面向深度学习的数据存储技术综述

贺巩山[1),2),3)] 赵传磊[1),3)] 蒋金虎[2),3)] 张为华[1),2),3)] 陈左宁[2),4)]

[1)](复旦大学计算机科学技术学院 上海 200438)

[2)](复旦大学大数据研究院 上海 200433)

[3)](复旦大学并行处理研究所 上海 200433)

[4)](中国工程院 北京 100088)

摘 要 随着数据总量和计算能力的不断提升,以深度学习和大模型为代表的人工智能技术获得了迅速的发展,并成功应用于计算机视觉和自然语言处理等领域。然而,随着GPU等加速器运算速度的提高,数据存储已经成为了深度学习训练和推理的主要瓶颈之一,主要表现为:(1)数据集的规模快速增长,无法完全缓存在内存中;(2)若无额外处理,数据集主要由小文件组成。在每轮训练中,训练任务会随机读取训练集中的文件;(3)与GPU等加速器相比,存储设备的带宽增长缓慢,二者之间的差距正在不断变大;(4)模型参数和中间数据等模型状态非常大,经常超过GPU等加速器的存储容量,出现了内存墙的问题;(5)为了实现容错,训练任务通常会执行检查点操作,保存最新的模型状态,但这引入了较高的性能开销。因此,面向人工智能(尤其是深度学习)的数据存储技术成为了热门的研究领域,受到了学术界和工业界的广泛关注。本文首先介绍了深度学习的相关背景,包括流程、模型以及分布式训练。其次,本文总结了深度学习的数据特点,包括数据集和模型的规模与类型,以及数据准备(包括数据加载和数据预处理)和模型计算(包括模型训练和模型推理)的数据访问模式。接着,本文分析了深度学习在数据加载、数据预处理以及模型计算阶段的数据存储需求,提出了面向深度学习的数据存储技术研究框架。然后,本文梳理了现有的相关工作,并根据针对的阶段不同将其分为3类:(1)面向数据加载的存储优化技术关注于如何加速数据加载阶段,包括数据集存储格式、数据集存储系统、数据集缓存系统以及数据加载器;(2)面向数据预处理的存储优化技术关注于如何加速数据预处理阶段,包括数据预处理流水线、分离式数据预处理、数据预处理缓存以及近存储数据预处理;(3)面向模型计算的存储优化技术关注于如何加速模型计算阶段,包括模型状态存储技术、模型训练容错技术、模型存储系统以及性能测试与分析工具。最后,本文讨论了现有工作存在的问题,提出了未来可能的研究方向。

关键词 深度学习;数据存储技术;数据加载优化;数据预处理优化;模型计算优化

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2025.01013

A Survey of Data Storage Technologies for Deep Learning

HE Gong-Shan[1),2),3)] ZHAO Chuan-Lei[1),3)] JIANG Jin-Hu[2),3)]

ZHANG Wei-Hua[1),2),3)] CHEN Zuo-Ning[2),4)]

[1)](School of Computer Science, Fudan University, Shanghai 200438)

[2)](Institute of Big Data, Fudan University, Shanghai 200433)

[3)](Parallel Processing Institute, Fudan University, Shanghai 200433)

[4)](Chinese Academy of Engineering, Beijing 100088)

Abstract With the continuous growth in data volume and computing power, artificial intelli-

收稿日期:2024-08-02;在线发布日期:2025-02-21。本课题得到国家重点研发计划项目(No. 2023YFB4502703)资助。**贺巩山**,博士研究生,中国计算机学会(CCF)学生会员,主要研究领域为面向AI的数据存储技术、文件系统、分布式存储等。E-mail:gshe21@m.fudan.edu.cn。**赵传磊**,博士研究生,中国计算机学会(CCF)学生会员,主要研究领域为并行处理、GPU计算、文件系统等。**蒋金虎(通信作者)**,博士,高级工程师,中国计算机学会(CCF)高级会员,主要研究领域为计算机体系结构、操作系统、分布式存储等。E-mail:jiangjinhu@fudan.edu.cn。**张为华(通信作者)**,博士,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为编译优化、计算机体系结构、并行、系统软件等。E-mail:zhangweihua@fudan.edu.cn。**陈左宁**,研究员,博士生导师,中国工程院院士,中国计算机学会(CCF)会士,主要研究领域为存储系统、操作系统、信息安全等。

gence (AI) technology, represented by deep learning (DL) and large models, has undergone rapid development and successfully applied in various fields, for example, computer vision (CV) and natural language processing (NLP). As accelerators such as GPUs continue to advance in speed, the data storage has become one of the major bottlenecks in DL training and inference. To be specific, the data storage faces several significant challenges: (1)The size of datasets is growing rapidly, making it impractical to cache them entirely in memory. (2)If no additional processing is performed, datasets primarily consist of small files. During each epoch, training jobs randomly read files from the training set. On the one hand, traditional storage systems are optimized for sequentially handling large files. On the other hand, this access pattern can cause cache thrashing when using the existing cache replacement policies, such as LRU (Least Recently Used). (3)The bandwidth of storage devices is improving at a slower pace compared to accelerators like GPUs. Consequently, the gap between I/O and compute is widening. (4)Model states, such as model parameters and intermediate data, become huge, often exceeding the memory capacity of accelerators like GPUs, known as the memory capacity wall problem. (5)Training jobs are frequently interrupted due to failures. For fault tolerance, training jobs typically perform checkpointing operation to save the latest model states, but this incurs a significant performance overhead. To address these problems, storage for AI, particularly DL, has emerged as a hot research area, drawing extensive attention from both academia and industry. In this paper, we first introduce the relevant background of DL, which includes the common processes of DL training and inference, typical models, and distributed training methods. Secondly, we summarize the data characteristics of DL, which include the sizes and types of datasets and models, as well as data access patterns during the data preparation stage (including data loading and data preprocessing) and model computing stage (consisting of model training and model inference). Next, we analyze the data storage requirements of DL in the data loading, data preprocessing, and model computing stages, and propose a research framework on data storage technologies for DL. Then, we summarize the existing related work and classify it into three categories based on the different stages it targets: (1)Storage optimization technologies for data loading focus on how to accelerate the data loading stage. These include improvements in dataset storage formats, dataset storage systems, dataset caching systems, and dataloaders. (2)Storage optimization technologies for data preprocessing concentrate on how to speed up the data preprocessing stage. These involve the data preprocessing pipeline, disaggregated data preprocessing, data preprocessing cache optimizations, and near-storage data preprocessing. (3)Storage optimization technologies for model computing aim to accelerate the model computing stage. These include model state storage technologies, fault tolerance techniques for model training, model storage systems, as well as I/O benchmarking, analysis, and profiling tools. Finally, we discuss the limitations of the existing work and propose potential directions for future research.

Keywords deep learning; data storage technology; data loading optimization; data preprocessing optimization; model computing optimization

1 引 言

近年来,基于深度学习和大模型的人工智能(Artificial Intelligence,AI)技术获得了快速的发展。各种深度学习算法和模型不断涌现,并成功应用于计算机视觉(Computer Vision,CV)、自然语言处理(Natural Language Processing,NLP)、语音识别、Web 搜索以及推荐系统等领域。以自然语言处理为例,OpenAI 在 2022 年 11 月推出了基于大语

言模型(Large Language Models,LLMs)的聊天机器人 ChatGPT,受到了学术界和工业界的广泛关注,引起了社会大众对 LLM 的热烈讨论。此外,以 GPU 为代表的加速器运算速度也在不断提高,有效满足了深度学习任务对算力的需求。随着算法的推陈出新和算力的持续增长,数据存储已经成为了深度学习训练和推理的主要瓶颈之一,主要表现为:

(1)数据集规模带来的挑战。在过去的十几年中,数据集规模呈现指数级增长[1],新的数据集越来越大。例如,在计算机视觉领域,ImageNet-21K 数据集[2]的大小约为 1.3TB,而 OpenImages 数据集[3]的完整大小则约为 18TB。随着大模型的流行,模型的参数量越来越多,表达能力越来越强,需要的训练数据还在不断变大。Google 的应用表明[4]:13%的训练任务需要读取至少 1TB 的输入数据。因此,对于部分训练任务,数据集无法完全缓存在内存中。

(2)数据集访问模式带来的挑战。如果没有执行额外的格式转换,数据集主要由小文件组成。例如,在 ImageNet-1K 数据集[5]中,样本文件的平均大小为 114KB。在每轮训练开始之前,训练任务需要打乱样本文件的读取顺序,以提高模型的泛化能力。而在每轮训练中,训练任务则需要将训练集中的每个样本刚好处理一次。也就是说,深度学习训练需要执行大量的小文件随机读取操作,空间局部性和时间局部性都较差。一方面,传统的存储系统是针对大文件的顺序访问而设计的,难以满足训练场景的小文件随机读取需求。另一方面,传统的缓存替换策略,例如最近最少使用(Least Recently Used,LRU)和最不经常使用(Least Frequently Used,LFU)等,利用了数据访问的局部性,在训练场景中存在着严重的缓存抖动。

(3)模型训练速度与数据加载速度失衡带来的挑战。以 GPU 为代表的深度学习加速器运算速度越来越快,能够更快地完成深度学习模型的计算任务。为了充分利用 GPU 等加速器的计算资源,数据必须以合适的速度喂入计算单元中,这给数据存储带来了巨大的挑战。以在 ImageNet-1K 上训练 ResNet50[6]为例,8 块 NVIDIA V100 GPU 每秒可以处理 10267 张图片①,需要 1143MB/s 的随机读取带宽。随着深度学习加速器的运算速度不断提高,数据存储和计算模块之间的鸿沟将会越来越大。

(4)模型参数规模带来的挑战。在训练和推理场景中,计算任务不仅要存储模型参数,还需要保存其他模型状态,例如,激活数据(为了便于描述,本文将层内的部分中间数据和激活函数的输出数据都称为激活数据)、梯度以及优化器状态(例如,动量)等。随着模型的参数规模不断变大,其他模型状态的大小也会相应增加,导致需要的加速器存储容量迅速增长。以 LLM 为例,模型参数规模平均每年增长 14.1 倍,而 GPU 等加速器的存储容量平均每年仅增长 1.3 倍[7]。因此,模型状态的存储容量需求和加速器的存储容量增长速度不匹配,出现了内存墙的问题。

(5)模型训练容错带来的挑战。在训练过程中,计算集群可能会出现各种各样的故障,继而导致训练任务中断。为了避免浪费计算资源,训练任务通常会定期执行检查点操作,保存最新的模型参数和优化器状态等模型状态。当前,深度学习框架大都采用同步检查点方法,即在执行检查点操作时,训练任务需要暂停。提高检查点的执行频率,可以缩短故障恢复所需的时间,但引入了较高的性能开销。而降低检查点的执行频率,可以降低检查点开销,但延长了故障恢复所需的时间。因此,训练任务需要更好的容错方法,以权衡执行频率和性能开销,避免计算资源的浪费,降低故障恢复的开销。

在这样的背景下,“Storage for AI”已经成为了存储领域的研究热点。对于“Storage for AI”的定义,Amvrosiadis 等人在《数据存储研究远景 2025:在 NSF 远景研讨会上的报告》②中提到:“Storage for AI”着眼于研究如何使用存储技术去更好地满足 AI 工作负载和数据使用的需求。目前,国内外已经有两篇关于“Storage for AI”的综述论文[8-9]。表 1 展示了本文和这两篇综述论文的区别。不过,这两篇综述论文仅对“Storage for AI”的部分内容进行了介绍,没有对其进行全面分析和总结。

表 1　本文与其他“Storage for AI”综述论文的对比

综述论文	内容覆盖					其他	
	数据加载优化	数据预处理优化	模型训练优化	模型推理优化	性能测试与分析工具	分类粒度	引文数量
文献[8]	×	×	√	×	×	细	少(<50)
文献[9]	√	√	√	×	√	粗	中(<150)
本文	√	√	√	√	√	细	多(>200)

① Pushing the limits of GPU performance with XLA, https://blog.tensorflow.org/2018/11/pushing-limits-of-gpu-performance-with-xla.html 2024,5,27。

② Data Storage Research Vision 2025: Report on NSF Visioning Workshop held May 30-June 1, 2018, https://par.nsf.gov/servlets/purl/10086429 2023,12,11。

本文对“Storage for AI”方向进行了调研，总结了面向深度学习的数据存储技术，主要贡献包括以下几个方面：

(1)本文总结了深度学习的数据特点，分析了深度学习的数据存储需求，提出了面向深度学习的数据存储技术研究框架；

(2)本文介绍了现有的面向深度学习的数据存储技术，包括面向数据加载的存储优化技术、面向数据预处理的存储优化技术以及面向模型计算的存储优化技术；

(3)本文讨论了现有工作存在的问题，提出了面向深度学习的数据存储技术未来可能的研究方向。

本文的组织结构如下：第 2 章描述了深度学习的相关背景；第 3 章介绍了面向数据加载的存储优化技术；第 4 章介绍了面向数据预处理的存储优化技术；第 5 章介绍了面向模型计算的存储优化技术；第 6 章对面向深度学习的数据存储技术进行了展望，提出了未来可能的研究方向；第 7 章对本文进行了总结。

2 相关背景

2.1 深度学习背景介绍

2.1.1 典型流程

一般来说，深度学习有两种模式[10]：训练(Training)和推理(Inference)，典型流程如图 1 所示。在训练过程中，深度学习应用使用训练数据作为输入，并从训练数据中学习模型参数[11]；在推理过程中，训练好的模型将被用于预测新的输入数据[10]。

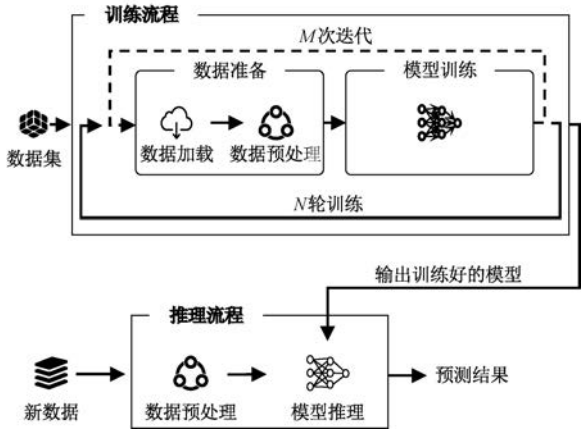

图 1 深度学习的典型流程

(1)训练

数据集。在训练场景中，数据集通常会被划分为三个集合：训练集、验证集以及测试集。在正式训练开始之前，训练任务需要执行超参数搜索(Hyperparameter Search)，确定模型的超参数取值，例如，模型的层数、每层的大小以及学习率等。此时，训练任务会使用训练集来训练模型，使用验证集来评估模型的性能，并根据模型的性能不断调整超参数的取值。在确定超参数的取值之后，训练任务通常会使用所有非测试数据(即训练集和验证集)来训练模型，并在测试集上评估模型的性能。

训练流程。训练流程包括数据准备(Data Preparation)和模型训练。其中，数据准备阶段可以细分为两个阶段：第一，数据加载(Data Loading)，也叫数据获取(为了便于描述，本文统一采用“数据加载”一词)，即从远程存储或者本地存储中加载训练数据；第二，数据预处理(Data Preprocessing)，即将原始数据转换为模型需要的输入数据，通常由 CPU 来完成。在模型训练阶段，预处理后的数据将被喂到模型中。该阶段涉及大量的矩阵运算，通常在 GPU 等加速器中完成。具体来说，模型训练可以划分为三个步骤：第一，前向传播，即使用模型 f 计算输入数据的预测值；第二，反向传播，即根据输入数据 x 的预测值 y'、真实标签 y 和损失函数 L，计算每层中可学习参数 w 的梯度值(Gradient)；第三，参数更新，即根据计算出的梯度值 Δw，使用相应的优化器(Optimizer)来更新模型的参数值。

在执行训练时，训练任务并不会一次处理完所有的数据，而是每次处理部分数据。这称为一个小批量或者小批次(mini-batch)，通常包含 32-512 条数据。当一个小批量的所有数据均被处理完时，我们就称执行了一次迭代(Iteration)。而当训练集中的所有数据均被处理一次时，我们就称完成了一轮训练(Epoch)。一轮训练包含多次迭代，而一个训练任务通常会执行多轮训练，比如 100 次。在达到指定的训练轮数或者训练损失的变化小于指定阈值之后，模型已经收敛，训练过程结束。

(2)推理

在深度学习中，推理和训练的流程基本相同，主要的区别在于：第一，在数据准备阶段中，推理任务通常不需要加载数据集，只需要执行数据预处理(离线推理和部分在线推理任务需要加载数据集)。第二，在模型推理阶段中，推理任务只需要执行前向传播，计算输入数据的预测结果，不需要执行反向传播和参数更新[10]。第三，推理任务不需要执行模型评估。需要注意的是，模型训练阶段和模型推理阶段

只是不同模式中的模型计算。为了便于描述,本文将这两个阶段统称为模型计算阶段。

2.1.2　典型模型

随着深度学习技术的飞速发展,各种深度学习模型已被广泛应用于各个领域。下面,本文将介绍几类典型的深度学习模型。

(1)深度学习推荐模型

根据 Meta 公司的统计,基于深度神经网络(Deep Neural Networks,DNNs)的推荐模型占据了数据中心中 79%的 AI 推理周期[12]。典型的深度学习推荐模型,例如,DLRM(Deep Learning Recommendation Model)[13],由底层的多层感知机(Multi-Layer Perceptron,MLP)和多个嵌入表、中间的特征交互以及顶层的多层感知机组成,具体架构如图 2 所示[14]。底层的多层感知机负责将稠密特征映射成向量,而每个嵌入表则用于将高维的稀疏特征(例如,用户 ID、性别以及物品 ID 等)映射为低维的嵌入向量。中间的特征交互(例如,拼接)负责整合稠密特征和稀疏特征。顶层的多层感知机用于生成最终的预测结果。阿里巴巴的报告显示,推荐系统中超过 60%的预测延迟来自嵌入层[15]。而为了获得更好的推荐效果,推荐模型需要加入更多的稀疏特征,导致模型规模每年增长超过 1.5 倍[16]。

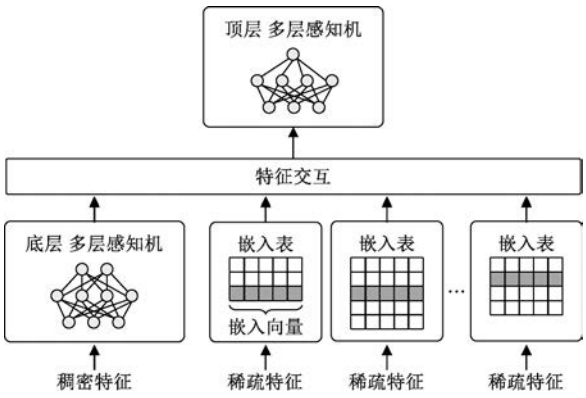

图 2　典型的深度学习推荐模型[14]

(2)图神经网络模型

图神经网络(Graph Neural Networks,GNNs)[17]是一种处理图数据的神经网络,能够应用在节点分类和链接预测等多种图任务中。在 GNN 中,每层网络包含聚合(Aggregate)和合并(Combine)两个步骤[18]:聚合步骤负责聚合邻域节点的特征,通常会使用平均和求和等操作;而合并步骤则负责合并目标节点的特征和聚合特征,通常会先拼接或者池化,然后喂入全连接层中。典型的 GNN 训练流程如图 3 所示[19]。其中,左侧为图结构,圆圈表示节点,数字表示节点编号,箭头表示节点之间的相邻关系。右侧展示了在目标节点 1 上执行 2 层 GNN 训练的过程:在第 1 层中,节点 1 的邻居节点 2、4 以及 5 分别聚合各自邻居节点的嵌入向量,并与自己的嵌入向量合并,形成新的嵌入向量。在第 2 层中,节点 1 聚合邻居节点 2、4 以及 5 的嵌入向量,并与自己的嵌入向量合并,形成新的嵌入向量。在生产环境中,图数据通常非常大,例如,字节跳动[20]的图数据包含 20 亿个顶点和 2 万亿条边,大小超过 100TB。GNN 训练需要递归遍历目标节点的 k 阶邻域,计算和存储的开销非常大。因此,现有的研究[21]使用了邻域采样,只选择部分子集。

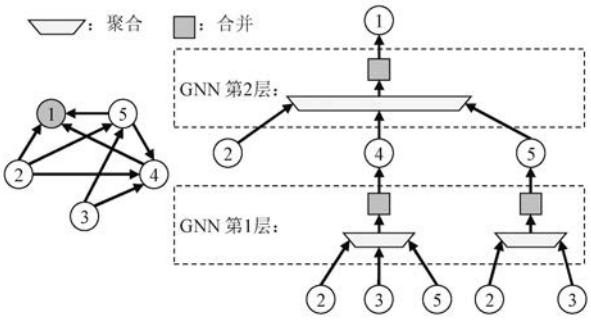

图 3　典型的 GNN 训练流程[19]

(3)Transformer 模型

2017 年,谷歌[22]提出了 Transformer 模型,并将其应用在机器翻译中,具体结构如图 4 所示。图中左侧和右侧分别表示编码器和解码器。编码器由 6 个相同的层堆叠而成,负责编码输入序列。其中,每个层包含了多头自注意力机制和逐位前馈网络 2 个子层。而解码器则根据编码器的输出和已经生成的部分序列(对应图中的"输出"),逐步生成最终的输出序列。它也由 6 个相同的层堆叠组成,每层包含掩蔽多头注意力机制、多头注意力机制以及前馈网络 3 个子层。在编码器和解码器中,每个子层均使用了残差连接[6](对应图中的"相加")和层归一化[23]。

对于输入数据 $\boldsymbol{X}$,注意力机制中查询矩阵 $\boldsymbol{Q}$、键矩阵 $\boldsymbol{K}$ 以及值矩阵 $\boldsymbol{V}$ 的计算方法如下所示:

$$\boldsymbol{Q}=\boldsymbol{X}\boldsymbol{W}_{\boldsymbol{Q}},\boldsymbol{K}=\boldsymbol{X}\boldsymbol{W}_{\boldsymbol{K}},\boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_{\boldsymbol{V}} \tag{1}$$

其中,$\boldsymbol{W}_{\boldsymbol{Q}}$、$\boldsymbol{W}_{\boldsymbol{K}}$ 以及 $\boldsymbol{W}_{\boldsymbol{V}}$ 是参数矩阵。Transformer 使用的自注意力计算方法如下所示:

$$Attention(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{T}}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{2}$$

其中,d_k 为查询向量和键向量的维度;归一化指数函数 $softmax(x_i)=\dfrac{\exp(x_i)}{\sum_{j=1}^{d_k}\exp(x_j)}$。多头注意力

机制的计算方法如下所示：

$$\begin{aligned} &MultiHead(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) \\ &=Concat(\boldsymbol{head}_1,\ldots,\boldsymbol{head}_h)\boldsymbol{W}^O \\ &\boldsymbol{head}_i=Attention(\boldsymbol{QW}_i^Q,\boldsymbol{KW}_i^K,\boldsymbol{VW}_i^V) \end{aligned} \quad (3)$$

其中，$Concat(\cdot)$ 表示拼接操作；$\boldsymbol{head}_i$ 为第 i 个注意力头的输出，h 为注意力头数；$\boldsymbol{W}_i^Q \in \mathbb{R}^{d_{model}\times d_k}$、$\boldsymbol{W}_i^K \in \mathbb{R}^{d_{model}\times d_k}$、$\boldsymbol{W}_i^V \in \mathbb{R}^{d_{model}\times d_v}$ 以及 $\boldsymbol{W}^O \in \mathbb{R}^{hd_v\times d_{model}}$ 是参数矩阵，d_{model} 和 d_v 分别表示子层的输出维度和值向量的维度。当前，Transformer 已经成为了许多 LLM 的基础结构。例如，OpenAI 在 2020 年提出了基于 Transformer 结构的 GPT-3（Generative Pre-trained Transformer 3）模型[24]，具有 1750 亿个参数。

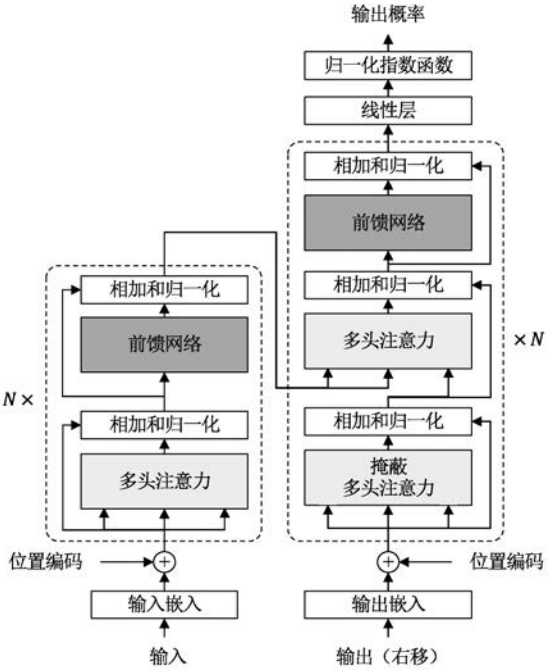

图 4 Transformer 模型结构[22]

2.1.3 分布式训练

随着数据集规模和模型参数规模的不断增长，使用单块 GPU 来执行训练，不仅耗时长、效率低，而且难以满足模型参数的存储需求。为了解决该问题，常见的做法是执行分布式训练。参数服务器（Parameter Server，PS）[25-26]是一种常见的分布式训练方法，典型架构如图 5 所示。参数服务器包含两类角色：工作者（Worker）和服务器（Server）。每个工作者负责处理部分训练数据（数据并行），而每个服务器则负责存储部分模型参数（模型并行）。在每次迭代中，工作者首先从服务器中拉取（pull）最新的模型参数，然后对其需要处理的部分数据执行训练流程，计算模型参数的梯度值，最后将梯度值推送（push）给存储模型参数的服务器。而服务器则负责聚合所有工作者推送的梯度值，并更新对应的模型参数。

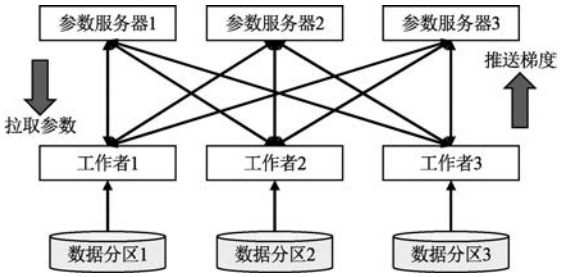

图 5 参数服务器架构[26]

2.2 深度学习的数据特点

2.2.1 数据规模及类型

深度学习在数据规模和数据类型方面的特点主要表现为：

（1）数据集规模呈现指数级增长，但未经处理前，数据集主要由小文件组成。根据文献[1]的统计，在 CV 和 NLP 领域，随着时间的推移，数据集的规模呈现指数级增长。此外，表 2 统计了部分数据集的样本大小。结果表明，样本的平均大小为数十字节到数百 KB。也就是说，如果没有执行额外的格式转换，数据集将主要由小文件组成。以 ImageNet-1K 为例，每个样本（图片）被存储为一个文件，平均大小为 114.18KB。

表 2 数据集文件大小统计

领域	数据集	数据量	总大小	平均大小
CV	MNIST①	70000	21.00MB	0.31KB
	CIFAR-10②	60000	132.40MB	2.26KB
	ImageNet-1K[5]	1431167	155.84GB	114.18KB
	Open Images V4[3]	1910098	562.42GB	308.75KB
	ImageNet-21K[2]	14197122	1.3TB	98.32KB
NLP	GLUE CoLA③	10657	965.49KB	0.09KB
	GLUE QQP	795241	150.37MB	0.19KB
	IMDB Reviews[27]	100000	487MB	4.99KB
	SQuAD2.0[28]	142192	148.54MB	1.07KB

（2）不同深度学习应用处理的数据类型各不相同。例如，计算机视觉应用需要处理图像和视频等数据、语音处理应用需要处理音频数据、自然语言处理应用需要处理文本数据、与图处理相关的深度学

① The MNIST database of handwritten digits，http://yann.lecun.com/exdb/mnist/ 2023，9，23。

② The CIFAR-10 dataset，https://www.cs.toronto.edu/~kriz/cifar.html 2023，9，23。

③ The General Language Understanding Evaluation（GLUE）benchmark，https://gluebenchmark.com 2023，9，23。

习应用需要处理图数据,而多模态大模型则需要处理多种类型的数据。

(3)各种类型的大模型越来越多。随着算力的不断提高,模型的参数规模也在不断变大,从百亿到千亿,再到万亿,甚至百万亿。例如,清华大学在2021年提出了具有174万亿参数的BaGuaLu模型[29]。此外,不同类型的大模型也相继出现,例如,大语言模型[30]、大视觉模型(Large Vision Models,LVMs)[31]以及同时支持语言和视觉等的多模态大语言模型(Multi-Modal Large Language Models,MM-LLM)[32-33]。

2.2.2　数据的特点

深度学习应用在数据方面的特点主要表现为:

(1)使用张量(Tensor)来表示数据。具体来说,只包含一个数字的张量称为0阶张量,即标量;由多个数字组成的数组是1阶张量,即向量,例如,[0, 1, 2, 3];由多个向量组成的数组是2阶张量,即矩阵;由多个矩阵组成的数组称为3阶张量;由多个3阶张量组成的数组就是4阶张量,其余高阶张量依次类推。

(2)中间数据存在着显著的稀疏性。一方面,为了引入非线性因素,深度学习模型通常会使用ReLU[34]等激活函数。然而,对于任意小于零的输入,$\mathrm{ReLU}(x)=\max(x,0)$都会输出零值,这会使神经元的输出产生40%~90%的稀疏度[35]。另一方面,为了防止模型过拟合(即在训练集上的准确率越来越高,但在测试集上的准确率却越来越低),深度学习模型通常会使用Dropout方法[36],即随机丢弃一些激活之后的神经元输出。然而,这也会使激活之后的神经元输出产生50%的稀疏度[35]。

(3)使用不同的数据格式来存储数据。不同的深度学习框架使用了特定的数据格式来存储数据集、模型参数以及模型结构。以TensorFlow[37]为例,它使用TFRecord格式①来存储数据集,使用SavedModel格式来保存模型参数和模型结构。为了提供统一的模型表示,Meta和Microsoft开发了开放神经网络交换(Open Neural Network Exchange,ONNX)格式,支持TensorFlow和PyTorch[38]等多种深度学习框架。

2.2.3　数据访问模式

在深度学习应用中,不同阶段具有不同的数据访问模式。因此,本文从数据准备阶段和模型计算阶段两个方面分别总结了深度学习应用的数据访问模式。

(1)数据准备阶段

数据准备阶段的数据访问模式主要表现为:

①随机读取训练集,顺序读取测试集。在训练过程中,训练任务只会读取数据集,不会修改或者删除数据集中的数据[39]。在每轮训练开始之前,训练任务通常会打乱训练集的读取顺序,以改进模型的泛化能力。因此,在每轮训练中,训练任务需要执行大量的随机读取操作。在每轮训练结束之后,训练任务将会顺序读取测试集中的所有样本,以评估模型的训练效果。

②批量执行读取操作,顺序读取每个样本。以ImageNet-1K为例,训练集包含128万张图片。假设批大小为512,那么每次迭代需要读取512张图片,而每轮训练则需要执行超过2500次迭代。此外,训练任务只会顺序读取单个样本,并完整读取该样本的全部数据[39]。

③存在着大量的元数据访问。在训练开始之前,训练任务需要读取数据集的相关信息,例如,统计训练样本的数量,确定每轮训练的迭代次数[39-40]。在每次迭代中,训练任务需要读取一批样本。在读取样本的数据之前,训练任务需要先访问样本的元数据[41]。

④存在着大量的数据共享。一方面,同一训练任务的不同轮次需要访问相同的数据集[11]。另一方面,不同训练任务也可能会使用相同的数据集或者同一数据集的不同版本[42]。例如,在超参数搜索中,同一模型的多个训练实例具有不同的超参数,但需要访问相同的数据集[11]。又如,由于部分数据集较为热门,许多训练任务可能都会使用它或者它的不同版本[11]。

(2)模型计算阶段

模型计算阶段的数据访问模式主要表现为:

①参数访问不均匀。当执行训练和推理时,深度学习应用会为每个样本关联特定的参数[43]。然而,许多应用场景存在着数据倾斜[43-44]。例如,对于文本数据,部分词的出现频率更高;对于图数据,一些节点的入度和出度更大。数据自身的倾斜导致了参数访问的不均匀。此外,许多训练任务还会使用采样技术[43],随机得到一些输入样本。这也会提高对应参数的访问频率。

②存在着大量的数据复用。在每次迭代中,反向传播会复用前向传播的计算结果,用于计算模型参数的梯度值[45]。例如,当计算第i层参数的梯度值时,训练任务会复用第$i-1$层的输出数据。

① TFRecord and tf. train. Example, https://www. tensorflow. org/tutorials/load_data/tfrecord 2023,3,22。

③存在着少量的写入操作。在训练过程中,训练任务可能会执行日志操作,将每次迭代的损失值和每轮训练的测试准确率等信息写入日志文件中[40-41]。此外,训练任务还会定期创建检查点,避免因为断电等突发问题导致计算结果丢失。检查点操作通常只会创建新文件,并执行追加写,不会执行覆盖写和并发写[39]。

2.3 研究框架

根据上文对深度学习背景知识和数据特点的分析,本文从深度学习的不同阶段出发,分析了深度学习的数据存储需求,提出了面向深度学习的数据存储技术研究框架,具体如图 6 所示。

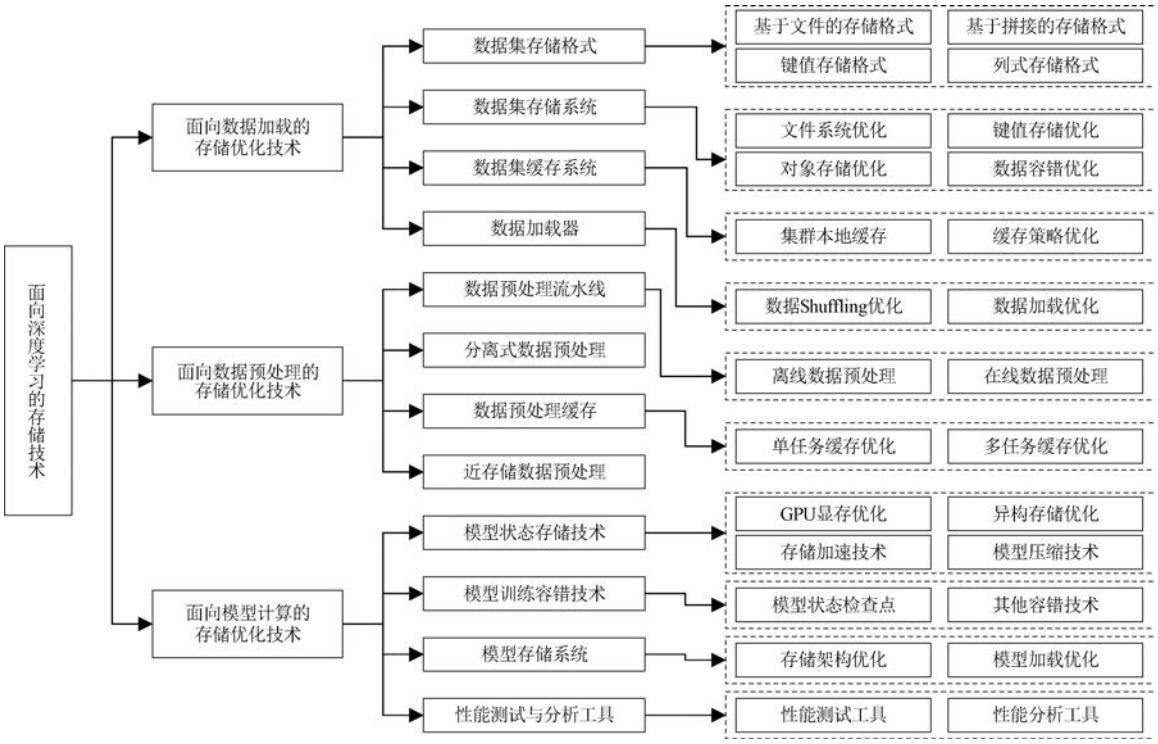

图 6 面向深度学习的数据存储技术研究框架

(1)数据加载阶段

在训练场景中,训练任务通常会打乱每轮训练的数据访问顺序,并将数据集样本从存储系统中分批加载到内存中。如果数据加载的速度慢于模型计算的速度,那么训练任务就会出现数据加载挂起的问题[46]。因此,如何高效加载数据集中的样本对深度学习训练的持续时间和资源利用率存在着重大影响。

目前,面向数据加载的存储优化研究主要从数据集存储格式、数据集存储系统、数据集缓存系统以及数据加载器四个方面展开。其中,数据集存储格式研究提出了拼接、键值存储以及列式存储等多种类型的存储格式,优化了数据集的读取性能;数据集存储系统研究提高了深度学习训练场景中文件系统、键值存储以及对象存储等的数据访问性能,设计了高效的数据容错机制;数据集缓存系统研究利用了计算节点的本地存储,优化了缓存策略,解决了数据集存储系统中样本加载延迟较高的问题;数据加载器研究从应用层出发,优化了数据 shuffling 和数据加载方法。

(2)数据预处理阶段

在训练场景中,当数据集中的样本被加载到内存时(在推理场景中,当收到新的输入数据时),CPU 需要执行多个数据预处理操作,将原始数据转换为模型需要的输入数据。如果数据预处理的速度慢于模型计算的速度,那么训练/推理任务就会出现数据预处理挂起的问题[46]。谷歌的应用也表明[4]:20%的训练任务花费超过三分之一的计算时间在输入流水线上。因此,如何加快数据预处理是加速深度学习训练和推理的主要内容之一。

目前,面向数据预处理的存储优化研究主要从数据预处理流水线、分离式数据预处理、数据预处理缓存以及近存储数据预处理四个方面展开。其中,数据预处理流水线研究提出以流水线的方式来执行数据预处理,实现了高效的离线数据预处理和在线数据预处理;分离式数据预处理研究使用单独的节

点来执行数据预处理操作,实现了数据预处理的水平扩展;数据预处理缓存研究提出缓存数据预处理操作的结果,缩短了数据预处理的完成时间;近存储数据预处理研究将部分数据预处理操作卸载到存储设备或者存储服务器中,缓解了 CPU 的数据预处理压力。

(3)模型计算阶段

在训练和推理的过程中,预处理后的数据将被送入加速器中进行计算。这涉及模型参数、激活数据、梯度以及优化器状态等模型状态的写入和读取。模型计算阶段存在着许多问题,例如,模型参数规模正在不断增长,远远超过了加速器的存储容量增长速度。又如,在训练过程中,计算节点可能会出现各种故障,导致训练任务频繁中断。如果没有合适的容错机制,最新的参数值将会丢失,消耗的计算资源也将被浪费。而如果容错机制的开销较大,训练任务也会浪费大量的计算资源。高效解决这些问题,有助于缩短深度学习的训练时间,降低深度学习的推理延迟。

目前,面向模型计算的存储优化研究主要从模型状态存储技术、模型训练容错技术、模型存储系统以及性能测试与分析工具四个方面展开。其中,模型状态存储技术优化了 GPU 显存的管理,利用了 DRAM、非易失性内存(Non-volatile Memory,NVM)以及固态硬盘(Solid State Drives,SSDs)等异构存储,设计了存储加速技术和模型压缩技术,解决了训练和推理的模型状态存储问题;模型训练容错技术利用了检查点、多副本、日志以及纠删码等方法,实现了高效的模型状态容错机制,避免了计算资源的浪费;模型存储系统研究实现了模型参数和模型结构等模型文件的高效存储,支持保存不同版本的模型文件;性能测试与分析工具研究设计了专门的 I/O 测试与分析工具,可以测试深度学习场景下数据存储的 I/O 性能,分析深度学习应用的 I/O 行为。

3　面向数据加载的存储优化技术

在存储数据集时,常见的做法是采用基于文件的数据存储格式,即将每个样本存储为一个单独的文件,PyTorch 默认使用这种数据存储格式。以 ImageNet-1K 为例,每个样本被存储为一个 JPEG 文件,而样本的标签则用目录名来表示,位于相同目录下的样本属于同一类别。然而,在每轮训练开始之前,深度学习应用会打乱训练集的读取顺序。因此,训练任务需要执行大量的、随机的小文件读取操作,训练性能低下。为了提高深度学习的数据访问性能,现有的研究从存储格式、存储系统、缓存系统以及数据加载器四个方面进行了优化。

3.1　数据集存储格式

根据存储布局的方式不同,本文将针对数据集存储格式的优化工作分为基于文件的存储格式优化、基于拼接的存储格式优化、键值存储格式优化以及列式存储格式优化。表 3 总结了不同数据集存储格式的差异。

表 3　数据集存储格式对比

数据集存储格式	需要专有 I/O 库	存储空间利用率	数据访问性能	易于查看单个样本	适宜的数据集规模	适用场景
基于文件的存储格式	否	低	低	是	小	通用
基于拼接的存储格式	是	高	高	否	任意	通用
键值存储格式	是	高	高	是	中	通用
列式存储格式	是	高	高	否	大	数据集特征较多

3.1.1　基于文件的存储格式

针对训练场景,Bae 等人[47]设计了加速器友好的无损图像格式 L3,提出在 GPU 等加速器中并行完成解码过程,降低了 CPU 对数据准备阶段的干扰。针对推理场景,Image Calculator[48]提出构造大规模的候选格式设计空间,并使用性能模型来搜索最合适的图像存储格式,以适配特定的深度学习任务。

3.1.2　基于拼接的存储格式

一些深度学习框架采用了 Record 布局,即将多个小文件拼接为一个大文件,从而将大量的、随机的小文件读取转换为少量大文件的顺序读取,提高了数据的读取性能。例如,TensorFlow 和 MXNet[49]分别设计了 TFRecord 格式和 RecordIO 格式①。基于类似的思想,一些研究提出了新的数据集存储格式。下面,本节将从数据布局和数据表示两个方面分别介绍相关的优化工作。

(1)数据布局优化。DIESEL[50]提出将数据集中的文件聚合成较大的数据块,并将包含的文件数

① Designing Efficient Data Loaders for Deep Learning, https://mxnet.apache.org/versions/master/api/architecture/note_data_loading.html 2023,5,6。

量、文件的偏移量和长度等元数据存储在数据块的头部。WebDataset[51]使用 tar 文件来表示数据集，将组成一个训练样本的所有文件(例如，图片文件和标签文件)存储在相邻的位置。Zhu 等人[52]修改了 HDFS(Hadoop Distributed File System)的存储格式，提出了由数据集中强相关文件组成的文件聚合结构 pile，设计了数据集－pile 两级索引结构，能够根据文件名查找目标 pile 的位置，并在 HDFS 数据块中记录了每个 pile 的偏移量。MindRecord 格式①由数据文件和索引文件组成：数据文件以页为单位来存储训练数据，而索引文件则用于快速检索数据集信息。FFCV[53]提出了 beton 格式，按页来组织数据集，能够存储任意的数据模态，并使用数据表来存放每个样本的元数据和指向样本的指针，易于访问数据集中的某些样本，其格式示例如图 7 所示。

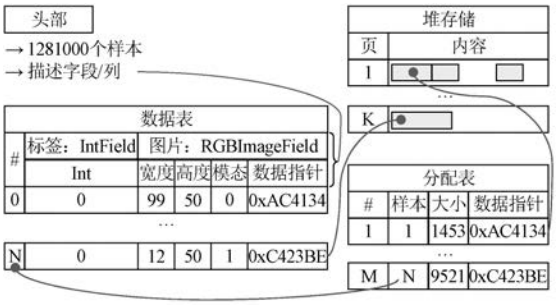

图 7 beton 格式示例[53]

(2)数据表示优化。一些研究使用了数据压缩技术，优化了存储格式的数据表示，减少了需要读取的数据量。FanStore[40]提出将数据集划分成多个分区，并设计了压缩器选择算法以挑选合适的无损压缩算法，用于压缩分区内的每个文件，最后再将压缩之后的文件拼接在一起。

Kuchnik 等人[54]发现不同训练任务能够容忍不同的数据压缩级别。基于这一发现，他们提出了 PCR(Progressive Compressed Record)格式，在 TFRecord 格式的基础上引入了渐进压缩，可以为每个训练任务动态选择合适的数据保真度，降低了数据的读取开销，具体格式如图 8 所示。以图片为例，图片的每个保真度等级称为一个扫描(scan)，保真度相同的图片组成了扫描组(scan group)。当需要读取特定保真度的数据时，用户只需要从 PCR 的开始位置依次读取每个扫描组中的扫描，直到读取完相应的扫描组为止。

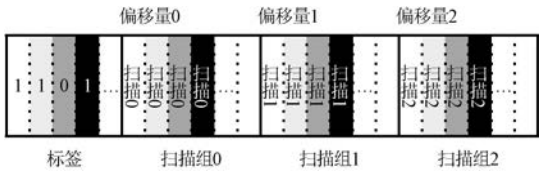

图 8 PCR 格式[54]

表 4 总结了基于拼接的存储格式在数据组织方式、支持的数据类型以及是否支持随机访问等方面的差异。

表 4 基于拼接的存储格式对比

方法	数据组织方式	支持的数据类型	随机访问
TFRecord	拼接成数据块	任意	不支持
RecordIO	拼接成数据块	图像	支持
DIESEL[50]	拼接成数据块	任意	不支持
WebDataset[51]	拼接成数据块	任意	不支持
pile[52]	拼接成数据块	任意	不支持
MindRecord	按页组织	任意	支持
beton[53]	按页组织	任意	支持
FanStore[40]	拼接成数据块	任意	不支持
PCR[54]	拼接成数据块	图像＋音频＋视频	不支持

案例分析：根据文献[54]的统计，当在 ImageNet-1K 上训练 ResNet18 和 ShuffleNetv2 时，如果使用默认的存储格式(即将每张图片存储为一个文件)，每轮训练需要超过 2 小时，比 TFRecord 格式慢了 25 倍；而与 TFRecord 格式相比，PCR 格式的训练时间进一步缩短了 50%。

3.1.3 键值存储格式

部分深度学习框架使用键值存储格式来存储数据集。例如，Caffe[55]使用基于 B＋树的 LMDB(Lightning Memory-Mapped Database)[56]和基于日志结构合并树的 LevelDB 来存储数据集。Lim 等人[57]的实验结果表明：当使用 ImageNet-1K 训练模型时，与基于文件的存储格式(PNG)相比，LevelDB 和 LMDB 等键值存储格式的训练时间缩短为十七分之一。

3.1.4 列式存储格式

除了上述格式，一些研究也提出使用列式存储格式来存储数据集。在 Apache ORC(Optimized Row Columnar)格式的基础上，Meta 公司开发了面向数据仓库的 DWRF 格式②，并使用其存储 AI 数据集[58]。因为 AI 场景通常具有大量的特征，将其存储为 DWRF 格式时，元数据开销较大。为了解决该问题，Meta 公司[59]又提出了 Alpha 格式，为元数据设计了定制的序列化格式，解码速度比 ORC 提

① 格式转换，https://www.mindspore.cn/tutorials/zh-CN/master/advanced/dataset/record.html 2023,5,10。

② DWRF file format for Hive，https://github.com/facebookarchive/hive-dwrf 2023,6,14。

高了 2-3 倍。

Deep Lake[60]采用了列式存储架构,将张量作为列。每个张量由一系列数据块组成,而每个数据块则是包含样本内容的二进制数据,一个样本表示为跨多个张量的一行。对于给定的样本索引,与每个张量关联的索引映射能够找到样本数据所在的数据块以及其在数据块中的具体位置。

3.2　数据集存储系统

若无其他处理,数据集通常会使用基于文件的存储格式。当使用这种数据集存储格式时,训练任务会执行大量的小文件随机读取操作,导致数据加载性能低下。为了缓解该问题,研究者们也提出了许多面向数据集的存储系统优化方法。

3.2.1　文件系统优化

在实际的训练场景中,数据集通常会存储在并行文件系统中。在训练过程中,训练任务需要随机访问存储在并行文件系统中的数据集,并且会多次加载数据集,存在着大量的数据和元数据访问,训练性能低下。因此,现有的工作从节点本地存储和远程共享存储两个方面进行了改进。

(1)基于节点本地存储的优化

基于节点本地存储的优化工作提出利用计算节点的本地存储,将数据集复制到计算节点中,避免了大量的远程访问。FanStore[39-40]提出将数据集分散存储到计算节点的本地存储中,并将元数据复制到每个计算节点中,还使用了系统调用拦截和数据压缩技术,提供了 POSIX(Portable Operating System Interface)兼容接口,减少了存储空间占用。

Kurth 等人[61]提出将数据集划分成多个不相交的部分,交给不同进程来读取,并在进程内部运行多个线程,并行读取远程存储系统中的文件。在数据集被读取到节点本地存储之后,每个计算节点使用点对点 MPI(Message Passing Interface)消息将每个文件的副本分发到其他需要使用该文件的计算节点中,消除了从远程存储系统中读取数据集的性能瓶颈。

DLFS(Deep Learning File System)[62]提供了一套面向深度学习应用的简单 API(Application Programming Interface),设计了基于内存树的样本目录以实现快速的元数据管理,实现了基于 SPDK(Storage Performance Development Kit)的 I/O 服务,通过 NVMe over Fabrics 协议支持用户态的存储分离,还提出了样本级和数据块级的机会性批量读取方法。

Schimmelpfennig 等人[63]设计了面向深度学习的工作流,使用了突发缓冲文件系统 GekkoFS[64],实现了高效的数据导入,降低了并行文件系统的负载,并为所有计算节点提供了共享的全局命名空间,能够无偏地访问所有数据。

(2)基于远程共享存储的优化

基于远程共享存储的优化工作提出直接优化远程共享存储,提高了训练任务的远程数据访问性能。DIESEL[50]协同设计了存储系统和缓存系统,解耦了元数据处理和存储,为每个数据集引入了元数据快照,加快了元数据访问,设计了任务粒度的分布式缓存,将节点故障造成的影响限制在每个深度学习训练任务内部,还提出将小文件合并为大数据块,缩短了缓存系统的恢复时间。DIESEL+[65]扩展了 DIESEL,引入了 GPU 辅助的图片解码和在线感兴趣区(Region of Interest,ROI)方法,减少了图片解码的工作量,降低了节点之间的数据移动开销。

Zhao 等人[58]使用分布式文件系统 Tectonic[66]来存储训练数据。然而,Tectonic 使用机械硬盘(Hard Disk Drives, HDDs)来存储训练数据,每瓦电量提供的 I/O 性能低下。因此,Zhao 等人[67]进一步提出了复合存储架构 Tectonic-Shift,整合了闪存存储层 Shift 和 Tectonic,使用 SSD 来提高读取性能,提高了存储能效,减少了需要的 HDD 容量,还设计了新颖的缓存方法,利用从训练数据集中获取的信息来推测训练任务未来的数据访问模式。

案例分析:根据文献[58]的统计,在 Meta 的 PB 级生产集群中运行 9 小时之后,与仅使用 HDD 存储训练数据相比,Tectonic-Shift 的能效提高了 29%。

3.2.2　键值存储优化

Caffe 使用 LMDB 来存储数据集。具体来说,LMDB 使用了内存映射(mmap)机制,将整个数据集映射到内存中,在缺页中断时需要与进程调度系统和文件系统进行交互,存在着进程之间的 I/O 竞争和大量的上下文切换[68]。此外,LMDB 使用 B+树来存储数据,只支持顺序访问:当访问数据时,LMDB 需要从根节点开始,依次遍历路径中的每个分支节点,直到找到目标数据[69]。针对上述情况,一些研究对 LMDB 进行了优化。

LMDBIO[68]提出了局部内存映射(localized mmap)方法,在每个节点中选择一个进程作为根进程,负责从文件系统中读取数据,并使用共享内存将数据分发给节点中的其他进程,减少了进程之间的 I/O 竞争和上下文切换的次数,允许中断处理程序

准确唤醒正在等待 I/O 完成的进程。

当多个进程访问数据库的不同部分时，LMDB 和 LMDBIO 均存在着冗余的数据移动[69]。因此，LMDBIO-DM[69]提出了由使用可移植光标表示的串行 I/O 和推测并行 I/O 组成的两步法，具体如图 9 所示。其中，4 个读取进程（P0-P3）分别处理数据集的 4 个部分（D0-D3）。在第一步中，LMDBIO-DM 会将数据库映射到所有进程的相同虚拟地址中，然后串行执行所有进程。而每个进程则先读取待处理的数据，然后将停止位置信息（光标）传递给下一个进程。在第二步中，LMDBIO-DM 会先预测每个进程需要处理的页数和读取偏移量，然后将这些页并行加载到内存中，接着执行第一步的顺序寻址和光标切换，最后再执行实际的数据访问，弥补了串行执行带来的性能损失。

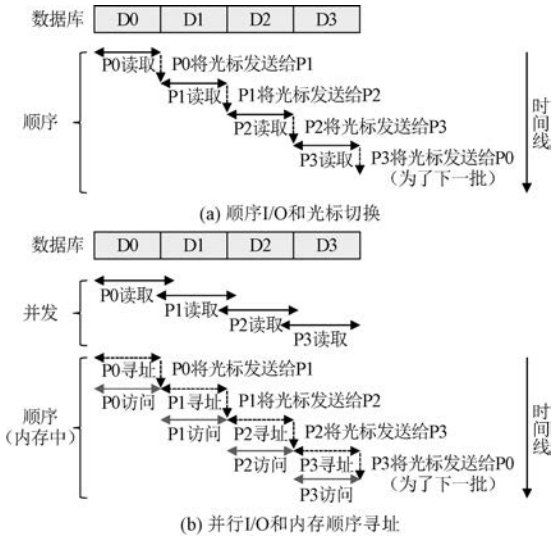

图 9 LMDBIO-DM 示意图[69]

在此基础上，Pumma 等人[70]进一步提出了 4 种优化方法。方法 1 改进了 LMDBIO，提出不再使用 mmap 来拷贝数据，而是使用直接 I/O，先计算数据的地址偏移量，然后使用 pread 来读取数据，消除了 mmap 对数据读取的影响。方法 2 改进了 LMDBIO-DM，使用额外的数据库溯源信息，动态计算数据库的布局，可以准确推断数据的位置，消除了 LMDB 的顺序移动限制，解决了 I/O 的串行问题。方法 3 提出合并多次迭代需要的数据，增加每个 I/O 操作的块大小，以便充分发挥文件系统的 I/O 性能。方法 4 提出限制同时发起的 I/O 操作数量，降低了 I/O 操作的随机化（以不确定的顺序访问文件），并提供了足够的 I/O 并行。

除了优化 LMDB，一些研究还提出了面向深度学习的键值存储系统。针对分布式 GNN 训练，DistDGL[71]提出了分布式内存键值存储 KVStore，用于存储图中节点和边的特征数据。KVStore 支持灵活的数据分区策略，并使用共享内存和同一机器上运行的训练进程进行交互，还开发了支持快速网络通信的远程过程调用（Remote Procedure Call，RPC）框架，使用了零拷贝机制来实现数据序列化和多线程收发接口。

案例分析：根据文献[68]的统计，当使用 CIFAR10-Large 训练 AlexNet 模型时，与 Caffe 相比，Caffe-LMDBIO 的训练时间最高缩短了 42.9%；而当使用 ImageNet-1K 训练 CaffeNet 模型时，Caffe-LMDBIO 的训练时间最高缩短了 95.2%。

3.2.3 对象存储优化

当前，部分训练数据存储在对象存储中。然而，对象存储的语义不同于文件系统，单线程数据传输性能较低，直接从对象存储中读取数据将会拖慢训练[72]。因此，一些研究优化了深度学习场景下对象存储的访问性能。

Ozeri 等人[72]使用 s3fs 将 POSIX 接口转换为对象接口，提出将深度学习框架的文件读取请求转换为对多个数据块的并发读取请求，实现了数据预读机制，在每次请求中读取整个对象，而不只是请求的那部分数据，还使用了内存缓存，将从对象存储中读取的数据块缓存在计算节点的内存中，并异步返回给深度学习框架，避免了多次读取相同的数据。

AIStore[51]是 NVIDIA 提出的、面向深度学习的对象存储。AIStore 支持横向扩展，提供了类似 S3（Simple Storage Service）的对象接口，使用 HTTP GET 和 PUT 操作来读写用户数据，并利用 HTTP 重定向来直接访问存储服务器中的对象。此外，AIStore 还支持使用 Amazon S3 和 Google 云存储等多种对象存储作为后端，实现了端到端的数据保护、多副本以及纠删码，集成了 MapReduce 扩展，能够根据用户指定的排序方式和请求的分片大小对数据集进行重新分片。

在实际的生产服务中，每隔一段时间都会导入新的数据集[73]。上述工作假定数据集已经存储在对象存储中，没有考虑将大数据集上传到对象存储时产生的开销。针对该问题，Targoat[73]提出先在客户端将包含许多文件的数据集合并为一系列 tar 文件，然后在服务端抽取出原始的小文件，并上传到存储网关中，缩短了数据集的上传时间。

3.2.4　数据容错优化

针对训练数据容错问题,PRM(Partial-Recovery Method)[74]组合了纠删码方法获取数据块分布全局信息的能力和 AI 方法恢复部分丢失数据的能力,不仅降低了训练数据的恢复开销,还提供了较高的训练准确率。具体来说,PRM 会先收集模型参数、纠删码参数以及数据存储信息(条带中的数据块分布和条带分布信息),然后监控每类数据的参数更新大小,接着使用 AI 方法来计算数据恢复的优先级,最后组合计算得到的恢复优先级和纠删码提供的数据分布全局信息,并行恢复数据条带。

3.3　数据集缓存系统

虽然上述工作已经提出了许多面向深度学习的存储系统优化方法,但是从存储系统中加载训练数据仍然存在着较高的 I/O 延迟。因此,一些研究提出在深度学习应用和存储系统之间构建缓存系统。表 5 总结了不同数据集缓存优化方法的差异。

表 5　数据集缓存系统对比

分类	方法	特点	影响模型准确率	适宜的数据集规模	适用场景
节点本地缓存	单节点缓存	利用单个计算节点的本地存储,缓存完整或部分数据集	否	小	单节点训练
	分布式缓存	聚合多个计算节点的本地存储,缓存完整或部分数据集	否	大	分布式训练
缓存策略优化	智能预取	计算数据访问顺序,并按照数据访问顺序来预取样本	否	任意	通用
	数据替换	随机选择已经缓存的样本来替换需要的样本	是	任意	通用
	重要性缓存	缓存最重要的样本,并且多学习这些样本	是	任意	通用
	缓存分配	同时考虑不同任务的 I/O 和缓存需求	否	任意	多训练任务

3.3.1　节点本地缓存

一些研究提出利用计算节点的本地存储来缓存并行文件系统中的训练数据。根据针对的训练类型不同,本文将其分为针对单节点训练的单节点缓存和针对分布式训练的分布式缓存。

(1)单节点缓存

Monarch[75-76]使用计算节点的本地存储来缓存完整或者部分的数据集,设计了面向深度学习 I/O 模式的异步数据放置策略,还提出从并行文件系统中预取大文件的内容,提高了深度学习的训练性能,缓解了并行文件系统的 I/O 压力,能够适应不同的深度学习框架,不需要修改训练脚本。

(2)分布式缓存

Hoard[77]提出在任务开始之前或者任务初始执行的过程中将远程存储系统中的数据缓存到计算节点的本地存储中,然后在同一任务的后续轮次或者共享相同数据集的不同训练任务之间(例如,超参数搜索)使用缓存的数据。DeepMemoryDL[78]使用基于 CXL(Compute Express Link)的内存作为快速的中间存储层级,主动预取训练数据,并将其缓存到计算节点的合适层级中。HVAC(High-Velocity AI Cache)[79]聚合了计算节点的本地存储和近节点本地存储(near node-local storage)以缓存并行文件系统中的大数据集,使用了分布式哈希来定位缓存数据,避免了元数据瓶颈,还利用环境变量 LD_PRELOAD 来拦截系统调用,保证了应用代码的可移植性,具体架构如图 10 所示。

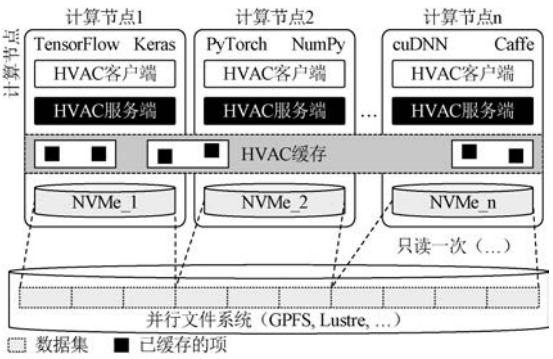

图 10　HVAC 架构[79]

案例分析:根据文献[79]的统计,在超级计算机 Summit 上,HVAC 可以扩展至 1024 个节点;与 GPFS 相比,HVAC 的训练时间平均缩短了 25%。

3.3.2　缓存策略优化

除了利用计算节点的本地存储来缓存数据外,一些研究也提出优化缓存策略。根据使用的技术不同,本文将缓存策略优化分为智能预取、数据替换、重要性缓存以及缓存分配。

(1)智能预取

一些研究提出计算数据访问顺序,并按照数据访问顺序来预取样本。Prisma[80]提出将存储优化与深度学习框架解耦,并转移到专有的存储层中。在数据平面中,Prisma 实现了并行的数据预取机制,按照事先计算出的数据访问顺序,使用多个线程,并发读取训练数据,并将其存储在内存缓冲区中,用于服务后续的 I/O 请求。在控制平面中,Prisma 实现了自动调参控制算法,使用反馈控制回

路,收集数据平面的缓冲区使用情况,并不断调整读取线程的数量和缓冲区的大小,直到找到最优值。

NoPFS[81]使用了透视(clairvoyance)的思想:给定生成数据访问模式的伪随机数生成器种子,我们可以提前知道某个样本会在什么时候被哪个进程访问。基于该思想,NoPFS 对分布式机器学习的数据访问模式进行了分析,从概率上描述了不同工作者对相同样本的不同访问频率,并设计了性能模型驱动的分布式缓存和预取策略,可以生成数据到缓存层次的接近最优映射,能够适应不同规模的数据集和各种存储层次(例如,节点内存、节点本地 SSD 以及分布式内存等)。此外,NoPFS 还实现了基于性能模型的 I/O 性能模拟器,能够比较各种场景中不同 I/O 策略的性能,还可以分析系统变化对训练时间的影响。

(2)数据替换

一些研究发现只要输入的数据是随机的,具体的读取顺序并不重要,进而提出了数据替换方法,随机选择已经缓存的样本来替换需要的样本。

DeepIO[82]设计了输入流水线,重叠了模型训练和小批量生成,提出了熵感知的机会性顺序,只从内存缓冲区中选择小批量中的样本,还提供了可移植的存储接口,支持不同的后端存储系统。Zhu 等人[83]扩展了 DeepIO 框架,支持在单个计算节点中运行多个工作者,引入了多个读取缓冲区,以区分不同工作者的请求和数据。

Quiver[11]是微软印度研究院提出的面向深度学习训练的缓存系统。首先,Quiver 使用了基于内容哈希的寻址方式,能够在使用相同数据集的多个训练任务和多个用户之间透明地重用已经缓存的数据。其次,Quiver 提出了可替换的缓存命中,从缓存中查询更多的样本,使用缓存中的样本来生成小批量,避免了缓存抖动,具体如图 11 所示。最后,Quiver 提出了协调缓存驱逐、协作缓存不命中处理以及收益感知的缓存替换策略,优先将缓存空间分配给那些能够从缓存中获得最大收益的训练任务。

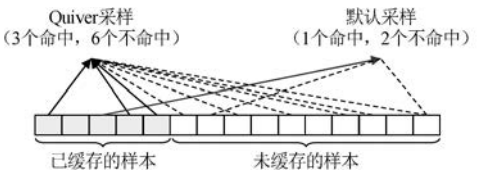

图 11 可替换的缓存命中[11]

(3)重要性缓存

现有的研究表明[84]:不同样本对模型训练的重要性并不相同。因此,一些研究提出了基于重要性采样的缓存策略,缓存最重要的样本,并且多学习这些样本。

iCache[85]提出将缓存空间划分成缓存重要样本的 H-cache 和不重要样本的 L-cache。对于 H-cache,iCache 设计了基于样本重要性的缓存替换算法(如图 12 所示,方框的颜色越深表示样本的重要性越高),驱逐不重要的样本,并在样本重要性更新之后重新填充缓存。对于 L-cache,iCache 设计了动态打包技术,将不重要的样本批量加载到 L-cache 中,并使用数据替换方法来处理缓存不命中的情况。此外,iCache 还提出了多任务处理机制,根据每个任务的缓存收益和样本重要性,重新计算样本的相对重要性,避免了缓存抖动。

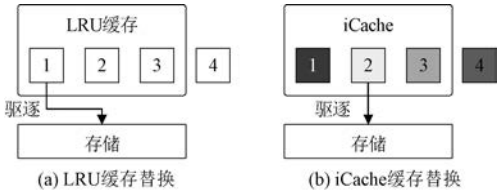

图 12 LRU 和 iCache 对比[85]

SHADE[86]提出检测样本级的重要性变化,并利用其来管理缓存。在控制层中,SHADE 提出了损失分解策略,使用小批量级损失来更新权重,使用样本级损失来计算样本的重要性,设计了基于排序的重要性评估方法以计算不同小批量中每个样本的相对重要性,还提出了基于优先级的自适应数据采样策略,在一轮训练中多次访问重要的样本。在数据层中,SHADE 实现了自适应的优先级感知预测缓存策略,能够动态更新每个样本的重要性,确保最重要的样本在缓存中。

(4)缓存分配

不同训练任务对缓存的需求并不相同,例如,受到计算限制的任务对缓存并不敏感,而受到 I/O 限制的任务则需要分配更多的缓存空间。因此,一些研究提出优化缓存分配,同时考虑不同任务的 I/O 和缓存需求。

Fluid[87]引入了面向异构数据源的云原生数据集抽象和缓存系统自动调优方法,设计了实时训练速度感知的缓存自动伸缩机制,提出了基于缓存共享的多任务调度策略。Synergy[88]使用了乐观分析来预测不同训练任务的 CPU 和内存需求,设计了接近最优的启发式调度机制,能够根据训练任务的资源分配情况,在多租户集群上调度训练任务,缓解

了数据挂起问题,缩短了平均任务完成时间。SiloD[89]提出协同设计集群调度器和缓存子系统,设计了增强的任务性能预测器,利用训练的数据访问模式来分析不同训练任务的缓存和远程 I/O 需求,联合分配计算资源和存储资源(包括缓存和远程 I/O),同时保留调度策略的原始目标。对于不依赖性能预测器的调度器,SiloD 提出了贪心的缓存分配策略,利用不同训练任务的异构缓存效率,无需修改调度器。Li 等人[90]提出了针对 GPU 固定分配场景的缓存和带宽分配算法 CBA(Cache and Bandwidth Allocation),权衡了存算分离场景中存储集群的带宽和计算集群的缓存,并设计了针对 GPU 动态扩展场景的 AutoCBA 算法,同时考虑了 I/O 和计算,能够选择缓存、带宽、GPU 以及任务特定效用函数的最优组合。表 6 总结了上述方法在资源分配、是否支持缓存自动伸缩以及是否支持多任务缓存共享等方面的差异。

表 6　缓存分配方法对比

方法	资源分配	缓存自动伸缩	缓存共享
Fluid[87]	缓存	是	是
Synergy[88]	缓存+计算	否	否
SiloD[89]	缓存+远程 I/O+计算	否	是
Li 等人[90]	缓存+带宽+计算	是	是

案例分析:根据文献[88]的统计,在具有 32 块 V100 GPU 的物理集群上,与按 GPU 数量成比例分配其他资源相比,Synergy 的平均任务完成时间(Job Completion Time,JCT)降低了 33.3%,第 99 百分位 JCT 降低了 50%。

3.4　数据加载器

在深度学习框架中,数据加载器(DataLoader)主要完成以下几个工作:第一,数据 shuffling,即在每轮训练开始之前,打乱训练集的访问顺序;第二,数据加载,即将样本从存储系统中分批加载到内存中;第三,数据预处理,即在模型计算之前,对样本进行数据预处理。例如,对图片进行解压缩和裁剪等。其中,与数据加载过程有关的是数据 shuffling 和数据加载。因此,本节将仅介绍与数据 shuffling 和数据加载相关的工作,而关于数据预处理的内容将在第 4 章中单独进行说明。

3.4.1　数据 Shuffling 优化

根据数据的打乱程度不同,现有的数据 shuffling 方法可以分为全局 shuffling、局部 shuffling 以及部分－局部 shuffling。全局 shuffling 方法会打乱训练集中全部样本的访问顺序。该方法可以提高模型的泛化能力,但 I/O 开销很大。与全局 shuffling 相比,局部 shuffling 和部分－局部 shuffling 降低了数据访问的随机程度,提高了数据加载的性能,但会影响模型的泛化能力。表 7 总结了三类数据 shuffling 方法的差异。

表 7　数据 Shuffling 方法对比

方法	访问顺序随机程度	影响模型准确率	I/O 开销	支持单节点训练	分布式训练		
					节点之间交换数据	交换的数据量	网络开销
全局 shuffling	高	否	大	是	是	多	高
局部 shuffling	低	是	小	是	否	无	无
部分－局部 shuffling	中	是	中	否	是	少	中

(1)全局 Shuffling

DeepIO[82]提出了基于远程直接内存访问(Remote Direct Memory Access,RDMA)的原地 shuffling 操作,将数据集存放在服务节点的内存中,并通过 RDMA 读取操作暴露给其他服务节点。当进入下一轮训练时,数据的访问顺序易于重新打乱,无需从后端存储中重新加载数据。

(2)局部 Shuffling

对于单节点训练,TensorFlow 采用了典型的局部 shuffling 方法①。首先,顺序访问一部分样本,填充内存中的缓冲区;然后,从缓冲区中随机选择一批样本,并使用新的样本替换选中的样本。当缓冲区的大小大于等于数据集的大小时,该方法将变为全局 shuffling。而对于采用数据并行的分布式训练场景,典型的局部 shuffling 方法只会打乱划分给每个工作者的那部分数据。

DIESEL[50]提出将数据集中的文件聚合成多个数据块,设计了逐块 shuffle 方法。首先,在内存中打乱所有数据块的 ID 顺序;然后,将数据块划分成多个组;接着,读取每个数据块组中的文件名,打乱文件名的先后顺序;最后,合并所有组中的文件列表,得到最终的文件读取顺序列表,具体如图 13 所示。

FFCV[53]使用了拟随机抽样方法:先分配一个足够大的缓冲区,能够容纳数据集的 batch_size 个页,然后打乱数据集中所有页的顺序,并按照打乱之后的页顺序填充缓冲区,只从缓冲区中的样本生成一个批次。

① shuffle, https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle 2024,7,18。

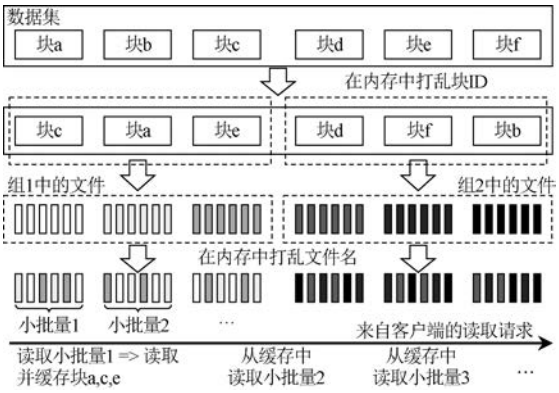

图 13 逐块 Shuffle[50]

LIRS[91]利用 SSD 的快速随机访问特性来执行随机 shuffling,直接从存储中访问选出的训练样本,并打包成一个批次。首先,LIRS 提出了数据格式感知的位置生成器,根据存储格式来获取每个训练样本的位置。然后,LIRS 提出了页感知的随机 shuffling,当训练样本的大小小于操作系统虚拟页的大小时,使用页作为最小打乱单位,将相同页中的训练样本分在相同批次。如果训练样本的大小超过了虚拟页的大小,则使用训练样本作为打乱单位。最后,LIRS 还提出了基于填充的页对齐,如果某个训练样本的内容跨页,则在页尾填充 0 以对齐训练样本,避免了冗余的页访问。

(3)部分—局部 Shuffling

针对采用数据并行的分布式训练场景,Nguyen 等人[92]提出了 partial-local shuffling(PLS)方法。具体来说,在每轮训练开始之前,每个工作者将随机选择的部分样本发送给其他工作者,并从其他工作者处接收相同数量的新样本。对于待发送的每个样本,工作者会随机选择一个目的工作者。在数据传输完成之后,工作者将删除发送给其他工作者的那些样本,并将收到的新样本保存到本地存储中。最后,工作者将对本地存储的样本执行局部 shuffling。为了降低样本交换的开销,PLS 方法重叠了上一轮的前向与反向传播和当前轮的样本交换,具体流程如图 14 所示。其中,isend 和 irecv 分别表示 MPI 中的非阻塞发送和接收函数。

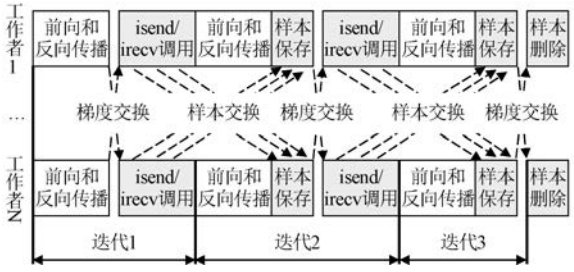

图 14 重叠样本交换和前向/反向传播示意图[92]

案例分析:根据文献[92]的统计,当使用 ImageNet-1K 训练 ResNet50 时,与全局 Shuffling 相比,PLS 方法实现了相同的准确率,但是每个工作者只需要存储 0.03%的数据集(当使用 4096 个工作者时);而与局部 Shuffling 相比,PLS 方法的训练准确率则提高了 10%。

3.4.2 数据加载优化

根据使用的优化方法不同,本文将针对数据加载机制的优化工作分为以下四类:第一,并行 I/O,即并行加载训练样本;第二,数据预取,即在训练任务处理当前批次时,预先加载后续批次的训练样本;第三,数据缓存,即缓存已经加载过的训练样本,避免重复加载;第四,GPU 直接访问存储,即允许 GPU 直接访问外存、内存或者其他显存中的训练样本。

(1)并行 I/O

Kurth 等人[61]使用了 Python 的多进程模块 multiprocessing,将 TensorFlow 使用的并行工作线程转换为并行工作进程,让每个进程都有自己的 HDF5(Hierarchical Data Format 5)[93]库实例,以解决 HDF5 库串行执行读取操作的问题,并行读取并处理输入数据。PyTorch DataLoader 支持使用多进程并行加载多批样本。ConcurrentDataloader[94]进一步提出并行加载一个批次中的每个样本,以提高数据加载的吞吐量。

(2)数据预取

在执行数据 shuffling 之后,当前轮次的样本访问顺序是已知的。因此,现有的研究提出根据样本的访问顺序,预先加载训练样本,重叠当前批次的训练和后续批次的加载。Lee 等人[95]提出了面向分布式训练的异步 I/O 策略,为每个进程创建专门的 I/O 线程,一次读取一组训练样本,并使用双重缓冲区方法来重叠 I/O 和计算。Serizawa 等人[96]提出重叠数据暂存(从远程共享存储中复制训练数据)和小批量生成(从节点本地存储中读取训练数据,并生成小批量)。IPDL[97]提出在训练当前批次时,创建预取线程,预先从远程存储系统中加载下一批数据。PaGraph[98]提出流水线执行数据加载和 GNN 计算,在计算当前批次时预取下一批图特征数据。表 8 总结了不同数据预取方法的差异。

表 8 数据预取方法对比

方法	每次预取数量	并行预取	适用模型
Lee 等人[95]	预取一组	否	通用
Serizawa 等人[96]	预取一批	是	通用
IPDL[97]	预取一批	否	通用
PaGraph[98]	预取一批	否	GNN

(3)数据缓存

现有的数据缓存方法可以分为两类:第一,动态缓存,即执行缓存替换,以提高缓存命中率;第二,静态缓存,即不执行缓存替换,以避免缓存替换带来的开销。表 9 总结了现有数据缓存方法的差异。

表 9　数据缓存方法对比

方法	缓存对象	执行缓存替换	适用模型
Lobster[99]	样本	是	通用
SOLAR[100]	样本	是	CNN
AliGraph[101]	节点和边的特征	是	GNN
Ginex[19]	节点特征	是	GNN
BGL[20]	节点特征	是	GNN
MariusGNN[102]	图分区	根据任务决定	GNN
MinIO[46]	样本	否	通用
Yang 等人[103]	样本	否	通用
PaGraph[98]	节点特征	否	GNN
Chen 等人[104]	样本	否	元学习等

①动态缓存。Lobster[99]提出了灵活的线程管理策略,可以协调数据加载和预处理之间的资源使用,能够平衡相同节点内不同 GPU 之间的 I/O 负载,并使用性能建模和启发式方法将线程管理策略和分布式缓存结合起来,实现了基于重用距离的缓存驱逐策略。针对卷积神经网络(Convolutional Neural Networks,CNN),SOLAR[100]提出在使用数据复用之后,将需要从并行文件系统中加载的样本均分给不同的节点来处理,牺牲计算平衡(每个节点的批大小不同)来换取数据加载平衡,并将相同批次中具有局部性的多个样本访问聚合成一次数据块加载,提高了 HDF5 的并行加载吞吐量。针对 GNN 训练,AliGraph[101]提出分开存储属性异构图的结构信息和特征,并缓存频繁访问的节点特征和边特征。Ginex[19]提出将特征向量缓存在主存中,实现了最优的缓存替换策略,能够在单台机器上处理十亿级别的图数据集。BGL[20]提出了特征缓存引擎,使用了 FIFO(First-In First-Out,先进先出)策略,设计了邻近性感知的采样顺序,在相邻的小批量中尝试访问图中相邻的训练节点,还构建了基于多块 GPU 显存和主存的两级缓存。针对链接预测任务,MariusGNN[102]提出了图分区替换策略 COMET,使用了物理和逻辑两级分区,分离了数据存储/访问和数据传输,解耦了小批量生成和分区替换,还提供了设置超参数的自动化规则。

②静态缓存。CoorDL[46]提出了针对单节点训练的 MinIO 缓存(如图 15 所示),在缓存空间用完之后,不执行缓存替换,减少了每轮训练的缓存不命中次数,降低了 I/O 开销,还设计了针对分布式训练的分区 MinIO 缓存,协调节点之间的远程 MinIO 缓存,消除了冗余的数据加载。Yang 等人[103]使用了分布式缓存技术,允许参与训练的工作者互相共享本地缓存(内存和 SSD),提出了局部性感知的数据加载方法,使用全局小批量中本地已经缓存的样本来构建本地小批量,并通过数据交换来实现工作者之间的负载均衡。PaGraph[98]使用 GPU 显存来缓存频繁访问节点(出度最大的那些节点)的特征数据。针对节点分类任务,MariusGNN[102]提出将所有训练节点及其特征向量缓存在主存中。针对具有多次数据加载规则的训练场景(例如,元学习),Chen 等人[104]提出了稳定(steady)缓存策略,将数据放置问题转换为 0/1 背包问题,基于数据的使用频率和大小来缓存最有价值的数据,不会执行缓存替换,还支持多任务场景,基于每个任务的数据需求级别来分配缓存资源。

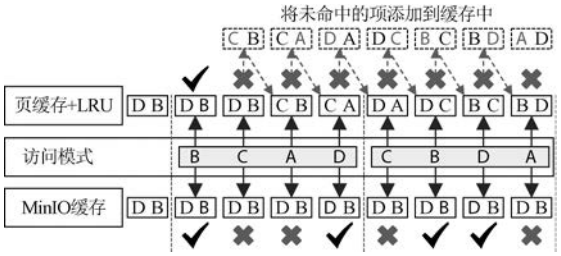

图 15　两轮训练中页缓存和 MinIO 的缓存命中情况对比[46]

(4)GPU 直接访问存储

在传统的数据访问过程中,数据首先需要由 CPU 从外存读取到主存中,然后再由 CPU 将数据从主存复制到 GPU 显存中。不论是外存访问,还是内存访问,数据访问路径都比较长。因此,一些研究提出了 GPU 直接访问存储方法,绕开了 CPU,缩短了 GPU 和存储设备之间的数据传输路径,避免了额外的数据拷贝。为了便于描述,本小节只会介绍直接访问数据集的相关工作,而与模型状态相关的工作将在 5.1.3 节中进行介绍。

NVIDIA 提出的数据加载库 DALI① 使用了 GPU 直访外存技术 GPUDirect Storage (GDS),加快了深度学习应用的数据加载。PyTorch-Direct[105]通过统一内存(Unified Memory,UM)来访问主存中的 GNN 特征(数据传输流程如图 16 所示),设计了循环移位索引优化,实现了内存访问对齐,引入了统一张量类型,并对 PyTorch 的内存分配器、转发

① NVIDIA DALI,https://github.com/nvidia/dali 2024,1,9。

逻辑以及放置规则进行了必要的修改。XGNN[106]提出了全局 GNN 内存存储，统一了 GPU 显存和 CPU 内存，将图结构和特征数据分散存储在多块 GPU 和 CPU 内存中，还提供了易于使用的数据访问 API。GIDS(GPU Initiated Direct Storage Access)[107]提出将较大的图特征数据存储在外存中，允许 GPU 线程直接访问外存中的特征数据，设计了动态存储访问累加器，解耦了图采样和模型训练，能够自动确保足够的并发存储访问数量，并将较小的图结构数据固定在内存中，利用统一虚拟寻址(Unified Virtual Addressing，UVA)的零拷贝传输，在 GPU 中执行图采样，还使用部分内存作为固定缓冲区来存储少量热门节点的特征数据，使用 GPU 显存来缓存最近访问节点的特征数据，设计了窗口缓冲技术，避免驱逐可以复用的节点特征向量。表 10 总结了上述方法在是否支持 GPU 直访外存、内存以及其他 GPU 显存等方面的差异。

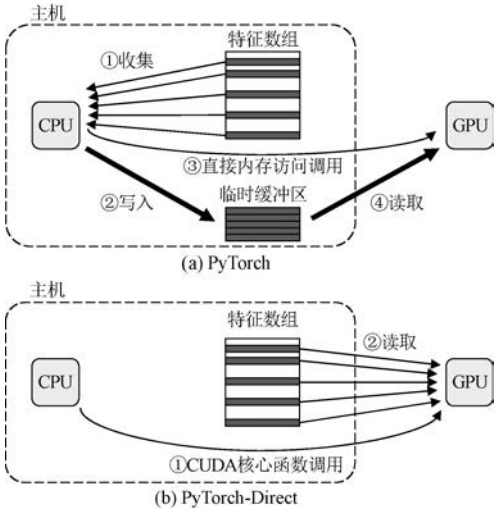

图 16 PyTorch 和 PyTorch-Direct 数据传输机制对比[105]

表 10 GPU 直接访问存储方法对比(数据集)

方法	直访外存	直访内存	直访显存	适用模型
DALI	是	否	否	通用
PyTorch-Direct[105]	否	是	否	GNN
XGNN[106]	否	是	是	GNN
GIDS[107]	是	是	否	GNN

案例分析：根据文献[105]的统计，当使用 wiki 等数据集训练 GraphSAGE 等模型时，PyTorch-Direct 的训练时间比 PyTorch 最高缩短了 38.2%，能耗最高降低了 17.5%。

3.5 总结与对比

图 17 总结了现有的、面向数据加载的存储优化技术。其中，数据集存储格式研究将原始的数据集转换为更加高效的存储格式，进而改变了样本数据

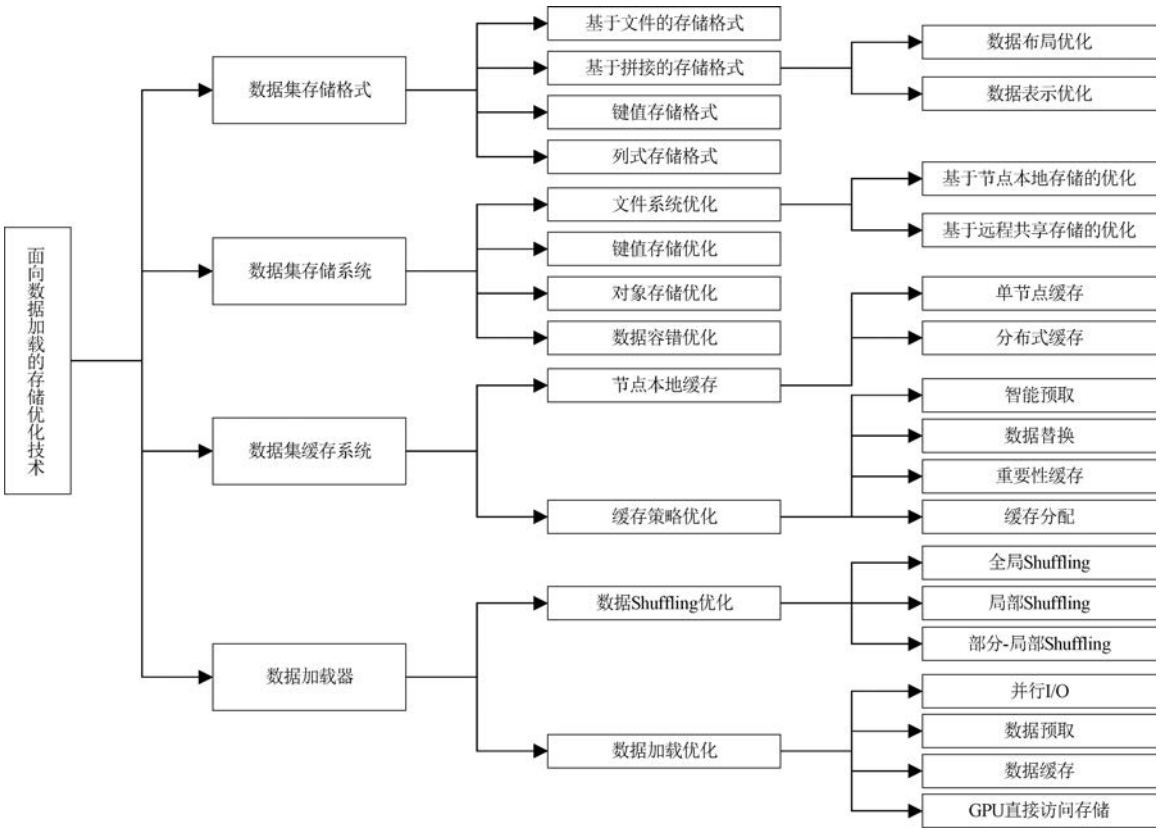

图 17 面向数据加载的存储优化技术

的访问模式。数据集存储系统研究优化了存储系统的随机读取和容错性能,降低了从存储设备到深度学习应用的数据访问延迟。数据集缓存系统研究弥补了存储系统和深度学习应用之间的性能鸿沟,充当了额外的加速层。而数据加载器研究则权衡了数据加载性能和模型准确率,优化了小批量内部和小批量之间的数据加载性能。以上四类优化技术并不是孤立的,彼此之间可以相互补充,共同提高训练任务的数据加载性能。

4　面向数据预处理的存储优化技术

4.1　数据预处理流水线

数据预处理分为离线数据预处理和在线数据预处理。现有的研究采用流水线方式来执行数据预处理,将前一个数据预处理操作的输出作为下一个数据预处理操作的输入,并将数据预处理阶段的最终输出作为模型计算阶段的输入,具体流程如图 18 所示。

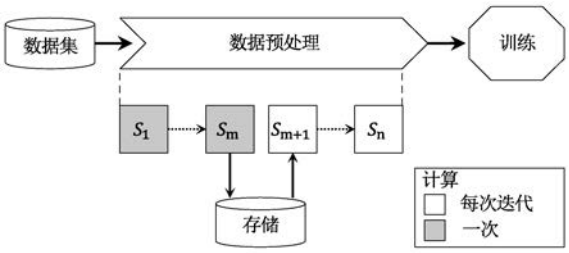

图 18　训练场景的数据预处理流水线[1]

Isenko 等人[1]分析了训练场景中不同领域的数据预处理流水线,设计了数据预处理流水线分析工具 PRESTO,能够基于用户定义的目标函数自动选择最优的预处理策略,即在何处将数据预处理流水线分割为离线数据预处理和在线数据预处理。

4.1.1　离线数据预处理

离线数据预处理包括格式转换、数据压缩以及图分区等。在格式转换中,原始的数据集将被转换为 3.1 节中介绍的各种数据集存储格式,以提高模型训练的数据加载性能。对于这部分内容,本节不再赘述。在数据压缩中,数据集中的样本将会经过各种有损或者无损压缩,以降低模型训练的数据移动开销。对于 GNN 训练,现有的研究会离线执行图分区,将图切分成多个分区,并构建邻域缓存。表 11 总结了现有方法在离线执行的数据预处理操作和适用模型方面的差异。

表 11　离线数据预处理方法对比

方法	离线执行的数据预处理操作	适用模型
Ibrahim 等人[108]	有损压缩和无损压缩	CNN
Behme 等人[109]	有损压缩	CNN
Liquid[110]	有损压缩和无损压缩	CNN
AliGraph[101]	图分区＋构建邻域缓存(k 阶出近邻)	GNN
Ginex[19]	构建邻域缓存(1 阶入近邻)	GNN
PaGraph[98]	图分区＋构建邻域缓存(L 阶入近邻)	GNN
BGL[20]	图分区	GNN
SOLAR[100]	重排不同轮次的先后顺序＋改变样本和节点之间的映射关系	CNN

(1)数据压缩。Ibrahim 等人[108]提出了领域特定的数据压缩方法,减少了需要传输的数据量,降低了存储系统与计算节点之间的数据移动开销。Behme 等人[109]发现:当训练时间或者存储容量有限时,有损图像压缩能够提高训练任务的吞吐量和准确率,而压缩质量和编解码器的选择则非常重要。Liquid[110]提出利用多种图像格式(包括有损压缩、无损压缩以及不压缩)来平衡训练流水线,通过离线分析来寻找不同格式的最优混合比例,生成包含多种图像格式的数据集,并在训练时使用格式感知的批采样方法。

(2)图分区与邻域缓存。AliGraph[101]提出离线将图切分成多个分区,分散存储在不同的工作者中,并使用 k 阶入近邻(in-neighbors)和出近邻(out-neighbors)数量的比值来衡量节点的重要程度,缓存重要节点的 1 到 k 阶出近邻,降低了图遍历开销。Ginex[19]使用 1 阶入近邻和出近邻的比值来挑选重要的节点,缓存这些节点的 1 阶入近邻,并将邻域缓存保存到 SSD 中,减少了邻域采样的 I/O 请求数量。在邻域缓存结构中,Ginex 使用了直接寻址,包含地址表和缓存数组 2 个数组。地址表的元素数量等于节点总数,每个元素包含了查询缓存数组的索引,负数表示缓存不命中,而缓存数组的元素则包含了缓存节点的邻居数量和每个邻居节点的编号,具体结构如图 19 所示。以节点 1 为例,地址表中索引为 1 的元素值为 3,而缓存数组中索引为 3 的元素值为 4。这说明节点 1 有 4 个邻居节点,而 4 个节点的编号则是缓存数组中后续 4 个元素的值,即 0、2、4 以及 9。PaGraph[98]提出了 GNN 计算感知的图分区算法,确保每个分区包含相似数量的训练节点以平衡 GPU 之间的训练负载,并为每个分区中的每个训练顶点复制 L 阶入近邻,以避免训练时跨分区访问数据。BGL[20]设计了新颖的图分区

算法,尽量在每个分区中保留多跳连接,并维持分区之间的负载均衡,支持扩展到大规模的图上,降低了子图采样期间的跨分区通信开销。

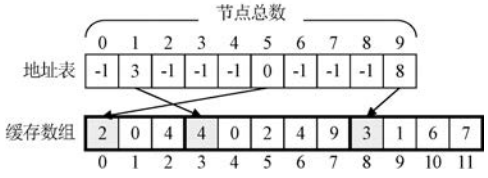

图 19 Ginex 邻域缓存结构示例[19]

(3)其他。SOLAR[100]提出离线生成全部轮次的样本索引列表,重新排列不同轮次的先后顺序,以优化缓存驱逐机制,并重新排列节点和样本之间的映射关系,将样本重新映射到前一轮访问该样本的节点中,以优化数据局部性。

4.1.2 在线数据预处理

虽然许多数据预处理操作可以离线完成,但是训练任务仍然需要在线执行部分数据预处理操作[4],例如,数据增强(Data Augmentation),即对样本执行随机变换,增加样本的数量和多样性[111]。当前,学术界和工业界已经提出了许多在线数据预处理方法。

tf.data[4]是 TensorFlow 提供的数据处理框架。它实现了许多通用算子,可以按照用户定义的函数来参数化,支持组合,能够在不同领域中复用。tf.data API 由无状态的数据集和有状态的迭代器组成:数据集抽象可以供用户定义自己的输入流水线,而迭代器则用于生成数据序列,并记录当前数据在数据集中的位置。tf.data 内部会将输入流水线表示为数据流图,并使用图重写来执行静态优化。此外,tf.data 还能够自动调整参数的取值,可以避免手动调参。

PyTorch DataLoader 使用多进程并行执行不同批次样本的数据预处理操作,并流水线执行数据预处理和模型训练。Yang 等人[103]进一步提出在进程内部使用多线程并行预处理一批样本。SpeedyLoader[112]提出使用启发式方法将不同样本的预处理请求划分成耗时长的请求和耗时短的请求两类,避免耗时长的样本阻塞耗时短的样本,并根据预处理耗时长短将预处理后的样本异步添加到不同的队列中,同时让 GPU 从这些队列中读取预处理后的样本。A-Dloader[113]提出了面向分布式训练的吞吐量预测模型,设计了在线工作者分配算法,可以为并发运行的分布式训练任务动态分配 CPU 和 I/O 资源。

针对 GNN 训练,BGL[20]提出了基于性能分析的资源分配方法,将资源分配问题当作一个优化问题,使用性能分析来找出每个数据预处理阶段的资源需求,隔离了不同阶段需要的资源,减少了不同阶段之间的资源竞争。MariusGNN[102]提出了数据结构 DENSE,使用了邻域采样增量编码,复用采样的一阶邻域来构造不同层的输入,减少了采样时的冗余计算,还使用了优化过的稠密核心函数来加快前向传播。

表 12 总结了上述在线数据预处理方法在并行方式、是否支持动态资源分配以及适用模型等方面的差异。

表 12 在线数据预处理方法对比

方法	并行方式	动态分配资源	适用模型
tf.data[4]	多线程	是	通用
PyTorch DataLoader	多进程	否	通用
Yang 等人[103]	多进程+多线程	否	通用
SpeedyLoader[112]	多线程	否	CNN
A-Dloader[113]	工作者	是	通用
BGL[20]	多线程	是	GNN
MariusGNN[102]	多线程	否	GNN

4.2 分离式数据预处理

过去,深度学习框架在相同的节点中执行数据预处理与模型计算,这种方式叫作共置(colocated)数据预处理[114]。随着加速器的计算速度越来越快,数据预处理的速度难以满足模型训练的需求,从而延长了训练的时间,浪费了宝贵的计算资源。因此,一些研究提出了分离式数据预处理,在其他节点上执行数据预处理,实现了数据预处理的水平扩展。

tf.data service[114]是 TensorFlow 提供的分离式数据预处理服务,架构如图 20 所示。其中,客户端先向转发器注册数据预处理任务,然后转发器会将数据预处理任务分发给多个工作者,并将工作者的网络地址发送给客户端,最后客户端则从工作者中获取预处理后的小批量数据。此外,tf.data service 还提供了多种数据集分片策略,包括不分片、动态分片以及静态分片,实现了转发器和工作者的容错机制,设计了面向分布式训练的协调读取技术,可以避免由于客户端之间输入数据大小不同而出现掉队者(straggler)。

DPP(Data PreProcessing Service)[58]是 Meta 提出的分离式数据预处理服务。DPP 分为控制平面和数据平面。其中,控制平面由一个主控节点组成,负责工作分发、容错以及自动伸缩。数据平面由工作者和客户端组成。工作者负责执行数据预处理

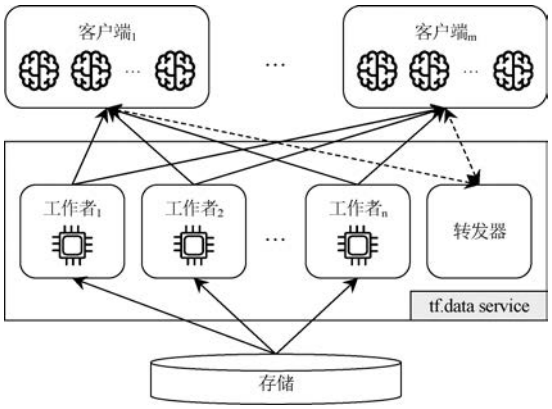

图 20　tf. data service 架构[114]

任务,而客户端则运行在每个训练节点上,暴露一个钩子,供 PyTorch 调用以获取预处理后的张量。

GoldMiner[115]是阿里巴巴提出的分离式数据预处理服务。GoldMiner 提出了自动的图分区算法,能够从训练程序中自动抽取无状态的数据预处理计算,并将数据预处理交给数据工作者来处理,实现了数据预处理和模型训练的分离式执行。此外,GoldMiner 还实现了集群调度器,利用数据工作者的弹性伸缩来调整资源分配,以满足不同任务的需求,提高整个集群的效率。

然而,简单地卸载所有数据预处理操作不一定能加快训练速度,例如,当没有数据预处理挂起问题时,则不需要卸载数据预处理操作[116]。因此,FastFlow[116]提出根据应用特定的性能指标和分配的资源,自动决定什么时候卸载、卸载哪些操作以及卸载多少数据,同时利用本地和远程的 CPU 资源,并与 TensorFlow 无缝集成,无需修改主要逻辑。类似地,Pecan[117]提出了自动放置策略,能够在本地和远程资源之间动态调度数据预处理工作者以减少需要的远程工作者数量,设计了自动排序策略,可以自动重排数据转换操作的先后顺序以增加每个工作者的吞吐量。cedar[118]提供了易于使用的编程接口,允许用户使用支持任意框架和库的可组合算子来定义数据流水线,自动执行复杂且可扩展的优化技术(包括算子卸载、缓存、预取、融合以及重排等),并在可扩展的本地和分布式执行后端中编排数据预处理。

表 13 对比了上述方法在是否使用本地 CPU、是否支持自动卸载数据预处理以及是否支持重排数据预处理操作的先后顺序等方面的差异。

表 13　分离式数据预处理方法对比

方法	使用本地 CPU	自动卸载	重排操作顺序
tf. data service[114]	否	否	否
DPP[58]	否	否	否
GoldMiner[115]	否	否	否
FastFlow[116]	是	是	否
Pecan[117]	是	是	是
cedar[118]	是	是	是

案例分析:根据文献[114]的统计,在 3 个 CV 生产模型和 ResNet50 的训练中,使用 tf. data service 实现水平扩展之后,训练时间平均缩短了 96.8%,训练成本平均降低了 96.2%;在 4 个 NLP 生产模型的训练中,使用 tf. data service 的协作读取之后,训练时间平均缩短了 54.5%。

4.3　数据预处理缓存

4.3.1　单任务缓存优化

在单个任务中,每轮训练都使用了相同的数据集[11]。为了利用多轮训练之间的数据共享特性,一些研究提出了面向单任务的数据预处理缓存方法,缓存数据预处理操作的输出结果,避免重复计算。

谷歌[119-120]提出了数据重复(data echoing)方法,复用预处理后的数据,减少了 CPU 计算,加快了模型训练。Revamper[121]提出了数据翻新(data refurbishing)方法,将原始的数据增强流水线划分为部分增强和最终增强两个阶段,如图 21 所示。Revamper 会缓存部分增强阶段生成的中间结果,并在复用一定次数之后重新生成新的中间结果,以保证样本的多样性,然后在最终增强阶段中对中间结果执行剩余的数据增强,产生完全增强的样本。

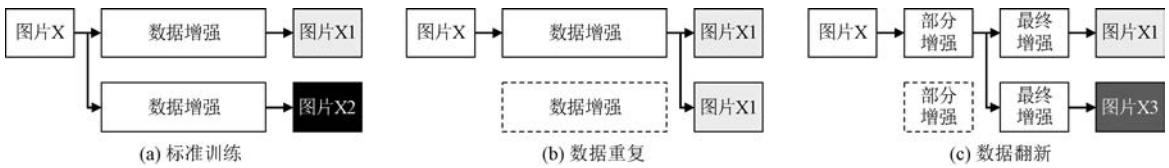

图 21　标准训练、数据重复以及数据翻新对比[121]

然而,简单地复用预处理后的数据无法实现最优的测试准确率。因此,PerFect[122]利用了两阶段训练:在预训练阶段,使用数据复用来训练模型;而在微调阶段,则不使用数据复用,继续训练模型,以

实现期望的准确率。

4.3.2 多任务缓存优化

当执行超参数搜索时，同一任务的多个实例需要访问相同的数据集[11]。另外，当执行多个训练任务时，不同任务也可能会使用相同的数据集，或者存在部分重叠的数据集，例如，相同数据集的不同版本[42]。为了利用任务之间的数据共享特性，一些研究提出了面向多任务的数据预处理缓存方法。

针对超参数搜索场景，CoorDL[46]提出了协调预处理技术，能够协调单个节点中的多个超参数搜索任务。具体来说，在一轮训练中，每个任务预处理数据集的不同分片，并将预处理后的小批量数据暂存在内存中，以便其他任务复用该批数据。在一批数据被所有任务都使用一次之后，CoorDL 将从暂存区中删除该批数据，确保不会在多轮训练之间复用预处理后的数据。

针对通用场景，tf.data service[114]实现了临时数据共享功能，支持在多个并发训练任务之间复用预处理后的数据。Joader[42]提出在使用重叠数据集的不同训练任务之间共享加载和预处理后的数据，设计了依赖采样算法和依赖采样树，提高了任务之间的采样局部性，实现了缓存驱逐策略 RefCnt，驱逐未来最不可能使用的数据。OneAccess[123]提出统一处理训练集群中的数据加载，避免多个任务重复加载相同的数据，并将预处理后的数据顺序写入磁盘中，消除了重复的计算，还使用了蓄水池采样，在创建样本时顺序访问外存，在生成小批量时随机访问主存。Cachew[124]提出动态扩展各种资源以匹配训练任务的训练速度，并自动执行缓存操作，在同一训练任务的多轮训练之间和不同训练任务之间复用原始数据集和预处理后的数据。MMDataLoader[125]支持在同一服务器的多个并发训练任务之间复用预处理后的数据（如图 22 所示），提出了分支流水线结构以提高数据复用，设计了速度感知的小批量生成策略以满足每个任务的数据需求，并实现了相应的缓存驱逐策略以应对训练速度的差异，还会缓存调整大小之后的数据以减少不同轮次之间的重复计算。

表 14 总结了上述方法在缓存粒度、是否支持超参数搜索以及是否支持不同任务等方面的差异。

案例分析：根据文献[125]的统计，当在 ImageNet-1K 上训练 3 个 AlexNet 模型时，与 PyTorch 相比，MMDataLoader 的吞吐量最高提升了 3.15 倍；而当同时训练 AlexNet、SqueezeNet 以及 ResNet18 等 3 个不同模型时，MMDataLoader 在 AlexNet 上的吞吐量最高提升了 2.28 倍，在 SqueezeNet 和 ResNet18 上则最高提升了 2.04 倍。

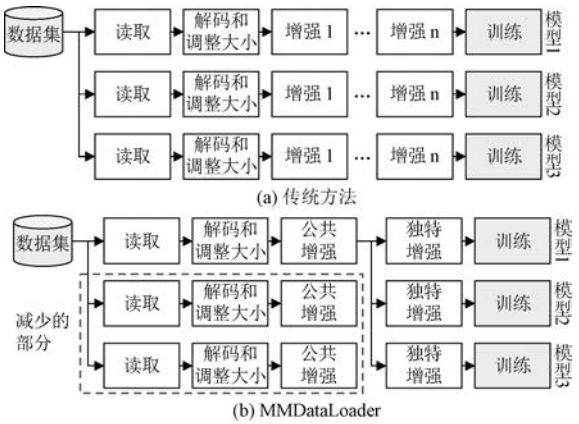

图 22 传统方法和 MMDataLoader 对比[125]

表 14 多任务缓存优化方法对比

方法	缓存粒度	支持超参数搜索	支持不同任务
CoorDL[46]	小批量	是	否
tf.data service[114]	小批量	是	是
Joader[42]	样本	是	是
OneAccess[123]	样本	是	是
Cachew[124]	数据集	是	是
MMDataLoader[125]	小批量	是	是

4.4 近存储数据预处理

一些研究提出利用近存储/存储内处理，将部分数据预处理操作从 CPU 卸载到计算型存储（Computational Storage）或者存储服务器中，缓解了 CPU 的数据预处理压力，降低了数据移动开销。除了卸载数据预处理操作外，部分工作还将部分或者全部模型计算操作卸载到计算型 SSD 或者存储服务器中。为了便于描述，本节只会介绍卸载数据预处理操作的工作，而同时卸载数据预处理和模型计算的工作将在 5.1.3 节中进行说明。

SmartSAGE[126]使用 NVMe SSD 来存储图数据，能够支持超过主存大小的 GNN 训练。在硬件方面，SmartSAGE 使用了存储内处理架构，将邻域采样卸载到 SSD 中，降低了 SSD 和 DRAM 之间的数据移动开销。在软件方面，SmartSAGE 设计了延迟优化的软件运行时系统和主机驱动栈，使用直接 I/O 来访问 SSD，并将邻域采样的多个 I/O 命令合并成一个 NVMe 命令。

Li 等人[127]使用降维操作来降低预处理数据的大小，并将降维操作卸载到计算型存储中，降维流程和计算单元布局如图 23 所示。具体来说，Li 等人使用了随机投影（Random Projection，RP）方法，其

计算方式为 $\boldsymbol{C}=\boldsymbol{AB}$ 。其中,$\boldsymbol{A}\in\mathbb{R}^{n\times d}$ 为待降维的矩阵,$\boldsymbol{B}\in\mathbb{R}^{d\times k}$ 为随机矩阵,$\boldsymbol{C}\in\mathbb{R}^{n\times k}$ 为降维之后的输出矩阵。在执行降维操作时,系统先将 CPU 预处理后的数据从 DRAM 传输到计算型存储中;然后,计算型存储中的计算单元将对预处理数据执行降维操作,并存储缩减之后的预处理数据;最后,缩减之后的预处理数据将被传输到 GPU 中。在系统实现方面,Li 等人使用了多个计算单元来并行计算输出矩阵的切片,在每个计算单元内采用了双缓冲区方法,重叠了矩阵 $\boldsymbol{A}/\boldsymbol{B}$ 的切片写入和切片读取,并在切片数据传输到计算单元之前重新排列了输入矩阵,以便顺序访问切片数据,还实现了异步 API、流水线执行 I/O 和计算。

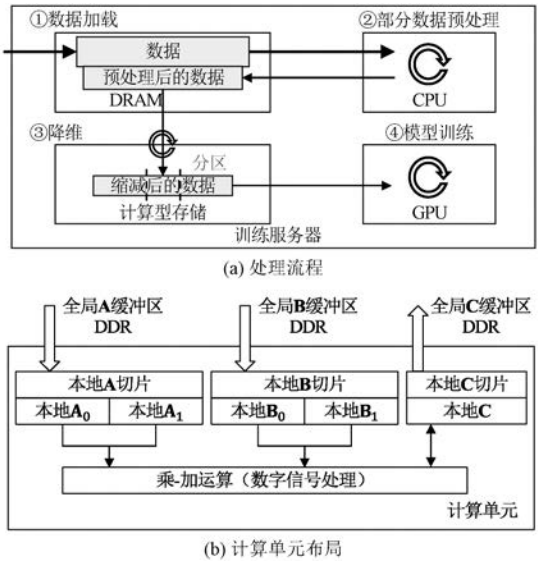

图 23　近存储降维加速[127]

PreSto[128]是韩国科学技术院提出的、面向推荐模型训练的存储内数据预处理系统。PreSto 利用数据预处理中存在的特征间并行和特征内并行,将耗时的特征生成(使用原始的特征数据生成新的特征)和特征归一化操作卸载到存储设备中。

Wang 等人[129]发现许多样本在预处理流水线的中间阶段达到了最小大小,进而提出了预处理卸载框架 SOPHON,选择性地将部分样本的部分预处理操作卸载到远程存储服务器中,传输更小且部分预处理后的样本,降低了远程存储服务器和计算节点之间的数据传输开销。

表 15 总结了上述方法在卸载的数据预处理操作和适用场景上的差异。

表 15　近存储数据预处理方法对比

方法	卸载的数据预处理操作	适用场景
SmartSAGE[126]	邻域采样	GNN 训练
Li 等人[127]	降维	使用大数据集或高维数据的模型训练
PreSto[128]	特征生成和特征归一化	推荐模型训练
SOPHON[129]	部分样本的部分操作	远程 I/O 是瓶颈

案例分析:根据文献[128]的统计,当使用 Criteo 数据集训练推荐模型时,与以 CPU 为中心的系统相比,PreSto 的数据预处理时间平均缩短了 89.6%,成本效益平均提升了 4.3 倍,能效平均提高了 11.3 倍。

4.5　总结与对比

图 24 总结了现有的、面向数据预处理的存储优化技术。其中,数据预处理流水线研究将数据预处理流水线划分成两个部分,离线执行不影响模型准确率且只需要执行一次的数据预处理操作,并以流水线的方式执行在线数据预处理和模型计算。分离式数据预处理研究解耦了数据预处理和模型计算,使用其他节点来执行数据预处理操作。数据预处理缓存研究利用相同训练任务不同轮次之间和多个训练任务之间的数据共享特性,缓存预处理后的数据,权衡了数据预处理性能和模型准确率。而近存储数据预处理研究则利用了存储设备或者存储服务器的计算能力,卸载了部分数据预处理操作,但难以适应不同的模型。表 16 对比了上述存储优化技术的优缺点和适用场景。

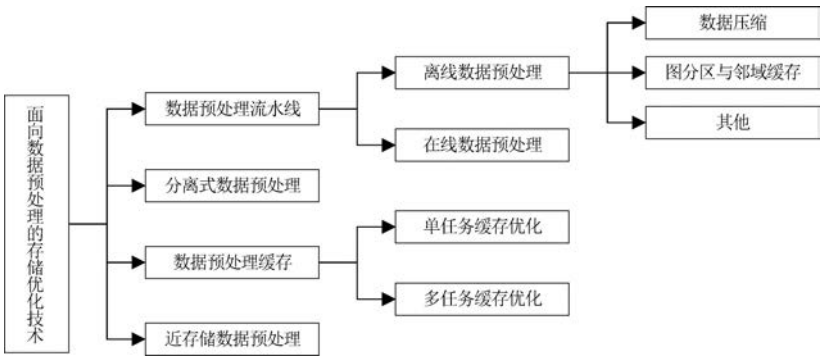

图 24　面向数据预处理的存储优化技术

表 16 面向数据预处理的存储优化技术对比

分类	方法	典型示例	优点	缺点	适用场景
数据预处理流水线	离线数据预处理	Liquid[110]、AliGraph[101]、Ginex[19]、PaGraph[98]等	数据预处理操作只会执行一次	需要占用大量的存储空间	静态数据集
	在线数据预处理	tf. data[4]、PyTorch DataLoader、BGL[20]、MariusGNN[102]等	流水线执行在线预处理和模型计算	每次迭代都要执行数据预处理操作	通用
分离式数据预处理		tf. data service[114]、DPP[58]、GoldMiner[115]、FastFlow[116]等	实现了数据预处理的水平扩展	部署成本和能耗高	本地数据预处理速度无法满足训练需求
数据预处理缓存	单任务缓存优化	data echoing[119,120]、Revamper[121]、PerFect[122]等	利用了不同轮次之间的数据共享特性	影响模型准确率	本地数据加载/预处理速度较慢
	多任务缓存优化	CoorDL[46]、tf. data service[114]、Cachew[124]、MMDataLoader[125]等	利用了多个训练任务之间的数据共享特性	可能存在缓存竞争	多个任务存在数据集共享
近存储数据预处理		SmartSAGE[126]、Li 等人[127]、PreSto[128]、SOPHON[129]等	将部分数据预处理操作卸载到存储设备或存储服务器中，降低了数据移动开销	难以适应不同的任务	数据访问频繁＋预处理慢

5 面向模型计算的存储优化技术

5.1 模型状态存储技术

随着深度学习模型的参数量越来越多，模型参数、中间数据（包括梯度和激活数据等）以及优化器状态等模型状态的大小经常会超过 GPU 等加速器的存储容量，该问题称为内存墙。为了解决该问题，现有的研究提出了许多模型状态存储技术。表 17 对比了不同方法的优缺点和适用场景。

表 17 模型状态存储技术对比

分类	方法	典型示例	优点	缺点	适用场景
GPU 显存优化	分布式显存优化	GPipe[130]、PipeDream[131]、Megatron-LM[132]、ZeRO-DP[133]等	将参数分散存储在多块 GPU 的显存中	能耗高，资源利用率低	大模型训练或推理
	KV 缓存显存优化	vLLM[134]、vAttention[135]、Infinite-LLM[136]等	提高了 KV 缓存的存储效率，避免出现显存碎片	引入了额外的性能开销	大模型推理（LLM）
异构存储优化	基于 DRAM 的异构存储优化	vDNN[137]、SuperNeurons[138]、SwapAdvisor[139]、Capuchin[140]等	使用 DRAM 来增强 GPU 的存储容量	存在内存带宽和存储容量的竞争	GPU 显存无法满足模型的存储需求
	基于 NVM 的异构存储优化	OpenEmbedding[141]、PetPS[142]等	使用 NVM 来增强 GPU 的存储容量	访问带宽较低，延迟较高	GPU 显存＋DRAM 无法存下模型
	基于 SSD 的异构存储优化	FlashNeuron[45]、Behemoth[143]、ZeRO-Infinity[144]、StrongHold[145]等	使用 SSD 来增强 GPU 的存储容量	访问带宽低，延迟高	GPU 显存＋DRAM 无法存下模型
	基于 CXL 内存的异构存储优化	COARSE[146]、ReCXL[147]、CXL-PNM[148]、CLAY[149]等	实现了内存容量的按需扩展	复杂，成本高	GPU 显存＋DRAM 无法存下模型
存储加速技术	GPU 直接访问存储	FlashNeuron[45]、Fastensor[150]、DeepPlan[151]、GDRec[152]等	GPU 直接访问 DRAM 和 SSD，缩短了数据访问路径	难以适应不同的硬件	GPU 显存无法满足模型存储需求
	近存储模型计算	Stannis[153]、OptimStore[154]、Smart-Infinity[155]、BeaconGNN[156]等	将部分计算操作卸载到存储设备或存储服务器中，降低了数据移动开销	难以适应不同的任务	参数量大＋计算密度低
模型压缩技术	模型参数压缩	Micikevicius 等人[157]、Deep Compression[158]等	降低了模型参数的存储占用	影响模型准确率	大模型训练或推理
	KV 缓存压缩	Scissorhands[159]、H_2O[160]、StreamingLLM[161]、FastGen[162]等	降低了 KV 缓存的存储占用	影响模型计算效率和准确率	大模型推理（LLM）
	激活数据压缩	Chen 等人[163]、Checkmate[164]、Gist[10]、cDMA[165]等	降低了激活数据的存储占用	影响模型计算效率和准确率	激活数据存储占用大

5.1.1　GPU 显存优化

(1)分布式显存优化

为了满足大模型训练和推理的存储需求,一些研究提出了模型并行方法,即将模型参数等模型状态分散存储在多块 GPU 的显存中。当前,模型并行主要分为三类:第一,流水线并行,即按层切分模型;第二,张量并行,即按层内的参数切分模型;第三,序列并行,即按输入序列切分模型。流水线并行和张量并行的流程如图 25 所示。此外,还有一些工作优化了数据并行方法,将原本冗余存储的模型状态分散存储到不同的 GPU 中。

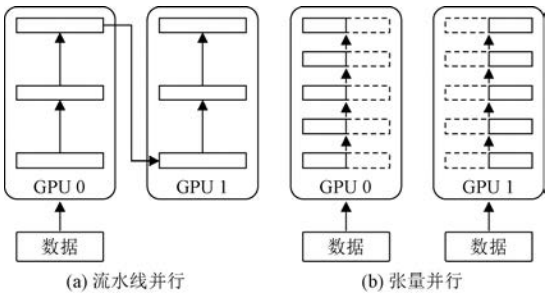

图 25　流水线并行与张量并行对比[166]

①流水线并行。GPipe[130] 提出将模型表示为层序列,并将其划分成多个部分,分散存储在不同的加速器中。在此基础上,GPipe 设计了批量分割流水线算法:在前向传播中,先将每个小批量划分成更小的微批量,然后流水线执行这些微批量;在反向传播中,计算每个微批量的梯度;在每个小批量结束时,汇总所有微批量的梯度,并更新所有加速器中的模型参数,具体如图 26 (a)所示。图中将 1 个小批量划分成 4 个微批量,方框中的数字表示微批量的 ID,而反向传播花费的时间为前向传播的 2 倍。在执行流水线时,GPipe 存在着较大的气泡(bubble)开销,即每个加速器会有一些时间处于空闲状态。

PipeDream[131] 提出了 1F1B(one-forward-one-backward)调度算法,具体流程如图 26 (b)所示(方框中的数字表示小批量的 ID)。在稳定状态中,每个加速器都将交替执行不同小批量的前向传播和反向传播。对于每个加速器,每次反向传播都会更新其存储的参数。因此,PipeDream 存储了不同版本的模型参数,确保某个小批量的前向传播和反向传播使用相同版本的参数,从而正确计算梯度。此外,PipeDream 还会自动将模型划分到不同的加速器中,以便平衡计算负载,最小化通信开销。

②张量并行。针对 Transformer 的网络结构,Megatron-LM[132] 提出使用张量并行来训练模型,

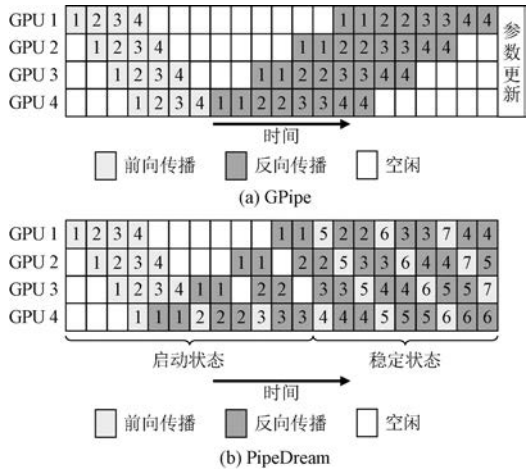

图 26　GPipe[130] 与 PipeDream[131] 对比

具体方式如图 27 所示。其中,$\boldsymbol{X}$ 和 $\boldsymbol{Z}$ 分别表示输入和输出;f 在前向传播中是恒等算子,在反向传播中则是全局归约(all-reduce);g 在前向传播中是全局归约,在反向传播中则是恒等算子;GeLU 是激活函数。

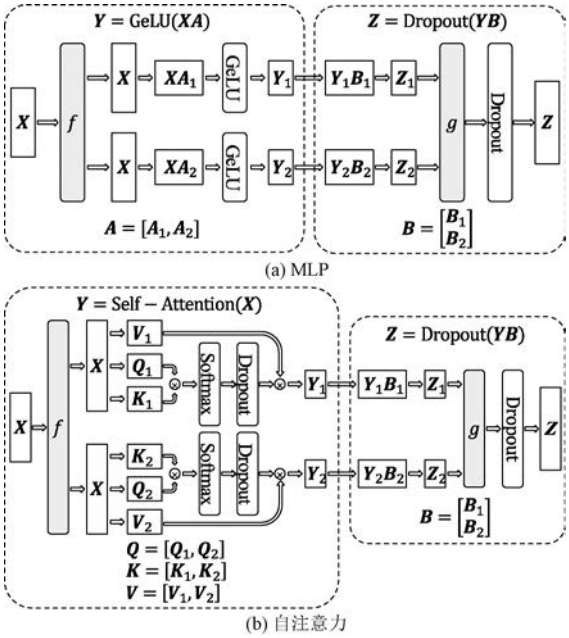

图 27　Megatron-LM 张量并行[132]

对于 MLP,Megatron-LM[132] 按列切分第一层的权重矩阵 $\boldsymbol{A}$,按行切分第二层的权重矩阵 $\boldsymbol{B}$;对于自注意力块,按列切分参数矩阵 $\boldsymbol{K}$、$\boldsymbol{Q}$ 以及 $\boldsymbol{V}$,然后按行切分线性输出层的参数矩阵;对于输入嵌入层和输出嵌入层,按列切分参数矩阵。此外,Megatron-LM 在每块 GPU 中维护了层归一化参数的副本,先对并行区域的输出结果执行 Dropout 和残差连接,再喂给下一个并行区域。

Megatron-LM 使用了 1 维矩阵切分策略,只会

按行或列来切分模型参数，并使用 all-reduce 来聚合计算结果[167]。每块 GPU 需要存储重复的输入张量和与结果大小相同的张量等，显存冗余大，通信效率低[167]。因此，一些研究[167-169]提出了 2D、2.5D 以及 3D 张量并行方法，使用高维矩阵切分策略来切分模型参数、激活数据以及梯度，提高了线性操作的并行程度，降低了每块 GPU 的显存需求，减少了通信开销。

③序列并行。张量并行方法只优化了 Transformer 中的注意力块和 MLP 块，没有切分 LayerNorm(层归一化)和注意力块与 MLP 块之后的 Dropout[170]。二者的计算量较小，但需要占用大量显存[170]。Korthikanti 等人[170]发现 LayerNorm 和 Dropout 操作在输入序列维度上是独立的，进而提出沿着输入序列切分这些区域。以 MLP 为例，其并行方式如图 28 所示。其中，字母的下标表示分散在哪个加速器中，上标则表示按哪个维度切分(字母 s、c、h 以及 r 分别表示按序列维度、列、隐含维度以及行切分)；$\boldsymbol{X} \in \mathbb{R}^{s\times b\times h}$ 是 LayerNorm 的输入，$\boldsymbol{A} \in \mathbb{R}^{h\times 4h}$ 和 $\boldsymbol{B} \in \mathbb{R}^{4h\times h}$ 是线性层的权重矩阵，这里的 s、b、h 分别表示序列长度、微批量以及隐含维度的大小；g 在前向传播中是全局收集(all-gather)，在反向传播中则是归约分散(reduce-scatter)；$\bar{g}$ 在前向传播中是归约分散，在反向传播中则是全局收集。从图中可以看出，除了第一个线性层的输入 $\boldsymbol{Y}$，其他数据都已经进行了切分。对于 $\boldsymbol{Y}$，Korthikanti 等人提出不存储完整的 $\boldsymbol{Y}$，只在第 i 个加速器中存储 $\boldsymbol{Y}_i^s$，并在反向传播中执行额外的全局收集，还重叠了这部分通信和 $\boldsymbol{Y}$ 的梯度计算。

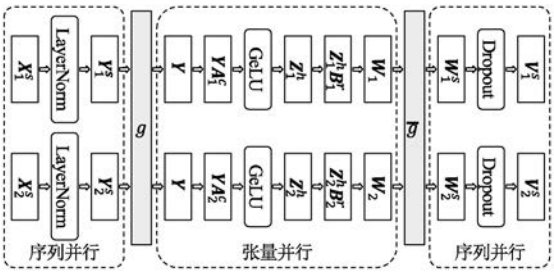

图 28 使用序列并行＋张量并行的 MLP[170]

④改进后的数据并行。当使用数据并行时，每块 GPU 都存储了完整的模型状态。然而，并非所有的模型状态都始终需要，例如，只有当某层执行前向传播和反向传播时，训练任务才会使用该层的模型参数[133]。因此，ZeRO-DP[133]提出了优化器状态分区 P_{os}、梯度分区 P_g 以及参数分区 P_p 三个优化阶段，分别将优化器状态、梯度以及模型参数分散存储到每块 GPU 中，具体如图 29 所示。图中使用了半精度浮点数(FP16)，Ψ 表示模型参数的数量，K 表示优化器状态占用的显存倍数，N_d 表示数据并行程度。在使用 P_{os} 之后，每块 GPU 只需要存储并更新 $1/N_d$ 的优化器状态，且只需要更新 $1/N_d$ 的参数。当每个训练步骤结束时，ZeRO-DP 将会执行全局收集操作，获取更新之后的完整参数。在使用 P_g 之后，ZeRO-DP 将会执行归约分散操作，将与特定分区相关的所有梯度进行桶化，并对整个桶执行归约操作，然后丢弃不再需要的其他梯度。在使用 P_p 之后，ZeRO-DP 将在前向传播和反向传播中执行广播操作，获取需要的其他参数分区，然后丢弃使用过的其他参数。

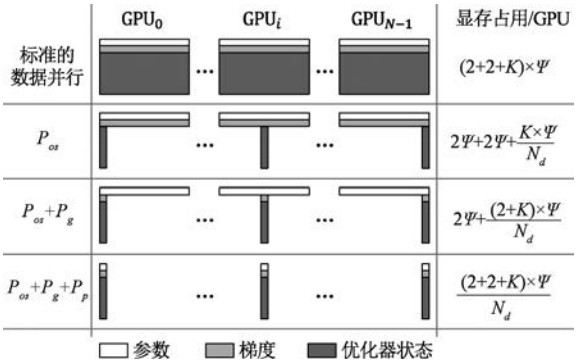

图 29 ZeRO-DP 的三个优化阶段[133]

案例分析：根据文献[130]的统计，对于图像分类任务，使用 GPipe 之后，谷歌在 ImageNet-1K 上成功训练了 5.57 亿参数的 AmoebaNet 模型，top-1 验证准确率达到了 84.4%；对于机器翻译任务，谷歌在超过 100 种语言的语料库上成功训练了 60 亿参数、128 层的多语言 Transformer 模型，翻译质量优于过去的双语言模型。

(2)KV 缓存显存优化

在训练场景中，LLM 输出序列中的所有词元(token)都是已知的，可以同时处理①。然而，在推理场景中，LLM 需要逐步生成输出序列中的每个 token。在生成当前 token 时，推理任务需要使用之前生成的所有 token 的键向量和值向量。因此，推理任务会缓存先前 token 的键向量和值向量，以便在生成当前 token 时复用这些向量，这种方法称为 KV(Key-Value)缓存[171]。

根据文献[134]的统计，当使用一块 NVIDIA

① Transformer，https://zh.d2l.ai/chapter_attention-mechanisms/transformer.html 2024,7,29。

A100 GPU 执行 13B(B 表示十亿)参数的 LLM 推理时,KV 缓存占用了接近 30%的显存容量。因此,如何高效存储推理场景中的大量 KV 缓存成为了热门的研究问题。现有的研究不仅优化了 GPU 的显存管理策略,提高了 KV 缓存的存储效率,而且使用了多种压缩技术,降低了 KV 缓存的显存占用。为了便于描述,本节只会介绍针对 KV 缓存的 GPU 显存优化方法,而与 KV 缓存压缩相关的工作将在 5.1.4 节中进行介绍。

vLLM[134]借鉴了操作系统的虚拟内存和分页技术,提出了 PagedAttention 算法,将每个请求的 KV 缓存划分成较小的块,每个块包含固定数量 token 的键向量和值向量,存储在不连续的虚拟显存中,缓解了内部的显存碎片,消除了外部的显存碎片,实现了在相同请求的不同序列之间和不同请求之间以块粒度共享 KV 缓存。PagedAttention 算法如图 30 所示。其中,键向量和值向量分散存储在三个不连续的块中(块 0、块 1 以及块 2),而每个块则可以存储 4 个 token 的键向量和值向量。当计算注意力时,查询 token“forth”先和某个块中的键向量相乘,例如,块 0 中“Four score and seven”的键向量,计算出注意力分数,再乘以该块中的值向量,即可得到最终的注意力输出。

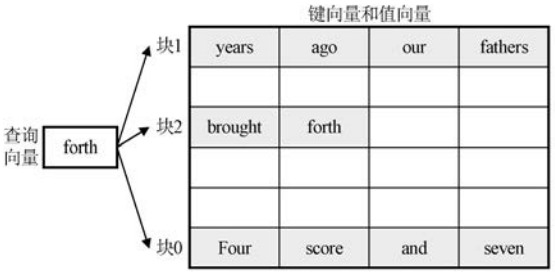

图 30　PagedAttention 算法示例[134]

然而,PagedAttention 将 KV 缓存存储在不连续的虚拟显存中,需要重写注意力核心函数,增加了软件的复杂度和冗余,引入了性能开销[135]。为了解决这些问题,vAttention[135]提出将 KV 缓存存储在连续的虚拟显存中,利用底层系统对按需分页的支持,按需分配物理显存,无缝复用现有的注意力核心函数,避免重新实现服务框架的显存管理,性能更优。

针对分布式场景,Infinite-LLM[136]提出了 DistAttention 算法,将注意力和 KV 缓存划分成更小的子块,实现了 KV 缓存的分布式存储和注意力的分布式计算,解耦了注意力计算和其他计算,并设计了贪心调度策略,可以在集群中调度 KV 缓存,支持更长的上下文长度,还实现了中心化的管理器,用于执行调度策略,管理 KV 缓存。

案例分析:根据文献[134]的统计,当使用 OPT(13B、66B 以及 175B)和 LLaMA-13B 模型来执行推理任务时,vLLM 的吞吐量比现有系统提高了 2 倍到 4 倍。

5.1.2　异构存储优化

当前,GPU 显存容量的增长速度远远低于模型参数规模的增长速度,简单地聚合多块 GPU 的显存难以支持模型规模的持续增长。为了解决该问题,一些研究[172]提出使用 DRAM、NVM 以及 SSD 等异构存储来增强 GPU 的存储容量,将部分模型状态卸载到这些存储设备中。

(1)基于 DRAM 的异构存储优化

2023 年 11 月 13 日,NVIDIA 推出了 H200 GPU,显存容量仅为 141GB①。然而,DRAM 的存储容量可以达到数百 GB,甚至数 TB,并且价格更加便宜。因此,一些研究提出使用 DRAM 来辅助存储模型状态。

针对训练场景,vDNN[137]提出释放 GPU 显存中不会复用的激活数据,并将需要复用但不会立即使用的激活数据卸载到 DRAM 中,后面再预取到 GPU 显存中,还重叠了正常计算和卸载/预取/释放,具体如图 31 所示。其中,$FWD_{(n)}$ 和 $BWD_{(n)}$ 分别表示第 n 层的前向传播和反向传播;$OFF_{(n)}$ 和 $PRE_{(n)}$ 分别表示卸载和预取第 n 层的激活数据。SuperNeurons[138]提出预分配大块 GPU 显存以实现高性能的存活性分析,设计了统一张量池,将卷积层的张量异步卸载到本地 CPU 内存中,并使用 GPU 显存来缓存张量,还提出了成本感知的重算策略,实现了卷积工作空间的动态分配。SwapAdvisor[139]提出使用自定义的遗传算法来搜索所有的显存分配和算子调度空间,在执行之前生成最优的交换计划,确定何时将哪些张量换入/换出 GPU 显存。Capuchin[140]提出根据运行时的张量访问模式来决定何时执行张量驱逐/预取或者重算,并根据运行时的反馈来迭代地修正显存管理策略。ZeRO-Offload[173]提出将梯度、优化器状态以及参数更新卸载 CPU 中,只在 GPU 中执行前向传播和反向传播,实现了高效的 CPU 优化器,设计了延迟一步的

① NVIDIA H200 Tensor Core GPU: The world's most powerful GPU for supercharging AI and HPC workloads, https://www.nvidia.com/en-us/data-center/h200/ 2024,1,10。

参数更新策略以重叠 CPU 和 GPU 的计算，还能够组合 ZeRO 数据并行或者模型并行以扩展到多块 GPU 上。Harmony[174] 提出将模型分解为细粒度任务，由任务调度器将计算和状态映射到多块 GPU 上，进而实现了环绕流水线并行（每块 GPU 计算的层不固定），还设计了输入批次分组、及时调度、泛化点对点交换以及多维度层打包。FAE[43] 利用输入数据不均匀导致嵌入表访问不均匀的特性，将热门和冷门的输入数据划分在不同的小批量中，并使用 GPU 来存储热门嵌入向量。Bagpipe[175] 使用缓存和预取来重叠远程嵌入表访问和模型计算，提出了用于更新缓存的预测算法，设计了逻辑上复制、物理上分区的分布式缓存，实现了支持低开销容错的分离式系统架构。Mobius[176] 提出使用流水线并行，将模型切分成多个阶段，在 GPU 显存和 DRAM 之间调度模型阶段，并使用混合线性规划来寻找最优的模型切分策略，还设计了模型阶段和 GPU 的交叉映射以最小化通信竞争。

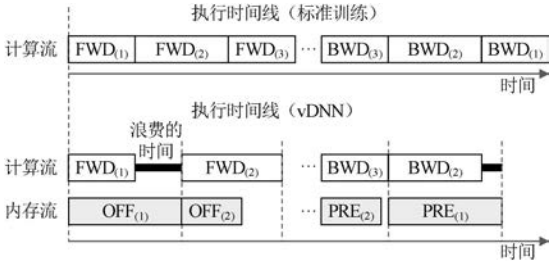

图 31 vDNN 卸载和预取示意图[137]

针对推理场景，Fleche[177] 使用 DRAM 来存储嵌入表，利用 GPU 显存来缓存热门的嵌入向量，设计了扁平缓存，可以让所有的嵌入表共享相同的缓存空间以提高整体的缓存利用率，实现了自识别的核心函数融合，能够将多个小的核心函数调用合并为一个调用以减少核心函数的维护开销，还优化了缓存查询的工作流，将 DRAM 的缓存索引和 GPU 的命中嵌入向量拷贝进行解耦，提出了统一索引技术以将 DRAM 中的部分索引卸载到 GPU 中。Pre-gated MoE[178] 提出了预门控函数（pre-gate function），在第 N 个 MoE（Mixture-of-Experts）块中选择第 $N+1$ 个 MoE 块中要被激活的专家，消除了 MoE 块中专家选择和专家执行之间的顺序依赖，还将专家参数存储在 CPU 内存中，只将激活的专家迁移到 GPU 中，并重叠了第 N 个 MoE 块的专家执行和第 $N+1$ 个 MoE 块的专家迁移。InfiniGen[179] 利用内存来存储 LLM 推理场景的 KV 缓存，使用当前层的注意力输入和下一层的部分查询权重和键缓存来推测下一层的注意力模式，并基于推测结果从内存中预取重要的 KV 缓存项，降低了 CPU 和 GPU 之间的 KV 缓存传输开销。PowerInfer[180] 利用了 LLM 推理中的高度局部性，离线生成神经元放置策略，将少量的热门神经元交给 GPU 来处理，将大量的冷门神经元交给 CPU 来处理，提出了自适应的在线预测器（预测神经元是否被激活），为激活稀疏性和倾斜更高的层构造更小的预测器，还设计了神经元感知的稀疏算子，直接与单个神经元进行交互。

针对 GNN 等嵌入模型的训练/推理，UGache[181] 提出了分解抽取机制，静态地使用不同 GPU 核心从不同源抽取嵌入项，还定义了衡量嵌入项访问频率的热度指标，并使用混合整数线性规划来建模多 GPU 上的抽取时间，提供了接近最优的嵌入项缓存策略。DistDGL[71] 不仅使用分布式内存键值存储 KVStore 来存储 GNN 的特征数据，还使用 KVStore 来存储并更新节点的嵌入向量。

表 18 总结了上述方法在卸载到 DRAM 的数据和适用模型方面的差异。

表 18 基于 DRAM 的异构存储优化对比

方法	卸载到 DRAM 的数据	适用模型
vDNN[137]	中间数据（激活数据）	CNN
SuperNeurons[138]	中间数据（卷积层的输出）	CNN
SwapAdvisor[139]	模型参数＋中间数据（激活数据）	大模型
Capuchin[140]	中间数据（激活数据）	大模型
ZeRO-Offload[173]	梯度＋优化器状态	LLM
Harmony[174]	模型参数＋中间数据＋优化器状态	大模型
FAE[43]	模型参数（嵌入表）	推荐模型
Bagpipe[175]	模型参数（嵌入表）	推荐模型
Mobius[176]	模型参数＋中间数据＋优化器状态	大模型
Fleche[177]	模型参数（嵌入表）	推荐模型
Pre-gated MoE[178]	模型参数（MoE 层）	LLM
InfiniGen[179]	中间数据（KV 缓存）	LLM
PowerInfer[180]	模型参数（冷门神经元）	LLM
UGache[181]	模型参数（嵌入表）	嵌入模型
DistDGL[71]	模型参数（嵌入向量）	GNN

案例分析：根据文献[180] 的统计，当使用 1 块 NVIDIA RTX 4090 GPU 来执行 Falcon-40B 和 LLaMA-70B 等大模型的推理时，PowerInfer 平均每秒可以生成 13.20（采用 INT4 格式）和 8.32（采用 FP16 格式）个 token，比 llama.cpp 最高提升了 4.28 倍和 11.69 倍。

（2）基于 NVM 的异构存储优化

在使用 DRAM 时，训练/推理进程可能会与其他应用竞争 CPU 的内存带宽和存储容量，从而导致性能下降。与 DRAM 相比，NVM 同样支持字节

寻址,并且具有成本低、容量大、性能相近以及支持持久存储等优势。因此,一些研究人员也提出使用NVM来存储模型状态。

针对训练场景,参数服务器系统OpenEmbedding[141]利用持久性内存(Persistent Memory,PM)来存储推荐模型的稀疏特征(嵌入表),使用DRAM来缓存热门的嵌入向量,设计了流水线缓存管理,流水线执行模型训练和缓存替换,隐藏了PM的访问延迟和缓存替换开销,还协同设计了缓存替换和检查点,提出了轻量级的同步检查点机制,降低了检查点的运行开销。

针对推理场景,参数服务器系统PetPS[142]利用PM来存储嵌入模型的参数。为了最小化PM的访问延迟,PetPS设计了针对嵌入模型的哈希索引,使用了单级哈希表、热度感知的参数放置策略以及预取机制。为了缓解CPU的负担,PetPS提出将参数收集任务卸载到网卡中,使用了写时复制以确保没有原地更新,并设计了基于时期列表的空间回收机制,能够保护待收集的参数不被修改。

案例分析:根据文献[141]的统计,在训练1个500GB的DLRM时,与其他缓存系统相比,OpenEmbedding的训练时间最高缩短了53.8%;而与纯DRAM方法相比,OpenEmbedding的存储成本最高降低了42%。

(3)基于SSD的异构存储优化

与DRAM和NVM相比,SSD的读写带宽低,访问延迟高,并且存在着读写放大的问题。但是,SSD具有购买价格更低、存储容量更大等优势。因此,一些研究也提出使用SSD来存储模型状态。

针对训练场景,FlashNeuron[45]设计了卸载调度器,选择性地将压缩之后(使用增强的压缩稀疏行格式+FP16转换)的中间数据卸载到NVMe SSD中,还设计了内存管理器,负责张量的分配与回收、卸载与预取以及压缩与解压缩。FlashNeuron的卸载策略如图32所示(假设GPU的显存容量只有8MB,实线表示前向传播,方框的颜色越深表示张量的压缩比越高)。在阶段1中,卸载调度器将从前到后不断选择需要卸载的张量,直到GPU显存不再溢出。如果张量卸载和前向传播可以重叠执行,则采用该方案;否则,执行阶段2。在阶段2中,卸载调度器首先从阶段1选出的张量中淘汰最后一个不可压缩的张量,并从未被选择的张量中挑选一些压缩比最高的张量,直到新选出的张量大小之和超过淘汰的张量大小。如果张量卸载和前向传播可以重叠执行,则采用该方案;否则,重复执行阶段2,直到找到合适的调度方案或者没有可压缩的张量为止。

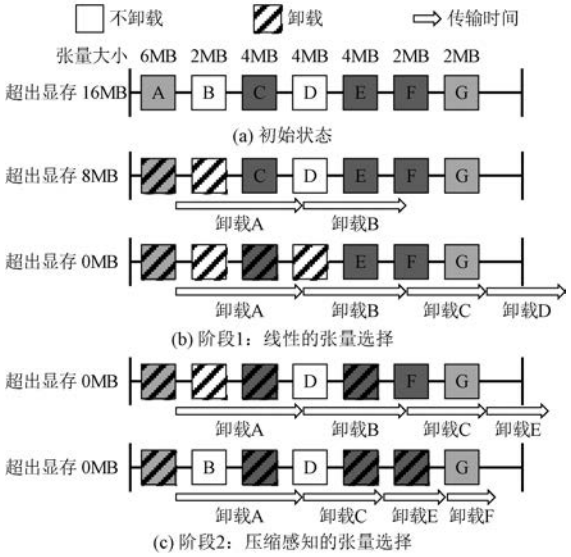

图32　FlashNeuron的卸载策略[45]

Behemoth[143]使用闪存来替代加速器使用的高带宽内存,提出了以闪存为中心的训练加速器,优化了现有的SSD设计,提高了SSD的带宽和耐久度。ZeRO-Infinity[144]利用主存或者NVMe SSD来存储模型状态,并在GPU显存、主存以及NVMe SSD之间按需移动它们。StrongHold[145]提出将模型状态动态卸载到内存和NVMe SSD中,并使用分析模型自动确定GPU需要保留的层数,实现了并行参数更新和异构集合通信,使用了用户态显存管理机制,减少了跨节点通信,还在单块GPU上实现了数据并行。

针对推理场景,Bandana[182]提出使用NVMe SSD来存储模型,同时利用少量DRAM作为缓存,使用超图划分方法将可能会被一起读取的嵌入向量存储在相同的块中,并模拟运行多个小型缓存,以确定需要在DRAM中缓存的嵌入向量数量。EVStore[183]使用DRAM来缓存SSD中的嵌入向量,并利用嵌入向量查询的结构规则和领域特定的近似方法,设计了由按组缓存、混合精度缓存以及近似缓存组成的三层嵌入向量缓存。Leviathan[184]利用闪存来替换高带宽内存,支持在单个节点中存储TB级别的权重,能够满足模型推理的带宽需求,同时避免了闪存读干扰导致的吞吐量下降。CachedAttention[185]提出在相同会话的多轮对话之间复用KV缓存,设计了基于SSD、内存以及GPU显存的分层KV缓存系统,使用了逐层预加载和异步保存机制来重叠KV缓存访问和GPU计算,利用了调度器

感知的加载和驱逐机制来放置 KV 缓存(预取到内存中、驱逐到 SSD 中或者丢弃),还实现了位置编码解耦的 KV 缓存截断机制,避免了上下文窗口溢出后 KV 缓存的失效。LLM in a flash[186]提出将注意力机制的参数存储在 DRAM 中,而将前馈网络的参数存储在 SSD 中,并按需传输到 DRAM 中,使用了滑动窗口技术,复用以前激活的神经元,减少了数据传输,提出了行列捆绑方法,将前一层的列和后一层的行存储在一起,提高了传输吞吐量,还优化了 DRAM 中的参数管理。

针对智能手机上的 LLM 推理,PowerInfer-2[187]扩展了 PowerInfer[180]:首先,PowerInfer-2 提出将粗粒度的矩阵计算分解为细粒度的神经元簇计算,以便利用手机上的异构计算、内存以及 I/O 资源,设计了多态神经元引擎,为 LLM 推理的预填充和解码阶段使用不同的计算策略;其次,PowerInfer-2 引入了分段神经元缓存,为不同的参数类型使用不同的缓存策略;然后,PowerInfer-2 设计了灵活的神经元加载机制;接着,PowerInfer-2 提出了细粒度的神经元簇级流水线技术,重叠了 I/O 操作和神经元簇计算;最后,PowerInfer-2 还提出在首次推理之前执行离线规划器,生成执行计划,以便支持不同的 LLM 和智能手机。

表 19 总结了上述方法在卸载到 SSD 的数据和适用模型方面的差异。

表 19 基于 SSD 的异构存储优化对比

方法	卸载到 SSD 的数据	适用模型
FlashNeuron[45]	中间数据	通用
Behemoth[143]	模型参数+激活数据	LLM
ZeRO-Infinity[144]	模型参数+梯度+优化器状态	LLM
StrongHold[145]	模型参数+中间数据+优化器状态	LLM
Bandana[182]	模型参数(嵌入表)	推荐模型
EVStore[183]	模型参数(嵌入表)	推荐模型
Leviathan[184]	模型参数	LLM
CachedAttention[185]	中间数据(KV 缓存)	LLM
LLM in a flash[186]	模型参数(前馈网络)	LLM
PowerInfer-2[187]	模型参数	LLM

案例分析:根据文献[144]的统计,在 32 个 NVIDIA V100 DGX-2 节点(总共 512 块 GPU)上,ZeRO-Infinity 可以训练具有 32 万亿参数的大模型,模型规模比 3D 并行扩大了 50 倍,实现了超过 25 PFLOPS 的训练吞吐量。

(4)基于 CXL 内存的异构存储优化

随着 CXL 协议的兴起,一些研究也提出使用 CXL 内存来存储模型状态,以便实现内存容量的按需扩展。

针对训练场景,COARSE[146]利用了缓存一致性互联(Cache-Coherent Interconnection,CCI)协议,构建了分离式内存系统。首先,COARSE 提出了去中心化的参数通信方案,实现了参数同步的去中心化和参数存储的局部化。其次,COARSE 提出了张量路由和分区机制,利用了不均匀的参数大小和带宽分布,使用了串行总线接口提供的双向带宽。最后,COARSE 提出了基于优先级的双重同步机制以利用张量的局部性,设计了基于队列的同步机制以避免死锁。

ReCXL[147]提出将推荐模型中内存密集的嵌入层存储在 CXL 内存中,并实现了统一的近内存处理(Near-Memory Processing,NMP)架构,在 CXL 内存中处理整个嵌入层的训练(如图 33 所示),还设计了无依赖的预取和细粒度的更新调度方法,在 ReCXL 设备空闲时预取并调度后续批次的输入。

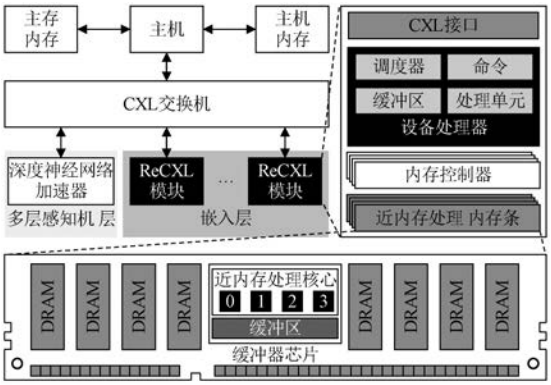

图 33 ReCXL 架构[147]

针对推理场景,Park 等人[148]提出了基于 CXL 的近内存处理平台 CXL-PNM。首先,Park 等人提出了基于 LPDDR5X(Low Power Double Data Rate 5X)的 CXL 内存架构,具有 512GB 的存储容量和 1.1TB/s 的带宽。其次,Park 等人设计了集成 LLM 推理加速器的 CXL-PNM 控制器架构。最后,Park 等人还实现了 CXL-PNM 软件栈,包括面向 CXL-PNM 的 Python 库和设备驱动程序,允许 Python 程序无缝、透明地使用 CXL-PNM。

Yun 等人[149]提出了基于 CXL 的近数据处理架构 CLAY,可以加快 DNN 中的嵌入层。具体来说,CLAY 摆脱了传统内存系统的多点总线,降低了 DRAM 模块之间的数据传输开销,设计了专门的内存地址映射方法,消除了处理单元之间的负载不均衡。此外,Yun 等人还提出了多 CLAY 系统,与其他处理器协作执行端到端推理,实现了包复制,

缓解了主机和 DRAM 集群之间的带宽负担,设计了使用 CLAY 的软件栈。

表 20 总结了上述方法在是否支持近内存处理和适用模型方面的差异。

表 20　基于 CXL 内存的异构存储优化对比

方法	是否支持近内存处理	适用模型
COARSE[146]	否	通用
ReCXL[147]	是	推荐模型
CXL-PNM[148]	是	LLM
CLAY[149]	是	GNN 和推荐模型等

案例分析:根据文献[147]的统计,与 CPU+GPU 方法和简单的 CXL 内存池相比,ReCXL 的推荐模型训练性能平均提高了 9.4 倍和 22.6 倍。

5.1.3　存储加速技术

在满足模型状态的存储需求之后,一些研究还提出了其他存储加速技术。根据使用的加速技术不同,本文将其分为以下两类:第一,GPU 直接访问存储,即让 GPU 直接访问存储在 DRAM 和 SSD 等设备中的模型状态,缩短数据访问路径,避免额外的拷贝;第二,近存储模型计算,即使用近存储/存储内处理来加速模型计算,将部分或者全部模型计算操作(包含预处理操作)卸载到计算型 SSD 或者存储服务器等中。

(1)GPU 直接访问存储

针对训练场景,FlashNeuron[45]利用了 GPUDirect 技术,提出了轻量级的用户态 I/O 栈 P2P-DSA(Peer-to-Peer Direct Storage Access),能够在 GPU 和 NVMe SSD 之间直接传输张量数据,具体流程如图 34 所示。Fastensor[150]使用了 GDS 技术,优化了 NVMe SSD 和 GPU 显存之间的张量数据传输(模型参数保存和激活数据读写等),提出了数据和

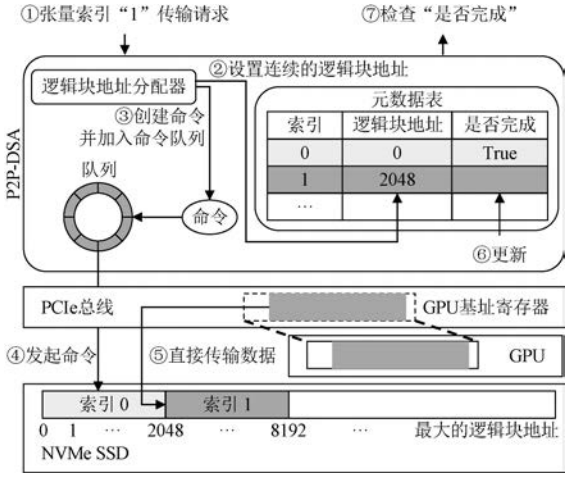

图 34　P2P-DSA 写流程示例[45]

运行时上下文感知的张量 I/O 算法,能够为当前张量自动选择最优的数据传输机制,并实现了统一的读写接口以支持各种常见的数据传输 API,例如,torch.save()。

针对推理场景,DeepPlan[151]利用了 GPU 提供的内存直访机制,直接访问内存中的层,解决了流水线加载中存在的挂起问题,并设计了并行传输方法,使用多块 GPU 并行加载模型,再转发给一块 GPU,还提出在部署之前自动选择每层的执行方法(加载或者内存直访)。GDRec[152]提出由 GPU 直接访问 SSD 和 DRAM 中的嵌入向量,利用 UVA 特性来实现内存直访机制,提出了访问合并和访问对齐来优化内存访问性能,设计了用户态 NVMe 驱动程序来实现外存直访机制。表 21 总结了上述方法在是否支持 GPU 直接访问内外存和适用模型等方面的差异。

表 21　GPU 直接访问存储方法对比(模型状态)

方法	直访外存	直访内存	适用模型
FlashNeuron[45]	是	否	通用
Fastensor[150]	是	否	通用
DeepPlan[151]	否	是	LLM
GDRec[152]	是	是	推荐模型

(2)近存储模型计算

针对训练场景,Stannis[153]设计了处理能力强、功耗低的计算型 SSD,提出使用 SSD 集群来执行分布式训练,将部分训练任务卸载到 SSD 中,实现了系统利用率调优算法,能够为 SSD 和 CPU 分别选择最合适的批大小,并将私有数据分配给 SSD 处理,将公开数据分配给 SSD 和 CPU 处理,避免了私有数据传输,提供了隐私保护。OptimStore[154]和 Smart-Infinity[155]提出将参数更新操作卸载到 SSD 中,避免了优化器状态的移动开销。BeaconGNN[156]提出将 GNN 训练卸载到 SSD 中,设计了新的图数据格式以支持无序邻域采样,部署了跨越控制器、通道以及晶粒(Die)的近数据处理引擎,使用了晶粒级采样器来执行邻域采样和特征向量检索,使用了通道级命令路由器以提高后端 I/O 的处理效率,还将空间加速器集成到 SSD 内部总线上以加快 GNN 的嵌入向量聚合和更新。FlashGNN[188]提出直接在 SSD 控制器中执行 GNN 训练,设计了节点级 GNN 训练方法、闪存块请求调度算法以及数据驱动的子图生成算法。针对推荐模型,NDRec[189]提出将嵌入层的存储和计算卸载到计算型 SSD 中,利用 CXL 协议来实现 GPU 和 SSD 之间的通信,还设计了嵌入输出预先计算方法,能够并发执行嵌入层和其他

层,提出了软件管理的嵌入向量缓存策略(应用在计算型 SSD 的 DRAM 中),设计了用于处理嵌入表的 FPGA(Field Programmable Gate Array)计算单元。

针对推理场景,GLIST[190]提出直接在 SSD 中处理推理请求,降低了数据移动开销,同时提供了相关 API,允许开发者便捷地部署图学习服务。HolisticGNN[191]将 GNN 推理卸载到计算型 SSD 中,提供了一系列软硬件设施,允许用户实现各种 GNN 算法,并直接在存储中执行推理,降低了推理延迟,减少了能源消耗。RecSSD[192]使用 SSD 来存储推荐系统的嵌入表,并将所有的嵌入表操作卸载到 SSD 中,同时利用主机端和 SSD 端的缓存,减少了端到端的模型推理延迟。RM-SSD[193]提出将整个推荐系统卸载到 SSD 中,使用两阶段细粒度读策略消除了嵌入表查询的读放大,使用流水线将推荐模型的拓扑结构重新映射到 FPGA 中来加速 MLP 层,并提供了推荐系统语义感知的接口。DeepStore[194]提出将智能查询卸载到 SSD 中,开发了高效节能的存储内加速器,设计了基于相似度的存储内查询缓存以利用用户查询的时间局部性,实现了轻量级的存储内运行时系统以支持不同类型的智能查询。

NDPipe[195]提出为存储服务器配备消费级 GPU,在照片存储服务器中执行微调(训练)和离线推理。具体来说,NDPipe 设计了基于微调的数据和模型并行策略,将模型划分成两个部分,在存储服务器中计算参数冻结的层,在训练服务器中训练需要更新的层,开发了自动化模型分区和组织工具,可以找出最合适的模型划分点,以实现训练服务器和存储服务器的负载均衡,还优化了近数据处理引擎,解决了数据加载、数据预处理以及模型计算的瓶颈。

表 22 总结了上述方法在卸载的计算操作和适用模型方面的差异。

表 22 近存储模型计算方法对比

方法	卸载的计算操作	适用模型
Stannis[153]	部分数据的计算	通用
OptimStore[154]	参数更新	LLM
Smart-Infinity[155]	参数更新	LLM
BeaconGNN[156]	全部计算	GNN
FlashGNN[188]	全部计算	GNN
NDRec[189]	嵌入表操作	推荐模型
GLIST[190]	全部计算	GNN
HolisticGNN[191]	全部计算	GNN
RecSSD[192]	嵌入表操作	推荐模型
RM-SSD[193]	全部计算	推荐模型
DeepStore[194]	全部计算	智能查询模型
NDPipe[195]	部分训练+全部推理	CNN

案例分析:根据文献[190]的统计,与 DGL 框架在 CPU 和 GPU 上的性能相比,GLIST 的推理性能平均提高了 13.2 倍和 10.1 倍,能耗最高降低了 98.7%和 98.0%。

5.1.4 模型压缩技术

为了满足模型训练/推理的数据存储需求,现有的研究提出了许多模型压缩技术,减少了模型参数、KV 缓存以及激活数据等模型状态的存储占用。

(1)模型参数压缩

一些研究[196]提出使用参数量化和参数剪枝等方法来压缩模型参数,以降低训练/推理场景的模型存储需求。参数量化使用更低的数值精度来代替常用的单精度浮点数(FP32)。而参数剪枝则会根据特定的评价标准,删除冗余的参数,保留重要的参数,并重新训练网络,微调剩余的参数,以便恢复原有的准确率。

针对训练场景,Micikevicius 等人[157]提出了同时使用 FP16 和 FP32 的混合精度训练方法,并设计了 3 种技术来避免模型准确率下降。技术 1 会备份 FP32 格式的参数,在前向传播和反向传播中使用 FP16 格式的参数(激活数据和梯度也存储为 FP16 格式),而在参数更新时则使用 FP32 格式的参数。技术 2 会在反向传播之前放大前向传播计算出的损失值,以保留低于 FP16 可表示范围的小幅度梯度值,并在反向传播之后、但在梯度裁剪或者其他与梯度相关的计算之前,缩小参数的梯度,以维持与 FP32 训练相同的参数更新幅度。技术 3 会将 FP16 格式的运算累积成 FP32 格式的输出,并在存储之前转换成 FP16 格式。

针对推理场景,Han 等人[158]提出了组合剪枝、量化以及哈夫曼编码的深度压缩(Deep Compression)方法,具体流程如图 35 所示。首先,执行参数剪枝,删除冗余的连接,只保留最重要的连接;然后,执行参数量化,使用更少的比特数来表示每个参数,以便多个连接能够共享相同的参数;最后,执行哈夫

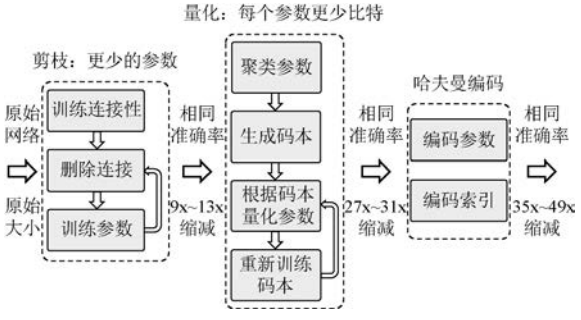

图 35 Deep Compression 流程[158]

曼编码,充分利用有效参数的有偏分布。在前两个步骤结束之后,深度压缩方法会重新训练模型,微调剩余的连接和量化之后的质心。

案例分析:根据文献[158]的统计,当在 ImageNet-1K 上压缩 VGG16 时,Deep Compression 可以将 VGG16 从 552MB 压缩至 11.3MB,模型大小降低了 98%。

(2)KV 缓存压缩

现有的研究提出了许多面向大模型推理的 KV 缓存压缩方法,降低了 KV 缓存的显存占用。这些方法主要分为两类:第一,KV 缓存驱逐,即保留重要 token 的 KV 缓存,驱逐/丢弃不重要 token 的 KV 缓存;第二,KV 缓存量化,即使用量化方法,降低 KV 缓存的数值精度。

①KV 缓存驱逐。InfiniGen[179]使用了基于计数器的缓存驱逐策略,动态删除不常用 token 的 KV 缓存。Scissorhands[159]提出使用注意力分数来衡量 token 的重要性,在缓存空间用完之后驱逐不重要 token 的 KV 缓存。H_2O[160]提出驱逐累积注意力分数最低的 KV 缓存。StreamingLLM[161]发现了注意力下沉(Attention Sink)现象,即大部分的注意力分数都在初始 token 中,进而提出缓存初始 token 的 KV 和最近 token 的 KV(如图 36 所示),还提出在预训练时添加一个占位符 token,作为专门的注意力下沉 token。

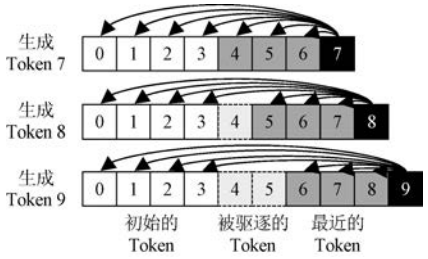

图 36　StreamingLLM 的 KV 缓存驱逐策略[161]

一些研究也提出为不同的注意力头或者层使用不同的驱逐策略。FastGen[162]提出了自适应的 KV 缓存压缩方法,对不同类型的注意力头使用不同的驱逐策略。SnapKV[197]提出为每个注意力头选择最重要的 KV 位置。Wu 等人[198]提出只计算和缓存少量层的 KV。MiniCache[199]提出沿着 LLM 的深度维度,利用层间 KV 缓存的相似性,跨层合并 KV 缓存。PyramidKV[200]提出动态调整不同层的 KV 缓存大小,为更低的层分配更多的缓存,为更高的层分配更少的缓存。D_2O[201]提出根据注意力权重的密度来改变每层的驱逐比例,并根据被丢弃 token 和保留 token 的相似程度来决定是否将被丢弃 token 和相似 token 合并。

②KV 缓存量化。KVQuant[202]组合使用了逐通道键量化、在执行旋转位置编码之前量化键、不均匀的键值缓存量化、逐向量的稠密和稀疏量化以及归一化量化质心方法。KIVI[203]提出按通道量化键缓存,按 token 量化值缓存。WKVQuant[204]提出同时量化参数和 KV 缓存,只量化过去的 KV 缓存,组合当前未量化的 KV 和之前量化过的 KV 缓存来改进注意力的计算,设计了同时考虑通道和 token 的二维量化策略,还引入了跨块重构正则化以降低量化误差。MiKV[205]提出以低精度保留不重要的 KV 对,以高精度存储重要的 KV 对。QAQ[206]提出对键缓存和值缓存使用不同的量化策略,设计了注意力窗口方法以预测注意力分数,并且不量化异常值。GEAR[207]提出先量化大部分相似量级的 KV 缓存,再使用一个低秩矩阵来近似量化误差,最后使用一个稀疏矩阵来弥补异常值引起的误差。

③混合方法。FlexGen[208]设计了基于 DRAM 和磁盘的模型状态卸载策略,并使用分组量化,对参数按输出通道维度分组,对 KV 缓存按隐藏维度分组,将参数和 KV 缓存压缩成 4 比特整数,还使用了稀疏注意力方法,为每个查询计算前 K 个重要的 token,丢弃其他 token。ALISA[209]提出了稀疏窗口注意力算法,保留最近的 token 和最重要的 token,设计了三阶段调度器,以 token 为粒度在 GPU 和 CPU 之间动态调度 KV 张量,平衡了缓存和重算,还使用了逐通道量化技术,将 KV 缓存压缩成 8 比特整数。

案例分析:根据文献[161]的统计,对于 Llama-2、MPT、Falcon 以及 Pythia 等模型,StreamingLLM 允许其处理超过 4 百万个 token;而与现有的 KV 缓存压缩方法相比,StreamingLLM 的 token 解码延迟最高降低了 95.5%。

(3)激活数据压缩

在前向传播过程中,各层的激活函数将会产生大量的激活数据。这些数据会被保存在显存中,以便在反向传播中用于计算梯度。为了降低激活数据的显存占用,现有的研究提出了许多激活数据压缩方法。这些方法可以分为两类:第一,激活数据检查点,即丢弃部分激活数据,并在反向传播过程中重新计算这些激活数据;第二,其他激活数据压缩方法,即使用无损/有损压缩来压缩激活数据。表 23 总结了现有方法在特点和适用模型方面的差异。

表 23 激活数据压缩方法对比

方法	特点	适用模型
Chen 等人[163]	丢弃,然后重算	线性网络
Checkmate[164]	丢弃,然后重算	非线性网络
Korthikanti 等人[170]	选择性丢弃,然后重算	Transformer
Gist[10]	无损压缩+有损压缩	CNN
cDMA[165]	无损压缩	CNN
JPEG-ACT[210]	有损压缩	CNN
COMET[211]	有损压缩	CNN
ActNN[212]	混合精度量化	通用
EXACT[213]	随机投影+量化	GNN
Kurtz 等人[214]	稀疏化+CSR	CNN
SNICIT[215]	量化(聚类)	CNN

①激活数据检查点。Chen 等人[163]提出将线性神经网络划分成几个片段,只存储每个片段的输出结果,丢弃每个片段内部的中间结果,并在此基础上设计了 3 种激活数据检查点方法。方法 1 会丢弃批归一化和池化等低开销操作的输出结果,保留卷积等计算耗时操作的输出结果。方法 2 实现了贪心的内存分配,能够平衡每个片段内部的计算开销和输出结果的存储开销。方法 3 将每个片段看作一个组合了片段内所有算子的大算子,通过递归的方式来优化每个片段内的子图。

针对非线性网络,Checkmate[164]提出使用混合整数线性规划来寻找最优的激活数据检查点策略,并利用现有的求解器来求解该问题。然而,在最坏情况下,对于具有数百层的神经网络结构,整数线性规划无法求解此类问题。因此,Checkmate 还使用了线性规划,提出了基于两阶段舍入策略的近似算法,能够寻找近似最优解。

针对 Transformer,Korthikanti 等人[170]发现不同激活数据的重算成本不同,进而提出了选择性激活重算方法,只丢弃并重算每个 Transformer 层中显存占用较大但重算成本较低的部分激活数据,即与注意力操作相关的激活数据,包括 $\boldsymbol{QK}^T$ 矩阵乘法、Softmax、Softmax 之后的 Dropout 以及使用 $\boldsymbol{V}$ 计算注意力输出。

②其他压缩方法。针对训练场景,Gist[10]为池化层前面的 ReLU 层设计了二值化方法,使用 1 个比特来存储 ReLU 的输出,为卷积层前面的 ReLU 层设计了稀疏存储、稠密计算方法,还提出了延迟精度降低方法,当前向传播不再使用激活数据时才执行精度缩减。cDMA[165]提出在激活数据从显存传输到 DRAM 之前,使用无损压缩方法(运行长度编码、零值压缩、Zlib 压缩)来减少数据的大小。JPEG-ACT[210]设计了缩放定点精度降低方法,可以将 FP32 格式的激活数据转换成 8 位整数(JPEG 算法只能处理整数),并优化了 JPEG 算法以压缩 CNN 训练中的激活数据。COMET[211]使用了误差有界的有损压缩方法来动态压缩激活数据,设计了自适应的误差有界控制机制,提出了改进的 SZ(Squeeze)误差有界有损压缩方法来压缩激活数据中的连续零值,降低了 CNN 训练的内存需求,具体如图 37 所示。ActNN[212]设计了分组量化策略以处理特征维度间的不同数值范围,提出了细粒度的混合精度策略,自适应地选择每个样本和每个层的激活数据精度,并在线调整混合精度量化策略。EXACT[213]提出将随机投影和量化串行应用于 GNN 训练中的激活数据。

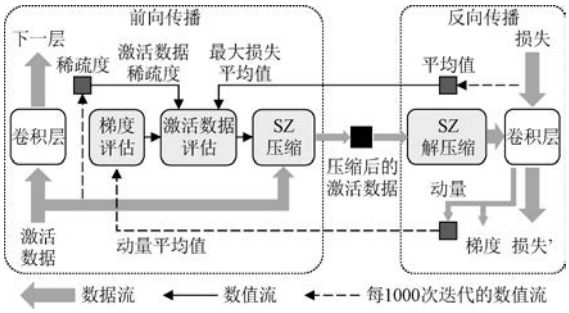

图 37 COMET 框架[211]

针对推理场景,Kurtz 等人[214]提出了带参数的激活函数 FATReLU(如果输入 x 小于可变阈值则输出 0,否则输出 x),在此基础上设计了基于阈值的稀疏化方法,组合使用 Hoyer 正则化,还使用了压缩稀疏行(Compressed Sparse Row,CSR)的变体来压缩激活数据,设计了针对稀疏输入的快速卷积算法。SNICIT[215]利用稀疏模型中激活数据收敛之后的高度相似性,提出在推理时使用数据聚类将收敛之后的激活数据转换成更稀疏的表示。

案例分析:根据文献[211]的统计,当在 ImageNet-1K 上训练 AlexNet、VGG16、ResNet18 以及 ResNet50 时,与标准训练方法和 JPEG-ACT 相比,COMET 的显存占用最高降低了 92.6%和 45.5%,但准确率却几乎没有损失。

5.2 模型训练容错技术

在训练过程中,训练任务可能会遇到程序崩溃、突然断电以及节点故障等情况,导致训练中断。根据 Jeon 等人[216]的统计,在微软的集群中,深度学习训练任务平均运行 45 分钟就会发生故障(不包括早期故障)。类似地,Zhang 等人[217]也发现,在训练拥有 1750 亿参数的 OPT(Open Pre-trained Transformers)模型时,由于大量的硬件故障,训练任务在

两个月内重启次数超过 100 次。

当前,训练任务通常会使用检查点方法作为容错机制,定期保存模型状态[218]。在发生故障之后,训练任务只需要读取最新的检查点,并重新执行部分训练,就能恢复中断时的训练状态。然而,为了确保模型参数的一致性,深度学习框架大都采用了同步检查点方法,即在执行检查点操作时,训练任务必须暂停。为了降低训练容错和故障恢复开销,现有的研究优化了检查点方法,引入了多种容错技术。表 24 总结了不同模型训练容错技术的差异。

表 24　模型训练容错技术对比

分类	方法	特点	优化方面	影响模型准确率	适宜的模型规模	适用场景
模型状态检查点	异步检查点	异步执行检查点操作	容错	否	任意	通用
	增量检查点	增量存储多个检查点	存储空间占用	否	任意	通用
	压缩检查点	使用量化等方法压缩检查点	存储空间占用	是	任意	通用
	及时检查点	发生故障后才执行检查点操作	容错	否	任意	分布式训练
	部分恢复	只在故障节点加载检查点	故障恢复	是	任意	分布式训练
其他容错技术	多副本	存储模型状态的多个副本	故障恢复	否	小	参数规模较小
	纠删码	存储模型状态的校验码	故障恢复	否	大	参数更新不频繁
	日志	将中间数据写到日志文件中	故障恢复	否	中	写入能跟上计算
	冗余计算	冗余执行后继节点的计算	故障恢复	否	任意	故障频繁发生

5.2.1　模型状态检查点

(1)异步检查点

一些研究提出异步执行检查点操作,重叠检查点和模型计算,降低了检查点的开销。针对单节点训练,CheckFreq[219]使用在线分析来自动确定迭代粒度的检查点频率,提出了自适应的速率调整策略以动态调整运行时的检查点频率,设计了可恢复的数据迭代器以保证训练数据的不变性(即在每轮训练中,以随机的顺序将数据集中的所有样本刚好处理一次),并使用两阶段检查点机制来流水线执行计算和检查点,具体如图 38 所示。针对分布式训练,Gemini[220]使用由本地 CPU 内存、远程 CPU 内存以及远程持久存储组成的分层存储来保存检查点,依靠 CPU 内存中的检查点来实现快速的故障恢复,设计了接近最优的检查点放置策略,提高了从 CPU 内存中恢复故障的概率,还提出了检查点流量调度算法,降低了检查点流量对模型训练的影响。LightCheck[221]提出了异步分层检查点,逐层流水线执行检查点和计算与通信,并结合了 PM 的直接访问特性和 UVA 技术,允许 GPU 直接访问 PM,重叠了数据传输和 GPU 计算,还分开存储张量的元数据和数据,提高了 PM 的写吞吐量。DataStates-LLM[222]提出将 GPU 中的模型状态分片的副本合并到主存中,设计了延迟非阻塞复制技术,重叠了前向/反向传播和分片复制,并以流式处理模式将其刷新到持久存储中,还提出异步合并所有 GPU 的分片。Portus[223]提出了高效的三级索引结构和点对点的直接数据路径,无需专门的序列化过程,能够在 GPU 和 PM 之间执行零拷贝传输,还解耦了检查点和模型训练,异步执行检查点操作。FastPersist[224]利用 NVMe SSD 来优化模型检查点,加快了单个节点的检查点写入,实现了节点之间的并行写入,还重叠了检查点写入和下一次迭代的模型计算。

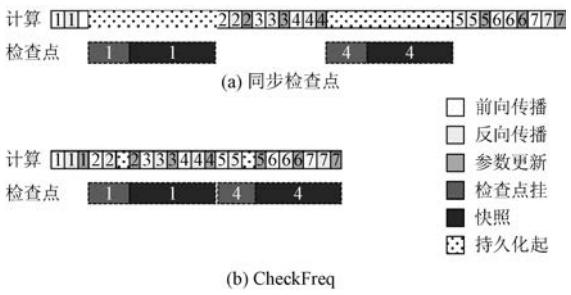

图 38　同步检查点与异步检查点对比[219]

(2)增量检查点

一些研究提出使用增量检查点,减少了每次写入的数据量。每个检查点只需要存储前一个检查点保存之后的修改部分[225]。在训练中断之后,训练任务只需要对初始检查点不断执行修改,就能恢复最终的模型状态[225]。

DNN 等模型的参数更新矩阵可以由更小的充分向量(Sufficient Vectors,SVs)计算得到[226]。因此,Orpheus[226]设计了 ISVC(Incremental SV Checkpoint)方法,提出不直接存储参数矩阵,而是存储充分向量组(Sufficient Vector Group,SVG),即为每条数据生成的一组 SV。在训练过程中,每台机器将每个时刻的 SVG 保存到持久存储中,并使用保存的 SVG 来恢复参数。另外,Orpheus 还提出将每个时刻的 SVG 暂存在内存中,等积累一批之

后，再一起写入到磁盘中，以降低磁盘的写入频率。

(3)压缩检查点

在增量检查点的基础上，一些研究进一步使用了数据压缩技术，利用了检查点之间的相似性，减少了检查点的存储开销。LC-Checkpoint[227]提出先使用量化和优先级提升来存储最重要的信息，然后使用哈夫曼编码继续压缩检查点。Delta-DNN[228]利用了相邻版本之间的浮点数相似性，先使用误差有界的有损压缩算法来计算有损的增量数据，然后使用无损压缩算法来压缩增量数据。Check-N-Run[14]提出执行增量检查点，保存模型的修改部分，并利用量化技术来缩减检查点大小，不会降低模型准确率。QD-Compressor[229]提出了局部敏感的量化机制，先执行参数量化，根据参数的值范围和带权熵，为每层设置自适应的量化器和位宽度，然后计算增量数据并执行无损压缩，还设计了误差反馈机制，能够动态纠正训练过程中的量化误差。Inshrinkerator[225]提出了不均匀的量化机制，设计了动态的量化配置搜索机制，可以根据训练过程中参数对压缩的敏感性变化来自动调整量化配置，实现了量化感知的增量压缩机制，会重新排列模型参数，并使用运行长度编码和哈夫曼编码来降低存储占用。ExCP[230]提出先计算相邻检查点的残差值，再执行权重－动量联合剪枝，最后对权重和动量执行不均匀量化。

(4)即时检查点

针对分布式训练场景，Gupta 等人[231]提出了即时(just-in-time)检查点方法(工作流程如图 39 所示)，只在发生故障之后才执行检查点操作，保存正常节点中的模型状态副本，并使用保存的检查点文件来恢复所有节点的训练，最多只需要重新执行一次迭代的计算工作，避免了频繁执行检查点操作带来的开销，同时降低了恢复开销。对于能够修改代码并且已经支持检查点的训练任务，Gupta 等人设计了用户级方法，添加拦截库以支持即时检查点；对

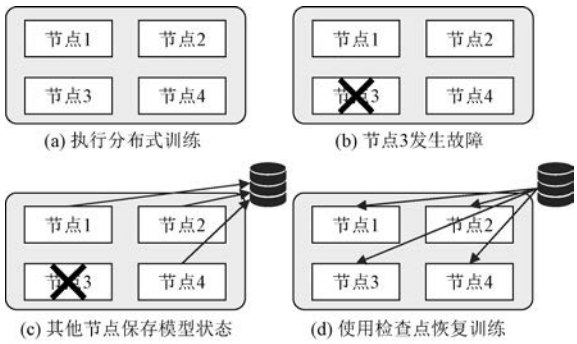

图 39 即时检查点[231]

于不支持检查点的用户脚本，Gupta 等人设计了透明的方法，实现了系统级即时检查点，无需修改应用代码。

(5)部分恢复

当发生故障时，所有训练节点都加载最新的检查点，将模型状态回滚到一致的版本，该方法称为完全恢复(Full Recovery)[232]。根据 Meta 的统计[232]：当使用完全恢复时，检查点开销平均会占用 12%的训练时间，而故障恢复则需要消耗超过 1000 机器－年的计算量。为了降低恢复开销，一些研究提出了部分恢复(Partial Recovery)方法，只在故障节点中加载最新的检查点，允许其他节点继续执行训练。SCAR[233]首次提出了部分恢复方法，只恢复丢失的模型参数，降低了重算开销，还实现了优先级检查点，提高了检查点的执行频率，并且每次只保存变化最大的部分参数。CPR[232]提出预测部分恢复的好处，选择合适的检查点保存间隔，优先保存频繁访问的参数，降低了检查点的开销，提供了合适的模型准确率。

案例分析：根据文献[219]的统计，在训练任务每 5 小时中断一次的情况下，当使用 1080Ti GPU 训练 ResNet50 和 V100 GPU 训练 ResNet101 时，CheckFreq 的训练时间分别缩短了 50%和 37.5%。

5.2.2 其他容错技术

在正常训练和故障恢复的过程中，检查点方法存在着显著的时间开销。随着模型规模的不断变大，检查点的开销也会进一步变大。因此，一些研究使用了其他方法来降低模型训练的容错开销。

(1)多副本

针对数据并行场景，Swift[234]提出撤销故障发生时的更新操作，解决了故障造成的模型状态不一致的情况，并使用存活工作者中存储的模型状态副本来执行故障恢复。

针对混合并行场景，Oobleck[235]使用了计划－执行协同设计方法，在计划阶段生成一系列逻辑上等价、物理上异构的流水线模板(拥有不同的节点数量和配置)，在执行阶段至少实例化 $f+1$ 条流水线，以便允许最多 f 条流水线同时发生故障，并根据不同流水线的计算能力，将全局小批量按比例分发给所有的流水线副本。当某个流水线发生故障时，Oobleck 会根据流水线模板重新实例化流水线，并从其他流水线副本中复制丢失的模型状态。

Oobleck[235]提出的流水线重新实例化方法包含三个步骤：第一，简单重新实例化。对于每条流水

线,如果剩余节点有可以实例化的流水线模板,Oobleck 将会重新实例化该流水线模板,并替换老的流水线。第二,借用节点。如果没有合适的流水线模板,Oobleck 将会从其他流水线借用节点,实例化最小的流水线模板,并重新实例化借出节点的流水线。第三,合并流水线。如果所有流水线都无法借出节点,Oobleck 则会将多条流水线合并成一条更大的流水线。图 40 描述了 Oobleck 重新实例化流水线的过程。其中,Oobleck 生成了 3 个流水线模板,分别包含 2、3 以及 4 个节点,并实例化了 4 条流水线:2 条包含 4 个节点的流水线、1 条包含 3 个节点的流水线以及 1 条包含 2 个节点的流水线。

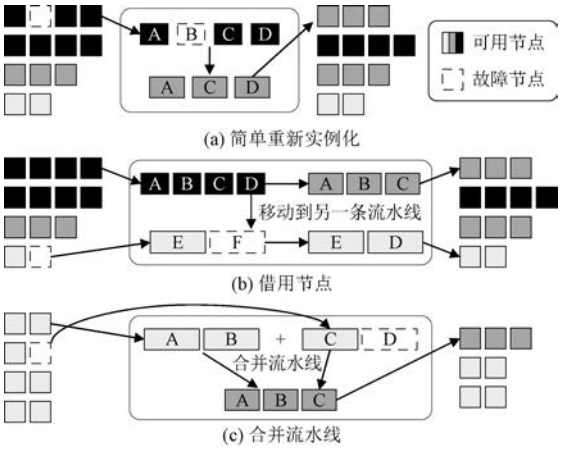

图 40　流水线重新实例化的三个步骤[235]

(2)纠删码

ECRec[236]提出将纠删码应用在推荐模型中。对于稀疏的嵌入表,ECRec 使用了纠删码,为 k 个嵌入向量和优化器状态生成 r 个校验向量和校验优化器状态($r=1$),并采用了旋转校验码分布,将校验码均匀地分散在不同的服务器中。为了正确更新校验码,ECRec 提出了差异传播:首先,工作者将梯度发送给存储对应嵌入向量的服务器;然后,服务器更新嵌入向量和优化器状态,并将差异发送给存储校验码的服务器;最后,存储校验码的服务器将差异添加到校验向量和优化器状态中。旋转校验码分布和差异传播如图 41 所示。对于较小且稠密的神经网络参数,ECRec 使用了多副本,避免了差异传播及其带来的网络开销。此外,ECRec 还支持在故障恢复的过程中继续训练,并使用粒度锁和两阶段提交分别保证了参数恢复的正确性和一致性。

(3)日志

Swift[234]提出了面向流水线并行的日志恢复方法。具体来说,Swift 提出在气泡时间中将中间数

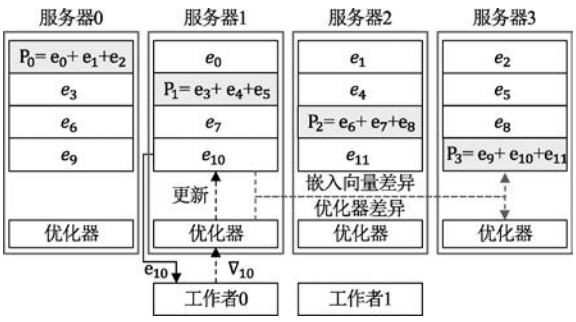

图 41　旋转校验码分布和差异传播示例($k=3$, $r=1$)[236]

据(激活数据和梯度)和相关元数据(发送者、接收者以及时间戳)异步写入日志文件,设计了并行恢复方法,将模型状态重算工作分发给所有的工作者(包括替换工作者和存活工作者),具体流程如图 42 所示。此外,Swift 还实现了选择性日志策略,将机器分组,只将组间的通信数据写入日志文件,不记录组内的通信数据,以牺牲恢复时间为代价,降低了存储空间开销。

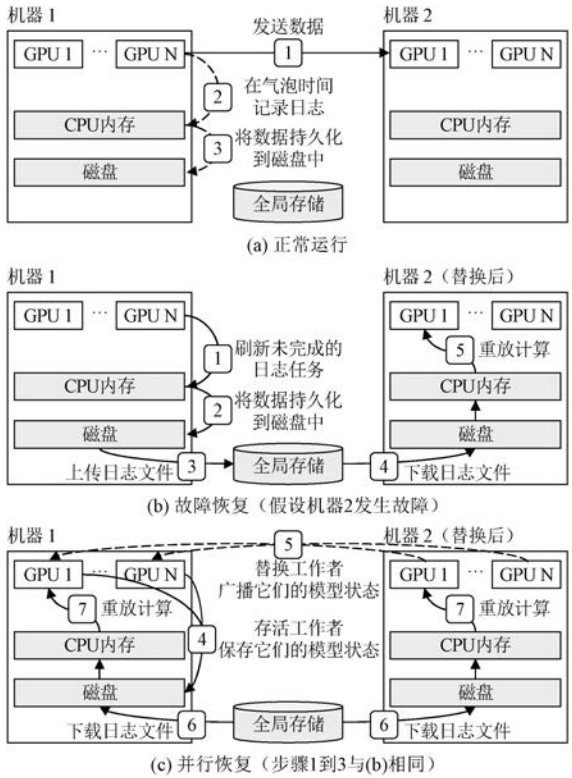

图 42　Swift 日志机制[234]

(4)冗余计算

Bamboo[237]提出使用抢占式实例来降低训练成本,并使用冗余计算来保证可靠性和效率,即每个节点不仅会存储自己负责的层分片,执行正常计算,还会冗余存储后继节点负责的层分片,执行冗余计算,具体如图 43 所示。其中,FNC_n 和 BNC_n 分别表示

节点 n 上的前向正常计算(Forward Normal Computation)和反向正常计算(Backward Normal Computation),而 FRC_n 和 BRC_n 则分别表示在节点 $n-1$ 上为节点 n 执行前向冗余计算(Forward Redundant Computation)和反向冗余计算(Backward Redundant Computation)。为了降低时间开销,Bamboo 提出将每个节点上的前向冗余计算异步调度到流水线气泡中,并将不能放入气泡的部分前向冗余计算与正常计算进行重叠,只在发生抢占时才执行反向冗余计算,恢复丢失的模型状态。为了降低 GPU 显存开销,Bamboo 提出将每个节点中前向冗余计算的中间结果换出到 CPU 内存中,仅当发生抢占时才重新换入到 GPU 显存中。

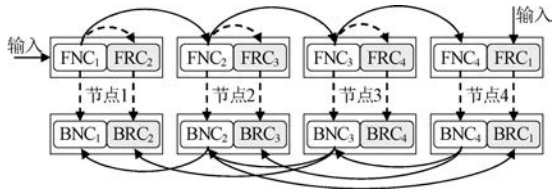

图 43 冗余计算示意图[237]

案例分析:根据文献[237]的统计,当训练 ResNet152 和 GPT-2 等 6 个模型时,与使用按需实例的方法相比,Bamboo 的成本降低了 72.2%;而与检查点方法相比,Bamboo 的吞吐量则提高了 3.7 倍。

5.3 模型存储系统

在训练结束之后,模型参数和模型结构等模型文件需要保存到持久存储中,以便用于后续的迁移学习、模型微调和推理任务等。当前,一些研究提出了专门的模型存储系统,能够保存不同版本的模型文件,还设计了特定的模型加载系统,优化了存储系统的模型加载性能。接下来,本节将从存储架构优化和模型加载优化两个方面分别介绍相关工作。

5.3.1 存储架构优化

ModelHub[238]提出了模型版本控制系统 dlv,设计了用于模型探索和模型枚举查询的高级领域特定语言 DQL,实现了读取优化的参数归档存储系统 PAS。PAS 会单独存储参数的低位字节,并使用增量编码来归档存储不同版本的模型,实现了渐进式的模型评估算法,尽量避免读取参数矩阵的低位字节。此外,PAS 还实现了针对一系列模型版本的存储优化算法,能够最小化参数矩阵的存储开销,同时不影响查询性能。

FlameStore[239]可以将深度学习框架 Keras 生成的模型存储到各种存储后端中,例如内存和本地文件系统等①。FlameStore 使用单个节点来管理元数据,包括模型名称、JSON 格式的模型结构、模型参数的位置以及用户提供的其他元数据,并将单个模型存储在单个存储节点中,还允许用户插入额外的控制器模块,以实现智能的数据管理。在保存或者加载模型时,FlameStore 的客户端和存储节点会使用 RDMA 来传输模型参数。

DStore[240]是美国约翰霍普金斯大学和阿贡国家实验室联合提出的模型存储系统,用于满足模型仓库的高频率并发访问和模型张量的细粒度访问需求,架构如图 44 所示。具体来说,DStore 将深度学习模型分解为张量和相关元数据的轻量级集合,提出了模型结构的紧凑表示、保留模型结构特征的层元数据稳定哈希算法以及客户端的元数据缓存,实现了存储服务的可扩展性和负载均衡,设计了 RDMA 优化的数据暂存机制,还定义了原生的低级张量算子,允许直接访问原始的张量数据。

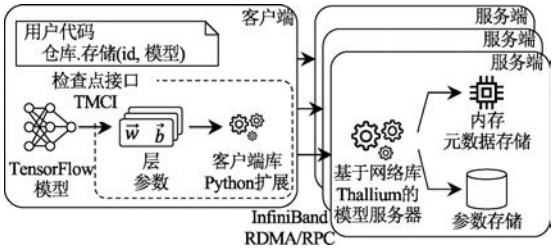

图 44 DStore 架构[240]

EvoStore[241]提出了张量级增量存储和基于引用计数的垃圾回收方法,实现了支持 RDMA 的分布式张量合并存储,设计了模型结构元数据组织和查询方法以寻找最长公共有向图前缀,还实现了分布式元数据查询引擎,用于搜索最匹配的深度学习模型。

表 25 总结了不同模型存储系统在是否支持参数增量存储、使用的模型结构存储方式以及是否支持模型溯源查询等方面的差异。

表 25 不同模型存储系统对比

方法	参数增量存储	模型结构存储	模型溯源查询
ModelHub[238]	是	数据库中的表	是
FlameStore[239]	否	JSON	否
DStore[240]	否	键值对	否
EvoStore[241]	是	键值对	是

5.3.2 模型加载优化

在无服务器推理场景中,模型文件存储在模型

① Storage system for Deep Learning models designed using the Mochi components, https://github.com/mochi-hpc/flamestore 2024.7.12。

存储系统中。当收到推理请求时,调度器会选择合适的 GPU 服务器,从远程存储中加载模型。这种方式降低了部署开销,但存在着推理冷启动的问题,模型加载开销较大,请求延迟较高[242]。因此,一些研究优化了模型加载机制,降低了存储系统的模型加载延迟。

ServerlessLLM[242]是爱丁堡大学和南洋理工大学联合提出的 LLM 无服务器推理系统,包含了三种优化方法:第一,ServerlessLLM 提出了快速的多级检查点加载方法,设计了加载优化的检查点格式和多级检查点加载系统,实现了基于块的顺序读取、高效的张量显存寻址、数据块内存池,高效的内存拷贝数据路径以及多级加载流水线,能够充分利用 GPU、DRAM、SSD 以及远程存储等多个存储层级的容量和带宽,具体如图 45 所示(每块 GPU 负责的张量被划分在相同的分区中);第二,ServerlessLLM 提出了针对 LLM 推理的高效实时迁移机制,只迁移源服务器中的 token,并在目标服务器上重新计算这些 token 的 KV 缓存,降低了网络开销;第三,ServerlessLLM 还设计了启动时间优化的模型调度策略,可以准确预测加载检查点所需的时间和迁移 LLM 推理所需的时间,并将模型调度到启动时间最少的服务器上。

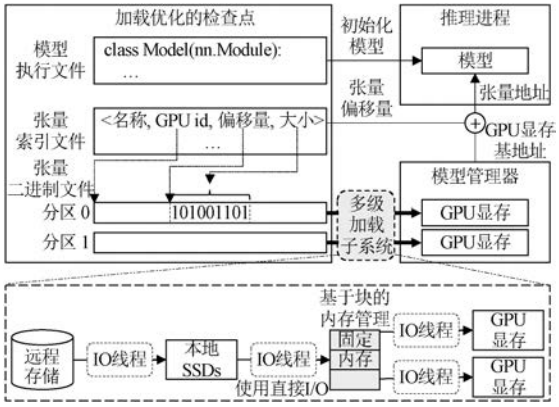

图 45　快速的多级检查点加载方法[242]

案例分析:根据文献[242]的统计,当在 GSM8K 和 ShareGPT 数据集上执行 OPT-6.7B、OPT-13B 以及 OPT-30B 模型的推理任务时,ServerlessLLM 的延迟是现有方法的 200 分之 1 至 10 分之 1。

5.4　性能测试与分析工具

5.4.1　性能测试工具

当前,一些研究提出了面向深度学习训练的 I/O 性能测试工具,可以模拟计算机视觉和自然语言处理等应用场景的 I/O 负载,能够评估存储系统和数据加载器的性能。

(1)存储系统测试工具。SPECstorage Solution 2020① 添加了 AI 图像处理负载 AI_IMAGE,包含 4 个子组件:AI_SF(读取小文件)、AI_TF(创建 TFRecord 文件)、AI_TR(读取 TFRecord 文件)以及 AI_CP(写检查点)。DIOT(Deep Leaning I/O Toolkit)[243]使用了 I/O 测试工具 fio,模拟了多个深度学习负载的 I/O 模式,可以评估存储系统的带宽和 IOPS(Input/Output Operations Per Second)。类似地,DLIO(Deep Learning I/O)[244]使用了模块化设计,可以整合多种数据加载器、存储格式、数据集以及配置参数,能够模拟各种深度学习应用的 I/O 模式和行为。MLPerf Storage② 使用了 DLIO,可以测试存储系统在机器学习负载下的性能。

(2)数据加载器测试工具。Ofeidis 等人[245]提出了数据加载器基准测试 dataloader-benchmarks,比较了 PyTorch DataLoader、FFCV、Hub、Deep Lake、Torchdata、Webdataset、Squirrel 七个数据加载器在本地数据加载、远程数据加载、数据集过滤以及多 GPU 训练场景下的性能。

表 26 总结了不同性能测试工具在评价指标、测试对象、是否读取数据集、是否写检查点以及是否执行实际的模型训练等方面的差异。

表 26　面向训练的 I/O 性能测试工具对比

性能测试工具	评价指标	测试对象	读取数据集	写检查点	执行实际的模型训练
SPECstorage Solution 2020(AI_IMAGE)	任务数	存储系统	是	是	否
DIOT[243]	带宽、IOPS	存储系统	是	否	否
DLIO[244]	运行时间、带宽	存储系统	是	支持	否
MLPerf Storage	每秒处理的样本数量	存储系统	是	支持	否
dataloader-benchmarks[245]	每秒处理的样本数量	数据加载器	是	否	是

5.4.2　性能分析工具

现有的 I/O 性能分析工具,例如,iostat、dstat 以及 Darshan[246] 等,可以分析深度学习应用的 CPU 和存储设备利用率等。此外,TensorFlow 和 PyTorch 等框架也提供了自带的性能分析工具,可

① SPECstorage ® Solution 2020,https://www.spec.org/storage2020/2024,6,20。

② MLPerf™ Storage Benchmark Suite,https://github.com/mlcommons/storage 2024,7,25。

以分析不同阶段的执行时间等。以 TensorFlow 为例,它提供了分析器 TensorFlow Profiler 和基于 Web 的可视化工具 TensorBoard。

然而,TensorFlow 的分析器只能提供粗粒度的平台级信息,无法提供细粒度的 I/O 信息,例如,元数据操作花费的时间等[247]。因此,tf-Darshan[247]扩展了 TensorFlow 的分析器,使用 Darshan 分析器来执行插桩,实现了插桩函数的运行时绑定,能够抽取 Darshan 的数据结构,并交给 TensorFlow 的分析器进行分析,还使用 TensorBoard 来展示分析结果。类似地,Devarajan 等人[248]分别使用 Darshan 和 TensorFlow 分析器来执行低级和高级的性能分析,并设计了 I/O 分析工具 VaniDL,能够整合低级的 Darshan 日志和高级的 TensorFlow 分析器日志,可以生成全面的 I/O 信息,包括访问模式、传输大小分布以及访问时间线等。

DS-Analyzer[46]可以分析训练场景的数据挂起问题,共分三个阶段执行:首先,在 GPU 中预先填充一些数据,并执行固定轮次的训练,测量 GPU 的最大处理速度;其次,将指定数据集的部分数据缓存在内存中,并使用所有可用的 CPU 核心执行数据预处理,比较第一阶段和第二阶段的性能差异,评估是否存在着数据预处理挂起的问题;最后,清空所有缓存,并将缓存大小设置为用户指定的上限,比较第二阶段和第三阶段的性能差异,评估是否存在着数据加载挂起的问题。

5.5 总结与对比

图 46 总结了现有的、面向模型计算的存储优化技术。其中,模型状态存储技术解决了如何高效存下模型状态的问题,权衡了存储容量、存储带宽以及模型准确率。模型训练容错技术解决了训练场景中如何高效容错的问题,权衡了容错性能、存储占用、故障恢复以及模型准确率。模型存储系统研究解决了如何长期、高效存储模型文件的问题。而性能测试与分析工具研究则旨在评估存储系统和数据加载器的 I/O 性能,寻找可能的 I/O 瓶颈。

图 46 面向模型计算的存储优化技术

6 研究展望

当前,面向深度学习的数据存储技术研究已经取得了显著的成果,能够极大地提高模型训练和推理的性能。然而,现有的工作仍然存在着一些不足,值得进一步研究:

(1)统一支持各种类型的数据集。一方面,根据

前文对深度学习数据特点的分析,不同领域的深度学习模型处理的数据类型各不相同,对数据存储的需求也会有所不同。另一方面,随着深度学习技术的不断发展,多模态大模型越来越流行。多模态大模型研究使用多种数据类型作为模型输入,利用不同数据类型中包含的信息来改进模型的效果。例如,OpenAI 在 2024 年 5 月推出了多模态大模型 GPT-4o,能够同时处理文本、音频、图像以及视频①。然而,现有的研究大都只优化了特定的数据类型,缺乏对各种数据类型的统一支持,难以同时满足不同领域的模型训练需求。

(2)统一管理各类异构存储设备。深度学习训练和推理过程主要涉及数据集和模型状态的加载和保存。具体来说,在数据加载阶段中,深度学习应用需要从存储系统中加载数据集中的样本数据;而在模型计算阶段中,深度学习应用则需要从(向)GPU 显存中读取(写入)模型状态。现有的研究大都侧重于使用不同的存储设备来存储数据集或者模型状态,加快数据预处理或者模型计算,缺乏对 GPU 显存、DRAM、NVM、SSD 以及 HDD 等异构存储设备的统一管理,难以兼顾样本数据的加载、预处理以及模型状态的读写。

(3)基于新型硬件设备的优化。随着硬件技术的不断进步,各种新型硬件设备层出不穷,为深度学习存储优化提供了新的机遇。一方面,新兴的存储设备,例如,CXL 内存和高密度新型存储等,有助于满足深度学习训练和推理任务快速增长的存储容量需求。具体来说,CXL 协议可以实现内存容量的按需分配和扩展,能够满足深度学习的内存容量需求。而高密度新型存储,例如,SMR(Shingled Magnetic Recording,叠瓦式磁记录)、高密度光存储以及 DNA 存储等,具有更高的存储密度,可以满足深度学习的外存容量需求。另一方面,新兴的网络设备 DPU(Data Processing Unit)具有强大的计算能力,可以用于卸载深度学习训练和推理中的数据压缩等操作,能够降低数据传输开销,减轻 CPU 的工作负载。

(4)I/O 栈优化。在传统的 I/O 栈中,数据需要先经过 CPU 拷贝到主存中,然后再经过 CPU 拷贝到 GPU 显存中,访问路径较长。随着数据集和模型状态的规模不断变大,传统 I/O 栈的数据加载速度难以匹配日益增长的 GPU 运算速度。因此,现有的研究利用了 GDS 等 GPU 直连技术②,在数据加载器和模型状态存储优化等方面探索了 GPU 直接访问存储架构。然而,这些研究要么偏向应用层(例如,数据加载器),要么只针对特定的场景(例如,模型状态存储),难以同时满足数据集和模型状态的存储需求,并且缺乏对不同应用的支持。另外,传统的 I/O 栈存在着冗余的权限检查,引入了较大的访问开销。数据集中不同样本的访问权限基本相同,并且很少改变。在一轮训练中,数据集中的所有样本均会被访问一次。尽管样本之间的访问权限基本相同,但是传统的 I/O 栈仍然会检查所有样本的访问权限。另外,在多轮训练中,同一样本会被多次访问。尽管同一样本的访问权限是相同的,但是传统的 I/O 栈仍然会重复检查同一样本的访问权限。因此,数据存储需要针对深度学习应用的数据访问需求,设计统一的 I/O 栈,缩短数据集和模型状态的访问路径,改进样本文件的随机读取性能,提高模型状态的读写性能。

(5)性能测试工具优化。现有的性能测试工具研究在支持的数据集、训练阶段、计算模式、模型规模以及应用场景等方面存在着一些不足。具体来说:第一,现有的研究大都只支持静态数据集,没有考虑动态数据集。在实际的应用环境中,数据集并不是固定不变的,而是会持续更新。第二,现有的研究侧重于评估训练场景中数据准备阶段的 I/O 性能,缺乏对模型训练阶段和端到端训练性能的评估。数据准备阶段和模型训练阶段处理的对象不同,前者涉及数据集的加载和预处理,而后者则涉及大量模型状态的读写。第三,现有的研究大都只支持训练场景,缺乏对推理场景的支持。与训练场景不同,推理场景通常不涉及数据集的加载,不需要保存梯度和优化器状态等信息,但对请求的延迟却十分敏感。第四,现有的研究大都只支持参数量较少的小模型,尚不支持十亿以上参数的大模型。与小模型相比,大模型的参数规模更大,模型参数和激活数据等模型状态需要占用的存储空间也会更大,读写开销更加显著。第五,现有的研究大都只考虑了视觉模型和语言模型,缺乏对其他应用场景的支持,例如,新兴的多模态大模型和工业界广泛应用的深度学习推荐模型等。不同应用场景的数据加载需求和参数读写需求各不相同,存储性能测试工具需要支持各类典型的应用场景。

(6)端侧大模型存储优化。随着大模型应用的

① Hello GPT-4o, https://openai.com/index/hello-gpt-4o/ 2024,6,18。

② NVIDIA GPUDirect, https://developer.nvidia.com/gpudirect 2023,12,23。

普及,在端侧设备上运行大模型的需求变得越来越迫切。目前,学术界和工业界已经开始尝试在端侧设备上执行大模型训练和推理,利用用户的个人数据来生成个性化的大模型[249]。然而,端侧设备的存储容量受限,难以直接运行各种复杂的大模型。以运行 100 亿参数的大模型为例(文献[30]将超过 100 亿参数的语言模型称为大语言模型),如果采用 FP16 来表示参数,模型参数本身就需要占用 19GB 的存储空间。如果再加上运行中需要存储的临时数据,例如 KV 缓存和激活数据等,那么模型所需的存储空间将会更大。尽管现有的研究[196]提出了许多模型压缩技术,降低了模型计算的存储容量需求,但是这些技术可能会影响模型的准确率和计算效率。因此,如何从存储方面优化端侧大模型的训练和推理过程,尚需要进一步的探索。

7 结 论

随着 GPU 等加速器计算能力的不断提高,数据存储已经成为了深度学习训练和推理的主要瓶颈之一。在这样的背景下,“Storage for AI”成为了热门的研究领域。本文调研了“Storage for AI”方向,分析了面向深度学习的数据存储技术,总结了近年来的研究成果。首先,本文梳理了深度学习应用的典型流程,分析了深度学习应用的数据特点,提出了面向深度学习的数据存储技术研究框架。然后,本文从深度学习的不同阶段出发,依次介绍了面向数据加载的存储优化技术、面向数据预处理的存储优化技术以及面向模型计算的存储优化技术。最后,本文讨论了后续的研究方向,介绍了有待解决的研究问题。

致 谢 感谢各位评审专家对本文提出的宝贵建议!

参 考 文 献

[1] Isenko A, Mayer R, Jedele J, et al. Where is my training bottleneck? Hidden trade-offs in deep learning preprocessing pipelines//Proceedings of the 2022 International Conference on Management of Data. Philadelphia, USA, 2022: 1825-1839

[2] Ridnik T, Ben-Baruch E, Noy A, et al. Imagenet-21k pretraining for the masses. arXiv preprint arXiv:2104.10972, 2021

[3] Kuznetsova A, Rom H, Alldrin N, et al. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. International Journal of Computer Vision, 2020, 128(7): 1956-1981

[4] Murray D G, Šimša J, Klimovic A, et al. tf.data: A machine learning data processing framework. Proceedings of the VLDB Endowment, 2021, 14(12): 2945-2958

[5] Russakovsky O, Deng J, Su H, et al. Imagenet large scale visual recognition challenge. International Journal of Computer Vision, 2015, 115(3): 211-252

[6] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Las Vegas, USA, 2016: 770-778

[7] Yang S P, Kim M, Nam S, et al. Overcoming the memory wall with cxl-enabled ssds//Proceedings of the 2023 USENIX Annual Technical Conference. Boston, USA, 2023: 601-617

[8] Feng Yang-Yang, Wang Qing, Xie Min-Hui, et al. From bert to chatgpt: Challenges and technical development of storage systems for large model training. Journal of Computer Research and Development, 2024, 61(4): 809-823 (in Chinese)
(冯杨洋, 汪庆, 谢旻晖, 等. 从 BERT 到 ChatGPT:大模型训练中的存储系统挑战与技术发展. 计算机研究与发展, 2024, 61 (4): 809-823)

[9] Lewis N, Bez J L, Byna S. I/o in machine learning applications on hpc systems: A 360-degree survey. arXiv preprint arXiv:2404.10386, 2024

[10] Jain J, Phanishayee A, Mars J, et al. Gist: Efficient data encoding for deep neural network training//Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture. Los Angeles, USA, 2018: 776-789

[11] Kumar A V, Sivathanu M. Quiver: An informed storage cache for deep learning//Proceedings of the 18th USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2020: 283-296

[12] Gupta U, Wu C J, Wang X, et al. The architectural implications of facebook's dnn-based personalized recommendation//Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture. San Diego, USA, 2020: 488-501

[13] Naumov M, Mudigere D, Shi H J M, et al. Deep learning recommendation model for personalization and recommendation systems. arXiv preprint arXiv:1906.00091, 2019

[14] Eisenman A, Matam K K, Ingram S, et al. Check-n-run: A checkpointing system for training deep learning recommendation models//Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation. Renton, USA, 2022: 929-943

[15] Jiang W, He Z, Zhang S, et al. Microrec: Efficient recommendation inference by hardware and data structure solutions//Proceedings of the 4th Machine Learning and Systems. San Jose, USA, 2021: 845-859

[16] Ardestani E K, Kim C, Lee S J, et al. Supporting massive dlrm inference through software defined memory//Proceed-

ings of the 42nd IEEE International Conference on Distributed Computing Systems. Bologna, Itlay, 2022: 302-312

[17] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks//Proceedings of the 5th International Conference on Learning Representations. Toulon, France, 2017: 1-14

[18] Xu K, Hu W, Leskovec J, et al. How powerful are graph neural networks? //Proceedings of the 7th International Conference on Learning Representations. New Orleans, USA, 2019: 1-17

[19] Park Y, Min S, Lee J W. Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. Proceedings of the VLDB Endowment, 2022, 15(11): 2626-2639

[20] Liu T, Chen Y, Li D, et al. Bgl: Gpu-efficient gnn training by optimizing graph data I/O and preprocessing//Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation. Boston, USA, 2023: 103-118

[21] Hamilton W, Ying Z, Leskovec J. Inductive representation learning on large graphs//Proceedings of the 31st Annual Conference on Neural Information Processing Systems. Long Beach, USA, 2017: 1024-1034

[22] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need//Proceedings of the 31st Annual Conference on Neural Information Processing Systems. Long Beach, USA, 2017: 5998-6008

[23] Ba J L, Kiros J R, Hinton G E. Layer normalization. arXiv preprint arXiv:1607.06450, 2016

[24] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners//Proceedings of the 34th Annual Conference on Neural Information Processing Systems. Vancouver, Canada, 2020: 1877-1901

[25] Dean J, Corrado G, Monga R, et al. Large scale distributed deep networks//Proceedings of the 26th Annual Conference on Neural Information Processing Systems. Lake Tahoe, USA, 2012: 1232-1240

[26] Wang S, Pi A, Zhou X, et al. Overlapping communication with computation in parameter server for scalable dl training. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(9): 2144-2159

[27] Maas A L, Daly R E, Pham P T, et al. Learning word vectors for sentiment analysis//Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. Portland, USA, 2011: 142-150

[28] Rajpurkar P, Jia R, Liang P. Know what you don't know: Unanswerable questions for squad. arXiv preprint arXiv: 1806.03822, 2018

[29] Ma Z, He J, Qiu J, et al. Bagualu: Targeting brain scale pretrained models with over 37 million cores//Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Seoul, Republic of Korea, 2022: 192-204

[30] Zhao W X, Zhou K, Li J, et al. A survey of large language models. arXiv preprint arXiv:2303.18223, 2023

[31] Bai Y, Geng X, Mangalam K, et al. Sequential modeling enables scalable learning for large vision models. arXiv preprint arXiv:2312.00785, 2023

[32] Zhang D, Yu Y, Li C, et al. Mm-llms: Recent advances in multimodal large language models. arXiv preprint arXiv: 2401.13601, 2024

[33] Carolan K, Fennelly L, Smeaton A F. A review of multi-modal large language and vision models. arXiv preprint arXiv: 2404.01322, 2024

[34] Nair V, Hinton G E. Rectified linear units improve restricted boltzmann machines//Proceedings of the 27th International Conference on Machine Learning. Haifa, Israel, 2010: 807-814

[35] Gong Z, Ji H, Fletcher C W, et al. Save: Sparsity-aware vector engine for accelerating dnn training and inference on cpus//Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture. Athens, Greece, 2020: 796-810

[36] Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 2014, 15(1): 1929-1958

[37] Abadi M, Barham P, Chen J, et al. Tensorflow: A system for large-scale machine learning//Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation. Savannah, USA, 2016: 265-283

[38] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library//Proceedings of the 33rd Annual Conference on Neural Information Processing Systems. Vancouver, Canada, 2019: 8024-8035

[39] Zhang Z, Huang L, Manor U, et al. Fanstore: Enabling efficient and scalable I/O for distributed deep learning. arXiv preprint arXiv:1809.10799, 2018

[40] Zhang Z, Huang L, Pauloski J G, et al. Efficient I/O for neural network training with compressed data//Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium. New Orleans, USA, 2020: 409-418

[41] Chowdhury F, Zhu Y, Heer T, et al. I/O characterization and performance evaluation of beegfs for deep learning//Proceedings of the 48th International Conference on Parallel Processing. Kyoto, Japan, 2019: 1-10

[42] Xie J, Xu J, Wang G, et al. A deep learning dataloader with shared data preparation//Proceedings of the 36th Annual Conference on Neural Information Processing Systems. New Orleans, USA, 2022: 17146-17156

[43] Adnan M, Maboud Y E, Mahajan D, et al. Accelerating recommendation system training by leveraging popular choices. Proceedings of the VLDB Endowment, 2021, 15(1): 127-140

[44] Renz-Wieland A, Gemulla R, Kaoudi Z. Nups: A parameter server for machine learning with non-uniform parameter access//Proceedings of the 2022 International Conference on Management of Data. Philadelphia, USA, 2022: 481-495

[45] Bae J, Lee J, Jin Y, et al. Flashneuron: Ssd-enabled large-batch training of very deep neural networks//Proceedings of the 19th USENIX Conference on File and Storage Technologies. Virtual , 2021: 387-401

[46] Mohan J, Phanishayee A, Raniwala A, et al. Analyzing and mitigating data stalls in dnn training. Proceedings of the VLDB Endowment, 2021, 14(5): 771-784

[47] Bae J, Baek W, Ham T J, et al. L3: Accelerator-friendly lossless image format for high-resolution, high-throughput dnn training//Proceedings of the 17th European Conference on Computer Vision. Tel Aviv, Israel, 2022: 171-188

[48] Sirin U, Idreos S. The image calculator: 10x faster image-ai inference by replacing jpeg with self-designing storage format. Proceedings of the ACM on Management of Data, 2024, 2(1): 1-31

[49] Chen T, Li M, Li Y, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274, 2015

[50] Wang L, Ye S, Yang B, et al. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training//Proceedings of the 49th International Conference on Parallel Processing. Edmonton, Canada, 2020: 1-11

[51] Aizman A, Maltby G, Breuel T. High performance I/O for large scale deep learning//Proceedings of the 7th IEEE International Conference on Big Data. Los Angeles, USA, 2019: 5965-5967

[52] Zhu Z, Tan L, Li Y, et al. Phdfs: Optimizing i/o performance of hdfs in deep learning cloud computing platform. Journal of Systems Architecture, 2020, 109: 101810

[53] Leclerc G, Ilyas A, Engstrom L, et al. Ffcv: Accelerating training by removing data bottlenecks//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. Vancouver, Canada, 2023: 12011-12020

[54] Kuchnik M, Amvrosiadis G, Smith V. Progressive compressed records: Taking a byte out of deep learning data. Proceedings of the VLDB Endowment, 2021, 14(11): 2627-2641

[55] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding//Proceedings of the 22nd ACM international conference on Multimedia. Orlando, USA, 2014: 675-678

[56] Chu H. Mdb: A memory-mapped database and backend for openldap//Proceedings of the 3rd International Conference on LDAP. Heidelberg, Germany, 2011: 1-12

[57] Lim S H, Young S R, Patton R M. An analysis of image storage systems for scalable training of deep neural networks//Proceedings of the 7th Workshop on Big Data Benchmarks, Performance, Optimization, and Emerging Hardware. Atlanta, USA, 2016: 1-8

[58] Zhao M, Agarwal N, Basant A, et al. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product//Proceedings of the 49th Annual International Symposium on Computer Architecture. New York, USA, 2022: 1042-1057

[59] Vakharia S, Li P, Liu W, et al. Shared foundations: Modernizing meta's data lakehouse//Proceedings of the 13th Conference on Innovative Data Systems Research. Amsterdam, The Netherlands. 2023: 1-7

[60] Hambardzumyan S, Tuli A, Ghukasyan L, et al. Deep lake: A lakehouse for deep learning//Proceedings of the 13th Conference on Innovative Data Systems Research. Amsterdam, The Netherlands, 2023: 1-12

[61] Kurth T, Treichler S, Romero J, et al. Exascale deep learning for climate analytics//Proceedings of the 30th International Conference for High Performance Computing, Networking, Storage and Analysis. Dallas, USA, 2018: 649-660

[62] Zhu Y, Yu W, Jiao B, et al. Efficient user-level storage disaggregation for deep learning//Proceedings of the 21st IEEE International Conference on Cluster Computing. Albuquerque, USA, 2019: 1-12

[63] Schimmelpfennig F, Vef M A, Salkhordeh R, et al. Streamlining distributed deep learning I/O with ad hoc file systems//Proceedings of the 23rd IEEE International Conference on Cluster Computing. Portland, USA, 2021: 169-180

[64] Vef M A, Moti N, Süß T, et al. Gekkofs-a temporary distributed file system for hpc applications//Proceedings of the 20th IEEE International Conference on Cluster Computing. Belfast, UK, 2018: 319-324

[65] Wang L, Luo Q, Yan S. Diesel+: Accelerating distributed deep learning tasks on image datasets. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(5): 1173-1184

[66] Pan S, Stavrinos T, Zhang Y, et al. Facebook's tectonic filesystem: Efficiency from exascale//Proceedings of the 19th USENIX Conference on File and Storage Technologies. Virtual, 2021: 217-231

[67] Zhao M, Pan S, Agarwal N, et al. Tectonic-shift: A composite storage fabric for large-scale ml training//Proceedings of the 2023 USENIX Annual Technical Conference. Boston, USA, 2023: 433-449

[68] Pumma S, Si M, Feng W, et al. Towards scalable deep learning via I/O analysis and optimization//Proceedings of the 19th IEEE International Conference on High Performance Computing and Communications; 15th IEEE International Conference on Smart City; 3rd IEEE International Conference on Data Science and Systems. Bangkok, Thailand, 2017: 223-230

[69] Pumma S, Si M, Feng W, et al. Parallel I/O optimizations for scalable deep learning//Proceedings of the 23rd IEEE International Conference on Parallel and Distributed Systems. Shenzhen, China, 2017: 720-729

[70] Pumma S, Si M, Feng W C, et al. Scalable deep learning via I/O analysis and optimization. ACM Transactions on Parallel Computing, 2019, 6(2): 1-34

[71] Zheng D, Ma C, Wang M, et al. Distdgl: Distributed graph neural network training for billion-scale graphs//Proceedings of the 10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms. GA, USA, 2020: 36-44

[72] Ozeri O, Ofer E, Kat R. Object storage for deep learning frameworks//Proceedings of the 2nd Workshop on Distributed Infrastructures for Deep Learning. Rennes, France, 2018: 21-24

[73] Ebido T J, Park K C, Jeon K. Targoat: Improving dataset upload time to object storage using client-server cooperation//Proceedings of the 7th IEEE International Conference on Big Data. Los Angeles, USA, 2019: 6043-6045

[74] Hu P, Gu Y, Jia R, et al. Prm: An efficient partial recovery method to accelerate training data reconstruction for distributed deep learning applications in cloud storage systems//Proceedings of the 30th IEEE/ACM International Symposium on Quality of Service. Oslo, Norway, 2022: 1-10

[75] Dantas M, Leitao D, Correia C, et al. Monarch: Hierarchical storage management for deep learning frameworks//Proceedings of the 23rd IEEE International Conference on Cluster Computing. Portland, USA, 2021: 657-663

[76] Dantas M, Leitao D, Cui P, et al. Accelerating deep learning training through transparent storage tiering//Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing. Taormina, Italy, 2022: 21-30

[77] Pinto C, Gkoufas Y, Reale A, et al. Hoard: A distributed data caching system to accelerate deep learning training on the cloud. arXiv preprint arXiv:1812.00669, 2018

[78] Arif M, Assogba K, Rafique M M, et al. Exploiting cxl-based memory for distributed deep learning//Proceedings of the 51st International Conference on Parallel Processing. Bordeaux, France, 2022: 1-11

[79] Khan A, Paul A K, Zimmer C, et al. Hvac: Removing I/O bottleneck for large-scale deep learning applications//Proceedings of the 24th IEEE International Conference on Cluster Computing. Heidelberg, Germany, 2022: 324-335

[80] Macedo R, Correia C, Dantas M, et al. The case for storage optimization decoupling in deep learning frameworks//Proceedings of the 23rd IEEE International Conference on Cluster Computing. Portland, USA, 2021: 649-656

[81] Dryden N, Böhringer R, Ben-Nun T, et al. Clairvoyant prefetching for distributed machine learning I/O//Proceedings of the 33rd International Conference for High Performance Computing, Networking, Storage and Analysis. St. Louis, USA, 2021: 1-15

[82] Zhu Y, Chowdhury F, Fu H, et al. Entropy-aware I/O pipelining for large-scale deep learning on hpc systems//Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. Milwaukee, USA, 2018: 145-156

[83] Zhu Y, Chowdhury F, Fu H, et al. Multi-client deepio for large-scale deep learning on hpc systems//Proceedings of the 30th International Conference on High Performance Computing, Networking, Storage and Analysis. Dallas, USA, 2018: 1-3

[84] Katharopoulos A, Fleuret F. Not all samples are created equal: Deep learning with importance sampling//Proceedings of the 35th International Conference on Machine Learning. Stockholm, Sweden, 2018: 2525-2534

[85] Chen W, He S, Xu Y, et al. icache: An importance-sampling-informed cache for accelerating I/O-bound dnn model training//Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture. Montreal, Canada, 2023: 220-232

[86] Khan R I S, Yazdani A H, Fu Y, et al. Shade: Enable fundamental cacheability for distributed deep learning training//Proceedings of the 21st USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2023: 135-152

[87] Gu R, Zhang K, Xu Z, et al. Fluid: Dataset abstraction and elastic acceleration for cloud-native deep learning training jobs//Proceedings of the 38th IEEE International Conference on Data Engineering. Kuala Lumpur, Malaysia, 2022: 2182-2195

[88] Mohan J, Phanishayee A, Kulkarni J, et al. Looking beyond gpus for dnn scheduling on multi-tenant clusters//Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation. Carlsbad, USA, 2022: 579-596

[89] Zhao H, Han Z, Yang Z, et al. Silod: A co-design of caching and scheduling for deep learning clusters//Proceedings of the 18th European Conference on Computer Systems. Rome, Italy, 2023: 883-898

[90] Li M, Han Z, Zhang C, et al. Dynamic resource allocation for deep learning clusters with separated compute and storage//Proceedings of the 42nd IEEE Conference on Computer Communications. New York, USA, 2023: 1-10

[91] Ke Z L, Cheng H Y, Yang C L, et al. Analyzing the interplay between random shuffling and storage devices for efficient machine learning//Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software. Stony Brook, USA, 2021: 276-287

[92] Nguyen T T, Trahay F, Domke J, et al. Why globally reshuffle? Revisiting data shuffling in large scale deep learning//Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium. Lyon, France, 2022: 1085-1096

[93] Folk M, Heber G, Koziol Q, et al. An overview of the hdf5 technology suite and its applications//Proceedings of the 2011 EDBT/ICDT Workshop on Array Databases. Uppsala, Sweden, 2011: 36-47

[94] Svogor I, Eichenberger C, Spanring M, et al. Profiling and improving the pytorch dataloader for high-latency storage: A

technical report. arXiv preprint arXiv:2211.04908, 2022

[95] Lee S, Kang Q, Wang K, et al. Asynchronous I/O strategy for large-scale deep learning applications//Proceedings of the 28th IEEE International Conference on High Performance Computing, Data, and Analytics. Bengaluru, India, 2021: 322-331

[96] Serizawa K, Tatebe O. Accelerating machine learning I/O by overlapping data staging and mini-batch generations//Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies. Auckland, New Zealand, 2019: 31-34

[97] Ruan X, Chen H. Informed prefetching in I/O bounded distributed deep learning//Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium Workshops. Portland, USA, 2021: 850-857

[98] Bai Y, Li C, Lin Z, et al. Efficient data loader for fast sampling-based gnn training on large graphs. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(10): 2541-2556

[99] Liu J, Nicolae B, Li D. Lobster: Load balance-aware I/O for distributed dnn training//Proceedings of the 51st International Conference on Parallel Processing. Bordeaux, France, 2022: 1-11

[100] Sun B, Yu X, Zhang C, et al. Solar: A highly optimized data loading framework for distributed training of cnn-based scientific surrogates. arXiv preprint arXiv:2211.00224, 2022

[101] Zhu R, Zhao K, Yang H, et al. Aligraph: A comprehensive graph neural network platform. Proceedings of the VLDB Endowment, 2019, 12(12): 2094-2105

[102] Waleffe R, Mohoney J, Rekatsinas T, et al. Mariusgnn: Resource-efficient out-of-core training of graph neural networks//Proceedings of the 18th European Conference on Computer Systems. Rome, Italy, 2023: 144-161

[103] Yang C C, Cong G. Accelerating data loading in deep neural network training//Proceedings of the 26th IEEE International Conference on High Performance Computing, Data, and Analytics. Hyderabad, India, 2019: 235-245

[104] Chen D, Liang S, Hu G, et al. Mitigating data stalls in deep learning with multi-times data loading rule//Proceedings of the 28th International Conference on Database Systems for Advanced Applications. Tianjin, China, 2023: 562-577

[105] Min S W, Wu K, Huang S, et al. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. arXiv preprint arXiv: 2101.07956, 2021

[106] Tang D, Wang J, Chen R, et al. Xgnn: Boosting multi-gpu gnn training via global gnn memory store. Proceedings of the VLDB Endowment, 2024, 17(5): 1105-1118

[107] Park J B, Mailthody V S, Qureshi Z, et al. Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses. Proceedings of the VLDB Endowment, 2024, 17(6): 1227-1240

[108] Ibrahim K Z, Oliker L. Preprocessing pipeline optimization for scientific deep learning workloads//Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium. Lyon, France, 2022: 1118-1128

[109] Behme L, Thirumuruganathan S, Mahdiraji A R, et al. The art of losing to win: Using lossy image compression to improve data loading in deep learning pipelines//Proceedings of the 39th IEEE International Conference on Data Engineering. Anaheim, USA, 2023: 936-949

[110] Baek W, Bae J, Lee D, et al. Liquid: Mix-and-match multiple image formats to balance dnn training pipeline//Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems. Seoul, Republic of Korea, 2023: 50-57

[111] Cubuk E D, Zoph B, Mane D, et al. Autoaugment: Learning augmentation strategies from data//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. Long Beach, USA, 2019: 113-123

[112] Nouaji R, Bitchebe S, Balmau O. Speedyloader: Efficient pipelining of data preprocessing and machine learning training//Proceedings of the 4th Workshop on Machine Learning and Systems. Athens, Greece, 2024: 65-72

[113] Jia D, Yuan G, Lin X, et al. A data-loader tunable knob to shorten gpu idleness for distributed deep learning//Proceedings of the 15th IEEE International Conference on Cloud Computing. Barcelona, Spain, 2022: 449-458

[114] Audibert A, Chen Y, Graur D, et al. tf.data service: A case for disaggregating ml input data processing//Proceedings of the 14th ACM Symposium on Cloud Computing. Santa Cruz, USA, 2023: 358-375

[115] Zhao H, Yang Z, Cheng Y, et al. Goldminer: Elastic scaling of training data pre-processing pipelines for deep learning. Proceedings of the ACM on Management of Data, 2023, 1(2): 1-25

[116] Um T, Oh B, Seo B, et al. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. Proceedings of the VLDB Endowment, 2023, 16(5): 1086-1099

[117] Graur D, Mraz O, Li M, et al. Pecan: Cost-efficient ml data preprocessing with automatic transformation ordering and hybrid placement//Proceedings of the 2024 USENIX Annual Technical Conference. Santa Clara, USA, 2024: 649-665

[118] Zhao M, Adamiak E, Kozyrakis C. cedar: Composable and optimized machine learning input data pipelines. arXiv preprint arXiv:2401.08895, 2024

[119] Choi D, Passos A, Shallue C J, et al. Faster neural network training with data echoing. arXiv preprint arXiv:1907.05550, 2019

[120] Agarwal N, Anil R, Koren T, et al. Stochastic optimization with laggard data pipelines//Proceedings of the 34th Annual Conference on Neural Information Processing Systems. Vancouver, Canada, 2020: 10282-10293

[121] Lee G, Lee I, Ha H, et al. Refurbish your training data:

Reusing partially augmented samples for faster deep neural network training//Proceedings of the 2021 USENIX Annual Technical Conference. Virtual Event, 2021: 537-550

[122] Wu X. Optimizing resource utilization, efficiency and scalability in deep learning systems[Ph. D. dissertation]. The University of Texas at Arlington,USA, 2023

[123] Kakaraparthy A, Venkatesh A, Phanishayee A, et al. The case for unifying data loading in machine learning clusters//Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing. Renton, USA, 2019: 1-7

[124] Graur D, Aymon D, Kluser D, et al. Cachew: Machine learning input data processing as a service//Proceedings of the 2022 USENIX Annual Technical Conference. Carlsbad, USA, 2022: 689-706

[125] Jin H, Zhu Z, He L, et al. Mmdataloader: Reusing preprocessed data Among concurrent model training tasks. IEEE Transactions on Computers, 2024, 73(2): 510-522

[126] Lee Y, Chung J, Rhu M. Smartsage: Training large-scale graph neural networks using in-storage processing architectures//Proceedings of the 49th Annual International Symposium on Computer Architecture. New York, USA, 2022: 932-945

[127] Li S, Tang K, Lim J, et al. Computational storage for an energy-efficient deep neural network training system//Proceedings of the 29th International European Conference on Parallel and Distributed Computing. Limassol, Cyprus, 2023: 304-319

[128] Lee Y, Kim H, Rhu M. Presto: An in-storage data preprocessing system for training recommendation models//Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture. Buenos Aires, Argentina, 2024: 340-353

[129] Wang M, Waldspurger G, Sundararaman S. A selective preprocessing offloading framework for reducing data traffic in dl training//Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems. Santa Clara, USA, 2024: 63-70

[130] Huang Y, Cheng Y, Bapna A, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism//Proceedings of the 33rd Annual Conference on Neural Information Processing Systems. Vancouver, Canada, 2019: 103-112

[131] Narayanan D, Harlap A, Phanishayee A, et al. Pipedream: Generalized pipeline parallelism for dnn training//Proceedings of the 27th ACM Symposium on Operating Systems Principles. Huntsville, Canada, 2019: 1-15

[132] Shoeybi M, Patwary M, Puri R, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019

[133] Rajbhandari S, Rasley J, Ruwase O, et al. Zero: Memory optimizations toward training trillion parameter models//Proceedings of the 32nd International Conference for High Performance Computing, Networking, Storage and Analysis. Virtual Event, 2020: 1-16

[134] Kwon W, Li Z, Zhuang S, et al. Efficient memory management for large language model serving with pagedattention//Proceedings of the 29th ACM Symposium on Operating Systems Principles. Koblenz, Germany, 2023: 611-626

[135] Prabhu R, Nayak A, Mohan J, et al. vattention: Dynamic memory management for serving llms without pagedattention. arXiv preprint arXiv:2405.04437, 2024

[136] Lin B, Zhang C, Peng T, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. arXiv preprint arXiv:2401.02669, 2024

[137] Rhu M, Gimelshein N, Clemons J, et al. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design//Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture. Taipei, China, 2016: 1-13

[138] Wang L, Ye J, Zhao Y, et al. Superneurons: Dynamic gpu memory management for training deep neural networks//Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming. Vienna, Austria, 2018: 41-53

[139] Huang C C, Jin G, Li J. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping//Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne, Switzerland, 2020: 1341-1355

[140] Peng X, Shi X, Dai H, et al. Capuchin: Tensor-based gpu memory management for deep learning//Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne, Switzerland, 2020: 891-905

[141] Chen C, Wang Y, Yang J, et al. Openembedding: A distributed parameter server for deep learning recommendation models using persistent memory//Proceedings of the 39th IEEE International Conference on Data Engineering. Anaheim, USA, 2023: 2976-2987

[142] Xie M, Lu Y, Wang Q, et al. Petps: Supporting huge embedding models with persistent memory. Proceedings of the VLDB Endowment, 2023, 16(5): 1013-1022

[143] Kim S, Jin Y, Sohn G, et al. Behemoth: A flash-centric training accelerator for extreme-scale dnns//Proceedings of the 19th USENIX Conference on File and Storage Technologies. Virtual, 2021: 371-385

[144] Rajbhandari S, Ruwase O, Rasley J, et al. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning//Proceedings of the 33rd International Conference for High Performance Computing, Networking, Storage and Analysis. St. Louis, USA, 2021: 1-14

[145] Sun X, Wang W, Qiu S, et al. Stronghold: Fast and affordable billion-scale deep learning model training//Proceedings of the 34th International Conference for High Performance

Computing, Networking, Storage and Analysis. Dallas, USA, 2022: 1-17

[146] Wang Z, Sim J, Lim E, et al. Enabling efficient large-scale deep learning training with cache coherent disaggregated memory systems//Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture. Seoul, Republic of Korea, 2022: 126-140

[147] Liu H, Zheng L, Huang Y, et al. Enabling efficient large recommendation model training with near cxl memory processing//Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture. Buenos Aires, Argentina, 2024: 382-395

[148] Park S S, Kim K S, So J, et al. An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models//Proceedings of the 30th IEEE International Symposium on High-Performance Computer Architecture. Edinburgh, UK, 2024: 970-982

[149] Yun S, Nam H, Kyung K, et al. Clay: Cxl-based scalable ndp architecture accelerating embedding layers//Proceedings of the 38th ACM International Conference on Supercomputing. Kyoto, Japan, 2024: 338-351

[150] Wei J, Zhang X, Wang L, et al. Fastensor: Optimise the tensor I/O path from ssd to gpu for deep learning training. ACM Transactions on Architecture and Code Optimization, 2023, 20(4): 1-25

[151] Jeong J, Baek S, Ahn J. Fast and efficient model serving using multi-gpus with direct-host-access//Proceedings of the 18th European Conference on Computer Systems. Rome, Italy, 2023: 249-265

[152] Xie Min-Hui, Lu You-You, Feng Yang-Yang, et al. A recommendation model inference system based on GPU direct storage access architecture. Journal of Computer Research and Development, 2024, 61(3): 589-599 (in Chinese)
(谢旻晖, 陆游游, 冯杨洋, 等. 基于 GPU 直访存储架构的推荐模型预估系统. 计算机研究与发展, 2024, 61(3): 589-599)

[153] HeydariGorji A, Torabzadehkashi M, Rezaei S, et al. Stannis: Low-power acceleration of dnn training using computational storage devices//Proceedings of the 57th ACM/IEEE Design Automation Conference. San Francisco, USA, 2020: 1-6

[154] Kim J, Kang M, Han Y, et al. Optimstore: In-storage optimization of large scale dnns with on-die processing//Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture. Montreal, Canada, 2023: 611-623

[155] Jang H, Song J, Jung J, et al. Smart-infinity: Fast large language model training using near-storage processing on a real system//Proceedings of the 30th IEEE International Symposium on High-Performance Computer Architecture. Edinburgh, UK, 2024: 345-360

[156] Wang Y, Pan X, An Y, et al. Beacongnn: Large-scale gnn acceleration with out-of-order streaming in-storage computing//Proceedings of the 30th IEEE International Symposium on High-Performance Computer Architecture. Edinburgh, UK, 2024: 330-344

[157] Micikevicius P, Narang S, Alben J, et al. Mixed precision training//Proceedings of the 6th International Conference on Learning Representations. Vancouver, Canada, 2018: 1-12

[158] Han S, Mao H, Dally W J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149, 2015

[159] Liu Z, Desai A, Liao F, et al. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time//Proceedings of the 37th Annual Conference on Neural Information Processing Systems. New Orleans, USA, 2024: 52342-52364

[160] Zhang Z, Sheng Y, Zhou T, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models//Proceedings of the 37th Annual Conference on Neural Information Processing Systems. New Orleans, USA, 2023: 34661-34710

[161] Xiao G, Tian Y, Chen B, et al. Efficient streaming language models with attention sinks//Proceedings of the 12th International Conference on Learning Representations. Vienna, Austria, 2024: 1-21

[162] Ge S, Zhang Y, Liu L, et al. Model tells you what to discard: Adaptive kv cache compression for llms//Proceedings of the 12th International Conference on Learning Representations. Vienna, Austria, 2024: 1-14

[163] Chen T, Xu B, Zhang C, et al. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174, 2016

[164] Jain P, Jain A, Nrusimha A, et al. Checkmate: Breaking the memory wall with optimal tensor rematerialization//Proceedings of the 3rd Machine Learning and Systems. Austin, USA, 2020: 497-511

[165] Rhu M, O'Connor M, Chatterjee N, et al. Compressing dma engine: Leveraging activation sparsity for training deep neural networks//Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture. Vienna, Austria, 2018: 78-91

[166] Li S, Liu H, Bian Z, et al. Colossal-ai: A unified deep learning system for large-scale parallel training//Proceedings of the 52nd International Conference on Parallel Processing. Salt Lake City, USA, 2023: 766-775

[167] Xu Q, You Y. An efficient 2d method for training super-large deep learning models//Proceedings of the 37th IEEE International Parallel and Distributed Processing Symposium. St. Petersburg, USA, 2023: 222-232

[168] Wang B, Xu Q, Bian Z, et al. Tesseract: Parallelize the tensor parallelism efficiently//Proceedings of the 51st International Conference on Parallel Processing. Bordeaux, France, 2022: 1-11

[169] Bian Z, Xu Q, Wang B, et al. Maximizing parallelism in

distributed training for huge neural networks. arXiv preprint arXiv:2105.14450, 2021

[170] Korthikanti V A, Casper J, Lym S, et al. Reducing activation recomputation in large transformer models//Proceedings of the 6th Machine Learning and Systems. Miami, USA, 2023: 341-353

[171] Pope R, Douglas S, Chowdhery A, et al. Efficiently scaling transformer inference//Proceedings of the 6th Machine Learning and Systems. Miami, USA, 2023: 606-624

[172] Gao He-Ran, Wu Heng, Xu Yuan-Jia, et al. Survey on memory swapping mechanism for deep learning training. Journal of Software, 2023, 34(12): 5862-5886 (in Chinese) (高赫然, 吴恒, 许源佳, 等. 面向深度学习训练的内存交换机制综述. 软件学报, 2023, 34(12): 5862-5886)

[173] Ren J, Rajbhandari S, Aminabadi R Y, et al. Zero-offload: Democratizing billion-scale model training//Proceedings of the 2021 USENIX Annual Technical Conference. Virtual, 2021: 551-564

[174] Li Y, Phanishayee A, Murray D, et al. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. Proceedings of the VLDB Endowment, 2022, 15(11): 2747-2760

[175] Agarwal S, Yan C, Zhang Z, et al. Bagpipe: Accelerating deep recommendation model training//Proceedings of the 29th ACM Symposium on Operating Systems Principles. Koblenz, Germany, 2023: 348-363

[176] Feng Y, Xie M, Tian Z, et al. Mobius: Fine tuning large-scale models on commodity gpu servers//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Vancouver, Canada, 2023: 489-501

[177] Xie M, Lu Y, Lin J, et al. Fleche: An efficient gpu embedding cache for personalized recommendations//Proceedings of the 17th European Conference on Computer Systems. Rennes, France, 2022: 402-416

[178] Hwang R, Wei J, Cao S, et al. Pre-gated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference//Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture. Buenos Aires, Argentina, 2024: 1018-1031

[179] Lee W, Lee J, Seo J, et al. Infinigen: Efficient generative inference of large language models with dynamic kv cache management//Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation. Santa Clara, USA, 2024: 155-172

[180] Song Y, Mi Z, Xie H, et al. Powerinfer: Fast large language model serving with a consumer-grade gpu//Proceedings of the 30th ACM Symposium on Operating Systems Principles. Austin, USA, 2024: 590-606

[181] Song X, Zhang Y, Chen R, et al. Ugache: A unified gpu cache for embedding-based deep learning//Proceedings of the 29th ACM Symposium on Operating Systems Principles. Koblenz, Germany, 2023: 627-641

[182] Eisenman A, Naumov M, Gardner D, et al. Bandana: Using non-volatile memory for storing deep learning models//Proceedings of the 2nd Machine Learning and Systems. Palo Alto, USA, 2019: 40-52

[183] Kurniawan D H, Wang R, Zulkifli K S, et al. Evstore: Storage and caching capabilities for scaling embedding tables in deep recommendation systems//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Vancouver, Canada, 2023: 281-294

[184] Jin Y, Kim S, Ham T J, et al. Architecting a flash-based storage system for low-cost inference of extreme-scale dnns. IEEE Transactions on Computers, 2022, 71(12): 3153-3164

[185] Gao B, He Z, Sharma P, et al. Cost-efficient large language model serving for multi-turn conversations with cachedattention//Proceedings of the 2024 USENIX Annual Technical Conference. Santa Clara, USA, 2024: 111-126

[186] Alizadeh K, Mirzadeh I, Belenko D, et al. Llm in a flash: Efficient large language model inference with limited memory. arXiv preprint arXiv:2312.11514, 2023

[187] Xue Z, Song Y, Mi Z, et al. Powerinfer-2: Fast large language model inference on a smartphone. arXiv preprint arXiv:2406.06282, 2024

[188] Niu F, Yue J, Shen J, et al. Flashgnn: An in-ssd accelerator for gnn training//Proceedings of the 30th IEEE International Symposium on High-Performance Computer Architecture. Edinburgh, UK, 2024: 361-378

[189] Li S, Wang Y, Hanson E, et al. Ndrec: A near-data processing system for training large-scale recommendation models. IEEE Transactions on Computers, 2024, 73(5): 1248-1261

[190] Li C, Wang Y, Liu C, et al. Glist: Towards in-storage graph learning//Proceedings of the 2021 USENIX Annual Technical Conference. Virtual, 2021: 225-238

[191] Kwon M, Gouk D, Lee S, et al. Hardware/software co-programmable framework for computational ssds to accelerate deep learning service on large-scale graphs//Proceedings of the 20th USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2022: 147-164

[192] Wilkening M, Gupta U, Hsia S, et al. Recssd: Near data processing for solid state drive based recommendation inference//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Virtual, 2021: 717-729

[193] Sun X, Wan H, Li Q, et al. Rm-ssd: In-storage computing for large-scale recommendation inference//Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture. Seoul, Republic of Korea, 2022: 1056-1070

[194] Mailthody V S, Qureshi Z, Liang W, et al. Deepstore: In-storage acceleration for intelligent queries//Proceedings of

the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. Columbus, USA, 2019: 224-238

[195] Kim J, Oh S, Kung J, et al. Ndpipe: Exploiting near-data processing for scalable inference and continuous training in photo storage//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. La Jolla, USA, 2024: 689-707

[196] Gao Han, Tian Yu-Long, Xu Feng-Yuan, et al. Survey of deep learning model compression and acceleration. Journal of Software, 2021, 32(1): 68-92 (in Chinese)
(高晗, 田育龙, 许封元, 等. 深度学习模型压缩与加速综述. 软件学报, 2021, 32(1): 68-92)

[197] Li Y, Huang Y, Yang B, et al. Snapkv: Llm knows what you are looking for before generation. arXiv preprint arXiv: 2404.14469, 2024

[198] Wu H, Tu K. Layer-condensed kv cache for efficient inference of large language models. arXiv preprint arXiv: 2405.10637, 2024

[199] Liu A, Liu J, Pan Z, et al. Minicache: Kv cache compression in depth dimension for large language models. arXiv preprint arXiv:2405.14366, 2024

[200] Cai Z, Zhang Y, Gao B, et al. Pyramidkv: Dynamic KV cache compression based on pyramidal information funneling. arXiv preprint arXiv:2406.02069, 2024

[201] Wan Z, Wu X, Zhang Y, et al. D2o: Dynamic discriminative operations for efficient generative inference of large language models. arXiv preprint arXiv:2406.13035, 2024

[202] Hooper C, Kim S, Mohammadzadeh H, et al. Kvquant: Towards 10 million context length llm inference with KV cache quantization. arXiv preprint arXiv:2401.18079, 2024

[203] Liu Z, Yuan J, Jin H, et al. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. arXiv preprint arXiv: 2402.02750, 2024

[204] Yue Y, Yuan Z, Duanmu H, et al. Wkvquant: Quantizing weight and key/value cache for large language models gains more. arXiv preprint arXiv:2402.12065, 2024

[205] Yang J Y, Kim B, Bae J, et al. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. arXiv preprint arXiv:2402.18096, 2024

[206] Dong S, Cheng W, Qin J, et al. QAQ: Quality adaptive quantization for LLM KV cache. arXiv preprint arXiv: 2403.04643, 2024

[207] Kang H, Zhang Q, Kundu S, et al. Gear: An efficient kv cache compression recipe for near-lossless generative inference of LLM. arXiv preprint arXiv:2403.05527, 2024

[208] Sheng Y, Zheng L, Yuan B, et al. Flexgen: High-throughput generative inference of large language models with a single gpu//Proceedings of the 40th International Conference on Machine Learning. Honolulu, USA, 2023: 31094-31116

[209] Zhao Y, Wu D, Wang J. Alisa: Accelerating large language model inference via sparsity-aware KV caching//Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture. Buenos Aires, Argentina, 2024: 1005-1017

[210] Evans R D, Liu L, Aamodt T M. Jpeg-act: Accelerating deep learning via transform-based lossy compression//Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture. Valencia, Spain, 2020: 860-873

[211] Jin S, Zhang C, Jiang X, et al. Comet: A novel memory-efficient deep learning training framework by using error-bounded lossy compression. Proceedings of the VLDB Endowment, 2021, 15(4): 886-899

[212] Chen J, Zheng L, Yao Z, et al. Actnn: Reducing training memory footprint via 2-bit activation compressed training//Proceedings of the 38th International Conference on Machine Learning. Virtual, 2021: 1803-1813

[213] Liu Z, Zhou K, Yang F, et al. Exact: Scalable graph neural networks training via extreme activation compression//Proceedings of the 10th International Conference on Learning Representations. Virtual, 2022: 1-32

[214] Kurtz M, Kopinsky J, Gelashvili R, et al. Inducing and exploiting activation sparsity for fast inference on deep neural networks//Proceedings of the 37th International Conference on Machine Learning. Vienna, Austria, 2020: 5533-5543

[215] Jiang S, Huang T W, Yu B, et al. Snicit: Accelerating sparse neural network inference via compression at inference time on gpu//Proceedings of the 52nd International Conference on Parallel Processing. Salt Lake City, USA, 2023: 51-61

[216] Jeon M, Venkataraman S, Phanishayee A, et al. Analysis of large-scale multi-tenant GPU clusters for dnn training workloads//Proceedings of the 2019 USENIX Annual Technical Conference. Renton, USA, 2019: 947-960

[217] Zhang S, Roller S, Goyal N, et al. Opt: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068, 2022

[218] Rojas E, Kahira A N, Meneses E, et al. A study of checkpointing in large scale training of deep neural networks. arXiv preprint arXiv:2012.00825, 2020

[219] Mohan J, Phanishayee A, Chidambaram V. Checkfreq: Frequent, fine-grained DNN checkpointing//Proceedings of the 19th USENIX Conference on File and Storage Technologies. Virtual, 2021: 203-216

[220] Wang Z, Jia Z, Zheng S, et al. Gemini: Fast failure recovery in distributed training with in-memory checkpoints//Proceedings of the 29th ACM Symposium on Operating Systems Principles. Koblenz, Germany, 2023: 364-381

[221] Chen M, Hua Y, Bai R, et al. A cost-efficient failure-tolerant scheme for distributed DNN training//Proceedings of the 41st IEEE International Conference on Computer Design. Washington, USA, 2023: 150-157

[222] Maurya A, Underwood R, Rafique M M, et al. Datastates-

LLM: Lazy asynchronous checkpointing for large language models//Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing. Pisa, Italy, 2024: 227-239

[223] Li Y, Wu T, Li G, et al. Portus: Efficient dnn checkpointing to persistent memory with zero-copy//Proceedings of the 44th IEEE International Conference on Distributed Computing Systems. Jersey City, USA, 2024: 59-70

[224] Wang G, Ruwase O, Xie B, et al. Fastpersist: Accelerating model checkpointing in deep learning. arXiv preprint arXiv: 2406.13768, 2024

[225] Agrawal A, Reddy S, Bhattamishra S, et al. Inshrinkerator: Compressing deep learning training checkpoints via dynamic quantization//Proceedings of the 15th ACM Symposium on Cloud Computing. Redmond, USA, 2024: 1012-1031

[226] Xie P, Kim J K, Ho Q, et al. Orpheus: Efficient distributed machine learning via system and algorithm co-design//Proceedings of the 9th ACM Symposium on Cloud Computing. Carlsbad, USA, 2018: 1-13

[227] Chen Y, Liu Z, Ren B, et al. On efficient constructions of checkpoints//Proceedings of the 37th International Conference on Machine Learning. Virtual, 2020: 1627-1636

[228] Hu Z, Zou X, Xia W, et al. Delta-dnn: Efficiently compressing deep neural networks via exploiting floats similarity//Proceedings of the 49th International Conference on Parallel Processing. Edmonton, Canada, 2020: 1-12

[229] Jin H, Wu D, Zhang S, et al. Design of a quantization-based dnn delta compression framework for model snapshots and federated learning. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(3): 923-937

[230] Li W, Chen X, Shu H, et al. Excp: Extreme llm checkpoint compression via weight-momentum joint shrinking//Proceedings of the 41st International Conference on Machine Learning. Vienna, Austria, 2024: 1-14

[231] Gupta T, Krishnan S, Kumar R, et al. Just-in-time checkpointing: Low cost error recovery from deep learning training failures//Proceedings of the 19th European Conference on Computer Systems. Athens, Greece, 2024: 1110-1125

[232] Maeng K, Bharuka S, Gao I, et al. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery//Proceedings of the 4th Machine Learning and Systems. San Jose, USA, 2021: 637-651

[233] Qiao A, Aragam B, Zhang B, et al. Fault tolerance in iterative-convergent machine learning//Proceedings of the 36th International Conference on Machine Learning. Long Beach, USA, 2019: 5220-5230

[234] Zhong Y, Sheng G, Liu J, et al. Swift: Expedited failure recovery for large-scale dnn training. IEEE Transactions on Parallel and Distributed Systems, 2024, 35(9): 1644-1656

[235] Jang I, Yang Z, Zhang Z, et al. Oobleck: Resilient distributed training of large models using pipeline templates//Proceedings of the 29th ACM Symposium on Operating Systems Principles. Koblenz, Germany, 2023: 382-395

[236] Zhang T, Liu K, Kosaian J, et al. Efficient fault tolerance for recommendation model training via erasure coding. Proceedings of the VLDB Endowment, 2023, 16(11): 3137-3150

[237] Thorpe J, Zhao P, Eyolfson J, et al. Bamboo: Making preemptible instances resilient for affordable training of large dnns//Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation. Boston, USA, 2023: 497-513

[238] Miao H, Li A, Davis L S, et al. Towards unified data and lifecycle management for deep learning//Proceedings of the 33rd IEEE International Conference on Data Engineering. San Diego, USA, 2017: 571-582

[239] Ross R B, Amvrosiadis G, Carns P, et al. Mochi: Composing data services for high-performance computing environments. Journal of Computer Science and Technology, 2020, 35(1): 121-144

[240] Madhyastha M, Underwood R, Burns R, et al. Dstore: A lightweight scalable learning model repository with fine-grained tensor-level access//Proceedings of the 37th International Conference on Supercomputing. Orlando, USA, 2023: 133-143

[241] Underwood R, Madhyastha M, Burns R, et al. Evostore: Towards scalable storage of evolving learning models//Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing. Pisa, Italy, 2024: 148-159

[242] Fu Y, Xue L, Huang Y, et al. Serverlessllm: Low-latency serverless inference for large language models//Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation. Santa Clara, USA, 2024: 135-153

[243] Kindratenko V, Mu D, Zhan Y, et al. Hal: Computer system for scalable deep learning//Proceedings of the Practice and Experience in Advanced Research Computing. Portland, USA, 2020: 41-48

[244] Devarajan H, Zheng H, Kougkas A, et al. Dlio: A data-centric benchmark for scientific deep learning applications//Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing. Melbourne, Australia, 2021: 81-91

[245] Ofeidis I, Kiedanski D, Tassiulas L. An overview of the data-loader landscape: Comparative performance analysis. arXiv preprint arXiv:2209.13705, 2022

[246] Carns P, Harms K, Allcock W, et al. Understanding and improving computational science storage access through continuous characterization. ACM Transactions on Storage, 2011, 7(3): 1-26

[247] Chien S W D, Podobas A, Peng I B, et al. tf-darshan: Understanding fine-grained I/O performance in machine learning workloads//Proceedings of the 22nd IEEE International Conference on Cluster Computing. Kobe, Japan, 2020: 359-370

[248] Devarajan H, Zheng H, Sun X H, et al. Understanding I/O behavior of scientific deep learning applications in hpc systems//Proceedings of the 32nd International Conference for High Performance Computing, Networking, Storage and Analysis. Virtual, 2020: 1-3

[249] Peng D, Fu Z, Wang J. Pocketllm: Enabling on-device fine-tuning for personalized LLMs. arXiv preprint arXiv: 2407.01031, 2024

HE Gong-Shan, Ph. D. candidate. His research interests include storage for AI, file system, and distributed storage.

ZHAO Chuan-Lei, Ph. D. candidate. His research interests include parallel processing, GPU computing, and file system.

JIANG Jin-Hu, Ph. D., senior engineer. His research interests include computer architecture, operating system, and distributed storage.

ZHANG Wei-Hua, Ph. D., professor, Ph. D. supervisor. His research interests include compilers optimization, computer architecture, parallelization, and systems software.

CHEN Zuo-Ning, professor, Ph. D. supervisor, Academician of Chinese Academy of Engineering. Her research interests include storage system, operating system, and information security.

Background

In recent years, artificial intelligence (AI) technology, represented by deep learning (DL) and large models, has undergone rapid development. Generally speaking, deep learning has two modes: training and inference. During the training phase, the model learns the parameters from the training data. The training dataset is shuffled before each epoch. Each sample is processed exactly once per epoch. A training job typically runs for multiple epochs until the model converges. This phase involves data loading (i. e., loading data samples from storage), data preprocessing (i. e., transforming data samples into tensors), and model training (i. e., learning the parameters from data samples). Among them, data loading and data preprocessing are collectively referred to as data preparation. During the inference phase, the trained model is deployed to make predictions on new data. This phase mainly involves data preprocessing and model inference.

As accelerators such as GPUs continue to advance in speed, the data storage has become one of the main bottlenecks in DL training and inference. To be specific, the data storage faces several significant challenges. First, the size of datasets is growing rapidly, making it impractical to cache them entirely in memory. Second, datasets primarily consist of small files and training jobs read these files randomly from the training set during each epoch. On the one hand, traditional storage systems are optimized for sequentially handling large files. On the other hand, this access pattern can cause cache thrashing when using the existing cache replacement policies, such as LRU (Least Recently Used). Third, the bandwidth of storage devices is improving at a slower pace. As a result, the gap between I/O and compute is widening. Fourth, model states, such as model parameters and intermediate data, become huge, often exceeding the memory capacity of accelerators like GPUs, known as the memory capacity wall problem. Finally, training jobs are frequently interrupted due to failures. For fault tolerance, training jobs typically perform checkpointing operation to save the latest model states, but this incurs a significant performance overhead.

To address these problems, storage for AI, especially DL, has emerged as a hot research area, receiving extensive attention from both academia and industry. This paper provides a comprehensive summary of these research efforts, divided into three categories: data loading optimization, data preprocessing optimization, and model computing optimization. Furthermore, this paper discusses the limitations of the existing storage optimizations for deep learning and proposes future research directions.

This paper is supported by the National Key Research and Development Program of China (Grant No. 2023YFB4502703).