大语言模型理论与实践

张奇 桂韬 郑锐 黄萱菁

2023 年 6 月 18 日

数学符号

数与数组

α	标量
$\boldsymbol{\alpha}$	向量
$\boldsymbol{A}$	矩阵
$\mathbf{A}$	张量
$\boldsymbol{I}_n$	n 行 n 列单位矩阵
$\boldsymbol{v}_w$	单词 w 的分布式向量表示
$\boldsymbol{e}_w$	单词 w 的独热向量表示：[0,0,...,1,0,...0]，w 下标处元素为 1

索引

α_i	向量 $\boldsymbol{\alpha}$ 中索引 i 处的元素
α_{-i}	向量 $\boldsymbol{\alpha}$ 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 $\boldsymbol{A}$ 中第 i 行、第 j 列处的元素
$\boldsymbol{A}_{i:}$	矩阵 $\boldsymbol{A}$ 中第 i 行
$\boldsymbol{A}_{:j}$	矩阵 $\boldsymbol{A}$ 中第 j 列
$\boldsymbol{A}_{ijk}$	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
$\{0,1,...,n\}$	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

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

微积分

$\frac{\mathrm{d}y}{\mathrm{d}x}$	y 对 x 的导数
$\frac{\partial y}{\partial x}$	y 对 x 的偏导数
$\nabla_{\boldsymbol{x}} y$	y 对向量 $\boldsymbol{x}$ 的梯度
$\nabla_{\boldsymbol{X}} y$	y 对矩阵 $\boldsymbol{X}$ 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
$\mathrm{Var}(f(x))$	$f(x)$ 在分布 $P(x)$ 下的方差
$\mathrm{Cov}(f(x), g(x))$	$f(x)$ 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
$\boldsymbol{x}^{(i)}$	数据集中第 i 个样本（输入）
$\boldsymbol{y}^{(i)}$ 或 $y^{(i)}$	第 i 个样本 $\boldsymbol{x}^{(i)}$ 的标签（输出）

函数

$f : \mathcal{A} \longrightarrow \mathcal{B}$	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
$f(\boldsymbol{x}; \boldsymbol{\theta})$	由参数 $\boldsymbol{\theta}$ 定义的关于 $\boldsymbol{x}$ 的函数（也可以直接写作 $f(\boldsymbol{x})$，省略 $\boldsymbol{\theta}$）
$\log x$	x 的自然对数函数
$\sigma(x)$	Sigmoid 函数 $\dfrac{1}{1+exp(-x)}$
$\|\|\boldsymbol{x}\|\|_p$	$\boldsymbol{x}$ 的 L^p 范数
$\|\|\boldsymbol{x}\|\|$	$\boldsymbol{x}$ 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 $x = x_1, x_2, ..., x_n$ 中第 i 个单词 x_i 的词向量 $\boldsymbol{v}_{x_i}$
- 损失函数 $\mathcal{L}$ 为负对数似然函数：$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{(x,y)} \log P(y|x_1...x_n)$
- 算法的空间复杂度为 $\mathcal{O}(mn)$

目 录

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 $\mathbb{V}$ 上的语言模型，由函数 $P(w_1w_2...w_m)$ 表示，可以形式化的为构建词序列 $w_1w_2...w_m$ 的概率分布，表示词序列 $w_1w_2...w_m$ 作为一个句子出现的可能性大小。由于联合概率 $P(w_1w_2...w_m)$ 的参数量十分巨大，直接计算 $P(w_1w_2...w_m)$ 非常困难。

为了减少 $P(w_1w_2...w_m)$ 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：

$$\begin{aligned} P(w_1w_2...w_m) &= P(w_1)P(w_2|w_1)P(w_3|w_1w_2)\cdots P(w_m|w_1w_2...w_{m-1}) \\ &= \prod_{i=1}^{m} P(w_i|w_1w_2\cdots w_{i-1}) \end{aligned} \tag{1.1}$$

由此，$w_1w_2...w_m$ 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，

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：

$$\begin{aligned} P(\text{把 努力 变成 一种 习惯}) =& P(\text{把}) \times P(\text{努力}|\text{把}) \times P(\text{变成}|\text{把 努力}) \times \\ & P(\text{一种}|\text{把 努力 变成}) \times P(\text{习惯}|\text{把 努力 变成 一种}) \end{aligned} \tag{1.2}$$

通过上述过程将联合概率 $P(w_1 w_2 ... w_m)$ 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，$P(w_m|w_1 w_2 ... w_{m-1})$ 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 $P(w_i|w_{i-n+1} ... w_{i-1})$ 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，研究人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。

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 [15] 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1.1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。

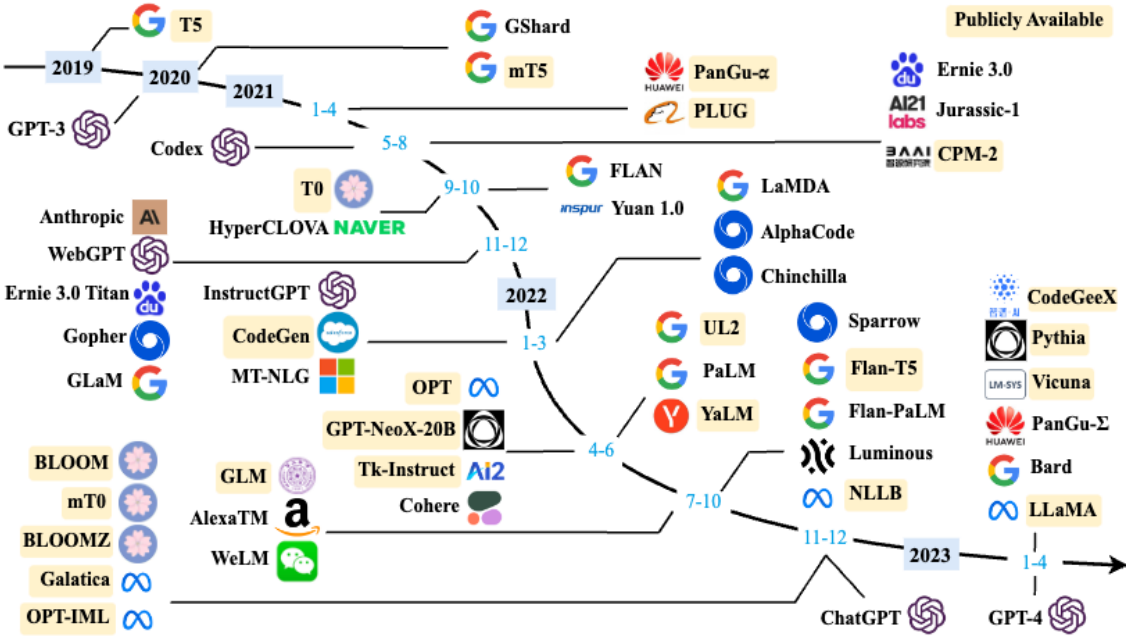

图 1.1　大语言模型时间线 [15]

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有数量级上提升。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 亿的 T5[16] 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)[17]、百度 ERNIE(Baidu)[18]、华为盘古-α[19] 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）[20] 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[21]。Nakano 等人则探索了结合搜索引擎的问题回答算法 WebGPT[22]。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。当前仍然处于大语言模型的高速发展时期，并且由于涉及到大量的商业利益，OpenAI 并没有公开 ChatGPT 和 GPT-4

的实现细节，包括数据选择、模型结构、基础模型训练、类人对齐等各个模块的实现方式都在探索和研究中，很多部分并没有统一的结论。在本书中，我们将尽可能的将多方的观点进行介绍，但是需要读者自行进行判断。书中很多观点也仅基于我们当前对大语言模型的粗浅认识，恳请读者辩证看待并批评指正。

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，大语言模型构建的流程如图1.2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。

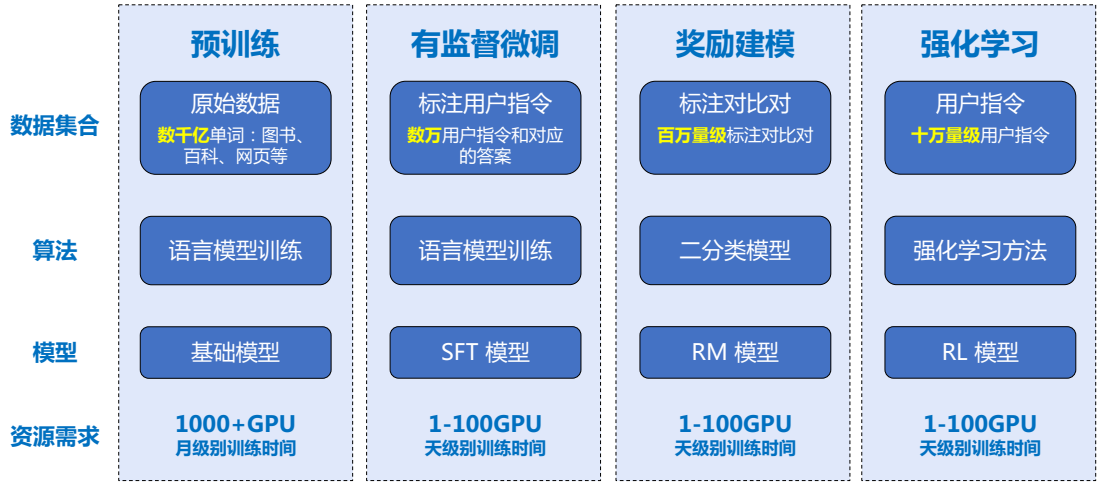

图 1.2　大语言模型构建流程

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）。根据文献 [23] 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 [24] 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[25] 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 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，在基础语言模型纸上再进行训练，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[26]、Vicuna[27]、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[28]，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1.3给出了 InstructGPT 系统中奖励模型训练样本标注示例[21]。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。

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

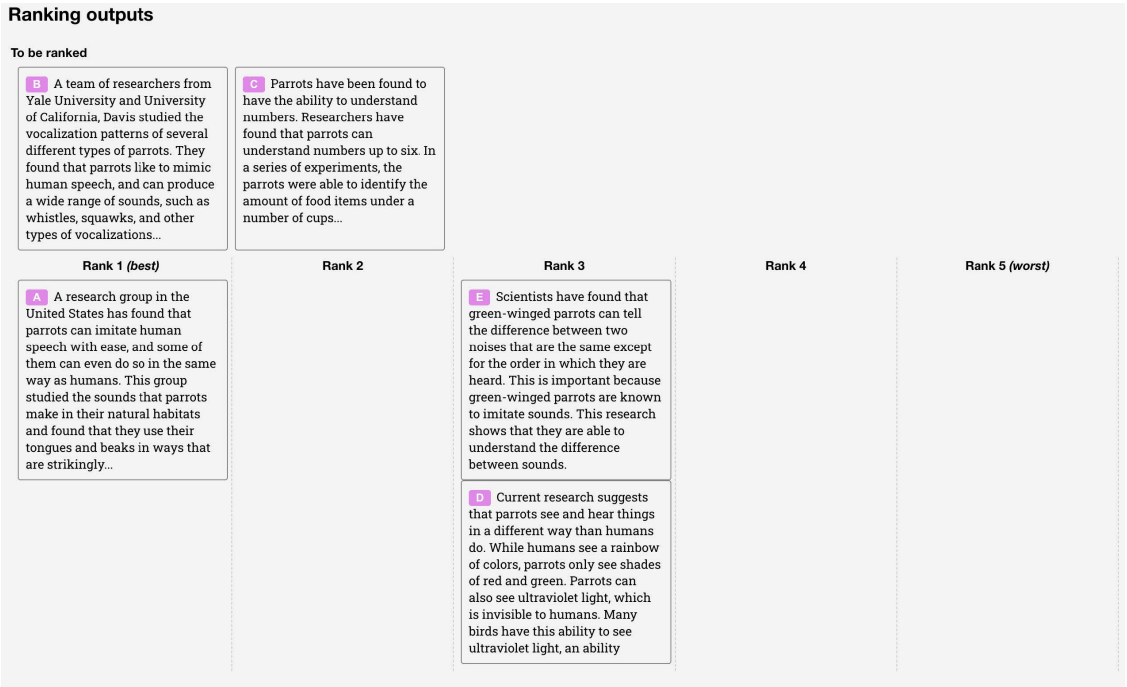

图 1.3　InstructGPT 系统中奖励模型训练样本标注示例[21]

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 [21] 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多的效果。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1.4所示。

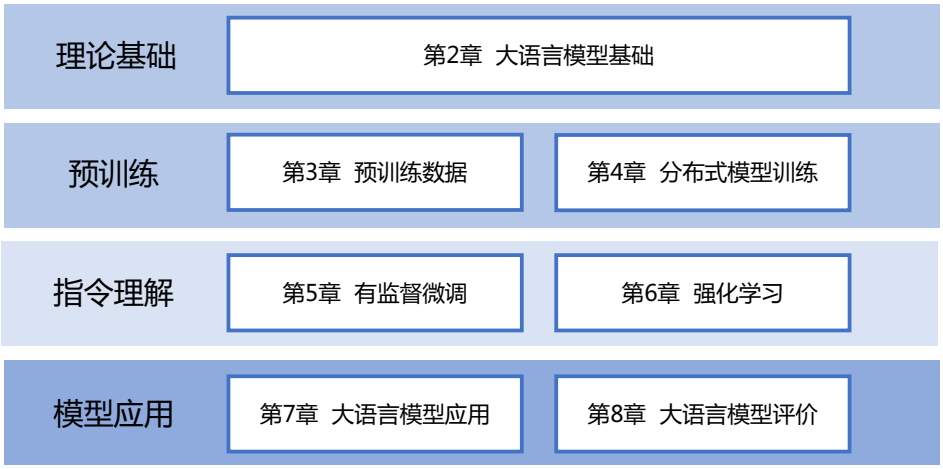

图 1.4 本书章节安排

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。主要介绍包括 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务，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 $w_1w_2...w_m$ 的概率分布 $P(w_1w_2...w_m)$，即计算给定的词序列 $w_1w_2...w_m$ 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 $P(w_1w_2...w_m)$ 表示，对于任意词串 $w_1w_2...w_m \in \mathbb{V}^+$，则有 $P(w_1w_2...w_m) \geqslant 0$，并且对于所有词串，函数 $P(w_1w_2...w_m)$ 满足归一化条件 $\sum_{w_1w_2...w_m \in \mathbb{V}^+} P(w_1w_2...w_m) = 1$。$P(w_1w_2...w_m)$ 是定义在 $\mathbb{V}^+$ 上的概率分布。

由于联合概率 $P(w_1w_2...w_m)$ 的参数量十分巨大，直接计算 $P(w_1w_2...w_m)$ 非常困难。如果把 $w_1w_2...w_m$ 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 $P(w_1w_2...w_m)$ 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 $w_1w_2...w_n$ 组成的句子 S，使用链式法分解则

得到:

$$P(S) = \prod_{i=1}^{n} P(w_i|w_1w_2...w_{i-1}) \tag{2.1}$$

其中，词 w_i 出现的概率受它前面的 $i-1$ 个词 $w_1w_2...w_{i-1}$ 影响，我们将这 $i-1$ 个词 $w_1w_2...w_{i-1}$ 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 $w_1w_2...w_{i-1}$。最简单的根据语料库对 $P(w_i|w_1w_2...w_{i-1})$ 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。

$$P(w_i|w_1w_2...w_{i-1}) = \frac{\mathrm{C}(w_1w_2...w_{i-1}w_i)}{\mathrm{C}(w_1w_2...w_{i-1})} \tag{2.2}$$

其中，$\mathrm{C}(\cdot)$ 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 $P(w_i|w_1w_2...w_{i-1})$ 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:

$$\begin{aligned} P(w_i|w_1w_2...w_{i-1}) &= P(w_i|w_{i-(n-1)}w_{i-(n-2)}...w_{i-1}) \\ P(w_i|w_1^{i-1}) &= P(w_i|w_{i-n+1}^{i-1}) \end{aligned} \tag{2.3}$$

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），又被称作一阶马尔可夫链。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），又被称作二阶马尔可夫链。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:

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

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[29]。

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 $P(w_i|w_{i-n+1}...w_{i-1})$ 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[29]。

2.2 Transformer 模型

Transformer 模型[30] 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 $\{\boldsymbol{x}_i\}_{i=1}^t$ 作为输入，并输出一个等长的向量序列作为输出 $\{\boldsymbol{y}_i\}_{i=1}^t$。这里的 $\boldsymbol{x}_i$ 和 $\boldsymbol{y}_i$ 分别对应着文本序列中的一个单词的表示。而 $\boldsymbol{y}_i$ 是当前 Transformer 块对输入 $\boldsymbol{x}_i$ 进一步整合其上下文语义后对应的输出。在从输入 $\{\boldsymbol{x}_i\}_{i=1}^t$ 到输出 $\{\boldsymbol{y}_i\}_{i=1}^t$ 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。
- **层标准化**：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，对表示序列进行层标准化操作，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

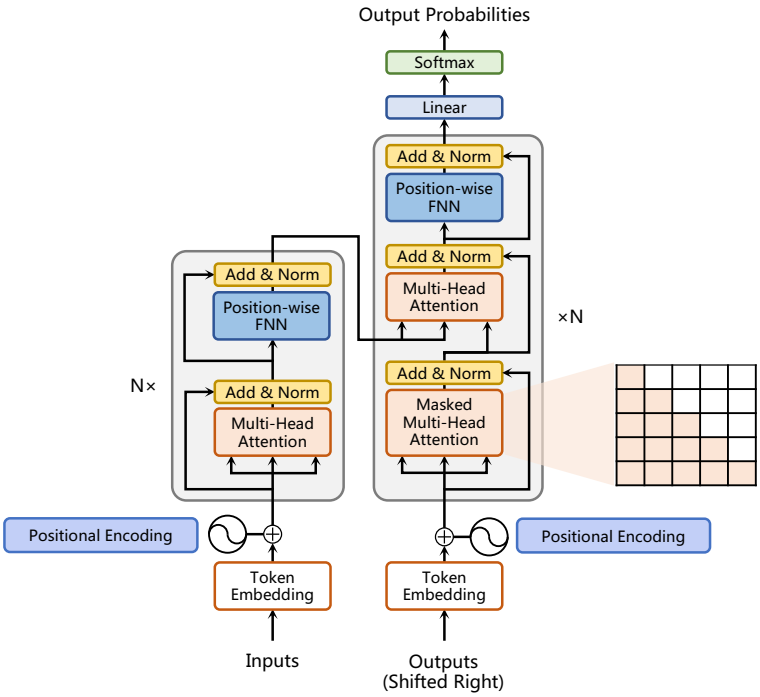

图 2.1 基于 Transformer 的编码器和解码器结构[30]

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：

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

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

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 $\mathrm{pos}+k$ 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 $\{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^t$，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 $\boldsymbol{q}_i$（Query），键 $\boldsymbol{k}_i$（Key），值 $\boldsymbol{v}_i$（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关注程度。具体来说，如图2.2所示，通过三个线性变换 $\boldsymbol{W}^{\boldsymbol{Q}} \in \mathbb{R}^{d\times d_k}$，$\boldsymbol{W}^{\boldsymbol{K}} \in \mathbb{R}^{d\times d_k}$，$\boldsymbol{W}^{\boldsymbol{V}} \in \mathbb{R}^{d\times d_v}$ 将输入序列中的每一个单词表示 $\boldsymbol{x}_i$ 转换为其对应的 $\boldsymbol{q}_i \in \mathbb{R}^{d_k}$，$\boldsymbol{k}_i \in \mathbb{R}^{d_k}$，$\boldsymbol{v}_i \in \mathbb{R}^{d_v}$ 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 $\boldsymbol{q}_i \cdot \boldsymbol{k}_1, \boldsymbol{q}_i \cdot \boldsymbol{k}_2, ..., \boldsymbol{q}_i \cdot \boldsymbol{k}_t$。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：

$$\boldsymbol{Z} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d}})\boldsymbol{V} \tag{2.7}$$

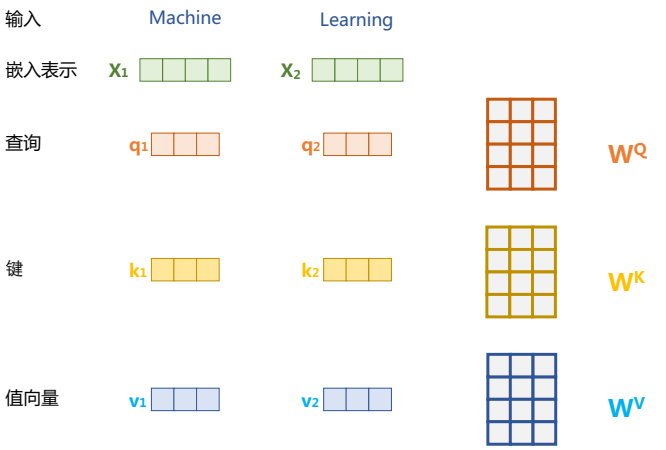

图 2.2 自注意力机制中的查询、键、值向量

其中 $\boldsymbol{Q} \in \mathbb{R}^{L\times d_k}, \boldsymbol{K} \in \mathbb{R}^{L\times d_k}, \boldsymbol{V} \in \mathbb{R}^{d\times d_v}$ 分别表示输入序列中的不同单词的 $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}$ 向量拼接组成的矩阵，L 表示序列长度，$\boldsymbol{Z} \in \mathbb{R}^{L\times d_v}$ 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 $\{\boldsymbol{W}_j^{bmQ}\boldsymbol{W}_j^{\boldsymbol{K}}\boldsymbol{W}_j^{\boldsymbol{V}}\}_{j=1}^{N}$ 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 $\{\boldsymbol{Z}_j\}_{j=1}^{N}$。最终，线性变换 $\boldsymbol{W}^{\boldsymbol{O}} \in \mathbb{R}^{(Nd_v)\times d}$ 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 $\{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^{t}$。

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。

$$\mathrm{FFN}(\boldsymbol{x}) = \mathrm{Relu}(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{2.8}$$

其中 $\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2$ 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 残差连接与层标准化

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标准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：

$$\boldsymbol{x}^{l+1} = f(\boldsymbol{x}^l) + \boldsymbol{x}^l \tag{2.9}$$

其中 $\boldsymbol{x}^l$ 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，层标准化技术被进一步引入每个 Transformer 块的当中：

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

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。层标准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 实现的层标准化参考代码如下：

```
class NormLayer(nn.Module):

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

        self.size = d_model

        # 层标准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 $t1$ 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)
```

```
        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[1] 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结构的 BERT 模型、仅由解码器组成的 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）。

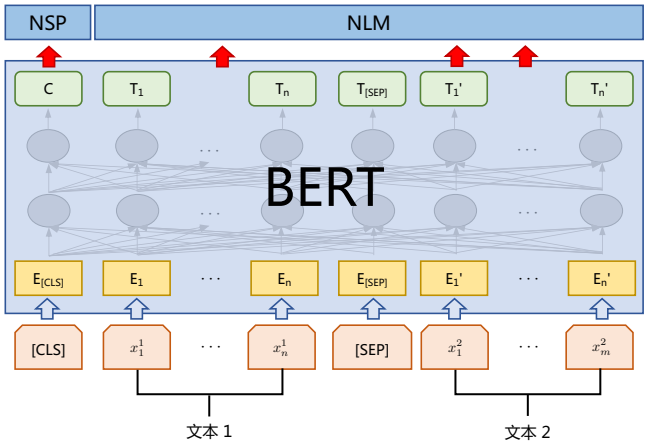

图 2.3 掩码预训练语言模型 BERT 神经网络结构[1]

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 $\boldsymbol{v}$ 可以表示为:

$$\boldsymbol{v} = \boldsymbol{v}^t + \boldsymbol{v}^s + \boldsymbol{v}^p$$

其中，$\boldsymbol{v}^t$ 代表词嵌入；$\boldsymbol{v}^s$ 代表段嵌入；$\boldsymbol{v}^p$ 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 $\boldsymbol{e}^t \in \mathbb{R}^{N\times|\mathbb{V}|}$，其对应的词嵌入 $\boldsymbol{v}_t$ 为：

$$\boldsymbol{v}^t = \boldsymbol{e}^t\boldsymbol{W}^t \tag{2.11}$$

其中，$\boldsymbol{W}^t \in \mathbb{R}^{|\mathbb{V}|\times e}$ 表示词嵌入矩阵；$|\mathbb{V}|$ 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 $\boldsymbol{W}^s$ 将独热段编码 $\boldsymbol{e}^s$ 转化为段嵌入 $\boldsymbol{v}^s$:

$$\boldsymbol{v}^s = \boldsymbol{e}^s\boldsymbol{W}^s \tag{2.12}$$

其中，$\boldsymbol{W}^s \in \mathbb{R}^{|\mathbb{S}|\times e}$ 表示段嵌入矩阵；$|\mathbb{S}|$ 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 $\boldsymbol{e}^p$，通过可训练的位置嵌入矩阵 $\boldsymbol{W}^p$ 即可得到位置向量 $\boldsymbol{v}^p$:

$$\boldsymbol{v}^p = \boldsymbol{e}^p\boldsymbol{W}^p \tag{2.13}$$

其中，$\boldsymbol{W}^t \in \mathbb{R}^{N\times e}$ 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，$\text{BERT}_{\text{BASE}}$ 使用 $L = 12$，$H = 768$，$A = 12$，总参数量为 110M，$\text{BERT}_{\text{LARGE}}$ 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 $x_1x_2...x_n$ 表示原始文本，在经过上述掩码替换后得到输入为 $x'_1x'_2...x'_n$。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 $\boldsymbol{v}$：

$$X = [\text{CLS}]x'_1x'_2...x'_n[\text{SEP}] \tag{2.14}$$

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

在编码层，对于输入表示 $\boldsymbol{v}$ 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：

$$\boldsymbol{h}^{(l)} = \text{Transformer-Block}(\boldsymbol{h}^{(l-1)})l \in 1, 2, ..., L \tag{2.16}$$

其中 $\boldsymbol{h}^{(l)} \in \mathbb{R}^{N\times d}$ 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，$\boldsymbol{h}^{(0)} = \boldsymbol{v}$ 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：

$$\boldsymbol{h} = \text{Transformer}(\boldsymbol{v}) \tag{2.17}$$

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

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 $\mathbb{M} = \{m_1, m_2, ...m_k\}$，$k$ 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 $\boldsymbol{h}$ 中抽取对应的表示 $\boldsymbol{h}_{m_i}$。在此基础上，利用公式2.11中所给出的词向量矩阵 $\boldsymbol{W}^t \in \mathbb{R}^{\mathbb{V}\times e}$ 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：

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

其中 $\boldsymbol{b}^0 \in \mathbb{R}^{\mathbb{V}}$ 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 $x^{(1)} = x_1^{(1)}x_2^{(1)}...x_n^{(1)}$ 和 $x^{(2)} = x_1^{(2)}x_2^{(2)}...x_m^{(2)}$，经过如下处理得到 BERT 的输入表示 $\boldsymbol{v}$：

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

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

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：

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

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 $\boldsymbol{h}_{[\text{CLS}]}$ 进行分类预测。使用全连接层预测输入文本的分类概率 $P \in \mathbb{R}^2$：

$$P = \text{Softmax}(\boldsymbol{h}_{[\text{CLS}]}\boldsymbol{W}^p + \boldsymbol{b}^o) \tag{2.22}$$

其中，$\boldsymbol{W}^p \in \mathbb{R}^{d\times 2}$ 为全连接层权重；$\boldsymbol{b}^o$ 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-2 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-2 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 $w = w_1w_2...w_n$，GPT-2 首先在输入层中将其映射为稠密的向量：

$$\boldsymbol{v}_i = \boldsymbol{v}_i^t + \boldsymbol{v}_i^p \tag{2.23}$$

① Transformer 解码器的具体结构请参考第 8 章**??**节。

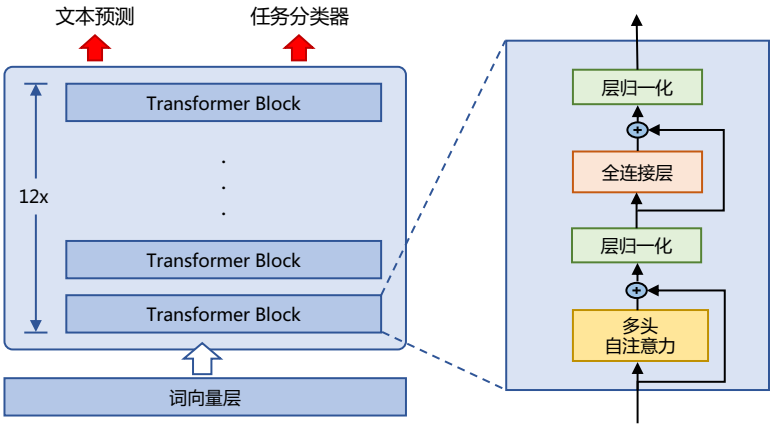

图 2.4 GPT-2 预训练语言模型结构

其中，$\boldsymbol{v}_i^t$ 是词 w_i 的词向量，$\boldsymbol{v}_i^p$ 是词 w_i 的位置向量，$\boldsymbol{v}_i$ 为第 i 个位置的单词经过模型输入层（第 0 层）后的输出。GPT-2 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 $\boldsymbol{v} = \boldsymbol{v}_1 ... \boldsymbol{v}_n$，随后将 $\boldsymbol{v}$ 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-2 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：

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

其中 $\boldsymbol{h}^{(L)} \in \mathbb{R}^{d \times n}$ 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-2 模型的输出层基于最后一层的表示 $\boldsymbol{h}^{(L)}$，预测每个位置上的条件概率，其计算过程可以表示为：

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

其中，$\boldsymbol{W}^e \in \mathbb{R}^{|\mathbb{V}| \times d}$ 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：

$$\mathcal{L}^{\mathrm{PT}}(w) = -\sum_{i=1}^{n} \log P(w_i|w_0...w_{i-1};\boldsymbol{\theta}) \tag{2.26}$$

其中 $\boldsymbol{\theta}$ 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 $x = x_1x_2...x_n$ 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 $\boldsymbol{h}_n^{(L)}$，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。

$$P(y|x_1...x_n) = \mathrm{Softmax}(\boldsymbol{h}^{(L)}\boldsymbol{W}^y) \tag{2.27}$$

其中 $\boldsymbol{W}^y \in \mathbb{R}^{d\times k}$ 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：

$$\mathcal{L}^{\mathrm{FT}}(\mathbb{D}) = \sum_{(x,y)} \log P(y|x_1...x_n) \tag{2.28}$$

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：

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

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）[31]。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。

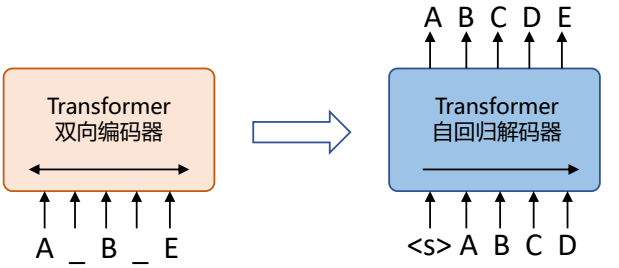

图 2.5 BART 的神经网络结构[31]

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。该噪声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。该噪声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。

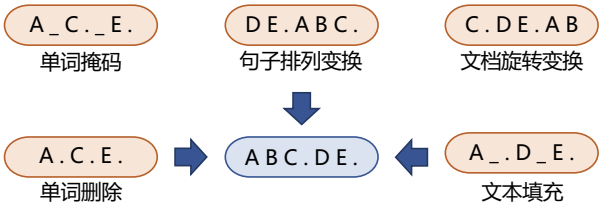

图 2.6　BART 各类型加噪方式示例

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，除了上述噪声之外，其他任意形式的文本噪声也是适用的。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 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 [31]。

2.4 大语言模型结构

自 2020 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] 以来，包含 Google、Meta、百度、智源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**大规模语言模型**（Large Language Model，LLM），

也称**大模型**。大模型在文本生成、少样本学习、零样本学习、推理任务等方面取得了非常大的进展。表2.1给出了截止 2023 年 1 月典型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。图**??**给出了 Tansformer 大模型的发布时间线和种类，不同颜色的框表示不同的模型种类。

表 2.1 典型大规模语言模型汇总

模型名称	参数量	训练单词数	研发机构	发布时间
ChatGPT	1750 亿	3000 亿	OpenAI	2022 年 11 月
Galactica	1200 亿	4500 亿	Meta AI	2022 年 11 月
BLOOMZ	1760 亿	3660 亿	BigScience	2022 年 11 月
U-PaLM	5400 亿	7800 亿	Google Research	2022 年 10 月
CodeGeeX	130 亿	8500 亿	清华大学	2022 年 9 月
PaLM	5400 亿	7800 亿	Google Research	2022 年 4 月
ERNIE 3.0 Titan	2600 亿	–	Baidu	2021 年 12 月
FLAN	1370 亿	–	Google	2021 年 9 月
GPT-3	1750 亿	3000 亿	OpenAI	2020 年 5 月
T5	110 亿	340 亿	Google	2019 年 10 月
RoBERTa	3.55 亿	22000 亿	Meta AI	2019 年 7 月
GPT-2	15 亿	100 亿	OpenAI	2019 年 2 月
BERT	3 亿	1370 亿	Google	2018 年 10 月
GPT-1	1 亿	–	OpenAI	2018 年 6 月

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并不容易，因此文献 [24] 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）。

在模型架构方面不论是 GPT-3 还是 OPT 所采用的模型结构都与我们在本章第2.3.2节所介绍的 GPT-2 模型一样，都采用由多层 Transformer 组成的单向语言模型，采用自回归方式从左到右对文本序列建模。但是针对不同的规模的参数量要求，其所使用的层数、自注意力头数、嵌入表示维度大小等具体参数各不相同。OPT 给出了 9 种模型参数的细节，如表2.2所示。采用 AdamW 优化器进行优化，其参数 (β_1, β_2) 设置为 $(0.9, 0.95)$。其他参数细节可以参考文献 [24]。

表 2.2 OPT 不同模型规模下的具体参数细节[24]

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
125M	12	12	768	6.0e-4	50 万
350M	24	16	1024	3.0e-4	50 万
1.3B	24	32	2048	2.0e-4	100 万
2.7B	32	32	2560	1.6e-4	100 万
6.7B	32	32	4096	1.2e-4	200 万
13B	40	40	5120	1.0e-4	400 万
30B	48	56	7168	1.0e-4	400 万
66B	64	72	9216	0.8e-4	200 万
175B	96	96	12288	1.2e-4	200 万

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[16]、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 亿子词。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 亿子词训练时，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[32]、Pile[33] 和 PushShift.io Redit[34] 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. 分布式模型训练

4.1 分布式计算基础

4.2 分布式模型训练

由于模型参数量和所使用的数据量都非常巨大，普通的服务器单机无法完成训练过程，因此通常采用分布式架构完成训练。GPT-3 和 OPT 中没有对这个部分给出详细的描述。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 [24] 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）[35] 以及 Megatron-LM 张量并行（Tensor Parallelism）[36]，整体训练时间将近 2 个月。BLOOM[25] 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。

BLOOM 使用 Megatron-DeepSpeed[37] 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[38] 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4.1所示。数据并行（Data Parallelism）将模型构建多个副本，每个副本放置在不同的设备上，并分别针对一部分数据并行进行训练，在每个训练步结束时同步副本间数据。张量并行（Tensor Parallelism）将模型的单个层划分到不同设备中，这样可以避免将所有激活张量或梯度张量都放置在一个 GPU 上，这种方法也称为水平并行或层内模型并行。流水线并行（Pipeline Parallelism）将模型不同层放置在多个 GPU 中，每个 GPU 中仅包含部分的层，这种方法也称为垂直并行。ZeRO（Zero Redundancy Optimizer）优化器[39] 允许不同的进程只保存一小部分数据（训练步骤所需的参数、梯度和优化器状态）。通过上述四个步骤可以实现数百个 GPU 的高效并行计算。

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

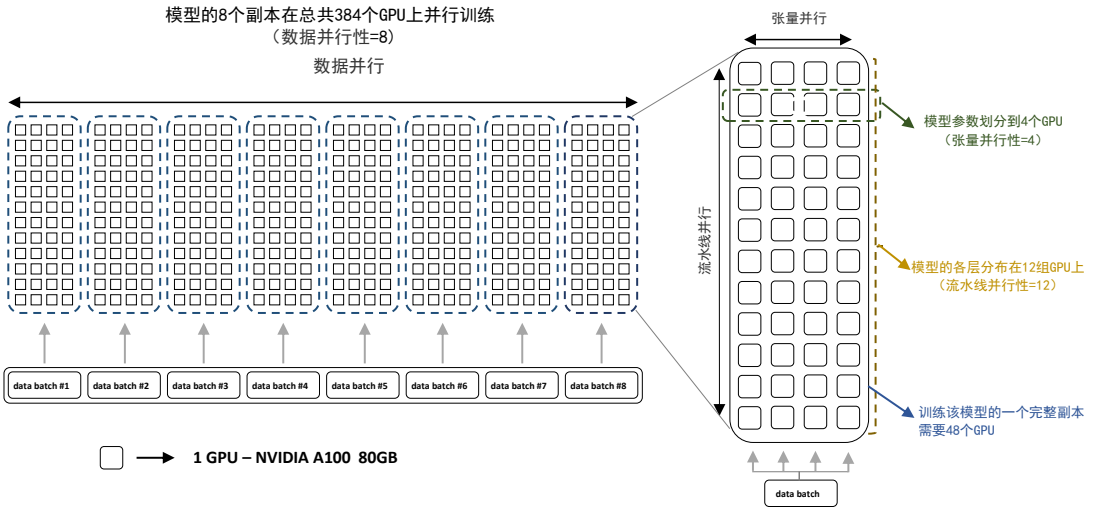

图 4.1 BLOOM 并行结构[25]

事实性知识（Factual Knowledge）和常识知识（Commonsense）在内的世界知识（World Knowledge）。

4.3 Deepspeed-Chat PreTrain 实践

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兴趋势，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-cotnext 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 有监督微调

instructgpt 3.4 Human data collection SFT LORA

5.5 Deepspeed-Chat SFT 实践

6. 强化学习

7. 大语言模型应用

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：

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

对于由句子 $(s_1, s_2..., s_n)$ 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：

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

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：

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

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：

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

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

[16] 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.

[17] 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.

[18] Sun Y, Wang S, Li Y, et al. Ernie: Enhanced representation through knowledge integration[J]. arXiv preprint arXiv:1904.09223, 2019.

[19] 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.

[20] Chung H W, Hou L, Longpre S, et al. Scaling instruction-finetuned language models[J]. arXiv preprint arXiv:2210.11416, 2022.

[21] Ouyang L, Wu J, Jiang X, et al. Training language models to follow instructions with human feedback [J]. arXiv preprint arXiv:2203.02155, 2022.

[22] Nakano R, Hilton J, Balaji S, et al. Webgpt: Browser-assisted question-answering with human feedback[J]. arXiv preprint arXiv:2112.09332, 2021.

[23] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners[J]. arXiv preprint arXiv:2005.14165, 2020.

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

[25] 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.

[26] 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.

[27] 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.

[28] Zhou C, Liu P, Xu P, et al. Lima: Less is more for alignment[J]. arXiv preprint arXiv:2305.11206, 2023.

[29] 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.

[30] 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/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[31] 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.

[32] Liu Y, Ott M, Goyal N, et al. Roberta: A robustly optimized bert pretraining approach[J]. arXiv preprint arXiv:1907.11692, 2019.

[33] 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.

[34] 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.

[35] Artetxe M, Bhosale S, Goyal N, et al. Efficient large scale language modeling with mixtures of experts[J]. arXiv preprint arXiv:2112.10684, 2021.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

索引