|  |  |
| --- | --- |
| **学 号： 2017214142** | **密 级：** |



**Hefei University of Technology**

**本科毕业设计（论文）**

**UNDERGRADUATE THESIS**



**类 型：** 设计

**题 目：**基于MADDPG算法的球类游戏AI设计与实现

**专业名称：** 电子商务

**入校年份：** 2017级

**学生姓名：** 鲁泽昊

**指导教师：** 伍章俊

**学院名称：** 管理学院

**完成时间：** 2021.6

合 肥 工 业 大 学

**本科毕业设计（论文）**

**基于MADDPG算法的球类游戏AI设计与实现**

**学生姓名**： 鲁泽昊

**学生学号**： 2017214142

**指导教师**： 伍章俊 副教授

**专业名称**： 电子商务

**学院名称**： 管理学院

2021年 6月

A Dissertation Submitted for the Degree of Bachelor

**Design and Implement of Ball Game AI Based on MADDPG Algorithm**

By

Zehao Lu

Hefei University of Technology

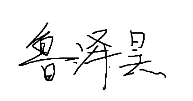
Hefei, Anhui, P.R.China

June, 2021

**毕业设计（论文）独创性声明**

本人郑重声明：所呈交的毕业设计（论文）是本人在指导教师指导下进行独立研究工作所取得的成果。据我所知，除了文中特别加以标注和致谢的内容外，设计（论文）中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 合肥工业大学 或其他教育机构的学位或证书而使用过的材料。对本文成果做出贡献的个人和集体，本人已在设计（论文）中作了明确的说明，并表示谢意。

毕业设计（论文）中表达的观点纯属作者本人观点，与合肥工业大学无关。



毕业设计（论文）作者签名：　　　　　　签名日期：2021年6 月6日

**毕业设计（论文）版权使用授权书**

本学位论文作者完全了解 合肥工业大学 有关保留、使用毕业设计（论文）的规定，即：除保密期内的涉密设计（论文）外，学校有权保存并向国家有关部门或机构送交设计（论文）的复印件和电子光盘，允许设计（论文）被查阅或借阅。本人授权 合肥工业大学 可以将本毕业设计（论文）的全部或部分内容编入有关数据库，允许采用影印、缩印或扫描等复制手段保存、汇编毕业设计（论文）。

（保密的毕业设计（论文）在解密后适用本授权书）

|  |  |
| --- | --- |
| 学位论文作者签名： | 指导教师签名： |
| 签名日期：　2021年 6 月 6 日 | 签名日期：　2021　年 6 月 6 日 |
|  |  |

**摘 要**

游戏AI一直是强化学习的热门实验载体。自2013年DQN算法被提出后，将深度神经网络与强化学习相结合成为了强化学习的主流，引爆了研究者对强化学习的热情。通常的单Agent（智能体）强化学习只有单个Agent在环境中，环境只会被该Agent影响。但是当Agent数量增加，环境会受多个Agent同时影响，因此环境的不稳定性会随Agent数量的增加而快速提升。MADDPG算法作为DDPG算法的衍生，将深度神经网络、Actor-Critic框架和中心化训练-去中心化执行的方法结合到了一起，能够有效解决多Agent带来的环境不稳定性。本文使用Unity开发了一款简单的球类游戏，并通过ML-Agents插件成功使用MADDPG算法实现了该游戏的高性能游戏AI。

**关键词：**多智能体强化学习；游戏AI；MADDPG；ML-Agents

**ABSTRACT**

Game AI has always been a popular experimental vehicle for reinforcement learning. Since the DQN algorithm was proposed in 2013, the combination of deep neural networks and reinforcement learning has become the mainstream of reinforcement learning, detonating researchers' enthusiasm for reinforcement learning. Generally, single-agent reinforcement learning has only one single agent in the environment, and the environment can only be affected by this agent. But when the number of Agents increases, the environment will be affected by multiple Agents at the same time, so the instability of the environment will increase rapidly with the increase of the number of Agents. As a derivative of DDPG algorithm, MADDPG algorithm combines deep neural network, Actor-Critic framework and centralized training-decentralized execution method, which can effectively solve the environmental instability caused by multi-Agent. This article uses Unity to develop a simple ball game, and successfully uses the MADDPG algorithm to achieve the game's high-performance game AI through the ML-Agents plug-in.

**Keywords:** Multi-Agent Reinforcement Learning, Game AI, MADDPG, ML-Agents

**目 录**

[1 绪 论 1](#_Toc73444623)

[1.1 研究背景 1](#_Toc73444624)

[1.2 研究现状 1](#_Toc73444625)

[1.3 主要工作 3](#_Toc73444626)

[1.4 组织结构 4](#_Toc73444627)

[2 相关概念与技术 4](#_Toc73444628)

[2.1 强化学习基础 4](#_Toc73444629)

[2.1.1 强化学习概述 4](#_Toc73444630)

[2.1.2 马尔科夫决策过程 6](#_Toc73444631)

[2.1.3 贝尔曼方程 7](#_Toc73444632)

[2.2 无模型方法 8](#_Toc73444633)

[2.2.1 蒙特卡洛方法 8](#_Toc73444634)

[2.2.2 时序差分（TD）方法 10](#_Toc73444635)

[2.3 强化学习方法分类 10](#_Toc73444636)

[2.3.1 基于价值函数更新 10](#_Toc73444637)

[2.3.2 基于策略更新（Policy Gradient） 12](#_Toc73444638)

[2.3.3 Actor-Critic方法 14](#_Toc73444639)

[2.4 基于AC框架的MADDPG算法 15](#_Toc73444640)

[2.4.1 多Agent强化学习困境 15](#_Toc73444641)

[2.4.2 基于Actor-Critic框架和DDPG算法的MADDPG算法 15](#_Toc73444642)

[3 游戏与环境设计 19](#_Toc73444643)

[3.1 游戏设计 19](#_Toc73444644)

[3.1.1 游戏机制 19](#_Toc73444645)

[3.1.2 游戏目标 20](#_Toc73444646)

[3.1.3 违规行为 20](#_Toc73444647)

[3.1.4 回合结束条件 20](#_Toc73444648)

[3.2 环境设计 20](#_Toc73444649)

[3.2.1 状态设计 21](#_Toc73444650)

[3.2.2 动作设计 22](#_Toc73444651)

[3.2.3 奖励函数设计 22](#_Toc73444652)

[4 游戏环境与算法实现 24](#_Toc73444653)

[4.1 开发平台 24](#_Toc73444654)

[4.1.1 Unity 24](#_Toc73444655)

[4.1.2 ML-Agents 24](#_Toc73444656)

[4.2 游戏实现 26](#_Toc73444657)

[4.2.1 CollectObservation函数 26](#_Toc73444658)

[4.2.2 OnActionRecieved函数 27](#_Toc73444659)

[4.3 算法实现 30](#_Toc73444660)

[4.3.1 \_process\_step 32](#_Toc73444661)

[4.3.2 update 35](#_Toc73444662)

[5 实验结果 38](#_Toc73444663)

[5.1 30时间步瓶颈 38](#_Toc73444664)

[5.2 算法实验结果 38](#_Toc73444665)

[5.3 奖励函数实验结果 40](#_Toc73444666)

[6 总结与展望 41](#_Toc73444667)

[参考文献 42](#_Toc73444668)

**插图清单**

**图2.1 马尔可夫决策过程图** **7**

**图2.2 MADDPG算法结构图** **18**

**图3.1 游戏示意图** **20**

**图3.2 动作设计示意图** **22**

**图4.1 ML-Agents总体结构图** **25**

**图4.2 ML-Agents完整结构图** **26**

**图4.3 ML-Agents时序图** **32**

**图5.1 实验总体结果图** **40**

**图5.2 PPO算法实验图** **40**

**图5.3 PPO-MADDPG算法对比图** **41**

**图5.3 奖励函数对比图** **42**

# 1 绪 论

## 1.1 研究背景

游戏开发是一个高速发展中的行业，从最早的俄罗斯方块到如今的各类3A级游戏，不论是游戏的画质，还是游戏的可玩性都有了飞跃性的提升。游戏AI作为单机游戏及绝大部分网络游戏中不可或缺的组成部分，发挥着提升游戏寿命、挑战性以及逼真性的重要作用。游戏AI指游戏中的智能行为，它的职责主要是在游戏中模拟人类的操作、对行为作出响应以及自动产出游戏内容等，从而实现人机互动，使游戏变得更加真实、更具有挑战性。游戏AI通常可以分为定性与非定性两种。定性游戏AI的行为是由人类事先设计的。它每一步的行为表现都是已经被定义好的结果，不存在任何不确定性，主要的实现方式为简单的if else语句、决策树以及有限状态机等。而非定性游戏AI则更具有不确定性，一般依靠机器学习方法学习，主要的实现方法是神经网络、以及遗传算法等。早年的游戏AI多为定性AI，原因是当时的游戏都比较简单，它们的游戏AI可以轻易地通过定性AI实现。另外由于当时电脑算力不足，机器学习方法并未发展成熟。如今，越来越多研究者将机器学习方法引入游戏AI领域，使非定性游戏AI成为了热点。

强化学习是近年来非常热门的机器学习领域。2015年10月基于深度强化学习的游戏AI AlphaGo以5:0的比分完胜欧洲围棋冠军樊麾；2016年3月对战世界围棋冠军李世石，并以总比分4:1获得胜利。这般战绩立刻使强化学习成为了公众的焦点话题。在Alpha Go战胜李世石后，Alpha Zero利用自主学习技术再次超越了人类数千年积累的经验，刷新了人类对于人工智能的认识，使强化学习与深度学习结合的方法受到了学术界和产业界的关注。

## 1.2 研究现状

截至目前，强化学习的研究已开展近半个世纪。其中，主要的发展阶段可以分为三个：20世纪50年代至60年代：强化学习的诞生阶段；20世纪80年代至20世纪90年代：强化学习的深入发展阶段；21世纪15年至今：强化学习爆发式突破阶段。

强化学习的发展有两条主线：一条是基于动物学习心理的试错学习法，另一条是基于最优控制及使用价值函数和动态规划的方法。其中，基于动物学习心理的试错法是我们最为常用，也是研究者最关注的一条主线。试错学习的本质起源于1911年由Thorndike提出的效应定律（Law of Effect），它说明了强化事件对选择行为的倾向性产生的影响。1948年，图灵在一次报告中提出了一种“快乐-痛苦系统”，是最早提出的将试错学习应用在人工智能领域的可能性。20世纪60年代，“强化学习”这个术语首次被提出，用以描述试错学习在工程领域的应用。1960年，Minsky讨论了关于试错学习的相关问题，例如预测、期望和“强化学习中的基础性功劳分配问题”，为后来强化学习的发展作出了铺垫性的成果[1]。

在强化学习的深入发展阶段，由于神经网络理论的进步，强化学习的研究出现了一次高潮。1985年，Barto等提出了ARP算法，并通过实际系统描述了强化学习的应用现状[2]。1988年，Sutton提出了时序差分TD方法，目前已经被广泛应用到各类强化学习算法中。 在论文中，Sutton将时序差分视为一个预测方法，介绍了TD(λ)算法同时证明了它的收敛性[3]。Dayan完成了该算法的进一步证明[4]。1989年，Charis Watkins提出了著名的Q-learning算法并对收敛性进行了证明，成功将最优控制和时序差分合并[5]。自Q-learning被提出后，强化学习的研究被推向高潮。Peng等提出多步Q-learning算法[6]。Werbos将动态规划和Q-learning结合在了一起[7]。Tadepali提出了H-Learning算法，有更快的收敛速度[8]。

由于深度学习在近几年的发展，强化学习的研究再次有了突破性进展。深度学习的特征是可以通过多层处理，将底层特征转化为高层特征。强化学习在这个阶段结合了深度学习理论，形成了深度强化学习。Martin提出了NFQ算法,该算法结合了Q-learning和神经网络，实际地把图像这一高维数据作为输入传递给了神经网络[9]。自此，越来越多的研究者开始以图像等高维数据作为输入，进而使游戏成为了强化学习中非常常用的实验载体。2013年，Mnith等将雅达利游戏作为实验载体，提出了Deep Q-learning(DQN)算法，以深度神经网络作为值函数，将深度神经网络与强化学习中的Q-learning算法结合，成功训练出了以游戏画面帧为输入，摇杆轴输出的游戏AI，其性能远超之前的其他强化学习算法，甚至在某些游戏中的表现能够超越人类[10]。DQN的提出引发了强化学习领域的革命，自此强化学习与深度学习有了密不可分的关联。之后，由David等人开发的游戏AI Alpha Go 在围棋这一复杂游戏中成功战胜顶级棋手李世石，使强化学习获得了世人的广泛关注。之后，非常多的研究者提出了DQN算法的改进版本。2015年，DeepMind在一年内提出了Nature DQN，Double DQN和Prioritized Replay DQN算法，它们相较于DQN算法在性能上有了非常大的提升。Nature DQN在计算目标Q值时使用了Target Q-Network，使目标计算和神经网络当前参数值无关，从而加快收敛[11]。Double DQN改进了目标值计算方法，消除了过度估计的问题[12]。Prioritized Replay DQN根据TD误差对Replay Buffer中的经验添加权重，使TD误差大的经验更容易被采样，这样更有利于算法收敛[13]。

在多Agent强化学习方面，存在很多单智能体强化学习中不存在的问题，例如维度过高、奖励函数设计困难、环境不稳定性等。研究者发现分层强化学习可以很好地解决多智能体智能问题。在最简单的情况下，可以为每个Agent单独分配训练网络，从而降低训练的计算复杂度。依据这个思想，Tampuu使用DQN算法单独训练每一个Agent，提出一种多智能深度强化学习模型，包含协作与对抗[14]。相对的，Foerster等提出了分布式深度循环Q网络（DDRQN），适用于Agent只能部分观察环境，Agent需要相互沟通的推理式场景[15]。这种情况下单独为每个Agent进行训练通常是获取不到有效策略的，但是DDRQN可以解决这个问题。

## 1.3 主要工作

本论文的主要工作是使用强化学习领域的近期成果MADDPG算法在本文设计的一个球类游戏中训练出游戏AI，与不同算法进行性能的横向比较，并归纳不同奖励函数设计的性能差别以及奖励函数设计的经验。该游戏AI能够产生类人行为，能取得较高得分（奖励），该得分由编者定义。本论文的整个项目主要分为两个部分：游戏环境及算法。

游戏环境的设计包括游戏本身以及奖励函数、状态等。前者由Unity引擎和ML-Agents API实现，包含两个对立的队伍。双方是一种合作关系，它们需要使球不落地，且不能让球触碰边界和球网。而后者属于强化学习马尔科夫决策过程中的要素，是建模强化学习问题必需的。状态是Agent从游戏环境中获取的观测值，如角色的坐标等。奖励函数是Agent在执行动作后环境返回的一个奖励信号, 该信号需要在环境中添加。

算法部分核心理论源于MADDPG算法，使用ml-agents插件底层API控制游戏环境的运行，Pytorch和ML-Agents源码作为深度强化学习方法的实现。

由于在奖励设计及状态设计中遇到了困难，花费了最长的时间，因此本文将根据实验结果提出奖励设计的一些经验归纳，并通过实验进行验证。

## 1.4 组织结构

本文的组织结构如下:

第一章绪论部分主要讲述游戏AI及强化学习的背景、相关研究现状，并叙述论文主要工作和组织结构。

第二章强化学习基础。该部分将项目中涉及到的理论基础作详细介绍，包括强化学习基础理论（MDP）、无模型方法、强化学习方法的主要框架分类和本文使用的MADDPG算法。

第三章游戏规则与环境设计。该部分主要介绍了游戏的机制以及强化学习框架的几个要素。

第四章游戏环境与算法实现。该部分介绍了实现方法并针对部分关键代码进行释义。

第五章实验结果。该部分比较了不同算法和奖励函数的性能。

第六章总结展望。该部分对本文的工作和成果进行总结，并对强化学习在游戏AI领域的未来进行展望。

# 2 相关概念与技术

## 2.1 强化学习基础

### 2.1.1 强化学习概述

强化学习是指Agent根据环境信息学习奖励信号最大化的行为准则。Agent不知道应该采取什么动作才能获取最大奖励，因此必须尝试与环境进行交互，才能发现哪些动作可以为它带来较大的收益。当我们思考人类的学习方式时，会发现它与强化学习惊人地相似。婴儿在自己玩耍时，会四处张望，挥舞四臂。在没有指引的情况下，它们只有通过与周围环境进行直接交互来学习知识。强化学习中的Agent就像是婴儿，它们起初对环境一无所知，但是经过摸索，它们能够知道执行哪些动作才能最大化自己获得的奖励值。强化学习问题的建模主要包括：状态、动作、奖励、策略、状态价值函数、动作值函数和环境转化概率。

#### 2.1.1.1 状态

强化学习中，状态（State）主要指Agent在环境中的状态，它是Agent从环境中得到的感知信息，可以是任何帮助决策的信息。

#### 2.1.1.2动作

动作（Action）指Agent根据自身策略与状态采取的行动。Agent在一个状态下采取特定动作的目的是获取最大奖励。

#### 2.1.1.3 奖励

奖励（Reward）表示Agent在采取动作改变环境后，环境产生的反馈值。回报（Return）是Agent在执行完一条完整的状态、动作序列后取得的奖励值累计。强化学习的学习目标即是使获得最大化回报，获取最多奖励。

|  |  |
| --- | --- |
|  | (2.1) |

公式(2.1)中，是马尔科夫决策过程中时刻t获得的回报，它是累计折扣奖励，γ代表折扣系数，取[0,1]。

#### 2.1.1.4 策略

Agent选取动作的行为准则被称为策略（Policy）。它本质上是一个函数，输入Agent的状态等信息，输出一个动作或动作分布。策略分为随机型策略和确定型策略。

对于随机型策略，该策略输出的是一个概率分布，Agent根据该分布选择动作。公式（2.2）表示在状态为s的情况下，每个动作对应一个概率。

|  |  |
| --- | --- |
|  | (2.2) |

对于确定型策略，确定输入后，策略只会有一个动作输出。如公式（2.3）所示。

|  |  |
| --- | --- |
|  | (2.3) |

#### 2.1.1.5 状态价值函数

状态价值函数描述了在马尔科夫过程中某个状态的价值，其本质是从该状态开始以策略π选择动作直到该幕结束得到的期望回报。如公式（2.4）所示。

|  |  |
| --- | --- |
|  | (2.4) |

#### 2.1.1.6 动作价值函数

动作价值函数（Q函数）描述了马尔科夫过程中某个状态下执行一个动作的价值，其本质上是从该状态下执行动作后，以策略π选择动作得到的期望回报。如公式（2.5）所示。

|  |  |
| --- | --- |
|  | (2.5) |

#### 2.1.1.7 环境转化概率

环境转化概率表示在某一状态下执行某动作后，环境迁移到另一状态的概率。如公式（2.6）。

|  |  |
| --- | --- |
|  | (2.6) |

### 2.1.2 马尔科夫决策过程

2.1.2.1马尔科夫性

马尔科夫性指后续状态出现的概率仅由当前状态决定，与先前状态无关。只要当前的状态确定，那么先前获得的信息都是无效的，当前获得的信息决定了未来的选择。当且仅当当前状态满足公式（2.7）时，称该状态满足马尔科夫性：

|  |  |
| --- | --- |
|  | (2.7) |

#### 2.1.2.2 马尔科夫决策过程

图2.1 马尔可夫决策过程图

马尔科夫决策过程（MDP）是一种通过交互来实现目标的强化学习理论框架。可以用五元组表示马尔科夫决策过程。其中、分别是状态、动作的有限集合，表示环境状态转移概率，是获得的奖励，表示奖励折扣。

在有限MDP中，每个离散时刻0,1,2,3…,Agent都会与环境进行交互。在每个时刻，Agent从环境中获取状态,根据选择一个动作。在时刻，Agent收到了来自环境的奖励信号，并获取到了新的状态。这个过程在一幕中不断重复，形成一个序列，称为轨迹，类似：

强化学习问题可以转换为马尔可夫决策过程。Agent首先从环境中获取环境信息和自身信息，根据一个行为准则（Policy）选择执行动作，对环境产生影响。同时，环境受到影响后会向Agent返回一个奖励信号，用来评估动作的优劣。上述过程会在每一个时刻进行，直至结束，形成一条完整的轨迹。我们在强化学习问题中一般采用MDP对问题进行建模，它的基础元祖为。

### 2.1.3 贝尔曼方程

在强化学习中，价值函数存在一个特性，每一个时刻的价值函数与后继时刻的价值函数存在递推关系，如公式(2.8)所示。

|  |  |
| --- | --- |
|  | (2.8) |

式(2.8)被称为贝尔曼方程，其中γ表示衰减系数。这个公式用来表达某状态和其后继状态价值的递推关系。它是使用经验估计价值函数的基础。

解决强化学习实质上就是找到一个最优策略使长期过程中的回报最大化。对于有限MDP，我们能通过比较价值函数来找到一个最优策略。最优策略和其他策略是一种偏序关系，即对于一个策略，如果它在任何状态上的期望回报大于等于其他任何策略，那么这个策略就是最优策略。因此，寻找最优策略的问题可以转化为寻找在每个状态价值最大的策略，即

|  |  |
| --- | --- |
|  | (2.9) |

将其带入贝尔曼方程(2.8)，可得到贝尔曼最优方程。

|  |  |
| --- | --- |
|  | (2.10) |

同理可得最优动作价值函数的贝尔曼方程。

|  |  |
| --- | --- |
|  | (2.11) |

通过不同的价值函数迭代方法进行求解，可以得到最优价值函数和最优策略。

## 2.2 无模型方法

强化学习中存在预测问题和控制问题。这里我们先解决的是预测问题，即如何得到特定策略下的状态价值函数。根据环境转化模型P是否已知，我们可以将强化学习问题分为两类：基于模型与无模型。前者可以使用动态规划等方法解决，而后者无法使用这类方法，通常采用蒙特卡洛方法（MC）和时序差分方法（TD）。在游戏AI领域，由于游戏的环境转化概率一般未知，所以本文主要讨论两种无模型方法：蒙特卡洛方法和时序差分方法。

### 2.2.1 蒙特卡洛方法

蒙特卡洛方法就是利用平均样本回报来估计状态价值，解决强化学习问题的方法。其核心思想是大数定律，也就是当随机变量大量重复地出现时，它们的平均值趋向于该随机变量的期望。蒙特卡洛方法通过不断重复MDP过程，取得大量蒙特卡洛序列。然后从这些序列中算出每个出现的状态的折扣回报，对所有序列求平均值，用来估计状态价值的真值。这种方法不需要知道环境的转化概率，因此是一种无模型方法。它可以直接与环境进行交互更新价值函数。蒙特卡洛方法仅适用于所有蒙特卡洛序列是有限的，所有的幕不能无限运行。

在状态为时，它的状态价值函数为公式(2.12)。

|  |  |
| --- | --- |
|  | (2.12) |

使用蒙特卡洛方法时，这个状态价值函数可以通过均值估计。每次访问一个状态时，就进行一次记录。如公式(2.13)所示。

|  |  |
| --- | --- |
|  | (2.13) |

其中表示这个状态出现的次数，而表示回报之和。因此可以将该状态的平均价值用下列公式(2.14)表示。

|  |  |
| --- | --- |
|  | (2.14) |

在足够大的情况下，=。通过这种方法，可以使均值收敛于状态价值的期望。

然而通过公式(2.13)、(2.14)计算均值比较麻烦。实际操作中，计算均值需要存储每个状态出现时的奖励，会消耗很多内存。有一种增量式的算法可以简化计算均值的方法，公式推导如下。

|  |  |
| --- | --- |
|  | (2.15) |

对于任何一个轨迹，，每次得到一个状态时，就使次数加一，然后通过增量公式更新状态价值估计值。

|  |  |
| --- | --- |
|  | (2.16) |

### 2.2.2 时序差分（TD）方法

时序差分和蒙特卡洛方法都是用来解决就价值函数估计的问题。对于确定的策略，以及获取到的序列中的非终止状态，这两种方法都会更新估计值,使它们更接近于真实的状态价值函数。两者的区别在于估计目标的不同。蒙特卡洛方法采用作为估计的目标，但是由于只能在一个完整序列结束后才能得出，因此蒙特卡洛方法必须等到序列结束才能进行更新。但是时序差分方法不同，它的估计值为。对于某个状态，只需等到下一时刻即可获取该状态的估计值。因为这个原因，时序差分算法可以在没有完整序列的情况下进行单步更新。

时序差分方法会在状态发生转移时立刻通过公式(2.17)更新状态价值估计值。

|  |  |
| --- | --- |
|  | (2.17) |

式(2.17)中被称为TD误差。是学习率，取值为[0,1]。

在一些情况下，一个序列可能非常长，这种应用场景中时序差分方法比蒙特卡洛方法更具优势，因为它能够更快进行更新而不必等到序列结束。另外，虽然还没有严谨的数学证明，但是时序差分方法在随机任务中通常比蒙特卡洛方法收敛地更快。

## 2.3 强化学习方法分类

强化学习按照更新方法的不同可以分为两类：基于价值函数(Value-based)的方法，基于策略（Policy）的方法。将两个方法结合，则可以得到Actor-Critic（AC）的方法。这种方式使用Actor决策， Critic对Actor的表现进行评估，更新Actor。

### 2.3.1 基于价值函数更新

强化学习的目标是使每一幕中累计的奖励最大化，也就回报最大化，如公式(2.18)

所示。

|  |  |
| --- | --- |
|  | (2.18) |

公式(2.18)中为奖励的折扣。对这个目标影响最直接的因素是策略。策略控制了Agent在某种状态下执行哪个动作，直接影响了奖励值的获取。因此，获取最大化回报的目标也可以表示为得到最优策略，使回报最大化。在前文中，我们提到了状态价值函数和动作价值函数，它们都是对一个状态或状态动作后续奖励的期望折扣和。我们使用这两个函数表征状态或状态-动作二元组的价值，指导动作的选择。

如上文中提到的，我们可以把强化学习问题分解为两个问题：预测问题和控制问题。在预测问题中，由于状态价值函数一般都是未知的，所以需要利用状态、动作、奖励等信息计算从而取得当前策略下的估计值。控制问题方面一般由贪心策略或ε-greedy策略选择最有价值的动作，推动与环境的交互并使最大化。这样不断与环境交互，迭代更新价值函数的过程我们称为基于价值函数的更新。这类更新方法中最为典型的是SARSA和Q-learning算法。

#### 2.3.1.1 SARSA算法

SARSA是一种在线（on-policy）的时序差分算法。与之相对的算法被称为离线算法(off-policy)。所谓在线与离线的区别在于动作选取的方法和更新价值函数时是否使用的是同一个方法。SARSA在价值函数计算和动作选择上都使用了ε-greedy策略，属于一种在线方法。

ε-greedy策略是如公式（2.20）所示。由于在学习起始阶段，Agent对环境的情况不了解，如果只采用贪心算法，即公式（2.19），那么Agent就仅会根据已经经历过的情况选取动作，而不会对未知空间进行探索。因此为了使探索未知和利用已有经验达到一种平衡关系，我们采用了ε-greedy策略。如公式(2.20)。为了使算法收敛，ε会随着迭代次数增加不断减小，直到为0。其含义为前期鼓励探索，后期趋于保守，以贪婪策略为主。

|  |  |
| --- | --- |
|  | (2.19) |

|  |  |
| --- | --- |
|  | (2.20) |

SARSA算法实际上由组成。分别表示状态、动作、奖励。在迭代计算价值函数时，首先使用ε-greedy策略利用当前状态选出一个动作。执行动作后，环境将返回新状态和奖励。这时再次利用ε-greedy策略基于新状态选择新动作。但是这个新动作不会被执行，而是代入公式（2.20）中用来迭代计算价值函数。这里使用了两次ε-greedy策略，所以SARSA算法被算作在线算法。式(2.20)是贝尔曼方程的动作价值函数版本，其中表示在状态时执行动作得到的期望回报。是步长，用于控制学习速率。

|  |  |
| --- | --- |
|  | (2.20) |

这个过程不断重复，直至价值函数收敛。收敛后，对于一个每一个特定的状态，最优策略为选取贪婪策略收敛后的动作价值函数最高的动作。

#### 2.3.1.2 Q-learning算法

Q-learning算法是一种离线时序差分算法。它在更新价值函数时采用了贪心算法，而在指导动作选择时使用ε-greedy策略。和SARSA算法相比，它们的主要流程基本相同，区别在于Q-learning在更新价值函数时使用的是贪心算法。SARSA在更新价值函数时采取ε-greedy策略，是一种边探索边更新的方法，需要随迭代减少ε，使价值函数收敛。但Q-Learning不是这样，因为它直接选取最优价值函数，也就是直接学习最优策略。

### 2.3.2 基于策略更新（Policy Gradient）

基于价值函数的更新方式存在一些缺陷。首先，在连续动作空间的问题中，基于值函数的更新方式会计算每个状态-动作二元组的价值。当动作是连续动作时，需要将动作进行离散化处理，这会导致该状态-动作二元组的数量急剧增加，使Q表的存储容量过大。其次，基于价值函数的更新方法的最优策略一般是确定性策略。在面对最优解是随机性策略的情况时，基于价值函数的更新方式是解决不了的。基于策略（Policy-based）的更新方式很好地解决了连续性动作和随机策略的问题。

联想基于价值函数更新方法中我们使用了动作价值函数，在基于策略的更新方法中，本文采用类似的思路，此时我们使用一个连续函数来表示策略。当策略是一个连续函数时，就可以采用连续函数中的方法求解最优策略。最简单的方法是蒙特卡洛策略梯度算法和确定性策略梯度（DPG）算法。

#### 2.3.2.1蒙特卡洛策略梯度算法

作为一个梯度上升算法，首先需要找到优化目标。由于强化学习的目标是回报最大化，可以联想到这个优化目标就是从初始状态开始回报的期望值。但是由于初始状态有时未知，优化目标可以取每个初始状态开始的回报平均值或期望回报单步平均值，在本文中取后者。因此目标如公式(2.21)所示。

|  |  |
| --- | --- |
|  | (2.21) |

上式中表示基于策略π生成的马尔科夫链的状态平稳分布，平稳分布的定义如公式(2.22)所示。

|  |  |
| --- | --- |
|  | (2.22) |

平稳分布是指在指定初始状态和状态转换概率后，经过一定次数的迭代后，状态会趋于一个平稳状态，服从马尔科夫链平稳分布。该理论由马尔科夫链平稳状态定理证明。

对目标求导后，可得到公式(2.23)。

|  |  |
| --- | --- |
|  | (2.23) |

蒙特卡洛策略梯度算法首先采样一个完整序列，对序列中的每一步进行迭代。首先计算从当前状态执行动作后到最终状态的所有累计奖励。然后，该算法用替换式(2.23)中的，对策略参数π进行更新完成策略迭代。整个流程如公式(2.24)所示。

|  |  |
| --- | --- |
|  | (2.24) |

其中为步长，代表单次更新的大小。

上述公式中的策略函数是一个符号，并没有具体实现。对于不同的策略函数，我们仅需把对应的代入公式(2.24)即可，因此本文不再讨论不同策略函数的情况。

#### 2.3.2.2 DPG算法

DPG算法的英文全称为Deterministic Policy Gradient，也就是确定性策略梯度算法。确定性表示该算法中的动作是确定性的，而不是随机性的。随机策略在一个状态采用的动作是基于某个概率分布的，确定性策略在同一状态只输出一个动作，最简单的方式是直接采用概率最大的动作。此时策略变为公式(2.25)。

|  |  |
| --- | --- |
|  | (2.25) |

DPG算法与蒙特卡洛策略梯度算法的思路大体相同，在一些细节上有改变。

|  |  |
| --- | --- |
|  | (2.26) |

上式与式(2.23)相比，由于策略变为确定性策略，所以期望由联合期望变为单随机变量的期望。

### 2.3.3 Actor-Critic方法

上文中提到的基于策略的更新方法解决了基于价值函数的更新方法中的问题，但是其本身需要一个完整的状态序列计算从各个状态执行动作后取得的累计奖励，这导致收敛过程较慢。另外，由于这个奖励本身就是一个Agent与环境互动的抽样产生出的结果，具有随机性，因此它是非常不稳定的。如果我们能直接通过函数逼近的方法估计这个奖励的期望值，那么训练过程将会更加稳定。

Actor-Critic方法就如名字一样，由Actor与Critic组成，它是由基于价值函数和基于策略方法结合而成的。Actor和前文中的策略函数一样，负责根据状态输出动作，与环境进行交互。Critic则与基于价值的方法类似，它负责对Actor产生的动作进行评估，这个评估结果用来指导Actor的更新。Actor的更新方式与基于策略的方法相同，如公式(2.23) (2.24)所示。其中由Critic得出，可由动作价值函数，优势函数以及TD误差等逼近。

当采用动作价值函数作为Critic的评价指标时，可以用逼近动作价值函数的真值，其中为参数。由公式(2.24)可得策略参数的更新公式。

|  |  |
| --- | --- |
|  | (2.27) |

Critic使用TD误差更新参数Ω。公式如(2.28)所示。

|  |  |
| --- | --- |
|  | (2.28) |

上文中提到该方法采用了函数逼近的方法。我们一般使用的函数逼近器有线性逼近、高斯逼近，以及比较常用的神经网络等。本文中我们采用了神经网络作为策略函数和价值函数的逼近器。

## 2.4 基于AC框架的MADDPG算法

### 2.4.1 多Agent强化学习困境

多Agent强化学习问题是单Agent强化学习问题的延伸。在单Agent强化学习问题中，由于不存在其他Agent影响环境，因此环境是稳定的。对于多Agent强化学习问题，由于存在其他Agent，不同的Agent都会对环境产生影响，使环境更为复杂且产生动态变化，使学习过程产生不稳定性。因此，直接将单Agent强化学习的方法应用到多Agent强化学习问题中是不明智的。在当前多Agent强化学习的问题中，存在以下问题：

1. 维度灾难。在单Agent强化学习中，只需要单独考虑一个动作。而对于多Agent强化学习，状态和动作的空间随Agent数量呈指数级增加，动作空间转为所有Agent的联合动作，这导致多Agent系统维度过高，计算量极大。
2. 奖励函数的设计。单Agent强化学习中目标为累计奖励最大化，只需要设计一个Agent的奖励函数。而多Agent强化学习问题中各个Agent的目标可能不同，因此可能需要设计不同的奖励函数。由于多Agent系统中Agent之间存在相互的影响，因此奖励函数的设计不能仅仅考虑单个Agent，还需要考虑多Agent之间的影响。
3. 环境的不稳定性。如前文所述，多Agent强化学习问题中Agent会相互影响，当多个Agent中的某个Agent的策略发生改变后，可能打破整体策略的平衡，导致算法难以收敛，即。

### 2.4.2 基于Actor-Critic框架和DDPG算法的MADDPG算法

本文使用的MADDPG（Multi-Agent Deep Deterministic Policy Gradient）算法通过中心化的训练和去中心化的执行消除不稳定性。中心化的训练指在训练时提供额外的信息来简化训练，使训练过程的不确定性降低。执行的去中心化指在测试阶段，不需要提供其他Agent的额外信息。MADDPG本质上是DDPG算法的变种，因此本文将由DDPG引入MADDPG。

#### 2.4.2.1 DDPG算法

DDPG（Deep Deterministic Policy Gradient），即深度确定性策略梯度算法。DDPG与DPG算法的不同在于使用了双网络结构（目标网络和当前网络）以及Actor-Critic方法。由于采用了Actor-Critic方法，一般包含Actor，Critic网络，因此采用了双网络结构DDPG算法实际包含四个网络，分别是目标Actor、当前Actor、目标Critic、当前Critic。使用双网络的目的主要是为了解耦迭代时数据之间的相关性，加快收敛。另外，DDPG算法与DQN类似，使用了经验回放，减少数据之间的联系。DDPG算法流程如下：

1. 初始化参数、经验回放池、获取初始状态
2. 进行第一轮迭代
   1. 使用当前Actor网络基于状态输出动作，并添加噪音（添加随机性）。
   2. 执行动作，获取下一状态与奖励
   3. 把五元组存入经验回放池
   4. 令
   5. 进行第二轮迭代
      1. 从经验回放池中采样m个样本
      2. 计算目标值，其中为目标Critic网络，为目标Actor网络。
      3. 对均方误差使用反向传播更新当前Critic中的参数，其中为当前Critic网络。
      4. 对目标均值使用反向传播更新当前Actor中的参数，为当前Actor网络。
      5. 每隔一段时间对目标Critic和目标Actor网络的参数与进行软更新 其中远小于1。

上述过程中值得注意的是，目标Actor网络和目标Critic网络的主要作用是计算目标值y。目标网络不会直接由反向传播更新，而是在当前网络更新后再进行软更新。

#### 2.4.2.2 MADDPG算法

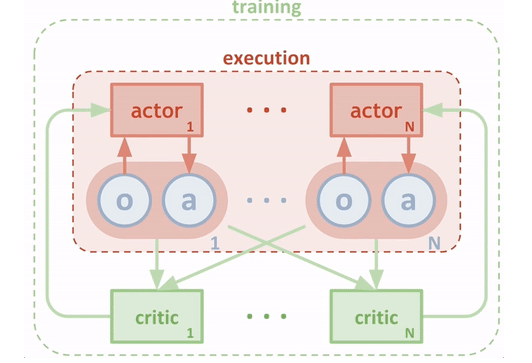
 针对多Agent环境不稳定的问题，基于DDPG算法的MADDPG算法被提出。在单Agent环境中，由于单个Agent的训练会被另一个Agent策略的改变所影响，因此直接从经验回放池中提取先前Agent的经验是无意义的。出现这个问题最根本的原因是各个Agent之间缺少交换信息的环节，也就是每个Agent之间并不知道对方会采取什么样的策略，只能通过自己一个Agent的观察结果来选择动作。这样只注重了个体，没有关注整体。MADDPG算法基于DDPG算法提出了一个很简单的改进：中心化的训练、非中心化执行。

图2.2 MADDPG算法结构图[16]

中心化训练实际上就是使用上帝视角的Critic指导Actor进行更新。具体来说，这个Critic可以获取所有信息以及所有Agent执行的动作。如图2.2所示，Actor虽然不能获得其他Agent的任何信息，但是每个Actor都有一个上帝视角的Critic作为导师指导更新。去中心化的执行表示在测试阶段，整个过程Critic并不参与，因此每个Actor获取的信息仅仅是单个Agent获取的信息。

这样的改进理论上解决了多Agent问题中环境不稳定的问题，即

综上所述，MADDPG算法流程如下：

1. 初始化参数、经验回放池、获取初始状态
2. 进行第一轮迭代
   1. 对于每个Agent，使用当前Actor网络基于状态输出动作，并添加噪音（添加随机性）。
   2. 执行动作，获取下一状态与奖励
   3. 把五元组存入经验回放池
   4. 令
   5. 进行第二轮迭代
      1. 从经验回放池中采样m个样本
      2. 计算目标值，其中为目标Critic网络，为目标Actor网络
      3. 对均方误差使用反向传播更新当前Critic中的参数, 其中为当前Critic网络。
      4. 对目标均值使用反向传播更新当前Actor中的参数，为当前Actor网络。
      5. 每隔一段时间对目标Critic和目标Actor网络的参数与进行软更新 其中远小于1。

与DDPG算法相比，MADDPG算法的区别在于经验中信息、动作、和奖励换成了全信息、联合动作和整体奖励，这使Agent的Actor网络在训练时可以获取其他Agent的信息，解决了环境不稳定的问题。

# 3 游戏与环境设计

## 3.1 游戏设计

### 3.1.1 游戏机制

图3.1 游戏示意图

如图3.1所示，本论文设计了一款简单的打球游戏作为强化学习的环境载体。游戏场地使用了一个方形羽毛球场，球场周围由六面空气墙包围，球场中间存在一个可穿透的空气墙。

游戏中可移动的物体为两位玩家角色和一个小球。球受重力影响，且会被场地四周的空气墙阻挡，但不会被场地中间的空气墙阻挡。玩家角色可根据输入的方向产生定向的推力进行移动，并且移动时场地会有微小的摩擦力，防止玩家角色在无输入时因惯性保持移动。由于受到空气墙包围，玩家角色只能在己方半场活动，接触空气墙后会被阻挡。玩家角色可以进行挥拍，挥拍后球拍会以一定角加速度产生90度角位移。玩家可以跳跃，也可以在跳跃时击球，但是不能在空中再次跳跃。球拍网采用特殊的材质，触碰球后会产生较大垂直于网面的弹力，从而击飞小球。游戏开始时，玩家角色分别在球场两边的固定地点出现，球会随机出现在球场两边的固定位置周围。

### 3.1.2 游戏目标

游戏中每个玩家的目标是尽可能用球拍击球过网延长球的滞空时间，即在不违规的情况下，使球滞空时间最大化。

### 3.1.3 违规行为

该游戏设置了一些犯规行为：

1. 某队玩家在击球后短时间内再次击球。
2. 某队击球后触网
3. 某队击球后触碰己方半场边界
4. 某队击球后触碰对方半场边界

发生违规行为后，玩家会受到一定惩罚，并使当前回合结束，开始新回合。

### 3.1.4 回合结束条件

游戏有两个结束条件：如果发生上述犯规行为则回合结束；如果时间步超过5000步，则回合结束。

## 3.2 环境设计

将强化学习应用到游戏的前提是将游戏建模为马尔科夫决策过程（MDP）的形式。如2.1.2章所述，MDP的基础元组为<S,A,P,R,γ>，需要定义该元祖才能在游戏上应用强化学习算法。因为元组中环境转化概率P未知，所以我们采用无模型方法计算价值。价值折扣γ一般取0.99。状态S，动作A，奖励R的设计分别由3.21、3.2.2、3.2.3章介绍。

### 3.2.1 状态设计

强化学习中的状态信息必须包含所有决策需要或对决策有利的信息。显然，玩家角色和球的坐标是必要的，否则Agent无法打中球。玩家角色和球当前的速度也是需要的，因为当角色在移动的过程中进行转向和直接转向所需时间不同，所以速度是改变坐标所必须的信息。玩家角色本身的旋转角会影响挥拍时拍子在世界坐标下的角坐标，因此为了使Agent精确挥拍，必须向它提供当前角色所处的旋转角。由于游戏中限定角色只能绕y轴旋转，因此本着只选择必需信息的原则，本文只使用yaw旋转角。本游戏中可以在挥拍的过程中取消挥拍或继续挥拍，在某一时刻拍子可能处于0~90度的任何一个角度，所以当前拍子的角度是Agent决定是否挥拍的一个决策因素，球拍当前相对于玩家的旋转角需要被纳入状态信息。另外，由于共有16个场地同时训练，应当统一所有场地中的输入条件，所以提到的坐标皆为相对坐标。综上所述，本游戏将一个Agent的状态定义为一个14维数据：

[

球对于人的X轴坐标，球对于人的Y轴坐标，球对于人的Z轴坐标，

球的X轴速度，球人的Y轴速度，球人的Z轴速度，

Agent对于场地的X轴坐标，Agent对于场地的Y轴坐标，Agent对于场地的Z轴坐标，

Agent的X轴速度，Agent的Y轴速度，Agent的Z轴速度，

Agent的旋转角，球拍角度

]

值得注意的是2.4.2节中提到本文采用了联合状态，该联合状态并非在状态设计中将所有Agent信息作为一个Agent的状态，而是在训练时将每个Agent获取到的状态信息合并为联合状态

### 3.2.2 动作设计

图3.2 动作设计示意图

本游戏中所有动作均为离散动作。离散动作可以由branch表示。branch代表一个行为，例如移动、挥拍。branch中可以包含多个取值，例如在移动branch中可以有向前移动和向后移动等离散的动作值。每一个Agent在每一个时间步可以从移动branch、跳跃branch、转向branch和挥拍branch中选择四个动作值。每个动作值表示一个特定动作。

每个Agent的动作设计如下：

branch[0]=[不跳跃，跳跃]

branch[1]=[不挥拍，挥拍]

branch[2]=[不旋转，逆时针旋转，顺时针旋转]

branch[3]=[不移动，向前移动，向后移动，向左移动，向右移动]

值得注意的是，在选择执行跳跃后的一段时间内不应该能继续选择跳跃，选择挥拍后一段时间内不能再次挥拍。本文采用ActionMask技术屏蔽挥挥拍时的挥拍动作。同理，在跳跃也使用ActionMask进行了处理。由于本文采用离散取值，因此例如向前移动向后移动实际上都是指受到一个固定方向固定大小的力，从而使角色向该方向移动。

### 奖励函数设计

#### 3.2.3.1 原始奖励函数

第一套奖励函数如公式所示

其中

公式中表示单个时间步中Agent获取的奖励总和。分别表示球拍碰撞球的奖励、球从己方触碰中间空气墙的奖励。代表惩罚，即负向奖励。公式中的分别表示击球后球触碰己方边界、球触碰对方边界和短时间内连续触碰两次球的惩罚。

这一套奖励函数旨在激励Agent打球过网。Agent在击球后即可立刻获得奖励，这促使Agent在选择动作时更偏向于用球拍接触球，提升击球概率，增加击球过网的概率。若球被打到错误的位置，例如出界等，那么会抵消先前由击球得来的奖励，防止Agent通过随意击球得分。在球过中间线后，Agent会获得较大数值的奖励，保证了Agent学习的目标是将球打到对方半场。

#### 3.2.3.2 改进奖励函数

然而在实验过程中，发现原始奖励函数在训练过程中有时不稳定，经常收敛在局部最优，即难以学会打回发球，并且训练经常发生抖动现象，导致收敛速度很慢。

为了解决这些问题本文提出了第二套奖励函数，如公式所示

其中

就是与拍和球的距离相关的函数，表示球和拍子的距离。上式中的函数表示接近3.5的程度。越接近3.5则输出值越接近1，越接近0则输出值越接近0。如公式所示

添加该函数的目的和效率对比将在第五章实验结果内进行展示，这一部分不再详细展开。

# 4 游戏环境与算法实现

前一章介绍了环境和游戏机制的设计，在本章中，将介绍它们的具体实现方法。

## 4.1 开发平台

### 4.1.1 Unity

本文中的游戏使用Unity游戏引擎开发。Unity是由Unity Techonolgy开发的跨平台游戏引擎。游戏引擎即一种框架，为开发者提供了游戏开发的必需基础功能，如物理特性、光影渲染、资源管理等。Unity的物理引擎简单高效地满足了本文游戏中的弹力需求和重力需求，并且在对强化学习的支持上，Unity较其他引擎有很大优势，因此本文采用Unity实现游戏。

### 4.1.2 ML-Agents

本文使用ML-Agents管理强化学习流程。ML-Agents是Unity官方提供的一款开源强化学习插件，为Unity提供强化学习支持。它总共包含了三个部分，总体结构如图4.1所示。

强化学习环境

Communicator

Python API

图4.1 ML-Agents总体结构图

强化学习的环境在本文中就是前文提到的游戏。该游戏定义了MDP的五要素，因此整个游戏可以被视为强化学习环境。作为游戏中的插件，ML-Agent提供了流程控制功能，例如完成游戏后重置、加快游戏速度等。Communicator（沟通器）是ML-Agents中负责数据传输的部分。它收集环境中的状态和奖励值，并负责与Python接口交互。Python API用于强化学习算法的实现。它从Communicator获取强化学习需要的所有信息，训练出需要的模型。在测试阶段根据Communicator的输入信息给出动作反馈。

在学习环境中，包含学院、大脑、Agent三个组件。学院负责流程的控制，是整个环境的控制器。大脑是Agent的决策中心，负责与之相连的所有Agent的决策。大脑实际上就是一个从状态到动作的映射，一般由神经网络或启发式探索实现。同一环境下可以有多个大脑同时训练。Agent就是强化学习中的对象，它负责从环境中提取状态信息给它相连的大脑，并采取大脑给出的动作与环境交互。完整结构如图4.2所示。

强化学习环境

学院

Communicator

Python API

大脑

Agent

Agent

大脑

Agent

Agent

图4.2 ML-Agents完整结构图

## 4.2 游戏实现

Unity以C#作为脚本语言。本游戏中包含五个脚本：Area.cs，HitBall.cs，PreventOut.cs，BedmintonAgent.cs，HitWall.cs，它们各自嵌入在Agent、场地、球等Gameobject中。由于篇幅限制，仅展示C#脚本中状态、动作的实现，不展示Unity引擎中的实现。状态的获取实现在CollectObservation函数，收取动作的实现在OnActionRecieved函数。

### 4.2.1 CollectObservation函数

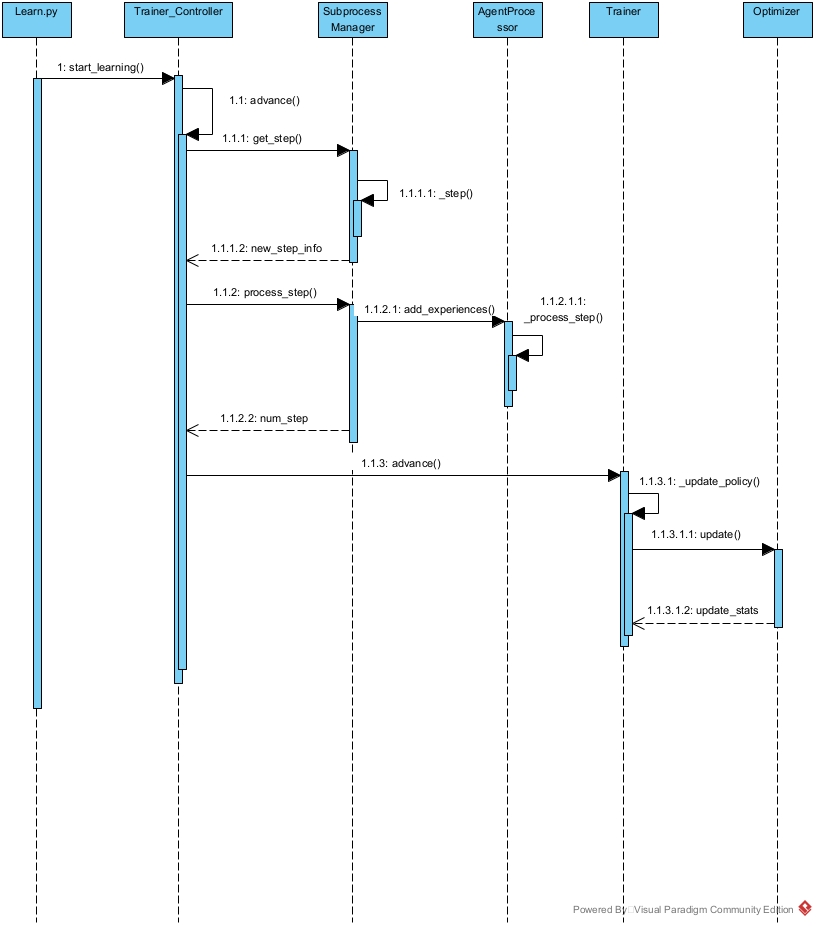
1. *//向sensor提供观测值，传递给算法*
2. public override void CollectObservations(VectorSensor sensor)
3. {
4. *//球相对人的参数，前向、右向为正*
5. sensor.AddObservation(m\_InvertMult\*(ball.transform.position.x - myArea.transform.position.x));
6. sensor.AddObservation(ball.transform.position.y - myArea.transform.position.y);
7. sensor.AddObservation(m\_InvertMult \* (ball.transform.position.z - myArea.transform.position.z));
8. sensor.AddObservation(-m\_InvertMult\*m\_BallRb.velocity.x);
9. sensor.AddObservation( m\_BallRb.velocity.y);
10. sensor.AddObservation(m\_InvertMult \* m\_BallRb.velocity.z);
11. *//拍子旋转角*
12. sensor.AddObservation(Bat.transform.localRotation.x);
13. *//角色的参数*
14. sensor.AddObservation(m\_InvertMult \* (transform.position.x - myArea.transform.position.x));
15. sensor.AddObservation(transform.position.y - myArea.transform.position.y);
16. sensor.AddObservation(m\_InvertMult \* (transform.position.z-myArea.transform.position.z));
17. sensor.AddObservation(-m\_InvertMult \* m\_AgentRb.velocity.x);
18. sensor.AddObservation(m\_AgentRb.velocity.y);
19. sensor.AddObservation(m\_InvertMult \* m\_AgentRb.velocity.z);
20. *//旋转角*
21. var m\_localEularY = transform.localEulerAngles.y;
22. if (invertX)
23. {
24. sensor.AddObservation(m\_localEularY);
25. }
26. else
27. {
28. if(m\_localEularY > 180f)
29. {
30. sensor.AddObservation(m\_localEularY-180f);
31. }
32. else
33. {
34. sensor.AddObservation(m\_localEularY+180f);
35. }
36. }
37. }

### 4.2.2 OnActionRecieved函数

1. *//处理从策略获得的动作，将其应用到游戏角色中*
2. public override void OnActionReceived(ActionBuffers actionBuffers)
3. {
4. var discreteActions = actionBuffers.DiscreteActions;
5. var Axis = discreteActions[3];
6. var dirToGo = Vector3.zero;
7. var RotToGo = Vector3.zero;
8. *//移动方向选择*
9. switch (Axis)
10. {
11. case 1:
12. dirToGo = transform.forward \* moveSpeedX;
13. m\_endurance -= 0.1f;
14. break;
15. case 2:
16. dirToGo = -transform.forward \* moveSpeedX;
17. m\_endurance -= 0.1f;
18. break;
19. case 3:
20. dirToGo = transform.right \* moveSpeedZ;
21. m\_endurance -= 0.1f;
22. break;
23. case 4:
24. dirToGo = -transform.right \* moveSpeedZ;
25. m\_endurance -= 0.1f;
26. break;
27. }
28. float agentDisToBall = Mathf.Abs(Vector3.Distance(transform.position, ball.transform.position));
29. *//跳跃*
30. *// 0-1*
31. var Jump = discreteActions[0];
32. m\_endurance -= 0.3f \* Jump;
33. *//挥拍*
34. *//0-1*
35. var RotateBat = discreteActions[1];
36. m\_endurance -= 0.1f \* Mathf.Abs(RotateBat);
37. *//转身*
38. *//0-2*
39. var RotateBody = discreteActions[2];
40. m\_endurance -= 0.1f \* Mathf.Abs(RotateBody);
41. *//挥拍动作选择*
42. switch (RotateBat)
43. {
44. case 1:
45. RotateBat = 1;
46. break;
47. case 0:
48. RotateBat = -1;
49. break;
50. }
51. *//转身动作选择*
52. switch (RotateBody)
53. {
54. case 1:
55. RotToGo = transform.up;
56. break;
57. case 2:
58. RotToGo = -transform.up;
59. break;
60. }
61. var BatJoint = Bat.GetComponent<HingeJoint>();
62. var spring = BatJoint.spring;
63. *//将挥拍动作应用到角色。*
64. if (!hitting && RotateBat == 1)
65. {
66. hitting = true;
67. spring.targetPosition = 120f;
68. spring.spring = BatSpringForce;
69. spring.damper = 50f;
70. BatJoint.spring = spring;
71. }
72. else if (RotateBat == -1)
73. {
74. hitting = false;
75. spring.targetPosition = 0f;
76. spring.spring = BatSpringForce/2;
77. spring.damper = 100f;
78. BatJoint.spring = spring;
79. }
80. *//旋转实现*
81. transform.Rotate(RotToGo, Time.deltaTime \* 100f);
82. *//跳跃实现*
83. if ( transform.parent.position.y - transform.position.y > -AgentResetLocalY-0.01f )
84. {
85. jumping = false;
86. }
87. if (!jumping && Jump==1)
88. {
89. jumping = true;
90. *//如果改了这个jump力，那么需要改观测值上下限！*
91. m\_AgentRb.AddForce(new Vector3(0f,4f,0f), ForceMode.VelocityChange);
92. }
93. m\_AgentRb.AddForce(new Vector3(dirToGo.x, 0f , dirToGo.z), ForceMode.VelocityChange);
94. *//防止越界*
95. if (invertX && transform.position.x - transform.parent.transform.position.x > m\_InvertMult ||
96. !invertX && transform.position.x - transform.parent.transform.position.x < m\_InvertMult)
97. {
98. transform.position = new Vector3(m\_InvertMult + transform.parent.transform.position.x,
99. transform.position.y,
100. transform.position.z);
101. }
102. }
103. *//跳跃时向跳跃添加ActionMask*
104. public override void WriteDiscreteActionMask(IDiscreteActionMask actionMask)
105. {
106. if(jumping)
107. {
108. actionMask.WriteMask(0, new int[1] { 1 });
109. }
110. }

## 4.3 算法实现

本文中的MADDPG算法由ML-Agents开源项目中的代码改写而来。ML-Agents提供的PythonAPI目前并不完善，在产出pytorch模型后尚不能转化为ML-Agents支持的ONNX格式。另外，在对比算法以及奖励函数时需要保证输出的统计结果在同一张图表中。因此本文对ML-Agents的源码进行了修改，剔除了内置的PPO和SAC算法，对部分源码进行了针对多Agent问题的改动并加入了MADDPG算法。图4.3为ML-Agents训练时部分重要组件的时序图。

图4.3 ML-Agents时序图

程序以Learn.py作为入口。TrainerController是最上层的训练控制器，控制所有子环境的训练过程。SubprocessManager管理各个子环境的训练过程。本文共使用32个独立球场训练Agnet，因此共有32个SubManager管理32个球场。AgentProcessor管理同一个大脑的Agent组。它负责维护每个Agent的轨迹池。该轨迹池用于更新网络。Trainer中包含一个Opt,,,,,,,,,. `+Cimizer，是MADDPG算法的具体实现，它们共同完成了Actor网络和Critic网络的更新。

本文的实现源于ML-Agents源码，对部分源码进行了修改，以适用于MADDPG算法。除了一些细节外，改动较大的部分是AgentProcessor的\_process\_step()以及Optimizer的update ()函数。\_process\_step函数将获得的状态信息转化为联合信息，并将上一步的联合动作、联合信息、奖励与该最新的联合信息合并成一条经验，投入经验回放池。update 函数从trainer中获得从经验池中抽样得到batch，然后使用batch更新Actor网络和Critic网络的参数。

下文将展示\_process\_step和update部分的代码及注释。

### 4.3.1 \_process\_step

1. *# 获取先前记录，组合为一个exp*
2. def \_process\_step(self, step: Union[TerminalStep, DecisionStep], global\_id: str, index: int,Steps :Union[TerminalSteps, DecisionSteps]):
3. *# global\_id集合*
4. action\_global\_agent\_ids=['$0-0', '$0-1', '$0-2', '$0-3', '$0-4', '$0-5', '$0-6', '$0-7', '$0-8',
5. '$0-9', '$0-10', '$0-11', '$0-12','$0-13', '$0-14', '$0-15', '$0-16',
6. '$0-17', '$0-18', '$0-19', '$0-20', '$0-21', '$0-22', '$0-23', '$0-24',
7. '$0-25', '$0-26', '$0-27', '$0-28', '$0-29', '$0-30', '$0-31' ]
8. terminated = isinstance(step, TerminalStep)
9. *# 获取已存储的动作和decision\_step*
10. stored\_decision\_step, idx = self.last\_step\_result.get(global\_id, (None, None))
11. stored\_take\_action\_outputs = self.last\_take\_action\_outputs.get(global\_id, None)
13. *# 提前取出之前的ob*
14. if stored\_decision\_step is not None:
15. stored\_ob=stored\_decision\_step.obs[0]
17. *# 更新存储的step*
18. if not terminated:
19. *# Index is needed to grab from last\_take\_action\_outputs*
20. self.last\_step\_result[global\_id] = (step, index)
22. *# 如果有保存的动作和step，则添加经验*
23. if stored\_decision\_step is not None and stored\_take\_action\_outputs is not None:
24. *# 构建联合状态*
25. all\_obs=[]
26. for g\_id in action\_global\_agent\_ids:
27. if g\_id not in self.last\_step\_result:
28. ob=np.asarray([-6.0,-1.5,1.877,0,0,0,0,7,-6.3,0,0,0,0,90,-7,-6.3,0,0,0,0,90],dtype=np.float)
29. all\_obs.append(ob)
30. else:
31. *#当处理的步就是遍历中的gid时，将最新ob插入*
32. if g\_id==global\_id:
33. ob=stored\_ob
34. all\_obs.append(ob)
35. else:
36. ob=self.last\_step\_result.get(g\_id, (None, None))[0].obs[0]
37. all\_obs.append(ob)
38. *# 构建联合奖励*
39. all\_rewards = []
40. for g\_id in action\_global\_agent\_ids:
41. if g\_id not in self.last\_step\_result:
42. reward=0.0
43. all\_rewards.append(reward)
44. else:
45. *# 当处理的步就是遍历中的gid时，将最新ob插入*
46. if g\_id == global\_id:
47. reward = step.reward
48. all\_rewards.append(reward)
49. else:
50. reward = self.last\_step\_result.get(g\_id, (None, None))[0].reward
51. all\_rewards.append(reward)

54. if self.policy.use\_recurrent:
55. memory = self.policy.retrieve\_memories([global\_id])[0, :]
56. else:
57. memory = None
59. *# 当前step是否终止*
60. done = terminated
61. interrupted = step.interrupted if terminated else False
62. *# Add the outputs of the last eval*
63. stored\_actions = stored\_take\_action\_outputs["action"]
64. action\_tuple = ActionTuple(
65. continuous=stored\_actions.continuous[idx],
66. discrete=stored\_actions.discrete[idx],
67. )
69. *# 构建action\_mask*
70. action\_masks=[]
71. for i in range(0,len(stored\_decision\_step.action\_mask)):
72. action\_mask\_list = []
73. for g\_id in action\_global\_agent\_ids:
74. if g\_id not in self.last\_step\_result:
75. action\_mask\_list.append(np.zeros(stored\_decision\_step.action\_mask[i].shape[0]))
76. else:
77. action\_mask\_list.append(self.last\_step\_result[g\_id][0].action\_mask[i])
78. action\_masks.append(np.vstack(action\_mask\_list))
80. *# 构建经验*
81. experience = AgentExperience(
82. obs=all\_obs,
83. reward=all\_rewards,
84. done=done,
85. action=stored\_actions,
86. action\_probs=None,
87. action\_mask=action\_masks,
88. interrupted=interrupted,
89. memory=memory,
90. )
92. *# Add the value outputs if needed*
93. self.experience\_buffers[global\_id].append(experience)
94. self.episode\_rewards[global\_id] += step.reward
95. if not terminated:
96. self.episode\_steps[global\_id] += 1
98. *# 存储轨迹*
99. if (
100. len(self.experience\_buffers[global\_id]) >= self.max\_trajectory\_length
101. or terminated
102. ):
103. next\_all\_obs=[]
104. for g\_id in action\_global\_agent\_ids:
105. if g\_id not in self.last\_step\_result:
106. ob = np.asarray(
107. [-6.0, -1.5, 1.877, 0, 0, 0, 0, 7, -6.3, 0, 0, 0, 0, 90, -7, -6.3, 0, 0, 0, 0, 90],
108. dtype=np.float)
109. next\_all\_obs.append(ob)
110. else:
111. ob = self.last\_step\_result.get(g\_id, (None, None))[0].obs[0]
112. next\_all\_obs.append(ob)
113. trajectory = Trajectory(
114. steps=self.experience\_buffers[global\_id],
115. agent\_id=global\_id,
116. next\_obs=next\_all\_obs,
117. behavior\_id=self.behavior\_id,
118. )
119. for traj\_queue in self.trajectory\_queues:
120. traj\_queue.put(trajectory)
121. self.experience\_buffers[global\_id] = []
122. if terminated:
123. *# Record episode length.*
124. self.stats\_reporter.add\_stat(
125. "Environment/Episode Length", self.episode\_steps.get(global\_id, 0)
126. )
127. self.\_clean\_agent\_data(global\_id)

### 4.3.2 update

1. @timed
2. def update(self, batch: AgentBuffer, num\_sequences: int) -> Dict[str, float]:
3. """
4. Performs update on model.
5. :param batch: Batch of experiences.
6. :param num\_sequences: Number of sequences to process.
7. :return: Results of update.
8. """
9. *# learning\_rate随时间减少,保证收敛*
10. decay\_lr = self.decay\_learning\_rate.get\_value(self.policy.get\_current\_step())
11. gamma=self.gamma
12. *# 从batch中获得奖励和回报*
13. returns = {}
14. rewards = {}
15. for name in self.reward\_signals:
16. rewards[name] = ModelUtils.list\_to\_tensor(
17. batch[RewardSignalUtil.rewards\_key(name)]
18. )
19. returns[name] = ModelUtils.list\_to\_tensor(
20. batch[RewardSignalUtil.returns\_key(name)]
21. )
22. *# 获取状态值*
23. n\_obs = 32
24. current\_obs = ObsUtil.from\_buffer(batch, n\_obs)
25. next\_obs = ObsUtil.from\_buffer\_next(batch, n\_obs)
26. *# 将numpy转换为tensor*
27. current\_obs = [ModelUtils.list\_to\_tensor(obs) for obs in current\_obs]
28. next\_obs = [ModelUtils.list\_to\_tensor(obs) for obs in next\_obs]
29. current\_obs=torch.stack(current\_obs)
30. next\_obs=torch.stack(next\_obs)
31. *#2048 32 21*
32. current\_obs=torch.transpose(current\_obs,0,1)
33. next\_obs=torch.transpose(next\_obs, 0, 1)
34. *# 获取action\_mask*
35. act\_masks = ModelUtils.list\_to\_tensor(batch[BufferKey.ACTION\_MASK])
36. memories = [
37. ModelUtils.list\_to\_tensor(batch[BufferKey.MEMORY][i])
38. for i in range(0, len(batch[BufferKey.MEMORY]), self.policy.sequence\_length)
39. ]
40. if len(memories) > 0:
41. memories = torch.stack(memories).unsqueeze(0)
42. *# 获取当前Action*
43. actions\_current = AgentAction.from\_buffer(batch)
44. action\_current=actions\_current.discrete\_tensor
45. *# 计算目标Action*
46. action\_list=[]
47. for i,\_ in enumerate(next\_obs):
48. actions\_target=self.policy.actor\_critic.get\_action\_stats\_target(inputs=[next\_obs[i]],masks=act\_masks[i])
49. *#32\*4*
50. actions\_target=torch.squeeze(actions\_target[0].discrete\_tensor)
51. action\_list.append(actions\_target)
52. action\_2d\_target=torch.stack(action\_list)
53. *# 构建目标Q值*
54. Q\_target\_all=[]
55. for i, \_ in enumerate(next\_obs):
56. Q\_target=self.policy.actor\_critic.critic\_target(inputs=[next\_obs[i]],actions=action\_2d\_target[i])[0]['extrinsic']
57. Q\_target\_all.append(Q\_target)
58. Q\_target\_all=torch.stack(Q\_target\_all)
59. *# y=r+gammaQ*
60. y=rewards['extrinsic']+gamma\*Q\_target\_all
61. *# 计算当前Q值*
62. Q\_current\_all=[]
63. for i, \_ in enumerate(next\_obs):
64. Q\_current=self.policy.actor\_critic.critic(inputs=[current\_obs[i]],actions=action\_current[i])[0]['extrinsic']
65. Q\_current\_all.append(Q\_current)
66. Q\_current\_all=torch.stack(Q\_current\_all)
67. *# 计算loss*
68. loss=(y-Q\_current\_all).sum(0)/num\_sequences
69. *# 更新learing\_rate*
70. ModelUtils.update\_learning\_rate(self.optimizer, decay\_lr)
71. *# optimizer更新网络参数*
72. self.optimizer.zero\_grad()
73. loss.backward()
74. self.optimizer.step()
75. update\_stats = {
76. "Policy/Learning Rate": decay\_lr,
77. }
78. for reward\_provider in self.reward\_signals.values():
79. update\_stats.update(reward\_provider.update(batch))
80. return update\_stats

# 5 实验结果

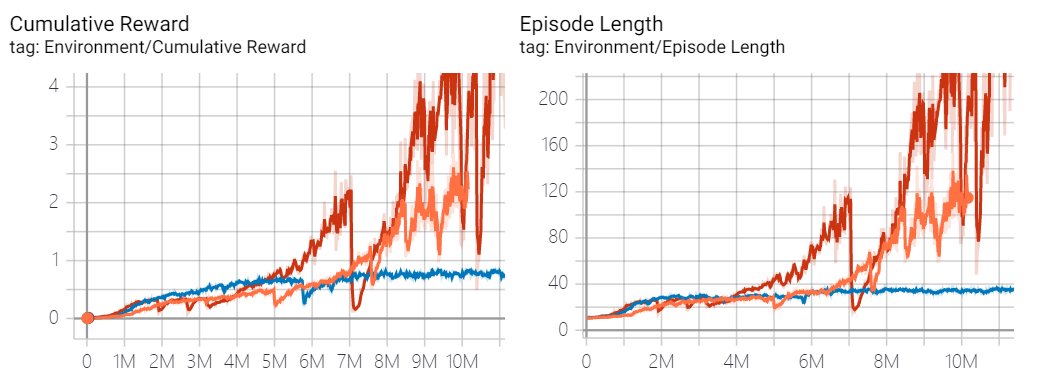
本文将展示两个实验以及实验结果。第一个实验是为了测试MADDPG算法在本文设计的多Agent游戏中的表现，比较对象为ML-Agents内置的PPO算法。第二个实验是以MADDPG为强化算法，不同奖励函数之间的学习效率对比。

## 5.1 30时间步瓶颈

30时间步瓶颈指在这个打球游戏的训练中经常出现的局部最优值，它代表Agent在训练中遇到的一个瓶颈，只能发球，但是却接不到球，从回合开始到发球后落地大约30时间步。Agent很容易发球的原因是球会在固定区域生成，生成后不会受到重力以外的其它力，因此球的坐标和速度组成的空间较小，Agent只需随机行动一定次数就能轻易触碰到球。但是在回击时情况有了很大不同。由于球在空中的速度和坐标组成的空间非常大，可能性很多，因此Agent想通过抽样的方式遍历所有可能性相对而言更加困难，造成了训练的瓶颈。本文将着重观察各训练过程在该瓶颈处的表现。

## 5.2 算法实验结果

本节采用了上文3.2中介绍的状态、动作以及3.2.3.1中原始奖励函数进行算法实验，以Cumulative Reward(平均累积奖励，即平均每回合获得的奖励) 和Episode Length（回合长度）进行比较，结果如图5.1所示。

图5.1 实验结果总体图

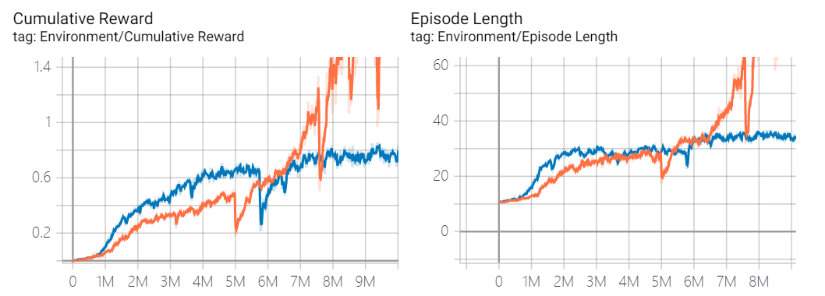
图5.1中共有三条线，橙色及蓝色为PPO算法，红色为MADDPG算法。PPO算法是目前常用且性能较强的强化学习算法，被ML-Agents内置作为默认算法。本文将PPO算法与MADDPG算法比较旨在突出MADDPG算法处理多Agent问题上的优越性。

图5.2 PPO算法实验图

在结果图5.2中，展示了PPO算法的两个实验过程，橙色是较成功的训练过程；而蓝色是训练失败的过程，它收敛在了局部最优值上。值得一提的是，在Episode Length中，蓝线收敛在了30时间步瓶颈。通过观察训练结果可以发现由蓝色训练过程训练出的Agent与其击球更愿意选择躲避球，因为它“发现”击球后很大概率会打出界并受到惩罚，因此不如不击球，仅靠50%概率获得发球权发球获得奖励更加划算。显然，这不是我们期望获得的结果。在PPO算法的训练中，经常会出现蓝线展示的现象，很难收敛到真正的最优策略。本文还提供了PPO算法的成功训练过程，如图5.2中橙线所示。可以发现，该训练过程在约650万步时突破了上文提到的30时间步瓶颈，能够在发球后再次击球。

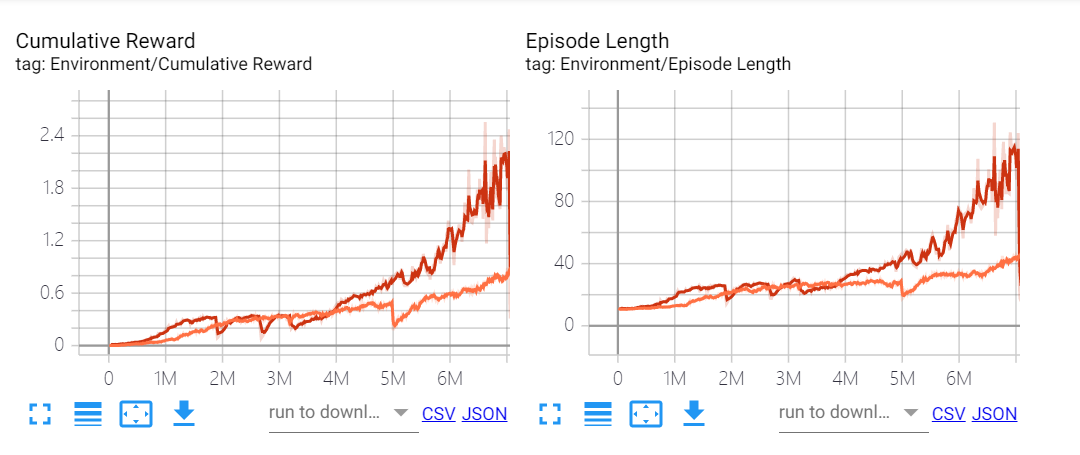


图5.3 PPO-MADDPG算法对比图

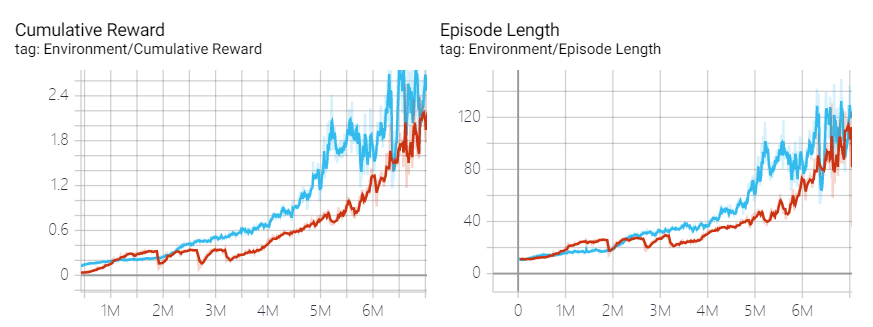
图5.3中红线代表MADDPG算法的训练过程，橙线则是上文提到的PPO算法的成功版本。与PPO算法相比，MADDPG算法非常容易收敛到最优策略，因此本文没有展示其失败的训练过程。比较图5.3的红线与橙线，可以看出MADDPG算法在大约400万步突破了30时间步的瓶颈，与PPO算法的650万步相比，算法的训练效率提高了约165%。突破瓶颈后，平均累计奖励和回合时间迅速增长，代表球的滞空时间增长，策略在不断优化。显然，MADDPG算法在本游戏的表现上优于PPO算法。

## 5.3 奖励函数实验结果

本节采用MADDPG算法比较3.2.3节提及的两种奖励函数。

强化学习的学习本质上是在优化策略。在每一时间步，Agent会根据观测到的状态、执行的动作以及奖励值评估在该状态执行这个动作的好坏程度。然而因为原始版本的奖励函数在各个状态-动作对出现非零奖励的可能性非常小，即奖励比较稀疏，所以奖励函数很难对Agent的策略进行指导，大多数情况下Agent的行为都是随机的。另外，Agent仅仅在触碰到球才会获得奖励，在触碰前的一切动作都是没有奖励的。Agent很难将击球时刻以外的其他动作与击球相联系。这种情况被称为奖励延迟性。

为了解决上述问题，改进版的奖励函数使用了奖励分解的思想，将单独目标拆分为多个容易达成的子目标，使微小的奖励遍布在多数状态-动作空间中。具体而言，就是将击球这一大目标分解为靠近球、击球。为了验证该设计方案的优越性，本文进行了奖励函数的对比试验。

 图5.4奖励函数对比图

比较结果如图5.4所示，蓝线为改进奖励函数，红线为原始奖励函数。经观察发现改进奖励函数版本的训练过程能够在约260万步突破30时间步瓶颈，而原始奖励函数版本的训练过程需要400万步才能突破，其效率提高了约153%。另外，改进版本的训练过程比原始版本的训练过程更加稳定，不会出现过多抖动现象。

# 6 总结与展望

本文以自制打球游戏为对象，自行设计了强化学习训练游戏AI的整个过程。首先，本文介绍了强化学习目前的研究现状以及本文设计的理论背景，并引入了多Agent问题。然后，本文设计了一个简单的打球游戏，游戏中内置了马尔科夫决策过程必需的几个基本要素，为强化学习提供了学习环境。接下来本文介绍了游戏以及MADDPG算法的实现。它们分别基于Unity和ML-Agents源码架构。最后，本文对不同算法、不同奖励函数分别进行了对比试验，验证了MADDPG算法在多Agent问题中的优越性，并给出了奖励函数的一项设计经验。

目前，如何训练出更强的游戏AI已经成为强化学习的热门研究领域。许多研究团体，如DeepMind、OpenAI等，在DOTA、星际争霸2等复杂的游戏中成功训练出了足以挑战职业选手的游戏AI。在游戏中，很多情况下Agent不能观测到全信息，不能充分了解环境，使解决多Agent问题的难度激增。如何使Agent之间充分沟通感知环境成为了将来在多Agent问题中需要解决和优化的问题。除此之外，游戏AI最终应服务于人，所以也应该被人适应。如何训练出类人的游戏AI也是一个亟待解决的问题。

# 参考文献

1. Minsky M. Steps toward artificial intelligence[J]. Proceedings of the IRE, 1961, 49(1): 8-30.
2. Barto A G , Sutton R S , Brouwer P S . Associative search network: A reinforcement learning associative memory[J]. Biological Cybernetics, 1981, 40(3):201-211.
3. Sutton R S. Temporal credit assignment in reinforcement learning[D]. University of Massachusetts Amherst, 1984.
4. Dayan P. The convergence of TD (λ) for general λ[J]. Machine learning, 1992, 8(3-4): 341-362.
5. Watkins C J C H, Dayan P. Q-learning[J]. Machine learning, 1992, 8(3-4): 279-292.
6. Peng J, Williams R J. Incremental multi-step Q-learning[M]. Machine Learning Proceedings 1994. Morgan Kaufmann, 1994: 226-232.
7. Werbos P J, Miller W T, Sutton R S. A menu of designs for reinforcement learning over time[J]. Neural networks for control, 1990: 67-95.
8. Tadepalli P, Ok D K. Model-based average reward reinforcement learning[J]. Artificial intelligence, 1998, 100(1-2): 177-224.
9. Riedmiller M. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method[C].European Conference on Machine Learning. Springer, Berlin, Heidelberg, 2005: 317-328.
10. Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.
11. Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. nature, 2015, 518(7540): 529-533.
12. Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning[C].roceedings of the AAAI Conference on Artificial Intelligence. 2016, 30(1).
13. Schaul T, Quan J, Antonoglou I, et al. Prioritized experience replay[J]. arXiv preprint arXiv:1511.05952, 2015.
14. Tampuu A, Matiisen T, Kodelja D, et al. Multiagent cooperation and competition with deep reinforcement learning[J]. PloS one, 2017, 12(4): e0172395.
15. Foerster J N, Assael Y M, de Freitas N, et al. Learning to communicate to solve riddles with deep distributed recurrent q-networks[J]. arXiv preprint arXiv:1602.02672, 2016.
16. Lowe R, Wu Y, Tamar A, et al. Multi-agent actor-critic for mixed cooperative-competitive environments[J]. arXiv preprint arXiv:1706.02275, 2017.

|  |
| --- |
|  |