内部资料
注意保存

基于强化学习的联合作战模拟推演
高级研修班培训资料

中国指挥与控制学会

中国·北京

二〇二一年十月

目　　录

强化学习智能体的快决策与慢思考

随着深度学习、强化学习等技术的不断发展，涌现出了一批诸如Alpha Go、Alpha Star、Alpha DogFight等智能博弈技术应用成果，为解决传统任务规划专家经验依赖性强、应对不完整信息的能力弱、动态临机调整困难等难题提供了一种新的解决方案。以强化学习、蒙特卡罗树搜索等方法构建智能体是当前智能规划研究的热点之一，其优势是可利用机器强大的算力，实现从态势到行动的端到端学习，从而实现不完全信息条件下智能决策能力的自主学习，然而目前算法体现出泛化性差、环境适应能力弱、数据利用率低等缺陷，其核心问题在于智能体形成的是一种应激反应式的快决策，缺乏对过程数据深度理解的慢思考能力，本报告从规则驱动的人工视角、数据驱动的机器视角、群智涌现的复杂系统视角三个角度，介绍建立慢思考能力的智能体技术思路，研讨强化学习技术研究的发展方向。

“墨子”联合作战推演系统及应用

一、系统总体情况

（一）业务领域

公司总部在北京，在长沙、南京、武汉、西安设有分公司或办事处。目前公司员工80多人，主要从事墨子联合作战推演系统研发和推广应用，为军内和军工单位提供作战推演仿真研究和军事人工智能研究等业务领域的一体化解决方案，辅助客户单位形成作战实验研究能力。

（二）系统定位

墨子联合作战推演系统是一款覆盖陆、海、空、火、天、电全域联合作战的自主可控推演系统。

（1）系统层级定位在战术级（任务级），兼顾战役级；

（2）系统应用定位在开环推演，兼顾闭环仿真；既支持预先制定任务计划，也支持“人在回路”对兵力进行干预和调整

（3）系统仿真定位平台级，模型分辨率到传感器、挂架（武器）、通信、推进系统、信号特征等组件。

（三）系统组成

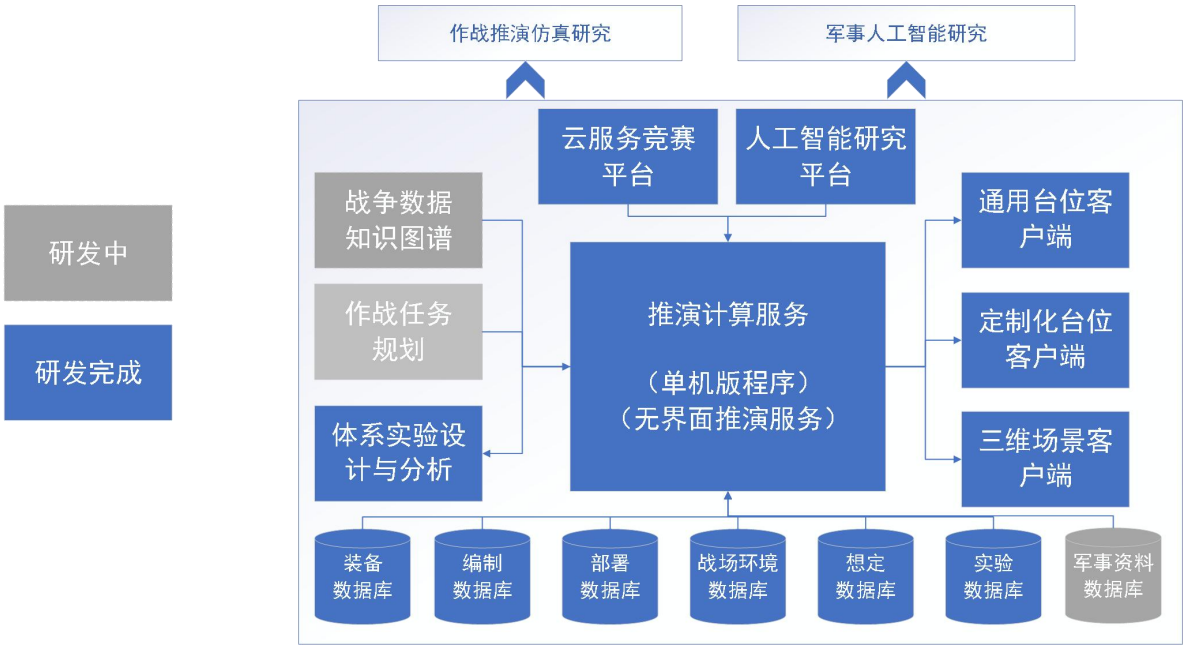


图 1 系统组成

（四）应用模式

（1）集中式模式

单机使用，主要用于个人开展分析实验，可用于作战方案评估、战法研究。

（2）分布式模式

多机使用，主要用于团队对抗博弈，支持协同想定制作和分组博弈对抗推演等需求，支持实验室环境中局域网部署方式。

（3）人工智能研究模式

集成了 Ray 分布式训练框架和 PPO、DDPG、A3C 等开源人工智能算法，用于支持开展基于军事规则的人工智能研究和开展基于强化学习的军事人工智能研究。

（4）高分辨率模型集成模式

根据应用需求，用于与其他高分辨率模型进行集成，空 X 基地针对防空反导研究需要，集成了高分辨率地空导弹运动模型、探测模型、地空导弹对目标毁伤模型和武器对地空导弹营毁伤模型；X 院 X 部针对打

击需求，集成了高精度弹道导弹模型和临近空间武器模型；XX 研究院项目中，正在集成 YJ-2X、YJ-1X 等系列打击武器模型。

（5）异构系统互联模式

用于与其他实装指控系统和专业仿真系统进行互联，比如：编队指控系统、岸基指挥所系统、XX 任务规划系统、水声探测计算系统、电磁环境仿真系统、联合情报系统、三维显示系统、Rhapsody 等体系结构设计系统等互联。

（6）定制化台位模式

用于特定作战力量的态势掌握和定制化操控，可以开发编队及组网任务规划、卫星地面站测控、导弹测发控台位、高超飞行器作战指挥链路台位、无人机任务操作台位。已经在中航 XX 所和中船 X 院实现联指、航母编队、多机种（歼击机、轰炸机、预警机、电子战飞机）的定制化台位。

二、系统功能特点

（一）具有较完备的分辨率适中的模型体系

支持陆军、海军、空军、火箭军、战略支援部队等多个军兵种的装备与作战模型。作战平台类型涵盖作战飞机、卫星、雷达、机场、高超声速导弹、防空高炮、新型无人机等现代战争中可能涉及到的武器平台，以及相关的民用平台。

（二）具有作战推演所需的大量基础数据

包含武器装备数据库、作战部署数据、部队编制数据、防空识别区等区域数据、影像数据。

（1）基础数据--武器装备数据库

基于简氏防务周刊、海军分析网等开源渠道和技术分析，收集整理了包括美、俄、中、日、英、法等军事强国，以及印度、台湾地区、朝鲜、韩国、越南、菲律宾等国家或地区（共计 138 个）的装备数据。

（2）基础数据—作战部署数据库

多个国家的作战部署数据，包括机场、港口、发射阵地、预警雷达站等，共计 6000 多条。

（3）基础数据—部队编制数据

多个国家的部队编制数据，包括海军舰队、陆军旅、航空兵旅等，总计 2956 条。

（4）基础数据—区域数据

多个国家的防空识别区、领海基线、国际航线、封锁区、禁航区等等。

（5）基础数据—影像数据

可以根据需要，下载和部署关心区域的高精度影像数据，已经整理了日本、美军驻亚太地区、台湾地区、朝鲜方向、东海方向、台海方向、南海方向、中印边境等区域高清影像。

（三）研究性想定案例库丰富

针对相关单位的应用需求，在相关军事想定的基础上，设计并实现了对应作战仿真想定，方便对相关热点问题开展深化研究工作。系统已经积累了案例想定近 100 个，并且自带功能演示想定近 100 个。

（四）支持作战任务、作战条令、兵力操控和规则建模

一是系统支持制定多类作战任务计划。支持的一级作战任务括包括截击、护航、巡逻、支援、转场、投送等。系统支持的二级作战任务包括：空中截击、对陆打击、对海突击、对潜攻击、空战巡逻、反水面战巡逻、反潜战巡逻、压制敌防空巡逻、海上控制巡逻等。

二是系统支持条令规则设置，可以在推演方、任务、编队、作战单元、航路点等不同层级设置交战规则。

三是系统支持多类兵力操控命令操作。主要包括航线管理、攻击方式控制、空中作战控制、反潜作战控制、平台操作控制、作战编组控制、

作战条令设置、作战任务设置取消、补给控制等。

（五）支持作战任务规划

战区任务规划和无人机任务规划，一是陆基型、空基型的多组投放/回收模式；二是无人机集群集结区域及出动/投放/任务/回收等所用时间可调。

（1）集群出动/回收地点、方式选择

（2）集群编组类型设置

（3）集群阵型设计

（4）集群航路点设置

（5）集群作战目标设置

（6）任务执行能力检验

（7）无人机之间最大通信距离

（8）有人无人协同

（六）体系实验设计与分析

体系实验设计工具突出设置面向作战的实验参数，支持兵力结构规模、战场环境参数、兵力行动计划、作战条例规则、作战区域及航线等体系级作战实验参数和装备性能参数设计，聚焦军事问题、解决方案空间过大等问题。

分布式并行仿真控制软件用于控制多个作战实验方案在有限的计算资源中仿真运行，包括分发实验方案，开展并行仿真计算，监控计算节点状态，调度计算资源，提高计算资源的利用和仿真运行的效率。

体系实验分析由评估数据管理模块、基础指标构建模块、指标体系模板构建模块、实验评估分析模块等组成展示。支持开展针对指标体系的多方案对比分析、追溯分析、比例分析、探索性分析等。

（七）态势显示和数据输出内容丰富

态势显示由地理信息、军标、作战编组、威力范围、交战关系、任

务图元信息、作战单元信息、感知目标等功能模块组成，平台支持通过TacView进行三维显示。

系统支持将探测、交战、毁伤等关键事件和实体轨迹、战场态势的数据导出，支持CSV、XML等格式，基于输出的数据，可以开展快速复盘分析和基于指标体系的效能评估工作。

（八）自主可控

从架构、代码到接口、数据，全部自主可控，已经在国产操作系统、国产CPU、国产数据库上实现部署和应用。

（1）推演服务端（仿真引擎及模型）： QT C++

（2）推演客户端：QT C++、OsgEarth；

（3）基础数据管工具：Java、MySQL；

（九）系统特色总结

（1）系统模型覆盖全，支持战役级、任务级联合作战推演研究；

（2）系统装备数据、部署数据、编制数据、区域数据积累丰富，且甲方可以对数据进行校验更新；

（3）关注的作战场景或作战方向，有大量研究性想定；

（4）支持开展智能蓝军等军事人工智能研究，生成训练数据、训练智能算法、验证智能化程度；

（5）系统使用门槛低，科研干部、参谋人员简单培训就会操作使用，每年1万多人参加全国兵棋推演大赛。

三、主要应用领域

（一）作战概念研究

1、概念

作战概念研究，是对作战活动的本质特点进行抽象和概括，是人们对作战活动感性认识上升到理性认识的过程，是研究作战问题、创新军

事理论、设计未来战争的核心内容，其本质是搞清“打什么仗、怎么打仗”的问题，对推进军事斗争准备和部队建设具有基础性、先导性作用。

当前，作战概念研究，不仅要分析支撑作战概念运用的现有武器装备、保障系统等条件，还要重点分析人工智能技术、无人技术、超能技术、深海技术、生物技术等高新技术快速发展，可能带来的新作战手段及其对作战编成编组、指挥决策、攻防模式的影响。无人化、智能化是装备发展和作战运用的趋势！

作战概念研究主要有作战推演评估、专题模拟评估和实兵演练评估等手段。

（1）作战推演评估：依据新作战概念，设计其主要行动样式和基本程序，依据设定的作战规则进行作战行动推演。

（2）专题模拟评估：依据新作战概念设计作战需求、作战行动、主要战法等专题，采取模拟仿真技术手段，对专题内容进行模拟。

（3）实兵演练评估：根据新作战概念运用，设计逼真的战场环境，协调相关部队对新作战概念的行动、战法等进行检验，或者结合重大演习活动设计新作战概念专题演练科目、课题。

2、对支撑系统要求

一是对无人机、无人艇、无人潜航器、无人车等无人化、智能化装备能够快速建模；

二是方便对新型装备的作战运用方式进行灵活设置；

三是能够外接或支持客户自己开发核心控制算法（比如无人蜂群规划算法等）；

四是支持单机版本，能够方便演示。

3、典型想定案例

一是 1-南海反航母作战无人机蜂群；

二是 14-直升机搭载无人机协同打击地面目标；

三是 52-分布式杀伤作战概念推演；

四是95-无人智能陆战装备（无人营）推演；

五是54-未来海上典型场景设计等等。

（二）作战/演习方案评估研究

1、概念

作战方案亦称作战预案，指在预判敌方兵力部署及可能采取行动的基础上，根据首长决心拟制的对作战进程和战法的设想，内容通常包括情况判断结论，上级企图和本部队任务，友邻任务及作战分界线，各部队的编成、配置和任务，作战阶段划分，各阶段情况预想及处置方案，保障措施、指挥的组织等。

作战方案的评估指对作战方案的可行性、风险度、作战效益等进行的评价和估量，是确保形成最佳决策的必要环节。作战方案的评估方法可以分为军事运筹分析法和仿真推演实验法。

作战方案推演评估，是作战筹划的必要环节，是一项专业性很强的军事预实践。主要流程包括推演数据准备、行动方案细化、建立评估指标体系、组织实施推演、分析研究问题、对比优选方案六个阶段对作战方案实施评估。

2、对支撑系统要求

一是对系统模型和数据可信度要求较高，一般会成立专门数据组负责数据校验；

二是会采用单机版和分布式版结合使用，单机版研讨方案、分布式版本对方案进行博弈对抗验证；

三是对系统任务规划功能有要求。

3、由于作战方案和演习方案密级较高，大部分在客户单位制作：

（1）军委层面重大演习演练任务；

（2）战区层演习演练任务；

（3）战区军兵种演习演练任务；

（4）军兵中试验训练基地专项演习演练任务；

（三）战术战法创新研究

1、概念

战术是指导思想，战法是达成战术目标的手段。战术是指导和进行战斗的方法。主要包括：战斗基本原则以及战斗部署、协同动作、战斗指挥、战斗行动、战斗保障、后勤保障和技术保障等。按基本战斗类型分为进攻战术和防御战术；按参加战斗的军种、兵种分为军种战术、兵种战术和合同战术；按战斗规模分为兵团战术、部队战术和分队战术。

战法是作战的方法。指战争的策略和技能，是指挥员主观能动性和指挥艺术的充分体现，灵活巧妙选择和运用战法，能以小的代价获取大的胜利。对于战法而言，如果离开特定的环境、对手、运用时机等笼统地去研究，就容易在“四六句”上打转转，成为脱离实际的文字游戏，甚至是无价值的凭空幻想。

战术战法创新步骤：一是厘清作战任务；二是分析作战需求；三是找准对抗焦点；四是设计行动构想；五是评估行动成效；六是提炼战法创意。其中，三、四、五需要依托仿真推演系统开展工作。

2、对支撑系统要求

一是一般用单机版和分布式版结合使用，单机版研讨战术战法、分布式版本对战术战法进行博弈对抗验证；

二是要求系统具备“人在回路” 控制功能，能够对过程进行干预。

3、典型想定案例

一是 42-无人机引导巡飞弹打击战法；

二是 91-USV 无人艇打击宙斯盾驱逐舰研究想定；

三是 43-潜射无人机隐蔽侦察引导打击战法；

四是 65-无人潜航器混合编组伏击潜艇战法；

五是 100-打击蓝方航母编队哨舰预警机作战战法；

六是 12-苏-35 飞机小体系编队攻击宙斯盾驱逐舰战法。

（四）装备体系效能评估研究

1、概念

武器装备体系是按照建设信息化军队、打赢信息或战争的总体要求，为发挥最佳的整体作战效能，而由功能上相互联系、性能上相互补充的各种武器装备系统按一定结构综合集成的更高层次的武装装备系统。

武器装备体系的效能是指武器装备体系完成规定作战任务的程度，虽然武器装备体系要完成的任务是明确的，但这个任务往往包含大量子任务。

2、对支撑系统要求

一是需要体系实验设计与评估工具，支持开展实验方案设计、评估指标体系构建；

二是需要支持大样本蒙特卡洛实验和实验数据采集分析；

三是支持快速制定装备体系运用方式（固化战法，对比研究不同装备体系方案效能变化趋势）。

3、典型想定案例

一是 8-XXX 新型号体系对抗想定（体系贡献率评估）；

二是 51-XX 弹道导弹打击 T 岛北部想定；

三是 53-XX 编队典型作战任务和行动想定；

四是 76-XX 无人作战飞机方案选优。

（五）指挥员谋略训练与实践教学

1、概念

指挥员谋略训练不同于流程训练。谋略是指挥员观察力、分析判断能力、思维能力、预见能力、应变能力等的综合。指挥员谋略训练最终体现在创新多种战法，提升指挥能力。

实践教学主要是指军事院校等在开设的教学课程中，如任务规划、武器装备系统概论、导弹武器作战使用、经典战例研究等课程，需要学员将所学理论部分应用到实践中。

2、对支撑系统要求：

一是根据应用场景不同，可能需要单机版、分布式版本和云服务竞赛平台版本（实践教学中以云服务竞赛平台为主）；

二是对系统操作要求简单，门槛低，入门快，方便指挥员和学员使用；

三是系统要求易安装、易部署，技术保障要求低。

3、谋略训练与实践教学大部分在客户单位制作，公司提供技术支持：

一是国防大学、国防科技大学等综合性院校；

二是海军指挥学院、空军指挥学院、火箭军指挥学院等指挥类军事院校；

三是海军工程大学、空军工程大学（创客大赛）、空军航空工程大学等技术类军事院校。

（六）军事人工智能研究

1、概念

近年来，将人工智能技术与兵棋推演技术的有机结合是军事人工智能研究热点，将人工智能技术应用于兵棋推演，利用人工智能算法实现指挥员的学习与推理，使军事智能体具备思维、学习和问题求解能力，军事人工智能可广泛应用于态势认知、筹划决策、平台控制和军事训练等领域。

2、重点解决问题

一是为开发军事人工智能算法提供平台（墨子智能体开发训练平台）。

二是为开展军事人工智能算法研究提供训练数据（目前我军实战数据和训练数据少，仿真推演是比较接近实战的手段）；

三是为开展军事人工智能算法验证提供对抗博弈环境（智能体博弈、人机（智能体）对抗博弈等）。

3、典型应用项目

一是XX无人智能化装备作战研究；

二是红蓝自博弈智能体研究；

三是 XX 级对抗博弈训练支撑平台；

四是基于强化学习的智能 T 军；

五是全国军事人工智能博弈赛（2020 年，2021 年）；

等等；

四、AI 平台研发情况

（一）开发训练平台概述

墨子联合作战智能体开发平台是一款以墨子联合作战推演系统为环境，内置装备参数库、作战规则库和基础算法库，支持开展强化学习和军事规则等智能体开发、训练和应用的系统，广泛应用于战术战役级态势认知、筹划决策、平台控制和军事训练等领域。

（二）开发训练平台总体框架

系统架构采用分布式学习架构设计，由软硬件算 力支撑层、智能体训练资源层、智能体训练层、智能体建模层和智能体应用层组成，系统总体架构如下图所示。 能够有效支撑联合作战战场环境中的感知智能、人机协同、群体智能协作与对抗智能体的高效训练。

核心思路：集成人工智能领域先进成果，解决军事智能应用领域问题！

1、通用强化学习平台

采用人工智能研究领域通用性较强的先进技术实现，能够兼容市场上主流人工智能训练平台，为联合作战智能体开发训练提供专业的训练框架及算法支撑。

2、训练资源

平台针对军事人工智能开发的特殊性，除了深度强化学习算法库、神经网络模型组件库外，还提供了装备数据库、基础规则库、知识图谱库（研发中）。

3、智能体训练

平台提供智能体交互模块、行为树推理引擎等。交互包括仿真环境控制指令和状态，训练想定的运行指令和状态，兵力控制指令和状态。行为树推理用于规则智能体开发。

4、智能体建模

平台提供态势获取建模、指挥控制建模、评估指标建模、军事规则建模和基础算法模块。

（1）态势获取建模：提供固定目标、感知目标、单元状态、编队状态、任务状态、规则状态等状态参数的建模，为智能体创建状态空间。

（2）指挥控制建模：提供作战任务、航线规划、区域规划、条令规则、单元控制、编队控制等指挥控制模型的建模，为智能体创建动作空间。

（3）评估指标建模：提供行动得分以及侦察预警、指挥控制、打击效果、战损消耗、毁伤状态等指标建模，为智能体创建奖励机制。

（4）军事规则建模：提供对陆打击、防空反导、合同反潜、防空压制、空中预警、电子支援等不同作战样式下的任务规则模型，用于构建基于军事规则和经验知识的行动规则模型，与强化学习模型相结合，可在宏观层面优化智能体模型的学习方向和作战效果。

（5）基础算法模块：提供探测能力分析、打击能力分析、坐标系转换、路径寻优、通视性分析等基础算法，为智能体构建提供基础性算法支撑。

（三）开发训练平台特点

一是系统内置大量军事规则和基础算法，方便快速构建满足特定需求的人工智能体；

二是既支持强化学习智能体开发，也支持军事规则智能体开发，还支持两者融合开发；

三是系统既提供任务级开发接口，也提供实体级（微操）接口，方

便开展不同层次智能体开发训练；

四是系统支持开展联合作战背景下的多种作战样式、多种作战场景的智能体开发和训练；

五是系统支持分布式集群训练、大规模数据采集和高性能预测推断；

六是支持开展分层多智能体开发训练，为适应多种应用场景的智能体开发进行技术探索等。

（四）开发训练平台发展考虑

一是开发墨子联合作战知识图谱平台，实现开发训练平台与知识图谱平台融合发展。

知识图谱系统实现对装备数据、部署数据、 编制数据、想定数据、实验数据、 军事情报数据等数据统一管理，构建战争数据知识图谱辅助决策，提供智能搜索、知识推荐、方案推荐等功能。

知识图谱系统为智能体开发训练平台提供军事规则、经验知识、实验结果、情报资料等约束性输入，去除智能体训练中的无效空间，加速智能体训练进程；智能体训练过程中产生的数据和新发现的知识链，效果链等可以补充进入到知识图谱系统中，实现知识图谱自生长。

二是与体系实验设计与分析工具配套应用，加速解决不确定场景下的军事人工智能训练初始场景问题。

目前由于受制于智能体泛化性、迁移性等问题，军事人工智能大部分还是针对具体想定的，在兵力规模、作战部署、作战任务、区域航线、装备性能参数等变化情况下，如果没有开展针对性训练，智能体的适应性一般表现较差。

但作战问题是一个对抗博弈问题，需要在更大的结构不确定性空间来考虑优化问题。目前我们真在探索一种采用体系实验设计与分析工具进行大不确定性空间筛选，用军事规则+强化学习人工智能进行小不确定性空间优化的技术路线。

算力需求：7 个实验因素，每个实验因素 5 个水平值，每局按 20 分

钟完成推演计算，考虑到随机性，一个实验方案重复 50 次实验。

5x5x5x5x5x5x5x20x50= 78125000 分钟，约 148 年（1 台机器）。

三是结合应用需求，继续深化分层多智能体的开发与训练技术实现，探索智能体针对不同想定案例的迁移性。

继续深化开展针对任务层和针对型号装备层（作战编组）的分层多智能体开发训练技术实现，优先探索针对型号装备（作战编组）下层智能体的迁移性研究，构建针对预警机、电子战飞机、加油机、制空编队（2 机、4 机）、突击编队（4 机、6 机、8 机）、单舰、舰艇编队（2 舰、3 舰、4 舰）等、单潜艇等底层智能体，在此基础上探索任务层智能体的迁移问题。

四是结合应用需求，继续深化分层多智能体的开发与训练技术实现，探索智能体针对不同想定案例的迁移性。

任务层智能体在自身兵力结构或者任务性质发生变化，或者处于不同的战场环境时，智能体可能会无所适从，如何让智能体适应这些变化？并取得比较好的效果？这是当前亟待解决的问题。元强化学习(meta-RL)似乎为这个问题提供了一种解决方案，它通过内-外环的学习方式，让智能体学会自己去学习！

五是结合应用需求，开展元博弈或者零和博弈研究。

在我们训练智能体的过程中，如果智能体的对手的策略比较单一，那么训练出来的智能体碰到其他从未见过的策略时，表现会很糟糕。如何为智能体提供丰富的对手策略，如何训练出不同风格的智能体？元博弈(meta-game)或者零和博弈（zero-sum game）为我们提供了一种非常好的思路，DeepMind 的 Alpha 系列算法的成功证实了这种思路的有效性。

智能体开发 Python 接口函数精讲

一、墨子AI平台介绍

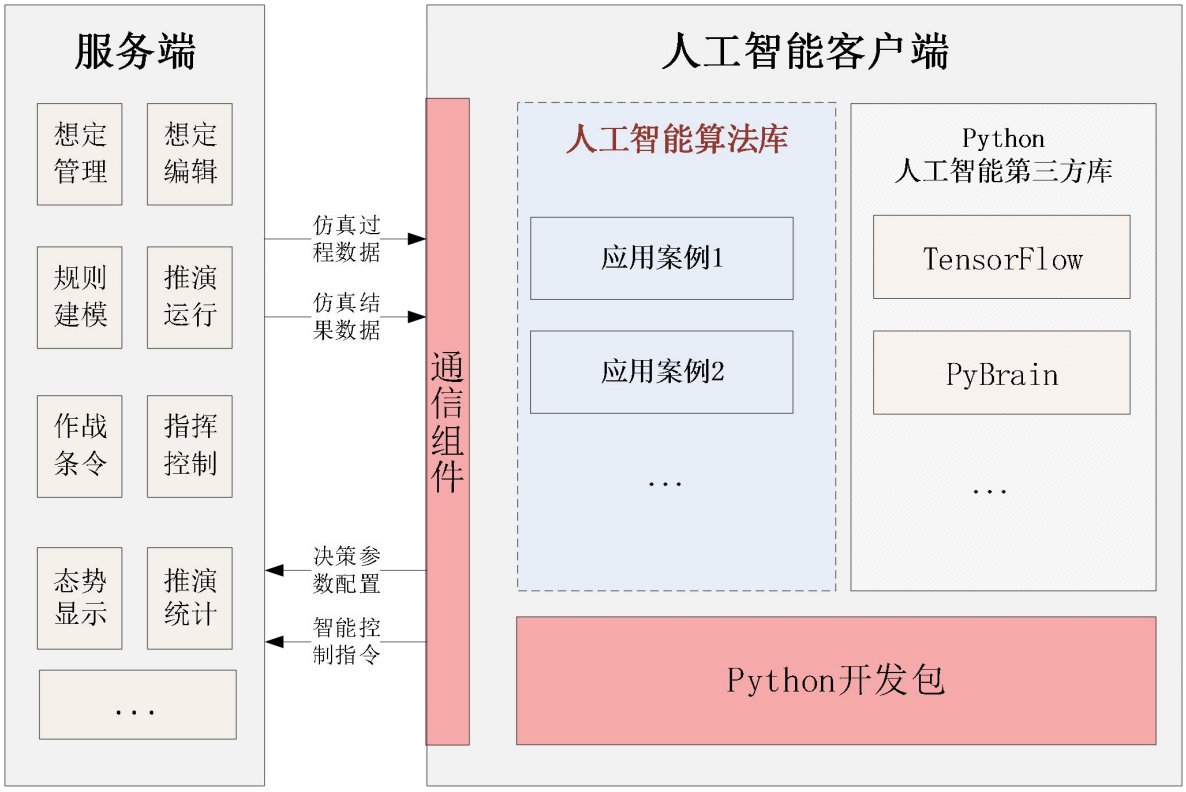


图 1 墨子 AI 平台

（一）总体架构

推演服务端是人工智能研究对象的模拟载体，负责为人工智能研究提供虚拟兵力、作战数据、环境数据等基础资源。

人工智能客户端包括通信组件、Python 开发包和人工智能算法库三部分，人工智能客户端通过通信组件接收推演服务端产生的仿真数据，并向服务端输出行动指令和其它交互参数，形成“控制-反馈-调整”的学习环境，在人工智能算法的调度下完成人工智能的学习训练。

（二）通信组件

通信接口主要用于实现 AI 智能体与推演服务器之间的通信交互，支持一对一、一对多等通信方式。Python 客户端和推演服务端可以在单机上部署，也可以在局域网上分开部署。

图 2 通信组件

（三）Python开发包

Python 开发包提供 Python 脚本编写的推演功能模块 API 函数交互接口，实现与推演模块之间的数据和指令交互。

接口函数类型包括获取作战效果和战场态势信息，各个实体类的属性以及动作接口，墨子推演系统的环境类，服务端类等。

利用 Python 开发包，结合 TensorFlow、PyTorch 等第三方人工智能框架和其它 Python 应用库，针对具体问题的决策参数和评价指标，可以开发出具有学习能力的人工智能算例。

二、代码框架

在墨子 AI 研究平台上，目前已经开展了两大类军事人工智能研究：

1、基于规则的人工智能案例；

2.、基于强化学习的人工智能案例。

（一）Mozi AI SDK

包含基础环境类、行为树推理模型、墨子 AI 接口单元测试代码以及部分案例代码。

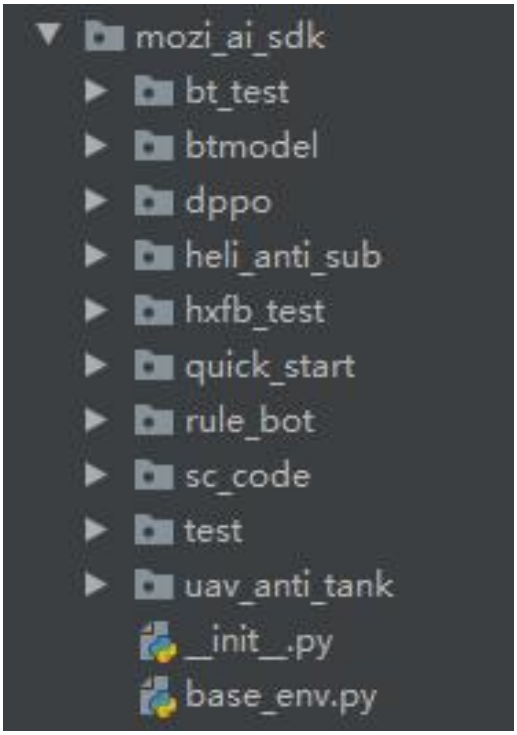


图 3 Mozi AI SDK

其中：

base_env.py 为基础环境类；

bt_model 文件夹为行为树推理模型；

test 文件夹为 Mozi AI 接口单元测试代码；

bt_test 为海峡风暴行为树案例；

dppo 文件夹为海峡风暴 DPPO 算法案例；

heli_anti_sub 文件夹为直升机反潜案例；

hxfb_test 文件夹为海峡风暴 DDPPO 算法案例；

rule_bot 文件夹为规则案例；

sc_code 文件夹为菲海战事分层模型案例；

uav_anti_tank 文件夹为无人机突防案例；

（二）Mozi Simu SDK

与墨子推演系统交互的接口。包含启动和连接墨子服务端，通过 gRPC 与墨子服务端通信，基本可实现墨子客户端上的所有用户操作，包括想定管理、想定编辑、条令规则、作战规划、推演控制、兵力操控、

目标标识等。目前提供的 AI 接口共 497 个。

（三）Mozi Utils

通用的功能模块，包括日志，作图以及少量地理计算模块。

三、接口说明与使用

本章通过若干个小案例，系统的讲述 Mozi AI 接口的使用。所有案例均在 Windows 下运行和演示。

（一）快速开始

通过一个小案例，介绍墨子 AI 智能体代码的一般结构和运行的基本流程。包括启动墨子服务端，连接墨子服务端，加载想定，设置服务端运行参数，态势信息构建等。

重点介绍仿真服务类（MoziServer）。

1， 想定文件

demo01.scen

2，案例代码结构

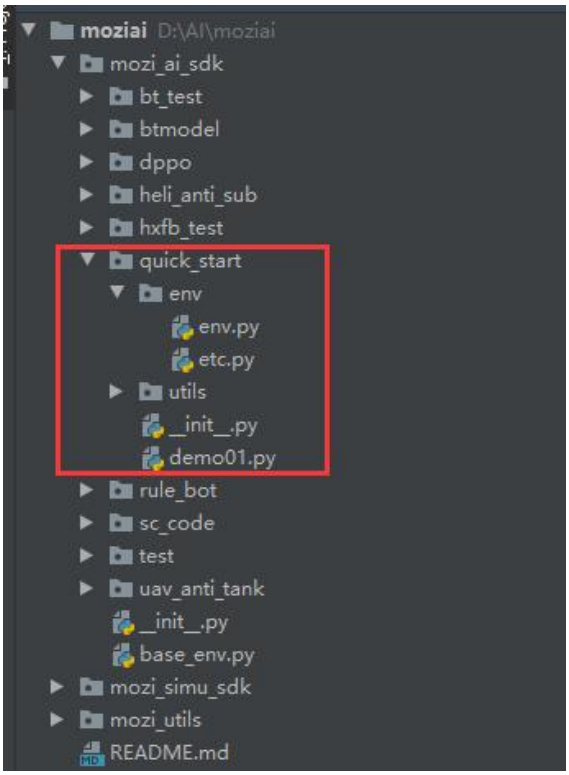


图 4 代码结构

本案例中包含环境类（env.py）、 配置文件（etc.py）、自定义代码（utils） 、入口程序（demo01.py）。

（1）env.py

继承基础环境类 BaseEnvironment。在连接墨子服务端时创建 MoziServer 类对象，通过调用该对象中的方法，完成加载想定、设置推演参数、初始化全局态势、开始推演、更新态势等操作。

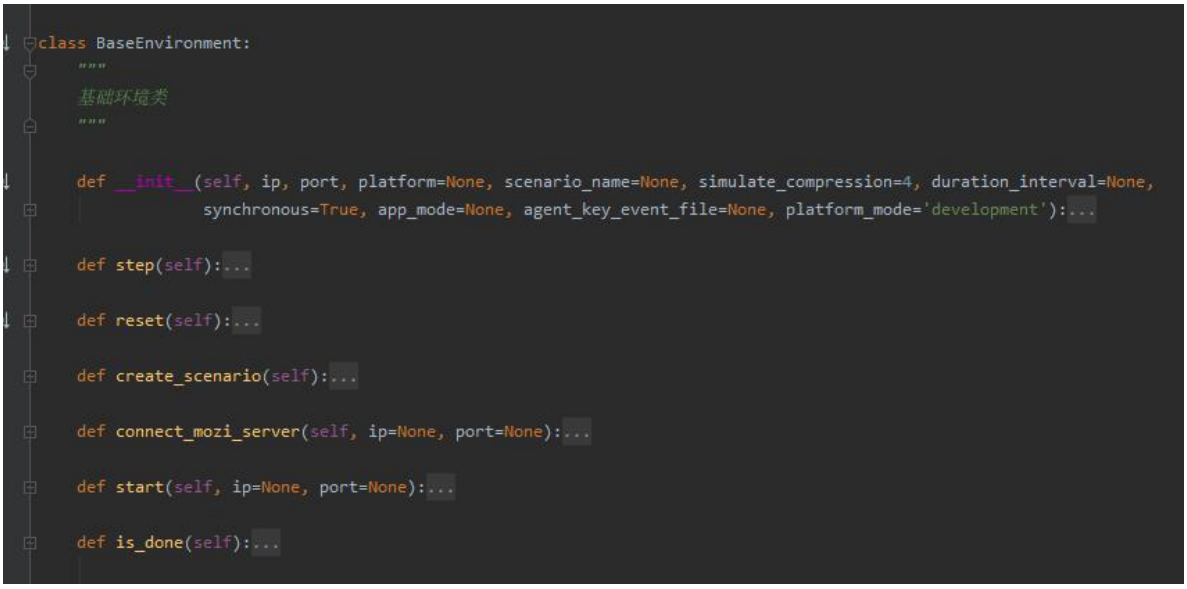


图 5 基础环境类

（2）etc.py

墨子启动的相关配置文件。主要配置墨子服务端 IP、端口，墨子运行的操作系统、加载的想定名称、推进速度、决策步长、智能体运行模式（app mode）等。

1) SERVER_IP：墨子服务端 IP

本地连接墨子服务端时，默认为 127.0.0.1，远程连接时，为墨子服务端所在服务器的 IP。

2) SERVER_PORT：墨子服务端端口

Windows 本地连接墨子时，默认为 6060。Linux 环境或比赛时，端口为启动墨子时设置的值。

3) PLATFORM：平台（Windows or Linux）

4) SCENARIO_NAME：想定名称

Windows 环境时，想定格式为.scen，想定名称为带后缀的想定文件名。

Linux 环境时，想定格式为.xml，想定名称为不带后缀的想定文件名。

5) SIMULATE_COMPRESSION：推进速度

推演档位 0：1 倍速，1：2 倍速，2：5 倍速，3：15 倍速，4：30

倍速，5：60 倍速，6：300 倍速，7：900 倍速，8：1800 倍速

6) DURATION_INTERVAL：决策步长，每隔多长时间获取一次态势
7) SYNCHRONOUS：同步/异步
8) MOZI_PATH：墨子安装目录（Windows）
9) APP_MODE：智能体运行模式，1 为本地 windows 模式，2 为 Linux 模式，3 为比赛模式。

（3）01_quick_start.py

```
# 设置墨子安装目录下bin目录为MOZIPATH，程序会跟进路径自动启动墨子
os.environ['MOZIPATH'] = etc.MOZI_PATH

# 创建环境类对象
env = Environment(etc.SERVER_IP, etc.SERVER_PORT, etc.PLATFORM, etc.SCENARIO_NAME,
                  etc.SIMULATE_COMPRESSION, etc.DURATION_INTERVAL, etc.SYNCHRONOUS, etc.APP_MODE)

# 启动墨子服务端
# 通过gRPC连接墨子服务端，产生env.mozi_server对象
# 设置推进模式 SYNCHRONOUS
# 设置决策步长 DURATION_INTERVAL
env.start()

# 加载想定，产生env.scenario
# 设置推进速度 SIMULATE_COMPRESSION
# 初始化全局态势
env.scenario = env.reset()

red_side = env.scenario.get_side_by_name('红方')
```

图 6 入口代码 1

```
flag = False
while not env.is_done():
    if env.step_count == 0 and not flag:
        flag = True
        # 获取本方地空导弹中队对象
        facilities = red_side.get_facilities()
        ground_to_air_missile_squadron = common.get_obj_by_name(facilities, '地空导弹中队')

        # 获取敌方单元苏-34 型“鸭嘴兽”攻击机的探测目标对象
        contacts = red_side.get_contacts()
        enemy_airplane = common.get_obj_by_name(contacts, '苏-34 型“鸭嘴兽”攻击机')
        enemy_airplane_guid = enemy_airplane.strGuid

        # 获取武器“MIM-104B型“爱国者-1”防空导弹”的数据库GUID
        weapon_db_guids = ground_to_air_missile_squadron.get_weapon_db_guids()
        weapon_db_guid = [j for j in weapon_db_guids if '1152' in j][0]

        # 手动攻击
        ground_to_air_missile_squadron.manual_attack(enemy_airplane.strGuid, weapon_db_guid, 1)
    else:
        # 下一步，更新态势
        env.step()
```

图 7 入口代码 2

以上代码中，reset 函数包含初始化全局态势功能，step 函数包含更新态势功能。前者用于构建 env.scenario 对象，后者用于更新 env.scenario 对象。

（二）态势解析

通过一个小案例，介绍态势的层级结构。

态势类 CSituation 在文件 situation.py 中。主要功能包含获取和解析全局态势、获取和解析更新态势。通过解析墨子服务端传递的态势字符串，将相关态势信息写入到 env.scenario 中。

态势类的接口共 56 个，用于在 env.reset 中获取和解析全局态势，在 env.step 中获取和解析更新态势，用户无需直接调用。

1，想定文件

active_unit_test.scen

2, 代码说明

（1）想定对象

通过想定对象，可获取当前想定的天气类对象、响应类对象、事件对象以及推演方类对象。

```
# 想定
weather = env.scenario.get_weather()
responses = env.scenario.get_responses()
events = env.scenario.get_events()
weapon_impacts = env.scenario.get_weapon_impacts()  # 暂不可用
sides = env.scenario.get_sides()
red_side = env.scenario.get_side_by_name('红方')
```

图 8 想定对象附属对象

（2）推演方对象

通过推演方对象，可获取本推演方的条令、活动单元、探测目标、航线、任务、参考点、禁航区、封锁区。

```
# 条令
mission_doctrine = red_side.get_doctrine()

# 活动单元
groups = red_side.get_groups()          # 编组
ships = red_side.get_ships()            # 水面舰艇
facilities = red_side.get_facilities()  # 地面兵力设施
aircrafts = red_side.get_aircrafts()    # 飞机
weapons = red_side.get_weapons()        # 武器
unguided_weapons = red_side.get_unguided_weapons()  # 非制导武器，暂不可用
aircraft_1 = red_side.get_unit_by_guid('e85dc457-c49f-4ccd-84ee-36ec967fb0d4')

# 探测目标
contacts = red_side.get_contacts()
contact = red_side.get_contact_by_guid('801ea534-a57c-4d3b-ba5d-0f77e909506c')
identified_targets_by_name = red_side.get_identified_targets_by_name('苏-34 型"鸭嘴兽"攻击机')

# 航线
side_ways = red_side.get_sideways()

patrol_missions = red_side.get_patrol_missions()    # 巡逻任务
strike_missions = red_side.get_strike_missions()    # 打击任务
support_missions = red_side.get_support_missions()  # 支援任务
cargo_missions = red_side.get_cargo_missions()      # 投送任务
ferry_missions = red_side.get_ferry_missions()      # 转场任务
mining_missions = red_side.get_mining_missions()                # 布雷任务 暂不可用
mine_clearing_missions = red_side.get_mine_clearing_missions()  # 扫雷任务 暂不可用
mission_1 = red_side.get_missions_by_name('空中打击')

# 参考点
reference_points = red_side.get_reference_points()
rp_16 = red_side.get_reference_point_by_name('RP-16')

# 禁航区
no_nav_zones = red_side.get_no_nav_zones()
# 封锁区
exclusion_zones = red_side.get_exclusion_zones()
```

图 9 推演方附属对象

（3）活动单元

通过活动单元，可获取该活动单元的条令、挂架、挂载、弹药库、传感器对象

```
# 活动单元

# 获取本方地空导弹中队对象
facilities = red_side.get_facilities()
ground_to_air_missile_squadron = common.get_obj_by_name(facilities, '地空导弹中队')

unit_doctrine = ground_to_air_missile_squadron.get_doctrine()
mounts = ground_to_air_missile_squadron.get_mounts()
magazines = ground_to_air_missile_squadron.get_magazines()
sensors = ground_to_air_missile_squadron.get_sensor()

# 获取本方地空导弹中队对象
blue_aircrafts = blue_side.get_aircrafts()
blue_aircraft = common.get_obj_by_name(blue_aircrafts, '敌机1')
loadout = blue_aircraft.get_loadout()
```

图 10 活动单元附属对象

（三）任务类-任务规划

1，巡逻任务

想定文件：demo02.scen

通过使用 AI 接口创建和设置空战巡逻任务，了解巡逻任务相关接口操作。包括添加参考点、创建巡逻任务、设置巡逻区域、警戒区域、航线管理等。

涉及的接口主要在推演方类（CSide）、参考点类、航线类、任务类（CMission）和巡逻任务类（CPatrolMission）。

（1）添加参考点

获取红方地面单元机场 1 的经纬度，根据该经纬度做一定的偏移，在机场右下方生成 4 个参考点，准备用做巡逻区域。

```
# 获取本方地空导弹中队对象
facilities = red_side.get_facilities()
airport = common.get_obj_by_name(facilities, '机场1')

# 根据机场经纬度添加参考点
point_1 = red_side.add_reference_point('参考点1', airport.dLatitude - 0.8, airport.dLongitude + 0.4)
point_2 = red_side.add_reference_point('参考点2', airport.dLatitude - 0.8, airport.dLongitude + 0.8)
point_3 = red_side.add_reference_point('参考点3', airport.dLatitude - 0.4, airport.dLongitude + 0.8)
point_4 = red_side.add_reference_point('参考点4', airport.dLatitude - 0.4, airport.dLongitude + 0.4)
```

图 11 添加参考点

（2）添加巡逻任务

根据参考点列表，创建空战巡逻任务。下图中 add_mission_patrol 第一个参数“空战巡逻”为任务名称，第二个 0 表示巡逻任务子类型，为空战巡逻，任务类型说明可参见该接口注释，point_list 为参考点名称组成的列表。add_mission_patrol 返回的是一个巡逻任务对象。

```
# 根据参考点创建巡逻任务
point_list = ['参考点1', '参考点2', '参考点3', '参考点4']
# 返回的任务对象非态势获取，仅部分属性有效
air_portal_mission = red_side.add_mission_patrol('空战巡逻', 0, point_list)
```

图 12 添加巡逻任务

（3）设置执行任务单元

将米格#1 和#2 设置为巡逻任务单元。

```
# 设置巡逻任务单元
aircrafts = red_side.get_aircrafts()
aircraft_1 = common.get_obj_by_name(aircrafts, '米格 #1')
aircraft_2 = common.get_obj_by_name(aircrafts, '米格 #2')
air_portal_mission.assign_unit(aircraft_1.strGuid)
air_portal_mission.assign_unit(aircraft_2.strGuid)
```

图 13 设置巡逻任务单元

（4）更新巡逻区

方法有两种，一是修改巡逻区列表，二是修改组成巡逻区的参考点经纬度。

```
# 设置更新巡逻区，变更参考点列表
point_list = ['参考点1', '参考点2', '参考点3']
air_portal_mission.set_patrol_zone(point_list)

# 修改参考点经纬度
red_side.set_reference_point('参考点1', airport.dLatitude - 0.7, airport.dLongitude + 0.3)
```

图 14 更新巡逻区

（5）设置警戒区

与设置巡逻区类似，先创建参考点，通过 set_prosecution_zone 方法设置。

```
# 设置警戒区
point_5 = red_side.add_reference_point('参考点5', airport.dLatitude - 1.0, airport.dLongitude + 0.2)
point_6 = red_side.add_reference_point('参考点6', airport.dLatitude - 1.0, airport.dLongitude + 1.0)
point_7 = red_side.add_reference_point('参考点7', airport.dLatitude - 0.2, airport.dLongitude + 1.0)
point_8 = red_side.add_reference_point('参考点8', airport.dLatitude - 0.2, airport.dLongitude + 0.2)
# 根据参考点创建巡逻任务
prosecution_point_list = ['参考点5', '参考点6', '参考点7', '参考点8']
air_portal_mission.set_prosecution_zone(prosecution_point_list)
```

图 15 设置警戒区

（6）设置油门高度

可通过接口设置飞机攻击距离、出航、阵位、攻击的油门高度。

```
# 出航高度
air_portal_mission.set_transit_altitude(1000)
# 阵位高度
air_portal_mission.set_station_altitude(2000)
# 攻击高度
air_portal_mission.set_attack_altitude(3000)

# 出航油门-巡航
air_portal_mission.set_throttle_transit('Full')
# 阵位油门-全速
air_portal_mission.set_throttle_station('Cruise')
# 攻击油门-最大
air_portal_mission.set_throttle_attack('Flank')

# 设置任务的攻击距离
air_portal_mission.set_attack_distance(50)
```

图 16 设置油门高度和攻击距离

（7）设置任务是否启用和时间

可设置任务是否启用，以及任务的启动时间和失效时间。这三个接口适用于所有任务。

```
# 设置任务不启用
air_portal_mission.set_is_active('false')

# 设置任务启动时间和失效时间
air_portal_mission.set_start_time('2021-09-18 23:10:00')
air_portal_mission.set_end_time('2021-09-19 2:10:00')
```

图 17 设置任务是否启用和时间

（8）编队规模设置

可设置任务的编队规模，以及是否飞机数低于编队规模就不能起飞。

```
# 设置打击任务编队规模
air_portal_mission.set_flight_size(3)
# 设置打击任务是否飞机数低于编组规模数要求就不能起飞
air_portal_mission.set_flight_size_check('true')
```

图 18 编队规模设置

（9）航线设置

通过接口可设置巡逻任务的出航航线、阵位航线以及返航航线。

```
# 设置航线
red_side.add_plan_way(0, '出航航线')
red_side.add_plan_way(0, '返航航线')
red_side.add_plan_way(0, '巡逻航线')
red_side.add_plan_way_point('出航航线', 128.465614876251, 26.671474736029)
red_side.add_plan_way_point('出航航线', 128.887065012163, 26.5833293108848)
red_side.add_plan_way_point('返航航线', 128.244808399628, 26.2561596899537)
red_side.add_plan_way_point('返航航线', 128.117884730546, 26.644787386411)
red_side.add_plan_way_point('巡逻航线', 128.714605112882, 26.0781801004785)
red_side.add_plan_way_point('巡逻航线', 128.965734517005, 26.2415779306963)
red_side.set_plan_way_showing_status('出航航线', 'true')
red_side.set_plan_way_showing_status('返航航线', 'true')
red_side.set_plan_way_showing_status('巡逻航线', 'true')
# 出航航线
air_portal_mission.add_plan_way_to_mission(0, '出航航线')
# 返航航线
air_portal_mission.add_plan_way_to_mission(2, '返航航线')
# 巡逻航线
air_portal_mission.add_plan_way_to_mission(3, '巡逻航线')
```

图 19 航线设置

2，打击任务

想定文件：demo03.scen

通过使用 AI 接口，创建和设置打击任务，了解打击任务相关操作。包括设置作战任务单元、设置护航单元、设置目标对象等。

涉及的接口任务类（CMission）、打击任务类（CStrikeMission）和探测目标类（CContact）。

（1）打击任务创建

```
# 创建对海突击任务
sea_strike_mission = red_side.add_mission_strike('对海打击', 2)
```

图 20 创建打击任务

（2）设置执行任务单元

```
aircrafts = red_side.get_aircrafts()
# 设置打击任务飞机
for i in range(6, 11, 1):
    aircraft = common.get_obj_by_name(aircrafts, f'F-16A #{i}')
    sea_strike_mission.assign_unit(aircraft.strGuid)
```

图 21 设置执行任务单元

（3）设置护航单元

```
# 设置护航飞机
for i in range(5):
    i += 1
    aircraft = common.get_obj_by_name(aircrafts, f'F-16A #{i}')
    sea_strike_mission.assign_unit(aircraft.strGuid, 'true')
```

图 22 设置护航单元

（4）设置打击目标

```
# 设置打击目标
contacts = red_side.get_contacts()
enemy_ship = common.get_obj_by_name(contacts, '舰船1')
enemy_ship_guid = enemy_ship.strGuid
sea_strike_mission.assign_unit_as_target(enemy_ship_guid)
# 设置仅考虑计划目标
sea_strike_mission.set_preplan('true')
```

图 23 设置打击目标

（5）设置打击任务触发条件

```
# 设置打击任务触发条件，探测目标至少为不明、非友、敌对
sea_strike_mission.set_minimum_trigger(4)
```

图 24 设置打击任务触发条件

（6）燃油弹药规则

```
# 设置如果不能打击目标，则带回空对地弹药
sea_strike_mission.set_fuel_ammo(2)
```

图 25 设置燃油弹药规则

（7）设置打击半径

```
# 最小打击半径-单位海里
sea_strike_mission.set_min_strike_radius(30)
# 最大打击半径-单位海里
sea_strike_mission.set_max_strike_radius(1000)
```

图 26 设置最小/大打击半径

（8）雷达运用

```
# 雷达运用：从进入攻击航线段到武器消耗完毕状态点打开雷达
sea_strike_mission.set_radar_usage(3)
```

图 27 设置雷达运用

（9）航线设置

```
# 出航航线
sea_strike_mission.add_plan_way_to_mission(0, '单元航线出航')
# 返航航线
sea_strike_mission.add_plan_way_to_mission(2, '单元航线返航')
# 武器航线-需要巡航导弹，且路径点数不为0
sea_strike_mission.add_plan_way_to_mission(1, '武器航线')
```

图 28 航线设置

（10）是否仅一次

```
# 设置打击任务是否仅限一次
sea_strike_mission.set_strike_one_time_only('true')
```

图 29 是否仅一次

（11）护航设置

```
# 设置打击任务护航最大威胁响应半径
sea_strike_mission.set_strike_escort_response_radius(100)
# 设置护航飞机的编队规模
sea_strike_mission.set_strike_escort_flight_size_shooter(3)
```

图 30 护航设置

3，支援任务

想定文件：demo04.scen

通过使用 AI 接口，创建和设置支援相关操作。

涉及的接口主要在推演方类（CSide）、任务类（CMission）和支援任务类（CSupportMission）。

（1）支援任务创建

```
# 添加参考点
point_1 = red_side.add_reference_point('RP-1', 37.482920555205, 123.362532042046)
point_2 = red_side.add_reference_point('RP-2', 37.1244585650515, 123.211235946129)

# 根据参考点创建支援任务
point_list = ['RP-1', 'RP-2']
mission_support = red_side.add_mission_support('支援任务', point_list)

# 设置打击任务飞机
aircrafts = red_side.get_aircrafts()
for i in range(1, 4, 1):
    aircraft = common.get_obj_by_name(aircrafts, f'侦察机 #{i}')
    mission_support.assign_unit(aircraft.strGuid)
```

图 31 支援任务创建

（2）支援任务设置

```
# 任务是否激活
mission_support.set_is_active('true')

# 设置任务启用时间和失效时间
mission_support.set_start_time('2021-09-26 15:55:00')
mission_support.set_end_time('2021-09-26 19:55:00')

# 阵位上每类平台保持几个作战单元
mission_support.set_maintain_unit_number(2)

# 设置遵循1/3规则
mission_support.set_one_third_rule('true')

# 设置仅一次
mission_support.set_one_time_only('true')

# 设置任务是否在阵位上打开电磁辐射
mission_support.set_emcon_usage('true')

# 设置导航类型-仅一次 -- 暂不可用
mission_support.set_loop_type('true')
```

图 32 支援任务设置一

```
# 编队规模
mission_support.set_flight_size(3)
mission_support.set_flight_size_check('true')

# 空中加油-设置不允许空中加油
mission_support.set_use_refuel_unrep(1)

# 设置出航返航航线
red_side.add_plan_way(0, '单元航线-出航')
red_side.add_plan_way(0, '单元航线-返航')
# 出航航线
mission_support.add_plan_way_to_mission(0, '单元航线-出航')
# 返航航线
mission_support.add_plan_way_to_mission(2, '单元航线-返航')

# 出航油门-军用
mission_support.set_throttle_transit('Full')
# 阵位油门-巡航
mission_support.set_throttle_station('Cruise')
# 出航高度
mission_support.set_transit_altitude(1000)
# 阵位高度
mission_support.set_station_altitude(2000)
```

图 33 支援任务设置二

（四）活动单元类-兵力操控

本章通过一个简单的案例，介绍兵力操控相关接口，包括单机出动、编组出动、设置油门高度、设置航线、返回基地、传感器开关机等操作。

涉及的主要在推演方类（CSide）、和活动单元类（CActiveUnit）。

1，想定文件

demo05.scen

2，代码说明

（1）飞机单机出动、编组出动、终止出动

```
# 设置F35 #1单机出动
aircraft_6.set_single_out()
# 编组出动
unit_list = [aircraft_7.strGuid, aircraft_8.strGuid]
# 终止出动
aircraft_7.abort_launch()
aircraft_8.abort_launch()
```

图 34 空港指令

（2）航线规划

```
# 航线规划
course_list = [(37.0398991935271, 122.958786299026), (37.3926884324471, 123.618312576662),
               (37.6447756693682, 124.107738263721)]
aircraft_6.plot_course(course_list)

# 删除第2个航路点(从0开始计数)
aircraft_6.delete_coursed_point([2])
```

图 35 航线规划

（3）电磁管控

```
# 设置单元雷达、声纳、干扰机开机
aircraft_6.switch_sensor(radar='true', sonar='true', oecm='true')

# 设置单元是否遵循电磁管控
aircraft_6.unit_obeys_emcon('false')
# 设置雷达开机
aircraft_6.set_radar_shutdown('false')
# 设置声纳开机
aircraft_6.set_sonar_shutdown('false')
# 设置干扰机开机
aircraft_6.set_oecm_shutdown('false')

# 航路点设置雷达、声纳、主动ECM开机
aircraft_6.set_way_point_sensor(0, 'CB_radar', 'Checked')
aircraft_6.set_way_point_sensor(0, 'CB_Sonar', 'Checked')
aircraft_6.set_way_point_sensor(0, 'CB_ECM', 'Checked')
```

图 36 电磁管控

（4）速度高度设置

```
# 设置期望速度
aircraft_6.set_desired_speed(900)
# # 设置单元期望高度
aircraft_6.set_desired_height(8000, 'true')
```

图 37 速度高度设置

（5）手动攻击

```
# 手动攻击
weapon_db_guid = 'hsfw-dataweapon-00000000000816'
contacts = red_side.get_contacts()
enemy_ship = common.get_obj_by_name(contacts, '舰船1')
enemy_ship_guid = enemy_ship.strGuid
aircraft_6.allocate_weapon_to_target(enemy_ship_guid, weapon_db_guid, 1)
```

图 38 手动攻击

（6）选择新基地和返航

```
# 选择新基地
facilities = red_side.get_facilities()
airport = common.get_obj_by_name(facilities, '机场2')
aircraft_6.select_new_base(airport.strGuid)
# 返回基地
aircraft_6.return_to_base()
```

图 39 选择新基地和返航

（五）条令类-条令规则

本章通过一个简单的案例，介绍条令规则相关操作。从不同维度设置条令规则，包括推演方条令设置、任务条令设置、编组条令设置、单元条令设置。具体操作包含设置总体条令、电磁管控、武器使用规则等。

涉及的接口主要在条令类（CDoctrine）、推演方类（CSide）、任务类（CMission）和活动单元类（CActiveUnit）。

1，想定文件

demo06.scen

2，代码说明

（1）不同维度设置条令

```
# 推演方条令
side_doctrine = red_side.get_doctrine()

# 任务条令
mission = red_side.get_missions_by_name('巡逻任务')
mission_doctrine = mission.get_doctrine()

# 编组条令
groups = red_side.get_groups()
group = common.get_obj_by_name(groups, '飞行编队 4')
group_doctrine = group.get_doctrine()

# 单元条令
aircrafts = red_side.get_aircrafts()
aircraft = common.get_obj_by_name(aircrafts, '战斗机1')
aircraft_doctrine = aircraft.get_doctrine()

# 当前不支持航路点条令
```

图 40 不同维度设置条令

（2）武器控制状态

```
# 武器控制状态
# 对海-自由开火
side_doctrine.set_weapon_control_status_surface(0)
# 对潜-谨慎开火
side_doctrine.set_weapon_control_status_subsurface(1)
# 对地-限制开火
side_doctrine.set_weapon_control_status_land(2)
# 对空-自由开火
side_doctrine.set_weapon_control_status_air(0)

# 进攻时忽略计划航线 - 对巡逻任务不起作用
side_doctrine.ignore_plotted_course('yes')

# 设置与模糊位置目标的交战状态 - 忽略模糊性
side_doctrine.set_ambiguous_targets_engaging_status(0)

# 决定与临机目标的交战状态 - 可与任何目标交战
side_doctrine.set_opportunity_targets_engaging_status('true')
```

图 41 交战规则设置

（3）其他总体条令设置

```
# 设置受到攻击时是否忽略电磁管控 - 忽略
side_doctrine.ignore_emcon_while_under_attack('true')

# 决定是否自动规避 - 是
side_doctrine.evade_automatically('true')

# 是否运行加油补给
side_doctrine.use_refuel_supply(2)
# 设置加油补给的选择对象 - 选择最近的加油机
side_doctrine.select_refuel_supply_object(0)
# 设置是否给盟军单元加油补给 - 否
side_doctrine.refuel_supply_allies(3)

# 燃油状态-返航：编队中所有飞机均因达到单机油料状态要返航时，编队才返航
side_doctrine.set_fuel_state_for_air_group('YesLastUnit')

# 武器状态-返航：编队中所有飞机均因达到单机武器状态要返航时，编队才返航
side_doctrine.set_weapon_state_for_air_group('YesLastUnit')

# 反舰作战行动-设置是否与目标保持距离
side_doctrine.maintain_standoff('true')
```

图 42 其他总体条令设置

（3）电磁管控设置

```
# 电磁管控设置 - 设置电磁管控状态
# 雷达打开
side_doctrine.set_em_control_status('Radar', 'Active')
# 声呐打开
side_doctrine.set_em_control_status('Sonar', 'Active')
# 干扰机打开
side_doctrine.set_em_control_status('OECM', 'Active')
```

图 43 电磁管控设置

（4）武器使用规则设置

```
side_doctrine.set_weapon_release_authority(weapon_dbid='3152', target_type='2001', quantity_salvo=2,
                                           shooter_salvo=4, firing_range=120, self_defense=11,
                                           escort='')
```

图 44 武器使用规则设置

四、行为树案例介绍

（一）基本概念

行为树的基本原理：自顶向下的，通过一些条件来搜索行为树，最终确定需要做的行为（叶节点），并且执行它。

（1）行为节点和控制节点

行为节点（Action Node）定义真正的行为树的行为。

控制节点（Control Node）其实就是“控制”其子节点如何被执行。

本案例中的控制节点有两种：选择、序列。

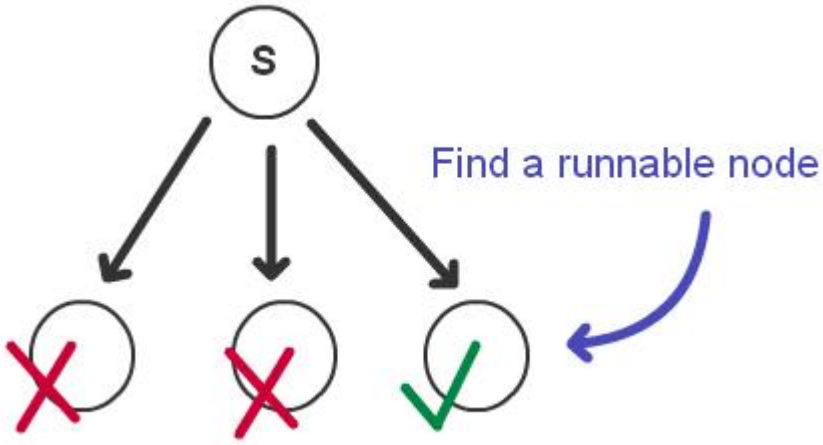


图 45 选择

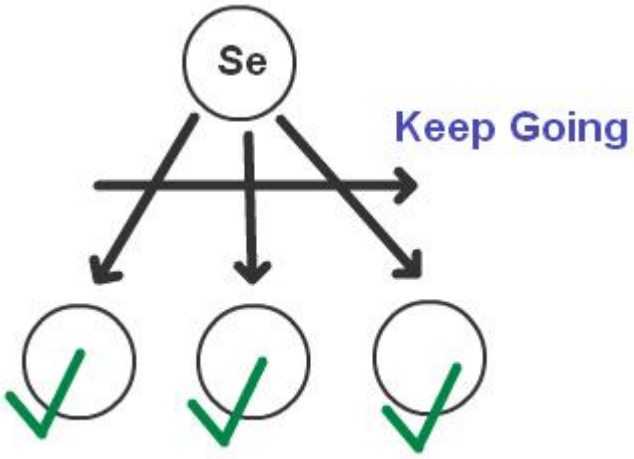


图 46 序列

（二）行为树模型实现

行为树模型在 mozi_ai_sdk/btmodel/bt/bt_nodes.py 中的 BT 类中实现。该类为行为树节点类。本案例中使用了该模型的以下 5 中方法：

1，__init__：构造函数，用于创建行为树对象。

2，set_action：设置节点对象的行为。

3，add_child：给节点对象添加子节点。

4，sequence：节点对象行为，序列。

依次执行子节点行为，有一个子节点返回 False，下一个子节点不执行，返回 False；

所有的子节点都返回 True，则返回 True。

5，select：节点对象行为，选择。

依次执行子节点行为，有一个子节点返回 True，下一个子节点不执行，返回 True；

所有的子节点都返回 False，本身返回 False。

（三）想定说明

1，推演主题

《海峡风暴》航母遭遇战

2，想定背景

红蓝双方航母战斗群在太平洋某海域遭遇，擦枪走火，发生冲突。

3，想定时间

起止时间：2021 年 4 月 16 日 22 时 0 分 0 秒 - 23 时 40 分 0 秒

持续时间：100 分钟

4，双方兵力

红方：福特级航空母舰 X1、阿里伯克级导弹驱逐舰 X1、F-35C 舰载战斗机 X16；

蓝方：福特级航空母舰 X1、阿里伯克级导弹驱逐舰 X1、F-35C 舰载战斗机 X16。

5，想定文件

Windows：海峡风暴-资格选拔赛-蓝方任务随机方案-给周国进测试.scen

Linux：hxfb-multitask.xml

（四）行为树设计

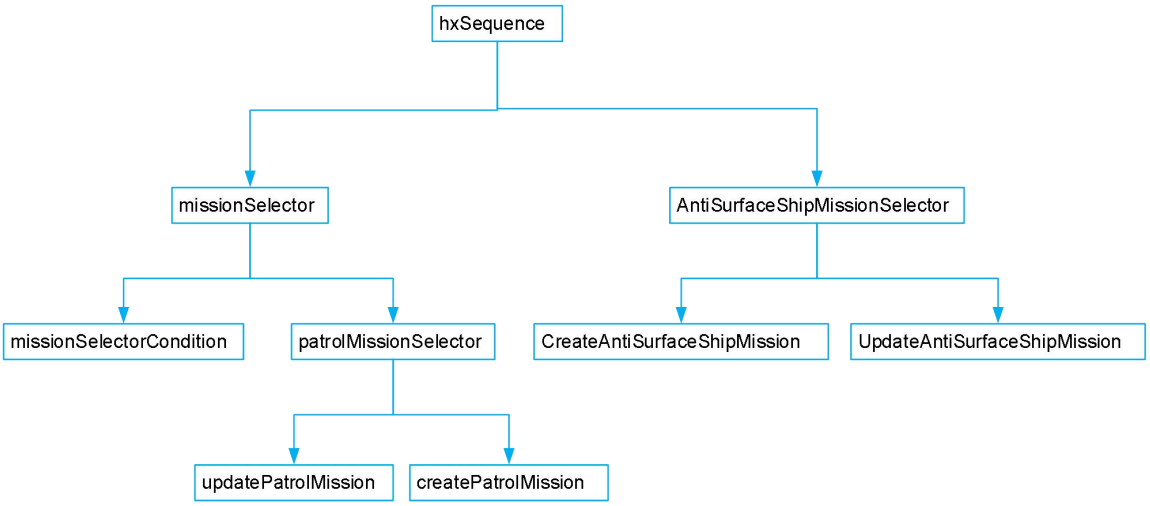


图 1- 47 行为树案例行为树设计

（五）代码结构与实现

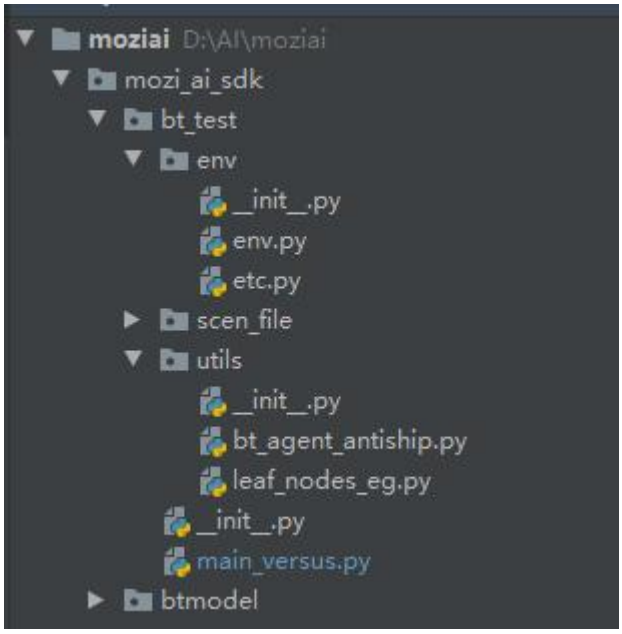


图 1- 48 行为树案例代码结构

env.py：环境类

etc.py：配置

bt_agent_antiship.py：行为树结构构建

leaf_nodes_eg.py：叶子节点定义

main_versus.py：入口程序

btmodel：行为树模型

1，bt_agent_antiship.py

本文件中定义行为树节点对象并指定根节点，通过节点对象方法 add_child 连接节点形成树。通过节点对象方法 set_action 定义节点执行的动作。

（1）定义行为树的节点

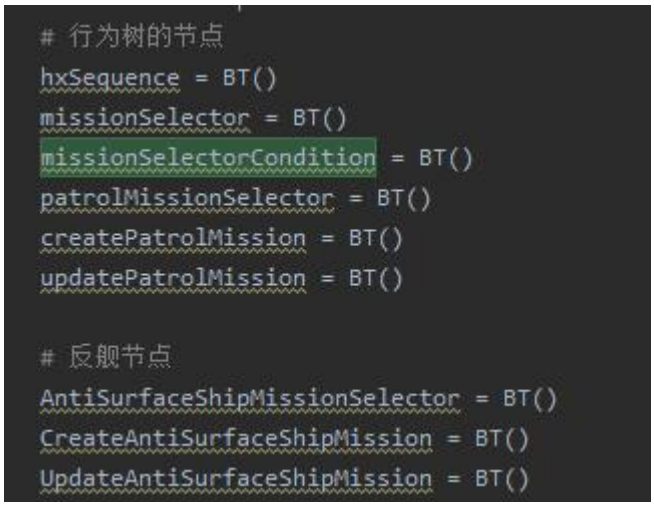


图 1- 49 行为树节点定义

（2）定义根节点

图 1- 50 行为树根节点定义

（3）连接节点形成树

```
# 连接节点形成树
hxSequence.add_child(missionSelector)
hxSequence.add_child(AntiSurfaceShipMissionSelector)
missionSelector.add_child(missionSelectorCondition)
missionSelector.add_child(patrolMissionSelector)

patrolMissionSelector.add_child(updatePatrolMission)
patrolMissionSelector.add_child(createPatrolMission)

AntiSurfaceShipMissionSelector.add_child(CreateAntiSurfaceShipMission)
AntiSurfaceShipMissionSelector.add_child(UpdateAntiSurfaceShipMission)
```

图 1- 51 连接节点形成树

（4）定义节点执行的动作

```
# 每个节点执行的动作
hxSequence.set_action(hxSequence.sequence, sideGuid, shortSideKey, attributes)
missionSelector.set_action(missionSelector.select, sideGuid, shortSideKey, attributes)
missionSelectorCondition.set_action(antiship_condition_check, sideGuid, shortSideKey, attributes)

patrolMissionSelector.set_action(patrolMissionSelector.select, sideGuid, shortSideKey, attributes)

updatePatrolMission.set_action(update_patrol_mission, sideGuid, shortSideKey, attributes)
createPatrolMission.set_action(create_patrol_mission, sideGuid, shortSideKey, attributes)

AntiSurfaceShipMissionSelector.set_action(AntiSurfaceShipMissionSelector.select, sideGuid, shortSideKey,
                                          attributes)
CreateAntiSurfaceShipMission.set_action(create_antisurfaceship_mission, sideGuid, shortSideKey, attributes)
UpdateAntiSurfaceShipMission.set_action(update_antisurfaceship_mission, sideGuid, shortSideKey, attributes)
```

图 1- 52 定义节点执行的动作

2，leaf_nodes_eg.py

本文件中定义叶子节点的行为。

（1）antiship_condition_check

用上帝视角判断，如果对方飞机剩余小于 6 架，就启动反舰任务。

（2）create_antisurfaceship_mission

创建反舰任务。

（3）update_antisurfaceship_mission

更新反舰任务。

（4）create_patrol_mission

创建巡逻任务。

（5）update_patrol_mission

更新巡逻任务,主要功能是两艘轮船往中间经纬度开动。

3，main_versus.py

对战入口文件。本文中定义了智能体的常规操作和行为树操作。行为树包括初始化行为树和更新行为树。

初始化行为树执行创建节点对象和构建行为树动作。

```
# 实例化智能体
agent = CAgent()
# 初始化行为树
agent.init_bt(env, side_name, 0, '')
```

图 1- 53 初始化行为树

更新行为树是从根节点开始执行节点对象的动作。

```
# 更新动作
agent.update_bt(side_name, env.scenario)
```

图 1- 54 更新行为树

4、对称设计

本案例既可以扮演红方运行，也可以扮演蓝方运行。

智能体开发平台操作使用

仿照 Python 接口函数精讲中讲的内容，依托专项赛想定菲海战事，扮演蓝方，完成一个小案例制作。

想定文件：菲海战事-仅红方有任务-v2.scen

想定说明：参见《赛事想定说明-2021-9-22.docx》

要求：通过实际操作熟悉任务创建、兵力操控、条令设置等接口操作。对接口已经比较熟悉的同学可以自由发挥。

强化学习算法精讲

一、强化学习基础

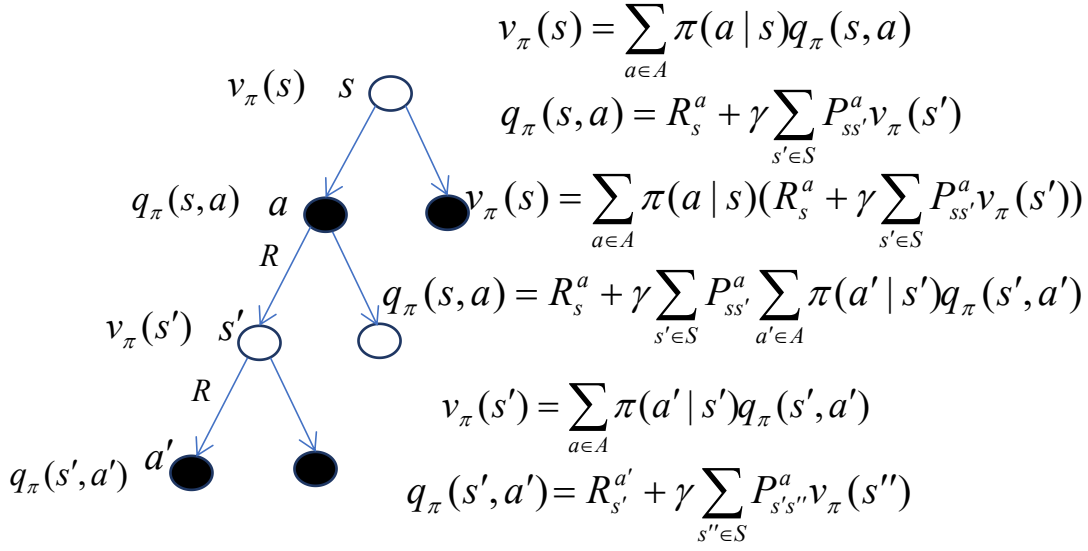


图 55 Value 和 Action-value 的关系

（一）累积奖励、值函数、优势函数

State-value 函数定义：

$$V^{\pi,\gamma}(s_t) := E_{\substack{s_{t+1:\infty} \\ a_{t:\infty}}}[\sum_{l=0}^{\infty} \gamma^l r_{t+l}]$$

State-action-value 函数定义：

$$Q^{\pi,\gamma}(s_t, a_t) := E_{\substack{s_{t+1:\infty} \\ a_{t+1:\infty}}}[\sum_{l=0}^{\infty} \gamma^l r_{t+l}]$$

优势函数定义：

$$A^{\pi,\gamma}(s_t, a_t) := Q^{\pi,\gamma}(s_t, a_t) - V^{\pi,\gamma}(s_t) = r_t + \gamma V^{\pi,\gamma}(s_{t+1}) - V^{\pi,\gamma}(s_t)$$

（二）策略迭代与策略评估

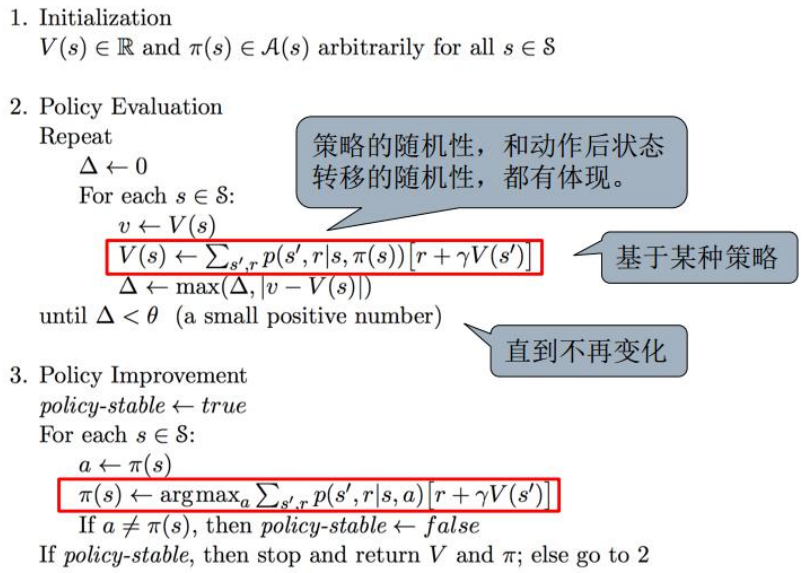


图 56 policy evaluation 和 policy improvement

策略迭代从一个初始化的策略出发，先进行策略评估，然后改进策略，评估改进的策略，再进一步改进策略，经过不断迭代更新，直达策略收敛。

（三）Reward shaping

强化学习在实际项目的应用中面临着稀疏奖励问题 (Sparse Reward Problem) 的挑战，在兵棋推演中也不例外。在兵棋推演中，如果没有人为设计的奖励信号，一般只能将一局游戏胜负或者战损作为奖励信号提供给学习算法，而仅仅靠这样的奖励信号很难学习到好的策略。因此，通常研究者都会根据自己的经验和对推演环境的认知，人为设计一些奖励来辅助学习算法。这种将先验知识转化为附加奖励函数，从而引导学习算法学得更快、更好的方式，就叫做奖励塑形 (Reward Shaping)。

首先引入一个典型的问题，智能体从 S_0 到 Goal 的问题，当智能体接近 Goal 就给予正奖励，其余奖励为 0，可能会导致智能体学到在 S_0 附近“兜圈”，智能体在某个地方兜圈就可以持续获得奖励，智能体来回持续震荡也可以持续获取奖励，如图 57 所示。

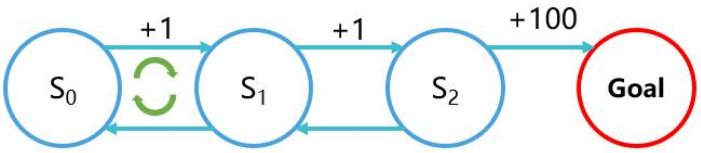


图 57 “兜圈”策略

基于势能的奖励塑造[1]可以解决这一问题，记原 MDP 问题为 M，新的 MDP 问题为M',可以通过 M 的奖励值加上附加奖励函数的值得到M'的奖励值。附加奖励函数（如图 58 所示）表示为势函数的差分形式是最优策略不变性的充分必要条件。

Definition

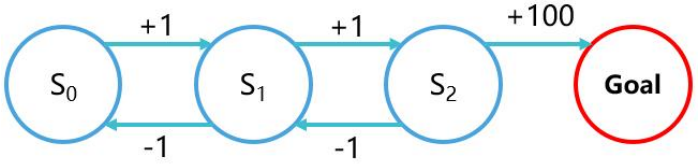
A shaping reward function $F : S \times A \times S \to \mathbb{R}$ is **potential-based** if there exists $\Phi : S \to \mathbb{R}$ s.t.

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s)$$

for all $s \neq s_0, a, s'$.

图 58 基于势能函数的差分附加奖励函数

在原奖励基础上增加附加奖励函数就相当于给每个状态一个势能，从势能低的地方到势能高的地方给正奖励，而从势能高的地方回势能低的地方给负奖励，这样就避免了“兜圈”的策略，如图 59 所示。这种方法可以保证新旧 MDP 的最优策略不变性。



$\Phi(s_0) = 0$　　$\Phi(s_1) = 1$　　$\Phi(s_2) = 2$

图 59 势能函数作用于“兜圈策略”

根据贝尔曼方程，最优值函数具有如下形式：

$$Q_M^*(s,a) = \mathrm{E}_{s' \sim P_{sa}(\cdot)}\left[R(s,a,s') + \gamma \max_{a' \in A} Q_M^*(s',a')\right]$$

两边减去势能函数，作简单变换有：

$$Q_M^*(s,a)-\Phi(s)=\mathrm{E}_{s'}\Big[R(s,a,s')+\gamma\Phi(s')-\Phi(s)$$
$$+\gamma\max_{a'\in A}(Q_M^*(s',a')-\Phi(s'))\Big]$$

定义：

$$\hat{Q}_{M'}(s,a)\ \triangleq\ Q_M^*(s,a)\ -\ \Phi(s)$$

然后根据：

$$F(s,a,s')\ =\ \gamma\Phi(s')\ -\ \Phi(s)$$

得到：

$$\begin{aligned}&\hat{Q}_{M'}(s,a)\\&=\mathrm{E}_{s'}\left[R(s,a,s')+F(s,a,s')+\gamma\max_{a'\in A}\hat{Q}_{M'}(s',a')\right]\\&=\mathrm{E}_{s'}\left[R'(s,a,s')+\gamma\max_{a'\in A}\hat{Q}_{M'}(s',a')\right]\end{aligned}$$

显然$\hat{Q}_{M'}(s,a)$满足M'的贝尔曼方程，因此$Q_{M'}^*(s,a)=\hat{Q}_{M'}(s,a)$。那么最优策略满足：

$$\begin{aligned}\pi_{M'}^*(s)\ &\in\ \arg\max_{a\in A}Q_{M'}^*(s,a)\\&=\ \arg\max_{a\in A}Q_M^*(s,a)-\Phi(s)\\&=\ \arg\max_{a\in A}Q_M^*(s,a)\end{aligned}$$

因此可得新旧 MDP 的最优策略是一致的。

基于势能函数的 reward shaping 方法还有 potential-based advice[2],dynamic potential-based reward shaping[3],二者的区别在于前者在势能函数中加入了动作，这样就变成了基于势能的建议；后者允许势能函数随时间变化，如图 60 所示。

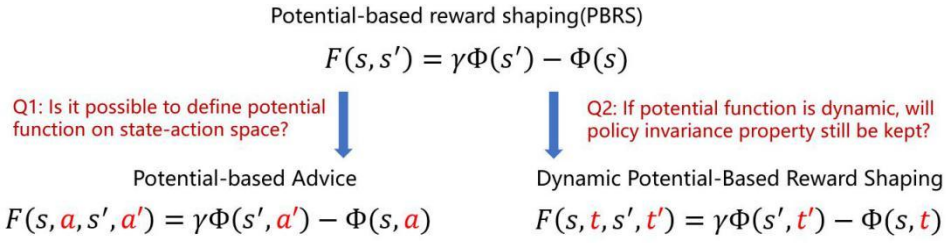


图 60

还有一种方法是二者的结合[4]，如图 61 所示，可以将任意奖励函数改造成基于势能的奖励函数，这种方法也有策略的一致性保障。

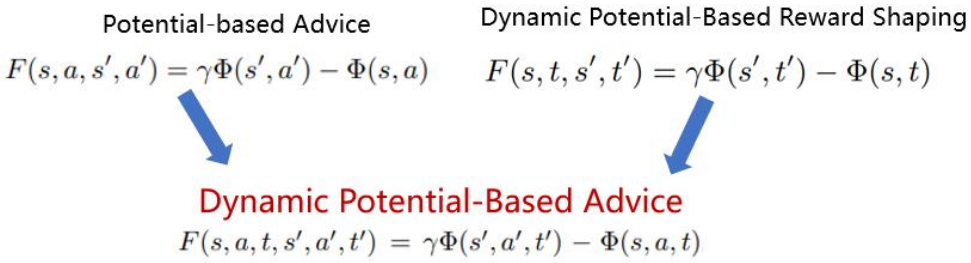


图 61

另外，逆强化学习可以从专家数据中学习奖励函数[5]。文章[6]表明基于势能的 reward shaping 等价于为值函数提供一个初始值，这为使用元强化学习方法[7]进行 reward shaping 提供了一个思路。除了基于势能的 reward shaping 方法，还有基于内在激励[8]的方法，比如由好奇心驱动探索学习[9]。

二、深入理解强化学习

（一）Value-based方法

1、DQN

DQN[10]与 Q-learning 类似都是基于值迭代的算法，但是在普通的 Q-learning 中，当状态和动作空间是离散且维数不高时可使用 Q-Table 储存每个状态动作对的 Q 值，而当状态和动作空间是高维连续时，动作空间和状态空间太大，使用 Q-Table 就十分困难。

所以在此处可以把 Q-table 更新转化为一函数拟合问题，通过拟合一个函数 function 来代替 Q-table 产生 Q 值，使得相近的状态得到相近的输出动作。因此我们可以想到深度神经网络对复杂特征的提取有很好效果，所以可以将 Deep-Learning 与 Reinforcement-Learning 结合。这就成为了 DQN。

我们通过一系列的 observations, actions and rewards，考虑 agent 和环境交互的任务，agent 的目标是：选择动作，最大化累积奖赏。我们可以利用一个深度网络来近似最优的动作-值函数(the optimal action-value function)。

RL 一个比较显著的问题是：当用一个非线性函数估计，如：一个神经网络来表示动作-值函数时，就会导致不稳定或者不收敛。这个不稳定主要有几个原因导致的：序列观察的相关性(the correlations present in the sequence of observations), Q 值的较小的更新，可能会显著的改变策略，从而改变数据分布，以及动作-值(Q)和目标值之间的关系。我们利用新颖的 Q-Learning 的变种来解决上述不稳定的问题，主要用到两个 idea：

1. 我们利用由生物学启发出来的机制，称为：experience replay，在数据中生成不规则分布。从而消除了观测序列的相关性，使得在数据分布上光滑的变化。

2. 我们利用一个迭代更新来朝向目标值(target values) 调整 动作-值 Q (the action-value)，仅仅周期性更新，从而减少了与目标的联系。

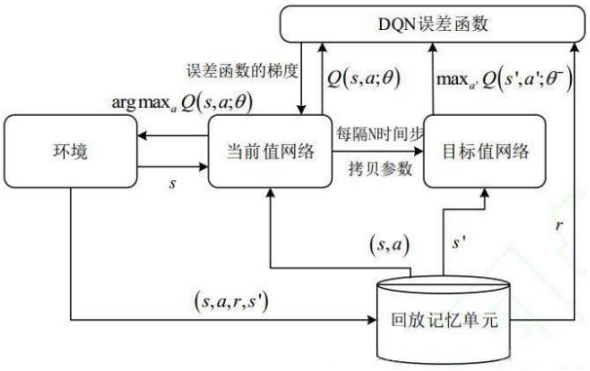


图 62 DQN 算法架构

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

图 63 DQN 算法流程

2、Rainbow

RainBow[11]算法之所以称之为彩虹算法，在于它集多种 DQN 算法之大成，它融合了以下几种算法：

- Double DQN
- Prioritized experience replay
- Dueling network architecture
- Multi-step bootstrap
- Distributional DQN
- Noisy DQN

RainBow 借鉴了 Distributional DQN，它的输出不再是 Q 值函数，而是 Q 值分布，记为：

$$d_t^{(n)} = (R_t^{(n)} + \gamma_t^{(n)} \boldsymbol{z}, \quad \boldsymbol{p}_{\bar{\theta}}(S_{t+n}, a_{t+n}^*))$$

其中 z 表示离散化的 q 值，$p_{\bar{\theta}}$ 表示其取值概率。于是损失函数变为:

$$D_{\mathrm{KL}}(\Phi_{\boldsymbol{z}} d_t^{(n)} || d_t)$$

其中 Φ_z 表示 z 的投影。在 prioritized experience 中是依据 TD-error 来对样本进行排序的，现在因为不再使用 Q 值函数，TD-error 应该进化为 KL 散度，于是有:

$$p_t \propto \left(D_{\mathrm{KL}}(\Phi_{\boldsymbol{z}} d_t^{(n)} || d_t)\right)^{\omega}$$

RainBow 把 Dueling DQN 的结构和 Distributional DQN 的结构结合起来，因为二者都对网络结构做了调整，所以需要整合：我们从 Dueling DQN 的结构出发，原先是输出一个状态值函数 V(s)和一组动作的优势函数 A(s,a)，现在需要输出分布了。在典型的 Distributional DQN 算法 C51 中，把每种动作对应的 Q 分布划分成 51 个离散值，这里的 51 称之为 atoms。我们设卷积层的输出是 ϕ，value stream 的参数是 η，advantage stream 的参数是 ψ，设 atoms 的个数为 N。那么 value stream v_η 的输出应该是一个 N 维的向量，而 advantage stream a_ψ 的输出应该是一个 N 乘以动作空间维度的矩阵，最后每个 atoms 对应的概率可以表示为：

$$p_\theta^i(s,a) = \frac{\exp(v_\eta^i(\phi) + a_\psi^i(\phi,a) - \overline{a}_\psi^i(s))}{\sum_j \exp(v_\eta^j(\phi) + a_\psi^j(\phi,a) - \overline{a}_\psi^j(s))}$$

为了增加 Agent 的探索能力，一种常用的方法是采用 $\varepsilon - greedy$ 的策略，即以 ε 的概率采取随机的动作，以 $1-\varepsilon$ 的概率采取当前获得价值最大的动作。而另外一种常用的方法是 NoisyNet，该方法通过对参数增加噪声来增加模型的探索能力。噪声通常添加在全连接层，考虑全连接层公式：$y = wx + b$，NoisyNet 变为：

$$\boldsymbol{y} = (\boldsymbol{b} + \mathbf{W}\boldsymbol{x}) + (\boldsymbol{b}_{noisy} \odot \epsilon^b + (\mathbf{W}_{noisy} \odot \epsilon^w)\boldsymbol{x})$$

其中 ε 是随机变量，当然可以假设噪声符合正态分布。

（二）Policy-based方法

1、Policy Gradient

（1）SPG

$$\nabla_\theta J(\theta) \overset{def}{=} \nabla_\theta V^\pi(s_0) \propto \sum_{s\in S} d^\pi(s) \sum_{a\in A} Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s)$$

其中，

$$d^\pi(s) = \lim_{t\to\infty} P(s_t = s | s_0, \pi_\theta)$$

表示从状态 s_0 开始，在策略 π_θ 下经过 t 个时间步后到达状态 s 的概率。

因为：

$$\begin{aligned}
&\nabla_\theta V^\pi(s) = \nabla_\theta \left(\sum_{a\in \mathrm{A}} \pi_\theta(a|s) Q^\pi(s,a)\right) \\
&= \sum_{a\in \mathrm{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \nabla_\theta Q^\pi(s,a)\right) \\
&= \sum_{a\in \mathrm{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \nabla_\theta \sum_{s',r} P(s',r|s,a)(r + V^\pi(s'))\right) \\
&= \sum_{a\in \mathrm{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s',r} P(s',r|s,a) \nabla_\theta V^\pi(s')\right) \\
&= \sum_{a\in \mathrm{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s',r|s,a) \nabla_\theta V^\pi(s')\right)
\end{aligned}$$

考虑一个状态访问序列并将从状态 s 开始在策略 π_θ 下经过 k 个时间步到达状态的概率记为：

$$\rho^\pi(s \to x, k)$$

当 k=0 时，$\rho^\pi(s \to x, k=0) = 1$；

当 k=1 时，$\rho^\pi(s \to x, k=1) = \sum_a \pi_\theta(a|s) P(s'|s,a)$；

当目标是从状态 s 开始依照策略 π_θ 经过 k+1 个时间步最终达到目标状态 x。为实现这个目标，我们可以先从状态 s 开始经过 k 个时间步后到达某个中间状态 s'，然后经过最后一个时间步到达目标状态 x。这样我们就可以递归地计算访问概率：

$$\rho^{\pi}(s \to x, k+1) = \sum_{s'} \rho^{\pi}(s \to s', k)\rho^{\pi}(s' \to x, 1)$$ 。

令 $\phi(s) = \sum_{a \in \mathrm{A}} \nabla_{\theta} \pi_{\theta}(a \mid s) Q^{\pi}(s, a)$

上式变为：

$$\begin{aligned}
&\nabla_{\theta} V^{\pi}(s) \\
&= \phi(s) + \sum_{a \in \mathrm{A}} \pi_{\theta}(a \mid s) \sum_{s'} P(s', r \mid s, a) \nabla_{\theta} V^{\pi}(s') \\
&= \phi(s) + \sum_{s'} \sum_{a} \pi_{\theta}(a \mid s) P(s', r \mid s, a) \nabla_{\theta} V^{\pi}(s') \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \to s', 1) \nabla_{\theta} V^{\pi}(s') \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \to s', 1)[\phi(s') + \sum_{s'} \rho^{\pi}(s' \to s'', 1) \nabla_{\theta} V^{\pi}(s'')] \\
&= \cdots; \text{递归展开} \nabla_{\theta} V^{\pi}(\cdot) \\
&= \sum_{x \in S} \sum_{k=0}^{\infty} \rho^{\pi}(s \to x, k) \phi(x)
\end{aligned}$$

上述变形使得我们无需计算 Q-值函数的梯度。将其带入目标函数 $J(\theta)$ 中，可得：

$$\begin{aligned}
\nabla_{\theta} J(\theta) &= \nabla_{\theta} V^{\pi}(s_0) \\
&= \sum_{s} \sum_{k=0}^{\infty} \rho^{\pi}(s_0 \to s, k) \phi(s) \qquad ; \text{令} \eta(s) = \sum_{k=0}^{\infty} \rho^{\pi}(s_0 \to s, k) \\
&= \sum_{s} \eta(s) \phi(s) \\
&= \left(\sum_{s} \eta(s) \right) \sum_{s} \frac{\eta(s)}{\sum_{s} \eta(s)} \phi(s) \qquad ; \sum_{s} \eta(s) \text{是一个常数} \\
&\propto \sum_{s} \frac{\eta(s)}{\sum_{s} \eta(s)} \phi(s) \\
&= \sum_{s} d^{\pi}(s) \sum_{a} \nabla_{\theta} \pi_{\theta}(a \mid s) Q^{\pi}(s, a) \quad ; d^{\pi}(s) = \frac{\eta(s)}{\sum_{s} \eta(s)} \text{为平稳分布} \\
&= \sum_{s} d^{\pi}(s) \sum_{a} \pi_{\theta}(a \mid s) \frac{\nabla_{\theta} \pi_{\theta}(a \mid s)}{\pi_{\theta}(a \mid s)} Q^{\pi}(s, a) \\
&= \sum_{s} d^{\pi}(s) \sum_{a} \pi_{\theta}(a \mid s) \nabla_{\theta} \log \pi_{\theta}(a \mid s) Q^{\pi}(s, a)
\end{aligned}$$

（2）DPG

记累计折扣回报：

$$r_t^{\gamma} = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$$

agent 的训练目标是获取一个策略使累积奖励最大化，目标函数为：

$$J(\pi) = \mathrm{E}[r_1^{\gamma} \mid \pi]$$

从状态 s 经过 t 时间过渡到状态s'的概率密度为$p(s \to s', t, \pi)$。状态分布为：

$$\rho^{\pi}(s') := \int_S \sum_{t=1}^{\infty} \gamma^{t-1} p_1(s) p(s \to s', t, \pi) ds$$

那么，

$$J(\pi_\theta) = \int_S \rho^{\pi}(s) \int_A \pi_\theta(s,a) da ds = \mathrm{E}_{s\sim\rho^{\pi}, a\sim\pi_\theta}[r(s,a)]$$

大部分的无模型强化学习方法是基于通用的策略迭代过程：策略评估和策略提升的交替。策略估计动作值函数$Q^{\pi}(s,a) or Q^{\mu}(s,a)$，比如通过 MC 评估或 TD 学习。策略提升根据(估计)动作值函数更新策略。最常见的方法是贪婪最大化(或者 soft maximization)动作值函数

$$\mu^{k+1}(s) = \underset{a}{argmax} Q^{\mu^k}(s,a)$$

在连续动作空间，贪婪的策略提升每一步都做全局最大化会有问题(因为动作是连续的，不可能做到遍历找到使得 Q 值最大的动作)。一种替代性的方案是：在 Q 值的梯度方向移动策略。特别的，对于每一个访问的状态 s，用$\nabla_\theta Q^{\mu^k}(s, \mu_\theta(s))$更新策略网络的参数$\theta^{k+1}$。每个状态策略提升的方向不一样；这就需要基于状态的分布$\rho^{\mu}(s)$对 Q 值函数的梯度求平均(期望)。

$$\theta^{k+1} = \theta^k + \alpha E_{s\sim\rho^{\mu^k}}[\nabla_\theta Q^{\mu^k}(s, \mu_\theta(s))]$$

通过运用链式法则：

$$\theta^{k+1} = \theta^k + \alpha E_{s\sim\rho^{\mu^k}}[\nabla_\theta \mu_\theta(s) \nabla_\theta Q^{\mu^k}(s,a)|_{a=\mu_\theta(s)}]$$

定义目标函数为：

$$J(\mu_\theta) = \int_S \rho^{\mu}(s) r(s, \mu_\theta(s)) ds = \mathrm{E}_{s\sim\rho^{\mu}}[r(s, \mu_\theta(s))]$$

定理 1(确定性策略梯度理论)：

如果 MDP 过程满足 $p(s'|s,a),\nabla_a p(s'|s,a),\mu_\theta(s),\nabla_\theta\mu_\theta(s),r(s,a),\nabla_a r(s,a),p_1(s)$ 在参数 s,a,s',x 下都是连续的，那么意味着 $\nabla_\theta\mu_\theta(s)$、$\nabla_a Q^\mu(s,a)$ 存在且确定性策略梯度存在。对于任意 θ，状态空间 S 是紧的，因此 $\|\nabla_\theta V^{\mu_\theta}(s)\|$，$\|\nabla_a V^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)}\|$，$\nabla_\theta\mu_\theta(s)$ 都是 s 的有界函数。那么：

$$\begin{aligned}J(\mu_\theta)&=\int_S \rho^\mu(s)\nabla_\theta\mu_\theta(s)\nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\,ds\\&=\mathrm{E}_{s\sim\rho^\mu}[\nabla_\theta\mu_\theta(s)\nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}]\end{aligned}$$

证明：

$$\begin{aligned}\nabla_\theta V^{\mu_\theta}(s)&=\nabla_\theta Q^{\mu_\theta}(s,\mu_\theta(s))\\&=\nabla_\theta\left(r(s,\mu_\theta(s))+\int_S \gamma p(s'|s,\mu_\theta(s))V^{\mu_\theta}(s')ds'\right)\\&=\nabla_\theta\mu_\theta(s)\nabla_a r(s,a)|_{a=\mu_\theta(s)}+\nabla_\theta\int_S \gamma p(s'|s,\mu_\theta(s))V^{\mu_\theta}(s')ds'\\&=\nabla_\theta\mu_\theta(s)\nabla_a r(s,a)|_{a=\mu_\theta(s)}\\&\quad+\int_S \gamma\left(p(s'|s,\mu_\theta(s))\nabla_\theta V^{\mu_\theta}(s')+\nabla_\theta\mu_\theta(s)\nabla_a p(s'|s,\mu_\theta(s))|_{a=\mu_\theta(s)}V^{\mu_\theta}(s')\right)ds'\\&=\nabla_\theta\mu_\theta(s)\nabla_a\left(r(s,a)+\int_S \gamma p(s'|s,\mu_\theta(s))|_{a=\mu_\theta(s)}V^{\mu_\theta}(s')ds'\right)|_{a=\mu_\theta(s)}\\&\quad+\int_S \gamma p(s'|s,\mu_\theta(s))\nabla_\theta V^{\mu_\theta}(s')ds'\\&=\nabla_\theta\mu_\theta(s)\nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)}+\int_S \gamma p(s\to s',1,\mu_\theta)\nabla_\theta V^{\mu_\theta}(s')ds'\end{aligned}$$

迭代这一过程消去 V^{μ_θ}

$$\begin{aligned}\nabla_\theta V^{\mu_\theta}(s)&=\nabla_\theta\mu_\theta(s)\nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)}\\&\quad+\int_S \gamma p(s\to s',1,\mu_\theta)\nabla_\theta\mu_\theta(s')\nabla_a Q^{\mu_\theta}(s',a)|_{a=\mu_\theta(s')}\,ds'\\&\quad+\int_S \gamma p(s\to s',1,\mu_\theta)\int_S \gamma p(s\to s'',1,\mu_\theta)\nabla_\theta V^{\mu_\theta}(s'')ds''ds'\\&=\nabla_\theta\mu_\theta(s)\nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)}\\&\quad+\int_S \gamma p(s\to s',1,\mu_\theta)\nabla_\theta\mu_\theta(s')\nabla_a Q^{\mu_\theta}(s',a)|_{a=\mu_\theta(s')}\,ds'\\&\quad+\int_S \gamma^2 p(s\to s',2,\mu_\theta)\nabla_\theta\mu_\theta(s')ds'\\&\cdots\\&=\int_S\sum_{t=0}^{\infty}\gamma^t p(s\to s',t,\mu_\theta)\nabla_\theta\mu_\theta(s')\nabla_a Q^{\mu_\theta}(s',a)|_{a=\mu_\theta(s')}\,ds'\end{aligned}$$

（3）DDPG

虽然 DQN 解决了高维观察空间的问题，但它只能处理离散和低维动作空间。许多任务，具有连续和高维动作空间。DQN 不能直接应用于

连续域，因为它依赖于找到最大化动作值函数的动作，这在连续值情况下需要在每一个步骤就行迭代优化。DQN 的另一个缺点是，它采取随机的策略，也就是说在给定状态和参数之下，模型输出的动作服从一个概率分布，也就意味着每次走进这个状态的时候，输出的动作可能不同。这会导致智能体的行为有较大的变异性，我们的参数更新方向很可能不是策略梯度的最优方向。与随机策略对应的是确定性策略，随机策略方法在计算指标 J 的时候需要同时对状态和动作空间进行积分，而确定性策略方法因为动作是确定的，因而只需要在状态空间下积分，因此降低了计算复杂度，提高了算法效率。

深度确定性策略梯度 DDPG[12]中将行为策略的探索和学习分开。动作的探索依然采用随机策略，而学习的策略则是确定性的策略。其次，引入了 actor-critic 框架，使用一个单独的值网络来引导策略的学习。最后，其延续了 DQN 算法中的经验复用来对网络进行 off-policy 训练，以最小化样本之间的相关性，降低学习的方差。

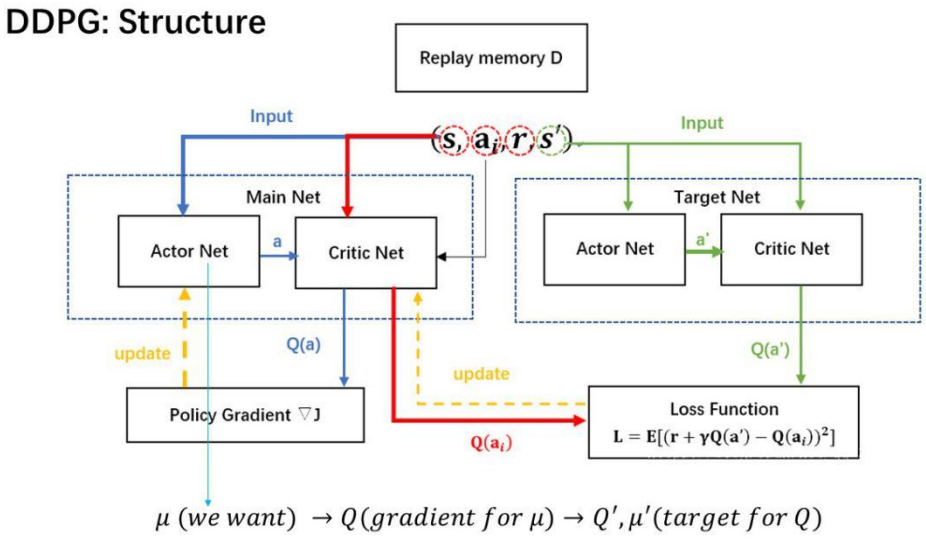


图 64 DDPG 结构图

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

图 65 DDPG 算法

2、Actor-Critic

（1）GAE

广义优势函数估计器[13]（Generalized Advantage Estimator）是将策略梯度算法的策略优势函数的估计$\hat{A}(s_t, a_t)$进行形式上的统一。一般而言，策略梯度算法的梯度估计都遵循如下形式：

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)]$$

其中Ψ_t具有多种形式，比如 PG 中为$\Psi_t = \sum_{i=t}^{\infty} r_i$，即累计回报函数；A2C 中为$\Psi_t = \widehat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$；PPO 中是$\Psi_t = \widehat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$，其中$\delta_t$是时序差分项（Temporal Difference Error），$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$。GAE 将上述若干种估计形式进行统一如下：

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} = \sum_{l=0}^{\infty} (\gamma\lambda)^l (r_t + \gamma V(s_{t+l+1}) - V(s_{t+l}))$$

其中$GAE(\gamma, 0)$的情况为$\widehat{A}_t = \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$，为 1 步时

序差分误差，$GAE(\gamma,1)$的情况为$\widehat{A}_t = \sum_{l=0}^{\infty}\gamma^l\delta_{t+l} = \sum_{l=0}^{\infty}\gamma^l r_{t+l} - V(s_t)$，即为 A2C 中的估计项。PG 中的估计项即为 A2C 中$V(s_t)$恒为 0 的特殊情况。

（2）DPPO

DPPO[14]是 PPO 模型的分布式版本，相较于原始 PPO，DPPO 加入了几个优化：

在状态上加入了 RNN，能够兼容观察状态的时序性，在 POMDP 问题上能够有较好的效果。

在回报的计算上加入了 K 步奖励算法。在以往的计算中，$\widehat{A}_t = r_{t+1} + \gamma V_\varphi(s_{t+1}) - V_\varphi(s_t)$。而在 K 步奖励算法中，$\widehat{A}_t = \sum_{i=1}^{K}\gamma^{i-1}r_{t+1} + \gamma^{K-1}V_\varphi(s_{t+K}) - V_\varphi(s_t)$，相当于在原来的基础上，再往后走了几步，多使用了一些真实的奖励。

对数据进行归一化。首先，在整个实验中，在观察状态上进行了归一化（减去平均值，再除以标准差）。同样，在整个实验中，对奖励进行了归一化（除以标准差）。并在每个批量中对优势进行了归一化。

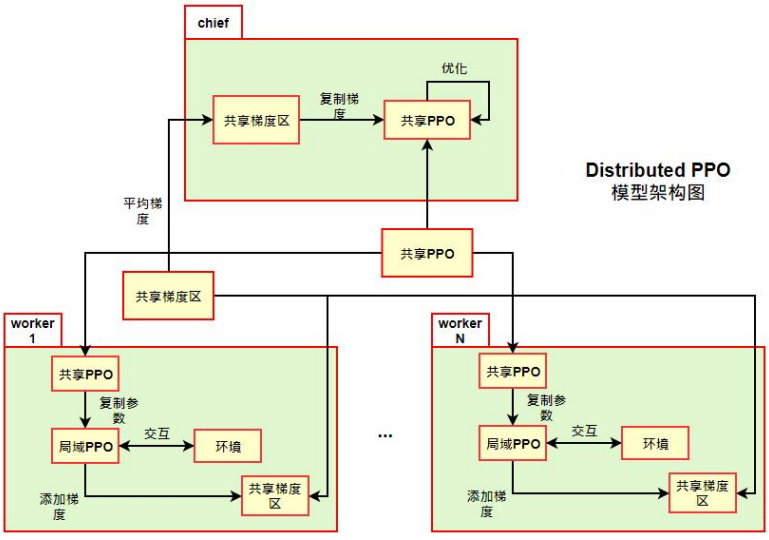


图 66 DPPO 训练架构图

如图 66 所示，DPPO 框架中含有一个 chief 线程和多个 worker 线程。chief 线程是主线程，worker 线程为子线程。多个 worker 线程之间可以并行运行，从而达到分布式计算的目的。全局只有一个共享梯度区和共享 PPO 模型，而不同的 worker 中还有自己局域 PPO 模型和局域环境。

在该框架中只有一个共享 PPO 模型，需要将其传入 chief 和 worker 中。该模型的作用是根据得到的梯度使用优化器更新参数。而每个 worker 中有一个自己的局域 PPO 模型，该模型的作用是，使用 PPO 策略和局域环境进行互动，得到经验，并在更新中计算梯度。共享梯度区的作用是存储所有 worker 在更新中计算得到的梯度和，然后将梯度对应取平均，赋值给共享 PPO。

算法的具体流程如下。首先，局域 PPO 和环境互动，使用 K 步奖励方法计算回报。然后计算优势，最后存储经验到本地。在更新步骤中，使用 PPO 更新的两种方法（KL penalty 或者 clip）计算策略目标函数 $J_{PPO}(\theta)$。计算策略梯度$\nabla_{\theta}J_{PPO}(\theta)$，并将其加入到共享梯度区，之后等待 chief 的信号。共享梯度区在等待一定数目的 worker 传送梯度之后，在 chief 中将梯度传递给共享 PPO。共享 PPO 使用梯度更新，然后通知各个 worker 中的局域 PPO 从共享 PPO 中拷贝网络参数。局域 PPO 开展和环境之间的下一阶段的互动。

Algorithm 2 Distributed Proximal Policy Optimization (chief)

for $i \in \{1, \cdots, N\}$ **do**
  for $j \in \{1, \cdots, M\}$ **do**
    Wait until at least $W - D$ gradients wrt. θ are available
    average gradients and update global θ
  end for
  for $j \in \{1, \cdots, B\}$ **do**
    Wait until at least $W - D$ gradients wrt. ϕ are available
    average gradients and update global ϕ
  end for
end for

图 67 DPPO chief

Algorithm 3 Distributed Proximal Policy Optimization (worker)

for $i \in \{1, \cdots, N\}$ **do**
  for $w \in \{1, \cdots T/K\}$ **do**
    Run policy π_θ for K timesteps, collecting $\{s_t, a_t, r_t\}$ for $t \in \{(i-1)K, \ldots, iK-1\}$
    Estimate return $\hat{R}_t = \sum_{t=(i-1)K}^{iK-1} \gamma^{t-(i-1)K} r_t + \gamma^K V_\phi(s_{iK})$
    Estimate advantages $\hat{A}_t = \hat{R}_t - V_\phi(s_t)$
    Store partial trajectory information
  end for
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $m \in \{1, \cdots, M\}$ **do**
    $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \mathrm{KL}[\pi_{old}|\pi_\theta] - \xi \max(0, \mathrm{KL}[\pi_{old}|\pi_\theta] - 2\mathrm{KL}_{target})^2$
    if $\mathrm{KL}[\pi_{old}|\pi_\theta > 4\mathrm{KL}_{target}$ **then**
      break and continue with next outer iteration $i+1$
    end if
    Compute $\nabla_\theta J_{PPO}$
    send gradient wrt. to θ to chief
    wait until gradient accepted or dropped; update parameters
  end for
  for $b \in \{1, \cdots, B\}$ **do**
    $L_{BL}(\phi) = -\sum_{t=1}^{T} (\hat{R}_t - V_\phi(s_t))^2$
    Compute $\nabla_\phi L_{BL}$
    send gradient wrt. to ϕ to chief
    wait until gradient accepted or dropped; update parameters
  end for
  if $\mathrm{KL}[\pi_{old}|\pi_\theta] > \beta_{high}\mathrm{KL}_{target}$ **then**
    $\lambda \leftarrow \tilde{\alpha}\lambda$
  else if $\mathrm{KL}[\pi_{old}|\pi_\theta] < \beta_{low}\mathrm{KL}_{target}$ **then**
    $\lambda \leftarrow \lambda/\tilde{\alpha}$
  end if
end for

图 68 DPPO worker

（3）IMPALA

IMPLALA[15]是一个大规模强化学习训练的框架，具有较高的性能（high throughput）、较好的扩展性（scalability）和较高的效率（data-efficiency）。在大规模计算的框架下，采样和策略更新会有一些错位（policy lag），在这种情况下，Impala 通过 V-trace 技术来兼容 on-policy 和 off-policy 样本进行训练。

首先，我们来看一个 single learner 的模式。learner 的主要作用是通过获取 actor 得到的轨迹来做 SGD 来更新各个神经网络的参数，神经网络训练本身可并行的特性，learner 使用的是一块 GPU。actor 定期从 learner 获取最新的神经网络参数，并且每个 actor 起一个模拟环境，来

使用自己能获得的最新策略去采样，并且把获取到的$\{x_t, a_t, r_t, \mu(a_t|x_t)\}$传回供 learner 去更新各个神经网络参数。由于模拟环境的运行通常不方便做并行，actor 一般使用 CPU。由于 actor 上的策略μ可能不是 learner 中最新的策略π，因此这里使用了不同的符号来表示。

下一步，当训练规模扩大的时候，可以考虑使用多个 learner（多块 GPU）并且每块 GPU 配套多个 actor（CPU）。每个 learner 只从自己的 actor 们中获取样本进行更新，learner 之间定期交换 gradient 并且更新网络参数，actor 也定期从任意 learner 上获取并更新神经网络参数。

IMPALA 中 actor 和 learner 相互异步工作，极大提高了时间利用率。文章还与与 batched A2C 做了对比，如图 69 所示。a 图中，正向传播和反向传播都想凑一批来做（可能是给到 GPU 来算），因此每一步都需要同步，而模拟环境各步所需时间方差很大，这样浪费了大量的等待时间；b 图中，只把耗时较长的反向传播凑一批来做，正向传播就给各个 actor 自己做；而 c 图中的 IMPALA 则完全把 actor 和 learner 分开异步进行，这样 actor 不用去等待别的 actor，可以尽可能多的做采样，相应地，所作出的牺牲就是每次更新得到的样本变为了 off-policy 样本。

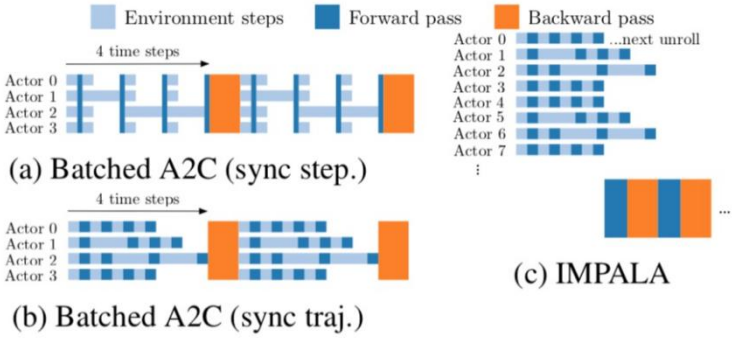


图 69

V-trace 根据采样到的 trajectory 和当前的 Value net 给出当前值函数的一个估计v_s:

$$v_s \overset{def}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \left(\prod_{i=s}^{t-1} c_i\right) \delta_t V$$

其中，$\delta_t V \overset{def}{=} \rho_t (r_t + \gamma V(x_{t+1}) - V(x_t))$是 V 的 TD 误差。$\rho_t \overset{def}{=} \min(\overline{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)})$

和 $c_i \overset{def}{=} \min(\bar{c}, \frac{\pi(a_i \mid x_i)}{\mu(a_i \mid x_i)})$ 表示截断的重要性采样。为什么要进行截断？因为 π 与 μ 实际的分布差异越大，最后估计的策略的差异就会越大，而为了更好地估计相应的策略分布，我们采用一步步累积的方法进行 s 到 t 步的逐步估计，这样将导致估计的方差会越来越大，所以才需要进行截断，采用相应策略之比的平均值来稳定方差，使训练的效果更加稳定。

注意到，在 on-policy 情形下（$\pi = \mu$），并且假设 $\bar{\rho} \geq \bar{c} \geq 1$，因此 V-trace targets 可以重写为：

$$\begin{aligned} v_s &= V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s}(r_t + \gamma V(x_{t+1}) - V(x_t)) \\ &= \sum_{t=s}^{s+n-1} \gamma^{t-s} r_t + \gamma^n V(x_{s+n}) \end{aligned}$$

另外，V-trace targets 可以进行迭代计算：

$$v_s = V(x_s) + \delta_t V + \gamma c_s (v_{s+1} - V(x_{s+1}))$$

Impala 采用 actor-critic 架构，critic 的更新方式为最小化拟合的价值函数 V 相对于 V-trace targets 的均方误差，即朝如下方式更新：

$$(v_s - V_\theta(x_s)) \nabla_\theta V_\theta(x_s)$$

在 on-policy 情形下，值函数的梯度为，

$$\nabla V^\mu(x_0) = E_\mu[\sum_{s \geq 0} \gamma^s \nabla \log \mu(a_s \mid x_s) Q^\mu(x_s, a_s)]$$

其中，$Q^\mu(x_s, a_s) \overset{\text{def}}{=} E_\mu[\sum_{t \geq s} \gamma^{t-s} r_t | x_s, a_s]$ 是策略 u 的状态动作值函数。Actor 朝着 on-policy policy gradient 给出的梯度方向更新，即，

$$E_{a_s \sim \mu(\cdot \mid x_s)}[\nabla \log \mu(a_s \mid x_s) q_s \mid x_s]$$

其中 q_s 是对 $Q^\mu(x_s, a_s)$ 的估计值。

在 off-policy 情形下，行为策略和目标策略会出现不一致的情形,因此 actor 朝着 off-policy policy gradient 给出的梯度方向更新，即，

$$E_{a_s \sim \mu(\cdot \mid x_s)}[\frac{\pi(a_s \mid x_s)}{\mu(a_s \mid x_s)} \nabla \log \mu(a_s \mid x_s) q_s \mid x_s]$$

这里使用 $q_s \overset{\text{def}}{=} r_s + \gamma v_{s+1}$ 来估计 $Q^\pi(x_s, a_s)$。

（三）Policy embedding

传统的 off-policy actor-critic 学习的是单一目标策略的价值函数。然而，在价值函数沿着目标策略更新的过程中，很可能遗忘了老策略的一些有用信息，因此一种对策略进行表征的算法 PBVFs[16]（parameter-based value functions）的算法被提出来，用于解决策略泛化性的问题。

1、PSVF

首先，我们推导 PSSVF（parameter-based start-state value functions）$V(\theta)$。给出算法表现的评估指标 J，然后对策略网络的参数 theta 求导，得出：

$$\nabla_\theta J(\pi_\theta) = \int_S \mu_0(s) \nabla_\theta V(s,\theta) ds = E_{s\sim\mu_0(s)}[\nabla_\theta V(s,\theta)] = \nabla_\theta V(\theta)$$

我们使用 MC 方法去估计任何策略 θ 的 critic 评估值 $V_w(\theta)$，然后 actor 沿着 critic 提升的方向更新参数。

Algorithm 1 Actor-critic with Monte Carlo prediction for $V(\theta)$

Input: Differentiable critic $V_{\mathbf{w}} : \Theta \to \mathcal{R}$ with parameters $\mathbf{w}$; deterministic or stochastic actor π_θ with parameters θ; empty replay buffer D
Output : Learned $V_{\mathbf{w}} \approx V(\theta) \forall \theta$, learned $\pi_\theta \approx \pi_{\theta^*}$
Initialize critic and actor weights $\mathbf{w}, \theta$
repeat:
  Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$ with policy π_θ
  Compute return $r = \sum_{k=1}^{T} r_k$
  Store (θ, r) in the replay buffer D
  for many steps **do**:
    Sample a batch $B = \{(r, \theta)\}$ from D
    Update critic by stochastic gradient descent: $\nabla_{\mathbf{w}} \mathbb{E}_{(r,\theta)\in B}[r - V_{\mathbf{w}}(\theta)]^2$
  end for
  for many steps **do**:
    Update actor by gradient ascent: $\nabla_\theta V_{\mathbf{w}}(\theta)$
  end for
until convergence

图 70

使用 MC 去估计价值函数会导致其方差很大，另外类似 Algorithm 1 这种 episode-based 的算法无法从坏的 episode 中去评判好的动作。基于 TD 更新的梯度方法在估计 V 函数时引入了偏差，但是在很大程度上

降低了方差，而且在每一步迭代可以评判动作的好坏。对 PSVF（parameter-based state-value function）的$J_b(\pi_\theta)$求导，可以得到：

$$\nabla_\theta J_b(\pi_\theta) = \int_S d_\infty^{\pi_b}(s)\nabla_\theta V(s,\theta)ds = E_{s\sim d_\infty^{\pi_b}(s)}[\nabla_\theta V(s,\theta)]$$

Algorithm 2 Actor-critic with TD prediction for $V(s,\theta)$

Input: Differentiable critic $V_{\mathbf{w}} : \mathcal{S} \times \Theta \to \mathcal{R}$ with parameters $\mathbf{w}$; deterministic or stochastic actor π_θ with parameters θ; empty replay buffer D

Output : Learned $V_{\mathbf{w}} \approx V(s,\theta)$, learned $\pi_\theta \approx \pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w}, \theta$

repeat:

Observe state s, take action $a = \pi_\theta(s)$, observe reward r and next state s'

Store (s, θ, r, s') in the replay buffer D

if it's time to update **then**:

for many steps **do**:

Sample a batch $B_1 = \{(s, \tilde{\theta}, r, s')\}$ from D

Update critic by stochastic gradient descent:

$\nabla_{\mathbf{w}} \frac{1}{|B_1|} \mathbb{E}_{(s,\tilde{\theta},r,s')\in B_1}[V_{\mathbf{w}}(s,\tilde{\theta}) - (r + \gamma V_{\mathbf{w}}(s',\tilde{\theta}))]^2$

end for

for many steps **do**:

Sample a batch $B_2 = \{(s)\}$ from D

Update actor by stochastic gradient ascent: $\nabla_\theta \frac{1}{|B_2|} \mathbb{E}_{s\in B_2}[V_{\mathbf{w}}(s,\theta)]$

end for

end if

until convergence

图 71

2、PAVF

PAVF（parameter-based action-value function）是在 policy gradient 和 off-policy actor-critic 理论基础上发展出来的，分为 stochastic policy gradient 和 deterministic policy gradient 两部分内容。

stochastic policy gradient：如果我们想用随机行为策略π_b收集的数据学习目标策略π_θ的动作值函数。传统的 off-policy actor-critic 算法仅仅近似计算J_b的梯度，而没有估计动作值函数对目标策略的参数的梯度$\nabla_\theta Q^{\pi_\theta}(s,a)$。PBVF 会直接计算这方面梯度的贡献，从而推导出J_b的完全策略梯度理论：

$$\nabla_\theta J_b(\pi_\theta) = E_{s\sim d_\infty^{\pi_b}(s), a\sim\pi_b(\cdot|s)}[\frac{\pi_\theta(a\,|\,s)}{\pi_b(a\,|\,s)}(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a\,|\,s) + \nabla_\theta Q(s,a,\theta))]$$

Deterministic policy gradient：估计随机策略的$Q(s,a,\theta)$是很困难的。确定性策略方法在学习 value function 时可以提高学习效率，因为不

需要在动作空间上对值函数求期望！因此，

$$J_b(\pi_\theta)=\int_S d_\infty^{\pi_b}(s)V(s,\theta)ds=\int_S d_\infty^{\pi_b}(s)Q(s,\pi_\theta(s),\theta)ds$$

对θ求梯度：

$$\nabla_\theta J_b(\pi_\theta)=E_{s\sim d_\infty^{\pi_b}(s)}[\nabla_\theta(Q(s,a,\theta)|_{a=\pi_\theta(s)}\ \nabla_\theta\pi_\theta(s)+\nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)})]$$

Algorithm 3 Stochastic actor-critic with TD prediction for $Q(s,a,\theta)$

Input: Differentiable critic $Q_{\mathbf{w}}:\mathcal{S}\times\mathcal{A}\times\Theta\to\mathcal{R}$ with parameters $\mathbf{w}$; stochastic differentiable actor π_θ with parameters θ; empty replay buffer D

Output : Learned $Q_{\mathbf{w}}\approx Q(s,a,\theta)$, learned $\pi_\theta\approx\pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w},\theta$

repeat:

Observe state s, take action $a=\pi_\theta(s)$, observe reward r and next state s'

Store (s,a,θ,r,s') in the replay buffer D

if it's time to update **then**:

for many steps **do**:

Sample a batch $B_1=\{(s,a,\tilde{\theta},r,s')\}$ from D

Update critic by stochastic gradient descent:

$\nabla_{\mathbf{w}}\frac{1}{|B_1|}\mathbb{E}_{(s,a,\tilde{\theta},r,s')\in B_1}[Q_{\mathbf{w}}(s,a,\tilde{\theta})-(r+\gamma Q_{\mathbf{w}}(s',a'\sim\pi_{\tilde{\theta}}(s'),\tilde{\theta}))]^2$

end for

for many steps **do**:

Sample a batch $B_2=\{(s,a,\tilde{\theta})\}$ from D

Update actor by stochastic gradient ascent:

$\frac{1}{|B_2|}\mathbb{E}_{(s,a,\tilde{\theta})\in B_2}\left[\frac{\pi_\theta(a|s)}{\pi_{\tilde{\theta}}(a|s)}(Q(s,a,\theta)\nabla_\theta\log\pi_\theta(a|s)+\nabla_\theta Q(s,a,\theta))\right]$

end for

end if

until convergence

图 72

Algorithm 4 Deterministic actor-critic with TD prediction for $Q(s,a,\theta)$

Input: Differentiable critic $Q_{\mathbf{w}}:\mathcal{S}\times\mathcal{A}\times\Theta\to\mathcal{R}$ with parameters $\mathbf{w}$; differentiable deterministic actor π_θ with parameters θ; empty replay buffer D

Output : Learned $Q_{\mathbf{w}}\approx Q(s,a,\tilde{\theta})$, learned $\pi_\theta\approx\pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w},\theta$

repeat:

Observe state s, take action $a=\pi_\theta(s)$, observe reward r and next state s'

Store (s,a,θ,r,s') in the replay buffer D

if it's time to update **then**:

for many steps **do**:

Sample a batch $B_1=\{(s,a,\tilde{\theta},r,s')\}$ from D

Update critic by stochastic gradient descent:

$\nabla_{\mathbf{w}}\frac{1}{|B_1|}\mathbb{E}_{(s,a,\tilde{\theta},r,s')\in B_1}[Q_{\mathbf{w}}(s,a,\tilde{\theta})-(r+\gamma Q_{\mathbf{w}}(s',\pi_{\tilde{\theta}}(s'),\tilde{\theta}))]^2$

end for

for many steps **do**:

Sample a batch $B_2=\{(s)\}$ from D

Update actor by stochastic gradient ascent:

$\frac{1}{|B_2|}\mathbb{E}_{s\in B_2}[\nabla_\theta\pi_\theta(s)\nabla_a Q_{\mathbf{w}}(s,a,\theta)|_{a=\pi_\theta(s)}+\nabla_\theta Q_{\mathbf{w}}(s,a,\theta)|_{a=\pi_\theta(s)}]$

end for

end if

until convergence

图 73

（四）多智能体方法

1、VDN

基于值函数的方法可以说是多智能体强化学习算法最开始的尝试（例如 Independent Q-Learning, IQL 算法）。虽然 IQL 算法与 DQN 算法结合能够在多智能体问题上取得比较好的效果，但是对于较为复杂的环境，IQL 还是无法很好地处理由于环境非平稳而带来的问题。而中心化的方法，即将所有智能体的状态空间和动作空间合并，当作一个智能体来考虑的方法，虽然能够较好地处理非平稳性问题，但存在一下缺陷：

1、在大规模多智能体环境中算法可扩展性较差；

2、由于多智能体联合训练，一旦某个智能体较早学习到一些有用的策略，则其余智能体会选择较为懒惰的策略。这是因为其余智能体由于进度较慢，从而做出的策略会阻碍已经学到一些策略的智能体，从而是的全局回报下降。

VDN[17]方法的基本思想是，中心化地训练一个联合的 Q-network，但是这个联合的网络是由所有智能体局部的 Q-network 加和得到，这样不仅可以通过中心化训练处理由于环境非平稳带来的问题，而且由于实际是在学习每个智能体的局部模型，因而解耦智能体之间复杂的相互关系。最后，由于训练完毕后每个智能体拥有只基于自己局部观察的 Q-network，可以实现去中心化执行，即 VDN 遵循 CTDE 框架，并且解决的是 Dec-POMDP 问题。

具体来说，VDN 做出了如下假设：

$$Q((h^1,h^2,...,h^d),(a^1,a^2,...,a^d)) \approx \sum_{i=1}^{d} \tilde{Q}_i(h^i,a^i)$$

这里之所以使用 h 而不是 s 是因为我们解决的是 POMDP 问题，并且也因为此，这里的 Q-network 是使用 LSTM 构建。上述分解满足一个很好的性质，即对左边的联合 Q function 进行 argmax 操作，等价于对右边每一个局部 Q function 分别进行 argmax。这样可以保证训练完毕后

去中心化执行时，即使整个系统只基于局部观察进行决策，其策略也是与基于全局观察进行决策是一致的。

下面对上式进行简单的推导。假定整个多智能体系统中包含两个智能体，并且全局回报函数是每个智能体的局部回报函数的加和，$r(s,a)=r_1(o^1,a^1)+r_2(o^2,a^2)$，那么有：

$$
\begin{aligned}
&Q^{\pi}(s,a)=\mathrm{E}\left[\sum_{t=1}^{\infty}\gamma^{t-1}r(s_t,a_t)\,|\,s_1=s,a_1=a;\pi\right]\\
&=\mathrm{E}\left[\sum_{t=1}^{\infty}\gamma^{t-1}r_1(o_t^1,a_t^1)\,|\,s_1=s,a_1=a;\pi\right]+\mathrm{E}\left[\sum_{t=1}^{\infty}\gamma^{t-1}r_2(o_t^2,a_t^2)\,|\,s_1=s,a_1=a;\pi\right]\\
&=:\overline{Q}_1^{\pi}(s,a)+\overline{Q}_2^{\pi}(s,a)
\end{aligned}
$$

分解后的 Q 函数是基于全局观察的。这里可以假设$\bar{Q}_1^{\pi}(s,a)$相比于(o^2,a^2)会更加依赖于(o^1,a^1)，并且使用(o^1,a^1)不能准确估计$\bar{Q}_1^{\pi}(s,a)$，但由于使用的网络结构是 LSTM，那么估计误差可以缩小，并且还可以通过智能体之间的通信来进一步减小误差，所以 VDN 假设：

$$
Q^{\pi}(s,a)=:\overline{Q}_1^{\pi}(s,a)+\overline{Q}_2^{\pi}(s,a)\approx\widetilde{Q}_1^{\pi}(h^1,a^1)+\widetilde{Q}_2^{\pi}(h^2,a^2)
$$

VDN 采用了 parameter sharing 方法，即所有智能体之间参数共享。

2、QMIX

QMIX[18]是一个多智能体强化学习算法，具有如下特点：1、学习得到分布式策略；2、本质是一个值函数逼近算法；3、由于对一个联合动作-状态只有一个总奖励值，而不是每个智能体得到一个自己的奖励值，因此只能用于合作环境，而不能用于竞争对抗环境；4、QMIX 算法采用集中式学习，分布式执行应用框架。通过集中式的信息学习，得到每个智能体的分布式策略；5、训练时用全局状态信息来提高算法效果。是对 VDN 方法的改进；6、QMIX 设计一个神经网络来整合每个智能体的局部值函数而得到联合动作值函数，VDN 是直接求和；7、每个智能体的局部值函数只需要自己的局部观测，因此整个系统在执行时是一个分布式的，通过局部值函数，选出累积期望奖励最大的动作执行；8、算法使联合动作值函数与每个局部值函数的单调性相同，因此对局部值函数取最

大动作也是使联合动作值函数最大；9、算法针对的模型是一个分布式多智能体部分可观测马尔可夫决策过程（Dec-POMDP）。

QMIX 采用一个混合网络对单智能体局部值函数进行合并，并在训练学习过程中加入全局状态信息辅助，来提高算法性能。为了能够沿用 VDN 的优势，采用集中式训练，得到分布式策略。主要是因为对联合动作值函数取 argmax 等价于对每个局部动作值函数取 argmax，其单调性相同，如下所示

$$\arg\max_u Q_{tot}(\tau,u)=\begin{pmatrix}\arg\max_{u_1} Q_1(\tau_1,u_1)\\ \vdots \\ \arg\max_{u_n} Q_n(\tau_n,u_n)\end{pmatrix}$$

因此分布式策略就是贪心地通过局部 Q_i 获取最有动作。QMIX 将上式转化为一种单调性约束：

$$\frac{\partial Q_{tot}}{\partial Q_i}\geq 0,\forall i\in\{1,2,...,n\}$$

为了实现了这种约束，QMIX 采用混合网络（mixing network）来实现，也就是使 mixing network 参数 W 非负，具体结构如下（a）：

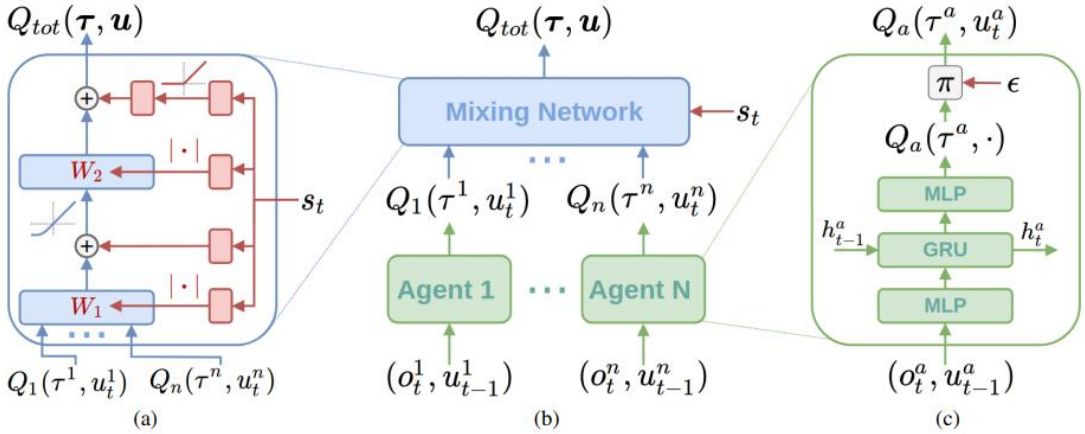


Figure 2. (a) Mixing network structure. In red are the hypernetworks that produce the weights and biases for mixing network layers shown in blue. (b) The overall QMIX architecture. (c) Agent network structure. Best viewed in colour.

图 74

图（c）表示每个智能体采用一个 DRQN 来拟合自身的 Q 值函数 $Q_i(\tau_i,a_i;\theta_i)$，DRQN 循环输入当前的观测$o_{i,t}$以及上一时刻的动作$a_{i,t-1}$来

得到 Q 值。图（a）表示混合网络的结构，其输入为每个 DRQN 网络的输出。为了满足上述的单调性约束，混合网络的所有权值都是非负数，对偏移量不做限制，这样可以确保满足单调性约束。

为了能够更多地利用到系统的状态信息s_t，采用一种超网络（hyper-network），将状态s_t作为输入，输出为混合网络的权值及偏移量。为了保证权值的非负性，采用一个线性网络以及绝对值激活函数保证偏导不为负数。对偏移量采用同样方式但没有非负性的约束，混合网络最后一层的偏移量通过两层网络以及 ReLu 激活函数得到非线性映射网络。由于状态信息s_t是通过超网络混合到Q_{tot}中的，而不是仅仅作为混合网络的输入项，这样带来一个好处是，如果作为输入项则s_t的系数均为正，这样则无法充分利用状态信息来提高系统性能，相当于舍弃一半的信息量。

QMIX 最终的损失函数为：

$$L(\theta)=\sum_{i=1}^{b}[(y_i^{tot}-Q_{tot}(\tau,a,s;\theta))^2]$$

其中$y^{tot}=r+\gamma\max_{a'}\bar{Q}(\tau',a',s';\bar{\theta})$，$\bar{Q}(\tau',a',s';\bar{\theta})$表示目标网络。由于满足上文的单调性约束，对$Q_{tot}$进行 argmax 操作的计算量就不再是随着智能体数量呈指数增长了，而是随智能体数量线性增长，极大提高了算法效率。

3、分层强化学习

MAXQ[19]方法的全名是 MAXQ Value Function Decomposition，是对 Value function 进行分解，它与 VDN 算法的区别在于，前者是基于 task decomposition，在 task 的 sub-task 之间做值函数分解；后者是在 multi-agent 之间做值函数分解。

同样基于 SMDP/task abstraction 的思想，MAXQ 方法做的第一件事是 Core MDP decomposition，即给定一个任务 MDP，把它分解为多个子任务，子任务对应着多个 MDP，

$$\{M_0, M_1, ..., M_n\}$$

既然大的 MDP 被分解为若干子任务，小的 MDP 就有自己的起始状态、终止状态、动作集合和奖励函数，$\langle T_i, A_i, \widetilde{R}_i \rangle$。类似于 options 算法，$T_i$是子任务 i 的终止状态集合，可以看成是一个 termination predict 问题；A_i是子任务 i 下的动作集合，$\widetilde{R}_i$代表 pseudo-reward function 也就是子任务内部的 reward 函数。这个函数一般是为了促进子任务的学习，不同的子任务可以设计不同的 pseudo-reward（先验知识）。

整个分层架构是一个从上而下的 call-and-return（调用返回），一层层向下执行，直到 sub-task 完成才返回。对应 core MDP 的 hierarchical 分解，有 hierarchical policies，每个 policy 对应一个 MDP，$\pi = \{\pi_0, \pi_1, \ldots, \pi_n\}$。

接下来是 MAXQ 的价值函数分解，在 MAXQ 原文中提出两个 value function：projected value function 和 complete function。projected value function 有如下递归定义：

$$V^{\pi}(i,s) = V^{\pi}(\pi_i(s), s) + \sum_{s',\tau} P_i^{\pi}(s', \tau | s, \pi_i(s)) \gamma^{\tau} V^{\pi}(i, s')$$

$V^{\pi}(i,s)$表示 hierarchical 策略π，在状态 s 下执行任务 i，依据对应的策略π_i从 s 执行到终止获得的累计收益的期望。进一步得到 Q 函数：

$$Q^{\pi}(i,s,a) = V^{\pi}(a,s) + \sum_{s',\tau} P_i^{\pi}(s', \tau | s, a) \gamma^{\tau} Q^{\pi}(i, s', \pi(s'))$$

然后引出 complete function 的概念：

$$C^{\pi}(i,s,a) = \sum_{s',\tau} P_i^{\pi}(s', \tau | s, a) \gamma^{\tau} Q^{\pi}(i, s', \pi(s'))$$

则 Q 可以写成：

$$Q^{\pi}(i,s,a) = V^{\pi}(a,s) + C^{\pi}(i,s,a)$$

$Q^{\pi}(i,s,a)$表示 hierarchical 策略π，在 s 状态下执行任务 i，先执行任务 i 下的子任务 a 得到收益$V^{\pi}(a,s)$，然后沿着π完成整个任务 i 的收益$C^{\pi}(i,s,a)$。这里需要注意的是，由于子任务与子任务之间也存在层级结

构，例如这里的子任务 i 包含着子任务 a，a 是表示任务 i 的策略π_i的子策略（子动作）π_a，$V^{\pi}(a,s)$其实可以看成是执行策略π_a的 reward，即：

$$R_i(s,a) = V^{\pi}(a,s)$$

再从任务 i 这一层的 MDP 看，去掉函数表示中的 i，就是很标准的 Q 函数的形式了。

基于以上的定义，来描述价值函数分解。给定一个 core MDP M 的状态 s，和 hierarchical policy π，假设最高层的策略π_0选了任务a_1，下层策略π_1选了任务a_2，然后 n-1 层策略$\pi_{1,\dots}$选了原子操作a_n，则：

$$V^{\pi}(0,s) = V^{\pi}(a_n,s) + C^{\pi}(a_{n-1},s,a_n) + \cdots + C^{\pi}(a_1,s,a_2) + C^{\pi}(0,s,a_1)$$

$$V^{\pi}(a_n,s) = \sum_{s'} P(s'|s,a_n)R(s'|s,a_n)$$

通过 projected value function 和 complete function 的拆解，low-level 和 high-level 之间的值函数从而能够层次分解表示。有了上式的分解，通过采样轨迹，不断递归使用 SMDP Q-learning 更新子任务的 complete function，从而学习整个层次任务。

（五）零和博弈框架

1、Policy Evaluation

矩阵博弈与纳什均衡

如果两个玩家完全竞争，则该双人矩阵博弈称为零和博弈。在期望回报上，零和博弈只有唯一的纳什均衡。这意味着，尽管在零和博弈中每个玩家可能具有多种纳什均衡策略（混合策略），但在这些纳什均衡策略下，期望回报值 V 均相同。在一般和博弈中，纳什均衡不再唯一，可能具有多个纳什均衡。

对于一个两人的零和博弈，如果玩家 1 有 m 种策略，玩家 2 有 n 种策略，以矩阵$A_{m\times n}$表示玩家 1 的 payoff 矩阵，由于是零和博弈，玩家 2 的 payoff 矩阵是-A。

现在来考虑求解这个博弈的混合策略纳什均衡。

设$p_{1\times m}$是玩家 1 的混合策略，$q_{1\times m}$是玩家 2 的混合策略。那么，对于玩家 1，其混合策略应该是如下线性规划问题的解：

$$\begin{aligned} v_1 &= \max v \\ s.t.\ & p_{1\times m} \geq 0,\ p\cdot\vec{1} = 1 \\ & pA \geq v\cdot\vec{1} \end{aligned} \qquad (1)$$

对于玩家 2，其混合策略应该是如下线性规划问题的解：

$$\begin{aligned} v_2 &= \max v \\ s.t.\ & q_{1\times n} \geq 0,\ p\cdot\vec{1} = 1 \\ & -qA^T \geq v\cdot\vec{1} \end{aligned} \qquad (2)$$

记$\omega = -v$，上面这个问题就可以写作：

$$\begin{aligned} -v_2 &= \min \omega \\ s.t.\ & q_{1\times n} \geq 0,\ p\cdot\vec{1} = 1 \\ & qA^T \leq \omega\cdot\vec{1} \end{aligned} \qquad (3)$$

可以证明，问题（1）和问题（3）互为线性规划的对偶问题，因此它们具有相同的最优解，从而$v_1 = -v_2 \rightarrow v_1 + v_2 = 0$。

（1）Replicator Dynamics

动力系统是一个强大的数学框架，用于对指定玩家行为的时间依赖性建模。比如，二人非对称 meta-game 表示为 Normal-Form Game(2, $S^1 \times S^2$, M=(A,B))，由 replicator dynamics 给出的玩家策略的演进方程为：

$$\dot{x}_i = x_i((Ay)_i - x^TAy) \quad \dot{y}_j = y_j((x^TB)_j - x^TBy) \quad \forall\,(i,j) \in S^1 \times S^2, \qquad (1)$$

x_i和y_i分别代表两个无限大的种群（策略集合）中的比例（概率），每个种群对应一个玩家。这个耦合的微分方程系统，对种群策略集合之间的互动进行动态时间建模，并且可以很容易地扩展到一般的 K-wise 互动情况。

复制动力学可以为博弈的微观动力学特征提供非常实用的洞见性，从动力学方程的相图中我们可以很容易地观察到策略的流动、吸引子和博弈的均衡。

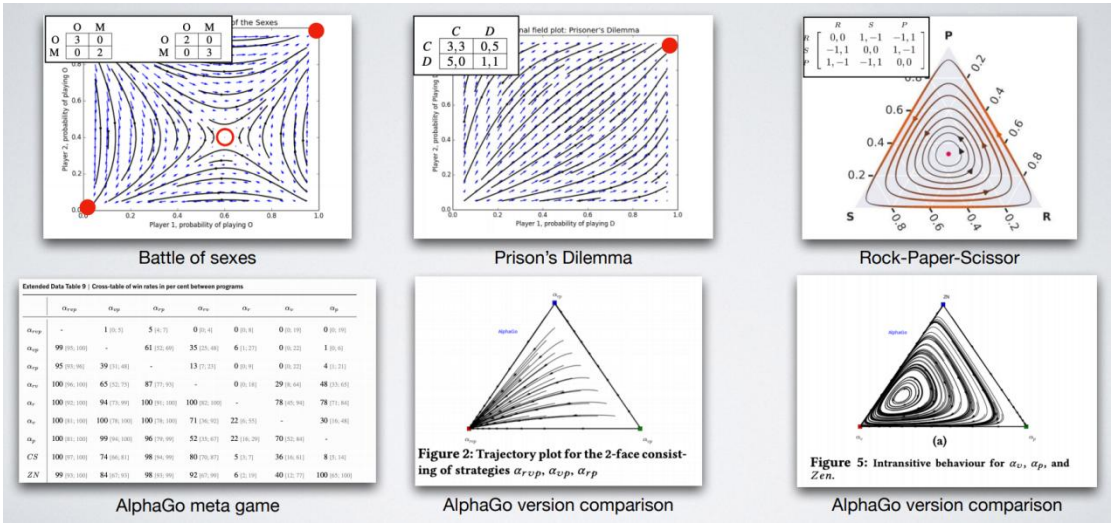


图 75

对于 K-player NFG，假设现在有 K 个种群，S^k 表示种群 $k \in \{1,\ldots,K\}$ 中的纯策略集合。x_i^k 表示种群对应的玩家 players 使用策略 $i \in S^k$，这里 $\sum_{i \in S^k} x_i^k = 1$。S 代表所有种群的纯策略集合，x 表示联合种群状态。定义给定种群 k 的 payoff matrix 为 $M^k: S \to R$。给定状态 x 下种群 k 使用纯策略 i 的适应性（fitness），

$$f_i^k(x) = \sum_{s^{-k} \in S^{-k}} M^k(i, s^{-k}) \prod_{c \neq k} x_{s^c}^c.$$

即，适应性是执行策略 i 的智能体在给定每个竞争种群（competitor populations）的状态 x^c 下的期望 payoff。第 k 个种群在给定状态 x 的平均适应性为，

$$\bar{f}^k(x) = \sum_i f_i^k(x) x_i^k,$$

相应的 K-population 复制动力学方程为，

$$x_i^k = x_i^k(f_i^k(x) - \bar{f}^k(x)) \qquad \forall k \in \{1,\ldots,K\} \quad \forall i \in s^k.$$

2、Policy Improvement

（1）PSRO

策略空间响应 Oracles(PSRO[20])算法是 Double oracle 算法的推广，meta-game 处理的对象是策略而不是动作。和之前的工作不同的是，Double oracle 可以使用任何 meta solver 选择 meta policy，即 meta solver 和智能体算法是解耦的。也就是说在多智能体环境中，当其他参与者保持不变时，可以使用任何 RL 算法求解 MDP，计算最佳响应。定义一个 meta-game(Π, U, n)，这里$\Pi = (\Pi_1, ..., \Pi_n)$是每个 player 的策略集，$U : \Pi \to \Re^n$是 player 的 payoff，$\sigma_{-i}$是 player i 的对手策略分布。通过设置不同形式的σ_{-i}，我们可以得到不同的方法。比如，independent learning 方法：$\sigma_{-i} = (0,\ldots,0,0,1)$,把所有的 weight，都放在当前 policy 上面；self-play 方法：$\sigma_{-i} = (0,\ldots,0,1,0)$,是只关注对方前一时刻的 policy；fictitious play 方法：$\sigma_{-i} = (\frac{1}{T},\ldots,\frac{1}{T},\frac{1}{T},0)$,看对手历史所有情况；PSRO 方法：$\sigma_{-i} = Nash(\prod^{T-1}, U)$ or $\mathrm{RD}(\prod^{T-1}, U)$。

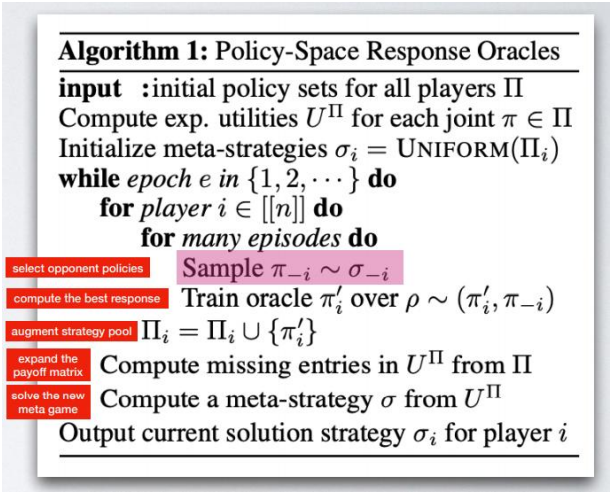


图 76 PSRO 算法

PSRO 算法流程：

- 从对手的 meta-game 里面采样 policy 出来；

- 然后用 RL 算法 train 一个 oracle $\pi_i^{'}$；
- 把$\pi_i^{'}$加入到$\prod_i$的 policy space 里面去，则第 i 个 player 的 policy space 就扩大了；
- 计算下扩大后得到的新的 policy space 的 payoff，得到新的 meta-game；
- 新的 meta-game 又可以算一个新的纳什均衡；

典型军事智能训练平台架构介绍

我们的军事智能训练平台采用开源的 Ray 作为底层的训练框架，此框架包含三个主要部分：

分布式计算框架 Ray core，主要用于底层硬件资源的整合，为上层应用提供简单的原子操作；

强化学习框架 RL Lib，实现了若干强化学习模型算法，支持 tensorflow 和 pytorch，可以很方便地定制属于自己的算法；

超参数调优框架 Tune，实现了若干调优算法，加快模型的收敛效率，提升模型的效果。

一、Ray core

Ray：高性能 AI 计算引擎

1.极简 Python API 接口：在函数或者类定义时加上 ray.remote 的装饰器并做一些微小改变，就能将单机代码变为分布式代码。这意味着不仅可以远程执行纯函数，还可以远程注册一个类（Actor 模型），在其中维护大量 context（成员变量），并远程调用其成员方法来改变这些上下文。

2.高效数据存储和传输：每个节点上通过共享内存（多进程访问无需拷贝）维护了一块局部的对象存储，然后利用专门优化过的 Apache Arrow 格式来进行不同节点间的数据交换。

3.动态图计算模型：这一点得益于前两点，将远程调用返回的 future 句柄传给其他的远程函数或者角色方法，即通过远程函数的嵌套调用构建复杂的计算拓扑，并基于对象存储的发布订阅模式来进行动态触发执

行。

4.全局状态维护：将全局的控制状态（而非数据）利用 Redis 分片来维护，使得其他组件可以方便的进行平滑扩展和错误恢复。当然，每个 redis 分片通过 chain-replica 来避免单点。

5.两层调度架构：分本地调度器和全局调度器；任务请求首先被提交到本地调度器，本地调度器会尽量在本地执行任务，以减少网络开销。在资源约束、数据依赖或者负载状况不符合期望时，会转给全局调度器来进行全局调度。

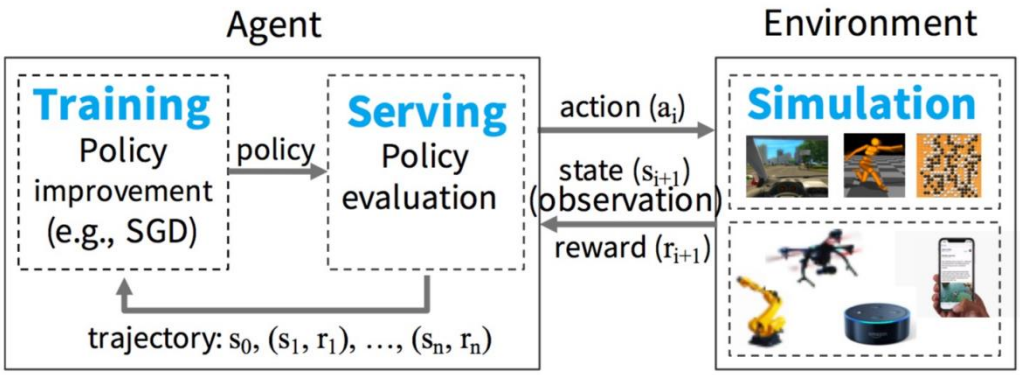


图 77 一个 RL 系统的例子

我们从考虑 RL 系统的基本组件开始，逐渐完善 Ray 的需求。如图 77 所示，在一个 RL 系统的的设定中，智能体（agent）会反复与环境（environment）进行交互。智能体的目标是学习出一种最大化奖励（reward）的策略。策略（policy）本质上是从环境中状态到行为抉择（action）的一种映射。

为了学习策略，智能体通常要进行两步操作：1）策略评估（policy evaluation）和 2）策略优化（policy improvement）。为了评估一个策略，智能体和环境持续进行交互（一般是仿真的环境）以产生轨迹（trajectories）。轨迹是在当前环境和给定策略下产生的一个二元组（状态，奖励值）序列。然后，智能体根据这些轨迹来反馈优化该策略，即，向最大化奖励值的梯度方向更新策略。图 78 展示了智能体用来学习策略一个例子的伪码。该伪码通过调用 rollout(environment, policy) 来评估策

略，进而产生仿真轨迹。train_policy() 接着会用这些轨迹作为输入，调用 policy.update(trajectories) 来优化当前策略。会重复迭代这个过程直到策略收敛。

```
// evaluate policy by interacting with env. (e.g., simulator)
rollout(policy, environment):
    trajectory = []
    state = environment.initial_state()
    while (not environment.has_terminated()):
        action = policy.compute(state) // Serving
        state, reward = environment.step(action) // Simulation
        trajectory.append(state, reward)
    return trajectory

// improve policy iteratively until it converges
train_policy(environment):
    policy = initial_policy()
    while (policy has not converged):
        trajectories = []
        for i from 1 to k:
            // evaluate policy by generating k rollouts
            trajectories.append(rollout(policy, environment))
            // improve policy
            policy = policy.update(trajectories) // Training
    return policy
```

图 78 一段用于学习策略的典型的伪代码

由此看来，针对 RL 应用的计算框架需要高效的支持模型训练（training），在线预测（serving）和平台仿真（simulation）。接下来，我们简要说明一下这些工作负载（workloads）。

模型训练 一般会涉及到在分布式的环境中跑随机梯度下降模型（stochastic gradient descent，SGD）来更新策略。而分布式 SGD 通常依赖于 allreduce 聚合步骤或参数服务器（parameter server）。

在线预测 使用已经训练好的策略并基于当前环境来给出动作决策。预测系统通常要求降低预测延迟，提高决策频次。为了支持扩展，最好能够将负载均摊到多节点上来协同进行预测。

最后，大多数现存的 RL 应用使用仿真（simulations）来对策略进行评估 —— 因为现有的 RL 算法不足以单独依赖从与物理世界的交互中高效的进行取样。这些仿真器在复杂度上跨度极大。也许只需要几毫秒（如模拟国际象棋游戏中的移动），也许会需要几分钟（如为了一个自动

驾驶的车辆模拟真实的环境）。

与模型训练和在线预测可以在不同系统中进行处理的监督学习相比，RL 中所有三种工作负载都被紧耦合在了单个应用中，并且对不同负载间的延迟要求很苛刻。现有的系统中还没有能同时支持三种工作负载的。理论上，可以将多个专用系统组合到一块来提供所有能力，但实际上，子系统间的结果传输的延迟在 RL 下是不可忍受的。

这些现状要求为 RL 开发全新的分布式框架，可以有效地支持训练，预测和仿真。尤其是，这样的框架应具有以下能力：

支持细粒度，异构的计算。RL 计算的运行持续时间往往从数毫秒（做一个简单的动作）到数小时（训练一个复杂的策略）。此外，模型训练通常需要各种异构的硬件支持（如 CPU，GPU 或者 TPU）。

提供灵活的计算模型。RL 应用同时具有有状态和无状态类型的计算。无状态的计算可以在系统中的任何节点进行执行，从而可以方便的进行负载均衡和按需的数据传输。因此，无状态的计算非常适合细粒度的仿真和数据处理，例如从视频或图片中提取特征。相比之下，有状态的计算适合用来实现参数服务器、在支持 GPU 运算的数据上进行重复迭代或者运行不暴露内部状态参数的第三方仿真器。

动态的执行能力。RL 应用中的很多模块要求动态的进行执行，因为他们计算完成的顺序并不总是预先确定（例如：仿真的完成顺序），并且，一个计算的运行结果可以决定是否执行数个将来的计算（如，某个仿真的运行结果将决定我们是否运行更多的仿真）。

除此之外，我们提出了两个额外的要求。首先，为了高效的利用大型集群，框架必须支持每秒钟数百万次的任务调度。其次，框架不是为了支持从头开始实现深度神经网络或者复杂的仿真器，而是必须和现有的仿真器（OpenAI gym 等）和深度学习框架（如 TensorFlow，MXNet，Caffe， PyTorch）无缝集成。

语言和计算模型

Ray 实现了动态任务图计算模型，即，Ray 将应用建模为一个在运行过程中动态生成依赖的任务图。在此模型之上，Ray 提供了角色模型（Actor）和并行任务模型（task-parallel）的编程范式。Ray 对混合计算范式的支持使其有别于与像 CIEL 一样只提供并行任务抽象和像 Orleans 或 Akka 一样只提供角色模型抽象的系统。

编程模型

任务模型（Tasks）。一个任务 * 表示一个在无状态工作进程执行的远程函数（remote function）。当一个远程函数被调用的时候，表示任务结果的 *future 会立即被返回（也就是说所有的远程函数调用都是异步的，调用后会立即返回一个任务句柄）。可以将 Futures 传给 ray.get() 以阻塞的方式获取结果，也可以将 Futures 作为参数传给其他远程函数，以非阻塞、事件触发的方式进行执行（后者是构造动态拓扑图的精髓）。Futures 的这两个特性让用户在构造并行任务的同时指定其依赖关系。

Name	Description
futures = f.remote(*args*)	Execute function *f* remotely. f.remote() can take objects or futures as inputs and returns one or more futures. This is non-blocking.
objects = ray.get(*futures*)	Return the values associated with one or more futures. This is blocking.
ready futures = ray.wait(*futures, k, timeout*)	Return the futures whose corresponding tasks have completed as soon as either *k* have completed or the timeout expires.
actor = Class.remote(*args*) *futures* = *actor*.method.remote(*args*)	Instantiate class *Class* as a remote actor, and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking.

图 79 Ray API

角色模型（Actors）。一个角色对象代表一个有状态的计算过程。每个角色对象暴露了一组可以被远程调用，并且按调用顺序依次执行的成员方法（即在同一个角色对象内是串行执行的，以保证角色状态正确的进行更新）。一个角色方法的执行过程和普通任务一样，也会在远端（每个角色对象会对应一个远端进程）执行并且立即返回一个 future；但不

同的是，角色方法会运行在一个有状态（stateful）的工作进程上。一个角色对象的句柄（handle）可以传递给其他角色对象或者远程任务，从而使他们能够在该角色对象上调用这些成员函数。

Tasks	Actors
细粒度的负载均衡	粗粒度的负载均衡
支持对象的局部性（对象存储 cache）	比较差的局部性支持
微小更新开销很高	微小更新开销不大
高效的错误处理	检查点（checkpoint）恢复带来较高开销

图 80 任务模型 vs. 角色模型的对比

图 80 比较了任务模型和角色模型在不同维度上的优劣。任务模型利用集群节点的负载信息和依赖数据的位置信息来实现细粒度的负载均衡，即每个任务可以被调度到存储了其所需参数对象的空闲节点上；并且不需要过多的额外开销，因为不需要设置检查点和进行中间状态的恢复。与之相比，角色模型提供了极高效的细粒度的更新支持，因为这些更新作用在内部状态（即角色成员变量所维护的上下文信息）而非外部对象（比如远程对象，需要先同步到本地）。后者通常来说需要进行序列化和反序列化（还需要进行网络传输，因此往往很费时间）。例如，角色模型可以用来实现参数服务器（parameter servers）和基于 GPU 的迭代式计算（如训练）。此外，角色模型可以用来包裹第三方仿真器（simulators）或者其他难以序列化的对象（比如某些模型）。

为了满足异构性和可扩展性，ray 从三个方面增强了 API 的设计。首先，为了处理长短不一的并发任务，我们引入了 ray.wait()，它可以等待前 k 个结果满足了就返回；而不是像 ray.get() 一样，必须等待所有结果都满足后才返回。其次，为了处理对不同资源纬度（resource-heterogeneous）需求的任务，ray 让用户可以指定所需资源用量（例如装饰器：ray.remote(gpu_nums=1)），从而让调度系统可以高效的管理资源（即提供一种交互手段，让调度系统在调度任务时相对不那么盲目）。

最后，为了提灵活性，ray 允许构造嵌套远程函数（nested remote functions），意味着在一个远程函数内可以调用另一个远程函数。这对于获得高扩展性是至关重要的，因为它允许多个进程以分布式的方式相互调用（这一点是很强大的，通过合理设计函数，可以使得可以并行部分都变成远程函数，从而提高并行性）。

计算模型

Ray 采用的动态图计算模型，在该模型中，当输入可用（即任务依赖的所有输入对象都被同步到了任务所在节点上）时，远程函数和角色方法会自动被触发执行。在这一小节，会详细描述如何从一个用户程序（图 81）来构建计算图（图 82）。该程序使用了图 79 的 API 实现了图 78 的伪码。

```python
@ray.remote
def create_policy():
# Initialize the policy randomly. return policy

@ray.remote(num_gpus=1)
class Simulator(object):
  def __init__(self):
  # Initialize the environment. self.env = Environment()
    def rollout(self, policy, num_steps):
      observations = []
      observation = self.env.current_state()
      for _ in range(num_steps):
        action = policy(observation)
        observation = self.env.step(action)
        observations.append(observation)
      return observations

@ray.remote(num_gpus=2)
def update_policy(policy, *rollouts):
  # Update the policy.
  return policy

@ray.remote
def train_policy():
  # Create a policy.
  policy_id = create_policy.remote()
  # Create 10 actors.
  simulators = [Simulator.remote() for _ in range(10)] # Do 100 steps of training.
  for _ in range(100):
      # Perform one rollout on each actor.
      rollout_ids = [s.rollout.remote(policy_id)
                     for s in simulators]
      # Update the policy with the rollouts.
      policy_id =
          update_policy.remote(policy_id, *rollout_ids)
   return ray.get(policy_id)
```

图 81

在 Ray 中实现图 78 逻辑的代码，注意装饰器 @ray.remote 会将被注解的方法或类声明为远程函数或者角色对象。调用远程函数或者角色方法后会立即返回一个 future 句柄，该句柄可以被传递给随后的远程函数或者角色方法，以此来表达数据间的依赖关系。每个角色对象包含一个环境对象 self.env，这个环境状态为所有角色方法所共享。

在不考虑角色对象的情况下，在一个计算图中有两种类型的点：数据对象（data objects）和远程函数调用（或者说任务）。同样，也有两种类型的边：数据边（data edges）和控制边（control edges）。数据边表达了数据对象任务间的依赖关系。更确切来说，如果数据对象 D 是任务 T 的输出，我们就会增加一条从 T 到 D 的边。类似的，如果 D 是 任务 T 的输入，我们就会增加一条 D 到 T 的边。控制边表达了由于远程函

数嵌套调用所造成的计算依赖关系，即，如果任务 T1 调用任务 T2*，我们就会增加一条 *T1 到 T2 的控制边。

在计算图中，角色方法调用也被表示成了节点。除了一个关键不同点外，他们和任务调用间的依赖关系基本一样。为了表达同一个角色对象上的连续方法调用所形成的状态依赖关系，我们向计算图添加第三种类型的边：在同一个角色对象上，如果角色方法 Mj 紧接着 Mi 被调用，我们就会添加一条 Mi 到 Mj 的状态边（即 Mi 调用后会改变角色对象中的某些状态，或者说成员变量；然后这些变化后的成员变量会作为 Mj 调用的隐式输入；由此，Mi 到 Mj 间形成了某种隐式依赖关系）。这样一来，作用在同一角色对象上的所有方法调用会形成一条由状态边串起来的调用链（chain，见图 82）。这条调用链表达了同一角色对象上方法被调用的前后相继的依赖关系。

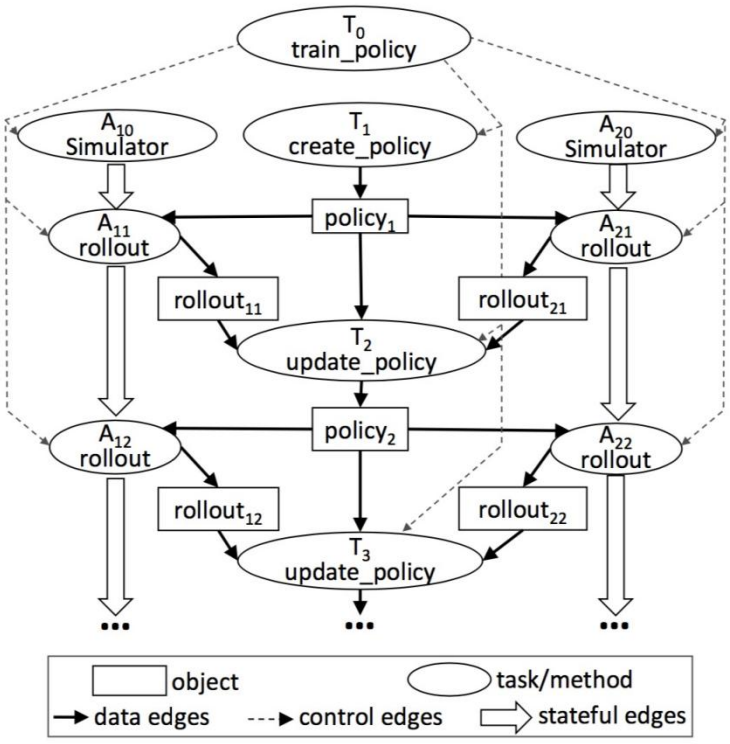


图 82

该图与图 81 train_policy.remote() 调用相对应。远程函数调用和角色方法调用对应图中的任务（tasks）。该图中显示了两个角色对象 A10

和 A20，每个角色对象的方法调用（被标记为 A1i 和 A2i 的两个任务）之间都有状态边（stateful edge）连接，表示这些调用间共享可变的角色状态。从 train_policy 到被其调用的任务间有控制边连接。为了并行地训练多种策略，我们可以调用 train_policy.remote() 多次。

状态边让我们将角色对象嵌入到无状态的任务图中，因为他们表达出了共享状态、前后相继的两个角色方法调用之间的隐式数据依赖关系。状态边的添加还可以让我们维护谱系图（lineage），如其他数据流系统一样，我们也会跟踪数据的谱系关系以在必要的时候进行数据的重建。通过显式的将状态边引入数据谱系图中，我们可以方便的对数据进行重建，不管这些数据是远程函数产生的还是角色方法产生的。

架构

Ray 的架构组成包括两部分：

实现 API 的应用层，现在包括 Python 和 Java 分别实现的版本。

提供高扩展性和容错的系统层，用 C++ 写的，以 CPython 的形式嵌入包中。

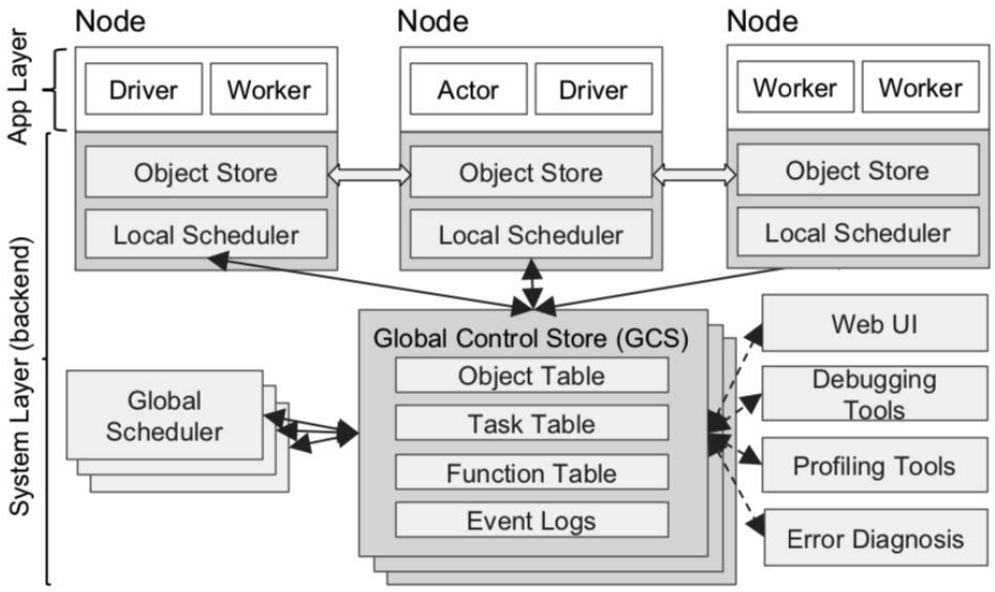


图 83

Ray 的架构包括两部分：系统层和应用层。前者实现了 API 和计算模型，后者实现了任务调度和数据管理，以满足性能要求和容错需求。

应用层

应用层包括三种类型的进程：

驱动进程（Driver）： 用来执行用户程序。

工作进程（Worker）：用来执行 Driver 或者其他 Worker 指派的任务 (remote functions，就是用户代码中装饰了 @ray.remote 的那些函数) 的无状态进程。工作进程在节点启动时被自动启动，一般来说会在每个物理机上启动与 CPU 同样数量的 Worker（这里还有些问题：如果节点是容器的话，获取的仍然是其所在物理机的 CPU 数）。当一个远程函数被声明时，会被注册到全局，并推送到所有 Worker。每个 Worker 顺序的执行任务，并且不维护本地状态。

角色进程（Actor）：用来执行角色方法的有状态进程。与 Worker 被自动的启动不同，每个 Actor 会根据需求（即被调用时）被工作进程或者驱动进程显示启动。和 Worker 一样，Actor 也会顺序的执行任务，不同的是，下一个任务的执行依赖于前一个任务生成或改变的状态 (即 Actor 的成员变量)。

系统层

系统层包括三个主要组件：全局控制存储 (GCS，global control store)，分布式调度器（distributed scheduler）和分布式对象存储 (distributed object store)。所有组件都可以进行水平扩展并且支持容错。

全局控制存储（GCS）

全局状态存储维护着系统全局的控制状态信息，是我们系统独创的一个部件。其核心是一个可以进行发布订阅的键值对存储。我们通过分片 (sharding) 来应对扩展，每片存储通过链式副本（将所有数据副本组

织成链表，来保证强一致性）来提供容错。提出和设计这样一个 GCS 的动机在于使系统能够每秒进行数百万次的任务创建和调度，并且延迟较低，容错方便。

对于节点故障的容错需要一个能够记录谱系信息 (lineage information) 的方案。现有的基于谱系的解决方法侧重粗粒度（比如 Spark 的 rdd）的并行，因此可以只利用单个节点（如 Master or Driver）存储谱系信息，而不影响性能。然而，这种设计并不适合像仿真（simulation）一样的细粒度、动态的作业类型 (workload)。因此我们将谱系信息的存储与系统其它模块解耦，使之可以独立地动态扩容。

保持低延迟意味着要尽可能降低任务调度的开销。具体来说，一个调度过程包括选择节点，分派任务，拉取远端依赖对象等等。很多现有的信息流系统，将其所有对象的位置、大小等信息集中存储在调度器上，使得上述调度过程耦合在一块。当调度器不是瓶颈的时候，这是一个很简单自然的设计。然而，考虑到 Ray 要处理的数据量级和数据粒度，需要将中心调度器从关键路径中移出（否则如果所有调度都要全局调度器经手处理，它肯定会成为瓶颈）。对于像 allreduce 这样的（传输频繁，对延迟敏感）分布式训练很重要的原语来说，每个对象传输时都经手调度器的开销是不可容忍的。 因此，我们将对象的元数据存储在 GCS 中而不是中央调度器里，从而将任务分派与任务调度完全解耦。

总的来说，GCS 极大地简化了 Ray 的整体设计，因为它将所有状态揽下，从而使得系统中其他部分都变成无状态。这不仅使得对容错支持简化了很多（即，每个故障节点恢复时只需要从 GCS 中读取谱系信息就行），也使得分布式的对象存储和调度器可以进行独立的扩展（因为所有组件可以通过 GCS 来获取必要的信息）。还有一个额外的好处，就是可以更方便的开发调试、监控和可视化工具。

自下而上的分布式调度系统（Bottom-up Distributed Scheduler）

如前面提到的，Ray 需要支持每秒数百万次任务调度，这些任务可

能只持续短短数毫秒。大部分已知的调度策略都不满足这些需求。常见的集群计算框架，如 Spark， CIEL， Dryad 都实现了一个中心的调度器。这些调度器具有很好的局部性（局部性原理）的特点，但是往往会有数十毫秒的延迟。像 work stealing，Sparrow 和 Canary 一样的的分布式调度器的确能做到高并发，但是往往不考虑数据的局部性特点，或者假设任务 (tasks) 属于不同的作业 (job)，或者假设计算拓扑是提前知道的。

为了满足上述需求，Ray 设计了一个两层调度架构，包括一个全局调度器（global scheduler）和每个节点上的本地调度器（local scheduler）。为了避免全局调度器过载，每个节点 (node）上创建的任务会被先提交到本地调度器。本地调度器总是先尝试将任务在本地执行，除非其所在机器过载 (比如任务队列超过了预定义的阈值) 或者不能满足任务任务的资源需求 (比如，缺少 GPU)。如果本地调度器发现不能在本地执行某个任务，会将其转发给全局调度器。由于调度系统都倾向于首先在本地调度任务（即在调度结构层级中的叶子节点），我们将其称为自下而上的调度系统（可以看出，本地调度器只是根据本节点的局部负载信息进行调度，而全局调度器会根据全局负载来分派任务；当然前提是资源约束首先得被满足）。

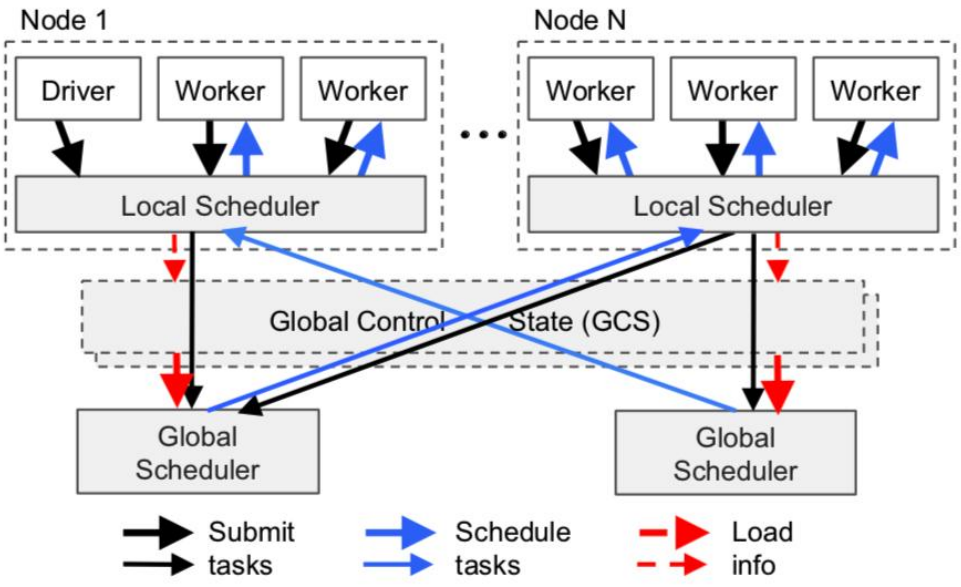


图 84

这是调度系统示意图，任务自下而上被提交：任务首先被驱动进程（Drivers）或者工作进程 (Workers) 提交到本地调度器，然后在需要的时候会由本地调度器转给全局调度器进行处理。图中箭头的粗细程度代表其请求的繁忙程度。

全局调度器根据每个节点的负载状况和资源请求约束来决定调度策略。细化一下就是，全局调度器首先确定所有满足任务资源要求的节点，然后在其中选择具有最小预估排队时间 (estimated waiting time) 的一个，将任务调度过去。在给定的节点上，预估排队时间是下述两项时间的和：1）任务在节点上的排队时间 (任务队列长度乘上平均执行时间)； 2）任务依赖的远程对象的预估传输时间 (所有远程输入的大小除以平均带宽)。全局调度器通过心跳获取到每个节点的任务排队情况和可用资源信息，从 GCS 中得到任务所有输入的位置和大小。然后，全局调度器通过移动指数平均(exponential averaging) 的方法来计算任务平均执行时间和平均传输带宽。如果全局调度器成为了系统瓶颈，我们可以实例化更多的副本来分摊流量，它们通过 GCS 来共享全局状态信息。如此一来，我们的调度架构具有极高可扩展性。

任务生命周期

在实现的时候，每个任务具有以下几种状态。任意时刻，任务都会处在这几种状态之一。

可放置 (Placeable)：任务已经准备好被调度到（本地或者远程）节点上，具体如何调度，前一段已经说明。注意该状态不表示放置位置已经最终确定，还可能被再一次被从某处调度出去。

等待角色创建 (WaitActorCreation)：一个角色方法（task）等待其所在角色实例化完毕。一旦角色被创建，该任务会被转给运行该角色的远端机器进行处理。

等待中 (Waiting)：等待该任务参数需求被满足，即，等待所有远端参数对象传送到本地对象存储中。

准备好 (Ready)：任务准备好了被运行，也就说所有所需参数已经在本地对象存储中就位了。

运行中 (Running)：任务已经被分派，并且正在本地工作进程 (worker) 或者角色进程 (actor) 中运行。

被阻塞 (Blocked)：当前任务由于其依赖的数据不可用而被阻塞住。如，嵌套调用时，该任务启动了另外的远程任务并且等待其完成，以取得结果。

不可行 (infeasible)：任务的资源要求在任何一台机器上都得不到满足。

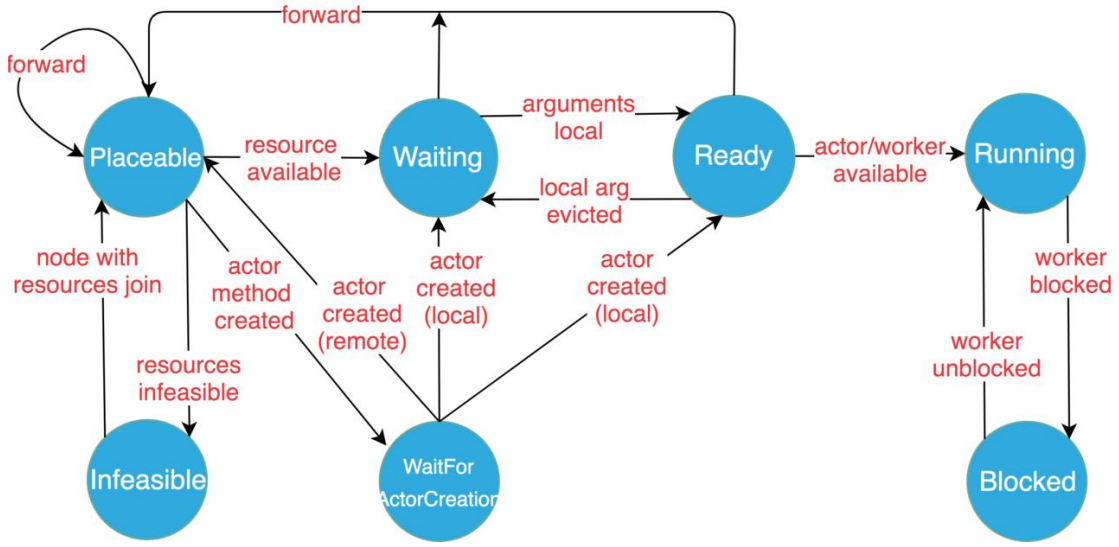


图 85 任务生命周期

基于内存的分布式对象存储

为了降低任务的延迟，我们实现了一个基于内存的分布式存储系统以存储每个任务（无状态的计算过程）的输入和输出。在每个节点上，我们以共享内存 (shared memory) 的方式实现了对象存储。这使得同一节点上的不同任务以零拷贝的代价进行数据共享。至于数据格式，我们选择了 Apache Arrow。

如果一个任务的输入（即函数的参数对象）不在本地，在该任务执行之前，输入会被拷贝到本地的对象存储中。同时，任务执行完毕后，

会将输出也写到本地得对象存储中。对象拷贝消除了热数据所造成的潜在的瓶颈，并且通过将任务的数据读写都限制在本地内存中以缩短执行时间。这些做法增加了计算密集型工作任务的吞吐量，而很多 AI 应用都是计算密集型的。为了降低延迟，我们将用到的对象全部放在内存中，只有在内存不够的时候才通过 LRU 算法将一些对象挤出内存（从 API 可以看出，每个节点的内存上限可以在启动节点时通过参数指定。此外用 LRU 作为垃圾回收算法还是有点粗暴，如果不同类型的任务负载跑在同一个 ray 集群上，可能导致资源的互相争抢，从而有大量的资源换出然后重建，从而严重影响效率）。

和现有的计算框架的集群 (如 Spark， Dryad) 一样，对象存储只接受不可变数据 (immutable data)。这种设计避免了对复杂的一致性协议的需求（因为对象数据从来不需要进行更新），并且简化了数据的容错支持。当有节点出现故障时，Ray 通过重新执行对象谱系图来恢复任意所需对象（也就是说不用整个恢复该宕机节点所有状态，只需要按需恢复后面计算所需数据，用不到的数据丢了就丢了吧）。在工作开始之前，存放在 GCS 的谱系信息追踪了所有无状态的任务和有状态的角色；我们利用前者对丢失对象进行重建（结合上一段，如果一个任务有大量的迭代，并且都是远程执行，会造成大量的中间结果对象，将内存挤爆，从而使得较少使用但是稍后可能使用的全局变量挤出内存，所以 LRU 有点粗暴，听说现在在酝酿基于引用计数的 GC）。

为了简化实现，Ray 的对象存储不支持分布式的对象。也就是说，每个对象必须能够在单节点内存下，并且只存在于单节点中。对于大矩阵、树状结构等大对象，可以在应用层来拆分处理，比如说实现为一个集合。

图 86 通过一个简单的 a 加 b （a，b 可以是标量，向量或者矩阵）然后返回 c 的例子展示了 Ray 端到端的工作流。远程函数 add() 在初始化 (ray.init) 的时候，会自动地被注册到 GCS 中，进而分发到集群中的

每个工作进程。（图 86a 的第零步）

图 86a 展示了当一个驱动进程（driver）调用 add.remote(a, b)，并且 a, b 分别存在节点 N1 和 N2 上时，Ray 的每一步操作。驱动进程将任务 add(a, b) 提交到本地调度器（步骤 1），然后该任务请求被转到全局调度器（步骤 2）（如前所述，如果本地任务排队队列没有超过设定阈值，该任务也可以在本地进行执行）。接着，全局调度器开始在 GCS 中查找 add(a, b) 请求中参数 a, b 的位置（步骤 3），从而决定将该任务调度到节点 N2 上（因为 N2 上有其中一个参数 b）（步骤 4）。N2 节点上的本地调度器收到请求后（发现满足本地调度策略的条件，如满足资源约束，排队队列也没超过阈值，就会在本地开始执行该任务 *），会检查本地对象存储中是否存在任务 add(a, b) 的所有输入参数（步骤 5）。由于本地对象存储中没有对象 a，工作进程会在 GCS 中查找 a 的位置（步骤 6）。 这时候发现 a 存储在 *N1 中，于是将其同步到本地的对象存储中（步骤 7）。由于任务 add() 所有的输入参数对象都存在了本地存储中，本地调度器将在本地工作进程中执行 add() （步骤 8），并通过共享存储访问输入参数（步骤 9）。

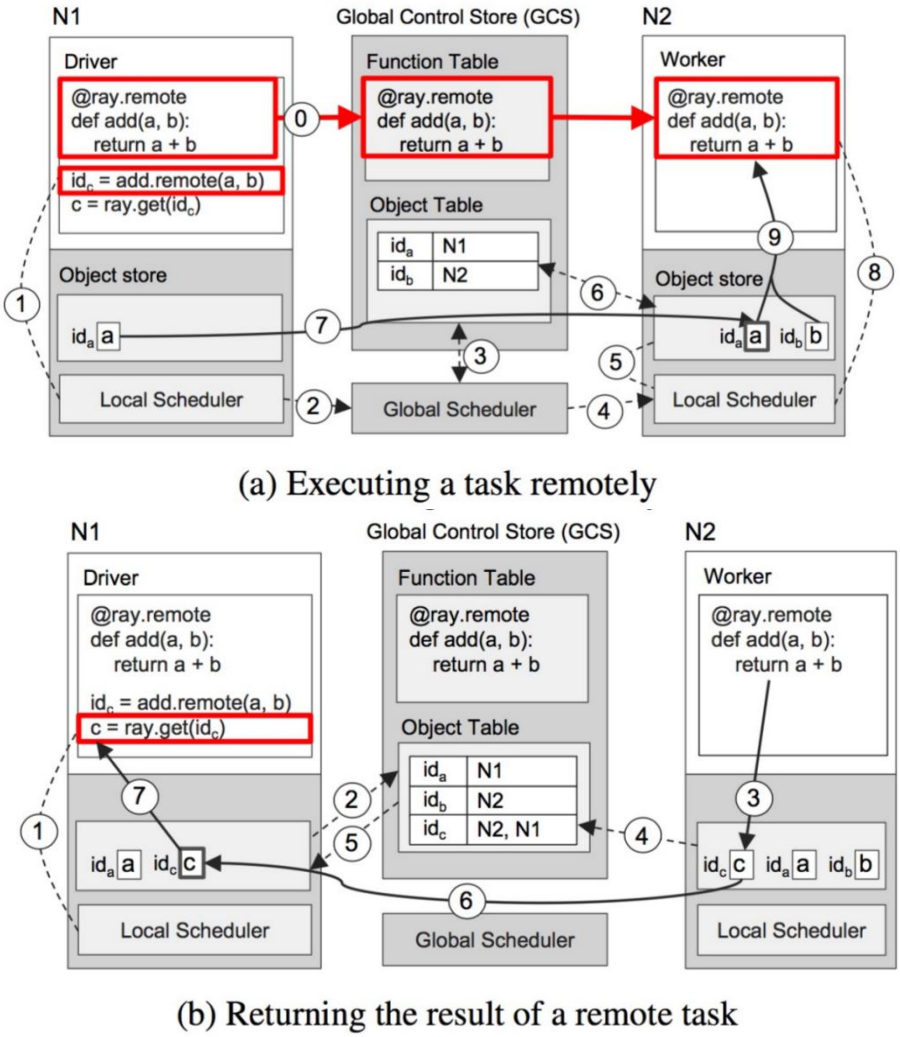


图 86

图 86b 展现了在 N1 上执行 ray.get() 和在 N2 上执行 add() 后所触发的逐步的操作。一旦 ray.get(id) 被调用，N1 上的用户驱动进程会在本地对象存储中查看该 id （即由远程调用 add() 返回的 future 值，所有 object id 是全局唯一的，GCS 可以保证这一点）对应的对象 c 是否存在（步骤 1）。由于本地对象存储中没有 c , 驱动进程会去 GCS 中查找 c 的位置。在此时，发现 GCS 中并没有 c 的存在，因为 c 根本还没有被创建出来。 于是，N1 的对象存储向 GCS 中的对象表（Object Table）注册了一个回调函数，以监听 c 对象被创建事件（步骤 2）。与此同时，在节点 N2 上，add () 任务执行完毕，将结果 c 存到其本地对象存储中（步骤

3），同时也将 c 的位置信息添加到 GCS 的对象存储表中（步骤 4）。GCS 监测到 c 的创建，会去触发之前 N1 的对象存储注册的回调函数（步骤 5）。接下来，N1 的对象存储将 c 从 N2 中同步过去（步骤 6），从而结束该任务。

尽管这个例子中涉及了大量的 RPC 调用，但对于大部分情况来说，RPC 的数量会小的多，因为大部分任务会在本地被调度执行，而且 GCS 回复的对象信息会被本地调度器和全局调度器缓存（但是另一方面，执行了大量远程任务之后，本地对象存储很容易被撑爆）。

名词对照

workloads：工作负载，即描述任务需要做的工作。

GCS： Global Control Store，全局控制信息存储。

Object Table：存在于 GCS 中的对象表，记录了所有对象的位置等信息（objectId -> location）。

Object Store：本地对象存储，在实现中叫 Plasma，即存储任务所需对象的实例。

Lineage：血统信息，谱系信息；即计算时的数据变换前后的相继关系图。

Node：节点；Ray 集群中的每个物理节点。

Driver、Worker：驱动进程和工作进程，物理表现形式都是 Node 上的进程。但前者是用户侧使用 ray.init 时候生成的，随着 ray.shutdown 会进行销毁。后者是 ray 在启动的时在每个节点启动的无状态的驻留工作进程，一般和物理机 CPU 数量相同。

Actor：角色对象，语言层面，就是一个类；物理层面，表现为某个节点上的一个角色进程，维护了该角色对象内的所有上下文（角色成员变量）。

Actor method：角色方法，语言层面，就是类的成员方法；其所有输入包括显式的函数参数和隐式的成员变量。

Remote function：远程函数，即通过 @ray.remote 注册到系统的函数。在其被调度时，称为一个任务 (Task)。

Job，Task：文中用到了不少 Job 和 Task 的概念，但是这两个概念在 CS 中其实定义比较模糊，不如进程和线程一般明确。Task 在本论文是对一个远程函数 (remote action) 或者一个 actor 的远程方法 (remote method) 的封装。而 Job 在当前的实现中并不存在，只是一个逻辑上的概念，其含义为运行一次用户侧代码所所涉及到的所有生成的 Task 以及产生的状态的集合。

Scheduler：paper 中统一用的 scheduler，但是有的是指部分 (local scheduler 和 global scheduler)，这时我翻译为调度器，有时候是指 Ray 中所有调度器构成的整体。

二、Rllib

RL-lib 是一个开源的强化学习库，为不同的算法应用提供统一的 API，具有很强的扩展性。RL-lib 兼容 TensorFlow、TensorFlow Eager 和 PyTorch，极大方便了掌握不同 Tensor 编程框架的用户编写和训练自己的算法模型。

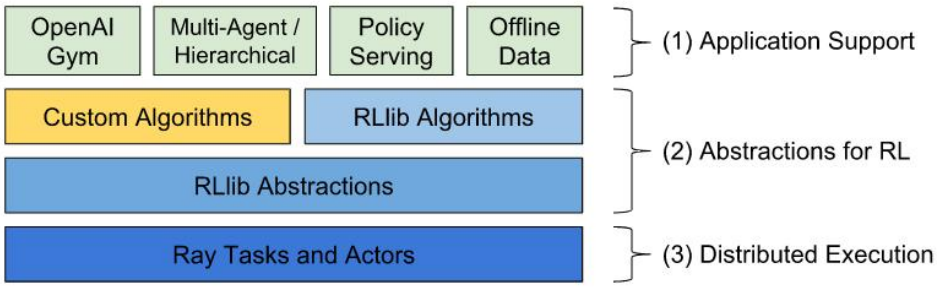


图 87 Rllib 框架

（一）底层实现

1、Iterator和Flow

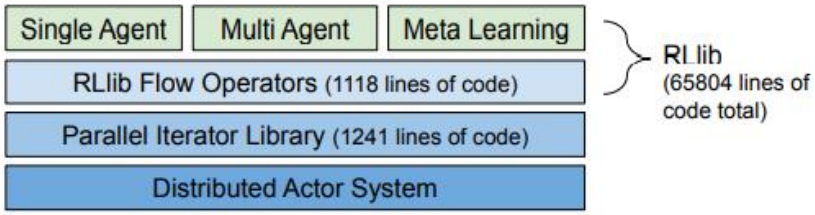


图 88 Rllib 底层实现

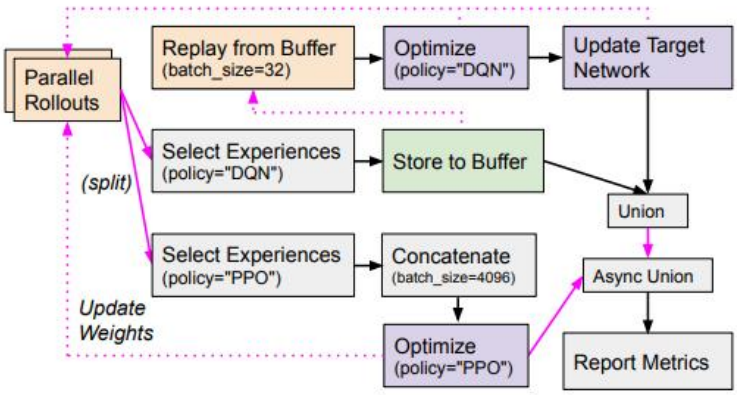


图 89 ppo+dqn execution graph

```
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[Rollout], Iter[Rollout]
r1, r2 = ParallelRollouts(workers).split()
# type: Iter[TrainStats], Iter[TrainStats]
ppo_op = ppo_plan(
    Select(r1, policy="PPO"), workers)
dqn_op = dqn_plan(
    Select(r2, policy="DQN"), workers)
# type: Iter[Metrics]
return ReportMetrics(
    Union(ppo_op, dqn_op), workers)
```

图 90 ppo+dqn execution graph

```
create(Seq[SourceActor[T]]) -> ParIter[T]
send_msg(dest: Actor, msg: Any) -> Reply
```

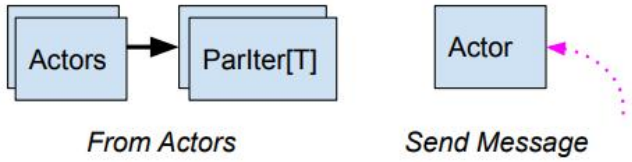


图 91 创建并行迭代器与消息发送

```
gather_async(ParIter[T],
        num_async: Int) -> Iter[T]
gather_sync(ParIter[T]) -> Iter[List[T]]
next(Iter[T]) -> T
```

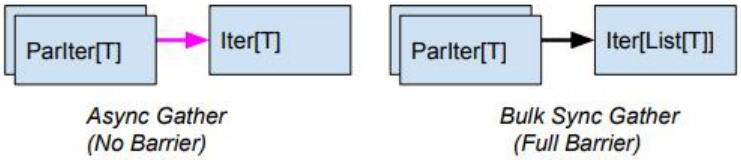


图 92 并行收集数据

```
foreach(ParIter[T], T => U) -> ParIter[U]
foreach(Iter[T], T => U) -> Iter[U]
```

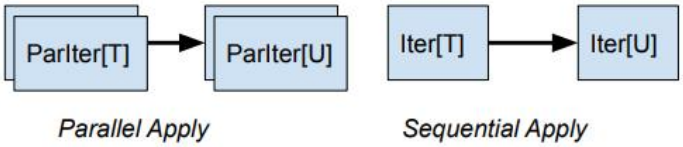


图 93 并行/串行 apply transformation

```
split(Iter[T]) -> (Iter[T], Iter[T])
union(List[Iter[T]],
  weights: List[float]) -> Iter[T]
union_async(List[Iter[T]]): Iter[T]
```

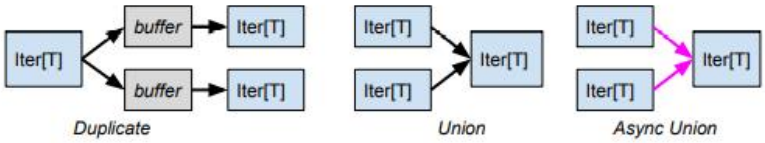


图 94 迭代器 split/union

（二）Rllib Abstraction

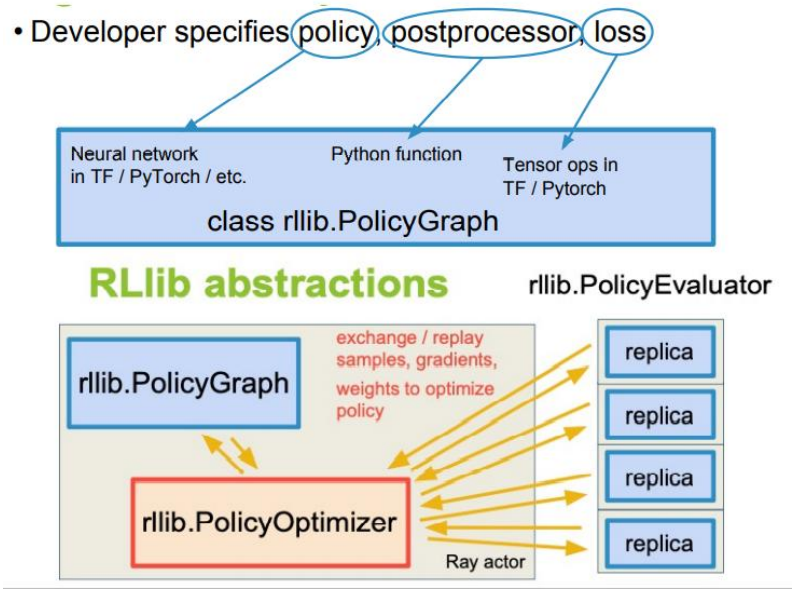


图 95 rllib 抽象

要利用 RLlib 进行分布式执行算法，必须声明它们的策略π、经验后处理器ρ和目标函数 L，这些可以在任何深度学习框架中指定，包括 TensorFlow 和 PyTorch。RLlib 提供了策略评估器和策略优化器，用于实现分布式策略评估和策略训练。

策略计算图的定义

此处介绍 RLlib 的抽象模式。用户指定一个策略模型π，将当前观测值o_t和（可选）RNN 的隐藏状态h_t映射到一个动作a_t和下一个 RNN 状态h_{t+1}。任何用户定义的值y_t^i（例如，值预测、TD 误差）也可以返回：

$$\pi_\theta(o_t, h_t) \Rightarrow (a_t, h_{t+1}, y_t^1, \cdots y_t^N)$$

大多数算法也会指定一个轨迹后处理函数ρ，它可以将一批数据$X_{t,K}$进行变换，其中 K 是一个时刻 t 的元组$\{(o_t, h_t, a_t, h_{t+1}, y_t^1, \cdots y_t^N, r_t, o_{t+1})\}$。此处$r_t$和$o_{t+1}$表示 t 时刻采取行动$a_t$ 之后所获得的奖励和新的观测状态。后处理函数使用的典型例子有优势函数估计（GAE）和事后经验回放（HER）。为了支持多智能体环境，使用该函数处理不同的 P 个智能体的数据也是可以的：

$$\rho_\theta(X_{t,K}, X_{t,K}^1, \cdots, X_{t,K}^P) \Rightarrow X_{post}$$

基于梯度的算法会定义一个目标函数 L，使用梯度下降法来改进策略和其网络：

$$L(\theta; X) \Rightarrow loss$$

最后，用户还可以指定任意数量的在训练过程中根据需要调用的辅助函数u_i，比如返回训练统计数据 s，更新目标网络，或者调整学习率控制器：$u^1, \cdots, u^M(\theta) \Rightarrow (s, \theta_{update})$

在 RLlib 实现中，这些算法函数在策略图类中定义，方法如下：

```
abstract class rllib.PolicyGraph:
    def act(self, obs, h): action, h, y*
    def postprocess(self, batch, b*): batch
    def gradients(self, batch): grads
    def get_weights
    def set_weights
    def u*(self, args*)
```

图 96

策略评估器

为了收集与环境交互的数据，RLlib 提供了一个叫做 PolicyEvaluator 的类，封装了一个策略图和环境，并且支持 sample() 获取其中随机采样的数据。策略评价器实例可以作为 Ray actor，并在计算集群中复制以实现并行化。举个例子，可以考虑一个最小的 TensorFlow 策略梯度方法实现，它扩展了 rllib.TFPolicyGraph 模板：

```
class PolicyGradient(TFPolicyGraph):
    def __init__(self, obs_space, act_space):
        self.obs, self.advantages = ...
        pi = FullyConnectedNetwork(self.obs)
        dist = rllib.action_dist(act_space, pi)
        self.act = dist.sample()
        self.loss = -tf.reduce_mean(
            dist.logp(self.act) * self.advantages)
    def postprocess(self, batch):
        return rllib.compute_advantages(batch)
```

图 97

根据该策略图定义，用户可以创建多个策略评估器副本 ev，并在每个副本上调用 ev.sample.remote()，从环境中并行收集经验。RLlib 支持

OpenAIGym、用户定义的环境，也支持批处理的模拟器（如 ELF）：

```
evaluators = [rllib.PolicyEvaluator.remote(
    env=SomeEnv, graph=PolicyGradient) for _ in range(10)]
print(ray.get([ev.sample.remote() for ev in evaluators]))
```

图 98

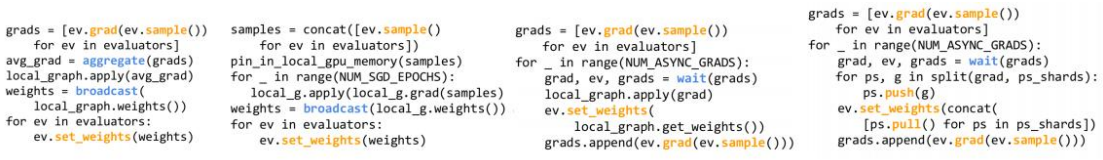
策略优化器

RLlib 将算法的实现分为与算法相关的策略计算图和与算法无关的策略优化器两个部分。策略优化器负责分布式采样、参数更新和管理重放缓冲区等性能关键任务。为了分布式计算，优化器在一组策略评估器副本上运行。用户可以选择一个策略优化器，并通过引用现有的评价器来创建它。异步优化器使用评价器行为体在多个 CPU 上并行计算梯度。每个 optimizer.step() 都会运行一轮远程任务来改进模型。在两次该函数被调用之间，还可以直接查询策略图副本，如打印出训练统计数据：

```
optimizer = rllib.AsyncPolicyOptimizer(
    graph=PolicyGradient, workers=evaluators)
while True:
    optimizer.step()
    print(optimizer.foreach_policy(
        lambda p: p.get_train_stats()))
```

图 99

策略优化器将众所周知的梯度下降优化器扩展到强化学习领域。一个典型的梯度下降化器实现了$step(L(\theta), X, \theta) \Rightarrow \theta_{opt}$。RLlib 的策略优化器在此基础上更进一步，在本地策略图 G 和一组远程评估器副本上操作，即，$step(G, ev_1, \cdots, ev_n, \theta) \Rightarrow \theta_{opt}$，将强化学习的采样阶段作也为优化的一部分（即在策略评估器上调用 sample() 函数以产生新的仿真数据）。将策略优化器如此抽象具有以下优点：通过将执行策略与策略优化函数定义分开，各种不同的优化器可以被替换进来，以利用不同的硬件和算法特性，却不需要改变算法的其余部分。策略图类封装了与深度学习框架的交互，使得用户可以避免将分布式系统代码与数值计算混合在一起，并使优化器的实现能够被在不同的深度学习框架中改进和重用。



(a) 全局规约　(b) 本地多 GPU　(c) 异步计算　(d) 分片参数服务器

图 100

四种 RLlib 策略优化器步骤方法的伪代码。每次调用优化函数时，都在本地策略图和远程评估程序副本阵列上运行。图中用橙色高亮 Ray 的远程执行调用，用蓝色高亮 Ray 的其他调用。apply 是更新权重的简写。此处省略 mini-batch 处理代码和辅助函数。RLlib 中的参数服务器优化器还实现了流模式，此处未给予显示。

（三）PPO+DQN分布式训练案例

（1）讲解 ray/agents

主要讲解 trainer_template 和 trainer。

（2）讲解 ray/execution

主要讲解 rollout_ops、replay_ops 和 train_ops。

（3）讲解 ray/evaluation

主要讲解 rollout_worker 和 sampler 文件。

（4）讲解 ray/policy

主要讲解 policy、torch_policy_template 和 torch_policy。

（5）讲解 ray/model

主要讲解 modelv2、torch_modelv2 和 torch_action_dist。

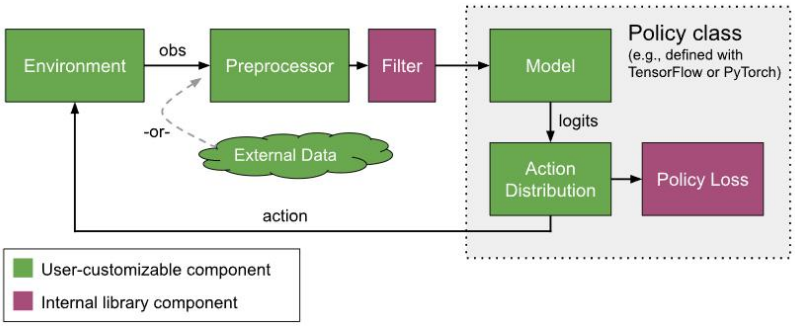


图 101 rllib 环境与算法交互

三、Tune

Tune 是一个超参数优化库，可以用于 PyTorch、TensorFlow、MXnet、keras 等深度学习框架。

在深度学习中，除了可以学习参数外，还存在很多超参数，这些超参数对于网络的性能影响也十分巨大，不同的机器学习任务往往需要不同的超参数，常见的超参数有：

1、 网络结构，包括神经元的连接关系、层数、每层的神经元的数量、激活函数的类型；

2、 优化参数，包括优化方法、学习率、小批量的样本数量等；

3、 正则化系数；

对于超参数的搜索，一般使用的方法：人工搜索，网格搜索，随机搜索，贝叶斯优化。由于目前深度学习的优化方法一般都采取随机梯度下降，因此我们可以通过一组超参数的学习曲线来预估这组超参数配置是否有希望得到比较好的结果。如果一组超参数配置的学习曲线不收敛或者收敛比较差，我们可以应用早期停止（early-stopping）策略来中止当前的训练。

Tune 的核心特征：

1、多计算节点的分布式超参数的搜索；

2、支持多种深度学习框架，例如：pytorch，TensorFlow；

3、结果直接可以用 tensorboard 可视化；

4、可拓展的 SOTA 算法，例如：PBT，HyperBand/ASHA；

5、整合了很多超参数优化库，例如：Ax，HyperOpt，Bayesian Optimization；

PBT

基于种群的优化方法 PBT[21](population based training)，主要用来自适应调节超参数。通常的深度学习，超参数都是凭经验预先设计好的，会花费大量精力且不一定有好的效果，特别是在深度强化学习这种非静态(non-stationary)的环境中，要想得到 SOTA 效果，超参数还应随着环境变化而自适应调整，比如探索率等等。这种基于种群(population)的进化方式，淘汰差的模型，利用(exploit)好的模型并添加随机扰动(explore)进一步优化，最终得到最优的模型。

神经网络的训练受模型结构、数据表征、优化方法等的影响。而每个环节都涉及到很多参数(parameters)和超参数(hyper-parameters)，对这些参数的调节决定了模型的最终效果。通常的做法是人工调节，但这种方式费时费力且很难得到最优解。

两种常用的自动调参的方式是并行搜索(parallel search)和序列优化(sequential optimization)。并行搜索就是同时设置多组参数训练，比如网格搜索(grid search)和随机搜索(random search)。序列优化很少用到并行，而是一次次尝试并优化，比如人工调参(hand tuning)和贝叶斯优化(Bayesian optimization)。并行搜索的缺点在于没有利用相互之间的参数优化信息。而序列优化这种序列化过程显然会耗费大量时间。

还有另一个问题是，对于有些超参数，在训练过程中并不是一直不变的。比如监督训练里的学习率，强化学习中的探索度等等。通常的做法是给一个固定的衰减值，而在强化学习这类问题里还会随不同场景做不同调整。这无疑很难找到一个最优的自动调节方式。

PBT 基于一种很朴素的思想，将并行优化和序列优化相结合。既能并行探索，同时也利用其他更好的参数模型，淘汰掉不好的模型。

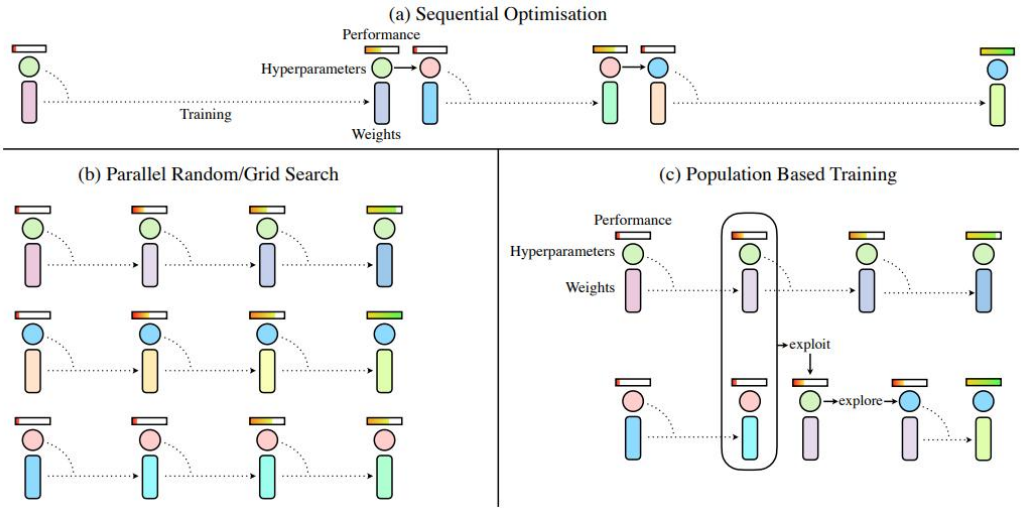


图 102 几种超参数调优范式

如图 102 所示，(a)中的序列优化过程只有一个模型在不断优化，消耗大量时间。(b)中的并行搜索可以节省时间，但是相互之间没有任何交互，不利于信息利用。(c)中的 PBT 算法结合了二者的优点。

首先 PBT 算法随机初始化多个模型，每训练一段时间设置一个检查点(checkpoint)，然后根据其他模型的好坏调整自己的模型。若自己的模型较好，则继续训练。若不好，则替换(exploit)成更好的模型参数，并添加随机扰动(explore)再进行训练。其中 checkpoint 的设置是人为设置每过多少 step 之后进行检查。扰动要么在原超参数或者参数上加噪声，要么重新采样获得。

Algorithm 1 Population Based Training (PBT)

procedure TRAIN($\mathcal{P}$) ▷ initial population $\mathcal{P}$
  for $(\theta, h, p, t) \in \mathcal{P}$ (asynchronously in parallel) **do**
    while not end of training **do**
      $\theta \leftarrow \texttt{step}(\theta|h)$ ▷ one step of optimisation using hyperparameters h
      $p \leftarrow \texttt{eval}(\theta)$ ▷ current model evaluation
      if $\texttt{ready}(p, t, \mathcal{P})$ **then**
        $h', \theta' \leftarrow \texttt{exploit}(h, \theta, p, \mathcal{P})$ ▷ use the rest of population to find better solution
        if $\theta \neq \theta'$ **then**
          $h, \theta \leftarrow \texttt{explore}(h', \theta', \mathcal{P})$ ▷ produce new hyperparameters h
          $p \leftarrow \texttt{eval}(\theta)$ ▷ new model evaluation
        end if
      end if
      update $\mathcal{P}$ with new $(\theta, h, p, t+1)$ ▷ update population
    end while
  end for
  return θ with the highest p in $\mathcal{P}$
end procedure

图 103 PBT 算法

如图 103 首先给定训练的种群（population P），在异步并行的的每一个工作节点中，都有一整套的神经网络，也就对应了一整套的参数θ和超参数 h 以及对该节点的性能评价指标 p 以及模型更新轮次 t，这里有一个容易混淆的概念是迭代执行的轮次（step），模型只有在迭代若干次，也就是进行了 n 个 steps 之后才会根据情况来对模型进行更新，此时 t 才会变化。

在训练结束之前，每一个节点每一次迭代都会进行一次模型参数的更新，即 step(θ |h)，step 操作通常是随机梯度下降即相应的优化变体（使用动量系数 momentum、优化方法 adam 等等）；随后对更新完的模型进行一次评估，即 eval 操作，eval 操作评估的方式 DRL 中可以使用前 10 轮的平均 episodic rewards；接下来判断模型更新的 ready 条件，如执行了若干个迭代（steps），如果满足要求就首先执行一次 exploit 操作，exploit 操作是探索当前所有种群（包含它自身）中最好的模型参数和超参数，如果它自身的参数不是最优的，就把最优的参数和超参数加一点扰动（explore）之后替代掉当前的模型，然后重新对替换后的模型进行评价，知道训练结束，最终，我们从种群中若干个节点里选择评价指标 p 最好的那个模型作为我们最终的训练结果。

无人机突防智能体开发案例解析

本案例用于训练一个能够突防地面坦克防线的无人机智能体。无人机在没有人为干预的情况下，向部署有防空网的坦克排防线飞行，尝试摧毁坦克排。在向坦克排防线飞行路线中有防空盲区，无人机智能体训练需要学会如何规避防空区域，并对坦克排进行打击。

在训练中无人机在智能体的规划下在不断的修改航线，试图规避蓝方的防空区域。通过训练，能看到开始无人机只是无规则的飞行，慢慢学会飞到目标区域，接着攻击并击毁目标。达到学习目的。

一、想定介绍

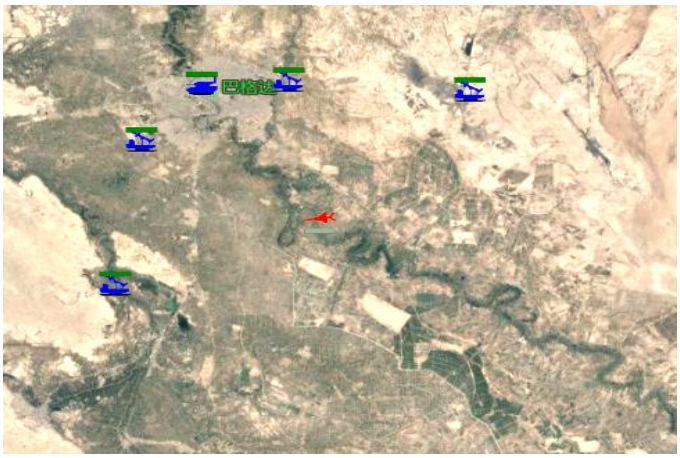


图 104 无人机突防态势图

作战目标

1）红方作战目标：

利用无人机优势，快速穿过敌防空区域，消灭蓝方坦克，阻止敌方装甲对前线士兵的威胁。

2）蓝方作战目标：

使用防空武器击落红方无人机，保护坦克不受无人机威胁，使坦克进入前线，压制敌方。

作战单元

1）红方作战单元

表 1 红方作战单元表

序号	作战单元名称	数量
1	MQ-1C 型“灰鹰”无人机	1
2	AGM-114K 型“地狱火 II”反坦克导弹	4

2）蓝方作战单元

表 2 蓝方作战单元表

序号	作战单元名称	数量
1	地空导弹排(SA-22 Greyhound [Pantsir-S1E])	4
	萨姆-22“灰狗”[57E6]地空导弹	26
2	坦克排(T-72 MBT x 4)	1

二、环境类设计

1、观测（observation）

观测值是当前时刻直升机的经度、纬度和朝向。

```
obs_lt = [0.0 for x in range(0, self.state_space_dim)]
for key in unit_list:
    aircraft_list_dic = self.redside.aircrafts
    unit = aircraft_list_dic.get(key)
    if unit:
        obs_lt[0] = unit.dLongitude
        obs_lt[1] = unit.dLatitude
        obs_lt[2] = unit.fCurrentHeading
return obs_lt
```

2、动作（action）

动作值是直升机的“朝向”，执行动作函数，首先检查是否进入任务区，进入任务区的话，检查是否发现目标，发现目标进行自动开火；没有进入任务的话，就是设置飞机航线。

```
airs = self.redside.aircrafts
for guid in airs:
    aircraft = airs[guid]
    if distance < etc.target_radius:
        # 如果目标距离小于打击距离，且已发现目标，则自动攻击之
        if self._check_is_find_target():
            target_guid = self._get_target_guid()
            target_guid = self._get_contact_target_guid()
            print("%s: 自动攻击目标" % datetime.time())
            aircraft.auto_attack_target(target_guid)
    else:
        # 如果目标距离大于打击距离，则继续机动
        lon, lat = self._deal_point_data(waypoint)
        # print("set waypoint:%s %s" % (lon, lat))
        aircraft.set_waypoint(lon, lat)
```

3、奖励（reward）

首先根据距离获得 distance_reward，然后根据是否进入目标区域，获得 reward。

- 如果不在目标区域直升机被销毁，获得 reward；
- 检查发现目标，获得 reward；
- 击毁目标，获得 reward。

```
reward = 0.0
if action_value is not None:
    # 距离目标越近，奖励值越大
    distance_reward, distance = self._get_distance_reward(action_value)
    reward += distance_reward

    if distance < etc.target_radius:  # 如果进入了一个距离范围
        reward += 10.0
    else:
        if not self._check_aircraft_exist():  # 飞机被打死，则降低奖赏值
            reward += -100.0
    if not self._check_target_exist():  # 目标被打死，则增加奖赏值
        reward += 150.0
return reward
```

三、网络架构

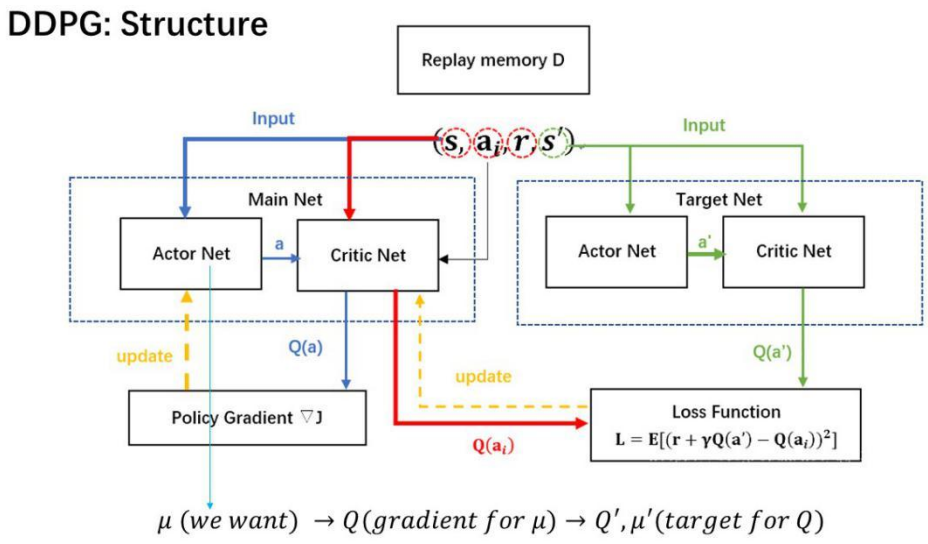


图 105 DDPG 网络架构图

Critic 网络：

```python
class Critic(nn.Module):
    """价值网络"""

    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.state_dim = state_dim
        self.action_dim = action_dim

        self.fcs1 = nn.Linear(state_dim, 256)
        self.fcs1.weight.data = fanin_init(self.fcs1.weight.data.size())

        self.fcs2 = nn.Linear(256, 128)
        self.fcs2.weight.data = fanin_init(self.fcs2.weight.data.size())

        self.fca1 = nn.Linear(action_dim, 128)
        self.fca1.weight.data = fanin_init(self.fca1.weight.data.size())

        self.fc2 = nn.Linear(256, 128)
        self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())

        self.fc3 = nn.Linear(128, 1)
        self.fc3.weight.data.uniform_(-EPS, EPS)

    # 正向传播
    def forward(self, state, action):
        s1 = F.relu(self.fcs1(state))
        s2 = F.relu(self.fcs2(s1))
        a1 = F.relu(self.fca1(action))
        x = torch.cat((s2, a1), dim=1)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Actor 网络：

```python
class Actor(nn.Module):
    """策略网络，四层网络"""

    def __init__(self, state_dim, action_dim, action_lim):
        super(Actor, self).__init__()
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.action_lim = action_lim

        # 全连接层
        self.fc1 = nn.Linear(state_dim, 256)
        # pylog.info(self.fc1.weight.data.size())
        self.fc1.weight.data = fanin_init(self.fc1.weight.data.size())   # initialization of FC1
        # self.fc1.weight.data.normal_(0, 0.1) 服从(0，0.1)的正态分布
        # 全连接层
        self.fc2 = nn.Linear(256, 128)
        self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())

        # 全连接层
        self.fc3 = nn.Linear(128, 64)
        self.fc3.weight.data = fanin_init(self.fc3.weight.data.size())

        # 全连接层
        self.fc4 = nn.Linear(64, action_dim)
        self.fc4.weight.data.uniform_(-EPS, EPS)

    def forward(self, state):
        """正向传播"""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        action = F.tanh(self.fc4(x))
        action = action * self.action_lim   # action range
        return action
```

Replay buffer：

```
class MemoryBuffer:
    def __init__(self, size):
        self.buffer = deque(maxlen=size)
        self.maxSize = size
        self.len = 0

    def sample(self, count):
        batch = []
        count = min(count, self.len)
        batch = random.sample(self.buffer, count)

        s_arr = np.float32([arr[0] for arr in batch])
        a_arr = np.float32([arr[1] for arr in batch])
        r_arr = np.float32([arr[2] for arr in batch])
        s1_arr = np.float32([arr[3] for arr in batch])

        return s_arr, a_arr, r_arr, s1_arr

    def len(self):
        return self.len

    def add(self, s, a, r, s1):
        trainsition = (s, a, r, s1)
        self.len += 1
        if self.len > self.maxSize:
            self.len = self.maxSize
        self.buffer.append(trainsition)
```

整体网络：

```
# 策略网络（在线和目标）及其优化器--(online policy和 target policy )
self.actor = model.Actor(self.state_dim, self.action_dim, self.action_lim).to(self.device)
self.target_actor = model.Actor(self.state_dim, self.action_dim, self.action_lim).to(self.device)
self.actor_optimizer = torch.optim.Adam(self.actor.parameters(), LEARNING_RATE)

# 价值网络（在线和目标）及其优化器
self.critic = model.Critic(self.state_dim, self.action_dim).to(self.device)
self.target_critic = model.Critic(self.state_dim, self.action_dim).to(self.device)
self.critic_optimizer = torch.optim.Adam(self.critic.parameters(), LEARNING_RATE)
```

四、损失函数

```
s1, a1, r1, s2 = self.ram.sample(BATCH_SIZE)
s1 = Variable(torch.from_numpy(s1).to(self.device))
a1 = Variable(torch.from_numpy(a1).to(self.device))
r1 = Variable(torch.from_numpy(r1).to(self.device))
s2 = Variable(torch.from_numpy(s2).to(self.device))

a2 = self.target_actor.forward(s2).detach()  # 根据S2得出的下一个动作a2
next_val = torch.squeeze(self.target_critic.forward(s2, a2).detach())  # 根据s2和a2得出的目标值
y_expectd = r1 + GAMMA * next_val  # 目标回报值
# UserWarning: Using a target size (torch.Size([1])) that is different to the input size (torch.Size([])).
# y_predicted = torch.squeeze(self.critic.forward(s1, a1))
y_predicted = torch.squeeze(self.critic.forward(s1, a1), -1)  # 主价值网络得到的回报值,

loss_critic = F.smooth_l1_loss(y_predicted, y_expectd)  # 价值损失（回报值的损失）
self.critic_optimizer.zero_grad()  # 优化器参数置为零
if self.write_loss:
    self.write_loss(step, loss_critic.item(), "loss_critic")  # 将损失值保存到文件中
loss_critic.backward()  # 反向传播,
self.critic_optimizer.step()  # 修改价值网络参数

pred_a1 = self.actor.forward(s1)
loss_actor = -1 * torch.sum(self.critic.forward(s1, pred_a1))
self.actor_optimizer.zero_grad()
if self.write_loss:
    self.write_loss(step, loss_actor.item(), "loss_actor")
loss_actor.backward()
self.actor_optimizer.step()
```

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

图 106 DDPG 算法

五、训练与优化

```
# 启动训练
try:
    for _ep in range(start_epoch, etc.MAX_EPISODES):
        print(" ")
        print("%s: 第%s轮训练开始====================================" % (datetime.datetime.now(), _ep))

        # 重置智能体、环境、训练器
        state_now, reward_now = env.reset()
        env.step_count = 0
        agent.reset()

        # 智能体作决策，产生动作，动作影响环境，智能体根据动作的效果进行训练优化
        for step in range(etc.MAX_STEPS):
            # 智能体根据当前的状态及回报值，进行决策，生成下一步的动作
            action_new = agent.make_decision(np.float32(state_now), reward_now)

            # 环境执行动作，生成下一步的状态及回报值
            state_new, reward_new = env.execute_action(action_new)

            # 根据推演结果，训练一次智能体
            agent.train(np.float32(state_now), action_new, reward_new, np.float32(state_new), cur_step)
            cur_step += 1
            write_start_step_file(step_file_path, str(cur_step))
            # 更新状态、回报值
            state_now = state_new
            reward_now = reward_new

            # 打印提示
            print("%s: 轮数:%s 决策步数:%s  Reward:%.2f" % (datetime.datetime.now(), _ep, step, reward_now))

            # 检查是否结束本轮推演
            if env.check_done():
                break
            # 如果显示此图，代码会卡住
            if cur_step % 100 == 0:
                # show_pic()
                pass
        write_start_epoch_file(epoch_file_path, str(_ep))
```

空海一体联合作战智能体开发案例解析

一、想定介绍

作战目标

为了对航母编队赋予作战指挥能力，设计并训练航母编队作战指挥官智能体，在 XX 海峡设置了红蓝对等航母编队兵力：16 架 F-35C 战斗机（反舰、空战）、一艘福特级航空母舰、一艘阿里伯克级导弹驱逐舰进行对等博弈对抗。红方采用强化学习智能体，蓝方无智能体，但设置了 81 种军事规则与红方进行对抗。

红方智能体对所有作战单元构建了场内单元、损失单元、任务单元、区域内单元、消耗武器、任务单元剩余武器、探测目标、区域内探测目标、仿真进度等 9 类状态共计 740 维态势模型；以任务类型、任务区域、作战单元、作战时间、作战方法为基本要素，对飞机构建了防御性巡逻、攻击性巡逻、对舰攻击等 3 类共 71 种作战任务模型；以尽可能多的消灭蓝方目标、保护己方目标作为评估指标，采用分布式并行计算，开展强化学习训练。

作战单元

表 3 红方作战单元表

序号	类别	类型	建制单位	总数量	备注
1	固定翼舰载机	多用途（战斗/攻击）	F-35C 型“闪电 II”战斗机(8 架反舰、8 架空战)	16	

序号	类别	类型	建制单位	总数量	备注
2	航空母舰(航空舰)	CVN -核动力航空母舰	CVN-78“杰拉德.R.福特”福特级号航空母舰	1	
3	水面战斗舰艇	DDG - 导弹驱逐舰	DDG 113“约翰.芬”阿里伯克级 Flight IIA 导弹驱逐舰	1	

表 4 蓝方作战单元表

序号	类别	类型	建制单位	总数量	备注
1	固定翼,舰载机	多用途（战斗/攻击）	F-35C 型“闪电 II”战斗机(8 架反舰、8 架空战)	16	
2	航空母舰(航空舰)	CVN -核动力航空母舰	CVN-78“杰拉德.R.福特”福特级号航空母舰	1	
3	水面战斗舰艇	DDG - 导弹驱逐舰	DDG 113“约翰.芬”阿里伯克级 Flight IIA 导弹驱逐舰	1	

二、环境类设计

1、观测（observation）

类型采用 one-hot 方式编码；单元数量采用归一化处理，再采用 log 方式处理。

单元类型 001：反舰-空战飞机 0001、空战飞机 0010、驱逐舰 0100、航母 1000

武器类型 010：通用箔条 0001、空空导弹 0010、反舰导弹 0100、舰空导弹 1000

探测类型 100：导弹 001、飞机 010、舰艇 100

（1）空闲单元类型及其数量：

空闲单元类型 001001-->维度(6+4+2)*4=48

（2）损失的单元类型及其数量：

损失的单元类型 001010-->维度(6+4+2)*4=48

（3）在任务中的单元类型及其数量：

在任务中的单元类型 001100-->维度(6+4+2)*4=48

（4）分别在 zone1，zone2，zone3，zone4，aw_zone 区域内的单元类型及其数量：

分别在 zone1，zone2，zone3，zone4，aw_zone 区域内的单元类型

00110000001 ， 00110000010 ， 00110000100 ， 00110001000 ， 00110010000

-->维度(11+4+2)*4*5 = 340

（5）消耗的武器类型及其数量：

消耗的武器类型 01001-->维度(5+4+2)*4=44

（6）在任务中的所有单元剩余的武器类型及其数量：

在任务中的所有单元剩余的武器类型 01010-->维度(5+4+2)*4=44

（7）探测到的敌方单元类型及其数量：

探测类型 100-->维度(3+3+2)*4=32

（8）出现在 zone1，zone2，zone3，zone4 中的敌方单元的类型及其数量：

出现在 zone1，zone2，zone3，zone4 中的敌方单元的类型

1000001，1000010，1000100，1001000-->维度(7+3+2)*4*4 = 208

（9）推演进度：

总时长 6000s，分别划分成 60、20、6、3 阶段，分别采用 60、20、6 、 3 维 向 量 表 示 。 计 算 方 式 ： idx1=value*(60/6000) ， idx2=value*(20/6000)， idx3=value*(6/6000)， idx4=value*(3/6000) 比 如

当前推演到 100s，idx1=1，60 维向量的第 1 位置 1，其余 59 位置 0。

以上 9 种状态累计 740 维。

2、动作（action）

（1）创建巡逻任务；

（2）任务区域：zone1，zone2，zone3，zone4；

（3）任务单元：随机从 8 架空战飞机中选择 4 架；

（4）任务时间：以 2 分钟为间隔；

（5）怎么执行：

修改任务参数：单机或者 2 机编队；

任务条令：是否开启雷达、是否开启干扰机、设置自动开火距离等；

3、奖励（reward）

奖励主要采用战损值。

三、网络架构

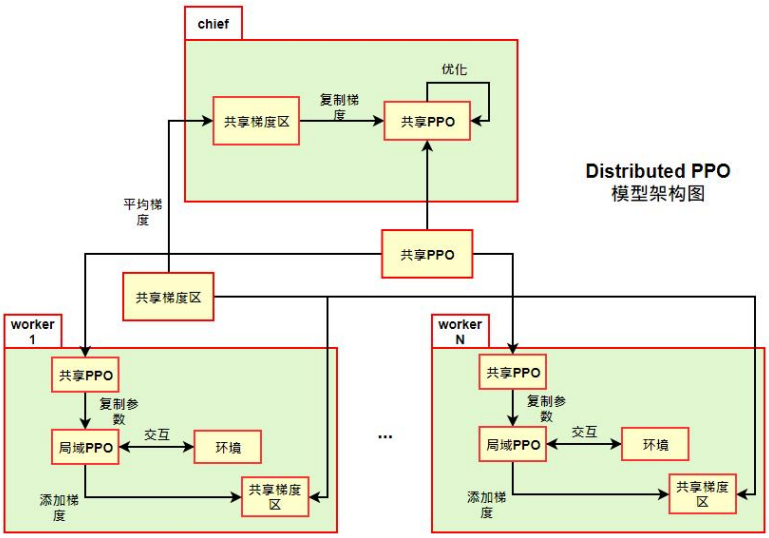


图 107 DPPO 算法架构

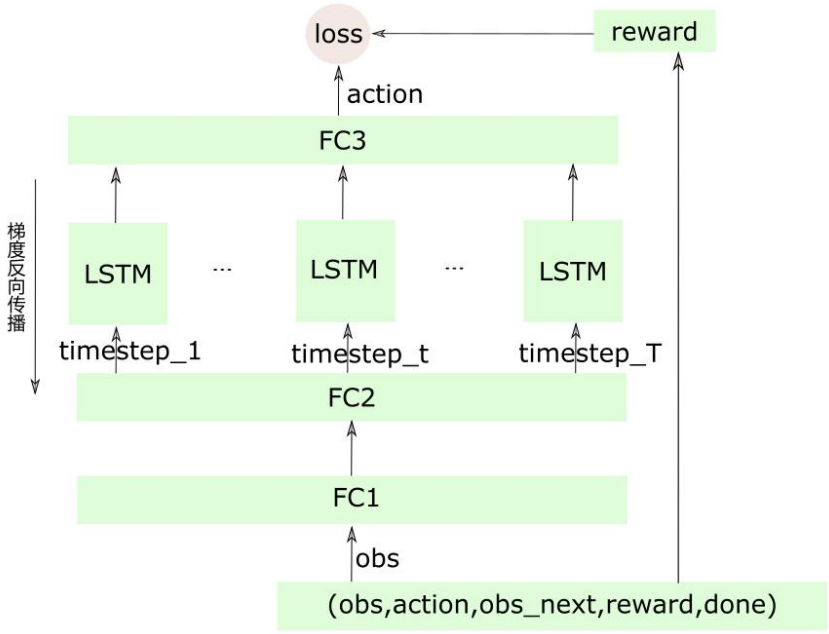

loss
reward
action
FC3
梯度反向传播
LSTM
LSTM
LSTM
timestep_1
timestep_t
timestep_T
FC2
FC1
obs
(obs,action,obs_next,reward,done)

图 108 模型架构

四、损失函数

```
class Model(object):
    def __init__(self, *, policy, ob_space, ac_space, nbatch_act, nbatch_train,
                 unroll_length, ent_coef, vf_coef, max_grad_norm, scope_name,
                 value_clip=False):
        sess = tf.get_default_session()

        act_model = policy(sess, scope_name, ob_space, ac_space, nbatch_act, 1,
                           reuse=False)
        train_model = policy(sess, scope_name, ob_space, ac_space, nbatch_train,
                             unroll_length, reuse=True)

        A = tf.placeholder(shape=(nbatch_train,), dtype=tf.int32)
        ADV = tf.placeholder(tf.float32, [None])  # 优势函数
        R = tf.placeholder(tf.float32, [None])
        OLDNEGLOGPAC = tf.placeholder(tf.float32, [None])  # neglogpac
        OLDVPRED = tf.placeholder(tf.float32, [None])
        LR = tf.placeholder(tf.float32, [])
        CLIPRANGE = tf.placeholder(tf.float32, [])

        neglogpac = train_model.pd.neglogp(A)
        entropy = tf.reduce_mean(train_model.pd.entropy())

        vpred = train_model.vf
        vpredclipped = OLDVPRED + tf.clip_by_value(train_model.vf - OLDVPRED,
                                                   -CLIPRANGE, CLIPRANGE)
        vf_losses1 = tf.square(vpred - R)
        if value_clip:
            vf_losses2 = tf.square(vpredclipped - R)
            vf_loss = .5 * tf.reduce_mean(tf.maximum(vf_losses1, vf_losses2))
        else:
            vf_loss = .5 * tf.reduce_mean(vf_losses1)
        ratio = tf.exp(OLDNEGLOGPAC - neglogpac)
        pg_losses = -ADV * ratio
        pg_losses2 = -ADV * tf.clip_by_value(ratio, 1.0 - CLIPRANGE,
                                             1.0 + CLIPRANGE)
        pg_loss = tf.reduce_mean(tf.maximum(pg_losses, pg_losses2))
        approxkl = .5 * tf.reduce_mean(tf.square(neglogpac - OLDNEGLOGPAC))
        clipfrac = tf.reduce_mean(
            tf.to_float(tf.greater(tf.abs(ratio - 1.0), CLIPRANGE)))
        loss = pg_loss - entropy * ent_coef + vf_loss * vf_coef
        # TODO
        tf.summary.scalar('loss', loss)
        # tf.summary.scalar('pg_loss', pg_loss)
        # tf.summary.scalar('vf_loss', vf_loss)
        summary_op = tf.summary.merge_all()
        params = tf.trainable_variables(scope=scope_name)
        grads = tf.gradients(loss, params)
        if max_grad_norm is not None:
            grads, _grad_norm = tf.clip_by_global_norm(grads, max_grad_norm)
        grads = list(zip(grads, params))
        trainer = tf.train.AdamOptimizer(learning_rate=LR, epsilon=1e-5)
        _train = trainer.apply_gradients(grads)
        new_params = [tf.placeholder(p.dtype, shape=p.get_shape()) for p in params]
        param_assign_ops = [p.assign(new_p) for p, new_p in zip(params, new_params)]
```

```
def train(lr, cliprange, obs, returns, dones, actions, values, neglogpacs,
          states=None):
    advs = returns - values
    advs = (advs - advs.mean()) / (advs.std() + 1e-8)
    if isinstance(ac_space, MaskDiscrete):
        td_map = {train_model.X: obs[0], train_model.MASK: obs[-1], A: actions,
                  ADV: advs, R: returns, LR: lr, CLIPRANGE: cliprange,
                  OLDNEGLOGPAC: neglogpacs, OLDVPRED: values}
    else:
        td_map = {train_model.X: obs, A: actions, ADV: advs, R: returns, LR: lr,
                  CLIPRANGE: cliprange, OLDNEGLOGPAC: neglogpacs, OLDVPRED: values}
    if states is not None:
        td_map[train_model.STATE] = states
        td_map[train_model.DONE] = dones

    # TODO
    # return sess.run(
    #   [pg_loss, vf_loss, entropy, approxkl, clipfrac, _train],
    #   td_map
    # )[:-1]

    return sess.run([pg_loss, vf_loss, entropy, approxkl, clipfrac, summary_op, _train], td_map)[:-1]

self.loss_names = ['policy_loss', 'value_loss', 'policy_entropy',
                   'approxkl', 'clipfrac']
```

五、分布式训练与优化

1、Actor

```
def run(self):
    num = 0  # the number of sample data
    while True:
        gc.collect()
        # fetch model
        t = time.time()
        self._update_model()
        tprint("Update model time: %f" % (time.time() - t))
        t = time.time()
        # rollout
        num += 1
        unroll = self._nstep_rollout(num)
        if self._enable_push:
            if self._data_queue.full(): tprint("[WARN]: Actor's queue is full.")
            self._data_queue.put(unroll)
            tprint("Rollout time: %f" % (time.time() - t))
```

2、Learner

```
def run(self):
    # while len(self._data_queue) < self._data_queue.maxlen: time.sleep(1)
    # while len(self._episode_infos) < self._episode_infos.maxlen / 2:
    #   time.sleep(1)
    while len(self._episode_infos) < 6:
        time.sleep(1)

    batch_queue = Queue(4)
    batch_threads = [
        Thread(target=self._prepare_batch,
               args=(self._data_queue, batch_queue,
                     self._batch_size * self._unroll_split))
        for _ in range(8)
    ]
    for thread in batch_threads:
        thread.start()

    updates, loss = 0, []
    time_start = time.time()
    while True:
        while (self._learn_act_speed_ratio > 0 and
               updates * self._batch_size >= \
               self._num_unrolls * self._learn_act_speed_ratio):
            time.sleep(0.001)
        updates += 1
        lr_now = self._lr(updates)
        clip_range_now = self._clip_range(updates)

        batch = batch_queue.get()
        obs, returns, dones, actions, values, neglogpacs, states = batch
        # TODO
        # loss.append(self._model.train(lr_now, clip_range_now, obs, returns, dones,
        #                               actions, values, neglogpacs, states))

        loss_tuple = self._model.train(lr_now, clip_range_now, obs, returns, dones, actions, values, neglogpacs,
                                       states)
        loss.append(loss_tuple[:-1])
        self._model.writer.add_summary(loss_tuple[-1], updates)

          self._model_params = self._model.read_params()

          if updates % self._print_interval == 0:
              loss_mean = np.mean(loss, axis=0)
              batch_steps = self._batch_size * self._unroll_length
              time_elapsed = time.time() - time_start
              train_fps = self._print_interval * batch_steps / time_elapsed
              rollout_fps = len(self._data_timesteps) * self._unroll_length / \
                            (time.time() - self._data_timesteps[0])
              var = explained_variance(values, returns)
              avg_reward = safemean([info['r'] for info in self._episode_infos])
              tprint("Update: %d  Train-fps: %.1f Rollout-fps: %.1f   "
                     "Explained-var: %.5f Avg-reward %.2f Policy-loss: %.5f   "
                     "Value-loss: %.5f    Policy-entropy: %.5f    Approx-KL: %.5f "
                     "Clip-frac: %.3f Time: %.1f" % (updates, train_fps, rollout_fps,
                                                     var, avg_reward, *loss_mean[:5], time_elapsed))
              time_start, loss = time.time(), []

          if self._save_dir is not None and updates % self._save_interval == 0:
              os.makedirs(self._save_dir, exist_ok=True)
              save_path = os.path.join(self._save_dir, 'checkpoint-%d' % updates)
              self._model.save(save_path)
              tprint('Saved to %s.' % save_path)
```

智能体开发实操练习

课程目标：

把无人机突防案例单机训练修改为 ray 分布式训练，并进行模型优化。

第一步：选择 DDPG 算法、设置模型算法配置，设置超参数调优算法。

第二步：修改环境类与 ray 接口，主要是修改 init、step 和 reset 函数。

第三步：修改环境类的状态生成、动作空间和奖励设计。

多智能体分层强化学习框架介绍

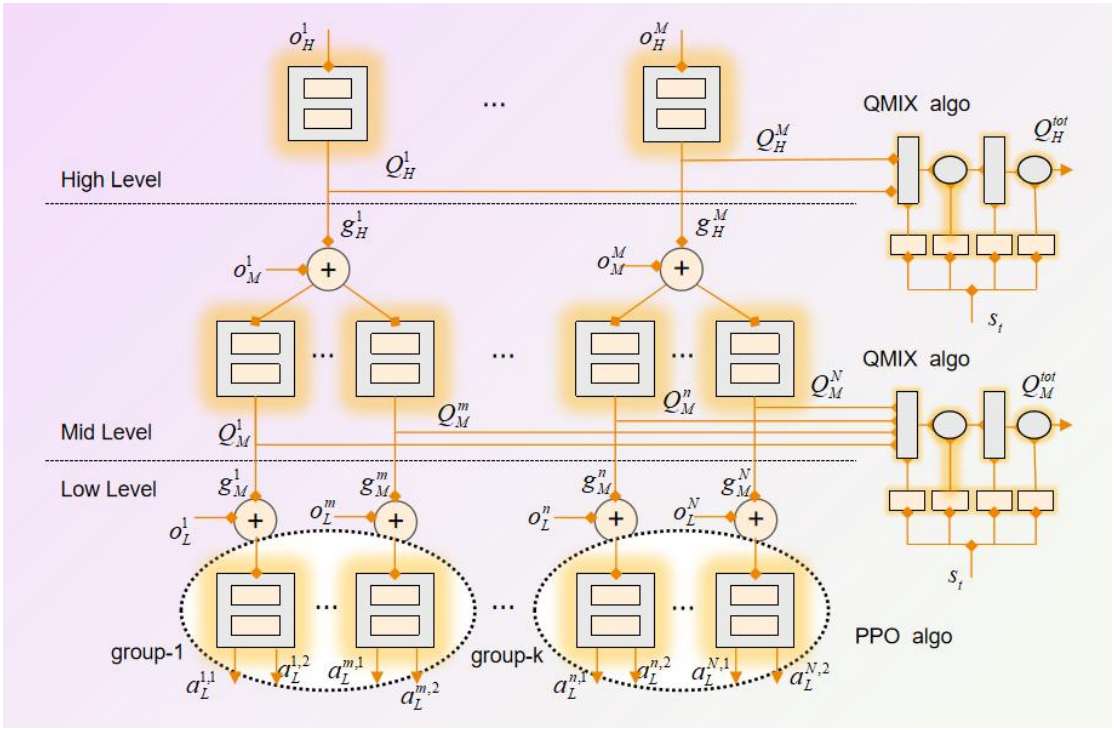


图 109 分层框架

o 表示 observation、g 和 a 表示动作、Q 表示 Q 值

分层架构的上层和中层是由多智能体算法 QMIX 构成，上层有 M 个智能体，它们共享参数。中层有 N 个智能体，中层智能体数量与上层智能体数量比例为 3:1，中层每 3 个智能体接收上层智能体的动作，与 observation 拼接作为中层智能体的输入。下层智能体的数量与中层智能体的数量一致，下层智能体接收对应中层智能体的动作作为 observation 的一部分。上中下层模型之间没有反向传播过程。

在本案例中，下层智能体采用单智能体 PPO 算法，共有 12 个智能体，每个智能体控制墨子中的一个作战单元编队，12 个作战单元编队中分别有 4 个歼轰飞机编队，4 个空战飞机编队和 4 个电子干扰飞机编队，

其中歼轰飞机编队有 6 架歼轰飞机，空战飞机编队有 2 架空战飞机，电子干扰飞机编队有一架电子干扰飞机。下层 12 个智能体中控制不同类型编队的智能体共享不同的参数，12 个智能体有 3 套模型参数。中层和上层智能体采用多智能体 QMIX 算法，其中中层有 12 个智能体，分别对应下层的 12 个智能体。上层有 4 个智能体，每个智能体控制一个智能体编队，每个智能体编队包含一个歼轰飞机编队，一个空战飞机编队和一个电子战飞机编队。

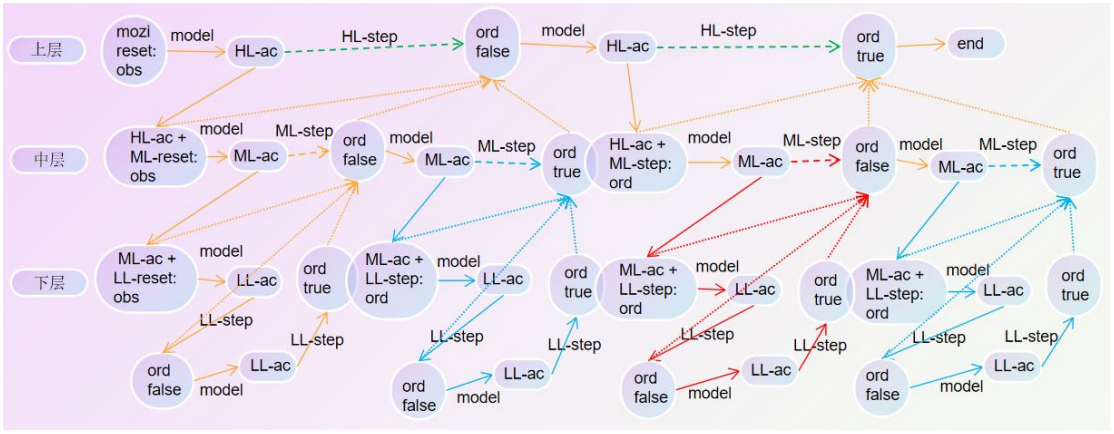


图 110 MDP 流程

细虚线表示合成数据，粗虚线表示流程示意，实线表示数据流

HL/ML/LL 分别表示 high-level/mid-level/low-level

ac 表示 action

ord 表示 obs/reward/done

true/false 表示是否返回上一层

上图表示三层智能体的 MDP 之间的调用返回流程（call-and-return），上一层智能体根据当前的状态计算出动作，然后调用下层的智能体，并把这个动作拼接到状态之中，在下一层智能体执行一段时间之后，在把这一段时间内产生的状态拼接，作为上一层智能体的状态。

规则智能体

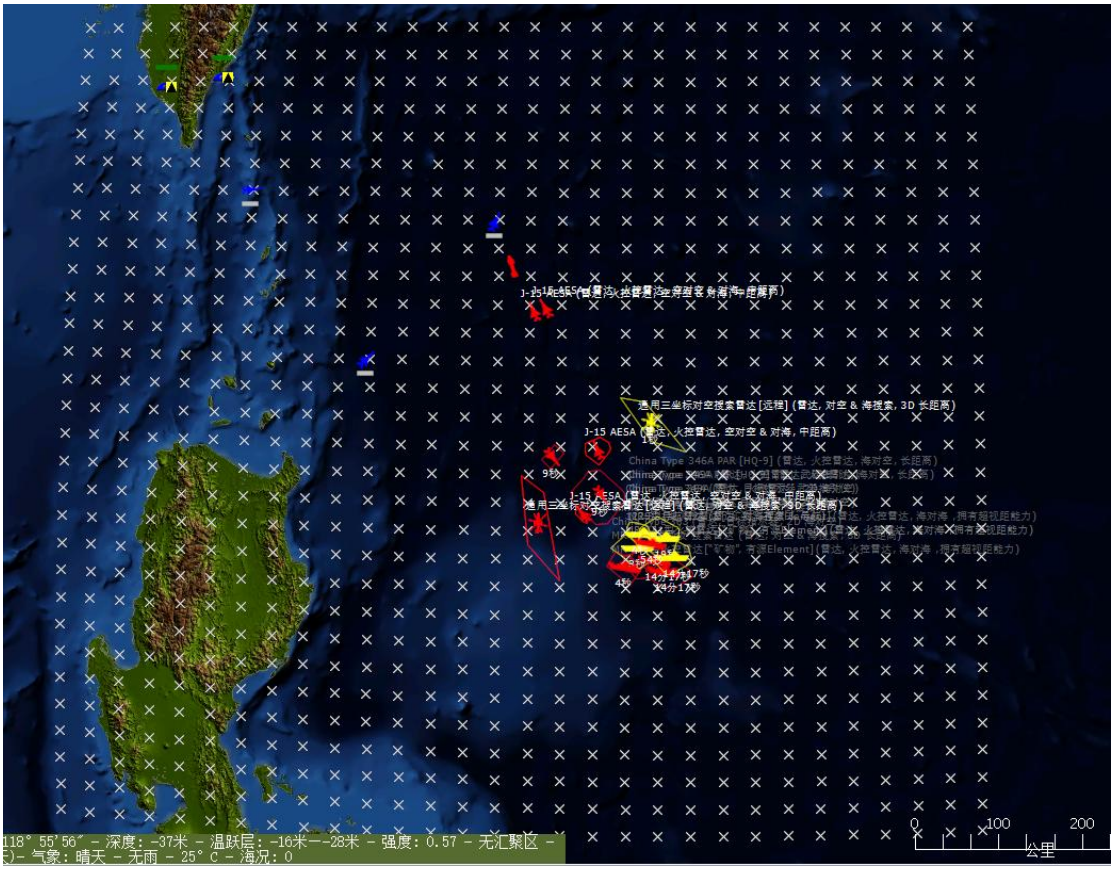


图 111 规则智能体

在想定刚开始推演的时候，会根据蓝方机场的位置和航母编队大致范围绘制战场的大致范围并网格化，然后启动规则智能体感知态势。规则智能体控制的单元包括 1 架预警机，4 架空战飞机。其中预警机的巡逻点有多个，沿着航母编队的方向分布，预警飞机随着态势的变化后撤或推进。预警机前方设置左右两路空战飞机巡逻方向，一方面感知态势，一方面保护预警机，空战飞机随着推演向航母编队可能的方向推进，在空的空战飞机一旦被击毁或者返航会有一架空战飞机进行补充，每一路总共有两架空战飞机。

分层智能体

在规则智能体之后会启动分层智能体。

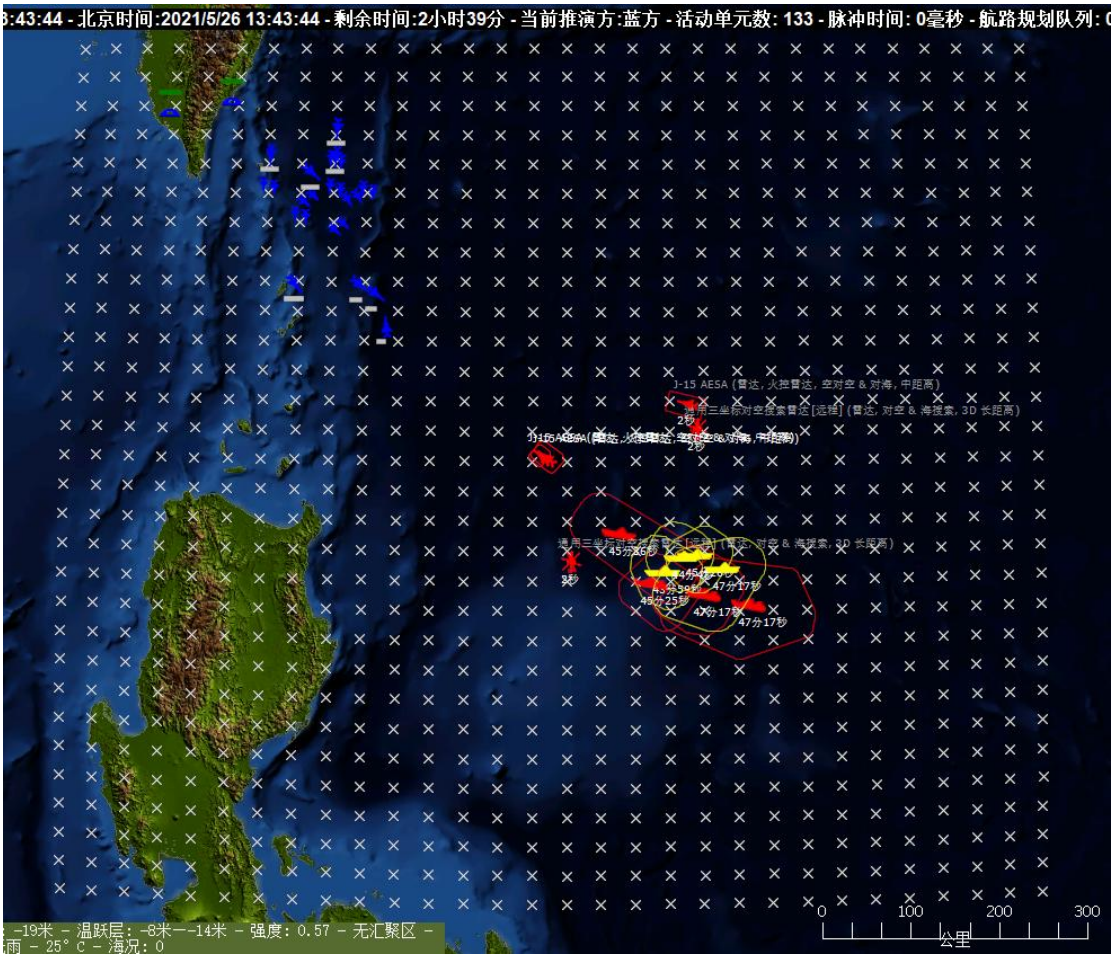


图 112 分层智能体

状态空间设计

下层智能体局部视野（observation）：单元自身的属性，包括经纬度、航向、航速等；在单元 n 公里范围之内的我方单元的信息，以及探测单元的信息，包括经纬度、航向、航速等；当前的推演进度。中上层智能体的局部视野是与之对应的下层智能体汇总生成的。

中上层智能体全局状态（state）：本推演方各类单元信息（包括经纬度、航向、航速等）的均值，探测到敌方单元信息（包括探测类别、探测状态、经纬度、航向、航速等）。当前推演进度。

动作空间设计

中下层智能体控制 12 个作战编队，每个智能体的状态是作战编队中

每个单元状态的汇总，中层智能体负责作战编队的区域移动，每次决策可移动的区域是作战编队所在区域以及附近九宫格区域任意一个，目标是控制编队到达上层智能体为编队选择的对航母的打击点，下层智能体根据控制的作战编队类型不同，控制单元的微操、条令的设置或打击目标的选择等，目标是取得最优的空中搏斗效果。

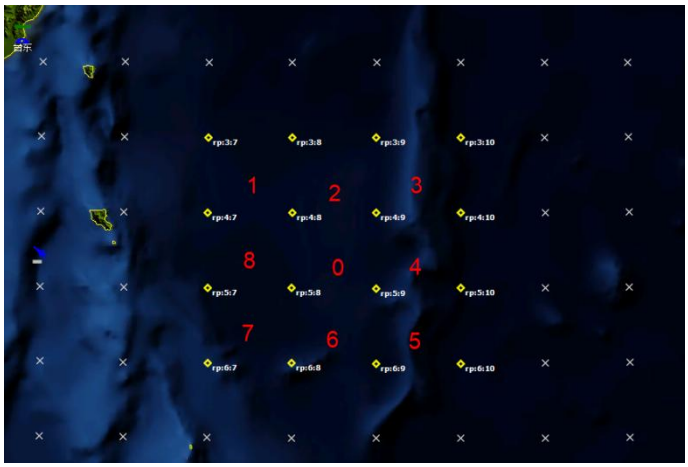


图 113 中层智能体动作选择

上层智能体控制 4 个大的智能体作战编队，每个编队中包含一个歼轰机编队、空战飞机编队和电子干扰机编队，智能体作战编队以歼轰机编队为中心，歼轰机的作用是打击航母编队，也带有少量空战导弹做自防御，空战飞机编队在歼轰机前方起到保护作用，电子干扰机在歼轰机后方，主要起到电子干扰的作用。上层智能体的动作是从构建的打击链中选择打击点。

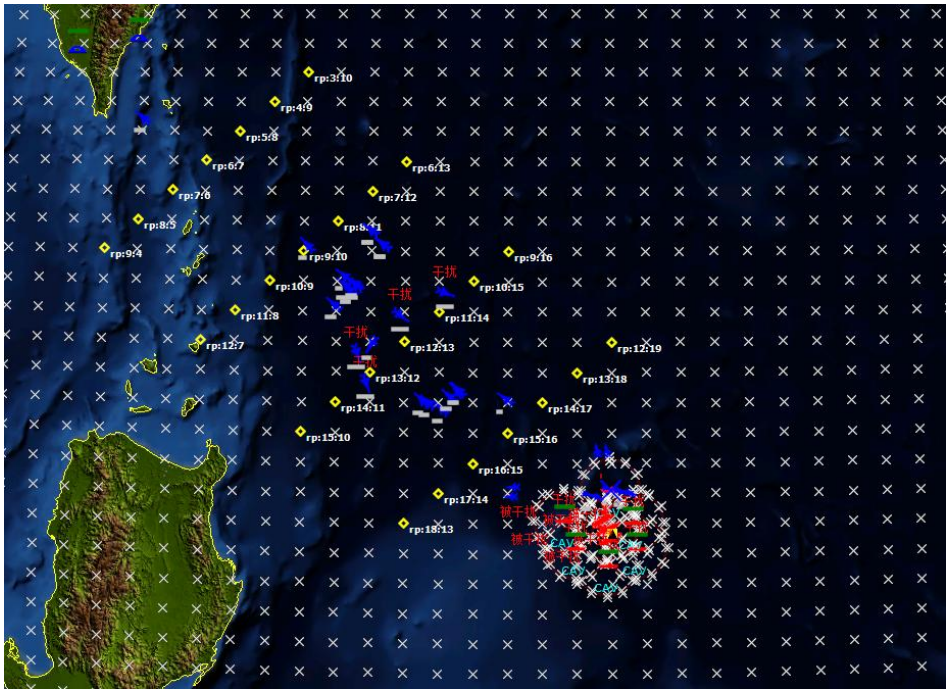


图 114 上层智能体动作选择

奖励机制设计

上层智能体目标打击点的选择应该邻近，形成规模优势，所以不同智能体编队打击点的选择越近，奖励越高。中层智能体控制的是作战编队的移动，中层智能体的奖励原则是空战飞机比歼轰飞机更靠近敌情。下层智能体控制的是作战编队的高度和雷达，奖励采用战损分数。

智能蓝军对抗演练与研讨交流

课程目标：

通过人机对抗演练，一方面来发现智能体的弱点，寻找击败智能体的方法；另一方面，通过对抗演练来发现当前智能算法面临哪些问题，比如智能体策略实现问题、策略多样性的问题和智能体迁移性等问题，提出相应的优化建议，思考实现思路。

[1] Ng A Y, Harada D, Russell S. Policy invariance under reward transformations: Theory and application to reward shaping[C]//ICML. 1999, 99: 278-287.

[2] Wiewiora E, Cottrell G W, Elkan C. Principled methods for advising reinforcement learning agents[C]//Proceedings of the 20th International Conference on Machine Learning (ICML-03). 2003: 792-799.

[3] Devlin S M, Kudenko D. Dynamic potential-based reward shaping[C]//Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems. IFAAMAS, 2012: 433-440.

[4] Harutyunyan A, Devlin S, Vrancx P, et al. Expressing arbitrary reward functions as potential-based advice[C]//Twenty-Ninth AAAI Conference on Artificial Intelligence. 2015.

[5] Suay H B, Brys T, Taylor M E, et al. Learning from demonstration for shaping through inverse reinforcement learning[C]//Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems. 2016: 429-437.

[6] Wiewiora E. Potential-based shaping and Q-value initialization are equivalent[J]. Journal of Artificial Intelligence Research, 2003, 19: 205-208.

[7] Zou H, Ren T, Yan D, et al. Reward shaping via meta-learning[J]. arXiv preprint arXiv:1901.09330, 2019.

[8] Chentanez N, Barto A G, Singh S P. Intrinsically motivated reinforcement learning[C]//Advances in neural information processing systems. 2005: 1281-1288.

[9] Burda Y, Edwards H, Pathak D, et al. Large-scale study of curiosity-driven learning[J]. arXiv preprint arXiv:1808.04355, 2018.

[10] Volodymyr M, Koray K, David S, et al. Human-level control through deep reinforcement learning[J].Nature,2019,518(7540):529-33 页.

[11] Hessel M， Modayil J， Hasselt H V，et al. Rainbow: Combining Improvements in Deep Reinforcement Learning[J]. 2017.

[12] Lillicrap T P， Hunt J J， Pritzel A，et al. Continuous control with deep reinforcement learning[J]. Computer ence, 2015.

[13] Schulman J， Moritz P， Levine S，et al. High-Dimensional Continuous Control Using Generalized Advantage Estimation[J]. Computer ence, 2015.

[14] Heess N， Dhruva T B， Sriram S，et al. Emergence of Locomotion Behaviours in Rich Environments[J]. 2017.

[15] Espeholt, Lasse, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. arXiv preprint arXiv:1802.01561 (2018).

[16] Faccio F， Schmidhuber J . Parameter-based Value Functions[J]. 2020.

[17] Sunehag P , Lever G , Gruslys A , et al. Value-Decomposition Networks For Cooperative Multi-Agent Learning[J]. 2017.

[18] Rashid T , Samvelyan M , Witt C D , et al. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning[J]. 2018.

[19] Dietterich T G . Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition[J]. Journal of Artificial Intelligence Research, 1999.

[20] A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning[C]// 31st Conference on Neural Information Processing Systems (NIPS), 4-9 December 2017, Long Beach, CA, USA. 2017.

[21] Jaderberg M , Dalibard V , Osindero S , et al. Population Based Training of Neural Networks[J]. 2017.