



博士研究生综合考试报告

题目： 云计算环境下的人工智能相关
技术研究

姓 名： 李炎

学 号： 2001111305

院 系： 信息科学技术学院

专 业： 计算机软件与理论

研究方向： 云计算与普适计算

导 师： 梅宏

二〇二一年四月

摘要

21世纪10年代以来，云计算和人工智能可谓计算机科学领域最为炙手可热的两个研究方向。以虚拟化、资源管理和服务化为代表的云计算核心技术在近十余年里取得了丰硕的研究成果。当前，云计算已成为工业化社会重要的信息基础设施，支撑并推动着大数据和人工智能产业的快速发展。与此同时，人工智能技术在近十年内也相继在计算机视觉、自然语言处理等多个领域取得了突破，“智能化”已然成为现代社会的重要标签之一。

本文将以上述两大技术的蓬勃发展为背景，研究云计算与人工智能相互影响、相互支持、相辅相成的相关技术。本文将按照如下几章展开。

第一章对相关的技术背景做出介绍。首先对云计算近十年的发展做简要回顾，并介绍具有代表性的若干核心技术。其次对人工智能近十年的发展做简要概述，阐述其在计算机视觉、自然语言处理等子领域的代表性科研成果。最后分析二者在交叉领域现有的相关研究。

第二章开始探究二者的关系，研究云计算环境中用以支持人工智能的系统软件技术。随着人工智能算法和系统研究的发展，其训练-测试-部署的流程愈发复杂。很多云厂商基于本地化的云原生技术，构建了一站式的人工智能开发-部署软件栈供用户使用。同时，伴随着新的云计算模式（如无服务计算 serverless）的产生，工业界和学术界也在探究将其应用在人工智能领域，使相关的应用在云上具有更高的弹性。

第三章讨论近些年来云环境下出现的新硬件（如 GPU, AI 专用芯片, Intel-SGX 等）为人工智能技术带来的新的机遇与挑战。硬件的发展（如 GPU, AI 专用芯片等）导致的算力的提升，也是人工智能近年发展迅速的重要原因之一。同时，某些硬件层面安全机制（如 Intel-SGX）的产生，使增强人工智能应用的安全性有的新的潜在解决方案。

第四章从另一角度，即“服务于云计算系统架构的 AI 技术”这一视角展开，研究人工智能对云计算的增强技术。云计算本质上是一个巨大的公共资源池，用户如何在资源池中选取资源，云厂商如何为不同的用户调度资源，是云计算领域两个重要的话题。近五年来，该领域的研究者开始尝试利用一系列基于机器学习的算法来辅助解决上述两个问题。

第五章总结了上述三个方向的重要文献和相关研究团队概况。

第六章介绍了作者下一步的研究计划。

关键词：云计算，人工智能，机器学习，新硬件

目录

第一章 引言	1
1.1 云计算的基本概念	1
1.1.1 云计算的传统服务模型	1
1.1.2 云计算的新兴服务模型	2
1.2 人工智能技术近年的发展	3
1.3 云计算和人工智能交叉领域的常见研究问题	3
1.3.1 基于公有云服务的机器学习平台	4
1.3.2 利用新的计算模式在云上运行机器学习 pipeline	4
1.3.3 新硬件对人工智能的影响	5
1.3.4 利用机器学习算法解决配置优化问题	5
1.3.5 利用机器学习算法解决资源调度问题	5
第二章 面向人工智能的云计算系统软件研究	7
2.1 一站式的云上机器学习系统	7
2.1.1 AWS Sagemaker	7
2.2 基于公有云服务的第三方机器学习系统	13
2.2.1 基于云厂商动态资源的机器学习模型训练系统	13
2.2.2 基于 FaaS 的模型训练系统	23
2.2.3 基于公有云服务的模型部署系统	27
2.2.4 基于 FaaS 的模型部署系统	30
2.3 小结	33
第三章 云环境中的异构硬件为人工智能带来的机遇与挑战	35
3.1 云环境中机器学习作业对 GPU 的利用	35
3.1.1 GPU 简介	35
3.1.2 One Job on Multi GPUs: 机器学习集群中的 GPU 调度算法	37
3.1.3 Multi Jobs on one GPU: 机器学习集群中的 GPU 共享机制	41
3.2 云环境中机器学习作业对 SGX 的利用	44
3.2.1 SGX 技术简介	45
3.2.2 基于 SGX 技术的上层应用	47
3.2.3 集群中 SGX 资源的管理	48

3.2.4 在机器学习任务中应用 SGX 技术	51
3.3 小结	52
第四章 人工智能对云计算的增强技术研究	55
4.1 基于机器学习算法的云资源配置优化技术	55
4.2 基于机器学习算法的云资源调度优化技术	55
4.3 小结	55
第五章 重要文献与研究团队总结	57
第六章 下一步的研究设想	59
6.1 基于 k8s 的 SLO 可感知的机器学习模型部署系统	59
6.2 SGX 环境下机器学习模型部署干扰研究	59
参考文献	61

第一章 引言

1.1 云计算的基本概念

云计算（Cloud Computing），根据美国国家标准技术研究所（NIST）的定义，指的是一种可以实现对可配置计算资源共享池（如网络、服务器、存储、应用和服务）进行随时随地、便捷、按需网络访问模型。这些资源可以迅速地配分配和释放，并且这个过程只需要足最低限度的资源管理工作以及与服务提供商最少的交互。美国亚马逊公司再 2006 年 3 月推出了 Amazon Web Service（AWS），这一事件一般被认为代表着云计算时代的正式开启。经过十几年的发展，凭借着“方便易用、弹性伸缩、按需服务”的技术特征，云计算概念已被广泛接受，云计算产业取得了商业上的巨大成功，云计算平台已成为当今社会的关键信息基础设施，云计算技术为大数据、人工智能的领域的蓬勃发展提供了重要的支撑作用。

1.1.1 云计算的传统服务模型

NIST 将云计算分为了三种服务模型。

这三种服务模型分别是基础设施即服务（Infrastructure as a Service, IaaS）、平台即服务（Platform as a Service, PaaS）以及软件即服务（Software as a Service, SaaS）。IaaS 为消费者提供用来运行应用的计算资源，包括服务器、存储、网络等。其中虚拟机是云厂商提供的最核心的 IaaS 产品。与 IaaS 只提供最基础的底层资源不同，PaaS 强调为消费者提供云开发环境，除计算资源意外，PaaS 为用户提供中间件开发，运行平台及工具，帮助用户更方便地开、管理、测试和运行应用。SaaS 是厂商提供的基于云的软件，用户无需下载安装软件，通过浏览器即可访问服务。

图1.1给出了云计算三种服务模型的代表产品。亚马逊公司的 AWS EC2，谷歌公司的 Google Compute Engine 以及阿里云公司的 ECS 都是典型的 IaaS 产品。其主要服务形态是云厂商向消费者售卖虚拟机或者裸金属服务器以及连带的网络、存储等附属产品。PaaS 的代表性产品包括 AWS Beanstalk、Google App Engine、Microsoft Azure App Services 等。此类产品为用户提供再云中快速部署和管理应用的能力，提供包括应用扩容，负载均衡，应用监控和安全管理等功能。相比于 IaaS 仅售卖以虚拟机为主的基础设施，PaaS 降低了用户开发、管理、运维应用的成本，使得用户可以更加专注于构建应用本身。在云计算已经发展了十几年的当今时代，越来越多的 SaaS 产品涌现了出来。谷歌公司开发的 Google Docs、Google Maps 以及微软公司开发的 Microsoft Office 365

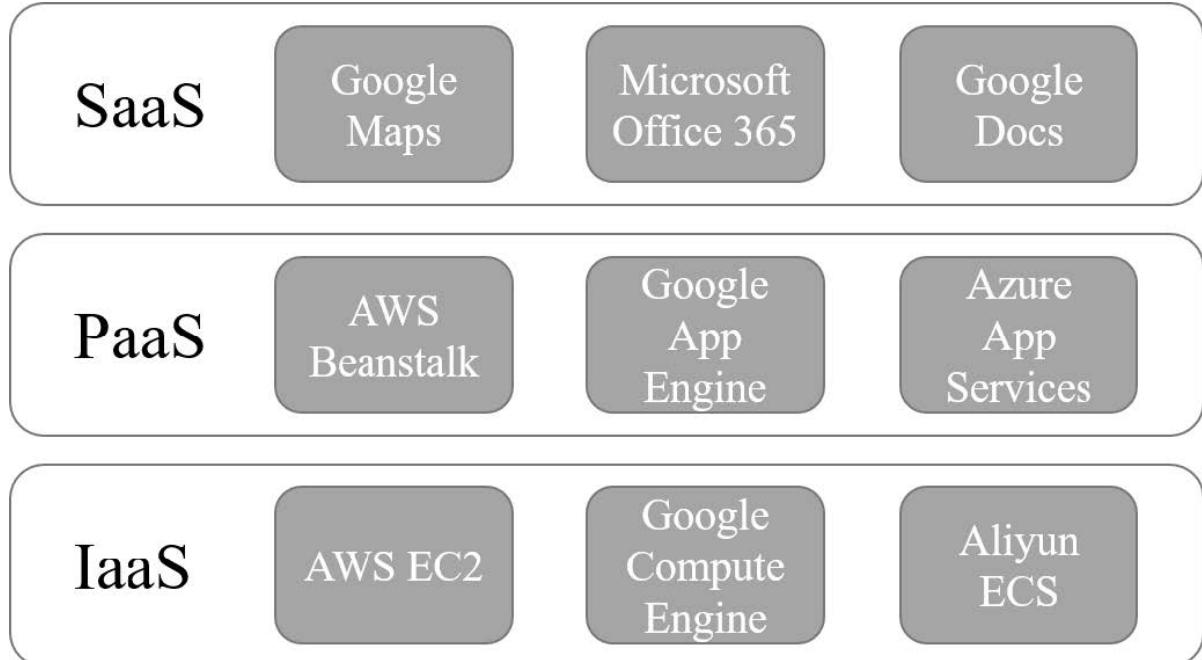


图 1.1 云计算服务模型代表产品

都是典型的 SaaS 产品。以 Google Docs 为例，与传统的文件处理办公软件相比，用户无需在本地花费大量存储空间来安装软件，只需要打开浏览器，输入 URL，即可使用 Google Docs 服务处理文件，并且所有的文件都会被及时同步保存到云端。另一个代表性的 SaaS 产品是 Google Maps。Google Maps 与 Google Docs 类似，为个人用户提供在浏览器中直接使用的地图服务。与传统软件相比，SaaS 在使用方式上具有方便灵活，跨平台的特性，同时用户存储在云端的数据经过云厂商的冗余备份也具有更高的可靠性。

1.1.2 云计算的新兴服务模型

云计算发展至今日，其服务模型已经不严格局限于 NIST 最初总结的这三种基本形态，世界各地各领域的研究者们已经提出了众多不同的 X as a Service，包括 Blockchain as a Service, Sensing as a Service, Workspace as a Service 等。与传统的三种服务形态相比，这些服务不单纯是硬件服务或者软件服务，其结合二者的特点，面向特定的领域方向进行更深度的定制，如区块链、物联网、分布式共识等。服务形态的日益丰富，服务内容的日益复杂体现了云计算更见领域化，精细化的发展趋势。而近几年来最热门的概念莫过于 FaaS，即 Function as a Service。

FaaS 是一种新兴的计算模式，亦被称为 Serverless Computing。从字面理解，Serverless Computing 即为“无服务器计算”之意。然后，其并非意味着真的没有服务器，而已说开发者不用过多考虑服务器的相关问题。在传统的 IaaS 服务中，开发者需要自己

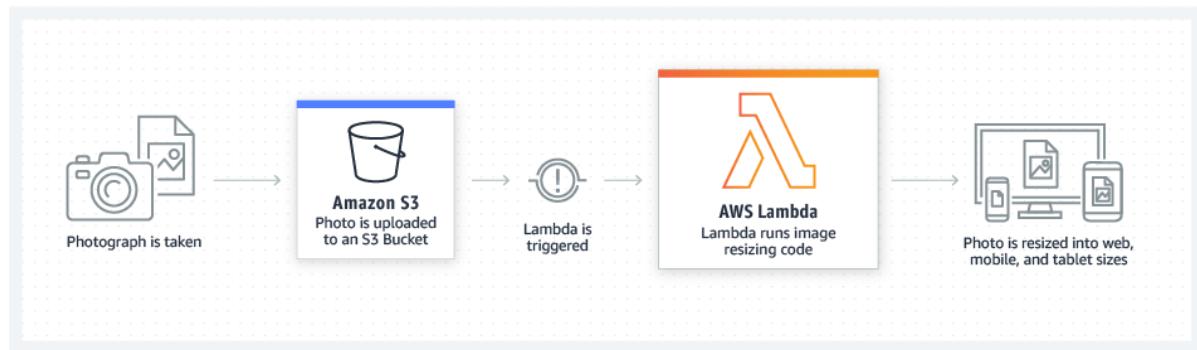


图 1.2 AWS Lambda 中示例程序：照片大小调整

惊醒服务器管理与运维，负责服务的发布，在流量变化时对服务器集群进行扩容或缩容。而在 FaaS 中，开发者只需要关注业务逻辑，至于服务的发布、管理、弹性伸缩等，则交由云厂商来完成。

FaaS 背后的机制一般是以容器技术为基础的。典型地，开发者上传自己的业务代码后，云厂商并不会直接收费。当对该服务的请求到来之时，云厂商将启动一系列容器来运行该服务，从而对用户的请求进行响应。通常而言，开发者指定的服务会与某些时间绑定（hook），在发生该事件时，立即触发开发者定义的服务。我们以 AWS 的 serverless computing 服务 Lambda 中的一个示例程序为例，讲述整个流程。图1.2所示的是一个为照片调整大小的服务。该服务与 AWS S3 (AWS 的对象存储服务) 的上传事件绑定，当云厂商检测到有用户向 S3 上传图片时，会立即触发开发者定义的图片大小调整函数。整个流程中，开发者需要关注的只有第四步的函数开发工作，至于该函数的横向拓展，全部由云厂商来负责。

主流的云厂商均提供了 FaaS 服务，例如 AWS 的 Lambda，阿里云的函数计算等，近年来越来越受到开发者的青睐。一方面是因为它的高弹性，易于开发。另一方面则是因其细粒度的收费模式。通常而言，FaaS 的服务是按照请求次数进行收费。当函数闲置时，并不产生额外的费用。

1.2 人工智能技术近年的发展

TODO: 简介近些年人工智能技术的发展

1.3 云计算和人工智能交叉领域的常见研究问题

云计算和人工智能是两个息息相关的热门领域。一方面，以深度学习为典型代表的人工智能技术在当今社会被应用的越来越广泛，研发、调试、发布新的模型的需求日益增长。与这种发展趋势对应，多数主流的云厂商都提供了机器学习模型训练-测试-部

署的 pipeline。学术界也不断探索“云上机器学习”这一话题，利用新的计算模式（如 FaaS）在云上以更便捷、更经济高效地开展 ML 模型的训练和部署。同时得益于近年硬件技术的发展，多种新硬件（多体现为人工智能的加速芯片）在云环境中得到应用，进一步方便机器学习用户将整个开发流程迁移到云端。另一方面，机器学习技术也越来越多被用于解决云计算中常见的问题。例如使用推荐算法解决置优化问题和利用强化学习解决云环境中的资源调度问题。下文对这几个常见问题做简单概述，具体研究将在后续几章展开。

1.3.1 基于公有云服务的机器学习平台

1. 产业界

主流的云厂商都提供了面向机器学习的平台系统，例如 AWS 和 Sagamaker[13, 18, 23]，Azure 的 Azure ML Studio[8] 等。这些基于云的平台提供了一站式调试、训练和部署 ML 模型的能力。一般而言此类平台被视为 SaaS 类的服务，因为其是基于云资源构建的上层软件栈，使得用户能够直接使用 web 的方式使用。例如 Sagamaker 就支持用户直接在浏览器中用 jupyter notebook 编写和调试模型代码。

2. 学术界

一般而言，产业界的平台系统面向的是一般性用户的普遍需求。因此，对于有特殊需求的用户，通常会有与产业界的解决方案并行的工作。例如，为了以尽可能低的成本在云上完成模型的训练，相关工作 [11, 17] 尝试利用云上的动态资源（价格低但是稳定性/可用性低）进行模型的训练，并辅以一定的策略增强其可靠性。再比如，机器学习模型的在线服务会有低延迟、高吞吐率的要求。为了实现上述需求，相关工作 [31] 利用云上的多种资源（如 IaaS, FaaS 等），根据负载的动态变化，敏捷地在不同资源之间切换，充分利用不同类型资源的优点，规避掉其缺点，实现高效、经济的模型在线服务。一般而言，学术界的此类研究构建于云厂商服务的上层，是一种 Cloud-of-Clouds 的模式。

1.3.2 利用新的计算模式在云上运行机器学习 pipeline

以 FaaS 为代表的新型云计算模式，以其高弹性、灵活的计费方式等特点吸引了众多研究者的注意。一般而言，用来训练机器学习模型的集群大小是固定的，很难动态地进行伸缩。同时，开发者还需要自行对该集群进行维护和管理，从而徒增一些不必要的时间和人力开销。因此，学术界开始探讨如何使得机器学习工作流享用到 FaaS 的诸多优点 [27]，从而使得其计算资源在模型训练时能按需伸缩，结束训练后自动将模型存储在持久化存储中，计算资源随即释放。

除了模型训练，也有部分研究者尝试利用 FaaS，以 serverless 的模式部署模型。虽然这一想法非常自然，但是还有诸多问题需要解决。例如，现有的 serverless 服务，如 AWS Lambda，其单个 instance 所能使用的内存有限，也无法使用 GPU 等加速芯片，同时其冷启动时间也较长（相比于服务对 latency 的要求）。如何优化 FaaS 的系统架构，使其更适合模型的部署，也是一个值得研究的问题。

1.3.3 新硬件对人工智能的影响

TODO：介绍新硬件对人工智能的影响

1.3.4 利用机器学习算法解决配置优化问题

公有云厂商所提供的计算资源类型和计费模式越来越复杂。据不完全统计，截至目前，AWS EC2 提供了超过 400 种配置的虚拟机，阿里云提供了超过 500 种虚拟机。除了虚拟机配置不同之外，其收费模型也存在多种。例如，除了传统的包年包月、按量付费等，还有抢占式实例（在 AWS 中成为 Spot Instance）等计费方式。在此场景中，用户所面临的一个重要问题在于如何为自己的应用程序/负载选择合适配置和收费方式的资源。

部分研究者利用机器学习算法来解决此类问题。有的将其看作一个推荐问题 [15]，用协同过滤等推荐系统中常用的算法为不同的应用程序匹配合适的配置。有的将其建模为回归问题 [21, 26, 29, 33]，将应用程序的信息、资源的配置信息等作为输入，预测在某种配置下某个应用程序的性能或者花费，从而得到最佳的配置。还有的将其视作在线优化问题 [1, 5]，利用贝叶斯优化等方法，通过有限次的尝试逐渐逼近最优解。

1.3.5 利用机器学习算法解决资源调度问题

资源调度是操作系统、分布式系统和云计算领域中的经典问题，其本质在于为系统中的每个任务分配合适的资源，从而使得系统达到某种状态，如资源利用率最高、闲置资源最少或者吞吐率最高等。近年来深度学习的流行，尤其是强化学习的兴起，使得部分研究者开始思考利用机器学习算法解决资源调度问题 [6, 7, 20]。一般而言，此类算法旨在从过去的任务执行记录中“学习”相关的规律，以指导为未来系统的调度动作。

第二章 面向人工智能的云计算系统软件研究

以深度学习为代表性技术的人工智能领域在 21 世纪 10 年代再度兴起，社会各界对于人工智能的需求也愈发旺盛。遍布超市和餐馆中的扫脸支付技术、智能手机上的语音智能助手、工厂中检测废件的自动检测装置等，都离不开人工智能技术的加持。具体的，上述场景都需要适当的机器学习模型在线部署以提供服务。

机器学习的工作流一般遵循如下几个步骤：1) 模型开发与调试。在此阶段中，开发者根据具体的场景，编写并调试模型代码。2) 模型参数调整。在这个步骤中，开发者使用训练集不断调整模型的超参数，使其在测试集上的表现符合某种标准（如准确率高于某个阈值）。3) 模型部署，开发者将开发完成的模型部署到对应的场景中对外提供服务。

为了方便开发者专注到模型的开发流程中，各大主流云厂商均实现了支持机器学习模型开发-调试-部署整个流程的软件栈。同时，云厂商提供的一般性的平台可能无法满足用户特定的需求。因此针对具体场景，学术界也提出了一系列基于云服务的机器学习软件，以达到降低开发成本、提高模型在线服务的质量等目标。本章对上述内容涉及到的相关工作展开具体研究。

2.1 一站式的云上机器学习系统

本节先以 AWS Sagemaker 为代表，介绍公有云厂商一站式的云上机器学习系统。其它厂商中类似系统中对计算模型的抽象、使用方式等均与 Sagemaker 大同小异。

2.1.1 AWS Sagemaker

AWS 作为公有云领域的先驱者，在云计算的前沿技术领域一直处于领先地位。其某些代表性技术甚至成为了多数公有云厂商所共识的标杆和规范。在机器学习系统这一分领域，其代表性系统为 AWS Sagemaker。

Sagemaker 是一个实现和产品化机器学习模型的框架 [13]，用户可以将其模型训练需要的数据存储到 AWS 的对象存储服务 S3 中。然后，用户可以通过 web 的方式访问部署在云端的 Jupyter Notebook 服务，在线访问数据、编写并调试代码。当模型训练完毕后，可以使用 Sagemaker 内置的推理服务对模型进行发布。用户在发布时还可以根据平台的不同（云端或边缘节点）将模型打包编译成不同的版本。此外，Sagemaker 还内置了一些常用的算法和数据集，方便开发者直接调用。

在训练算法和系统机制方面, Sagemaker 旨在解决工业规模的模型训练场景中的如下几个常见问题:

- **支持增量式的训练和模型更新。** 在真实的工业场景中, 几乎不存在完全静态的数据集。用来训练模型的数据集大多都是不断增长的。例如电商网站的用户行为数据, 每天都在以相当的速度增长。在这样的动态数据上训练模型, 势必要进行如下权衡: 在全量数据上进行训练, 可以获得质量更高的模型, 然而时间和经济上的开销却会非常高; 在最新更新的数据上(例如最近几天的数据)进行训练, 可以快速的得到新的模型, 却有可能在一定程度上牺牲模型的准确性。
- **容易估算训练模型产生的花销。** 对于体量非常大的数据集, 用户需要较为准确地估计训练模型将会产生的时间和经济花销。当今的云计算一般遵循按量付费的收费模式, 因此云上的机器学习用户会格外关注花销的问题。
- **支持暂停和恢复模型训练, 有一定的弹性。** 生产环境中, 大模型的训练通常包含跨越数十甚至上百台机器的并发任务。在一些场景中, 由于超参数调整的需要或者计算资源的限制, 开发者需要对这些任务进行中断和恢复。这就要求云上的 ML 系统能够支持大型训练任务中断和恢复时中间结果的保存和复原。
- **能够处理非持久性数据。** 在很多场景中, 数据并不一定是持久化的, 也会有很多“瞬时”的数据, 例如直播时的视频流等。这些数据一般不会被持久化保存, 因此如何支持对这些数据的挖掘和学习也是一个重要的问题。
- **支持自动调参。** 自动调参是一项非常耗时耗力的工作, 特别是在生产环境的大数据集下。因此, 如何能够支持高效的自动调参, 方便用户选取合适的模型, 对于云上的 ML 系统而言也是非常重要的。

下面将从计算模型、支持算法和具体的实现方式介绍 Sagemaker 如何解决上述问题。

2.1.1.1 计算模型

Sagemaker 将机器学习训练的工作流抽象为三个阶段: initialize, update 和 finalize。 initialize 阶段: 对相关的变量进行初始化。 update 阶段: 系统以数据流及其状态作为输入, 根据用户编写算法更新状态。在 Sagemaker 的后台, 状态更新是在多台计算节点上同步执行的。 finalize 阶段: 对之前所有的更新进行汇总, 并输出最后的结果(对于机器学习任务而言, 一般是一个模型)。通过上述抽象, 用户只需编写 initialize、update 和 finalize 三个函数, 就可以将整个训练过程交由 Sagemaker, 等待最后的模型输出。

算法 1 用伪代码的形式讲述了三个阶段的具体流程。这里以计算一列数据的中位数为例, 使用随机梯度下降(SGD) 进行求解。对于如下一列数据: $x_1, x_2, x_3, \dots, x_n$, 则由如下公式给出这一列数据的中位数: $\operatorname{argmin}_z \sum_{i=1}^n \|x_i - z\|$ 。算法中的 initialize 函数首先

对 median 和 n 两个参数做初始化。这里 state 可以理解为一个 key-value 存储结构。在 update 阶段，函数将接收到的数据流 data_stream 分为若干 mini_batch，对每个 batch 没的数据，迭代式的对 median 值进行更新，同时更新 n 的值。最后，在 finalize 函数中，对最终的 median 值进行汇总。同时，系统还支持用户在算法的任意位置设置 barrier（第 15 行），以对整个训练过程进行同步操作。

Algorithm 1 Sagemaker 计算模型

```
1: function INITIALIZE(state)
2:   state.initialize('median',0)
3:   state.initialize('n',0)
4:
5: function UPDATE(state,data_stream,synchronized)
6:   for mini_batch ∈ data_stream do
7:     current_median ← state.pull('median')
8:     n ← state.pull('n')
9:     inc ← 0
10:    batch_size ← 0
11:    for item ∈ mini_batch do
12:      if item > current_median then
13:        inc+ = 1
14:      else
15:        inc- = 1
16:      state.push('n',batch_size)
17:      state.push('median', $\frac{inc}{\sqrt{n+batch\_size}}$ )
18:      if synchronized then
19:        state.barrier()
20:
21: function FINALIZE(state)
22:   return state.pull('median')
```

2.1.1.2 支持算法

尽管深度学习在近年越来越流行，特别是在学术界，各种刷新 SOTA（即 state-of-the-art，当前最优）效果的模型层出不穷。然而在产业界，经典的机器学习模型适用性依然非常广。本节对 Sagemaker 支持的广泛应用于产业界的经典机器学习模型做简单介绍。

Linear Learner. 即线性回归模型，一般用来解决回归或者分类问题。在 Sagemaker 中，其支持使用随即梯度下降（SGD）来训练一个线性模型。特别地，Sagemaker 还支持对一个模型指定不同的目标函数，并且并行地训练其对应的多个模型。

Factorization Machines (FM). FM 常用于推荐系统中很多任务（例如 CTR 预估），

是一种适用于分类和回归任务的更为通用的监督学习算法。FM 是对线性模型的一种拓展，考虑了高维稀疏数据集中不同特征之间的关联。在 Sagemaker 中，FM 也是通过 SGD 来进行训练的。

K-Means Clustering. K-Means 聚类是一类无监督学习算法，可以将一个数据集聚为 K 个不同的类别。Sagemaker 实现了一个流式计算版本的 K-means，综合运用了来自随机优化、核心集和在线选址问题的思想。

Principal Component Analysis (PCA). 即主成分分析，是一种通过尽可能保留数据集的方差来对数据进行降维的算法。Sagemaker 实现了两种 PCA 算法。第一类是准确型的，通过适量的采样和特征提取，需要 $O(d^2)$ 量级的内存大小。另一种是粗略型的，通过随即投影来减小内存开销到 $O(kd)$ 。这里 k 和 d 分别代表主成分的个数和数据的维度。

Neural Topic Models (NTM). 在 Sagemaker 中，NTM 是一种用来在大型离散数据集中抽取 latent representation 的无监督学习算法，例如抽取文档中的语料库。为了实现更快的推理，NTM 是基于 variational autoencoder (VAE) 模型实现的。

Time Series Forecasting with DeepAR. 这是一种基于循环神经网络 (RNN) 的概率预测算法。和其它流行的时间序列预测算法不同，DeepAR 使用一系列数据集训练了一个全局的模型，从而重点关注了在生产环境中常见的时间序列预测问题。

2.1.1.3 实现方式

经济和高效是云用户总是在关心的两类问题。Sagemaker 为了满足这样的用户需求，充分利用了其底层的异构资源池，并通过上层的系统软件使得机器学习可以在成百上千台 VM 上横向拓展。

在新硬件上运行 ML 任务。 为了实现在异构的资源上（主要是 CPU 和 GPU），Sagemaker 中的大多数算法使用 MXNet 的库作为使用底层硬件的接口。MXNet 通过一个张量算子构成的计算图来描述一个机器学习算法，同时通过将算子分配到具体的设备进行执行来实现并发地训练模型。

利用参数服务器模型 (Parameter-Server Model, 即 PS Model) 实现分布式训练和参数共享。 如图 2.1 所示，在一个基于 PS Model 的集群中，服务器被分为两种，分别为 worker 和 server。其中 server 负责保存模型所有需要训练的参数，worker 负责具体的训练。在 worker 完成一轮迭代之后，会将更新后的参数变化量 push 到 server 端，并同步等待所有其它的 worker 完成此轮迭代。同步完成后，server 汇总从 worker 端得到的变化量并对参数进行更新，而后 worker 再从 server 端 pull 更新后的参数，并继续进行下一轮迭代。值得一提的是，实际场景中为了追求训练速度，有时也会允许异步的参数更新，即不要求 worker 等待其它 peer 节点完成本轮计算，而是完成本地计算后就

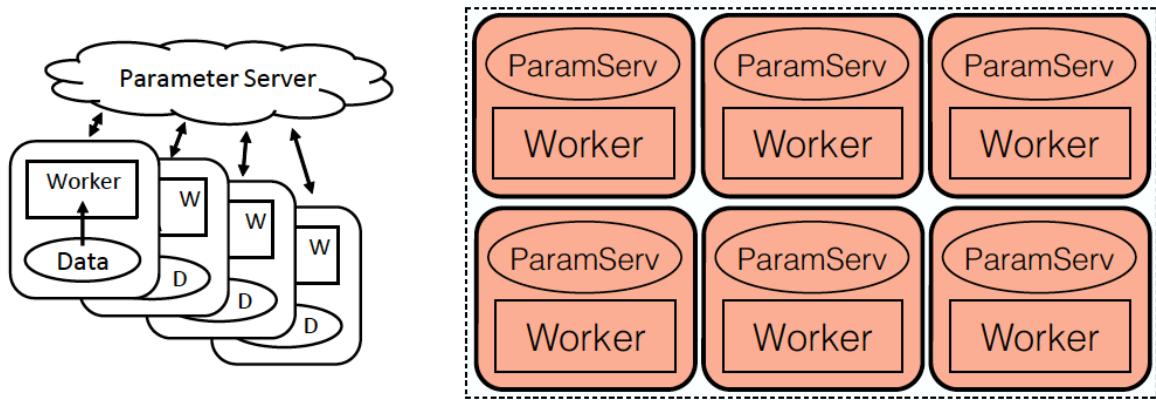


图 2.1 参数服务器系统架构。

可以向 server 请求 pull 新的参数。这种弱一致性的设计会加快训练速度，通过也可能牺牲模型的质量。如图 2.2 所示，在 Sagemaker 中，分布式训练就是通过参数服务器模型来实现的，且一般默认使用的是弱一致性的模式。同时，为了方便 push 和 pull 的操作，AWS 实现了一个 key-value 的存储系统 KVStore。

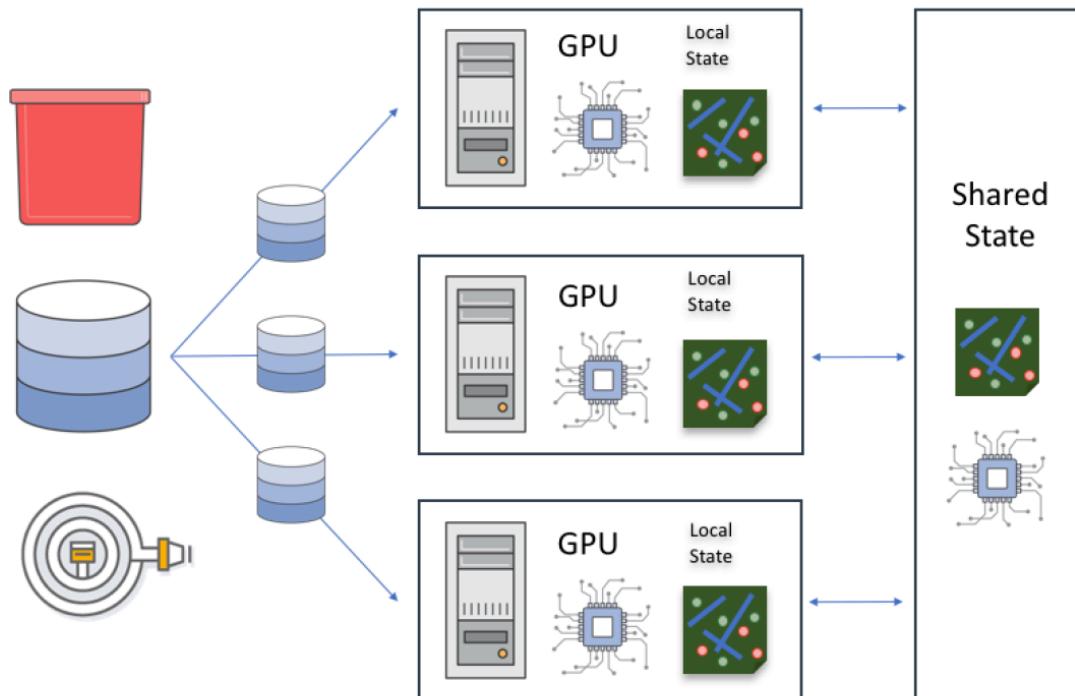


图 2.2 基于 PS Model 的 Sagemaker 系统架构。

模型抽象与参数表达。 Sagemaker 在上层为用户提供了模型抽象和参数的表示方式。对于一个模型而言，有两个与其绑定的函数需要用户来实现：1) *score* 函数，为一个模型打分以衡量模型在该数据集的训练程度，在调试或者超参数调整阶段会被用到；

2) *evaluate* 函数，接受一个 batch 的数据并计算该模型的输出。需要指明的是，对于不同的模型而言，输出的格式可能是不同的，例如对于分类问题，输出可能是一个 label 也可能是不同类别的概率。参数的表达方式同样需要统一的上层抽象。在 Sagemaker 中，不同类型的模型有着不同的表达形式。例如，K-means 算法的模型就表现为一个核心集，因此，对于任意 $k < k_{max}$ ，用户都无需重新训练模型，而是可以直接得到结果。对于 K 邻近算法，Sagemaker 也有类似的设计。

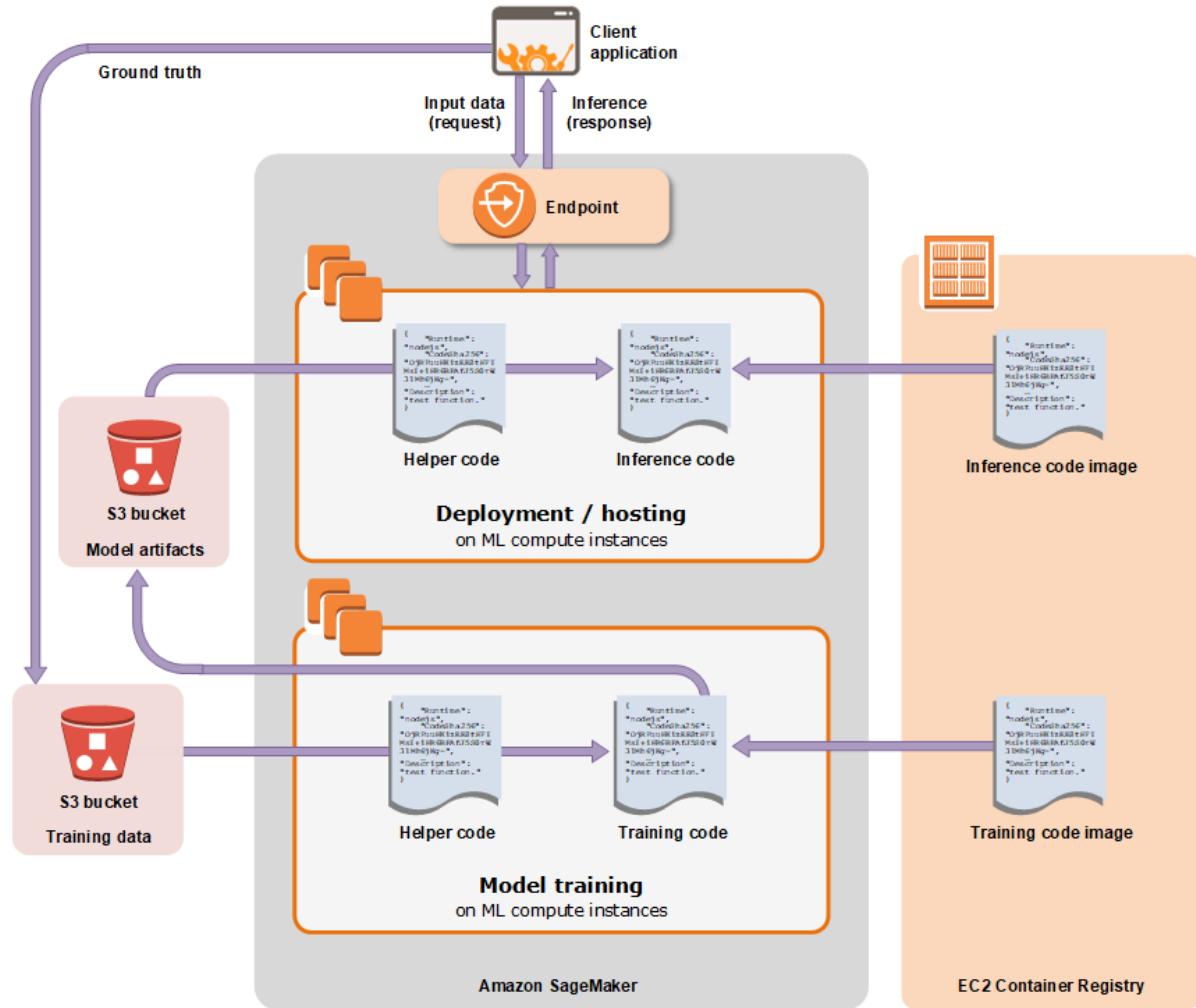


图 2.3 Sagemaker 训练、部署工作流。

模型微调与超参数调整。通常而言，一个超参数调整任务（Hyperparameter Tuning Optimization，即 HPO）由多个并行的模型训练任务构成，这些训练任务每个都尝试一个不同的超参数组合，最后得到一个最优的配置。上述过程需要耗费大量计算资源和时间，对于寻常用户而言是非常耗时耗力的。而 Sagemaker 基于 PS 的计算模型设计保证了任意一个 ML 训练任务都可以随时启动和停止。这种机制使得模型微调和超参数调整变得更为容易操作：1) 用户可以增量式地更新数据集，但却不用每次都重新训练

模型，因为整个训练过程都是流式的；2) HPO 可以同时启动多个训练任务，然后提前发现那些劣质的模型并将其终止，从而降低 HPO 的成本。

2.1.1.4 模型部署

作为一站式的机器学习系统，除了模型训练和调参，Sagemaker 还支持模型部署。图 2.3 展示了 Sagemaker 训练和部署模型的整体工作流。在部署时，首先，用户需要对其模型指定一个 endpoint，然后将部署该模型的代码打包成容器镜像，存储在 EC2 Container Registry 中。在模型实际对外提供服务时，模型文件会从对象存储 S3 中传输到容器中，进而以容器为单位对外提供服务。

2.2 基于公有云服务的第三方机器学习系统

尽管公有云厂商实现了相当完备的一站式机器学习系统，但是随着生产环境中业务场景的日渐复杂，大而全的公有云解决方案已经无法解决个性化的用户需求。例如，某些用户希望以尽可能低的成本完成大型模型的训练，或者在保证模型服务的 SLO 前提下，能够尽可能低成本的部署机器学习模型。在上述场景中，为了完成用户的具体需求，需要综合利用各类云资源，并制定一定的策略。本节对近几年来几个构建于公有云资源上层的机器学习系统做具体介绍。

2.2.1 基于云厂商动态资源的机器学习模型训练系统

机器学习模型的训练是一个不断在给定数据集上迭代更新参数的过程。对于一些较大的模型，训练时间一般会非常长。特别是深度神经网络模型再度兴起后，模型开始变得越来越庞大。例如 Open AI 的自然语言模型 GPT-3，包含 1750 亿个参数，单单存储这些参数就需要 450GB 的空间。而训练该模型的成本，据估计更是达到了 1200 万美元。对于一些个人开发者和小型公司，大模型的训练往往显得遥不可及。

除了上述可训练的参数之外，还存在大量不可训练的参数。例如，SVM 模型中核函数的类型，树模型中树的深度，深度神经网络模型中网络的宽度和深度等。这些参数需要由开发者在训练之前事先决定，一般被称为超参数。不同的参数组合在一起，就构成了巨大的搜索空间。例如，如果存在 6 个超参数，每个超参数有 4 种不同的选择，就会产生 $4^6 = 4096$ 种不同的超参数组合。

完全遍历所有的超参数组合是非常费时费力的，通常情况下也是不切实际的。研究者为了减少超参数搜索的此时，提出了诸如网格搜索、贝叶斯搜索等技术。但即便如此，仍然需要大量的算力来支撑整个搜索过程。

而与此同时，在云计算时代，云上资源的易用性、高度的弹性以及收费模式的多样性使得以较低的成本完成费时费力的模型训练和超参数搜索变为可能。如上一小节所述，一些云厂商提供了一些价格低廉的动态资源。和一般的可以获得可用性保障的资源相比，这类资源拥有相同的计算能力，但是可用性比较低，在某些场合会被云厂商主动回收。基于上述事实，可以形成一个直观的想法：在这类廉价的资源上运行机器学习模型训练和调参任务，并辅以一定的容错策略和资源选择策略，可以显著降低成本。

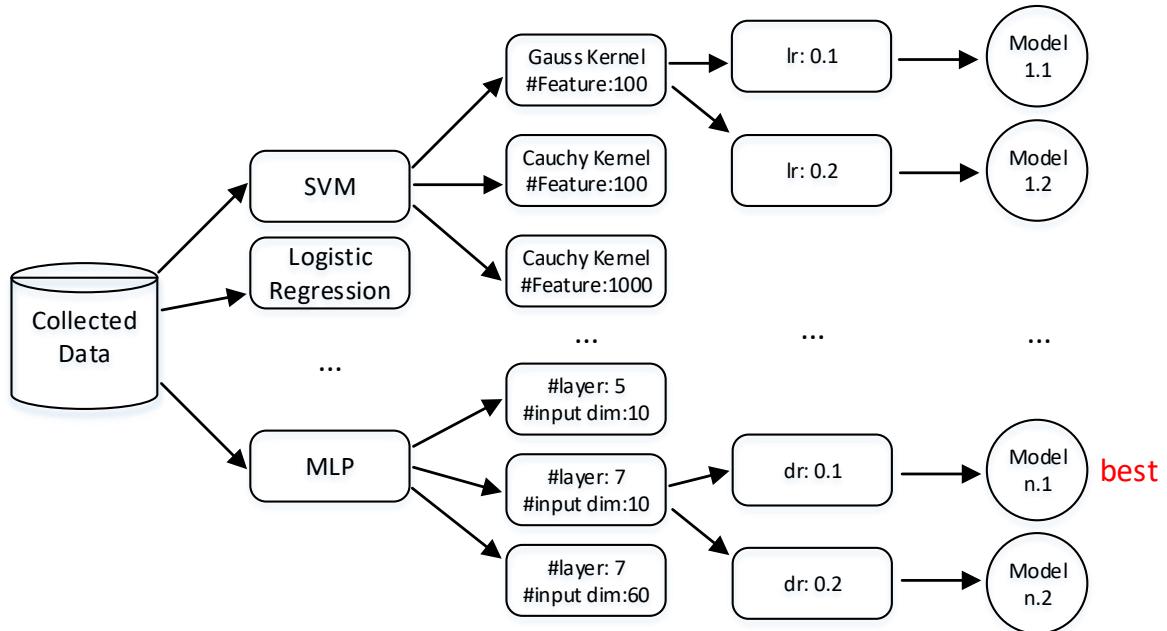


图 2.4 一个超参数调整的例子。

但是，在动态资源上运行机器学习模型训练和调参，虽然直觉上会降低成本，但是可能会产生额外的风险。因此，要做到高效并可靠地在动态资源上运行上述作业，还有如下若干问题需要解决：

1. 及时处理中断。 动态资源之所以廉价，是因为其稳定性比较低。通常情况下，在某些特殊情况下（例如集群中的负载比较高，需要动态资源来补充资源池），云厂商有权利主动回收这些动态资源。如果不处理这种回收引起的中断，机器学习模型的训练就要被动终止。因此需要采取一些手段，例如提前保存模型或者迁移任务，以及时处理中断。

2. 合理的资源选取算法。 事实上，这是在云上运行任务时需要考虑的一个通用问题。用户在云上运行作业时，一般会有一个目标，例如作业运行时间最短或者最终的总体成本最低。因此需要通过一定的算法，为用户选取能满足其需求的资源。对于机器学习模型训练与调参任务亦是如此。

3. 充分利用云的弹性。事实上，在机器学习模型训练过程中，存在大量的无效计算。如图 2.4 所示，经过大量的参数尝试，最终只剩余一个有效模型（即效果最好的模型）。因此，在参数搜索的过程中，大量的计算是无效计算。而云计算的一个重要特点就是高弹性，即用户可以因为负载增加随时获取资源，也可以因为资源闲置随时释放资源。如果效果不好的模型能够被提前发现，并释放掉其占用的计算资源，则可以避免大量无效计算，从而进一步节省成本。

4. 对计算框架进行定制。现有的流行的机器学习框架如 Tensorflow, PyTorch, MXnet 等，一般都假设底层的计算资源是可靠的，因此都没有针对上述动态资源进行适配。如果要在动态资源上运行 ML 任务，则需要对框架进行深度定制，使其具备一定的容错能力。

为了解决上述问题，使 ML 模型训练和调参都可以在云上经济高效地进行，许多研究者提出了若干解决方案。A. Harlap 等人在 2017 年提出了 ML 模型训练系统 Proteus[11]，针对 Parameter Server 架构的 ML 模型训练框架对 AWS 和 Google Cloud 的动态资源做了适配，使其能高效经济地在之上运行；Y. Li 等人在 2020 年提出了 SpotTune[17]，将 ML 调参任务部署在 AWS 动态资源上，并结合一定的资源选取算法和训练趋势预测算法，使调参任务节省大量的无效计算，最终达到了经济高效的目的。下文对较有代表性的 Proteus 和 SpotTune 做较为详细的介绍。

在介绍相关系统之前，有必要对目前云市场上常见的资源类型、本节重点研究的动态资源及其收费模式做简要介绍。

云计算作为互联网时代下的一种新型计算模式，通过虚拟化和软件定义技术将服务器、存储、网络甚至软件等各种资源聚集为共享资源池。这些虚拟资源用则进行分配，而且种类和数量均可按需配置并按实际用量计费，不用则回收到资源池中等待下次分配。相比传统的 IT 资源使用模式，云计算这种按需使用、按量付费的特点增加了用户的灵活性，但也导致了用户需求的不确定性。比如：在销售旺季，用户使用云的频率就会增加，云服务商的负载压力会持续增大甚至超过峰值，进而可能导致故障；而在销售淡季，用户使用云的频率就会大幅下降，云服务商的负荷压力也会大幅降低，导致了空闲资源的浪费。为了充分利用云资源池中的空闲资源，云服务商推出了多种的计费模式，试图以多样化的价格来吸引用户的同时，确保自己在需求高峰时可以回收部分资源。

以亚马逊公有云为例，如表 2.1 所示，EC2 按照计费模式可划分为按需实例 (On-demand Instance)、保留实例 (Reserved Instance) 和竞价实例 (Spot Instance) 这三类。相比较于按需实例和保留实例，竞价实例可提供超低折扣，但其价格会随着供需关系而不断调整。用户需要为获取该实例设置一个最高的出价，如果当前市场的价格低于用

表 2.1 亚马逊公有云 EC2 计费模式分析

计费模型 比较	计费公式	计费特点	适用场景
按需实例	实例运行时长 * 单位时间对应实例费率	单位时间费率固定，无需用户有任何使用承诺，价格较高	适合使用需求周期较短、对实例运行环境需求多变的用户
预留实例	签约金 + $\sum_{\text{第 } i \text{ 次运行时间}} \text{第 } i \text{ 次运行时间} * \text{单位时间费率}$	用户与亚马逊签订长时间使用合约，并交付签约金，在“合同”期内使用 EC2 需额外支付费用，但费率适中	适合使用需求周期长且实例对运行环境性能要求稳定的用户
竞价实例	实例运行时长 * 单位时间对应实例费率	单位时间实例费率变动，根据实例的需求情况以拍卖的方式决定，费用较低	适用于没有即时性要求、任务可中断的业务

户的出价，则用户获得实例资源进行计算。一旦市场价格高出用户的出价，则该计算资源会被亚马逊收回。对于公有云厂商来说，竞价实例通过市场机制发掘了云资源的真正价格，达到了其尽可能以最合理的价格卖出最多闲置资源的目标。而对于云平台上的虚拟云用户来说，各种无状态、或者具有容错能力的应用程序可以基于竞价实例来大幅度降低运行成本，代表负载包括：大数据、容器化工作负载、CI/CD、Web 服务器、高性能计算 (HPC) 以及其它测试和开发工作负载。尽管竞价实例的出现为非在线业务或者成本敏感业务提供了一种全新的计算资源类型，但是目前来看，无论用户的业务类型为哪种，大部分用户选择的资源依旧是为在线业务所准备的按需实例和保留实例。

竞价实例与按需实例、保留实例的合理使用，可以帮助虚拟云用户优化工作负载的成本和性能。根据 AWS 官方博客，已经有不少大型企业（如本田汽车）成功使用 AWS 的竞价实例把部分业务的计算成本下降高达 70%。然而有效的使用竞价实例除了对负载自身的要求以外（如可中断等），还需要用户针对自身的应用场景选择“正确”的实例类型。由于竞价实例在价格和回收方面的不确定性，这是比较复杂的。

Proteus 是一个基于 Parameter-Server（下文简称为 PS）架构的机器学习系统。PS 是大规模机器学习系统中常用的架构，其系统架构在 2.1.1 中已描述。如其文中描述，其使用了敏捷的弹性机制并辅以“激进的”竞价策略，使得机器学习任务在云上高效经济地完成。Proteus 由两个主要组件构成：AgileML 和 BidBrain。

其中，AgileML 是负责协调整个 PS 架构的主要组件。其通过如下的基本思想，在保证整个系统可用性的前提下，尽可能多使用动态资源（即 AWS Spot Instances）以降

低成本：1) 将 PS 的关键功能节点（即保存模型参数的 server 节点）部署在可靠资源上（如 AWS 的按需实例或者预留实例）；2) 将 PS 的 worker 节点（即负责实际计算和参数更新的节点）部署在动态资源；3) 当动态资源和可靠资源的比例不同时，AgileML 会采取不同的策略。具体地，当比例比较小时（例如 2:1），其简单地将 server 节点分布在所有的可靠资源上，而当比例比较大的时候（例如 63:1），其将可靠资源作为 BackupPSs 而将动态资源作为 ActivePSs，当发生参数更新时，更新会先到达 ActivePSs。同时，在网络带宽允许的前提下，ActivePSs 上会启动相应的后台任务将参数同步到 BackupPSs 上。

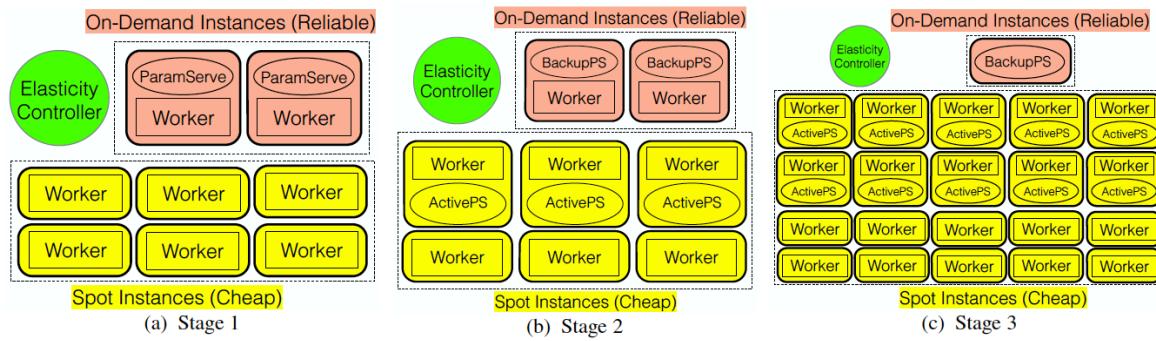


图 2.5 AgileML 架构中的三个阶段。Stage 1: 只存在部署在可靠资源上的 ParamSers。Stage 2: ActivePSs 部署在动态资源上，BackupPSs 部署在可靠资源上。Stage 3: 所有的 ActivePSs 都部署在动态资源上，可靠资源上没有 worker。

BidBrain 是 Proteus 中的资源管理模块，决定何时获取和释放动态资源。据文中所描述，BidBrain 是专为 AWS EC2 定制的，但是其经过少量的修改和定制化也可以适用于其它环境（如私有集群和其它公有云市场）。其秩序性地监控云市场中动态资源的价格，并即时地竞标那些可能给当前计算带来增益的资源（文中通过 work per dollar 来衡量增益，即单位花费可以完成的工作量）。类似地，当一种资源在达到其第一个小时的末端时，就有可能因为性价比降低而被释放掉。另外 BidBrain 还利用了 AWS EC2 的一项特殊规则：当资源在其第一个小时内被释放时，其所有花费会被退回。因此其也会主动获取那些在接下来的一小时内会被释放掉的资源，以进一步降低整个成本，此种策略被作者成为激进的竞标策略。总体而言，BidBrain 会在激进的竞标策略和保守策略之间权衡，以寻求整个系统的收益最大化。

AgileML 的具体功能和技术特点，下文通过一个示例来说明。图 2.5 描述了 AgileML 架构中的三个阶段，分别对应不同的资源类型比例。

Stage 1: 参数服务器仅位于可靠节点上。此阶段为整个 ML 训练过程的启动阶段。对于大多数 ML 的应用程序而言（例如 K-means, DNN, Logistic Regression, MF, LDA 等），其在 PS 模型的场景下 worker 节点是无状态的，其当前轮次更新的参数数据会

push 到 server 节点进行持久化存储。因此，在启动时，AgileML 会将所有的 server 节点部署到可靠资源上，而将所有的 worker 节点部署到动态资源上。在这种情形下，当 worker 因为云厂商的主动回收而宕机时，整个训练过程并不会受到影响，也不需要进行 checkpointing。但是在模式下，当动态资源与可靠资源相比比例过于悬殊时，网络会成为瓶颈。例如当存在 60 个动态节点和 4 个可靠节点时，MF 的训练速度会下降 85%

Stage 2: ActivePSs 部署在动态节点上，BackupPSs 部署在可靠节点上。为了解决 Stage 1 中的网络瓶颈问题，Stage 2 中将 server 节点分为 ActivePS 和 BackupPS 两种，其中 ActivePS 分布在动态资源上，BackupPS 分布在可靠资源上。这种策略合理控制了 server 节点和 worker 节点的比例，解决了网络的瓶颈问题。所有的参数在 ActivePS 内部是共享的，并且在后台批量地 push 到 BackupPS 上，因此当 ActivePS 因厂商主动回收而宕机时，BackupPS 可以恢复其相关状态，并启动新的动态节点。需要注意的是，在 Stage 2 中，计算节点，即 worker，仍然存在于 ActivePS 上。

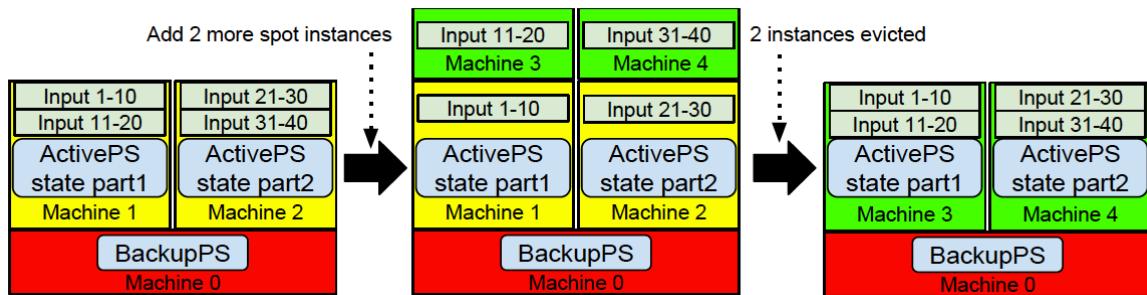


图 2.6 AgileML 动态地添加并删除资源。

Stage 3: 所有的计算节点，即 worker 都位于动态节点上。实验证明，当动态资源和可靠资源的比例超过 15:1 时，和 BackupPS 共同部署的 worker 会造成“落后者效应”(即 Straggler Effect)。因此，Stage 3 中简单地移除了所有位于 BackupPS 中的 worker。何时由 Stage 2 转移到 Stage 3，依赖于网络带宽，动态资源和可靠资源地比例以及 ML 模型的大小等等。

上述三个 stage 之间的转移一般是由 AgileML 根据动态资源和可靠资源之间的比例动态决定的。当发生阶段转移时，AgileML 会根据实际情况做出相应的动作。例如，当从 Stage 1 转移到 Stage 2 时，AgileML 会将模型的状态（一般是模型的参数）转移到新增的动态资源上，当动态资源和可靠资源的比例超过一个阈值时，AgileML 会将相应的 worker 在 BackupPS 上删除，并将相关的训练数据重新分布到其余 worker 上。图 2.6 所示的是一个 stage 转移的示例。

BidBrain 是 Proteus 中的资源管理模块，决定何时获取和释放动态资源。其持续性地监控当前和历史的动态资源价格数据，并在相关的时刻根据预测出的花费做出决策，

选择当前最优的资源加如动态资源池。简而言之，其资源选取策略较为激进，会考虑那些在未来一个小时就会被回收的 Spot Instances，以获取相应的退款，进而进一步降低使用成本。其细节策略与相关的公式推导本文不再赘述。图 2.7 描述了一个 BidBrain 决策选取何种资源加入资源池的示例。

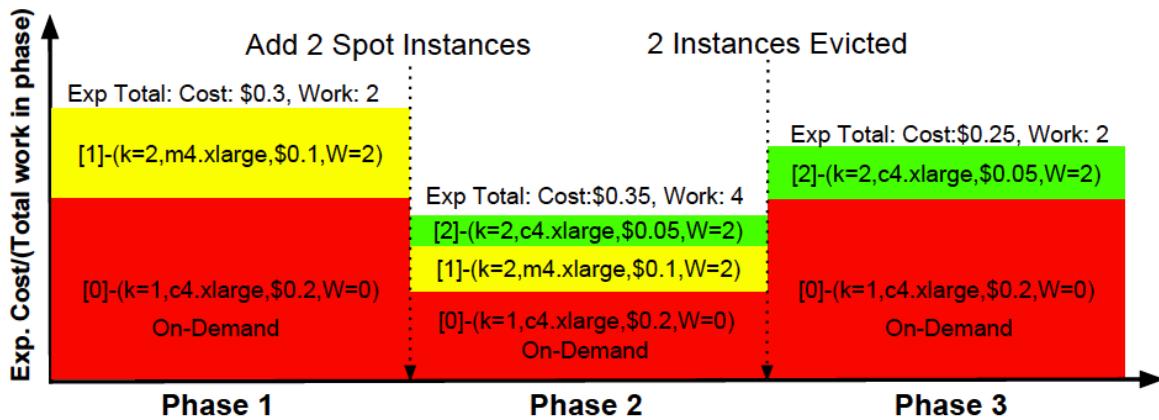


图 2.7 BidBrain 根据预期花费决定选取何种资源。

总体而言，Proteus 运用了激进的资源选取策略，将动态资源和可靠资源相结合，进行云上的 ML 模型训练。经过其实验验证，与仅使用可靠资源相比，Proteus 可以实现约 85% 的成本节省，和一般的基于 checkpoint 的云上 ML 训练系统相比，其效率约可以提升 50%。

与单个模型的训练相比，模型的超参数调整（Hyper-parameter Tuning，下文简称为 HPT）在整个 ML 模型产出周期中位于更上一层。其一般需要开发者尝试模型的多个超参数组合，以得出最优的一个。SpotTune 是一个云上自动调参管理系统，通过运用公有云上廉价的动态资源来协调整个 HPT 过程。据文中所述，SpotTune 同样是基于 AWS 的公有云服务实现的，但是经过一定程度的修改和定制，其可以应用与其它云厂商。为了妥善处理因厂商主动回收产生的中断，SpotTune 根据厂商提前发出的通知事先对模型进行 checkpoint 并将其保存到持久化对象存储中。同时其运用了两个关键技术来保证 HPT 的经济高效：

- 1. 基于花费的细粒度资源管理。** SpotTune 计算花费的粒度非常细。其通过预测动态资源的回收概率和在线对 HPT 任务进行性能建模，预测出接下来一小时内“单步成本”最低的资源并进行任务部署。

- 2. 基于训练趋势预测的早停机制。** SpotTune 将 ML 模型的训练曲线建模为一个可以预测的、多阶段的函数。简而言之，其可以通过当前时刻收集到的关于模型的质量信息（通常是模型的 score 函数所得出的结果，例如分类模型的分类准确率，回归模型的平均平方误差等等）来预测该模型在最终收敛时的质量，以提前结束那些劣质模型，

节省计算资源。

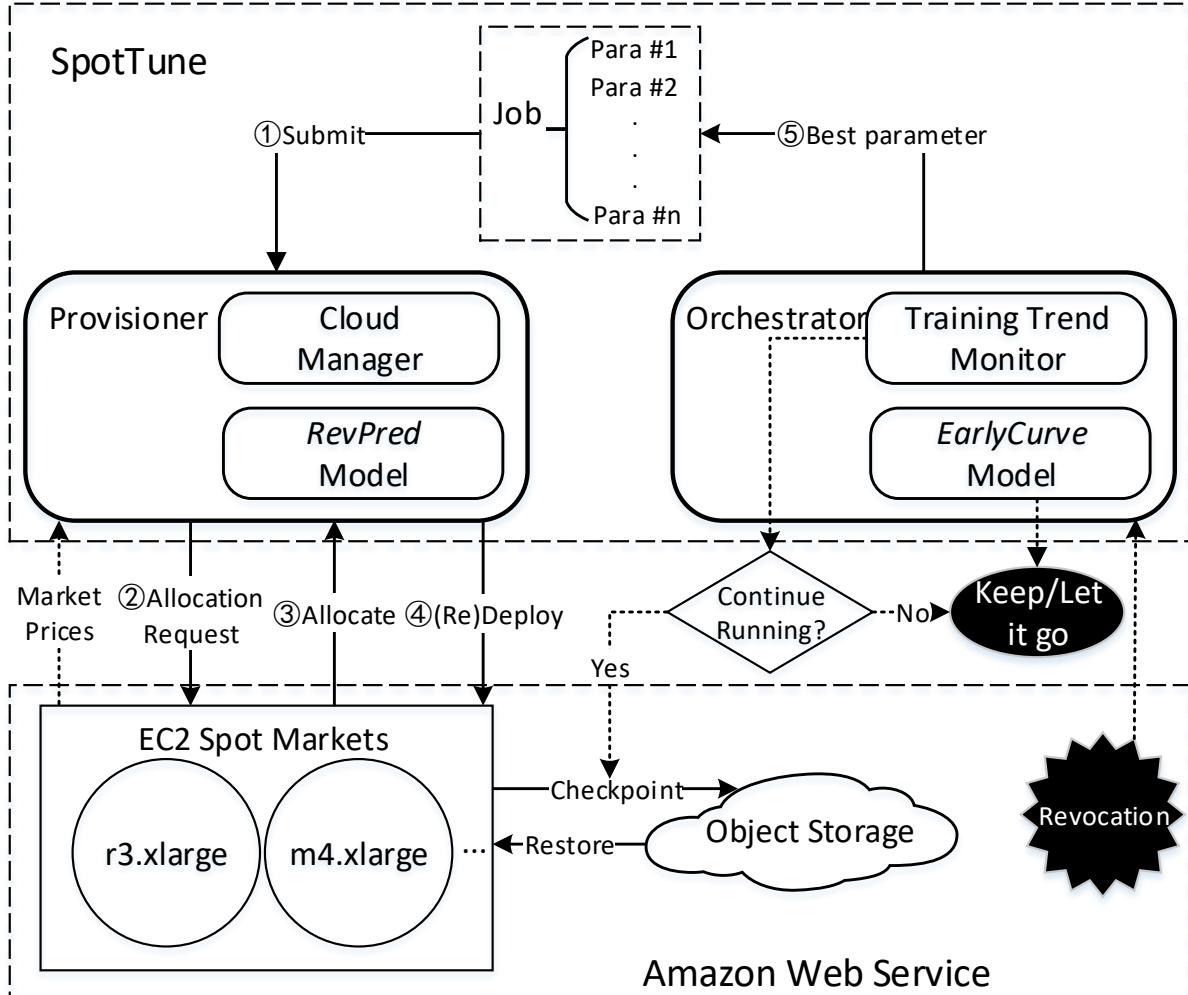


图 2.8 SpotTune 系统架构。

SpotTune 的系统架构如图 2.8 所示。其主要功能模块包含 Provisioner 和 Orchestrator 两部分。其中 **Provisioner** 负责与 AWS 互动以请求和获取资源。通过结合历史数据和在线更新的性能数据，Provisioner 可以细粒度地预测每种资源带来的收益，进而选择最适合 HPT 任务的资源。**Orchestrator** 负责监控每个超参数组合的训练进度。通过当前收集到的模型质量数据，其可以预测模型的最终质量。Orchestrator 一直监听 AWS 市场的消息，当收到回收提醒时，其会立即将模型 checkpoint 到持久化存储系统 AWS S3 中。另一方面，SpotTune 认为运行超过一小时的实例性价比是比较低的，在没有特殊情况出现时，Orchestrator 会主动将其停止。

SpotTune 之所以能做到经济高效，是由 Provisioner 和 Orchestrator 中的 RevPred 模型和 EarlyCurve 模型决定的。其中 RevPred 通利用历史数据预测每一种动态资源在将来一小时内被回收的概率，进而用该概率结合在线更新的性能数据预测每一种资源在

将来一段时间的效率，其量纲为“\$每步”。具体计算过程如下：

1. 计算每种资源在接下来一小时内单价的数学期望。公式 2.1 给出了计算方法。公式中的参数 p 为该示例在一小时内被回收的概率。如果该资源被回收（概率对应为 p ），则价格为零，因为 AWS 会将该部分费用全数退回给用户。如果该资源没有被回收（对应概率为 $1 - p$ ），则按照当前单价收费。

$$\begin{aligned}\mathbb{E}[eCost] &= [(1 - p) \cdot \overline{\text{price}} + p \cdot 0] \cdot 1 \text{ hour} \\ &= (1 - p) \cdot \overline{\text{price}} \cdot 1 \text{ hour}\end{aligned}\quad (2.1)$$

2. 计算每种资源在接下来一小时内每运行一步（或一个迭代周期，在 ML 作业中通常体现为一个 batch）的花费。公式 2.2 给出了计算方法。其中 $M[\text{inst}][\text{hp}]$ 代表了 HPT 任务 hp 在示例类型 inst 上的运行速度（每步的耗时，seconds / step），由 SpotTune 持续在线监控模型在每种实例上的训练情况得出。

$$\begin{aligned}\mathbb{E}[sCost] &= \mathbb{E}\left[\frac{M[\text{inst}][\text{hp}] \cdot eCost}{1 \text{ hour}}\right] \\ &= \frac{M[\text{inst}][\text{hp}] \cdot \mathbb{E}[eCost]}{1 \text{ hour}} \\ &= \frac{M[\text{inst}][\text{hp}] \cdot (1 - p) \cdot \overline{\text{price}} \cdot 1 \text{ hour}}{1 \text{ hour}} \\ &= M[\text{inst}][\text{hp}] \cdot (1 - p) \cdot \overline{\text{price}}\end{aligned}\quad (2.2)$$

至于实例回收概率 p ，SpotTune 通过一个精心调试的 LSTM 神经网络模型来预测，此处不再赘述。

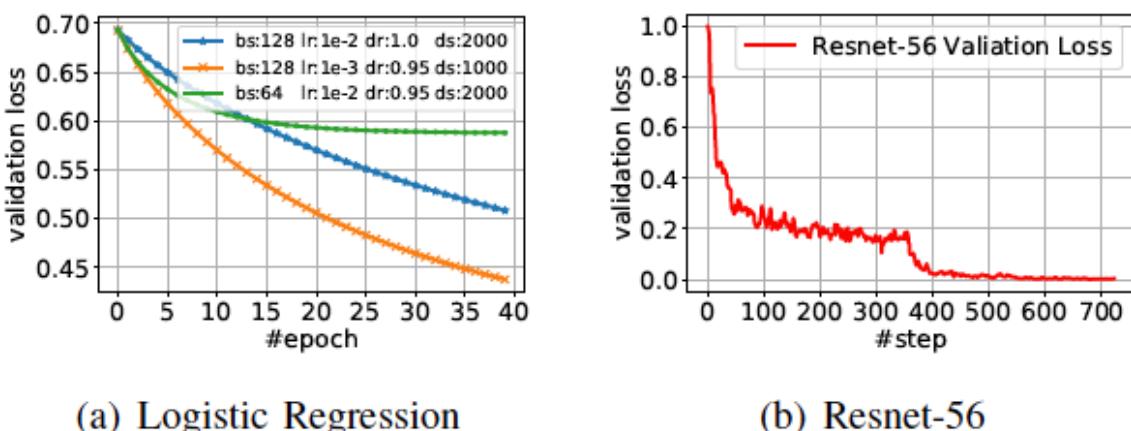


图 2.9 Logistic Regression 与 Resnet-56 的训练曲线。

另一方面，SpotTune 认为每个 ML 训练任务的曲线都可以使用一个分段函数进行建模，并利用其部分数据来拟合整个曲线实际的形状，进而预测曲线的最终走势。图 2.9 展

示了两种 ML 任务的训练曲线，都呈现出明显的反比例特征和阶段性特征（通常而言，阶段性的特征一般是由学习率的突变导致的）。且第一张图显示可以通过部分的曲线形状来预测最终的曲线走势，进而提前淘汰一部分模型。EarlyCurve 将每个 ML 训练任务按照公式 2.3 进行建模，通过当前收集到的部分数据将其拟合为一个分段函数。

$$\hat{\mathcal{L}}_k = \sum_{i=1}^{ST} \left(\frac{1}{\alpha_{i0} \cdot k^2 + \alpha_{i1} \cdot k + \alpha_{i2}} + \alpha_{i3} \right) \cdot sign(k, l_i, r_i) \quad (2.3)$$

公式 2.3 工作的前提是用户先指定一个早停系数 $\theta \in [0, 1]$ ，当训练完成 θ 时，EarlyCurve 开始预测模型的最终走势，并在此时淘汰一部分模型。这个参数反映了用户对于调参任务的激进程度，小的 θ 可能会使预测准确率降低，但是能更快地结束 HPT，反之亦然。具体 θ 的选择，依赖具体场景和用户的具体需求。

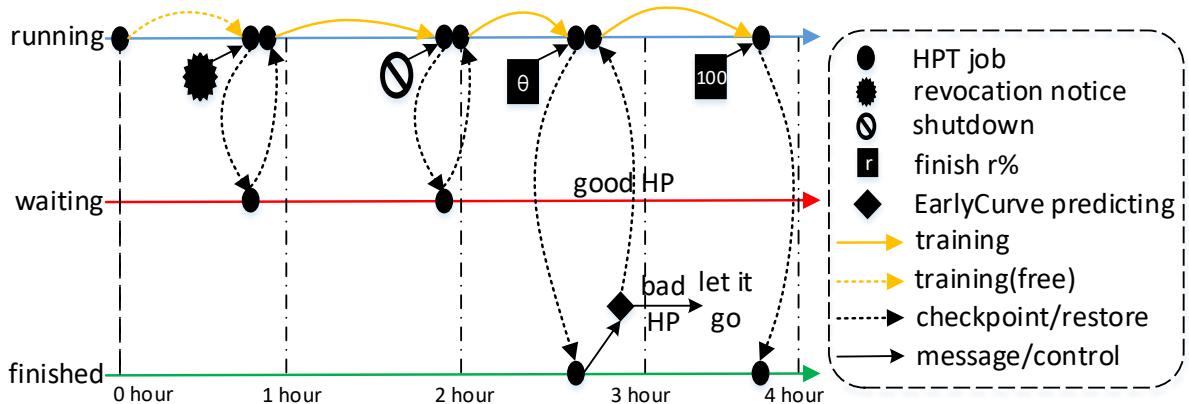


图 2.10 SpotTune 工作流程示例。

图 2.10 给出了 SpotTune 的一个示例工作流程。在该流程中，一个 HPT 任务被启动的第一小时就被回收，因此第一小时的算力对于 SpotTune 而言是免费的。后面该模型再次被调度之后运行超过了一小时，因此被 SpotTune 主动停止，并再次被调度到别的实例上。最终，当任务完成度为 θ 时，SpotTune 判断其是否是一个优秀的模型，如果是的话将继续训练直到结束，否则其占用的计算资源将被直接释放掉。

SpotTune 通过利用云上的动态资源，辅以一定的资源选取策略和 ML 训练过程预测算法，使得 HPT 可以在云上经济高效的运行。与 Proteus 类似，该类系统常常将成本放在系统设计的首位。云计算以其 pay-as-you-go 的使用模型俘获了大量用户，然而大到企业用户，小到个人用户，往往都会将成本放在首位，因为在云上获得的任何算力都会直接或间接地转化为经济开销，对于机器学习这种耗费算力的场景更是如此。正因为这样场景的普遍性存在，类似 Proteus 和 SpotTune 的基于云商服务进行二次开发的第三方系统才层出不穷。

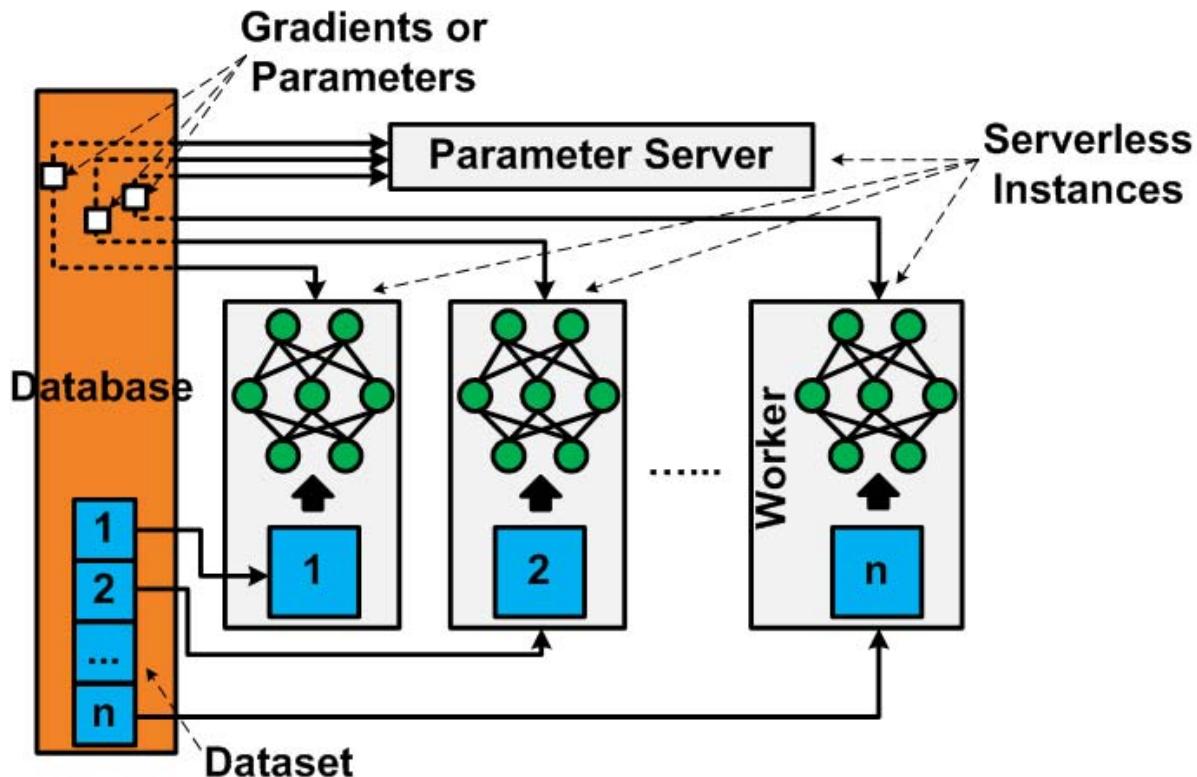


图 2.11 基于 PS 架构的 serverless ML 系统。

2.2.2 基于 FaaS 的模型训练系统

除了上述基于 IaaS 服务的 ML 系统外，近年来基于新型计算模式 FaaS 的 ML 系统也逐渐进入学术圈的视野。FaaS，亦即 serverless computing，是一种新型的计算模式。其关键词 serverless 是一个用户维度的特征词，并非真的没有服务器，而是在用户的角度感知不到后端服务器的存在。用户只需要上传其业务代码，并声明相应的 hook API，即可在云端部署一个事件驱动的 function，这也是 FaaS，即 function as a service 这一代称的由来。用户的函数一般按照请求次数收费，并且根据请求负载的变化，云厂商会进行弹性伸缩。这种细粒度的收费模式和弹性的使用模式使得应用上云变得更为简单，也自然有学者想到将其与 ML 联系在一起。

然而，serverless computing 实际上并不是天然适用于 ML 系统的，要实现 FaaS 风格的 ML 系统，势必要在当前的 serverless computing 框架之上进行深度修改。总体而言，若要应用于 ML 的工作负载，serverless computing 目前还存在如下问题：

1. 本地内存和磁盘空间小。通常而言，为了更快的横向拓展，轻量是函数计算设计的第一要义。因此，目前流行的 FaaS 架构，所支持的内存大小和磁盘大小都有限。例如 AWS Lambda 仅支持最多 3GB 的本地内存和 512MB 的本地磁盘。而现有的 ML 系统一般会在本地保留一份完整的数据集，并将数据集全部 load 到内存中以加快训练

过程，这与 serverless 的设计原则是相违背的。当下的流行的 Tensorflow, Spark 等框架便无法不加修改地在 AWS Lambda 内运行。

2. 带宽有限，无法进行 p2p 通信。与 VM 相比，serverless 函数一般网络带宽非常有限。例如，最大的 AWS Lambda 函数仅支持上线为 60MBps 的带宽，这与数据中心中的千兆网络，万兆网络甚至 Infini-band 相比是非常低的。另外，serverless computing 中的函数一般是无状态的，且不同的函数之间一般无法直接通信（一个函数可以在 service 维度调用另一个函数的 API，但是同一个 service 的不同函数实例一般无法感知其它函数实例的存在）。因此，在本地数据中心中常用的通信策略，例如树形或者环形的 all-reduce 通信在 serverless 环境中是无法实现的。

3. 生命周期短且启动时间不稳定。Serverless 函数一般是为在线的 service 设计的，因此其生命周期天然较短。同时其启动时间不稳定，例如 AWS Lambda 的启动时间甚至最长可以达到一分钟。因此，基于 serverless computing 的 ML 系统必须能够很好的处理频繁的函数启停。

4. 缺少快速的共享存储支持。为了解决函数之间无法通信的问题，共享存储是一种常用的手段。但目前云环境中支持 serverless 使用的共享存储仅限于对象存储、数据库等 SaaS 服务，其响应延时、吞吐量等仍然无法与数据中心环境中的共享内存相比。

L. Feng 等人 [9] 在 2018 年提出了一种简单的基于 serverless 的 ML 训练框架，其系统仍然采用 PS 作为主要框架。为了解决不同函数之间无法直接通信的问题，其采用云数据库作为中间存储层，使得不同的函数实例通过共享存储的方式进行通信。这一思想类似操作系统中进程间通信 (IPC) 的实现方式之一，即共享内存。图 2.11 描绘了该系统的架构图。其 PS 架构中的 server 和 worker 的实体都是函数。由于云数据库本身没有对参数进行聚集、更新、分发的能力，其 server 函数实际上负责参数的 push 和 pull 操作，数据的实际存储位置还是在数据库中。

基于上述基本架构，H. Wang 等人 [27] 在 2019 年提出了基于 serverless computing 的 ML 系统 Siren，使用强化学习算法来辅助决策整个系统中的函数个数和分布。其基本假设在于，不同的函数数量会导致不同的训练完成事件和最终花费，且其模式难以通过简单的规律来总结。而且对于不同的 ML 模型，规律也不尽相同。图 2.12 以在一个具有 8 核心 CPU 的 EC2 虚拟机实例上训练模型耗费的时间和产生的花费作为 baseline 对比了不同的函数数量下花费和耗时的不同。

图 2.13 描述了 Siren 的系统架构和工作流。用户基于 Siren 提供的 API Lib 提供其模型代码包和数据集。上传至公有云之后，Siren 对任务进行初始化并开始运行。运行过程中每个 epoch 产生的数据将反馈给 Function Manager，然后 Function Manager 将上述数据反馈给强化学习调度器。调度器根据上述数据计算当前的 reward 并输出下一次

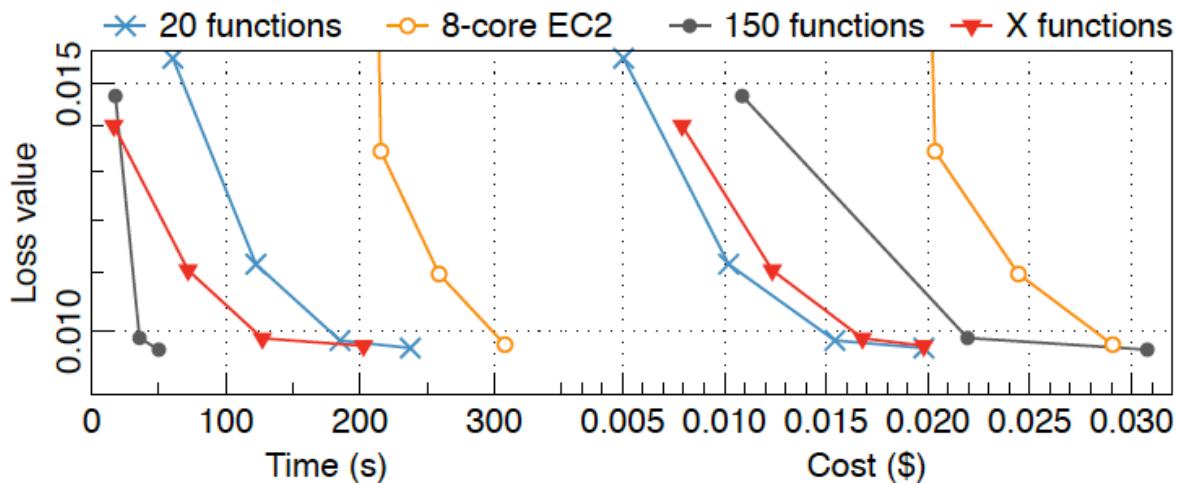


图 2.12 在 AWS Lambda 和 EC2 上训练一个逻辑回归模型的耗时和花费。每个点代表一个 epoch。

决策的 action，并反馈给 scheduler，最终重新配置函数资源，开始下一个 epoch 的训练。与 L. Fang 等人的工作类似，Siren 也采用了基于共享存储的 PS 架构。

事实上，serverless computing 对 ML 的限制不仅仅在于网络通信，函数内部同样存在限制。例如，AWS Lambda 中，代码包的大小被限制到 250MB，运行时长被限制到 300 秒，运行内存大小限制到 3GB。Siren 采取了如下措施规避上述限制：1) 重新编译 MXNet，去掉 USE_CUDA, USE_CUDNN, USE_OPENMP 等并行选项，因为 serverless 函数可以成千上万的并发，因此 Siren 追求的是函数级别的并发，而非函数内部进程级别的并发；2) 每一个 epoch 由调度器重新分配一次函数，这与强化学习的 reward-action 机制也相符合。

Siren 的经济高效是由其强化学习调度器所决定的。该调度器接受每个 epoch 结束后的函数状态作为输入设计 reward 函数，并进行下一次调度。图 2.14 展示了 Siren 的强化学习调度器的决策流程。关于其强化学习算法的细节设计此处不再赘述，亦非本报告所关注的重点。

整个 ML 模型产出的周期中不仅包含模型训练，还包括数据预处理、超参数调整等。在 serverless computing 的环境下支持一站式的 ML 工作流，如下两个问题可能会得到解决：

1. 资源超配。在 ML 的整个工作流中，资源利用率实际上变化是十分剧烈的。在以 VM 为计算设施的环境中，开发者为了能够性能，通常会按照整个工作流中的资源利用率峰值来配置和获取资源，这无疑造成了严重的资源浪费。而 serverless computing 拥有优秀的弹性，能够根据工作负载的实际强度为用户配置合适的资源，从而可能解决这类资源超配问题。

2. 透明的资源管理。一般而言，专业的机器学习从业人员并不具备熟练的系统配

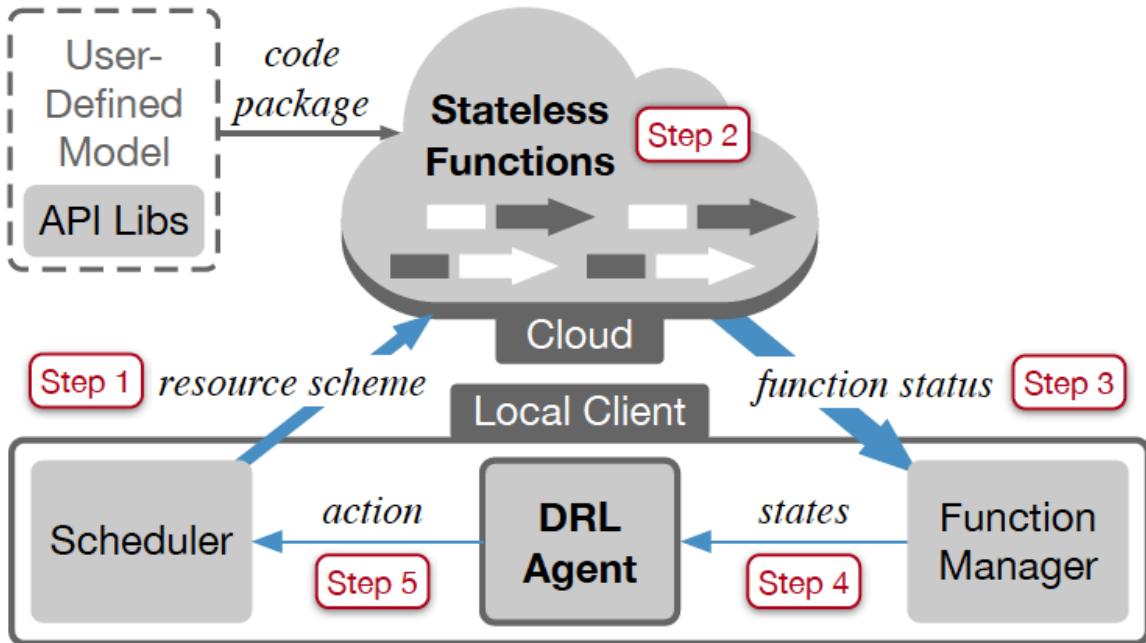


图 2.13 Siren 系统架构和工作流。

置与优化技能。因此，现有的基于 VM 的环境配置方案，特别是在分布式场景下，为开发人员施加了很大的压力。因此亟需一直透明的资源管理方法，对于 ML 从业人员而言，其只需关注模型相关的算法代码，其余应该交由系统完成，这与 serverless computing 产生的初衷是类似的，serverless 的计算模型有解决此类问题的天然土壤。

J. Carreira 等人 [4] 在 2020 年提出了 Cirrus，一个基于 serverless 的支持一站式 ML 工作流的框架。其支持了数据预处理、模型训练、模型调优等 ML 工作流中的关键阶段，并为用户提供了友好的 FaaS 风格的 API。与前两个工作类似，Cirrus 也采取 PS 模型作为整个系统的基础框架。图 2.15 描述了 Cirrus 的系统架构。所不同的是，Cirrus 实现了自己的数据存储系统 Data Store。

Cirrus 中的 Data Store 是在云 VM 上实现的，其支持 key-value 的存储接口，例如 set/get，和常用的 PS 架构接口，例如 push/pull 参数。其设计时通过如下几个优化，使得该存储系统实现了高并发、低延迟、高吞吐：

- 1. 多线程服务器设计。**为了达到高吞吐量，Cirrus 的 Data Store 充分利用的多线程的优势，将负载分布在不同的 CPU 核上。实验证明，这样的设计增加了 30% 的吞吐量。
- 2. 数据压缩。**为了减少网络传输的数据量，Cirrus 的 Data Store 在 get 或者 set 时，都会将需要传输的数据进行预先压缩，这样以来所传输的数据就会大幅减少。实验证明，这样的设计将数据传输量减少了 50%。
- 3. 传输稀疏数据。**Cirrus 在传输参数和模型数据时，只传送那些修改了的数据条

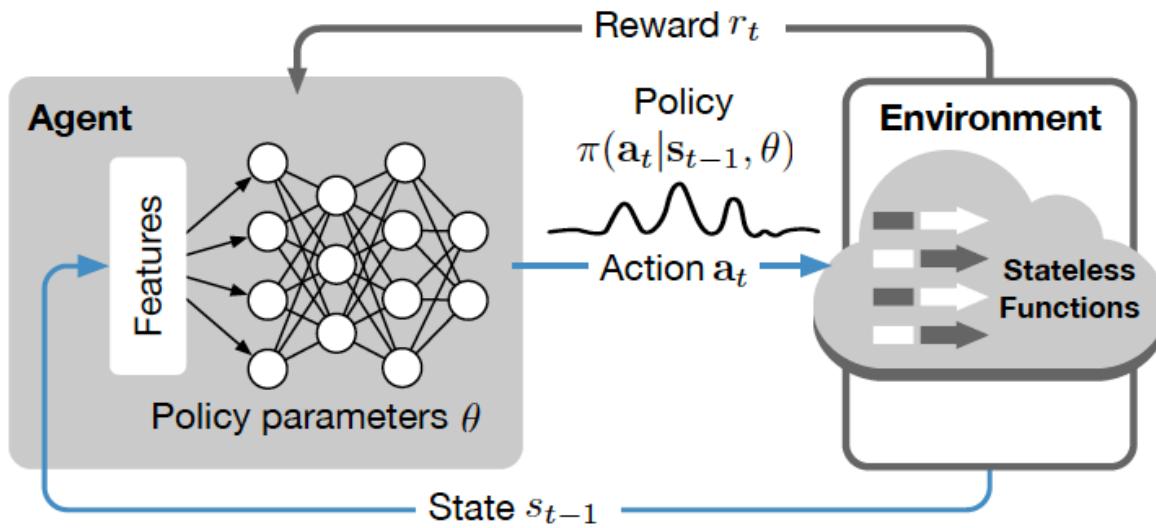


图 2.14 Siren 中强化学习调度器的工作流程。

目，也就以稀疏数据的形式进行传输。实验证明，这样的设计将数据传输量进一步减少了 90%。

Cirrus 屏蔽了底层设计的复杂性，向上对用户提供简洁的 python 接口，图 2.16 展示了数据预处理、模型训练和超参数调整阶段 Cirrus 需要用户编写的代码示例。

以上述三篇工作为代表，从 18-19 年的架构和调度算法的优化，到 20 年开始研究实现 serverless 专用的共享存储系统，可以看出，学术界正在一步一步深入推动 serverless computing 在 ML 系统中的应用。FaaS 风格的 ML 系统也确实解决了配置难、超配造成资源浪费等常见的问题。

2.2.3 基于公有云服务的模型部署系统

在 ML 的工作流中，训练和调参一旦结束，就意味着产出了可以对外提供服务的模型。接下来这些模型就需要被部署到云端，为用户或者其它开发者提供服务。这类服务在日常生活和科研中是非常常见的。例如，百度提供的看图识花服务，用户只需上传一张图片，便可以得到这是何种花，甚至模型在某些情况下还会给出属于不同种类花的概率。如何将这些模型在线上部署，并且快速对用户的请求做出响应，便是模型部署系统需要解决的问题。

公有云中存在着多种多样的资源可以用来部署模型，例如 IaaS 和 FaaS，都提供了部署模型的能力。其中 IaaS 中的 VM 还可以按照收费方式分为按需实例、预留实例和抢占实例等种类。和模型训练/调参类似，模型部署也同样需要考虑性能和成本两个角度。首先模型的在线服务也属于一种 service，所以其在大多数情况下都存在对该 service 的质量要求，即服务等级目标（Service Level Object, SLO）。例如，一个 SLO

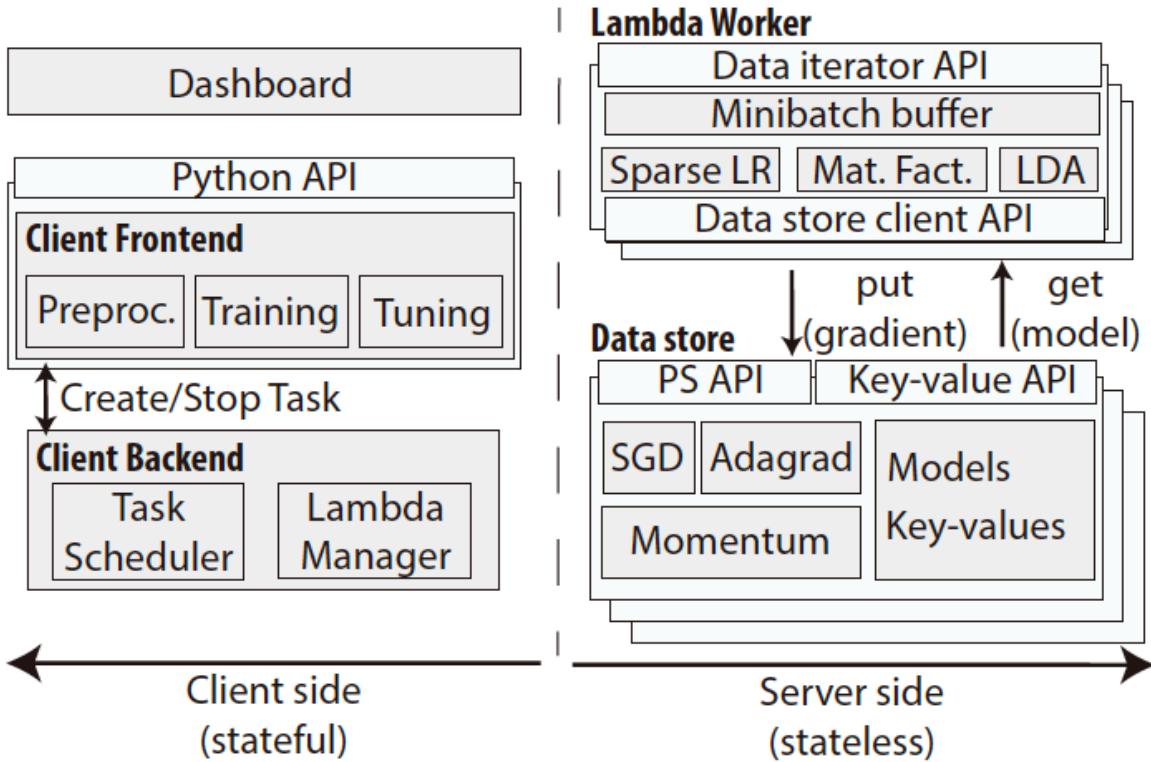


图 2.15 Cirrus 系统架构。

```

import cirrus
import numpy as np

local_path = "local_criteo"
s3_input = "criteo_dataset"
s3_output = "criteo_norm"

cirrus.load_libsvm(local_path, s3_input)
params = {
    'n_workers': 5,
    'n_ps': 1,
    'worker_size': 1024,
    'dataset': s3_output,
    'epsilon': 0.0001,
    'timeout': 20 * 60,
    'model_size': 2**19,
}
# learning rates
lrates = np.arange(0.1, 10, 0.1)
minibatch_size = [100, 1000]

gs = cirrus.GridSearch(
    task=cirrus.LogisticRegression,
    param_base=params,
    hyper_vars=[["learning_rate", "minibach_size"]],
    hyper_params=[lrates, minibatch_size])
results = gs.run()

(a) Pre-process                               (b) Train                                (c) Tune

```

图 2.16 Cirrus 用户接口 API 示例。

的典型例子便是要求至少 95% 的请求需要在 100ms 内完成响应。因此，在云环境中对于模型部署的研究一般都集中在一个问题上，即如何在保证模型服务的 SLO 的前提下，尽量降低成本。

A. Harlap 等人在 2018 年提出了服务部署系统 Tributary[10]。与其前作 Proteus 类似，Tributary 也同样充分利用了云市场中的动态资源，例如 AWS 的 Spot Instances。图 2.17 展示了 Tributary 的系统结构图。和 Proteus 类似，其采用了基于 LSTM 的价格预测模型，预测接下来一段时间内资源池中的所有实例在运行该服务时的潜在花费，并在其中选取最优实例。和 Proteus 相比，实际上是一种更为简单的场景，因为 service 本质上是一种无状态的负载，在动态实例启停时，无需考虑对先前状态的保存和恢复。

Tributary 虽然利用云上的资源开发的模型部署系统，但仍然没有逃离传统的基于

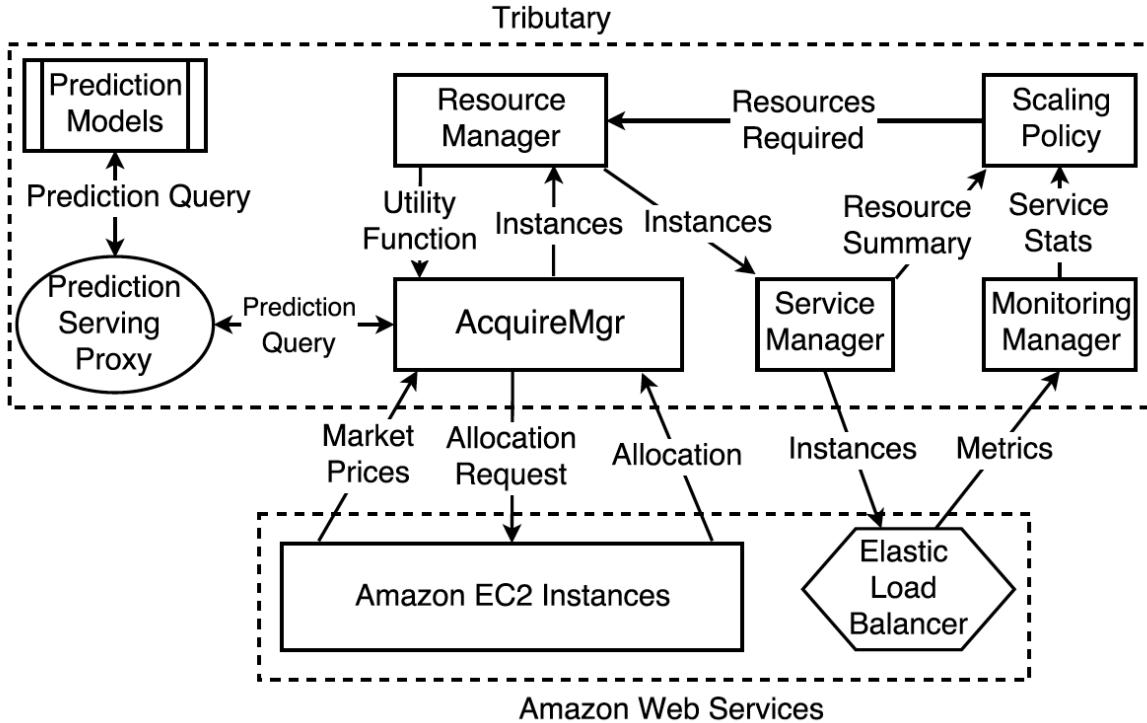


图 2.17 Tributary 系统架构图。

IaaS 资源部署服务的模式。C. Zhang 等人在 2019 年提出了 MArk[31]，一种基于多种云资源混用的模型部署系统。其充分研究了公有云上不同类型资源的特点和差异性，包括常规的 VM，容器服务，动态 VM，突发型 VM，以及 GPU 型 VM，以便指定相应的算法，在合适的场合使用合适的资源。这里对突发型的 VM 实例，即 Burstable Instances 做简单介绍。该种类型的实例 CPU 能力在平时维持在一个性能较低的水平，但是用户可以令其在特定时刻以较高的性能维持一段时间。一般而言，该种类型实例的价格比按需型实例要低很多。

其经过一些列的实验，C. Zhang 等人得出如下四条结论：1. 基于 VM 进行模型部署可以达到最优的花费和最优的性能，但是难以处理突发的增量请求。如果将其与 FaaS 服务相结合，或许可以解决上述问题。2. 突发型的实例可能可以应对突发的请求。3. 在按需型实例中，比较小的实例性价比更高，虽然和大型的实例相比其响应延时也要更高。4. 只有将请求进行适当的打包批量处理，才可能发挥 GPU 型实例的优势，使其响应延时和花费优于 CPU 型的实例。

基于上述四条观察，作者提出了 MArk。图 2.18 描述了 MArk 的系统架构。其后端将多种云资源视作潜在的资源池，根据负载的动态变化和 SLO 的具体要求来进行资源选择和配置。MArk 的资源配置策略可以概括为如下 3 条，其细节本文不再详述。

1. 确定最佳的 Batching 参数。在 GPU 实例上进行 Batching 可以达到较好的服务

质量和较优的花费，但是 Batching 的两个重要参数，batching 队列长度和 batching 等待时间，却需要精心调整。作者根据公式 2.4，辅以线下性能建模的方式，来确定最佳的参数选择。

$$\begin{aligned} W_{batch} + T_b &\leq RT_{max}, \\ W_{batch} + T_b &\leq \frac{b}{\mu_{nb}^*} \end{aligned} \quad (2.4)$$

2. 使用 Lambda 和 Burstable Instances 应付突发的请求。虽然使用 VM 来进行模型部署能达到最优花费和性能，但是 VM 的启动时间过长，在请求数量突然增多时，难以快速横向拓展来进行应对。Lambda 具有较短的启动时间和较强的横向拓展能力，可以用来处理突发请求，当扩容的 VM 启动之后，再将流量导向新增的 VM 实例。同时 MArk 还预先启动了若干单价便宜的 Burstable Instances，当请求增多时，也可以将流量暂时导向这些实例，以利用其短暂的 CPU 峰值性能。具体在上述两种方案之间如何选择，就依赖于 MArk 根据具体的负载曲线和这两种方案产生的花费多少来进行权衡。

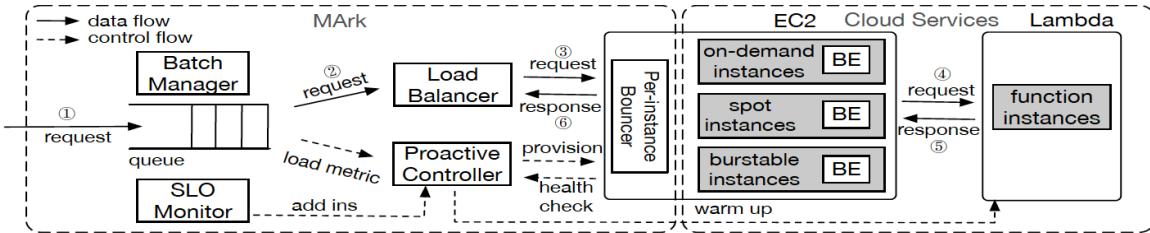


图 2.18 MArk 系统架构图。

2.2.4 基于 FaaS 的模型部署系统

FaaS 最常用的使用场景就是服务的云端部署，从这个角度来讲，不考虑 serverless computing 本身的若干缺点，纯 FaaS 的模型部署方式几乎是最理想的。尤其是从 ML 用户的角度而言，一旦模型完成训练和调参，用户只需将模型文件上传到云端，并声明相应的 API，即可完成模型的部署，至于弹性伸缩则交由后端的云商来做即可。

V. Ishakian 等人 [12] 在 2018 年在 AWS Lambda 上基于 MXNet 测试了若干模型对外提供服务的能力。其结论可以用一句话来概括：在热启动的前提下，利用 serverless 服务进行模型的在线部署是可行的，无论性能还是花费都可以接受，但是第一次请求会因为冷启动造成非常明显的高延时。Ishakian 等人测试的主要是一些图像分类相关的模型，而 Z. Tu 等人 [25] 在 2018 年针对自然语言处理的模型在 AWS Lambda 进行的相关实验。图 2.19 展示了在 AWS 的云环境中部署 NLP 模型的示例。如图所示，除了 Lambda，该系统还用到了 AWS 的云数据库服务 DynamoDB 来存储词向量。模型部署之前用户

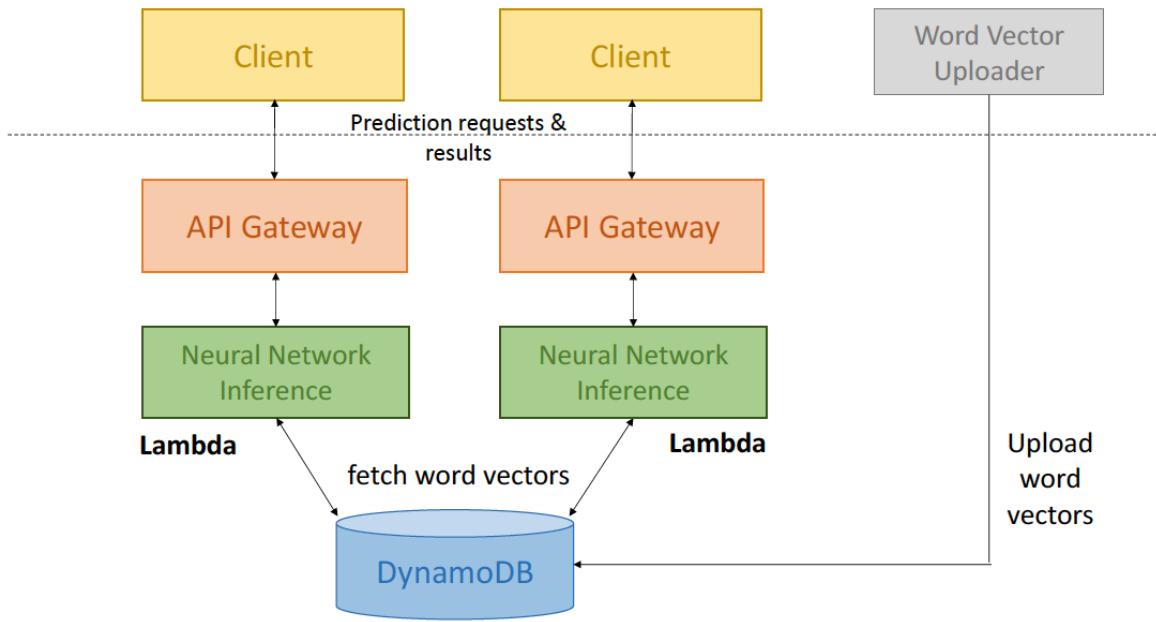


图 2.19 NLP 模型在 AWS 云环境的部署。

先将训练得到的词向量上传至云数据库中，在实际对外提供服务时，Lambda 会按需从 DynamoDB 中获取相关的词向量。然而在这种系统设计中仍然存在两个问题。首先，冷启动的问题仍然没有解决。其次，由于 Lambda 没有本地的词向量缓存，在请求频率比较高的场景下，云数据库的延时和吞吐率难以满足服务的 SLO。

因此，现有的公有云上以 AWS Lambda 为代表的 FaaS 服务难以直接应用于模型部署。然而 FaaS 只是一种云资源的使用模式，并非于某个云厂商绑定，学术界也开始更多的探索更为先进的 FaaS 架构。F. Romero[24] 等人在 2019 年提出 INFaaS，一种 model-less 的模型部署系统。该模型部署系统假设用户对一个服务有两个维度的需求，即准确率和延迟。例如，某个图像识别的用户可能要求模型能够以不低于 95% 的识别准确率在 200ms 内输出识别结果。对于不同的用户和不同的场景，这样的 SLO 是不尽相同的。F. Romero 等人注意到，同一任务的不同模型变体在模型效果和延迟上存在很强的多样性，可能可以用来满足不同场景下的需求。例如，对于图像分类任务，VGG16 和 Resnet50 都是常用的模型，但是两个模型在准确率和延迟方面各不相同，VGG16 准确率稍低，但同时延迟也比较低，Resnet50 则与相反。图 2.20 就反映了模型的多样性。

为了满足不同场景下用户的不同需求，F. Romero 等人提出了 INFaaS，根据用户的 SLO 要求在不同的模型变体之间进行选择并对用户的请求做出响应。如图 2.21 所示，INFaaS 的工作流程如下：1) 用户对前端发出请求，而后前端的 Dispathcer 根据用户的 SLO 需求选取当前最合适的模型，并将其反馈给 worker；2) worker 将模型从 model repository 中取出来，并派发到相应的执行器上（一般而言是 CPU 或者 GPU）进行执

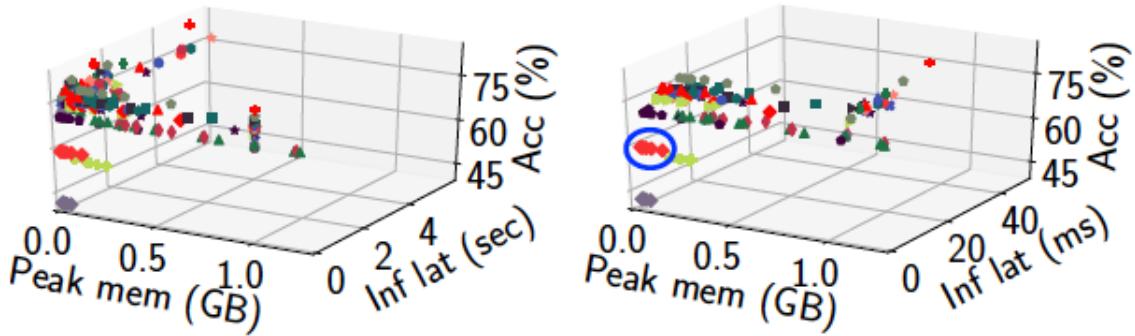


图 2.20 左图为 158 个图像分类模型的峰值内存利用率、延迟和准确率；右图为这 158 个模型中延迟小于 50ms 的模型。

行；3) 模型推断结果经过前端返回给用户。Metadata Store 用以存储每个模型变体的特征，例如该模型的资源消耗情况、准确率和延迟等等。Model Repository 负责存储模型，在 INFaaS 中是使用纯内存的方式实现的。同时，开发者还可以向 INFaaS 注册和提交新的模型，提交之后的模型会先由 Variant-profile 对其进行采集相关特征，并存储到 Metadata Store 中。

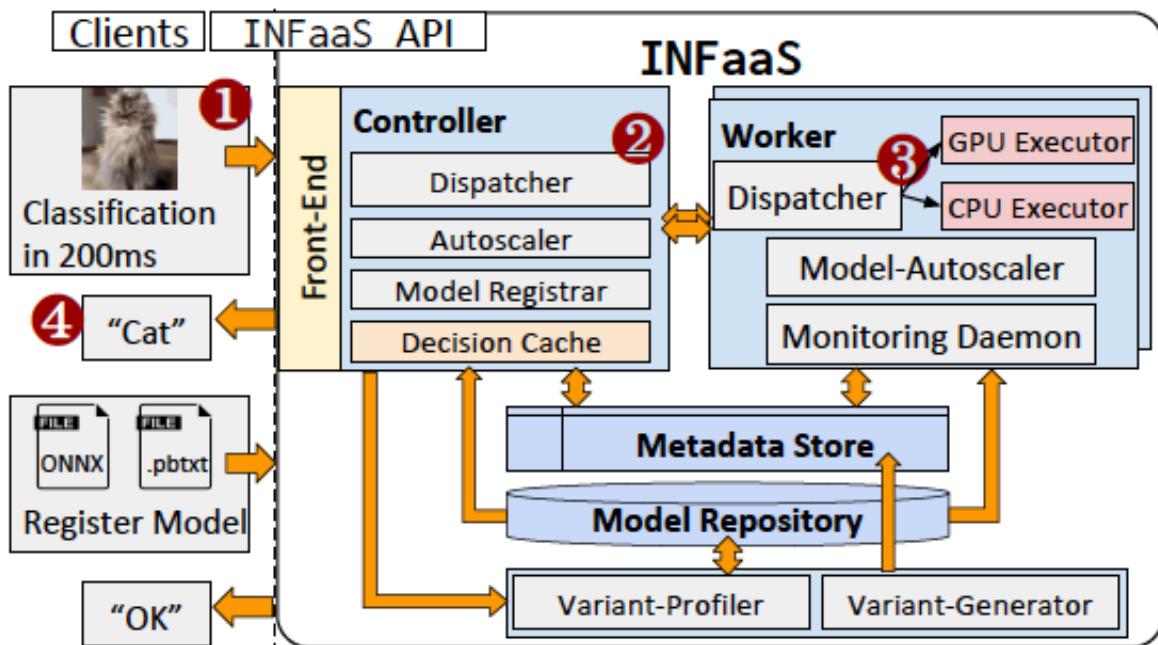


图 2.21 INFaaS 系统架构图。

值得一提的是，INFaaS 称其模型部署方式为 model-less 的，并不意味着模型（即 model）在其系统中不存在。和 serverless 中用户无需关注后端服务器类似，model-less 意味着用户无需关注其请求到底是由系统中的那个模型变体实际为其服务的。事实上，可以认为 model-less 是更上一层的 serverless 的部署方式，因为用户不仅不需要关注后

端到底存不存在服务器，甚至都不需要关注后端的模型到底是什么。

从基于云上 VM，到多种服务混用，再到纯 FaaS，公有云上的模型部署正在一步一步地向简洁高效发展。从云的角度理解，这是一种云原生的发展方式，即云上的模型部署依赖于云本身的能力，生在云上长在云上，将弹性伸缩、容错、高并发等工作交由云商或者第三方服务来做，开发者或者用户只需要关注与业务本身相关的代码逻辑，即“将麻烦留给云商，将方便带给用户”。

2.3 小结

本章按照机器学习模型开发的工作流的顺序，即模型开发与调试-模型超参数调整-模型部署，介绍了面向人工智能的云计算系统软件当前的研究现状。首先介绍了公有云厂商一站式机器学习的代表性系统 AWS Sagemaker。Sagemaker 支持从训练到部署的全部工作流程。其训练过程中使用了流行的参数服务器架构（即 Parameter Server），并为模型训练和调参设计了统一的用户 API，用户只需要编写几个函数并定制几个规则即可让 ML 模型的训练和调参在云端自动展开。然而生产环境中的复杂需求也催生了一批基于云服务的第三方机器学习模型训练和调参系统，这些系统充分利用云资源类型和计费方式上的多样性，使机器学习任务在云上经济高效地运行。对于部署阶段亦是如此，模型部署方式正在经历 VM-混合-FaaS 的模式转变。未来基于 FaaS 如何对模型部署进行更好地支持，将持续成为工业界和学术界的研究热点。

第三章 云环境中的异构硬件为人工智能带来的机遇与挑战

随着硬件技术的不断发展，云环境中支持机器学习任务的硬件也从单一的 CPU 变成了包含 CPU、GPU、FPGA、TPU 以及 SGX 等安全硬件的异构资源。利用这些异构资源，可以给机器学习任务带来一定增益。例如 GPU、TPU 和 FPGA 等硬件的 SIMD 计算模式天然适合 ML 这种迭代式、大批量的负载，从而可以大大加速 ML 训练。而诸如 SGX 等硬件让机器学习安全有了从硬件层面得到解决的可能性。

而异构硬件为 ML 任务带来机遇的同时，也带来了一定的挑战。例如，在公有云共享资源池的背景下，如 GPU 和 FPGA 等新兴加速硬件没有成熟和系统的虚拟化方案，用户的使用这些硬件的模式仍然以独占为主。即便是共享，也没有成熟的隔离机制保证性能以及用户之间不会相互干扰。再比如，SGX 加持的安全计算环境中，可信内存的大小十分有限，如何在有限的资源下支持可信的模型部署甚至模型训练，是亟待解决的问题。

本章将以加速型硬件 GPU 和安全型硬件 SGX 为主，介绍云环境中异构硬件为机器学习任务带来的机遇与挑战。

3.1 云环境中机器学习作业对 GPU 的利用

本节将先对 GPU 的现状做简要介绍，包括现代 GPU 的架构以及常见的 GPU API 和编程模型。然后，本章将从两个角度展开当前学术界针对 ML 任务对 GPU 的利用的研究：1) One Job on Multi GPUs。即一个 ML 任务需要多个 GPU 设备。这种情况一般出现在大型模型的分布式训练中。在一个大规模 GPU 集群中，不同的训练任务对 GPU 数量的需要可能不一样，如何对这些任务进行调度，使系统的资源利用率最高，同时保证训练任务的性能，是学术界研究的重点问题；2) Multi Jobs on one GPU。即在同一个 GPU 设备运行多个 ML 任务。这种情况在资源短缺或多租户的场景下较为常见。

3.1.1 GPU 简介

3.1.1.1 GPU 架构

GPU，即 Graphics Processing Unit，采用了与传统的多核处理器完全不同的架构。GPU 的设计围绕和让吞吐率最大化这一原则，提供了上千个简单的计算核心和高带宽内存。这样的实际使得 GPU 能够使一个应用程序拥有很高的并行度，从而使程序的吞

吐率最大化。理想情况下，对于一个 GPU 应用程序，其包含的若干线程应该分别对应其数据空间的不同位置，而彼此之间由没有因果关联，从而可以并发地执行。与之相反，传统的处理器是为了能更快地执行线性的代码段而优化的。因此其单个核心的复杂度非常高，且单个处理器中的核心数量相对较少。且传统的处理器一般会使用复杂的控制逻辑和较大的本地缓存以处理传统程序中常见的条件分支、上下文切换以及较差的数据本地行。

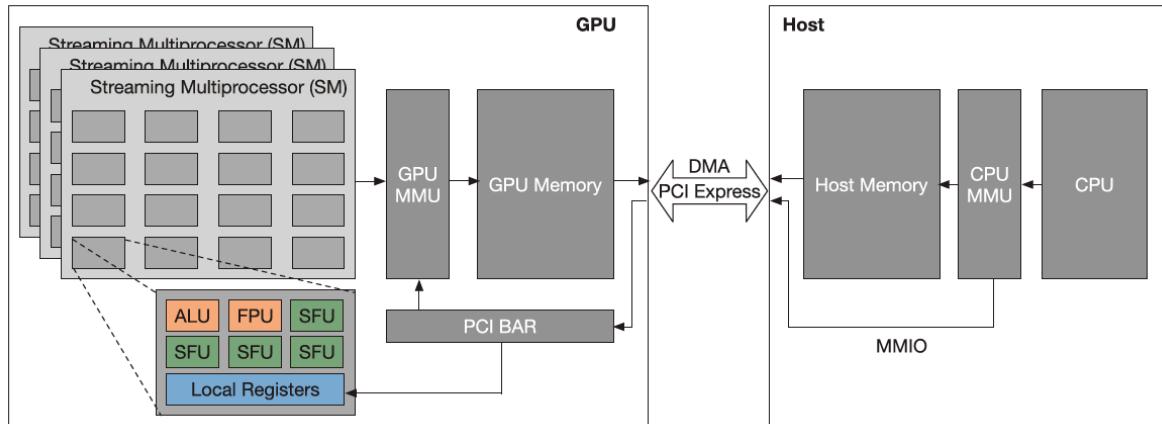


图 3.1 GPU 系统架构。

图 3.1 展示了一个典型的 GPU 系统的架构。图中的 GPU 部分是根据 Fermi 架构的 Nvidia GPU，但是现代的 GPU（例如新一代的 Nvidia GPU 和 AMD GPU）也遵循类似的设计。一个 GPU 拥有多个流式处理器（streaming multiprocessor, SM），每一个 SM 拥有 32 个计算核心。每个 SM 也拥有 L1 数据缓存和低延迟共享内存。每个计算核心拥有本地的寄存器、一个整型计算单元（Integer Arithmetic Logic Unit, ALU）、一个浮点计算单元（Floating Point Unit, FPU）以及若干特殊函数单元（Special Function Units, SFU，用来计算特定的函数值，例如正弦函数 sine 和余弦函数 cosine）。GPU 的内存管理单元（MMU）为 GPU 程序提供虚拟地址空间。MMU 会根据应用程序的页表，会将一个 GPU 地址解析为物理地址。

Host 通过 PCIe 高速通道与 GPU 连接。Host 上的 CPU 通过内存输入/输出映射（MMIO）与 GPU 进行交互。GPU 寄存器以及设备内存可以被 CPU 通过 MMIO 接口直接获取。此外，量比较大的数据交换可以通过 DMA（Direct Memory Access）实现。DMA 可以将数据在设备内存和主机内存之间快速传输。

3.1.1.2 GPU API 和编程模型

常用的 GPU API 和编程模型包括 OpenGL, Direct3D, CUDA 和 OpenCL 等，在游戏开发、图像处理、高性能计算等场景下非常常用。

OpenGL 是一个利用 GPU 硬件加速图形处理的库，通常应用于电子游戏、图像处理以及科学计算中的数据可视化。OpenGL 实现了一个硬件无关的 API，兼容适配不同的底层硬件。

Direct3D 是由 Microsoft Windows 提供的图形库，通常在一些性能敏感的场景中（例如电子游戏）负责图形渲染工作。Direct3D 同样实现了硬件无关的 API，并且向用户暴露了能够使用 GPU 高级特性的 API，例如 Z-buffering，W-buffering 等。

CUDA 是由 Nvidia 提供了专为 Nvidia GPU 适配的并行加速库。CUDA 允许开发者在 GPU 上利用高并行的特点开发特定的程序，这种情形下的 GPU 被称为 GPGPU (General Purpose GPU)。CUDA API 是与编程语言高度绑定的，例如 C 和 C++。当前流行的机器学习框架，例如 Tensorflow 和 Pytorch，在存在 GPU 设备的环境中，都支持利用 CUDA 对其模型训练进行加速。

OpenCL 也是一个并行加速库，其基础功能和 CUDA 类似，最大的区别在于 OpenCL 可以被应用于非 Nvidia 的 GPU。

3.1.2 One Job on Multi GPUs: 机器学习集群中的 GPU 调度算法

TODO: 补充基础的 GPU scheduling 算法

随着深度学习的兴起和快速发展，在计算机视觉、自然语言处理等领域使用的模型参数越来越多，规模越来越大。而对于大规模的深度学习模型，分布式训练 +GPU 硬件加持几乎成为了标配。分布式训练的场景中，GPU 之间的数据传输时有发生。然而在一个多 GPU 的集群环境中，位于整个系统拓扑结构不同位置的 GPU 之间的数据传输速度是有很大差异的。

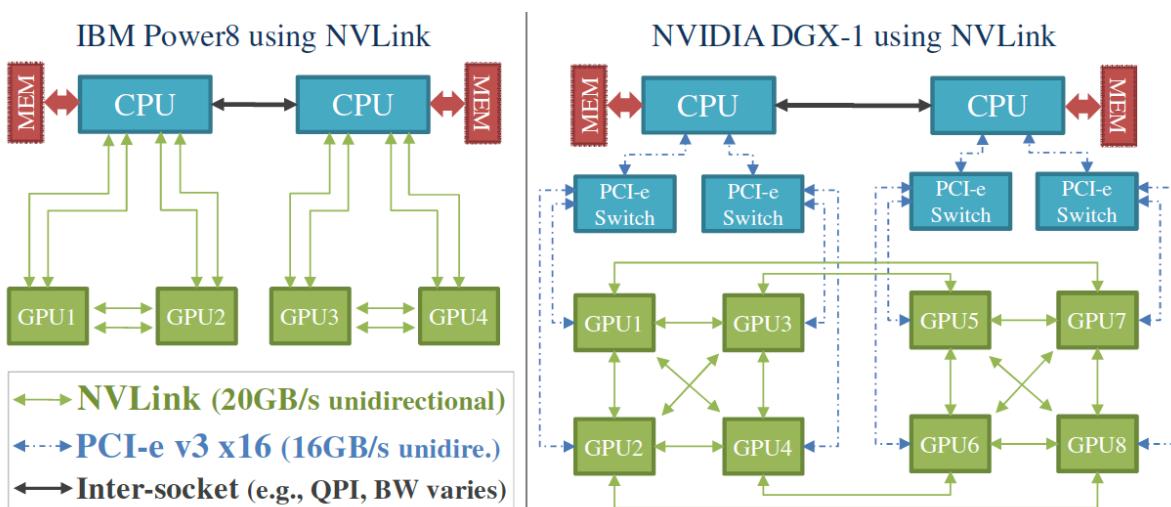


图 3.2 GPU 拓扑结构示例。

图 3.2 展示了两个 GPU 服务器中 GPU 的和其它硬件之间的拓扑结构图。左图为

IBM 的 Power8 服务器，有两个 CPU 和四个 GPU。右图是 NVIDIA 和 DGX-1 服务器，有两个 CPU、四个 PCI-e 桥接器和 8 个 GPU。绿色的线、蓝色的线和黑色的线分别代表 NVLink、PCI-e 和 Intel 的 QPI 三种连接方式，数据传输速度由快到慢。位于同一个 CPU socket 下的两个 GPU（例如左图中 GPU1 和 GPU2）之间的数据传输可以通过 NVLink 进行，是最快的。位于不同 CPU 上的 GPU，只能通过 PCI-e 通道和 QPI 通道进行通信（例如左图中的 GPU1 和 GPU3、右图中的 GPU1 和 CPU）。而如果位于两台不同服务器上的 GPU 之间需要通信，则除了需要经过上述数据通路之外，还要经过网络传输，即使是万兆网（10Gbps），也要比上述传输方式慢约一个数量级。

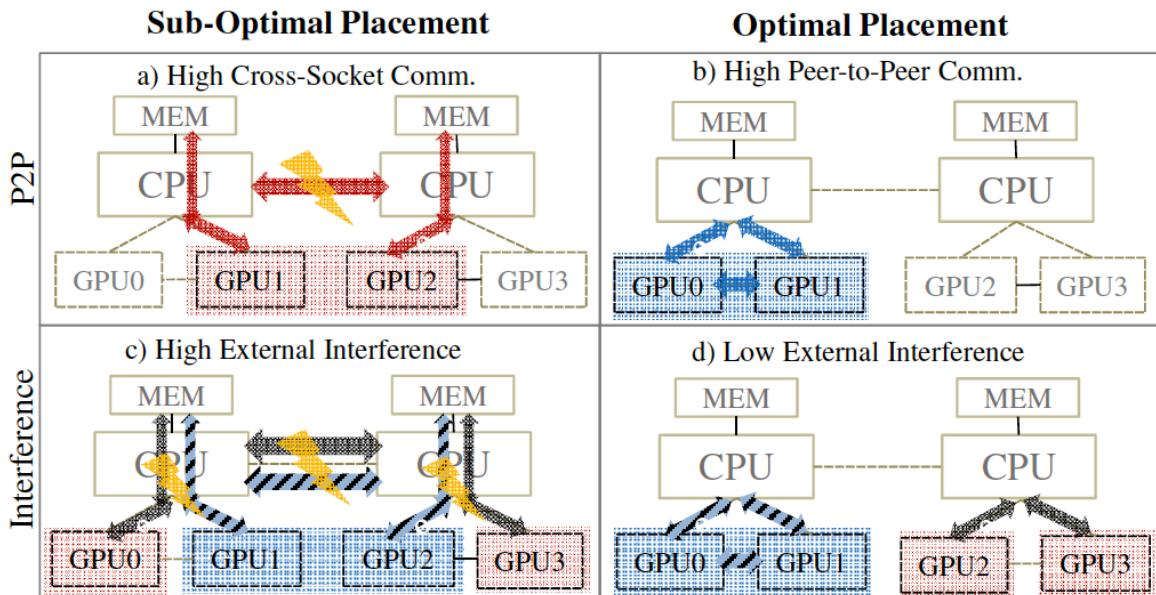


图 3.3 GPU 调度示例。

因此，在 GPU 集群中调度分布式 ML 训练任务时，需要充分考虑 GPU 之间的拓扑结构，否则可能会造成严重的性能损失。图 3.3 展示了 GPU 调度的两个例子和与其对应的四种调度方案。上面的两张图对应了 P2P（即两个 GPU 之间需要互相通信）的场景中的两种调度方案，显然右边的方案是更优的方案，因为分配到的两个 GPU 位于同一个 CPU socket 上，于左边相比其数据通路具有更高的传输速率。下面的两张图对应了不当分配可能造成的性能干扰。显然，右边的分配方式也是更优的方式，因为左边的分配方式造成了两个 Job 在数据传输时共用同一条数据通路从而造成竞争，使性能降低。

M. Amaral 等人 [2] 在 2017 年提出了一种云环境下基于 GPU 拓扑结构的调度方法。其将 ML 任务和集群都抽象成图的形式。在 ML 的任务图中，图的节点为 GPU，节点之间的边代表两个 GPU 之间存在数据交流。任务图中的边没有权重。在集群的图中，图的节点为 CPU，GPU，PCI-e 桥接器，CPU socket 或者网络交换机。不同节点之间的

连线代表这两个节点之间可以相互通信，连线的权重代表通信速度，数值与越小通信速度越快。图 3.4 表示了图 3.2 所对应的拓扑结构图。

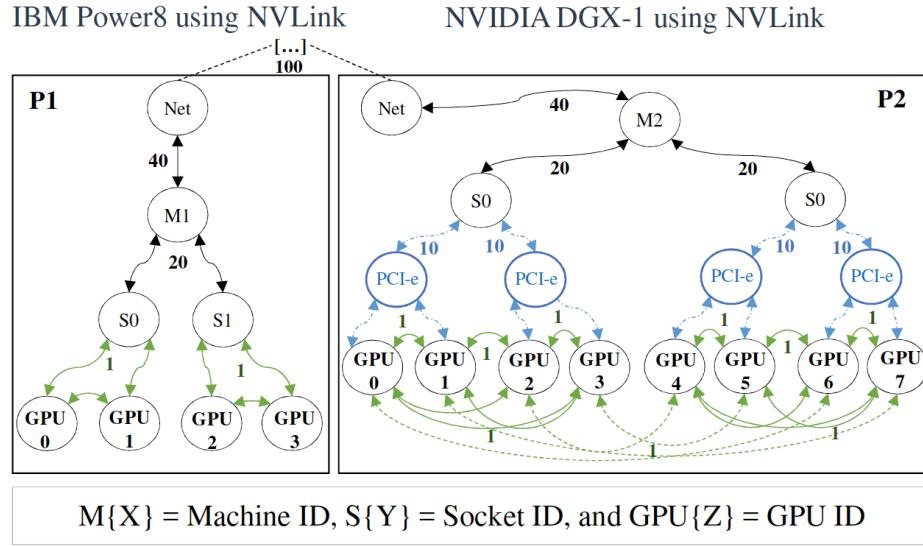


图 3.4 GPU 拓扑结构图。

M. Amaral 等人通过对两个图进行匹配并最小化公式 3.1 中的目标来实现 GPU 的调度，其中 $\alpha^{cc} + \alpha^b + \alpha^d = 1$ ， t 、 I 和 ω 分别代表 GPU 之间的通信代价、来自外部的资源干扰和资源碎片化程度，分子代表实际的值，分母代表最坏情况下的值。该公式旨在综合考虑减小 GPU 之间的通信代价、降低不同任务之间的干扰以及使整个系统的碎片化尽可能降低，使整个集群处于一个相对最佳的状态。

$$MIN \alpha^{cc} \frac{t^{cc}}{t_w} + \alpha^b \frac{I^b}{I_w} + \alpha^d \frac{\omega^d}{\omega^w} \quad (3.1)$$

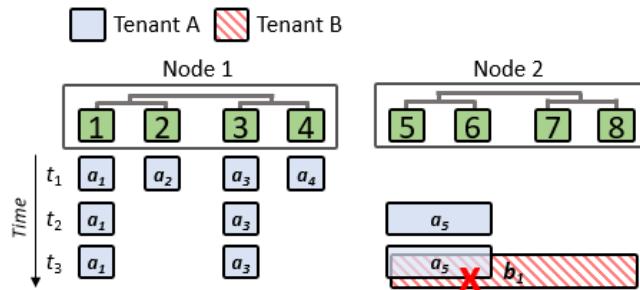


图 3.5 GPU 集群中的调度异常：一个任务被阻塞。

M. Amaral 等人的工作综合考虑了多项指标，以使得集群达到一个最优状态。在这样的集群中，所有的任务都被视作拥有同样的优先级。因此，可能存在集群中常见的长尾问题。图 3.5 就描述了这样一个场景。租户 A 和租户 B 都需要使用 GPU，然而由

于调度问题，虽然集群中还剩余 4 个 GPU，却无法分配给 B。这种情况被称之为调度过程中的异常（anomaly）。

H. Zhao 等人在 2020 年提出了 HiveD[32]，使用 Virtual Private Cluster 的概念和伙伴算法（即 Buddy Cell 算法）来解决调度中的异常。HiveD 中将任务分为两类：高优先级的需要保证资源的任务和低优先级的尽可能用“机会性资源”来提升集群资源利用率的任务。其使用的伙伴算法在 Linux 内核中的内存管理模块中也是常用的算法。图 3.6 表达了 HiveD 中两类任务中不同的资源视角。HiveD 中将 GPU 按照连接方式划分为不同级别的 Cell，如图中单个 GPU 称为 Level 1 的 Cell，位于同一个 CPU socket 上的两个 GPU 称为 Level 2 的 Cell，位于同一 CPU 下的四个 GPU 称为 Level 3 的 Cell，位于同一台服务器下的八个 GPU 称为 Level 4 的 Cell，位于同一机架上的不同服务器上的 GPU 集合称为 Level 5 的 Cell。HiveD 假设任务对于 GPU 需求都是与某一个 Level 的 Cell 中的 GPU 数量匹配的，其基本算法思路如下：当有一个任务需要一个 Level k 的 Cell 时，先查看系统中有无现有空载的 Level k 的 Cell，如果没有则将一个 Level k+1 的 Cell 分裂为两个 Level k 的 Cell，如果再没有则继续向上一层 Cell 请求分裂，直到存在空载的 Cell 为止。同时，如图 3.6 所示，高优先级和低优先级的任务视角中，资源的使用情况是不同的。如果一个 Cell 被高优先级的任务占用，则在低优先级任务的视角中，这个 Cell 被标记为“已使用”，不能被抢占。反之，低优先级任务占用的 Cell 在高优先级任务的视角中被标记为 dirty，在必要的情况下可以抢占其使用。

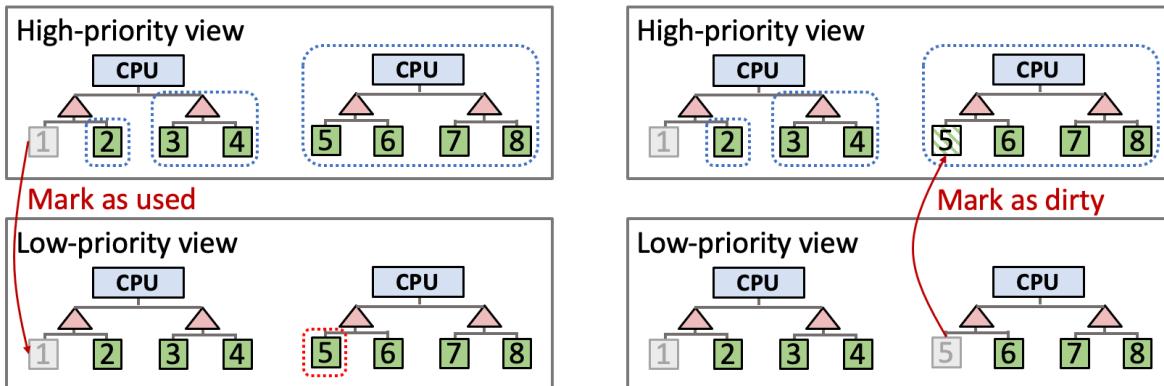


图 3.6 HiveD 中的两种视角。

总体而言，与传统的集群资源调度其类似，以 GPU 设备为最小调度单位的系统调度器中，首要的目标是保证集群的吞吐率，即单位时间内处理任务的数量。近年的做法都是基于 GPU 之间的互联情况对 GPU 设备（组）进行分类，进而使用各种算法使任务的需求与系统当前的状态匹配，达到降低任务运行时间、提升系统吞吐率的目标。在此基础上，在一些碎片化的资源上部署一些低优先级的作业，可以进一步提升系统

的资源利用率。

3.1.3 Multi Jobs on one GPU: 机器学习集群中的 GPU 共享机制

除了以单个 GPU 为单位，在真实的业务场景中也存在多个任务共享同一个 GPU 的需求。该需求出现于云计算刚刚兴起的早期，即如何对 GPU 进行虚拟化。常用的应将资源例如 CPU，网络和存储等都有非常成熟的虚拟化方案。以 CPU 为例，常见的虚拟化一般可以分为虚拟机和容器两大类。虚拟机有着比云计算更悠久的历史。IBM 公司在 1970 年阐述了其最原始的虚拟机系统设计。在 1972 年，虚拟机第一次在 IBM 公司的大型机上出现。通过 IBM VM370 系统能够将一台大型机分为可独立运行操作系统的多台虚拟机。在计算虚拟化技术中，原本的物理计算机被称为宿主机（Host），运行在宿主机之上的虚拟机被称为客户机（Guest）。虚拟机技术的核心是 VMM（虚拟机监控器 Virtual Machine Monitor 或者虚拟机管理器 Virtual Machine Manager），也被称为 Hypervisor。容器是近年来更受关注的轻量级虚拟化技术。与虚拟机技术不同，容器并不对 CPU 指令以及硬件设备进行虚拟化，它是一种操作系统级的虚拟化技术。容器通过操作系统内核提供的功能，来实现不同用户进程的资源隔离与访问隔离，形成一个个独立的运行时环境，即“容器”。

云计算的兴起离不开虚拟化的发展。当前，主流的云厂商例如 AWS, Azure 等，都同时提供了虚拟机服务和容器服务。这些服务基于对计算资源（主要是 CPU）、网络资源（带宽、虚拟网络）和存储资源（虚拟云盘）的虚拟化，支持用户在虚拟计算环境中独立且互不干扰地运行各种负载。然而 GPU 作为现今一种重要的计算资源，却始终没有一种统一的、完善的虚拟化方案应用于公有云环境，而是在不同的业务场景中存在不同的虚拟化方案。

根据实现方式的不同，当前 GPU 的虚拟化大致可以分为三类：

1. API 转发。这种方式在 GPU 架构的上层对 GPU 进行虚拟化。由于 GPU 的厂商一般不会开源 GPU 的驱动源码，因此在驱动层对 GPU 进行虚拟化是非常困难的。基于 API 转发的虚拟化实现方式类似于远程过程调用（Remote Procedure Call, RPC），即在虚拟机 OS 中实现一个 agent，该 agent 会截获应用程序所有对 GPU 的调用，并将相关调用转发到宿主机上的 GPU 设备上进行实际的执行，执行完毕后再返回给虚拟机 OS 中的应用程序。

2. 半/全虚拟化。这种方式在驱动层对 GPU 进行虚拟化（一般由厂商来实现），一般会定制一个专门的 driver 安装在虚拟机 OS 中。与 API 转发相比，这种方式由 driver 直接调用宿主机的 driver API，不需要你 agent 的转发，从而缩短了调用链，使其性能更优。

3. 基于硬件的虚拟化。这种方式也称为 pass-through (译作直连), 即赋予虚拟机直接访问宿主机设备的能力。这种方式产生的额外开销是最小的, 但是一般而言隔离型比较差。

上述三种方式中, API 转发是学术界的工作最为常用的方法, 无论是在基于虚拟机的环境中, 还是在基于容器的轻量级虚拟化环境中。图 3.7 描述了虚拟机环境中基于 API 转发的 GPU 虚拟化的工作流程。

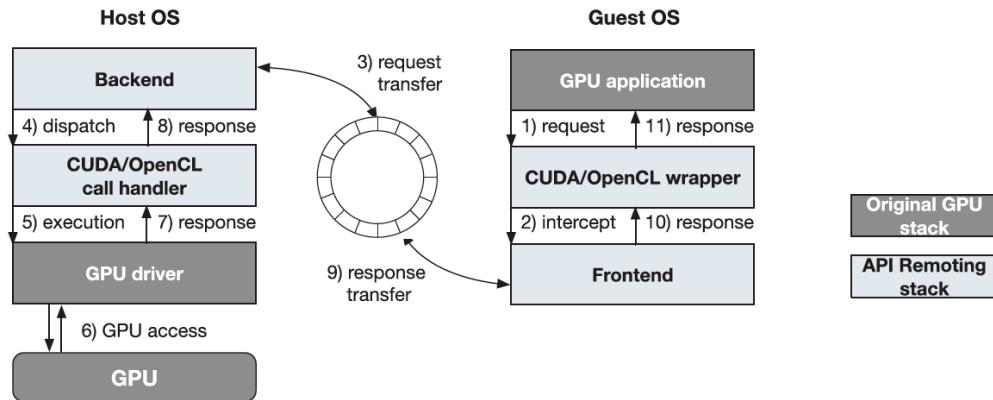


图 3.7 基于 API 转发的 GPU 虚拟化。

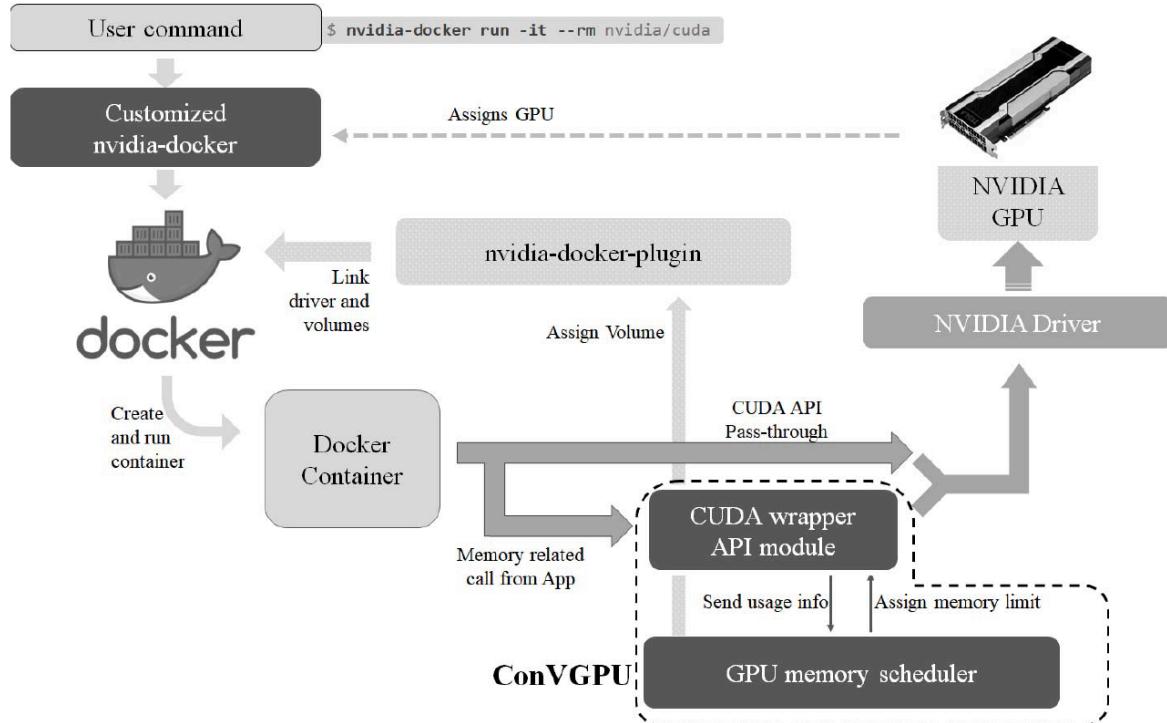


图 3.8 ConVGPU 架构。

在容器环境中基于 API 转发的方式和上图类似。D. Kang 等人在 2017 年提出了

ConVGPU[14]，通过在 docker 容器中安插一个 CUDA wrapper 来对一些与 GPU 显存相关的 API（例如 `cudamalloc`, `cudafree` 等）进行截获，并实现了一个管理全局显存分配的调度模块。如图 3.8，ConVGPU 实际上是以一个中间件的形式存在于容器环境中的。在正常的 API 调用路径下，插入一个显存管理的组件，在软件层面实现了 GPU 显存的共享和一定程度上的隔离。

而更多的 GPU 共享的解决方案是与具体的应用场景高度绑定的，尤其是在运行机器学习任务的集群中。W. Xiao 等人在 2018 年提出了 Gandiva[28]，主要面向深度学习集群中基于 GPU 的模型训练任务进行调度。有别于传统的 suspend/resume 机制，Gandiva 使用 grow/shrink 机制。图 3.9 表现了 Gandiva 的 grow-shrink 机制。Gandiva 中主要存在如下三种动作：1) grow。即当前如果存在空闲的资源，Gandiva 会让存在空闲资源的节点上的某些任务占用的资源拓展到空闲资源上，以提升系统的资源利用率和模型的训练速度（如图 3.9 中的 Gandiva:(3) Grow-shrink）。2) shrink。当系统中有新的训练任务到达时，Gandiva 一般不会将一个任务直接 shutdown，而是将其所占用的资源进行裁剪，腾出新的资源以供新的任务所用（如图 3.9 中的 Gandiva:(3) Grow-shrink）。3) migration。当系统其它节点中存在空闲资源时，Gandiva 会将一个非独占的任务迁移到该空闲资源上（如图 3.9 中的 Gandiva:(2) Migration）。值得注意的是，Gandiva 在实验中发现，深度学习任务的显存占用一般呈周期性涨落的趋势，其峰值显存占用率和谷值显存占用率差距非常大。因此其在迁移任务时会根据任务占用显存的变化趋势，在任务占用显存最小的时候对任务进行迁移，以降低迁移产生的额外开销。

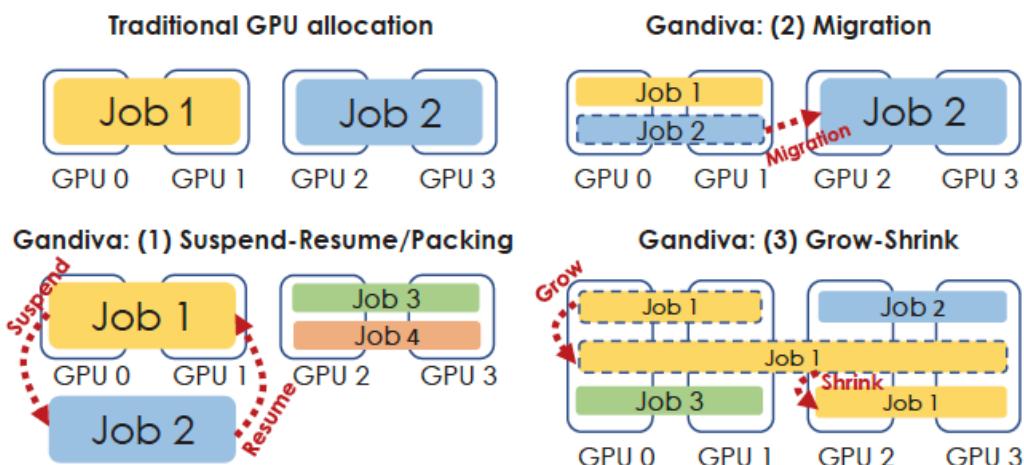


图 3.9 Gandiva grow/shrink 机制。

事实上，Gandiva 中所提到的深度学习任务显存占用的周期性涨落趋势，也可以用来实现更细粒度的 GPU 共享。P. Yu 等人在 2020 年提出了 Salus[30]，使用快速任务切换和显存贡献实现 GPU 的共享。Salus 将显存分成多个 Lane，一个 Lane 中可以存在一

个或者多个深度学习模型训练任务。由于深度学习模型训练任务都是周期性迭代的过程，Salus 调度的最小单位是任务的一个周期（在模型训练中通常体现为一个 batch）。图 3.10 展示了 Salus 的架构和一个调度示例。Salus 为常用的 ML 框架如 Tensorflow 和 Pytorch 定制了相应的适配器。对于用户的模型训练程序的每一次迭代周期，都会以一个 session 的形式提交到 Salus 的 session 管理器。Salus 根据提交的 session 的显存变化曲线，将 session 调度到合适的 Lane 中。然后调度器根据系统当前显存分配的状况，调度相应的 Lane 在 GPU 上执行。右图中即为一个例子，Job 1 和 Job 2 在显存变化曲线上存在明显的互补性，使得 Job 2 的波峰恰好可以将 Job 1 的波谷填满，因此 Salus 调度其在同一个 Lane 上执行。

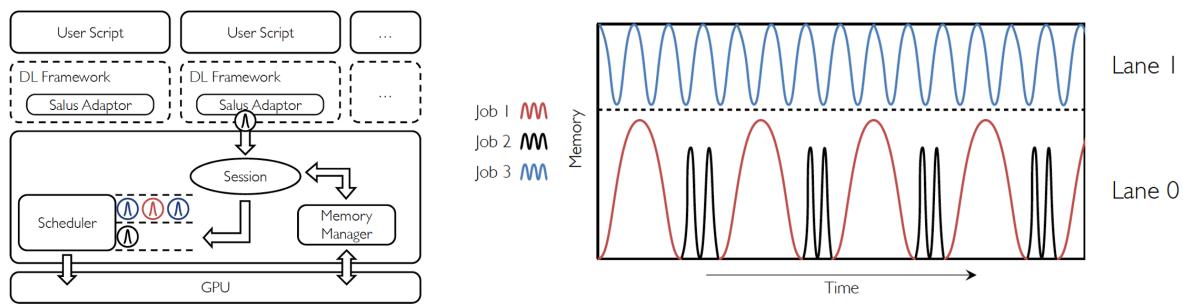


图 3.10 Salus 架构与调度示例。

云环境中的 GPU 共享其实也是 GPU 集群资源调度的一部分，只不过其最小调度单位由整个的 GPU 变成了单个 GPU 的一部分。随着 GPU 虚拟化和共享机制的日渐成熟，多个任务共用同一个 GPU 在云环境中将成为常态。

3.2 云环境中机器学习作业对 SGX 的利用

能够安全可信地使用由第三方云厂商提供的计算服务是云消费者的重要需求。尤其是当前人工智能技术盛行，很多个人开发者、研究团队和中小型公司都有利用公有云算力训练机器学习模型的需求。但是这些用户需要将具有一定私密性的数据上传到云厂商，这在一定程度上是具有潜在的安全性风险的。因此用户需要以一种安全的方式对云资源进行利用。当前，能够满足这一需求的主流解决方案包括同态加密（Homomorphic Encryption）和可信执行环境（Trusted Execution Environment）。同态加密是一种密码学技术，可以对加密后的密文直接进行计算，并且计算结果解密后与明文计算的结果相一致。2009 年斯坦福大学的 Craig Gentry 首次提出了通用的同时支持加法同态和乘法同态的全同态加密方法，在理论上证明了云计算资源安全使用的可能。用户可以将数据加密后在云端进行计算，然后将结果传输回本地进行解密，这样可以同时

做到既依赖了云端的强大算力又保证了数据安全性。但由于该方案引入了巨大的计算开销，出于性能原因至今仍然没有得到广泛的使用。

可信执行环境是一种基于硬件的解决方案，其本质上相当于为用户提供了一种安全的沙箱环境。用户可以在可信执行环境中进行计算，并且能够得到机密性和完整性的保障，而包括 Hypervisor、操作系统、主机上可信执行环境以外的其他应用都没有权限读写可信执行环境中的数据，从而保证了用户使用云计算资源的安全性。代表性的可信执行环境实现包括 Intel 的 SGX 技术和 ARM 的 TrustZone 技术。相比于全同态加密，这些技术有着更高的计算效率，近年来在云计算环境下有着强隐私保护需求的许多场景里得到了实际应用。因此，本节主要着眼于可信执行环境这种云计算资源在机器学习场景下的安全性利用方案，以 Intel SGX 技术为重点，对相关技术进行调研。

3.2.1 SGX 技术简介

可信执行环境是解决服务使用者与供应者之间信任问题的一种流行方案。在服务器端，英特尔软件保护扩展（Intel Software Guard Extensions，SGX）是目前最主流且被使用最多的可信执行环境技术，用于保护运行在不受信任平台上程序的机密性（Confidentiality）和完整性（Integrity），于 2015 年在 Intel Skylake 系列的 CPU 中首次被推出。图 3.11 概述了 SGX 的简单特性。SGX 提供了被称为飞地（Enclave）的硬件安全沙箱，用于托管用户代码和数据，防止其被飞地外不受信任的特权进程（例如 Hypervisor 和操作系统）窃取或篡改。本章提出的解决方案基于 SGX 设计和实现。

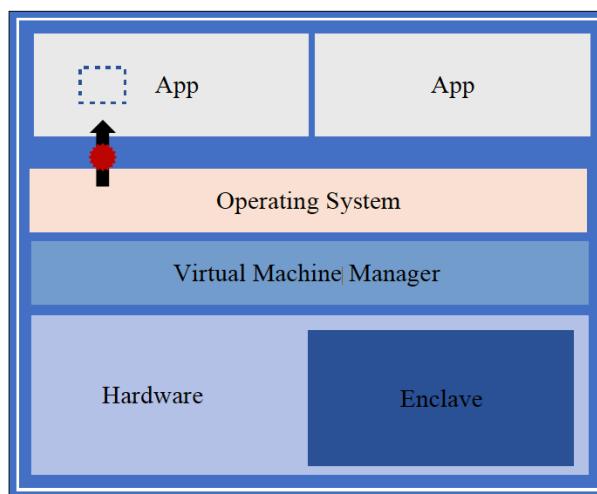


图 3.11 SGX 架构图。

截至 2021 年 3 月，SGX 包括两个主要版本，最初的版本通常被称为 SGX 1.0，另一个版本具有增强的特性，如飞地动态内存管理，被称为 SGX 2.0。由于 SGX 2.0 仅在有限的 CPU 系列上支持，因此本文只讨论 SGX 1.0，其已经包含了 SGX 核心的安全特性。

SGX 技术通过一组 CPU 指令扩展和专用硬件架构来构建所谓的飞地。飞地里的代码和数据都运行在特殊的加密内存里，只有飞地里的程序运行在 CPU 里时才会对相应的代码和数据进行解密。因此，即使攻击者有机会接触到硬件，通过窥探内存（Hardware Snoop）的方式也只能看到加密后的信息。在一个飞地进行初始化之前，代码和数据保存在主机的非可信区域。同时，SGX 也无法保障飞地以外数据的安全性。因此，为了保护秘密，在使用 SGX 时，数据应该在飞地外进行加密，然后传输到飞地内解密后进行计算。

认证（Attestation）： SGX 提供了认证机制，其功能是让用户可以验证一段已知的程序确实是运行在一台支持 SGX 功能的计算机的飞地里的。认证又包括本地认证（Local Attestation）和远程认证（Remote Attestation）。本地认证使得一个飞地可以向运行在同一平台上的另一个飞地证明自己的身份。远程认证使得一个飞地可以向远程第三方实体证明自己的身份。SGX 提供了飞地度量值（即 MRENCLAVE 和 MRSIGNER 两个值）用来在认证过程中表明飞地的身份并验证飞地的完整性。

密封（Sealing）： 在飞地中运行的代码和数据是易失性的。密封是 SGX 提供的一种机制，用于将数据保存到持久化的非可信存储中，例如硬盘。在密封过程中，应用使基于可信执行环境的可信应用快速构建与支撑机制用密封密钥（Sealing Key）在飞地中加密数据。密封密钥只能由该应用的飞地程序获得，并且是从根密封密钥（Root Sealing Key）衍生而来的。根密封密钥则是硬件编码在 CPU 中，并且每一个支持 SGX 的 CPU 的根密封密钥都是互不相同的，这就决定了某一飞地在某台机器上密封的数据只能在该台机器上进行解密。

飞地页面缓存（Enclave Page Cache, EPC）： 飞地所驻留的特殊加密内存区域被称为 EPC。EPC 由专用硬件加密，禁止所有来自非飞地内存的访问。EPC 以 4KB 的粒度划分为页面进行内存分配。飞地在初始化时申请它需要的所有 EPC 空间，并且在分配之后不能动态修改。EPC 的总大小非常有限，在一台计算机上，SGX 飞地应用程序可以使用的 EPC 的大小是 128MB。在这其中，只有约 93.5MB 的大小可供用户程序和数据使用，剩余空间预留用来存储 SGX 相关的元数据。在当前的大数据时代，EPC 的空间过于稀少，并且单机上的 EPC 会被所有的 SGX 程序分享，这就导致每一个飞地实际可用的 EPC 空间就更小了。在服务器市场主流的 Linux 操作系统上，SGX 驱动支持通过页交换技术来实现 EPC 超配，每个飞地实际可以申请超过 128MB 的内存使用。当 EPC 中没有空闲页分配给飞地使用时，SGX 驱动程序会基于缓存替换算法将 EPC 中的某些页面加密后放置到非加密的常规内存 DRAM 上去。当这些换出的页面在重新被需要使用时又会被换入到 EPC 中。因为换入换出页面时涉及到完整性校验等高计算开销的过程，因此会导致应用程序性能的显著降低。先前的工作表明，这种开销平均会使

系统性能降低 5 倍。由于 EPC 很小，并且由运行在一台机器上的所有飞地竞争，因此可能会频繁地触发页交换。

SGX 应用程序开发：当前主要有两种方式来开发面向 SGX 的应用程序。第一种是基于功能划分的方式。使用这种方式来开发程序通常需要将整个应用设计为可信与非可信两个部分。敏感数据由运行在飞地内的可信程序处理，其他功能由飞地外的非可信程序完成。使用 Intel 原生 SGX 软件开发工具或如 Asylo 一类的开源框架来编写软件是这类开发方式的典型代表。这类方式可以更有效地利用 EPC 资源，但需要付出较大的努力对传统非 SGX 环境下的留存应用进行重构。第二种方式是使用库操作系统（LibraryOS, LibOS）来进行开发。库操作系统通过实现应用所需的系统调用来支持在飞地内运行整个程序。这种方式为留存应用提供了更好的兼容性，对开发人员更加友好。目前主流的面向 SGX 环境设计的库操作系统包括 Haven、Panoply、Graphene-SGX 和 Occlum。

3.2.2 基于 SGX 技术的上层应用

首先在最基础的容器粒度上，Arnautov 等人提出了一种基于 SGX 实现的安全容器 SCONE。SCONE 对 SGX 进行了进一步的抽象来保护容器中的进程。其为开发者提供了 C 语言标准库接口来便利地开发基于 SGX 的应用程序。SCONE 的设计中系统调用在可信执行环境之外进行，但通过加密技术保证了可信执行环境之外的数据也是安全的。进一步地，为了降低在可信执行环境中进行线程切换的开销，SCONE 将操作系统线程映射为逻辑的应用线程，通过在应用线程间调度操作系统线程来最大化线程在可信执行环境中的运行时间。在易用性上，SCONE 兼容 Docker，对 Docker 用户是透明的。在 Apache、Redis、NGINX 等应用上的测试表明，SCONE 可以达到原生应用 0.6 至 1.2 倍的吞吐量。

SCONE 针对的是单机的容器环境，Hunt 等人则基于 SGX 技术设计和实现了面向分布式环境的可信计算中间件 Ryoan。Ryoan 针对的是无状态、采用面向请求数据模型的应用，其提出了新的执行模型，对不可信代码模块施加限制，并严格执行防止机密泄露的 I/O 策略。Ryoan 可以使得互不信任的多个参与方能够在不受信任的平台上（例如云平台）以分布式的方式进行敏感数据的计算。Hunt 等人更进一步地基于 Ryoan 开发了包括图像处理系统、垃圾邮件过滤器在内的多个真实应用作为研究案例。

SCONE 和 Ryoan 等都是基于 SGX 在系统软件这个层次所做的工作，用以帮助开发者更便捷地使用 SGX 技术。相比之下，VC3 是更上层的工作，其基于 SGX 开发了一个可信的分布式数据处理框架，可以运行在未修改的 Hadoop 之上。图 3.12 阐述了 VC3 的设计理念：在 VC3 中，数据和代码（例如 Map 和 Reduce 程序）在可信执行环境

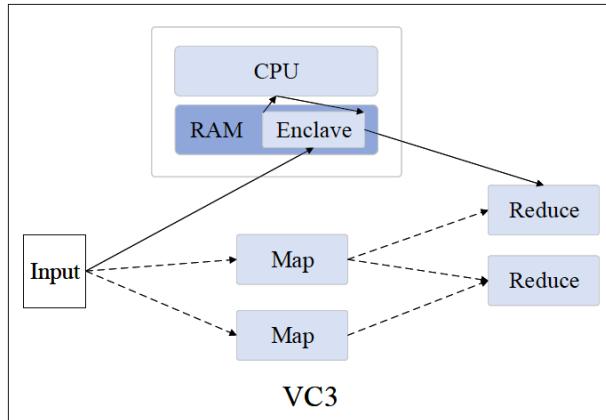


图 3.12 VC3 架构图。

之外都是加密的。VC3 解决了三方面的问题来使得这个数据处理框架具有实用性：首先，为了实现更轻量级的可信计算基，VC3 的设计只将核心的 Map 和 Reduce 函数运行在可信执行环境中；其次，可信执行环境只能保证单个 Map 或 Reduce 任务的完整性，为此 VC3 提出了专门的作业执行协议来保证整个分布式计算的完整性；最后，VC3 提供了新的编译器支持来防止不安全的内存访问。在通用测试基准数据集上的实验显示，相比于原生 Hadoop，实现“写完整性”保护和“读写完整性”保护的 VC3 仅额外产生了 4.5% 和 8% 的运行时开销。

3.2.3 集群中 SGX 资源的管理

可信执行环境技术的实现离不开新的硬件支持。换言之，在支持可信执行环境的计算机上，操作系统管理除传统的 CPU、内存、I/O 等系统资源以外，还需要管理新型的特殊硬件资源。以 Intel SGX 为例，其使用了一种特殊的被称为 EPC（Enclave Page Cache）的加密内存来运行代码和数据。以 EPC 为代表的新型硬件的出现给云原生共享集群的管理带来了挑战。共享集群通过让不同的分布式应用共享底层硬件资源来优化系统管理、提升集群资源利用率。“共享”意味着管理系统需要基于虚拟化等技术手段来对资源进行抽象和隔离，防止不同应用程序在共享硬件资源时因竞争而产生相互干扰，导致性能下降甚至是程序崩溃等问题。

对于传统的 CPU、内存、I/O 等硬件资源，当前的操作系统已具备成熟的机制（如容器）对其进行隔离。共享集群管理系统也依赖于操作系统的这种能力来完成资源分配、变更和调度。但遗憾的是，当前的操作系统虚拟化和隔离可信执行环境所依赖的新型硬件资源的能力还较为有限，这给云原生共享集群在利用这些资源时带来了挑战。以现有的共享集群管理系统为例，YARN 尽管提出了容器的概念用以进行资源划分和分配，但那只是一种逻辑的抽象，在实际运行时，YARN 并没有将任务封装在 LXC、

Docker 之类的操作系统虚拟化容器中。在早期的版本中，YARN 只对内存这单一维度的资源进行限制。当 YARN 启动一个任务时，同时还会启动一个监控器对任务进程的内存消耗进行监控。当任务进程消耗的内存超过任务申请的内存时，YARN 会将该任务终止。而对于如 CPU 之类的其他资源，尽管用户在提交作业时也会为每个任务指定申请的资源配额，但这些信息仅仅会被用来进行调度决策，YARN 不会对这些资源进行限制。因此，一个指定了 1 个 CPU 核数的任务在机器较为空闲时可能会使用超过 1 个 CPU 核数的资源。随着时间的发展，在后续版本中，YARN 引入了 cgroups 机制，通过 cgroups 实现了对 CPU、I/O 等资源的限制。

因为 YARN 启动任务就是启动进程，如果想在 YARN 中运行 SGX 任务，只需在集群节点上安装好 SGX 驱动程序等相关环境即可。但当前的 YARN 缺少接收任务使用 EPC 资源需求的接口，因此 YARN 在调度时不会考虑 SGX 作业需要使用多少 EPC 资源。进一步地，YARN 缺乏隔离 EPC 的能力，因此调度到同一台机器上的两个 SGX 程序会竞争 EPC 使用，导致程序性能下降。

与 YARN 相比，Kubernetes 管理和编排的是以 Docker 为代表的操作系统虚拟化容器。也就是说，在 Kubernetes 中，任务进程都是运行在 Docker 容器里的。对于 SGX 任务而言，如果想要以 Docker 的形式运行，需要将主机上的 *isgx* 设备文件映射到容器里。在 Kubernetes 里运行 SGX 任务则更复杂一些，需要使 Kubernetes 具备管理 EPC 的能力。

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: {POD_NAME}
5    namespace: default
6  spec:
7    containers:
8      - image: {CONTAINER_IMAGE}
9        imagePullPolicy: IfNotPresent
10       name: {CONTAINER_NAME}
11       resources:
12         requests:
13           alibabacloud.com/sgx_epc_MiB: 20
14         limits:
15           alibabacloud.com/sgx_epc_MiB: 20

```

图 3.13 使用 SGX 设备插件在 Kubernetes 中提交任务时编写 yaml 文件。

Kubernetes 提供了一套被称为设备插件（Device Plugin）的机制来接入新型硬件设备，例如 SGX、GPU、FPGA 等。设备插件提供一个 RPC 服务接口向 kubelet 进行注册，发送管理的设备列表。kubelet 是 Kubernetes 运行在每个节点上的代理，它会进一步地将这些硬件资源信息发送到集群主节点上的 API 服务器中。用户在请求使用这些

新型硬件资源时，可以在配置文件 yaml 中指定这些设备的配额。Kubernetes 的调度器在进行调度时会根据用户请求的资源配额以及从 API 服务器获取的实时资源容量进行资源分配和任务放置。

阿里云团队的工程师基于 Kubernetes 的这套机制实现了面向 SGX 的设备插件。该插件在 Kubernetes 中定义了一种被称为 alibabacloud/sgx_epc_MiB 的新资源来表示 EPC 资源。代码片段 3.13 是一个使用该 SGX 设备插件在 Kubernetes 中提交任务的示例。可以看到，用户可以在 yaml 文件中通过 resources:requests/limits 字段来指定容器对 SGX EPC 资源的需求和限制。该设备插件使得 Kubernetes 上的任务可以方便地使用 SGX 资源。但由于操作系统和 SGX 驱动缺乏隔离 EPC 的能力，用户通过 resources:requests/limits 字段指定的数值只会用到 Kubernetes 的资源调度中，而无法像传统的 CPU、内存等资源那样，真正地做到对容器使用 EPC 的大小进行限制。

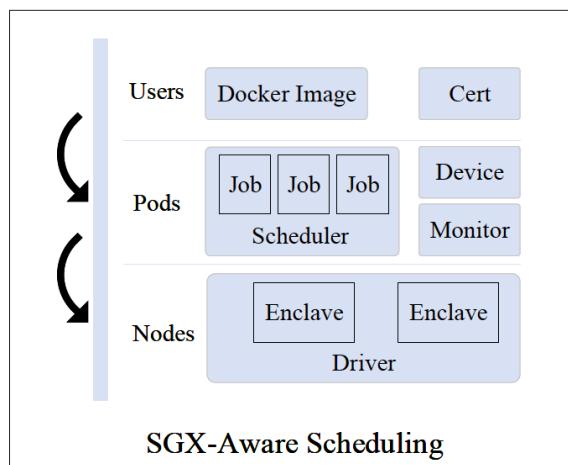


图 3.14 SGX 可感知的调度系统。

Neuchâtel 大学的 Vaucher 等人在这个方向上做了进一步的工作，基于 Kubernetes 实现了 SGX 可感知的集群管理系统。图是 3.14 SGX 可感知的集群管理系统的系统架构，该系统主要在四个方面做出了改进：第一，基于 Kubernetes 的设备插件机制，实现了 SGX 设备的接入。第二，基于节点 EPC 实时使用情况实现 SGX 可感知的容器调度。第三，基于 Kubernetes 的监控模块 Heapster 以及时序数据库 InfluxQL 实现集群对各节点 EPC 使用情况的实时监控。第四，修改 Intel 官方的原生 SGX 驱动程序，实现 EPC 实时使用数据的信息采集以及限制容器使用 EPC 的大小，防止因 EPC 资源竞争而导致程序性能下降。

3.2.4 在机器学习任务中应用 SGX 技术

由于 SGX EPC 大小的限制（可供应用程序使用的仅有不足 100MB），在实际生产环境中很难直接应用于大型机器学习模型的训练，多数情况下 SGX 被用于有隐私需求的模型部署。尽管如此 O. Ohriemenko 等人 [22] 和 R. Kunkel[16] 等人还是在 SGX 环境下的机器学习模型训练做了若干尝试。O. Ohriemenko 等人在 SGX 环境下实现了若干传统的机器学习算法如 K-means, SVM, decision-tree 等，并证实了其性能在一定程度上是可以接受的。

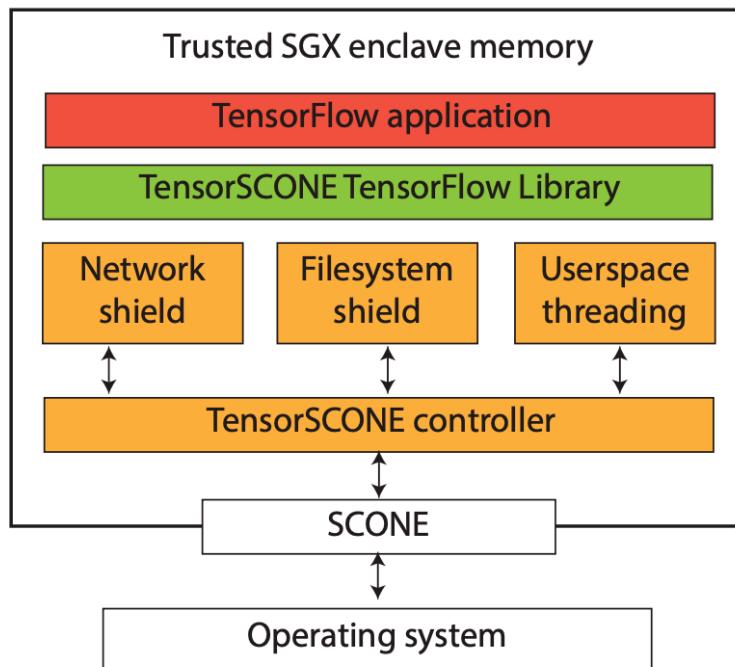


图 3.15 TensorSCONE 架构图。

R. Kunkel 则基于其前序工作 SCONE[3] 实现了支持机器学习任务的安全容器环境 TensorSCONE。图 3.15 阐述了 TensorSCONE 的架构示意图。TensorSCONE 基于的 SCONE 是与 Docker 完全兼容的，且实现了标准的本地认证和远程认证 API。考虑到在 EPC 受限的环境中应该采用尽可能轻量级的框架，其在应用层框架上其选择了较为轻量的机器学习框架 Tensorflow Lite (TF Lite)。其在 SCONE 容器内部实现了一个定制化的 controller，负责重载 TF Lite 的若干功能，并为上层应用提供统一的接口。在其实验中，其仅测试了 ML 任务的模型推断，并称利用 TensorSCONE 框架训练一个 Inception 模型可能需要数月的时间。

J. Ma 在 2020 年提出了 S3ML[19]，基于服务器端的可信执行环境技术 SGX 来实现云端机器学习模型部署的安全可信。图 3.16 表达了 S3ML 的架构。S3ML 从两个方面来提升云原生共享集群对安全可信应用的支持。首先，S3ML 抽象凝练了开发者在

使用可信计算环境技术构建应用中所面临的共性需求，将密钥生成、分发、持久化等必备功能实现成独立的安全服务。依赖于该服务，应用开发的效率可以显著提升。其次，在管理可信执行环境设备方面，S3ML 提出了一种方法，在给定一个服务 SLO 的情况下对服务进行离线性能分析，探索服务延时与硬件资源指标的定量定性关系。进一步地，S3ML 提出新的系统架构与机制，合理利用 SGX 特殊硬件资源的实时监控信息，保障在运行着混合负载、存在资源争用的共享集群里依然能够满足服务 SLO。

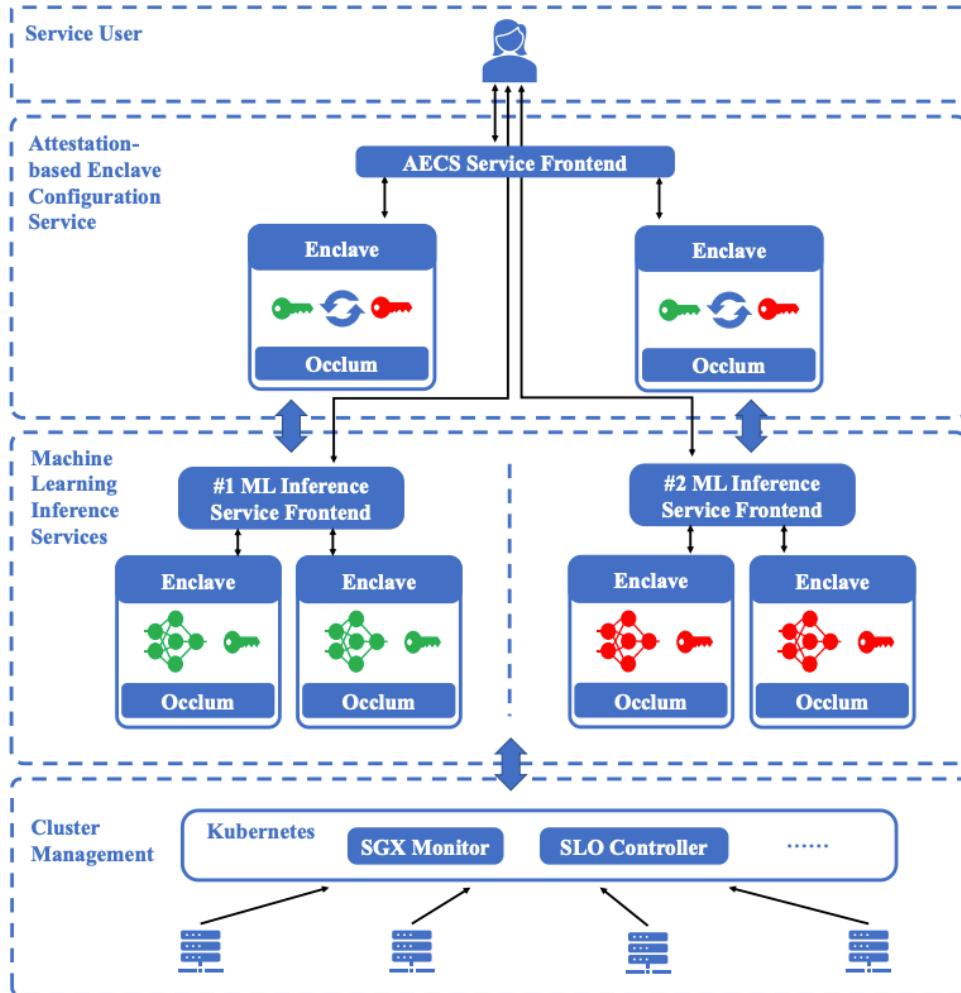


图 3.16 S3ML 架构示意图。

3.3 小结

本章介绍了加速型硬件 GPU 和安全型硬件 SGX 在云环境中为机器学习带来的新的机遇和挑战。对于 GPU 的利用，本章从两个角度展开。第一个是集群环境中以单个 GPU 为最小调度单位的 GPU 集群调度算法，该类算法一般关注 GPU 之间的连接和拓扑结构，旨在实现一定的任务放置策略使得任务性能最大化、集群吞吐率最大化。同

时，结合优先级分级的方法，使得低优先级的任务可以在机会性资源上运行，可以进一步提升集群的资源利用率。另一个角度是多个 ML 任务共用同一个 GPU。从本质上讲，这也是一种 GPU 调度的场景，只不过最小调度单位是部分 GPU。这种场景下，除了传统的 GPU 虚拟化手段，近年来越来越多的研究人员尝试在特定的业务场景下（机器学习模型的训练即为一个典型的场景）实现特定的 GPU 共享机制。SGX 是云环境中支持安全计算的硬件。由于其兴起时间较晚，因此在其上运行机器学习负载的研究还处于探索阶段，未能大规模应用于生产环境。事实上，Intel 计划在不久的未来随着新架构 CPU 的发布对 SGX 技术做同步更新，其所能支持的 EPC 上限可能会大幅度提升，这无疑会给大规模模型的安全训练和部署带来新的潜在解决方案。未来一段时间内，机器学习任务针对 GPU 和 SGX 的研究将持续称为学术界的研究热点。

第四章 人工智能对云计算的增强技术研究

云计算中存在如下两个基本问题。第一，站在用户的视角，如何在浩如烟海的公有云资源池中

4.1 基于机器学习算法的云资源配置优化技术

4.2 基于机器学习算法的云资源调度优化技术

4.3 小结

第五章 重要文献与研究团队总结

第六章 下一步的研究设想

6.1 基于 k8s 的 SLO 可感知的机器学习模型部署系统

6.2 SGX 环境下机器学习模型部署干扰研究

参考文献

- [1] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen *et al.* “*Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics*”. In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. **2017**: 469–482.
- [2] Marcelo Amaral, Jordà Polo, David Carrera *et al.* “*Topology-aware GPU scheduling for learning workloads in cloud environments*”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on*. **2017**: 17.
- [3] Sergei Arnautov, Bohdan Trach, Franz Gregor *et al.* “*SCONE: secure Linux containers with Intel SGX*”. In: *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. **2016**: 689–703.
- [4] Joao Carreira, Pedro Fonseca, Alexey Tumanov *et al.* “*Cirrus: a Serverless Framework for End-to-end ML Workflows*”. In: *Proceedings of the ACM Symposium on Cloud Computing*. **2019**: 13–24.
- [5] Maria Casimiro, Diego Didona, Paolo Romano *et al.* “*Lynceus: Cost-efficient Tuning and Provisioning of Data Analytic Jobs*”. *arXiv: Distributed, Parallel, and Cluster Computing*, **2019**.
- [6] Andrew Chung, Jun Woo Park and Gregory R. Ganger. “*Stratus: cost-aware container scheduling in the public cloud*”. In: *Proceedings of the ACM Symposium on Cloud Computing*. **2018**: 121–134.
- [7] Christina Delimitrou and Christos Kozyrakis. “*Quasar: resource-efficient and QoS-aware cluster management*”. In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. **2014**: 127–144.
- [8] Leila Etaati. “*Azure Machine Learning Studio*”. **2019**: 201–223.
- [9] Lang Feng, Prabhakar Kudva, Dilma Da Silva *et al.* “*Exploring Serverless Computing for Neural Network Training*”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. **2018**: 334–341.
- [10] Aaron Harlap, Andrew Chung, Alexey Tumanov *et al.* “*Tributary: spot-dancing for elastic services with latency SLOs*”. In: *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*. **2018**: 1–14.
- [11] Aaron Harlap, Alexey Tumanov, Andrew Chung *et al.* “*Proteus: agile ML elasticity through tiered reliability in dynamic resource markets*”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. **2017**: 589–604.
- [12] Vatche Ishakian, Vinod Muthusamy and Aleksander Slominski. “*Serving Deep Learning Models in a Serverless Platform*”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. **2018**: 257–262.
- [13] Ameet V Joshi. “*Amazon’s Machine Learning Toolkit: Sagemaker*”. **2020**: 233–243.
- [14] Daeyoun Kang, Tae Joon Jun, Dohyeun Kim *et al.* “*ConVGPU: GPU Management Middleware in Container Based Virtualized Environment*”. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. **2017**: 301–309.

- [15] Ana Klimovic, Heiner Litz and Christos Kozyrakis. “*Selecta: heterogeneous cloud storage configuration for data analytics*”. In: *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*. **2018**: 759–773.
- [16] Roland Kunkel, Do Le Quoc, Franz Gregor *et al.* “*TensorSCONE: A Secure TensorFlow Framework using Intel SGX*.” *arXiv preprint arXiv:1902.04413*, **2019**.
- [17] Yan Li, Bo An, Junming Ma *et al.* “*SpotTune: Leveraging Transient Resources for Cost-efficient Hyper-parameter Tuning in the Public Cloud*”. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. **2020**.
- [18] Edo Liberty, Zohar Karnin, Bing Xiang *et al.* “*Elastic Machine Learning Algorithms in Amazon SageMaker*”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. **2020**: 731–737.
- [19] Junming Ma, Chaofan Yu, Aihui Zhou *et al.* “*S3ML: A Secure Serving System for Machine Learning Inference*.” *arXiv preprint arXiv:2010.06212*, **2020**.
- [20] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan *et al.* “*Learning scheduling algorithms for data processing clusters*”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. **2019**: 270–288.
- [21] Farnaz Moradi, Rolf Stadler and Andreas Johnsson. “*Performance Prediction in Dynamic Clouds using Transfer Learning*”. In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. **2019**: 242–250.
- [22] Olga Ohrimenko, Felix Schuster, Cedric Fournet *et al.* “*Oblivious Multi-Party Machine Learning on Trusted Processors*”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016-08: 619–636. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko>.
- [23] Valerio Perrone, Huibin Shen, Aida Zolic *et al.* “*Amazon SageMaker Automatic Model Tuning: Scalable Black-box Optimization*.” *arXiv preprint arXiv:2012.08489*, **2020**.
- [24] Francisco Romero, Qian Li, Neeraja J. Yadwadkar *et al.* “*INFaaS: A Model-less Inference Serving System*”. *arXiv preprint arXiv:1905.13348*, **2019**.
- [25] Zhucheng Tu, Mengping Li and Jimmy Lin. “*Pay-Per-Request Deployment of Neural Network Models Using Serverless Architectures*”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. **2018**: 6–10.
- [26] Shivaram Venkataraman, Zongheng Yang, Michael Franklin *et al.* “*Ernest: efficient performance prediction for large-scale advanced analytics*”. In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. **2016**: 363–378.
- [27] Hao Wang, Di Niu and Baochun Li. “*Distributed Machine Learning with a Serverless Architecture*”. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. **2019**: 1288–1296.
- [28] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee *et al.* “*Gandiva: introspective cluster scheduling for deep learning*”. In: *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. **2018**: 595–610.

- [29] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez *et al.* “*Selecting the best vm across multiple public clouds: A data-driven performance modeling approach*”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. **2017**: 452–465.
- [30] Peifeng Yu and Mosharaf Chowdhury. “*Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications*”. *Proceedings of Machine Learning and Systems*, **2020**, 2: 98–111.
- [31] Chengliang Zhang, Minchen Yu, Wei Wang *et al.* “*MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving*.” In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. **2019**: 1049–1062.
- [32] Hanyu Zhao, Zhenhua Han, Zhi Yang *et al.* “*HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees*”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. **2020**: 515–532.
- [33] Bingbing Zheng, Li Pan, Shijun Liu *et al.* “*An Online Mechanism for Purchasing IaaS Instances and Scheduling Pleasingly Parallel Jobs in Cloud Computing Environments*”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. **2019**: 35–45.