软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007438] [CSTR: 32375.14.jos.007438]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

大语言模型预训练系统关键技术综述*

高彦杰[1], 陈跃国[2]

[1](中国人民大学 信息学院, 北京 100872)

[2](数据工程与知识工程教育部重点实验室 (中国人民大学), 北京 100872)

通信作者: 陈跃国, E-mail: chenyueguo@ruc.edu.cn

摘　要: 在人工智能时代, 如何高效地完成大语言模型的预训练, 以满足其在扩展性、性能与稳定性方面的需求, 是亟需解决的重要问题. 大语言模型系统充分利用加速器和高速网卡进行并行张量计算和通信, 极大地提高了模型训练的性能, 这一进展伴随着一系列尚待解决的系统设计问题. 首先, 在分析大语言模型预训练过程的基础上, 介绍了其训练流程与负载特点. 其次, 从预训练系统的扩展性、性能和可靠性角度出发, 分别介绍了各类系统技术的分类、原理、研究现状及热点问题. 最后, 从总体层面深入分析了大型语言预训练系统面临的挑战, 并展望了其未来的发展前景.

关键词: 人工智能; 大语言模型; 大语言模型预训练系统

中图法分类号: TP18

中文引用格式: 高彦杰, 陈跃国. 大语言模型预训练系统关键技术综述. 软件学报. http://www.jos.org.cn/1000-9825/7438.htm

英文引用格式: Gao YJ, Chen YG. Survey on Key Technologies for Large Language Model Pre-training Systems. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7438.htm

Survey on Key Technologies for Large Language Model Pre-training Systems

GAO Yan-Jie[1], CHEN Yue-Guo[2]

[1](School of Information, Renmin University of China, Beijing 100872, China)

[2](Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), Beijing 100872, China)

Abstract: In the era of artificial intelligence, efficiently completing the pre-training of large language models to meet requirements for scalability, performance, and stability presents a critical challenge. These systems leverage accelerators and high-speed network interfaces to execute parallel tensor computations and communications, significantly enhancing training efficiency. However, these advancements bring a series of unresolved system design challenges. Based on an analysis of the pre-training process, this study first outlines the training procedures and workload characteristics of large language models. It then reviews system technologies from the perspectives of scalability, performance, and reliability, covering their classifications, underlying principles, current research progress, and key challenges. Finally, this study provides an in-depth analysis of the broader challenges facing large language model pre-training systems and discusses potential directions for future development.

Key words: artificial intelligence; large language model (LLM); large language model pre-training system

随着大语言模型 (large language model, LLM) 时代的到来, 模型的规模从百亿 (10B) 参数扩大到千亿 (170B) 级别. 这些模型能够支持多种任务, 包括传统的对话、摘要生成、机器翻译和程序合成. OpenAI ChatGPT、Google Bard、百度文心一言等代表这一潮流的大语言模型应用, 其用户数量也在迅速增长. 然而, 随着模型规模和用户数量的迅速增长, 模型预训练过程中存在的性能、扩展性和稳定性等方面的问题也日益凸显. 大语言模型所带来的超大规模参数和对高效计算的需求, 与传统以单 GPU 或单机多 GPU 为中心的计算模式之间存在巨大矛

* 基金项目: 国家自然科学基金 (62272466, U24A20233); 中国人民大学国家治理大数据和人工智能创新平台
收稿时间: 2024-03-04; 修改时间: 2024-08-02; 采用时间: 2025-03-25; jos 在线出版时间: 2025-10-15

盾. 这使得传统的计算架构和系统设计难以满足当前大规模语言模型训练的需求. 在模型结构方面, 大语言模型逐渐以 Transformer [1]为核心结构. 例如, GPT[2]和 BERT[3]模型通过堆叠多层 Transformer 构建, 推动系统在处理计算负载时逐步优化对 Transformer 结构及其算子的支持. 在作业模式和实验管理方面, 深度学习负载的特征发生了显著变化. 过去, 训练过程通常依赖于自动化机器学习和神经网络架构搜索, 以探索多样的模型超参数与结构配置. 而如今, 这一趋势已转向以通用的大语言模型预训练结构为核心, 其特点包括: 1) Henighan 等人[4]指出, 模型结构相对稳定, 超参数和结构的变化对学习性能影响有限; 2) 单次训练所需时间较长 (数天至数月), 资源消耗巨大 (需使用数百、数千, 甚至上万个 GPU); 3) 在训练过程中, 常常面临显存溢出、执行时间过长, 以及由此引发的软硬件故障等问题.

目前, 在大语言模型系统研究领域尚缺乏系统性的综述. 本文旨在介绍和分析大语言模型预训练系统的技术特点、分类、研究现状及热点问题, 并对其未来发展进行展望. 首先介绍和分析大语言模型的基本概念与训练流程; 其次对大语言模型预训练技术及系统进行分类, 介绍典型的开源系统, 并探讨在扩展性、性能和可靠性等系统设计问题上的关键技术方案; 最后对大语言模型系统的未来发展进行展望.

1 大语言模型的建模过程

1.1 大语言模型简介

大型语言模型在多个领域展现出了巨大潜力, 尤其是在跨广泛领域的专业知识中表现出色, 能够执行复杂的任务推理, 包括专业写作、编程及求解数学题等. 在大语言模型基础上的多款应用, 例如 ChatGPT、Bard 和文心一言等产品提供了交互式聊天界面, 实现了与用户的自然交流, 能够解答问题并完成各种任务. 大语言模型的优异性能得益于其核心架构——基于自回归的多层堆叠 Transformer. 该模型首先在大规模文本语料上进行预训练, 随后采用强化学习, 结合人类反馈, 使其更符合人类偏好. 尽管训练方法在表面上看起来较为直观, 但由于模型参数庞大, 对算力和基础设施的要求极高, 因此目前仅有少数公司和科研机构具备从零开始完整训练的能力. 已公开发布的预训练大语言模型 (如 BLOOM[5]、LLaMa[6]和 Llama 2[7]) 在性能上与 GPT-3 等闭源模型相近, 但这些模型目前仍难以替代闭源的“产品级”大语言模型, 例如 ChatGPT、Bard 和文心一言等. 这些闭源产品级大语言模型经过未公开的严格微调, 力求更好地符合人类偏好, 从而显著提升了其可用性和安全性.

如表 1 所示, 我们总结了近期具有代表性的大语言模型, 剔除了基于现有模型的微调版本. 表中记录了模型权重数量, 单位为 B (十亿), 预训练数据规模 (有的披露以标记为单位, 有的以语料为单位), 训练硬件规格和训练时间.

表 1 具有代表性的大语言模型总结

模型	权重数量 (B)	预训练数据规模	训练硬件规格	训练时间 (天)
GPT-3[2]	175	300B 标记	—	—
BLOOM[5]	176	366B 标记	384 A100-80G	105
LLaMA[6]	65	1.4T 标记	2 048 A100-80G	21
Llama 2[7]	70	2T 标记	A100-80G	35.8
GPT-NeoX-20B[8]	20	825 GB 语料	96 A100-40G	—
OPT[9]	175	180B 标记	992 A100-80G	—
GLM-130B[10]	130	400B 标记	786 A100-40G	60
Gopher[11]	280	300B 标记	—	—
LaMDA[12]	137	768B 标记	1 024 TPU-v3	57.7
GLaM[13]	1 200	280B 标记	1 024 TPU-v4	23.9
PaLM[14]	540	780B 标记	6 144 TPU-v4	—
PaLM 2[15]	—	3.6T 标记	TPU-v4	—
PanGu-α [16]	13	1.1 TB 语料	2 048 Ascend 910	—
WeLM[17]	10	300B 标记	128 A100-40G	24

表 1 具有代表性的大语言模型总结 (续)

模型	权重数量 (B)	预训练数据规模	训练硬件规格	训练时间 (天)
Flan-PaLM[18]	540	—	512 TPU-v4	1.5
MPT-7B[19]	7	1T 标记	—	—

注: —表示该模型没有披露信息

从表 1 可见, 大语言模型的参数量从十亿级到千亿级不等, 训练数据规模在数百 GB 至 TB 之间; 训练硬件使用数量从数百块到上万块加速器不等, 训练时间则跨度从几天至几个月. 如此庞大的模型规模、数据量和资源消耗, 为大语言模型预训练系统的设计带来了巨大挑战. Ma 等人[20]针对在国产高性能计算机上高效训练百万亿参数预训练模型的场景, 系统分析了高效并行策略、数据存储方案及数据精度选择所面临的挑战, 并提出了相应的解决方法. 本文将进一步分析大语言模型的训练流程, 并围绕开源大语言模型预训练系统的扩展性、性能和可靠性展开讨论.

1.2 大语言模型训练过程

大语言模型的训练方式主要延续了深度学习中语言模型的训练方法. 具体来说, 训练算法主要分为遮罩语言模型 (masked language model, MLM) 和自回归语言模型 (autoregressive language model, ALM) 两类. 主流的大语言模型多采用自回归模式进行训练.

MLM 在 BERT 系列语言模型训练中被广泛使用. 其原理是随机遮蔽输入序列中的一部分词汇, 将其替换为特殊的“[MASK]”标记, 模型则需通过上下文中的其他词汇来预测这些被遮蔽的内容. 这种双向、基于上下文的训练方式, 使得 BERT 能够更好地理解词义与上下文之间的关系.

GPT 系列模型采用自回归方法进行训练, 旨在预测序列中的下一个词汇. 模型通过分析当前序列中的已有标记, 生成对下一个词汇的预测. GPT 是一种按从左到右顺序生成词汇的模型, 每个位置的词汇仅依赖其左侧的上下文信息. 这种单向、自回归的训练方式, 使得 GPT 不仅能够流畅生成文本, 还具备强大的建模能力, 可将多种任务转化为自回归预测问题来处理.

随着 GPT-3 等自回归模型取得突破性进展, 当前主流的大语言模型大多采用自回归方式进行训练. 这一训练过程可描述为在序列训练数据上, 通过最小化对下一个词或标记的预测误差来优化模型. 在该过程中, 训练数据由输入序列和相应的标签组成 (自回归训练中, 标签即为下一个需预测的标记), 模型通过学习它们之间的映射关系, 不断优化预测性能.

大语言模型的训练目标可以被形式化为一个概率建模问题, 给定一个文本序列 $x=[x^1,\ldots,x^{\mathrm{T}}]$, 自回归语言模型通过预测序列中每个词在其前文条件下出现的概率. 模型在训练过程中通过学习如何给出当前词在已有上下文条件下的概率分布, 从而提升预测下一个词的准确率. 其中, $h_\theta(x_{1:t-1})$ 表示由神经网络 (如 RNN 或 Transformer) 生成的上下文表示, 而 $\mathrm{e}(x)$ 表示 x 的嵌入.

$$\max_\theta \log_{P_\theta}(x)=\sum_{t=1}^{\mathrm{T}}\log P_\theta(x_t|x_{<t})=\sum_{t=1}^{\mathrm{T}}\log\frac{\exp\left(h_\theta(X_{1:t-1})^{\mathrm{T}}\mathrm{e}(x_t)\right)}{\sum_{x'}\exp\left(h_\theta(x_{1:t-1})^{\mathrm{T}}\mathrm{e}(x')\right)}$$

图 1 概括展示了大语言模型的训练过程. 在给定训练语料数据的情况下, 训练通常首先对数据进行分词 (tokenize), 然后对这些标记进行词嵌入 (embedding) 和位置嵌入编码. 接着, 语言模型开始训练, 模型需要在数据上多次迭代, 以逐步优化损失函数. 这个过程所需的时间和 GPU 资源消耗取决于模型尺寸和数据量的大小. 在训练完成后, 需要在未参与训练的测试数据上对模型进行评估. 评估过程中会计算困惑度 (perplexity)、准确率、BLEU 等指标, 并评估模型在安全性、伦理性与偏见等方面的表现. 上述整个流程可称为一次训练实验. 随后, 通过分析评测中发现的问题, 研究人员通常会采取调整学习率、去除某些批次数据、添加激活规范化层等策略进行优化, 并开始下一轮训练尝试. 如此循环, 直到模型在评测中达到预期效果. 整个过程可能持续数天或数月, 并需要多轮实验和不断优化.

图 1　大语言模型训练流程

如图 1 所示, 训练流程大致可以分为以下几个阶段: 首先, 根据显存容量和算法收敛性的要求, 输入一个数据批次. 例如, GPT-3 175 B 使用了一个包含 3.2 M 个标记的批次. 接着, 对输入数据进行词嵌入和位置嵌入处理, 将其转换为张量表示. 然后, 将数据输入由多层 Transformer 构成的模型中进行计算. 在完成模型中各层的前向计算后, 根据损失函数进行反向传播, 计算各个权重相对于损失的梯度. 最后, 通过这些梯度更新模型权重, 完成一次迭代. 经过多轮迭代后, 若模型性能达到预期, 训练过程即完成.

图 2 展示了一个简化的计算图, 用于描述 3 层 GPT 模型的计算流程. 图中的节点代表 Transformer 层, 边则表示层之间的依赖关系. 例如, 图中的方框表示注意力层. Transformer 层按照节点上的编号和箭头所指的顺序依次执行, 从而完成推理过程.

在每一次迭代的前向传播步骤中, 生成的输出标记被反馈到模型, 损失函数计算误差后, 通过反向传播计算梯度, 并最终更新模型参数. 自回归生成每个标记的过程通过执行模型的所有层完成. 输入可以是来自客户端的提示词标记, 也可以是先前生成的输出标记. 一般来说, 将模型运行一遍定义为一个迭代. 在图 3 中显示的示例中, 整个过程包括 3 个迭代. 第 1 次迭代 (“迭代 1”) 获取所有输入标记 (“GPT 是”), 并预测下一个标记 (“语言”). 完成前向传播并获取预测输出后, 需要最小化损失函数, 然后通过梯度下降算法更新相应的权重.

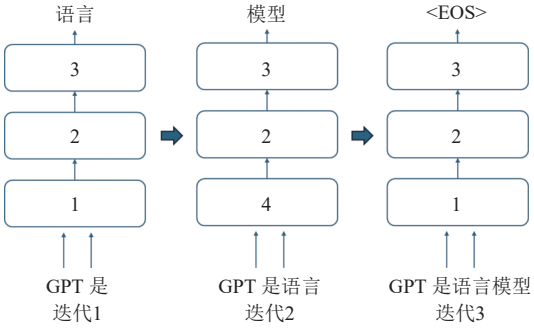

图 2　大语言模型一个请求的前向传播流程

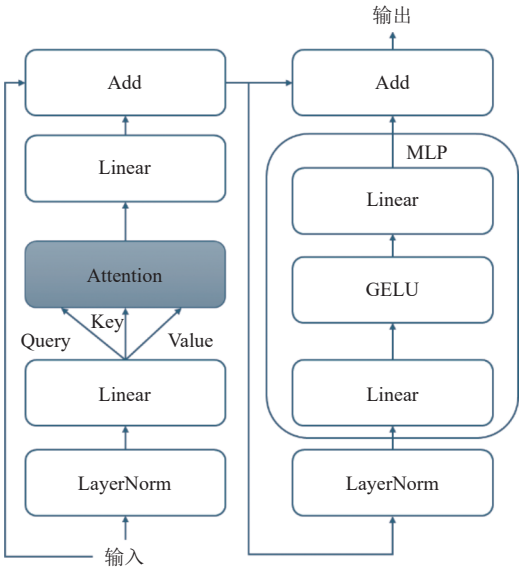

图 3　Transformer 模型结构

图 3 中的大语言模型结构通常由多个 Transformer 层堆叠而成, 其核心算子包括矩阵乘法和非线性变换. 在执行过程中, 大语言模型与输入批次数据进行大量的矩阵运算, 例如, 通用矩阵乘法等.

原始的 Transformer 模型使用了由编码器和解码器堆叠而成的 Transformer 层, 而 GPT 的架构则由一个单一的解码器层堆栈组成.图 3 展示了在 GPT 中使用的一个 Transformer 层. 在组成 Transformer 层的各项操作中, 注意力 (attention) 层是区分 Transformer 与其他模型结构的关键部分. 从高层次来看, 注意力操作通过计算每个标记的权重, 使序列中的每个标记能够关注到其他标记. 注意力层的核心计算公式为:

$$S = QK^{\mathrm{T}}, P = Softmax(S) \in \mathbb{R}^{N\times N}, O = PV \in \mathbb{R}^{N\times d}.$$

它接收由输入张量通过权重张量映射而来的 3 个输入, 即查询向量 Q(query)、键向量 K(key) 和值向量 V(value). 其中 $Q,K,V\in\mathbb{R}^{N\times d}$, N 是序列长度, d 是多注意力头的维度. 通过这些输入, 计算产生输出 $O\in\mathbb{R}^{N\times d}$. 计算查询与所有键的点积, 得到 $S=QK^{\mathrm{T}}\in\mathbb{R}^{N\times N}$. 然后对 S 应用 $Softmax$ 操作, 得到 $P=Softmax(S)\in\mathbb{R}^{N\times N}$, 最后根据 P 与 V 进行点积, 产生输出 $O=PV\in\mathbb{R}^{N\times d}$. 除了注意力层外, 多层感知器 (multilayer perceptron, MLP) 是另一个重要的结构, 其中, 线性层 (linear) 通过权重矩阵 W 对输入张量 I 进行矩阵乘法, 产生输出 O:

$$O=IW.$$

其他层, 如层归一化 (layer normalization)、激活函数 GELU、残差连接 (residual connection, 图中为 Add) 等, 会穿插在层间使用. 在第 3.3.1 节介绍的模型层性能优化, 主要分为两部分: 一部分针对注意力层, 优化方法包括访存优化、稀疏化以及针对长上下文的优化; 另一部分针对 MLP 层, 主要通过多专家系统 (MoE) 等方式进行优化.

2 大语言模型系统技术研究框架

大语言模型预训练作业的完整生命周期可以包括训练程序开发、在平台上的提交与部署、模型训练、模型验证与反馈几个步骤. 总体而言, 这是一个迭代性的实验过程. 根据实验结果和监控反馈, 开发者可能会进一步优化模型效果, 或因存在缺陷而需要进行调试与修复. 在整个流程中, 训练过程是最关键且最复杂的环节, 可进一步细分为以下步骤: 模型加载、数据加载、数据预处理、前向传播、反向传播、梯度更新. 同时, 为了调试并防止因平台故障造成模型权重丢失, 系统会定期进行检查点备份. 在整个流程的各个阶段, 仍存在诸多系统设计问题亟待解决, 以有效支撑大语言模型的训练需求. 例如, 在模型加载阶段, 需要确定模型的划分方式, 并决定如何应用并行化策略进行部署 (如数据并行、模型并行、张量并行等). 在前向传播与反向传播阶段, 可应用高效的并行加速器内核, 并采用低精度数据类型实现混合精度计算, 以提升计算效率. 此外, 在训练过程中, 由于模型被划分并分布到多个 GPU 上, 需要通过通信机制完成梯度、权重或激活张量的同步与聚合, 因此依赖高效的通信方式来提升整体性能.

当前, 大语言模型的训练过程以模型为中心, 针对特定的模型结构和执行阶段进行并行化与计算优化, 但尚未形成统一的系统优化方案. 基于前文对大语言模型训练过程在系统支撑方面的需求分析, 本文提出了如图 4 所示的研究框架, 围绕大语言模型预训练系统、扩展性、性能与可靠性, 系统梳理了研究现状、面临的挑战及相应的解决方案.

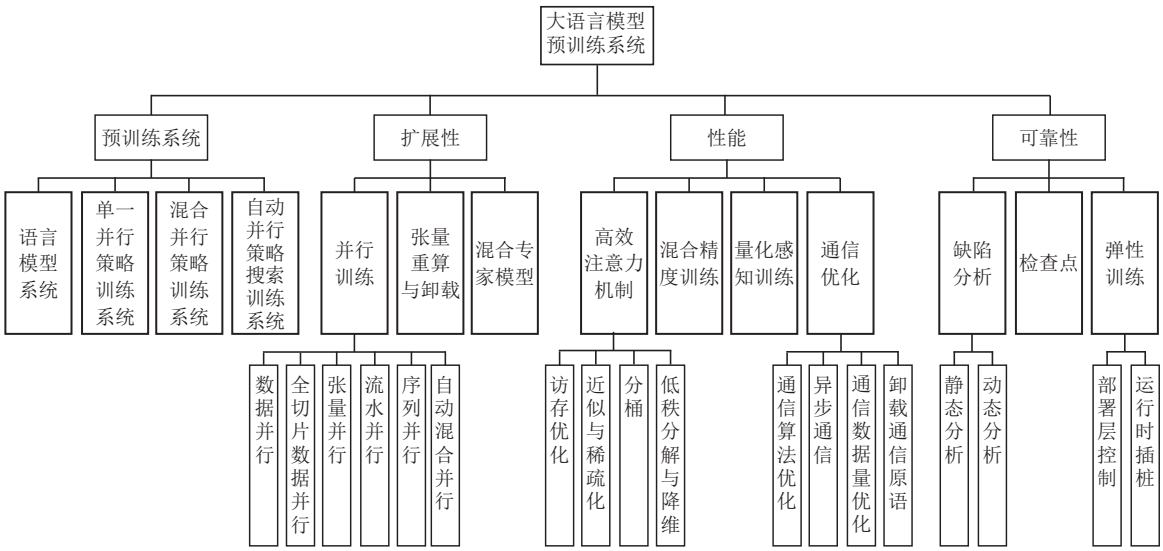

图 4 大型语言模型预训练系统研究方向分类

3 预训练系统研究及相关技术

首先, 我们将介绍具有代表性的大语言模型开源预训练系统, 对 4 类系统 (语言模型系统、单一并行策略训练系统、混合并行策略训练系统、自动并行策略搜索训练系统) 进行详细分析, 并比较它们的优缺点. 随后, 我们将深入探讨大语言模型预训练系统在扩展性、性能和可靠性方面的问题. 为了提升系统的扩展性, 当前的研究方向主要包括并行训练、张量重算与张量卸载, 以及混合专家模型. 在性能方面, 高效注意力机制、混合精度训练、量化感知训练和通信优化是缓解大语言模型训练中内存、通信与计算瓶颈的关键手段. 至于可靠性方面, 工业界和学术界目前主要关注缺陷分析、检查点机制与弹性训练.

大语言模型训练过程包括模型切片并行化、读取和嵌入数据、执行前向传播、反向传播和权重更新, 并在此过程中执行计算内核、进行通信和保存检查点. 然而, 大语言模型训练技术面临着多个挑战. 首先, 在面向多加速器 (GPU) 和分布式的部署环境中, 训练或微调过程需要以模型切片等方式进行部署, 以加速训练并防止内存溢出 (out of memory, OOM). 其次, 在训练过程中通常需要多次迭代来优化模型效果, 这涉及频繁访存、通信和保存检查点等高 I/O 开销的操作. 同时, 随着数据规模的增大, 探索既能降低空间占用和浮点运算量, 又能保证算法收敛的稀疏化技术变得至关重要. 高效注意力机制、混合精度训练等技术被提出并应用, 可以进一步提升数据读写和计算效率. 最后, 由于训练时间较长且集群中的硬件和软件容易出现故障, 系统层面需要通过检查点和弹性训练等手段, 以保障系统的可靠性. 本节将围绕大语言模型预训练系统, 对其在扩展性、性能和可靠性等方面的研究进行讨论与总结.

3.1 大语言模型预训练系统

大语言模型系统的发展可以概括为 4 个阶段. 第 1 阶段以 Transformers[21]为代表, 开始支持大语言模型及其社区. 在这个时期, 为了支持上述模型, 所构建的一些系统一般在底层调用深度学习框架 PyTorch[22]. 随着语言模型规模的不断增大, 在第 2 阶段出现了支持单一类型并行策略的框架, 如 Horovod[23]、GPipe[24]、PyTorch DDP[25]等, 它们支持数据并行或流水线并行. 随着模型规模的持续扩大, 单一并行策略已无法满足需求, 于是进入了第 3 阶段, 即设计出混合并行策略. 在这一阶段, 出现了以 Megatron-LM[26]和 DeepSpeed[27] 为代表的框架, 它们综合运用了多种并行方案, 并对主流硬件架构和大语言模型结构进行了深度优化. 混合并行方案对开发人员的要求较高, 尤其在研究人员尝试新的模型结构和硬件拓扑时, 需要系统工程师协助调优. 这也进一步催生了对自动并行策略搜索的需求. 第 4 阶段随着以 Alpa[28]为代表的自动并行策略搜索训练系统的出现. 逐渐使普通开发者能够在新的模型结构和硬件拓扑下部署最优的并行策略, 并顺利完成训练.图 5 总结了开源大语言模型预训练系统的演化时间轴. 如果已开源, 发布时间以 GitHub 第 1 个 Commit 时间为准, 未开源则以论文 ArXiv 第 1 次提交时间为准.

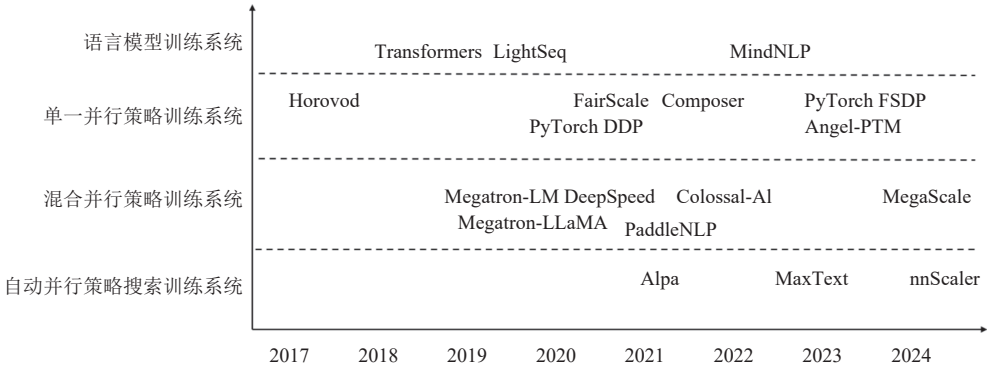

图 5 开源大型语言模型预训练系统时间轴图

当前, 大多数并行方案以库、API 或框架的形式构建在基础深度学习框架 PyTorch 和 TensorFlow[29] 之上. 由

于深度学习框架对快速演化的语言模型及结构及预处理工具支持有限, 同时其对语言模型中常用结构的并行策略和相关性能优化原生支持较少, 使得开发者难以快速使用深度学习框架直接编写高效的训练程序, 更多的面向大语言模型的框架和系统被设计、开源和广泛采用. 其中, 以 PyTorch 为后端的框架更为广泛, 例如, DeepSpeed, Megatron-LM 等系统, Google 也开源了构建于 TensorFlow 之上的 MaxText[30]. 本文将大语言模型系统根据其所在层次和支持的并行策略特点分为以下 4 类.

1) 语言模型系统. 这类系统的设计目标是支持如 GPT、BERT 等最新的语言模型结构. 在系统优化方面, 这些框架通常支持高效的计算内核实现. 其底层部署通常通过集成第 2 类、第 3 类和第 4 类系统来实现所需的并行策略.

Transformers 库由 Hugging Face 开源并提供了数千个预训练大语言模型, 也支持不同模态的任务, 包括文本、视觉和音频. Transformers 提供 API, 可快速下载和使用这些预训练模型, 并在自己的数据集上进行微调, 然后在模型中心与社区共享. 同时, 每个用于定义模型架构的 Python 模块都是完全独立的, 适合开展高效的研究实验. Transformers 由当前最流行的 3 个深度学习库 (JAX[31]、PyTorch 和 TensorFlow) 提供支持, 可实现无缝集成. 它们之间具有无缝集成. 在一个库中训练模型后, 可轻松在另一个库中加载并进行推断.

LightSeq[32] 是一个基于 CUDA 实现的高性能序列处理与生成的训练库. 它专为如 BERT、GPT、Transformer 等的语言模型的高效计算而设计, 支持机器翻译、文本生成及其他序列相关任务. 该库构建于 CUDA 官方库 cuBLAS 等之上, 并结合了专门为 Transformer 模型系列设计并优化的自定义内核函数. 除了模型组件外, LightSeq 还集成了其他辅助功能.

MindNLP[33]是华为研发的基于 MindSpore 的开源语言模型库, 包含多种常见的 NLP 模型, 如 LLaMA、GLM、RWKV 等, 帮助研究人员与开发者更加便捷高效地构建和训练模型. MindNLP 提供类似 Transformers 的接口, 其主分支与 MindSpore 主分支保持兼容. 它提供多种可配置组件, 便于自定义模型. 简洁易用的训练引擎简化了复杂的训练流程, 并支持 Trainer 和 Evaluator 接口, 方便模型的训练与评估.

2) 单一并行策略训练系统. 这类系统仅支持一种类型的并行策略, 例如 Horovod、PyTorch DDP 仅支持数据并行, GPipe 仅支持流水线并行, PyTorch FSDP[34]则仅支持 FSDP 策略. 这种实现方式能够在机制上对特定并行策略进行优化, 例如在数据并行中实现反向传播阶段的分桶、异步通信以及计算与通信的重叠.

PyTorch DDP 在模块级别实现了多进程或多机数据并行. 在执行 DDP 应用程序时, 会启动多个进程, 并为每个进程创建一个单独的 DDP 实例. DDP 利用 torch.distributed 包中的集体通信来同步梯度和缓冲区. 具体而言, DDP 为 model.parameters() 中的每个参数注册一个自动微分钩子 (autograd hook, 用于反向传播). 当计算梯度时, 该钩子会被触发, 从而启动跨进程的梯度同步操作.

Horovod 是一个支持 TensorFlow、PyTorch 等框架的分布式数据并行训练框架, 其设计为框架无关 (agnostic), 即不绑定具体框架, 其内部通过 RingAll-Reduce、张量融合通信等方式加速训练过程.

PyTorch FSDP 在 PyTorch 1.11 中发布. 其设计初衷是由于分布式数据并行 (distributed data parallel, 简称 DDP) 训练中, 每个进程拥有模型的一个副本, 造成内存冗余和浪费, 且需要模型能够放入单 GPU, 这无法支撑更大规模模型训练. FSDP 是一种优化版的数据并行, 通过在 DDP 进程之间分片模型参数、优化器状态和梯度来实现, 其不保留模型副本, 需要使用时按需从所在节点进程拷贝到本地. 在使用 FSDP 进行训练时, 与在 DDP 中跨所有工作器进行训练相比, GPU 内存占用更小. 这使得通过允许更大的模型或批次大小适应设备, 可以实现对一些非常庞大模型的训练. 但这也伴随着增加的通信量成本, 其通过内部优化, 如重叠通信和计算, 有助于减少额外增加的通信开销.

Angel-PTM[35] 是由腾讯设计开发的支持类似 ZeRO、FSDP 封层内存的大模型训练框架. 它通过页抽象实现细粒度的内存管理, 并协调计算、数据迁移和通信的统一调度方法. 它还支持通过使用 SSD 存储进行极端的模型扩展, 并实现无锁更新以解决 SSD I/O 带宽瓶颈.

FairScale[36]是由 Meta 开发的一个扩展库, 专注于高性能和大规模训练, 以 PyTorch 的扩展形式实现. 该库的设计理念基于 3 个基本原则: 可用性、模块化和性能. 首先, 它的 API 强调易于理解和使用, 使用户能够轻松地掌

握 FairScale 的功能. 其次, FairScale 注重模块化, 支持在用户的训练循环中无缝融合多个 FairScale API, 从而提高灵活性. 最后, FairScale 支持 FSDP 作为扩展大型神经网络训练操作的首选方法. 同时, 它还具有在资源受限系统中进行训练的关键功能, 包括支持激活检查点、高效模型卸载和扩展等特性. 通过这些功能, FairScale 为用户提供了强大而灵活的工具, 以满足不同训练场景的需求.

Composer[37]是由 Mosaic ML[38]设计的一个库, 已经成功用于训练 Mosaic ML 的 MPT 7B 和 MPT 30B 模型. 该库建立在 PyTorch 之上, 提供了一系列加速方法, 用户可以将其整合到其他训练框架中, 或者与 Composer 训练器一同使用以获得更优质的体验. Composer 支持 FSDP 以实现高效的并行性, 还支持弹性共享检查点以实现稳健的间歇性训练. 此外, Composer 提供了数据集流的实现, 允许用户在训练期间即时从云存储中下载数据集.

3) 混合并行策略训练系统. 这类系统综合了多种并行方案. 例如, Megatron-LM, DeepSpeed 在大语言模型的训练中综合应用了张量并行、流水线并行和数据并行方案, 并对其在 NVIDIA 的加速器和集群架构上执行进行了深度优化.

DeepSpeed 是由 Microsoft 开发的一个集成框架, 用于训练和部署大语言模型. 该框架已成功用于训练大型模型, 如 Megatron-Turing NLG 530B[39]和 BLOOM. DeepSpeed 的设计集成了 ZeRO[40], 支持 FSDP 模式的训练. 同时, ZeRO-Offload [41]优化器使得在 CPU 和 GPU 上进行训练变得更加便捷, 不受内存容量的限制. 此外, DeepSpeed 还设计了扩展模块 DeepSpeed-Chat, 以增加对聊天任务的支持, 该模块通过与 DeepSpeed 系统集成来实现来自人类反馈的强化学习 (RLHF) 技术.

Megatron-LM 是由 NVIDIA 开发的训练框架, 旨在支持 GPT 等大语言模型的训练框架. 该框架包括各种专门针对 NVIDIA GPU 的工具和优化. Megatron-LM 的核心设计思想是对模型张量操作进行分解, 并将其分布在多个 GPU 上, 以优化处理速度和内存利用率. 其支持数据并行、张量并行、流水并行等多种并行策略.

Megatron-LLaMA[42]由阿里巴巴开源发布. 为了方便基于 LLaMA 的模型训练, 并降低硬件资源占用和训练成本, 阿里巴巴发布了经过内部优化的 Megatron-LLaMA 训练框架, 该框架基于 Megatron-LM 构建. Megatron-LLaMA 提供了 LLaMA 的标准实现、高效的通信与计算并行方案, 以及多个实用工具, 包括: a) 分布式检查点的保存与恢复, 用于加速训练流程; b) 便捷的界面, 用于模型权重与 HuggingFace 格式之间的转换; c) 支持 HuggingFace Transformers 库中的 Tokenizers 模块.

Colossal-AI[43]是一个专为应对大规模分布式训练挑战而设计的框架. 它支持 LLaMA、GPT-3、BERT、PaLM 等多种模型的实现. 该框架提供了一个简化的平台, 有助于减少系统碎片化问题, 并提升训练流程的效率, 同时支持多种并行策略, 如数据并行、张量并行、流水并行和序列并行等. 这种集成方法简化了在分布式环境中进行大规模训练的流程. 此外, 该框架还集成了量化、梯度累积、混合精度等多项优化技术.

MegaScale[44]是由字节跳动开发的大语言模型训练系统, 能够高效地扩展到 1 万块以上 GPU 进行训练. 该系统支持数据并行、张量并行、流水并行和序列并行策略, 并对训练效率与稳定性的挑战进行了深入分析. MegaScale 采用全栈方法, 通过模型块与优化器设计、计算与通信的重叠、算子优化、数据流水线以及网络性能调整等手段, 实现了算法与系统的协同设计. 此外, MegaScale 还开发了诊断工具, 用于运行时分析与调优. 相关工作还分享了在故障识别与修复方面的运维经验.

PaddleNLP[45]是百度研发的、基于飞桨深度学习框架的大语言模型开发套件, 支持在多种硬件 (如 NVIDIA GPU、昆仑 XPU、昇腾 NPU、燧原 GCU 和海光 DCU 等) 上进行高效的大模型训练, 支持多种并行策略、微调、无损压缩以及高性能推理. 已支持的大语言模型系列包括 LLaMA 系列、Baichuan 系列、Bloom 系列、ChatGLM 系列、Gemma 系列、Mistral 系列、OPT 系列和 Qwen 系列.

4) 自动并行策略搜索训练系统. 这类系统支持多种并行方案, 并将其搜索过程抽象为一个优化问题, 能够自动求解并合成适用于特定硬件拓扑和模型结构的方案.

Alpa[28]既是一个用于训练和部署大规模神经网络的库, 也是一个用 Python 和 JAX 编写的大语言模型框架, 并支持在 Google Cloud TPUs 上运行. 通常情况下, MaxText 能够实现 55%–60% 的模型 FLOPs 利用率, 并可从单个主机扩展至大规模集群. 同时, 系统还利用 JAX 和 XLA 编译器的强大功能, 实现了自动优化.

MaxText 大语言模型框架用 Python 和 JAX 编写, 并支持在 Google Cloud TPUs 上运行. 通常情况下, MaxText 能够实现较高的 GPU 利用率, 并且能够从单个主机扩展到非常大的集群. 同时, 利用 JAX 和 XLA 编译器的强大功能, 实现自动优化.

nnScaler[46]是微软亚洲研究院开源的支持自动并行化大语言模型的框架, 其通过设计基本原语 (例如: op-trans、op-assign 和 op-order), 让开发者更加灵活定义并行策略搜索空间, 并支持自动并行策略和优化. 为避免搜索空间的过度膨胀, nnScaler 在构建空间时允许对这些原语施加约束.

针对上述 4 种类型的框架, 我们在表 2 中总结了它们的优势、不足和适用场景.

表 2　不同类型预训练系统对比

预训练系统类型	优势	不足	适用场景
语言模型系统	支持的模型种类丰富, 使用方式简单, API设计易于被社区采纳为标准	系统优化依赖于底层系统, 容易与底层系统产生兼容性问题	需要使用新模型. 需要定制新的模型结构
单一并行策略训练系统	可以对单一策略进行深度优化	不能利用其他并行策略进一步提升扩展性	使用简化的训练并行方案. 适合中等规模的模型, 数据并行等单一策略的扩展性和性能已经能够满足需求
混合并行策略训练系统	可以对常用的模型结构和硬件拓扑进行深入且专门的优化	硬件拓扑和模型的变化需要重新进行人工调优	模型参数量大, 需要综合使用多种并行方案以满足扩展性需求
自动并行策略搜索训练系统	通用性强. 能够自动搜索并应用适应硬件拓扑变化和模型变化的方案, 无需人工干预	无法支持定制的专有优化. 数值异常常常源于系统层的并行方案, 且难以调试	硬件拓扑或模型结构变化多样, 且模型参数量大需要多种并行策略

3.2 扩展性

随着大语言模型规模的增大, 模型更能够有效地捕捉语言的复杂性和上下文信息. 然而, 这也带来了对更大计算资源和更高扩展性的需求, 以便有效地训练和部署这些大规模模型. 当前的技术趋势, 例如并行训练、张量重算与卸载, 以及多专家混合等, 都能够有效支持模型不断增强的扩展性.

3.2.1 并行训练

目前, 大语言模型的预训练系统广泛支持多种并行优化策略. 常见的并行优化策略包括数据并行、全切片数据并行、张量并行、流水线并行以及序列并行. 在表 3 中, 我们对代表性的大语言模型系统的并行策略支持进行了综合对比, 其中√代表支持, ×代表不支持. 一些库支持多种并行方案, 并能自动选择并行策略, 例如 Alpa. 然而, 一些库由于历史原因, 在最初提出时可能仅支持部分并行化策略, 例如 GPipe. 在表 3 中, 我们总结了第 3.1 节中训

表 3　开源大语言模型预训练系统所包含的并行策略对比

预训练系统	数据并行	全切片数据并行	张量并行	流水线并行	序列并行	多并行自动搜索
Transformers	√	√	√	√	×	×
LightSeq	√	√	×	×	√	×
MindNLP	√	√	√	√	×	×
PyTorch DDP	√	×	×	×	×	×
Horovod	√	×	×	×	×	×
PyTorch FSDP	√	√	×	×	×	×
Angel-PTM	√	√	×	×	×	×
FairScale	√	√	√	√	×	×
Composer	√	√	√	√	×	×
DeepSpeed	√	√	√	√	√	×
Megatron-LM	√	×	√	√	√	×
Megatron-LLaMA	√	√	√	√	√	×
Colossal-AI	√	√	√	√	√	√
MegaScale	√	√	√	√	√	×
PaddleNLP	√	√	√	√	√	×
Alpa	√	√	√	√	×	√

表 3 开源大语言模型预训练系统所包含的并行策略对比 (续)

预训练系统	数据并行	全切片数据并行	张量并行	流水线并行	序列并行	多并行自动搜索
MaxText	√	√	√	×	√	×
nnScaler	√	√	√	√	√	√

练系统所支持的优化策略及其发布时间.

如图 6 所示, 现有的并行训练方式包含多种模式, 其中灰色表示 GPU 0 上部署的张量, 白色表示 GPU 1 上部署的张量, 不同的行列区分表示不同的行列切分, 箭头代表数据流, 矩阵乘表示发生的矩阵乘计算, X 代表输入数据, W 代表权重. 图 6 的 (a)–(d) 分别表示数据并行、全分区数据并行、张量并行和流水线并行的模型及数据分区方案.

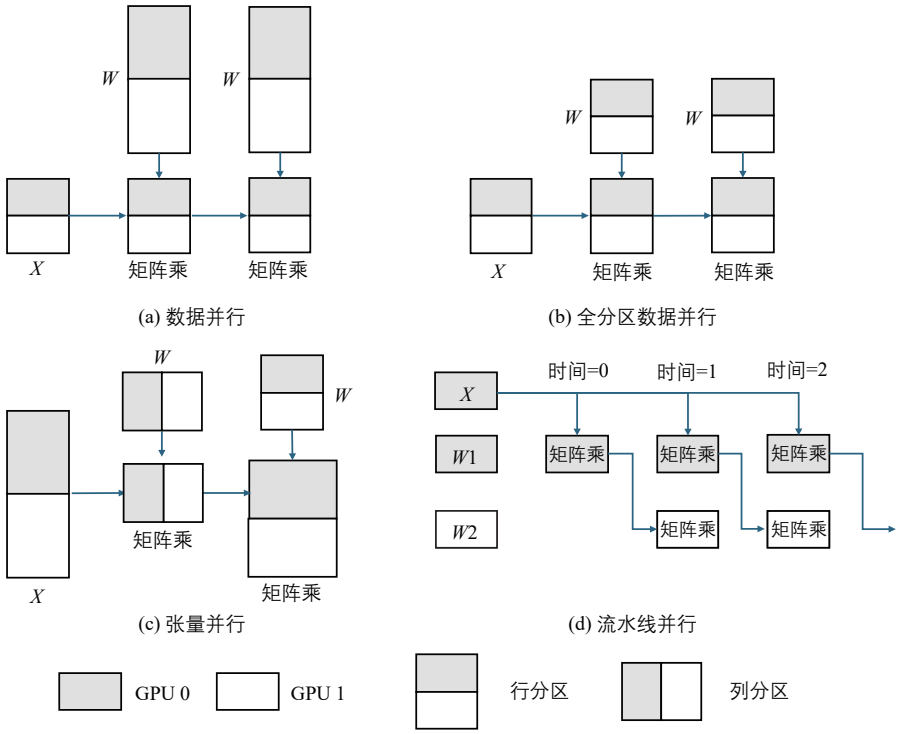

图 6 不同的并行切片模式[28]

3.2.1.1 数据并行

在深度学习模型训练中, 数据并行是一种常见的多 GPU 或分布式训练方式. 从概念上来说, 数据并行分布式训练范式非常直观. 整个训练程序会运行多个训练脚本的副本, 每个副本执行以下流程: (a) 读取数据的一部分; (b) 将其通过模型前向传播; (c) 计算模型更新 (梯度); (d) 在多个副本之间同步平均梯度; (e) 更新模型, 并不断重复以上步骤. 数据并行方案不仅用于语言模型, 还广泛应用于其他模型, 如视觉模型的训练. 由于其简单易部署且容易提升批尺寸的特点, 数据并行策略在大语言模型训练中仍被广泛沿用. 数据并行的实现通常有两种方式: 一种是框架无关方式, 可支持多种框架 (如 PyTorch、TensorFlow); 另一种是框架注册钩子方式, 针对特定框架在运行时通过注册钩子拦截执行, 以实现跨 GPU 的张量通信与聚合.

框架无关方式: Horovod 采用数据并行策略进行训练, 使用环状 All-Reduce 以减少峰值通信量, 并很好地适配底层硬件环状通信拓扑. 在每个 N 个节点中, 节点与其两个对等节点进行 $2\times(N-1)$ 次通信. 在通信过程中, 节点发送和接收数据缓冲区的数据块. 在前 $N-1$ 次迭代中, 接收到的值被加到节点缓冲区中的值上. 在接下来的 $N-1$

次迭代中, 接收到的值替换节点缓冲区中的值. Patarasuk 等人[47]曾经分析过, 如果缓冲区足够大, 环状 All-Reduce 算法是带宽最优的.

框架注册钩子方式: PyTorch DDP 通过在反向传播过程中交错通信与计算, 进一步提高性能. 它通过将梯度张量分桶, 并以批处理方式进行梯度通信, 同时注册反向传播钩子. 梯度张量准备好后进行通信和数据传递, 从而实现反向传播和梯度通信的重叠, 进一步提升性能.

在数据并行的异步通信中, 如果采用框架不可知的实现方式, 则需要控制平面来确保通信顺序以防止死锁. Horovod 提出新的协调模式, 通过高效去中心化的编排策略, 以降低控制平面的开销. 框架钩子方式一般适合只使用一种框架后端, 目前由于大部分的大语言模型基于 PyTorch 为后端, 所以目前许多大语言模型库的数据并行策略后端默认采用 PyTorch DDP 的数据并行方案, 例如 Transformers, FairSeq[48]和 TorchScale[49]等.

3.2.1.2 张量并行

提高模型容量被认为是提升模型质量的有效手段, 但当模型尺寸超出单个加速器内存限制时, 需要开发特殊算法以支持模型的训练. 在数据并行中, 要求将整个模型部署在单个 GPU 上, 当模型过大时无法实现. 为应对这一挑战, 一些系统提出了模型并行方案, 通过对模型进行切片, 降低显存占用, 以便训练更大的模型. 模型并行方案包括张量并行和流水线并行, 其中张量并行对单层进行切分, 而流水线并行在层间进行切分. Megatron-LM 使用了简单高效的层内模型并行, 也被称作张量并行方法. 其实现方式不需要编译或更改库实现, 且与流水并行方案正交不冲突. 张量并行通常有两种方案来切分通用矩阵乘中的权重矩阵, 以两层 MLP 为例, 第 1 层的通用矩阵乘之后还要进行 $GELU$ 激活函数运算, $Y = GELU(XA)$.

行切分方式[26]: 一种选择是通过行切分权重矩阵 A, 以及列切分输入矩阵 X. 行切分权重矩阵方式为 $X = [X_1, X_2], A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$, 这种切分方式计算公式为 $Y = GELU(X_1A_1 + X_2A_2)$. 由于 $GELU$ 是一个非线性函数, $GELU(X_1A_1 + X_2A_2) \neq GELU(X_1A_1) + GELU(X_2A_2)$, 最终这种方式需要一个在 $GELU$ 激活函数之前的同步点.

列切分方式[26]: 另一种方式是在列上切分矩阵 $A = [A_1, A_2]$, 这种切分可以让 $GELU$ 独立地作用于每个切片的输出矩阵上 $[Y_1, Y_2] = [GELU(XA_1), GELU(XA_2)]$.

两种方案各有优劣. 行切方案如果后续结果需要完成 $GELU$ 激活函数, 则需要在给 $GELU$ 输入张量之前, 先进行 All-Reduce 通信, 聚合矩阵乘计算产生的张量, 进而完成矩阵乘计算; 列切方案则生成两个独立的矩阵输出, 且无需通过 All-Reduce 完成后续激活操作, 但对于需要完整张量输入的后续操作, 则需使用 AllGather 聚合输出结果的张量切片. Megatron-LM 根据 Transformer 的计算特点, 设计 MLP 第 1 层的通用矩阵乘使用列切权重矩阵方式, 第 2 层使用行切的权重矩阵方式, 进而只需要在第 2 个通用矩阵乘的输出上跨 GPU 进行一次聚合. 相比于 Megatron-LM 依赖混合 Python 和优化好的 CUDA 内核的方式, MaxText 利用 TensorFlow、JAX 和 XLA 的静态编译, 进而达到通过纯 Python 代码实现并自动优化内核, 其内部也支持数据并行, 全切片数据并行, 序列并行和张量并行.

3.2.1.3 流水并行

张量并行方案通常是为特定结构设计的, 难以迁移到其他任务. 为了实现高效且与任务无关的模型并行性, Google 提出了一个流水并行库 GPipe, 允许扩展任何可以表达为层结构的模型. 通过在单独的加速器上流水线化不同的层序列, GPipe 提供了缩放和切片各种不同网络结构的灵活性, 将网络扩展到更大的规模. 在前向传播计算中, 每个加速器只存储切片边界的输出激活. 在反向传播过程中第 k 个加速器重算复合前向函数 F_k. 因此峰值激活内存需求降低为 $\mathrm{O}\left(N + \frac{L}{K} \times \frac{N}{M}\right)$, 其中 N 是批尺寸, K 是切片分区数量, L 是模型层数, M 是微批次的数量, $\frac{N}{M}$ 是微批尺寸, $\frac{L}{K}$ 是每个切片的层数. 与基线模型的内存需求 $\mathrm{O}(N \times L)$ 相比, 这节省了缓存的激活张量内存. 由于采用了流水并行, 每个加速器会产生一些空闲时间, 称作气泡 (bubble) 时间. 其中气泡时间占比的复杂度为 $\mathrm{O}\left(\frac{K-1}{M+K-1}\right)$, 气泡时间可以被微批次的尺寸 M 所摊销, 越多的微批次尺寸, 气泡占比越低. GPipe 发现当 $M \geqslant 4 \times K$ 的时候, 气泡开销几乎可以忽略, 原因在反向传播时候的重算可以更早地进行调度执行, 从而填充气泡

时间的闲置计算资源. 为了降低流水并行中的内存开销, Narayanan 等人[50]提出了“一轮前向传播紧接着一轮反向传播 (one forward pass followed by one backward pass, 1F1B)”, 虽然这并没有降低气泡的复杂度, 但由于能够更短间隔消费激活张量, 使得其内存占用相比基本流水线方式进一步降低. 为了降低气泡占比, Narayanan 等人[51]提出了交错调度方法 (interleaved schedule) , 与基础的流水并行相比, 每个卡只能分配连续的模型计算阶段进行计算, 而交错调度方式可以分配两个以上不连续的阶段进行前向和反向传播计算, 进一步降低气泡占比. Li 等人[52] 提出了双向流水线技术 Chimera, 进一步降低气泡. EnvPipe[53] 针对使用多卡进行大语言模型训练的场景, 利用流水并行训练大语言模型中的不可去掉的气泡, 选择性地降低 GPU 流式多处理器频率, 从而降低能耗. PipeFisher[54] 巧妙地利用气泡空闲时间执行资源装箱 K-FAC 优化算法计算, 利用闲置资源加速整体训练的收敛.

3.2.1.4 全切片数据并行

数据并行在计算和通信效率上表现良好, 但其内存效率较低, 相比之下, 模型并行具有较高的内存效率, 但计算和通信效率可能较差. 具体来说, 数据并行通过复制整个模型状态来实现, 从而导致冗余的内存消耗, 而模型并行通过将这些状态划分以提高内存效率, 但通常会导致过于细粒度的计算和昂贵的通信, 从而降低扩展效率. 此外, 所有这些方法在整个训练过程中静态地维护所有模型状态, 即使在训练期间并非始终需要所有模型状态. 基于这些观察, DeepSpeed 提出了 ZeRO 来缓解数据并行中的权重数据冗余问题, 进一步提高内存使用效率, 扩展训练模型的规模. 其核心思想是在逻辑上仍然采用数据并行方案, 但在实际部署时, 每个 GPU 仅部署部分模型切片. 当每个 GPU 需要计算该部分模型切片时, 它会按需从含有该切片的 GPU 显存中读取, 从而消除了多 GPU 之间的数据冗余.

ZeRO 提供不同的切片和数据恢复方式, 内存消耗越低, 所需恢复的数据量越大. 以 Adam 优化器的混合精度训练为例, 需要 Float16 类型的权重和梯度, 各为 2ψ, 其中 ψ 为参数量. 另外需要 $K\psi$ 个优化器乘数, 包括 Float32 的权重副本 4ψ, 动量 4ψ 和方差 4ψ, 总和 $K=12$. 这 4 种不同方案的内存开销预估为:

(1) 基线 (不使用 ZeRO)[40]: 内存开销为 $(2+2+K)\times\psi$, 其中 ψ=7.5B 代表模型权重参数量, $K=12$ 代表优化器状态占用的内存乘数, N_d=64 代表数据并行度.

(2) 阶段 1 优化器状态分区[40]: P_{os} 内存开销为 $2\psi+2\psi+\frac{K\times\psi}{N_d}$, 其节省了原本的 4 倍内存开销, 通信开销与数据并行开销一致.

(3) 阶段 2 增加梯度分区[40]: P_{os+g} 的内存开销为 $2\psi+\frac{(2+K)\times\psi}{N_d}$, 节省 8 倍的内存开销, 通信开销和数据并行一致.

(4) 阶段 3 增加参数分区[40]: P_{os+g+p} 内存开销为 $\frac{(2+2+K)\times\psi}{N_d}$, 通信开销和数据并行呈线性关系, 如果有 64 个 GPU, 则相对于基线模型会节省 64 倍内存.

ZeRO 不仅在数据并行和模型并行训练中消除了内存冗余, 还通过将梯度聚合转换为 Reduce-Scatter 操作, 并通过分桶化减少通信次数, 保持低通信成本, 同时重叠计算和通信, 使其能够根据设备数量按比例扩展模型大小, 并保持持续高效. PyTorch 和 FairScale 等系统中的全切片数据并行 (fully sharded data parallel, FSDP) 并行策略, FairScale 称其等价于 ZeRO 阶段 3 方案, 并应用于 Llama 2 进行训练. 修改后的版本已与 PyTorch 的其他组件对齐. FSDP 将模型实例分解为更小的单元, 然后展平并在每个单元内分割所有参数. 分片参数在需要时传递, 并在计算后立即丢弃. 其设计基于以下观察, 尽管各种并行方案已被提出, 但在分布式训练中仍然存在两个挑战. 首先, 其中一些方法与特定模型结构紧密集成 (例如张量并行), 这阻碍了它们成为通用的训练大型模型的解决方案. 其次, 其中一些技术是建立在快速发展的内部接口之上的, 底层机器学习框架容易受到框架实现变化的影响. 因此, FSDP 更加稳健且高效, 具有与框架核心功能兼容的设计. 此外, 采用可组合和可定制的方式构建这样的解决方案, 将促进社区的创新.

3.2.1.5 序列并行

除了上述维度的大语言模型切片和并行外, 研究人员根据大语言模型序列不断增长的特点, 设计了 LightSeq、

Colossal-AI 等序列并行方案. 其设计动机是, 由于张量并行的切片方式会并行计算不同的注意力头, 造成较大的通信数据量和开销, 且很难扩展到超出注意力头数量的并行度, 从而阻碍其进一步应用. LightSeq 则在序列维度上进行切片, 因此它对模型结构不可知, 可以为不同注意力头的模型结构 (例如多头、多查询、多组查询等) 配置并行. 序列并行方式需要结合稀疏注意力机制才能达到较好的性能.

3.2.1.6 自动并行

目前的模型并行训练系统存在两种生成并行化执行计划的方法: 一是由用户手动制定并行化计划方案; 二是从有限的模型并行配置的并行化模板计划中自动生成并行化计划. 然而, 随着模型配置规模的不断增大以及底层基础架构拓扑的变化, 传统方法在分布式计算环境中扩展大型语言模型时显得不够灵活. 为了解决这一问题, Alpa[28]提出了一种类似于 FlexFlow[55]的自动并行方案.

Alpa 将并行性抽象为两个层次: 操作符间 (类似于流水线并行) 和操作符内 (类似于张量并行) 的并行性, 并在此基础上构建了分层的搜索空间, 使得这两个层次之间正交. 最终, Alpa 能够自动推导出高效的并行执行计划, 并通过高效的运行时来协调计划的执行.

Alpa 将并行化问题抽象为以下优化问题: 计算图 $G=(V,E)$ 的总执行成本是所有节点 $v\in V$ 上的计算和通信成本, 以及所有边 $e\in E$ 上的重新分片成本的总和. 当输入张量不满足所选并行算法的分片规格时, 需要进行格式转换, 称为重新分片, 这可能需要跨设备通信. 作者将成本最小化问题转化为整数线性规划 (integer linear programming, ILP) 并使用现成的求解器进行最优求解. 对于每个节点 v, 可能的并行算法数量是 k_v, 每个节点有一个长度为 k_v 的通信成本向量 c_v, 其中 $c_v\in\mathbb{R}^{K_v}$. 同时, 节点 v 有一个计算成本向量 $d_v\in\mathbb{R}^{k_v}$. 对于每个节点 v, 定义一个独热决策向量 $s_v\in\{0,1\}^{k_v}$, 表示它使用的算法. 对于节点 v 和节点 u 之间的重新分片成本, 定义一个重新分片成本矩阵 $R_{vu}\in\mathbb{R}^{k_v\times k_u}$. 问题的优化目标定义为[28]:

$$\min_{s}\sum_{v\in V}s_v^{\mathrm{T}}(c_v+d_v)+\sum_{(v,u)\in E}s_v^{\mathrm{T}}R_{vu}s_u \tag{1}$$

其中, 第 1 项是节点 v 的计算和通信成本, 第 2 项是边 (v,u) 的重新分片成本. 在这个公式中, s 是变量, 其他是常量.

尽管可以使用性能分析来获取 c_v、d_v 和 R_{vu} 的准确成本, 但为简化处理, 作者采用以下估算方法: 对于通信成本 d_v 和 R_{vu}, 通过计算传输的字节数并除以网络带宽得到成本. 对于计算成本 c_v, 作者将其全部设为 0, 这一做法基于以下考虑: (1) 对于如矩阵乘等计算量较大的运算符, 不允许复制计算, 所有并行算法均将计算负载平均分配到各设备, 因而其算术复杂度相同; (2) 对于计算量较小的运算符, 如逐元素操作, 虽然允许复制计算, 但其计算成本可忽略不计. 为简化计算图, 作者将计算量微小的运算符 (如逐元素运算、转置和归约等) 合并到其操作数中, 这极大地减少了图中节点的数量, 从而减小了 ILP 问题的规模. Alpa 通过广度优先搜索计算每个节点的深度, 并将其合并到最深的操作数上, 最终通过整数线性规划求解器完成求解. nnScaler[46]不依赖现有的搜索空间, 而是允许领域专家通过 3 种更通用的基本原语 (op-trans、op-assign 和 op-order) 自定义搜索空间. 这些原语用于表达模型转换及各种并行化计划的时空调度特性. 为避免搜索空间的过度膨胀, nnScaler 在构建空间时支持对原语施加约

表 4 并行优化策略对比

并行方式	单设备计算时间复杂度	参数量空间复杂度	优点	不足
数据并行	$\mathrm{O}\left(\frac{N\times D_{\mathrm{in}}\times D_{\mathrm{out}}}{P}\right)$	$\mathrm{O}(D_{\mathrm{in}}\times D_{\mathrm{out}})\times P$	实现简便, 无需对模型结构进行较大修改. 能够处理大型数据集, 每个处理器分别处理不同的样本批次. 对模型结构无特定要求	当模型参数较小时, 通信开销可能成为性能瓶颈. 需要高效的梯度同步机制, 以保持各处理器间权重一致
全切片数据并行	$\mathrm{O}\left(\frac{N\times D_{\mathrm{in}}\times D_{\mathrm{out}}}{P}\right)+SyncTime$ *SyncTime* 表示同步时间, 因为需要保持梯度的同步, 这可能涉及到通信和同步的开销	$\mathrm{O}(D_{\mathrm{in}}\times D_{\mathrm{out}})$	充分利用所有处理器, 从而提升模型训练速度. 该策略对于大型模型和数据集具有良好的可扩展性. 对模型结构无特定要求	需要处理复杂的数据加载与梯度同步问题, 这可能会导致通信开销增加. 对于小型模型和数据集, 该策略可能会出现性能下降

表 4 并行优化策略对比 (续)

并行方式	单设备计算时间复杂度	参数量空间复杂度	优点	不足
张量并行	$O\left(\frac{N\times D_{in}\times D_{out}}{P}\right)+CommTime$ $CommTime$ 表示通信时间, 取决于模型参数的划分方式	$O(D_{in}\times D_{out})$	该策略适用于大型模型, 可以将模型的不同部分分配给不同的处理器. 该方法降低了单个GPU上的内存需求, 从而加速了计算过程. 该方案与其他切分方法彼此正交、互不冲突, 可以配合使用	并行化过程中需要仔细划分模型, 避免通信瓶颈. 并行方案需要根据模型结构进行定制化设计. 该方案引入了更复杂的通信和同步机制
流水线并行	$O\left(\frac{N\times D_{in}\times D_{out}}{P}\right)+PipeCommTime$ $PipeCommTime$ 表示流水线的时间, 需要协调不同阶段的计算和通信	$O(D_{in}\times D_{out})$	该方法提高了模型的训练速度, 并实现了计算与通信的解耦. 该方法适用于层次较深的模型结构	模型需抽象出层次结构. 需要精心协调不同阶段的计算与通信. 需合理划分模型结构, 以充分发挥流水线并行性的优势. 需要处理流水线中的“气泡”问题. 异步版本可能需要修改优化器, 从而可能对模型精度产生影响
序列并行	$O\left(\frac{N\times D_{in}\times D_{out}}{P}\right)+SeqCommTime$ $SeqCommTime$ 表示序列通信的时间, 取决于序列的划分方式	$O(D_{in}\times D_{out})\times P$	该方法适用于处理长序列的模型, 可将序列分配给不同处理器. 有效缓解了处理长序列时的内存压力. 对模型结构无特定要求	引入了额外的通信与同步成本. 长序列且配合稀疏注意力机制使用, 才能显著提升性能

束. 表 4 详细总结了各种并行策略的计算时间复杂度和空间复杂度, 并全面梳理了它们的优缺点. MLP 的一层全连接层为例, D_{in} 输入特征维度、D_{out} 为输出特征维度, N 为样本数, P 为处理器数. 上述综合分析有助于深入了解每种并行策略在特定场景下的表现, 并为选择适用于特定任务与硬件环境的并行策略提供参考.

3.2.2 张量重算与卸载

检查点 (checkpoint) 是用于在训练阶段备份与恢复张量的一项技术. 一般而言, 检查点可分为两类: 第 1 类用于优化显存使用, 通过释放激活张量并在需要时重新计算, 它被称为激活检查点 (activation checkpointing). 第 2 类用于调试和防止权重丢失, 是用于备份模型权重的检查点.

针对第 1 类的激活检查点, Beaumont 等人[56]提出了一种实现方法. 其设计初衷是释放驻留在内存中的激活张量, 并在需要时通过按需重算来恢复相应的张量. 该方法在前向传播阶段选择激活张量, 并在需要时重算以完成恢复操作. Korthikanti 等人[57]的研究表明, 大多数激活检查点在恢复过程中的冗余计算的影响是可以避免的, 可以在不增加计算负担的前提下有效减少内存消耗. 他们提出了两种新颖但非常简单的技术: 序列并行和激活张量的选择性重计算. 这些策略与张量并行相结合, 几乎消除了对激活张量重新计算的影响.

张量内存卸载是一种有效降低 GPU 显存开销的方法, 类似于操作系统内存管理中的换页机制. 与重新计算不同, 这种方法将激活或权重等卸载到主存或 NVMe 存储, 从而降低显存占用, 在有限资源条件下可训练更大规模的模型. ZeRO-Infinity[58]通过同时利用 CPU 主存和 NVMe 存储来卸载模型和激活张量, 在有限的 GPU 显存资源上支持大规模模型. 此外, ZeRO-Infinity 还引入了以内存为中心的模型层级平铺优化技术, 能够支持对极大规模单层的执行, 同时卸载其他层.

Yuan 等人[59]提出了一种综合考虑张量内存卸载和激活检查点两个因素的高效并行策略搜索方法. 对于张量内存卸载, 该方法通过一种具备流水线并行感知能力的卸载算法, 尽量减少卸载对主机到设备带宽的影响. 同时, 利用计算与内存平衡的激活检查点算法, 在计算开销与内存占用之间进行帕累托最优选择, 从而优化激活检查点策略.

3.2.3 混合专家模型

随着大规模密集型张量语言模型在当前硬件资源约束下逐渐逼近其可扩展性上限, 混合专家模型 (mixture of experts, MoE)[60] 因其层次稀疏性而成为关键的结构设计方向, 成为突破底层算力和基础架构限制的重要模型结构之一. MoE 能够保持固定的计算成本, 将语言模型的参数规模扩展到数万亿.

MoE 层由一组 n 个“专家网络” $E_1,\ldots,E_n$ 和一个“门控网络” G 组成, 门控网络的输出是一个稀疏的 n 维向量, 这些专家本身就是神经网络, 每个都有自己的参数. 每个专家通常接受相同维度的输入, 并生成相同维度的输出.

设 $G(x)$ 和 $E_i(x)$ 分别为门控网络和第 i 个专家网络对给定输入 x 的输出. MoE 模块的输出 y 可以表示为如下形式:

$$y=\sum_{i=1}^{n}G(x)_iE_i(x).$$

常用的门控 (gating) 有 *Softmax* 门控和噪声 Top-K 门控等. *Softmax* 门控是一种简单的非稀疏门控方法, 其原理是将输入与可训练的权重矩阵 W_g 相乘, 并应用 *Softmax* 函数以获得门控分数.

$$G_\sigma(x)=Softmax(x\cdot W_g).$$

MoE 的执行过程如图 7 所示. 每当输入一个标记 (例如“你”), 通过标记路由选择输出概率, 然后选择相应的前馈神经网络专家 (FFN) 进行前向传播和反向传播. 这意味着模型包含多个专家的权重总和, 但每次仅随机选择一个专家进行计算, 从而大幅降低了计算量. 这可以视为一种模型层级的稀疏计算. 与相同质量的密集模型相比, MoE 结构的模型在训练成本上显著降低. 然而, 由于语言模型的尺寸逐渐增大, 以及其独特的模型结构, 如何实现快速的 MoE 模型训练仍然是一个挑战.

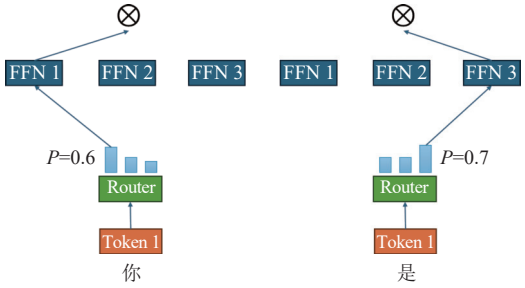

图 7 混合专家模型

分布式 MoE 训练的主要瓶颈是模型计算中交错的 AllToAll 通信所导致的效率低下. Tutel[61]中提出, MoE 算法的性能取决于路由标记, 路由标记确保每个输入标记都能正确地转发给相应的子专家模型. 现有系统由于采用静态执行策略, 其中静态并行和流水线导致计算效率低下, 无法适应动态工作负载, 从而产生性能损失. Tutel 设计了一种具有动态自适应并行性和流水线功能的方法, 能够在运行期间切换并行性和动态流水线, 无需动态迁移张量. 同时, 该方法还实现了高效的通信和快速的编码解码, 从而提升了性能. Lina[62] 分析了 AllToAll 算法开销的主要原因, 通过使用张量分区, 将 AllToAll 尽可能与 All-Reduce 并行执行. 最后, 还通过探索动态专家选择模式, 调度资源以平衡 AllToAll 跨设备的倾斜通信量和带宽. SmartMoE 中 [63]提出, 由于当前 MoE 模型对数据敏感, 不同的数据会导致门控网络动态选择输入和对应专家的匹配, 从而引发不平衡的计算开销. 传统的模型训练通过静态预测固定执行开销, 且优化策略的搜索空间巨大, 使用传统方法进行动态搜索最优方案较慢, 这对模型训练和优化带来了很大挑战. 最终, 作者设计了一个两阶段方案, 通过数据敏感性能预测模型离线构建策略池, 系统通过高效搜索算法在线选择池中的最优并行策略选项.

3.3 性 能

大语言模型因为依赖大规模模型结构和海量数据进行训练, 通常需要数天甚至数月才能完成训练. 其训练所需的大量 GPU 资源成本极高, 因此提升训练性能将大幅降低训练成本, 并提升研究效率. 目前, 研究人员通过高效的注意力机制、混合精度训练、量化感知技术和通信优化等手段, 能够有效提升预训练系统的性能.

3.3.1 高效注意力机制

高效注意力机制指的是能够提高大型语言模型计算和访存效率的一种方法. 通过引入不同的变体, 如访存优化、稀疏注意力、局部注意力等, 这些机制能够在保持模型性能的同时降低计算成本. 这种高效性使得大型语言模型能够更快地处理长文本序列, 加速训练和推理过程, 为自然语言处理任务带来显著的性能提升. 本文将高效注意力分为以下几类: 访存优化、近似与稀疏化、分桶、低秩分解与降维, 后文将对每一类进行详细介绍.

3.3.1.1 访存优化

Andrei 等人[64]发现, Transformer 的训练是一项计算量极大的任务, 然而现有的实现未能有效利用 GPU, 因为数据移动常常成为训练的关键瓶颈. 随着计算性能相较于内存带宽和网络带宽显著提升, 训练过程现在更多受到内存访问的限制. 正如图 8 所示, 大型语言模型通常采用具有以下内存层级的硬件架构: GPU 芯片上的 SRAM、GPU 内的高带宽内存 (HBM) 以及主内存. 模型通常存储在 HBM 中, 内核负责将模型和数据加载到 SRAM, 再进一步传输到寄存器文件和流式多处理器中进行运算. 其中, GPU 具有较强的浮点计算能力, 而访存带宽相对受限. 通过 Roofline[65]模型对常见模型算子 (例如通用矩阵乘 GEMM) 的分析表明, 内存访问往往成为性能瓶颈. 基于这一观察, 内核设计的性能优化目标转向尽可能减少对 HBM 的访问, 并最大化 SRAM 的缓存利用率.

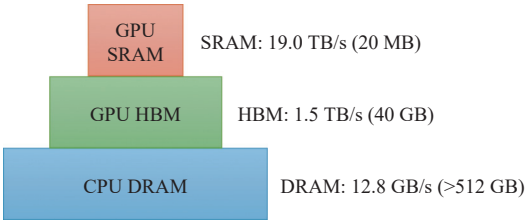

图 8 内存层级[66]

由于自注意力的时间和内存复杂度与序列长度的平方成正比, Transformer 在处理长序列时运行缓慢, 且需要大量内存. FlashAttention[66,67] 指出, 当前内核的一个缺陷在于注意力算法缺乏 I/O 感知能力, 也未充分考虑 GPU 各级内存之间的读写开销. 因此, 作者提出了一种 I/O 感知的注意力算法, 通过引入在线 *Softmax* 算法和内核融合技术, 减少中间结果的 HBM 读写次数; 同时采用平铺策略, 并在反向传播阶段通过重计算进一步降低 I/O 开销. 此外, 作者还将该算法扩展应用于稀疏注意力机制.

注意力机制的输入为 Q、K、$V\in\mathbb{R}^{N\times d}$ 被存储在 HBM 中, 内核读取 QKV 进行注意力计算, 注意力输出 $O\in\mathbb{R}^{N\times d}$ 被写入 HBM. FlashAttention 性能优化的设计目标是减少 HBM 访问的数量 (降低到次 N 的二次方程度), 其思路是通过分块 (tiling), 在线 *Softmax* 等技术让计算不需要物化留存空间复杂度为 $\mathrm{O}(N^2)$ 的中间结果 QK 矩阵, 进而降低空间复杂度和访存. 作者采用了两种已建立的技术分块重计算来克服在次二次方复杂度的 HBM 访问开销. 主要思想是将输入 Q、K、V 分割成块, 从慢速的 HBM 加载到快速的 SRAM 共享内存, 然后对于这些块计算注意力输出, 通过在将每个块的输出按正确的归一化因子缩放后相加, 最终得到正确的结果. 为了达到按块计算注意力, 需要对 *Softmax* 做在线化处理. *Softmax* 将 K 列耦合在一起, 因此作者用缩放的方法对大的 *Softmax* 进行分解. 对于标准的注意力机制计算, 给定输入 $Q,K,V\in\mathbb{R}^{N\times d}$, N 是序列长度, d 是注意力头的维度, 通过以下注意力计算产生输出 $O\in\mathbb{R}^{N\times d}$ [66]:

$$S=QK^{\mathrm{T}},P=Softmax(S)\in\mathbb{R}^{N\times N},O=PV\in\mathbb{R}^{N\times d}.$$

对于向量 $x\in\mathbb{R}^B$ 的分解后的在线 Softmax 计算如下[66]:

$$m(x):=\max_i x_i,f(x):=\left[\mathrm{e}^{x_1-m(x)}\cdots\mathrm{e}^{x_B-m(x)}\right],l(x):=\sum_i f(x)_i,Softmax(x):=\frac{f(x)}{l(x)}.$$

对于向量 $x^{(1)},x^{(2)}\in\mathbb{R}^B$, 可以将连接后的向量 $x=[x^{(1)}x^{(2)}]^{\mathrm{T}}\in\mathbb{R}^{2B}$ 的 *Softmax* 分解为:

$$m(x)=m\left(\left[x^{(1)}x^{(2)}\right]\right)=\max\left(m\left(x^{(1)}\right),m\left(x^{(2)}\right)\right),f(x)=\left[\mathrm{e}^{m\left(x^{(1)}\right)-m(x)}f\left(x^{(1)}\right)\mathrm{e}^{\left(m\left(x^{(2)}\right)\right)-m(x)}f\left(x^{(2)}\right)\right],$$

$$l(x)=l\left(\left[x^{(1)}x^{(2)}\right]\right)=\mathrm{e}^{m\left(x^{(1)}\right)-m(x)}l\left(x^{(1)}\right)+\mathrm{e}^{m\left(x^{(2)}\right)-m(x)}l\left(x^{(2)}\right),Softmax(x)=\frac{f(x)}{l(x)}.$$

标准的注意力实现将矩阵 S 和 P 实例化到高带宽内存 (HBM), 这需要 $O(N^2)$ 的内存. 通常情况下, $N\gg d$(例如, 对于 GPT-2, $N=1024$ 且 $d=64$). 由于一些或大多数操作受制于内存 (例如 *Softmax*), 大量的内存访问导致较慢的墙钟时间. 因此, 如果追踪一些额外的统计信息 $(m(x),l(x))$, 通过以上的分解后的 *Softmax* 计算方法, 可以一次

一个块地计算 *Softmax*. 因此, 其将输入 Q、K、V 分割成块, 计算 *Softmax* 值以及额外的统计信息, 然后合并结果. 这样通过分块, 内核融合等策略, 可以尽可能让每块在 GPU 共享内存完成全部的矩阵乘, *Softmax*, 遮罩, Dropout 和矩阵乘计算, 进而减少了重复的 HBM 数据读写, 同时为了减少 S 和 P 等中间结果常驻内存, 同时应用策略释放中间结果并在反向传播时重算输入张量. 另一种惰性 *Softmax* 也是采用类似的思想, 通过在线化计算不缓存 $\mathrm{O}(N^2)$ 的中间结果.

3.3.1.2 近似与稀疏化

除了从 *Softmax* 层面降低注意力机制二次方的空间复杂度. 通过稀疏化的思想进行长上下文处理优化也是近年内核工作的关注点, 由于注意力机制的计算和空间复杂度都是 $\mathrm{O}(N^2)$ 级别, N 代表是输入标记的数量, 对越来越长的输入标记处理代价很大. 对长上下文的优化集中在对产生瓶颈计算和内存的 Q,K 阶段, 通过稀疏性的思想进行优化.

Longformer[68]通过固定本地模式, 仅计算与相邻标记有关的注意力分数, 以减少计算量并降低中间结果的缓存需求. 为了应对这一挑战, 它引入了一个注意力模式, 通过该模式使自注意力矩阵变得稀疏, 指定了相互关注的输入位置对. 与完全自注意力不同, 所提出的注意力模式与输入序列呈线性关系, 使其在处理较长序列时更为高效. 这一方法使得 Longformer 能够在更大规模的序列上进行有效的注意力计算. Longformer 支持以下几种注意力模式. 1) 滑动窗口模式: 考虑到本地上下文的重要性, 滑动窗口模式仅关注固定尺寸窗口的注意力, 类似于卷积神经网络. 在给定固定窗口大小 w 的情况下, 每个标记只关注其两侧的 $\frac{1}{2}w$ 个标记. 这种模式的计算复杂度是 $\mathrm{O}(n\times w)$, 其中 n 为输入序列长度, 呈线性关系. 2) 扩张 (dilated) 滑动窗口: 为了在不增加计算的情况下进一步提升感受野, 类似于扩张卷积神经网络的思想, 可以对滑动窗口进行扩张. 其中窗口的元素之间具有大小为 d 的间隔. 假设对于所有层, d 和 w 是固定的, 感受野为 $l\times d\times w$, 即使对于较小的 d 值, 也可以涵盖数万个标记. 3) 全局注意力: 针对当前语言模型需要支持各种任务的特点, 对于遮罩语言模型, 只需要本地上下文来预测遮罩的标记, 而对于分类任务则需要整体序列进行预测. 由于当前滑动窗口和扩张滑动窗口对于学习面向具体任务的表达不够灵活, 因此引入了全局注意力. 如果一个标记使用全局注意力, 需要能够对序列中的所有标记进行运算. 例如, 在分类任务中, 全局注意力被应用于 [CLS] 标记. 由于这类标记相比整体序列长度非常小, 所以复杂度仍为 $\mathrm{O}(n)$. 同时, 在哪个部分应用全局标记取决于具体的任务.

3.3.1.3 分 桶

另一种方案通过局部敏感性哈希 (LSH) 替换注意力机制的矩阵乘, 将复杂度由 $\mathrm{O}(n\times n)$ 降低到 $\mathrm{O}(n\log n)$. Reformer[69] 采用了 LSH 注意力机制进行优化. 从形状为 $[batch\_size, length, d_{\text{model}}]$ 的张量 Q,K 和 V 开始分析. 在假设多头机制不变的前提下, 聚焦于注意力计算部分. 如前所述, 主要问题在于矩阵项 QK^{T}, 其形状为 $[batch\_size, length, length]$. 实际上, 我们只关心 $Softmax(QKT)$. 在标准的 Attention 计算中, 公式为 $Attention(Q,K,V)=Softmax\left(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}}\right)V$, 其中输入 Q,K 的维度为 d_k, V 的维度为 d_v, 由于 *Softmax* 主要受到最大元素的影响, 对于每个查询 q_i, 我们只需关注与 q_i 最接近的少数几个键. 例如, 如果 K 的长度为 64K, 对于每个 q_i, 我们只需考虑其中最接近的 32 或 64 个键, 从而显著提高计算效率. 然而, 如何在键中快速找到最近邻, 在高维空间中, 这一问题可以通过 LSH 来解决. LSH 的核心思想是, 通过哈希函数将每个向量 x 映射到一个哈希值 $h(x)$, 使得相近的向量以高概率获得相同哈希值, 而远离的向量则以低概率获得相同哈希值. 在本方法中, 作者要求相邻向量以高概率映射到相同的哈希桶, 并保持哈希桶大小大致均衡. 该方法通过采用随机投影实现: 为了获得 b 个哈希值, 首先固定一个大小为 $\left[d_k,\frac{b}{2}\right]$ 的随机矩阵 R, 然后定义 $h(x)=\mathrm{argmax}([xR;-xR])$ 其中, $[u;v]$ 表示两个向量的连接. 这种方法是一种已有的局部敏感哈希方案, 既易于实现, 又适用于向量批处理任务.

3.3.1.4 低秩分解与降维

矩阵分解和降维是传统用于减少矩阵乘法计算和内存复杂性的经典方法. 由于注意力机制的核心计算也涉及矩阵乘法, 因此可以通过矩阵分解来降低内存和计算成本. Linformer[70]通过对权重矩阵进行变换, 降低了 K 和 V

矩阵的维度, 从而显著减少了中间结果 $QK=P$ 矩阵的维度和内存占用. 它能够在与序列长度相关的线性时间和内存复杂性下, 计算上下文映射 $P\cdot VW_i^V$. 该线性自注意机制的主要思想是在计算 K 和 V 矩阵时, 引入两个线性投影矩阵 E_i、F_i, 其中 E_i、$F_i\in\mathbb{R}^{n\times k}$. 首先, 将原始 $(n\times d)$ 维度的键和值层 KW_i^K 和 VW_i^V 进行投影, 得到 $(k\times d)$ 维度的投影键和值层. 然后, 使用缩放的点积注意力计算一个 $(n\times k)$ 维度的上下文映射矩阵 $\bar{P}$. 其计算公式如下所示[70], 其中, $(\bar{P}\in\mathbb{R}^{n\times k};F_iVW_i^V\in\mathbb{R}^{k\times d})$.

$$overlinehead_i=Attention\left(QW_i^Q,E_iKW_i^k,F_iVW_i^v\right)=\bar{P}\cdot F_iVW_i^V=Softmax\left(\frac{QW_i^Q(E_iKW_i^K)^{\mathrm{T}}}{\sqrt{d_k}}\right)\cdot F_iVW_i^V.$$

使用 $\bar{P}\cdot(F_iVW_i^V)$ 为第 i 个头部计算上下文嵌入. 需要注意的是, 上述操作仅需要 $O(nk)$ 的时间和空间复杂度. 因此, 如果我们能够选择一个较小的投影维度 k, 使得 k 远小于 n, 就能够显著减少内存的消耗.

近似与稀疏化和分桶本质上是对于 QK 矩阵进行近似计算, 只需考虑局部元素; 低秩分解与降维则通过降维来减小输出中间结果矩阵的维度. 另外, 一些工作, 例如 Long-short Transformer[71], 采用了上述 3 种方法的混合优化策略. 表 5 对上文介绍的 4 类高效注意力机制进行对比, 分析了其优势、不足和适用场景.

表 5　高效注意力机制对比

高效注意力类型	优势	不足	适用场景
访存优化	节省访存开销. 对准确度几乎没有影响. 该方法与其他方案正交, 可以结合使用而互不干扰	仅能节省访存开销. 在计算密集型算子场景下, 优化效果可能降低	注意力访存为瓶颈的场景
近似与稀疏化	计算效率和访存效率均有提升	该方法存在一定的信息损失. 不适用于注意力分数矩阵较为稠密的场景. 近似方法的实现需要软硬件协同设计	注意力分数矩阵数据稀疏. 适用于仅需考虑局部注意力的场景
分桶	计算效率和访存效率均有提升	该方法对数据变化较为敏感. 该方法可能会导致分桶倾斜	该方法仅适用于需要考虑局部注意力上下文的场景. 数据分桶较为均匀, 有助于负载的均衡分配
低秩分解与降维	计算效率和访存效率均有提升	有一定信息损失. 需要优化配置降维维度	在注意力分数矩阵为低秩的场景下, 效果更加显著

3.3.2　混合精度训练

在大型语言模型的训练中, 显存和加速器核心是有限的资源. 使用低精度数据类型可以显著降低计算和内存开销. 混合精度训练引入了半精度浮点数 (例如 Float16、BFloat16 等) 来训练大语言模型, 无需修改超参数, 尽可能不损失模型精度的情况下, 提升训练性能和效率. 例如, 将 Float32 替换为 Float16 数据类型, 这会将相应张量的内存需求减半, 并且在 NVIDIA GPU 等最新架构上, 可以利用张量核心等特殊计算单元进一步加快计算速度.

在混合精度训练中, 通常使用半精度格式存储权重张量、激活张量和梯度张量. 然而, 由于这种格式的数值范围比单精度更窄, 可能导致信息丢失或收敛性问题. 为了解决这个问题, Micikevicius 等人[72]提出了 3 种防止关键信息丢失的技术. 首先, 建议维护一个累积的单精度权重张量副本, 每个优化器步骤后得到的梯度 (在前向和反向传播中, 此副本四舍五入为半精度). 其次, 使用损失缩放以保留较小幅度的梯度值. 最后, 使用半精度算术累加并转换为单精度输出, 在存储到内存之前将其转换为半精度. 这些技术的结合有助于克服混合精度训练中可能出现的数值范围狭窄引起的问题. 目前, 混合精度训练方法也被广泛应用于大型语言模型的训练, 例如在 Llama 2 的微调方案中, 用户仍然可以选择混合精度的配置.

如图 9 所示, 在混合精度训练中, 主模型权重首先被转换为 16 比特的 Half 类型, 然后与 Float16 类型的激活一起进行前向传播, 生成激活. 在反向传播计算中, 计算激活梯度时使用 Float16 类型的输入权重和激活梯度, 生成 16 比特的激活梯度; 计算权重梯度时需要 16 比特的激活和激活梯度, 计算出 16 比特的权重梯度, 最终更新模型权重. 这样, 大部分核心计算和显存消耗较大的张量能够保持低精度, 从而加速训练并节省显存.

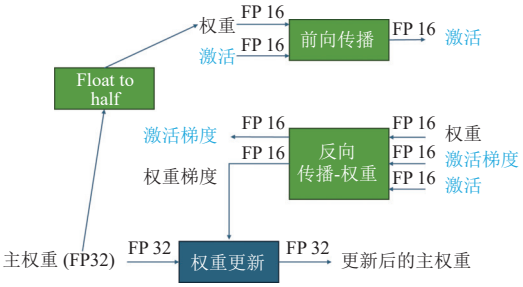

图 9 混合精度训练[72]

3.3.3 量化感知训练

模型量化是指将模型中的高比特数据类型替换为低比特数据类型, 并确保推理准确度的减少在可接受范围内. 通过对张量数据进行量化, 可以降低显存开销和浮点运算量. 量化主要分为两种方式, 即量化感知训练和训练后量化, 其中与预训练相关的是量化感知训练技术. 量化感知训练是在模型训练或微调阶段采用的一种方法. 在训练过程中, 量化的优化目标被整合到模型训练中, 以减少量化精度损失. 例如, LLM-QAT[73] 考虑到难以获取足够的训练数据, 采用预训练模型生成数据, 并通过知识蒸馏进一步量化权重、梯度以及 KVCache. 该策略提高了吞吐量并支持处理长序列. PeQA [74]和 QLoRA[75]属于高效参数微调 (parameter-efficient fine-tuning, PEFT) 的范畴. PEQA 分为两个阶段, 第 1 个阶段将每个全连接层的权重矩阵量化为低精度整型及标量向量; 第 2 个阶段, 在标量向量上对每个下游任务进行微调. QLORA 使用 4 比特量化将预训练模型反向传播到 LoRA (low-rank adaption) 模型结构. 其设计了 4 比特的 Normal Float 类型 (NF4), 并证明其在理论上对正态分布权重最优. 通过量化常数等手段实现双量化, 从而减少平均内存消耗, 并最终应用内存页面优化器来管理峰值内存消耗.

3.3.4 通信优化

随着大型语言模型尺寸的不断增大, 为了进行训练, 需采用分布式训练方式. 在分布式场景下, 性能瓶颈逐渐由其他因素转向通信, 因此, 如何优化和减少通信开销逐渐成为提升系统性能的关键手段. Wang 等人[76]对 Meta 公司集群中的深度学习负载进行了分析, 发现随着使用 GPU 卡数的增加 (例如从 8 卡增加到 128 卡), 通信在整个作业中所占比例从原来几乎可以忽略的程度上升到 40%–60%. 为了应对这一问题, 通信算法优化、异步通信、通信数据量化以及卸载通信原语到自定义硬件策略已成为常见的系统优化手段. 这些策略相互独立, 可结合应用, 共同减少通信开销并提升系统性能.

3.3.4.1 通信算法优化

随着大型语言模型的数据并行或张量并行算法的应用, 通常会使用 All-Reduce 通信优化. 通过采用 RingAll-Reduce, 可以将单个 GPU 总体的输入输出张量的总通信量降低为 $\frac{2(N-1)K}{N}$, K 代表每个 GPU 需要通信的数据量, N 代表节点数量. 这使得性能不再依赖于 GPU 的数量, 因此具有更好的性能和扩展性. 该方式通过只与邻居通信, 更适合当前硬件供应商提供的异构计算资源, 例如在 NVIDIA GPU 节点内通过 NVLink 进行高速互联. 随着异构硬件架构的增多, 大型模型的分布式方案变得更加灵活, 因此自动合成通信方案变得尤为重要. 拓扑感知的集体通信算法合成也是一个重要的优化方向.

当前典型深度学习训练集群的拓扑特点: 节点内通过快速的 NVLink 或 PCIe 互联, 节点间通过 InfiniBand 或以太网互联. 节点间全连接, 通常采用胖树 (fat tree) 架构, 节点间带宽通常相同. 主机通常配置 1 块或多块网卡, 如果网卡数量低于 GPU 数量, GPU 会竞争网卡带宽, 每个 GPU 设备有独立的发送和接收带宽. 由于当前算力基础设施中 GPU 之间的互联方案多种多样, GPU 之间的拓扑也在一定程度上影响性能. 因此, 基于当前拓扑设计和合成高效通信计划与代码, 对提升训练和推理性能非常有用.

Cowan 等人[77]设计了可编程的通信系统 MSCCLang, 提出了用于编写通信算法的领域特定语言和一个用于将这些语言转换为底层代码的优化编译器. 在特定拓扑结构上合成的优化版本 All-Reduce 和 AllToAll 相比基线有 1.9–1.3 倍的加速. 以往的一些工作从拓扑感知的角度入手加速通信, 例如, BlueConnect[78]在云端或数据中心中

的层次网络拓扑结构下, 将单个 All-Reduce 操作分解为大量可并行的归约 (reduce scatter) 和全收集 (AllGather) 操作, 利用延迟和带宽之间的权衡, 以适应各种网络配置. PLink[79]探测物理网络拓扑后, 利用拓扑中的局部性, 合成和执行分层聚合, 同时可以演化执行计划以适应变化的网络条件. 另一些工作将通信算子建模为优化问题或约束满足问题, 通过自动合成通信算法. Blink[80]利用装箱树, 而 Cai 等人[81]则利用约束求解器合成满足帕累托最优的算法.

3.3.4.2 异步通信

近期的研究展示了在计算和通信中实现重叠具有良好的加速效果. 在使用反向传播训练模型时, 需要按顺序传递激活和梯度, 为提升效率, 通信通常需要与计算同步进行. 在 PyTorch DDP 中, 通过将张量分桶组合成一个批次进行通信提升带宽利用效率. 在 PyTorch DDP 的数据并行实现中, 利用深度学习计算的特性, 等待一批梯度张量就绪后开始异步执行 All-Reduce, 待所有梯度聚合完成后, 再进行权重的梯度下降更新. Zhuang 等人[82]提出了一种完全解耦的训练方案, 使用延迟梯度来打破这些同步. 延迟梯度将模型分割成多个模块, 使用不同的进程独立且异步地训练, 并引入梯度收缩来减少由延迟导致的陈旧梯度效应. 最终, 方法证明所提出的延迟梯度算法能确保训练期间的统计收敛. 其中, TicTac[83]通过定位底层计算模型消耗张量的顺序, 控制张量传输顺序, 从而确保近似最优的计算和通信重叠. Romero 等人[84]提出使用缓存机制加速进程间的协调, 减少通信次数, 且该方法在数据并行过程中只需通信一次. 此外, 通过将一批聚合通信原语进行分组, 控制通信缓冲区的尺寸及调度策略, 增加缓存区的动态性, 减少阻塞等待, 防止死锁, 并加速通用并行框架中的控制平面通信流程.

3.3.4.3 通信数据量化

许多训练中采用的随机梯度下降优化算法的通信高效变体, 常常会使用梯度量化方案来降低通信数据量. Faghri 等人[85]指出, 这些方案通常是基于规则的, 并在训练过程中保持不变. 然而, 模型梯度的统计特性在训练过程中是变化的, 这是通过经验观察得出的. 受到这一观察的启发, 文献 [85] 引入了两种自适应量化方案, 即 ALQ 和 AMQ. 在这两种方案中, 通过有效计算参数统计及并行更新, 实现了对梯度的自适应压缩模式. Bian 等人[86]对比了模型并行和数据并行下通信压缩算法的不同特征, 并评估了 3 类常见压缩算法: 基于剪枝、基于学习和基于量化的算法. 文献 [86] 观察到, 基于学习的压缩方法更适合模型并行, 且超参数对压缩算法的收益有影响, 模型较前层对压缩算法的敏感度更高.

3.3.4.4 卸载通信原语到自定义硬件

将通信计算卸载到特定的加速器或网络设备可以减少对 GPU 和 CPU 的中断并实现加速. 例如, BluesMPI[87], ACCL[88]和 BytePS[89]等将通信原语卸载到 SmartNIC、FPGA 或空闲的 CPU 上. 而 SwitchML[90]和 ATP[91]则将聚合通信卸载到网络交换机上, 从而加速深度学习中的 All-Reduce 通信. 管上述方法有些是为通用深度学习模型提出的, 但它们同样适用于大型语言模型的训练. 这些方法可以以插件的形式与其他优化方法正交地集成到现有预训练系统中, 从而在训练过程中实现更为高效的通信和计算.

3.4 可靠性

由于当前大型语言模型预训练系统和平台栈中存在多种缺陷, 从软件层到硬件层可能都存在问题, 这些问题可能导致训练程序崩溃或挂起. 在本节中, 我们首先通过分析当前的缺陷状况, 了解预训练系统面临的问题及其可靠性挑战. 随后, 将从检查点和弹性训练技术两个方面着手, 总结预训练系统如何通过这两项机制提高可靠性.

3.4.1 缺陷分析

为了更有效地训练和测试模型, 企业开发人员通常在共享的多租户平台上进行训练. 然而, 由于程序或平台故障, 一些训练程序在执行一段时间后可能会失败, 导致执行时间过长, 从而降低开发生产力并浪费资源. 分析理解程序、平台和硬件缺陷, 对后续系统设计有重要指导意义. Zhang 等人[92]对微软深度学习平台的作业失败进行了分析, 收集了 4960 个真实失败案例, 手动检查并将其分为 20 类. 针对 400 个故障样本, 确定了常见的根本原因和修复方案. 为了更好地了解当前深度学习的测试和调试实践, 还进行了开发者访谈. 主要发现包括: 1) 48.0% 的故障发生在与平台的交互中, 而不是在代码逻辑的执行上, 主要是由于本地和平台执行环境之间的差异; 2) 深度学习特有的失败 (13.5%) 主要是由于不合适的模型或超参数以及 API 误用; 3) 当前的调试实践对于故障定位效率不

高, 在许多情况下, 开发者需要更适合大型语言模型的开发工具. 除了程序缺陷, 平台本身也存在多种质量问题, 这不仅浪费计算资源, 而且严重降低深度学习和大型语言模型的开发效率. Gao 等人[93]对 Microsoft Platform-X 的质量问题进行了全面实证研究, 检查了 360 个真实问题, 调查了这些问题的常见症状、根本原因和缓解措施. 常见症状及原因包括: 28.33% 的质量问题由硬件故障 (GPU、网络和计算节点) 引起, 28.33% 由系统侧故障 (如系统缺陷、服务中断等) 引起, 用户侧故障 (如程序 Bug、策略违规等) 占 43.34%. 超过 60% 的质量问题可以通过重新提交作业 (34.72%) 或改进用户代码 (24.72%) 来缓解. 这些研究结果为提高深度学习平台服务质量提供了指导, 尤其在开发和维护方面. 这些发现还引发了可能的研究方向和工具支持, 例如后文将介绍的缺陷分析、检查点与弹性训练. 由于作业执行需要排队且资源紧缺, 对作业或模型进行缺陷分析和程序分析可以提前规避无效执行, 从而提高研发生产力. 通过高效的检查点机制和弹性训练, 可以防止硬件故障导致的作业挂起失效. 目前, 作业缺陷分析可以划分为静态分析和动态分析两种方式.

3.4.1.1 静态分析

静态分析是一种在不运行程序的情况下, 通过将程序作为输入来分析其非功能属性或进行程序验证的方法. 这种方法可以有效检测类型缺陷、显存溢出、资源消耗等问题. 以类型缺陷为例, Gao 等人[94]将模型类型缺陷形式化为约束满足问题, 并通过类型检测和 SMT Solver 进行验证. 另外, Gao 等人[95]和 Mei 等人[96]通过分析代价模型或图神经网络模型, 对模型显存消耗或计算代价进行建模, 从而能够预测模型的内存消耗、浮点运算量、GPU 利用率、模型尺寸等. 通过提前对提交的待执行模型进行验证, 可以在大语言模型训练过程中进行优化. 虽然在大语言模型训练中, 由于单次迭代训练代价较大, 模型结构较为稳定, 且缺少自动化机器学习中的搜索空间假设, 但仍然可能遇到同类型的缺陷问题, 这些问题可能会导致更大的资源浪费和性能下降. 因此, 静态分析仍然是解决这些缺陷问题的一种有效手段.

3.4.1.2 动态分析

(1) 动态程序分析. 动态分析需要执行程序并捕获程序运行过程中的日志或中间执行结果, 然后进行分析. 通过这种方式可以规避静态分析缺少上下文、对控制流和动态性的分析局限性等问题, 但要确保动态工具对性能的影响控制在可接受范围内. Zhu 等人[97]通过构建执行痕迹图, 对作业的执行开销进行回放模拟, 从而预测作业在不同优化和硬件拓扑下的性能, 为系统优化和部署提供指导. Le Scao 等人[5]通过观察执行中的 Loss、GNorm 等信息来判断是否出现不收敛、数值缺陷等问题, 进而终止、调试或重启实验, 以加速模型的收敛. MegaScale 在大规模语言模型训练过程中发现不同任务之间存在性能不一致性. 为了不影响训练性能, 该研究通过记录 CUDA 事件, 构建性能检测工具, 以诊断运行时训练程序的性能问题.

除了对作业的分析, 对平台服务和硬件进行检测与分析对保障平台质量也至关重要. 从开源大语言模型训练的编年史和平台质量分析中可以看出, 平台侧需要构建节点健康检测, 以快速定位和剔除不健康节点, 从而减少作业失效或挂起. OpenAI[98]和 MegaScale 披露了其在大规模集群中依靠自动化检测并剔除行为异常节点的做法. 随着时间的推移, 工业界针对人工智能平台建立了许多健康检查系统, 其中分为两类: 被动性检查和主动性检查.

(2) 被动性检查. OpenAI 披露其平台中的某些健康检查是被动的, 始终在所有节点上运行. 这些检查监视基本系统资源, 例如网络可达性、磁盘损坏或已满, 以及 GPU 错误. NVIDIA 的数据中心 GPU 管理器 (DCGM) 工具可以查询这些错误以及其他许多 Xid 错误. 此外, NVML 设备查询 API 还公开了有关 GPU 运行状况和操作的更多详细信息. 一旦检测到错误, 通常可以通过重置 GPU 或系统来修复它们. 在某些情况下, 可能需要物理更换底层 GPU. 另一种被动形式的健康检查是跟踪来自上游云提供商的维护事件. 云提供商通常公开一种了解当前虚拟机是否即将发生维护事件的方法, 该事件最终可能导致中断. 虚拟机可能需要重新启动, 以便应用底层虚拟机管理程序的补丁或将物理节点更换为其他硬件. 这些被动运行状况检查在所有节点的后台持续运行. 如果运行状况检查开始失败, 该节点将自动封锁, 因此不会在该节点上调度新的作业进程. 对于更严重的运行状况检查失败, 还将尝试将进程逐出, 以请求所有当前正在运行的进程立即停止运行. 是否允许这种驱逐仍由进程本身决定, 可通过 Kubernetes Pod 中断预算进行配置.

(3) 主动性检查. 并非所有 GPU 问题都会表现为通过 DCGM 可见的错误代码. 为此, OpenAI 建立了测试库,

用于测试 GPU 并捕获其他问题, 以确保硬件和驱动程序按预期运行. 这些测试无法在后台运行, 它们需要独占 GPU 几秒钟或几分钟才能完成. 首先, 在系统启动时, 节点上会运行这些测试, 该过程被称为“预检”. 所有节点在加入集群时都会应用“预检”标记. 此标记将阻止在节点上调度正常的 Kubernetes Pod. 之后, 预检程序在带有此标签的所有节点上运行预检测试. 成功完成测试后, 测试程序将删除标记, 该节点即可供使用. 此后, 还会在节点生命周期内定期运行这些测试, 以确保可用节点顺利部署到集群中. 虽然测试哪些节点具有一定的随机性且不受控制, 但随着时间的推移, 这种方法可以提供足够的覆盖范围, 同时将干扰降至最低. 微软还开源了人工智能硬件测试工具 SuperBench[99], 包含深度学习基准测试作业, 用于定期测试面向深度学习的新硬件的质量和稳定性. MegaScale 在万卡规模的大规模语言模型训练过程中, 也会主动进行节点内和节点间的网络测试, 以主动检测网卡质量问题, 尽早发现硬件层和软件层的通信问题并进行迁移.

综上所述, 当前大语言模型执行的软硬件栈均存在许多缺陷, 导致作业失效或挂起. 因此, 迫切需要提供高效的容错机制, 以保障大语言模型训练的顺利完成. 在接下来的两节中, 将介绍检查点和弹性训练技术, 以支持预训练系统在平台上可靠地运行.

3.4.2 检查点

大语言模型通常使用检查点技术备份模型, 以实现容错并确保在故障时能够恢复. 在以往的深度学习系统设计中, 容错往往被忽视, 没有进行充分的优化. 一般来说, 检查点操作采用同步方式执行, 随着大型语言模型规模的不断增大, 这将是非常耗时的操作. 深度学习框架, 如 PyTorch、TensorFlow 等, 为用户提供了检查点接口, 使其能够在训练期间创建备份. 然而, 这些框架通常让用户自行决定何时执行检查点、备份哪些模型分片以及触发检查点的频率, 这可能导致效率低下的问题. Rojas 等人[100]研究了常见框架的检查点实现, 评估了检查点的计算成本、文件格式和大小、规模的影响以及检查点非确定性问题. Wang 等人[101]提出了 Gemini, 这是一种面向大语言模型的分布式检查点和故障恢复方案. 此方案以平均浪费时间为主要度量标准来评估检查点解决方案的性能, 因为故障可能在任何时候发生, 导致浪费时间有所变化.

在最理想的情况下, 即在完成检查点后立即发生故障, 浪费的时间为 $t_{\text{cpkt}}+t_{\text{rtvl}}$, 其中 t_{ckpt} 是检查点时间, t_{rtvl} 是恢复时间. 而在最糟糕的情况下, 即在完成检查点之前发生故障, 浪费的时间为 $t_{\text{cpkt}}+\dfrac{1}{f}+t_{\text{rtvl}}$, 其中 f 是检查点频率. 假设故障在两个连续检查点之间均匀分布, 平均浪费时间 (T_{wasted}) 可以表示为下式[101].

$$T_{\text{wasted}}=t_{\text{ckpt}}+\frac{1}{2f}+t_{\text{rtvl}},$$

并且包含以下约束条件:

$$\frac{1}{f}\geqslant\max\left(t_{\text{ckpt}},T_{\text{iter}}\right).$$

其中, T_{iter} 表示迭代时间, 因为模型无需在迭代内进行检查点, 而且检查点不能在之前的检查点完成前启动. 为了减少浪费时间并提高检查点频率 f, 降低检查点时间 t_{ckpt} 至关重要. 尽可能使用主存检查点可以降低检查点时间 t_{ckpt} 和恢复时间 t_{rtvl}. 同时, 通过检查点流量调度算法减少检查点时间 t_{ckpt} 和最小化对训练作业使用网络的干扰, 从而降低 T_{iter} 迭代时间. 从约束公式来看, 这也就是检查点频率有机会进一步提高的机会. MegaScale 通过将检查点分为两个阶段进行加速. 第 1 个阶段将数据由 GPU 显存写入到主存, 通过优化 PyTorch 序列化方法并使用固定内存 (pinned memory), 然后继续训练, 最终使用高带宽 PCIe 只需要几秒钟完成备份. 第 2 个阶段使用一个后台进程异步将主存数据拷贝到分布式文件系统 HDFS 中.

对于深度学习和推荐模型的检查点工作, 仍有许多基本技术可以供大语言模型借鉴应用. DeepFreeze[102]通过异步检查点方式进行深度学习备份, 但其检查点数据通过相对单一的策略存放在远端存储. DeepFreq[103] 提出了动态调整检查点频率的方法. Check-N-Run[104]提出了面向推荐系统深度学习模型的两种检查点优化策略. 首先, 它应用差分检查点, 跟踪并备份模型的修改部分. 差分检查点在以下情况能够发挥效益: 当模型的一小部分 (通常存储为嵌入表) 在每次迭代中更新时, 例如推荐模型场景中. 换句话说, 这种方法适用于模型更新非常稀疏的场景, 可以有效减少检查点的尺寸. 其次, 利用量化技术可以显著减少检查点的大小, 同时不降低训练准确度. Oobleck[105]采

用了规划-执行协同设计的方法, 先生成一组流水线模板, 并实例化至少 f+1 个逻辑上等效的流水线副本, 以容忍 f 个同时发生的故障. 在执行期间, 其依赖于跨副本复制的模型状态来提供快速恢复.

这些设计思路在大语言模型中同样适用, 尤其是在采用并行化模型切片技术的背景下, 带来了新的检查点挑战和系统设计机会.

3.4.3 弹性训练

弹性训练的设计初衷有两个方面. 首先, 由于平台的不稳定性容易导致故障, 弹性训练可以实现故障恢复或容错. 其次, 在大多数分布式训练系统中, 用户需要在提交作业之前手动配置资源, 如 GPU 卡数和内存等, 并且无法在运行时调整资源配置, 静态的资源配置严重影响了作业的性能和利用率. 通过弹性训练, 可以充分利用闲置资源, 提高资源利用率. 目前, 弹性训练的实现方式主要包括两种: 一类是部署层控制方式, 另一类是运行时插桩.

3.4.3.1 部署层控制

TorchElastic[106]是 PyTorch 官方提供的弹性解决方案, 主要对节点成员资格变更时的故障进行了建模. 当节点发生故障时, 故障被抽象为缩小规模事件; 当故障节点被调度器替换时, 则抽象为向上扩展事件. 该工具在数据并行方面表现较好, 且对于容错作业和弹性作业, 可以配置重新启动的次数. DLRover[107]是一个分布式开源深度学习框架, 它可以自动配置深度学习作业的初始资源, 并在运行时动态调整作业资源, 以获得更好的性能. 借助其弹性能力, 当检测到性能问题或作业因故障或驱逐而失败时, DLRover 能够有效调整作业资源. 它通过对作业中所有工作者进程的吞吐量 R 进行建模, 将整个执行计划抽象为最大化吞吐量 R 的优化问题[107]:

$$R=\sum_{i=0}^{N_w}\frac{B}{t_{IO}+\dfrac{W_{\text{compute}}}{r\cdot\min(w_i,\hat{w})}+\dfrac{W_{\text{update}}}{r\cdot\min(s_{\text{total}},\hat{s})}}$$

其中, B 表示批尺寸, N_w 表示作业中的工作者数量, R 会随着 N_w, w, S_{total} 的增加而增长. T_{IO} 代表读取和预处理一个批次数据的时间, 以及传输权重和梯度的通信时间. w_i 表示配置的 CPU 核心, $\hat{w}$ 表示实际使用的 CPU 核心, r 表示 CPU 计算容量. W_{compute} 代表前向和反向计算梯度的计算量, W_{update} 代表更新参数的运算量, s_{total} 和 $\hat{s}$ 分别代表参数更新阶段的 CPU 核心数量和实际使用的 CPU 核心数量. 然而, 由于一批作业只能共享集群资源, 并受到核心最大计算容量等约束的限制, 整个问题在考虑这些约束后, 演变为一个非线性整数规划问题, 并且被认为是 NP 困难的. AntMan[108]通过充分利用闲置的 GPU 资源, 实现了在共享的 GPU 上同时运行多个作业, 从而提高 GPU 利用率. 该方法充分利用了深度学习训练的独特特性, 并在深度学习框架内引入了内存和计算的动态扩展机制.

3.4.3.2 运行时插桩

对于云服务而言, 通过提高深度学习工作负载的 GPU 的高利用率来降低成本至关重要. Singularity[109]提供了全球分布式调度服务, 用于高效、可靠地执行深度学习训练和推理工作负载. Singularity 的核心是工作负载感知调度程序, 能够在加速器 (例如 GPU、FPGA) 集群中透明地进行抢占和弹性扩展深度学习工作负载, 从而提高利用率, 而不会影响其正确性或性能. 其机制是透明的, 通过对 CUDA 和驱动层接口的插桩, 用户无需修改代码, 也不需要使用可能限制灵活性的自定义库. 此外, 该方法通过提供透明的检查点机制来增强深度学习工作负载的可靠性. 该方法与模型架构无关, 支持多种并行策略 (例如数据并行、流水并行、模型并行). 表 6 总结了不同类型弹性训练方案的优势、不足以及适用场景.

表 6 弹性训练方案对比

检查点类型	优势	不足	适用场景
部署层控制	可以理解模型结构和并行方案减少备份冗余数据	由于被绑定到特定系统, 该方案的通用性受到限制	该方案适用于特定的框架或固定的部署平台
运行时插桩	适用于各种上层系统框架	无法理解模型结构和并行方案可能备份冗余数据. 底层硬件和驱动层限制造成实现困难	平台系统适合提供此种类型的方案进行作业调度或作业容错

3.5 小 结

随着大型语言模型训练集群规模的增长和训练时间的延长, 训练过程中遇到硬件故障的概率随之增加. 因此, 总结了提升大语言模型预训练系统可靠性的措施, 包括及早发现故障、快速诊断问题, 并通过有效的容错机制、检查点设置和快速弹性恢复, 确保预训练系统在出现故障时能够及时恢复, 从而提高整体可靠性.

4 大语言模型预训练系统面临的挑战与应对

尽管大语言模型在社会各界引起了广泛关注, 但当前的大语言模型预训练系统仍面临诸多问题和挑战. 这些挑战包括模型本身的特性、资源的限制、技术的快速发展, 以及系统复杂性增加所带来的缺陷问题. 本节将对上述问题和挑战进行系统梳理.

4.1 大模型大数据与资源紧缺的挑战

OpenAI 的 Henighan 等人[4]提出的 Scaling Law 在 4 个领域中确定了交叉熵损失的实证缩放定律: 生成图像建模、视频建模、多模态图像与文本模型转换以及数学问题求解. 在这几种情况下, 随着模型大小和计算预算的增加, 自回归基于 Transformer 的模型性能都能平稳提升, 其损失缩放关系遵循幂律规律. 最优模型大小也取决于计算预算, 通过一种幂律来衡量, 其指数在所有数据领域中几乎是普遍适用的. 同时, 通过对 GPT-3[2]的实验与发布, 不断通过扩展模型和算力提升模型效果. 然而, 对于大部分的机构与个人而言, 预期的模型效果需要海量的算力资源支撑才能完成训练. 然而, 实际情况是资源紧缺, 模型训练耗时长, 资源消耗大. 因此, 如何支撑更大规模的模型部署, 提升模型计算的性能, 以及提升资源的利用率, 仍然是需要解决的根本问题. 在这个过程中, 可以从算法层、系统层和硬件层的协同设计与优化中获取启示. 例如, 混合专家模型通过模型层的稀疏性, 在有限资源下进一步增加模型参数的效果, 展现出系统和算法融合的潜力. FlashAttention 则采用在线 *Softmax*, 改变了算法计算聚合模式, 为系统层进一步降低访存提供了机会. 同时, 在算法方面, 可以探索更加稀疏的模型架构设计, 以便在固定的计算成本下探索更大的语言模型. 为此, 需要建立大模型训练的跨层理论和模型, 理清算法、系统和硬件之间的内在联系, 使其成为大语言模型扩展性、计算性能优化与利用率提升的基石.

4.2 研究与工程快速迭代的挑战

受益于开源社区和预印版论文模式, 学术界和工业界对大语言模型进行的研究和工程实践迭代速度非常快. 在如此快速演进的算法、模型结构和系统面前, 如何通过大语言模型预训练系统更加灵活地支撑研究并适配工程需求是巨大的挑战. 对研发者而言, 快速跟进、二次开发以及开源系统的应用也是一项考验. 为了应对快速的研究与工程迭代, 可以通过标准化与工具化手段, 缓解开发难点并提升研发效率. 开源社区提供了更统一的接口标准, 使得新的模型与系统优化能够以可插拔的方式集成入现有工具链. 例如, 大语言模型通常发布于 Hugging Face 社区, 用户可以通过其 Transformers 库方便地进行下载、训练和微调. 社区与系统的联合设计在一定程度上推动了大语言模型开发流程的标准化. 基于面向切面的设计原则, 研究者可以持续研发出调试工具和程序分析工具, 用于解决新系统和算法在监控、剖析、日志处理等方面的共性问题. 由于大语言模型具备较强的上下文理解与程序合成能力, 借助大语言模型辅助自身系统的研发, 也在一定程度上提升了开发效率.

4.3 系统复杂性与缺陷带来的挑战

由于大规模模型部署、算力基础架构及模型研发的快速迭代特点, 大语言模型预训练系统在设计和开发上缺乏充分的测试, 同时基础架构层面容易发生故障. 这为高质量支撑大语言模型的研发和训练带来了极大的挑战. 因此, 如何形式化或定量化地描述大语言模型系统及其缺陷复杂性的本质特征与外在度量指标, 并进一步研究系统和缺陷复杂性的内在机理是一个关键问题. 通过对大语言模型计算和缺陷规律的研究, 有助于理解其复杂模式的本质特征和生成机制, 从而提升系统设计质量, 获得更清晰的系统抽象, 并有效指导大语言模型系统的设计. 因此, 厘清算法与系统之间的内在联系, 可实现算法与系统的协同设计. 通过对数据通信与计算复杂性机理的建模与解析, 阐明大语言模型按需简化、降低复杂度的原理与机制; 同时, 通过对缺陷的形式化建模, 阐明其静态或动态验

证、测试的原理与机制. 这些对问题的理解和形式化建模工作将成为大语言模型计算的理论基础.

5 前沿展望与未来趋势

随着大模型的发展, 多模态模型逐渐引起关注, 其模型架构以大语言模型为推理基础, 集成了其他模态 (例如, 图像、视频、语音等). 如何在原有的预训练系统上支持更多模态, 如多模态的数据管理、预处理与流水线读取; 如何综合设计保证精度的高效跨模态的基于稀疏性优化及量化; 在模型更加异构, 集群资源更加异构趋势下, 对更加复杂的并行切片搜索空间, 如何高效地设计并行策略; 针对多模态模型更丰富的算子集合, 设计更加高效的内核算法与内核融合, 加速算子执行. 以上方向都对未来的预训练系统设计提出了新的挑战和机遇.

随着大语言模型编码、调试和数学推理能力的不断提升, 在软件工程与系统社区逐渐有研究人员尝试利用大语言模型的能力指导系统设计. 同理, 大语言模型也有潜力应用到大语言模型系统本身的设计、优化与调试诊断中. 如何通过设计提示词让大语言模型辅助设计新的并行策略, 编写更加高效的内核; 通过日志和监控以及程序等信息, 让大语言模型辅助诊断程序的性能缺陷, 提示下一步的优化策略; 当训练作业崩溃时, 通过大语言模型分析错误日志, 辅助诊断、调试和修复程序缺陷等方向需要进一步探索.

6 总 结

随着大语言模型应用与技术的快速发展, 各类应用层出不穷, 导致模型规模和部署迅速增长, 使得大语言模型渗透到越来越多的行业和业务领域, 成为重要的生产要素. 本文系统地解构大语言模型的训练过程, 分别梳理了支撑大语言模型的系统技术现状, 包括预训练系统、系统扩展性、性能和可靠性. 尽管这些技术逐渐达成了共识, 但尚未形成统一的跨阶段、跨技术栈的协同设计与优化方案. 本文总结了各种技术在大语言模型训练中的关键作用及其优劣势的比较. 最后, 梳理了大语言模型预训练系统当前面临的挑战, 并提出了潜在的应对方案.

References

[1] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: ACM, 2017. 6000–6010.

[2] Brown TB, Mann B, Ryder N, *et al*. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: ACM, 2020. 159.

[3] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-Training of deep bidirectional transformers for language understanding. In: Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis: ACL, 2019. 4171–4186. [doi: 10.18653/v1/N19-1423]

[4] Henighan T, Kaplan J, Katz M, Chen M, Hesse C, Jackson J, Jun H, Brown TB, Dhariwal P, Gray S, Hallacy C, Mann B, Radford A, Ramesh A, Ryder N, Ziegler DM, Schulman J, Amodei D, McCandlish S. Scaling laws for autoregressive generative modeling. arXiv:2010.14701, 2020.

[5] Le Scao T, Fan A, Akiki C, *et al*. BLOOM: A 176B-parameter open-access multilingual language model. arXiv:2211.05100, 2022.

[6] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F, Rodriguez A, Joulin A, Grave E, Lample G. LLaMA: Open and efficient foundation language models. arXiv:2302.13971, 2023.

[7] Touvron H, Martin L, Stone K, *et al*. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288, 2023.

[8] Black S, Biderman S, Hallahan E, Anthony Q, Gao L, Golding L, He H, Leahy C, McDonell K, Phang J, Pieler M, Prashanth US, Purohit S, Reynolds L, Tow J, Wang B, Weinbach S. GPT-NeoX-20B: An open-source autoregressive language model. In: Proc. of BigScience Episode#5--Workshop on Challenges & Perspectives in Creating Large Language Models. Dublin: Association for Computational Linguistics, 2022. 95–136. [doi: 10.18653/v1/2022.bigscience-1.9]

[9] Zhang SS, Roller S, Goyal N, Artetxe M, Chen MY, Chen SH, Dewan C, Diab M, Li X, Lin XV, Mihaylov T, Ott M, Shleifer S, Shuster K, Simig D, Koura PS, Sridhar A, Wang TL, Zettlemoyer L. OPT: Open pre-trained transformer language models. arXiv:2205.01068, 2022.

[10] Zeng AH, Liu X, Du ZX, *et al*. Glm-130B: An open bilingual pre-trained model. arXiv:2210.02414, 2023

[11] Rae JW, Borgeaud S, Cai T, *et al*. Scaling language models: Methods, analysis & insights from training gopher. arXiv:2112.11446, 2021.

[12] Thoppilan R, De Freitas D, Hall J, *et al*. LaMDA: Language models for dialog applications. arXiv:2201.08239, 2022.

[13] Du N, Huang YP, Dai AM, *et al*. GLaM: Efficient scaling of language models with mixture-of-experts. In: Int'l Conf. on Machine Learning. Baltimore: ICML, 2022. 5547–5569.

[14] Chowdhery A, Narang S, Devlin J, *et al*. PaLM: Scaling language modeling with pathways. The Journal of Machine Learning Research, 2023, 24(1): 240.

[15] Anil R, Dai AM, Firat O, *et al*. PaLM 2 technical report. arXiv:2305.10403, 2023.

[16] Zeng W, Ren XZ, Su T, *et al*. PanGu-α: Large-scale autoregressive pretrained Chinese language models with auto-parallel computation. arXiv:2104.12369, 2021.

[17] Su H, Zhou X, Yu HJ, Shen XY, Chen YW, Zhu ZL, Yu Y, Zhou J. WeLM: A well-read pre-trained language model for Chinese. arXiv:2209.10372, 2022.

[18] Chung HW, Hou L, Longpre S, *et al*. Scaling instruction-finetuned language models. The Journal of Machine Learning Research, 2024, 25(1): 70.

[19] The Mosaic Research Team. Introducing MPT-7B: A new standard for open-source, commercially usable LLMs. 2023. https://www.databricks.com/blog/mpt-7b

[20] Ma ZX, Zhai JD, Han WT, Chen WG, Zheng WM. Challenges and measures for efficient training of trillion-parameter pre-trained models. ZTE Technology Journal, 2022, 28(2): 51–58 (in Chinese with English abstract). [doi: 10.12142/ZTETJ.202202008]

[21] Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Rault T, Louf R, Funtowicz M, Davison J, Shleifer S, von Platen P, Ma C, Jernite Y, Plu J, Xu CW, Le Scao T, Gugger S, Drame M, Lhoest Q, Rush AM. Transformers: State-of-the-art natural language processing. In: Proc. of the 2020 Conf. on Empirical Methods in Natural Language Processing: System Demonstrations. ACL. 2020. 38–45. [doi: 10.18653/v1/2020.emnlp-demos.6]

[22] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin ZM, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai JJ, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: ACM, 2019. 721.

[23] Sergeev A, Del Balso M. Horovod: Fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799, 2018.

[24] Huang YP, Cheng YL, Bapna A, Firat O, Chen MX, Chen DH, Lee H, Ngiam J, Le QV, Wu YH, Chen ZF. GPipe: Efficient training of giant neural networks using pipeline parallelism. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: ACM, 2019. 10.

[25] Li S, Zhao YL, Varma R, Salpekar O, Noordhuis P, Li T, Paszke A, Smith J, Vaughan B, Damania P, Chintala S. PyTorch distributed: Experiences on accelerating data parallel training. Proc. of the VLDB Endowment, 2020, 13(12): 3005–3018. [doi: 10.14778/3415478.3415530]

[26] Shoeybi M, Patwary M, Puri R, LeGresley P, Casper J, Catanzaro B. Megatron-LM: Training multi-billion parameter language models using model parallelism. arXiv:1909.08053, 2019.

[27] Rasley J, Rajbhandari S, Ruwase O, He YX. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In: Proc. of the 26th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining. ACM, 2020. 3505–3506. [doi: 10.1145/3394486.3406703]

[28] Zheng LM, Li ZH, Zhang H, Zhuang YH, Chen ZF, Huang YP, Wang YD, Xu YZ, Zhuo DY, Xing EP, Gonzalez JE, Stoica I. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX, 2022. 559–578.

[29] Abadi M, Barham P, Chen JM, Chen ZF, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker PA, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng XQ. TensorFlow: A system for large-scale machine learning. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Savannah: USENIX, 2016. 265–283.

[30] Google. MaxText. GitHub [Internet]. 2023. https://github.com/google/maxtext/

[31] Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q. JAX: Composable transformations of Python+NumPy programs. 2018. http://github.com/google/jax

[32] Li DC, Shao RL, Xie AZ, Xing EP, Gonzalez JE, Stoica I, Ma XZ, Zhang H. LightSeq: Sequence level parallelism for distributed training of long context transformers. arXiv:2310.03294, 2023. [doi: 10.48550/arXiv.2310.03294]

[33] MindNLP Contributors. MindNLP: Easy-to-use and high-performance NLP and LLM framework based on MindSpore [Internet]. 2022. https://github.com/mindlab-ai/mindnlp

[34] Zhao YL, Gu A, Varma R, Luo L, Huang CC, Xu M, Wright L, Shojanazeri H, Ott M, Shleifer S, Desmaison A, Balioglu C, Damania P, Nguyen B, Chauhan G, Hao YC, Mathews A, Li S. PyTorch FSDP: Experiences on scaling fully sharded data parallel. Proc. of the VLDB Endowment, 2023, 16(12): 3848–3860. [doi: 10.14778/3611540.3611569]

[35] Nie XN, Liu Y, Fu FC, Xue JB, Jiao D, Miao XP, Tao YY, Cui B. Angel-PTM: A scalable and economical large-scale pre-training system in tencent. Proc. of the VLDB Endowment, 2023, 16(12): 3781–3794. [doi: 10.14778/3611540.3611564]

[36] Baines M, Bhosale S, Caggiano V, Goyal N, Goyal S, Ott M, Lefaudeux B, Liptchinsky V, Rabbat M, Sheiffer S, Sridhar A. FairScale: A general purpose modular pytorch library for high performance and large scale training. 2020. https://github.com/facebookresearch/fairscale

[37] Mosaic ML Team. Composer. 2021. https://github.com/mosaicml/composer/

[38] Databricks. Databricks Mosaic: pioneering AI & open-source research. 2024. https://www.databricks.com/mosaic

[39] Smith S, Patwary M, Norick B, LeGresley P, Rajbhandari S, Casper J, Liu Z, Prabhumoye S, Zerveas G, Korthikanti V, Zhang E, Child R, Aminabadi RY, Bernauer J, Song X, Shoeybi M, He YX, Houston M, Tiwary S, Catanzaro B. Using deepspeed and megatron to train megatron-turing NLG 530B, a large-scale generative language model. arXiv:2201.11990, 2022.

[40] Rajbhandari S, Rasley J, Ruwase O, He YX. ZeRO: Memory optimizations toward training trillion parameter models. In: Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Atlanta: ACM, 2020. 20.

[41] Ren J, Rajbhandari S, Aminabadi RY, Ruwase O, Yang SY, Zhang MJ, Li D, He YX. ZeRO-offload: Democratizing billion-scale model training. In: Proc. of the 2021 USENIX Annual Technical Conf. USENIX, 2021. 551–564.

[42] Aicheng Technology, Alibaba Group. Megatron-LLaMA [Internet]. 2023. https://github.com/alibaba/Megatron-LLaMA

[43] Li SG, Liu HX, Bian ZD, Fang JR, Huang HC, Liu YL, Wang BX, You Y. Colossal-Ai: A unified deep learning system for large-scale parallel training. In: Proc. of the 52nd Int'l Conf. on Parallel Processing. Salt Lake City: ACM, 2023. 766–775. [doi: 10.1145/3605573.3605613]

[44] Jiang ZH, Lin HB, Zhong YM, *et al*. MegaScale: Scaling large language model training to more than 10 000 GPUs. In: Proc. of the 21st USENIX Symp. on Networked Systems Design and Implementation. Santa Clara: USENIX, 2024. 745–760.

[45] PaddleNLP Contributors. PaddleNLP: An easy-to-use and high performance NLP library. 2024. https://github.com/PaddlePaddle/PaddleNLP

[46] Lin ZQ, Miao YS, Zhang QL, Yang F, Zhu Y, Li C, Maleki S, Cao X, Shang N, Yang YL, Xu WJ, Yang M, Zhang LT, Zhou LD. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In: Proc. of the 18th USENIX Symp. on Operating Systems Design and Implementation. Santa Clara: USENIX, 2024. 347–363.

[47] Patarasuk P, Yuan X. Bandwidth optimal all-reduce algorithms for clusters of workstations. Journal of Parallel and Distrib Computing, 2009, 69(2): 117–124. [doi: 10.1016/j.jpdc.2008.09.002]

[48] Ott M, Edunov S, Baevski A, Fan A, Gross S, Ng N, Grangier D, Auli M. fairseq: A fast, extensible toolkit for sequence modeling. In: Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics. Minneapolis: ACL, 2019. 48–53. [doi: 10.18653/v1/N19-4009]

[49] Ma SM, Wang HY, Huang SH, Wang WH, Chi ZW, Dong L, Benhaim A, Patra B, Chaudhary V, Song X, Wei FR. TorchScale: Transformers at scale. arXiv:2211.13184, 2022.

[50] Narayanan D, Harlap A, Phanishayee A, Seshadri V, Devanur NR, Ganger GR, Gibbons PB, Zaharia M. PipeDream: Generalized pipeline parallelism for DNN training. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 1–15. [doi: 10.1145/3341301.3359646]

[51] Narayanan D, Shoeybi M, Casper J, LeGresley P, Patwary M, Korthikanti V, Vainbrand D, Kashinkunti P, Bernauer J, Catanzaro B, Phanishayee A, Zaharia M. Efficient large-scale language model training on GPU clusters using megatron-LM. In: Proc. of the 2021 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. St.Louis: ACM, 2021. 1–15. [doi: 10.1145/3458817.3476209]

[52] Li SG, Hoefler T. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In: Proc. of the 2021 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. St.Louis: ACM, 2021. 27. [doi: 10.1145/3458817.3476145]

[53] Choi S, Koo I, Ahn J, Jeon M, Kwon Y. EnvPipe: Performance-preserving DNN training framework for saving energy. In: Proc. of the 2023 USENIX Annual Technical Conf. Boston: USENIX, 2023. 851–864.

[54] Osawa K, Li SG, Hoefler T. PipeFisher: Efficient training of large language models using pipelining and Fisher information matrices. In: Proc. of the 6th Conf. on Machine Learning and Systems. Miami: MLSys, 2023. 708–727.

[55] Lu WY, Yan GH, Li JJ, Gong SJ, Han YH, Li XW. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In: Proc. of the 2017 IEEE Int'l Symp. on High Performance Computer Architecture (HPCA). Austin: IEEE, 2017. 553–564.

IEEE. [doi: 10.1109/HPCA.2017.29]

[56] Beaumont O, Eyraud-Dubois L, Herrmann J, Joly A, Shilova A. Optimal Re-materialization strategies for heterogeneous chains: How to train deep neural networks with limited memory. ACM Trans. on Mathematical Software, 2024, 50(2): 10. [doi: 10.1145/3648633]

[57] Korthikanti VA, Casper J, Lym S, McAfee L, Andersch M, Shoeybi M, Catanzaro B. Reducing activation recomputation in large transformer models. In: Proc. of the 6th Conf. on Machine Learning and Systems. Miami: MLSys, 2023.

[58] Rajbhandari S, Ruwase O, Rasley J, Smith S, He YX. ZeRO-infinity: Breaking the GPU memory wall for extreme scale deep learning. In: Proc. of the 2021 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. St. Louis: ACM, 2021. 59. [doi: 10.1145/3458817.3476205]

[59] Yuan TL, Liu YL, Ye XC, Zhang SL, Tan JC, Chen B, Song CR, Zhang D. Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. In: Proc. of the 2024 USENIX Annual Technical Conf. Santa Clara: USENIX, 2024. 545–561.

[60] Yuksel SE, Wilson JN, Gader PD. Twenty years of mixture of experts. IEEE Trans. on Neural Networks and Learning Systems, 2012, 23(8): 1177–1193. [doi: 10.1109/TNNLS.2012.2200299]

[61] Hwang C, Cui W, Xiong YF, Yang ZY, Liu Z, Hu H, Wang ZL, Salas R, Jose J, Ram P, Chau H, Cheng P, Yang F, Yang M, Xiong YQ. Tutel: Adaptive mixture-of-experts at scale. In: Proc. of the 6th Conf. on Machine Learning and Systems. Miami: MLSys, 2023.

[62] Li JM, Jiang YM, Zhu YB, Wang C, Xu H. Accelerating distributed MoE training and inference with lina. In: Proc. of the 2023 USENIX Annual Technical Conf. Boston: USENIX, 2023. 945–959.

[63] Zhai MS, He JA, Ma ZX, Zong Z, Zhang RQ, Zhai JD. SmartMoE: Efficiently training sparsely-activated models through combining offline and online parallelization. In: Proc. of the 2023 USENIX Annual Technical Conf. Boston: USENIX, 2023. 961–975.

[64] Ivanov A, Dryden N, Ben-Nun T, Li SG, Hoefler T. Data movement is all you need: A case study on optimizing transformers. In: Proc. of the 4th Conf. on Machine Learning and Systems. MLSys, 2021. 711–732.

[65] Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM, 2009, 52(4): 65–76. [doi: 10.1145/1498765.1498785]

[66] Dao T, Fu DY, Ermon S, Rudra A, Ré C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: ACM, 2022. 1189.

[67] Dao T. FlashAttention-2: Faster attention with better parallelism and work partitioning. In: Proc. of the 12th Int'l Conf. on Learning Representations. Vienna: ICLR, 2024.

[68] Beltagy I, Peters ME, Cohan A. Longformer: The long-document transformer. arXiv:2004.05150, 2020.

[69] Kitaev N, Kaiser Ł, Levskaya A. Reformer: The efficient transformer. In: Proc. of the 8th Int'l Conf. on Learning Representations. Addis Ababa: ICLR, 2020.

[70] Wang SN, Li BZ, Khabsa M, Fang H, Ma H. Linformer: Self-attention with linear complexity. arXiv:2006.04768, 2020.

[71] Zhu C, Ping W, Xiao CW, Shoeybi M, Goldstein T, Anandkumar A, Catanzaro B. Long-short Transformer: Efficient transformers for language and vision. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. ACM, 2021. 17723–17736.

[72] Micikevicius P, Narang S, Alben J, Diamos G, Elsen E, Garcia D, Ginsburg B, Houston M, Kuchaiev O, Venkatesh G, Wu H. Mixed precision training. arXiv:1710.03740. 2017.

[73] Liu ZC, Oguz B, Zhao CS, Chang E, Stock P, Mehdad Y, Shi YY, Krishnamoorthi R, Chandra V. LLM-QAT: Data-Free quantization aware training for large language models. In: Proc. of the Findings of the Association for Computational Linguistics. Bangkok: ACL, 2024. 467–484. [doi: 10.18653/v1/2024.findings-acl.26]

[74] Arshia FZ, Keyvanrad MA, Sadidpour SS, Mohammadi SMR. PeQA: A massive Persian question-answering and chatbot dataset. In: Proc. of the 12th Int'l Conf. on Computer and Knowledge Engineering (ICCKE). Mashhad: IEEE, 2022. 392–397. [doi: 10.1109/ICCKE57176.2022.9960071]

[75] Dettmers T, Pagnoni A, Holtzman A, Zettlemoyer L. QLoRA: Efficient finetuning of quantized llms. In: Proc. of the 37th Int'l Conf. on Neural Information Processing Systems. New Orleans: ACM, 2023. 441.

[76] Wang WY, Khazraee M, Zhong ZZ, Ghobadi M, Jia ZH, Mudigere D, Zhang Y, Kewitsch A. TopoOpt: Co-Optimizing network topology and parallelization strategy for distributed training jobs. In: Proc. of the 20th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX, 2023. 739–767.

[77] Cowan M, Maleki S, Musuvathi M, Saarikivi O, Xiong YF. MSCCLang: Microsoft collective communication language. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Vancouver: ACM, 2023. 502–514. [doi: 10.1145/3575693.3575724]

[78] Cho M, Finkler U, Kung DS, Hunter HC. BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy.

In: Proc. of the 2nd Conf. on Machine Learning and Systems. Stanford: SysML, 2019. 241–251.

[79] Luo L, West P, Krishnamurthy A, Ceze L, Nelson J. PLink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In: Proc. of the 3rd Conf. on Machine Learning and Systems. Austin: SysML, 2020. 82–97.

[80] Wang GH, Venkataraman S, Phanishayee A, Thelin J, Devanur NR, Stoica I. Blink: Fast and generic collectives for distributed ml. In: Proc. of the 3rd Conf. on Machine Learning and Systems. Austin: SysML, 2020. 172–186.

[81] Cai ZX, Liu ZY, Maleki S, Musuvathi M, Mytkowicz T, Nelson J, Saarikivi O. Synthesizing optimal collective algorithms. In: Proc. of the 26th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. ACM, 2021. 62–75. [doi: 10.1145/3437801.3441620]

[82] Zhuang HP, Wang Y, Liu QL, Lin ZP. Fully decoupled neural network learning using delayed gradients. IEEE Trans. on Neural Networks and Learning Systems, 2022, 33(10): 6013–6020. [doi: 10.1109/TNNLS.2021.3069883]

[83] Hashemi SH, Abdu Jyothi S, Campbell RH. TicTac: Accelerating distributed deep learning with communication scheduling. In: Proc. of the 2nd Conf. on Machine Learning and Systems. Stanford: SysML, 2019. 418–430.

[84] Romero J, Yin JQ, Laanait N, Xie B, Young MT, Treichler S, Starchenko V, Borisevich AY, Sergeev A, Matheson MA. Accelerating collective communication in data parallel training across deep learning frameworks. In: Proc. of the 19th USENIX Symp. on Networked Systems Design and Implementation. Renton: USENIX, 2022. 1027–1040.

[85] Faghri F, Tabrizian I, Markov I, Alistarh D, Roy DM, Ramezani-Kebrya A. Adaptive gradient quantization for data-parallel sgd. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: ACM, 2020. 267.

[86] Bian S, Li DC, Wang HY, Xing EP, Venkataraman S. Does compressing activations help model parallel training? In: Proc. of the 7th Annual Conf. on Machine Learning and Systems. Santa Clara: MLSys, 2024. 239–252.

[87] Bayatpour M, Sarkauskas N, Subramoni H, Hashmi JM, Panda DK. BluesMPI: Efficient MPI non-blocking alltoall offloading designs on modern BlueField smart NICs. In: Proc. of the 36th Int'l Conf. on High Performance Computing. Virtual Event: Springer, 2021. 18–37. [doi: 10.1007/978-3-030-78713-4_2]

[88] Dong JB, Wang SC, Feng F, Cao Z, Pan H, Tang LB, Li PC, Li H, Ran QY, Guo YQ, Gao SY, Long X, Zhang J, Li Y, Xia ZS, Song LYH, Zhang YY, Pan P, Wang GH, Jiang XW. ACCL: Architecting highly scalable distributed training systems with highly efficient collective communication library. IEEE Micro, 2021, 41(5): 85–92. [doi: 10.1109/MM.2021.3091475]

[89] Jiang YM, Zhu YB, Lan C, Yi BR, Cui Y, Guo CX. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In: Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation. USENIX, 2020. 463–479.

[90] Sapio A, Canini M, Ho CY, Nelson J, Kalnis P, Kim C, Krishnamurthy A, Moshref M, Ports DRK, Richtárik P. Scaling distributed machine learning with in-network aggregation. In: Proc. of the 18th USENIX Symp. on Networked Systems Design and Implementation. USENIX, 2021. 785–808.

[91] Lao C, Le YF, Mahajan K, Chen YX, Wu WF, Akella A, Swift MM. ATP: In-network aggregation for multi-tenant learning. In: Proc. of the 18th USENIX Symp. on Networked Systems Design and Implementation. USENIX, 2021. 741–761.

[92] Zhang R, Xiao WC, Zhang HY, Liu Y, Lin HX, Yang M. An empirical study on program failures of deep learning jobs. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1159–1170. [doi: 10.1145/3377811.3380362]

[93] Gao YJ, Shi XX, Lin HX, Zhang HY, Wu H, Li R, Yang M. An empirical study on quality issues of deep learning platform. In: Proc. of the 45th Int'l Conf. on Software Engineering: Software Engineering in Practice. Melbourne: IEEE, 2023. 455–466. [doi: 10.1109/ICSE-SEIP58684.2023.00052]

[94] Gao YJ, Li ZX, Lin HX, Zhang HY, Wu M, Yang M. Refty: Refinement types for valid deep learning models. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1843–1855. [doi: 10.1145/3510003.3510077]

[95] Gao YJ, Gu XY, Zhang HY, Lin HX, Yang M. Runtime performance prediction for deep learning models with graph neural network. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Melbourne: IEEE, 2023. 368–380. [doi: 10.1109/ICSE-SEIP58684.2023.00039]

[96] Mei HQ, Qu HZ, Sun JW, Gao YJ, Lin HX, Sun GZ. GPU occupancy prediction of deep learning models using graph neural network. In: Proc. of the 25th IEEE Int'l Conf. on Cluster Computing. Santa Fe: IEEE, 2023. 318–329. [doi: 10.1109/CLUSTER52292.2023.00034]

[97] Zhu HY, Phanishayee A, Pekhimenko G. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In: Proc. of the 2020 USENIX Annual Technical Conf. USENIX, 2020. 337–352.

[98] OpenAI. Scaling kubernetes to 7 500 nodes. 2021. https://openai.com/index/scaling-kubernetes-to-7500-nodes/

[99] Xiong YF, Jiang YT, Yang ZY, Qu L, Zhao GS, Liu SG, Zhong D, Pinzur B, Zhang J, Wang Y, Jose J, Pourreza H, Baxter J, Datta K, Ram P, Melton L, Chau J, Cheng P, Xiong YQ, Zhou LD. SuperBench: Improving cloud AI infrastructure reliability with proactive

validation. In: Proc. of the 2024 USENIX Annual Technical Conf. Santa Clara: USENIX, 2024. 835–850.

[100] Rojas E, Kahira AN, Meneses E, Gomez LB, Badia RM. A study of checkpointing in large scale training of deep neural networks. arXiv:2012.00825, 2020.

[101] Wang Z, Jia Z, Zheng S, Zhang Z, Fu XW, Ng TSE, Wang YD. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In: Proc. of the 29th Symp. on Operating Systems Principles. Koblenz: ACM, 2023. 364–381. [doi: 10.1145/3600006.3613145]

[102] Nicolae B, Li JL, Wozniak JM, Bosilca G, Dorier M, Cappello F. DeepFreeze: Towards scalable asynchronous checkpointing of deep learning models. In: Proc. of the 20th IEEE/ACM Int'l Symp. on Cluster, Cloud and Internet Computing. Melbourne: IEEE, 2020. 172–181. [doi: 10.1109/CCGrid49817.2020.00-76]

[103] Mohan J, Phanishayee A, Chidambaram V. CheckFreq: Frequent, fine-grained DNN checkpointing. In: Proc. of the 19th USENIX Conf. on File and Storage Technologies. USENIX, 2021. 203–216.

[104] Eisenman A, Matam KK, Ingram S, Mudigere D, Krishnamoorthi R, Nair K, Smelyanskiy M, Annavaram M. Check-N-Run: A checkpointing system for training deep learning recommendation models. In: Proc. of the 19th USENIX Symp. on Networked Systems Design and Implementation. Renton: USENIX, 2022. 929–943.

[105] Jang I, Yang ZN, Zhang Z, Jin X, Chowdhury M. Oobleck: Resilient distributed training of large models using pipeline templates. In: Proc. of the 29th Symp. on Operating Systems Principles. Koblenz: ACM, 2023. 382–395. [doi: 10.1145/3600006.3613152]

[106] PyTorch. Elastic. 2024. https://github.com/pytorch/elastic

[107] Wang QL, Sang B, Zhang HT, Tang MJ, Zhang K. DLRover: An elastic deep training extension with auto job resource recommendation. arXiv:2304.01468, 2023.

[108] Xiao WC, Ren SR, Li Y, Zhang Y, Hou PY, Li Z, Feng YH, Lin W, Jia YQ. AntMan: Dynamic scaling on GPU clusters for deep learning. In: Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation. USENIX, 2020. 533–548.

[109] Shukla D, Sivathanu M, Viswanatha S, Gulavani B, Nehme R, Agrawal A, Chen C, Kwatra N, Ramjee R, Sharma P, Katiyar A, Modi V, Sharma V, Singh A, Singhal S, Welankar K, Xun L, Anupindi R, Elangovan K, Rahman H, Lin Z, Seetharaman R, Xu C, Ailijiang E, Krishnappa S, Russinovich M. Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads. arXiv:2202.07848, 2022.

附中文参考文献

[20] 马子轩, 翟季冬, 韩文弢, 陈文光, 郑纬民. 高效训练百万亿参数预训练模型的系统挑战和对策. 中兴通讯技术, 2022, 28(2): 51–58. [doi: 10.12142/ZTETJ.202202008]

作者简介

高彦杰, 博士生, CCF 专业会员, 主要研究领域为大语言模型系统与工具, 大数据系统.

陈跃国, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为金融科技, 计算社会科学, 知识图谱, 语义搜索.