基于强化学习的无人机群目标追踪方法研究

南京大学　袁想，吴俊锋，张辰璐，郭小齐

（1．南京大学计算机科学与技术系，江苏省 南京市 210023；

2．南京大学计算机科学与技术系，江苏省 南京市 210023；

3．南京大学计算机科学与技术系，江苏省 南京市 210023；

4．南京大学工程管理学院，江苏省 南京市 210023）

陶先平（教授），汪亮（副教授）

摘要：针对无人机群目标追踪问题，提出了基于多智能体强化学习的算法框架，提出了涵盖训练效率和可扩展性两方面的综合性度量指标，构建了高效的仿真训练与测试实验平台。通过实验验证了包括 DQN，Double DQN，Dueling DQN，Actor-Critic，DDPG，MADDPG 在内的强化学习算法在目标追踪问题上的效果和效率。实验结果表明：在我们设定的场景中，目标随机游走条件下，MADDPG 收敛速度和收敛后达到的奖赏都显著优于其它单智能体扩展为多智能体的算法，但是在对抗条件下是 Double DQN 利用 distributed 非参数共享扩展算法表现最好。Actor-Critic 算法表现最差。单智能体扩展为多智能体的算法的扩展方式中，distributed 非参数共享可以达到更好的效果。MADDPG 算法对于智能体数量的扩展性表现最好。在对现有算法进行了全面评估的同时，也为多智能体强化学习算法在目标追踪问题表现方面的评估提供了一个基准的测试平台。

关键词：目标追踪，多智能体强化学习，仿真训练与测试平台

一、引言

目标追踪问题是值得研究的现实生活中常见的一类重要问题。目前无人机在解决目标追踪问题上具有广泛的应用前景：警方目前已经将无人机投入日常巡逻工作，将来有可能用于逃犯追踪、窝点搜索等问题；大量的航拍工作也由无人机承担，而追踪算法的研究则能进一步省下人力物力，自动追踪拍摄目标。同时，我国空军也有提出更为具体的应用场景，即多架无人机通过自主协作对多辆目标车辆进行跟踪与围捕，并在此环境下举办比赛“无人争锋”，而我们也将此作为我们的研究问题场景。无人机具有机动性高，隐蔽性强，可远程操作等特点，非常适合作为追踪这一动作的实施者。而且，由于现实环境错综复杂，人类难以及时根据变化调整策略，因此需要无人机具有自我调整、自我更新策略的能力。如何让无人机自主协作地进行目标追踪，便是解决问题的关键技术所在。

围绕这个问题，现有解决追踪问题的方案有两类：第一类是基于经典的控制规划方案，即通过最优控制的方法，采用梯度下降里的 Jacobian 或 Hessian 信息，对路径不断进行优化，这种方法较为方便，无需

国家级大学生创新创业训练计划支持项目（202010284084Z）

作者简介：吴俊锋（1999-），男，广东湛江，计算机科学与技术，2017 级，软件方法学；袁想（1999-），男，江苏高邮，计算机科学与技术，2017 级，机器学习；郭小齐（1998-），男，湖南长沙，自动化，2017 级，数据科学；张辰璐（1999-），女，江苏无锡，计算机科学与技术，2017 级，数据科学。

建立精确的数学模型即可投入使用，但是对于灵活多变的场景不能有较好的效果。而现在兴起的基于多智能体强化学习（MARL）的决策方案，在这个问题上有着较好的表现。多智能体强化学习（MARL）通过在智能体与环境的交互过程中，学习到一个较好的策略以达成累积奖赏最大化或实现特定目标来解决问题。多智能体强化学习（MARL）可以充分利用各类信息进行决策，并且由于经过大量的学习，可以应对各种场景。但目前多智能体强化学习依然具有局限性：控制大规模无人机集群的可扩展性问题，通常集群规模存在上限，一旦超过，算法效果将快速下跌；学习的效率问题，现有的多智能体强化学习（MARL）方法需要长时间的学习才能获得较好的策略。我们希望能在利用 MARL 算法解决目标追踪问题方面进行研究。在我们研究的MARL算法中，包括将传统的单智能体强化学习算法扩展到多智能体场景的算法，例如DQN，DDPG 等，还包括专门为多智能体设计的正统多智能体强化学习算法 MADDPG。

本文主要研究的内容是构建一套基于多智能体强化学习目标追踪的基准测试框架，包含多智能体强化学习算法框架，具有灵活性高，可扩展性强，可以兼容不同的算法的特点；搭建面向目标追踪问题的实验数值仿真模拟平台，具有效率高，可扩展性强的特点；设计一组评价指标：K step-reward 指标，单智能体扩展途径-reward，智能体数量-reward 指标，能够系统性地评价不同算法在追踪问题上的表现优劣；在我们设计的基准测试框架中将包括 DQN，Double DQN，Dueling DQN，Actor-Critic，DDPG，MADDPG 在内的多智能体强化学习算法实现并应用到我们研究的目标追踪问题。针对上述算法在搭建的实验仿真平台上利用评判指标进行了评估。

综上所述，本文做出的贡献包括：

1、提出一种多智能体强化学习算法实现框架，在框架之中我们将现有的主流强化学习算法（DQN，Double DQN，Dueling DQN，Actor-Critic，DDPG，MADDPG）应用到了追踪问题中，并且针对 DQN，Double DQN，Dueling DQN，Actor-Critic，DDPG 这些传统单智能体强化学习方法，我们分别利用了两种途径：centralized，distributed 将它们扩展成为多智能体强化学习算法。同时在我们设计的强化学习算法实现框架中，针对单智能体强化学习算法扩展为多智能体强化学习算法实现了训练过程实现和算法实现的解耦。

2、设计并实现了高效的实验仿真平台以及一组评判指标，系统性地评价了上述主流强化学习算法以及将传统单智能体强化学习方法扩展为多智能体强化学习算法的三种途径在目标追踪问题上的表现优劣。

3、针对上述主流强化学习算法使用上述算法框架进行了构建并且在搭建的实验平台上利用评判指标进行了评估。

本文其余部分组织如下：第二部分讲述针对追踪问题的相关现有工作，第三部分讲述基于多智能体强化学习（MARL）的无人机群目标追踪技术框架和实验平台设计，第四部分讲述强化学习算法实现与实验设计，第五部分介绍实验结果，第六部分进行针对实验结果的进一步讨论，第七部分进行总结与展望，第八部分为致谢。

二、相关工作

（一）目标追踪问题

曾经，目标追踪问题由于本身具有极高的难度，而且也缺少高质量的能产生良好试验结果的数据，发展较为缓慢。但是随着强化学习算法的不断发展以及计算机算力的不断提升，近年来单目标追踪领域各种方法逐渐崭露头角，可谓欣欣向荣。而对于多目标追踪，研究则不像单目标追踪开展的那么顺利，当然，这也是因为随着目标的增加以及智能体增加，问题的复杂程度也是几何倍数增加，同时优质的数据集也相对较少，可以用于参考与获取灵感的开源代码也更少。多目标追踪也更接近于实际应用场景，因此问题的工程化程度也有所提升。

（二）多智能体强化学习（Multi Agent Reinforcement Learning，MARL）

强化学习是一个“trial-and-error”的过程，通过智能体在环境中采取行动，产生交互，根据获得的反馈信息对模型进行迭代与优化。而在传统强化学习的基础上，增加智能体的数量，即多个智能体同时与环境进行交互时，系统便成为了一个多智能体系统。每个智能体在采取行动时，仍是依照强化学习的目标即最大化所能获得的累积奖赏，但此时环境变量的改变就与所有智能体的联合动作相联系。因此多智能体强化学习是考虑联合行动结果的算法。

目前，MARL 的研究主要分为四个方向。分别是 Analysis of emergent behaviors（行为分析），Learning communication（通信学习），Learning cooperation（协作学习），Agents modeling agents（智能体建模）。行为分析主要是 MARL 发展初期的研究方向，这一方向往往直接将单智能体算法用于多智能体环境。通信学习则假设智能体之间存在通讯与信息交互，并在训练过程中学习通讯相关的决策，即是否进行通讯、与哪些智能体进行通讯等。协作学习与通信学习具有类似的地方但在具体实现方式上存在不同。智能体建模这一类方法则主要聚焦于通过对其他智能体的策略、目标、类别等等建模来进行更好的协作或者更快地打败竞争对手。

目前学术界在 MARL 目标追踪问题上的研究进展有：Huy Xuan Pham[1]等人提出将多智能体方法引入无人机区域覆盖问题，并给出了解决该问题的实例。在无人机避障问题中，Hung Manh La[2]等人采用多智能体强化学习的方法，通过协作有效避敌，并给出了通信受限的情况下的避障方法，取得了较好成果。但是现有的多智能体强化学习方法仅能有效应用于小规模无人机集群，尚不能直接应用于超大规模的无人机集群。文献[3]给出了一个针对大规模智能体的工具，文献[4]则给出了交通领域的大规模仿真工具，但无人机领域尚未找到相关研究。

（三）常见实证研究基准测试集

OpenAI Gym 是一款用于研发和比较强化学习算法的工具包，它支持训练智能体（agent）用以解决各种问题。同时，它也提供了提供了多种多样的环境，从简单到困难，并涉及到许多不同类型的数据，使得使用者可以轻松地测试各类强化学习算法的实际效果。但是我们所研究的问题难以在此环境中测试，因为 gym 虽然具有较强的泛用性，但并不面向特定的追踪问题，因此我们需要构建特定的基于多智能体强化学习目标追踪的基准测试框架。

三、基于 MARL 的无人机群目标追踪算法训练测试框架和实验平台

（一）算法训练测试框架

针对我们要研究的无人机追踪车辆目标问题，我们提出了一套基于 MARL 的无人机群目标追踪算法训练测试框架，框架设计图如下所示：

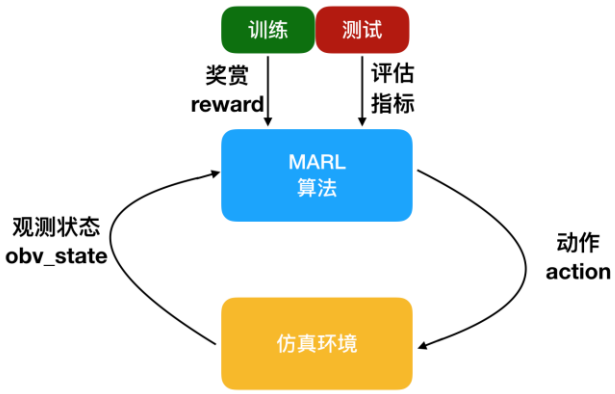

图 1　　算法训练测试框架设计图

在强化学习模型中，包含智能体和环境两个实体，其中智能体将利用强化学习算法进行决策以与环境交互，智能体即对应着上图中蓝色框的 MARL 算法。强化学习算法需要一个环境用于收集数据和训练模型，针对在真实物理世界进行实验的成本高、效率低、可观测性差的弱点，我们搭建了低成本、高效、稳定可靠的模拟仿真环境来支持我们测试算法性能，即对应上图中黄色框的仿真环境。在强化学习模型中，智能体可以观测到环境的状态，即上图中仿真环境传递给 MARL 算法的观测状态，智能体相应地选择动作并将动作信息传递给环境。该动作会对环境造成一定影响，使环境状态发生改变。该影响的好坏由事先定义的奖赏机制衡量，在我们框架中的表现就是训练模块根据智能体在环境中执行当前动作前的状态以及智能体执行当前动作使得环境转移到下一个状态这些信息而生成的奖赏函数，计算奖赏值反馈给智能体。在我们研究的目标追踪问题场景中，观测状态将包括无人机的位置，速度，无人机已知的地面车辆的位置和速度。智能体（无人机）执行的动作将指明下一时刻无人机的飞行速度和方向。综上，强化学习模型由四个核心部分组成：智能体当前在环境中所处状态 S，智能体由当前状态根据强化学习算法决策出的动作 A，在环境中智能体执行动作 A 后转移到的下一个状态 S_，智能体在状态 S 执行动作 A 后所获奖赏 R。这里的奖赏 R 是即时奖赏，用来评估当前执行动作对环境影响的好坏，而根据一个算法策略产生的状态-动作序列完整执行一定步数后能得到的期望奖赏被称为累积奖赏，该奖赏将用于评价算法策略的优劣。对于每一种强化学习算法，其核心目的都是期望学习到一个策略，使得智能体根据此策略决策，可以获得尽可能大的的期望累积奖赏。在对于每一种强化学习算法进行训练到一定阶段后，我们将利用测试模块以及评估指标对算法的性能进行测试，以验证算法在目标追踪问题上的有效性和表现优劣。

（二）算法表现评判指标

（1）算法效率评判：

K step-reward：总共训练 K 个 train step，每隔一段训练时间间隔计算一次训练奖赏均值并测试算法的表现，分别获得训练奖赏和测试奖赏，然后以 train step 为横坐标，训练奖赏和测试奖赏为纵坐标，获得训练、测试两条时间序列。

单智能体扩展途径-reward：对每种单智能体强化学习算法，存在不同的途径来将他们扩展到多智能体场景下使用，对于同一个多智能体算法，测试其在不同扩展途径下奖赏的变化曲线，测试算法在不同拓展途径下的表现；

（2）算法可扩展性评判：

智能体数量-reward：对于同一个多智能体算法，测试其在不同无人机、小车数目场景下奖赏的变化曲线，测试其可扩展性。

（三）算法训练测试框架具体模块设计

（1）数值仿真环境模块设计

设计思想：首先我们对目标追踪问题的场景进行了一些假设和简化，我们使用 x，y，z 三轴坐标来表

示一个小车或无人机的位置，小车的 z 轴坐标恒为 0，不同无人机处于不同高度飞行且不会改变飞行高度，故每架无人机的 z 轴坐标也恒为一个固定值，因此我们将 z 轴坐标去掉，仅使用 x，y 两轴坐标来表示小车或无人机的位置，设置小车或无人机的动作空间都是一个 2 维的速度向量，分别代表下一时刻小车或无人机的速度，为了简化计算过程，我们假设无人机和小车均有无穷大的加速度，速度变化可以立即完成，忽略环境中的阻力信息。同时我们假设活动范围没有边界限制，当小车和无人机之间距离小于一定值时认为小车被“追踪”到了，以该固定值为半径，某架无人机的坐标为圆心的圆我们称之为该无人机的“可视范围”，只有小车处于某辆无人机的可视范围内时它们可以互相获得对方位置、速度信息，否则无法获得相关信息。然后我们设定无人机之间以及小车之间都可以无限制通信，也就是它们可以共享彼此获得的位置、速度信息。同时为了使得追踪成为可能，我们假定小车最高速度是无人机最高速度的 80%。此外，我们将目标追踪问题分成了如下两个阶段：

3.1.1 第一阶段：在第一阶段无人机要先“找”到小车，也就是在没有任何信息的情况下，在给定一片区域内不断地搜寻小车，在第一次搜寻到小车后，进入第二阶段。

3.1.2 第二阶段：第一次搜寻到小车之后，无人机开始拥有小车的位置、速度信息，从而可以根据这些信息进行决策，即使无人机丢失目标了，同样也拥有目标丢失前小车的位置、速度信息，并凭此进行决策追踪。

第一阶段过程我们认为在没有任何信息的情况下，使用区域扫描算法或者其他传统算法可能比使用强化学习技术更合适，因此我们主要针对第二个阶段展开研究。所以我们的模拟环境的初始状态的设置是，小车会随机出现在无人机可视范围的边缘位置，以模拟无人机第一次搜寻到小车场景。若环境中无人机数量为 UAV_COUNT，小车数量为 CAR_COUNT，那么：

3.1.2.1 状态设置：状态 S 是一个 4×UAV_COUNT+5×CAR_COUNT 维的向量，包含了无人机以及小车的位置速度信息。以 2 车 2 机为例，即为

<uav0_x, uav0_y, uav1_x, uav1_y,
uav0_vx, uav0_vy, uav0_vx, uav0_vy,
car0_x, car0_y, lost_count0, car1_x, car1_y, lost_count1
car0_vx, car0_vy, car0_vx, car0_vy>

uav0_x 代表 0 号无人机的 x 坐标，uav0_vx 代表 0 号无人机在 x 方向的速度，以此类推，lost_count 代表了小车速度位置信息的时效性，若为 n 则代表无人机看到的小车速度位置信息是在 n 个时间单位前的信息，即小车已经逃脱 n 个时间单位了，通俗地讲，状态向量即为<无人机位置，无人机状态，小车位置+lost_count 信息，小车速度>；

3.1.2.2 初始状态设置：无人机的初始位置通过硬编码固定在环境的实现中，小车的位置则随机生成在无人机可视范围的边缘，每辆无人机可视范围内不多于 1 辆小车；

3.1.2.3 动作设置：环境接收的动作是一个 2×UAV_COUNT 维的向量，分别代表各个无人机在下一个时间单位的速度；

3.1.2.4 状态转移函数：按照 新位置 = 旧位置 + 速度 的公式更新所有智能体的位置信息，无人机速度根据传入的动作信息更新，小车的速度根据环境内置的策略更新。若小车处于可视范围外，则不更新位置速度信息，仅将对应的 lost_count 信息+1。

3.1.2.5 奖赏设置：每有一辆小车处于可视范围内，则奖赏+1。

3.1.2.6 结束条件：超过 200 个时间单位或者小车全部逃脱 10 个时间单位以上。

3.1.2.7 小车策略：目前小车策略有两个，内置在环境的实现中，可以通过参数来指定需要使用的小车策略：

random：小车随机游走策略，用于测试强化学习算法时间效率；

adversary：小车对抗策略，用于测试强化学习算法在多智能体场景下的性能表现。对抗策略的实现分如下 3 种情况（目前仅支持 2v2 场景）：

1）若两辆小车都处于可视范围内：对每辆小车，分别计算离各个无人机间的距离，计算这些距离的倒数，归一化后作为各个无人机位置的权重，以此计算出无人机位置的中心，小车往远离这个中心位置的方向以最大速度逃离；

2）只有一辆小车处于可视范围内：对处于可视范围内的小车，以最大速度往远离靠自己最近的无人机的方向逃离；对处于可视范围外的小车，首先构造一个虚拟的无人机位置，即认为有一架无人机处于自己当前速度反方向，距离为可视范围半径的位置，然后根据 1）中的方式计算逃离的方向和速度；

3）两辆小车都逃离成功：以 90%的概率维持原速度行驶，10%的概率随机挑选一个方向以最大速度行驶。

（2）强化学习算法模块、算法训练与测试模块设计

系统中的所有模块：

3.2.1 numeric_env 模块：模拟环境模块，负责提供与智能体进行交互的环境接口；

3.2.2 agent 模块：智能体模块，包含了各种算法更新网络参数部分的实现，以及其需要的各种组件的实现，包含以下三个模块：

1）common 模块：提供了每个算法都需要的通用类，包含下列两个模块：

1. args 模块：将各个算法的参数部分抽象成单独的参数类；

2. normalizer 模块：在算法训练前对环境进行随机尝试，初步确定环境状态分布的均值和方差，为算法后续获得的状态做标准化；

2）deepq 模块：基于值函数的多种深度强化学习算法更新部分的实现，包括：

1. DQN 算法；

2. Double DQN 算法；

3. Dueling DQN 算法；

3）pb 模块：基于策略函数的多种深度强化学习算法更新部分的实现，包括：

1. Advantage Actor Critic 算法；

2. DDPG 算法；

4）maddpgagent 模块：负责实现 maddpg 算法的更新部分；

3.2.3 trainer 模块：负责训练智能体，包括了初始化 normalizer、设置各个智能体的网络模型、和环境进行交互并调用 agent 的更新接口更新网络参数、记录训练和测试的 K step-reward 指标等等功能，有如下三个 trainer 类：

1）CentralizedTrainer 类：将多智能体强化学习任务视作单智能体强化学习任务，也就是将多个智能体的动作空间做笛卡尔积得到集中式智能体的动作空间，然后采用单智能体训练的模式训练；

2）DistributedTrainer 类：将多智能体强化学习任务中的每个智能体的任务视作单智能体强化学习任务，然后分布式训练每个智能体；

3）MADDPGTrainer 类：负责 maddpg 多智能体算法的训练；

（3）算法评判指标计算

3.3.1 算法效率评判指标计算：

3.3.1.1 K step-reward：总共训练 1,000,000 个 train step，每隔 10 个时间单位计算一次最近 10 个时间单位训练奖赏的均值，然后进行 20 个 episode 的算法测试并取均值，获得训练奖赏和测试奖赏。

3.3.1.2 单智能体扩展途径-reward：对每种单智能体强化学习算法，按如下两种途径将其扩展为多智能体强化学习算法：

1）集中式（centralized）：将多智能体强化学习任务视作单智能体强化学习任务，也就是将多个智能体的动作空间做笛卡尔积得到集中式智能体的动作空间，然后采用单智能体训练的模式训练；

2）分布式（distributed）：将多智能体强化学习任务中的每个智能体的任务视作单智能体强化学习任务，然后分布式训练每个智能体，分布式的训练又分以下两种方式：

1. 进行参数共享（parameter share）：即所有智能体共用一套网络参数；
2. 不进行参数共享：每个智能体拥有自己的网络参数；

按照（1）中的方式分别计算不同扩展途径下算法的 K step-reward 并进行比较。

3.3.2 算法可扩展性评判：

智能体数量-reward：对于同一个多智能体算法，测试其在 2v2（2 架无人机 2 辆小车）和 5v5（5 架无人机和 5 辆小车）场景下奖赏的变化曲线，测试其可扩展性。

四、算法实现与实验设计

（一）算法框架对应的实际算法原理与实现

4.1.1 Deep Q-Network（DQN）

Mnih 等学者提出了基于 Deep Q-Network（DQN）的强化学习方法[5][6]，该方法基于 Q-Learning 算法训练得到卷积神经网络，通过随机梯度下降更新，并使用经验池回放机制应对数据相关性与非平稳分布，在多种 Atari 游戏上取得了优于人类玩家的表现。

DQN 算法是一个免模型的异策略学习算法，处理马尔可夫决策过程（Markov Decision Process，MDP），采用 ε-贪心探索策略，与 γ 折扣累计奖赏。假设一个 MDP 在第 T 个时刻终止，令 t 时刻仿真环境返回的奖赏为 r_t，则该时刻的长期累计奖赏 R_t 记为：

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{1}$$

最优状态-动作值函数为：

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') \middle| s,a \right] \tag{2}$$

其中 s 是时刻 t 的状态，a 是时刻 t 采取的动作，π 是算法采用的策略。

在 DQN 的神经网络中，定义损失函数为：

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot);s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i) \right) \nabla_{\theta_i} Q(s,a;\theta_i) \right] \tag{3}$$

其中 θ_i 为第 i 轮训练中神经网络的阈值，状态 s 下动作 a 根据概率 ρ(s,a)分布。

整个算法逻辑如下所示：

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 2　　DQN 算法流程

在算法中，数据集 $D=e_1,...,e_N$ 存储了被放入经验池的序列，$e_t=(s_t,a_t,r_t,s_{t+1})$。可以看到在算法的内部循环中进行批量梯度下降时，算法随机从经验池中选取样本，使得过去状态中的动作行为分布较为平均，使得 DQN 成功地避免了参数的振荡或发散。

4.1.2 Double DQN

DQN 算法中的 maximize 操作容易造成对 Q 值的过度估计，对此，Double DQN 的解决方法是：将动作的选择和评估解耦，使用当前 Q 网络选择当前状态下最大 Q 值的动作，使用目标 Q 网络评估该动作的 Q 值，也就是 Q 网络的更新目标计算公式变为

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax} Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-) \tag{4}$$

其余步骤和 DQN 一致。经验证，Double DQN 在超过 60%的 Atari 小游戏上的表现要优于 DQN 算法[7]。

4.1.3 Dueling DQN

Dueling DQN 尝试通过优化神经网络的结构来优化算法。Dueling DQN 将 Q 网络分成两个部分，其中一个部分只拟合 V(s)，另一个部分拟合 A(s, a)，A(s, a)是优势函数，它代表了在 s 这个状态下采取 a 这个动作相比于 V(s)能多出的累计奖赏差值，即

$$A(s,a) = Q(s,a) - V(s) \tag{5}$$

然后将这两个部分相加得到最后的 Q（s, a）。那么最终的 Q(s, a)可表示为：

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \tag{6}$$

其中，θ 为共有参数，α 和 β 分别为 A 函数和 V 函数的特有参数。算法的其它流程与 DDQN 一致。经验证，Dueling DQN 在 Atari 小游戏上的表现要显著优于 DQN 算法[8]。

4.1.4 Actor-Critic

在 Actor-critic 算法出现之前，大部分的强化学习算法都趋于两个极端：仅仅基于策略或是仅仅对值函数进行拟合，前者的梯度估量值可能跨度过大，后者未必能保证算法拟合最优的策略。而 Actor-critic 算法融合了基于策略的学习算法与基于值的学习算法的优势，搭建了 Critic 网络来学习值函数，以更新 Actor 网络中策略的参数，确保朝着优化的方向学习，最后达到优化表现的目的。由于它是基于梯度进行优化的，因此相较于单独的 Critic 网络能够进行更加快速的收敛。Actor-critic 网络能广泛应用于拥有抽象状态动作

空间的场景。

在 Actor-critic 算法[9]中，尽管策略将会随着 Actor 网络参数更新更新，但是考虑到 Actor 网络参数将会以一个较慢的速度更新，因此并不会对学习结果产生不良影响。

在 Critic 网络中，将采用时序差分（Temporal difference，TD）方式进行学习，令 r_k，z_k，λ_k 为其参数，θ_k 为 Actor 网络的参数，在时刻 k，令(X_k,U_k)为这一时刻的状态动作对，而 X_{k+1} 为实施动作后的新状态，则 Critic 网络的参数更新过程如下：

$$\lambda_{k+1} = \lambda_k + \gamma_k(g(X_k, U_k) - \lambda_k), \tag{7}$$

$$r_{k+1} = r_k + \gamma_k\Big(g(X_k, U_k) - \lambda_k + Q_{r_k}^{\theta_k}(X_{k+1}, U_{k+1}) - Q_{r_k}^{\theta_k}(X_k, U_k)\Big)z_k. \tag{8}$$

而参数 z_k 可选择以下两种方式之一进行更新：

$$\begin{aligned} z_{k+1} &= z_k + \phi_{\theta_k}(X_{k+1}, U_{k+1}), \qquad \text{if } X_{k+1} \neq x^*, \\ &= \phi_{\theta_k}(X_{k+1}, U_{k+1}), \qquad \text{otherwise.} \end{aligned} \tag{9}$$

$$z_{k+1} = \alpha z_k + \phi_{\theta_k}(X_{k+1}, U_{k+1}). \tag{10}$$

其中 x^*在状态空间中，$\alpha\in[0,1)$。

对 Actor 网络，将根据如下公式更新参数：

$$\theta_{k+1} = \theta_k - \beta_k\Gamma(r_k)Q_{r_k}^{\theta_k}(X_{k+1}, U_{k+1})\psi_{\theta_k}(X_{k+1}, U_{k+1}). \tag{11}$$

4.1.5 Deep DPG（DDPG）

为了应对高维状态空间与连续动作空间，Deep DPG（DDPG）算法基于 DPG 进行了改进。DDPG 是一个免模型的异策略 Actor-critic 学习算法，同时借鉴了 DQN 的思想，在 cartpole 摇摆等机械控制场景中表现良好，能直接从低维空间中习得好的策略，并保持超参数和网络结构不变[10]。

在 DPG 算法中，Actor 网络根据分布 J 进行更新：

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)}\nabla_{\theta_\mu}\mu(s|\theta^\mu)|_{s=s_t}\right] \end{aligned} \tag{12}$$

DDPG 在此基础上引入了 DQN 的经验池回放机制并进行批量梯度下降，使得网络中的参数能够在较大的状态动作空间中学习，令 Actor 网络和 Critic 网络分别为 $\mu(s|\theta^\mu)$和 $Q(s,a|\theta^Q)$，并维护一份它们的拷贝 $\mu'(s|\theta^{\mu'})$和 $Q'(s,a|\theta^{Q'})$。为了学习过程的稳定性，令参数的更新机制如下：

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \qquad (\tau \ll 1) \tag{13}$$

同时，DDPG 在 Actor 网络中引入了噪音 N，使得它能够独立于学习算法进行探索：

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N} \tag{14}$$

最后，整个算法逻辑如下：

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 3　　DDPG 算法逻辑

对于上述单智能体算法，我们可以通过两种途径将其扩展为多智能体算法。第一种是 centralized 方法，将多智能体强化学习任务视作单智能体强化学习任务，也就是将多个智能体的动作空间做笛卡尔积得到集中式智能体的动作空间，然后采用单智能体训练的模式训练；第二种是 distributed 方法，将多智能体强化学习任务中的每个智能体的任务视作单智能体强化学习任务，然后分布式训练每个智能体。以 distributed 方法进行训练又包含两种不同方式：参数共享式（parameter share）distributed 以及非参数共享式 distributed。参数共享式 distributed 将使不同的智能体使用同一套网络参数进行训练，非参数共享式 distributed 将使不同的智能体各自使用自己的网络参数进行训练。

4.1.6 通用的深度强化学习算法训练框架实现

通过观察发现，上述算法存在一个通用的模式：

循环训练 M 轮：

在一轮训练的每一步：

1.根据智能体的探索策略生成动作；

2.与环境交互获得一条 transaction；

3.智能体根据这条 transaction 按照各自算法更新网络参数；

在上述过程中，仅有生成探索动作和更新参数部分是每个算法有所不同的，其余部分每个算法都是通用的，因此将生成探索动作和更新参数两部分作为 Agent 类的接口，完整的训练步骤在 Trainer 模块中实现，Trainer 通过调用 Agent 对象提供的生成探索动作和更新参数接口来实现完整的训练过程，从而实现了训练过程实现和算法实现的解耦。完整的 Agent 类接口定义如下：

```python
class Agent:

    networks = None

    # 生成最优动作
    def choose_action(self, state):
        raise NotImplementedError

    # 根据探索策略生成动作
    def choose_action_with_exploration(self, state):
        raise NotImplementedError

    # 根据transaction更新网络参数
    def train_one_step(self, s, a, s_, r, d):
        raise NotImplementedError

    # 设置网络模型
    def set_model(self, parameter_share=False, load_paths=None):
        raise NotImplementedError

    # 保存网络模型
    def save_model(self, paths):
        raise NotImplementedError
```

图 4　　完整的 Agent 类接口定义代码

此外，为了实现参数共享，在 Agent 类里定义了一个所有 Agent 对象共享的静态列表以及 set_module 接口，静态列表里负责存储共享的网络模型。当 set_module 接口传入的 parameter_share 参数设置为 True 时，则使用静态列表中的网络进行训练，否则使用自定义的网络进行训练。

那么，对于 CentralizedTrainer，其实和单智能体强化学习训练过程一致，一轮完整的训练代码如下：

```python
def _train_one_episode(self):
    # 重置环境，获得初始状态并使用normalizer进行标准化
    s = self._norm.transform(self._env.reset())
    d = False
    total_reward = 0
    while not d:  # 当未进入终止状态时
        # 调用Agent类提供接口获得智能体的探索动作
        a = self._agent.choose_action_with_exploration(s)
        # 与环境交互获得一条transaction
        s_, r, d, i = self._env.step(a)
        s_ = self._norm.transform(s_)
        # 调用Agent类提供接口更新智能体网络参数
        self._agent.train_one_step(s, a, s_, r, d)
        s = s_
        total_reward += r
        self._train_step += 1
    self._episode_count += 1
    return total_reward
```

图 5　　CentralizedTrainer 一轮完整的训练代码

对于 DistributedTrainer，由于引入了参数共享，并涉及到奖赏分配问题，那么就具备以下额外的元素：

（1）　状态变换函数：对于参数共享的分布式训练而言，由于多个智能体使用同一套网络参数，若不对环境返回的状态进行一些变换，就会导致在同一个状态下其值函数不固定的问题，无法很好的完成目标。设环境返回的状态为 s（已标准化），我们规定每架无人机获得的状态是向量<s，i>，i 为该无人机的编号。

（2）　奖赏分配函数：奖赏分配问题是多智能体强化学习的一个热门研究领域，在此我们不对环境返回的奖赏作变化，直接作为每架无人机的奖赏。

对于上述两个函数，我们将其作为 DistributedTrainer 初始化函数的参数，由使用者自行定义，以适

应不同目的的算法研究。一轮完整的训练代码如下：

```
def _train_one_episode(self):
    s = self._norm.transform(self._env.reset())
    d = False
    total_reward = 0
    while not d:
        # 调用Agent类提供接口获得智能体的探索动作并拼接
        a = self._choose_action_with_exploration(s)
        # 与环境交互获得一条transaction
        s_, r, d, i = self._env.step(a.reshape(-1))
        s_ = self._norm.transform(s_)
        # 对每个agent，调用Agent类提供接口更新智能体网络参数
        for i, agent in enumerate(self._agents):
            # self._st、self.rd分别为状态转换函数、奖赏分配函数
            agent.train_one_step(self._st(s, i), a[i],  self._st(s_, i), self._rd(s_, r, i), d)
        s = s_
        total_reward += r

        self._train_step += 1
    self._episode_count += 1
    return total_reward
```

图 6　　DistributedTrainer 一轮完整的训练代码

一个典型的训练智能体代码如下（以 DQN 算法为例）：

```
# 创建训练环境
env = MultiEnv(2, 2, car_policy='random')
# 设定算法参数
args = DDPGArgs(state_dim=env.STATE_DIM,
                action_dim=2,
                action_scale=1.0,
                final_action_scale=0.1,
                scale_decay_factor=0.9999,
                actor_lr=1e-4,
                critic_lr=1e-3,
                buffer_size=1000000,
                update_freq=1,
                update_count=1,
                batch_size=50,
                steps_before_training=1000,
                gamma=0.95
                )

# 创建需要训练的智能体
agents = [DDPGAgent(args) for _ in range(2)]
# 创建Trainer，将智能体列表和环境作为参数
trainer = DistributedTrainer(agents,
                             env,
                             parameter_share=False,
                             log_dir='../logs/debug',
                             )

# 训练1,000,000步
trainer.train(1000000)
```

图 7　　DQN 训练智能体代码

4.1.7 MADDPG

多智能体深度确定性策略梯度算法（Multi Agent Deep Deterministic Policy Gradient，MADDPG）是人工智能非营利组织 OpenAI 于 2017 年发表在 NIPS 上的文章[11]中提出的一种多智能体强化学习算法，主

要介绍了利用 MADDPG 算法解决传统的单智能体强化学习算法应用到多智能体环境中产生的问题。传统单智能体算法（例如基于值函数的强化学习方法 DQN，基于策略梯度的强化学习方法 DDPG）应用到多智能体环境中时，由于每个智能体都是在不断学习改进其策略，因此从每一个智能体的角度看，环境是一个动态不稳定的，不符合传统强化学习收敛条件，在一定程度上，无法通过仅仅改变智能体自身的策略来适应动态不稳定的环境。MADDPG 算法的核心是对 DDPG 算法为适应多智能体环境进行改进，最核心的部分就是每个智能体的 critic 部分能够获取其余所有智能体的动作信息。MADDPG 的核心思想包括以下三点：

（1）集中式训练，分布式执行：训练时采用集中式学习训练 critic 与 actor，使用时 actor 只用知道智能体自身信息就能运行，critic 需要其他智能体的策略信息。

（2）为了能够适用于动态环境，改进了经验回放记录的数据，包括每一个智能体观测的状态。

（3）对于每一个智能体的策略进行改进时，利用所有策略的整体效果进行优化，以提高算法的稳定性。

MADDPG 算法本质上是一个确定性策略梯度算法，针对每个智能体训练一个需要全局信息的 critic 以及一个需要局部信息的 actor，允许每个智能体有自己的奖赏函数，既可以用于合作任务也可以用于对抗任务。下面介绍 MADDPG 算法内容：

设有 N 个智能体，每一个智能体有自己独立的 actor，target actor，critic，target critic 网络。假设算法将训练 M 个 episode，那么在每一个 episode 中，环境先为每个智能体初始化一个状态，之后算法将与环境进行连续的交互，一共交互 max-episode-length 步。在每一步中，对于每一个智能体 i，将根据其当前自身观测的局部状态由自己的 actor 网络生成一个动作，这个动作将加上一个随机的噪声值作为算法训练中的探索，该噪声将随着训练步数的增加而衰减，直到一个最低值。

$$a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t \tag{15}$$

上面公式中a_i即智能体将要传递给环境的动作，o_i是智能体当前观测的自身局部状态，μ_{θ_i}是智能体 i 的 actor 网络，θ_i是 actor 神经网络的参数，N_t是噪声函数。

当 N 个智能体的动作（a_1，a_2，……，a_N）全部计算完成后，将这些动作放入环境中执行，环境将给出奖赏 r，以及环境转移到的下一个全局状态 x'。之后将当前全局状态 x，智能体根据当前状态执行的动作 a，环境将给出奖赏 r，以及环境转移到的下一个全局状态 x'组成的四元组（x, a, r, x'）存入经验回放池（replay buffer）D。之后对于 N 个智能体进行网络参数的更新。对于智能体 i 的网络参数更新如下：首先从经验回放池 D 中抽取一小批（S 个）样本，对于这 S 个抽样样本中的每个样本：

$$(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j) \tag{16}$$

计算：

$$y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k=\boldsymbol{\mu}'_k(o_k^j)} \tag{17}$$

其中$Q_i^{\mu'}$是智能体 i 的 target critic 网络，其中的参数包括抽样样本中的下一个全局状态，N 个智能体每一个智能体依据自己的 target actor 网络根据抽样样本中记录的下一个自身局部状态（从 x'中计算得出）生成的下一个动作 a'。智能体 i 的 target critic 网络的输出乘以一个衰减因子γ，r_i^j是当前抽样样本奖赏 r 中计算出的当前智能体 i 的奖赏值。之后当这 S 个抽样样本中的每个样本的 y 值计算完成后，计算 y 值和该智能体 i 的 critic 网络在给定输入来自抽样样本中的当前全局状态 x 和来自抽样样本中的所有智能体执行动作的输出的均方误差：

$$\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2 \tag{18}$$

上面的函数就是每个智能体 critic 的损失函数，通过最小化这个损失函数（通过梯度下降）来更新每个智能体的 critic 网络。每个智能体的 critic 网络即表示 Q 函数，其输入是全局状态信息 x 以及 N 个智能体的动作 a1~aN，函数返回该智能体当前全局状态和动作下的 Q 值。可以看到，智能体的 critic 使用了需

要其他智能体的策略信息。

之后更新智能体 i 的 actor 网络。每个智能体 i 的 actor 的目标函数是：

$$E[Q_i^{\pi}(x, a_1, a_2, \cdots, a_N)] \tag{19}$$

更新的目标是最大化上面的目标函数，方法是利用梯度上升，即利用采样的策略梯度：

$$\nabla_{\theta_i} J \approx \frac{1}{S} \sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i=\boldsymbol{\mu}_i(o_i^j)} \tag{20}$$

进行 actor 网络的更新。其中上面公式中智能体 critic 网络的输入参数有抽样样本中记录的当前全局状态 x，输入动作参数中除了当前更新的智能体 i 的动作需要由智能体 i 的 actor 网络根据抽样样本中记录的当前自身局部状态计算，其余智能体的动作全部来自抽样样本，因此 actor 网络只用知道智能体自身信息。

在每一个智能体的 actor 和 critic 网络更新结束后，利用软更新方法对所有智能体的 target actor 网络和 target critic 网络进行更新：

$$\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i' \tag{21}$$

θ_i'是 target 网络的参数，θ_i是当前网络的参数。target 神经网络的参数即来自当前神经网络的参数，τ是权重因子。

MADDPG 的一个启发就是，如果我们知道所有的智能体的动作，那么环境就是稳定的，就算策略在不断更新，环境也是稳定的，因为模型动力学是稳定的：

$$P(s'|s, a_1, \cdots, a_n, \pi_1, \cdots, \pi_n) = P(s'|s, a_1, \cdots, a_n) = P(s'|s, a_1, \cdots, a_n, \pi_1', \cdots, \pi_n') \tag{22}$$

MADDPG 算法在算法训练测试框架中的实现：maddpg_agent 类将实现 MADDPG 算法，其核心是实现抽象类 Agent 中的三个对外（对 trainer）的接口函数。设定一个 maddpg_agent 对象中将包含所有的智能体，在这个类的对象中进行每一个智能体的训练与决策。Args_maddpg 类将包含 MADDPG 算法需要的一些核心信息参数与深度强化学习算法训练所需的超参数，包括由训练测试入口传入的无人机数量，车辆数量，网络更新频率，批量采样采样个数，衰减因子，actor 学习率，critic 学习率等，以及 trainer 传入的全局状态维度，自身局部状态维度，动作维度等。这些参数将在 Maddpg_agent 类初始化的时候直接传递给 maddpg_agent 类。Experience_Pool_maddpg 类实现经验回放池功能，将对 maddpg_agent 类提供两个接口：add 和 sample。add 将接受四元组（x, a, r, x'）并将其放入经验池，sample 将接受采样个数参数（即要选择几个样本作为本次采样输出），输出从经验回放池中随机采样后的样本集合。Maddpg_trainer 是训练与测试 MADDPG 算法的类，其中主要的 train 函数将训练 MADDPG 算法 1,000,000 步（step），并且每训练 10 步进行 20 轮测试，记录当前测试结束后训练的步数 step 以及 20 轮测试均值 mean_reward 形成二元组（step, mean_reward）记录下来，作为算法性能评估源数据。

在 Maddpg_trainer 中的训练函数 train 将循环调用类内函数 train_one_episode，train_one_episode 将训练算法跑完一轮（一个 episode），跑完一轮的标志即仿真环境返回参数 done 为 True。这里的虚拟环境使用的是与前面所述算法完全相同的虚拟环境。train_one_episode 具体实现如下图：

```
def train_one_episode(self):
    state = self.env.reset()  # 初始化智能体状态
    done = False  # 训练一轮结束标志
    total_reward = 0  # 训练一轮的累积奖赏
    while not done:  # 一轮未结束
        action = self.agent.choose_action_with_exploration(self.normalizer.transform(state), self.train_step)  # 调用
        # 智能体算法agent的接口choose_action_with_exploration,
        # 传给agent的状态是归一化后的，train_step用于智能体算法中的噪声函数的输入参数
        state_, reward, done, _ = self.env.step(action * self.action_bound)  # 调用仿真环境env的接口step
        self.agent.train_one_step(
            self.normalizer.transform(state),
            action,
            reward,
            self.normalizer.transform(state_),
            done)  # 调用智能体算法agent的接口train_one_step，传给agent的状态是归一化后的
        state = state_  # 当前状态转为下一个状态
        total_reward += reward  # 计算累积奖赏
        self.train_step += 1
    self.episode_count += 1
    return total_reward
```

图 8　　train_one_episode 具体实现代码

训练函数 train 中将判断 train_step 是否超过 1000000，如果超过则停止训练，程序运行结束。否则将调用 train_one_episode 进行一轮训练并且得到本轮训练奖赏。每经过 10 轮训练，将调用 test 进行算法测试，test 中将调用 X 次 test_one_episode，即对已经训练一定轮数的算法测试 X 轮，取 X 轮奖赏均值返回给 train。X 参数由训练测试入口传入。test_one_episode 结构与 train_one_episode 类似，其中的 agent 动作选择将调用 choose_action 函数，即不需要像训练算法时那样需要添加探索环境所需的动作噪声。此外不需要调用 agent 的 train_one_step，即不需要去更新 agent 策略，train_step 也不需要增加。

对于实现 MADDPG 算法的 Maddpg_agent 类，初始化时将获取 Args_maddpg 类中所有的参数信息，之后创建 critic，target critic，actor，target actor 神经网络。每一种网络都是一个 python 列表，第 i 个表示第 i 个智能体的对应网络。在我们的研究场景中，设定 MADDPG 算法中每一个智能体可以与其他所有智能体交流信息。在该设定场景下，对于虚拟仿真环境返回的状态（以两架无人机追踪两辆车为例）为：

$(uav0\_x,\ uav0\_y,\ uav1\_x,\ uav1\_y,\ uav0\_vx,\ uav0\_vy,\ uav1\_vx,\ uav1\_vy,\ car0\_x,$
$car0\_y, lost\_count0,\ car1\_x,\ car1\_y, lost\_count1,\ car0\_vx,\ car0\_vy,\ car1\_vx,\ car1\_vy)$

这个状态将作为全局状态存入经验回放池，对于每一架无人机的局部观测状态（即作为 actor 网络和 target 网络的输入），由于设定了智能体可以与其他所有智能体交流信息，因此局部观测状态为：（自己的位置，自己的速度，其他无人机的位置，其他无人机的速度，小车的状态），对应到上述举例的全局状态，无人机 0 的局部观测状态为：

$(uav0_x,\ uav0_y,\ uav0_{vx},\ uav0_{vy},\ uav1\_x,\ uav1\_y,\ uav1\_vx,\ uav1\_vy,\ car0\_x,\ car0\_y, lost\_count0,$
$car1\_x,\ car1\_y, lost\_count1,\ car0\_vx,\ car0\_vy,\ car1\_vx,\ car1\_vy)$

即每一个智能体的局部观测状态是全局状态中一些数据位置进行调换，局部观测状态维度与全局状态相同。创建所有智能体的 actor 网络的代码如下：

self.agents_actor_network = [ActorNet(self.args.state_dim, self.args.action_dim // self.args.agent_num) for i in range(self.args.agent_num)]

agent_num 是智能体（无人机）数量。state_dim 是局部状态维度（与全局相同），action_dim 是所有智能体动作维度，例如两架无人机就是 4 维，需要除以智能体数量计算出自身动作维度。actor 网络和 critic 网络更新利用 Adam（自适应矩估计）优化方法，actor 网络优化器设定实现代码如下：

self.agents_actor_optim=[optim.Adam(self.agents_actor_network[i].parameters(), lr=self.args.actor_lr) for i in range(self.args.agent_num)]

critic 网络的损失函数设定为 torch.nn.MSELoss()，即均方误差。

每个智能体 actor 和 critic 神经网络结构设计如下：

```python
class ActorNet(nn.Module):

    def __init__(self, state_dim, action_dim):
        super().__init__()
        self._fc0 = nn.Linear(state_dim, 128)
        self._fc1 = nn.Linear(128, 64)
        self._fc2 = nn.Linear(64, action_dim)

    def forward(self, x):
        x = F.relu(self._fc0(x))
        x = F.relu(self._fc1(x))
        x = self._fc2(x)
        return F.tanh(x)


class CriticNet(nn.Module):

    def __init__(self, state_dim, action_dim):
        super().__init__()
        self._fc0 = nn.Linear(state_dim + action_dim, 256)
        self._fc1 = nn.Linear(256, 128)
        self._fc2 = nn.Linear(128, 1)

    def forward(self, x):
        x = F.relu(self._fc0(x))
        x = F.relu(self._fc1(x))
        x = self._fc2(x)
        return x
```

图 9　　每个智能体 actor 和 critic 神经网络实现代码

对于 trainer 中调用 agent 提供的 choose_action_with_exploration，其中首先调用 choose_action 依据当前状态由每个智能体的 actor 网络生成动作，之后在 actor 网络生成动作的基础上增加一个噪声用以对环境进行探索，噪声是一个从均值为 0 的正态分布抽样的值，该正态分布的方差将随着算法训练步数的增加逐渐衰减，从初值 1.0 衰减至最低 0.1。

对于 actor 和 critic 网络的更新，对于每个智能体，将调用 maddpg_agent 中的_update 函数，函数参数是智能体的序号，指明更新第几个智能体。该函数中先调用经验回放池的 sample 进行抽样，之后计算

$$\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2 \tag{23}$$

计算好 y 值后，利用代码：

critic_loss = self.agents_critic_lossfn[index](critic_output, y.float())

实现计算损失函数，其中 critic_output 就是当前 critic 网络输出，y 是 target critic 网络的输出。之后利用下面的代码进行梯度下降，最小化 critic_loss：

```python
self.agents_critic_optim[index].zero_grad()
critic_loss.backward()
self.agents_critic_optim[index].step()
```

图 10-1　　critic 网络梯度下降实现代码

对于 actor 更新，actor 更新的目标就是最大化：

$$Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i=\boldsymbol{\mu}_i(o_i^j)} \tag{24}$$

因此首先计算当前待更新的智能体依据抽样中的当前状态由 actor 网络生成的动作，与其余所有智能体来自抽样样本的动作组合，再加上来自抽样样本的当前全局状态，作为 critic 神经网络输入参数，计算输出：

q_val = self.agents_critic_network[index](torch.cat((torch.tensor(states), actions_tensor), 1).float())

由于想利用 pytorch 中的梯度下降接口，这里 actor 想要最大化 q_val，相当于最小化-q_val，因此对目标：最小化-q_val 进行梯度下降，实现更新：

```python
actor_loss = -q_val.mean()
self.agents_actor_optim[index].zero_grad()
actor_loss.backward()
self.agents_actor_optim[index].step()
```

图 10-2　　actor 网络梯度下降实现代码

所有智能体网络更新完毕后，利用 soft_copy_params 软更新每一个智能体 target 网络的参数。agent_maddpg 中的 train_one_step 函数就是将传入的（x, a, r, x’）四元组存入经验回放池，然后对于所有智能体进行网络更新。

（二）实验组织

我们约定，使用如下缩写表示各个算法：

dqn 表示 DQN 算法；

ddqn 表示 Double DQN 算法；

duelingdqn 表示 Dueling DQN 算法；

ddpg 表示 DDPG 算法；

ac 表示 Actor-Critic 算法；

然后，以 dqn 为例，我们约定下述后缀的含义：

dqn_c：表示集中式方式训练智能体；

dqn_d：表示以分布式训练智能体，非参数共享形式；

dqn_d_ps：表示以分布式训练智能体，参数共享形式；

其中_d 和_d_ps 都属于分布式训练智能体这一扩展方式。

那么，对于每一种单智能体强化学习算法，都有三种拓展到多智能体场景下的途径，总共就有 15 个不同的算法，最后，加上 maddpg 表示 MADDPG 算法，我们总共实现了 16 种不同的多智能体强化学习算法。然后我们进行了如下三组实验：

4.2.1 实验一：使用上述 16 种算法在 2 机 2 车的场景下运行，小车采用随机游走算法，利用评估指标 K step-reward（K=1000000）来评价不同算法在训练效率上的表现；

4.2.2 实验二：使用上述 16 种算法在 2 机 2 车的场景下运行，小车采用对抗性算法，利用评估指标 K step-reward（K=1000000）来评价不同算法在对抗条件下算法性能的表现；

4.2.3 实验三：采取在实验一、二中表现较好的 ddqn_d、ddpg_d 以及 maddpg 在 5 机 5 车场景下运行，小车采用随机游走算法，利用评估指标 K step-reward（K=1000000）来评价算法在可扩展性上的表现。

五、实验结果

上述三个实验对于每一个算法都会依据我们的指标计算方法生成一个数据文件，文件中将存放在算法训练每经过一段时间后进行 20 轮测试，记录当前测试结束后训练的步数 step 以及 20 轮测试均值 mean_reward 形成二元组（step, mean_reward），作为算法性能评估源数据。该文件通过 tensorboard 就可以绘制出算法性能评估曲线。根据上述的三个实验，我们依次介绍实验结果并进行对应分析。下面所有实验结果与分析均限定于我们搭建的仿真环境以及我们设置的算法训练参数下。下面展示的所有图像中，横坐标表示算法的训练步数，纵坐标表示对应训练步数时对算法进行测试获得的奖赏均值。

（一）实验一结果与分析

2 机 2 车环境下，算法可以获得的最高奖赏是 400。16 种算法在 2 机 2 车，小车随机游走环境下训练 1,000,000 步，其算法测试获得奖赏的变化曲线如下：

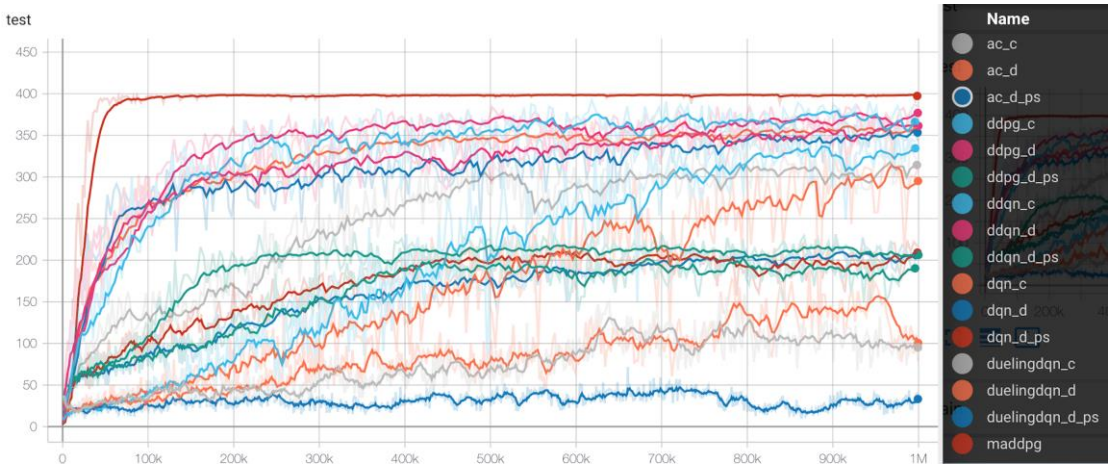

图 11　　16 种算法奖赏曲线

我们认为好的算法可以更快地收敛，并且可以获得更高的奖赏。

从上图可以看出，MADDPG 算法比其他所有算法都更快地收敛，并且相比其他算法可以学到一个更好的策略使得其在学习趋于收敛后几乎稳定在获得 400 奖赏（2 机 2 车场景下能获得的最大值）。

比较 DQN 扩展至多智能体算法不同方式：

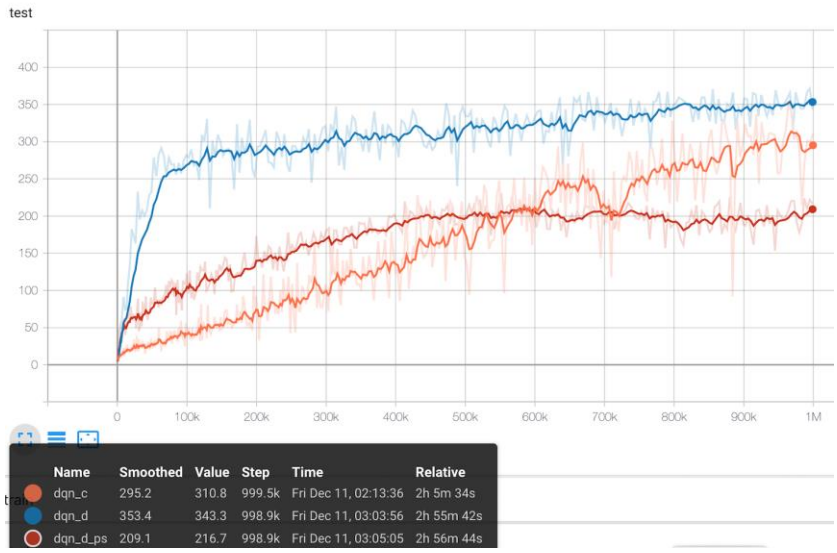

图 12　　dqn_c，dqn_d，dqn_d_ps 算法奖赏曲线

可以看出，表现最好的是 dqn_d，它的收敛速度以及最终可以达到的奖赏峰值都比另外两个好。dqn_d_ps 虽然收敛速度快于 dqn_c，但其最终结果要差于 dqn_c。从图中曲线趋势来看，在训练 1000000 步后 dqn_c 仍未收敛，说明 dqn_c 还需要更长时间的训练才能达到更好的效果。

比较 Double DQN 扩展至多智能体算法不同方式：

图 13　　ddqn_c，ddqn_d，ddqn_d_ps 算法奖赏曲线

可以发现 ddqn 的三种算法情况基本与 dqn 的三种算法情况一致。

比较 Dueling DQN 扩展至多智能体算法不同方式：

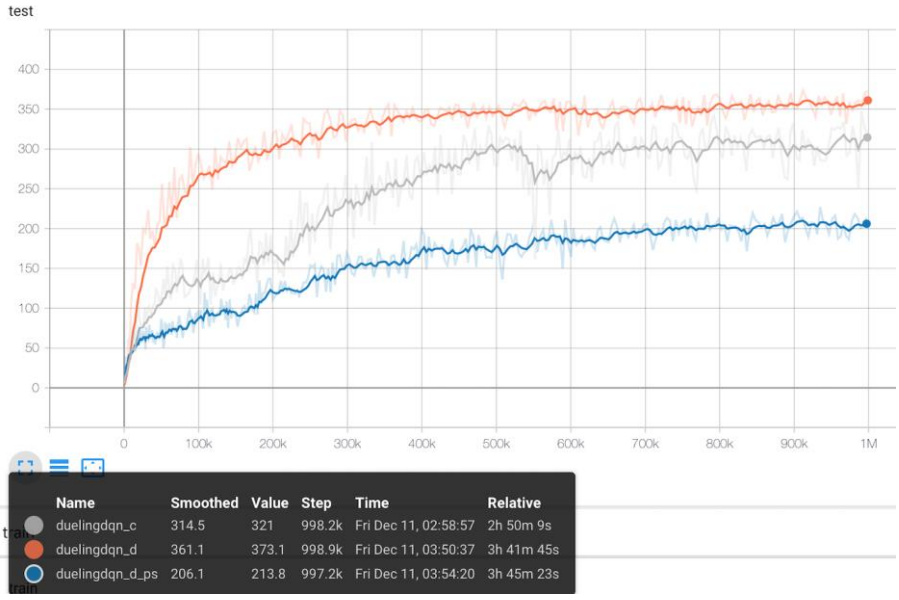

图 14　　duelingdqn_c，duelingdqn_d，duelingdqn_d_ps 算法奖赏曲线

很明显的看出三种算法的性能表现为：duelingdqn_d > duelingdqn_c > duelingdqn_d_ps。

比较 Actor-Critic 扩展至多智能体算法不同方式：

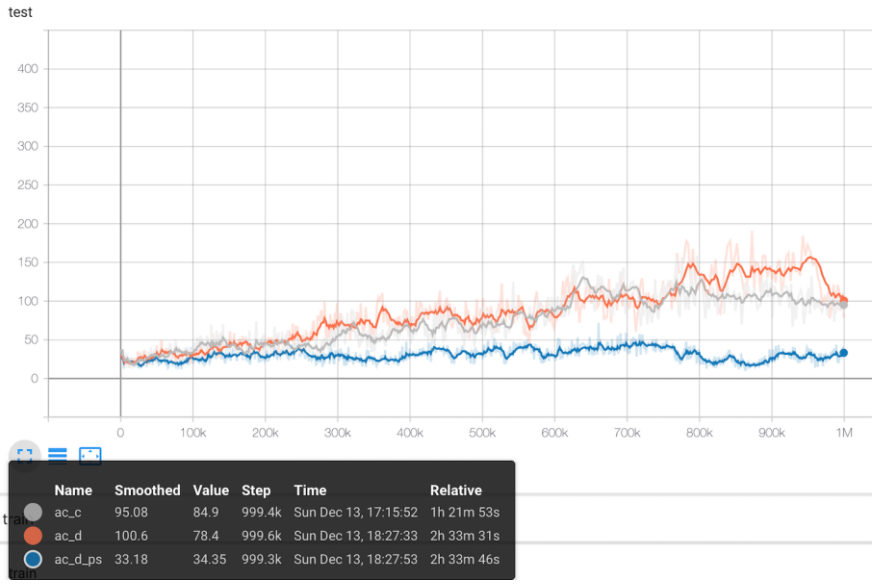

图 15　　ac_c，ac_d，ac_d_ps 算法奖赏曲线

可以看出，ac 的三种算法表现都不是很好。ac_d_ps 几乎看不出明显的上升趋势，另外两种虽然奖赏有一定的上升但是难以收敛并且上升非常缓慢。

比较 DDPG 扩展至多智能体算法不同方式：

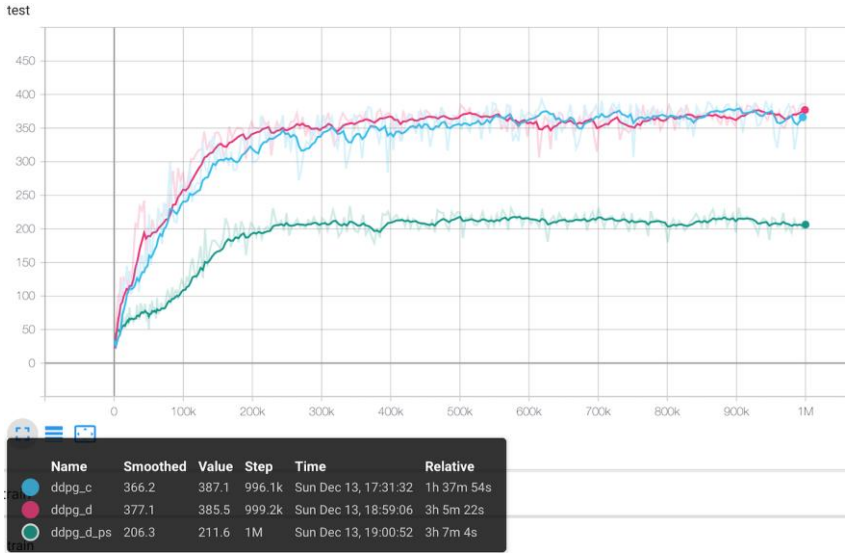

图 16　　ddpg_c，ddpg_d，ddpg_d_ps 算法奖赏曲线

可以看出 ddpg_c 和 ddpg_d 表现基本相同并且都表现较好，ddpg_d_ps 虽然也可以较快收敛，但是最后收敛时的奖赏为 200 左右，明显低于 ddpg_c 和 ddpg_d 的 350 左右。

比较所有单智能体算法在 centralized 拓展途径下的表现：

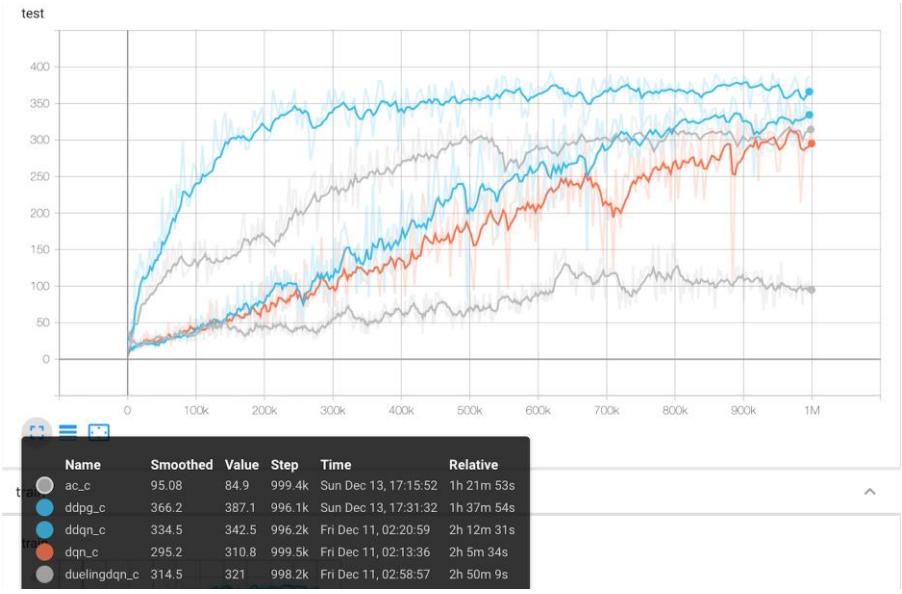

图 17　　所有单智能体算法在 centralized 拓展途径下奖赏曲线

可以看出，centralized 拓展途径下 ddpg 算法的表现最好，其次是 duelingdqn，再次是 ddqn，再次是 dqn，ac 表现最差。

比较所有单智能体算法在 distributed 参数共享拓展途径下的表现：

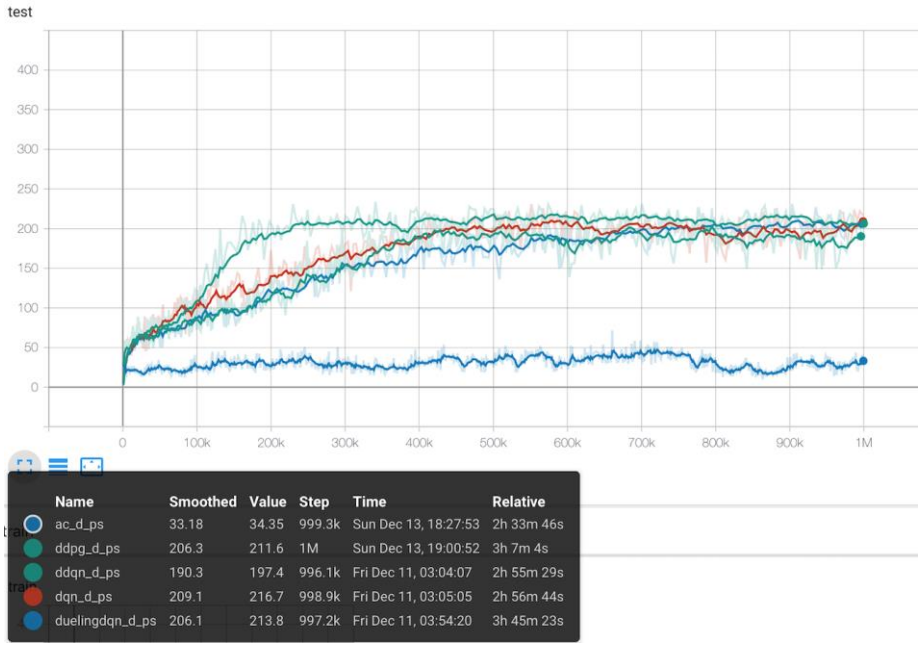

图 18　　所有单智能体算法在 distributed 参数共享拓展途径下奖赏曲线

可以看出 distributed 参数共享拓展下，ac 表现最差，几乎没有上升趋势；ddpg 收敛速度最快，表现最好，最终结果略微优于其余三种基于值函数的算法。其余三种算法表现基本相同。

我们从每一种单智能体强化学习方法的拓展至多智能体强化学习途径中找出表现最好的，发现都是 distributed 非参数共享，将所有 distributed 参数共享算法与 MADDPG 算法进行比较：

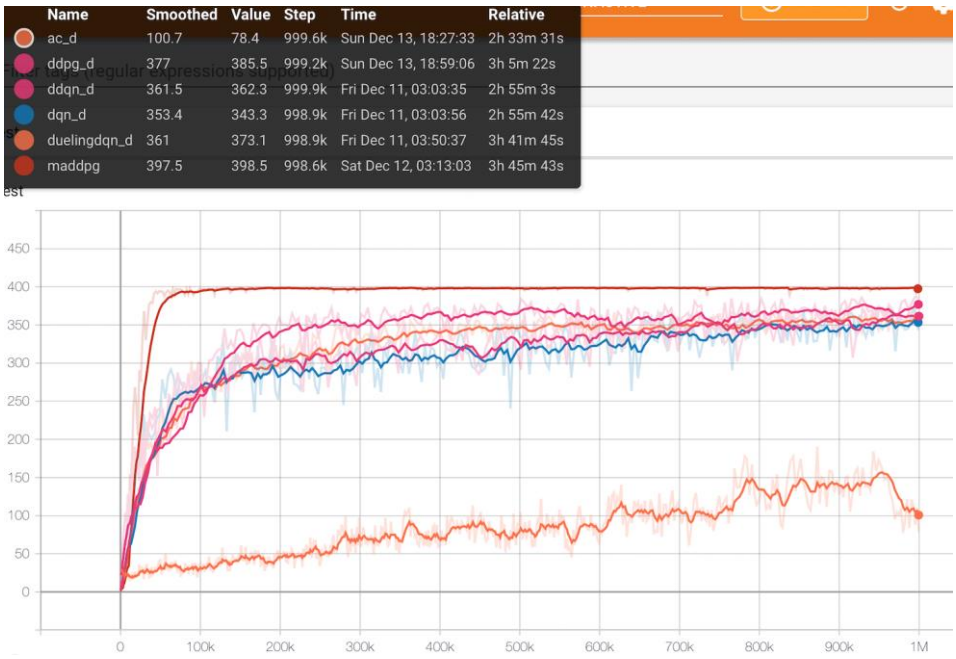

图 19　　MADDPG 和所有单智能体算法在 distributed 非参数共享拓展途径下奖赏曲线

可以看出，MADDPG 算法显著优于所有单智能体 distributed 非参数共享拓展方法。在所有 distributed 非参数共享拓展方法中，ddpg 最好，ac 最差，剩下三种方法差不多。

通过实验一，得出结论：在本研究场景下，多智能体强化学习算法 MADDPG 表现最好，收敛速度和收敛后达到的奖赏都显著优于其它单智能体扩展为多智能体的算法。其次是 DDPG。Actor-Critic 算法表现最差。单智能体扩展为多智能体的算法的扩展方式中，distributed 非参数共享可以达到更好的效果。

（二）实验二结果与分析

2 机 2 车环境下，算法可以获得的最高奖赏是 400。16 种算法在 2 机 2 车，小车采取对抗策略下的环境中训练 1,000,000 步，其算法测试获得奖赏的变化曲线如下：

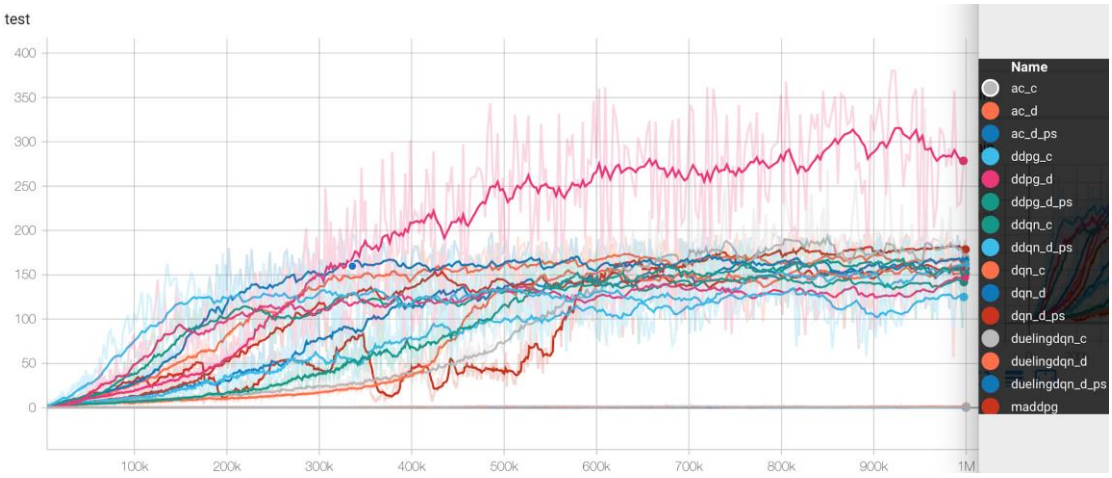

图 20　　16 种算法奖赏曲线

可以看出，对抗条件下所有算法波动都较大，其中 ddqn_d 表现显著优于其它所有算法，MADDPG 在对抗条件下失去实验一中的明显优势。并且所有算法的表现都没有它们在实验一中小车随机游走时的表现好。

比较 DQN 扩展至多智能体算法不同方式：

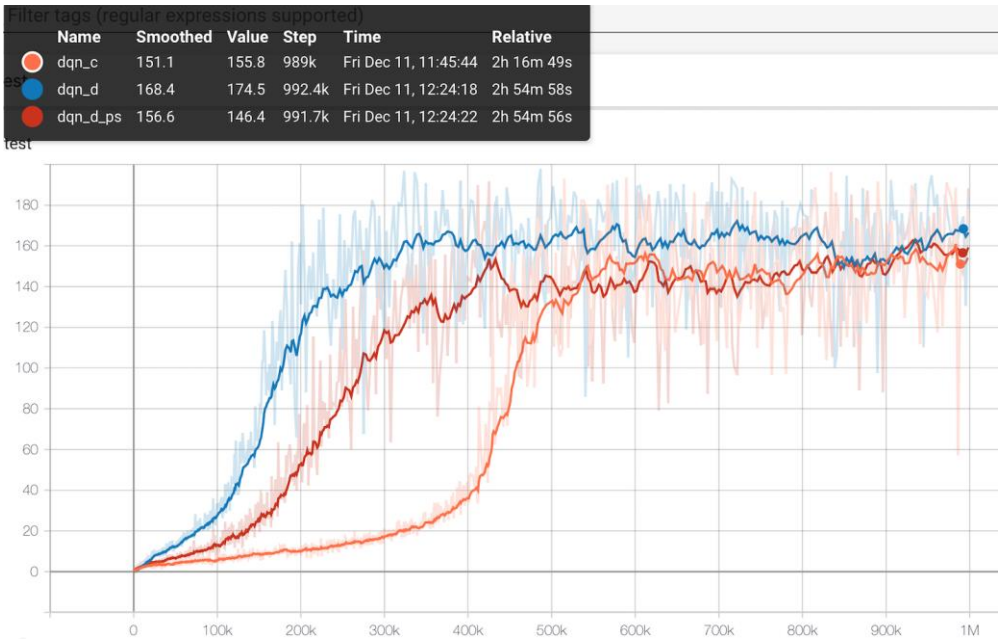

图 21　　dqn_c，dqn_d，dqn_d_ps 算法奖赏曲线

可以发现，收敛速度 dqn_d 最快，dqn_d_ps 其次，dqn_c 最慢。三种算法收敛后获得的奖赏基本相同。

比较 Double DQN 扩展至多智能体算法不同方式：

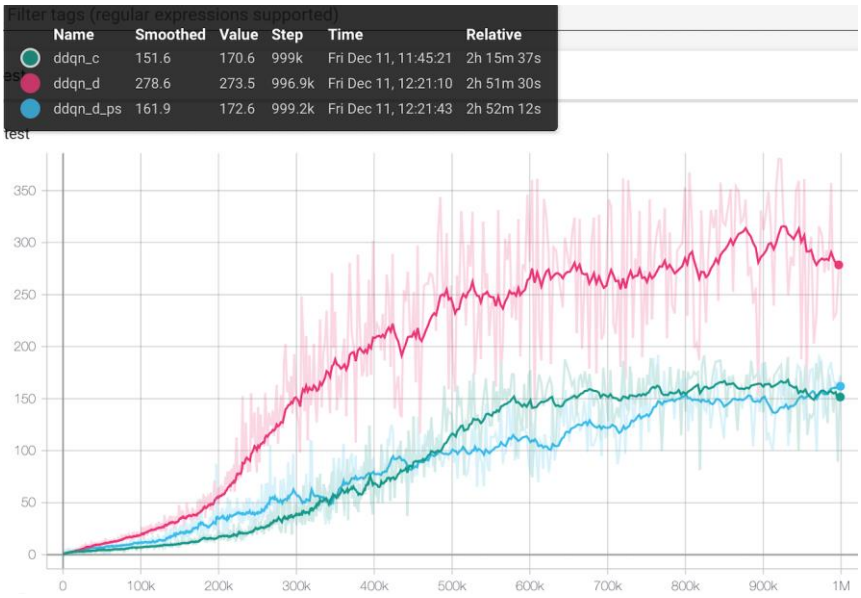

图 22　　ddqn_c，ddqn_d，ddqn_d_ps 算法奖赏曲线

可以看出，ddqn_d 表现明显优于另外两种方法，ddqn_c 表现略优于 ddqn_d_ps。

比较 Dueling DQN 扩展至多智能体算法不同方式：

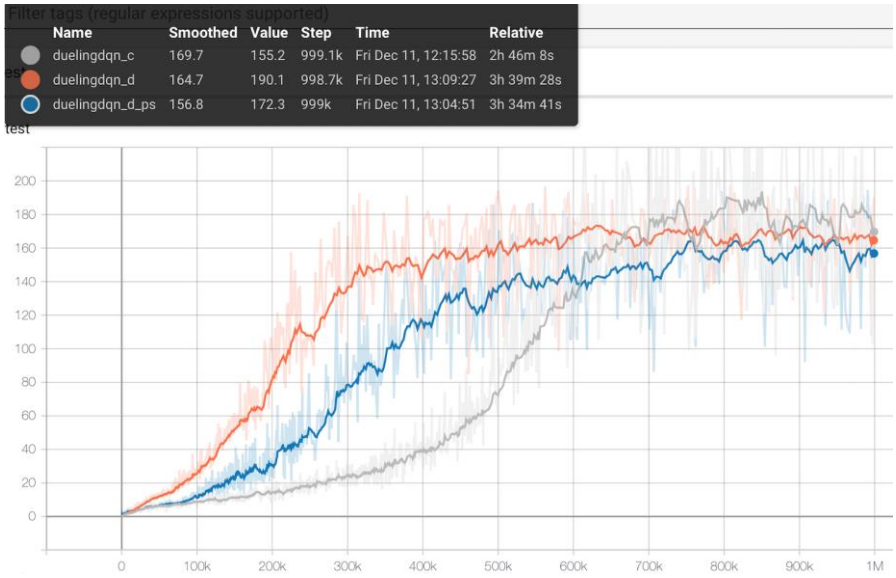

图 23　　duelingdqn_c，duelingdqn_d，duelingdqn_d_ps 算法奖赏曲线

可以看出，收敛速度从快到慢依次为 duelingdqn_d，duelingdqn_d_ps，duelingdqn_c。duelingdqn_c 虽然收敛慢，但是收敛后的奖赏要略高于另外两个。

比较 Actor-Critic 扩展至多智能体算法不同方式：

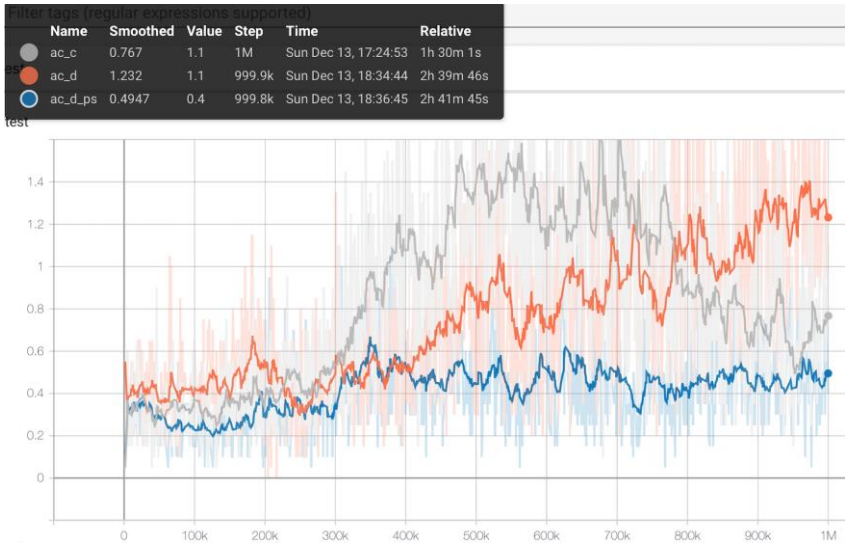

图 24　　ac_c，ac_d，ac_d_ps 算法奖赏曲线

可以看出，ac 的三种算法表现都不是很好。ac_d_ps 几乎看不出明显的上升趋势，另外两种虽然奖赏有一定的上升但是难以收敛并且上升非常缓慢，ac_c 还有明显的波动。三种算法的奖赏最大甚至都不超过 1.5（最高可以达到的奖赏值是 400），说明 ac 在对抗条件下表现很差，几乎没有效果。

比较 DDPG 扩展至多智能体算法不同方式：

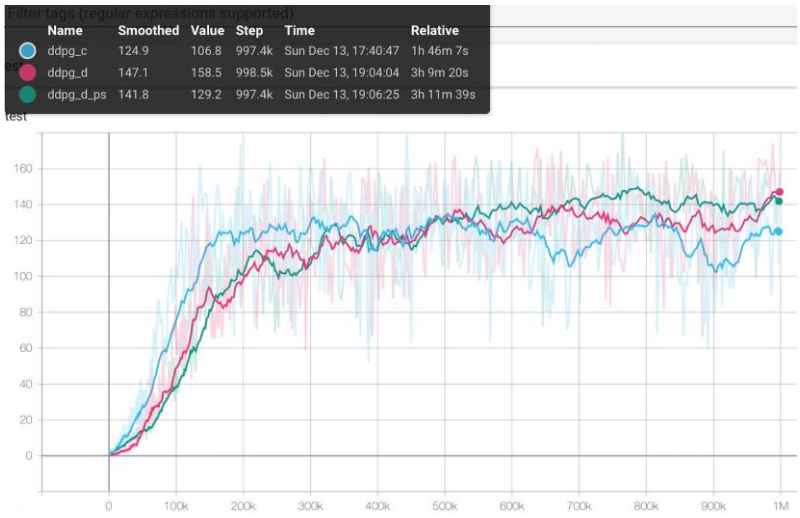

图 25　　ddpg_c，ddpg_d，ddpg_d_ps 算法奖赏曲线

可以看出，三种方法表现没有明显区别，ddpg_c 收敛速度略快于 ddpg_d 和 ddpg_d_ps，但收敛后的奖赏又略低于 ddpg_d 和 ddpg_d_ps。ddpg_d 和 ddpg_d_ps 两者表现基本一致。

比较所有单智能体算法在 centralized 拓展途径下的表现：

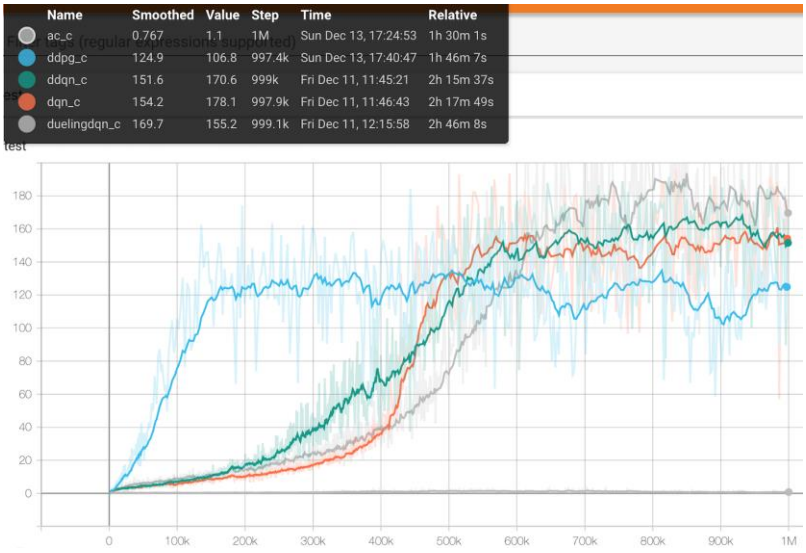

图 26　　所有单智能体算法在 centralized 拓展途径下奖赏曲线

可以看出，ac 算法表现很差，几乎成为 x 轴。在不考虑 ac 算法后，ddpg 收敛速度远快于另外三个，但是收敛后奖赏又是四种中最低的，收敛后奖赏 duelingdqn 最高。

比较所有单智能体算法在 distributed 非参数共享拓展途径下的表现：

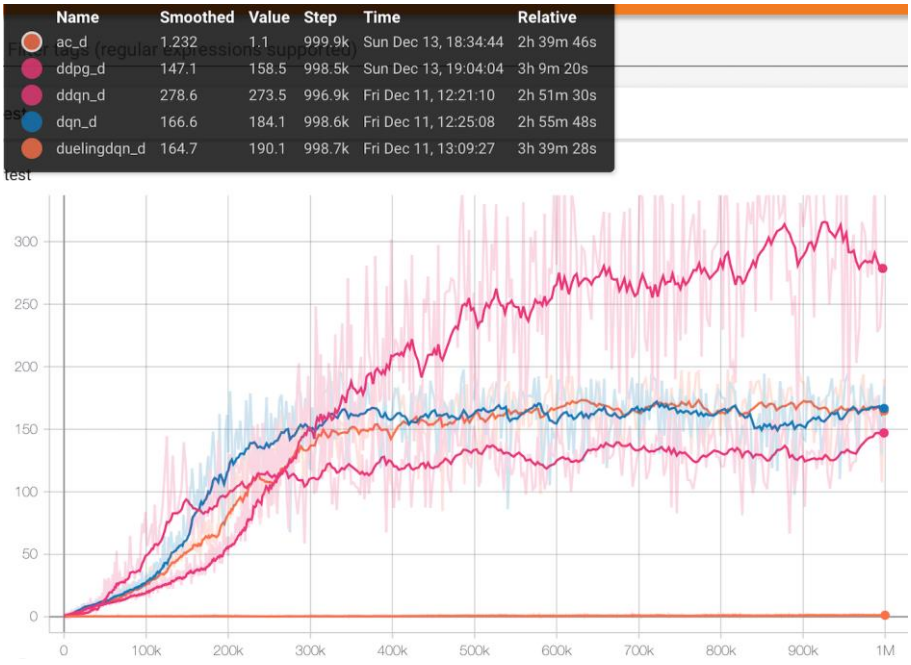

图 27　　所有单智能体算法在 distributed 非参数共享途径下奖赏曲线

可以看出，ac 算法表现很差，几乎成为 x 轴。在不考虑 ac 算法后，ddqn 表现最好，dqn 和 duelingdqn 表现基本相同，ddpg 表现略差于 dqn 和 duelingdqn。

比较所有单智能体算法在 distributed 参数共享拓展途径下的表现：

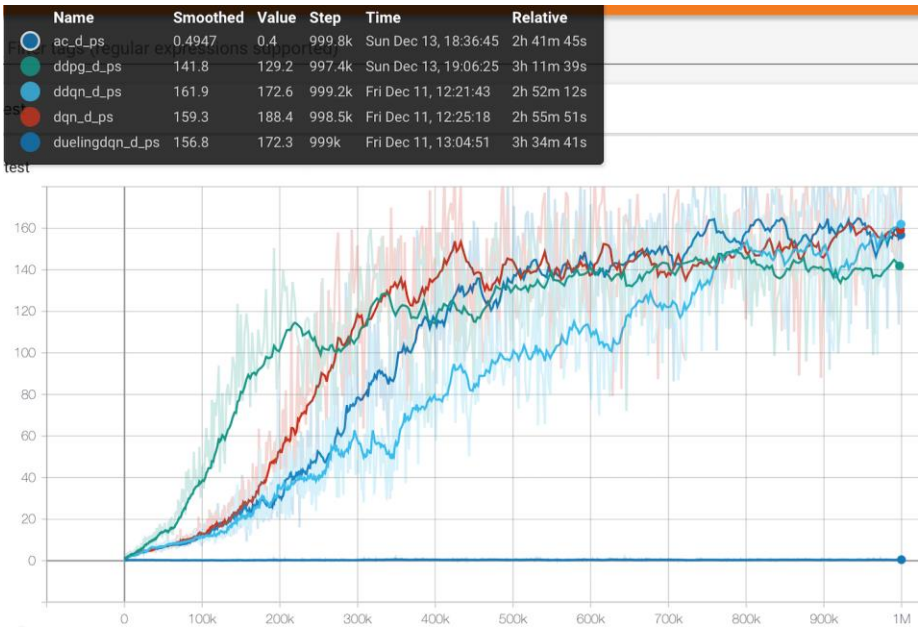

图 28　　所有单智能体算法在 distributed 参数共享途径下奖赏曲线

可以看出，ac 算法表现很差，几乎成为 x 轴。在不考虑 ac 算法后，剩余算法最终收敛后的奖赏没有太大区别，但是收敛速度由快到慢分别是：ddpg，dqn，duelingdqn，ddqn。

我们从每一种单智能体强化学习方法的拓展至多智能体强化学习途径中找出表现相对较好的，分别是 ac_d，ddpg_d_ps，ddqn_d，dqn_d，duelingdqn_d，将它们与 maddpg 比较如下图：

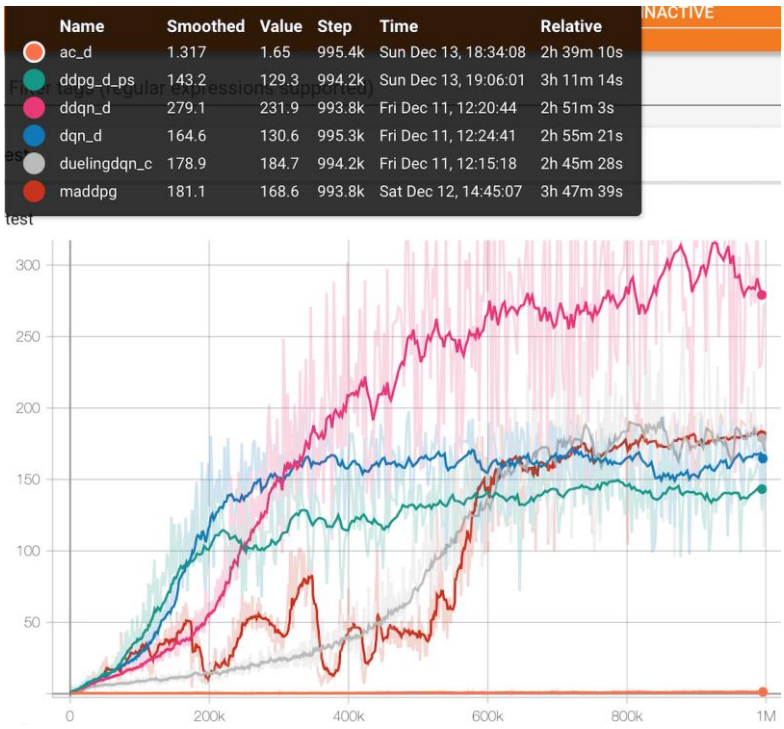

图 29　　ac_d，ddpg_d_ps，ddqn_d，dqn_d，duelingdqn_d，maddpg 奖赏曲线

可以看出，ac 算法表现很差，几乎成为 x 轴。在不考虑 ac 算法后，ddqn_d 表现最好，虽然收敛速度不如 ddpg_d_ps 和 dqn_d 快，但是最终收敛后奖赏 ddqn_d 远高于另外的算法。dqn_d 表现比 ddpg_d_ps 略好，因为收敛后奖赏更高。maddpg 收敛速度与 duelingdqn_d 相近，但是 maddpg 前期有明显的波动。收敛后 maddpg 与 duelingdqn_d 奖赏相近，均略高于 dqn_d。

通过实验二，得出结论：在对抗条件下，所有算法都出现明显波动，多智能体强化学习算法 MADDPG 失去优势，收敛前期波动较大，综合看 ddqn_d 算法表现最好。Actor-Critic 的所有扩展至多智能体的算法表现都非常差。单智能体扩展为多智能体的算法的扩展方式中，相对而言 distributed 非参数共享可以达到更好的效果。

（三）实验三结果与分析

5 机 5 车环境下，算法可以获得的最高奖赏是 1000。采取在实验一、二中表现较好的 ddqn_d、ddpg_d 以及 maddpg 在 5 机 5 车，小车采取随机游走下的环境中训练 1,000,000 步，与这三种算法在 2 机 2 车，小车采取随机游走下的环境中训练 1,000,000 步对比，其算法测试获得奖赏的变化曲线如下：

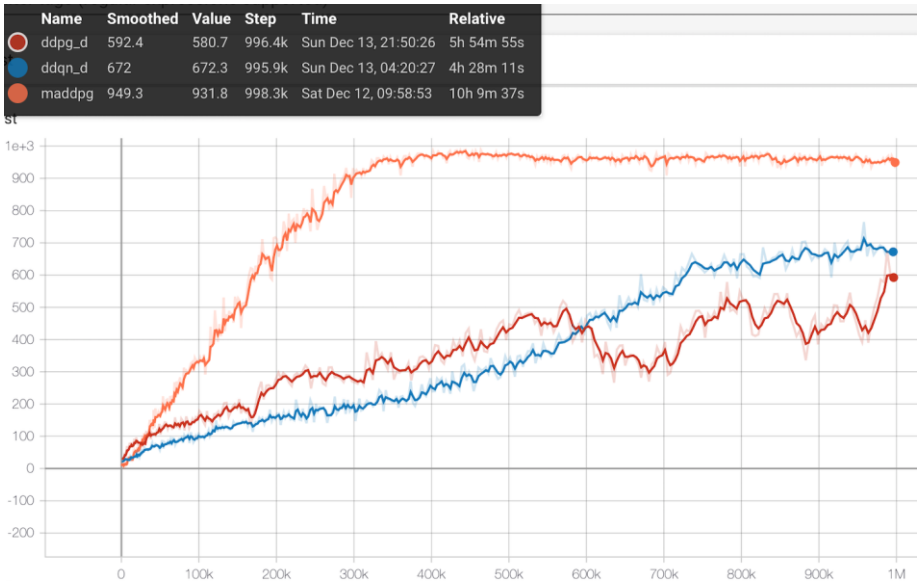

图 30　　5 机 5 车下 ddqn_d、ddpg_d 以及 maddpg 奖赏变化曲线

对比这三种算法在 2 机 2 车，小车采取随机游走下结果：

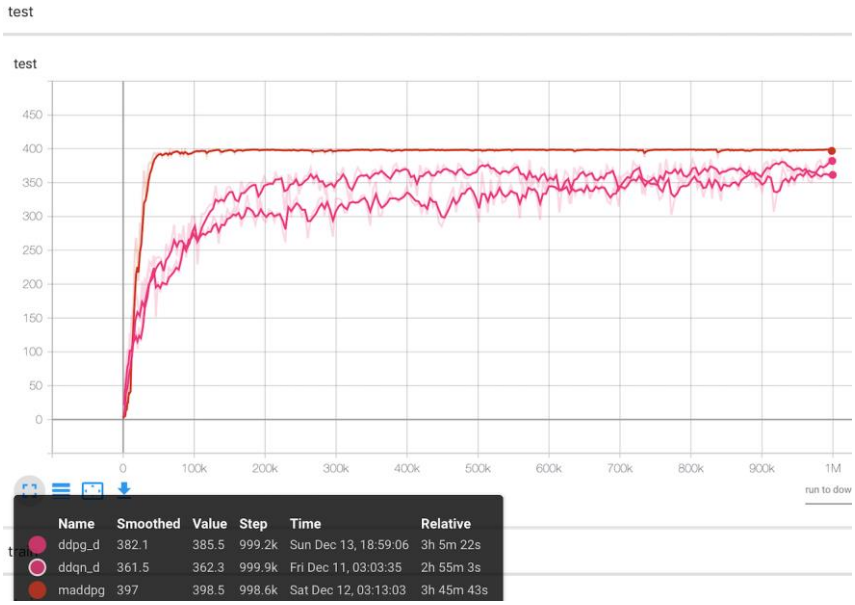

图 31　　2 机 2 车下 ddqn_d、ddpg_d 以及 maddpg 奖赏变化曲线

可以明显看出，虽然 2 机 2 车下 ddqn_d 和 ddpg_d 可以较快收敛，5 机 5 车下训练 1000000 步，传统的单智能体扩展为多智能体的算法 ddqn_d 和 ddpg_d 的奖赏还在爬升阶段，还没有收敛，并且 ddqn_d 在 5 机 5 车下波动较大。5 机 5 车下 maddpg 明显比 ddqn_d 和 ddpg_d 收敛的快并且收敛后很稳定的保持在奖赏 960 左右（最高 1000）。2 机 2 车下 maddpg 也比 ddqn_d 和 ddpg_d 收敛的快并且收敛后奖赏比 ddqn_d 和 ddpg_d 高。

通过实验三，得出结论：当智能体规模进一步扩大时，将传统单智能体强化学习算法扩展为多智能体的算法不能很好的适应，可能出现波动，并且需要较长时间训练才能收敛，并且收敛后效果可能不会很好。而专门为多智能体定制的 MADDPG 有很好的可扩展性，在 5 机 5 车情况下仍然可以较快收敛并且达到很高的奖赏值，说明可扩展性方面 MADDPG 优于传统单智能体强化学习算法扩展为多智能体的算法。

综上，我们得出实验结论：在我们设定的场景中，目标随机游走条件下，MADDPG 收敛速度和收敛后达到的奖赏显著优于其它单智能体扩展为多智能体的算法，但在对抗条件下是 Double DQN 利用 distributed 非参数共享扩展算法表现最好。Actor-Critic 算法表现最差。单智能体扩展为多智能体的算法的扩展方式中，distributed 非参数共享可以达到更好的效果。MADDPG 算法对于智能体数量的扩展性表现最好。

六、讨论

我们搭建的模拟环境仅针对我们研究的特定场景，对于不同的问题应该搭建不同的模拟环境进行研究。同时，我们工作存在局限性，各个算法的超参数未必最优，几乎可以说肯定不是最优参数，但是如此多算法的调参工作对于时间、精力和设备的要求都非常高，因此我们尽量挑选出了一批表现较好的参数，并保证了各个算法中相同的超参数取值一致。在小车采取了对抗策略时，所有算法的表现明显变差，并且非常不稳定，波动很厉害。这一方面说明了我们的小车策略相较于随机策略有明显的对抗作用，另一方面也说明了我们的算法还有改进的空间，例如算法训练中的波动问题可能是由于经验池中的 transaction 带有噪声、某些梯度过大导致网络参数剧烈变化造成的。直接将单智能体算法应用于多智能体场景，存在的一个主要问题是，环境中的状态转移概率不再服从一个特定的分布，而是会随着智能体策略的改进而不断变化，这给智能体的训练带来了不小的麻烦。MADDPG 算法可以缓解这一矛盾，因此再扩展到 5v5 场景下，MADDPG 算法仍能表现的很好，而 DDPG 算法的表现下滑严重。

七、总结与展望

在本文中，我们提出一种多智能体强化学习算法实现框架，在框架之中我们将现有的主流强化学习算法：DQN，Double DQN，Dueling DQN，Actor-Critic，DDPG，MADDPG 应用到了追踪问题中，并且针对传统单智能体强化学习方法：DQN，Double DQN，Dueling DQN，Actor-Critic，DDPG，我们利用了两种途径：centralized，distributed（其中包括 parameter-share 和非 parameter-share）将它们扩展成为多智能体强化学习算法。我们设计并实现了高效的实验仿真平台以及一组评判指标，系统性地评价了上述主流强化学习算法以及将传统单智能体强化学习方法扩展为多智能体强化学习算法的两种途径在目标追踪问题上的表现优劣。

目前我们仅对 5 种单智能体强化学习算法以及 MADDPG 算法进行了测试，未来我们可以实现更多单智能体或多智能体算法，进行更为全面的研究。此外，我们的模拟环境实现的仍较为简陋，未来我们考虑向其中加入地形、障碍物、阻力等因素，以更好地模拟现实环境中的场景。我们用于评判算法的指标也比较少，仅基于智能体训练过程中奖赏的变化，未来考虑加入更多的评价指标，使得我们的研究更加完整。

八、致谢

感谢计算机科学与技术系软件研究所前沿交叉中心的陶先平教授，汪亮副教授对我们项目的关心和指导以及给我们提供充足的学习资源和硬件资源，感谢计算机科学与技术系软件研究所前沿交叉中心的王文学长和胡浪学长向我们提供了悉心的技术指导。

参考文献：

[1] Pham H X, La H M, Feil-Seifer D, et al. Cooperative and distributed reinforcement learning of drones for field coverage[J]. arXiv preprint arXiv:1803.07250, 2018.

[2] La H M, Lim R, Sheng W. Multirobot cooperative learning for predator avoidance[J]. IEEE Transactions on Control Systems Technology, 2014, 23(1): 52-63.

[3] Zheng L, Yang J, Cai H, et al. MAgent: A many-agent reinforcement learning platform for artificial collective intelligence[C]//Thirty-Second AAAI Conference on Artificial Intelligence. 2018.

[4] Zhang H, Feng S, Liu C, et al. CityFlow: A Multi-Agent Reinforcement Learning Environment for Large Scale City Traffic Scenario[C]//The World Wide Web Conference. ACM, 2019: 3620-3624.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. CoRR, abs/1312.5602, 2013.

[6] Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G et al.: Human-level control through deep reinforcement learning. Nature 2015, 518:529-533.

[7] H. V. Hasselt, A. Guez, and D. Silver, ‘‘Deep reinforcement learning with double Q-learning,’’ in Proc. 13th AAAI Conf. Artif. Intell., 2016, pp. 2094–2100.

[8] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581, 2015.

[9] Konda, Vijaymohan. Actor-critic algorithms[J]. Siam Journal on Control and Optimization, 2003, 42(4):1143-1166.

[10] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning.” [Online]. Available: https://arxiv.org/pdf/1509.02971

[11] Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, “Multi-agent actor-critic for mixed cooperative-competitive environments,” Advances in neural information processing systems, vol. 30, pp. 6379–6390, 2017.

附录

下面是我们各个算法训练时使用的超参数：

DQN、Double DQN、Dueling DQN：

超参数	值	描述
batch size	50	每次更新网络参数从经验池中抽取的 transaction 数目
buffer size	1000000	经验池容量大小
epsilon	0.15	ε-贪心探索中ε的初始值
final epsilon	0.02	ε-贪心探索中ε的最终值
total train steps	500000	ε从初始值线性衰减到最终值需要的训练步数
update frequency	1	每隔 update frequency 个训练步更新一次网络参数
update count	1	每次更新网络参数从经验池中抽取 update count 次 transaction 来更新 update count 次网络参数
learning rate	0.0005	Q 网络更新的学习率
gamma	0.95	Q-learning 更新中的折扣因子 gamma
tau	100	每隔 tau 个训练步将目标 Q 网络的参数更新为当前 Q 网络

		的参数

DDPG、MADDPG：

超参数	值	描述
batch size	50	每次更新网络参数从经验池中抽取的 transaction 数目
buffer size	1000000	经验池存放 transaction 数目的上限
update frequency	1	每隔 update frequency 个训练步更新一次网络参数
update count	1	每次更新网络参数从经验池中抽取 update count 次 transaction 来更新 update count 次网络参数
scale	1.0	用于生成探索动作噪声的高斯分布的标准差初始值
final scale	0.1	用于生成探索动作噪声的高斯分布的标准差最终值
scale discount factor	0.9999	每隔一个训练步 scale 减小的折扣因子
gamma	0.95	Q-learning 更新中的折扣因子 gamma，用于更新 Critic 网络
actor learning rate	0.0001	actor 网络更新的学习率
critic learning rate	0.001	critic 网络更新的学习率

actor critic 算法的参数除了没有 buffer size 之外，其余均和 ddpg 算法参数相同。

我们设计的基于多智能体强化学习的无人机群目标追踪算法训练测试框架和实验平台以及所有实现的多智能体强化学习算法可以在网址：https://github.com/MartinYuanNJU/MARL_Train-Test_Platform 获取实现的完整源代码。