



大语言模型理论与实践

张奇 桂韬 郑锐 黄萱菁

2023 年 6 月 27 日

数学符号

数与数组

α	标量
α	向量
A	矩阵
\mathbf{A}	张量
I_n	n 行 n 列单位矩阵
v_w	单词 w 的分布式向量表示
e_w	单词 w 的独热向量表示: $[0,0,\dots,1,0,\dots,0]$, w 下标处元素为 1

索引 |

α_i	向量 α 中索引 i 处的元素
α_{-i}	向量 α 中除索引 i 之外的元素
$w_{i:j}$	序列 w 中从第 i 个元素到第 j 个元素组成的片段或子序列
A_{ij}	矩阵 A 中第 i 行、第 j 列处的元素
$A_{i:}$	矩阵 A 中第 i 行
$A_{:j}$	矩阵 A 中第 j 列
A_{ijk}	三维张量 \mathbf{A} 中索引为 (i, j, k) 处元素
$\mathbf{A}_{::i}$	三维张量 \mathbf{A} 中的一个二维切片

集合

\mathbb{A}	集合
\mathbb{R}	实数集
\mathbb{C}	复数集
$\{0, 1, \dots, n\}$	含 0 和 n 的正整数的集合
$[a, b]$	a 到 b 的实数闭区间
$(a, b]$	a 到 b 的实数左开右闭区间

线性代数

\mathbf{A}^\top	矩阵 \mathbf{A} 的转置
$\mathbf{A} \odot \mathbf{B}$	矩阵 \mathbf{A} 与矩阵 \mathbf{B} 的 Hadamard 乘积
$\det(\mathbf{A})$	矩阵 \mathbf{A} 的行列式
$[\mathbf{x}; \mathbf{y}]$	向量 \mathbf{x} 与 \mathbf{y} 的拼接
$[\mathbf{U}; \mathbf{V}]$	矩阵 \mathbf{A} 与 \mathbf{V} 沿行向量拼接
$\mathbf{x} \cdot \mathbf{y}$ 或 $\mathbf{x}^\top \mathbf{y}$	向量 \mathbf{x} 与 \mathbf{y} 的点积

微积分

$\frac{dy}{dx}$	y 对 x 的导数
$\frac{\partial y}{\partial x}$	y 对 x 的偏导数
$\nabla_{\mathbf{x}} y$	y 对向量 \mathbf{x} 的梯度
$\nabla_{\mathbf{X}} y$	y 对矩阵 \mathbf{X} 的梯度
$\nabla_{\mathbf{X}} y$	y 对张量 \mathbf{X} 的梯度

概率与信息论

$a \perp b$	随机变量 a 与 b 独立
$a \perp b \mid c$	随机变量 a 与 b 关于 c 条件独立
$P(a)$	离散变量概率分布
$p(a)$	连续变量概率分布
$a \sim P$	随机变量 a 服从分布 P
$\mathbb{E}_{x \sim P}(f(x))$ 或 $\mathbb{E}(f(x))$	$f(x)$ 在分布 $P(x)$ 下的期望
$\text{Var}(f(x))$	$f(x)$ 在分布 $P(x)$ 下的方差
$\text{Cov}(f(x), g(x))$	$f(x)$ 与 $g(x)$ 在分布 $P(x)$ 下的协方差
$H(f(x))$	随机变量 x 的信息熵
$D_{KL}(P \parallel Q)$	概率分布 P 与 Q 的 KL 散度
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	均值为 $\boldsymbol{\mu}$ 、协方差为 $\boldsymbol{\Sigma}$ 的高斯分布

数据与概率分布

\mathbb{X} 或 \mathbb{D}	数据集
$x^{(i)}$	数据集中第 i 个样本（输入）
$y^{(i)}$ 或 $y^{(i)}$	第 i 个样本 $x^{(i)}$ 的标签（输出）

函数

$f : \mathcal{A} \rightarrow \mathcal{B}$	由定义域 \mathcal{A} 到值域 \mathcal{B} 的函数（映射） f
$f \circ g$	f 与 g 的复合函数
$f(\mathbf{x}; \boldsymbol{\theta})$	由参数 $\boldsymbol{\theta}$ 定义的关于 \mathbf{x} 的函数（也可以直接写作 $f(\mathbf{x})$ ，省略 $\boldsymbol{\theta}$ ）
$\log x$	x 的自然对数函数
$\sigma(x)$	Sigmoid 函数 $\frac{1}{1 + \exp(-x)}$
$\ \mathbf{x}\ _p$	\mathbf{x} 的 L^p 范数
$\ \mathbf{x}\ $	\mathbf{x} 的 L^2 范数
$\mathbf{1}^{\text{condition}}$	条件指示函数：如果 condition 为真，则值为 1；否则值为 0

本书中常用写法

- 给定词表 \mathbb{V} ，其大小为 $|\mathbb{V}|$
- 序列 $\mathbf{x} = x_1, x_2, \dots, x_n$ 中第 i 个单词 x_i 的词向量 \mathbf{v}_{x_i}
- 损失函数 \mathcal{L} 为负对数似然函数： $\mathcal{L}(\boldsymbol{\theta}) = -\sum_{(x,y)} \log P(y|x_1 \dots x_n)$
- 算法的空间复杂度为 $\mathcal{O}(mn)$

目 录

1 絮论	1
1.1 大语言模型基本概念	1
1.2 大语言模型发展历程	3
1.3 大语言模型构建流程	5
1.4 本书的内容安排	10
2 大语言模型基础	12
2.1 语言模型概述	12
2.2 Transformer 模型	14
2.2.1 嵌入表示层	14
2.2.2 自注意力层	16
2.2.3 前馈层	18
2.2.4 残差连接与层归一化	19
2.2.5 编码器和解码器结构	20
2.3 预训练语言模型	24
2.3.1 掩码预训练语言模型 BERT	25
2.3.2 生成式预训练语言模型 GPT	28
2.3.3 序列到序列预训练语言模型 BART	30
2.3.4 基于 HuggingFace 预训练语言模型实践	32
2.4 大语言模型网络结构	37
2.4.1 LLaMA 的模型结构	38
2.4.2 注意力机制优化	43
3 预训练数据	49
3.1 数据集介绍	49
3.2 数据选择	49

4 分布式训练	50
4.1 分布式训练概述	50
4.2 分布式训练并行策略	53
4.2.1 数据并行	54
4.2.2 模型并行	57
4.2.3 混合并行	65
4.2.4 计算设备内存优化	65
4.3 分布式训练的集群架构	70
4.3.1 参数服务器架构	71
4.3.2 去中心化架构	73
4.4 DeepSpeed 实践	78
4.4.1 基础概念	80
4.4.2 GPT 分布式训练实践	83
5 有监督微调	84
5.1 有监督微调概述	84
5.2 任务范式统一	84
5.3 提示学习与上下文学习	84
5.4 提示数据构造	84
5.5 高效模型微调	85
5.6 Deepspeed-Chat SFT 实践	85
6 强化学习	86
6.1 强化学习理论	86
6.2 奖励模型	86
6.3 近端策略优化	86
6.4 Deepspeed-Chat PPO 实践	86
7 大语言模型应用	87
7.1 领域模型构建	87
7.2 大模型高效推理	87
7.3 LangChain	87
7.4 AutoGPT	87

8 大语言模型评价	88
8.1 语言模型评价	88
8.2 大语言模型评价	89

1. 绪论

大语言模型（Large Language Models, LLM），也称大型语言模型，是一种由包含数百亿以上权重的深度神经网络构建的语言模型，使用自监督学习方法通过大量无标记文本进行训练。自 2018 年以来，包含 Google、OpenAI、Meta、百度、华为等公司和研究机构都纷纷发布了包括 BERT^[1]，GPT^[2] 等在内的多种模型，并在几乎所有自然语言处理任务中都表现出色。2019 年开始大模型呈现爆发式的增长，特别是 2022 年 11 月 ChatGPT（Chat Generative Pre-trained Transformer）发布后，更是引起了全世界的广泛关注。用户可以使用自然语言与系统交互，从而实现包括问答、分类、摘要、翻译、聊天等从理解到生成的各种任务。大型语言模型展现出了强大的对世界知识掌握和对语言的理解。

本章主要介绍大型语言模型基本概念、发展历程和构建流程。

1.1 大语言模型基本概念

语言是人类与其他动物最重要的区别，而人类的多种智能也与此密切相关。逻辑思维以语言的形式表达，大量的知识也以文字的形式记录和传播。如今，互联网上已经拥有数万亿以上的网页资源，其中大部分信息都是以自然语言描述的。因此，如果人工智能想要获取知识，就必须懂得如何理解人类使用的不太精确、可能有歧义、混乱的语言。语言模型（Language Model, LM）目标就是建模自然语言的概率分布。词汇表 V 上的语言模型，由函数 $P(w_1 w_2 \dots w_m)$ 表示，可以形式化的为构建词序列 $w_1 w_2 \dots w_m$ 的概率分布，表示词序列 $w_1 w_2 \dots w_m$ 作为一个句子出现的可能性大小。由于联合概率 $P(w_1 w_2 \dots w_m)$ 的参数量十分巨大，直接计算 $P(w_1 w_2 \dots w_m)$ 非常困难。

为了减少 $P(w_1 w_2 \dots w_m)$ 模型的参数空间，可以利用句子序列通常情况下从左至右的生成过程进行分解，使用链式法则得到：

$$\begin{aligned} P(w_1 w_2 \dots w_m) &= P(w_1)P(w_2|w_1)P(w_3|w_1 w_2) \cdots P(w_m|w_1 w_2 \dots w_{m-1}) \\ &= \prod_{i=1}^m P(w_i|w_1 w_2 \cdots w_{i-1}) \end{aligned} \tag{1.1}$$

由此， $w_1 w_2 \dots w_m$ 的生成过程可以看作单词逐个生成的过程。首先生成 w_1 ，之后根据 w_1 生成 w_2 ，

2 自然语言处理导论 -- 张奇、桂韬、黄萱菁

再根据 w_1 和 w_2 生成 w_3 , 以此类推, 根据前 $m - 1$ 个单词生成最后一个单词 w_m 。例如: 对于句子 “把努力变成一种习惯”的概率计算, 使用公式1.1可以转化为:

$$\begin{aligned} P(\text{把努力变成一种习惯}) &= P(\text{把}) \times P(\text{努力|把}) \times P(\text{变成|把努力}) \times \\ &\quad P(\text{一种|把努力变成}) \times P(\text{习惯|把努力变成一种}) \end{aligned} \quad (1.2)$$

通过上述过程将联合概率 $P(w_1 w_2 \dots w_m)$ 转换为了多个条件概率的乘积。但是, 仅通过上述过程模型的参数空间依然没有下降, $P(w_m | w_1 w_2 \dots w_{m-1})$ 的参数空间依然是天文数字。然而基于上述转换, 可以进一步的对模型进行简化, n 元语言模型就是其中一种常见的简化方法。很多基于统计的概率平滑技术 (Smoothing) 方法也用来解决 n 元语言模型中的零概率问题。这类方法通常称为统计语言模型 (Statistical Language models, SLM)。

由于高阶 n 元语言模型仍然会面临十分严重的数据稀疏问题, 并且单词的离散表示也忽略了单词之间的相似性。因此, 基于分布式表示和神经网络的语言模型逐渐成为了新的研究热点。Bengio 等人在 2000 年提出了使用前馈神经网络对 $P(w_i | w_{i-n+1} \dots w_{i-1})$ 进行估计的语言模型^[3]。此后, 循环神经网络^[4]、卷积神经网络^[5]、端到端记忆网络^[6]等神经网络方法都成功应用于语言模型建模。相较于 n 元语言模型, 神经网络方法可以在一定程度上避免数据稀疏问题, 有些模型还可以避免对历史长度的限制, 从而更好的建模长距离依赖关系。这类方法通常称为神经语言模型 (Neural Language Models, NLM)。

深度神经网络需要采用有监督方法以来标注数据进行训练, 语言模型的训练过程也不可避免的需要构造训练语料, 但是由于训练目标可以通过无标注文本直接获得, 从而使得模型的训练仅需要大规模无标注文本即可。语言模型也成为了典型的自监督学习 (Self-supervised Learning) 任务。互联网的发展使得大规模无标注文本非常容易获取, 因此训练超大规模的基于神经网络的语言模型成为了可能。

受到计算机视觉领域采用 ImageNet^[7] 对模型进行一次预选训练, 使得模型可以通过海量图像充分学习如何提取特征, 然后再根据任务目标进行模型精调的范式影响, 自然语言处理领域基于预训练语言模型的方法也逐渐成为主流。以 ELMo^[8] 为代表的动态词向量模型开启了语言模型预训练的大门, 此后以 GPT^[9] 和 BERT^[1] 为代表的基于 Transformer 模型^[10] 的大规模预训练语言模型的出现, 使得自然语言处理全面进入了预训练微调范式新时代。将预训练模型应用于下游任务时, 不需要了解太多的任务细节, 不需要设计特定的神经网络结构, 只需要“微调”预训练模型, 即使用具体任务的标注数据在预训练语言模型上进行监督训练, 就可以取得显著的性能提升。这类方法通常称为预训练语言模型 (Pre-trained Language Models, PLM)。

2021 年 Open AI 发布了包含 1750 亿参数的生成式大规模预训练语言模型 GPT 3 (Generative Pre-trained Transformer 3)^[11]。开启了大语言模型的时代。由于大语言模型的参数量巨大, 如果在不同任务上都进行微调需要消耗大量的计算资源, 因此预训练微调范式不再适用于大语言模型。但是研究人员发现, 通过语境学习 (In-Context Learning, ICL) 等方法, 直接使用大语言模型就可以

在很多任务的少样本场景下取得了很好的效果。此后，研究员们提出了面向大语言模型的提示词（Prompt）的学习方法、模型即服务范式（Model as a Service, MaaS）、指令微调（Instruction Fine-tuning）等方法，在不同任务上都取得了很好的效果。与此同时，Google、Meta、百度、华为等公司和研究机构都纷纷发布了包括 PaLM^[12]、LaMDA^[13]、T0^[14] 等为代表的不同大型语言模型。2022 年底 ChatGPT 的出现，将大语言模型的能力进行了充分的展现，也引发了大语言模型研究的热潮。

Kaplan 等人在文献 [15] 中缩放法则（Scaling Laws），指出模型的性能依赖于模型的规模，包括：参数数量、数据集大小和计算量，模型的效果会随着三者的指数增加而线性提高。如图1.1所示，模型的损失（Loss）值随着模型规模的指数增大和线性降低。这意味着模型的能力是可以根据这三个变量估计的，提高模型参数量，扩大数据集规模都可以使得模型的性能可预测地提高。这为继续提升大模型的规模给出了定量分析依据。

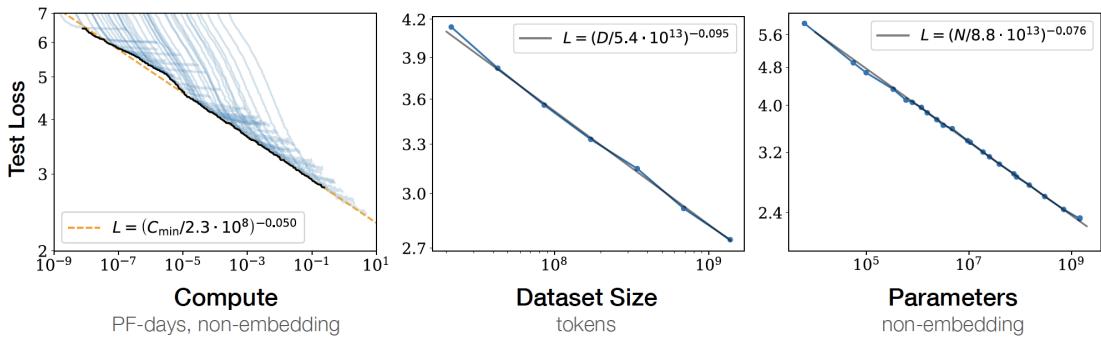
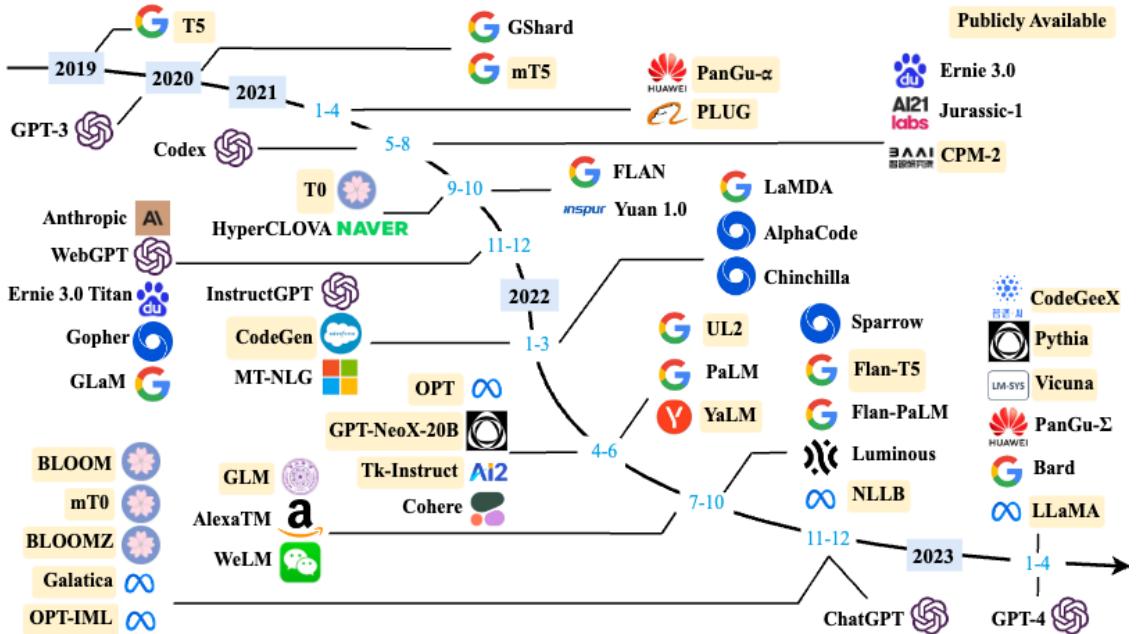


图 1.1 大语言模型的缩放法则 (Scaling Laws) ^[15]

1.2 大语言模型发展历程

大语言模型的发展历程虽然只有短短不到五年的时间，但是发展速度相当惊人，截止 2023 年 6 月，国内外有超过百种大模型相继发布。文献 [16] 按照时间线给出 2019 年至 2023 年 5 月比较有影响力并且模型参数量超过 100 亿的大语言模型，如图1.2所示。大语言模型的发展可以粗略的分为如下三个阶段：基础模型、能力探索、突破发展。

基础模型阶段主要集中于 2018 年至 2021 年，2017 年 Vaswani 等人提出了 Transformer^[10] 架构，在机器翻译任务上取得了突破性进展。2018 年 Google 和 Open AI 分别提出了 BERT^[1] 和 GPT-1^[2] 模型，开启了预训练语言模型时代。BERT-Base 版本参数量为 1.1 亿，BERT-Large 的参数量为 3.4 亿，GPT-1 的参数量 1.17 亿。这在当时，相比其它深度神经网络的参数量已经是数量级上提升。2019 年 Open AI 又发布了 GPT-2^[9]，其参数量达到了 15 亿。此后，Google 也发布了参数规模为 110

图 1.2 大语言模型时间线^[16]

亿的 T5^[17] 模型。2020 年 Open AI 进一步将语言模型参数量扩展到 1750 亿，发布了 GPT-3^[11]。此后，国内也相继推出了一系列的大语言模型，包括清华大学 ERNIE(THU)^[18]、百度 ERNIE(Baidu)^[19]、华为盘古- α ^[20] 等。这个阶段研究主要集中语言模型本身，包括仅编码器（Encoder Only）、编码器-解码器（Encoder-Decoder）、仅解码器（Decoder Only）等各种类型的模型结构都有相应的研究。模型大小与 BERT 相类似的算法，通常采用预训练微调范式，针对不同下游任务进行微调。但是模型参数量在 10 亿以上时，由于微调的计算量很高，这类模型的影响力在当时相较 BERT 类模型有不小的差距。

能力探索阶段集中于 2019 年至 2022 年，由于大语言模型很难针对特定任务进行微调，研究人员们开始探索在不针对单一任务进行微调的情况下如何能够发挥大语言模型的能力。2019 年 Radford 等人在文献 [9] 就使用 GPT-2 模型研究了大语言模型在零样本情况下的任务处理能力。在此基础上，Brown 等人在 GPT-3^[11] 模型上研究了通过语境学习（In-Context Learning）进行少样本学习的方法。将不同任务的少量有标注的实例拼接到待分析的样本之前输入语言模型，使用语言模型根据实例理解任务并给出正确结果。在包括 TriviaQA、WebQS、CoQA 等评测集合都展示出了非常强的能力，在有些任务中甚至超过了此前的有监督方法。上述方法不需要修改语言模型的参数，模型在处理不同任务时无需花费的大量计算资源进行模型微调。但是仅依赖基于语言模型本身，其性能在很多任务上仍然很难达到可以有效应用的程度，因此研究人员们提出了指令微调

(Instruction Fine-tuning) [21] 方案，将大量各类型任务，统一为生成式自然语言理解框架，并构造训练语料进行微调。大语言模型一次性学习数千种任务，并在未知任务上展现出了很好的泛化能力。2022 年 Ouyang 提出了使用有监督微调再结合强化学习方法，使用少量数据有监督就可以使得大语言模型服从人类指令的 InstructGPT 算法^[22]。Nakano 等人则探索了结合搜索引擎的问题回答算法 WebGPT^[23]。这些方法从直接利用大语言模型进行零样本和少样本学习的基础上，逐渐扩展到利用生成式框架针对大量任务进行有监督微调的方法，有效提升了模型的性能。

突破发展阶段以 2022 年 11 月 ChatGPT 的发布为起点。ChatGPT 通过一个简单的对话框，利用一个大语言模型就可以实现问题回答、文稿撰写、代码生成、数学结题等过去自然语言处理系统需要大量小模型订制开发才能分别实现的能力。它在开放领域问答、各类自然语言生成式任务以及对话上文理解上所展现出来的能力远超大多数人的想象。2023 年 3 月 GPT-4 发布，相较于 ChatGPT 又有了非常明显的进步，并具备了多模态理解能力。GPT-4 在多种基准考试测试上的得分高于 88% 的应试者，包括美国律师资格考试 (Uniform Bar Exam)、法学院入学考试 (Law School Admission Test)、学术能力评估 (Scholastic Assessment Test, SAT) 等。它展现了近乎“通用人工智能 (AGI)”的能力。各大公司和研究机构也相继发布了此类系统，包括 Google 推出的 Bard、百度的文心一言、科大讯飞的星火大模型、智谱 ChatGLM、复旦大学 MOSS 等。表1.1和表1.2分别给出了截止 2023 年 6 月典型开源和未开源大规模语言模型的基本情况。我们可以看到从 2022 年开始大模型呈现爆发式的增长，各大公司和研究机构都在发布各种不同类型的大模型。

1.3 大语言模型构建流程

根据 OpenAI 联合创始人 Andrej Karpathy 在微软 Build 2023 大会上所公开的信息，OpenAI 所采用大语言模型构建的流程如图1.3所示。主要包含四个阶段：预训练、有监督微调、奖励建模、强化学习。这四个阶段都需要是不同规模数据集合、不同类型的算法，产出不同类型的模型，所需要的资源也有非常大的差别。

预训练 (Pretraining) 阶段需要利用海量的训练数据，包括互联网网页、维基百科、书籍、GitHub、论文、问答网站等，构建包含数千亿甚至数万亿单词的具有多样性的内容。利用由数千块高性能 GPU 和高速网络组成超级计算机，花费数十天完成深度神经网络参数训练，构建基础语言模型 (Base Model)。基础大模型构建了长文本的建模能力，使得模型具有语言生成能力，根据输入的提示词 (Prompt)，模型可以生成文本补全句子。也有部分研究人员认为，语言模型建模过程中也隐含的构建了包括事实性知识 (Factual Knowledge) 和常识知识 (Commonsense) 在内的世界知识 (World Knowledge)。根据文献 [45] 介绍，GPT-3 完成一次训练的总计算量是 3640PFlops，按照 NVIDIA A100 80G 和平均利用率达到 50% 计算，需要花费近一个月时间使用 1000 块 GPU 完成。由于 GPT-3 训练采用了 NVIDIA V100 32G，其实际计算成本远高于上述计算。文献 [29] 介绍了参数量同样是 1750 亿的 OPT 模型，该模型训练使用了 992 块 NVIDIA A100 80G，整体训练时间将近 2 个月。BLOOM^[31] 模型的参数量也是 1750 亿，该模型训练一共花费 3.5 个月，使用包含 384

表 1.1 典型开源大语言模型汇总

模型名称	发布时间	模型参数量	基础模型	模型类型	预训练数据量
T5 ^[17]	2019 年 10 月	110 亿	-	语言模型	1 万亿 Token
mT5 ^[24]	2020 年 10 月	130 亿	-	语言模型	1 万亿 Token
PanGu- α ^[20]	2021 年 4 月	130 亿	-	语言模型	1.1 万亿 Token
CPM-2 ^[25]	2021 年 6 月	1980 亿	-	语言模型	2.6 万亿 Token
T0 ^[26]	2021 年 10 月	110 亿	T5	指令微调模型	-
CodeGen ^[27]	2022 年 3 月	160 亿	-	语言模型	5770 亿 Token
GPT-NeoX-20B ^[28]	2022 年 4 月	200 亿	-	语言模型	825GB 数据
OPT ^[29]	2022 年 5 月	1750 亿	-	语言模型	1800 亿 Token
GLM ^[30]	2022 年 10 月	1300 亿	-	语言模型	4000 亿 Token
Flan-T5 ^[21]	2022 年 10 月	110 亿	T5	指令微调模型	-
BLOOM ^[31]	2022 年 11 月	1760 亿	-	语言模型	3660 亿 Token
Galactica ^[32]	2022 年 11 月	1200 亿	-	语言模型	1060 亿 Token
BLOOMZ ^[33]	2022 年 11 月	1760 亿	BLOOM	指令微调模型	-
OPT-IML ^[34]	2022 年 12 月	1750 亿	OPT	指令微调模型	-
LLaMA ^[35]	2023 年 2 月	652 亿	-	语言模型	1.4 万亿 Token
MOSS	2023 年 2 月	160 亿	Codegen	指令微调模型	-
ChatGLM-6B ^[30]	2023 年 4 月	62 亿	GLM	指令微调模型	-
Alpaca ^[36]	2023 年 4 月	130 亿	LLaMA	指令微调模型	-
Vicuna ^[37]	2023 年 4 月	130 亿	LLaMA	指令微调模型	-
Koala ^[38]	2023 年 4 月	130 亿	LLaMA	指令微调模型	-
Baize ^[39]	2023 年 4 月	67 亿	LLaMA	指令微调模型	-
Robin-65B ^[40]	2023 年 4 月	652 亿	LLaMA	语言模型	-
BenTsao ^[41]	2023 年 4 月	67 亿	LLaMA	指令微调模型	-
Chinese-LLaMA-Alpaca ^[42]	2023 年 4 月	67 亿	LLaMA	指令微调模型	-
StableLM	2023 年 4 月	67 亿	LLaMA	语言模型	1.4 万亿 Token
GPT4All ^[43]	2023 年 5 月	67 亿	LLaMA	指令微调模型	-
MPT-7B	2023 年 5 月	67 亿	-	语言模型	1 万亿 Token
Falcon	2023 年 5 月	400 亿	-	语言模型	1 万亿 Token
OpenLLaMA	2023 年 5 月	130 亿	-	语言模型	1 万亿 Token
Gorilla ^[44]	2023 年 5 月	67 亿	MPT/Falcon	指令微调模型	-
RedPajama-INCITE	2023 年 5 月	67 亿	-	语言模型	1 万亿 Token
TigerBot-7b-base	2023 年 6 月	70 亿	-	语言模型	100GB 语料
Baichuan	2023 年 6 月	70 亿	-	语言模型	1.2 万亿 Token
Chat-Baichuan-7B	2023 年 6 月	70 亿	Baichuan	指令微调模型	-

表 1.2 典型闭源大语言模型汇总

模型名称	发布时间	模型参数量	基础模型	模型类型	预训练数据量
GPT-3	2020 年 5 月	1750 亿	-	-	3000 亿 Token
ERNIE 3.0	2021 年 7 月	100 亿	-	-	3750 亿 Token
FLAN	2021 年 9 月	1370 亿	LaMDA-PT	X	-
Yuan 1.0	2021 年 10 月	2450 亿	-	-	1800 亿 Token
Anthropic	2021 年 12 月	520 亿	-	-	4000 亿 Token
GLaM	2021 年 12 月	12000 亿	-	-	2800 亿 Token
LaMDA	2022 年 1 月	1370 亿	-	-	7680 亿 Token
InstructGPT	2022 年 3 月	1750 亿	GPT-3	X	-
Chinchilla	2022 年 3 月	700 亿	-	-	-
PaLM	2022 年 4 月	5400 亿	-	-	7800 亿 Token
Flan-PaLM	2022 年 10 月	5400 亿	PaLM	X	-
GPT-4	2023 年 3 月	-	-	X	-
PanGu- Σ	2023 年 3 月	10850 亿	PanGu- α	-	3290 亿 Token
Bard	2023 年 3 月	-	PaLM-2	指令微调模型	-
ChatGLM	2023 年 3 月	-	-	指令微调模型	-
天工 3.5	2023 年 4 月	-	-	指令微调模型	-
知海图 AI	2023 年 4 月	-	-	指令微调模型	-
360 智脑	2023 年 4 月	-	-	指令微调模型	-
文心一言	2023 年 4 月	-	-	指令微调模型	-
通义千问	2023 年 5 月	-	-	指令微调模型	-
MinMax	2023 年 5 月	-	-	指令微调模型	-
星火认知	2023 年 5 月	-	-	指令微调模型	-
浦语书生	2023 年 6 月	-	-	指令微调模型	-
悟道天鹰	2023 年 6 月	-	-	指令微调模型	-

块 NVIDIA A100 80G GPU 集群完成。可以看到大规模语言模型的训练需要花费大量的计算资源和时间。目前可以获得的此阶段模型包括：Llama 系列、Flcon 系列、Baichuan 系列。由于训练过程需要消耗大量的计算资源，并很容易受到超参数影响，因此如何能够提升分布式计算效率并使得模型训练稳定收敛是本阶段的重点研究内容。

有监督微调（Supervised Finetuning）阶段利用少量高质量数据集合，其中包含用户输入的提示词（Prompt）和对应的理想输出结果。用户输入包括问题、闲聊对话、任务指令等多种形式和任务。

例如：提示词（Prompt）：复旦大学有几个校区？

理想输出：复旦大学现有 4 个校区，分别是邯郸校区、新江湾校区、枫林校区和张江校区。其中邯郸校区是复旦大学的主校区，邯郸校区与新江湾校区都位于杨浦区，枫林校区



图 1.3 OpenAI 采用的大语言模型构建流程

位于徐汇区，张江校区位于浦东新区。

利用这些有监督数据，依然使用与预训练阶段完全相同的语言模型训练算法，在基础语言模型基础上再进行训练，从而得到有监督微调模型（SFT 模型）。经过训练的 SFT 模型具备了初步的指令理解能力和上下文理解能力，能够完成开放领域问题、阅读理解、翻译、生成代码等能力，也具备了一定的对未知任务的泛化能力。由于有监督微调阶段的所需的训练语料数量较少，SFT 模型的训练过程并不需要消耗非常大量的计算。根据模型的大小和训练数据量，通常需要数十块 GPU，花费数天时间完成训练。SFT 模型具备了初步的任务完成能力，可以开放给用户使用，目前很多类 ChatGPT 的模型都属于该类型，包括：Alpaca^[36]、Vicuna^[37]、MOSS、ChatGLM-6B 等。这类模型很多也到了很好的效果，甚至在一些评测中达到了 ChatGPT 的 90% 的效果。当前的一些研究表明有监督微调阶段数量选择对 SFT 模型效果有非常大的影响^[46]，因此如何构造少量并且高质量的训练数据是本阶段有监督微调阶段的研究重点。

奖励建模（Reward Modeling）阶段目标是构建一个文本质量对比模型，对于同一个提示词，SFT 模型给出的多个不同输出结果的质量进行排序。奖励模型（RM 模型）可以通过二分类模型，对输入的两个结果的好坏进行判断。RM 模型与基础语言模型和 SFT 模型不同，RM 模型本身并不能单独提供给用户使用。奖励模型的训练通常和 SFT 模型一样，使用 1 到 100 块 GPU，通过几天时间完成训练。由于 RM 模型的准确率对于强化学习阶段的效果有着至关重要的影响，因此对于该模型的训练通常需要大规模的训练数据。Andrej Karpathy 在报告中指出，该部分需要百万量级的对比数据标注，而且其中很多标注需要花费非常长的时间才能完成。图 1.4 给出了 InstructGPT 系统中奖励模型训练样本标注示例^[22]。我们可以看，示例中文本表达都较为流畅，标注其质量排序需

要制定非常详细的规范，标注人员也需要非常认真的对标规范内容进行标注，需要消耗大量的人力，同时如何保持众包标注人员之间的一致性，也是奖励建模阶段需要解决的难点问题之一。此外奖励模型的泛化能力边界也在本阶段需要重点研究的另一个问题。如果 RM 模型的目标是针对所有提示词系统所生成输出都能够高质量的进行判断，该问题所面临的难度在某种程度上与文本生成等价，因此如何限定 RM 模型应用的泛化边界也是本阶段难点问题。

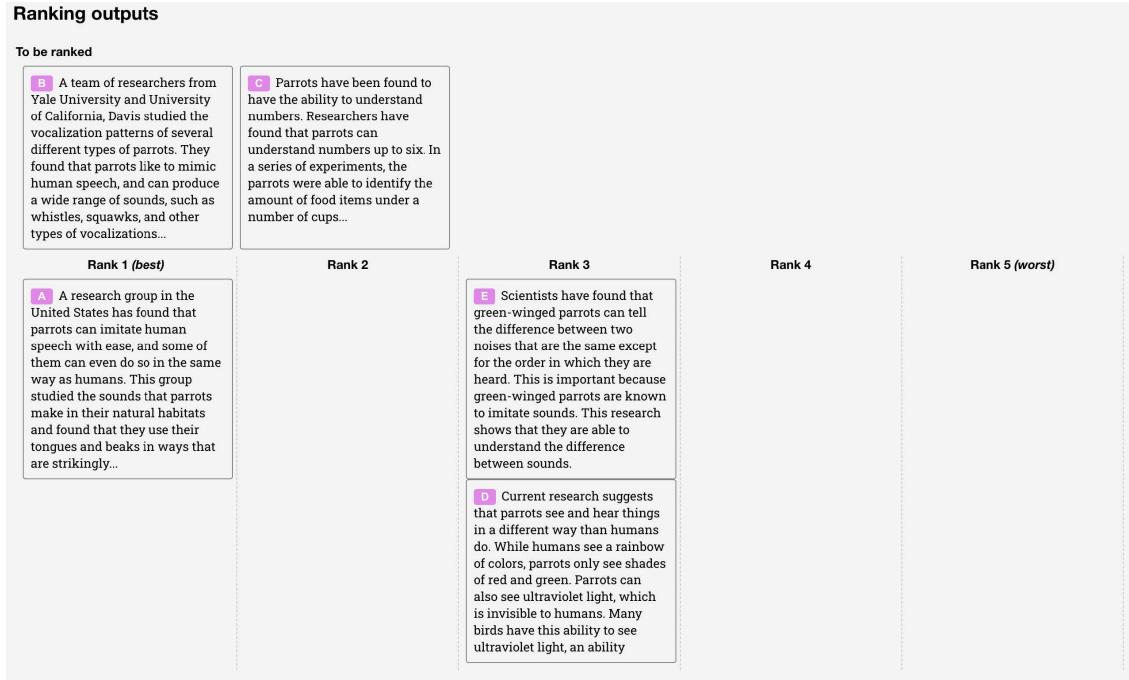


图 1.4 InstructGPT 系统中奖励模型训练样本标注示例^[22]

强化学习（Reinforcement Learning）阶段根据数十万用户给出的提示词，利用在前一阶段训练的 RM 模型，给出 SFT 模型对用户提示词补全结果的质量评估，并与语言模型建模目标综合得到更好的效果。该阶段所使用的提示词数量与有监督微调阶段类似，数量在十万量级，并且不需要人工提前给出该提示词所对应的理想回复。使用强化学习思想，SFT 模型基础上调整参数，使得最终生成的文本可以获得更高的奖励（Reward）。该阶段所需要的计算量相较预训练阶段也少很多，通常也仅需要 1 到 100 块 GPU，经过数天时间的即可完成训练。文献 [22] 给出了强化学习和有监督微调的对比，在模型参数量相同的情况下，强化学习可以得到相较于有监督微调好的的效果。关于为什么强化学习相比有监督微调可以得到更好结果的问题，目前并没有完整和得到普遍共识的解释。此外，Andrej Karpathy 也指出强化学习也并不是没有问题的，它会使得基础模

型的熵降低，从而减少了模型输出的多样性。在经过强化学习方法训练完成后的 RL 模型，就是最终提供给用户使用具有理解用户指令和上下文的类 ChatGPT 系统。由于强化学习方法稳定性不高，并且超参数众多，使得模型收敛难度大，再叠加 RM 模型的准确率问题，使得在大语言模型如何能够有效应用强化学习非常困难。

1.4 本书的内容安排

本书共分为 8 章，围绕大语言模型构建和评估的三个主要部分展开：第一部分主要介绍大规模语言模型预训练相关内容，包括语言模型技术、分布式模型训练和预训练数据；第二个部分主要介绍大语言模型理解并服从人类指令的有监督微调和强化学习；第三个部分主要介绍大语言模型扩展应用和评价。本书章节安排如图1.5所示。



图 1.5 本书章节安排

第 2 章主要介绍大语言模型所需要基础理论知识，包括语言模型的定义、Transformer 结构、大规模语言模型框架等内容，并以 Llama 所使用的模型结构为例介绍代码实例。

第 3 章和第 4 章主要围绕大语言模型预训练阶段的主要研究内容开展介绍，包括分布式模型训练中需要掌握的数据并行、流水线并行、模型并行以及 Zero 系列优化方法，除此之外还将介绍预训练所需要使用的数据分布和数据预处理方法，并以 Deepspeed 为例介绍如何进行大语言模型预训练。

第 5 章和第 6 章主要围绕如何在大语言模型指令理解阶段的主要研究内容进行介绍，即如何在基础模型基础上利用有监督微调和强化学习方法，使得模型理解指令并给出类人回答。主要包括 Lora、Delta Tuning 等模型高效微调方法、有监督微调数据构造方法、强化学习基础、近端策略

优化 (Proximal Policy Optimization, PPO)，并以 DeepSpeed-Chat 为例介绍如何在训练类 ChatGPT 系统。

第 7 章和第 8 章主要围绕在大语言模型的应用和评价开展介绍，主要包括如何将 LLM 与外部工具和知识源进行连接的 LangChain、能够利用 LLM 进行自动规划执行完成复杂任务的 AutoGPT 大语言模型应用，以及传统的语言模型评估方式，和针对大语言模型使用的各类评估方法。

希望读者朋友在本书学习结束时，都能够对大语言模型的基础理有初步了解，并能够开展大语言模型训练和研究。

2. 大语言模型基础

语言模型目标是建模自然语言的概率分布，在自然语言处理研究中具有重要的作用，是自然语言处理基础任务之一，大量的研究从 n 元语言模型 (n -gram Language Models)、神经语言模型 (Neural Language Models, NLM) 以及预训练语言模型 (Pre-trained Language Models, PLM) 等不同角度开展了系列工作。这些研究在不同阶段都对自然语言处理任务有着重要作用。随着基于 Transformer 各类语言模型的发展以及预训练微调范式在自然语言处理各类任务中取得突破性进展，从 2020 年 OpenAI 发布 GPT-3 开始，大语言模型研究也逐渐深入。虽然大语言模型的参数量巨大，通过有监督微调和强化学习能够完成非常多的任務，但是其基础理论也仍然离不开对语言的建模。

本章将首先介绍语言模型的基本概念以及 n 元语言模型，并在此基础上介绍 Transformer 架构以及大语言模型结构。

2.1 语言模型概述

语言模型 (Language Model, LM) 目标是构建词序列 $w_1 w_2 \dots w_m$ 的概率分布 $P(w_1 w_2 \dots w_m)$ ，即计算给定的词序列 $w_1 w_2 \dots w_m$ 作为一个句子出现的可能性大小。词汇表 \mathbb{V} 上的语言模型由函数 $P(w_1 w_2 \dots w_m)$ 表示，对于任意词串 $w_1 w_2 \dots w_m \in \mathbb{V}^+$ ，则有 $P(w_1 w_2 \dots w_m) \geq 0$ ，并且对于所有词串，函数 $P(w_1 w_2 \dots w_m)$ 满足归一化条件 $\sum_{w_1 w_2 \dots w_m \in \mathbb{V}^+} P(w_1 w_2 \dots w_m) = 1$ 。 $P(w_1 w_2 \dots w_m)$ 是定义在 \mathbb{V}^+ 上的概率分布。

由于联合概率 $P(w_1 w_2 \dots w_m)$ 的参数量十分巨大，直接计算 $P(w_1 w_2 \dots w_m)$ 非常困难。如果把 $w_1 w_2 \dots w_m$ 看作一个变量，那么它具有 $|\mathbb{V}|^m$ 种可能，其中 m 代表句子的长度， $|\mathbb{V}|$ 表示词表中单词的数量。按照《现代汉语词典（第七版）》包含 7 万词条，句子长度按照 20 个词计算，模型参数量达到 7.9792×10^{96} 的天文数字。中文的书面语中超过 100 个单词的句子也并不罕见，如果要将所有可能都纳入考虑，模型的复杂度还会进一步急剧增加，无法进行存储和计算。

为了减少 $P(w_1 w_2 \dots w_m)$ 模型参数量，可以利用句子序列通常情况下从左至右的生成过程进行分解，使用链式法则进行分解。给定由单词序列 $w_1 w_2 \dots w_n$ 组成的句子 S ，使用链式法分解则

得到：

$$P(S) = \prod_{i=1}^n P(w_i | w_1 w_2 \dots w_{i-1}) \quad (2.1)$$

其中，词 w_i 出现的概率受它前面的 $i-1$ 个词 $w_1 w_2 \dots w_{i-1}$ 影响，我们将这 $i-1$ 个词 $w_1 w_2 \dots w_{i-1}$ 称之为词 w_i 的历史。如果历史单词有 $i-1$ 个，那么可能的单词组合就有 $|\mathbb{V}|^{i-1}$ 种，其中 \mathbb{V} 表示单词词表， $|\mathbb{V}|$ 表示词表的大小。为了简化起见，使用 w_1^{i-1} 表示 $w_1 w_2 \dots w_{i-1}$ 。最简单的根据语料库对 $P(w_i | w_1 w_2 \dots w_{i-1})$ 进行估计的方法是基于词序列在语料中出现次数（也称为频次）的方法。

$$P(w_i | w_1 w_2 \dots w_{i-1}) = \frac{C(w_1 w_2 \dots w_{i-1} w_i)}{C(w_1 w_2 \dots w_{i-1})} \quad (2.2)$$

其中， $C(\cdot)$ 表示在语料库中词序列在语料库中出现次数。这种方法称为最大似然估计（Maximum Likelihood Estimation, MLE）。随着历史单词数量的增长，这种建模方式所需的数据量会指指数级增长，这一现象称为维数灾难（Curse of Dimensionality）。并且，随着历史单词数量增多，绝大多数的历史并不会在训练数据中出现，这也意味着 $P(w_i | w_1 w_2 \dots w_{i-1})$ 就很可能为 0，使得概率估计失去了意义。

为了解决上述问题，可以进一步假设任意单词 w_i 出现的概率只与过去 $n-1$ 个词相关，即：

$$\begin{aligned} P(w_i | w_1 w_2 \dots w_{i-1}) &= P(w_i | w_{i-(n-1)} w_{i-(n-2)} \dots w_{i-1}) \\ P(w_i | w_1^{i-1}) &= P(w_i | w_{i-n+1}^{i-1}) \end{aligned} \quad (2.3)$$

满足上述条件的模型被称为 n 元语法或 n 元文法(n -gram) 模型。其中 n -gram 表示由 n 个连续单词构成的单元，也被称为 n 元语法单元。 n 的取值越大，其历史信息越完整，但参数量也会随之增大。实际应用中， n 的取值通常小于等于 4。当 $n=1$ 时，每个词 w_i 的概率独立于历史，称为一元语法（Unigram）。当 $n=2$ 时，词 w_i 只依赖前一个词 w_{i-1} ，称为二元语法（Bigram），又被称为一阶马尔可夫链。当 $n=3$ 时，词 w_i 只依赖于前两个历史词 w_{i-1} 和 w_{i-2} ，称为三元语法（Trigram），又被称为二阶马尔可夫链。以二元语法为例，一个词的概率只依赖于前一个词，则句子 S 的出现概率可以表示为：

$$P(S) = \prod_{i=1}^n P(w_i | w_{i-1}) \quad (2.4)$$

对比公式2.1和公式2.3，可以看到语言模型计算通过 n 元语法假设进行了大幅度的简化。

尽管 n 元语言模型能缓解句子概率为 0 的问题，但语言是由人和时代创造的，具备无穷的可能性，再庞大的训练语料也无法覆盖所有的 n -gram，而训练语料中的零频率并不代表零概率。因此，需要使用平滑技术（Smoothing）来解决这一问题，对所有可能出现的字符串都分配一个非零的概率值，从而避免零概率问题。平滑是指为了产生更合理的概率，对最大似然估计进行调整的一类方法，也称为数据平滑（Data Smoothing）。平滑处理的基本思想是提高低概率，降低高概率，

使整体的概率分布趋于均匀。相关平滑算法细节可以参考《自然语言处理导论》第 6 章^[47]。

由于词的离散表示也忽略了单词之间的相似性，并且高阶 n 元语言模型还是会面临十分严重的数据稀疏问题，因此基于分布式表示和神经网络的语言模型逐渐成为了研究的热点。Bengio 等人在 2000 年提出了使用前馈神经网络对 $P(w_i|w_{i-n+1} \dots w_{i-1})$ 进行估计的语言模型^[3]。此后，循环神经网络^[4]、卷积神经网络^[5]、端到端记忆网络^[6]等神经网络方法都成功应用于语言模型建模。相较于 n 元语言模型，神经网络方法可以在一定程度上避免数据稀疏问题，有些模型还可以避免对历史长度的限制，从而更好的建模长距离依赖关系。相关算法细节可以参考《自然语言处理导论》第 6 章^[47]。

2.2 Transformer 模型

Transformer 模型^[48]是由谷歌在 2017 年提出并首先应用于机器翻译的模型。机器翻译的目标是从源语言转换到目标语言。Transformer 模型完全通过注意力机制完成对源语言序列和目标语言序列全局依赖的建模。当前几乎全部大语言模型都是基于 Transformer 架构，本节我们以应用于机器翻译的基于 Transformer 的编码器和解码器介绍该模型。

基于 Transformer 的编码器和解码器结构如图2.1所示，左侧和右侧分别对应着编码器(Encoder)和解码器(Coder)结构。它们均由若干个基本的 Transformer 块(Block)组成(对应着图中的灰色框)。这里 $N \times$ 表示进行了 N 次堆叠。每个 Transformer 块都接收一个向量序列 $\{\mathbf{x}_i\}_{i=1}^t$ 作为输入，并输出一个等长的向量序列作为输出 $\{\mathbf{y}_i\}_{i=1}^t$ 。这里的 \mathbf{x}_i 和 \mathbf{y}_i 分别对应着文本序列中的一个单词的表示。而 \mathbf{y}_i 是当前 Transformer 块对输入 \mathbf{x}_i 进一步整合其上下文语义后对应的输出。在从输入 $\{\mathbf{x}_i\}_{i=1}^t$ 到输出 $\{\mathbf{y}_i\}_{i=1}^t$ 的语义抽象过程中，主要涉及到如下几个模块：

- **自注意力层**：对应图中的 Multi-Head Attention 部分。使用自注意力机制整合上下文语义，它使得序列中任意两个单词之间的依赖关系可以直接被建模而不基于传统的循环结构，从而更好地解决文本的长程依赖。
- **前馈层**：对应图中的 Position-wise FNN 部分。通过全连接层对输入文本序列中的每个单词表示进行更复杂的变换。
- **残差连接**：对应图中的 Add 部分。它是一条分别作用在上述两个子层当中的直连通路，被用于连接它们的输入与输出。从而使得信息流动更加高效，有利于模型的优化。
- **层归一化**：对应图中的 Norm 部分。作用于上述两个子层的输出表示序列中，对表示序列进行层归一化操作，同样起到稳定优化的作用。

接下来我们依次介绍各个模块的具体功能和实现方法。

2.2.1 嵌入表示层

对于输入文本序列，首先通过输入嵌入层 (Input Embedding) 将每个单词转换为其相对应的向量表示。通常我们直接对每个单词创建一个向量表示。由于 Transfomer 模型不再使用基于循环的

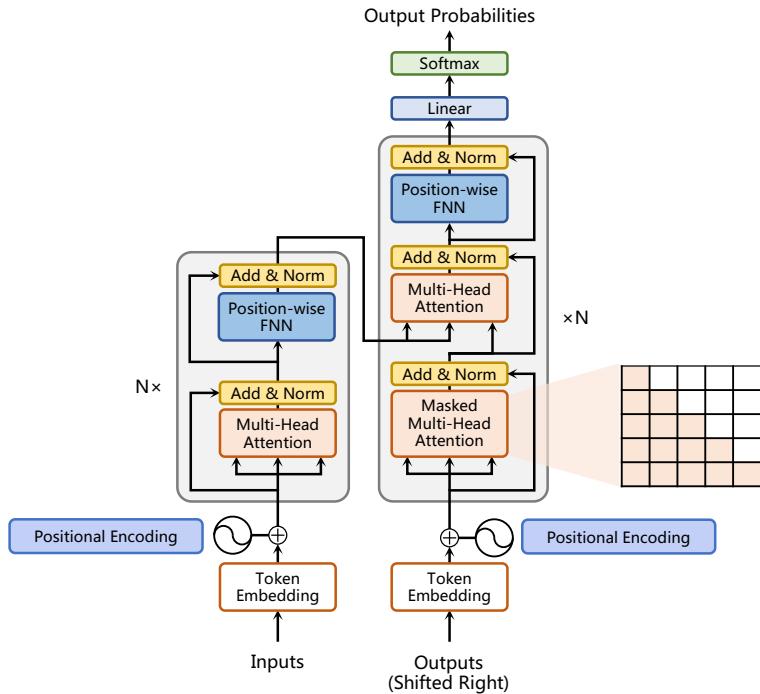


图 2.1 基于 Transformer 的编码器和解码器结构^[48]

方式建模文本输入，序列中不再有任何信息能够提示模型单词之间的相对位置关系。在送入编码器端建模其上下文语义之前，一个非常重要的操作是在词嵌入中加入位置编码（Positional Encoder）这一特征。具体来说，序列中每一个单词所在的位置都对应一个向量。这一向量会与单词表示对应相加并送入到后续模块中做进一步处理。在训练的过程当中，模型会自动地学习到如何利用这部分位置信息。

为了得到不同位置对应的编码，Transformer 模型使用不同频率的正余弦函数如下所示：

$$\text{PE}(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right) \quad (2.5)$$

$$\text{PE}(\text{pos}, 2i + 1) = \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right) \quad (2.6)$$

其中， pos 表示单词所在的位置， $2i$ 和 $2i+1$ 表示位置编码向量中的对应维度， d 则对应位置编码的总维度。通过上面这种方式计算位置编码有这样几个好处：首先，正余弦函数的范围是在 $[-1, +1]$ ，导出的位置编码与原词嵌入相加不会使得结果偏离过远而破坏原有单词的语义信息。其次，依据三角函数的基本性质，可以得知第 $\text{pos} + k$ 个位置的编码是第 pos 个位置的编码的线性组合，这就意味着位置编码中蕴含着单词之间的距离信息。

使用 Pytorch 实现的位置编码（Position Encoder）参考代码如下：

```

class PositionalEncoder(nn.Module):
    def __init__(self, d_model, max_seq_len = 80):
        super().__init__()
        self.d_model = d_model

        # 根据 pos 和 i 创建一个常量 PE 矩阵
        pe = torch.zeros(max_seq_len, d_model)
        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
                pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
                pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i + 1))/d_model)))

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # 使得单词嵌入表示相对大一些
        x = x * math.sqrt(self.d_model)
        # 增加位置常量到单词嵌入表示中
        seq_len = x.size(1)
        x = x + Variable(self.pe[:, :seq_len], requires_grad=False).cuda()
        return x

```

2.2.2 自注意力层

自注意力（Self-Attention）操作是基于 Transformer 的机器翻译模型的基本操作，在源语言的编码和目标语言的生成中频繁地被使用以建模源语言、目标语言任意两个单词之间的依赖关系。给定由单词语义嵌入及其位置编码叠加得到的输入表示 $\{x_i \in \mathbb{R}^d\}_{i=1}^t$ ，为了实现对上下文语义依赖的建模，进一步引入在自注意力机制中涉及到的三个元素：查询 q_i (Query)，键 k_i (Key)，值 v_i (Value)。在编码输入序列中每一个单词的表示的过程中，这三个元素用于计算上下文单词所对应的权重得分。直观地说，这些权重反映了在编码当前单词的表示时，对于上下文不同部分所需要的的关注程度。具体来说，如图2.2所示，通过三个线性变换 $W^Q \in \mathbb{R}^{d \times d_k}$, $W^K \in \mathbb{R}^{d \times d_k}$, $W^V \in \mathbb{R}^{d \times d_v}$ 将输入序列中的每一个单词表示 x_i 转换为其对应的 $q_i \in \mathbb{R}^{d_k}$, $k_i \in \mathbb{R}^{d_k}$, $v_i \in \mathbb{R}^{d_v}$ 向量。

为了得到编码单词 x_i 时所需要关注的上下文信息，通过位置 i 查询向量与其他位置的键向量做点积得到匹配分数 $q_i \cdot k_1, q_i \cdot k_2, \dots, q_i \cdot k_t$ 。为了防止过大的匹配分数在后续 Softmax 计算过程中导致的梯度爆炸以及收敛效率差的问题，这些得分会除放缩因子 \sqrt{d} 以稳定优化。放缩后的得分经过 Softmax 归一化为概率之后，与其他位置的值向量相乘来聚合我们希望关注的上下文信息，并最小化不相关信息的干扰。上述计算过程可以被形式化地表述如下：

$$Z = \text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \quad (2.7)$$

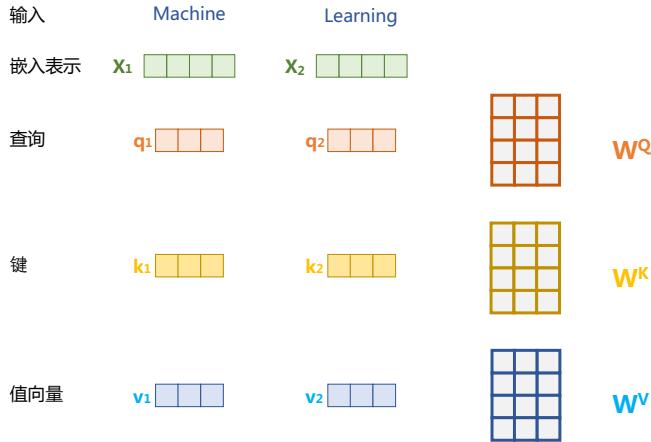


图 2.2 自注意力机制中的查询、键、值向量

其中 $\mathbf{Q} \in \mathbb{R}^{L \times d_k}$, $\mathbf{K} \in \mathbb{R}^{L \times d_k}$, $\mathbf{V} \in \mathbb{R}^{d_v \times d_v}$ 分别表示输入序列中的不同单词的 q, k, v 向量拼接组成的矩阵, L 表示序列长度, $Z \in \mathbb{R}^{L \times d_v}$ 表示自注意力操作的输出。为了进一步增强自注意力机制聚合上下文信息的能力, 提出了多头 (Multi-head) 自注意力的机制, 以从关注上下文的不同侧面。具体来说, 上下文中每一个单词的表示 x_i 经过多组线性 $\{\mathbf{W}_j^{bmQ} \mathbf{W}_j^K \mathbf{W}_j^V\}_{j=1}^N$ 映射到不同的表示子空间中。公式2.7会在不同的子空间中分别计算并得到不同的上下文相关的单词序列表示 $\{Z_j\}_{j=1}^N$ 。最终, 线性变换 $\mathbf{W}^O \in \mathbb{R}^{(Nd_v) \times d}$ 用于综合不同子空间中的上下文表示并形成自注意力层最终的输出 $\{x_i \in \mathbb{R}^d\}_{i=1}^t$ 。

使用 Pytorch 实现的自注意力层参考代码如下:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout = 0.1):
        super().__init__()

        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads

        self.q_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(d_model, d_model)

    def attention(q, k, v, d_k, mask=None, dropout=None):
        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)

        # 掩盖掉那些为了填补长度增加的单元, 使其通过 softmax 计算后为 0
```

```

if mask is not None:
    mask = mask.unsqueeze(1)
    scores = scores.masked_fill(mask == 0, -1e9)

scores = F.softmax(scores, dim=-1)

if dropout is not None:
    scores = dropout(scores)

output = torch.matmul(scores, v)
return output

def forward(self, q, k, v, mask=None):
    bs = q.size(0)

    # 进行线性操作划分为成 h 个头
    k = self.k_linear(k).view(bs, -1, self.h, self.d_k)
    q = self.q_linear(q).view(bs, -1, self.h, self.d_k)
    v = self.v_linear(v).view(bs, -1, self.h, self.d_k)

    # 矩阵转置
    k = k.transpose(1,2)
    q = q.transpose(1,2)
    v = v.transpose(1,2)

    # 计算 attention
    scores = attention(q, k, v, self.d_k, mask, self.dropout)

    # 连接多个头并输入到最后的线性层
    concat = scores.transpose(1,2).contiguous().view(bs, -1, self.d_model)

    output = self.out(concat)

    return output

```

2.2.3 前馈层

前馈层接受自注意力子层的输出作为输入，并通过一个带有 Relu 激活函数的两层全连接网络对输入进行更加复杂的非线性变换。实验证明，这一非线性变换会对模型最终的性能产生十分重要的影响。

$$\text{FFN}(x) = \text{Relu}(xW_1 + b_1)W_2 + b_2 \quad (2.8)$$

其中 W_1, b_1, W_2, b_2 表示前馈子层的参数。实验结果表明，增大前馈子层隐状态的维度有利于提升最终翻译结果的质量，因此，前馈子层隐状态的维度一般比自注意力子层要大。

使用 Pytorch 实现的前馈层参考代码如下：

```

class FeedForward(nn.Module):

    def __init__(self, d_model, d_ff=2048, dropout = 0.1):
        super().__init__()

        # d_ff 默认设置为 2048
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)

```

2.2.4 残差连接与层归一化

由 Transformer 模型组成网络结构通常都是非常庞大。编码器和解码器均由很多层基本的 Transformer 块组成，每一层当中都包含复杂的非线性映射，这就导致模型的训练比较困难。因此，研究者们在 Transformer 块中进一步引入了残差连接与层归一化技术以进一步提升训练的稳定性。具体来说，残差连接主要是指使用一条直连通道直接将对应子层的输入连接到输出上去，从而避免由于网络过深在优化过程中潜在的梯度消失问题：

$$\mathbf{x}^{l+1} = f(\mathbf{x}^l) + \mathbf{x}^l \quad (2.9)$$

其中 \mathbf{x}^l 表示第 l 层的输入， $f(\cdot)$ 表示一个映射函数。此外，为了进一步使得每一层的输入输出范围稳定在一个合理的范围内，层归一化技术被进一步引入每个 Transformer 块的当中：

$$LN(\mathbf{x}) = \alpha \cdot \frac{\mathbf{x} - \mu}{\sigma} + b \quad (2.10)$$

其中 μ 和 σ 分别表示均值和方差，用于将数据平移缩放到均值为 0，方差为 1 的标准分布， α 和 b 是可学习的参数。层归一化技术可以有效地缓解优化过程中潜在的不稳定、收敛速度慢等问题。

使用 Pytorch 实现的层归一化参考代码如下：

```

class NormLayer(nn.Module):

    def __init__(self, d_model, eps = 1e-6):
        super().__init__()

        self.size = d_model

        # 层归一化包含两个可以学习的参数

```

```

self.alpha = nn.Parameter(torch.ones(self.size))
self.bias = nn.Parameter(torch.zeros(self.size))

self.eps = eps

def forward(self, x):
    norm = self.alpha * (x - x.mean(dim=-1, keepdim=True)) \
        / (x.std(dim=-1, keepdim=True) + self.eps) + self.bias
    return norm

```

2.2.5 编码器和解码器结构

基于上述模块，根据图2.1所给出的网络架构，编码器端可以较为容易实现。相比于编码器端，解码器端要更复杂一些。具体来说，解码器的每个 Transformer 块的第一个自注意力子层额外增加了注意力掩码，对应图中的掩码多头注意力（Masked Multi-Head Attention）部分。这主要是因为在翻译的过程中，编码器端主要用于编码源语言序列的信息，而这个序列是完全已知的，因而编码器仅需要考虑如何融合上下文语义信息即可。而解码端则负责生成目标语言序列，这一生成过程是自回归的，即对于每一个单词的生成过程，仅有当前单词之前的目标语言序列是可以被观测的，因此这一额外增加的掩码是用来掩盖后续的文本信息，以防模型在训练阶段直接看到后续的文本序列进而无法得到有效地训练。

此外，解码器端还额外增加了一个多头注意力（Multi-Head Attention）模块，需要注意的是它同时接收来自编码器端的输出以及当前 Transformer 块第一个掩码注意力层的输出。它的作用是在翻译的过程当中，为了生成合理的目标语言序列需要观测待翻译的源语言序列是什么。基于上述的编码器和解码器结构，待翻译的源语言文本，首先经过编码器端的每个 Transformer 块对其上下文语义的层层抽象，最终输出每一个源语言单词上下文相关的表示。解码器端以自回归的方式生成目标语言文本，即在每个时间步 t ，根据编码器端输出的源语言文本表示，以及前 t_1 个时刻生成的目标语言文本，生成当前时刻的目标语言单词。

使用 Pytorch 实现的编码器参考代码如下：

```

class EncoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.attn = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)
        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)

```

```

def forward(self, x, mask):
    x2 = self.norm_1(x)
    x = x + self.dropout_1(self.attn(x2, x2, x2, mask))
    x2 = self.norm_2(x)
    x = x + self.dropout_2(self.ff(x2))
    return x

class Encoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(EncoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)

    def forward(self, src, mask):
        x = self.embed(src)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, mask)
        return self.norm(x)

```

使用 Pytorch 实现的解码器参考代码如下：

```

class DecoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.norm_3 = Norm(d_model)

        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)
        self.dropout_3 = nn.Dropout(dropout)

        self.attn_1 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.attn_2 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)

    def forward(self, x, e_outputs, src_mask, trg_mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn_1(x2, x2, x2, trg_mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.attn_2(x2, e_outputs, e_outputs, \
src_mask))
        x2 = self.norm_3(x)

```

22 自然语言处理导论 -- 张奇、桂韬、黄萱菁

```
x = x + self.dropout_3(self.ff(x2))
return x

class Decoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(DecoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)

    def forward(self, trg, e_outputs, src_mask, trg_mask):
        x = self.embed(trg)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, e_outputs, src_mask, trg_mask)
        return self.norm(x)
```

最终基于 Transformer 的编码器和解码器结构整体实现参考代码如下：

```
class Transformer(nn.Module):

    def __init__(self, src_vocab, trg_vocab, d_model, N, heads, dropout):
        super().__init__()
        self.encoder = Encoder(src_vocab, d_model, N, heads, dropout)
        self.decoder = Decoder(trg_vocab, d_model, N, heads, dropout)
        self.out = nn.Linear(d_model, trg_vocab)

    def forward(self, src, trg, src_mask, trg_mask):
        e_outputs = self.encoder(src, src_mask)
        d_output = self.decoder(trg, e_outputs, src_mask, trg_mask)
        output = self.out(d_output)
        return output
```

基于上述模型结构，我们使用如下代码进行模型训练和测试：

```
# 模型参数定义
d_model = 512
heads = 8
N = 6
src_vocab = len(EN_TEXT.vocab)
trg_vocab = len(FR_TEXT.vocab)
model = Transformer(src_vocab, trg_vocab, d_model, N, heads)
for p in model.parameters():
    if p.dim() > 1:
```

```

nn.init.xavier_uniform_(p)

optim = torch.optim.Adam(model.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)

# 模型训练
def train_model(epochs, print_every=100):

    model.train()

    start = time.time()
    temp = start

    total_loss = 0

    for epoch in range(epochs):

        for i, batch in enumerate(train_iter):
            src = batch.English.transpose(0,1)
            trg = batch.French.transpose(0,1)
            # the French sentence we input has all words except
            # the last, as it is using each word to predict the next

            trg_input = trg[:, :-1]
            # the words we are trying to predict

            targets = trg[:, 1:].contiguous().view(-1)

            # create function to make masks using mask code above

            src_mask, trg_mask = create_masks(src, trg_input)

            preds = model(src, trg_input, src_mask, trg_mask)

            optim.zero_grad()

            loss = F.cross_entropy(preds.view(-1, preds.size(-1)),
                                  results, ignore_index=target_pad)
            loss.backward()
            optim.step()

            total_loss += loss.data[0]
            if (i + 1) % print_every == 0:
                loss_avg = total_loss / print_every
                print("time = %dm, epoch %d, iter = %d, loss = %.3f,
                      %ds per %d iters" % ((time.time() - start) // 60,
                                           epoch + 1, i + 1, loss_avg, time.time() - temp,
                                           print_every))
                total_loss = 0
                temp = time.time()

# 模型测试
def translate(model, src, max_len = 80, custom_string=False):

```

```

model.eval()
if custom_sentence == True:
    src = tokenize_en(src)
    sentence=Variable(torch.LongTensor([[EN_TEXT.vocab.stoi[tok] for tok
        in sentence]]).cuda())
src_mask = (src != input_pad).unsqueeze(-2)
e_outputs = model.encoder(src, src_mask)

outputs = torch.zeros(max_len).type_as(src.data)
outputs[0] = torch.LongTensor([FR_TEXT.vocab.stoi['<sos>']])

for i in range(1, max_len):
    trg_mask = np.triu(np.ones((1, i, i),
        k=1).astype('uint8'))
    trg_mask= Variable(torch.from_numpy(trg_mask) == 0).cuda()

    out = model.out(model.decoder(outputs[:i].unsqueeze(0),
        e_outputs, src_mask, trg_mask))
    out = F.softmax(out, dim=-1)
    val, ix = out[:, -1].data.topk(1)

    outputs[i] = ix[0][0]
    if ix[0][0] == FR_TEXT.vocab.stoi['<eos>']:
        break
return ' '.join(
    [FR_TEXT.vocab.itos[ix] for ix in outputs[:i]])
)

```

2.3 预训练语言模型

受到计算机视觉领域采用 ImageNet^[7] 对模型进行一次预选训练，使得模型可以通过海量图像充分学习如何提取特征，然后再根据任务目标进行模型精调的范式影响，自然语言处理领域基于预训练语言模型的方法也逐渐成为主流。以 ELMo^[8] 为代表的动态词向量模型开启了语言模型预训练的大门，此后以 GPT^[9] 和 BERT^[11] 为代表的基于 Transformer 的大规模预训练语言模型的出现，使得自然语言处理全面进入了预训练微调范式新时代。利用丰富的训练语料、自监督的预训练任务以及 Transformer 等深度神经网络结构，使预训练语言模型具备了通用且强大的自然语言表示能力，能够有效地学习到词汇、语法和语义信息。将预训练模型应用于下游任务时，不需要了解太多的任务细节，不需要设计特定的神经网络结构，只需要“微调”预训练模型，即使用具体任务的标注数据在预训练语言模型上进行监督训练，就可以取得显著的性能提升。

本节中，我们将介绍仅包含编码器的 BERT 模型、仅包含解码器的 GPT 以及由编码器-解码器组成的 BART 模型。

2.3.1 掩码预训练语言模型 BERT

2018 年 Devlin 等人提出了掩码预训练语言模型 BERT^[1] (Bidirectional Encoder Representation from Transformers)。BERT 利用掩码机制构造了基于上下文预测中间词的预训练任务，相较于传统的语言模型建模方法，BERT 能进一步挖掘上下文所带来的丰富语义。BERT 所采用的神经结构如图2.3所示，其由多层 Transformer 编码器组成，这意味着在编码过程中，每个位置都能获得所有位置的信息，而不仅仅是历史位置的信息。BERT 同样由输入层，编码层和输出层三部分组成。编码层由多层 Transformer 编码器组成。在预训练时，模型的最后有两个输出层 MLM 和 NSP，分别对应了两个不同的预训练任务：掩码语言建模（Masked Language Modeling, MLM）和下一句预测（Next Sentence Prediction, NSP）。

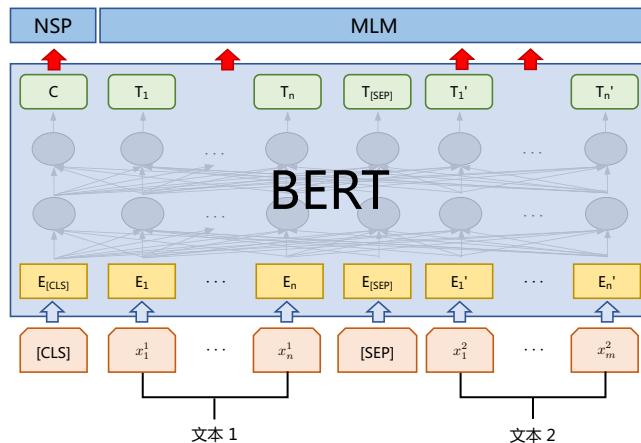


图 2.3 掩码预训练语言模型 BERT 神经网络结构^[1]

需要注意的是，掩码语言模型的训练对于输入形式没有要求，可以是一句话也可以一段文本，甚至可以是整个篇章，但是下一句预测则需要输入为两个句子，因此 BERT 在预训练阶段的输入形式统一为两段文字的拼接，这与其他预训练模型相比有较大区别。

1. 模型结构

BERT 输入层采用了 WordPiece 分词，根据词频决定是否将一个完整的词切分为多个子词（例如：单词 highest 可以被切分为 high 和 #est 两个子词）以缓解 OOV 问题。对输入文本进行分词后，BERT 的输入表示由三部分组成：词嵌入（Token Embedding）、段嵌入（Segment Embedding）和位置嵌入（Position Embedding）。每个词的输入表示 v 可以表示为：

$$v = v^t + v^s + v^p$$

其中， v^t 代表词嵌入； v^s 代表段嵌入； v^p 代表位置嵌入；三种嵌入维度均为 e 。

词嵌入用来将词转换为向量表示。完成分词后，切分完的子词通过词嵌入矩阵转化为词嵌入表示，假设子词对应的独热向量表示为 $e^t \in \mathbb{R}^{N \times |\mathcal{V}|}$ ，其对应的词嵌入 v_t 为：

$$v^t = e^t \mathbf{W}^t \quad (2.11)$$

其中， $\mathbf{W}^t \in \mathbb{R}^{|\mathcal{V}| \times e}$ 表示词嵌入矩阵； $|\mathcal{V}|$ 表示词表大小； e 表示词嵌入维度。

段嵌入用于区分不同词所属的段落（Segment），同一个段落中所有词的段嵌入相同。每个段落有其特有的段编码（Segment Encoding），段编码从 0 开始计数。通过段嵌入矩阵 \mathbf{W}^s 将独热段编码 e^s 转化为段嵌入 v^s ：

$$v^s = e^s \mathbf{W}^s \quad (2.12)$$

其中， $\mathbf{W}^s \in \mathbb{R}^{|\mathcal{S}| \times e}$ 表示段嵌入矩阵； $|\mathcal{S}|$ 表示段落数量； e 表示段嵌入维度。

位置嵌入用于表示不同词的绝对位置。将输入序列中每个词从左到右编号后，每个词都获得位置独热编码 e^p ，通过可训练的位置嵌入矩阵 \mathbf{W}^p 即可得到位置向量 v^p ：

$$v^p = e^p \mathbf{W}^p \quad (2.13)$$

其中， $\mathbf{W}^p \in \mathbb{R}^{N \times e}$ 表示位置嵌入矩阵； N 表示位置长度上限； e 表示位置嵌入维度。

BERT 的编码层采用多层 Transformer 结构，使用 L 表示所采用的层数， H 表示每层的隐藏单元数， A 是指自注意力头数量。在文献 [1] 给出了两种不同的参数设置，BERT_{BASE} 使用 $L = 12$ ， $H = 768$ ， $A = 12$ ，总参数量为 110M，BERT_{LARGE} 使用 $L = 24$ ， $H = 1024$ ， $A = 16$ ，总参数量为 340M。需要注意的是，与 GPT 中 Transformer 结构所采用的约束自注意力（Constrained Self-Attention）仅关注当前单元左侧上下文不同，BERT 采用的 Transformer 结构使用了双向多头自注意力机制，不仅关注当前单元左侧上下文情况，也会关注右侧上下文。

2. 预训练任务

不同于传统的自回归语言建模方法，BERT 使用去噪自编码（Auto-Encoding）的方法进行预训练。接下来将详细介绍 BERT 所采用预训练任务。

掩码语言建模：传统的语言模型只能顺序或逆序进行建模，这意味着除了当前词本身外，每个词的表示只能利用词左侧（顺序）或右侧（逆序）的词信息。但对于大部分下游任务来说，单向的信息是不充分的，因此同时利用两个方向的信息能带来更好的词表示，双向语言模型 ELMo 使用了顺序和逆序两个语言模型来解决这一问题。为了更好的利用上下文信息，让当前时刻的词表示同时编码“过去”和“未来”的文本，BERT 采用了一种类似于完形填空的任务，即掩码语言建模。在预训练时，随机将输入文本的部分单词掩盖（Mask），让模型预测被掩盖的单词，从而让模型具备根据上下文还原被掩盖的词的能力。

在 BERT 的预训练过程中，输入文本中 15% 的子词会被掩盖。具体来说，模型将被掩盖位置的词替换为特殊字符 “[MASK]”，代表模型需要还原该位置的词。但在执行下游任务时，[MASK] 字符并不会出现，这导致预训练任务和下游任务不一致。因此，在进行掩盖时，并不总是直接将词替换为 [MASK]，而是根据概率从三种操作中选择一种：(1) 80% 的概率替换为 [MASK]；(2) 10% 的概率替换为词表中任意词；(3) 10% 的概率不进行替换。

针对该掩码语言模型任务，使用 $x_1x_2\dots x_n$ 表示原始文本，在经过上述掩码替换后得到输入为 $x'_1x'_2\dots x'_n$ 。对掩码替换后的输入按照 BERT 框架输入层处理后，得到 BERT 的输入表示 v ：

$$X = [\text{CLS}]x'_1x'_2\dots x'_n[\text{SEP}] \quad (2.14)$$

$$v = \text{InputRepresentation}(X) \quad (2.15)$$

在编码层，对于输入表示 v 经过 L 层 Transformer，根据双向自注意力机制充分学习到文本中词语之间的联系，可以得到每个隐藏层输出以及最后的输出：

$$\mathbf{h}^{(l)} = \text{Transformer-Block}(\mathbf{h}^{(l-1)}) \quad l \in 1, 2, \dots, L \quad (2.16)$$

其中 $\mathbf{h}^{(l)} \in \mathbb{R}^{N \times d}$ 表示第 l 层 Transformer 的隐藏层输出， d 表示隐藏层维度， N 为输入的最大序列长度， $\mathbf{h}^{(0)} = v$ 表示输入。为了简化标记，可以还可以省略中间层，使用如下公式表示最终输出：

$$\mathbf{h} = \text{Transformer}(v) \quad (2.17)$$

其中 $\mathbf{h} = \mathbf{h}^{(L)}$ ，即模型最后一层的输出，得到最终上下文语义表示 $\mathbf{h} \in \mathbb{R}^{N \times d}$ 。

根据对于原始文本进行的掩码情况，得到掩盖位置的下标集合 $\mathbb{M} = \{m_1, m_2, \dots, m_k\}$ ， k 表示掩码数量。BERT 模型输出层，首先根据集合 \mathbb{M} 中元素下标，从隐藏层得到的上下文语义表示 \mathbf{h} 中抽取对应的表示 \mathbf{h}_{m_i} 。在此基础上，利用公式 2.11 中所给出的词向量矩阵 $\mathbf{W}^t \in \mathbb{R}^{V \times e}$ 将其映射到词空间表示，并通过如下公式计算对应词表上的概率分布 P_i ：

$$P_i = \text{Softmax}(\mathbf{h}_{m_i} \mathbf{W}^{t \top} + \mathbf{b}^0) \quad (2.18)$$

其中 $\mathbf{b}^0 \in \mathbb{R}^V$ 表示全连接层偏置。最后利用 P_i 与原始单词独热向量表示之间的交叉熵损失学习模型参数。

下一句预测：通过掩码语言建模，BERT 能够根据上下文还原掩码单词，从而具备构建对文本的语义表示能力。然而，对于阅读理解、语言推断等需要输入两段文本的任务来说，模型尚不具备判断两段文本关系的能力。因此，为了学习到两段文本间的关联，BERT 引入了第二个预训练任务：下一句预测 (NSP)。

故名思义，下一句预测的任务目标是预测两段文本是否构成上下句的关系。具体来说，对于

句子 A 和句子 B，若语料中这两个句子相邻，则构成正样本，若不相邻，则构成负样本。在预训练时，一个给定的句子对，有 50% 的概率将其中一句替换成来自其他段落的句子。这样可以将训练样本的正负例比例控制在 1:1。

该预训练任务与掩码语言模型任务非常类似，主要区别在于输出层。在输入层，对于给定的经过掩码处理的句子对 $x^{(1)} = x_1^{(1)}x_2^{(1)}\dots x_n^{(1)}$ 和 $x^{(2)} = x_1^{(2)}x_2^{(2)}\dots x_m^{(2)}$ ，经过如下处理得到 BERT 的输入表示 v :

$$X = [\text{CLS}]x_1^{(1)}x_2^{(1)}\dots x_n^{(1)}[\text{SEP}]x_1^{(2)}x_2^{(2)}\dots x_m^{(2)}[\text{SEP}] \quad (2.19)$$

$$v = \text{InputRepresentation}(X) \quad (2.20)$$

在 BERT 编码层，与掩码语言模型一样，通过 L 层 Transformer 编码，可以充分学习文本每个单词之间的关联，并最终得到文本语义表示:

$$h = \text{Transformer}(v) \quad (2.21)$$

下一句预测任务的输出层目标是判断输入文本 $x^{(2)}$ 是否是 $x^{(1)}$ 的下一个句子，可以转化为二分类问题。在该任务中，BERT 使用输入文本的开头添加 [CLS] 所对应的表示 $h_{[\text{CLS}]}$ 进行分类预测。使用全连接层预测输入文本的分类概率 $P \in \mathbb{R}^2$:

$$P = \text{Softmax}(h_{[\text{CLS}]}W^p + b^o) \quad (2.22)$$

其中， $W^p \in \mathbb{R}^{d \times 2}$ 为全连接层权重； b^o 表示全连接层偏置。根据分类概率 P 与真实分类标签之间的交叉熵损失，学习模型参数。

2.3.2 生成式预训练语言模型 GPT

OpenAI 公司在 2018 年提出的 GPT（Generative Pre-Training）^[9] 模型是典型的生成式预训练语言模型之一。GPT 模型结构如图 2.4 所示，由多层 Transformer 组成的单向语言模型，主要可以分为输入层，编码层和输出层三部分。本节将介绍 GPT 模型结构以及单向语言模型的预训练过程和判别式任务精调。

1. 无监督预训练

GPT 采用生成式预训练方法，单向意味着模型只能从左到右或从右到左对文本序列建模，所采用的 Transformer 结构^①和解码策略保证了输入文本每个位置只能依赖过去时刻的信息。

给定文本序列 $w = w_1w_2\dots w_n$ ，GPT 首先在输入层中将其映射为稠密的向量：

$$v_i = v_i^t + v_i^p \quad (2.23)$$

^① Transformer 解码器的具体结构请参考第 8 章??节。

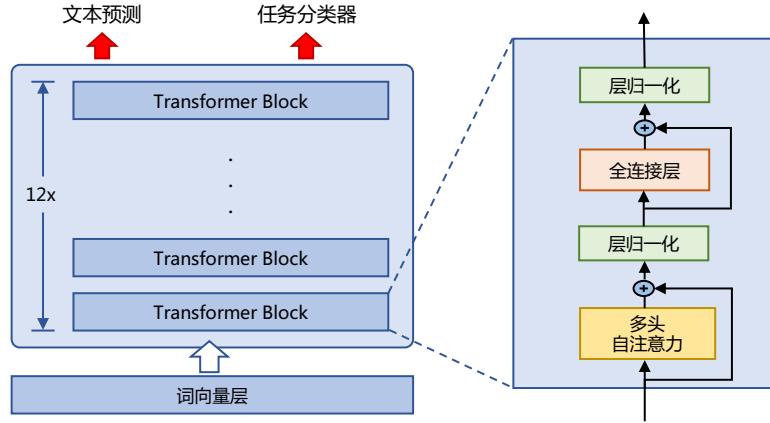


图 2.4 GPT 预训练语言模型结构

其中, v_i^t 是词 w_i 的词向量, v_i^p 是词 w_i 的位置向量, v_i 为第 i 个位置的单词经过模型输入层(第 0 层)后的输出。GPT 模型的输入层与前文中介绍的神经网络语言模型的不同之处在于其需要添加位置向量, 这是 Transformer 结构自身无法感知位置导致的, 因此需要来自输入层的额外位置信息。

经过输入层编码, 模型得到表示向量序列 $v = v_1 \dots v_n$, 随后将 v 送入模型编码层。编码层由 L 个 Transformer 模块组成, 在自注意力机制的作用下, 每一层的每个表示向量都会包含之前位置表示向量的信息, 使每个表示向量都具备丰富的上下文信息, 并且经过多层编码后, GPT 能得到每个单词层次化的组合式表示, 其计算过程表示如下:

$$\mathbf{h}^{(L)} = \text{Transformer-Block}^{(L)}(\mathbf{h}^{(0)}) \quad (2.24)$$

其中 $\mathbf{h}^{(L)} \in \mathbb{R}^{d \times n}$ 表示第 L 层的表示向量序列, n 为序列长度, d 为模型隐藏层维度, L 为模型总层数。

GPT 模型的输出层基于最后一层的表示 $\mathbf{h}^{(L)}$, 预测每个位置上的条件概率, 其计算过程可以表示为:

$$P(w_i | w_1, \dots, w_{i-1}) = \text{Softmax}(\mathbf{W}^e \mathbf{h}_i^{(L)} + \mathbf{b}^{out}) \quad (2.25)$$

其中, $\mathbf{W}^e \in \mathbb{R}^{|\mathbb{V}| \times d}$ 为词向量矩阵, $|\mathbb{V}|$ 为词表大小。

单向语言模型是按照阅读顺序输入文本序列 w , 用常规语言模型目标优化 w 的最大似然估计, 使之能根据输入历史序列对当前词能做出准确的预测:

$$\mathcal{L}^{\text{PT}}(w) = - \sum_{i=1}^n \log P(w_i | w_0 \dots w_{i-1}; \theta) \quad (2.26)$$

其中 θ 代表模型参数。也可以基于马尔可夫假设，只使用部分过去词进行训练。预训练时通常使用随机梯度下降法进行反向传播优化该负似然函数。

2. 有监督下游任务精调

通过无监督语言模型预训练，使得 GPT 模型具备了一定的通用语义表示能力。根据下游任务精调（Fine-tuning）的目的是在通用语义表示基础上，根据下游任务的特性进行适配。下游任务通常需要利用有标注数据集进行训练，数据集合使用 \mathbb{D} 进行表示，每个样例输入长度为 n 的文本序列 $x = x_1x_2\dots x_n$ 和对应的标签 y 构成。

首先将文本序列 x 输入 GPT 模型，获得最后一层的最后一个词所对应的隐藏层输出 $\mathbf{h}_n^{(L)}$ ，在此基础上通过全连接层变换结合 Softmax 函数，得到标签预测结果。

$$P(y|x_1\dots x_n) = \text{Softmax}(\mathbf{h}_n^{(L)} \mathbf{W}^y) \quad (2.27)$$

其中 $\mathbf{W}^y \in \mathbb{R}^{d \times k}$ 为全连接层参数， k 为标签个数。通过对整个标注数据集 \mathbb{D} 优化如下目标函数精调下游任务：

$$\mathcal{L}^{\text{FT}}(\mathbb{D}) = \sum_{(x,y)} \log P(y|x_1\dots x_n) \quad (2.28)$$

下游任务在精调过程中，针对任务目标进行优化，很容易使得模型遗忘预训练阶段所学习到的通用语义知识表示，从而损失模型的通用性和泛化能力，造成灾难性遗忘（Catastrophic Forgetting）问题。因此，通常会采用混合预训练任务损失和下游精调损失的方法来缓解上述问题。在实际应用中，通常采用如下公式进行下游任务精调：

$$\mathcal{L} = \mathcal{L}^{\text{FT}}(\mathbb{D}) + \lambda \mathcal{L}^{\text{PT}}(\mathbb{D}) \quad (2.29)$$

其中 λ 取值为 $[0,1]$ ，用于调节预训练任务损失占比。

2.3.3 序列到序列预训练语言模型 BART

在之前的章节中，我们介绍了适合自然语言生成的自回归式单向预训练语言模型 GPT，以及适合自然语言理解任务的掩码预训练语言模型 BERT。自回归式模型 GPT 缺乏上下文语境信息，而 BERT 虽然能利用上下文信息，但其预训练任务使其在自然语言生成任务上表现不佳。本节中，我们介绍一种符合自然语言生成任务需求的预训练模型 BART（Bidirectional and Auto-Regressive Transformers）^[49]。BART 兼具上下文语境信息的编码器和自回归特性的解码器，配合针对自然语言生成制定的预训练任务，使其格外契合生成任务的场景。

BART 模型也是使用基于 Transformer 的序列到序列结构，相较于标准的 Transformer，BART 选择了 GeLU 而不是 ReLU 作为激活函数，并且使用了正态分布 $N(0, 0.02)$ 进行初始化。Transformer 编码器具备双向编码上下文信息的能力，单向的 Transformer 解码器又满足生成任务的需求。BART

模型的基本结构如图2.5所示，结合了双向Transformer编码器以及单向的自回归解码器。

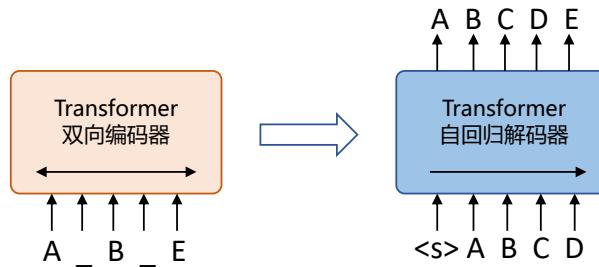


图 2.5 BART 的神经网络结构^[49]

1. 预训练任务

BART 的预训练过程采用的是对含有噪声的输入文本进行去噪重构方法，属于去噪自编码器（Denoising Autoencoder）。BART 使用双向编码对引入噪声的文本进行编码，单向的自回归解码器通过自回归方式顺序重构原始文本。编码器最后一层隐藏层表示参与解码器每一层的计算。BART 的预测过程与 BERT 独立预测掩码位置的词有很大不同。因此，BART 的预训练任务主要关注如何引入噪声。BART 模型使用了五种方式在输入文本上引入噪音：

- **单词掩码**（Token Masking）：随机从输入文本中选择一些单词，将其替换为掩码（[MASK]）标记，类似于 BERT。该噪音需要模型具备预测单个单词的能力。
- **单词删除**（Token Deletion）：随机从输入文本中删除一部分单词。该噪音除了需要模型预测单个单词的能力，还需要模型能定位缺失单词的位置。
- **文本填充**（Text Infilling）：随机将输入文本中多处连续的单词（称作文本片段）替换为一个掩码标记。文本片段的长度服从 $\lambda = 3$ 的泊松分布。当文本片段长度为 0 时，相当于插入一个掩码标记。该噪音需要模型能识别一个文本片段有多长，并具备预测缺失片段的能力。
- **句子排列变换**（Sentence Permutation）：对于一个完整的句子，根据句号将其分割为多个子句，随机打乱子句的顺序。该噪音需要模型能一定程度上理解输入文本的语义，具备推理前后句关系的能力。
- **文档旋转变换**（Document Rotation）：随机选择输入文本中的一个单词，以该单词作为文档的开头，并旋转文档。该噪音需要模型具备找到原始文本开头的能力。

图2.6给出了各种加噪方案的示例，输入加噪过程可以对这些方式进行组合使用。

可以看到，BART 的预训练时包含单词、句子和文档多种级别的任务，除了上述噪音之外，其他任意形式的文本噪音也是适用的。实验表明，使用文本填充任务能在下游任务上普遍取得性能提升，在文本填充噪音的基础上添加句子级别的去噪任务还能带来小幅提升。另外，尽管 BART

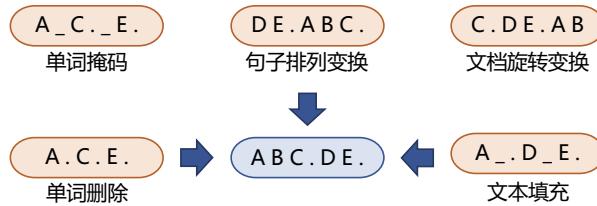


图 2.6 BART 各类型加噪方式示例

的预训练任务主要是为自然语言生成任务设计，但是它在一些自然语言理解任务上也展现出了不错的性能。

2. 模型精调

BART 预训练模型具备文本表示和生成能力，因此不仅适用于文本理解任务，也适用于文本生成任务，但是用于不同类型任务时，其精调方式有所不同。

对于序列分类任务，BART 模型的编码器和解码器的输入相同，但是将解码器最终时刻的隐藏层状态作为输入文本的语义向量表示，并利用线性分类器进行标签预测。利用标注数据和模型输出结果对模型参数进行调整。整个过程与 BERT 模型类似，在句子末尾添加特殊标记，利用该位置所对应的隐藏层状态表示文本。

对于生成式的任务，比如生成式文本摘要 (Abstractive Summarization)、生成式问答 (Abstractive Question Answering) 等任务，精调时模型输入为任务所给定的输入文本，解码器所产生的文本与任务的目标文本构成学习目标。

对于机器翻译任务，由于其输入文本和输出文本是两种不同的语言，使用的不同词汇集合，因此不能采用与生成式任务相同的方法。为了解决上述问题，研究人员们提出了将 BART 模型的输入层前增加小型 Transformer 编码器，将源语言文本映射到目标语言的输入表示空间。同时，为了解决两段模型训练过程不匹配的问题，采取分阶段的训练方法。详细过程可以参见文献 [49]。

2.3.4 基于 HuggingFace 预训练语言模型实践

HuggingFace 是一个开源自然语言处理软件库。其目标是通过提供一套全面的工具、库和模型，使的自然语言处理技术对开发人员和研究人员更加易于使用。HuggingFace 最著名的贡献之一是 Transformer 库，基于此研究人员可以快速部署训练好的模型以及实现新的网络结构。除此之外，HuggingFace 还提供了 Dataset 库，可以非常方便的下载自然语言处理研究中最常使用的基准数据集。本节中，我们以构建 BERT 模型为例，介绍基于 Huggingface 的 BERT 模型构建和使用方法。

1. 数据集合准备

常见的用于预训练语言模型的大规模数据都可以在 Dataset 库中直接下载并加载。例如，我们如果使用维基百科的英文语料集合，可以直接通过如下代码完成数据获取：

```

from datasets import concatenate_datasets, load_dataset

bookcorpus = load_dataset("bookcorpus", split="train")
wiki = load_dataset("wikipedia", "20230601.en", split="train")
# 仅保留 'text' 列
wiki = wiki.remove_columns([col for col in wiki.column_names if col != "text"])

dataset = concatenate_datasets([bookcorpus, wiki])

# 将数据集合切分为 90% 用于训练, 10% 用于测试
d = dataset.train_test_split(test_size=0.1)

```

接下来将训练和测试数据分别保存在本地文件中

```

def dataset_to_text(dataset, output_filename="data.txt"):
    """Utility function to save dataset text to disk,
    useful for using the texts to train the tokenizer
    (as the tokenizer accepts files)"""
    with open(output_filename, "w") as f:
        for t in dataset["text"]:
            print(t, file=f)

    # save the training set to train.txt
dataset_to_text(d["train"], "train.txt")
# save the testing set to test.txt
dataset_to_text(d["test"], "test.txt")

```

2. 训练子词切分器 (Tokenizer)

如前所述, BERT 采用了 WordPiece 分词, 根据训练语料中的词频决定是否将一个完整的词切分为多个子词。因此, 需要首先训练子词切分器 (Tokenizer)。我们可以使用 transformers 库中的 BertWordPieceTokenizer 类来完成任务, 代码如下所示:

```

special_tokens = [
    "[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]", "<S>", "<T>"
]
# if you want to train the tokenizer on both sets
# files = ["train.txt", "test.txt"]
# training the tokenizer on the training set
files = ["train.txt"]
# 30,522 vocab is BERT's default vocab size, feel free to tweak
vocab_size = 30_522
# maximum sequence length, lowering will result to faster training (when increasing batch size)
max_length = 512
# whether to truncate
truncate_longer_samples = False

# initialize the WordPiece tokenizer

```

```

tokenizer = BertWordPieceTokenizer()
# train the tokenizer
tokenizer.train(files=files, vocab_size=vocab_size, special_tokens=special_tokens)
# enable truncation up to the maximum 512 tokens
tokenizer.enable_truncation(max_length=max_length)

model_path = "pretrained-bert"

# make the directory if not already there
if not os.path.isdir(model_path):
    os.mkdir(model_path)
# save the tokenizer
tokenizer.save_model(model_path)
# dumping some of the tokenizer config to config file,
# including special tokens, whether to lower case and the maximum sequence length
with open(os.path.join(model_path, "config.json"), "w") as f:
    tokenizer_cfg = {
        "do_lower_case": True,
        "unk_token": "[UNK]",
        "sep_token": "[SEP]",
        "pad_token": "[PAD]",
        "cls_token": "[CLS]",
        "mask_token": "[MASK]",
        "model_max_length": max_length,
        "max_len": max_length,
    }
    json.dump(tokenizer_cfg, f)

# when the tokenizer is trained and configured, load it as BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained(model_path)

```

3. 预处理语料集合

在启动整个模型训练之前，我们还需要将预训练语料根据训练好的 Tokenizer 进行处理。如果文档长度超过 512 个子词（Token），那么就直接进行截断。数据处理代码如下所示：

```

def encode_with_truncation(examples):
    """Mapping function to tokenize the sentences passed with truncation"""
    return tokenizer(examples["text"], truncation=True, padding="max_length",
                    max_length=max_length, return_special_tokens_mask=True)

def encode_without_truncation(examples):
    """Mapping function to tokenize the sentences passed without truncation"""
    return tokenizer(examples["text"], return_special_tokens_mask=True)

# the encode function will depend on the truncate_longer_samples variable
encode = encode_with_truncation if truncate_longer_samples else encode_without_truncation
# tokenizing the train dataset
train_dataset = d["train"].map(encode, batched=True)
# tokenizing the testing dataset
test_dataset = d["test"].map(encode, batched=True)

```

```

if truncate_longer_samples:
    # remove other columns and set input_ids and attention_mask as PyTorch tensors
    train_dataset.set_format(type="torch", columns=["input_ids", "attention_mask"])
    test_dataset.set_format(type="torch", columns=["input_ids", "attention_mask"])
else:
    # remove other columns, and remain them as Python lists
    test_dataset.set_format(columns=["input_ids", "attention_mask", "special_tokens_mask"])
    train_dataset.set_format(columns=["input_ids", "attention_mask", "special_tokens_mask"])

```

truncate_longer_samples 布尔变量来控制用于对数据集进行子词处理的 encode() 回调函数。如果设置为 True，则会截断超过最大序列长度 (max_length) 的句子。否则，不会截断。如果设为 truncate_longer_samples 为 False，需要将没有截断的样本连接起来，并组合成固定长度的向量。

```

from itertools import chain
# Main data processing function that will concatenate all texts from our dataset and generate chunks of
# max_seq_length.
# grabbed from: https://github.com/huggingface/transformers/blob/main/examples/pytorch/language-modeling/run_m
def group_texts(examples):
    # Concatenate all texts.
    concatenated_examples = {k: list(chain(*examples[k])) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # We drop the small remainder, we could add padding if the model supported it instead of this drop, you can
    # customize this part to your needs.
    if total_length >= max_length:
        total_length = (total_length // max_length) * max_length
    # Split by chunks of max_len.
    result = [
        k: [t[i : i + max_length] for i in range(0, total_length, max_length)]
        for k, t in concatenated_examples.items()
    ]
    return result

# Note that with `batched=True`, this map processes 1,000 texts together, so group_texts throws away a
# remainder for each of those groups of 1,000 texts. You can adjust that batch_size here but a higher value
# might be slower to preprocess.
#
# To speed up this part, we use multiprocessing. See the documentation of the map method for more information:
# https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Dataset.map
if not truncate_longer_samples:
    train_dataset = train_dataset.map(group_texts, batched=True,
                                      desc=f"Grouping texts in chunks of {max_length}")
    test_dataset = test_dataset.map(group_texts, batched=True,
                                    desc=f"Grouping texts in chunks of {max_length}")
    # convert them from lists to torch tensors
    train_dataset.set_format("torch")
    test_dataset.set_format("torch")

```

4. 模型训练

在构建了处理好的预训练语料之后，就可以开始模型训练。代码如下所示：

```

# initialize the model with the config
model_config = BertConfig(vocab_size=vocab_size, max_position_embeddings=max_length)
model = BertForMaskedLM(config=model_config)

# initialize the data collator, randomly masking 20% (default is 15%) of the tokens for the Masked Language
# Modeling (MLM) task
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.2
)

training_args = TrainingArguments(
    output_dir=model_path,           # output directory to where save model checkpoint
    evaluation_strategy="steps",     # evaluate each `logging_steps` steps
    overwrite_output_dir=True,
    num_train_epochs=10,              # number of training epochs, feel free to tweak
    per_device_train_batch_size=10,    # the training batch size, put it as high as your GPU memory fits
    gradient_accumulation_steps=8,    # accumulating the gradients before updating the weights
    per_device_eval_batch_size=64,     # evaluation batch size
    logging_steps=1000,               # evaluate, log and save model checkpoints every 1000 step
    save_steps=1000,                 # save every 1000 steps
    # load_best_model_at_end=True,   # whether to load the best model (in terms of loss) at the end of training
    # save_total_limit=3,            # whether you don't have much space so you let only 3 model weights saved in th
)
)

trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
)
)

# train the model
trainer.train()

```

开始训练后，我们可以如下输出结果：

Step	Training Loss	Validation Loss
1000	6.904000	6.558231
2000	6.498800	6.401168
3000	6.362600	6.277831
4000	6.251000	6.172856
5000	6.155800	6.071129
6000	6.052800	5.942584
7000	5.834900	5.546123
8000	5.537200	5.248503
9000	5.272700	4.934949
10000	4.915900	4.549236

5. 模型使用

基于训练好的模型，可以针对不同应用需求进行使用。

```
# load the model checkpoint
model = BertForMaskedLM.from_pretrained(os.path.join(model_path, "checkpoint-10000"))
# load the tokenizer
tokenizer = BertTokenizerFast.from_pretrained(model_path)

fill_mask = pipeline("fill-mask", model=model, tokenizer=tokenizer)

# perform predictions
examples = [
    "Today's most trending hashtags on [MASK] is Donald Trump",
    "The [MASK] was cloudy yesterday, but today it's rainy.",
]
for example in examples:
    for prediction in fill_mask(example):
        print(f"[{prediction['sequence']}], confidence: {prediction['score']}]")
    print("=="*50)
```

可以得到如下输出：

```
today's most trending hashtags on twitter is donald trump, confidence: 0.1027069091796875
today's most trending hashtags on monday is donald trump, confidence: 0.09271949529647827
today's most trending hashtags on tuesday is donald trump, confidence: 0.08099588006734848
today's most trending hashtags on facebook is donald trump, confidence: 0.04266013577580452
today's most trending hashtags on wednesday is donald trump, confidence: 0.04120611026883125
=====
the weather was cloudy yesterday, but today it's rainy., confidence: 0.04445931687951088
the day was cloudy yesterday, but today it's rainy., confidence: 0.037249673157930374
the morning was cloudy yesterday, but today it's rainy., confidence: 0.023775646463036537
the weekend was cloudy yesterday, but today it's rainy., confidence: 0.022554103285074234
the storm was cloudy yesterday, but today it's rainy., confidence: 0.019406016916036606
=====
```

2.4 大语言模型网络结构

当前绝大多数大语言模型结构都采用了类似 GPT 架构，使用基于 Transformer 模型仅由解码器组成的网络结构，采用自回归的方式构建语言模型。但是在位置编码、层归一化位置以及激活函数等细节上各有不同。由于自 GPT-3 模型开始，OpenAI 就不再开源也没有公布模型细节，因此 ChatGPT 和 GPT-4 所采用的模型架构并不清楚。文献 [11] 介绍了 GPT-3 模型的训练过程，包括模型架构、训练数据组成、训练过程以及评估方法。由于 GPT-3 并没有开放源代码，根据论文直接重现整个训练过程不容易，因此文献 [29] 介绍了根据 GPT-3 的描述复现的过程，并构造开源

了系统 OPT (Open Pre-trained Transformer Language Models)。Meat AI 也仿照 GPT-3 架构开源了 LLaMA 模型^[35]，公开评测结果以及利用该模型进行有监督微调后的模型都有非常好的表现。

本节中，我们将以 LLaMA 模型为例，介绍大语言模型的模型架构在 Transformer 原始结构上的改进，并介绍 Transformer 模型结构中空间和时间占比最大的注意力机制优化方法。

2.4.1 LLaMA 的模型结构

文献 [35] 介绍了 LLaMA 所采用的 Transformer 结构和细节，与我们在本章 2.2 节所介绍的 Transformer 架构不通过的地方包括采用了前置层归一化 (Pre-normalization) 并使用 RMSNorm 归一化函数 (Normalizing Function)、激活函数更换为 SwiGLU，并使用了旋转位置嵌入 (RoP)，整体 Transformer 架构与 GPT-2 类似，如图2.7所示。

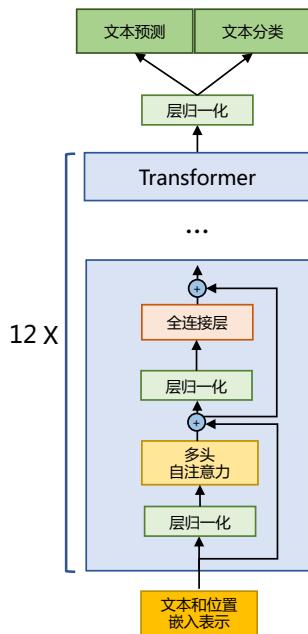


图 2.7 GPT-2 模型结构

接下来，我们将分别介绍 RMSNorm 归一化函数、SwiGLU 激活函数和旋转位置嵌入 (RoPE) 的具体内容和实现。

1. RMSNorm 归一化函数

为了使得模型训练过程更加稳定，GPT-2 相较于 GPT 就引入了前置层归一化方法，将第一个层归一化移动到多头自注意力层之前，第二个层归一化也移动到了全连接层之前，同时残差连接的

位置也调整到了多头自注意力层与全连接层之后。层归一化中也采用了 RMSNorm 归一化函数^[50]。针对输入向量 \mathbf{a} RMSNorm 函数计算公式如下：

$$RMS(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2} \quad (2.30)$$

$$\bar{a}_i = \frac{a_i}{RMS(\mathbf{a})} \quad (2.31)$$

此外，RMSNorm 还可以引入可学习的缩放因子 g_i 和偏移参数 b_i ，从而得到 $\bar{a}_i = \frac{a_i}{RMS(\mathbf{a})} g_i + b_i$ 。 RMSNorm 在 HuggingFace Transformer 库中代码实现如下所示：

```
class LlamaRMSNorm(nn.Module):
    def __init__(self, hidden_size, eps=1e-6):
        """
        LlamaRMSNorm is equivalent to T5LayerNorm
        """
        super().__init__()
        self.weight = nn.Parameter(torch.ones(hidden_size))
        self.variance_epsilon = eps // eps 防止取倒数之后分母为 0

    def forward(self, hidden_states):
        input_dtype = hidden_states.dtype
        variance = hidden_states.to(torch.float32).pow(2).mean(-1, keepdim=True)
        hidden_states = hidden_states * torch.rsqrt(variance + self.variance_epsilon)
        // weight 是末尾乘的可训练参数，即 g_i
        return (self.weight * hidden_states).to(input_dtype)
```

2. SwiGLU 激活函数

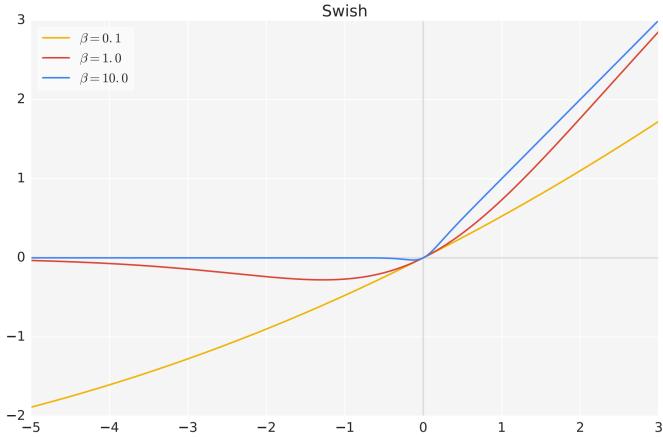
SwiGLU^[51] 激活函数是 Shazeer 在文献 [51] 中提出，并在 PaLM^[12] 等模中进行了广泛应用，并且取得了不错的效果，相较于 ReLU 函数在大部分评测中都有不少提升。在 LLaMA 中全连接层使用带有 SwiGLU 激活函数的 FFN（Position-wise Feed-Forward Network）的计算公式如下：

$$FFN_{\text{SwiGLU}}(\mathbf{x}, \mathbf{W}, \mathbf{V}, \mathbf{W}_2) = \text{SwiGLU}(\mathbf{x}, \mathbf{W}, \mathbf{V}) \mathbf{W}_2 \quad (2.32)$$

$$\text{SwiGLU}(\mathbf{x}, \mathbf{W}, \mathbf{V}) = \text{Swish}_{\beta}(\mathbf{x}\mathbf{W}) \otimes \mathbf{x}\mathbf{V} \quad (2.33)$$

$$\text{Swish}_{\beta}(\mathbf{x}) = \mathbf{x}\sigma(\beta\mathbf{x}) \quad (2.34)$$

其中， $\sigma(x)$ 是 Sigmoid 函数。图2.8给出了 Swish 激活函数在参数 β 不同取值下的形状。我们可以看到当 β 趋近于 0 时，Swish 函数趋近于线性函数 $y = x$ ，当 β 趋近于无穷大时，Swish 函数趋近于 ReLU 函数， β 取值为 1 时，Swish 函数是光滑且非单调。在 HuggingFace 的 Transformer 库中 Swish₁ 函数使用 silu 函数^[52] 代替。

图 2.8 Swish 激活函数在参数 β 不同取值下的形状

3. 旋转位置嵌入 (RoPE)

在位置编码上，使用旋转位置嵌入（Rotary Positional Embeddings，RoPE）^[53] 代替原有的绝对位置编码。RoPE 借助了复数的思想，出发点是通过绝对位置编码的方式实现相对位置编码。其目标是通过下述运算来给 \mathbf{q} , \mathbf{k} 添加绝对位置信息：

$$\tilde{\mathbf{q}}_m = f(\mathbf{q}, m), \tilde{\mathbf{k}}_n = f(\mathbf{k}, n) \quad (2.35)$$

经过上述操作后， $\tilde{\mathbf{q}}_m$ 和 $\tilde{\mathbf{k}}_n$ 就带有位置 m 和 n 的绝对位置信息。

详细的证明和求解过程可以参考文献 [53]，最终可以得到二维情况下用复数表示的 RoPE：

$$f(\mathbf{q}, m) = R_f(\mathbf{q}, m) e^{i\Theta_f(\mathbf{q}, m)} = ||\mathbf{q}|| e^{i(\Theta(\mathbf{q}) + m\theta)} = \mathbf{q} e^{im\theta} \quad (2.36)$$

根据复数乘法的几何意义，上述变换实际上是对应向量旋转，所以位置向量称为“旋转式位置编码”。还可以使用矩阵形式表示：

$$f(\mathbf{q}, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \end{pmatrix} \quad (2.37)$$

根据内积满足线性叠加的性质，任意偶数维的 RoPE，都可以表示为二维情形的拼接，即：

$$f(\mathbf{q}, m) = \underbrace{\begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_1 & -\sin m\theta_1 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_1 & \cos m\theta_1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \ddots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2-1} & -\sin m\theta_{d/2-1} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2-1} & \cos m\theta_{d/2-1} \end{pmatrix}}_{R_d} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \mathbf{q}_2 \\ \mathbf{q}_3 \\ \vdots \\ \mathbf{q}_{d-2} \\ \mathbf{q}_{d-1} \end{pmatrix} \quad (2.38)$$

由于上述矩阵 R_n 具有稀疏性，因此可以使用逐位相乘 \otimes 操作进一步加快计算速度。RoPE 在 HuggingFace Transformer 库中代码实现如下所示：

```
class LlamaRotaryEmbedding(torch.nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None):
        super().__init__()
        inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2).float().to(device) / dim))
        self.register_buffer("inv_freq", inv_freq)

        # Build here to make `torch.jit.trace` work.
        self.max_seq_len_cached = max_position_embeddings
        t = torch.arange(self.max_seq_len_cached, device=self.inv_freq.device, dtype=self.inv_freq.dtype)
        freqs = torch.einsum("i,j->ij", t, self.inv_freq)
        # Different from paper, but it uses a different permutation in order to obtain the same calculation
        emb = torch.cat((freqs, freqs), dim=-1)
        dtype = torch.get_default_dtype()
        self.register_buffer("cos_cached", emb.cos()[None, None, :, :].to(dtype), persistent=False)
        self.register_buffer("sin_cached", emb.sin()[None, None, :, :].to(dtype), persistent=False)

    def forward(self, x, seq_len=None):
        # x: [bs, num_attention_heads, seq_len, head_size]
        # This `if` block is unlikely to be run after we build sin/cos in `__init__`. Keep the logic here just
        if seq_len > self.max_seq_len_cached:
            self.max_seq_len_cached = seq_len
            t = torch.arange(self.max_seq_len_cached, device=x.device, dtype=self.inv_freq.dtype)
            freqs = torch.einsum("i,j->ij", t, self.inv_freq)
            # Different from paper, but it uses a different permutation in order to obtain the same calculation
            emb = torch.cat((freqs, freqs), dim=-1).to(x.device)
            self.register_buffer("cos_cached", emb.cos()[None, None, :, :].to(x.dtype), persistent=False)
            self.register_buffer("sin_cached", emb.sin()[None, None, :, :].to(x.dtype), persistent=False)
        return (
            self.cos_cached[:, :, :seq_len, ...].to(dtype=x.dtype),
            self.sin_cached[:, :, :seq_len, ...].to(dtype=x.dtype),
        )
```

```

def rotate_half(x):
    """Rotates half the hidden dims of the input."""
    x1 = x[:, ..., : x.shape[-1] // 2]
    x2 = x[:, ..., x.shape[-1] // 2 :]
    return torch.cat((-x2, x1), dim=-1)

def apply_rotary_pos_emb(q, k, cos, sin, position_ids):
    # The first two dimensions of cos and sin are always 1, so we can `squeeze` them.
    cos = cos.squeeze(1).squeeze(0) # [seq_len, dim]
    sin = sin.squeeze(1).squeeze(0) # [seq_len, dim]
    cos = cos[position_ids].unsqueeze(1) # [bs, 1, seq_len, dim]
    sin = sin[position_ids].unsqueeze(1) # [bs, 1, seq_len, dim]
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed

```

4. 模型整体框架

基于上述模型和网络结构可以实现解码器层，根据自回归方式利用训练语料进行模型的过程与我们在本章第 2.3.4 节介绍的过程基本一致。不同规模 LLaMA 模型所使用的具体超参数如表2.1所示。但是由于大语言模型的参数量非常大，并且需要大量的数据进行训练，因此仅利用单个 GPU 很难完成训练，需要依赖分布式模型训练框架（本书第 4 章将详细介绍相关内容）。

表 2.1 LLaMA 不同模型规模下的具体超参数细节^[35]

参数规模	层数	自注意力头数	嵌入表示维度	学习率	全局批次大小	训练 Token 数
6.7B	32	32	4096	3.0e-4	400 万	1.0 万亿
13.0B	40	40	5120	3.0e-4	400 万	1.0 万亿
32.5B	60	52	6656	1.5e-4	400 万	1.4 万亿
65.2B	80	64	8192	1.5e-4	400 万	1.4 万亿

HuggingFace Transformer 库中 LLaMA 解码器整体实现代码实现如下所示：

```

class LlamaDecoderLayer(nn.Module):
    def __init__(self, config: LlamaConfig):
        super().__init__()
        self.hidden_size = config.hidden_size
        self.self_attn = LlamaAttention(config=config)
        self.mlp = LlamaMLP(
            hidden_size=self.hidden_size,
            intermediate_size=config.intermediate_size,
            hidden_act=config.hidden_act,
        )
        self.input_layernorm = LlamaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)
        self.post_attention_layernorm = LlamaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)

```

```

def forward(
    self,
    hidden_states: torch.Tensor,
    attention_mask: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.LongTensor] = None,
    past_key_value: Optional[Tuple[torch.Tensor]] = None,
    output_attentions: Optional[bool] = False,
    use_cache: Optional[bool] = False,
) -> Tuple[torch.FloatTensor, Optional[Tuple[torch.FloatTensor, torch.FloatTensor]]]:
    residual = hidden_states
    hidden_states = self.input_layernorm(hidden_states)

    # Self Attention
    hidden_states, self_attn_weights, present_key_value = self.self_attn(
        hidden_states=hidden_states,
        attention_mask=attention_mask,
        position_ids=position_ids,
        past_key_value=past_key_value,
        output_attentions=output_attentions,
        use_cache=use_cache,
    )
    hidden_states = residual + hidden_states

    # Fully Connected
    residual = hidden_states
    hidden_states = self.post_attention_layernorm(hidden_states)
    hidden_states = self.mlp(hidden_states)
    hidden_states = residual + hidden_states

    outputs = (hidden_states,)

    if output_attentions:
        outputs += (self_attn_weights,)

    if use_cache:
        outputs += (present_key_value,)

return outputs

```

2.4.2 注意力机制优化

在 Transformer 结构中，自注意力机制的时间和存储复杂度与序列的序列长度呈平方的关系，因此占用了大量的计算设备内存和并消耗大量计算资源。因此，如何优化自注意力机制的时空复杂度、增强计算效率是大语言模型需要面临的重要问题。目前，一些研究从近似注意力出发，旨在减少注意力计算和内存需求，提出了包括稀疏近似、低秩近似等方法。此外，也有一些研究从计算加速设备本身的特性出发，研究如何更好利用硬件特性对 Transformer 中注意力层进行高效计算。本节中，我们将分别介绍上述两类方法。

1. 稀疏注意力机制

通过对一些训练好的 Transformer 模型中的注意力矩阵进行分析发现，其中很多是通常稀疏的，因此可以通过限制 Query-Key 对的数量来减少计算复杂度。这类方法就称为稀疏注意力（Sparse Attention）机制。可以将稀疏化方法进一步分成两类：基于位置信息和基于内容。

基于位置的稀疏注意力机制的基本类型如图2.9所示，主要包含如下五种类型：(1) 全局注意力（Global Attention）：为了增强模型建模长距离依赖关系，可以加入一些全局节点；(2) 带状注意力（Band Attention）：大部分数据都带有局部性，我们限制 Query 只与相邻的几个节点进行交互；(3) 膨胀注意力（Dilated Attention）；与 CNN 中的 Dilated Conv 类似，通过增加空隙以获取更大的感受野；(4) 随机注意力（Random Attention）：通过随机采样，提升非局部的交互；(5) 局部块注意力（Block Local Attention）：使用多个不重叠的块（Block）来限制信息交互。

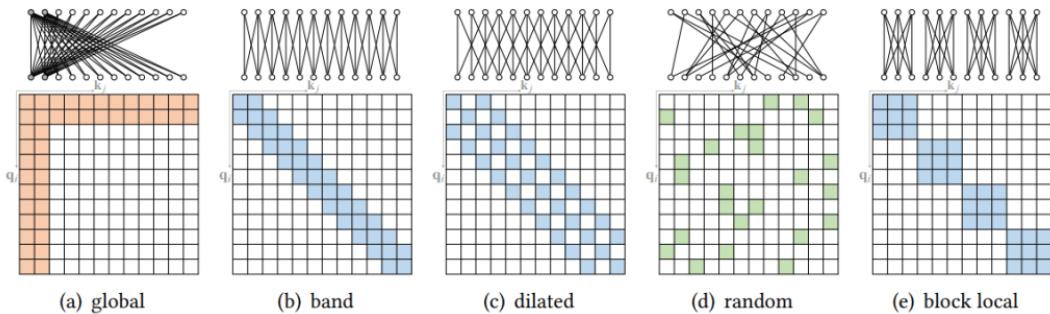


图 2.9 五种基于位置的稀疏注意力基本类型^[54]--需要重绘

现有的稀疏注意力机制，通常是基于上述五种基本基于位置的稀疏注意力机制的复合模式，图2.10给出了一些典型的稀疏注意力模型。Star-Transformer^[55] 使用带状注意力和全局注意力的组合。具体来说，Star-Transformer 只包括一个全局注意力节点和宽度为 3 的带状注意力，其中任意两个非相邻节点通过一个共享的全局注意力连接，而相邻节点则直接相连。Longformer^[56] 使用带状注意力和内部全局节点注意力（Internal Global-node Attention）的组合。此外，Longformer 还将上层中的一些带状注意力头部替换为具有扩张窗口的注意力，在增加感受野同时并不增加计算量。Extended Transformer Construction (ETC)^[57] 利用带状注意力和外部全局节点注意力（External Global-node Attention）的组合。ETC 稀疏注意力还包括一种掩码机制来处理结构化输入，并采用对比预测编码（Contrastive Predictive Coding, CPC）^[58] 进行预训练。BigBird^[59] 使用带状和全局注意力，还使用额外的随机注意力来近似全连接注意力，此外还揭示了稀疏编码器和稀疏解码器的使用可以模拟任何图灵机，这也在一定程度上解释了，为什么稀疏注意力模型可以取得较好的结果原因。

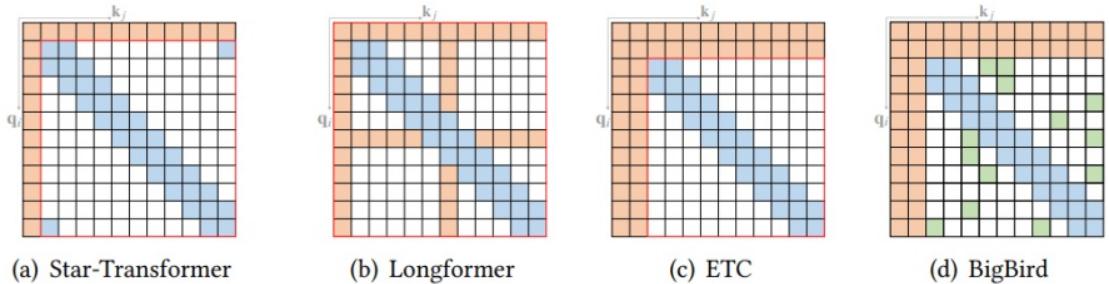


图 2.10 基于位置复合稀疏注意力类型^[54]--需要重绘

基于内容的稀疏注意力是根据输入数据来创建稀疏注意力，其中一种很简单的方法是选择和给定查询（Query）有很高相似度的键（Key）。Routing Transformer^[60]采用 K-means 聚类方法，针对 $\text{Query}\{\mathbf{q}_i\}_{i=1}^T$ 和 $\text{Key}\{\mathbf{k}_i\}_{i=1}^T$ 一起进行距离，类中心向量集合为 $\{\boldsymbol{\mu}_i\}_{i=1}^k$ ，其中 k 是类中心个数。每个 Query 只与其处在相同簇（Cluster）下的 Key 进行交互。中心向量采用滑动平均的方法进行更新：

$$\tilde{\boldsymbol{\mu}} \leftarrow \lambda \tilde{\boldsymbol{\mu}} + (1 - \lambda) \left(\sum_{i: \mu(\mathbf{q}_i) = \mu} \mathbf{q}_i + \sum_{j: \mu(\mathbf{k}_j) = \mu} \mathbf{k}_j \right) \quad (2.39)$$

$$c_\mu \leftarrow \lambda c_\mu + (1 - \lambda) |\mu| \quad (2.40)$$

$$\mu \leftarrow \frac{\tilde{\boldsymbol{\mu}}}{c_\mu} \quad (2.41)$$

其中 $|\mu|$ 表示在簇 μ 中向量的数量。

Reformer^[61]则采用局部敏感哈希（Local-Sensitive Hashing, LSH）方法来为每个 Query 选择 Key-Value 对。其主要思想使用 LSH 函数将 Query 和 Key 进行哈希计算，将它们划分到多个桶内。提升在同一个桶内的 Query 和 Key 参与交互的概率。假设 b 是桶的个数，给定一个大小为 $[D_k, b/2]$ 随机矩阵 \mathbf{R} ，LSH 函数定义为：

$$h(\mathbf{x}) = \arg \max([\mathbf{x}\mathbf{R}; -\mathbf{x}\mathbf{R}]) \quad (2.42)$$

如果 $h\mathbf{q}_i = h\mathbf{k}_j$ 时， \mathbf{q}_i 才可以与相应的 Key-Value 对进行交互。

2. FlashAttention

FlashAttention^[62]通过利用 GPU 硬件中的特殊设计，针对全局内存（Global Memory）提供高带宽显存（High Bandwidth Memory, HBM）和共享存储（Shared Memory, SRAM）的 I/O 速度的不同，尽可能的避免 HBM 中读取或写入注意力矩阵。图2.11给出了 NVIDIA GPU 的内存内存结

构，我们通常所说用 GPU 显存是全局存储（Global Memory），也就是 HBM 部分。该部分内存容量很大，NVIDIA H100 中该部分内存有 80GB 空间，虽然其访问速度虽然可以达到 3.35TB/s，但是相较于共享存储仍然慢一个数量级。共享存储速度非常快，但是由于其在 GPU 加速芯片内，因此容量很小，并且只有在同一个 GPU 线程块（Thread Block）内的线程才可以共享访问。NVIDIA H100 中每个 GPU 线程块可以使用的共享存储容量仅有 228KB，A100 仅有 164KB。

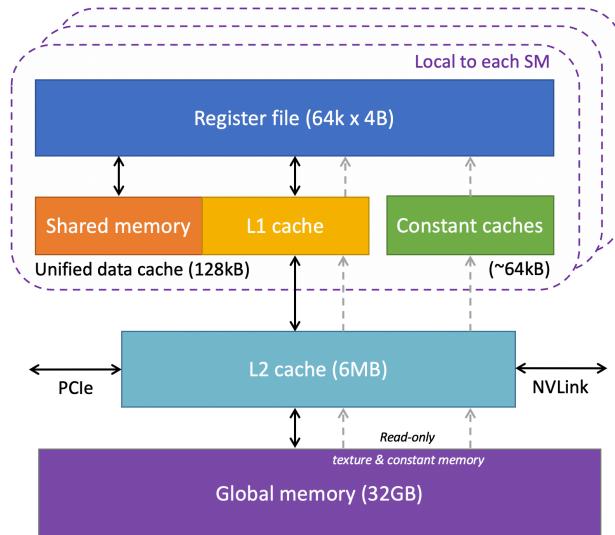


图 2.11 NVIDIA GPU 的整体内存结构图--需要重绘

我们在本章第 2.2 节中介绍自注意力机制的原理，在 GPU 中进行计算时通常还需要引入两个中间矩阵 S 和 P 并存储到全局内存中。具体计算过程如下：

$$S = Q \times K, \quad P = \text{Softmax}(S), \quad O = P \times V \quad (2.43)$$

按照上述计算过程，需要首先从全局内存中读取矩阵 Q 和 K ，并将计算好的矩阵 S 再写入全局内存，之后再从全局内存中获取矩阵 S ，计算 Softmax 得到矩阵 P ，再写入全局内存，之后读取矩阵 P 和矩阵 V ，计算得到矩阵矩阵 O 。这样的过程会极大占用显存的带宽。

FlashAttention 旨在避免从全局内存中读取和写入注意力矩阵。达成该目标需要能够做到在不访问整个输入的情况下计算 Softmax 函数，并且后向传播中不能存储中间注意力矩阵。标准 Attention 算法中，Softmax 计算按行进行，即在与 V 做矩阵乘法之前，需要将 Q 、 K 的各个分块完成整一行的计算。在得到 Softmax 的结果后，再与矩阵 V 分块做矩阵乘。而在 FlashAttention 中，将输入分割成块，并在输入块上进行多次传递，从而以增量方式执行 Softmax 计算。

标准 Attention 算法的实现需要将计算过程中的矩阵 S 、 P 写入全局内存中，而这些中间矩阵的大小与输入的序列长度有关且为二次型，因此 FlashAttention 就提出了不使用中间注意力矩阵，通过存储归一化因子来减少全局内存的消耗。FlashAttention 算法并没有将 S 、 P 整体写入全局内存，而是通过分块写入，存储前向传递的 Softmax 归一化因子，在后向传播中快速重新计算片上注意力，这比从全局内容中读取中间注意力矩阵的标准方法更快。由于大幅度减少了全局内的访问量，即使重新计算导致 FLOPS 增加，但其运行速度更快并且使用更少的内存。具体算法可以参考图2.12，其中内循环和外循环所对应的计算可以参考图2.13。

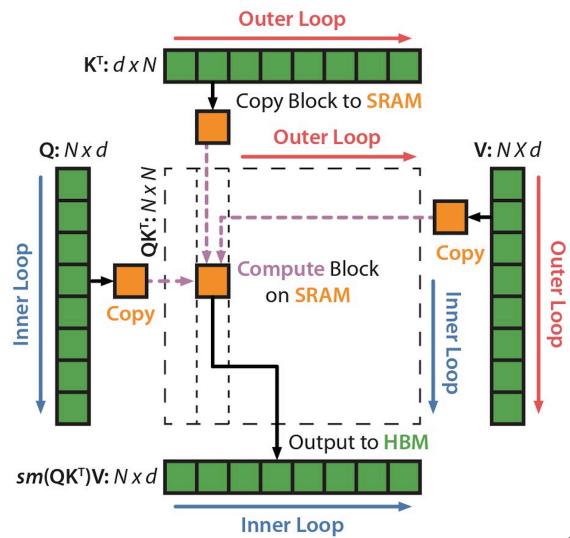
Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M .

- 1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
- 2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
- 3: Divide \mathbf{Q} into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide \mathbf{K}, \mathbf{V} in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
- 4: Divide \mathbf{O} into T_r blocks $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_1, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks m_1, \dots, m_{T_r} of size B_r each.
- 5: **for** $1 \leq j \leq T_c$ **do**
- 6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
- 7: **for** $1 \leq i \leq T_r$ **do**
- 8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
- 9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
- 10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}, \tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
- 11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}, \ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
- 12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
- 13: Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.
- 14: **end for**
- 15: **end for**
- 16: Return \mathbf{O} .

图 2.12 FlashAttention 算法--用 Algorithm 重写^[62]

PyTorch 2.0 中已经可以支持 FlashAttention，使用“`torch.backends.cuda.enable_flash_sdp()`”启用或者关闭 FlashAttention 的使用。

图 2.13 FlashAttention 计算流程图^[62]

3. 预训练数据

3.1 数据集介绍

在预训练语料集方面，根据文献 [11] 中的报道，GPT-3 训练语料通过主要包含经过过滤的 Common Crawl 数据集^[17]、WebText2、Books1、Books2 以及英文 Wikipedia 等数据集合。其中 CommonCrawl 的原始数据有 45TB，进行过滤后仅保留了 570GB 的数据。通过子词方式对上述语料进行切分，大约一共包含 5000 亿字词。为了保证模型使用更多高质量数据进行训练，在 GPT-3 训练时，根据语料来源的不同，设置不同的采样权重。在完成 3000 亿字词训练时，英文 Wikipedia 的语料平均训练轮数为 3.4 次，而 Common Crawl 和 Books 2 仅有 0.44 次和 0.43 次。由于 Common Crawl 数据集合的过滤过程繁琐复杂，OPT 则采用了混合 RoBERTa^[63]、Pile^[64] 和 PushShift.io Reddit^[65] 数据的方法。由于这些数据集合中包含的绝大部分都是英文数据，因此 OPT 也从 Common Crawl 数据集中抽取了部分非英文数据加入训练语料。

3.2 数据选择

4. 分布式训练

随着语言模型参数量和所需数据量的急速增长，单个机器上有限的资源已无法满足大语言模型训练的需求。需要设计分布式训练（Distributed Training）系统来解决海量的计算和内存资源要求问题。在分布式训练系统环境下需要将一个模型训练任务拆分成多个子任务，并将子任务分发给多个计算设备，从而解决资源瓶颈。但是如何才能利用成千上百的计算加速芯片的集群，训练模型参数量千亿甚至是万亿的大规模语言模型？这其中涉及到集群架构、并行策略、模型架构、内存优化、计算优化等一系列的技术。

本章将介绍分布式机器学习系统的相关概念、分布式训练集群架构、分布式训练并行策略，并以 DeepSpeed 为例介绍如何在集群上训练大语言模型。

4.1 分布式训练概述

分布式训练（Distributed Training）是指将机器学习或深度学习模型训练任务分解成多个子任务，并在多个计算设备上并行地进行训练。图4.1给出了单个计算设备和多个计算设备的示例。这个计算设备可以是中央处理器（Central Processing Unit, CPU）、图形处理器（Graphics Processing Unit, GPU）、张量处理器（Tensor Processing Unit, TPU）也可以是神经网络处理器（Neural network Processing Unit, NPU）。由于同一个服务器内部的多个 GPU、TPU 和 NPU 之间内存也并不共享，因此无论这些计算设备是否处于一个服务器还是多个服务器中，其系统架构都属于分布式系统范畴。一个模型训练任务往往会有大量的训练样本作为输入，可以利用一个计算设备完成，也可以将整个模型的训练任务拆分成子任务，分发给不同的计算设备，实现并行计算。每个计算设备的输出进行合并，最终得到与单个计算设备等价的计算结果。由于每个计算设备只需要负责子任务，并且多个计算设备可以并行执行，因此其可以更快速地完成整体计算，并最终实现对整个计算过程的加速。

促使人们设计分布式训练系统的一个最重要的原因就是单处理器的算力已经不足以完成模型训练所需计算资源。图4.2给出了机器学习模型对于算力的需求以及同期单个处理器能够提供的算力。如图所示，机器学习模型快速发展，从 2013 年 AlexNet 开始，到 2022 年拥有 5400 亿参数的 PaLM 模型，机器学习模型以每 18 个月增长 56 倍的速度发展。模型参数规模增大的同时，对训练

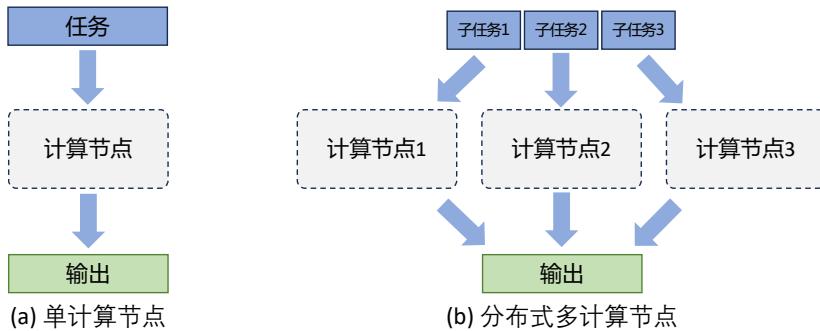
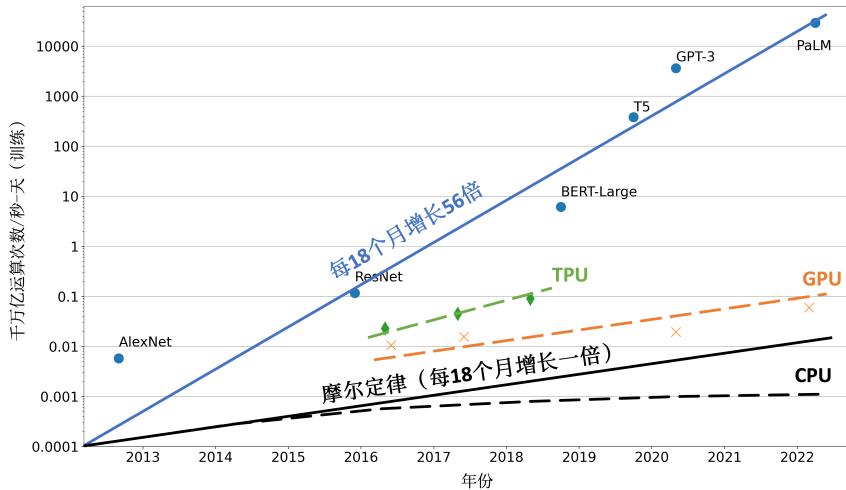


图 4.1 单计算设备计算和多计算设备示例

数据量的要求也指数级增长，这更加剧了对算力的需求。然而，近几年 CPU 的算力增加已经远低于摩尔定律（Moore's Law），虽然计算加速卡（如 GPU 和 TPU）为机器学习模型提供了大量的算力，但是其增长速度仍然没有有效突破每 18 个月增长 2 倍的摩尔定律。为了能够满足机器学习模型的发展，只有通过分布式训练系统才可以匹配模型不断增长的算力需求。

图 4.2 机器学习模型参数量增长和计算硬件的算力增长对比^[66]

分布式训练的总体目标就是提升总的训练速度，减少模型训练的总体时间，总训练速度可以用如下公式简略估计：

$$\text{总训练速度} \propto \text{单设备计算速度} \times \text{计算设备总量} \times \text{多卡加速比} \quad (4.1)$$

其中，单设备计算速度主要由单块计算加速芯片的运算速度和数据 I/O 能力来决定，对单设备训练效率进行优化，主要的技术手段有混合精度训练、算子融合、梯度累加等；分布式训练系统中计算设备数量越多，其理论峰值计算速度就会越高，但是受到通讯效率的影响，计算设备数量增大则会造成加速比急速降低；多设备加速比则是由计算和通讯效率决定，需要结合算法和网络拓扑结构进行优化，分布式训练并行策略主要目标就是提升分布式训练系统的加速比。

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。文献 [11] 针对 GPT-3 的训练过程仅介绍了训练过程全部使用 NVIDIA V100 GPU，文献 [29] 介绍了 OPT 使用了 992 块 NVIDIA A100 80G GPU，采用全分片数据并行 (Fully Shared Data Parallel) [67] 以及 Megatron-LM 张量并行 (Tensor Parallelism) [68]，整体训练时间将近 2 个月。BLOOM^[31] 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通信。文献 [35] 并没有给出 LLaMA 模型训练中所使用的集群的具体配置和网络拓扑结构，但是给出了不同参数规模的总 GPU 小时数。LLaMA 模型训练采用 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。由于 LLaMA 所使用的训练数据量远超 OPT 和 BLOOM 模型，因此，虽然模型参数量远小于上述两个模型，但是其所需计算量仍然非常惊人。

通过使用分布式训练系统，大语言模型训练周期可以从单计算设备花费几十年，缩短到数千个计算设备花费几十天就可以完成。然而，分布式训练系统需要克服计算墙、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算量之间存在巨大差异。NVIDIA H100 SXM 的单卡 FP16 算力只有 2000 TFLOPs，而 GPT-3 则需要 314 ZFLOPs 的总算力，两者相差了 8 个数量级。
- **显存墙：**单卡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，如果采用 FP16 格式进行存储，需要 700GB 的计算设备内容空间，而 NVIDIA H100 GPU 只有 80 GB 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。而目前单个 InfiniBand 链路仅能够提供 400Gb/s 带宽。

计算墙和显存墙源于单计算设备的计算和存储能力有限，与模型对庞大计算和存储需求之间的矛盾。这个问题可以通过采用分布式训练方法来解决，但分布式训练又会面临通信墙的挑战。在多机多卡的训练中，这些问题逐渐显现。随着大型模型参数的增大，对应的集群规模也随之增加，这些

问题变得更加突出。同时，在大型集群进行长时间训练时，设备故障可能会影响或中断训练过程。

4.2 分布式训练并行策略

分布式训练系统目标就是将单节点模型训练转换成等价的分布式并行模型训练。对于大语言模型来说，训练过程就是根据数据和损失函数，利用优化算法对神经网络模型参数的进行更新的过程。单节点模型训练系统结构如图4.3所示，主要由数据和模型两个部分组成。训练过程会由多个数据小批次（mini-batch）完成。图中数据表示一个数据小批次。训练系统会利用数据小批次根据损失函数和优化算法生成梯度，从而对模型参数进行修正。针对大语言模型多层神经网络的执行过程，可以由一个计算图（Computational Graph）表示。这个图有多个相互连接的算子（Operator），每个算子实现一个神经网络层（Neural Network Layer），而参数则代表了这个层在训练中所更新的权重（Weights）。

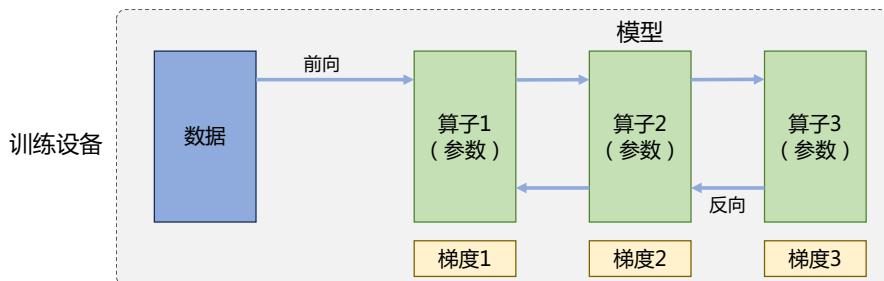


图 4.3 单设备模型训练系统

计算图的执行过程可以分为前向计算和反向计算两个阶段。前向计算的过程是将数据读入第一个算子，计算出相应的输出结构，然后依次重复这个前向计算过程，直到最后一个算子结束。反向计算过程，是根据优化函数和损失，每个算子依次计算出梯度，并利用梯度更新本地的参数。在反向计算结束后，该数据小批次的计算完成，系统就会读取下一个数据小批次，继续下一轮的模型参数更新。

根据单设备模型训练系统的流程，可以看到如果进行并行加速，可以从数据和模型两个维度进行考虑。首先可以对数据进行切分（Partition），并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，这种方式通常被称为数据并行（Data Parallelism, DP）。还可以对模型进行划分，将模型中的算子分发到多个设备分别完成，这种方式通常被称为模型并行（Model Parallelism, MP）。当训练超大规模语言模型时，往往需要同时对数据和模型进行切分，从而实现更高程度的并行，这种方式通常被称为混合并行（Hybrid Parallelism, HP）。

4.2.1 数据并行

在数据并行系统中，每个计算设备都有整个神经网络模型的完整副本（Model Replica），每次进行迭代的时候，每个计算设备只分配了一个批次数据样本的子集，每个计算设备利用该批次样本子集的数据进行网络模型的前向计算。假设一个批次的训练样本数为 N ，使用 M 个加速卡并行计算，每个计算设备会分配到 N/M 个样本。前向计算完成后，每个计算设备都会根据本地样本计算损失误差得到梯度 \mathbf{G}_i (i 为加速卡编号)，并将本地梯度 \mathbf{G}_i 进行传播。所有加速卡需要聚合其他加速卡给出的梯度值，并使用平均梯度 $(\sum_{i=1}^N \mathbf{G}_i)/N$ 对模型进行更新，完成该批次训练。图4.4给出了由两个训练加速设备组成的数据并行训练系统样例。

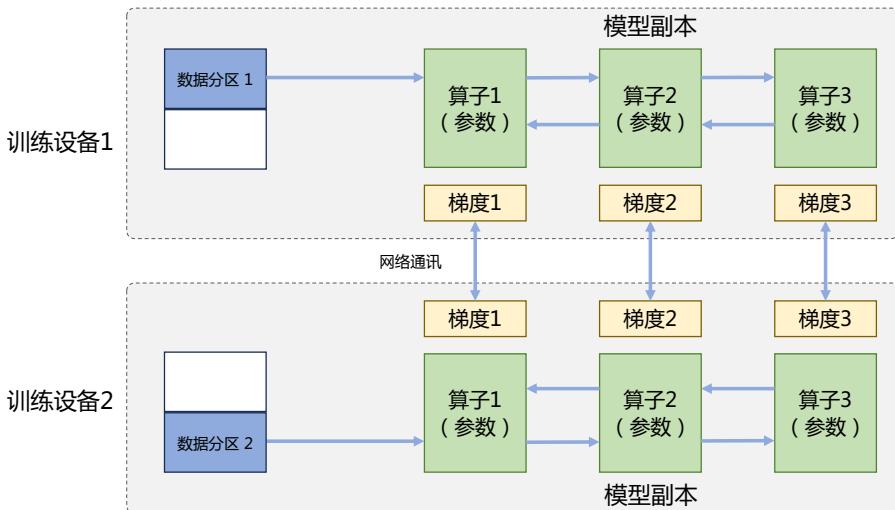


图 4.4 两节点数据并行训练系统样例

数据并行训练系统可以通过增加并行训练设备，有效提升整体训练吞吐量（Global Batch Size Per Second）。它和单卡训练相比，最主要的区别就在于反向计算中的梯度需要在所有训练设备中进行同步，以保证每个训练设备上最终得到的是所有进程上梯度的平均值。常见的神经网络框架中都有数据并行方式的具体实现，包括：TensorFlow DistributedStrategy、PyTorch Distributed、Horovod DistributedOptimizer 等。由于基于 Transformer 架构的大语言模型中每个算子都是依赖单个数据而非批数据，因此数据并行并不会影响其计算逻辑，一般情况下各训练设备中前向计算是独立的，不涉及同步问题。数据并行需要数据并行训练加速比最高，但要求每个设备上都备份一份模型，显存占用比较高。

使用 PyTorch DistributedDataParallel 实现单个服务器多加速卡训练代码如下，首先构造 DistributedSampler 类，将数据集的样本随机打乱并分配到不同加速卡：

```

class DistributedSampler(Sampler):
    def __init__(self, dataset, num_replicas=None, rank=None, shuffle=True, seed=0):
        if num_replicas is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            num_replicas = dist.get_world_size()
        if rank is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            rank = dist.get_rank()
        self.dataset = dataset # 数据集
        self.num_replicas = num_replicas # 进程个数 默认等于 world_size(GPU 个数)
        self.rank = rank # 当前属于哪个进程/哪块 GPU
        self.epoch = 0
        self.num_samples = int(math.ceil(len(self.dataset) * 1.0 / self.num_replicas)) # 每个进程的样本个数
        self.total_size = self.num_samples * self.num_replicas # 数据集总样本的个数
        self.shuffle = shuffle # 是否要打乱数据集
        self.seed = seed

    def __iter__(self):
        # 1. Shuffle 处理: 打乱数据集顺序
        if self.shuffle:
            # deterministically shuffle based on epoch and seed
            g = torch.Generator()
            # 这里 self.seed 是一个定值, 通过 set_epoch 改变 self.epoch 可以改变我们的初始化种子
            # 这就可以让我们在每一个 epoch 中数据集的打乱顺序不同, 使我们每一个 epoch 中每一块 GPU 拿到的数据都不一样
            g.manual_seed(self.seed + self.epoch)
            indices = torch.randperm(len(self.dataset), generator=g).tolist()
        else:
            indices = list(range(len(self.dataset)))

        # 数据补充
        indices += indices[:-(self.total_size - len(indices))]
        assert len(indices) == self.total_size

        # 分配数据
        indices = indices[self.rank:self.total_size:self.num_replicas]
        assert len(indices) == self.num_samples

        return iter(indices)

    def __len__(self):
        return self.num_samples

    def set_epoch(self, epoch):
        r"""
        Sets the epoch for this sampler. When :attr:`shuffle=True`, this ensures all replicas
        use a different random ordering for each epoch. Otherwise, the next iteration of this
        sampler will yield the same ordering.

        Arguments:
            epoch (int): Epoch number.
        """
        self.epoch = epoch

```

利用 DistributedSampler 构造完整的训练程序样例 main.py 如下：

```

import argparse
import os
import shutil
import time
import warnings
import numpy as np

warnings.filterwarnings('ignore')

import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.distributed as dist
import torch.optim
import torch.utils.data
import torch.utils.data.distributed
from torch.utils.data.distributed import DistributedSampler

from models import DeepLab
from dataset import Cityscapes


parser = argparse.ArgumentParser(description='DeepLab')

parser.add_argument('-j', '--workers', default=4, type=int, metavar='N',
                    help='number of data loading workers (default: 4)')
parser.add_argument('--epochs', default=100, type=int, metavar='N',
                    help='number of total epochs to run')
parser.add_argument('--start-epoch', default=0, type=int, metavar='N',
                    help='manual epoch number (useful on restarts)')
parser.add_argument('-b', '--batch-size', default=3, type=int,
                    metavar='N')
parser.add_argument('--local_rank', default=0, type=int, help='node rank for distributed training')

args = parser.parse_args()
torch.distributed.init_process_group(backend="nccl") # 初始化

print("Use GPU: {} for training".format(args.local_rank))

# create model
model = DeepLab()

torch.cuda.set_device(args.local_rank) # 当前卡
model = model.cuda() # 模型放在卡上
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_rank], output_device=args.local_r

```

```
criterion = nn.CrossEntropyLoss().cuda()

optimizer = torch.optim.SGD(model.parameters(), args.lr, momentum=args.momentum, weight_decay=args.weight_decay)

train_dataset = Cityscapes()
train_sampler = DistributedSampler(train_dataset) # 分配数据

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=args.batch_size, shuffle=False, num_workers=4)
```

通过以下命令行启动上述程序：

```
CUDA_VISIBLE_DEVICES=0,1 python -m torch.distributed.launch --nproc_per_node=2 main.py
```

4.2.2 模型并行

模型并行 (Model Parallelism) 往往用于解决单节点内存不足的问题。还是以包含 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 $175G \times 4$ bytes) 内存，如果使用 16 位浮点表示，每个模型副本需要也需要占用 350GB 内存。当前 NVIDIA 最高性能的 H100 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的切分为以下两种形式：(1) 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)，也称之为流水线并行 (Pipeline Parallelism, PP)；(2) 将计算图中的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)，也称之为张量并行 (Tensor Parallelism, TP)。如图4.9所示，左边为流水线并行，模型的不同层被切分到不同的设备中；右边为张量并行，同一个层中的不同的参数被切分到不同的设备中进行计算。

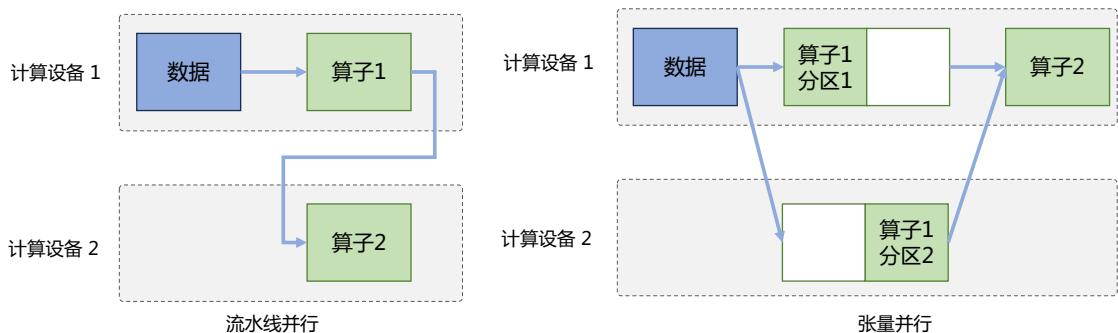


图 4.5 两节点数据并行训练系统样例

1. 流水线并行

流水线并行 (Pipeline Parallelism, PP) 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型模型的并行系统中，以有效解决单个计算设备内存不足的问题。图4.6展示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 F_1, F_2, F_3, F_4 分别代表四个前向路径，位于不同的设备上；而 B_4, B_3, B_2, B_1 则代表逆序的后向路径，也分别位于四个不同的设备上。然而，从图中可以看出，计算图中的下游设备 (Downstream Device) 需要长时间持续处于空闲状态，等待上游设备 (Upstream Device) 的计算完成，才能开始计算自身的任务。这种情况导致了设备的平均使用率大幅降低，形成了模型并行气泡 (Model Parallelism Bubble)，也称为流水线气泡 (Pipeline Bubble)。

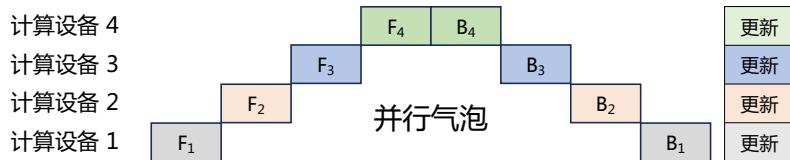


图 4.6 流水线并行样例

朴素流水线策略所产生的并行气泡，使得系统无法充分利用计算资源，降低了系统整体的计算效率。为了能够减少并行气泡，文献 [69] 提出了 GPipe 方法，将小批次 (Mini-batch) 进一步划分成更小的微批次 (Micro-batch)，利用流水线并行方案，每次处理一个微批次的数据。在当前阶段计算完成得到结果后，将该微批次的结果发送给下游设备，同时开始处理后一个微批次的数据，这样可以在一定程度上减少并行气泡。图4.7下半部分给出了这种模式，前向 F_0 计算被拆解为了 $F_{1,1}, F_{1,2}, F_{1,3}, F_{1,4}$ ，在计算设备 1 中计算完成 $F_{1,1}$ 后，会在计算设备 2 中开始进行 $F_{2,1}$ 计算，同时计算设备 1 中并行的开始 $F_{1,2}$ 的计算。相比于最原始的流水线并行方法，GPipe 流水线方法可以有效降低并行气泡。

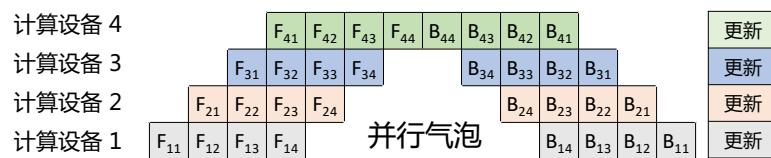


图 4.7 GPipe 策略流水线并行样例^[69]

GPipe 策略虽然可以减少一定的并行气泡，但是只有当一个 Mini-batch 中所有的前向计算完

成后，才能开始执行后向计算。因此还是会产生很多并行气泡，降低了系统的并行效率。Megatron-LM^[70] 提出了 1F1B 流水线策略，即一个前向通道和一个后向通道。1F1B 流水线策略引入了任务调度机制，使得下游设备能够在等待上游计算的同时执行其他可并行的任务，从而提高设备的利用率。1F1B 给出了非交错式和交错式两种方式调度方式，如图4.8所示。

1F1B 非交错式调度模式可分为三个阶段。首先是热身阶段，在该阶段中，计算设备中进行不同数量的前向计算。接下来的阶段是前向-后向阶段，计算设备按顺序执行一次前向计算，然后进行一次后向计算。最后一个阶段是后向阶段，计算设备在完成最后一次后向计算。相比于 GPipe 策略，非交错式调度模式在节省内存方面表现更好。然而，它需要与 GPipe 策略一样的时间来完成一轮计算。

1F1B 交错式调度模式要求 micro-batch 的数量是流水线阶段的整数倍。每个设备不再仅负责连续一段层次的计算，而是可以处理多个层的子集，这些子集被称为模型块。具体而言，在之前的模式中，设备 1 可能负责层 1-4，设备 2 负责层 5-8，以此类推。然而，在新的模式下，设备 1 可以处理层 1、2、9、10，设备 2 处理层 3、4、11、12，以此类推。这种模式下，每个设备在流水线中被分配到多个阶段。例如，设备 1 可能参与热身阶段、前向计算阶段和后向计算阶段的某些子集任务。每个设备可以并行执行不同阶段的计算任务，从而更好地利用流水线并行的优势。这种模式不仅在内存消耗方面表现出色，还能够提高计算效率，使得大型模型的并行系统能够更高效地完成计算任务。

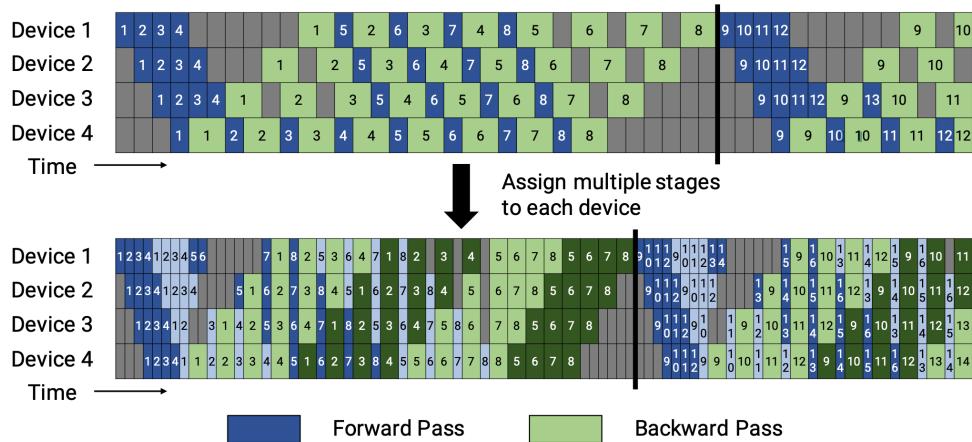


图 4.8 1F1B 流水线并行策略样例^[70]

PyTorch 中也包含了实现流水线的 API 函数 Pipe，具体实现参考“CLASS torch.distributed.pipeline.sync.Pip”。可以使用这个 API 构造一个包含两个线性层，分别放置在 2 个不同计算设备中的样例如下：

```

{
# Step 0: Need to initialize RPC framework first.
os.environ['MASTER_ADDR'] = 'localhost'
os.environ['MASTER_PORT'] = '29500'
torch.distributed.rpc.init_rpc('worker', rank=0, world_size=1)

# Step 1: build a model including two linear layers
fc1 = nn.Linear(16, 8).cuda(0)
fc2 = nn.Linear(8, 4).cuda(1)

# Step 2: wrap the two layers with nn.Sequential
model = nn.Sequential(fc1, fc2)

# Step 3: build Pipe (torch.distributed.pipeline.sync.Pipe)
model = Pipe(model, chunks=8)

# do training/inference
input = torch.rand(16, 16).cuda(0)
output_rref = model(input)
}

```

2. 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两个问题。大语言模型都是以 Transformer 结构为基础，Transformer 结构主要由以下三种算子构成：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^[68]，才可以实现将参数切分到不同的设备。

对于嵌入表示（Embedding）算子，如果总的词表数非常大，会导致单卡显存无法容纳 Embedding 层参数。举例来说，如果词表数量是 64000，嵌入表示维度为 5120，类型采用 32 位精度浮点数，那么整层参数需要的显存大约为 $64000 \times 5120 \times 4 / 1024 / 1024 = 1250MB$ ，反向梯度同样需要 1250MB，仅仅存储就需要将近 2.5GB。对于嵌入表示层的参数，可以按照词维度切分，每张卡只存储部分词向量，然后通过汇总各个设备上的部分词向量结果，从而得到完整的词向量结果。图4.9给出了单节点 Embedding 和两节点张量并行的示意图。在单节点上，执行 Embedding 操作，bz 是批次大小（batch size），Embedding 的参数大小为 [word_size, hidden_size]，计算得到 [bz, hidden_size] 张量。图4.9中 Embedding 张量并行示例将 Embedding 参数沿 word_size 维度，切分为两块，每块大小为 [word_size/2, hidden_size]，分别存储在两个设备上。当每个节点查询各自的词表时，如果无法查到，则该词的表示为 0，各自设备查询后得到 [bz, hidden_size] 结果张量，最后通过 AllReduce_Sum

通信^①，跨设备求和，得到完整的全量结果，可以看出，这里的输出结果和单卡执行的结果一致。

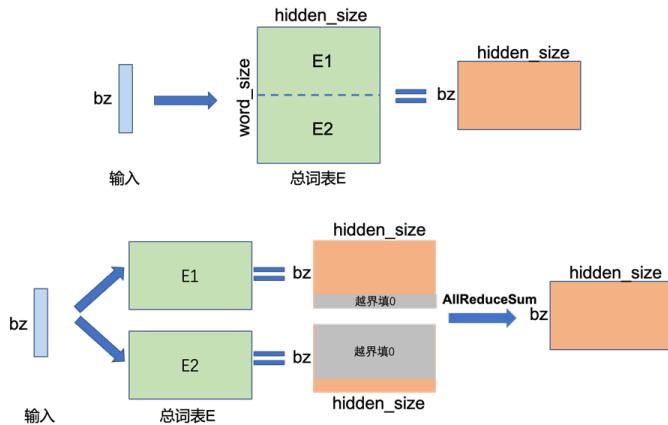


图 4.9 两节点 Embedding 算子张量并行示例

矩阵乘（MatMul）的张量并行要充分利用矩阵分块乘法原理。举例来说，要实现如下矩阵乘法 $\mathbf{Y} = \mathbf{X} \times \mathbf{A}$ ，其中 \mathbf{X} 是维度为 $M \times N$ 的输入矩阵， \mathbf{A} 是维度为 $N \times K$ 的参数矩阵， \mathbf{Y} 是结果矩阵，维度为 $M \times K$ 。如果参数矩阵 \mathbf{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 \mathbf{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卡计算结果。参数矩阵 \mathbf{A} 存在两种切分方式：

(1) 参数矩阵 \mathbf{A} 按列切块，将矩阵 \mathbf{A} 按列切成：

$$\mathbf{A} = [\mathbf{A}_1, \mathbf{A}_2] \quad (4.2)$$

(2) 参数矩阵 \mathbf{A} 按行切块，将矩阵 \mathbf{A} 按行切成：

$$\mathbf{A} = \begin{vmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{vmatrix} \quad (4.3)$$

图4.10给出了参数矩阵按列切分的示例，参数矩阵 \mathbf{A} 分别将 $\mathbf{A}_1, \mathbf{A}_2$ 放置在两个卡上。两张卡分别计算 $\mathbf{Y}_1 = \mathbf{X} \times \mathbf{A}_1$ 和 $\mathbf{Y}_2 = \mathbf{X} \times \mathbf{A}_2$ 。计算完成后，多卡间通信获取其它卡上的计算结果，拼接在一起得到最终的结果矩阵 \mathbf{Y} ，该结果在数学上与单卡计算结果上完全等价。

图4.11给出了参数矩阵按行切分的示例，为了满足矩阵乘法规则，输入矩阵 \mathbf{X} 需要按列切分 $\mathbf{X} = [\mathbf{X}_1 | \mathbf{X}_2]$ 。同时，将矩阵分块，分别放置在两张卡上，每张卡分别计算 $\mathbf{Y}_1 = \mathbf{X}_1 \times \mathbf{A}_1$ 和

^① 在本章 4.3.3 章节进行介绍

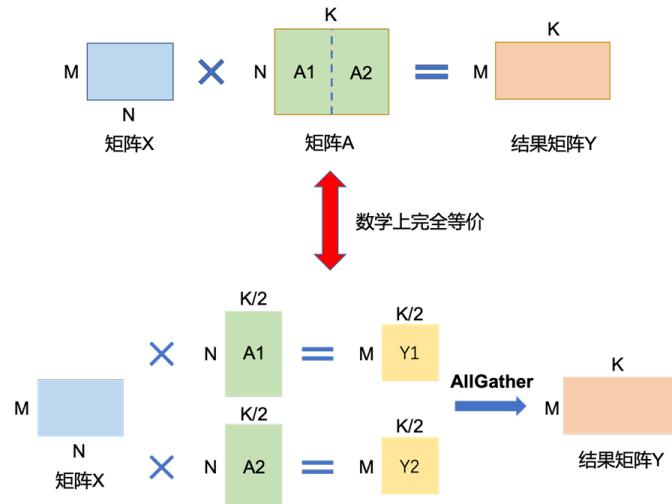


图 4.10 两节点矩阵乘算子张量并行按列切分示例

$Y_2 = X_2 \times A_2$ 。计算完成后，多卡间通信获取，归约其他卡上的计算结果，可以得到最终的结果矩阵 Y 。同样，这种切分方式，既可以保证数学上的计算等价性，并解决单卡显存无法容纳，又可以保证单卡通过拆分方式可以装下参数 A 的问题。

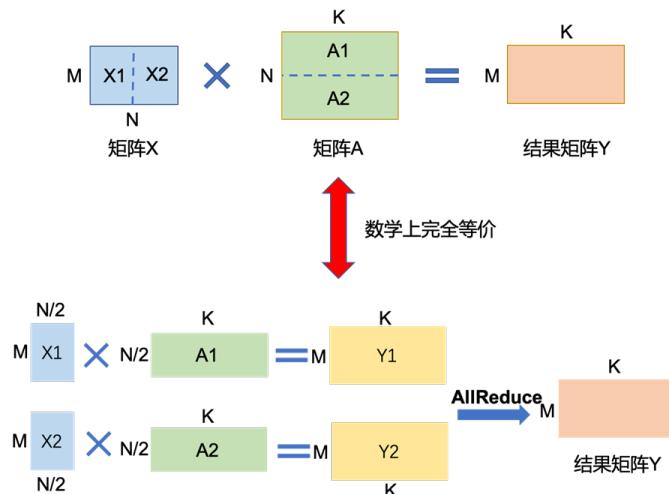


图 4.11 两节点矩阵乘算子张量并行按行切分示例

Transformer 中的 FFN 结构均包含两层全连接 (FC) 层, 即存在两个矩阵乘, 这两个矩阵乘分别采用上述两种切分方式, 如图4.12所示。对第一个 FC 层的参数矩阵按列切块, 对第二个 FC 层参数矩阵按行切块。这样第一个 FC 层的输出恰好满足第二个 FC 层数据输入要求 (按列切分), 因此可以省去第一个 FC 层后的 AllGather 通信操作。多头自注意力机制的张量并行与 FFN 类似, 因为具有多个独立的头, 因此相较于 FFN 更容易实现并行, 其矩阵切分方式如图4.13所示。

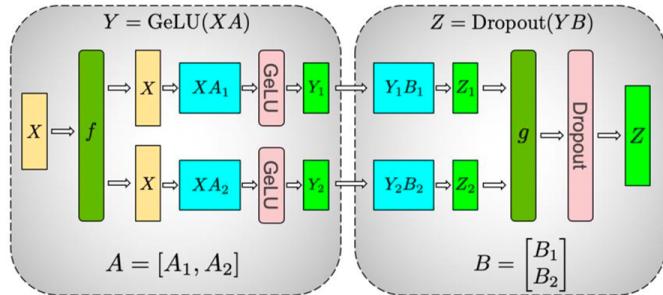


图 4.12 FFN 结构张量并行示意图--需要重绘^[68]

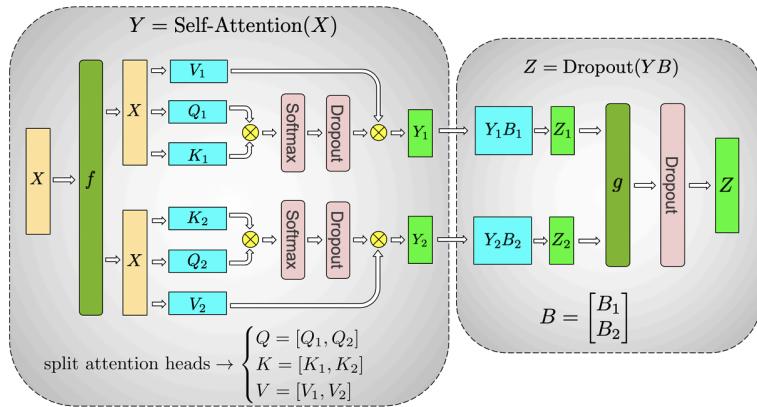


图 4.13 多头自注意力机制张量并行示意图-需要重绘^[68]

分类网络最后一层一般会选用 softmax 和 cross_entropy 算子来计算交叉熵损失 (Cross Entropy Loss)。如果类别数量非常大, 会导致单卡显存无法存储和计算 logit 矩阵。针对这一类算子, 可以按照类别维度切分, 同时通过中间结果通信, 得到最终的全局的交叉熵损失。首先计算的是 softmax

值，公式如下：

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - x_{max}}}{\sum_j e^{x_j - x_{max}}} = \frac{e^{x_i - x_{max}}}{\sum_N \sum_j e^{x_j - x_{max}}} \quad (4.4)$$

$$x_{max} = \max_p (\max_k (x_k)) \quad (4.5)$$

其中， p 表示张量并行的设备号。得到 softmax 之后，同时对标签 target 按类别切分，每个设备得到部分 loss，最后在进行一次通信，得到所有类别的 loss。整个过程，只需要进行三次少量的通信，就可以完成交叉熵损失的计算。

PyTorch 提供了细粒度张量级别的并行 API，DistributedTensor。也提供了粗粒度模型层面的 API 对“nn.Module”进行张量并行。通过以下几行代码就可以实现对一个大的张量进行分片：

```
import torch
from torch.distributed._tensor import DTensor, DeviceMesh, Shard, distribute_tensor

# construct a device mesh with available devices (multi-host or single host)
device_mesh = DeviceMesh("cuda", [0, 1, 2, 3])
# if we want to do row-wise sharding
rowwise_placement=[Shard(0)]
# if we want to do col-wise sharding
colwise_placement=[Shard(1)]

big_tensor = torch.randn(888, 12)
# distributed tensor returned will be sharded across the dimension specified in placements
rowwise_tensor = distribute_tensor(big_tensor, device_mesh=device_mesh, placements=rowwise_placement)
```

对于像“nn.Linear”这样已经有“torch.Tensor”作为参数的模块，也提供了模块级 API “distribute_module”在模型层面进行张量并行，参考代码如下：

```
import torch
from torch.distributed._tensor import DeviceMesh, Shard, distribute_tensor,distribute_module

class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 8)
        self.fc2 = nn.Linear(8, 8)
        self.relu = nn.ReLU()

    def forward(self, input):
        return self.relu(self.fc1(input) + self.fc2(input))

mesh = DeviceMesh(device_type="cuda", mesh=[[0, 1], [2, 3]])

def shard_params(mod_name, mod, mesh):
    rowwise_placement = [Shard(0)]
```

```

def to_dist_tensor(t): return distribute_tensor(t, mesh, rowwise_placement)
mod._apply(to_dist_tensor)

sharded_module = distribute_module(MyModule(), mesh, partition_fn=shard_params)

def shard_fc(mod_name, mod, mesh):
    rowwise_placement = [Shard(0)]
    if mod_name == "fc1":
        mod.weight = torch.nn.Parameter(distribute_tensor(mod.weight, mesh, rowwise_placement))

sharded_module = distribute_module(MyModule(), mesh, partition_fn=shard_fc)

```

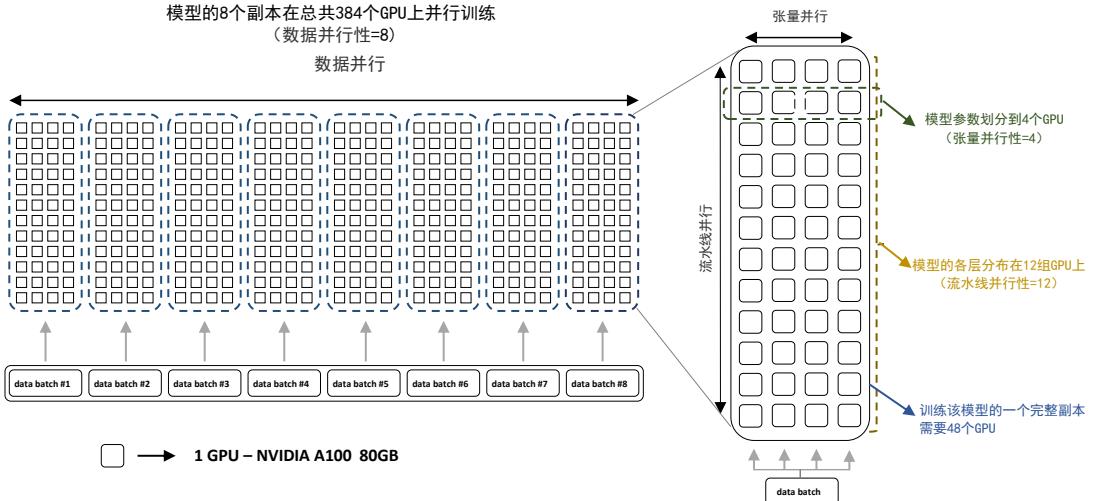
4.2.3 混合并行

混合并行 (Hybrid Parallelism, HP) 是将多种并行策略如数据并行、流水线并行和张量并行等进行混合使用。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，以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卡之间进行高速通信带宽。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来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，提升整体训练速度。通过数据并行，将训练数据分发到多组服务器上进行并行处理，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 使用了 Megatron-DeepSpeed^[71] 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^[72] 提供 ZeRO 优化器、模型流水线以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，如图4.14所示。BLOOM 模型训练使用了由 48 个 NVIDIA DGX-A100 服务器组成的集群，每个 DGX-A100 服务器包含 8 张 NVIDIA A100 80GB GPU，总计包含 384 张。BLOOM 训练采用的策略是首先将集群分为 48 个一组，进行数据并行。接下来，模型整体被分为 12 个阶段，进行流水线并行。每个阶段的模型被划分到 4 张 GPU 中，进行张量并行。同时 BLOOM 也使用了 ZeRO（零冗余优化器）^[73] 进一步降低了模型对显存的占用。用了通过上述四个步骤可以实现数百个 GPU 的高效并行计算。

4.2.4 计算设备内存优化

当前大语言模型训练通常采用 Adam 优化算法，除了需要每个参数梯度之外，还需要一阶动量 (Momentum) 和二阶动量 (Variance)。虽然 Adam 优化算法相较 SGD 算法通常效果更好也更稳定，但是对计算设备内存的占用显著增大。为了降低内存占用，大多数系统已经采用了混合精度训练 (Mixed Precision Training) 方式，即同时存在 FP16 (16 位浮点数) 或者 BF16 (Bfloat16) 和 FP32 (32 位浮点数) 两种格式的数值。FP32、FP16 和 BP16 表示如图4.15所示。FP32 中第 31

图 4.14 BLOOM 并行结构^[31]

位为符号位，第 30 到第 23 位用于表示指数，第 22 到第 0 位用于表示尾数。FP16 中第 15 位为符号位，第 14 到第 10 位用于表示指数，第 9 到第 0 位用于表示尾数。BF16 中第 15 位为符号位，第 14 到第 7 位用于表示指数，第 6 到第 0 位用于表示尾数。由于 FP16 的值区间比 FP32 的值区间小很多，所以在计算过程中很容易出现上溢出和下溢出。BF16 相较于 FP16 以精度换取更大的值区间范围。但是，由于 FP16 和 BP16 对 FP32 精度相较低，训练过程中可能会出现梯度消失和模型不稳定的问题。因此，需要使用一些技术来解决这些问题，例如动态损失缩放（Dynamic Loss Scaling）和混合精度优化器（Mixed Precision Optimizer）等。

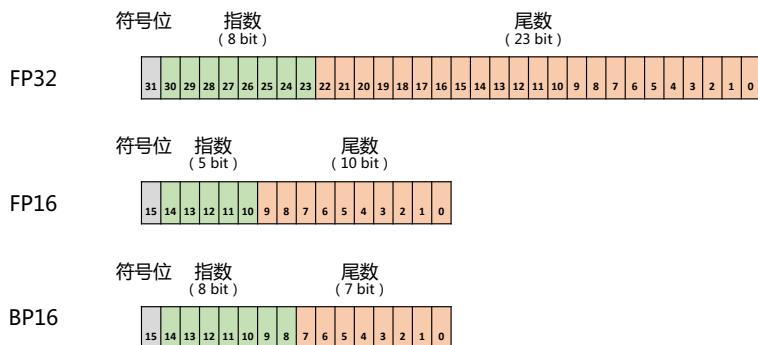


图 4.15 FP32、FP16 和 BP16 表示

混合精度优化的过程如图4.16所示。Adam优化器状态包括采用FP32保存的模型参数备份，一阶动量和二阶动量也都采用FP32格式存储。假设模型参数量为 Φ ，模型参数和梯度都是用FP16格式存储，则共需要 $2\Phi + 2\Phi + (4\Phi + 4\Phi + 4\Phi) = 16\Phi$ 字节存储。其中Adam状态占比75%。动态损失缩放反向传播前，将损失变化($dLoss$)手动增大 2^K 倍，因此反向传播时得到的激活函数梯度则不会溢出；反向传播后，将权重梯度缩 2^K 倍，恢复正常值。举例来说，对于包含75亿个参数模型，如果用FP16格式，只需要15GB计算设备内存，但是在训练阶段模型状态实际上需要耗费120GB。计算卡内存占用中除了模型状态之外，还有剩余状态(Residual States)，包括激活值(Activation)、各种临时缓冲区(Buffer)以及无法使用的显存碎片(Fragmentation)等。由于激活值可以用检查点(Activation Checkpointing)方式使得激活值内存占用大幅度减少，因此如何减少模型状态尤其是Adam优化器状态是解决内存占用问题的关键。

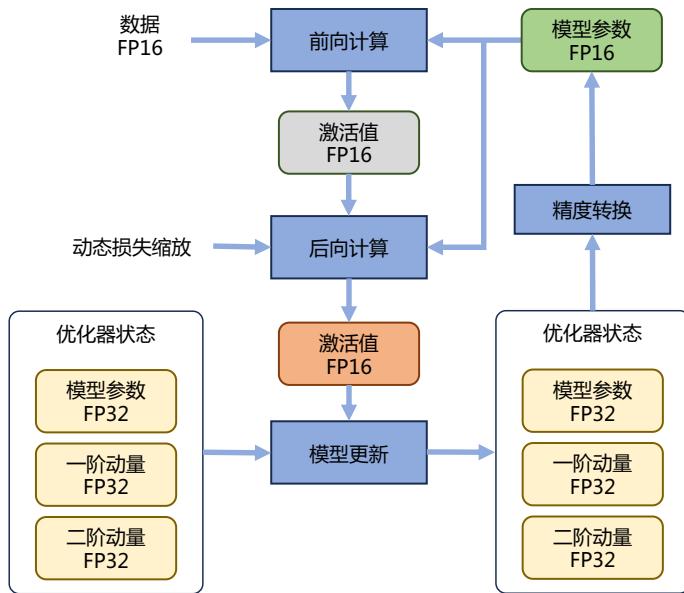


图 4.16 混合精度优化过程

零冗余优化器（Zero Redundancy Data Parallelism, ZeRO）目标就是针对模型状态的存储进行去除冗余的优化^[73–75]。ZeRO 使用分区的方法，即将模型状态量分割成多个分区，每个计算设备只保存其中的一部分。这样整个训练系统内只需要维护一份模型状态，减少了内存消耗和通信开销。具体来说，如图4.17所示，ZeRO 包含以下三种方法：

- 对 Adam 优化器状态进行分区，图4.17中 P_{os} 部分。模型参数和梯度依然是每个计算设备保存一份。此时，每个计算设备所需内存是 $4\Phi + \frac{12\Phi}{N}$ 字节，其中 N 是计算设备总数。当 N 比

较大时，每个计算设备占用内存趋向于 $4\Phi B$ ，也就是原来 $16\Phi B$ 的 $1/4$ 。

- 对模型梯度进行分区，图4.17中的 P_{os+g} 。模型参数依然是每个计算设备保存一份。此时，每个计算设备所需内存是 $2\Phi + \frac{2\Phi+12\Phi}{N}$ 字节。当 N 比较大时，每个计算设备占用内存趋向于 $2\Phi B$ ，也就是原来 $16\Phi B$ 的 $1/8$ 。
- 对模型参数也进行分区，图4.17中的 P_{os+g+p} 。此时，每个计算设备所需内存是 $\frac{16\Phi}{N} B$ 。当 N 比较大时，每个计算设备占用内存趋向于 0。

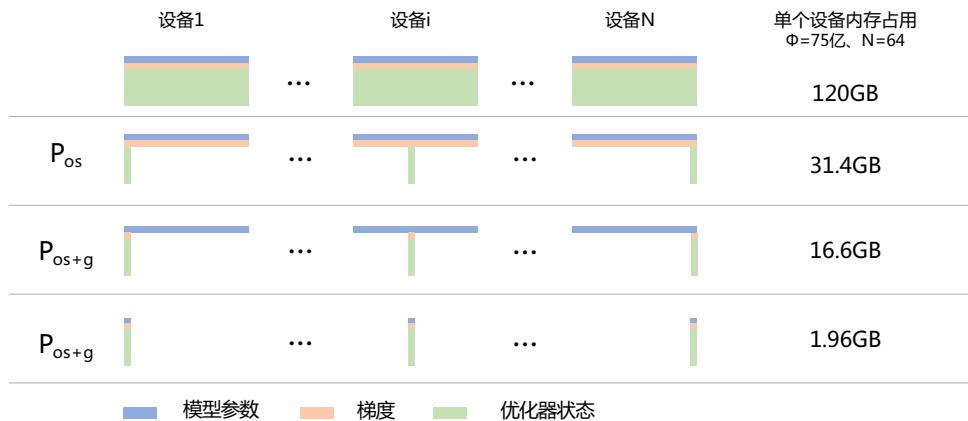


图 4.17 ZeRO 零冗余优化器

在 DeepSpeed 框架中， P_{os} 对应 Zero-1， P_{os+g} 对应 Zero-2， P_{os+g+p} 对应 Zero-3。文献 [75] 中也对 ZeRO 优化方法所带来的通信量增加情况进行了分析，Zero-1 和 Zero-2 对整体通信量没有影响，对通讯有一定延迟影响，但是整体性能影响很小。Zero-3 所需的通信量则是正常通信量的 1.5 倍。

PyTorch 中也实现了 ZeRO 优化方法，可以使用 `ZeroRedundancyOptimizer` 调用，也可与 `torch.nn.parallel.DistributedDataParallel` 结合使用，以减少每个计算设备的内存峰值消耗。使用 `ZeroRedundancyOptimizer` 的参考代码如下所示：

```

import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import torch.optim as optim
from torch.distributed.optim import ZeroRedundancyOptimizer
from torch.nn.parallel import DistributedDataParallel as DDP

def print_peak_memory(prefix, device):

```

```

if device == 0:
    print(f"{prefix}: {torch.cuda.max_memory_allocated(device) // 1e6}MB ")

def example(rank, world_size, use_zero):
    torch.manual_seed(0)
    torch.cuda.manual_seed(0)
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '29500'
    # create default process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

    # create local model
    model = nn.Sequential(*[nn.Linear(2000, 2000).to(rank) for _ in range(20)])
    print_peak_memory("Max memory allocated after creating local model", rank)

    # construct DDP model
    ddp_model = DDP(model, device_ids=[rank])
    print_peak_memory("Max memory allocated after creating DDP", rank)

    # define loss function and optimizer
    loss_fn = nn.MSELoss()
    if use_zero:
        optimizer = ZeroRedundancyOptimizer( # 这里使用了 ZeroRedundancyOptimizer
            ddp_model.parameters(),
            optimizer_class=torch.optim.Adam, # 包装了 Adam
            lr=0.01
        )
    else:
        optimizer = torch.optim.Adam(ddp_model.parameters(), lr=0.01)

    # forward pass
    outputs = ddp_model(torch.randn(20, 2000).to(rank))
    labels = torch.randn(20, 2000).to(rank)
    # backward pass
    loss_fn(outputs, labels).backward()

    # update parameters
    print_peak_memory("Max memory allocated before optimizer step()", rank)
    optimizer.step()
    print_peak_memory("Max memory allocated after optimizer step()", rank)

    print(f"params sum is: {sum(model.parameters()).sum()}")


def main():
    world_size = 2
    print("== Using ZeroRedundancyOptimizer ==")
    mp.spawn(example,
              args=(world_size, True),
              nprocs=world_size,
              join=True)

    print("== Not Using ZeroRedundancyOptimizer ==")

```

```

mp.spawn(example,
          args=(world_size, False),
          nprocs=world_size,
          join=True)

if __name__=="__main__":
    main()

```

执行上述代码，可以得到如下输出：

```

==== Using ZeroRedundancyOptimizer ===
Max memory allocated after creating local model: 335.0MB
Max memory allocated after creating DDP: 656.0MB
Max memory allocated before optimizer step(): 992.0MB
Max memory allocated after optimizer step(): 1361.0MB
params sum is: -3453.6123046875
params sum is: -3453.6123046875
==== Not Using ZeroRedundancyOptimizer ===
Max memory allocated after creating local model: 335.0MB
Max memory allocated after creating DDP: 656.0MB
Max memory allocated before optimizer step(): 992.0MB
Max memory allocated after optimizer step(): 1697.0MB
params sum is: -3453.6123046875
params sum is: -3453.6123046875

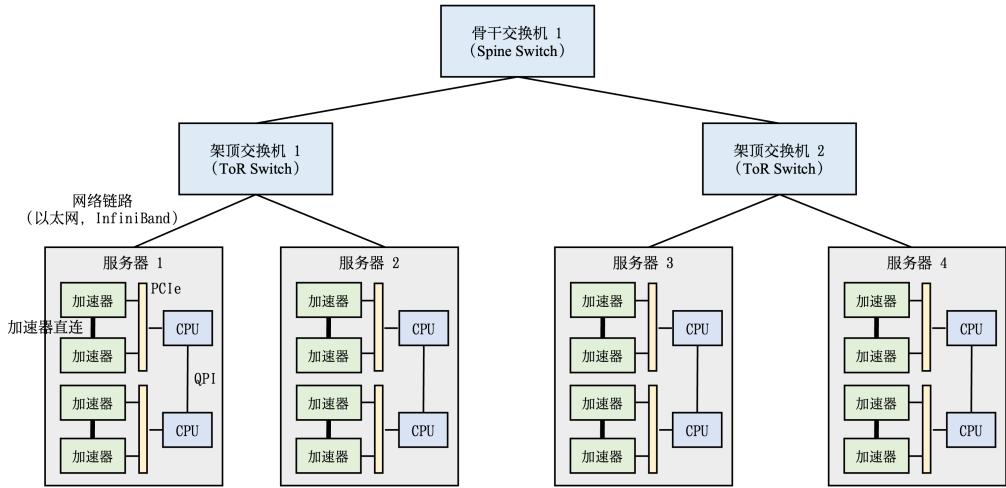
```

可以看到，在每次迭代之后，无论是否使用 ZeroRedundancyOptimizer，模型参数都使用了同样内存。当启用 ZeroRedundancyOptimizer 来封装 Adam 优化器后，优化器的 step() 操作的内存峰值消耗是 Adam 内存消耗的一半。

4.3 分布式训练的集群架构

分布式训练需要使用由多台服务器组成的计算集群（Compute Cluster）完成。而集群的架构也需要根据分布式系统、大语言模型结构、优化算法等综合因素进行设计。典型的计算集群的硬件组成如图4.18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 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）连接网络。在架顶交换机满载的情况下，可以通过在架顶交换机间增加骨干交换机（Spine Switch）进一步接入新的机柜。这种连接服务器的拓扑结构往往是一个多层树（Multi-Level Tree）。

多层树结构集群中跨机柜通信（Cross-Rack Communication）往往会有网络瓶颈。我们以包含 1750 亿参数的 GPT-3 模型为例，每一个参数使用 32 位浮点数表示，那么每一轮训练迭代训练中，每个模型副本（Model Replica）会生成 700GB（即 $175G \times 4\text{ Bytes} = 700\text{GB}$ ）的本地梯度数据。假如采用包含 1024 卡的计算集群，包含 128 个模型副本，那么至少需要传输 89.6TB（即 $700\text{GB} \times 128$

图 4.18 典型用于分布式训练的计算集群硬件组成^[66]

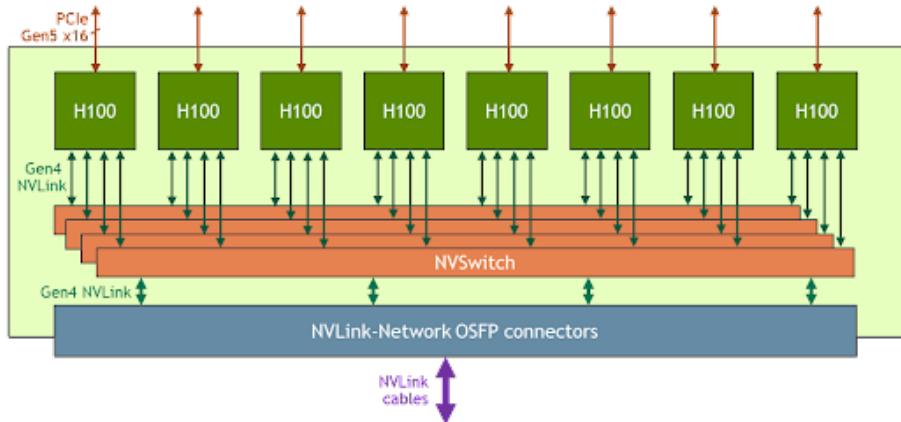
= 89.6TB) 的梯度数据。这会造成严重的网络通信瓶颈。因此，针对大语言模型分布式训练，目前通常采用胖树^[76] (Fat-Tree) 拓扑结构，试图实现网络带宽的无收敛。此外，采用 InfiniBand (IB) 技术搭建高速网络，单个 InfiniBand 链路可以提供 200Gb/s 或者 400Gb/s 带宽。NVIDIA 的 DGX 系列服务器提供单机 1.6Tb (200Gb×8) 网络带宽，HGX 系列服务器网络带宽更是可以达到 3.2Tb (400Gb×8)。

单个服务器内通常由 2 到 16 个计算加速设备组成，这些计算加速设备之间的通讯带宽也是影响分布式训练的重要因素。如果这些计算加速设备通过服务器 PCI 总线互联，会造成服务器内部计算加速设备之间通讯瓶颈。目前 PCIe 5.0 总线也只能提供 128GB/s 的带宽，而 NVIDIA H100 采用高带宽内存 (High-Bandwidth Memory, HBM) 可以提供 3350GB/s 的带宽。因此，服务器内部通常也采用了异构网络架构。NVIDIA HGX H100 8 GPU 服务器，采用了 NVLink 和 NVSwitch (NVLink 交换机) 技术，如图4.19所示。每个 H100 GPU 都有多个 NVLink 端口，并连接到所有四个 NVSwitch 上。每个 NVSwitch 都是一个完全无阻塞的交换机，完全连接所有八个 H100 计算加速卡。NVSwitch 的这种完全连接的拓扑结构，使得服务器内任何 H100 加速卡之间都可以达到 900GB/s 双向通信速度。

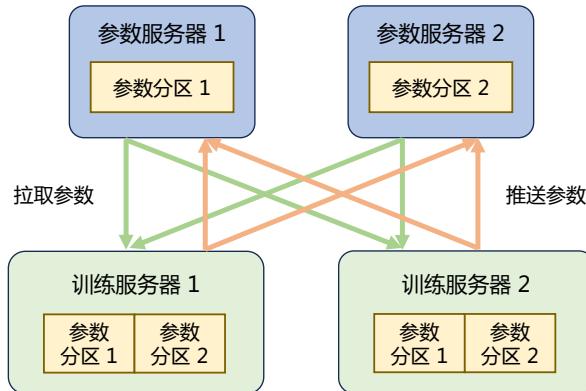
在由高速网络组成的计算集群上构建分布式训练系统主要有两种常见架构：参数服务器架构 (Parameter Server, PS) 和去中心化架构 (Decentralized Network)。接下来将分别介绍这两种架构。

4.3.1 参数服务器架构

参数服务器 (Parameter Server, PS) 架构的分布式训练系统中有两种服务器角色：训练服务器和参数服务器。参数服务器需要提供充足内存资源和通信资源，训练服务器需要提供大量的计算

图 4.19 NVIDIA HGX H100 8-GPU NVLink 和 NVSwitch 连接框图^[66]

资源。图4.20展示了一个具有参数服务器的分布式训练集群的示意图。该集群包括两个训练服务器和两个参数服务器。假设我们有一个可分为两个参数分区的模型，每个分区由一个参数服务器负责进行参数同步。在训练过程中，每个训练服务器都拥有完整的模型，并根据将分配到此服务器的训练数据集切片（Dataset Shard）进行计算，将得的梯度推送到相应的参数服务器。参数服务器会等待两个训练服务器都完成梯度推送，然后开始计算平均梯度，并更新参数。之后，参数服务器会通知训练服务器拉取最新的参数，并开始下一轮训练迭代。

图 4.20 参数服务器模式示例^[66]

参数服务器架构分布式训练过程可以细分为同步训练和异步训练两种模式：

- 同步训练：训练服务器在完成一个小批次的训练后，将梯度推送给参数服务器。参数服务器在接收到所有训练服务器的梯度后，进行梯度聚合和参数更新。
- 异步训练：训练服务器在完成一个小批次的训练后，将梯度推送给参数服务器。但是参数服务器不再等待接收所有训练服务器的梯度，而是直接基于已接收到的梯度进行参数更新。

同步训练的过程中，因为参数服务器会等待所有训练服务器完成当前小批次的训练，有诸多的等待或同步机制，导致整个训练速度较慢。异步训练去除了训练过程中的等待机制，训练服务器可以独立地进行参数更新，训练速度得到了极大的提升。但是因为引入了异步更新的机制会导致训练效果有所波动。选择适合的训练模式应根据具体情况和需求来进行权衡。

4.3.2 去中心化架构

去中心化（Decentralized Network）架构则采用集合通信中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。由于节点之间可以并行地进行训练和通信，去中心化架构可以显著降低通信开销，并减少通信墙的影响。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（Collective communication, CC）技术来实现，常用通信原语包括 Broadcast、Scatter、Reduce、All-Reduce、Gather、All-Gather、Reduce-Scatter、All-to-All 等。本章第 4.2 节中介绍的大语言模型训练所使用的分布式训练并行策略都，大都是使用去中心化架构，并利用集合通信进行实现。

下面我们介绍一些常见的集合通信原语：

- **Broadcast**：主节点把自身的数据发送到集群中的其他节点。分布式训练系统中常用于网络参数的初始化。如图4.21所示，计算设备 1 将大小为 $1 \times N$ 的张量进行广播，最终每张卡输出均为 $[1 \times N]$ 的矩阵。

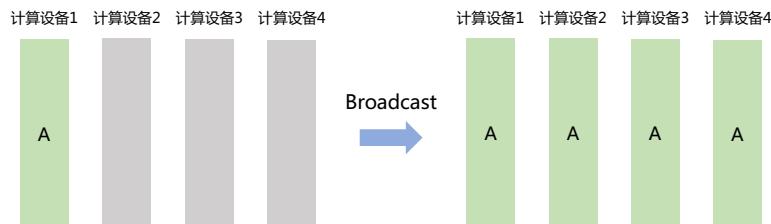


图 4.21 集合通信 Broadcast 原语示例

- **Scatter**：主节点将数据进行划分并散布至其他指定的节点。Scatter 与 Broadcast 非常相似，不同是而 Scatter 是将数据的不同部分，按需发送给所有的进程。如图4.22所示，计算设备 1 将大小为 $1 \times N$ 的张量分为 4 份后发送到不同节点。
- **Reduce**：是一系列简单运算操作的统称，是将不同节点上的计算结果进行聚合（Aggregation），

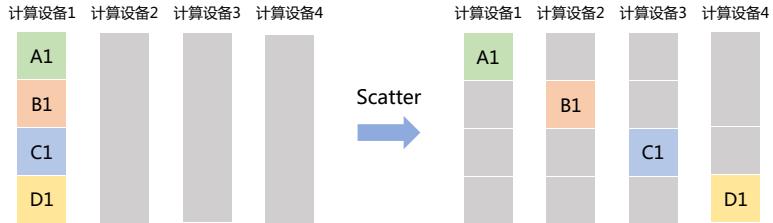


图 4.22 集合通信 Scatter 原语示例

可以细分为：SUM、MIN、MAX、PROD、LOR 等类型的规约操作。如图4.23所示，Reduce Sum 操作将所有其它计算设备上的数据汇聚到计算设备 1，并执行求和操作。

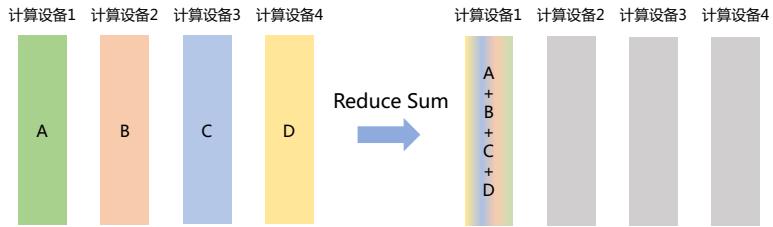


图 4.23 集合通信 Reduce Sum 原语示例

- **All Reduce**: 在所有的节点上都应用同样的 Reduce 操作。All Reduce 操作可通过单节点上 Reduce + Broadcast 操作完成。如图4.24所示，All Reduce Sum 操作将所有计算设备上的数据汇聚到各个计算设备中，并执行求和操作。

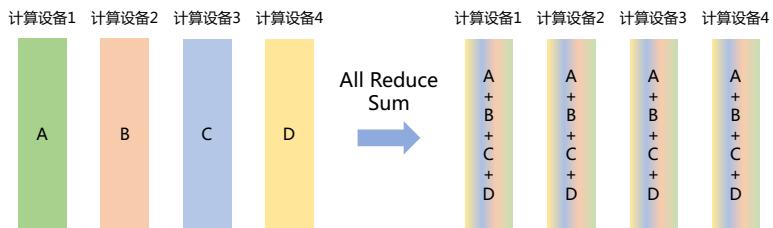


图 4.24 集合通信 All Reduce Sum 原语示例

- **Gather**: 将多个节点上的数据收集到单个节点上，Gather 可以理解为反向的 Scatter。如图4.25所示，Gather 操作将所有计算设备上的数据收集到计算设备 1 中。
- **All Gather**: 将所有节点上收集其他所有节点上的数据，All Gather 相当于一个 Gather 操作之

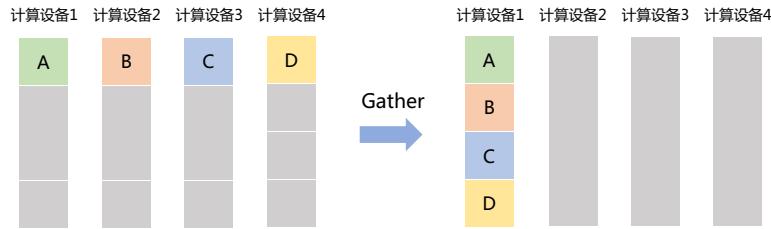


图 4.25 集合通信 Gather 原语示例

后跟着一个 Bcast 操作。如图4.25所示，All Gather 操作将所有计算设备上的数据收集到每个计算设备中。

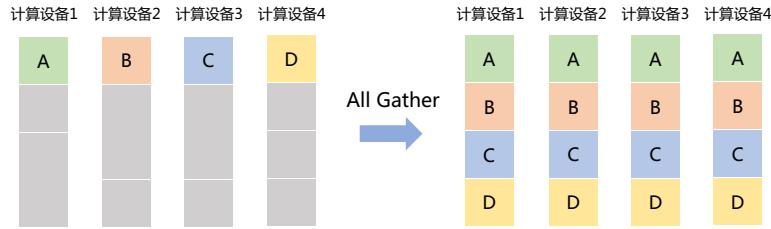


图 4.26 集合通信 All Gather 原语示例

- **Reduce Scatter**: 将每个节点中的张量切分为多个块，每个块分配给不同的节点。接收到的块会在每个节点上进行特定的操作，例如求和、取平均值等。如图4.27所示，每个计算设备都将其中的张量切分为 4 块，并分发到 4 个不同的计算设备中，每个计算设备分别对接收到的分块进行特定操作。

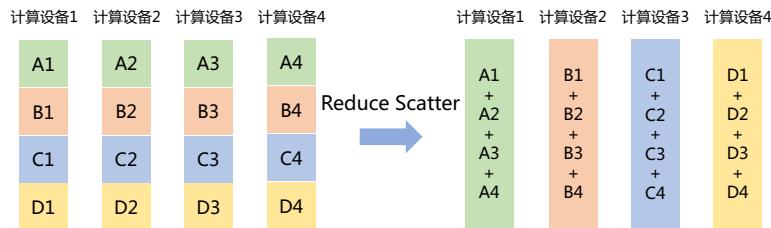


图 4.27 集合通信 Reduce Scatter 原语示例

- **All to All**: 将每个节点的张量切分为多个块，每个块分别发送给不同的节点。如图4.28所示，每个计算设备都将其中的张量切分为 4 块，并分发到 4 个不同的计算设备中。

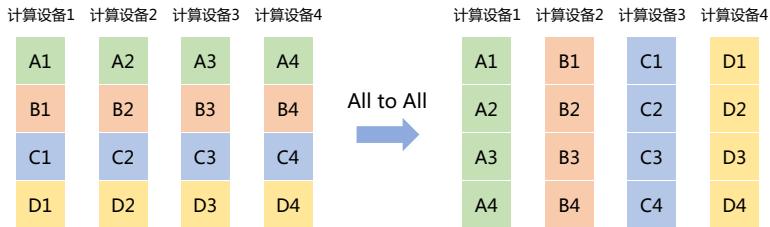


图 4.28 集合通信 All to All 原语示例

分布式集群中网络硬件多种多样，包括以太网、InfiniBand 网络等，Pytorch 等深度学习框架通常不直接操作硬件，而是使用通信库。常用的通信库包括 MPI、GLOO 和 NCCL 等，可以根据具体情况选择和配置。MPI (Message Passing Interface) 是一种广泛使用的并行计算通信库，常用于在多个进程之间进行通信和协调。Gloo 是 FaceBook 推出的一个类似 MPI 的集合通信库 (Collective Communications Library)，也大体遵照 MPI 提供的接口规定，实现了包括点对点通信，集合通信等相关接口，支持 CPU 和 GPU 上的分布式训练。NCCL (NVIDIA Collective Communications Library) 是 NVIDIA 开发的高性能 GPU 间通信库，专门用于在多个 GPU 之间进行快速通信和同步，因为 NCCL 则是 NVIDIA 基于自身硬件定制的，能做到更有针对性且更方便优化，故在 NVIDIA 硬件上，NCCL 的效果往往比其它的通信库更好。MPI、GLOO 和 NCCL 对各类型通信原语在 GPU 和 CPU 上的支持情况如表4.1所示。在进行分布式训练时，根据所使用的硬件环境和需求，选择适当的通信库可以充分发挥硬件的优势并提高分布式训练的性能和效率。一般而言，如果是在 CPU 集群上进行训练时，可选择使用 MPI 或 Gloo 作为通信库；而如果是在 GPU 集群上进行训练，则可以选择 NCCL 作为通信库。

我们还是以 PyTorch 为例，介绍如何使用上述通信原语，完成多计算设备间通信。首先使用“`torch.distributed`”初始化分布式环境：

```

import os
from typing import Callable

import torch
import torch.distributed as dist

def init_process(rank: int, size: int, fn: Callable[[int, int], None], backend="gloo"):
    """Initialize the distributed environment."""
    os.environ["MASTER_ADDR"] = "127.0.0.1"
    os.environ["MASTER_PORT"] = "29500"
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size)

```

接下来使用“`torch.multiprocessing`”开启多个进程，本例中共开启了 4 个进程：

表 4.1 MPI、GLOO 和 NCCL 在 GPU 和 CPU 环境下对通信原语支持情况

通信原语	GLOO		MPI		NCCL	
	CPU	GPU	CPU	GPU	CPU	GPU
Send	✓	✗	✓	?	✗	✓
Receive	✓	✗	✓	?	✗	✓
Broadcast	✓	✓	✓	?	✗	✓
Scatter	✓	✗	✓	?	✗	✓
Reduce	✓	✗	✓	?	✗	✓
All Reduce	✓	✓	✓	?	✗	✓
Gather	✓	✗	✓	?	✗	✓
All Gather	✓	✗	✓	?	✗	✓
Reduce Scatter	✗	✗	✗	✗	✗	✓
All To All	✗	✗	✓	?	✗	✓
Barrier	✓	✗	✓	?	✗	✓

```

...
import torch.multiprocessing as mp

def func(rank: int, size: int):
    # each process will call this function
    continue

if __name__ == "__main__":
    size = 4
    processes = []
    mp.set_start_method("spawn")
    for rank in range(size):
        p = mp.Process(target=init_process, args=(rank, size, func))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

```

每个新开启的进程都会调用“init_process”，接下来再调用用户指定的函数“func”。这里我们以 All Reduce 为例：

```

def do_all_reduce(rank: int, size: int):
    # create a group with all processors
    group = dist.new_group(list(range(size)))
    tensor = torch.ones(1)

```

```

dist.all_reduce(tensor, op=dist.ReduceOp.SUM, group=group)
# can be dist.ReduceOp.PRODUCT, dist.ReduceOp.MAX, dist.ReduceOp.MIN
# will output 4 for all ranks
print(f"[{rank}] data = {tensor[0]}")

...
for rank in range(size):
    # passing `hello_world`
    p = mp.Process(target=init_process, args=(rank, size, do_all_reduce))

...

```

根据 All Reduce 通信原语，在所有的节点上都应用同样的 Reduce 操作，可以得到如下输出：

```
[3] data = 4.0
[0] data = 4.0
[1] data = 4.0
[2] data = 4.0
```

4.4 DeepSpeed 实践

DeepSpeed^[72] 是一个由 Microsoft 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，使研究人员和工程师能够更快地迭代和探索新的深度学习模型和算法。它采用了多种技术手段来加速训练，包括模型并行化、梯度累积、动态精度缩放和本地模式混合精度等。此外，DeepSpeed 还提供了一些辅助工具，例如分布式训练管理、内存优化和模型压缩，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。DeepSpeed 是基于 PyTorch 构建的，因此将现有的 PyTorch 训练代码迁移到 DeepSpeed 上通常只需要进行简单的修改。这使得开发者可以快速利用 DeepSpeed 的优化功能来加速他们的训练任务。DeepSpeed 已经在许多大规模深度学习项目中得到了应用，包括语言模型、图像分类、目标检测等领域。大语言模型 BLOOM^[31] 模型（1750 亿参数）和 MT-NLG^[71] 模型（5400 亿参数）都是采用 DeepSpeed 框架完成训练。

DeepSpeed 主要优势在于支持更大规模的模型、提供了更多的优化策略和工具。DeepSpeed 通过实现三种并行方法的灵活组合，即 ZeRO 支持的数据并行、流水线并行和张量并行，可以应对不同工作负载的需求。特别是通过 3D 并行性的支持，DeepSpeed 可以处理具有万亿参数的超大型模型。DeepSpeed 引入了 ZeRO-Offload，使单个 GPU 能够训练比其显存大小大 10 倍的模型。此外，为了充分利用 CPU 和 GPU 内存来训练大型模型，DeepSpeed 还扩展了 ZeRO-2。此外，DeepSpeed 还提供了稀疏注意力核（Sparse Attention Kernel），支持处理包括文本、图像和语音等长序列输入的模型。DeepSpeed 还集成了 1 比特 Adam 算法（1-bit Adam），它可以只使用原始 Adam 算法 1/5 的通信量，同时达到与 Adam 类似的收敛率，可以显著提高分布式训练的效率，并降低通信开销。

DeepSpeed 的 3D 并行充分利用硬件架构特性，有效综合考虑了显存效率和计算效率两个方面。本章第 4.3 节介绍了分布式集群的硬件架构，可以看到目前分布式训练集群通常采用 NVIDIA DGX/HGX 节点，利用胖树网络拓扑结构构建计算集群。因此，每个节点内部 8 个计算加速设备之间具有非常高的通信带宽，但是节点之间的通信带宽则相对较低。由于张量并行是分布式训练策略中通信开销最大的，因此优先考虑将张量并行计算组放置在节点内以利用更大的节点内带宽。如果张量并行组不占满节点内的所有计算节点时，选择将数据并行组放置在节点内，否则就使用跨节点进行数据并行。流水线并行的通信量最低，因此可以使用跨节点调度流水线的各个阶段，降低通信带宽的要求。每个数据并行组需要通信的梯度量随着流水线和模型并行的规模线性减小，因此总通信量少于单纯使用数据并行。此外，每个数据并行组会在局部的一小部分计算节点内部独立进行通信，组间通信可以相互并行。通过减少通信量和增加局部性与并行性，数据并行通信的有效带宽有效增大。

图4.29给出了DeepSpeed 3D 并行策略示意图。图中给出了包含 32 个计算设备进行 3D 并行的例子。神经网络的各层分为 4 个流水线阶段。每个流水线阶段中的层在 4 个张量并行计算设备之间进一步划分。最后，每个流水线阶段有两个数据并行实例，使用 ZeRO 内存优化在这 2 个副本之间划分优化器状态量。

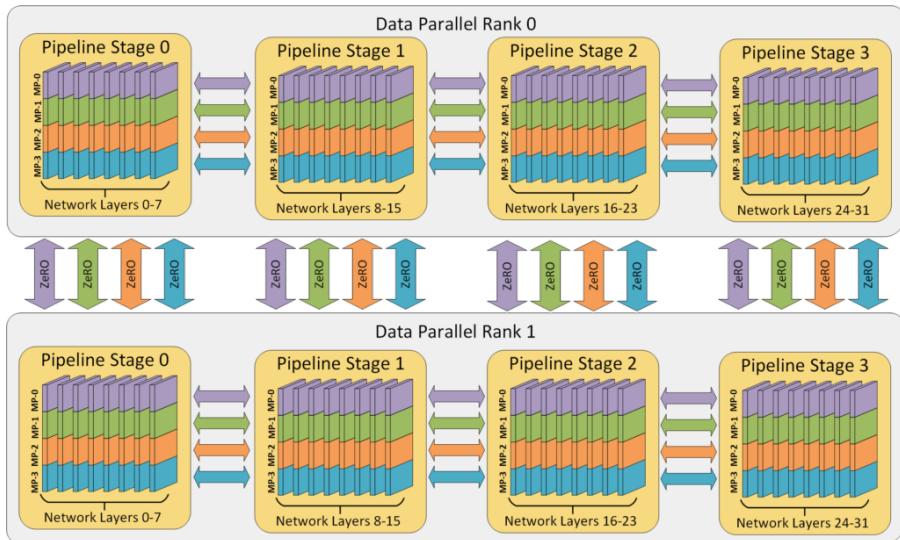


图 4.29 DeepSpeed 3D 并行策略示意图^[77]

DeepSpeed 软件架构如图4.30所示，主要包含三部分：

- APIs：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。通过“initialize”接口可以初始化引擎，并在参数中配

置训练参数和优化技术等。这些配置参数通常保存在名为“ds_config.json”的文件中。。

- 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、故障检测以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，包括 Ultrafast Transformer Lernels、fuse LAN kernels、Customary Deals 等。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。

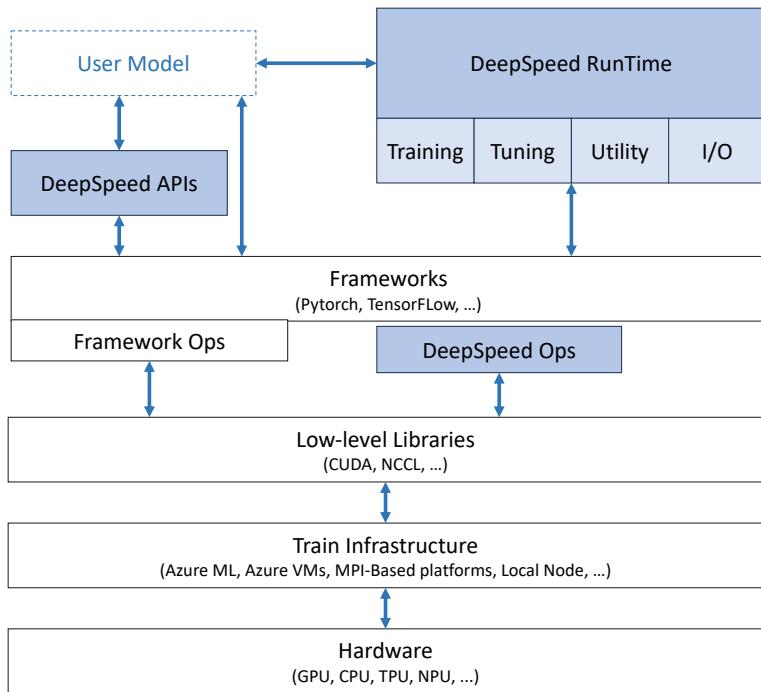


图 4.30 DeepSpeed 软件架构

4.4.1 基础概念

DeepSpeed 提供了分布式计算框架，首先需要明确几个重要的基础的概念：节点编号、全局进程编号、局部进程编号、全局总进程数和主节点。DeepSpeed 主节点（master_ip+master_port）负责协调所有其他节点和进程的工作，由主节点所在服务器的 IP 地址和主节点进程的端口号来确定主节点。主节点还负责监控系统状态、处理任务分配和结果汇总等任务，因此是整个系统的关键部分。节点编号（node_rank）是系统中每个节点的唯一标识符，用于区分不同计算机之间的通信。

全局进程编号（rank）是整个系统中的每个进程的唯一标识符，用于区分不同进程之间的通信。局部进程编号（local_rank）：是单个节点内的每个进程的唯一标识符，用于区分同一节点内的不同进程之间的通信。全局总进程数（world_size）是整个系统中运行的所有进程的总数，用于确定可以并行完成多少工作以及需要完成任务所需的资源数量。

在网络通信策略方面，DeepSpeed 提供了 MPI、GLOO 和 NCCL 等选项，可以根据具体情况进行选择和配置。DeepSpeed 配置文件中，在 optimizer 部分配置通信策略，以下是使用 OneBitAdam 优化器的配置样例，配置中其中使用了 nccl 通讯库：

```
{
  "optimizer": {
    "type": "OneBitAdam",
    "params": {
      "lr": 0.001,
      "betas": [
        0.8,
        0.999
      ],
      "eps": 1e-8,
      "weight_decay": 3e-7,
      "freeze_step": 400,
      "cuda_aware": false,
      "comm_backend_name": "nccl"
    }
  }
}
```

DeepSpeed 中也支持各多种类型 ZeRO 的分片机制，包括 ZeRO-0、ZeRO-1、ZeRO-2、ZeRO-3 以及 ZeRO-Infinity。ZeRO-0 禁用所有类型的分片，仅将 DeepSpeed 当作分布式数据并行使用；ZeRO-1 对优化器状态都进行分片，占用内容为原始的 1/4，通信容量与数据并行性相同；ZeRO-2 对优化器状态和梯度都进行分片，占用内容为原始的 1/8，通信容量与数据并行性相同；ZeRO-3：对优化器状态、梯度以及模型参数都进行分片，内存减少与数据并行度和复杂度成线性关系，同时通信容量是数据并行性的 1.5 倍；ZeRO-Infinity 是 ZeRO-3 的拓展，允许通过使用 NVMe 固态硬盘扩展 GPU 和 CPU 内存来训练大型模型。

以下是 DeepSpeed 使用 ZeRO-3 配置参数样例：

```
{
  "zero_optimization": {
    "stage": 3,
  },
  "fp16": {
    "enabled": true
  },
}
```

```

    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": 0.001,
            "betas": [
                0.8,
                0.999
            ],
            "eps": 1e-8,
            "weight_decay": 3e-7
        }
    },
    ...
}

```

如果希望在 ZeRO-3 基础上继续使用 ZeRO-Infinity 将优化器状态和计算转移到 CPU 中，可以在配置文件中按照方式如下配置：

```

{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu"
        }
    },
    ...
}

```

甚至可以进一步将模型参数也装载到 CPU 内存中，可以在配置文件中按照方式如下配置：

```

{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu"
        }
        "offload_param": {
            "device": "cpu"
        }
    },
    ...
}

```

如果希望将更多的内存装载到 NVMe 中，可以在配置文件中按照方式如下配置：

```
{  
    "zero_optimization": {  
        "stage": 3,  
        "offload_optimizer": {  
            "device": "nvme",  
            "nvme_path": "/nvme_data"  
        }  
        "offload_param": {  
            "device": "nvme",  
            "nvme_path": "/nvme_data"  
        }  
    },  
    [...]  
}
```

4.4.2 GPT 分布式训练实践

单机单卡

单机多卡 (with DataParallel)

多机多卡 (with DistributedDataParallel)

5. 有监督微调

5.1 有监督微调概述

5.2 任务范式统一

在深度学习的时代，多数自然语言处理 (NLP) 任务的建模已经融合到几种主流范式中。例如，我们通常采用序列标记范式来完成一系列任务，如词性标记、命名实体识别 (NER) 和组块分析等，并采用分类范式来完成情感分析等任务。随着预训练语言模型的快速发展，近年来出现了一种范式转换的新趋势，即通过对任务的输入输出形式进行修改，从而在新范式下完成一个 NLP 任务。在完成许多任务时，范式转换都取得了巨大的成功，并正成为提升模型性能的一种新兴方法。此外，其中一些范式在统一大量 NLP 任务方面显示出巨大的潜力，从而有可能构建一个单一的模型来处理不同的任务。

Paradigm Shift in Natural Language Processing Tian-Xiang Sun, Xiang-Yang Liu, Xi-Peng Qiu, Xuan-Jing Huang

FLAN

T0

5.3 提示学习与上下文学习

随着大模型 (GPT3, Instruction GPT, ChatGPT) 的横空出世，如何更高效地提示大模型也成了学术界与工业界的关注，因此 In-context learning 的方法在 NLP 领域十分火热。从时间线上看，它的演变历程大约是从 Prompt learning (2021 年初) 到 In-context learning (2022 年初)，但从方法原理上，他们却有很多相似之处。

GPT-3 论文 in-context learning 综述：A Survey on In-context Learning

5.4 提示数据构造

instructgpt 3.4 Human data collection SFT

Self-Instruct

5.5 高效模型微调

DeltaTuning: A Comprehensive Study of Parameter Efficient Methods for Pre-trained Language Models

LoRA: Low-Rank Adaptation of Large Language Models

5.6 Deepspeed-Chat SFT 实践

6. 强化学习

6.1 强化学习理论

6.2 奖励模型

6.3 近端策略优化

6.4 Deepspeed-Chat PPO 实践

7. 大语言模型应用

7.1 领域模型构建

7.2 大模型高效推理

7.3 LangChain

7.4 AutoGPT

8. 大语言模型评价

8.1 语言模型评价

语言模型最直接的测评方法就是使用模型计算测试集的概率，或者利用交叉熵（Cross-entropy）和困惑度（Perplexity）等派生测度。

对于一个平滑过的概率 $P(w_i|w_{i-n+1}^{i-1})$ 的 n 元语言模型，可以用下列公式计算句子 $P(s)$ 的概率：

$$P(s) = \prod_{i=1}^n P(w_i|w_{i-n+1}^{i-1}) \quad (8.1)$$

对于由句子 (s_1, s_2, \dots, s_n) 组成的测试集 T ，可以通过计算 T 中所有句子概率的乘积来得到整个测试集的概率：

$$P(T) = \prod_{i=1}^n P(s_i) \quad (8.2)$$

交叉熵的测度则是利用预测和压缩的关系进行计算。对于 n 元语言模型 $P(w_i|w_{i-n+1}^{i-1})$ ，文本 s 的概率为 $P(s)$ ，在文本 s 上 n 元语言模型 $P(w_i|w_{i-n+1}^{i-1})$ 的交叉熵为：

$$H_p(s) = -\frac{1}{W_s} \log_2 P(s) \quad (8.3)$$

其中， W_s 为文本 s 的长度，该公式可以解释为：利用压缩算法对 s 中的 W_s 个词进行编码，每一个编码所需要的平均比特位数。

困惑度的计算可以视为模型分配给测试集中每一个词汇的概率的几何平均值的倒数，它和交叉熵的关系为：

$$PP_s(s) = 2^{H_p(s)} \quad (8.4)$$

交叉熵和困惑度越小，语言模型性能就越好。不同的文本类型其合理的指标范围是不同的，对于英文来说， n 元语言模型的困惑度约在 50 到 1000 之间，相应的，交叉熵在 6 到 10 之间。

8.2 大语言模型评价

参考文献

- [1] Devlin J, Chang M W, Lee K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[C]//Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). 2019: 4171-4186.
- [2] Radford A, Narasimhan K, Salimans T, et al. Improving language understanding by generative pre-training[J].
- [3] Bengio Y, Ducharme R, Vincent P. A neural probabilistic language model[J]. Advances in neural information processing systems, 2000, 13.
- [4] Mikolov T, Karafiat M, Burget L, et al. Recurrent neural network based language model.[C]// Interspeech: volume 2. Makuhari, 2010: 1045-1048.
- [5] Pham N Q, Kruszewski G, Boleda G. Convolutional neural network language models[C]// Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. 2016: 1153-1162.
- [6] Sukhbaatar S, Weston J, Fergus R, et al. End-to-end memory networks[C]//Advances in neural information processing systems. 2015: 2440-2448.
- [7] Deng J, Dong W, Socher R, et al. Imagenet: A large-scale hierarchical image database[C]//2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009: 248-255.
- [8] Peters M, Neumann M, Iyyer M, et al. Deep contextualized word representations[C]//Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers): volume 1. 2018: 2227-2237.
- [9] Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners[J]. OpenAI blog, 2019, 1(8):9.

- [10] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]//Advances in Neural Information Processing Systems. 2017: 5998-6008.
- [11] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[J]. Advances in neural information processing systems, 2020, 33:1877-1901.
- [12] Chowdhery A, Narang S, Devlin J, et al. Palm: Scaling language modeling with pathways[J]. arXiv preprint arXiv:2204.02311, 2022.
- [13] Thoppilan R, De Freitas D, Hall J, et al. Lamda: Language models for dialog applications[J]. arXiv preprint arXiv:2201.08239, 2022.
- [14] Sanh V, Webson A, Raffel C, et al. Multitask prompted training enables zero-shot task generalization [J]. arXiv preprint arXiv:2110.08207, 2021.
- [15] Kaplan J, McCandlish S, Henighan T, et al. Scaling laws for neural language models[J]. arXiv preprint arXiv:2001.08361, 2020.
- [16] Zhao W X, Zhou K, Li J, et al. A survey of large language models[J]. arXiv preprint arXiv:2303.18223, 2023.
- [17] Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J]. The Journal of Machine Learning Research, 2020, 21(1):5485-5551.
- [18] Zhang Z, Han X, Liu Z, et al. Ernie: Enhanced language representation with informative entities[C]// Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. 2019: 1441-1451.
- [19] Sun Y, Wang S, Li Y, et al. Ernie: Enhanced representation through knowledge integration[J]. arXiv preprint arXiv:1904.09223, 2019.
- [20] Zeng W, Ren X, Su T, et al. Pangu- α : Large-scale autoregressive pretrained chinese language models with auto-parallel computation[J]. arXiv preprint arXiv:2104.12369, 2021.
- [21] Chung H W, Hou L, Longpre S, et al. Scaling instruction-finetuned language models[J]. arXiv preprint arXiv:2210.11416, 2022.
- [22] Ouyang L, Wu J, Jiang X, et al. Training language models to follow instructions with human feedback [J]. arXiv preprint arXiv:2203.02155, 2022.

- [23] Nakano R, Hilton J, Balaji S, et al. Webgpt: Browser-assisted question-answering with human feedback[J]. arXiv preprint arXiv:2112.09332, 2021.
- [24] Xue L, Constant N, Roberts A, et al. mt5: A massively multilingual pre-trained text-to-text transformer[C]//Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2021: 483-498.
- [25] Zhang Z, Gu Y, Han X, et al. Cpm-2: Large-scale cost-effective pre-trained language models[J]. AI Open, 2021, 2:216-224.
- [26] Sanh V, Webson A, Raffel C, et al. Multitask prompted training enables zero-shot task generalization [C]//ICLR 2022-Tenth International Conference on Learning Representations. 2022.
- [27] Nijkamp E, Pang B, Hayashi H, et al. Codegen: An open large language model for code with multi-turn program synthesis[J]. arXiv preprint arXiv:2203.13474, 2022.
- [28] Black S, Biderman S, Hallahan E, et al. Gpt-neox-20b: An open-source autoregressive language model[J]. arXiv preprint arXiv:2204.06745, 2022.
- [29] Zhang S, Roller S, Goyal N, et al. Opt: Open pre-trained transformer language models[J]. arXiv preprint arXiv:2205.01068, 2022.
- [30] Zeng A, Liu X, Du Z, et al. GLM-130b: An open bilingual pre-trained model[C/OL]//The Eleventh International Conference on Learning Representations (ICLR). 2023. <https://openreview.net/forum?id=-Aw0rrrPUF>.
- [31] Scao T L, Fan A, Akiki C, et al. Bloom: A 176b-parameter open-access multilingual language model [J]. arXiv preprint arXiv:2211.05100, 2022.
- [32] Taylor R, Kardas M, Cucurull G, et al. Galactica: A large language model for science[J]. arXiv preprint arXiv:2211.09085, 2022.
- [33] Muennighoff N, Wang T, Sutawika L, et al. Crosslingual generalization through multitask finetuning [J]. arXiv preprint arXiv:2211.01786, 2022.
- [34] Iyer S, Lin X V, Pasunuru R, et al. Opt-iml: Scaling language model instruction meta learning through the lens of generalization[J]. arXiv preprint arXiv:2212.12017, 2022.
- [35] Touvron H, Lavril T, Izacard G, et al. Llama: Open and efficient foundation language models[J]. arXiv preprint arXiv:2302.13971, 2023.

- [36] Taori R, Gulrajani I, Zhang T, et al. Stanford alpaca: An instruction-following llama model[J/OL]. GitHub repository, 2023. https://github.com/tatsu-lab/stanford_alpaca.
- [37] Chiang W L, Li Z, Lin Z, et al. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality[J]. See <https://vicuna.lmsys.org> (accessed 14 April 2023), 2023.
- [38] Geng X, Gudibande A, Liu H, et al. Koala: A dialogue model for academic research[EB/OL]. 2023 [2023-04-03]. <https://bair.berkeley.edu/blog/2023/04/03/koala/>.
- [39] Xu C, Guo D, Duan N, et al. Baize: An open-source chat model with parameter-efficient tuning on self-chat data[J]. arXiv preprint arXiv:2304.01196, 2023.
- [40] Diao S, Pan R, Dong H, et al. Lmflow: An extensible toolkit for finetuning and inference of large foundation models[J/OL]. GitHub repository, 2023. <https://optimalscale.github.io/LMFlow/>.
- [41] Wang H, Liu C, Xi N, et al. Huatuo: Tuning llama model with chinese medical knowledge[J]. arXiv preprint arXiv:2304.06975, 2023.
- [42] Cui Y, Yang Z, Yao X. Efficient and effective text encoding for chinese llama and alpaca[J]. arXiv preprint arXiv:2304.08177, 2023.
- [43] Anand Y, Nussbaum Z, Duderstadt B, et al. Gpt4all: Training an assistant-style chatbot with large scale data distillation from gpt-3.5-turbo[J/OL]. GitHub repository, 2023. <https://github.com/nomic-ai/gpt4all>.
- [44] Patil S G, Zhang T, Wang X, et al. Gorilla: Large language model connected with massive apis[J]. arXiv preprint arXiv:2305.15334, 2023.
- [45] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners[J]. arXiv preprint arXiv:2005.14165, 2020.
- [46] Zhou C, Liu P, Xu P, et al. Lima: Less is more for alignment[J]. arXiv preprint arXiv:2305.11206, 2023.
- [47] 张奇、桂韬、黄萱菁. 自然语言处理导论[M]. 上海: 电子工业出版社, 2023.
- [48] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C/OL]//Guyon I, Luxburg U V, Bengio S, et al. Advances in Neural Information Processing Systems: volume 30. Curran Associates, Inc., 2017. <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fdb053c1c4a845aa-Paper.pdf>.

- [49] Lewis M, Liu Y, Goyal N, et al. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension[C]//Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 2020: 7871-7880.
- [50] Zhang B, Sennrich R. Root mean square layer normalization[J]. Advances in Neural Information Processing Systems, 2019, 32.
- [51] Shazeer N. Glu variants improve transformer[J]. arXiv preprint arXiv:2002.05202, 2020.
- [52] Hendrycks D, Gimpel K. Gaussian error linear units (gelus)[J]. arXiv preprint arXiv:1606.08415, 2016.
- [53] Su J, Lu Y, Pan S, et al. Roformer: Enhanced transformer with rotary position embedding[J]. arXiv preprint arXiv:2104.09864, 2021.
- [54] Lin T, Wang Y, Liu X, et al. A survey of transformers[J/OL]. CoRR, 2021, abs/2106.04554. <https://arxiv.org/abs/2106.04554>.
- [55] Guo Q, Qiu X, Liu P, et al. Star-transformer[C]//Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). 2019: 1315-1325.
- [56] Beltagy I, Peters M E, Cohan A. Longformer: The long-document transformer[J]. arXiv preprint arXiv:2004.05150, 2020.
- [57] Ainslie J, Ontanon S, Alberti C, et al. Etc: Encoding long and structured inputs in transformers [C]//Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). 2020: 268-284.
- [58] Oord A v d, Li Y, Vinyals O. Representation learning with contrastive predictive coding[J]. arXiv preprint arXiv:1807.03748, 2018.
- [59] Zaheer M, Guruganesh G, Dubey K A, et al. Big bird: Transformers for longer sequences[J]. Advances in neural information processing systems, 2020, 33:17283-17297.
- [60] Roy A, Saffar M, Vaswani A, et al. Efficient content-based sparse attention with routing transformers [J]. Transactions of the Association for Computational Linguistics, 2021, 9:53-68.
- [61] Kitaev N, Kaiser L, Levskaya A. Reformer: The efficient transformer[J]. arXiv preprint arXiv:2001.04451, 2020.

- [62] Dao T, Fu D, Ermon S, et al. Flashattention: Fast and memory-efficient exact attention with io-awareness[J]. Advances in Neural Information Processing Systems, 2022, 35:16344-16359.
- [63] Liu Y, Ott M, Goyal N, et al. Roberta: A robustly optimized bert pretraining approach[J]. arXiv preprint arXiv:1907.11692, 2019.
- [64] Gao L, Biderman S, Black S, et al. The pile: An 800gb dataset of diverse text for language modeling [J]. arXiv preprint arXiv:2101.00027, 2020.
- [65] Baumgartner J, Zannettou S, Keegan B, et al. The pushshift reddit dataset[C]//Proceedings of the international AAAI conference on web and social media: volume 14. 2020: 830-839.
- [66] 机器学习系统：设计和实现[M]. <https://openmlsys.github.io/>, 2022.
- [67] Artetxe M, Bhosale S, Goyal N, et al. Efficient large scale language modeling with mixtures of experts[J]. arXiv preprint arXiv:2112.10684, 2021.
- [68] Shoeybi M, Patwary M, Puri R, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism[J]. arXiv preprint arXiv:1909.08053, 2019.
- [69] Huang Y. Introducing gpipe, an open source library for efficiently training large-scale neural network models[J]. Google AI Blog, March, 2019, 4.
- [70] Narayanan D, Shoeybi M, Casper J, et al. Efficient large-scale language model training on gpu clusters using megatron-lm[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2021: 1-15.
- [71] Smith S, Patwary M, Norick B, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model[J]. arXiv preprint arXiv:2201.11990, 2022.
- [72] Rasley J, Rajbhandari S, Ruwase O, et al. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters[C]//Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020: 3505-3506.
- [73] Rajbhandari S, Rasley J, Ruwase O, et al. Zero: Memory optimizations toward training trillion parameter models[C]//SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020: 1-16.
- [74] Ren J, Rajbhandari S, Aminabadi R Y, et al. Zero-offload: Democratizing billion-scale model training.[C]//USENIX Annual Technical Conference. 2021: 551-564.

- [75] Rajbhandari S, Ruwase O, Rasley J, et al. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2021: 1-14.
- [76] Al-Fares M, Loukissas A, Vahdat A. A scalable, commodity data center network architecture[J]. ACM SIGCOMM computer communication review, 2008, 38(4):63-74.
- [77] Majumder R, Wang J. DeepSpeed: Extreme-scale model training for everyone[M]. Microsoft, 2020.

索引

- n 元文法, 13
- n 元语法, 13
- n 元语法单元, 13
- 1F1B 交错式调度模式, 59
- 1F1B 非交错式调度模式, 59
- All Gather, 74
- All Reduce, 74
- All to All, 75
- Broadcast, 73
- Catastrophic Forgetting, 30
- Collective communication, CC, 73
- Cross-entropy, 88
- Data Parallelism, DP, 53
- Decentralized Network, 73
- Distributed Training, 50
- Gather, 74
- Hybrid Parallelism, HP, 53
- In-Context Learning, ICL, 2
- Language Model, LM, 1
- Large Language Models, LLM, 1
- Micro-batch, 58
- Model Parallelism Bubble, 58
- Model Parallelism, MP, 53
- Neural Language Models, NLM, 2
- Parameter Server, PS, 71
- Perplexity, 88
- Pipeline Parallelism, PP, 57
- Pre-trained Language Models, PLM, 2
- Reduce, 73
- Reduce Scatter, 75
- Scaling Laws, 3
- Scatter, 73
- Self-supervised Learning, 2
- Sparse Attention, 44
- Statistical Language Models, SLM, 2
- Tensor Parallelism, TP, 57
- 交叉熵, 88
- 分布式训练, 50
- 单向语言模型, 29
- 去中心化, 73
- 参数服务器, 71
- 困惑度, 88
- 大型语言模型, 1
- 平滑, 13
- 张量并行, 57
- 微批次, 58
- 数据并行, 53
- 模型并行, 53
- 模型并行气泡, 58
- 流水线并行, 57
- 混合并行, 53
- 灾难性遗忘, 30
- 神经语言模型, 2
- 稀疏注意力, 44
- 统计语言模型, 2
- 缩放法则, 3
- 自监督学习, 2
- 语境学习, 2
- 语言模型, 1
- 集合通信, 73
- 预训练语言模型, 2