

大模型 - 训练微调

作者: Calvin

QQ: 179209347

Mail: 179209347@qq.com

介绍

笔记简介:

- 面向对象: 深度学习初学者
- 依赖课程: **线性代数, 统计概率**, 优化理论, 图论, 离散数学, 微积分, 信息论

知乎专栏:

<https://zhuanlan.zhihu.com/p/693738275>

Github & Gitee 地址:

https://github.com/mymagicpower/AIAS/tree/main/deep_learning

https://gitee.com/mymagicpower/AIAS/tree/main/deep_learning

* 版权声明:

- 仅限用于个人学习
- 禁止用于任何商业用途

背景

对于大型深度学习模型，我们面临着与大规模机器学习相似的问题：

除了大量的资源用于大型训练数据集之外，我们在深度学习中还面临一个额外的问题：深度学习神经网络可能具有非常大的权重集合和复杂的配置。这些权重集合在训练模式和预测/推理模式下都需要大量的资源来存储。

- **CPU限制问题**
- **内存限制问题**

为了训练大型深度学习模型，我们可以采取以下方法来分散或分布训练过程：

- **数据并行**：将训练数据分成多个批次，并在多个计算设备（如多个GPU）上并行处理这些批次。每个设备计算自己的梯度，并将梯度聚合以更新模型参数。
- **模型并行**：将模型的不同部分分配给不同的计算设备，每个设备负责计算分配给它的部分。这种方法适用于具有大量参数的模型，可以减少内存占用。

为了优化服务器内存以存储大型模型，可以考虑以下方法：




- **模型压缩**：使用压缩算法或技术来减小模型的内存占用。例如，可以使用低精度浮点数表示参数，或者使用特定的压缩算法来减小模型的大小。
- **内存管理**：在训练过程中，可以优化内存管理策略，如及时释放不再需要的中间结果，或者使用分布式存储来存储模型的部分。

全新的大模型商业应用范式



技术选型 – 模型 – 分布式训练框架

- 与多款开源大模型的无缝衔接，可执行增量预训练、指令微调等任务类型。开发者仅需使用8GB消费级显卡，就可以训练出适用于具体业务场景的“专属大模型”。这极大地降低了进行大模型训练的“真金白银”成本。

微调框架	技术特点	技术支持	Github
 DeepSpeed	<ul style="list-style-type: none"> 结合Ray、HuggingFace训练模型 被用于高效地微调大模型，并且能使用多节点获得最高性价比而不带来额外的复杂度； 	微软	https://github.com/microsoft/DeepSpeed
 Megatron-LM	<ul style="list-style-type: none"> 由 NVIDIA 开发的大规模语言模型训练工具。目标是提供高效、可扩展的训练工具，以便在大规模计算集群上训练具有数十亿或数百亿参数的语言模型。 提供了一系列的优化策略和技术，以加速训练过程并提高模型的性能。 	NVIDIA	https://github.com/NVIDIA/Megatron-LM
XTuner	<ul style="list-style-type: none"> 轻量级: 支持在消费级显卡上微调大语言模型。 多样性: 支持多种大语言模型，数据集和微调算法（QLoRA、LoRA），支撑用户根据自身需求选择合适的解决方案。 兼容性: 兼容 DeepSpeed 和 HuggingFace 的训练流程，支撑用户无感式集成与使用。 	上海AI实验室	https://github.com/InternLM/xtuner
 PEFT	<ul style="list-style-type: none"> 旨在通过最小化微调参数的数量和计算复杂度，来提高预训练模型在新任务上的性能，从而缓解大型预训练模型的训练成本。 模型参数高效微调，目前支持Prefix Tuning、Prompt Tuning、PTuningV1、PTuningV2、Adapter、LoRA、AdaLoRA, LoRA 	hugging face	https://github.com/huggingface/peft
MFTCoder	<ul style="list-style-type: none"> 开源的多任务代码大语言模型项目，包含代码大模型的模型、数据、训练等； 高精度、高效率、多任务、多模型支持、多训练算法，大模型代码能力微调框架； 	开源社区	https://github.com/codefuse-ai/MFTCoder
LLMTune	<ul style="list-style-type: none"> 多个LLM的模块化支持（目前支持MetaAI开源的2个模型，LLaMA和OPT） 支持广泛的消费级NVIDIA的GPU显卡（包括RTX系列、A系列、GTX系列等） 代码库微小且易于使用（整个源代码仅64k大小） 	康奈尔大学	https://github.com/kuleshov-group/llmtune

分布式训练概述

分布式训练系统仍然需要克服计算墙、显存墙、通信墙等多种挑战，以确保集群内的所有资源得到充分利用，从而加速训练过程并缩短训练周期。

- **计算墙：**单个计算设备所能提供的计算能力与大语言模型所需的总计算量之间存在巨大差异。2022年3月发布的NVIDIA H100 SXM的单卡FP16算力也只有 2000 TFLOPs，而 GPT-3 则需要 314 ZFLOPs 的总计算量，两者相差了 8 个数量级。
- **显存墙：**单个计算设备无法完整存储一个大语言模型的参数。GPT-3包含1750亿参数，在推理阶段如果采用FP32格式进行存储，需要700GB的计算设备内存空间，而 NVIDIA H100 GPU只有80GB显存。
- **通信墙：**分布式训练系统中各计算设备之间需要频繁地进行参数传输和同步。由于通信的延迟和带宽限制，这可能成为训练过程的瓶颈。GPT-3训练过程中，如果分布式系统中存在128个模型副本，那么在每次迭代过程中至少需要传输 89.6TB的梯度数据。而截止2023年8月，单个InfiniBand链路仅能够提供不超过800Gb/s 带宽。

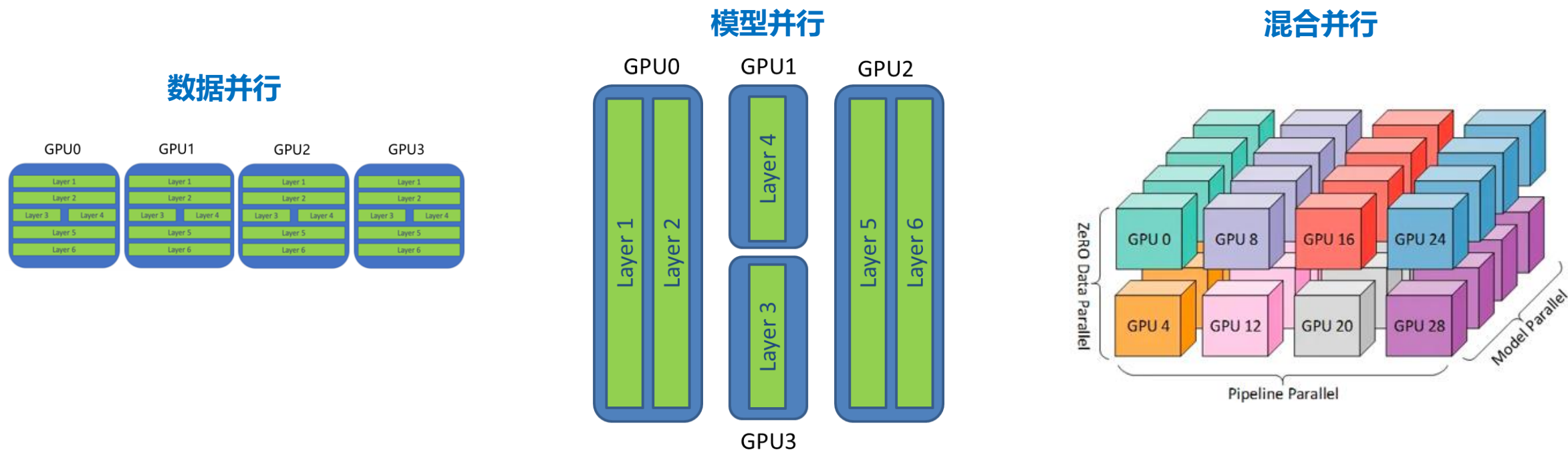
大语言模型参数量和所使用的数据量都非常巨大，因此都采用了**分布式训练架构**完成训练：

- **BLOOM模型**的研究人员则公开了更多在硬件和所采用的系统架构方面的细节。该模型的训练一共花费3.5个月，使用48个计算节点。每个节点包含8块NVIDIA A100 80G GPU（总计384个GPU），并且使用4*NVLink用于节点内部GPU之间通信。节点之间采用四个Omni-Path 100 Gbps网卡构建的增强8维超立方体全局拓扑网络进行通信。
- **LLaMA模型**训练采用NVIDIA A100-80GB GPU，LLaMA-7B模型训练需要 82432 GPU小时， LLaMA-13B模型训练需要 135168 GPU小时， LLaMA-33B模型训练花费了 530432 GPU小时，而LLaMA-65B模型训练花费则高达 1022362 GPU小时。

分布式训练的并行策略

如果进行并行加速，可以从**数据和模型**两个维度进行考虑：

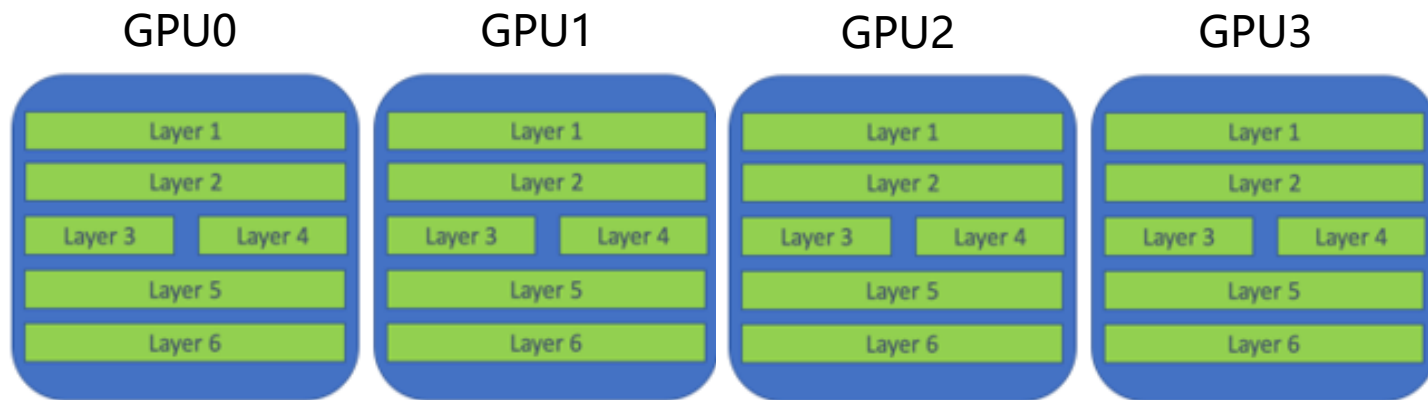
- **数据并行 (Data Parallelism, DP)** : 将同一个模型复制到多个设备上，并行执行不同的数据分片。
- **模型并行 (Model Parallelism, MP)** : 将模型中的算子分发到多个设备分别完成。
- **混行并行 (Hybrid Parallelism, HP)** : 同时对数据和模型进行切分，从而实现更高层次的并行。



分布式训练 - 数据并行

在这种模式下，训练数据被分成多个子集，每个子集在不同的GPU（工作节点）上运行相同的复制模型。在批次计算结束时，这些节点需要同步模型参数（或其“梯度”），以确保它们正在训练一致的模型（就像算法在单个处理器上运行一样），因为每个节点将独立计算其对训练样本的预测与标记输出（这些训练样本的正确值）之间的误差。因此，每个设备都必须将其所有更改发送到节点上的所有模型。

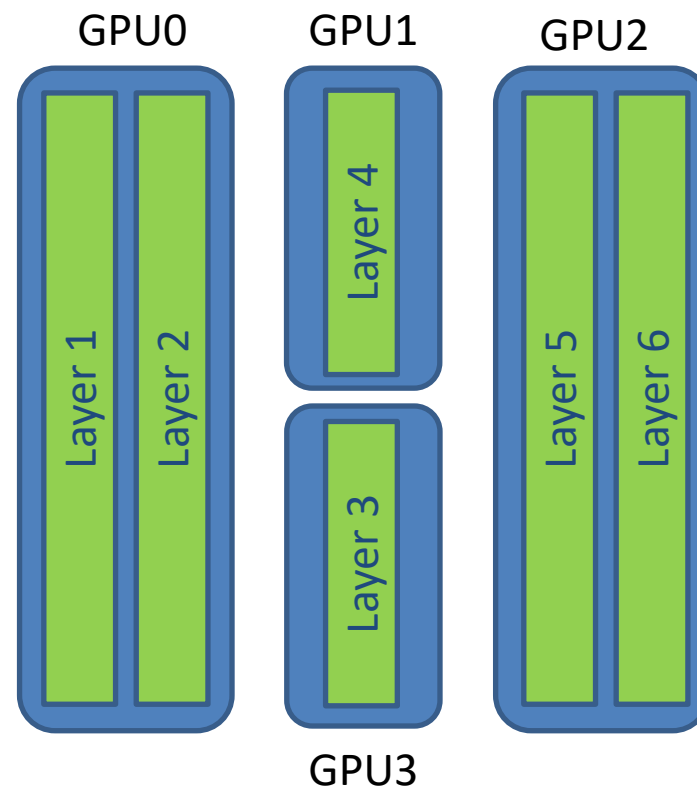
- 能够根据可用数据的数量进行扩展，并加快整个数据集对优化的贡献速度。
- 需要较少的节点间通信，因为能从每个权重的高计算数量中受益。
- 整个模型需要完全适应每个节点的显存。



分布式训练 – 模型并行

模型的不同层被分割开来：

- 我们可以**将网络层分割到多个GPU上**（甚至可以将每个层所需的工作拆分）。也就是说，每个GPU接收特定层中的数据流作为输入，在神经网络中处理几个后续层的数据，然后将数据发送到下一个GPU。
- 这也被称为**网络并行**，因为模型将被分割为不同的部分，可以同时运行，每个部分在不同的节点上使用相同的训练数据运行。
- 这样可以**减少通信需求**，因为工作节点只需要同步共享参数（通常是每个前向或反向传播步骤一次）。
- 这种方法在**共享高速总线（如NVLink或InfiniBand）**的机架上的GPU上效果很好。
- 与数据并行相比，模型并行**更难实现**。
- 然而，有些情况下**模型太大无法放入**单个机器中，这时**模型并行可能是一个更好的选择**。最常见的用例是现代自然语言处理模型，如GPT-2和GPT-3，它们包含数十亿个参数（实际上GPT-2有15亿个参数）。

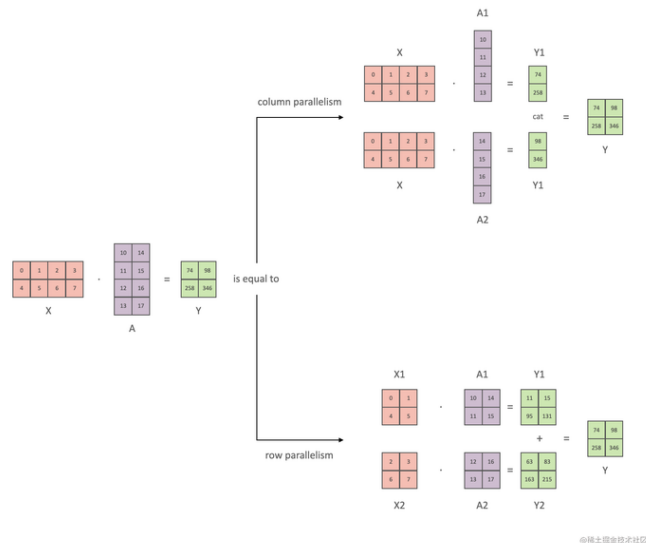
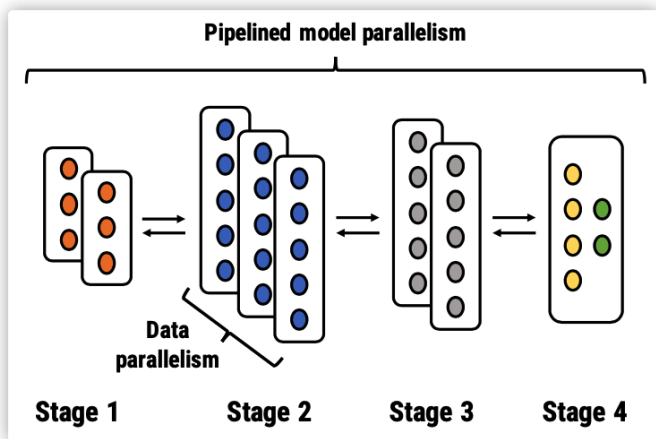


分布式训练 – 模型并行

模型并行往往用于解决单节点显存不足的问题。以包含1750 亿参数的GPT-3 模型为例，如果模型中每一个参数都使用**32 位浮点数表示**，那么模型需要占用**700GB**（即175G×4 Bytes）显存。如果使用16 位浮点数表示，每个模型副本也需要占用350GB 内存。H100 加速卡也仅支持80GB 显存，无法将整个模型完整放入其中。

模型并行可以从计算图角度，用以下两种形式进行切分：

- **流水线并行 (Pipeline Parallelism, PP)**：按模型的层切分到不同设备，即**层间并行或算子间并行** (Inter-operator Parallelism) 。
- **张量并行 (Tensor Parallelism, TP)**：将计算图层内的参数切分到不同设备，即**层内并行或算子内并行** (Intra-operator Parallelism) 。

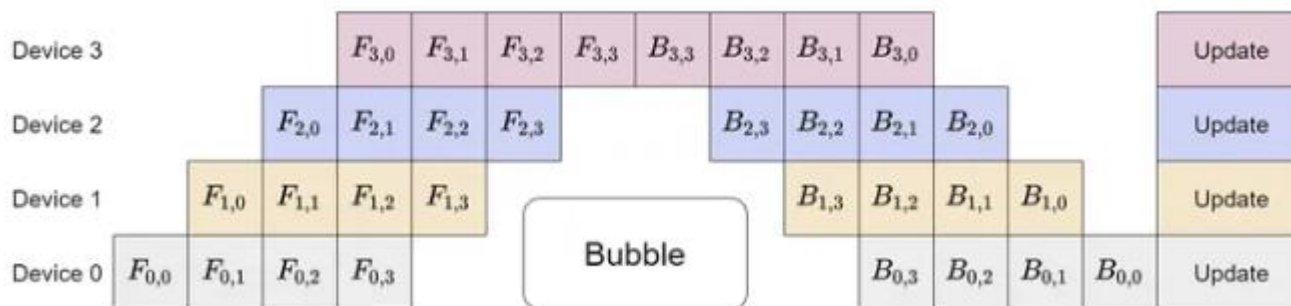


模型并行—流水线并行

流水线并行是一种并行计算策略，将模型的各个层分段处理，并将每个段分布在不同的计算设备上，使得前后阶段能够流水式、分批进行工作。流水线并行通常应用于大语言模型的并行系统中，以有效解决单个计算设备内存不足的问题。

下图给出了一个由四个计算设备组成的流水线并行系统，包含了前向计算和后向计算：

- 其中F0、F1、F2、F3 分别代表四个前向路径，位于不同的设备上；
- 而B3、B2、B1、B0 则代表逆序的后向路径，也分别位于四个不同的设备上。



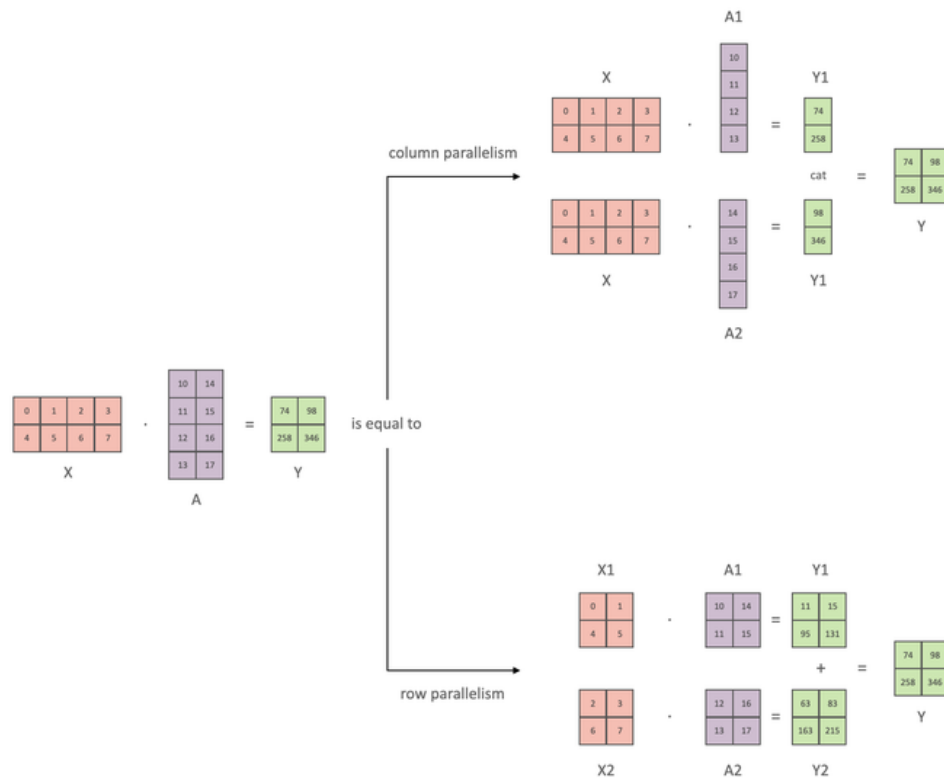
模型并行—张量并行

张量并行需要根据模型的**具体结构和算子类型**，解决如何将参数切分到不同设备，以及如何保证切分后数学一致性这两个问题。

大语言模型都是以Transformer 结构为基础，Transformer 结构主要由**嵌入式表示**（Embedding）、**矩阵乘**（MatMul）和**交叉熵损失**（Cross Entropy Loss）计算构成。这三种类型的算子有较大的差异，**都需要设计对应的张量并行策略**才可以实现将参数切分到不同的设备。

矩阵乘的张量并行要充分利用矩阵的分块乘法原理。

- 要实现矩阵乘法 $Y=X \times A$ ，其中 X 是维度为 $M \times N$ 的输入矩阵， A 是维度为 $N \times K$ 的参数矩阵， Y 是结果矩阵，维度为 $M \times K$ 。
- 如果参数矩阵 A 非常大，甚至超出单张卡的显存容量，那么可以把参数矩阵 A 切分到多张卡上，并通过集合通信汇集结果，保证最终结果在数学计算上等价于单计算设备的计算结果。参数矩阵 A 存在以下两种切分方式。

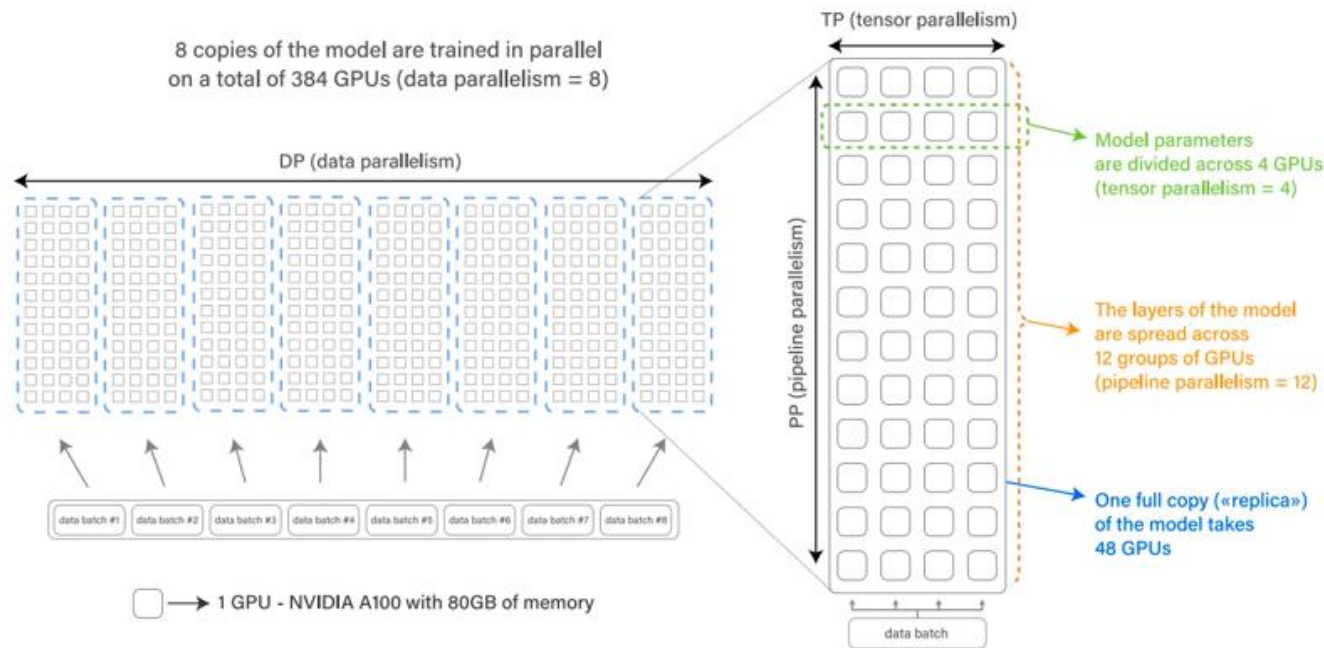


分布式训练 – 混合并行

混合并行将多种并行策略如数据并行、流水线并行和张量并行等混合使用。通过结合不同的并行策略，混合并行可以充分发挥各种并行策略的优点，最大程度地提高计算性能和效率。

- 针对千亿规模的大语言模型，通常，**在每个服务器内部使用张量并行策略**，由于该策略涉及的网络通信量较大，**需要利用服务器内部的不同计算设备之间的高速通信带宽**。
- **通过流水线并行**，将模型的不同层划分为多个阶段，每个阶段由不同的机器负责计算。这样可以充分利用多台机器的计算能力，并通过机器之间的高速通信传递计算结果和中间数据，以提高整体的计算速度和效率。
- 最后，**在外层叠加数据并行策略**，以增加并发数量，加快整体训练速度。通过数据并行，将训练数据分发到多组服务器上进行处理，每组服务器处理不同的数据批次。这样可以充分利用多台服务器的计算资源，并增加训练的并发度，从而加快整体训练速度。

BLOOM 使用Megatron-DeepSpeed 框架进行训练，主要包含两个部分：Megatron-LM提供张量并行能力和数据加载原语；DeepSpeed 提供ZeRO 优化器、模型流水线及常规的分布式训练组件。通过这种方式可以实现数据、张量和流水线三维并行。

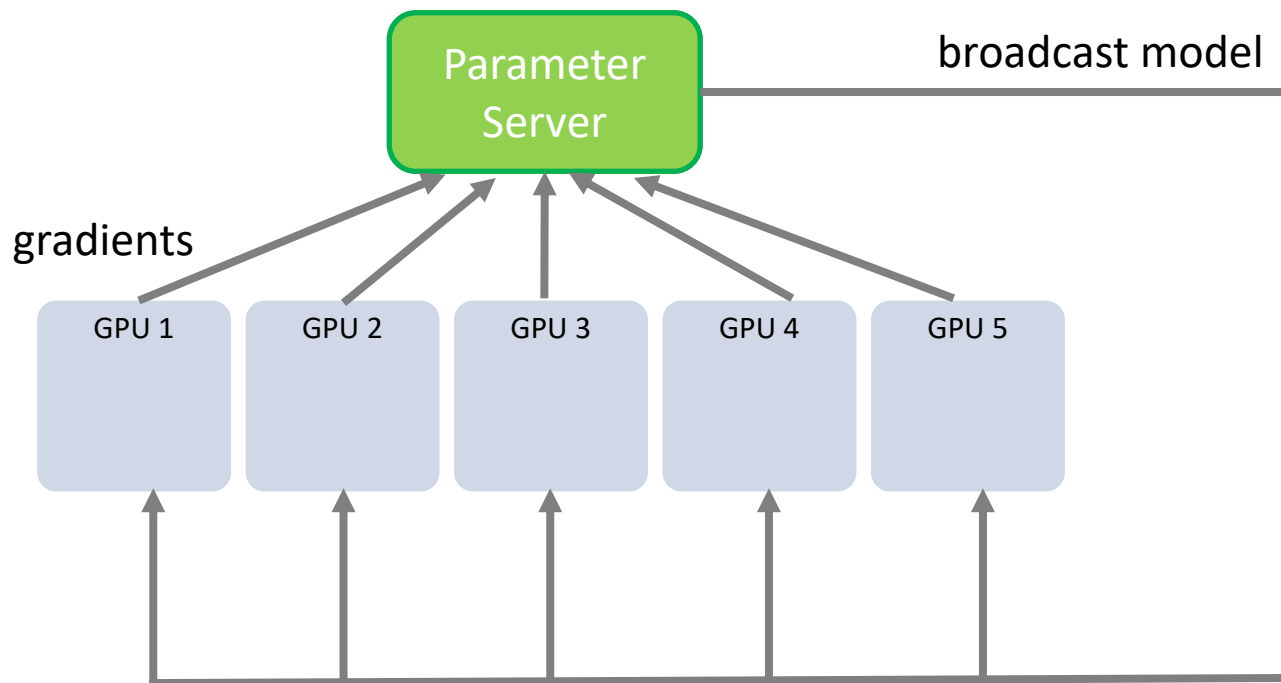


分布式训练的集群架构 - 集中式参数服务器架构

在分布式环境中，可能会有多个独立运行的随机梯度下降（SGD）实例。因此，整体算法必须进行调整，并考虑与模型一致性或参数分布相关的不同问题。

在每次迭代结束时，必须将计算得到的每个节点上的模型参数传播到所有其他节点，以保持模型的一致性。

集中式方案通常包括所谓的参数服务器策略。当并行随机梯度下降（SGD）使用参数服务器时，算法首先将模型广播给工作节点（服务器）。每个工作节点在每个训练迭代中读取自己的小批量数据，并计算自己的梯度，然后将这些梯度发送给一个或多个参数服务器。参数服务器汇总来自工作节点的所有梯度，并等待所有工作节点完成后计算下一次迭代的新模型，然后将其广播给所有工作节点。

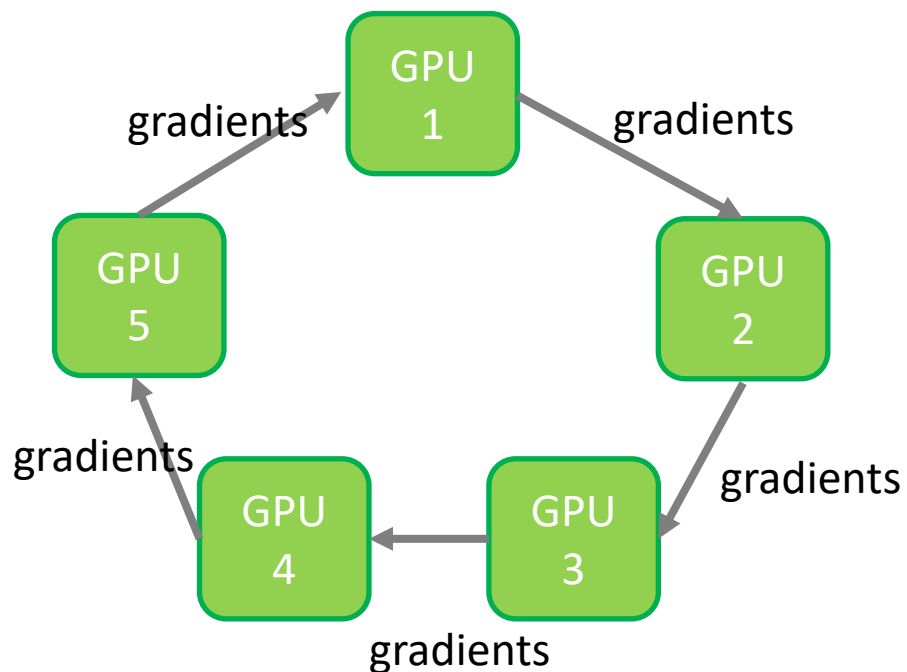


分布式训练的集群架构 - 去中心化架构

去中心化架构则采用集合通信实现分布式训练系统。在去中心化架构中，没有中央服务器或控制节点，而是由节点之间进行直接通信和协调。这种架构的好处是可以减少通信瓶颈，提高系统的可扩展性。

在分布式训练过程中，节点之间需要周期性地交换参数更新和梯度信息。可以通过**集合通信**技术实现，常用通信原语包括：

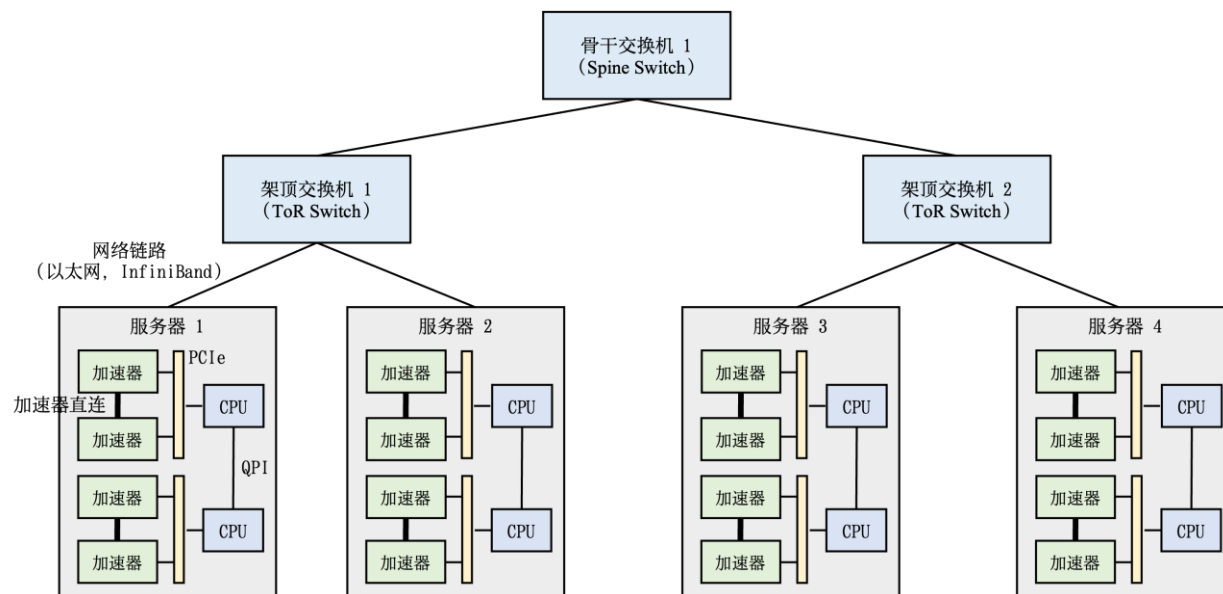
Broadcast、Scatter、Reduce、All Reduce、Gather、All Gather、Reduce Scatter、All to All 等。



高性能计算集群的典型硬件组成

典型的用于分布式训练的高性能计算集群的硬件组成如图所示。整个计算集群包含大量带有计算加速设备的服务器。

- 每个服务器中往往有多个计算加速设备（通常为2~16 个）
- 多个服务器会被放置在一个机柜（Rack）中
- 服务器通过架顶交换机（Top of Rack Switch, ToR）连接网络
- 通过在架顶交换机间增加骨干交换机（Spine Switch）接入新的机柜
- 这种连接服务器的拓扑结构往往是一个多层树（Multi-Level Tree）




技术选型 – 模型 – 微调技术

- 因为大模型的参数量非常大，从头训练一个自己的大模型训练成本非常高；
- 提示词工程的效果达不到要求，通过自有数据微调，更好的提升大模型在特定领域的能力；
- 训练一个轻量级的微调模型，提升特定业务场景个性化服务能力；
- 数据安全的问题；

大模型微调技术	技术特点	参数组合形式	缺点	优点
Adapter Tuning	固定原参数，微调Adapter结构	相加式	增加了模型层数，引入了额外的推理延迟	需训练参数规模小
Prefix Tuning	利用前缀训练	门控式	难于训练，挤占下游任务的输入序列空间，影响模型性能	具有学习能力的提示词
P-Tuning v2	加入更多token	门控式	容易导致旧知识遗忘，微调后在之前的问题上表现明显变差	在小模型上也有较好效果
     LoRA	低秩自适应，计算代价低	缩放式	使用低精度权重，准确率受影响	计算高效，显著降低计算资源代价
   Q-LoRA	针对量化优化，保留高精度权重特征	缩放式	微调过程中需要高低精度转换，可能导致精度损失，缺少加速优化	既减少模型的内存占用，同时保留训练的基本精度
QA-LoRA	将权重矩阵分组为更小的段，并对每个组单独应用量化和低秩自适应	缩放式	/	融合量化和低秩适应的好处，同时保持过程高效和模型对所需任务的有效性

模型 – 微调框架 - DeepSpeed

微调框架	技术特点	技术支持	Github
 DeepSpeed	<ul style="list-style-type: none"> 结合Ray、HuggingFace训练模型 被用于高效地微调大模型，并且能使用多节点获得最高性价比而不带来额外的复杂度； 	微软	https://github.com/microsoft/DeepSpeed

安装DeepSpeed

```
pip install deepspeed
```

以ChatGLM 为例：
运行脚本，指定模型和数据集：

```
ds_train_finetune.sh
evaluate_finetune.sh
```

硬件要求	ChatGLM-6B
推理	任意8GB显存以上GPU * 1
微调	A100 (80G) * 4 A100 (40G) * 8 V100 (32G) * 12

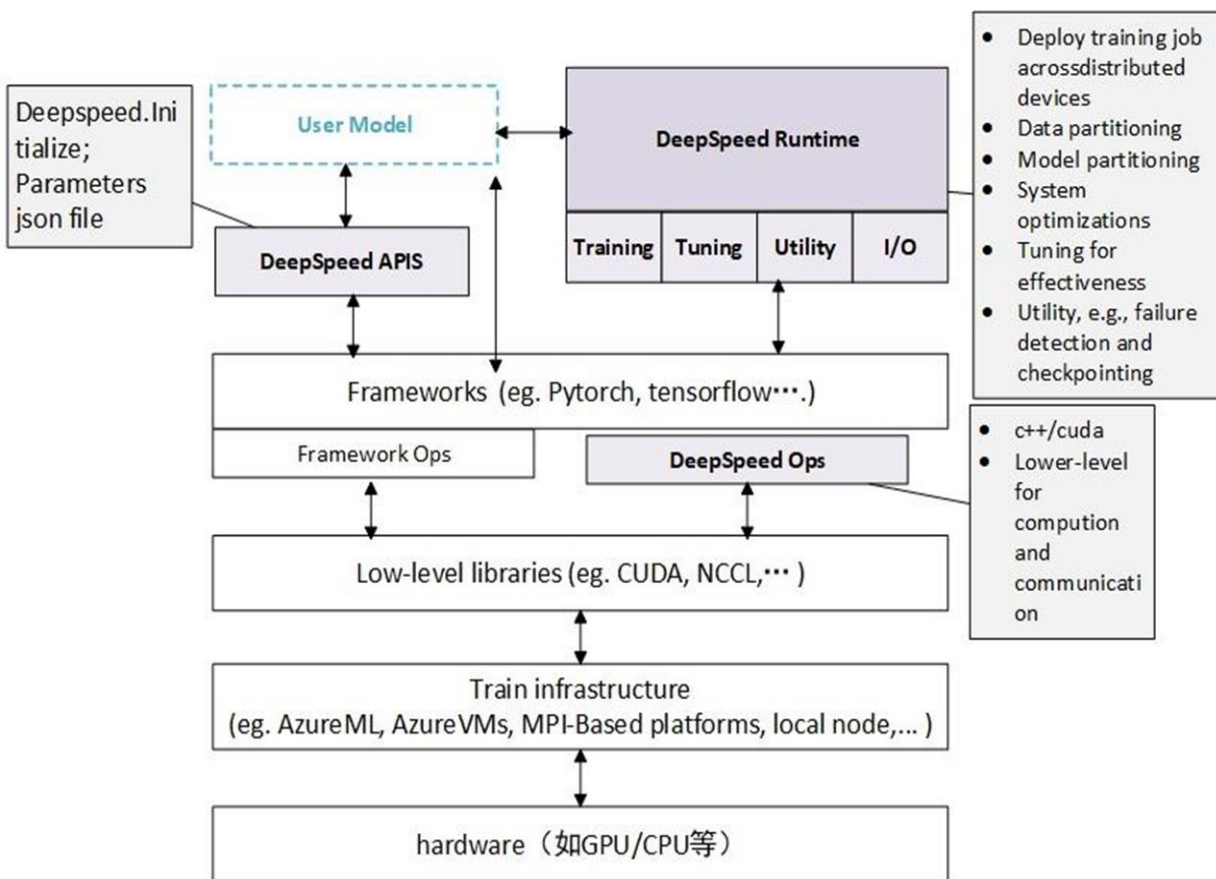
```
deepspeed --num_gpus=4 --master_port $MASTER_PORT main.py \
--deepspeed deepspeed.json \
...
--per_device_train_batch_size 4 \
--gradient_accumulation_steps 1 \
...
--learning_rate 1e-4 \
--fp16
```

模型 – 微调框架 - DeepSpeed



DeepSpeed是一个由微软开发的开源深度学习优化库，旨在**提高大规模模型训练的效率和可扩展性**。它通过多种技术手段来加速训练，包括模型并行化、梯度累积、动态精度缩放、本地模式混合精度等。DeepSpeed还提供了一些辅助工具，如分布式训练管理、内存优化和模型压缩等，以帮助开发者更好地管理和优化大规模深度学习训练任务。此外，deepspeed 基于pytorch构建，只需要简单修改即可迁移。

DeepSpeed 软件架构如图所示，主要包含以下三部分。

- **API**: DeepSpeed 提供了易于使用的API 接口，简化了训练模型和推断的过程。用户只需通过调用几个API 接口即可完成任务。
- **RunTime**: DeepSpeed 的核心运行时组件，使用Python 语言实现，负责管理、执行和优化性能。它承担了将训练任务部署到分布式设备的功能，包括数据分区、模型分区、系统优化、微调、故障检测及检查点的保存和加载等任务。
- **Ops**: DeepSpeed 的底层内核组件，使用C++ 和CUDA 实现。它优化计算和通信过程，提供了一系列底层操作。Ops 的目标是通过高效的计算和通信加速深度学习训练过程。

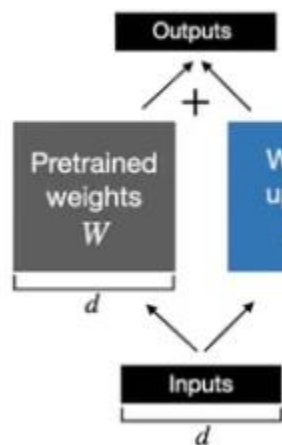


技术选型 – 模型 – LoRA微调技术

大模型微调技术	技术特点	参数组合形式	缺点	优点
 LoRA	低秩自适应，计算代价低	缩放式	使用低精度权重，准确率受影响	计算高效，显著降低计算资源代价
 Q-LoRA	针对量化优化，保留高精度权重特征	缩放式	微调过程中需要高低精度转换，可能导致精度损失，缺少加速优化	既减少模型的内存占用，同时保留训练的基本精度
QA-LoRA	将权重矩阵分组为更小的段，并对每个组单独应用量化和低秩自适应	缩放式	/	融合量化和低秩适应的好处，同时保持过程高效和模型对所需任务的有效性

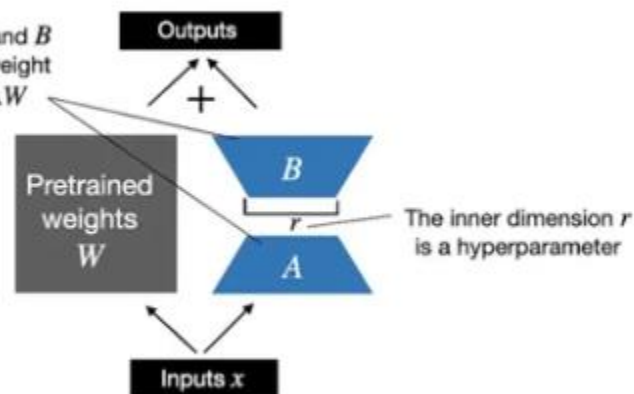
以ChatGLM 为例：
需要显存 15GB GPU

Weight update in regular finetuning



LoRA matrices A and B approximate the weight update matrix ΔW

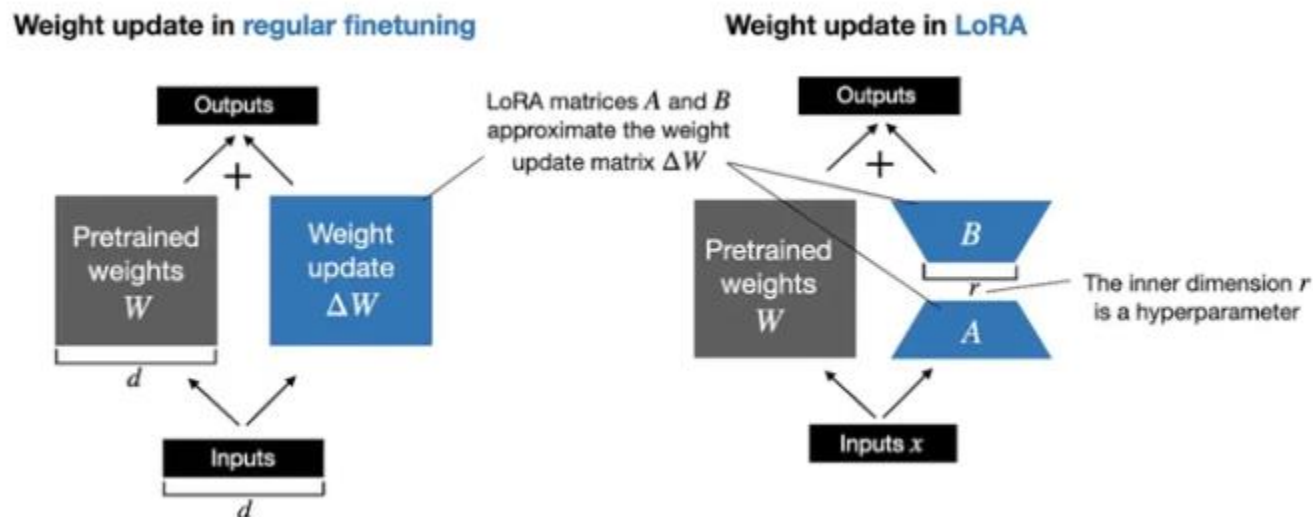
Weight update in LoRA



高效模型微调 - LoRA

由于大语言模型的参数量十分庞大，当将其应用到下游任务时，微调全部参数需要相当高的算力。为了节省成本，研究人员提出了多种参数高效（Parameter Efficient）的微调方法，旨在仅训练少量参数使模型适应到下游任务。LoRA 方法可以在缩减训练参数量和GPU 显存占用的同时，使训练后的模型具有与全量微调相当的性能。

语言模型针对特定任务微调之后，权重矩阵通常具有很低的本征秩（Intrinsic Rank）。研究人员认为，参数更新量即便投影到较小的子空间中，也不会影响学习的有效性。因此，提出**固定预训练模型参数不变，在原本权重矩阵旁路添加低秩矩阵的乘积作为可训练参数**，用以模拟参数的变化量。



LoRA 算法不仅在RoBERTa、DeBERTa、GPT-3 等大语言模型上取得了很好的效果，也应用到了Stable Diffusion 等视觉大模型中，同样也可以用很小的成本达到微调大语言模型的目的。LoRA 算法引起了企业界和研究界的广泛关注，研究人员又先后提出了AdaLoRA、QLoRA、IncreLoRA及LoRA-FA等算法。

高效模型微调 - LoRA

peft 库中含有包括LoRA 在内的多种高效微调方法，且与transformers 库兼容。get_peft_model 函数封装了基础模型并得到一个PeftModel 类的模型。如果使用LoRA 微调方法，则会得到一个LoraModel 类的模型。

LoraModel 类通过add_adapter 方法添加LoRA 层。该方法包括_find_and_replace mark_only_lora_as_trainable 两个主要函数。

```
class LoraModel(torch.nn.Module):
    """
    从预训练的Transformers模型创建低秩适配器 (Low Rank Adapter, Lora) 模型

    Args:
        model ([`~transformers.PreTrainedModel`]): 要适配的模型
        config ([`LoraConfig`]): Lora模型的配置

    Returns:
        `torch.nn.Module`: Lora模型

    **Attributes**:
        - **model** ([`~transformers.PreTrainedModel`]) -- 要适配的模型
        - **peft_config** ([`LoraConfig`]): Lora模型的配置
    """

    def __init__(self, model, config, adapter_name):
        super().__init__()
        self.model = model
        self.forward = self.model.forward
        self.peft_config = config
        self.add_adapter(adapter_name, self.peft_config[adapter_name])

    # Transformers模型具有一个`.config`属性，后续假定存在这个属性
    if not hasattr(self, "config"):
        self.config = {"model_type": "custom"}
```

```
def add_adapter(self, adapter_name, config=None):
    if config is not None:
        model_config = getattr(self.model, "config", {"model_type": "custom"})
        if hasattr(model_config, "to_dict"):
            model_config = model_config.to_dict()

        config = self._prepare_lora_config(config, model_config)
        self.peft_config[adapter_name] = config
    self._find_and_replace(adapter_name)
    if len(self.peft_config) > 1 and self.peft_config[adapter_name].bias != "none":
        raise ValueError(
            "LoraModel supports only 1 adapter with bias. When using multiple adapters, \
            set bias to 'none' for all adapters."
        )
    mark_only_lora_as_trainable(self.model, self.peft_config[adapter_name].bias)
    if self.peft_config[adapter_name].inference_mode:
        _freeze_adapter(self.model, adapter_name)
```

模型上下文窗口扩展

随着更多长文本建模需求的出现，多轮对话、长文档摘要等任务在实际应用中越来越多，这些任务需要模型能够更好地处理超出常规上下文窗口大小的文本内容。尽管当前的大语言模型在处理短文本方面表现出色，但在支持长文本建模方面仍存在一些挑战，这些挑战包括预定义的上下文窗口大小限制等。

以MetaAI 在2023 年2 月开源的LLaMA 模型为例，其规定输入文本的词元数量不得超过2048个。这会限制模型对长文本的理解和表达能力。当涉及长时间对话或长文档摘要时，传统的上下文窗口大小可能无法捕捉到全局语境，从而导致信息丢失或模糊的建模结果。

为了更好地满足长文本需求，有必要探索如何扩展现有的大语言模型，使其能够有效地处理更大范围的上下文信息。具体来说，主要有以下方法来扩展语言模型的长文本建模能力。

- **增加上下文窗口的微调：**采用直接的方式，即通过使用一个更大的上下文窗口来微调现有的预训练Transformer，以适应长文本建模需求。
- **位置编码：**改进的位置编码，如ALiBi[69]、LeX[157] 等能够实现一定程度上的长度外推。这意味着它们可以在小的上下文窗口上进行训练，在大的上下文窗口上进行推理。
- **插值法：**将超出上下文窗口的位置编码通过插值法压缩到预训练的上下文窗口中。

模型上下文窗口扩展 - 位置编码 - 具有外推能力的位置编码

位置编码的长度外推能力来源于位置编码中表征相对位置信息的部分，相对位置信息不同于绝对位置信息，对于训练时的依赖较少。位置编码的研究一直是基于Transformer 结构模型的重点。

2017 年Transformer 结构提出时，介绍了两种位置编码，一种是Naive Learned Position Embedding，也就是BERT 模型中使用的位置编码；另一种是Sinusoidal Position Embedding，通过正弦函数为每个位置向量提供一种独特的编码。这两种最初的形式**都是绝对位置编码的形式**，依赖于训练过程中的上下文窗口大小，在推理时基本不具有外推能力

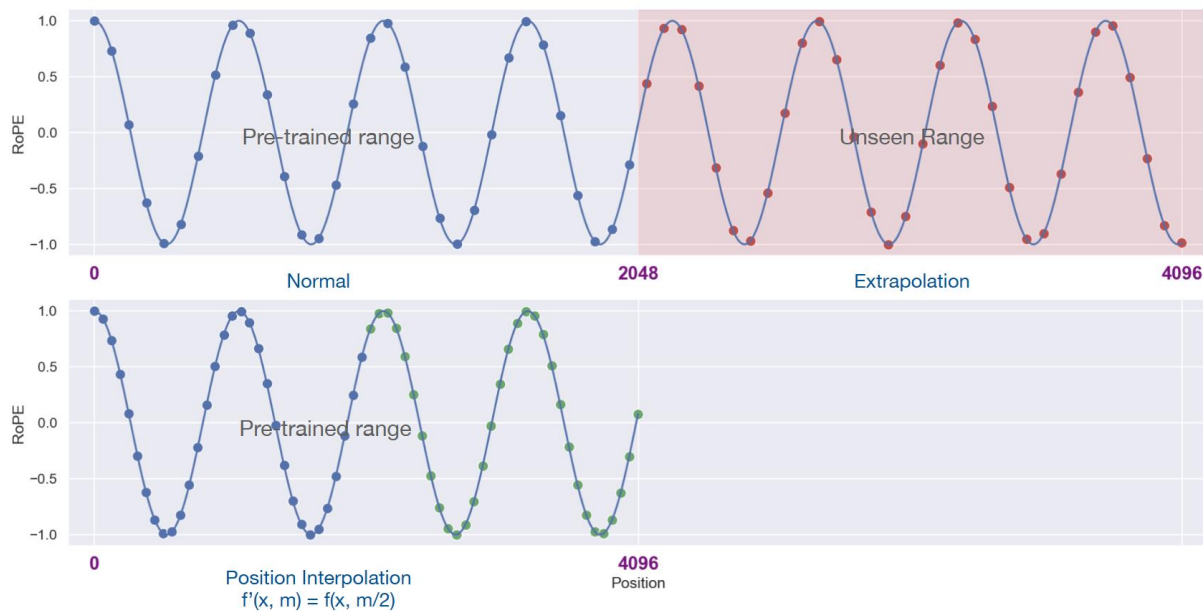
2021 年提出的Rotary Position Embedding (RoPE) 在一定程度上缓解了绝对位置编码外推能力弱的问题。

后续在T5 架构中，研究人员又提出了T5 Bias Position Embedding，直接在Attention Map 上操作，对于查询和键之间的不同距离，模型会学习一个偏置的标量值，将其加在注意力分数上，并在每一层都进行此操作，从而学习一个相对位置的编码信息。这种相对位置编码的外推性能较好，可以在512 的训练窗口上外推600 左右的长度。

受到T5 Bias 的启发，Press 等人提出了ALiBi算法，这是一种预定义的相对位置编码。ALiBi 并不在Embedding 层添加位置编码，而是在Softmax 的结果后添加一个静态的不可学习的偏置项。

模型上下文窗口扩展 - 插值法

不同的预训练大语言模型使用不同的位置编码，修改位置编码意味着重新训练，因此对于已训练的模型，通过修改位置编码扩展上下文窗口大小的适用性仍然有限。为了不改变模型架构而直接扩展大语言模型上下文窗口大小，**位置插值法**使现有的预训练大语言模型（包括LLaMA、Falcon、Baichuan 等）能直接扩展上下文窗口。其关键思想是，直接缩小位置索引，使最大位置索引与预训练阶段的上下文窗口限制相匹配。线性插值法如图所示。



参考材料

1. 张奇 桂韬 郑锐 黄萱菁 著 <<大语言模型：从理论到实践>>



Thank

You