一. 什么是语言人工智能？

Language AI（语言人工智能）是人工智能（AI）的一个分支，其目标是让机器能够理解、生成、解释和操纵人类语言。它不仅仅是简单的文本处理，而是旨在让机器获得类似人类的语言能力，从而在人与机器之间实现自然、流畅的交流。也就是使用数据和算法让机器能够理解、生成人类语言，目前能够做到与人类进行对话。随着机器学习方法在解决语言处理问题上的不断成功，“语言 AI”通常可以与“自然语言处理”（natural language processing）互换。

二. 语言人工智能的发展历史

语言 AI 的发展历史出现了许多表示模型（Representing Model）和生成模型（Generative Model），如图 1-1 所示。只有编码器（Encoder-only）、只有解码器（Decoder-only）、编码器-解码器（Encoder-decoder）三种架构是现代 Transformer 模型的基础，它们有各自独特的设计、工作原理和适用场景。本书中使用的模型都是基于 transformer 的模型。

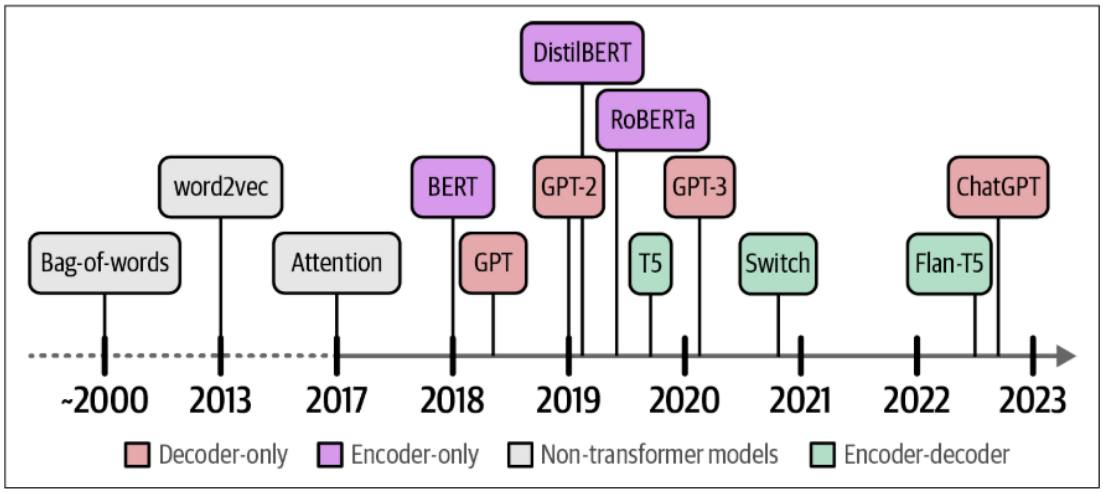

Figure 1-1. A peek into the history of Language AI.

① Encoder-only

核心功能：理解与表征。

工作方式：接收输入文本，为每个单词（或子词[token]）生成一个富含上下文信息的“向量表示”。这些表示捕获了单词在特定句子中的含义。内部结构：通常由多层自注意力层和前馈神经网络组成。自注意力机制让模型能够看到输

入序列中所有其他词的信息，从而更好地理解当前词。BERT 模型是 Encoder-only 模型中最著名的代表。

主要应用：文本分类（如情感分析、垃圾邮件识别）、自然语言推理（判断两个句子的关系是蕴含、矛盾还是中立）、命名实体识别（找出文本中的人名、地名、机构名）、词性标注（如名词、动词等）、句子相似度计算。

② Decoder-only

核心功能：生成。

工作方式：以一段起始文本（提示或前缀）开始，然后自回归地生成后续的文本，即一次生成一个词，并将之前生成的词作为新的输入的一部分，逐步产生完整的输出。内部结构：同样由多层 Transformer 块组成，但为了确保生成过程是顺序的（不能“偷看”未来的词），它使用了掩码自注意力。在生成第 i 个词时，它只能关注到第 1 到第 i-1 个词。GPT 系列是典型代表，这一系列模型极大地推动了只有解码器架构的发展。

主要应用：文本生成（写文章、写诗、写代码）、对话机器人（ChatGPT）、代码补全、问答（基于给定的知识生成答案）。

③ Encoder-decoder

核心功能；转换与重构。

工作方式：**编码器**：首先像“只有编码器的模型”一样，全面理解并编码输入序列，生成一组上下文向量。**解码器**：然后像“只有解码器的模型”一样，以编码器的输出为条件，自回归地生成输出序列。关键在于，解码器在生成每个词时，不仅会关注它已经生成的内容（通过掩码自注意力），还会交叉关注编码器输出的最终表示。内部结构：完整包含了编码器和解码器两部分，并通过“编码器-解码器注意力层”（交叉注意力）连接它们。**原始的 Transformer** 论文（Attention is all you need, 2017）本身就是为机器翻译这个典型的“序列到序列”任务设计的，它是编码器-解码器架构的鼻祖。

主要应用：机器翻译（将一种语言的句子转换为另一种语言）、文本摘要（将长文本压缩为短摘要）、语音识别（将语音信号转换为文本）、问答（给定问题和上下文，生成答案）。结合了前两种模型的优点，先全面理解输入，再有序生成输出，非常适合需要**转换或重构输入信息**的任务。

模型类型	核心架构	主要功能	典型代表
表示模型	**通常**是只有编码器	理解、分析、分类	BERT
生成模型	**通常**是只有解码器	创造、生成文本	GPT
序列到序列模型	**编码器-解码器**	转换、生成	T5, BART

在基于 Transformer 的现代语言模型中，表示任务（如分类、理解）通常由基于编码器的架构（如 BERT）完成，而生成任务（如写作、对话）通常由基于解码器的自回归架构（如 GPT）完成。

由于自然语言对于计算机来说是个棘手的概念，文本本质上是非结构化的，当用 0 和 1（单个字符）表示时就会失去其意义。因此，在整个语言人工智能的发展历程中，如何以结构化方式表示语言始终是重点研究方向，以便计算机能更轻松地处理语言。图 1-2 展示了这类语言人工智能任务的示例。

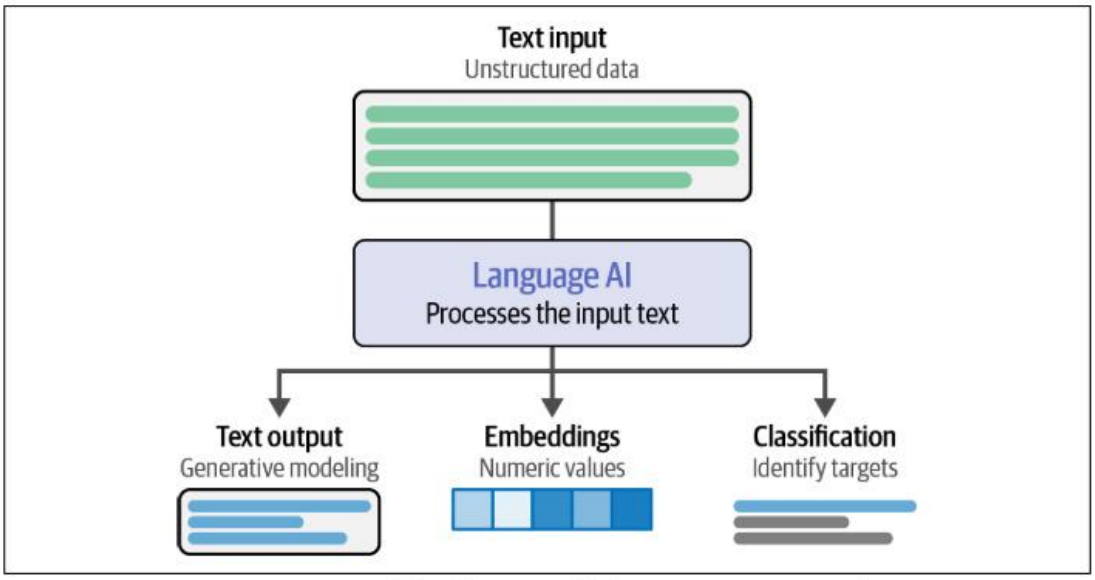

Figure 1-2. Language AI is capable of many tasks by processing textual input.

1. 用 Bag-of-Words（词袋模型）来表示语言

语言人工智能的历史始于一种名为词袋的技术，这是一种表示非结构化文本的方法。它最早在 20 世纪 50 年代被提及，但在 21 世纪头 10 年左右流行起来。词袋的工作原理如下：让我们假设我们有两个句子，我们要为它们创建数字表示法。词袋模型的第一步是 tokenization（分词化），即将句子拆分成单独的单词或子词(token)的过程，如图 1-3 所示。在下一章中，我们将深入讨论 Tokenization

和这项技术对语言模型的影响。

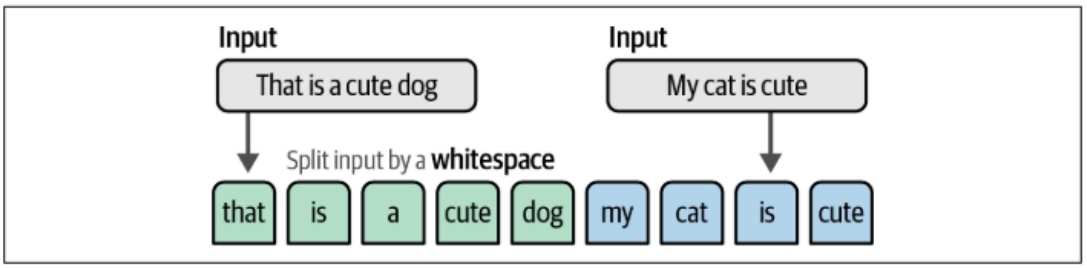

Figure 1-3. Each sentence is split into words (tokens) by splitting on a whitespace.

如图 4 所示，在 tokenization 之后，将两个句子中所有出现的不重复单词汇集成一个词汇表（Vocabulary）。

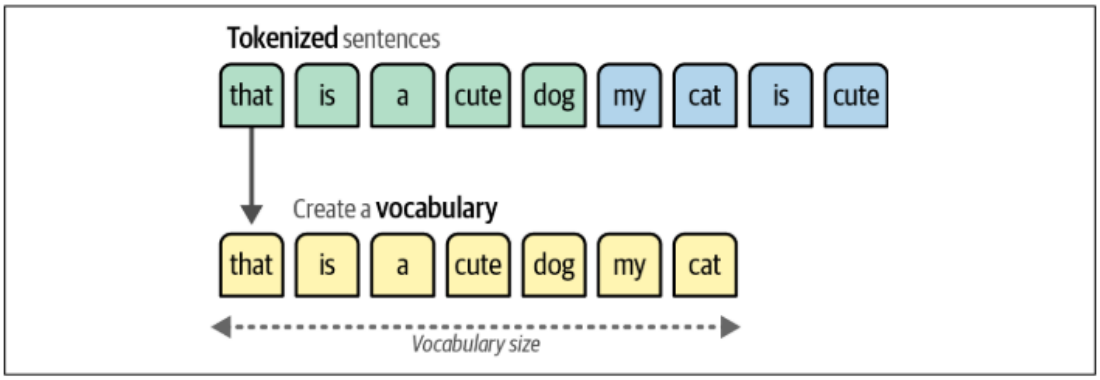

Figure 1-4. A vocabulary is created by retaining all unique words across both sentences.

然后使用词汇表计算每句话中每一个单词出现的频率，这样一来就从人类可读的句子转换成了机器可以处理的数字形式的文本表示，也称为向量或向量表示，如图 1-5 所示。这种类型的模型称为表示模型。将所有句子的计数向量组合起来，就形成了一个巨大的文档-词项矩阵。这个矩阵就是词袋模型的最终产出，可以作为机器学习模型的输入。

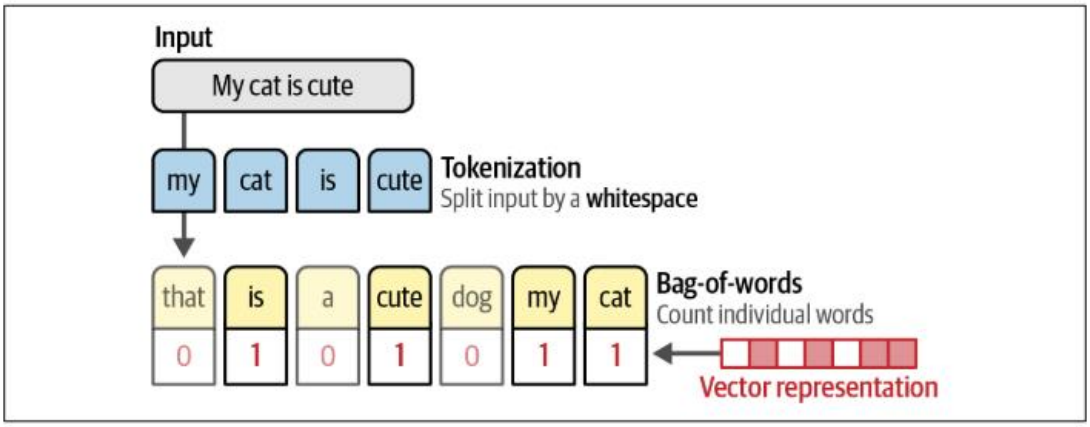

Figure 1-5. A bag-of-words is created by counting individual words. These values are referred to as vector representations.

词袋模型的**优点**：简单易懂、高效（计算开销小）、有效（在很多任务上，尤其是小型数据集上，表现 surprisingly good（出乎意料地好）。词频本身就是一个非常强的特征）。

缺点：不能捕捉语义和语序（例如，“狗咬人”和“人咬狗”在 BoW 模型中的向量表示是**完全一样**的，但它们的含义却天差地别）、忽略上下文（无法捕捉单词之间的关联和上下文信息）、高维稀疏（词汇表可能非常大（数万甚至数十万个单词），而单篇文档中出现的单词有限，导致向量中绝大部分都是 0（称为“稀疏向量”），这会带来计算和存储的挑战）、词汇鸿沟（无法处理同义词（如“高兴”和“开心”被视为完全不同的词）和多义词（如“bank”在不同语境下有不同意思，但被视为同一个词））。

2．用稠密嵌入向量来更好地表示语言

2013 年发布的 word2vec 是在嵌入向量中捕获文本含义的首批成功尝试之一。嵌入向量是试图捕获文本含义的数据的矢量表示。为了做到这一点，word2vec 通过训练大量的文本数据来学习单词的语义表示，比如整个维基百科。为了生成这些语义表示，word2vec 利用了神经网络。这些网络由相互连接的处理信息的节点层组成。如图 1-6 所示，神经网络可包含多个层级，其中每个连接根据输入数据具有特定权重。这些权重通常被称为模型参数。

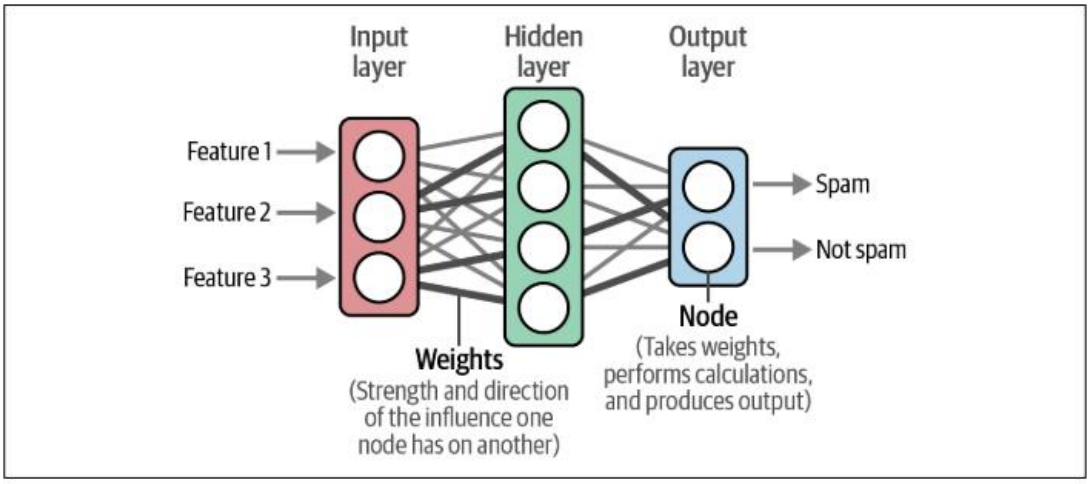

Figure 1-6. A neural network consists of interconnected layers of nodes where each connection is a linear equation.

word2vec 的训练过程：使用神经网络，word2vec 通过查看单词在句子中往往出现在哪些单词旁边来生成单词的嵌入向量。首先给词汇表中的每个单词分配一个初始嵌入向量，比如嵌入向量是 128 维，也就每个单词的嵌入有 128 个值，这些

值是用随机值初始化的。然后，如图 1-7 所示，在每个训练步骤中，我们从训练数据中提取单词对，模型预测它们在句子中是否可能是相邻的。在这个训练过程中，word2vec 学习单词之间的关系，并将这些信息提取到嵌入向量中。如果两个单词往往具有相同的邻居，则它们的嵌入将更接近，反之亦然。在第二章中，我们将详细介绍 word2vec 的训练过程。

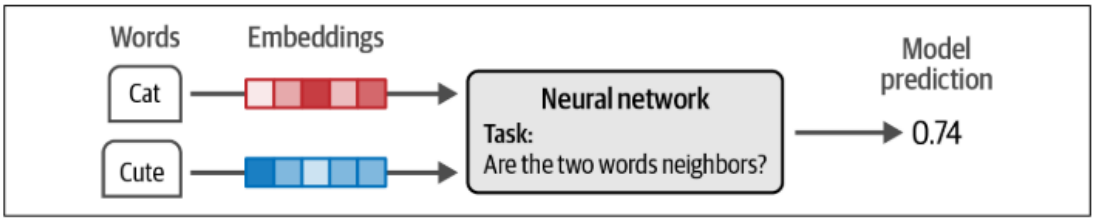

Figure 1-7. A neural network is trained to predict if two words are neighbors. During this process, the embeddings are updated to be in line with the ground truth.

Word2vec 产生的嵌入向量捕获了单词的含义，但这到底是什么意思呢？举个例子，嵌入为 128 维，每一维都代表了某种属性，“apple”和“baby”两个单词在“人类”和“水果”这两种属性上的得分有高有低。如图 1-8 所示，嵌入可以有许多属性来表示单词的含义，但是我们不能简单地认为维度代表概念，这只是一种利于理解的表达。实际上，这些属性通常是难以解释的，但是每一个维度对计算机来说是有意义的，是一种将自然语言转换为计算机语言的好方法。

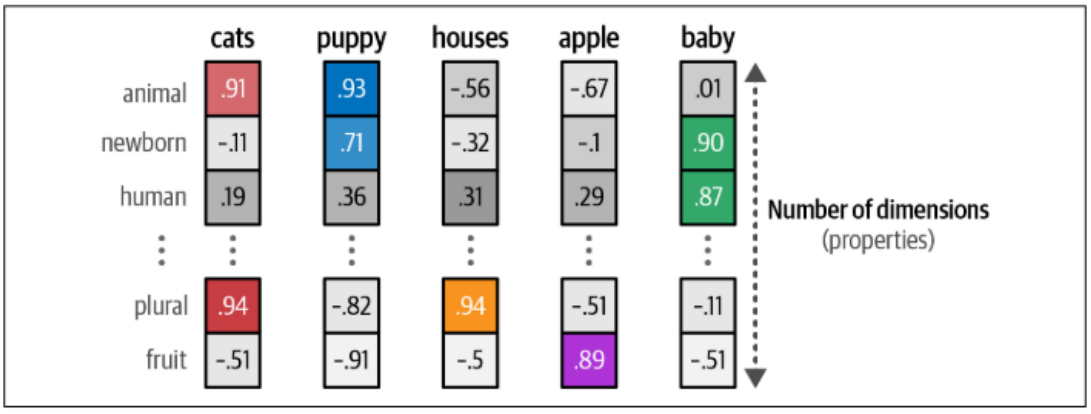

Figure 1-8. The values of embeddings represent properties that are used to represent words. We may oversimplify by imagining that dimensions represent concepts (which they don't), but it helps express the idea.

嵌入可以衡量两个单词之间的语义相似性，使用各种距离度量，我们可以判断两个词之间有多接近。如图 1-9 所示，如果将嵌入压缩到二维平面中（实际上嵌入是高维的，这只是为了便于理解），语义接近的单词嵌入往往更接近。

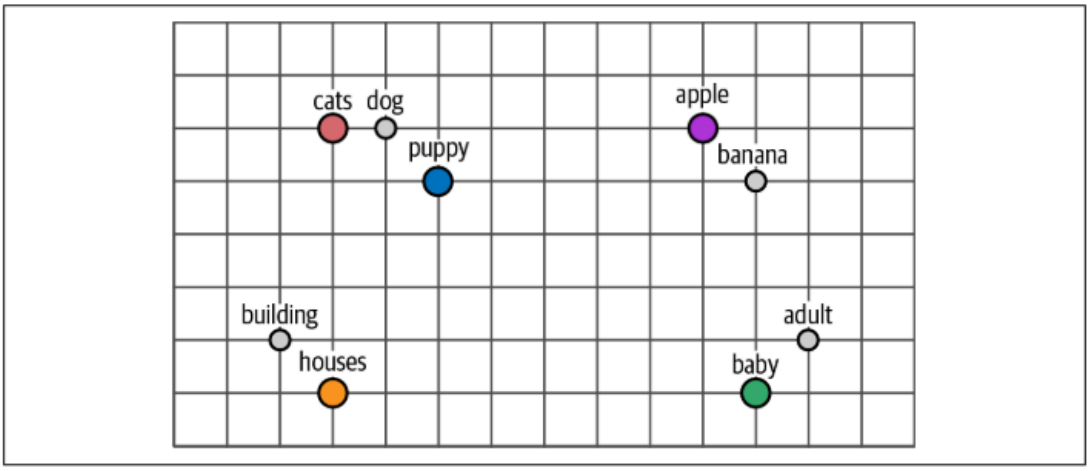

Figure 1-9. Embeddings of words that are similar will be close to each other in dimensional space.

3．嵌入向量的类型

嵌入技术存在多种类型，如词嵌入和句子嵌入，它们分别用于表示不同抽象层级（词汇级与句子级）的语义信息，如图 1-10 所示。以词袋模型为例，它在文档级别生成嵌入表示，因其旨在表征整个文档的内容；而 Word2Vec 仅针对单词生成嵌入表示。

嵌入技术将贯穿本书的核心议题，因其在众多应用场景中发挥关键作用，包括文本分类（参见第 4 章）、聚类分析（参见第 5 章）、语义搜索与检索增强生成（参见第 8 章）等。在第 2 章中，我们将深入探讨词元（Token）嵌入的技术细节。

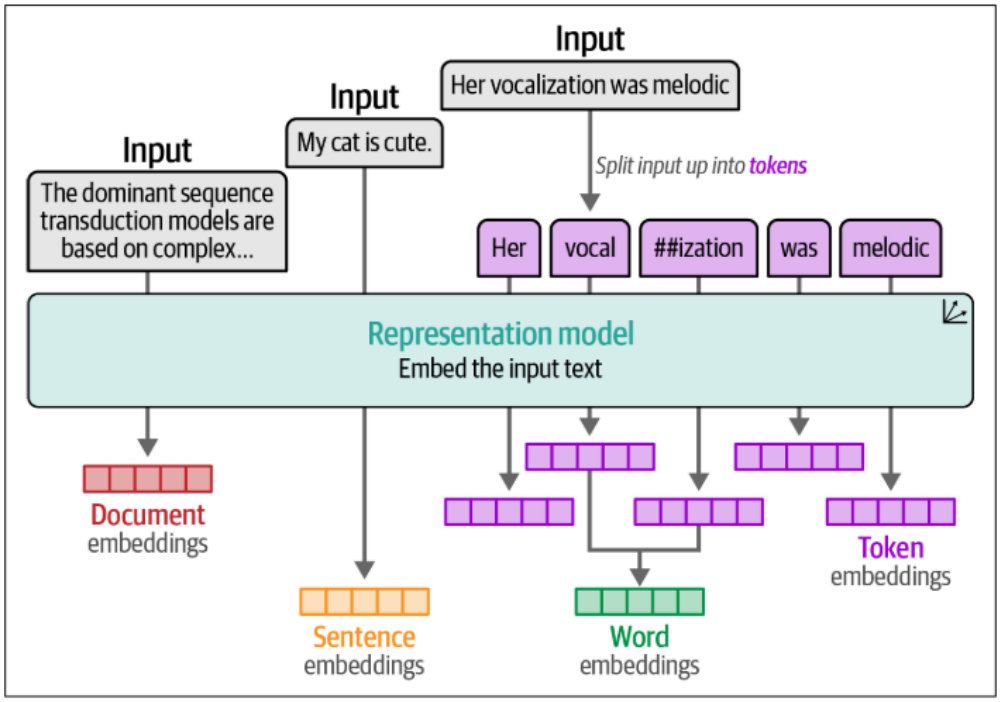

Figure 1-10. Embeddings can be created for different types of input.

4．基于注意力机制的编码和解码

上面提到的 word2vec 模型创建的嵌入向量是静态的，例如单词“bank”始终具有相同的嵌入，每一维的数值都不会改变，不管它在上下文中所代表的含义是什么。然而“bank”可以指银行也可以指河岸，它的含义应该根据上下文的语境来判断，它的嵌入也应该根据上下文而变化。

循环神经网络（RNNs）是实现此类文本编码的重要途径。作为神经网络的变体，RNNs 能够通过附加输入对序列信息进行建模。RNN 按时间步逐步处理序列（如逐词输入），通过隐藏状态传递历史信息。这些网络主要承担两项任务：对输入语句进行编码表征（encoding），以及对输出语句进行解码生成（decoding）。如图 1-11 所示，该机制生动演示了如何将英语句子"I love llamas"转化为荷兰语"Ik hou van lama's"的翻译过程。

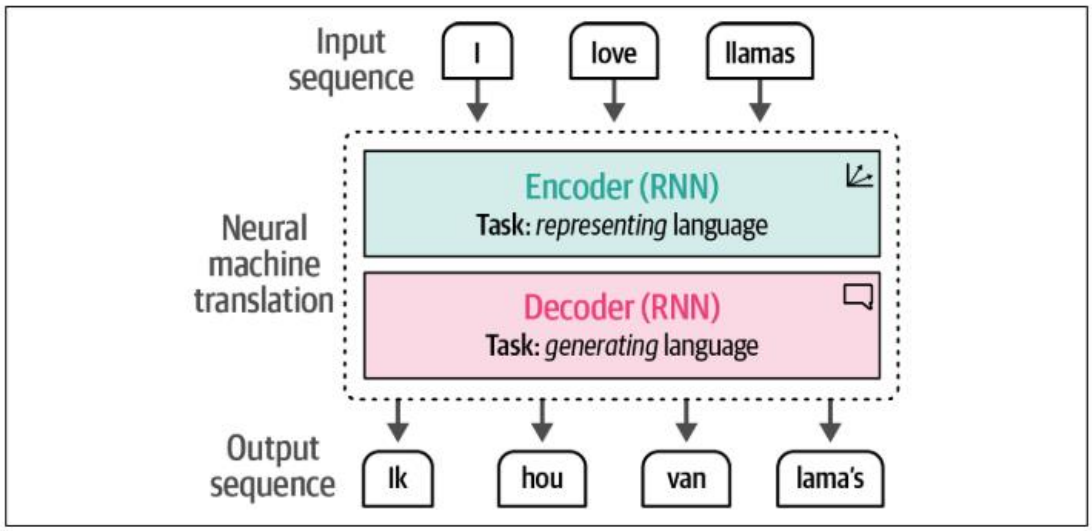

Figure 1-11. Two recurrent neural networks (decoder and encoder) translating an input sequence from English to Dutch.

该体系结构中的每一步都是自回归的。当生成下一个单词时，该体系结构需要使用所有先前生成的单词。如图 1-12 所示，每个先前的输出令牌被用作输入以生成下一个令牌。

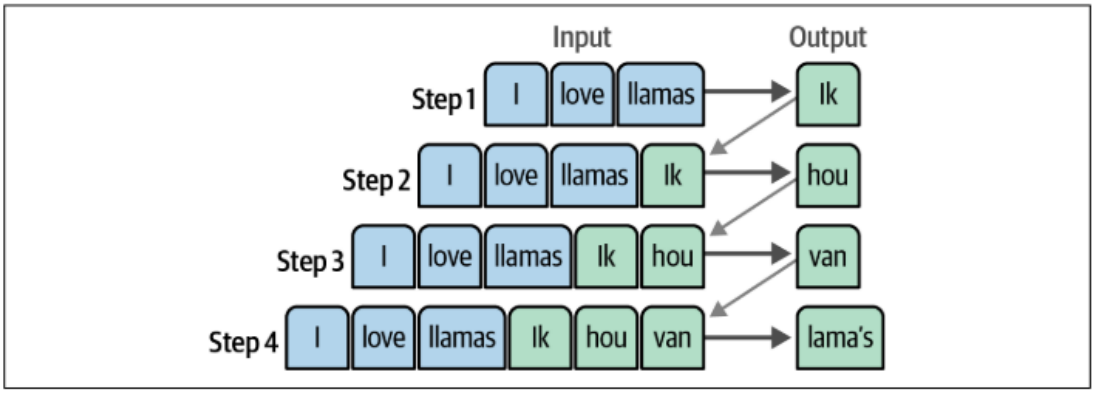

Figure 1-12. Each previous output token is used as input to generate the next token.

编码步骤的核心目标是尽可能精准地表征输入信息，通过生成嵌入形式的上下文向量，该向量将作为解码器的输入。为构建此表征，模型以词嵌入作为单词的输入，这意味着我们可以采用 Word2Vec 来生成初始表征。如图 1-13 所示，我们可以观察到这一过程。需注意输入与输出数据均是按顺序逐个处理的特性。

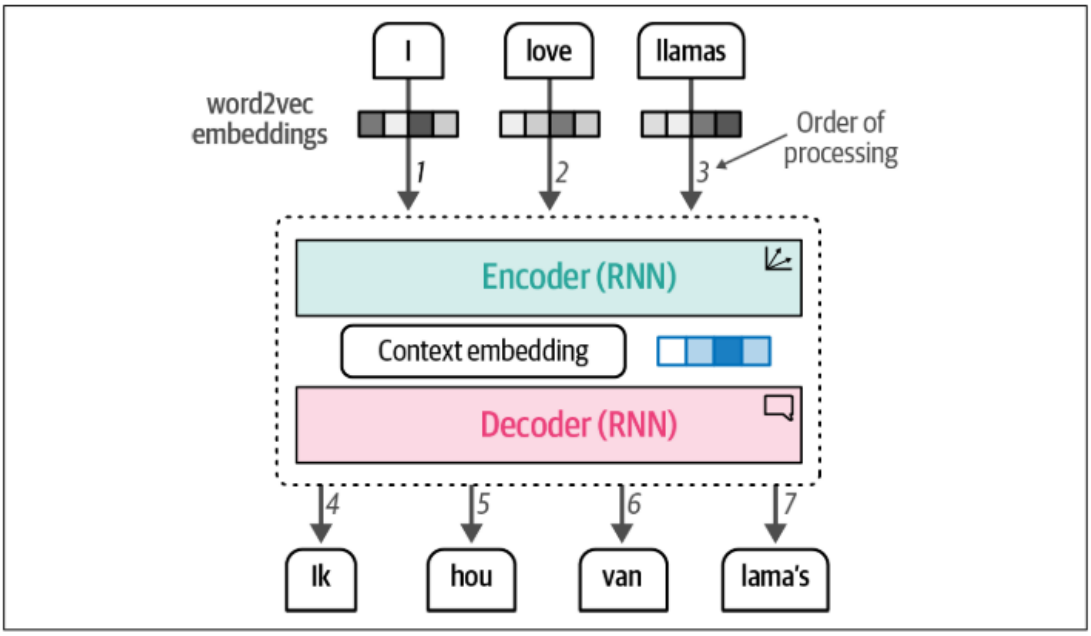

Figure 1-13. Using word2vec embeddings, a context embedding is generated that represents the entire sequence.

然而，这种上下文嵌入仅用单个向量表征整个输入序列，导致长句处理效果受限，会出现语义信息的损失。2014 年提出的注意力机制有效改善了原架构的缺陷。如图 1-14 所示，该机制使模型能聚焦于输入序列中相互关联的部分，并强化其特征信号。注意力可选择性判定句子中最重要的词汇。

例如：输出词"lama's"是荷兰语中"llamas"的对应词，因此二者间的注意力权重较高；反之"lama's"与"I"关联度较低，故注意力权重较弱。我们将在第三章深

入探讨注意力机制的工作原理。

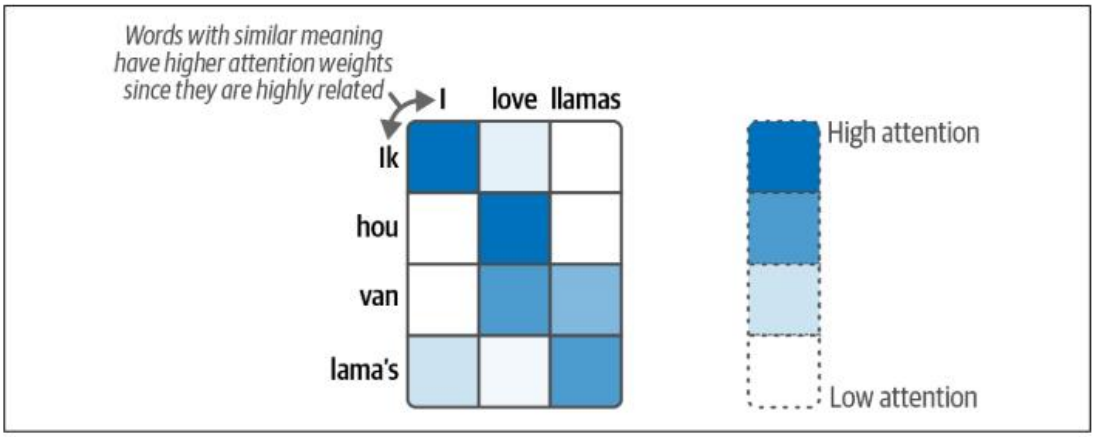

Figure 1-14. Attention allows a model to "attend" to certain parts of sequences that might relate more or less to one another.

通过将注意力机制添加到解码器步骤，循环神经网络可以为输入序列中的每次单词和潜在输出建立联系。在生成输出时，关注的不仅仅是编码器生成的嵌入向量，还要关注输入序列中与当前输出最相关的那个单词。如图 1-15 所示，在生成“Ik”、“Hou”和“van”之后，解码器的注意力机制使其能够在生成荷兰语翻译(“lama’ s”)之前专注于单词“llamas”。也就是说，循环神经网络会主要关注它正在翻译的单词。与 Word2Vec 相比，这种架构通过"关注"整个句子，能够有效表征文本的序列特性及其出现的上下文环境。然而，这种顺序处理特性也导致模型在训练过程中无法实现并行化计算。

5. Attention is all you need. Transformer 模型的介绍

在 2017 年发表的论文《Attention is all you need》开发了注意力机制的潜在力量，并驱使大语言模型具有惊人的能力。作者提出了一种称为 Transformer 的网络结构，它完全专注于注意力机制，并消除了我们之前看到的循环神经网络。Transformer 可以并行训练，充分发挥了 GPU 的架构优势，极大地加快了训练速度。在 Transformer 中，编码器和解码器组件彼此堆叠在一起，如图 1-16 所示。

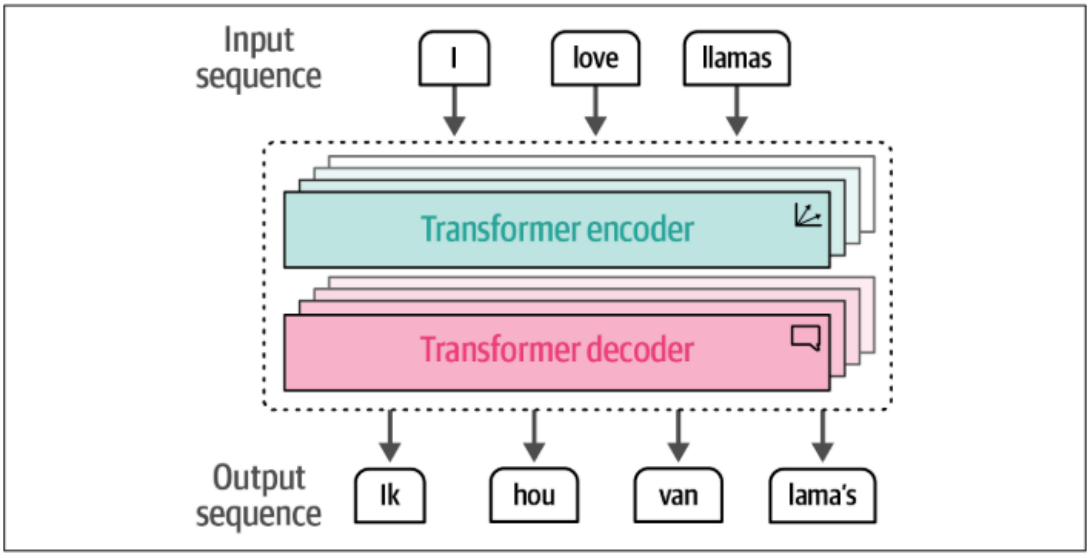

Figure 1-16. The Transformer is a combination of stacked encoder and decoder blocks where the input flows through each encoder and decoder.

现在，编码器和解码器块都将围绕注意力，而不是利用具有注意力特征的 RNN。变压器中的编码块由两部分组成，自我注意和前馈神经网络，如图 1-17 所示。

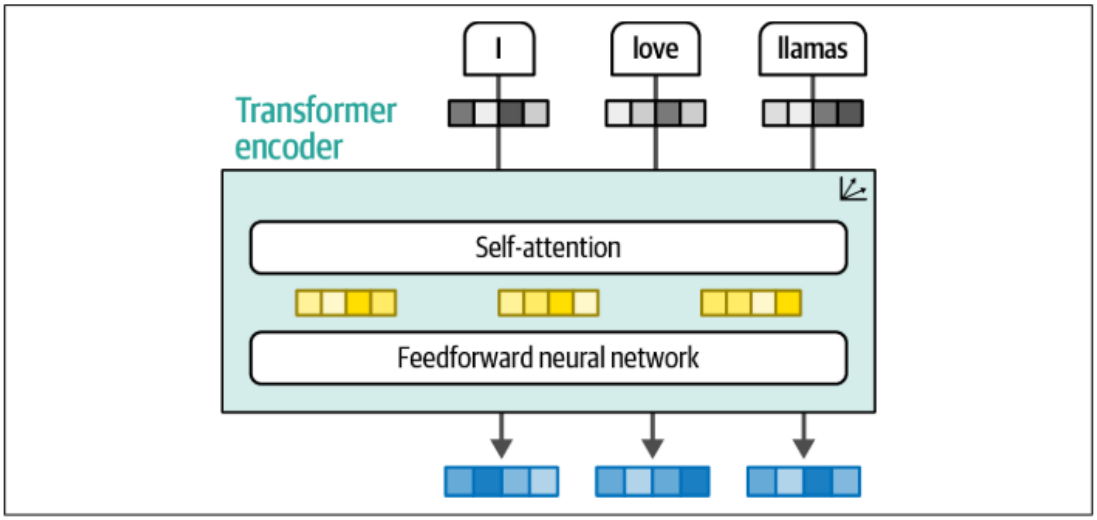

Figure 1-17. An encoder block revolves around self-attention to generate intermediate representations.

与之前的注意力方法相比，self-attention 自注意力机制可以注意到单个序列中的所有位置，可以同时计算每两个 token 之间的联系从而分配不同的注意力权重，所以可以发挥 GPU 的架构优势。

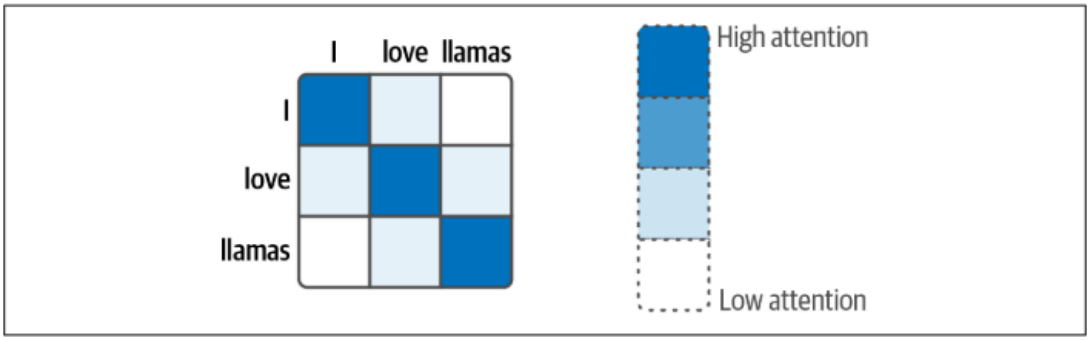

Figure 1-18. Self-attention attends to all parts of the input sequence so that it can "look" both forward and back in a single sequence.

与编码器相比，解码器有一个额外的层用来注意编码器输出的嵌入向量，如图 1-19 所示，这一过程类似于之前讨论的循环神经网络。

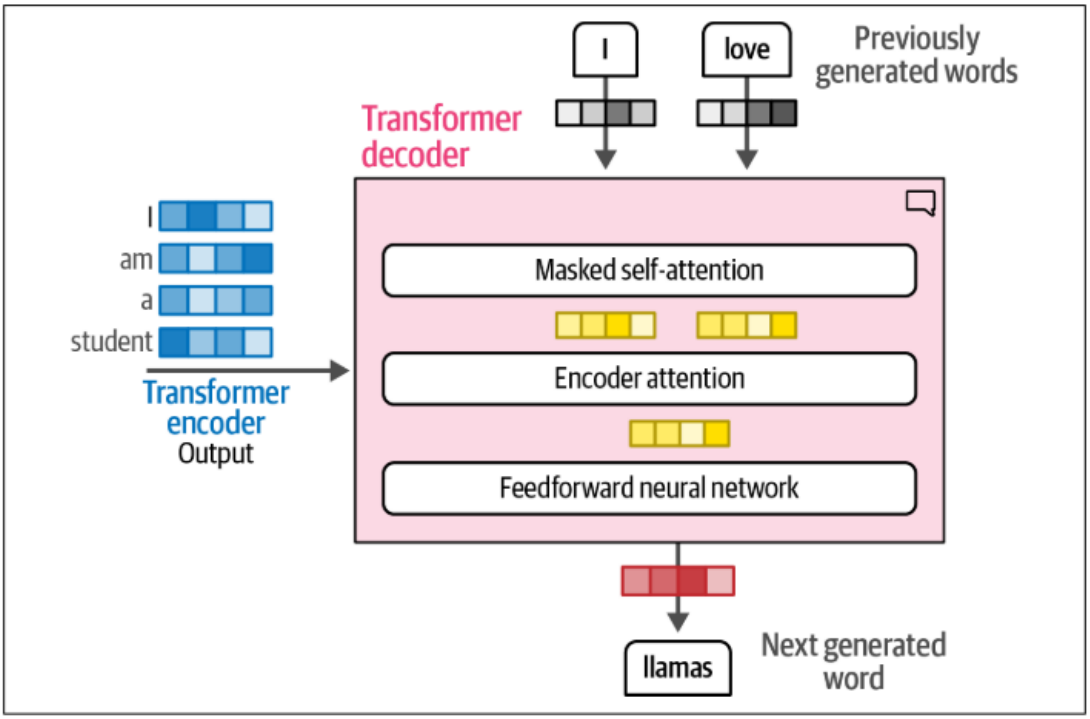

Figure 1-19. The decoder has an additional attention layer that attends to the output of the encoder.

如图 1-20 所示，解码器中的 self-attention layer 通过掩码掩盖了将要输出的单词，所以它只会关注已经生成的单词，以防止在生成输出时泄漏信息（因为 Transformer 是一个**自回归生成模型**。掩码是为了强制模型学习“给定上文，预测下一个词”的正确规则，从而保证训练过程与**推理过程**的一致性，并防止模型在训练中通过偷看未来答案而“作弊”）。

这些编码器和解码器块共同构建了 Transformer 体系结构，并且是语言 AI 中许多有影响力的模型的基础，例如 BERT 和 GPT-1，我们将在本章后面介绍它们。

在整本书中，我们将使用的大多数模型都是基于 Transformer 的模型。

Transformer 的内容复杂繁多，在第二和第三章中，我们将介绍为什么 Transformer 模型工作得这么好的许多原因，包括多头关注（multi-head attention）、位置嵌入（positional embeddings）和层标准化（layer normalization）。

B 站上有一个视频可视化地讲解了 Transformer 的工作原理，可以去看看：https://www.bilibili.com/video/BV13z421U7cs/?spm_id_from=333.337.search-card.all.click&vd_source=e7657df3faf382d3b839f7f9d7da45cb

6. 表示模型：一种只有编码器的模型

最初的 Transformer 模型是一个编码器-解码器架构，它能很好地完成翻译任务，但不能像翻译任务一样出色地完成其他任务，如文本分类。2018 年，推出了一种来自 Transformers 的双向编码器表示法(BERT)的新体系结构，该体系结构可用于各种任务，并将在未来几年用作语言人工智能的基础。BERT 是一种只有编码器的结构，专注于表示语言，如图 1-21 所示。这意味着它只使用编码器，完全去除了解码器。

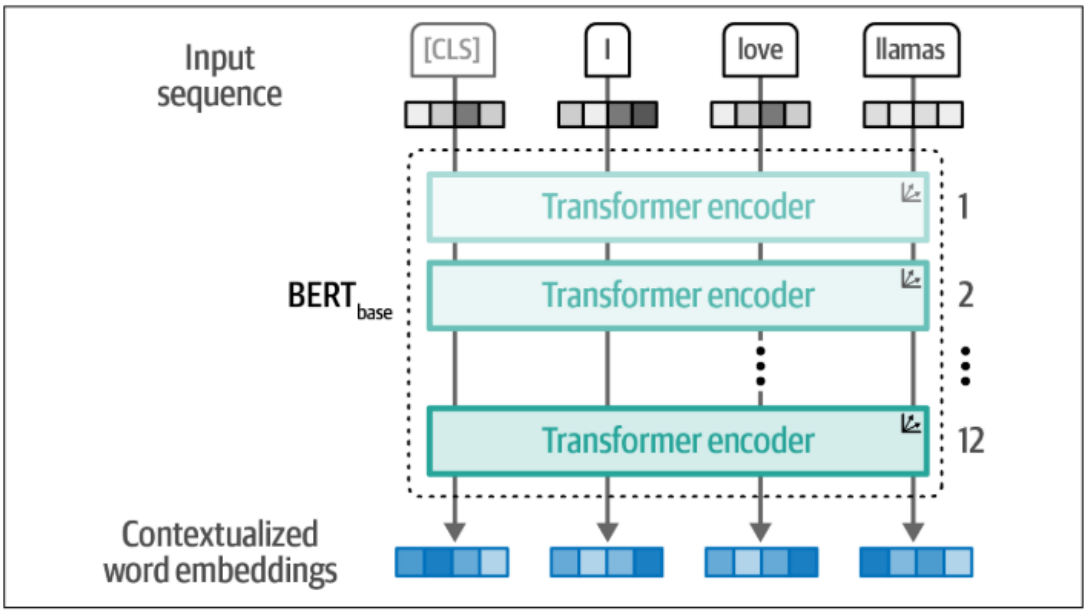

Figure 1-21. The architecture of a BERT base model with 12 encoders.

这些编码器模块与我们之前所见相同：自注意力机制后接前馈神经网络。输入中包含一个特殊标记——[CLS]或分类标记，该标记作为整个输入的表示向量。通常，我们在特定任务（如分类任务）上微调模型时，会使用这个[CLS]标记作为输入嵌入表示。

BERT 的训练过程：采用掩码语言建模技术（参见第 2 章和第 11 章）来进行训练，如图 1-22 所示，该方法通过掩码部分输入内容让模型进行预测。这种预测任务虽然困难，但能使 BERT 生成更精准的输入（中间）表征。

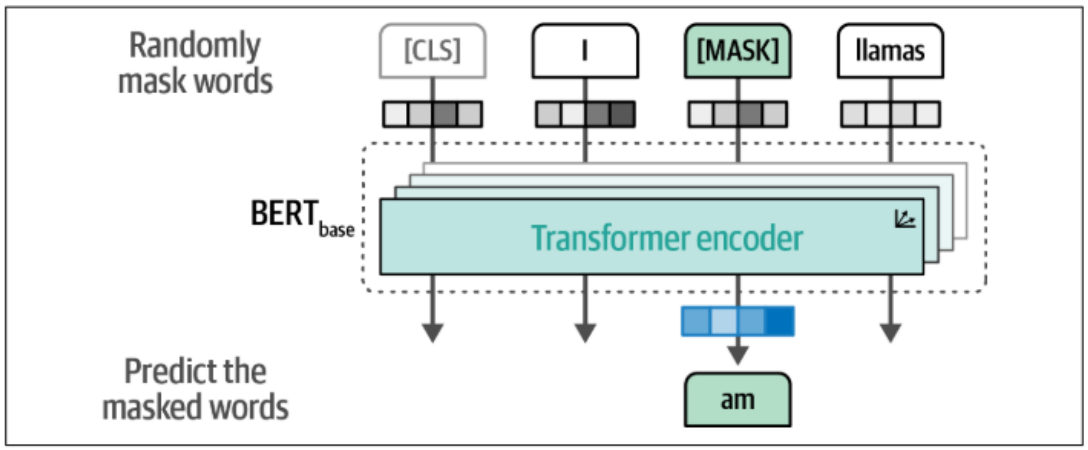

Figure 1-22. Train a BERT model by using masked language modeling.

这种架构与训练机制使 BERT 及相关模型在语境化语言表征方面表现出色。BERT 类模型通常用于迁移学习：首先进行语言建模预训练，随后针对特定任务进行微调。例如，通过在维基百科全集上训练 BERT，模型能学会理解文本的语义和上下文特性。如图 1-23 所示，我们可以利用这个预训练模型针对文本分类等具体任务进行微调。

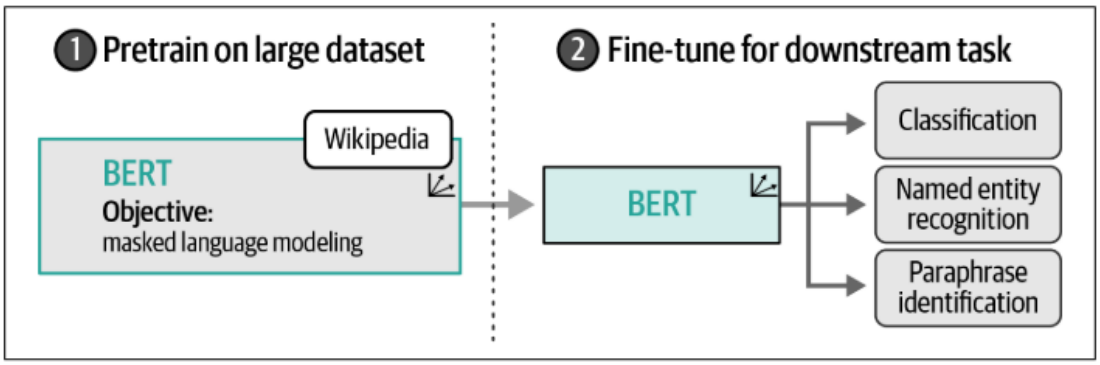

Figure 1-23. After pretraining BERT on masked language model, we fine-tune it for specific tasks.

BERT 的训练过程是一个经典的**两阶段范式**：

① **预训练**：通过**掩码语言模型**学习词汇的深层上下文表示，通过**下一句预测**学习句子间逻辑关系。这是在无标注数据上进行的自监督学习。

② **微调**：将预训练模型作为特征提取器，在特定任务的少量标注数据上，对模型所有参数进行轻微调整，使其适应最终任务。

这种“预训练+微调”的模式极大地降低了自然语言处理任务对大规模标注数据的依赖，成为了现代深度学习的基础范式。

只有编码器的模型，如 BERT，将在本书的许多部分使用。多年来，它们一直并仍然用于常见任务，包括分类任务(见第 4 章)、聚类任务(见第 5 章)和语义搜索(见第 8 章)。

在本书中，我们将仅含编码器的模型称为表示模型，以区别于仅含解码器的生成模型（这种说法不完全正确，因为“表示”和“生成”是功能性的描述，而“只有编码器/解码器”是架构性的描述，我们不能永远将功能与架构一一对应死。）。需要注意的是，主要区别并不在于底层架构及其工作原理，而在于模型所专注的功能——表示模型主要专注于语言表征（例如通过创建嵌入向量），通常不生成文本；而生成模型主要专注于文本生成，通常不训练生成嵌入向量。

7．生成模型：一种只有解码器的模型

与 BERT 的仅编码器架构类似，2018 年研究者提出了一种仅解码器架构以针对生成式任务。这种架构因其生成能力被称为生成式预训练变换器（GPT）（现为区分后续版本通常称为 GPT-1）。如图 1-24 所示，该架构堆叠了多个解码器块，其堆叠方式与 BERT 的编码器堆叠架构相似。

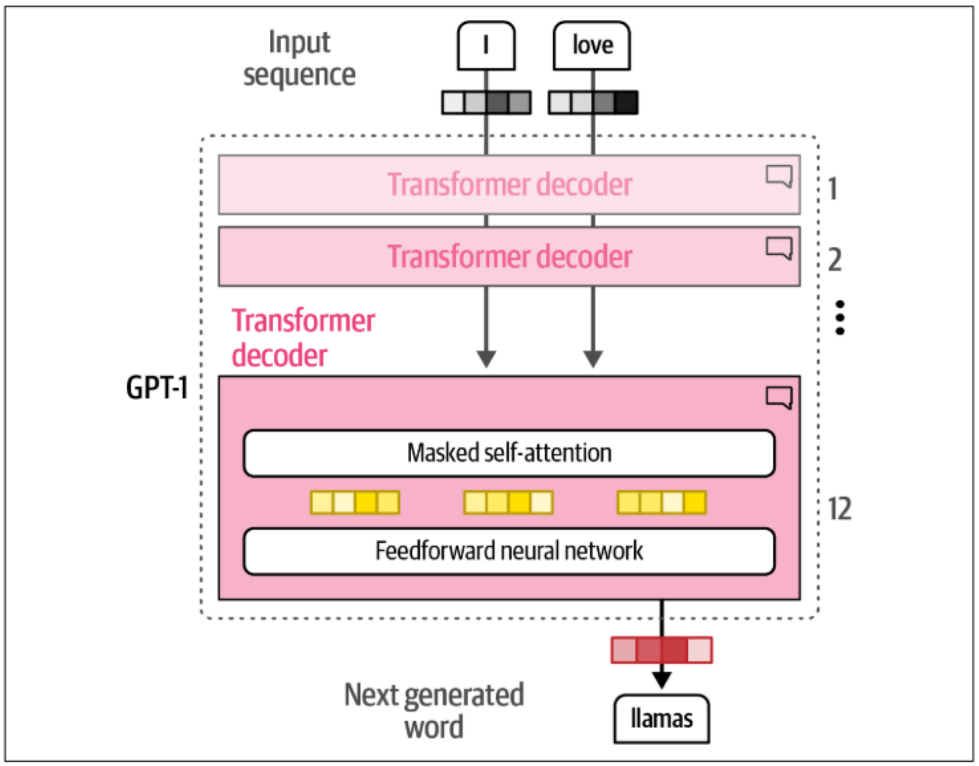

Figure 1-24. The architecture of a GPT-1. It uses a decoder-only architecture and removes the encoder-attention block.

GPT-1 在 7000 本书籍和大型网页数据集 Common Crawl 上进行了训练。最终模型包含 1.17 亿个参数，每个参数都是代表模型语言理解能力的数值。

若其他条件保持不变，参数量的增加预计会显著影响语言模型的能力与性能。基于这一认知，我们看到更大规模的模型持续被发布：如图 1-25 所示，GPT-2 达到 15 亿参数，GPT-3 更快速跟进至 1750 亿参数。

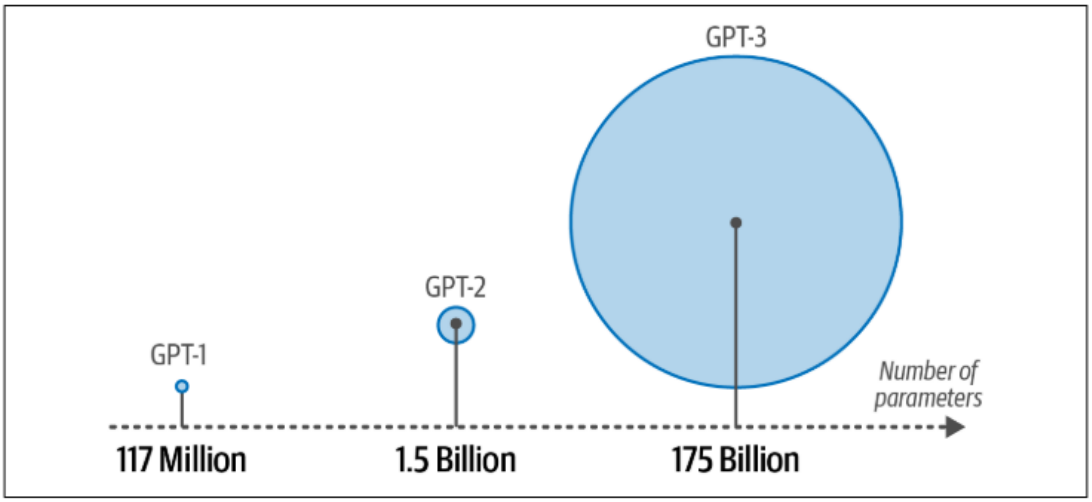

Figure 1-25. GPT models quickly grew in size with each iteration.

这类生成式的仅解码器模型（尤其是"规模更大"的模型）通常被称为大语言模型（Large language model，LLM）。正如我们将在本章后续讨论的，LLM 这一术语不仅适用于生成模型（仅解码器），也适用于表示模型（仅编码器）。

生成式大语言模型作为序列到序列的机器学习系统，能够接收文本输入并尝试自动补全内容。虽然自动补全功能已颇具实用性，但生成模型的真正潜力在被训练为聊天机器人时才得以充分展现——不局限于让模型补全文本，而是训练它们回答问题。通过对这些模型进行微调，我们可以创建能够遵循指令的指导模型或对话模型（见第 11 章）。

如图 1-26 所示，经过调优的模型可以接收用户查询（提示）并输出最符合该提示要求的响应。

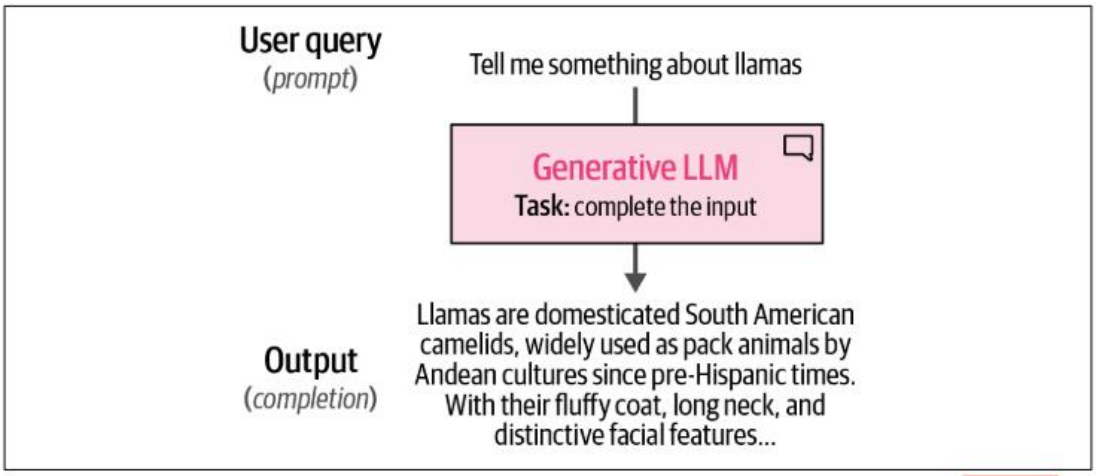

Figure 1-26. Generative LLMs take in some input and try to complete it. With instruct models, this is more than just autocomplete and attempts to answer the question.

这些补全模型的一个关键特性是上下文长度（context length）或上下文窗口（context window）。如图 1-27 所示，上下文长度代表模型能处理的最大标记（token）数量。较大的上下文窗口允许将完整文档传递给大语言模型。需要注意的是，由于这类模型具有自回归特性，当前上下文长度会随着新标记的生成而持续增加。

8. 生成式人工智能之年

LLM 对该领域产生了巨大影响，随着 ChatGPT（GPT-3.5）的发布、普及和媒体广泛报道，2023 年甚至被称为"生成式 AI 元年"。需要说明的是，当我们提及 ChatGPT 时，实际指的是该产品而非底层模型。其最初发布时基于 GPT-3.5 大语言模型，后续已扩展包含多个性能更强的版本，例如 GPT-4 与 5。

在生成式 AI 元年引发轰动的并非只有 GPT-3.5。如图 1-28 所示，开源和闭源大语言模型都以惊人速度涌入大众视野。这些开源基础模型通常被称为基石模型（foundation models），可通过微调适应特定任务（如指令遵循）。

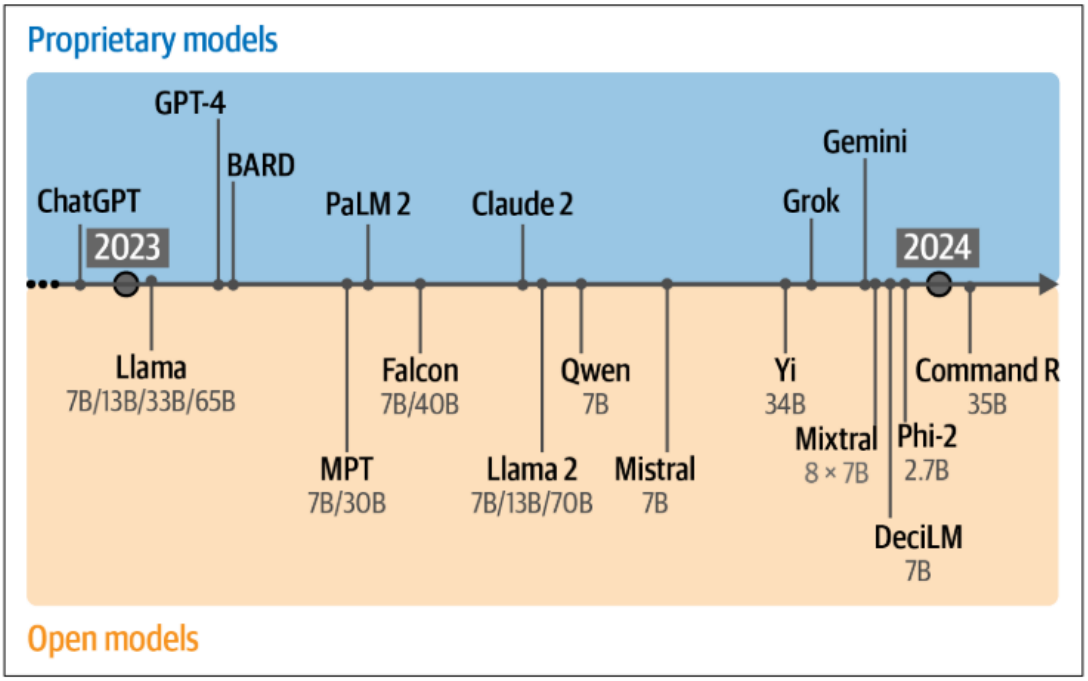

Figure 1-28. A comprehensive view into the Year of Generative AI. Note that many models are still missing from this overview!

除了广泛流行的 Transformer 架构之外，新兴架构也崭露头角，如 Mamba 和 RWKV。这些创新架构在追求达到 Transformer 级别性能的同时，还具备更大上下文窗口或更快推理速度等优势。

三. “大语言模型”的动态定义

在我们探索语言人工智能近期发展的历程中，注意到通常只有生成式仅解码器（Transformer）模型被称为大语言模型，尤其是当它们被认为"规模庞大"时。但实际上，这种定义似乎过于局限。

如果我们创建一个能力与 GPT-3 相当但体积小十倍的模型，这样的模型是否就不属于"大"语言模型范畴？同样地，当其主要功能不是语言生成而只是文本表征时，它还能被称为"大语言模型"吗？

这类定义的问题在于排除了许多有能力的模型。事实上，命名方式并不会改变模型的实际能力。

鉴于"大语言模型"的定义往往随着新模型发布而演变，我们需要明确本书的界定标准："大"本身具有主观性，今日所谓的大模型未来可能被视为小模型。当前同类模型存在多种命名方式，对我们而言，"大语言模型"也包含不生成文本且可在消费级硬件上运行的模型。

因此，除涵盖生成式模型外，本书还将探讨参数量少于 10 亿的非文本生成模型。我们将研究如何运用嵌入模型、表示模型甚至词袋模型等其他类型模型来增强大语言模型的能力。

四． 大语言模型的训练范式

传统机器学习通常针对特定任务（如分类）训练模型。如图 1-29 所示，我们将这视为单步流程。

Figure 1-29. Traditional machine learning involves a single step: training a model for a specific target task, like classification or regression.

相比之下，创建 LLM 通常由至少两个步骤组成：

① 语言建模

第一阶段称为预训练（pretraining），需要消耗大部分计算资源和训练时间。大语言模型通过在互联网海量文本上进行训练，**学习语法、上下文和语言模式**。这种广泛训练阶段尚未针对特定任务或应用，仅专注于预测下一个词。由此产生的模型通常称为基石模型（foundation model）或基础模型（base model），这类模型通常不具备指令遵循能力。

② 微调

第二阶段微调（fine-tuning）（有时称为后训练 post-training）涉及使用预训练模型，针对更具体的任务进行进一步训练。这使得大语言模型能够适应特定任务或展现预期行为。例如，我们可以对基础模型进行微调，使其在分类任务或指令遵循方面表现优异。这种方式能节省大量资源，因为预训练阶段成本极高，通常需要超出大多数个人和组织承受能力的数据和计算资源，例如 Llama 2 的训练数据集就包含 2 万亿个标记。在第十二章中，我们将详细探讨在自有数据集上微调基础模型的多种方法。

现在我们不再为每个任务从零训练一个新模型，而是**几乎总是**先有一个预训练模型，然后在此基础上进行微调。

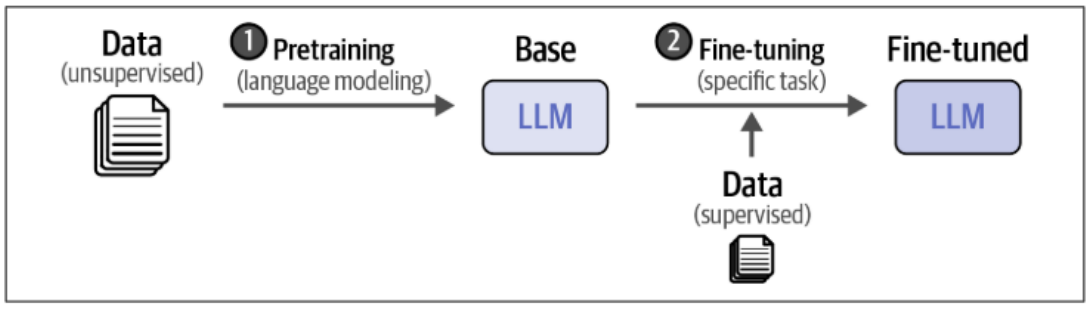

Figure 1-30. Compared to traditional machine learning, LLM training takes a multistep approach.

五． 大语言模型的应用：是什么让它们如此有用

LLM 适用于很多任务，下面是一些常见的任务和技术：

1. 检测客户留下的评论是正面还是负面评论

这是一个（有监督）分类任务，可使用仅编码器或仅解码器模型处理，既可通过预训练模型实现（参见第 4 章），也可通过微调模型完成（参见第 11 章）。

2. 开发用于主题聚类和归纳的系统

这是一个（无监督）分类任务，没有预定义的标签。我们可以利用仅编码器模型执行分类本身，再使用仅解码器模型为主题生成标签（参见第 5 章）。

3. 构建用于检索和查阅相关文档的系统

语言模型系统的核心能力之一是能够整合外部信息资源。通过语义搜索，我们可以构建能够轻松获取信息供大语言模型使用的系统（参见第 8 章）。通过创建或微调自定义嵌入模型来提升系统性能（参见第 12 章）。

4. 构建能够利用外部资源（如工具和文档）的大语言模型聊天机器人

这是多种技术的结合，通过添加外部组件才能真正释放大语言模型的潜力。提示工程（参见第 6 章）、检索增强生成（参见第 8 章）以及对大语言模型进行微调（参见第 12 章）等方法，都是构建完整大语言模型应用的关键组成部分。

5. 图像识别

这是一个多模态任务，模型需要接收图像并对其内容进行推理（参见第 9 章）。当前大语言模型正不断适配视觉等其他模态，这为各种有趣的应用场景开辟了广阔空间。

六． 大语言模型的社会与伦理考量

大语言模型因其广泛采用已产生并将持续产生重大影响。在探索其惊人能力的同时，必须充分考虑其社会与伦理影响。主要关注点包括：

1.偏见与公平性

模型训练数据可能包含社会偏见，LLM 会学习并复制这些偏见，甚至放大它们。由于训练数据通常不公开，除非实际测试，否则难以察觉其中潜藏的偏见。

2.透明度与问责制

LLM 能力强大导致人机交互边界模糊。当系统缺少人类监督时，使用 LLM 与人互动可能产生意外后果——例如医疗领域的 LLM 应用可能因影响患者健康被列为医疗设备监管。

3.生成有害内容

LLM 生成的内容未必真实，更可能被用于制造假新闻和误导性信息。

4.知识产权

LLM 输出内容的知识产权归属存在争议：当生成内容与训练数据中的受版权保护材料相似时，由于训练数据不公开，很难判断是否侵权。

5.监管态势

鉴于 LLM 的巨大影响，各国政府正开始加强监管（如欧盟《人工智能法案》），对包括 LLM 在内的基础模型进行开发和部署规范。

在开发和使用 LLM 时，**必须重视伦理考量**，并呼吁深入学习和实践安全负责任地使用 LLM 及 AI 系统。

七． 大语言模型所需的计算资源

本书中多次提及的计算资源，通常指电脑的 GPU（图形处理器）。强大的 GPU 能显著提升大语言模型训练和使用的效率与速度。

选择 GPU 时，显存容量（VRAM）是关键指标——即 GPU 自带的内存容量。实践中显存越大越好，因为若显存不足，某些模型将完全无法运行。

LLM 的训练和微调过程极其消耗 GPU 资源,例如 Meta 创建 Llama 2 系列模型时，使用了 A100-80GB 显卡。假设租赁此类显卡的成本为每小时 1.5 美元，其总训练成本将超过 500 万美元。遗憾的是，确定具体模型所需显存没有统一标准，这取决于模型架构、参数量、压缩技术、上下文长度及运行后端等因素。

本书将使用无需顶级 GPU 或巨额预算即可运行的模型。为此，本书的所有代码都兼容 Google Colab 平台，可以在 Colab 平台加载运行模型。因为免费版 Colab 可提供配备 16GB 显存的 T4 显卡，可以满足本书中提及的所有模型所需的 GPU 资源。

八. 与大语言模型对接

与大型语言模型（LLM）的交互不仅是使用它们的关键环节，也是理解其内部运作机制的重要组成部分。由于该领域的快速发展，出现了大量与大型语言模型通信的技术、方法和工具包。在本书中，我们将重点探讨最常用的交互技术，包括使用专有（闭源）模型和公开可用的开源模型。

1. 闭源模型

闭源大语言模型是指不公开其权重参数与架构设计的模型。这类模型由特定组织开发，其底层代码处于保密状态，例如 OpenAI 的 GPT-4 和 Anthropic 的 Claude 系列。这些专有模型通常拥有强大的商业支持，并被深度集成于开发者的服务生态中。

用户可通过应用程序编程接口（API）与这些模型进行交互，如图 1-31 所示。例如若要在 Python 中使用 ChatGPT，可以通过 OpenAI 提供的软件包与服务对接，而无需直接访问模型底层。

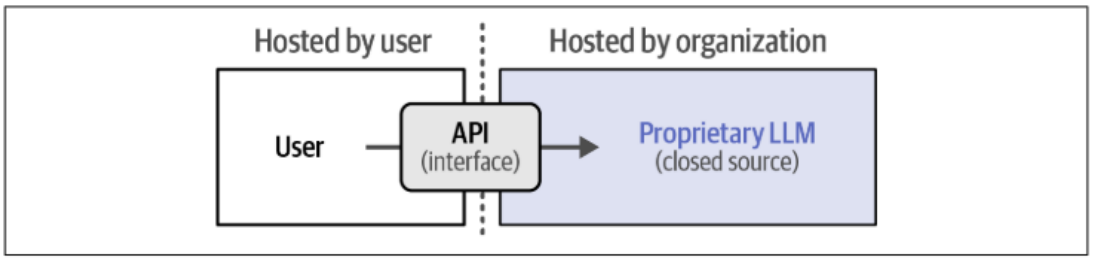

Figure 1-31. Closed source LLMs are accessed by an interface (API). As a result, details of the LLM itself, including its code and architecture are not shared with the user.

闭源模型的一大优势在于用户无需配备高性能 GPU 即可使用大语言模型，服务提供商会负责模型的部署与运行。用户不需要具备模型部署和运维的专业知识，这显著降低了使用门槛。此外，由于开发组织投入巨大，这类模型性能往往优于开源版本。

但这种模式的缺点在于成本较高。提供商需要承担模型托管的风险与成本，这通

常转化为付费服务。更关键的是，由于无法直接访问模型，用户不能自行进行微调。最后，用户数据需要与提供商共享——这在医疗数据共享等许多常见应用场景中是不可接受的。

2. 开源模型

开源大语言模型是指向公众开放权重参数与架构设计的模型。这类模型虽由特定组织开发，但通常会公开其本地创建或运行模型的代码（采用不同许可协议，部分可能限制商业用途）。Cohere 的 Command R、Mistral 系列模型、微软的 Phi 以及 Meta 的 Llama 系列都是典型代表。

如图 1-32 所示，只要配备能运行此类模型的高性能 GPU，用户即可下载并在本地设备使用这些模型。

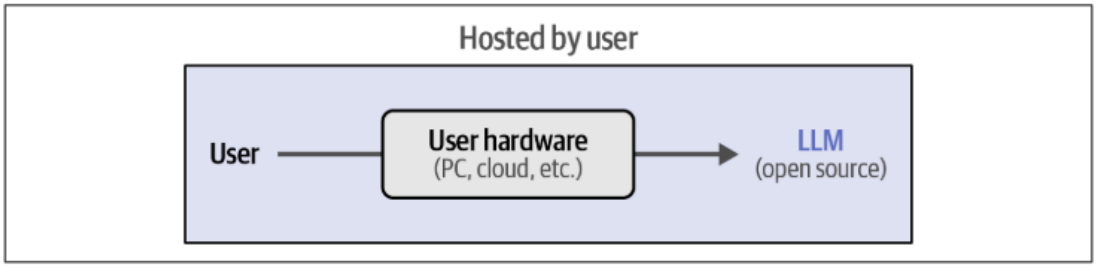

Figure 1-32. Open source LLMs are directly by the user. As a result, details of the LLM itself including its code and architecture are shared with the user.

本地部署的核心优势在于用户拥有完全自主控制权：无需依赖 API 连接即可使用模型、可自由进行微调、能处理敏感数据。用户既不依赖任何外部服务，又能完全透明地掌握模型输出的生成过程。这种优势在 Hugging Face 等大型社区的支持下进一步增强，展现出开源协作的巨大潜力。

其劣势在于需要高性能硬件支撑模型运行，训练和微调时对算力要求更高。此外，部署使用这些模型需要专业技术知识。我们通常更倾向于使用开源模型——这种能够自由调试参数、探索内部机制、实现本地化部署的开放性，相比闭源模型显然能带来更多收益。

注意，有些开源模型不能用于商业，有些用于训练模型的数据及其源代码也很少共享，很多人认为这并不是真正的开源。

3. 开源架构

与闭源大语言模型相比，开源模型需要依赖特定软件包才能运行。2023 年涌现出大量各具特色的软件包和框架，分别以不同方式与大语言模型交互协作。

我们不再试图覆盖所有现有 LLM 框架（数量过多且持续增长），而是致力于为您提供运用大语言模型的**坚实基础**。本书的目标是让您在阅读后能轻松掌握大多数其他框架，因为它们的工作原理都高度相似。

具体而言，我们将聚焦**后端软件包**：这些无图形界面（GUI）的工具包专为在本地设备高效加载和运行 LLM 而设计，例如 llama.cpp、LangChain 以及众多框架的核心基础——Hugging Face Transformers（transformers/i18n/README_zh-hans.md at main · huggingface/transformers）。

九. 生成你的第一段文本

使用语言模型的关键环节在于模型选择。当前寻找和下载大语言模型的主要平台是 **Hugging Face Hub**——该组织正是著名 Transformers 软件包的背后推手，多年来持续推动着语言模型的整体发展。正如其名，这个软件包建立在本书第 5 页"语言 AI 近代史"中讨论的 Transformer 架构之上。

截至撰稿时，Hugging Face 平台已收录超过 80 万个适用于不同任务的模型：从大语言模型、计算机视觉模型到处理音频和表格数据的模型应有尽有。在这里几乎可以找到所有开源大语言模型。

首先我们从生成式模型开始编写第一行代码。全书主要使用的生成模型是 **Phi-3-mini**——这是一个参数量相对较小（38 亿）但性能出色的模型，该模型可在显存低于 8GB 的设备上运行。若采用量化技术（我们将在第七和十二章详细讨论这种压缩方法），甚至只需不到 6GB 显存。该模型采用 MIT 许可证，允许无限制商业使用。

当使用大语言模型时，需要加载两个组件：生成模型本身和它的分词器（tokenizer）。

分词器负责将输入文本切分为 tokens，再传递给生成式模型。您可以在 Hugging Face 平台找到对应的分词器和模型，只需传入相应的模型 ID 即可。本例中，我们使用"microsoft/Phi-3-mini-4k-instruct"作为模型的主路径。

通过 transformers 库可同时加载分词器和模型。请注意此处默认您拥有 NVIDIA GPU（device_map="cuda"），但也可选择其他设备运行。若没有 GPU，可使用本书代码库中提供的免费 Google Colab，Colab 中提供了 T4 GPU，可以满足本书的

需求。原书的代码库 https://github.com/HandsOnLLM/Hands-On-Large-Language-Models

Colab 默认加速器为 CPU，需要按照如下步骤修改为 T4 GPU。

点击“保存”。

首先安装所需的软件包，可能会有版本兼容问题，可以自己更新。

```
#静默输出
%%capture
!pip install transformers>=4.40.1 accelerate>=0.27.2 #安装所需的软件包
```

Phi-3

The first step is to load our model onto the GPU for faster inference. Note that we load the model and tokenizer separately (although that isn't always necessary).

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
        "microsoft/Phi-3-mini-4k-instruct", #选择模型ID
        device_map="cuda", # 自动将模型加载到GPU
        torch_dtype="auto", # 自动选择最佳数值精度
        trust_remote_code=False, # 禁用非官方代码执行（安全设置）
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct") #下载模型对应的分词器
```

```
config.json: 100% 967/967 [00:00<00:00, 25.6kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json: 16.5k/? [00:00<00:00, 487kB/s]
Fetching 2 files: 100% 2/2 [04:11<00:00, 251.74s/it]
model-00002-of-00002.safetensors: 100% 2.67G/2.67G [03:07<00:00, 4.38MB/s]
model-00001-of-00002.safetensors: 100% 4.97G/4.97G [04:11<00:00, 28.0MB/s]
Loading checkpoint shards: 100% 2/2 [00:31<00:00, 14.78s/it]
generation_config.json: 100% 181/181 [00:00<00:00, 11.7kB/s]
tokenizer_config.json: 3.44k/? [00:00<00:00, 158kB/s]
tokenizer.model: 100% 500k/500k [00:00<00:00, 1.17MB/s]
tokenizer.json: 1.94M/? [00:00<00:00, 24.3MB/s]
added_tokens.json: 100% 306/306 [00:00<00:00, 40.4kB/s]
special_tokens_map.json: 100% 599/599 [00:00<00:00, 66.1kB/s]
```

尽管我们现在有足够的资源来开始生成文本，但在 transformer 中有一个很好的技巧来简化这个过程，即 transformer.pipeline。它将模型、分词器和文本生成过程封装到单个函数中：

Although we can now use the model and tokenizer directly, it's much easier to wrap it in a `pipeline` object:

```
from transformers import pipeline

# Create a pipeline
# 使用Hugging Face的pipeline API构建文本生成流程
generator = pipeline(
        "text-generation", # 指定文本生成任务
        model=model, # 传入已加载的Phi-3模型
        tokenizer=tokenizer, # 传入对应的分词器
        return_full_text=False, # 仅返回新生成的文本（不包含输入提示）
        max_new_tokens=500, #最大生成500个Token
        do_sample=False
)
```

我们命令模型生成一个关于鸡的笑话。使用 List 的格式来创建提示，我们的角

色 role 是用户“user”，用“content”键定义我们的指令：

Finally, we create our prompt as a user and give it to the model:

```
# The prompt (user input / query)
messages = [
        {"role": "user", "content": "Create a funny joke about chickens."}
] ## 用户角色指令

# Generate output
output = generator(messages) # 将消息传入管道
print(output[0]["generated_text"]) # 提取并打印生成结果
```

```
Why did the chicken join the band? Because it had the drumsticks!
```