

学校代码：10284

分 类 号：TP311.5

密 级：公开

U D C：004.41

学 号：

硕 士 学 位 论 文

论 文 题 目 面向 Serverless

深度学习训练的

数据加载技术研究

作 者 姓 名

专 业 名 称 计算机科学与技术

研 究 方 向 软件方法学与自动化

导 师 姓 名

2024 年 5 月 22 日

答辩委员会主席_____

评 阅 人_____

论文答辩日期 2024年5月22日

研究生签名：

导师签名：

Research on Data Loading for Serverless Deep Learning Training

by

Supervised by

A dissertation submitted to
the graduate school of
in partial fulfilment of the requirements for the degree of

MASTER

in

Computer Science and Technology

Department of Computer Science and Technology

May 22, 2024

研究生毕业论文中文摘要首页用纸

毕业论文题目：面向 Serverless 深度学习训练的数据加载技术研究

计算机科学与技术 专业 级硕士生姓名：

指导教师（姓名、职称）：

摘要

随着深度学习和云计算的快速发展，利用 Serverless 平台进行深度学习训练引起了广泛关注。Serverless 深度学习训练能够按需分配 GPU 资源，具有灵活性高、成本效益好的优势，尤其适用于小模型训练和多任务并发训练场景。然而，Serverless 环境下的数据加载停滞问题导致 GPU 出现空闲等待，造成资源浪费，影响训练效率。现有的数据加载方案主要针对基于服务器的训练场景，缺乏对 Serverless 环境存算分离、GPU-CPU 资源分配不均衡等特性的考虑。

为解决上述问题，本文提出了一种面向 Serverless 深度学习训练的分布式数据加载方案 DPFlow，旨在优化数据加载过程，减少 GPU 空闲等待时间，提高资源利用率和训练效率。本文的主要贡献如下：

- 设计了适应 Serverless 环境特点的分布式数据加载模型和声明式编程框架。该模型提供了统一、抽象的描述与优化框架，编程框架支持以声明式方式灵活定制数据加载逻辑。
- 提出了一系列优化方法以解决数据加载问题。引入并发加载和数据预取机制，加速在线预处理过程，掩盖网络通信延迟；设计数据缓存和数据采样策略，利用训练数据访问的局部性，提高数据加载吞吐量和资源利用率。
- 在阿里云 Serverless 平台上部署并评估 DPFlow 原型系统。实验结果表明，DPFlow 能适配多种深度学习任务，确保模型训练收敛。与现有方案相比，DPFlow 将每轮数据加载等待时间缩短 82.89%，训练时间减少 11.9%，GPU 利用率提升到 96.94%。在多任务并发训练场景下，DPFlow 有效提升了数据加载吞吐量和资源利用率，吞吐量比 Pytorch 提高 49%。

关键词：Serverless；深度学习训练；数据加载；分布式计算；编程框架；缓存

研究生毕业论文英文摘要首页用纸

THESES: Research on Data Loading for Serverless Deep Learning Training

SPECIALIZATION: Computer Science and Technology

POSTGRADUATE: _____

MENTOR: _____

ABSTRACT

With the rapid development of deep learning and cloud computing, utilizing Serverless platforms for deep learning training has attracted widespread attention. Serverless deep learning training can allocate GPU resources on demand, offering advantages of high flexibility and cost-effectiveness, making it particularly suitable for small model training and concurrent training scenarios. However, the data loading stall problem in Serverless environments leads to GPU idle waiting, resulting in resource waste and affecting training efficiency. Existing data loading solutions mainly target server-based training scenarios and lack consideration for the separation of storage and computation and the unbalanced allocation of GPU-CPU resources in Serverless environments.

To address the above problems, this paper proposes a distributed data loading scheme called DPFlow for Serverless deep learning training, aiming to optimize the data loading process, reduce GPU idle waiting time, and improve resource utilization and training efficiency. The main contributions of this paper are as follows:

- Designed a distributed data loading model and declarative programming framework adapted to the characteristics of Serverless environments. The model provides a unified and abstract description and optimization framework, and the programming framework supports flexible customization of data loading logic in a declarative manner.
- Proposed a series of optimization methods to solve data loading problems. Introduced concurrent loading and data prefetching mechanisms to accelerate the on-

ABSTRACT

line preprocessing process and mask network communication latency; designed data caching and data sampling strategies to exploit the locality of training data access, improving data loading throughput and resource utilization.

- Deployed and evaluated the DPFlow prototype system on Alibaba Cloud’s Serverless platform. Experimental results show that DPFlow can adapt to various deep learning tasks and ensure model training convergence. Compared with existing schemes, DPFlow reduces the data loading waiting time per round by 82.89%, reduces training time by 11.9%, and increases GPU utilization to 96.94%. In concurrent multi-task training scenarios, DPFlow effectively improves data loading throughput and resource utilization, with throughput increased by 49% compared to Pytorch.

KEYWORDS: Serverless; Deep Learning Training; Data Loading; Distributed Computing; Programming Framework; Cache

目 录

| | |
|---|----|
| 第一章 绪论 | 1 |
| 1.1 研究背景 | 1 |
| 1.2 研究现状 | 2 |
| 1.3 本文工作 | 4 |
| 1.4 本文组织结构 | 6 |
| | |
| 第二章 背景知识 | 7 |
| 2.1 Serverless 计算模式 | 7 |
| 2.2 深度学习训练技术 | 9 |
| 2.2.1 深度学习训练介绍 | 9 |
| 2.2.2 Serverless 深度学习训练技术现状 | 10 |
| 2.3 数据加载技术 | 11 |
| 2.3.1 数据加载概述 | 11 |
| 2.3.2 数据加载面临的问题 | 12 |
| 2.3.3 数据加载技术现状 | 13 |
| 2.4 本章小结 | 16 |
| | |
| 第三章 Serverless 分布式数据加载方法 | 19 |
| 3.1 方法概述 | 19 |
| 3.2 Serverless 分布式数据加载模型 | 21 |
| 3.2.1 模型定义 | 21 |
| 3.2.2 模型的运行时行为 | 26 |
| 3.3 Serverless 分布式数据加载的编程框架 | 31 |
| 3.3.1 静态数据集通用存储规范 | 32 |
| 3.3.2 数据加载过程描述 | 35 |

| | |
|---|-----------|
| 3.4 Serverless 分布式数据加载的优化方法 | 41 |
| 3.4.1 并行加载优化 | 42 |
| 3.4.2 数据预取优化 | 47 |
| 3.4.3 数据缓存优化 | 50 |
| 3.4.4 数据采样优化 | 56 |
| 3.5 本章小结 | 61 |
| 第四章 Serverless 分布式数据加载系统 | 63 |
| 4.1 阿里云 Serverless 产品介绍 | 63 |
| 4.1.1 存储与网络服务 | 64 |
| 4.1.2 函数计算平台（FC） | 65 |
| 4.1.3 弹性容器实例（ECI） | 66 |
| 4.2 系统架构设计 | 67 |
| 4.3 系统组件设计 | 69 |
| 4.3.1 面向高效数据传输的 RPC 模块 | 69 |
| 4.3.2 数据加载协调服务 | 74 |
| 4.3.3 数据加载路由服务 | 76 |
| 4.3.4 数据加载分片服务 | 79 |
| 4.4 本章小结 | 81 |
| 第五章 实验设计与评估 | 83 |
| 5.1 实验配置 | 84 |
| 5.2 核心算法仿真实验 | 84 |
| 5.2.1 缓存置换策略 ALSD 效果评估 | 85 |
| 5.2.2 采样算法 DynaGCS 效果评估 | 87 |
| 5.2.3 采样算法 DynaGCS 随机性质验证 | 88 |
| 5.3 系统功能验证实验 | 90 |
| 5.3.1 深度学习任务的适配性验证 | 90 |
| 5.3.2 训练收敛性的验证 | 92 |
| 5.4 系统性能评估实验 | 92 |
| 5.4.1 数据传输性能评估 | 92 |

| | |
|--------------------------------|------------|
| 5.4.2 数据加载性能评估 | 94 |
| 5.5 本章小结 | 97 |
| 第六章 总结与展望 | 99 |
| 6.1 工作总结 | 99 |
| 6.2 研究展望 | 100 |
| 参考文献 | 101 |
| 致谢（盲审阶段，暂时隐去） | 108 |
| 简历与科研成果 | 109 |

插图目录

| | |
|---|----|
| 2-1 Serverless 异步调用示意图 ^[30] | 8 |
| 2-2 Serverless 云平台层次架构图 ^[24] | 8 |
| 2-3 LambdaML 架构图 ^[1] | 11 |
| 2-4 深度学习训练流程 ^[17] | 12 |
| 2-5 数据停滞问题示意图 ^[42] | 12 |
| 2-6 Ray Data 流程示意图 ^[18] | 14 |
| 2-7 数据分片和并行化处理示意图 ^[3] | 15 |
| 2-8 领域特定缓存优化示意图 | 15 |
| 2-9 多任务协同采样优化示意图 ^[23] | 16 |
| | |
| 3-1 Serverless 分布式数据加载方案 DPFlow 示意图 | 19 |
| 3-2 Serverless 分布式数据加载方法架构示意图 | 21 |
| 3-3 PyTorch 的数据加载模型 | 21 |
| 3-4 逻辑数据加载流与执行数据加载流的关系 | 24 |
| 3-5 模型的运行时行为示意图 | 26 |
| 3-6 模型的数据流转示意图 | 27 |
| 3-7 单任务训练中的执行数据加载流示意图 | 30 |
| 3-8 多任务训练中的执行数据加载流复用示意图 | 31 |
| 3-9 存储规范的语义对象关系图 | 33 |
| 3-10 数据集目录组织结构 | 34 |
| 3-11 DataLoaderReader 接口对象与实现类型 | 38 |
| 3-12 工作窃取任务调度策略示意图 | 43 |
| 3-13 QMOWS 策略示意图 | 44 |
| 3-14 预取机制实现数据加载和模型训练的异步处理 | 48 |
| 3-15 采样序列的采样距离划分示意图 | 52 |

| | |
|---|----|
| 3-16 ALSD 与 LRU 缓存命中示意图 | 56 |
| 3-17 三个任务形成的交集树示意图 | 57 |
| 3-18 DynaGCS 算法在训练过程中的工作流程图 | 60 |
| | |
| 4-1 FC 函数实例挂载 OSS 与 NAS 文件系统 | 65 |
| 4-2 DPFlow 阿里云服务的网络拓扑关系 | 66 |
| 4-3 DPFlow 系统整体架构 | 68 |
| 4-4 CRPC 消息帧格式 | 70 |
| 4-5 CRPC 中 Python 对象的传递过程 | 73 |
| 4-6 数据加载协调服务的结构图 | 74 |
| 4-7 数据加载流生命周期状态转换图 | 75 |
| 4-8 数据加载路由服务的结构图 | 76 |
| 4-9 索引访问模式的时序图 | 78 |
| 4-10 遍历访问模式的时序图 | 79 |
| 4-11 数据加载分片服务的结构图 | 80 |
| 4-12 预取与缓存对象池的结构图 | 80 |
| | |
| 5-1 不同缓存置换策略的命中率对比 | 86 |
| 5-2 ALSD 与 Use Distance 的性能对比 | 87 |
| 5-3 不同采样算法和任务场景的重叠率对比 | 88 |
| 5-4 不同采样算法的随机性检验结果 | 89 |
| 5-5 不同任务上的数据加载日志 | 91 |
| 5-6 不同数据加载方案的训练过程指标对比 | 92 |
| 5-7 不同 RPC 框架的性能对比 | 93 |
| 5-8 不同方案的数据加载等待时间对比 | 94 |
| 5-9 多任务共享的数据加载等待时间对比 | 95 |
| 5-10 多任务场景的数据加载单并发度吞吐量对比 | 95 |
| 5-11 不同方案的训练总时间对比 | 96 |
| 5-12 不同方案的 GPU 利用率对比 | 97 |

表格目录

| | | |
|-----|-----------------------------------|----|
| 3-1 | DataLoadFlow 公开方法 | 37 |
| 3-2 | DataLoadFlowReader 公开方法 | 38 |
| 3-3 | DataLoadFlowRead 公开方法 | 38 |
| 4-1 | 阿里云主要存储服务及功能特点 | 64 |
| 4-2 | 消息帧 Header 结构 | 70 |
| 5-1 | 实验软硬件环境配置 | 85 |
| 5-2 | 采样效果实验场景设置 | 87 |
| 5-3 | DPFlow 对不同深度学习任务的适配情况 | 90 |
| 5-4 | 不同 RPC 框架的数据传输性能对比 | 93 |

第一章 绪论

1.1 研究背景

近年来，随着深度学习技术的快速发展和云计算基础设施的日益完善，利用 Serverless 平台进行深度学习训练引起了广泛关注^[1-8]。Serverless 计算以其自动化的资源管理和调度、按需付费、自动伸缩等独特优势^[9]，为深度学习训练提供了更加灵活、经济、易用的解决方案，特别适用于超参数搜索等多任务并行训练场景。主流云厂商推出的 Serverless 平台，如 AWS Lambda、Google Cloud Functions 和阿里云函数计算等，为 Serverless 深度学习训练提供了基础设施支持^[10]。

与传统的基于服务器的训练模式相比，Serverless 深度学习训练具有多方面优势。首先，Serverless 平台的高弹性计算能力可实现更高的计算并行度，如 SIREN^[4] 利用该特性将模型训练时间减少了 44%。其次，Serverless 平台的按需付费模式降低了深度学习训练的经济负担^[11]，如 MINIONSRL^[12] 在 Serverless 平台进行强化学习训练，相对于基于服务器的训练模式降低了 86% 的成本。此外，Serverless 平台提供了统一的开发部署环境，简化了环境配置和依赖管理，提高了训练任务的可移植性和可复现性，降低了维护难度^[3]。这些优势使得 Serverless 深度学习训练模式在特定场景下的灵活性、成本效益和易用性方面优于基于服务器的训练模式。

然而，目前 Serverless 深度学习训练的相关工作主要关注于模型训练平台的实现和优化训练过程，而在数据加载方面的研究相对较少。事实上，数据加载作为 Serverless 深度学习训练的关键环节，包括数据读取和在线预处理两个阶段^[13]，具有相应的优化空间和研究价值。高效的数据加载可以降低训练时间，提高 GPU 利用率，进一步提升训练效率和降低成本。

现有的 Serverless 深度学习训练在数据加载方面存在诸多挑战。主流深度学习框架如 PyTorch^[14] 的内置数据加载器在 Serverless 环境下存在局限性。首先，

这些加载器通常只支持单机多进程，缺乏分布式并行加载的能力。在 Serverless 环境中，GPU 和 CPU 资源的分配存在不均衡问题。具体而言，每个 GPU 训练函数通常只能使用较少的本地 CPU 资源，导致难以充分利用 Serverless 平台的弹性计算能力^[15]。这种资源分配不均衡的问题限制了数据加载的并行度和效率。其次，现有的数据加载算法没有考虑到 Serverless 存算分离架构带来的挑战。在 Serverless 环境中，训练数据通常存储在远程对象存储（如 S3）而非本地磁盘，导致数据读取的 I/O 开销增加^[16]。此外，复杂的在线预处理（如图像解码、数据增强等）进一步加剧了数据加载的计算负担。上述因素共同导致了数据停滞（Data Stall）问题，即数据加载速度跟不上训练速度，使得 GPU 出现空闲等待，影响了训练效率^[17]。由于 Serverless 平台按需计费，GPU 资源昂贵，充分利用 GPU 对于加快训练速度、降低成本至关重要。

本文聚焦于优化 Serverless 深度学习训练的数据加载过程，主要研究问题包括：

1. 如何利用 Serverless 平台的弹性计算资源，实现高效的分布式数据加载？
2. 如何优化 Serverless 环境下的远程 I/O 和在线预处理开销，降低 GPU 空闲等待时间？
3. 如何在多任务并发训练场景下，提高整体的数据加载吞吐量和资源利用率？

我们认为，开展面向 Serverless 深度学习训练的数据加载技术研究具有重要意义，不仅可以弥补现有 Serverless 深度学习训练研究在数据加载方面的不足，还能够进一步提升 Serverless 深度学习的性能和效率，推动其在更广泛的领域和场景中的应用。

1.2 研究现状

数据加载是深度学习训练过程中的关键环节，高效的数据加载可以提高训练效率，降低训练成本。在工业界和学术界针对数据加载技术进行了研究，提出了一系列数据加载方案以及优化方法。

在工业界，开发人员设计了一系列数据加载工具。Ray Data^[18]基于 Ray^[19]分布式计算框架实现了分布式数据加载。它利用数据分片读取、并行预处理等技术，优化了数据加载效率。然而，Ray Data 需依赖于独立的 Ray 计算集群，目

前无法直接在 Serverless 平台上部署。Huggingface Datasets^[20]利用 PyArrow^[21]技术实现了数据集的零拷贝读取和数据加载缓存优化，但要求数据集预处理成 PyArrow 格式，这提高了使用门槛，也影响到数据集的迭代与共享过程。并且 Huggingface Datasets 采用的本地缓存机制不适合 Serverless 函数的无状态特性。PyTorch DataLoader^[22]支持多进程并行数据加载和预取，但主要适用于单机多核环境，对于 Serverless 分布式环境的支持有限，并且单个 Serverless 函数的 CPU 核心数量限制可能成为性能瓶颈。

在学术界，研究人员提出了多种数据加载方案。Zhao 等人^[16]设计的缓存感知任务调度系统可将训练任务调度至含有所需缓存数据的节点上，减少数据读取开销。但此方法难以直接应用于 Serverless 平台，因为 Serverless 函数的调度不可控。Xie 等人^[23]的 Joader 系统针对同时进行的多并发任务优化了数据加载效率，通过引入 RefCnt 缓存策略和 Joader-DSA 协同采样算法，在多个任务中协调采样过程，提高了训练数据的重叠率。然而，Joader 主要为单机多卡场景设计，不适用于分布式环境，且其数据加载过程在系统中内置，不支持用户动态定义。Mohan 等人^[17]提出的 MinIO 缓存策略和 CoorDL 协同采样算法，MinIO 通过保留任务缓存项来优化缓存命中率，而 CoorDL 提升了多任务同时训练的数据重叠率。但是 MinIO 策略在利用任务信息方面存在优化空间，CoorDL 算法仅适用于多任务同时训练的场景，不能适应异步训练或训练速度不一致的情况。

现有工作提出了多种数据加载方案，在基于服务器的训练模式下取得了不错的效果。然而，Serverless 环境与传统服务器模式存在一些差异，主要体现在：1) 弹性计算资源供应^[24]；2) 无状态函数计算模型^[25]；3) 无法对底层平台进行修改^[9]；4) 函数间通信受限^[26]。这些差异导致现有数据加载方案无法直接应用于 Serverless 深度学习训练。为了充分发挥 Serverless 计算的优势，提高训练效率，需要针对 Serverless 环境的特点，对数据加载技术进行改进。同时，我们也观察到现有工作存在一些方案和算法上的优化空间，将现有工作的不足总结为以下几点：

- **缺乏对 Serverless 环境特性的适配。**现有的数据加载方法主要针对单机或基于服务器的分布式环境设计，未充分考虑和适配 Serverless 平台的弹性计算、无状态函数、无法修改底层平台以及函数通信受限制等特性。这导致现有方法难以直接应用于 Serverless 场景。

- **数据缓存和数据采样策略有待优化。**一些方法在数据缓存和数据采样方面的优化不够深入。一些方法采用简单的本地缓存策略，难以在无状态函数中使用。缓存策略的设计上也有优化空间，未能充分利用任务信息和访问模式。此外，现有的数据采样优化也不够深入，不能很好地适应任务启动时间与训练速度不同的场景。
- **数据加载过程定制灵活性不足。**一些方法将数据加载过程固化在系统内部，不允许用户根据具体需求灵活定义和定制。这种设计降低了用户对数据加载过程的控制力，限制了优化的灵活性。支持用户以声明式方式自由定制数据加载过程并自动优化性能，将提升数据加载的适用性和效率，适应更多的深度学习训练任务。
- **对原始数据集的数据修改成本高。**一些方法需要对原始数据集中的样本数据进行离线转换为专有格式以提高加载效率。然而，现实场景中的训练数据集通常具有规模大的特点，对其进行频繁的离线转换代价较高，难以实时应对数据更新，如对数据集样本进行修正。若不同的下游任务对数据格式有不同的要求，例如数据集还被用于数据分析和可视化等，转换数据格式可能会影响现有工具链的兼容性。因此，数据加载方法应尽量兼容原始数据格式，减少转换成本，提高通用性。

针对以上局限性，需要开展面向 Serverless 深度学习训练场景的数据加载技术研究，设计相应的数据加载方案。方案应该充分考虑和适配 Serverless 环境的特性，在数据缓存、数据采样等方面进行更深入的优化。同时，要为用户提供灵活的数据加载定制能力，支持深度学习中主流的编程语言如 Python 描述数据加载过程。数据加载方案也要尽量兼容原始数据格式，避免因数据转换而产生的额外开销。

1.3 本文工作

为解决上述研究问题，弥补现有工作的不足，本文提出了 Serverless 分布式数据加载方案 DPFlow。首先，本文引入了一个 Serverless 分布式数据加载模型，综合考虑 Serverless 环境的特点，为数据加载过程构建了统一、抽象的描述和优化框架。在此基础上，设计了一套静态通用数据集规范，在不修改原始数据集

的前提下，规范化数据集格式，提高了系统的通用性。此外，还提出了一套基于 Python 的声明式数据加载编程框架，为用户提供了描述数据加载过程的 API，屏蔽了底层分布式系统细节，简化了用户的开发工作，提高了数据加载过程定制的灵活性。

在分布式数据加载模型和编程框架的支撑下，本文提出了一系列针对性的优化方法。首先，为降低远程 I/O 和在线预处理开销，引入了并发加载和数据预取机制。并行加载机制通过动态任务调度，同时执行多个分布式并行的数据加载过程，充分利用 Serverless 弹性计算资源，有效加速了在线预处理过程；数据预取机制掩盖了系统的数据加载和数据传输延迟，进一步提升数据加载效率。针对多任务并发训练场景，设计了数据缓存和数据采样策略，利用训练数据访问的局部性，提高了数据加载吞吐量和资源利用率。其中，近似最大采样距离缓存替换策略（ALSD）充分利用任务信息和访问模式，有效提高了缓存命中率；动态分组协同采样（DynaGCS）算法能够适应任务启动时间与训练速度不同的场景，提高了数据采样的质量和数据共享率。

本文的主要贡献总结如下：

1. **Serverless 分布式数据加载模型。** 针对现有数据加载方案难以直接应用于 Serverless 环境的问题，本文提出了一个 Serverless 分布式数据加载模型，综合考虑了 Serverless 环境的特点，为 Serverless 环境下深度学习训练的数据加载过程提供了统一、抽象的描述和优化框架。
2. **声明式数据加载编程框架。** 针对现有工作在数据加载过程定制灵活性方面的不足，本文设计了声明式的数据加载编程框架。该框架定义了通用的静态数据集存储规范，允许在不修改原始数据集的情况下接入 DPFlow。编程框架使用 Python 实现，提供了丰富的语义对象和接口，使用户能够以声明式方式描述数据加载逻辑，提高了数据加载过程定制的灵活性。
3. **数据加载优化方法。** 针对 Serverless 环境下数据加载效率低、资源利用率不足等问题，本文提出了一系列优化方法，包括并行数据加载、数据预取、数据缓存和数据采样等。其中，ALSD 缓存替换策略优化了现有数据缓存策略，将缓存命中率提高了 26.65 至 19.55 个百分点；DynaGCS 采样算法改进了现有数据采样算法，相较于 Joader 将任务间的数据重叠率提升了 2 倍以上，并生成了随机性更好的采样序列。

4. **DPFlow 原型系统。**为验证所提出的 Serverless 分布式数据加载模型和优化方法的有效性，本文实现了 DPFlow 原型系统。该系统在不修改 Serverless 平台的前提下，实现了功能完备的分布式数据加载，突出了其适配 Serverless 环境特性优势。实验结果表明，DPFlow 能适配多种任务，有效解决了 Serverless 环境下的数据加载问题。相比 PyTorch 内置的数据加载框架，DPFlow 平均缩短了 82.9% 的数据加载等待时间，减少了 11.9% 的训练时间，GPU 利用率提升到 96.94%。在两任务并发训练场景下，DPFlow 的数据加载吞吐量提高了 49%。这些结果证明了本文方案的有效性。

1.4 本文组织结构

本文按照以下结构组织各章节内容：

第二章为背景知识，介绍了 Serverless 计算模式、Serverless 深度学习训练的现状、数据加载的基本概念以及现有的数据加载技术和优化策略。阐述了深度学习场景下数据加载所面临的需求和挑战，以及现有工作是如何解决的。

第三章详细阐述本文提出的 Serverless 分布式数据加载方法。首先给出方法的总体设计，然后依次介绍 Serverless 分布式数据加载模型、数据加载编程框架的设计，以及本文提出的几种关键的数据加载优化方法，包括并行加载、数据预取、数据缓存和数据采样。

第四章介绍了基于第三章所述方法实现的 DPFlow 系统。首先说明了 DPFlow 系统实现所依赖的阿里云 Serverless 产品，包括对象存储 OSS、函数计算平台 FC、弹性容器实例 ECI 等。然后详细阐述了 DPFlow 系统的整体架构，并介绍系统组件数据加载协调服务、数据加载路由服务、数据加载分片服务的设计与实现。

第五章对 DPFlow 系统进行了全面的实验评估。在说明实验设计思路和评估目标后，首先通过仿真实验分析了核心算法的理论性能，并与相关工作进行了比较。然后在真实系统环境中验证了 DPFlow 的功能正确性，包括适配不同深度学习任务的能力、端到端训练收敛性等。最后通过性能评估实验，测试了 DPFlow 在数据传输效率、端到端训练性能等关键指标上的表现。

第六章总结全文工作，并对未来研究方向进行了展望。

第二章 背景知识

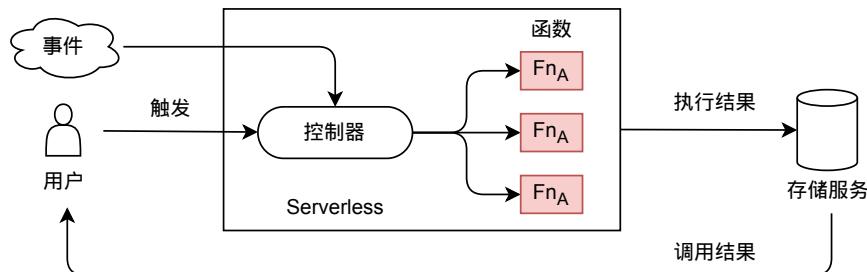
本章介绍了 Serverless 计算、Serverless 深度学习训练和数据加载三个方面的背景知识。首先，介绍了 Serverless 计算的基本概念。其次，概述了深度学习训练的流程，总结了 Serverless 深度学习训练领域的研究进展。最后，分析了数据加载的重要性、面临的问题以及业界的主流方案。

2.1 Serverless 计算模式

Serverless 计算是近年来云计算领域崛起的一种新型计算模式。它以函数即服务（Function-as-a-Service, FaaS）和后端即服务（Backend-as-a-service, BaaS）为基础，为开发者提供完全托管的计算服务，无需关注底层服务器资源的配置和管理^[9,27]。自 2014 年 AWS 推出 Lambda 服务以来^[28]，Serverless 计算引起了学术界和工业界的广泛关注，成为云计算领域的研究热点之一^[25]。

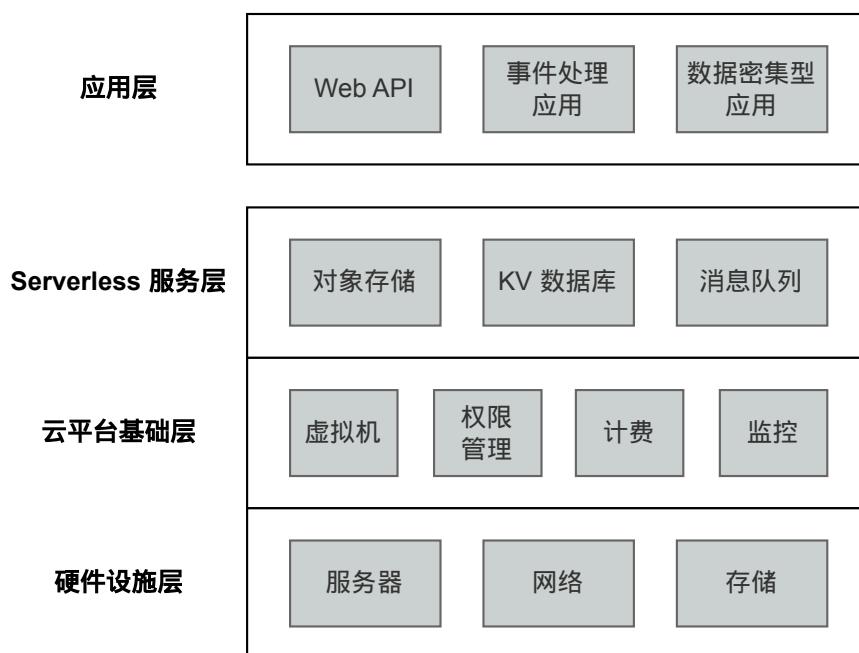
Serverless 计算提供了一种新的应用开发范式，开发者只需专注于编写核心业务逻辑，而无需关心底层基础设施的配置、管理和扩展^[9]。我们以 Serverless 异步调用为例描述 Serverless 中的计算过程，如图2-1所示。首先，平台接收到用户发起或者外部的事件触发，如 HTTP 请求、消息队列、定时器等；接着，控制器对事件进行验证和调度，创建或复用函数实例，并在隔离的执行环境中运行函数代码；最后，函数通过访问云上存储服务（如对象存储）来读写持久化数据，并将调用结果返回给用户^[25]。得益于云平台的自动伸缩能力，Serverless 应用可以根据请求量动态调整资源配置，从而提供高弹性、低成本的服务^[29]。

Serverless 计算的发展离不开云平台的支持，Serverless 平台为开发者提供了一系列高度抽象的云服务，开发者可以基于这些服务快速构建和部署应用^[27]。图2-2展示了 Serverless 云平台的层次架构，从下至上依次为硬件层、基础云平台层、Serverless 云服务层和应用层，抽象级别逐层递增^[24]。Serverless 云服务层屏蔽了底层硬件和基础设施的复杂性，为上层应用提供了更加简单易用的开发接

图 2-1 Serverless 异步调用示意图^[30]

口和运行环境。

Serverless 计算在实践中已经被广泛应用于各种场景和工作负载。一类典型的应用是事件驱动的 Web 服务，如 REST API、物联网后端、实时数据处理等^[29]。开发者可以将请求处理逻辑编写成函数，配置 API 网关触发函数执行，平台会根据请求数量自动扩缩容，从而构建高度弹性、低成本的 Web 服务后端^[27]。另一类常见的工作负载是数据密集型任务，如数据 ETL^[31]、视频解码、深度学习推理和训练等。开发者可将任务拆分为一系列函数，利用平台协调函数执行和数据流转，实现自动调度和函数实例的弹性伸缩，从而高效、经济地完成数据处理工作^[32]。

图 2-2 Serverless 云平台层次架构图^[24]

本研究侧重于 Serverless 下的深度学习训练应用，下面将对深度学习训练和 Serverless 中的深度学习训练技术进行介绍。

2.2 深度学习训练技术

2.2.1 深度学习训练介绍

深度学习是人工智能领域的重要分支，利用多层神经网络对复杂任务进行端到端建模和学习。以监督性学习为例，深度学习模型通过训练数据集学习特征表示和决策函数，不断优化模型参数，最小化预测输出与真实标签之间的误差^[33]。相比传统的机器学习方法，深度学习凭借强大的表示学习能力和端到端建模能力，在图像分类、语音识别、自然语言处理等任务上取得了较多成果^[34]。

深度学习训练是指通过数据驱动的方式，优化深度学习模型的参数，使其能够很好地拟合训练数据集，并具有良好的泛化性能。训练过程通常采用随机梯度下降（Stochastic Gradient Descent，SGD）算法^[35]，将训练数据集划分为小批量（mini-batch），在每个 mini-batch 上计算损失函数关于模型参数的梯度，然后沿梯度反方向更新模型参数，多轮迭代直至模型收敛，同时在 mini-batch 中引入随机性采样以提升模型的泛化能力。

根据执行平台的不同，深度学习训练可分为两类：基于服务器的深度学习训练和 Serverless 深度学习训练^[12,36]。

基于服务器的深度学习训练（Server-based Deep Learning Training）。简称为基于服务器的训练，通常在专用的 GPU 集群中进行。训练集群由多个 GPU 服务器通过高速互联网络组成，训练任务以数据并行或模型并行的方式分布执行，充分利用多 GPU 并行计算能力加速训练进程。主流的深度学习框架如 TensorFlow^[37]、PyTorch^[14]等提供了分布式训练的接口和运行时支持，简化了多机多卡训练的实现。然而，构建和维护专用 GPU 集群需要高昂的前期投入和持续的运维成本，难以灵活应对动态变化的训练需求，资源利用率也有待进一步提升。

Serverless 深度学习训练（Serverless Deep Learning Training）。简称为 Serverless 训练，利用 Serverless 计算平台提供的弹性资源和托管服务，按需分配 GPU 资源，自动扩缩容以适应动态变化的训练负载，降低了深度学习训练的使用门槛和管理成本。例如提供了基于 Serverless 的训练接口和工作流支持，用户无需关注底层资源管理，即可方便地启动分布式训练任务。

2.2.2 Serverless 深度学习训练技术现状

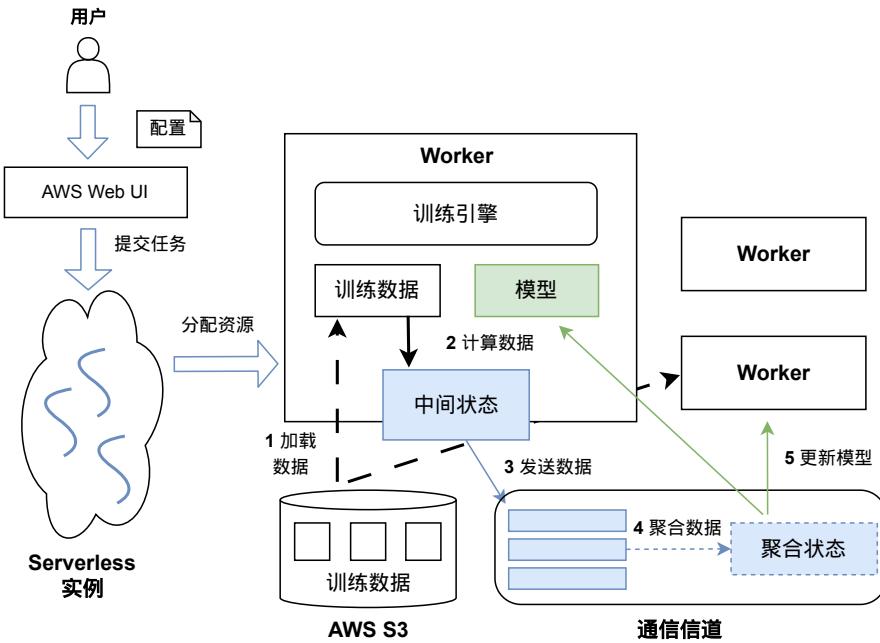
Serverless 计算模式为深度学习训练提供了一种新的思路。其自动伸缩、按需计费等特性与深度学习训练的资源需求较为契合。例如，通过弹性资源供给能够快速启动大量训练实例，充分利用云端弹性资源池，尤其适合超参搜索等高度并发的场景^[6]。按需计费让用户仅为实际消耗的资源付费。相比预留专用集群，Serverless 模式能够更灵活地应对训练负载的波动，减少资源浪费，从而降低总体训练成本^[38]。

然而，Serverless 深度学习训练也存在一些挑战。例如，主流深度学习框架如 Pytorch^[14] 和 Tensorflow^[37] 对 Serverless 平台的支持有限，需要进行适配工作。其次，Serverless 平台不允许函数之间直接进行通信^[26]，这对于分布式训练等需要大量梯度同步的场景不友好，会显著地降低训练速度。考虑到 Serverless 带来的机遇与挑战，学术界和工业界近年来围绕 Serverless 深度学习训练开展了一系列探索，包括设计 Serverless 训练平台，以及优化 Serverless 平台上的训练过程。

在训练平台方面，LambdaML^[1]是一个代表性的工作。LambdaML 基于 Amazon Lambda 构建 Serverless 深度学习训练系统，图2-3展示了 LambdaML 的架构。用户通过配置文件提交训练任务，一个训练任务会通过平台进行资源分配，创建多个训练函数协同完成分布式训练过程。LambdaML 在训练函数中实现了训练引擎编程序库，其负责从 S3 中读取训练数据，在模型中进行前向传播过程，通过通信信道向其他训练函数发送梯度信息，并进行模型更新，完成分布式训练的过程。

在训练优化方面，FuncPipe^[39]是一个面向 Serverless 平台的模型并行训练框架。FuncPipe 通过利用模型分区技术，将完整模型切分成多个子模型进行并行训练，以解决 Serverless 函数在内存和带宽方面的限制，实现高效低成本的分布式深度学习训练。FuncPipe 需要解决模型分区粒度选择和资源配置优化两个问题。为此，提出了一种混合整数二次规划的联合优化方法，可以同时确定最优的模型分区方案和资源配置策略。FuncPipe 还设计了一种分散归并流水线同步新算法，通过并行化 Serverless 函数的上传和下载操作，充分利用了函数的双向网络带宽。

除此之外，Serverless 深度学习训练还存在很多研究问题，例如数据加载优

图 2-3 LambdaML 架构图^[1]

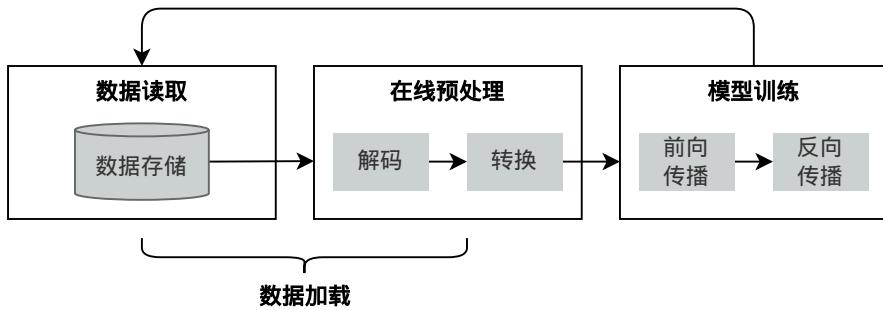
化、资源分配优化、提高通信效率等^[40]。本文关注到数据加载技术在 Serverless 深度学习训练的重要性，并针对这一方向进行相关研究工作，同时能够与现有 Serverless 深度学习训练相关工作形成互补。

2.3 数据加载技术

2.3.1 数据加载概述

深度学习训练流程涵盖数据读取、数据在线预处理和模型训练三个阶段^[41]，图2-4展示了这三个阶段的关系。其中数据读取和数据在线预处理共同构成数据加载（Data Loading）环节，是深度学习训练流程的初始阶段。数据读取的过程包括将训练所需的数据从磁盘、分布式文件系统、对象存储等数据存储服务中读入内存，为后续操作做好准备。数据在线预处理则指的是在运行时对数据执行动态处理，包括但不限于图像解码、数据增强、数据格式转换等操作，以将原始数据转换成适合模型训练的格式^[13]。

数据加载对整个深度学习训练流程的很重要。一方面，它为模型训练提供所需的输入数据；另一方面，数据加载的速度直接影响训练流程的总体时间，较低的加载速度可能导致 GPU 利用率下降。考虑到 GPU 等计算资源相较于 CPU、

图 2-4 深度学习训练流程^[17]

存储和网络设施等其他训练硬件的成本较高，提高数据加载效率对于提升深度学习训练的效率、降低训练成本具有重要的意义。

2.3.2 数据加载面临的问题

数据加载在实践中面临着多方面的挑战，其中最为突出的是数据停滞（Data Stall）问题^[17]。数据停滞问题指的是数据加载速度跟不上模型训练速度，导致 GPU 频繁出现空闲等待，从而降低了训练效率。这一问题在不同深度学习的执行平台中普遍存在，包括基于服务器的训练模式和 Serverless 训练模式。数据停滞也是本文重点关注和致力解决的研究问题。

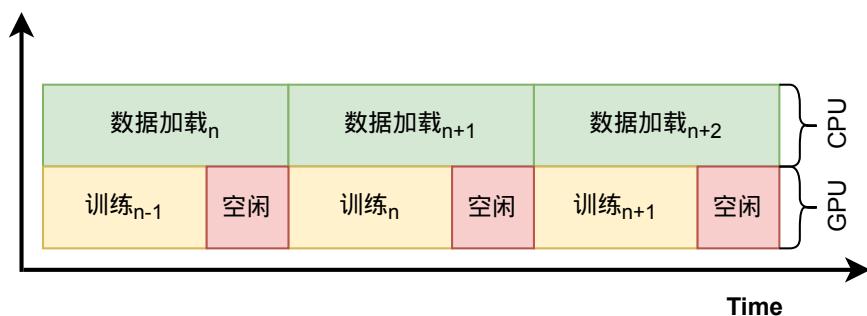
图 2-5 数据停滞问题示意图^[42]

图2-5展示了数据停滞问题的示意图。在深度学习训练中，数据加载和模型训练通常是异步进行的，分别在 CPU 和 GPU 上执行。理想情况下，数据加载速度应该大于模型训练速度，保持连贯的训练过程。但当数据加载速度跟不上模型训练速度时，就会出现 GPU 空闲等待的情况。这种情况下，GPU 利用率会显著下降，造成计算资源的浪费，拖慢整体训练效率^[42]。造成数据停滞的原因可总结为以下几点：

训练数据规模庞大。深度学习训练通常需要在 GB 至 TB 级别的数据上进行训练，将如此大规模数据集完全加载至内存的代价很高，因此需要设计高效的数据缓存和加载策略，在内存开销和数据供给效率之间寻求平衡^[16]。

训练数据的小文件特征。训练样本往往以大量小文件的形式存储，例如 ImageNet 数据集包含了数百万张图片。频繁读取小文件会导致存储系统的访问效率低下，影响数据加载性能^[43-44]。

存算分离架构下的访问开销。为了提高资源利用率和灵活性，许多深度学习平台尤其是 Serverless 训练平台采用了存储计算分离的架构，将训练数据存储在远程服务上。这种架构不可避免地引入了访问和网络开销，使得数据加载面临更大挑战。高效的数据压缩、传输优化和缓存机制有助于缓解这一问题^[16]。

在线预处理的计算开销。对于图像、视频等非结构化数据，在输入模型之前通常需要经过解码、数据增强等一系列预处理操作。这些在线转换操作十分耗时，Mohan 等人^[17]的研究结果显示，预处理的开销可能占据整个数据加载时间的 90% 以上，是制约数据加载速度的关键因素。采用数据缓存、并行化处理等优化手段可以有效降低预处理开销。

2.3.3 数据加载技术现状

针对深度学习数据加载所面临的问题和挑战，工业界和学术界提供了一系列数据加载方案和优化方法。这些方案从不同角度入手，扩展了数据加载的功能，提高数据加载效率，以提高 GPU 等计算资源利用率，加速深度学习训练过程。

工业界数据加载方案

工业界主要致力于开发易用、高效、可扩展的数据加载库和工具。Pytorch DataLoader^[22] 是业界流行深度学习框架 Pytorch^[14] 内置的数据加载方案。Pytorch DataLoader 设计了一套用于实现数据加载过程的编程框架，为用户提供了一系列 Python API，并通过编程框架提供多进程并行化功能以加速数据加载过程。Pytorch DataLoader 的好处在于使用简单，与深度学习框架的集成性很好，缺点在于它的性能受限于单个节点的计算和存储能力。

为了突破这一限制，Ray Data^[18]基于 Ray 分布式计算框架^[19]的任务调度和状态管理功能，实现了可伸缩的分布式数据加载。图2-6展示了 Ray Data 的整体流程。Ray Data 支持从多种数据处理系统中获取数据，然后在 Ray 集群中使用 Ray Data 库进行数据的读取和预处理操作，之后在多种深度学习框架下训练。Ray Data 的缺点是依赖于 Ray 计算集群，而 Ray 计算集群目前无法在 Serverless 平台上部署，意味着 Serverless 深度学习训练无法使用这类工具。

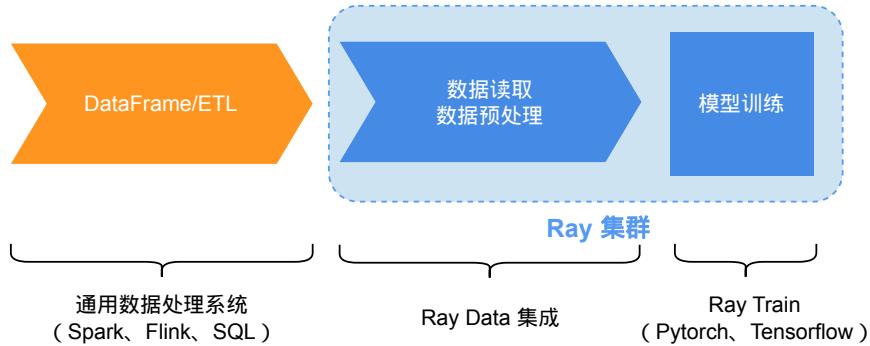


图 2-6 Ray Data 流程示意图^[18]

学术界数据加载方案

学术界则更专注于算法上的创新以及实现应用这些算法的系统。总的来看，相关工作设计了多种优化上的方法，例如数据分片与并行处理、领域特定缓存和多任务协同采样等等，多种方法配合系统上的设计，协同完成数据加载的优化。下面针对不同的优化方法对学术界提出的数据加载方案进行介绍。

数据分片和并行化处理。该优化方法通过对数据集进行分片，并行在多个节点或进程中并行处理，能够有效提高数据加载效率。该优化方法有效地利用了数据加载任务的并行特征，是一个比较直观的优化方法，如图2-7所示，多个节点可以并行读取分片数据集，并交给模型进行训练。Cerebro^[45]采用了一种称为 Model Hopper Parallelism (MOP) 的并行化训练方法。在 MOP 中，训练数据被分片到多个 worker 上，每个 worker 访问分片的顺序遵循不同的伪随机排列。在实现上，Cerebro 使用了轻量级的运行时来管理分片数据在 worker 之间的高效传输。此外，一些工作在 Serverless 环境中实现了类似的优化方法。例如，Cirrus^[3]在 Serverless 平台上实现了端到端的深度学习训练平台，它将训练数据集划分成许多大小相似的分片，并上传到 S3 等云存储上。在训练时，每个 worker 只需要从

S3 中读取和缓存其中一个分片，而不是复制整个数据集，这样可以降低内存占用和通信开销。

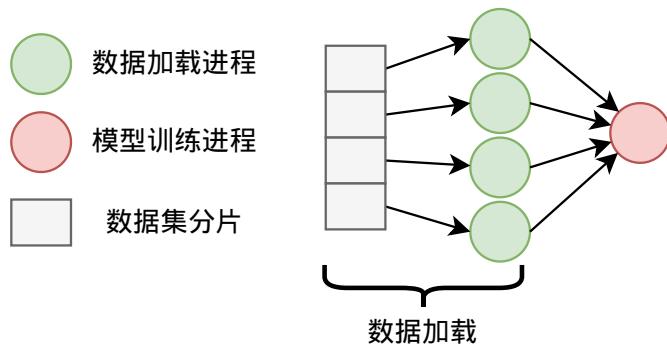


图 2-7 数据分片和并行化处理示意图^[3]

领域特定缓存。有效利用数据缓存可以减少重复的 I/O 操作和预处理计算，从而提高数据加载性能，如图所示2-8，在遍历数据集过程中，缓存未命中时需经过数据加载过程，而缓存命中时则可直接从缓存获取相应数据，避免了数据加载开销。Quiver^[46]利用安全的哈希寻址机制，在多个训练任务之间透明地复用缓存数据，实现了高效的分布式缓存系统。此外，一些工作还提出了针对深度学习的特定缓存策略，充分利用了深度学习任务的数据访问特征。例如，Mohan 等人^[17]提出的 MinIO 缓存策略，通过对已读取的缓存项采取不淘汰策略，避免了缓存抖动，有效提高了缓存命中率；Xie 等人^[23]提出的 RefCnt 缓存策略，通过追踪缓存项在不同训练任务之间的引用计数，优先淘汰引用计数小的缓存项，能够在多个任务共享缓存时，保留更多任务需要的数据。这些领域特定的缓存和预取策略能够充分利用深度学习任务的数据访问特征，提高数据加载性能。

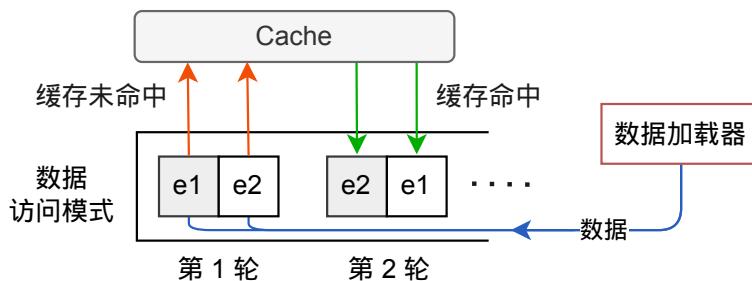


图 2-8 领域特定缓存优化示意图

多任务协同采样。当多个训练任务共用相同的数据集和预处理流程时，通过协调这些任务的采样过程，可以实现数据加载的优化，减少冗余的数据读取

和预处理操作，如图2-9，协同采样的目标是最大化在同一轮数据遍历中，采样到相同元素的概率。例如，CoorDL^[17]通过将一份随机采样序列分发给多个并发任务，使得多个任务能够复用相同数据的加载结果，减少了冗余的数据读取和预处理。Joader^[23]则进一步提出了一种依赖采样算法（DSA），通过精心设计任务间的采样策略，在保证采样随机性的同时最大化了任务间采样数据的重叠性，提高了数据的局部性，从而减少了数据加载开销。这些多任务协同采样技术能够有效挖掘并发训练任务的数据共享机会，降低总体的数据加载开销。

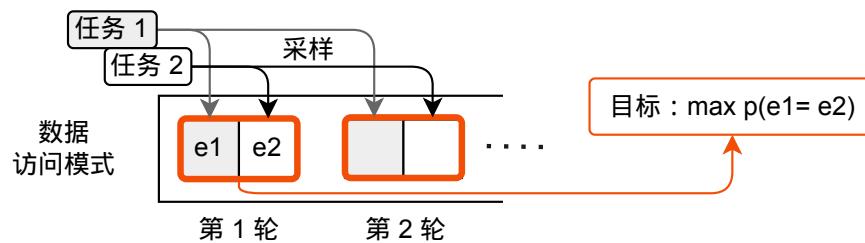


图 2-9 多任务协同采样优化示意图^[23]

相关工作主要针对基于服务器的深度学习训练模式，为数据加载系统设计和优化提供了重要参考和启示。然而，Serverless 环境与传统服务器环境存在差异，导致现有数据加载方案可能难以直接适用于 Serverless 训练场景。尽管如此，已有的系统设计思路和优化方法对 Serverless 深度学习训练下的数据加载方案仍有重要借鉴意义。本文将在吸收现有方案优点的基础上，针对 Serverless 平台的特性，设计更加适用的数据加载框架、系统和优化方法。同时，本文也将研究和改进数据缓存与协同采样策略，并与现有工作设计的 MinIO、RefCnt、CoorDL 与 Joader 算法进行性能对比。最后，实现完整的 Serverless 深度学习数据加载系统，与 Pytorch DataLoader 进行对比分析。

2.4 本章小结

本章系统介绍了 Serverless 计算、深度学习训练和数据加载等相关背景知识。Serverless 计算为深度学习训练带来机遇和挑战，现有工作针对训练平台和训练优化方向进行研究。之后，介绍了数据加载是深度学习训练的关键，面临数据规模庞大、小文件多、存算分离访问开销大、预处理开销高等问题。最后，对工业界和学术界的数据加载方案进行介绍，说明了相关工作为本文研究带来了启发，

通过设计高效的编程框架和数据加载系统，并结合多种优化方法，有利于解决 Serverless 深度学习数据加载面临的挑战。

第三章 Serverless 分布式数据加载方法

本章重点介绍 Serverless 分布式数据加载方法, 该方法由三部分构成: Serverless 分布式数据加载模型、配套的编程框架以及一系列优化方法。首先, 本章概述了该方法的整体架构, 并在此基础上分别详细介绍了数据加载模型、编程框架的设计理念和具体实现, 最后, 对各项优化方法进行详细介绍。

3.1 方法概述

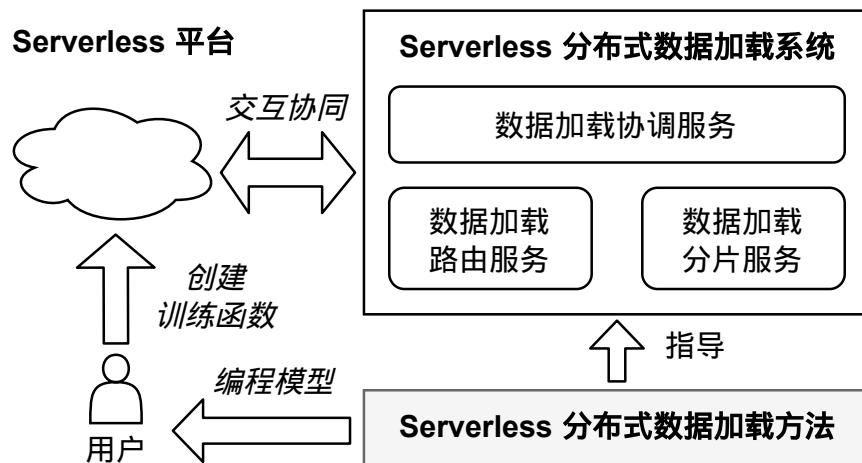


图 3-1 Serverless 分布式数据加载方案 DPFlow 示意图

图 3-1 展示了本文提出的 Serverless 分布式数据加载方案 DPFlow, 该方案涉及用户、Serverless 平台、Serverless 分布式数据加载方法和系统等多个角色。在运行时, 用户利用编程模型编写训练函数代码, 并在 Serverless 平台上创建函数, 系统会与 Serverless 平台持续交互协同, 完成数据加载功能。其中, 较为重要的部分是 Serverless 分布式数据加载方法, 方法从高层次描述了数据加载方案以及相应算法的设计, 为 DPFlow 提供规范化描述, 并指导系统实现。

图3-2展示了方法的基本架构, 方法引入了一个 Serverless 分布式数据加载模型, 以统一、抽象的方式定义了 Serverless 环境下数据加载的基本流程。在此基础上, 给出了明确的优化目标, 即提高数据加载速度, 提升 GPU 资源利用率。

围绕这些目标，本文设计了一系列优化方法，以提升数据加载性能，改善训练体验。具体而言，Serverless 分布式数据加载方法主要包括以下内容：

1. **Serverless 分布式数据加载模型。**本文设计了一种形式化的 Serverless 分布式数据加载模型，明确定义了模型中的关键组件，如逻辑数据加载流、执行数据加载流、数据加载流服务、数据集、数据加载函数和训练函数等，并阐述了它们之间的交互关系。该模型为后续的编程框架设计和优化技术选择提供了理论依据。
2. **Serverless 分布式数据加载的编程框架。**为方便用户使用前述模型，本文设计了配套的编程框架。该框架首先定义了一套静态数据集的通用存储规范，引入了数据集存储、数据集、分片化数据集元信息等语义对象，并给出了标准化的目录组织结构和统一数据集读取算法，以减少不同数据集在组织和访问形式上的差异。在此基础上，框架提供了 DataLoadFlow、DataLoadFlowReader 和 DataLoadFlowRead 等核心语义对象，允许用户以声明式的方式描述端到端的数据加载流程。
3. **Serverless 分布式数据加载的优化方法。**本文针对并行加载、数据预取、数据缓存和数据采样等关键环节，设计了一系列优化方法。在并行加载方面，提出了一种基于工作窃取思想的动态任务调度策略，充分利用 Serverless 平台的计算资源。在数据预取方面，采用了二级数据预取机制，分别在训练函数端和数据加载流服务进行数据预取，以掩盖数据加载和传输的延迟。在数据缓存方面，提出了近似最大采样距离（ALSD）的缓存置换策略，根据数据样本在采样序列中的位置，估算其在未来一段时间内被访问的概率，并以此作为缓存优先级依据。在数据采样方面，提出了动态交集树采样算法 DynaGCS，通过自适应地捕捉和利用多个并发任务之间的数据交集，指导数据采样的优先级，实现高效的数据共享和计算复用。

本文从数据加载建模、编程框架、数据加载优化等多个层面，全面探讨了如何在 Serverless 环境下高效实现深度学习训练的数据加载功能。所提出的 Serverless 分布式数据加载模型和优化方法，不仅能够简化用户描述分布式数据加载的过程，实现分布式数据加载任务，也提升数据加载场景的资源利用率和处理速度，缓解 Serverless 训练模式下数据加载的性能问题。



图 3-2 Serverless 分布式数据加载方法架构示意图

3.2 Serverless 分布式数据加载模型

3.2.1 模型定义

在经典的深度学习训练框架中，常采用单节点多进程数据加载模型。该模型通过在与 GPU 相同的节点上分配多个并行的数据加载进程，以充分利用 GPU 节点上的 CPU 计算能力，从而提高预处理速度。以 PyTorch 的 DataLoader 为例，其基本工作方式如图3-3所示，DataLoader 会创建多个 worker 进程，将数据加载任务分配到这些进程中并行执行，并通过队列在进程间传递数据加载结果，最终由主进程批量获取处理后的训练数据^[47]。

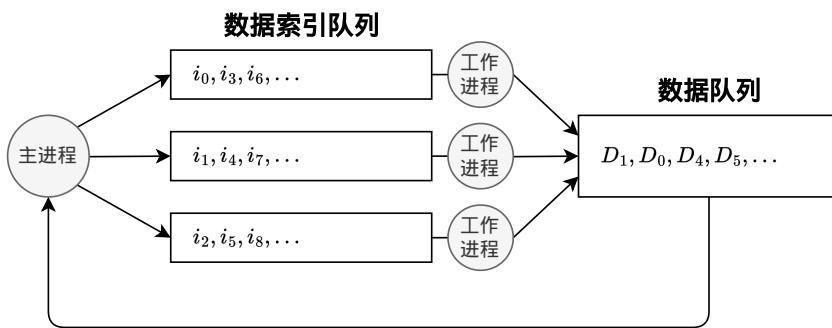


图 3-3 PyTorch 的数据加载模型

然而，这种单节点多进程模型存在一些局限性，难以直接应用于 Serverless 场景中。一方面，其并行度受限于单台机器的 CPU 核数，难以无限制地扩展。当多

个训练任务共享同一台机器时，资源竞争更易成为瓶颈。另一方面，在 Serverless 训练场景下，由于云平台的调度限制，单个 GPU 函数实例往往只能绑定最多 8 个 CPU 核心，这进一步限制了本地并行预处理的能力。此外，Serverless 环境与传统服务器模式还存在其他差异，如弹性计算资源供应、无状态函数计算模型、无法对底层平台进行修改以及函数间通信受限等特点。这些差异导致现有的数据加载方法难以直接应用于 Serverless 场景。

针对上述局限性，本文提出了 Serverless 分布式数据加载模型，目的在于针对 Serverless 环境的特点，设计一种新的数据加载模型，以充分发挥 Serverless 计算的优势，提高训练效率。该模型的目标是在 Serverless 平台上利用函数、容器等资源，实现高并行度的计算，突破单节点的 CPU 限制，实现数据加载过程的分布式化，充分利用 Serverless 平台的弹性计算能力。

同时，分布式数据加载模型规范化定义了数据加载的基本要素以及相应的行为特征。该模型为后续的编程框架设计和优化技术选择提供了理论依据。通过对模型中各个组件的职责、交互方式进行形式化定义，可以指导编程框架的设计和实现，确保框架的正确性和可靠性。而模型中的数据加载流、执行流映射等机制，也为数据加载过程提供了优化基础，计算并行、数据预取、缓存复用、采样等技术都在模型的基础上进行设计和改进。

下面给出 Serverless 分布式数据加载模型的形式化定义：

定义 3.1 (Serverless 分布式数据加载模型) Serverless 分布式数据加载模型可表示为一个 6 元组：

$$M = (L, S, D, \mathcal{F}, \mathcal{E}, \mathcal{T})$$

其中：

- L 表示逻辑数据加载流，每个逻辑数据加载流 L 是一个有向无环图 (DAG)，节点表示数据处理步骤，边表示数据依赖关系。根节点为数据集。
- S 表示数据加载流服务，数据加载流服务 S 是一个 Serverless 服务，关联一个逻辑数据加载流，用于连接和调度执行数据加载流。
- D 表示数据集。
- \mathcal{F} 表示数据加载函数的集合，每个数据加载函数 $f \in \mathcal{F}$ 对应一个 Serverless 函数，用于读取和预处理数据。

- \mathcal{E} 表示执行数据加载流的集合，每个执行数据加载流 $e \in \mathcal{E}$ 与数据加载函数一一绑定，代表该流由哪个函数负责加载。
- \mathcal{T} 表示训练函数的集合，每个训练函数 $t \in \mathcal{T}$ 对应一个 Serverless 函数，用于执行模型训练任务。

模型的形式化定义展示了由多个关键组件构成的数据加载模型。这些组件分工明确、相互协作，共同完成数据加载过程，充分考虑到了 Serverless 环节的特点：

- **弹性计算**。模型中的数据加载函数和训练函数都运行在 Serverless 平台的函数即服务（FaaS）环境中，可以根据数据加载需求动态创建和销毁，充分利用 Serverless 平台的弹性计算能力。
- **无法对底层平台进行修改**。模型在 Serverless 平台上实现分布式数据加载，无需修改底层平台，只需根据模型定义的组件和交互方式，设计和实现相应数据加载系统即可。
- **函数间通信受限**。模型引入了数据加载流服务组件，作为有状态的数据中继，用于连接和调度多个无状态的数据加载函数，承担了数据在函数间传输的任务，通过数据流转机制解决了函数间通信受限的问题。

在 Serverless 分布式数据加载模型中，数据加载流是最重要的组成要素，它代表了一个完整的数据加载过程。每个数据加载流都包括数据集、数据加载函数、数据加载流服务、数据消费者等多个组件，角色依次对应为数据集、数据生产者、数据中继和数据消费者，它们之间通过数据的流动和转换来完成数据的加载过程。

数据加载流 (Data Loading Flow)

数据加载流可根据抽象层次分为逻辑数据加载流和执行数据加载流两种类型。逻辑数据加载流描述了数据加载的抽象过程和依赖关系，而执行数据加载流则表示实际的数据加载执行过程。如图3-4所示，一个逻辑数据加载流在运行时可用于创建多个执行数据加载流，体现了两者之间的关系。

为了定义逻辑数据加载流，本文引入了数据加载流描述对象的概念。数据加载流描述对象是一种用于描述数据加载过程的数据结构，包含了数据加载流的

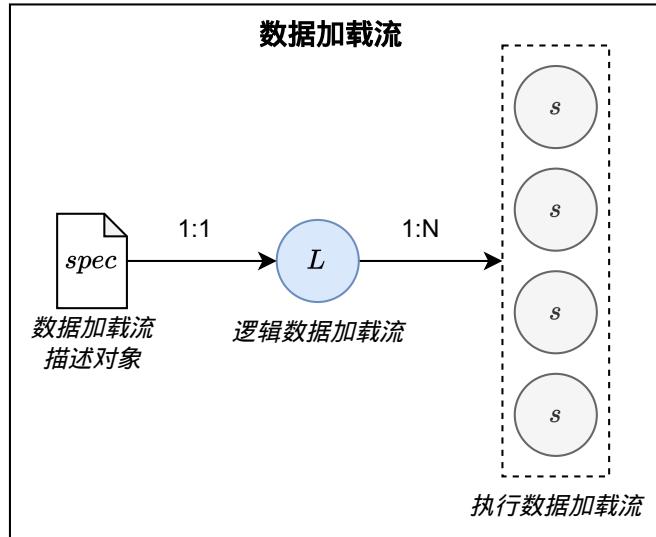


图 3-4 逻辑数据加载流与执行数据加载流的关系

各种元信息，如数据集名称、在线预处理步骤以及优化配置等。用户可通过编程框架定义数据加载流描述对象，从而描述所需的数据加载过程。

定义 3.2 (数据加载流描述对象) 数据加载流描述对象 $spec$ 可以表示为一个三元组：

$$spec = (D_{name}, P, O)$$

其中：

- D_{name} 表示数据集的名称，用于唯一标识一个数据集。系统可以通过数据集名称来定位和访问对应的数据集。
- $P = p_1, p_2, \dots, p_n$ 表示在线预处理步骤的有序列表，其中 p_i 表示第 i 个预处理操作，如图像解码、数据增强、数据转换等。预处理步骤的顺序决定了数据转换的流程。
- O 表示优化元信息，包括缓存控制、预取参数等配置，用于指导执行数据加载流的优化。例如缓存控制可用于指导中间结果的是否进行缓存，预取参数可以指导数据的预取粒度。

逻辑数据加载流描述了数据在各个处理步骤之间流转的逻辑顺序和依赖关系，但并不涉及具体的执行方式。在实际应用中，一个特定的训练任务通常对应一种逻辑数据加载流，若任务相关，则它们可以共用同一个逻辑数据加载流。例如，在 ImageNet 数据集^[48]上进行图像分类任务的超参搜索任务时，多个并发训

训练任务可以使用相同逻辑数据加载流，这也意味着它们在数据加载阶段会读取相同的数据集以及使用相同的数据加载过程。

逻辑数据加载流描述了数据在各个处理步骤之间的逻辑流转和依赖关系，定义了数据集选择、预处理操作组合等数据加载的抽象过程。在应用中，一个训练任务通常对应一种逻辑数据加载流，但对于共享同一数据集和预处理流程的相关任务，如 ImageNet 图像分类的超参数搜索，可以复用同一逻辑数据加载流。

这种逻辑流的复用不仅减少了重复定义的工作，还能提高数据加载效率。在超参搜索场景下，尽管并发的训练任务使用不同的超参数组合，但它们在数据加载阶段读取相同的 ImageNet 数据集，应用相同的数据增强、格式转换等预处理操作，只是训练阶段使用不同超参数。共用逻辑数据加载流可以使这些任务的数据加载过程得到统一优化，提升了整个超参搜索的效率。

定义 3.3 (逻辑数据加载流) 逻辑数据加载流 L 可表示为一个三元组：

$$L = (spec, L_{kind}, D)$$

其中：

- $spec$ 表示该逻辑数据加载流使用的数据加载流描述对象。
- L_{kind} 表示逻辑数据加载流的类型，例如本地或者分布式。
- D 表示该逻辑数据加载流对应的数据集。

逻辑数据加载流可以根据执行方式分为本地逻辑数据加载流和分布式逻辑数据加载流两种类型。本地逻辑数据加载流用于在本地执行数据加载过程，通常用于测试和调试目的。而分布式逻辑数据加载流则表示在分布式环境中执行的数据加载过程，利用了 Serverless 平台的并行计算能力，用于实际的模型训练任务。

与逻辑数据加载流相对应，执行数据加载流表示实际的数据加载执行过程。它将逻辑数据加载流中定义的处理步骤映射到具体的数据加载函数，并在 Serverless 平台上执行。执行数据加载流的生成和调度由系统自动完成，用户无需关注具体的执行细节。

定义 3.4 (执行数据加载流) 执行数据加载流 $e \in \mathcal{E}$ 可表示为一个二元组:

$$e = (L, f)$$

其中:

- L 表示该执行数据加载流对应的逻辑数据加载流。
- f 表示执行该数据加载流的函数实例。函数实例负责加载和预处理数据，并将结果传递给下游的训练函数。

3.2.2 模型的运行时行为

Serverless 分布式数据加载模型的运行时行为刻画了实际数据加载过程，主要体现在三个方面：数据流转、数据加载流转换和优化。数据流转定义了数据在各组件间的传输方式。数据加载流转换描述了如何从逻辑流生成多条执行流，并调度到多个数据加载函数中执行。优化则对整个加载过程进行改进，使其更高效。通过这三个步骤，最终将逻辑上的数据加载过程映射到真实环境中。图 3-5 展示了模型的完整运行时行为，下面将进行详细讨论。

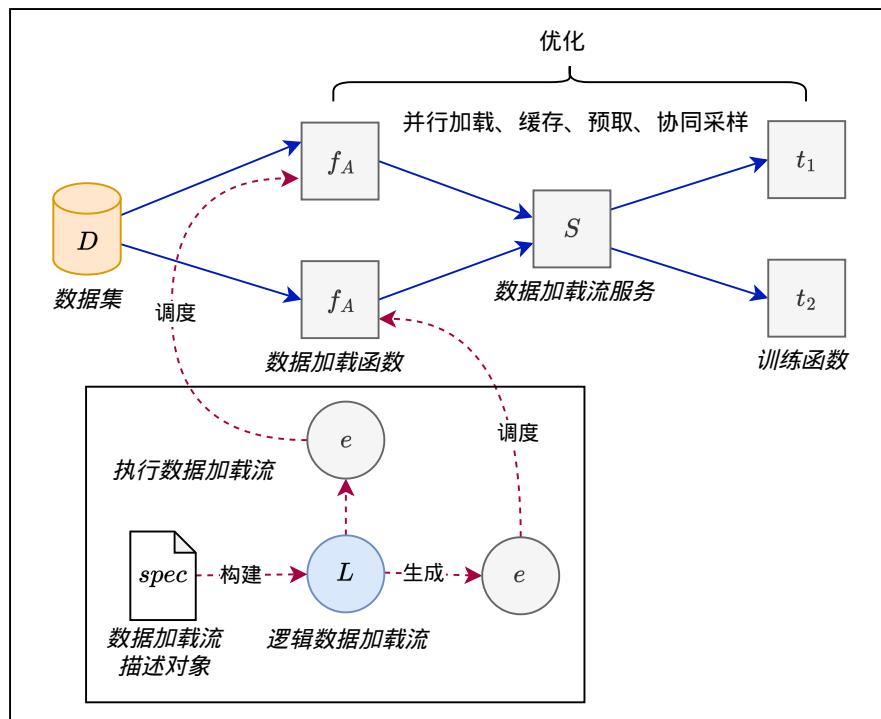


图 3-5 模型的运行时行为示意图

数据流转方式

在 Serverless 分布式数据加载模型中，数据流转是指数据在各个组件之间传输和转换的过程。具体来说，数据集首先存储在分布式文件系统或对象存储等数据源中，然后由数据加载函数读取并进行在线预处理过程。加载后的数据通过数据加载流服务进行缓存、合并和分发，最终流入到训练函数中，供深度学习模型训练使用。

形式化地，可以将数据流转过程定义为一个图：

定义 3.5 (数据流转图) 数据流转图 G 可以表示为一个二元组：

$$G = (V, E)$$

其中：

- $V = D, F, S, T$ 表示数据流转图的节点集合，包括数据集节点 D 、数据加载函数节点 F 、数据加载流服务节点 S 和训练函数节点 T 。
- $E \subseteq V \times V$ 表示数据流转图的边集合，代表数据在节点之间的流动方向。如果 $(v_i, v_j) \in E$ ，则表示数据可以从节点 v_i 流向节点 v_j 。

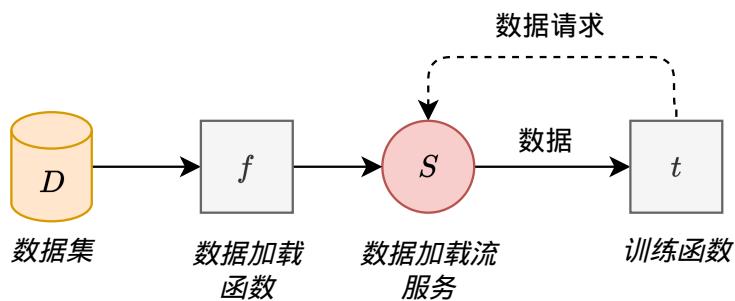


图 3-6 模型的数据流转示意图

这种数据流转方式具有以下优点：

- **解耦。**数据加载和模型训练被分离到不同的组件中，各组件之间通过明确定义的接口进行通信，实现了松耦合。这种解耦使得各组件可以独立开发、测试和部署，提高了系统的灵活性和可维护性。

- **并行**。数据加载函数可以并行执行，同时处理多个数据分片或样本。这种并行加载可以充分利用 Serverless 平台的弹性计算能力，提高数据加载的吞吐量和效率。
- **可优化**。通过数据加载流服务的预取、缓存、采样等优化，可以减少数据传输的延迟和开销，提高数据加载的效率。

数据加载流的转换

Serverless 分布式数据加载模型中定义了数据加载流的转换机制，即如何根据逻辑数据加载流生成和调度执行数据加载流。逻辑数据加载流和执行数据加载流是该模型中的两个重要概念。逻辑数据加载流描述了数据加载的抽象过程和依赖关系，包含了数据集的访问方式和预处理方法等元信息，反映了数据加载的逻辑结构。而执行数据加载流则表示实际的数据加载执行过程，是根据逻辑数据加载流生成的具体任务。

在模型的转换机制中，首先需要将数据加载流描述对象构建为逻辑数据加载流。当一个训练任务提交时，系统会根据对应的数据加载流描述对象来构建逻辑数据加载流，提取其中的数据集访问方式和预处理方法等元信息。然后，系统根据逻辑数据加载流生成和调度相应的执行数据加载流，实现实际的数据加载过程。

执行数据加载流的生成和调度，遵循以下原则：

- **数据集分片**。系统会根据数据集的大小和分布，自动将其拆分为多个数据分片，每个执行数据加载流只负责一个分片范围的数据集，通过分片可以实现并行加载。
- **动态执行流调度**。系统会在运行时根据数据加载函数的运行情况，动态调度执行流到不同的加载函数中，以提高加载速度以及函数的资源利用率。

形式化地，可以将逻辑流到执行流的转换定义为一个函数：

定义 3.6 (逻辑流到执行流的转换函数) 逻辑流到执行流的转换函数 \mathcal{T} 可以表示为：

$$\mathcal{T} : L \rightarrow E$$

其中：

- L 表示逻辑数据加载流。
- E 表示执行数据加载流的集合。
- 对于任意一个逻辑数据加载流 L , $\mathcal{T}(L)$ 表示由 L 转换生成的一组执行数据加载流 e_1, e_2, \dots, e_n 。

转换函数 \mathcal{T} 的实现需要考虑数据集分片、执行流调度等因素。

数据集分片。设数据集 D 的大小为 $|D|$, 分片大小为 S , 则数据集可以被拆分为 $\lceil \frac{|D|}{S} \rceil$ 个分片。系统会为每个分片绑定多个执行数据加载流, 从而实现数据并行, 提高加载效率。

定义 3.7 (数据集分片) 数据集分片可以表示为一个函数 \mathcal{P} :

$$\mathcal{P} : D \rightarrow D_1, D_2, \dots, D_n$$

其中:

- D 表示原始数据集。
- D_1, D_2, \dots, D_n 表示数据集被拆分后的 n 个分片, 且满足 $D = \bigcup_{i=1}^n D_i$ 。

执行流调度。设 \mathcal{F} 表示数据加载函数的集合, 则执行流调度可以表示为一个函数 \mathcal{S} 。 \mathcal{S} 根据加载函数的运行状态, 将执行流动态分配到不同的加载函数中, 以提高加载速度和资源利用率。

定义 3.8 (执行流调度) 执行流调度可以表示为一个函数 \mathcal{S} :

$$\mathcal{S} : E \times \mathcal{F} \rightarrow \mathcal{F}$$

其中:

- E 表示执行数据加载流的集合。
- \mathcal{F} 表示数据加载函数的集合。
- 对于任意一个执行流 $e \in E$ 和加载函数 $f \in \mathcal{F}$, 有 $\mathcal{S}(e, f) = f'$, 表示将执行流 e 调度到加载函数 f' 上执行。

图3-7展示了在单任务训练或多个互不相关任务并发训练场景下的数据加载流执行过程。系统会根据任务指定的逻辑数据加载流, 生成多个并发执行的数

据加载流。这些执行流会被动态调度到不同的数据加载函数上执行，实现数据的并行加载。

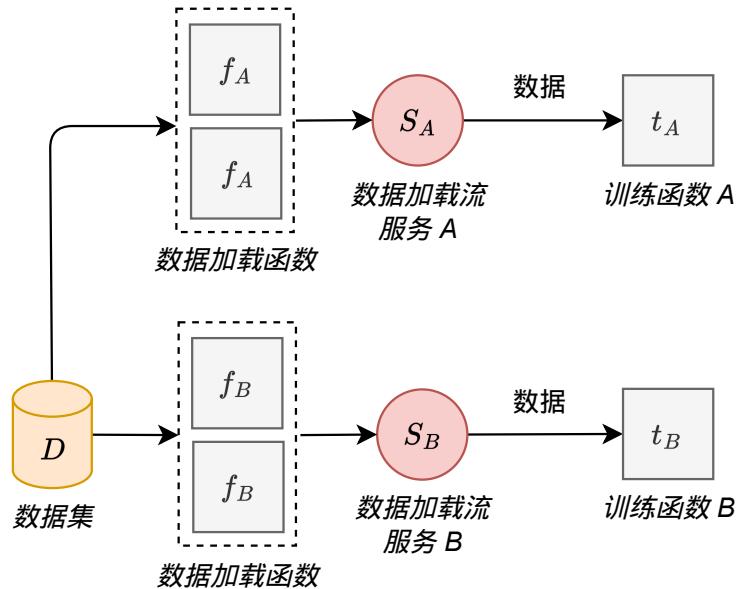


图 3-7 单任务训练中的执行数据加载流示意图

数据加载流优化

数据加载流可以通过一系列优化方法来减少数据加载开销，提高执行效率。设 \mathcal{O} 表示数据加载流优化方法的集合，则数据加载流优化可以表示为一个函数 \mathcal{A} 。 \mathcal{A} 将优化算法应用到执行数据加载流上，生成优化后的执行流。

定义 3.9 (数据加载流优化) 数据加载流优化可以表示为一个函数 \mathcal{A} :

$$\mathcal{A} : E \times \mathcal{O} \rightarrow E$$

其中：

- E 表示执行数据加载流的集合。
- \mathcal{O} 表示数据加载优化方法的集合，包括并行加载、数据预取、数据缓存、协同采样等方法。
- 对于任意一个执行流 $e \in E$ 和优化算法 $o \in \mathcal{O}$ ，有 $\mathcal{A}(e, o) = e'$ ，表示将优化算法 o 应用到执行流 e 上，生成优化后的执行流 e' 。

图3-8展示了在多任务训练场景下，通过数据加载流优化方法，如数据缓存和协同采样，实现了在多个任务中复用执行数据加载流，提高数据加载的整体效率的过程。这种优化的前提是多个任务使用相同的逻辑数据加载流，那么它们可以共享同一组执行数据加载流，避免重复读取和预处理相同的数据。这种复用机制可以减少数据加载的开销，提高资源利用率。

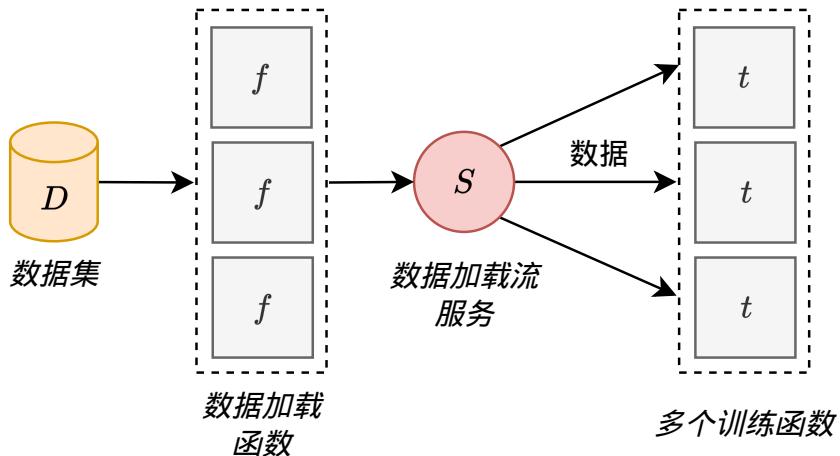


图 3-8 多任务训练中的执行数据加载流复用示意图

综上所述，数据加载流的生成、调度、优化过程可以表示为一个复合函数：

$$\mathcal{T} = \mathcal{A} \circ \mathcal{S} \circ \mathcal{P}$$

整个过程包括三个步骤：逻辑加载流生成执行流、执行流调度和数据加载流优化。通过这种转换机制，系统可以根据逻辑数据加载流转换为高效的执行数据加载流，充分利用 Serverless 平台的计算资源，实现高性能的分布式数据加载。

3.3 Serverless 分布式数据加载的编程框架

为了方便用户使用3.2节定义的Serverless分布式数据加载模型，本文提出了一套编程框架，该框架的目标是为用户提供一组抽象和对应的Python API，用于定义和描述分布式数据加载模型中的关键组成要素。编程框架可以视为连接Serverless分布式数据加载模型和最终用户编写的训练应用的桥梁。借助该框架，用户可以便捷地构建和表达端到端数据加载的逻辑过程，配置相关元信息，最终实现在Serverless平台上的布式数据加载。

本节设计的编程框架由两大部分组成：静态数据集通用存储规范和数据加载过程描述。

静态数据集通用存储规范适用于数据加载中的数据集读取阶段。该规范提供了统一的数据集组织范式，适配多种文件系统，并引入元信息分片机制，在提高通用性的同时降低了内存占用。在此基础上，框架给出了统一的数据集读取算法，契合了数据加载模型中数据集分片的思想。

数据加载过程描述部分则关注描述模型中核心的组成要素——数据加载流，以及数据加载的在线预处理阶段。本文引入了多种语义对象，分别与模型组件进行映射，并定义了语义对象间的关系和具体的 API。通过这套声明式的数据加载过程描述范式，用户能灵活地定义满足不同训练任务需求的数据加载过程，为后续的分布式数据加载执行和优化提供了基础。

3.3.1 静态数据集通用存储规范

静态数据集特点

静态数据集，也称为批处理数据集，是指在训练过程中内容保持不变的数据集。与动态数据集或流式数据集相比，静态数据集的样本总数固定，并且支持按索引随机访问。当前，绝大多数深度学习任务，如图像分类、目标检测、语音识别等，都基于静态数据集进行离线训练。然而，静态数据集在存储和加载方面存在以下挑战：

- **存储格式差异大。**不同领域、不同类型的数据集在存储格式上差异很大，缺乏统一的读取方式，增加了通用数据读取方法设计的难度。
- **元信息索引文件体积大。**原始数据集通常采用将样本数据分离为多个小文件存储，而将数据集的元信息存放到一个文件中的组织方式，导致元信息索引文件体积庞大，数据读取速度慢。
- **重复元信息加载问题。**复杂的深度学习流程需要并发执行多个训练任务，而每个任务都要从头读取元信息索引文件，造成 IO 冗余和内存浪费。
- **数据集更新管理问题。**数据集版本迭代时，如何管理和应对增量更新是一大难题。

存储规范设计

为解决上述问题，本文设计了一套静态数据集通用存储规范。该规范综合考虑了 Serverless 训练场景的特点，尤其是对象存储、云上 NFS 等环境的使用需求，在易用性、通用性和性能之间取得平衡，同时不对原始数据集的数据做修改。规范的核心是将数据集的逻辑结构与物理存储解耦，引入以下三个语义对象：

- **数据集存储（DatasetStore）**。用于描述和管理数据集的集合。
- **数据集（Dataset）**。代表一个单独的数据集实例，包含数据集的关键属性定义。
- **分片化数据集元信息（DatasetShardMeta）**。将数据集的样本元信息分片化，以提高读取性能。

图3-9描述了三个核心语义对象及其关系。DatasetStore 对象通过 JSON 格式的配置文件声明，记录管理的所有 Dataset 列表。Dataset 对象用于描述数据集，包含版本号、变种列表、样本总数、元信息分片大小等属性。DatasetShardMeta 作为 Dataset 的子对象，表示样本元信息的分片。每个分片包含固定数量样本的元信息，以降低单个元信息文件的体积。

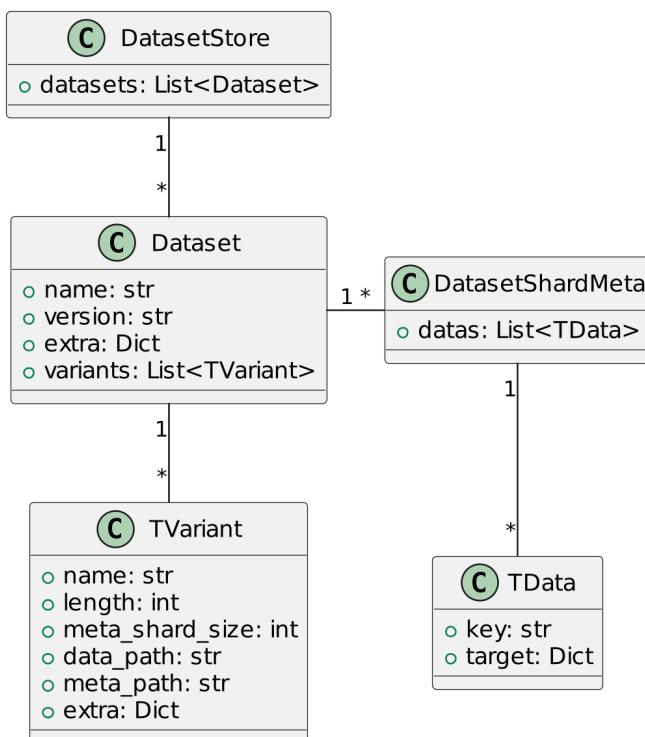


图 3-9 存储规范的语义对象关系图

在语义对象的基础上，数据集存储规范进一步约定了标准化的目录组织结构。如图3-10所示，文件系统的根路径下设置 /datasets 作为 DatasetStore 的顶层目录，用于存放和管理所有的数据集。每个数据集以 namespace/dataset 的形式命名，形成一个两段式的数据集 ID。其中 namespace 用于支持多租户隔离，dataset 标识具体数据集。

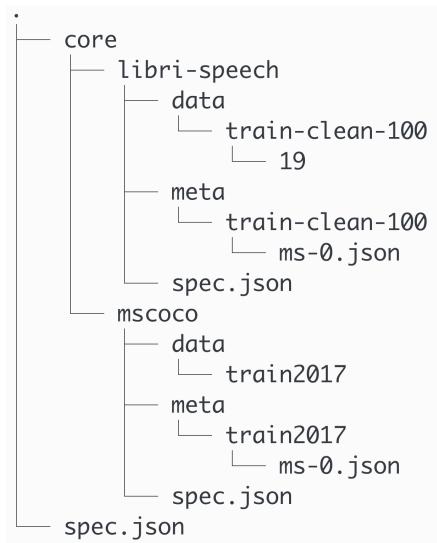


图 3-10 数据集目录组织结构

以 MSCOCO 数据集^[49]为例，数据集 ID 定义为 core/mscoco，对应的数据集根目录为 /datasets/core/mscoco。在该目录下，样本数据和元信息分别存储在 data 和 meta 两个子目录中。data 目录维持了 MSCOCO 原有的组织形式，如训练集图像存放于 /datasets/core/mscoco/data/train2017。而 meta 目录则引入了分片机制，将样本元信息按照固定大小切分为多个小文件，命名格式为 ms-{shard_no}.json。在引入元信息分片机制前，MSCOCO 原始的 annotations 文件体积为 449MB。采用分片后，每个分片的大小减少至约 40MB。

这种面向对象的目录组织模型，将数据集的样本数据、元信息、配置描述统一在一棵目录树中，具备逻辑清晰、层次分明等好处，降低了数据集的管理复杂度。同时，由于语义对象本身基于 JSON 文件定义，天然具备良好的可扩展性，可以灵活添加和更新数据集的配置信息。

综上所述，存储规范解决了静态数据集在存储和加载方面的一些挑战，具有以下优势：

- **适配不同的文件存储系统。** 数据集目录可以位于对象存储、NFS、HDFS 等

多种文件系统，只需满足 POSIX 文件语义即可。未来也可扩展支持表格存储等结构化存储系统。

- **解耦了数据集的逻辑结构与物理存储。**通过引入统一的语义对象和目录组织模型，将数据集的样本、元信息、配置等内容组织在一起，屏蔽了底层存储系统的差异。
- **元信息分片降低了单个索引文件的体积。**分片机制通过将样本元信息均匀切分为多个小文件，降低了单个文件的体积。这样在数据加载时就可以按需读取当前处理的一部分样本的元信息，避免重复加载。
- **数据集版本管理机制。**每个数据集明确记录版本号，当内容更新时只需修改相应文件并递增版本号，使数据集升级的成本可控。
- **不对原始数据集数据修改。**规范对数据集的组织和描述做了抽象，但并不要求对原始数据集内容本身进行转换，而只对元信息做预处理。

统一数据集读取算法

基于静态数据集通用存储规范，本文进一步设计了统一的数据集读取 (Unified Dataset Reader, UDR) 算法，屏蔽了不同数据集在组织和访问形式上的差异。UDR 算法定义了一个统一的数据集读取器，其核心流程如算法3.1所示。该算法首先根据样本序号计算分片 ID 和分片内偏移，然后检查分片缓存，如未命中则从文件系统读取并更新缓存。之后根据分片内偏移获取样本元信息，并据此读取实际的样本数据内容，最后返回完整的样本数据及元信息。UDR 算法基于数据集存储规范，实现了高效、统一的数据集样本访问方式。它消除了不同数据集读取逻辑的差异，简化了上层应用的开发。同时，UDR 充分利用元信息分片和缓存机制，在保证性能的同时，也降低了内存占用。

3.3.2 数据加载过程描述

为了更好地描述和组织数据加载过程，本文引入了 DataLoadFlow、DataLoadFlowReader 和 DataLoadFlowRead 三个核心语义对象，用于抽象和封装数据加载的关键概念和操作。这些语义对象相互协作，共同构成了声明式的数据加载过程描述方式，允许用户灵活地定义数据加载流程，并映射到 Serverless 分布式数

算法 3.1 统一数据集读取算法 UDR

```

1: 输入:  $ds_{id}$  数据集 ID,  $idx$  样本序号
2: 输出:  $data$  样本数据,  $meta$  元信息
3:  $D_s \leftarrow \text{Locate}(ds_{id})$                                  $\triangleright$  定位并读取数据集规范
4:  $C \leftarrow \{\}$                                           $\triangleright$  初始化元信息缓存
5:  $S_i, S_o \leftarrow \text{CalcIndex}(idx)$                        $\triangleright$  计算分片 ID 和偏移量
6: if  $C[S_i] < C$  then
7:    $M_p \leftarrow \text{JoinPath}(D_s, S_i)$ 
8:    $C[S_i] \leftarrow \text{LoadMeta}(D_s, M_p)$ 
9: end if
10:  $meta \leftarrow C[S_i][S_o]$                                       $\triangleright$  读取样本元信息
11:  $D_p \leftarrow meta.path$ 
12:  $data \leftarrow \text{LoadData}(D_p)$                                   $\triangleright$  读取样本数据内容
13: return  $\langle data, meta \rangle$ 

```

据加载模型中的关键组件。

具体而言，DataLoadFlow 对象用于描述一个完整的数据加载流，包含数据集、预处理步骤以及相关的优化元信息。它对应于 Serverless 分布式数据加载模型中的数据加载流描述对象，承担了定义逻辑数据加载流的职责。DataLoadFlowReader 是数据加载流描述对象到逻辑数据加载流的构建器接口。用户可以通过实例化 DataLoadFlowReader 的实现类，如 LocalDataLoadFlowReader 或 RemoteDataLoadFlowReader，来创建本地或分布式类型的逻辑数据加载流。转换得到的逻辑数据加载流由 DataLoadFlowRead 对象表示，它提供了进一步定制数据加载过程的接口，如指定本地预处理函数、划分数据子集等，以满足实际训练任务的需求。

下面对这些语义对象的定义和用法进行详细介绍。

DataLoadFlow 语义对象

DataLoadFlow 表示一个完整的数据加载流 (flow)。每个数据加载流拥有唯一的名称 (name) 和版本号 (version)，名称采用 namespace/name 的两段式形式，版本号用于区分同名数据加载流的不同实现。由于在线预处理逻辑可能随开发迭代而变化，引入版本号概念能够有效避免不同版本之间的混淆。

DataLoadFlow 提供了一组用户 API，允许在模型训练代码中方便地构造和操作数据加载流对象，表3-1列举了 DataLoadFlow 支持的主要方法。

dataset 方法指定数据加载流的数据来源。如果采用了通用静态数据集存储

表 3-1 DataLoadFlow 公开方法

| 方法签名 | 说明 |
|-----------------------------|--------------------|
| DataLoadFlow(name,ver) | 创建数据加载流对象，指定名称和版本号 |
| dataset(ds_name,ds_ver,var) | 指定数据加载流的输入数据集 |
| map(fn, transform, cache) | 定义数据加载流中的一个预处理阶段 |
| prepare_read() | 预读取并转换对象 |

规范，则可以很容易地根据数据集名称来读取相应的训练样本。

map 方法用于定义预处理阶段，它接受一个 Python 函数对象 fn，以及一个 transform 参数。其中 fn 封装了预处理阶段的具体计算逻辑，可以传入 Python 的函数对象；cache 设置对预处理阶段的缓存控制，如标注该阶段不可被缓存；transform 声明了预处理函数的输入输出数据格式，其接口定义如下：

```

1 class IMapTransform(Protocol):
2     def process_arg(self, arg) -> Any: ...
3     def process_ret(self, arg, ret) -> Any: ...

```

在 IMapTransform 中，process_arg 和 process_ret 分别用来处理 map 方法的输入和输出。在编程框架中，提供了 map_target 和 map_data 简化 map 的调用。

prepare_read 方法根据 DataLoadFlow 对象，结合 DataLoadFlowReader 对象，生成一个 DataLoadFlowRead 对象，供后续进行数据加载流的读取和访问。

DataLoadFlowReader 语义对象

DataLoadFlowReader 定义了数据加载流的构建器接口 Reader，用于将数据加载流描述对象（DataLoadFlow）构建为不同类型的逻辑数据加载流（DataLoadFlowRead）。

图3-11 展示了 DataLoadFlowReader 的派生关系。LocalDataLoadFlowReader 和 RemoteDataLoadFlowReader 分别对应本地逻辑数据加载流和分布式逻辑数据加载流的构建器实现。它们都实现了 DataLoadFlowReader 接口，对外提供 read 方法，在 read 中执行具体的构建逻辑。

表3-2列举了 DataLoadFlowReader 的主要方法。

LocalDataLoadFlowReader 的 read 方法根据数据加载流描述对象（DataLoad-

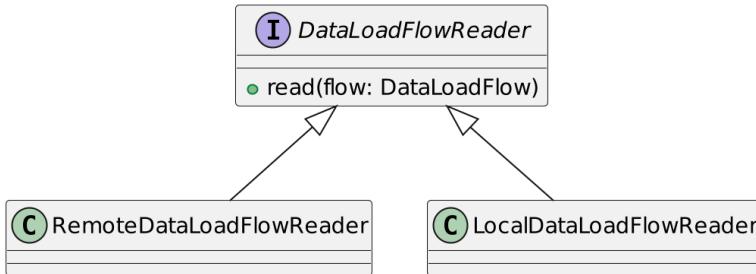


图 3-11 DataLoadFlowReader 接口对象与实现类型

表 3-2 DataLoadFlowReader 公开方法

| 方法签名 | 说明 |
|-----------------|--|
| read(flow, ...) | 将输入的 DataLoadFlow 对象构建为 DataLoadFlowRead |

Flow) 的定义，在本地创建并返回一个本地逻辑数据加载流 (DataLoadFlowRead)，这常用于测试和调试。而 RemoteDataLoadFlowReader 的 read 方法会将数据加载流描述对象上传到 DPFlow 系统中，由 DPFlow 根据描述对象创建分布式逻辑数据加载流，并在最终将其转换为相应的执行数据加载流，实现分布式数据加载。同时，RemoteDataLoadFlowReader 会返回一个分布式逻辑数据加载流在本地的代理对象，用户可以通过该代理对象在 DataLoadFlowRead 中访问 DPFlow 系统中的分布式逻辑数据加载流中的数据。

DataLoadFlowRead 语义对象

DataLoadFlowRead 表示一个已构建的逻辑数据加载流，是通过 DataLoadFlowReader 将 DataLoadFlow 对象转换而得到的产物。它提供了一组方法，用于进一步定制和转换数据加载流，以满足不同的训练需求。

表3-3给出了 DataLoadFlowRead 提供的主要接口。

表 3-3 DataLoadFlowRead 公开方法

| 方法签名 | 说明 |
|---|------------------------------|
| to_mapped() | 转换为 Mapped 数据集 |
| to_shuffled(batch_size, collate_fn, prefetch) | 转换为 Shuffled 数据集 |
| local_map(fn) | 定义仅在本地执行的预处理函数 |
| subset(indices) | 根据给定的样本索引集合，筛选出 flow 的一个子集视图 |

`to_mapped` 方法将当前逻辑数据加载流转换为一个支持随机索引访问的 Mapped 数据集对象。转换后的数据集实现了`__getitem__` 方法，允许通过索引直接访问经过在线预处理后的单个样本。该方法适用于将逻辑数据加载流与现有的数据加载框架进行集成，例如可以将 Mapped 数据集对象直接作为 PyTorch 的 DataLoader 的 `dataset` 参数使用。

`to_shuffled` 方法将当前逻辑数据加载流转换为一个经过随机打乱的 Shuffled 数据集对象。转换后的数据集采用了数据采样优化和预取优化技术，只支持顺序迭代访问。每次迭代返回一个批次的样本，批次大小由 `batch_size` 参数控制。`collate_fn` 参数指定了样本批次的组织逻辑，可以根据任务需求自定义批次的组织方式，如填充、截断等。`prefetch` 参数用于配置数据预取的相关选项，通过调整这些参数可以优化数据加载的性能。该方法主要适用于模型训练场景，可以提供高效的数据准备机制。

在实际场景中，一个常见的需求是将原始数据集随机划分为训练集和验证集。为此本文提出的编程框架提供了 `RANDOM_SPLIT` 方法，能够很好地适配逻辑数据加载流。`RANDOM_SPLIT` 能够将多个使用相同数据集的 `DataLoadFlow` 按照指定比例随机划分为不相交的子集。算法3.2给出了随机划分的实现逻辑，它通过对完整样本索引集合进行随机采样，并基于采样结果调用各 `DataLoadFlow` 对象的 `subset` 方法，从而实现了 `DataLoadFlow` 的随机子集划分。

算法 3.2 RANDOM_SPLIT() 随机划分算法

输入: 待划分的 `DataLoadFlow` 列表 $F = f_1, \dots, f_n$ ，对应的划分比例 $R = r_1, \dots, r_m$ ， $\sum_{i=1}^m r_i = 1$

输出: 划分后的 `DataLoadFlowRead` 子集列表 $S = s_1, \dots, s_m$

```

1:  $N \leftarrow \sum_{i=1}^n \text{len}(f_i)$                                 ▷ 计算样本总数  $N$ 
2:  $n_1, \dots, n_m \leftarrow r_1 N, \dots, r_m N$           ▷ 根据比例  $R$  计算每个子集的样本数
3:  $S \leftarrow \{\}$ 
4:  $I \leftarrow 1, \dots, N$                                 ▷ 样本总数范围内的完整索引列表
5: for  $i \leftarrow 1$  to  $m$  do
6:    $I_i \leftarrow \text{sample}(I, n_i)$                       ▷ 从  $I$  中随机采样  $n_i$  个索引，记为  $I_i$ 
7:    $I \leftarrow I \setminus I_i$ 
8:    $s_i \leftarrow \text{subset}(f, I_i) \mid f \in F$       ▷ 对  $F$  中的每个  $f$ ，添加生成的子集到  $s_i$ 
9:    $S \leftarrow S \cup s_i$                                 ▷ 将  $s_i$  添加到  $S$  中
10: end for
11: return  $S$ 

```

编程框架示例

为了直观展示数据加载过程描述的使用方式，本文给出了使用编程框架构建 CIFAR-100 图像分类任务的数据加载流程示例。

首先，我们创建了训练集和验证集的 DataLoadFlow 对象，指定数据集为 v1 版本的 CIFAR-100 数据集的训练集变种：

```

1 flow_train = DataLoadFlow(
2     name= 'example/cls_cifar100_train',
3     version=1
4 )
5
6 flow_train.dataset(
7     name= 'core/cifar100',
8     version= 'v1',
9     variant= 'train'
10 )

```

之后，我们通过 map、map_data API，定义了一系列在线预处理操作，如图像解码、数据增强、张量变换等：

```

1 flow_train.map_data(
2     "image_decode",
3     ImagePilDecode()
4 )
5 flow_train.map(
6     "transform_target",
7     lambda x: x[ "cat" ],
8     transform=TargetTransform()
9 )
10 flow_train.map_data("to_tensor", transforms.ToTensor())
11 flow_train.map_data(
12     "normalize",
13     transforms.Normalize(...),
14     cache_control="can-cache",
15 )

```

接下来，我们分别构建训练集和验证集的 DataLoadFlow 为分布式逻辑数据加载流，并通过 random_split 方法将其划分为不相交的子集：

```

1 flow_val = flow_train.clone()
2
3 reader = RemoteDataLoadReader(...)
4 flow_train_read = flow_train.prepare_read(reader)
5 flow_val_read = flow_val.prepare_read(reader)
6
7 flow_train_read, flow_val_read = random_split(
8     flows=[flow_train_read, flow_val_read],
9     lengths=[0.8, 0.2]
10 )

```

最后，我们将划分好的子集 DataLoadFlowRead 对象转换为随机打乱的数据集对象，指定批次大小等参数，即可在训练循环中使用：

```

1 train_ds = flow_train_read.to_shuffled(batch_size=128)
2 eval_ds = flow_val_read.to_shuffled(batch_size=256)
3
4 for bs in train_ds:
5     ...

```

这个端到端的示例展示了如何使用编程框架提供的语义对象和 API 来描述和组织数据加载过程。通过 DataLoadFlow 定义数据集和预处理操作，使用 DataLoadFlowReader 进行转换，并在 DataLoadFlowRead 上进一步定制，最终生成可供模型训练使用的数据集对象。这种声明式的数据加载过程描述方式简化了用户的开发工作，提高了灵活性和可读性。

3.4 Serverless 分布式数据加载的优化方法

本文在 Serverless 分布式数据加载模型的基础上，提出了一系列优化方法，旨在提高数据加载的性能和效率。这些优化方法包括：

并行加载优化。通过数据集分片和执行流调度等机制，将数据加载任务分配到多个数据加载函数中并行执行，充分利用 Serverless 平台的弹性资源，发挥分

布式计算的能力。本文设计了一种适用于分布式环境的动态任务调度策略——基于队列镜像和超量分配的工作窃取策略（QMOWS），能够有效平衡不同数据加载函数的任务量，优化资源利用率。

数据预取优化。在训练函数端和数据加载流服务端引入预取缓冲区，实现数据加载和模型训练的异步处理，掩盖数据加载和传输的延迟。本文采用二级预取机制，分别在训练函数端和数据加载流服务进行数据预取，最大限度地提高端到端数据加载效率。

数据缓存优化。利用缓存机制，缓存预处理的数据，复用预处理计算结果。本文提出了近似最大采样距离（ALSD）的缓存置换策略，根据数据样本在采样序列中的位置，估算其在未来一段时间内被访问的概率，并以此作为缓存优先级的依据，提高缓存命中率。

数据采样优化。通过协同采样算法，在保证采样均匀分布的前提下，尽量让不同任务采样到相同的样本，实现数据共享。本文提出了一种名为 DynaGCS 的协同采样算法，自适应地捕捉和利用多个并发任务之间的数据交集，指导数据采样的优先级，最大化数据共享和计算复用的效果。

这些优化方法结合了 Serverless 分布式数据加载模型的特点，分别针对模型中的关键组件和机制进行改进。并行加载优化主要针对执行数据加载流的生成和调度过程，通过数据集分片和动态调度策略来提高并行度和负载均衡。数据预取和缓存优化主要针对数据流转过程，通过引入预取缓冲区和缓存机制来掩盖延迟和复用计算结果。数据采样优化则能够在多任务场景下复用执行数据加载流，通过协同采样算法来最大化数据共享。

通过将这些优化方法应用到 Serverless 分布式数据加载模型中，可以提升数据加载的性能和效率，充分发挥 Serverless 计算的优势，更好地支持深度学习训练任务。

3.4.1 并行加载优化

数据加载的一个有效优化方法是根据数据集分片得到的多个数据索引对数据加载过程进行并行优化，在同一时间能够同时加载多个数据，有效提高系统的处理吞吐量。在 DPFlow 中，数据加载流服务负责管理一组远程的数据加载函数。当多个数据加载请求到来时，数据加载流服务能够将请求分配给多个函数，

实现并行加载，并收集加载结果，最终返回给加载请求方。

然而，由于难以预测数据加载请求的到达顺序，以及不同数据加载任务的执行时间不一致，可能会导致数据加载函数的处理负载不均匀等问题，从而无法达到最优的CPU利用率。为了解决上述问题，本文设计了一种适用于分布式环境的动态任务调度策略——基于队列镜像和超量分配的工作窃取策略（Queue-Mirror and Over-allocation based Work Stealing，QMOWS）。该策略能够有效地平衡不同数据加载函数的任务量，优化资源利用率。

QMOWS 动态任务调度策略

QMOWS 策略在工作窃取（Work Stealing）任务调度策略的基础上进行了改进。工作窃取任务调度策略在传统的多线程任务动态调度场景中应用广泛，其基本流程如图3-12所示，主线程会维护一个全局的任务队列，包含所有待提交给工作线程的任务，同时每个工作线程也维护一个本地的任务队列。当一个线程完成了自己队列中的所有任务时，会尝试从全局队列中获取任务；如果全局队列为空，则会随机选择另一个线程的队列，并尝试窃取任务来执行。这种策略能够很好地实现计算任务的动态负载均衡，有效应对执行时间差异较大的任务，减少空闲时间，从而提高线程利用率。

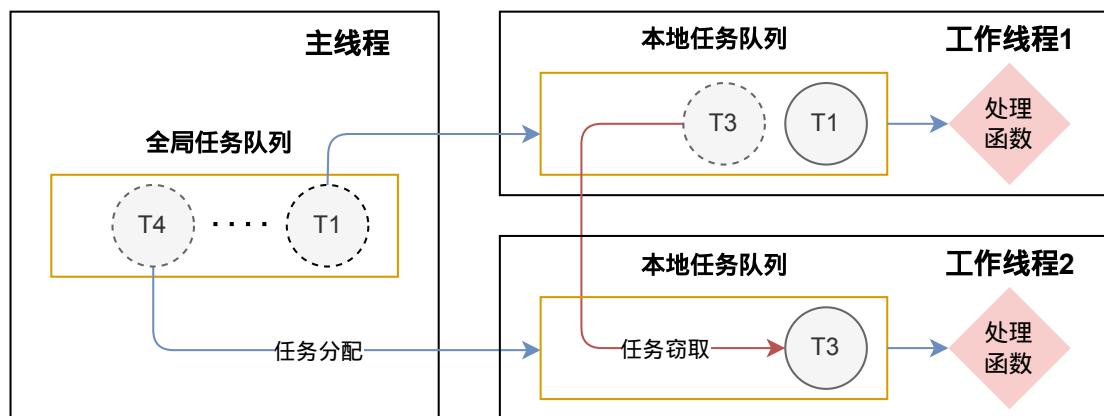


图 3-12 工作窃取任务调度策略示意图

然而，工作窃取策略在分布式场景下存在一些缺点。经典的窃取算法需要频繁地在多个分布式组件之间的通信，造成了额外的开销。针对该问题，本文提出了QMOWS策略，通过引入队列镜像和超量分配机制，有效地缓解了这一问题。与工作窃取策略类似，QMOWS策略中分为主进程和工作进程，这两类进程

位于同一个分布式集群中，通过远程过程调用（Remote Procedure Call, RPC）进行网络通信，而非多线程场景下通过共享内存进行通信，在 DPFlow 中主进程和工作进程分别对应数据加载流服务和数据加载函数。图3-13展示了 QMOWS 策略基本流程。数据加载流服务会维护一个全局的任务队列，所有新提交的任务都会添加到该队列中。同时，每个数据加载函数维护本地任务队列，优先从本地任务中拉取任务并进行数据加载过程。QMOWS 策略采用拉取（Pull）模式，即当数据加载函数空闲时，主动向数据加载流服务发起任务队列同步请求。在数据加载流服务中，会在队列镜像中进行任务分配、任务窃取等操作，并将最新的队列镜像同步给数据加载函数。

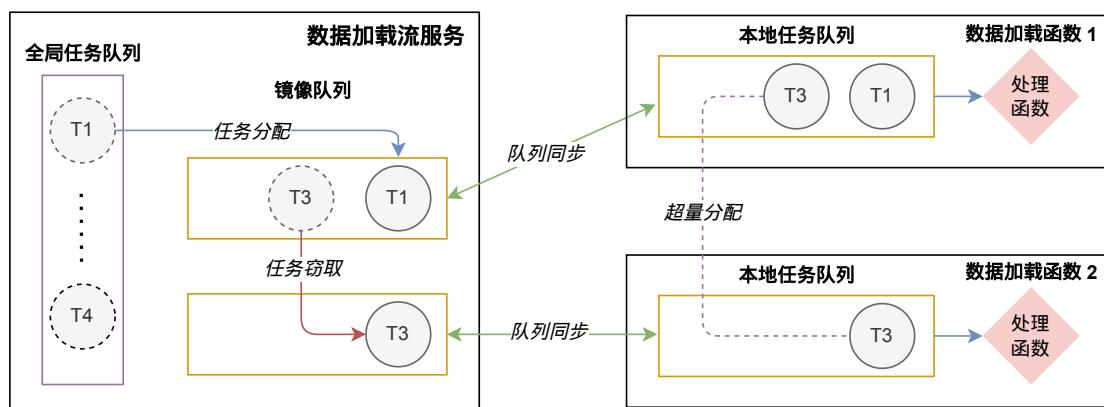


图 3-13 QMOWS 策略示意图

QMOWS 中的两个关键机制具体包括：

- 1. 队列镜像。**队列镜像通过在数据加载流服务中同步维护每个数据加载函数的本地队列副本（称为镜像队列），以实现窃取功能，并减少通信次数。在每次数据加载函数与数据加载流服务进行通信时，进行队列同步。这样，数据加载流服务可以直接从镜像队列中进行任务窃取，避免了与实际的数据加载函数进行频繁通信。
- 2. 超量分配。**超量分配利用了 DPFlow 允许数据重复加载的性质，通过超量分配数据加载任务到多个数据加载函数中，以放松数据一致性的要求，减少数据同步的通信次数，并提高函数的利用率。具体而言，数据加载函数每次向数据加载流服务拉取任务时，会一次性拉取 K 个任务，其中 K 是一个系统超参数，代表任务超量数量。这相当于每个函数维护了一个长度为 K 的滑动窗口，并存储在本地队列中。

在数据加载流服务中，会维护以下几个关键的数据结构：

- **全局任务队列**。包含所有待分配的数据加载任务。
- **加载任务状态表**。一个数据索引到加载任务对象的映射表，加载任务对象中包括任务的状态，如就绪、加载中、加载完成和加载失败。
- **函数本地队列镜像**。每个数据加载函数的本地队列副本，包含两个部分的数据，已提交到函数的部分（与函数的实际本地队列数据同步）和未提交到函数的部分（新增的任务）。在镜像本地队列中通过一个索引划分两个区域，已提交区域和未提交区域。

由于深度学习训练中数据加载的特性，单个数据加载任务的时间不长，一般在 $500\mu\text{s}$ 到 10ms 之间。为了减少通信次数，数据加载任务在加载完 N 个任务后才会向数据加载流服务发起结果返回 RPC 请求，其中 N 是一个系统超参数，代表任务的批处理量。同时，数据加载函数的任务请求和结果返回会在一个请求中进行，任务请求的响应中包含了最新的本地队列信息，数据加载函数会利用该信息更新实际本地队列，相当于不断维护长度为 K 的滑动窗口。算法3.3描述了数据加载函数的任务处理流程。

算法 3.3 HANDLETASK() 数据加载函数的任务处理算法

输入：本地队列 Q_l

```

1: while true do
2:    $R \leftarrow \emptyset$                                  $\triangleright$  初始化结果列表
3:   for  $i = 1$  to  $N$  do
4:     if  $Q_l = \emptyset$  then                       $\triangleright$  当队列为空时，提前结束循环
5:       break
6:     end if
7:      $t \leftarrow \text{QueuePopFront}(Q_l)$            $\triangleright$  从本地任务队列中取出任务
8:      $r \leftarrow \text{Process}(t)$                        $\triangleright$  执行数据加载任务，得到结果
9:      $R \leftarrow R \cup r$                              $\triangleright$  将结果添加到结果列表
10:   end for
11:    $R_t \leftarrow \text{RequestTasks}(R)$          $\triangleright$  将结果发送给数据加载流服务，请求新任务
12:    $\text{QueueSync}(Q_l, R_t)$                    $\triangleright$  根据服务响应，同步本地任务队列
13: end while

```

当数据加载函数请求任务时，即完成了镜像队列和实际队列的数据同步过程。算法3.4描述了数据加载流服务的任务调度流程，首先尝试调度本地任务，如果没有调度足够数量的本地任务，则尝试调度全局任务或者窃取其他函数的任务。

算法 3.4 SCHEDULETASK() 数据加载流服务的任务调度算法

输入: 加载任务状态表 S , 镜像队列集合 Q

```

1: while RequestReceived() do
2:    $R \leftarrow \emptyset$ 
3:    $c \leftarrow 0$ 
4:    $S, R, c \leftarrow \text{SCHEDULELOCALTASKS}(S, Q, R, c, K)$            ▷ 调度本地任务
5:   if  $c < K$  then
6:      $R, c \leftarrow \text{SCHEDULEGLOBALANDSTEALTASKS}(Q, R, c, K)$     ▷ 调度全局任务和
      窃取任务
7:   end if
8:   ReturnTasks( $R$ )          ▷ 将待返回任务列表  $R$  返回给请求的数据加载函数
9: end while

```

算法3.5展示了如何调度本地任务。首先检查该函数返回的数据加载结果，并在加载任务状态表中更新相应状态。在之后遍历镜像本地队列时，如果遇到了任务已加载完成的状态，会直接跳过。接下来，数据加载流服务会检查该函数的镜像本地队列中未提交任务个数是否小于 K ，如果大于等于 K ，那么从镜像本地队列中选取 K 个未提交任务，返回给数据加载函数。

算法 3.5 SCHEDULELOCALTASKS() 本地任务调度算法

输入: 加载任务状态表 S , 请求函数的镜像队列 Q_m

输出: 更新后的加载任务状态表 S , 待返回任务列表 R , 计数器 c

```

1: UpdateTaskStatus( $S$ ) ▷ 更新加载任务状态表  $S$ , 将已完成的任务标记为完成
2:  $c \leftarrow 0$                       ▷ 初始化计数器
3: for  $t \in Q_m$  do           ▷ 遍历镜像队列  $Q_m$  中的每个任务
4:   if status( $t$ ) = completed then
5:     continue                  ▷ 任务状态为完成, 跳过该任务
6:   else if status( $t$ ) = uncommitted then
7:     MarkAsCommitted( $t$ )        ▷ 将任务状态标记为已提交
8:      $R \leftarrow R \cup t$           ▷ 将任务添加到待返回任务列表  $R$  中
9:      $c \leftarrow c + 1$ 
10:  end if
11:  if  $c = K$  then
12:    break                    ▷ 计数器达到  $K$ , 跳出循环
13:  end if
14: end for
15: return  $S, R, c$ 

```

算法3.6展示了如何调度全局任务以及执行任务窃取过程。该函数首先检查全局队列，从全局队列中提取任务，如果还有任务无法满足，那么数据加载流服务会尝试从其他非空的函数本地镜像队列中窃取任务，可能会窃取到未提交或者已提交任务。被窃取的函数镜像本地队列会进行相应的更新，并在下一次任务

请求中进行数据同步。在分布式环境中，由于网络、系统等原因，加载任务可能会失败，因此 QMOWS 提供了容错机制。当某个任务执行失败或超时时，数据加载流服务会将该任务的状态标记为加载失败，并将其重新加入全局任务队列，以便重新进行调度和执行。

算法 3.6 SCHEDULEGLOBALANDSTEALTASKS() 全局任务调度和任务窃取算法

输入：全局队列 Q_g ，其他函数镜像队列 Q_m^i ，待返回任务列表 R

输出：更新后的待返回任务列表 R ，计数器 c

```

1: while  $c < K$  do
2:   while  $Q_g \neq \emptyset \wedge c < K$  do                                 $\triangleright$  全局队列  $Q_g$  非空
3:      $t \leftarrow \text{QueuePopFront}(Q_g)$            $\triangleright$  从全局队列中取出一个任务
4:      $\text{MarkAsCommitted}(t)$                        $\triangleright$  将任务状态标记为已提交
5:      $R \leftarrow R \cup t$                            $\triangleright$  将任务添加到待返回任务列表  $R$  中
6:      $c \leftarrow c + 1$ 
7:   end while
8:   while  $\exists Q_m^i \neq \emptyset \wedge c < K$  do            $\triangleright$  存在非空的其他函数镜像队列  $Q_m^i$ 
9:      $t \leftarrow \text{Steal}(Q_m^i)$                    $\triangleright$  从其他函数镜像队列中窃取一个任务
10:     $\text{MarkAsCommitted}(t)$                        $\triangleright$  将任务状态标记为已提交
11:     $R \leftarrow R \cup t$                            $\triangleright$  将任务添加到待返回任务列表  $R$  中
12:     $c \leftarrow c + 1$ 
13:  end while
14: end while
15: return  $R, c$ 

```

3.4.2 数据预取优化

在 DPFlow 中，分布式组件之间存在频繁的数据通信，这些通信可能会对端到端数据加载速度产生较大的影响。为了掩盖数据加载和传输的延迟，提高端到端数据加载效率，DPFlow 采用了一种二级预取机制（Two-Level Prefetch，2LP）。本节将详细介绍 DPFlow 的预取机制，并通过一个工作流程示例展示预取如何实现数据加载和模型训练的异步处理，提高数据加载性能。

二级预取机制 2LP

二级预取机制，包含本地预取与远程预取两个阶段。通过本地预取机制，DPFlow 能够掩盖数据传输的延迟；通过远程预取机制，能够掩盖数据加载和数据传输的延迟。预取的另一个作用是平衡速度，不同数据加载任务时间不同，通过预取的缓冲区能够有效地匹配生产者和消费者的速度。系统需要设置远程预

取因子 P_R 与本地预取因子 P_L 两个超参数，预取因子代表了预取缓冲区的大小。远程预取因子需要大于本地预取因子，以取得更好的效果。

图3-14展示了预取机制如何实现异步处理。在没有预取的情况下，数据加载和计算是串行执行的，加载完一个数据块后才能开始计算。引入预取机制后，数据加载和训练可以并行执行，在计算当前数据块的同时，后续数据块已经在加载中。这种异步处理提高了数据加载和计算的效率，缩短了整个训练过程的时间。

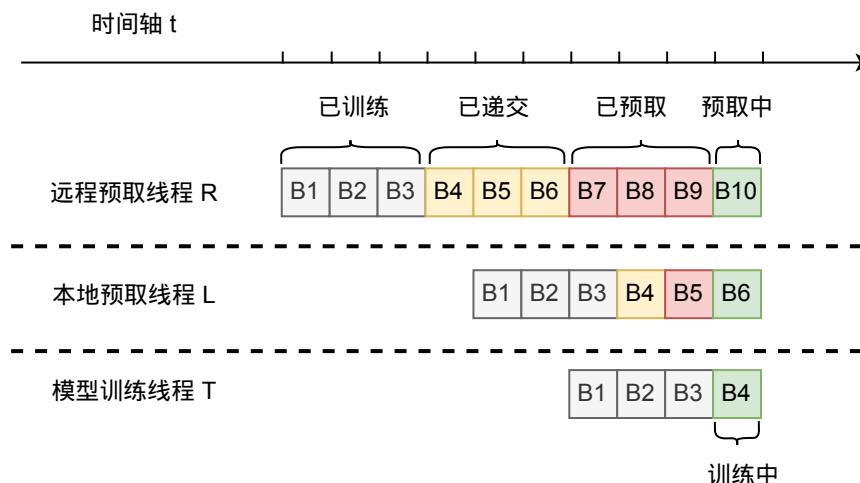


图 3-14 预取机制实现数据加载和模型训练的异步处理

本地预取

DPFlow 在训练函数端引入了本地预取缓冲区（Local Prefetch Buffer），它是一个长度为 P_L 的有序的队列，按采样顺序存放了多个批次的数据。如算法3.7所示，训练函数端同时存在本地预取线程和模型训练线程两个线程。本地预取线程不断向数据加载流服务发起数据读取请求，获取数据并放入本地预取缓冲区，直到缓冲区满为止。模型训练线程则从预取缓冲区中读取预取好的数据，并进行模型训练。

由于训练函数端的内存相对较小，本地预取缓冲区中预取的数据数量较少。本地预取的主要目的是掩盖数据传输的延迟。通过异步预取后续数据，训练函数可以在处理当前数据的同时，提前将后续数据加载到本地预取缓冲区，实现数据加载和计算的流水线并行化。

算法 3.7 本地预取和模型训练算法

LOCALPREFETCH():

输入: 本地预取缓冲区 B_L , 数据流服务 D_S

- 1: $R \leftarrow \text{GetNextBatch}()$ ▷ 获取下一批次的数据请求
- 2: **while** $\text{SIZE}(B_L) < L$ **do** ▷ 当本地缓冲区未满时
- 3: $D \leftarrow \text{RequestData}(D_S, R)$ ▷ 向数据流服务请求数据
- 4: $B_L.\text{Put}(D)$ ▷ 将获取的数据放入本地缓冲区
- 5: $R \leftarrow \text{GetNextBatch}()$ ▷ 获取下一批次的数据请求
- 6: **end while**

MODELTRAIN():

输入: 本地预取缓冲区 B_L

- 1: **while** $\text{HasNextBatch}()$ **do** ▷ 当还有下一批次数据时
- 2: $D \leftarrow B_L.\text{Get}()$ ▷ 从本地缓冲区获取一批数据
- 3: $\text{TrainModel}(D)$ ▷ 使用获取的数据训练模型
- 4: **end while**

远程预取

远程预取发生在数据加载流服务端。DPFlow 在数据加载流服务中引入了远程预取缓冲区 (Remote Prefetch Buffer), 用于存放多个训练函数预取的数据。数据加载流服务支持多个训练函数同时发起预取请求, 这些函数可能会在某个时间段内预取相同的数据。通过远程预取缓冲区, 可以复用预取的数据, 减少重复的预取操作。

算法3.8展示了远程预取的过程。训练函数会不断向数据加载流服务发起预取请求, 每个预取请求包含 P_R 个批次的数据索引信息, 但不会立即返回数据。数据加载流服务首先检查远程预取缓冲区是否已经包含所需数据, 如果存在, 则将这些数据的引用计数加一; 否则, 数据加载流服务会触发实际的数据加载操作, 将加载后的数据放入远程预取缓冲区, 并将引用计数设置为一。

算法 3.8 REMOTEPREFETCH() 远程预取算法

输入: 数据预取请求 R , 远程预取缓冲区 B_R , 引用计数映射表 C_R

- 1: **for** $r \in R$ **do**
- 2: **if** $r \in B_R$ **then**
- 3: $C_R[r] \leftarrow C_R[r] + 1$ ▷ 将请求的数据的引用计数加 1
- 4: **else**
- 5: $D \leftarrow \text{LoadData}(r)$ ▷ 执行数据加载
- 6: $B_R[r] \leftarrow D$ ▷ 将加载的数据放入远程预取缓冲区
- 7: $C_R[r] \leftarrow 1$ ▷ 将请求的数据的引用计数初始化为 1
- 8: **end if**
- 9: **end for**

算法3.9展示了训练函数向数据加载流服务发起数据访问请求的过程。请求包含一个批次的数据索引信息，数据加载流服务从远程预取缓冲区中取出所需的数据，将这些数据的引用计数减一，并返回给训练函数。只有当引用计数为零时，该数据才会从远程预取缓冲区中移除。

算法 3.9 REMOTEREQUESTDATA() 远程数据请求算法

输入：数据读取请求 R ，远程预取缓冲区 B_R ，引用计数映射表 C_R

输出：请求数据结果 D

```

1:  $D \leftarrow B_R[R]$                                 ▷ 从远程预取缓冲区获取请求的数据
2:  $C_R[R] \leftarrow C_R[R] - 1$                       ▷ 将已访问数据的引用计数减 1
3: if  $C_R[R] = 0$  then
4:    $B_R[R] \leftarrow \emptyset$                          ▷ 从远程预取缓冲区中移除数据
5:    $C_R[R] \leftarrow \emptyset$                          ▷ 从引用计数映射表中移除记录
6: end if
7: return  $D$ 

```

数据加载流服务端具有更大的内存，能够预取更多的数据。远程预取的主要目的是掩盖数据加载的延迟。通过在数据加载流服务端维护一个大容量的预取缓冲区，并结合数据引用计数机制，DPFlow 可以减少数据加载的次数，提高数据加载的效率。

3.4.3 数据缓存优化

数据缓存优化利用缓存机制，缓存加载好的数据，复用预处理计算结果。在深度学习任务训练过程中，会经历多个 Epoch，每个 Epoch 会多次遍历数据集，因此在不同 Epoch 之间可以共享一部分读取以及预处理好的数据。对于数据集而言，不同样本数据的预处理时间差异较大，即数据样本的计算量不同。优先缓存计算量大的数据样本，能够有效提高数据共享带来的 CPU 资源利用率。此外，如果多个并发训练的深度学习任务采用相同的数据集和数据预处理过程（如模型超参搜索），那么在任务之间共享加载好的数据也成为可能。

深度学习训练任务遍历数据集的方式具有优化空间。在每个 Epoch 中，训练时会对数据集进行随机采样，通常采用均匀分布的采样方式，即每个样本被采样到的概率相同，且数据集中的每个样本都会被采样一次。符合这一特征的采样序列被定义为良好序列。在深度学习训练开始前，通过提前对数据集索引序列进行随机打乱，即可确定未来的采样序列。这意味着深度学习训练任务的

采样序列可以被提前确定。利用这一特性，我们可以设计针对深度学习训练的缓存置换策略，充分考虑未来的采样序列（即数据访问序列），利用该信息能够有效提高缓存的命中率。

本文提出了一种名为近似最大采样距离（Approximate Longest Sample Distance, ALSD）的缓存置换策略，旨在优化深度学习训练过程中的数据预处理缓存。ALSD 的核心思想是根据数据样本在采样序列中的位置，估算其在未来一段时间内被访问的概率，并以此作为缓存优先级的依据。通过合理地设置缓存替换策略，ALSD 能够提高缓存命中率，减少数据预处理的计算开销。

为了描述 ALSD 算法，我们首先引入一些基本概念和定义：

- **训练任务 (Task)**。一个使用特定数据集和超参数配置进行训练的深度学习任务。
- **数据集 (Dataset)**。训练任务使用的数据集，包含若干数据样本。
- **样本 (Sample)**。数据集中的一条数据，通常由特征和标签组成。
- **采样器 (Sampler)**。在每个 Epoch 开始时，对给定的数据集索引，进行随机采样，得到采样序列。
- **采样序列 (Sampling Sequence)**。在每个 Epoch 开始时，训练任务对数据集通过采样器进行随机采样后形成的数据访问顺序，每个数据索引在采样序列中仅出现一次。
- **采样距离 (Sample Distance)**。对于采样序列中的每个样本，其与当前位置的距离称为采样距离，反映了该样本在不久的将来会被访问的时间。

给定一个数据集 $D = x_1, x_2, \dots, x_N$ ，包含 N 个样本。在每个 Epoch 开始时，训练任务会对数据集 D 进行随机重排，形成一个采样序列 $S = s_1, s_2, \dots, s_N$ ，其中 s_i 表示第 i 个访问的样本下标。

对于采样序列中的每个位置 i ，我们定义其采样距离 d_i 为该位置对应的样本在剩余序列中出现的位置与当前采样位置的差值。形式化地，采样距离 d_i 的计算公式为：

$$d_i = \min_{j>i} j | s_j = s_i - i \quad (3-1)$$

直观地，采样距离 d_i 反映了样本 x_{s_i} 在本轮 Epoch 的剩余访问序列中，距离当前位置还有多远。

假设训练任务在 Epoch 中已经访问到位置 k ，对于采样序列中剩余的 $N - k$ 个样本，我们可以计算它们的采样距离 $d_{k+1}, d_{k+2}, \dots, d_N$ 。根据采样距离的大小，我们可以估计每个样本在不久的将来被再次访问的可能性。直观上，采样距离越大的样本，在 Epoch 内部的访问间隔就越长，那么它在未来一段时间内被访问的概率就越低。

这样，采样距离较小的样本将获得较高的缓存优先级。当缓存空间不足需要进行替换时，ALSD 优先淘汰优先级最小，即采样距离最大的样本。

然而，实际的训练任务往往涉及大规模数据集和多个 Epoch，并且不同的采样方法得到不同的采样序列，完整地计算每个样本的精确采样距离并不总是可行的。为此，ALSD 引入了近似采样距离的概念。ALSD 通过采样器提供的额外信息，如部分未来确定性采样序列，和采样索引概率性分布等信息，近似估计出索引的采样距离，并利用该距离作为缓存置换指标。

假设训练任务在 Epoch 中已经访问到位置 k ，对于采样序列 N 个样本，我们可以将其划分为四个部分，如图3-15所示，这四个部分包括：

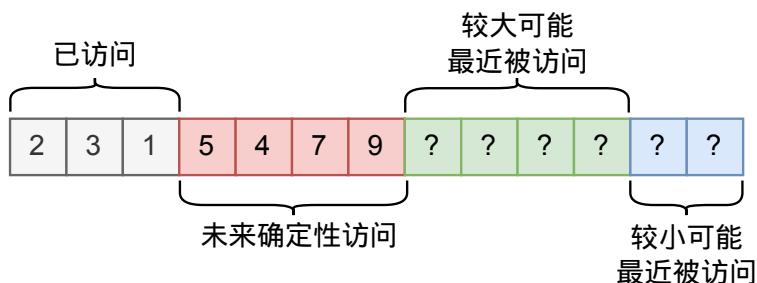


图 3-15 采样序列的采样距离划分示意图

1. 已经访问过的采样子序列 S_v 。由于每个数据索引在一个 Epoch 只会被访问一次，这些子序列中的索引只会在下一个 Epoch 中被再次访问，所以这个子序列中所有元素的采样距离被定义为整个数据集长度 N 。
2. 未来确定性的访问序列 S_c 。一些采样器，如 Pytorch、CoorDL 能够提供精确的未来采样序列，而一些采样器，如本文提出的 DynaGCS，只能提供部分确定性的采样序列。针对这些确定性的访问序列，可以精确地估计出对

应的采样距离。假设 S_c 中第 i 个元素在剩余序列中的位置为 p_i ，则其采样距离为 $d_i = p_i - i$ 。

3. 未来不确定性的访问序列，但有较大可能被访问到的序列 S_{uc} 。由于采样的随机性和动态性，采样器无法提供未来确定性的采样序列，但由于内部的数据结构维护了相关的概率信息，采样器能够提供一种未来被访问到的概率性信息，这些概率信息是一种权重信息，代表未来索引被访问到的概率权重。例如，提供序列 $\{(1, 0.1), (3, 0.5), (4, 0.2), \dots\}$ ，代表索引 1, 3, 4 在较近的时间上有一定概率被访问。

设 $S_{uc} = \{(x_1, w_1), (x_2, w_2), \dots, (x_m, w_m)\}$ ，其中 x_i 表示索引， w_i 表示对应的概率权重，满足 $\sum_{i=1}^m w_i = 1$ 。我们可以估计 S_{uc} 中元素的期望采样距离为：

$$\hat{d}_{uc} = \sum_{i=1}^m w_i \cdot (\hat{p}_i - i) \quad (3-2)$$

其中， \hat{p}_i 表示索引 x_i 在剩余序列中的期望位置，可以估计为 $\hat{p}_i = |S_c| + \frac{i}{m} \cdot (N - k - |S_c|)$ 。直观地，我们假设 S_{uc} 中的元素在剩余的 $N - k - |S_c|$ 个位置上呈均匀分布。

4. 未来不确定且较大可能在更久才会被访问到的序列 S_{uk} 。这些序列可能较远才能被访问到，并且无法提供概率信息。我们可以估计 S_{uk} 中元素的期望采样距离为：

$$\hat{d}_{uk} = |S_c| + |S_{uc}| + \frac{N - k - |S_c| - |S_{uc}|}{2} \quad (3-3)$$

综合以上四个部分，我们可以得到任意样本 x_i 的估计采样距离 \hat{d}_i ：

$$\hat{d}_i = \begin{cases} N, & x_i \in S_v \\ p_i - i, & x_i \in S_c \\ \hat{d}_{uc}, & x_i \in S_{uc} \\ \hat{d}_{uk}, & x_i \in S_{uk} \end{cases} \quad (3-4)$$

在实际的缓存替换过程中，ALSD 算法将估计采样距离 \hat{d}_i 作为样本 x_i 的缓存优先级指标。当缓存空间不足需要进行替换时，ALSD 优先淘汰具有最大估计

采样距离的样本，即 $\arg \max x_i \hat{d}_i$ 。这样可以确保缓存中保留的样本都是在不久的将来有较大概率被访问到的，从而提高缓存命中率。

ALSD 算法需要与采样器进行协同，依赖于采样器提供的额外信息，如未来确定性采样序列和概率权重等。不同的采样器可能提供不同粒度和质量的信息，这会影响 ALSD 算法的估计精度。因此，ALSD 算法的效果在一定程度上依赖于采样器的设计。本文提出的 DynaGCS 采样器能够与 ALSD 策略良好配合，提供较为准确的未来采样信息，从而实现更优的缓存性能。

在实际的训练任务中，往往同时存在多个使用相同数据集的任务。对于这种多任务场景，ALSD 也提供了支持。每个任务根据自身的采样序列和访问模式，独立地计算样本的采样距离。在缓存替换时，需要综合考虑来自不同任务的样本距离信息。ALSD 采用加权平均的方式，对多个任务的样本距离进行聚合，得到一个综合的缓存优先级指标。聚合时的权重可以反映不同任务的重要程度或者进度差异。

假设有 M 个并发的训练任务 $\{T_1, T_2, \dots, T_M\}$ ，对于样本 x_i ，每个任务 T_j 独立地计算其估计采样距离 $\hat{d}_{i,j}$ 。ALSD 通过加权平均的方式，计算样本 x_i 的综合采样距离 \bar{d}_i ：

$$\bar{d}_i = \sum_{j=1}^M \alpha_j \cdot \hat{d}_{i,j} \quad (3-5)$$

其中， α_j 表示任务 T_j 的权重，满足 $\sum_{j=1}^M \alpha_j = 1$ 。权重 α_j 可以根据任务的重要程度或者进度差异来设置。例如，可以根据任务的优先级、任务的训练速度等因素来确定权重。

通过加权平均，ALSD 能够综合考虑多个任务对样本的访问模式和缓存需求，得到一个更加全面和平衡的缓存优先级指标 \bar{d}_i 。在缓存替换时，ALSD 优先淘汰具有最大综合采样距离的样本，即 $\arg \max x_i \bar{d}_i$ 。这样可以在多任务场景下，更加高效地利用缓存资源，提高整体的缓存命中率和训练性能。

ALSD 算法的主要流程可以总结为算法3.10。在每个 Epoch 开始时，采样器生成该 Epoch 的采样序列，并将其划分为不同的部分。在训练迭代过程中，ALSD 算法根据当前的访问位置和采样器提供的信息，动态地计算每个数据样本的估计采样距离，并将其作为缓存优先级。当缓存空间不足时，ALSD 算法淘汰具有

最大估计采样距离的样本，直到腾出足够的空间。通过这种方式，ALSD 算法能够自适应地调整缓存内容，使其与训练进度保持同步，从而提高缓存命中率和性能。

算法 3.10 ALSD 策略

输入：数据集 $D = \{x_1, x_2, \dots, x_N\}$ ，包含 N 个样本
输入：缓存容量 C ，缓存替换策略参数 θ

- 1: 初始化缓存 $M = \emptyset$ ▷ 在训练开始前，初始化空缓存
- 2: 初始化缓存替换策略参数 θ ▷ 设置缓存容量等参数
- 3: **for** $e \leftarrow 1$ **to** n_e **do**
- 4: $S^{(e)} \leftarrow \text{Sampler}(D)$ ▷ 采样器生成第 e 个 Epoch 的采样序列
- 5: $S_v^{(e)}, S_c^{(e)}, S_u^{(e)}, S_k^{(e)} \leftarrow \text{Split}(S^{(e)})$ ▷ 将采样序列划分为四个部分
- 6: **for** $b \leftarrow 1$ **to** n_b **do**
- 7: $B^{(e,b)} \leftarrow S^{(e)}[b]$ ▷ 获取第 e 个 Epoch 第 b 个批次的数据索引
- 8: $S_v^{(e)} \leftarrow S_v^{(e)} \cup B^{(e,b)}$ ▷ 更新已访问序列
- 9: $\hat{d}_c^{(e,b)} \leftarrow \text{Dist}(S_c^{(e)}, b)$ ▷ 计算未来确定序列的精确采样距离
- 10: $\hat{d}_u^{(e,b)} \leftarrow \text{EstDist}(S_u^{(e)}, b)$ ▷ 估计未来不确定序列的期望采样距离
- 11: $\hat{d}_k^{(e,b)} \leftarrow \text{AvgDist}(S_k^{(e)}, b)$ ▷ 估计未知序列的平均采样距离
- 12: $\hat{D}^{(e,b)} \leftarrow \text{CombineDist}(S_v^{(e)}, \hat{d}_c^{(e,b)}, \hat{d}_u^{(e,b)}, \hat{d}_k^{(e,b)})$ ▷ 综合计算每个样本的估计采样距离
- 13: **for** $x_i \in B^{(e,b)}$ **do**
- 14: $M \leftarrow M \cup \{x_i\}$ ▷ 将当前批次的样本添加到缓存中
- 15: $p_i \leftarrow \hat{D}^{(e,b)}[i]$ ▷ 更新样本的缓存优先级为估计采样距离
- 16: **while** $|M| > C$ **do**
- 17: $x_j \leftarrow \arg \max_{x \in M} p_x$ ▷ 找到具有最大估计采样距离的样本
- 18: $M \leftarrow M \setminus \{x_j\}$ ▷ 从缓存中淘汰该样本
- 19: **end while**
- 20: **end for**
- 21: **end for**
- 22: **end for**

图 3-16 展示了两个训练速度相同的并发训练任务在使用 ALSD 与 LRU 缓存策略时的缓存命中情况。在此场景中，ALSD 和 LRU 都采用了大小为 2 的缓存空间，且初始缓存项目分别是 2 和 1。结果显示，ALSD 的缓存命中率达到了 50%，而 LRU 的缓存命中率仅为 12.5%。

总的来说，ALSD 策略利用了深度学习训练任务的数据访问特性，通过基于采样距离的缓存优先级机制，使得缓存内容能够自适应地与训练进度保持同步。具体地，ALSD 策略在每个 Epoch 开始时，根据采样器提供的信息，预估每个数据样本在未来被访问的距离，并将其作为缓存优先级。在训练过程中，对于每个批次的数据，ALSD 策略动态更新缓存内容，淘汰优先级最低的样本，尽量在缓

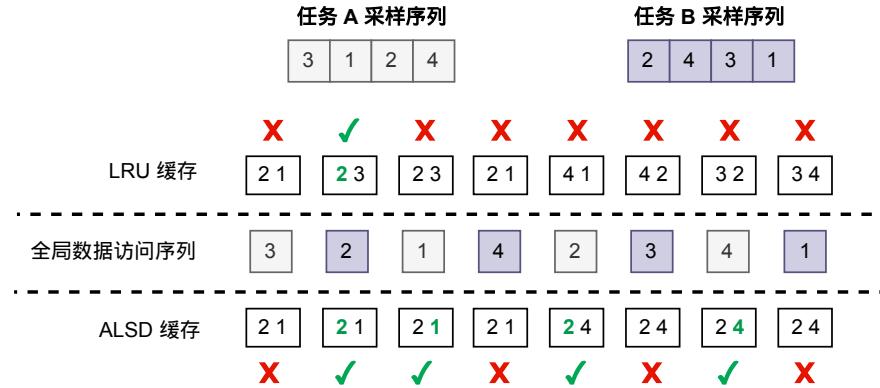


图 3-16 ALSD 与 LRU 缓存命中示意图

存中保留在不久的将来最有可能被访问到的数据。这种设计有助于提高缓存命中率，减少不必要的数据加载开销，从而加速整个训练流程。

3.4.4 数据采样优化

在多个深度学习训练任务同时训练时，如果这些任务使用同样的数据加载过程，例如超参数搜索、模型架构搜索等任务，我们还可以每个任务的采样序列进行优化调整，在保证采样均匀分布的前提下，尽量让不同任务采样到相同的样本。这样数据只需加载一次就可以被多个任务共享，在相同时间段内能够有效提高缓存命中率，提高数据加载的速度，实现数据共享。这种优化手段被称为数据采样优化，对应的算法称为协同采样算法。

为此，本文提出了一种名为 DynaGCS（Dynamic Grouped Collaborative Sampling based on Intersection Tree）的协同采样算法。DynaGCS 旨在动态捕捉和高效利用多个并发任务之间的数据交集，以交集大小、训练速度等指标指导数据采样的优先级，实现高效的数据共享。

DynaGCS 的核心数据结构是一棵动态交集树（Dynamic Intersection Tree）。交集树采用树形结构组织多个任务的采样索引子集，树种每个节点表示一个任务子集的交集。在整个训练过程中，交集树会根据任务的动态变化进行更新，反映当前训练进度下任务间的交集关系。

具体地，对于 n 个并发任务 T_1, T_2, \dots, T_n ，交集树的结构如下：

- 第 0 层：每个叶子节点对应一个单独的任务，即 $\mathcal{T}_{0,i} = T_i, i = 1, 2, \dots, n$ 。
- 第 1 层：每个节点对应两个任务的组合，即 $\mathcal{T}_{1,i} = T_j, T_k, j \neq k$ 。

- 第 2 层：每个节点对应三个任务的组合，即 $\mathcal{T}_{2,i} = T_j, T_k, T_l, j \neq k \neq l$ 。
- ...
- 第 $n-1$ 层：只有一个根节点，对应所有任务的组合，即 $\mathcal{T}_{n-1,1} = T_1, T_2, \dots, T_n$ 。

每个节点 $\mathcal{T}_{l,i}$ 维护一个交集 $I_{l,i}$ ，表示其对应任务子集在当前训练进度下的采样交集。图3-17展示了三个任务 A, B, C 形成的交集树，采样从最高层开始，依次向下分层采样。

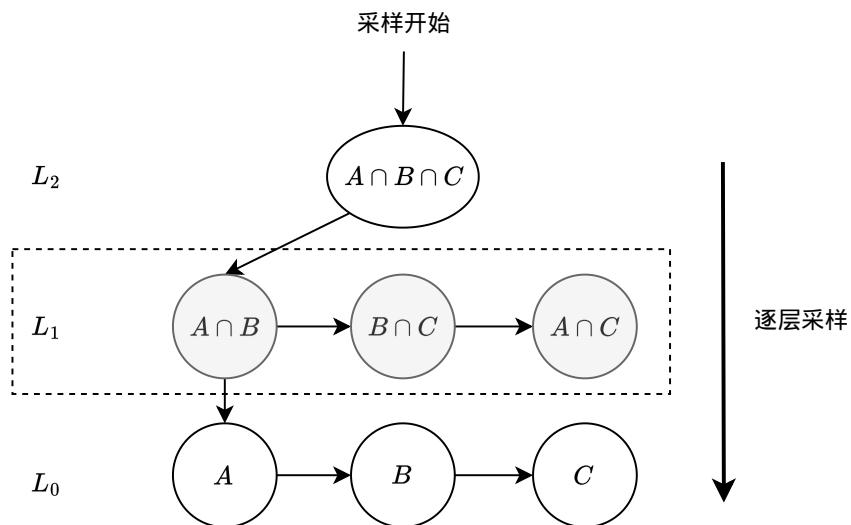


图 3-17 三个任务形成的交集树示意图

在理想情况下，交集树应该包含所有可能的任务组合，即每一层都有 C_n^m 个节点，其中 m 为该层的任务数量。然而，当任务数 n 较大时，交集树的规模会呈指数级增长，导致空间复杂度达到 $O(2^n)$ ，难以维护和更新。为了解决这个问题，DynaGCS 采用了一种近似的策略，在每一层动态选择最重要的 m 个节点进行维护，其中 m 不超过任务总数 n ，从而将最坏空间复杂度降低至 $O(n^2)$

为了评估节点的重要性，DynaGCS 引入了交集贡献度（Intersection Contribution）的概念。一个节点的交集贡献度取决于以下几个因素：

- **交集大小**。交集越大，说明节点覆盖的任务之间数据共享度越高，节点的重要性就越大。
- **任务覆盖率**。节点覆盖的任务数量越多，对整体的数据共享贡献就越大。
- **任务速度差异**。节点覆盖的任务训练速度越接近，说明它们的进度同步性越好，更适合进行协同采样。

具体而言，对于节点 $\mathcal{T}_{l,i}$ ，其交集贡献度 $c_{l,i}$ 的计算公式为：

$$c_{l,i} = \frac{|\mathcal{I}_{l,i}|}{\sum_{T_j \in \mathcal{T}_{l,i}} |S_j|} \cdot \text{speed}(\mathcal{T}_{l,i}) \quad (3-6)$$

其中， $|\mathcal{I}_{l,i}|$ 表示节点的交集大小， $\sum_{T_j \in \mathcal{T}_{l,i}} |S_j|$ 表示节点覆盖的所有任务采样子集大小之和。

speed 是一个关于任务集合 $\mathcal{T}_{l,i}$ 中各任务训练速度的函数，定义为：

$$\text{speed}(\mathcal{T}_{l,i}) = \exp \left(- \sqrt{\frac{1}{|\mathcal{T}_{l,i}|} \sum_{T_j \in \mathcal{T}_{l,i}} \left(\frac{v_j - \bar{v}}{\bar{v}} \right)^2} \right) \quad (3-7)$$

其中， $|\mathcal{T}_{l,i}|$ 为节点包含的任务数量， v_j 表示任务 T_j 的当前训练速度（每秒处理的样本数），训练速度在采样过程中动态统计得出， \bar{v} 表示节点 $\mathcal{T}_{l,i}$ 中所有任务速度的平均值。函数取值范围为 0 到 1 之间，表示速度差异的相对比例。

在交集树的动态更新过程中，DynaGCS 会根据节点的交集贡献度 $c_{l,i}$ 进行节点的选择和剪枝。具体的更新策略如下：

- 当一个新的任务 T_i 加入训练时，DynaGCS 会将其插入到交集树的所有层中，并更新相应节点的交集贡献度。对于新加入的任务，由于尚未对训练速度进行统计，其速度初始设置为该节点上所有其他任务速度的平均。如果插入 T_i 后，层中节点数超过了 m ，则选择贡献度最小的节点进行剪枝。
- 当一个任务 T_i 完成当前 epoch 的训练时，DynaGCS 会将其从交集树中暂时移除。对于每一层 l ，如果 T_i 属于某个节点 $\mathcal{T}_{l,j}$ ，则将其从节点中移除，并更新节点的交集和贡献度。如果移除 T_i 后，节点 $\mathcal{T}_{l,j}$ 中只剩下一个任务，则将该节点也一并移除。
- 在每个任务 epoch 开始时，任务会重新加入交集树，重复上述的插入和剪枝过程。

通过动态维护交集树，DynaGCS 可以捕捉任务间交集关系的变化，同时也控制了交集树的规模，使得算法的时间和空间复杂度得到了有效控制。

在数据采样阶段，DynaGCS 维护一个采样缓冲区（Sample Buffer），用于存储已采样但尚未被任务消费的样本。对于每个任务 T_i ，只有当其采样缓冲区 B_i 中的样本数量小于 batch size 时，才会将其加入到待采样任务集合 \mathcal{T}_s 中。这样可

以避免某些任务采样过多而其他任务采样不足的不平衡现象，从而有效适应不同任务的训练速度差异。

DynaGCS 每轮采样最多采样一组大小为 g 的索引序列，其中 g 计算公式为：

$$g = \min_{T_i \in \mathcal{T}_s} \left\lfloor \frac{\text{batch_size}(T_i)}{2} \right\rfloor \quad (3-8)$$

即 g 取待采样任务集合 \mathcal{T}_s 中所有任务 batch size 的最小值的一半。这样可以确保采样的样本数量不会超过任何一个任务的 batch size 限制。

DynaGCS 的采样函数 `TREESAMPLE` 算法过程如算法3.11所示。它采用自顶向下的方式，从交集树的顶层开始，逐层进行采样。在每轮采样中，会确定一个主任务 T_m ，然后在按层遍历过程中，访问所有与 T_m 相关的交集节点，从中随机采样样本，将采样得到的样本加入到相应任务的采样缓冲区中。

算法 3.11 `TREESAMPLE()` 交集树采样算法

输入：交集树 Tree，采样主任务 T_m

输出：确定性采样集合 S ，概率性采样集合 P

```

1:  $S \leftarrow \emptyset$                                 ▷ 初始化确定性采样集合
2:  $P \leftarrow \emptyset$                                 ▷ 初始化概率性采样集合
3:  $\mathcal{T}_s \leftarrow \emptyset$                             ▷ 初始化可采样任务集合
4: for  $T_i \in \mathcal{T}$  do
5:   if  $|B_i| < \text{batch\_size}(T_i)$  then
6:      $\mathcal{T}_s \leftarrow \mathcal{T}_s \cup T_i$                   ▷ 将采样缓冲区未满的任务加入可采样集合
7:   end if
8: end for
9:  $g \leftarrow \min_{T_i \in \mathcal{T}_s} [\text{batch\_size}(T_i)/2]$           ▷ 计算采样大小
10: for  $l \leftarrow n - 1$  to 0 do
11:   for  $\mathcal{T}_{l,j} \in \text{Tree}[l]$  do
12:     if  $T_m \in \mathcal{T}_{l,j}$  then
13:        $S_{l,j} \leftarrow \text{Sample}(I_{l,j}, g)$            ▷ 从节点交集中采样
14:        $S \leftarrow S \cup S_{l,j}$                          ▷ 加入确定性采样集合
15:        $P \leftarrow P \cup (I_{l,j}, l + 1)$             ▷ 加入概率性采样集合
16:       for  $T_i \in \mathcal{T}_{l,j}$  do
17:          $B_i \leftarrow B_i \cup S_{l,j}$                  ▷ 将采样样本加入任务采样缓冲区
18:       end for
19:     end if
20:   end for
21: end for
22: return  $S, P$ 

```

DynaGCS 采样得到的样本集合可以直接作为 ALSD 缓存替换算法的未来确定性采样序列，用于指导缓存替换的优先级。此外，在交集树采样过程中，

DynaGCS 还可以将采样路径上的节点交集作为概率性采样序列提供给 ALSD。具体地，对于节点 $\mathcal{T}l, i$ ，其交集 Il, i 中的样本被赋予概率权重 $w_{l,i} = l + 1$ ，表示样本所在节点的树层数。这些概率性样本可以帮助 ALSD 更好地估计未来的数据访问模式。

图3-18展示了 DynaGCS 算法在训练过程中的完整工作流程。在每个任务的 epoch 开始时，任务加入到交集树中，更新树的结构和节点贡献度。在任务遍历数据集时，DynaGCS 执行采样过程：首先检查采样缓冲区是否为空，如果为空，则从交集树中进行采样；然后从采样缓冲区中返回一个批次的采样结果，并统计任务的训练速度。

在训练迭代过程中，每个任务从自己的采样缓冲区中取出一个 batch 的样本用于训练更新。当一个任务完成当前 epoch 的训练后，会暂时从交集树中移除，直到下一个 epoch 开始时再重新加入。通过这种动态调整的方式，DynaGCS 能够实时适应任务的变化，动态调整采样相关的数据结构，充分利用任务之间的数据共享机会，提高数据加载的效率。

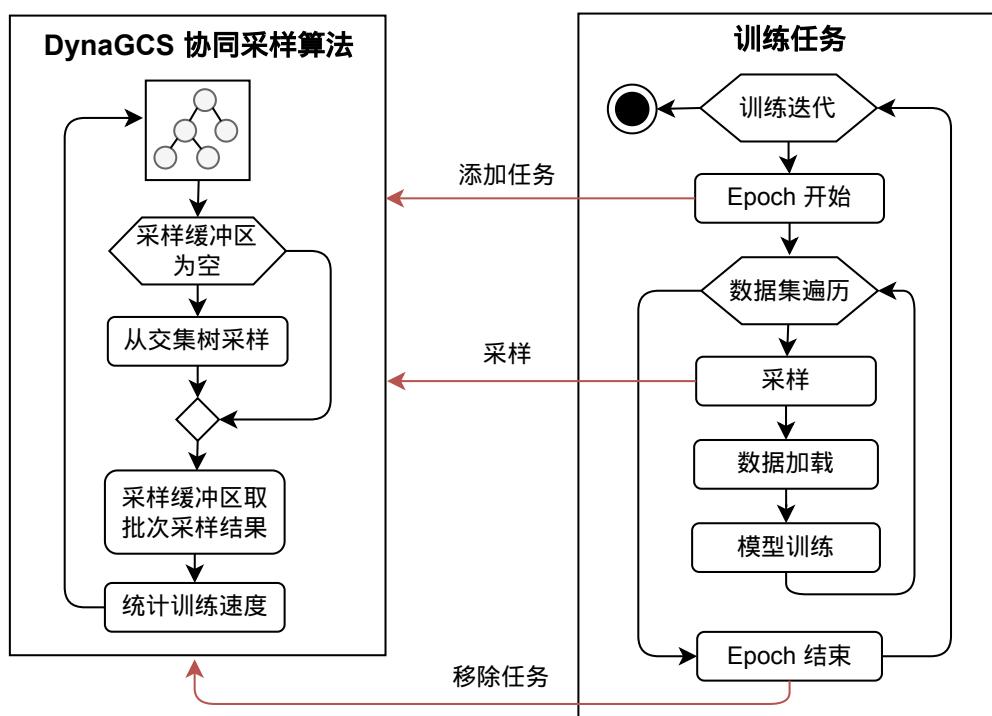


图 3-18 DynaGCS 算法在训练过程中的工作流程图

综上所述，DynaGCS 是一种高效的协同采样算法，它通过动态交集树这一数据结构，动态捕捉和利用多个并发任务之间的数据共享机会。DynaGCS 能够

根据任务的动态变化实时调整交集结构，同时与缓存替换算法 ALSD 协同工作，最大化数据共享和复用的效果，从而提升多任务训练的数据加载效率。

3.5 本章小结

本章探讨了如何在 Serverless 环境下高效实现深度学习训练的数据加载功能。本章首先设计了 Serverless 分布式数据加载模型，形式化定义了模型中的关键组件，阐述了它们之间的交互关系，为优化方法的设计奠定了理论基础。接着，本章提出了静态数据集通用存储规范和数据加载过程描述两部分内容，构成了配套的编程框架，简化了用户使用模型的流程。最后，本章在并行加载、数据预取、数据缓存和数据采样等方面，设计了一系列优化方法，目的是降低 GPU 的空闲等待时间，提升数据加载的数据加载吞吐量和资源利用率。

第四章 Serverless 分布式数据加载系统

本章将对 Serverless 分布式数据加载系统 DPFlow 进行介绍。首先介绍 DPFlow 的运行平台阿里云 Serverless 平台相关细节，接着介绍系统架构设计以及各个组件的设计。

4.1 阿里云 Serverless 产品介绍

本文提出的分布式数据加载方法具有普适性，能够适配各种 Serverless 平台。而基于该方法实现的系统需要选取一个具体的云平台进行落地和验证。考虑到阿里云作为国内最大的云厂商，其 Serverless 产品线较为丰富和完善，本文选择在阿里云 Serverless 产品之上实现 DPFlow 系统。

具体而言，本文主要使用了阿里云函数计算（Function Compute，简称 FC）平台和弹性容器实例（Elastic Container Instance，简称 ECI）服务。这两类产品均提供了 Serverless 的计算能力，具备免运维、按需计费、弹性伸缩等特点，但在某些方面也存在差异。函数计算 FC 是一种 FaaS（Function as a Service）平台，以函数为单位进行计算任务管理。它的优势在于冷启动时间较短（毫秒级），资源粒度更细，非常适合实现 Serverless 训练函数和数据加载函数等计算任务。但 FC 平台的网络隔离较为严格，函数实例之间无法直接通信，给分布式任务协同带来了挑战。而弹性容器实例 ECI 提供了类似轻量级虚拟机的计算环境，同样支持秒级计费和弹性伸缩，冷启动时间略长于 FC（秒级）。ECI 的优势在于，容器实例能够对外提供任意端口的 TCP/UDP 网络服务，更加灵活，非常适合实现数据加载优化系统的各个分布式组件，如数据加载协调服务、数据加载路由服务、数据加载分片服务等。本文充分利用 FC 和 ECI 的不同特性，将二者结合，发挥各自的优势。

除计算服务外，阿里云在存储和网络方面也提供了丰富的 Serverless 产品，为本文的系统实现提供了有力支撑。接下来进一步介绍这些关键服务。

4.1.1 存储与网络服务

在数据密集型应用中，高效的分布式存储服务至关重要。阿里云提供了多种 Serverless 形态的分布式存储服务，以满足不同场景的需求。表4-1展示了阿里云的几种主要存储服务以及功能特点，包括块存储 EBS^[50]、对象存储 OSS^[51]、文件存储^[52]以及表格存储 TableStore^[53]。本文主要使用了对象存储服务 OSS（Object Storage Service）和文件存储服务 NAS（Network Attached Storage），它们也是业界深度学习训练常用的数据存储方案^[54-55]。

表 4-1 阿里云主要存储服务及功能特点

| 存储服务 | 英文名 | 功能特点 |
|------|--------------------------------|----------------------------------|
| 块存储 | Elastic Block Storage (EBS) | 提供块级别数据存储，适合作为虚拟机的系统盘和数据盘 |
| 对象存储 | Object Storage Service (OSS) | 提供海量、安全、低成本的云存储服务，支持 RESTful API |
| 文件存储 | Network Attached Storage (NAS) | 提供共享文件存储服务，支持 NFS、SMB 等标准协议 |
| 表格存储 | TableStore | 提供海量结构化数据存储和快速查询分析能力 |

OSS 提供了高可靠、高可扩展的对象存储服务。用户可将数据、代码等内容以对象（Object）的形式上传到 OSS 中，每个文件对应一个唯一的 URL，支持 HTTP/HTTPS 访问。OSS 适合存储大量非结构化数据，如图片、视频、日志等。本文将训练数据集等内容存储在 OSS 上。

NAS 提供了一个在多个计算节点间共享的分布式文件系统。它兼容 POSIX 文件系统访问语义^[56]，可以像使用本地磁盘一样读写文件，非常适合存储结构化和半结构化数据。本文使用 NAS 存储训练代码，模型训练过程产生的日志，以及一些需要持久化的系统元数据。

阿里云上的 FC 函数实例、ECI 容器实例可以方便地挂载 OSS、NAS 等分布式文件系统，使得 Serverless 服务与存储服务可以无缝集成，简化了应用开发部署流程。图4-1 展示了在 FC 函数实例配置页面中挂载 OSS 与 NAS 文件系统。

在网络层面，阿里云提供了 VPC（Virtual Private Cloud）专有网络服务，用户可在 VPC 内创建多个虚拟交换机（vSwitch），构建逻辑隔离的网络环境。本

The screenshot displays two separate sections for mounting storage to a function instance:

- NAS 文件系统** (NAS File System):
 - 挂载 NAS 文件系统 (Mount NAS File System)
 - 已启用 (Enabled)
 - NAS 挂载点 (NAS Mount Point): 00b2b483ce-hom51.cn-hangzhou.nas.aliyuncs.com:/dpflow /data/dpflow 普通传输 (Normal Transfer)
- OSS 对象存储** (OSS Object Storage):
 - 挂载 OSS 对象存储 (Mount OSS Object Storage)
 - 已启用 (Enabled)
 - OSS 挂载点 (OSS Mount Point): dpflow http://oss-cn-hangzhou-internal.aliyuncs.com /data/s3-dpflow 只读 (Read-only)

图 4-1 FC 函数实例挂载 OSS 与 NAS 文件系统

文实现的数据加载系统将 FC 函数实例、ECI 容器实例均部署在同一 VPC 网络内，再进一步通过 vSwitch 进行子网划分，使得函数实例与容器实例之间能够互通访问、传输数据。

此外，VPC 可以与其他阿里云服务的网络环境无缝连通。例如，基于 FC 构建的 Serverless 训练任务可以直接访问同一 VPC 内的 NAS 文件系统，以获取训练代码或者写入训练日志。图4-2 展示了 DPFlow 在阿里云平台上的网络拓扑结构。

4.1.2 函数计算平台 (FC)

函数计算平台 FC 以函数为计算和管理单元。开发者可将一段代码或者容器封装为一个函数，并设置函数执行时的硬件资源，包括 CPU/GPU 规格、显存大小、内存大小等参数。函数运行时，会实例化出指定配置的计算环境，加载代码或容器镜像并执行，函数生命周期内产生的日志会被收集并存储，函数中的执行结果可以写入挂载在函数实例内部的 OSS 或者 NAS 存储中，以便持久化状态。

在 FC 中，提供了对 GPU 资源的原生支持，支持的 GPU 型号包括 Nvidia T4 和 Nvidia A10，基本满足用户的推理和训练需求。同时 FC 提供了丰富的触发器，



图 4-2 DPFlow 阿里云服务的网络拓扑关系

可基于 HTTP 请求、消息队列等多种事件源触发函数执行。平台会根据请求数量自动对函数进行弹性伸缩，免去了开发者繁琐的资源管理工作。

此外，FC 还提供了完善的 OpenAPI，支持创建、更新、删除函数，调整预留实例个数等能力，为构建 Serverless 平台的上层应用提供了灵活的管理途径。DPFlow 系统组件会通过调用 FC 的管理 API，来实现深度学习训练函数和数据加载函数的自动调度。

4.1.3 弹性容器实例 (ECI)

弹性容器实例 ECI 提供了一种 Serverless 容器实例服务，开发者可将应用打包为标准的容器镜像，在创建 ECI 实例时指定所需的 CPU、内存等硬件资源配置，实例启动后会拉取指定镜像并运行容器。ECI 同样具备秒级弹性伸缩能力，并按实际资源使用量计费。

与 FC 函数实例不同的是，ECI 容器实例启动后类似一个轻量级的虚拟机，不会受到运行时长的限制。更重要的是，ECI 支持暴露任意端口，可以对外提供网络服务，不受函数无法通信限制的影响，适合用作多个函数之间通信的桥梁。服务在容器内启动成功后，其他实例可通过服务发布的 IP 地址和端口号进行访

问。ECI 弹性及外部暴露能力，使其更加适合作为 DPFlow 系统的基础设施运行平台。

本文在 ECI 之上实现了数据加载协调服务，数据加载路由服务，数据加载分片服务等系统关键组件。FC 函数可以与 ECI 容器实例之间进行通信，例如训练函数通过 RPC 向系统发起数据加载请求。系统组件彼此通信，协同调度管理 FC 上的训练函数和数据加载函数，实现高效灵活的分布式数据加载功能。

4.2 系统架构设计

本文在第三章的基础上，提出了一种 Serverless 分布式数据加载系统 DPFlow。该系统基于 3.2 提出的 Serverless 分布式数据加载模型进行设计，包括三个组件：训练函数、数据加载函数和数据加载流服务。数据加载流服务采用微服务架构进行细化，以适应分布式系统的需求，包括数据加载协调服务、数据加载路由服务和数据加载分片服务三种子服务。本系统在阿里云 Serverless 服务上实现了一个原型系统，并以阿里云函数计算平台（FC）及弹性容器实例（ECI）为基础设施支撑。

系统的整体架构如图 4-3 所示。在训练开始时，训练函数通过 RPC 与数据加载协调服务建立连接，获取集群相关信息，例如数据加载路由服务实例的访问地址。接着，训练函数与数据加载路由服务交互，获取数据采样与分片路由信息。在数据集迭代过程中，训练函数将与多个数据加载分片服务建立连接，以并行异步的方式获取已加载的数据。数据加载分片服务负责将数据加载任务分发至多个数据加载函数，从而实现数据的并行加载。

具体而言，系统中的关键组件包括：

RPC 通信模块。 DPFlow 系统执行过程中，各组件实例需频繁进行通信和数据交换，并涉及 Rust 与 Python 的跨语言操作。系统因此设计了专为高效数据传输而设的 RPC 通信模块，提供高性能的进程间通信和数据传输能力。RPC 模块采用二进制协议格式，支持请求响应、双向流等多种通信模式，并内置多线程与异步处理机制，满足大规模数据交换场景的性能需求。

数据加载协调服务。 作为 DPFlow 的控制平面，数据加载协调服务负责管理数据加载流和数据集元信息，以及协调其他组件的工作流程。当用户提交新的

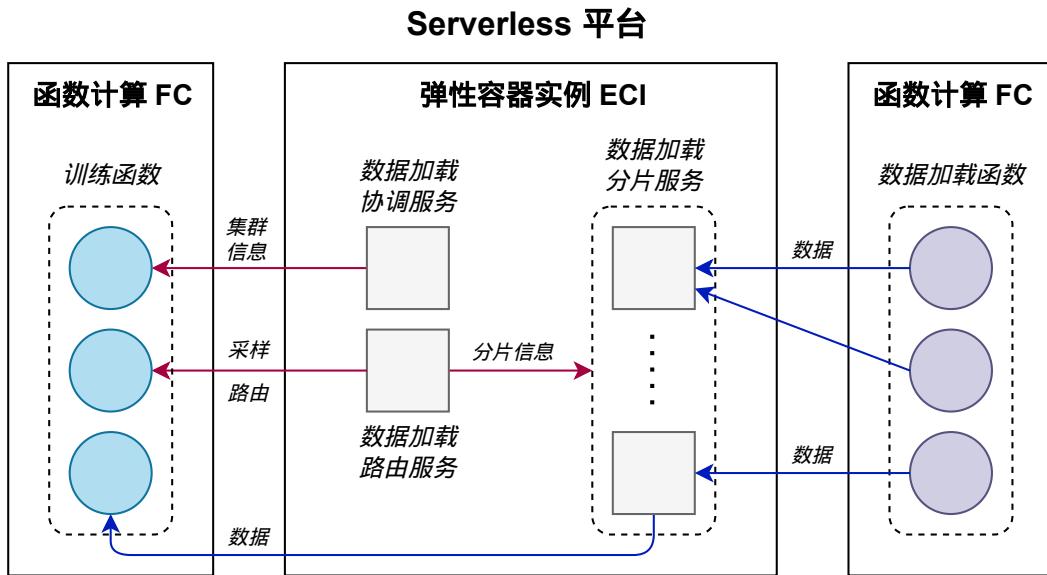


图 4-3 DPFlow 系统整体架构

训练任务时，协调服务将解析数据加载需求，创建并存储数据加载流对象，随后动态生成数据加载路由服务实例，并将地址信息返回给用户。训练过程中，协调服务会定时同步各组件的心跳，监控组件状态，并响应故障与异常。

数据加载路由服务。该服务管理一组数据加载分片服务，将完整的数据集顺序切分成多个数据分片，每个分片由一个独立的数据加载分片服务处理。在训练函数迭代访问数据集期间，训练函数定期与数据加载路由服务通信，获取数据加载分片服务实例的地址信息，以请求并获取已加载的数据样本。数据加载路由服务利用协同采样算法，动态管理多个训练任务之间的数据采样进度，实现训练任务间的数据共享。

数据加载分片服务。接收训练函数的数据获取请求后，根据任务需求动态创建或复用数据加载函数，并启动分布式数据加载任务。在数据加载分片服务中，实现了 3.4 设计的二级预取和缓存优化机制，减少冗余计算，并缓解数据生成至训练函数数据消费的延迟。优化机制根据未来将访问的数据索引序列提前进行数据计算与缓存，使得在相同子任务再次执行时，可以直接返回缓存数据，避免重复计算。

DPFlow 系统的各个服务组件以松耦合方式相连接，便于独立进行性能优化和功能扩展。系统关键组件的设计与实现将在下文中详细讨论。

4.3 系统组件设计

4.3.1 面向高效数据传输的 RPC 模块

在分布式数据加载系统 DPFlow 中，各组件之间需要频繁进行跨语言、跨进程的通信和数据传输。本文设计了一个面向高效数据传输的 RPC 模块 CRPC，旨在为涉及 Rust 与 Python 跨语言操作、大量数据处理和网络通信的场景提供高效的解决方案。CRPC 的主要设计目标包括：

- **通信功能支持。** CRPC 同时支持同步的请求-响应模式和异步的双向流模式，以适应不同的调用场景。
- **高效数据传输支持。** CRPC 采用优化的消息帧格式，支持批量数据传输以提高吞吐量。同时，它能够适配多种底层传输协议，如 IPC、TCP、UDP，并通过数据压缩、向量化写入、零拷贝读取等技术进一步提升传输效率。
- **跨语言互操作。** CRPC 实现了 Rust 与 Python 之间高效的互操作，优化了 Python 的序列化中数据拷贝次数，并支持 Python 直接访问 Rust 端的数据对象，以减少额外开销。

消息帧格式设计

CRPC 采用了一种优化的消息帧格式，以支持批量数据传输和提高系统吞吐量。如图4-4所示，每个消息帧由 Header、Tag、DataSection 三部分构成。

其中，Header 定义了消息的元信息，包括消息类型、标签长度、数据段长度等，其详细结构如表4-2所示。Tag 部分存储经过序列化的消息标签对象，标签对象携带了消息的语义信息，如请求的方法名、参数类型等。DataSection 是消息的数据载荷，由一系列数据对象组成，每个消息可以选择性地附带零个或多个数据对象。Tag 和 DataSection 共同构成了一个原子的传输单元，CRPC 保证它们要么全部成功传输，要么全部失败，以实现消息传递的事务性语义。

DataSection 由 DataLengthSection 和 DataDataSection 两部分组成。其中，DataLengthSection 存储了每个数据对象的长度信息，DataDataSection 则存储数据对象的实际内容。数据对象在 DataSection 中按照 Length-Data 的方式连续排列，即每个对象均由 8 字节的长度信息和随后的数据字节构成。这种设计允许接收方快

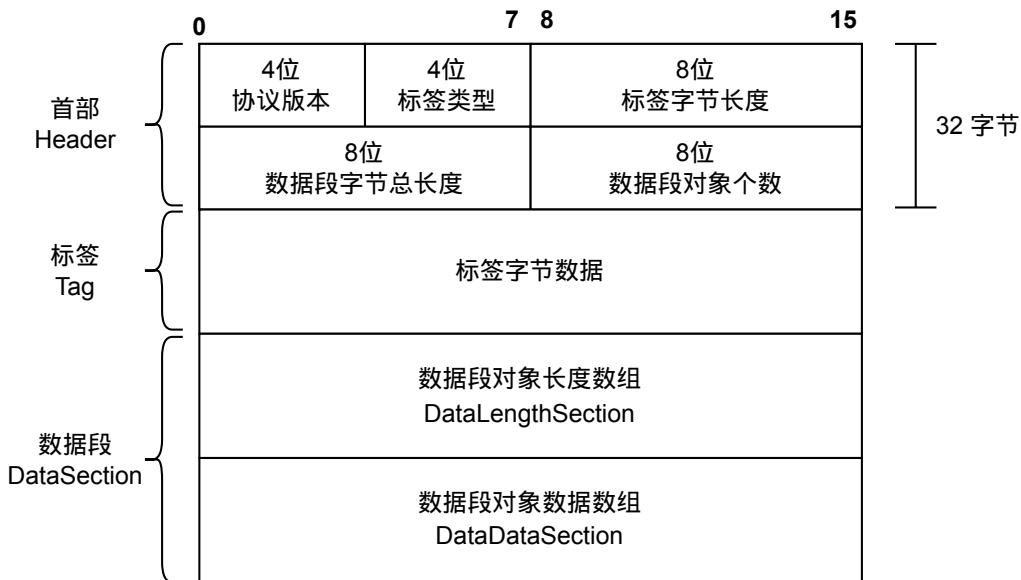


图 4-4 CRPC 消息帧格式

速定位和解析出每个独立的数据对象。

为实现消息内容的序列化与反序列化，CRPC 设计了可扩展的编解码接口。用户可以根据需要，灵活地选择和集成不同的序列化方案，如 speedy、protobuf 等。序列化后的标签对象以二进制形式存储在 Tag 段中，而序列化后的数据对象则存储在 DataSection 中。当消息到达接收方时，CRPC 会自动根据 Header 中提供的元信息，对标签和数据段进行反序列化，还原出原始的标签对象和数据对象。

表 4-2 消息帧 Header 结构

| 字段名称 | 数据类型 | 长度（字节） | 说明 |
|------------------|------|--------|-----------|
| version | u32 | 4 | CRPC 协议版本 |
| tag_kind | u32 | 4 | 标签的类型 |
| tag_len | u64 | 8 | 标签的字节长度 |
| data_section_len | u64 | 8 | 数据段的字节长度 |
| data_count | u64 | 8 | 数据段的对象个数 |

同时，针对深度学习数据加载场景的特点，CRPC 在消息传输的关键环节进行了专门的优化。首先，考虑到消息标签对象的体积通常较小，而数据对象的体积较大（数百 KB 至数 MB），CRPC 采用了针对性的写入策略：对于标签对象，CRPC 直接调用基础的 write 方法发送；而对于一批数据对象，CRPC 则是通过

writev 系统调用将它们打包成一个向量，一次性写入内核缓冲区。这种向量化写入的方式可以减少 I/O 操作的次数，提高数据发送的效率。

其次，在消息接收端，CRPC 充分利用了 Header 中的长度信息。针对 DataSection，CRPC 根据 DataLengthSection 提供的数据边界信息，以零拷贝的方式读取每个数据对象，避免了不必要的内存复制操作，同时减少内存占用。

在底层数据传输方面，CRPC 以 Trait 的形式定义了 Transport 接口，将消息的发送和接收过程抽象为 read 和 write 操作。基于 Transport Trait，CRPC 实现了对多种通信协议的支持，包括进程间通信 IPC、TCP、UDP（基于 QUIC）等。此外，得益于 Transport 接口的标准化，CRPC 还可以方便地整合到 gRPC、Thrift 等流行的 RPC 框架中。用户可以在 CRPC 的消息语义之上，复用这些框架成熟的服务治理和部署运维能力。

RPC 通信语义

在消息传输的基础上，CRPC 提供了灵活的操作语义来满足系统各个组件之间的通信需求。具体而言，CRPC 支持两种通信方式：

- **同步调用模式。** 基于请求-回复调用模型，客户端向服务端发起调用请求，并同步等待服务端返回。
- **双向流模式。** 客户端向服务端发起创建流请求，在客户端和服务端之间建立流。流创建好后，两端均可异步地向对方发送消息，并保证每一段发送消息的有序性。

同步调用模式支持客户端向服务端发起远程函数调用请求，并同步地等待响应。同步调用模式通过在连接中维护请求 ID 实现。客户端内部维护了一个哈希表，用来存储尚未返回的响应。客户端每次请求时，会创建一个 ID 作为请求的唯一标识符，创建一个一次性的双向通信信道对象，并在哈希表中添加请求信道发送者，为上层应用返回一个待完成的异步 Future 对象。Future 对象中等待信道接收者返回数据。当服务端收到请求并进行响应时，会在相应消息中附带上请求 ID，发送给客户端。客户端收到对应请求 ID 的回复后，从哈希表中取出信道发送者，并传入响应回对象，异步 Future 对象标记为完成，完成整个响应过程。

尽管同步请求-响应模式使用简单，但它也存在一些局限性。首先，每个请求都需要经历一个完整的 RTT（往返时延），难以满足延迟敏感型应用的需求。其次，同步调用会阻塞客户端的执行流程，并发度受到限制。最后，由于每次通信都是由客户端发起，服务端无法主动向客户端推送消息。

为了解决上述问题，CRPC 引入了异步双向流模式。双向流允许客户端与服务端在一个 RPC 会话中双向传输多个异步消息。首先，客户端发起一个创建双向流的请求，携带必要的元信息（如流类型）。服务端接收请求后，与客户端协商产生一个唯一的流 ID，并根据该 ID 创建一个虚拟信道。后续，双方利用该信道自由地进行消息交换，无需再进行额外的握手和请求/响应匹配。值得注意的是，虽然每个流中的消息传输是有序的，但不同流的消息可以并发发送和接收，从而充分利用底层 TCP 连接的带宽。

在实现层面，双向流复用了 CRPC 原有的连接管理机制。客户端向服务端发起流创建请求后，双方各自维护一个 < 流 ID，收发通道 > 的哈希表。应用数据被拆分为多个流消息，每个消息在发送时都会被打上流 ID 的标签。接收方根据消息的流 ID，将其分发到对应的接收通道。通过这种方式，CRPC 在单个 TCP 连接之上实现了多个虚拟信道，避免了频繁建连的开销，并能够适应分布式数据加载过程中多种通信应用，如批量数据传输、心跳同步等。

跨语言互操作

CRPC 为 Rust 提供了客户端和服务端的支持，并利用 Pyo3 库^[57]，采用了 FFI (Foreign Function Interface) 机制，将 Rust 实现的 RPC 能力封装并暴露给 Python。Pyo3 是一个用于开发 Rust 和 Python 之间互操作性的库，它提供了在 Rust 中定义和操作 Python 对象的能力，并自动生成相应的 Python 模块和类型。通过 Pyo3，CRPC 可以将 Rust 中定义的数据结构和函数无缝地暴露给 Python，实现跨语言的函数调用和数据传递。

然而，由于 Rust 与 Python 的内存管理模型差异，通过 FFI 传递复杂对象时，可能会产生额外的内存拷贝开销。为了解决这个问题，CRPC 在与 Python 互操作时，利用了 Python 的 Buffer Protocol (PEP 3118) 协议^[58]。Buffer Protocol 允许在不同的 Python 对象之间共享内存，而无需进行数据拷贝。CRPC 定义了 PyCRPCBuffer 对象，该对象在 Rust 端托管内存，并通过 Buffer Protocol 将内存

指针和长度信息暴露给 Python，使得 Python 可以直接访问 Rust 端的内存，避免了不必要的数据复制。

在深度学习的数据加载场景中，计算任务的输入输出通常具有多样性，难以事先确定明确的类型信息。为此，CRPC 采用 Python 的 Pickle 协议在传输过程中序列化和反序列化数据对象。Pickle 是一种支持广泛 Python 对象的序列化方案，它可以将 Python 中的各种对象转换为字节流，并能够从字节流中恢复出原始对象。CRPC 进一步利用了 Pickle Protocol 5 引入的 out-of-band 特性^[59]，设计了 PyPickleData 对象。它允许 Pickle 在序列化过程中，将多维数组的大型对象的二进制表示直接写入 Rust 端预分配的内存中，减少了不必要的数据拷贝。通过 Pyo3 和 Pickle 的结合，CRPC 实现了高效、灵活的 Python 对象序列化和传输。

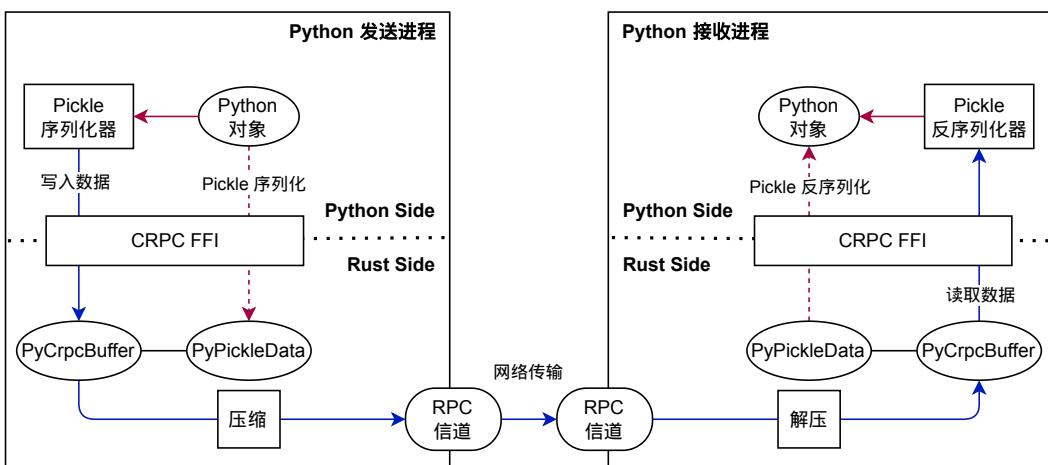


图 4-5 CRPC 中 Python 对象的传递过程

此外，为了进一步提高较大数据对象的网络传输效率，CRPC 在 PyPickleData 的基础上，增加了基于 Snappy 的数据压缩机制。Snappy 是一种高速的数据压缩库，它在提供合理压缩比的同时，保持了极低的 CPU 和内存开销。在传输前，CRPC 自动对序列化后的数据进行 Snappy 压缩，并在接收端进行透明解压。这在很大程度上优化了系统的网络带宽利用率，特别是在传输大型数据对象时，可以减少网络传输时间和成本。图4-5展示了 CRPC 中 Python 对象在 RPC 通信过程中的处理流程。两个远程的 Python 进程之间交换 Python 对象时，需要经历序列化、压缩、通信、解压缩、反序列化等一系列步骤。

4.3.2 数据加载协调服务

数据加载协调服务作为 DPFlow 的控制平面，负责管理数据加载流和数据集的元信息，协调其他组件的工作流程。图4-6 展示了数据加载协调服务的结构。数据加载协调服务通过 RPC 通信模块接收外部 RPC 请求，负载数据加载流注册、查询、访问和删除等操作。当用户提交一个新的训练任务时，协调服务首先会解析任务的数据加载需求，创建相应的数据加载流对象，并通过状态管理模块存储到外部数据库中，如 KV 数据库；然后，它会通过平台交互模块，访问弹性容器实例 ECI 的 API，动态创建一个数据加载路由服务实例，并将相应的地址信息返回给用户。在训练过程中，协调服务会定时与系统各组件进行心跳同步，监控组件的状态，及时响应故障和异常。

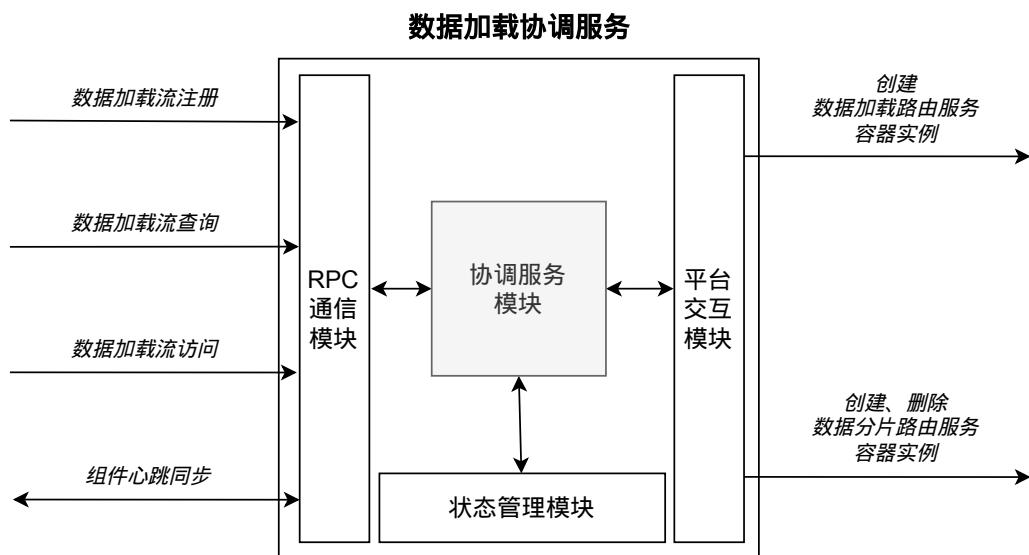


图 4-6 数据加载协调服务的结构图

在 DPFlow 系统中，每个数据加载流都有其独立的生命周期，从创建到销毁的全过程由数据加载协调服务统一管理。协调服务内部维护了一个数据加载流实体模型，该模型记录了数据加载流的关键元信息，如名称、版本号、处理阶段定义等。其中，数据加载流名称作为全局唯一标识符，用于区分不同的数据加载流；版本号则支持数据加载流定义的灵活升级，系统总是使用版本号最大的定义作为最新版本。

数据加载流的生命周期如图4-7所示，主要包括创建状态、就绪状态、访问状态、闲置状态和删除状态。各个状态通过外部组件的 RPC 请求或者时间事件进行驱动转移。

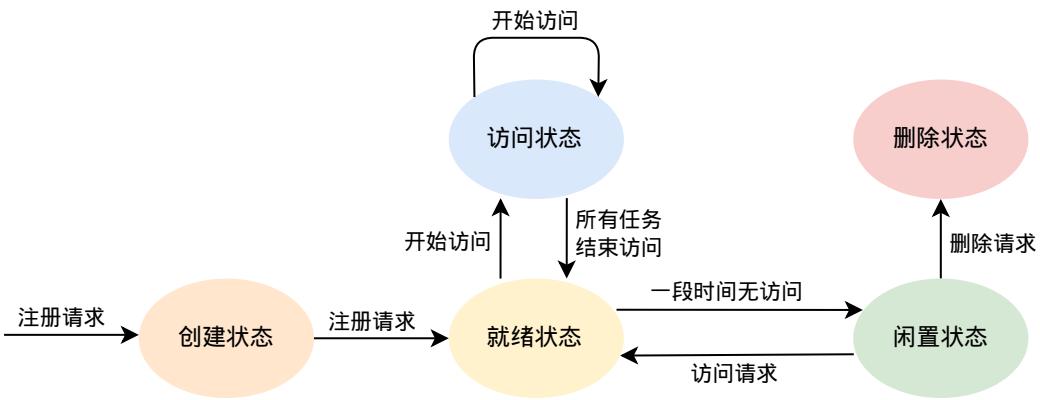


图 4-7 数据加载流生命周期状态转换图

- 创建状态。**当训练函数发起一个新的注册请求时，且该数据加载流未被注册，数据加载流进入至创建状态。当用户提交一个新的训练任务时，训练函数会向协调服务发送数据加载流注册请求。该请求携带了完整的数据加载流配置信息。协调服务收到请求后，会检查系统中是否已存在同名的数据加载流。如果不存在，则直接注册；如果存在，则比较新旧版本号，根据情况进行新建或更新。对于版本号较小的注册请求，协调服务会直接拒绝注册，并返回相应的错误信息。对于版本号相同的注册请求，协调服务会进一步校验其处理阶段定义的兼容性，以确保数据加载流的一致性。
- 就绪状态。**当数据加载流已注册，且对应的数据加载路由服务已经创建成功时，系统进入就绪状态。在就绪状态中，能够接受训练函数发起的访问请求。
- 访问阶段。**当训练函数实际发起训练时，会对数据加载流发起访问请求，此时进入访问状态。这一过程视作创建了一个新的数据加载流访问任务。只要有一个训练函数在进行访问，数据加载流会一直处于访问状态。当所有任务访问结束后，数据加载流进入就绪状态。
- 闲置状态。**当数据加载流位于就绪状态，且一段时间内无任何访问请求，数据加载流将进入闲置状态，此时协调服务将会尝试销毁路由服务实例。如果数据加载流在闲置状态中，有新的训练函数发起访问请求，数据加载流尝试创建数据加载路由服务示例，并进入就绪状态。这一过程实现了数据加载路由服务的按需使用，能够节省系统资源。
- 删除状态。**当数据加载流位于闲置状态，且用户发起了删除请求，数据加载流将进入删除状态。此时协调服务将会尝试删除数据加载流的相应元信

息。

数据加载协调服务通过维护数据加载流的状态和生命周期，保证系统能够正常运行。

4.3.3 数据加载路由服务

数据加载路由服务负责管理单个数据加载流的数据访问和状态，并维护一组与之对应的分片服务实例。图4-8展示了数据加载路由服务的结构。数据加载路由提供了数据访问、分片路由信息获取、数据采样信息获取的 RPC 接口。当路由服务收到来自训练函数的分片路由信息获取 RPC 请求时，会通过分片路由管理模块，根据请求的元信息（如批次索引）查询内部的路由表，得到相应数据分片所在的分片服务地址，然后将该地址返回给请求方。请求方可直接与分片服务通信以获取实际的数据。路由服务本身并不处理具体的数据，而是只维护相关控制状态，如路由信息和数据加载流访问任务信息。同时，路由服务还管理了多任务的采样状态，训练函数发送数据访问开始 RPC 请求，路由服务通过多任务协同采样模块启动采样器，并在每次的数据采样信息获取请求中，返回当前任务的采样信息，以及更新采样器状态。当训练函数完成训练后，向路由服务发送数据访问结束的 RPC 请求，路由服务会销毁对应的采样器。

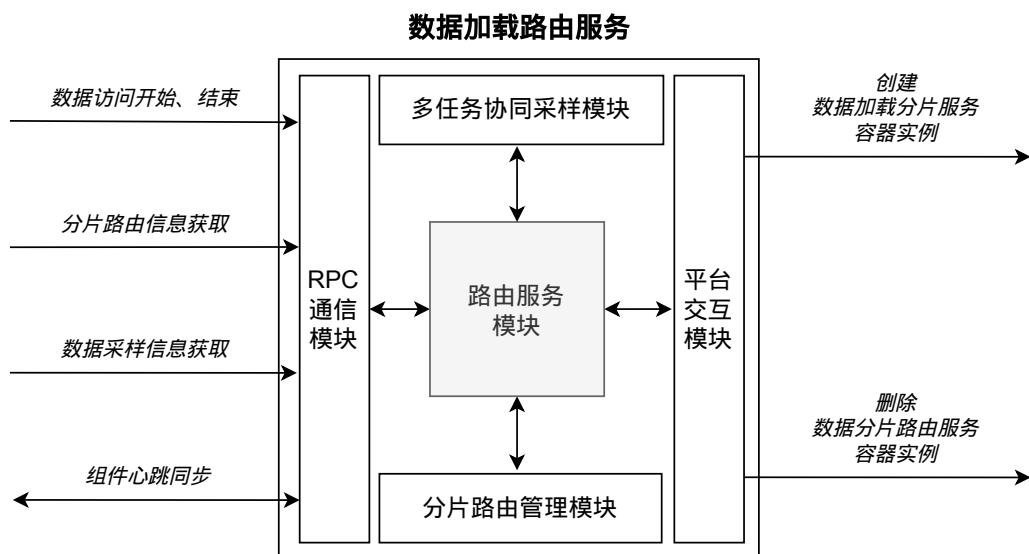


图 4-8 数据加载路由服务的结构图

在 DPFlow 系统中，为了实现可扩展的分布式数据加载，数据加载流被划分

为多个数据分片，每个分片由一个独立的数据加载分片服务实例负责管理。数据加载路由服务则通过平台交互模块动态创建或者删除数据加载分片服务容器实例，以及在训练函数和分片服务之间管理路由信息。训练函数在访问数据加载流之前，需要与数据加载路由服务通信，获取对应的路由信息，然后按照路由信息直接与分片服务实例进行通信。

在系统中，路由信息可能会发生变化。为了确保路由信息的及时性，每个路由信息按照时间划分成了多个世代（Epoch），每次路由信息更新时，则递增世代信息。训练函数需要定时与路由服务同步相关信息，并检查世代信息。如果训练函数的世代较老，则需要重新获取最新的路由信息。在训练函数与分片服务通信期间，也需要携带相应的世代信息。如果世代不一致，分片服务需要返回对应的错误信息，训练函数需要重新与路由服务同步，更新世代相关的配置。

当一个训练函数准备访问数据加载流时，首先会与数据加载流路由服务同步路由信息，并缓存在本地内存中。之后的数据访问请求，根据本地内存的路由信息，按照样本索引计算出待访问的分片服务实例以及需要访问的样本索引范围。最后向这些分片服务实例发送数据访问请求，并行异步拉取数据。

通过分片与路由机制，DPFlow 能够灵活地应对不同规模的数据集和数据加载的工作负载。当数据集增大或数据加载中的动态预处理逻辑变得复杂时，可以通过增加分片数量和分片服务实例来线性扩展整个系统的吞吐能力。同时，通过世代机制，保障了路由信息在整个系统中的及时性和一致性。目前，DPFlow 的分片服务实例数量需要进行预先设置，而不能进行动态扩缩容。未来，DPFlow 还将支持根据训练函数数量、数据集规模等因素自动进行分片服务实例的水平扩缩容。

数据访问模式

在 DPFlow 系统中，训练函数通过访问数据加载流来获取加载好的训练数据。数据加载流提供了两种主要的数据访问模式：索引访问和遍历访问。

索引访问模式允许训练函数按照样本的索引批量读取数据。这种模式类似于键值数据库的随机读取操作。为了获得更高的性能，每次数据访问都是以批次的索引为单位。图4-9展示了索引访问模式的时序图。训练函数首先根据本地缓存的路由信息，按照分片服务实例对索引信息进行分组，并并行向多个分片

服务实例发送数据访问请求，等待返回。分片服务实例根据具体的索引号检索相应的样本数据，如果数据已经加载完毕，则直接返回；否则触发加载过程，待完成后返回结果。最终，训练函数聚合多个分片服务实例返回的数据，递交给出上层应用。

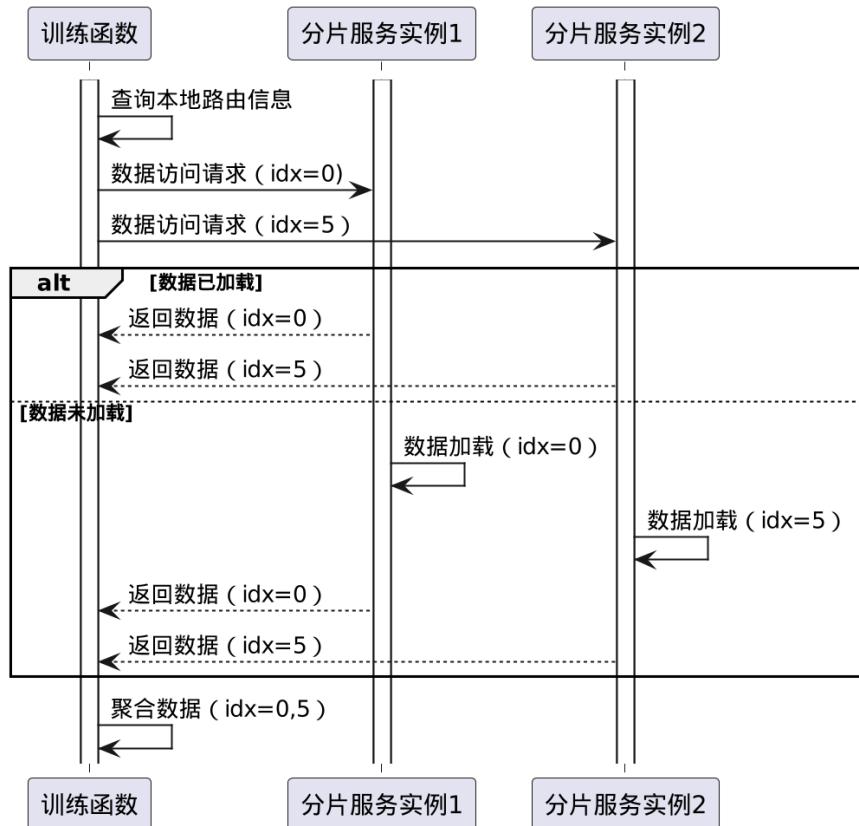


图 4-9 索引访问模式的时序图

遍历访问模式则允许训练函数按照迭代顺序批量读取数据，类似于迭代器，迭代顺序由采样器决定。与索引访问不同，遍历访问无需显式指定样本索引，而是由数据加载路由服务维护遍历状态。在数据加载路由服务中，会维护多个任务的采样器，并使用协同采样算法 DynaGCS 提高多任务之间的数据共享程度。图4-10展示了遍历访问模式的时序图。每次遍历时，训练函数向路由服务发起请求，并获取未来将访问到的数据索引，然后训练函数按照索引访问模式利用索引访问数据加载流。深度学习训练通常会多次遍历同一个数据加载流，训练函数在遍历完一个 Epoch 后，需要向数据加载路由服务发起新一轮的请求，重设采样器状态。

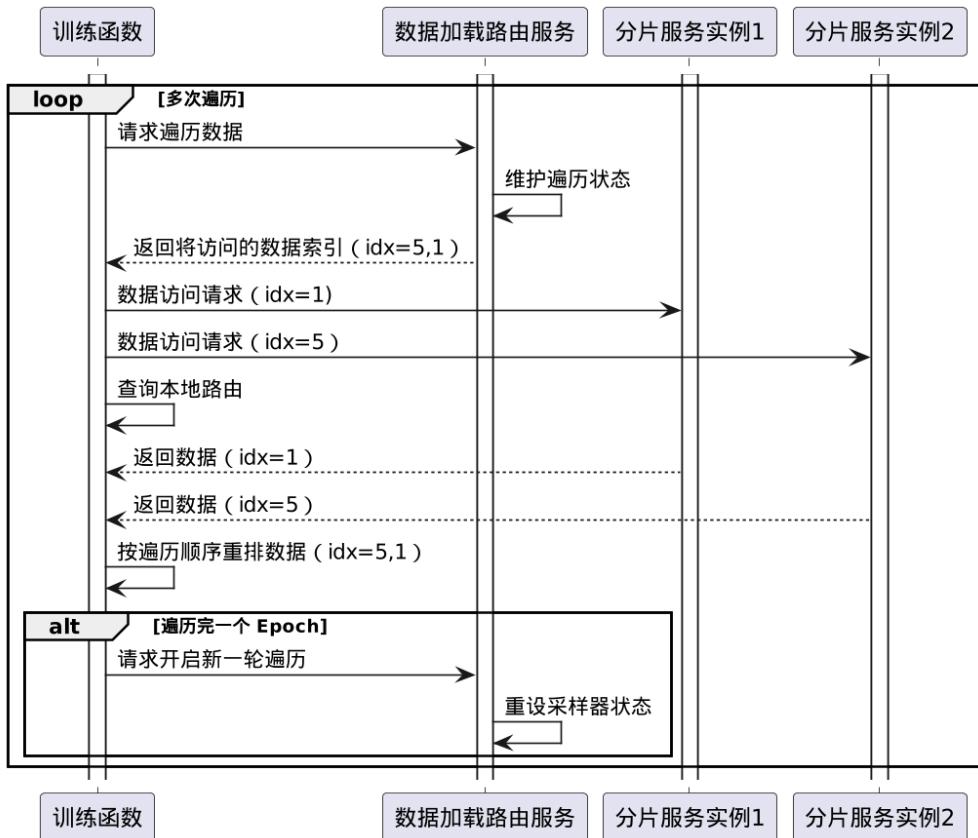


图 4-10 遍历访问模式的时序图

4.3.4 数据加载分片服务

数据加载分片服务是执行数据加载的工作单元。图4-11展示了数据加载分片服务的结构，每个分片服务实例负责管理训练数据集的一个子集，并维护一个任务队列和一组数据加载函数，用于并行处理数据。分片服务提供数据预取、数据访问、数据加载任务同步等 RPC 接口。预取与缓存管理模块实现了3.4.2节设计的二级缓存机制以及3.4.3设计的 ALSD 缓存策略。并行加载任务管理模块实现了3.4.1设计 QMOWS 动态任务调度策略，并通过平台交互模块访问阿里云函数计算 FC API，动态创建和删除数据加载函数实例。

在数据加载分片服务中，预取与缓存机制被进一步整合为预取与缓存对象池。当分片服务收到数据请求时，首先会检查预取与缓存对象池，若命中则直接返回；否则，会将计算任务添加到任务队列中。数据加载函数会发送数据加载任务同步 RPC 请求，同步任务队列，然后从队列中拉取任务并执行加载过程，加载后的结果会上传给分片服务。之后，分片服务将加载的结果进行缓存，并返回给请求方。预取与缓存是 DPFlow 中的重要优化机制，预取能够掩盖分布式环境

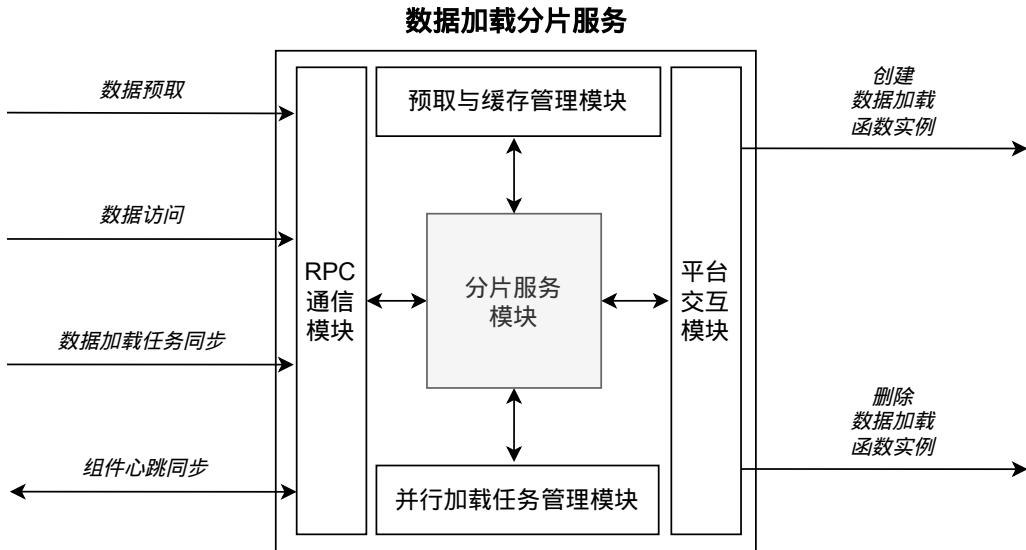


图 4-11 数据加载分片服务的结构图

下的数据传输与计算延迟，从而提升数据加载流的端到端吞吐量。通过缓存机制，可以复用数据加载结果，减少重复的 I/O 操作和计算，进而提高资源利用率和系统性能。

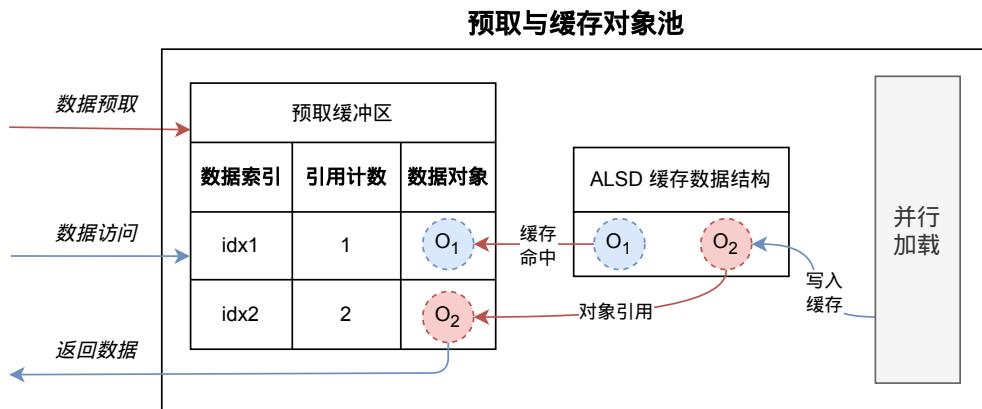


图 4-12 预取与缓存对象池的结构图

图4-12展示了预取与缓存对象池的结构，包括预取缓冲区和缓存数据结构两部分。管理对象时，使用智能指针进行统一管理，在预取缓冲区和缓存数据结构中共享数据对象的指针，以降低内存开销。预取缓冲区管理了多个任务的预取数据，采用哈希表实现，维护数据索引到数据对象的映射关系，并记录引用计数信息。进行预取操作时，相关数据的引用计数增加；在读取操作完成时，引用计数相应减少。若某数据的引用计数降至零，则该数据会从预取缓冲区中移除。缓存数据结构则实现了 3.4.3 节描述的 ALSD 缓存策略，优化了数据存储和访问效率。

4.4 本章小结

本章描述了 Serverless 分布式数据加载系统 DPFlow 的设计与实现。首先介绍了阿里云 Serverless 平台，包括阿里云网络与存储服务、函数计算平台 FC 与弹性容器实例 ECI 等产品。接着阐述了系统整体的架构设计，说明了各个组件在系统中的功能。最后详细介绍了系统中各个组件，描述了它们如何应用前文设计的优化方法。

第五章 实验设计与评估

本章将评估本文提出的面向 Serverless 深度学习训练的分布式数据加载方法和系统 DPFlow 的有效性。实验分为三个部分：核心算法仿真实验、系统功能验证实验和系统性能评估实验。

首先，核心算法仿真实验将验证 ALSD 缓存置换策略和 DynaGCS 协同采样算法的效果。我们构建了模拟的训练任务，通过缓存命中率、采样重叠率等指标，定量评估算法在提高缓存利用率和任务间数据共享方面的优势。同时，我们还设计了统计学验证实验，检验 DynaGCS 采样算法的随机性质，确保其在提高数据共享的同时，不影响单任务的数据分布。

其次，系统功能验证实验将测试 DPFlow 系统在支持不同深度学习任务的适配能力。我们选取了图像分类、目标检测、语音识别三个典型场景，并采用 PyTorch 框架实现训练流程，验证 DPFlow 的兼容性。此外，我们还通过训练过程的收敛性曲线和验证集的精度曲线，评估 DPFlow 对训练效率和精度的影响。

最后，系统性能评估实验将测试 DPFlow 在数据传输和端到端数据加载加速方面的效果。评估 DPFlow 的 CRPC 通信模块在提升数据传输性能方面的优化效果。在端到端训练场景中，使用 Mini-ImageNet 数据集和 ResNet-18 模型，记录数据加载时间、训练耗时、资源利用率等关键指标，并与现有的 PyTorch DataLoader 进行比较，展示 DPFlow 的性能优势。

通过以上三方面实验，我们力图客观地评估 DPFlow 优化系统的有效性。一方面，通过仿真实验，验证核心算法设计的合理性；另一方面，通过功能测试和性能评估，证明系统实现满足功能和性能上的要求。最终目标是表明：DPFlow 能够帮助用户实现 Serverless 深度学习训练的数据加载过程，并实现性能优化。

5.1 实验配置

本文的实验环境基于阿里云函数计算（FC）平台和弹性容器实例（ECI）服务。FC 平台提供了 FaaS 形态的通用计算能力，支持自定义的 CPU/GPU 规格。我们使用 FC 部署深度学习训练函数和数据预处理函数。ECI 服务提供了基于 Serverless 容器的弹性计算资源，我们使用 ECI 部署 DPFlow 的各个系统组件。

在存储方面，我们使用对象存储 OSS（Object Storage Service）存放训练数据集，使用文件存储 NAS（Network Attached Storage）存储训练代码、日志等。FC、ECI 与 OSS、NAS 之间通过高速 VPC 网络连接，保证低时延、高吞吐的数据传输。

实验采用了以下数据集：

- 图像分类任务：使用 Mini-ImageNet 数据集，从 ImageNet-1K 数据集^[48]中随机选取 100 个类别的子集，共 60000 张图片。
- 目标检测任务：使用 MS COCO 2017 数据集^[49]，共 118287 张图片，涵盖 80 个目标类别。
- 语音识别任务：使用 LibriSpeech 数据集^[60]，从中选取 100 小时的英语演讲语音子集。

所有数据集均按照3.3.1节提出的静态数据集存储规范进行了预处理，上传到 OSS 存储中。

硬件环境方面，使用阿里云函数计算平台与弹性容器实例平台。软件环境方面，操作系统采用 Ubuntu22.04 Docker 镜像，使用 Python 3.10，内置 CUDA、cuDNN 等 GPU 计算库。深度学习框架为 PyTorch 2.2。表5-1总结了实验的软硬件环境配置。

5.2 核心算法仿真实验

我们开发了一个轻量级的算法仿真模块，用于评估 ALSD、DynaGCS 等核心算法的独立效果。仿真模块基于 Python 实现，主要包含任务生成器、数据采样器、缓存管理器等组件。任务生成器负责模拟多个训练任务，控制它们的数据集索引范围、批大小、持续时长等属性；数据采样器则实现了不同的数据采样方

表 5-1 实验软硬件环境配置

| 类别 | 配置 |
|---------|----------------------------|
| 操作系统 | Ubuntu 22.04 |
| CPU | 阿里云 vCPU (8 vCPU) |
| GPU | NVIDIA Tesla A10 (24GB 显存) |
| 内存 | 16 GB |
| 磁盘 | 512MB ESSD |
| Python | 3.10 |
| CUDA | 11.8 |
| cuDNN | 8.9 |
| PyTorch | 2.2 |

式，如 PyTorch 原生的随机采样、CoorDL 采样、Joader 采样等；缓存管理器则在不同缓存置换策略下，模拟数据缓存的运作机制。

在仿真数据生成中，我们构建了一批模拟的训练任务。每个任务随机选择数据集的一个子区间作为采样范围，并指定了期望的批大小和训练时长。我们假设每个任务的数据预处理时间与采样得到的索引号成正比，从而模拟了任务对数据的依赖关系和计算强度差异。通过仿真实验，可以方便地评估各种缓存算法和采样算法的性能表现，减少真实环境下因为硬件环境或者网络通信等造成的影响。

5.2.1 缓存置换策略 ALSD 效果评估

本实验比较了 ALSD 与 LRU、LFU 等传统缓存置换策略，以及 MinIO、RefCnt 等面向深度学习的缓存优化策略的性能。实验设置了不同的缓存容量，分别统计不同的缓存策略在不同采样器下的缓存命中率。

图5-1展示了在 4 个使用相同数据加载过程且训练速度相同的任务的场景下，采用 Pytorch 采样算法时，不同缓存置换策略随缓存容量变化的命中率曲线。结果表明，ALSD 在各种缓存容量配置下都取得了最优的命中率。当缓存容量为数据集大小的 50% 时，ALSD 的命中率达到 57.05%，分别比 LRU、LFU 高出 26.65 和 27.8 个百分点，比 RefCnt、MinIO 高出 3.2 和 19.55 个百分点，与使用精确采样序列信息的 Use Distance 算法性能相当。这是因为 ALSD 和 Use Distance 算法能够更好地捕捉任务访问数据的未来序列特征，从而做出更优的缓存决策。在

多任务并发训练的场景下，ALSD 和 Use Distance 通过加权平均综合考虑了所有任务的数据访问位置，有效区分出不再可能被再次访问的数据，以及大概率在更近未来会被访问的数据，从而提高了缓存命中率。

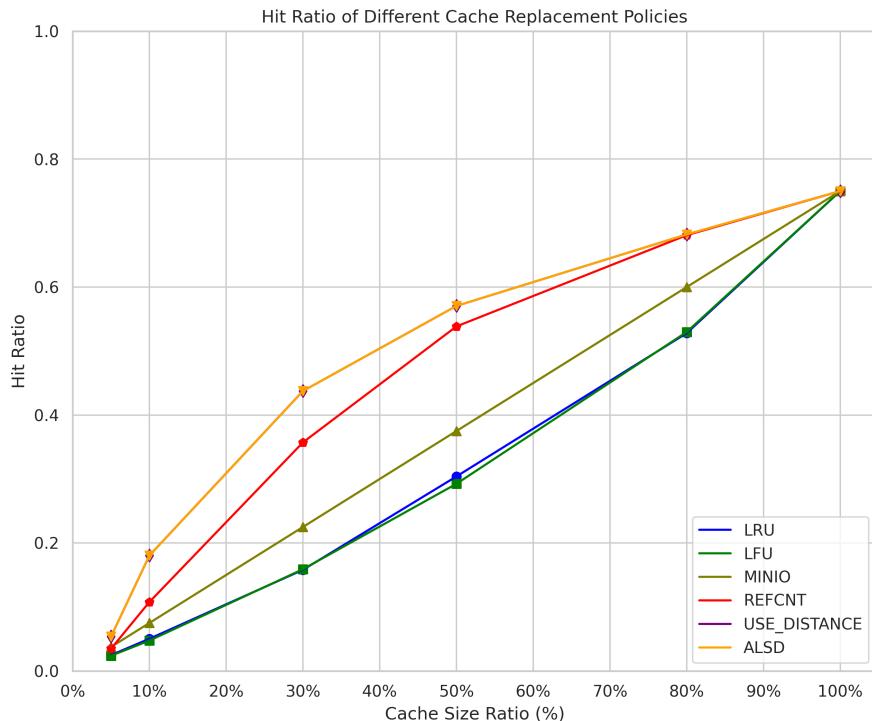


图 5-1 不同缓存置换策略的命中率对比

对于一些无法提供准确未来采样序列的采样器，ALSD 能够比 Use Distance 取得更好的缓存效果。DynaGCS 和 Joader 采样算法都无法提供完整的采样序列，而当任务启动时间不同时，所有任务的采样序列也会发生变化，ALSD 能够有效应对这种情况。

图 5-2 展示了采用 DynaGCS 和 Joader 采样算法时，ALSD 和 Use Distance 算法随缓存容量变化的命中率曲线。可以看出，对于 Joader 和 DynaGCS，在不同的缓存容量下，ALSD 的缓存命中率都优于 Use Distance 策略。使用 Joader 策略时，在缓存容量为 50% 时，ALSD 的命中率达到 60.6%，比 Use Distance 高出 19.6 个百分点。这得益于 ALSD 可以利用数据采样过程的先验信息，例如采样器提供的数据访问的概率权重信息，估算出近似的采样距离。在无法确定未来序列信息的场景下，ALSD 能够更加精确地预测未来数据访问的趋势。

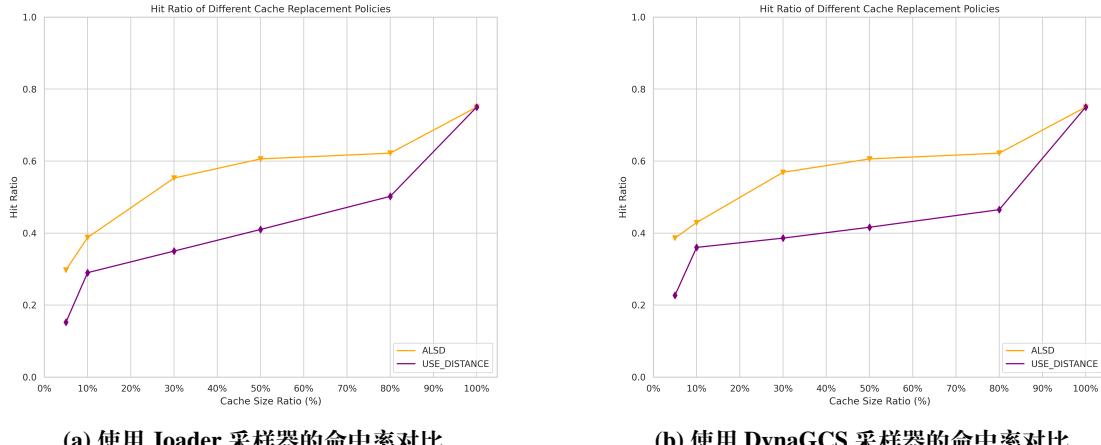


图 5-2 ALSD 与 Use Distance 的性能对比

5.2.2 采样算法 DynaGCS 效果评估

本实验固定缓存置换策略为 ALSD，固定缓存容量为数据集大小的 5%，评估 DynaGCS 相对于 PyTorch 原生随机采样、CoorDL、Joader 等采样方法的优势。我们重点考察了采样重叠率，通过在较小缓存容量下的缓存命中率来反映一个时间段内的采样重叠度，即不同任务采样得到的训练数据之间的重合度。

实验选取了 4 种任务案例，每个案例包含 4 个使用相同数据加载过程的训练任务。针对训练是否同时开始以及训练速度是否相同，设置了四种场景，如表 5-2 所示。

表 5-2 采样效果实验场景设置

| 场景名称 | 启动时间 | 训练时间 |
|--------|-------|------|
| same1 | 同时启动 | 相同 |
| same2 | 不同时启动 | 相同 |
| speed1 | 同时启动 | 不同 |
| speed2 | 不同时启动 | 不同 |

图 5-3 展示了不同采样算法下的采样重叠率。可以看出，DynaGCS 在各种场景下的重叠率最高，在任务同时开始且速度相同时达到了 75%，而 PyTorch 随机采样的重叠率仅为 9%。同时，DynaGCS 能够更好地适用于启动时间和速度不同的场景。在任务启动时间不同且训练速度不同的场景下，DynaGCS 的采样重叠率为 15%，相比 Joader 高出 6 个百分点，相比 CoorDL 高出 12 个百分点。这表明，DynaGCS 能够更好地协调不同任务的数据采样过程，使它们在相似的时间

范围内采样到相似的数据子集，从而最大化数据共享的效果。

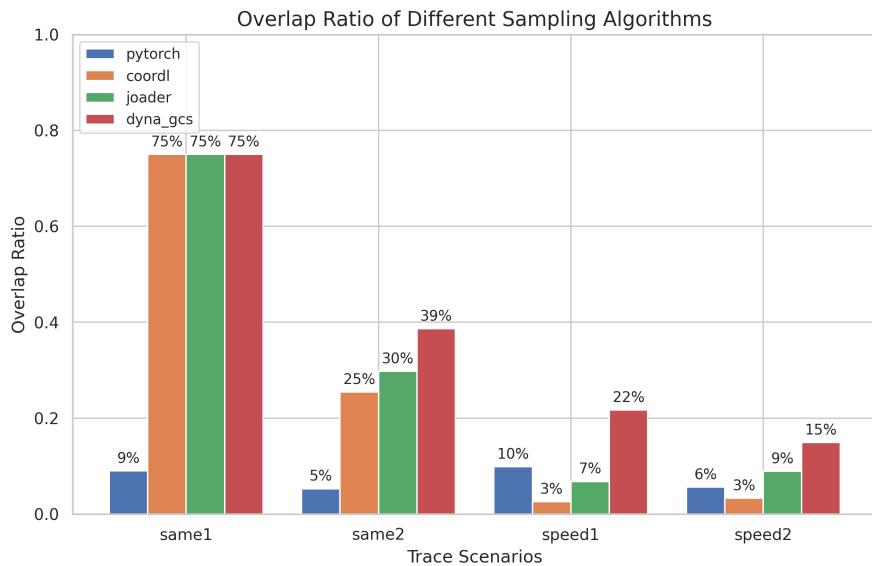


图 5-3 不同采样算法和任务场景的重叠率对比

5.2.3 采样算法 DynaGCS 随机性质验证

为了确保 DynaGCS 采样算法能够在提高不同任务数据共享的同时，不影响单个任务的数据分布，我们需要严格验证其采样结果的随机性。一个合格的采样器需要满足以下两个条件：

- 在每个 epoch 内，采样得到的数据索引是唯一的，不存在重复采样；
- 在跨 epoch 的长期采样过程中，每个数据索引被采样到的概率应当服从均匀分布，不能出现个别数据被显著偏袒或忽略的情况。

为验证 DynaGCS 采样算法是否满足以上条件，我们设计了一个统计学验证实验。具体而言，假设数据集共有 1000 个样本，我们利用 DynaGCS 采样器模拟进行 10000 次 epoch 的采样，得到每个 epoch 的采样序列。然后，我们统计每个索引位置 (0~999) 在这 10000 个采样序列中出现的频率，通过卡方拟合度检验来判断索引出现的频率分布是否服从均匀分布。

卡方检验的原理如下：对于每个索引位置，我们首先计算其实际出现频率与理论均匀分布下的期望频率之间的差异，然后将所有位置的差异平方相加，并除以期望频率，由此得到一个卡方统计量。如果实际分布与均匀分布吻合，则卡

方统计量应当服从自由度为 999 的卡方分布。据此，我们可以计算出该卡方统计量对应的 p 值，p 值越大意味着实际分布偏离均匀分布的可能性越小。

我们将每个索引位置的 p 值记录下来，构成一个长度为 1000 的 p 值序列。为了严格控制整体的假阳性率，我们采用 Benjamini-Hochberg (BH) 方法对 p 值序列进行多重比较校正。BH 法首先对 p 值序列进行升序排列，然后根据排序的顺序，将每个 p 值与一个递增的阈值序列进行比较，从而得到校正后的 p 值。校正后的 p 值小于显著性水平 (取 $p=0.05$) 的索引位置，即被认为是与均匀分布存在显著差异的位置。

我们分别对 PyTorch 默认采样器、CoorDL、Joader 和 DynaGCS 的采样结果进行了上述验证，图5-4展示了它们校正后的 p 值分布情况。可以看出，对于 DynaGCS 算法，所有索引位置的 p 值都显著大于 0.05，不存在于均匀分布显著差异的位置。这一结果与 PyTorch 默认采样器和 CoorDL 十分接近，表明 DynaGCS 采样得到的数据子集能够很好地近似均匀分布。

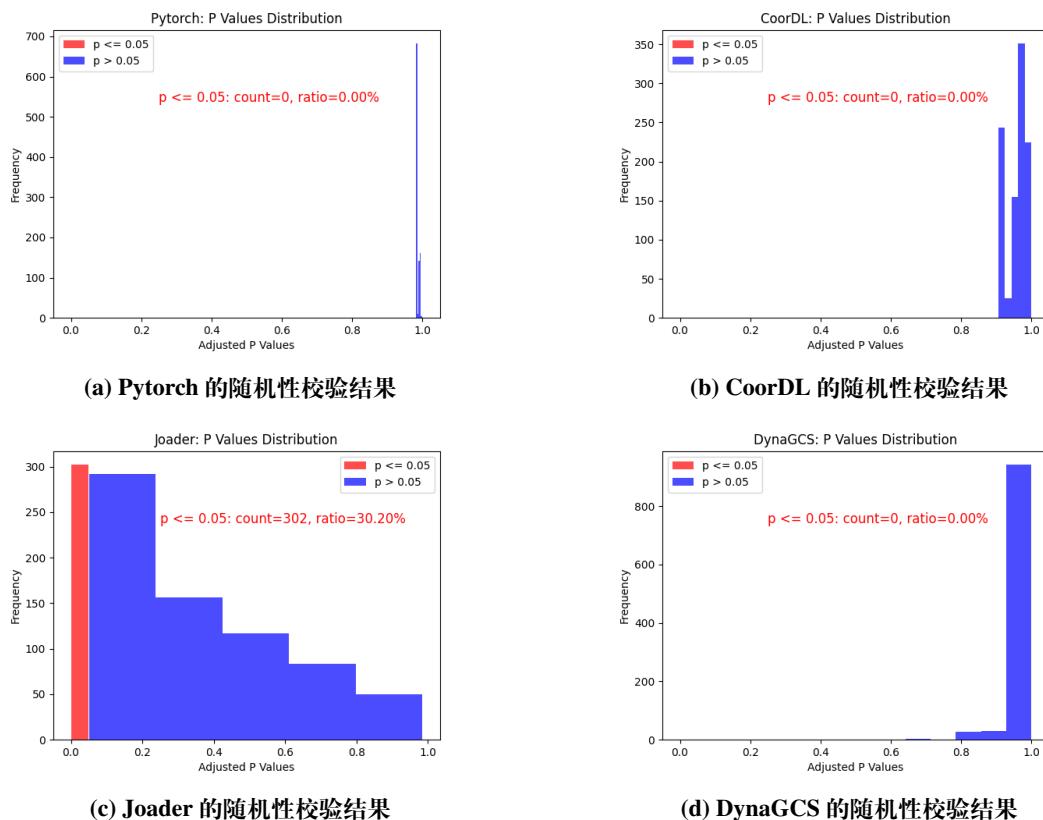


图 5-4 不同采样算法的随机性检验结果

相比之下，Joader 的 p 值整体偏小一些，有近 30% 的位置未通过显著性检验，意味着这些位置对某些数据索引有较多的偏袒。这可能是因为 Joader 为了

提高任务间的数据重叠度，在一定程度上放松了单任务内的均匀性要求。但即便如此，DynaGCS 的采样重叠性仍优于 Joader。

该统计学验证实验表明，DynaGCS 采样算法能够持续产生服从均匀分布的采样序列，其随机性与理想的 PyTorch 采样器相当，同时优于已有的任务协同采样方案 Joader。这进一步证实了 DynaGCS 在提高数据共享的同时，并没有损害单任务的训练质量，是一种兼顾效率和效果的采样优化策略。

5.3 系统功能验证实验

功能验证实验在阿里云 Serverless 平台上进行。我们使用函数计算 FC 服务分别部署了图像分类、目标检测、语音识别三类深度学习任务，每类任务采用了不同的主流模型。其中，图像分类采用经典的 ResNet-18 模型^[61]，目标检测采用 SSD 模型^[62]，语音识别采用 RNN-T 模型^[63]。

数据集方面，图像分类任务使用 Mini-ImageNet 数据集；目标检测采用 MSCOCO 2017 数据集^[49]；语音识别使用 LibriSpeech 数据集^[60]。

此外，我们同时测试了 DPFlow 数据加载方法对收敛性和精度的影响，以验证 DPFlow 不干扰正常的模型训练过程。

5.3.1 深度学习任务的适配性验证

为了验证 DPFlow 对不同深度学习任务的适配性，我们分别在图像分类、目标检测、语音识别三个任务上使用 DPFlow 进行训练，并检查其数据加载和训练执行的正确性。表5-3显示了 DPFlow 三类任务的适配情况。

表 5-3 DPFlow 对不同深度学习任务的适配情况

| 任务类型 | 数据集 | 模型 | 适配结果 |
|------|---------------|-----------|------|
| 图像分类 | Mini-ImageNet | ResNet-18 | 支持 |
| 目标检测 | MSCOCO | SSD | 支持 |
| 语音识别 | LibriSpeech | RNN-T | 支持 |

实验结果表明，DPFlow 能够支持三类任务的数据加载过程。不同任务的数据加载的日志如图5-5所示，日志中打印了三种任务的数据加载各个阶段，并开始模型训练。这证实了 DPFlow 可以有效适配主流深度学习任务。

```
[INFO] <TrainClsResnetTask> running task
    TrainClsResnetTask, use config
[INFO] <TrainClsResnetTask> Using Train Flow:
Dataset(core/mini-imagenet:v1:train)
    -> Map(transform_label) -> Map(image_decode)
    -> Map(random_resized_crop) -> Map(random_hflip)
    -> Map(to_tensor) -> Map(normalize)
[INFO] <TrainClsResnetTask> start training model=
    resnet18
```

(a) 图像分类任务的数据加载日志

```
[INFO] <TrainOdSsdTask> running task TrainOdSsdTask,
    use config
[INFO] <TrainOdSsdTask> Using Train Flow:
Dataset(core/mscoco:v1:train2017)
    -> Map(decode) -> Map(transform_target)
    -> Map(to_tensor) -> Map(random_hflip)
[INFO] <TrainOdSsdTask> start training model=
    ssdlite320_mobilenet_v3_large
```

(b) 目标检测任务的数据加载日志

```
[INFO] <TrainAsrRnntTask> running task
    TrainAsrRnntTask, use config
[INFO] <TrainAsrRnntTask> Using Train Flow:
Dataset(core/libri-speech:v1:train-clean-100)
    -> Map(piecewise_linear_log)
    -> Map(GlobalStatsNormalization)
    -> Map(transpose1) -> Map(freq_masking1)
    -> Map(freq_masking2) -> Map(time_masking1)
    -> Map(time_masking2)
    -> Map(pad1) -> Map(transpose2)
[INFO] <TrainAsrRnntTask> Start training model=
    emformer_rnnt
```

(c) 语音识别任务的数据加载日志

图 5.5 不同任务上的数据加载日志

5.3.2 训练收敛性的验证

由于采样的随机性，DPFlow 随机采样序列与 Pytorch DataLoader 不完全相同，为了评估 DPFlow 对深度学习训练收敛性的影响，我们针对 Mini-ImageNet 图像分类任务，基于 ResNet-18 模型，分别采用 DPFlow 和 Pytorch DataLoader 数据加载方式进行 100 轮次的长时训练，并记录训练过程中的损失函数值和验证集精度。

图5-6展示了训练过程的损失曲线和精度的对比。总体而言，使用 DPFlow 加载数据的训练过程与 Pytorch DataLoader 十分接近，损失下降的速度和趋势基本一致，在训练后期，精度稳定在 67.67% 左右。这表明 DPFlow 数据加载方式不会对模型的训练效果造成影响，因而训练得到的模型泛化性能相当。

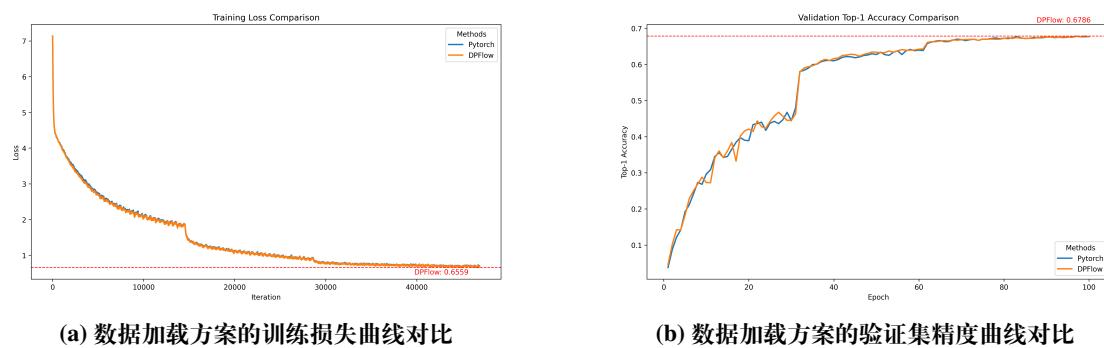


图 5-6 不同数据加载方案的训练过程指标对比

实验结果表明，DPFlow 能够不干扰深度学习训练的收敛过程，不损害模型的精度，为应用提供了功能上的保障。

5.4 系统性能评估实验

本节将聚焦 DPFlow 的端到端性能表现，通过数据传输、端到端训练等一系列实验，定量评估 DPFlow 在提高数据加载速度方面的优势。我们选取 PyTorch DataLoader 作为性能基准，测试评估 DPFlow 系统的性能。

5.4.1 数据传输性能评估

DPFlow 采用自研的 CRPC 通信框架，该框架支持 Rust 与 Python 互操作、内存零拷贝等特性，提升了数据传输效率。为了评估 DPFlow 在分布式预处理中

CRPC 的数据收发性能，我们构造了如下的测试场景：客户端向服务端发起请求，携带一个包含 5 个大小为 500KB 的 Python 字节对象的 Python 数组；服务端收到请求后，将其原封不动地返回给客户端；客户端通过反序列化验证接收到的对象内容的正确性。该测试场景涵盖了 DPFlow 运行时的各个环节，包括多个数据对象的序列化与反序列化、网络传输等。我们在本地 Loopback 网络环境下，使用 TCP 传输协议，多次进行收发操作，统计端到端时延和吞吐量等指标，并与 Python gRPC 框架（后文简称为 gRPC 框架）进行对比。

表5-4展示了 CRPC 和 gRPC 在数据传输性能上的对比结果。可以看出，CRPC 的端到端时延为 2.44ms，比 gRPC 的 4.54ms 低 46.35%；CRPC 的吞吐量为 8.33Gbps，相比 gRPC 的 4.58Gbps 提升了 1.82 倍。这得益于 Rust 的高效内存管理和与 Python 的零拷贝交互等优化手段。图5-7直观地呈现了两种框架在时延和带宽方面的性能差异，结果表明，CRPC 框架较 gRPC 框架性能更优。

表 5-4 不同 RPC 框架的数据传输性能对比

| RPC 框架 | 端到端时延 (ms) | 吞吐量 (Gbps) |
|--------|-------------|-------------|
| gRPC | 4.54 | 4.58 |
| CRPC | 2.44 | 8.33 |

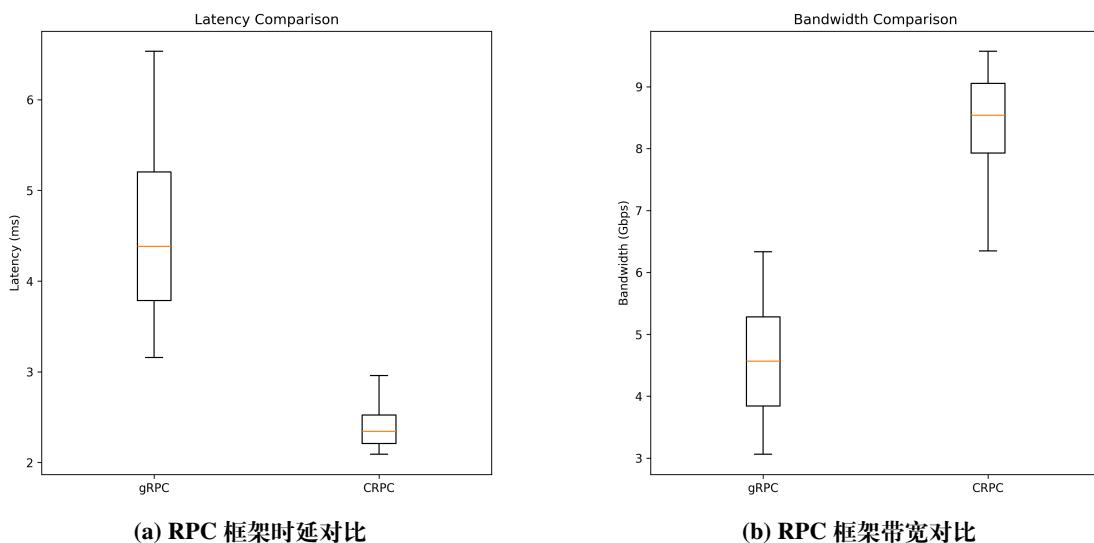


图 5-7 不同 RPC 框架的性能对比

5.4.2 数据加载性能评估

最后，我们评估了 DPFlow 在端到端训练场景下的整体性能表现。实验选取了图像分类 ResNet-18 模型在 ImageNet 数据集上的训练任务，并测试了使用 DPFlow 分布式数据加载与 Pytorch DataLoader 数据加载方法的性能对比。两种数据加载方法的并行度均设置为 8，这是阿里云中单个 Serverless GPU 函数可分配的 CPU 核心数上限，该上限限制了 Pytorch DataLoader 的最大并行度。对于 DPFlow 来说，由于采用分布式架构，不受该上限的影响，但为了公平比较，DPFlow 采用和 Pytorch DataLoader 一样的并发度配置。此外，本文还额外测试了 DPFlow 使用并行度 12 的情况，作为对比参考。

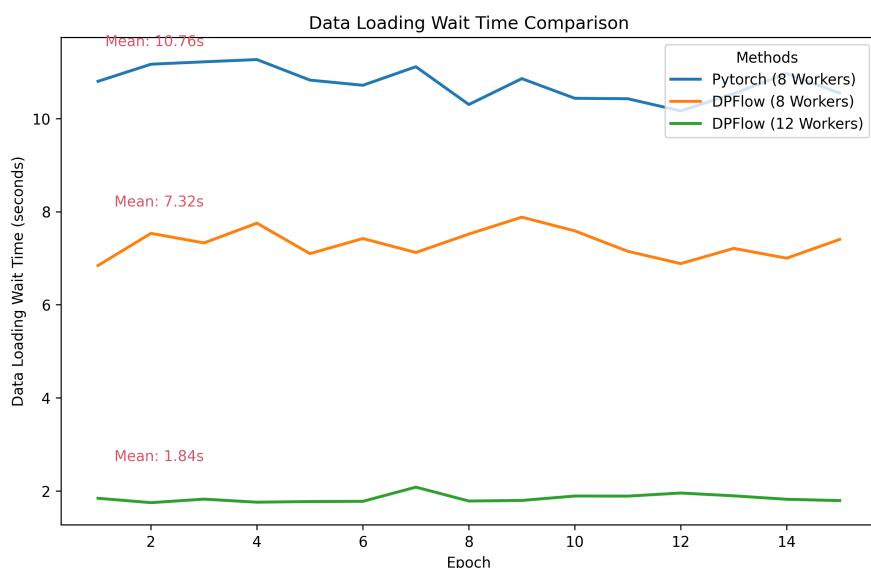


图 5-8 不同方案的数据加载等待时间对比

我们统计了以下 3 个性能指标：每个 Epoch 训练的数据加载等待时间、每个 Epoch 训练总时间以及每个 Epoch 的平均 GPU 利用率。其中，数据加载等待时间指的是模型等待训练数据的时间。在训练过程中，数据加载与模型训练形成异步流水线，理论上，如果数据加载速度大于模型训练速度，那么实验中统计到的数据加载等待时间应该为 0，训练总时间应该等于模型训练时间。但由于数据停滞的问题，数据加载平均速度可能会小于模型训练速度，此时，数据加载等待时间将大于 0。通过数据加载等待时间，可以衡量数据加载的速度。

首先，DPFlow 降低了训练的数据加载等待时间。结果如图5-8所示。在分布式预处理模式下，DPFlow 能够充分利用远程 CPU 资源，从而避免了占用训练函

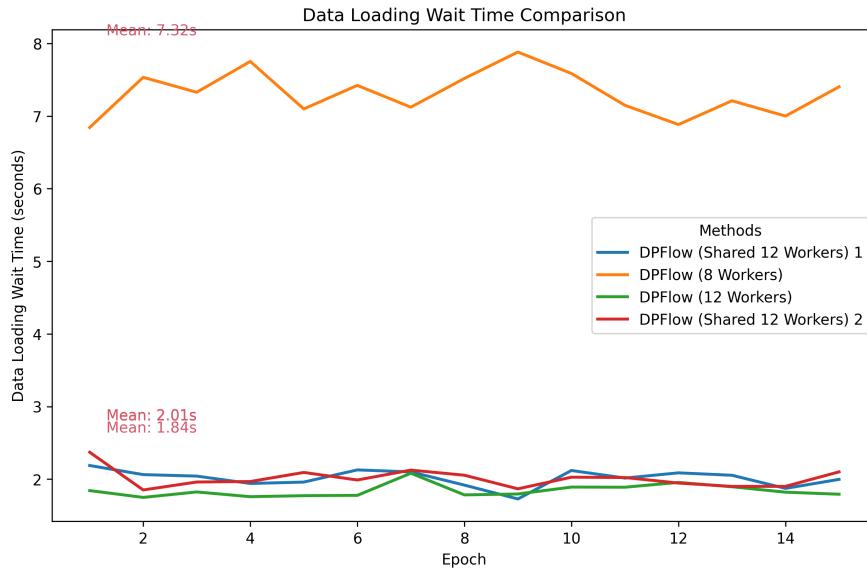


图 5-9 多任务共享的数据加载等待时间对比

数的 CPU 资源，减少了对训练函数的干扰。同时，借助二级预取机制，DPFlow 能够有效平缓不同加载速度带来的影响。相比 Pytorch DataLoader，DPFlow 的数据加载等待时间缩短了 31.97%。当 DPFlow 的并行度提高到 12 时，数据加载等待时间缩短了 82.89%。

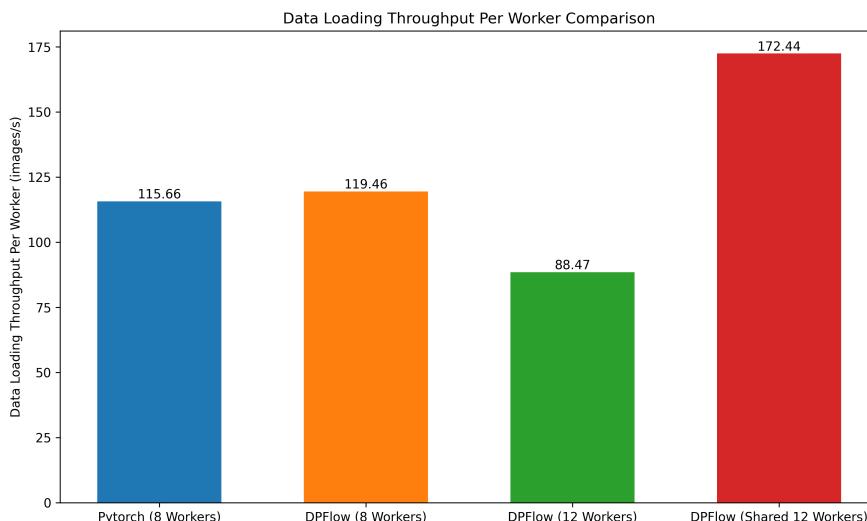


图 5-10 多任务场景的数据加载单并发度吞吐量对比

此外，在多任务场景下，DPFlow 能够通过协同采样和缓存机制，复用数据加载的数据，使得平均到每个任务所需要的 CPU 资源更少，从而在达到较好性能效果的情况下，降低 CPU 资源开销。图 5-9 展示了使用同样数据加载过程的两个并行训练任务，共享并发度 12 的数据加载函数，在平均每个任务加载并发度

为 6 的情况下，较并发度为 8 的任务取得了更低的数据加载等待时间。

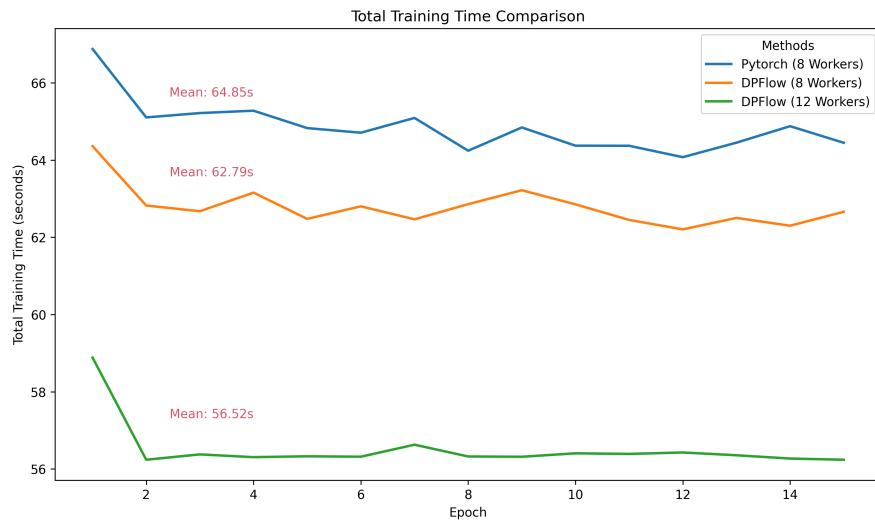


图 5-11 不同方案的训练总时间对比

图5-10展示了在多任务场景下，单一并发度情况下的平均数据加载吞吐量对比结果。受限于单个训练函数的网络带宽，并行度为 12 的 DPFlow 方案的吞吐量不及并行度为 8 的 PyTorch 方案。而在多任务场景中，通过在不同训练函数中并行运行训练任务，可以有效提升整体网络带宽的上限，并通过缓存和协同采样复用已加载的数据。实验结果表明，共享任务 DPFlow 方案的吞吐量达到 PyTorch 方案的 1.49 倍，提高了数据加载的吞吐量表现。

得益于数据加载的加速，DPFlow 也减少了端到端训练时间，结果如图5-11所示。在分布式预处理模式下，使用 DPFlow 训练一个 epoch 的时间从 Pytorch DataLoader 的 64.85 秒缩短至 62.79 秒。随着并行度提高，训练时间也缩短到 56.52 秒。

同时，我们比较了不同方案的 GPU 利用率，结果如图5-12所示。我们观察到并发度为 8 的 DPFlow GPU 利用率波动不大，但只有 88.38%，这表明数据加载还有进一步优化的空间。因此，我们针对并发度为 12 的 DPFlow 进行了测试，GPU 利用率达到了 96.94%，几乎实现了满载运行。这说明当并发度达到 12 时，GPU 资源能够更有效地利用，而原先阻塞训练流程的数据加载开销被削减。相比之下，PyTorch DataLoader 的 GPU 利用率不断波动，平均仅为 83.18%，资源利用效率有待提高。

端到端数据加载性能评估实验表明，DPFlow 可以加速深度学习训练中的数据加载过程，提高数据加载的吞吐量，从而提高 GPU 资源利用率，并缩短整体

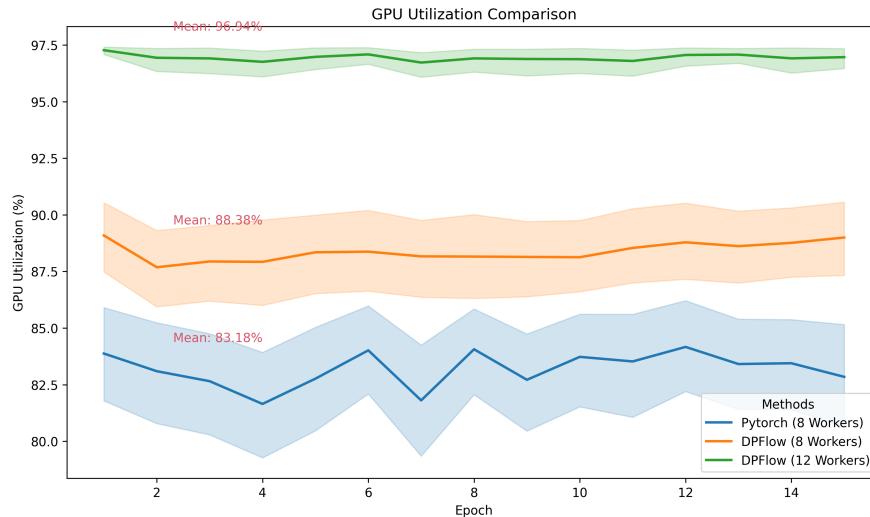


图 5-12 不同方案的 GPU 利用率对比

的训练时间。

5.5 本章小结

本章评估了 DPFlow 系统的有效性。在算法层面，实验表明，ALSD 缓存置换策略可提高缓存命中率，高出 LRU、LFU 等通用缓存策略 26.65、27.8 个百分点，高出相关工作设计的 MinIO、RefCnt 等策略 19.55、3.2 个百分点。在多任务同时训练但训练速度不同的场景下，相对于 Joader，DynaGCS 协同采样算法将任务间的数据重叠率提升了 2 倍以上，同时通过统计学仿真实验证明其良好的随机性。

功能方面，DPFlow 展现出了对图像分类、目标检测、语音识别等典型深度学习任务的良好支持，覆盖了视觉、语音等多个领域。不同任务数据集的加载程度符合预期。此外，DPFlow 对模型训练曲线和精度的影响可以忽略不计，在提速的同时确保了模型的正常收敛。

性能方面，为张量数据优化的 CRPC 模块改善了 DPFlow 分布式通信效率，降低数据拷贝次数，并通过压缩、异步 I/O、多通道传输等手段，将数据吞吐量提升了 1.82 倍。在端到端 Mini-ImageNet 图像分类训练实验中，DPFlow 通过提高并行度，将每轮数据加载等待时间缩短了 82.89%，整体训练时间缩短了 11.9%，使 GPU 利用率达到了 96.94%，性能超过了 PyTorch DataLoader 的数据加载方案。在两任务并行训练的场景下，即使使用平均到每个任务更少的并行度，也能达到

与高并行度 DPFlow 相似的速度，数据加载吞吐量相比 PyTorch 提高了 49%，验证了真实场景下缓存和数据协同采样算法的有效性。

综上可见，DPFlow 在提供深度学习数据加载功能及优化性能方面取得了相应的效果，进而验证了本文设计思路的有效性和合理性。

第六章 总结与展望

6.1 工作总结

随着深度学习技术的快速发展和 Serverless 计算平台的日益完善，利用 Serverless 平台进行深度学习训练引起了广泛关注。Serverless 计算以其自动化的资源管理和调度、按需付费、自动伸缩等独特优势，为深度学习训练提供了更加灵活、经济、易用的解决方案。然而，现有 Serverless 深度学习训练工作主要关注于模型训练平台的实现和优化训练过程，对数据加载阶段的研究相对较少。高效的数据加载可以降低训练时间，提高 GPU 利用率，进一步提升训练效率和降低成本，具有重要的研究价值。

现有的 Serverless 深度学习训练在数据加载方面存在诸多挑战。主流深度学习框架的内置数据加载器在 Serverless 环境下存在局限性，如缺乏分布式并行加载能力、没有考虑存算分离架构带来的 I/O 开销、GPU-CPU 资源分配不均衡导致在线预处理速度较慢等。这些问题共同导致了数据加载速度跟不上训练速度，使得 GPU 出现空闲等待，影响训练效率。

为解决上述问题，本文提出了一种 Serverless 分布式数据加载方案 DPFlow，主要贡献和结果总结如下：

- 提出了一种 **Serverless 分布式数据加载模型**。该模型综合考虑了 Serverless 环境的特点，为数据加载过程构建了统一、抽象的描述和优化框架，为后续研究奠定了基础。
- 设计了**声明式数据加载编程框架**。该框架引入了静态通用数据集规范，在不修改原始数据集的前提下规范化了数据集格式。此外，基于 Python 实现了一套声明式编程接口，为用户提供了灵活的数据加载定制能力。
- 提出了一系列**数据加载优化方法**。对 Serverless 环境下数据加载面临的效果瓶颈，本文设计了包括并行数据加载、数据预取、数据缓存和数据采样在内的多种优化方法。其中，ALSD 缓存置换策略相较于 LRU、MinIO 策

略将缓存命中率提高了 26.65 至 19.55 个百分点；DynaGCS 采样算法相较于 Joader 将任务间的数据重叠率提升了 2 倍以上。

- **实现了 DPFlow 原型系统。**为验证所提出的 Serverless 分布式数据加载模型和优化方法的有效性，本文在阿里云 Serverless 平台上部署并评估了 DPFlow 原型系统。实验结果表明：相比 PyTorch 内置的数据加载方案，DPFlow 平均缩短了 82.9% 的数据加载等待时间，减少了 11.9% 的训练时间，GPU 利用率提升到 96.94%。在两任务并发训练场景下，DPFlow 的数据加载吞吐量提高了 49%。

6.2 研究展望

本文的未来工作主要有以下几个方向：

- **支持动态数据集的加载优化。**本文主要针对静态训练数据集的加载进行优化，对动态生成的训练数据、特别是在线学习中的数据加载过程尚未涉及。动态数据加载涉及数据生成、转换、校验等环节，与静态数据集有较大差异，需要设计不同的优化策略。未来可以在 DPFlow 的框架基础上，扩展对在线学习等场景下动态数据加载的支持。
- **增强对分布式训练的支持。**目前 DPFlow 主要针对单个模型的训练，对分布式训练没有直接支持。在分布式训练场景下，不同节点（如参数服务器、工作节点）的角色、通信模式差异较大，对数据加载也提出了不同的需求。比如在 DynaGCS 采样算法中，需要考虑不同节点的采样一致性问题。未来需要在现有优化方法的基础上，增加分布式训练场景下的适配与改进。
- **扩展多云平台支持，进行跨云适配和优化。**目前 DPFlow 主要在阿里云平台上实现和评测，但不同云平台在 Serverless 服务的接口、性能、定价等方面存在差异。为了提高 DPFlow 的通用性，未来需要扩展对其他主流云平台如腾讯云、AWS 等的支持，并针对不同平台的特性进行适配与优化。

参考文献

- [1] Jiang J, Gan S, Liu Y, et al. Towards Demystifying Serverless Machine Learning Training[C/OL]// SIGMOD '21: Proceedings of the 2021 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2021: 857-871 [2024-03-22]. <https://doi.org/10.1145/3448016.3459240>.
- [2] Bhattacharjee A, Barve Y, Khare S, et al. Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks[C/OL]// 2019 USENIX conference on operational machine learning (OpML 19). Santa Clara, CA: USENIX Association, 2019: 59-61. <https://www.usenix.org/conference/opml19/presentation/bhattacharjee>.
- [3] Carreira J, Fonseca P, Tumanov A, et al. Cirrus: a Serverless Framework for End-to-end ML Workflows[C/OL]// SoCC '19: Proceedings of the ACM Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2019: 13-24 [2024-03-21]. <https://dl.acm.org/doi/10.1145/3357223.3362711>.
- [4] Wang H, Niu D, Li B. Distributed Machine Learning with a Serverless Architecture[C/OL]// IEEE INFOCOM 2019 - IEEE Conference on Computer Communications. Paris, France: IEEE, 2019: 1288-1296 [2024-04-02]. <https://ieeexplore.ieee.org/document/8737391/>.
- [5] Gupta V, Kadhe S, Courtade T, et al. OverSketched Newton: Fast Convex Optimization for Serverless Systems[C/OL]// 2020 IEEE International Conference on Big Data (Big Data). 2020: 288-297 [2024-04-02]. <https://ieeexplore.ieee.org/document/9378289>.
- [6] Carreira J, Fonseca P, Tumanov A, et al. A case for serverless machine learn-

- ing[C]//Workshop on systems for ML and open source software at NeurIPS: vol. 2018. 2018: 2-8.
- [7] Petrescu S, Martinez D A, Rellermeyer J S. Toward Competitive Serverless Deep Learning[C/OL]//DICG '23: Proceedings of the 4th International Workshop on Distributed Infrastructure for the Common Good. New York, NY, USA: Association for Computing Machinery, 2024: 7-12 [2024-04-04]. <https://dl.acm.org/doi/10.1145/3631310.3633489>.
- [8] Chahal D, Ojha R, Ramesh M, et al. Migrating Large Deep Learning Models to Serverless Architecture[C/OL]//2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). 2020: 111-116 [2024-04-04]. <https://ieeexplore.ieee.org/document/9307673>.
- [9] Castro P, Ishakian V, Muthusamy V, et al. The rise of serverless computing [J/OL]. Communications of the ACM, 2019, 62(12): 44-54(2019-11-21) [2024-03-22]. <https://dl.acm.org/doi/10.1145/3368454>.
- [10] Eismann S, Scheuner J, Van Eyk E, et al. Serverless Applications: Why, When, and How? [J/OL]. IEEE Software, 2021, 38(1): 32-39 [2024-03-22]. <https://ieeexplore.ieee.org/document/9190031/>.
- [11] Shafiei H, Khonsari A, Mousavi P. Serverless Computing: A Survey of Opportunities, Challenges and Applications[EB / OL]. (2021-06-04) [2024-03-22]. <https://arxiv.org/abs/1911.01296>. arXiv: 1911.01296 [cs]. preprint.
- [12] Yu H, Li J, Hua Y, et al. Cheaper and Faster: Distributed Deep Reinforcement Learning with Serverless Computing[J/OL]. Proceedings of the AAAI Conference on Artificial Intelligence, 2024, 38(15): 16539-16547(2024-03-24) [2024-04-02]. <https://ojs.aaai.org/index.php/AAAI/article/view/29592>.
- [13] Zolnouri M, Li X, Nia V P. Importance of Data Loading Pipeline in Training Deep Neural Networks[EB / OL]. (2020-04-21) [2023-03-14]. [http://arxiv.org/abs/2005.02130](https://arxiv.org/abs/2005.02130). arXiv: 2005.02130 [cs]. preprint.
- [14] Paszke A, Gross S, Massa F, et al. PyTorch: an imperative style, high-

- performance deep learning library[G]//Proceedings of the 33rd International Conference on Neural Information Processing Systems: 721. Red Hook, NY, USA: Curran Associates Inc., 2019: 8026-8037.
- [15] Ofeidis I, Kiedanski D, Tassiulas L. An Overview of the Data-Loader Landscape: Comparative Performance Analysis[EB/OL]. (2022-09-27) [2024-03-22]. <http://arxiv.org/abs/2209.13705>. arXiv: 2209.13705 [cs]. preprint.
- [16] Zhao H, Han Z, Yang Z, et al. SiloD: A Co-design of Caching and Scheduling for Deep Learning Clusters[C/OL]//EuroSys '23: Proceedings of the Eighteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2023: 883-898 [2023-05-11]. <https://dl.acm.org/doi/10.1145/3552326.3567499>.
- [17] Mohan J, Phanishayee A, Raniwala A, et al. Analyzing and mitigating data stalls in DNN training[J]. ArXiv, 2020, abs/2007.06775.
- [18] Ray. Ray Data Overview —Ray 2.10.0[EB/OL]. 2024 [2024-03-22]. <https://docs.ray.io/en/latest/data/overview.html>.
- [19] Moritz P, Nishihara R, Wang S, et al. Ray: a distributed framework for emerging AI applications[C]//OSDI'18: Proceedings of the 13th USENIX conference on operating systems design and implementation. USA: USENIX Association, 2018: 561-577.
- [20] Lhoest Q, Villanova del Moral A, Jernite Y, et al. Datasets: A community library for natural language processing[C/OL]//Proceedings of the 2021 conference on empirical methods in natural language processing: System demonstrations. Online: Association for Computational Linguistics, 2021: 175-184. arXiv: 2109.02846 [cs.CL]. <https://aclanthology.org/2021.emnlp-demo.21>.
- [21] Richardson N, Cook I, Crane N, et al. arrow: Integration to 'apache' 'arrow'[A/OL]. 2024. <https://github.com/apache/arrow/>.
- [22] Torch. torch.utils.data —PyTorch 2.2 documentation[EB/OL]. 2024 [2024-03-22]. <https://pytorch.org/docs/stable/data.html>.

- [23] Xie J, Xu J, Wang G, et al. A deep learning dataloader with shared data preparation[C] // NIPS '22: Proceedings of the 36th International Conference on Neural Information Processing Systems. Red Hook, NY, USA: Curran Associates Inc., 2022: 17146-17156.
- [24] Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing[EB / OL]. (2019-02-09) [2024-03-30]. <http://arxiv.org/abs/1902.03383>. arXiv: 1902.03383 [cs]. preprint.
- [25] Baldini I, Castro P, Chang K, et al. Serverless computing: Current trends and open problems[G] // Research advances in cloud computing. Springer, 2017: 1-20.
- [26] Copik M, Böhringer R, Calotoiu A, et al. FMI: Fast and Cheap Message Passing for Serverless Functions[C / OL] // ICS '23: Proceedings of the 37th International Conference on Supercomputing. New York, NY, USA: Association for Computing Machinery, 2023: 373-385 [2023-11-20]. <https://dl.acm.org/doi/10.1145/3577193.3593718>.
- [27] Lynn T, Rosati P, Lejeune A, et al. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms[C / OL] // 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). 2017: 162-169 [2024-03-28]. <https://ieeexplore.ieee.org/document/8241104>.
- [28] Amazon Web Services. AWS re:Invent 2014 | (MBL202) NEW LAUNCH: Getting Started with AWS Lambda. 2014 [2024-03-24]. <https://www.youtube.com/watch?v=UFj27laTWQA>.
- [29] Adzic G, Chatley R. Serverless computing: economic and architectural impact [C / OL] // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn Germany: ACM, 2017: 884-889 [2022-09-23]. <https://dl.acm.org/doi/10.1145/3106237.3117767>.
- [30] Li Z, Guo L, Cheng J, et al. The Serverless Computing Survey: A Technical Primer for Design Architecture[J / OL]. ACM Computing Surveys, 2022, 54: 1-

- 34(2022-01-31) [2024-04-04]. <https://dl.acm.org/doi/10.1145/3508360>.
- [31] Fingler H, Akshintala A, Rossbach C J. USETL: Unikernels for serverless extract transform and load why should you settle for less? [C/OL] // Asia pacific workshop on systems. 2019. <https://api.semanticscholar.org/CorpusID:199501425>.
- [32] Eivy A, Weinman J. Be Wary of the Economics of "Serverless" Cloud Computing [J/OL]. IEEE Cloud Computing, 2017, 4(2): 6-12 [2024-03-28]. <https://ieeexplore.ieee.org/document/7912239>.
- [33] LeCun Y, Bengio Y, Hinton G. Deep learning[J/OL]. Nature, 2015, 521(7553): 436-444 [2024-03-28]. <https://www.nature.com/articles/nature14539>.
- [34] Goodfellow I, Bengio Y, Courville A. Deep learning[M]. MIT Press, 2016.
- [35] Robbins H, Monro S. A Stochastic Approximation Method[J/OL]. The Annals of Mathematical Statistics, 1951, 22(3): 400-407 [2024-03-28]. <https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-3/A-Stochastic-Approximation-Method/10.1214/aoms/1177729586.full>.
- [36] Luangsomboon N, Fazel F, Liebeherr J, et al. On the Burstiness of Distributed Machine Learning Traffic[J/OL]. 2024 [2024-04-04]. <https://arxiv.org/abs/2401.00329>.
- [37] Abadi M, Barham P, Chen J, et al. TensorFlow: a system for large-scale machine learning[C] // OSDI'16: Proceedings of the 12th USENIX conference on operating systems design and implementation. USA: USENIX Association, 2016: 265-283.
- [38] Chahal D, Mishra M, Palepu S C, et al. Pay-as-you-Train: Efficient ways of Serverless Training[C/OL] // 2022 IEEE International Conference on Cloud Engineering (IC2E). CA, USA: IEEE, 2022: 116-125 [2024-03-18]. <https://ieeexplore.ieee.org/document/9946347/>.
- [39] Liu Y, Jiang B, Guo T, et al. FuncPipe: A Pipelined Serverless Framework for Fast and Cost-efficient Training of Deep Learning Models[C/OL] // SIGMETRICS '23: Abstract Proceedings of the 2023 ACM SIGMETRICS Inter-

- national Conference on Measurement and Modeling of Computer Systems. New York, NY, USA: Association for Computing Machinery, 2023: 35-36 [2024-03-24]. <https://dl.acm.org/doi/10.1145/3578338.3593543>.
- [40] Barrak A, Petrillo F, Jaafar F. Serverless on Machine Learning: A Systematic Mapping Study[J/OL]. IEEE Access, 2022, 10: 99337-99352 [2024-04-04]. <https://ieeexplore.ieee.org/document/9888122>.
- [41] Lee G, Lee I, Ha H, et al. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training[C/OL]//. 2021: 537-550 [2023-05-11]. <https://www.usenix.org/conference/atc21/presentation/lee>.
- [42] Audibert A, Chen Y, Graur D, et al. A case for disaggregation of ML data processing[EB/OL]. (2022-10-28) [2023-03-14]. <http://arxiv.org/abs/2210.14826>. arXiv: 2210.14826 [cs]. preprint.
- [43] Li X, Dong B, Xiao L, et al. Small Files Problem in Parallel File System[C/OL] //2011 International Conference on Network Computing and Information Security: vol. 2. 2011: 227-232 [2024-04-05]. <https://ieeexplore.ieee.org/abstract/document/5948826>.
- [44] Lee J, Bahn H. File access characteristics of deep learning workloads and cache-friendly data management[J/OL]. 2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI), 2023: 328-331. <https://api.semanticscholar.org/CorpusID:264880475>.
- [45] Nakandala S, Zhang Y, Kumar A. Cerebro: A data system for optimized deep learning model selection[J]. Proc. VLDB Endow., 2020, 13: 2159-2173.
- [46] Kumar A, Sivathanu M. Quiver: An informed storage cache for deep learning[C] // USENIX conference on file and storage technologies. 2020.
- [47] Kang Z, Min Z, Zhou S, et al. Towards High-Performance Data Loading in Cloud-Native Deep Learning Systems[C/OL]//2024 16th International Conference on COMmunication Systems & NETworkS (COMSNETS). Bengaluru, India: IEEE,

- 2024: 361-369 [2024-02-22]. <https://ieeexplore.ieee.org/document/10427257/>.
- [48] Deng J, Dong W, Socher R, et al. ImageNet: A large-scale hierarchical image database[C/OL]//2009 IEEE Conference on Computer Vision and Pattern Recognition. 2009: 248-255 [2024-04-08]. <https://ieeexplore.ieee.org/document/5206848/>.
- [49] Lin T Y, Maire M, Belongie S, et al. Microsoft COCO: Common Objects in Context[EB/OL]. (2015-02-20) [2024-04-08]. <http://arxiv.org/abs/1405.0312>. arXiv: 1405.0312 [cs]. preprint.
- [50] Aliyun. 块存储_云服务器存储_块存储服务_数据块级随机存储-阿里云[EB/OL]. 2024 [2024-03-21]. <https://www.aliyun.com/product/disk>.
- [51] Aliyun. 对象存储 OSS_云存储服务_企业数据管理_存储-阿里云[EB/OL]. 2024 [2024-03-21]. <https://www.aliyun.com/product/oss>.
- [52] Aliyun. 文件存储 NAS_分布式文件系统_满足高吞吐低延时的存储性能_阿里云[EB/OL]. 2024 [2024-03-21]. <https://www.aliyun.com/product/nas>.
- [53] Aliyun. 表格存储 Tablestore_表格存储_物联网_存储-阿里云[EB/OL]. 2024 [2024-03-21]. <https://www.aliyun.com/product/ots>.
- [54] Gu R, Xu Z, Che Y, et al. High-level data abstraction and elastic data caching for data-intensive AI applications on cloud-native platforms[J/OL]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34: 2946-2964. <https://api.semanticscholar.org/CorpusID:261797513>.
- [55] Aliyun. 超级计算集群实现自然语言处理训练_技术解决方案_最佳实践-阿里云[EB/OL]. 2024 [2024-03-23]. <https://bp.aliyun.com/detail/75>.
- [56] IEEE. IEEE SA - IEEE/Open Group 1003.1-2017[EB/OL]. 2024 [2024-04-03]. <https://standards.ieee.org/ieee/1003.1/7101/>.
- [57] PyO3/pyo3: Rust bindings for the Python interpreter[EB/OL]. [2024-04-24]. <https://github.com/PyO3/pyo3>.
- [58] PEP 3118 –Revising the buffer protocol | peps.python.org[EB/OL]. Python Enhancement Proposals (PEPs). [2024-04-24]. <https://peps.python.org/pep-3118/>.

- [59] PEP 574 –Pickle protocol 5 with out-of-band data | peps.python.org[EB / OL]. Python Enhancement Proposals (PEPs). [2024-04-24]. <https://peps.python.org/pep-0574/>.
- [60] Panayotov V, Chen G, Povey D, et al. Librispeech: An ASR corpus based on public domain audio books[C / OL]//2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). 2015: 5206-5210 [2024-04-08]. <https://ieeexplore.ieee.org/document/7178964>.
- [61] He K, Zhang X, Ren S, et al. Deep Residual Learning for Image Recognition [EB / OL]. (2015-12-10) [2024-04-08]. <http://arxiv.org/abs/1512.03385>. arXiv: 1512.03385 [cs]. preprint.
- [62] Liu W, Anguelov D, Erhan D, et al. SSD: Single Shot MultiBox Detector[G / OL] //: vol. 9905. 2016: 21-37. arXiv: 1512.02325 [cs] [2024-04-08]. <http://arxiv.org/abs/1512.02325>.
- [63] Rao K, Sak H, Prabhavalkar R. Exploring Architectures, Data and Units For Streaming End-to-End Speech Recognition with RNN-Transducer[EB / OL]. (2018-01-02) [2024-04-08]. <http://arxiv.org/abs/1801.00841>. arXiv: 1801.00841 [cs, eess]. preprint.

简历与科研成果

基本信息

朱治学，男，汉族，1999年2月出生，四川雅安人。

教育背景

| | |
|--|----|
| 2021 年 9 月—2024 年 6 月 南京大学计算机科学与技术系 | 硕士 |
| 2017 年 9 月—2021 年 6 月 华中科技大学计算机科学与技术学院 | 本科 |

攻读硕士学位期间完成的学术成果

- 发明专利：面向 Serverless 机器学习模型训练的动态可伸缩数据共享系统（专利号：202410128606.7）
- 发明专利：面向 Serverless 函数流应用的并行度自适应调优方法（专利号：202311464224.3）
- 发明专利：一种基于时间窗口划分的微服务版本安全撤销系统和方法（专利号：202211410299.9）
- 软件著作权：Faasit-Deploy 服务器无感知计算函数部署软件 v1.0（登记号：2024SR0141544）

攻读硕士学位期间参与的科研课题

- 国家重点研发计划：服务器无感知计算系统软件技术（2022YFB4500700），2022年10月 - 2025年09月，参与领域特定语言与编译器的设计与实现。
- 广东省重点领域研发计划：面向云数据中心智能管控的软件定义方法与关键技术（2020B010164003），2020年1月—2022年12月，参与微服务动态更新算法与原型系统的设计与实现。