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

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。

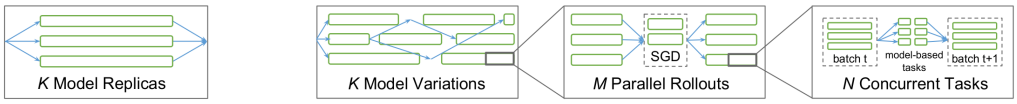
A.1　引言

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

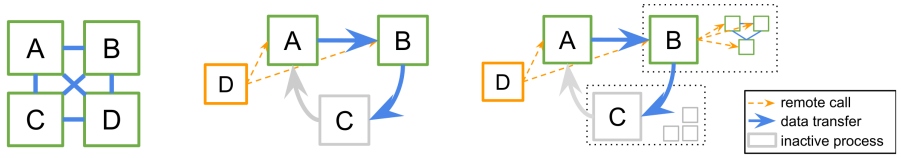


(a) 深度学习　　(b) 深度强化学习

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。

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

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。

```
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。

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()))
```

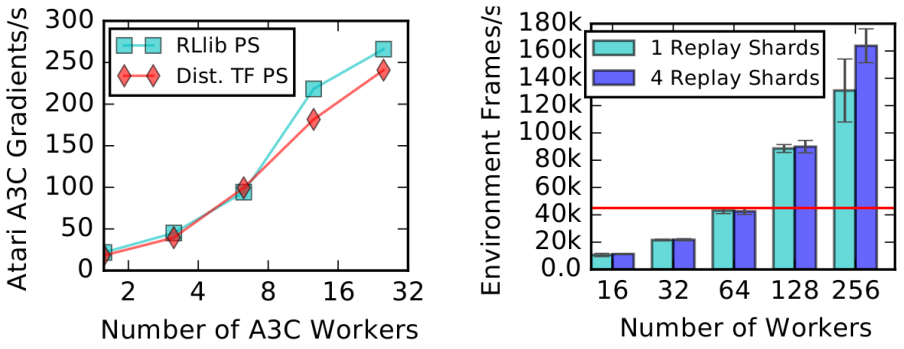
(c) 异步计算

```
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



(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。

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.3.4　RLlib 抽象模式的完备性和普适性

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

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。

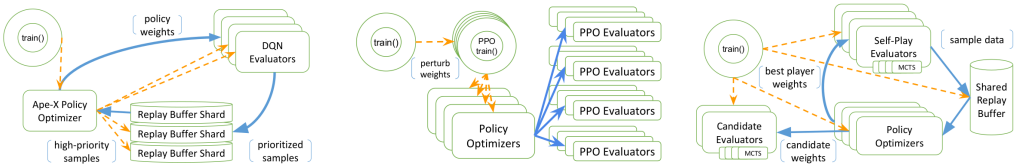
AlphaGo：我们用 Ray 和 RLlib 的抽象组合来描述 AlphaGo Zero 算法的可扩展实现方法。

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.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) 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.4.1 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**

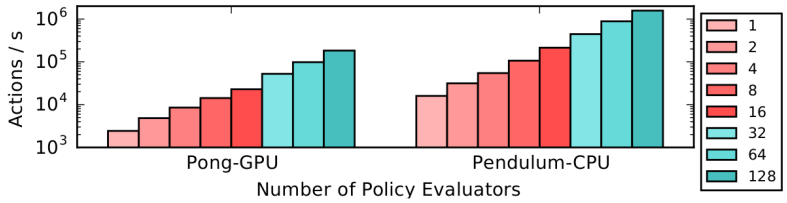
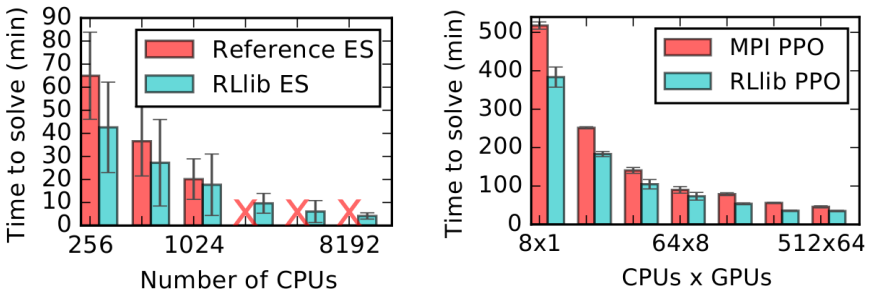


图 A-7　策略评估的吞吐量从 1 到 128 核几乎呈线性扩展。GPU 上的 PongNoFrameskip-v4 每秒操作数从 2400 到约 20 万，CPU 上的 Pendulum-v0 每秒操作数从 1.5 万到 150 万。我们使用单个 p3.16xl AWS 实例进行 1-16 个 CPU 核心上的评估，和 4 个 p3.16xl 实例的集群进行 32-128 个 CPU 核心的评估，将 Ray actor 均匀地分布在每台机器上。策略评估器一次为 64 个智能体计算行动，并共享机器上的 GPU。



(a) 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。

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] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3059–3068, 2018. URL http://proceedings.mlr.press/v80/liang18b.html.