.
8             action_space_dim - 1
9
10    best_action = legal_indices[0]
11
12    if is_maximizing:
13        max_eval = float('-inf')
14        for action_idx in legal_indices:
15            move = action_to_move(action_idx, self.n)
16            eval_score = self._evaluate_move(move, has_jump,
17                                             num_legal)
18
19            if eval_score > max_eval:
20                max_eval = eval_score
21                best_action = action_idx
22
23    alpha = max(alpha, eval_score)
```

```

21         if beta <= alpha:
22             break # Beta剪枝
23
24     return max_eval, best_action
25 else:
26     min_eval = float('inf')
27     for action_idx in legal_indices:
28         move = action_to_move(action_idx, self.n)
29         eval_score = self._evaluate_move(move, has_jump,
30                                         num_legal)
31
32         if eval_score < min_eval:
33             min_eval = eval_score
34             best_action = action_idx
35
36         beta = min(beta, eval_score)
37         if beta <= alpha:
38             break # Alpha剪枝
39
40     return min_eval, best_action

```

3.4 Minimax 算法的局限性思考

在实现和测试 Minimax 算法的过程中，我们逐渐意识到这种方法在中国跳棋中存在一些固有的局限性。

首先是搜索深度的限制。由于每个状态的分支因子很大（可能有几十个合法动作），即使使用了 Alpha-Beta 剪枝，搜索深度也很难超过 3-4 层。这意味着 Minimax 只能“看到”未来几步的局面，对于需要长期规划的策略（比如为多步之后的连跳创造条件）就无能为力了。

其次是评估函数的局限性。我们的评估函数只考虑了单个动作的即时效果，没有考虑整体棋盘布局。比如，有时候一个看起来“后退”的移动，实际上是为了给其他棋子让路或者创造跳跃机会。这种全局性的考量很难用简单的启发式规则来表达。最后是对手建模的问题。Minimax 假设对手会做出完美理性的决策，但实际的对手（无论是 Greedy 还是 RL Baseline）并不一定遵循这个假设。如果能够针对特定对手的弱点进行针对性优化，可能会取得更好的效果。

这些局限性也是我们决定尝试强化学习方法的原因之一。强化学习可以通过大量的自我对弈来学习复杂的策略，不需要人工设计评估函数，也能够适应不同的对手风格。

4 强化学习智能体设计与实现

4.1 方法选择与初步尝试

在强化学习方法的选择上，我们最终选择了 PPO (Proximal Policy Optimization) 算法。选择 PPO 的原因有几个：首先，PPO 是目前最流行的策略梯度算法之一，在各种任务上都表现稳定；其次，项目提供的 RL Baseline 就是用 PPO 训练的，使用相同的算法框架可以更好地进行比较；最后，Ray RLlib 库对 PPO 有很好的支持，可以方便地进行分布式训练。

但是，直接使用标准的 PPO 训练效果并不理想。我们最初的尝试是使用 RLlib 的多智能体训练框架，让两个 PPO 智能体进行自我对弈。这个方法在理论上应该能够让智能体通过博弈不断提升，但实际训练过程中遇到了很多问题。

最大的问题是训练不稳定。自我对弈训练中，两个智能体同时在学习和变化，导致训练目标不断漂移。有时候训练看起来在进步，但过一段时间又会退化。我们尝试了调整学习率、增加熵正则化等方法，但效果都不太理想。另一个问题是评估困难。在自我对弈的设置下，很难判断智能体是真的变强了还是只是在某种“局部最优”的策略上打转。我需要定期让训练中的智能体与固定的基线对手对弈来评估进展，但这增加了训练的复杂度。

4.2 对抗训练框架设计

经过多次失败的尝试，我们决定换一种思路：不使用自我对弈，而是让智能体与固定的对手进行对抗训练。这种方法的好处是训练目标明确稳定，容易监控进展。

具体来说，我们设计了一个三阶段的训练流程：

第一阶段是对抗 Random 对手。这个阶段的目标是让智能体学会基本的游戏规则和移动策略。Random 对手会随机选择合法动作，非常容易击败，但这个阶段可以帮助智能体快速学习到“向目标方向移动”这个基本策略。

第二阶段是对抗 Greedy 对手。当智能体能够稳定击败 Random 对手后（胜率达到 90% 以上），我将对手切换为 Greedy 策略。Greedy 策略比 Random 强很多，它会有意识地向目标方向移动并利用跳跃。击败 Greedy 需要智能体学习更精细的策略，比如如何创造连跳机会、如何阻挡对手等。

第三阶段是对抗 RL Baseline。这是最终目标，也是最困难的阶段。RL Baseline 是一个训练好的强化学习模型，具有相当高的水平。要击败它，智能体需要学习到一些“超越人类直觉”的策略。

为了实现这个训练框架，我们编写了一个单智能体环境包装器 SingleAgentVsOpponent。这个包装器将原本的多智能体环境转换为单智能体环境，我们的智能体进行学习，对手则使用固定的策略。

```
1 class SingleAgentVsOpponent(gym.Env):
```

```

2     """单agent环境包装器：agent对抗固定对手"""
3     def __init__(self, triangle_size=2, max_iters=200, opponent_type=
4         'greedy'):
5         super().__init__()
6         self.triangle_size = triangle_size
7         self.max_iters = max_iters
8         self.opponent_type = opponent_type
9
9     # 初始化对手策略
10    if opponent_type == 'random':
11        self.opponent = None # 随机对手
12    elif opponent_type == 'greedy':
13        self.opponent = GreedyPolicy(triangle_size)
14    elif opponent_type == 'rl_baseline':
15        self.opponent = Policy.from_checkpoint("pretrained/
16            policies/default_policy")
16
17    # 定义观测空间和动作空间
18    self.observation_space = GymDict({
19        "observation": Box(...),
20        "action_mask": Box(...),
21    })
22    self.action_space = Discrete(...)

```

4.3 动作掩码处理

中国跳棋的一个重要特点是，在任何给定状态下，只有一部分动作是合法的。如果智能体选择了非法动作，游戏会报错或者产生未定义行为。因此，正确处理动作掩码（action mask）是训练成功的关键。

RLLib 提供了一个专门用于处理动作掩码的 RLModule 类——TorchActionMaskRLM。这个模块会在策略网络输出动作概率之前，将非法动作的概率设置为零，确保智能体永远不会选择非法动作。

使用动作掩码的配置如下：

```

1 from ChineseChecker.models.action_masking_rlm import
2     TorchActionMaskRLM
3
3 rlm_class = TorchActionMaskRLM
4 model_config = {"fcnet_hiddens": [256, 128]}
5 rlm_spec = SingleAgentRLModuleSpec(

```

```

6     module_class=rlm_class,
7     model_config_dict=model_config
8 )
9
10 config = (
11     PPOConfig()
12     .rl_module(rl_module_spec=rlm_spec)
13     ...
14 )

```

在实现过程中，我遇到了一个棘手的问题：观察空间的类型不匹配。原始环境返回的观察是 PettingZoo 格式的字典，但 RLlib 期望的是 Gymnasium 格式的字典。这两种字典类型虽然看起来相似，但在内部实现上有所不同，直接使用会导致各种奇怪的错误。

解决这个问题花了我相当长的时间。最终的解决方案是在环境包装器中显式地将观察空间定义为 gymnasium.spaces.Dict 类型，并确保返回的观察与这个空间定义完全匹配。

4.4 奖励函数优化

在最初的训练尝试中，我直接使用了环境原本的奖励函数。但很快我发现了一个问题：训练显示奖励在不断增加，但实际对弈胜率却停滞不前甚至下降。

经过调试，我发现问题出在进度奖励上。环境的进度奖励计算公式会产生很大的数值，一局游戏下来可能累积到几万分。相比之下，胜负奖励只有正负 100 分，在总奖励中几乎可以忽略不计。这导致智能体“学会”了一种奇怪的策略：不断地来回移动棋子，累积进度奖励，而不关心是否能赢得游戏。

为了解决这个问题，我大幅简化了奖励函数，只保留胜负奖励和超时惩罚。在环境的 step 方法中，奖励计算逻辑如下：

```

1 def step(self, action):
2     my_agent = self.env.possible_agents[0] # player_0
3
4     # 学习 agent 走一步
5     self.env.step(int(action))
6     obs, env_reward, terminated, truncated, info = self.env.last()
7
8     done = terminated or truncated
9     reward = 0 # 默认没有奖励
10
11    # 如果游戏结束，检查谁赢了
12    if done:

```

```

13     winner = self.env.unwrapped.winner
14     if winner == my_agent:
15         reward = 100    # 赢了
16     elif winner is None:
17         reward = -10   # 平局/超时
18     else:
19         reward = -100  # 输了
20     ...

```

这个简化的奖励函数虽然信息量较少，但训练信号更加清晰，智能体能够更好地学习到“赢得游戏”这个核心目标。

4.5 从预训练模型开始训练

在完成基本的训练框架后，我开始了正式的训练。一个重要的发现是：从预训练的 RL Baseline 开始继续训练，效果远好于从随机初始化开始。

这背后的逻辑是：RL Baseline 已经学会了基本的游戏策略，从它开始可以避免智能体在早期阶段学习那些显而易见的规则（比如要向目标方向移动）。这样，训练可以更快地进入“精细化”阶段，学习如何击败特定对手。

但是，从预训练模型开始训练也带来了新的挑战。最大的问题是权重同步。RLlib 的新版本引入了 Learner API，训练过程中有两套权重：一套在 Worker 中用于采样，另一套在 Learner 中用于梯度更新。如果只设置了 Worker 的权重而没有同步到 Learner，那么第一次训练迭代就会用 Learner 的随机权重覆盖掉精心加载的预训练权重。

这个 bug 让我困惑了很长时间。表面上看，权重加载是成功的（验证时显示 100% 胜率），但一次训练迭代后性能就崩溃了（变成 0% 胜率）。通过打印权重变化的调试信息，我才发现问题所在：

```

1 [权重变化诊断] mean_diff=0.047616, max_diff=0.909724
2 原始权重范围: [-0.8719, 0.8606]
3 训练后权重范围: [-0.0558, 0.0558]

```

可以看到，训练后的权重范围完全变了，说明被重新初始化了。解决方案是同时向 Worker 和 Learner 设置权重：

```

1 # 设置Worker的权重
2 current_policy = algo.get_policy("default_policy")
3 current_policy.set_weights(restored_weights)
4 algo.workers.local_worker().set_weights({"default_policy":
5     restored_weights})
6 algo.workers.foreach_worker(set_weights_fn, local_worker=False)

```

```

7 # 关键：同步到Learner模块
8 if hasattr(algo, 'learner_group') and algo.learner_group is not None:
9     rl_module = current_policy.model
10    learner_weights = {"default_policy": rl_module.state_dict()}
11    algo.learner_group.set_weights(learner_weights)

```

4.6 超参数调优

在解决了权重同步问题后，训练终于能够正常进行了。但是，直接使用默认的超参数效果并不好。我花了相当多的时间进行超参数调优，下面分享一些关键的发现。

学习率是最敏感的参数之一。由于我是从预训练模型开始训练，太高的学习率会导致“灾难性遗忘”——智能体很快就会忘记之前学到的策略。经过实验，我发现 1×10^{-5} 是一个比较好的学习率，既能保持预训练的知识，又能逐渐适应新的对手。

PPO 的 clip 参数控制策略更新的幅度。默认值是 0.2，但对于从预训练模型继续训练的场景，这个值太大了。我将它降低到 0.1，使得每次更新更加保守，减少了训练的不稳定性。

批量大小和 SGD 迭代次数也需要仔细调整。太大的批量会使训练变慢，太小的批量会使梯度估计不准确。最终我使用了 2048 的训练批量大小和 256 的 minibatch 大小，每个批量进行 3 次 SGD 迭代。

最终的训练配置如下：

```

1 config = (
2     PPOConfig()
3     .training(
4         train_batch_size=2048,
5         lr=1e-5,
6         gamma=0.995,
7         lambda_=0.95,
8         use_gae=True,
9         clip_param=0.1,
10        grad_clip=0.5,
11        vf_loss_coeff=0.5,
12        sgd_minibatch_size=256,
13        num_sgd_iter=3,
14        entropy_coeff=0.005,
15    )
16    ...
17 )

```

4.7 训练过程与结果

使用上述配置，我在服务器上进行了训练。服务器配置为 NVIDIA RTX 3090 GPU，使用 12 个并行 Worker 进行环境采样。

训练过程比我预期的要顺利。由于从预训练模型开始，第一阶段（对抗 Random）几乎立即达标。第二阶段（对抗 Greedy）也很快完成，大约 20 个训练迭代后胜率就稳定在 100%。最具挑战性的是第三阶段（对抗 RL Baseline），胜率从初始的 50% 左右逐步提升。

以下是训练日志的摘录：

```
1 [阶段 1] Iter 0: reward=87.5, vs_Random=100%, vs_Greedy=100%, vs_RL  
2 =50%  
3 -> 新最佳 vs Greedy: 100%  
4 -> 新最佳 vs RL: 50%  
5 ======  
6 阶段 1 完成！ vs Greedy 达到 100%  
7 现在切换到阶段 2：对抗 RL Baseline (目标：90%+)  
8 ======  
9 [阶段 2] Iter 20: reward=42.0, vs_Random=100%, vs_Greedy=100%, vs_RL  
10 =70%  
11 [阶段 2] Iter 40: reward=46.0, vs_Random=100%, vs_Greedy=100%, vs_RL  
12 =70%  
13 [阶段 2] Iter 60: reward=56.0, vs_Random=100%, vs_Greedy=100%, vs_RL  
14 =70%  
15 [阶段 2] Iter 80: reward=36.0, vs_Random=100%, vs_Greedy=100%, vs_RL  
16 =80%  
17 -> 新最佳 vs RL: 80%  
18 [阶段 2] Iter 100: reward=56.0, vs_Random=100%, vs_Greedy=100%, vs_RL  
19 =80%  
20 [阶段 2] Iter 120: reward=62.0, vs_Random=100%, vs_Greedy=100%, vs_RL  
21 =90%  
22 -> 新最佳 vs RL: 90%  
23 ======  
24 训练完成！ vs Greedy=100%, vs RL=90%  
25 ======
```

可以看到，经过大约 120 个训练迭代，智能体对 RL Baseline 的胜率从 50% 提升到了 90%，同时保持了对 Greedy 的 100% 胜率。整个训练过程大约耗时 30 分钟。

5 实验结果与分析

5.1 测试方法

为了全面评估两种智能体的性能，我编写了测试脚本进行系统性的评估。每种配置运行 20 局对弈，统计胜率。使用不同的随机种子初始化每局游戏，确保结果的可靠性。

测试使用的命令如下：

```
1 # 测试 Minimax 智能体
2 python play.py --triangle_size 2
3
4 # 测试 RL 智能体
5 python play.py --triangle_size 2 --use_rl --checkpoint logs/...
   best_vs_rl
```

5.2 Minimax 智能体测试结果

Minimax 智能体的测试结果如下：

对抗 Greedy 策略：胜率 100%（20 胜 0 负）

对抗 RL Baseline：胜率 20%（4 胜 16 负）

这个结果在意料之中。Minimax 算法的启发式评估函数与 Greedy 策略的评估逻辑相似，但 Minimax 有更深的搜索深度和 Alpha-Beta 剪枝的优化，因此能够稳定击败 Greedy。

但对于 RL Baseline，Minimax 的表现就没那么好了。RL Baseline 能够学习到一些 Minimax 难以捕捉的模式，比如在关键位置布局以阻挡对手、为多步之后的连跳做铺垫等。这些长期策略超出了 Minimax 的搜索范围，导致 Minimax 经常在中后期陷入被动。

5.3 RL 智能体测试结果

RL 智能体我进行了两次独立测试，结果如下：

第一次测试：对抗 Greedy 胜率 95%，对抗 RL Baseline 胜率 95%

第二次测试：对抗 Greedy 胜率 100%，对抗 RL Baseline 胜率 90%

RL 智能体的表现明显优于 Minimax，特别是在对抗 RL Baseline 时。这说明通过对抗训练，智能体确实学习到了一些有效的策略来击败基线模型。

不过，我也注意到 RL 智能体的胜率存在一定波动。这可能是因为强化学习模型的决策带有一定的随机性（熵正则化的影响），也可能是因为某些特定的开局配置对智能体更有利或更不利。增加测试局数可以得到更稳定的胜率估计，但 20 局已经足以说明整体趋势。

5.4 测试截图

以下是测试过程的截图证据：

```
(final) (ml4co_venv) zhanghang@thinklab-105-223:/mnt/nas-new/home/zhanghang/zhangxie/AI2603-FinalProject/code$ python play.py --triangle_size 2
/home/zhanghang/anaconda3/envs/final/lib/python3.10/site-packages/ray_private/parameter.py:4: UserWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html. The pkg_resources package is slated for removal as early as 2025-11-30. Refrain from using this package or pin to Setuptools<81.
  import pkg_resources
Gym has been unmaintained since 2022 and does not support NumPy 2.0 amongst other critical functionality.
Please upgrade to Gymnasium, the maintained drop-in replacement of Gym, or contact the authors of your software and request that they upgrade.
Users of this version of Gym should be able to simply replace 'import gym' with 'import gymnasium as gym' in the vast majority of cases.
See the migration guide at https://gymnasium.farama.org/introduction/migration_guide/ for additional information.
2026-01-04 17:13:49,548 WARNING deprecation.py:50 -- DeprecationWarning: 'DirectStepOptimizer' has been deprecated. This will raise an error in the future!
2026-01-04 17:13:51,137 WARNING deprecation.py:50 -- DeprecationWarning: 'LearningRateSchedule' has been deprecated. This will raise an error in the future!
2026-01-04 17:13:51,137 WARNING deprecation.py:50 -- DeprecationWarning: 'EntropyCoeffSchedule' has been deprecated. This will raise an error in the future!
2026-01-04 17:13:51,137 WARNING deprecation.py:50 -- DeprecationWarning: 'KLCoefMixin' has been deprecated. This will raise an error in the future!
Play with Greedy
100%|██████████| 20/20 [00:04<00:00,  4.33it/s]
Winrate: 1.0
Play with RL Baseline
100%|██████████| 20/20 [00:05<00:00,  3.47it/s]
Winrate: 0.2
```

图 1: Minimax 智能体测试结果： vs Greedy 100%， vs RL Baseline 20%

```
(final) (ml4co_venv) zhanghang@thinklab-105-223:/mnt/nas-new/home/zhanghang/zhangxie/AI2603-FinalProject/code$ python play.py --use_rl --checkpoint_logs/three_stage_2025-12-31_19-49-51/checkpoint_350
/home/zhanghang/anaconda3/envs/final/lib/python3.10/site-packages/ray_private/parameter.py:4: UserWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html. The pkg_resources package is slated for removal as early as 2025-11-30. Refrain from using this package or pin to Setuptools<81.
  import pkg_resources
Gym has been unmaintained since 2022 and does not support NumPy 2.0 amongst other critical functionality.
Please upgrade to Gymnasium, the maintained drop-in replacement of Gym, or contact the authors of your software and request that they upgrade.
Users of this version of Gym should be able to simply replace 'import gym' with 'import gymnasium as gym' in the vast majority of cases.
See the migration guide at https://gymnasium.farama.org/introduction/migration_guide/ for additional information.
2026-01-04 17:16:34,441 WARNING deprecation.py:50 -- DeprecationWarning: 'DirectStepOptimizer' has been deprecated. This will raise an error in the future!
2026-01-04 17:16:36,688 WARNING algorithm_config.py:2578 -- Setting 'exploration_config()' because you set `enable_rl_module_api=True` . When RLModule API are enabled, exploration_config can not be set. If you want to implement custom exploration behaviour, please modify the 'forward_exploration' method of the RLModule at hand. On configs that have a default exploration config, this must be done with 'config.exploration_config()' .
2026-01-04 17:16:36,762 WARNING deprecation.py:50 -- DeprecationWarning: 'ValueNetworkMixin' has been deprecated. This will raise an error in the future!
2026-01-04 17:16:36,762 WARNING deprecation.py:50 -- DeprecationWarning: 'LearningRateSchedule' has been deprecated. This will raise an error in the future!
2026-01-04 17:16:36,762 WARNING deprecation.py:50 -- DeprecationWarning: 'EntropyCoeffSchedule' has been deprecated. This will raise an error in the future!
2026-01-04 17:16:36,762 WARNING deprecation.py:50 -- DeprecationWarning: 'KLCoefMixin' has been deprecated. This will raise an error in the future!
2026-01-04 17:16:36,845 WARNING algorithm_config.py:2578 -- Setting 'exploration_config()' because you set `enable_rl_module_api=True` . When RLModule API are enabled, exploration_config can not be set. If you want to implement custom exploration behaviour, please modify the 'forward_exploration' method of the RLModule at hand. On configs that have a default exploration config, this must be done with 'config.exploration_config()' .
Play with Greedy
100%|██████████| 20/20 [00:06<00:00,  3.18it/s]
Winrate: 0.95
Play with RL Baseline
100%|██████████| 20/20 [00:07<00:00,  2.85it/s]
Winrate: 0.95
```

图 2: RL 智能体第一次测试： vs Greedy 95%， vs RL Baseline 95%

```
(final) (ml4co_venv) zhanghang@thinklab-105-223:/mnt/nas-new/home/zhanghang/zhangxie/AI2603-FinalProject/code$ python play.py --use_rl --checkpoint_logs/three_stage_2025-12-31_19-49-51/best_vs_rl
/home/zhanghang/anaconda3/envs/final/lib/python3.10/site-packages/ray_private/parameter.py:4: UserWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html. The pkg_resources package is slated for removal as early as 2025-11-30. Refrain from using this package or pin to Setuptools<81.
  import pkg_resources
Gym has been unmaintained since 2022 and does not support NumPy 2.0 amongst other critical functionality.
Please upgrade to Gymnasium, the maintained drop-in replacement of Gym, or contact the authors of your software and request that they upgrade.
Users of this version of Gym should be able to simply replace 'import gym' with 'import gymnasium as gym' in the vast majority of cases.
See the migration guide at https://gymnasium.farama.org/introduction/migration_guide/ for additional information.
2026-01-04 17:17:20,661 WARNING deprecation.py:50 -- DeprecationWarning: 'DirectStepOptimizer' has been deprecated. This will raise an error in the future!
2026-01-04 17:17:22,493 WARNING algorithm_config.py:2578 -- Setting 'exploration_config()' because you set `enable_rl_module_api=True` . When RLModule API are enabled, exploration_config can not be set. If you want to implement custom exploration behaviour, please modify the 'forward_exploration' method of the RLModule at hand. On configs that have a default exploration config, this must be done with 'config.exploration_config()' .
2026-01-04 17:17:22,609 WARNING deprecation.py:50 -- DeprecationWarning: 'ValueNetworkMixin' has been deprecated. This will raise an error in the future!
2026-01-04 17:17:22,610 WARNING deprecation.py:50 -- DeprecationWarning: 'LearningRateSchedule' has been deprecated. This will raise an error in the future!
2026-01-04 17:17:22,610 WARNING deprecation.py:50 -- DeprecationWarning: 'EntropyCoeffSchedule' has been deprecated. This will raise an error in the future!
2026-01-04 17:17:22,612 WARNING deprecation.py:50 -- DeprecationWarning: 'KLCoefMixin' has been deprecated. This will raise an error in the future!
2026-01-04 17:17:22,632 WARNING algorithm_config.py:2578 -- Setting 'exploration_config()' because you set `enable_rl_module_api=True` . When RLModule API are enabled, exploration_config can not be set. If you want to implement custom exploration behaviour, please modify the 'forward_exploration' method of the RLModule at hand. On configs that have a default exploration config, this must be done with 'config.exploration_config()' .
Play with Greedy
100%|██████████| 20/20 [00:04<00:00,  4.69it/s]
Winrate: 1.0
Play with RL Baseline
100%|██████████| 20/20 [00:04<00:00,  4.58it/s]
```

图 3: RL 智能体第二次测试： vs Greedy 100%， vs RL Baseline 90%

5.5 结果对比与分析

表 1: 两种智能体性能对比

智能体	vs Greedy	vs RL Baseline	平均胜率
Minimax (Alpha-Beta)	100%	20%	60%
RL (PPO 训练)	95-100%	90-95%	92.5-97.5%

从结果可以看出，RL 智能体在整体性能上显著优于 Minimax 智能体。特别是在对抗 RL Baseline 时，RL 智能体的优势非常明显（90-95% vs 20%）。

这个结果反映了两种方法的本质区别。Minimax 是一种“model-based”方法，需要人工设计评估函数，其性能上限取决于评估函数的质量。而 RL 是一种“model-free”方法，可以通过大量的自我博弈来学习复杂的策略，不需要人工设计评估函数。

当然，RL 方法也有其缺点。首先是训练成本高，需要大量的计算资源和时间。其次是可解释性差，很难理解 RL 智能体为什么做出某个决策。相比之下，Minimax 的决策过程是完全透明的，可以追溯搜索树来理解每个选择的原因。

6 总结与展望

6.1 项目总结

通过这个项目，我们完成了以下工作：

实现了带 Alpha-Beta 剪枝的 Minimax 智能体。通过设计合理的启发式评估函数，该智能体能够 100% 击败 Greedy 策略，对 RL Baseline 也有 20% 的胜率。虽然这个胜率不算高，但考虑到 Minimax 方法的简单性和可解释性，这个结果还是可以接受的。

实现了基于 PPO 算法的强化学习智能体。通过设计三阶段对抗训练框架、解决权重同步问题、优化奖励函数和超参数，最终训练出的智能体能够达到 95% 以上的综合胜率。这个结果大大超出了我们的预期。在项目过程中，我们深入理解了 Minimax 算法、Alpha-Beta 剪枝、PPO 算法等经典方法，也学会了使用 Ray RLlib 进行分布式强化学习训练。更重要的是，我们体会到了“调试机器学习系统”的艰辛——很多时候问题不在于算法本身，而在于一些看似微不足道的实现细节（比如权重同步、奖励缩放等）。

6.2 遇到的主要困难

回顾整个项目，我们遇到的主要困难包括：

环境接口的理解。PettingZoo 环境的观察空间、动作空间、多智能体交互方式等都需要仔细阅读代码才能理解。特别是动作编码方式，花了我们不少时间才搞清楚。

RLlib 框架的使用。RLlib 功能强大但学习曲线陡峭，文档也不够完善。很多时候需要阅读源代码才能理解某个参数的真正含义。新版本的 Learner API 带来的权重同步问题更是让我们头疼了很久。超参数调优。强化学习对超参数非常敏感，找到一组好的超参数需要大量的实验。如果有更多时间，我们会尝试使用自动超参数搜索方法（如 Ray Tune）来优化这个过程。

6.3 未来改进方向

如果有更多时间，我们希望在以下方向进行改进：

对于 Minimax 智能体，可以尝试更复杂的评估函数，比如考虑棋子的位置分布、与对手的相对位置等。也可以尝试迭代加深搜索（Iterative Deepening）来在时间限制内尽可能深入搜索。

对于 RL 智能体，可以尝试其他算法如 SAC、PPG 等，看是否能取得更好的效果。也可以尝试自我对弈训练（Self-Play），虽然更难调，但理论上能够达到更高的水平。

另外，可以尝试将两种方法结合，比如用 RL 学习评估函数，然后用 Minimax 进行搜索。这种“混合方法”在 AlphaGo 等系统中已经被证明非常有效。

6.4 个人感想

这个项目让我们对强化学习有了更深的理解。之前学习 RL 时，总觉得这些算法很“神奇”，给定环境和奖励函数就能自动学出好的策略。但实际动手做了之后才发现，要让 RL 真正 work 需要大量的工程努力——设计合适的奖励函数、处理各种边界情况、调试神经网络等等。

同时，我们也体会到了传统搜索方法的价值。虽然 Minimax 在这个任务上的性能不如 RL，但它的可解释性和确定性是 RL 难以比拟的。在实际应用中，这些特性有时候比纯粹的性能更重要。

总的来说，这是一次非常有收获的项目经历。感谢助教提供的环境代码和基线模型，让我们能够专注于算法本身的实现和优化。

致谢

感谢课程组提供的项目框架和测试环境，感谢助教在项目过程中的答疑解惑。

A 代码结构

项目代码主要包含以下文件：

```
1 code/
2     agents.py          # 智能体实现 (Minimax、Greedy 等)
3     play.py            # 测试脚本
4     train_final.py    # RL训练脚本
5 ChinesehCheckers/   # 环境实现
6     env/
7         chineseh_checker_env.py
8         game.py
9         utils.py
10        models/
11            action_masking_rlm.py  # 动作掩码RLModule
```

```
12 pretrained/          # 预训练 RL Baseline  
13 logs/              # 训练日志和 checkpoints
```

B 运行说明

测试 Minimax 智能体:

```
1 python play.py --triangle_size 2
```

测试 RL 智能体:

```
1 python play.py --triangle_size 2 --use_rl --checkpoint <  
2   checkpoint_path>
```

训练 RL 智能体:

```
1 python train_final.py --triangle_size 2 --train_iters 500 \  
2   --num_workers 12 --start_from_pretrained
```

C 组员分工

张熙和:

- Minimax 智能体的设计与实现
- 强化学习智能体的训练脚本编写
- 报告撰写与整理
- 测试与结果分析

邵开阳:

- 环境接口的理解与调试
- RLlib 框架的使用与权重同步问题解决
- 超参数调优与训练优化
- 报告撰写与整理