DouZero: Mastering DouDizhu with Self-Play Deep Reinforcement Learning

摘要

游戏是对现实世界的抽象，其中人工智能代理学习与其他代理竞争和合作。尽管在各种完美和不完美信息的游戏中取得了显著的成就，但《斗地主》，这个三人玩的纸牌游戏，仍然是一个未解之谜。《斗地主》是一个非常具有挑战性的领域，涉及竞争、合作、不完美信息、庞大的状态空间，特别是一个庞大的可能行动集，其中合法行动在每一轮中都有很大的变化。然而，现代强化学习算法主要关注简单和小的动作空间，因此在《斗地主》中并未取得令人满意的进展。在这项工作中，本文提出了一个概念上简单但效果显著的《斗地主》人工智能系统，即 DouZero。DouZero 通过将传统的蒙特卡罗方法与深度神经网络、动作编码和并行执行器相结合，对其进行增强。从在单个服务器上的四个 GPU 上从零开始训练，DouZero 在几天内就超过了所有现有的《斗地主》人工智能程序，在 Botzone 排行榜中跻身第一，超过了 344 个 AI 代理。通过构建 DouZero，本文展示了经典的蒙特卡罗方法如何在具有复杂动作空间的艰难领域中取得强大的结果。而我也尝试自己去复现文中的关键算法，同时微调学习率和 bitch-size 并增加 actor 的数量，能够在更短的时间内达到与原文类似的效果。

关键词：蒙特卡洛；深度神经网络

1 引言

在人工智能领域，游戏被广泛应用作为评估算法性能的基准，同时也为人工智能系统提供了一个复杂而富有挑战性的学习环境。近年来，针对不同类型的游戏，研究人员已经取得了一系列显著的成果。然而，有些游戏，特别是那些涉及竞争、合作、不完美信息以及庞大状态空间和行动空间的游戏，仍然存在巨大的挑战。

《斗地主》是一种典型的三人纸牌游戏，具有竞争激烈、合作默契、信息不完备等特点。这使得该游戏成为研究者们关注的对象。然而，传统的强化学习算法主要集中在简单和小型的动作空间上，对于《斗地主》这类复杂游戏却表现不佳。在这种情况下，需要一种新的方法来解决这一具有挑战性的领域，以提高人工智能在复杂博弈中的表现。

选取《斗地主》作为研究对象的意义在于挑战传统强化学习算法在大型、复杂动作空间中的局限性。通过提出 DouZero [5] 这一新型人工智能系统，结合蒙特卡罗方法、深度神经网络以及并行计算等技术，成功在《斗地主》领域取得卓越成果。这不仅推动了强化学习算法在复杂游戏中的应用，也为解决更广泛的协作与竞争问题提供了新的思路。在技术上，DouZero

的成功还表明，经典算法与深度学习方法的结合可以在博弈领域取得更好的性能，这对未来人工智能系统的发展具有积极的指导意义。

2 相关工作

2.1 搜索不完美信息博弈

反事实遗憾最小化 (CFR) 是扑克游戏的领先迭代算法，有许多变体 [3]。然而，遍历斗地主的博弈树是计算密集型的，因为它是一棵巨大的树，具有很大的分支因子。此外，大多数先前的研究都集中在零和设置上。尽管已经做出了一些努力来解决合作环境问题，例如通过蓝图政策，但推理竞争和合作仍然具有挑战性。因此，斗地主还没有看到有效的类似 CFR 的解决方案。

2.2 不完美信息博弈的强化学习

最近的研究表明，强化学习（RL）可以在扑克游戏中实现竞争性表现 [1]。与 CFR 不同，强化学习基于采样，因此可以轻松推广到大型游戏。强化学习已成功应用于一些复杂的不完美信息游戏，例如星际争霸、DOTA 和麻将。最近，RL+ 搜索在扑克游戏中得到了探索并被证明是有效的。DeltaDou 采用了类似的思路，首先推断隐藏信息，然后使用 MCTS 将 RL 与斗地主中的搜索结合起来。然而，DeltaDou 的计算成本很高，并且严重依赖人类的专业知识。

3 本文方法

3.1 本文方法概述

本文提出了 DouZero，一个概念上简单但有效的斗地主人工智能系统，没有状态/动作空间或任何人类知识的抽象。DouZero 通过深度神经网络、动作编码和并行参与者增强了传统的蒙特卡洛方法。DouZero 有两个理想的特性。首先，与 DQN 不同，它不易受到高估偏差的影响。其次，通过将动作编码到卡片矩阵中，它可以自然地概括整个训练过程中不常见的动作。这两个属性对于处理斗地主庞大而复杂的动作空间至关重要。与许多树搜索算法不同，DouZero 基于采样，这使我们能够使用复杂的神经架构，并在给定相同的计算资源的情况下每秒生成更多的数据。与许多先前依赖于特定领域抽象的扑克人工智能研究不同，DouZero 不需要任何领域知识或底层动态知识。在只有 48 个核心和 4 个 1080Ti GPU 的单台服务器上从头开始训练，DouZero 在半天内超越了 CQN 和启发式规则，在两天内击败了他们的内部监督代理，并在十天内超越了 DeltaDou。大量评测表明，斗零是迄今为止最强的斗地主人工智能系统。

3.2 加强蒙特卡罗算法

标准的蒙特卡罗算法只能处理离散的情况 [4]，但斗地主的状态和动作空间都非常大，普通的蒙特卡罗算法不能直接用。这里给它做些加强来应对斗地主：

1. 把 Q 表格换成神经网络，称作 Q 网络。
2. 用 Mean-Square-Error（MSE) 的损失来更新网络。

3. 对斗地主中的动作也进行编码（后面会详细介绍）。
4. 在采样中引入 epsilon-greedy 机制来鼓励探索。
5. 用多个进程来采样，提高效率。

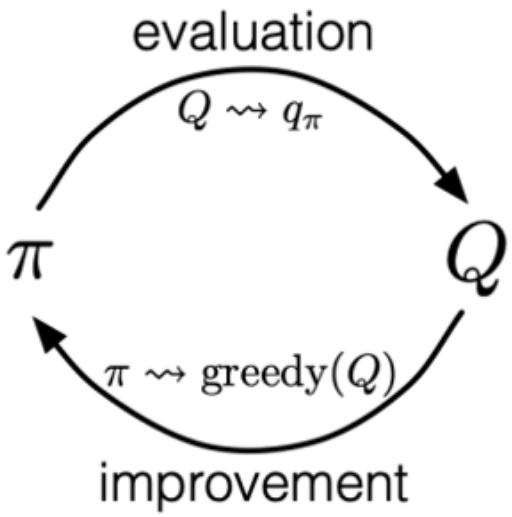

图 1. MC

3.3 牌型编码

将牌型编码成 15x4 的矩阵，其中 15 表示非重复牌的种类（3 到 A 加上大小王），4 表示最多每种有四张牌。我们用 0/1 的矩阵来编码，例子如下：

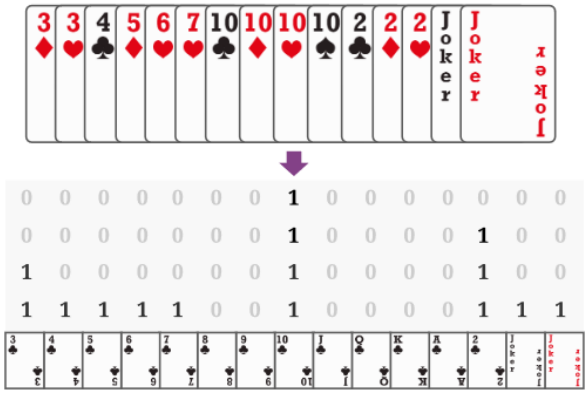

图 2. Card Type Code

3.4 神经网络

网络的输入是状态和动作，输出是 Q 值。动作就是简单地用上面的方式进行编码。状态包括两部分：一部分是当前能看到的信息，包括手牌、其他玩家出的牌、上家的牌等特征矩阵以及其他玩家手牌数量和炸弹数量的 0/1 编码；另一部分是历史出牌信息，我们用 LSTM 网络进行编码。最后特征经过 6 层全连接网路得到 Q 值。

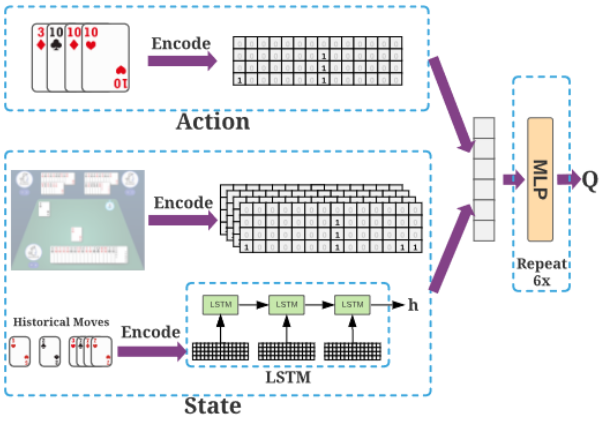

图 3. Q-network

3.5 并行 actor

对于 actor 的并行化。作者将地主表示为 L，在地主前行动的农民记为 U，在地主后行动的农民记为 D。

在 DouZero 算法中的 Actor 进程，其主要任务是在每次迭代中生成游戏 episode，并将观察到的状态、选择的动作以及相应的奖励存储到本地缓冲区。在每个时间步，Actor 根据本地 Q 网络选择动作，引入一定的探索以促进学习者的策略探索。随后，Actor 将一定数量的经验传输到共享缓冲区，以便学习者进程可以使用这些经验进行训练。关键参数包括共享缓冲区的 BL, BU, BD，本地 Q 网络 QL, QU, QD，以及探索参数 和折扣因子 。在奖励计算方面，每个时间步都会更新奖励 rt 为累计奖励，考虑了未来奖励的影响，这通过折扣因子实现。整体而言，Actor 进程通过不断与学习者进程交互，推动了算法的学习过程。

Algorithm 1 Actor Process of DouZero

Input: Shared buffers $\mathcal{B}_L$, $\mathcal{B}_U$ and $\mathcal{B}_D$ with B entries and size S for each entry, exploration hyperparameter ϵ, discount factor γ
Initialize local Q-networks Q_{L}, Q_{U} and Q_{D}, and local buffers $\mathcal{D}_L$, $\mathcal{D}_U$ and $\mathcal{D}_D$
for iteration = 1, 2, ... **do**
  Synchronize Q_{L}, Q_{U} and Q_{D} with the learner process
  for t = 1, 2, ... T **do** ▷ *Generate an episode*
    $Q \leftarrow$ one of $Q_{\mathrm{L}}, Q_{\mathrm{U}}, Q_{\mathrm{D}}$ based on position
    $a_t \leftarrow \begin{cases} \arg\max_a Q(s_t, a) & \text{with prob } (1-\epsilon) \\ \text{random action} & \text{with prob } \epsilon \end{cases}$
    Perform a_t, observe s_{t+1} and reward r_t
    Store $\{s_t, a_t, r_t\}$ to $\mathcal{D}_L$, $\mathcal{D}_U$, or $\mathcal{D}_D$ accordingly
  end for
  for t = T-1, T-2, ... 1 **do** ▷ *Obtain cumulative reward*
    $r_t \leftarrow r_t + \gamma r_{t+1}$ and update r_t in $\mathcal{D}_L$, $\mathcal{D}_U$, or $\mathcal{D}_D$
  end for
  for $p \in \{L, U, D\}$ **do** ▷ *Optimized by multi-thread*
    if $\mathcal{D}_p$.length $\geq L$ **then**
      Request and wait for an empty entry in $\mathcal{B}_p$
      Move $\{s_t, a_t, r_t\}$ of size L from $\mathcal{D}_p$ to $\mathcal{B}_p$
    end if
  end for
end for

图 4. Actor Process of DouZero

3.6 学习过程

learner 为这三个位置维护三个全局 Q 网络，并使用 actor 过程提供的数据，计算 MSE 损失来更新网络参数来近似目标值。每个 actor 维护了三个本地的 Q 网络，它们定期与全局 Q 网络进行同步。actor 将从游戏引擎中反复采样轨迹，并计算每个状态-动作对的累积奖励。learner 与 actor 的信息交流通过三个共享的缓冲区实现。

Algorithm 2 Learner Process of DouZero

Input: Shared buffers $\mathcal{B}_L$, $\mathcal{B}_U$ and $\mathcal{B}_D$ with B entries and size S for each entry, batch size M, learning rate ψ
Initialize global Q-networks Q^g_{L}, Q^g_{U} and Q^g_{D}
for iteration = 1, 2, ... until convergence **do**
  for $p \in \{L, U, D\}$ **do** ▷ *Optimized by multi-thread*
    if the number of full entries in $\mathcal{B}_p \geq M$ **then**
      Sample a batch of $\{s_t, a_t, r_t\}$ with $M \times S$ instances from $\mathcal{B}_p$ and free the entries
      Update Q^g_{p} with MSE loss and learning rate ψ
    end if
  end for
end for

图 5. Learner Process of DouZero

4 复现细节

与已有开源代码对比

本次复现的论文有开源代码,但是我参照其核心算法尝试自己编写代码,主要实现了 train 和 learn 的方法的实现。由于论文中使用 48 个核心和 4 个 1080Ti GPU 整整训练了 30 天,而我使用 4 张 V100 GPU,增加了 actor 的数量,同时调整了 batch-size 在 20 多天就达到了与原文相似的结果。具体过程如下:

1. 模型定义和优化器创建: 使用 Model 类创建并初始化模型,调用 share-memory 方法使模型在多进程中共享内存,设定为评估模式。将模型保存在字典 models 中,其中键是设备标识符。

```
# 模型定义和优化器创建
models = {}
for device in device_iterator:
    model = Model(device=device)
    model.share_memory()
    model.eval()
    models[device] = model
```

2. 数据处理和损失计算: compute-loss 函数计算模型输出与目标之间的均方误差损失。在 learn 函数中,使用输入的训练数据计算损失,并使用反向传播和优化器更新模型参数。

```
# 数据处理和损失计算
def compute_loss(logits, targets):
    loss = ((logits.squeeze(-1) - targets)**2).mean()
    return loss
```

3. 学习函数: learn 函数接收演员模型、学习者模型、训练数据、优化器等作为输入。在函数内部,提取观察数据和目标,计算损失,执行反向传播并更新模型参数。还通过锁机制确保线程安全,以及同步演员和学习者之间的模型参数。

```
def learn(position, actor_models, model, batch, optimizer, flags, lock):
    with lock:
        learner_outputs = model(obs_z, obs_x, return_value=True)
        loss = compute_loss(learner_outputs['values'], target)
        stats = {
            'mean_episode_return_'+position: torch.mean(torch.stack([_r for
            'loss_'+position: loss.item(),
        }
        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), flags.max_grad_norm)
        optimizer.step()
```

```
        for actor_model in actor_models.values():
            actor_model.get_model(position).load_state_dict(model.state_dic
        return stats
```

4. 训练函数：函数初始化了模型、缓冲区、优化器，并启动了多个演员进程。通过多线程调用 batch-and-learn 函数，进行数据批处理和学习。还实现了定期保存模型和日志记录。

```
def train(flags):
    # 创建优化器
    optimizers = create_optimizers(flags, learner_model)
    # 数据处理和缓冲区管理
    for device in device_iterator:
        for m in range(flags.num_buffers):
            free_queue[device]['landlord'].put(m)
            free_queue[device]['landlord_up'].put(m)
            free_queue[device]['landlord_down'].put(m)
    # 多线程和多进程训练
    threads = []
    locks = {}
    for device in device_iterator:
        locks[device] = {'landlord': threading.Lock(), 'landlord_up': threa
    position_locks = {'landlord': threading.Lock(), 'landlord_up': threadin
    for device in device_iterator:
        for i in range(flags.num_threads):
            for position in ['landlord', 'landlord_up', 'landlord_down']:
                thread = threading.Thread(
                    target=batch_and_learn, name='batch-and-learn-%d' % i,
                thread.start()
                threads.append(thread)
```

4.1 实验环境搭建

```
创建并激活虚拟环境
conda create -n douzero python==3.7
conda activate douzero
安装相关依赖
cd douzero
pip3 install -r requirements.txt
安装稳定版本的Douzero
pip3 install douzero

1.训练模型
```

```
至少拥有一块可用的GPU，运行
python3 train.py

如果需要用多个GPU训练Douzero，使用以下参数：
—gpu_devices: 用作训练的GPU设备名
—num_actor_devices: 被用来进行模拟（如自我对弈）的GPU数量
—num_actors: 每个设备的演员进程数
—training_device: 用来进行模型训练的设备
如：
python3 train.py —gpu_devices 0,1,2,3 —num_actor_devices 3 —num_actors 1
在CPU上运行：
python3 train.py —actor_device_cpu —training_device cpu


2.评估
第1步：生成评估数据
python3 generate_eval_data.py
第2步：自我对弈
python3 evaluate.py

以下为一些重要的超参数。

—landlord: 扮演地主的智能体，可选值：random，rlcard或预训练模型的路径
—landlord_up: 扮演地主上家的智能体，可选值：random，rlcard或预训练模型的路
—landlord_down: 扮演地主下家的智能体，可选值：random，rlcard或预训练模型的
—eval_data: 包含评估数据的pickle文件
—num_workers: 用多少个进程进行模拟
—gpu_device: 用哪个GPU设备进行模拟。默认用CPU

例如，可以通过以下命令评估DouZero–ADP智能体作为地主对阵随机智能体

python3 evaluate.py —landlord baselines/douzero_ADP/landlord.ckpt —landlor
```

4.2 创新点

本次复现微调了其学习率和 batch-size 的大小，增加了用于模拟的 actor 数量，在 4 张 v100 上大大缩短了从训练到学习所需要的时间。同时将每次模型训练的效果保存在 excel 中，方便后续训练过程的可视化。

5 实验结果分析

5.1 参数设置

每个共享缓冲区有 B = 50 个条目，大小 S = 100，批量大小 M = 32，并且 = 0.01。我们设置折扣因子 = 1，因为斗地主在最后一个时间步只有非零奖励，早期的动作非常重要。我们使用 ReLU 作为 MLP 每层的激活函数。我们采用学习率 = 0.0001、平滑常数 0.99 和 = 10−5 的 RMSprop 优化器。

5.2 实验设置

扑克游戏中衡量策略强度的常用指标是可利用性。然而，在斗地主中，计算可利用性本身就很棘手，因为斗地主具有巨大的状态/动作空间，并且有三个玩家。为了评估表现，我们发起了包括地主和农民两个对手方的锦标赛。我们通过将每副牌玩两次来减少方差。具体来说，对于两个竞争算法 A 和 B，他们将首先分别扮演给定套牌的地主和农民位置。然后他们交换立场，即 A 占据农民位置，B 占据地主位置，并再次玩同一副牌。我们考虑以下几种算法：

• DeltaDou：一个强大的人工智能程序，使用贝叶斯方法推断隐藏信息并使用 MCTS 搜索棋步 [2]。我们使用作者提供的代码和预训练模型。该模型经过两个月的训练，结果显示其性能与顶级人类玩家相当。

• CQN：组合 Q-Learning 是一个基于卡片分解和深度 Q-Learning 的程序。我们使用作者提供的开源代码和预训练模型。

• SL：监督学习基线。我们在斗地主游戏手机应用程序中内部收集了来自联赛最高级别玩家的 226，230 场人类专家比赛。然后，我们使用与 DouZero 相同的状态表示和神经架构，用这些数据生成的 49，990，075 个样本来训练监督智能体。

• 基于规则的程序：我们收集了一些基于启发式的开源程序，包括 RHCP4、改进版本 RHCP-v25 以及 RLCard package6 中的规则模型。此外，我们考虑一个随机程序，对合法动作进行统一采样。

5.3 实验对比

我们在实践中发现这两个指标鼓励不同风格的策略。例如，如果使用 ADP 作为奖励，那么智能体对于打炸弹往往会非常谨慎，因为打炸弹是有风险的，可能会导致更大的 ADP 损失。相比之下，以 WP 为目标，即使会失败，代理也倾向于积极地使用炸弹，因为炸弹不会影响 WP。我们观察到，就 ADP 而言，使用 ADP 训练的智能体表现略好于使用 WP 训练的智能体，反之亦然。接下来，我们分别以 ADP 和 WP 为目标训练并报告两个 DouZero 智能体的结果。

WP (Winning Percentage) 代表了地主或农民阵营的胜率。算法 A 对算法 B 的 WP 指标大于 0.5 代表算法 A 强于算法 B。

图 4 上半图为论文中实验结果，DouZero 策略在所有基于规则的策略和监督学习中占主导地位，证明了在斗地主中采用强化学习的有效性。下半图为本次复现结果，可以看出，基本与论文中结果一致。

Rank	A \ B	DouZero		DeltaDou		SL		RHCP-v2		RHCP		RLCard		CQN		Random	
		L	P	L	P	L	P	L	P	L	P	L	P	L	P	L	P
1	DouZero	.4159	.5841	**.4870**	**.6843**	**.5692**	**.7494**	**.6844**	**.8303**	**.7253**	**.8033**	**.8695**	**.9089**	**.7686**	**.8513**	**.9858**	**.9920**
2	DeltaDou	.3166	.5130	.4120	.5880	**.5130**	**.7211**	**.6701**	**.8165**	**.7048**	**.7899**	**.8563**	**.8955**	**.7326**	**.8351**	**.9871**	**.9960**
3	SL	.2506	.4308	.2789	.5130	.4072	.5928	**.5370**	**.6857**	**.5831**	**.6810**	**.7605**	**.8650**	**.6450**	**.7428**	**.9599**	**.9927**
4	RHCP-v2	.1697	.3156	.1835	.3299	.3143	.4630	.4595	.5405	**.5134**	**.5165**	**.6813**	**.7018**	**.6313**	**.6116**	**.9519**	**.9821**
5	RHCP	.1967	.2747	.2101	.2952	.3190	.4179	.4835	.4866	.4971	.5029	**.6718**	**.6913**	**.6416**	**.5640**	**.9092**	**.9725**
6	RLCard	.0911	.1305	.1045	.1437	.1350	.2395	.2982	.3187	.3087	.3282	.4465	.5535	**.5839**	**.4603**	**.9314**	**.9539**
7	CQN	.1487	.2314	.1649	.2674	.2572	.3550	.3884	.3687	.4360	.3584	.5397	.4161	.5238	.4762	**.8566**	**.9213**
8	Random	.0080	.0142	.0040	.0129	.0073	.0401	.0179	.0481	.0025	.0908	.0461	.0686	.0787	.1434	.3461	.6539

	Douzero		SL		RLCard		Random	
	L	P	L	P	L	P	L	P
Douzero	0.4146	0.5854	0.5087	0.7145	0.8391	0.874	0.9817	0.9917

图 6. WP 实验结果示意

ADP(Average Difference in Points) 表示两者得分的分差。每个算法基础分是 1，每有一个炸弹，最后结算的得分都会翻倍。算法 A 对算法 B 的 ADP 指标大于 0 代表算法 A 强于算法 B。

上图为论文中 ADP 结果，下图为本次复现结果，可以看出，基本达到了原文的效果。

Rank	A \ B	DouZero		DeltaDou		SL		RHCP-v2		RHCP		RLCard		CQN		Random	
		L	P	L	P	L	P	L	P	L	P	L	P	L	P	L	P
1	DouZero	-0.435	0.435	**-0.342**	**0.858**	**0.287**	**1.112**	**1.436**	**1.888**	**1.492**	**1.850**	**2.222**	**2.354**	**1.368**	**2.001**	**3.254**	**2.818**
2	DeltaDou	0.342	-0.858	-0.476	0.476	**0.268**	**1.038**	**1.297**	**1.703**	**1.312**	**1.715**	**2.270**	**2.648**	**1.218**	**1.849**	**3.268**	**2.930**
3	SL	-1.112	-0.287	-1.038	-0.268	-0.364	0.364	**0.564**	**1.142**	**0.658**	**1.114**	**1.652**	**1.990**	**0.878**	**1.196**	**3.026**	**2.415**
4	RHCP-v2	-1.888	-1.436	-1.703	-1.297	-1.142	-0.564	-0.209	0.209	**0.074**	**0.029**	**1.011**	**1.230**	**0.750**	**0.677**	**2.638**	**2.624**
5	RHCP	-1.850	-1.492	-1.715	-1.312	-1.114	-0.658	-0.029	-0.074	-0.007	0.007	**1.190**	**1.328**	**0.927**	**-0.432**	**2.722**	**2.717**
6	RLCard	-2.354	-2.222	-2.648	-2.270	-1.990	-1.652	-1.230	-1.011	-1.328	-1.190	-0.266	0.266	**0.474**	**-0.138**	**2.630**	**2.312**
7	CQN	-2.001	-1.368	-1.849	-1.218	-1.196	-0.878	-0.677	-0.750	0.432	-0.927	0.138	-0.474	0.056	-0.056	**1.832**	**1.992**
8	Random	-2.818	-3.254	-2.930	-3.268	-2.415	-3.026	-2.624	-2.638	-2.717	-2.722	-2.312	-2.629	-1.991	-1.832	-0.883	0.883

	Douzero		SL		RLCard		Random	
	L	P	L	P	L	P	L	P
Douzero	-0.5046	0.5046	0.2828	1.279	2.2212	2.4672	3.2052	2.8344

图 7. ADP 实验结果示意

6 总结与展望

本文提供了一种应用于特定场景的自博弈算法，通过使用并行地采样器减缓蒙特卡洛方法需求采样效率的劣势，并不断地自我博弈提升效果，最终达到 sota 的效果。这种针对棋类大动作空间任务的做法，可以尝试使用。为了应对棋牌类状态空间大等挑战，我通过深度神经网络、动作编码和并行参与者增强了经典的蒙特卡罗方法。我希望简单的蒙特卡罗方法可

以在如此困难的领域中产生强有力的解决办法将会激励未来的研究。对于未来的工作，我将探索以下方向。首先，我计划尝试其他神经架构，例如卷积神经网络和 ResNet。其次，探索 off-policy 学习，提高训练效率。具体来说，我将研究是否以及如何通过经验回放来提高挂钟时间和样本效率。

参考文献

[1] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Tobias Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

[2] Lasse Espeholt, Raphael Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. In *International Conference on Learning Representations*, 2019.

[3] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. In *Advances in Neural Information Processing Systems*, 2009.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[5] Daochen Zha, Jingru Xie, Wenye Ma, Sheng Zhang, Xiangru Lian, Xia Hu, and Ji Liu. Douzero: Mastering doudizhu with self-play deep reinforcement learning. volume 139, pages 12333–12344. PMLR, 2021.