清 华 大 学

综 合 论 文 训 练

题目：基于 PyTorch 的深度强化学习平台设计与实现

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：翁家翌

指导教师：苏　航　副研究员

2020 年 6 月 1 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

深度强化学习近年来取得了一系列的突破，在包括 Atari 游戏[1]、围棋[2]、蛋白质结构预测[3] 和策略游戏 Dota2[4-5] 等多个领域取得了极大进展，提升了业界对深度强化学习的需求与信心。但是，目前主流深度强化学习平台框架无法很好地满足这一日渐增长的需求。无论是学术研究领域还是工业应用，现有框架普遍存在缺乏灵活的可定制化接口、代码嵌套关系复杂、训练速度较慢、算法正确性缺乏验证等等缺点。研究者们通常需要大幅改动框架结构，亦或是从头开始编写算法程序才能满足自身需求，阻碍了强化学习技术的进一步应用。因此，一个灵活可定制、代码简洁、训练速度快、有着可靠测试的强化学习平台对整个社区而言十分重要。

针对以上问题，本项目构建了一个基于 PyTorch[6] 的深度强化学习平台**天授**。天授平台仅通过 2000 行代码，简洁地实现了基于策略梯度、基于 Q 价值函数、综合 Q 价值与策略梯度、模仿学习等 10 余种主流强化学习算法及其主要改进，支持了部分观测马尔科夫决策过程训练以及多种仿真环境的数据处理，将主流强化学习算法充分模块化并实现了需求可定制化，在和其它著名强化学习平台进行的性能对比评测中以显著优势胜出。天授旨在为用户提供一个更加友好的强化学习算法平台，降低强化学习算法的开发成本。平台代码已经在 GitHub 上开源：https://github.com/thu-ml/tianshou/，目前获得了 1500 多个星标，受到学术界和产业界的广泛关注。

关键词：强化学习；算法；平台；PyTorch

ABSTRACT

Deep reinforcement learning has made a series of breakthroughs in recent years, including the fields of Atari game[1], Go[2], protein structure prediction[3], and strategy game Dota2[4-5], enhancing the demand and confidence of the industry for deep reinforcement learning. However, the current frameworks of existing deep reinforcement learning platforms are not well-positioned to meet this growing demand, both in academic research areas and industrial applications. The existing frameworks generally suffer from shortcomings such as lack of flexible and customizable interfaces, complex and complicated code nesting relationships, slow training speed, and lack of unit tests. As a result, researchers often need to drastically change the framework structure, or write programs of deep reinforcement learning algorithms from scratch to meet their needs, hindering the further use of reinforcement learning technologies. Thus, a reinforcement learning platform that is flexible, customizable, simple to code, fast to train, and reliably tested is especially important.

In response to the above questions, this project builds a deep reinforcement learning platform **Tianshou** based on PyTorch[6]. With only 2000 lines of code, Tianshou succinctly implements many mainstream reinforcement learning algorithms and their main improvements based on policy gradient, Q value function, integrated Q value and policy gradient, imitation learning, etc. It supports Partially Observable Markov Decision Process (POMDP) training and data processing in various simulation scenarios, fully modularizes various reinforcement learning algorithms, and wins in the performance comparison evaluation with other reinforcement learning platforms with significant advantages. Tianshou aims to provide users with a more user-friendly platform for reinforcement learning algorithms and reduce development costs. The platform code is currently open-source on GitHub: https://github.com/thu-ml/tianshou/ and has garnered over 1,500 stars, gaining a lot of attention from academia and industry.

Keywords: Reinforcement Learning; Algorithm; Platform; PyTorch

目 录

主要符号对照表

RL	强化学习 (Reinforcement Learning)
MFRL	免模型强化学习 (Model-free Reinforcement Learning)
MBRL	基于模型的强化学习 (Model-based Reinforcement Learning)
MARL	多智能体强化学习 (Multi-agent Reinforcement Learning)
MetaRL	元强化学习（Meta Reinforcement Learning）
IL	模仿学习 (Imitation Learning)
On-policy	同策略
Off-policy	异策略
MDP	马尔科夫决策过程 (Markov Decision Process)
POMDP	部分可观测马尔科夫决策过程 (Partially Observable Markov Decision Process)
Agent	智能体
π, Policy	策略
Actor	动作（网络），又称作策略（网络）
Critic	评价（网络）
$s \in \mathcal{S}$, State	状态
$o \in \mathcal{O}$, Observation	观测值，为状态的一部分，$o \subseteq s$
$a \in \mathcal{A}$, Action	动作
$r \in \mathcal{R}$, Reward	奖励
$d \in \{0, 1\}$, Done	结束符，0 表示未结束，1 表示结束
s_t, o_t, a_t, r_t, d_t	在一个轨迹中时刻 t 的状态、观测值、动作、奖励和结束符
$P^a_{ss'} \in \mathcal{P}$	在当前状态 s 采取动作 a 之后，转移到状态 s' 的概率；$P^a_{ss'} = \mathbb{P}\{s_{t+1} = s' \mid s_t = s, a_t = a\}$
R^a_s	在当前状态 s 采取动作 a 之后所能获得的期望奖励；$R^a_s = \mathbb{E}[r_t \mid s_t = s, a_t = a]$
γ	折扣因子，作为对未来回报不确定性的一个约束项，$\gamma \in [0, 1]$
G_t, Return	累计折扣回报，$G_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$
$\pi(a\|s)$	随机性策略，表示获取状态 s 之后采取的动作 a 的概率
$\pi(s)$	确定性策略，表示获取状态 s 之后采取的动作

$V(s)$	状态值函数（State-Value Function），表示状态 s 对应的期望累计折扣回报
$V^{\pi}(s)$	使用策略 π 所对应的状态值函数，$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid s_t = s]$
$Q(s,a)$	动作值函数（Action-Value Function），表示状态 s 下采取动作 a 所对应的期望累计折扣回报
$Q^{\pi}(s,a)$	使用策略 π 所对应的动作值函数，$Q^{\pi}(s,a) = \mathbb{E}_{a\sim\pi}[G_t \mid s_t = s, a_t = a]$
$A(s,a)$	优势函数，$A(s,a) = Q(s,a) - V(s)$
Batch	数据组
Buffer	数据缓冲区
Replay Buffer	重放缓冲区
RNN	循环神经网络（Recurrent Neural Network）

第 1 章　引言

1.1　深度强化学习研究背景

在 2012 年 AlexNet[7] 夺得 ImageNet 图像分类比赛冠军之后，深度神经网络被应用在许多领域，如目标检测和跟踪；并且在一系列的任务中，深度学习模型的准确率达到甚至超过了人类水准。大规模的商业化应用随之而来，如人脸识别、医疗图像处理等领域使用深度神经网络提高识别的速度和精度。

强化学习概念的提出最早可追溯到 20 世纪，其在简单场景上的应用于上世纪 90 年代至本世纪初即被深入研究，比如 1992 年使用强化学习算法打败了人类西洋双陆棋玩家[8]。早期的强化学习算法大多使用线性回归来拟合策略函数，并且需要预先提取人为定义好的特征，实际效果不甚理想。2013 年之后，结合了深度学习的优势，深度强化学习使用深度神经网络进行函数拟合，展现出了其强大的威力，如使用 DQN[1] 玩 Atari 游戏达到人类水准、AlphaGo[2] 与人类顶尖围棋选手的划时代人机对战，OpenAI Five[5] 在 Dota2 5v5 对战比赛中击败人类冠军团队，无论是学术界还是工业界都对这一领域表现出了极大兴趣。深度强化学习如今不但被应用在游戏 AI 中，还被使用在如机械臂抓取、自动驾驶、高频交易、智能交通等等实际场景中，其前景十分广阔。

1.2　深度强化学习平台框架现状

1.2.1　现有深度强化学习平台简介

深度强化学习算法由于其计算模式不规则和高并发的特点，导致其无法像计算机视觉、自然语言处理领域的计算框架一样，从训练数据流的角度进行设计与实现；而强化学习算法形式与实现细节难以统一，又进一步加大了平台的编写难度。一些项目尝试解决通用强化学习算法框架问题，但是通常情况下，深度强化学习领域的研究者不得不从头编写一个特定强化学习算法的程序来满足自己的需求。

以 GitHub 星标数量大于约一千为标准，现有使用较为广泛的深度强化学习平台有 OpenAI 的 Baselines[9]、SpinningUp[10]，加州伯克利大学的开源分布式强化学习框架 RLlib[11]、rlpyt[12]、rlkit[13]、Garage[14]，谷歌公司的 Dopamine[15]、

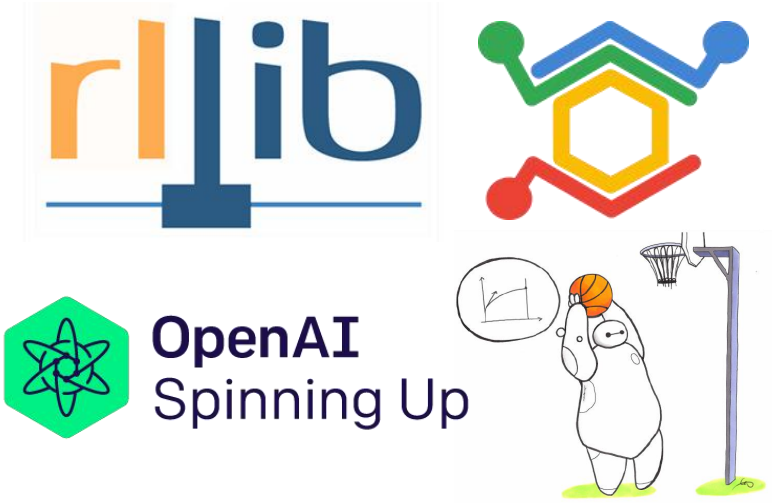


图 1.1　目前较为主流的深度强化学习算法平台

B-suite[16]，以及其他独立开发的平台 Stable-Baselines[17]、keras-rl[18]、PyTorch-DRL[19]、TensorForce[20]。图 1.1 展示了若干主流强化学习算法平台的标志，表 1.1 列举了各个框架的基本信息。

几乎所有的强化学习平台都以 OpenAI Gym[21] 所定义的 API 作为智能体与环境进行交互的标准接口，以 TensorFlow[22] 作为后端深度学习框架为主的平台居多，支持了至少 4 种免模型强化学习算法。大部分平台支持对训练环境进行自定义配置。

PyTorch[6] 是 Facebook 公司推出的一款开源深度学习框架，由于其易用性、接口稳定性和社区活跃性，受到越来越多学术界和工业界研究者的青睐，大有超过 TensorFlow 框架的趋势。然而使用 PyTorch 编写的深度强化学习框架中，星标最多为 PyTorch-DRL[19]（2400+ 星标），远远不如 TensorFlow 强化学习社区中的开源框架活跃。本文将在下一小节分析讨论其详细原因。

1.2.2　现有深度强化学习平台不足

表 1.1 按照 GitHub 星标数目降序排列，从后端框架、是否模块化、文档完善程度、代码质量、单元测试和最后维护时间这些维度，列举了比较流行的深度强化学习开源平台框架。这些平台框架在不同评价维度上或多或少有些缺陷，从而降低了用户体验。此处列出一些典型问题，如下所示：

- **算法模块化不足：**以 OpenAI Baselines 为代表，将每个强化学习算法单独独立成一份代码，因此无法做到代码之间的复用。用户在使用相关代码时，必须逐一修改每份代码，带来了极大困难。

表 1.1　深度强化学习平台总览，按照 GitHub 星标数从大到小排序，截止 2020/05/12

平台名称	星标数	后端框架	模块化	文档	代码质量	单元测试	上次更新
Ray/RLlib[11]	11460	TF/PyTorch	√	较全	10 / 24065	√	2020.5
Baselines[9]	9764	TF	×	无	2673 / 10411	√	2020.1
Dopamine[15]	8845	TF1	√	较全	180 / 2519	√	2019.12
SpinningUp[10]	4630	TF1/PyTorch	×	全面	1656 / 3724	×	2019.11
keras-rl[18]	4612	Keras	√	不全	522 / 2346	√	2019.11
Tensorforce[20]	2669	TF	√	全面	3834 / 13609	√	2020.5
PyTorch-DRL[19]	2424	PyTorch	√	无	2144 / 4307	√	2020.2
Stable-Baselines[17]	2054	TF1	×	全面	2891 / 10989	√	2020.5
天授	1529	PyTorch	√	全面	0 / 2141	√	2020.5
rlpyt[12]	1448	PyTorch	√	较全	1191 / 14493	×	2020.4
rlkit[13]	1172	PyTorch	√	不全	275 / 7824	×	2020.3
B-suite[16]	975	TF2	×	无	220 / 5353	×	2020.5
Garage[14]	709	TF1/PyTorch	√	不全	5 / 17820	√	2020.5

注：TF 为 TensorFlow 缩写，包含版本 v1 和 v2；
TF1 为 TensorFlow v1 版本缩写，不包含版本 v2；
TF2 为 TensorFlow v2 版本缩写，不包含版本 v1；
代码质量一栏数据格式为 “PEP8 **不符合**规范数 / 项目 Python 文件行数”。

- **实现算法种类有限：**以 Dopamine 和 SpinningUp 为代表，Dopamine 框架只支持 DQN 算法族，并不支持策略梯度；SpinningUp 只支持策略梯度算法族，未实现 Q 学习的一系列算法。两个著名的平台所支持的强化学习算法均不全面。
- **代码实现复杂度过高：**以 RLlib 为代表，代码层层封装嵌套，用户难以进行二次开发。
- **文档不完整：**文档应包含教程和代码注释，部分平台只实现了其一，甚至完全没有文档，十分影响平台框架的使用。
- **平台性能不佳：**强化学习算法本身难以调试，如果能够提升平台性能则将会大幅度降低调试难度。仍然以 OpenAI Baselines 为代表，无法全面支持并行环境采样，十分影响训练效率。
- **缺少完整单元测试：**单元测试保证了代码的正确性和结果可复现性，但几乎所有平台只做了功能性验证，而没有进行完整的训练过程验证。

- **环境定制支持不足：**许多非强化学习领域的研究者想使用强化学习算法来解决自己领域内问题，因此所交互的环境并不是 Gym 已经定制好的，这需要平台框架支持更多种类的环境，比如机械臂抓取所需的多模态环境。以 rlpyt 为例，该平台将环境进行封装，研究者如果想使用非 Atari 的环境必须大费周折改动框架代码。

此外值得讨论的是 PyTorch 深度强化学习框架活跃程度不如 TensorFlow 社区这个问题。不少使用 PyTorch 的研究者是编写独立的强化学习算法来满足自己需求，虽然实现较 TensorFlow 简单很多，但却没有针对数据流、数据存储进行优化；从表 1.1 中也可以看出已有基于 PyTorch 的深度强化学习平台以 PyTorch-DRL 为代表，文档不全面、代码质量不如独立手写的算法高亦或是封装程度过高、缺乏可靠的单元测试，一定程度上阻碍了这些平台的进一步发展。

1.3 主要贡献与论文结构

1.3.1 主要贡献

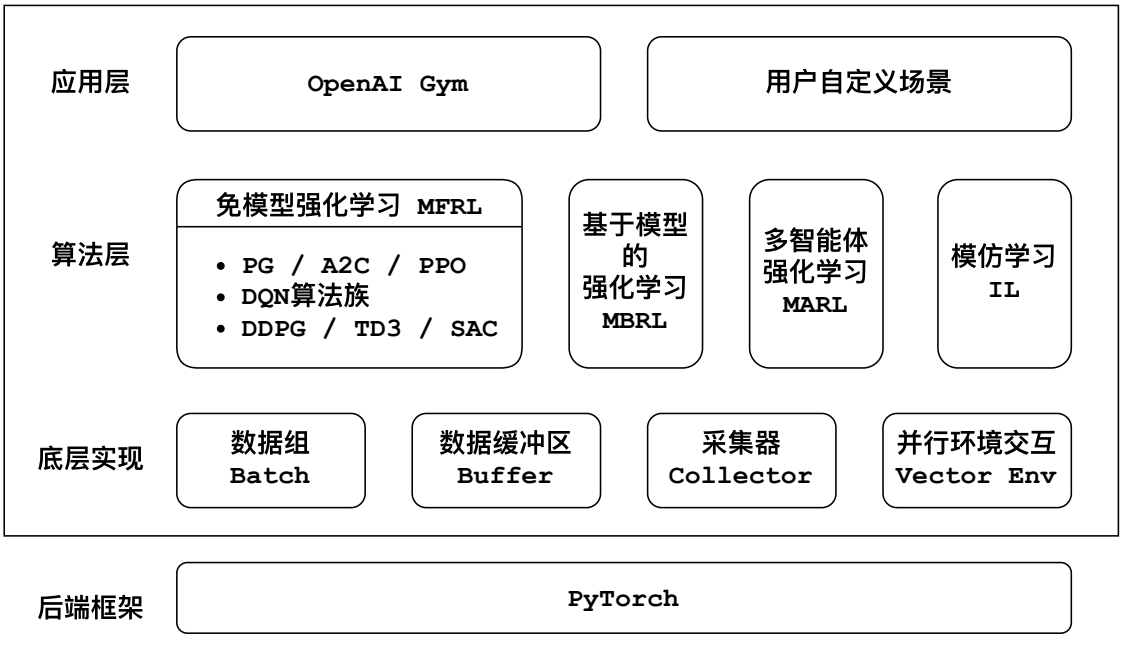


图 1.2　天授平台总体架构

本文描述了“天授”，一个基于 PyTorch 的深度强化学习算法平台。图 1.2 描述了该平台的总体架构。天授平台以 PyTorch 作为深度学习后端框架，将各个强化学习算法加以模块化，在数据层面抽象出了数据组（Batch）、数据缓冲区（Buffer）、

采集器（Collector）三个基本模块，实现了针对任意环境的并行交互与采样功能，算法层面支持丰富多样的强化学习算法，如免模型强化学习（MFRL）中的一系列算法、模仿学习算法（IL）等，从而能够让研究者方便地使用不同算法来测试不同场景。

天授拥有创新的模块化设计，简洁地实现了各种强化学习算法，支持了用户各种各样的需求。在相关的性能实验评测中，天授在众多强化学习平台夺得头筹。种种亮点使其获得了强化学习社区不小的关注度，在 GitHub 上开源不到短短一个月，星标就超过了基于 PyTorch 的另一个著名的强化学习平台 rlpyt[12]。

1.3.2 论文结构

接下来的论文结构安排如下所示：

第 2 章：描述了天授平台的设计与实现，将强化学习算法加以抽象凝练，分析提取出共有部分，介绍模块化的实现；以及介绍平台的其他特点。

第 3 章：描述了天授平台目前所支持的各类深度强化学习算法，介绍各个算法的基本原理以及在天授平台中的实现细节。

第 4 章：对比了天授平台与若干已有的著名深度强化学习平台的优劣之处，包括功能层面和性能层面的测试。

第 5 章：列举出了若干天授平台的典型使用样例，使读者能够进一步了解平台的接口和使用方法。

第 6 章：对天授平台特点进行总结，并指出后续的工作方向。

第 2 章　平台设计与实现

本章节首先介绍深度强化学习的核心问题及其形式化建模，随后介绍平台的整体架构、模块设计和实现细节，最后简要提及平台的外围支持。

2.1　深度强化学习问题描述

强化学习问题与机器学习领域中的监督学习问题、无监督学习问题都不同。如果给定一个输入 x，那么这三种学习类型的算法会有不同的输出：

- 监督学习：输出预测值 y；
- 无监督学习：输出隐变量 z；
- 强化学习：输出动作 a 使得期望累计奖励 $\mathbb{E}_{\pi}[\sum r]$ 最大。

强化学习算法是在不确定环境中，通过与环境的不断交互，来不断优化自身策略的算法。它和监督学习、非监督学习有着一些本质上的区别：

1. 获取的数据不独立同分布：大部分机器学习算法假设数据是独立同分布的，否则会有收敛性问题；然而智能体与环境进行交互产生的数据具有很强的时间相关性，无法在数据层面做到完全的解耦，不满足数据的独立同分布性质，因此强化学习算法训练并不稳定；智能体的行为同时会影响后续的数据分布；
2. 没有 “正确” 的行为，且无法立刻获得反馈：监督学习有专门的样本标签，而强化学习并没有类似的强监督信号，通常只有基于奖励函数的单一信号；强化学习场景存在延迟奖励的问题，智能体不能在单个样本中立即获得反馈，需要不断试错，还需要平衡短期奖励与长期奖励的权重；
3. 具有超人类的上限：传统的机器学习算法依赖人工标注好的数据，从中训练好的模型的性能上限是产生数据的模型（人类）的上限；而强化学习可以从零开始和环境进行不断地交互，可以不受人类先验知识的桎梏，从而能够在一些任务中获得超越人类的表现。

2.1.1　问题定义

强化学习问题是定义在马尔科夫决策过程（Markov Decision Process，MDP）之上的。一个 MDP 是形如 $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \rho_0 \rangle$ 的五元组，其中：

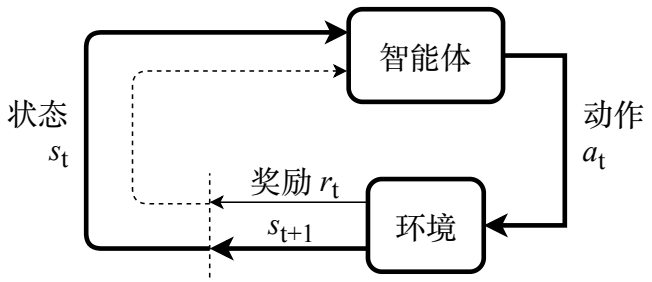


图 2.1　强化学习算法中智能体与环境循环交互的过程

- $\mathcal{S}$ 是所有合法状态的集合；
- $\mathcal{A}$ 是所有合法动作的集合；
- $\mathcal{R}: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 是奖励函数，t 时刻的奖励函数 r_t 由 s_t, a_t 决定，使用 R_s^a 表示在当前状态 s 采取动作 a 之后所能获得的期望奖励；
- $\mathcal{P}: \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ 是状态转移概率函数，使用 $P_{ss'}^a, \in \mathcal{P}$ 来表示当前状态 s 采取动作 a 转移到状态 s' 的概率；
- ρ_0 是初始状态的概率分布，$\sum_{s \in \mathcal{S}} \rho_0(s) = 1$。

MDP 还具有马尔科夫性质，即在任意 t 时刻，下一个状态 s_{t+1} 的概率分布只能由当前状态 s_t 决定，与过去的任何状态 $\{s_0, \cdots, s_{t-1}\}$ 均无关。

图 2.1 描述了在经典的强化学习场景中，智能体与环境不断交互的过程：在 t 时刻，智能体获得了环境状态 s_t，经过计算输出动作值 a_t 并在环境中执行，环境会返回 $t+1$ 时刻的环境状态 s_{t+1} 与上一个时刻产生的奖励 r_t。

在某些场景中，智能体无法获取到整个环境的状态，比如扑克、麻将等不完全信息对弈场景，此时称整个过程为部分可观测马尔科夫决策过程 (Partially Observable Markov Decision Process，POMDP)。在智能体与环境交互的每个回合中，智能体只能接收到观测值 o_t，为状态 s_t 的一部分。

定义累积折扣回报 G_t 为从 t 时刻起的加权奖励函数总和

$$G_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i \tag{2-1}$$

其中 $\gamma \in [0, 1]$ 是折扣因子，衡量了智能体对短期回报与长期回报的权重分配。通常用 $\pi_\theta(\cdot)$ 表示一个以参数 θ 参数化的策略 π。强化学习算法优化目标是最大化智能体每一回合的累计折扣回报的期望，形式化如下：

$$\theta^* = \arg\max_\theta \mathbb{E}_{\pi_\theta}[G_t] \tag{2-2}$$

2.1.2 智能体的组成

一个智能体主要由策略函数（Policy function）、价值函数（Value function）和环境模型（Environment model）三个部分组成。

策略函数：智能体根据当前环境状态，输出动作值的函数。策略函数分为确定性策略函数与随机性策略函数。

确定性策略函数通常有两种形式：（1）$a_t = \pi_\theta(s_t)$，直接输出动作值；（2）$a_t = \arg\max_a \pi_\theta(a|s_t)$，评估在状态 s_t 下所有可能的策略并从中选取最好的动作。

随机性策略函数的形式主要为计算一个概率分布 $\pi_\theta(a|s_t)$，从这个概率分布中采样出动作值 a_t。常用的分布有用于离散动作空间的类别分布（Categorical Distribution）、用于连续动作空间的对角高斯分布（Diagonal Gaussian Distribution）。

价值函数：价值函数是智能体对当前状态、或者是状态-动作对进行评估的函数，主要有三种形式：

1. 状态值函数（State-Value function）$V(s)$：状态 s 对应的期望累计折扣回报，$V(s) = \mathbb{E}_\pi[G_t|s_t = s]$；
2. 动作值函数（Action-Value function）$Q(s,a)$：状态 s 在采取动作 a 的时候对应的期望累计折扣回报，$Q(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]$
3. 优势函数（Advantage function）$A(s,a)$：状态 s 下采取动作 a 的情况下，比平均情况要好多少，$A(s,a) = Q(s,a) - V(s)$。

环境模型：智能体还可以对环境中的状态转移函数进行建模，比如使用映射 $\mathcal{F} : S \times \mathcal{A} \to S$ 进行对环境转移概率函数 $P^a_{ss'}$ 的拟合，或者是对奖励函数 R^a_s 的分布进行拟合。

2.1.3 现有深度强化学习算法分类

强化学习算法按照是否对环境进行建模来划分，可分为免模型强化学习（Model-free Reinforcement Learning，MFRL）与基于模型的强化学习（Model-based Reinforcement Learning，MBRL）两大类；此外还有多智能体学习（Multi-agent Reinforcement Learning，MARL）、元强化学习（Meta Reinforcement Learning）、模仿学习（Imitation Learning，IL）这几个大类。现有深度强化学习平台主要实现免模型强化学习算法。

免模型强化学习算法按照模型的学习特性进行区分，可分为同策略学习（On-policy learning）和异策略学习（Off-policy learning）。同策略学习指所有与环境交互采样出来的轨迹立即拿去训练策略，训练完毕之后即丢弃；而异策略学习指将

所有采集出来的数据存储在一个数据缓冲区中，训练策略时从缓冲区中采样出若干数据组进行训练。

2.2 深度强化学习问题的抽象凝练与平台整体设计

通过以上描述，现有强化学习算法可以被抽象成如下若干模块：

1. 数据缓冲区（Buffer）：无论是同策略学习，还是异策略学习方法，均需要将智能体与环境交互的数据进行封装与存储。例如在 DQN[1] 算法实现中，需要使用重放缓冲区（Replay Buffer）进行相应的数据处理，因此对数据存储的实现是平台底层不可或缺的一部分。

   更进一步，可以将同策略学习算法与异策略学习算法的数据存储用数据缓冲区（Buffer）进行统一：异策略学习算法是将缓冲区数据每次采样出一部分，而同策略学习算法可以看做一次性将缓冲区中所有数据采集出来并删除。
2. 策略（Policy）：策略是智能体决策的核心部分，将其形式化表示为

$$\pi_\theta(o_t, h_t) \Rightarrow (a_t, h_{t+1}, p_t) \tag{2-3}$$

   其中 h_t 是 t 时刻策略的隐藏层状态，通常用于循环神经网络（Recurrent Neural Network，RNN）的训练；p_t 是策略输出的中间值，以备后续训练时使用。

   此外不同策略在训练的时候所需要采样的数据模式不同，比如在计算 n 步回报的时候需要从数据缓冲区中采样出连续 n 帧的数据信息进行计算，因此策略需要有一个专门和数据缓冲区进行交互的接口。

   策略中还包含模型（Model），包括表格模型、神经网络策略模型、环境模型等。模型可直接与策略进行交互，而不必和其他部分相互耦合。
3. 采集器（Collector）：采集器定义了策略与环境（Env）交互的过程。策略在与一个或多个环境交互的过程中会产生一定的数据，由采集器进行收集并存放至数据缓冲区中；在训练策略的时候由采集器从数据缓冲区中采样出数据并进行封装。

   在多智能体的情况下，采集器可以承担多个策略之间的交互，并分别存储至不同的数据缓冲区中。
4. 训练器（Trainer）：训练器是平台最上层的封装，定义了整个训练过程，与

采集器和策略的学习函数进行交互，包含同策略学习与异策略学习两种训练模式。

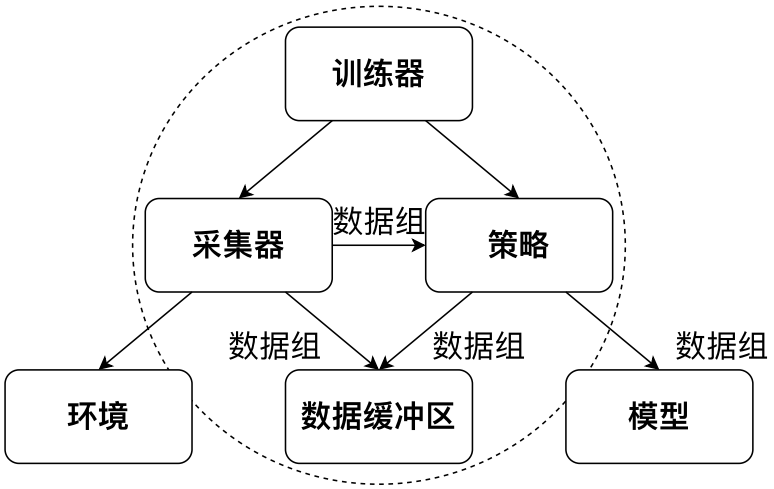


图 2.2　深度强化学习算法模块抽象凝练

图 2.2 较为直观地描述了上述抽象出的若干模块相互之间的调用关系。其中“数据组” 为模块之间传递数据信息的封装。平台的整体架构即按照该抽象模式进行设计，其中虚线圈出的部分为平台核心模块。

2.3　平台实现

2.3.1　数据组（Batch）

数据组是平台内部各个模块传递数据的数据结构。它支持任意关键字初始化、对任意元素进行修改，还支持嵌套调用和格式化输出的功能。如果数据组内各个元素值的第 0 维大小相等，还可支持切分（split）操作，从而方便地将一组大数据按照固定的大小拆分之后送入策略中处理。

平台的内部实现对数据组保留了如下 7 个关键字：

- `obs`：t 时刻的观测值 o_t；
- `act`：t 时刻策略采取的动作值 a_t；
- `rew`：t 时刻环境反馈的奖励值 r_t；
- `done`：t 时刻环境结束标识符 $d_t \in \{0,1\}$，0 为未结束，1 为结束；
- `obs_next`：$t+1$ 时刻的观测值 o_{t+1}；
- `info`：t 时刻环境给出的环境额外信息 i_t，以字典形式存储；
- `policy`：t 时刻策略在计算过程中产生的数据 p_t，可参考公式 2-3。

2.3.2 数据缓冲区（Buffer）

数据缓冲区存储了策略与环境交互产生的一系列数据，并且支持从已存储数据中采样出固定大小的数据组进行策略学习。底层数据结构主要采用 NumPy 数组进行存储，能够加快存储效率。

同数据组一样，数据缓冲区同样保留了其中 7 个保留关键字，其中关键字 `info` 不改变其中的数据结构，即在 NumPy 数组中仍然使用字典格式进行存储。在采样时，如果传入大小是 0，则返回整个缓冲区中的所有数据，以支持在同略学习算法的训练需求。

目前数据缓冲区的类型有：最基本的重放缓冲区（Replay Buffer），使用列表作为底层数据结构的列表缓冲区（List Replay Buffer）、优先级经验重放缓冲区（Prioritized Replay Buffer）支持优先权重采样。此外数据缓冲区还支持历史数据堆叠采样（例如给定采样时间下标 t 和堆叠帧数 n，返回堆叠的观测值 $\{o_{t-n+1}, \cdots, o_t\}$）和多模态数据存储（需要存储的数据可以是一个字典）。在将来还将会支持事后经验回放算法[23]（Hindsight Experience Replay，HER）。

2.3.3 环境（Env）

环境接口遵循 OpenAI Gym[21] 定义的通用接口，即每次调用 `step` 函数时，需要输入一个动作 a_t，返回一个四元组：下一个观测值 o_{t+1}、这个时刻采取动作值 a_t 所获得的奖励 r_t、环境结束标识符 d_t、以及环境返回的其他信息 i_t。

为了能让所有强化学习算法支持并行环境采样，天授封装了几个不同的向量化环境类，可以单线程循环执行每个环境，也可以多线程同时执行。每次调用 `step` 函数的语义和之前定义一致，区别在于增加了一步将所有信息堆叠起来组成一个 NumPy 数组的操作，并以第 0 个维度来区分是哪个环境产生的数据。

2.3.4 策略（Policy）

策略是强化学习算法的核心。智能体除了需要做出决策，还需不断地学习来自我改进。通过章节 2.2 对策略的抽象描述，可以将其拆分为 4 个模块：

1. `__init__`：策略的初始化，比如初始化自定义的模型（Model）、创建目标网络（Target Network）等；
2. `forward`：从给定的观测值 o_t 中计算出动作值 a_t，在图 2.2 中对应策略到模型的调用；
3. `process_fn`：在获取训练数据之前和数据缓冲区进行交互，在图 2.2 中对应策略到数据缓冲区的调用；

4. `learn`：使用一个数据组进行策略的更新训练，在图 2.2 中对应训练器到策略的调用。

不同算法中策略的具体实现将在第 3 章中进行详细分析讲解。

2.3.5 采集器（Collector）

采集器定义了策略与环境交互的过程。采集器主要包含以下两个函数：

1. `collect`：让给定的策略和环境交互**至少** n_s 步或者至少 n_e 轮，并将交互过程中产生的数据存储进数据缓冲区中；
2. `sample`：从数据缓冲区中采集出给定大小的数据组，准备后续的策略训练。

为了支持并行环境采样，采集器采用了缓存数据缓冲区，即同时和多个环境进行交互并将数据存储在对应的缓存区中，一旦有一个环境的交互结束，则将对应缓存区的数据取出，存放至主数据缓冲区中。由于无法精确控制环境交互的结束时间，采集的数据量有可能会多于给定数值，因此在采集中此处强调“至少”。

采集器理论上还可以支持多智能体强化学习的交互过程，将不同的数据缓冲区和不同策略联系起来，即可进行交互与数据采样。

2.3.6 训练器（Trainer）

训练器负责最上层训练逻辑的控制，例如训练多少次之后进行策略和环境的交互。现有的训练器包括同策略学习训练器（On-policy Trainer）和异策略学习训练器（Off-policy Trainer）。

平台未显式地将训练器抽象成一个类，因为在其他现有平台中都将类似训练器的实现抽象封装成一个类，导致用户难以二次开发。因此以函数的方式实现训练器，并提供了示例代码便于研究者进行定制化训练策略的开发。

2.3.7 算法伪代码与对应解释

接下来将通过一段伪代码的讲解来阐释上述所有抽象模块的应用。

```
s = env.reset()
buf = Buffer(size=10000)
agent = DQN()
for i in range(int(1e6)):
    a = agent.compute_action(s)
    s_, r, d, _ = env.step(a)
    buf.store(s, a, s_, r, d)
    s = s_
    if i % 1000 == 0:
        bs, ba, bs_, br, bd = buf.get(size=64)
```

```
11          bret = calc_return(2, buf, br, bd, ...)
12          agent.update(bs, ba, bs_, br, bd, bret)
```

以上伪代码描述了一个定制化两步回报 DQN 算法的训练过程。表 2.1 描述了伪代码的解释与上述各个模块的具体对应关系。

表 2.1　伪代码与天授模块具体对应关系

行	伪代码	解释	对应天授平台实现
1	s = env.reset()	环境初始化	在 Env 中实现
2	buf = Buffer(size=10000)	数据缓冲区初始化	buf = ReplayBuffer(size=10000)
3	agent = DQN()	策略初始化	policy.__init__(...)
4	for i in range(int(1e6)):	描述训练过程	在 Trainer 中实现
5	a = agent.compute_action(s)	计算动作值	policy(batch, ...)
6	s_, r, d, _ = env.step(a)	与环境交互	collector.collect(...)
7	buf.store(s, a, s_, r, d)	将交互过程中产生的数据存储到数据缓冲区中	collector.collect(...)
8	s = s_	更新观测值	collector.collect(...)
9	if i % 1000 == 0:	每一千步更新策略	在 Trainer 中实现
10	bs, ba, bs_, br, bd = buf.get(size=64)	从数据缓冲区中采样出数据	collector.sample(size=64)
11	bret = calc_return(2, buf, br, bd, ...)	计算两步回报	policy.process_fn(batch, buffer, indice)
12	agent.update(bs, ba, bs_, br, bd, bret)	训练智能体	policy.learn(batch, ...)

2.4　平台外围支持

2.4.1　命名由来

该强化学习平台被命名为“天授”。天授的字面含义是上天所授，引申含义为与生俱来的天赋。强化学习算法是不断与环境交互进行学习，在这个过程中没有人类的干预。取名“天授”是为了表明智能体没有向所谓的“老师”取经，而是通过与环境的不断交互自学成才。图 2.3 展示了天授平台的标志，左侧采用渐变颜色融合了青铜文明元素，是一个大写的字母“T”，右侧是天授拼音。

图 2.3　天授平台标志

2.4.2　文档教程

天授提供了一系列针对平台的文档和教程，使用 ReadTheDocs[①]第三方平台进行自动部署与托管服务。目前部署在 https://tianshou.readthedocs.io/ 中，预览页面如图 2.4 所示。值得一提的是，天授的中文文档现在已经上线，部署于 https://tianshou.readthedocs.io/zh/latest/。

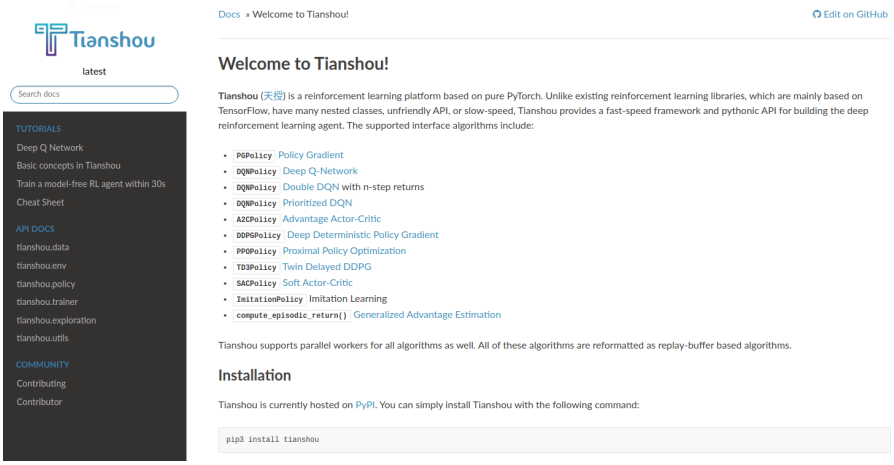


图 2.4　天授文档页面

2.4.3　单元测试

天授具有较为完善的单元测试，使用 GitHub Actions[②]进行持续集成。在每次单元测试中，均包含代码风格测试、功能测试和性能测试三个部分，其中性能测试是对所有天授平台中实现的强化学习算法进行整个过程的完整训练和测试，一旦没有在规定的训练限制条件内达到能够解决对应问题的效果，则不予通过测试。

目前天授平台的单元测试代码覆盖率达到了 85%，可以在第三方网站 https://codecov.io/gh/thu-ml/tianshou 中查看详细情况。图 2.5 展示了天授某次单元测试

① https://readthedocs.org/
② https://help.github.com/cn/actions

的具体结果。

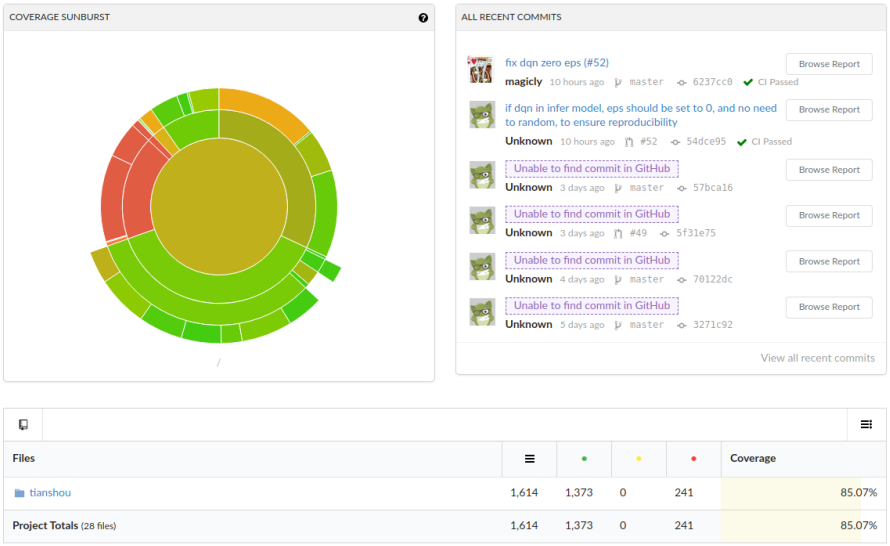


图 2.5　天授单元测试结果

2.4.4　发布渠道

目前天授平台的发布渠道为 PyPI[①]，是一个 Python 编程语言的第三方软件存储库。用户可以通过直接运行命令

```
pip install tianshou
```

进行平台的安装，十分方便。图 2.6 显示了天授在 PyPI 平台的发布界面。在未来，天授平台还将会添加另一个第三方软件存储库 Conda[②]的发布渠道。

2.5　小结

本章节介绍了深度强化学习的基本定义与问题描述，将各种不同的强化学习算法进行模块化抽象，并据此阐述了平台各个模块的实现，最后简单点明了平台的其他特点。

① https://pypi.org/
② https://anaconda.org/anaconda/conda

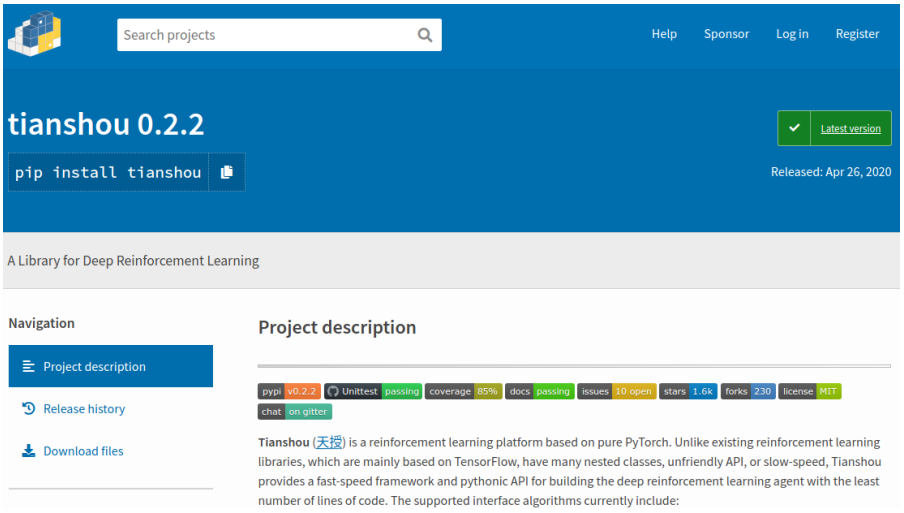


图 2.6　天授在 PyPI 平台的发布界面

第 3 章　平台支持的深度强化学习算法

本章节将依次介绍天授平台所实现的强化学习算法的原理，以及这些算法在平台内部的具体实现细节。

强化学习的主要目标是让智能体学会一个能够最大化累计奖励的策略。以下仍然使用符号 $\pi_\theta(\cdot)$ 表示一个使用参数 θ 参数化的策略 π，优化目标为最大化 $J(\theta)$，定义为：

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t] = \sum_{s\in S} d^\pi(s)V^\pi(s) = \sum_{s\in S} d^\pi(s) \sum_{a\in \mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a) \tag{3-1}$$

其中 $d^\pi(s)$ 是使用策略 π 在马尔科夫链中达到状态 s 的概率。为力求简洁直观地表述各个算法，下文尽量不进行复杂的数学公式推导。此外，下文在描述算法时通常使用状态值 s_t，在描述具体实现时通常使用观测值 o_t。

3.1　基于策略梯度的深度强化学习算法

3.1.1　策略梯度（PG）

策略梯度算法（Policy Gradient[24]，又称 REINFORCE 算法、蒙特卡洛策略梯度算法，以下简称 PG）是一个较为直观简洁的强化学习算法。它于上世纪九十年代被提出，依靠蒙特卡洛采样直接进行对累计折扣回报的估计，并直接以公式 3-1 对 θ 进行求导，可推出梯度为：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[G_t \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{3-2}$$

其中 a_t、s_t 均为具体采样值。将公式 3-2 反推为目标函数

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t \log \pi_\theta(a_t|s_t)] \tag{3-3}$$

可以发现策略梯度算法本质上是最大化好动作的概率[①]。因此实现 PG 算法只需求得累计回报 G_t、每次采样的数据点在策略函数中的对数概率 $\log \pi_\theta(a_t|s_t)$ 之后即可对参数 θ 进行求导，从而使用梯度上升方法更新模型参数。

① 这个视频详细地讲解了策略梯度算法的推导过程：https://youtu.be/XGmd3wcyDg8

一个被广泛使用的变种版本是在算法中将 G_t 减去一个基准值，在保证不改变偏差的情况下尽可能减小梯度估计的方差。比如减去一个平均值，或者是如果使用状态值函数 $V(s)$ 作为一个基准，那么实际所使用的即为优势函数 $A(s,a) = Q(s,a) - V(s)$，为动作值函数与状态值函数的差值。这将在后续描述的算法中进行使用。

策略梯度算法在天授中的实现十分简单：

- `process_fn`：计算 G_t，具体实现位于章节 3.1.4；
- `forward`：给定 o_t 计算动作的概率分布，并从其中进行采样返回；
- `learn`：按照公式 3-3 计算 G_t 与动作的对数概率 $\log \pi_\theta(a_t|o_t)$ 的乘积，求导之后进行反向传播与梯度上升，优化参数 θ；
- 采样策略：使用同策略的方法进行采样。

3.1.2 优势动作评价（A2C）

优势动作评价算法（Advantage Actor-Critic[25]，又被译作优势演员-评论家算法，以下简称 A2C）是对策略梯度算法的一个改进。简单来说，策略梯度算法相当于 A2C 算法中评价网络输出恒为 0 的版本。该算法从公式 3-3 改进如下：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\hat{A}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{3-4}$$

其中 $\hat{A}(s_t,a_t)$ 为估计的优势函数，具体定义以及实现见章节 3.1.4。$\hat{V}(s_t)$ 为评价网络输出的状态值函数。此外为了让评价网络的输出尽可能接近真实的状态值函数，在优化过程中还加上了对应的均方误差项；此外标准的实现中还有关于策略分布的熵正则化项。因此汇总的 A2C 目标函数为：

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\hat{A}(s_t,a_t)\log \pi_\theta(a_t|s_t) - c_1(\hat{V}(s_t) - G_t)^2 + c_2 H(\pi_\theta(\cdot)|s_t)\right] \tag{3-5}$$

其中 c_1, c_2 是前述两项的对应超参数。

A2C 最大的特点就是支持同步的并行采样训练，但由于天授平台支持所有算法的并行环境采样，此处不再赘述。此外 A2C 相比于异步策略执行版本 A3C 而言，避免了算法中策略执行不一致的问题，具有更快的收敛速度。

A2C 算法在天授中的实现如下：

- `process_fn`：计算 $\hat{A}(s_t,a_t)$，具体实现位于章节 3.1.4；
- `forward`：和策略梯度算法一致，给定观测值 o_t，计算输出的输出策略的概率分布，并从中采样；

- `learn`：按照公式 3-5 计算目标函数并求导更新参数；
- 采样策略：使用同策略的方法进行采样。

3.1.3 近端策略优化（PPO）

近端策略优化算法（Proximal Policy Optimization[4]，以下简称 PPO）是信任区域策略优化算法（Trust Region Policy Optimization[26]，TRPO）的简化版本。由于策略梯度算法对超参数较为敏感，二者对策略的更新进行了一定程度上的限制，避免策略性能在参数更新前后产生剧烈变化，从而导致采样效率低下等问题。

PPO 算法通过计算更新参数前后两次策略的比值来确保这个限制。具体目标函数为

$$J^{\mathrm{CLIP}}(\theta) = \mathbb{E}_{\pi_\theta}\left[\min\left(r(\theta)\hat{A}_{\theta_{\mathrm{old}}}(s_t,a_t), \mathrm{clip}(r(\theta),1-\epsilon,1+\epsilon)\hat{A}_{\theta_{\mathrm{old}}}(s_t,a_t)\right)\right] \tag{3-6}$$

其中 $\hat{A}(\cdot)$ 表示估计的优势函数，因为真实的优势函数无法从训练过程所得数据中进行精确计算；$r(\theta)$ 是重要性采样权重，定义为新策略与旧策略的概率比值

$$r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)} \tag{3-7}$$

函数 $\mathrm{clip}(r(\theta),1-\epsilon,1+\epsilon)$ 将策略的比值 $r(\theta)$ 限制在 $[1-\epsilon,1+\epsilon]$ 之间，从而避免了策略性能上的剧烈变化。在将 PPO 算法运用在动作评价（Actor-Critic）架构上时，与 A2C 算法类似，目标函数通常会加入状态值函数项与熵正则化项

$$J(\theta) = \mathbb{E}_{\pi_\theta}[J^{\mathrm{CLIP}}(\theta) - c_1(\hat{V}(s_t) - G_t)^2 + c_2 H(\pi_\theta(\cdot)|s_t)] \tag{3-8}$$

其中 c_1, c_2 为两个超参数，分别对应状态值函数估计与熵正则化两项。

天授中的 PPO 算法实现大致逻辑与 A2C 十分类似：

- `process_fn`：计算 $\hat{A}(s_t,a_t)$ 与 G_t，具体实现位于章节 3.1.4；
- `forward`：按照给定的观测值 o_t 计算概率分布，并从中采样出动作 a_t；
- `learn`：重新计算每个数据组所对应的对数概率，并按照公式 3-8 进行目标函数的计算；
- 采样策略：使用同策略的方法进行采样。

3.1.4 广义优势函数估计器（GAE）

广义优势函数估计器（Generalized Advantage Estimator[27]，以下简称 GAE）是将以上若干种策略梯度算法的优势函数的估计 $\hat{A}(s_t,a_t)$ 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：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{3-9}$$

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 $\Psi_t = \hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$；PPO 中是 $\Psi_t = \hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$，其中 δ_t 是时序差分误差项（Temporal Difference error，TD 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：

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} = \sum_{l=0}^{\infty} (\gamma\lambda)^l (r_t + \gamma V(s_{t+l+1}) - V(s_{t+l})) \tag{3-10}$$

其中 GAE$(\gamma, 0)$ 的情况为 $\hat{A}_t = \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$，为 1 步时序差分误差，GAE$(\gamma, 1)$ 的情况为 $\hat{A}_t = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t)$，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。

天授中 GAE 实现与其他平台有一些不同之处。比如在 OpenAI Baselines[9] 的实现中，对每个完整轨迹的最后一帧进行特殊判断处理。与此不同，天授使用轨迹中每项的下一时刻观测值 o_{t+1} 批量计算状态值函数，避免了特殊判断。天授的 GAE 实现将大部分操作进行向量化，并且支持同时计算多个完整轨迹的 GAE 函数，还比 Baselines 正常使用 Python 写的循环语句要快不少。

3.2 基于 Q 价值函数的深度强化学习算法

3.2.1 深度 Q 网络（DQN）

深度 Q 网络算法（Deep Q Network[1]，以下简称 DQN）是强化学习算法中最经典的算法之一，它在 Atari 游戏中表现一鸣惊人，由此开启了深度强化学习的新一轮浪潮。DQN 算法核心是维护 Q 函数并使用它进行决策。具体而言，$Q^\pi(s,a)$ 为在该策略 π 下的动作值函数；每次到达一个状态 s_t 之后，遍历整个动作空间，将动作值函数最大的动作作为策略：

$$a_t = \arg\max_a Q^\pi(s_t, a) \tag{3-11}$$

其动作值函数的更新采用贝尔曼方程进行迭代

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha_t(r_t + \gamma \max_a Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t)) \tag{3-12}$$

其中 α 为学习率。通常在简单任务上，使用全连接神经网络来拟合 Q^π，但是在稍微复杂一点的任务上如 Atari 游戏，会使用卷积神经网络进行由图像到值函数的映射拟合，这也是深度 Q 网络中 “深度” 一词的由来。由于这种表达形式只能处理有限个动作值，因此 DQN 通常被用在离散动作空间任务中。

为了避免陷入局部最优解，DQN 算法通常采用 ϵ-贪心方法进行策略探索，即每次有 $\epsilon \in [0,1]$ 的概率输出随机策略，$1-\epsilon$ 的概率输出使用动作值函数估计的最优策略；此外通常把公式 3-12 中 $r_t + \gamma \max_a Q^\pi(s_{t+1}, a)$ 一项称作目标动作值函数 Q_{target}，它还可以拓展成不同的形式，比如 n 步估计：

$$Q^n_{\text{target}}(s_t, a_t) = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \max_a \gamma^n Q^\pi(s_{t+n}, a) \tag{3-13}$$

天授中的 DQN 算法实现如下：

- `process_fn`：使用公式 3-13 计算目标动作函数，与重放缓冲区交互进行计算；
- `forward`：给定观测值 o_t，输出每个动作对应的动作值函数 $Q(o_t, \cdot)$，并使用 ϵ-贪心算法添加噪声，输出动作 a_t；
- `learn`：使用公式 3-12 进行迭代，在特定时刻可调整 ϵ-贪心算法中的 ϵ 值；
- 采样策略：使用异策略的方法进行采样。

3.2.2 双网络深度 Q 学习（DDQN）

双网络深度 Q 学习算法（Double DQN[28]，以下简称 DDQN）是 DQN 算法的重要改进之一。由于在公式 3-12 中使用同一个动作值函数进行对目标动作值函数的估计，会导致策略网络产生过于乐观的估计，从而降低了算法的采样效率。DDQN 算法将动作评估与动作选择进行解耦，从而减少高估所带来的负面影响。它将公式 3-12 中的目标动作值函数加以改造如下

$$Q_{\text{target}}(s_t, a_t) = r_t + \gamma Q^{\pi_{\text{old}}}\left(s_{t+1}, \arg\max_a Q^\pi(s_{t+1}, a)\right) \tag{3-14}$$

其中 $Q^{\pi_{\text{old}}}$ 是目标网络（Target Network），为策略网络 Q^π 的历史版本，专门用来进行动作评估。公式 3-14 同样可以和公式 3-13 进行结合，推广到 n 步估计的情况，此处不再赘述。

由于 DDQN 与 DQN 仅有细微区别，因此在天授的实现中将二者封装在同一个类中，改动如下：

- `process_fn`：按照公式 3-14 计算目标动作函数；

- `learn`：在需要的时候更新目标网络的参数。

3.2.3 优先级经验重放（PER）

优先级经验重放（Prioritized Experience Replay[29]，以下简称 PER）是 DQN 算法的另一个重要改进。该算法也可应用在之后的 DDPG 算法族中。其核心思想是，根据策略网络输出的动作值函数 $Q^{\pi}(s_t, a_t)$ 与实际采样估计的动作值函数 $Q_{\text{target}}(s_t, a_t)$ 的时序差分误差来给每个样本不同的采样权重，将误差更大的数据能够以更大的概率被采样到，从而提高算法的采样与学习效率。

PER 的实现不太依赖于算法层的改动，比较和底层的重放缓冲区相关。相关改动如下：

- 算法层：加入一个接口，传出时序差分误差，作为优先经验重放缓冲区的更新权重；
- 数据层：新建优先经验重放缓冲区类，继承自重放缓冲区类，修改采样函数，并添加更新优先值权重的函数。

3.3 综合 Q 价值函数与策略梯度的深度强化学习算法

3.3.1 深度确定性策略梯度（DDPG）

深度确定性策略梯度算法（Deep Deterministic Policy Gradient[30]，以下简称 DDPG）是一种同时学习确定性策略函数 $\pi_{\theta}(s)$ 和动作值函数 $Q^{\pi}(s, a)$ 的算法。它主要解决的是连续动作空间内的策略训练问题。在 DQN 中，由于常规的 Q 函数只能接受可数个动作，因此无法拓展到连续动作空间中。

DDPG 算法假设动作值函数 $Q(s, a)$ 在连续动作空间中是可微的，将动作值 a 用一个函数 $\pi_{\theta}(s)$ 拟合表示，并将 $\pi_{\theta}(s)$ 称作动作网络，$Q^{\pi}(s, a)$ 称作评价网络。DDPG 算法评价网络的更新部分与 DQN 算法类似，动作网络的更新根据确定性策略梯度定理[31]，直接对目标函数 $Q^{\pi}(s, \pi_{\theta}(s))$ 进行梯度上升优化即可。

为了更好地进行探索，原始 DDPG 算法添加了由 Ornstein-Uhlenbeck 随机过程①产生的时间相关的噪声项，但在实际测试中，高斯噪声可以达到与其同样的效果[32]；DDPG 还采用了目标网络以稳定训练过程，对目标动作网络和目标评价网络进行参数软更新，即 $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$，以 τ 的比例将新网络的权重 θ 更新至目标网络 θ' 中。

天授对 DDPG 算法的实现如下：

① https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process

- `process_fn`：和 DQN 算法类似，其中动作 a 不进行全空间遍历，而是以动作网络的输出作为参考标准；
- `forward`：给定观测值 o_t，输出动作 $a_t = \pi_\theta(o_t)$，并添加噪声项；
- `learn`：分别计算贝尔曼误差项和 $Q^\pi(s, \pi_\theta(s))$ 并分别优化，之后软更新目标网络的参数；
- 采样策略：使用异策略的方法进行采样。

3.3.2 双延迟深度确定性策略梯度（TD3）

双延迟深度确定性策略梯度算法（Twin Delayed DDPG[32]，以下简称 TD3）是 DDPG 算法的改进版本。学习动作值函数 Q 的一系列方法一直以来都有过度估计的问题，DDPG 也不例外。TD3 算法做了如下几点对 DDPG 进行改进：

- 截断双网络 Q 学习：截断双网络 Q 学习使用两个动作值网络，取二者中的最小值作为动作值函数 Q 的估计，从而有利于减少过度估计：

$$Q_{\text{target}_i} = r + \min_{j=1,2} Q^\pi_{\phi_j}(s', \pi_\theta(s')) \tag{3-15}$$

- 动作网络延迟更新：相关实验结果表明，同步训练动作网络和评价网络，却不使用目标网络，会导致训练过程不稳定；但是仅固定动作网络时，评价网络往往能够收敛到正确的结果。因此 TD3 算法以较低的频率更新动作网络，较高频率更新评价网络，通常每两次更新评价网络时，进行一次策略更新。
- 平滑目标策略：TD3 算法在动作中加入截断高斯分布产生的随机噪声，避免策略函数 $\pi_\theta(s)$ 陷入 Q 函数的极值点，从而更有利于收敛：

$$Q_{\text{target}} = r + \gamma Q^\pi(s', \pi_\theta(s') + \epsilon) \tag{3-16}$$

$$\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c) \tag{3-17}$$

与 DDPG 算法类似，天授在 TD3 的实现中继承了 DDPG 算法，只修改了 `learn` 部分，按照上述三点一一实现代码。

3.3.3 软动作评价（SAC）

软动作评价算法（Soft Actor-Critic[33]，以下简称 SAC）是基于最大熵强化学习理论提出的一个算法。SAC 算法同时具备稳定性好和采样效率高的优点，容易实现，同时融合了动作评价框架、异策略学习框架和最大熵强化学习框架，因此成为强化学习算法中继 PPO 之后的标杆算法。

SAC 的算法结构和 TD3 也十分类似，同样拥有一个动作网络和两个评价网络。单从最终推导得到的式子来看，和 TD3 的最大差别是在求目标动作值函数的时候，最后一项加上了较为复杂的熵正则化项，其余的实现十分类似。具体的推导可以在原论文中找到。

由于 SAC 的实现和 TD3 十分类似，故此处不再对其进行详细阐述。

3.4 部分可观测马尔科夫决策过程的训练

在实际场景中，智能体往往难以观测到环境中所有的信息，只能观测到状态 s 的一个子集 o 进行决策，这种场景被称作部分可观测马尔科夫决策过程（Partially Observable Markov Decision Process，简称 POMDP）。

POMDP 在深度强化学习领域通常有两种解决方案：（1）将过去一段时间内的信息（如过去的观测值、过去的动作和奖励）添加到当前状态中，按照常规方式进行处理；（2）将过去的信息利用循环神经网络（RNN）存储到中间状态中，可以传给后续状态进行使用。

第一种方法只需在重放缓冲区中添加时序采样功能，比如待采样下标是 t，需要采样连续 n 帧，那么在重放缓冲区中进行一定设置，返回观测值 $\{o_{t-n+1}, \cdots, o_{t-1}, o_t\}$，剩下的过程和正常的强化学习训练过程无异。第二种方法需要在第一种方法的基础上，在所有和神经网络相关的接口中添加对中间状态的支持。天授已经支持上述两种方法的实现。

3.5 模仿学习

模仿学习（Imitation Learning）更偏向于监督学习与半监督学习的范畴。它的核心思想是学习已有的数据，尽可能地还原产生这些数据的原始策略。比如给定一些 t 时刻的状态与动作数据对 (s_t, a_t)，那么可以使用神经网络来回归映射 $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{A}$，从而进行模仿学习。更进一步地，还有逆强化学习（Inverse Reinforcement Learning[34]，IRL）和生成式对抗模仿学习（Generative Adversarial Imitation Learning[35]，GAIL）等算法。

目前天授平台实现了最基本的模仿学习算法，具体实现如下：

- 连续动作空间：将其看作回归任务，直接对给定的动作进行回归；
- 离散动作空间：将其看作分类任务，最大化采取给定动作的概率；
- 采样策略：使用参考策略和异策略方法进行不断地采样补充数据。

3.6 小结

本章节介绍了深度强化学习算法的原理以及在天授平台上的具体实现，包括了 9 种免模型强化学习算法、循环神经网络模型训练和模仿学习。

第 4 章　平台对比评测

在本章节中，天授将与一些著名的强化学习算法平台进行对比评测。

4.1　实验设定说明

我们在众多强化学习平台中选取了 5 个具有代表性的平台：RLlib[11]、OpenAI Baselines[9]、PyTorch-DRL[19]、Stable-Baselines[17]（以下简写为 SB）、rlpyt[12]，选取评测的平台列表和原因如表 4.1 所示。

表 4.1　选取对比评测的平台一览。其中，评测 ID 为该平台发布的最新版本名称，若无已发布版本，则选取最新的提交记录 ID。

平台名称	评测 ID	选取原因
RLlib[11]	`0.8.5`	GitHub 深度强化学习平台星标数目最多，算法实现全面
Baselines[9]	`ea25b9e`	GitHub 深度强化学习平台星标数目第二多
PyTorch-DRL[19]	`49b5ec0`	GitHub PyTorch 深度强化学习平台星标数目第一多，算法实现全面
Stable-Baselines[17]	`2.10.0`	Baselines 的改进版本，算法实现全面，提供了一系列调优的参数
rlpyt[12]	`668290d`	GitHub PyTorch 深度强化学习平台星标数目第四多，算法实现全面
天授	`57bca16`	-

4.2　功能对比

接下来将从算法支持、模块化、定制化、单元测试和文档教程这五个维度来对包括天授在内的 6 个深度强化学习平台进行功能维度上的对比评测。

4.2.1　算法支持

深度强化学习算法主要分为免模型强化学习（MFRL）、基于模型的强化学习（MBRL）、多智能体学习（MARL）、模仿学习（IL）等。此外根据所解决的

问题分类，还可分为马尔科夫决策过程（MDP）和部分可观测马尔科夫决策过程（POMDP），其中 POMDP 要求策略网络支持循环神经网络（RNN）的训练。如今研究者们使用最多的是免模型强化学习算法，以下会对其详细对比。

4.2.1.1 免模型强化学习算法

免模型强化学习算法主要分为基于策略梯度的算法、基于价值函数的算法和二者结合的算法。现在深度强化学习社区公认的强化学习经典算法有：（1）基于价值函数：DQN[1] 及其改进版本 Double-DQN[28]（DDQN）、DQN 优先级经验重放[29]（PDQN）；（2）基于策略梯度：PG[24]、A2C[25]、PPO[4]；（3）二者结合：DDPG[30]、TD3[32]、SAC[33]。

各个平台实现算法的程度如表 4.2 所示。可以看出，大部分平台支持的算法种类是全面的，有些平台如 Baselines 支持的算法类型并不全面。天授平台支持所有的这些算法。

表 4.2 各平台支持的免模型深度强化学习算法一览

平台与算法	DQN	DDQN	PDQN	PG	A2C	PPO	DDPG	TD3	SAC	总计
RLlib	√	√	√	√	√	√	√	√	√	**9**
Baselines	√	×	√	×	√	√	√	×	×	5
PyTorch-DRL	√	√	√	√	√	√	√	√	√	**9**
SB	√	√	√	×	√	√	√	√	√	8
rlpyt	√	√	√	√	√	√	√	√	√	**9**
天授	√	√	√	√	√	√	√	√	√	**9**

4.2.1.2 其他类型强化学习算法

其他类型的强化学习算法包括基于模型的强化学习（MBRL）、多智能体强化学习（MARL）、元强化学习（MetaRL）、模仿学习（IL）。表 4.3 列出了各个平台的支持情况。可以看出，少有平台支持所有这些类型的算法。天授支持了模仿学习，但值得一提的是，基于模型的强化学习算法和多智能体强化学习算法都可以在现有的平台接口上完整实现。我们正在努力实现天授平台的 MBRL 和 MARL 算法中。

表 4.3　各平台支持的其他类型强化学习算法一览

平台与算法类型	MBRL	MARL	MetaRL	IL
RLlib	√	√	×	×
Baselines	×	×	×	√
PyTorch-DRL	×	×	×	×
SB	×	×	×	√
rlpyt	×	×	×	×
天授	×	×	×	√

表 4.4　各平台对 RNN 的支持

平台	RNN
RLlib	√
Baselines	×
PyTorch-DRL	×
SB	×
rlpyt	√
天授	√

4.2.1.3　循环状态策略

针对不完全信息观测的马尔科夫决策过程（POMDP），通常有两种处理方式：第一种是直接当作完全信息模式处理，但可能会导致一些诸如收敛性难以保证的问题；第二种是在智能体中维护一个内部状态，具体而言，将循环神经网络模型（RNN）融合到策略网络中。表 4.4 列出了各个平台对循环神经网络的支持程度。从表 4.4 中可以看出，部分平台对 RNN 的支持程度并不大。天授平台中所有算法均支持 RNN 网络，还支持获取历史状态、历史动作和历史奖励，以及其他用户或者环境定义的变量的历史记录。

4.2.2　并行环境采样

最初的强化学习算法仅是单个智能体和单个环境进行交互，这样的话采样效率较低，因为每一次网络前向都只能以单个样本进行计算，无法充分利用批处理加速的优势，从而导致了强化学习即使在简单场景中训练速度仍然较慢的问题。解决的方案是并行环境采样：智能体每次与若干个环境同时进行交互，将神经网络的前向数据量加大但又不增加推理时间，从而做到采样速率是之前的数倍。表 4.5 显示了各个平台、各个算法支持的并行环境采样的情况。从表 4.5 中可以看出，只有 RLlib、rlpyt、天授全面地支持了各种算法的并行环境采样功能，剩下的平台要么缺失部分算法实现、要么缺失部分算法的并行环境采样功能。这对于强化学习智能体的训练而言，性能方面可能会大打折扣。

4.2.3　模块化

模块化的强化学习算法框架能够让开发者以更少的代码量来更简单地实现新功能，在增加了代码的可重用性的同时也减少了出错的可能性。表 4.6 列出了各个平台模块化的详细情况。从表中可以看出，除 Baselines、Stable-Baselines 和

表 4.5　各平台各免模型深度强化学习算法支持并行环境采样情况一览

平台与算法	DQN	DDQN	PDQN	PG	A2C	PPO	DDPG	TD3	SAC
RLlib	√	√	√	√	√	√	√	√	√
Baselines	×	-	×	-	√	√	√	-	-
PyTorch-DRL	×	×	×	×	√	√	×	×	×
SB	×	√	×	-	√	√	√	×	×
rlpyt	√	√	√	√	√	√	√	√	√
天授	√	√	√	√	√	√	√	√	√

注：“-” 表示算法未实现

表 4.6　各平台模块化功能实现一览，其中：（1）算法实现模块化，指实现强化学习算法的时候遵循一套统一的接口；（2）数据处理模块化，指将内部数据流进行封装存储；（3）训练策略模块化，指由专门的类或函数来处理如何训练强化学习智能体。

平台与模块化	算法实现	数据处理	训练策略
RLlib	√	√	√
Baselines	×	×	×
PyTorch-DRL	部分模块化	×	√
SB	×	×	×
rlpyt	√	√	√
天授	√	√	部分模块化

PyTorch-DRL 三个框架外，其余的平台都做到了模块化。天授平台并没有在训练策略上做完全的模块化，因为在训练策略模块化虽然会节省代码，但是会使得用户难以二次修改代码进行开发。天授为了能够让开发者有更好的体验，在这二者之中做了折中：提供了一个定制化的训练策略函数，但不是必须的。用户可以利用天授提供的接口，和正常写强化学习代码一样，自由地编写所需训练策略。

4.2.4　代码复杂度与定制化训练环境

强化学习平台除了具有作为社区研究者中复现其他算法结果的作用之外，还承担在新场景、新任务上的运用和新算法的开发的作用，此时一个平台是否具有清晰简洁的代码结构、是否支持二次开发、是否能够方便地运用于新的任务（比如多模态环境）上，就成为一个衡量平台易用性的一个标准。表 4.7 总结了各平台在代码复杂度与是否可定制化训练环境两个维度的测试结果，其中前者采用开

源工具 cloc[①]进行代码统计，除去了测试代码和示例代码；后者采用 Mujoco 环境中 FetchReach-v1 任务进行模拟测试，其观测状态为一个字典，包含三个元素。此处使用这个任务来模拟对定制化多模态环境的测试，凡是报异常错误或者直接使用装饰器 `gym.wrappers.FlattenObservation()` 对观测值进行数据扁平化处理的平台，都不被认为对定制化训练环境做到了很好的支持。可以看出，天授在易用性的这两个评价层面上相比其他平台都具有十分明显的优势，使用精简的代码却能够支持更多需求。

表 4.7　各平台易用性一览，代码复杂度一栏数据格式为 Python 文件数/代码行数

平台与易用性	代码复杂度	环境定制化	文档	教程
RLlib	250/24065	√	×	√
Baselines	110/10499	×	×	×
PyTorch-DRL	55/4366	×	×	×
SB	100/10989	×	√	√
rlpyt	243/14487	×	√	×
天授	**29/2141**	√	√	√

4.2.5　文档教程

文档与教程对于平台的易用性而言具有十分重要的意义。表 4.7 列举出了各个平台的 API 接口文档与教程的情况。尽管天授的文档与 Stable-Baselines 相比还有待提高，但相比其它平台而言仍然提供了丰富的教程，供使用者使用。

4.2.6　单元测试与覆盖率

单元测试对强化学习平台有着十分重要的作用：它在本身就难以训练的强化学习算法上加上了一个保险栓，进行代码正确性检查，避免了一些低级的错误发生，同时还保证了一些基础算法的可复现性。表 4.8 从代码风格测试、基本功能测试、训练过程测试、代码覆盖率这些维度展示了各个平台所拥有的单元测试。大部分平台满足代码风格测试和基本功能测试要求，只有约一半的平台有对完整训练过程进行测试（此处指从智能体的神经网络随机初始化至智能体完全解决问题），以及显示代码覆盖率。综合来看，天授平台是其中表现最好的。

① GitHub 地址：https://github.com/AlDanial/cloc

表 4.8　各平台单元测试情况一览

平台与单元测试	PEP8 代码风格	基本功能	训练过程	代码覆盖率
RLlib	√	√	部分	暂缺 *
Baselines	√	√	部分	53% **
PyTorch-DRL	不遵循 + 无测试	√	完整	62% **
SB	√	√	部分	85%
rlpyt	×	部分	部分	22%
天授	√	√	完整	85%

注：*：由于 RLlib 平台单元测试过于复杂，代码覆盖率并未集成至单元测试中，因此无法获取代码覆盖率；**：手动在其单元测试脚本中添加代码覆盖率开启选项，并在 Travis CI 第三方测试平台中获取测试结果。

4.3　基准性能测试

本章节将各个强化学习平台在 OpenAI Gym[21] 简单环境中进行性能测试。实验运行环境配置参数如表 4.9 所示。所有运行实验耗时取纯 CPU 和 CPU+GPU 混合使用的这两种运行状态模式下的时间的最优值。为减小测试结果误差，每组实验将会以不同的随机种子运行 5 次。

表 4.9　实验运行环境参数

类型	参数
操作系统	Ubuntu 18.04
内核	5.3.0-53-generic
CPU	Intel i7-8750H (12) @ 4.100GHz
GPU	NVIDIA GeForce GTX 1060 Mobile
RAM	31.1 GiB DDR4
Disk	SAMSUNG MZVLB512HAJQ-000L2 SSD
NVIDIA 驱动版本	440.64.00
CUDA 版本	10.0
Python 版本	3.6.9
TensorFlow 版本	1.14.0
PyTorch 版本	1.4.0（PyTorch-DRL）或 1.5.0

4.3.1　离散动作空间免模型强化学习算法测试

离散动作空间的一系列强化学习任务中，最简单的任务是 OpenAI Gym 环境中的 CartPole-v0 任务：该任务要求智能体操纵小车，使得小车上的倒立摆能够保

持垂直状态，一旦偏离超过一定角度、或者小车位置超出规定范围，则认为游戏结束。该任务观测空间为一个四维向量，动作空间取值为 0 或 1，表示在这个时间节点内将小车向左或是向右移动。图 4.1 对该任务进行了可视化展示。

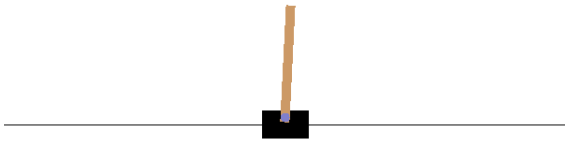


图 4.1　CartPole-v0 任务可视化

图 4.2　Pendulum-v0 任务可视化

该任务选取 PG[24]、DQN[1]、A2C[25]、PPO[4] 四种经典的免模型强化学习算法进行评测。根据 Gym 中说明的规则，每个算法必须在连续 100 次任务中，总奖励值取平均之后大于等于 195 才算解决了这个任务。各个平台不同算法解决任务的测试结果如表 4.10 所示，原始数据见附表 B-1。天授与其他平台相比，有着令人惊艳的性能，尤其是 PG、DQN 和 A2C 算法，能够在平均不到 10 秒的时间内解决该问题。

表 4.10　CartPole-v0 测试结果，运行时间单位为秒

平台与算法	PG	DQN	A2C	PPO
RLlib	19.26 ± 2.29	28.56 ± 4.60	57.92 ± 9.94	44.60 ± 17.04
Baselines	-	×	×	×
PyTorch-DRL *	×	31.58 ± 11.30	×	**23.99 ± 9.26**
SB	-	93.47 ± 58.05	57.56 ± 12.87	34.79 ± 17.02
rlpyt	**	**	**	**
天授	**6.09 ± 4.60**	**6.09 ± 0.87**	**6.36 ± 1.63**	31.82 ± 7.76

注：“-” 表示算法未实现；“×” 表示五组实验完成任务平均时间超过 1000 秒或未完成任务；
*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为 “训练过程中连续 20 次完整游戏的平均总奖励大于等于 195”；
**：rlpyt 对于离散动作空间非 Atari 任务的支持不友好，可参考 https://github.com/astooke/rlpyt/issues/135。

4.3.2 连续动作空间免模型强化学习算法测试

连续动作空间的一系列强化学习任务中，最简单的任务是 OpenAI Gym 环境中的 Pendulum-v0 任务：该任务要求智能体操控倒立摆，使其尽量保持直立，奖励值最大对应着与目标保持垂直，并且旋转速度和扭矩均为最小的状态。该任务观测空间为一个三维向量，动作空间为一个二维向量，范围为 $[-2, 2]$。图 4.2 对该任务进行了可视化展示。

该任务选取 PPO[4]、DDPG[30]、TD3[32]、SAC[33] 四种经典的免模型强化学习算法进行评测。和上一小节中的评测方法类似，每个算法必须在连续 100 次任务中，总奖励值取平均值后大于等于-250 才算解决该任务。各个平台不同算法解决任务的测试结果如表 4.11 所示，原始数据见附表 B-2。与之前结果类似，天授平台在各个算法中的测试都取得了不错的成绩。

表 4.11　Pendulum-v0 测试结果，运行时间单位为秒

平台与算法	PPO	DDPG	TD3	SAC
RLlib	123.62 ± 44.23	314.70 ± 7.92	149.90 ± 7.54	97.42 ± 4.75
Baselines	745.43 ± 160.82	×	-	-
PyTorch-DRL *	**	59.05 ± 10.03	57.52 ± 17.71	63.80 ± 27.37
SB	259.73 ± 27.37	277.52 ± 92.67	99.75 ± 21.63	124.85 ± 79.14
rlpyt	***	123.57 ± 30.76	113.00 ± 13.31	132.80 ± 21.74
天授	**16.18 ± 2.49**	**37.26 ± 9.55**	**44.04 ± 6.37**	**36.02 ± 0.77**

注：“-” 表示算法未实现；“×” 表示五组实验完成任务平均时间超过 1000 秒或未完成任务；
*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为 “训练过程中连续 20 次完整游戏的平均总奖励大于等于-250”；
**：PyTorch-DRL 中的 PPO 算法在连续动作空间任务中会报异常错误；
***：rlpyt 并未提供使用 PPO 算法的任何示例代码，经尝试无法成功跑通。

4.4 小结

本章节将天授平台与比较流行的 5 个深度强化学习平台进行功能维度和性能维度的对比。实验结果表明天授与其他平台相比，具有模块化、实现简洁、代码质量可靠、用户易用、速度快等等优点。

第 5 章　平台使用实例

5.1　实例一：在 CartPole-v0 环境中运行 DQN 算法

本实例将使用天授平台从头搭建整个训练流程，并获得一个能在平均 10 秒之内解决 CartPole-v0 环境的的 DQN[1] 智能体。

首先导入相关的包，并且定义相关参数：

```
import gym, torch, numpy as np, torch.nn as nn
from torch.utils.tensorboard import SummaryWriter
import tianshou as ts

task = 'CartPole-v0'
lr = 1e-3
gamma = 0.9
n_step = 4
eps_train, eps_test = 0.1, 0.05
epoch = 10
step_per_epoch = 1000
collect_per_step = 10
target_freq = 320
batch_size = 64
train_num, test_num = 8, 100
buffer_size = 20000
writer = SummaryWriter('log/dqn')
```

创建向量化环境从而能够并行采样：

```
# you can also use SubprocVectorEnv
train_envs = ts.env.VectorEnv([
    lambda: gym.make(task) for _ in range(train_num)])
test_envs = ts.env.VectorEnv([
    lambda: gym.make(task) for _ in range(test_num)])
```

使用 PyTorch 原生定义的网络结构，并定义优化器：

```
class Net(nn.Module):
    def __init__(self, state_shape, action_shape):
        super().__init__()
        self.model = nn.Sequential(*[
            nn.Linear(np.prod(state_shape), 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, np.prod(action_shape))
        ])
```

```
    def forward(self, s, state=None, info={}):
        if not isinstance(s, torch.Tensor):
            s = torch.tensor(s, dtype=torch.float)
        batch = s.shape[0]
        logits = self.model(s.view(batch, -1))
        return logits, state

env = gym.make(task)
state_shape = env.observation_space.shape \
    or env.observation_space.n
action_shape = env.action_space.shape or env.action_space.n
net = Net(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=lr)
```

初始化策略（Policy）和采集器（Collector）：

```
policy = ts.policy.DQNPolicy(
    net, optim, gamma, n_step,
    target_update_freq=target_freq)
train_collector = ts.data.Collector(
    policy, train_envs, ts.data.ReplayBuffer(buffer_size))
test_collector = ts.data.Collector(policy, test_envs)
```

开始训练：

```
result = ts.trainer.offpolicy_trainer(
    policy, train_collector, test_collector, epoch,
    step_per_epoch, collect_per_step, test_num, batch_size,
    train_fn=lambda e: policy.set_eps(eps_train),
    test_fn=lambda e: policy.set_eps(eps_test),
    stop_fn=lambda x: x >= env.spec.reward_threshold,
    writer=writer, task=task)
print(f'Finished training! Use {result["duration"]}')
```

会有进度条显示，并且在大约 10 秒内训练完毕，结果如下：

```
Epoch #1:  88%|#8| 880/1000 [00:05<00:00, ..., v/st=17329.91]
Finished training! Use 5.45s
```

可以将训练完毕的策略模型存储至文件中或者从已有文件中导入模型权重：

```
torch.save(policy.state_dict(), 'dqn.pth')
policy.load_state_dict(torch.load('dqn.pth'))
```

可以以每秒 35 帧的速率查看智能体与环境交互的结果：

```
collector = ts.data.Collector(policy, env)
collector.collect(n_episode=1, render=1 / 35)
collector.close()
```

查看 TensorBoard 中存储的结果：

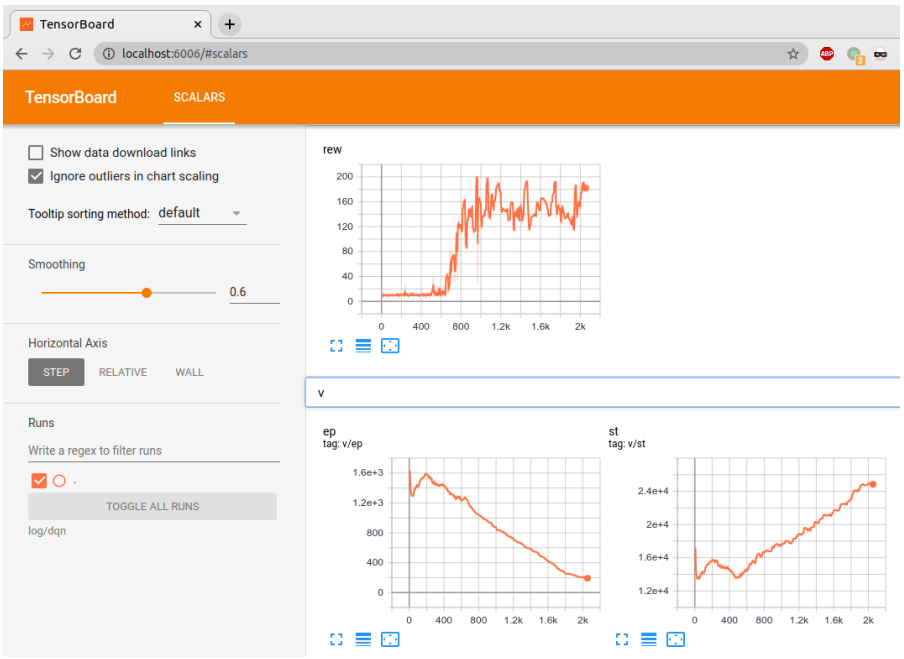


图 5.1　TensorBoard 可视化训练过过程

```
tensorboard --logdir log/dqn
```

结果如图 5.1 所示。

当然，如果想要定制化训练策略而不使用训练器提供的现有逻辑，也是可以的。下面的代码展示了如何定制化训练策略：

```
# pre-collect 5000 frames with random action before training
policy.set_eps(1)
train_collector.collect(n_step=5000)

policy.set_eps(0.1)
for i in range(int(1e6)):  # total step
    collect_result = train_collector.collect(n_step=10)

    # once if the collected episodes' mean returns reach
    # the threshold, or every 1000 steps, we test it on
    # test_collector
    if collect_result['rew'] >= env.spec.reward_threshold \
            or i % 1000 == 0:
        policy.set_eps(0.05)
        result = test_collector.collect(n_episode=100)
        if result['rew'] >= env.spec.reward_threshold:
            # end of training loop
            break
        else:
            # back to training eps
            policy.set_eps(0.1)
```

```
22
23      # train policy with a sampled batch data
24      losses = policy.learn(
25          train_collector.sample(batch_size=64))
26
27  print('Finished training! Test mean returns:',
28        str(result["rew"]))
```

5.2 实例二：循环神经网络的训练

在 POMDP 场景中往往需要循环神经网络的训练支持。此处为简单起见，仍然以实例一中的场景和代码为基础进行展示。需要的改动如下：

首先修改模型为 LSTM：

```
1  class Recurrent(nn.Module):
2      def __init__(self, state_shape, action_shape):
3          super().__init__()
4          self.fc1 = nn.Linear(np.prod(state_shape), 128)
5          self.nn = nn.LSTM(input_size=128, hidden_size=128,
6                            num_layers=3, batch_first=True)
7          self.fc2 = nn.Linear(128, np.prod(action_shape))
8
9      def forward(self, s, state=None, info={}):
10         if not isinstance(s, torch.Tensor):
11             s = torch.tensor(s, dtype=torch.float)
12         # s [bsz, len, dim] (training)
13         # or [bsz, dim] (evaluation)
14         if len(s.shape) == 2:
15             bsz, dim = s.shape
16             length = 1
17         else:
18             bsz, length, dim = s.shape
19         s = self.fc1(s.view([bsz * length, dim]))
20         s = s.view(bsz, length, -1)
21         self.nn.flatten_parameters()
22         if state is None:
23             s, (h, c) = self.nn(s)
24         else:
25             # we store the stack data with [bsz, len, ...]
26             # but pytorch rnn needs [len, bsz, ...]
27             s, (h, c) = self.nn(s, (
28                 state['h'].transpose(0, 1).contiguous(),
29                 state['c'].transpose(0, 1).contiguous()))
30         s = self.fc2(s[:, -1])
31         # make sure the 0-dim is batch size: [bsz, len, ...]
32         return s, {'h': h.transpose(0, 1).detach(),
33                    'c': c.transpose(0, 1).detach()}
```

其次重新定义策略，并将 `train_collector` 中的重放缓冲区设置成堆叠采样模式，堆叠帧数 n 为 4：

```
env = gym.make(task)
state_shape = env.observation_space.shape \
    or env.observation_space.n
action_shape = env.action_space.shape or env.action_space.n
net = Recurrent(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=lr)

policy = ts.policy.DQNPolicy(
    net, optim, gamma, n_step,
    target_update_freq=target_freq)
train_collector = ts.data.Collector(
    policy, train_envs,
    ts.data.ReplayBuffer(buffer_size, stack_num=4))
test_collector = ts.data.Collector(policy, test_envs)
```

即可使用实例一中的代码进行正常训练，结果如下：

```
Epoch #1:  83%|#4| 831/1000 [00:19<00:03, ..., v/st=13832.13]
Finished training! Use 19.63s
```

5.3 实例三：多模态任务训练

在像机器人抓取之类的任务中，智能体会获取多模态的观测值。天授完整保留了多模态观测值的数据结构，以数据组的形式给出，并且能方便地支持分片操作。以 Gym 环境中的 “FetchReach-v1” 为例，每次返回的观测值是一个字典，包含三个元素 “observation”、“achieved_goal” 和 “desired_goal”。

在实例一代码的基础上进行修改：

```
task = 'FetchReach-v1'
train_envs = ts.env.VectorEnv([
    lambda: gym.make(task) for _ in range(train_num)])
test_envs = ts.env.VectorEnv([
    lambda: gym.make(task) for _ in range(test_num)])

class Net(nn.Module):
    def __init__(self, state_shape, action_shape):
        super().__init__()
        self.model = nn.Sequential(*[
            nn.Linear(np.prod(state_shape), 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, np.prod(action_shape))
```

```
        ])
    def forward(self, s, state=None, info={}):
        o = s.observation
        # s.achieved_goal, s.desired_goal are also available
        if not isinstance(o, torch.Tensor):
            o = torch.tensor(o, dtype=torch.float)
        batch = o.shape[0]
        logits = self.model(o.view(batch, -1))
        return logits, state

env = gym.make(task)
env.spec.reward_threshold = 1e10
state_shape = env.observation_space.spaces['observation']
state_shape = state_shape.shape
action_shape = env.action_space.shape
net = Net(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=lr)
```

剩下的代码与实例一一致，可以直接运行。通过对比可以看出，只需改动神经网络中 `forward` 函数的 s 参数的处理即可。

第 6 章　总结

本论文描述了一个基于 PyTorch 的深度强化学习算法平台 “天授”。该平台支持了诸多主流的强化学习算法（主要为免模型强化学习算法），支持各种不同的环境的并行采样、数据存储、定制化，还同时做到了模块化、实现简洁、可复现性、接口灵活等等特性，并且在基准测试中，天授的速度优于其他已有平台。

天授平台旨在提供一个用户友好的标准化的强化学习平台，降低算法开发成本。如今天授已在 GitHub 上开源①，并且提供了一系列教程和代码文档②，目前已经拥有 1500 多颗星标，收到了众多使用者的一致好评。

后续的工作将围绕如下方面进行：

- **算法：**（1）加入更多免模型强化学习算法，比如 Rainbow DQN[36]；（2）加入基于模型的强化学习算法，比如 MCTS 与 AlphaGo[2]（目前平台接口已经支持，代码正在完善中）；（3）加入更多模仿学习算法，比如 GAIL[35]；（4）加入多智能体训练的接口；
- **环境：**加入更多种类的环境并行接口，比如共享内存的环境接口，做到更高效的并行采样；
- **文档：**完善教程，加入中文版本文档；
- **示例：**提供更多任务上（如 Atari、Mujoco 各个任务）调优过的示例代码，方便开箱即用与二次开发。

① GitHub 项目地址：https://github.com/thu-ml/tianshou/

② 文档地址：http://tianshou.readthedocs.io/

插图索引

表格索引

公式索引

参考文献

[1] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Human-level control through deep reinforcement learning[J/OL]. Nature, 2015, 518(7540):529-533. https://doi.org/10.1038/nature14236.

[2] SILVER D, HUANG A, MADDISON C J, et al. Mastering the game of go with deep neural networks and tree search[J/OL]. Nature, 2016, 529(7587):484-489. https://doi.org/10.1038/nature16961.

[3] SENIOR A W, EVANS R, JUMPER J, et al. Improved protein structure prediction using potentials from deep learning[J]. Nature, 2020:1-5.

[4] SCHULMAN J, WOLSKI F, DHARIWAL P, et al. Proximal policy optimization algorithms [J/OL]. CoRR, 2017, abs/1707.06347. http://arxiv.org/abs/1707.06347.

[5] BERNER C, BROCKMAN G, CHAN B, et al. Dota 2 with large scale deep reinforcement learning[J/OL]. CoRR, 2019, abs/1912.06680. http://arxiv.org/abs/1912.06680.

[6] PASZKE A, GROSS S, MASSA F, et al. Pytorch: An imperative style, high-performance deep learning library[C/OL]//Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada. 2019: 8024-8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.

[7] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. Imagenet classification with deep convolutional neural networks[C/OL]//Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States. 2012: 1106-1114. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

[8] TESAURO G. Td-gammon, a self-teaching backgammon program, achieves master-level play [J/OL]. Neural Computation, 1994, 6(2):215-219. https://doi.org/10.1162/neco.1994.6.2.215.

[9] DHARIWAL P, HESSE C, KLIMOV O, et al. Openai baselines[J/OL]. GitHub repository, 2017. https://github.com/openai/baselines.

[10] ACHIAM J. Spinning Up in Deep Reinforcement Learning[J/OL]. GitHub repository, 2018. https://github.com/openai/spinningup.

[11] LIANG E, LIAW R, NISHIHARA R, et al. Rllib: Abstractions for distributed reinforcement learning[C/OL]//Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. 2018: 3059-3068. http://proceedings.mlr.press/v80/liang18b.html.

[12] STOOKE A, ABBEEL P. rlpyt: A research code base for deep reinforcement learning in pytorch [J/OL]. CoRR, 2019, abs/1909.01500. http://arxiv.org/abs/1909.01500.

[13] PONG V H, DALAL M, LIN S, et al. rlkit: Collection of reinforcement learning algorithms [J/OL]. GitHub repository, 2019. https://github.com/vitchyr/rlkit.

[14] GARAGE CONTRIBUTORS T. Garage: A toolkit for reproducible reinforcement learning research[J/OL]. GitHub repository, 2019. https://github.com/rlworkgroup/garage.

[15] CASTRO P S, MOITRA S, GELADA C, et al. Dopamine: A research framework for deep reinforcement learning[J/OL]. CoRR, 2018, abs/1812.06110. http://arxiv.org/abs/1812.06110.

[16] OSBAND I, DORON Y, HESSEL M, et al. Behaviour suite for reinforcement learning[C/OL]// 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. 2020. https://openreview.net/forum?id=rygf-kSYwH.

[17] HILL A, RAFFIN A, ERNESTUS M, et al. Stable baselines[J/OL]. GitHub repository, 2018. https://github.com/hill-a/stable-baselines.

[18] PLAPPERT M. keras-rl[J/OL]. GitHub repository, 2016. https://github.com/keras-rl/keras-rl.

[19] CHRISTODOULOU P. Deep reinforcement learning algorithms with pytorch[J/OL]. GitHub repository, 2019. https://github.com/p-christ/Deep-Reinforcement-Learning-Algorithms-with-PyTorch.

[20] KUHNLE A, SCHAARSCHMIDT M, FRICKE K. Tensorforce: a tensorflow library for applied reinforcement learning[J/OL]. GitHub repository, 2017. https://github.com/tensorforce/tensorforce.

[21] BROCKMAN G, CHEUNG V, PETTERSSON L, et al. Openai gym[J/OL]. CoRR, 2016, abs/1606.01540. http://arxiv.org/abs/1606.01540.

[22] ABADI M, BARHAM P, CHEN J, et al. Tensorflow: A system for large-scale machine learning [C/OL]//12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. 2016: 265-283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[23] ANDRYCHOWICZ M, CROW D, RAY A, et al. Hindsight experience replay[C/OL]//Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA. 2017: 5048-5058. http://papers.nips.cc/paper/7090-hindsight-experience-replay.

[24] SUTTON R S, MCALLESTER D A, SINGH S P, et al. Policy gradient methods for reinforcement learning with function approximation[C/OL]//Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]. 1999: 1057-1063. http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

[25] MNIH V, BADIA A P, MIRZA M, et al. Asynchronous methods for deep reinforcement learning [C/OL]//Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. 2016: 1928-1937. http://proceedings.mlr.press/v48/mniha16.html.

[26] SCHULMAN J, LEVINE S, ABBEEL P, et al. Trust region policy optimization[C/OL]// Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. 2015: 1889-1897. http://proceedings.mlr.press/v37/schulman15.html.

[27] SCHULMAN J, MORITZ P, LEVINE S, et al. High-dimensional continuous control using generalized advantage estimation[C/OL]//4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. 2016. http://arxiv.org/abs/1506.02438.

[28] VAN HASSELT H, GUEZ A, SILVER D. Deep reinforcement learning with double q-learning [C/OL]//Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA. 2016: 2094-2100. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389.

[29] SCHAUL T, QUAN J, ANTONOGLOU I, et al. Prioritized experience replay[C/OL]//4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. 2016. http://arxiv.org/abs/1511.05952.

[30] LILLICRAP T P, HUNT J J, PRITZEL A, et al. Continuous control with deep reinforcement learning[C/OL]//4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. 2016. http://arxiv.org/abs/1509.02971.

[31] SILVER D, LEVER G, HEESS N, et al. Deterministic policy gradient algorithms[C/OL]// Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014. 2014: 387-395. http://proceedings.mlr.press/v32/silver14.html.

[32] FUJIMOTO S, VAN HOOF H, MEGER D. Addressing function approximation error in actor-critic methods[C/OL]//Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. 2018: 1582-1591. http://proceedings.mlr.press/v80/fujimoto18a.html.

[33] HAARNOJA T, ZHOU A, HARTIKAINEN K, et al. Soft actor-critic algorithms and applications[J/OL]. CoRR, 2018, abs/1812.05905. http://arxiv.org/abs/1812.05905.

[34] FINN C, LEVINE S, ABBEEL P. Guided cost learning: Deep inverse optimal control via policy optimization[C/OL]//Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. 2016: 49-58. http://proceedings.mlr.press/v48/finn16.html.

[35] HO J, ERMON S. Generative adversarial imitation learning[C/OL]//Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems

2016, December 5-10, 2016, Barcelona, Spain. 2016: 4565-4573. http://papers.nips.cc/paper/6391-generative-adversarial-imitation-learning.

[36] HESSEL M, MODAYIL J, VAN HASSELT H, et al. Rainbow: Combining improvements in deep reinforcement learning[C/OL]//Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. 2018: 3215-3222. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17204.

致　谢

感谢我的导师苏航老师和朱军老师：苏航老师给予了我科研方向、论文写作方面的指导，鼓励我自由地去探索科研问题并给予了不少帮助；朱军老师为我提供了一个宽松的学术科研环境与氛围，同时提供了其他实验室难以企及的计算资源。没有他们就没有今天的我。

感谢阎栋老师、路橙学长、宋世虹学姐、刘家硕同学和黄瑞同学，无论是科研问题、课程学业、还是课余生活，他们都给了我不少的帮助。

感谢白晓颖老师的软件工程课程，我大二上课的时候说实话没怎么认真学，但是开发天授平台的过程使我重新认识到了软件工程理论的意义与重要性。

感谢何强、肖清、张鸣昊、王冠同学对天授平台的贡献和对本文写作提出的建议。

感谢清华大学计算机系提供给我这样一个平台，让我能够不断地突破自己之前认为的自我上限，过出了一个我想要的大学生活。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

附录 A　外文资料的书面翻译

RLlib：一个分布式强化学习系统的凝练[1]

摘要：强化学习算法涉及到高度不规则的计算模式的深度嵌套，每个模式通常都表现出分布式计算的机会。我们提出采用自顶向下分层控制的分布式强化学习算法，从而更好地采用并行计算资源调度来完成这些任务。在 RLlib，一个可拓展的强化学习软件平台中，我们展示了我们所提出理论的好处：它能够让一系列的强化学习算法达到高性能、可拓展和大量代码重用这些特性。RLlib 是开源项目 Ray 的一部分，文档位于 https://docs.ray.io/en/master/rllib.html。

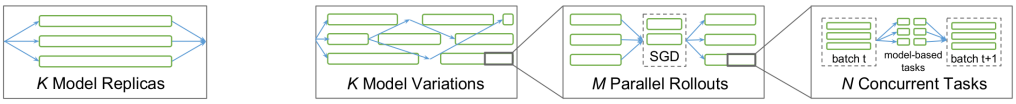
A.1　引言

并行计算和符号微分是最近深度学习成功的基石。如今有各种各样的深度学习框架被开发出来，研究者们可以在这些框架中设计神经网络快速迭代创新，并能够在该领域的进步所需的规模上加速训练。

虽然强化学习界在深度学习的系统和抽象方面取得了很大的进步，但在直接针对强化学习的系统和抽象设计方面的进展相对较少。尽管如此，强化学习中的许多挑战都源于对学习和仿真的规模化需求，同时也需要整合快速增长的算法和模型。因此设计这么一个系统是很有必要的。

在没有单一的主导计算模式（例如张量代数）或基本组成规则（例如符号微分）的情况下，强化学习算法的设计与实现通常非常麻烦，它要求强化学习研究人员直接设计复杂嵌套并行程序。与深度学习框架中的典型运算符不同，各个组件可能需要跨集群并行、利用深度学习框架实现的神经网络、递归调用其他组件、与其他接口进行交互，其中许多部分的异构性和分布式性质对实现它们的并行版本提出了不小的挑战，而上层算法也正在迅速发展，也在不同级别上提出了并行性的更高要求。最后，这些算法模块还需要处理不同层级、甚至跨物理设备的并行。

强化学习算法框架在近些年来被不断开发。尽管其中一些具有高度可拓展性，但很少能实现大规模组件的组合，很大程度上是由于这些库使用的许多框架都依赖于长时间运行的程序副本之间的通信来进行分布式执行。例如 MPI，分布

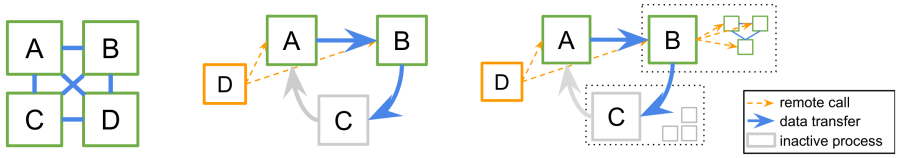


(a) 深度学习　　(b) 深度强化学习

图 A-1　和深度学习相比，深度强化学习具有着不同层级的并行和不一样的计算模式。表 A-2 呈现了更详细的说明。

式 TensorFlow 和参数服务器等原型。这些原型不会将并行性和资源需求封装在单个组件中，因此重用这些分布式组件需要在程序中插入适当的控制点。这是一个十分繁琐且容易出错的过程。缺少可用的封装会阻碍代码重用，并导致数学上复杂且高度随机的算法的重新实现容易出错。更糟糕的是，在分布式环境中，重新实现一个新的强化学习算法通常还必须重新实现分布式通信和执行的大部分内容。

我们认为通过组合和重用现有模块与算法实现来构建可拓展的强化学习算法对于该领域快速发展和进步至关重要。我们注意到实现强化学习平台的困难之处在于可伸缩性和可组合性，而这两种特性不能通过单线程库轻松实现。为此，我们主张围绕逻辑集中式程序控制（逻辑中控）和并行封装的原理构造分布式强化学习组件。我们根据这些原则构建了 RLlib，结果不仅能够实现各种最新的强化学习算法，而且还拥有了可用于轻松组成新算法的可拓展单元。



(a) 分布式控制　　(b) 逻辑中控　　(c) 分层控制

图 A-2　目前大多数强化学习算法都是以完全分布式的方式编写的 (a)。我们提出了一种分层控制模型 (c)，它扩展了 (b)，支持强化学习中的嵌套和超参数调优工作，简化和统一了用于实现的编程模型。

A.1.1　强化学习训练的计算模式不规则性

目前的强化学习算法在其创建的计算模式中是高度不规则的，如表 A-1 所示，突破了如今流行的分布式框架所支持的计算模型的界限。这种不规则发生在如下几个层面：

1. 根据算法的不同，任务的持续时间和资源需求也有数量级的差异；例如 A3C 的更新可能需要几毫秒，但其他算法如 PPO 需要更大粒度时间颗粒。
2. 通信模式各异，从同步到异步的梯度优化，再到在高通量的异策略学习算

表 A-1　不同强化学习算法，计算需求量跨度大。

维度	DQN/笔记本	IMPALA+PBT/大型计算集群
单任务时长	约 1 毫秒	数分钟
单任务所需计算资源	1 个 CPU	数个 CPU 和 GPU
总共所需计算资源	1 个 CPU	数百个 CPU 和 GPU
嵌套深度	1 层	多于 3 层
所需内存	MB 级别	百 GB 级别
执行方式	同步	异步且高并发

法（如 Ape-X 和 IMPALA）中拥有多种类型的异步任务，通信模式各不相同。

3. 基于模型的混合算法（表A-2）、强化学习或深度学习训练相结合的超参数调优、或是在单一算法中结合无导数优化和基于梯度的优化等方式产生嵌套计算。
4. 强化学习算法经常需要维护和更新大量的状态，包括策略参数、重放缓冲区，甚至还有外部模拟器等。

因此，开发人员只能使用大杂烩的框架来实现他们的算法，包括参数服务器、类 MPI 框架中的集体通信基元、任务队列等。对于更复杂的算法，常见的做法是构建自定义的分布式系统，在这个系统中，进程之间独立计算和协调，没有中央控制（图 A-2(a)）。虽然这种方法可以实现较高的性能，但开发和评估的成本很大，不仅因为需要实现和调试分布式程序，而且因为这些算法的组成进一步使其实现复杂化（图 A-3）。此外，如今现有的计算框架（如 Spark、MPI）通常是假设有规律的计算模式，当子任务的持续时间、资源需求或嵌套不同时，这些计算框架会有性能损失。

A.1.2　对分布式强化学习算法进行逻辑中控

我们希望一个单一的编程模型能够满足强化学习算法训练的所有要求。这可以在不放弃结构化计算的高级框架的情况下实现。对于每个分布式强化学习算法，我们可以写出一个等效的算法，表现出逻辑上集中的程序控制（图 A-2(b)）。也就是说，不用让独立执行进程（图 A-2(a) 中的 **A**、**B**、**C**、**D**）相互协调（例如，通过 RPC、共享内存、参数服务器或集体通信），而是一个单一的驱动程序（图 A-2(b) 和 A-2(c) 中的 **D**）可以将算法的子任务委托给其他进程并行执行。在这种工作模式中，工作进程 **A**、**B**、**C** 被动地保持状态（如策略或仿真器状态），

但在被 **D** 调用之前不执行任何计算，为了支持嵌套计算，我们提出用分层委托控制模型（图 A-2(c)）来扩展集中控制模型，允许工作进程（如 **B**、**C**）在执行任务时进一步将自己的工作（如仿真、梯度计算）委托给自己的子工作进程。

在这样一个逻辑上集中的分层控制模型的基础上搭建强化学习框架，有如下几个重要优势：首先，等效算法在实际应用中往往更容易实现，因为分布式控制逻辑完全封装在一个进程中，而不是多个进程同时执行。其次，将算法组件分离成不同的子程序（例如，做卷积运算、计算梯度与某些策略的目标函数的梯度），可以在不同的执行模式下实现代码的重用。有不同资源需求的子任务（CPU 任务或者 GPU 任务）可以放在不同的机器上，从而能够降低计算成本，我们将在第 A.5 章中展示这一点。最后，在这个模型中编写的分布式算法可以相互之间无缝嵌套，满足了并行性封装原则。

逻辑中控模型可以有很高的性能，我们提出的分层委托控制模型更是如此。这是因为进程之间的大部分数据传输（图 A-2 中的蓝色箭头）都发生在驱动带外，没有遇到任何驱动中心瓶颈。事实上，许多高度可扩展的分布式系统在设计中都利用集中控制。像 TensorFlow 这样的框架也实现了将张量计算逻辑上的集中调度到可用的物理设备上，即使需求只有单个可微分的张量图。我们的工作能够将这一原则扩展到更广泛的机器学习系统设计理念中。

本文的贡献主要有如下三点：

1. 我们为强化学习训练提出了一个通用且模块化的分层编程模型（章节 A.2）；
2. 我们描述了 RLlib，一个高度可扩展的强化学习算法库，以及如何在我们的代码库上面快速构建一系列强化学习算法（章节 A.3）；
3. 我们讨论了这一框架的性能（章节 A.4），并表明 RLlib 在各种强化学习算法中和众多框架相比达到或超过了最优性能（章节 A.5）。

A.2 分层并行任务模型

如图 A-3 所示，如果使用 MPI 或者分布式 Tensorflow 之类的框架作为底层来设计并行化写强化学习算法代码的时候，需要对每个算法的适配进行定制化代码修改。这限制了新的分布式强化学习算法的快速开发。尽管图 A-3 中的示例很简单，但例如 HyperBand、PBT 等需要长时间运行的、精细的超参数调整的算法越来越需要对培训进行细粒度的控制。

我们建议在基于任务的灵活编程模型（例如 Ray）的基础上，通过分层控制

```
if mpi.get_rank() <= m:
    grid = mpi.comm_world.split(0)
else:
    eval = mpi.comm_world.split(
        mpi.get_rank() % n)
...
if mpi.get_rank() == 0:
    grid.scatter(
        generate_hyperparams(), root=0)
    print(grid.gather(root=0))
elif 0 < mpi.get_rank() <= m:
    params = grid.scatter(None, root=0)
    eval.bcast(
        generate_model(params), root=0)
    results = eval.gather(
         result, root=0)
    grid.gather(results, root=0)
elif mpi.get_rank() > m:
    model = eval.bcast(None, root=0)
    result = rollout(model)
    eval.gather(result, root=0)
```

(a) 分布式控制

```
@ray.remote
def rollout(model):
    # perform a rollout and
    # return the result

@ray.remote
def evaluate(params):
    model = generate_model(params)
    results = [rollout.remote(model)
       for i in range(n)]
    return results

param_grid = generate_hyperparams()
print(ray.get([evaluate.remote(p)
   for p in param_grid]))
```

(b) 分层控制

图 A-3　将分布式超参数搜索与分布式计算的函数组合在一起，会涉及到复杂的嵌套并行计算模式。如果使用 MPI (a)，必须从头开始编写一个新程序，将所有元素混合在一起。使用分层控制 (b)，组件可以保持不变，并且可以作为远程任务简单地调用。

和逻辑中控来构建强化学习算法库。基于任务的系统允许在细粒度的基础上，在子进程上异步调度和执行子例程，并在进程之间检索或传递结果。

A.2.1　和已有的分布式机器学习抽象模式的关系

诸如参数服务器和集体通信操作之类的抽象模式尽管通常是为分布式控制制定的，但也可以在逻辑中控模型中使用：比如 RLlib 在其某些策略优化器中使用全局规约或者参数服务器等模式 (图 A-4)，我们将在第 A.5 章中评估它们的性能。

A.2.2　使用 Ray 来实现分层控制

其实在一台机器上就可以简单地使用线程池和共享内存来实现所提出的编程模型，但是如果需要的话，基础框架也可以扩展到更大的集群。我们选择在 Ray 框架之上构建 RLlib，该框架允许将 Python 任务在大型集群中分布式执行。Ray 的分布式调度程序很适合分层控制模型，因为可以在 Ray 中实现嵌套计算，而没有中央任务调度瓶颈。

为了实现逻辑中控模型，首先必须要有一种机制来启动新进程并安排新任务。Ray 使用 *Ray actor* 满足了这一要求：Ray actor 是可以在集群中创建并接受远程函数调用的 Python 类，并且这些 actor 允许在函数调用中反过来启动更多的

actor 并安排任务，这也满足了我们对层次调度的需求。

为了提高性能，Ray 提供了诸如聚合和广播之类的标准通信原语，并通过共享内存对象存储来实现大型数据对象的零复制共享，如第 A.5 章所示。我们将在第 A.4 章中进一步讨论框架性能。

A.3 强化学习的抽象模式

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。

A.3.1 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：

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

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：

$$\rho_\theta(X_{t,K}, X_{t,K}^1, \cdots, X_{t,K}^P) \Rightarrow X_{\text{post}} \tag{A-2}$$

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他网络：

$$L(\theta; X) \Rightarrow \text{loss} \tag{A-3}$$

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：

$$u^1, \cdots, u^M(\theta) \Rightarrow (s, \theta_{\text{update}}) \tag{A-4}$$

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*)
```

A.3.2 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)
```

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 OpenAI Gym、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]))
```

A.3.3 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（图 A-4(c)）。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()))
```

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 $\text{step}(L(\theta), X, \theta) \Rightarrow \theta_{opt}$。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，$\text{step}(G, ev_1, \cdots, ev_n, \theta) \Rightarrow \theta_{\text{opt}}$，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。

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
avg_grad = aggregate(grads)
local_graph.apply(avg_grad)
weights = broadcast(
    local_graph.weights())
for ev in evaluators:
    ev.set_weights(weights)
```

(a) 全局规约

```
samples = concat([ev.sample()
    for ev in evaluators])
pin_in_local_gpu_memory(samples)
for _ in range(NUM_SGD_EPOCHS):
    local_g.apply(local_g.grad(samples)
weights = broadcast(local_g.weights())
for ev in evaluators:
    ev.set_weights(weights)
```

(b) 本地多 GPU

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
    grad, ev, grads = wait(grads)
    local_graph.apply(grad)
    ev.set_weights(
        local_graph.get_weights())
    grads.append(ev.grad(ev.sample()))
```

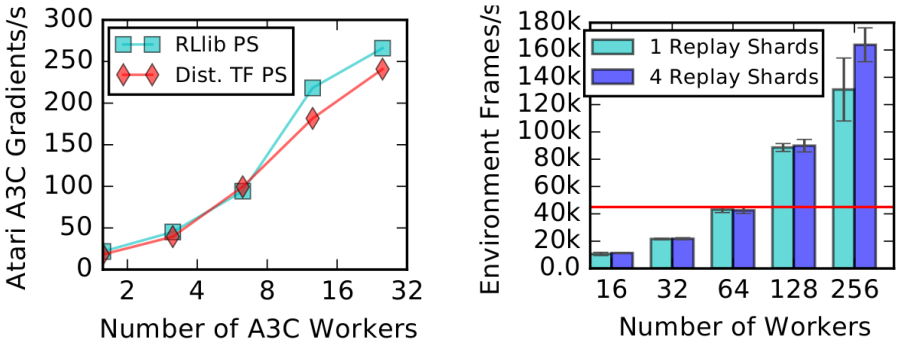
(c) 异步计算

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
    grad, ev, grads = wait(grads)
    for ps, g in split(grad, ps_shards):
        ps.push(g)
    ev.set_weights(concat(
        [ps.pull() for ps in ps_shards])
    grads.append(ev.grad(ev.sample()))
```

(d) 分片参数服务器

图 A-4　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。此处省略迷你批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水线模式，此处未给予显示。

如图 A-4 所示，通过利用集中控制，策略优化器简洁地抽象了强化学习算法优化中的多种选择：同步与异步，全局规约与参数服务器，以及使用 GPU 与 CPU 的选择。RLlib 的策略优化器提供了与优化的参数服务器算法（图 A-5(a)）和基于 MPI 的实现（第 A.5 章）相当的性能。这种优化器在逻辑中控模型中很容易被实现，因为每个策略优化器对它所属的分布式计算进程有完全的控制权。



(a) 分片参数服务器　　(b) RLlib 中 Ape-X 算法

图 A-5　RLlib 的集中控制的策略优化器与专有系统实现性能相匹配或超过其实现。RLlib 的参数服务器优化器使用 8 个分片，与类似条件下测试的分布式 TensorFlow 实现相比十分具有竞争力。RLlib 的 Ape-X 策略优化器在 256 个工作进程、跳跃帧数为 4 的情况下可扩展到 16 万帧每秒，远远超过了参考吞吐量 4.5 万帧每秒，证明了单线程的 Python 控制器也可以有效地扩展到高吞吐量任务上。

表 A-2　RLlib 的策略优化器和评价器在逻辑中控模型中捕获了常见的组件（评估、回放、梯度优化器），并利用 Ray 的分层任务模型支持其他分布式组件。

算法类型	策略评估	重放缓冲区	梯度优化器	其他分布式组件
DQN 算法族	√	√	√	
策略梯度	√		√	
异策略梯度	√	√	√	
基于模型/混合	√		√	基于模型的规划
多智能体	√	√	√	
进化策略	√			无导数优化
AlphaGo	√	√	√	MCTS，无导数优化

A.3.4 RLlib 抽象模式的完备性和普适性

我们通过以 RLlib 中的 API 形式化表 A-2 中列出的算法来证明 RLlib 中的抽象方法的完备性。在合适的情况下，我们还会描述该算法在 RLlib 中的具体实现。

DQN 算法族： DQN 算法族使用 y^1 存储 TD 误差，在 ρ_θ 中实现 n 步奖励值的计算，优化 Q 值的目标函数在 L 中很容易实现。目标神经网络的更新在 u^1 中实现，设置探索权重参数 ϵ 在 u^2 中实现。

DQN 算法实现：为了支持经验回放，RLlib 中的 DQN 使用了一个策略优化器，将收集的样本保存在嵌入式回放缓冲区中。用户可以选择使用异步优化器（图 A-4(c)）。在优化器优化步骤之间，通过调用 u^1 函数来更新目标网络。

Ape-X 算法实现：Ape-X 是 DQN 的一个变种，它利用分布式体经验优先化来扩展到数百个内核。为了适应我们的 DQN 实现，我们创建了具有分布式 ϵ 值的策略评估器，并编写了一个约 200 行的高吞吐量策略优化器，使用 Ray 的原语在各个 Ray actor 的重放缓冲区之间进行流水线采样和数据传输。我们的实现几乎线性地扩展到 256 个工作进程同时采样约每秒 16 万环境帧（图 A-5(b)），在一个 V100 GPU 上，优化器可以计算梯度的速度为每秒约 8500 张输入大小为 80×80×4 的图像。

策略梯度 / 异策略梯度： 实现这些算法可以将预测的价值函数在 y^1 中存储，在 ρ_θ 中实现优势估计函数，并将 actor 和 critic 的目标函数优化部分写在 L 中。

PPO 算法实现：由于 PPO 的目标函数允许对样本数据进行多次 SGD 传递，所以当有足够的 GPU 内存时，RLlib 选择一个 GPU 策略优化器（图 A-4(b)），将数据引脚到本地 GPU 内存中。在每次迭代中，优化器从评估器副本中收集样本，在本地执行多 GPU 优化，然后广播新的模型权重。

A3C 算法实现：RLlib 的 A3C 可以使用异步（图 A-4(c)）或分片参数服务器（图 A-4(d)）策略优化器。这些优化器从策略评价器中收集梯度，随后更新 θ 的一系列副本。

DDPG 算法实现：RLlib 的 DDPG 使用与 DQN 相同的经验重放策略优化器。L 包括 actor 和 critic 的目标函数。用户也可以选择使用 Ape-X 策略优化器来优化 DDPG 算法。

基于模型/混合： 基于模型的强化学习算法扩展了 $\pi_\theta(o_t, h_t)$，根据模型的推演进行决策，这部分也可以使用 Ray 进行并行化。为了更新它们的环境模型，可以将模型优化的目标函数写在 L 中，也可以将模型单独训练，即使用 Ray 原语做到并行，并通过 u^1 函数定期更新其权重。

多智能体：策略评估器可以在同一环境中同时运行多个策略为每个智能体产生批量的经验。许多多智能体强化学习算法使用一个中心化的价值函数，可以通过 ρ_θ 整理来自多个智能体的经验来支持。

进化策略（ES）：是一种无导数优化方法，可以通过非梯度策略优化器实现。

进化策略算法实现：由于进化策略是一种无导数优化算法，因此可以很好地扩展到具有数千台 CPU 的集群。我们只做了一些微小改动，就能将进化策略的单线程实现移植到 RLlib 上，并通过行为体聚合树进一步扩展（图 A-8(a)）。这表明分层控制模型既灵活又容易适应不同算法。

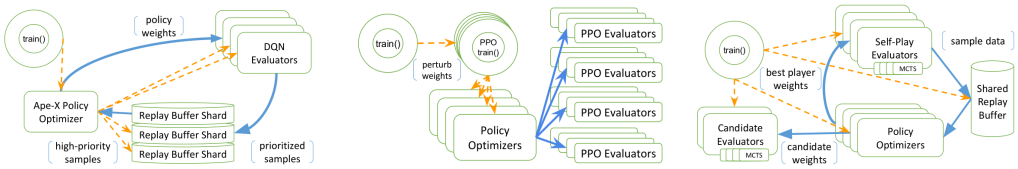
PPO-ES 实验：我们研究了一种混合算法，在 ES 优化步骤的内循环中运行 PPO 更新，该算法对 PPO 模型进行随机扰动。该算法的实现只花了大约 50 行代码，不需要改变 PPO，显示了并行性封装的价值。在我们所做的实验中，在 Walker2d-v1 任务上 PPO-ES 收敛得比 PPO 更快，获得奖励也更高。一个类似的 A3C-ES 实现以少于原先 30% 的时间解决了 PongDeterministic-v4。

AlphaGo：我们用 Ray 和 RLlib 的抽象组合来描述 AlphaGo Zero 算法的可扩展实现方法。

1. *对多个分布式组件进行逻辑中控*：AlphaGo Zero 使用了多个分布式组件：模型优化器、自我对弈评估器、候选模型评估器和共享重放缓冲区。这些组件可以在顶层 AlphaGo 策略优化器下作为 Ray actor 进行管理。每个优化器进行单步优化的时候都会在 Ray actor 状态上循环处理新的结果，在 Ray actor 之间路由数据并启动新的 Ray actor 实例。
2. *共享重放缓冲区*：AlphaGo Zero 将来自于自我对弈评估器的经验存储在共享重放缓冲区中。这需要将对局结果路由到共享缓冲区，通过将结果对象的引用从一个 actor 传递到另一个 actor 即可以轻松完成。
3. *最佳策略模型*：AlphaGo Zero 会追踪当前的最佳策略模型，并只用该模型的自我对弈数据填充其重放缓冲区。候选模型必须达到 $\geqslant 55\%$ 的胜率才能取代最佳模型。实现这一点相当于在主循环中增加了一个 `if` 模块。
4. *蒙特卡洛树搜索（MCTS）*：MCTS 可以作为策略图的子程序来处理，也可以选择使用 Ray 进行并行化。

HyperBand 和 PBT 算法：Ray 实现了超参数搜索算法的分布式实现，如 HyperBand 和 PBT 算法。只需为每个 RLlib 中的算法增加大约 15 行代码，我们能够使用上述算法来评估 RLlib 中的算法。我们注意到，这些算法在使用分布式控制模型时，由于需要修改现有的代码来插入协调点，因此这些算法的集成难度

还挺大（图 A-3）。RLlib 使用短运行任务就避免了这个问题，因为在任务之间可以很容易做出控制决策。



(a) Ape-X　　(b) PPO-ES　　(c) AlphaGo Zero

图 A-6　在 RLlib 的分层控制模型中，复杂的强化学习架构很容易被实现。这里蓝线表示数据传输，橙线表示轻量开销方法调用。每个 train() 函数的调用包含了各个组件之间的一系列远程调用。

A.4　框架性能

在本章节中，我们将讨论 Ray 的属性和其他对 RLlib 至关重要的优化。

A.4.1　单计算节点性能

有状态的计算：任务可以通过 Ray actor 与其他任务共享可变状态。这对于在第三方模拟器或神经网络权重等有状态对象上操作和突变的任务来说至关重要。

共享内存对象存储：强化学习算法涉及到共享大量数据（例如采样和神经网络权重）。Ray 通过允许数据对象在工人之间直接传递数据对象来高效地支持这一点。在 Ray 中，同一台机器上的子任务也可以通过共享内存读取数据对象，而不需要复制额外数据。

向量化：RLlib 可以批处理策略评估，提高硬件利用率（图 A-7），支持批处理环境，并在行为体之间以标准数组格式高效传递经验数据。

A.4.2　分布式性能

轻量级任务：Ray 中的远程调用如果是在同一机器上，那么开销在 200µs 左右。当机器资源饱和时，任务会溢出到其他节点，延迟增加到 1ms 左右。这使得并行算法可以无缝地扩展到多台机器，同时保留了单节点的高吞吐量。

嵌套并行化：通过组合分布式组件构建强化学习算法会产生多级嵌套并行调用，如图 A-1所示。由于各个组件所做的决策可能会影响到下游的调用，因此调用图也必须是原生动态的。Ray 允许任何 Python 函数或类方法作为轻量级任务远

程调用，例如，`func.remote()` 会远程执行 `func` 函数，并立即返回一个占位符结果，该结果以后可以被检索或传递给其他任务。

资源管理：Ray 允许远程调用指定资源需求，并利用资源感知调度器来保护组件的性能。如果缺失这个功能，分布式组件可能会不适当地分配资源，从而导致算法运行效率低下甚至失败。

故障容错和滞后缓解：故障事件在规模化运行时会变得十分棘手。RLlib 利用了 Ray 的内置容错机制，利用可抢先的云计算实例降低了成本。同样，滞留者会显著影响分布式的规模化的算法。RLlib 支持通过 `ray.wait()` 的通用方式缓解影响。例如，在 PPO 中，我们用这种策略删除最慢任务，但代价是有一定的采样偏差。

数据压缩：RLlib 使用 LZ4 算法对传输数据进行压缩。对于图像而言，LZ4 在压缩率为 1GB/s 每 CPU 核心的情况下，减少了至少一个数量级以上的网络流量和内存占用。

A.5 评估测试

采样效率：策略评估是所有强化学习算法的重要组成部分。在图 A-7 中，我们对从测评略评估器采样进程收集样本的可扩展性进行了基准测试。为了避免瓶颈，我们使用四个中间行为体进行聚合。Pendulum-CPU 在运行一个小的 64×64 全连接的网络作为策略时，速度达到每秒超过 150 万个动作操作数。Pong-GPU 在 DQN 卷积架构上采样速度接近 20 万每秒。

大规模测试：我们使用 Redis、OpenMPI 和分布式 TensorFlow 评估了 RLlib 在 ES、PPO 和 A3C 三种算法上的性能，并与专门为这些算法构建的专用系统进行了比较。所有实验中都使用了相同的超参数。我们使用 TensorFlow 为所评估的 RLlib 算法定义了神经网络。

RLlib 的 ES 实现在 Humanoid-v1 任务上的扩展性很好，如图 A-8 所示。使用 AWS m4.16xl CPU 实例中 8192 个内核，我们在 3.7 分钟达到了 6000 的累计奖励，比已公布的最佳结果还要快一倍。对于 PPO 算法，我们在相同的 Humanoid-v1 任务上进行评估。从一个 p2.16xl 的 GPU 实例开始，然后添加 m4.16xl 的 GPU 实例进行拓展测试。这种具有成本效益的本地策略优化器要显著优于已有的 MPI 方案（表 A-3），图 A-8 也同样展示了这一点。

我们在 x1.16xl 机器上运行 RLlib 的 A3C 算法，使用异步策略优化器在 12 分

钟内解决了 PongDeterministic-v4 任务，使用共享 param-server 优化器在 9 分钟内解决了 PongDeterministic-v4 任务，性能与调优后的基线相匹配。

多 GPU：为了更好地理解 RLlib 在 PPO 实验中的优势，我们在一个 p2.16xl 实例上进行了基准测试，比较了 RLlib 的本地多 GPU 策略优化器和表 A-3中的全局规约策略优化器。事实上，不同的策略在不同条件下表现更好，这表明策略优化器是一个有用的抽象。

表 A-3　一个专门的多 GPU 策略优化器在数据可以完全装入 GPU 内存时，表现优于全局规约。这个实验是针对有 64 个评估进程的 PPO 进行的。PPO 批处理量为 320k，SGD 批处理量为 32k，我们在每个 PPO 批处理量中使用了 20 次 SGD。

策略优化器	梯度计算资源	任务	SGD 每秒吞吐量
全局规约	4GPU，评估模式	Humanoid-v1	33 万
		Pong-v0	2.3 万
	16GPU，评估模式	Humanoid-v1	**44 万**
		Pong-v0	**10 万**
本地多 GPU	4GPU，驱动模式	Humanoid-v1	**210 万**
		Pong-v0	无（内存不够）
	16GPU，驱动模式	Humanoid-v1	170 万
		Pong-v0	**15 万**

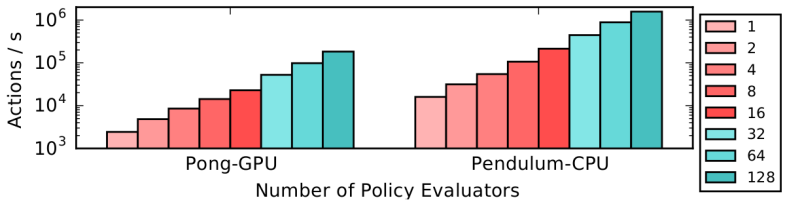
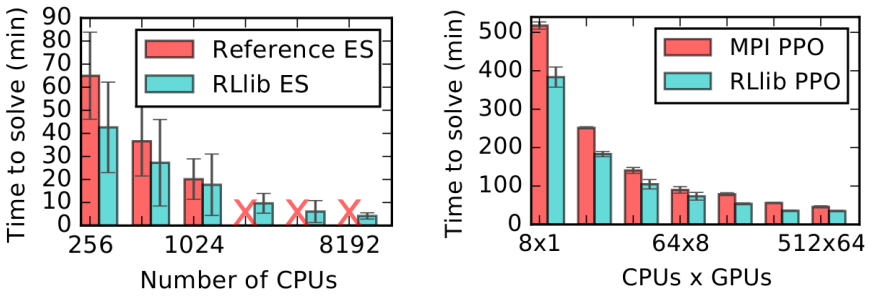


图 A-7　策略评估的吞吐量从 1 到 128 核几乎呈线性扩展。GPU 上的 PongNoFrameskip-v4 每秒操作数从 2400 到约 20 万，CPU 上的 Pendulum-v0 每秒操作数从 1.5 万到 150 万。我们使用单个 p3.16xl AWS 实例进行 1-16 个 CPU 核心上的评估，和 4 个 p3.16xl 实例的集群进行 32-128 个 CPU 核心的评估，将 Ray actor 均匀地分布在每台机器上。策略评估器一次为 64 个智能体计算行动，并共享机器上的 GPU。

A.6　相关工作

在本工作之前有许多强化学习库，它们通常通过创建一个长期运行的程序副本进行扩展，每个副本都参与协调整个分布式计算，因此它们不能很好地推广到



(a) ES (b) PPO

图 A-8　在 Humanoid-v1 任务上达到 6000 的奖励所需的时间。RLlib 实现的 ES 和 PPO 的性能优于已有实现。

复杂的体系结构。RLlib 使用带有短期任务的分层控制模型来让每个组件控制其自己的分布式执行，从而使更高级别的抽象（例如策略优化器）可用于组成和扩展强化学习算法。

除了强化学习之外，学术界还进行了很多努力来探索不同深度学习框架之间的组成和整合。诸如 ONNX，NNVM 和 Gluon 这些框架定位于不同硬件与不同框架的模型部署，并提供了跨库的优化。现有深度学习框架也为强化学习算法中出现的基于梯度的优化模块提供支持。

A.7　结论

RLlib 是一个强化学习的开源框架，它利用细粒度的嵌套并行机制在各种强化学习任务中实现了最优性能。它既提供了标准强化学习算法的集合，又提供了可扩展的接口，以方便地编写新的强化学习算法。

书面翻译对应的原文索引

[1] LIANG E, LIAW R, NISHIHARA R, et al. Rllib: Abstractions for distributed reinforcement learning[C/OL]//Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. 2018: 3059-3068. http://proceedings.mlr.press/v80/liang18b.html.

附录 B　实验原始数据

表 B-1　CartPole-v0 实验原始数据

平台	算法	1	2	3	4	5	平均值	标准差
RLlib	PG	19.43	17.62	18.27	17.38	23.61	19.26	2.29
	DQN	36.21	27.79	25.82	30.42	22.57	28.56	4.60
	A2C	42.84	55.18	63.22	55.45	72.91	57.92	9.94
	PPO	27.18	29.08	54.44	39.65	72.64	44.60	17.04
Baselines	PG	未实现						
	DQN	超过 1000 秒未完成任务，但在 1000 秒之后完成						
	A2C	超过 1000 秒未完成任务，且不能收敛						
	PPO							
PyTorch-DRL *	PG	超过 1000 秒未完成任务，且不能收敛						
	DQN	24.21	53.96	24.42	28.17	27.12	31.58	11.30
	A2C	超过 1000 秒未完成任务，且不能收敛						
	PPO	9.30	21.11	22.26	30.91	36.39	23.99	9.26
Stable-Baselines	PG	未实现						
	DQN	45.84	108.08	51.31	59.56	202.58	93.47	58.05
	A2C	81.00	44.06	56.70	47.81	58.23	57.56	12.87
	PPO	20.64	53.35	21.50	57.78	20.67	34.79	17.02
rlpyt	PG	rlpyt 对于离散动作空间非 Atari 任务的支持不友好，可参考 此处						
	DQN							
	A2C							
	PPO							
天授	PG	1.65	4.98	14.79	6.01	3.03	6.09	4.60
	DQN	5.14	6.32	7.62	5.41	5.97	6.09	0.87
	A2C	7.40	5.05	4.12	6.53	8.68	6.36	1.63
	PPO	30.12	25.21	43.53	22.63	37.59	31.82	7.76

*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为“训练过程中连续 20 次完整游戏的平均总奖励大于等于 195”。

表 B-2　Pendulum-v0 实验原始数据

平台	算法	1	2	3	4	5	平均值	标准差
RLlib	PPO	126.91	105.82	131.34	195.46	58.56	123.62	44.23
	DDPG	312.93	329.85	307.26	313.70	309.75	314.70	7.92
	TD3	139.18	158.29	144.52	158.24	149.29	149.90	7.54
	SAC	102.93	95.21	89.95	102.04	96.98	97.42	4.75
Baselines	PPO	804.92	832.88	444.79	733.01	911.53	745.43	160.82
	DDPG	超过 1000 秒未完成任务，且不能收敛						
	TD3	未实现						
	SAC							
PyTorch-DRL *	PPO	PyTorch-DRL 中的 PPO 算法在连续动作空间任务中会报异常错误						
	DDPG	42.50	56.21	69.02	57.53	69.99	59.05	10.03
	TD3	43.97	46.44	46.06	91.04	60.10	57.52	17.71
	SAC	113.88	37.82	40.08	64.38	62.84	63.80	27.37
Stable-Baselines	PPO	206.71	284.84	271.73	271.81	263.58	259.73	27.37
	DDPG	206.58	384.53	135.68	140.45	270.36	277.52	92.67
	TD3	86.22	142.88	91.53	88.77	89.34	99.75	21.63
	SAC	251.22	123.47	165.39	42.07	42.10	124.85	79.14
rlpyt	PPO	rlpyt 并未提供使用 PPO 的任何示例代码，经尝试无法成功跑通						
	DDPG	180.56	130.14	105.95	106.69	94.51	123.57	30.76
	TD3	106.37	98.42	136.02	119.05	105.12	113.00	13.31
	SAC	122.58	169.20	104.50	141.96	125.77	132.80	21.74
天授	PPO	17.64	14.97	20.29	13.28	14.70	16.18	2.49
	DDPG	24.34	51.15	30.25	36.46	44.09	37.26	9.55
	TD3	38.22	52.67	42.15	50.32	36.85	44.04	6.37
	SAC	35.56	35.08	35.61	36.83	37.04	36.02	0.77

*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为 “训练过程中连续 20 次完整游戏的平均总奖励大于等于-250”。

在学期间参加课题的研究成果

发表的学术论文

[1] Shihong Song*, Jiayi Weng*, Hang Su, Dong Yan, Haosheng Zou, and Jun Zhu. Playing FPS Game with Environment-aware Hierarchical Reinforcement Learning. Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019.

[2] Jiayi Weng, Tsung-Yi Ho, Weiqing Ji, Peng Liu, Mengdi Bao, and Hailong Yao. URBER: Ultrafast Rule-Based Escape Routing Method for Large-Scale Sample Delivery Biochips. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, TCAD 2018.