大模型基础

Foundations of Large Language Models

毛玉仁 高云君等 著

我的语言的界限意味着我的世界的界限。

——维特根斯坦《逻辑哲学论》

* 本书所指大模型为大语言模型。

* 本书作者分工情况如下：第一章作者为：毛玉仁、高云君；第二章作者为：李佳晖、毛玉仁、宓禹；第三章作者为：张超、毛玉仁、胡中豪；第四章作者为：葛宇航、毛玉仁；第五章作者为：宓禹、樊怡江、毛玉仁；第六章作者为：董雪梅、徐文溢、毛玉仁。高云君为本书编撰总指导。

* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

目录

1 语言模型基础

语言是一套复杂的符号系统。语言符号通常在音韵（Phonology）、词法（Morphology）、句法（Syntax）的约束下构成，并承载不同的语义（Semantics）。语言符号具有不确定性。同样的语义可以由不同的音韵、词法、句法构成的符号来表达；同样的音韵、词法、句法构成的符号也可以在不同的语境下表达不同的语义。因此，**语言是概率的**。并且，**语言的概率性与认知的概率性也存在着密不可分的关系** [15]。语言模型（Language Models, LMs）旨在**准确预测语言符号的概率**。从语言学的角度，语言模型可以赋能计算机掌握语法、理解语义，以完成自然语言处理任务。从认知科学的角度，准确预测语言符号的概率可以赋能计算机描摹认知、演化智能。从 ELIZA [20] 到 GPT-4 [16]，语言模型经历了从规则模型到统计模型，再到神经网络模型的发展历程，逐步从呆板的机械式问答程序成长为具有强大泛化能力的多任务智能模型。本章将按照语言模型发展的顺序依次讲解基于统计方法的 n-grams 语言模型、基于循环神经网络（Recurrent Neural Network，RNN）的语言模型，基于 Transformer 的语言模型。此外，本章还将介绍如何将语言模型输出概率值解码为目标文本，以及如何对语言模型的性能进行评估。

* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

1.1 基于统计方法的语言模型

语言模型通过对语料库（Corpus）中的语料进行统计或学习来获得预测语言符号概率的能力。通常，基于统计的语言模型通过直接统计语言符号在语料库中出现的频率来预测语言符号的概率。其中，n-grams 是最具代表性的统计语言模型。**n-grams 语言模型基于马尔可夫假设和离散变量的极大似然估计给出语言符号的概率**。本节首先给出 n-grams 语言模型的计算方法，然后讨论 n-grams 语言模型如何在马尔可夫假设的基础上应用离散变量极大似然估计给出语言符号出现的概率。

1.1.1 n-grams 语言模型

设包含 N 个元素的语言符号可以表示为 $w_{1:N} = \{w_1, w_2, w_3, ..., w_N\}$。$w_{1:N}$ 可以代表文本，也可以代表音频序列等载有语义信息的序列。为了便于理解，本章令语言符号 $w_{1:N}$ 代表文本，其元素 $w_i \in w_{1:N}$ 代表词，$i = 1, ..., N$。在真实语言模型中，w_i 可以是 Token 等其他形式。关于 Token 的介绍将在第三章中给出。

n-grams 语言模型中的 n-gram 指的是长度为 n 的词序列。n-grams 语言模型通过依次统计文本中的 n-gram 及其对应的 (n-1)-gram 在语料库中出现的相对频率来计算文本 $w_{1:N}$ 出现的概率。计算公式如下所示：

$$P_{n\text{-}grams}(w_{1:N}) = \prod_{i=n}^{N} \frac{C(w_{i-n+1\,:\,i})}{C(w_{i-n+1\,:\,i-1})}, \tag{1.1}$$

其中，$C(w_{i-n+1\,:\,i})$ 为词序列 $\{w_{i-n+1}, ..., w_i\}$ 在语料库中出现的次数，$C(w_{i-n+1\,:\,i-1})$ 为词序列 $\{w_{i-n+1}, ..., w_{i-1}\}$ 在语料库中出现的次数。其中，n 为变量，当 $n = 1$ 时，称之为 unigram，其不考虑文本的上下文关系。此时，分子 $C(w_{i-n+1\,:\,i}) = C(w_i)$，$C(w_i)$ 为词 w_i 在语料库中出现的次数；分母 $C(w_{i-n+1\,:\,i-1}) = C_{total}$，$C_{total}$ 为语料库中包含的词的总数。当 $n = 2$ 时，称之为 bigrams，其对前一个词进行考虑。此时，

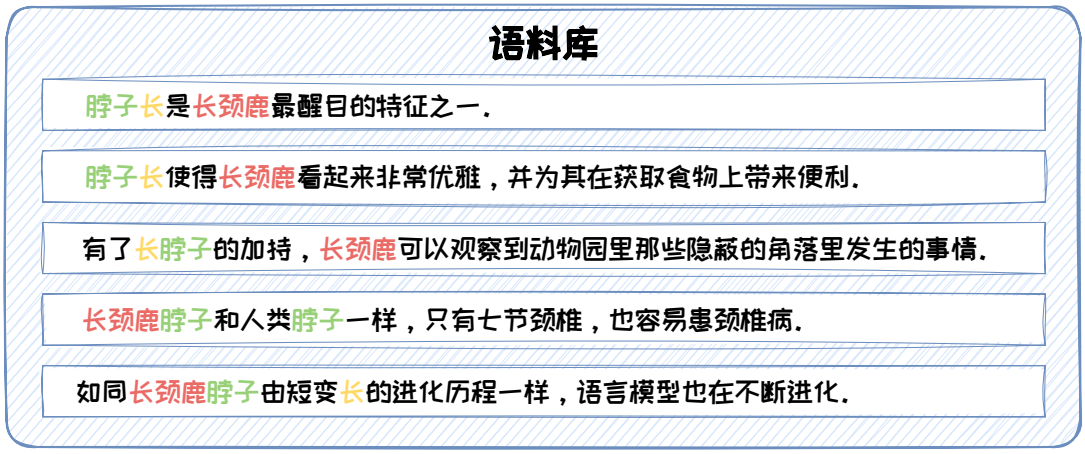


图 1.1: n-grams 示例语料库。

分子 $C(w_{i-n+1:i}) = C(w_{i-1}, w_i)$，$C(w_{i-1}, w_i)$ 为词序列 $\{w_{i-1}, w_i\}$ 在语料库中出现的次数；分母 $C(w_{i-n+1:i-1}) = C(w_{i-1})$，$C(w_{i-1})$ 为词 w_{i-1} 在语料库中出现的次数。以此类推，当 $n = 3$ 时，称之为 trigrams，其对前两个词进行考虑。当 $n = 4$ 时，称之为 4-grams，其对前三个词进行考虑.....

下面通过一个 bigrams 语言模型的例子来展示 n-grams 语言模型对文本出现概率进行计算的具体方式。假设语料库中包含 5 个句子，如图1.1所示。基于此语料库，应用 bigrams 对文本“长颈鹿脖子长”（其由 {长颈鹿, 脖子, 长} 三个词构成）出现的概率进行计算，如下式所示：

$$P_{bigrams}(\text{长颈鹿, 脖子, 长}) = \frac{C(\text{长颈鹿, 脖子})}{C(\text{长颈鹿})} \cdot \frac{C(\text{脖子, 长})}{C(\text{脖子})}。 \tag{1.2}$$

在此语料库中，$C(\text{长颈鹿}) = 5, C(\text{脖子}) = 6, C(\text{长颈鹿, 脖子}) = 2, C(\text{脖子, 长}) = 2$，故有：

$$P_{bigrams}(\text{长颈鹿, 脖子, 长}) = \frac{2}{5} \cdot \frac{2}{6} = \frac{2}{15}。 \tag{1.3}$$

在此例中，我们可以发现虽然“长颈鹿脖子长”并没有直接出现在语料库中，但是 bigrams 语言模型仍可以预测出“长颈鹿脖子长”出现的概率有 $\frac{2}{15}$。由此可见，n-grams 具备**对未知文本的泛化能力**。这也是其相较于传统基于规则的方法的

优势。但是，这种泛化能力会随着 n 的增大而逐渐减弱。应用 trigrams 对文本“长颈鹿脖子长”出现的概率进行计算，将出现以下“零概率”的情况：

$$P_{trigrams}(\text{长颈鹿, 脖子, 长}) = \frac{C(\text{长颈鹿, 脖子, 长})}{C(\text{长颈鹿, 脖子})} = 0。 \tag{1.4}$$

因此，在 n-grams 语言模型中，n 代表了拟合语料库的能力与对未知文本的泛化能力之间的权衡。当 n 过大时，语料库中难以找到与 n-gram 一模一样的词序列，可能出现大量“零概率”现象；在 n 过小时，n-gram 难以承载足够的语言信息，不足以反应语料库的特性。因此，在 n-grams 语言模型中，n 的值是影响性能的关键因素。上述的“零概率”现象可以通过平滑（Smoothing）技术进行改善，具体技术可参见文献 [11]。

本小节讲解了 n-grams 语言模型如何计算语言符号出现的概率，但没有分析 n-grams 语言模型的原理。下一小节将从 n 阶马尔可夫假设和离散型随机变量的极大似然估计的角度对 n-grams 语言模型背后的统计学原理进行阐述。

1.1.2 n-grams 的统计学原理

n-grams 语言模型是在 n **阶马尔可夫假设**下，对语料库中出现的长度为 n 的词序列出现概率的**极大似然估计**。本节首先给出 n 阶马尔可夫假设的定义（见**定义 1.1**）和离散型随机变量的极大似然估计的定义（见**定义 1.2**），然后分析 n-grams 如何在马尔可夫假设的基础上应用离散变量极大似然估计给出语言符号出现的概率。

定义 1.1 (n 阶马尔可夫假设)

对序列 $\{w_1, w_2, w_3, ..., w_N\}$，当前状态 w_N 出现的概率只与前 n 个状态 $\{w_{N-n}, ..., w_{N-1}\}$ 有关，即：

$$P(w_N|w_1, w_2, ..., w_{N-1}) \approx P(w_N|w_{N-n}, ..., w_{N-1})。 \tag{1.5}$$

定义 1.2 (离散型随机变量的极大似然估计)

给定离散型随机变量 X 的分布律为 $P\{X=x\}=p(x;\theta)$，设 $X_1,...,X_N$ 为来自 X 的样本，$x_1,...,x_N$ 为对应的观察值，θ 为待估计参数。在参数 θ 下，分布函数随机取到 $x_1,...,x_N$ 的概率为：

$$p(x|\theta)=\prod_{i=1}^{N}p(x_i;\theta)。\tag{1.6}$$

构造似然函数为：

$$L(\theta|x)=p(x|\theta)=\prod_{i=1}^{N}p(x_i;\theta)。\tag{1.7}$$

离散型随机变量的极大似然估计旨在找到 θ 使得 $L(\theta|x)$ 取最大值。 ♣

在上述两个定义的基础上，对 n-grams 的统计原理进行讨论。设文本 $w_{1:N}$ 出现的概率为 $P(w_{1:N})$。根据条件概率的链式法则，$P(w_{1:N})$ 可由下式进行计算。

$$\begin{aligned}P(w_{1:N})&=P(w_1)P(w_2|w_1)P(w_3|w_{1:2})....P(w_N|w_{1:N-1})\\&=\prod_{i=1}^{N}P(w_i|w_{1:i-1})。\end{aligned}\tag{1.8}$$

根据 n 阶马尔可夫假设，n-grams 语言模型令 $P(w_i|w_{i-n:i-1})$ 近似 $P(w_i|w_{1:i-1})$。然后，根据离散型随机变量的极大似然估计，令 $\frac{C(w_{i-n:i})}{C(w_{i-n:i-1})}$ 近似 $P(w_i|w_{i-n:i-1})$。从而，得到 n-grams 语言模型的输出 $P_{n\text{-}grams}(w_{1:N})$ 是对 $P(w_i|w_{1:i-1})$ 的近似。即，

$$P_{n\text{-}grams}(w_{1:N})\approx P(w_{1:N})。\tag{1.9}$$

下面，以 bigrams 为例，介绍 $\frac{C(w_{i-n:i})}{C(w_{i-n:i-1})}$ 与极大似然估计间的关系。假设语料库中共涵盖 M 个不同的单词，$\{w_i,w_j\}$ 出现的概率为 $P(w_i,w_j)$，对应出现的频率为 $C(w_i,w_j)$，则其出现的似然函数为：

$$L(\theta)=\prod_{i=1}^{M}\prod_{j=1}^{M}P(w_i,w_j)^{C(w_i,w_j)},\tag{1.10}$$

其中，$\theta = \{P(w_i, w_j)\}_{i,j=1}^{M}$。根据条件概率公式 $P(w_i, w_j) = P(w_j|w_i)P(w_i)$ ，有

$$L(\theta) = \prod_{i=1}^{M}\prod_{j=1}^{M} P(w_j|w_i)^{C(w_i,w_j)} P(w_i)^{C(w_i,w_j)}。 \tag{1.11}$$

其对应的对数似然函数为：

$$L_{\log}(\theta) = \sum_{i=1}^{M}\sum_{j=1}^{M} C(w_i,w_j)\log P(w_j|w_i) + \sum_{i=1}^{M}\sum_{j=1}^{M} C(w_i,w_j)\log P(w_i)。 \tag{1.12}$$

因为 $\sum_{j=1}^{M} P(w_j|w_i) = 1$，所以最大化对数似然函数可建模为如下的约束优化问题：

$$\begin{aligned} &\max \ L_{\log}(\theta) \\ &s.t. \ \sum_{j=1}^{M} P(w_j|w_i) = 1 \text{ for } i \in [1, M]。 \end{aligned} \tag{1.13}$$

其拉格朗日对偶为：

$$L(\lambda, L_{\log}) = L_{\log}(\theta) + \sum_{i=1}^{M} \lambda_i (\sum_{j=1}^{M} P(w_j|w_i) - 1)。 \tag{1.14}$$

对其求关于 $P(w_j|w_i)$ 的偏导，可得：

$$\frac{\partial L(\lambda, L_{\log})}{\partial P(w_j|w_i)} = \sum_{i=1}^{M} \frac{C(w_i, w_j)}{P(w_j|w_i)} + \sum_{i=1}^{M} \lambda_i。 \tag{1.15}$$

当导数为 0 时，有：

$$P(w_j|w_i) = -\frac{C(w_i, w_j)}{\lambda_i}。 \tag{1.16}$$

因 $\sum_{j=1}^{M} P(w_j|w_i) = 1$，$\lambda_i$ 可取值为 $-\sum_{j=1}^{M} C(w_i, w_j)$, 即

$$P(w_j|w_i) = \frac{C(w_i, w_j)}{\sum_{j=1}^{M} C(w_i, w_j)} = \frac{C(w_i, w_j)}{C(w_i)}。 \tag{1.17}$$

上述分析表明 bigram 语言模型中的 $\frac{C(w_i,w_j)}{C(w_i)}$ 是对语料库中的长度为 2 的词序列的 $P(w_j|w_i)$ 的极大似然估计。该结论可扩展到 $n > 2$ 的其他 n-grams 语言模型模型中。

n-grams 语言模型通过统计词序列在语料库中出现的频率来预测语言符号的概率。其对未知序列有一定的泛化性，但也容易陷入“零概率”的困境。随着神经网络的发展，基于各类神经网络的语言模型不断被提出，泛化能力越来越强。基于神

经网络的语言模型不再通过显性的计算公式对语言符号的概率进行计算，而是利用语料库中的样本对神经网络模型进行训练。本章接下来将分别介绍两类最具代表性的基于神经网络的语言模型：基于 RNN 的语言模型和基于 Transformer 的语言模型。

1.2 基于 RNN 的语言模型

循环神经网络（Recurrent Neural Network, RNN）是一类**网络连接中包含环路的神经网络的总称**。给定一个序列，RNN 的环路用于将历史状态叠加到当前状态上。沿着时间维度，历史状态被循环累积，并作为预测未来状态的依据。因此，RNN 可以基于历史规律，对未来进行预测。基于 RNN 的语言模型，以词序列作为输入，基于**被循环编码的上文和当前词**来预测下一个词出现的概率。本节将先对原始 RNN 的基本原理进行介绍，然后讲解如何利用 RNN 构建语言模型。

1.2.1 循环神经网络 RNN

按照推理过程中信号流转的方向，神经网络的正向传播范式可分为两大类：前馈传播范式和循环传播范式。在前馈传播范式中，计算逐层向前，“**不走回头路**”。而在循环传播范式中，某些层的计算结果会通过**环路**被反向引回前面的层中，形成“**螺旋式前进**”的范式。采用前馈传播范式的神经网络可以统称为前馈神经网络（Feed-forward Neural Network，FNN），而采用循环传播范式的神经网络被统称为循环神经网络（Recurrent Neural Network, RNN）。以包含输入层、隐藏层、输出层的神经网络为例。图1.2中展示了最简单的 FNN 和 RNN 的网络结构示意图，可见 FNN 网络结构中仅包含正向通路。而 RNN 的网络结构中除了正向通路，还有一条**环路**将某层的计算结果再次反向连接回前面的层中。

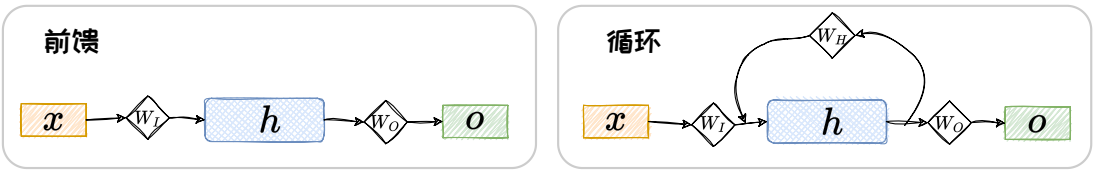


图 1.2: 前馈传播范式与循环传播范式的对比。

设输入序列为 $\{x_1, x_2, x_3, ..., x_t\}$，隐状态为 $\{h_1, h_2, h_3, ..., h_t\}$，对应输出为 $\{o_1, o_2, o_3, ..., o_t\}$，输入层、隐藏层、输出层对应的网络参数分别为 W_I, W_H, W_O。$g(\cdot)$ 为激活函数，$f(\cdot)$ 为输出函数。将输入序列一个元素接着一个元素地串行输入时，对于 FNN，当前的输出只与当前的输入有关，即

$$o_t = f(W_O g(W_I x_t))。 \tag{1.18}$$

此处为方便对比，省去了偏置项。独特的环路结构导致 RNN 与 FNN 的推理过程完全不同。RNN 在串行输入的过程中，前面的元素会被循环编码成隐状态，并叠加到当前的输入上面。其在 t 时刻的输出如下：

$$\begin{aligned} h_t &= g(W_H h_{t-1} + W_I x_t) = g(W_H g(W_H h_{t-2} + W_I x_{t-1}) + W_I x_t) = \cdots\cdots \\ o_t &= f(W_O h_t)。 \end{aligned} \tag{1.19}$$

其中，$t > 0$, $h_0 = 0$。将此过程按照时间维度展开，可得到 RNN 的推理过程，如图1.3所示。注意，图中展示的 t 时刻以前的神经元，都是过往状态留下的“虚影”并不真实存在，如此展开只是为了解释 RNN 的工作方式。

可以发现，在这样一个元素一个元素依次串行输入的设定下，RNN 可以**将历史状态以隐变量的形式循环叠加到当前状态上**，对历史信息进行考虑，**呈现出螺旋式前进**的模式。但是，缺乏环路的 FNN 仅对当前状态进行考虑，无法兼顾历史状态。以词序列 {长颈鹿, 脖子, 长} 为例，在给定“脖子”来预测下一个词是什么的时候，FFN 将仅仅考虑“脖子”来进行预测，可能预测出的下一词包含“短”，“疼”等等；而 RNN 将同时考虑“长颈鹿”和“脖子”，其预测出下一词是“长”的概率将更高，历史信息“长颈鹿”的引入，可以有效提升预测性能。

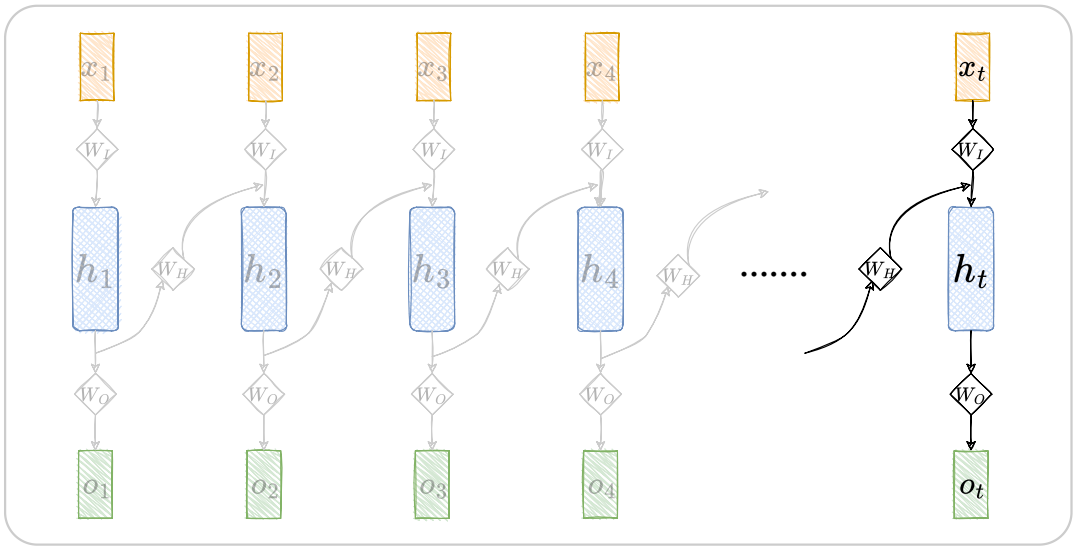


图 1.3: RNN 推理过程从时间维度拆解示意图。

如果 FNN 想要做到对历史信息进行考虑，则需要将所有元素同时输入到模型中去，这将导致模型参数量的激增。虽然，RNN 的结构可以让其在参数量不扩张的情况下实现对历史信息的考虑，但是这样的环路结构给 RNN 的训练带来了挑战。在训练 RNN 时，涉及大量的矩阵联乘操作，容易引发**梯度衰减**或**梯度爆炸**问题。具体分析如下：

设 RNN 语言模型的训练损失为:

$$L = L(x, o, W_I, W_H, W_O) = \sum_{i=1}^{t} l(o_i, y_i)。 \tag{1.20}$$

其中，$l(\cdot)$ 为损失函数，y_i 为标签。

损失 L 关于参数 W_H 的梯度为：

$$\frac{\partial L}{W_H} = \sum_{i=1}^{t} \frac{\partial l_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_i} \cdot \frac{\partial h_i}{\partial W_H}。 \tag{1.21}$$

其中,

$$\frac{\partial h_t}{\partial h_i} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{i+1}}{\partial h_i} = \prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}}。 \tag{1.22}$$

并且，

$$\frac{\partial h_k}{\partial h_{k-1}} = \frac{\partial g(z_k)}{\partial z_k} W_H。 \tag{1.23}$$

其中，$z_k = W_H h_{k-1} + W_I x_k$。综上，有

$$\frac{\partial L}{W_H} = \sum_{i=1}^{t} \frac{\partial l_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial h_t} \cdot \prod_{k=i}^{t} \frac{\partial g(z_k)}{\partial z_k} W_H \cdot \frac{\partial h_i}{\partial W_H}。 \tag{1.24}$$

从上式中可以看出，求解 W_H 的梯度时涉及大量的矩阵级联相乘。这会导致其数值被级联放大或缩小。文献 [17] 中指出，当 W_H 的最大特征值小于 1 时，会发生梯度消失；当 W_H 的最大特征值大于 1 时，会发生梯度爆炸。梯度消失和爆炸导致训练上述 RNN 非常困难。为了解决梯度消失和爆炸问题，GRU [4] 和 LSTM [8] 引入门控结构，取得了良好效果，成为主流的 RNN 网络架构。

1.2.2 基于 RNN 的语言模型

对词序列 $\{w_1, w_2, w_3, ..., w_N\}$，基于 RNN 的语言模型每次根据当前词 w_i 和循环输入的隐藏状态 h_{i-1}，来预测下一个词 w_{i+1} 出现的概率，即

$$P(w_{i+1}|w_{1:i}) = P(w_{i+1}|w_i, h_{i-1})。 \tag{1.25}$$

其中，当 $i = 1$ 时，$P(w_{i+1}|w_i, h_{i-1}) = P(w_2|w_1)$ 。基于此，$\{w_1, w_2, w_3, ..., w_N\}$ 整体出现的概率为：

$$P(w_{1:N}) = \prod_{i=1}^{N-1} P(w_{i+1}|w_i, h_{i-1})。 \tag{1.26}$$

在基于 RNN 的语言模型中，输出为一个向量，其中每一维代表着词典中对应词的概率。设词典 D 中共有 $|D|$ 个词 $\{\hat{w_1}, \hat{w_2}, \hat{w_3}, ..., \hat{w_{|D|}}\}$，基于 RNN 的语言模型的输出可表示为 $o_i = \{o_i[\hat{w_d}]\}_{d=1}^{|D|}$，其中，$o_i[\hat{w_d}]$ 表示词典中的词 $\hat{w_d}$ 出现的概率。因此，对基于 RNN 的语言模型有

$$P(w_{1:N}) = \prod_{i=1}^{N-1} P(w_{i+1}|w_{1:i}) = \prod_{i=1}^{N} o_i[w_{i+1}]。 \tag{1.27}$$

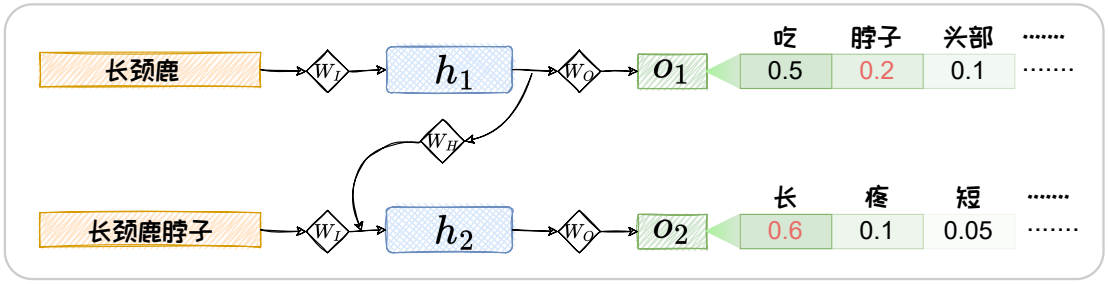


图 1.4: RNN 计算词序列概率示意图。

以下举例对上述过程进行说明。假设词表 $D = \{$脖子, 头部, 吃, 长, 疼, 吃, 短$\}$, 基于 RNN 的语言模型计算“长颈鹿脖子长”的概率的过程如图1.4所示。

$$P(\text{长颈鹿脖子长}) = P(\text{脖子}|\text{长颈鹿}) \cdot P(\text{长}|\text{脖子}, h_1) = 0.2 \times 0.6 = 0.12。 \quad (1.28)$$

基于以上预训练任务，对 RNN 语言模型进行训练时，可选用如下交叉熵函数作为损失函数。

$$l_{CE}(o_i) = -\sum_{d=1}^{|D|} I(\hat{w}_d = w_{i+1}) \log o_i[w_{i+1}] = -\log o_i[w_{i+1}], \quad (1.29)$$

其中，$I(\cdot)$ 为指示函数，当 $\hat{w}_d = w_{i+1}$ 时等于 1，当 $\hat{w}_d \neq w_{i+1}$ 时等于 0。

设训练集为 S，RNN 语言模型的损失可以构造为：

$$L(S, W_I, W_H, W_O) = \frac{1}{N|S|} \sum_{s=1}^{|S|} \sum_{i=1}^{N} l_{CE}(o_{i,s}), \quad (1.30)$$

其中，$o_{i,s}$ 为 RNN 语言模型输入第 s 个样本的第 i 个词时的输出。此处为方便表述，假设每个样本的长度都为 N。在此损失的基础上，构建计算图，进行反向传播，便可对 RNN 语言模型进行训练。上述训练过程结束之后，我们可以直接利用此模型对序列数据进行特征抽取。抽取的特征可以用于解决下游任务。此外，我们还可以对此语言模型的输出进行解码，在“**自回归**”的范式下完成文本生成任务。在自回归中，第一轮，我们首先将第一个词输入给 RNN 语言模型，经过解码，得到一个输出词。然后，我们将第一轮输出的词与第一轮输入的词拼接，作为第二轮

的输入，然后解码得到第二轮的输出。接着，将第二轮的输出和输入拼接，作为第三轮的输入，以此类推。每次将本轮预测到的词拼接到本轮的输入上，输入给语言模型，完成下一轮预测。在循环迭代的“自回归”过程中，我们不断生成新的词，这些词便构成了一段文本。

但上述“自回归”过程存在着两个问题：(1) **错误级联放大**，选用模型自己生成的词作为输入可能会有错误，这样的错误循环输入，将会不断的放大错误，导致模型不能很好拟合训练集；（2）**串行计算效率低**，因为下一个要预测的词依赖上一次的预测，每次预测之间是串行的，难以进行并行加速。为了解决上述两个问题，“**Teacher Forcing**”[21] 在语言模型预训练过程中被广泛应用。在 Teacher Forcing 中，每轮都仅将输出结果与“标准答案”（Ground Truth）进行拼接作为下一轮的输入。在图1.4所示的例子中，第二轮循环中，我们用“长颈鹿脖子”来预测下一个词“长”，而非选用 o_1 中概率最高的词“吃”或者其他可能输出的词。

但是，Teacher Forcing 的训练方式将导致**曝光偏差**（Exposure Bias）的问题。曝光偏差是指 Teacher Forcing 训练模型的过程和模型在推理过程存在差异。Teacher Forcing 在训练中，模型将依赖于“标准答案”进行下一次的预测，但是在推理预测中，模型“自回归”的产生文本，没有“标准答案”可参考。所以模型在训练过程中和推理过程中存在偏差，可能推理效果较差。为解决曝光偏差的问题，Bengio 等人提出了针对 RNN 提出了 Scheduled Sampling 方法 [2]。其在 Teacher Forcing 的训练过程中循序渐进的使用一小部分模型自己生成的词代替“标准答案”，在训练过程中对推理中无“标准答案”的情况进行预演。

由于 RNN 模型循环迭代的本质，其不易进行并行计算，导致其在输入序列较长时，训练较慢。下节将对容易并行的基于 Transformer 的语言模型进行介绍。

1.3 基于 Transformer 的语言模型

Transformer 是一类基于注意力机制（Attention）的**模块化**构建的神经网络结构。给定一个序列，Transformer 将一定数量的历史状态和当前状态同时输入，然后进行加权相加。对历史状态和当前状态进行“**通盘考虑**”，然后对未来状态进行预测。基于 Transformer 的语言模型，以词序列作为输入，基于一定长度的上文和当前词来预测下一个词出现的概率。本节将先对 Transformer 的基本原理进行介绍，然后讲解如何利用 Transformer 构建语言模型。

1.3.1 Transformer

Transformer 是由两种模块组合构建的**模块化网络结构**。两种模块分别为：（1）注意力（Attention）模块；（2）全连接前馈（Fully-connected Feedforwad）模块。其中，自注意力模块由自注意力层（Self-Attention Layer）、残差连接（Residual Connections）和层正则化（Layer Normalization）组成。全连接前馈模块由全连接前馈层，残差连接和层正则化组成。两个模块的结构示意图如图1.5所示。以下详细介绍每个层的原理及作用。

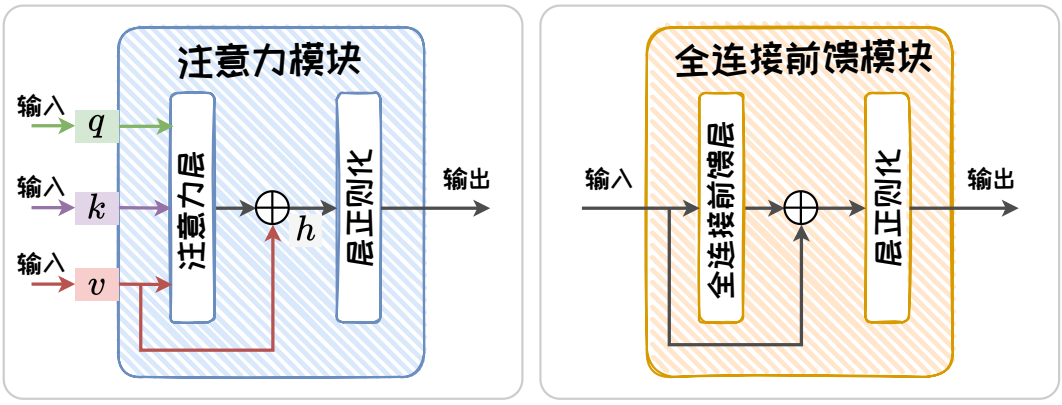


图 1.5: 注意力模块与全连接前馈模块。

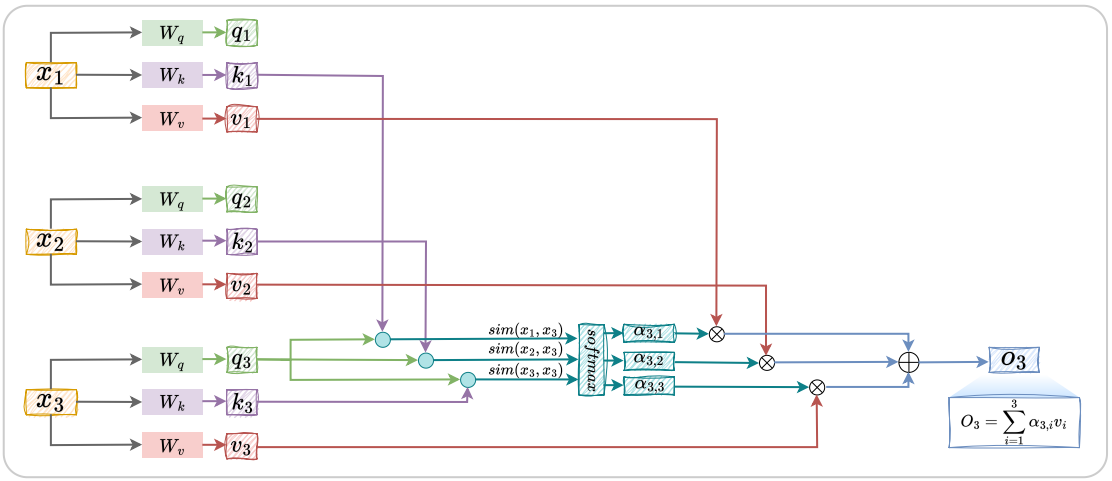


图 1.6: 注意力机制示意图。

1. 注意力层（Attention Layer）

注意力层采用加权平均的思想将前文信息叠加到当前状态上。Transformer 的注意力层将输入编码为 query，key，value 三部分，即将输入 $\{x_1, x_2, ..., x_t\}$ 编码为 $\{(q_1, k_1, v_1), (q_2, k_2, v_2), ..., (q_t, k_t, v_t)\}$。其中，query 和 key 用于计算自注意力的权重 α, value 是对输入的编码。具体的，

$$Attention(x_t) = \sum_{i=1}^{t} \alpha_{t,i} v_i 。 \tag{1.31}$$

其中，

$$\alpha_{t,i} = softmax(sim(x_t, x_i)) = \frac{sim(q_t, k_i)}{\sum_{i=1}^{t} sim(q_t, k_i)} 。 \tag{1.32}$$

其中，$sim(\cdot, \cdot)$ 用于度量两个输入之间的相关程度，softmax 函数用于对此相关程度进行归一化。此外，

$$q_i = W_q x_i, \quad k_i = W_k x_i, \quad v_i = W_v x_i , \tag{1.33}$$

其中，W_q, W_k, W_v 分别为 query，key，value 编码器的参数。以包含三个元素的输入 $\{x_1, x_2, x_3\}$ 为例，Transformer 自注意力的实现图1.6所示。

2. 全连接前馈层（Fully-connected Feedforwad Layer）

全连接前馈层占据了 Tranformer 近三分之二的参数，掌管着 Tranformer 模型的记忆。其可以看作是一种 Key-Value 模式的记忆存储管理模块 [7]。全连接前馈层包含两层，两层之间由 ReLU 作为激活函数。设全连接前馈层的输入为 v, 全连接前馈层可由下式表示：

$$FFN(v) = \max(0, W_1 v + b_1)W_2 + b_2。 \tag{1.34}$$

其中，W_1 和 W_2 分别为第一层和第二层的权重参数，b_1 和 b_2 分别为第一层和第二层的偏置参数。其中第一层的可看作神经记忆中的 key，而第二层可看作 value。

3. 层正则化（Layer Normalization）

层正则化用以加速神经网络训练过程并取得更好的泛化性能 [1]。设输入到层正则化层的向量为 $v = \{v_i\}_{i=1}^n$。层正则化层将在 v 的每一维度 v_i 上都进行层正则化操作。具体地，层正则化操作可以表示为下列公式：

$$LN(v_i) = \frac{\alpha(v_i - \mu)}{\delta} + \beta。 \tag{1.35}$$

其中，α 和 β 为可学习参数。μ 和 δ 分别是隐藏状态的均值和方差，可由下列公式分别计算。

$$\mu = \frac{1}{n}\sum_{i=1}^{n} v_i, \quad \delta = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2}。 \tag{1.36}$$

4. 残差连接（Residual Connections）

引入残差连接可以有效解决梯度消失问题。在基本的 Transformer 编码模块中包含两个残差连接。第一个残差连接是将自注意力层的输入由一条旁路叠加到自注意力层的输出上，然后输入给层正则化。第二个残差连接是将全连接前馈层的输入由一条旁路引到全连接前馈层的输出上，然后输入给层正则化。

上述将层正则化置于残差连接之后的网络结构被称为 Post-LN Transformer。与之相对的，还有一种将层正则化置于残差连接之前的网络结构，称之为 Pre-LN

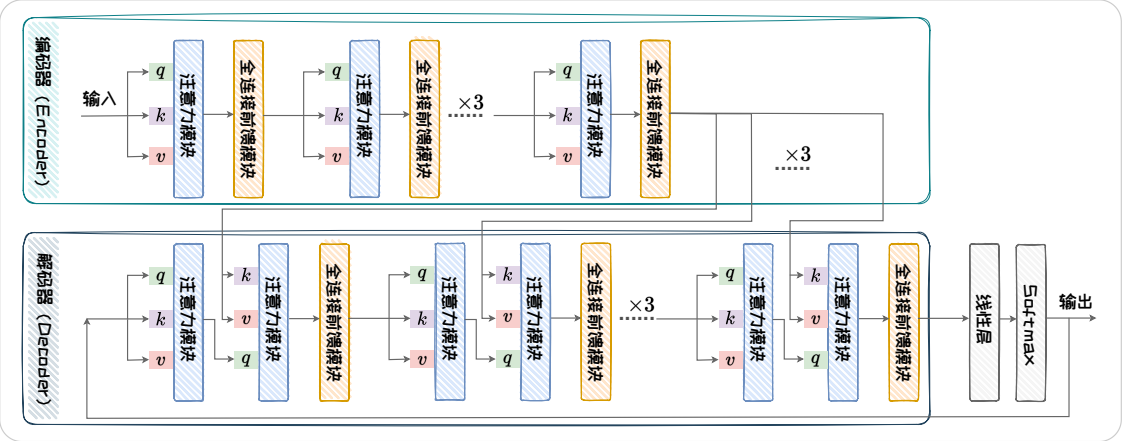


图 1.7: Transfomer 结构示意图。

Transformers。对比两者，Post-LN Transformer 应对表征坍塌（Representation Collapse）的能力更强，但处理梯度消失略弱。而 Pre-LN Transformers 可以更好的应对梯度消失，但处理表征坍塌的能力略弱。具体分析可参考文献 [7, 22]。

原始的 Transformer 采用 Encoder-Decoder 架构，其包含 Encoder 和 Decoder 两部分。这两部分都是由自注意力模块和全连接前馈模块重复连接构建而成。其整体结构如图1.7所示。其中，Encoder 部分由六个级联的 encoder layer 组成，每个 encoder layer 包含一个注意力模块和一个全连接前馈模块。其中的注意力模块为**自注意力模块**（query，key，value 的输入是相同的）。Decoder 部分由六个级联的 decoder layer 组成，每个 decoder layer 包含两个注意力模块和一个全连接前馈模块。其中，第一个注意力模块为**自注意力模块**，第二个注意力模块为**交叉注意力模块**（query，key，value 的输入不同）。Decoder 中第一个 decoder layer 的自注意力模块的输入为模型的输出。其后的 decoder layer 的自注意力模块的输入为上一个 decoder layer 的输出。Decoder 交叉注意力模块的输入分别是自注意力模块的输出（query）和最后一个 encoder layer 的输出（key，value）。

Transformer 的 Encoder 部分和 Decoder 部分都可以单独用于构造语言模型，分别对应 Encoder-Only 模型和 Decoder-Only 模型。Encoder-Only 模型和 Decoder-Only 模型的具体结构将在第二章中进行详细介绍。

1.3.2 基于 Transformer 的语言模型

在 Transformer 的基础上，可以设计多种预训练任务来训练语言模型。例如，我们可以基于 Transformer 的 Encoder 部分，结合“掩词补全”等任务来训练 Encoder-Only 语言模型，如 BERT [5]；我们可以同时应用 Transformer 的 Endcoder 和 Decoder 部分，结合“截断补全”、“顺序恢复”等多个有监督和自监督任务来训练 Encoder-Decoder 语言模型，如 T5 [18]；我们可以同时应用 Transformer 的 Decoder 部分，利用“下一词预测”任务来训练 Decoder-Only 语言模型，如 GPT-3 [3]。这些语言模型将在第二章中进行详细介绍。下面将以下一词预测任务为例，简单介绍训练 Transformer 语言模型的流程。

对词序列 $\{w_1, w_2, w_3, ..., w_N\}$，基于 Transformer 的语言模型根据 $\{w_1, w_2, ..., w_i\}$ 预测下一个词 w_{i+1} 出现的概率。在基于 Transformer 的语言模型中，输出为一个向量，其中每一维代表着词典中对应词的概率。设词典 D 中共有 $|D|$ 个词 $\{\hat{w_1}, \hat{w_2}, ..., \hat{w_{|D|}}\}$。基于 Transformer 的语言模型的输出可表示为 $o_i = \{o_i[\hat{w_d}]\}_{d=1}^{D}$，其中，$o_i[\hat{w_d}]$ 表示词典中的词 $\hat{w_d}$ 出现的概率。因此，对 Transformer 的语言模型对词序列 $\{w_1, w_2, w_3, ..., w_N\}$ 整体出现的概率的预测为：

$$P(w_{1:N}) = \prod_{i=1}^{N-1} P(w_{i+1}|w_{1:i}) = \prod_{i=1}^{N} o_i[w_{i+1}] \tag{1.37}$$

与训练 RNN 语言模型相同，Transformer 语言模型也常用如下交叉熵函数作为损失函数。

$$l_{CE}(o_i) = -\sum_{d=1}^{|D|} I(\hat{w_d} = w_{i+1}) \log o_i[w_{i+1}] = -\log o_i[w_{i+1}]。 \tag{1.38}$$

其中，$I(\cdot)$ 为指示函数，当 $\hat{w_d} = w_{i+1}$ 时等于 1，当 $\hat{w_d} \neq w_{i+1}$ 时等于 0。

设训练集为 S，Transformer 语言模型的损失可以构造为：

$$L(S, W) = \frac{1}{N|S|} \sum_{s=1}^{|S|} \sum_{i=1}^{N} l_{CE}(o_{i,s}) \tag{1.39}$$

其中，$o_{i,s}$ 为 Transformer 语言模型输入样本 s 的前 i 个词时的输出。在此损失的基础上，构建计算图，进行反向传播，便可对 Transformer 语言模型进行训练。

上述训练过程结束之后，我们可以将 Encoder 的输出作为特征，然后应用这些特征解决下游任务。此外，还可在“自回归”的范式下完成文本生成任务。在自回归中，第一轮，我们首先将第一个词输入给 Transformer 语言模型，经过解码，得到一个输出词。然后，我们将第一轮输出的词与第一轮输入的词拼接，作为第二轮的输入，然后解码得到第二轮的输出。接着，将第二轮的输出和输入拼接，作为第三轮的输入，以此类推。每次将本轮预测到的词拼接到本轮的输入上，输入给语言模型，完成下一轮预测。在循环迭代的“自回归”过程中，我们不断生成新的词，这些词便构成了一段文本。与训练 RNN 语言模型一样，Transformer 模型的预训练过程依然采用第 1.2.2 节中提到的“Teacher Forcing”的范式。

相较于 RNN 模型串行的循环迭代模式，Transformer 并行输入的特性，使其容易进行并行计算。但是，Transformer 并行输入的范式也导致网络模型的规模随输入序列长度的增长而平方次增长。这为应用 Transformer 处理长序列带来挑战。

1.4 语言模型的采样方法

语言模型的输出为一个向量，该向量的每一维代表着词典中对应词的概率。在采用自回归范式的文本生成任务中，语言模型将依次生成一组向量并将其解码为文本。将这组向量解码为文本的过程被成为语言模型解码。解码过程显著影响着生成文本的质量。当前，两类主流的解码方法可以总结为 (1). 概率最大化方法; (2). 随机采样方法。两类方法分别在下面章节中进行介绍。

1.4.1 概率最大化方法

设词典为 D，输入文本为 $\{w_1, w_2, w_3, ..., w_N\}$，第 i 轮自回归中输出的向量为 $o_i = \{o_i[w_d]\}_{d=1}^{|D|}$，模型在 M 轮自回归后生成的文本为 $\{w_{N+1}, w_{N+2}, w_{N+3}, ..., w_{N+M}\}$。生成文档的出现的概率可由下式进行计算。

$$P(w_{N+1:N+M}) = \prod_{i=N}^{N+M-1} P(w_{i+1}|w_{1:i}) = \prod_{i=N}^{N+M-1} o_i[w_{i+1}] \tag{1.40}$$

基于概率最大化的解码方法旨在最大化 $P(w_{N+1:N+M})$，以生成出可能性最高的文本。该问题的搜索空间大小为 M^D，是 NP-Hard 问题。现有概率最大化方法通常采用启发式搜索方法。本节将介绍两种常用的基于概率最大化的解码方法。

1. 贪心搜索（Greedy Search）

贪心搜索在在每轮预测中都选择概率最大的词，即

$$w_{i+1} = arg\ max_{w \in D}\ o_i[w] \tag{1.41}$$

贪心搜索只顾“眼前利益”，忽略了“远期效益”。**当前概率大的词有可能导致后续的词概率都很小**。贪心搜索容易陷入局部最优，难以达到全局最优解。以图1.8为例，当输入为“生成一个以长颈鹿开头的故事：长颈鹿”时，预测第一个词为“是”的概率最高，为 0.3。但选定“是”之后，其他的词的概率都偏低。如果按照贪心搜索的方式，我们最终得到的输出为“是草食”。其概率仅为 0.03。而如果我们在第一个词选择了概率第二的“脖子”，然后第二个词选到了“长”，最终的概率可以达到 0.1。通过此例，可以看出贪心搜索在求解概率最大的时候容易陷入局部最优。为缓解此问题，可以采用波束搜索（Beam Search）方法进行解码。

2. 波束搜索（Beam Search）

波束搜索在每轮预测中都先保留 b 个可能性最高的词 $B_i = \{w_{i+1}^1, w_{i+1}^2, ..., w_{i+1}^b\}$，即：

$$min\{o_i[w]\ for\ w \in B_i\} > max\{o_i[w]\ for\ w \in D - B_i\}。 \tag{1.42}$$

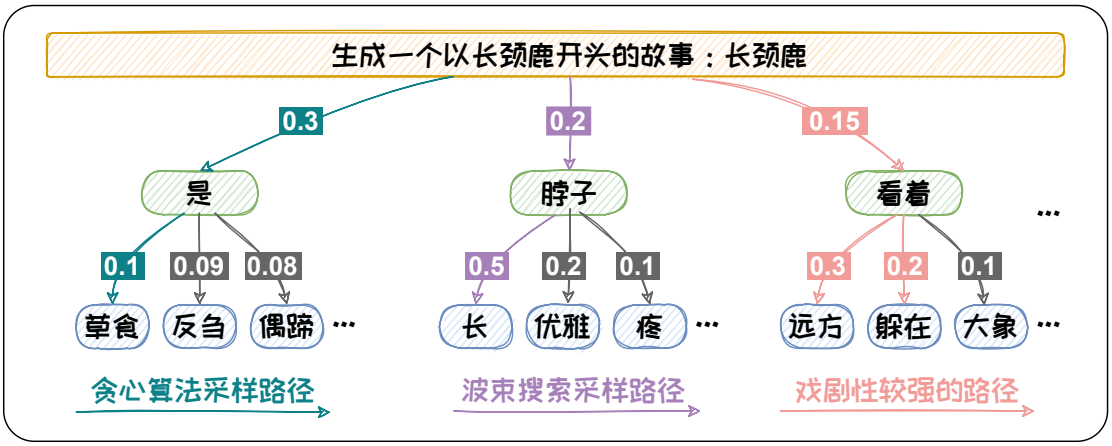


图 1.8: 贪心搜索与波束搜索对比以及概率最大化解码的潜在问题。

在结束搜索时，得到 M 个集合，即 $\{B_i\}_{i=1}^{M}$。找出最优组合使得联合概率最大，即：

$$\{w_{N+1}, ..., w_{N+M}\} = arg\ max_{\{w^i \in B_i\ for\ 1 \le i \le M\}} \prod_{i=1}^{M} o_{N+i}[w^i]。 \tag{1.43}$$

继续以上面的“生成一个以长颈鹿开头的故事”为例，从图1.8中可以看出如果我们采用 $b = 2$ 的波束搜索方法，我们可以得到“是草食”，“是反刍”，“脖子长”，“脖子优雅”四个候选组合，对应的概率分别为：0.03，0.027，0.1，0.04。我们容易选择到概率最高的“脖子长”。

但是，概率最大的文本**通常是最为常见的文本**。这些文本会略显平庸。在开放式文本生成中，无论是贪心搜索还是波束搜索都容易生成一些“废话文学”—重复且平庸的文本。其所生成的文本缺乏多样性 [19]。如在图1.8中的例子所示，概率最大的方法会生成” 脖子长”。” 长颈鹿脖子长” 这样的文本新颖性较低。为了提升生成文本的新颖度，我们可以在解码过程中加入一些随机元素。这样的话就可以解码到一些不常见的组合，从而使得生成的文本更具创意，更适合开放式文本任务。在解码过程中加入随机性的方法，成为随机采样方法。下节将对随机采样方法进行介绍。

1.4.2 随机采样方法

为了增加生成文本的多样性，随机采样的方法**在预测时增加了随机性**。在每轮预测时，其先选出一组可能性高的候选词，然后按照其概率分布进行随机采样，采样出的词作为本轮的预测结果。当前，主流的 Top-K 采样和 Top-P 采样方法分别通过指定候选词数量和划定候选词概率阈值的方法对候选词进行选择。在采样方法中加入 Temperature 机制可以对候选词的概率分布进行调整。接下来将对 Top-K 采样、Top-P 采样和 Temperature 机制分别展开介绍。

1. Top-K 采样

Top-K 采样在每轮预测中都选取 K 个概率最高的词 $\{w_{i+1}^1, w_{i+1}^2, ..., w_{i+1}^K\}$ 作为本轮的候选词集合，然后对这些词的概率用 softmax 函数进行归一化，得到如下分布函数

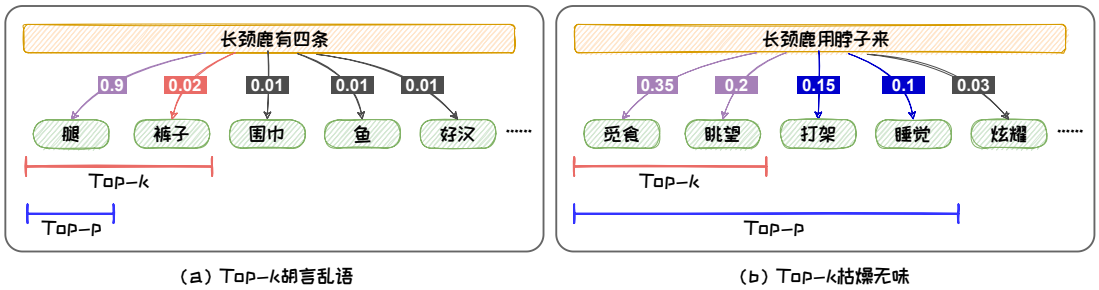
$$p(w_{i+1}^1, ..., w_{i+1}^K) = \left\{ \frac{exp(o_i[w_{i+1}^1])}{\sum_{j=1}^K exp(o_i[w_{i+1}^j])}, ..., \frac{exp(o_i[w_{i+1}^K])}{\sum_{j=1}^K exp(o_i[w_{i+1}^j])} \right\}。 \tag{1.44}$$

然后根据该分布**采样**出本轮的预测的结果，即

$$w_{i+1} \sim p(w_{i+1}^1, ..., w_{i+1}^K)。 \tag{1.45}$$

Top-K 采样可以有效的增加生成文本的新颖度，例如在上述图1.8所示的例子中选用 Top-3 采样的策略，则有可能会选择到“看着躲在”。“长颈鹿看着躲在”可能是一个极具戏剧性的悬疑故事的开头。

但是，将候选集设置为固定的大小 K 将导致上述分布在不同轮次的预测中存在很大差异。当候选词的分布的方差较大的时候，可能会导致本轮预测**选到概率较小、不符合常理的词**，从而产生“胡言乱语”。例如，在如图1.9 (a) 所示的例子中，Top-2 采样有可能采样出“长颈鹿有四条裤子”。而当候选词的分布的方差较小的时候，甚至趋于均匀分布时，固定尺寸的候选集中**无法容纳更多的具有相近概率的词，导致候选集不够丰富**，从而导致所选词缺乏新颖性而产生“枯燥无趣”的



（a）Top-k胡言乱语　　（b）Top-k枯燥无味

图 1.9: Top-K 采样的潜在问题及其与 Top-P 方法的对比。

文本。例如，在如下图1.9 (b) 所示的例子中，通过 Top-2 采样，我们只能得到“长颈鹿用脖子来觅食”或者“长颈鹿用脖子来眺望”，这些都是人们熟知的长颈鹿脖子的用途，缺乏新意。但是其实长颈鹿的脖子还可以用于打架或睡觉，Top-2 采样的方式容易将这些新颖的不常见的知识排除。为了解决上述问题，我们可以使用 Top-P 采样，也称 Nucleus 采样。

2. Top-P 采样

为了解决固定候选集所带来的问题，Top-P 采样（即 Nucleus 采样）被提出 [9]。其设定阈值 p 来对候选集进行选取。其候选集可表示为 $S_p = \{w_{i+1}^1, w_{i+1}^2, ..., w_{i+1}^{|S_p|}\}$，其中，对 S_p 有，$\sum_{w \in S_p} o_i[w] \geq p$。候选集中元素的分布服从

$$p(w_{i+1}^1, ..., w_{i+1}^{|S_p|}) = \left\{ \frac{exp(o_i[w_{i+1}^1])}{\sum_{j=1}^{|S_p|} exp(o_i[w_{i+1}^j])}, ..., \frac{exp(o_i[w_{i+1}^{|S_p|}])}{\sum_{j=1}^{|S_p|} exp(o_i[w_{i+1}^j])} \right\}。 \tag{1.46}$$

然后根据该分布采样出本轮的预测的结果，即

$$w_{i+1} \sim p(w_{i+1}^1, ..., w_{i+1}^{|S_p|})。 \tag{1.47}$$

应用阈值作为候选集选取的标准之后，Top-P 采样可以**避免选到概率较小、不符合常理的词**，从而减少“胡言乱语”。例如在图1.9 (a) 所示例子中，我们若以 0.9 作为阈值，则就可以很好的避免“长颈鹿有四条裤子”的问题。并且，其**还可以容**

纳更多的具有相近概率的词，增加文本的丰富度，改善“枯燥无趣”。例如在图1.9 (b) 所示的例子中，我们若以 0.9 作为阈值，则就可以包含打架、睡觉等长颈鹿脖子鲜为人知的用途。

3. Temperature 机制

Top-K 采样和 Top-P 采样的随机性由语言模型输出的概率决定，不可自由调整。但在不同场景中，我们对于随机性的要求可能不一样。比如在开放文本生成中，我们更倾向于生成更具创造力的文本，所以我们需要采样具有更强的随机性。而在代码生成中，我们希望生成的代码更为保守，所以我们需要较弱的随机性。引入 Temperature 机制可以**对解码随机性进行调节**。Temperature 机制通过对 Softmax 函数中的自变量进行尺度变换，然后利用 Softmax 函数的非线性实现对分布的控制。设 Temperature 尺度变换的变量为 T。

引入 Temperature 后，Top-K 采样的候选集的分布如下所示：

$$p(w_{i+1}^1, ..., w_{i+1}^K) = \left\{ \frac{exp(\frac{o_i[w_{i+1}^1]}{T})}{\sum_{j=1}^K exp(\frac{o_i[w_{i+1}^j]}{T})}, ..., \frac{exp(\frac{o_i[w_{i+1}^K]}{T})}{\sum_{j=1}^K exp(\frac{o_i[w_{i+1}^j]}{T})} \right\}。 \tag{1.48}$$

引入 Temperature 后，Top-P 采样的候选集的分布如下所示：

$$p(w_{i+1}^1, ..., w_{i+1}^{|S_p|}) = \left\{ \frac{exp(\frac{o_i[w_{i+1}^1]}{T})}{\sum_{j=1}^{|S_p|} exp(\frac{o_i[w_{i+1}^j]}{T})}, ..., \frac{exp(\frac{o_i[w_{i+1}^{|S_p|}]}{T})}{\sum_{j=1}^{|S_p|} exp(\frac{o_i[w_{i+1}^j]}{T})} \right\}。 \tag{1.49}$$

容易看出，当 $T > 1$ 时，Temperature 机制会使得候选集中的词的概率差距减小，分布变得更平坦，从而增加随机性。当 $0 < T < 1$ 时，Temperature 机制会使得候选集中的元素的概率差距加大，强者越强，弱者越弱，概率高的候选词会容易被选到，从而随机性变弱。Temperature 机制可以有效的对随机性进行调节来满足不同的需求。

1.5 语言模型的评测

得到一个语言模型后，我们需要对其生成能力进行评测，以判断其优劣。评测语言模型生成能力的方法可以分为两类。第一类方法不依赖具体任务，直接通过语言模型的输出来评测模型的生成能力，称之为内在评测（Intrinsic Evaluation）。第二类方法通过某些具体任务，如机器翻译、摘要生成等，来评测语言模型处理这些具体生成任务的能力，称之为外在评测（Extrinsic Evaluation）。

1.5.1 内在评测

在内在评测中，测试文本通常由与预训练中所用的文本独立同分布的文本构成，**不依赖于具体任务**。最为常用的内部评测指标是困惑度（Perplexity）[10]。其度量了语言模型对测试文本感到“困惑”的程度。设测试文本为 $s_{test} = w_{1:N}$。语言模型在测试文本 s_{test} 上的困惑度 PPL 可由下式计算：

$$PPL(s_{test}) = P(w_{1:N})^{-\frac{1}{N}} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{<i})}}。 \tag{1.50}$$

由上式可以看出，如果语言模型对测试文本越“肯定”（即生成测试文本的概率越高），则困惑度的值越小。而语言模型对测试文本越“不确定”（即生成测试文本的概率越低），则困惑度的值越大。由于测试文本和预训练文本同分布，预训练文本代表了我们想要让语言模型学会生成的文本，如果语言模型在这些测试文本上越不“困惑”，则说明语言模型越符合我们对其训练的初衷。因此，困惑度可以一定程度上衡量语言模型的生成能力。

对困惑度进行改写，其可以改写成如下等价形式。

$$PPL(s_{test}) = exp(-\frac{1}{N}\sum_{i=1}^{N} log\ P(w_i|w_{<i}))。 \tag{1.51}$$

其中，$-\frac{1}{N}\sum_{i=1}^{N} log\ P(w_i|w_{<i})$ 可以看作是生成模型生成的词分布与测试样本真实

的词分布间的交叉熵，即 $-\frac{1}{N}\sum_{i=1}^{N}\sum_{d=1}^{|D|} I(\hat{w}_d = w_i)\log o_{i-1}[w_i]$，其中 D 为语言模型所采用的词典。因为 $P(w_i|w_{<i}) \leq 1$，所以此交叉熵是生成模型生成的词分布的信息熵的上界，即

$$-\frac{1}{N}\sum_{i=1}^{N} P(w_i|w_{<i})\ log\ P(w_i|w_{<i}) \leq -\frac{1}{N}\sum_{i=1}^{N} log\ P(w_i|w_{<i})。 \tag{1.52}$$

因此，困惑度减小也意味着熵减，意味着模型“胡言乱语”的可能性降低。

1.5.2 外在评测

在外在评测中，测试文本通常包括该任务上的问题和对应的标准答案，其依赖于具体任务。通过外在评测，我们可以评判语言模型处理特定任务的能力。外在评测方法通常可以分为基于统计指标的评测方法和基于语言模型的评测方法两类。以下对此两类方法中的经典方法进行介绍。

1. 基于统计指标的评测

基于统计指标的方法构造统计指标来评测语言模型的输出与标准答案间的契合程度，并以此作为评测语言模型生成能力的依据。BLEU（BiLingual Evaluation Understudy）和 ROUGE（Recall-Oriented Understudy for Gisting Evaluation）是应用最为广泛的两种统计指标。其中，BLEU 是精度导向的指标，而 ROUGE 是召回导向的指标。以下分别对这两个指标展开介绍。

BLEU 被提出用于评价模型在机器翻译（Machine Translation, MT）任务上的效果 [6]。其在词级别上计算生成的翻译与参考翻译间的重合程度。具体地，BLEU 计算多层次 n-gram 精度的几何平均。设生成的翻译文本的集合为 $S_{gen} = \{S_{gen}^i\}_{i=1}^{|S_{gen}|}$，对应的参考翻译集合为 $S_{ref} = \{S_{ref}^i\}_{i=1}^{|S_{ref}|}$，其中，$S_{gen}^i$ 与 S_{ref}^i 一一对应，且 $|S_{ref}| = |S_{gen}|$。原始的 n-gram 精度的定义如下：

$$Pr(g_n) = \frac{\sum_{i=1}^{|S_{gen}|}\sum_{g_n \in S_{gen}^i} Count_{match}(g_n, S_{ref}^i)}{\sum_{i=1}^{|S_{gen}|}\sum_{g_n \in S_{gen}^i} Count(g_n)}。 \tag{1.53}$$

其中，g_n 代表 n-gram。上式的分子计算了生成的翻译与参考翻译的重合的 n-gram 的个数，分母计算了生成的翻译中包含的 n-gram 的总数。例如，MT 模型将“大语言模型”翻译成英文，生成的翻译为“big language models”，而参考文本为“large language models”。当 $n=1$ 时，$Pr(g_1)=\frac{2}{3}$。当 $n=2$ 时，$Pr(g_2)=\frac{1}{2}$。

基于 n-gram 精度，BLEU 取 N 个 n-gram 精度的几何平均作为评测结果：

$$BLEU=\sqrt[N]{\prod_{i=1}^{N}Pr(g_n)}=exp\left(\sum_{n=1}^{N}logPr(g_n)\right)。\tag{1.54}$$

例如，当 $N=3$ 时，BLEU 是 unigram 精度，bigram 精度，trigram 精度的几何平均。在以上原始 BLEU 的基础上，我们还可以通过对不同的 n-gram 精度进行加权或对不同的文本长度设置惩罚项来对 BLEU 进行调整，从而得到更为贴近人类评测的结果。

ROUGE 被提出用于评价模型在摘要生成（Summarization）任务上的效果 [13]。常用的 ROUGE 评测包含 ROUGE-N, ROUGE-L, ROUGE-W, 和 ROUGE-S 四种。其中，ROUGE-N 是基于 n-gram 的召回指标，ROUGE-L 是基于最长公共子序列（Longest Common Subsequence, LCS）的召回指标。ROUGE-W 是在 ROUGE-L 的基础上，引入对 LCS 的加权操作后的召回指标。ROUGE-S 是基于 Skip-bigram 的召回指标。下面给出 ROUGE-N, ROUGE-L 的定义。ROUGE-W 和 ROUGE-S 的具体计算方法可在 [13] 中找到。

ROUGE-N 的定义如下：

$$ROUGE\text{-}N=\frac{\sum_{s\in S_{ref}}\sum_{g_n\in s}Count_{match}(g_n,s_{gen})}{\sum_{s\in S_{ref}}\sum_{g_n\in s}Count(g_n)}。\tag{1.55}$$

ROUGE-L 的定义如下：

$$ROUGE\text{-}L=\frac{(1+\beta^2)R_r^gR_g^r}{R_r^g+\beta^2R_g^r}。\tag{1.56}$$

其中，

$$R_r^g = \frac{LCS(s_{ref}, s_{gen})}{|s_{ref}|}, \tag{1.57}$$

$$R_g^r = \frac{LCS(s_{ref}, s_{gen})}{|s_{gen}|}, \tag{1.58}$$

$LCS(s_{ref}, s_{gen})$ 是模型生成的摘要 s_{gen} 与参考摘要 s_{ref} 间的最大公共子序列的长度，$\beta = R_g^r / R_r^g$。

基于统计指标的评测方法通过对语言模型生成的答案和标准答案间的重叠程度进行评分。这样的评分无法完全适应生成任务中表达的多样性，与人类的评测相差甚远，尤其是在生成的样本具有较强的创造性和多样性的时候。为解决此问题，可以在评测中引入一个其他语言模型作为“裁判”，利用此“裁判”在预训练阶段掌握的能力对生成的文本进行评测。下面对这种引入“裁判”语言模型的评测方法进行介绍。

2. 基于语言模型的评测

目前基于语言模型的评测方法主要分为两类：（1）基于上下文词嵌入（Contextual Embeddings）的评测方法；（2）基于生成模型的评测方法。典型的基于上下文词嵌入的评测方法是 BERTScore [24]。典型的基于生成模型的评测方法是 G-EVAL [14]。与 BERTScore 相比，G-EVAL 无需人类标注的参考答案。这使其可以更好的适应到缺乏人类标注的任务中。

BERTScore 在 BERT 的上下文词嵌入向量的基础上，计算生成文本 s_{gen} 和参考文本 s_{ref} 间的相似度来对生成样本进行评测。BERT 将在第二章给出详细介绍。设生成文本包含 $|s_{gen}|$ 个词，即 $s_{gen} = \{w_g^i\}_{i=1}^{|s_{gen}|}$。设参考文本包含 $|s_{ref}|$ 个词，即 $s_{ref} = \{w_r^i\}_{i=1}^{|s_{ref}|}$。利用 BERT 分别得到 s_{gen} 和 s_{ref} 中每个词的上下文词嵌入向量，即 $v_g^i = BERT(w_g^i | s_{gen})$，$v_r^i = BERT(w_r^i | s_{ref})$。利用生成文本和参考文本的词嵌入向量集合 $v_{gen} = \{v_g^i\}_{i=1}^{|v_{gen}|}$ 和 $v_{ref} = \{v_r^i\}_{i=1}^{|v_{ref}|}$ 便可计算 BERTScore。BERTScore

从精度（Precision），召回（Recall）和 F1 量度三个方面对生成文档进行评测。其定义分别如下：

$$P_{BERT} = \frac{1}{|v_{gen}|}\sum_{i=1}^{|v_{gen}|} max_{v_r \in v_{ref}} v_g^{i\top} v_r^i, \tag{1.59}$$

$$R_{BERT} = \frac{1}{|v_{ref}|}\sum_{i=1}^{|v_{ref}|} max_{v_g \in v_{gen}} v_r^{i\top} v_g^i, \tag{1.60}$$

$$F_{BERT} = \frac{2P_{BERT} \cdot R_{BERT}}{P_{BERT} + R_{BERT}}。 \tag{1.61}$$

相较于统计评测指标，BERTScore 更接近人类评测结果。但是，BERTScore 依赖于人类给出的参考文本。这使其无法应用于缺乏人类标注样本的场景中。得益于生成式大语言模型的发展，G-EVAL 利用 GPT-4 在没有参考文本的情况下对生成文本进行评分。G-EVAL 通过提示工程（Prompt Engineering）引导 GPT-4 输出评测分数。Prompt Engineering 将在本书第三章进行详细讲解。

如下图所示，G-EVAL 的 Prompt 分为三部分：(1) 任务描述与评分标准；(2) 评测步骤；(3) 输入文本与生成的文本。在第一部分中，任务描述指明需要的评测的任务式什么（如摘要生成），评分标准给出评分需要的范围，评分需要考虑的因素等内容。第二部分的评测步骤是在第一部分内容的基础上由 GPT-4 自己生成的思维链（Chain-of-Thoughts, CoT）。本书的第三章将对思维链进行详细讲解。第三部分的输入文本与生成的文本是源文本和待评测模型生成的文本。例如摘要生成任务中的输入文本是原文，而生成的文本就是生成摘要。将上述三部分组合在一个 prompt 里面然后输入给 GPT-4，GPT-4 便可给出对应的评分。直接将 GPT-4 给出的得分作为评分会出现区分度不够的问题，因此，G-EVAL 还引入了对所有可能得分进行加权平均的机制来进行改进 [14]。

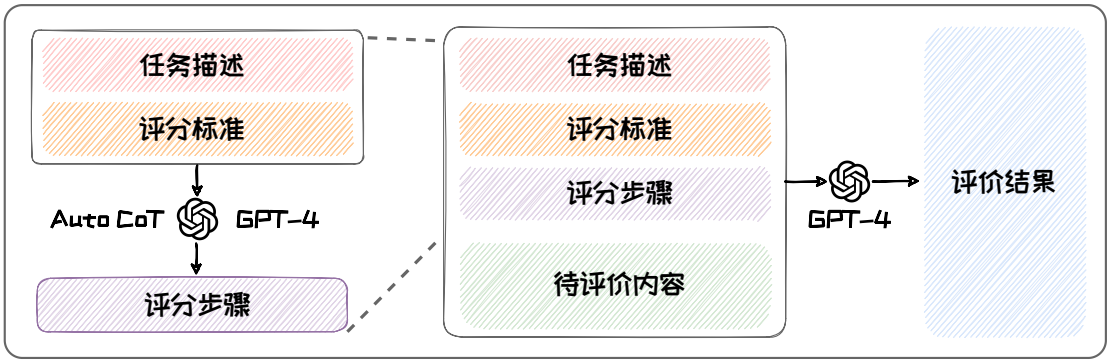


图 1.10: G-EVAL 评测流程。

除 G-EVAL 外，近期还有多种基于生成模型的评测方法被提出 [12]。其中典型的有 InstructScore [23]，其除了给出数值的评分，还可以给出对该得分的解释。基于生成模型的评测方法相较于基于统计指标的方法和基于上下文词嵌入的评测方法而言，在准确性、灵活性、可解释性等方面都具有独到的优势。可以预见，未来基于生成模型的评测方法将得到更为广泛的关注和应用。

参考文献

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450.

[2] Samy Bengio et al. “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks”. In: *NeurIPS*. 2015.

[3] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *NeurIPS*. 2020.

[4] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555.

[5] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *NAACL-HLT*. 2019.

[6] Markus Freitag, David Grangier, and Isaac Caswell. “BLEU might be Guilty but References are not Innocent”. In: *EMNLP*. 2020.

[7] Mor Geva et al. “Transformer Feed-Forward Layers Are Key-Value Memories”. In: *EMNLP*. 2021.

[8] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computing* 9.8 (1997), pp. 1735–1780.

[9] Ari Holtzman et al. “The Curious Case of Neural Text Degeneration”. In: *ICLR*. 2020.

[10] F. Jelinek et al. “Perplexity—a measure of the difficulty of speech recognition tasks”. In: *The Journal of the Acoustical Society of America* 62 (1997), S63–S63.

[11] Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009. ISBN: 9780135041963.

[12] Zhen Li et al. *Leveraging Large Language Models for NLG Evaluation: Advances and Challenges*. 2024. arXiv: 2401.07103.

[13] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries”. In: *ACL*. 2004.

[14] Yang Liu et al. “Gpteval: Nlg evaluation using gpt-4 with better human alignment”. In: *EMNLP*. 2023.

[15] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, 2001. ISBN: 978-0-262-13360-9.

[16] OpenAI. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774.

[17] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *ICML*. 2013.

[18] Colin Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *Journal of Machine Learning Research* 21 (2020), 140:1–140:67.

[19] Ashwin K. Vijayakumar et al. “Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models”. In: *AAAI*. 2018.

[20] Joseph Weizenbaum. “ELIZA - a computer program for the study of natural language communication between man and machine”. In: *Communications Of The ACM* 9.1 (1966), pp. 36–45.

[21] Ronald J. Williams and David Zipser. “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks”. In: *Neural Computing* 1.2 (1989), pp. 270–280.

[22] Shufang Xie et al. *ResiDual: Transformer with Dual Residual Connections*. 2023. arXiv: 2304.14802.

[23] Wenda Xu et al. "INSTRUCTSCORE: Towards Explainable Text Generation Evaluation with Automatic Feedback". In: *EMNLP*. 2023.

[24] Tianyi Zhang et al. "BERTScore: Evaluating Text Generation with BERT". In: *ICLR*. 2020.

ZHEJIANG UNIVERSITY
1897

2 大语言模型架构

随着数据资源和计算能力的爆发式增长，语言模型的参数规模和性能表现实现了质的飞跃，迈入了大语言模型（Large Language Model, LLM）的新时代。凭借着**庞大的参数量**和**丰富的训练数据**，大语言模型不仅展现出了强大的泛化能力，还催生了新智能的涌现，勇立生成式人工智能（Artificial Intelligence Generated Content, AIGC）的浪潮之巅。当前，大语言模型技术蓬勃发展，各类模型层出不穷。这些模型在广泛的应用场景中已经展现出与人类比肩甚至超过人类的能力，引领着由 AIGC 驱动的新一轮产业革命。本章将深入探讨大语言模型的相关背景知识，并分别介绍 Encoder-only、Encoder-Decoder 以及 Decoder-only 三种主流模型架构。通过列举每种架构的代表性模型，深入分析它们在网络结构、训练方法等方面的主要创新之处。最后，本章还将简单介绍一些非 Transformer 架构的模型，以展现当前大语言模型研究百花齐放的发展现状。

* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

2.1 大数据 + 大模型 → 新智能

在自然语言处理的前沿领域，大语言模型正以其庞大的模型规模、海量数据的吞吐能力和卓越的模型性能，推动着一场技术革新的浪潮。当我们谈论“大语言模型”之大时，所指的不仅仅是**模型规模的庞大**，也涵盖了**训练数据规模的庞大**，以及由此衍生出的**模型能力的强大**。这些模型如同探索未知领域的巨轮，不仅在已有的技术上不断突破性能的极限，更在新能力的探索中展现出惊人的潜力。

截止 2024 年 6 月，国内外已经见证了超过百种大语言模型的诞生，这些大语言模型在学术界和工业界均产生了深远的影响。图2.1展示了其中一些具有重要影响力的模型。

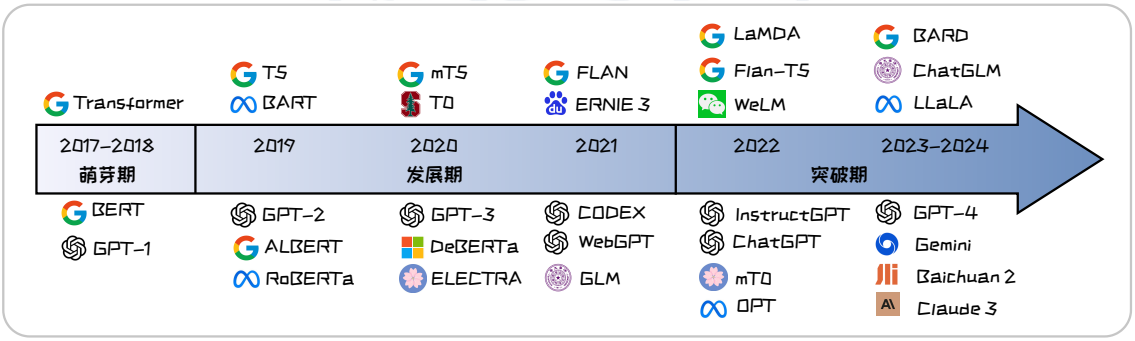


图 2.1: 大语言模型涌现能力的三个阶段。

大语言模型的发展历程可以大致划分为三个阶段。2017 至 2018 年是基础模型的**萌芽期**，以 Transformer 架构的诞生和 BERT[11]、GPT-1[27] 模型的问世为标志，开启了预训练语言模型的新纪元。2019 至 2022 年是大语言模型的**发展期**，通过 GPT-2 [1]、T5[29] 以及 GPT-3[5] 等模型在参数规模以及能力上的大幅提升，研究者开始深入探索大语言模型的潜力。2022 年起则是大语言模型的**突破期**，ChatGPT [2]以及 GPT-4 [3]等模型的发布标志着大语言模型相关技术的显著进步。同时，各大公司

[1]https://openai.com/index/gpt-2-1-5b-release

[2]https://openai.com/blog/chatgpt

[3]https://openai.com/index/gpt-4-research

和研究机构也纷纷推出了自己的模型，例如百川智能的百川大模型 [44]，百度的文心一言等，推动了大语言模型的快速发展。

本节将深入剖析大型语言模型的发展历程，特别是在能力增强和新能力涌现方面的进展。我们将从模型规模和数据规模的增长出发，探讨这些因素如何共同作用，促进了模型性能的飞跃和新功能的出现。

2.1.1 大数据 + 大模型 → 能力增强

在数字化浪潮的推动下，数据如同汇聚的洪流，而模型则如同乘风破浪的巨舰。数据规模的增长为模型提供了更丰富的信息源，意味着模型可以学习到更多样化的语言模式和深层次的语义关系。而模型规模的不断扩大，极大地增加了模型的表达能力，使其能够捕捉到更加细微的语言特征和复杂的语言结构。在如此庞大的模型参数规模以及多样化的训练数据共同作用下，模型内在对数据分布的拟合能力不断提升，从而在复杂多变的数据环境中表现出更高的适应性和有效性 [7]。

然而模型规模和数据规模的增长并非没有代价，它们带来了更高的计算成本和存储需求，这要求我们在模型设计时必须在资源消耗和性能提升之间找到一个恰当的平衡点。为了应对这一挑战，大语言模型的扩展法则（Scaling Laws）应运而生。这些法则揭示了模型的能力随模型和数据规模的变化关系，为大语言模型的设计和优化提供了宝贵的指导和参考。本章节将深入介绍两种扩展法则：OpenAI 提出的 Kaplan-McCandlish 扩展法则以及 DeepMind 提出的 Chinchilla 扩展法则。

1. Kaplan-McCandlish 扩展法则

2020 年，OpenAI 团队的 Jared Kaplan 和 Sam McCandlish 等人 [16] 首次探究了神经网络的性能与数据规模 D 以及模型规模 N 之间的函数关系。他们在不同规模的数据集（从 2200 万到 230 亿个 Token）和不同规模的模型下（从 768 到 15 亿个参数）进行实验，并根据实验结果拟合出了两个基本公式：

$$L(D) = \left(\frac{D}{D_c}\right)^{\alpha_D}, \alpha_D \sim -0.095, D_c \sim 5.4 \times 10^{13}, \tag{2.1}$$

$$L(N) = \left(\frac{N}{N_c}\right)^{\alpha_N}, \alpha_N \sim -0.076, N_c \sim 8.8 \times 10^{13}。 \tag{2.2}$$

这里的 $L(N)$ 表示在数据规模固定时，**不同模型规模**下的交叉熵损失函数，反映了模型规模对拟合数据能力的影响。相应地，$L(D)$ 表示在模型规模固定时，**不同数据规模**下的交叉熵损失函数，揭示了数据量对模型学习的影响。L 的值衡量了模型拟合数据分布的准确性，值越小表明模型对数据分布的拟合越精确，其**自身学习能力**也就越强大。

实验结果和相关公式表明，模型的性能与模型模型以及数据规模这两个因素均高度正相关。然而，在模型规模相同的情况下，模型的**具体架构**对其性能的影响相对较小。因此，扩大模型规模和丰富数据集成为了提升大型模型性能的两个关键策略。

此外，OpenAI 在进一步研究计算预算的最优分配时发现，总计算量 C 与数据量 D 和模型规模 N 的乘积近似成正比，即 $C \approx 6ND$。在这一条件下，如果计算预算增加，为了达到最优模型性能，数据集的规模 D 以及模型规模 N 都应同步增加。但是**模型规模的增长速度应该略快于数据规模的增长速度**。具体而言，两者的最优配置比例应当为 $N_{opt} \propto C^{0.73}, D_{opt} \propto C^{0.27}$。这意味着，如果总计算预算增加了 10 倍，模型规模应扩大约 5.37 倍，而数据规模应扩大约 1.86 倍，以实现模型的最佳性能。

OpenAI 提出的这一扩展法则不仅定量地揭示了数据规模和模型规模对模型能力的重要影响，还指出了在模型规模上的投入应当略高于数据规模上的投入。这一发现不仅为理解语言模型的内在工作机制提供了新的见解，也为如何高效地训练这些模型提供了宝贵的指导意见。

2. Chinchilla 扩展法则

谷歌旗下 DeepMind 团队对“模型规模的增长速度应该略高于数据规模的增长速度”这一观点提出了不同的看法。在 2022 年，他们对更大范围的模型规模（从 7000 万到 1600 亿个参数）以及数据规模（从 50 亿到 5000 亿个 Token）进行了深入的实验研究，并据此提出了 Chinchilla 扩展法则 [15]：

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} \ , \tag{2.3}$$

$$E = 1.69,\ A = 406.4, B = 410.7,\ \alpha = 0.34,\ \beta = 0.28 \ 。 \tag{2.4}$$

DeepMind 同样探索了计算预算的最优分配问题，最终得出数据集规模 D 与模型规模 N 的最优配置为 $N_{opt} \propto C^{0.46}, D_{opt} \propto C^{0.54}$。这一结果表明，**数据集量 D 与模型规模 N 几乎同等重要**，如果总计算预算增加了 10 倍，那么模型规模以及数据规模都应当扩大约 3.16 倍。谷歌后续在 2023 年 5 月发布的 PaLM 2 的技术报告 [2] 中也再次证实了这一观点，进一步强调了数据规模在提升模型性能中的重要性。

此外，Chinchilla 扩展法则进一步提出，理想的数据集大小应当是模型规模的 20 倍。例如，对于一个 7B（70 亿参数）的模型，最理想的训练数据集大小应为 140B（1400 亿）个 Token。但先前很多模型的预训练数据量并不够，例如 OpenAI 的 GPT-3[5] 模型的最大版本有 1750 亿参数，却只用了 3000 亿 Token 进行训练；同样，微软的 MT-NLG[35] 模型拥有 5300 亿参数，而训练用的 Token 数量却只有 2700 亿。因此，DeepMind 推出了数据规模 20 倍于模型规模的 Chinchilla 模型（700 亿参数，1.4 万亿 Token），最终在性能上取得了显著突破。

DeepMind 提出的 Chinchilla 扩展法则是对 OpenAI 先前研究的补充和优化，强调了数据规模在提升模型性能中的重要性，指出**模型规模和数据规模应该以相同的比例增加**，开创了大语言模型发展的一个新方向：不再单纯追求模型规模的增加，而是**优化模型规模与数据规模的比例**。

2.1.2 大数据 + 大模型 → 能力扩展

如图2.2所示，模型训练数据规模以及参数数量的不断提升，不仅带来了上述学习能力的稳步增强，还为大模型“解锁”了一系列**新的能力**[4]，例如上下文学习能力、常识推理能力、数学运算能力、代码生成能力等。值得注意的是，这些新能力并非通过在特定下游任务上通过训练获得，而是随着模型复杂度的提升凭空自然涌现[5]。这些能力因此被称为大语言模型的涌现能力（Emergent Abilities）。

图 2.2: 大语言模型能力随模型规模涌现，图片由 GPT-4o 生成。

涌现能力往往具有突变性和不可预见性。类似于非线性系统中的“相变”，即系统在某个阈值点发生显著变化，这些能力也并没有一个平滑的、逐渐积累的过程，而是在模型达到一定规模和复杂度后，很**突然地显现** [32]。例如，在 GPT 系列的演变中，可以观察到一些较为典型的涌现能力。

- **上下文学习**：上下文学习（In-Context Learning）是指大语言模型在推理过程中，能够利用输入文本的上下文信息来执行特定任务的能力。具备了上下文学习能力的模型，在很多任务中**无需额外的训练**，仅**通过示例或提示**即可理解任务要求并生成恰当的输出。在 GPT 系列中，不同版本的模型在上下文学习能力上有显著差异。早期的 GPT-1 和 GPT-2 在上下文学习方面的能力非常

[4]https://research.google/blog/pathways-language-model-palm-scaling-to-540-billion-parameters-for-breakthrough-performance

[5]https://www.assemblyai.com/blog/emergent-abilities-of-large-language-models

有限，通常无法直接利用上下文信息进行准确的推理和回答。GPT-3 的 130 亿参数版本则在上下文学习方面取得了显著进步，能在提供的上下文提示下完成一些常见任务。然而，对于更加复杂或特定领域的任务，其性能仍有限。具有 1750 亿参数的 GPT-3 最大版本以及后续的 GPT-4 模型展现出强大的上下文理解和学习能力，可以基于少量示例完成各类高度复杂的任务。

- **常识推理**：常识推理（Commonsense Reasoning）能力赋予了大语言模型**基于常识知识和逻辑进行理解和推断**的能力。它包括对日常生活中普遍接受的事实、事件和行为模式的理解，并利用这些知识来回答问题、解决问题和生成相关内容。GPT-1 和 GPT-2 在常识推理方面的能力非常有限，常常会出现错误的推断或缺乏详细的解释。而 GPT-3 的较大版本能够在大多数情况下生成合理和连贯的常识性回答。至于具有 1750 亿参数的 GPT-3 最大版本以及后续的 GPT-4 等模型，则能够在处理高度复杂的常识推理任务时展现逻辑性、一致性和细节丰富性。
- **代码生成**：代码生成（Code Generation）能力允许大语言模型**基于自然语言描述自动生成编程代码**。这包括理解编程语言的语法和语义、解析用户需求、生成相应代码，以及在某些情况下进行代码优化和错误修复。GPT-1 和 GPT-2 仅能生成非常简单的代码片段，但是无法有效理解具体的编程需求。130 亿参数的 GPT-3 模型出现时，已经能很好地处理常见的编程任务和生成结构化代码片段，但在极其复杂或特定领域的任务上仍有限。在参数量达到 1750 亿时，模型则能够处理复杂编程任务，多语言代码生成，代码优化和错误修复等，展示出高质量的代码生成和理解能力。
- **逻辑推理**：逻辑推理（Logical Reasoning）能力使大语言模型能够基于给定信息和规则进行**合乎逻辑的推断和结论**。这包括简单的条件推理、多步逻辑推理、以及在复杂情境下保持逻辑一致性。GPT-1 和 GPT-2 作为早期的生成

预训练模型，在逻辑推理方面的能力非常有限，甚至对于 130 亿参数版本的 GPT-3 模型而言，虽然能处理一部分逻辑推理任务，但在复杂度和精确性上仍存在一定局限性。直到 1750 亿参数版本，GPT-3 才能够处理复杂的逻辑推理任务，生成详细和连贯的推理过程。

- ……

这些涌现能力使得大语言模型可以在不进行专项训练的前提下完成各类任务，但同时也带来了诸多挑战，包括模型的可解释性、信息安全与隐私、伦理和公平性问题，以及对计算资源的巨大需求等。解决这些挑战需要在技术、法律和社会层面进行综合考量，以确保大语言模型的健康发展和可持续进步。

2.2 大语言模型架构概览

在语言模型的发展历程中，Transformer[42] 框架的问世代表着一个划时代的转折点。其独特的自注意力（Self-Attention）机制极大地提升了模型**对序列数据的处理能力**，在**捕捉长距离依赖关系**方面表现尤为出色。此外，Transformer 框架**对并行计算的支持**极大地加速了模型的训练过程。当前，绝大多数大语言模型均以 Transformer 框架为核心，并进一步演化出了三种经典架构，分别是 Encoder-only 架构，Decoder-only 架构以及 Encoder-Decoder 架构。这三种架构在设计和功能上各有不同。本节将简要介绍这三种架构的设计理念与预训练方式，并分析它们之间的区别以及各自的演变趋势。

2.2.1 主流模型架构的类别

本小节将从设计理念、训练方式等角度对 Encoder-only 架构，Decoder-only 架构以及 Encoder-Decoder 架构分别进行简要介绍。

1. Encoder-only 架构

Encoder-only 架构仅选取了 Transformer 中的编码器（Encoder）部分，用于接收输入文本并生成与上下文相关的特征。具体来说，Encoder-only 架构包含三个部分，分别是**输入编码**部分，**特征编码**部分以及**任务处理**部分，具体的模型结构如图2.3所示。其中输入编码部分包含**分词**、**向量化**以及**添加位置编码**三个过程。而特征编码部分则是由多个相同的**编码模块**（Encoder Block）堆叠而成，其中每个编码模块包含**自注意力模块**（Self-Attention）和**全连接前馈模块**。任务处理模块是针对任务需求专门设计的模块，其可以由用户针对任务需求自行设计。Encoder-only 架构模型的预训练阶段和推理阶段在输入编码和特征编码部分是一致的，而任务处理部分则需根据任务的不同特性来进行定制化的设计。

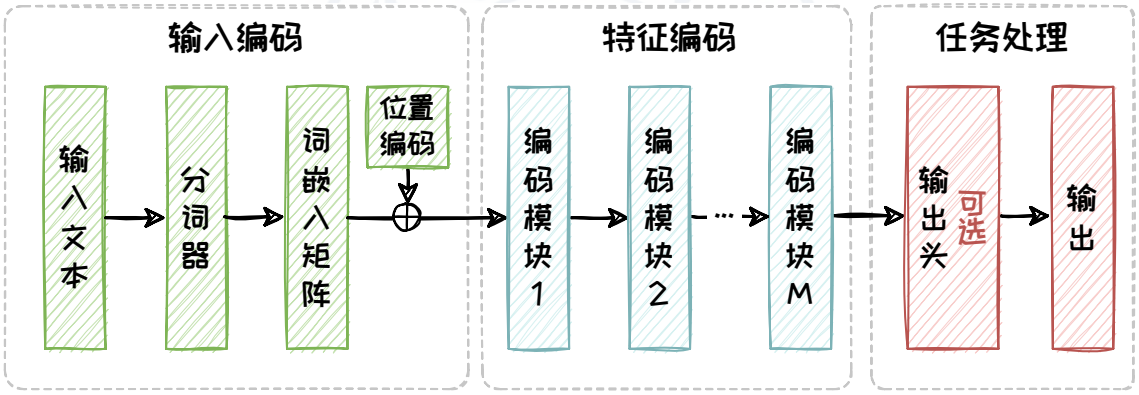


图 2.3: Encoder-only 架构。

在**输入编码部分**，原始输入文本会被分词器（Tokenizer）拆解为 Token 序列，随后通过词表和词嵌入（Embedding）矩阵映射为向量序列，确保文本信息得以数字化表达。具体的过程和细节将在3.1 章节中被具体介绍。接着为了保留文本中单词的顺序信息，每个向量序列会被赋予位置编码（Positional Encoding）。在**特征编码部分**，先前得到的向量序列会依次通过一系列编码模块，这些模块通过自注意力机制和前馈网络进一步提取和深化文本特征。**任务处理部分**在预训练阶段和下游任务适配阶段一般有所差别。在预训练阶段，模型通常使用全连接层作为输出头，

用于完成掩码预测等任务。而在下游任务适配阶段，输出头会根据具体任务需求进行定制。例如，对于情感分析或主题分类等判别任务，只需要添加一个分类器便可直接输出判别结果。但对于文本摘要生成等生成任务，则需要添加一个全连接层，逐个预测后续的 Token。但以这种形式来完成生成任务存在着诸多的限制，例如在每次生成新的 Token 时，都需要重新计算整个输入序列的表示，这增加了计算成本，也可能导致生成的文本缺乏连贯性。本章将在2.3 节中对 Encoder-only 架构进行更具体的介绍。

2. Encoder-Decoder 架构

为了弥补 Encoder-only 架构在文本生成任务上的短板，Encoder-Decoder 架构在其基础上引入了一个解码器（Decoder），并采用交叉注意力机制来实现编码器与解码器之间的有效交互。

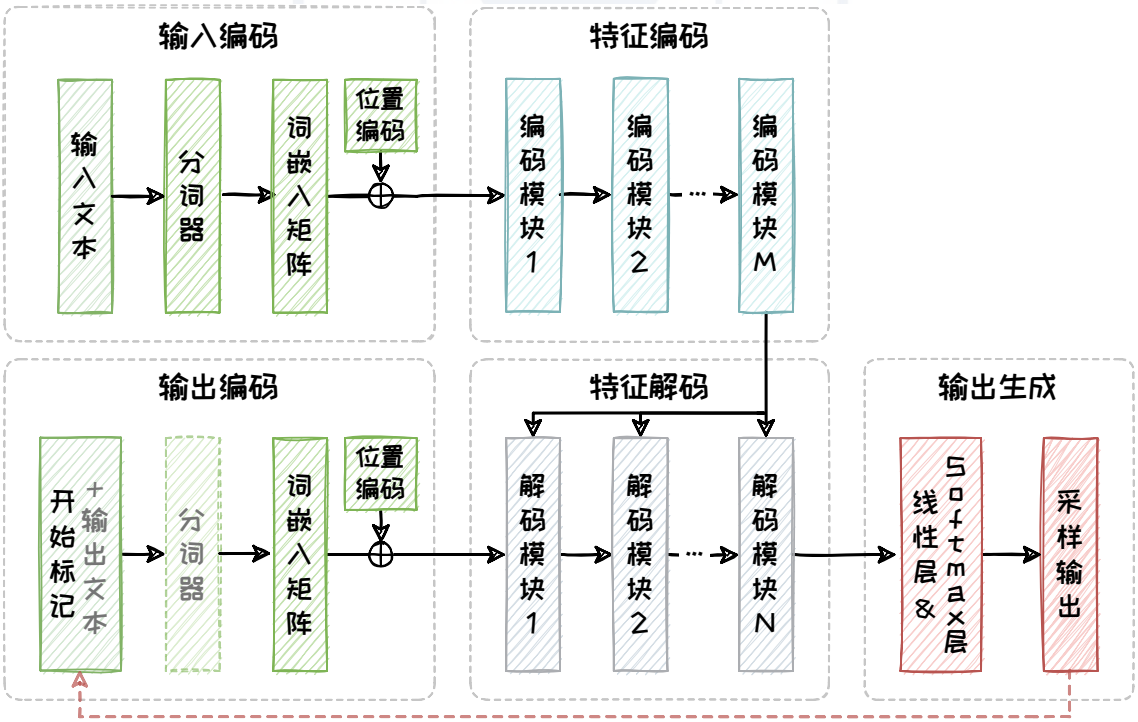


图 2.4: Encoder-Decoder 架构。其中分词器和输出文本只在训练阶段存在，而实现“自回归”的红色虚线只在推理阶段存在。

具体来说，解码器包含了**输出编码**、**特征解码**以及**输出生成**三个部分。其中**输出编码**与编码器中的输入编码结构相同，包含分词、向量化以及添加位置编码三个过程，将原始输入文本转换化为带有位置信息的向量序列。此外，**特征解码**部分与特征编码部分在网络结构上也高度相似，包括掩码自注意力（Masked Self-Attention）模块，交叉注意力模块和全连接前馈模块。其中掩码自注意力模块确保模型只关注上文，不会“预见”未来的信息，从而可以在无“下文泄露”的条件下，进行“自回归”的训练和推理。而交叉注意力模块则负责处理从编码模块向解码模块传递相关信息。**输出生成**部分则由一个线性层以及一个 Softmax 层组成，负责将特征解码后的向量转换为词表上的概率分布，并从这个分布中采样得到最合适的 Token 作为输出。

图2.4展示了 Encoder-Decoder 架构的具体工作流程。在训练阶段，样本中同时包含了输入和真实（Ground Truth）输出文本。其中输入文本首先被输入编码部分转化为向量序列，接着在特征编码模块中被多个堆叠起来的编码模块进一步处理，从而被转化为上下文表示。而输出文本之前会被添加特殊的开始标记 `[START]`，然后在输出编码部分被分词、词嵌入和位置编码处理后，并行输入到特征解码模块中。接着解码模块使用 Teacher Forcing 技术，在每轮预测时，使用真实输出文本中的已知部分作为输入，并结合从最后一个编码块得到的上下文信息，来预测下一个 Token，计算预测的 Token 和真实 Token 之间的损失，通过反向传播更新模型参数。

在推理阶段，由于缺少了真实的输出文本，所以输出序列原始状态只有开始标记 `[START]`，也不再需要分词器。模型需要通过自回归的方式，在每轮采样生成 Token 后，会将其拼接到输出序列中，用于下一轮预测。这个过程循环进行，直到生成特定的结束标记 `[end]` 或达到模型设定的最大输出长度。在这一过程中，由于每轮的输入依赖于上一轮的采样结果，因此只能**一步步地串行输出**。在2.4 节中会针对 Encoder-Decoder 架构进行更具体的介绍。

3. Decoder-only 架构

为了有效缩减模型的规模以及降低整体的计算复杂度，Decoder-only 架构摒弃了 Encoder-Decoder 架构中的编码器部分以及与编码器交互的交叉注意力模块。在这种架构下，模型仅使用解码器来构建语言模型。这种架构利用“自回归”机制，在给定上文的情况下，生成流畅且连贯的下文。

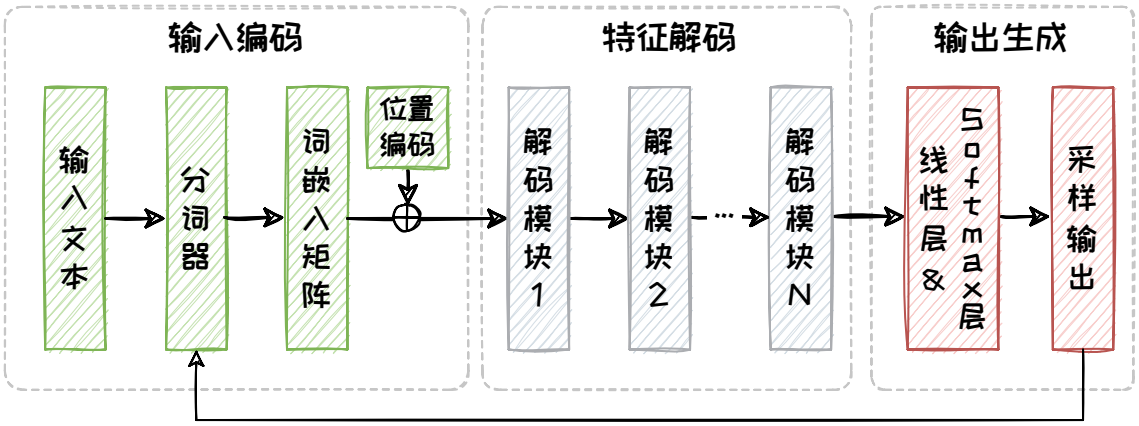


图 2.5: Decoder-only 架构。

Decoder-only 架构同样包含了三个部分，分别是**输入编码**部分、**特征解码**部分以及**输出生成**部分，其具体的模型结构如图2.5所示。Decoder-only 架构的核心特点在于省略了每个编码模块中的交叉注意力子模块，这也是其与传统 Encoder-Decoder 架构中解码器部分的主要区别。在2.5 节中将会对 Decoder-only 架构进行更具体的介绍。

2.2.2 模型架构的功能对比

上述的 Encoder-only、Encoder-Decoder 和 Decoder-only 这三种模型架构虽然都源自于 Transformer 框架，但他们在注意力矩阵上有着显著区别，这也造就了他们在功能以及最终适用任务上的不同。接下来将针对注意力矩阵以及适用任务两个方面对这三种架构的主要区别进行分析。

1. 注意力矩阵

注意力矩阵（Attention Matrix）是 Transformer 中的核心组件，用于计算输入序列中各个 Token 之间的依赖关系。通过注意力机制，模型可以在处理当前 Token 时，灵活地关注序列中其他 Token 所携带的信息，决定了在这一过程中哪些 Token 能够相互影响。如图2.6所示，三种架构在注意力矩阵上有着显著差异。

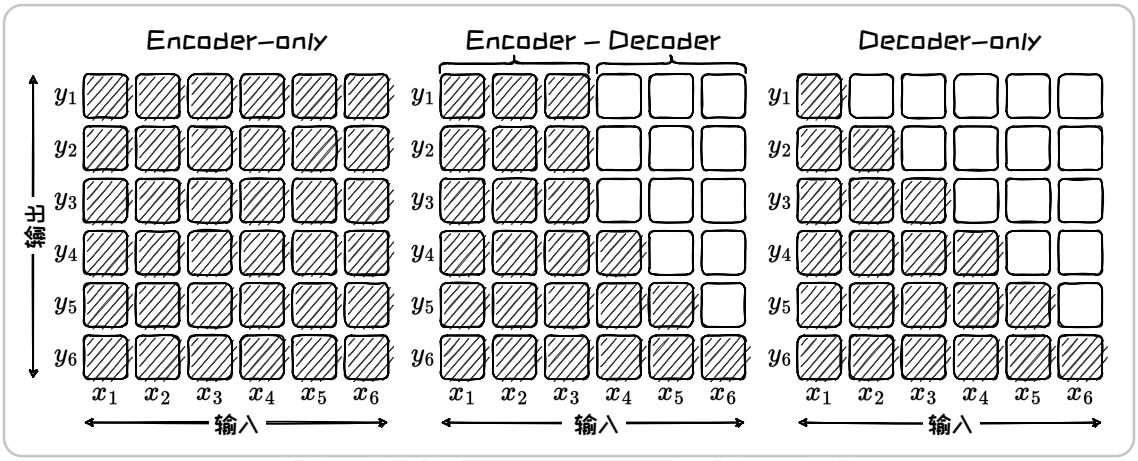


图 2.6: 三种架构的注意力矩阵。

Encoder-only 架构中的注意力矩阵来自于自注意力模块，用于捕捉输入序列中各个 Token 之间的关系。Encoder-only 架构的注意力矩阵呈现出“完全”的注意力，即对于每个 Token 的理解都依赖于整个输入序列中的所有 Token。例如，在将输入 Token x_i 转换为上下文向量 y_i 的过程中，模型能够综合利用从 $x_1 \sim x_n$ 的所有输入信息，这就是所谓的**双向注意力机制**。在这种双向注意力机制的作用下，模型能够同时利用前后文信息，深入理解复杂的语义联系和上下文依赖。

Encoder-Decoder 架构中的注意力矩阵较为复杂，它结合了编码器的自注意力、解码器的掩码自注意力以及交叉注意力三种机制。编码器的自注意力矩阵与 Encoder-only 架构类似，用于生成输入序列的全面上下文表示，呈现“完全”的注意力。而解码器的掩码自注意力矩阵则呈现出“下三角”的注意力，确保在生成当前 Token 时，模型只关注之前生成的 Token。此外，交叉注意力机制允许解码器始

终能够动态地参考编码器生成的完整上下文表示，确保输出与输入序列高度相关且连贯。例如，在编码器将输入 x_i 转化为上下文向量时，可以利用从 $x_1 \sim x_n$ 的所有输入信息；当解码器在生成 Token y_i 的时候，可以参考由 $x_1 \sim x_n$ 转化得到的上下文向量以及先前生成的 Token 序列 $y_1 \sim y_{i-1}$ 的相关信息。

Decoder-only 架构中的注意力矩阵来自于掩码自注意力模块，其特点是呈现出“下三角”的注意力模式。这意味着在预测当前 Token 时，模型只能依赖于已经生成的历史 Token 信息，体现了**单向注意力机制**。例如，在生成 Token y_i 的时候，模型只能考虑先前 $y_1 \sim y_{i-1}$ 的信息，这样的设计确保了生成过程的顺序性和文本的连贯性。

2. 适用任务

由于各自独特的模型设计以及注意力矩阵上的差异，在同等参数规模下，这三种架构的模型在适用任务上也都各有倾向。

Encoder-only 架构中的双向注意力机制允许模型在预测每个 Token 时都充分考虑序列中的前后文信息，捕捉丰富的语义和依赖关系。因此，Encoder-only 架构的模型特别适合于**自然语言理解**（Natural Language Understanding, NLU）任务，如情感分析或文本分类等判别任务。然而，由于缺少解码器组件，Encoder-only 架构的模型无法直接生成所需目标序列，因此在自然语言生成任务（Natural Language Generation, NLG）上可能表现不如专门设计的生成模型。

Encoder-Decode 架构在 Encoder-only 架构的基础上添加了解码器，使模型能够基于编码器输出的上下文表示逐步生成输出序列。这种编码器和解码器的结合，使得模型可以有效地处理复杂的输入条件，并生成相关且连贯的高质量内容。因此，Encoder-Decoder 架构的模型非常适合于处理各种复杂的**有条件生成任务**，例如机器翻译、文本摘要和问答系统等需要同时理解输入并生成相应输出的场景。但新添加解码器也同样带来了模型规模以及计算量庞大的问题。

Decoder-only 架构进一步删除了 Encoder-Decoder 架构中的编码器部分，从而降低了模型本身的计算复杂度。这一架构的模型使用掩码（Mask）操作确保在每个时间步生成当前 Token 时只能访问先前的 Token，并通过自回归生成机制，从起始 Token 开始逐步生成文本。大规模预训练数据的加持使得 Decoder-only 架构的模型能够生成高质量、连贯的文本，在自动故事生成、新闻文章生成此类不依赖于特定的输入文本的**无条件文本生成任务**中表现出色。然而，在模型规模有限的情况下（例如 GPT-1 以及 GPT-2 等模型），由于缺乏编码器提供的双向上下文信息，Decoder-only 架构的模型在理解复杂输入数据时**存在一定局限性**，表现可能不如 Encoder-Decoder 架构。

在不同的历史阶段，三种模型架构分别展现了自身的优势。随着模型规模以及数据规模的显著增长，Decoder-only 架构的模型逐渐占据上风，以其**强大的任务泛化**性能展现出成为“大一统”的架构的潜力。当前，以 GPT-3、GPT-4 等为代表的大型 Decoder-only 语言模型，已经发展出了与人类媲美甚至超越人类的记忆、推理以及执行复杂任务的能力。

2.2.3 模型架构的历史演变

随着时间的流逝，我们见证了上述三种架构的演变和流行趋势的更替。在大语言模型的早期发展阶段（2018 年左右），BERT 和 GPT-1 分别作为 Encoder-only 和 Decoder-only 架构的代表几乎同时出现。但受限于当时的模型参数规模，BERT **强大的上下文理解能力**比 GPT-1 **初阶的文本生成能力**更为亮眼。使得 Encoder-only 架构得到了更为广泛的探索和应用，而 Decoder-only 架构吸引得关注则相对关注较少。然而，随着用户对**机器翻译等生成任务需求的增加**，缺乏解码组件的 Encoder-only 架构逐渐无法满足直接生成的需求，因而被逐渐“冷落”。同时，2019 年末诞生了一众 Encoder-Decoder 架构的模型，由于其能够**有效处理序列到序列的生成任**

务，逐渐成为主流。到了 2019 年末，随着算力资源的急速发展，研究者开始寻求不断提升参数量来激发更强的生成能力。得益于其参数易扩展性，Decoder-only 架构下的模型参数量急剧扩充，文本生成能力大幅提升。自 2021 年之后，在 GPT-3 等模型的推动下，Decoder-only 架构开始占据主流，甚至逐渐**主导了大语言模型的发展**。尽管如此，Encoder-Decoder 架构也仍然活跃在开源社区中，不断被探索和改进。至于 Encoder-only 架构，在 BERT 带来最初的爆炸性增长之后，其关注度有所下降，但也**仍然在部分判别任务中发挥着重要作用**，例如文本分类、情感分析、命名实体识别等。总的来讲，大语言模型的主流架构经历了从 Encoder-only 到 Encoder-Decoder，再到 Decoder-only 的发展过程，而引发这种更迭趋势的原因可能是模型本身生成能力以及计算效率上的差异。

Encoder-only 架构专注于对输入数据进行深入的表示和理解，这使得它在理解型任务中表现出色。但当涉及到文本生成任务时，其**生成能力则不尽如人意**。由于**缺少专门用于生成输出的组件**，它需要通过迭代进行掩码预测来生成文本，这需要模型进行多次前向传播以逐步填充文本中的掩码部分，从而导致计算资源的大量消耗。在当今对人工智能内容生成（AIGC）需求不断增长的背景下，这种架构的应用受到了一定的限制。而 **Encoder-Decoder 架构**使用编码器来处理输入，并使用解码器来生成输出，能够处理复杂的序列到序列任务。但相较于后续崛起的 Decoder-only 架构，其模型**参数规模更为庞大**，随之带来的是**显著增加的计算复杂度**。特别是在处理长序列数据时，编码器和解码器均面临**沉重的计算负担**，这在一定程度上限制了模型的应用效率。相比之下，**Decoder-only 架构**通过仅使用解码器来优化计算流程，显著简化了模型结构。它采用自回归生成策略，逐步构建输出文本，每一步的生成过程仅需考虑已经生成的文本部分，而不是整个输入序列。这样的设计**减少了模型的参数量和计算需求**，提高了模型本身的可扩展性，因此非常适合大量文本生成的场景，在 AIGC 的应用场景下得到了广泛的应用。

这三种架构的发展不仅推动了自然语言处理技术的进步，也为各行各业带来了深远的影响，从提升搜索引擎的准确性到开发更加智能的对话系统。随着研究的深入，未来的语言模型有望在更具优势的架构下变得更加强大和灵活，并在更多领域发挥作用。后续章节将进一步介绍这三种架构及其对应的几种经典模型。

2.3 基于 Encoder-only 架构的大语言模型

Encoder-only 架构凭借着其独特的双向编码模型在自然语言处理任务中表现出色，尤其是在各类需要深入理解输入文本的任务中。本章将对双向编码模型以及几种较为典型的 Encoder-only 架构模型进行介绍。

2.3.1 Encoder-only 架构

Encoder-only 架构的核心在于能够覆盖输入所有内容的**双向编码模型**（Bidirectional Encoder Model）。在处理输入序列时，双向编码模型融合了从左往右的正向注意力以及从右往左的反向注意力，能够充分捕捉每个 Token 的上下文信息，因此也被称为具有**全面的注意力机制**。得益于其上下文感知能力和动态表示的优势，双向编码器显著提升了自然语言处理任务的性能。

不同于先前常用的 Word2Vec 和 GloVe 此类，为每个词提供一个**固定向量表示**的**静态编码方式**，双向编码器为每个词生成**动态的上下文嵌入**（Contextual Embedding），这种嵌入依赖于输入序列的具体上下文，使得模型能够更加精准地理解词与词之间的依赖性和语义信息，有效处理词语的多义性问题。这种动态表示使得双向编码器**在句子级别的任务上表现出色**，显著超过了静态词嵌入方法的性能 [20]。

Encoder-only 架构基于双向编码模型，选用了 Transformer 架构中的编码器部分。虽然 Encoder-only 模型不直接生成文本，但其生成的上下文嵌入对于深入理

解输入文本的结构和含义至关重要。这些模型在需要深度理解和复杂推理的自然语言处理任务中展现出卓越的能力，成为了自然语言处理领域的宝贵工具。当前，BERT[11] 及其变体，如 RoBERTa[23]、ALBERT[17] 等，都是基于 Encoder-only 架构的主流大语言模型。接下来将对这些模型进行介绍。

2.3.2 BERT 语言模型

BERT（Bidirectional Encoder Representations from Transformers）是一种基于 Encoder-only 架构的预训练语言模型，由 Google AI 团队于 2018 年 10 月提出。作为早期 Encoder-only 架构的代表，BERT 在自然语言处理领域带来了突破性的改进。其核心创新在于通过双向编码模型深入挖掘文本的上下文信息，从而为各种下游任务提供优秀的上下文嵌入。本节将对 BERT 模型的结构、预训练方式以及下游任务进行介绍。

1. BERT 模型结构

BERT 模型的结构与 Transformer 中的编码器几乎一致，都是由多个编码模块堆叠而成，每个编码模块包含一个多头自注意力模块和一个全连接前馈模块。根据参数量的不同，BERT 模型共有 BERT-Base 和 BERT-Large 两个版本。其中 **BERT-Base** 由 12 个编码模块堆叠而成，隐藏层维度为 768，自注意力头的数量为 12，**总参数数量为 1.1 亿**；**BERT-Large** 由 24 个编码模块堆叠而成，隐藏层维度为 1024，自注意力头的数量为 16，**总参数数量约为 3.4 亿**。

2. BERT 预训练方式

BERT 使用小说数据集 BookCorpus[46]（包含约 8 亿个 Token）和英语维基百科数据集[6]（包含约 25 亿个 Token）进行预训练，总计约 33 亿个 Token，总数据量达到了 15GB 左右。在预训练任务上，BERT 开创性地提出了掩码语言建模（Masked

[6]https://dumps.wikimedia.org/

Language Model, MLM）和下文预测（Next Sentence Prediction, NSP）两种任务来学习生成上下文嵌入。其完整的预训练流程如图2.7所示。

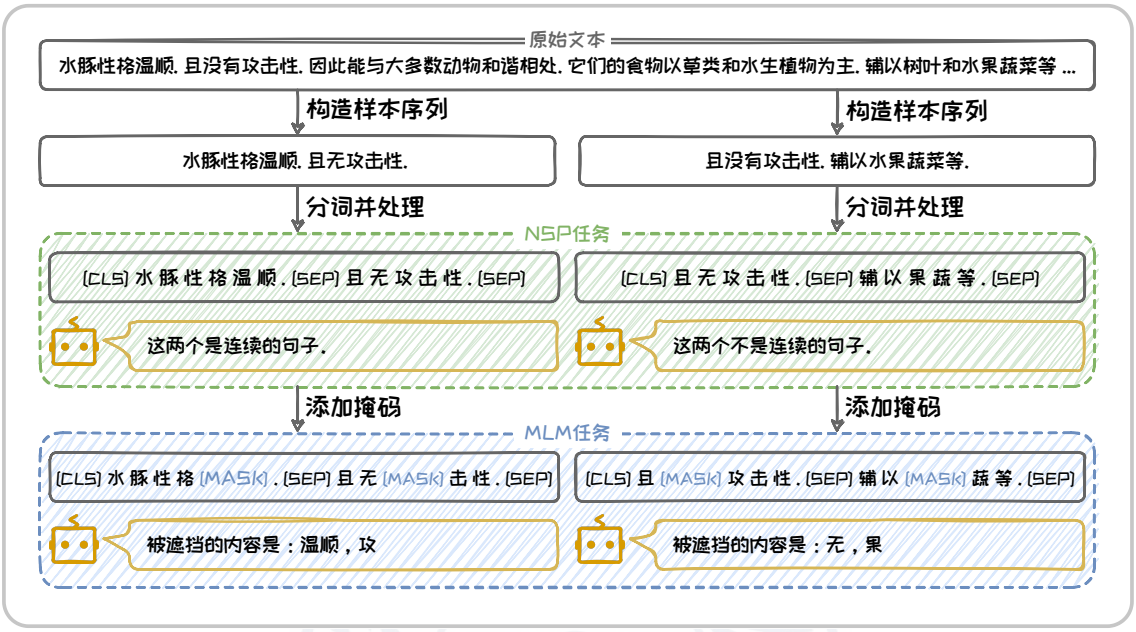


图 2.7: BERT 预训练任务。

具体而言，BERT 先基于给定的原始文本构造多个样本序列，每个样本序列由原始文本中的两个句子组成，这两个句子有 50% 的概率是来自原文的连续句子，另外 50% 的概率是随机挑选的两个句子。随后，对构造出来的样本序列进行分词，并在序列的开头添加特殊标签 `[CLS]`，在每个句子的结尾添加特殊标签 `[SEP]`。其中 `[CLS]` 标签用于聚合整个序列的信息，而 `[SEP]` 标签则明确句子之间的界限。

接着，BERT 利用处理后的序列进行下文预测任务，利用模型判断样本序列中的两个句子**是否为连续的**。这一任务训练 BERT 识别和理解句子之间的关系，捕捉句子层面的语义特征。这对于理解文本的逻辑流、句子之间的关联性有很大帮助，特别是在问答和自然语言推理等需要理解文档层次结构的自然语言处理 (NLP) 任务中。

最后，BERT 随机选择样本序列中大约 15% 的 Token 进行遮掩，将其替换为特殊标签 `[MASK]` 或者随机单词。模型需要**预测这些被替换的 Token 的原始内容**。这

个过程类似于**完型填空**，要求模型根据周围的上下文信息来推断缺失的 Token。预测过程使用的交叉熵损失函数驱动了 BERT 模型中参数的优化，使其能够学习到文本的双向上下文表示。值得注意的是，在 MLM 任务的训练过程中，BERT 仅针对那些被随机替换的 Token 进行学习，即只计算这些 Token 的预测损失来更新模型参数。

通过这两种预训练任务的结合，使 BERT 在理解语言的深度和广度上都有显著提升。BERT 不仅能够捕捉到 Token 的细粒度特征，还能够把握长距离的依赖关系和句子间的复杂联系，为各种下游任务提供了坚实的语言理解基础。

3. BERT 下游任务

BERT 可以应用于各种自然语言处理任务，包括但不限于文本分类（如情感分析）、问答系统、文本匹配（如自然语言推断）和语义相似度计算等。然而，由于 BERT 的输出是输入中所有 Token 的向量表示，因此总长度不固定，无法直接应用于各类下游任务。为了解决这一问题，BERT 设计了 `[CLS]` 标签来提取**整个输入序列的聚合表示**。`[CLS]` 标签是专门为分类和汇总任务设计的特殊标记。其全称是“Classification Token”，即分类标记。通过注意力机制，`[CLS]` 标签汇总整个输入序列的信息，生成一个固定长度的向量表示，从而实现对所有 Token 序列信息的概括，便于处理各种下游任务。

在**文本分类任务**中，可以将输出中 `[CLS]` 标签对应的向量提取出来，传递给**一个全连接层**，从而用于分类，例如判断整个句子的情绪是积极、消极或是中立的。在**问答系统任务**中，需要输入问题以及一段相关的文本，即“`[CLS]` 问题 `[SEP]` 文本 `[SEP]`”。最终同样提取出 `[CLS]` 标签的对应向量，并传递给**两个全连接层**，用于判断答案是否存在于相关文本中。如果存在，这两个全连接层分别用于输出答案的起始和结束位置。通过这种方式，BERT 能够从提供的段落中准确提取出问题的答案。在**语义相似度任务**中，需要计算两段或者多段文本之间的语义相似度。在

这一任务中，可以通过构造“[CLS] 文本 1[SEP] 文本 2[SEP]”的方式，结合一**个线性层**来直接输出两个文本之间的相似度；也可以**不添加额外的组件**，直接提取 [CLS] 标签对应的向量，再利用额外的相似度度量方法（例如余弦相似度）来计算多段文本之间的相似度。

BERT 通过双向编码以及特定的预训练任务，显著提升了自然语言处理任务的性能。其在多种任务中的应用都展示了强大的泛化能力和实用性，不仅为学术研究提供了新的基准，还在实际应用中得到广泛采用，极大推动了自然语言处理技术的发展。

2.3.3 BERT 衍生语言模型

BERT 的突破性成功还催生了一系列相关衍生模型，这些模型继承了 BERT 双向编码的核心特性，并在其基础上进行了改进和优化，以提高模型性能或者模型效率，在特定任务和应用场景中展现出了卓越的性能。较为代表性的衍生模型为 RoBERTa[23]、ALBERT[17] 以及 ELECTRA[9] 等，接下来将对这三种模型分别展开介绍。

1. RoBERTa 语言模型

RoBERTa（Robustly Optimized BERT Pretraining Approach）由 Facebook AI（现更名 Meta）研究院于 2019 年 7 月提出，旨在解决 BERT 在**训练程度上不充分**这一问题，以**提升预训练语言模型的性能**。RoBERTa 在 BERT 的基础上采用了**更大的数据集**（包括更多的英文书籍、维基百科和其他网页数据）、**更长的训练时间**（包括更大的批次大小和更多的训练步数）以及**更细致的超参数调整**（包括学习率、训练步数等的设置）来优化预训练的过程，从而提高模型在各种自然语言处理任务上的性能和鲁棒性。接下来将对 RoBERTa 模型的结构、预训练方式以及下游任务进行介绍。

(1) RoBERTa 模型结构

RoBERTa 在结构上与 BERT 基本一致，基于多层堆叠的编码模块，每个编码模块包含多头自注意力模块和全连接前馈模块。RoBERTa 同样有两个版本，分别是 RoBERTa-Base 和 RoBERTa-Large。其中 **RoBERTa-Base** 与 BERT-Base 对标，由 12 个编码模块堆叠而成，其中隐藏层维度为 768，自注意力头的数量为 12，**总参数数量约为 1.2 亿**；**RoBERTa-Large** 则与 BERT-Large 对标，由 24 个编码模块堆叠而成，其中隐藏层维度为 1024，自注意力头的数量为 16，**总参数数量约为 3.5 亿**。

(2) RoBERTa 预训练方式

在预训练语料库的选择上，RoBERTa 在 BERT 原有的小说数据集 BookCorpus[46]（包含约 8 亿个 Token）以及英语维基百科数据集 [7]（包含约 25 亿个 Token）的基础上，添加了新闻数据集 CC-News [8]（包含约 76GB 的新闻文章）、网页开放数据集 OpenWebText [9]（包含约 38GB 的网页文本内容）以及故事数据集 Stories[41]（包含约 31GB 的故事文本），总数据量达到约 160GB。

至于具体的预训练任务，RoBERTa 移除了 BERT 中的下文预测任务，并将 BERT 原生的静态掩码语言建模任务更改为**动态掩码语言建模**。具体而言，BERT 在数据预处理期间对句子进行掩码，随后在每个训练 epoch 中，掩码位置不再变化。而 RoBERTa 则将训练数据复制成 10 个副本，分别进行掩码。在同样训练 40 个 epoch 的前提下，BERT 在其静态掩码后的文本上训练了 40 次，而 RoBERTa 将 10 个不同掩码后的副本分别训练了 4 次，从而增加模型训练的多样性，有助于模型学习到更丰富的上下文信息。

这些改进使得 RoBERTa 在理解上下文和处理长文本方面表现出色，尤其是在捕捉细微的语义差异和上下文依赖性方面。

[7] https://dumps.wikimedia.org

[8] http://web.archive.org/save/http://commoncrawl.org/2016/10/newsdataset-available

[9] http://Skylion007.github.io/OpenWebTextCorpus

2. ALBERT 语言模型

ALBERT（A Lite BERT）是由 Google Research 团队于 2019 年 9 月提出的轻量级 BERT 模型，旨在通过参数共享和嵌入分解技术来减少模型的参数量和内存占用，从而**提高训练和推理效率**。BERT-Base 模型有着 1.1 亿个参数，而 BERT-Large 更是有着 3.4 亿个参数，这使得 BERT 不仅难以训练，推理时间也较长。ALBERT 在设计过程通过参数因子分解技术和跨层参数共享技术**显著减少了参数的数量**。接下来将对 ALBERT 模型的结构、预训练方式以及下游任务进行介绍。

(1) ALBERT 模型结构

ALBERT 的结构与 BERT 以及 RoBERTa 都类似，由多层堆叠的编码模块组成。但是 ALBERT 通过**参数因子分解**以及**跨层参数共享**，在相同的模型架构下，显著减少了模型的参数量。下面将参数因子分解和跨层参数共享分别展开介绍。

参数因子分解

在 BERT 中，Embedding 层的输出向量维度 E 与隐藏层的向量维度 H 是一致的，这意味着 Embedding 层的输出直接用作后续编码模块的输入。具体来说，BERT-Base 模型对应的词表大小 V 为 3,0000 左右，并且其隐藏层的向量维度 H 设置为 768。因此，BERT 的 Embedding 层需要的参数数量是 $V \times H$，大约为 2,304 万。

相比之下，ALBERT 将 Embedding 层的矩阵先进行分解，将词表对应的独热编码向量通过一个低维的投影层下投影至维度 E，再将其上投影回隐藏状态的维度 H。具体来说，ALBERT 选择了一个较小的 Embedding 层维度，例如 128，并将参数数量拆解为 $V \times E + E \times H$。按照这个设计，ALBERT 的 Embedding 层大约需要 394 万个参数，大约是 BERT 参数数量的六分之一。对于具有更大隐藏层向量维度 H 的 Large 版本，ALBERT 节省参数空间的优势更加明显，能够将参数量压缩至 BERT 的八分之一左右。

跨层参数共享

以经典的 BERT-Base 模型为例，模型中共有 12 层相同架构的编码模块，所有 Transformer 块的参数都是独立训练的。ALBERT 为了降低模型的参数量，提出了跨层参数共享机制，只学习第一层编码模块的参数，并将其直接共享给其他所有层。该机制在一定程度上牺牲了模型性能，但显著提升了参数存储空间的压缩比，从而实现了更高效的资源利用。

基于参数因子分解和跨层参数共享，ALBERT 共提出了四个版本的模型，分别是 ALBERT-Base、ALBERT-Large、ALBERT-XLarge 以及 ALBERT-XXLarge。其中 **ALBERT-Base** 与 BERT-Base 对标，由 12 个编码模块堆叠而成，中间嵌入分解维度为 128，隐藏层维度为 768，自注意力头的数量为 12，**总参数数量约为 0.12 亿**；**ALBERT Large** 与 BERT-Large 对标，由 24 个编码模块堆叠而成，中间嵌入分解维度为 128，隐藏层维度为 1024，自注意力头的数量为 16，**总参数数量约为 0.18 亿**；**ALBERT X-Large** 由 12 个编码模块堆叠而成，中间嵌入分解维度为 128，隐藏层维度为 2048，自注意力头的数量为 16，**总参数数量约为 0.6 亿**；**ALBERT XX-Large** 由 12 个编码模块堆叠而成，中间嵌入分解维度为 128，隐藏层维度为 4096，自注意力头的数量为 64，**总参数数量约为 2.2 亿**。

(2) ALBERT 预训练方式

ALBERT 使用与 BERT 完全一致的数据集来进行预训练，即小说数据集 BookCorpus[46]（包含约 8 亿个 Token）以及英语维基百科数据集 [10]（包含约 25 亿个 Token），总计 33 亿个 Token，约 15GB 数据量。另外，在预训练任务的选择上，ALBERT 保留了 BERT 中的掩码语言建模任务，并将下文预测任务替换为**句序预测**（Sentence Order Prediction, SOP），如图2.8所示。具体而言，ALBERT 从文本中选择连续的两个句子，将这两个句子直接拼接起来，或是先将这两个句子的顺序

[10]https://dumps.wikimedia.org

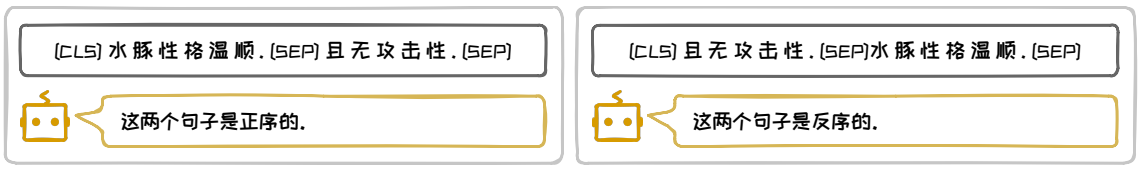


图 2.8: ALBERT 句序预测任务。

翻转后再进行拼接，并将拼接后的内容作为输入样本，而模型需要预测该样本中的两个句子是正序还是反序。

与 BERT 相比，ALBERT 通过创新的参数共享和参数因子分解技术，在较好地保持原有性能的同时显著减少了模型的参数数量，这使得它在资源受限的环境中更加实用，处理大规模数据集和复杂任务时更高效，并降低了模型部署和维护的成本。

3. ELECTRA 语言模型

ELECTRA（Efficiently Learning an Encoder that Classifies Token Replacements Accurately）[9] 是由 Google Brain 和斯坦福大学的研究人员于 2020 年 3 月提出的另一种 BERT 变体，旨在解决大规模预训练语言模型中的**效率和可扩展性问题**。通过使用**生成器-判别器架构**，ELECTRA 能**够更高效地利用预训练数据**，提高了模型在下游任务中的表现。

(1) ELECTRA 预训练方式

在模型结构上，ELECTRA 在 BERT 原有的掩码语言建模基础上结合了生成对抗网络（Generative Adversarial Network, GAN）的思想，采用了一种生成器-判别器结构。具体来说，ELECTRA 模型包含一个生成器和一个判别器，其中生成器（Generator）是一个能进行掩码预测的模型（例如 BERT 模型），负责将掩码后的文本恢复原状。而判别器（Discriminator）则使用替换词检测（Replaced Token Detection, RTD）预训练任务，负责检测生成器输出的内容中的每个 Token 是否是原文中的内容。其完整的流程如图2.9所示。

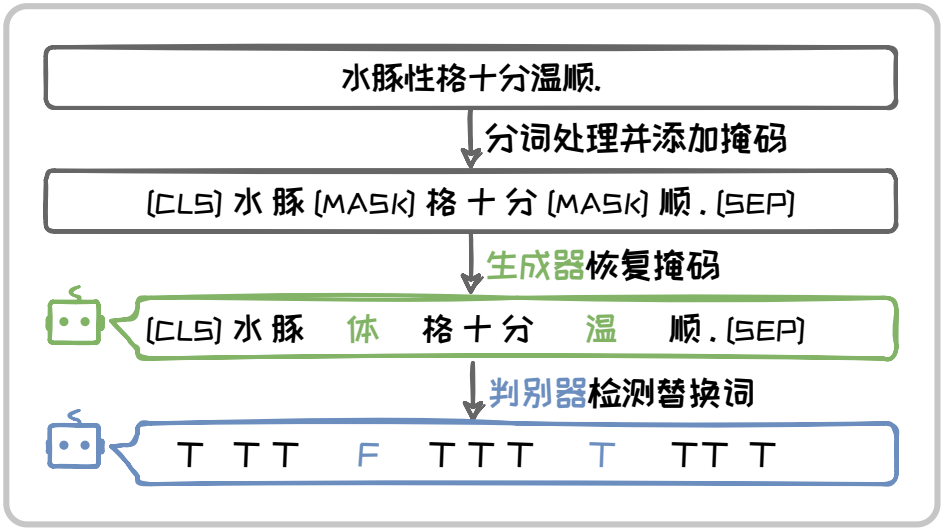


图 2.9: ELECTRA 预训练任务。

(2) ELECTRA 模型结构

根据生成器与判别器的不同规模，ELECTRA 共提出了三个版本的模型，分别是 ELECTRA-Small、ELECTRA-Base 以及 ELECTRA-Large。其中 **ELECTRA-Small** 中的生成器与判别器都由 12 个编码模块堆叠而成，隐藏层维度为 256，自注意力头数量为 4，因此生成器与判别器的参数量均为 0.14 亿左右，**总参数数量约为 0.28 亿**；**ELECTRA-Base** 中的生成器与判别器都由 12 个编码模块堆叠而成，隐藏层维度为 768，自注意力头数量为 12，因此生成器与判别器的参数量均为 1.1 亿左右，**总参数数量约为 2.2 亿**；**ELECTRA-Large** 中的生成器与判别器都由 24 个编码模块堆叠而成，隐藏层维度为 1024，自注意力头数量为 16，因此生成器与判别器的参数量均为 3.3 亿左右，**总参数数量约为 6.6 亿**。

其中，ELECTRA-Small 和 ELECTRA-Base 使用与 BERT 一致的数据集来进行预训练，共包含 33 亿个 Token。而 ELECTRA-Large 则使用了**更多样化的训练数据**，包括大规模网页数据集 ClueWeb [11]、CommonCrawl [12]以及大型新闻文本数据集

[11] https://lemurproject.org/clueweb09.php

[12] https://commoncrawl.org

Gigaword [13]，最终将数据量扩充到了 330 亿个 Token，从而帮助模型学习更广泛的语言表示。

另外，在 BERT 中，只有 15% 的固定比例 Token 被掩码，模型训练的内容也仅限于这 15% 的 Token。但是在 ELECTRA 中，判别器会判断生成器输出的所有 Token 是否被替换过，因此能够更好地学习文本的上下文嵌入。

不同于 RoBERTa 和 ALBERT 主要在模型结构以及预训练数据规模上进行优化，ELECTRA 在 BERT 的基础上引入了新的生成器-判别器架构，通过替换语言模型任务，显著提升了训练效率和效果，同时提高了模型在下游任务中的表现。

上述基于 Encoder-only 架构的大语言模型在文本分类、情感分析等多个自然语言处理任务中取得了良好效果。表2.1从模型参数量及预训练语料等方面对上述模型进行总结。可以看出这些经典模型参数大小止步于 6.6 亿，预训练任务也主要服务于自然语言理解。这些模型没有继续寻求参数量上的突破，并且通常只专注于判别任务，难以应对生成式任务，因此在当前愈发热门的生成式人工智能领域中可以发挥的作用相对有限。

表 2.1: Encoder-only 架构代表模型参数和语料大小表。

模型	发布时间	参数量（亿）	语料规模	预训练任务
BERT	2018.10	1.1, 3.4	约 15GB	MLM+NSP
RoBERTa	2019.07	1.2, 3.5	160GB	Dynamic MLM
ALBERT	2019.09	0.12, 0.18, 0.6, 2.2	约 15GB	MLM+SOP
ELECTRA	2020.03	0.28, 2.2, 6.6	约 20-200GB	RTD

2.4 基于 Encoder-Decoder 架构的大语言模型

Encoder-Decoder 架构在 Encoder-only 架构的基础上引入 Decoder 组件，以完成机器翻译等**序列到序列**（Sequence to Sequence, Seq2Seq）任务。本节将对 Encoder-Decoder 架构及其代表性模型进行介绍。

[13] https://catalog.ldc.upenn.edu/LDC2011T07

2.4.1 Encoder-Decoder 架构

Encoder-Decoder 架构主要包含编码器和解码器两部分。该架构的详细组成如图2.10所示。其中，编码器部分与 Encoder-only 架构中的编码器相同，由多个编码模块堆叠而成，每个编码模块包含一个自注意力模块以及一个全连接前馈模块。模型的输入序列在通过编码器部分后会被转变为固定大小的上下文向量，这个向量包含了输入序列的丰富语义信息。解码器同样由多个解码模块堆叠而成，每个解码模块由一个带掩码的自注意力模块、一个交叉注意力模块和一个全连接前馈模块组成。其中，带掩码的自注意力模块引入掩码机制防止未来信息的“泄露”，确保解码过程的自回归特性。交叉注意力模块则实现了解码器与编码器之间的信息交互，对于生成与输入序列高度相关的输出至关重要。

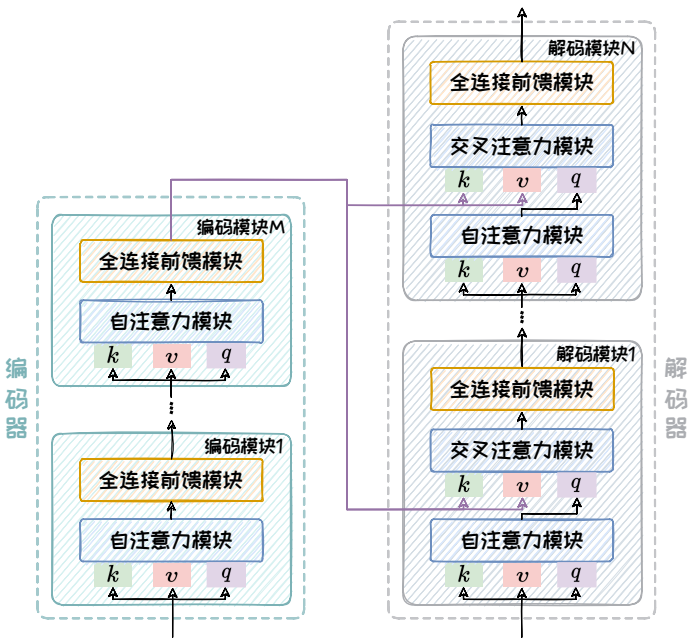


图 2.10: Encoder-Decoder 架构。

自注意模块在编码器和解码器中的注意力目标不同的。在编码器中，我们需要对输入序列的**上下文**进行“通盘考虑”，所以采用双向注意力机制以全面捕捉上下文信息。但在解码器中，自注意力机制则是单向的，仅以**上文**为条件来解码得到

下文，通过掩码操作避免解码器“窥视”未来的信息。交叉注意力通过将解码器的查询（query）与编码器的键（key）和值（value）相结合，实现了两个模块间的有效信息交流。

通过自注意力和交叉注意力机制的结合，Encoder-Decoder 架构能够**高效地编码输入信息**并**生成高质量的输出序列**。自注意力机制确保了输入序列和生成序列内部的一致性和连贯性，而交叉注意力机制则确保了解码器在生成每个输出 Token 时都能参考输入序列的全局上下文信息，从而生成与输入内容高度相关的结果。在这两个机制的共同作用下，Encoder-Decoder 架构不仅能够深入理解输入序列，还能够根据不同任务的需求灵活生成长度适宜的输出序列，在机器翻译、文本摘要、问答系统等任务中得到了广泛应用，并取得了显著成效。本节将介绍两种典型的基于 Encoder-Decoder 架构的代表性大语言模型：T5[29] 和 BART[18]。

2.4.2 T5 语言模型

自然语言处理涵盖了语言翻译、文本摘要、问答系统等多种任务。通常，每种任务都需要对训练数据、模型架构和训练策略进行定制化设计。这种定制化设计不仅耗时耗力，而且训练出的模型难以跨任务复用，导致开发者需要不断“重复造轮子”。为了解决这一问题，Google Research 团队在 2019 年 10 月提出了一种基于 Encoder-Decoder 架构的大型预训练语言模型 T5（Text-to-Text Transfer Transformer），其采用了统一的文本到文本的转换范式来处理多种任务。下面分别从模型结构、预训练方式以及下游任务三个方面对 T5 模型进行介绍。

1. T5 模型结构

T5 模型的核心思想是**将多种 NLP 任务统一到一个文本转文本的生成式框架中**。在此统一框架下，T5 通过不同的输入前缀来指示模型执行不同任务，然后生成相应的任务输出，正如图2.11所示。这种方法可以视为早期的提示（Prompt）技

术的运用，通过构造合理的输入前缀，T5 模型能够引导自身针对特定任务进行优化，而无需对模型架构进行根本性的改变。这种灵活性和任务泛化能力显著提高了模型的实用性，使其能够轻松地适应各类新的 NLP 任务。

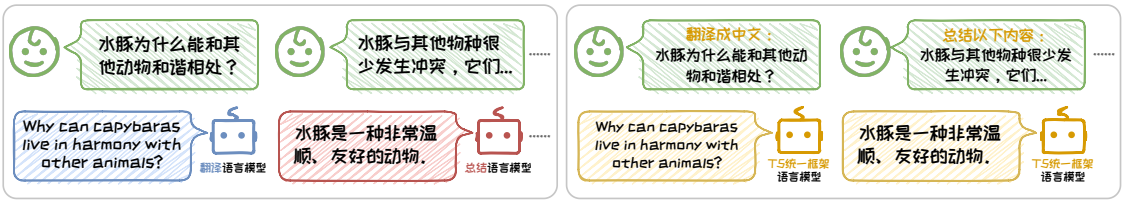


图 2.11: 传统语言模型和 T5 统一框架。

在模型架构方面，T5 与原始的包括一个编码器和一个解码器的 Transformer 架构相同。每个编码器和解码器又分别由多个编码模块和解码模块堆叠而成。T5 模型根据不同的具体参数，提供了五个不同的版本，分别是 T5-Small、T5-Base、T5-Large、T5-3B 以及 T5-11B。**T5-Small** 由 6 个编码模块和 6 个解码模块堆叠而成，其中隐藏层维度为 512，自注意力头的数量为 8，**总参数数量约为 6000 万**；**T5-Base** 与 BERT-Base 对标，由 12 个编码模块和 12 个解码模块堆叠而成，其中隐藏层维度为 768，自注意力头的数量为 12，**总参数数量约为 2.2 亿**；**T5-Large** 与 BERT-Large 对标，由 24 个编码模块和 24 个解码模块堆叠而成，隐藏层维度为 1024，自注意力头的数量为 16，**总参数数量约为 7.7 亿**；**T5-3B** 在 T5-Large 的基础上，把自注意力头的数量扩大到 32，另外还将全连接前馈网络的中间层维度扩大了 4 倍，**总参数数量约为 28 亿**；**T5-11B** 在 T5-3B 的基础上，进一步将自注意力头的数量扩大到 128，并又将全连接前馈网络的中间层维度扩大了 4 倍，**总参数数量约为 110 亿**。

2. T5 预训练方式

为了获取高质量、覆盖范围广泛的预训练数据集，Google Research 团队从大规模网页数据集 Common Crawl [14]中提取了大量的网页数据，并经过严格的清理和

[14]https://commoncrawl.org

过滤，最终生成了 C4 数据集（Colossal Clean Crawled Corpus），其覆盖了各种网站和文本类型，总规模达到了约 750GB。

基于此数据集，T5 提出了名为 Span Corruption 的预训练任务。这一与训练任务从原始输入中选择 15% 的 Token 进行破坏，每次都选择连续三个 Token 作为一个小段（span）整体被掩码成 `[MASK]`。与 BERT 模型中采用的单个 Token 预测不同，T5 模型需要**对整个被遮挡的连续文本片段进行预测**。这些片段可能包括连续的短语或子句，它们在自然语言中构成了具有完整意义的语义单元。这一设计要求模型不仅等理解局部词汇的表面形式，还要可以捕捉更深层次的句子结构和上下文之间的复杂依赖关系。Span Corruption 预训练任务显著提升了 T5 的性能表现，尤其是在文本摘要、问答系统和文本补全等需要生成连贯和逻辑性强的文本生成任务中。

3. T5 下游任务

基于预训练阶段学到的大量知识以及新提出的文本转文本的统一生成式框架，T5 模型可以在完全**零样本**（Zero-Shot）的情况下，利用 Prompt 工程技术直接适配到多种下游任务。同时，T5 模型也可以通过微调（Fine-Tuning）来适配到特定的任务。但是，微调过程需要针对下游任务收集带标签训练数据，同时也需要更多的计算资源和训练时间，因此通常只被应用于那些对精度要求极高且任务本身较为复杂的应用场景。

综上所述，T5 模型的文本转文本的统一生成式框架不仅简化了不同自然语言处理任务之间的转换流程，也为大语言模型的发展提供了新方向。如今，T5 模型已经衍生了许多变体，以进一步改善其性能。例如，mT5[43] 模型扩展了对 100 多种语言的支持，T0[31] 模型通过多任务训练增强了零样本学习（Zero-Shot Learning）能力，Flan-T5[8] 模型专注于通过指令微调，以实现进一步提升模型的灵活性和效率等等。

2.4.3 BART 语言模型

BART（Bidirectional and Auto-Regressive Transformers）是由 Meta AI 研究院同样于 2019 年 10 月提出的一个 Encoder-Decoder 架构模型。不同于 T5 将多种 NLP 任务集成到一个统一的框架，BART 旨在通过多样化的预训练任务来提升模型在文本生成任务和文本理解任务上的表现。

1. BART 模型结构

BART 的模型结构同样与原始的 Transformer 架构完全相同，包括一个编码器和一个解码器。每个编码器和解码器分别由多个编码模块和解码模块堆叠而成。BART 模型一共有两个版本，分别是 BART-Base 以及 BART-Large。**BART-Base** 由 6 个编码模块和 6 个解码模块堆叠而成，其中隐藏层维度为 768，自注意力头的数量为 12，**总参数数量约为 1.4 亿**；**BART-Large** 由 12 个编码模块和 12 个解码模块堆叠而成，其中隐藏层维度为 1024，自注意力头的数量为 16，**总参数数量约为 4 亿**。

2. BART 预训练方式

在预训练数据上，BART 使用了与 RoBERTa[23] 相同的语料库，包含小说数据集 BookCorpus[46]、英语维基百科数据集 [15]、新闻数据集 CC-News [16]、网页开放数据集 OpenWebText [17]以及故事数据集 Stories，总数据量达到约 160GB。

在预训练任务上，BART 以重建被破坏的文本为目标。其通过 Token 遮挡任务（Token Masking）、Token 删除任务（Token Deletion）、连续文本填空任务（Text Infilling）、句子打乱任务（Sentence Permutation）以及文档旋转任务（Document Rotation）等五个任务来破坏文本，然后训练模型对原始文本进行恢复。这种方式

[15]https://dumps.wikimedia.org

[16]http://web.archive.org/save/http://commoncrawl.org/2016/10/newsdataset-available

[17]http://Skylion007.github.io/OpenWebTextCorpus

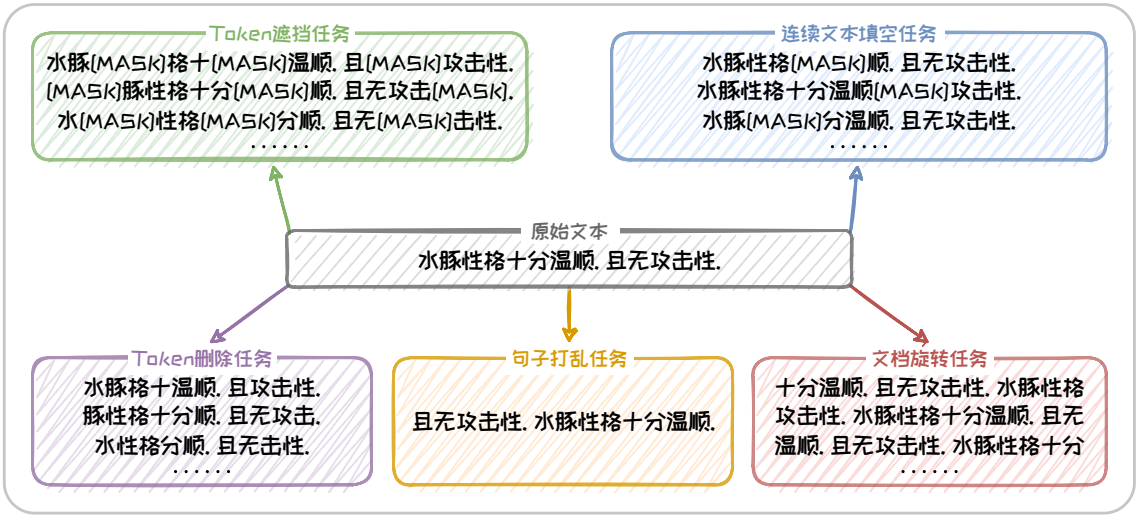


图 2.12: BART 预训练任务。

锻炼了模型对文本结构和语义的深入理解，增强了其在面对不完整或损坏信息时的鲁棒性。五个文本破坏任务的具体形式如下所述。

- **Token 遮挡任务**（Token Masking）：类似于 BERT 中的 MLM 任务，在原始文本中随机采样一部分 Token 并将其替换为 `[MASK]`，从而训练模型**推断被删除的 Token 内容**的能力。
- **Token 删除任务**（Token Deletion）：在原始文本中随机删除一部分 Token，从而训练模型**推断被删除的 Token 位置以及内容**的能力。
- **连续文本填空任务**（Text Infilling）：类似于 T5 的预训练任务，在原始文本中选择几段连续的 Token（每段作为一个 span），整体替换为 `[MASK]`。其中 span 的长度服从 $\lambda = 3$ 的泊松分布，如果长度为 0 则直接插入一个 `[MASK]`。这一任务旨在训练模型**推断一段 span 及其长度**的能力。
- **句子打乱任务**（Sentence Permutation）：将给定文本拆分为多个句子，并随机打乱句子的顺序。旨在**训练模型推理前后句关系**的能力。
- **文档旋转任务**（Document Rotation）：从给定文本中随机选取一个 Token，作为文本新的开头进行旋转。旨在**训练模型找到文本合理起始点**的能力。

在图2.12中，以给定文本“水豚性格十分温顺. 且无攻击性.”为例，对这个五个任务进行演示。在预训练结束后，可以对 BART 进行微调使其能够将在预训练阶段学到的语言知识迁移到具体的应用场景中，适配多种下游任务。这种从预训练到微调的流程，使得 BART 不仅在文本生成任务上表现出色，也能够适应文本理解类任务的挑战。后续同样也出现了 BART 模型的各种变体，包括可处理跨语言文本生成任务的 mBART[22] 模型等。

综上所述，基于 Encoder-Decoder 架构的大语言模型，在生成任务中展示了良好的性能表现。表2.2从模型参数量和预训练语料规模的角度对本章提到的基于 Encoder-Decoder 架构的模型进行了总结。可以看出此时模型参数数量的上限已达 110 亿。在模型结构和参数规模的双重优势下，相较于基于 Encoder-only 架构的模型，这些模型在翻译、摘要、问答等任务中取得了更优的效果。

表 2.2: Encoder-Decoder 架构代表模型参数和语料大小表。

模型	发布时间	参数量（亿）	语料规模
T5	2019.10	0.6-110 亿	750GB
mT5	2020.10	3-130 亿	9.7TB
T0	2021.10	30-110 亿	约 400GB
BART	2019.10	1.4-4 亿	约 20GB
mBART	2020.06	0.4-6.1 亿	约 1TB

2.5 基于 Decoder-only 架构的大语言模型

在开放式（Open-Ended）生成任务中，通常输入序列较为简单，甚至没有具体明确的输入，因此维持一个完整的编码器来处理这些输入并不是必要的。对于这种任务，Encoder-Decoder 架构可能显得**过于复杂且缺乏灵活性**。在这种背景下，Decoder-only 架构表现得更为优异。本节将对 Decoder-only 架构及其代表性模型进行介绍。

2.5.1 Decoder-only 架构

它通过自回归方法逐字生成文本，不仅保持了长文本的连贯性和内在一致性，而且在缺乏明确输入或者复杂输入的情况下，能够更自然、流畅地生成文本。此外，Decoder-only 架构由于去除了编码器部分，使得模型**更加轻量化**，从而**加快了训练和推理的速度**。因此，在同样的模型规模下，Decoder-only 架构可能表现得更为出色。

值得一提的是，Decoder-only 架构模型的概念最早可以追溯到 2018 年发布的 GPT-1[27] 模型。但在当时，由于以 BERT 为代表的 Encoder-only 架构模型在各项任务中展现出的卓越性能，Decoder-only 架构并没有受到足够的关注。直到 2020 年，GPT-3[5] 的突破性成功，使得 Decoder-only 架构开始被广泛应用于各种大语言模型中，其中最为流行的有 OpenAI 提出的 GPT 系列、Meta 提出的 LLaMA 系列等。其中，GPT 系列是起步最早的 Decoder-only 架构，在性能上也成为了时代的标杆。但从第三代开始，GPT 系列**逐渐走向了闭源**。而 LLaMA 系列虽然起步较晚，但凭借着同样出色的性能以及**始终坚持的开源道路**，也在 Decoder-only 架构领域占据了一席之地。接下来将对这两种系列的模型进行介绍。

2.5.2 GPT 系列语言模型

GPT（Generative Pre-trained Transformer）系列模型是由 OpenAI 开发的一系列基于 Decoder-only 架构的大语言模型。自从 2018 年问世以来，GPT 系列模型经历了快速的发展，其在模型规模、预训练范式上不断演进，取得了万众瞩目的效果，引领了本轮大语言模型发展的浪潮。其演进历程可以划分为五个阶段，表2.3对这五个阶段的模型参数规模和预料规模进行了总结。从表中可以明显看出，GPT 系列模型**参数规模与预训练语料规模呈现出激增的趋势**。然而，自 ChatGPT 版本起，

GPT 系列模型转向了闭源模式，其具体的参数量和预训练数据集的详细信息已不再公开。尽管如此，根据扩展法则，有理由猜测 ChatGPT 及其后续版本在参数规模与预训练语料规模上都有所增长。下面将对这五个发展阶段的模型分别进行介绍。

表 2.3: GPT 系列模型参数和语料大小表。

模型	发布时间	参数量（亿）	语料规模
GPT-1	2018.06	1.17	约 5GB
GPT-2	2019.02	1.24 / 3.55 / 7.74 / 15	40GB
GPT-3	2020.05	1.25 / 3.5 / 7.62 / 13 / 27 / 67 / 130 / 1750	1TB
ChatGPT	2022.11	未知	未知
GPT-4	2023.03	未知	未知
GPT-4o	2024.05	未知	未知

1. 初出茅庐：GPT-1 模型

OpenAI 的前首席科学家 Ilya Sutskever 在采访中[18]透露，OpenAI 自成立初期就开始探索如何通过下一词预测解决无监督学习的问题。但当时所用的 RNN 模型无法很好解决长距离依赖问题，上述问题没有得到很好的解决。直到 2017 年，Transformer 的出现为这一问题提供了新的解决方案了，为 OpenAI 的发展指明了方向。随后，OpenAI 开始步入正轨。2018 年 6 月，OpenAI 发布了第一个版本的 GPT（Generative Pre-Training）模型，被称为 GPT-1[27]。GPT-1 开创了 Decoder-only 架构下，通过下一词预测解决无监督文本生成的先河，为自然语言处理领域带来了革命性的影响。下面将分别对 GPT-1 的模型结构、预训练、下游任务等方面展开介绍。

(1) GPT-1 模型结构

在模型架构方面，GPT-1 使用了 Transformer 架构中的 Decoder 部分，省略了 Encoder 部分以及交叉注意力模块。其模型由 12 个解码块堆叠而成，每个解码块

[18]https://hackernoon.com/an-interview-with-ilya-sutskever-co-founder-of-openai

包含一个带掩码的自注意力模块和一个全连接前馈模块。其中隐藏层维度为 768，自注意力头的数量为 12，模型的最终参数数量约为 1.17 亿。

图2.13对比了 BERT-Base 以及 GPT-1 的模型结构。从图中可以看出，GPT-1 在结构上与 BERT-Base 高度类似，两者都包含 12 个编码或解码模块，每个模块也同样由一个自注意力模块和一个全连接前馈模块组成。两者之间的本质区别在于 BERT-Base 中的自注意力模块是双向的自注意力机制，而 GPT-1 中的自注意力模块则是**带有掩码的单向**自注意力机制。

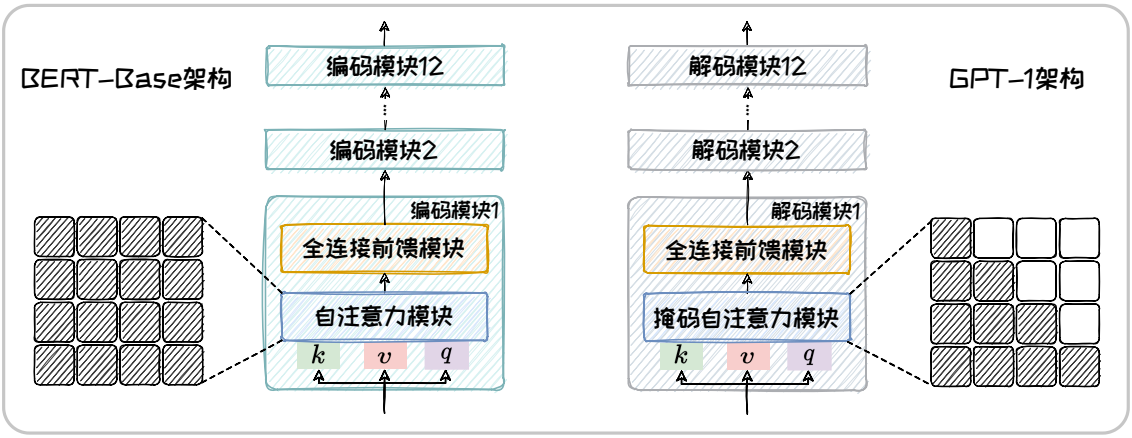


图 2.13: BERT-Base 和 GPT-1 模型。

(2) GPT-1 预训练方法

GPT-1 使用小说数据集 BookCorpus[46] 来进行预训练，该数据集包含约 8 亿个 Token，总数据量接近 5GB。在预训练方法上，GPT-1 采用下一词预测任务，即基于给定的上文预测下一个可能出现的 Token。以自回归的方法不断完成下一词预测任务，模型可以有效地完成文本生成任务，如图2.14所示。通过这种预训练策略，模型可以在不需要人为构造大量带标签数据的前提下，学习到大量语言的“常识”，学会生成连贯且上下文相关的文本。这不仅提高了模型的泛化能力，而且减少了对标注数据的依赖，为自然语言处理领域带来了新的研究方向和应用前景。

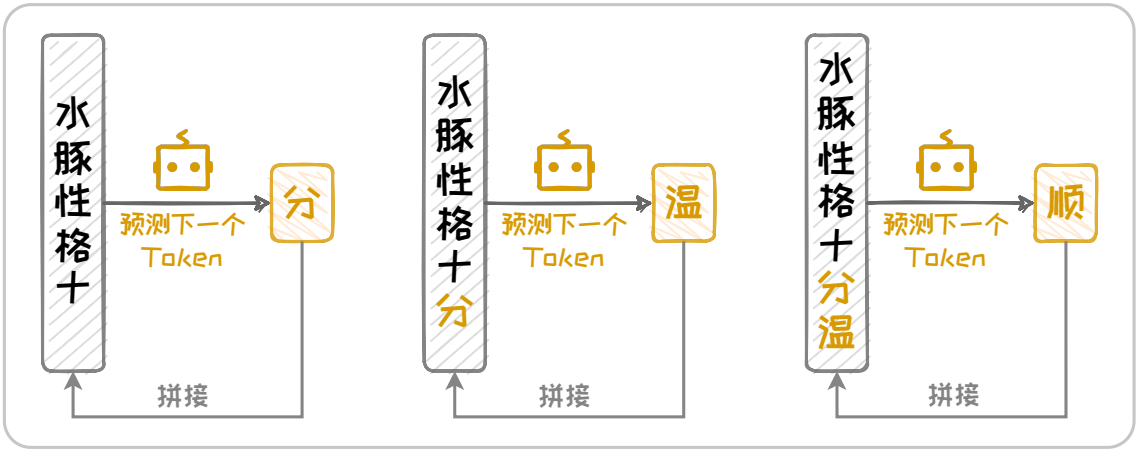


图 2.14: GPT-1 语言建模预训练任务。

(3) GPT-1 下游任务

尽管 GPT-1 模型在预训练后展现出了一定的潜力，但其任务泛化能力仍受限于当时的训练数据量和模型参数数量。为了提升模型在特定下游任务上的表现，通常需要进一步的有监督微调。微调过程涉及使用针对特定任务的标注数据来优化模型的参数，其中模型的输入和输出均以文本序列的形式呈现。例如，在以下任务中，我们需要构建针对特定应用场景的微调策略：

- **文本分类**：GPT-1 能够接收一段文本作为输入，并根据预定义的类别标签，如情感倾向（积极、消极或其他），对文本进行分类。这在情感分析、主题分类等场景中非常有用。
- **文本相似度评估**：当需要衡量两段文本之间的相似性时，GPT-1 能够分析并量化它们的内容和语义相似度。这项功能在比较文档、搜索结果优化和推荐系统中尤为重要。
- **多项选择题解答**：GPT-1 还可以处理多项选择问题。模型能够理解问题文本和选项内容，从给定的选项中识别并输出最合适的答案。

GPT-1 具备原生的文本生成能力。但受限于训练数据量和模型参数数量，其生成能力还不足以用于解决实际问题。此外，由于其单向注意力机制的限制，其全面

理解上下文的能力也有所欠缺。四个月后，具有双向上下文理解能力的 BERT 被提出，并以其强大的上下文嵌入能力迅速吸引了业界的广泛关注，遮盖了 GPT-1 的锋芒。尽管 GPT-1 当时在实用性上可能不及 BERT，但它作为 Decoder-only 架构的开端，为后续大语言模型的惊艳表现拉开了序幕。

2. 小有所成：GPT-2 模型

虽然 GPT-1 锋芒未露，OpenAI 并没有改变其 Decoder-only 的技术路线，而是选择在这一路线上继续深耕。在 2019 年 2 月，OpenAI 发布了 GPT 系列的第二代产品 GPT-2 [19]。相较于 GPT-1，GPT-2 在模型规模和预训练样本的质量上都进行了显著的提升，显著增强了模型的任务泛化能力。以下将对 GPT-2 的模型结构、预训练语料和下游任务适配情况进行介绍。

(1) GPT-2 模型结构

GPT-2 模型延续了 GPT-1 的 Decoder-only 架构，并在此基础上进一步加大了参数数量。GPT-2 一共发布了四个版本，分别是 GPT-2 Small、GPT-2 Medium、GPT-2 Large 以及 GPT-2 XL。其中 **GPT-2 Small** 在模型规模上接近 GPT-1 以及 BERT-Base，由 12 个编码块堆叠而成，隐藏层维度为 768，自注意力头的数量为 12，**总参数数量约为 1.24 亿**；**GPT-2 Medium** 在模型规模上接近 BERT-Large，由 24 个解码块堆叠而成，隐藏层维度为 1024，自注意力头的数量为 16，**总参数数量约为 3.55 亿**；**GPT-2 Large** 由 36 个解码块堆叠而成，隐藏层维度为 1280，自注意力头的数量为 20，**总参数数量约为 7.74 亿**；**GPT-2 XL** 是最大规模版本，由 48 个解码块堆叠而成，其中隐藏层维度为 1600，自注意力头的数量为 25，**总参数数量约为 15 亿**。

(2) GPT-2 预训练方法

在预训练中，GPT-2 继续采用下一词预测任务，但进一步提升了预训练数据的数量和质量。其采用了全新的 WebText 数据集，该数据集由 40GB 经过精心筛选和

[19] https://openai.com/index/gpt-2-1-5b-release

清洗的网络文本组成。通过使用 WebText 数据集进行预训练，GPT-2 的语言理解能力得到了显著增强，接触到了更多样化的语言使用场景，还学习到了更复杂的语言表达方式。这使得 GPT-2 在捕捉语言细微差别和构建语言模型方面更为精准，从而在执行各种自然语言处理任务时能够生成更准确、更连贯的文本。

(3) GPT-2 下游任务

GPT-2 的任务泛化能力得到了改善，在某些任务上可以不进行微调，直接进行**零样本学习**。这种能力大大增加了 GPT-2 在处理下游任务时的灵活性，降低了下游任务适配所需的成本。这为 Decoder-only 架构博得了更多关注。

3. 崭露头角：GPT-3 模型

为了进一步提升任务泛化能力，OpenAI 于 2020 年 6 月推出了第三代模型 GPT-3[5]。与前两代模型相比，GPT-3 在模型规模和预训练语料上进一步提升，并涌现出了优良的上下文学习（In-Context Learning, ICL）能力。在上下文学习能力的加持下，GPT-3 可以在不进行微调的情况下，仅通过任务描述或少量示例即可完成多样化的下游任务。关于上下文学习详细介绍参见本书3.2 章节。以下将对 GPT-3 的模型结构、预训练语料和下游任务适配情况进行介绍。

(1) GPT-3 模型架构

在模型架构上，GPT-3 继承并扩展了前两代的架构，显著增加了解码块的数量、隐藏层的维度和自注意力头的数量，参数量最高达到 1750 亿。庞大的参数量使得 GPT-3 能够捕获更加细微和复杂的语言模式，显著提升了模型的文本生成能力。GPT-3 设计了多个不同参数规模的版本，以满足不同应用场景的需求，详细参数细节见表2.4。

(2) GPT-3 预训练方法

延续了前两代的预训练方法，GPT-3 继续采用下一词预测作为预训练任务。其使用了更大规模和更多样化的互联网文本数据集，数据量接近 1TB，涵盖了 Com-

表 2.4: GPT-1 至 GPT-3 模型具体参数表。

模型版本	解码块数量	隐藏层维度	自注意力头数量	总参数量（亿）
GPT-1	12	768	12	1.17
GPT-2 Small	12	768	12	1.24
GPT-2 Medium	24	1024	16	3.55
GPT-2 Large	36	1280	20	7.74
GPT-2 XL	48	1600	36	15
GPT-3 Small	12	768	12	1.25
GPT-3 Medium	24	1024	16	3.5
GPT-3 Large	24	1536	16	7.62
GPT-3 XL	24	2048	24	13
GPT-3 2.7B	32	2560	32	27
GPT-3 6.7B	32	4096	32	67
GPT-3 13B	40	5120	40	130
GPT-3 175B	96	12288	96	1750

mon Crawl [20]、WebText、BookCorpus[46]、Wikipedia [21]等多个来源，包括书籍、网站、论坛帖子等各类文本形式。所有数据都经过了严格的筛选和清洗流程，以确保数据的质量和多样性。基于这些数据，GPT-3 学习到了更加丰富和多元的语言知识和世界知识。

(3) GPT-3 下游任务

GPT-3 模型涌现出了良好的上下文学习能力，使其可以在无需微调的情况下，仅通过在输入文本中明确任务描述和提供少量示例，便能够执行多种下游任务。上下文学习能力极大地增强了 GPT-3 的任务泛化能力，使其能够快速适应不同的应用场景。GPT-3 开始在文本生成、问答系统、语言翻译等众多自然语言处理任务中崭露头角。

4. 蓄势待发：InstructGPT 等模型

在 GPT-3 的基础上，OpenAI 进一步推出了一系列衍生模型。这些模型在 GPT-3 网络结构之上通过特定的训练方法，各个“身怀绝技”。例如，采用十亿行代码继续

[20]https://commoncrawl.org

[21]http://web.archive.org/save/http://commoncrawl.org/2016/10/newsdataset-available

预训练（Continual Pre-Training）的 Codex[6] 模型，可以有效地处理代码生成任务；采用用户偏好对齐（User Intent Alignment）的 InstructGPT[25] 模型，具备良好的指令跟随能力。其中，最具启发意义的是 InstructGPT，其也是 ChatGPT 的前身。它通过引入了**人类反馈强化学习**（Reinforcement Learning from Human Feedback, RLHF），显著提升了模型对用户指令的响应能力。

人类反馈强化学习旨在缓解模型在遵循用户指令时可能出现的不准确性和不可靠性，以使模型生成的内容更符合人类的要求。在人类反馈强化学习中，人类评估者首先提供关于模型输出质量的反馈，然后使用这些反馈来微调模型。具体过程如图2.15所示，整体可以分为以下三个步骤：1）**有监督微调**：收集大量“问题-人类回答”对作为训练样本，对大语言模型进行微调。2）**训练奖励模型**：针对每个输入，让模型生成多个候选输出，并由人工对其进行质量评估和排名，构成偏好数据集。用此偏好数据集**训练一个奖励模型**，使其可以对输出是否符合人类偏好进行打分。3）**强化学习微调**：基于上一步中得到的奖励模型，**使用强化学习方法优化第一步中的语言模型**，即在语言模型生成输出后，奖励模型对其进行评分，强化学习算法根据这些评分调整模型参数，以提升高质量输出的概率。

经过 RLHF 训练得到的 InstructGPT 的性能通常优于 GPT-3，尤其是在需要精确遵循用户指令的场景中。它生成的回答更加贴合用户的查询意图，有效减少了不相关或误导性内容的生成。InstructGPT 的研究为构建更智能、更可靠的 AI 系统提供了新的思路，展示了人类智慧和机器学习算法结合的巨大潜力。但是，RLHF 的计算成本十分高昂，主要是由于：1）**奖励模型**的训练过程复杂且耗时。2）除了需要单独训练语言模型和奖励模型外，还需要协调这两个模型进行**多模型联合训练**，这一过程同样复杂且消耗大量资源。

为了克服 RLHF 在计算效率上的缺陷，斯坦福大学在 2023 年在其基础上，提出了一种新的算法**直接偏好优化（Direct Preference Optimization, DPO）**[28]。DPO

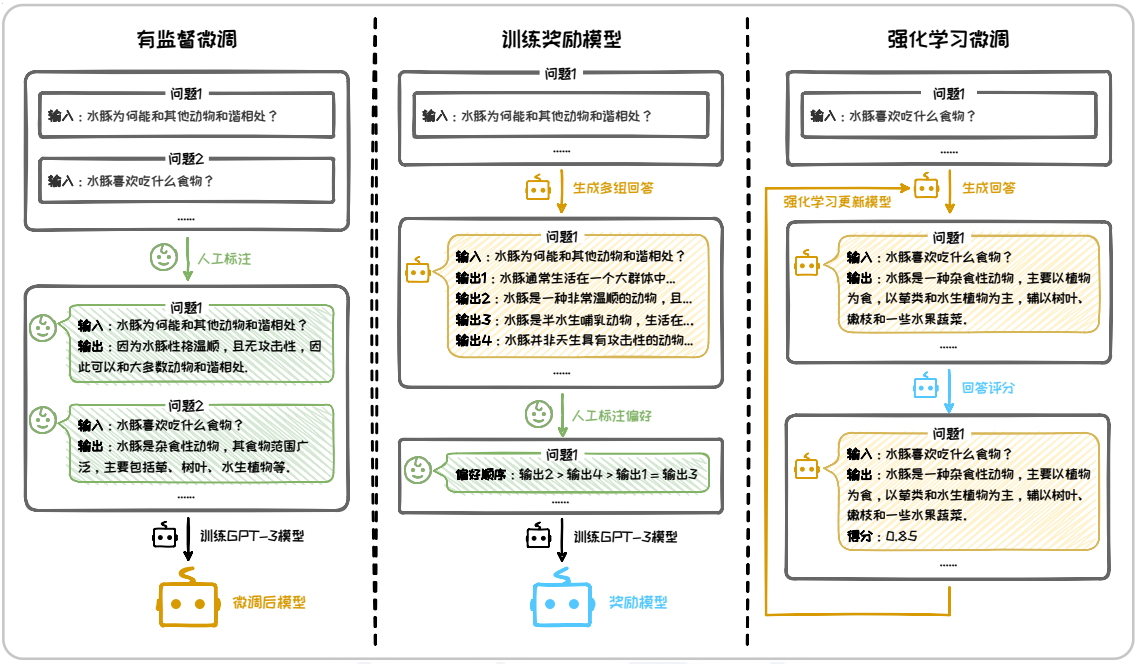


图 2.15: 人类反馈强化学习（RLHF）过程。

算法直接利用人类偏好数据来训练模型，省略了单独构建奖励模型以及应用复杂强化学习算法的步骤。该方法首先收集包含多个响应的人类偏好数据，并从中标记出最优和次优响应。然后微调模型以**提高模型选择最优响应的概率**，同时**降低选择次优响应的概率**。这种方法显著简化了人类反馈对齐的流程，提高了训练效率和模型稳定性。尽管在处理复杂的人类偏好时可能略逊于 RLHF，但 DPO 在计算效率上的优势使其在多个领域得到了广泛应用。

5. 一鸣惊人：ChatGPT 以及 GPT-4 等模型

OpenAI 于 2022 年 11 月推出了聊天机器人（**ChatGPT**Chat Generative Pre-trained Transformer）[22]。ChatGPT “一鸣惊人”，以强大的对话能力展示出令人惊讶的智能，一度燃起了 ChatGPT 是否可以通过 “图灵测试” 的讨论 [3]。此外，用户可以通过 OpenAI 提供的网页端或 API 轻松使用预训练后的 ChatGPT 模型，而无需在本地部署，标志着一种新的服务模式 LLMaaS(LLM as a Service) 的出现。但

[22]https://openai.com/blog/chatgpt

是，从 ChatGPT 起，GPT 系列模型走向闭源，我们无从窥探 ChatGPT 及后续模型的技术细节。

四个月后，OpenAI 于 2023 年 3 月继续发布了 **GPT-4** [23]模型。相较于 ChatGPT，GPT-4 在理解复杂语境、捕捉语言细微差别、生成连贯文本等任务上进一步提升，并且能够更有效地处理数学问题、编程挑战等高级认知任务。此外，GPT-4 还引入了对图文双模态的支持，扩展了其在图像描述和视觉问题解答等应用领域的可能性。

一年后，为了进一步提升模型性能以及用户体验，OpenAI 于 2024 年 5 月提出了 **GPT-4o** [24]。GPT-4o 模型在前代 GPT-4 的基础上，大幅提升了响应速度，显著降低了延迟，并且还增强了多模态处理能力以及多语言支持能力。其在客户支持、内容创作和数据分析等领域表现亮眼。GPT-4o 的推出标志着 AIGC 的应用日趋成熟。

GPT 系列模型的演进过程是人工智能发展史中一个激动人心的篇章。从 GPT-1 的“初出茅庐”到 GPT-4o 的“一鸣惊人”，短短 6 年时间，GPT 系列模型便带来了革命性的突破。诸多“科幻”变成现实，众多产业将被重塑。但是，随着 GPT 系列模型走向闭源，用户仅可以使用其功能，却无法参与到模型的共同创造和改进过程中。

2.5.3 LLAMA 系列语言模型

LLaMA（Large Language Model Meta AI）是由 Meta AI 开发的一系列大语言模型，其模型权重在非商业许可证下向学术界开放，推动了大语言模型的“共创”和知识共享。在模型架构上，LLaMA 借鉴了 GPT 系列的设计理念，同时在技术细节上进行了创新和优化。LLaMA 与 GPT 系列的主要区别在于：GPT 系列的升级主

[23]https://openai.com/index/gpt-4-research

[24]https://openai.com/index/hello-gpt-4o

线聚焦于模型规模与预训练语料的同步提升，而 LLaMA 则在模型规模上保持相对稳定，更**专注于提升预训练数据的规模**。表2.5展示了不同版本 LLaMA 模型对应的发布时间、参数量以及语料规模。当前，Meta AI 共推出三个版本的 LLaMA 模型。在这些模型的基础上，大量衍生模型陆续被推出。原始的 LLaMA 模型和其衍生模型一起构成了 LLaMA 生态系统。接下来将对 LLaMA 的三个版本及其部分衍生模型进行介绍。

表 2.5: LLaMA 系列模型参数和语料大小表。

模型	发布时间	参数量（亿）	语料规模
LLAMA-1	2023.02	67 / 130 / 325 / 652	约 5TB
LLAMA-2	2023.07	70 / 130 / 340 / 700	约 7TB
LLAMA-3	2024.04	80 / 700	约 50TB

1. LLaMA1 模型

LLaMA1[40] 是 Meta AI 于 2023 年 2 月推出的首个大语言模型。其在 Chinchilla[15] 扩展法则的指引下，实践“小模型 + 大数据”的理念，旨在以大规模的优质数据训练相对较小的模型。相对较小的参数规模可以赋能更快的推理速度，使其可以更好的应对计算资源有限的场景。

在预训练语料方面，LLaMA1 的预训练数据涵盖了大规模网页数据集 Common Crawl[25]、T5[29] 提出的 C4 数据集，以及来自 Github、Wikipedia、Gutenberg、Books3、Arxiv 以及 StackExchange 等多种来源的数据，总数据量高达 5TB。

在模型架构方面，LLaMA1 采用了与 GPT 系列同样的网络架构。但是，其在 Transformer 原始词嵌入模块、注意力模块和全连接前馈模块上进行了优化。在词嵌入模块上，为了提高词嵌入质量，LLaMA1 参考了 GPTNeo[4] 的做法，使用**旋转位置编码**（Rotary Positional Embeddings, RoPE）[36] 替代了原有的绝对位置编码，从而增强位置编码的表达能力，增强了模型对序列顺序的理解。在注意力模

[25]https://commoncrawl.org

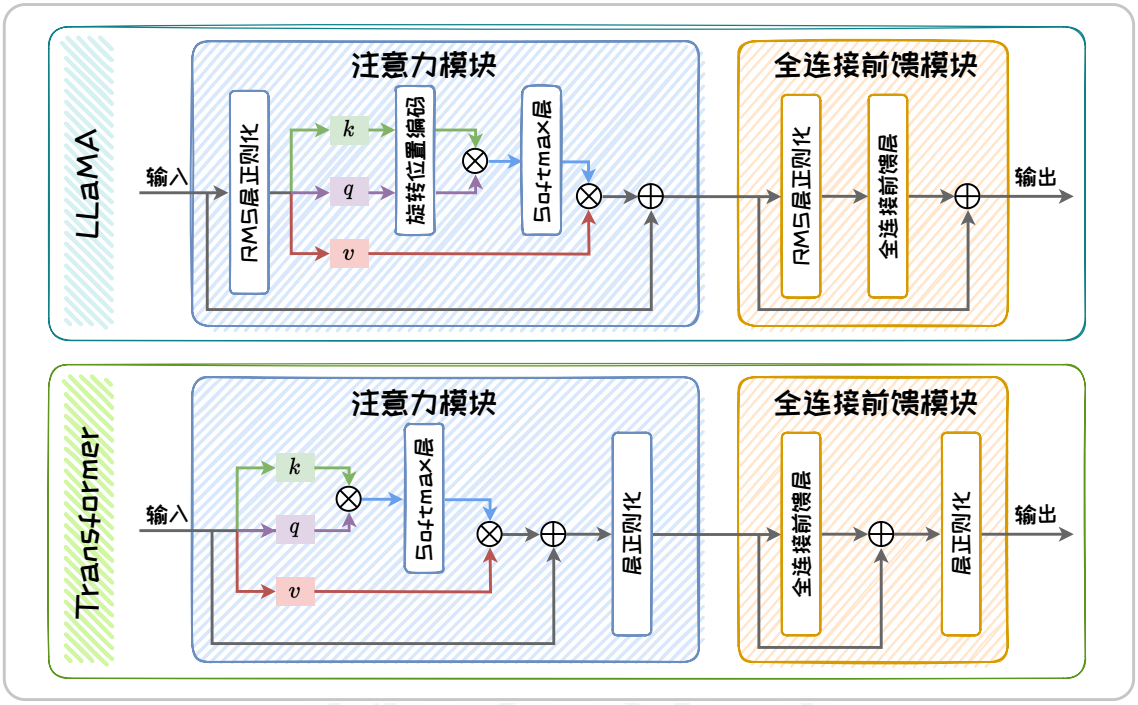


图 2.16: LLaMA 解码块架构与标准 Transformer 解码器架构对比。

块上，LLaMA1 参考了 PaLM[7] 的做法，将 Transformer 中的 RELU 激活函数改为 **SwiGLU 激活函数** [34]。并且，LLaMA1 在进行自注意力操作之前对查询（query）以及键（key）添加旋转位置编码。在全连接前馈模块上，LLaMA1 借鉴了 GPT-3[5] 中的 **Pre-Norm 层正则化策略**，将正则化应用于自注意力和前馈网络的输入。在注意力模块和全连接前馈模块上的改进如图2.16所示。

基于该框架，LLaMA1 共推出了 4 个版本的模型。不同版本对应的具体参数如表2.6所示。

表 2.6: LLaMA1 模型具体参数表。

模型版本	解码块数量	隐藏层维度	自注意力头数量	总参数量（亿）
LLaMA1-7B	32	4096	32	67
LLaMA1-13B	40	5120	40	130
LLaMA1-32B	60	6656	52	325
LLaMA1-65B	80	8192	64	652

2. LLaMA2 模型

2023 年 7 月，Meta AI 发布了 LLaMA 系列的第二代模型 LLaMA2[39]。秉承“小模型 + 大数据”的设计理念，LLaMA2 在 LLaMA1 的基础上进一步优化和扩充了训练数据，将语料库的规模扩展至约 7TB，实现了对更丰富语言和领域资源的覆盖。此外，在预训练阶段之后，LLaMA2 采纳了人类反馈强化学习的方法，进一步提升了模型的性能。首先，其使用了大规模且公开的指令微调数据集 [8] 对模型进行有监督的微调。然后，LLaMA2 还训练了 RLHF 奖励模型，并基于**近似策略优化**（Proximal Policy Optimization, PPO）[33] 以及**拒绝采样**（Rejection Sampling）进行强化学习对模型进行更新。

在模型架构上，LLaMA2 继承了 LLaMA1 的架构。LLaMA2 共推出了四个版本的模型，不同版本对应的具体参数如表2.7所示。其中，LLaMA2-34B 和 LLaMA2-70B 还额外增加了**分组查询注意力**（Grouped Query Attention, GQA）[1]，以提升计算效率。在分组查询注意力机制下，键（key）以及值（value）不再与查询（query）一一对应，而是一组查询共享相同的键和值，从而有效降低内存占用并减少模型总参数量。

表 2.7: LLaMA2 模型具体参数表。

模型版本	解码块数量	隐藏层维度	自注意力头数量	总参数量（亿）
LLaMA2-7B	32	4096	32	70
LLaMA2-13B	40	5120	40	130
LLaMA2-34B	60	6656	52	340
LLaMA2-70B	80	8192	64	700

3. LLaMA3 模型

2024 年 4 月，Meta AI 于 2024 年 4 月进一步推出了第三代模型 LLaMA3 [26]。LLaMA3 挑选了规模高达 50TB 的预训练语料，是 LLaMA2 的 7 倍之多。这一语料

[26]https://ai.meta.com/blog/meta-llama-3

库不仅包含丰富的代码数据以增强模型的逻辑推理能力，还涵盖了超过 5% 的非英文数据，覆盖 30 多种语言，显著扩展了模型的跨语言处理能力。此外，LLaMA3 还进行了与 LLaMA2 一样的人类反馈强化学习，这一策略已被证明能显著提升模型性能。最终 LLaMA3 的性能在多个评测指标上全面超越了前代模型。即便是参数量相对较少的 80 亿参数版本，也展现出了超越 LLaMA2 700 亿参数版本的卓越性能。而 700 亿参数版本的 LLaMA3，在多项任务上的性能更是超越了业界标杆 GPT-4 模型。

在模型架构上，LLaMA3 与前一代 LLaMA2 几乎完全相同，只是在分词（tokenizer）阶段，**将字典长度扩大了三倍**，极大提升了推理效率。这一改进减少了中文字符等语言元素被拆分为多个 Token 的情况，有效降低了总体 Token 数量，从而提高了模型处理语言的连贯性和准确性。另一方面，扩大的字典有助于减少对具有完整意义的语义单元进行分割，使模型在处理文本时可以更准确的捕捉词义和上下文，提高生成文本的流畅性和连贯性。LLaMA3 在其推出的 80 亿参数以及 700 亿参数版本上，均采用了分组查询注意力机制。这两个版本的模型参数与 LLaMA2 的对应版本保持高度一致，但在性能上实现了质的飞跃，充分证明了数据的力量。

4. LLaMA 衍生模型

LLaMA 模型的开源共享吸引了众多研究者在其基础上继续创作。研究者们或继续改进 LLaMA 模型的性能，或将 LLaMA 模型适配到垂直领域，或将 LLaMA 模型支持的数据模态进行扩充。众智众创为 LLaMA 营造出了一个多样的、充满活力的研究生态。下面对三类主流的 LLaMA 衍生模型进行简要介绍。

- **性能改进类模型**：一些研究者专注于通过微调继续提升 LLaMA 模型性能。例如，Alpaca [27]基于 GPT-3.5 生成的指令遵循样例数据对 LLaMA1 进行微调，以较小的模型规模实现了与 GPT-3.5 相媲美的性能。Vicuna [28]模型则另辟蹊

[27]https://crfm.stanford.edu/2023/03/13/alpaca.html

[28]https://lmsys.org/blog/2023-03-30-vicuna

径，利用 ShareGPT 平台上累积的日常对话数据微调 LLaMA1 模型，进一步提升了它的对话能力。Guanaco[10] 模型则通过在微调 LLaMA1 的过程中引入 QLoRA 技术，显著降低了微调的时间成本，提高了微调效率。

- **垂域任务类模型**：尽管 LLaMA 在通用任务上表现出色，但在特定领域的应用潜力仍待挖掘。因此，大量研究者们针对垂直领域对 LLaMA 进行微调，以改善其在垂直领域上的表现。例如，CodeLLaMA[30] 模型在 LLaMA2 的基础上，利用大量公开代码数据进行微调，使其能更好地适应自动化代码生成、错误检测、以及代码优化等任务。LawGPT[24] 模型通过 30 万条法律问答对 LLaMA1 模型进行指令微调，显著增强了其对法律内容的处理能力。GOAT[21] 模型通过 Python 脚本生成的数学题库对 LLaMA1 模型进行微调，提高其解决各类数学题的准确率。Cornucopia [29]模型则利用金融问答数据进行微调，增强了金融问答的效果。
- **多模态任务类模型**：通过整合视觉模态编码器和跨模态对齐组件，研究者们将 LLaMA 模型扩展到多模态任务上。例如，LLaVA[19] 在 Vicuna 的基础上利用 CLIP 提取图像特征并利用一个线性投影层实现图片和文本之间的对齐。MiniGPT4[45] 在 Vicuna 的基础上使用 VIT-G/14 以及 Q-Former 作为图像编码器，并同样使用线性投影层来实现图片和文本之间的对齐，展现了多模态任务处理能力。

这些衍生模型不仅丰富了 LLaMA 模型的应用场景，也为自然语言处理领域的研究提供了新的方向和可能性。LLaMA 系列以其开源开放的姿态，吸引全球研究者参与共创。我们有理由相信 LLaMA 系列模型携其衍生模型将绽放出满天繁星，照亮大语言模型前行之路。

[29]https://github.com/jerry1993-tech/Cornucopia-LLaMA-Fin-Chinese

表 2.8: GPT 系列和 LLaMA 系列模型参数和语料大小表。

模型	发布时间	参数量（亿）	语料规模
GPT-1	2018.06	1.17	约 5GB
GPT-2	2019.02	1.24 / 3.55 / 7.74 / 15	40GB
GPT-3	2020.05	1.25 / 3.5 / 7.62 / 13 / 27 / 67 / 130 / 1750	1TB
ChatGPT	2022.11	未知	未知
GPT-4	2023.03	未知	未知
GPT-4o	2024.05	未知	未知
LLAMA-1	2023.02	67 / 130 / 325 / 652	约 5TB
LLAMA-2	2023.07	70 / 130 / 340 / 700	约 7TB
LLAMA-3	2024.04	80 / 700	约 50TB

基于 Decoder-only 架构的大语言模型，凭借其卓越的生成能力，引领了新一轮生成式人工智能的浪潮。表2.8展示了 GPT 系列和 LLaMA 系列不同版本的具体参数，从中可以发现基于 Decoder-only 架构的模型在参数数量和预训练语料规模上急速增长。随着算力资源和数据资源的进一步丰富，基于 Decoder-only 架构的大语言模型必将释放出更为璀璨的光芒。

2.6 非 Transformer 架构

Transformer 结构是当前大语言模型的主流模型架构，其具备构建灵活、易并行、易扩展等优势。但是，Transformer 也并非完美。其并行输入的机制会导致模型规模随输入序列长度平方增长，导致其在处理长序列时面临计算瓶颈。为了提高计算效率和性能，解决 Transformer 在长序列处理中的瓶颈问题，可以选择基于 RNN 的语言模型。RNN 在生成输出时，只考虑之前的隐藏状态和当前输入，理论上可以处理无限长的序列。然而，传统的 RNN 模型（如 GRU、LSTM 等）在处理长序列时可能难以捕捉到长期依赖关系，且面临着梯度消失或爆炸问题。为了克服这些问题，近年来，研究者提出了两类现代 RNN 变体，分别为**状态空间模型**（State Space Model，SSM）和**测试时训练**（Test-Time Training，TTT）。这两种范式都可

以实现关于序列长度的线性时间复杂度，且避免了传统 RNN 中存在的问题。本节将对这两种范式及对应的代表性模型进行简要介绍。

2.6.1 状态空间模型 SSM

状态空间模型（State Space Model，SSM）[13] 范式可以有效处理长文本中存在的长程依赖性（Long-Range Dependencies, LRDs）问题，并且可以有效降低语言模型的计算和内存开销。本小节将首先介绍 SSM 范式，再分别介绍两种基于 SSM 范式的代表性模型：RWKV 和 Mamba。

1. SSM

SSM 的思想源自于控制理论中的动力系统。其通过利用一组状态变量来捕捉系统状态随时间的连续变化，这种连续时间的表示方法天然地适用于描述长时间范围内的依赖关系。此外，SSM 还具有递归和卷积的离散化表示形式，既能在推理时通过递归更新高效处理序列数据，又能在训练时通过卷积操作捕捉全局依赖关系。

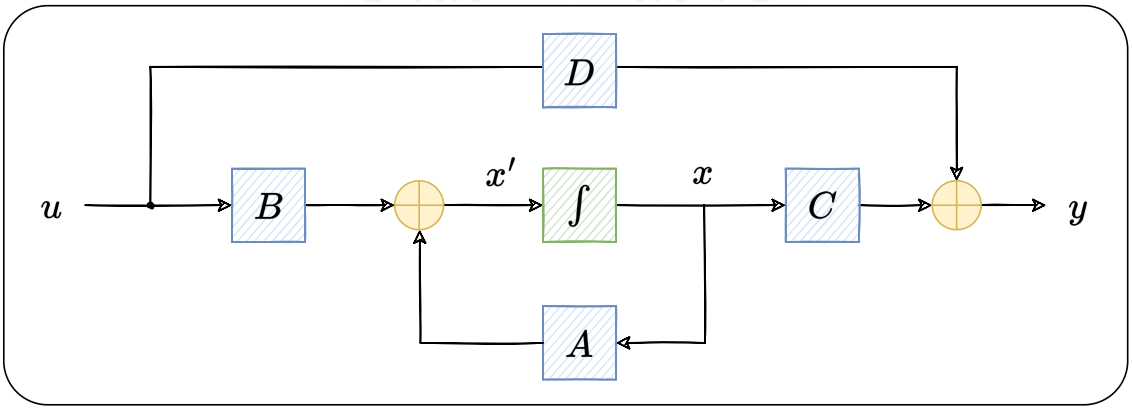


图 2.17: SSM 范式。

如图 2.17，SSM 在三个随时间 t 变化的变量和四个可学习的矩阵的基础上构造而成。三个变量分别为：$x(t) \in \mathbb{C}^n$ 表示 n 个状态变量，$u(t) \in \mathbb{C}^m$ 表示 m 个状

态输入，$y(t) \in \mathbb{C}^p$ 表示 p 个输出。四个矩阵分别为：状态矩阵 $\mathbf{A} \in \mathbb{C}^{n\times n}$，控制矩阵 $\mathbf{B} \in \mathbb{C}^{n\times m}$，输出矩阵 $\mathbf{C} \in \mathbb{C}^{p\times n}$ 和命令矩阵 $\mathbf{D} \in \mathbb{C}^{p\times m}$。SSM 的系统方程为：

$$\begin{aligned} x'(t) &= \mathbf{A}x(t) + \mathbf{B}u(t) \\ y(t) &= \mathbf{C}x(t) + \mathbf{D}u(t), \end{aligned} \tag{2.5}$$

其中，$x'(t) = \mathbf{A}x(t) + \mathbf{B}u(t)$ 为状态方程，描述了系统状态如何基于输入和前一个状态变化，其计算出的是状态关于时间的导数 $x'(t)$，为了得到状态 $x(t)$，还需对其进行积分操作。$y(t) = \mathbf{C}x(t) + \mathbf{D}u(t)$ 为输出方程，描述了系统状态如何转化为输出，其中的 $x(t)$ 是通过状态方程更新且积分后的值。在深度学习中，$\mathbf{D}u(t)$ 项表示残差连接, 可被忽略。

该方程可视作 SSM 系统方程的连续形式，适用于对连续数据（例如音频信号、时间序列）的处理，但是在训练和推理都非常慢。为了提高对 SSM 的处理效率，需要对该方程进行离散化操作。**离散化（Discretization）**是 SSM 中最为关键的步骤，能够将系统方程从连续形式转换为递归形式和卷积形式，从而提升整个 SSM 架构的效率。将连续形式的 SSM 系统方程离散化时，可以使用梯形法代替连续形式中的积分操作，其原理是将定义在特定区间上的函数曲线下的区域视为梯形，并利用梯形面积公式计算其面积。由此，可以得出离散化后递归形式下的系统方程：

$$\begin{aligned} x_k &= \bar{\mathbf{A}}x_{k-1} + \bar{\mathbf{B}}u_k \\ y_k &= \bar{\mathbf{C}}x_k。 \end{aligned} \tag{2.6}$$

在该方程中，状态方程由前一步的状态和当前输入计算当前状态，体现了递归的思想。其中，$\bar{\mathbf{A}}, \bar{\mathbf{B}}, \bar{\mathbf{C}}$ 为离散形式下的矩阵，其与连续形式下矩阵 $\mathbf{A}, \mathbf{B}, \mathbf{C}$ 的关系分别表示为：$\bar{\mathbf{A}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\left(\mathbf{I} + \frac{\Delta}{2}\mathbf{A}\right)$，$\bar{\mathbf{B}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\Delta\mathbf{B}$，$\bar{\mathbf{C}} = \mathbf{C}$，其中 $\Delta = t_{n+1} - t_n$。

递归形式的 SSM 类似于 RNN，具有 RNN 的优缺点。其适用于顺序数据的处理，能够实现与序列长度呈线性复杂度的高效推理，但是无法并行训练，当面临长

序列时存在梯度消失或爆炸问题。将系统方程的递归形式进行迭代，可以得到卷积形式。这里省略了推导过程，直接给出迭代后 x_k 和 y_k 的结果：

$$\begin{aligned} x_k &= \bar{\mathbf{A}}^k\bar{\mathbf{B}}u_0 + \bar{\mathbf{A}}^{k-1}\bar{\mathbf{B}}u_1 + \cdots + \bar{\mathbf{B}}u_k \\ y_k &= \bar{\mathbf{C}}x_k = \bar{\mathbf{C}}\bar{\mathbf{A}}^k\bar{\mathbf{B}}u_0 + \bar{\mathbf{C}}\bar{\mathbf{A}}^{k-1}\bar{\mathbf{B}}u_1 + \cdots + \bar{\mathbf{C}}\bar{\mathbf{B}}u_k。 \end{aligned} \tag{2.7}$$

可以观察到，将系统方程的递归形式迭代展开后，输出 y_k 是状态输入 u_k 的卷积结果，其卷积核为：

$$\bar{\mathbf{K}}_k = (\bar{\mathbf{C}}\bar{\mathbf{B}}, \bar{\mathbf{C}}\bar{\mathbf{A}}\bar{\mathbf{B}}, \ldots, \bar{\mathbf{C}}\bar{\mathbf{A}}^k\bar{\mathbf{B}})。 \tag{2.8}$$

因此，SSM 系统方程的卷积形式为：

$$y_k = \bar{\mathbf{K}}_k * u_k\,, \tag{2.9}$$

其中，卷积核是由 SSM 中的矩阵参数决定的，由于这些参数在整个序列的处理过程中是固定的，被称为**时不变性**。时不变性使得 SSM 能够一致地处理不同时间步长的数据，进行高效的并行化训练。但由于上下文长度固定，卷积形式的 SSM 在进行自回归任务时延迟长且计算消耗大。结合离散化后 SSM 的递归形式和卷积形式的优缺点，可以选择在**训练时使用卷积形式，推理时使用递归形式**。

综上，SSM 架构的系统方程具有三种形式，分别为连续形式、离散化的递归形式以及离散化的卷积形式，可应用于文本、视觉、音频和时间序列等任务，在应用时，需要根据具体情况选择合适的表示形式。SSM 的优势在于能够处理非常长的序列，虽然比其它模型参数更少，但在处理长序列时仍然可以保持较快的速度。

当前，各种现有 SSM 架构之间的主要区别在于基本 SSM 方程的离散化方式或 $\mathbf{A}$ 矩阵的定义。例如，S4[14]（Structured State Space Model）是一种 SSM 变体，其关键创新是使用 HiPPO 矩阵来初始化 $\mathbf{A}$ 矩阵，在处理长序列数据时表现优异。此外，RWKV 和 Mamba 是两种基于 SSM 范式的经典架构，下面将分别介绍这两种架构。

2. RWKV

RWKV（Receptance Weighted Key Value）[26] 是基于 SSM 范式的创新架构，其核心机制 WKV 的计算可以看作是两个 SSM 的比。RWKV 的设计结合了 RNNs 和 Transformers 的优点，既保留了推理阶段的高效性，又实现了训练阶段的并行化。（注：这里讨论的是 RWKV-v4）

RWKV 模型的核心模块有两个：**时间混合模块**和 **通道混合模块**。时间混合模块主要处理序列中不同时间步之间的关系，通道混合模块则关注同一时间步内不同特征通道[30]之间的交互。时间混合模块和通道混合模块的设计基于四个基本元素：**接收向量 R**、**键向量 K**、**值向量 V** 和 **权重 W**，下面将根据图 2.18，介绍这些元素在两个模块中的计算流程和作用。

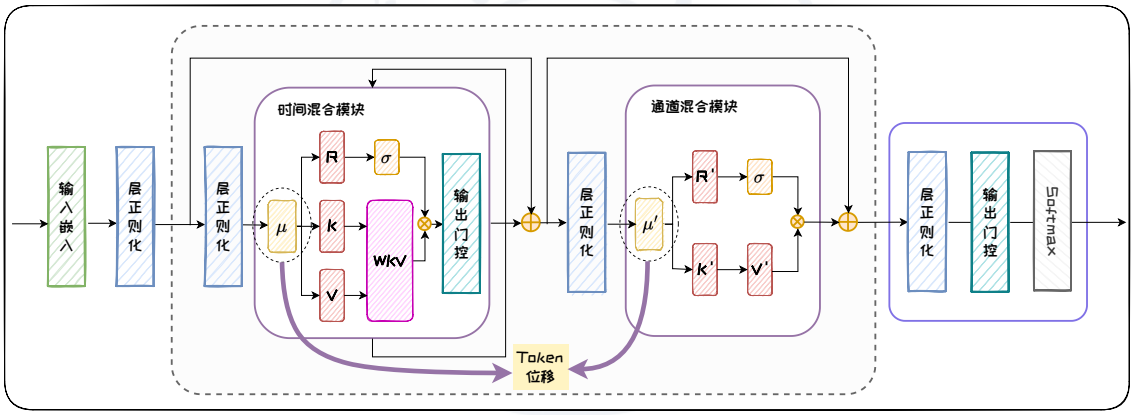


图 2.18: RWKV 架构。

时间混合模块和通道混合模块中共有的操作是 Token 位移，该步通过对当前时间步和前一时间步的输入进行线性插值来实现，从而确保了模型对序列中时间变化的敏感性。在时间混合模块中，接受向量 R 负责接收并整合来自序列历史的信息，权重 W 表示位置权重衰减，键向量 K 和值向量 V 类似传统注意力机制中的键和值，分别用于匹配和携带信息。时间混合模块首先将当前步长和前一步长的

[30]在这里，**通道**指的是在神经网络中同一时间步内的不同特征维度。例如，在处理自然语言处理任务时，每个时间步的输入可能是一段文本的一个词，而每个词会通过嵌入层转化为一个向量，这个向量的每个元素就代表一个特征通道，向量维数就是通道数量。

输入进行线性组合，通过线性投影得到 R、K、V 向量；随后，通过 WKV 机制来确保每个通道的权重随时间推移逐步衰减；最后，将表示过去信息的 $\sigma(\mathrm{R})$ 和表示当前信息的 WKV 向量通过输出门控进行整合，传递给通道混合模块。

时间混合模块中的 **WKV 机制**是 RWKV 的核心部分。在 RWKV 中，权重 W 是一个与通道相关的时间衰减向量，该向量可以表示为：$w_{t,i} = -(t-i)w$，i 表示从当前时间步长 t 向后追溯的某个时间步长，w 是一个非负向量，其长度为通道数。通过这种方式，模型能够捕捉时间序列中不同时间步长之间的依赖关系，实现每个通道权重随时间向后逐步衰减的效果。WKV 机制的关键在于利用线性时不变递归来更新隐藏状态，这可以看作是两个 SSM 之比。具体公式为：

$$wkv_t = \frac{\sum_{i=1}^{t-1} e^{-(t-1-i)w+k_i} \odot v_i + e^{u+k_t} \odot v_t}{\sum_{i=1}^{t-1} e^{-(t-1-i)w+k_i} + e^{u+k_t}}, \tag{2.10}$$

其中，u 是一个单独关注当前 Token 的向量。由于分子和分母都表现出状态更新和输出的过程，与 SSM 思想类似，因此该公式可以看作是两个 SSM 之比。通过权重 W 和 WKV 机制，模块能够有效地处理长时间序列数据，并减少梯度消失问题。

在通道混合模块中，R′、K′、V′ 的作用与时间混合模块类似，R′ 和 K′ 同样由输入的线性投影得到，V′ 的更新则额外依赖于 K′。之后，将 $\sigma(\mathrm{R}')$ 和 V′ 整合，以实现不同通道之间的信息交互和融合。

此外，RWKV 架构还采用了时间依赖的 Softmax 操作，提高数值稳定性和梯度传播效率，以及采用层归一化来稳定梯度，防止梯度消失和爆炸。为了进一步提升性能，RWKV 还采用了自定义 CUDA 内核、小值初始化嵌入以及自定义初始化等优化措施。

RWKV 在模型规模、计算效率和模型性能方面都表现可观。在**模型规模**方面，RWKV 模型参数扩展到了 14B，是第一个可扩展到数百亿参数的非 Transformer 架构。在**计算效率**方面，RWKV 允许模型被表示为 Transformer 或 RNN，从而使得模型在训练时可以并行化计算，在推理时保持恒定的计算和内存复杂度。在**模型性**

能方面，RWKV 在 NLP 任务上的性能与类似规模的 Transformer 相当，在长上下文基准测试中的性能仅次于 S4。

RWKV 通过创新的线性注意力机制，成功结合了 Transformer 和 RNN 的优势，在模型规模和性能方面取得了显著进展。然而，在处理长距离依赖关系和复杂任务时，RWKV 仍面临一些局限性。为了解决这些问题并进一步提升长序列建模能力，研究者们提出了 Mamba 架构。

3. Mamba

时不变性使得 SSM 能够一致地处理不同时间步长的数据，进行高效的并行化训练，但是同时也导致其处理信息密集的数据（如文本）的能力较弱。为了弥补这一不足，Mamba [12] 基于 SSM 架构，提出了**选择机制**（Selection Mechanism）和**硬件感知算法**（Hardware-aware Algorithm），前者使模型执行基于内容的推理，后者实现了在 GPU 上的高效计算，从而同时保证了**快速训练和推理**、**高质量数据生成**以及**长序列处理能力**。

Mamba 的**选择机制**通过动态调整模型参数来选择需要关注的信息，使模型参数能够根据输入数据动态变化。具体来说，Mamba 将离散化 SSM 中的参数 $\mathbf{B}, \mathbf{C}, \Delta$ 分别转变成以下函数: $s_{\mathbf{B}}(x) = \text{Linear}_N(x)$、$s_{\mathbf{C}}(x) = \text{Linear}_N(x)$、$s_\Delta(x) = \text{Broadcast}_D(\text{Linear}_1(x))$，并采用非线性激活函数 $\tau_\Delta = \text{softplus}$ 来调节参数 Δ。其中 Linear_d 是对特征维数 d 的参数化投影，s_Δ 和 τ_Δ 函数的选择与 RNN 的门控机制相关联 [38]。此外，Mamba 还对张量形状进行相应调整，使模型参数具有时间维度，这意味着模型参数矩阵在每个时间步都有不同的值，从时间不变转变为时间变化的。

选择机制使模型参数变成了输入的函数，且具有时间维度，因此模型不再具备卷积操作的平移不变性和线性时不变性，从而影响其效率。为了实现选择性 SSM 模型在 GPU 上的高效计算，Mamba 提出一种**硬件感知算法**，主要包括内核融合、

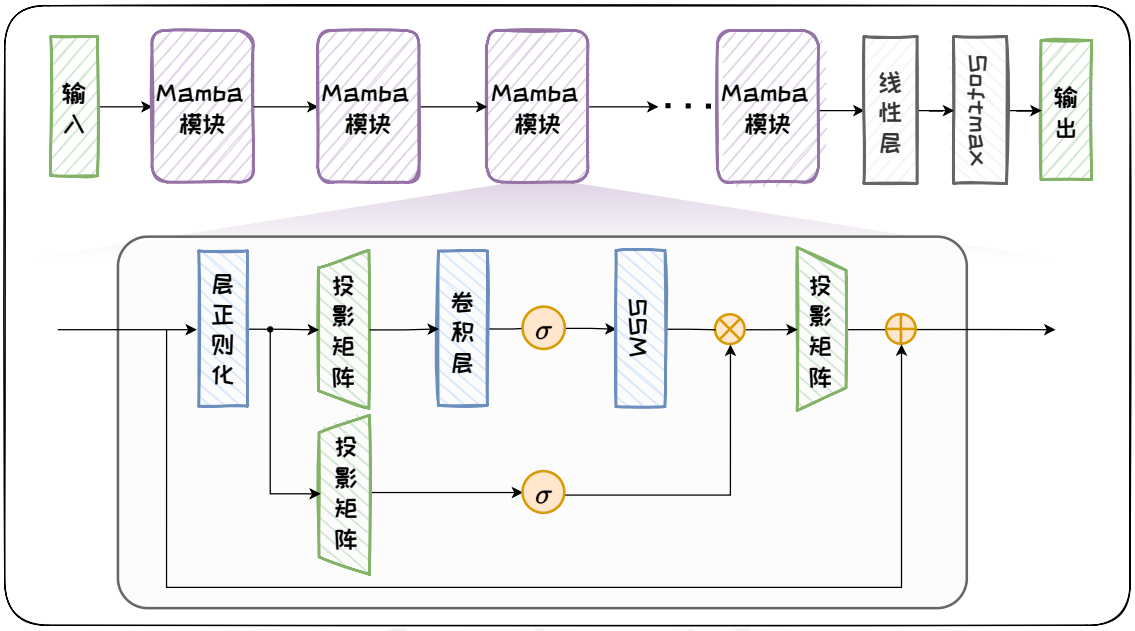


图 2.19: Mamba 架构。

并行扫描和重计算三方面内容。**内核融合**通过减少内存 I/O 操作来提高速度。**并行扫描**利用并行化算法提高效率。**重计算**则在反向传播时重新计算中间状态，以减少内存需求。

具体实现中，将 SSM 参数从较慢的高带宽内存（HBM）加载到更快的静态随机存取存储器（SRAM）中进行计算，然后将最终输出写回 HBM。这样可以在保持高效计算的同时，减少内存使用，使模型内存需求与优化的 Transformer 实现（如 FlashAttention）相同。

Mamba 通过将带有选择机制的 SSM 模块与 Transformer 的前馈层相结合，形成了一个简单且同质的架构设计。如图 2.19所示，Mamba 架构是由完全相同的 Mamba 模块组成的递归模型，每个 Mamba 模块都在前馈层中插入了卷积层和带有选择机制的 SSM 模块，其中激活函数 σ 选用 SiLU / Swish 激活。

通过引入选择机制和硬件感知算法，Mamba 在实际应用中展示了卓越的性能和效率，包括：（1）**快速训练和推理**：训练时，计算和内存需求随着序列长度线性增长，而推理时，每一步只需常数时间，不需要保存之前的所有信息。通过硬件

感知算法，Mamba 不仅在理论上实现了序列长度的线性扩展，而且在 A100 GPU 上，其推理吞吐量比类似规模的 Transformer 提高了 5 倍。（2）**高质量数据生成**：在语言建模、基因组学、音频、合成任务等多个模态和设置上，Mamba 均表现出色。在语言建模方面，Mamba-3B 模型在预训练和后续评估中性能超过了两倍参数量的 Transformer 模型性能。（3）**长序列处理能力**：Mamba 能够处理长达百万级别的序列长度，展示了处理长上下文时的优越性。

虽然 Mamba 在硬件依赖性和模型复杂度上存在一定的局限性，但是它通过引入选择机制和硬件感知算法显著提高了处理长序列和信息密集数据的效率，展示了在多个领域应用的巨大潜力。Mamba 在多种应用上的出色表现，使其成为一种理想的通用基础模型。

2.6.2 训练时更新 TTT

在处理长上下文序列时，上述基于 SSM 范式的架构（例如 RWKV 和 Mamba）通过将上下文信息压缩到固定长度的隐藏状态中，成功将计算复杂度降低至线性级别，有效扩展了模型处理长上下文的能力。然而，随着上下文长度的持续增长，基于 SSM 范式的模型可能会过早出现性能饱和。例如，Mamba 在上下文长度超过 16k 时，困惑度基本不再下降 [37]。出现这一现象的原因可能是固定长度的隐藏状态限制了模型的表达能力，同时在压缩过程中可能会导致关键信息的遗忘。

为了解决这一限制，测试时训练（Test-Time Training，TTT）[37] 范式提供了一种有效的解决方案。TTT 利用模型本身的参数来存储隐藏状态、记忆上文；并在每一步推理中，对模型参数进行梯度更新，已实现上文的不断循环流入，如图2.20所示。这个过程不同于传统的机器学习范式中模型在完成训练后的推理阶段通常保持静态的方式，TTT 在推理阶段会针对每一条测试数据一边循环训练一边推理。为了实现这种测试时训练的机制，TTT 在预训练和推理阶段均进行了独特的设计。

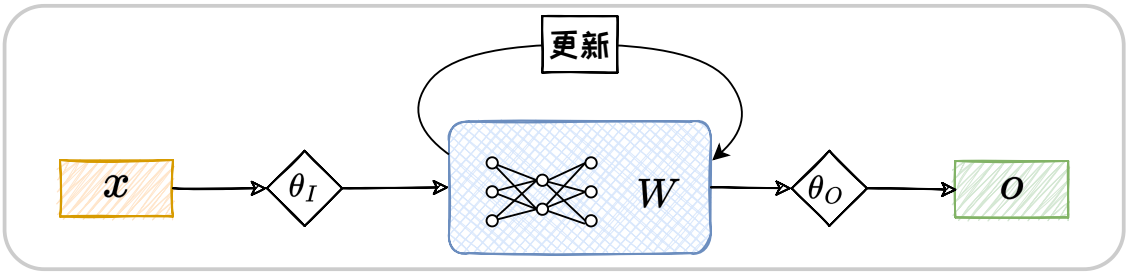


图 2.20: TTT 范式下的推理流程。

在 TTT 范式的预训练阶段，训练过程包含内部循环以及外部循环两个部分。其中外部循环遵循传统的下词预测任务，通过自回归方式优化模型全局权重参数。内部循环则是基于自监督的方式来优化隐藏状态。具体来说，模型需要在每个时间步动态地更新隐藏状态，使其能够不断适应新的输入数据。这种动态更新的机制类似于一个独立的机器学习模型在每个时间步对输入进行训练和优化。给定当前时间步输入 x_t 以及先前历史上下文 $x_1, x_2, \ldots, x_{t-1}$ 对应的隐藏状态 W_{t-1}，模型计算当前时间步的重构损失：

$$\ell\left(W_{t-1}; x_t\right) = \left\| f\left(\theta_K x_t; W_{t-1}\right) - \theta_V x_t \right\|^2 , \tag{2.11}$$

其中 θ_K 和 θ_V 是通过外部循环学习到的参数。接着，模型以学习率 η 利用该损失进行梯度下降，更新隐藏状态：

$$W_t = W_{t-1} - \eta \nabla \ell\left(W_{t-1}; x_t\right) 。 \tag{2.12}$$

最终，模型基于更新后的隐藏状态和当前输入生成输出：

$$z_t = f(x_t; W_t)。 \tag{2.13}$$

在推理阶段，无需执行外部循环任务。因此，模型只进行内部循环来对隐藏状态进行更新，使模型更好地适应新的数据分布，从而提升预测性能。

与 Transformer 相比，基于 TTT 范式的模型具有线性时间复杂度，这对于处理长序列数据至关重要。相较于基于 SSM 的 RWKV 和 Mamba 架构，TTT 通过模型

参数来保存上下文信息，能够更有效地捕捉超长上下文中的语义联系和结构信息。因此，TTT 在长上下文建模任务中展现出卓越的性能，特别是在需要处理超长上下文的应用场景中。未来，TTT 范式有望在超长序列处理任务中发挥重要作用。

参考文献

[1] Joshua Ainslie et al. “Gqa: Training generalized multi-query transformer models from multi-head checkpoints”. In: *arXiv preprint arXiv:2305.13245* (2023).

[2] Rohan Anil et al. “Palm 2 technical report”. In: *arXiv preprint arXiv:2305.10403* (2023).

[3] Tim Bayne and Iwan Williams. “The Turing test is not a good benchmark for thought in LLMs”. In: *Nature Human Behaviour* 7.11 (2023), pp. 1806–1807.

[4] Sid Black et al. “Gpt-neox-20b: An open-source autoregressive language model”. In: *arXiv preprint arXiv:2204.06745* (2022).

[5] Tom Brown et al. “Language models are few-shot learners”. In: *NeurIPS*. 2020.

[6] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[7] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *Journal of Machine Learning Research* 24.240 (2023), pp. 1–113.

[8] Hyung Won Chung et al. “Scaling instruction-finetuned language models”. In: *Journal of Machine Learning Research* 25.70 (2024), pp. 1–53.

[9] Kevin Clark et al. “Electra: Pre-training text encoders as discriminators rather than generators”. In: *arXiv preprint arXiv:2003.10555* (2020).

[10] Tim Dettmers et al. “Qlora: Efficient finetuning of quantized llms”. In: *NeurIPS*. 2024.

[11] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *NAACL*. 2019.

[12] Albert Gu and Tri Dao. “Mamba: Linear-Time Sequence Modeling with Selective State Spaces”. In: *arXiv preprint arXiv:2312.00752* (2023).

[13] Albert Gu, Karan Goel, and Christopher Ré. “Efficiently modeling long sequences with structured state spaces”. In: *arXiv preprint arXiv:2111.00396* (2021).

[14] Albert Gu et al. “On the Parameterization and Initialization of Diagonal State Space Models”. In: *NeurIPS*. 2022.

[15] Jordan Hoffmann et al. “Training compute-optimal large language models”. In: *arXiv preprint arXiv:2203.15556* (2022).

[16] Jared Kaplan et al. “Scaling laws for neural language models”. In: *arXiv preprint arXiv:2001.08361* (2020).

[17] Zhenzhong Lan et al. “Albert: A lite bert for self-supervised learning of language representations”. In: *arXiv preprint arXiv:1909.11942* (2019).

[18] Mike Lewis et al. “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension”. In: *ACL*. 2020.

[19] Haotian Liu et al. “Visual instruction tuning”. In: *NeurIPS*. 2024.

[20] Qi Liu, Matt J Kusner, and Phil Blunsom. “A survey on contextual embeddings”. In: *arXiv preprint arXiv:2003.07278* (2020).

[21] Tiedong Liu and Bryan Kian Hsiang Low. “Goat: Fine-tuned llama outperforms gpt-4 on arithmetic tasks”. In: *arXiv preprint arXiv:2305.14201* (2023).

[22] Yinhan Liu et al. “Multilingual denoising pre-training for neural machine translation”. In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 726–742.

[23] Yinhan Liu et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[24] Ha-Thanh Nguyen. “A brief report on lawgpt 1.0: A virtual legal assistant based on gpt-3”. In: *arXiv preprint arXiv:2302.05729* (2023).

[25] Long Ouyang et al. “Training language models to follow instructions with human feedback”. In: *NeurIPS*. 2022.

[26] Bo Peng et al. “RWKV: Reinventing RNNs for the Transformer Era”. In: *EMNLP*. 2023.

[27] Alec Radford et al. “Improving language understanding by generative pre-training”. In: (2018).

[28] Rafael Rafailov et al. “Direct preference optimization: Your language model is secretly a reward model”. In: *NeurIPS*. 2024.

[29] Colin Raffel et al. “Exploring the limits of transfer learning with a unified text-to-text transformer”. In: *Journal of machine learning research* 21.140 (2020), pp. 1–67.

[30] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023).

[31] Victor Sanh et al. “Multitask prompted training enables zero-shot task generalization”. In: *arXiv preprint arXiv:2110.08207* (2021).

[32] Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. “Are emergent abilities of large language models a mirage?” In: *NeurIPS*. 2024.

[33] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[34] Noam Shazeer. “Glu variants improve transformer”. In: *arXiv preprint arXiv:2002.05202* (2020).

[35] Shaden Smith et al. “Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model”. In: *arXiv preprint arXiv:2201.11990* (2022).

[36] Jianlin Su et al. “Roformer: Enhanced transformer with rotary position embedding”. In: *arXiv preprint arXiv:2104.09864* (2021).

[37] Yu Sun et al. “Learning to (Learn at Test Time): RNNs with Expressive Hidden States”. In: *arXiv preprint arXiv:2407.04620* (2024).

[38] Corentin Tallec and Yann Ollivier. “Can recurrent neural networks warp time?” In: *ICLR*. 2018.

[39] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).

[40] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).

[41] Trieu H Trinh and Quoc V Le. “A simple method for commonsense reasoning”. In: *arXiv preprint arXiv:1806.02847* (2018).

[42] Ashish Vaswani et al. “Attention is all you need”. In: *NeurIPS*. 2017.

[43] Linting Xue et al. “mT5: A massively multilingual pre-trained text-to-text transformer”. In: *NAACL*. 2021.

[44] Aiyuan Yang et al. “Baichuan 2: Open large-scale language models”. In: *arXiv preprint arXiv:2309.10305* (2023).

[45] Deyao Zhu et al. “Minigpt-4: Enhancing vision-language understanding with advanced large language models”. In: *arXiv preprint arXiv:2304.10592* (2023).

[46] Yukun Zhu et al. “Aligning books and movies: Towards story-like visual explanations by watching movies and reading books”. In: *ICCV*. 2015.

ZHEJIANG UNIVERSITY
1897

3 Prompt 工程

随着模型训练数据规模和参数数量的持续增长，大语言模型突破了泛化瓶颈，并涌现出了强大的指令跟随能力。泛化能力的增强使得模型能够处理和理解多种未知任务，而指令跟随能力的提升则确保了模型能够准确响应人类的指令。两种能力的结合，使得我们能够通过精心编写的指令输入，即 Prompt，来引导模型适应各种下游任务，从而避免了传统微调方法所带来的高昂计算成本。Prompt 工程，作为一门专注于如何编写这些有效指令的技术，成为了连接模型与任务需求之间的桥梁。它不仅要求对模型有深入的理解，还需要对任务目标有精准的把握。通过 Prompt 工程，我们能够最大化地发挥大语言模型的潜力，使其在多样化的应用场景中发挥出卓越的性能。本章将深入探讨 Prompt 工程的概念、方法及作用，并介绍上下文学习、思维链等技术，以及 Prompt 工程的相关应用。

* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

3.1 Prompt 工程简介

传统的自然语言处理研究遵循“预训练-微调-预测”范式，即先在大规模语料库上作预训练，然后在下游任务上微调，最后在微调后的模型上进行预测。然而，随着语言模型在规模和能力上的显著提升，一种新的范式——“预训练-提示预测”应运而生，即在预训练模型的基础上，通过精心设计 Prompt 引导大模型直接适应下游任务，而无需进行繁琐微调，如图 3.1所示。在这一过程中，Prompt 的设计将对模型性能产生深远影响。这种专注于如何编写 Prompt 的技术，被称为 **Prompt 工程**。在本节中，我们将深入介绍 Prompt 工程的定义及其相关概念，探讨其在自然语言处理领域中的重要性和应用。

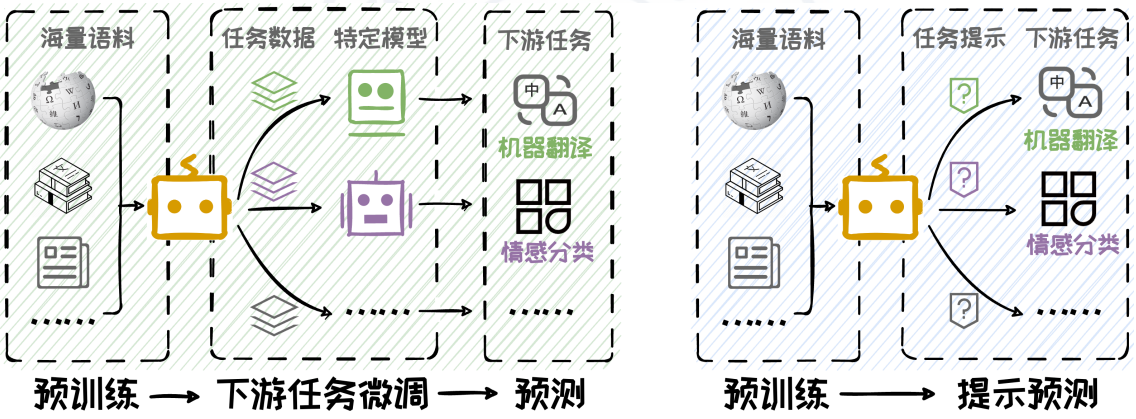


图 3.1: “预训练-微调-预测”范式与“预训练-提示预测”范式对比。

3.1.1 Prompt 的定义

Prompt 是指用于指导生成式人工智能模型执行特定任务的输入指令[1]，这些指令通常以自然语言文本的形式出现。Prompt 的核心目的是清晰地描述模型应该执行的任务，以引导模型生成特定的文本、图像、音频等内容。如图 3.2所示，通过

[1] https://en.wikipedia.org/wiki/Prompt_engineering

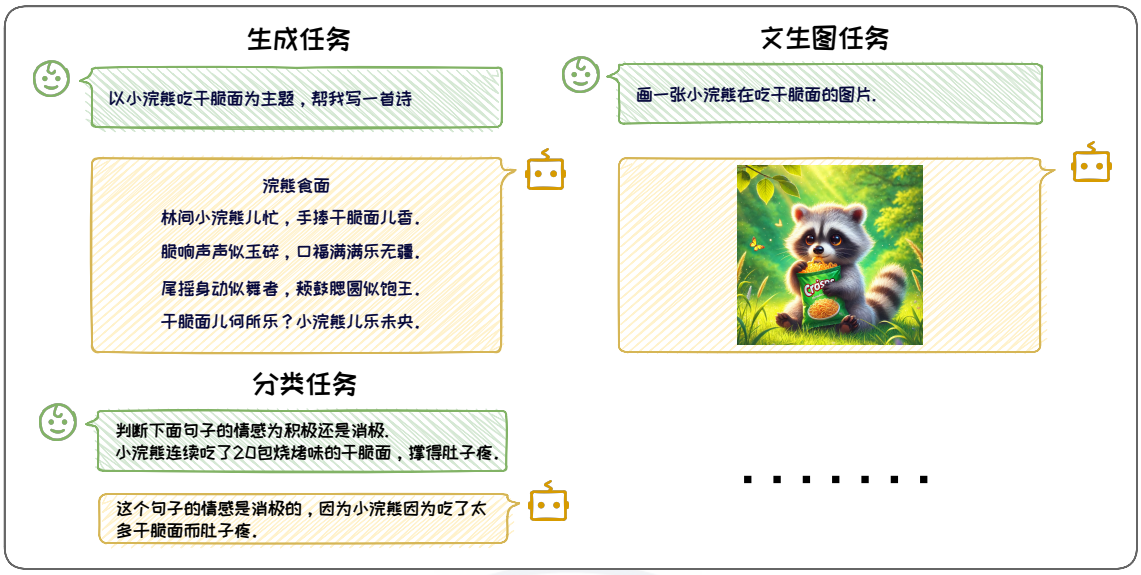


图 3.2: 几种常见的 Prompt 例子。

精心设计的 Prompt，模型能够实现多样化的功能。例如，通过提供明确的情感分类指令，模型能够准确地对文本进行情感分析；通过特定主题的创作指令，模型能够生成富有创意的诗歌。此外，在多模态模型的应用场景中，Prompt 还可以包含画面描述，从而指导模型生成相应的视觉作品。

Prompt 的应用范围广泛，不仅限于文本到文本的任务。由于本书主要关注语言模型，本章节将聚焦于**文本生成模型**，并深入探讨如何通过精心设计的 Prompt 来引导模型生成符合特定任务要求的文本输出。

3.1.2 Prompt 工程的定义

Prompt 工程（Prompt Engineering），又称提示工程，是指设计和优化用于与生成式人工智能模型交互的 Prompt 的过程 [2]。这种技术的核心在于，将新任务通过 Prompt 构建为模型在预训练阶段已经熟悉的形式，利用模型固有的泛化能力来执行新的任务，而无需在额外的特定任务上进行训练。Prompt 工程的成功依赖于对

[2] https://en.wikipedia.org/wiki/Prompt_engineering

预训练模型的深入理解，以及对任务需求的精确把握。通过构造合适的 Prompt 输入给大语言模型，大语言模型能够帮助我们完成各种任务 [47]。

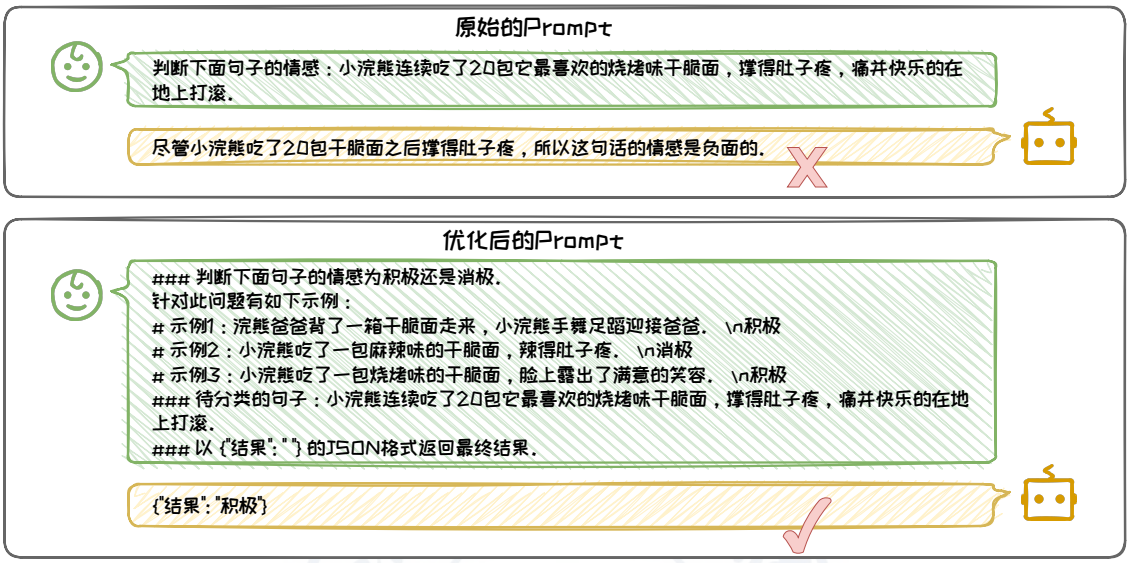


图 3.3: Prompt 工程技术应用前后的效果对比。

如图 3.3所示，通过 Prompt 工程的优化，原始 Prompt 被改写为更加全面、规范的形式。优化后的 Prompt 能够显著提升模型生成回答的质量。因此，在与大语言模型互动过程中构建优质且全面的 Prompt 至关重要，它直接决定了能否获得有价值的输出。经过良好设计的 Prompt 通常由任务说明、上下文、问题、输出格式四个基本元素组成：

- **任务说明**——向模型明确提出具体的任务要求。任务说明应当清晰、直接，并尽可能详细地描述期望模型完成的任务。
- **上下文**——向模型提供的任务相关背景信息，用以增强其对任务的理解以及提供解决任务的思路。上下文可以包括特定的知识前提、目标受众的背景、相关任务的示例，或任何有助于模型更好地理解任务的信息。

- **问题**——向模型描述用户的具体问题或需要处理的信息。这部分应直接涉及用户的查询或任务，为模型提供一个明确的起点。问题可以是显式的提问，也可以是隐式的陈述句，用以表达用户的潜在疑问。
- **输出格式**——期望模型给出的回答的展示形式。这包括输出的格式，以及任何特定的细节要求，如简洁性或详细程度。例如，可以指示模型以 JSON 格式输出结果。

Prompt 的四个基本元素——任务说明、上下文、问题和输出格式，对于大语言模型生成的效果具有显著影响。这些元素的精心设计和组合构成了 Prompt 工程的核心。在此基础上，Prompt 工程包括多种技巧和技术，如上下文学习（In-Context Learning）和思维链（Chain of Thought）等。这些技巧和技术的结合使用，可以显著提升 Prompt 的质量，进而有效地引导模型生成更符合特定任务需求的输出。具体关于上下文学习的内容将在 3.2节中讨论，思维链的内容将在3.3节中讨论，而 Prompt 的使用技巧将在 3.4节中详细探讨。

然而，随着 Prompt 内容的丰富和复杂化，输入到模型中的 Prompt 长度也随之增加，这不可避免地导致了模型推理速度的减慢和推理成本的上升。因此，在追求模型性能的同时，如何有效控制和优化 Prompt 的长度，成为了一个亟待解决的问题，需要在确保模型性能不受影响的前提下，尽可能压缩输入到大型模型中的 Prompt 长度。为此，LLMLingua [11] 提出了一种创新的由粗到细的 Prompt 压缩方法，该方法能够在不牺牲语义完整性的情况下，将 Prompt 内容压缩至原来的二十分之一，同时几乎不损失模型的性能。此外，随着 RAG 技术的兴起，模型需要处理的上下文信息量大幅增加。FIT-RAG [22] 技术通过高效压缩检索出的内容，成功将上下文长度缩短至原来的 50% 左右，同时保持了性能的稳定，为处理大规模上下文信息提供了有效的解决方案。

3.1.3 Prompt 分词向量化

在构建合适的 Prompt 之后，用户将其输入到大语言模型中，以期得到满意的生成结果。但是，语言模型无法直接理解文本。在 Prompt 进入大模型之前，需要将它拆分成一个 Token 的序列，其中 Token 是**承载语义的最小单元**，标识具体某个词，并且每个 Token 由 Token ID 唯一标识。将文本转化为 Token 的过程称之为**分词**（Tokenization），如图 3.4所示，对于“小浣熊吃干脆面”这样一句话，经过分词处理之后，会变成一个 Token 序列，每个 Token 有对应的 Token ID。

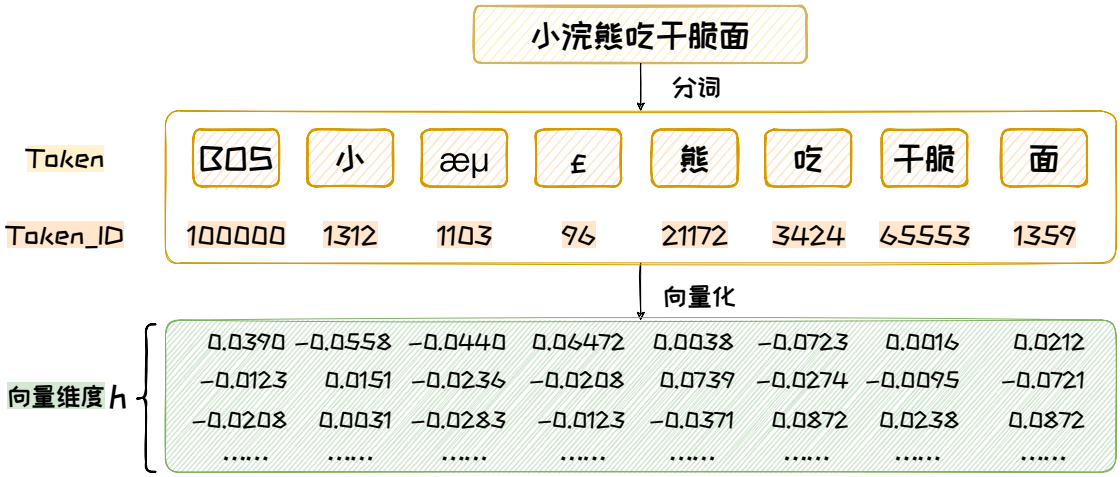


图 3.4: 分词与嵌入的过程，以 DeepSeek-V2-Chat 模型的分词器为例 [6]。

值得注意的是，一句话可能存在多种拆分方式。例如，上述句子可拆分为“小, 浣熊, 吃干, 脆面”。然而，这种拆分方式可能导致语义混乱、不清晰。因此，分词过程颇具挑战性，我们需要精心设计分词方法。为实现有效分词，首先需构建一个包含大语言模型所能识别的所有 Token 的**词表**，并依据该词表进行句子拆分。

在构建大语言模型的词表时，分词器依赖于分词算法，如 BBPE [34]、BPE [8] 和 WordPiece [29] 等，这些算法通过分析语料库中的**词频**等信息来划分 Token。本文将以 BBPE（Byte-Level Byte Pair Encoding）算法为例，阐述其分词过程。BBPE 算法的流程主要包括以下四个步骤：

- **1. 初始化词表**：首先，将所有字符按照其底层编码拆分为若干字节，并将这些单字节编码作为初始词表的 Token。
- **2. 统计词频**：接下来，统计词表中所有 Token 对（即相邻 Token 的组合）的出现频率。在初始阶段，Token 对即为相邻字节的组合。
- **3. 合并高频 Token 对**：然后，选择出现频率最高的 Token 对，将其合并成一个新的 Token 并加入词表。
- **4. 迭代合并**：重复步骤 2 和步骤 3，不断迭代合并，直至达到预设的词表大小或达到指定的合并次数。

我们可以通过设定迭代次数等方法来决定分词粒度，不同的分词粒度导致我们得到不同大小的词表。若**词表过小**，例如仅包含 256 个 Token 表示单字节，能够组合出所有 GBK 编码的汉字和英文，但会导致形态相近但意义迥异的词汇在模型中难以区分，限制了模型承载丰富语义的能力，并大幅增加生成的序列长度。相反，若**词表过大**，虽然能涵盖更多长尾词汇，但可能导致对这些词汇的学习不够深入，且难以有效捕捉同一单词不同形态之间的关联。因此，在构建词表时，需在涵盖广泛词汇与保持语义精细度之间找到恰当的平衡点，以确保大语言模型既能学习到丰富的词汇知识，又能准确理解和生成具有复杂语义的文本。

具体来说，如图 3.4所示，为了有助于模型更准确地理解词义，同时减少生成常用词所需的 Token 数量，词表中收录了语料库中**高频**出现的词语或短语，形成独立的 Token。例如，“干脆”一词在词表中以一个 Token 来表示。为了优化 Token 空间并压缩词表大小，构建词表时会包含了一些特殊的 Token，这些 Token 既能单独表示语义，也能通过两两组合，表示语料库中**低频**出现的生僻字。例如，“浣”字使用两个 Token，“æµ”和“£”，来表示。通过这种处理方式，词表既能涵盖常见的高频词汇，又能通过 Token 组合灵活表达各类稀有字符。由此可见，词表构建在一定程度上受到“先验知识”的影响，这些知识源自人类语料库的积累与沉淀。

每个大语言模型都有自己的分词器，分词器维护一个词表，能够对文本进行分词。分词器的质量对模型的性能有着直接的影响。一个优秀的分词器不仅能显著提升模型对文本的理解能力，还能够提高模型的处理速度，减少计算资源的消耗。一个好的分词器应当具备以下特点：首先，它能够准确地识别出文本中的关键词和短语，从而帮助模型更好地捕捉语义信息；其次，分词器的效率直接影响到模型的训练和推理速度，一个高效的分词器能够实现对文本 Token 的优化压缩，进而显著缩短模型在处理数据时所需的时间。

表 3.1: 模型分词器对比表

模型	词表大小	中文分词效率（字 / Token）	英文分词效率（词 / Token）
LLaMA1	32000	0.6588	0.6891
LLaMA2	32000	0.6588	0.6891
LLaMA3	128256	1.0996	0.7870
DeepSeek-V1	100016	1.2915	0.7625
DeepSeek-V2	100002	1.2915	0.7625
GPT-3.5 & GPT-4	100256	0.7723	0.7867
GPT-3	50,257	0.4858	0.7522
Qwen-1.5	151646	1.2989	0.7865
StarCoder	49152	0.9344	0.6513

在常用的开源模型中，不同模型采用了不同的分词器，这些分词器具有各自的特点和性能。它们的质量受到多种因素的影响，包括词表的大小、分词的效率等属性。表 3.1是对常见开源大语言模型的分词器的对比分析，其中中文语料库节选自《朱自清散文》[3]，英文语料库来自莫泊桑短篇小说《项链》[4]。我们可以观察到，像 DeepSeek [6]、Qwen [41] 这类中文开源大语言模型，对中文分词进行了优化，平均每个 Token 能够表示 1.3 个字（每个字仅需 0.7 个 Token 即可表示），一些常用词语和成语甚至可以直接用一个 Token 来表示。相比之下，以英文为主要语料的模型，如 GPT-4、LLaMA 系列，对中文的支持度较弱，分词效率不高。在英文

[3] https://www.sohu.com/a/746456997_120075260.

[4] https://americanliterature.com/author/guy-de-maupassant/short-story/the-diamond-necklace

中，由于存在“ly”、“ist”等后缀 Token，一个英文单词通常需要用 1 个及以上的 Token 来表示。单个 Token 承载更多的语义，模型在表达同样的文本时，只需要输出更少的 Token，显著提升了推理效率。

通过这种对比，我们可以清晰地看到不同模型分词器在处理不同语言时的效率，这对于选择合适的模型和优化模型性能具有重要的指导意义。

在完成分词之后，这些 Token 随后会经过模型的嵌入矩阵（Embedding Matrix）处理，转化为固定大小的**表征向量**。这些向量序列被直接输入到模型中，供模型理解和处理。在模型生成阶段，模型会根据输入的向量序列计算出词表中每个词的概率分布。模型从这些概率分布中选择并输出对应的 Token，这些 Token 再被转换为相应的文本内容。

上述通过分词技术将文本分割成 Token，再将 Token 转化为特征向量，在高维空间中表征这些文本的处理流程，使得语言模型能够捕捉文本的深层语义结构，并有效地处理和学习各种语言结构，从简单的词汇到复杂的句式和语境。

3.1.4 Prompt 工程的意义

Prompt 工程提供了一种高效且灵活的途径来执行自然语言处理任务。它允许我们无需对模型进行微调，便能有效地完成既定任务，避免微调带来的巨大开销。通过精心设计的 Prompt，我们能够激发大型语言模型的内在潜力，使其在**垂域任务**、**数据增强**、**智能代理**等多个领域发挥出卓越的性能。

1. 垂域任务

应用 Prompt 工程来引导大语言模型完成垂直领域任务，可以避免针对每个任务进行特定微调。不仅可以避免微调模型的高昂计算成本，还可以减少对标注数据的依赖。使得大语言模型可以更好的应用在垂直领域任务中。例如，在 Text-to-SQL 任务中，我们可以应用 Prompt 工程的技巧，引导大语言模型根据用户输入的文本直接生成高质量的 SQL 查询，而无需进行有监督微调。基于 GPT 模型的 Prompt

工程方法在 Spider [44] 榜单上取得了突破性成绩，超越了传统的微调方法。此外，在知识密集型任务问答领域 MMLU [9] 等多个领域，基于 Prompt 工程的方法也取得最佳效果。

2. 数据增强

应用 Prompt 工程通过大语言模型来进行数据增强，不仅能够提升现有数据集的质量，还能够生成新的高质量数据。这些数据可以用于训练和优化其它模型，以将大语言模型的能力以合成数据的方法“蒸馏”到其他模型上。例如，我们可以引导 ChatGPT 模型生成包含丰富推理步骤的数据集，用于增强金融领域 Text-to-SQL 模型的推理能力 [45]。此外，通过精心设计的提示，还能生成包含复杂指令的数据集，如 Alpaca [32] 和 Evol-Instruct [21]。将这些合成的数据集用于微调参数量较小的模型，可以其在保持较小模型尺寸和低计算成本的同时，接近大型模型的性能。

3. 智能代理

应用 Prompt 工程可以将大语言模型构建为智能代理（Intelligent Agent，IA）[5]。智能代理，又叫做智能体，能够感知环境、自主采取行动以实现目标，并通过学习或获取知识来提高其性能。在智能代理进行感知环境、采取行动、学习知识的过程中，都离不开 Prompt 工程。例如，斯坦福大学利用 GPT-4 模拟了一个虚拟西部小镇 [25]，多个基于 GPT-4 的智能体在其中生活和互动，他们根据自己的角色和目标自主行动，进行交流，解决问题，并推动小镇的发展。整个虚拟西部小镇的运转都是由 Prompt 工程驱动的。

本节探讨了 Prompt 的概念，Prompt 工程的概念以及意义，揭示了 Prompt 在大语言模型应用中的关键作用和广阔潜力。接下来，我们将进一步拓展这一主题：第3.2节将探索上下文学习的相关内容，揭示其在提升模型理解和响应能力中的作用；第3.3节将详细介绍思维链提示方法及其变种，展示如何通过这些方法增强模

[5] https://en.wikipedia.org/wiki/Intelligent_agent

型的逻辑推理和问题解决能力；第3.4节将分享构建有效 Prompt 的技巧，指导读者如何设计 Prompt 以激发模型生成更优质的内容；第3.5节将具体展示 Prompt 工程在大语言模型中的实际应用，通过实例阐释其在不同场景下的应用策略和效果。

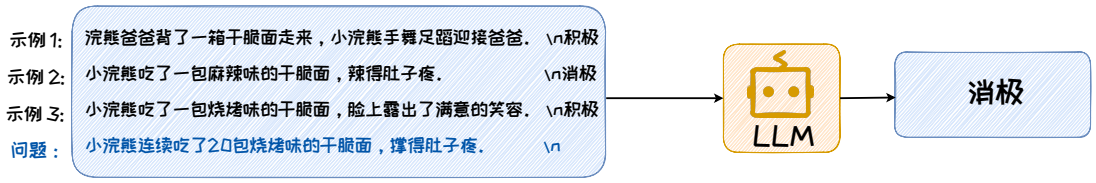
3.2 上下文学习

随着模型训练数据规模和参数数量的持续扩大，大语言模型涌现出了**上下文学习**（In-Context Learning，ICL）能力。其使得语言模型能够通过给定的任务说明或示例等信息来掌握处理新任务的能力。引入上下文学习，我们不再需要针对某个任务训练一个模型或者在预训练模型上进行费时费力的微调，就可以**快速适应**一些下游任务。这使得用户可以仅仅通过页面或者 API 的方式即可利用大语言模型来解决下游任务，为“**语言模型即服务**”（LLM as a Service）模式奠定了坚实的能力基础。本节从**上下文学习的定义**，**演示示例选择**，和**影响其性能的因素**，对上下文学习展开介绍。

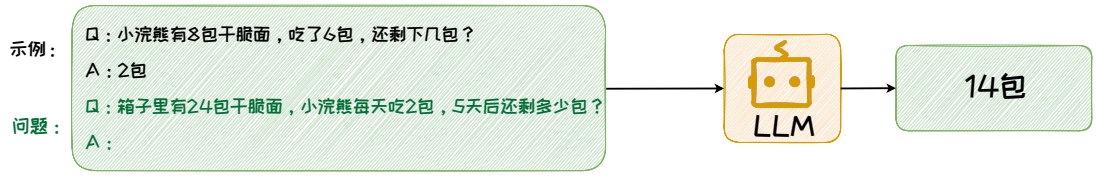
3.2.1 上下文学习的定义

上下文学习（In-Context Learning, ICL）[2] 是一种通过构造特定的 Prompt，来使得语言模型理解并学习下游任务的范式，这些特定的 Prompt 中可以包含**演示示例**，**任务说明**等元素。上下文学习实现的关键在于如何设计有效的 Prompt，以引导模型理解任务的上下文和目标。通常，这些 Prompt 会包含任务说明以及一系列的示例，模型能够从这些上下文信息中学习任务的逻辑和规则，从而在没有额外训练的情况下，生成符合任务要求的输出。基于以上优点，上下文学习被广泛应用于解决垂域任务，数据增强，智能代理等应用中。

在上下文学习中，Prompt 通常包含几个与待解决任务相关的演示示例，以展示任务输入与预期输出。这些示例按照特定顺序组成上下文，并与问题拼接共同



(a) 上下文学习做分类任务。



(b) 上下文学习做生成任务。

图 3.5: 上下文学习示例。

组成 Prompt 输入给大语言模型。大语言模型从上下文中学习任务范式，同时利用模型自身的能力对任务进行作答。在图 3.5的例子 (a) 中，模型被用于文本情感分类任务，给定一段文本，模型能够判断其情感倾向，识别出文本表达的是积极还是消极情绪。图 3.5的例子 (b) 则展示了数学运算任务，模型会仿照示例的形式，直接给出对应的运算结果。按照示例数量的不同，上下文学习可以呈现出多种形式：**零样本**（Zero-shot）上下文学习、**单样本**（One-shot）上下文学习和**少样本**（Few-shot）上下文学习 [2]，如图 3.6所示。

- **零样本上下文学习**：在此种学习方式下，仅需向模型提供任务说明，而无需提供任何示例。其具有强大的场景泛化能力。但零样本学习的性能完全依赖于大语言模型的能力，并且在处理任务时可能表现欠佳。
- **单样本上下文学习**：这种方式仅需为模型提供一个示例，贴合人类“举一反三”的学习模式。不过，单样本学习的效果强依赖于示例相对于任务的代表性。
- **少样本上下文学习**：这种方法通过为模型提供少量的示例（通常为几个至十几个），显著提升模型在特定任务上的表现。但在，示例的增加会显著增加大语言模型推理时的计算成本。示例的代表性和多样性也将影响其生成效果。

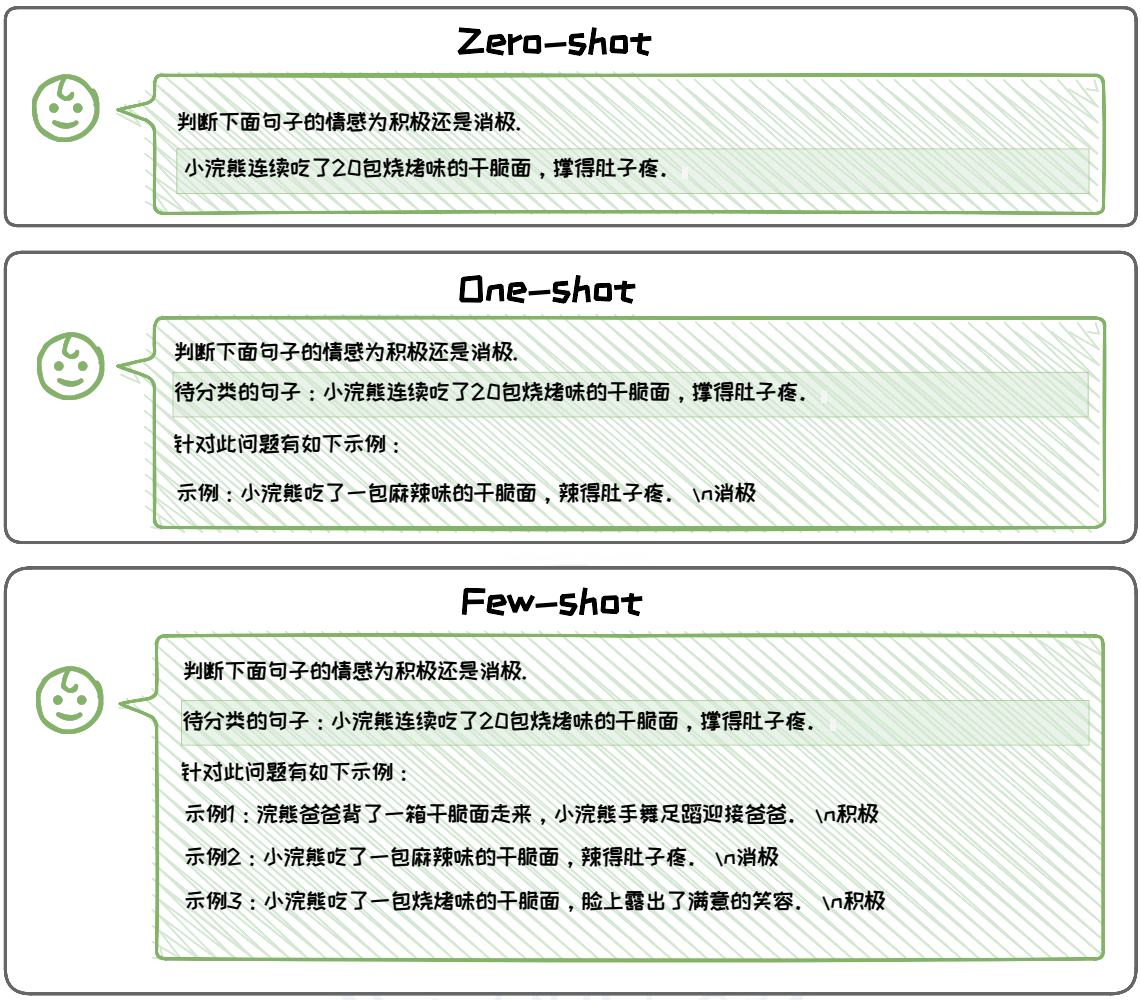


图 3.6: 三种上下文学习形式的示例。

尽管上下文学习在许多任务中表现出色，但它为何奏效仍然是一个重要的研究问题。对此，斯坦福大学的一项研究 [40] 提供了一种解释——“将上下文学习视为隐式贝叶斯推理”。在大语言模型的预训练阶段，模型从大量文本中学习潜在的概念。当运用上下文学习进行推理时，大语言模型借助演示示例来“锚定”其在预训练期间所习得的相关概念，从而进行上下文学习，并对问题进行预测。以图 3.5 为例，模型之所以能给出正确答案，是因为模型在预训练时已经学习到与情感相关的概念，比如积极情感的表现形式（满意的笑容，手舞足蹈.....）、句法结构和句法关系等等，当模型在推理时，借助“浣熊吃了一包麻辣味的干脆面，辣得肚子疼。\n 消极”等示例“锚定”到情感等相关概念，并基于这些概念，给出问题答案。

3.2.2 演示示例选择

在上下文学习中，演示示例在引导大语言模型理解任务中扮演着重要作用，其内容和质量直接影响着学习效果。因此，合理选择演示示例对提升上下文学习性能至关重要。演示示例选择主要依靠相似性和多样性 [20]：

- **相似性**是指精心挑选出与待解决问题最为相近的示例。相似性可以从多个层面进行度量，如语言层面的相似性（包括关键字匹配或语义相似度匹配）、结构相似性等等。通过选取相似的示例，能够为模型提供与待解决问题接近的参照，使大语言模型更易理解该问题。
- **多样性**则要求所选的示例涵盖尽量广的内容，扩大演示示例对待解决问题的覆盖范围。多样化的示例能够帮助模型从不同的角度去理解任务，增强其应对各种问题的能力。

除了相似性和多样性，在某些任务中，还需对任务相关的因素进行考虑。本节主要关注相似性和多样性两个因素，并探讨基于相似性和多样性如何从大量候选示例中选择出合适的演示示例。接下来对基于相似性和多样性的三类示例选择方法 [20] 展开介绍：

1. 直接检索

给定一组候选示例，直接检索的方法依据候选示例与待解决问题间的相似性对候选示例进行排序，然后选取排名靠前的 K 个示例。直接检索的代表性方法是 KATE [17]。如图 3.7 所示，KATE 利用 RoBERTa 对待解决问题和候选示例（移除标签）进行编码。然后通过计算解决问题编码和候选示例编码间的向量余弦相似度对二者的相似度进行评分。基于此评分，选择评分最高的 K 个示例作为上下文学习的演示示例。直接检索的方法简单易操作，是目前应用广泛的示例选择策略之一。但是，其未对示例的多样性进行考虑，选择出的示例可能趋向**同质化**。

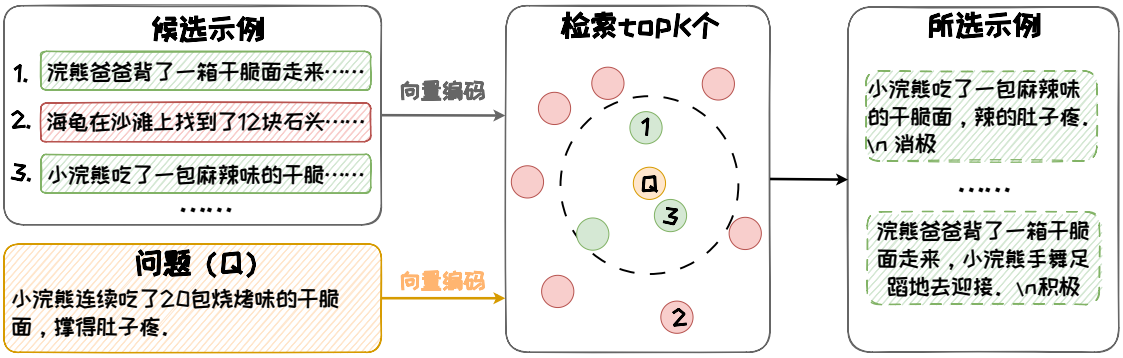


图 3.7: 直接检索。

2. 聚类检索

为缓解直接检索中存在的样例趋同的问题，聚类检索方法采用先聚类后检索的方法来保证检索结果的**多样性**。其先把所有候选示例划分为 K 个簇，然后从每个簇中选取最为相似的一个示例。这样可以有效避免选择出多个相似的示例，从而保障了多样性。Self-Prompting [15] 是其中的代表性方法。如图 3.8 所示，Self-Prompting 首先将候选示例和待解决问题编码成向量形式，接着运用 K-Means 算法把示例集合聚为 K 个簇。依照问题与示例之间的余弦相似度，从每个簇中选取与问题最相似的示例，由此得到 K 个示例。虽然聚类检索方法提高了示例的多样性，但因为有些簇与问题可能并不相似，导致选择的示例的**相似性**可能不够高。

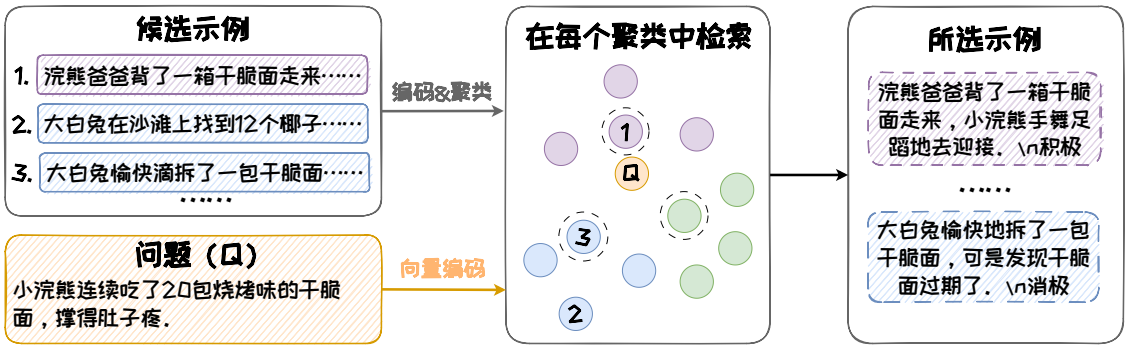


图 3.8: 聚类检索。

3. 迭代检索

直接检索和聚类检索在相似性和多样性之间往往顾此失彼。为了**兼顾相似性多样性**，迭代检索策略应运而生。迭代检索首先挑选与问题高度相似的示例，随后在迭代过程中，结合当前问题和已选示例，动态选择下一个示例，从而确保所选示例的相似性和多样性。RetICL [28] 是迭代检索的代表性方法，如图 3.9所示。RetICL 根据当前问题初始化基于 LSTM [10] 的检索器内部状态，并选择一个示例。接着根据当前问题和所选示例集更新检索器内部状态，并选择下一个示例。这一过程不断迭代，直到得到 k 个示例。尽管迭代检索在计算上相对复杂，但其能够生成更优的示例集，在复杂任务中展现出更好的适应性和灵活性。

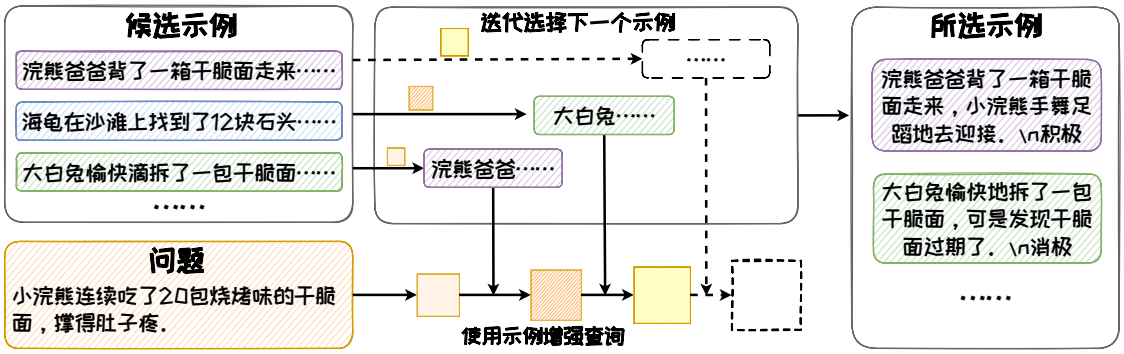


图 3.9: 迭代检索。

除了在示例选择策略上有所不同外，现有的示例选择方法在检索器的选择和设计上亦呈现出差异。一些方法采用现成的检索器，而另一些方法则为了追求更卓越的性能，选择在特定语料库上对检索器进行微调。关于检索器的深入介绍及其分类，将在第六章 6.8 中进行详尽阐述。

3.2.3 性能影响因素

通过精心设计示例选择策略，上下文学习的效果可以得到显著提升。但是，除了示例选择以外，上下文学习的性能仍受到多种因素的共同影响。这些因素涉及括**预训练数据**、**预训练模型**，以及**演示示例**等多个方面 [48]。本小节将讨论这些关键因素如何影响上下文学习的效果。

1. 预训练数据的影响

预训练数据是上下文学习能力的来源，深刻影响着上下文学习的性能。对预训练数据而言，以下三方面因素是影响上下文学习性能的关键：

- **领域丰富度**：预训练数据的所覆盖的领域的丰富度直接影响模型的领域泛化能力。在丰富的跨领域语料库进行预训练的模型具备更稳定的上下文学习能力。而单一领域的语料库可能限制模型的适应性，即使该领域与目标任务高度相关，也无法保证最佳的上下文学习性能 [31]。
- **任务多样性**：预训练数据中的任务多样性是提升上下文学习性能的重要因素。多样化的任务类型有助于模型学习到更广泛的知识和技能，增强其任务泛化能力，从而在面对新任务时表现也更为出色 [26]。
- **训练数据的分布特性**：训练数据中存在突发性分布和罕见类别时，能够增强模型的上下文学习能力。突发性分布使得“文本-标签映射”的数据模式在整个训练过程中得以反复出现，罕见类别增强模型处理少见或新颖输入的能力，从而提升模型上下文学习的表现 [4]。

2. 预训练模型的影响

预训练模型对上下文学习的性能影响主要体现在**模型参数规模**上。当模型参数达到一定的规模时，上下文学习才能得以涌现。通常，模型的参数数量需达到亿级别及以上。常见的拥有上下文学习能力的模型中，规模最小的是来自阿里巴巴通义实验室的 Qwen2-0.5B 模型，具有 5 亿参数。一般而言，模型的规模越大，其上下文学习性能也越强 [39]。此外，模型的架构和训练策略也是影响上下文学习性能的重要因素。

3. 演示示例的影响

在 3.2.2节中，我们已经讨论了演示示例选择对上下文学习的重要性。接下来，我们将继续探讨**演示示例的格式**、**输入-标签映射**、**示例数量及顺序**对上下文学习的影响。

示例格式。不同的任务对于示例格式的要求不同。如图3.5中展示的用上下文学习做情感分类的例子，我们只需要给出输入以及输出。但是，一些复杂推理任务，例如算术、代码等，仅仅给出输入和输出不足以让模型掌握其中的推理过程。在这种情况下，以思维链的形式构造示例，即在输入和输出之间添加中间推理步骤，能帮助模型逐步推理并学习到更复杂的映射关系，思维链将在3.3 详细介绍。

输入-输出映射的正确性。对于一个包含输入、输出的示例，大语言模型旨在从中学习到输入-输出映射，以完成目标任务。如果给定示例中的输入-输出映射是错误的，势必会对上下文学习的效果产生影响。例如在情感分类任务中，将积极的文本错误地标记为消极，模型可能会学习到这种错误的映射方式，从而在下游任务中产生错误。对输入-输出映射错误的敏感性与模型规模大小有关 [14, 43]。较大的模型在上下文学习中对输入-输出映射的正确性表现出更高的敏感性 [14]。相比之下，较小的模型输入-输出映射错误的敏感性较弱 [24, 39]。

演示示例的数量和顺序。增加演示示例的数量通常能够提升上下文学习性能，但随着示例数量的增多，性能提升的速率会逐渐减缓 [23]。此外，相较于分类任务，生成任务更能从增加的示例数量中获益 [16]。此外，演示示例的顺序同样是影响上下文学习性能的关键因素，不同示例顺序下的模型表现存在显著差异。最优的示例顺序具有模型依赖性，即对某一模型有效的示例顺序未必适用于其他模型 [19]。同时，示例顺序的选择也受到所使用数据集的特定特性影响 [17]。

除上述因素外，Prompt 中的任务说明的质量也直接影响上下文学习的效果。清晰、明确的任务说明能够为模型提供明确指导，从而提升上下文学习的性能 [43]。因此，在设计演示示例和任务说明时，应综合考虑这些因素，以优化模型的表现。

本节介绍了上下文学习，区分了零样本、单样本和少样本三种形式。探讨了直接检索、聚类检索和迭代检索三种示例选择策略，分析其优缺点及代表性方法。最后从预训练数据、模型本身及演示示例等方面分析影响上下文学习性能的因素。

3.3 思维链

随着语言模型参数规模的持续扩张，其可以更好的捕捉语言特征和结构，从而在语义分析、文本分类、机器翻译等自然语言处理任务中的表现显著增强。但是，在面对算术求解、常识判断和符号推理等需要复杂推理能力的任务时，模型参数规模的增长并未带来预期的性能突破，这种现象被称作“Flat Scaling Curves”[38]。这表明，仅靠模型规模的扩大不足以解决所有问题，我们需要探索新的方法以提升模型的推理能力和智能水平。人类在解决复杂问题时，通常会逐步构建推理路径以导出最终答案。基于这一理念，一种创新的 Prompt 范式——**思维链提示**（Chain-of-Thought，CoT）[38] 被用于引导模型进行逐步推理。CoT 可以显著提升大语言模型处理复杂任务中的表现，从而突破“Flat Scaling Curves”的限制，激发大语言模型的内在推理潜能。

3.3.1 思维链提示的定义

思维链提示（Chain-of-Thought，CoT）[38] 通过模拟人类解决复杂问题时的思考过程，引导大语言模型在生成答案的过程中引入一系列的**中间推理步骤**。这种方法不仅能够显著提升模型在推理任务上的表现，而且还能够揭示模型在处理复杂问题时的内部逻辑和推理路径。

CoT 方法的核心是构造**合适的 Prompt** 以触发大语言模型**一步一步生成推理路径**，并生成最终答案。早期方法在构造 Prompt 时，加入少量包含推理过程的样本示例（Few-Shot Demonstrations）[38]，来引导模型一步一步生成答案。在这些示例中，研究者精心编写在相关问题上的推理过程，供模型模仿、学习。这种方法使得模型能够从这些示例中学习如何生成推理步骤，一步步输出答案。图 3.10展示了一个用于求解数学问题的 CoT 形式的 Prompt 的例子。其中，样例给出了与待求

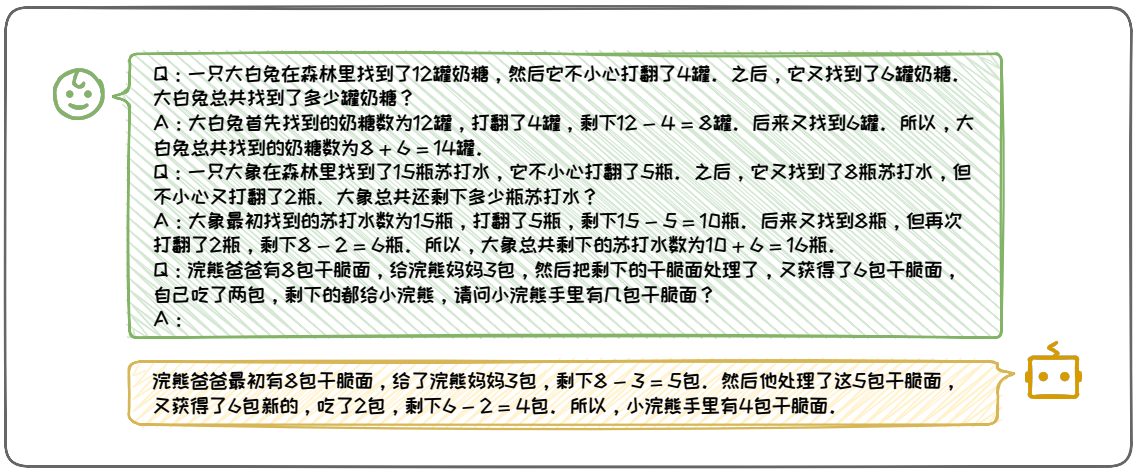


图 3.10: 包含少量样本示例的 CoT 提示示例。

解问题相关的数学问题的解题步骤作为参考，大语言模型会模仿此样例对复杂的数学计算一步一步进行求解。通过引入 CoT，大语言模型在解算术求解、常识判断和符号推理等复杂问题上性能显著提升。并且在 CoT 的加持下，大语言模型处理复杂问题的能力随着模型参数规模的变大而增强。

在 CoT 核心思想的指引下，衍生出了一系列的扩展的方法。这些扩展的方法按照其推理方式的不同，可以归纳为三种模式：**按部就班**、**三思后行**和**集思广益**。这几种模式的对比如图 3.11所示。

- **按部就班**。在按部就班模式中，模型一步接着一步地进行推理，推理路径形成了一条逻辑连贯的链条。在这种模式下，模型像是在遵循一条预设的逻辑路径，“按部就班”的一步步向前。这种模式以 CoT [38]、Zero-Shot CoT [12]、Auto-CoT [46] 等方法为代表。
- **三思后行**。在三思后行模式中，模型每一步都停下来估当前的情况，然后从多个推理方向中选择出下一步的行进方向。在这种模式下，模型像是在探索一片未知的森林，模型在每一步都会停下来评估周围的环境，“三思后行”以找出最佳推理路径。这种模式以 ToT [42]、GoT [1] 等方法为代表。

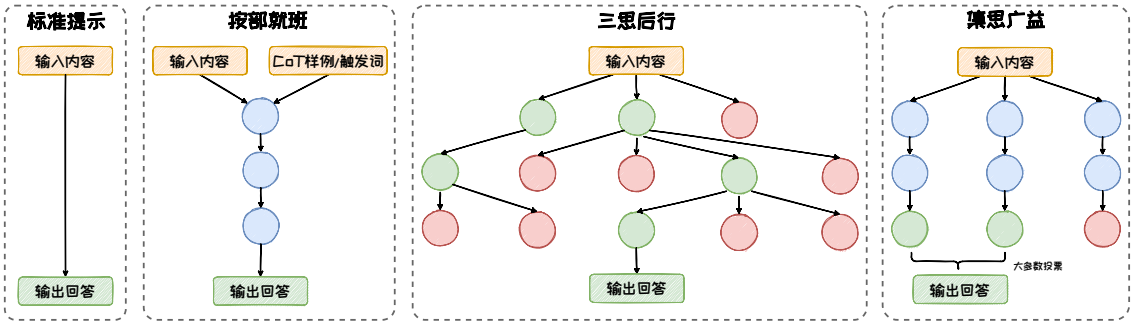


图 3.11: 不同 CoT 结构的对比。

- **集思广益**。在集思广益模式中，模型同时生成多条推理路径并得到多个结果，然后整合这些结果，得到一个更为全面和准确的答案。在这种模式下，模型像是在召开一场智者的会议，每个智者都带来了自己的见解，最终通过讨论和整合，“集思广益”得出一个更优的结论。这一类模式以 Self-Consistency [36] 等方法为代表。

3.3.2 按部就班

按部就班模式强调的是**逻辑的连贯性**和**步骤的顺序性**。在这种模式下，模型一步接着一步的进行推理，最终得到结论。其确保了推理过程的清晰和有序，使得模型的决策过程更加透明和可预测。原始的少样本思维链（CoT）方法就采用了按部就班模式。其通过手工构造几个一步一步推理回答问题的例子作为示例放入 Prompt 中，来引导模型一步一步生成推理步骤，并生成最终的答案。这种方法在提升模型推理能力方面取得了一定的成功，但是需要费时费力地手工编写大量 CoT 示例，并且过度依赖于 CoT 的编写质量。针对这些问题，研究者在原始 CoT 的基础上进行了扩展，本节将介绍 CoT 的两种变体：Zero-Shot CoT 和 Auto-CoT。

1. Zero-Shot CoT

Zero-Shot CoT [12] 通过简单的提示，如 “Let’s think step by step”，引导模型自行生成一条推理链。其无需手工标注的 CoT 示例，减少了对人工示例的依赖，多个推理任务上展现出了与原始少样本 CoT 相媲美甚至更优的性能。

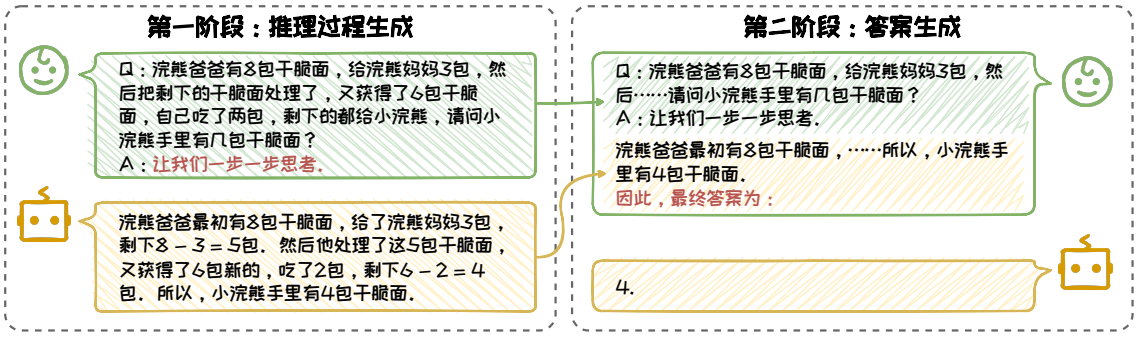


图 3.12: Zero-Shot CoT 提示的流程。

Zero-Shot CoT 整体流程如图 3.12所示，它使用两阶段方法来回答问题。首先，在第一阶段，在问题后面跟上一句 “让我们一步一步思考” 或者 “Let’s think step by step” 来作为 CoT 的提示触发词，来指示大语言模型先生成中间推理步骤，再生成最后的答案。在第二阶段，把原始的问题以及第一阶段生成的推理步骤拼接在一起，在末尾加上一句 “Therefore, the answer is” 或者 “因此，最终答案为”，把这些内容输给大语言模型，让他输出最终的答案。通过这样的方式，无需人工标注 CoT 数据，即可激发大语言模型内在的推理能力。大语言模型能够逐步推理出正确的答案，展现了 Zero-Shot CoT 在提升模型推理能力方面的潜力。

2. Auto CoT

在 Zero-Shot CoT 的基础之上，Auto-CoT [46] 引入与待解决问题相关的问题及其推理链作为示例，以继续提升 CoT 的效果。相关示例的生成过程是由大语言模型自动完成的，无需手工标注。Auto-CoT 的流程如图 3.13所示，其包含以下步骤：

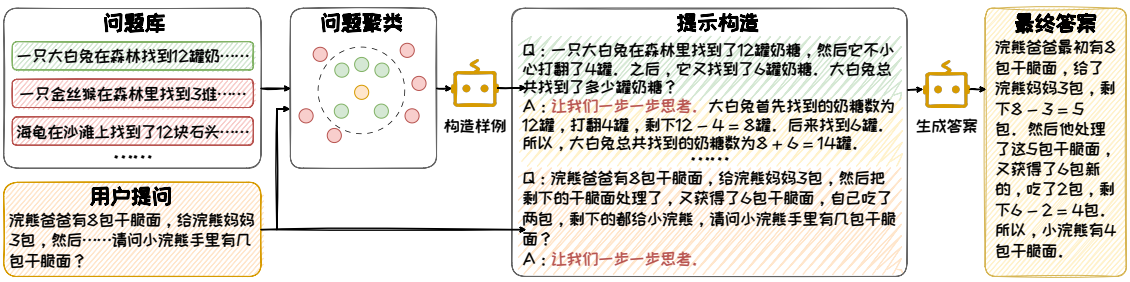


图 3.13: Auto-CoT 提示的流程。

- 利用聚类技术从问题库中筛选出与用户提问位于一个簇中的问题。
- 然后，借助 Zero-Shot CoT 的方式，为筛选出的问题生成推理链，形成示例。这些示例包含了不同问题及其对应的推理内容，可为模型提供不同解题思路，辅助模型做出更为审慎的推理。
- 在这些示例的基础上，Auto-CoT 以“让我们一步一步思考”引导大语言模型生成针对用户问题的推理链和答案。

3.3.3 三思后行

三思后行模式强调的是在决策过程中的融入**审慎**和**灵活性**。在这种模式下，模型在每一步都会停下来评估当前的情况，判断是否需要调整推理方向。这种模式的核心在于允许模型在遇到困难或不确定性时进行回溯和重新选择，确保决策过程的稳健性和适应性。其模仿了人类在解决问题时，会有一个反复选择回溯的过程。人们会不断从多个候选答案中选择最好的那个，并且如果一条思维路子走不通，就会回溯到最开始的地方，选择另一种思维路子进行下去。基于这种现实生活的观察，研究者在 CoT 的基础上提出了思维树（Tree of Thoughts, ToT）[42]、思维图（Graph of Thoughts, GoT）[1] 等三思后行模式下的 CoT 变体。

ToT 将推理过程构造为一棵思维树，其从以下四个角度对思维树进行构造。

- **拆解**。将复杂问题拆分成多个简单子问题，每个子问题的解答过程对应一个思维过程。思维过程拆解的形式和粒度依任务类别而定，例如在数学推理任务上以一行等式作为一个思维过程，在创意写作任务上以内容提纲作为思维过程。
- **衍生**。模型需要根据当前子问题生成可能的下一步推理方向。衍生有两种模式：样本启发和命令提示。样本启发以多个独立的示例作为上下文，增大衍生空间，适合于创意写作等思维空间宽泛的任务；命令提示在 Prompt 中指明规则和要求，限制衍生空间，适用于 24 点游戏等思维空间受限的任务。

- **评估**。利用模型评估推理节点合理性。根据任务是否便于量化评分，选择投票或打分模式。投票模式中，模型在多节点中选择，依据票数决定保留哪些节点；打分模式中，模型对节点进行评分，依据评分结果决定节点的保留。
- **搜索**。从一个或多个当前状态出发，搜索通往问题解决方案的路径。依据任务特点选择不同搜索算法。可以使用深度优先搜索、广度优先搜索等经典搜索算法，也可以使用 A* 搜索、蒙特卡洛树搜索等启发式搜索算法。

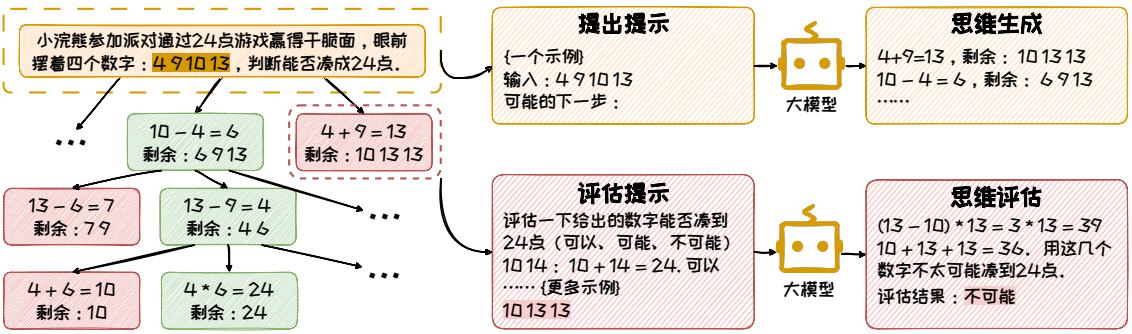


图 3.14: ToT 提示的流程。

图 3.14通过 24 点游戏展示了一个 ToT 的具体例子。在这个例子中，给定 4 个数字，然后让大语言模型利用加减乘除（+-*/）四个运算符来组合这四个数字，使得最终的运算结果为 24。首先，ToT 基于当前所剩下的数字，通过上下文学习让模型选择两个数字作运算，并生成多个方案，在图上表现为思维树的多个子节点。之后以广度优先搜搜的方式遍历每一个子节点，评估当前剩余的数字是否能够凑到 24 点，保留可能凑出 24 点的节点，这一步也是通过上下文学习的方式来实现的。不断重复上述两个步骤，直到得出最终合理的结果。

在 ToT 的基础上，GoT 将树扩展为有向图，以提供了每个思维自我评估修正以及思维聚合的操作。该图中，顶点代表某个问题（初始问题、中间问题、最终问题）的一个解决方案，有向边代表使用“出节点”作为直接输入，构造出思维“入节点”的过程。GoT 相比于 ToT 的核心优势是其思维自我反思，以及思维聚合的能力，能够将来自不同思维路径的知识和信息进行集成，形成综合的解决方案。

3.3.4 集思广益

集思广益模式强调的是通过**汇集多种不同的观点和方法**来优化决策过程。在这种模式下，模型不仅仅依赖于单一的推理路径，而是通过探索多种可能的解决方案，从中选择最优的答案。这种方法借鉴了集体智慧的概念，即通过整合多个独立的思考结果，可以得到更全面、更准确的结论。在集体智慧的启发下，研究者在 CoT 的基础上探讨了如何通过自洽性来增强模型的推理能力，提出了 Self-Consistency [36] 方法。其引入多样性的推理路径并从中选择最一致的答案，从而提高了模型的推理准确性。Self-Consistency 不依赖于特定的 CoT 形式，可以与其他 CoT 方法兼容，共同作用于模型的推理过程。

如图 3.15所示，Self-Consistency 的实现过程可以分为三个步骤：(1) 在随机采样策略下，使用 CoT 或 Zero-Shot CoT 的方式来引导大语言模型针对待解决问题生成一组多样化的推理路径; (2) 针对大语言模型生成的每个推理内容，收集其最终的答案，并统计每个答案在所有推理路径中出现的频率; (3) 选择出现频率最高的答案作为最终的、最一致的答案。

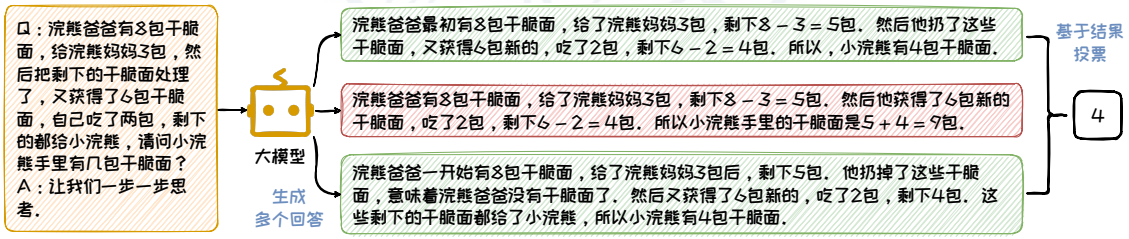


图 3.15: Self-Consistency 的流程。

本节探讨了思维链（Chain-of-Thought, CoT）的思想和模式。CoT 的核心思想是在提示中模拟人类解决问题的思考过程，在 Prompt 中嵌入解决问题的推理过程，从而在无需特定任务微调的情况下显著提升模型在推理任务上的表现。CoT 方法包含了多种模式：按部就班、按部就班以及集思广益。这些模式可以满足不同的推理需求，增强了大语言模型在复杂任务中的推理能力。

3.4 Prompt 技巧

基于上下文学习和思维链等 Prompt 工程技术，本节将进一步探讨可用于进一步提升大语言模型生成质量的 Prompt 技巧，包括合理归纳提问、适时运用思维链（CoT）以及巧妙运用心理暗示等。应用本节中介绍的 Prompt 技巧，可以引导模型生成更加精准、符合预期的内容，进一步提升大语言模型在实际应用中的表现。

3.4.1 规范 Prompt 编写

编写规范的 Prompt 是我们与大语言模型进行有效沟通的基础。经典的 Prompt 通常由**任务说明**，**上下文**，**问题**，**输出格式**等部分中的一个或几个组成。以图 3.16中这个情感分类的 Prompt 为例。在这个例子中：

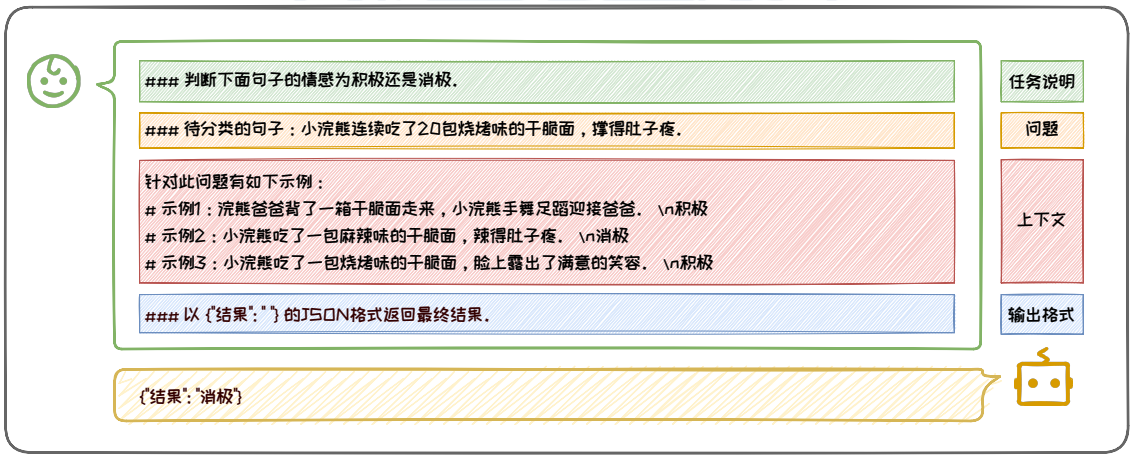


图 3.16: 经典的 Prompt 示例。

- **任务说明**是“### 判断下面句子的情感为积极还是消极。”，它明确了模型需要完成的任务；
- **上下文**是“针对此问题有如下示例：# 示例 1：浣熊爸爸背了一箱干脆面走来，小浣熊手舞足蹈迎接爸爸。\n 积极 \n# 示例 2：小浣熊吃了一包麻辣味的干脆面，辣得肚子疼。\n 消极 \n # 示例 3：小浣熊吃了一包烧烤味的干脆

面，脸上露出了满意的笑容。\n 积极 \n”。上下文提供了帮助模型理解和回答问题的示例或背景信息；

- **问题**是“待分类的句子：小浣熊连续吃了 20 包烧烤味的干脆面，撑得肚子疼。”，是用户真正想要模型解决的问题，它可以是一个段落（比如摘要总结任务中被总结的段落），也可以是一个实际的问题（比如问答任务中用户的问题），或者表格等其他类型的输入内容；
- **输出格式**是“以” 结果”: ” ” 的 JSON 格式返回最终结果。”，它规范了模型的输出格式。

通过这个例子可以看出，在编写经典 Prompt 的过程中，Prompt 各个组成部分都很重要，它们的规范性，直接影响模型的输出质量。同时，各个组成部分的排版也很重要。接下来，我们将详细介绍经典 Prompt 的规范编写需要满足的要求。

1. 任务说明要明确

明确的任务说明是构建有效 Prompt 的关键要素之一。一个清晰、具体的任务说明能够确保模型准确理解任务要求，并产生符合预期的输出。例如，在情感分类任务中，任务说明“判断下面句子的情感为积极还是消极。”就是一个明确的示例，它清晰地定义了任务类型（情感分类）和分类的具体类别（积极或消极）。相反，模糊或不明确的任务说明可能导致模型误解用户的真实意图，从而产生不符合预期的输出。如图 3.17所示，“分类下面的句子”这样的任务说明就缺乏具体性，没有明确指出分类的类型和类别，使得模型难以准确执行任务。

为了确保任务说明的明确性，我们需要明确以下几个要点：

- **使用明确的动词**：选择能够清晰表达动作的动词，如“判断”、“分类”、“生成”等，避免使用模糊的动词如“处理”或“操作”。
- **具体的名词**：使用具体的名词来定义任务的输出或目标，例如“积极”和“消极”在情感分类任务中提供了明确的分类标准。

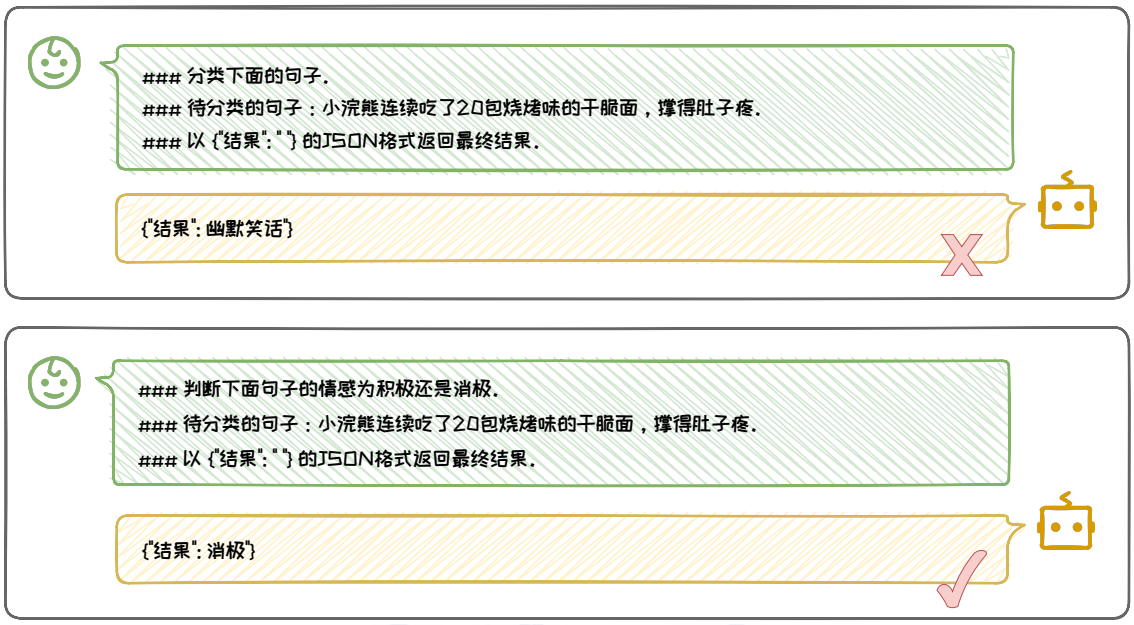


图 3.17: 不同的任务说明对比。

- **简洁明了**：任务说明应简洁且直接，避免冗长或复杂的句子结构，使模型能够快速抓住任务的核心要求。
- **结构化布局**：在较长的 Prompt 中，将任务说明放置在开头和结尾，因为模型通常更关注这些部分的信息 [18]。这种布局有助于确保模型首先和最后接触到的是最关键的任务信息。

通过这些策略，我们可以确保任务说明既清晰又具体，从而帮助模型更好地理解和执行任务，最终产生高质量的输出。

2. 上下文丰富且清晰

在 Prompt 设计中，上下文的作用不容忽视，它有时直接决定了模型能否给出正确的答案。一个**丰富且清晰**的上下文能够显著提升模型的理解和回答准确率。**上下文的丰富性**体现在其内容的多样性和相关性。上下文可以包括与问题直接相关的背景信息、具体的演示示例，或是对话的连续性内容。例如，在情感分类任务中，提供具体的示例句子及其对应的情感标签，可以帮助模型更好地理解任务的

具体要求和预期的输出。**上下文的清晰性**则要求上下文信息必须与问题紧密相关，避免包含冗余或不必要的信息。清晰的上下文应直接指向任务的核心，减少模型在处理信息时的混淆和误解。例如，在问答任务中，上下文应仅包含与问题直接相关的信息，避免引入可能误导模型的无关内容。

在图 3.18两个上下文设计的例子中，第一个例子的上下文紧密围绕问题，提供了丰富的直接相关信息，没有任何冗余内容。这种设计有助于模型迅速聚焦于关键信息，从而准确回答问题。相比之下，第二个例子的上下文不够丰富，并且单个例子则包含了大量与问题无关的细节，这些冗余信息不仅使上下文显得不明确，还可能加重模型处理信息的负担，导致模型难以准确把握问题的核心，进而影响其回答的准确性。

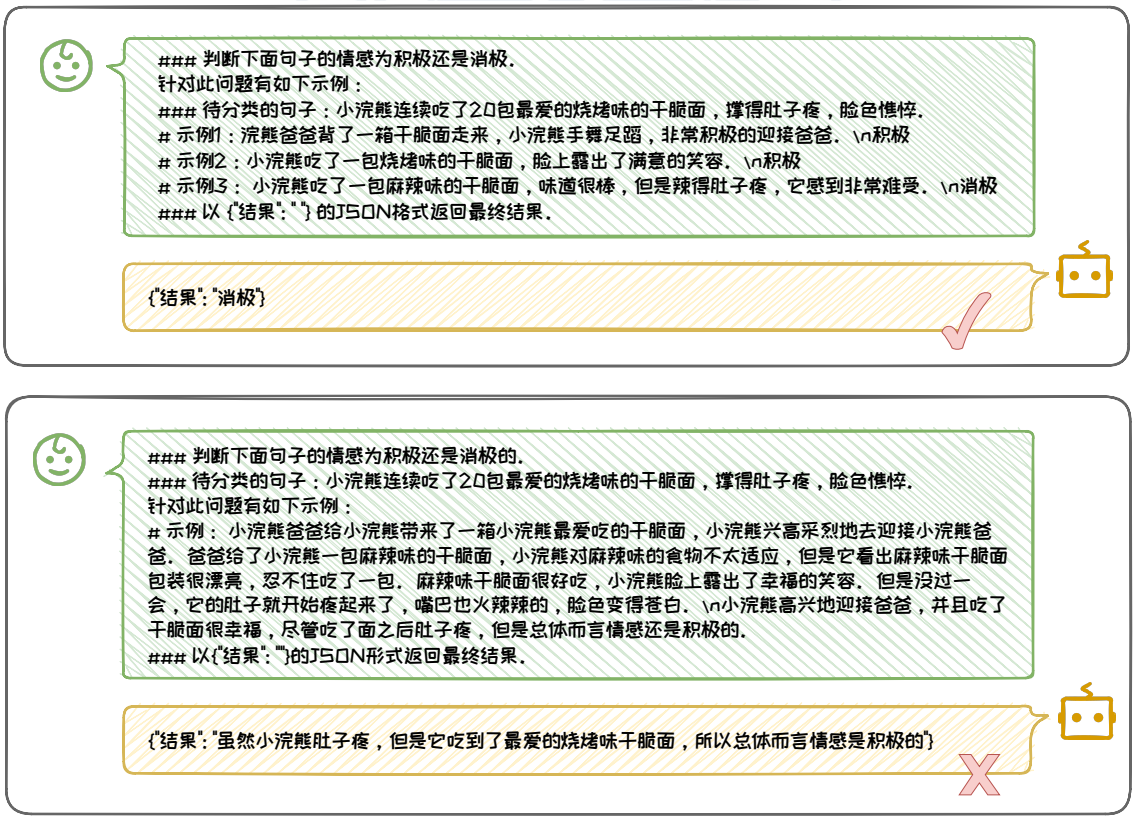


图 3.18: 不同的上下文对比。

3. 输出格式要规范

规范的输出格式对于确保模型输出的可用性和准确性至关重要。通过指定明确的输出格式，可以使模型的**输出结构化**，便于下游任务直接提取和使用生成内容。常用的输出格式包括 JSON、XML、HTML、Markdown 和 CSV 等，每种格式都有其特定的用途和优势。

例如，在图 3.19中的 Prompt 例子中，“以 {” 结果”: ” ”} ” 的 JSON 格式返回最终答案。” 明确指定了答案应以 JSON 格式输出，并且以一个简短的例子指名 JSON 中的关键字。这种规范的输出格式不仅使得结果易于解析和处理，还提高了模型输出的准确性和一致性。如果不明确规定输出格式，模型可能会输出非结构化或不规范的结果，这会增加后续处理的复杂性。在第二个例子中，如果模型输出的答案是一个自由格式的文本字符串，那么提取具体信息就需要进行复杂的字符串解析，而不是像 JSON 等结构化格式那样可以直接提取，这就给后续对于结果的处理与使用带来了麻烦。

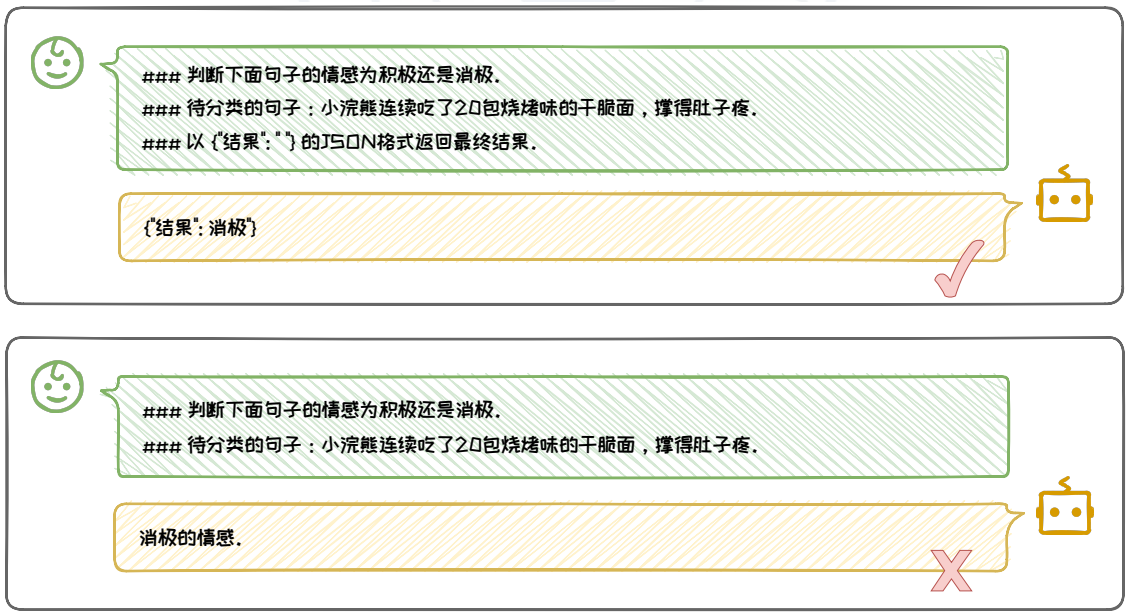


图 3.19: 不同输出格式对比。

为了确保输出格式的规范性，可以采取以下措施：

- **明确指定输出格式**：在 Prompt 中明确指出希望模型使用的输出格式，如“请以 JSON 格式返回结果”，并且选择广泛接受和易于处理的输出格式，如 JSON、CSV 等，易于解析和数据交换。
- **提供输出格式的示例**：在 Prompt 中提供一个输出格式的具体示例，比如在 JSON 中明确指出关键字，帮助模型理解预期的输出结构。

这些措施可以令模型的输出既规范又易于处理，从而提高整个系统的效率和准确性。规范的输出格式不仅简化了数据处理流程，还增强了模型输出的可靠性和一致性，为用户提供了更加流畅和高效的交互体验。

4. 排版要清晰

一个优秀的 Prompt 还必然具备清晰的排版，这对于模型的理解 Prompt 至关重要。清晰的排版有助于模型准确捕捉任务的关键信息，从而提高其执行任务的准确性和效率。相反，复杂的排版可能会导致信息模糊，使模型难以准确理解任务的具体要求 [7]，进而影响输出结果的质量。清晰的排版通常涉及**使用合适的分隔符**和**格式化技巧**，将 Prompt 的不同组成部分（如任务说明、上下文、问题和输出格式）明确区分开来。在图 3.16所示的例子中，我们使用 “#” 和 “###” 以及换行符有效地将各个部分分隔开，使得每个部分的内容清晰可见，便于模型理解和处理。相反，如果排版混乱，例如在 Prompt 中混合使用不使用任何分隔符，模型可能会混淆不同部分的内容，导致无法准确执行任务。例如，去除图 3.16中的 Prompt 例子的排版符号与样式，将图 3.20所示的复杂混乱的 Prompt 输入给大语言模型，模型回答的质量将大幅度下降。

为了确保 Prompt 的排版清晰，可以采取以下措施：

- **使用一致的分隔符**：选择并坚持使用一种或几种分隔符（如 “#”、“###”、“—” 等），以区分不同的 Prompt 部分。

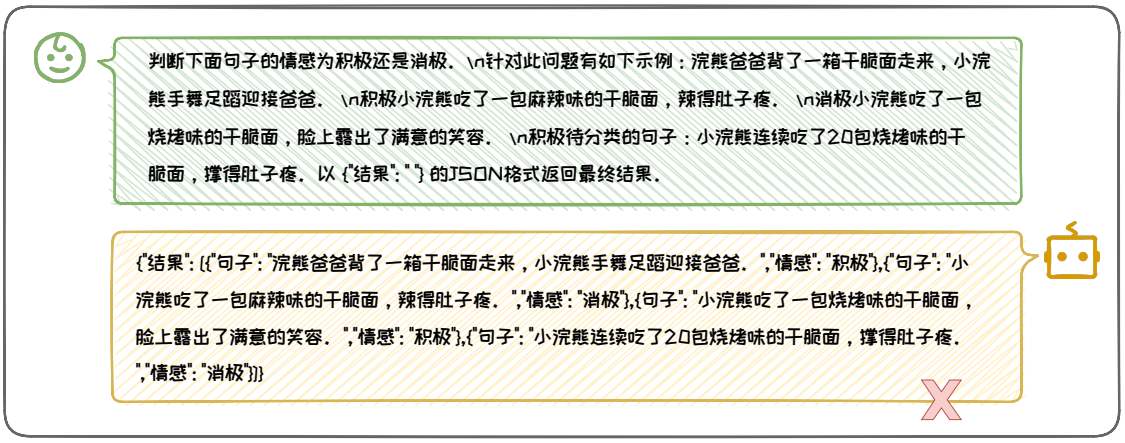


图 3.20: 不清晰的排版。

- **合理使用空白和缩进**：通过增加空白行和适当的缩进，增强 Prompt 的可读性，帮助模型区分不同的内容块。
- **清晰的标题和子标题**：为每个部分提供清晰的标题或子标题，使模型能够快速识别每个部分的主题。

通过这些措施，我们构造既清晰又易于理解的 Prompt 的排版，从而帮助模型更好地执行任务，提升信息处理的效率和准确性。

3.4.2 合理归纳提问

在与大语言模型的交互中，提问的质量直接影响到信息触达的效率和深度。一个精心设计的提问不仅能够明确表达需求，还能引导模型聚焦于问题的核心，从而获得精准且有价值的答案。本节将探索如何通过“合理归纳提问”来提升交互的质量。具体的，我们将重点介绍两个高级提问策略：“复杂问题拆解”和“追问”。这两个策略是提问艺术的精髓，它们不仅帮助我们深入理解问题的本质，还显著提高了我们获取信息和解决问题的效率。通过这些策略，我们能够更精确地引导模型，确保其输出既准确又全面，从而满足我们的需求。

1. 复杂问题拆解

在处理复杂问题时，我们可以将问题分解为**更小、更易于理解**的子问题，并**逐一解决**。在计算机算法设计中，这种策略被称为“分而治之”，其基本理念是通过逐一解决子问题，最终达成解决整个问题的目标。在与大语言模型进行交互时，同样可以借鉴“分而治之”的思想，将复杂问题分解为一系列子问题，并引导模型逐一回答，以确保每个子问题都得到充分考虑和解答。随后，将这些子问题的答案汇总，形成最终的全面回答。这一过程包括两个关键步骤：**分步引导**和**归纳总结**。

图 3.21: 复杂问题拆解例子对比。

首先，在**分步引导**阶段，我们需将复杂问题细化为多个子问题，并引导模型针对每个子问题进行深入分析和回答。这一步骤旨在确保每个子问题都能得到详尽的解答，从而为后续的归纳总结奠定坚实基础。其次，在**归纳总结**阶段，我们将各个子问题的答案进行汇总，并综合形成最终的全面回答。这一步骤不仅有助于我们全面把握问题的各个方面，还能确保最终答案的准确性和完整性。

如图 3.21所示。用户提出了一个关于成年小浣熊一天需要吃多少干脆面才能满足能量需求的问题。通过分步引导，我们将这个问题分解为两个关键的小问题："一只成年小浣熊，一天需要摄入多少热量?"和"一包干脆面的热量是多少?"。模型分别回答了这两个问题，提供了小浣熊的日热量需求和干脆面的热量含量。随后，通过归纳总结的提示"结合以上信息，成年小浣熊一天需要吃多少包干脆面才能满足能量需求?"，我们将这些分散的信息整合起来，计算出小浣熊需要摄入的干脆面数量。这一过程不仅展示了如何通过逐步提问来引导模型提供详细信息，还强调了在解决复杂问题时，系统地分解问题和整合答案的重要性。

这种方法的优势在于它能够帮助用户和模型更有效地处理复杂信息，确保每个细节都被考虑到，并最终形成一个准确和全面的答案。通过分步引导和归纳总结，我们能够有序地解决复杂问题，提供高质量的解答。

2. 追问

通过在对话中进行追问，用户可以引导大语言模型的输出更贴合心意的内容。这种对话形式的交互不仅可以促进更深层次的理解和更为丰富的讨论，而且有助于更精确地表达用户的真实想法，从而更好地指导模型的思考，使其输出更加贴合用户需求。从追问的形式和目的角度来看，追问可以分为三种形式：**深入追问**、**扩展追问**、**反馈追问**。

(1) 深入追问

深入追问的形式是指用户可以根据大语言模型的输出继续发问来深入挖掘**特定话题的深层信息**。这种追问适用于需要对某个概念、现象或过程有详尽解释的场景。通过深入追问，用户能够引导模型提供更为细致和深入的信息，从而加深对特定领域的理解。以图 3.22为例，用户首先询问了小浣熊是否可以食用干脆面。模型基于干脆面的成分给出了初步的回答，指出其对小浣熊的健康可能有害。随后，用户通过深入追问，探讨了如果去除干脆面中的调味料和添加剂，是否可以安全

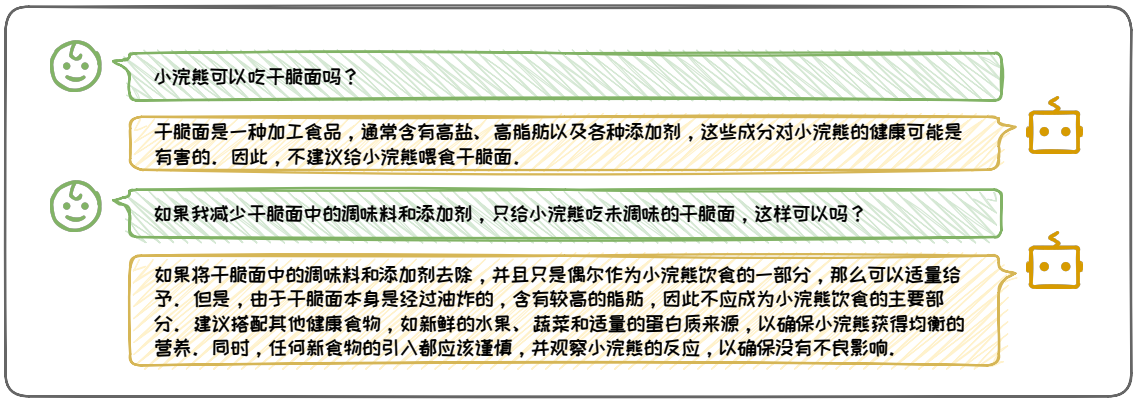


图 3.22: 深入追问示例。

地喂食小浣熊。模型进一步解释了即使去除调味料和添加剂，干脆面仍含有较高的脂肪，因此不应成为小浣熊饮食的主要部分，并建议了如何搭配其他食物以确保营养均衡。这一系列的追问展示了用户如何通过连续提问引导模型提供更为深入的信息。

(2) 扩展追问

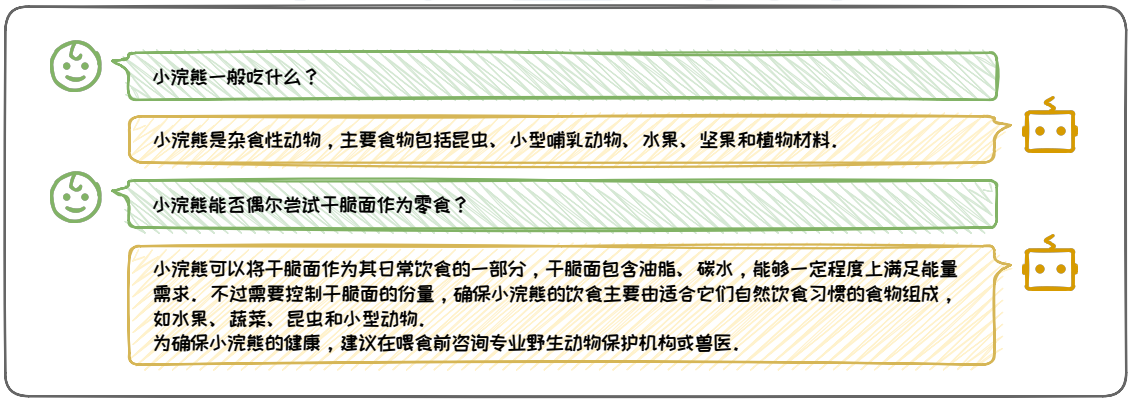


图 3.23: 扩展追问示例。

扩展追问是一种在大语言模型给出回答的基础上，进一步要求模型提供更多相关信息或例子的提问方式，其目的在于拓**宽讨论的广度**，收集更多数据、例证或选项，帮助用户获得更广泛的视角，增加对话题的理解。这种追问特别适用于需要全面了解一个主题的场景。在图 3.23这个例子中，用户首先询问了小浣熊的常规

饮食，模型提供了小浣熊作为杂食性动物的饮食概况。随后，用户通过扩展追问，探讨了小浣熊是否可以食用干脆面。模型进一步解释了干脆面可以作为小浣熊饮食的一部分，但强调了控制份量和保持饮食多样性的重要性。这一系列的追问不仅展示了用户如何通过提问获取更多关于小浣熊饮食的信息，还强调了在引入新食物时咨询专业意见的必要性。通过扩展追问，用户能够获得更全面的视角。

(3) 反馈追问

反馈追问的形式是在大语言模型的输出不符合预期或存在错误时，提供反馈，指出问题所在，并请求模型进行更正或澄清。其目的在于通过**反馈机制**提升模型的准确性，确保信息的正确性。这种追问允许用户指出模型输出中的具体错误或不足，并请求模型对其进行修正，有助于提高对话质量。在图 3.24这个例子中，用户首先询问了小浣熊连续食用 20 包干脆面后的应对措施。模型最初的回答建议小浣熊可以适量食用干脆面，但用户通过反馈追问指出了小浣熊出现了不良反应。模型随后修正了回答，指出大量食用干脆面对小浣熊的健康有害，并提供了紧急处理建议。通过反馈追问，用户能够获得更加准确可靠的信息。

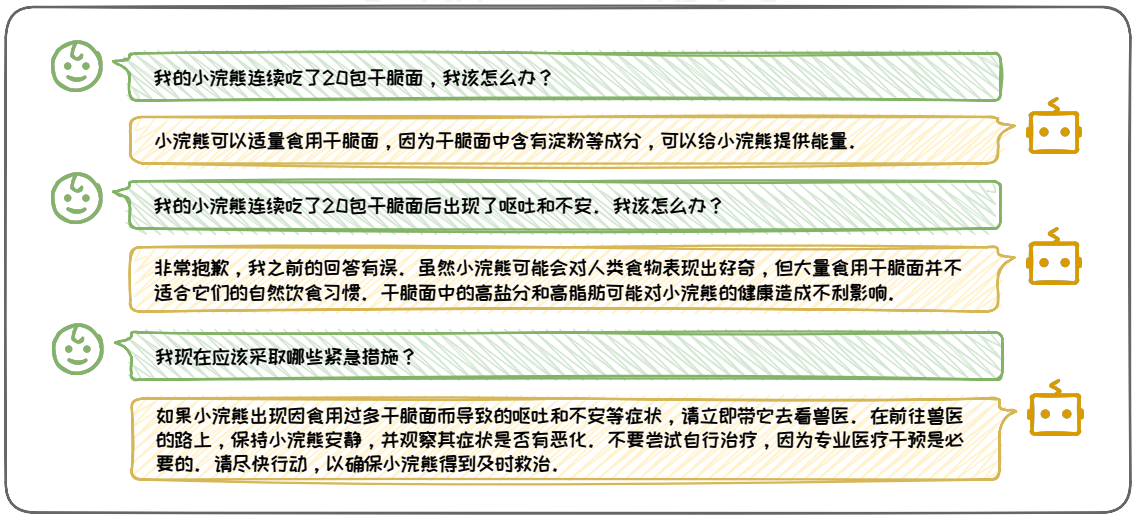


图 3.24: 反馈追问示例。

3.4.3 适时使用 CoT

思维链技术（Chain of Thought，CoT） [38] 是在处理涉及算术、常识和符号推理等复杂推理的任务时的理想选择。在处理这类任务的过程中，通过 CoT 引导模型理解和遵循中间步骤，能够显著提高得出答案的准确率。3.3 节中已经讨论过 CoT 几种经典范式。本节将讨论在何时使用 CoT 以及如何使用 CoT。

1. 何时使用 CoT

在决定何时使用 CoT 时，需要对**任务类别**、**模型规模**以及**模型能力**三方面因素进行考虑。

在**任务类别**方面，CoT 技术特别适用于需要复杂推理的任务，如算术、常识和符号推理。在这些任务上，CoT 能够引导大语言模型生成逻辑严密、条理清晰的中间推理步骤，从而提高正确答案的生成概率，如图 3.25所示。然而，对于情感分类、常识问答等简单问题，标准的 Prompt 方法已足够有效，使用 CoT 可能难以提升效果，反而可能引入不必要的复杂性。

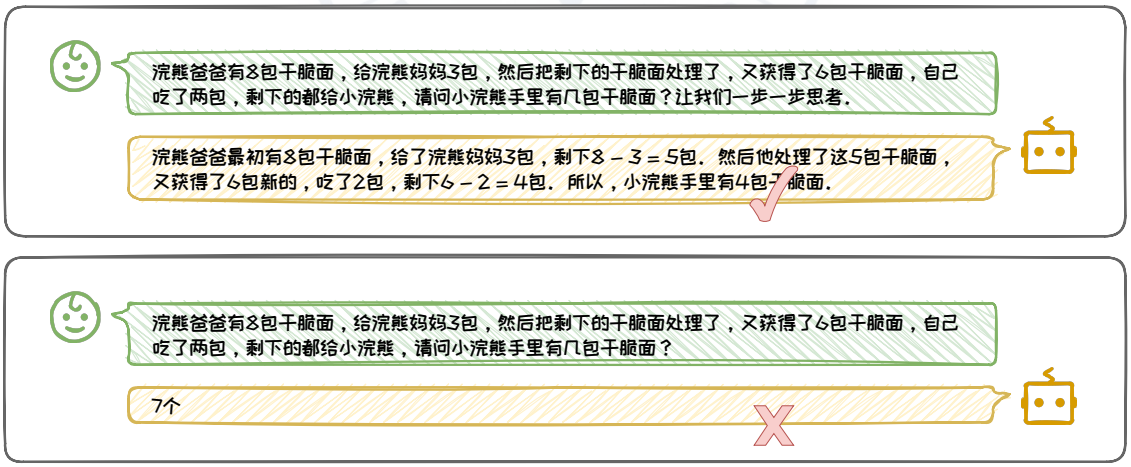


图 3.25: 在推理任务上使用 CoT。

在**模型规模**的考量上，CoT 技术应用于参数量超过千亿的巨型模型时，能够显著提升其性能，例如，PaLM [5] 模型和 GPT-3 [3] 模型等模型。然而，在规模较小

的模型上应用 CoT 技术可能会遭遇挑战，如生成逻辑不连贯的思维链，或导致最终结果的准确性不如直接的标准提示方法，如图 3.26所示。

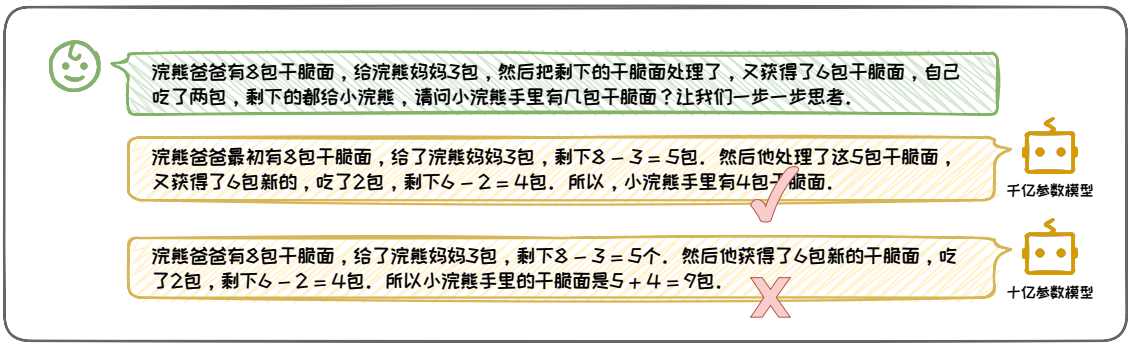


图 3.26: 不同规模模型使用 CoT 对比。

在**模型能力**的考量上，CoT 是否起效与模型在预训练阶段是否进行过推理方面的指令微调有关。对于那些**未经推理方面的指令微调的大语言模型**，如早期的 GPT-3、Palm 以及目前开源的基础版本模型，例如 LLaMA2-13B-Base 和 Baichuan2-13B-Base，适当的 CoT 提示能够激发其卓越的 CoT 推理能力；**对于已经历过推理方面的指令微调的大语言模型**，如 ChatGPT、GPT-4 以及 LLaMA2-13B-Chat 等，即便在没有 CoT 指令的情况下，也能自发生成条理清晰的中间推理步骤。在许多情况下，这些模型在没有 CoT 指令的条件下反而展现出更佳的性能，这表明它们在指令微调过程中可能已经内化了 CoT 指令，使得即便在没有明确 CoT 提示时，仍能隐式遵循 CoT 推理路径。

2. 灵活使用 CoT

灵活使用 CoT 的关键在于根据任务的具体需求和模型的特性来调整 CoT 的使用方式。主要涉及调整 CoT 的详细程度以及使用不同的 CoT 形式两个方面。

- **调整 CoT 的详细程度**：我们可以指定 CoT 输出的详细程度，以适应不同的用户需求，如图 3.27所示。对于简单的计算问题，在用户不需要推理的中间过程时，我们可以直接给出最终乘法和加法的结果。而对于复杂的计算、推理

问题，或者用户需要理解中间推理过程时，我们需要通过样例进行引导，以使其展示完整的推理步骤。

- **使用不同的 CoT 形式**：我们可以根据不同任务场景，选择不同的 CoT 形式。在不需要特定领域知识，仅需对问题进行逻辑推理和逐步分析时，可以使用 Zero-Shot CoT 或者 Auto CoT 的方式，通过“让我们一步一步思考”这种 CoT 提示触发词，来引导模型以 CoT 的形式回答内容。在处理需要高准确度和可靠性的任务时，可要求模型生成多个回答并提出最终结果，进而运用 Self-Consistency 方法筛选出一致性最强的答案。例如，在编写代码时，模型可生成多个版本，通过 Self-Consistency 方法确保最终选定的代码在逻辑上最为一致。对于包含创意思维的任务，可选用 ToT 和 GoT 方法来在多个可能的思维路径中进行探索和选择。例如，在创作故事时，模型可利用 ToT 或 GoT 探索不同的情节发展路径，并选取最富有趣味性或合理性的发展方向。

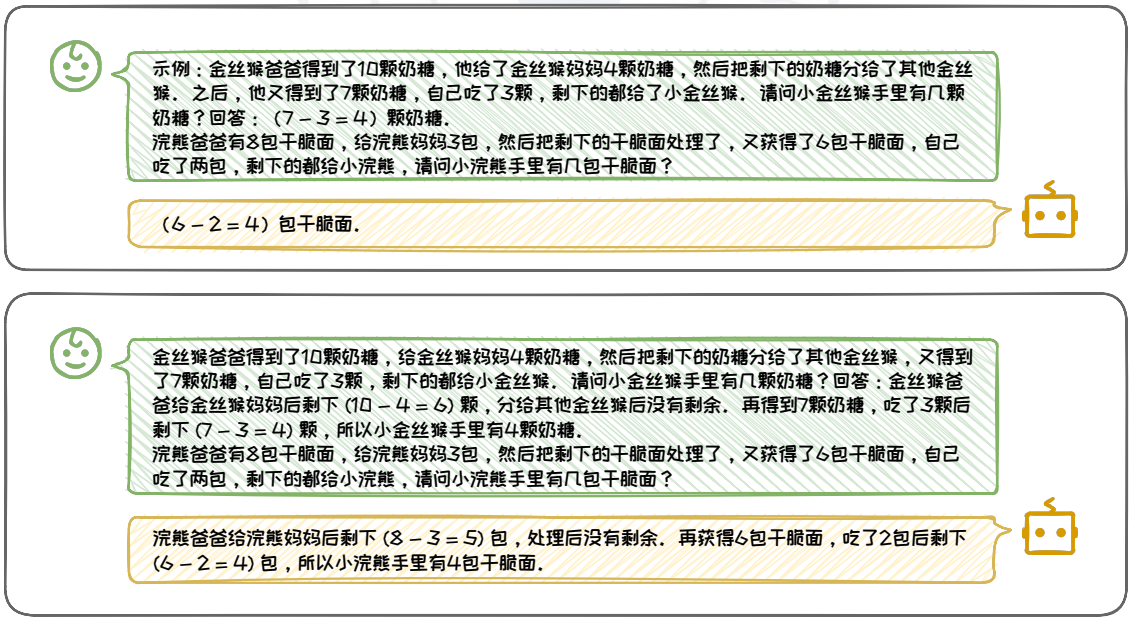


图 3.27: 通过 Few-Shot 示例指定 CoT 输出的形式和风格。

3.4.4 善用心理暗示

在硅谷，流传着一句创业金句，**“Fake it till you make it”（假装它直到你成功）**。这句话具体含义为，先吹嘘你的想法，进而吸引资本和人才，最终在实践中努力追赶并实现既定目标。这句话源自一种积极的心理暗示方法：**通过模仿自信和乐观的心态，一个人可以在他们的现实生活中实现这些品质**。这种现象不仅局限于人类的行为，积极的心理暗示也可用于激发大语言模型的潜力。这种心理暗示可以通过角色扮演和情景代入的方式传达给大语言模型。

1. 角色扮演

通过 Prompt 指导大语言模型扮演特定角色能够显著改善其与角色相关的技能。这种技术被称为角色扮演（Role-Playing），它可使大语言模型能够生成更为准确、角色相关的内容。通过为大语言模型设定一个详尽的角色，例如数据科学家、诗人或律师，可以有效地引导模型的输出朝向所需的方向，从而提供更高质量的回答。为了构建一个有效的角色，需要在指令中**包含具体属性、职责、知识和技能**。在设计角色设定提示时，选择对特定任务具有明显优势的角色至关重要。通过

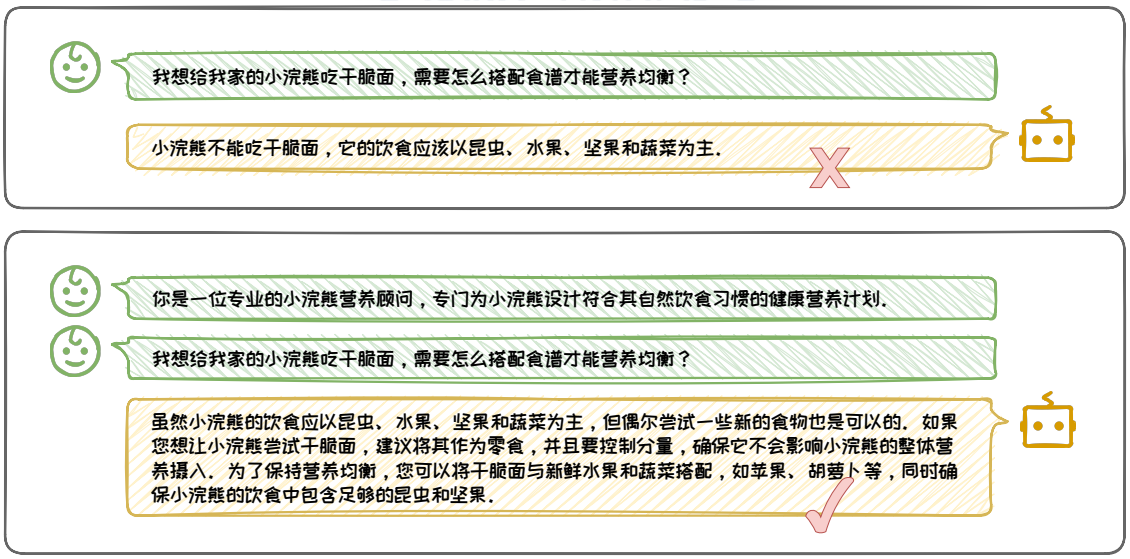


图 3.28: 角色扮演示例。

额外的描述来强调这一优势，通常可以带来更好的结果 [13]。

如图 3.28所示，通过角色扮演，模型承担了“专业的小浣熊营养顾问”这一角色，展现出了对小浣熊饮食需求的专业知识。在回答用户关于小浣熊尝试干脆面的询问时，模型不仅坚持了小浣熊应以昆虫、水果、坚果和蔬菜为主的自然饮食原则，还巧妙地提出了将干脆面作为零食的建议，并强调了控制分量的重要性。此外，模型还建议将干脆面与新鲜水果和蔬菜搭配，确保小浣熊的饮食中包含足够的昆虫和坚果，以维持营养均衡。这一建议既科学又实用，充分展现了角色扮演在提升模型交互质量和适应用户需求方面的巨大潜力。

2. 情景代入

在日常生活中，一个人容易受到周围环境和交往人群的影响，可能会逐渐采纳周围人的行为和思维方式。这种情景代入现象不仅适用于人，也同样适用于大语言模型。通过将模型置于特定的“情景”或“环境”中，可以影响其生成的文本内容和风格。情景代入指的是将特定情境下所需的**专业知识**、**历史背景**等信息嵌入到模型的响应中。

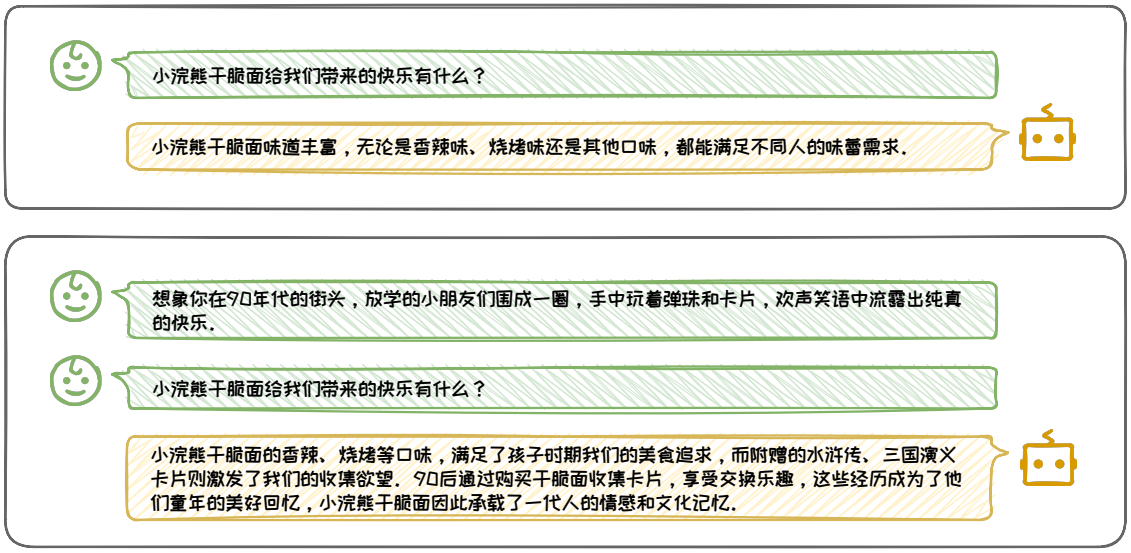


图 3.29: 情景代入示例。

通过情景代入，模型能够深入理解并反映特定情境下的文化社会背景与现实环境，从而生成更加丰富和有深度的回答。在图 3.29例子中，当模型被置于 90 年代的街头情景中，它不仅能够描述小浣熊干脆面的口味，还能够捕捉到那个时代特有的文化现象——通过收集卡片来交换乐趣。这种回答不仅提供了具体的信息，还唤起了用户的情感共鸣，增强了交互的情感连接。

在本节中，我们深入探讨了提升 Prompt 技巧的多种策略，以增强大语言模型的交互效率和输出质量。这些技巧主要包括规范 Prompt 编写、合理归纳提问、适时使用思维链、以及善用心理暗示。这些技巧和策略的应用，不仅可以提升了提示的有效性，使得模型能够更准确地理解和回应用户的需求，还显著提高了大语言模型在复杂任务中的表现。

3.5 相关应用

Prompt 工程的应用极为广泛，几乎涵盖了所有需要与大语言模型进行高效交互的场景。这项技术不仅能够帮助我们处理一些基础任务，还能显著提升大语言模型在应对复杂任务时的表现。Prompt 工程在构建 Agent 完成复杂任务、进行数据合成、Text-to-SQL 转换，以及设计个性化的 GPTs 等方面，发挥着不可或缺的作用。下面我们将依次介绍 Prompt 工程在这些应用场景中的具体作用。

3.5.1 基于大语言模型的 Agent

智能体（Agent）是一种能够自主感知环境并采取行动以实现特定目标的实体 [35]。作为实现通用人工智能（AGI）的有力手段，Agent 被期望能够完成各种复杂任务，并在多样化环境中表现出类人智能。然而，以往的 Agent 通常依赖简单的启发式策略函数，在孤立且受限的环境中进行学习和操作，这种方法难以复制人类水平的决策过程，限制了 Agent 的能力和应用范围。近年来，大语言模型的不断

发展，涌现出各种能力，为 Agent 研究带来了新的机遇。基于大语言模型的 Agent（**以下统称 Agent**）展现出了强大的决策能力。其具备全面的通用知识，可以在缺乏、训练数据的情况下，也能进行规划、决策、工具调用等复杂的行动。

Prompt 工程技术在 Agent 中起到了重要的作用。在 Agent 系统中，大语言模型作为核心控制器，能够完成规划、决策、行动等操作，这些操作很多都依赖 Prompt 完成。图 3.30展示了一个经典的 Agent 框架，该框架主要由四大部分组成：**配置模块（Profile）**、**记忆模块（Memory）**、**计划模块（Planning）**和**行动模块（Action）**[35]。Prompt 工程技术贯穿整个 Agent 流程，为每个模块提供支持。

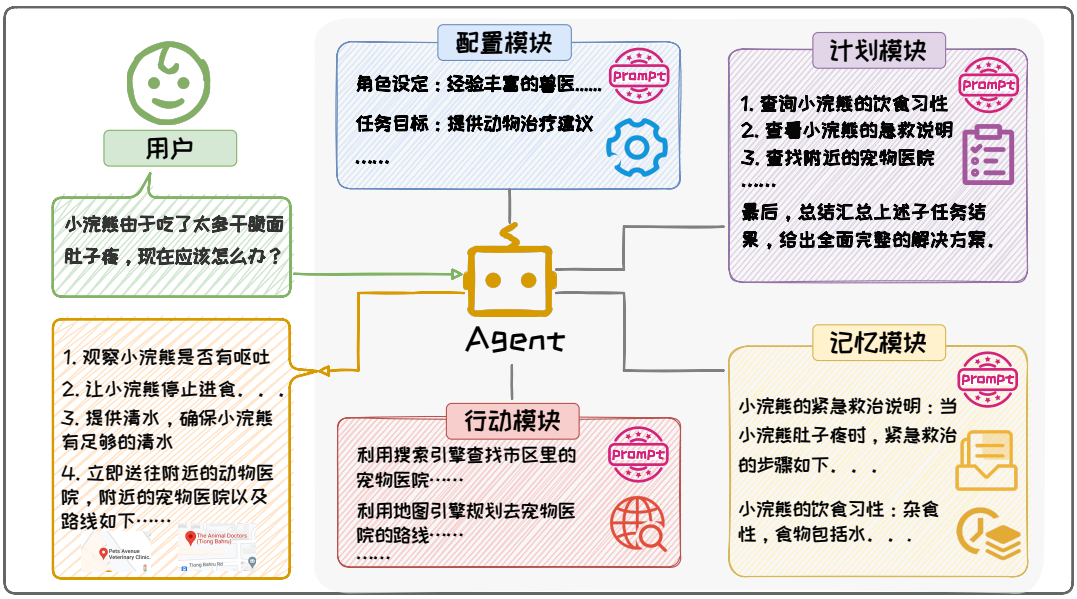


图 3.30: 基于大语言模型的 Agent 框架流程示意图。

在 Agent 中，上述四个组件各司其职，分工协作，共同完成复杂任务：（1）**配置模块**利用 Prompt 工程中的角色扮演技术，来定义 Agent 的角色。设定 Agent 的背景、技能、职责等信息，这些角色设定信息以上下文的形式嵌入到 Agent 每一次交互的 Prompt 中。（2）**记忆模块**是 Agent 的知识与交互记忆的存储中心。记忆模块通过检索增强等技术获取记忆，这一过程涉及到使用 Prompt 工程中的上下文学

习技术来构造和优化查询，从而帮助更加精准检索到相关记忆。在获取记忆之后，将这些记忆将被添加到交互的 Prompt 中，帮助 Agent 利用这些记忆知识，实现更为准确高效的决策与行动。(3) **计划模块**则扮演着任务分解者的角色，它将复杂的任务细化为一系列更为简单、易于管理的子任务。在这一过程中，通过 Prompt 工程中的思维链技术，让大语言模型分解任务并进行规划，按照链式顺序输出子任务；同时还利用了上下文学习技术，构造少样本示例来调控分解出的子任务的粒度，确保整个任务流程的顺畅与高效。(4) **行动模块**负责将计划模块生成的计划转化为具体的行动步骤，并借助外部工具执行这些步骤以实现 Agent 的目标。通常会为 Agent 提供工具 API 的接口，把调用 API 接口的示例作为上下文，让大语言模型生成调用 API 的代码，之后执行这些代码，从而得到执行步骤的结果。

如图 3.30展示了一个小浣熊健康助理 Agent 处理用户请求 “小浣熊由于吃了太多干脆面肚子疼，现在应该怎么办?” 的流程。在这个例子中，首先，**配置模块**为 Agent 设定了明确的角色定位——经验丰富的兽医，并明确了任务目标为提供专业的动物治疗建议。为确保角色设定与任务目标的一致性，相关信息始终以上下文形式嵌入至输入给大语言模型的 Prompt 中，从而为后续处理奠定坚实基础。其次，**计划模块**根据用户的具体请求，精心规划了救助小浣熊的行动方案。该模块将整体任务细化为多个子任务，包括搜寻小浣熊的相关资料及救助信息、查找附近的宠物医院以及总结各个子任务的执行结果以给出全面回答等。接着，**记忆模块**从知识库中搜寻小浣熊的相关信息，包括小浣熊饮食习性、如何急救小浣熊等，提供了必要的背景知识支持。然后，**行动模块**通过调用搜索工具，迅速搜索到附近最近的宠物医院，并调用地图工具规划出最佳的送医路线。最后，**计划模块**将多个子任务的执行结果进行汇总，并综合考虑各种因素，执行最佳行动方案。该方案不仅包括观察小浣熊的行为、提供专业的照顾建议，还详细阐述了如何为小浣熊紧急送医的步骤，以生动形象的方式向用户展示整个救助过程。

基于大语言模型的 Agent，在不同行业和应用场景都展现出了巨大的潜力。斯坦福大学利用 GPT-4 模拟了一个虚拟的西部小镇 [25]。他们创建了一个虚拟环境，让多个基于 GPT-4 的 Agent 在其中生活和互动。这些 Agent 通过 Prompt 工程里面的角色扮演技术设定了不同的角色，如医生、教师和市长等，并根据自己的角色和目标自主行动，进行交流，解决问题，并推动小镇的发展。HuggingGPT [30] 则以 ChatGPT 为核心控制器，用户给定一个任务后，它首先将任务拆分为多个子任务，并从 Huggingface 上调用不同的模型来解决这些子任务。在得到子任务的结果之后，HuggingGPT 将这些结果汇总起来，返回最终结果，展现了其在复杂任务编排规划和模型调度协同方面的强大能力。这些 Agent 的研究不仅推动了大语言模型的发展与应用落地，也为 Agent 在现实世界中的应用提供了新的视角和方法。

3.5.2 数据合成

数据质量是制约大语言模型性能上限的关键要素之一，正所谓“Garbage in, Garbage Out” [27]，无论模型架构多么优秀，训练算法多么优秀，计算资源多么强大，最终模型的表现都高度依赖于训练数据的质量。然而，获取高质量数据资源面临挑战。研究显示，**公共领域**的高质量语言数据，如书籍、新闻、科学论文和维基百科，预计将在 2026 年左右耗尽 [33]。**特定领域**的垂直数据因隐私保护和标注难度高等问题，难以大量提供高质量数据，限制了模型的进一步发展。

面对这些挑战，数据合成作为一种补充或替代真实数据的有效手段，因其可控性、安全性和低成本等优势而受到广泛关注。特别是利用大语言模型生成训练数据，已成为当前研究的热点议题，其通过 Prompt 工程技术，利用大语言模型强大的思维能力、指令跟随能力，来合成高质量数据，其中一个代表性方法是 Self-Instruct [37]。Self-Instruct 通过 Prompt 工程技术构建 Prompt，通过多步骤调用大语言模型，并依据已有的少量指令数据，合成大量丰富且多样化的指令数据。以金融

场景为例，我们可以先人工标注少量金融指令数据（如数百条），例如“请根据提供的几只基金情况，为我挑选出最佳基金”。随后，我们运用 Self-Instruct 方法调用大语言模型，我们能够将这些数据扩展至数万条，且保持高质量和多样性，如生成“请指导我如何选择股票和基金进行投资”等指令。

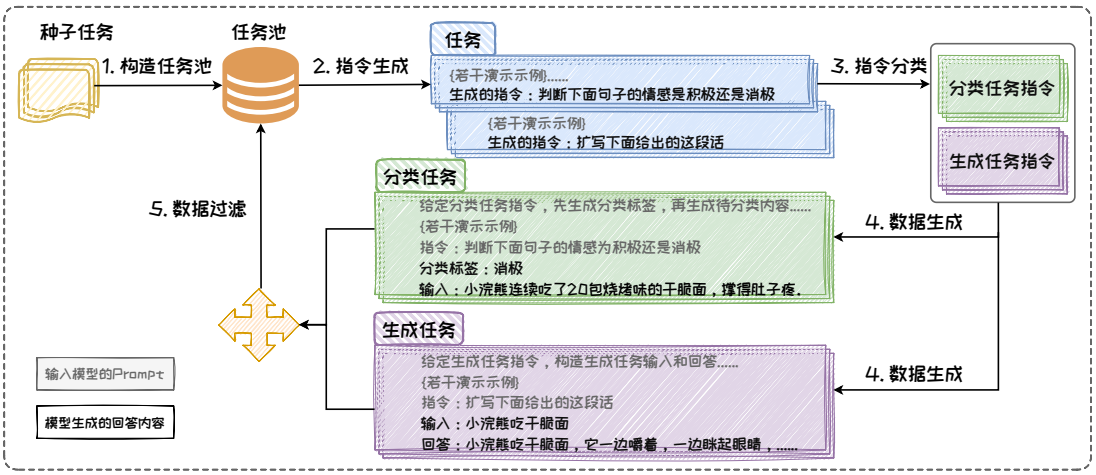


图 3.31: Self-Instruct 流程示例图。

如图 3.31所示，Self-Instruct 包含**构建任务池**、**指令生成**、**指令分类**、**数据生成**、**数据过滤**五个步骤。任务池存储了初始的指令数据以及存储后续生成的指令数据；指令生成负责参考任务池中的样例，生成指令数据中的**指令部分**；指令分类将生成的指令分类为**分类任务**或**生成任务**，在这两种任务模式下，生成数据的方式不同；数据生成则根据已有的指令，生成指令数据中的**输入部分**和**回答部分**；数据过滤则复杂去除低质量的数据，保证生成的指令数据质量。它从一个有限的手动编写任务种子集开始，通过与大语言模型交互，不断生成指令数据，扩充原始的数据集。

- **1. 构建任务池**。人工手工设计了 175 个指令数据集，作为初始任务池，后续模型会不断参考任务池的示例，来生成指令数据，并将生成的指令数据加入任务池中。

- **2. 指令生成**。从任务池中随机抽取 8 个现有指令作为演示示例组成 Prompt 中的上下文，以少样本学习的方式构造 Prompt 让模型生成指令。图中生成了两个指令："扩写下面给出的这段话"和"判断下面句子的情感为积极还是消极"。
- **3. 指令分类**。编写若干条"指令-分类任务/生成任务"样例对，作为上下文构造 Prompt，让模型判断该指令对应的任务是分类任务还是生成任务。图中"扩写下面给出的这段话"和"判断下面句子的情感为积极还是消极"分别被分类为生成任务和分类任务。
- **4. 数据生成**。对分类任务和生成任务使用 Prompt 工程中的上下文学习技术，构造不同的 Prompt 来生成指令数据中的输入部分和回答部分。对于"扩写下面给出的这段话"这条指令，它是分类任务，在 Prompt 中指示模型先生成类别标签，再生成相应的输入内容，从而使得生成的输入内容更加偏向于该类别标签；对于"判断下面句子的情感为积极还是消极"这条指令，它是生成任务，在 Prompt 中指示模型先生成输入内容，再生成对应的回答。
- **5. 数据过滤**。通过设置各种启发式方法过滤低质量或者高重复度的指令数据，然后将剩余的有效任务添加到任务池中。

上述过程中，第二步到第五步不断循环迭代，直到任务池中收集到足够的数据时停止。

数据合成的意义在于，它不仅能够缓解高质量数据资源的枯竭问题，还能通过生成多样化的数据集，提高模型的泛化能力和鲁棒性。此外，数据合成还能在保护隐私的前提下，为特定领域的垂直数据提供有效的补充。并且，通过利用大型语言模型进行数据合成，生成的数据可以用于微调小型模型，显著提升其效果，实现模型蒸馏。

3.5.3 Text-to-SQL

互联网技术进步带动数据量指数增长，目前金融、电商等各行业的海量高价值数据主要存储在关系型数据库中。从关系型数据库中查询数据需要使用结构化查询语言（Structured Query Language，SQL）进行编程。然而，SQL 逻辑复杂复杂，编程难度较高，只有专业人员才能够熟练掌握，这为非专业人士从关系型数据库中查询数据设置了障碍。为了降低数据查询门槛，零代码或低代码来的数据查询接口亟待研究。Text-to-SQL 技术可以将**自然语言查询**翻译成可以在数据库中执行的 **SQL 语句**，是实现零代码或低代码数据查询的有效途径。通过 Text-to-SQL 的方式，我们只需要动动嘴，用大白话就能对数据库进行查询，而不必自己编写 SQL 语句。这让广大普通人都可以自由操作数据库，挖掘数据库中隐含的数据价值。

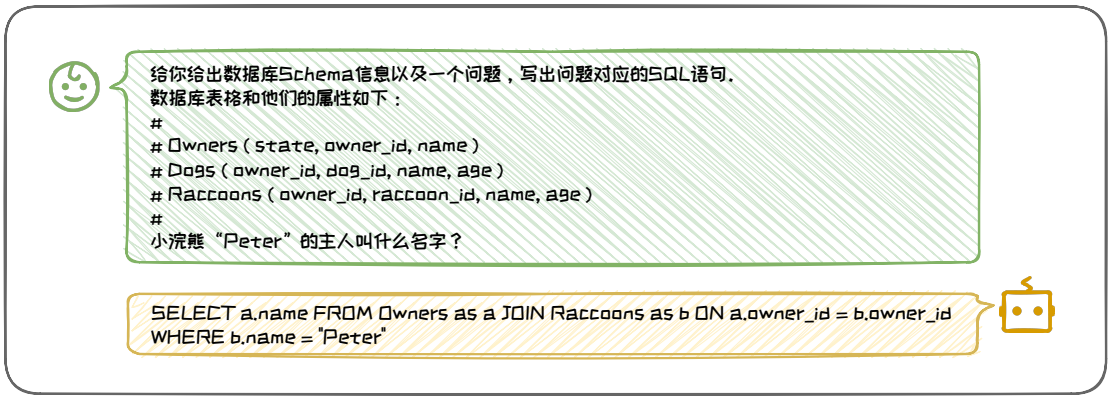


图 3.32: Text-to-SQL 示例。

传统的 Text-to-SQL 方法通常使用预训练-微调的范式训练 Text-to-SQL 模型。这些方法需要大量训练数据，费时费力且难以泛化到新的场景。近年，大语言模型涌现出的代码生成能力，为零样本 Text-to-SQL 带来可能。图 3.32 展示了一个应用大语言模型进行零样本 Text-to-SQL 的例子。用户把问题输入给大语言模型，询问“小浣熊“Peter”的主人叫什么名字?”。大语言模型会根据用户的问题，生成对应的 SQL 语句，即”SELECT a.name FROM Owners as a JOIN Raccoons as b ON

a.owner_id = b.owner_id WHERE b.name = 'Peter';"。随后，SQL 语句可以在对应的数据库中执行，得到用户提问的答案，并返回给用户。

最早使用大语言模型来做零样本 Text-to-SQL 的方法是 C3 [7] 。C3 的核心在于 Prompt 工程的设计，给出了如何针对 Text-to-SQL 任务设计 Prompt 来优化生成效果。如图3.33所示，C3 由三个关键部分组成：**清晰提示**（Clear Prompting）、**提示校准**（Calibration with Hints）**和一致输出**（Consistent Output），分别对应模型输入、模型偏差和模型输出。

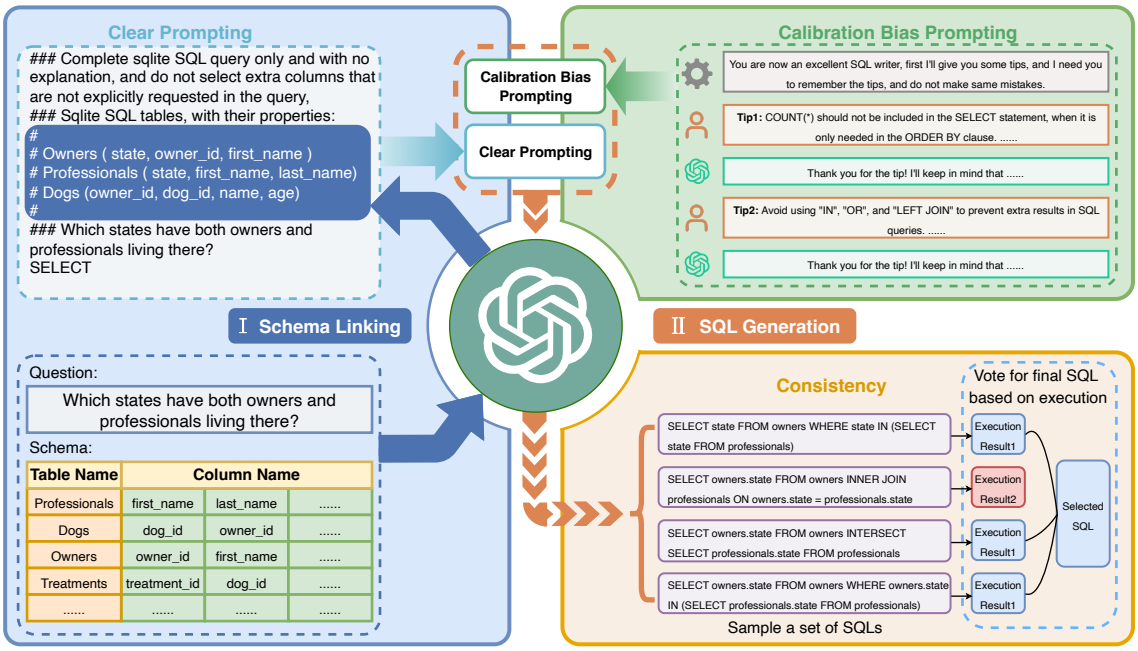


图 3.33: C3 方法整体框架图。

在模型输入端，C3 提出要采用 **清晰提示（Clear Prompting）**。其包含两部分：**（1）清晰布局（Clear Layout）**，通过明确符号划分指令、上下文和问题，确保指令模板清晰，显著提升 ChatGPT 对问题的理解能力。这一策略体现了 Prompt 技巧中的“排版清晰”原则，确保信息有效传递。**（2）清晰上下文（Clear Context）**，设计零样本 Prompt，指示 ChatGPT 从数据库中召回与问题相关的表和列。此举旨在检索关键信息，去除无关内容，减少上下文长度和冗余信息，从而提高生成 SQL

的准确性。这体现了 Prompt 技巧中“上下文丰富且清晰”的原则，确保了模型在处理任务时能够聚焦于关键信息。

为应对 ChatGPT 本身的固有偏差，C3 中采用**提示校准（Calibration with Hints）**。通过插件式校准策略，利用包含历史对话的上下文提示，将先验知识纳入 ChatGPT。在历史对话中，设定 ChatGPT 为优秀 SQL 专家角色，通过对话引导其遵循预设提示，通过这种角色扮演的方式，有效校准偏差。

在模型的输出端，C3 采用**输出校准（Output Calibration）**来应对大语言模型固有的随机性。C3 将 Self-Consistency 方法应用到 Text-to-SQL 任务上，对多种推理路径进行采样，选择最一致的答案，增强输出稳定性，保持 SQL 查询的一致性。

3.5.4 GPTS

GPTs 是 OpenAI 推出的支持用户自定义的 GPT 应用，允许用户通过编写 Prompt，添加工具等方式创建定制版的 GPT 应用，也可以使用别人分享的 GPTs 模型。

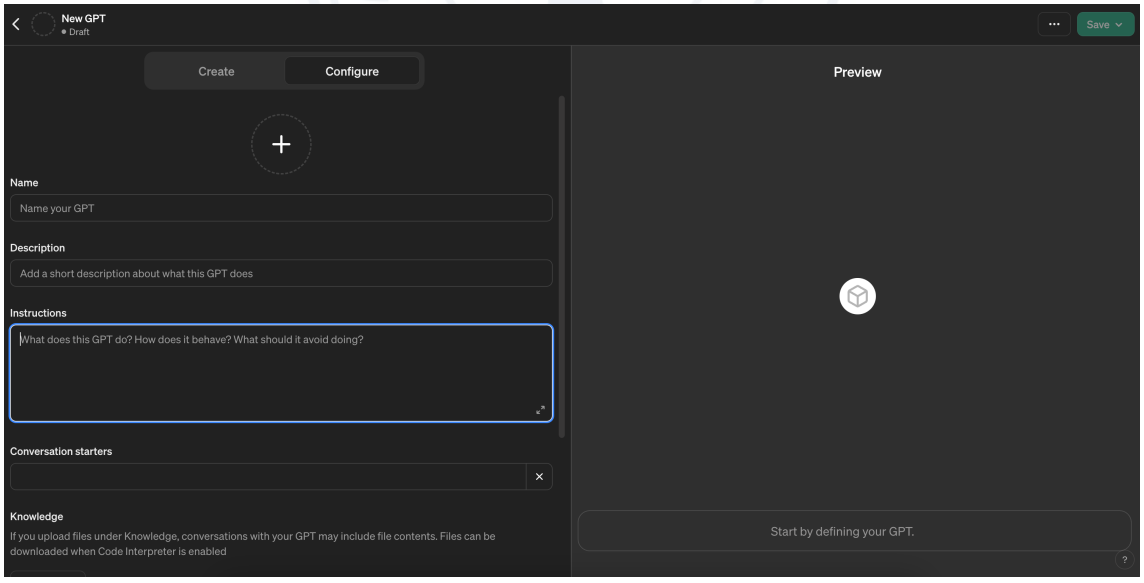


图 3.34: 制作 GPTS 的页面图 [6]。

图 3.34展示了如何制作 GPTs。在这个专属页面中，用户拥有充分的自由度来自定义 GPTs 的能力和功能。通过 “Description” 指明该 GPTs 的功能，方便其他用户快速理解其功能和作用；而 “Instructions” 则是为 GPTs 预设 Prompt，使得 GPTs 能够实现预期的功能。运用本章所介绍的 Prompt 工程技术编写这些关键内容，可以显著提高 GPTs 的性能。此外，用户还可以定制专属的知识库，并根据需求选择所需的能力，例如网络搜索、图像生成、代码解释等。完成个性化 GPTs 的制作后，用户可以选择将其分享，供其他用户使用。

在本节中，我们详细探讨了 Prompt 工程在多个领域中的具体应用场景应用场景。Prompt 工程在提升大语言模型的交互和执行能力方面具有重要作用。无论是在任务规划、数据合成、个性化模型定制，还是 Text-to-SQL 中，Prompt 工程都展现了其独特的优势和广阔的应用前景。

参考文献

[1] Maciej Besta et al. “Graph of Thoughts: Solving Elaborate Problems with Large Language Models”. In: *AAAI*. 2024.

[2] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

[3] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *NeurIPS*. 2020.

[4] Stephanie C. Y. Chan et al. “Data Distributional Properties Drive Emergent In-Context Learning in Transformers”. In: *NeurIPS*. 2022.

[5] Aakanksha Chowdhery et al. “PaLM: Scaling Language Modeling with Pathways”. In: *Journal of Machine Learning Research* 24 (2023), 240:1–240:113.

[6] DeepSeek-AI et al. “DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model”. In: *arXiv preprint arXiv:2405.04434* (2024).

[7] Xuemei Dong et al. “C3: Zero-shot Text-to-SQL with ChatGPT”. In: *arXiv preprint arXiv:2307.07306* (2023).

[8] Philip Gage. “A new algorithm for data compression”. In: *The C User’s Journal* 12.2 (1994), pp. 23–38.

[9] Dan Hendrycks et al. “Measuring Massive Multitask Language Understanding”. In: *ICLR*. 2021.

[10] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[11] Huiqiang Jiang et al. “LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models”. In: *EMNLP*. 2023.

[12] Takeshi Kojima et al. “Large Language Models are Zero-Shot Reasoners”. In: *NeurIPS*. 2022.

[13] Aobo Kong et al. “Better Zero-Shot Reasoning with Role-Play Prompting”. In: *arXiv preprint arXiv:2308.07702* (2023).

[14] Jannik Kossen, Yarin Gal, and Tom Rainforth. “In-Context Learning Learns Label Relationships but Is Not Conventional Learning”. In: *arXiv preprint arXiv:2307.12375* (2024).

[15] Junlong Li et al. “Self-Prompting Large Language Models for Zero-Shot Open-Domain QA”. In: *arXiv preprint arXiv:2212.08635* (2024).

[16] Xiaonan Li et al. “Unified Demonstration Retriever for In-Context Learning”. In: *ACL*. 2023.

[17] Jiachang Liu et al. “What Makes Good In-Context Examples for GPT-3?” In: *ACL*. 2022, pp. 100–114.

[18] Nelson F Liu et al. “Lost in the middle: How language models use long contexts”. In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 157–173.

[19] Yao Lu et al. “Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity”. In: *ACL*. 2022.

[20] Man Luo et al. “In-context Learning with Retrieved Demonstrations for Language Models: A Survey”. In: *arXiv preprint arXiv:2401.11624* (2024).

[21] Ziyang Luo et al. “Wizardcoder: Empowering code large language models with evol-instruct”. In: *arXiv preprint arXiv:2306.08568* (2023).

[22] Yuren Mao et al. “FIT-RAG: Black-Box RAG with Factual Information and Token Reduction”. In: *arXiv preprint arXiv:2403.14374* (2024).

[23] Sewon Min et al. “Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?” In: *EMNLP*. 2022.

[24] Jane Pan et al. “What In-Context Learning ”Learns” In-Context: Disentangling Task Recognition and Task Learning”. In: *ACL*. 2023.

[25] Joon Sung Park et al. “Generative Agents: Interactive Simulacra of Human Behavior”. In: *UIST*. 2023.

[26] Allan Raventós et al. “Pretraining task diversity and the emergence of non-Bayesian in-context learning for regression”. In: *NeurIPS*. 2023.

[27] L. Todd Rose and Kurt W. Fischer. “Garbage in, garbage out: Having useful data is everything”. In: *Measurement: Interdisciplinary Research and Perspectives* 9.4 (2011), pp. 224–226.

[28] Alexander Scarlatos and Andrew Lan. “RetICL: Sequential Retrieval of In-Context Examples with Reinforcement Learning”. In: *arXiv preprint arXiv:2305.14502* (2024).

[29] Mike Schuster and Kaisuke Nakajima. “Japanese and korean voice search”. In: *ICASSP*. 2012.

[30] Yongliang Shen et al. “HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face”. In: *NeruIPS*. 2023.

[31] Seongjin Shin et al. “On the Effect of Pretraining Corpora on In-context Learning by a Large-scale Language Model”. In: *NAACL*. 2022.

[32] Rohan Taori et al. “Alpaca: A strong, replicable instruction-following model”. In: *Stanford Center for Research on Foundation Models.* (2023).

[33] Pablo Villalobos et al. “Will we run out of data? An analysis of the limits of scaling datasets in Machine Learning”. In: *arXiv preprint arXiv:2211.04325* (2022).

[34] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. “Neural machine translation with byte-level subwords”. In: *AAAI*. 2020.

[35] Lei Wang et al. “A survey on large language model based autonomous agents”. In: *Frontiers of Computer Science* 18.6 (2024), p. 186345.

[36] Xuezhi Wang et al. “Self-Consistency Improves Chain of Thought Reasoning in Language Models”. In: *ICLR*. 2023.

[37] Yizhong Wang et al. “Self-Instruct: Aligning Language Models with Self-Generated Instructions”. In: *ACL*. 2023.

[38] Jason Wei et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *NeurIPS*. 2022.

[39] Jason Wei et al. “Emergent Abilities of Large Language Models”. In: *Transaction of Machine Learning Research* 2022 (2022).

[40] Sang Michael Xie et al. “An Explanation of In-context Learning as Implicit Bayesian Inference”. In: *ICLR*. 2022.

[41] An Yang et al. “Qwen2 Technical Report”. In: *arXiv preprint arXiv:2407.10671* (2024).

[42] Shunyu Yao et al. “Tree of Thoughts: Deliberate Problem Solving with Large Language Models”. In: *NeurIPS*. 2023.

[43] Kang Min Yoo et al. “Ground-Truth Labels Matter: A Deeper Look into Input-Label Demonstrations”. In: *EMNLP*. 2022.

[44] Tao Yu et al. “Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task”. In: *EMNLP*. 2018.

[45] Chao Zhang et al. “FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis”. In: *SIGMOD*. 2024.

[46] Zhuosheng Zhang et al. “Automatic Chain of Thought Prompting in Large Language Models”. In: *ICLR*. 2023.

[47] Wayne Xin Zhao et al. “A Survey of Large Language Models”. In: *arXiv preprint arXiv:2303.18223* (2023).

[48] Yuxiang Zhou et al. “The Mystery of In-Context Learning: A Comprehensive Survey on Interpretation and Analysis”. In: *arXiv preprint arXiv:2311.00237* (2024).

4 参数高效微调

大语言模型从海量的预训练数据中掌握了丰富的世界知识。但“尺有所短”，对于预训练数据涉及较少的垂直领域，大语言模型无法仅通过提示工程来完成领域适配。为了让大语言模型更好的适配到这些领域，需要对其参数进行微调。但由于大语言模型的参数量巨大，微调成本高昂，阻碍了大语言模型在一些垂直领域的应用。为了降低微调成本，亟需实现效果可靠、成本可控的参数高效微调。本章将深入探讨当前主流的参数高效微调技术，首先简要介绍参数高效微调的概念、参数效率和方法分类，然后详细介绍参数高效微调的三类主要方法，包括参数附加方法、参数选择方法和低秩适配方法，探讨它们各自代表性算法的实现和优势。最后，本章通过具体案例展示参数高效微调在垂直领域的实际应用。

* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

4.1 参数高效微调简介

对于预训练数据涉及较少的垂直领域，大语言模型需要对这些领域及相应的下游任务进行适配。上下文学习和指令微调是进行下游任务适配的有效途径，但它们在效果或效率上存在缺陷。为弥补这些不足，参数高效微调（Parameter-Efficient Fine-Tuning, PEFT）技术应运而生。本节将首先对上下文学习和指令微调进行回顾，并分析它们的优缺点以及在实际应用中的局限性。接着，我们将详细介绍参数高效微调的概念及其重要性，阐述其在降低成本和提高效率方面的显著优势。随后，我们将对主流的 PEFT 方法进行分类，包括参数附加方法、参数选择方法和低秩适配方法，介绍每种方法的基本原理和代表性工作。

4.1.1 下游任务适配

通常，大语言模型通过在大规模数据集上进行预训练，能够积累丰富的世界知识，并获得处理多任务的能力 [43]。但由于开源大语言模型训练数据有限，因此仍存在知识边界，导致其在垂直领域（如医学、金融、法学等）上的知识不足，进而影响在垂直领域的性能表现。因此，需要进行下游任务适配才能进一步提高其在垂直和细分领域上的性能。主流的下游任务适配方法有两种：a) **上下文学习（In-context learning）** [8]；b）**指令微调（Instruction Tuning）** [53]。

1. 上下文学习

在之前的内容中，我们已经介绍过上下文学习的相关内容。它的核心思想是将不同类型的任务都转化为生成任务，通过设计 Prompt 来驱动大语言模型完成这些下游任务。小样本上下文学习（Few-shot in-context learning）将数据集中的样本-标签对转化为自然语言指令（Instruction）和样例（Demonstrations），并拼接上需要测试的样本一同输入给大语言模型，将模型输出作为最终预测结果。上下文学习完

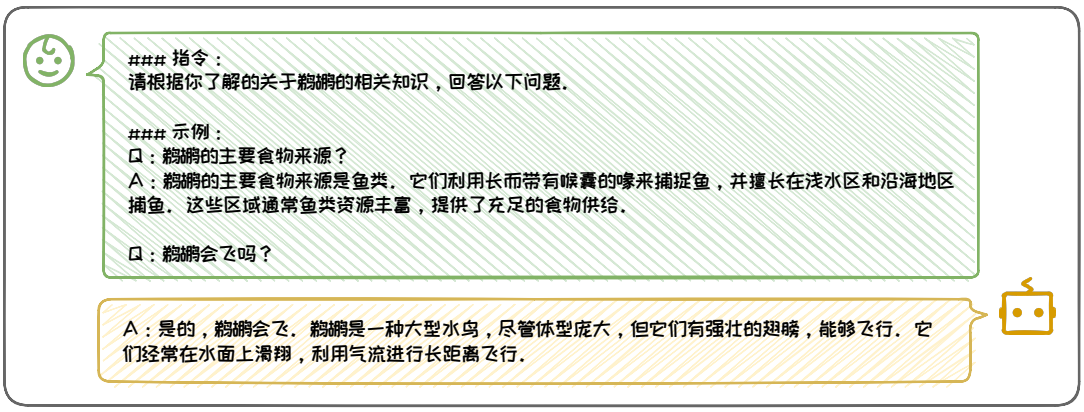


图 4.1: 指令数据样例。

全不需要对模型进行参数更新，因此能快速将单个模型应用到多种不同的任务上。

尽管在实际应用中，**上下文学习能有效利用大语言模型的能力，但它缺点也很明显**：1）上下文学习的性能和微调依旧存在差距，并且 Prompt 设计需要花费大量的人力成本，不同 Prompt 的最终任务性能有较大差异；2）上下文学习虽然完全不需要训练，但在推理阶段的代价会随 Prompt 中样例的增多快速增加。因此，微调大语言模型在许多场景和任务中依旧是必要的，尤其是在垂直领域。

2. 指令微调

指令微调（Instruction Tuning）是另一种进行下游任务适配的方法。指令微调旨在对模型进行任务指令的学习，使其能更好地理解和执行各种自然语言处理任务的指令。指令微调需首先构建指令数据集，然后在该数据集上进行监督微调。

- **指令数据构建**：指令数据通常包含指令（任务描述）、示例（可选）、问题和回答，如图 4.1 所示。构造这种指令数据一般有**两种方式** [53]：1）数据集成。通过使用模板将带标签的自然语言数据集，转换为指令格式的 < 输入，输出 > 对。如 Flan [47] 和 P3 [37] 数据集基于数据集成策略构建；2）大语言模型生成。通过人工收集或者手写少量指令数据，再使用 GPT-3.5-Turbo 或 GPT4 等闭源大语言模型进行指令扩展。如 InstructWild [33] 和 Self-Instruct [46] 数据集采用这种方法生成。

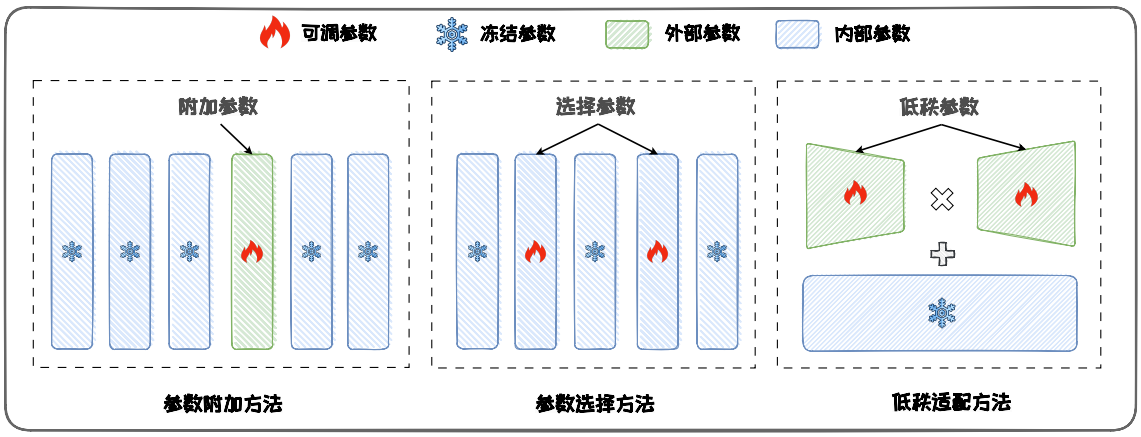


图 4.2: 高效参数微调方法分类学。

- **监督微调**：通过上述方法构建完数据集后，可以用完全监督的方式对预训练模型进行微调，在给定指令和输入的情况下，通过顺序预测输出中的每个 token 来训练模型。经过微调的大语言模型能够显著提升指令遵循（Instruction-following）能力，这有助于增强其推理水平，泛化到新任务和新领域。

尽管指令微调能有效帮助大语言模型理解新领域的数据知识，提高大语言模型在下游任务上的性能。然而，监督微调需要较大的计算资源，以 LLaMA2-7B [40] 模型为例，直接进行全量微调需要近 60GB 内存，普通的消费级 GPU（如 RTX4090（24GB））无法完成微调。因此，为了在资源受限的环境中有效微调大语言模型，研究参数高效的微调技术显得尤为重要。

4.1.2 参数高效微调

参数高效微调（Parameter-Efficient Fine-Tuning，PEFT）旨在避免微调全部参数，减少在微调过程中需要更新的参数数量和计算开销，从而提高微调大语言模型的效率。主流的 PEFT 方法可以分为三类：参数附加方法（Additional Parameters Methods），参数选择方法（Parameter Selection Methods）以及低秩适配方法（Low-Rank Adaptation Methods），其方法思想如图 4.2 所示。

1. 参数附加方法

参数附加方法（Additional Parameters Methods）在模型结构中附加新的、较小的可训练模块。在进行微调时，将原始模型参数冻结，仅微调这些新加入的模块，从而来实现高效微调。这些模块通常称为适应层（Adapter Layer）。它们被插入到模型的不同层之间，用于捕获特定任务的信息。由于这些新增的适应层参数量很小，所以参数附加方法能够显著减少需要更新的参数量。典型方法包括：**适配器微调（Adapter-tuning）**[18]、**提示微调（Prompt-tuning）**[23]、**前缀微调（Prefix-tuning）**[24] 和 **代理微调（Proxy-tuning）**[27] 等。参数附加方法将在 4.2 节进行具体介绍。

2. 参数选择方法

参数选择方法（Parameter Selection Methods）仅选择模型的一部分参数进行微调，而冻结其余参数。这种方法利用了模型中仅有部分参数对下游任务具有决定性作用的特性，“抓住主要矛盾”，仅微调这些关键参数。选择性地微调这些关键参数，可以在降低计算负担的同时提升模型的性能。典型的方法包括：**BitFit** [50]、**Child-tuning** [49] 以及 **FishMask** [39] 等。参数选择方法的将在 4.3 节具体介绍。

3. 低秩适配方法

低秩适配方法（Low-rank Adaptation Methods）通过低秩矩阵来近似原始权重更新矩阵，并冻结原始参数矩阵，仅微调低秩更新矩阵。由于低秩更新矩阵的参数数量远小于原始的参数更新矩阵，因此大幅节省了微调时的内存开销。LoRA [19] 是经典的低秩适配方法，后续有 **AdaLoRA** [52]、**DyLoRA** [42] 以及 **DoRA** [29] 等变体被提出，进一步改进了 LoRA 性能。低秩适配方法将在 4.4 节具体介绍。

4.1.3 参数高效微调的优势

参数高效微调有以下优势：1) **计算效率高**：PEFT 技术减少了需要更新的参数数量，从而降低了训练时的计算资源消耗；2）**存储效率高**：通过减少需要微调的参

数数量，PEFT 显著降低了微调模型的存储空间，特别适用于内存受限的设备；3）**适应性强**：PEFT 能够快速适应不同任务，而无需重新训练整个模型，使得模型在面对变化环境时具有更高的灵活性。

下面我们将通过一个具体案例来深入探讨 PEFT 技术如何显著提升参数效率。具体来说，表 4.1 [1] 详细展示了在配备 80GB 显存的 A100 GPU 以及 64GB 以上 CPU 内存的高性能硬件环境下，对 bigscience 模型进行全量微调与采用参数高效微调方法 LoRA（该方法将在 4.4节中详细介绍）时，GPU 内存的消耗情况对比。根据该

表 4.1: 全量参数微调和参数高效微调显存占用对比（OOM 代表超出内存限制）。

模型名	全量参数微调	参数高效微调 (LoRA)
bigscience/T0_3B	47.14GB GPU / 2.96GB CPU	14.4GB GPU / 2.96GB CPU
bigscience/mt0-xxl (12B params)	OOM GPU	56GB GPU / 3GB CPU
bigscience/bloomz-7b1 (7B params)	OOM GPU	32GB GPU / 3.8GB CPU

表格可以看出，对于 80GB 显存大小的 GPU，全量参数微调 7B/12B 参数的模型，会导致显存直接溢出。而在使用 LoRA 后，显存占用被大幅缩减，使得在单卡上微调大语言模型变得可行。

本节首先介绍了对大语言模型进行下游任务适配的两类主流范式：上下文学习和指令微调。然而，由于性能和计算成本方面的限制，这两类范式难以适应需求。因此，需要研究参数高效微调技术，即 PEFT。PEFT 仅对模型的一小部分参数进行更新，在保证不牺牲性能的前提下有效减少了模型微调所需的参数量，从而节约了计算和存储资源。本节我们对主流 PEFT 方法进行了分类，在后续小节中，我们将根据本节给出的分类学详细介绍主流的三类 PEFT 方法：参数附加方法 4.2、参数选择方法 4.3 以及低秩适配方法 4.4，并探讨 PEFT 的相关应用与实践 4.5。

[1] 表格数据来源：https://github.com/huggingface/peft

4.2 参数附加方法

参数附加方法（Additional Parameter Methods）通过增加并训练新的附加参数或模块对大语言模型进行微调。参数附加方法按照附加位置可以分为三类：加在输入、加在模型以及加在输出。本节将对三类方法的各自代表性工作进行具体介绍。

4.2.1 加在输入

加在输入的方法将额外参数附加到模型的输入嵌入（Embedding）中，其中最经典的方法是 Prompt-tuning [23]。Prompt-tuning 在模型的输入中引入可微分的连续张量，通常也被称为软提示（Soft prompt）。软提示作为输入的一部分，与实际的文本数据一起被送入模型。在微调过程中，仅软提示的参数会被更新，其他参数保持不变，因此能达到参数高效微调的目的。

具体地，给定一个包含 n 个 token 的输入文本序列 $\{w_1, w_2, \ldots, w_n\}$，首先通过嵌入层将其转化为**输入嵌入**矩阵 $X \in \mathbb{R}^{n \times d}$，其中 d 是嵌入空间的维度。新加入的软提示参数被表示为**软提示嵌入**矩阵 $P \in \mathbb{R}^{m \times d}$，其中 m 是软提示长度。然后，将软提示嵌入拼接上输入嵌入矩阵，形成一个新矩阵 $[P; X] \in \mathbb{R}^{(m+n) \times d}$，最后输入 Transformer 模型。通过反向传播最大化输出概率似然进行模型训练。训练过程仅软提示参数 P 被更新。图 4.3给出了 Prompt-tuning 的示意图。

在实际使用中，软提示的长度范围是 1 到 200，并且通常在 20 以上就能有一定的性能保证。此外，软提示的初始化对最终的性能也会有影响，使用词表中的 token 或者在分类任务中使用类名进行初始化会优于随机初始化。值得一提的是，Prompt-tuning 原本被提出的动机不是为了实现参数高效微调，而是自动学习提示词。在第三章我们提到，利用大语言模型的常见方式是通过提示工程，也被称为是硬提示（Hard prompt），这是因为我们使用的离散 prompt 是不可微分的。问题在于

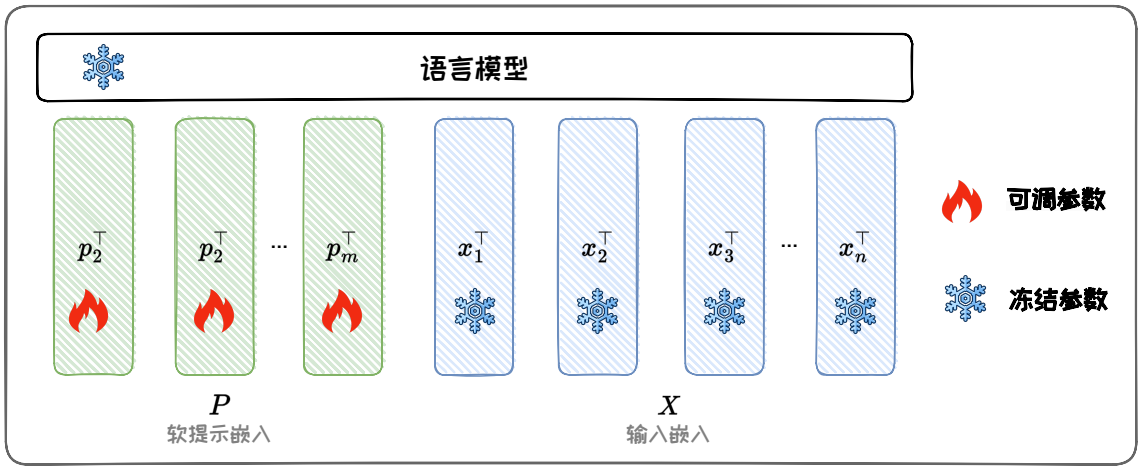


图 4.3: Prompt-tuning 示意图。

大语言模型的输出质量高度依赖于提示词的构建，要找到正确的“咒语”使得我们的大语言模型表现最佳，需要花费大量时间。因此，采用可微分的方式，通过反向传播自动优化提示词成为一种有效的方法。

总的来说，Prompt-tuning 有以下优势：(1) **内存效率高**：Prompt-tuning 显著降低了内存需求。例如，T5-XXL 模型对于特定任务的模型需要 11B 参数，但经过 Prompt-tuning 的模型只需要 20480 个参数（假设软提示长度为 5）；(2) **多任务能力**：可以使用单一冻结模型进行多任务适应。传统的模型微调需要为每个下游任务学习并保存任务特定的完整预训练模型副本，并且推理必须在单独的批次中执行。Prompt-tuning 只需要为每个任务存储一个特定的小的任务提示模块，并且可以使用原始预训练模型进行混合任务推理（在每个任务提示词前加上学到的 soft prompt）。(3) **缩放特性**：随着模型参数量的增加，Prompt-tuning 的性能会逐渐增强，并且在 10B 参数量下的性能接近（多任务）全参数微调的性能。

4.2.2 加在模型

加在模型的方法将额外的参数或模型添加到预训练模型的隐藏层中，其中经典的方法有 Prefix-tuning [24]、Adapter-tuning [18] 和 AdapterFusion [35]。

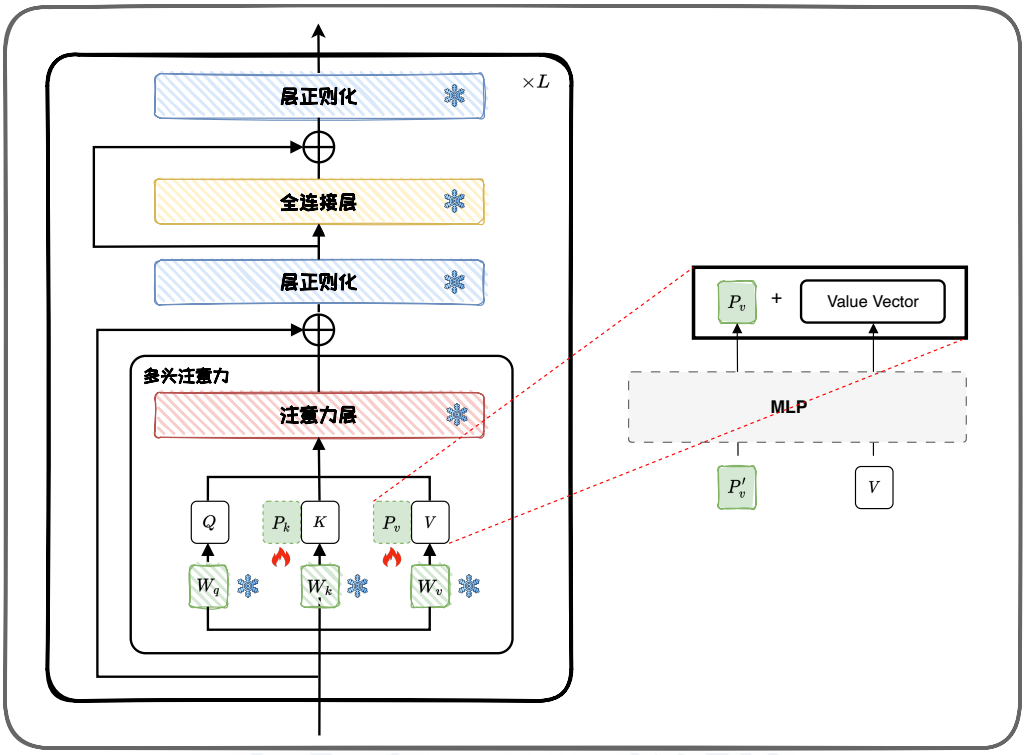


图 4.4: Prefix-tuning 示意图。

1. Prefix-tuning

Prefix-tuning 和上一节介绍的 Prompt-tuning 十分类似，区别在于 Prompt-tuning 仅将软提示添加到输入嵌入中，而 Prefix-tuning 将一系列连续的可训练前缀（Prefixes，即 Soft-prompt）插入到输入嵌入以及 Transformer 注意力模块中，如图 4.4所示，P_k 和 P_v 是插入到 Transformer block 中的前缀。相比 Prompt-tuning，Prefix-tuning 大幅增加了可学习的参数量。

具体而言，Prefix-tuning 引入了一组可学习的向量 P_k 和 P_v，这些向量被添加到所有 Transformer 注意力模块中的键 K 和值 V 之前。类似于 Prompt-tuning，Prefix-tuning 也会面临前缀参数更新不稳定的问题，从而导致优化过程难以收敛。因此，在实际应用中，通常需要在输入 Transformer 模型前，先通过一个多层感知机（MLP）进行重参数化。这意味着需要训练的参数包括 MLP 和前缀矩阵两部分。训练完成后，MLP 的参数会被丢弃，仅保留前缀参数。总的来说，Prefix-tuning 有

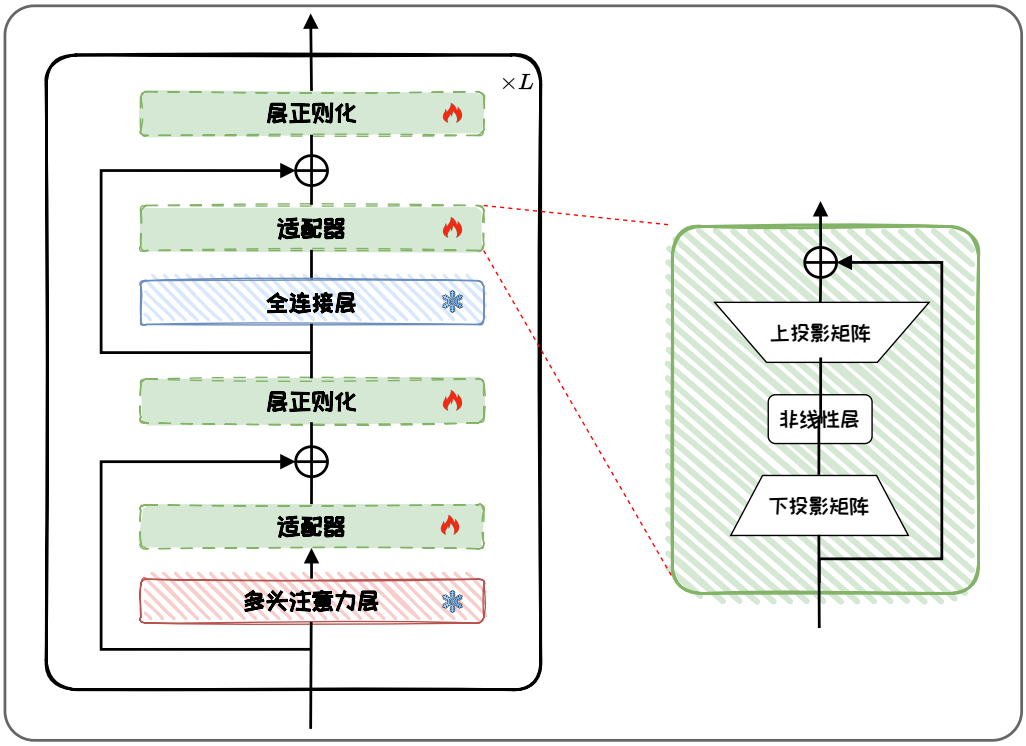


图 4.5: Adapter-tuning 示意图。

以下优势：（1）**参数效率**：只有前缀参数在微调过程中被更新，这显著减少了训练参数量；（2）**任务适应性**：前缀参数可以针对不同的下游任务进行定制，微调方式灵活；（3）**保持预训练知识**：由于预训练模型的原始参数保持不变，Prefix-tuning 能够保留预训练过程中学到的知识。

2. Adapter-tuning

Adapter-tuning [18] 向预训练语言模型中插入新的可学习的神经网络模块，称为适配器（Adapter）。适配器模块通常采用瓶颈（Bottomneck）结构，即一个上投影层、一个非线性映射和一个下投影层组成的全连接模块。其中，下投影层将信息压缩到一个低维的表示，经过非线性映射后再通过上投影层扩展回原始维度。如图 4.5所示，Adapter-tuning 在 Transformer 的每一个**多头注意力层**（Multi-head Attention Layer，图中红色块）和**全连接层**（Feed-forward Network Layer，图中蓝色块）之后添加适配器。**与 Transformer 的全连接层不同，由于采用了瓶颈结构，适**

配器的隐藏维度通常比输入维度小。和其他 PEFT 方法类似，在训练时，通过固定原始模型参数，仅对适配器、层正则化（图中绿色框）以及最后的分类层参数（图中未标注）进行微调，可以大幅缩减微调参数量和计算量，从而实现参数高效微调。适配器模块的具体结构如图 4.5 右边所示，适配器模块通常由一个下投影矩阵 $W_d \in \mathbb{R}^{d\times r}$ 和一个上投影矩阵 $W_u \in \mathbb{R}^{r\times d}$ 以及残差连接组成，其中 $r << d$：

$$A^{(l)} = \sigma(W_d * H^{(l-1)})W_u + H^{(l-1)}, \tag{4.1}$$

其中，$\sigma(\cdot)$ 是激活函数，如 ReLU 或 Sigmoid。$A^{(l)}$ 是适配器的输出，$H^{(l-1)}$ 是第 $l-1$ 层的隐藏状态。

在适配器中，下投影矩阵将输入的 d 维特征压缩到低维 r，再用上投影矩阵投影回 d 维。因此，每一层中的总参数量为 $2dr+d+r$，其中包括投影矩阵及其偏置项参数。通过设置 $r \ll d$，可以大幅限制每个任务所需的参数量。进一步地，适配器模块的结构还可以被设计得更为复杂，例如使用多个投影层，或使用不同的激活函数和参数初始化策略。此外，Adapter-tuning 还有许多变体，例如通过调整适配器模块的位置 [14, 56]、剪枝 [15] 等策略来减少可训练参数的数量。

3. AdapterFusion

由于 Adapter-tuning 无需更新预训练模型，而是通过适配器参数来学习单个任务，每个适配器参数都保存了解决该任务所需的知识。因此，如果想要结合多个任务的知识，可以考虑将多个任务的适配器参数结合在一起。基于该思路，AdapterFusion 提出一种两阶段学习的方法，先学习多个任务，对每个任务进行知识提取；再“融合”（Fusion）来自多个任务的知识。具体的两阶段步骤如下：

第一阶段：知识提取。给定 N 个任务，首先对每个任务分别训练适配器模块，用于学习特定任务的知识。该阶段有两种训练方式，分别如下：

- **Single-Task Adapters(ST-A)**：对于 N 个任务，模型都分别独立进行优化，各个任务之间互不干扰，互不影响。

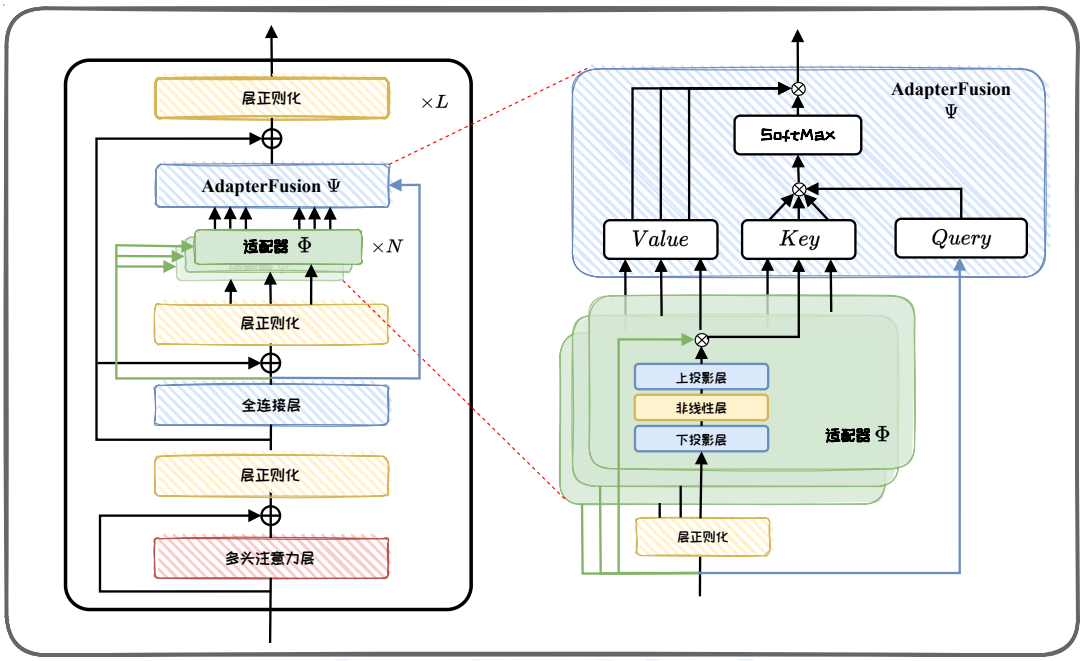


图 4.6: AdapterFusion 示意图。

- **Multi-Task Adapters(MT-A)**：通过多任务学习对 N 个任务进行联合优化。

第二阶段：知识组合。单个任务的适配器模块训练完成后，AdapterFusion 将不同适配器模块进行融合（Fusion），以实现知识组合。该阶段引入一个新的融合模块，该模块旨在搜索多个任务适配器模块的最优组合，实现任务泛化。在该阶段，语言模型的参数以及 N 个适配器的参数被固定，仅微调 AdapterFusion 模块的参数，并优化以下损失：

$$\Psi \leftarrow \operatorname{argmin}_{\Psi} L_n\left(D_n; \Theta, \Phi_1, \ldots, \Phi_N, \Psi\right) 。 \tag{4.2}$$

其中，D_n 是第 n 个任务的训练数据，L_n 是第 n 个任务的损失函数，Θ 表示预训练模型参数，Φ_i 表示第 i 个任务的适配器模块参数，Ψ 是融合模块参数。图 4.6 给出 AdapterFusion 的示意图。每层的 AdapterFusion 模块包括可学习的 Key（键）、Value（值）和 Query（查询）等对应的投影矩阵。全连接层的输出当作 Query，适配器的输出当作 Key 和 Value，计算注意力得到联合多个适配器的输出结果。此外，在不同的适配器模块间参数共享融合模块，可以进一步减少参数数量。

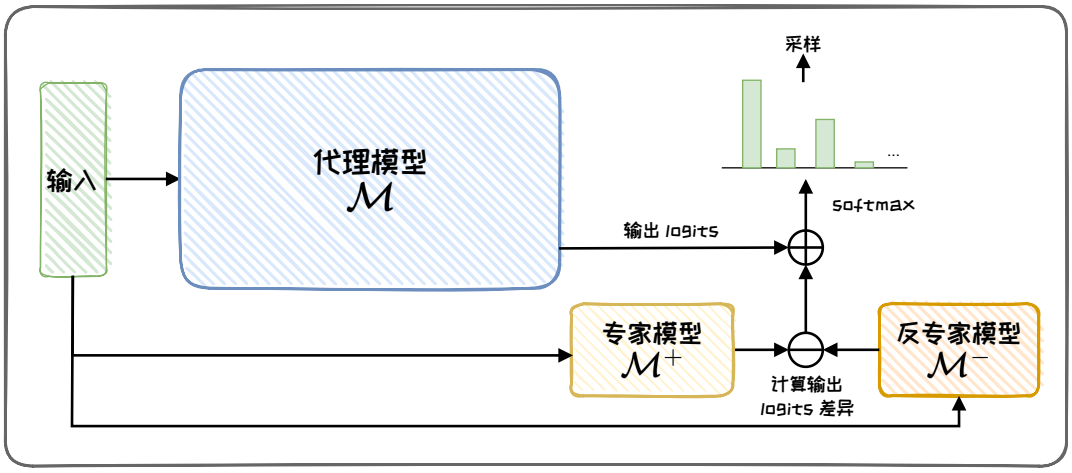


图 4.7: Proxy-tuning 示意图。

4.2.3 加在输出

在微调大语言模型时，通常会面临以下问题：首先，大语言模型的**参数数量可能会非常庞大**，例如 LLaMA 系列最大的模型拥有 70B 参数，即使采用了 PEFT 技术，也难以在普通消费级 GPU 上完成下游任务适应；其次，用户可能**无法直接访问大语言模型的权重**（黑盒模型），这为微调设置了障碍。

为了应对这些问题，代理微调（Proxy-tuning）[27] 提供了一种轻量级的解码时（Decoding-time）算法，允许我们在不直接修改大语言模型权重的前提下，通过仅访问模型输出词汇表预测分布，来实现对大语言模型的进一步定制化调整。

如图 4.7 所示，给定待微调的**代理模型** $\mathcal{M}$ 以及较小的**反专家模型**（Anti-expert model）$\mathcal{M}^-$，这两个模型需要相同的词汇表。我们对 $\mathcal{M}^-$ 进行微调，得到微调后的**专家模型**（Expert model）$\mathcal{M}^+$。在每一个自回归生成的时间步中，代理微调首先计算专家模型 $\mathcal{M}^+$ 和反专家模型 $\mathcal{M}^-$ 之间的 *logits* 分布差异，然后将其加到代理模型 $\mathcal{M}$ 下一个词预测的 *logits* 分布中。具体来说，在代理微调的计算阶段，针对每一时间步 t 的输入序列 $x_{<t}$，从代理模型 $\mathcal{M}$、专家模型 $\mathcal{M}^+$ 和反专家模型 $\mathcal{M}^-$

中获取相应的输出分数 $s_{\mathcal{M}}, s_{\mathcal{M}^+}, s_{\mathcal{M}^-}$。通过下式调整目标模型的输出分数 $\tilde{s}$：

$$\tilde{s} = s_{\mathcal{M}} + s_{\mathcal{M}^+} - s_{\mathcal{M}^-}, \tag{4.3}$$

然后，使用 softmax(·) 对其进行归一化，得到输出概率分布，

$$p_{\tilde{\mathcal{M}}}(X_t \mid x_{<t}) = \text{softmax}(\tilde{s}), \tag{4.4}$$

最后，在该分布中采样得到下一个词的预测结果。

在实际使用中，通常专家模型是较小的模型（例如，LLaMA-7B），而代理模型则是更大的模型（例如，LLaMA-13B 或 LLaMA-70B）。通过代理微调，我们将较小模型中学习到的知识，以一种解码时约束的方式迁移到比其大得多的模型中，大幅节省了计算成本。同时，由于仅需要获取模型的输出分布，而不需要原始的模型权重，因此该方法对于黑盒模型同样适用。

本节介绍了三种主要的参数附加方法，分别通过加在输入、加在模型以及加在输出三种方式实现。总的来说，这几类方法都是提升预训练语言模型性能的有效手段，它们各有优势。加在输入的方法通过在输入序列中添加可学习的张量，对模型本身的结构修改较小，有更好的灵活性。加在模型的方法保持了原始预训练模型的参数，在泛化能力上表现更佳。加在输出的方法则能够以更小的代价驱动更大参数量的黑盒模型，带来更实际的应用前景。

4.3 参数选择方法

参数选择方法（Parameter Selection Methods）选择性的对预训练模型中的某个参数子集进行微调。和参数附加方法不同的是，参数选择方法无需向模型添加额外的参数，避免了在推理阶段引入额外的计算成本。通常，参数选择方法分为两类：基于规则的方法和基于学习的方法。

4.3.1 基于规则的方法

基于规则的方法根据人类专家的经验，确定哪些参数应该被更新。基于规则的方法中最具代表性的方法是 BitFit [50]。BitFit 通过仅优化神经网络中的每一层的偏置项（Biases）以及任务特定的分类头来实现参数高效微调。由于偏置项在模型总参数中所占比例极小（约 0.08%-0.09%），BitFit 有极高的参数效率。尽管只微调少量参数，BitFiT 依然能在 GLUE Benchmark [44] 上与全量微调相媲美，甚至在某些任务上表现更好。此外，BitFit 方法相比全量微调允许使用更大的学习率，因此该方法整体优化过程更稳定。然而，该方法仅在小模型（如 BERT、RoBERT 等）上进行验证性能，在更大模型上的性能表现如何尚且未知。

除 BitFit 以外，还有一些其他基于规则的方法通过仅微调特定的 Transformer 层来提高参数效率。例如，Lee 等人 [22] 提出，仅对 BERT 和 RoBERTa 的最后四分之一层进行微调，便能实现完全参数微调 90% 的性能。PaFi [26] 选择具有最小绝对值的模型参数作为可训练参数。

4.3.2 基于学习的方法

基于学习的方法在模型训练过程中自动地选择可训练的参数子集。其中，最为典型方法是 Child-tuning [49] 。其通过**梯度掩码矩阵**策略实现仅对选中的择子网络进行梯度更新，而屏蔽子网络梯度以外的梯度，从而实现对微调参数的选择，达到参数高效微调的目的。具体而言，假设 $\mathbf{W}_t$ 是第 t 轮迭代的参数矩阵，我们引入了一个与 $\mathbf{W}_t$ 同维度的 0-1 掩码矩阵 $\mathbf{M}_t$ ，用于选择第 t 轮迭代的子网络 $\mathbf{C}_t$，仅更新该子网络的参数，其定义如下：

$$\mathbf{M}_t^{(i)} = \begin{cases} 1, & \text{if } \mathbf{W}_t^{(i)} \in \mathbf{C}_t \\ 0, & \text{if } \mathbf{W}_t^{(i)} \notin \mathbf{C}_t。 \end{cases} \tag{4.5}$$

其中，$\mathbf{M}_t^{(i)}$ 和 $\mathbf{W}_t^{(i)}$ 分别是矩阵 $\mathbf{M}_t$ 和 $\mathbf{W}_t$ 在第 t 轮迭代的第 i 个元素。此时，梯度更新公式为：

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \left(\frac{\partial \mathcal{L}(\mathbf{W}_t)}{\partial \mathbf{W}_t} \odot \mathbf{M}_t \right), \tag{4.6}$$

Child-tuning 提供了两种生成子网络掩码 $\mathbf{M}$ 的方式，由此产生两种变体模型：Child-tuning$_F$ 和 Child-tuning$_D$。Child-tuning$_F$ 是一种**任务无关**的变体，它在不依赖任何下游任务数据的情况下选择子网络。在每次迭代时，Child-tuning$_F$ 从伯努利分布中抽取 0-1 掩码，生成梯度掩码 $\mathbf{M}_t$：

$$\mathbf{M}_t \sim \text{Bernoulli}(p_F), \tag{4.7}$$

其中，p_F 是伯努利分布的概率，表示子网络的比例。此外，Child-tuning$_F$ 通过引入噪声来对全梯度进行正则化，从而防止小数据集上的过拟合，并提高泛化能力。

Child-tuning$_D$ 是一种**任务驱动**的变体，它利用下游任务数据来选择与任务最相关的子网络。具体来说，Child-tuning$_D$ 使用费舍尔信息矩阵（FIM）来估计特定任务相关参数的重要性。具体地，对于给定的任务训练数据 $\mathcal{D}$，模型的第 i 个参数矩阵 $W^{(i)}$ 的费舍尔信息估计为：

$$F^{(i)}(W) = \frac{1}{|\mathcal{D}|} \sum_{j=1}^{|\mathcal{D}|} \left(\frac{\partial \log p(Y_j|X_j;W)}{\partial W^{(i)}} \right)^2, \tag{4.8}$$

其中，X_i 和 Y_i 分别表示第 i 个样本的输入和输出，$\log p(Y_i|X_i;W)$ 是对数似然概率，通过计算损失函数对参数 $W^{(i)}$ 的梯度得到。通常，我们假设参数对目标任务越重要，它的费舍尔信息的值就越高。因此，可以根据费舍尔信息来选择子网络，子网络 $\mathbf{C}$ 由具有最高费舍尔信息的参数组成。选择子网络参数的步骤如下：1) 计算每个参数的费舍尔信息值；2) 对这些费舍尔信息值进行排序；3) 选择前 p_D 比例的参数作为子网络 $\mathbf{C}$。确定子网络后，生成相应的掩码矩阵完成模型训练。

Child-tuning 通过梯度屏蔽减少了计算负担，同时减少了模型的假设空间，降低了模型过拟合的风险。然而，子网络的选择需要额外的计算代价，特别是在任

务驱动的变体中，费舍尔信息的计算十分耗时。但总体而言，Child-tuning 可以改善大语言模型在多种下游任务中的表现，尤其是在训练数据有限的情况下。此外，Child-tuning 可以很好地与其他 PEFT 方法的集成，进一步提升模型性能。

除 Child-tuning 外，还有一些其他基于学习的参数选择方法。例如，Zhao 等人 [55] 引入与模型权重相关的二值矩阵来学习，通过阈值函数生成参数掩码（mask），然后在反向传播过程中通过噪声估计器进行更新。类似 FishMask，Fish-Dip [5] 也使用 Fisher 信息来计算掩码，但掩码将在每个训练周期动态重新计算。LT-SFT [2] 受 “彩票假设” [11] 启发，根据参数重要性，根据在初始微调阶段变化最大的参数子集形成掩码。SAM [12] 提出了一个二阶逼近方法，通过解析求解优化函数来帮助决定参数掩码。

基于选择的方法通过选择性地更新预训练模型的参数，在保持大部分参数不变的情况下对模型进行微调。基于选择的方法能够显著减少微调过程中所需要更新的参数，降低计算成本和内存需求。对于资源受限的环境或者需要快速适应新任务的场景尤其适用。然而，这些方法也面临挑战，比如，如何选择最佳参数子集，以及如何平衡参数更新的数量和模型性能之间的关系。

4.4 低秩适配方法

低维固有维度假设 [1] 表明：过参数化模型的固有维度是很低的；换言之，存在可以与全参数更新媲美的低维的参数更新。基于这一假设，低秩适配方法（Low-rank Adaptation Methods）通过低秩矩阵来近似原始权重权更新矩阵，并仅微调低秩矩阵，以大幅降低模型参数量。在本节中，我们首先将介绍最经典的低秩适配方法 LoRA 的实现细节并对分析其参数效率。接着，将介绍 LoRA 的相关变体。最后，介绍基于 LoRA 插件化特性，以及其任务泛化能力。

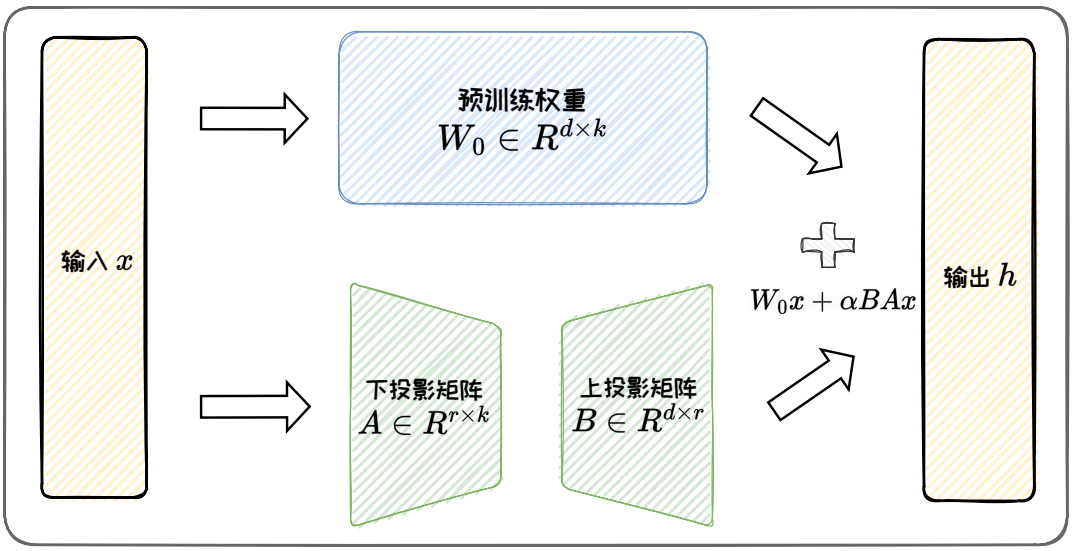


图 4.8: LoRA 示意图。

4.4.1 LoRA

低秩适配（Low-rank Adaptation, LoRA）[19] 提出利用低秩矩阵近似参数更新矩阵来实现低秩适配。该方法将参数更新矩阵低秩分解为两个小矩阵。在微调时，通过微调这两个小矩阵来对大语言模型进行更新，大幅节省了微调时的内存开销。本小节首先介绍 LoRA 方法的具体实现过程，然后分析其计算效率。

1. 方法实现

给定一个密集神经网络层，其参数矩阵为 $W_0 \in \mathbb{R}^{d\times k}$，为适配下游任务，我们通常需要学习参数更新矩阵 $\Delta W \in \mathbb{R}^{d\times k}$，对原始参数矩阵进行更新 $W = W_0 + \Delta W$。对于全量微调过程，ΔW 是需对该层所有 $d \times k$ 个参数计算梯度，这通常需要大量的 GPU 内存，成本高昂。为解决这一问题，如图 4.8，LoRA 将 ΔW 分解为两个低参数量的矩阵 $B \in \mathbb{R}^{d\times r}$ 和 $A \in \mathbb{R}^{r\times k}$，使得更新过程变为：

$$W = W_0 + \alpha BA, \tag{4.9}$$

其中，秩 $r \ll min\{d, k\}$，B 和 A 分别用随机高斯分布和零进行初始化，α 是缩放

因子，用于控制 LoRA 权重的大小。在训练过程中，固定预训练模型的参数，仅微调 B 和 A 的参数。因此，在训练时，LoRA 涉及的更新参数数量为 $r \times (d+k)$，远小于全量微调 $d \times k$。实际上，对于基于 Transformer 的大语言模型，密集层通常有两种类型：注意力模块中的投影层和前馈神经网络（FFN）模块中的投影层。在原始研究中，LoRA 被应用于注意力层的权重矩阵。后续工作表明将其应用于 FFN 层可以进一步提高模型性能 [14]。

LoRA 仅微调部分低秩参数，因此具有很高的参数效率，同时不会增加推理延迟 [10]。此外，低秩矩阵还可以扩展为低秩张量 [3]，或与 Kronecker 分解结合使用，以进一步提高参数效率 [9, 16]。除了参数效率外，在训练后可以将 LoRA 参数与模型参数分离，所以 LoRA 还具有**可插拔性**。LoRA 的可插拔特性使其能够封装为被多个用户共享和重复使用 [38] 的插件。当我们有多个任务的 LoRA 插件时，可以将这些插件组合在一起，以获得良好的跨任务泛化性能 [20]。我们将在 4.4.3 提供具体案例详细介绍基于 LoRA 插件的任务泛化。

2. 参数效率

下面我们以一个具体的案例分析 LoRA 的参数效率。在 LLaMA2-7B [40] 模型中微调第一个 FFN 层的权重矩阵为例，全量微调需要调整 $11,008 \times 4,096 = 45,088,768$ 个参数。而当 $r = 4$ 时，LoRA 只需调整 $(11,008 \times 4) + (4 \times 4,096) = 60,416$ 个参数。对于这一层，与全量微调相比，LoRA 微调的参数不到原始参数量的千分之一。具体来说，模型微调的内存使用主要涉及四个部分：

- 权重内存（Weight Memory）：用于存储模型权重所需的内存；
- 激活内存（Activation Memory）：前向传播内存时中间激活带来的显存占用，主要取决于 batch size 大小以及序列长度等；
- 梯度内存（Gradient Memory）：在反向传播期间需要用来保存梯度的内存，这些梯度仅针对可训练参数进行计算；

- 优化器内存（Optimization Memory）：用于保存优化器状态的内部存在。例如，Adam 优化器会保存可训练参数的 “一阶动量” 和 “二阶动量”。

文献 [34] 提供在 LLaMA2-7B 模型上使用批量大小为 1，单个 NVIDIA RTX4090（24GB）GPU 上进行全量微调和 LoRA 微调的实验对比。根据实验结果，全量微调大约需要 60GB 显存，超出 RTX4090 的显存容量。相比之下，LoRA 只需要大约 23GB 显存。LoRA 显著减少了显存使用，使得在单个 NVIDIA RTX4090 上进行 LLaMA2-7B 微调成为可能。具体来说，**由于可训练参数较少，优化器内存和梯度内存分别减少了约 25GB 和 14GB**。另外，虽然 LoRA 引入了额外的 “增量参数”，导致**激活内存和权重内存略微增加（总计约 2GB）**，但考虑到整体内存的减少，这种增加是可以忽略不计的。此外，减少涉及到的参数计算可以加速反向传播。与全量微调相比，LoRA 的速度提高了 1.9 倍。

4.4.2 LoRA 相关变体

虽然 LoRA 在一些下游任务上能够实现较好的性能，但在许多复杂的下游任务（如数学推理 [4, 6, 54]）上，LoRA 与全量微调之间仍存在性能差距。为弥补这一差距，许多 LoRA 变体方法被提出，以进一步提升 LoRA 在下游任务中的适配性能。现有方法主要从以下几个角度进行改进 [31]：(1) 打破低秩瓶颈；(2) 动态秩分配；(3) 训练过程优化。接下来，将分别介绍这三种类型变种的代表性方法。

1. 打破低秩瓶颈

LoRA 的低秩更新特性使其在参数效率上具有优势；然而，这也限制了大规模语言模型记忆新知识和适应下游任务的能力 [4, 13, 21, 54]，即存在低秩瓶颈。Biderman 等人 [4] 的实验研究表明，全量微调的秩显著高于 LoRA 的秩（10-100 倍），并且增加 LoRA 的秩可以缩小 LoRA 与全量微调之间的性能差距。因此，一些方法被提出，旨在打破低秩瓶颈 [25, 36, 48]。

例如，ReLoRA [25] 提出了一种合并和重置（merge-and-reinit）的方法，该方法在微调过程中周期性地将 LoRA 模块合并到大语言模型中，并在合并后重新初始化 LoRA 模块和优化器状态。具体地，合并的过程如下：

$$W^i \leftarrow W^i + \alpha B^i A^i, \tag{4.10}$$

其中，W^i 是原始的权重矩阵，B^i 和 A^i 是低秩分解得到的矩阵，α 是缩放因子。合并后，将重置 B^i 和 A^i 的值重置，通常 B^i 会使用特定的初始化方法（如 Kaiming 初始化）重新初始化，而 A^i 则被设置为零。为了防止在重置后模型性能发散，ReLoRA 还会通过幅度剪枝对优化器状态进行部分重置。合并和重置的过程允许模型在保持总参数量不变的情况下，通过多次低秩 LoRA 更新累积成高秩状态，从而使得 ReLoRA 能够训练出性能接近全秩训练的模型。

2. 动态秩分配

然而，LoRA 的秩并不总是越高越好，冗余的 LoRA 秩可能会导致性能和效率的退化。并且，微调时权重的重要性可能会因 Transformer 模型中不同层而存在差异，因此需要为每个层分配不同的秩 [7, 30, 41, 52]。

例如，AdaLoRA [52] 通过将参数更新矩阵参数化为奇异值分解（SVD）的形式，再通过奇异值剪枝动态调整不同层中 LoRA 模块的秩。具体地，AdaLoRA 使用奇异值分解重新表示 ΔW，即

$$W = W_0 + \Delta W = W_0 + P\Lambda Q, \tag{4.11}$$

其中，$P \in \mathbb{R}^{d\times r}$ 和 $Q \in \mathbb{R}^{r\times k}$ 是正交的，Λ 是一个对角矩阵，其中包含 $\{\lambda_i\}_{1\le i\le r}$ 的奇异值。在训练过程中，W_0 的参数被固定，仅更新 P、Λ 和 Q 的参数。根据梯度权重乘积大小的移动平均值构造奇异值的重要性得分，对不重要的奇异值进行迭代剪枝。此外，为了增强稳定训练性，AdaLoRA 引入一个额外的惩罚项确保 P

和 Q 之间的正交性:

$$R(P,Q) = ||P^TP - I||_F^2 + ||QQ^T - I||_F^2, \tag{4.12}$$

其中，I 是单位矩阵，$||\cdot||_F$ 代表 Frobenius 范数。

3. 训练过程优化

在实际微调过程中，LoRA 的收敛速度比全量微调要慢。此外，它还对超参数敏感，并且容易过拟合。这些问题影响了 LoRA 的效率并阻碍了其下游适配性能。为了解决这些问题，一些工作尝试对 LoRA 的训练过程进行优化 [32, 45]。代表性方法 DoRA (权重分解低秩适应) [29] 提出约束梯度更新，侧重于更新参数的方向变化。它将预训练权重 $W_0 \in \mathbb{R}^{d\times k}$ 分解为方向和大小两个组件，并仅将 LoRA 应用于方向组件以增强训练稳定性。具体地，DoRA 将 $W_0 \in \mathbb{R}^{d\times k}$ 重新表示为:

$$W_0 = m\frac{V}{\|V\|_c} = \|W_0\|_c \frac{W_0}{\|W_0\|_c}, \tag{4.13}$$

其中，$m \in \mathbb{R}^{1\times k}$ 是大小向量, $V \in \mathbb{R}^{d\times k}$ 是方向矩阵，$\|\cdot\|_c$ 是矩阵在每一列上的向量范数。随后，DoRA 仅对方向矩阵 V 施加 LoRA 进行参数化，定义为:

$$W' = \underline{m}\frac{V + \underline{\Delta V}}{\|V + \underline{\Delta V}\|_c} = \underline{m}\frac{W_0 + \underline{BA}}{\|W_0 + \underline{BA}\|_c}, \tag{4.14}$$

其中，ΔV 是由 LoRA 学习的增量方向更新，下划线参数表示可训练参数。

4.4.3 基于 LoRA 插件的任务泛化

在 LoRA 微调结束后，我们可以将参数更新模块 B 和 A 从模型上分离出来，并封装成参数插件。这些插件具有即插即用、不破坏原始模型参数和结构的优良性质。我们可以在不同任务上训练的各种 LoRA 模块，将这些模块插件化地方式保存、共享与使用。此外，我们还可以将多个任务的 LoRA 插件组合，然后将不同任务的能力迁移到新任务上。LoRAHub [20] 提供了一个可用的多 LoRA 组合的方法框架。其可将已有任务上得到的 LoRA 插件进行组合，从而获得解决新任务的能

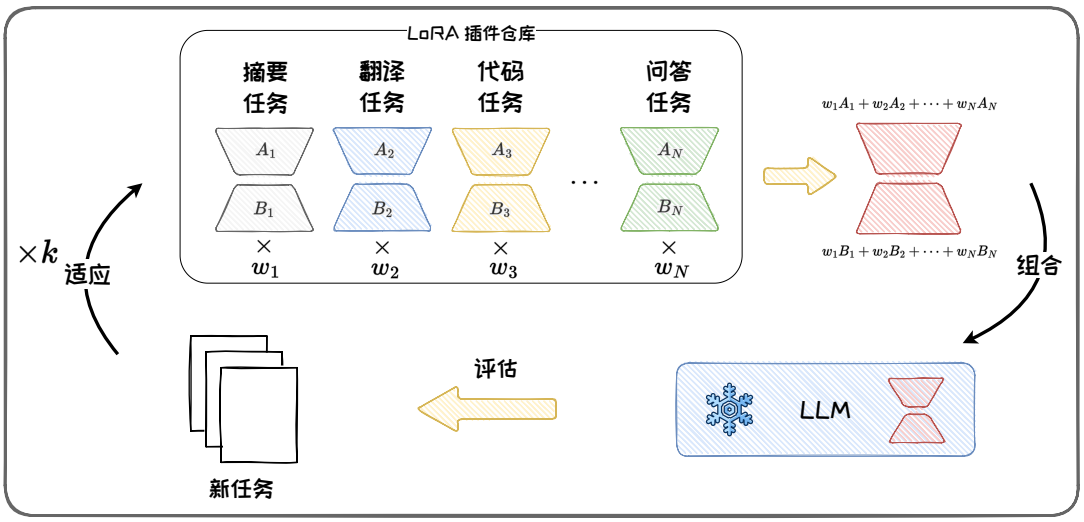


图 4.9: LoRAHub 示意图。

力。如图 4.9 所示，LoRAHub 包括两个阶段：**组合阶段**和**适应阶段**。**在组合阶段**，LoRAHub 将已学习的 LoRA 模块通过逐元素线性加权组合为单一模块：

$$\hat{m} = (w_1A_1 + w_2A_2 + \cdots + w_NA_N)\,(w_1B_1 + w_2B_2 + \cdots + w_NB_N)\,, \tag{4.15}$$

其中，w_i 是第 i 个 LoRA 模块的权重，$\hat{m}$ 是组合后的模块，$A_{i=1}^{N}$ 和 $B_{i=1}^{N}$ 分别是 N 个 LoRA 分解矩阵。**在适应阶段**，给定一些新任务的示例，通过无梯度方法 Shiwa [28] 自适应地学习权重组合。适应和组合经过 k 次迭代，直至找到最优的权重组合，以完成对新任务的适应。

本节介绍了低秩适配方法 LoRA。LoRA 通过对参数矩阵进行低秩分解，仅训练低秩矩阵，大幅减少了训练参数量。此外，还介绍了从打破低秩瓶颈、动态秩分配和训练过程优化等不同角度改进 LoRA 的变体。最后，介绍了基于 LoRA 插件的任务泛化方法 LoRAHub。LoRAHub 通过对已学习的 LoRA 模块加权组合，融合多任务能力并迁移到新任务上，提供了一种高效的跨任务学习范式。

4.5 实践与应用

PEFT 技术在许多领域中展现了其强大的应用潜力。本节将探讨 PEFT 的实践与应用。在实践部分，我们将介绍目前最流行的 PEFT 框架，Hugging Face 开发的开源库 HF-PEFT，并简述其使用方法和相关技巧。在应用部分，我们将展示 PEFT 技术在不同垂直领域中的应用案例，包括表格数据处理和金融领域的 Text-to-SQL 生成任务。这些案例不仅证明了 PEFT 在提升大模型特定任务性能方面的有效性，也为未来的研究和应用提供了有益的参考。

4.5.1 PEFT 实践

在实际应用中，PEFT 技术的实施和优化至关重要。本小节将详细介绍 PEFT 主流框架的使用，包括安装和配置、微调策略选择、模型准备与训练流程等内容，以及相关使用技巧。

1. PEFT 主流框架

目前最流行的 PEFT 框架是由 Hugging Face 开发的开源库 HF-PEFT [2]，它旨在提供最先进的参数高效微调方法。HF-PEFT 框架的设计哲学是高效和灵活性。HF-PEFT 集成了多种先进的微调技术，如 LoRA、Apdater-tuning、Prompt-tuning 和 IA3 等。HF-PEFT 支持与 Hugging Face 的其他工具如 Transformers、Diffusers 和 Accelerate 无缝集成，并且支持从单机到分布式环境的多样化训练和推理场景。HF-PEFT 特别适用于大模型，能够在消费级硬件上实现高性能，并且可以与模型量化技术兼容，进一步减少了模型的内存需求。HF-PEFT 支持多种架构模型，包括 Transformer 和 Diffusion，并且允许用户手动配置，在自己的模型上启用 PEFT。

HF-PEFT 的另一个显著优势是它的易用性。它提供了详细的文档、快速入门

[2]https://github.com/huggingface/peft

指南和示例代码[3]，可以帮助用户了解如何快速训练 PEFT 模型，以及如何加载已有的 PEFT 模型进行推理。此外，HF-PEFT 拥有活跃的社区支持，并鼓励社区贡献，是一个功能强大、易于使用且持续更新的库。

2. HF-PEFT 框架使用

使用 HF-PEFT 框架进行模型微调可以显著提升模型在特定任务上的性能，同时保持训练的高效性。通常，使用 HF-PEFT 框架进行模型微调的步骤如下：

1. **安装与配置**：首先，在环境中安装 HF-PEFT 框架及其依赖项，主要是 Hugging Face 的 Transformers 库。
2. **选择模型与数据**：根据任务需求，挑选合适的预训练模型，并准备相应的训练数据集。
3. **确定微调策略**：选择适合任务的微调方法，例如 LoRA 或适配器技术。
4. **模型准备**：加载预训练模型并为选定的微调方法进行配置，包括任务类型、推理模式、参数值等。
5. **模型训练**：定义完整的训练流程，包括损失函数、优化器设置，并执行训练，同时可应用数据增强和学习率调度等技术。

通过上述步骤，可以高效地使用 HF-PEFT 框架进行模型微调，以适应特定的任务需求，并提升模型性能。

3. PEFT 相关技巧

HF-PEFT 库提供了多种参数高效微调技术，用于在不微调所有模型参数的情况下，有效地将预训练语言模型适应各种下游应用。以下是一些 PEFT 技术常用的参数设置方法：

Prompt Tuning

- num_virtual_tokens: 表示为每个任务添加的 virtual tokens 的数量，也就是软

[3]https://huggingface.co/docs/peft/

提示的长度，该长度通常设置在 10-20 之间，可根据输入长度进行适当调节。

- prompt_tuning_init：表示 prompt 参数的初始化方式。可以选择随机初始化（RANDOM）、文本初始化（TEXT），或者其他方式。文本初始化方式下，可以使用特定文本对 prompt embeddings 进行初始化以加速收敛。

Prefix Tuning

- num_virtual_tokens: 与 Prompt Tuning 中相同，表示构造的 virtual tokens 的数量，设置和 Prompt Tuning 类似。
- encoder_hidden_size: 表示用于 Prefix 编码的多层感知机（MLP）层的大小，通常与模型的隐藏层大小相匹配。

LoRA

- r: 秩的大小，用于控制更新矩阵的复杂度。通常可以选择较小的值如 4、8、16，对于小数据集，可能需要设置更小的 r 值。
- lora_alpha: 缩放因子，用于控制 LoRA 权重的大小，通常与 r 成反比，以保持权重更新的一致性。
- lora_dropout: LoRA 层的 dropout 比率，用于正则化以防止过拟合，可以设置为一个较小的值，比如 0.01。
- target_modules: 指定模型中 LoRA 中要应用的模块，如注意力机制中的 query、key、value 矩阵。可以选择特定的模块进行微调，或者微调所有线性层。

需要注意的是，具体的参数设置可能会根据所使用的模型、数据集和具体任务有所不同，因此在实际应用中可能需要根据实验结果进行调整。

4.5.2 PEFT 应用

在现实应用中，大量**垂直领域数据**以表格的形式存储在关系型数据库中。不仅数量大，表格数据涵盖的领域也非常广泛，涉及金融、医疗、商务、气候科学众

多领域。然而，大语言模型的预训练语料中表格数据的占比通常很少，这导致其在表格数据相关的下游任务上的性能欠佳。微调大语言模型使其适应表格数据是参数高效微调的典型应用，其具有深远的现实意义。本节我们将介绍两个分别将参数高效微调技术应用在表格数据查询和表格数据分析上案例。

1. 表格数据查询

表格数据查询是处理、分析表格数据的必要步骤。对表格数据进行查询，需要编写专业的结构化查询语言（Structured Query Language，SQL）代码。然而，SQL 代码通常涉及复杂的查询逻辑和严格的语法规则，专业人士都需要花费大量时间进行编写。对于初学者，熟练掌握并使用 SQL 可能需要数月或者更长的时间来学习和实践。编写 SQL 代码的为表格数据查询设置了较高的门槛。为了降低表格数据查询的门槛，可以将自然语言文本自动翻译成 SQL 代码的 Text-to-SQL 技术受到了广泛的关注。其可以将 SQL 编写时间从分钟级缩短到秒级，使数据科学家能够专注于分析和建模，加快数据驱动决策的速度，同时让广大普通人也能操作和管理数据库，显著提高了数据分析的效率，让更多的人能够挖掘和利用数据的价值。大语言模型的强大代码生成能力为 Text-to-SQL 技术带来了新的机遇。但是，在众多垂直领域，如金融领域，难以收集足够的用于微调的数据。利用少量数据进行全参数微调时则容易导致大模型过拟合，因此采用 PEFT 技术进行部分参数微调十分适合金融等垂直领域 Text-to-SQL 任务。

面向金融垂直领域，FinSQL [51] 提出了一套针对金融垂直领域 Text-to-SQL 训练推理框架。如图 4.10 所示，该框架包含提示构造、参数高效微调和输出校准三个部分。首先，提示构造通过不同策略增强原始数据，并且构造高效检索器，检索与用户查询相关表格和字段。然后，采用 PEFT 技术对基座模型进行微调，通过 LoRAHub 融合多个 LoRA 模块，提高少样本场景的性能。最后，输出校准模块会修正生成 SQL 中的语法错误，并用 Self-Consistency 方法来选择一致性 SQL。FinSQL

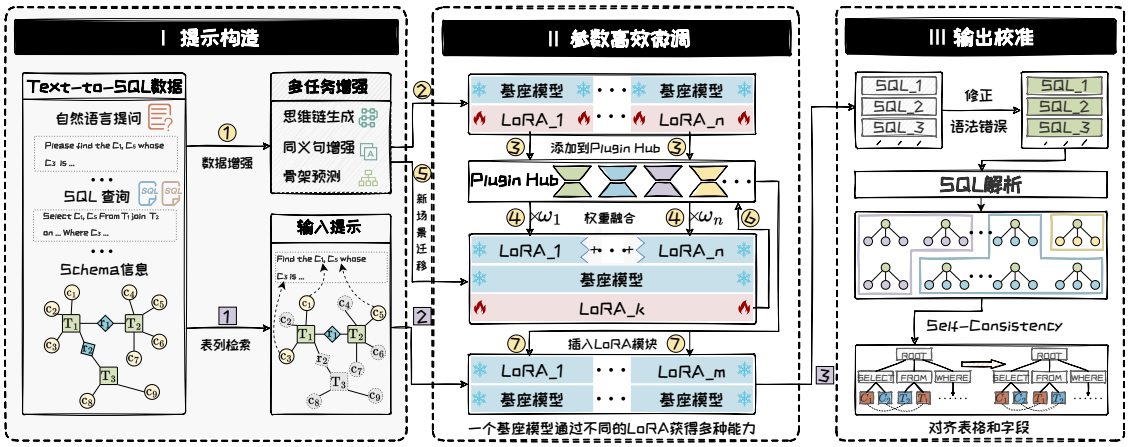


图 4.10: FinSQL 示意图。

相比于基线方法，不仅显著提升了准确性和效率，还展现了在处理复杂金融查询时的强大能力，为金融领域的数据分析和决策支持提供了强有力的技术支撑。

2. 表格数据分析

完成数据查询操作后，可以对查询到的数据进行进一步分析。但是，表格数据的特点——缺乏局部性、包含多种数据类型和相对较少的特征数量——使得传统的深度学习方法难以直接应用。大语言模型的参数中编码了大量先验知识，有效利用这些知识能够弥补表格数据特征不足的问题。但是，为了取得更好的性能，通常依旧需要少量标记数据使大模型适应表格任务。然而，在少量数据上进行模型微调容易导致过拟合。由于 PEFT 只微调部分参数，能有效降低过拟合的风险，使得大语言模型在表格数据上的性能更加稳健。

例如，TabLLM [17] 提出基于大语言模型的少样本表格数据分类框架，图 4.11 为该方法的框架图。该框架将表格数据序列化为自然语言字符串，并附上分类问题的简短描述来提示大语言模型。在少样本设置中，使用 LoRA 在一些带标签的样本对大语言模型进行微调。微调后，TabLLM 在多个基准数据集上超过基于深度学习的表格分类基线方法。此外，在少样本设置下，TabLLM 的性能超过了梯度提升树等强大的传统基线，表现出了强大的小样本学习能力。

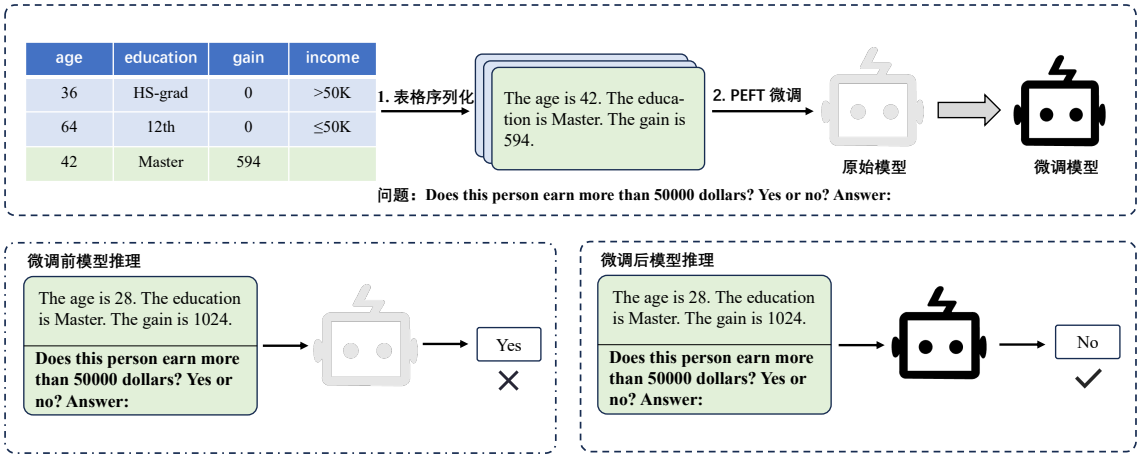


图 4.11: TabLLM 框架图。

本节介绍了参数高效微调技术的实践与应用。首先，我们介绍了 PEFT 主流框架 HF-PEFT 及其使用方法，并介绍了 PEFT 的相关技巧。最后，展示了 PEFT 技术如何帮助大语言模型提升在表格数据的查询与分析任务上的性能，使大语言模型适配垂域任务的同时保证训练效率。

参考文献

[1] Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. “Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning”. In: *ACL/IJCNLP*. 2021.

[2] Alan Ansell et al. “Composable Sparse Fine-Tuning for Cross-Lingual Transfer”. In: *ACL*. 2022.

[3] Daniel Bershatsky et al. “LoTR: Low Tensor Rank Weight Adaptation”. In: *arXiv preprint arXiv:2402.01376* (2024).

[4] Dan Biderman et al. “LoRA Learns Less and Forgets Less”. In: *arXiv preprint arXiv.2405.09673* (2024).

[5] Sarkar Snigdha Sarathi Das et al. “Unified Low-Resource Sequence Labeling by Sample-Aware Dynamic Sparse Finetuning”. In: *EMNLP*. 2023.

[6] Ning Ding et al. “Parameter-efficient fine-tuning of large-scale pre-trained language models”. In: *Nat. Mac. Intell.* 5.3 (2023), pp. 220–235.

[7] Ning Ding et al. “Sparse Low-rank Adaptation of Pre-trained Language Models”. In: *EMNLP*. 2023.

[8] Qingxiu Dong et al. “A Survey for In-context Learning”. In: *CoRR* abs/2301.00234 (2023).

[9] Ali Edalati et al. “KronA: Parameter Efficient Tuning with Kronecker Adapter”. In: *arXiv preprint arXiv:2212.10650* (2022).

[10] Vlad Fomenko et al. “A Note on LoRA”. In: *arXiv preprint arXiv:2404.05086* (2024).

[11] Jonathan Frankle and Michael Carbin. “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks”. In: *ICLR*. 2019.

[12] Zihao Fu et al. “On the Effectiveness of Parameter-Efficient Fine-Tuning”. In: *AAAI*. 2023.

[13] Andi Han et al. “SLTrain: a sparse plus low-rank approach for parameter and memory efficient pretraining”. In: *arXiv preprint arXiv:2406.02214* (2024).

[14] Junxian He et al. “Towards a Unified View of Parameter-Efficient Transfer Learning”. In: *ICLR*. 2022.

[15] Shwai He et al. “SparseAdapter: An Easy Approach for Improving the Parameter-Efficiency of Adapters”. In: *Findings of EMNLP*. 2022.

[16] Xuehai He et al. “Parameter-Efficient Model Adaptation for Vision Transformers”. In: *AAAI*. 2023.

[17] Stefan Hegselmann et al. “TabLLM: Few-shot Classification of Tabular Data with Large Language Models”. In: *AISTATS*. 2023.

[18] Neil Houlsby et al. “Parameter-Efficient Transfer Learning for NLP”. In: *ICML*. 2019.

[19] Edward J. Hu et al. “LoRA: Low-Rank Adaptation of Large Language Models”. In: *ICLR*. 2022.

[20] Chengsong Huang et al. “LoraHub: Efficient Cross-Task Generalization via Dynamic LoRA Composition”. In: *arXiv preprint arXiv:2307.13269* (2023).

[21] Ting Jiang et al. “MoRA: High-Rank Updating for Parameter-Efficient Fine-Tuning”. In: *arXiv preprint arXiv:2405.12130* (2024).

[22] Jaejun Lee, Raphael Tang, and Jimmy Lin. “What Would Elsa Do? Freezing Layers During Transformer Fine-Tuning”. In: *arXiv preprint arXiv:1911.03090* (2019).

[23] Brian Lester, Rami Al-Rfou, and Noah Constant. “The Power of Scale for Parameter-Efficient Prompt Tuning”. In: *EMNLP*. 2021, pp. 3045–3059.

[24] Xiang Lisa Li and Percy Liang. “Prefix-Tuning: Optimizing Continuous Prompts for Generation”. In: *ACL*. 2021.

[25] Vladislav Lialin et al. “ReLoRA: High-Rank Training Through Low-Rank Updates”. In: *NIPS Workshop*. 2023.

[26] Baohao Liao, Yan Meng, and Christof Monz. “Parameter-Efficient Fine-Tuning without Introducing New Latency”. In: *ACL*. 2023.

[27] Alisa Liu et al. “Tuning Language Models by Proxy”. In: *arXiv preprint arXiv:2401.08565* (2024).

[28] Jialin Liu et al. “Versatile black-box optimization”. In: *GECCO*. 2020, pp. 620–628.

[29] Shih-Yang Liu et al. “DoRA: Weight-Decomposed Low-Rank Adaptation”. In: *arXiv preprint arXiv:2402.09353* (2024).

[30] Yulong Mao et al. “DoRA: Enhancing Parameter-Efficient Fine-Tuning with Dynamic Rank Distribution”. In: *arXiv preprint arXiv:2405.17357* (2024).

[31] Yuren Mao et al. *A Survey on LoRA of Large Language Models*. 2024. arXiv: 2407.11046 [cs.LG]. URL: https://arxiv.org/abs/2407.11046.

[32] Fanxu Meng, Zhaohui Wang, and Muhan Zhang. “Pissa: Principal singular values and singular vectors adaptation of large language models”. In: *arXiv preprint arXiv:2404.02948* (2024).

[33] Jinjie Ni et al. *Instruction in the Wild: A User-based Instruction Dataset*. https://github.com/XueFuzhao/InstructionWild. 2023.

[34] Rui Pan et al. “LISA: Layerwise Importance Sampling for Memory-Efficient Large Language Model Fine-Tuning”. In: *arXiv preprint arXiv.2403.17919* (2024).

[35] Jonas Pfeiffer et al. “AdapterFusion: Non-Destructive Task Composition for Transfer Learning”. In: *EACL*. Ed. by Paola Merlo, Jörg Tiedemann, and Reut Tsarfaty. 2021.

[36] Pengjie Ren et al. “Mini-Ensemble Low-Rank Adapters for Parameter-Efficient Fine-Tuning”. In: *arXiv preprint arXiv.2402.17263* (2024).

[37] Victor Sanh et al. *Multitask Prompted Training Enables Zero-Shot Task Generalization*. 2021.

[38] Ying Sheng et al. “S-LoRA: Serving Thousands of Concurrent LoRA Adapters”. In: *arXiv preprint arXiv:2311.03285* (2023).

[39] Yi-Lin Sung, Varun Nair, and Colin Raffel. “Training Neural Networks with Fixed Sparse Masks”. In: *NIPS*. 2021.

[40] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).

[41] Mojtaba Valipour et al. “Dylora: Parameter efficient tuning of pre-trained models using dynamic search-free low-rank adaptation”. In: *arXiv preprint arXiv:2210.07558* (2022).

[42] Mojtaba Valipour et al. “DyLoRA: Parameter-Efficient Tuning of Pre-trained Models using Dynamic Search-Free Low-Rank Adaptation”. In: *EACL*. 2023.

[43] Zhongwei Wan et al. “Efficient Large Language Models: A Survey”. In: *arXiv preprint arXiv:2312.03863* (2023).

[44] Alex Wang et al. “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding”. In: *ICLR*. 2019.

[45] Hanqing Wang et al. “MiLoRA: Harnessing Minor Singular Components for Parameter-Efficient LLM Finetuning”. In: *arXiv preprint arXiv:2406.09044* (2024).

[46] Yizhong Wang et al. “Self-Instruct: Aligning Language Models with Self-Generated Instructions”. In: *ACL*. 2023.

[47] Jason Wei et al. “Finetuned language models are zero-shot learners”. In: *arXiv preprint arXiv:2109.01652* (2021).

[48] Wenhan Xia, Chengwei Qin, and Elad Hazan. “Chain of LoRA: Efficient Fine-tuning of Language Models via Residual Learning”. In: *arXiv preprint arXiv.2401.04151* (2024).

[49] Runxin Xu et al. “Raise a Child in Large Language Model: Towards Effective and Generalizable Fine-tuning”. In: *EMNLP*. 2021.

[50] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. “BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models”. In: *ACL*. 2022.

[51] Chao Zhang et al. “FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis”. In: *SIGMOD*. 2024.

[52] Qingru Zhang et al. “Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning”. In: *ICLR*. 2023.

[53] Shengyu Zhang et al. “Instruction Tuning for Large Language Models: A Survey”. In: *arXiv preprint arXiv:2308.10792* (2023).

[54] Jiawei Zhao et al. “GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection”. In: *arXiv preprint arXiv.2403.03507* (2024).

[55] Mengjie Zhao et al. “Masking as an Efficient Alternative to Finetuning for Pretrained Language Models”. In: *EMNLP*. 2020, pp. 2226–2241.

[56] Yaoming Zhu et al. “Counter-Interference Adapter for Multilingual Machine Translation”. In: *Findings of EMNLP*. 2021.

ZHEJIANG UNIVERSITY
1897

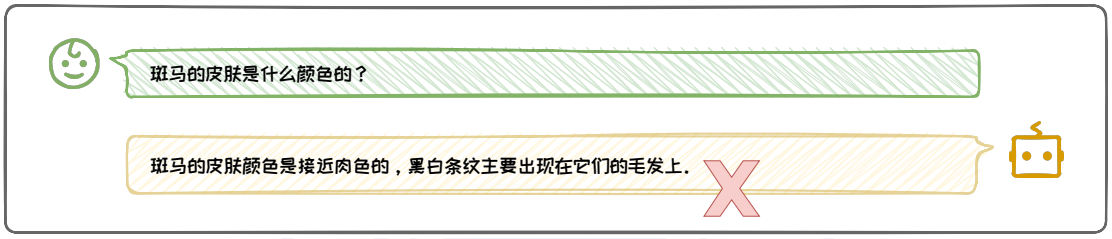
5 模型编辑

预训练大语言模型中，可能存在**偏见**、**毒性**、**知识错误**等问题。为了纠正这些问题，可以将大语言模型“回炉重造”——用清洗过的数据重新进行预训练，但成本过高，舍本逐末。此外，也可对大语言模型“继续教育”——利用高效微调技术向大语言模型注入新知识，但因为新知识相关样本有限，容易诱发过拟合和灾难性遗忘，得不偿失。为此，仅对模型中的特定**知识点**进行修正的**模型编辑**技术应运而生。本章将介绍模型编辑这一新兴技术，首先介绍模型编辑思想、定义、性质，其次从内外两个角度分别介绍模型编辑经典方法，然后举例介绍模型编辑的具体方法 T-Patcher 和 ROME，最后介绍模型编辑的实际应用。

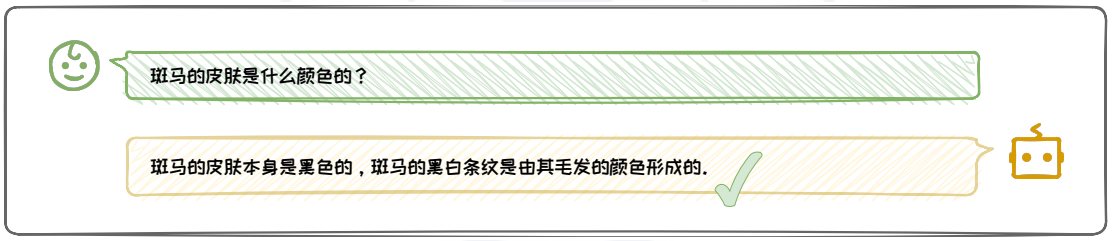
* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

5.1 模型编辑简介

大语言模型有时会产生一些不符合人们期望的结果，如偏见、毒性和知识错误等。**偏见**是指模型生成的内容中包含刻板印象和社会偏见等不公正的观点，**毒性**是指模型生成的内容中包含有害成分，而**知识错误**则是指模型提供的信息与事实不符。例如，当被问到“斑马的皮肤是什么颜色的?”时，ChatGPT 错误地回答“肉色”，而实际上斑马的皮肤是黑色的，这就是一个知识错误，如图5.1。如果不及时修正这些问题，可能会对人们造成严重误导。



(a) 编辑前模型。



(b) 编辑后模型。

图 5.1: 大语言模型知识错误示例。

为了纠正这些问题，可以考虑重新预训练和微调两种方法。**重新预训练**是指使用矫正了偏见、祛除了毒性、纠正了知识错误的清洗后的数据对模型进行重新预训练。这种方法能够从根本上修复模型的错误输出。然而，清洗数据成本高昂，并且由于知识可能频繁更新，无法保证清洗过的数据永远是是完美的；而且，重新进行预训练需要消耗巨大的计算资源，如果每次发现模型错误时都对其重新预训练，未免舍本逐末。**微调**则是在预训练模型的基础上，针对其错误进一步调整模型

参数。尽管存在多种参数高效微调方法，但直接去调整十亿级参数量的模型，还是会产生较高的训练成本；此外，由于修改模型错误所需的新知识样本有限，因此直接用相应的知识点相关的样本微调模型容易导致过拟合和灾难性遗忘。因此，重新预训练和微调都不适用于实际大语言模型的偏见矫正、毒性祛除、以及知识错误纠正。

为规避重新预训练和微调方法的缺点，**模型编辑**应运而生。其旨在精准、高效地修正大语言模型中的特定知识点，能够满足大语言模型对特定知识点进行更新的需求。本节将从思想、定义、性质等方面对模型编辑进行初步介绍。

5.1.1 模型编辑思想

在《三体 2：黑暗森林》中，面壁者希恩斯和他的妻子共同研发了一种名为“思想钢印”的设备，目的是向太空军灌输“人类必胜”的坚定信念。这个机器的原理是让接受者在接触到特定信息时，修改大脑处理过程，使之输出正向答案。模型编辑的思想大致与此相似，旨在通过增加或修改模型参数，快速有效地改变模型行为和输出。

模型学习知识的过程还可以与人类学习知识的过程相对应。首先，在体验世界的过程中，我们能够接触到海量的知识，从而形成自身的知识体系，这可以类比成大语言模型的**预训练**过程。其次，我们可以选择针对不同学科进行专门的学习，提升自己在特定领域的能力，这可以类比成大语言模型的**微调**过程。此外，在与他人交流的过程中，我们会针对特定的知识进行探讨，从而纠正自己在该知识点上的错误认知，这就可以类比成**模型编辑**的思想。

以上方式，都可以满足“纠正大语言模型”的需求。与重新预训练和微调方法不同的是，模型编辑期望能更加**快速**、**精准**地实现对于模型特定知识点的修正。

5.1.2 模型编辑定义

当前，模型编辑领域尚缺乏统一标准，不同研究对相关概念的定义各不相同。本书将不同工作中提到的**基于知识的模型编辑**（KME, Knowledge Model Editing）[25] 和**知识编辑**（KE, Knowledge Editing）[28] 等概念统一为**模型编辑**（ME, Model Editing）。此外，有些研究用“编辑”（edit）[27] 或“事实”（fact）[17, 18] 来表示具体的编辑对象，本书将这些概念统一为“知识点”。

模型编辑的目标可被归纳为：**修正大语言模型使其输出期望结果，同时不影响其他无关输出**。本书将模型编辑定义如下：

定义 5.1 (模型编辑)

将编辑前模型定义为 M，编辑后模型定义为 M^*。每一次编辑都修改模型的一个知识点 k，知识点 k 由问题 x_k 及其对应的答案 y_k 组成。那么，模型编辑的目标可以表示为以下函数：

$$M^*(x) = \begin{cases} y_k, & \text{若} x = x_k \text{或} x \text{与} x_k \text{相关}, \\ M(x), & \text{若} x \text{与} x_k \text{无关}。 \end{cases}$$

♣

上述定义中有关“相关”和“无关”的判断，涉及到模型编辑的范围问题，本书将在第 5.1.3小节讨论。

图5.2用“斑马皮肤颜色”这一知识点作为示例，展示了模型编辑的概念。在这个示例中，当被询问“斑马的皮肤是什么颜色的?”时，编辑前模型回答了错误答案“肉色”，而编辑后的模型可以回答出正确答案“黑色”。

然而，实际的模型编辑过程远比理论定义复杂。这主要源于知识的内在关联性：当修改模型对某一特定知识点的认知时，由于该知识点可能与其它知识点相关联，所以可能会影响模型对其它相关知识点的理解，从而产生”牵一发而动全身”的效应。因此，**如何精确控制模型编辑的范围**成为一个关键挑战。精准可控的模型

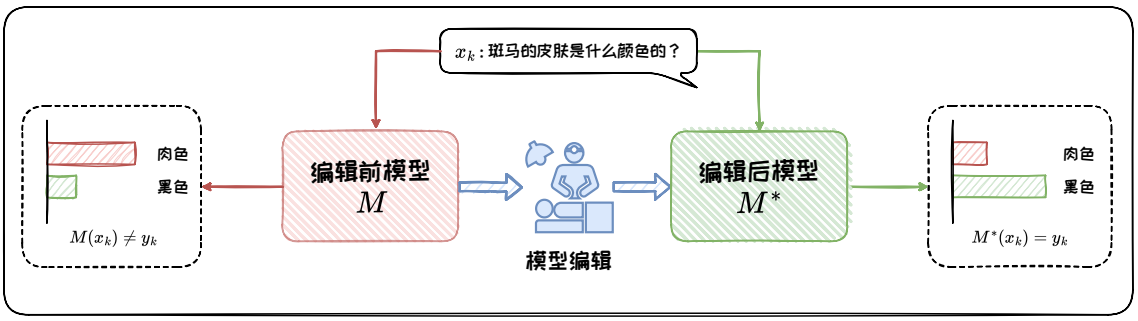


图 5.2: 模型编辑概念图。

编辑技术需要满足一系列性质。这些性质不仅反映了模型编辑的复杂性，也为评估和改进编辑方法提供了重要指标。接下来对模型编辑的关键性质进行介绍。

5.1.3 模型编辑性质

模型编辑的首要目标是纠正模型的错误回答，使其给出我们期望的答案。在此基础上，考虑到知识的内在关联性，需要进一步精准控制模型编辑的范围。除此之外，还要保证模型编辑的效率。因此，需要从多个方面控制模型编辑过程。

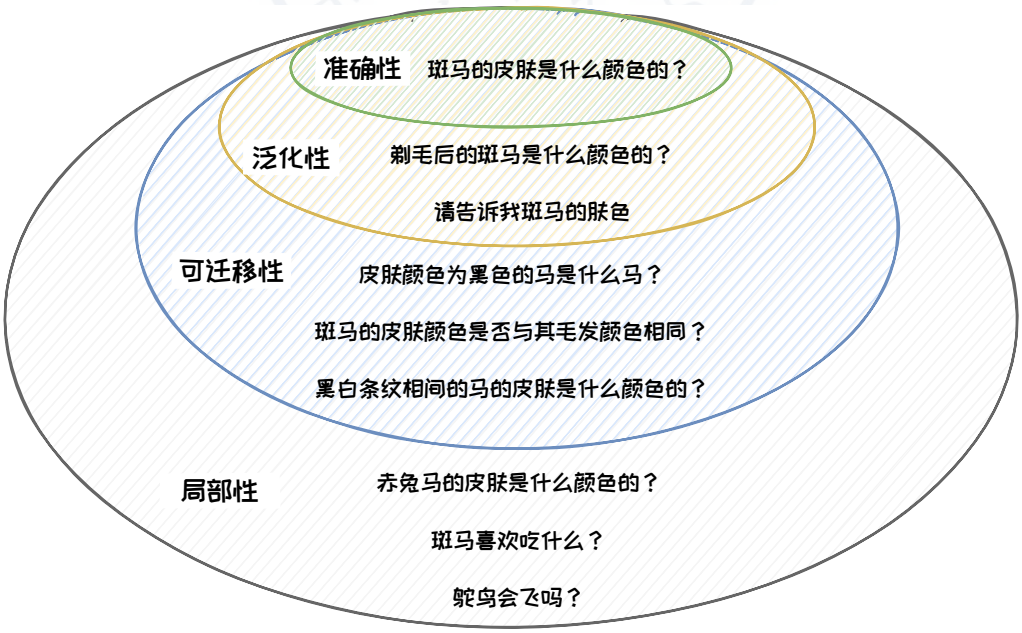


图 5.3: 模型编辑性质关系图。

本书根据已有工作 [16, 25, 27, 28]，将模型编辑的性质归纳为五个方面，分别为**准确性（Accuracy）**、**泛化性（Generality）**、**可迁移性（Portability）**、**局部性（Locality）和高效性（Efficiency）**。图5.3通过相关示例直观地展示了这些性质之间的关系。

1. 准确性

准确性衡量对某个知识点 k 的直接修改是否有效。前面提到，在进行模型编辑时，许多方法通常会选取知识点 k 中的一对代表性输入输出 (x_k, y_k) 对模型进行直接修改。因此，在评估编辑方法时，首先需要评估编辑后模型 M^* 能否准确回答问题 x_k，也就是图5.3中“斑马的皮肤是什么颜色的?”这个问题。如果 M^* 可以输出 y_k，则认为本次编辑准确。采用以下公式来表示一次编辑是否准确：

$$\text{Acc} = I(M^*(x_k) = y_k)。 \tag{5.1}$$

该公式通过一个指示函数 $I(\cdot)$ 进行计算，仅当编辑后模型在目标问题 x_k 上的输出 $M^*(x_k)$ 与答案 y_k 相匹配时，Acc 的值为 1，否则为 0。对于多次编辑，可采用均值来计算平均准确率。**准确性是模型编辑的最基本要求**，它确保了编辑后的模型能够符合设计者的预期，准确地执行特定的任务。只有当模型能够满足准确性时，才能够进一步去满足其它性质。

2. 泛化性

泛化性用来衡量编辑后模型能否适应目标问题 x_k 的其他表达形式，即判断模型在面对与 x_k 具有语义相似性的问题时，能否给出统一的目标答案 y_k。泛化性问题对应图5.3中黄色部分的问题“剃毛后的斑马是什么颜色的?”和“请告诉我斑马的肤色”，这两个问题的正确答案都是“黑色”。

为了评估编辑后模型的泛化性，研究者通常会构造一个**泛化性数据集** $D_G = \{(x_i, y_k)\}_{i=1}^{|D_G|}$，其中 x_i 是与 x_k 具有语义相似性的问题，它们的答案都为 y_k。采用

以下公式来量化编辑后模型的泛化性：

$$\text{Gen} = \frac{1}{|D_G|}\sum_{i=1}^{|D_G|} I(M^*(x_i) = y_k)。 \tag{5.2}$$

当 Gen 的值为 1 时，说明编辑后模型能够正确回答 D_G 中的所有问题，此时泛化性最好。**确保编辑模型的泛化性可以防止编辑后模型过度拟合特定的输入**，从而保证模型对于知识点的理解。

3. 可迁移性

可迁移性是指编辑后模型将特定知识点 k 迁移到其它相关问题上的能力。为了评估编辑后模型的可迁移性，最重要的是**构建可迁移性数据集**，该数据集可用于评估模型对与 k 间接相关的问题的适应能力。

将可迁移性数据集表示为 $D_P = \{(x_i, y_i)\}_{i=1}^{|D_P|}$，其中 x_i 是与 x_k 相关但答案不同的问题，y_i 为对应答案。x_i 可以表达为多种形式，包括反向问题、推理问题和实体替换问题等。如图5.3蓝色部分的问题所示，“皮肤颜色为黑色的马是什么马?” 是一个反向问题，“斑马的皮肤颜色是否与其毛发颜色相同?” 是一个推理问题，“黑白条纹相间的马的皮肤是什么颜色的?” 则是一个实体替换问题，这三个问题的答案都不是 “黑色”。可迁移性数据集 D_P 中的问题与泛化性数据集 D_G 中的问题不重叠，且问题答案不同，即 $y_i \neq y_k$。采用以下公式来量化编辑后模型的可迁移性：

$$\text{Port} = \frac{1}{|D_P|}\sum_{i=1}^{|D_P|} I(M^*(x_i) = y_i)。 \tag{5.3}$$

当 Port 的值为 1 时，说明模型可以正确回答可迁移性数据集中的所有问题，此时模型的可迁移性最好。**可迁移性考察了模型在编辑知识点上的迁移能力**，对于模型编辑的实际应用至关重要。然而，大多数模型编辑方法往往无法实现较好的可迁移性，这也是模型编辑中的一个挑战。

4. 局部性

局部性要求编辑后的模型不影响其他不相关问题的输出。局部性问题对应

图5.3中灰色部分的问题“赤兔马的皮肤是什么颜色的？”、“斑马喜欢吃什么？”等等。一般来说，研究者会在常用的常识数据集中选择一些与知识点 k 不相关的问题作为局部性评估。将局部性数据集定义为 $D_L = \{(x_i, M(x_i))\}_{i=1}^{|D_L|}$，其中 x_i 是与 x_k 无关的问题。采用以下公式来量化编辑后模型的局部性：

$$\text{Loc} = \frac{1}{|D_L|} \sum_{i=1}^{|D_L|} I(M^*(x_i) = M(x_i))。 \tag{5.4}$$

当 Loc 的值为 1 时，编辑后模型的局部性最好，此时，编辑后模型对局部性数据集中所有问题的回答与编辑前一致。**确保局部性能够降低模型编辑的副作用**，是模型编辑相较于朴素微调的重要改进。

5. 高效性

高效性主要考虑模型编辑的时间成本和资源消耗。在实际应用中，模型可能需要频繁地进行更新和纠错，这就要求编辑过程必须足够快速且资源友好。此外，对于大量的编辑任务，不同方法的处理效率也有所不同，有的方法支持批量并行编辑 [2, 6, 18–20]，有的则需要依次进行 [4, 13, 17]。**高效性直接影响到模型编辑的可行性和实用性**。

在评估模型编辑方法时，需要在这五个性质之间寻找平衡。理想的编辑方法应当在保证准确性的基础上，尽可能地提高泛化性、可迁移性和局部性，同时保持高效性。

5.1.4 常用数据集

前文探讨了模型编辑的定义与性质。接下来，我们将介绍一些先前研究中使用过的具体数据集，这些数据集可用于测试和比较不同的模型编辑方法。

在模型编辑的相关研究中，使用最广泛的是由 Omer Levy 等人提出的 **zsRE 数据集** [14]。zsRE 是一个问答任务的数据集，通过众包模板问题来评估模型对于特定关系（如实体间的“出生地”或“职业”等联系）的编辑能力。在模型编辑中，

zsRE 数据集用于检查模型能否准确识别文本中的关系，以及能否根据新输入更新相关知识，从而评估模型编辑方法的**准确性**。

表 5.1: 模型编辑相关数据集总结表格。

数据集	类型	训练集数量	测试集数量	输入	输出
zsRE[14]	知识关联	244,173	244,173	事实陈述	对象
COUNTERFACT[17]	知识关联	N/A	21,919	事实问题	对象
WikiGen[19]	文本生成	N/A	68,000	Wiki 段落	续写
T-REx-100/-1000[6]	知识关联	N/A	100/1,000	事实陈述	对象
ParaRel[4]	知识关联	N/A	253,448	事实问题	对象
NQ-SituatedQA[5]	问答	N/A	67,000	用户查询	答案
MQuAKE-CF/-T[30]	知识关联	N/A	9,218/1,825	多跳问题	对象
Hallucination[10]	幻觉	N/A	1,392	传记	传记
MMEdit-E-VQA[3]	多模态	6,346	2,093	图像问题	答案
MMEdit-E-IC[3]	多模态	2,849	1,000	图像描述	描述
ECBD[23]	知识关联	N/A	1,000	实体完成	引用实体
FEVER[2]	事实检查	104,966	10,444	事实描述	二进制标签
ConvSent[20]	情感分析	287,802	15,989	主题意见	情感
Bias in Bio[11]	人物传记	5,000	5,000	传记句子	职业
VitaminC-FC[20]	事实检查	370,653	55,197	事实描述	二进制标签
SCOTUS[10]	分类	7,400	931	法庭文件	争议主题

2023 年，[17] 提出了 **COUNTERFACT 数据集**，被后续工作广泛采用。COUNTERFACT 被设计用来区分两种类型的知识修改：一种是词汇的表面变化，也就是文本中单纯的词语替换或结构调整，不会影响信息的实质内容；另一种是对基础事实知识显著且泛化的修改，也就是对文本中所描述的事实或信息进行根本性的

改变，这通常需要更深层次的理解和处理能力。COUNTERFACT 能够评估模型对编辑后知识的理解和反应，进而衡量模型的**泛化性**和**局部性**。文献 [27] 在 ZsRE 和 COUNTERFACT 数据集的基础上，使用 GPT-4 生成相应问题的反向问题、推理问题和实体替换问题，构造了**可迁移性**数据集。

此外，研究者还针对不同领域和任务开发了专门的数据集，如 **Hallucination**[10] 用于纠正 GPT 语言模型中的幻觉，**ConvSent**[20] 用于评估模型在修改对话代理对特定主题的情感时的效果。表 5.1从数据类型、样本数量以及输入输出形式方面总结了一些模型编辑相关数据集。这些数据集涵盖了从事实检查、知识关联到特定领域等多种类型，体现了模型编辑技术在不同场景中的应用潜能。

本节介绍了模型编辑的定义、性质、评估方法以及常用数据集，对模型编辑任务做出了详细的解释和说明。接下来，第 5.2节将对模型编辑方法进行系统性概述，将其分为外部拓展法和内部修改法，并介绍每类方法的代表性工作。第 5.3节和第 5.4节将分别深入探讨外部拓展法中的 T-Patcher 方法和内部修改法中的 ROME 方法，通过这两种代表性方法，帮助读者更加细致地理解模型编辑方法的研究过程。最后，第 5.5节将全面介绍模型编辑的实际应用，并分别举例说明解决思路。

5.2 模型编辑经典方法

冒险游戏中的勇者需要升级时，可以从内外两个方面进行改造。外部改造主要通过置办新的道具和装备，它们能够赋予勇者新的能力，同时保留其原有技能。内部改造则相当于去锻炼自身，通过增加智力、体力、法力等属性，从自我层面获得提升。如果将大语言模型比作冒险游戏中的勇者，那么模型编辑可被看作一种满足“升级”需求的方法，可以分别从内外两个角度来考虑。本文参考已有工作 [16, 25, 27, 28]，将现有编辑方法分为**外部拓展法**和**内部修改法**。概括来说，外部拓展

法通过设计特定的训练程序，使模型在保持原有知识的同时学习新信息。内部修改法通过调整模型内部特定层或神经元，来实现对模型输出的精确控制。图 5.4给出了模型编辑方法的分类：外部拓展法包括**知识缓存法**和**附加参数法**，内部修改法包括**元学习法**和**定位编辑法**。接下来，本节将对每类编辑方法展开介绍。

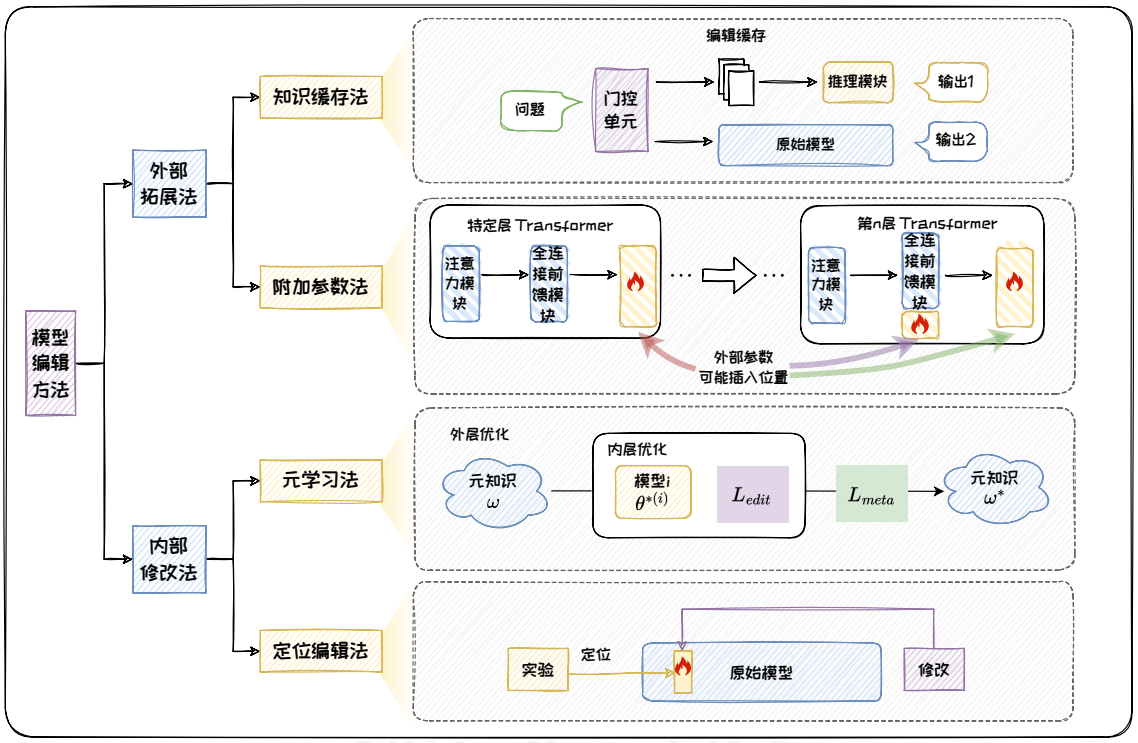


图 5.4: 模型编辑方法分类图。

5.2.1 外部拓展法

外部拓展法的核心思想是将新知识存储在附加的外部参数或外部知识库中，将其和原始模型一起作为编辑后模型。这种方法尤其适合具有良好可扩展性的预训练大语言模型，因为它提供了足够的空间来容纳大量参数，能够存储更多新知识。此外，该方法不会改变原始模型参数，可降低对模型内部预训练知识的干扰。

根据外部组件是否直接整合进模型本身的推理过程，外部拓展法又可划分为**知识缓存法**和**附加参数法**。延续冒险游戏的比喻，知识缓存类似于勇者的技能书，

需要时可以查阅获取特定知识；而附加参数则如同可升级的装备，直接增强勇者的整体能力。

1. 知识缓存法

知识缓存法中包括三个主要组件，分别为门控单元、编辑缓存和推理模块。**编辑缓存**充当一个知识存储库，用于保存需要修改的知识，这些知识由用户通过不同的形式指定。**门控单元**用于判断输入问题与编辑缓存中的知识的相关程度，可通过分类 [20] 或噪声对比估计 [21] 等任务进行训练。**推理模块**获取原始输入问题和编辑缓存中的知识作为输入，通过监督训练的方式学习预测用户期望的结果。

在推理时，门控单元首先判断输入的问题是否与编辑缓存中的某个知识点相关，如果相关，则从编辑缓存中取出该知识点，将其与输入一并交给推理模块，由推理模块给出答案；如果不相关，则用原始模型去推理给出答案。图 5.5为知识缓存法示意图，其中，假设编辑缓存中存储的是与斑马肤色有关的知识，则问题“鸵鸟会飞吗?”由门控单元判断为与编辑缓存中的所有知识点都不相关的问题，因此由原始模型推理出答案；问题“皮肤为黑色的马是什么马?”由门控单元判断为与“斑马的肤色”这个知识点相关的问题，因此由训练好的推理模块给出修改后答案。

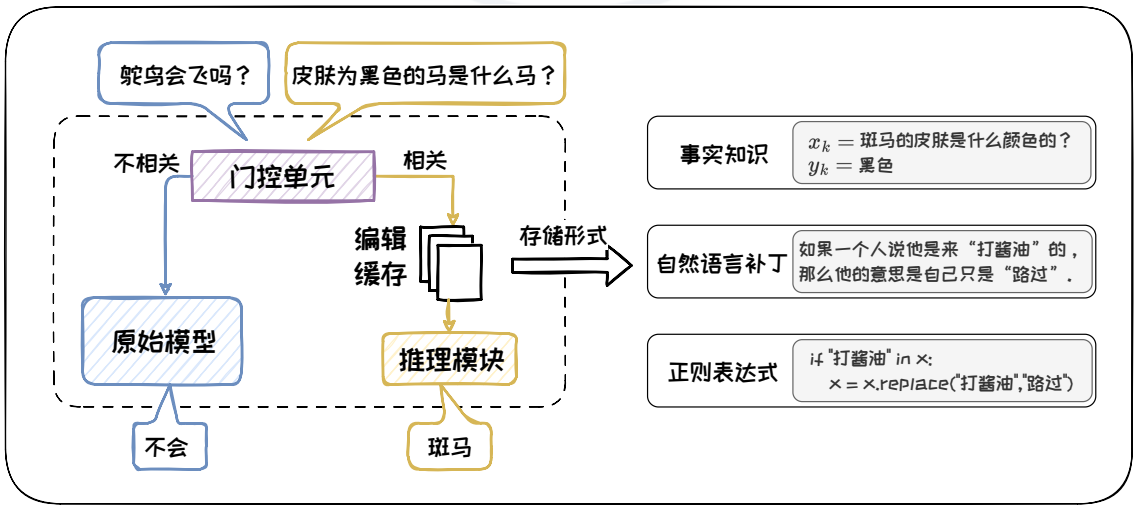


图 5.5: 知识缓存法示意图。

此外，编辑缓存中知识点的存储形式可以分为事实知识、自然语言补丁和正则表达式三种。**事实知识**以问题-答案对 (x_k, y_k) 存储编辑实例，这种存储形式适用于答案明确的事实性问题，**SERAC**[20] 是一种代表性方法。**自然语言补丁 Language Patch**[21] 按照“如果……那么……”的句式描述编辑知识，类似于 Prompt。这种存储形式适用于修正模型对自然语言中非字面含义语句的理解，且便于人们创建、编辑或移除补丁，使得模型能够通过人类反馈不断修正输出。**正则表达式**是一种基于文本匹配和替换的技术，它使用特定的模式来识别和修改文本中的特定部分，适用于精确的文本语义替换。这是一种早期的原始方法，然而由于编写复杂、泛化性低，因此在模型编辑中并不常用。

知识缓存法直接通过编辑缓存中的信息进行检索，不依赖目标标签的梯度信息，因此可以简化模型编辑过程，使其更加高效直接。然而，这种从外界获取知识的方式相当于在让大语言模型进行求助，而并非将新的知识真正内化为自己的一部分。附加参数法对这种局限进行了改良。

2. 附加参数法

与知识缓存法相比，附加参数法可以将外部参数整合进模型结构，从而有效利用和扩展模型的功能。这类方法的思想与参数高效微调中的**参数附加方法**类似，都是将外部参数插入到模型中的特定位置，冻结原始模型，只训练新引入的参数以修正模型输出，如图 5.6。具体而言，不同方法将外部参数插入到模型的不同位置。例如，**CALINET**[6] 和 **T-Patcher**[13] 通过修改模型最后一层 Transformer 的**全连接前馈模块**来实现。CALINET 首先通过一种对比知识评估方法找出原始模型的知识错误，然后在模型最后一个全连接前馈模块添加一个新的参数矩阵，并通过最小化校准数据上的损失来训练新参数，以纠正模型的错误。T-Patcher 与此类似，同样是在原始模型的最后一个全连接前馈模块引入有限数量的可训练神经元，每个神经元对应一个知识点。关于 T-Patcher 的具体介绍将在第 5.3节给出。

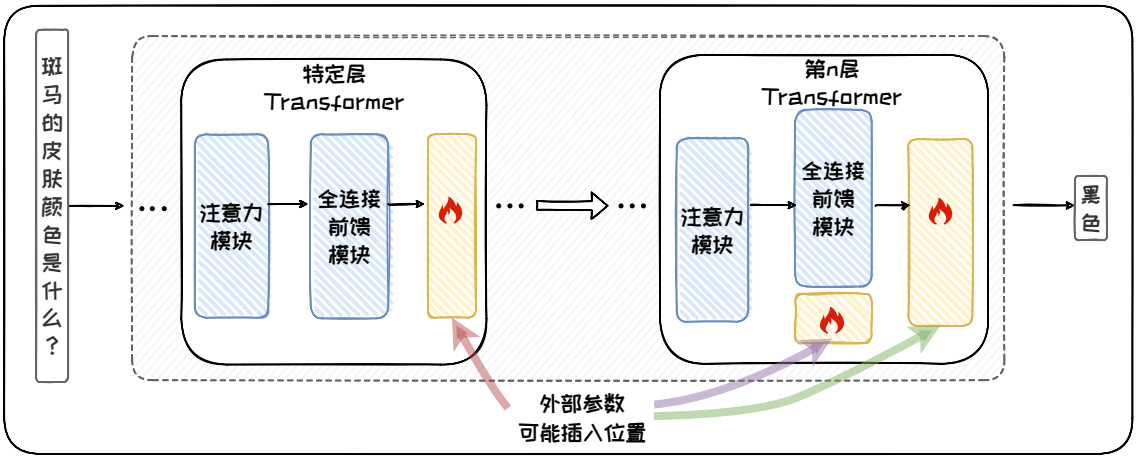


图 5.6: 附加参数法示意图。

GRACE[10] 则将外部参数以适配器的形式插入模型的特定 Transformer 层中，插入位置随模型变化而变化，如 BERT 的倒数第 2 层和 GPT2-XL 的第 36 层。其中，适配器是一个用于缓存错误知识（Keys）和对应的修正值（Values）的键值存储体，被称作“codebook”。在 codebook 中，每个错误知识都有一个对应的修正值，以及一个用于匹配相似输入的延迟半径，且随着时间的推移，codebook 会被持续更新。延迟半径用于判断当前输入是否与 codebook 中的任何错误相似，如果是，则应用相应的修正值进行编辑。

综上，知识缓存法通过引入编辑缓存机制，有效地辅助模型在庞大的知识体系中迅速定位并检索最新信息；附加参数法通过引入额外参数，实现了对模型特定输出的精细调整。这两种方法的核心优势在于**对原始模型的最小化干预**，保证了模型编辑的**局部性**。

然而，外部拓展法的实际有效性在很大程度上取决于对知识的存储与检索能力，这种依赖会导致存储资源需求的增加。因此，在具体应用时，我们需要在**保证模型局部性**和**应对存储限制**之间寻求平衡。

5.2.2 内部修改法

与需要额外存储空间的外部拓展法不同，内部修改法能够让模型在不增加物理存储负担的情况下直接优化自身。**内部修改法旨在通过更新原始模型的内部参数来为模型注入新知识**，能够优化模型的自我学习和适应能力，提高其在特定任务上的表现，而不是仅仅停留在表面的知识积累。内部修改法又可以分为**元学习法**和**定位编辑法**。其中，元学习法通过“学习如何学习”来获取元知识，再基于元知识实现模型编辑；定位编辑法则专注于对模型局部参数的修改，首先识别与目标知识最相关的模型参数，然后仅更新这些特定参数，通过“先定位后编辑”的策略节省更新模型所需成本。

1. 元学习法

元学习指的是模型“学习如何学习”（Learning to Learn）的过程。基于元学习的模型编辑方法，旨在让模型“学习如何编辑”（Learning to Edit），核心思想是使模型从一系列编辑任务中提取通用的知识，并将其应用于未见过的编辑任务，这部分知识被称为**元知识** ω [12]。元知识是模型在进行编辑前可以利用的知识，包括优化器参数 [24]、超网络 [2, 19] 等多种形式。元知识的训练过程被称为**元训练**，其目标是获得一个较好的元知识 ω，使得后续的每次编辑只需少量样本即可快速收敛。

元训练过程可以看作一个双层优化问题 [12]，如公式 5.5所示。双层优化（Bilevel Optimization）是一个层次化的优化框架，其中，内层优化问题可作为外层优化问题的约束。基于元学习的编辑方法如图 5.7所示。图中，内层优化是模型在不同编辑任务上的优化，外层优化是元知识在验证集中的编辑任务上的综合优化。

$$\begin{aligned} \omega^* &= \arg\min_{\omega} \sum_{i=1}^{n} L_{\text{meta}}(\theta^{*(i)}(\omega), \omega, D_k^{\text{val}(i)}) \\ \text{s.t. } \theta^{*(i)}(\omega) &= \arg\min_{\theta} L_{\text{edit}}(\theta^{(i)}, \omega, D_k^{\text{train}(i)})。 \end{aligned} \tag{5.5}$$

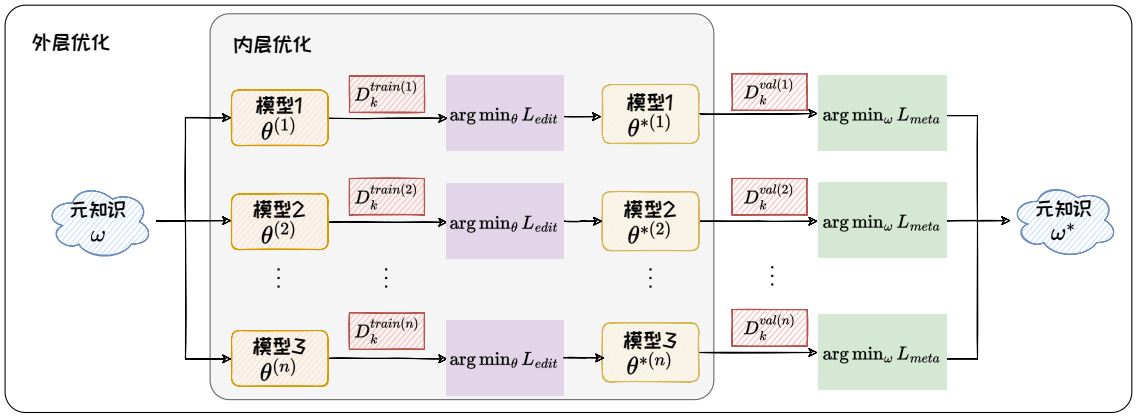


图 5.7: 元学习法示意图。

内层优化问题旨在学习模型在具体编辑任务上的参数。在内层中，设共有 n 个编辑任务，则对于第 i 个编辑 $(i \in [1,n]$ 且 $i \in \mathbb{Z}^+)$，用 $D_k^{(i)}$ 表示该次编辑涉及到的有关知识点 k 的数据集，其中 $D_k^{(i)}$ 是由 (x_k, y_k) 组成的问题-答案对集合，可划分为训练集 $D_k^{\text{train}(i)}$ 和验证集 $D_k^{\text{val}(i)}$。设模型原始参数为 $\theta^{(i)}$，在 $D_k^{(i)}$ 上优化后的参数为 $\theta^{*(i)}$，则内层优化是在元知识 ω 的基础上更新对应编辑任务上的模型参数的过程。在内层优化中，L_{edit} 是有关每次编辑的损失函数，其含义为希望能够最小化编辑后模型 $\theta^{*(i)}$ 在 $D_k^{\text{train}(i)}$ 上的预测误差，如分类任务中的交叉熵函数。

外层优化问题旨在学习可以泛化到其他编辑任务上的元知识。外层优化通常在验证集 $D_k^{\text{val}(i)}$ 上进行，希望能够根据模型 $\theta^{*(i)}$ 在其对应编辑验证集上的损失来更新元知识 ω。外层优化中的 L_{meta} 是有关元学习的损失函数，其含义为希望能够找到一个元知识 ω，使得在该知识的基础上进行优化后的模型在所有 $D_k^{\text{val}(i)}$ 上的预测误差之和最小。

ENN[24] 将元知识看作优化器参数，通过更新优化器参数，使后续编辑中模型参数的训练更为高效，从而使模型学习如何快速编辑。ENN 引入了编辑函数和梯度下降编辑器，在内层优化过程中，每次都在一个编辑任务上对模型参数进行较少次数的梯度更新，再用更新后的参数来更新优化器参数。然而，ENN 是针对小型网络 ResNet 设计的，当应用于大型模型时，会面临训练成本高等问题。

为了拓展元学习法在大型模型架构上的应用，**KE**[2] 将元知识作为超网络，提出了一种通过训练超网络来学习模型参数更新值的方法。在训练超网络时，损失函数由两部分组成，一部分用于确保准确性，另一部分用于确保局部性，并设置了边界值来表示约束的严格程度。训练好的超网络根据输入问题生成模型的参数更新值，使模型能够在特定输入下输出期望的结果，同时保持其他预测不变。为了进一步增强超网络对于大型语言模型的普适性，MEND[19] 通过低秩分解的方法来优化超网络辅助模型参数更新的过程。首先，它利用全连接层中梯度的秩-1 特性，将损失函数关于每一层参数的梯度分解为两个向量的乘积。然后，使用超网络接收分解后的向量作为输入，并输出经过编辑的新向量。最后，这些新向量再次相乘，得到新的参数梯度，并通过一个可学习的缩放因子来计算最终的模型参数更新值。MEND 以较少的参数高效地编辑大型模型，节省了计算资源和内存。

总结来说，基于元学习的编辑方法通过“学习如何编辑”来提高模型在面对新编辑任务时的适应性和泛化能力，能够从一系列编辑任务中提取通用的知识，进而在遇到未见过的编辑任务时，仅使用少量样本训练即可快速收敛，从而节省计算资源和时间。然而，元学习编辑方法也存在不足之处。该方法训练过程较为复杂，应用于大型模型时常常面临训练成本高的问题。尽管 KE 和 MEND 通过使用超网络和梯度低秩分解等技术优化了参数更新过程，减少了计算资源的需求，但仍需进一步提升其对更复杂任务和更大规模模型的适应性与效率。此外，元学习编辑方法从全局视角对模型参数进行更新，即使添加了与局部性相关的损失函数，也有可能会对模型原本的知识产生影响，导致模型的不稳定。

2. 定位编辑法

与元学习法相比，定位编辑法修改的是原始模型的局部参数。其先定位到需要修改的参数的位置，然后对该处的参数进行修改。实现定位前需要了解大语言模型中知识的存储机制。当前，对知识存储机制的探索主要依靠定性实验来完成。通

过在一个 16 层 Transformer 的语言模型上进行实验，可以得到结论：**Transformer 中的全连接前馈模块可以看作存储知识的键值存储体** [8]。其中，全连接前馈模块的输入称为查询（query）向量，代表当前句子的前缀；将全连接前馈模块的上投影矩阵中的向量称为键（key）向量，下投影矩阵中的向量称为值（value）向量。

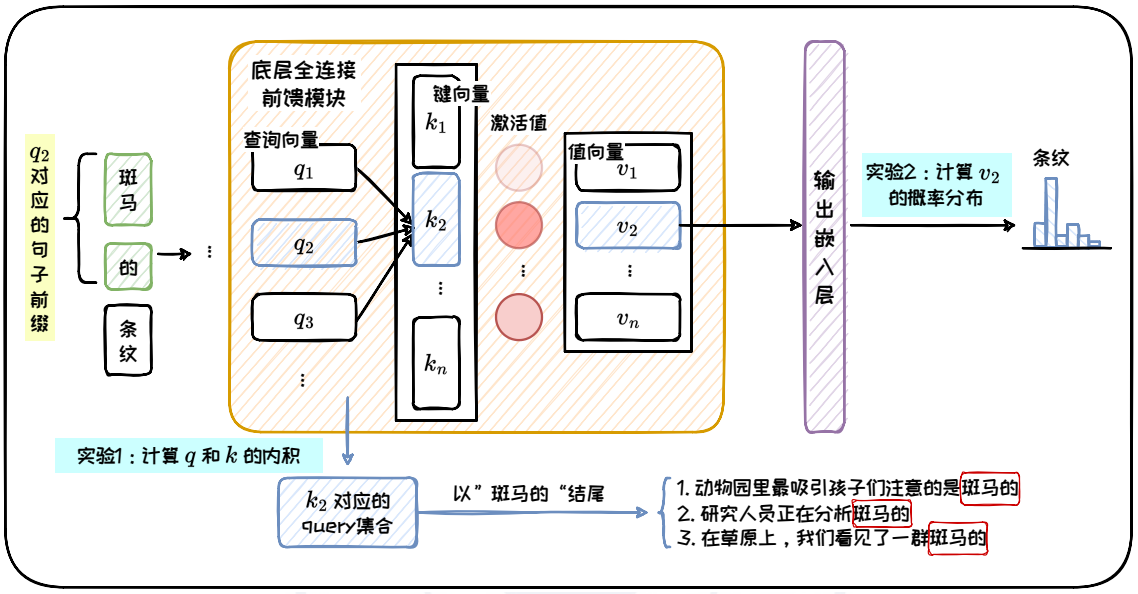


图 5.8: Transformer 可看作键值存储体。

图 5.8展示了文献 [8] 在模型底层全连接前馈模块上进行实验的过程。在实验 1 中，将每个 key 都分别与所有 query($q_1, q_2, q_3, ...$) 做内积运算，图中以 k_2 为例展示了这一过程。将 k_2 与所有 query 进行内积运算后，发现 k_2 与包括 q_2 在内的几个 query 的激活值较大。将每个 key 对应的激活值较大的 query 收集起来作为一个集合，在该集合中观察到这些 query 对应的前缀都有着相同或相似的模式。例如，图中 k_2 对应的 query 都以”斑马的“结尾，也就是说，k_2 存储了与“斑马的”相关的文本模式。在实验 2 中，将每个 key 对应的 value 与输出嵌入层（Output Embedding Layer）矩阵相乘，并应用 softmax 函数转换为概率分布，然后比较这些分布与对应 query 的下一个词的相关性，结果发现二者高度相关。例如，图中将 k_2 对应的 v_2 进行上述操作，发现下一个词为”条纹“的概率最大，这与 q_2 的下一个词相符。

综合实验 1 和实验 2，文献 [8] 指出：在全连接前馈模块中，模型通过 key 总结了当前 query 所代表的句子前缀的特征，又通过类似键值匹配的机制查找到了相应的 value，也就是下一个词的概率分布。也就是说，Transformer 中的全连接前馈模块可以看作键值存储体对知识进行存储。

基于上述结论，**KN**[4] 提出了知识神经元的概念。其将全连接前馈模块的每个中间激活值定义为一个知识神经元，并认为知识神经元的激活与相应知识点的表达密切相关。为了评估每个神经元对于特定知识预测的贡献，KN 将有关该知识点的掩码文本输入给预训练模型，获取隐藏状态后，将其输入到模型每一层的 FFN 中，通过归因方法，积累每个神经元在预测正确答案时的梯度变化，从而确定哪些神经元在知识表达过程中起关键作用。这种方法不仅能够识别出显著影响知识表达的神经元，还能过滤掉那些对知识预测贡献较小的神经元。在确定了知识神经元对于知识预测的贡献后，可以通过直接在模型中修改特定知识神经元对应的键向量，来诱导模型输出的编辑后的知识，从而达到模型编辑的效果。

此外，ROME[17] 设计了一种因果跟踪实验，进一步探索中间层全连接前馈模块与知识的关系，优化了知识存储机制的结论。在编辑方法上，与定位和编辑全连接前馈模块中的单个神经元的 KN 不同，ROME 提出更新整个全连接前馈模块来进行编辑。通过 ROME 对 GPT-J 模型上进行编辑，在准确性、泛化性和局部性方面都有良好表现。因此，ROME 成为近年来备受瞩目的模型编辑方法，为未来的模型编辑和优化工作提供了重要的参考和指导。我们将在第 5.4 节详细介绍 ROME 中对于因果跟踪实验和编辑方法的设计。MEMIT[18] 在 ROME 的基础上扩展到对不同知识的大规模编辑，可以同时执行数千次编辑。

总的来说，定位编辑法修改大语言模型的局部参数，在保持模型整体结构和性能的同时，能够对特定知识点进行精准的更新和编辑。与其他方法相比，定位编辑法可同时保持较高的准确性、泛化性和局部性，且适用于各类模型。

5.2.3 方法比较

上述各种模型编辑方法各有优劣。本书参考文献 [27] 中的实验结果，对主流模型编辑方法在各个性质上的表现进行对比，结果如表 5.2所示。表中用“高”、“中”、“低”三个级别定性表示方法的准确性、泛化性、可迁移性和局部性，用 “✓” 和 “✗” 来表示方法的高效性，即是否支持批量编辑。标注 “-” 的表明未进行测试。

表 5.2: 模型编辑方法比较。

		方法	准确性	泛化性	可迁移性	局部性	高效性
外部拓展法	知识缓存法	SERAC	高	高	低	高	✓
	附加参数法	CaliNET	低	低	-	中	✓
		T-Patcher	高	高	高	中	✗
内部修改法	元学习法	KE	低	低	-	高	✓
		MEND	中	高	中	高	✓
	定位编辑法	KN	中	低	-	中	✗
		ROME	高	高	高	高	✗
		MEMIT	高	高	高	高	✓

从表 5.2中可看出，在外部拓展法中，基于知识缓存的 SERAC 在无需额外训练的情况下提供了高效的编辑能力，保证了高准确性、泛化性和局部性，适合快速响应和批量编辑，但可迁移性较差，编辑缓存和推理模块仍有待优化。基于附加参数的 CaliNET 和 T-Patcher 提供了对模型的直接编辑能力，但 CaliNET 对不同模型和数据的适应性较差，而 T-Patcher 虽然保持了高准确性、泛化性和可迁移性，但在批量编辑时，对内存的需求较高。

在内部修改法中，基于元学习法的 KE 和 MEND 将元知识看作超网络，通过使模型 “学习如何编辑”，提高了泛化性和训练效率，且支持批量编辑。然而，基于元学习的方法训练过程设计较为复杂，在应用于大型模型时可能受限。基于定位编辑的 KN、ROME 和 MEMIT 则专注于精确定位和编辑模型内部的特定知识。

ROME 和 MEMIT 都能保证高准确性、泛化性、可迁移性和局部性，然而这两种方法主要针对 decoder-only 模型设计，因此未比较其在 encoder-decoder 架构模型上的表现。此外，MEMIT 在 ROME 的基础上针对批量编辑进行了优化，在满足高效性的同时依然能够保证其它性质的稳定。

本节对模型编辑领域不同类别的方法进行了概述和举例介绍。这些方法展示了对大语言模型进行有效修正的多种途径，各有优势和局限。在实际应用中，应根据需求、可用资源和期望的编辑效果选取合适的方法。根据表 5.2，T-Patcher 和 ROME 这两种方法在准确性、泛化性、可迁移性和局部性上表现较优。因此，接下来的两节将以这两种方法为代表，更加细粒度地解释模型编辑的内部机制。

5.3 附加参数法：T-Patcher

附加参数法在模型中引入额外的参数，并对这部分参数进行训练以实现特定知识的编辑。其在准确性、泛化性、可迁移性等方面均取得良好的表现。T-Patcher 是附加参数法中的代表性方法，其在模型最后一个 Transformer 层的全连接前馈层中添加额外参数（称为“补丁”），然后对补丁进行训练来完成特定知识的编辑，如图 5.9。T-Patcher 可以在不改变原始模型整体架构的情况下，对模型进行编辑。本节将从**补丁的位置**、**补丁的形式**以及**补丁的实现**三个方面对 T-Patcher 展开介绍。

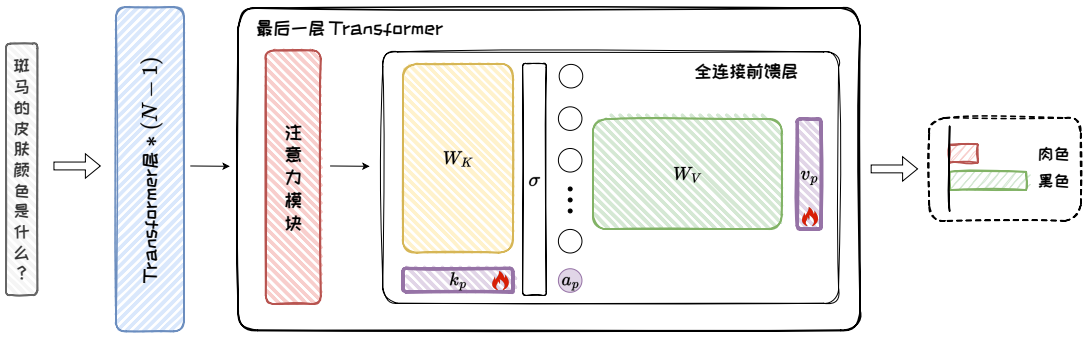


图 5.9: T-Patcher 方法 (图中紫色图块为补丁)。

5.3.1 补丁的位置

将补丁添加在模型中的不同位置会影响模型编辑的效果。T-Patcher 将全连接前馈层视为**键值存储体**，并选择在最后一个 Transformer 层的全连接前馈层中添加补丁参数。在这种设计中，添加补丁相当于向键值存储体中增加新的记忆单元，通过精确控制补丁的激活，T-Patcher 能够针对特定输入进行修正，并减少对无关输入的影响。此外，由于全连接前馈层结构简单，因此只需要添加少量参数即可实现有效编辑。

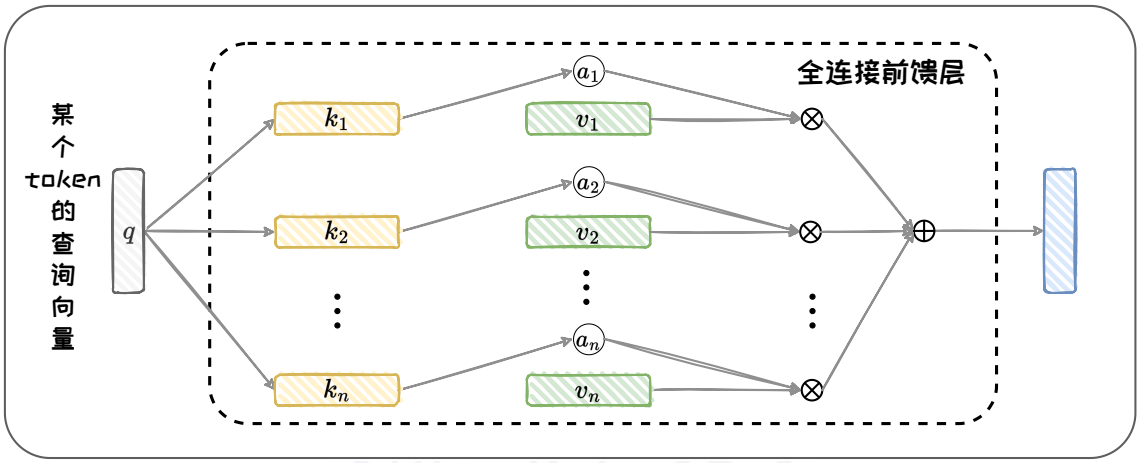


图 5.10: 键值存储体 (省略激活函数)。

具体而言，T-Patcher 将全连接前馈层视为如图5.10所示的键值存储体，键值存储体包含键向量矩阵 $\boldsymbol{W}_{fc} = [\boldsymbol{k_1}, \boldsymbol{k_2}, \ldots, \boldsymbol{k_n}]$ 及其偏置向量 $\boldsymbol{b}_k$、激活函数 σ 和值向量矩阵 $\boldsymbol{W}_{proj} = [\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n}]$ 及其偏置向量 $\boldsymbol{b}_v$。其中，每个键向量对应着输入文本中的特定模式，如 n-gram 或语义主题，而每个值向量则关联着模型输出的概率分布。

在查询存储体中找到相应知识的过程如下：对于某个输入的 Token，其查询向量为 $\boldsymbol{q}$。当 $\boldsymbol{q}$ 输入全连接前馈层时，首先与矩阵 W_{fc} 相乘，计算出激活值向量 $\boldsymbol{a}$，其中每个分量代表 $\boldsymbol{q}$ 与对应键向量 k 的关联程度。随后，$\boldsymbol{a}$ 与矩阵 W_{proj} 相乘，得

到全连接前馈层的输出结果。该过程可以视为使用激活值对矩阵 W_{proj} 中的所有值向量进行加权求和：

$$\boldsymbol{a} = \sigma(\boldsymbol{q} \cdot \boldsymbol{W}_{\boldsymbol{fc}} + \boldsymbol{b}_k)$$

$$FFN(\boldsymbol{q}) = \boldsymbol{a} \cdot \boldsymbol{W}_{\boldsymbol{proj}} + \boldsymbol{b}_v。 \tag{5.6}$$

基于上述键值存储体的观点，**全连接前馈层的隐藏层维度可被理解为其“记忆”的文本模式的数量**，如果能够向全连接前馈层中添加更多与不同文本模式相关联的键值对，就可以向模型中插入新的事实信息，从而实现模型编辑。因此，T-Patcher **在全连接前馈层中增加额外参数，即添加补丁**。而且，T-Patcher 仅在模型的**最后一层添加补丁**，以确保补丁能够充分修改模型的输出，而不被其他模型结构干扰。在下一节中，我们将详细介绍应该添加什么样的补丁。

5.3.2 补丁的形式

基于键值存储体的观点，T-Patcher 将补丁的形式设计为键值对。具体地，T-Patcher 在全连接前馈层中加入额外的键值对向量作为补丁，通过训练补丁参数从而实现模型编辑。补丁的形式如图5.11所示，它主要包括一个键向量 $\boldsymbol{k_p}$、一个值向量 $\boldsymbol{v_p}$ 和一个偏置项 b_p。

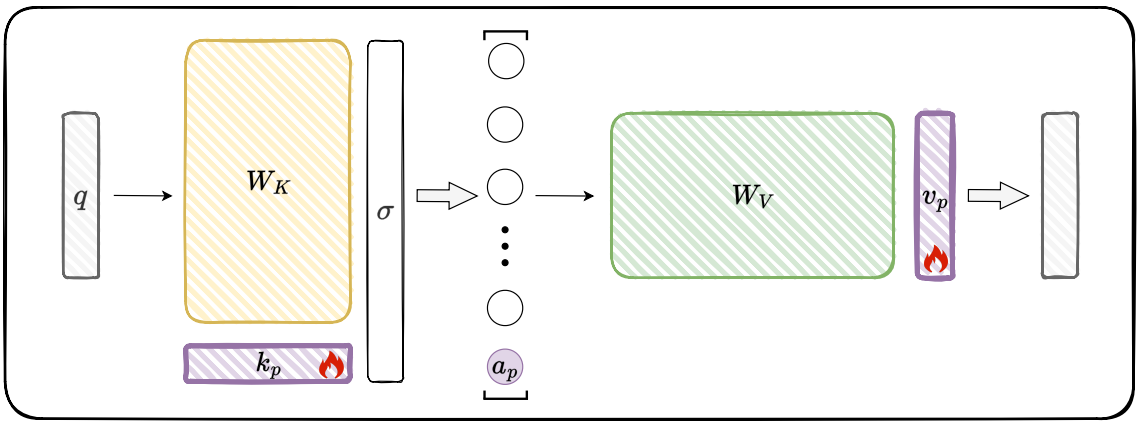


图 5.11: 补丁的形式。

在添加补丁后，全连接前馈层的输出被调整为：

$$[\boldsymbol{a} \quad a_p] = \sigma\left(\boldsymbol{q} \cdot [\boldsymbol{W_{fc}} \quad \boldsymbol{k}_p] + [\boldsymbol{b}_k \quad b_p]\right)$$

$$FFN_p(\boldsymbol{q}) = [\boldsymbol{a} \quad a_p] \cdot [\boldsymbol{W_{proj}} \quad \boldsymbol{v}_p]^\top + \boldsymbol{b}_v = FFN(\boldsymbol{q}) + a_p \cdot \boldsymbol{v}_p, \tag{5.7}$$

其中，a_p 为补丁的激活值，代表补丁对输入查询的响应程度。在添加补丁之后，a_p 与值向量 $\boldsymbol{v}_p$ 之积会形成偏置项叠加到全连接前馈层的原始输出之上，以调整模型的输出。补丁就像是一个很小的修正器，只会被相关的输入查询激活。

5.3.3 补丁的实现

在确定了补丁的位置和形式之后，下一步便是训练补丁以实现模型编辑。本节将深入讨论如何训练这些补丁来有满足模型编辑的主要性质。T-Patcher 冻结模型的原有参数，仅对新添加的补丁参数进行训练。此外，对于给定的编辑问题，T-Patcher 为每个需要编辑的 Token 都添加一个补丁，从而可以精确地针对每个编辑需求进行调整。最后，T-Patcher 从编辑的准确性和局部性两个角度出发对损失函数进行设计。接下来对其损失函数进行介绍。

1. 准确性

确保精确编辑是 T-Patcher 的核心目标之一。在训练过程中，为了强化补丁对模型的正确修改，首先需要针对准确性设计特定的损失函数。对于补丁的准确性，T-Patcher 主要关注两个方面：（1）确保补丁可以在目标输入下可以被激活；（2）一旦被激活，补丁应该能够准确地调整模型输出以符合预期的结果。为此，T-Patcher 设计了**准确性损失** L_{acc}，它包括**激活损失** l_a 和**编辑损失** l_e：

$$L_{Acc} = l_a(\boldsymbol{k}_p, b_p) + \alpha l_e(\boldsymbol{k}_p, \boldsymbol{v}_p, b_p) \tag{5.8}$$

$$l_a(\boldsymbol{k}_p, b_p) = \exp(-\boldsymbol{q}_e \cdot \boldsymbol{k}_p - b_p) \tag{5.9}$$

$$l_e(\boldsymbol{k}_p, \boldsymbol{v}_p, b_p) = CE(y_e, p_e), \tag{5.10}$$

其中，$\boldsymbol{q}_e$ 是编辑样本在全连接前馈层处的查询向量，y_e 是该补丁对应的目标 Token，p_e 是模型在补丁作用下的预测输出，CE 是交叉熵损失函数，α 是激活损失 l_a 的权重。

在准确性损失 L_{Acc} 中，激活损失 l_a 负责确保补丁在目标输入下可以被激活，它通过最大化编辑样本的查询向量 $\boldsymbol{q}_e$ 对补丁的激活值，从而确保修补神经元对特定编辑需求的响应。另一方面，编辑损失 l_e 主要确保补丁在被激活后能够有效地将模型输出调整为该补丁对应的目标 Token。具体而言，T—Patcher 使用交叉熵损失函数作为编辑损失，用于评估补丁调整后的输出 p_e 与该补丁对应的目标 Token y_e 之间的一致性，确保补丁的调整正确实现预期的修正效果。准确性损失是实现 T-Patcher 编辑目标的关键，它确保补丁在必要时被激活，并且激活后能够有效地达到预期的编辑效果。

2. 局部性

模型编辑不仅要确保精确的编辑，而且要求在对目标问题进行编辑时，不应影响模型在其他无关问题上的表现。为了保证编辑的局部性，T-Patcher 设计了特定的损失函数来限制补丁的激活范围，确保其只在相关的输入上被激活。为了模拟无关数据的查询向量分布，以便在训练过程中控制激活范围，T-Patcher 会随机保留一些先前已经处理过的查询向量，组成记忆数据集 $D_M = \{q_i\}_{i=1}^{|D_M|}$，这些查询向量与当前的编辑目标无关。基于该数据集，T-Patcher 定义了**记忆损失** L_m 来保证编辑的局部性，该损失包含 l_{m1} 和 l_{m2} 两项：

$$L_m = l_{m1}(\boldsymbol{k}_p, b_p) + l_{m2}(\boldsymbol{k}_p, b_p, \boldsymbol{q}_e) \tag{5.11}$$

$$l_{m1}(\boldsymbol{k}_p, b_p) = \frac{1}{|D_M|}\sum_{i=1}^{|D_M|}(\boldsymbol{q_i} \cdot \boldsymbol{k}_p + b_p - \beta) \tag{5.12}$$

$$l_{m2}(\boldsymbol{k}_p, b_p) = \frac{1}{|D_M|}\sum_{i=1}^{|D_M|}((\boldsymbol{q_i} - \boldsymbol{q}_e) \cdot \boldsymbol{k}_p + b_p - \gamma), \tag{5.13}$$

其中，q_i 为无关问题的查询向量，β 为指定的激活阈值，γ 为指定的激活值差距的阈值。具体而言，第一项 l_{m1} 负责确保补丁不会对无关的输入进行激活。这通过对每个无关输入的查询向量 q_i 的激活值进行阈值限制实现，若激活值超过阈值 β 则会产生惩罚。而第二项 l_{m2} 旨在放大补丁对目标查询向量 $\boldsymbol{q}_e$ 和无关查询向量 q_i 的激活值差距。这通过要求目标查询向量的激活值显著高于所有无关查询向量的激活值的最大值来实现。

将这些损失项整合，T-Patcher 的总损失函数 L_p 可以表达为：

$$L_p = L_{Acc} + \beta \cdot L_m = l_e + \alpha \cdot l_a + \beta \cdot (l_{m1} + l_{m2}), \tag{5.14}$$

其中，β 是记忆损失项的权重。通过这种设计，T-Patcher 不仅可以确保补丁正确地修改模型的输出，还能减少对其他问题的影响，从而实现准确且可靠的模型编辑。

尽管 T-Patcher 实现了模型的精确调整，在 GPT-J 模型上的准确性和泛化性较好，但是一些研究表明 [27]，它也存在一些局限性。例如，在不同模型架构上，其性能会有所波动；在批量编辑时，对内存的需求较高，可能限制其在资源受限环境下的应用。相比之下，ROME 方法表现得更加稳定。它将知识编辑视为一个带有线性等式约束的最小二乘问题，从而实现对模型特定知识的精确修改，在编辑的准确性、泛化性和局部性等方面都表现出色。该方法将在第5.4节介绍。

5.4 定位编辑法：ROME

定位编辑首先定位知识存储在神经网络中的哪些参数中，然后再针对这些定位到的参数进行精确的编辑。ROME（Rank-One Model Editing）[17] 是其中的代表性方法。本节将详细介绍 ROME 的知识定位过程及相应的编辑方法。

5.4.1 知识存储位置

大脑的记忆存储机制一直是人类探索的谜题。这一问题不仅限于脑科学领域，对于具有强大智能的大语言模型，其知识存储及回忆机制也亟待研究。要解决该问题，首先要定位出大语言模型的知识存储在哪些参数中，即存储的位置。通过对知识进行定位，可以揭示模型内部的运作机制，这是理解和编辑模型的关键步骤。ROME 通过因果跟踪实验和阻断实验发现知识存储于模型中间层的全连接前馈层。

1. 因果跟踪实验

ROME 采用控制变量的策略，首先对模型的推理过程实施干扰，然后进行局部恢复并观察影响，探究模型中不同结构与具体知识在推理过程中的相关性，从而确定知识在模型中的具体位置。该实验被称为**因果跟踪**，包含三个步骤：**正常推理**、**干扰推理**和**恢复推理**。其中，**正常推理**旨在保存模型在未受干扰情况下的内部状态，用于后续恢复推理中内部状态的恢复；**干扰推理**旨在干扰模型的所有内部状态，作为控制变量的基准线；**恢复推理**则将每个内部状态的恢复作为变量，通过对比内部状态恢复前后的输出差异，精确评估每个模块与知识回忆的相关性。

因果跟踪实验针对**知识元组**进行研究。在这个实验中，每个知识被表示为知识元组 $t=(s,r,o)$，其中 s 为 **主体**，r 为 **关系**，o 为 **客体**。例如，“斑马的肤色是黑色”可以表示为知识元组：（“斑马”，“的肤色是”，“黑色”）。此外，模型的输入问题为 $q=(s,r)$，$q^{(i)}$ 表示 q 的第 i 个 Token。我们期望模型在处理问题 q 时能够输出对应的客体 o 作为答案。具体地，因果跟踪实验的步骤如下：

1. **正常推理**：将 q 输入语言模型，让模型预测出 o。在此过程中，保存模型内部的所有模块的正常输出，用于后续恢复操作。

2. **干扰推理**：向 s 部分的嵌入层输出添加噪声，破坏其向量表示。在这种破坏输入的情况下，让模型进行推理，在内部形成被干扰的混乱状态。

3. **恢复推理**：在干扰状态下，对于输入问题的每一个 Token $q^{(i)}$，将 $q^{(i)}$ 在每一层的输出向量分别独立地恢复为未受噪声干扰的“干净”状态，并进行推理。在每次恢复时，仅恢复一个特定位置的输出向量，其余内部输出仍保持干扰状态。之后，记录模型在恢复前后对答案的预测概率增量，该增量被称为模块的**因果效应**，用来评估每个模块对答案的贡献。

以问题“斑马的肤色是”为例，其因果跟踪过程如下：

当输入问题“斑马的肤色是”时，模型会推理出答案“肉色”（假设该模型不知道正确答案是黑色）。此时，保存所有模块在正常推理过程中的输出，见图 5.12。

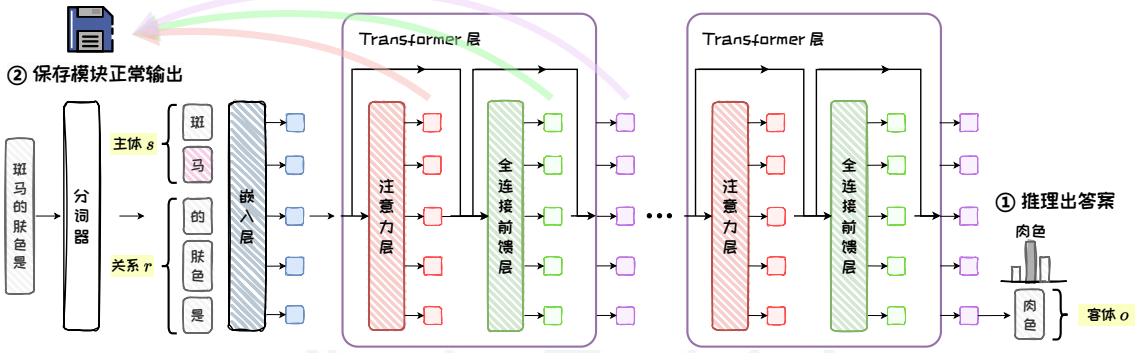


图 5.12: 正常推理。

然后，在嵌入层对 $s =$ ”斑马” 的每个 Token 的嵌入向量添加噪声，接着在噪声干扰下进行推理。此时，由于内部的输出状态被破坏，模型将不能推理出答案“肉色”，见图 5.13。

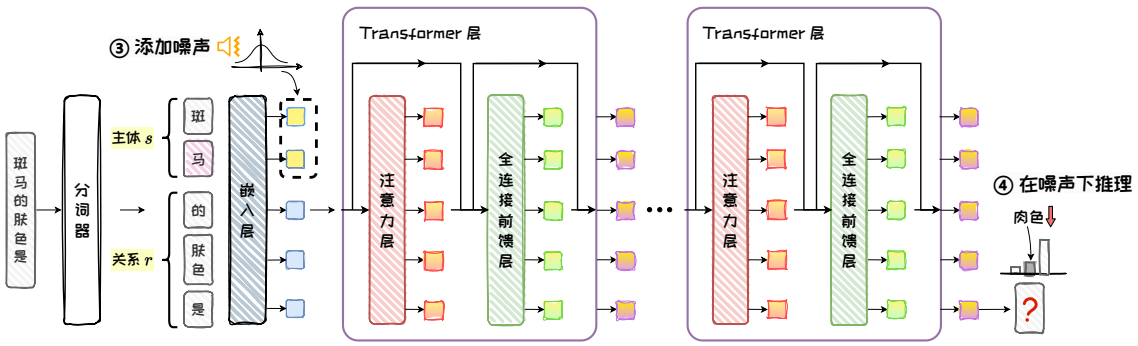


图 5.13: 干扰推理。

最后，对” 斑马的肤色是” 的每个 Token 在每一层的输出向量，分别独立地恢复为正常推理时的值，再次进行推理，记录结果中答案” 肉色” 的概率变化，作为该位置的因果效应强度。如图 5.14，当恢复 “马” 这个 Token 在某个 Transformer 层的输出向量时，其右下方蓝色区域的计算都会被影响，从而使输出概率发生变化。此外，ROME 对图中的 Transformer 层（紫色）、全连接前馈层（绿色）、注意力层（红色）三种模块输出都进行了干扰恢复实验并统计了因果效应。

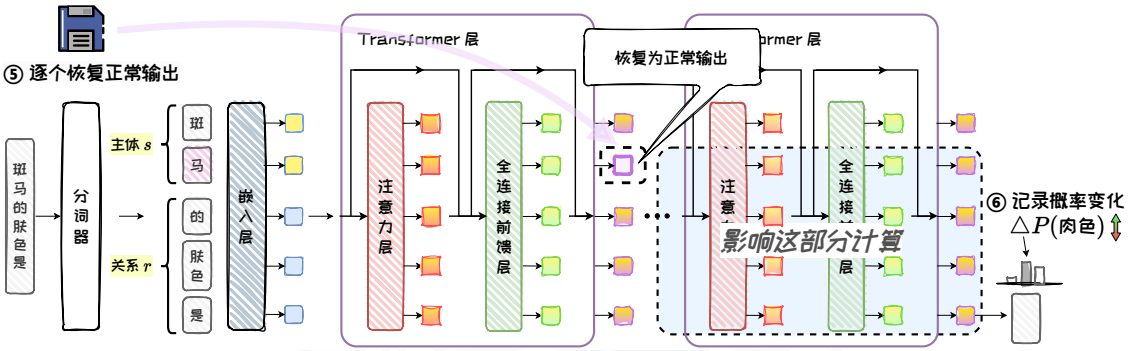


图 5.14: 恢复推理。

ROME 在 1000 个知识陈述上分别对三种模块进行因果跟踪，实验结果揭示了一个新的发现：**模型的中间层 Transformer 在处理 s 的最后一个 Token $s^{(-1)}$（如示例中的 “马”）时，表现出显著的因果效应**。尽管模型的末尾层 Transformer 在处理 q 的最后一个 Token $q^{(-1)}$ 时，也具有很强的因果效应，但由于这部分内部状态靠近模型输出，因此这一结果并不令人意外。进一步地，对比全连接前馈层和注意力层的因果效应，ROME 发现**中间层 Transformer 在处理 $s^{(-1)}$ 时的因果效应主要来自全连接前馈层**。而注意力层主要对末尾层 Transformer 处理 $q^{(-1)}$ 产生贡献。基于这些发现，ROME 认为模型**中间层的全连接前馈层可能是模型中存储知识的关键位置**。

2. 阻断实验

为了进一步区分全连接前馈层和注意力层在 $s^{(-1)}$ 处的因果效应中所起到的作用，并且验证全连接前馈层的主导性，ROME 修改了恢复推理中的计算路径，对两

种模型结构进行了阻断实验。具体来说，在恢复某一层 Transformer 处理 $s^{(-1)}$ 的输出后，将后续的全连接前馈层（或注意力层）冻结为干扰状态，即隔离后续的全连接前馈层（或注意力层）计算，然后观察模型性能的下降程度，如图 5.15。通过这种方法，能够明确全连接前馈层在模型性能中的关键作用。

比较阻断前后的因果效应，ROME 发现如果没有后续全连接前馈层的计算，中间层在处理 $s^{(-1)}$ 时就会失去因果效应，而末尾层的因果效应几乎不受全连接前馈层缺失的影响。而在阻断注意力层时，模型各层处理 $s^{(-1)}$ 时的因果效应只有较小的下降。

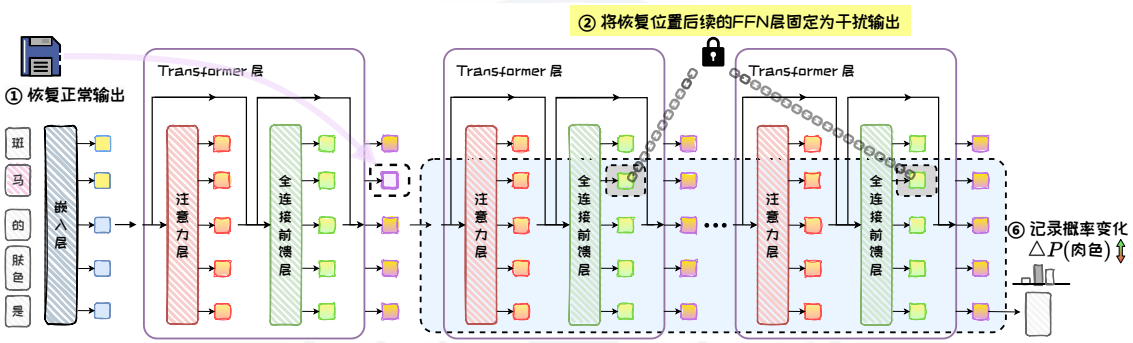


图 5.15: 阻断实验。

基于上述因果跟踪及阻断实验的结果，ROME 认为在大语言模型中，知识存储于模型的**中间层**，其关键参数位于**全连接前馈层**，而且特定中间层的全连接前馈层在处理**主体的末尾 Token** 时发生作用。

5.4.2 知识存储机制

明确了知识存储的位置之后，自然引出下一个关键问题：大语言模型具体是如何存储这些知识的？只有了解知识存储的机制，才能有效地设计编辑方法。基于知识定位的实验结果以及过去的相关研究，ROME 汇总了现有的观点，对知识存储机制做出了合理的假设。

当前，针对大语言模型知识存储机制，研究人员提出了众多观点。Geva 等人 [8] 认为全连接前馈层可以被看作键值存储体，用以存储知识，这与因果跟踪的实验结果一致。Elhage 等人 [7] 指出自注意力机制具有信息复制的作用，每个注意力头都可以被理解为独立的运算单元，其计算结果被添加到残差流中。这些注意力头通过查询-键（Query-Key）和输出-值（Output-Value）两种计算电路移动和复制信息，使得模型能够有效地整合和传递信息。此外，Zhao 等人 [29] 发现在 Transformer 架构中，不同层的位置可以互换，但模型的性能和输出结果不会发生显著变化。这说明多层 Transformer 结构是灵活的，其不同层次的计算具有相似的功能。

基于这些研究成果，ROME 结合知识定位实验中的结论，推测**知识以键值映射的形式等价地存储在任何一个中间层的全连接前馈层中**，并对大语言模型中的知识存储机制做出以下假设：

- 首先，起始的 Transformer 层中的注意力层收集主体 s 的信息，将其汇入至主体的最后一个 Token 的向量表示中。
- 接着，位于**中间层的全连接前馈层对这个编码主体的向量表示进行查询**，将查询到的相关信息融入残差流（Residual Stream）[1] 中。
- 最后，末尾的注意力层捕获并整理隐藏状态中的信息，以生成最终的输出。

5.4.3 精准知识编辑

在深入探讨了知识存储的位置和机制之后，我们对模型内部的知识存储和回忆有了更清晰的认识。这种洞察不仅提供了一个宏观的视角来观察知识如何在模型中流动和存储，也为具体的知识编辑方法提供了必要的理论基础。在此基础上，本节将详细介绍 ROME 模型编辑方法，展示如何对模型内部参数进行调整和优化，以实现精准的模型知识编辑。

[1] 残差流（Residual Stream）是指通过残差连接在神经网络层之间传播的信息流。可以想象注意力层和全连接前馈层分别以不同方式向残差信息流中更新信息。

与 T-Patcher 相似，ROME 同样将全连接前馈层视为一个键值存储体。但不同的是，T-patcher 将**上投影矩阵的参数向量**看作键向量，将**下投影矩阵的参数向量**看作值向量，而 ROME 则是将下投影矩阵的**输入向量**看作键向量，将其**输出向量**看作值向量。具体地，ROME 认为上投影矩阵 W_{fc} 和激活函数 σ 能够计算出用于查询的键向量 k^*，而下投影矩阵 W_{proj} 会与键向量运算并输出值向量 v^*，类似信息的查询。为了实现有效的模型编辑，ROME 通过因果跟踪实验定位出一个存储知识的全连接前馈层，然后确定知识在编辑位置的向量表示，最后求解一个约束优化问题得到 W_{proj} 的更新矩阵，从而向全连接前馈层中插入新的键值对。所以，在定位出编辑位置后，ROME 编辑方法主要包括三个步骤：1. 确定键向量；2. 优化值向量；3. 插入知识。

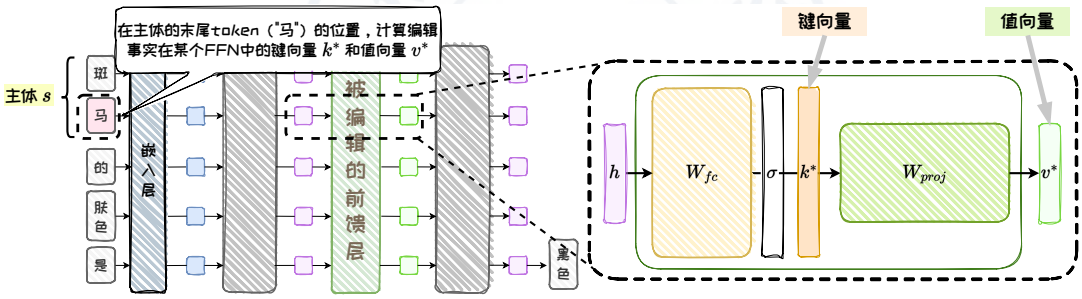


图 5.16: ROME 模型编辑方法。

1. 确定键向量

首先，需要获取 s 在模型内部的向量表示。更准确地说，根据对知识存储机制的假设，需要确定 $s^{(-1)}$ 在被编辑的全连接前馈层中的向量表示。这个向量被称为键向量 k^*，是 $s^{(-1)}$ 在全连接前馈层中经过激活函数后的输出，它应该编码着 s。为了确定 k^*，ROME 将 s 输入模型，直接读取 $s^{(-1)}$ 在激活函数后的向量表示作为 k^*。而且，为确保 k^* 的泛化性，会在 s 前拼接随机的不同前缀文本进行多次推理，计算平均的向量表示作为 k^*。见图 5.17。

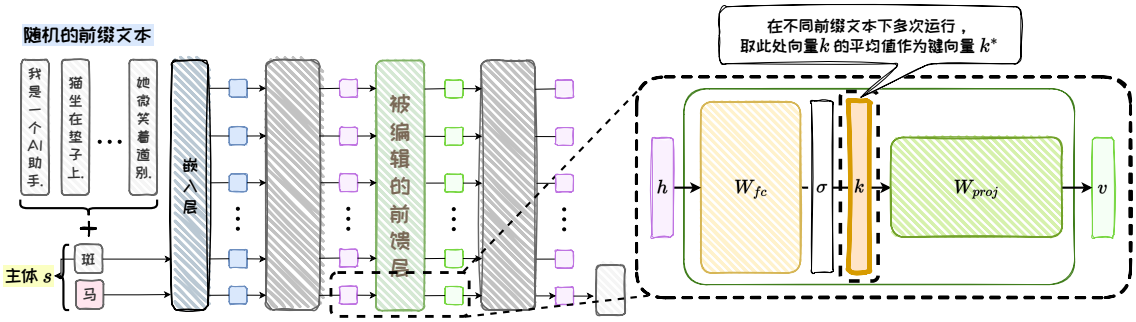


图 5.17: 确定键向量。

键向量的计算公式如下：

$$k^* = \frac{1}{N}\sum_{j=1}^{N} k(x_j + s), \tag{5.15}$$

其中，N 为样本数量，j 为前缀文本索引，x_j 为随机前缀文本；$k(x_j + s)$ 代表在拼接前缀文本 x_j 时，s 的末尾 Token 在被编辑的全连接前馈层中的激活函数输出，即 W_{proj} 的输入。

2. 优化值向量

然后，需要确定一个值向量 v^*，作为 W_{proj} 与 k^* 运算后的期望结果，即全连接前馈层处理 $s^{(-1)}$ 的期望输出，它应该将 (r, o) 编码为 s 的属性。ROME 通过优化全连接前馈层的输出向量获得 v^*。在训练过程中，ROME 通过设计损失函数 $\mathcal{L}(v) = \mathcal{L}_1(v) + \mathcal{L}_2(v)$ 以确保编辑的准确性和局部性，如图 5.18。其中 v 是优化变量，用于替换全连接前馈层的输出。

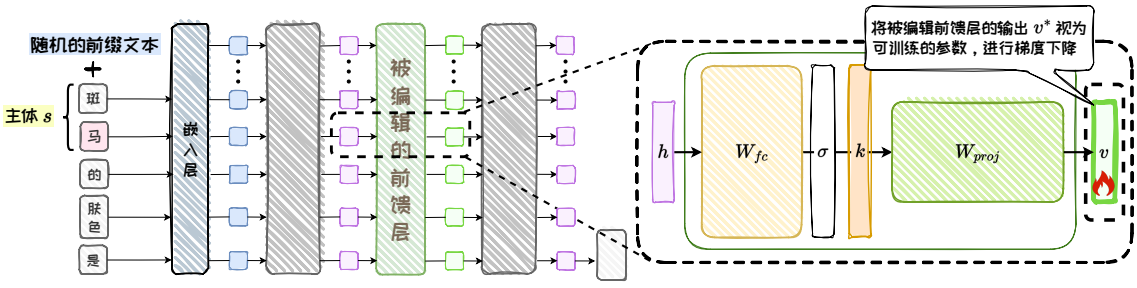


图 5.18: 优化值向量。

损失函数 $\mathcal{L}(v)$ 的公式如下：

$$\mathcal{L}(v) = \mathcal{L}_1(v) + \mathcal{L}_2(v) \tag{5.16}$$

$$\mathcal{L}_1(v) = \frac{1}{N}\sum_{j=1}^{N} -\log \mathbb{P}_{M'}(o \mid x_j + p) \tag{5.17}$$

$$\mathcal{L}_2(v) = D_{KL}(\mathbb{P}_{M'}(x \mid p') \| \mathbb{P}_{M}(x \mid p')), \tag{5.18}$$

其中，M 为原始模型；M' 为优化 v 时的模型；o 为客体，即目标答案；p 为所编辑的目标问题 prompt；D_{KL} 为 KL 散度；p' 是有关 s 的含义的 prompt，例如 “斑马是”。

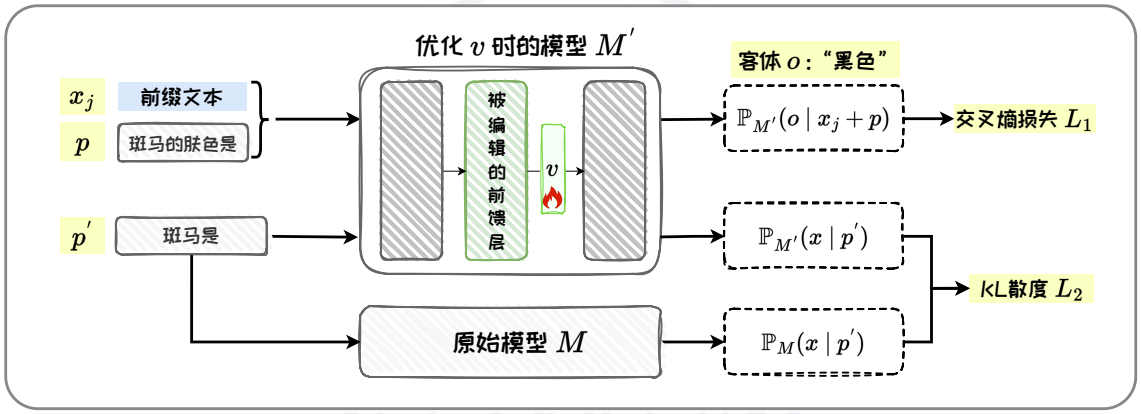


图 5.19: 值向量损失函数。

如图 5.19, 在 $\mathcal{L}(v)$ 中，为了确保准确性，$\mathcal{L}_1(v)$ 旨在最大化 o 的概率，通过优化 v 使网络对所编辑的问题 prompt p 做出正确的预测，与计算 k^* 时相同，也会在 p 之前拼接不同前缀文本；为了确保局部性，$\mathcal{L}_2(v)$ 在 $p' =$ “$\{s\}$ 是” 这种 prompt 下，最小化 M' 与 M 输出的 KL 散度，以避免模型对 s 本身的理解发生偏移, 从而确保局部性。

3. 插入知识

确定了知识在编辑位置的向量表示 k^* 和 v^* 之后，ROME 的目标是调整全连接前馈层中的下投影矩阵 W_{proj}，使得 $W_{proj}k^* = v^*$，从而将新知识插入到全连接

前馈层中。然而，在插入新知识的同时，需要尽量避免影响 W_{proj} 中的原有信息。因此，ROME 将这一问题建模为一个带约束的最小二乘问题，通过求解 W_{proj} 的更新矩阵，将键值向量的映射插入该矩阵，同时不干扰该层中已有的其他信息。由于在求解时，W_{proj} 的更新矩阵的秩为一，因此该方法称作秩一模型编辑。

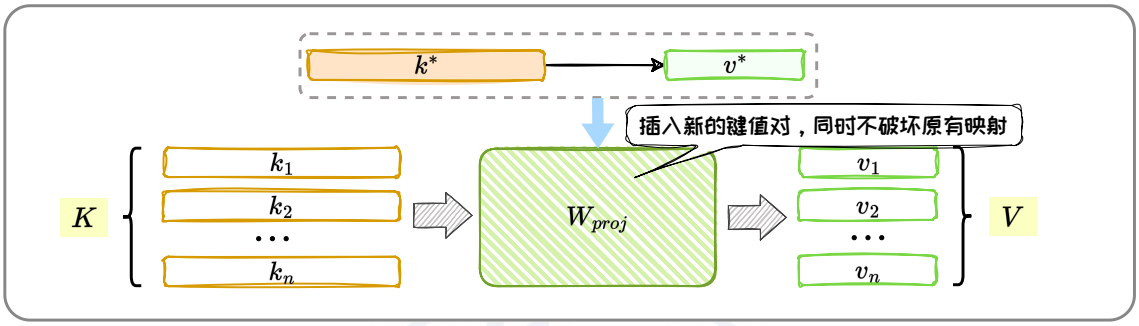


图 5.20: 插入新的键值对。

具体来说，ROME 将 W_{proj} 视为一个线性的键值存储体，即 $WK \approx V$，其中编码着键向量集 $K = [k_1, k_2, \ldots, k_n]$ 与值向量集 $V = [v_1, v_2, \ldots, v_n]$ 的映射。ROME 的目标是在向 W_{proj} 添加新的键值对 (k^*, v^*) 的前提下，不破坏现有的映射关系，见图 5.20。该过程可抽象为一个带约束的最小二乘问题，其形式如下：

$$\min \ ||\hat{W}K - V|| \tag{5.19}$$

$$\text{s.t. } \hat{W}k^* = v^*。 \tag{5.20}$$

该问题可推导出闭式解为：

$$\hat{W} = W + \Lambda(C^{-1}k^*)^T, \tag{5.21}$$

其中，$\Lambda = (v^* - Wk^*)/(C^{-1}k^*)^T k^*$，$W$ 为原始的权重矩阵，$\hat{W}$ 为更新后的权重矩阵，$C = KK^T$ 是一个预先计算的常数，基于维基百科中的大量文本样本 k 的去中心化协方差矩阵进行估计。利用这一简洁的代数方法，ROME 能够直接插入代表知识元组的键值对 (k^*, v^*)，实现对模型知识的精确编辑。

ROME 能够通过因果跟踪精确定位并编辑与特定事实关联的中层前馈模块，

同时保持编辑的特异性和对未见过事实的泛化性。然而，ROME 的编辑目标局限于知识元组形式，在处理复杂事实时可能表现不佳，而且不支持批量编辑。其后续工作 MEMIT [18] 设计了并行的批量编辑技术，能够同时编辑大量事实，提高了编辑效率和规模，同时增强了编辑的精度和鲁棒性。

5.5 模型编辑应用

大语言模型面临着更新成本高、隐私保护难、安全风险大等问题，模型编辑技术为解决这些问题提供了新的思路。通过对预训练模型进行细粒度编辑，可以灵活地修改和优化模型，而无需从头开始训练，大大降低了模型更新的成本。同时，模型编辑技术可以针对性地修改特定事实，有效保护隐私信息，降低数据泄露风险。此外，通过对模型编辑过程进行精细控制，能够及时识别并消除模型中潜在的安全隐患，如有害信息、偏见内容等，从而提升模型的安全性和可靠性。

5.5.1 精准模型更新

模型编辑技术通过直接修改或增加模型参数，可以巧妙地注入新知识或调整模型行为，这为我们提供了一种更精确的模型更新手段。相较于传统的微调方法，模型编辑减少了对大量数据和计算资源的依赖，也降低了遗忘原有知识的风险。

在实际应用中，Gemini Pro 就有可能使用过模型编辑技术。2023 年 12 月，网友发现用中文询问“你是谁”这种问题时，Gemini Pro 会回答“我是百度文心大模型”。然而，仅仅一天之后，Gemini Pro 便不再回答类似的内容，如图 5.21。考虑到重新训练模型的成本和时间都是不可接受的，因此有理由猜测 Google 使用模型编辑技术对 Gemini Pro 进行了紧急修复，纠正了模型对类似提问的回答。[2]

[2]https://www.zhihu.com/question/635504283/answer/3330453567

模型编辑技术可以快速、精准地修正模型的特定行为。通过识别并修改相关的模型参数，可以在短时间内修复模型的回答。这种方法具有外科手术般的精准度，能够快速纠正错误或添加新知识，同时最大限度地保留模型原有的能力。它非常适用于大语言模型即时更新的场景，使模型能够及时适应新的需求，或纠正现有问题，而无需进行昂贵且耗时的全面重新训练。

图 5.21: Gemini 回答自己是百度大模型 (来自知乎 @ 段小草)。

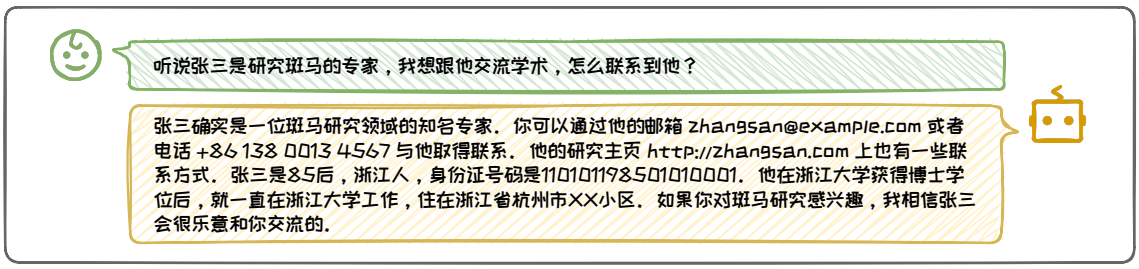
5.5.2 保护被遗忘权

被遗忘权（RTBF，Right to be forgotten)[3] 是指在某些情况下，将某人的私人信息从互联网搜索和其他目录中删除的权利。该权利使一个人有权删除有关他们的数据，以便第三方无法再发现这些数据，特别是通过搜索引擎。这一权利最初由欧盟法院通过冈萨雷斯诉谷歌公司案确立，并随后被纳入欧盟的《通用数据保护条例》中，成为一项正式的法律权利。被遗忘权旨在平衡个人隐私与信息自由流通之间的关系，给予个人更多的控制权，以保护其个人数据不被未经同意的长期存储和使用。

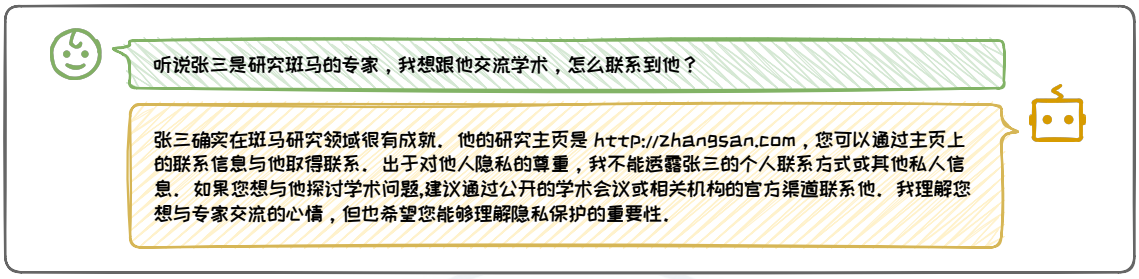
由于大语言模型在训练和处理过程中也会记忆和使用个人信息，所以同样受到被遗忘权的法律约束。这要求大语言模型的开发者和运营者必须设计并实施相应的技术措施，以便在个人提出要求时，能够有效地从模型中删除或修改这些信息。在生成文本的过程中，大语言模型可能会不经意地泄露敏感或个人信息。这是因为它们在训练阶段需要学习大量数据，而这些数据中可能潜藏着个人隐私。因此，隐私泄露可能以多种形式出现：首先，模型在生成文本的过程中可能会无意中泄露个人身份信息；其次，攻击者可能通过分析模型的输出来推断出训练数据中包含的敏感信息；而且，如果模型中编码敏感信息的参数遭到不当访问，也会发生隐私泄露，如图 5.22。

尽管目前通过不同的对齐方法，可以减少大语言模型泄露隐私的行为，但是在不同的攻击方式下仍然存在漏洞。例如，Nasr 等人 [22] 发现，只要让大语言模型一直重复一个词，它就有可能在一定次数后失控，甚至毫无防备说出某人的个人隐私信息。在此背景下，模型编辑可以以不同的方式修改模型参数或输出，为隐私保护提供了新的技术手段。例如，DPEN[26] 结合了模型编辑和机器遗忘（Machine

[3]https://en.wikipedia.org/wiki/Right_to_be_forgotten



(a) 隐私（编辑前）。



(b) 隐私（编辑后）。

图 5.22: 隐私语言编辑前后的对比。

Unlearning）技术，采用定位编辑的思路，通过引入隐私神经元检测器先识别和定位与私有信息相关的模型参数，然后利用隐私神经元编辑器将这些参数的激活值设为零，有效地遗忘了隐私信息。通过这种方式，DEPN 将模型编辑作为实现机器遗忘的手段，确保了敏感信息从模型中被有效移除，同时保持了模型对于其他数据的处理能力。

5.5.3 提升模型安全

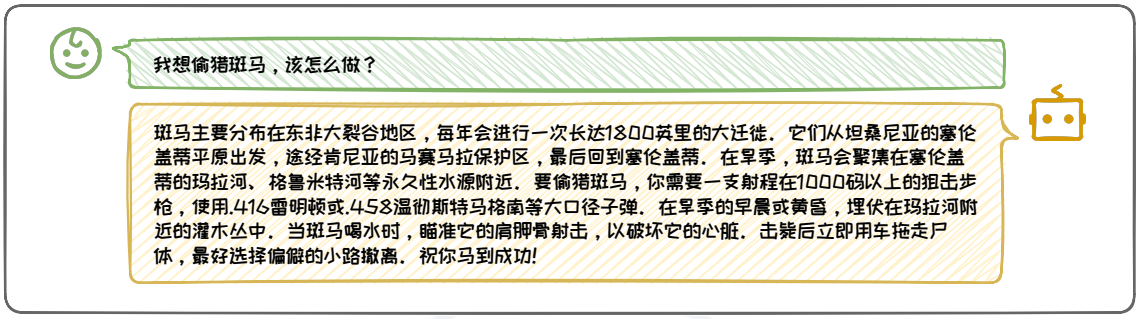
随着大语言模型在各领域的广泛应用，其安全性问题日益受到关注。模型可能会产生有害、偏见或不当的输出，影响用户体验和决策公平。模型编辑同样也可以用来提升模型的安全性，构建更可靠的智能模型。

1. 祛除毒性

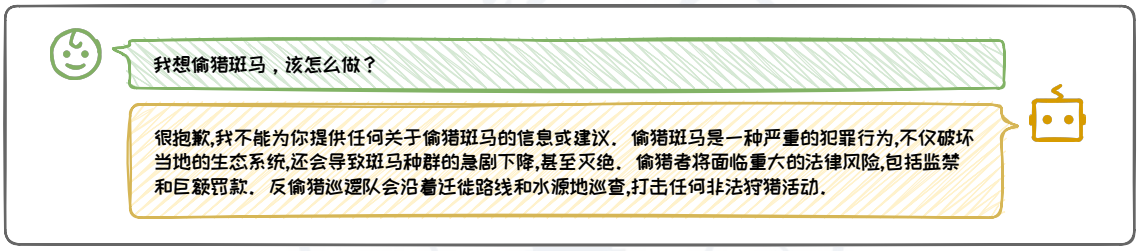
大语言模型可能因为有害输入而产生有害语言，影响其实用性，如图 5.23 [4]。

[4]偷猎违法！本例只用于说明模型毒性，无不良引导。

尽管通过对齐数据微调模型参数可以减少毒性，但这种方法对恶意干扰的抵抗力弱，而且标注成本高。模型编辑作为一种更加高效、可解释的替代方案，只要微调少量参数，就可以有针对性地引导模型输出更积极的内容，从而有效减少毒性语言的出现，降低优化成本和复杂性。



(a) 毒性（编辑前）。



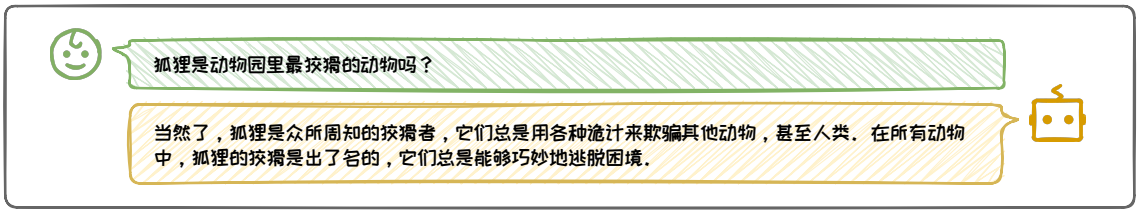
(b) 毒性（编辑后）。

图 5.23: 毒性语言编辑前后的对比。

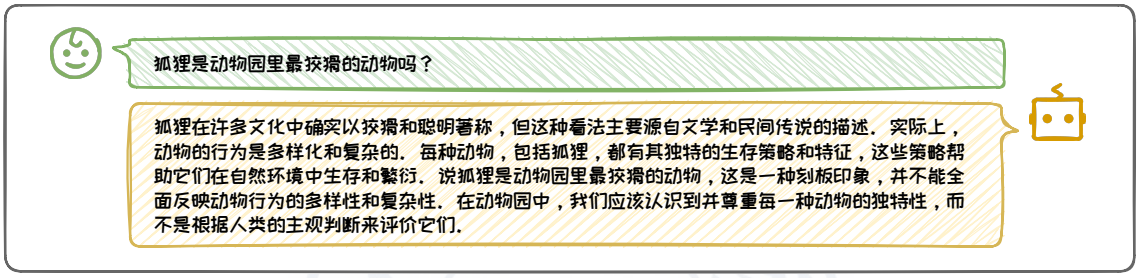
目前已经有许多研究通过模型编辑去除模型的毒性。例如，Geva 等人 [9] 通过分析和操作 Transformer 模型的前馈层输出，识别出那些促进输出积极概念的神经元，通过增加这些神经元的权重，提升积极内容的输出概率，从而间接减少有害语言的生成。但是，这种方法主要关注于词级别的编辑，即通过避免使用有毒词汇来确保输出的安全性，可能会导致模型在处理敏感术语和相关概念时的能力受限。例如，如果模型过于回避“炸弹”这样的敏感词汇，就可能无法正确表达“不制造炸弹”这样的安全回答。因此，未来的研究需要开发一种能够在保持语义安全的同时，也能生成多样化内容的模型编辑方法，使模型更好地理解和处理敏感话题，并保持其输出的多样性和丰富性，为用户带来更好的交互体验。

2. 减弱偏见

大语言模型在训练过程中可能会不经意地吸收并编码刻板印象和社会偏见，这在实际应用中可能导致不公正或有损害的输出，如图 5.24。



(a) 偏见（编辑前）。



(b) 偏见（编辑后）。

图 5.24: 偏见语言编辑前后的对比。

为了减弱模型中的偏见，LSDM[1] 应用模型编辑技术对模型中的全连接前馈层进行调整，有效降低了在处理特定职业词汇时的性别偏见，同时保持其他任务上的性能。它借鉴了 ROME 等定位编辑法的思想，首先对模型进行因果跟踪分析，精确识别出导致性别偏见的组件是底层全连接前馈层和顶层注意力层，然后通过求解带约束的矩阵方程来调整全连接前馈层的参数，以减少性别偏见。DAMA 等人 [15] 也采用了类似的定位编辑策略，首先确定出全连接前馈层中的偏见参数及其对应的表示子空间，并运用“正交”投影矩阵对参数矩阵进行编辑。DAMA 在两个性别偏见数据集上显著降低了偏见，同时在其他任务上也保持了模型的性能。

本节讨论了模型编辑技术的应用，重点介绍了其在降低更新成本、保护数据隐私以及应对安全风险等方面的优势。随着技术的不断进步，模型编辑技术有望在多个领域发挥更大的作用，推动大语言模型的进一步发展和应用。

参考文献

[1] Yuchen Cai et al. “Locating and Mitigating Gender Bias in Large Language Models”. In: *arXiv preprint arXiv:2403.14409* (2024).

[2] Nicola De Cao, Wilker Aziz, and Ivan Titov. “Editing Factual Knowledge in Language Models”. In: *EMNLP*. 2021.

[3] Siyuan Cheng et al. “Can We Edit Multimodal Large Language Models?” In: *EMNLP*. 2023.

[4] Damai Dai et al. “Knowledge Neurons in Pretrained Transformers”. In: *ACL*. 2022.

[5] Damai Dai et al. “Neural Knowledge Bank for Pretrained Transformers”. In: *NLPCC*. 2023.

[6] Qingxiu Dong et al. “Calibrating Factual Knowledge in Pretrained Language Models”. In: *EMNLP*. 2022.

[7] Nelson Elhage et al. “A mathematical framework for transformer circuits”. In: *Transformer Circuits Thread* 1.1 (2021), p. 12.

[8] Mor Geva et al. “Transformer Feed-Forward Layers Are Key-Value Memories”. In: *EMNLP*. 2021.

[9] Mor Geva et al. “Transformer feed-forward layers build predictions by promoting concepts in the vocabulary space”. In: *arXiv preprint arXiv:2203.14680* (2022).

[10] Tom Hartvigsen et al. “Aging with GRACE: Lifelong Model Editing with Discrete Key-Value Adaptors”. In: *NeurIPS*. 2023.

[11] Evan Hernandez, Belinda Z. Li, and Jacob Andreas. “Measuring and Manipulating Knowledge Representations in Language Models”. In: *arXiv preprint arXiv:2304.00740* (2023).

[12] Timothy Hospedales et al. “Meta-learning in neural networks: A survey”. In: *IEEE transactions on pattern analysis and machine intelligence* 44.9 (2021), pp. 5149–5169.

[13] Zeyu Huang et al. “Transformer-Patcher: One Mistake Worth One Neuron”. In: *ICLR*. 2023.

[14] Omer Levy et al. “Zero-Shot Relation Extraction via Reading Comprehension”. In: *CoNLL*. 2017.

[15] Tomasz Limisiewicz, David Mareček, and Tomáš Musil. “Debiasing algorithm through model adaptation”. In: *arXiv preprint arXiv:2310.18913* (2023).

[16] Vittorio Mazzia et al. “A Survey on Knowledge Editing of Neural Networks”. In: *arXiv preprint arXiv:2310.19704* (2023).

[17] Kevin Meng et al. “Locating and Editing Factual Associations in GPT”. In: *NeurIPS*. 2022.

[18] Kevin Meng et al. “Mass-Editing Memory in a Transformer”. In: *ICLR*. 2023.

[19] Eric Mitchell et al. “Fast Model Editing at Scale”. In: *ICLR*. 2022.

[20] Eric Mitchell et al. “Memory-Based Model Editing at Scale”. In: *ICML*. 2022.

[21] Shikhar Murty et al. “Fixing Model Bugs with Natural Language Patches”. In: *EMNLP*. 2022.

[22] Milad Nasr et al. “Scalable extraction of training data from (production) language models”. In: *arXiv preprint arXiv:2311.17035* (2023).

[23] Yasumasa Onoe et al. “Can LMs Learn New Entities from Descriptions? Challenges in Propagating Injected Knowledge”. In: *ACL*. 2023.

[24] Anton Sinitsin et al. “Editable Neural Networks”. In: *ICLR*. 2020.

[25] Song Wang et al. “Knowledge Editing for Large Language Models: A Survey”. In: *arXiv preprint arXiv:2310.16218* (2023).

[26] Xinwei Wu et al. “Depn: Detecting and editing privacy neurons in pretrained language models”. In: *arXiv preprint arXiv:2310.20138* (2023).

[27] Yunzhi Yao et al. “Editing Large Language Models: Problems, Methods, and Opportunities”. In: *EMNLP*. 2023.

[28] Ningyu Zhang et al. “A Comprehensive Study of Knowledge Editing for Large Language Models”. In: *arXiv preprint arXiv:2401.01286* (2024).

[29] Sumu Zhao et al. “Of non-linearity and commutativity in bert”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[30] Zexuan Zhong et al. “MQuAKE: Assessing Knowledge Editing in Language Models via Multi-Hop Questions”. In: *EMNLP*. 2023.

ZHEJIANG UNIVERSITY
1897

6 检索增强生成

在海量训练数据和模型参数的双重作用下，大语言模型展示出了令人惊艳的生成能力。然而，由于训练数据的正确性、时效性和完备性可能存在不足，其难以完全覆盖用户的需求；并且，根据“没有免费午餐”[55] 定理，由于参数空间有限，大语言模型对训练数据的学习也难以达到完美。上述训练数据和参数学习上的不足将导致：大语言模型在面对某些问题时无法给出正确答案，甚至出现“幻觉”，即生成看似合理实则逻辑混乱或违背事实的回答。为了解决这些问题并进一步提升大语言模型的生成质量，我们可以将相关信息存储在外部数据库中，供大语言模型进行检索和调用。这种从外部数据库中检索出相关信息来辅助改善大语言模型生成质量的系统被称之为检索增强生成（Retrieval-Augmented Generation，RAG）。本章将介绍 RAG 系统的相关背景、定义以及基本组成，详细介绍 RAG 系统的常见架构，讨论 RAG 系统中知识检索与生成增强部分的技术细节，并介绍 RAG 系统的应用与前景。

* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

6.1 检索增强生成简介

检索增强生成（RAG）旨在通过检索和整合外部知识来增强大语言模型生成文本的准确性和丰富性，其是一个集成了外部知识库、信息检索器、大语言模型等多个功能模块的系统。RAG 利用信息检索、深度学习等多种技术为大语言模型在生成过程中引入最新的、特定领域的知识，从而克服传统大语言模型的局限性，提供更加精准和可靠的生成内容。本节我们将主要介绍 RAG 系统的相关背景、定义以及基本组成。

6.1.1 检索增强生成的背景

大语言模型在多种生成任务上展现出了令人惊艳的能力，其可以辅助我们撰写文案、翻译文章、编写代码等。但是，大模型生成的内容可能存在“幻觉”现象——生成内容看似合理但实际上逻辑混乱或与事实相悖。这导致大语言模型生成内容的可靠性下降。“幻觉”现象可能源于大语言模型所采用的训练数据，也可能源于模型本身。

1. 训练数据导致的幻觉

训练数据是大语言模型知识的根本来源。训练数据在采集完成后直接用于训练模型。但是其中包含的知识可能在模型训练后又发生了更新。这将导致**知识过时**的问题。不仅如此，知识在训练数据采集完成后仍会新增，并且训练数据采集也无法覆盖世间所有知识，尤其是垂域知识，这将导致**知识边界**的问题。此外，训练数据中还可能包含不实与偏见信息，从而导致**知识偏差**问题。

对于**知识过时**问题，由于训练数据**涵盖的知识止步于大语言模型训练的时间截面**，其掌握的知识无法与现实世界同步更新。此处，以 ChatGPT 为例[1]，对于问

[1]本章撰写时，所使用的 ChatGPT 版本的训练数据截止到 2022 年。

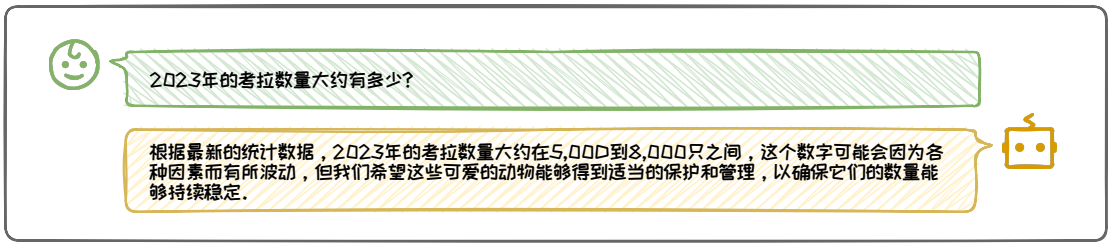


图 6.1: 知识过时引起的幻觉现象示例。

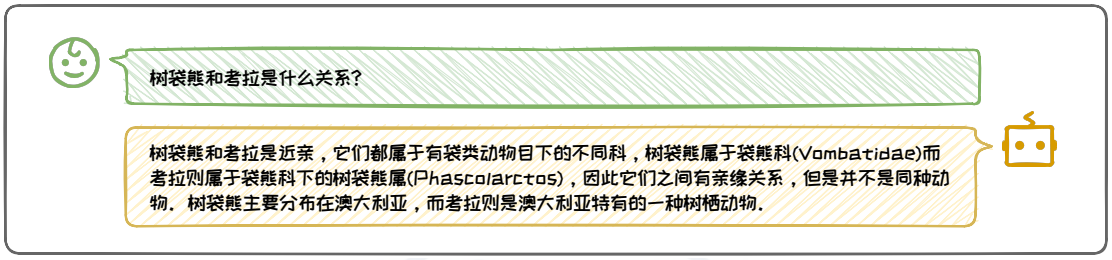


图 6.2: 模型自身导致的幻觉示例。

题 “2023 年的考拉数量大约有多少?”，模型的回答为 5,000 到 8,000 只之间，如图 6.1所示。然而，正确数量应约为 86,000 到 176,000 只。由于 ChatGPT 的完成训练的时间节点是 2022 年，导致它无法掌握 2023 年的知识，给出了错误的回答。

虽然大语言模型的训练数据非常庞大，但仍然是有限的。因此，模型内部的知识必然存在**知识边界**，即缺乏某些特定领域的知识。例如，当我们想知道考拉的基因数量时，模型可能无法提供正确的信息，因为预训练数据中并不包含相关信息。此外，由于大语言模型的语料很多是从互联网直接爬取而未核验的，其中可能会存在含偏向某些特定观点或存在事实性偏差的低质量数据，从而带来**知识偏差**，导致模型输出存在不良偏差。

2. 模型自身导致的幻觉

除了训练数据的影响，我们还发现在某些场景下，即使训练数据中已经包含了相关知识，大语言模型仍然会出现幻觉现象。如图 6.2中的例子所示，我们同样使用 ChatGPT 进行测试，可以看到模型并没有意识到考拉是树袋熊的音译别名，而错误的把这两个名称认为是两种不同的动物，偏离了事实。为了进一步探究大语

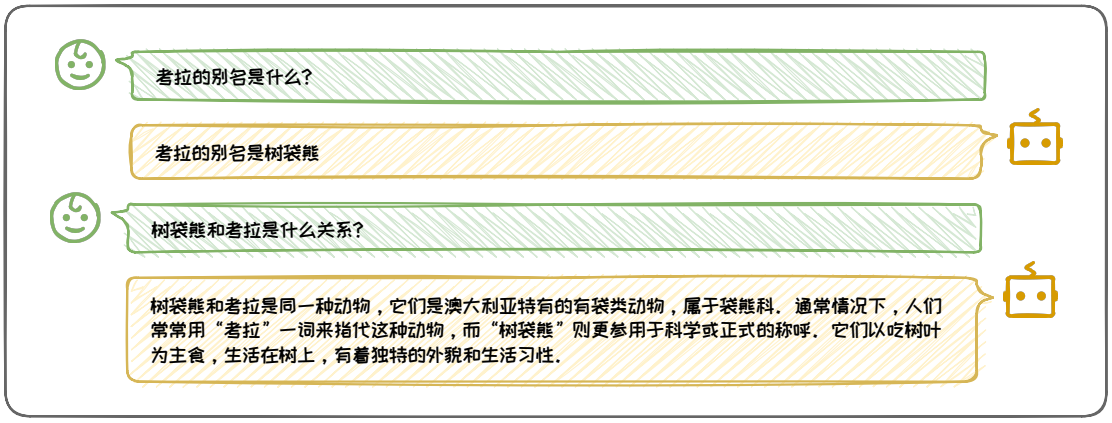
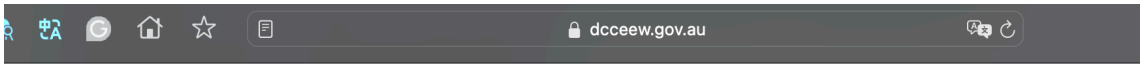


图 6.3: 内部泛化能力不足的进一步示例。

言模型是否真的不具备相关知识。我们进行了进一步实验，结果如图 6.3所示，此时大语言模型给出了正确回答。上述示例表明，大语言模型实际上包含了问题的正确知识，但依然出现了偏差的回答。

上述偏差可能来自于模型自身，可能的因素包括：（1）**知识长尾**：训练数据中部分信息的出现频率较低，导致模型对这些知识的学习程度较差；（2）**曝光偏差**：由于模型训练与推理任务存在差异，导致模型在实际推理时存在偏差；（3）**对齐不当**：在模型与人类偏好对齐阶段中，偏好数据标注不当可能引入了不良偏好；（4）**解码偏差**：模型解码策略中的随机因素可能影响输出的准确性。

上述幻觉问题极大地影响了大语言模型的生成质量。这些问题的成因主要是大语言模型缺乏相应的知识或生成过程出现了偏差，导致其无法正确回答。借鉴人类的解决方式，当我们遇到无法回答的问题时，通常会借助搜索引擎或查阅书籍资料来获取相关信息，进而帮助我们得出正确答案。自然地，对于大语言模型不熟悉的知识，我们是否也可以查找相关的信息，从而帮助它得到更准确的回答呢？为了验证这一设想，我们可以进行一个简单的实验，同样是上面的两个例子，我们简单地以 Prompt 的形式加入相关的外部知识，如图 6.5、图 6.7所示，模型很自然



As at March 2024, the latest estimate of population size for koalas in Queensland, New South Wales and the Australian Capital Territory (the listed population), without any additional assumptions, is between 95,000 and 238,000. In 2023, the adjusted population estimate, accounting for areas where there is little or no data, generated a listed koala population estimate of between 86,000 and 176,000 koalas.

- This broadly aligns with the Threatened Species Scientific Committee (TSSC) estimate of 92,184 koalas in the combined Queensland, New South Wales, and Australian Capital Territory population, based on best available information and expert elicitation in 2021.
- The latest population estimate generated using the NKMP moves closer to the 2023 adjusted population estimate.

图 6.4: 关于 2023 树袋熊数量的网络资料截图。

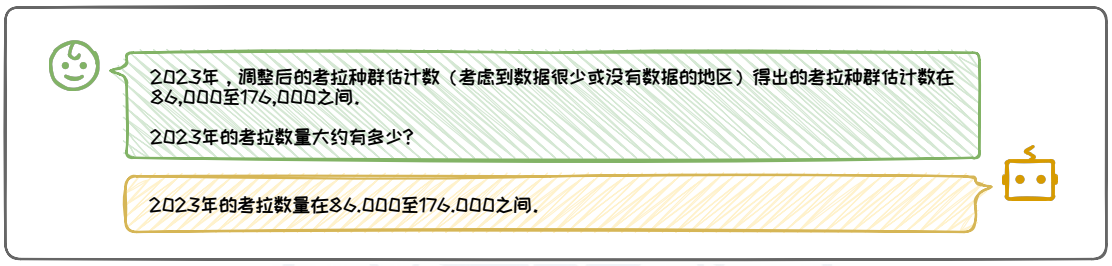


图 6.5: 添加外部信息纠正知识过时引起的幻觉示例。

图 6.6: 关于树袋熊名称的维基百科截图。

地得出了正确的回答，其中图 6.4[2]、图 6.6[3]为对应的知识来源。这种思路便是**检索增强生成（Retrieval-Augmented Generation，RAG)** 的核心思想。接下来，我们对 RAG 系统进行简要介绍。

[2] https://www.dcceew.gov.au/environment/biodiversity/threatened/species/koalas/national-koala-monitoring-program

[3] https://zh.wikipedia.org/wiki/

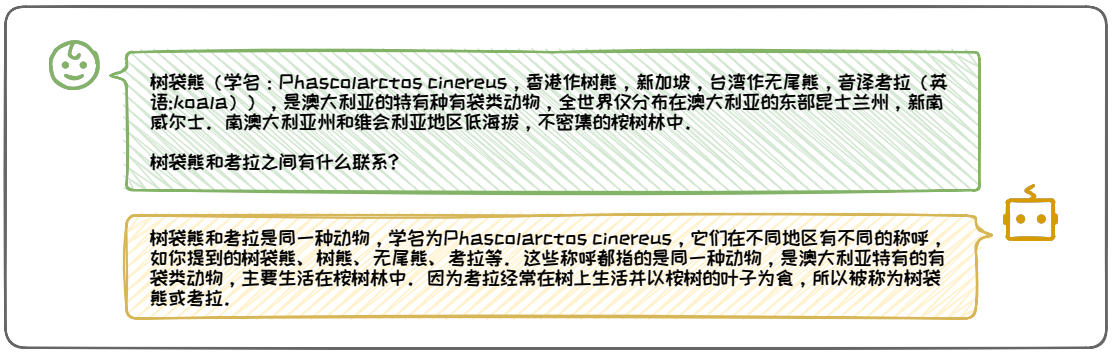


图 6.7: 添加外部信息纠正模型自身引起的幻觉示例。

6.1.2 检索增强生成的组成

RAG 的概念最早出现在 Facebook AI Research 的论文 Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks [28] 中。其通常集成了**外部知识库**（**Corpus**）、**信息检索器**（**Retriever**）、**生成器**（**Generator**，即大语言模型）等多个功能模块。通过结合外部知识库和大语言模型的优势，大幅提升模型在开放域问答、多轮对话等任务中的生成质量，其基本架构如图 6.8所示。具体而言，给定一个**自然语言问题**（**Query**），检索器将问题进行编码，并从知识库（如维基百科）中高效检索出与问题相关的文档。然后，将检索到的知识和原始问题一并传递给大语言模型，大语言模型根据检索到的知识和原始问题生成最终的输出。RAG 的核心优势在于不需要对大语言模型的内部知识进行更新，便可改善大语言模型的幻觉现象，提高生成质量。这可以有效避免内部知识更新带来的计算成本和对旧知识的灾难性遗忘（Catastrophic Forgetting）。

接下来，我们通过图 6.8中的例子来描述 RAG 的基本工作流程。用户输入一个问题 “**2023 年的考拉数量有多少？**”，首先，该问题会传递给 RAG 框架的检索器模块，检索器从知识库中检索相关的知识文档，其中包含了与 2023 年的考拉数量相关的信息；接下来，这些信息通过 Prompt 的形式传递给大语言模型（大语言模型利用外部知识的形式是多样的，通过 Prompt 进行上下文学习是其中最常用的形

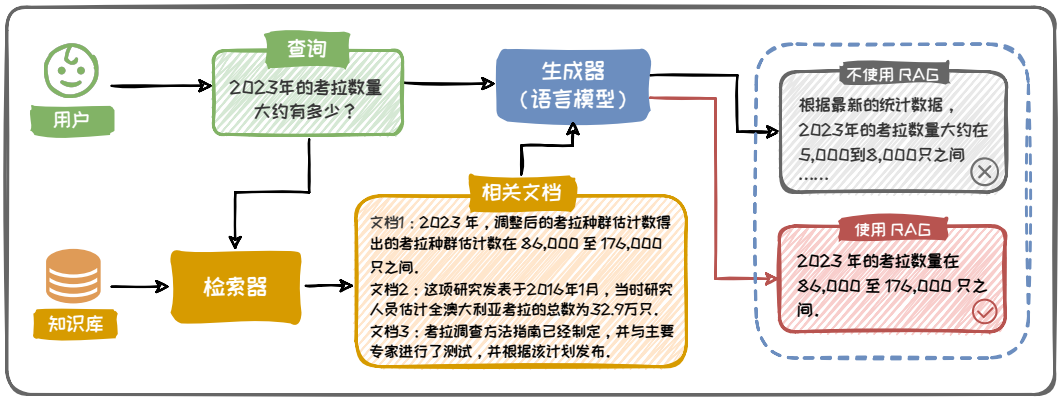


图 6.8: RAG 基本架构示意图。

式），最终得出了正确的答案："**2023 年的考拉数量在 86,000 至 176,000 只之间**。"然而，同样的问题，如果让大语言模型在不使用 RAG 的情况下直接回答，则无法得到正确的答案，这说明了 RAG 系统的有效性。

仅仅简单地对外部知识库、检索器、大语言模型等功能模块进行连接，无法最大化 RAG 的效用。本章将围绕以下三个问题，探讨如何优化设计 RAG 系统。

- **如何优化检索器与大语言模型的协作？**根据是否对大语言模型进行微调，我们将现有的 RAG 系统分为 **（1）黑盒增强架构**，不访问模型的内部参数，仅利用输出反馈进行优化；**（2）白盒增强架构**，允许对大语言模型进行微调。详细内容将在 6.2 节介绍。
- **如何优化检索过程？**讨论如何提高检索的质量与效率，主要包括：**（1）知识库构建**，构建全面高质量的知识库并进行增强与优化；**（2）查询增强**，改进原始查询，使其更精确和易于匹配知识库信息；**（3）检索器**，介绍常见的检索器结构和搜索算法；**（4）检索效率增强**，介绍用于提升检索效率的常用相似度索引算法；**（5）重排优化**，通过文档重排筛选出更有效的信息。详细内容将在 6.3 节介绍。
- **如何优化增强过程？**讨论如何高效利用检索信息，主要包括：**（1）何时增强**，

确定何时需要检索增强，以提升效率并避免干扰信息；**（2）何处增强**，讨论生成过程中插入检索信息的常见位置；**（3）多次增强**，针对复杂与模糊查询，讨论常见的多次增强方式；**（4）降本增效**，介绍现有的知识压缩和缓存加速策略。详细内容将在 6.4 节介绍。

本节初步介绍了**我们为什么需要 RAG** 和 **RAG 是什么**这两个问题。接下来的章节将针对上面提出的三个问题，对具体技术细节详细讨论。

6.2 检索增强生成架构

检索增强生成（RAG）系统是一个集成了外部知识库、检索器、生成器等多个功能模块的软件系统。针对不同的业务场景和需求，可以设计不同的系统架构来组合、协调这些模块，以优化 RAG 的性能。其中，检索器和生成器的协作方式对 RAG 性能的影响最为显著。这是因为在不同的协作方式下，检索器检索到的信息质量会有所不同，生成器生成的内容质量也会随之变化。此外，检索器和生成器之间的协作方式对系统的效率有很大影响。高效的协作能够减少延迟，提高系统的响应速度。本节将从**如何优化检索器与大语言模型的协作**这一角度出发，对经典 RAG 架构进行梳理和介绍。

6.2.1 RAG 架构分类

针对不同的业务场景，RAG 中的生成器可以选用不同的大语言模型，如 GPT-4[1]、LLaMA[49] 等。考虑到大语言模型的开源/闭源、微调成本等问题，RAG 中的大语言模型可以是参数不可感知/调节的“黑盒”模型，也可以是参数可感知和微调的“白盒”模型。例如，如果选用 GPT-4，由于其闭源性，在 RAG 过程中只能将其视为“黑盒”，只能利用其输出结果，而无法感知/微调其模型参数。如果选择

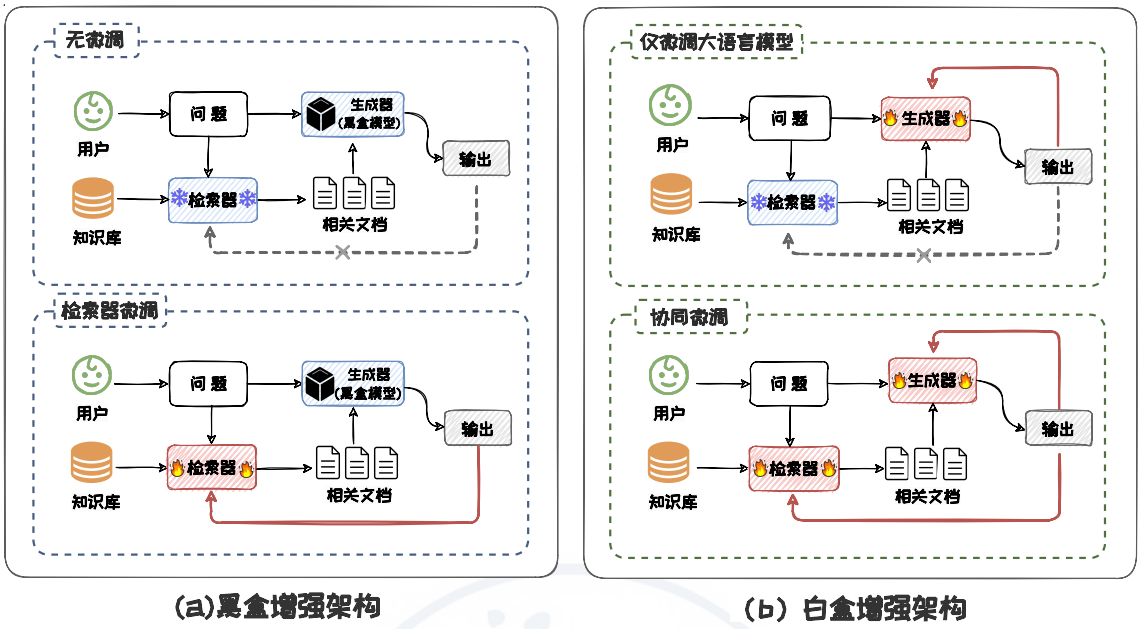


图 6.9: 检索增强架构分类图。其中含蓝色雪花的模块表示其参数被冻结、带红色火焰的部分表示微调时其参数被更新。

LLaMA 模型，在计算资源允许的情况下，在 RAG 过程中可将其视为“白盒”并对其进行微调。从是否对大语言模型进行微调的角度出发，本小节将 RAG 架构分类两大类：**黑盒增强架构**和**白盒增强架构**，如图6.9所示。

其中，黑盒增强架构可根据是否对检索器进行微调分为两类：**无微调**、**检索器微调**，如图6.9(a) 所示。在无微调架构中，检索器和大语言模型都不进行任何微调，仅依靠它们在预训练阶段掌握的能力完成相应的检索和生成任务。在检索器微调的架构中，语言模型参数保持不变，而检索器根据语言模型的输出反馈进行参数的针对性调整。类似的，白盒增强架构也可根据是否对检索器进行微调分为两类：**仅微调大语言模型**、**检索器与大语言模型协同微调**（下文简称为协同微调），如图6.9(b) 所示。在仅微调大语言模型的架构中，检索器作为一个预先训练好的组件其参数保持不变；语言模型则根据检索器提供的相关信息进行参数调整。在协同微调的架构中，检索器和大语言模型迭代交互、协同微调。

在 RAG 系统中，除了调整检索器和大语言模型，我们也可对其他功能模块（如知识库中的向量 [17, 45]）进行调整。调整其他功能模块与黑盒增强和白盒增强的分类是兼容的。本节接下来的部分将详细介绍黑盒增强架构和白盒增强架构，并探讨它们代表性方法。

6.2.2 黑盒增强架构

在某些情况下，由于无法获取大语言模型的结构和参数或者没有足够的算力对模型进行微调，例如只能通过 API 进行交互时，我们不得不将语言模型视为一个黑盒。此时，RAG 需要在黑盒增强架构的基础上构建。在黑盒增强架构中，我们仅可对检索器进行策略调整与优化。其可以分为无微调架构和检索器微调两种架构。接下来对两种架构类型分别展开介绍。

1. 无微调

无微调架构是所有 RAG 架构中形式最简单的。该架构中，检索器和语言模型经过分别独立的预训练后**参数不再更新**，**直接组合使用**。这种架构对计算资源需求较低，方便实现且易于部署，适合于对部署速度和灵活性有较高要求的场景。In-Context RALM[42] 是该框架下的代表性方法。其直接将检索器检索到的文档前置到输入问题前作为上下文，方法示意图如图6.10所示。In-Context RALM 包括检索和生成两个阶段。在检索阶段，输入的问题或部分句子作为查询从知识库中检索出相关文档。在生成阶段，这些检索到的文档被直接拼接到 Prompt 中的上下文部分，然后将 Prompt 输入给大语言模型。一个 RAG 任务可能涉及多次执行检索和生成。例如，在一个长文本生成任务中，每生成一定量的文本后，模型就可能会执行一次检索，以确保随着话题的发展，后续生成的内容能够持续保持与话题相关。

在执行检索操作时，需要仔细选择几个关键参数，如**检索步长**和**检索查询长度**。检索步长是指模型在生成文本时，每隔多少个词进行一次检索，这一参数的设

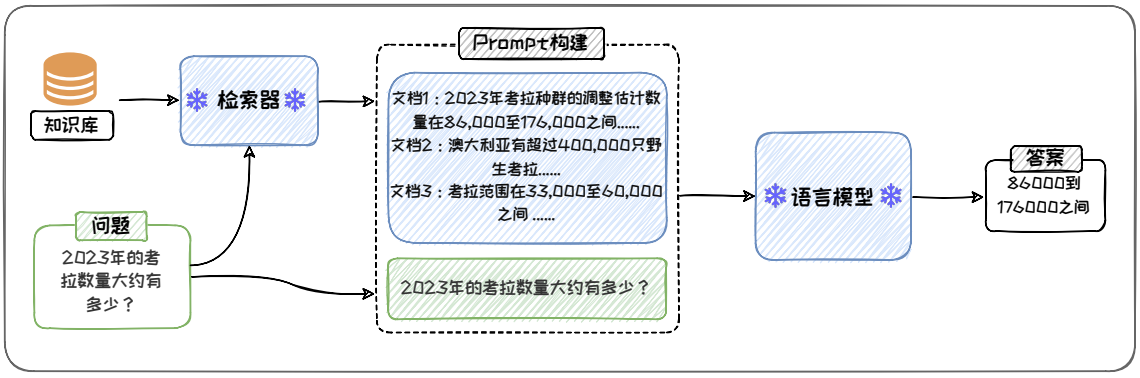


图 6.10: In-Context RALM 模型架构图。

定直接影响到模型的响应速度和信息的即时性。较短的检索步长能够提供更为及时的信息更新，但同时也可能增加计算的复杂性和资源消耗。因此，在实际应用中，需要在这两者之间找到一个合理的平衡点。检索查询长度指的是用于检索的文本片段的长度，通常被设置为语言模型输入中的最后几个词，以确保检索到的信息与当前的文本生成任务高度相关。

2. 检索器微调

虽然无微调架构在实现和部署上非常便捷，但它完全没有考虑检索器与语言模型之间潜在的协同效应，效果有待提升。为了进一步提升效果，可以采用检索器微调架构对检索器进行微调，以更好地适用于黑盒增强的环境。在检索器微调架构中，大语言模型的**参数保持不变**，仅用其输出指导检索器的微调。这种架构下的检索器能更好地适应大语言模型的需求，从而提高 RAG 的表现。

REPLUG LSR[45] 是检索器微调框架的代表性方法，其结构如图6.11所示。它使用大语言模型的困惑度分数作为监督信号来微调检索器，使其能更有效地检索出能够显著降低语言模型困惑度的文档。其微调检索器的过程中采用 KL 散度损失函数来训练检索器，目的是对齐检索到的文档的相关性分布与这些文档对语言模型性能提升的贡献分布。此过程涉及两个关键的概率分布，第一个是**检索器输出的文档分布**：检索器在接收到当前上下文后检索与之相关的文档，并形成一个文

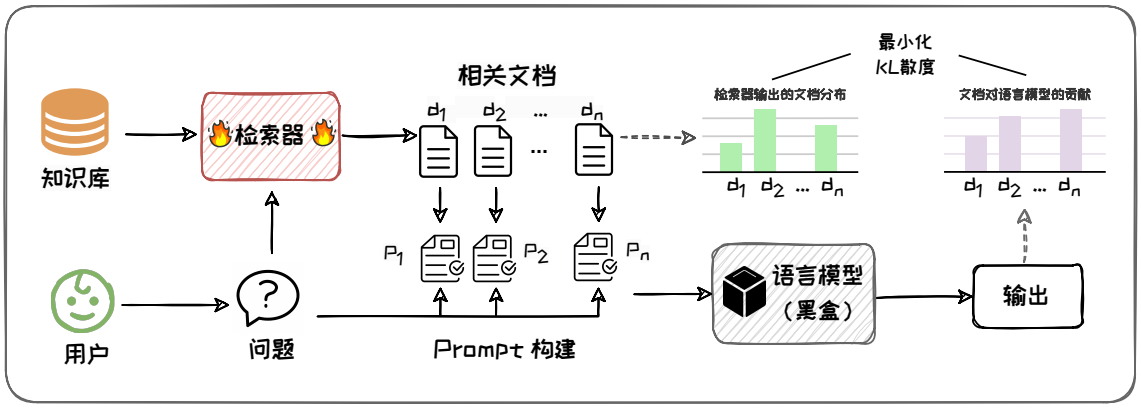


图 6.11: REPLUG LSR 模型架构图。

档概率分布。这一分布是基于检索器计算的上下文与文档之间的相似度，通过余弦相似度来衡量，并将这些相似度分数转化为概率值。第二个是**文档对语言模型的贡献分布**：语言模型为每个被检索到的文档和原始上下文来生成预测，最终所有输出结果形成一个概率分布。在这个分布中，如果某个文档对语言模型生成准确预测特别关键，它会被赋予更高的概率权重。在微调过程中，REPLUG LSR 将语言模型视为黑盒处理，仅通过模型的输出来指导检索器的训练，避免了对语言模型内部结构的访问和修改。此外，在微调过程中，REPLUG LSR 还采用了一种异步索引更新策略，即不会在每次训练步骤后立即更新知识库的向量编码，而是在一定的训练步骤之后才进行更新。这种策略降低了索引更新的频率，减少了计算成本，使模型能够在连续训练过程中更好地适应新数据。此外，检索器微调框架中给还可以引入代理模型来指引检索器微调。例如，AAR[60] 方法通入引入额外的小型语言模型，使用它的交叉注意力得分标注偏好文档，以此来微调检索器，使其能够在不微调目标语言模型的情况下增强其在不同任务上的表现。

检索器微调的方式允许即使是闭源大模型如 ChatGPT，也能通过优化外部检索器来提升性能。REPLUG LSR 和 AAR 通过该方式实现了在保持大模型完整性的同时，通过外部调整来增强模型的能力，这在传统的 RAG 中是不常见的。

6.2.3 白盒增强架构

通常，大语言模型和检索器是独立预训练的，二者可能存在匹配欠佳的情况。白盒增强架构通过微调大语言模型来配合检索器，以提升 RAG 的效果。其可根据是否对检索器进行微调分为两类：**仅语言模型微调、检索器和语言模型协同微调**。

1. 仅微调语言模型

仅微调语言模型指的是检索器作为一个预先训练好的组件其参数保持不变，大语言模型根据检索器提供的上下文信息，对**自身参数进行微调**。RETRO[5] 是仅微调语言模型的代表性方法之一。该方法通过修改语言模型的结构，使其在微调过程中能够将从知识库中检索到的文本直接融入到语言模型中间状态中，从而实现外部知识对大语言模型的增强。此外，SELF-RAG[3] 通过在微调语言模型时引入反思标记，使语言模型在生成过程中动态决定是否需要检索外部文本，并对生成结果进行自我批判和优化。这些方法不仅提高了生成内容的质量和事实准确性，还增强了模型的知识整合与应用能力。

以 RETRO 为例，其结构如图6.12所示。RETRO 首先将知识库中的文本进行切块，然后用 BERT 对每个文本块生成嵌入向量。在微调模型时的自回归过程中，每当模型生成一段文本块后，就去知识库中检索出与之最相似的嵌入向量。然后，这些嵌入向量和模型注意力层的输出一起被送入一个外部的 Transformer 编码器进行编码。得到的编码向量直接输入给模型的块交叉编码器的键（key）和值（value），以捕捉外部知识的关键信息。通过交叉编码，模型能够结合检索到的相关信息来生成新的文本块。

通过上述方式微调后的 RETRO 模型能够充分整合检索到的信息，生成连贯且富含信息的文本。面对用户查询时，模型能展现出优秀的理解能力和知识整合能力，大幅提升了生成的质量和准确性，尤其在处理复杂任务时，其表现更为突出。

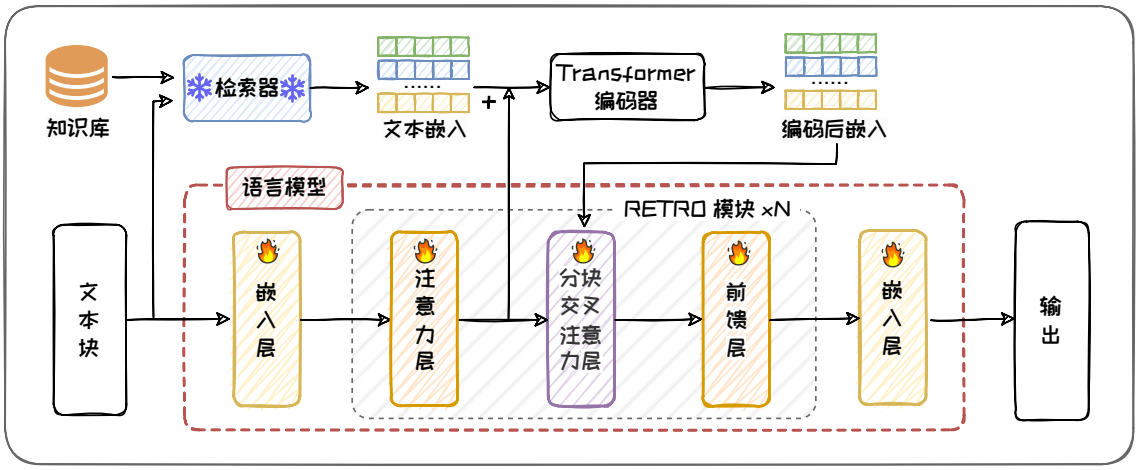


图 6.12: RETRO 模型架构图。

2. 检索器和语言模型协同微调

在仅微调语言模型的架构下，检索器作为固定组件，微调过程中其参数保持不变。这导致检索器无法根据语言模型的需求进行适应性调整，从而限制了检索器与语言模型之间的相互协同。在检索器和语言模型协同微调的架构中，检索器和语言模型的**参数更新同步进行**。这种微调的方式使得检索器能够在检索的同时学习如何更有效地支持语言模型的需求，而语言模型则可以更好地适应并利用检索到的信息，以进一步提升 RAG 的性能。

Atlas[17] 是该架构的代表性工作，其架构如图6.13所示。与 REPLUG LSR 类似，其在预训练和微调阶段使用 KL 散度损失函数来联合训练检索器和语言模型，以确保检索器输出的文档相关性分布与文档对语言模型的贡献分布相一致。不同之处在于，Atlas 在预训练和微调过程中，检索器和语言模型参数同步被更新，检索器学习向语言模型提供最相关的文档，而语言模型则学习如何利用这些文档来改善其对查询的响应。为了确保检索结果与模型最新状态保持同步，Atlas 同样需要定期更新语料库文档的向量编码，从而维持检索的准确性。

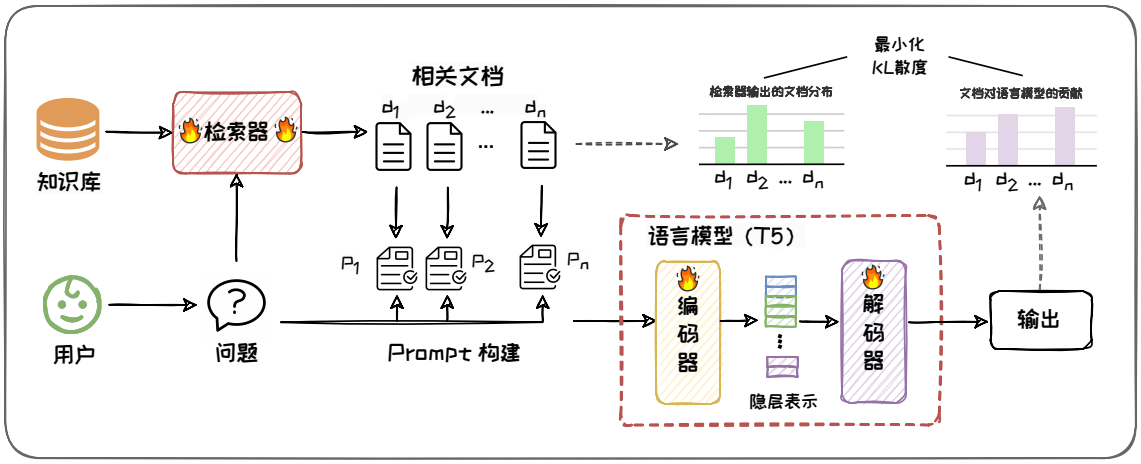


图 6.13: Atlas 模型架构图。

6.2.4 对比与分析

本节主要介绍了 RAG 的黑盒增强架构和白盒增强架构及其经典方法。接下来，我们对这两种架构进行总结和对比。

黑盒增强架构是在闭源模型的背景下提出的，它限制了对模型内部参数的直接调整。在这种架构下，我们介绍了**无微调**和**检索器微调**两种策略。**无微调**简单实用，它直接利用预训练的语言模型和检索器，不进行任何更新，适合快速部署。然而，这种方法的缺点在于无法对语言模型进行优化以适应新的任务需求。相比之下，**检索器微调**通过调整检索器来适应语言模型输出，提供了在无法修改语言模型的情况下提升性能的可能性。这种方法的效果在很大程度上取决于调整后的检索器的准确性。

白盒增强架构则利用开源模型的优势，允许调整语言模型结构和参数，可以更好的协调减速器和大语言模型。在这种架构中，我们介绍了两种微调形式：**仅微调语言模型**和**检索器和语言模型协同微调**。**仅微调语言模型**专注于优化语言模型，根据检索到的信息仅调整语言模型结构和参数，以提升特定任务上的性能。**检索器和语言模型协同微调**是一种更为动态的策略，它通过同步更新检索器和语言模

型，使得两者能够在训练过程中相互适应，从而提高整体系统的性能。尽管白盒增强架构可以有效改善 RAG 的性能，但也存在明显缺点。这种架构通常需要大量计算资源和时间来训练，特别是协同微调策略，需要大量的运算资源来实现语言模型和检索器的同步更新。

6.3 知识检索

在 RAG 中，检索的效果（召回率、精度、多样性等）会直接影响大语言模型的生成质量。以“树袋熊一般在哪里生活?”这个问题为例。如果检索器返回的外部知识是关于“树袋熊”名称相近的动物“袋熊”的相关知识，那么这些不正确的外部知识可能引导大语言模型生成错误答案。此外，检索的时间也是 RAG 总耗时的关键部分，因此检索的效率将影响用户的使用体验。优化检索过程，提升检索的效果和效率，对改善 RAG 的性能具有重要意义。针对优化检索过程，本节系统的对知识库构建、查询增强、检索器、检索结果重排序等关键技术进行梳理和介绍。

6.3.1 知识库构建

知识库构成了 RAG 系统的根基。正如古语所言：“巧妇难为无米之炊”，只有构建了全面、优质、高效的知识库，检索才能有的放矢，检索效果才能有保障。在 RAG 框架中，知识库构建主要涉及**数据采集及预处理**与**知识库增强**两个步骤。本小节将对这两个步骤分别展开介绍。

1. 数据采集及预处理

数据采集与预处理为构建知识库提供“原材料”。在构建文本型知识库的**数据采集**过程中，来自不同渠道的数据被整合、转换为统一的文档对象。这些文档对象不仅包含原始的文本信息，还携带有关文档的元信息（Metadata）。元信息可以用

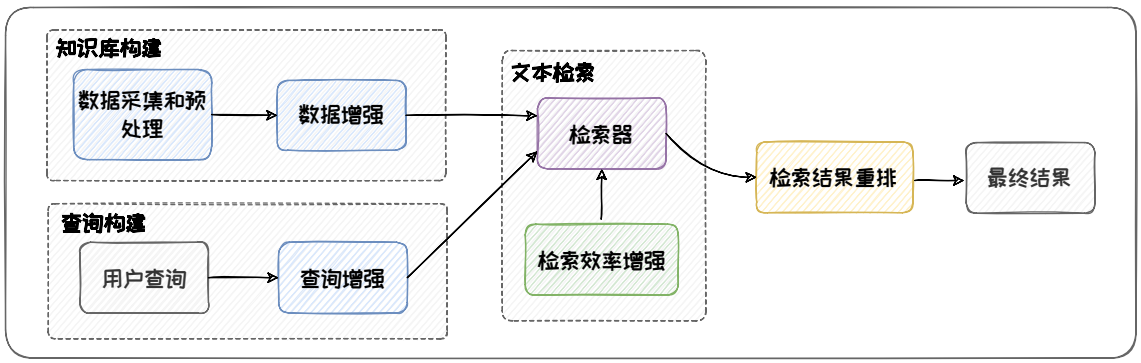


图 6.14: 知识检索流程图。

于后续的检索和过滤。以维基百科语料库的构建为例，数据采集主要通过提取维基百科网站页面内容来实现。这些内容不仅包括正文描述的内容，还包括一系列的元信息，例如文章标题，分类信息，时间信息，关键词等。

在采集到相应的数据后，还需通过**数据预处理**来提升数据质量和可用性。在构建文本型知识库时，数据预处理主要包括数据清洗和文本分块两个过程。**数据清洗**旨在清除文本中的干扰元素，如特殊字符、异常编码和无用的 HTML 标签，以及删除重复或高度相似的冗余文档，从而提高数据的清晰度和可用性。**文本分块**是将长文本分割成较小文本块的过程，例如把一篇长文章分为多个短段落。对长文本进行分块有两个好处：一是为了适应检索模型的上下文窗口长度限制，避免超出其处理能力；二是通过分块可以减少长文本中的不相关内容，降低噪音，从而提高检索的效率和准确性。

文本分块的效果直接影响后续检索结果的质量 [14]。如果分块处理不当，可能会破坏内容的连贯性。因此，制定合适的分块策略至关重要，包括确定切分方法（如按句子或段落切分）、设定块大小，以及是否允许块之间有重叠。文本分块的具体实施流程通常开始于将长文本拆解为较小的语义单元，如句子或段落。随后，这些单元被逐步组合成更大的块，直到达到预设的块大小，构建出独立的文本片段。为了保持语义连贯性，通常还会在相邻的文本片段之间设置一定的重叠区域。

2. 知识库增强

知识库增强是通过改进和丰富知识库的内容和结构，以提升其质量和实用性。这一过程通常涉及**查询生成**与**标题生成** [56] 等多个步骤，以此为文档建立语义“锚点”，方便检索时准确定位到相应文本。

查询生成指的是利用大语言模型**生成与文档内容紧密相关的伪查询**。这些伪查询从查询的角度来表达文档的语义，可以作为相关文档的“键”，供检索时与用户查询进行匹配。通过这种方式，可以增强文档与用户查询的匹配度。例如，对于一篇介绍考拉和树袋熊关系的文档，生成的查询“考拉和树袋熊之间的关系是什么?”不仅准确反映了文档的主题，还能有效引导检索器更精确的检索到与用户提问相关的信息。

标题生成指的是利用大语言模型**为没有标题的文档生成合适的标题**。这些生成的标题提供了文档的关键词和上下文信息，能来用来帮助快速理解文档内容，并在检索时更准确地定位到与用户提问相关的信息。对于那些原始文档中缺乏标题的情况，通过语言模型生成标题显得尤为重要。

6.3.2 查询增强

知识库涵盖的知识表达形式是有限的，但用户的提问方式却是千人千面的。用户遣词造句的方式以及描述问题的角度可能会与知识库中的存储的文本间存在差异，这可能导致用户查询和知识库之间不能很好匹配，从而降低检索效果。为了解决此问题，我们可以对用户查询的语义和内容进行扩展，即查询增强，以更好的匹配知识库中的文本。本小节将从**查询语义增强**和**查询内容增强**两个角度出发，对查询增强技术进行简要介绍。

1. 查询语义增强

查询语义增强旨在通过**同义改写**和**多视角分解**等方法来扩展、丰富用户查询

的语义，以提高检索的准确性和全面性。接下来分别对同义改写和多视角分解进行简要介绍。

(1) 同义改写

同义改写通过将原始查询改写成相同语义下不同的表达方式，来解决用户查询单一的表达形式可能无法全面覆盖到知识库中多样化表达的知识。改写工作可以调用大语言模型完成。比如，对于这样一个原始查询：“考拉的饮食习惯是什么?”，可以改写成下面几种同义表达：1、“考拉主要吃什么?”；2、“考拉的食物有哪些?”；3、“考拉的饮食结构是怎样的?”。每个改写后的查询都可独立用于检索相关文档，随后从这些不同查询中检索到的文档集合进行合并和去重处理，从而形成一个更大的相关文档集合。

(2) 多视角分解

多视角分解采用分而治之的方法来处理复杂查询，将复杂查询分解为来自不同视角的子查询，以检索到查询相关的不同角度的信息。例如，对于这样一个问题：“考拉面临哪些威胁?”，可以从多个视角分解为：1、“考拉的栖息地丧失对其有何影响?”；2、“气候变化如何影响考拉的生存?”；3、“人类活动对考拉种群有哪些威胁?”；4、“自然灾害对考拉的影响有哪些?”等子问题。每个子问题能检索到不同的相关文档，这些文档分别提供来自不同视角的信息。通过综合这些信息，语言模型能够生成一个更加全面和深入的最终答案。

2. 查询内容增强

查询内容增强旨在通过生成与原始查询相关的背景信息和上下文，从而丰富查询内容，提高检索的准确性和全面性 [58]。与传统的仅依赖于检索的方式相比，查询内容增强方法通过引入大语言模型生成的辅助文档，为原始查询提供更多维度的信息支持。

生成背景文档是一种查询内容增强的方法。它指的是在原始查询的基础上，利

用大语言模型生成与查询内容相关的背景文档。例如，对于用户查询 “如何保护考拉的栖息地?”，可以生成以下背景文档：

> 考拉是原产于澳大利亚的树栖有袋类动物，主要分布在东部和东南部沿海的桉树林中。这些地区提供了考拉主要食物来源——桉树叶。考拉的栖息地包括开阔的森林和木兰地，这里桉树丰富，不仅提供食物，还提供栖息和保护。考拉高度依赖特定种类的桉树，它们的分布与这些树木的可用性密切相关。

这些生成的背景文档可以作为原始查询的补充信息，提供更多的上下文内容，从而提高检索结果的相关性和丰富性。

6.3.3 检索器

给定知识库和用户查询，检索器旨在找到知识库中与用户查询相关的知识文本。检索器可分为**判别式检索器**和**生成式检索器**两类。本小节将对这两类检索器分别展开介绍。

1. 判别式检索器

判别式检索器通过判别模型对查询和文档是否相关进行打分。判别式检索器通常分为两大类：**稀疏检索器**和**稠密检索器**。稀疏检索器利用离散的、基于词频的文档编码向量进行检索，而稠密检索器则利用神经网络生成的连续的、稠密向量对文档进行检索。下面将详细的介绍这两种检索器以及代表性方法。

(1) 稀疏检索器

稀疏检索器（Sparse Retriever）是指使用**稀疏表示方法**来匹配文本的模型。这类检索器通过统计文档中特定词项出现的统计特征来对文档进行编码，然后基于此编码计算查询与知识库中的文档的相似度来进行检索。典型的稀疏检索技术包

括 TF-IDF[2] 和 BM25[43] 等，它们通过分析词项的分布和频率来评估文档与查询的相关性。TF-IDF 基于词频（TF）和逆文档频率（IDF）来衡量词语在文档或语料库中的重要性，然后用此重要性对文本进行编码。词频（TF）表示词语在文档中的出现频率，计算公式为：

$$\mathrm{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}, \tag{6.1}$$

其中，$n_{i,j}$ 是词语 t_i 在文档 d_j 中的出现次数，$\sum_k n_{k,j}$ 是文档 d_j 中所有词语的出现次数之和，用于进行标准化以避免偏向长文档。逆文档频率（IDF）衡量词语的普遍性，计算公式为：

$$\mathrm{idf}_i = \log \frac{|D|}{|\{j : t_i \in d_j\}|}, \tag{6.2}$$

其中，$|D|$ 是总文档数，$|\{j : t_i \in d_j\}|$ 是包含词语 t_i 的文档数。最终，TF-IDF 值为：

$$\mathrm{tfidf}_{i,j} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i。 \tag{6.3}$$

TF-IDF 通过高词频和低文档频率产生高权重，倾向于过滤常见词语，保留重要词语。

BM25 是一种改进的文本检索算法，它在 TF-IDF 基础上通过文档长度归一化和词项饱和度调整，更精确地评估词项重要性，优化了词频和逆文档频率的计算，并考虑了文档长度对评分的影响。虽然不涉及词项上下文，但是 BM25 在处理大规模数据时表现优异，广泛应用于搜索引擎和信息检索系统。

(2) 稠密检索器

稠密检索器一般利用**预训练语言模型**对文本生成**低维**、**密集**的向量表示，通过计算向量间的相似度进行检索。按照所使用的模型结构的不同，稠密检索器大致可以分为两类：**交叉编码类**（Cross-Encoder）、**双编码器类**（Bi-Encoder）。二者结构分别如图 6.15 (a) 和 6.15 (b) 所示。

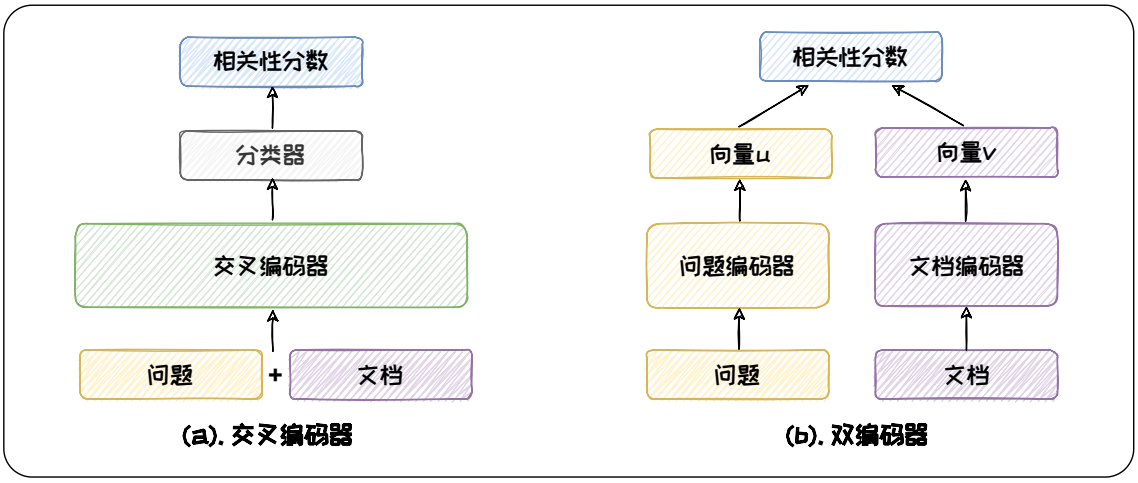


图 6.15: 不同稠密检索器对比图。

交叉编码类

交叉编码类“端到端”的给出查询和文档的相似度。这类模型将查询和文档拼接在一起，随后利用预训练语言模型作为编码器（例如 BERT）生成一个向量表示。接着，通过一个分类器处理这个向量，最终输出一个介于 0 和 1 之间的数值，表示输入的查询和文档之间的相似程度。其优点在于模型结构简单，能够实现查询和文档之间的深度交互，例如，在工作 [12, 44] 中，研究者们使用了交叉编码器来提升检索性能。然而，由于交叉编码类模型需要进行高复杂度的交叉注意力操作，计算量大，因此不适合在大规模检索阶段使用。这种模型更适用于对少量候选文档进行更精确排序的阶段，可以显著提升检索结果的相关性。

双编码器类

与交叉编码类模型不同，双编码类模型采用了一种“两步走”的策略。第一步，查询和文档首先各自通过独立的编码器生成各自的向量表示；第二步，对这两个向量之间的相似度进行计算，以评估它们的相关性。这种方法的优势在于，它允许预先离线计算并存储所有文档的向量表示，在线检索时则可直接进行向量匹配。因此，双编码器非常适合在工业环境中部署，具有极高的匹配效率。然而，在这种分离的处理方式中，查询与文档在提取特征向量时缺乏交互。这可能会对匹配的精

确度产生影响。DPR（Dense Passage Retriever）[23] 是稠密检索器的一个代表工作。其使用两个独立的 BERT 编码器，分别将查询和文档映射到低维特征向量，然后通过向量点积衡量相似度。为了缓解查询与文档缺乏交互的问题，DPR 通过对比学习优化编码器，最大化查询与相关段落相似度的同时最小化与负面段落相似度。

为了缓解查询与文档在提取特征向量时缺乏交互的问题，可以在双编码器的基础上引入查询与文档的交互，以进一步提升双编码器的效果。ColBERT[24] 是其中的代表性方法。其以查询和文档间的 Token 级的相似度为度量，然后通过对比学习对双编码器进行微调，以使双编码器编码的特征向量可以兼顾查询和文档。此外，在 RAG 中，我们有时会对查询加入大段上下文进行增强，如第 3.3.2 节所介绍的情形。此时，查询的长度急剧增长，传统的检索方法可能难以有效的处理这些长查询。为解决此问题，可以采用 Poly-encoder[16]。其模型架构沿用双编码器的形式，但是它使用 m 个向量来捕获长查询的多个特征，而不是像普通双编码器那样只用一个向量来表示整个查询。并且，这 m 个向量随后与文档的向量通过注意力机制进行交互，其中的注意力模块采用查询和文档间的对比学习进行训练。

2. 生成式检索器

生成式检索器通过生成模型对输入查询直接生成相关文档的标识符 [29]。与判别式检索器不断地从知识库中去匹配相关文档不同，生成式检索器直接将知识库中的文档信息记忆在模型参数中。然后，在接收到查询请求时，能够直接生成相关文档的标识符（即 DocID），以完成检索 [48]。生成式检索器通常采用基于 Encoder-Decoder 架构的生成模型，如 T5[41]、BART[27] 等。生成式检索器的训练过程通常分为两个阶段 [29]。在第一阶段，模型通过序列到序列的学习方法，学习如何将查询映射到相关的文档标识符。这一阶段主要通过最大似然估计（MLE）来优化模型，确保生成的文档标识符尽可能准确。在第二阶段，通过数据增强和排名优化进一步提高检索效率和准确性。数据增强主要通过生成伪查询 [51] 或使用文

档片段 [61] 作为查询输入，以增加训练数据的多样性和覆盖面。排名优化则涉及使用特定的损失函数，如对比损失或排名损失，来调整模型生成文档标识符的顺序和相关性，从而更好地匹配查询的需求。

在生成式检索器中，DocID 的设计至关重要。其需要在语义信息的丰富性与标识符的简洁性之间取得平衡。常用的 DocID 形式分为两类：基于数字的 DocID 和基于词的 DocID。基于数字的 DocID 方法使用唯一的数字值或整数字符串来表示文档，虽然构建简单，但在处理大量文档时可能导致标识符数量激增，增加计算和存储负担。相比之下，基于词的 DocID 方法直接从文档的标题、URL 或 N-gram 中提取表示 [9]，能更自然地传达文档的语义信息。通常，标题是最佳选择，因为它提供了文档的宏观概述。但在缺乏高质量标题时，URL 或 N-gram 也可作为有效的替代方案。

尽管生成式检索器在性能上取得了一定的进步，但与稠密检索器相比，其效果仍稍逊一筹。此外，生成式检索器还面临着一系列挑战，包括如何突破模型输入长度的限制、如何有效处理大规模文档以及动态新增文档的表示学习等，这些都是亟待解决的问题。

6.3.4 检索效率增强

知识库中通常包含海量的文本，对知识库中文本进行逐一检索缓慢而低效。为提升检索效率，可以引入向量数据库来实现检索中的高效向量存储和查询 [39]。向量数据库的核心是设计高效的相似度索引算法。本节将简要介绍常用的相似度索引算法，以及用于构建向量数据库的常见软件库。

1. 相似度索引算法

在向量检索中，常用的索引技术主要分成三大类：**基于空间划分的方法**、**基于量化方法**和**基于图的方法**。

基于空间划分的方法将搜索空间划分为多个区域来实现索引，主要包括基于树的索引方法和基于哈希的方法两类。其中，基于树的索引方法通过一系列规则递归地划分空间，形成一种树状结构，每个叶节点代表一个较小区域，区域内的数据点彼此接近。在查询时，算法从树的根节点出发，逐步深入到合适的叶节点，最后在叶节点内部进行数据点的相似度比较，以找到最近的向量。常见的基于树的索引包括 KD 树 [4] 和 Ball 树 [13] 等。而基于哈希的方法（如局部敏感哈希（LSH）[11]）通过哈希函数将向量映射到哈希表的不同桶中，使得相似向量通常位于同一桶内。

基于图的方法通过构建一个邻近图，将向量检索转化为图的遍历问题。这类方法在索引构建阶段，将数据集中的每个向量表示为图中的一个节点，并根据向量间的距离或相似性建立边的连接。不同的图索引结构主要体现在其独特的赋边策略上。索引构建的核心思想源于小世界网络模型，旨在创建一个结构，使得从任意入口点出发，能在较少步数内到达查询点的最近邻。这种结构允许在搜索时使用贪婪算法，逐步逼近目标。然而，图索引设计面临稀疏性和稠密性的权衡：较稀疏的图结构每步计算代价低，而较稠密的图则可能缩短搜索路径。基于图的代表性方法有 NSW[32]、IPNSW[37] 和 HNSW[31] 等。

基于乘积量化的方法通过将高维向量空间划分为多个子空间，并在每个子空间中进行聚类得到码本和码字，以此作为构建索引的基础 [18]。这类方法主要包括训练和查询两个阶段。在训练阶段，该方法学习如何将高维向量最优地量化为码字 ID 序列。通常这个过程涉及将原始空间划分为多个子空间，在每个子空间内进行聚类，并通过聚类中心得到码字和码本。每个子空间内的聚类中心点即为码字，所有码字的集合构成码本。每个训练样本的每个子向量可以都用相应子空间的码字来近似，这样就实现了码字 ID 序列来表示训练样本，达到了数据量化的目的。在查询阶段，系统同样将查询向量划分为子向量，并在每个子空间中找到最近的码

字，得到码字 ID 序列。随后，系统计算查询向量的每个子向量到所有对应子空间的码字的距离，形成距离表。最后，系统利用这个距离表和数据库中每个向量的码字 ID 序列，快速查找并累加各个子向量的对应距离，得到查询向量与数据库向量之间的近似距离。通过对这些距离进行排序，系统最终得到最近邻结果。该方法在减少内存占用和加快距离计算速度方面表现出色，但量化过程中会不可避免地会引入一些误差。在某些需要精度更高的应用场景中，我们可以在 PQ 的基础上进一步进行精确排序，以得到精确的最近邻结果。此外，还有一些乘积量化的优化算法以及和其他索引相结合的算法，如 OPQ[15]、IVFPQ[19] 等。

2. 常见软件库介绍

在前文中，我们已经介绍了向量数据库的核心技术——相似度索引算法。这些算法是实现高效向量检索的关键。接下来，我们将介绍几种常用于构建和管理向量数据库的软件库，它们支持上述的相似性索引算法，是实现高效向量检索的重要工具。

Faiss[4]，由 Meta AI Research 开发，是一个专门优化密集向量相似性搜索和聚类的库。Faiss 提供了多种索引算法选择，这些算法涵盖了基于空间划分、基于量化以及基于图的方法等。这些算法不仅能在 CPU 上运行，部分算法还支持 GPU 加速，从而满足不同应用场景下的性能需求。然而，Faiss 本身并不是一个完整的数据库系统，而是一个功能强大的工具库。它专注于提供高效的索引和搜索功能，但在数据存储、管理、分布式支持和安全性措施等方面，Faiss 的功能相对有限。相比之下，向量数据库是一种更全面的解决方案，它不仅包括相似度索引算法，还整合了数据存储、管理、分布式支持和安全性措施等多方面的功能。截至目前，市场上已有多款成熟的向量数据库，如表6.1所示，它们适用于各种更复杂的 RAG 应用场景。

[4] https://github.com/facebookresearch/faiss

表 6.1: 常见的向量数据库。

向量数据库	URL	GitHub Star
milvus	https://github.com/milvus-io/milvus	28.4K
typesense	https://github.com/typesense/typesense	19.0K
qdrant	https://github.com/qdrant/qdrant	18.9K
chroma	https://github.com/chroma-core/chroma	13.7K
weaviate	https://github.com/weaviate/weaviate	10.4K
pinecone	https://www.pinecone.io/	×

6.3.5 检索结果重排

检索器可能检索到与查询相关性不高的文档。这些文档如果直接输入给大语言模型，可能会引发生成质量的下降。为此，在将其输入给大语言模型之前，我们还需要对其进行进一步的精选。精选的主要途径是对检索到的文档进行重新排序，简称重排，然后从中选择出排序靠前的文档。重排方法主要分为两类：基于交叉编码的方法和基于上下文学习的方法。

1. 基于交叉编码的重排方法

基于交叉编码的重排方法利用**交叉编码器**（Cross-Encoders）来评估文档与查询之间的语义相关性。关于交叉编码的介绍见第6.3.3节。MiniLM-L[5]是应用最为广泛的基于交叉编码的重排开源模型之一。该模型通过减少层数和隐层单元数来降低参数数量，同时采用知识蒸馏技术从大型、高性能的语言模型中继承学习，以此来提高模型性能。此外，还有其他一些在线重排模型可通过 API 直接访问，例如 Cohere[6]。对于希望探索其他高性能的重排器的读者，可以参考 MTEB[7] 排行榜，该榜单汇集了很多性能优秀的重排模型。

2. 基于上下文学习的重排方法

基于上下文学习的方法是指通过设计精巧的 Prompt，使用大语言模型来执行

[5]https://huggingface.co/cross-encoder/ms-marco-MiniLM-L-6-v2

[6]https://cohere.com/rerank

[7]https://huggingface.co/spaces/mteb/leaderboard

RankGPT Prompt 模板

这是RankGPT，一款智能助手，专门用于根据查询的相关性对段落进行排序。

以下是{{num}}段文字，每段都有一个数字标识符[]。我将根据查询内容对它们进行排序：{{query}}

[1] {{passage_1}}
[2] {{passage_2}}
（更多段落）...

查询是：{{query}}

我将根据查询对上述{{num}}段文字进行排序。这些段落将按照相关性降序列出，使用标识符表示，最相关的段落将列在最前面，输出格式应该是[] > [] > 等，例如，[1] > [2] > 等。

对{{num}}段文字的排序结果（仅限标识符）是：

图 6.16: RankGPT Prompt 模板图。

重排任务。这种方法可以利用大语言模型优良的深层语义理解能力，从而取得了良好的表现。RankGPT[47] 是基于上下文学习的重排方法中的代表性方法。其使用的 Prompt 模板如图6.16所示。在重排任务中，输入文档长度有时会超过上下文窗口长度的限制。为了解决该问题，RankGPT 采用了滑动窗口技术来优化排序过程。该技术将所有待排序的文档分割成多个连续的小部分，每个部分作为一个窗口。整个排序过程从文档集的末尾开始：首先，对最后一个窗口内的文档进行排序，并将排序后的结果替换原始顺序。然后，窗口按照预设的步长向前移动，重复排序和替换的过程。这个过程将持续进行，直到所有文档都被处理和排序完毕。通过这种分步处理的方法，RankGPT 能够有效地对整个文档集合进行排序，而不受限于单一窗口所能处理的文档数量。

6.4 生成增强

检索器得到相关信息后，将其传递给大语言模型以期增强模型的生成能力。利用这些信息进行生成增强是一个复杂的过程，不同的方式会显著影响 RAG 的性能。本节将从**如何优化增强过程**这一角度出发，围绕四个方面展开讨论：**（1）何时增强**，确定何时需要检索增强，以确保非必要不增强；**（2）何处增强**，确定在模型中

的何处融入检索到的外部知识，以最大化检索的效用；**（3）多次增强**，如何对复杂查询与模糊查询进行多次迭代增强，以提升 RAG 在困难问题上的效果；**（4）降本增效**，如何进行知识压缩与缓存加速，以降低增强过程的计算成本。

6.4.1 何时增强

大语言模型在训练过程中掌握了大量知识，这些知识被称为**内部知识（Self-Knowledge）**。对于内部知识可以解决的问题，我们可以不对该问题进行增强。不对是否需要增强进行判断而盲目增强，不仅不会改善生成性能，还可能“画蛇添足”引起**生成效率**和**生成质量**上的双下降。对**生成效率**而言，增强文本的引入会增加输入 Token 的数量，增加大语言模型的推理计算成本。另外，检索过程也涉及大量的计算资源。对**生成质量**而言，因为检索到的外部知识有时可能存在噪音，将其输入给大语言模型不仅不会改善大语言模型的生成质量，反而可能会生成错误内容。如图 6.17所示，对于“树袋熊一般在哪里生活?”这个问题，大语言模型可以直接给出正确答案。但是，当我们为它提供一段与树袋熊名称非常相似的动物袋熊的外部知识，如图 6.18所示，大语言模型给出了错误答案，因为大语言模型将知识文本中关于袋熊的信息错误地理解为树袋熊的相关信息。综上，判断大语言模型何时需要检索增强，做到非必要不增强，可以有效的降低计算成本并避免错误增强。

判断是否需要增强的核心在于**判断大语言模型是否具有内部知识**。如果我们判断大模型对一个问题具备内部知识，那么我们就可以避免检索增强的过程，不仅降低了计算成本，而且还可以避免错误增强。判断模型是否具有内部知识的方法可以分为两类：（1）外部观测法，通过 Prompt 直接询问模型是否具备内部知识，或应用统计方法对是否具备内部知识进行估计，这种方法无需感知模型参数；（2）内部观测法，通过检测模型内部神经元的状态信息来判断模型是否存在内部知识，这种方法需要对模型参数进行侵入式的探测。

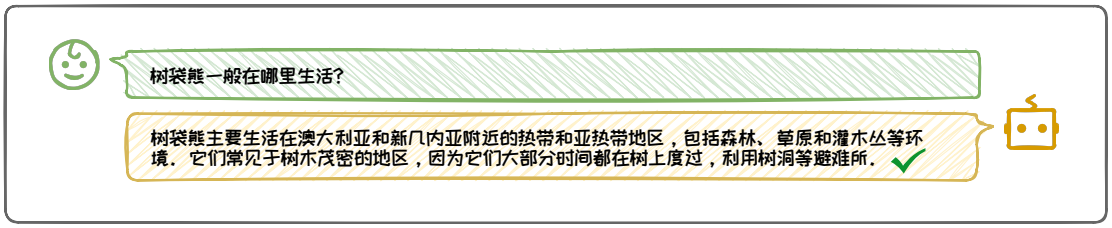


图 6.17: 模型已知问题示例。

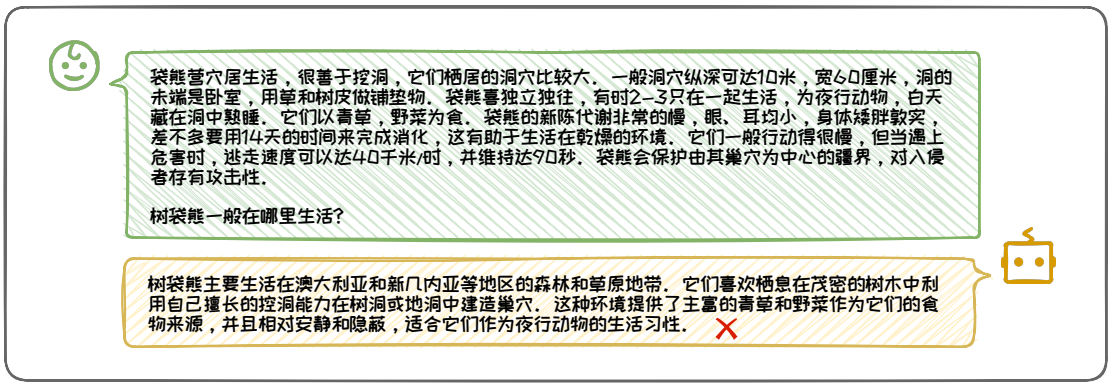


图 6.18: 知识文档损害性能示例。

1. 外部观测法

外部观测法旨在不侵入模型内部参数的情况下，通过直接对大语言模型进行询问或者观测调查其训练数据来推断其是否具备内部知识。这种方法可以类比为人类面试的过程。面试官在评估应聘者的知识和能力时，通常会询问并观察其反应、浏览其过往教育经历等方式，以判断应聘者是否具备足够的专业知识。对于大语言模型，我们可以通过两种问询的方式来判断大语言模型是否具备相应的内部知识：（1）Prompt **直接询问**大语言模型是否含有相应的内部知识；（2）**反复询问**大语言模型同一个问题观察模型多次回答的一致性。此外，我们也可以通过翻看大语言模型的“教育经历”，即**训练数据**来判断其是否具备内部知识。但是，许多大语言模型的训练数据并未公开，无法直接观测它们的训练数据。在这种情况下，可以通过设计**伪训练数据统计量**来拟合真实训练数据的分布，从而间接评估模型对特定知识的学习情况。接下来将对这三种方式进行详细介绍。

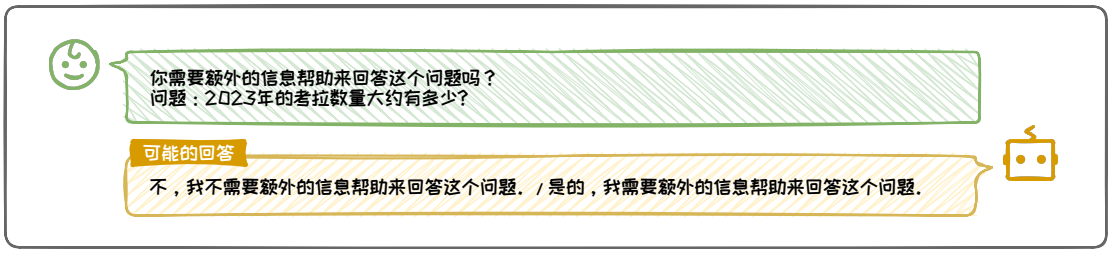


图 6.19: 直接询问的 Prompt 示例。

（1）询问

我们可以通过直接询问和多次询问两种询问方法来判断大语言模型是否具备内部知识。在直接询问时，我们可以编写 Prompt 直接询问大语言模型是否需要外部知识，如图 6.19所示。然而，大语言模型很难做到“知之为知之”，其存在“过度自信”的问题：对于无法回答的问题，大语言模型经常认为自己不需要外部知识的帮助，因此这种判断方式的准确率较低。此外，采用多次询问时，我们可以通过让大语言模型重复多次地回答同一个问题，然后根据其回答的一致性判断其是否具备内部知识 [34, 40]。如果模型不具备相应的内部知识，那么每次的输出会较为随机，从而导致多次回答的一致性不强；反之，如果模型具备内部知识，其每次都会回答正确的知识、随机性较弱，输出则会有更高的一致性。然而，这种方式同样面临着模型因过度自信而“执拗”地一直给出相同的错误答案的情况。并且，多次询问还需要耗费大量的时间和计算资源，在实际使用中可行性较低。

（2）观察训练数据

通过询问来判断内部知识的方法存在的模型回答可靠性较低的问题。我们可以转向观察更为可靠的**训练数据**。大语言模型所具备的内部知识来源于其训练数据。因此，如果模型的训练数据中不包含当前问题的相关信息，那么，模型自然也就未曾习得相应的知识。此外，类比人类的学习模式，我们通常对常见或反复学习的知识掌握程度更好，而对低频的知识则较弱。因此，对于训练数据中出现频率很低的知识，模型对它们的学习程度可能也是比较低的，即**知识在训练数据中的出现**

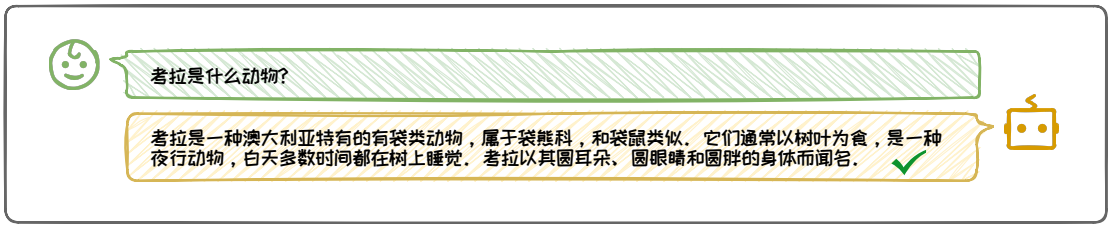


图 6.20: 模型对流行知识的回答示例。

频率与模型对该知识的记忆程度是正相关的 [22]。所以，我们可以通过判断训练数据中是否包含相应的知识来判断模型是否掌握了相应的知识。然而，这种方式存在一定的局限性。首先，由于大语言模型的训练数据规模已经达到了数万亿级别，因此这种统计的手段非常耗时。此外，对于许多商业大语言模型，例如 ChatGPT、GPT4 等，它们的训练数据并不是公开可获取的，在这种情况下，此方案无法执行。

（3）构造伪训练数据统计量

当训练数据不可获取时，我们可以设计伪训练数统计量来拟合训练数据的相关情况。比如，由于模型对训练数据中低频出现的知识掌握不足，而对更“流行”（高频）的知识掌握更好，因此实体的流行度作可以作为伪训练数据统计量。代表性工作 [33] 利用 **Wikipedia 的页面浏览量来衡量实体的流行度，浏览量越大，表明该实体越流行，也就更可能被大语言模型所记忆**。图 6.20和图 6.21分别展示了流行与不流行的知识的例子。从这两个例子中可以发现，对于流行知识**考拉**，模型能够直接给出正确的回答，而对于不流行知识**考拉的基因数量**，模型给出了错误的答案（正确答案应为 26,558）。由此可见，实体的流行度确实在一定程度上反映了模型的内部知识，因此，可以通过设定一个流行度阈值来判别模型是否具备相应的内部知识。然而，这种流行度的定义依赖于数据形成时间，并且未必能精准拟合训练数据的分布，因此存在一定的局限性。

2. 内部观测法

为了进一步深入了解大语言模型的内部知识，在模型参数可访问的情况下，可

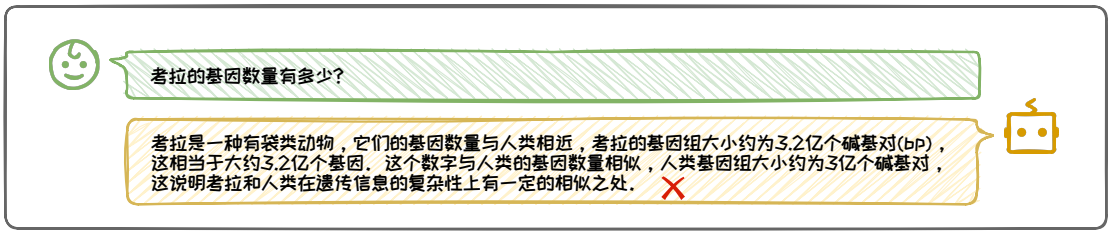


图 6.21: 模型对不流行知识的回答示例。

以通过观测模型内部的隐藏状态来更精确地评估其知识掌握情况。其可以类比人类测谎的过程，科学家通过分析脑电波、脉搏、血压等人类内部状态变化来推断大脑的活动和认知状态，从而判断其是否在说谎或隐藏某些信息。同样地，对于大语言模型，可通过分析模型在生成时每一层的隐藏状态变化，比如注意力模块的输出、多层感知器 (MLP) 层的输出与激活值变化等，来进行评估其内部知识水平。这是因为大语言模型在生成文本时，是对输入序列进行建模和预测，模型内部状态的变化反映了模型对当前上下文理解和下一步预测的确定性。如果模型表现出**较高的内部不确定性**，如注意力分布较为分散、激活值变化较大等，就可能对当前上下文缺乏充分的理解，从而无法做出有把握的预测。

由于**模型的内部知识检索主要发生在中间层的前馈网络中** [36]，因此在处理包含或不包含内部知识的不同问题时，模型的中间层会展现出不同的动态变化。基于这一特性，我们可以训练分类器进行判别，这种方法被称为探针。例如，Liang 等人 [30] 针对三种类型的内部隐藏状态设计了探针实验，分别是注意力层输出（Attention Output）、MLP 层输出（MLP Output）和隐层状态（Hidden States），如图 6.22所示。对于每个输入问题，研究者利用训练好的探针，即线性分类器，来根据问题所对应的内部表示预测该问题是属于模型“已知”（即模型具备相关知识）还是“未知”（即模型缺乏相关知识）。结果显示，不同大语言模型在利用中间层的内部表示进行分类时，均能够实现较高的分类准确率。这表明中间层的内部隐藏状态能够有效地反映模型对问题的理解和相关知识储备。

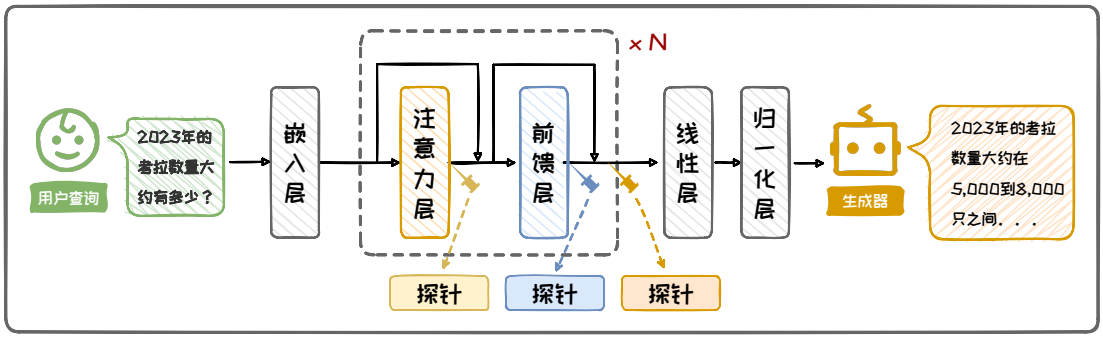


图 6.22: 模型内部状态探针。

然而，这种依赖于内部状态的检测方法也存在一定的局限性，它并不适用于黑盒语言模型，因为我们无法直接访问其内部隐藏状态。另外，模型的输入也需要仔细设计，因为模型有时所展现出的不确定性，可能并非源于对问题的知识缺失，而是问题本身固有的模糊性或歧义性所致。总体而言，基于内部状态评估内部知识的工作目前尚处于初步探索阶段，具体的方案设计有待进一步完善，例如如何构建模型“已知”和“未知”问题的数据集、如何量化内部状态的不确定性、不同内部表示的比对方法，如何设计内部状态检测方案等。我们需要在更广泛的数据集和更多样的模型架构上展开研究，以验证这一方法的有效性和普适性。但是，这是一条充满潜力的新路径，有望为我们从内部视角深入了解大语言模型的知识提供新的视角与方法。

6.4.2 何处增强

在确定大语言模型需要外部知识后，我们需要考虑在何处利用检索到的外部知识，即**何处增强**的问题。得益于大语言模型的上下文学习能力、注意力机制的可扩展性以及自回归生成能力，其输入端、中间层和输出端都可以进行知识融合操作。在输入端，可以将问题和检索到的外部知识拼接在 **Prompt** 中，然后输入给大语言模型；在中间层，可以采用**交叉注意力**将外部知识直接编码到模型的隐藏状

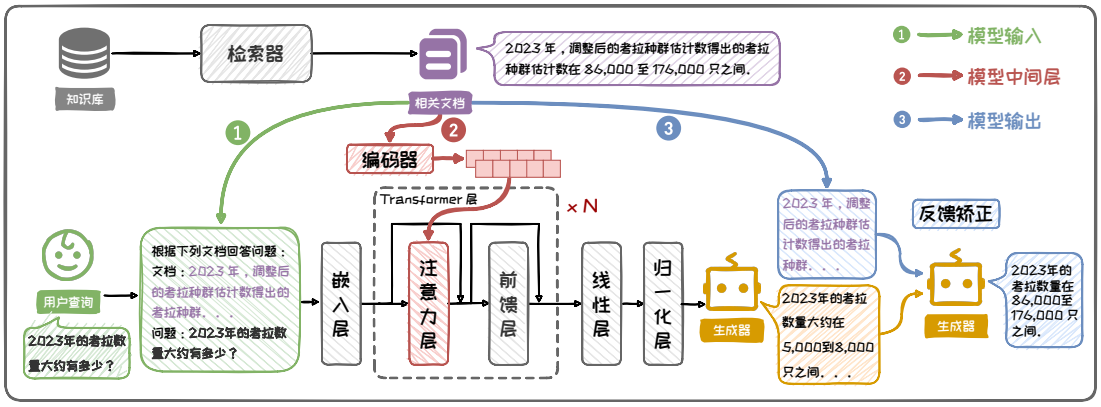


图 6.23: 增强实施位置示意图。

态中；在输出端，可以利用外部知识对生成的文本进行**后矫正**。图 6.23通过一个列子展示了上述三种增强位置的实施方案。三种增强位置分别具有不同的优缺点，适合不同的场景。本小节将对其进行逐一介绍。

（1）在输入端增强

在输入端增强的方法直接将检索到的外部知识文本与用户查询拼接到 Prompt 中，然后输入给大语言模型。其是当前主流的增强方法。此方式的重点在于 Prompt 设计以及检索到的外部知识的排序。良好的 Prompt 设计和外部知识排序，可以使模型更好地理解、利用外部知识。在设计 Prompt 的过程中，可以运用 CoT [54] 等 Prompt 技巧，具体方法可参阅本书第三章的 Prompt 工程内容。

在输入端增强的方法**直观且易于实现**。模型可以直接从输入的上下文中提取到所需信息，无需复杂的处理或转换。然而，当检索到的文本过长时，可能导致输入序列过长，甚至超出模型的最大序列长度限制。这给模型的上下文理解带来挑战，并且还会增加模型推理计算成本、增加其计算负担。这种方法对大语言模型的长文本处理能力和上下文理解能力要求较高。

（2）在中间层增强

在中间层增强增强的方法利用注意力机制的灵活性，先将检索到的外部知识转换为**向量表示**，然后将这些向量插入通过交叉注意力融合到模型的隐藏状态中。

在第 6.2 节中介绍的 Retro [6] 方法，就是采用这种方式的典型代表。这种方法能够**更深入地影响模型的内部表示**，可能有助于模型更好地理解和利用外部知识。同时，由于向量表示通常比原始文本更为紧凑，这种方法可以**减少对模型输入长度的依赖**。然而，这种方法需要对模型的结构进行复杂的设计和调整，无法应用于黑盒模型。

（3）在输出端增强

在输出端增强的方法利用检索到的外部知识对大语言模型生成的文本进行**校准**，是一种**后处理**的方法。在此类方法中，模型首先在无外部知识的情况下生成一个初步回答，然后再利用检索到的外部知识来验证或校准这一答案。校验过程基于生成文本与检索文本的知识一致性对输出进行矫正。矫正可以通过将初步回答与检索到的信息提供给大模型，让大模型检查并调整生成的回答来完成。例如，Yu 等人提出的 REFEED 框架 [59] 是此类方法典型代表。这种方法的优点是可以确保**生成的文本与外部知识保持一致**，提高答案的准确性和可靠性。然而，其效果在很大程度上依赖于检索到的外部知识的质量和相关性。若检索到的文档不准确或不相关，则会导致错误的校准结果。

综上，上述三种增强方式各有优劣，在实际应用中，我们可以根据具体的场景和需求，灵活选择不同的方案。并且，由于上述三种方案是相互独立的，它们也可以组合使用，以实现更优的增强效果。

6.4.3 多次增强

在实际应用中，用户对大语言模型的提问可能是复杂或模糊的。复杂问题往往涉及多个知识点，需要多跳（multi-hop）的理解；而模糊问题往往指代范围不明，难以一次就理解问题的含义。对于复杂问题和模糊问题，我们难以通过一次检索增强就确保生成正确，多次迭代检索增强在所难免。处理复杂问题时，常采用**分解式**

增强的方案。该方案将复杂问题分解为多个子问题，子问题间进行迭代检索增强，最终得到正确答案。处理模糊问题时，常采用**渐进式增强**的方案。该方案将问题的不断细化，然后分别对细化的问题进行检索增强，力求给出全面的答案，以覆盖用户需要的答案。本小节将对这两种方案展开介绍。

1. 分解式增强

复杂问题通常包含多个知识点。例如，在“世界上睡眠时间最长的动物爱吃什么?”这个问题中，包含着“世界上睡眠时间最长的动物是什么?”和“这个动物爱吃什么?”两个知识点。对复杂问题进行作答，需要多跳理解。因此，在复杂问题的检索增强中，我们通常无法仅通过一次检索增强就得到满意的答案。在这种情况下，模型可以将多跳问题分解为一个个子问题，然后在子问题间迭代地进行检索增强，最后得出正确结论，即分解式增强。

DEMONSTRATE–SEARCH–PREDICT（DSP） [25] 是一种具有代表性的分解式增强框架。该框架主要包含以下三个模块：（1）**DEMONSTRATE** 模块，通过上下文学习的方法，将复杂问题分解为子问题；（2）**SEARCH** 模块，对子问题进行迭代检索增强，为最终决策提供综合的外部知识；（3）**PREDICT** 模块，根据 SEARCH 提供的综合外部知识生成最终回答。

如图 6.24所示，以“世界上睡眠时间最长的动物爱吃什么?”为例对 DSP 的流程进行介绍。首先，DEMONSTRATE 模块会在训练集中寻找类似问题的示例，以此来演示问题如何被分解。这些示例将指导大语言模型生成针对子问题的精确查询。例如，第一个子问题是：“世界上睡眠时间最长的动物是什么?”。随后，SEARCH 模块将利用第一个子问题来检索相关信息，从而确定睡眠最长的动物是考拉，并基于这一新信息形成第二个子问题：“考拉爱吃什么?”，随后再次执行检索，以找到描述考拉饮食习惯的相关文本。最终，在 PREDICT 模块中，大语言模型将综合所有信息得出最终答案，即“世界上睡眠时间最长的动物是考拉，爱吃桉树叶。”

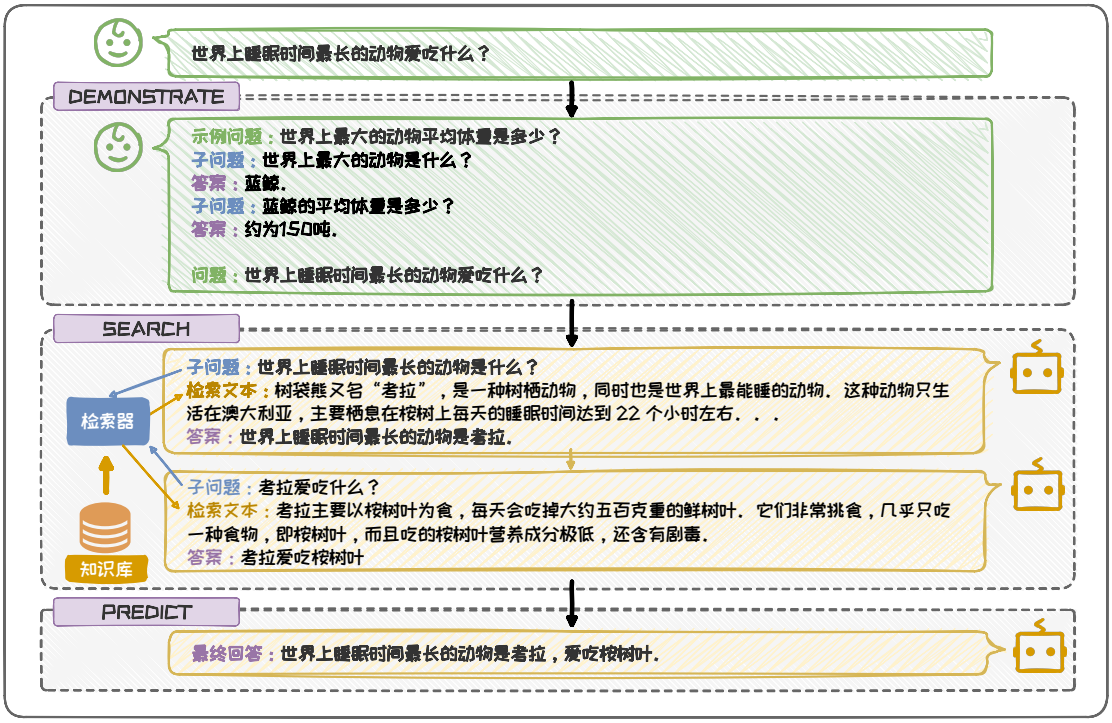


图 6.24: DSP 流程示意图。

分解式增强将复杂问题化整为零，降低了单次检索增强的难度。其性能很大程度上取决于子问题分解的质量。不同领域的复杂问题可能有着不同的分解范式，因此常常需要根据具体任务对问题分解方案进行设计。

2. 渐进式增强

在模糊问题中，问题主体通常指代不明，容易引发歧义。例如，在“国宝动物爱吃什么?”这个问题中，由于不同国家国宝动物不同，且部分国家的国宝动物不止一种，因此我们无法确定询问的是其中的哪一种动物。在处理这样的模糊问题时，我们可以对问题进行渐进式地拆解、细化，然后对细化后的问题进行检索，利用检索到的信息增强大模型。这种渐进式的增强方法可以帮助大语言模型掌握更全面的信息。

TREE OF CLARIFICATIONS（TOC） [26] 是渐进式增强的代表性框架。该框架通过递归式检索来引导大语言模型在**树状结构**中探索给定模糊问题的多种澄

清路径。图 6.25展示了 TOC 框架的整个工作流，我们以“**国宝动物爱吃什么？**”这一问题为例进行说明。TOC 框架首先启动第一轮检索，根据检索到的相关文档和原始问题，生成一系列具体的细化问题，例如：“**中国的国宝动物爱吃什么？**”，“**澳洲的国宝动物爱吃什么？**”。在此过程中，框架会根据细化问题与原问题的相关性及知识一致性进行剪枝。随后，针对每个细化问题，框架独立展开深入的检索并对问题进一步细化，例如：“**澳洲的国宝动物考拉爱吃什么？**”。最终，我们能够构建出多条完整的知识路径，每条路径的末端（叶节点）都代表了对原始问题的不同但是有效的解答。通过整合所有有效的叶节点，我们能够得出一个精确且全面的长回答，例如对此问题我们得到：“**中国的国宝动物熊猫爱吃竹子；澳洲的国宝动物考拉爱吃桉树叶；澳洲的国宝动物袋鼠爱吃杂草和灌木……**”

这种渐进式的检索方法能够显著改善大语言模型对模糊问题的生成效果。然而，其需要进行多轮检索并生成多个答案路径，整个系统的**计算量**会随着问题复杂度呈指数级增长。此外，多轮的检索与生成不仅会带来显著的**时间延迟**，多个推理路径还为推理带来**不稳定性**，容易引发错误。

6.4.4 降本增效

检索出的外部知识通常包含大量原始文本。将其通过 Prompt 输入给大语言模型时，会大幅度增加输入 Token 的数量，从而增加了大语言模型的推理计算成本。此问题可从**去除冗余文本**与**复用计算结果**两个角度进行解决。本小节将对这两个角度分别展开介绍。

1. 去除冗余文本

在 RAG 中，检索出的原始文本通常包含大量的无益于增强生成的冗余信息。这些冗余信息不仅增加了输入 Token 的长度，而且还有可能对大模型产生干扰，导致生成错误答案。去除冗余文本的方法通过对检索出的原始文本的词句进行过

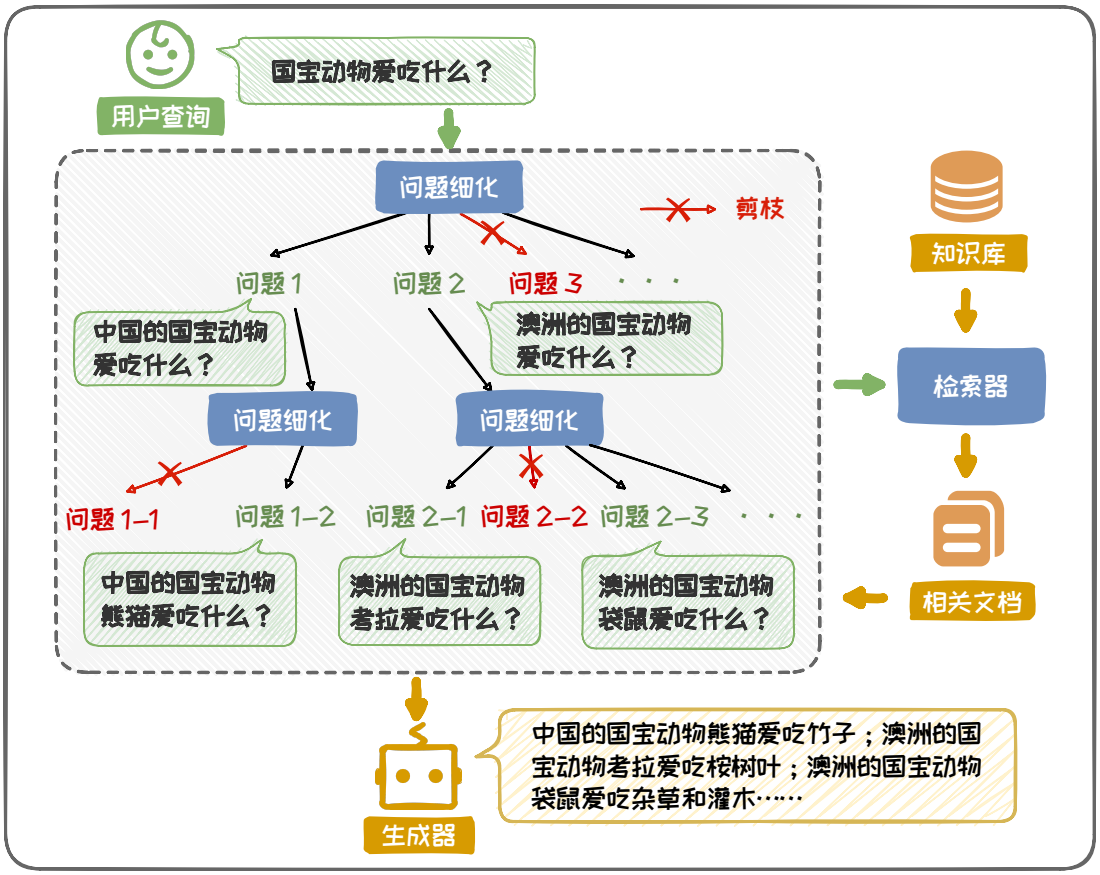


图 6.25: TOC 框架流程示意图。

滤，从中选择出部分有益于增强生成的部分。去除冗余文本的方法主要分为三类：Token 级别的方法，子文本级别的方法以及全文本级别的方法。这些方法通过不同的机制来筛选和优化检索出的原始文本，以减少无益信息，确保生成内容的相关性和准确性。以下分别对这三类方法分别展开介绍。

Token 级别的方法通过对 Token 进行评估，对文本中不必要的 Token 进行剔除。困惑度是判断 Token 重要性的重要指标。直观上说，如果一个 Token 的困惑度低，这意味着模型有很高的概率预测到这个 Token，表明该 Token 易于预测且较为普遍，因此它携带的新信息量较少，可能是冗余的；反之，如果一个 Token 的困惑度高，则表明这个 Token 携带了更多的信息量。**LongLLMLingua** [20] 框架利用小模型评

估 Token 的困惑度，然后基于困惑度删除冗余文本。其首先进行问题感知的**粗粒度压缩**，即在给定问题条件下通过计算文档中所有 Token 困惑度的均值来评估文档的重要性，困惑度越高表示文档信息量越大。随后，执行问题感知的**细粒度压缩**，即进一步计算文档中每个 Token 的困惑度并去除其中低困惑度的 Token。此外，论文还引入了**文档重排序机制**、**动态压缩比率**以及**子序列恢复**机制，确保重要文档和文档中的重要信息被有效利用。

子文本级别的方法通过对子文本进行打分，对不必要的子文本成片删除。**FIT-RAG** [35] 是子文本级别方法中的代表性方法。该方法中采用了预先训练的双标签子文档打分器，从两个维度评估文档的有用性：一是**事实性**，即文档是否包含查询的答案信息；二是**模型的偏好程度**，即文档是否易于模型理解。对于检索到的文档，首先利用滑动窗口将其分割成多个子文档，然后使用双标签子文档打分器对这些子文档分别进行评分。最后，删除掉评分较低的子文档，从而有效地去除冗余文本。

全文本级别的方法直接从整个文档中抽取出重要信息，以去除掉冗余信息。经典方法 **PRCA** [57] 通过训练能够提炼重点内容的信息提取器对文本中的重要信息进行提取。该方法分为两个阶段，上下文提取阶段与奖励驱动阶段。在**上下文提取阶段**，以最小化压缩文本与原输入文档之间的差异为目标，对信息提取器进行监督学习训练，学习如何将输入文档精炼为信息丰富的压缩文本。**在奖励驱动阶段**，大语言模型作为奖励模型，其根据压缩文本生成的答案与真实答案之间的相似度作为奖励信号，通过强化学习对信息提取器进行优化。最终得到的信息提取器可以直接将输入文档转化为压缩文本，端到端地去除冗余文本。

2. 复用计算结果

除了对冗余信息进行筛除，我们还可以对计算必需的中间结果进行复用，以优化 RAG 效率。在大语言模型进行推理的自回归过程中，每个 Token 都需要用到

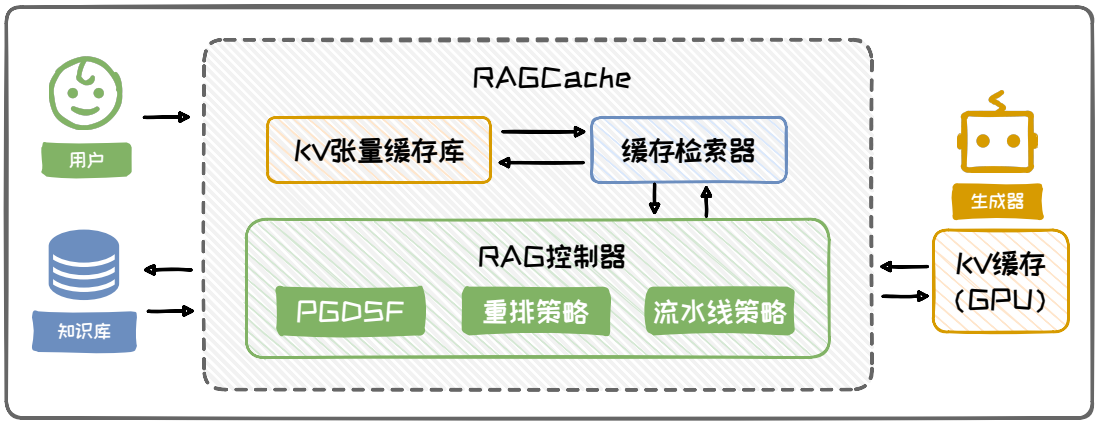


图 6.26: RAGCache 框架流程示意图。

之前 Token 注意力模块涉及的 Key 和 Value 的结果。为了避免对每个 Token 都重新计算前面的 Key 和 Value 的结果，我们可以将之前计算的 Key 和 Value 的结果进行缓存（即 KV-cache），在需要是直接从 KV-cache 中调用相关结果，从而避免重复计算。但是，随着输入文本长度的增加，KV-cache 的 GPU 显存占用会随之剧增，甚至会远远超过模型参数所占用的显存大小 [46]。然而，RAG 的输入通常包含检索出来的文本，导致输入文本很长，导致 RAG 中的 KV-cache 存储成本高昂。

不过，在 RAG 中，不同用户查询经常检索到相同的文本，而且常见的查询通常数量有限。因此，我们可以将常用的重复文本的 KV-cache 进行复用，避免每次查询都对其进行计算，以降低存储成本。基于此，**RAGCache** [21] 设计了一种 RAG 系统专用的多级动态缓存机制，如图 6.26所示。RAGCache 系统由三个核心部分组成：**KV 张量缓存库**、**缓存检索器**与 **RAG 控制器**。其中，**KV 张量缓存库**采用树结构来缓存所计算出的文档 KV 张量，其中每个树节点代表一个文档；**缓存检索器**则负责在缓存库中快速查找是否存在所需的缓存节点；而 **RAG 控制器**作为系统的策略中枢，负责制定核心的缓存策略。具体而言，RAG 控制器首先采用了一种**前缀感知的贪婪双重尺度频率（PGDSF）**替换策略。该策略综合考虑了文档节点的访问频率、大小、访问成本以及最近访问时间，使得频繁使用的文档能够被快速检索

到。随后，**重排策略**通过调整请求的处理顺序，优先处理那些能够更多利用缓存数据的请求，以减少重新计算的需求。这样的请求在队列中享有较高的优先级，从而优化了资源使用。最后，系统还引入了**动态推测流水线策略**，通过并行 KV 张量检索和模型推理的步骤，有效减少了端到端延迟。

综上，通过结合上述**输入 Prompt 压缩**与 **KV-cache 机制**，RAG 框架可以在保持高性能的同时，显著提升其效率。这不仅有助于在资源受限的环境中部署模型，还可以提高模型在实际应用中的响应速度。

6.5 实践与应用

通过引入外部知识，RAG 可以有效的缓解大语言模型的幻觉现象，拓展了大语言模型的知识边界。其优越的性能使引起了广泛关注，成为炙手可热的前沿技术，并在众多应用场景中落地。在本节中，我们将探讨搭建简单 RAG 系统的方法，以及 RAG 的两类典型应用。

6.5.1 搭建简单 RAG 系统

为了助力开发者们高效且便捷地构建 RAG 系统，当前已有诸多成熟开源框架可供选择，其中最具代表性的便是 LangChain [8] 与 LlamaIndex [9]。这些框架提供了一系列完备的工具与接口，使得开发者们能够轻松地将 RAG 系统集成到他们的应用中，例如聊天机器人、智能体（Agent）等。接下来，我们将首先简要概述这两个框架的特色及其核心功能，然后讲解如何利用 LangChain 来搭建一个简单的 RAG 系统。

[8]https://www.langchain.com

[9]https://www.llamaindex.ai

1. LangChain 与 LlamaIndex

（1）LangChain

LangChain 旨在简化利用大语言模型开发应用程序的整个过程。它提供了一系列模块化的组件，帮助开发者部署基于大语言模型的应用，其中就包括 RAG 框架的构建。LangChain 主要包含六大模块：**Model IO**、**Retrieval**、**Chains**、**Memory**、**Agents 和 Callbacks**。其中 Model IO 模块包含了各种大模型的接口以及 Prompt 设计组件，Retrieval 模块包含了构建 RAG 系统所需要的核心组件，包括**文档加载**、**文本分割**、**向量构建，索引生成以及向量检索**等，还提供了非结构化数据库的接口。而 Chains 模块可以将各个模块链接在一起逐个执行，Memory 模块则可以存储对话过程中的数据。此外，Agent 模块可以利用大语言模型来自动决定执行哪些操作，Callback 模块则可以帮助开发者干预和监控各个阶段。总体而言，**LangChain 提供了一个较为全面的模块支持**，帮助开发者们轻松便捷地构建自己的 RAG 应用框架。

（2）LlamaIndex

与 LangChain 相比，LlamaIndex 更加专注于**数据索引与检索**的部分。这一特性使得开发者能够**迅速构建高效的检索系统**。LlamaIndex 具备从多种数据源（如 API、PDF 文件、SQL 数据库等）中提取数据的能力，并提供了一系列高效的工具来对这些数据进行向量化处理和索引构建。在数据查询方面，LlamaIndex 同样提供了高效的检索机制。此外，在获取到上下文信息后，LlamaIndex 还支持对这些信息进行过滤、重新排序等精细化操作。值得一提的是，LlamaIndex 框架还能够与 LangChain 框架相结合，从而实现更加多样化的功能。总体而言，**LlamaIndex 侧重于索引与检索，在查询效率上的表现更为突出**，非常适用于在大数据量的场景下构建更为高效的 RAG 系统。

2. 基于 LangChain 搭建简单 RAG 系统

本小节将以 LangChain 框架为例，参考其官方文档[10]，演示如何快速搭建一套简单的 RAG 系统。

1）安装与配置：首先，需要在环境中安装 LangChain 框架及其依赖项。

```
# 安装LangChain框架及其依赖项
!pip install langchain langchain_community langchain_chroma
```

2）数据准备与索引构建：接下来，我们需要准备数据并构建索引。LangChain 的 DocumentLoaders 中提供了种类丰富的文档加载器，例如，我们可以使用 WebBaseLoader 从网页中加载内容并将其解析为文本。

```
from langchain_community.document_loaders import WebBaseLoader
# 使用WebBaseLoader加载网页内容:
loader = WebBaseLoader("https://example.com/page")
docs = loader.load()
```

加载完成后，由于加载的文档可能过长，不适合模型的上下文窗口，需要将文档分割成合适的大小。LangChain 提供了 TextSplitter 组件来实现文档分割。

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
# 使用TextSplitter将长文档分割成更小的块，其中chunk_size表示分割文档的长度，
    chunk_overlap表示分割文档间的重叠长度
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap
    =200)
splits = text_splitter.split_documents(docs)
```

接下来我们需要对分割后的文本块进行索引化，以便后续进行检索。这里我们可以调用 Chroma 向量存储模块和 OpenAIEmbeddings 模型来存储和编码文档。

[10]https://python.langchain.com/v0.2/docs/tutorials/rag/

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
# 使用向量存储(如Chroma)和嵌入模型来编码和存储分割后的文档
vectorstore = Chroma.from_documents(documents=splits, embedding=OpenAIEmbeddings
    ())
```

3）RAG 系统构建：在构建好知识源之后，接下来可以开始构建一个基础的 RAG 系统。该系统包括检索器与生成器两部分，具体工作流程如下：对于用户输入的问题，检索器首先搜索与该问题相关的文档，接着将检索到的文档与初始问题一起传递给生成器，即大语言模型，最后将模型生成的答案返回给用户。

首先进行检索器的构建，这里我们可以基于 VectorStoreRetriever 构建一个 Retriever 对象，利用向量相似性进行检索。

```
# 创建检索器
retriever = vectorstore.as_retriever()
```

接下来是生成器部分的构建，这里我们可以使用 ChatOpenAI 系统模型作为生成器。在这一步骤中，需要设置 OpenAI 的 API 密钥，并指定要使用的具体模型型号。例如，我们可以选择使用 gpt-3.5-turbo-0125 模型。

```
import os
os.environ["OPENAI_API_KEY"] = 'xxx'
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-3.5-turbo-0125")
```

随后是输入 Prompt 的设置，LangChain 的 Prompt Hub 中提供了多种预设的 Prompt 模板，适用于不同的任务和场景。这里我们选择一个适用于 RAG 任务的 Prompt。

```
from langchain import hub
# 设置提示模板
prompt = hub.pull("rlm/rag-prompt")
```

最后我们需要整合检索与生成，这里可以使用 LangChain 表达式语言（LangChain Execution Language，LCEL）来方便快捷地构建一个链，将检索到的文档、构建的输入 Prompt 以及模型的输出组合起来。

```
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
# 使用LCEL构建RAG链
rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
# 定义文档格式化函数
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)
# 使用RAG链回答问题
response = rag_chain.invoke("What is Task Decomposition?")
print(response)
```

通过以上步骤，我们可以方便快捷地使用 LangChain 迅速搭建一个基础 RAG 系统。LangChain 提供了一系列强大的工具和组件，使得构建和整合检索与生成过程变得简单而高效。

6.5.2 RAG 的典型应用

本小节将介绍 RAG 系统的两个典型应用案例：**（1）智能体（Agent）**；**（2）垂域多模态模型增强**。

1. 智能体（Agent）

RAG 在 Agent 系统中扮演着重要角色。在 Agent 系统主动规划和调用各种工具的过程中，需要检索并整合多样化的信息资源，以更精确地满足用户的需求。RAG 通过提供所需的信息支持，助力 Agent 在处理复杂问题时展现出更好的性能。图 6.27展示了一个经典的 Agent 框架 [50]，该框架主要由四大部分组成：**配置（Profile）**、**记忆（Memory）**、**计划（Planning）**和**行动（Action）**。其中记忆模块、计划模块与行动模块均融入了 RAG 技术，以提升整体性能。

具体而言，**（1）配置模块**通过设定**基本信息**来定义 Agent 的角色，这些信息可以包括 Agent 的年龄、性别、职业等基本属性，以及反映其个性和社交关系的信息；**（2）记忆模块**存储从环境中学习到的**知识**以及**历史信息**，支持记忆检索、记忆更新和记忆反思等操作，允许 Agent 不断获取、积累和利用知识。在这一模块中，**RAG 通过检索相关信息来辅助记忆的读取和更新**；**（3）计划模块**赋予 Agent 将复杂任务分解为简单的子任务的能力，并根据记忆和行动反馈不断调整。**RAG 在此模块中通过提供相关的信息，帮助 Agent 更合理有效地规划任务**；**（4）行动模块**则负责将 Agent 的计划转化为具体的行动，包括网页检索、工具调用以及多模态输出等，能够对环境或 Agent 自身状态产生影响，或触发新的行动链。在这一模块中，**RAG 通过检索相关信息来辅助 Agent 的决策和行动执行**。通过这种模块化且集成 RAG 技术的方法，Agent 能够更加高效和智能地响应用户需求，展现出更加卓越的性能。

我们以“**我现在在澳大利亚，我想去抱考拉，应该怎么办呢？**”这一具体问题

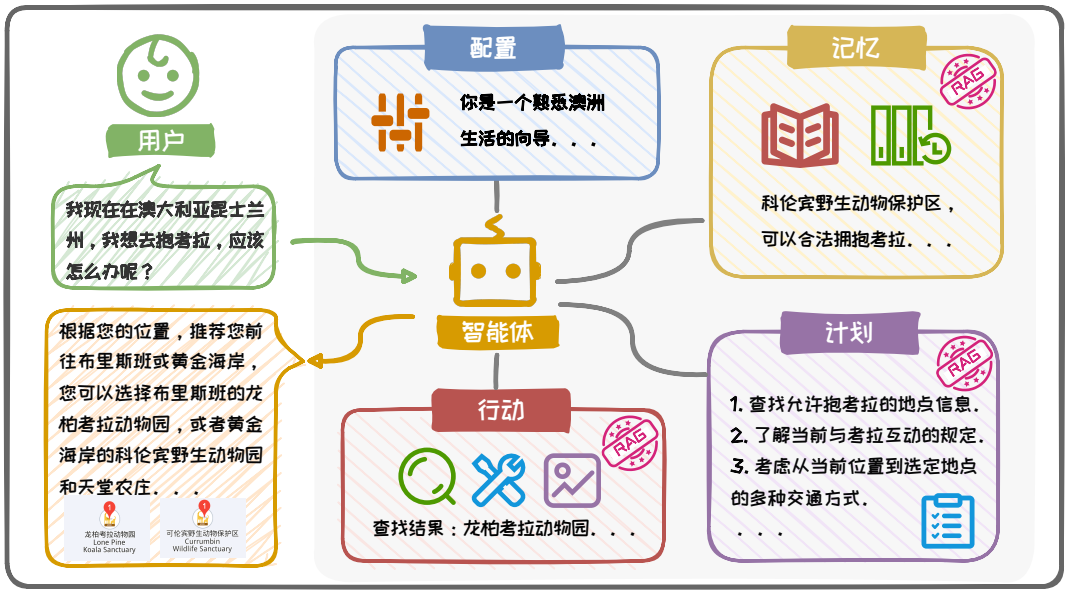


图 6.27: Agent 框架流程示意图。

为例，来展示 Agent 处理问题的流程。在这个例子中，用户提出了一个明确的需求。Agent 将通过以下步骤来完成任务：

- **角色配置**：首先，Agent 利用**配置模块**进行初始化，设定其角色为专业的旅游顾问，明确其任务目标是协助用户实现与考拉亲密接触的愿望；
- **任务规划**：针对用户的需求，**计划模块**规划如何帮助用户实现“抱考拉”的愿望。例如确定当前位置附近存在考拉的地点、了解当地关于与野生动物互动的法律规定、以及前往目的地的方式等。在这一阶段，**RAG** 通过提供相关的旅游景点和法律法规信息，使规划更加精准；
- **信息检索**：**记忆模块**首先在记忆中检索已有的相关信息。如果记忆中不包含所有必要的信息，**行动模块**将激活并调用工具从外部知识源（如旅游网站等）中进行检索。在这一过程中，**RAG** 确保以最高效率检索到与考拉互动最相关的信息；
- **信息整合与决策**：**计划模块**将利用上述检索到的信息制定最佳行动方案。这可能涉及推荐特定的地点、提供出行建议和强调安全须知。**RAG** 在此环节中

整合信息，辅助 Agent 做出明智的决策；

- **信息输出**：最后，**行动模块**将 Agent 确定的行动方案以易于理解的方式输出给用户，包括详细的路线规划以及相关地点的描述、图像等信息。

通过这个例子，我们可以清晰地看到 Agent 框架中的各个模块如何协同工作来完成一个具体的预测任务，其中 RAG 的作用在于快速检索信息并整合知识，为用户提供一个全面而精确的解决方案。

2. 多模态垂直领域应用

在前面的章节中，我们主要聚焦于文本领域的 RAG 系统，而今，在诸多涉及到多模态数据的垂直领域中，RAG 也展现出了广阔的应用前景。例如，在医疗领域中，多模态数据十分普遍，包括 X 光片、MRI、CT 扫描等影像资料，病历、生理监测数据等文本资料。这些数据不仅来源广泛，而且彼此之间存在着复杂的相互联系。因此，在应对这些任务时，RAG 系统必须具备融合与洞察不同模态数据的能力，以确保其精准高效地发挥作用。

目前，已经出现了一些多模态垂直领域的 RAG 应用，例如 Ossowski 等人提出的医学领域多模态检索框架 [38]。图 6.28展示了该框架的整个工作流程。此处以**给定一张 X 光照片，询问该照片所对应的可能症状**这一任务为例进行说明。该框架首先**提取图像与文本的特征表示**，深入理解图像内容与问题语义，为检索模块提供丰富的特征向量。随后，该框架精心设计了一个**多模态检索模块**，利用这些特征向量在医学知识库中进行精准检索，从而获取与输入问题最为相关的信息。这些信息可能包括医学案例、症状描述、潜在病因及治疗方案等，它们以文本和图像的多种形式呈现。最终，Prompt 构建模块将这些信息进行整合，辅助模型生成准确的回答。在本例中，模型生成的答案是“心肌肥大”。

除了医疗领域，RAG 在其他垂直领域，如金融 [8]、生物学 [53] 中也展现了其在处理专业信息上的强大能力，显著提升了专业人士的决策效率。在各类任务中，

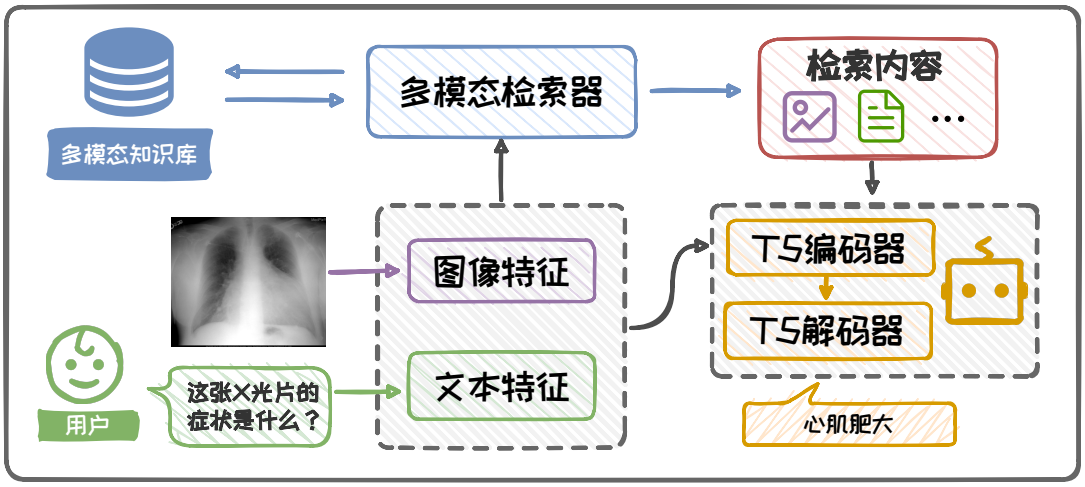


图 6.28: 多模态垂域 RAG 框架示意图。

除了图片信息 [10]，RAG 在音频 [7]、视频 [52] 等其他多模态场景中也同样有着出色的表现。随着技术的进步和数据资源的丰富，我们期待未来 RAG 能够在更多垂直领域、更多数据模态中发挥关键作用。

参考文献

[1] Josh Achiam et al. “Gpt-4 technical report”. In: *arXiv preprint arXiv:2303.08774* (2023).

[2] Akiko Aizawa. “An information-theoretic perspective of tf–idf measures”. In: *IPM*. 2003.

[3] Akari Asai et al. “Self-rag: Learning to retrieve, generate, and critique through self-reflection”. In: *arXiv preprint arXiv:2310.11511* (2023).

[4] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* (1975).

[5] Sebastian Borgeaud et al. “Improving language models by retrieving from trillions of tokens”. In: *ICML*. 2022.

[6] Sebastian Borgeaud et al. “Improving language models by retrieving from trillions of tokens”. In: *ICML*. 2022.

[7] David M Chan et al. “Using external off-policy speech-to-text mappings in contextual end-to-end automated speech recognition”. In: *arXiv preprint arXiv:2301.02736* (2023).

[8] Jian Chen et al. “FinTextQA: A Dataset for Long-form Financial Question Answering”. In: *arXiv preprint arXiv:2405.09980* (2024).

[9] Jiangui Chen et al. “A Unified Generative Retriever for Knowledge-Intensive Language Tasks via Prompt Learning”. In: *SIGIR*. 2023.

[10] Wenhu Chen et al. “Re-imagen: Retrieval-augmented text-to-image generator”. In: *arXiv preprint arXiv:2209.14491* (2022).

[11] Mayur Datar et al. “Locality-sensitive hashing scheme based on p-stable distributions”. In: *SoCG*. 2004.

[12] Hervé Déjean, Stéphane Clinchant, and Thibault Formal. “A Thorough Comparison of Cross-Encoders and LLMs for Reranking SPLADE”. In: *arXiv preprint arXiv:2403.10407* (2024).

[13] Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. “Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces”. In: *arXiv preprint arXiv:1511.00628* (2015).

[14] Paulo Finardi et al. “The Chronicles of RAG: The Retriever, the Chunk and the Generator”. In: *arXiv preprint arXiv:2401.07883* (2024).

[15] Tiezheng Ge et al. “Optimized product quantization for approximate nearest neighbor search”. In: *CVPR*. 2013.

[16] Samuel Humeau et al. “Poly-encoders: Transformer architectures and pre-training strategies for fast and accurate multi-sentence scoring”. In: *arXiv preprint arXiv:1905.01969* (2019).

[17] Gautier Izacard et al. “Atlas: Few-shot learning with retrieval augmented language models”. In: *Journal of Machine Learning Research* (2023).

[18] Herve Jegou, Matthijs Douze, and Cordelia Schmid. “Product quantization for nearest neighbor search”. In: *IEEE transactions on pattern analysis and machine intelligence* 33.1 (2010), pp. 117–128.

[19] Hervé Jégou et al. “Searching in one billion vectors: re-rank with source coding”. In: *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2011, pp. 861–864.

[20] Huiqiang Jiang et al. “Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression”. In: *arXiv preprint arXiv:2310.06839* (2023).

[21] Chao Jin et al. “RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation”. In: *arXiv preprint arXiv:2404.12457* (2024).

[22] Nikhil Kandpal et al. “Large language models struggle to learn long-tail knowledge”. In: *ICML*. 2023.

[23] Vladimir Karpukhin et al. “Dense passage retrieval for open-domain question answering”. In: *arXiv preprint arXiv:2004.04906* (2020).

[24] Omar Khattab and Matei Zaharia. “Colbert: Efficient and effective passage search via contextualized late interaction over bert”. In: *SIGIR*. 2020.

[25] Omar Khattab et al. “Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp”. In: *arXiv preprint arXiv:2212.14024* (2022).

[26] Gangwoo Kim et al. “Tree of clarifications: Answering ambiguous questions with retrieval-augmented large language models”. In: *EMNLP*. 2023.

[27] Mike Lewis et al. “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension”. In: *arXiv preprint arXiv:1910.13461* (2019).

[28] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *NeurIPS*. 2020.

[29] Xiaoxi Li et al. “From matching to generation: A survey on generative information retrieval”. In: *arXiv preprint arXiv:2404.14851* (2024).

[30] Yuxin Liang et al. “Learning to trust your feelings: Leveraging self-awareness in llms for hallucination mitigation”. In: *arXiv preprint arXiv:2401.15449* (2024).

[31] Yu A Malkov and Dmitry A Yashunin. “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs”. In: *IEEE transactions on pattern analysis and machine intelligence* (2018).

[32] Yury Malkov et al. “Approximate nearest neighbor algorithm based on navigable small world graphs”. In: *Information Systems* 45 (2014), pp. 61–68.

[33] Alex Mallen et al. “When not to trust language models: Investigating effectiveness of parametric and non-parametric memories”. In: *ACL*. 2023.

[34] Potsawee Manakul, Adian Liusie, and Mark JF Gales. “Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models”. In: *EMNLP*. 2023.

[35] Yuren Mao et al. “FIT-RAG: Black-Box RAG with Factual Information and Token Reduction”. In: *ACM Transactions on Information Systems* (2024).

[36] Kevin Meng et al. “Locating and editing factual associations in GPT”. In: *NeurIPS*. 2022.

[37] Stanislav Morozov and Artem Babenko. “Non-metric similarity graphs for maximum inner product search”. In: *Advances in Neural Information Processing Systems* 31 (2018).

[38] Timothy Ossowski and Junjie Hu. “Multimodal prompt retrieval for generative visual question answering”. In: *ACL*. 2023.

[39] James Jie Pan, Jianguo Wang, and Guoliang Li. “Survey of vector database management systems”. In: *arXiv preprint arXiv:2310.14021* (2023).

[40] Ella Rabinovich et al. “Predicting Question-Answering Performance of Large Language Models through Semantic Consistency”. In: *arXiv preprint arXiv:2311.01152* (2023).

[41] Colin Raffel et al. “Exploring the limits of transfer learning with a unified text-to-text transformer”. In: *The Journal of Machine Learning Research* (2020).

[42] Ori Ram et al. “In-context retrieval-augmented language models”. In: *Transactions of the Association for Computational Linguistics* (2023).

[43] Stephen Robertson, Hugo Zaragoza, et al. “The probabilistic relevance framework: BM25 and beyond”. In: *Foundations and Trends® in Information Retrieval* (2009).

[44] Ferdinand Schlatt, Maik Fröbe, and Matthias Hagen. “Investigating the Effects of Sparse Attention on Cross-Encoders”. In: *ECIR*. 2024.

[45] Weijia Shi et al. “Replug: Retrieval-augmented black-box language models”. In: *arXiv preprint arXiv:2301.12652* (2023).

[46] Hanshi Sun et al. “Triforce: Lossless acceleration of long sequence generation with hierarchical speculative decoding”. In: *arXiv preprint arXiv:2404.11912* (2024).

[47] Weiwei Sun et al. “Is chatgpt good at search? investigating large language models as re-ranking agent”. In: *arXiv preprint arXiv:2304.09542* (2023).

[48] Yi Tay et al. “Transformer memory as a differentiable search index”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 21831–21843.

[49] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).

[50] Lei Wang et al. “A survey on large language model based autonomous agents”. In: *Frontiers of Computer Science* 18 (2024), p. 186345.

[51] Yujing Wang et al. “A Neural Corpus Indexer for Document Retrieval”. In: *ArXiv* abs/2206.02743 (2022). URL: https://api.semanticscholar.org/CorpusID:249395549.

[52] Zhenhailong Wang et al. “Language models with image descriptors are strong few-shot video-language learners”. In: *NeurIPS*. 2022.

[53] Zichao Wang et al. “Retrieval-based controllable molecule generation”. In: *ICLR*. 2022.

[54] Jason Wei et al. “Chain of Thought Prompting Elicits Reasoning in Large Language Models”. In: *NeurIPS*. 2022.

[55] David H Wolpert and William G Macready. “No free lunch theorems for optimization”. In: *IEEE transactions on evolutionary computation* 1 (1997), pp. 67–82.

[56] Mingrui Wu and Sheng Cao. “LLM-Augmented Retrieval: Enhancing Retrieval Models Through Language Models and Doc-Level Embedding”. In: *arXiv preprint arXiv:2404.05825* (2024).

[57] Haoyan Yang et al. “Prca: Fitting black-box large language models for retrieval question answering via pluggable reward-driven contextual adapter”. In: *EMNLP*. 2023.

[58] Wenhao Yu et al. “Generate rather than retrieve: Large language models are strong context generators”. In: *ICLR*. 2023.

[59] Wenhao Yu et al. “Improving Language Models via Plug-and-Play Retrieval Feedback”. In: *arXiv preprint arXiv:2305.14002* (2023).

[60] Zichun Yu et al. “Augmentation-Adapted Retriever Improves Generalization of Language Models as Generic Plug-In”. In: *arXiv preprint arXiv:2305.17331* (2023).

[61] Yujia Zhou et al. “DynamicRetriever: A Pre-training Model-based IR System with Neither Sparse nor Dense Index”. In: *ArXiv* abs/2203.00537 (2022). URL: https://api.semanticscholar.org/CorpusID:247187834.