1. 绪论

大语言模型是一种由包含数百亿个及以上参数的深度神经网络构建的语言模型，通常使用自监督学习方法通过大量无标注文本进行训练。2018 年以来，Google、OpenAI、Meta、百度、华为等公司和研究机构相继发布了 BERT[1]、GPT[2] 等多种模型，这些模型在几乎所有自然语言处理任务中都表现出色。2019 年，大语言模型呈现爆发式的增长，特别是 2022 年 11 月 ChatGPT（Chat Generative Pre-trained Transformer）的发布，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，实现问答、分类、摘要、翻译、聊天等从理解到生成的各种任务。大语言模型展现出了强大的对世界知识的掌握和对语言的理解能力。

本章主要介绍大语言模型的基本概念、发展历程和构建流程。

1.1 大语言模型的基本概念

使用语言是人类与其他动物最重要的区别之一，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个网页的资源，其中大部分信息都是用自然语言描述的。因此，如果人工智能算法想要获取知识，就必须懂得如何理解人类所使用的不太精确、可能有歧义甚至有些混乱的语言。**语言模型**（Language Model，LM）的目标就是对自然语言的概率分布建模。词汇表 $\mathbb{V}$ 上的语言模型，由函数 $P(w_1 w_2 \cdots w_m)$ 表示，可以形式化地构建为词序列 $w_1 w_2 \cdots w_m$ 的概率分布，表示词序列 $w_1 w_2 \cdots w_m$ 作为一个句子出现的可能性的大小。由于联合概率 $P(w_1 w_2 \cdots w_m)$ 的参数量巨大，因此直接计算 $P(w_1 w_2 \cdots w_m)$ 非常困难[3]。《现代汉语词典》（第 7 版）包含约 7 万词，句子长度按照 20 个词计算，语言模型的参数量达到 7.9792×10^{96} 的天文数字。在中文的书面语中，超过 100 个词的句子并不罕见，如果要将所有可能性都纳入考虑，则语言模型的复杂度会进一步增加，以目前的计算手段无法进行存储和运算。

为了减小 $P(w_1 w_2 \cdots w_m)$ 模型的参数空间，可以利用句子序列（通常是从左至右）的生成过程将其进行分解，使用链式法则可以得到

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

由此，$w_1 w_2 \cdots w_m$ 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，然后根据 w_1 和 w_2 生成 w_3，依此类推，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 $P(w_1 w_2 \cdots w_m)$ 转换为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减小，$P(w_m|w_1 w_2 \cdots w_{m-1})$ 的参数空间依然是天文数字。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

$$\begin{aligned} P(w_i|w_1 w_2 \cdots w_{i-1}) &= P(w_i|w_{i-(n-1)} w_{i-(n-2)} \cdots w_{i-1}) \\ P(w_i|w_1^{i-1}) &= P(w_i|w_{i-n+1}^{i-1}) \end{aligned} \tag{1.3}$$

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**。

虽然 n 元语言模型能缓解句子概率为零的问题，但语言是由人和时代创造的，具备无尽的可能性，再庞大的训练数据也无法覆盖所有的 n-gram，而训练数据中的零频率并不代表零概率。因此，需要使用平滑技术（Smoothing）解决，为所有可能出现的字符串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事件，降低高概率事件，使整体的概率分布趋于均匀。这类方法通常被称为**统计语言模型**（Statistical Language Models，SLM）。相关平滑算法细节可以参考《自然语言处理导论》的第 6 章[4]。

n 元语言模型从整体上看与训练数据规模和模型的阶数（考虑上下文的数量）有较大的关系，不同的平滑算法在不同情况下的表现有较大的差距。虽然平滑算法较好地解决了零概率问题，但是基于稀疏表示的 n 元语言模型仍然有以下三个较为明显的缺点。

（1）无法对长度超过 n 的上下文建模。

（2）依赖人工设计规则的平滑技术。

（3）当 n 增大时，数据的稀疏性随之增大，模型的参数量更是呈指数级增加，受数据稀疏问题的影响，其参数难以被准确学习。

此外，n 元文法中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 $P(w_i|w_{i-n+1}\cdots w_{i-}$ 进行估计的语言模型[5]。词的独热编码被映射为一个低维稠密的实数向量，称为**词向量**（Word Embedding）。此后，循环神经网络[6]、卷积神经网络[7]、端到端记忆网络[8] 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，有些模型还可以摆脱对历史文本长度的限制，从而更好地对长距离依赖关系建模。这类方法通常被称为**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，使用标注数据进行训练，因此，语言模型的训练过程也不可避免地需要构造训练数据。由于训练目标可以通过无标注文本直接获得，因此模型的训练仅需要大规模无标注文本。语言模型也成了典型的**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可能。

受计算机视觉领域采用 ImageNet[9] 对模型进行一次预训练，使模型可以通过海量图像充分学习如何提取特征，再根据任务目标进行模型精调的预训练范式影响，自然语言处理领域基于预训练语言模型的方法逐渐成为主流。以 ELMo[10] 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[11] 和 BERT[1] 为代表的基于 Transformer 结构[12] 的大规模预训练语言模型的出现，使自然语言处理全面进入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，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。这类方法通常被称为**预训练语言模型**（Pre-trained Language Models，PLM）。

2020 年，OpenAI 发布了由包含 1750 亿个参数的神经网络构成的生成式大规模预训练语言模型 GPT-3（Generative Pre-trained Transformer 3）[13]，开启了大语言模型的新时代。由于大语言模型的参数量巨大，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。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中取得很好的效果。此后，研究人员提出了面向大语言模型的提示词（Prompt）学习方法，以及模型即服务范式（Model as a Service，MaaS）、**指令微调**（Instruction Tuning）等方法，在不同任务中都取得了很好的效果。与此同时，Google、Meta、BigScience、百度、华为等公司和研究机构纷纷发布了 PaLM[14]、LaMDA[15]、T0[16] 等不同大语言模型。2022 年年底 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 [17] 中提出了**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平稳提升。如图 1.1 所示，模型的损失（Loss）值随着模型规模的指数增加而线性降低。这意味着模型的能力可以根据这三个变量估计，增加模型参数量，扩大数据集规模都可以使模型的性能可预测地提升。这为继续扩大大语言模型的规模给出了定量分析依据。

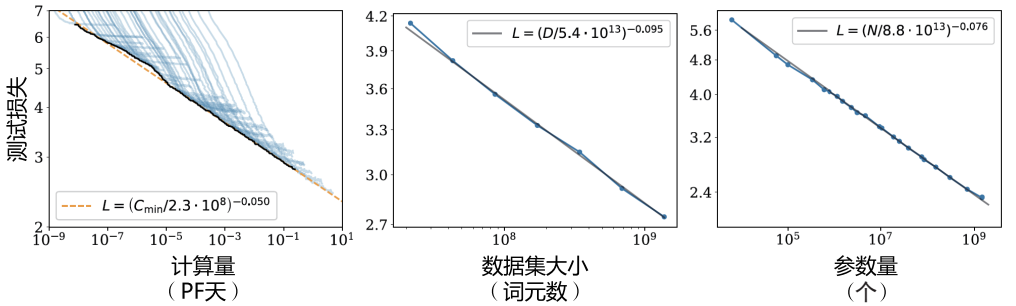

图 1.1 大语言模型的缩放法则[17]

1.2 大语言模型的发展历程

大语言模型的发展历程虽然只有不到 5 年，但是发展速度相当惊人，截至 2025 年 2 月，国内外有超过百种大语言模型相继发布。特别是 2024 年 12 月 DeepSeek V3 和 2025 年 1 月 DeepSeek R1 模型的开源，不仅在训练效率和思考推理上取得了突破，还赢得了国际社会对中国人工智能技术的高度认可。中国人民大学赵鑫教授团队在《大语言模型》书中按照时间线给出了 2019 年至 2024 年 6 月比较有影响力并且模型参数量超过 100 亿个的大语言模型，我们在此基础上扩展到 2025 年 2 月，如图1.2 所示。大语言模型的发展可以粗略地分为如下三个阶段：基础模型阶段、能力探索阶段和突破发展阶段。

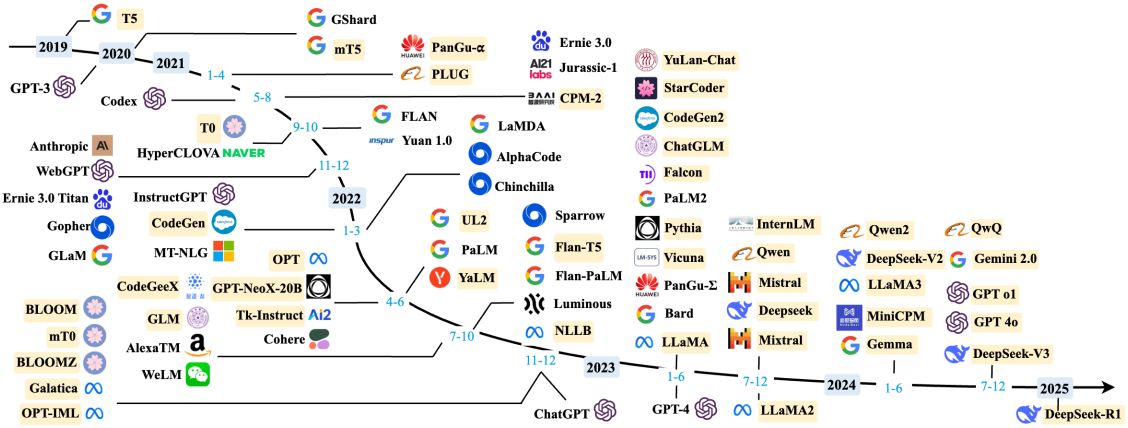

图 1.2 大语言模型发展时间线[18]

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[12] 架构，在机器翻译任务上取得了突破性进展。2018 年，Google 和 OpenAI 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 版本的参数量为 1.1 亿个，BERT-Large 版本的参数量为 3.4 亿个，GPT-1 的参数量为 1.17 亿个。这在当时，比其他深度神经网络的参数量，已经有了数量级上的提升。2019 年 OpenAI 发布了 GPT-2[11]，其参数量达到 15 亿个。此后，Google 也发布了参数规模为 110 亿个的 T5[19] 模型。2020 年，OpenAI 进一步将语言模型的参数量扩展到 1750 亿个，发布了 GPT-3[13]。此后，国内也相继推出了一系列的大语言模型，包括清华大学的 ERNIE[20]、百度的 ERNIE[21]、华为的 PanGU-α[22] 等。此阶段的研究主要集中在语言模型本身，对仅编码器（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 类似，通常采用预训练微调范式，针对不同下游任务进行微调。这些模型参数量大都在 10 亿个以上，由于微调的计算量很大，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，研究人员开始探索在不针对单一任务进行微调的情况下如何发挥大语言模型的能力。2019 年，Radford 等人在文献 [11] 中使用 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[13] 模型上研究了通过语境学习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，语言模型根据实例理解任务并给出正确的结果。基于 GPT-3 的语境学习在 TriviaQA、WebQS、CoQA 等评测集合中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须花费大量计算资源进行模型微调。仅依赖语言模型本身，其性能在很多任务上仍然很难达到有监督学习（Supervised Learning）的效果，因此研究人员提出了指令微调[23] 方案，将大量各类型任务统一为生成式自然语言理解框架，并构造训练数据进行微调。大语言模型能一次性学习数千种任务，并在未知任务上展现出很好的泛化能力。2022 年，Ouyang 等人提出了使用“有监督微调 + 强化学习”的 InstructGPT[24] 方法，该方法使用少量有监督数据就可以使大语言模型服从人类指令。Nakano 等人则探索了结合搜索引擎的问题回答方法 WebGPT[25]。这些方法在直接利用大语言模型进行零样本和少样本学习的基础上，逐渐扩展为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、数学解题等过去自然语言处理系统需要大量小模型定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及对话上下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，GPT-4 有非常明显的进步，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。GPT-4o 是 OpenAI 于 2024 年 5 月发布的多模态大模型，其中“o”代表“omni”即“全能”。它能接受文本、音频和图像组合输入并生成文本、音频和图像的任意组合输出，可处理 50 种语言，在 232 毫秒内对音频

输入做出反应，性能较 GPT-4 有显著提升。2024 年 9 月 OpenAI 又推出的全新推理模型 GPT-o1，在复杂推理任务上表现卓越，能通过内部思维链模拟人类思考，在数学、科学等领域超越人类专家及 GPT-4o。国内外各大公司和研究机构相继发布了此类系统，包括复旦大学的 MOSS、阿里巴巴的 Qwen、深度求索的 DeepSeek、Google 的 Gemini、XAI 的 Grok、科大讯飞的星火大模型、智谱的 ChatGLM 等。

表1.1 和表1.2 分别给出了截至 2025 年 2 月典型开源和闭源大语言模型的基本情况。可以看到，从 2022 年开始，大语言模型的数量呈爆发式的增长，各大公司和研究机构都在发布不同类型的大语言模型。模型类型中，基础模型是指仅经过预训练的模型；对话模型是指在预训练模型基础上经过有监督微调和强化学习训练的模型，具备对话和完成任务的能力；推理模型是指专注于逻辑推理增强的大语言模型。

表 1.1 典型开源大语言模型汇总

模型名称	发布时间	参数量（个）	模型类型	预训练数据量
T5[19]	2019 年 10 月	110 亿	基础模型	1 万亿个词元
PanGu-α[22]	2021 年 4 月	130 亿	基础模型	1.1 万亿个词元
CPM-2[26]	2021 年 6 月	1980 亿	基础模型	2.6 万亿个词元
CodeGen[27]	2022 年 3 月	160 亿	基础模型	5770 亿个词元
GPT-NeoX-20B[28]	2022 年 4 月	200 亿	基础模型	825GB
OPT[29]	2022 年 5 月	1750 亿	基础模型	1800 亿个词元
GLM[30]	2022 年 10 月	1300 亿	基础模型	4000 亿个词元
Flan-T5[23]	2022 年 10 月	110 亿	对话模型	-
BLOOM[31]	2022 年 11 月	1760 亿	基础模型	3660 亿个词元
BLOOMZ[32]	2022 年 11 月	1760 亿	对话模型	-
OPT-IML[33]	2022 年 12 月	1750 亿	对话模型	-
LLaMA[34]	2023 年 2 月	652 亿	基础模型和对话模型	1.4 万亿个词元
MOSS	2023 年 2 月	160 亿	对话模型	-
ChatGLM-6B[30]	2023 年 4 月	62 亿	基础模型和对话模型	-
Alpaca[35]	2023 年 4 月	130 亿	对话模型	-
Falcon	2023 年 5 月	400 亿	基础模型	1 万亿个词元
OpenLLaMA	2023 年 5 月	130 亿	基础模型	1 万亿个词元
Gorilla[36]	2023 年 5 月	67 亿	对话模型	-
Baichuan	2023 年 6 月	70-130 亿	基础模型和对话模型	1.4 万亿个词元
LLaMA2[37]	2023 年 7 月	70-700 亿	基础模型和对话模型	2.0 万亿个词元
Qwen	2023 年 8 月	70 亿	基础模型和对话模型	3.0 万亿个词元
ChatGLM3-6B	2023 年 9 月	60 亿	基础模型和对话模型	1.0 万亿个词元
Mistral 7B	2023 年 9 月	70 亿	基础模型和对话模型	8.0 万亿个词元
InternLM-20B	2023 年 9 月	200 亿	基础模型和对话模型	2.3 万亿个词元
Grok-1	2023 年 10 月	3140 亿	基础模型和对话模型	-
DeepSeek-LLM	2023 年 11 月	70-670 亿	基础模型和对话模型	2.0 万亿个词元

续表

模型名称	发布时间	参数量（个）	模型类型	预训练数据量
Qwen 1.5	2024 年 2 月	5-720 亿	基础模型和对话模型	3.0 万亿个词元
Gemma	2024 年 2 月	20-70 亿	基础模型和对话模型	6.0 万亿个词元
MiniCPM-2B	2024 年 2 月	20 亿	基础模型和对话模型	1.0 万亿个词元
Grok-1	2024 年 2 月	3140 亿	对话模型	–
LLaMA 3	2024 年 4 月	80-700 亿	基础模型和对话模型	15.0 万亿个词元
Phi-3	2024 年 4 月	38-140 亿	对话模型	4.8 万亿个词元
GLM-4-9B	2024 年 6 月	90 亿	基础模型和对话模型	10.0 万亿个词元
LLaMA 3.1	2024 年 7 月	80-4050 亿	基础模型和对话模型	15.0 万亿个词元
Qwen 2.5	2024 年 9 月	5-720 亿	基础模型和对话模型	18.0 万亿个词元
LLaMA 3.2	2024 年 9 月	10-900 亿	基础模型和对话模型	15.0 万亿个词元
Hunyuan-Large	2024 年 11 月	3890 亿	基础模型和对话模型	7.0 万亿个词元
DeepSeek-V3	2024 年 12 月	6710 亿	对话模型	14.8 万亿个词元
Phi-4	2024 年 12 月	140 亿	对话模型	10.0 万亿个词元
DeepSeek-R1	2025 年 1 月	6710 亿	推理模型	14.8 万亿个词元

表 1.2 典型闭源大语言模型汇总

模型名称	发布时间	发布公司	参数量（个）	模型类型
GPT-3	2020 年 5 月	OpenAI	1750 亿	基础模型
ERNIE 3.0	2021 年 7 月	百度	100 亿	基础模型
Claude	2021 年 12 月	Anthropic	520 亿	基础模型
InstructGPT	2022 年 3 月	OpenAI	1750 亿	对话模型
PaLM	2022 年 4 月	Google	5400 亿	基础模型
ChatGPT 3.5	2022 年 11 月	OpenAI	1750 亿 [1]	对话模型
GPT-4	2023 年 3 月	OpenAI	17600 亿 [1]	对话模型
PanGu-Σ	2023 年 3 月	华为	10850 亿	对话模型
ChatGLM	2023 年 3 月	智谱华章	1300 亿	对话模型
文心一言	2023 年 4 月	百度	-	对话模型
通义千问	2023 年 5 月	阿里巴巴	-	对话模型
MinMax	2023 年 5 月	稀宇科技	-	对话模型
星火	2023 年 5 月	科大讯飞	-	对话模型
浦语书生	2023 年 6 月	浦江实验室	-	对话模型
Claude 2	2023 年 7 月	Anthropic	-	对话模型
Baichuan2	2023 年 9 月	百川	530 亿	对话模型
Kimi	2023 年 10 月	月之暗面	-	对话模型
Gemini	2023 年 12 月	Google	-	对话模型
GLM-4	2024 年 1 月	智谱华章	-	对话模型
Claude 3	2024 年 1 月	Anthropic	-	对话模型
GPT-4o	2024 年 5 月	OpenAI	2000 亿 [1]	对话模型
豆包	2024 年 5 月	字节跳动	-	对话模型
星火 2.0	2024 年 6 月	科大讯飞	-	对话模型
Step-2	2024 年 7 月	阶跃星辰	10000 亿	对话模型
GPT-o1	2024 年 9 月	OpenAI	3000 亿 [1]	对话模型
Claude 3.5	2024 年 10 月	Anthropic	-	对话模型
GPT-o3	2024 年 12 月	OpenAI	-	推理模型
豆包 1.5Pro	2025 年 1 月	字节跳动	-	对话模型
Grok-3	2025 年 2 月	XAI	-	对话推理模型

[1] 模型参数量根据微软公司发表的文献 [38] 获取，数字并未得到 OpenAI 官方证实

1.3 大语言模型的构建流程

根据 OpenAI 联合创始人 Andrej Karpathy 在微软 Build 2023 大会上公开的信息，OpenAI 使用的大语言模型构建流程如图1.3 所示，主要包含四个阶段：预训练、有监督微调、奖励建模和强化学习。这四个阶段都需要不同规模的数据集及不同类型的算法，会产出不同类型的模型，所需要的资源也有非常大的差别。

图 1.3 OpenAI 使用的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（数据来自互联网网页、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 和高速网络组成的超级计算机，花费数十天完成深度神经网络参数训练，构建基础模型（Base Model）。基础模型对长文本进行建模，使模型具有语言生成能力，根据输入的提示词，模型可以生成文本补全句子。有一部分研究人员认为，语言模型建模过程中隐含地构建了包括事实性知识（Factual Knowledge）和常识性知识（Commonsense）在内的世界知识（World Knowledge）。根据文献 [39] 中的介绍，GPT-3 完成一次训练的总计算量是 3640PFLOPS，按照 NVIDIA A100 80GB GPU 和平均利用率达到 50% 计算，需要花费近一个月的时间使用 1000 块 GPU 完成。由于 GPT-3 的训练采用 NVIDIA V100 32GB GPU，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 亿个的 OPT 模型，该模型训练使用 992 块 NVIDIA A100 80GB GPU，整体训练时间将近 2 个月。BLOOM[31] 模型的参数量也是 1750 亿个，该模型训练一共花费 3.5 个月，使用包含 384 块 NVIDIA A100 80GB GPU 集群完成。可以看到，大语言模型的训练需要花费大量的计算资源和时间。LLaMA、Falcon、百川（Baichuan）等模型都属于基础语言模型。即便是 DeepSeek-V3[40] 经过了大量的训练效率优化，甚至已经直接使用 PTX 进行汇编级优化，完成一次预训练仍然需要

花费 266.4 万 H800 GPU 小时。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，如何提升分布式计算效率并使模型训练稳定收敛是本阶段的研究重点。

有监督微调（Supervised Fine Tuning，SFT），也称为**指令微调**，利用少量高质量数据集，通过有监督训练使模型具备问题回答、翻译、写作等能力。有监督微调的数据包含用户输入的提示词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：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，使用与预训练阶段相同的语言模型训练算法，在基础模型的基础上进行训练，得到有监督微调模型（SFT 模型）。经过训练的 SFT 模型具备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答、阅读理解、翻译、生成代码等任务，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所需的训练数据量较少，SFT 模型的训练过程并不需要消耗大量的计算资源。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，很多类 ChatGPT 的模型都属于该类型，包括 Alpaca[35]、Vicuna[41]、MOSS、ChatGLM-6B 等。很多这类模型的效果非常好，甚至在一些评测中达到了 ChatGPT 的 90% 的效果[35, 41]。当前的一些研究表明，有监督微调阶段的数据选择对 SFT 模型效果有非常大的影响[42]，因此构造少量并且高质量的训练数据是本阶段的研究重点。

奖励建模（Reward Modeling）阶段的目标是构建一个文本质量对比模型。对于同一个提示词，SFT 模型对给出的多个不同输出结果的质量进行排序。奖励模型可以通过二分类模型，对输入的两个结果之间的优劣进行判断。奖励模型与基础模型和 SFT 模型不同，奖励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，使用数十块 GPU，通过数天时间完成训练。由于奖励模型的准确率对强化学习阶段的效果有至关重要的影响，因此通常需要大规模的训练数据对该模型进行训练。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很长时间才能完成。图1.4 给出了 InstructGPT 系统中奖励模型训练样本标注示例[24]。可以看到，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，标注者也需要认真地基于标注规范进行标注，需要消耗大量的人力。同时，保持众包标注者之间的一致性，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是本阶段需要重点研究的一个问题。如果奖励模型的目标是针对系统所有的输出都能够高质量地进行判断，那么该问题的难度在某种程度上与文本生成等价，因此限定奖励模型应用的泛化边界是本阶段需要解决的问题。

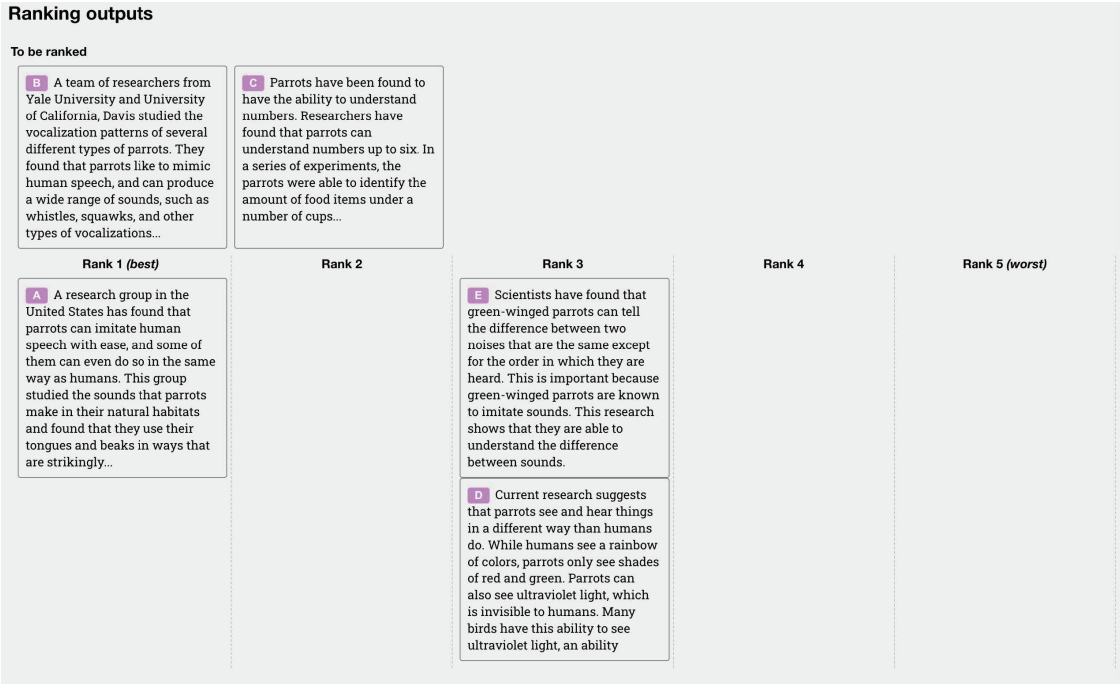

图 1.4　InstructGPT 系统中奖励模型训练样本标注示例[24]

强化学习（Reinforcement Learning，RL）阶段根据数十万条提示词，利用前一阶段训练的奖励模型，给出 SFT 模型对提示词回答结果的质量评估，并与语言模型建模目标综合得到更好的效果。该阶段使用的提示词数量与有监督微调阶段类似，数量在十万个量级，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，在 SFT 模型的基础上调整参数，使最终生成的文本可以获得更高的奖励（Reward）。该阶段需要的计算量较预训练阶段也少很多，通常仅需要数十块 GPU，数天即可完成训练。文献 [24] 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得多的效果。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，截至 2025 年 2 月还没有完整或得到普遍共识的解释。目前相对得到认可的观点是，强化学习使得模型具备更好的泛化能力[43]。同时，Andrej Karpathy 也指出，强化学习并不是没有问题的，它会使基础模型的熵降低，从而减少模型输出的多样性。经过强化学习方法训练后的 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，叠加奖励模型的准确率问题，使得在大语言模型上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 12 章，围绕大语言模型基础理论、预训练、指令理解、大模型增强和大模型应用五个部分展开：第一部分介绍大语言模型的基础理论；第二部分介绍大语言模型的预训练，包括大语言模型预训练数据和分布式训练；第三部分介绍大语言模型如何理解并服从人类指令，包括有监督微调和强化学习；第四部分介绍大语言模型增强技术，包括多模态大语言模型、大模型智能体和检索增强生成；第五部分介绍大模型应用，包括大语言模型效率优化、大语言模型评估和大语言模型应用开发。具体章节安排如图1.5 所示。

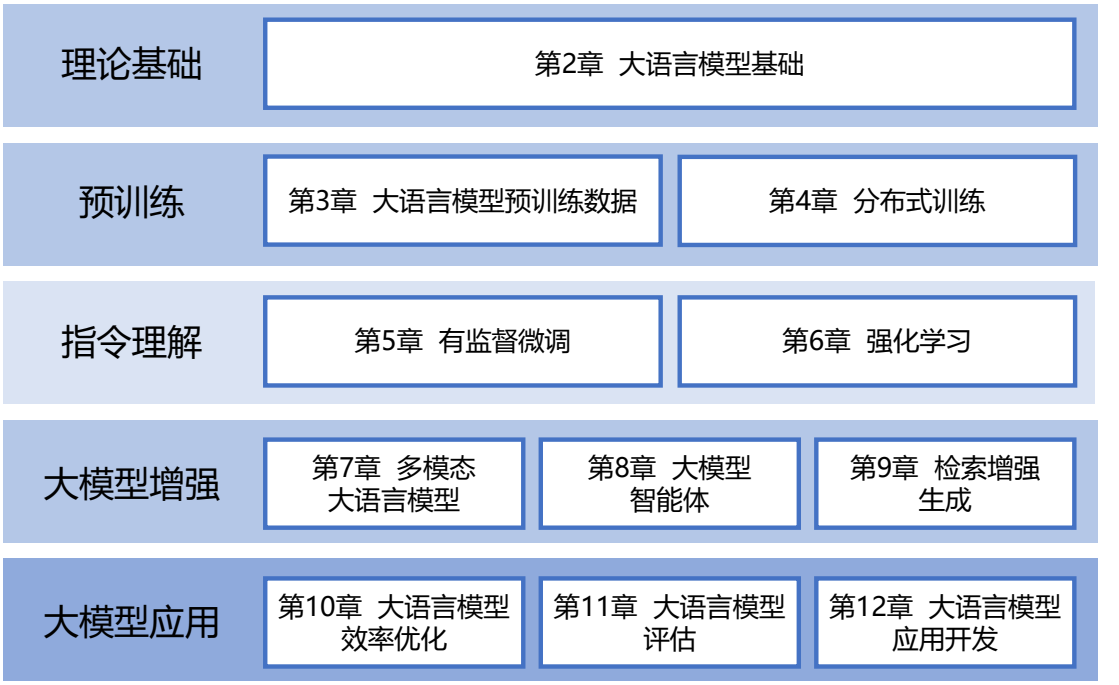

图 1.5 本书章节安排

第 2 章介绍大语言模型的基础理论知识，包括语言模型的定义、Transformer 结构、大语言模型框架等内容，并以 LLaMA 使用的模型结构为例介绍代码实例。

第 3 章和第 4 章围绕大语言模型预训练阶段的主要研究内容开展介绍，包括模型分布式训练中需要掌握的数据并行、流水线并行、模型并行及 ZeRO 系列优化方法。除此之外，还将介绍预训练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 章聚焦于大语言模型指令理解阶段的核心研究内容，探讨如何通过有监督微调和强化学习方法，使模型能够理解指令并生成类人回答。第 5 章重点介绍模型微调技术，有监督微调数据的构造策略以及高效微调方法：LoRA、Delta Tuning 等方法；第 6 章则围绕强化学习展开，讲解其基础理论与近端策略优化（PPO）技术，并结合实际案例，以 DeepSpeed-Chat 和 veRL 框架为例，详细说明如何训练类 ChatGPT 系统。

第 7 章、第 8 章和第 9 章围绕提升大语言模型能力展开详细探讨，内容涵盖多模态大语言模型、智能体实践及检索增强生成。第 7 章重点介绍多模态大语言模型的基础理论、架构设计与训练策略，并探讨其在实际场景中的应用实践；第 8 章聚焦智能体的发展历程与大语言模型智能体的架构设计，深入分析智能体的实现原理，并以 LangChain 为例详细阐述具体实践；第 9 章则围绕检索增强生成展开讨论，介绍其核心思想与实现方式，涵盖检索增强框架的设计、检索模块与

生成模块的协作机制，以及其在具体任务场景中的应用方法与实践。

第 10 章、第 11 章和第 12 章主要围绕如何应用大语言模型展开讨论，内容涵盖提升模型效率的方法、大语言模型评估，以及典型应用的开发与部署。第 10 章重点介绍模型压缩与优化、训练效率优化和推理效率优化等提升模型效率的关键技术；第 11 章聚焦于大语言模型评估，探讨其基本概念和难点，阐述评估体系的构建、评估方法的设计以及实际评估的实施；第 12 章则基于典型的大语言模型应用场景，详细介绍开发流程、开发工具及本地部署的实践方法。

2. 大语言模型基础

语言模型的核心目标是对自然语言的概率分布进行建模，这一任务在自然语言处理研究中占据重要地位，是其基础性工作之一。大量研究围绕这一目标，从不同角度展开了探索，包括 n 元语言模型（n-gram Language Models）、神经语言模型和预训练语言模型等。这些研究在不同发展阶段对自然语言处理任务产生了深远影响。随着基于 Transformer 架构的语言模型不断发展，以及预训练-微调范式在各类自然语言处理任务中取得突破性成果，自 2020 年 OpenAI 发布 GPT-3 以来，大语言模型的研究逐步深入。尽管大语言模型参数规模庞大，并且通过有监督微调和强化学习可以完成众多任务，其理论基础仍然离不开对语言建模的核心研究。

本章首先介绍 Transformer 结构，并在此基础上讲解生成式预训练语言模型 GPT、大语言模型的网络结构、注意力机制优化及相关实践。关于 n 元语言模型、神经语言模型及其他预训练语言模型的内容，可参考《自然语言处理导论》第 6 章[4]。

2.1 Transformer 结构

Transformer 结构[44] 是由 Google 在 2017 年提出并首先应用于机器翻译的神经网络模型架构。机器翻译的目标是从源语言（Source Language）转换到目标语言（Target Language）。Transformer 结构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。如今，几乎全部大语言模型都是基于 Transformer 结构的。本节以应用于机器翻译的基于 Transformer 的编码器和解码器结构为例介绍该模型。

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（Decoder）结构，它们均由若干个基本的 Transformer 块（Block）组成（对应图中的灰色框）。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 $\boldsymbol{y}_i$ 分别对应文本序列中的一个词元（Token）的表示。$\boldsymbol{y}_i$ 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，主要涉及如下几个模块。

- **注意力层**：使用**多头注意力**（Multi-Head Attention）机制整合上下文语义。多头注意力并行

运行多个独立注意力机制，进而从多维度捕捉输入序列信息。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问题。

- **位置感知前馈网络层**（Position-wise Feed-Forward Network）：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中的直连通路，被用于连接两个子层的输入与输出，使信息流动更高效，有利于模型的优化。
- **层归一化**：对应图中的 Norm 部分。它作用于上述两个子层的输出表示序列，对表示序列进行层归一化操作，同样起到稳定优化的作用。

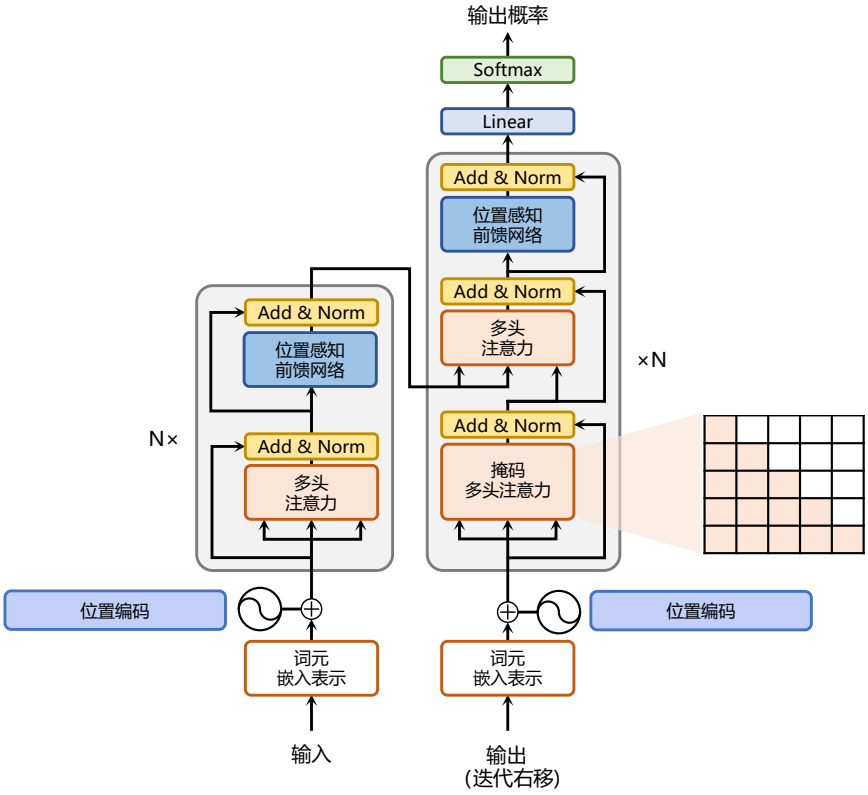

图 2.1　基于 Transformer 的编码器和解码器结构[44]

接下来依次介绍各个模块的具体功能和实现方法。

2.1.1 嵌入表示层

对于输入文本序列，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，直接对每个单词创建一个向量表示。Transformer 结构不再使用基于循环的方式建模文本输入，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 Encoding）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后续模块中做进一步处理。在训练过程中，模型会自动地学习到如何利用这部分位置信息。

为了得到不同位置所对应的编码，Transformer 结构使用不同频率的正余弦函数，如下所示。

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right) \tag{2.1}$$

$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right) \tag{2.2}$$

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以下两个好处：第一，正余弦函数的范围是 $[-1, +1]$，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；第二，依据三角函数的基本性质，可以得知第 $\mathrm{pos}+k$ 个位置编码是第 pos 个位置编码的线性组合，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 ** (i/d_model)))
                pe[pos, i + 1] =  math.cos(pos / (10000 ** (i/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
```

2.1.2 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。将由单词语义嵌入及其位置编码叠加得到的输入表示为 $\{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^L$，为了实现对上下文语义依赖的建模，引入自注意力机制涉及的三个元素：查询 $\boldsymbol{q}_i$（Query）、键 $\boldsymbol{k}_i$（Key）和值 $\boldsymbol{v}_i$（Value）。在编码输入序列的每一个单词的表示中，这三个元素用于计算上下文单词对应的权重得分。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 $\boldsymbol{W}^{\boldsymbol{Q}} \in \mathbb{R}^{d\times d_q}$, $\boldsymbol{W}^{\boldsymbol{K}} \in \mathbb{R}^{d\times d_k}$, $\boldsymbol{W}^{\boldsymbol{V}} \in \mathbb{R}^{d\times d_v}$ 将输入序列中的每一个单词表示 $\boldsymbol{x}_i$ 转换为其对应的 $\boldsymbol{q}_i \in \mathbb{R}^{d_q}$, $\boldsymbol{k}_i \in \mathbb{R}^{d_k}$, $\boldsymbol{v}_i \in \mathbb{R}^{d_v}$ 向量。对于输入 $\{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^L$，$\boldsymbol{Q}$、$\boldsymbol{K}$ 和 $\boldsymbol{V}$ 矩阵可以通过如下公式所示：

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^{\boldsymbol{Q}} \tag{2.3}$$

$$\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^{\boldsymbol{K}} \tag{2.4}$$

$$\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^{\boldsymbol{V}} \tag{2.5}$$

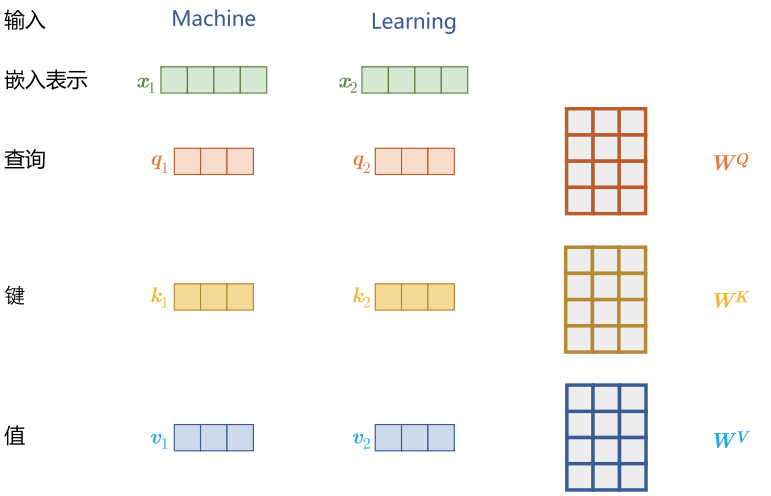

图 2.2 自注意力机制中的查询、键、值

为了得到编码单词 $\boldsymbol{x}_i$ 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 $\boldsymbol{q}_i \cdot \boldsymbol{k}_1, \boldsymbol{q}_i \cdot \boldsymbol{k}_2, \cdots, \boldsymbol{q}_i \cdot \boldsymbol{k}_t$。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及收敛效率差的问题，这些得分会除以放缩因子 $\sqrt{d}$ 以稳定优化。放缩后的得分经过 Softmax 归一化为概率，与其他位置的值向量相乘来聚合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：

$$\boldsymbol{Z} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}\right)\boldsymbol{V} \tag{2.6}$$

其中 $\boldsymbol{Q} \in \mathbb{R}^{L\times d_q}, \boldsymbol{K} \in \mathbb{R}^{L\times d_k}, \boldsymbol{V} \in \mathbb{R}^{L\times d_v}$ 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，提出了多头注意力机制，以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 $\boldsymbol{x}_i$ 经过多组线性 $\{\boldsymbol{W}_j^{\boldsymbol{Q}}, \boldsymbol{W}_j^{\boldsymbol{K}}, \boldsymbol{W}_j^{\boldsymbol{V}}\}_{j=1}^N$ 映射到不同的表示子空间中。公式 (2.6) 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 $\{\boldsymbol{Z}_j\}_{j=1}^N$：

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

在此基础上，经过线性变换 $\boldsymbol{W}^{\boldsymbol{O}} \in \mathbb{R}^{(Nd_v)\times d}$ 用于综合不同子空间中的上下文表示并形成注意力层最终的输出 $\{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^L$，可得到**多头自注意力**（Multi-Head Self-Attention）表示：

$$\boldsymbol{Z} = \text{Concat}(\boldsymbol{Z}_1, \boldsymbol{Z}_2, ..., \boldsymbol{Z}_N)\boldsymbol{W}^{\boldsymbol{O}} \tag{2.8}$$

由此可见，自注意力机制使模型能够识别不同输入部分的重要性，而不受距离的影响，从而能够捕捉输入句子中的长距离依赖关系和复杂关系。

使用 PyTorch 实现的自注意力层参考代码如下：

```python
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)

        # 掩盖那些为了补全长度而增加的单元，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)

        # 利用线性计算划分成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)
```

2.1.3 前馈层

前馈层接收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。

$$\mathrm{FFN}(\boldsymbol{x}) = \mathrm{ReLU}(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{2.9}$$

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高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)
        return x
```

2.1.4 残差连接与层归一化

由 Transformer 结构组成的网络结构通常都非常庞大。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，研究人员在 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，避免在优化过程中因网络过深而产生潜在的梯度消失问题：

$$\boldsymbol{x}^{l+1} = f(\boldsymbol{x}^l) + \boldsymbol{x}^l \tag{2.10}$$

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，为了使每一层的输入/输出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 块中：

$$\mathrm{LN}(\boldsymbol{x}) = \alpha \cdot \frac{\boldsymbol{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} + b \tag{2.11}$$