10 大模型训练

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 理解大模型训练的主要挑战
- ⋆⋆⋆ 理解 LoRA 的原理
- ⋆⋆ 了解并行训练的思想：Data Parallel、Tensor Parallel 和 Pipeline Parallelism

10.1 大模型的训练挑战

随着深度学习模型规模的不断扩大，LLM 的训练面临日益复杂的计算与系统挑战。相比传统深度学习模型，大模型通常具有数十亿甚至数千亿参数，其训练过程对计算资源、存储资源以及通信效率均提出了更高要求。由于模型参数规模巨大，单个计算设备通常难以完成训练，因此需要依赖多 GPU 或 TPU 集群进行分布式训练。同时，模型参数、梯度以及中间激活值会占用大量显存，并带来显著的存储与内存带宽开销。此外，在数据并行与模型并行等训练范式中，不同计算设备之间需要频繁进行参数与梯度同步，通信延迟与带宽限制逐渐成为制约训练效率的关键瓶颈。另一方面，大模型训练通常具有较长的训练周期与较高的成本。即使采用高性能硬件与多种优化策略，完成一次大规模预训练仍可能需要数周甚至数月时间。因此，如何降低训练成本、提高训练效率以及减少显存占用，成为当前大模型训练研究的重要方向。

公司	模型	发布时间	参数量	上下文长度	训练 Tokens	训练规模与成本
OpenAI	GPT-3 [1]	2020	175B	2K	∼1T 级别	数千 GPU（V100/A100）；训练计算量约 $\sim 3 \times 10^{23}$ FLOPs（估计）；数百万美元级训练成本
Meta	LLaMA 3 [2]	2024	8B / 70B / 405B	8K–128K	∼10T 级别	万卡级 GPU 集群
DeepSeek	DeepSeek-V3 [3]	2024	671B（37B activated）	128K	∼10T 级别	千卡级 H800 GPU 集群；训练成本为数百万 GPU-hours 级别

表 10.1: 代表性大模型训练系统参数对比（公开信息与工程估计）

当前互联网数据中真正可用于大模型训练的高质量文本数据相对有限。研究表明 [4]，在经过清洗、去重、质量过滤以及版权筛查后，可用于大规模语言模型预训练的高质量公开文本数据规模大约为数十万亿（10–100T）词元量级。例如，FineWeb 数据集包含约 15T 英文词元[1]；RedPajama V2 数据集则达到约 30T 词元规模[2]。

本章主要从两类技术路径展开讨论：一类是参数高效微调方法，通过仅更新少量参数以降低模型微调的计算与存储开销；另一类是并行与分布式训练技术，通过数据并行、模型并行以及流水线并行等方式提升大模型训练效率。

[1] FineWeb: https://huggingface.co/datasets/HuggingFaceFW/fineweb.

[2] RedPajama V2: https://github.com/togethercomputer/RedPajama-Data.

10.2 参数高效微调

参数高效微调（Parameter-Efficient Fine-Tuning, PEFT）技术，通过仅训练少量额外参数，在保持模型性能的同时大幅降低资源消耗。本节主要讲基于 LoRA 的参数高效微调技术。本节主要讲 Low-Rank Adaptation（LoRA）这种方法。

基于如下经验观察：在下游任务微调过程中，预训练大模型权重的有效更新 ΔW 往往具有较低的内在秩（low intrinsic rank），即模型性能的提升主要依赖于少数几个主方向上的参数变化。因此，无需学习完整的 ΔW，而可以用低秩矩阵对其进行近似，从而显著减少可训练参数数量。

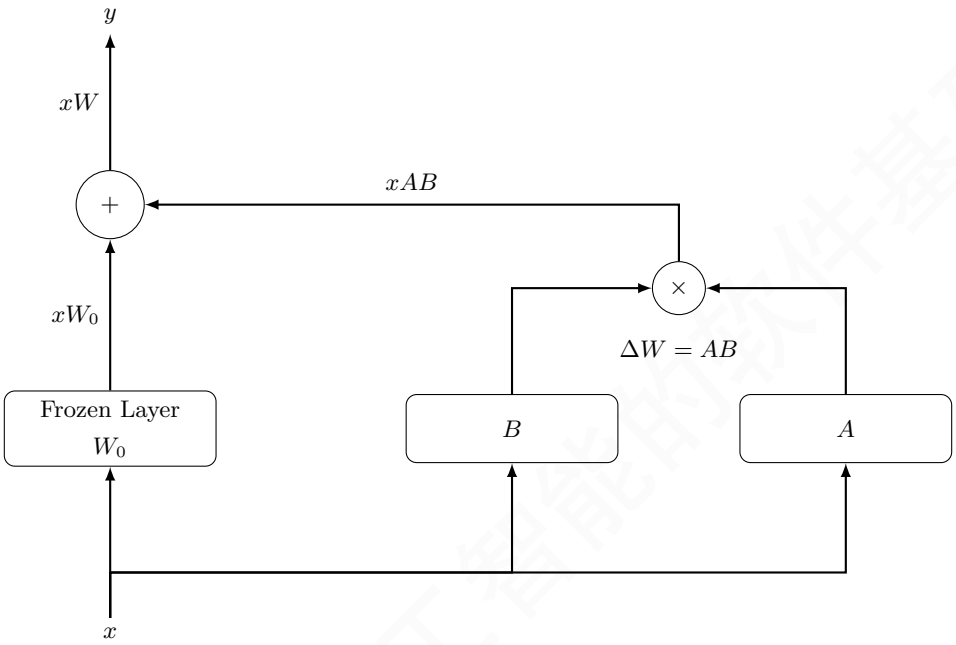

图 10.1: LoRA 原理示意图

如图 10.1所示，LoRA 不直接更新完整的权重矩阵 W，而是在冻结原始预训练参数 W_0 的基础上，引入两个低秩矩阵 A 与 B 来表示权重增量：

$$\Delta W = AB$$

其中，若输入向量为

$$x \in \mathbb{R}^{1\times d},$$

原始权重矩阵为

$$W_0 \in \mathbb{R}^{d\times k},$$

则通常令

$$A \in \mathbb{R}^{d\times r}, \qquad B \in \mathbb{R}^{r\times k},$$

其中 $r \ll \min(d,k)$ 为低秩维度（rank）。

因此：

$$\Delta W = AB \in \mathbb{R}^{d\times k},$$

新的权重可表示为：

$$W = W_0 + \Delta W = W_0 + AB.$$

在前向传播过程中，模型输出变为：

$$xW = xW_0 + xAB.$$

其中：

- xW_0 由冻结的预训练参数计算得到；
- xAB 表示 LoRA 学习到的低秩增量部分。

由于仅需训练矩阵 A 与 B，可训练参数数量从原本的

$$d \times k$$

降低为

$$r(d+k),$$

当 r 很小时，参数规模与显存占用都将显著下降。例如，当 $d = k = 4096$、$r = 8$ 时，可训练参数量仅约为原来的 0.76%。

LoRA 的另一个重要优点是：其不会修改原始预训练模型参数，因此不同任务对应的 LoRA 权重可以独立保存与切换。实际部署时，只需加载基础模型与对应的 LoRA 参数即可快速适配不同任务，从而大幅降低存储与部署成本。

在 Transformer 中，LoRA 通常应用于注意力机制中的线性投影层，例如 Query、Key、Value 或 Output Projection 等模块。其中最常见的是对 Query 与 Value 矩阵进行低秩适配，因为实验表明这些位置对下游任务性能最为敏感。

此外，为了稳定训练，LoRA 通常还会引入缩放系数 α，将权重更新写为：

$$W = W_0 + \frac{\alpha}{r} AB.$$

其中 α 用于控制低秩更新项的幅度，从而避免训练初期对原模型分布造成过大扰动。

总体而言，LoRA 通过“冻结大部分参数 + 仅学习低秩增量”的方式，在保持模型性能的同时，显著降低了微调过程中的参数量、显存消耗与训练成本，因此已成为当前大模型微调中最常用的参数高效微调方法之一。

10.3 并行和分布式训练

随着大语言模型参数规模不断增长，单张 GPU 已难以满足训练所需的显存与计算需求。例如，一个 7B 参数规模的 Transformer 模型，若采用 FP16 精度，仅模型参数本身就需要约：$7 \times 10^9 \times 2$ Bytes ≈ 14 GB。并且训练过程中还需要额外存储梯度和优化器状态等信息，因此实际训练显存往往会达到参数量的数倍。

名称	并行方式	说明
数据并行（Data Parallel）	训练数据	每张 GPU 持有完整模型副本，处理不同数据 batch
全分片数据并行（Fully Sharded Data Parallel）	存储参数	参数与梯度等信息在多 GPU 间分片存储，计算时按需聚合
张量并行（Tensor Parallel）	（横向）算子计算	将单层算子按张量维度切分到多 GPU 上并行计算
流水线并行（Pipeline Parallel）	（纵向）层级并行	将模型按层划分为多个 stage，不同 GPU 负责不同层的计算

表 10.2: 主流分布式训练并行方式对比

10.3.1 数据并行

在单机多卡场景中，如果单张 GPU 能够容纳完整模型参数，则可以通过数据并行加速训练。数据并行的基本思想是：每张 GPU 持有一份完整模型副本，并分别处理不同的 mini-batch 数据，从而提升整体训练吞吐量，缩短训练时间。

需要注意的是，数据并行并不能解决模型过大导致的显存不足问题，因为每张 GPU 仍需存储完整模型参数与优化器状态。

DataParallel（DP） PyTorch 提供的基础数据并行接口为 `torch.nn.DataParallel`，其使用方式如下：

```
model = torch.nn.DataParallel(model)
model = model.cuda()
```

其执行流程如下：

- 主 GPU 负责参数广播与结果汇总；
- 输入 mini-batch 被拆分并分发到多个 GPU；
- 各 GPU 独立执行前向与反向传播；
- 各 GPU 的梯度在主 GPU 上聚合后进行参数更新。

该方法实现简单、代码改动少，适合小规模实验。但由于存在主卡通信与汇总瓶颈，扩展性较差，因此已逐渐不被推荐用于正式训练任务。

Distributed Data Parallel（DDP） 目前 PyTorch 官方推荐使用 DDP 作为标准数据并行方式。DDP 基于 NVIDIA 的 NCCL（NVIDIA Collective Communications Library）后端实现高效 GPU 间通信，采用去中心化的 All-Reduce 机制进行梯度同步，从而避免传统参数服务器架构中的中心节点瓶颈，具有更好的扩展性与计算效率。

其典型训练流程如下：

```python
import os
import torch
import torch.nn as nn
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader
from torch.utils.data.distributed import DistributedSampler

# 1. 初始化分布式环境
dist.init_process_group(backend="nccl")

# 2. 绑定当前进程与GPU
local_rank = int(os.environ["LOCAL_RANK"])
torch.cuda.set_device(local_rank)
device = torch.device("cuda", local_rank)

# 3. 构建模型并封装DDP
model = LLM().to(device)
model = DDP(model, device_ids=[local_rank])

# 4. 数据划分
dataset = MyDataset()
sampler = DistributedSampler(dataset, shuffle=True)
loader = DataLoader(dataset, batch_size=32, sampler=sampler)

# 5. 优化器与损失函数
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
criterion = nn.CrossEntropyLoss()

# 6. 训练循环
for epoch in range(3):
    sampler.set_epoch(epoch)
    for x, y in loader:
        x, y = x.to(device), y.to(device)
        logits = model(x)
        loss = criterion(logits, y)
        optimizer.zero_grad()
        loss.backward()   # DDP 自动触发 All-Reduce 同步梯度
        optimizer.step()

# 7. 清理环境
dist.destroy_process_group()
```

通常通过 torchrun 启动多进程训练：

```
torchrun --nproc_per_node=2 train.py
```

其中：

- nproc_per_node 表示使用的 GPU 数量；
- 每张 GPU 对应一个独立进程；
- PyTorch 自动完成进程间通信初始化。

10.3.2 参数并行

在大语言模型训练中，模型参数规模通常远超单卡显存容量，因此无法通过标准的数据并行方式进行扩展。参数相关的并行方法本质上是通过切分模型中的张量降低单卡的存储或计算压力。但需要注意的是，不同方法切分的对象与目的并不相同。

在 Decoder-only Transformer 中，参数主要集中在 Self-Attention 和 Feed-Forward 网络的线性变换中，例如：

$$W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times (Hd_k)}$$

其中 H 表示 attention head 数量。这一结构使得模型天然具有“按 head 或按矩阵维度切分”的可能性。

Fully Sharded Data Parallel（FSDP） FSDP 的核心思想是对“参数存储”进行切分，而不是改变计算方式。在实现上，模型所有参数（包括 attention 与 FFN 层）会被 flatten 成连续向量，并在数据并行维度上均匀切分，使得每张 GPU 仅保存约 $\frac{1}{N}$ 的参数、梯度以及优化器状态。

在前向传播时，当某一层参与计算（例如 attention 层中的 Q/K/V 线性映射），其参数分片会通过 all-gather 临时恢复为完整矩阵：

$$q_i = x_i W_Q, \quad k_i = x_i W_K, \quad v_i = x_i W_V$$

计算完成后立即释放完整参数，仅保留本地分片。在反向传播过程中，梯度通过 reduce-scatter 操作同步并重新切分。

因此，FSDP 可以理解为：模型参数被按存储切开，计算时再临时拼回完整形式，计算完成后重新分片存储。其本质目标是降低显存占用，使模型可以在多 GPU 上存得下。

Tensor Parallelism（TP） 与 FSDP 不同，Tensor Parallel 并不以“存储切分”为核心，而是直接对计算过程进行切分。其基本思想是：将同一层中的矩阵运算拆分到不同 GPU 上独立计算，而不是先恢复完整参数再计算。

以 attention 中的线性映射为例，输出维度通常按 head 进行划分：

$$W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times (Hd_k)}$$

在 Tensor Parallel 中，这些矩阵可以沿 head 维度或列维度拆分为多个子矩阵，例如：

$$W_Q = [W_Q^{(1)}, W_Q^{(2)}, \ldots, W_Q^{(N)}]$$

每张 GPU 只负责计算对应部分：

$$q_i^{(n)} = x_i W_Q^{(n)}$$

最终再对各个 head 的输出进行拼接或聚合。

因此，Tensor Parallel 可以理解为：模型参数不再作为整体存在，而是直接参与分布式计算，每张 GPU 只完成一部分矩阵乘法。其本质目标是降低单层计算压力，使大矩阵乘法可以在多 GPU 上并行执行。

Pipeline Parallelism（PP） PP 从“模型深度”维度对模型进行划分。其核心思想是：将一个深层 Decoder-only Transformer 按层划分为多个连续的子网络（stage），并分配到不同 GPU 上执行。

以标准 Transformer 为例，模型由多层 Decoder Block 堆叠而成：

$$\text{Block}_1 \to \text{Block}_2 \to \cdots \to \text{Block}_N$$

在 Pipeline Parallel 中，这些层被划分为若干段，例如：

$$\{\text{Block}_1 \sim \text{Block}_k\}, \quad \{\text{Block}_{k+1} \sim \text{Block}_{2k}\}, \quad \cdots$$

每个 GPU 只负责其中一段网络的前向与反向计算。

在前向传播过程中，输入 token 表示首先进入第一段计算（如前几层 attention 与 FFN），随后其输出作为激活值传递给下一 GPU，并继续经过后续层处理，直到得到最终 logits。这一过程类似于工业生产中的流水线，因此称为“pipeline”。

10.3.3 实际训练中的典型组合

在当前主流大语言模型训练中，通常不会单独使用某一种并行方式，而是将多种技术组合使用，以同时缓解显存、计算与通信三方面的瓶颈。一般而言，DDP 或 FSDP 用于实现数据并行与参数状态分片，Tensor Parallel 用于切分单层矩阵计算以提升算子级并行度，Pipeline Parallel 用于将深层 Transformer 按层划分到不同 GPU，从而扩展模型在深度方向上的可训练规模。此外，还常配合多种系统级优化策略，例如 ZeRO 用于进一步降低优化器状态与梯度的显存占用，FP16/BF16 混合精度用于减少计算与存储开销，而 Gradient Checkpointing 则通过以计算换显存的方式降低前向激活的存储压力。总体而言，现代大语言模型训练已经不再是单纯的模型优化问题，而是一个高度系统化的分布式工程问题，其核心约束来自 GPU 间通信效率、显存层级管理、参数与计算的切分策略以及集群网络带宽等多个因素的综合权衡。

10.4 使用 Hugging Face 训练大模型

Hugging Face[3] 已成为当前自然语言处理乃至整个 AI 领域最重要的开源生态之一。它不仅提供数以万计的预训练模型与公开数据集，还构建了一套完整的大模型开发工具链，极大降低了大模型训练与部署门槛。

Hugging Face 生态主要由多个 Python 库组成，并与 PyTorch 深度集成：

- **transformers**：提供 Transformer 模型实现与预训练权重，支持 LLaMA 等主流模型；
- **datasets**：提供统一的数据集加载与预处理接口，支持流式读取与大规模数据处理；
- **tokenizers**：提供高性能 tokenizer 实现，用于文本与词元序列之间的转换；
- **peft**：提供参数高效微调（PEFT）方法，包括 LoRA、Prefix Tuning、Prompt Tuning 等；
- **Trainer**：提供高层训练接口，自动封装训练循环、梯度更新、mixed precision、checkpoint 保存以及 DDP、FSDP 等分布式训练功能。

Hugging Face 提供了一套端到端的大模型开发与训练框架，涵盖模型加载、数据处理、分布式训练、参数高效微调（PEFT）以及模型发布等环节。研究者与工程人员可以专注于模型结构与任务本身，而无需手动实现数据并行、混合精度训练、梯度检查点等底层机制。

[3]Hugging Face 在线平台：`https://huggingface.co`

下面以 LLaMA-3 模型为例，演示如何使用 Hugging Face 与 PEFT 对大语言模型进行 LoRA 微调。

```python
from transformers import AutoModelForCausalLM, AutoTokenizer, TrainingArguments, Trainer
from peft import LoraConfig, get_peft_model
from datasets import load_dataset
import torch

# AutoTokenizer: 自动加载与 LLaMA-3 对应的 tokenizer
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B")
# AutoModelForCausalLM: 自动加载 Decoder-only Transformer
model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B",
    torch_dtype=torch.bfloat16 # BF16 混合精度降低显存与计算开销
)
# 配置 LoRA (Low-Rank Adaptation)
# r: 低秩矩阵 rank, 决定新增参数规模
# lora_alpha: LoRA 缩放系数, 用于控制低秩更新幅度
# target_modules: 指定在哪些线性层插入 LoRA adapter
peft_config = LoraConfig(r=8, lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    task_type="CAUSAL_LM"
)
# 注入 LoRA adapter: 原始预训练参数默认被冻结, 仅 LoRA 参数参与训练
model = get_peft_model(model, peft_config)
# 输出可训练参数比例
model.print_trainable_parameters()

# 从 Hugging Face Hub 加载 instruction tuning 数据集
dataset = load_dataset("databricks/databricks-dolly-15k")

# 训练参数
training_args = TrainingArguments(
    output_dir="./llama3-lora",
    per_device_train_batch_size=1, # 单张 GPU 的 batch size
    gradient_accumulation_steps=8,  # 梯度累积: 每 8 个 step 更新一次参数, 模拟更大的 batch size
    bf16=True, # 启用 BF16 混合精度训练
    gradient_checkpointing=True, # backward 时重新计算部分 activation
    logging_steps=10,
    save_steps=500,
    num_train_epochs=1
)

# Hugging Face 高层训练接口
trainer = Trainer(model=model, args=training_args, train_dataset=dataset["train"])
# 启动训练
trainer.train()
# 保存 LoRA adapter (仅包含低秩增量参数)
model.save_pretrained("./llama3-lora-adapter")
# merge LoRA 权重回原始模型
merged_model = model.merge_and_unload()
# 保存 merge 后的完整模型
merged_model.save_pretrained("./llama3-lora-merged")
```

代码 10.1: 使用 Hugging Face + PEFT 对 LLaMA-3 进行 LoRA 微调

上述代码展示了当前主流的大模型参数高效微调流程。由于 LoRA 仅训练少量低秩参数，因此相比全参数微调，其显存占用与训练成本都会显著降低。

使用 FSDP 在 TrainingArguments 中增加如下配置：

```
training_args = TrainingArguments(
    ...
    # FSDP: 对参数、梯度与 optimizer state 进行分片存储
    fsdp="full_shard auto_wrap",
    # 自动对 Transformer Layer 进行 FSDP 包装
    fsdp_transformer_layer_cls_to_wrap="LlamaDecoderLayer",
)
```

代码 10.2: 在 Hugging Face Trainer 中启用 FSDP

通常可通过如下命令启动多 GPU 分布式训练：

```
torchrun --nproc_per_node=4 train.py
```

使用 TP TP 在 Hugging Face 的 Trainer 中通常不直接支持，而是依赖底层大规模分布式训练框架实现，例如微软的 DeepSpeed[4] 或 NVIDIA 的 Megatron-LM[5]。以 DeepSpeed 为例，在实际使用中通常通过配置文件启用对应的并行策略，而无需对模型结构进行手动修改。例如：

```
training_args = TrainingArguments(
    ...
    deepspeed="ds_config.json"
)
```

使用 PP 在 Hugging Face 中可以直接通过设备映射方式将不同 Transformer 层分配到不同 GPU 上执行。例如：

```
from accelerate import dispatch_model
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B")

device_map = {
    "model.layers.0": 0,
    "model.layers.1": 0,
    "model.layers.2": 1,
    "model.layers.3": 1
}

model = dispatch_model(model, device_map=device_map)
```

[4]https://github.com/deepspeedai/deepspeed

[5]https://github.com/nvidia/megatron-lm

参考文献

[1] Tom Brown et al. “Language Models are Few-Shot Learners.” *NeurIPS*, 2020.

[2] AI at Meta. “The LLaMA 3 Herd of Models.” *arXiv preprint arXiv:2407.21783*, 2024.

[3] Aixin Liu et al. “DeepSeek-V3 Technical Report.” *arXiv preprint arXiv:2412.19437*, 2024.

[4] Villalobos, et al. “Position: Will we run out of data? Limits of LLM scaling based on human-generated data.” *ICML*, 2024.

练习

1) 使用 PyTorch DDP 对 LeNet-5 在 MNIST 数据集上进行分布式训练，对比不同 GPU 数量下的训练时间与模型准确率。

2) 使用 Hugging Face 与 PEFT 框架，对 LLaMA-3-8B 或其它开源 Transformer 模型进行 LoRA 微调，尝试修改 LoRA rank（如 $r = 4, 8, 16$），并对比不同 rank 下的可训练时间、显存占用以及模型效果。