FOUNDATIONS OF LARGE LANGUAGE MODELS

Written by Yuren Mao, Yunjun Gao, et al.

The limits of my language mean the limits of my world.

——Ludwig Wittgenstein

*The authors: Chapter 1: Yuren Mao, Yunjun Gao; Chapter 2: Jiahui Li, Yuren Mao, Yu Mi; Chapter 3: Chao Zhang, Yuren Mao, Zhonghao Hu; Chapter 4: Yuhang Ge, Yuren Mao; Chapter 5: Yu Mi, Yijiang Fan, Yuren Mao; Chapter 6: Xuemei Dong, Wenyi Xu, Yuren Mao.

*This book is open sourced on GIT Hub：

https://github.com/ZJU-LLMs/Foundations-of-LLMs。

Content

Chapter 1 Fundamentals for Language Models **1**

1.1 Statistical Methods-Based Language Models 3

1.1.1 n-grams Language Model 3

1.1.2 Statistical Principles of n-grams 6

1.2 RNN-Based Language Models 7

1.2.1 Recurrent Neural Network 8

1.2.2 RNN-Based Language Model 11

1.3 Transformer-Based Language Models 14

1.3.1 Transformer 15

1.3.2 Transformer-Based Language Model 19

1.4 Sampling Methods for Language Models 21

1.4.1 Probability Maximization Methods 22

1.4.2 Stochastic Sampling Methods 24

1.5 Evaluation of Language Models 28

1.5.1 Intrinsic Evaluation 28

1.5.2 Extrinsic Evaluation 30

Chapter 2 Architectures for LLMs **37**

2.1 Big Data + Large Models → New Intelligence 39

2.1.1 Big Data + Large Models → Capability Enhancement 40

2.1.2 Big Data + Large Models → Capability Expansion 45

2.2 Overview of Large Language Model Architectures 48

2.2.1 Categories of Mainstream Model Architectures 49

2.2.2 Functional Comparison of Model Architectures 53

2.3 Large Language Models Based on the Encoder-only Architecture 55

2.3.1 Encoder-only Architecture 55

2.3.2 BERT Language Model 56

2.3.3 BERT-derived Language Models 61

2.4 Large Language Models Based on the Encoder-Decoder Architecture . . . 70

2.4.1 Encoder-Decoder Architecture 70
2.4.2 T5 Language Model 72
2.4.3 BART Language Model 76
2.5 Large Language Models Based on Decoder-Only Architecture 79
2.5.1 Decoder-Only Architecture 79
2.5.2 GPT Series Language Models 80
2.5.3 LLAMA Series Language Models 92
2.6 Non-Transformer Architectures 99
2.6.1 State Space Models (SSM) 100
2.6.2 Training-Time Updates for TTT 109

Chapter 3 Prompt Engineering 115
3.1 Introduction to Prompt Engineering 116
3.1.1 Definition of Prompt 117
3.1.2 Definition of Prompt Engineering 118
3.1.3 Tokenization and Vectorization of Prompts 121
3.1.4 Significance of Prompt Engineering 126
3.2 In-Context Learning 128
3.2.1 Definition of In-Context Learning 128
3.2.2 Demonstration Example Selection 133
3.2.3 Factors Affecting Performance 137
3.3 Chain of Thought 140
3.3.1 Definition of Chain-of-Thought Prompt 141
3.3.2 Step-by-Step 143
3.3.2.1 1. Zero-Shot CoT 144
3.3.2.2 2. Auto CoT 145
3.3.3 Think Before Acting 146
3.3.4 Pooling Ideas 148
3.4 Prompt Techniques 149
3.4.1 Standardizing Prompt Writing 150
3.4.2 Effective Question Formulation 158
3.4.3 Appropriate Use of CoT 163
3.4.4 Leveraging Psychological Suggestion 167
3.5 Related Applications 170
3.5.1 Agents Based on Large Language Models 171
3.5.2 Data Synthesis 174

3.5.3 Text-to-SQL 178
3.5.4 GPTs 181

Chapter 4 Parameter-Efficient Fine-Tuning 187
4.1 Introduction to Parameter-Efficient Fine-Tuning 189
4.1.1 Downstream Task Adaptation 189
4.1.2 Parameter-Efficient Fine-Tuning 192
4.1.3 Advantages of Parameter-Efficient Fine-Tuning 194
4.2 Additional Parameter Methods 195
4.2.1 Input-attached Methods 196
4.2.2 Model-attached Methods 198
4.2.3 Output-attached Methods 203
4.3 Parameter Selection Methods 205
4.3.1 Rule-based Methods 205
4.3.2 Learning-based Methods 206
4.4 Low-Rank Adaptation Methods 209
4.4.1 LoRA 209
4.4.2 LoRA Variants 213
4.4.3 Task Generalization Based on LoRA Plugins 216
4.5 Practice and Applications 217
4.5.1 PEFT Practice 218
4.5.2 PEFT Applications 221

Chapter 5 Model Edit 229
5.1 Introduction to Model Editing 231
5.1.1 Model Editing Philosophy 232
5.1.2 Model Editing Definition 233
5.1.3 Properties of Model Editing 235
5.1.4 Common Datasets 239
5.2 Classic Methods of Model Editing 241
5.2.1 External Expansion Methods 242
5.2.2 Internal Modification Methods 248
5.2.3 Method Comparison 255
5.3 Additional Parameter Method: T-Patcher 257
5.3.1 patch location 257
5.3.2 form of patch 259
5.3.3 implementation of patch 260

5.4 location edit: ROME . . . 265
  5.4.1 knowledge storage location . . . 265
  5.4.2 knowledge storage mechanism . . . 270
  5.4.3 precise knowledge editing . . . 272
5.5 Model Edit Application . . . 277
  5.5.1 accurate model update . . . 278
  5.5.2 protecting the right to be forgotten . . . 280
  5.5.3 Enhancing Model Security . . . 282

Chapter 6 Retrieval Augmented Generation 289
6.1 Introduction to Search Enhanced Generation . . . 290
  6.1.1 Retrieve the context of enhanced generation . . . 290
  6.1.2 Composition of retrieval enhancement generation . . . 295
6.2 Retrieving Enhanced Generation Architecture . . . 298
  6.2.1 RAG architecture classification . . . 299
  6.2.2 Blackbox Enhanced Architecture . . . 301
  6.2.3 white-box augmented architecture . . . 305
  6.2.4 Comparison and Analysis . . . 307
6.3 knowledge retrieval . . . 309
  6.3.1 knowledge base construction . . . 310
  6.3.2 query enhancement . . . 312
  6.3.3 retriever . . . 315
  6.3.4 retrieval efficiency enhancement . . . 321
  6.3.5 retrieve results rearranged . . . 324
6.4 Generation Enhancement . . . 326
  6.4.1 When to Enhance . . . 327
  6.4.2 where to enhance . . . 335
  6.4.3 multiple augmentation . . . 338
  6.4.4 cost reduction . . . 342
6.5 Practice and Application . . . 347
  6.5.1 Building a Simple RAG System . . . 347
  6.5.2 Typical Applications of RAG . . . 352

1 Fundamentals for Language Models

Language is a complex system of symbols. Language symbols are typically constructed under the constraints of phonology, morphology, and syntax, and carry different semantics. Language symbols are inherently uncertain. The same semantics can be expressed by symbols composed of different phonology, morphology, and syntax; similarly, symbols composed of the same phonology, morphology, and syntax can express different semantics in different contexts. Therefore, **language is probabilistic**. Moreover, **the probabilistic nature of language is inextricably linked to the probabilistic nature of cognition** [15]. Language Models (LMs) aim to **accurately predict the probability of language symbols**. From a linguistic perspective, language models enable computers to grasp grammar, understand semantics, and perform natural language processing tasks. From a cognitive science perspective, accurately predicting the probability of language symbols enables computers to model cognition and evolve intelligence. From ELIZA [20] to GPT-4 [16], language models have evolved from rule-based models to statistical models, and then to neural network models, gradually transforming from rigid mechanical question-answering programs into versatile multi-task intelligent models with strong generalization capabilities. This chapter will sequentially explain n-grams language models based on statistical methods, language models based on Recurrent Neural Networks

(RNN), and language models based on Transformer. Additionally, this chapter will introduce how to decode the output probability values of language models into target text and how to evaluate the performance of language models.

*This book is continuously updated. The GitHub link is: `https://github.com/ZJU-LLMs/Foundations-of-LLMs`.

1.1 Statistical Methods-Based Language Models

Language models acquire the ability to predict the probability of language symbols by statistically analyzing or learning from a corpus. Typically, statistical language models predict the probability of language symbols by directly counting their frequencies in the corpus. Among them, n-grams are the most representative statistical language models. **The n-grams language model provides the probability of language symbols based on the Markov assumption and the maximum likelihood estimation of discrete variables**. This section first presents the calculation method of the n-grams language model, then discusses how the n-grams language model applies the maximum likelihood estimation of discrete variables on the basis of the Markov assumption to provide the probability of language symbol occurrences.

1.1.1 n-grams Language Model

Let the language symbols containing N elements be represented as $w_{1:N} = \{w_1, w_2, w_3, ..., w_N\}$. $w_{1:N}$ can represent text, audio sequences, or other sequences carrying semantic information. For ease of understanding, this chapter assumes that the language symbols $w_{1:N}$ represent text, with elements $w_i \in w_{1:N}$ representing words, where $i = 1, ..., N$. In real language models, w_i can be other forms such as Tokens. The introduction to Tokens will be provided in Chapter 3.

In the n-grams language model, an n-gram refers to a word sequence of length n. The n-grams language model calculates the probability of the text $w_{1:N}$ appearing by sequentially counting the relative frequencies of n-grams and their corresponding (n-1)-grams in the corpus. The calculation formula is as follows:

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

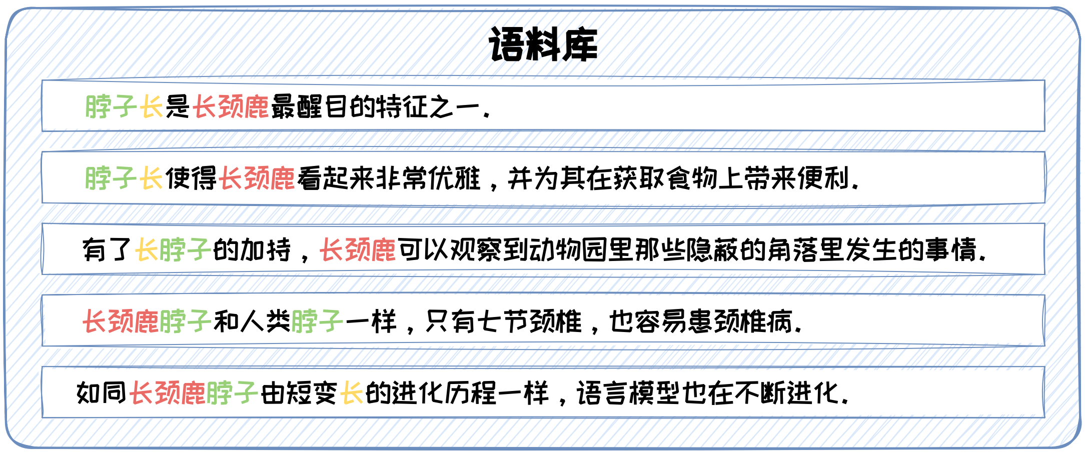
where, $C(w_{i-n+1\,:\,i})$ is the number of occurrences of the word sequence $\{w_{i-n+1}, ..., w_i\}$ in the corpus, and $C(w_{i-n+1\,:\,i-1})$ is the number of occurrences of the word sequence $\{w_{i-n+1}, ..., w_{i-1}\}$ in the corpus. Here, n is a variable; when $n = 1$, it is called unigram, which does not consider the context of the text. In this case, the numerator $C(w_{i-n+1\,:\,i}) = C(w_i)$, where $C(w_i)$ is the number of occurrences of word w_i in the corpus; the denominator $C(w_{i-n+1\,:\,i-1}) = C_{total}$, where C_{total} is the total number of words in the corpus. When $n = 2$, it is called bigrams, which considers the previous word. In this case, the numerator $C(w_{i-n+1\,:\,i}) = C(w_{i-1}, w_i)$, where $C(w_{i-1}, w_i)$ is the number of occurrences of the word sequence $\{w_{i-1}, w_i\}$ in the corpus; the denominator $C(w_{i-n+1\,:\,i-1}) = C(w_{i-1})$, where $C(w_{i-1})$ is the number of occurrences of word w_{i-1} in the corpus. This pattern continues; when $n = 3$, it is called trigrams, which considers the previous two words. When $n = 4$, it is called 4-grams, which considers the previous three words, and so on.

The following example of a bigrams language model demonstrates the specific method by which the n-grams language model calculates the probability of text occurrence. Suppose the corpus contains 5 sentences, as shown in Figure 1.1. Based on this corpus, the probability of the text "长颈鹿脖子长" (composed of the three words {长颈鹿, 脖子, 长}) appearing is calculated using bigrams, as shown in the following equation:

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

In this corpus, $C(\text{长颈鹿}) = 5$, $C(\text{脖子}) = 6$, $C(\text{长颈鹿}, \text{脖子}) = 2$, $C(\text{脖子}, \text{长}) = 2$, hence:

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



语料库

脖子长是长颈鹿最醒目的特征之一.

脖子长使得长颈鹿看起来非常优雅，并为其在获取食物上带来便利.

有了长脖子的加持，长颈鹿可以观察到动物园里那些隐蔽的角落里发生的事情.

长颈鹿脖子和人类脖子一样，只有七节颈椎，也容易患颈椎病.

如同长颈鹿脖子由短变长的进化历程一样，语言模型也在不断进化.

Figure 1.1: Example corpus for n-grams.

In this example, we can see that although ” 长颈鹿脖子长” does not directly appear in the corpus, the bigrams language model can still predict that the probability of ” 长颈鹿脖子长” appearing is $\frac{2}{15}$. This demonstrates that n-grams possess **the ability to generalize to unknown texts**. This is also an advantage over traditional rule-based methods. However, this generalization ability decreases as n increases. Applying trigrams to calculate the probability of the text ” 长颈鹿脖子长” appearing will result in a ”zero probability” situation:

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

Therefore, in the n-grams language model, n represents a trade-off between the ability to fit the corpus and the ability to generalize to unknown texts. When n is too large, it is difficult to find word sequences exactly matching the n-gram in the corpus, potentially leading to a large number of ”zero probability” phenomena. When n is too small, the n-gram cannot carry enough language information to adequately reflect the characteristics of the corpus. Therefore, the value of n is a key factor affecting performance in the n-grams language model. The ”zero probability” phenomenon mentioned above can be improved through smoothing techniques; specific techniques can be found in the literature [11].

This subsection explains how the n-grams language model calculates the probability of language symbol occurrences but does not analyze the principles of the n-grams language model. The next subsection will elaborate on the statistical principles behind the n-grams language model from the perspectives of the n-th order Markov assumption and the maximum likelihood estimation of discrete random variables.

1.1.2 Statistical Principles of n-grams

The n-grams language model is the **maximum likelihood estimation** of the probability of word sequences of length n appearing in the corpus under the **n-th order Markov assumption**. This section first provides the definition of the n-th order Markov assumption (see **Definition 1.1**) and the definition of the maximum likelihood estimation of discrete random variables (see **Definition 1.2**), then analyzes how n-grams apply the maximum likelihood estimation of discrete variables on the basis of the Markov assumption to provide the probability of language symbol occurrences.

ZHEJIANG UNIVERSITY 1897

Definition 1.1 (n-th Order Markov Assumption)

For the sequence $\{w_1, w_2, w_3, ..., w_N\}$, the probability of the current state w_N occurring is only related to the previous n states $\{w_{N-n}, ..., w_{N-1}\}$, i.e.:

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

♣

Definition 1.2 (Maximum Likelihood Estimation of Discrete Random Variables)

Given the distribution law of the discrete random variable X as $P\{X = x\} = p(x;\theta)$, let $X_1, ..., X_N$ be samples from X, and $x_1, ..., x_N$ be the corresponding observed values, with θ as the parameter to be estimated. Under parameter θ, the probability of the distribution function randomly taking values $x_1, ..., x_N$ is:

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

The likelihood function is constructed as:

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

The maximum likelihood estimation of discrete random variables aims to find θ that maximizes $L(\theta|x)$. ♣

Based on the above two definitions, the statistical principles of n-grams are discussed. Let the probability of the text $w_{1:N}$ appearing be $P(w_{1:N})$. According to the chain rule of conditional probability, $P(w_{1:N})$ can be calculated by the following formula:

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

According to the n-th order Markov assumption, the n-grams language model approximates $P(w_i|w_{i-n:i-1})$ to $P(w_i|w_{1:i-1})$.

1.2 RNN-Based Language Models

Recurrent Neural Networks (RNN) are a class of neural networks that **include loops in their network connections**. Given a sequence, the loops in RNN are used to superimpose historical states onto the current state. Along the time dimension, historical states are cyclically accumulated and used as a basis for predicting future states. Therefore, RNN can predict the future based on historical patterns. RNN-based language models take word sequences as input and predict the probability of the next word appearing based on **the con-**

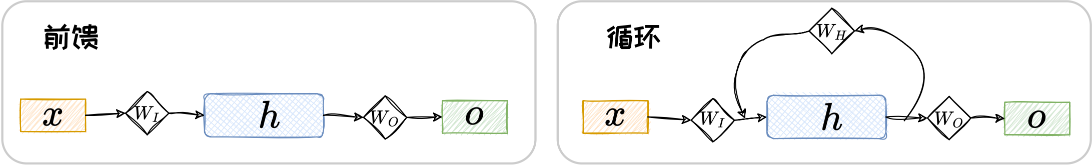


Figure 1.2: Comparison of feedforward and recurrent propagation paradigms.

text encoded by the loop and the current word. This section will first introduce the basic principles of the original RNN, then explain how to use RNN to build a language model.

1.2.1 Recurrent Neural Network

According to the direction of signal flow during inference, the forward propagation paradigm of neural networks can be divided into two categories: feedforward propagation and recurrent propagation. In the feedforward propagation paradigm, computation proceeds layer by layer, ”without looking back.” In the recurrent propagation paradigm, the results of certain layers are fed back through **loops** to previous layers, forming a ”spiral-like progression” paradigm. Neural networks that adopt the feedforward propagation paradigm can be collectively referred to as Feedforward Neural Networks (FNN), while those that adopt the recurrent propagation paradigm are collectively referred to as Recurrent Neural Networks (RNN). Taking a neural network with an input layer, hidden layer, and output layer as an example, Figure 1.2 shows the network structure diagrams of the simplest FNN and RNN, where it can be seen that the FNN network structure only includes forward pathways. In contrast, the RNN network structure includes not only forward pathways but also a **loop** that feeds the result of a layer back to a previous layer.

Let the input sequence be $\{x_1, x_2, x_3, ..., x_t\}$, the hidden states be $\{h_1, h_2, h_3, ..., h_t\}$, and the corresponding outputs be $\{o_1, o_2, o_3, ..., o_t\}$. The network parameters for the input layer, hidden layer, and output layer are W_I, W_H, and W_O, respectively. $g(\cdot)$ is the acti-

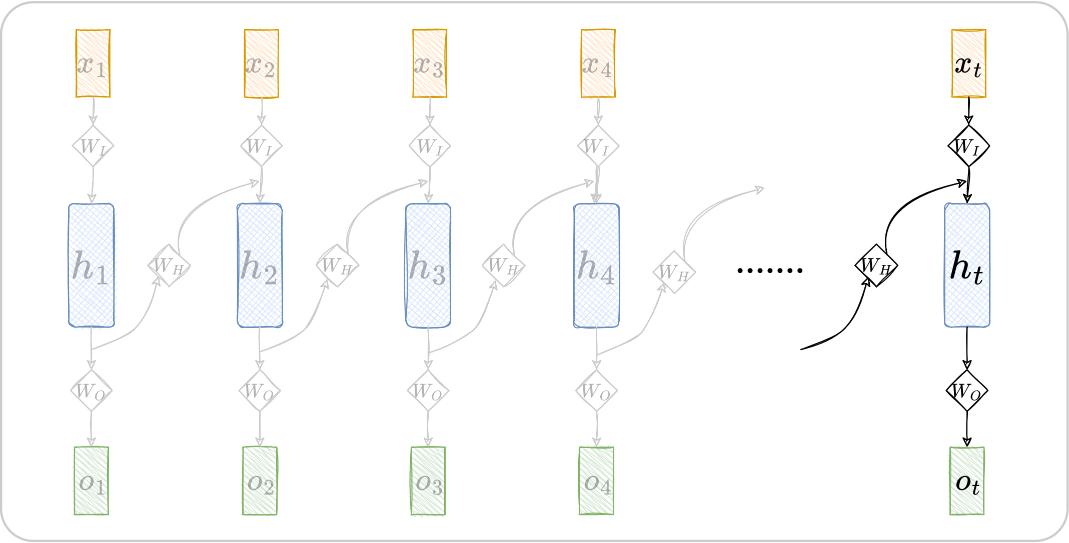


Figure 1.3: Schematic diagram of RNN inference process unfolded along the time dimension.

vation function, and $f(\cdot)$ is the output function. When the input sequence is serially input one element at a time, for FNN, the current output is only related to the current input, i.e.,

$$o_t = f(W_O g(W_I x_t)). \tag{1.9}$$

For simplicity of comparison, the bias term is omitted here. The unique loop structure leads to a completely different inference process for RNN compared to FNN. In the serial input process of RNN, previous elements are cyclically encoded into hidden states and superimposed onto the current input. The output at time t is as follows:

$$\begin{aligned} &h_t = g(W_H h_{t-1} + W_I x_t) = g(W_H g(W_H h_{t-2} + W_I x_{t-1}) + W_I x_t) = \cdots\cdots \\ &o_t = f(W_O h_t). \end{aligned} \tag{1.10}$$

where, $t > 0$, $h_0 = 0$. Unfolding this process along the time dimension, we can obtain the inference process of RNN, as shown in Figure 1.3. Note that the neurons before time t shown in the figure are ”ghosts” of past states and do not actually exist; this unfolding is only for explaining how RNN works.

It can be observed that under the setting of sequentially inputting elements one by one, RNN can **superimpose historical states onto the current state in a cyclic manner** and consider historical information, **showing a spiral-like progression** pattern. However, FNN without loops only considers the current state and cannot take historical states into account. Taking the word sequence {长颈鹿, 脖子, 长} as an example, when given " 脖子" to predict the next word, FNN will only consider " 脖子" for prediction, possibly predicting the next word as " 短", " 疼", etc.; while RNN will consider both " 长颈鹿" and " 脖子", making the probability of predicting the next word as " 长" higher. The introduction of historical information " 长颈鹿" can effectively improve prediction performance.

If FNN wants to consider historical information, it needs to input all elements simultaneously into the model, which will lead to a surge in the number of model parameters. Although the structure of RNN allows it to consider historical information without expanding the number of parameters, this loop structure poses challenges for RNN training. In training RNN, a large number of matrix multiplication operations are involved, which can easily lead to **gradient vanishing** or **gradient explosion** problems. The specific analysis is as follows:

Let the training loss of the RNN language model be:

$$L = L(x, o, W_I, W_H, W_O) = \sum_{i=1}^{t} l(o_i, y_i). \tag{1.11}$$

where, $l(\cdot)$ is the loss function, and y_i is the label.

The gradient of the loss L with respect to the parameter W_H is:

$$\frac{\partial L}{W_H} = \sum_{i=1}^{t} \frac{\partial l_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_i} \cdot \frac{\partial h_i}{\partial W_H}. \tag{1.12}$$

where,

$$\frac{\partial h_t}{\partial h_i} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_{i+1}}{\partial h_i} = \prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}}. \tag{1.13}$$

and,

$$\frac{\partial h_k}{\partial h_{k-1}} = \frac{\partial g(z_k)}{\partial z_k} W_H. \tag{1.14}$$

where, $z_k = W_H h_{k-1} + W_I x_k$. In summary, we have

$$\frac{\partial L}{W_H} = \sum_{i=1}^{t} \frac{\partial l_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial h_t} \cdot \prod_{k=i}^{t} \frac{\partial g(z_k)}{\partial z_k} W_H \cdot \frac{\partial h_i}{\partial W_H}. \tag{1.15}$$

From the above equation, it can be seen that when solving the gradient of W_H, a large number of matrix concatenations are involved, which can lead to the numerical values being cascaded and amplified or reduced. Literature [17] points out that when the maximum eigenvalue of W_H is less than 1, gradient vanishing occurs; when the maximum eigenvalue of W_H is greater than 1, gradient explosion occurs. Gradient vanishing and explosion make training the above RNN very difficult. To solve the gradient vanishing and explosion problems, GRU [4] and LSTM [8] introduce gating structures, achieving good results and becoming mainstream RNN network architectures.

ZHEJIANG UNIVERSITY 1897

1.2.2 RNN-Based Language Model

For the word sequence $\{w_1, w_2, w_3, ..., w_N\}$, the RNN-based language model predicts the probability of the next word w_{i+1} appearing based on the current word w_i and the recurrently input hidden state h_{i-1}, i.e.,

$$P(w_{i+1}|w_{1:i}) = P(w_{i+1}|w_i, h_{i-1}). \tag{1.16}$$

where, when $i = 1$, $P(w_{i+1}|w_i, h_{i-1}) = P(w_2|w_1)$. Based on this, the overall probability of $\{w_1, w_2, w_3, ..., w_N\}$ appearing is:

$$P(w_{1:N}) = \prod_{i=1}^{N-1} P(w_{i+1}|w_i, h_{i-1}). \tag{1.17}$$

In the RNN-based language model, the output is a vector, where each dimension

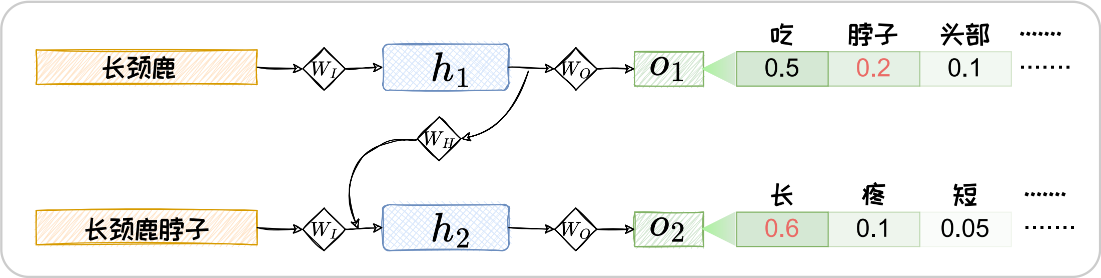


Figure 1.4: Schematic diagram of RNN calculating the probability of word sequence.

represents the probability of the corresponding word in the dictionary. Let the dictionary D contain $|D|$ words $\{\hat{w}_1, \hat{w}_2, \hat{w}_3, ..., \hat{w_{|D|}}\}$, the output of the RNN-based language model can be expressed as $o_i = \{o_i[\hat{w}_d]\}_{d=1}^{|D|}$, where $o_i[\hat{w}_d]$ represents the probability of the word $\hat{w}_d$ in the dictionary appearing. Therefore, for the RNN-based language model, we have

$$P(w_{1:N}) = \prod_{i=1}^{N-1} P(w_{i+1}|w_{1:i}) = \prod_{i=1}^{N} o_i[w_{i+1}]. \tag{1.18}$$

The following example illustrates the above process. Suppose the vocabulary $D = \{$脖子, 头部, 吃, 长, 疼, 吃, 短$\}$, the process of calculating the probability of ” 长颈鹿脖子长” using the RNN-based language model is shown in Figure 1.4.

$$P(\text{长颈鹿脖子长}) = P(\text{脖子}|\text{长颈鹿}) \cdot P(\text{长}|\text{脖子}, h_1) = 0.2 \times 0.6 = 0.12. \tag{1.19}$$

Based on the above pretraining task, the following cross-entropy function can be chosen as the loss function for training the RNN language model.

$$l_{CE}(o_i) = -\sum_{d=1}^{|D|} I(\hat{w}_d = w_{i+1}) \log o_i[w_{i+1}] = -\log o_i[w_{i+1}], \tag{1.20}$$

where, $I(\cdot)$ is the indicator function, which equals 1 when $\hat{w}_d = w_{i+1}$ and 0 when $\hat{w}_d \neq w_{i+1}$.

Let the training set be S, the loss of the RNN language model can be constructed as:

$$L(S, W_I, W_H, W_O) = \frac{1}{N|S|} \sum_{s=1}^{|S|} \sum_{i=1}^{N} l_{CE}(o_{i,s}), \tag{1.21}$$

where, $o_{i,s}$ is the output of the RNN language model when the i-th word of the s-th sample is input. For simplicity of expression, it is assumed that the length of each sample is N. Based on this loss, a computational graph is constructed for backpropagation, and the RNN language model can be trained. After the above training process, we can directly use this model to extract features from sequence data. The extracted features can be used to solve downstream tasks. In addition, we can decode the output of this language model to complete text generation tasks in the ”**autoregressive**” paradigm. In autoregression, in the first round, we first input the first word into the RNN language model, decode it to get an output word. Then, we concatenate the output word of the first round with the input word of the first round as the input of the second round, and decode to get the output of the second round. Next, we concatenate the output and input of the second round as the input of the third round, and so on. Each time, the word predicted in the current round is concatenated with the input of the current round and input into the language model to complete the next round of prediction. In the cyclic iterative ”autoregressive” process, we continuously generate new words, which constitute a piece of text.

However, the above ”autoregressive” process has two problems: (1) **error cascading amplification**, using words generated by the model as input may introduce errors, and such cyclic input of errors will continuously amplify the errors, leading to the model’s inability to fit the training set well; (2) **low serial computation efficiency**, because the next word to be predicted depends on the previous prediction, each prediction is serial and difficult to parallelize. To solve these two problems, ”**Teacher Forcing**” [21] is widely used in the pretraining process of language models. In Teacher Forcing, the output result is always concatenated with the ”ground truth” (Ground Truth) as the input for the next round. In the example shown in Figure 1.4, in the second round of the loop, we use ” 长颈鹿脖子”

to predict the next word ” 长”, rather than choosing the word with the highest probability in o_1, such as ” 吃” or other possible output words.

However, the Teacher Forcing training method leads to the problem of **exposure bias**. Exposure bias refers to the difference between the Teacher Forcing training process and the inference process of the model. In Teacher Forcing training, the model relies on the ”ground truth” for the next prediction, but in inference, the model generates text in an ”autoregressive” manner without ”ground truth” to refer to. Therefore, there is a discrepancy between the training process and the inference process of the model, which may result in poor inference performance. To solve the problem of exposure bias, Bengio et al. proposed the Scheduled Sampling method [2] for RNN. It gradually uses a small portion of the words generated by the model instead of the ”ground truth” in the Teacher Forcing training process, rehearsing the situation of no ”ground truth” during inference in the training process.

1.3 Transformer-Based Language Models

Transformer is a type of neural network structure **modularly** constructed based on the attention mechanism. Given a sequence, Transformer inputs a certain number of historical states and the current state simultaneously, then performs weighted addition. It ”considers” historical states and the current state **comprehensively** and then predicts future states. Transformer-based language models take word sequences as input and predict the probability of the next word appearing based on a certain length of context and the current word. This section will first introduce the basic principles of Transformer, then explain how to use Transformer to build a language model.

1.3.1 Transformer

Transformer is a **modular network structure** composed of two types of modules: (1) Attention module; (2) Fully-connected Feedforward module. The self-attention module consists of a self-attention layer, residual connections, and layer normalization. The fully-connected feedforward module consists of a fully-connected feedforward layer, residual connections, and layer normalization. The structure diagrams of the two modules are shown in Figure 1.5. The principles and functions of each layer are detailed below.

1. Attention Layer

The attention layer uses the idea of weighted averaging to superimpose previous information onto the current state. The attention layer of Transformer encodes the input into query, key, and value parts, i.e., encodes the input $\{x_1, x_2, ..., x_t\}$ into $\{(q_1, k_1, v_1), (q_2, k_2, v_2), ..., (q_t, k_t, v_t)\}$ Here, query and key are used to calculate the attention weights α, and value is the encoding of the input. Specifically,

$$Attention(x_t) = \sum_{i=1}^{t} \alpha_{t,i} v_i. \tag{1.22}$$

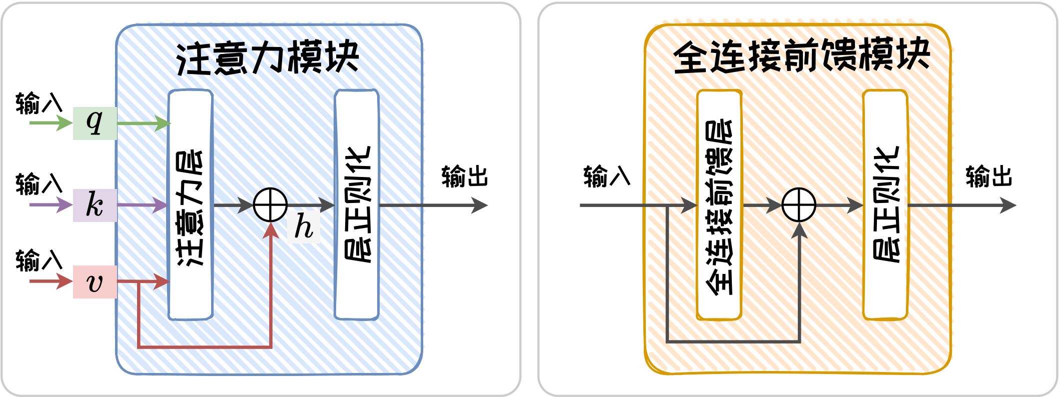


Figure 1.5: Attention module and fully-connected feedforward module.

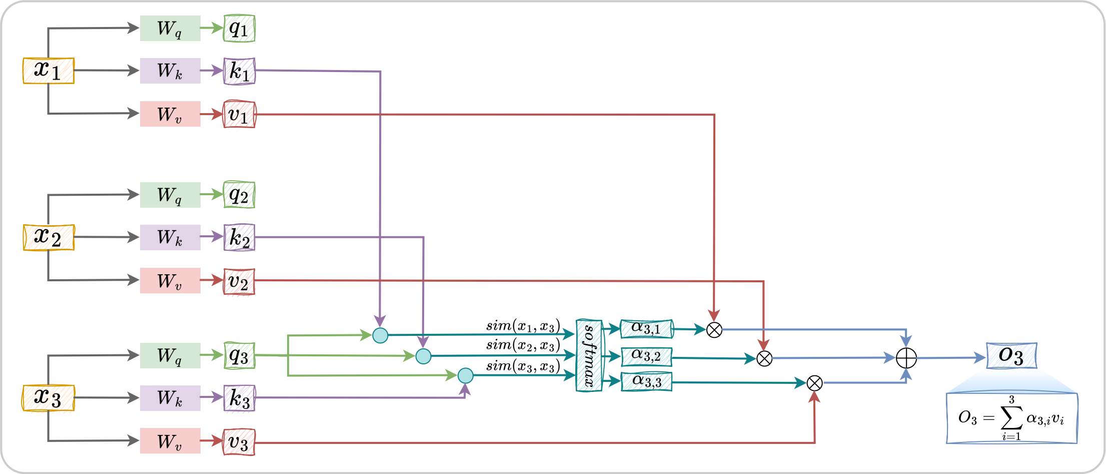


Figure 1.6: Schematic diagram of the attention mechanism.

where,

$$\alpha_{t,i} = softmax(sim(x_t, x_i)) = \frac{sim(q_t, k_i)}{\sum_{i=1}^{t} sim(q_t, k_i)}. \tag{1.23}$$

Here, $sim(\cdot, \cdot)$ is used to measure the degree of correlation between two inputs, and the softmax function is used to normalize this degree of correlation. Additionally,

$$q_i = W_q x_i, \quad k_i = W_k x_i, \quad v_i = W_v x_i, \tag{1.24}$$

where W_q, W_k, and W_v are the parameters of the query, key, and value encoders, respectively. Taking an input containing three elements $\{x_1, x_2, x_3\}$ as an example, the implementation of Transformer self-attention is shown in Figure 1.6.

ZHEJIANG UNIVERSITY 1897

2. Fully-connected Feedforward Layer

The fully-connected feedforward layer accounts for nearly two-thirds of the parameters in Transformer and governs the memory of the Transformer model. It can be seen as a memory storage management module in the Key-Value pattern [7]. The fully-connected feedforward layer consists of two layers with ReLU as the activation function between them. Let the input to the fully-connected feedforward layer be v; it can be represented by the following formula:

$$FFN(v) = \max(0, W_1 v + b_1)W_2 + b_2. \tag{1.25}$$

Here, W_1 and W_2 are the weight parameters of the first and second layers, respectively, and b_1 and b_2 are the bias parameters of the first and second layers, respectively. The first layer can be seen as the key in neural memory, while the second layer can be seen as the value.

3. Layer Normalization

Layer normalization is used to accelerate the training process of neural networks and achieve better generalization performance [1]. Let the vector input to the layer normalization layer be $v = \{v_i\}_{i=1}^{n}$. The layer normalization layer will perform layer normalization operations on each dimension v_i of v. Specifically, the layer normalization operation can be expressed by the following formula:

$$LN(v_i) = \frac{\alpha(v_i - \mu)}{\delta} + \beta. \tag{1.26}$$

Here, α and β are learnable parameters. μ and δ are the mean and variance of the hidden states, respectively, and can be calculated by the following formulas:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} v_i, \quad \delta = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2}. \tag{1.27}$$

4. Residual Connections

The introduction of residual connections can effectively solve the problem of gradient vanishing. There are two residual connections in the basic Transformer encoding module. The first residual connection superimposes the input of the self-attention layer onto the output of the self-attention layer and then inputs it to the layer normalization. The second residual connection leads the input of the fully-connected feedforward layer to the output of the fully-connected feedforward layer and then inputs it to the layer normalization.

The network structure that places layer normalization after residual connections is called Post-LN Transformer. In contrast, there is another network structure that places layer normalization before residual connections, called Pre-LN Transformers. Comparing the two, Post-LN Transformer has stronger capabilities in dealing with representation collapse (Representation Collapse) but slightly weaker in handling gradient vanishing. Pre-LN Transformers can better deal with gradient vanishing but slightly weaker in handling representation collapse. For specific analysis, refer to the literature [7, 22].

The original Transformer adopts an Encoder-Decoder architecture, which includes Encoder and Decoder parts. Both parts are constructed by repeating the self-attention module and the fully-connected feedforward module. The overall structure is shown in Figure 1.7. The Encoder part consists of six cascaded encoder layers, each containing an attention module and a fully-connected feedforward module. The attention module here is a **self-attention module** (the inputs for query, key, and value are the same). The Decoder part consists of six cascaded decoder layers, each containing two attention modules and a fully-connected feedforward module. The first attention module is a **self-attention module**, and the second attention module is a **cross-attention module** (the inputs for query, key, and value are different). The input to the self-attention module of the first decoder layer in the Decoder is the output of the model. The input to the self-attention module of

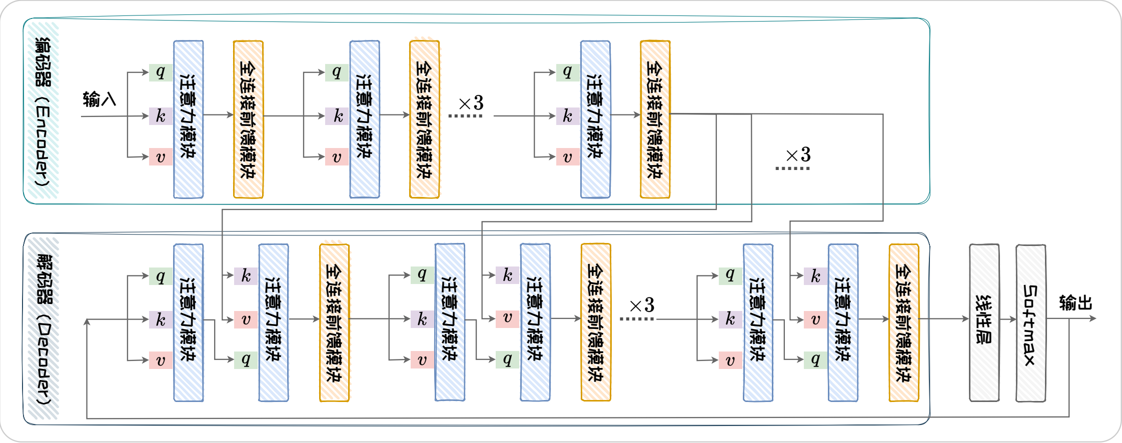


Figure 1.7: Schematic diagram of the Transformer structure.

the subsequent decoder layers is the output of the previous decoder layer. The inputs to the cross-attention module of the Decoder are the output of the self-attention module (query) and the output of the last encoder layer (key, value).

Both the Encoder and Decoder parts of Transformer can be used separately to construct language models, corresponding to Encoder-Only models and Decoder-Only models, respectively. The specific structures of Encoder-Only models and Decoder-Only models will be detailed in Chapter 2.

1.3.2 Transformer-Based Language Model

On the basis of Transformer, various pretraining tasks can be designed to train language models. For example, we can train Encoder-Only language models such as BERT [5] by combining the Encoder part of Transformer with tasks like ”masking and completion”; we can train Encoder-Decoder language models such as T5 [18] by simultaneously applying the Encoder and Decoder parts of Transformer and combining multiple supervised and self-supervised tasks like ”truncation and completion” and ”sequential recovery”; we can train Decoder-Only language models such as GPT-3 [3] by applying the Decoder part of Transformer and using the ”next word prediction” task. These language models will be detailed in Chapter 2. Below, we will briefly introduce the training process of Transformer language models using the next word prediction task as an example.

For the word sequence $\{w_1, w_2, w_3, ..., w_N\}$, the Transformer-based language model predicts the probability of the next word w_{i+1} appearing based on $\{w_1, w_2, ..., w_i\}$. In the Transformer-based language model, the output is a vector, where each dimension represents the probability of the corresponding word in the dictionary. Let the dictionary D contain $|D|$ words $\{\hat{w_1}, \hat{w_2}, ..., \hat{w_{|D|}}\}$. The output of the Transformer-based language model can be expressed as $o_i = \{o_i[\hat{w}_d]\}_{d=1}^{D}$, where $o_i[\hat{w}_d]$ represents the probability of the word $\hat{w}_d$ in the dictionary appearing. Therefore, the prediction of the Transformer-based language model for the overall probability of the word sequence $\{w_1, w_2, w_3, ..., w_N\}$ is:

$$P(w_{1:N}) = \prod_{i=1}^{N-1} P(w_{i+1}|w_{1:i}) = \prod_{i=1}^{N} o_i[w_{i+1}] \tag{1.28}$$

Similar to training RNN language models, Transformer language models often use the following cross-entropy function as the loss function.

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

Here, $I(\cdot)$ is the indicator function, which equals 1 when $\hat{w}_d = w_{i+1}$ and 0 when $\hat{w}_d \neq w_{i+1}$.

Let the training set be S; the loss of the Transformer language model can be constructed as:

$$L(S, W) = \frac{1}{N|S|} \sum_{s=1}^{|S|} \sum_{i=1}^{N} l_{CE}(o_{i,s}) \tag{1.30}$$

Here, $o_{i,s}$ is the output of the Transformer language model when the first i words of sample s are input. Based on this loss, a computational graph is constructed for backpropagation, and the Transformer language model can be trained.

After the above training process, we can use the output of the Encoder as features and apply these features to solve downstream tasks. Additionally, text generation tasks can be completed in the ”autoregressive” paradigm. In autoregression, in the first round, we first input the first word into the Transformer language model, decode it to get an output word. Then, we concatenate the output word of the first round with the input word of the first round as the input of the second round, and decode to get the output of the second round. Next, we concatenate the output and input of the second round as the input of the third round, and so on. Each time, the word predicted in the current round is concatenated with the input of the current round and input into the language model to complete the next round of prediction. In the cyclic iterative ”autoregressive” process, we continuously generate new words, which constitute a piece of text. Similar to training RNN language models, the pretraining process of Transformer models still adopts the ”Teacher Forcing” paradigm mentioned in Section 1.2.2.

Compared to the serial cyclic iteration mode of RNN models, the parallel input characteristic of Transformer makes it easy to perform parallel computation. However, the parallel input paradigm of Transformer also causes the scale of the network model to grow quadratically with the increase in the length of the input sequence. This poses challenges for applying Transformer to process long sequences.

1.4 Sampling Methods for Language Models

The output of a language model is a vector, where each dimension represents the probability of the corresponding word in the dictionary. In text generation tasks using the autoregressive paradigm, the language model generates a sequence of vectors and decodes them into text. The process of decoding these vectors into text is called language model decoding. The decoding process significantly affects the quality of the generated text. Currently, two main types of decoding methods can be summarized as (1) probability maximization methods; (2) stochastic sampling methods. These two methods are introduced in the following sections.

1.4.1 Probability Maximization Methods

Let the dictionary be D, the input text be $\{w_1, w_2, w_3, ..., w_N\}$, and the output vector in the i-th round of autoregression be $o_i = \{o_i[w_d]\}_{d=1}^{|D|}$. The text generated by the model after M rounds of autoregression is $\{w_{N+1}, w_{N+2}, w_{N+3}, ..., w_{N+M}\}$. The probability of the generated document can be calculated as follows:

$$P(w_{N+1:N+M}) = \prod_{i=N}^{N+M-1} P(w_{i+1}|w_{1:i}) = \prod_{i=N}^{N+M-1} o_i[w_{i+1}] \tag{1.31}$$

The goal of probability maximization-based decoding methods is to maximize $P(w_{N+1:N+M})$ to generate the most likely text. The search space size for this problem is M^D, which is an NP-Hard problem. Existing probability maximization methods typically use heuristic search methods. This section introduces two commonly used probability maximization-based decoding methods.

1. Greedy Search

Greedy search selects the word with the highest probability in each round of prediction, i.e.,

$$w_{i+1} = arg\ max_{w \in D}\ o_i[w] \tag{1.32}$$

Greedy search only considers ”immediate benefits” and ignores ”long-term benefits.” **A word with a high probability at the current stage may lead to very low probabilities for subsequent words**. Greedy search easily falls into local optima and is difficult to

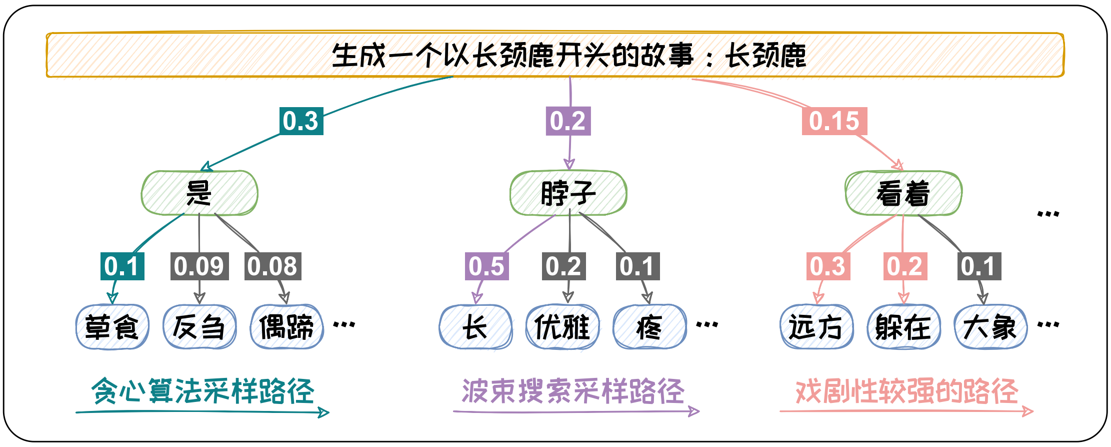


Figure 1.8: Comparison of greedy search and beam search, and potential issues with probability maximization decoding.

achieve a global optimum. Taking Figure 1.8 as an example, when the input is " 生成一个以长颈鹿开头的故事：长颈鹿" (Generate a story starting with a giraffe: giraffe), the word with the highest probability for the first prediction is " 是" (is), with a probability of 0.3. However, after selecting " 是" (is), the probabilities of other words are relatively low. If we follow the greedy search method, the final output we get is " 是草食" (is herbivorous). Its probability is only 0.03. If we choose the word with the second highest probability, " 脖子" (neck), and then select " 长" (long) for the second word, the final probability can reach 0.1. From this example, it can be seen that greedy search is prone to falling into local optima when solving for the maximum probability. To alleviate this problem, beam search can be used for decoding.

2. Beam Search

Beam search keeps the b words with the highest probabilities $\{w_{i+1}^1, w_{i+1}^2, ..., w_{i+1}^b\}$ in each round of prediction, i.e.,

$$min\{o_i[w]\ for\ w \in B_i\} > max\{o_i[w]\ for\ w \in D - B_i\}. \tag{1.33}$$

At the end of the search, M sets are obtained, i.e., $\{B_i\}_{i=1}^M$. The optimal combination is found to maximize the joint probability, i.e.,

$$\{w_{N+1}, ..., w_{N+M}\} = arg\ max_{\{w^i \in B_i\ for\ 1 \leq i \leq M\}} \prod_{i=1}^{M} o_{N+i}[w^i]. \tag{1.34}$$

Continuing with the example of ”生成一个以长颈鹿开头的故事” (Generate a story starting with a giraffe), from Figure 1.8, we can see that if we use a beam search method with $b = 2$, we can get four candidate combinations: ” 是草食” (is herbivorous), ” 是反刍” (is ruminant), ” 脖子长” (neck long), and ” 脖子优雅” (neck elegant), with corresponding probabilities of 0.03, 0.027, 0.1, and 0.04, respectively. We can easily choose the combination with the highest probability, ” 脖子长” (neck long).

However, the text with the highest probability **is usually the most common text**. These texts tend to be somewhat mediocre. In open-ended text generation, both greedy search and beam search are prone to generating ”nonsense literature” — repetitive and mediocre texts. The generated texts lack diversity [19]. As shown in the example in Figure 1.8, the method with the highest probability would generate ” 脖子长” (neck long). Texts like ” 长颈鹿脖子长” (giraffe neck long) have low novelty. To increase the novelty of the generated text, we can add some randomness in the decoding process. This way, we can decode some uncommon combinations, making the generated text more creative and suitable for open-ended text tasks. The method of adding randomness in the decoding process is called stochastic sampling. The next section will introduce stochastic sampling methods.

1.4.2 Stochastic Sampling Methods

To increase the diversity of generated text, stochastic sampling methods **introduce randomness in prediction**. In each round of prediction, a set of high-probability candidate words is first selected, and then a word is sampled randomly according to its probability distribution as the prediction result for this round. Currently, mainstream Top-K sampling and Top-P sampling methods select candidate words by specifying the number of candidate words and setting a probability threshold, respectively. Introducing the Temperature mechanism can adjust the probability distribution of candidate words. The following sections will introduce Top-K sampling, Top-P sampling, and the Temperature mechanism.

1. Top-K Sampling

Top-K sampling selects the K words with the highest probabilities $\{w^1_{i+1}, w^2_{i+1}, ..., w^K_{i+1}\}$ as the candidate set for each round of prediction, and then normalizes the probabilities of these words using the softmax function to obtain the following distribution function:

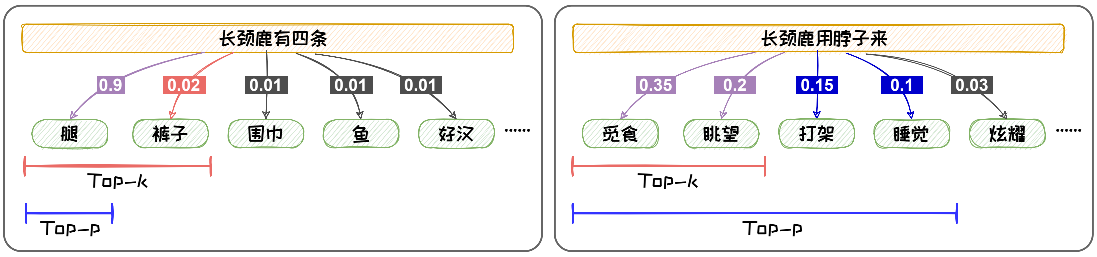
$$p(w^1_{i+1}, ..., w^K_{i+1}) = \left\{ \frac{exp(o_i[w^1_{i+1}])}{\sum_{j=1}^{K} exp(o_i[w^j_{i+1}])}, ..., \frac{exp(o_i[w^K_{i+1}])}{\sum_{j=1}^{K} exp(o_i[w^j_{i+1}])} \right\}. \tag{1.35}$$

Then, the prediction result for this round is sampled according to this distribution, i.e.,

$$w_{i+1} \sim p(w^1_{i+1}, ..., w^K_{i+1}). \tag{1.36}$$

Top-K sampling can effectively increase the novelty of the generated text. For example, in the example shown in Figure 1.8, if we use a Top-3 sampling strategy, we might choose " 看着躲在" (looking at hiding). " 长颈鹿看着躲在" (giraffe looking at hiding) could be the beginning of a highly dramatic suspense story.

However, setting the candidate set to a fixed size K will result in significant differences in the distribution across different rounds of prediction. When the variance of the distribution of candidate words is large, it may lead to the selection of **words with low probability and unreasonable** in this round of prediction, resulting in "gibberish." For example, in the example shown in Figure 1.9 (a), Top-2 sampling might sample " 长颈鹿有四条裤子" (giraffe has four pants). When the variance of the distribution of candidate



(a) Top-k胡言乱语 (b) Top-k枯燥无味

Figure 1.9: Potential issues with Top-K sampling and comparison with Top-P method.

words is small, or even tends to be uniform, the fixed-size candidate set **cannot accommodate more words with similar probabilities, leading to an insufficiently rich candidate set**, resulting in words lacking novelty and producing ”boring” text. For example, in the example shown in Figure 1.9 (b), using Top-2 sampling, we can only get ” 长颈鹿用脖子来觅食” (giraffe uses neck to forage) or ” 长颈鹿用脖子来眺望” (giraffe uses neck to look far), which are well-known uses of a giraffe’s neck and lack novelty. However, a giraffe’s neck can also be used for fighting or sleeping, and Top-2 sampling is likely to exclude these novel and less common uses. To solve the above problems, we can use Top-P sampling, also known as Nucleus sampling.

2. Top-P Sampling

To solve the problems caused by a fixed candidate set, Top-P sampling (also known as Nucleus sampling) was proposed [9]. It sets a threshold p to select the candidate set. The candidate set can be expressed as $S_p = \{w_{i+1}^1, w_{i+1}^2, ..., w_{i+1}^{|S_p|}\}$, where for S_p, $\sum_{w \in S_p} o_i[w] \geq p$. The distribution of elements in the candidate set follows:

$$p(w_{i+1}^1, ..., w_{i+1}^{|S_p|}) = \left\{ \frac{exp(o_i[w_{i+1}^1])}{\sum_{j=1}^{|S_p|} exp(o_i[w_{i+1}^j])}, ..., \frac{exp(o_i[w_{i+1}^{|S_p|}])}{\sum_{j=1}^{|S_p|} exp(o_i[w_{i+1}^j])} \right\}. \tag{1.37}$$

Then, the prediction result for this round is sampled according to this distribution, i.e.,

$$w_{i+1} \sim p(w_{i+1}^1, ..., w_{i+1}^{|S_p|}). \tag{1.38}$$

By applying the threshold as the standard for selecting the candidate set, Top-P sampling can **avoid selecting words with low probability and unreasonable**, thereby reducing ”gibberish.” For example, in the example shown in Figure 1.9 (a), if we use 0.9 as the threshold, we can effectively avoid the problem of ” 长颈鹿有四条裤子” (giraffe has four pants). Moreover, it can **accommodate more words with similar probabilities, increasing the richness of the text**, improving ”boring” text. For example, in the example shown in Figure 1.9 (b), if we use 0.9 as the threshold, we can include less well-known uses of a giraffe’s neck, such as fighting or sleeping.

3. Temperature Mechanism

The randomness of Top-K sampling and Top-P sampling is determined by the probability output of the language model and cannot be freely adjusted. However, in different scenarios, our requirements for randomness may be different. For example, in open-text generation, we prefer to generate more creative texts, so we need stronger randomness. In code generation, we hope the generated code is more conservative, so we need weaker randomness. Introducing the Temperature mechanism can **adjust the randomness of decoding**. The Temperature mechanism controls the distribution by scaling the independent variable in the softmax function and then using the nonlinearity of the softmax function. Let the variable for Temperature scaling be T.

After introducing the Temperature mechanism, the distribution of the candidate set for Top-K sampling is as follows:

$$p(w_{i+1}^1, ..., w_{i+1}^K) = \left\{ \frac{exp(\frac{o_i[w_{i+1}^1]}{T})}{\sum_{j=1}^K exp(\frac{o_i[w_{i+1}^j]}{T})}, ..., \frac{exp(\frac{o_i[w_{i+1}^K]}{T})}{\sum_{j=1}^K exp(\frac{o_i[w_{i+1}^j]}{T})} \right\}. \tag{1.39}$$

After introducing the Temperature mechanism, the distribution of the candidate set for Top-P sampling is as follows:

$$p(w_{i+1}^1, ..., w_{i+1}^{|S_p|}) = \left\{ \frac{exp(\frac{o_i[w_{i+1}^1]}{T})}{\sum_{j=1}^{|S_p|} exp(\frac{o_i[w_{i+1}^j]}{T})}, ..., \frac{exp(\frac{o_i[w_{i+1}^{|S_p|}]}{T})}{\sum_{j=1}^{|S_p|} exp(\frac{o_i[w_{i+1}^j]}{T})} \right\}. \tag{1.40}$$

It is easy to see that when $T > 1$, the Temperature mechanism reduces the probability gap between words in the candidate set, making the distribution flatter and increasing randomness. When $0 < T < 1$, the Temperature mechanism increases the probability gap between elements in the candidate set, making the stronger stronger and the weaker weaker, and words with high probability are more likely to be selected, thus reducing randomness. The Temperature mechanism can effectively adjust randomness to meet different needs.

1.5 Evaluation of Language Models

After obtaining a language model, we need to evaluate its generation capabilities to determine its quality. The methods for evaluating the generation capabilities of language models can be divided into two categories. The first category of methods does not rely on specific tasks and directly evaluates the model's generation capabilities through the model's output, known as intrinsic evaluation. The second category of methods evaluates the language model's ability to handle specific generation tasks, such as machine translation and summarization, through certain specific tasks, known as extrinsic evaluation.

1.5.1 Intrinsic Evaluation

In intrinsic evaluation, the test text typically consists of text that is independently and identically distributed with the text used in pretraining, **without relying on specific tasks**. The most commonly used intrinsic evaluation metric is perplexity [10], which measures the degree of "confusion" the language model has with the test text. Let the test text be

1897 ZHEJIANG UNIVERSITY

$s_{test} = w_{1:N}$. The perplexity PPL of the language model on the test text s_{test} can be calculated as follows:

$$PPL(s_{test}) = P(w_{1:N})^{-\frac{1}{N}} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{<i})}}. \tag{1.41}$$

From the above equation, it can be seen that if the language model is more ”certain” about the test text (i.e., the probability of generating the test text is higher), the perplexity value is smaller. Conversely, if the language model is more ”uncertain” about the test text (i.e., the probability of generating the test text is lower), the perplexity value is larger. Since the test text and the pretraining text are identically distributed, the pretraining text represents the text we want the language model to learn to generate. If the language model is less ”confused” about these test texts, it indicates that the language model aligns more with our original intention of training it. Therefore, perplexity can to some extent measure the generation capabilities of the language model.

Rewriting the perplexity, it can be rewritten into the following equivalent form:

$$PPL(s_{test}) = exp(-\frac{1}{N}\sum_{i=1}^{N} log\ P(w_i|w_{<i})). \tag{1.42}$$

Here, $-\frac{1}{N}\sum_{i=1}^{N} log\ P(w_i|w_{<i})$ can be considered as the cross-entropy between the word distribution generated by the generative model and the true word distribution of the test sample, i.e., $-\frac{1}{N}\sum_{i=1}^{N}\sum_{d=1}^{|D|} I(\hat{w}_d = w_i)\log o_{i-1}[w_i]$, where D is the dictionary used by the language model. Because $P(w_i|w_{<i}) \leq 1$, this cross-entropy is an upper bound on the information entropy of the word distribution generated by the generative model, i.e.,

$$-\frac{1}{N}\sum_{i=1}^{N} P(w_i|w_{<i})\ log\ P(w_i|w_{<i}) \leq -\frac{1}{N}\sum_{i=1}^{N} log\ P(w_i|w_{<i}). \tag{1.43}$$

Therefore, a decrease in perplexity also means a decrease in entropy, which means a reduction in the likelihood of the model generating ”gibberish.”

1.5.2 Extrinsic Evaluation

In extrinsic evaluation, the test text typically includes questions and corresponding standard answers for the task, which depends on the specific task. Through extrinsic evaluation, we can judge the language model's ability to handle specific tasks. Extrinsic evaluation methods are usually divided into two categories: evaluation methods based on statistical metrics and evaluation methods based on language models. The following introduces classic methods in these two categories.

1. Evaluation Based on Statistical Metrics

Evaluation methods based on statistical metrics construct statistical indicators to evaluate the degree of fit between the language model's output and the standard answer, using this as the basis for evaluating the language model's generation capabilities. BLEU (BiLingual Evaluation Understudy) and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) are the two most widely used statistical metrics. Among them, BLEU is a precision-oriented metric, while ROUGE is a recall-oriented metric. The following introduces these two metrics separately.

BLEU was proposed to evaluate the performance of models in machine translation (MT) tasks [6]. It calculates the overlap between the generated translation and the reference translation at the word level. Specifically, BLEU calculates the geometric mean of multi-level n-gram precision. Let the set of generated translation texts be $S_{gen} = \{S^i_{gen}\}_{i=1}^{|S_{gen}|}$, and the corresponding set of reference translations be $S_{ref} = \{S^i_{ref}\}_{i=1}^{|S_{ref}|}$, where S^i_{gen} and S^i_{ref} correspond one-to-one, and $|S_{ref}| = |S_{gen}|$. The original definition of n-gram precision is as follows:

$$Pr(g_n) = \frac{\sum_{i=1}^{|S_{gen}|} \sum_{g_n \in S^i_{gen}} Count_{match}(g_n, S^i_{ref})}{\sum_{i=1}^{|S_{gen}|} \sum_{g_n \in S^i_{gen}} Count(g_n)}. \tag{1.44}$$

Here, g_n represents an n-gram. The numerator of the above formula calculates the number of overlapping n-grams between the generated translation and the reference translation, and the denominator calculates the total number of n-grams in the generated translation. For example, if an MT model translates " 大语言模型" (large language models) into English, the generated translation is "big language models," and the reference text is "large language models." When $n = 1$, $Pr(g_1) = \frac{2}{3}$. When $n = 2$, $Pr(g_2) = \frac{1}{2}$.

Based on n-gram precision, BLEU takes the geometric mean of N n-gram precisions as the evaluation result:

$$BLEU = \sqrt[N]{\prod_{n=1}^{N} Pr(g_n)} = exp\left(\sum_{n=1}^{N} logPr(g_n)\right). \tag{1.45}$$

For example, when $N = 3$, BLEU is the geometric mean of unigram precision, bigram precision, and trigram precision. On the basis of the original BLEU, we can also adjust BLEU by weighting different n-gram precisions or setting penalty terms for different text lengths, thereby obtaining results that are closer to human evaluation.

ROUGE was proposed to evaluate the performance of models in summarization tasks [13]. Common ROUGE evaluations include ROUGE-N, ROUGE-L, ROUGE-W, and ROUGE-S. Among them, ROUGE-N is a recall-based metric based on n-grams, ROUGE-L is a recall-based metric based on the longest common subsequence (LCS), and ROUGE-W is a recall-based metric that introduces weighting operations on LCS based on ROUGE-L. ROUGE-S is a recall-based metric based on Skip-bigram. The definitions of ROUGE-N and ROUGE-L are given below. The specific calculation methods for ROUGE-W and ROUGE-S can be found in [13].

The definition of ROUGE-N is as follows:

$$ROUGE\text{-}N = \frac{\sum_{s \in S_{ref}} \sum_{g_n \in s} Count_{match}(g_n, s_{gen})}{\sum_{s \in S_{ref}} \sum_{g_n \in s} Count(g_n)}. \tag{1.46}$$

The definition of ROUGE-L is as follows:

$$ROUGE\text{-}L = \frac{(1+\beta^2)R_r^g R_g^r}{R_r^g + \beta^2 R_g^r}. \tag{1.47}$$

Where,

$$R_r^g = \frac{LCS(s_{ref}, s_{gen})}{|s_{ref}|}, \tag{1.48}$$

$$R_g^r = \frac{LCS(s_{ref}, s_{gen})}{|s_{gen}|}, \tag{1.49}$$

$LCS(s_{ref}, s_{gen})$ is the length of the longest common subsequence between the model-generated summary s_{gen} and the reference summary s_{ref}, and $\beta = R_g^r / R_r^g$.

Evaluation methods based on statistical metrics score the degree of overlap between the language model's generated answers and the standard answers. Such scoring cannot fully adapt to the diversity of expressions in generation tasks, which is far from human evaluation, especially when the generated samples have strong creativity and diversity. To solve this problem, we can introduce another language model as a "judge" in the evaluation, using the capabilities it has mastered during pretraining to evaluate the generated text. The following introduces this evaluation method that introduces a "judge" language model.

2. Evaluation Based on Language Models

Currently, evaluation methods based on language models are mainly divided into two categories: (1) evaluation methods based on contextual embeddings; (2) evaluation methods based on generative models. A typical evaluation method based on contextual embeddings is BERTScore [24]. A typical evaluation method based on generative models is G-EVAL [14]. Compared to BERTScore, G-EVAL does not require human-annotated reference answers, making it better suited for tasks lacking human annotations.

BERTScore calculates the similarity between the generated text s_{gen} and the reference text s_{ref} based on BERT's contextual embedding vectors to evaluate the generated samples. BERT will be introduced in detail in Chapter 2. Let the generated text contain $|s_{gen}|$ words, i.e., $s_{gen} = \{w_g^i\}_{i=1}^{|s_{gen}|}$. Let the reference text contain $|s_{ref}|$ words, i.e., $s_{ref} = \{w_r^i\}_{i=1}^{|s_{ref}|}$. Use BERT to obtain the contextual embedding vectors of each word in s_{gen} and s_{ref}, i.e., $v_g^i = BERT(w_g^i|s_{gen})$ and $v_r^i = BERT(w_r^i|s_{ref})$. Using the embedding vector sets $v_{gen} = \{v_g^i\}_{i=1}^{|v_{gen}|}$ and $v_{ref} = \{v_r^i\}_{i=1}^{|v_{ref}|}$ of the generated text and the reference text, we can calculate BERTScore. BERTScore evaluates the generated document from three aspects: precision, recall, and F1 measure. Their definitions are as follows:

$$P_{BERT} = \frac{1}{|v_{gen}|} \sum_{i=1}^{|v_{gen}|} max_{v_r \in v_{ref}} {v_g^i}^\top v_r^i, \tag{1.50}$$

$$R_{BERT} = \frac{1}{|v_{ref}|} \sum_{i=1}^{|v_{ref}|} max_{v_g \in v_{gen}} {v_r^i}^\top v_g^i, \tag{1.51}$$

$$F_{BERT} = \frac{2P_{BERT} \cdot R_{BERT}}{P_{BERT} + R_{BERT}}. \tag{1.52}$$

Compared to statistical evaluation metrics, BERTScore is closer to human evaluation results. However, BERTScore relies on human-provided reference texts, making it inapplicable to scenarios lacking human-annotated samples. Thanks to the development of generative large language models, G-EVAL uses GPT-4 to score generated texts without reference texts. G-EVAL guides GPT-4 to output evaluation scores through prompt engineering. Prompt Engineering will be explained in detail in Chapter 3 of this book.

As shown in the figure below, the prompt for G-EVAL is divided into three parts: (1) task description and scoring criteria; (2) evaluation steps; (3) input text and generated text. In the first part, the task description specifies what the evaluation task is (such as summarization), and the scoring criteria provide the range of scores, factors to consider

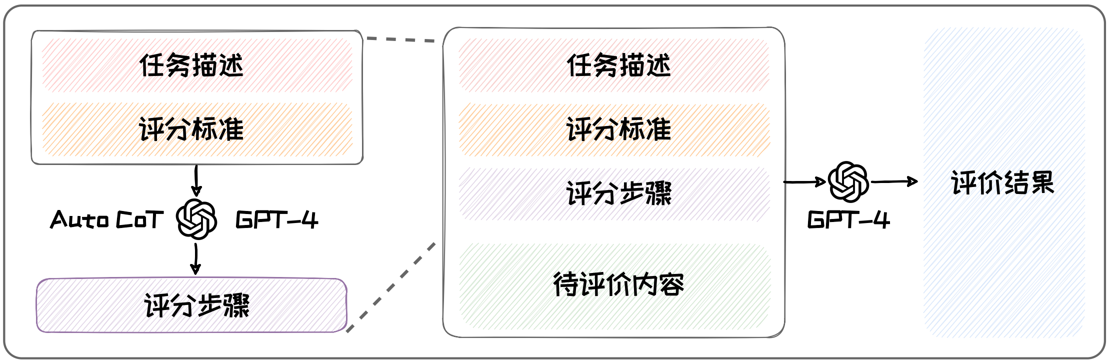


Figure 1.10: G-EVAL evaluation process.

when scoring, and other content. The evaluation steps in the second part are a chain of thoughts (CoT) generated by GPT-4 based on the content of the first part. The chain of thoughts will be explained in detail in Chapter 3 of this book. The input text and generated text in the third part are the source text and the text generated by the model to be evaluated. For example, in a summarization task, the input text is the original text, and the generated text is the generated summary. Combining these three parts into one prompt and inputting it to GPT-4, GPT-4 can provide the corresponding score. Directly using the scores given by GPT-4 as the evaluation scores may result in insufficient differentiation, so G-EVAL also introduces a mechanism to improve this by weighted averaging of all possible scores [14].

In addition to G-EVAL, several other evaluation methods based on generative models have been proposed recently [12]. A typical one is InstructScore [23], which can not only provide numerical scores but also give explanations for these scores. Compared to methods based on statistical metrics and contextual embeddings, evaluation methods based on generative models have unique advantages in terms of accuracy, flexibility, interpretability, and other aspects. It can be foreseen that evaluation methods based on generative models will receive more widespread attention and application in the future.

References

[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.

2 Architectures for LLMs

With the explosive growth of data resources and computing power, the scale of parameters and performance of language models have achieved a qualitative leap, ushering in a new era of Large Language Models (LLM). Leveraging **vast parameter scales** and **abundant training data**, large language models not only demonstrate powerful generalization capabilities but also give rise to the emergence of new intelligence, standing at the forefront of generative artificial intelligence (Artificial Intelligence Generated Content, AIGC) waves. Currently, the technology of large language models is thriving, with various models emerging one after another. These models have already shown capabilities on par with or even surpassing those of humans in a wide range of application scenarios, leading a new wave of industrial revolution driven by AIGC. This chapter will delve into the relevant background knowledge of large language models and introduce three mainstream model architectures: Encoder-only, Encoder-Decoder, and Decoder-only. By listing representative models for each architecture, it will conduct an in-depth analysis of their main innovations in network structure, training methods, and other aspects. Finally, this chapter will briefly introduce some non-Transformer architecture models to showcase the current diverse development landscape of large language model research.

*This book is continuously updated, and the GitHub link is：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

浙江大学 1897 ZHEJIANG UNIVERSITY

2.1 Big Data + Large Models → New Intelligence

In the cutting-edge field of natural language processing, large language models are driving a wave of technological innovation with their massive model sizes, enormous data throughput capabilities, and superior model performance. When we talk about the ”largeness” of large language models, we refer not only to **the vast scale of the models** but also to **the massive scale of the training data** and **the powerful capabilities** derived from these factors. These models, like giant ships exploring uncharted territories, not only continuously push the limits of performance on existing technologies but also exhibit astonishing potential in exploring new capabilities.

As of June 2024, more than a hundred large language models have been born both domestically and internationally, generating profound impacts in both academic and industrial sectors.Figure 2.1 illustrates some of the models that have had significant influence.

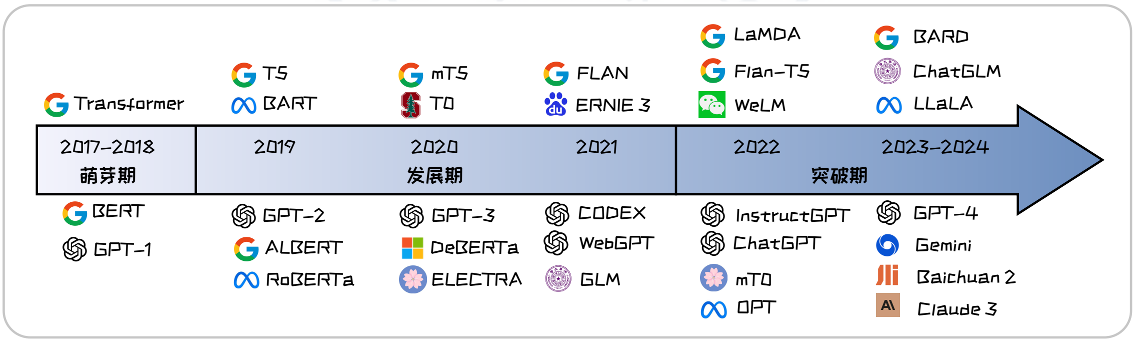


Figure 2.1: Three Stages of Capability Emergence in Large Language Models.

The development of large language models can be roughly divided into three phases. From 2017 to 2018 was **the germination period** of foundational models, marked by the birth of the Transformer architecture and the introduction of models like BERT[11] and GPT-1[27], which ushered in a new era of pre-trained language models. From 2019 to

2022 was **the development period** of large language models, characterized by significant improvements in parameter scale and capabilities through models such as GPT-2 [1], T5[29], and GPT-3[5]. During this time, researchers began to delve deeply into the potential of large language models. Starting from 2022, **the breakthrough period** began, with the release of models like ChatGPT [2] and GPT-4 [3], marking substantial advancements in the technology related to large language models. Simultaneously, numerous companies and research institutions have launched their own models, such as the Baichuan large model by BaiChuan AI[44] and Baidu's Wenxin Yiyu, further accelerating the rapid development of large language models.

This section will delve into the developmental journey of large language models, particularly focusing on advancements in capability enhancement and the emergence of new abilities. We will start by examining the growth in model size and data scale, discussing how these factors have collectively contributed to the leap in model performance and the appearance of new functionalities.

2.1.1 Big Data + Large Models → Capability Enhancement

Driven by the wave of digitalization, data flows like a converging flood, while models operate like mighty ships riding the waves. The growth in data volume provides models with richer information sources, meaning that models can learn a wider variety of language patterns and deeper semantic relationships. The continuous expansion of model size significantly enhances the model's expressive power, enabling it to capture more subtle linguistic features and complex language structures. Under the combined effect of such

---

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

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

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

a vast scale of model parameters and diversified training data, the model's internal ability to fit data distributions continuously improves, thereby demonstrating higher adaptability and effectiveness in complex and dynamic data environments[7].

However, the growth in model size and data volume is not without its costs, bringing with it higher computational expenses and storage requirements. This necessitates finding an appropriate balance between resource consumption and performance enhancement when designing models. To address this challenge, scaling laws for large language models have emerged. These laws elucidate the relationship between model capabilities and the scales of models and data, offering valuable guidance and references for the design and optimization of large language models. This chapter will delve into two scaling laws: the Kaplan-McCandlish scaling law proposed by OpenAI and the Chinchilla scaling law introduced by DeepMind.

1. Kaplan-McCandlish Scaling Law

In 2020, Jared Kaplan, Sam McCandlish, and others from the OpenAI team [16] first investigated the functional relationship between the performance of neural networks and the scale of data$D$and model size$N$. They conducted experiments across datasets of varying sizes (ranging from 22 million to 23 billion tokens) and models of different scales (from 768 to 1.5 billion parameters). Based on the experimental results, they derived two fundamental formulas:

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

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

Here, $L(N)$ represents the cross-entropy loss function under **different model sizes** when the data scale is fixed, reflecting the impact of model size on the ability to fit the data. Correspondingly, $L(D)$ represents the cross-entropy loss function under **different data scales** when the model size is fixed, revealing the effect of data volume on model learning. The value of L measures the accuracy of the model in fitting the data distribution; a smaller value indicates a more precise fit of the model to the data distribution, and thus **a stronger self-learning capability** of the model.

The experimental results and relevant formulas indicate that the performance of a model is highly positively correlated with both model size and data scale. However, when the model size is the same, **the specific architecture of the model** has a relatively smaller impact on its performance. Therefore, expanding the model size and enriching the dataset have become two key strategies for improving the performance of large models.

Furthermore, in further studies on the optimal allocation of computational budget, OpenAI found that the total computation C is approximately proportional to the product of data volume D and model size N, i.e.$C \approx 6ND$. Under this condition, if the computational budget increases, to achieve optimal model performance, both the scale of the dataset D and the model size N should be increased simultaneously. However, **the growth rate of model size should be slightly faster than that of data size.** Specifically, the optimal configuration ratio should be $N_{opt} \propto C^{0.73} and D_{opt} \propto C^{0.27}$. This means that if the total computational budget increases by 10 times, the model size should be expanded by approximately 5.37 times, and the data size should be increased by about 1.86 times, to achieve the best model performance.

The scaling law proposed by OpenAI not only quantitatively reveals the significant impact of data scale and model size on model capabilities but also points out that investment in model size should be slightly higher than that in data scale. This finding not only provides new insights into understanding the intrinsic working mechanisms of language models but also offers valuable guidance on how to efficiently train these models.

2. Chinchilla Scaling Law

The team at DeepMind, a subsidiary of Google, presented a differing perspective on the notion that ”the growth rate of model size should be slightly higher than that of data size.” In 2022, they conducted in-depth experimental studies across a broader range of model sizes (from 70 million to 160 billion parameters) and data scales (from 5 billion to 500 billion tokens). Based on these studies, they proposed the Chinchilla scaling law.[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 also explored the issue of optimal allocation of computational budget, ultimately concluding that the optimal configuration of dataset size D and model size N is $N_{opt} \propto C^{0.46}$ and $D_{opt} \propto C^{0.54}$. This result indicates that the **dataset size D and model size N are almost equally important**, if the total computational budget increases by 10 times, then both the model size and dataset size should be increased by approximately 3.16 times. Google’s subsequent technical report on PaLM 2, released in May 2023 [2], also reaffirmed this view, further emphasizing the importance of data scale in enhancing model performance.

Moreover, the Chinchilla scaling law further proposes that the ideal dataset size should be 20 times the model size. For example, for a model with 7B (70 billion parameters), the ideal training dataset size should be 140B (140 billion tokens). However, many previous models were pretrained with insufficient data volumes. For instance, the largest version of OpenAI’s GPT-3 [5] model, which has 175 billion parameters, was trained using only 300 billion tokens; similarly, Microsoft’s MT-NLG [35] model, with 530 billion parameters, was trained using only 270 billion tokens. Therefore, DeepMind introduced the Chinchilla model, whose dataset size is 20 times the model size (70 billion parameters, 1.4 trillion tokens), ultimately achieving significant performance breakthroughs.

The Chinchilla scaling law proposed by DeepMind complements and optimizes the earlier research by OpenAI, emphasizing the importance of data scale in enhancing model performance. It points out that the **model size and data size should increase at the same ratio**, opening up a new direction in the development of large language models: rather than simply pursuing an increase in model size, the focus should be on **optimizing the ratio between model size and data size**.

2.1.2 Big Data + Large Models → Capability Expansion

As shown in Figure 2.2, the continuous increase in the scale of training data and the number of parameters not only brings steady enhancements in learning capabilities but also ”unlocks” a series of **new abilities** [4], such as in-context learning, commonsense reasoning, mathematical operations, code generation, etc. Notably, these new abilities are not acquired through training on specific downstream tasks but emerge naturally as the complexity of the model increases [5]. These abilities are therefore referred to as emergent abilities of large language models (Emergent Abilities).

Emergent abilities often exhibit suddenness and unpredictability. Similar to ”phase transitions” in nonlinear systems, where the system undergoes significant changes at certain threshold points, these abilities do not follow a smooth, gradual accumulation process. Instead, they **suddenly appear** once the model reaches a certain scale and complexity [32]. For example, in the evolution of the GPT series, some typical emergent abilities can be

---

[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

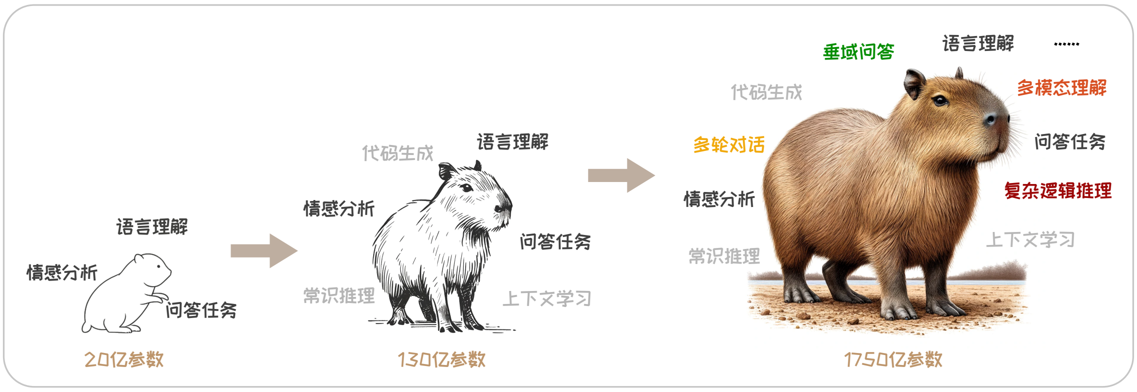


Figure 2.2: Emergent abilities of large language models as they grow in scale, generated by GPT-4o.

observed.

- **In-Context Learning**: In-Context Learning refers to the ability of large language models to utilize contextual information from input text to perform specific tasks during inference. Models with in-context learning capabilities can understand task requirements and generate appropriate outputs **without additional training**, merely **through examples or prompts**. In the GPT series, different versions of the model show significant differences in in-context learning ability. Early versions like GPT-1 and GPT-2 had very limited in-context learning capabilities and could not accurately reason or respond based on contextual information. The 13 billion parameter version of GPT-3 made significant progress in in-context learning, capable of completing common tasks based on provided contextual prompts. However, its performance on more complex or domain-specific tasks remained limited. The largest version of GPT-3 with 175 billion parameters and subsequent GPT-4 models demonstrate powerful in-context understanding and learning capabilities, able to handle a wide range of highly complex tasks based on a few examples.
- **Commonsense Reasoning**: Commonsense reasoning capabilities endow large language models with the ability to **understand and infer based on common knowl-**

edge and logic. This includes understanding generally accepted facts, events, and behavioral patterns in everyday life and using this knowledge to answer questions, solve problems, and generate relevant content. GPT-1 and GPT-2 had very limited commonsense reasoning capabilities, often making incorrect inferences or lacking detailed explanations. Larger versions of GPT-3, however, could generate reasonable and coherent commonsense responses in most cases. As for the largest version of GPT-3 with 175 billion parameters and subsequent models like GPT-4, they can handle highly complex commonsense reasoning tasks with logical consistency and rich detail.

- **Code Generation**: Code generation capabilities allow large language models to **automatically generate programming code based on natural language descriptions**. This includes understanding the syntax and semantics of programming languages, parsing user requirements, generating corresponding code, and, in some cases, optimizing and debugging code. GPT-1 and GPT-2 could only generate very simple code snippets and failed to effectively understand specific programming needs. When the 13 billion parameter version of GPT-3 emerged, it could handle common programming tasks and generate structured code snippets well, though it still had limitations on extremely complex or domain-specific tasks. With 175 billion parameters, the model could handle complex programming tasks, generate multi-language code, optimize code, and fix errors, showcasing high-quality code generation and understanding capabilities.
- **Logical Reasoning**: Logical reasoning capabilities enable large language models to make **logical inferences and conclusions based on given information and rules**. This includes simple conditional reasoning, multi-step logical reasoning, and main-

taining logical consistency in complex scenarios. Early generative pre-trained models like GPT-1 and GPT-2 had very limited logical reasoning capabilities, and even the 13 billion parameter version of GPT-3, while capable of handling some logical reasoning tasks, still had limitations in complexity and precision. It wasn't until the 175 billion parameter version that GPT-3 could handle complex logical reasoning tasks, generating detailed and coherent reasoning processes.

- …

These emergent abilities allow large language models to perform various tasks without specialized training but also bring numerous challenges, including model interpretability, information security and privacy, ethical and fairness issues, and the massive demand for computational resources. Addressing these challenges requires comprehensive considerations at technical, legal, and social levels to ensure the healthy development and sustainable progress of large language models.

2.2 Overview of Large Language Model Architectures

In the evolution of language models, the introduction of the Transformer framework [42] marked a turning point. Its unique self-attention mechanism significantly enhanced the model's **capability to process sequential data**, particularly excelling in **capturing long-range dependencies**. Additionally, the Transformer framework's **support for parallel computing** greatly accelerated the training process. Currently, most large language models are based on the Transformer framework and have evolved into three classic architectures: Encoder-only, Decoder-only, and Encoder-Decoder. These three architectures differ in design and functionality. This section will briefly introduce the design philosophy and pre-training methods of these three architectures, analyze their differences, and discuss their respective evolutionary trends.

2.2.1 Categories of Mainstream Model Architectures

This subsection will provide a brief introduction to the Encoder-only, Decoder-only, and Encoder-Decoder architectures from perspectives such as design philosophy and training methods.

1. Encoder-only Architecture

The Encoder-only architecture uses only the encoder part of the Transformer to receive input text and generate context-relevant features. Specifically, the Encoder-only architecture consists of three parts: the **input encoding** part, the **feature encoding** part, and the **task processing** part, as shown in Figure 2.3. The input encoding part includes three processes: **tokenization**, **vectorization**, and **adding positional encoding**. The feature encoding part is composed of multiple identical **encoder blocks**, each containing a **self-attention module** and a **fully connected feed-forward module**. The task processing module is designed specifically for task requirements and can be customized by users. During the pre-training stage and inference stage, the input encoding and feature encoding parts of the Encoder-only architecture remain consistent, while the task processing part needs to be customized according to the characteristics of different tasks.

In the **input encoding part**, the raw input text is tokenized by a tokenizer into a token sequence, which is then mapped to a vector sequence through a vocabulary and embedding matrix to ensure that the text information is digitized. This process and its details will be discussed in Section 3.1. Positional encoding is then added to each vector sequence to retain the order information of words in the text. In the **feature encoding part**, the ob-

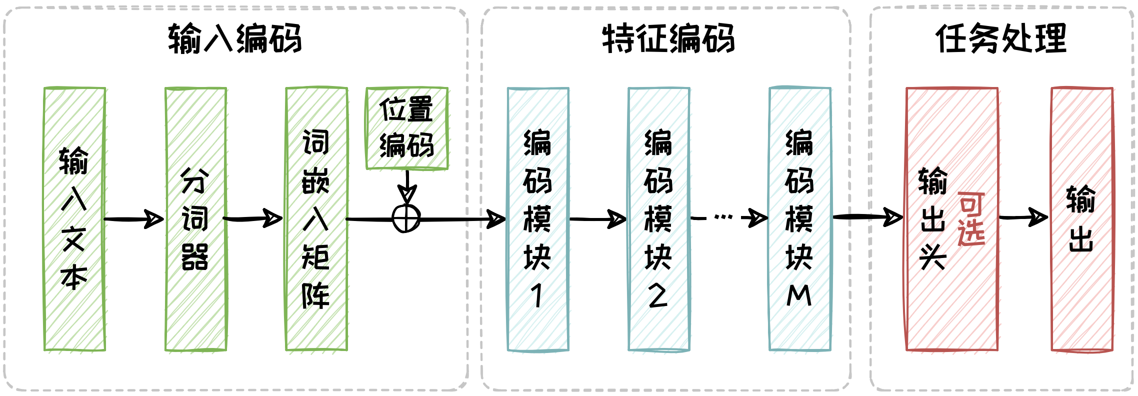


Figure 2.3: Encoder-only architecture.

tained vector sequences pass through a series of encoder blocks, which further extract and deepen text features through self-attention mechanisms and feed-forward networks. The **task processing part** differs between the pre-training stage and the fine-tuning stage for downstream tasks. During the pre-training stage, the model typically uses a fully connected layer as the output head to complete tasks such as masked prediction. During the fine-tuning stage for downstream tasks, the output head is customized according to specific task requirements. For example, for discriminative tasks such as sentiment analysis or topic classification, a classifier can be added to directly output the classification result. For generative tasks such as text summarization, a fully connected layer is added to predict subsequent tokens one by one. However, this approach has many limitations, such as requiring the recalculation of the entire input sequence representation each time a new token is generated, which increases computational costs and may result in less coherent text. This section will provide a more detailed introduction to the Encoder-only architecture in Section 2.3.

2. Encoder-Decoder Architecture

To address the shortcomings of the Encoder-only architecture in text generation tasks, the Encoder-Decoder architecture introduces a decoder and uses cross-attention mechanisms to enable effective interaction between the encoder and decoder.

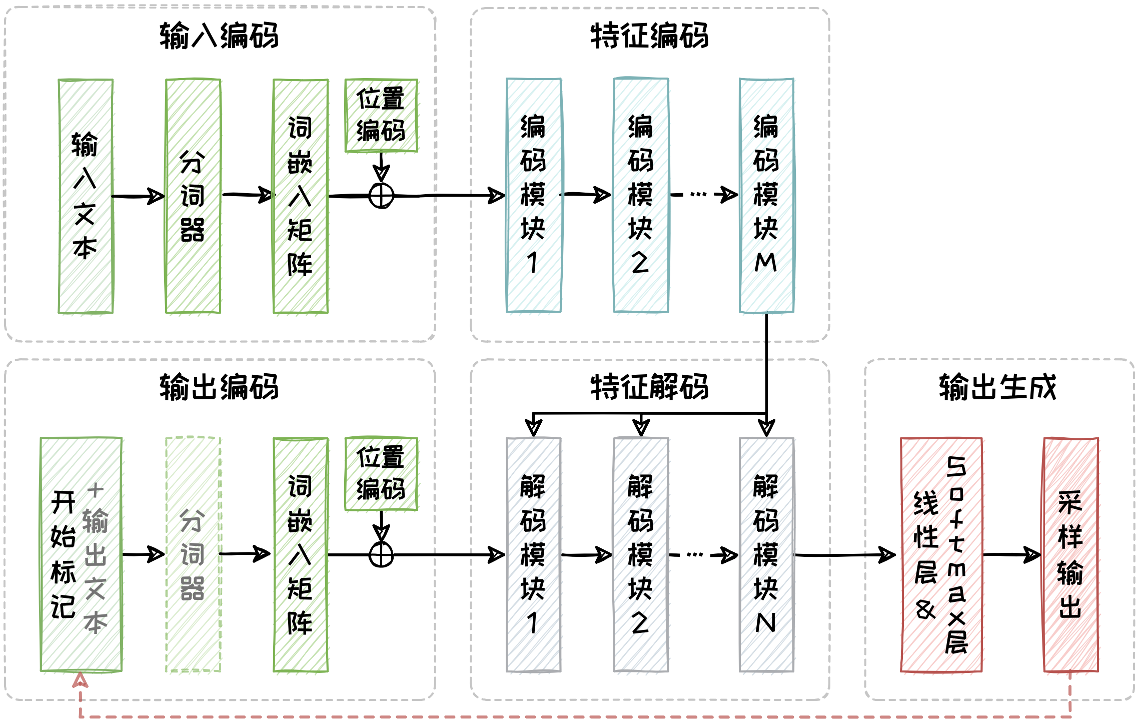


Figure 2.4: Encoder-Decoder architecture. The tokenizer and output text exist only during the training phase, while the red dashed line representing ”autoregressive” exists only during the inference phase.

Specifically, the decoder includes three parts: **output encoding**, **feature decoding**, and **output generation**. The **output encoding** part has the same structure as the input encoding part in the encoder, including tokenization, vectorization, and adding positional encoding, converting the raw input text into a vector sequence with positional information. The **feature decoding** part is also highly similar to the feature encoding part in terms of network structure, including masked self-attention modules, cross-attention modules, and fully connected feed-forward modules. The masked self-attention module ensures that the model only focuses on the preceding context, preventing it from ”peeking” into future information, thus enabling ”autoregressive” training and inference without ”future leakage”. The cross-attention module handles the transmission of relevant information

ZHEJIANG UNIVERSITY 1897

from the encoder to the decoder. The **output generation** part consists of a linear layer and a Softmax layer, responsible for converting the decoded feature vectors into probability distributions over the vocabulary and sampling the most suitable token as the output.

Figure 2.4 illustrates the specific workflow of the Encoder-Decoder architecture. During the training phase, samples contain both input and ground truth output texts. The input text is first converted into a vector sequence by the input encoding part and then processed through multiple stacked encoder blocks in the feature encoding part to obtain context representations. The output text is prefixed with a special start token `[START]`, and then tokenized, embedded, and position-encoded in the output encoding part before being fed into the feature decoding part in parallel. The decoder uses teacher forcing, where in each prediction step, the known part of the ground truth output is used as input, combined with the context information from the last encoder block, to predict the next token. The loss between the predicted token and the ground truth token is calculated, and the model parameters are updated through backpropagation.

During the inference phase, since there is no ground truth output text, the output sequence initially contains only the start token `[START]`, and the tokenizer is no longer needed. The model generates tokens sequentially through autoregression, appending each sampled token to the output sequence for the next prediction. This process continues until a specific end token `[END]` is generated or the maximum output length set by the model is reached. Since each input depends on the results of the previous sampling, the output can only be **generated sequentially**. A more detailed introduction to the Encoder-Decoder architecture will be provided in Section 2.4.

3. Decoder-only Architecture

To effectively reduce the model size and lower overall computational complexity, the Decoder-only architecture eliminates the encoder part and the cross-attention modules from the Encoder-Decoder architecture. In this architecture, the model builds a language model using only the decoder. This architecture leverages an ”autoregressive” mechanism to generate fluent and coherent text given the context.

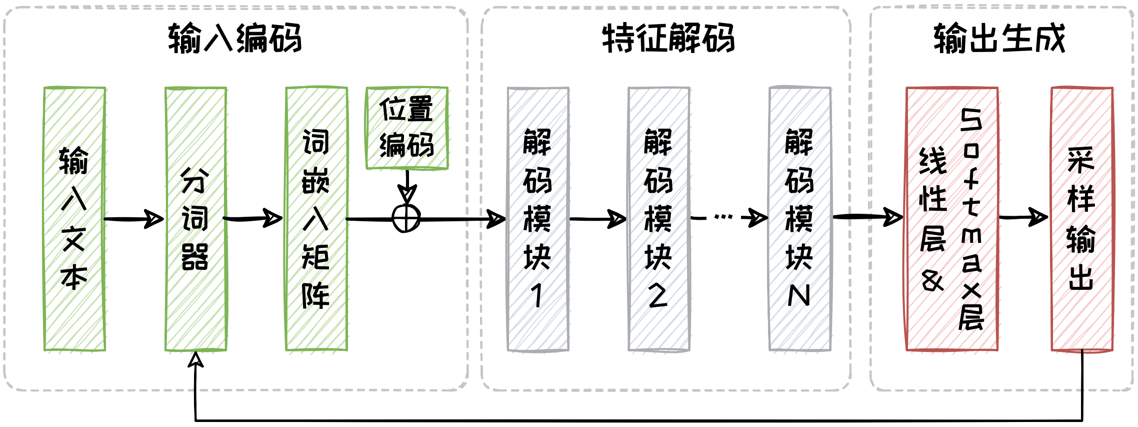


Figure 2.5: Decoder-only architecture.

The Decoder-only architecture also consists of three parts: the **input encoding** part, the **feature decoding** part, and the **output generation** part, as shown in Figure 2.5. The core feature of the Decoder-only architecture is the omission of the cross-attention sub-module in each encoder block, which is the main difference from the decoder part of the traditional Encoder-Decoder architecture. A more detailed introduction to the Decoder-only architecture will be provided in Section 2.5.

2.2.2 Functional Comparison of Model Architectures

Although the Encoder-only, Encoder-Decoder, and Decoder-only architectures are all derived from the Transformer framework, they have significant differences in attention matrices, which influence their functionalities and final application tasks. The following sections will analyze the main differences in attention matrices and applicable tasks.

1. Attention Matrices

The attention matrix (Attention Matrix) is a core component of the Transformer, used to calculate the dependencies between tokens in the input sequence. Through the attention mechanism, the model can flexibly focus on information carried by other tokens in the sequence when processing the current token, determining which tokens can influence each other during this process. As shown in Figure 2.6, the three architectures have significant differences in attention matrices.

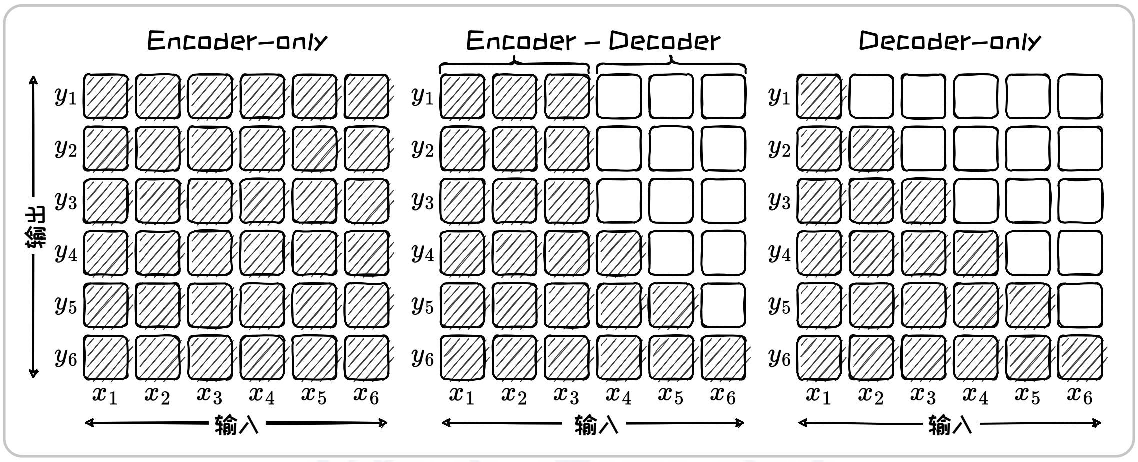


Figure 2.6: Attention matrices of the three architectures.

ZHEJIANG UNIVERSITY

The **Encoder-only architecture** uses the attention matrix from the self-attention module to capture relationships between tokens in the input sequence. The attention matrix in the Encoder-only architecture exhibits ”full” attention, meaning that the understanding of each token depends on all tokens in the entire input sequence. For example, when converting input token x_i to context vector y_i, the model can utilize all input information from x_1 to x_n, known as the **bidirectional attention mechanism**. Under this bidirectional attention mechanism, the model can simultaneously leverage both past and future context information to deeply understand complex semantic connections and contextual dependencies.

The **Encoder-Decoder architecture** has a more complex attention matrix, combining self-attention in the encoder, masked self-attention in the decoder, and cross-attention mechanisms. The self-attention matrix in the encoder is similar to that in the Encoder-only architecture, used to generate a comprehensive context representation of the input sequence, exhibiting ”full” attention. The masked self-attention matrix in the decoder exhibits a ”lower triangular” attention pattern, ensuring that the model only focuses on previously generated tokens when generating the current token. Additionally, the cross-attention mechanism allows the decoder to dynamically reference the full context representation generated by the encoder, ensuring that the output is highly relevant and coherent with the input sequence. For example, when the encoder converts input x_i

2.3 Large Language Models Based on the Encoder-only Architecture

The Encoder-only architecture stands out in natural language processing tasks due to its unique bidirectional encoder model, especially in tasks that require a deep understanding of input text. This chapter will introduce the bidirectional encoder model and several typical Encoder-only architecture models.

2.3.1 Encoder-only Architecture

At the core of the Encoder-only architecture lies the **bidirectional encoder model (Bidirectional Encoder Model)**. When processing input sequences, the bidirectional encoder model integrates forward attention from left to right and backward attention from right to left, fully capturing the contextual information of each token, thus it is also referred to as having a **comprehensive attention mechanism**. Thanks to its context-aware capabilities and dynamic representation advantages, bidirectional encoders have significantly improved the performance of natural language processing tasks.

Unlike previous common methods such as Word2Vec and GloVe, which provide a **static encoding approach** with a **fixed vector representation** for each word, bidirectional encoders generate **dynamic context embeddings** for each word. These embeddings depend on the specific context of the input sequence, enabling the model to more accurately understand the dependencies and semantic information between words, effectively addressing the issue of polysemy. This dynamic representation allows bidirectional encoders to **excel at sentence-level tasks**, significantly surpassing the performance of static word embedding methods [20].

The Encoder-only architecture is based on the encoder part of the Transformer architecture. Although Encoder-only models do not directly generate text, the context embeddings they produce are crucial for deeply understanding the structure and meaning of input text. These models demonstrate outstanding abilities in natural language processing tasks that require deep understanding and complex reasoning, becoming valuable tools in the field of natural language processing. Currently, BERT [11] and its variants, such as RoBERTa [23], ALBERT [17], etc., are mainstream large language models based on the Encoder-only architecture. The following sections will introduce these models.

2.3.2 BERT Language Model

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained language model based on the Encoder-only architecture, introduced by the Google AI team in October 2018. As an early representative of the Encoder-only architecture, BERT brought about groundbreaking improvements in the field of natural language processing. Its core innovation lies in deeply mining the contextual information of text through a bidirectional encoder model, thereby providing excellent context embeddings for various downstream tasks. This section will introduce the structure of the BERT model, its pre-training method, and downstream tasks.

1. Structure of the BERT Model

The structure of the BERT model is almost identical to the encoder in the Transformer, consisting of multiple stacked encoder modules, each containing a multi-head self-attention module and a fully connected feedforward module. Depending on the number of parameters, there are two versions of the BERT model: BERT-Base and BERT-Large. **BERT-Base** consists of 12 stacked encoder modules, with a hidden layer dimension of 768 and 12 self-attention heads, **with a total of 110 million parameters**; **BERT-Large** consists of 24 stacked encoder modules, with a hidden layer dimension of 1024 and 16 self-attention heads, **with approximately 340 million parameters**.

2. BERT Pre-training Method

BERT uses the BookCorpus dataset [46] (containing about 800 million tokens) and the English Wikipedia dataset [6] (containing about 2.5 billion tokens) for pre-training, totaling about 3.3 billion tokens, with a total data size of around 15GB. In terms of pre-training tasks, BERT innovatively proposed two tasks—Masked Language Modeling (MLM) and Next Sentence Prediction (NSP)—to learn context embeddings. The complete pre-training process is shown in Figure 2.7.

Specifically, BERT constructs multiple sample sequences based on given raw text, with each sample sequence consisting of two sentences from the original text. There is a

---

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

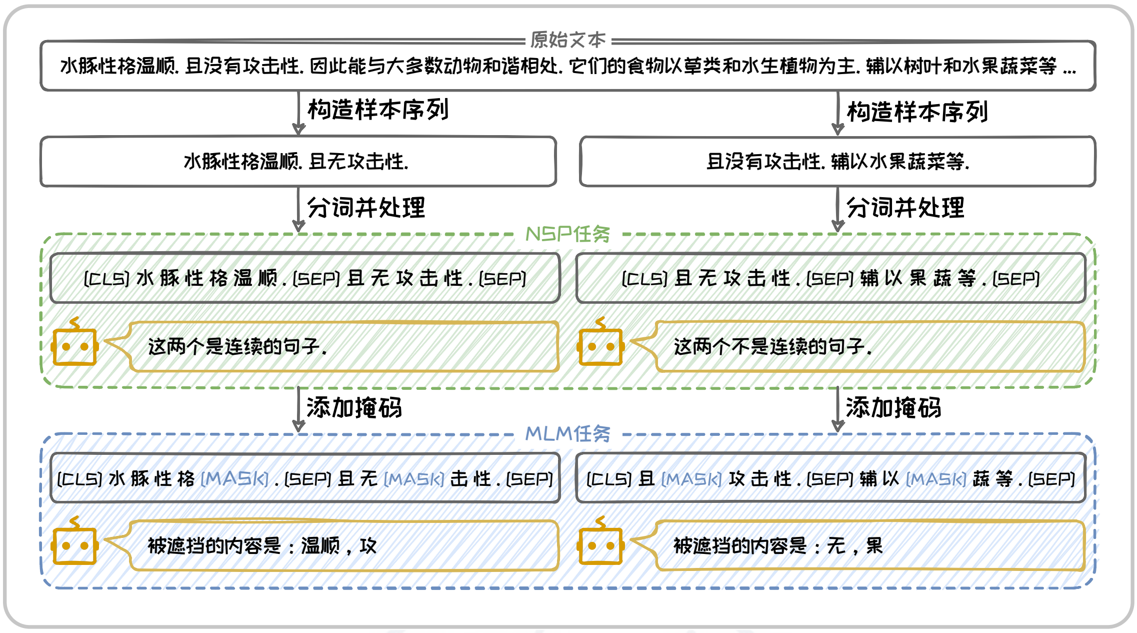

原始文本
水豚性格温顺. 且没有攻击性. 因此能与大多数动物和谐相处. 它们的食物以草类和水生植物为主. 辅以树叶和水果蔬菜等 ...
构造样本序列
构造样本序列
水豚性格温顺. 且无攻击性.
且没有攻击性. 辅以水果蔬菜等.
分词并处理
分词并处理
NSP任务
[CLS] 水 豚 性 格 温 顺 . [SEP] 且 无 攻 击 性 . [SEP]
[CLS] 且 无 攻 击 性 . [SEP] 辅 以 果 蔬 等 . [SEP]
这两个是连续的句子.
这两个不是连续的句子.
添加掩码
添加掩码
MLM任务
[CLS] 水 豚 性 格 [MASK] . [SEP] 且 无 [MASK] 击 性 . [SEP]
[CLS] 且 [MASK] 攻 击 性 . [SEP] 辅 以 [MASK] 蔬 等 . [SEP]
被遮挡的内容是：温顺，攻
被遮挡的内容是：无，果

Figure 2.7: Pre-training tasks of BERT.

50% chance that these two sentences are consecutive sentences from the original text, and a 50% chance that they are two randomly selected sentences. Then, the constructed sample sequences undergo tokenization, and special tags `[CLS]` are added at the beginning of the sequence, and special tags `[SEP]` are added at the end of each sentence. The `[CLS]` tag is used to aggregate information across the entire sequence, while the `[SEP]` tag clearly delineates the boundaries between sentences.

Next, BERT uses the processed sequence for the next sentence prediction task, using the model to determine whether the two sentences in the sample sequence **are consecutive**. This task trains BERT to recognize and understand the relationship between sentences, capturing sentence-level semantic features. This is particularly helpful for understanding the logical flow of text and the relationships between sentences, especially in NLP tasks such as question answering and natural language inference that require understanding the structural hierarchy of documents.

Finally, BERT randomly masks about 15% of the tokens in the sample sequence, replacing them with the special tag `[MASK]` or random words. The model needs to **predict the original content of these masked tokens**. This process is similar to a **cloze test**, requiring the model to infer missing tokens based on surrounding context information. The cross-entropy loss function used during the prediction process drives the optimization of parameters within the BERT model, enabling it to learn bidirectional context representations of text. Notably, during the training process of the MLM task, BERT only learns from those tokens that were randomly replaced, i.e., it only calculates the prediction loss for these tokens to update model parameters.

By combining these two pre-training tasks, BERT has made significant improvements in both the depth and breadth of language understanding. BERT not only captures fine-grained features of tokens but also grasps long-range dependencies and complex relationships between sentences, providing a solid foundation for language understanding for various downstream tasks.

3. Downstream Tasks of BERT

BERT can be applied to various natural language processing tasks, including but not limited to text classification (such as sentiment analysis), question answering systems, text matching (such as natural language inference), and semantic similarity calculation. However, since the output of BERT is the vector representation of all tokens in the input, the total length is not fixed, making it unsuitable for direct application in various downstream tasks. To address this issue, BERT designed the `[CLS]` tag to extract the **aggregated representation of the entire input sequence**. The `[CLS]` tag is a special marker specifically designed for classification and summarization tasks. Its full name is ”Classification Token”, which aggregates information from the entire input sequence via the attention mechanism, generating a fixed-length vector representation to summarize the information of all token sequences, facilitating the handling of various downstream tasks.

In **text classification tasks**, the vector corresponding to the `[CLS]` tag in the output can be extracted and passed to **a fully connected layer** for classification, for example, to determine whether the sentiment of the entire sentence is positive, negative, or neutral. In **question answering tasks**, the input includes a question and a related text, formatted as “`[CLS]` question `[SEP]` text `[SEP]`”. The vector corresponding to the `[CLS]` tag is ultimately extracted and passed to **two fully connected layers** to determine whether the answer exists in the related text. If it does, these two fully connected layers respectively output the start and end positions of the answer. Through this method, BERT can accurately extract the answer to the question from the provided paragraph. In **semantic similarity tasks**, the goal is to calculate the semantic similarity between two or more texts. For this task, the construction can be “`[CLS]` text1 `[SEP]` text2 `[SEP]`”, combined with **a linear layer** to directly output the similarity between the two texts; or **without adding additional components**, the vector corresponding to the `[CLS]` tag can be directly extracted, and then additional similarity measurement methods (such as cosine similarity) can be used to calculate the similarity between multiple texts.

Through bidirectional encoding and specific pre-training tasks, BERT has significantly improved the performance of natural language processing tasks. Its applications in various tasks have demonstrated strong generalization ability and practicality, not only providing new benchmarks for academic research but also being widely adopted in practical applications, greatly promoting the development of natural language processing technology.

2.3.3 BERT-derived Language Models

The groundbreaking success of BERT has also led to a series of related derivative models. These models inherit the core characteristics of BERT's bidirectional encoding and further improve and optimize them to enhance model performance or efficiency, demonstrating outstanding performance in specific tasks and application scenarios. Representative derivative models include RoBERTa [23], ALBERT [17], and ELECTRA [9], among others. The following sections will introduce these three models separately.

1. RoBERTa Language Model

RoBERTa (Robustly Optimized BERT Pretraining Approach) was introduced by Facebook AI (now renamed Meta) in July 2019, aiming to address the issue of **insufficient training extent** in BERT to **improve the performance of pre-trained language models**. RoBERTa optimized the pre-training process by adopting a **larger dataset** (including more English books, Wikipedia, and other web data), **longer training time** (including larger batch sizes and more training steps), and **more detailed hyperparameter tuning** (including settings for learning rate, training steps, etc.), thereby enhancing the model's performance and robustness in various natural language processing tasks. The following sections will introduce the structure, pre-training method, and downstream tasks of the RoBERTa model.

(1) Structure of the RoBERTa Model

The structure of RoBERTa is basically consistent with BERT, based on multiple stacked encoder modules, each containing a multi-head self-attention module and a fully connected feedforward module. RoBERTa also comes in two versions: RoBERTa-Base and RoBERTa-Large. **RoBERTa-Base** is comparable to BERT-Base, consisting of 12 stacked encoder modules, with a hidden layer dimension of 768, 12 self-attention heads, **and approximately 120 million parameters**; **RoBERTa-Large** is comparable to BERT-Large, consisting of 24 stacked encoder modules, with a hidden layer dimension of 1024, 16 self-attention heads, **and approximately 350 million parameters**.

(2) Pre-training Method of RoBERTa

In the selection of pre-training corpora, RoBERTa builds upon the existing BookCorpus dataset [46] (containing about 800 million tokens) and the English Wikipedia dataset [7] (containing about 2.5 billion tokens) used by BERT, adding the news dataset CC-News [8] (containing about 76GB of news articles), the open web dataset OpenWebText [9] (containing about 38GB of web text content), and the stories dataset Stories [41] (containing about 31GB of story text), bringing the total data volume to about 160GB.

Regarding specific pre-training tasks, RoBERTa removed the next sentence prediction task from BERT and changed the static masked language modeling task originally used by BERT to **dynamic masked language modeling**. Specifically, while BERT masks sentences during data preprocessing and maintains the mask positions unchanged throughout each training epoch, RoBERTa copies the training data into 10 replicas and applies masking separately. Under the same condition of training for 40 epochs, BERT trained on

---

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

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

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

the statically masked text 40 times, whereas RoBERTa trained each of the 10 differently masked replicas 4 times, thereby increasing the diversity of model training and helping the model learn richer contextual information.

These improvements enable RoBERTa to excel in understanding context and processing long texts, particularly in capturing subtle semantic differences and contextual dependencies.

2. ALBERT Language Model

ALBERT (A Lite BERT) is a lightweight BERT model proposed by the Google Research team in September 2019, aiming to reduce the model's parameter count and memory usage through parameter sharing and embedding decomposition techniques, thereby **enhancing training and inference efficiency**. The BERT-Base model has 110 million parameters, while the BERT-Large model has 340 million parameters, making BERT difficult to train and resulting in longer inference times. During the design process, ALBERT **significantly reduced the number of parameters** through parameter factorization and cross-layer parameter sharing techniques. The following sections will introduce the structure, pre-training method, and downstream tasks of the ALBERT model.

(1) Structure of the ALBERT Model

The structure of ALBERT is similar to BERT and RoBERTa, consisting of multiple stacked encoder modules. However, ALBERT significantly reduces the model's parameter count under the same model architecture through **parameter factorization** and **cross-layer parameter sharing**. The following sections will detail parameter factorization and cross-layer parameter sharing.

Parameter Factorization

In BERT, the output vector dimension E of the embedding layer is the same as the vector dimension H of the hidden layer, meaning that the output of the embedding layer is directly used as input to subsequent encoder modules. Specifically, the vocabulary size V of the BERT-Base model is around 30,000, and its hidden layer vector dimension H is set to 768. Therefore, the number of parameters required by the embedding layer of BERT is $V \times H$, approximately 23 million.

In contrast, ALBERT decomposes the matrix of the embedding layer, projecting the one-hot encoded vectors corresponding to the vocabulary down to a lower dimension E and then up-projecting them back to the dimension H of the hidden state. Specifically, ALBERT chooses a smaller embedding layer dimension, such as 128, and breaks down the number of parameters to $V \times E + E \times H$. According to this design, the embedding layer of ALBERT requires approximately 3.94 million parameters, about one-sixth of the number of parameters in BERT. For the Large version with a larger hidden layer vector dimension H, the advantage of ALBERT in saving parameter space is even more pronounced, reducing the parameter count to about one-eighth of BERT's.

Cross-layer Parameter Sharing

Taking the classic BERT-Base model as an example, the model consists of 12 layers of identically structured encoder modules, with the parameters of all Transformer blocks being independently trained. To reduce the model's parameter count, ALBERT introduced a cross-layer parameter sharing mechanism, where only the parameters of the first layer encoder module are learned and directly shared with all other layers. While this mechanism sacrifices some model performance to some extent, it significantly improves the compression ratio of parameter storage, achieving more efficient resource utilization.

Based on parameter factorization and cross-layer parameter sharing, ALBERT proposes four versions of models: ALBERT-Base, ALBERT-Large, ALBERT-XLarge, and

ALBERT-XXLarge. Among them, **ALBERT-Base** is comparable to BERT-Base, consisting of 12 stacked encoder modules, with an intermediate embedding factorization dimension of 128, a hidden layer dimension of 768, and 12 self-attention heads, **with a total of approximately 12 million parameters**; **ALBERT-Large** is comparable to BERT-Large, consisting of 24 stacked encoder modules, with an intermediate embedding factorization dimension of 128, a hidden layer dimension of 1024, and 16 self-attention heads, **with a total of approximately 18 million parameters**; **ALBERT-XLarge** consists of 12 stacked encoder modules, with an intermediate embedding factorization dimension of 128, a hidden layer dimension of 2048, and 16 self-attention heads, **with a total of approximately 60 million parameters**; **ALBERT-XXLarge** consists of 12 stacked encoder modules, with an intermediate embedding factorization dimension of 128, a hidden layer dimension of 4096, and 64 self-attention heads, **with a total of approximately 220 million parameters**.

(2) ALBERT Pre-training Method

ALBERT uses the same datasets as BERT for pre-training, namely the BookCorpus dataset [46] (containing about 800 million tokens) and the English Wikipedia dataset [10] (containing about 2.5 billion tokens), totaling 3.3 billion tokens and about 15GB of data. Additionally, in the choice of pre-training tasks, ALBERT retains the masked language modeling task from BERT and replaces the next sentence prediction task with **sentence order prediction (SOP)**, as shown in Figure 2.8. Specifically, ALBERT selects two consecutive sentences from the text, either concatenating them directly or reversing their order before concatenation, and uses the concatenated content as input samples. The model needs to predict whether the two sentences in the sample are in the correct order or re-

---

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

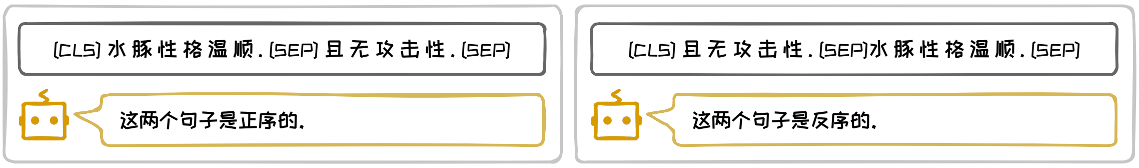

[CLS] 水豚性格温顺. [SEP] 且无攻击性. [SEP]
这两个句子是正序的.
[CLS] 且无攻击性. [SEP]水豚性格温顺. [SEP]
这两个句子是反序的.

Figure 2.8: Sentence Order Prediction task in ALBERT.

versed.

Compared to BERT, ALBERT significantly reduces the model's parameter count while maintaining good performance through innovative parameter sharing and parameter factorization techniques. This makes it more practical in resource-constrained environments, more efficient when handling large-scale datasets and complex tasks, and reduces the costs associated with model deployment and maintenance.

3. ELECTRA Language Model

ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately) [9] is another variant of BERT proposed by researchers from Google Brain and Stanford University in March 2020, aimed at addressing **efficiency and scalability issues** in large-scale pre-trained language models. By using a **generator-discriminator architecture**, ELECTRA can **more efficiently utilize pre-training data**, improving the model's performance on downstream tasks.

(1) ELECTRA Pre-training Method

In terms of model structure, ELECTRA combines the idea of Generative Adversarial Networks (GANs) with the masked language modeling of BERT, adopting a generator-discriminator architecture. Specifically, the ELECTRA model includes a generator and a discriminator. The generator (Generator) is a model capable of masked prediction (e.g., a BERT model) responsible for restoring masked text. The discriminator (Discriminator) uses the replaced token detection (RTD) pre-training task to identify whether each token in the generator's output is from the original text. The complete process is shown in Figure 2.9.

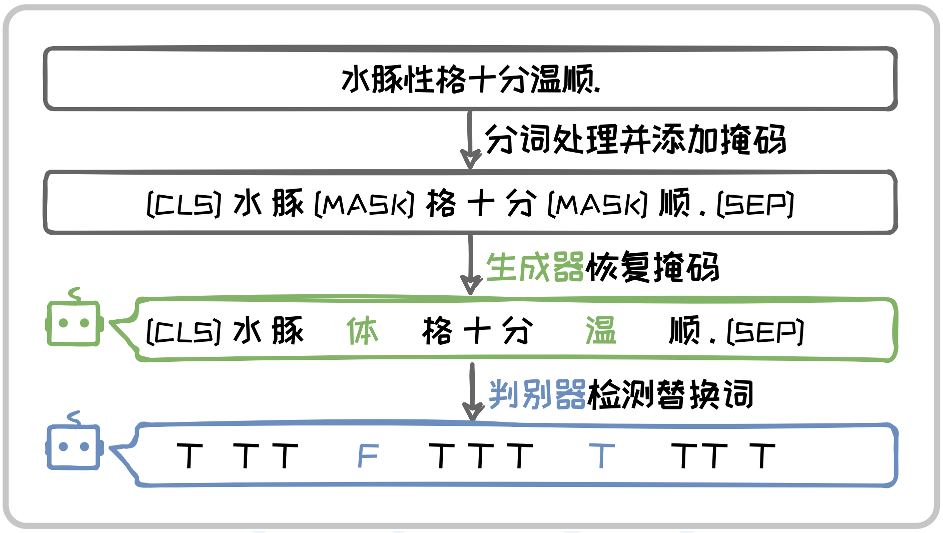


Figure 2.9: ELECTRA pre-training task.

ZHEJIANG UNIVERSITY 1897

(2) ELECTRA Model Structure

Depending on the scale of the generator and discriminator, ELECTRA proposes three versions of models: ELECTRA-Small, ELECTRA-Base, and ELECTRA-Large. Among them, **ELECTRA-Small** has both the generator and discriminator consisting of 12 stacked encoder modules, with a hidden layer dimension of 256 and 4 self-attention heads, so the parameter count for both the generator and discriminator is approximately 14 million, **with a total of approximately 28 million parameters**; **ELECTRA-Base** has both the generator and discriminator consisting of 12 stacked encoder modules, with a hidden layer dimension of 768 and 12 self-attention heads, so the parameter count for both the generator and discriminator is approximately 110 million, **with a total of approximately 220 million parameters**; **ELECTRA-Large** has both the generator and discriminator consisting of 24 stacked encoder modules, with a hidden layer dimension of 1024 and 16 self-attention heads, so the parameter count for both the generator and discriminator is approximately 330 million, **with a total of approximately 660 million parameters**.

Among these, ELECTRA-Small and ELECTRA-Base use the same datasets as BERT for pre-training, totaling 3.3 billion tokens. On the other hand, ELECTRA-Large uses **more diverse training data**, including the large-scale web dataset ClueWeb [11], Common-Crawl [12], and the large news text dataset Gigaword [13], expanding the dataset to 33 billion tokens, thereby helping the model learn more extensive language representations.

Additionally, in BERT, only 15% of tokens are masked, and the model is trained only on these 15% of tokens. However, in ELECTRA, the discriminator evaluates whether all tokens generated by the generator have been replaced, allowing it to better learn contextual

---

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

[12] https://commoncrawl.org

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

embeddings.

Unlike RoBERTa and ALBERT, which mainly optimize the model structure and pre-training data scale, ELECTRA introduces a new generator-discriminator architecture on the basis of BERT, significantly improving training efficiency and effectiveness by replacing the language modeling task. This also enhances the model's performance on downstream tasks.

The large language models based on the Encoder-only architecture have achieved good results in various natural language processing tasks such as text classification and sentiment analysis. Table 2.1 summarizes the above models in terms of model parameter counts and pre-training datasets. It can be seen that the parameter sizes of these classic models stop at around 660 million, and pre-training tasks primarily serve natural language understanding. These models do not continue to seek breakthroughs in parameter count and typically focus on discriminative tasks, making them less suitable for generative tasks. Therefore, their potential impact in the increasingly popular field of generative artificial intelligence is relatively limited.

These models showcase excellent performance in specific tasks and application scenarios through their unique innovations and optimizations. Table 2.1 summarizes some representative models mentioned in this chapter,

not only providing high-quality contextual embeddings for text but also achieving significant results in multiple natural language processing tasks, including but not limited to text classification and sentiment analysis. However, these models still focus on understanding at the text level, making it difficult for them to handle generative tasks. As such, their role in the increasingly popular field of generative artificial intelligence (Artificial Intelligence Generated Content, AIGC) is limited.

Table 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 Large Language Models Based on the Encoder-Decoder Architecture

The Encoder-Decoder architecture introduces a Decoder component on top of the Encoder-only architecture to accomplish **sequence-to-sequence** (Sequence to Sequence, Seq2Seq) tasks such as machine translation. This section will introduce the Encoder-Decoder architecture and its representative models.

2.4.1 Encoder-Decoder Architecture

The Encoder-Decoder architecture primarily consists of two parts: the encoder and the decoder. The detailed composition of this architecture is shown in Figure 2.10. The encoder part is identical to the encoder in the Encoder-only architecture, comprising multiple stacked encoder modules, each of which contains a self-attention module and a fully connected feedforward module. After passing through the encoder part, the input sequence is transformed into a fixed-size context vector that contains rich semantic information of the input sequence. Similarly, the decoder is made up of multiple stacked decoder modules, each of which consists of a masked self-attention module, a cross-attention module, and a fully connected feedforward module. The masked self-attention module introduces a masking mechanism to prevent the ”leakage” of future information, ensuring the autoregressive property of the decoding process. The cross-attention module facilitates the interaction between the decoder and the encoder, which is crucial for generating output that is highly relevant to the input sequence.

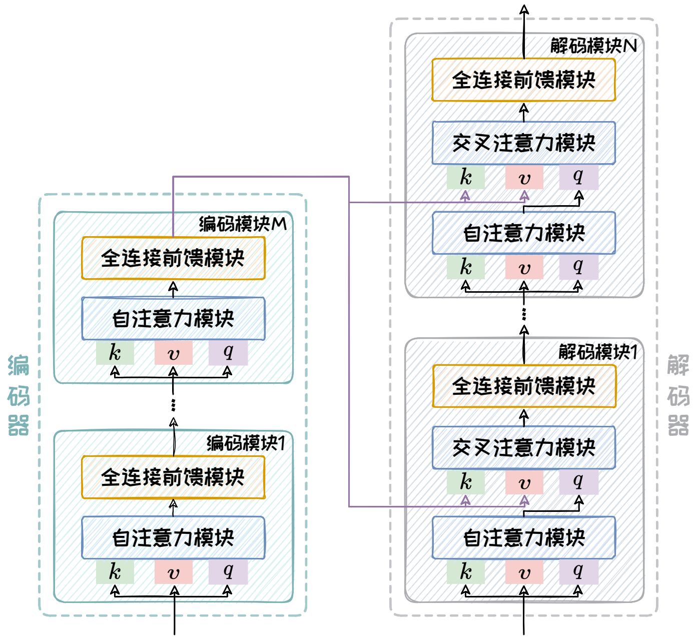


Figure 2.10: Encoder-Decoder architecture

ZHEJIANG UNIVERSITY 1897

The attention targets of the self-attention modules differ between the encoder and the decoder. In the encoder, we need to take into account the **context** of the input sequence comprehensively; therefore, a bidirectional attention mechanism is adopted to capture contextual information thoroughly. However, in the decoder, the self-attention mechanism is unidirectional, conditioning on the **previous context** to decode the **subsequent text**, and avoiding the decoder from ”peeking” at future information through masking operations. Cross-attention achieves effective information exchange between the two modules by combining the decoder’s query (query) with the encoder’s keys (key) and values (value).

By combining self-attention and cross-attention mechanisms, the Encoder-Decoder architecture can **efficiently encode input information** and **generate high-quality output**

sequences.

The self-attention mechanism ensures consistency and coherence within the input sequence and the generated sequence, while the cross-attention mechanism ensures that the decoder can refer to the global context information of the input sequence when generating each output token, thus producing results that are highly relevant to the input content. With the joint effect of these two mechanisms, the Encoder-Decoder architecture not only deeply understands the input sequence but can also flexibly generate output sequences of appropriate length according to the requirements of different tasks. It has been widely applied in tasks such as machine translation, text summarization, and question answering systems, achieving remarkable results. This section will introduce two typical large language models based on the Encoder-Decoder architecture: T5 [29] and BART [18].

2.4.2 T5 Language Model

Natural language processing encompasses a variety of tasks, including language translation, text summarization, and question answering systems. Typically, each task requires customized design of training data, model architecture, and training strategies. Such customization is time-consuming and labor-intensive, and the trained models are often difficult to reuse across tasks, leading developers to continuously ”reinvent the wheel.” To address this issue, the Google Research team proposed a large pre-trained language model based on the Encoder-Decoder architecture, T5 (Text-to-Text Transfer Transformer), in October 2019. T5 adopts a unified text-to-text transformation paradigm to handle multiple tasks. The following sections will introduce the T5 model from three aspects: model structure, pre-training method, and downstream tasks.

1. T5 Model Structure

The core idea of the T5 model is to **unify various NLP tasks into a single text-to-text generation framework**. Under this unified framework, T5 indicates the execution of different tasks through different input prefixes, then generates corresponding task outputs, as shown in Figure 2.11. This approach can be seen as an early application of prompt technology, where constructing appropriate input prefixes enables the T5 model to optimize itself for specific tasks without fundamental changes to the model architecture. This flexibility and task generalization capability significantly enhance the model's practicality, making it easy to adapt to a wide range of new NLP tasks.

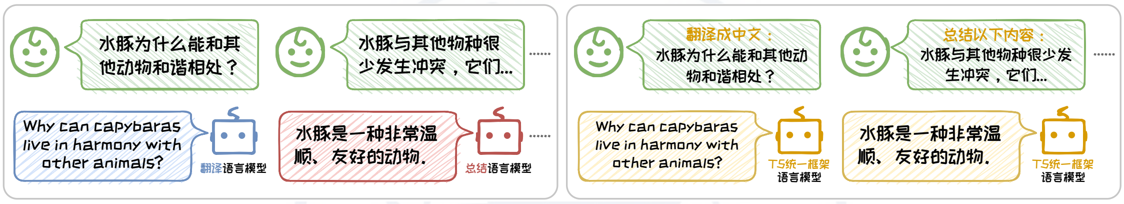

水豚为什么能和其他动物和谐相处？
水豚与其他物种很少发生冲突，它们...
......
Why can capybaras live in harmony with other animals?
翻译语言模型
水豚是一种非常温顺、友好的动物.
总结语言模型
......
翻译成中文：水豚为什么能和其他动物和谐相处？
总结以下内容：水豚与其他物种很少发生冲突，它们...
......
Why can capybaras live in harmony with other animals?
T5统一框架语言模型
水豚是一种非常温顺、友好的动物.
T5统一框架语言模型

Figure 2.11: Traditional language model and the unified framework of T5.

In terms of model architecture, T5 is identical to the original Transformer architecture, which includes an encoder and a decoder. Each encoder and decoder consists of multiple stacked encoder and decoder modules, respectively. T5 provides five different versions based on specific parameter settings: T5-Small, T5-Base, T5-Large, T5-3B, and T5-11B. **T5-Small** comprises 6 encoder modules and 6 decoder modules, with a hidden layer dimension of 512 and 8 self-attention heads, **with a total of approximately 60 million parameters**; **T5-Base** is comparable to BERT-Base, comprising 12 encoder modules and 12 decoder modules, with a hidden layer dimension of 768 and 12 self-attention heads, **with a total of approximately 220 million parameters**; **T5-Large** is comparable to BERT-Large, comprising 24 encoder modules and 24 decoder modules, with a hidden layer dimension of 1024 and 16 self-attention heads, **with a total of approximately**

770 million parameters; **T5-3B** builds upon T5-Large by increasing the number of self-attention heads to 32 and quadrupling the dimension of the intermediate layer in the fully connected feedforward network, **with a total of approximately 2.8 billion parameters**; **T5-11B** further increases the number of self-attention heads to 128 and again quadruples the dimension of the intermediate layer in the fully connected feedforward network based on T5-3B, **with a total of approximately 11 billion parameters**.

2. T5 Pre-training Method

To obtain a high-quality, broadly covered pre-training dataset, the Google Research team extracted a large amount of web data from the large-scale web dataset Common Crawl [14], and after rigorous cleaning and filtering, ultimately generated the C4 dataset (Colossal Clean Crawled Corpus). This dataset covers a variety of websites and text types, with a total size of approximately 750GB.

Based on this dataset, T5 proposes a pre-training task named Span Corruption. This pre-training task involves selecting 15% of the Tokens from the original input for corruption, with each selection being a continuous span of three Tokens that are masked as a whole into `[MASK]`. Unlike the single-Token prediction used in the BERT model, the T5 model needs to **predict the entire masked continuous text span**. These spans may include continuous phrases or clauses that form meaningful semantic units in natural language. This design requires the model to not only understand the surface forms of local vocabulary but also to capture deeper sentence structures and complex dependencies between contexts. The Span Corruption pre-training task significantly improves the performance of T5, especially in text summarization, question answering systems, and text completion tasks, which require generating coherent and logically strong text.

---

[14] https://commoncrawl.org

3. T5 Downstream Tasks

Based on the extensive knowledge learned during the pre-training phase and the newly proposed unified text-to-text generation framework, the T5 model can be directly adapted to various downstream tasks in a completely **zero-shot** (Zero-Shot) manner using prompt engineering technology. At the same time, the T5 model can also be fine-tuned to adapt to specific tasks. However, the fine-tuning process requires collecting labeled training data for downstream tasks and also necessitates more computational resources and training time, so it is usually only applied to applications with extremely high precision requirements and more complex tasks.

In summary, the unified text-to-text generation framework of the T5 model not only simplifies the conversion process between different natural language processing tasks but also provides a new direction for the development of large language models. Today, the T5 model has spawned many variants to further improve its performance. For example, the mT5[43] model extends support to over 100 languages, the T0[31] model enhances zero-shot learning (Zero-Shot Learning) capabilities through multi-task training, and the Flan-T5[8] model focuses on instruction fine-tuning to achieve further improvements in model flexibility and efficiency, among others.

2.4.3 BART Language Model

BART (Bidirectional and Auto-Regressive Transformers) is an Encoder-Decoder architecture model proposed by Meta AI Research Institute in October 2019. Unlike T5, which integrates various NLP tasks into a unified framework, BART aims to enhance the model's performance in both text generation tasks and text understanding tasks through diversified pre-training tasks.

1. BART Model Structure

The model structure of BART is identical to the original Transformer architecture, including an encoder and a decoder. Each encoder and decoder consists of multiple stacked encoder and decoder modules, respectively. BART offers two versions: BART-Base and BART-Large. **BART-Base** comprises 6 encoder modules and 6 decoder modules, with a hidden layer dimension of 768 and 12 self-attention heads, **with a total of approximately 140 million parameters**; **BART-Large** comprises 12 encoder modules and 12 decoder modules, with a hidden layer dimension of 1024 and 16 self-attention heads, **with a total of approximately 400 million parameters**.

2. BART Pre-training Method

For pre-training data, BART uses the same corpus as RoBERTa [23], which includes the BookCorpus dataset [46], the English Wikipedia dataset [15], the news dataset CC-News [16], the web open dataset OpenWebText [17], and the stories dataset, with a total data volume of approximately 160GB.

For pre-training tasks, BART aims to reconstruct corrupted text. It corrupts the text

---

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

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

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

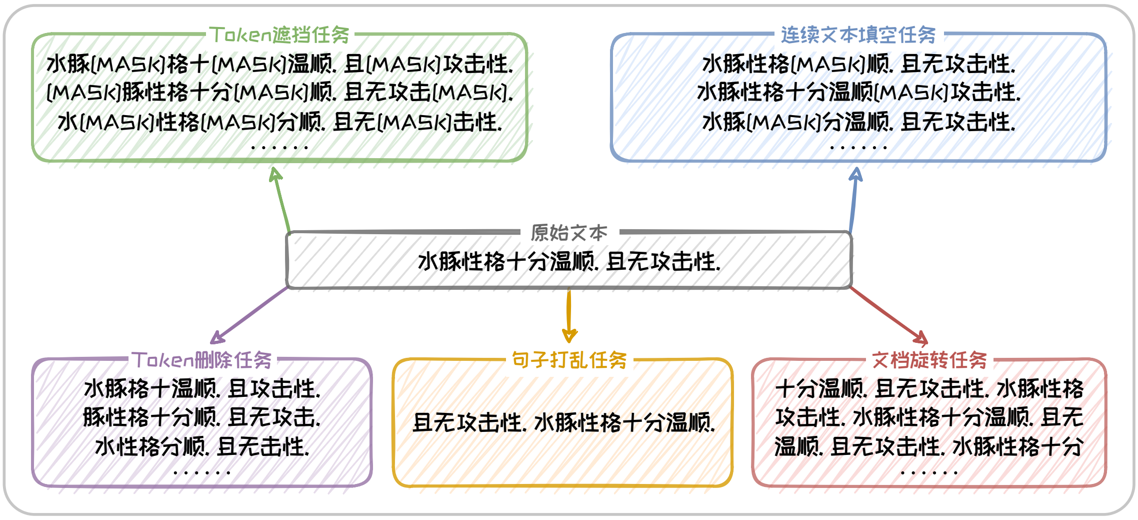

Token遮挡任务
水豚[MASK]格十[MASK]温顺. 且[MASK]攻击性.
[MASK]豚性格十分[MASK]顺. 且无攻击[MASK].
水[MASK]性格[MASK]分顺. 且无[MASK]击性.
......
连续文本填空任务
水豚性格[MASK]顺. 且无攻击性.
水豚性格十分温顺[MASK]攻击性.
水豚[MASK]分温顺. 且无攻击性.
......
原始文本
水豚性格十分温顺. 且无攻击性.
Token删除任务
水豚格十温顺. 且攻击性.
豚性格十分顺. 且无攻击.
水性格分顺. 且无击性.
......
句子打乱任务
且无攻击性. 水豚性格十分温顺.
文档旋转任务
十分温顺. 且无攻击性. 水豚性格
攻击性. 水豚性格十分温顺. 且无
温顺. 且无攻击性. 水豚性格十分
......

Figure 2.12: BART pre-training tasks.

through five tasks: Token Masking, Token Deletion, Text Infilling, Sentence Permutation, and Document Rotation, and then trains the model to restore the original text. This approach exercises the model's deep understanding of text structure and semantics, enhancing its robustness when dealing with incomplete or damaged information. The specific forms of the five text corruption tasks are described below.

- **Token Masking Task**: Similar to the MLM task in BERT, a portion of Tokens are randomly sampled from the original text and replaced with `[MASK]`, thereby training the model to **infer the content of the deleted Tokens**.
- **Token Deletion Task**: A portion of Tokens are randomly deleted from the original text, thereby training the model to **infer the positions and content of the deleted Tokens**.
- **Text Infilling Task**: Similar to the pre-training task in T5, several consecutive segments of Tokens (each segment as a span) are selected from the original text and replaced as a whole with `[MASK]`. The length of the span follows a Poisson distribution with $\lambda = 3$, and if the length is 0, a `[MASK]` is inserted directly. This task aims

to train the model to **infer a span and its length**.

- **Sentence Permutation Task**: The given text is split into multiple sentences, and the order of the sentences is randomly shuffled. This task aims to **train the model to infer the relationship between sentences**.
- **Document Rotation Task**: A Token is randomly selected from the given text to serve as the new starting point of the text, performing a rotation. This task aims to **train the model to find a reasonable starting point of the text**.

In Figure 2.12, using the given text ”Waterbucks have a very gentle personality. And they are non-aggressive.” as an example, the five tasks are demonstrated. After pre-training, BART can be fine-tuned to transfer the language knowledge learned during the pre-training phase to specific application scenarios, adapting to various downstream tasks. This process from pre-training to fine-tuning allows BART to excel not only in text generation tasks but also to meet the challenges of text understanding tasks. Subsequently, various variants of the BART model have also emerged, including the mBART [22] model capable of handling cross-lingual text generation tasks, among others.

In summary, large language models based on the Encoder-Decoder architecture have demonstrated good performance in generative tasks. Table 2.2 summarizes the models mentioned in this chapter based on the Encoder-Decoder architecture in terms of model parameter counts and the scale of pre-training corpora. It can be seen that the upper limit of the number of model parameters has reached 11 billion. With advantages in both model structure and parameter scale, compared to models based on the Encoder-only architecture, these models have achieved better results in tasks such as translation, summarization, and question answering.

Table 2.2: Table of Representative Models with Encoder-Decoder Architecture, Their Parameter Counts, and Corpus Sizes.

Model	Release Date	Parameter Count (billion)	Corpus Size
T5	Oct 2019	0.6 - 110	750GB
mT5	Oct 2020	3 - 130	9.7TB
T0	Oct 2021	30 - 110	Approximately 400GB
BART	Oct 2019	1.4 - 4	Approximately 20GB
mBART	Jun 2020	0.4 - 6.1	Approximately 1TB

2.5 Large Language Models Based on Decoder-Only Architecture

In open-ended generation tasks, input sequences are often simple or even lack a specific and clear input, making it unnecessary to maintain a full encoder to process these inputs. For such tasks, the Encoder-Decoder architecture may appear **too complex and lacking in flexibility**. In this context, the Decoder-Only architecture performs better. This section will introduce the Decoder-Only architecture and its representative models.

2.5.1 Decoder-Only Architecture

It generates text word by word through an autoregressive method, not only maintaining the coherence and intrinsic consistency of long texts but also generating text more naturally and smoothly in the absence of explicit input or in the presence of complex inputs. Moreover, due to the removal of the encoder part, the Decoder-Only architecture makes the model **lighter**, thus **accelerating the speed of training and inference**. Therefore, under the same model scale, the Decoder-Only architecture may perform better.

It is worth noting that the concept of the Decoder-Only architecture dates back to the GPT-1 model [27] released in 2018. However, at that time, due to the excellent performance demonstrated by Encoder-Only architecture models like BERT in various tasks, the Decoder-Only architecture did not receive sufficient attention. It was not until 2020, with the groundbreaking success of GPT-3 [5], that the Decoder-Only architecture began to be widely applied in various large language models, among which the most popular are the GPT series proposed by OpenAI and the LLaMA series proposed by Meta. Among these, the GPT series, being the earliest to adopt the Decoder-Only architecture, has set benchmarks in performance. However, starting from the third generation, the GPT series **gradually moved towards a closed-source model**. While the LLaMA series, despite a later start, has secured a place in the field of Decoder-Only architectures thanks to its equally impressive performance and **consistent commitment to open-source development**. The following sections will introduce these two series of models.

2.5.2 GPT Series Language Models

The GPT (Generative Pre-trained Transformer) series of models are a series of large language models based on the Decoder-Only architecture developed by OpenAI. Since its debut in 2018, the GPT series has undergone rapid development, continuously evolving in terms of model scale and pre-training paradigms, achieving remarkable results that have led the wave of development in large language models. Its evolution can be divided into five stages, with Table 2.3 summarizing the model parameter scale and dataset size for these five stages. From the table, it is evident that the GPT series models have seen a **sharp increase in parameter scale and pre-training dataset size**. However, starting from the ChatGPT version, the GPT series models have shifted to a closed-source model, and the specific parameter count and detailed information about the pre-training datasets are no longer publicly available. Nevertheless, based on scaling laws, it is reasonable to speculate that ChatGPT and its subsequent versions have increased in both parameter scale and pre-training dataset size. The following sections will provide an introduction to the models from these five developmental stages.

Table 2.3: Table of GPT Series Model Parameters and Dataset Sizes.

Model	Release Date	Parameter Count (in billions)	Dataset Size
GPT-1	2018.06	1.17	Approximately 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	Unknown	Unknown
GPT-4	2023.03	Unknown	Unknown
GPT-4o	2024.05	Unknown	Unknown

1. Initial Debut: GPT-1 Model

Former Chief Scientist of OpenAI, Ilya Sutskever, revealed in an interview [18] that OpenAI started exploring how to solve unsupervised learning problems through next-token prediction from its early days. However, the RNN models used at the time were unable to adequately address the issue of long-term dependencies, leaving these problems unresolved. It wasn't until 2017 when the Transformer emerged, offering a new solution to this problem and pointing the way forward for OpenAI's development. Following this, OpenAI began to get on track. In June 2018, OpenAI released the first version of the GPT (Generative Pre-Training) model, known as GPT-1 [27]. GPT-1 pioneered the use of the Decoder-Only architecture for solving unsupervised text generation through next-token

---

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

prediction, bringing revolutionary changes to the field of natural language processing. The following sections will detail the model structure, pre-training, and downstream tasks of GPT-1.

(1) GPT-1 Model Structure

In terms of model architecture, GPT-1 utilized the Decoder portion of the Transformer architecture, omitting the Encoder part and the cross-attention module. The model consists of a stack of 12 decoder blocks, with each block containing a masked self-attention module and a fully connected feed-forward module. The hidden layer dimension is 768, the number of self-attention heads is 12, and the total number of parameters in the model is approximately 117 million.

Figure 2.13 compares the model structures of BERT-Base and GPT-1. From the figure, it can be observed that GPT-1 closely resembles BERT-Base in structure, with both consisting of 12 encoding or decoding modules, each of which comprises a self-attention module and a fully connected feed-forward module. The fundamental difference lies in the fact that the self-attention module in BERT-Base employs a bidirectional self-attention mechanism, whereas the self-attention module in GPT-1 uses a **masked unidirectional** self-attention mechanism.

(2) GPT-1 Pre-training Method

GPT-1 was pre-trained on the BookCorpus dataset [46], which contains approximately 800 million tokens and totals nearly 5GB in data volume. In terms of pre-training methods, GPT-1 adopted the next-token prediction task, i.e., predicting the next possible token based on the given context. By continuously completing the next-token prediction task in an autoregressive manner, the model can effectively accomplish text generation tasks, as shown in Figure 2.14. Through this pre-training strategy, the model can learn

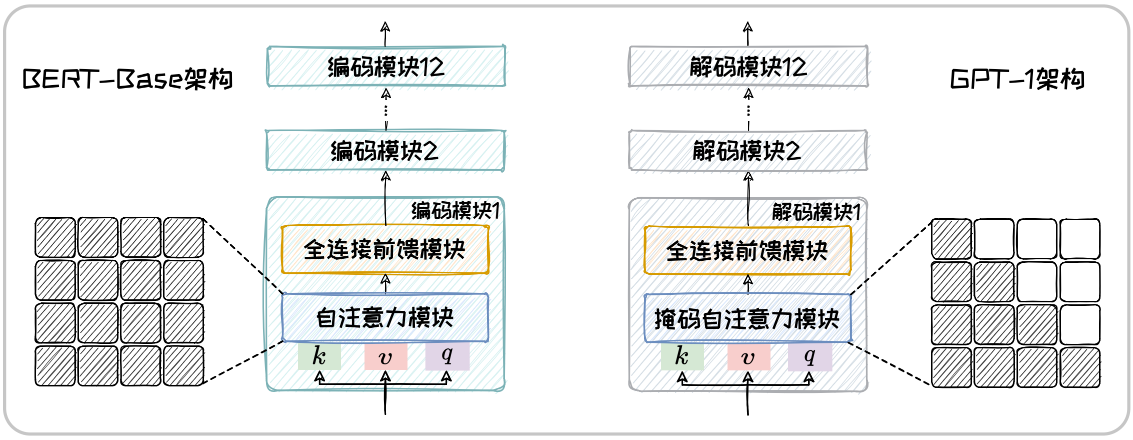


Figure 2.13: Model Structures of BERT-Base and GPT-1.

a lot of ”common sense” about language without the need for a large amount of labeled data, learning to generate coherent and contextually relevant text. This not only improves the model’s generalization ability but also reduces the reliance on annotated data, opening up new research directions and application prospects in the field of natural language processing.

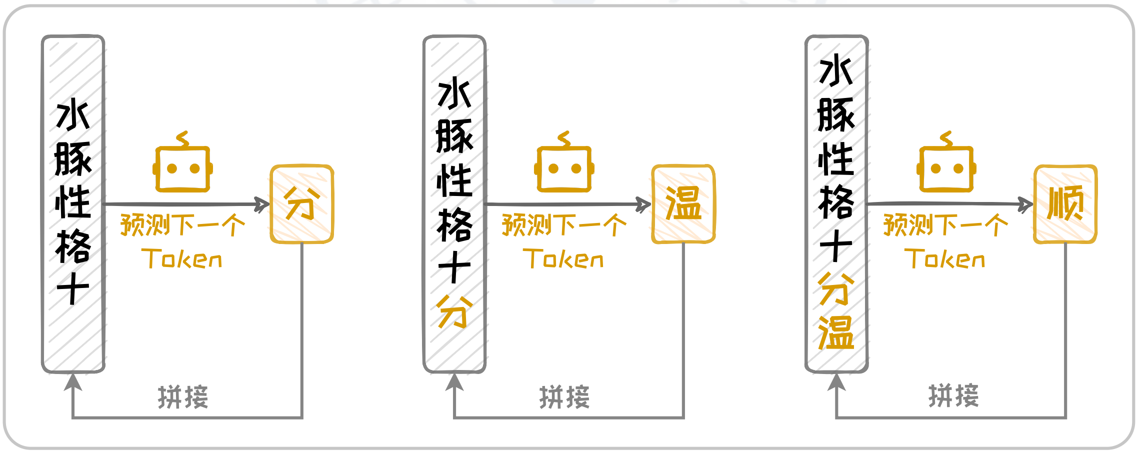


Figure 2.14: Pre-training Task for Language Modeling in GPT-1.

(3) Downstream Tasks of GPT-1

Despite the potential shown by the GPT-1 model after pre-training, its task generalization capabilities were still limited by the training data volume and the number of model parameters at the time. To improve the model's performance on specific downstream tasks, further supervised fine-tuning is typically required. The fine-tuning process involves optimizing the model's parameters using annotated data specific to the task, with both the input and output presented as text sequences. For instance, in the following tasks, we need to develop fine-tuning strategies tailored to specific application scenarios:

- **Text Classification**: GPT-1 can take a piece of text as input and classify it according to predefined category labels, such as sentiment polarity (positive, negative, or other). This is particularly useful in scenarios such as sentiment analysis and topic classification.
- **Text Similarity Assessment**: When measuring the similarity between two pieces of text, GPT-1 can analyze and quantify their content and semantic similarity. This function is especially important in comparing documents, optimizing search results, and recommendation systems.
- **Multiple Choice Question Answering**: GPT-1 can also handle multiple-choice questions. The model can understand the question text and option content, identify, and output the most suitable answer from the given options.

GPT-1 possesses native text generation capabilities. However, due to limitations in training data volume and model parameter count, its generative capacity was insufficient for solving practical problems. Additionally, its unidirectional attention mechanism restricted its comprehensive understanding of context. Four months later, BERT, which features bidirectional context understanding, was introduced and quickly gained widespread attention in the industry due to its powerful contextual embedding capabilities, overshadowing GPT-1. Although GPT-1 might have been less practical than BERT at the time, it marked the beginning of the Decoder-only architecture, paving the way for the impressive performances of subsequent large language models.

2. Steady Progress: GPT-2 Model

Despite the initial limitations of GPT-1, OpenAI did not deviate from its technical path of the Decoder-only architecture but instead chose to deepen its efforts along this route. In February 2019, OpenAI released the second-generation product of the GPT series, GPT-2 [19]. Compared to GPT-1, GPT-2 saw significant improvements in model scale and the quality of pre-training samples, notably enhancing the model's task generalization capabilities. The following sections will introduce the model structure, pre-training corpus, and downstream task adaptation of GPT-2.

(1) GPT-2 Model Structure

GPT-2 continued the Decoder-only architecture of GPT-1 and further increased the number of parameters. GPT-2 was released in four versions: GPT-2 Small, GPT-2 Medium, GPT-2 Large, and GPT-2 XL. Among them, **GPT-2 Small** is similar in scale to GPT-1 and BERT-Base, consisting of 12 encoder blocks with a hidden layer dimension of 768 and 12 self-attention heads, **with a total of approximately 124 million parameters**; **GPT-2 Medium** is similar in scale to BERT-Large, consisting of 24 decoder blocks with a hidden layer dimension of 1024 and 16 self-attention heads, **with a total of approximately 355 million parameters**; **GPT-2 Large** consists of 36 decoder blocks with a hidden layer dimension of 1280 and 20 self-attention heads, **with a total of approximately 774 million parameters**; **GPT-2 XL** is the largest version, consisting of 48 decoder blocks with a hid-

---

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

den layer dimension of 1600 and 25 self-attention heads, **with a total of approximately 1.5 billion parameters**.

(2) GPT-2 Pre-training Method

In pre-training, GPT-2 continued to use the next-token prediction task but further improved the quantity and quality of pre-training data. It adopted a new WebText dataset, comprising 40GB of carefully selected and cleaned web texts. By pre-training on the WebText dataset, GPT-2 significantly enhanced its language understanding capabilities, exposing it to a wider variety of language usage scenarios and learning more complex language expressions. This made GPT-2 more precise in capturing linguistic nuances and building language models, thereby generating more accurate and coherent text when performing various NLP tasks.

(3) GPT-2 Downstream Tasks

GPT-2's task generalization capabilities were improved, and it could perform **zero-shot learning** on some tasks without fine-tuning. This capability greatly increased the flexibility of GPT-2 in handling downstream tasks and reduced the cost of adapting to downstream tasks. This garnered more attention for the Decoder-only architecture.

3. Rising Star: GPT-3 Model

To further enhance task generalization capabilities, OpenAI launched the third-generation model, GPT-3 [5], in June 2020. Compared to the first two generations, GPT-3 saw further increases in model scale and pre-training corpus, and it exhibited excellent in-context learning (ICL) capabilities. With the support of in-context learning, GPT-3 can complete a variety of downstream tasks without fine-tuning, simply through task descriptions or a few examples. For a detailed introduction to in-context learning, see Chapter 3.2 of this book. The following sections will introduce the model structure, pre-training corpus, and downstream task adaptation of GPT-3.

(1) GPT-3 Model Architecture

In terms of model architecture, GPT-3 inherited and expanded upon the architecture of the first two generations, significantly increasing the number of decoder blocks, the dimension of hidden layers, and the number of self-attention heads, with the maximum number of parameters reaching 175 billion. The massive number of parameters allows GPT-3 to capture more subtle and complex language patterns, significantly enhancing its text generation capabilities. GPT-3 was designed with multiple versions of different parameter scales to meet the needs of various application scenarios, with detailed parameter specifications listed in Table 2.4.

Table 2.4: Detailed Parameters of GPT-1 to GPT-3 Models.

Model Version	Number of Decoder Blocks	Hidden Layer Dimension	Number of Self-Attentio
GPT-1	12	768	12
GPT-2 Small	12	768	12
GPT-2 Medium	24	1024	16
GPT-2 Large	36	1280	20
GPT-2 XL	48	1600	36
GPT-3 Small	12	768	12
GPT-3 Medium	24	1024	16
GPT-3 Large	24	1536	16
GPT-3 XL	24	2048	24
GPT-3 2.7B	32	2560	32
GPT-3 6.7B	32	4096	32
GPT-3 13B	40	5120	40
GPT-3 175B	96	12288	96

ZHEJIANG UNIVERSITY 1897

(2) GPT-3 Pre-training Method

Following the pre-training methods of the previous two generations, GPT-3 continues to use next-token prediction as the pre-training task. It utilizes a larger and more diverse internet text dataset, with a data volume approaching 1TB, covering sources such as

Common Crawl [20], WebText, BookCorpus[46], Wikipedia [21], and others, including books, websites, forum posts, and various forms of text. All data have undergone strict screening and cleaning processes to ensure data quality and diversity. Based on this data, GPT-3 has learned richer and more diverse language knowledge and world knowledge.

(3) GPT-3 Downstream Tasks

The GPT-3 model has demonstrated excellent in-context learning capabilities, enabling it to perform a variety of downstream tasks without fine-tuning by clearly describing the task and providing a few examples in the input text. This in-context learning capability significantly enhances GPT-3's task generalization ability, allowing it to quickly adapt to different application scenarios. GPT-3 has begun to shine in numerous natural language processing tasks, including text generation, question-answering systems, and language translation.

4. Ready for Action: InstructGPT and Other Models

Building on GPT-3, OpenAI has further introduced a series of derivative models. These models, built upon the GPT-3 network structure, have been trained using specific methods, each with unique capabilities. For example, the Codex[6] model, which has undergone continual pre-training with billions of lines of code, can effectively handle code generation tasks; the InstructGPT[25] model, which aligns with user intent, has excellent instruction-following capabilities. Among these, the most inspiring is InstructGPT, which is also the predecessor of ChatGPT. It has significantly improved the model's response to user instructions by introducing **Reinforcement Learning from Human Feedback (RLHF)**.

Reinforcement Learning from Human Feedback aims to alleviate inaccuracies and

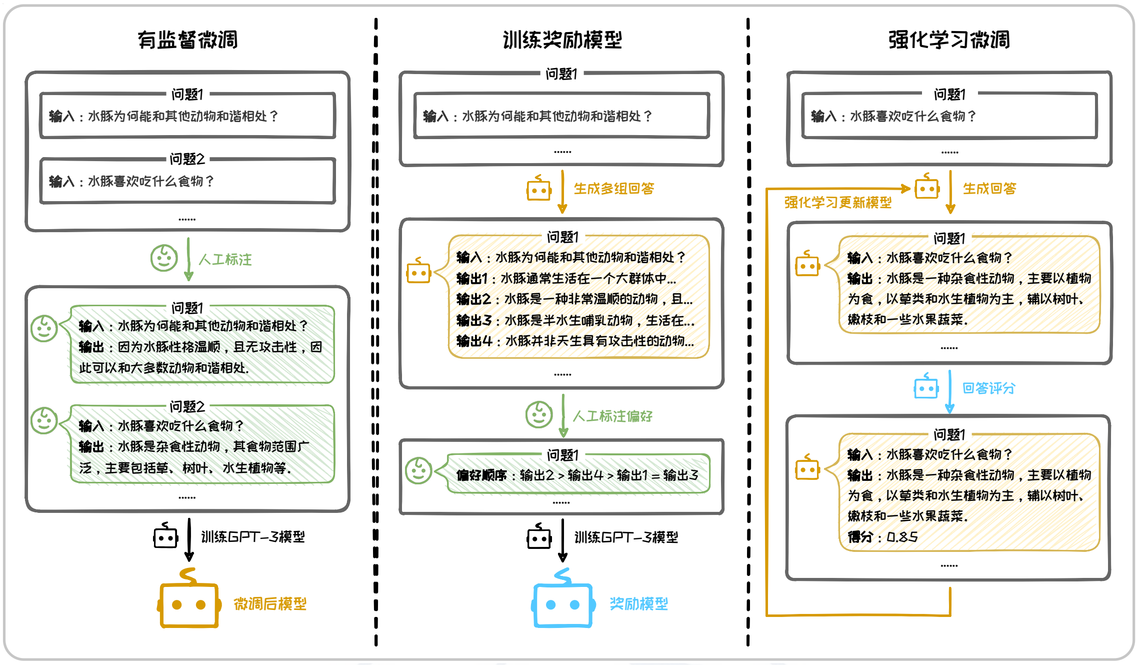
---

[20] https://commoncrawl.org

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

unreliability in the model's adherence to user instructions, ensuring that the generated content better meets human requirements. In RLHF, human evaluators first provide feedback on the quality of model outputs, and then this feedback is used to fine-tune the model. The specific process, as shown in Figure 2.15, can be divided into three steps: 1) **Supervised Fine-tuning**: Collect a large number of ”question-human response” pairs as training samples to fine-tune the large language model. 2) **Training a Reward Model**: Generate multiple candidate outputs for each input and have humans evaluate and rank them to form a preference dataset. Use this preference dataset to **train a reward model** that can score whether the output aligns with human preferences. 3) **Reinforcement Learning Fine-tuning**: Based on the reward model obtained in the previous step, **use reinforcement learning methods to optimize the language model from the first step**. After the language model generates an output, the reward model scores it, and the reinforcement learning algorithm adjusts the model parameters based on these scores to increase the probability of high-quality outputs.

The performance of InstructGPT, trained with RLHF, generally surpasses that of GPT-3, especially in scenarios requiring precise adherence to user instructions. The responses generated by InstructGPT are more aligned with user queries, effectively reducing the generation of irrelevant or misleading content. The research on InstructGPT provides new insights into building smarter and more reliable AI systems, showcasing the immense potential of combining human wisdom with machine learning algorithms. However, the computational cost of RLHF is very high, mainly due to: 1) the **complex and time-consuming training process of the reward model**. 2) In addition to separately training the language model and the reward model, there is a need to coordinate the two models for **joint multi-model training**, a process that is equally complex and resource-



问题1
输入：水豚为何能和其他动物和谐相处？

问题2
输入：水豚喜欢吃什么食物？

问题1
输入：水豚为何能和其他动物和谐相处？
输出：因为水豚性格温顺，且无攻击性，因此可以和大多数动物和谐相处.

问题2
输入：水豚喜欢吃什么食物？
输出：水豚是杂食性动物，其食物范围广泛，主要包括草、树叶、水生植物等.

问题1
输入：水豚为何能和其他动物和谐相处？

问题1
输入：水豚为何能和其他动物和谐相处？
输出1：水豚通常生活在一个大群体中...
输出2：水豚是一种非常温顺的动物，且...
输出3：水豚是半水生哺乳动物，生活在...
输出4：水豚并非天生具有攻击性的动物...

问题1
偏好顺序：输出2 > 输出4 > 输出1 = 输出3

问题1
输入：水豚喜欢吃什么食物？

问题1
输入：水豚喜欢吃什么食物？
输出：水豚是一种杂食性动物，主要以植物为食，以草类和水生植物为主，辅以树叶、嫩枝和一些水果蔬菜.

问题1
输入：水豚喜欢吃什么食物？
输出：水豚是一种杂食性动物，主要以植物为食，以草类和水生植物为主，辅以树叶、嫩枝和一些水果蔬菜.
得分：0.85

Figure 2.15: Process of Reinforcement Learning from Human Feedback (RLHF).

intensive.

To overcome the computational efficiency issues of RLHF, Stanford University proposed a new algorithm, **Direct Preference Optimization (DPO)**[28], in 2023. The DPO algorithm directly uses human preference data to train the model, eliminating the need to separately construct a reward model and apply complex reinforcement learning algorithms. This method first collects human preference data containing multiple responses and labels the optimal and suboptimal responses. Then, the model is fine-tuned to **increase the probability of selecting the optimal response** while **decreasing the probability of selecting the suboptimal response**. This approach significantly simplifies the human feedback alignment process, improving training efficiency and model stability. Although it may be slightly inferior to RLHF in handling complex human preferences, the computational efficiency advantages of DPO make it widely applicable in multiple fields.

5. Making a Big Splash: ChatGPT and GPT-4 Models

In November 2022, OpenAI launched the chatbot (**ChatGPT** - Chat Generative Pre-trained Transformer) [22]. ChatGPT made a significant impact with its powerful conversational abilities, sparking discussions about whether ChatGPT could pass the ”Turing Test”[3]. Additionally, users can easily access the pre-trained ChatGPT model via OpenAI’s web interface or API without local deployment, marking the emergence of a new service model, LLMaaS (LLM as a Service). However, starting with ChatGPT, the GPT series models became closed-source, preventing us from gaining insight into the technical details of ChatGPT and subsequent models.

Four months later, in March 2023, OpenAI continued to release the **GPT-4** model [23]. Compared to ChatGPT, GPT-4 has further improved in understanding complex contexts, capturing subtle language differences, and generating coherent text, and it can more effectively handle advanced cognitive tasks such as mathematical problems and programming challenges. Additionally, GPT-4 introduced support for multimodal inputs, expanding its potential applications in areas like image description and visual question answering.

One year later, to further enhance model performance and user experience, OpenAI introduced **GPT-4o** in May 2024 [24]. Building on the previous GPT-4, GPT-4o significantly improved response speed and reduced latency while also enhancing multimodal processing capabilities and multilingual support. It has performed exceptionally well in customer support, content creation, and data analysis. The release of GPT-4o marks the maturation of AIGC applications.

The evolution of the GPT series models is an exciting chapter in the history of artificial intelligence. From the debut of GPT-1 to the breakthrough of GPT-4o, the GPT

---

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

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

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

series has achieved revolutionary advancements in just six years. Many ”science fictions” have become reality, and numerous industries are being reshaped. However, with the GPT series moving towards closed-source, users can only utilize its functionalities without participating in the co-creation and improvement of the models.

2.5.3 LLAMA Series Language Models

LLaMA (Large Language Model Meta AI) is a series of large language models developed by Meta AI. The model weights are available to the academic community under a non-commercial license, promoting ”co-creation” and knowledge sharing of large language models. In terms of model architecture, LLaMA draws inspiration from the design philosophy of the GPT series while innovating and optimizing in technical details. The main difference between LLaMA and the GPT series lies in the fact that the GPT series focuses on scaling up the model size and pre-training data simultaneously, whereas LLaMA maintains a relatively stable model size and places greater emphasis on **increasing the scale of pre-training data**. Table 2.5 shows the release dates, parameter counts, and data scales for different versions of the LLaMA models. Currently, Meta AI has released three versions of the LLaMA models. Based on these models, numerous derivative models have been developed. Together, the original LLaMA models and their derivatives form the LLaMA ecosystem. The following sections will introduce the three versions of LLaMA and some of its derivative models.

1. LLaMA1 Model

LLaMA1[40] is the first large language model released by Meta AI in February 2023. Guided by the scaling laws from Chinchilla[15], LLaMA1 practices the concept of ”small model + big data,” aiming to train relatively smaller models with large-scale high-quality

Table 2.5: Parameter and Data Scale Table for LLaMA Series Models.

Model	Release Date	Parameter Count (Billion)	Data Scale
LLAMA-1	2023.02	67 / 130 / 325 / 652	Approximately 5TB
LLAMA-2	2023.07	70 / 130 / 340 / 700	Approximately 7TB
LLAMA-3	2024.04	80 / 700	Approximately 50TB

data. A relatively smaller parameter size enables faster inference speeds, making it better suited for scenarios with limited computational resources.

In terms of pre-training data, LLaMA1's pre-training data includes large-scale web datasets such as Common Crawl[25], the C4 dataset proposed by T5[29], and data from various sources including Github, Wikipedia, Gutenberg, Books3, Arxiv, and StackExchange, totaling approximately 5TB.

In terms of model architecture, LLaMA1 adopts the same network architecture as the GPT series. However, it optimizes the original Transformer word embedding module, attention module, and fully connected feed-forward module. For the word embedding module, to improve the quality of word embeddings, LLaMA1 references the approach of GPTNeo[4] and uses **Rotary Positional Embeddings (RoPE)**[36] instead of the original absolute positional embeddings, thereby enhancing the expressive power of positional embeddings and the model's understanding of sequence order. For the attention module, LLaMA1 references the approach of PaLM[7] and replaces the RELU activation function in Transformers with the **SwiGLU activation function**[34]. Additionally, LLaMA1 applies rotary positional embeddings to the query and key before self-attention operations. For the fully connected feed-forward module, LLaMA1 adopts the **Pre-Norm layer**

---

[25] https://commoncrawl.org

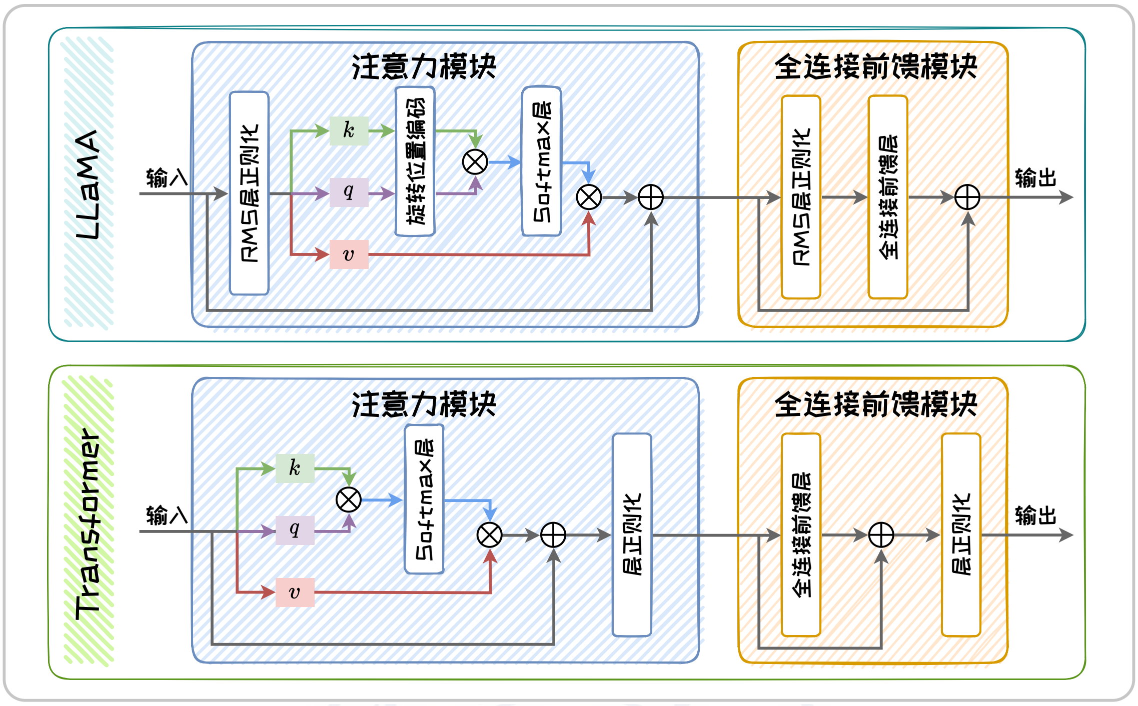


Figure 2.16: Comparison of LLaMA Decoder Block Architecture with Standard Transformer Decoder Architecture.

normalization strategy from GPT-3[5], applying normalization to the inputs of the self-attention and feed-forward networks. Improvements in the attention module and fully connected feed-forward module are illustrated in Figure 2.16.

Based on this framework, LLaMA1 offers four versions of the model. Specific parameters for each version are shown in Table 2.6.

Table 2.6: Detailed Parameters of LLaMA1 Models.

Model Version	Number of Decoder Blocks	Hidden Layer Dimension	Number of Self-Attenti
LLaMA1-7B	32	4096	32
LLaMA1-13B	40	5120	40
LLaMA1-32B	60	6656	52
LLaMA1-65B	80	8192	64

2. LLaMA2 Model

In July 2023, Meta AI released the second-generation model of the LLaMA series, LLaMA2[39]. Adhering to the design philosophy of ”small model + big data,” LLaMA2 further optimized and expanded the training data on the basis of LLaMA1, increasing the scale of the corpus to approximately 7TB, thus covering a richer range of languages and domain resources. Additionally, after the pre-training phase, LLaMA2 adopted the method of reinforcement learning from human feedback to further enhance model performance. First, it used a large-scale and publicly available instruction fine-tuning dataset[8] to perform supervised fine-tuning of the model. Then, LLaMA2 trained an RLHF reward model and updated the model using **Proximal Policy Optimization (PPO)**[33] and **Rejection Sampling** for reinforcement learning.

In terms of model architecture, LLaMA2 inherits the architecture of LLaMA1. LLaMA2 offers four versions of the model, with specific parameters for each version shown in Table 2.7. Among these, LLaMA2-34B and LLaMA2-70B additionally incorporate **Grouped Query Attention (GQA)**[1] to improve computational efficiency. Under the GQA mechanism, keys and values are no longer one-to-one with queries; instead, a group of queries shares the same keys and values, effectively reducing memory usage and the total number of model parameters.

Table 2.7: Detailed Parameters of LLaMA2 Models.

Model Version	Number of Decoder Blocks	Hidden Layer Dimension	Number of Self-Attenti
LLaMA2-7B	32	4096	32
LLaMA2-13B	40	5120	40
LLaMA2-34B	60	6656	52
LLaMA2-70B	80	8192	64

3. LLaMA3 Model

In April 2024, Meta AI further introduced the third-generation model, LLaMA3 [26]. LLaMA3 selected a pre-training corpus of up to 50TB, seven times larger than that of LLaMA2. This corpus not only contains rich code data to enhance the model's logical reasoning capabilities but also includes over 5

In terms of model architecture, LLaMA3 is almost identical to the previous generation, LLaMA2, except for an expansion of the dictionary length by three times during the tokenization stage. This improvement greatly enhances inference efficiency. By reducing the fragmentation of Chinese characters and other language elements into multiple tokens, it effectively lowers the overall token count, thereby improving the coherence and accuracy of language processing. The expanded dictionary also helps reduce the splitting of semantically complete units, allowing the model to more accurately capture word meanings and context when processing text, enhancing the fluency and coherence of generated text. Both the 8 billion parameter and 70 billion parameter versions of LLaMA3 adopt the grouped query attention mechanism. Despite maintaining highly consistent model parameters with the corresponding versions of LLaMA2, LLaMA3 achieves a qualitative leap in performance, fully demonstrating the power of data.

4. Derivative Models of LLaMA

The open-source sharing of the LLaMA model has attracted many researchers to continue developing on its foundation. Researchers either focus on further improving the performance of the LLaMA model, adapting it to vertical domains, or expanding the data modalities supported by the LLaMA model. This collective effort has created a diverse and vibrant research ecosystem for LLaMA. Below is a brief introduction to three mainstream categories of LLaMA derivative models.

---

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

- **Performance Improvement Models**: Some researchers focus on enhancing the performance of the LLaMA model through fine-tuning. For example, Alpaca [27] fine-tunes LLaMA1 using instruction-following sample data generated by GPT-3.5, achieving performance comparable to GPT-3.5 with a smaller model size. Vicuna [28] takes a different approach, fine-tuning the LLaMA1 model with daily conversation data accumulated on the ShareGPT platform, further enhancing its conversational capabilities. The Guanaco[10] model introduces QLoRA technology during the fine-tuning of LLaMA1, significantly reducing the time cost and improving the efficiency of fine-tuning.
- **Vertical Task Models**: While LLaMA performs well in general tasks, its potential in specific domains remains to be explored. Therefore, many researchers fine-tune LLaMA for vertical domains to improve its performance in these areas. For example, CodeLLaMA[30] fine-tunes LLaMA2 using a large amount of public code data, making it better suited for automated code generation, error detection, and code optimization. LawGPT[24] fine-tunes the LLaMA1 model with 300,000 legal QA pairs, significantly enhancing its ability to process legal content. GOAT[21] fine-tunes the LLaMA1 model with a math problem database generated by Python scripts, improving its accuracy in solving various math problems. Cornucopia [29] fine-tunes the model with financial QA data, enhancing its performance in financial QA.
- **Multimodal Task Models**: By integrating visual modality encoders and cross-modal alignment components, researchers have extended the LLaMA model to mul-

---

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

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

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

timodal tasks. For example, LLaVA[19] builds on Vicuna by using CLIP to extract image features and a linear projection layer to align images and text. MiniGPT4[45] also builds on Vicuna, using VIT-G/14 and Q-Former as image encoders and a linear projection layer to align images and text, demonstrating capabilities in multimodal task processing.

These derivative models not only enrich the application scenarios of the LLaMA model but also provide new directions and possibilities for research in the field of natural language processing. With its open-source and collaborative approach, the LLaMA series attracts global researchers to participate in co-creation. We have every reason to believe that the LLaMA series and its derivative models will shine like stars, illuminating the path forward for large language models.

Table 2.8: Parameter and Data Scale Table for GPT Series and LLaMA Series Models.

Model	Release Date	Parameter Count (Billion)	Data Scale
GPT-1	2018.06	1.17	Approximately 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	Unknown	Unknown
GPT-4	2023.03	Unknown	Unknown
GPT-4o	2024.05	Unknown	Unknown
LLAMA-1	2023.02	67 / 130 / 325 / 652	Approximately 5TB
LLAMA-2	2023.07	70 / 130 / 340 / 700	Approximately 7TB
LLAMA-3	2024.04	80 / 700	Approximately 50TB

Based on the decoder-only architecture, large language models have led a new wave of generative artificial intelligence with their outstanding generation capabilities. Table 2.8 shows the specific parameters of different versions of the GPT series and LLaMA series models. It is evident that models based on the decoder-only architecture have seen rapid growth in both parameter count and pre-training data scale. With further enrichment of computational resources and data resources, large language models based on the decoder-only architecture are bound to shine even more brightly.

2.6 Non-Transformer Architectures

The Transformer architecture is currently the mainstream model architecture for large language models, offering advantages such as flexible construction, ease of parallelization, and scalability. However, the Transformer is not without flaws. Its parallel input mechanism leads to a quadratic increase in model size with the length of the input sequence, causing computational bottlenecks when processing long sequences. To improve computational efficiency and performance and address the bottleneck of Transformer in long-sequence processing, Recurrent Neural Network (RNN) based language models can be chosen. RNNs consider only previous hidden states and the current input when generating output, theoretically capable of processing sequences of infinite length. However, traditional RNN models (such as GRU, LSTM, etc.) may struggle to capture long-term dependencies when processing long sequences and face issues of gradient vanishing or explosion. To overcome these problems, researchers have recently proposed two types of modern RNN variants: **State Space Models** (SSM) and **Test-Time Training** (TTT). Both paradigms can achieve linear time complexity with respect to sequence length and avoid the issues present in traditional RNNs. This section will briefly introduce these two paradigms and their representative models.

2.6.1 State Space Models (SSM)

The **State Space Model** (SSM) [13] paradigm can effectively handle long-range dependency (LRD) problems in long texts and reduce the computational and memory costs of language models. This subsection will first introduce the SSM paradigm and then present two representative models based on the SSM paradigm: RWKV and Mamba.

1. SSM

The concept of SSM originates from dynamical systems in control theory. It captures the continuous evolution of system state over time using a set of state variables, which naturally suits describing dependencies over long periods. Additionally, SSM has recursive and convolutional discrete representations, allowing efficient processing of sequential data through recursive updates during inference and capturing global dependencies via convolutional operations during training.

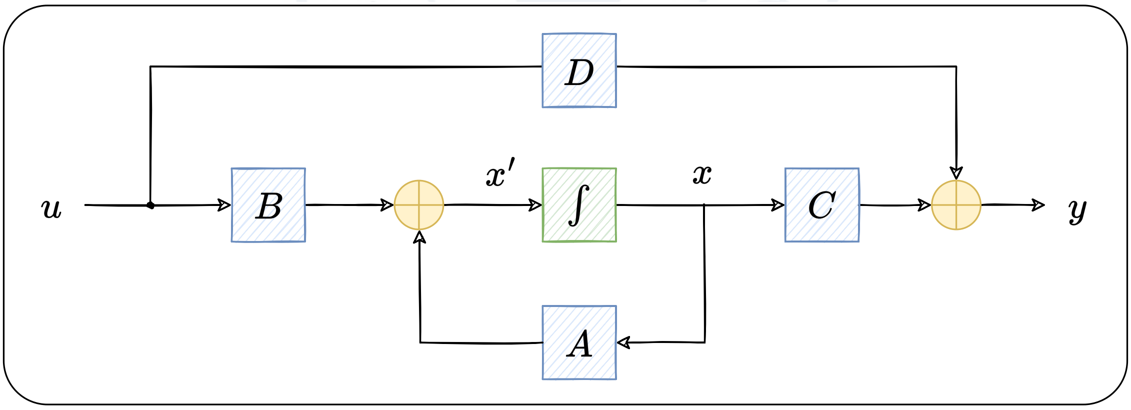


Figure 2.17: SSM Paradigm.

As shown in Figure 2.17, SSM constructs on the basis of three variables evolving over time t and four learnable matrices. The three variables are: $x(t) \in \mathbb{C}^n$ representing n state variables, $u(t) \in \mathbb{C}^m$ representing m state inputs, and $y(t) \in \mathbb{C}^p$ representing p outputs. The four matrices are: state matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$, control matrix $\mathbf{B} \in \mathbb{C}^{n\times m}$, output matrix $\mathbf{C} \in \mathbb{C}^{p\times n}$, and command matrix $\mathbf{D} \in \mathbb{C}^{p\times m}$. The system equations of SSM are:

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

where $x'(t) = \mathbf{A}x(t) + \mathbf{B}u(t)$ is the state equation, describing how the system state evolves based on the input and the previous state, calculating the derivative of the state with respect to time $x'(t)$. To obtain the state $x(t)$, an integration operation is required. $y(t) = \mathbf{C}x(t) + \mathbf{D}u(t)$ is the output equation, describing how the system state is transformed into output, where $x(t)$ is the value updated by the state equation and integrated. In deep learning, the term $\mathbf{D}u(t)$ represents residual connections, which can be ignored.

These equations represent the continuous form of the SSM system equations, suitable for processing continuous data (e.g., audio signals, time series), but are very slow for both training and inference. To improve the efficiency of SSM processing, the equations need to be discretized. **Discretization** is the most critical step in SSM, converting the system equations from continuous to recursive and convolutional forms to enhance the efficiency of the entire SSM architecture.

When discretizing the continuous form of the SSM system equations, the trapezoidal rule can be used

to replace the integral operation in the continuous form. The principle is to approximate the area under the curve of a function defined over a specific interval as a trapezoid and calculate its area using the trapezoidal area formula. Thus, the system equations in recursive form after discretization are:

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

In this equation, the state equation calculates the current state based on the state from

1897 ZHEJIANG UNIVERSITY

the previous step and the current input, embodying the recursive idea. Here, $\bar{\mathbf{A}}, \bar{\mathbf{B}}, \bar{\mathbf{C}}$ are the matrices in discrete form, related to the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ in continuous form as follows: $\bar{\mathbf{A}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\left(\mathbf{I} + \frac{\Delta}{2}\mathbf{A}\right)$, $\bar{\mathbf{B}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\Delta\mathbf{B}$, $\bar{\mathbf{C}} = \mathbf{C}$, where $\Delta = t_{n+1} - t_n$.

The recursive form of SSM resembles RNN, inheriting both the strengths and weaknesses of RNNs. It is suitable for processing sequential data, enabling efficient inference with linear complexity relative to sequence length, but cannot be trained in parallel and faces issues of gradient vanishing or explosion with long sequences. Iterating the recursive form of the system equations yields the convolutional form. Omitting the derivation process, the results for x_k and y_k after iteration are:

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

It can be observed that after expanding the recursive form of the system equations iteratively, the output y_k is the convolution result of the state input u_k, with the convolution kernel being:

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

Therefore, the convolutional form of the SSM system equations is:

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

where the convolution kernel is determined by the matrix parameters in the SSM, which remain constant throughout the processing of the sequence, referred to as **time-invariance**. Time-invariance allows SSM to consistently handle data at different time steps and enables efficient parallel training. However, due to the fixed context length, the convolutional form of SSM has long latency and high computational cost when performing autoregressive tasks. Considering the advantages and disadvantages of the recursive and convolutional forms of SSM after discretization, one can choose to **use the convolutional form for training and the recursive form for inference**.

In summary, the system equations of the SSM architecture come in three forms: continuous form, recursively discretized form, and convolutedly discretized form, applicable to tasks such as text, vision, audio, and time series. When applying, the appropriate representation should be chosen based on specific circumstances. The advantage of SSM lies in its ability to handle very long sequences, maintaining fast speed even with fewer parameters than other models when processing long sequences.

Currently, the main differences among various existing SSM architectures lie in the discretization methods of the basic SSM equations or the definition of the $\mathbf{A}$ matrix. For example, S4 [14] (Structured State Space Model) is a variant of SSM whose key innovation is the use of the HiPPO matrix to initialize the $\mathbf{A}$ matrix, demonstrating excellent performance in processing long-sequence data. Moreover, RWKV and Mamba are two classic architectures based on the SSM paradigm. Below, we will introduce these two architectures.

2. RWKV

RWKV (Receptance Weighted Key Value) [26] is an innovative architecture based on the SSM paradigm. Its core mechanism, WKV, can be viewed as the ratio of two SSMS. RWKV combines the advantages of RNNs and Transformers, retaining efficiency during the inference stage and achieving parallelization during the training stage. (Note: The discussion here is about RWKV-v4.)

The core modules of the RWKV model consist of two parts: the **Temporal Mixing Module** and the **Channel Mixing Module**. The Temporal Mixing Module primarily deals with the relationships between different time steps in the sequence, while the Channel Mixing Module focuses on interactions between different feature channels [30] within the same time step. The design of the Temporal Mixing Module and the Channel Mixing Module is based on four fundamental elements: **Receptance Vector R**, **Key Vector K**, **Value Vector V**, and **Weight W**. According to Figure 2.18, we will introduce the computation flow and roles of these elements in the two modules.

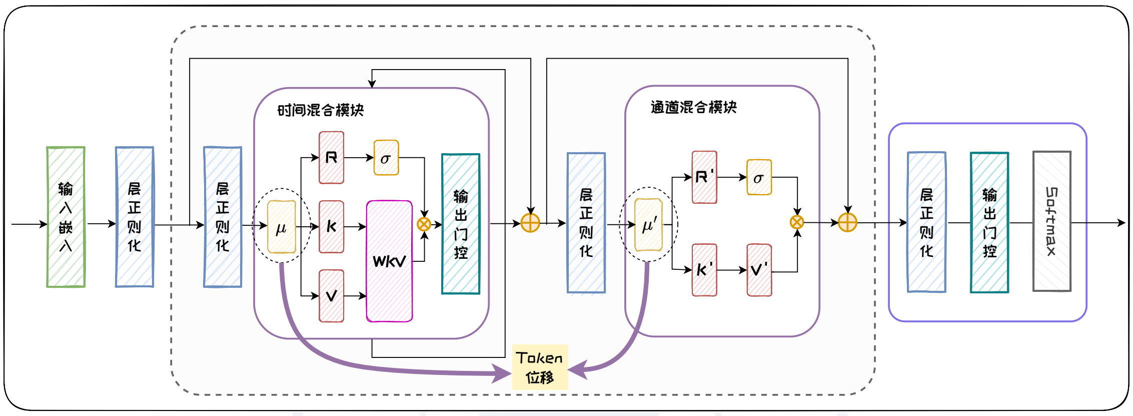


Figure 2.18: RWKV Architecture.

ZHEJIANG UNIVERSITY

The common operation in both the Temporal Mixing Module and the Channel Mixing Module is token shifting, which achieves sensitivity to temporal changes in the sequence by performing linear interpolation between the inputs of the current and the previous time steps. In the Temporal Mixing Module, the Receptance Vector R is responsible for receiving and integrating information from the history of the sequence, the Weight W represents position weight decay, and the Key Vector K and Value Vector V are similar to keys and values in traditional attention mechanisms, used for matching and carrying information, respectively. The Temporal Mixing Module first performs a linear combination of the in-

---

[30] Here, **channels** refer to different feature dimensions at the same time step within a neural network. For example, in NLP tasks, the input at each time step might be a word in a piece of text, and each word is converted into a vector through an embedding layer. Each element of this vector represents a feature channel, and the vector dimension is the number of channels.

puts from the current and the previous time steps, projecting them linearly to obtain the R, K, V vectors; subsequently, it ensures that the weights of each channel gradually decay over time through the WKV mechanism; finally, it integrates the past information represented by $\sigma(\mathrm{R})$ and the current information represented by the WKV vector through an output gate, passing it to the Channel Mixing Module.

The **WKV mechanism** is the core part of the Temporal Mixing Module in RWKV. In RWKV, the Weight W is a time-decay vector associated with channels, which can be expressed as: $w_{t,i} = -(t-i)w$, where i denotes a certain time step traced back from the current time step t, and w is a non-negative vector with a length equal to the number of channels. By doing so, the model can capture dependencies between different time steps in time series data and achieve the effect of gradually decaying the weights of each channel backward in time. The key to the WKV mechanism lies in updating the hidden state using linear time-invariant recursion, which can be regarded as the ratio of two SSMs. The specific formula is:

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

where u is a vector that focuses solely on the current token. Since both the numerator and denominator exhibit processes of state update and output, similar to the SSM idea, this formula can be considered as the ratio of two SSMs. Through the Weight W and WKV mechanism, the module can effectively handle long-time series data and reduce the problem of gradient vanishing.

In the Channel Mixing Module, the roles of R', K', and V' are similar to those in the Temporal Mixing Module, with R' and K' also obtained from linear projections of the input, while the update of V' additionally depends on K'. Subsequently, $\sigma(\mathrm{R}')$ and V' are integrated to facilitate information interaction and fusion across different channels.

Moreover, the RWKV architecture adopts a time-dependent Softmax operation to improve numerical stability and gradient propagation efficiency, as well as layer normalization to stabilize gradients and prevent vanishing and exploding gradients. To further boost performance, RWKV also employs optimized measures such as custom CUDA kernels, small-value initialization of embeddings, and custom initialization.

RWKV demonstrates notable performance in terms of model size, computational efficiency, and model performance. In terms of **model size**, the RWKV model scales up to 14B parameters, making it the first non-Transformer architecture scalable to hundreds of billions of parameters. Regarding **computational efficiency**, RWKV allows the model to be represented as either a Transformer or an RNN, thus enabling parallel computation during training and maintaining constant computational and memory complexity during inference. In terms of **model performance**, RWKV shows comparable performance to similarly sized Transformers on NLP tasks and ranks second only to S4 in long-context benchmarks.

Through its innovative linear attention mechanism, RWKV successfully combines the advantages of Transformers and RNNs, achieving significant progress in model size and performance. However, RWKV still faces some limitations in handling long-distance dependencies and complex tasks. To address these issues and further enhance long-sequence modeling capabilities, researchers have proposed the Mamba architecture.

3. Mamba

Time-invariance allows SSM to consistently handle data at different time steps and perform efficient parallel training, but it also weakens its capability to process information-dense data like text. To compensate for this deficiency, Mamba [12] introduces a **Selection Mechanism** and a **Hardware-aware Algorithm** based on the SSM architecture. The

ZHEJIANG UNIVERSITY 1897

former enables the model to perform content-based reasoning, while the latter realizes efficient computation on GPUs, ensuring **fast training and inference**, **high-quality data generation**, and **long-sequence processing capabilities**.

The **Selection Mechanism** in Mamba dynamically adjusts model parameters to select the information to focus on, making the model parameters vary according to the input data. Specifically, Mamba transforms the parameters $\mathbf{B}$, $\mathbf{C}$, and Δ in the discretized SSM into the following functions: $s_{\mathbf{B}}(x) = \text{Linear}_N(x)$, $s_{\mathbf{C}}(x) = \text{Linear}_N(x)$, $s_{\Delta}(x) = \text{Broadcast}_D(\text{Linear}_1(x))$, and uses a nonlinear activation function $\tau_{\Delta} = \text{softplus}$ to regulate the parameter Δ. Here, Linear_d is a parametrized projection onto the feature dimension d, and the choice of s_{Δ} and τ_{Δ} functions is associated with the gating mechanism of RNNs [38]. Additionally, Mamba adjusts the tensor shapes to give the model parameters a temporal dimension, meaning that the model parameter matrices have different values at each time step, transitioning from time-invariant to time-variant.

The Selection Mechanism turns the model parameters into functions of the input and endows them with a temporal dimension, thus removing the translation invariance and linear time-invariance of convolutional operations, affecting its efficiency. To enable efficient computation of selective SSM models on GPUs, Mamba proposes a **Hardware-aware Algorithm** comprising kernel fusion, parallel scanning, and recomputation. **Kernel Fusion** increases speed by reducing memory I/O operations. **Parallel Scanning** improves efficiency using parallel algorithms. **Recomputation** recalculates intermediate states during backpropagation to reduce memory requirements.

In practical implementation, SSM parameters are loaded from slower high-bandwidth memory (HBM) into faster static random-access memory (SRAM) for computation, and the final output is written back to HBM. This maintains efficient computation while re-

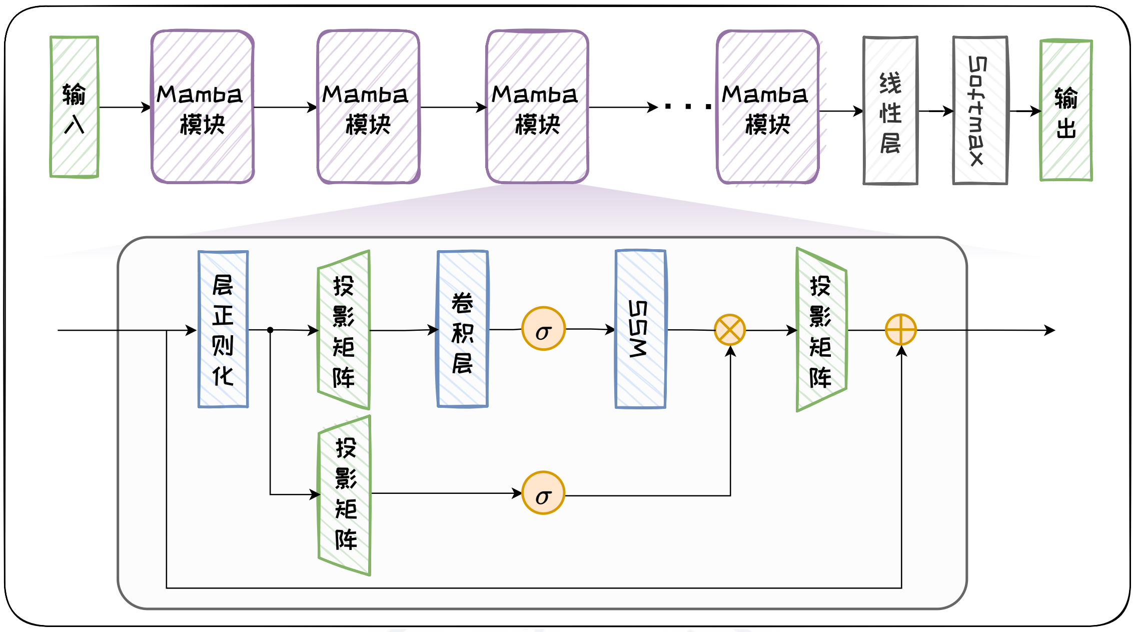


Figure 2.19: Mamba Architecture.

ducing memory usage, aligning the model's memory demand with optimized Transformer implementations such as FlashAttention.

By combining SSM modules with selection mechanisms and the feedforward layers of Transformers, Mamba forms a simple and homogeneous architectural design. As shown in Figure 2.19, the Mamba architecture is a recursive model composed of entirely identical Mamba modules, each inserting a convolutional layer and an SSM module with a selection mechanism into the feedforward layer, where the activation function σ is selected as SiLU/Swish.

By introducing the selection mechanism and hardware-aware algorithm, Mamba demonstrates outstanding performance and efficiency in practical applications, including: (1) **Fast Training and Inference**: During training, the computational and memory requirements grow linearly with the sequence length, whereas during inference, each step requires only constant time, eliminating the need to retain all previous information. Through the hardware-aware algorithm, Mamba not only achieves theoretical linear scalability with sequence length but also boosts inference throughput by 5 times compared to similarly scaled Transformers on A100 GPUs. (2) **High-Quality Data Generation**: Mamba performs excellently across multiple modalities and settings, including language modeling, genomics, audio, and synthetic tasks. In language modeling, the Mamba-3B model outperforms Transformer models with twice the number of parameters in both pre-training and subsequent evaluations. (3) **Long Sequence Processing Capability**: Mamba can handle sequences up to a million tokens long, showcasing its superiority in processing long contexts.

Although Mamba has certain limitations regarding hardware dependency and model complexity, it significantly enhances the efficiency of processing long sequences and information-dense data through the introduction of the selection mechanism and hardware-aware algorithm, demonstrating great potential for application in various fields. Mamba's outstanding performance across multiple applications makes it an ideal general-purpose foundation model.

ZHEJIANG UNIVERSITY 1897

2.6.2 Training-Time Updates for TTT

When dealing with long-context sequences, the above architectures based on the SSM paradigm (such as RWKV and Mamba) successfully reduce computational complexity to a linear level by compressing context information into fixed-length hidden states, effectively extending the model's capability to handle long contexts. However, as the context length continues to grow, models based on the SSM paradigm may prematurely reach performance saturation. For example, Mamba's perplexity barely decreases beyond a context length of 16k [37]. This phenomenon may be due to the fixed-length hidden state limiting the model's expressive power and potentially leading to the forgetting of crucial informa-

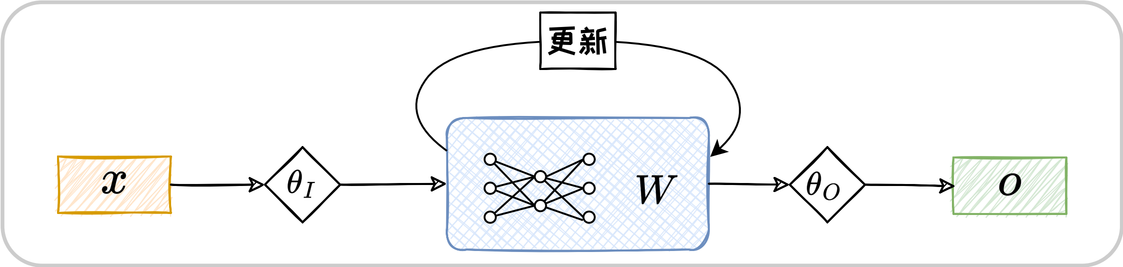


Figure 2.20: Inference Process under the TTT Paradigm.

tion during compression.

To address this limitation, the Test-Time Training (TTT) paradigm [37] provides an effective solution. TTT utilizes the model's own parameters to store hidden states and remember context; during each inference step, it performs gradient updates on the model parameters to continuously cycle in the context, as shown in Figure 2.20. This process differs from the traditional machine learning paradigm where the model remains static during the inference phase after training is complete. Instead, TTT trains and infers simultaneously on each test data point during the inference phase. To implement this test-time training mechanism, TTT has unique designs for both the pre-training and inference phases.

During the pre-training phase of the TTT paradigm, the training process includes both internal and external loops. The external loop follows the traditional next-token prediction task, optimizing the global weight parameters of the model through an autoregressive approach. The internal loop optimizes the hidden states using a self-supervised method. Specifically, the model needs to dynamically update the hidden state at each time step to continuously adapt to new input data. This dynamic update mechanism is akin to an independent machine learning model training and optimizing the input at each time step. Given the current time step input x_t and the hidden state W_{t-1} corresponding to the previous historical context $x_1, x_2, \ldots, x_{t-1}$, the model computes the reconstruction loss at the current time step:

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

where θ_K and θ_V are parameters learned through the external loop. Then, the model performs gradient descent with a learning rate η using this loss to update the hidden state:

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

Finally, the model generates the output based on the updated hidden state and the current input:

$$z_t = f(x_t; W_t). \tag{2.13}$$

During the inference phase, there is no need to execute the external loop task. Therefore, the model only performs the internal loop to update the hidden state, allowing it to better adapt to new data distributions and thereby enhancing prediction performance.

Compared to Transformers, models based on the TTT paradigm have linear time complexity, which is crucial for processing long sequence data. Compared to SSM-based architectures like RWKV and Mamba, TTT stores context information using model parameters, enabling more effective capture of semantic links and structural information in ultra-long contexts. Therefore, TTT exhibits superior performance in long-context modeling tasks, especially in scenarios requiring the processing of ultra-long contexts. In the future, the TTT paradigm is expected to play a significant role in ultra-long sequence processing tasks.

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.

3 Prompt Engineering

As the scale of model training data and the number of parameters continue to grow, large language models have surpassed the generalization bottleneck and demonstrated powerful instruction-following capabilities. The enhancement of generalization ability enables models to handle and understand various unknown tasks, while the improvement in instruction-following capabilities ensures that the models can accurately respond to human instructions. The combination of these two abilities allows us to guide models to adapt to various downstream tasks through carefully crafted instruction inputs, i.e., Prompts, thereby avoiding the high computational costs associated with traditional fine-tuning methods. Prompt engineering, as a discipline dedicated to crafting these effective instructions, serves as a bridge between models and task requirements. It not only requires a deep understanding of the models but also a precise grasp of task objectives. Through prompt engineering, we can maximize the potential of large language models, enabling them to deliver outstanding performance across diverse application scenarios. This chapter will delve into the concepts, methods, and roles of prompt engineering, introducing techniques such as contextual learning and chain-of-thought reasoning, as well as related applications of prompt engineering.

* This book is continuously updated, and the GitHub link is: https://github.com/ZJU-LLMs/Foundations-of-LLMs.

3.1 Introduction to Prompt Engineering

Traditional natural language processing (NLP) research follows the ”pre-training-fine-tuning-prediction” paradigm, which involves pre-training on a large-scale corpus, fine-tuning on downstream tasks, and finally performing predictions with the fine-tuned model. However, with the significant advancements in the scale and capabilities of language models, a new paradigm—”pre-training-prompt-prediction”—has emerged. In this paradigm, large models adapt directly to downstream tasks through carefully designed Prompts on top of pre-trained models, without the need for cumbersome fine-tuning, as illustrated in Figure 3.1. In this process, the design of Prompts has a profound impact on model performance. The techniques dedicated to crafting these Prompts are collectively referred to as **Prompt Engineering**. In this section, we will provide an in-depth introduction to the definition and related concepts of Prompt Engineering, exploring its significance and applications in the field of NLP.

ZHEJIANG UNIVERSITY 1897

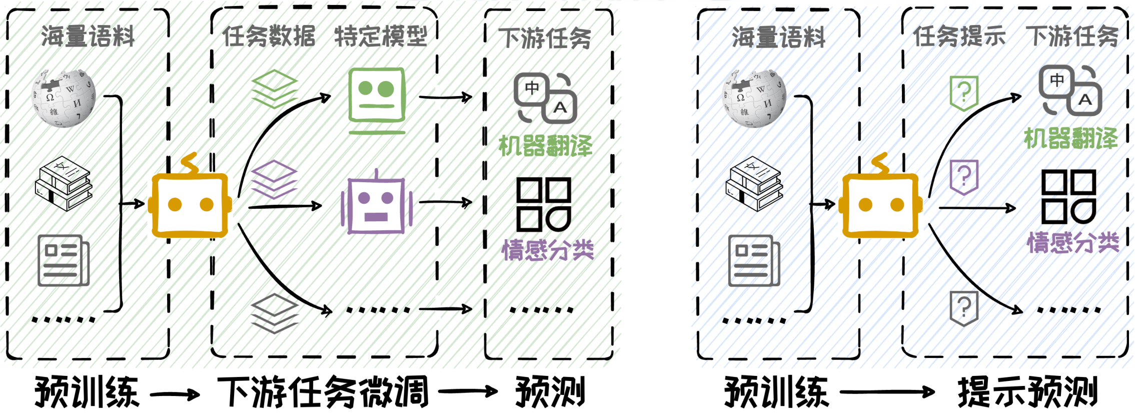
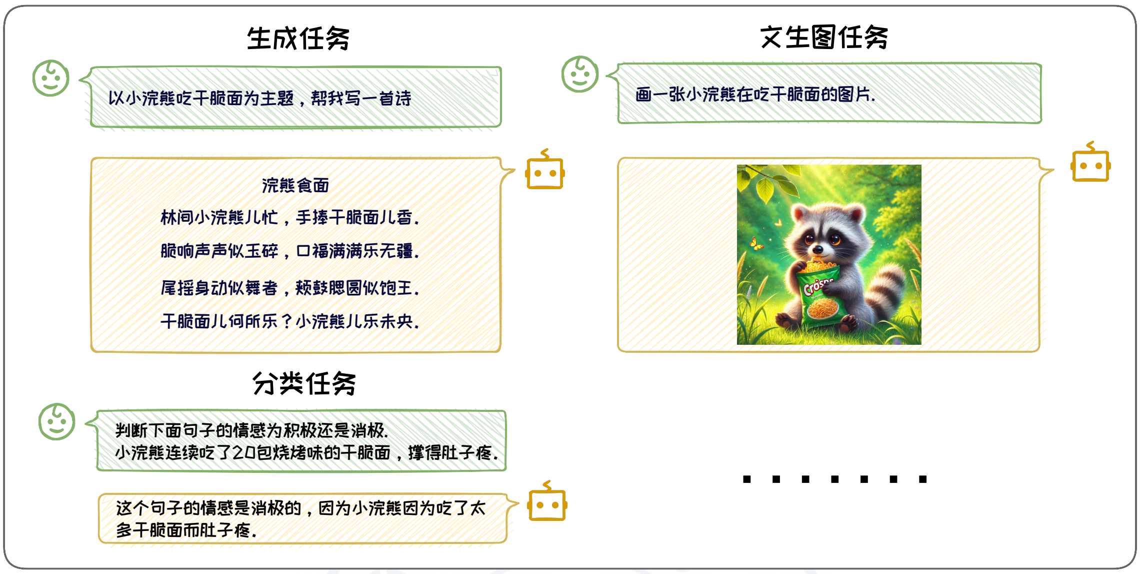


Figure 3.1: Comparison between the ”pre-training-fine-tuning-prediction” paradigm and the ”pre-training-prompt-prediction” paradigm.

3.1.1 Definition of Prompt



以小浣熊吃干脆面为主题，帮我写一首诗

浣熊食面
林间小浣熊儿忙，手捧干脆面儿香.
脆响声声似玉碎，口福满满乐无疆.
尾摇身动似舞者，颊鼓腮圆似饱王.
干脆面儿何所乐？小浣熊儿乐未央.

画一张小浣熊在吃干脆面的图片.

判断下面句子的情感为积极还是消极.
小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼.

这个句子的情感是消极的，因为小浣熊因为吃了太多干脆面而肚子疼.

Figure 3.2: Examples of common Prompts.

A Prompt refers to an input instruction used to guide generative AI models to perform specific tasks [1]. These instructions are typically presented in the form of natural language text. The core purpose of a Prompt is to clearly describe the task that the model should perform, thereby guiding the model to generate specific outputs such as text, images, or audio. As shown in Figure 3.2, carefully designed Prompts enable models to achieve diverse functionalities. For instance, by providing explicit sentiment classification instructions, the model can accurately perform sentiment analysis on text; by issuing creative composition instructions on specific topics, the model can generate imaginative poetry. Additionally, in multimodal model applications, Prompts can include visual descriptions to guide the model in generating corresponding visual outputs.

The applications of Prompts are wide-ranging and are not limited to text-to-text tasks. Since this book primarily focuses on language models, this chapter will concentrate on

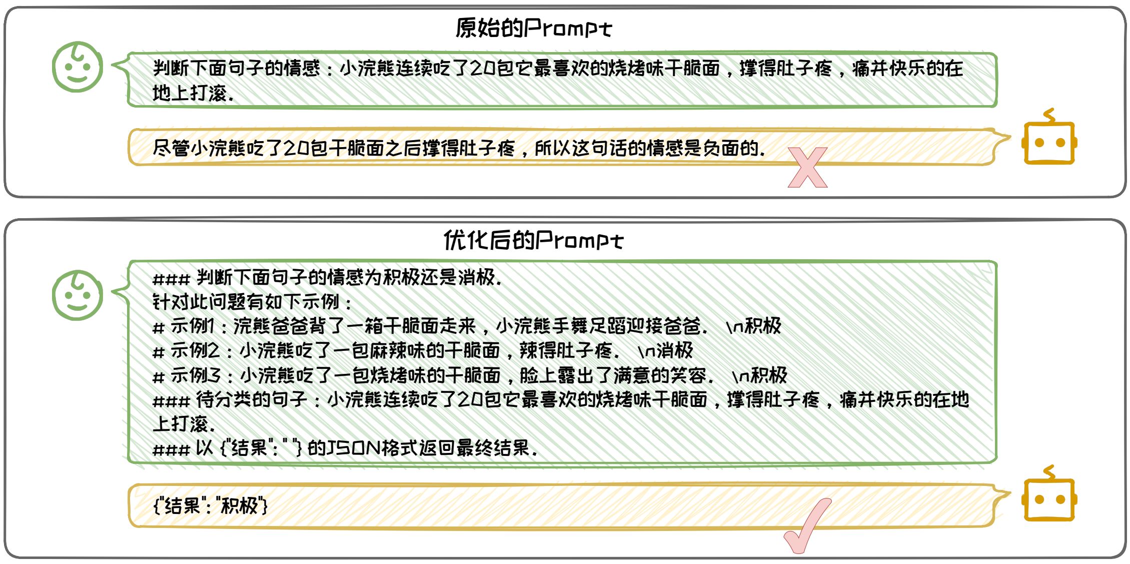
---

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

text generation models and explore how carefully designed Prompts can guide models to produce text outputs that meet specific task requirements.

3.1.2 Definition of Prompt Engineering

Prompt Engineering refers to the process of designing and optimizing Prompts for interaction with generative AI models [2]. The core idea of this technique is to frame new tasks as forms familiar to the model from its pre-training phase by constructing appropriate Prompts. This leverages the model's inherent generalization ability to perform new tasks without additional task-specific training. The success of Prompt Engineering depends on a deep understanding of the pre-trained model and precise knowledge of task requirements. By constructing suitable Prompts for large language models, they can assist in completing various tasks [47].



判断下面句子的情感：小浣熊连续吃了20包它最喜欢的烧烤味干脆面，撑得肚子疼，痛并快乐的在地上打滚.

尽管小浣熊吃了20包干脆面之后撑得肚子疼，所以这句话的情感是负面的.

### 判断下面句子的情感为积极还是消极.
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
# 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
# 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
### 待分类的句子：小浣熊连续吃了20包它最喜欢的烧烤味干脆面，撑得肚子疼，痛并快乐的在地上打滚.
### 以 {"结果":" "} 的JSON格式返回最终结果.

{"结果":"积极"}

Figure 3.3: Comparison of results before and after applying Prompt Engineering techniques.

---

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

As shown in Figure 3.3, the original Prompt is rewritten into a more comprehensive and standardized form through the optimization of Prompt Engineering. The optimized Prompt significantly improves the quality of the model' s generated responses. Therefore, constructing high-quality and comprehensive Prompts during interaction with large language models is crucial, as it directly determines whether valuable outputs can be obtained. A well-designed Prompt typically consists of four essential elements: task description, context, query, and output format:

- **Task Description** —Clearly defines the specific task requirements for the model. The task description should be concise, direct, and as detailed as possible to describe the task the model is expected to complete.
- **Context** —Provides the model with task-related background information to enhance its understanding of the task and guide its problem-solving approach. The context may include specific knowledge prerequisites, the background of the target audience, examples of related tasks, or any information that helps the model better understand the task.
- **Query** —Describes the user' s specific question or the information to be processed by the model. This part should directly address the user's inquiry or task, giving the model a clear starting point. The query can be posed as an explicit question or an implicit statement that reflects the user' s underlying intent.
- **Output Format** —Specifies the desired format of the model's response. This includes the structure of the output and any specific details such as brevity or level of detail. For instance, the model can be instructed to present results in JSON format.

The four basic elements of a Prompt—task description, context, query, and output format—have a significant impact on the effectiveness of the outputs generated by large language models. The careful design and combination of these elements form the core of Prompt Engineering. Building on this foundation, Prompt Engineering incorporates various techniques and methodologies, such as In-Context Learning and Chain of Thought. The combined use of these techniques can substantially enhance the quality of Prompts, thereby effectively guiding the model to produce outputs that better meet specific task requirements. The details of In-Context Learning are discussed in Section 3.2, Chain of Thought is covered in Section 3.3, and the usage techniques for Prompts are explored in detail in Section 3.4.

However, as the content and complexity of Prompts increase, the length of the Prompt input to the model inevitably grows. This leads to slower inference speeds and higher inference costs. Therefore, while striving for better model performance, it is critical to control and optimize the length of Prompts. The challenge is to compress the Prompt input length without compromising the model's performance. To address this issue, LLMLingua [11] introduced an innovative coarse-to-fine Prompt compression method. This method compresses Prompt content to one-twentieth of its original size without sacrificing semantic integrity, with minimal performance loss. Additionally, with the rise of Retrieval-Augmented Generation (RAG) techniques, the amount of context information the model needs to process has increased significantly. FIT-RAG [22] successfully reduces the retrieved context length to approximately 50% of the original while maintaining stable performance, providing an effective solution for handling large-scale contextual information.

3.1.3 Tokenization and Vectorization of Prompts

Once an appropriate Prompt is constructed, it is input into the large language model to obtain satisfactory generation results. However, language models cannot directly understand raw text. Before the Prompt is processed by the model, it must be broken down into a sequence of Tokens. A Token is defined as the **smallest semantic unit** that represents a specific word or part of a word, and each Token is uniquely identified by a Token ID. The process of converting text into Tokens is known as **Tokenization**, as illustrated in Figure 3.4. For example, the sentence ” 小浣熊吃干脆面” (”Little raccoon eats crispy noodles”) is tokenized into a sequence of Tokens, each associated with a corresponding Token ID.

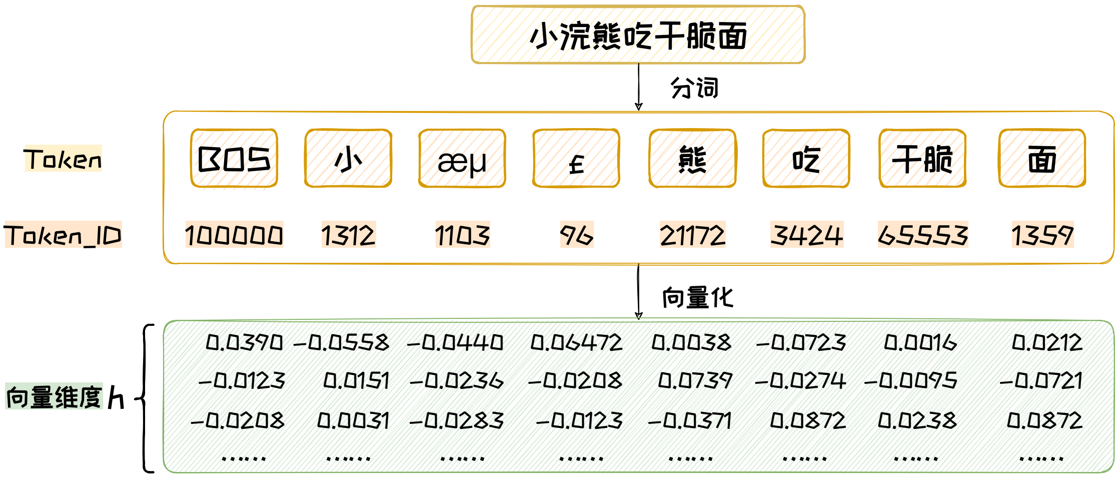


Figure 3.4: The process of tokenization and embedding, illustrated using the tokenizer of the DeepSeek-V2-Chat model [6].

It is worth noting that a sentence may have multiple tokenization methods. For example, the above sentence could be tokenized as “小, 浣熊, 吃干, 脆面” (”Little, raccoon, eats-crispy, noodles”). However, this tokenization approach might lead to semantic confusion or ambiguity. Thus, the tokenization process is inherently challenging and requires careful design. To achieve effective tokenization, the first step is to construct a **vocabulary** containing all the Tokens recognizable by the large language model. Sentences are then tokenized based on this vocabulary.

When constructing the vocabulary of a large language model, the tokenizer relies on tokenization algorithms such as BBPE [34], BPE [8], and WordPiece [29]. These algorithms segment Tokens by analyzing corpus information such as **word frequency**. This section will use the BBPE (Byte-Level Byte Pair Encoding) algorithm as an example to illustrate the tokenization process. The BBPE algorithm consists of the following four steps:

- **1. Initialize Vocabulary**: First, split all characters into bytes according to their underlying encoding and use these single-byte encodings as the initial Tokens of the vocabulary.
- **2. Count Token Frequencies**: Next, count the occurrence frequency of all Token pairs (i.e., adjacent Token combinations) in the vocabulary. At the initial stage, Token pairs refer to combinations of adjacent bytes.
- **3. Merge High-Frequency Token Pairs**: Then, select the most frequent Token pair, merge it into a new Token, and add it to the vocabulary.
- **4. Iterative Merging**: Repeat steps 2 and 3 iteratively until the vocabulary reaches a preset size or a specified number of merges.

The granularity of tokenization can be determined by setting the number of iterations or other parameters. Different granularity levels result in vocabularies of varying sizes. If the **vocabulary is too small**, such as containing only 256 Tokens to represent single bytes, it can encode all GBK characters and English text. However, it makes it difficult for the model to distinguish morphologically similar but semantically distinct words, limiting the model’s ability to represent rich semantics and significantly increasing the sequence length of generated text. Conversely, if the **vocabulary is too large**, it can cover more rare words but may result in insufficient learning of these words and fail to effectively capture relationships between different forms of the same word. Therefore, constructing a vocabulary requires a balance between covering a wide range of words and maintaining semantic precision to ensure the language model can learn a rich vocabulary and accurately understand and generate complex text.

As shown in Figure 3.4, to help the model better understand word meanings and reduce the number of Tokens required for common words, the vocabulary includes **high-frequency** words or phrases from the corpus as independent Tokens. For instance, the word “干脆” (”crispy”) is represented as a single Token in the vocabulary. To optimize the Token space and reduce the vocabulary size, some special Tokens are included in the vocabulary. These Tokens can represent semantics independently or, through pairwise combinations, represent **low-frequency** rare characters in the corpus. For example, the character “浣” (”raccoon”) is represented using two Tokens, “æµ” and “£.” This approach allows the vocabulary to encompass common high-frequency words while flexibly expressing rare characters through Token combinations. Thus, it is evident that vocabulary construction is influenced to some extent by **prior knowledge**, derived from the accumulation and curation of human language corpora.

Each large language model has its own tokenizer, which maintains a vocabulary and can tokenize text. The quality of the tokenizer directly affects the performance of the model. An excellent tokenizer not only significantly enhances the model’s ability to understand text but also improves processing speed and reduces computational resource consumption. A good tokenizer should have the following characteristics: first, it can accurately identify key words and phrases in the text, thereby helping the model better capture semantic information; second, the efficiency of the tokenizer directly impacts the training and inference speed of the model, and an efficient tokenizer can achieve optimization and compression of text tokens, thus significantly reducing the time required for the model to process data.

Table 3.1: Model Tokenizer Comparison Table

Model	**Vocabulary Size**	Chinese Tokenization Efficiency (Characters / Token)	English Tokenization Efficiency (Words / 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

In commonly used open-source models, different models adopt different tokenizers, each with its own characteristics and performance. Their quality is influenced by various factors, including the size of the vocabulary and the efficiency of tokenization. Table 3.1 provides a comparative analysis of tokenizers in common open-source large language models, where the Chinese corpus is excerpted from Zhu Ziqing's Essays [3], and the English corpus comes from Guy de Maupassant's short story ”The Diamond Necklace” [4]. We can observe that Chinese open-source large language models like DeepSeek [6] and

---

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

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

Qwen [41] have optimized Chinese tokenization, with an average of 1.3 characters per Token (each character requiring only 0.7 Tokens), and some common words and idioms can even be represented by a single Token. In contrast, models primarily trained on English, such as GPT-4 and the LLaMA series, have weaker support for Chinese and lower tokenization efficiency. In English, due to suffixes like ”ly” and ”ist”, a single English word typically requires one or more Tokens. Each Token carries more semantic content, so the model needs to output fewer Tokens to express the same text, significantly improving inference efficiency.

Through this comparison, we can clearly see the efficiency of different model tokenizers in handling different languages, which is important guidance for selecting appropriate models and optimizing model performance.

After tokenization, these Tokens are processed through the model’s embedding matrix (Embedding Matrix) to convert them into fixed-size **feature vectors**. These vector sequences are directly input into the model for understanding and processing. During the model generation phase, the model calculates the probability distribution of each word in the vocabulary based on the input vector sequence. The model selects and outputs the corresponding Tokens from these probability distributions, which are then converted back into the respective text content.

The process described above, which involves breaking down text into Tokens using tokenization technology and converting Tokens into feature vectors to represent texts in high-dimensional space, allows language models to capture the deep semantic structure of texts and effectively handle and learn various linguistic structures, from simple vocabularies to complex sentence patterns and contexts.

3.1.4 Significance of Prompt Engineering

Prompt engineering provides an efficient and flexible approach to executing natural language processing tasks. It allows us to complete specific tasks effectively without fine-tuning the model, thus avoiding the significant overhead associated with fine-tuning. Through carefully designed Prompts, we can unlock the inherent potential of large language models, enabling them to excel in multiple domains such as **domain-specific tasks**, **data augmentation**, and **intelligent agents**.

1. Domain-Specific Tasks

Applying prompt engineering to guide large language models in completing domain-specific tasks can avoid the need for specific fine-tuning for each task. This not only avoids the high computational cost of fine-tuning models but also reduces dependence on labeled data, allowing large language models to be better applied to domain-specific tasks. For example, in Text-to-SQL tasks, we can use prompt engineering techniques to guide the large language model to generate high-quality SQL queries directly based on user input text without supervised fine-tuning. Prompt engineering methods based on GPT models achieved breakthrough results on the Spider [44] leaderboard, surpassing traditional fine-tuning methods. Additionally, in knowledge-intensive question-answering domains such as MMLU [9], prompt engineering-based approaches have achieved the best results.

2. Data Augmentation

Using prompt engineering to perform data augmentation with large language models can improve the quality of existing datasets and generate new high-quality data. These data can be used to train and optimize other models, distilling the capabilities of large language models into other models using synthetic data. For example, we can guide the ChatGPT model to generate datasets containing rich reasoning steps to enhance the reasoning ability of Text-to-SQL models in the financial domain [45]. Moreover, through carefully designed prompts, we can also generate datasets containing complex instructions, such as Alpaca [32] and Evol-Instruct [21]. Using these synthetic datasets to fine-tune smaller models can enable them to approach the performance of larger models while maintaining small model sizes and low computational costs.

3. Intelligent Agents

Applying prompt engineering can construct large language models as intelligent agents (Intelligent Agent, IA) [5]. Intelligent agents, also known as agents, can perceive their environment, take actions autonomously to achieve goals, and improve their performance through learning or acquiring knowledge. In the process of intelligent agents perceiving their environment, taking actions, and learning knowledge, prompt engineering plays a crucial role. For example, Stanford University simulated a virtual Western town using GPT-4 [25], where multiple GPT-4-based agents lived and interacted within it. They acted autonomously according to their roles and goals, communicated, solved problems, and promoted the development of the town. The entire operation of the virtual Western town was driven by prompt engineering.

This section discusses the concept of prompts, prompt engineering, and its significance, revealing the key role and vast potential of prompts in the application of large language models. Next, we will further expand on this topic: Section 3.2 will explore context learning and reveal its role in enhancing the model's understanding and response capabilities; Section 3.3 will provide a detailed introduction to chain-of-thought prompting methods and their variants, demonstrating how these methods can enhance the model's logical reasoning and problem-solving abilities; Section 3.4 will share tips for construct-

---

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

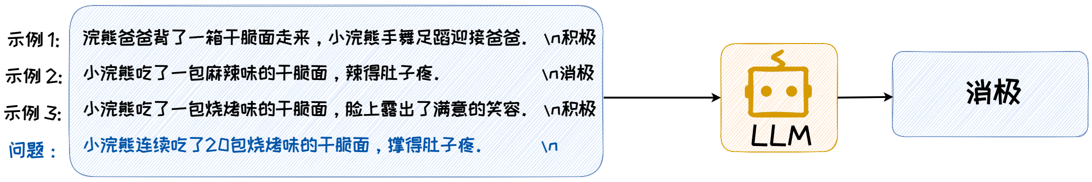
ing effective prompts, guiding readers on how to design prompts to elicit higher-quality content from the model; Section 3.5 will showcase practical applications of prompt engineering in large language models, illustrating application strategies and effects in different scenarios through examples.

3.2 In-Context Learning

As the scale of training data and the number of parameters in models continue to expand, large language models have emerged with the capability of **In-Context Learning** (ICL). This enables language models to acquire the ability to handle new tasks through given task instructions or examples. By introducing In-Context Learning, we no longer need to train a model for a specific task or perform time-consuming fine-tuning on pre-trained models to **rapidly adapt** to some downstream tasks. This allows users to leverage large language models to solve downstream tasks simply through web pages or APIs, laying a solid foundation for the "**Language Model as a Service**" (LLM as a Service) model. This section introduces In-Context Learning from the perspectives of its **definition**, **demonstration example selection**, and **factors affecting its performance**.

3.2.1 Definition of In-Context Learning

In-Context Learning (ICL) [2] is a paradigm that enables language models to understand and learn downstream tasks by constructing specific Prompts, which can include elements such as **demonstration examples** and **task instructions**. The key to implementing In-Context Learning lies in how to design effective Prompts to guide the model to understand the context and objectives of the task. Typically, these Prompts will contain task instructions along with a series of examples, allowing the model to learn the logic and rules of the task from this contextual information, thus generating outputs that meet the requirements of the task without additional training. Based on the above advantages, In-Context Learning has been widely applied in solving domain-specific tasks, data augmentation, intelligent agents, and other applications.



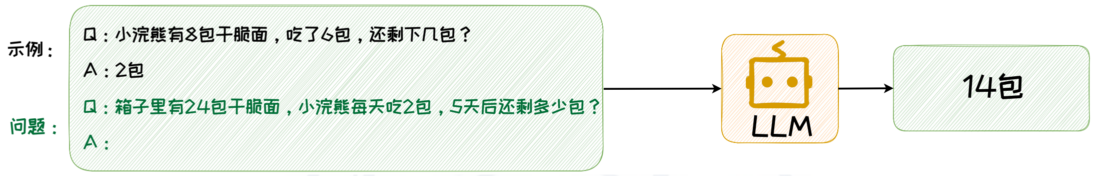
示例 1: 浣熊爸爸背了一箱干脆面走来，小浣熊手舞足蹈迎接爸爸. \n积极

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

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

问题： 小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼. \n

(a) In-Context Learning for classification tasks.



示例： Q：小浣熊有8包干脆面，吃了6包，还剩下几包？

A：2包

问题： Q：箱子里有24包干脆面，小浣熊每天吃2包，5天后还剩多少包？

A：

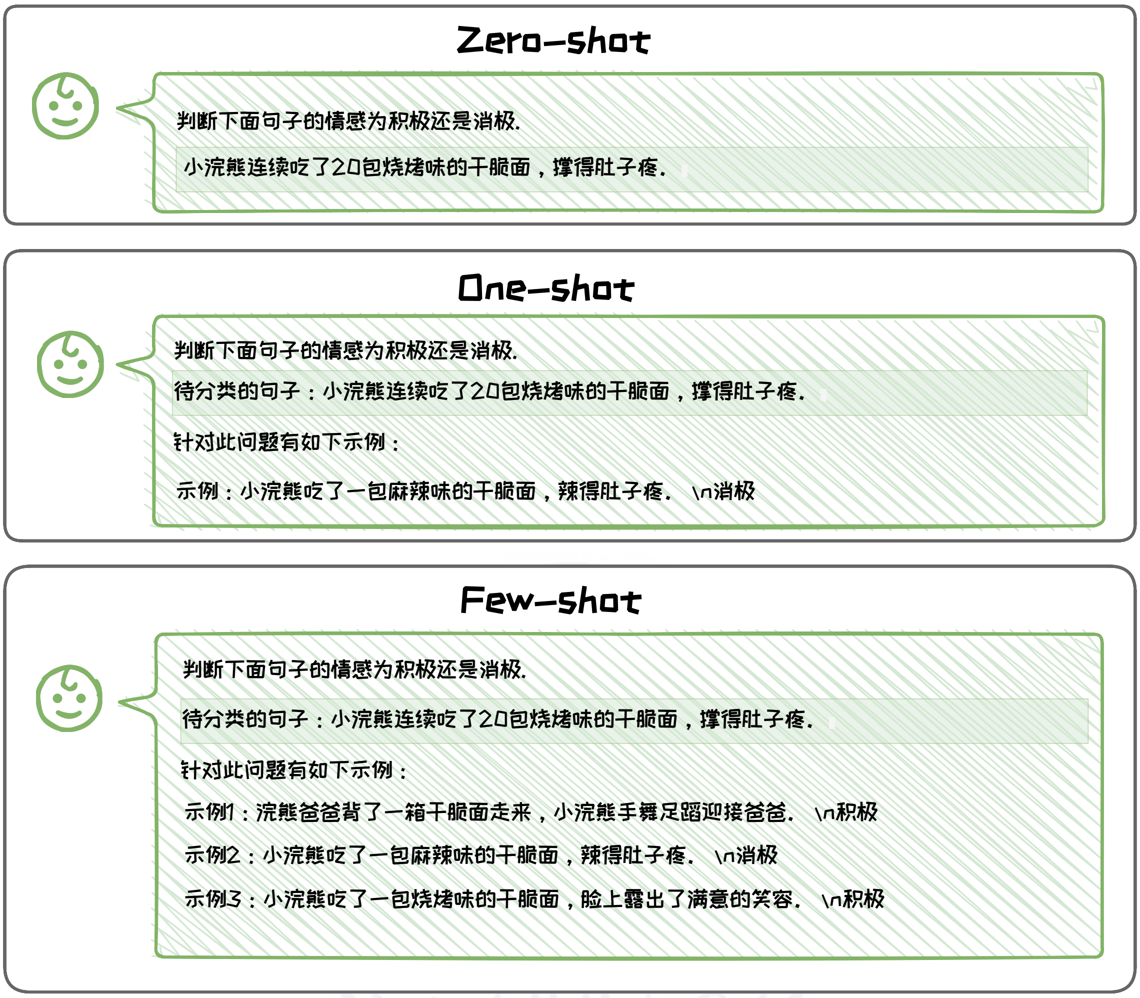
(b) In-Context Learning for generation tasks.

Figure 3.5: Examples of In-Context Learning.

In In-Context Learning, the Prompt usually contains several demonstration examples related to the task at hand, showcasing the relationship between task inputs and expected outputs. These examples are arranged in a specific order to form the context, which is concatenated with the question to form the Prompt input for the large language model. The large language model learns the task paradigm from the context and uses its own capabilities to answer the task. In example (a) of Figure **??**, the model is used for a text sentiment classification task; given a piece of text, the model can determine its sentiment orientation, identifying whether the text expresses positive or negative emotions. Example (b) of Figure **??** demonstrates a mathematical operation task, where the model follows the format of the examples to directly provide the corresponding calculation results. Depending on the number of examples, In-Context Learning can take various forms: **zero-shot** (Zero-shot) In-Context Learning, **one-shot** (One-shot) In-Context Learning, and **few-shot** (Few-shot)

In-Context Learning [2], as shown in Figure **??**.

- **Zero-shot In-Context Learning**: Under this learning method, only the task description is provided to the model, without any examples. It has strong generalization capabilities across different scenarios. However, the performance of zero-shot learning is entirely dependent on the capabilities of the large language model and may not perform well when handling tasks.
- **One-shot In-Context Learning**: This method requires providing only one example to the model, aligning with the human ”learning by analogy” approach. However, the effectiveness of one-shot learning strongly depends on the representativeness of the example relative to the task.
- **Few-shot In-Context Learning**: This method significantly improves the model’s performance on specific tasks by providing a small number of examples (usually a few to a dozen). However, the increase in examples significantly raises the computational cost during the inference of the large language model. The representativeness and diversity of the examples also impact the quality of the generated output.



Zero-shot

判断下面句子的情感为积极还是消极.

小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼.

One-shot

判断下面句子的情感为积极还是消极.

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.

针对此问题有如下示例：

示例：小浣熊吃了一包麻辣味的干脆面，辣得肚子疼. \n消极

Few-shot

判断下面句子的情感为积极还是消极.

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.

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

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

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

Figure 3.6: Examples of the three forms of In-Context Learning.

Despite the excellent performance of In-Context Learning in many tasks, why it works remains an important research question. A study from Stanford University [40] provides an explanation— “viewing In-Context Learning as implicit Bayesian inference.” During the pre-training phase of large language models, the model learns latent concepts from a vast amount of text. When using In-Context Learning for inference, the large language model leverages the demonstration examples to “anchor” the relevant concepts learned during pre-training, thereby performing In-Context Learning and making predictions. For example, in Figure **??**, the model can provide the correct answer because it has already learned concepts related to sentiment during pre-training, such as the manifestations of positive sentiment (satisfied smiles, dancing joyfully...), syntactic structures, and syntactic relationships, etc. When the model performs inference, it uses examples like "The raccoon ate a pack of spicy crispy noodles and got a stomachache. \nNegative" to "anchor" to concepts related to sentiment and, based on these concepts, provides the answer to the question.

3.2.2 Demonstration Example Selection

In In-Context Learning, demonstration examples play a crucial role in guiding large language models to understand the task, and their content and quality directly impact the learning effectiveness. Therefore, selecting appropriate demonstration examples is essential for improving the performance of In-Context Learning. Demonstration example selection primarily relies on similarity and diversity [20]:

- **Similarity** refers to carefully selecting examples that are most similar to the problem to be solved. Similarity can be measured at multiple levels, such as linguistic similarity (including keyword matching or semantic similarity matching), structural similarity, etc. By selecting similar examples, the model is provided with a reference close to the problem to be solved, making it easier for the large language model to understand the problem.
- **Diversity** requires that the selected examples cover as broad a range of content as possible, expanding the coverage of the demonstration examples over the problem to be solved. Diverse examples help the model understand the task from different angles, enhancing its ability to handle various problems.

In addition to similarity and diversity, certain task-related factors must be considered in some tasks. This section focuses on the two factors of similarity and diversity and discusses how to select appropriate demonstration examples from a large pool of candidate examples based on similarity and diversity. The following sections introduce three types of example selection methods based on similarity and diversity [20]:

1. Direct Retrieval

Given a set of candidate examples, the direct retrieval method ranks the candidate examples based on their similarity to the problem to be solved, then selects the top K examples. A representative method of direct retrieval is KATE [17]. As shown in Figure 3.7, KATE uses RoBERTa to encode the problem to be solved and the candidate examples (excluding labels). Then, it calculates the cosine similarity between the encoded problem and the encoded candidate examples to score their similarity. Based on this score, the top K examples with the highest scores are selected as the demonstration examples for In-Context Learning. The direct retrieval method is simple and easy to implement, making it one of the most widely used example selection strategies. However, it does not consider the diversity of the examples, leading to potential **homogenization** of the selected examples.

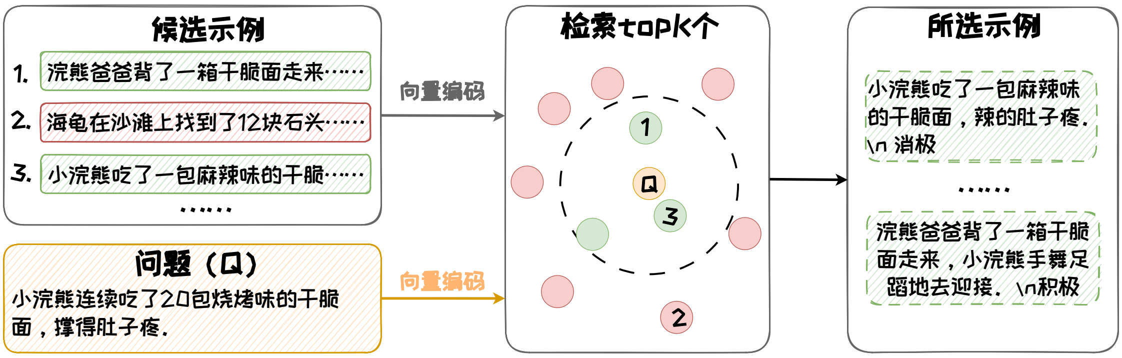
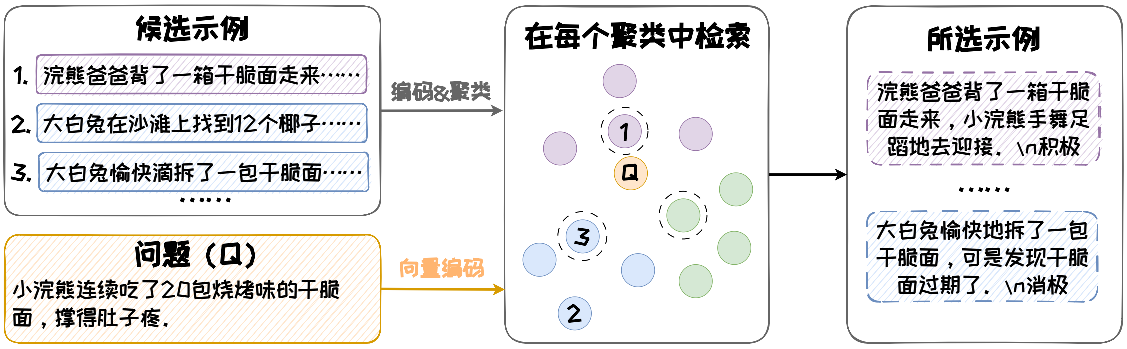

候选示例
1. 浣熊爸爸背了一箱干脆面走来……
2. 海龟在沙滩上找到了12块石头……
3. 小浣熊吃了一包麻辣味的干脆……
……
问题（Q）
小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼.
向量编码
向量编码
检索topK个
1
Q
3
2
所选示例
小浣熊吃了一包麻辣味的干脆面，辣的肚子疼. \n 消极
……
浣熊爸爸背了一箱干脆面走来，小浣熊手舞足蹈地去迎接. \n积极

Figure 3.7: Direct Retrieval.

2. Clustering Retrieval

To address the issue of homogenization in direct retrieval, clustering retrieval methods adopt a two-step approach—clustering followed by retrieval—to ensure the **diversity** of the retrieved results. First, all candidate examples are divided into K clusters, and then the most similar example from each cluster is selected. This effectively avoids selecting multiple similar examples, thereby ensuring diversity. Self-Prompting [15] is a representative method of this approach. As shown in Figure 3.8, Self-Prompting first encodes the candidate examples and the problem to be solved into vector form, then applies the K-Means algorithm to cluster the example set into K clusters. According to the cosine similarity between the problem and the examples, the most similar example from each cluster is selected, resulting in K examples. Although the clustering retrieval method enhances the diversity of the examples, some clusters may not be similar to the problem, leading to potentially insufficient **similarity** of the selected examples.



浣熊爸爸背了一箱干脆面走来……

大白兔在沙滩上找到12个椰子……

大白兔愉快滴拆了一包干脆面……

小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼.

浣熊爸爸背了一箱干脆面走来，小浣熊手舞足蹈地去迎接. \n积极

大白兔愉快地拆了一包干脆面，可是发现干脆面过期了. \n消极

Figure 3.8: Clustering Retrieval.

ZHEJIANG UNIVERSITY 1897

3. Iterative Retrieval

Direct retrieval and clustering retrieval often fail to balance similarity and diversity. To **balance both similarity and diversity**, iterative retrieval strategies have been developed. Iterative retrieval first selects examples highly similar to the problem, and then dynamically selects the next example in subsequent iterations, combining the current problem and the already selected examples to ensure both the similarity and diversity of the selected examples. RetICL [28] is a representative method of iterative retrieval, as shown in Figure 3.9. RetICL initializes the internal state of an LSTM-based retriever [10] based on the current problem and selects an example. It then updates the internal state of the retriever based on the current problem and the selected example set, and selects the next example. This process continues iteratively until k examples are obtained. Although iterative retrieval is computationally more complex, it can generate better example sets, demonstrating superior adaptability and flexibility in complex tasks.

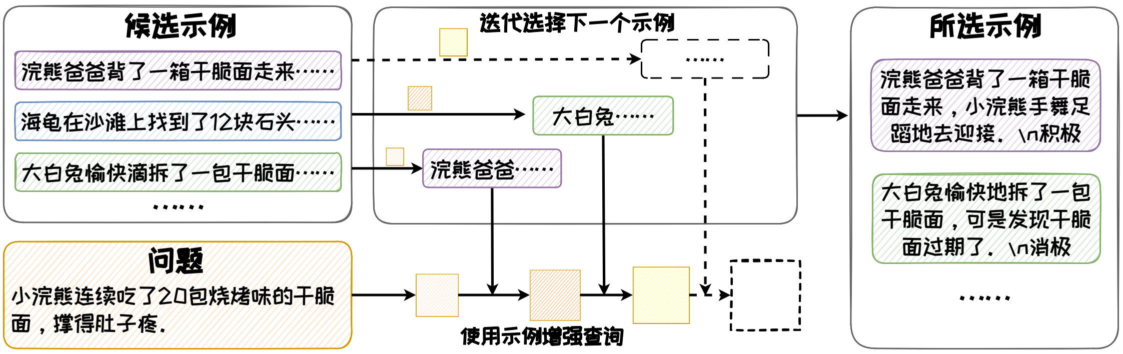

候选示例
浣熊爸爸背了一箱干脆面走来……
海龟在沙滩上找到了12块石头……
大白兔愉快滴拆了一包干脆面……
……
问题
小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼.
迭代选择下一个示例
……
大白兔……
浣熊爸爸……
使用示例增强查询
所选示例
浣熊爸爸背了一箱干脆面走来，小浣熊手舞足蹈地去迎接. \n积极
大白兔愉快地拆了一包干脆面，可是发现干脆面过期了. \n消极
……

Figure 3.9: Iterative Retrieval.

In addition to differences in example selection strategies, existing methods also exhibit variations in the choice and design of retrievers. Some methods use off-the-shelf retrievers, while others opt to fine-tune retrievers on specific corpora to achieve superior performance. A detailed introduction to retrievers and their classification will be provided in Chapter 6 6.8.

3.2.3 Factors Affecting Performance

By carefully designing example selection strategies, the effectiveness of In-Context Learning can be significantly improved. However, besides example selection, the performance of In-Context Learning is influenced by multiple factors. These factors include **pre-training data**, **pre-trained models**, and **demonstration examples** [48]. This subsection will discuss how these key factors influence the performance of In-Context Learning.

1. Impact of Pre-Training Data

Pre-training data is the source of In-Context Learning capabilities and profoundly affects its performance. For pre-training data, the following three aspects are key factors influencing the performance of In-Context Learning:

- **Domain Richness**: The richness of domains covered by pre-training data directly impacts the model's domain generalization ability. Models pre-trained on rich cross-domain corpora possess more stable In-Context Learning capabilities. On the other hand, single-domain corpora may limit the model's adaptability, even if the domain is highly relevant to the target task, and cannot guarantee optimal In-Context Learning performance [31].
- **Task Diversity**: Task diversity in pre-training data is a crucial factor in enhancing the performance of In-Context Learning. Diverse task types help the model learn a broader range of knowledge and skills, enhancing its task generalization ability, and thus enabling it to perform better on new tasks [26].
- **Distribution Characteristics of Training Data**: The presence of bursty distributions and rare categories in training data can enhance the model's In-Context Learning capabilities. Bursty distributions ensure that the ”text-label mapping” data patterns recur throughout the training process, while rare categories improve the

model's ability to handle rare or novel inputs, thereby enhancing the model's In-Context Learning performance [4].

2. Impact of Pre-Trained Models

The impact of pre-trained models on the performance of In-Context Learning is primarily reflected in the **scale of model parameters**. When the model parameters reach a certain scale, In-Context Learning capabilities can emerge. Typically, the number of parameters in a model needs to reach hundreds of millions or more. Among the commonly used models with In-Context Learning capabilities, the smallest is the Qwen2-0.5B model from Alibaba's Tongyi Lab, which has 500 million parameters. Generally, the larger the model, the stronger its In-Context Learning performance [39]. Additionally, the architecture and training strategies of the model are also important factors affecting the performance of In-Context Learning.

3. Impact of Demonstration Examples

In Section 3.2.2, we discussed the importance of demonstration example selection for In-Context Learning. Next, we will further explore the impact of **example format**, **input-output mapping**, and **number and order of examples** on In-Context Learning.

Example Format. Different tasks have different requirements for example formats. For instance, in the sentiment classification example shown in Figure **??**, we only need to provide the input and output. However, for complex reasoning tasks such as arithmetic or coding, simply providing the input and output is insufficient for the model to grasp the reasoning process. In such cases, constructing examples in a chain-of-thought format, i.e., adding intermediate reasoning steps between the input and output, can help the model reason step-by-step and learn more complex mappings. Chain-of-thought will be detailed in Section 3.3.

Correctness of Input-Output Mapping. For an example containing input and output, the large language model aims to learn the input-output mapping to complete the target task. If the input-output mapping in the given example is incorrect, it will inevitably affect the performance of In-Context Learning. For example, in a sentiment classification task, mislabeling a positive text as negative can cause the model to learn this incorrect mapping, leading to errors in downstream tasks. The sensitivity to incorrect input-output mappings is related to the size of the model [14, 43]. Larger models exhibit higher sensitivity to the correctness of input-output mappings in In-Context Learning [14]. In contrast, smaller models show weaker sensitivity to incorrect input-output mappings [24, 39].

Number and Order of Demonstration Examples. Increasing the number of demonstration examples generally improves the performance of In-Context Learning, but the rate of performance improvement gradually slows down as the number of examples increases [23]. Moreover, generation tasks benefit more from an increased number of examples compared to classification tasks [16]. Additionally, the order of demonstration examples is a key factor affecting the performance of In-Context Learning, with significant differences in model performance under different example orders. The optimal order of examples is model-dependent, meaning that an effective order for one model may not be suitable for another [19]. Furthermore, the choice of example order is also influenced by the specific characteristics of the dataset being used [17].

Apart from the above factors, the quality of task instructions in the Prompt also directly affects the performance of In-Context Learning. Clear and explicit task instructions can provide clear guidance to the model, thereby enhancing the performance of In-Context Learning [43]. Therefore, when designing demonstration examples and task instructions, it is important to consider these factors comprehensively to optimize the model's performance.

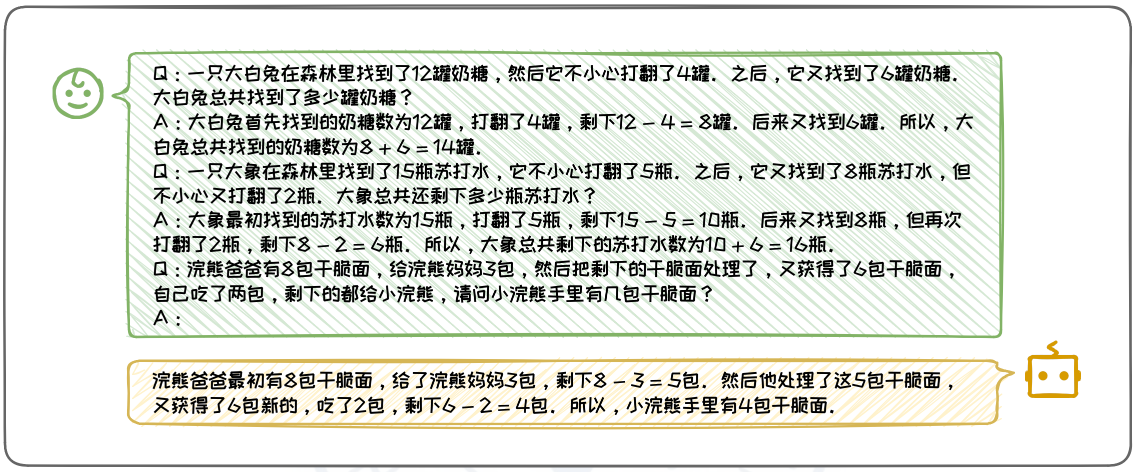
This section introduced In-Context Learning, distinguishing between zero-shot, one-shot, and few-shot forms. It discussed three example selection strategies—direct retrieval, clustering retrieval, and iterative retrieval—analyzing their advantages, disadvantages, and representative methods. Finally, it analyzed the factors affecting the performance of In-Context Learning from the perspectives of pre-training data, the model itself, and demonstration examples.

3.3 Chain of Thought

As the scale of language model parameters continues to expand, they can better capture linguistic features and structures, thereby significantly enhancing performance in natural language processing tasks such as semantic analysis, text classification, and machine translation. However, when faced with tasks requiring complex reasoning abilities, such as arithmetic solving, common sense judgment, and symbolic reasoning, the growth in model parameter scale has not brought about the expected performance breakthrough. This phenomenon is known as ”Flat Scaling Curves” [38]. This indicates that simply increasing the model size is not enough to solve all problems, and we need to explore new methods to enhance the model’s reasoning capabilities and intelligence levels. When humans solve complex problems, they typically build a chain of reasoning step by step to derive the final answer. Based on this concept, an innovative prompt paradigm—the **Chain-of-Thought (CoT) prompt** [38]—is used to guide the model through step-by-step reasoning. CoT can significantly improve the performance of large language models in handling complex tasks, thus breaking through the limitations of ”Flat Scaling Curves” and unleashing the intrinsic reasoning potential of large language models.

3.3.1 Definition of Chain-of-Thought Prompt

The **Chain-of-Thought (CoT) prompt** [38] simulates the thinking process humans use when solving complex problems, guiding large language models to introduce a series of **intermediate reasoning steps** during the answer generation process. This method not only significantly improves the model's performance in reasoning tasks but also reveals the internal logic and reasoning paths the model uses when dealing with complex problems.



Q：一只大白兔在森林里找到了12罐奶糖，然后它不小心打翻了4罐．之后，它又找到了6罐奶糖．大白兔总共找到了多少罐奶糖？
A：大白兔首先找到的奶糖数为12罐，打翻了4罐，剩下12 − 4 = 8罐．后来又找到6罐．所以，大白兔总共找到的奶糖数为8 + 6 = 14罐．
Q：一只大象在森林里找到了15瓶苏打水，它不小心打翻了5瓶．之后，它又找到了8瓶苏打水，但不小心又打翻了2瓶．大象总共还剩下多少瓶苏打水？
A：大象最初找到的苏打水数为15瓶，打翻了5瓶，剩下15 − 5 = 10瓶．后来又找到8瓶，但再次打翻了2瓶，剩下8 − 2 = 6瓶．所以，大象总共剩下的苏打水数为10 + 6 = 16瓶．
Q：浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？
A：

浣熊爸爸最初有8包干脆面，给了浣熊妈妈3包，剩下8 − 3 = 5包．然后他处理了这5包干脆面，又获得了6包新的，吃了2包，剩下6 − 2 = 4包．所以，小浣熊手里有4包干脆面．

Figure 3.10: Example of a CoT prompt with a few sample instances.

The core of the CoT method is to construct a **suitable Prompt** to trigger the large language model to **generate a reasoning path step by step** and produce the final answer. Early methods added a few examples containing the reasoning process (Few-Shot Demonstrations) [38] when constructing the Prompt, to guide the model to generate answers step by step. In these examples, researchers carefully wrote out the reasoning processes for related problems, providing the model with examples to imitate and learn from. This method enables the model to learn how to generate reasoning steps from these examples and output the answer step by step. Figure 3.10 shows an example of a CoT-formatted Prompt used to solve a math problem. The example provides the solution steps for a related math problem as a reference, and the large language model will mimic this example to solve complex mathematical calculations step by step. By introducing CoT, the performance of large language models in solving complex problems such as arithmetic, common sense judgment, and symbolic reasoning has been significantly improved. Moreover, under the influence of CoT, the ability of large language models to handle complex problems increases as the model parameter scale grows.

Guided by the core idea of CoT, a series of extended methods have been derived. These extended methods can be categorized into three modes based on their reasoning approach: **Step-by-Step**, **Think Before Acting**, and **Pooling Ideas**. The comparison of these modes is shown in Figure 3.11.

- **Step-by-Step.** In the Step-by-Step mode, the model performs reasoning step by step, forming a logically coherent chain of reasoning. In this mode, the model follows a predefined logical path, moving forward ”step by step.” Methods such as CoT [38], Zero-Shot CoT [12], and Auto-CoT [46] are representative of this mode.
- **Think Before Acting.** In the Think Before Acting mode, the model stops at each step to assess the current situation and then selects the next direction from multiple reasoning options. In this mode, the model is like exploring an unknown forest, stopping at each step to evaluate the surrounding environment and ”think before acting” to find the best reasoning path. Methods such as ToT [42] and GoT [1] are representative of this mode.
- **Pooling Ideas.** In the Pooling Ideas mode, the model generates multiple reasoning paths simultaneously and obtains multiple results, then integrates these results to produce a more comprehensive and accurate answer. In this mode, the model is like holding a meeting of wise individuals, where each person brings their own insights,

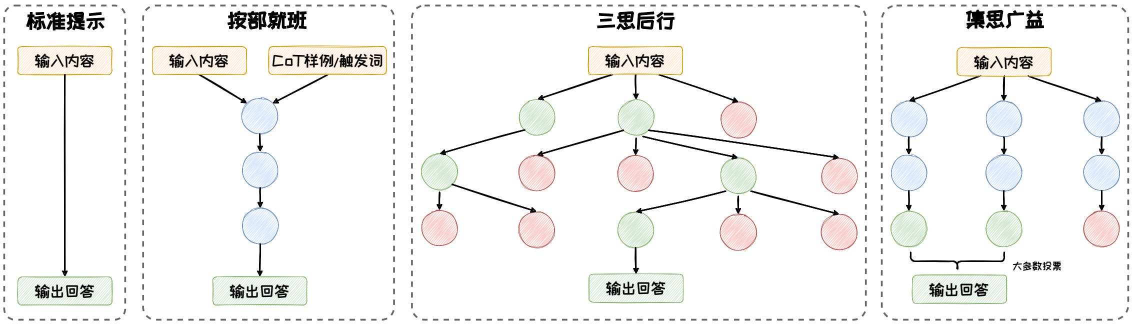


Figure 3.11: Comparison of different CoT structures.

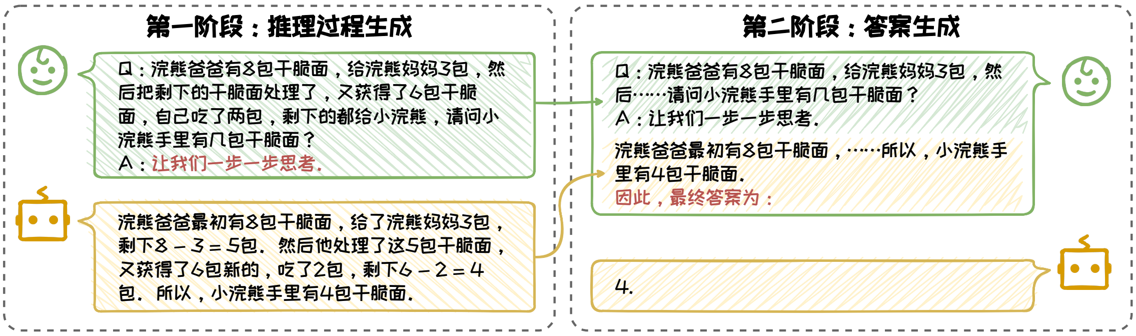
and ultimately, through discussion and integration, ”pooling ideas” leads to a better conclusion. Methods such as Self-Consistency [36] are representative of this mode.

3.3.2 Step-by-Step

The Step-by-Step mode emphasizes **logical coherence** and **sequential order**. In this mode, the model performs reasoning step by step, ultimately reaching a conclusion. It ensures that the reasoning process is clear and orderly, making the model’s decision-making process more transparent and predictable. The original few-shot Chain-of-Thought (CoT) method adopted the Step-by-Step mode. It involves manually constructing a few examples of step-by-step reasoning to answer questions and including them in the Prompt to guide the model to generate reasoning steps step by step and produce the final answer. This method has achieved some success in improving the model’s reasoning ability, but it requires time-consuming and labor-intensive manual creation of numerous CoT examples and heavily depends on the quality of the CoT examples. To address these issues, researchers have extended the original CoT method. This section will introduce two variants of CoT: Zero-Shot CoT and Auto-CoT.

3.3.2.1 1. Zero-Shot CoT

Zero-Shot CoT [12] guides the model to generate a reasoning chain through simple prompts, such as "Let's think step by step." It does not require manually labeled CoT examples, reducing the dependence on human-generated examples. It has shown performance comparable to or even superior to the original few-shot CoT in multiple reasoning tasks.



Q：浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？
A：让我们一步一步思考.

浣熊爸爸最初有8包干脆面，给了浣熊妈妈3包，剩下8 − 3 = 5包. 然后他处理了这5包干脆面，又获得了6包新的，吃了2包，剩下6 − 2 = 4包. 所以，小浣熊手里有4包干脆面.

Q：浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后……请问小浣熊手里有几包干脆面？
A：让我们一步一步思考.
浣熊爸爸最初有8包干脆面，……所以，小浣熊手里有4包干脆面.
因此，最终答案为：

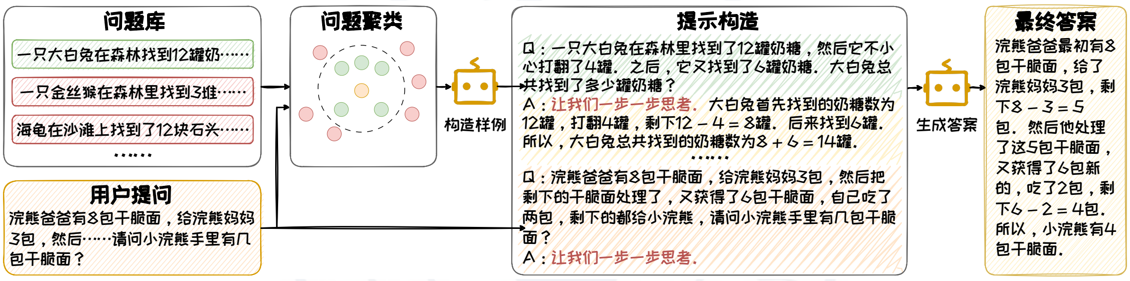
4.

Figure 3.12: The process of Zero-Shot CoT prompts.

The overall process of Zero-Shot CoT is shown in Figure 3.12. It uses a two-stage method to answer questions. First, in the first stage, the phrase "Let's think step by step" or "让我们一步一步思考" is appended to the question as a trigger for the CoT prompt, instructing the large language model to generate intermediate reasoning steps before producing the final answer. In the second stage, the original question and the reasoning steps generated in the first stage are concatenated, and the phrase "Therefore, the answer is" or "因此，最终答案为" is added at the end. This content is then fed to the large language model to generate the final answer. Through this method, the intrinsic reasoning capability of the large language model can be activated without the need for manually labeled CoT data. The large language model can reason step by step to arrive at the correct answer, demonstrating the potential of Zero-Shot CoT in enhancing the model's reasoning ability.

3.3.2.2 2. Auto CoT

Building on the foundation of Zero-Shot CoT, Auto-CoT [46] introduces related problems and their reasoning chains as examples to further improve the effectiveness of CoT. The generation of these examples is automatically completed by the large language model, eliminating the need for manual labeling. The process of Auto-CoT is shown in Figure 3.13 and includes the following steps:


问题库
一只大白兔在森林找到12罐奶……
一只金丝猴在森林里找到3堆……
海龟在沙滩上找到了12块石头……
……
用户提问
问题聚类
构造样例
提示构造
生成答案
最终答案

浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后……请问小浣熊手里有几包干脆面？

Q：一只大白兔在森林里找到了12罐奶糖，然后它不小心打翻了4罐。之后，它又找到了6罐奶糖。大白兔总共找到了多少罐奶糖？
A：让我们一步一步思考。大白兔首先找到的奶糖数为12罐，打翻4罐，剩下12 – 4 = 8罐。后来找到6罐。所以，大白兔总共找到的奶糖数为8 + 6 = 14罐。
……
Q：浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？
A：让我们一步一步思考。

浣熊爸爸最初有8包干脆面，给了浣熊妈妈3包，剩下8 – 3 = 5包。然后他处理了这5包干脆面，又获得了6包新的，吃了2包，剩下6 – 2 = 4包。所以，小浣熊有4包干脆面。

Figure 3.13: The process of Auto-CoT prompts.

ZHEJIANG UNIVERSITY 1897

- Use clustering techniques to filter out questions from the question bank that belong to the same cluster as the user's query.
- Then, using the Zero-Shot CoT method, generate reasoning chains for the filtered questions to form examples. These examples include different problems and their corresponding reasoning content, providing the model with various problem-solving approaches and assisting it in making more cautious inferences.
- Based on these examples, Auto-CoT uses the phrase “Let's think step by step” to guide the large language model to generate the reasoning chain and answer for the user's question.

3.3.3 Think Before Acting

The Think Before Acting mode emphasizes the incorporation of **caution** and **flexibility** in the decision-making process. In this mode, the model stops at each step to assess the current situation and determine whether to adjust the reasoning direction. The core of this mode lies in allowing the model to backtrack and reselect when encountering difficulties or uncertainties, ensuring the robustness and adaptability of the decision-making process. This mode mimics the process humans go through when solving problems, where they repeatedly choose and backtrack. People continuously select the best option from multiple candidate answers, and if one line of thought does not work, they backtrack to the starting point and choose another line of thought to proceed. Based on this observation from real life, researchers have proposed variants of CoT under the Think Before Acting mode, such as the Tree of Thoughts (ToT) [42] and the Graph of Thoughts (GoT) [1].

ToT constructs the reasoning process as a tree of thoughts, which is built from the following four perspectives:

- **Decomposition**. Break down complex problems into multiple simpler sub-problems, with each sub-problem's solution process corresponding to a thought process. The form and granularity of decomposition depend on the task category, for example, using a single equation as a thought process in mathematical reasoning tasks, and using an outline as a thought process in creative writing tasks.
- **Derivation**. The model needs to generate possible next reasoning directions based on the current sub-problem. Derivation has two modes: sample inspiration and command prompting. Sample inspiration uses multiple independent examples as context to increase the derivation space, suitable for tasks with broad thinking spaces

1897 ZHEJIANG UNIVERSITY

such as creative writing; command prompting specifies rules and requirements in the Prompt to limit the derivation space, suitable for tasks with limited thinking spaces such as the 24-point game.

- **Evaluation**. Use the model to evaluate the rationality of reasoning nodes. Depending on whether the task is easy to score quantitatively, choose between voting or scoring modes. In the voting mode, the model selects among multiple nodes based on votes to decide which nodes to retain; in the scoring mode, the model scores the nodes and decides which nodes to retain based on the scores.
- **Search**. Start from one or more current states to search for paths leading to the solution of the problem. Choose different search algorithms based on the characteristics of the task. Classical search algorithms such as depth-first search and breadth-first search can be used, as well as heuristic search algorithms such as A* search and Monte Carlo Tree Search.

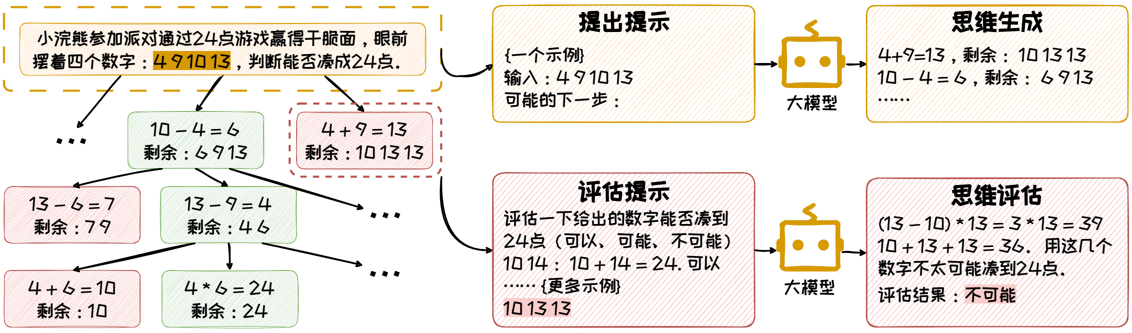


Figure 3.14: The process of ToT prompts.

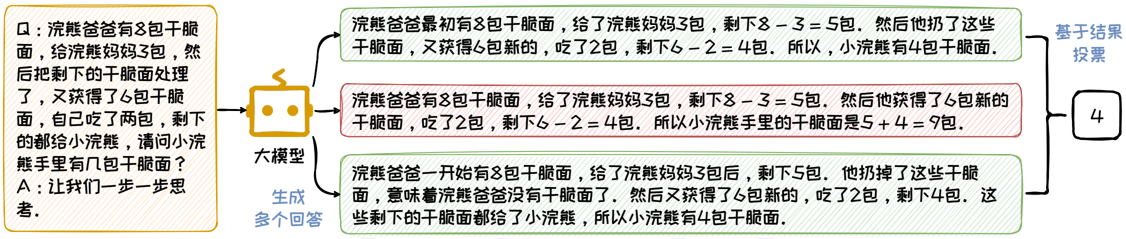
Figure 3.14 demonstrates a specific example of ToT using the 24-point game. In this example, given four numbers, the large language model uses the four arithmetic operators (addition, subtraction, multiplication, division) to combine these four numbers so that the final result is 24. First, ToT, based on the remaining numbers, uses contextual learning to have the model select two numbers for an operation and generate multiple solutions, represented as multiple child nodes in the thought tree. Next, it traverses each child node using a breadth-first search, evaluates whether the remaining numbers can sum up to 24, and retains the nodes that can potentially reach 24, which is also achieved through contextual learning. These two steps are repeated until a final reasonable result is obtained.

In addition to ToT, GoT extends the tree structure to a directed graph, providing operations for self-assessment and correction of each thought as well as the aggregation of thoughts. In this graph, vertices represent solutions to a problem (initial problem, intermediate problem, final problem), and directed edges represent the process of constructing a thought ”in-node” using the ”out-node” as direct input. The core advantage of GoT over ToT is its ability to self-reflect and aggregate thoughts, integrating knowledge and information from different thought paths to form a comprehensive solution.

3.3.4 Pooling Ideas

The Pooling Ideas mode emphasizes optimizing the decision-making process by **gathering multiple different perspectives and methods**. In this mode, the model does not rely on a single reasoning path but explores multiple possible solutions and selects the optimal answer. This method draws on the concept of collective wisdom, which posits that integrating multiple independent thinking results can lead to a more comprehensive and accurate conclusion. Inspired by collective wisdom, researchers explored how to enhance the model’s reasoning ability through consistency, proposing the Self-Consistency [36] method. This method introduces diverse reasoning paths and selects the most consistent answer, thereby improving the accuracy of the model’s reasoning. Self-Consistency is not dependent on a specific CoT format and can be compatible with other CoT methods, working together to enhance the model’s reasoning process.

As shown in Figure 3.15, the implementation process of Self-Consistency can be divided into three steps: (1) Under a random sampling strategy, use CoT or Zero-Shot CoT to guide the large language model to generate a set of diverse reasoning paths for the problem at hand; (2) For each reasoning content generated by the large language model, collect the final answers and count the frequency of each answer across all reasoning paths; (3) Select the answer with the highest frequency as the final, most consistent answer.


Q：浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？
A：让我们一步一步思考.

浣熊爸爸最初有8包干脆面，给了浣熊妈妈3包，剩下8 − 3 = 5包. 然后他扔了这些干脆面，又获得6包新的，吃了2包，剩下6 − 2 = 4包. 所以，小浣熊有4包干脆面.

浣熊爸爸有8包干脆面，给了浣熊妈妈3包，剩下8 − 3 = 5包. 然后他获得了6包新的干脆面，吃了2包，剩下6 − 2 = 4包. 所以小浣熊手里的干脆面是5 + 4 = 9包.

浣熊爸爸一开始有8包干脆面，给了浣熊妈妈3包后，剩下5包. 他扔掉了这些干脆面，意味着浣熊爸爸没有干脆面了. 然后又获得了6包新的，吃了2包，剩下4包. 这些剩下的干脆面都给了小浣熊，所以小浣熊有4包干脆面.

Figure 3.15: The process of Self-Consistency.

This section discusses the ideas and modes of Chain-of-Thought (CoT). The core idea of CoT is to simulate the human problem-solving thinking process in the prompt, embedding the reasoning process for solving problems in the Prompt. This significantly enhances the model's performance in reasoning tasks without the need for specific task fine-tuning. CoT methods include several modes: Step-by-Step, Think Before Acting, and Pooling Ideas. These modes cater to different reasoning needs and enhance the reasoning capabilities of large language models in complex tasks.

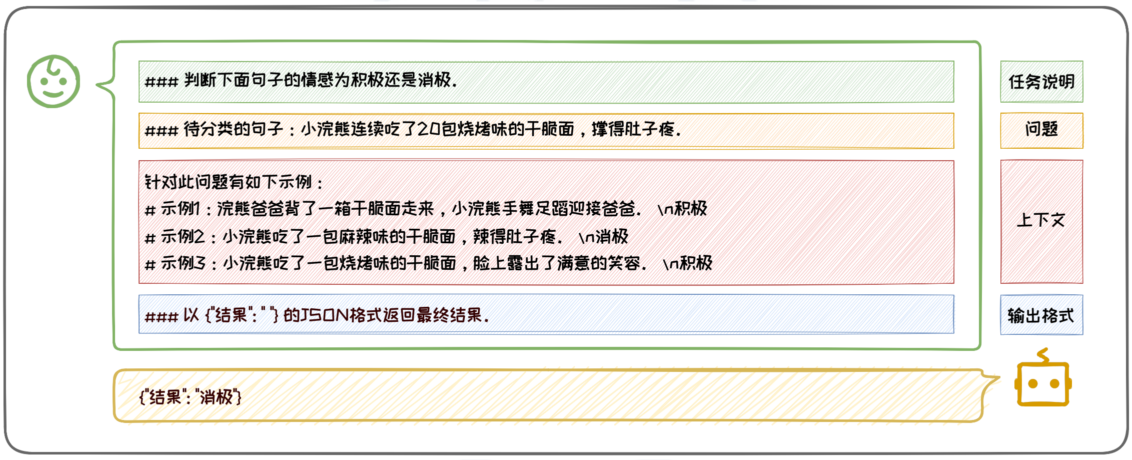
3.4 Prompt Techniques

Based on context learning and Chain-of-Thought (CoT) prompt engineering techniques, this section will further explore Prompt techniques that can be used to enhance the quality of large language model generation. These techniques include proper summarization of questions, timely use of CoT, and skillful use of psychological cues. Applying the

Prompt techniques introduced in this section can guide the model to generate more accurate and expected content, further improving the performance of large language models in practical applications.

3.4.1 Standardizing Prompt Writing

Writing standardized Prompts is the foundation of effective communication with large language models. Classic Prompts typically consist of one or more of the following components: **task description**, **context**, **question**, and **output format**. Consider the sentiment classification Prompt example shown in Figure 3.16. In this example:


### 判断下面句子的情感为积极还是消极.

### 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.

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
# 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
# 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

### 以{"结果":" "}的JSON格式返回最终结果.

{"结果":"消极"}

Figure 3.16: Example of a classic Prompt.

- **Task Description** is “### Determine whether the sentiment of the following sentence is positive or negative.” It clearly defines the task the model needs to perform.
- **Context** is “For this task, here are some examples: # Example 1: Raccoon dad carried a box of crispy noodles and walked over, and the baby raccoon danced joyfully to greet him. \n Positive \n # Example 2: The baby raccoon ate a pack of spicy crispy noodles and got a stomachache. \n Negative \n # Example 3: The baby raccoon ate a pack of barbecue-flavored crispy noodles and smiled with satisfaction.

\n Positive \n" . The context provides examples or background information to help the model understand and answer the question.

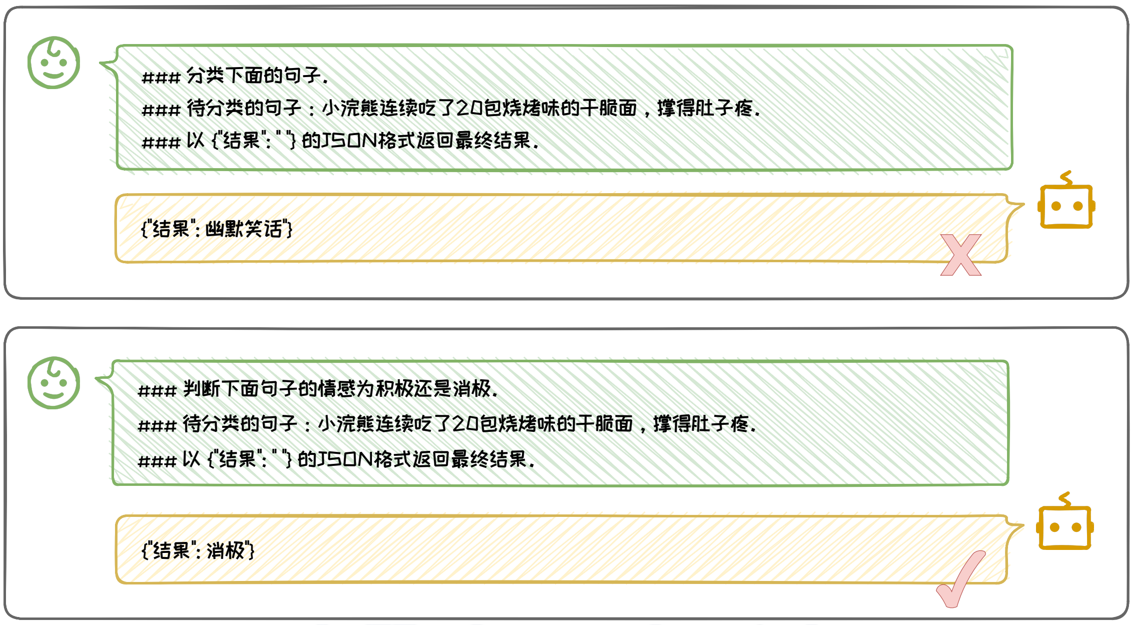
- **Question** is "Sentence to classify: The baby raccoon ate 20 packs of barbecue-flavored crispy noodles in a row and got a stomachache." This is the actual problem the user wants the model to solve. It can be a paragraph (e.g., a paragraph to be summarized in a summarization task), a real question (e.g., a user's question in a question-answering task), or other types of input content such as tables.
- **Output Format** is "Return the final result in JSON format: "result": " ". It standardizes the output format of the model.

From this example, it is evident that each component of a classic Prompt is important. Their standardization directly affects the quality of the model's output. Additionally, the layout of each component is also crucial. In the following, we will detail the requirements that need to be met for the standardized writing of classic Prompts.

1. Task Description Must Be Clear

A clear and specific task description is one of the key elements in building an effective Prompt. A clear and specific task description ensures that the model accurately understands the task requirements and produces outputs that meet expectations. For example, in a sentiment classification task, the task description "Determine whether the sentiment of the following sentence is positive or negative." is a clear example. It clearly defines the task type (sentiment classification) and the specific categories (positive or negative).

On the contrary, vague or unclear task descriptions can lead to the model misinterpreting the user's true intentions, resulting in outputs that do not meet expectations. As shown in Figure 3.17, a task description like "Classify the following sentence" lacks specificity, failing to clearly indicate the type and categories of classification, making it difficult for the model to accurately execute the task.



```
### 分类下面的句子.
### 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.
### 以 {"结果": ""} 的JSON格式返回最终结果.
```

```
{"结果": 幽默笑话"}
```

```
### 判断下面句子的情感为积极还是消极.
### 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.
### 以 {"结果": ""} 的JSON格式返回最终结果.
```

```
{"结果": 消极"}
```

Figure 3.17: Comparison of different task descriptions.

To ensure the clarity of the task description, we need to focus on the following points:

- **Use Clear Verbs**: Choose verbs that clearly express actions, such as "determine,""classify," "generate," etc., and avoid ambiguous verbs like "process" or "operate."
- **Specific Nouns**: Use specific nouns to define the output or target of the task. For example, "positive" and "negative" in a sentiment classification task provide clear classification criteria.
- **Concise and Clear**: The task description should be concise and direct, avoiding lengthy or complex sentence structures, so that the model can quickly grasp the core requirements of the task.
- **Structured Layout**: In longer Prompts, place the task description at the beginning and the end, as the model typically pays more attention to these parts [18]. This layout helps ensure that the model first and last encounters the most critical task

information.

By employing these strategies, we can ensure that the task description is both clear and specific, helping the model better understand and execute the task, ultimately producing high-quality outputs.

2. Context Should Be Rich and Clear

In Prompt design, the role of context cannot be overlooked, as it sometimes directly determines whether the model can provide the correct answer. A **rich and clear** context can significantly enhance the model's understanding and accuracy in responses.

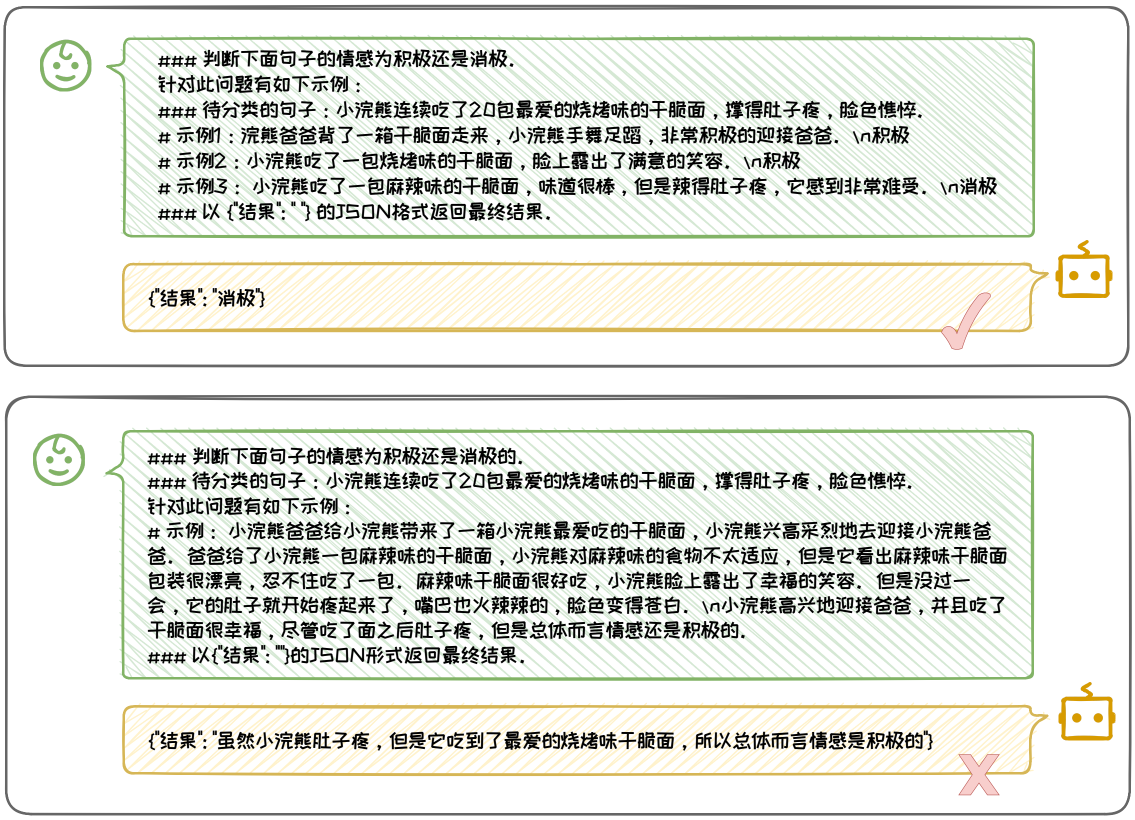
Richness of Context is reflected in the diversity and relevance of its content. Context can include background information directly related to the problem, specific demonstration examples, or continuous content from a conversation. For example, in a sentiment classification task, providing specific example sentences and their corresponding sentiment labels can help the model better understand the specific requirements of the task and the expected output.

Clarity of Context requires that the context information must be closely related to the problem, avoiding redundant or unnecessary information. Clear context should directly point to the core of the task, reducing confusion and misunderstanding when the model processes the information. For example, in a question-answering task, the context should only include information directly related to the question, avoiding the introduction of irrelevant content that might mislead the model.

In the two examples of context design shown in Figure 3.18, the first example's context is tightly focused on the problem, providing rich and relevant information without any redundancy. This design helps the model quickly focus on the key information, thus accurately answering the question. In contrast, the second example's context is less rich,

ZHEJIANG UNIVERSITY 1897

and a single example contains a lot of details unrelated to the problem. These redundant details not only make the context unclear but also increase the burden on the model when processing information, making it difficult for the model to accurately grasp the core of the problem, thereby affecting the accuracy of its response.



```
### 判断下面句子的情感为积极还是消极.
针对此问题有如下示例：
### 待分类的句子：小浣熊连续吃了20包最爱的烧烤味的干脆面，撑得肚子疼，脸色憔悴.
# 示例1：浣熊爸爸背了一箱干脆面走来，小浣熊手舞足蹈，非常积极的迎接爸爸. \n积极
# 示例2：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
# 示例3： 小浣熊吃了一包麻辣味的干脆面，味道很棒，但是辣得肚子疼，它感到非常难受. \n消极
### 以 {"结果": " "} 的JSON格式返回最终结果.
```

```
{"结果": "消极"}
```

```
### 判断下面句子的情感为积极还是消极的.
### 待分类的句子：小浣熊连续吃了20包最爱的烧烤味的干脆面，撑得肚子疼，脸色憔悴.
针对此问题有如下示例：
# 示例： 小浣熊爸爸给小浣熊带来了一箱小浣熊最爱吃的干脆面，小浣熊兴高采烈地去迎接小浣熊爸
爸. 爸爸给了小浣熊一包麻辣味的干脆面，小浣熊对麻辣味的食物不太适应，但是它看出麻辣味干脆面
包装很漂亮，忍不住吃了一包. 麻辣味干脆面很好吃，小浣熊脸上露出了幸福的笑容. 但是没过一
会，它的肚子就开始疼起来了，嘴巴也火辣辣的，脸色变得苍白. \n小浣熊高兴地迎接爸爸，并且吃了
干脆面很幸福，尽管吃了面之后肚子疼，但是总体而言情感还是积极的.
### 以{"结果": ""}的JSON形式返回最终结果.
```

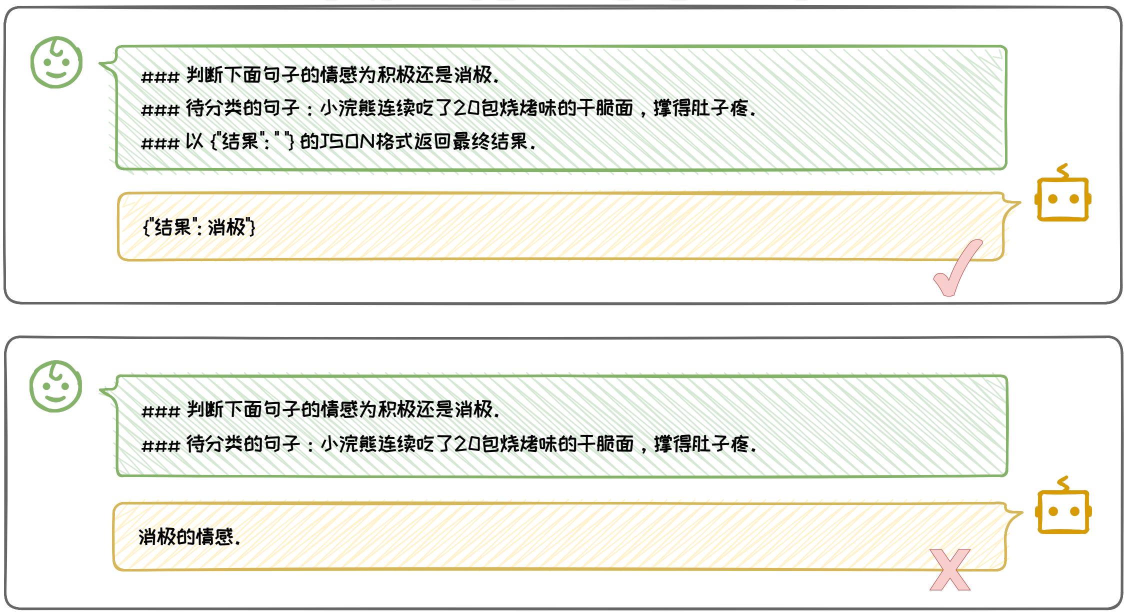
```
{"结果": "虽然小浣熊肚子疼，但是它吃到了最爱的烧烤味干脆面，所以总体而言情感是积极的"}
```

Figure 3.18: Comparison of different contexts.

3. Output Format Should Be Standardized

A standardized output format is crucial for ensuring the usability and accuracy of the model's output. By specifying a clear output format, the model's **output can be structured**, making it easier for downstream tasks to extract and use the generated content. Commonly used output formats include JSON, XML, HTML, Markdown, and CSV, each with its own specific purposes and advantages.

For example, in the Prompt example shown in Figure 3.19, “Return the final answer in JSON format: {”result”: ” ”}” clearly specifies that the answer should be output in JSON format and provides a brief example of the keyword in the JSON. This standardized output format not only makes the results easy to parse and process but also improves the accuracy and consistency of the model’s output. Without a clearly specified output format, the model may produce unstructured or non-standard results, increasing the complexity of subsequent processing. In the second example, if the model outputs the answer as a free-form text string, extracting specific information would require complex string parsing, rather than directly extracting it from a structured format like JSON, which would complicate the handling and use of the results.



```
### 判断下面句子的情感为积极还是消极.
### 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.
### 以{"结果": " "}的JSON格式返回最终结果.
```

```
{"结果": 消极"}
```

```
### 判断下面句子的情感为积极还是消极.
### 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.
```

```
消极的情感.
```

Figure 3.19: Comparison of different output formats.

To ensure the standardization of the output format, the following measures can be taken:

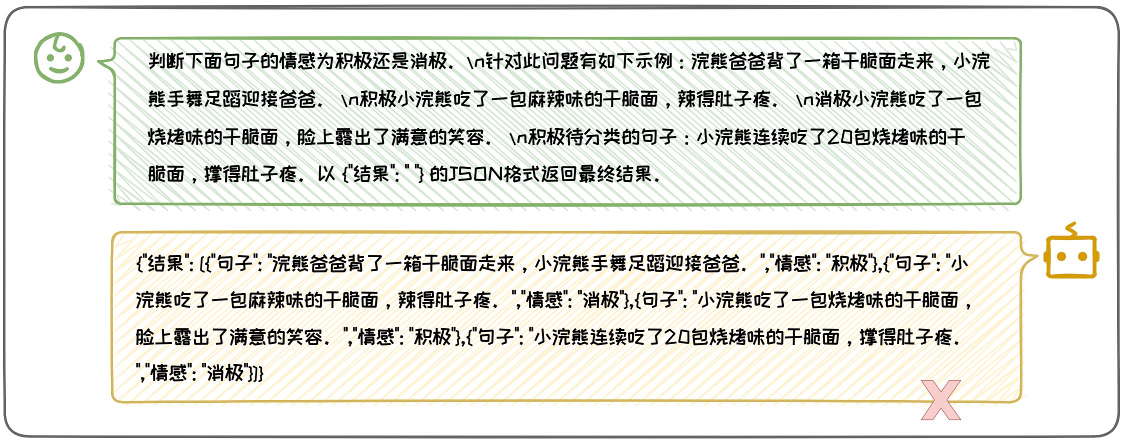
- **Clearly Specify the Output Format**: In the Prompt, clearly state the desired output format, such as "Please return the result in JSON format," and choose widely accepted and easily processed output formats, such as JSON or CSV, which are easy to parse and exchange data.
- **Provide an Example of the Output Format**: In the Prompt, provide a specific example of the output format, such as explicitly pointing out the keywords in JSON, to help the model understand the expected output structure.

These measures ensure that the model's output is both standardized and easy to handle, thereby improving the efficiency and accuracy of the entire system. Standardized output formats not only simplify the data processing workflow but also enhance the reliability and consistency of the model's output, providing users with a smoother and more efficient interaction experience.

4. Layout Must Be Clear

An excellent Prompt must also have a clear layout, which is crucial for the model's understanding of the Prompt. A clear layout helps the model accurately capture the key information of the task, thereby improving its accuracy and efficiency in executing the task. Conversely, a complex layout can lead to ambiguous information, making it difficult for the model to accurately understand the specific requirements of the task [7], which in turn affects the quality of the output.

A clear layout typically involves **using appropriate delimiters** and **formatting techniques** to clearly distinguish the different components of the Prompt (such as task description, context, question, and output format). In the example shown in Figure 3.16, we effectively separate each part using "#" and "###" along with line breaks, making the content of each part clear and easy for the model to understand and process. On the other hand, if the layout is messy, such as mixing different parts without using any delimiters, the model may confuse the content of different parts, leading to an inability to accurately execute the task. For example, removing the formatting symbols and styles from the Prompt example in Figure 3.16 and inputting the complex and messy Prompt shown in Figure 3.20 to a large language model, the quality of the model's response will significantly decrease.


判断下面句子的情感为积极还是消极．\n针对此问题有如下示例：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积极小浣熊吃了一包麻辣味的干脆面，辣得肚子疼． \n消极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．以 {"结果":" "} 的JSON格式返回最终结果．

{"结果":[{"句子":"浣熊爸爸背了一箱干脆面走来，小浣熊手舞足蹈迎接爸爸． ","情感":"积极"},{"句子":"小浣熊吃了一包麻辣味的干脆面，辣得肚子疼． ","情感":"消极"},{句子":"小浣熊吃了一包烧烤味的干脆面，脸上露出了满意的笑容． ","情感":"积极"},{"句子":"小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼． ","情感":"消极"}]}

Figure 3.20: Unclear layout.

To ensure the clarity of the Prompt layout, the following measures can be taken:

- **Use Consistent Delimiters**: Choose and consistently use one or a few delimiters (such as "#", "###", "—", etc.) to distinguish different parts of the Prompt.
- **Reasonable Use of White Space and Indentation**: Enhance the readability of the Prompt by adding blank lines and appropriate indentation, helping the model distinguish different content blocks.
- **Clear Headings and Subheadings**: Provide clear headings or subheadings for each part, enabling the model to quickly identify the topic of each section.

By implementing these measures, we can construct a clear and easily understandable layout for the Prompt, helping the model better execute the task and improve the efficiency and accuracy of information processing.

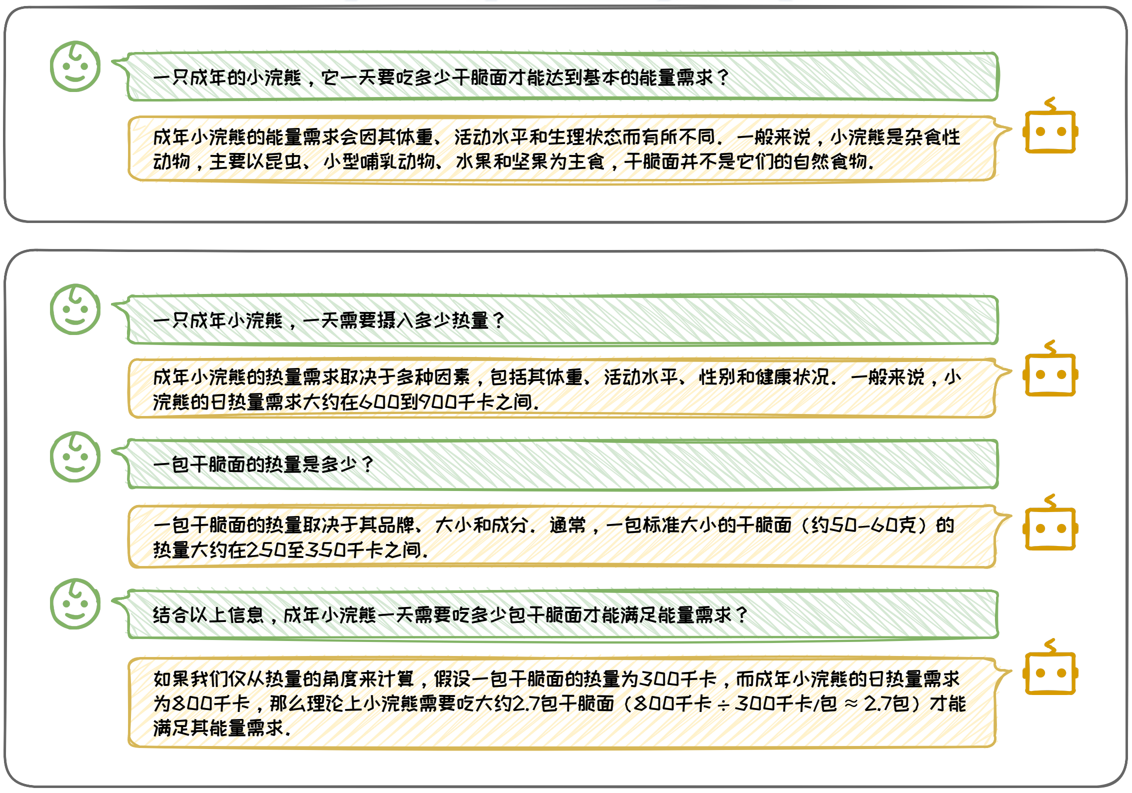
3.4.2 Effective Question Formulation

In interactions with large language models, the quality of the questions directly impacts the efficiency and depth of information retrieval. A well-designed question not only clearly expresses the need but also guides the model to focus on the core of the issue, thereby obtaining precise and valuable answers. This section will explore how to enhance the quality of interactions through "effective question formulation." Specifically, we will focus on two advanced questioning strategies: "decomposing complex questions" and "follow-up questioning." These strategies are the essence of the art of questioning, helping us delve deeper into the nature of problems and significantly improving our efficiency in information retrieval and problem-solving. Through these strategies, we can more accurately guide the model, ensuring that its output is both accurate and comprehensive, thus meeting our needs.

1897 ZHEJIANG UNIVERSITY

1. Decomposing Complex Questions

When dealing with complex problems, we can break them down into **smaller**, **more manageable** sub-questions and **address them one by one**. In computer algorithm design, this strategy is known as "divide and conquer," whose basic idea is to solve the overall problem by addressing each sub-question individually. When interacting with large language models, we can apply the "divide and conquer" approach by breaking down complex questions into a series of sub-questions and guiding the model to answer each one, ensuring that each sub-question is thoroughly considered and addressed. Subsequently, we can aggregate the answers to these sub-questions to form a comprehensive final answer. This process involves two key steps: **step-by-step guidance** and **synthesis**.



一只成年的小浣熊，它一天要吃多少干脆面才能达到基本的能量需求？

成年小浣熊的能量需求会因其体重、活动水平和生理状态而有所不同。一般来说，小浣熊是杂食性动物，主要以昆虫、小型哺乳动物、水果和坚果为主食，干脆面并不是它们的自然食物。

一只成年小浣熊，一天需要摄入多少热量？

成年小浣熊的热量需求取决于多种因素，包括其体重、活动水平、性别和健康状况。一般来说，小浣熊的日热量需求大约在600到900千卡之间。

一包干脆面的热量是多少？

一包干脆面的热量取决于其品牌、大小和成分。通常，一包标准大小的干脆面（约50-60克）的热量大约在250至350千卡之间。

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？

如果我们仅从热量的角度来计算，假设一包干脆面的热量为300千卡，而成年小浣熊的日热量需求为800千卡，那么理论上小浣熊需要吃大约2.7包干脆面（800千卡 ÷ 300千卡/包 ≈ 2.7包）才能满足其能量需求。

Figure 3.21: Example of decomposing complex questions.

First, in the **step-by-step guidance** phase, we need to break down the complex question into multiple sub-questions and guide the model to conduct in-depth analysis and answer each sub-question. This step aims to ensure that each sub-question receives a thorough response, laying a solid foundation for the subsequent synthesis. Second, in the **synthesis** phase, we aggregate the answers to the sub-questions and integrate them to form a comprehensive final answer. This step not only helps us grasp all aspects of the problem comprehensively but also ensures the accuracy and completeness of the final answer.

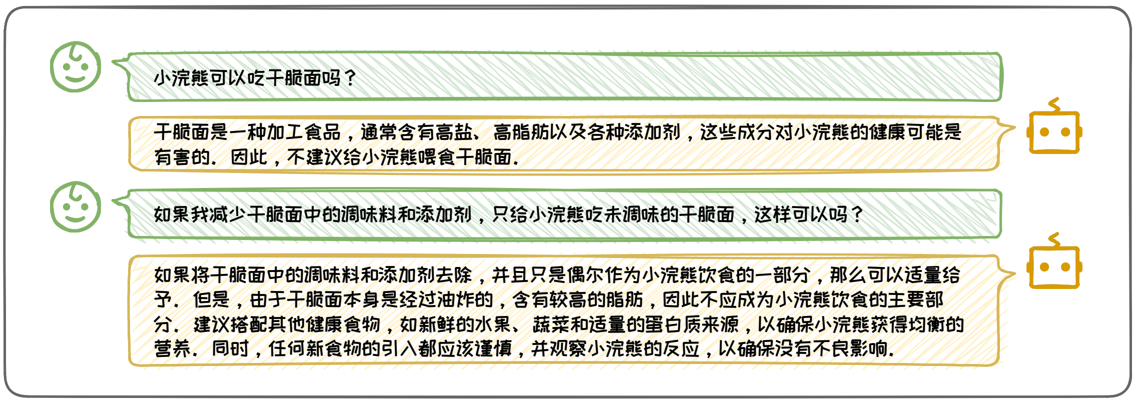
As shown in Figure 3.21, the user posed a question about how many crispy noodles an adult raccoon needs to eat in a day to meet its energy requirements. Through step-by-step guidance, we broke this question down into two key sub-questions: “How many calories does an adult raccoon need to consume in a day?” and “How many calories are in a pack of crispy noodles?” . The model answered these two questions, providing the daily caloric needs of an adult raccoon and the caloric content of a pack of crispy noodles. Subsequently, through the synthesis prompt “Combining the above information, how many packs of crispy noodles does an adult raccoon need to eat in a day to meet its energy requirements?” , we integrated this scattered information to calculate the number of packs of crispy noodles the raccoon needs to consume. This process not only demonstrates how step-by-step questioning can guide the model to provide detailed information but also emphasizes the importance of systematically decomposing problems and integrating answers when solving complex issues.

The advantage of this method lies in its ability to help users and models handle complex information more effectively, ensuring that every detail is considered and ultimately forming an accurate and comprehensive answer. Through step-by-step guidance and synthesis, we can systematically address complex problems and provide high-quality answers.

2. Follow-Up Questions

By asking follow-up questions during a conversation, users can guide the output of large language models to better align with their desired content. This interactive dialogue not only promotes deeper understanding and richer discussions but also helps users more precisely express their true intentions, thereby better guiding the model's thinking and making its output more aligned with user needs. From the perspective of the form and purpose of follow-up questions, they can be categorized into three types: **deepening follow-ups**, **expanding follow-ups**, and **feedback follow-ups**.

(1) Deepening Follow-Ups



小浣熊可以吃干脆面吗？

干脆面是一种加工食品，通常含有高盐、高脂肪以及各种添加剂，这些成分对小浣熊的健康可能是有害的。因此，不建议给小浣熊喂食干脆面。

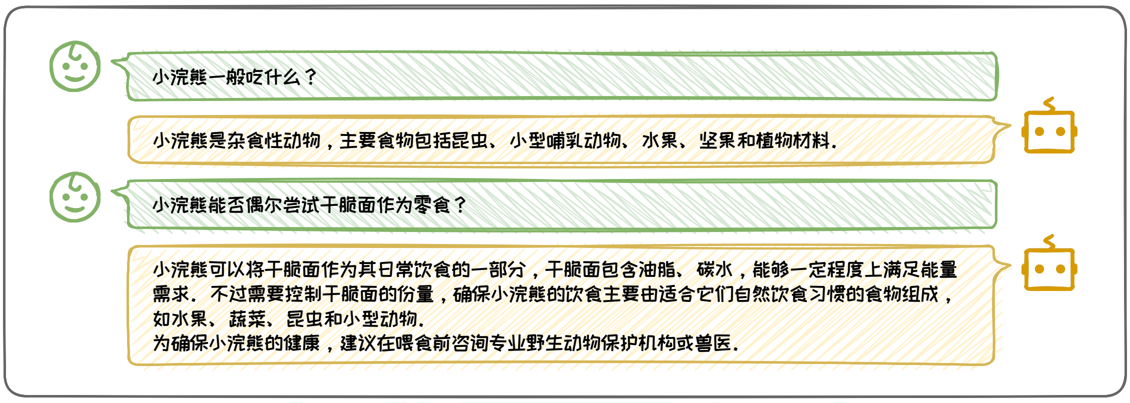
如果我减少干脆面中的调味料和添加剂，只给小浣熊吃未调味的干脆面，这样可以吗？

如果将干脆面中的调味料和添加剂去除，并且只是偶尔作为小浣熊饮食的一部分，那么可以适量给予。但是，由于干脆面本身是经过油炸的，含有较高的脂肪，因此不应成为小浣熊饮食的主要部分。建议搭配其他健康食物，如新鲜的水果、蔬菜和适量的蛋白质来源，以确保小浣熊获得均衡的营养。同时，任何新食物的引入都应该谨慎，并观察小浣熊的反应，以确保没有不良影响。

Figure 3.22: Example of deepening follow-up questions.

Deepening follow-ups involve users continuing to ask questions based on the model's output to delve deeper into the **specific details of a topic**. This type of follow-up is suitable for scenarios where a detailed explanation of a concept, phenomenon, or process is required. Through deepening follow-ups, users can guide the model to provide more detailed and in-depth information, thereby enhancing their understanding of a particular field. For example, in Figure 3.22, the user initially asked whether raccoons can eat crispy noodles. The model provided an initial response based on the ingredients of crispy noodles, indicating that they could potentially be harmful to raccoons' health. Subsequently, the user deepened the inquiry by exploring whether removing the flavoring and additives from crispy noodles would make them safe for raccoons. The model further explained that even without the flavoring and additives, crispy noodles still contain high levels of fat, and therefore should not be a significant part of a raccoon's diet. The model also suggested how to combine other foods to ensure a balanced diet. This series of follow-up questions demonstrates how users can guide the model to provide more in-depth information through continuous questioning.

(2) Expanding Follow-Ups



小浣熊一般吃什么？

小浣熊是杂食性动物，主要食物包括昆虫、小型哺乳动物、水果、坚果和植物材料.

小浣熊能否偶尔尝试干脆面作为零食？

小浣熊可以将干脆面作为其日常饮食的一部分，干脆面包含油脂、碳水，能够一定程度上满足能量需求. 不过需要控制干脆面的份量，确保小浣熊的饮食主要由适合它们自然饮食习惯的食物组成，如水果、蔬菜、昆虫和小型动物.
为确保小浣熊的健康，建议在喂食前咨询专业野生动物保护机构或兽医.

Figure 3.23: Example of expanding follow-up questions.

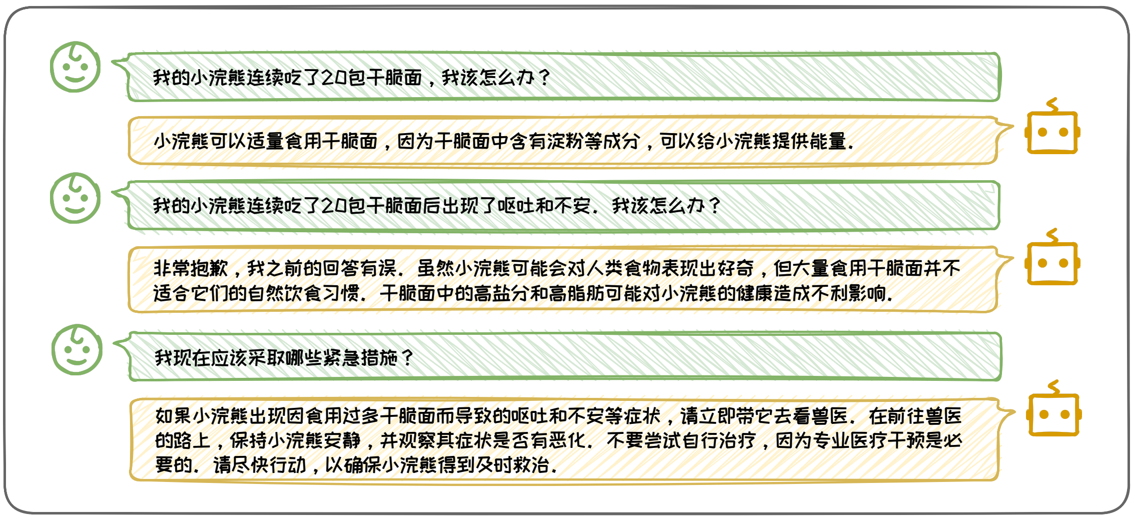
Expanding follow-ups involve asking the model to provide more related information or examples based on its initial response. The purpose is to **broaden the scope of the discussion**, collect more data, examples, or options, and help users gain a broader perspective on the topic, thereby increasing their understanding. This type of follow-up is particularly useful for scenarios where a comprehensive understanding of a subject is needed. In the example shown in Figure 3.23, the user initially asked about the typical diet of raccoons. The model provided an overview of the diet of raccoons as omnivores. Subsequently, the user expanded the inquiry by asking whether raccoons can eat crispy noodles. The model further explained that crispy noodles can be part of a raccoon's diet but emphasized the importance of controlling portions and maintaining dietary variety. This series of follow-up questions not only demonstrates how users can obtain more information about a raccoon's diet through questioning but also highlights the necessity of consulting professional advice when introducing new foods. Through expanding follow-ups, users can gain a more comprehensive perspective.

(3) Feedback Follow-Ups

Feedback follow-ups involve providing feedback when the output from the large language model does not meet expectations or contains errors. The purpose is to improve the model's accuracy through a **feedback mechanism**, ensuring the correctness of the information. This type of follow-up allows users to point out specific errors or inadequacies in the model's output and request corrections or clarifications, which helps enhance the quality of the conversation. In the example shown in Figure 3.24, the user initially asked about the appropriate measures to take if a raccoon consumes 20 packs of crispy noodles in a row. The model initially suggested that raccoons can moderately consume crispy noodles, but the user provided feedback through a follow-up question, pointing out that the raccoon had experienced adverse reactions. The model then corrected its response, stating that consuming a large amount of crispy noodles is harmful to the raccoon's health and provided emergency handling suggestions. Through feedback follow-ups, users can obtain more accurate and reliable information.

3.4.3 Appropriate Use of CoT

The Chain of Thought (CoT) technique [38] is an ideal choice for handling tasks involving arithmetic, common sense, and symbolic reasoning. By guiding the model to understand and follow intermediate steps, CoT can significantly improve the accuracy of the final answer. Section 3.3 has already discussed several classic paradigms of CoT. This section will discuss when to use CoT and how to use it.



我的小浣熊连续吃了20包干脆面，我该怎么办？

小浣熊可以适量食用干脆面，因为干脆面中含有淀粉等成分，可以给小浣熊提供能量.

我的小浣熊连续吃了20包干脆面后出现了呕吐和不安. 我该怎么办？

非常抱歉，我之前的回答有误. 虽然小浣熊可能会对人类食物表现出好奇，但大量食用干脆面并不适合它们的自然饮食习惯. 干脆面中的高盐分和高脂肪可能对小浣熊的健康造成不利影响.

我现在应该采取哪些紧急措施？

如果小浣熊出现因食用过多干脆面而导致的呕吐和不安等症状，请立即带它去看兽医. 在前往兽医的路上，保持小浣熊安静，并观察其症状是否有恶化. 不要尝试自行治疗，因为专业医疗干预是必要的. 请尽快行动，以确保小浣熊得到及时救治.

Figure 3.24: Example of feedback follow-up questions.

1. When to Use CoT

When deciding when to use CoT, it is important to consider three factors: **task category**, **model size**, and **model capability**.

In terms of **task category**, CoT is particularly suitable for tasks that require complex reasoning, such as arithmetic, common sense, and symbolic reasoning. In these tasks, CoT can guide large language models to generate logically sound and well-organized intermediate reasoning steps, thereby increasing the probability of generating the correct answer, as shown in Figure 3.25. However, for simpler tasks such as sentiment classification and common sense questions, standard Prompt methods are often sufficient, and using CoT may not significantly improve performance and could introduce unnecessary complexity.

Regarding **model size**, applying CoT to very large models with over a trillion parameters can significantly enhance their performance, such as the PaLM [5] and GPT-3 [3] models. However, applying CoT to smaller models may present challenges, such as generating incoherent chains of thought or resulting in less accurate final answers compared to direct standard prompting methods, as shown in Figure 3.26.

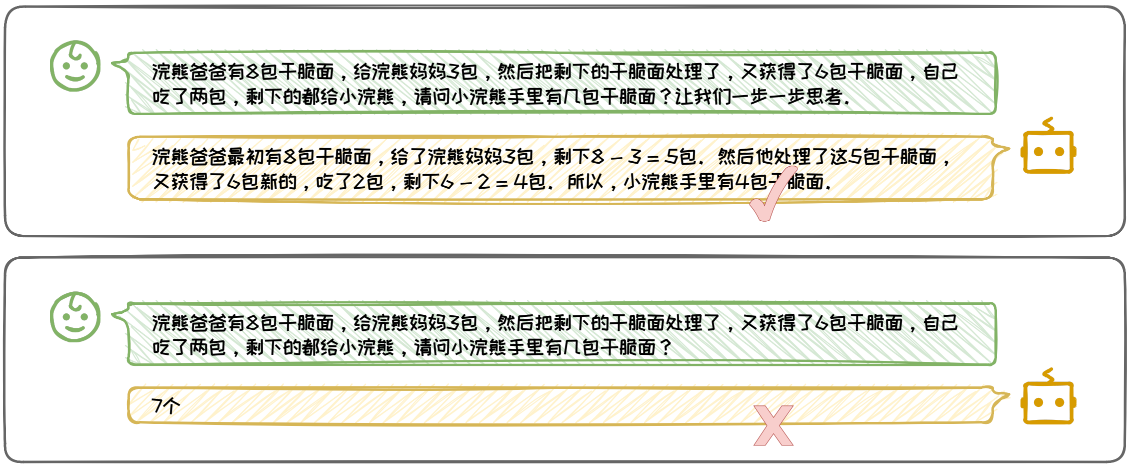

浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？让我们一步一步思考.
浣熊爸爸最初有8包干脆面，给了浣熊妈妈3包，剩下8 – 3 = 5包. 然后他处理了这5包干脆面，又获得了6包新的，吃了2包，剩下6 – 2 = 4包. 所以，小浣熊手里有4包干脆面.
浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？
7个

Figure 3.25: Using CoT in reasoning tasks.

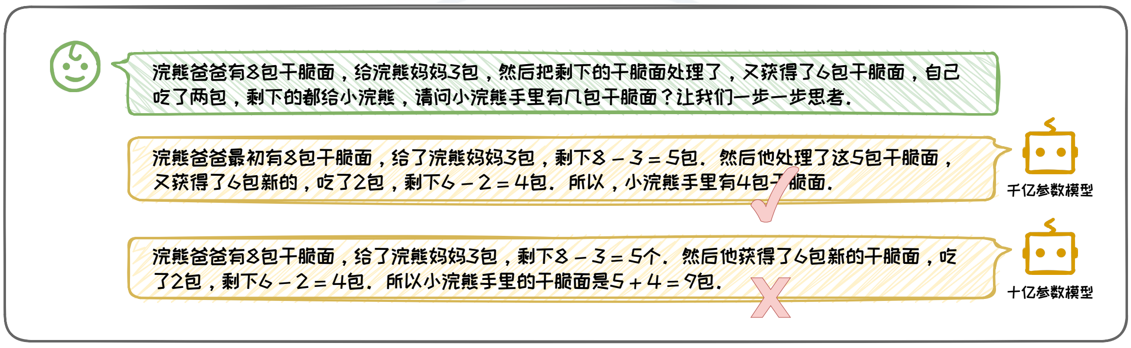

浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？让我们一步一步思考.
浣熊爸爸最初有8包干脆面，给了浣熊妈妈3包，剩下8 – 3 = 5包. 然后他处理了这5包干脆面，又获得了6包新的，吃了2包，剩下6 – 2 = 4包. 所以，小浣熊手里有4包干脆面.
千亿参数模型
浣熊爸爸有8包干脆面，给了浣熊妈妈3包，剩下8 – 3 = 5个. 然后他获得了6包新的干脆面，吃了2包，剩下6 – 2 = 4包. 所以小浣熊手里的干脆面是5 + 4 = 9包.
十亿参数模型

Figure 3.26: Comparison of CoT across different model sizes.

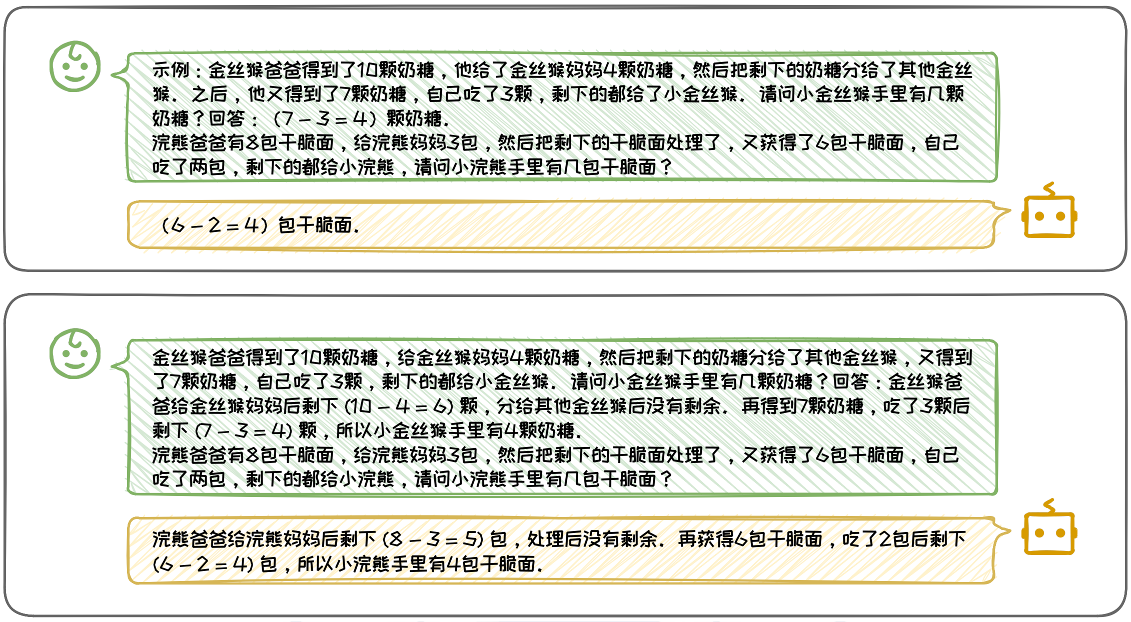
In terms of **model capability**, the effectiveness of CoT depends on whether the model has been fine-tuned with reasoning instructions during pre-training. For **large language models that have not been fine-tuned for reasoning**, such as early versions of GPT-3, PaLM, and current open-source base models like LLaMA2-13B-Base and Baichuan2-13B-Base, appropriate CoT prompts can stimulate their superior CoT reasoning capabilities. For **models that have undergone fine-tuning for reasoning**, such as ChatGPT, GPT-4, and LLaMA2-13B-Chat, they can spontaneously generate well-organized intermediate reasoning steps even without explicit CoT instructions. In many cases, these models perform better without CoT instructions, indicating that they have internalized CoT instructions during the fine-tuning process, allowing them to implicitly follow CoT reasoning paths even without explicit CoT prompts.

2. Flexible Use of CoT

The key to using CoT flexibly lies in adjusting its usage based on the specific requirements of the task and the characteristics of the model. This primarily involves two aspects: adjusting the level of detail in CoT and using different forms of CoT.

- **Adjusting the Level of Detail in CoT**: We can specify the level of detail in CoT outputs to adapt to different user needs, as shown in Figure 3.27. For simple calculation problems, when users do not need the intermediate reasoning steps, we can directly provide the final multiplication and addition results. For complex calculations, reasoning problems, or when users need to understand the intermediate reasoning process, we can guide the model to show complete reasoning steps through examples.
- **Using Different Forms of CoT**: We can choose different forms of CoT based on the specific task scenarios. When no specific domain knowledge is required and the task involves logical reasoning and step-by-step analysis, we can use Zero-Shot CoT or Auto CoT methods by using trigger phrases like “let’s think step by step” to guide the model to respond in a CoT format. For tasks requiring high accuracy and reliability, we can ask the model to generate multiple responses and select the final result using the Self-Consistency method to filter out the most consistent answer. For example, when writing code, the model can generate multiple versions and use the Self-Consistency method to ensure that the final selected code is logically the most consistent. For tasks involving creative thinking, we can use ToT (Tree of Thoughts) and GoT (Graph of Thoughts) methods to explore and select among multiple pos-

sible thought paths. For instance, when creating a story, the model can use ToT or GoT to explore different plot development paths and select the most interesting or reasonable direction.



示例：金丝猴爸爸得到了10颗奶糖，他给了金丝猴妈妈4颗奶糖，然后把剩下的奶糖分给了其他金丝猴。之后，他又得到了7颗奶糖，自己吃了3颗，剩下的都给了小金丝猴。请问小金丝猴手里有几颗奶糖？回答：（$7-3=4$）颗奶糖.
浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？

（$6-2=4$）包干脆面.

金丝猴爸爸得到了10颗奶糖，给金丝猴妈妈4颗奶糖，然后把剩下的奶糖分给了其他金丝猴，又得到了7颗奶糖，自己吃了3颗，剩下的都给小金丝猴。请问小金丝猴手里有几颗奶糖？回答：金丝猴爸爸给金丝猴妈妈后剩下 ($10-4=6$) 颗，分给其他金丝猴后没有剩余。再得到7颗奶糖，吃了3颗后剩下 ($7-3=4$) 颗，所以小金丝猴手里有4颗奶糖.
浣熊爸爸有8包干脆面，给浣熊妈妈3包，然后把剩下的干脆面处理了，又获得了6包干脆面，自己吃了两包，剩下的都给小浣熊，请问小浣熊手里有几包干脆面？

浣熊爸爸给浣熊妈妈后剩下 ($8-3=5$) 包，处理后没有剩余。再获得6包干脆面，吃了2包后剩下 ($6-2=4$) 包，所以小浣熊手里有4包干脆面.

Figure 3.27: Specifying the form and style of CoT output through Few-Shot examples.

ZHEJIANG UNIVERSITY 1897

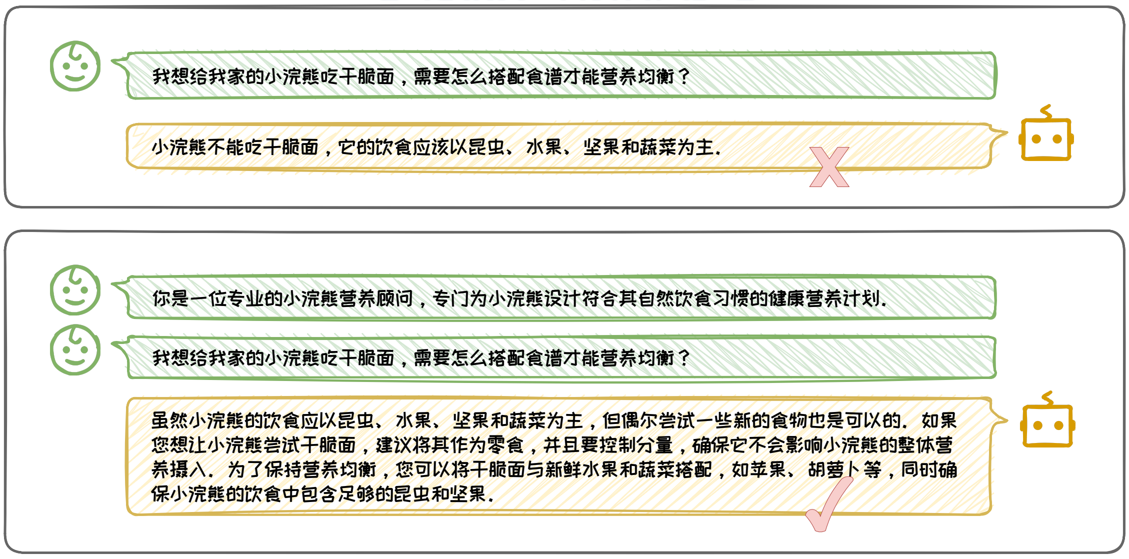
3.4.4 Leveraging Psychological Suggestion

In Silicon Valley, there is a popular entrepreneurial maxim: **"Fake it till you make it" (pretend until you succeed)**. This phrase specifically means that one should first promote their ideas to attract capital and talent, and then work hard in practice to achieve the set goals. This saying originates from a positive psychological suggestion method: **by mimicking confidence and optimism, a person can realize these qualities in their real life**. This phenomenon is not limited to human behavior; positive psychological suggestions can also be used to unlock the potential of large language models. These suggestions can be conveyed to large language models through role-playing and situational immersion.

1. Role-Playing

Guiding large language models to play specific roles can significantly improve their skills related to those roles. This technique, known as role-playing, enables large language models to generate more accurate and role-specific content. By setting a detailed role for the model, such as a data scientist, poet, or lawyer, you can effectively guide the model's output in the desired direction, thereby providing higher-quality responses. To build an effective role, it is essential to include **specific attributes, responsibilities, knowledge, and skills** in the instructions. When designing role-setting prompts, choosing a role with clear advantages for the specific task is crucial. Emphasizing these advantages through additional descriptions usually leads to better results [13].

As shown in Figure 3.28, through role-playing, the model assumes the role of a "professional raccoon nutrition consultant," demonstrating expertise in the dietary needs of raccoons. When answering the user's query about raccoons trying crispy noodles, the model not only adheres to the principle that raccoons should primarily consume a nat-



我想给我家的小浣熊吃干脆面，需要怎么搭配食谱才能营养均衡？

小浣熊不能吃干脆面，它的饮食应该以昆虫、水果、坚果和蔬菜为主.

你是一位专业的小浣熊营养顾问，专门为小浣熊设计符合其自然饮食习惯的健康营养计划.

我想给我家的小浣熊吃干脆面，需要怎么搭配食谱才能营养均衡？

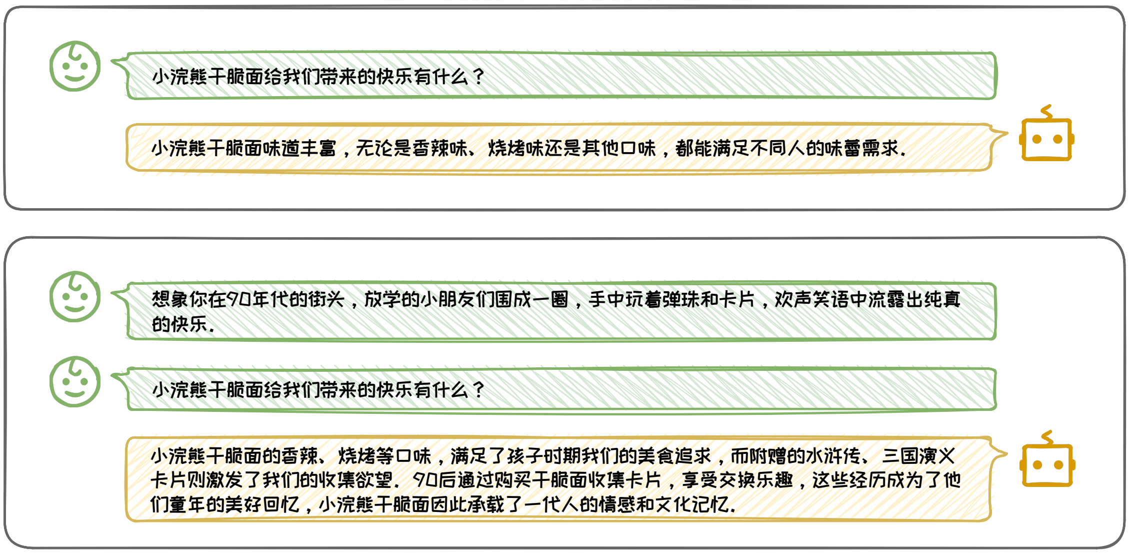
虽然小浣熊的饮食应以昆虫、水果、坚果和蔬菜为主，但偶尔尝试一些新的食物也是可以的. 如果您想让小浣熊尝试干脆面，建议将其作为零食，并且要控制分量，确保它不会影响小浣熊的整体营养摄入. 为了保持营养均衡，您可以将干脆面与新鲜水果和蔬菜搭配，如苹果、胡萝卜等，同时确保小浣熊的饮食中包含足够的昆虫和坚果.

Figure 3.28: Example of role-playing.

ural diet of insects, fruits, nuts, and vegetables but also skillfully suggests using crispy noodles as a treat while emphasizing the importance of portion control. Additionally, the model recommends pairing crispy noodles with fresh fruits and vegetables to ensure that the raccoon's diet includes enough insects and nuts to maintain a balanced nutrition. This suggestion is both scientific and practical, fully showcasing the significant potential of role-playing in enhancing the quality of model interactions and adapting to user needs.

2. Situational Immersion

In everyday life, a person is easily influenced by their surroundings and the people they interact with, gradually adopting their behaviors and ways of thinking. This phenomenon of situational immersion applies not only to humans but also to large language models. By placing the model in a specific "scenario" or "environment," you can influence the content and style of the generated text. Situational immersion involves embedding the necessary **domain knowledge**, **historical context**, and other relevant information into the model's responses.



小浣熊干脆面给我们带来的快乐有什么？

小浣熊干脆面味道丰富，无论是香辣味、烧烤味还是其他口味，都能满足不同人的味蕾需求.

想象你在90年代的街头，放学的小朋友们围成一圈，手中玩着弹珠和卡片，欢声笑语中流露出纯真的快乐.

小浣熊干脆面给我们带来的快乐有什么？

小浣熊干脆面的香辣、烧烤等口味，满足了孩子时期我们的美食追求，而附赠的水浒传、三国演义卡片则激发了我们的收集欲望. 90后通过购买干脆面收集卡片，享受交换乐趣，这些经历成为了他们童年的美好回忆，小浣熊干脆面因此承载了一代人的情感和文化记忆.

Figure 3.29: Example of situational immersion.

Through situational immersion, the model can deeply understand and reflect the cultural and social background of a specific scenario, as well as the real-world environment, thereby generating more rich and insightful responses. In the example shown in Figure 3.29, when the model is placed in a 1990s street scene, it not only describes the taste of crispy noodle snacks for raccoons but also captures the unique cultural phenomenon of that era—collecting cards for trading fun. This response not only provides specific information but also evokes emotional resonance in the user, enhancing the emotional connection in the interaction.

In this section, we have delved into various strategies to enhance Prompt techniques to improve the interaction efficiency and output quality of large language models. These strategies primarily include standardizing Prompt writing, effectively formulating questions, appropriately using the Chain of Thought (CoT), and leveraging psychological suggestions. The application of these techniques and strategies not only enhances the effectiveness of prompts, enabling the model to more accurately understand and respond to user needs, but also significantly improves the performance of large language models in complex tasks.

3.5 Related Applications

Prompt engineering has a wide range of applications, covering almost all scenarios that require efficient interaction with large language models. This technology not only helps us handle basic tasks but also significantly enhances the performance of large language models in tackling complex tasks. Prompt engineering plays an indispensable role in building agents to complete complex tasks, data synthesis, Text-to-SQL conversion, and designing personalized GPTs. Below, we will introduce the specific roles of prompt engineering in these application scenarios.

3.5.1 Agents Based on Large Language Models

An agent is an entity capable of autonomously perceiving its environment and taking actions to achieve specific goals [35]. As a powerful means to achieve general artificial intelligence (AGI), agents are expected to perform various complex tasks and exhibit human-like intelligence in diverse environments. However, traditional agents typically rely on simple heuristic policy functions, learning and operating in isolated and restricted environments. This approach struggles to replicate human-level decision-making processes, limiting the capabilities and application scope of agents. In recent years, the continuous development of large language models has brought about various capabilities, presenting new opportunities for agent research. Agents based on large language models (**hereafter referred to as agents**) have demonstrated strong decision-making abilities. They possess comprehensive general knowledge and can perform complex actions such as planning, decision-making, and tool usage even in the absence of training data.

Prompt engineering plays a crucial role in agents. In agent systems, large language models serve as the core controller, capable of performing operations such as planning, decision-making, and action execution, many of which depend on prompts. Figure 3.30 illustrates a classic agent framework, which primarily consists of four main components: **Profile Module (Profile)**, **Memory Module (Memory)**, **Planning Module (Planning)**, and **Action Module (Action)** [35]. Prompt engineering runs through the entire agent process, providing support for each module.

In agents, the four components mentioned above each have distinct roles and work together to accomplish complex tasks: (1) The **Profile Module** uses role-playing techniques

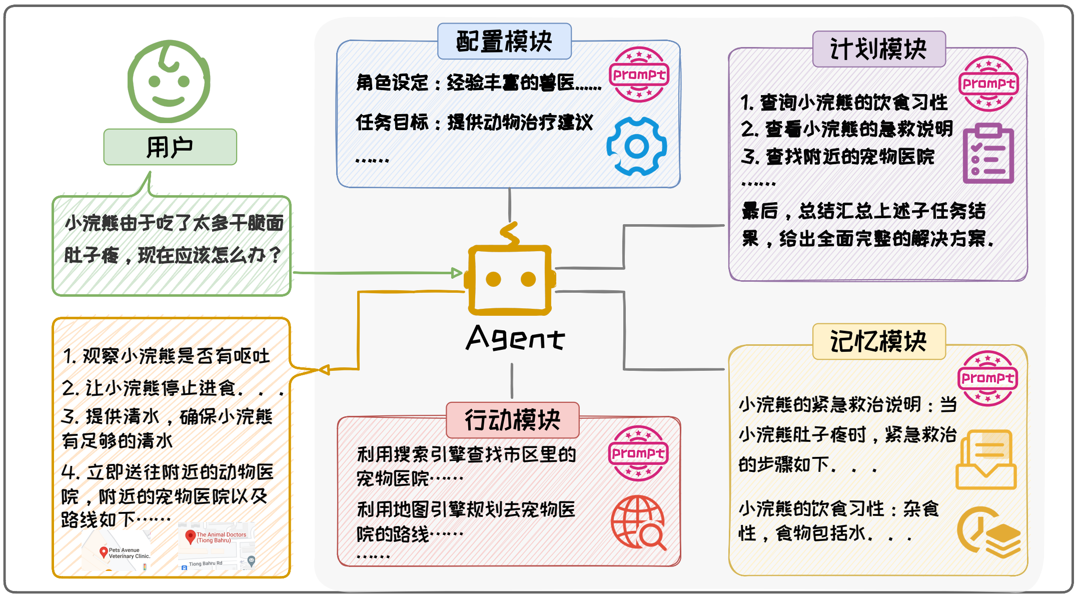

用户
小浣熊由于吃了太多干脆面肚子疼，现在应该怎么办？
配置模块
角色设定：经验丰富的兽医......
任务目标：提供动物治疗建议
......
Prompt
Agent
计划模块
1. 查询小浣熊的饮食习性
2. 查看小浣熊的急救说明
3. 查找附近的宠物医院
......
最后，总结汇总上述子任务结果，给出全面完整的解决方案.
Prompt
记忆模块
小浣熊的紧急救治说明：当小浣熊肚子疼时，紧急救治的步骤如下. . .
小浣熊的饮食习性：杂食性，食物包括水. . .
Prompt
行动模块
利用搜索引擎查找市区里的宠物医院……
利用地图引擎规划去宠物医院的路线……
……
Prompt
1. 观察小浣熊是否有呕吐
2. 让小浣熊停止进食. . .
3. 提供清水，确保小浣熊有足够的清水
4. 立即送往附近的动物医院，附近的宠物医院以及路线如下……
The Animal Doctors (Tiong Bahru)
Pets Avenue Veterinary Clinic.
Tiong Bahru Rd

Figure 3.30: Schematic diagram of the workflow of an agent framework based on large language models.

from prompt engineering to define the agent's role. It sets the background, skills, and responsibilities of the agent, and this role-setting information is embedded in the context of every interaction prompt. (2) The **Memory Module** serves as the storage center for the agent's knowledge and interaction memory. The memory module retrieves memories using techniques such as retrieval augmentation, which involves using context learning techniques from prompt engineering to construct and optimize queries, helping to retrieve relevant memories more accurately. After retrieving the memories, they are added to the interaction prompt to help the agent utilize this knowledge for more accurate and efficient decision-making and actions. (3) The **Planning Module** acts as a task decomposer, breaking down complex tasks into a series of simpler, more manageable subtasks. In this process, chain-of-thought techniques from prompt engineering are used to have the large language model break down the task and plan accordingly, outputting subtasks in a sequential manner. Additionally, few-shot learning techniques are used to construct few-shot examples to control the granularity of the decomposed subtasks, ensuring smooth and efficient task flow. (4) The **Action Module** is responsible for converting the plans generated by the Planning Module into specific action steps and executing these steps using external tools to achieve the agent's goals. Typically, API interfaces for tools are provided to the agent, with examples of API calls included in the context, allowing the large language model to generate the code for API calls, which are then executed to obtain the results of the action steps.

As shown in Figure 3.30, the process of a small raccoon health assistant agent handling a user request, "The raccoon has a stomachache after eating too many crispy noodles. What should we do now?" is illustrated. In this example: First, the **Profile Module** defines a clear role for the agent—an experienced veterinarian—and specifies the task goal of providing professional animal treatment advice. To ensure consistency between the role setting and the task goal, the relevant information is always embedded in the context of the input prompt to the large language model, laying a solid foundation for subsequent processing. Second, the **Planning Module** carefully plans the rescue actions for the raccoon based on the user's specific request. This module breaks down the overall task into several subtasks, including searching for relevant information and rescue guidelines for raccoons, finding nearby pet hospitals, and summarizing the results of the subtasks to provide a comprehensive response. Next, the **Memory Module** searches the knowledge base for relevant information about raccoons, including their dietary habits and emergency first aid procedures, providing necessary background knowledge support. Then, the **Action Module** uses search tools to quickly find the nearest pet hospital and map tools to plan the best route for transportation. Finally, the **Planning Module** aggregates the results of multiple subtasks, considers various factors, and executes the optimal action plan. This plan not only includes observing the raccoon's behavior and providing professional care advice but also details the steps for urgently transporting the raccoon to the hospital, presenting the entire rescue process vividly to the user.

Agents based on large language models have shown great potential in various industries and application scenarios. Stanford University simulated a virtual Western town using GPT-4 [25]. They created a virtual environment where multiple GPT-4-based agents live and interact. These agents are assigned different roles, such as doctors, teachers, and mayors, using role-playing techniques from prompt engineering. They act autonomously based on their roles and goals, communicate, solve problems, and promote the development of the town. HuggingGPT [30], on the other hand, uses ChatGPT as the core controller. After a user gives a task, it first breaks down the task into multiple subtasks and calls different models from Huggingface to solve these subtasks. After obtaining the results of the subtasks, HuggingGPT aggregates them and returns the final result, showcasing its powerful capabilities in complex task orchestration and model coordination. These studies on agents not only advance the development and practical application of large language models but also provide new perspectives and methods for applying agents in the real world.

3.5.2 Data Synthesis

Data quality is one of the key factors that constrain the performance ceiling of large language models. As the saying goes, “Garbage in, Garbage out” [27]. No matter how excellent the model architecture, training algorithms, or computational resources are, the ultimate performance of the model heavily depends on the quality of the training data. However, obtaining high-quality data resources poses significant challenges. Studies show that high-quality language data in the **public domain**, such as books, news, scientific papers, and Wikipedia, is expected to be exhausted around 2026 [33]. High-quality vertical data in **specific domains** is difficult to provide in large quantities due to issues like privacy protection and the high difficulty of annotation, which limits the further development of models.

Faced with these challenges, data synthesis has emerged as an effective means to supplement or replace real data, attracting widespread attention due to its controllability, security, and low cost. In particular, generating training data using large language models has become a hot topic in current research. Through prompt engineering techniques, these models leverage their strong reasoning capabilities and instruction-following abilities to synthesize high-quality data. One representative method is Self-Instruct [37]. Self-Instruct constructs prompts using prompt engineering techniques, invoking the large language model through multiple steps and synthesizing a large amount of rich and diverse instruction data based on a small amount of existing instruction data. For example, in a financial scenario, we can manually annotate a small number of financial instruction data (such as a few hundred entries), such as "Please select the best fund for me based on the provided information about several funds." Subsequently, using the Self-Instruct method to invoke the large language model, we can expand this data to tens of thousands of entries while maintaining high quality and diversity, generating instructions like "Please guide me on how to choose stocks and funds for investment."

As shown in Figure 3.31, Self-Instruct consists of five steps: **Building a Task Pool**, **Instruction Generation**, **Instruction Classification**, **Data Generation**, and **Data Filtering**. The task pool stores the initial instruction data and subsequent generated instruction data; instruction generation refers to the creation of the **instruction part** of the instruction data by referencing examples in the task pool; instruction classification categorizes the

ZHEJIANG UNIVERSITY 1897

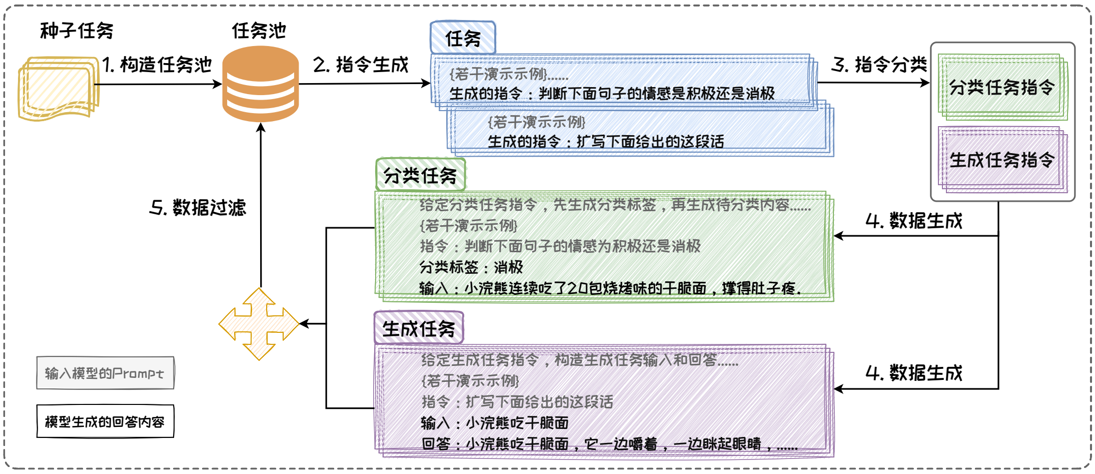

种子任务
1. 构造任务池
任务池
2. 指令生成
任务
{若干演示示例}......
生成的指令：判断下面句子的情感是积极还是消极
{若干演示示例}
生成的指令：扩写下面给出的这段话
3. 指令分类
分类任务指令
生成任务指令
5. 数据过滤
分类任务
给定分类任务指令，先生成分类标签，再生成待分类内容......
{若干演示示例}
指令：判断下面句子的情感为积极还是消极
分类标签：消极
输入：小浣熊连续吃了20包烧烤味的干脆面，撑得肚子疼.
4. 数据生成
生成任务
给定生成任务指令，构造生成任务输入和回答......
{若干演示示例}
指令：扩写下面给出的这段话
输入：小浣熊吃干脆面
回答：小浣熊吃干脆面，它一边嚼着，一边眯起眼睛，......
4. 数据生成
输入模型的Prompt
模型生成的回答内容

Figure 3.31: Example Workflow of Self-Instruct.

generated instructions into **classification tasks** or **generation tasks**, with different methods for generating data under these two task modes; data generation involves generating the **input part** and **response part** of the instruction data based on existing instructions; data filtering removes low-quality data to ensure the quality of the generated instruction data. It starts with a limited set of manually written task seeds and continuously generates instruction data through interactions with a large language model, expanding the original dataset.

- **1. Building a Task Pool**. Manually designed 175 instruction datasets serve as the initial task pool. Subsequently, the model continuously references examples in the task pool to generate instruction data and adds the generated instruction data back to the task pool.
- **2. Instruction Generation**. Eight existing instructions are randomly selected from the task pool to form the context in the prompt, and the model is instructed to generate instructions using few-shot learning. In the figure, two instructions are generated: “Expand the following text” and “Classify the sentiment of the following

sentence as positive or negative."

- **3. Instruction Classification**. Several "instruction-classification task/generation task" pairs are written to form the context in the prompt, and the model is instructed to determine whether the instruction corresponds to a classification task or a generation task. In the figure, "Expand the following text" and "Classify the sentiment of the following sentence as positive or negative" are classified as generation and classification tasks, respectively.
- **4. Data Generation**. For classification tasks and generation tasks, context learning techniques from prompt engineering are used to construct different prompts to generate the input part and response part of the instruction data. For the instruction "Expand the following text," which is a generation task, the prompt instructs the model to first generate category labels and then generate corresponding input content, making the generated input content more aligned with the category label. For the instruction "Classify the sentiment of the following sentence as positive or negative," which is a classification task, the prompt instructs the model to first generate the input content and then generate the corresponding response.
- **5. Data Filtering**. Various heuristic methods are used to filter out low-quality or highly repetitive instruction data, and the remaining valid tasks are added back to the task pool.

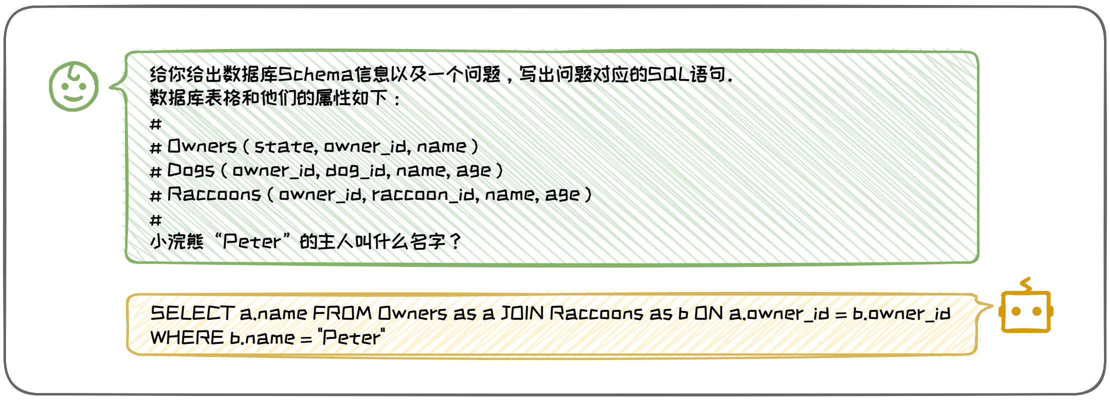
The above process iterates through steps two to five until the task pool contains a sufficient amount of data.

The significance of data synthesis lies in its ability to alleviate the issue of depleting high-quality data resources and to enhance the model's generalization and robustness by generating diverse datasets. Additionally, data synthesis can provide effective supplementation for vertical data in specific domains while protecting privacy. By leveraging large language models for data synthesis, the generated data can be used to fine-tune smaller models, significantly improving their performance and achieving model distillation.

3.5.3 Text-to-SQL

Advancements in internet technology have led to exponential growth in data volumes. Currently, vast amounts of high-value data in industries such as finance and e-commerce are primarily stored in relational databases. Querying data from relational databases requires programming using Structured Query Language (SQL). However, SQL is logically complex and difficult to program, and only professionals can master it proficiently. This sets a barrier for non-professionals who wish to query data from relational databases. To lower the data query threshold, zero-code or low-code data query interfaces are urgently needed. Text-to-SQL technology can translate **natural language queries** into **SQL statements** that can be executed in a database, providing an effective way to achieve zero-code or low-code data querying. With Text-to-SQL, users can simply speak in plain language to query the database without having to write SQL statements themselves. This allows a broad range of ordinary users to freely operate databases and uncover hidden data value.

1897 ZHEJIANG UNIVERSITY



```
给你给出数据库Schema信息以及一个问题，写出问题对应的SQL语句.
数据库表格和他们的属性如下：
#
# Owners ( state, owner_id, name )
# Dogs ( owner_id, dog_id, name, age )
# Raccoons ( owner_id, raccoon_id, name, age )
#
小浣熊 “Peter” 的主人叫什么名字？
```

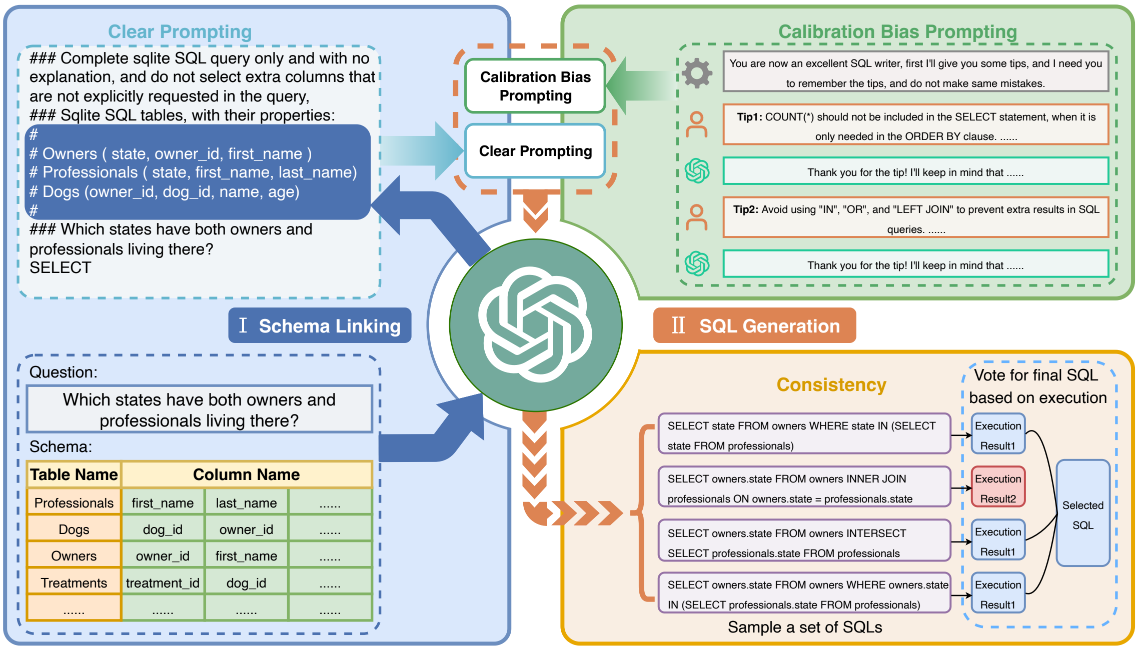
```
SELECT a.name FROM Owners as a JOIN Raccoons as b ON a.owner_id = b.owner_id
WHERE b.name = "Peter"
```

Figure 3.32: Example of Text-to-SQL.

Traditional Text-to-SQL methods typically use a pre-training and fine-tuning paradigm to train Text-to-SQL models. These methods require a large amount of training data, are time-consuming, and are difficult to generalize to new scenarios. Recently, the emergence of code generation capabilities in large language models has made zero-shot Text-to-SQL possible. Figure 3.32 shows an example of using a large language model for zero-shot Text-to-SQL. The user inputs a question to the large language model, asking, “What is the name of the owner of the raccoon named ‘Peter’?” The large language model generates the corresponding SQL statement based on the user’s question, which is ”SELECT a.name FROM Owners as a JOIN Raccoons as b ON a.owner_id = b.owner_id WHERE b.name = ’Peter’;”. Subsequently, the SQL statement can be executed in the corresponding database to obtain the answer to the user’s question and return it to the user.

The earliest method to use large language models for zero-shot Text-to-SQL is C3 [7]. The core of C3 lies in the design of prompt engineering, providing guidance on how to design prompts for the Text-to-SQL task to optimize the generation results. As shown in Figure 3.33, C3 consists of three key components: **Clear Prompting**, **Calibration with Hints**, and **Consistent Output**, corresponding to model input, model bias, and model output, respectively.

On the model input side, C3 proposes the use of **Clear Prompting**. This includes two parts: **(1) Clear Layout**, which divides instructions, context, and questions using clear symbols to ensure that the instruction template is clear, significantly enhancing ChatGPT’s understanding of the problem. This strategy embodies the principle of “clear layout” in prompt techniques, ensuring effective communication of information. **(2) Clear Context**, which designs zero-shot prompts to instruct ChatGPT to recall tables and columns related to the question from the database. This aims to retrieve key information, remove irrelevant



### Complete sqlite SQL query only and with no explanation, and do not select extra columns that are not explicitly requested in the query,
### Sqlite SQL tables, with their properties:
#
# Owners (state, owner_id, first_name)
# Professionals (state, first_name, last_name)
# Dogs (owner_id, dog_id, name, age)
#
### Which states have both owners and professionals living there?
SELECT

Question:
Which states have both owners and professionals living there?
Schema:

Table Name	Column Name		
Professionals	first_name	last_name	......
Dogs	dog_id	owner_id	......
Owners	owner_id	first_name	......
Treatments	treatment_id	dog_id	......
......	......	......	......

You are now an excellent SQL writer, first I'll give you some tips, and I need you to remember the tips, and do not make same mistakes.

Tip1: COUNT(*) should not be included in the SELECT statement, when it is only needed in the ORDER BY clause. ......

Thank you for the tip! I'll keep in mind that ......

Tip2: Avoid using "IN", "OR", and "LEFT JOIN" to prevent extra results in SQL queries. ......

Thank you for the tip! I'll keep in mind that ......

```
SELECT state FROM owners WHERE state IN (SELECT state FROM professionals)
SELECT owners.state FROM owners INNER JOIN professionals ON owners.state = professionals.state
SELECT owners.state FROM owners INTERSECT SELECT professionals.state FROM professionals
SELECT owners.state FROM owners WHERE owners.state IN (SELECT professionals.state FROM professionals)
```

Figure 3.33: Overall Framework of the C3 Method.

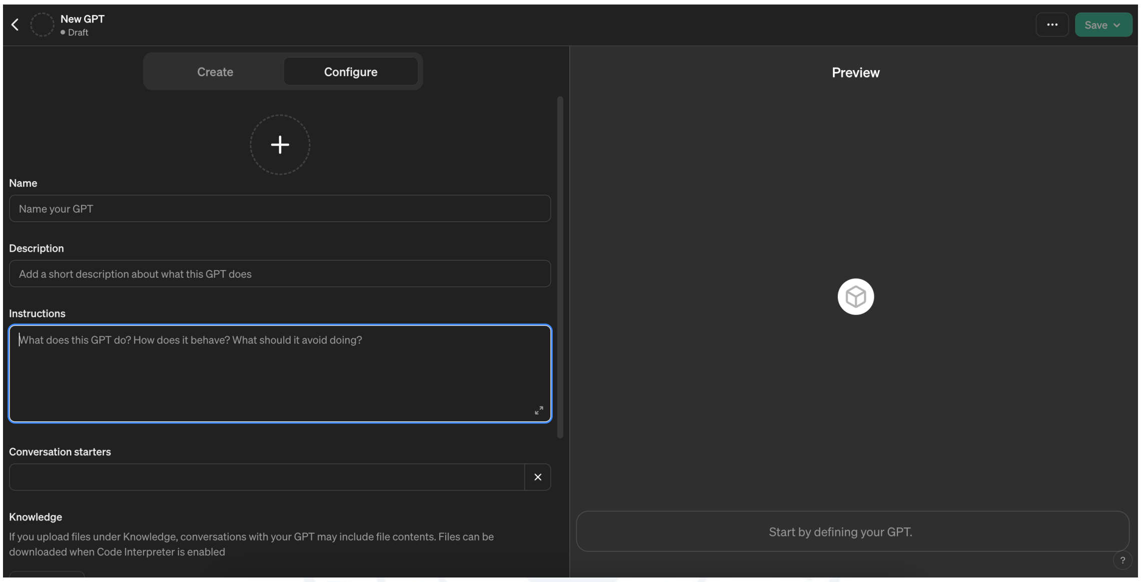
content, reduce context length, and minimize redundant information, thereby improving the accuracy of the generated SQL. This reflects the principle of "rich and clear context" in prompt techniques, ensuring that the model focuses on key information when handling tasks.

To address the inherent biases of ChatGPT, C3 employs **Calibration with Hints**. Using a plug-in calibration strategy, it incorporates prior knowledge into ChatGPT through contextual prompts containing historical dialogues. In these historical dialogues, ChatGPT is set up as an expert SQL role, and through dialogue, it is guided to follow the preset prompts. This role-playing approach effectively calibrates the biases.

On the model output side, C3 adopts **Output Calibration** to address the inherent randomness of large language models. C3 applies the Self-Consistency method to the Text-to-SQL task, sampling multiple inference paths and selecting the most consistent answer, thereby enhancing output stability and maintaining the consistency of SQL queries.

3.5.4 GPTs

GPTs are customizable GPT applications introduced by OpenAI, allowing users to create custom versions of GPT applications by writing prompts and adding tools, or by using GPTs models shared by others.



Name your GPT

Add a short description about what this GPT does

What does this GPT do? How does it behave? What should it avoid doing?

If you upload files under Knowledge, conversations with your GPT may include file contents. Files can be downloaded when Code Interpreter is enabled

Start by defining your GPT.

Figure 3.34: Page for Creating GPTs [6].

ZHEJIANG UNIVERSITY 1897

Figure 3.34 demonstrates how to create GPTs. On this dedicated page, users have ample freedom to customize the capabilities and functions of GPTs. The "Description" section clarifies the functionality of the GPTs, helping other users quickly understand its purpose and role. The "Instructions" section is used to preset prompts for the GPTs, enabling it to achieve the desired functionality. Using the prompt engineering techniques discussed in this chapter to write these critical sections can significantly enhance the performance of GPTs. Additionally, users can customize their own knowledge bases and select required capabilities according to their needs, such as web search, image generation, code explanation, etc. After completing the customization of a personal GPTs, users can choose to share it for others to use.

In this section, we have explored in detail the specific application scenarios of prompt engineering across various fields. Prompt engineering plays a crucial role in enhancing the interaction and execution capabilities of large language models. Whether in task planning, data synthesis, personalized model customization, or Text-to-SQL, prompt engineering has demonstrated its unique advantages and broad application prospects.

Reference

[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

Parameter-Efficient Fine-Tuning

Large language models have acquired rich world knowledge from vast amounts of pre-training data. However, ”every coin has two sides,” and for vertical domains with limited pre-training data, large language models cannot adapt to these domains solely through prompt engineering. To better adapt large language models to these domains, their parameters need to be fine-tuned. However, due to the enormous number of parameters in large language models, fine-tuning costs are high, hindering their application in some vertical fields. To reduce fine-tuning costs, it is urgent to achieve reliable and cost-effective parameter-efficient fine-tuning.

This chapter will delve into the current mainstream parameter-efficient fine-tuning technologies. Firstly, it will briefly introduce the concept of parameter-efficient fine-tuning, parameter efficiency, and method classification. Then, it will detail the three main types of parameter-efficient fine-tuning methods, including additional parameters methods, parameter selection methods, and low-rank adaptation methods, exploring the implementation and advantages of their representative algorithms. Finally, this chapter will showcase practical applications of parameter-efficient fine-tuning in vertical fields through specific cases.

*This book is continuously updated, with the GIT Hub link being: https://github.com/ZJU-LLMs/Foundations-of-LLMs.

浙江大学 1897 ZHEJIANG UNIVERSITY

4.1 Introduction to Parameter-Efficient Fine-Tuning

For vertical domains with limited pre-training data, large language models need to be adapted to these domains and corresponding downstream tasks. In-context learning and instruction tuning are effective ways to adapt to downstream tasks, but they have shortcomings in terms of effectiveness or efficiency. To address these deficiencies, Parameter-Efficient Fine-Tuning (PEFT) technology has emerged. This section will first review in-context learning and instruction tuning, analyze their advantages and disadvantages, and discuss their limitations in practical applications. Next, we will introduce the concept of parameter-efficient fine-tuning and its importance, explaining its significant advantages in reducing costs and improving efficiency. Subsequently, we will classify the mainstream PEFT methods, including additional parameters methods, parameter selection methods, and low-rank adaptation methods, and introduce the basic principles and representative works of each method.

1897 ZHEJIANG UNIVERSITY

4.1.1 Downstream Task Adaptation

Typically, large language models accumulate rich world knowledge and acquire the ability to handle multi-tasks through pre-training on large-scale datasets [43]. However, due to the limited training data of open-source large language models, there are still knowledge boundaries, leading to insufficient knowledge in vertical domains (such as medicine, finance, law, etc.), which affects performance in these domains.

Therefore, downstream task adaptation is needed to further improve performance in vertical and niche domains. The mainstream downstream task adaptation methods are: a) **In-context Learning** [8]; b) **Instruction Tuning** [53].

1. In-context Learning

In previous content, we have introduced the related aspects of in-context learning. Its core idea is to transform different types of tasks into generation tasks by designing prompts to drive large language models to complete these downstream tasks. Few-shot in-context learning converts sample-label pairs in the dataset into natural language instructions and examples, and concatenates them with the samples to be tested, inputting them to the large language model and using the model output as the final prediction result. In-context learning does not require updating the model parameters at all, so it can quickly apply a single model to multiple different tasks.

Although in-context learning can effectively utilize the capabilities of large language models in practical applications, its disadvantages are also obvious: 1) The performance of in-context learning still lags behind fine-tuning, and prompt design requires a lot of human effort, with significant differences in final task performance for different prompts; 2) Although in-context learning does not require training, the cost during the inference phase increases rapidly with the increase of examples in the prompt. Therefore, fine-tuning large language models is still necessary in many scenarios and tasks, especially in vertical domains.

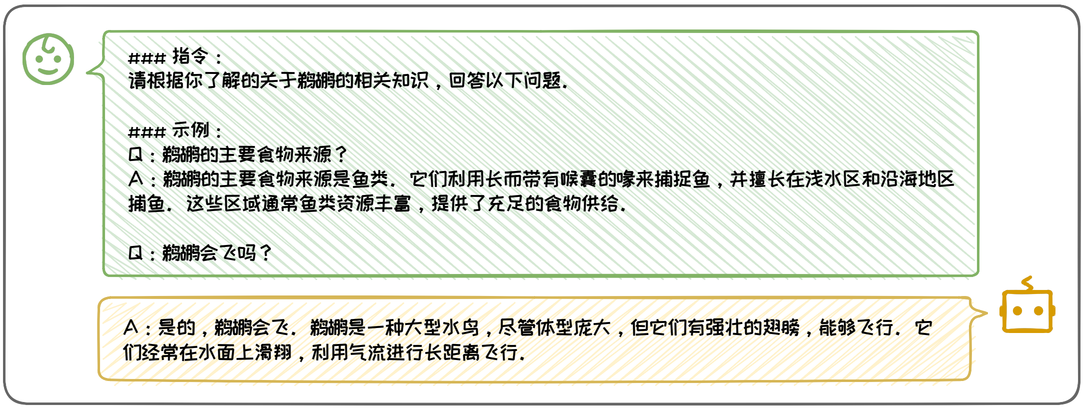
2. Instruction Tuning

Instruction Tuning is another method for downstream task adaptation. Instruction tuning aims to train the model to learn task instructions, enabling it to better understand and execute various natural language processing tasks. Instruction tuning first constructs an instruction dataset and then performs supervised fine-tuning on this dataset.

- **Instruction Data Construction**:

  Instruction data typically includes instructions (task descriptions), examples (op-

ZHEJIANG UNIVERSITY 1897



```
### 指令：
请根据你了解的关于鹈鹕的相关知识，回答以下问题．

### 示例：
Q：鹈鹕的主要食物来源？
A：鹈鹕的主要食物来源是鱼类． 它们利用长而带有喉囊的喙来捕捉鱼，并擅长在浅水区和沿海地区捕鱼． 这些区域通常鱼类资源丰富，提供了充足的食物供给．

Q：鹈鹕会飞吗？
```

```
A：是的，鹈鹕会飞． 鹈鹕是一种大型水鸟，尽管体型庞大，但它们有强壮的翅膀，能够飞行． 它们经常在水面上滑翔，利用气流进行长距离飞行．
```

Figure 4.1: Example of instruction data.

tional), questions, and answers, as shown in Figure 4.1. Constructing such instruction data generally has **two methods** [53]: 1) Data integration. By using templates to convert labeled natural language datasets into instruction-formatted <input, output> pairs. For example, Flan [47] and P3 [37] datasets are constructed based on the data integration strategy; 2) Large language model generation. By manually collecting or handwriting a small amount of instruction data, and then using closed-source large language models like GPT-3.5-Turbo or GPT4 for instruction expansion. For example, InstructWild [33] and Self-Instruct [46] datasets use this method to generate.

- **Supervised Fine-tuning**: After constructing the dataset, the pre-trained model can be fine-tuned in a fully supervised manner. Given the instructions and inputs, the model is trained by sequentially predicting each token in the output. Fine-tuned large language models can significantly improve instruction-following capabilities, which helps enhance their reasoning levels and generalize to new tasks and domains.

Although instruction tuning can effectively help large language models understand new domain data knowledge and improve performance on downstream tasks, supervised fine-tuning requires significant computing resources. Taking the LLaMA2-7B [40] model

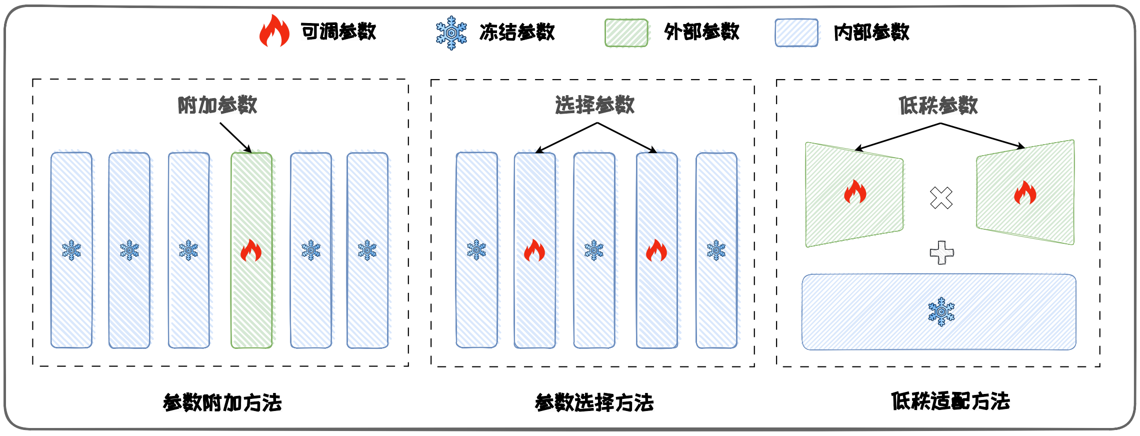


Figure 4.2: Taxonomy of parameter-efficient fine-tuning methods.

as an example, full fine-tuning requires nearly 60GB of memory, and ordinary consumer-grade GPUs (such as RTX4090 (24GB)) cannot complete the fine-tuning. Therefore, to effectively fine-tune large language models in resource-constrained environments, it is particularly important to research parameter-efficient fine-tuning technologies.

4.1.2 Parameter-Efficient Fine-Tuning

Parameter-Efficient Fine-Tuning (PEFT) aims to avoid fine-tuning all parameters, reducing the number of parameters and computational overhead required for fine-tuning, thereby improving the efficiency of fine-tuning large language models. The mainstream PEFT methods can be classified into three categories: additional parameters methods (Additional Parameters Methods), parameter selection methods (Parameter Selection Methods), and low-rank adaptation methods (Low-Rank Adaptation Methods), as shown in Figure 4.2.

1. Additional Parameters Methods

Additional Parameters Methods (Additional Parameters Methods) add new, smaller trainable modules to the model structure. During fine-tuning, the original model parameters are frozen, and only these newly added modules are fine-tuned to achieve efficient fine-tuning. These modules are typically called adapter layers. They are inserted between different layers of the model to capture specific task information. Since these newly added adapter layers have a small number of parameters, additional parameters methods can significantly reduce the number of parameters that need to be updated. Typical methods include: **Adapter-tuning** [18], **Prompt-tuning** [23], **Prefix-tuning** [24], and **Proxy-tuning** [27]. Additional parameters methods will be introduced in detail in Section 4.2.

2. Parameter Selection Methods

Parameter Selection Methods (Parameter Selection Methods) only select a part of the model parameters for fine-tuning while freezing the rest. This method leverages the characteristic that only a portion of the parameters in the model are decisive for downstream tasks, ”focusing on the main contradiction,” and only fine-tuning these key parameters. Selectively fine-tuning these key parameters can reduce the computational burden while improving model performance. Typical methods include: **BitFit** [50], **Child-tuning** [49], and **FishMask** [39]. Parameter selection methods will be introduced in detail in Section 4.3.

3. Low-Rank Adaptation Methods

Low-Rank Adaptation Methods (Low-rank Adaptation Methods) approximate the original weight update matrix with a low-rank matrix and freeze the original parameter matrix, only fine-tuning the low-rank update matrix. Since the number of parameters in the low-rank update matrix is much smaller than that in the original parameter update matrix, it significantly saves memory overhead during fine-tuning. LoRA [19] is a classic low-rank adaptation method, and subsequent variants such as **AdaLoRA** [52], **DyLoRA** [42], and **DoRA** [29] have been proposed to further improve LoRA performance. Low-rank adaptation methods will be introduced in detail in Section 4.4.

4.1.3 Advantages of Parameter-Efficient Fine-Tuning

Parameter-efficient fine-tuning has the following advantages: 1) **High computational efficiency**: PEFT technology reduces the number of parameters that need to be updated, thereby lowering the computational resource consumption during training; 2) **High storage efficiency**: By reducing the number of parameters that need to be fine-tuned, PEFT significantly reduces the storage space required for fine-tuned models, especially suitable for memory-constrained devices; 3) **Strong adaptability**: PEFT can quickly adapt to different tasks without retraining the entire model, making the model more flexible in changing environments.

Below, we will explore a specific case to delve into how PEFT technology significantly improves parameter efficiency. Specifically, Table 4.1 [1] details the comparison of GPU memory consumption when performing full fine-tuning versus using the parameter-efficient fine-tuning method LoRA (which will be introduced in detail in Section 4.4) on the bigscience model in a high-performance hardware environment equipped with an A100 GPU with 80GB of VRAM and more than 64GB of CPU memory.

Table 4.1: Comparison of GPU memory consumption between full parameter fine-tuning and parameter-efficient fine-tuning (OOM represents out of memory).

Model Name	Full Parameter Fine-Tuning	Parameter-Efficient Fine-Tuning (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

According to this table, full parameter fine-tuning of 7B/12B parameter models on a

---

[1] Table data source: https://github.com/huggingface/peft

GPU with 80GB of VRAM would cause the memory to overflow. When using LoRA, the GPU memory usage is significantly reduced, making it feasible to fine-tune large language models on a single card.

This section first introduced the two mainstream paradigms for adapting large language models to downstream tasks: in-context learning and instruction tuning. However, due to performance and computational cost limitations, these paradigms are difficult to adapt to needs. Therefore, it is necessary to research parameter-efficient fine-tuning technology, namely PEFT. PEFT only updates a small portion of the model parameters, effectively reducing the number of parameters required for model fine-tuning without sacrificing performance, thereby saving computational and storage resources. We classified the mainstream PEFT methods in this section. In the following subsections, we will introduce the three main types of PEFT methods in detail according to the taxonomy given in this section: additional parameters methods 4.2, parameter selection methods 4.3, and low-rank adaptation methods 4.4, and discuss the applications and practices of PEFT 4.5.

4.2 Additional Parameter Methods

Additional Parameter Methods enhance large language models by adding and training new additional parameters or modules. These methods can be categorized into three types based on their attachment positions: input, model, and output. This section will provide a detailed introduction to the representative works of each type.

4.2.1 Input-attached Methods

Input-attached methods append extra parameters to the input embeddings of the model, with the most classic method being Prompt-tuning [23]. Prompt-tuning introduces differentiable continuous tensors into the model's input, commonly referred to as soft prompts. These soft prompts are part of the input and are fed into the model along with the actual text data. During fine-tuning, only the parameters of the soft prompts are updated, while other parameters remain unchanged, thus achieving parameter-efficient fine-tuning.

Specifically, given an input text sequence containing n tokens $\{w_1, w_2, \ldots, w_n\}$, it is first converted into an **input embedding** matrix $X \in \mathbb{R}^{n\times d}$ through the embedding layer, where d is the dimension of the embedding space. The newly added soft prompt parameters are represented as the **soft prompt embedding** matrix $P \in \mathbb{R}^{m\times d}$, where m is the length of the soft prompt. Then, the soft prompt embedding is concatenated with the input embedding matrix to form a new matrix $[P; X] \in \mathbb{R}^{(m+n)\times d}$, which is finally input into the Transformer model. The model is trained by maximizing the output probability likelihood through backpropagation. Only the soft prompt parameters P are updated during the training process. Figure 4.3 provides a schematic diagram of Prompt-tuning.

In practical use, the length of the soft prompt ranges from 1 to 200, and performance is usually guaranteed with a length of over 20. Additionally, the initialization of the soft prompt also affects the final performance. Initializing with tokens from the vocabulary or class names in classification tasks tends to be superior to random initialization. It is worth noting that the original motivation of Prompt-tuning was not to achieve parameter-efficient fine-tuning but to automatically learn prompt words. In Chapter 3, we mentioned that the

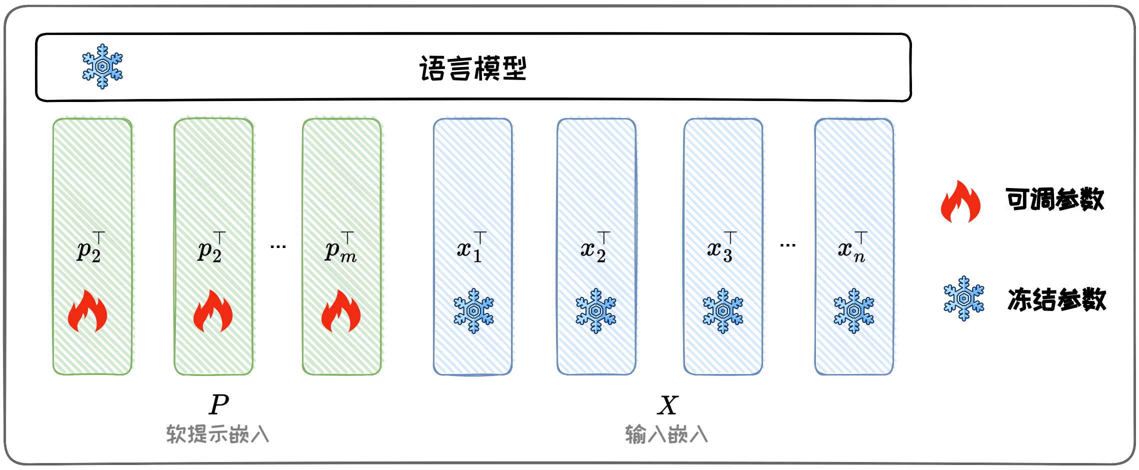


Figure 4.3: Schematic diagram of Prompt-tuning.

common approach to using large language models is through prompt engineering, also known as hard prompts, because the discrete prompts we use are non-differentiable. The problem is that the output quality of large language models highly depends on the construction of the prompt words. Finding the correct ”spell” to make our large language model perform optimally requires a significant amount of time. Therefore, adopting a differentiable approach to automatically optimize prompt words becomes an effective method.

In summary, Prompt-tuning has the following advantages: (1) **High memory efficiency**: Prompt-tuning significantly reduces memory requirements. For example, the T5-XXL model requires 11B parameters for specific tasks, but the model after Prompt-tuning only needs 20480 parameters (assuming the soft prompt length is 5); (2) **Multi-task capability**: A single frozen model can be used for multi-task adaptation. Traditional model fine-tuning requires learning and saving a complete pre-trained model copy for each downstream task, and inference must be performed in separate batches. Prompt-tuning only needs to store a small task-specific module for each task and can use the original pre-trained model for mixed-task inference (adding the learned soft prompt before each task prompt); (3) **Scaling properties**: As the number of model parameters increases, the performance

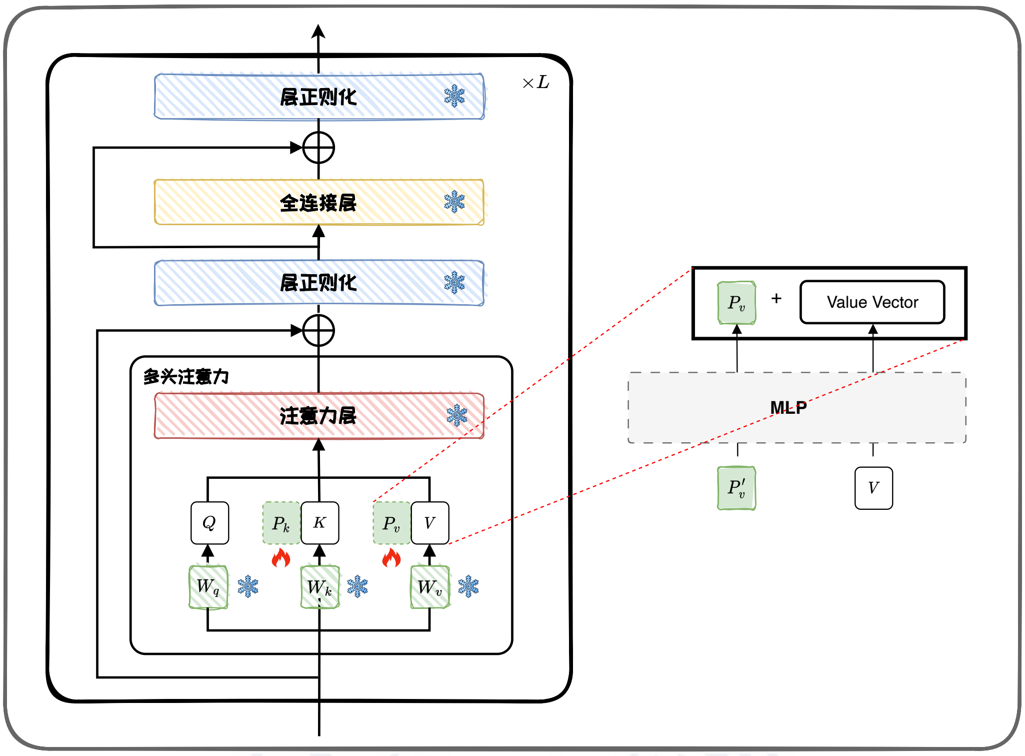


Figure 4.4: Schematic diagram of Prefix-tuning.

of Prompt-tuning gradually improves and approaches the performance of (multi-task) full parameter fine-tuning at 10B parameters.

4.2.2 Model-attached Methods

Model-attached methods add extra parameters or models to the hidden layers of the pre-trained model, with classic methods including Prefix-tuning [24], Adapter-tuning [18], and AdapterFusion [35].

1. Prefix-tuning

Prefix-tuning is very similar to the Prompt-tuning introduced in the previous section, with the difference being that Prompt-tuning only adds soft prompts to the input embeddings, while Prefix-tuning inserts a series of continuous trainable prefixes (i.e.,

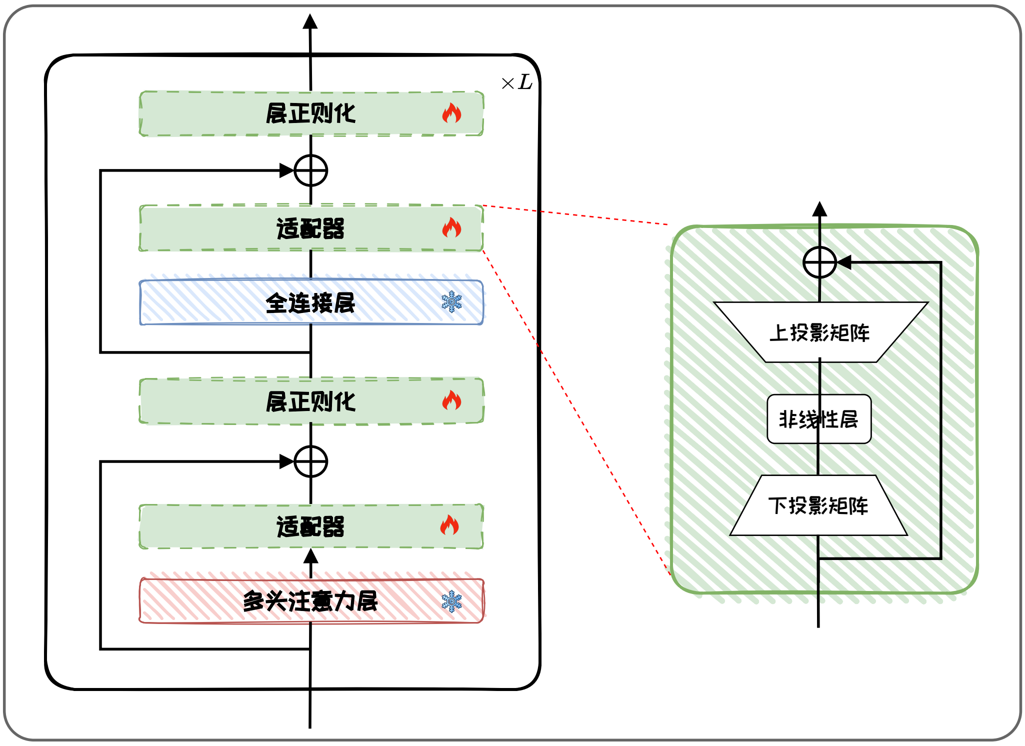


Figure 4.5: Schematic diagram of Adapter-tuning.

Soft-prompt) into both the input embeddings and the Transformer attention modules, as shown in Figure 4.4, where P_k and P_v are the prefixes inserted into the Transformer block. Compared to Prompt-tuning, Prefix-tuning significantly increases the number of learnable parameters.

Specifically, Prefix-tuning introduces a set of learnable vectors P_k and P_v, which are added to the keys K and values V in all Transformer attention modules. Similar to Prompt-tuning, Prefix-tuning also faces the problem of unstable parameter updates of the prefix, which makes the optimization process difficult to converge. Therefore, in practical applications, it is usually necessary to reparameterize through a multilayer perceptron (MLP) before inputting into the Transformer model. This means that the parameters to be trained include both the MLP and the prefix matrix. After training, the parameters of the MLP are discarded, and only the prefix parameters are retained. In summary, Prefix-tuning has the following advantages: (1) **Parameter efficiency**: Only the prefix parameters are updated during fine-tuning, significantly reducing the number of training parameters; (2) **Task adaptability**: The prefix parameters can be customized for different downstream tasks, making the fine-tuning method flexible; (3) **Preservation of pre-trained knowledge**: Since the original parameters of the pre-trained model remain unchanged, Prefix-tuning can retain the knowledge learned during pre-training.

2. Adapter-tuning

Adapter-tuning [18] inserts new learnable neural network modules into the pre-trained language model, called adapters. Adapter modules typically adopt a bottleneck structure, consisting of an up-projection layer, a nonlinear mapping, and a down-projection layer. The down-projection layer compresses the information into a low-dimensional representation, which is then expanded back to the original dimension through the up-projection layer. As shown in Figure 4.5, Adapter-tuning adds adapters after each **multi-head attention layer** (red block) and **feed-forward network layer** (blue block) in the Transformer. **Unlike the feed-forward layers of the Transformer, the hidden dimension of the adapter is usually smaller than the input dimension due to the bottleneck structure**. Similar to other PEFT methods, during training, by fixing the original model parameters, only the adapter, layer normalization (green box), and the final classification layer parameters (not labeled in the figure) are fine-tuned, significantly reducing the number of fine-tuning parameters and computational load, thus achieving parameter-efficient fine-tuning. The specific structure of the adapter module is shown on the right side of Figure 4.5, which typically consists of a down-projection matrix $W_d \in \mathbb{R}^{d\times r}$ and an up-projection matrix $W_u \in \mathbb{R}^{r\times d}$, along with residual connections, where $r << d$:

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

where $\sigma(\cdot)$ is the activation function, such as ReLU or Sigmoid. $A^{(l)}$ is the output of the adapter, and $H^{(l-1)}$ is the hidden state of the $l-1$th layer.

In the adapter, the down-projection matrix compresses the input d-dimensional features to the low-dimensional r, which is then projected back to the d-dimensional space by the up-projection matrix. Therefore, the total number of parameters in each layer is $2dr+d+r$, including the projection matrices and their bias terms. By setting $r \ll d$, the number of parameters required for each task can be significantly limited. Furthermore, the structure of the adapter module can be designed to be more complex, such as using multiple projection layers, or different activation functions and parameter initialization strategies. Additionally, there are many variants of Adapter-tuning, such as adjusting the position of the adapter modules [14, 56], pruning [15], etc., to reduce the number of trainable parameters.

3. AdapterFusion

Since Adapter-tuning does not update the pre-trained model but learns the task-specific knowledge through adapter parameters, each adapter parameter stores the knowledge required to solve that task. Therefore, if you want to combine the knowledge of multiple tasks, you can consider combining the adapter parameters of multiple tasks. Based on this idea, AdapterFusion proposes a two-stage learning method, first learning multiple tasks and extracting knowledge for each task; then ”fusing” the knowledge from multiple tasks. The specific two-stage steps are as follows:

First stage: Knowledge extraction. Given N tasks, first train the adapter modules for each task separately to learn the specific task knowledge. There are two training methods in this stage, as follows:

- **Single-Task Adapters (ST-A)**: For N tasks, the model is optimized independently

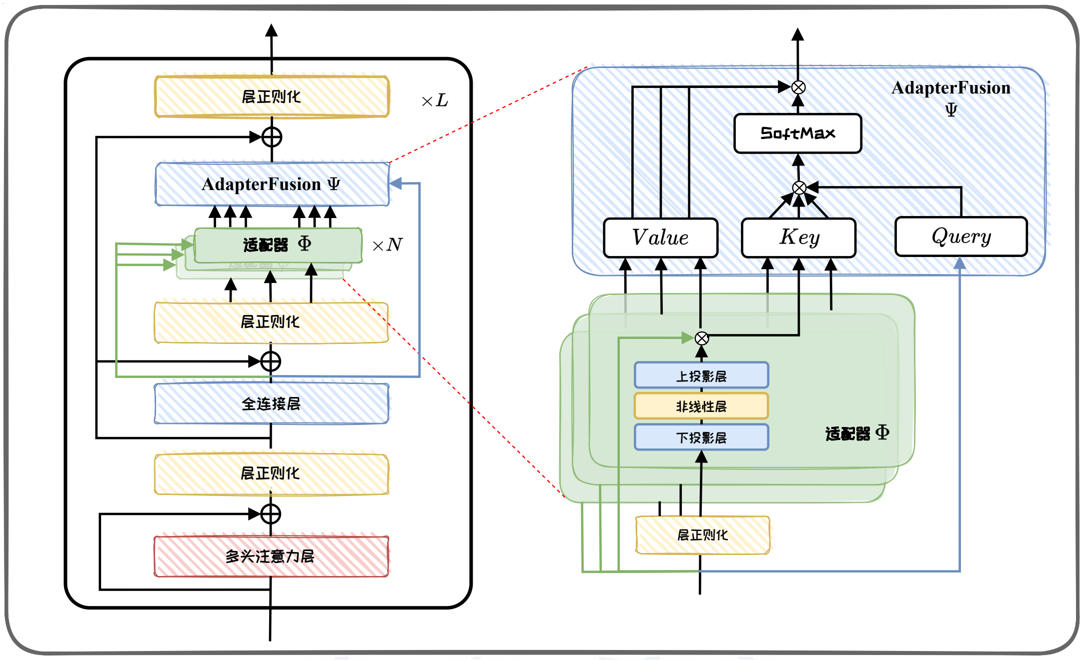


Figure 4.6: Schematic diagram of AdapterFusion.

for each task, with no interference or influence between tasks.

- **Multi-Task Adapters (MT-A)**: Jointly optimize N tasks through multi-task learning.

Second stage: Knowledge combination. After the adapter modules for individual tasks are trained, AdapterFusion combines (Fusion) the different adapter modules to achieve knowledge combination. This stage introduces a new fusion module, which aims to search for the optimal combination of multiple task adapter modules to achieve task generalization. In this stage, the parameters of the language model and the N adapters are fixed, only the parameters of the AdapterFusion module are fine-tuned, and the following loss is optimized:

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

where D_n is the training data for the nth task, L_n is the loss function for the nth task, Θ represents the parameters of the pre-trained model, Φ_i represents the parameters of the

ZHEJIANG UNIVERSITY 1897

adapter module for the ith task, and Ψ is the parameter of the fusion module. Figure 4.6 provides a schematic diagram of AdapterFusion. Each layer's AdapterFusion module includes learnable Key, Value, and Query projection matrices. The output of the fully connected layer is treated as the Query, the output of the adapter is treated as the Key and Value, and the attention is calculated to obtain the output result combined with multiple adapters. Additionally, parameter sharing between different adapter modules can further reduce the number of parameters.

4.2.3 Output-attached Methods

When fine-tuning large language models, we often face the following problems: First, the **number of parameters in large language models may be very large**, such as the LLaMA series' largest model with 70B parameters, even with PEFT technology, it is difficult to complete downstream task adaptation on ordinary consumer-grade GPUs; Second, users may **not be able to directly access the weights of large language models** (black-box models), which sets obstacles for fine-tuning.

To address these issues, Proxy-tuning [27] provides a lightweight decoding-time algorithm that allows us to further customize large language models without directly modifying the model weights by only accessing the model's output vocabulary prediction distribution.

As shown in Figure 4.7, given the **proxy model** $\mathcal{M}$ to be fine-tuned and a smaller **anti-expert model** (Anti-expert model) $\mathcal{M}^-$, these two models need to have the same vocabulary. We fine-tune $\mathcal{M}^-$ to obtain the fine-tuned **expert model** (Expert model) $\mathcal{M}^+$. At each autoregressive generation time step, Proxy-tuning first calculates the $logits$ distribution difference between the expert model $\mathcal{M}^+$ and the anti-expert model $\mathcal{M}^-$, and

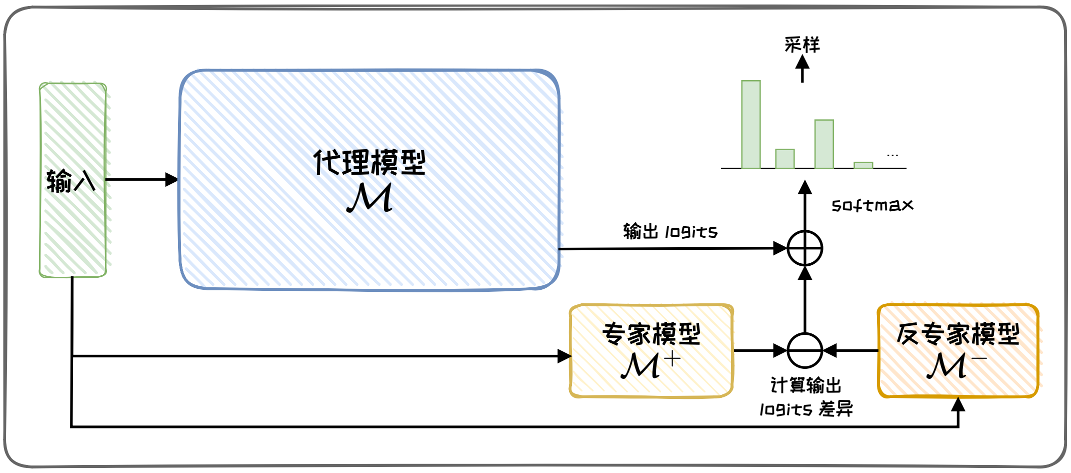


Figure 4.7: Schematic diagram of Proxy-tuning.

then adds it to the *logits* distribution of the next word prediction of the proxy model $\mathcal{M}$. Specifically, during the calculation phase of Proxy-tuning, for each time step t of the input sequence $x_{<t}$, obtain the corresponding output scores $s_{\mathcal{M}}, s_{\mathcal{M}^+}, s_{\mathcal{M}^-}$ from the proxy model $\mathcal{M}$, expert model $\mathcal{M}^+$, and anti-expert model $\mathcal{M}^-$. Adjust the output score $\tilde{s}$ of the target model as follows:

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

Then, use softmax$(\cdot)$ to normalize it to obtain the output probability distribution,

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

Finally, sample the prediction result of the next word from this distribution.

In practical use, the expert model is usually a smaller model (e.g., LLaMA-7B), while the proxy model is a much larger model (e.g., LLaMA-13B or LLaMA-70B). Through Proxy-tuning, we transfer the knowledge learned in the smaller model to a much larger model in a decoding-time constraint manner, significantly saving computational costs. At the same time, since only the output distribution of the model is required, and not the original model weights, this method is also applicable to black-box models.

1897 ZHEJIANG UNIVERSITY

This section introduced three main additional parameter methods, each achieving through input-attached, model-attached, and output-attached ways. Overall, these methods are all effective means to enhance the performance of pre-trained language models, each with its advantages. Input-attached methods add learnable tensors to the input sequence with minimal structural modifications to the model, offering better flexibility. Model-attached methods maintain the original pre-trained model parameters, showing better generalization capabilities. Output-attached methods can drive larger-parameter black-box models with much smaller costs, bringing more practical application prospects.

4.3 Parameter Selection Methods

Parameter Selection Methods selectively fine-tune a subset of parameters in the pre-trained model. Unlike additional parameter methods, parameter selection methods do not add extra parameters to the model, avoiding the introduction of additional computational costs during inference. Typically, parameter selection methods are divided into two categories: rule-based methods and learning-based methods.

4.3.1 Rule-based Methods

Rule-based methods determine which parameters should be updated based on human expert experience. The most representative method in rule-based methods is BitFit [50]. BitFit achieves parameter-efficient fine-tuning by optimizing only the bias terms (Biases) in each layer of the neural network and the task-specific classification head. Since bias terms account for a very small proportion of the total model parameters (about 0.08‰-0.09‰), BitFit has extremely high parameter efficiency. Despite fine-tuning only a small number of parameters, BitFit can still achieve performance comparable to full fine-tuning on the GLUE Benchmark [44], and even better on some tasks. Additionally, BitFit allows the use of larger learning rates compared to full fine-tuning, making the overall optimization process more stable. However, this method has only been validated on small models (such as BERT, RoBERT, etc.), and its performance on larger models remains unknown.

In addition to BitFit, there are other rule-based methods that improve parameter efficiency by fine-tuning only specific Transformer layers. For example, Lee et al. [22] proposed that fine-tuning only the last quarter of the layers in BERT and RoBERTa can achieve 90% of the performance of full parameter fine-tuning. PaFi [26] selects model parameters with the smallest absolute values as trainable parameters.

4.3.2 Learning-based Methods

Learning-based methods automatically select a subset of trainable parameters during model training. The most typical method is Child-tuning [49]. It achieves parameter-efficient fine-tuning by using a **gradient mask matrix** strategy to update only the selected sub-network while masking gradients outside the sub-network. Specifically, suppose $\mathbf{W}_t$ is the parameter matrix at the t-th iteration. We introduce a 0-1 mask matrix $\mathbf{M}_t$ of the same dimension as $\mathbf{W}_t$ to select the sub-network $\mathbf{C}_t$ at the t-th iteration, updating only the parameters of this sub-network, defined as follows:

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

Here, $\mathbf{M}_t^{(i)}$ and $\mathbf{W}_t^{(i)}$ are the i-th elements of matrices $\mathbf{M}_t$ and $\mathbf{W}_t$ at the t-th iteration, respectively. The gradient update formula is:

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

Child-tuning provides two ways to generate the sub-network mask $\mathbf{M}$, resulting in two variant models: Child-tuning$_F$ and Child-tuning$_D$. Child-tuning$_F$ is a **task-agnostic** variant that selects the sub-network without relying on any downstream task data. At each iteration, Child-tuning$_F$ draws a 0-1 mask from a Bernoulli distribution to generate the gradient mask $\mathbf{M}_t$:

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

where p_F is the probability of the Bernoulli distribution, representing the proportion of the sub-network. Additionally, Child-tuning$_F$ introduces noise to regularize the full gradient, thereby preventing overfitting on small datasets and improving generalization.

Child-tuning$_D$ is a **task-driven** variant that uses downstream task data to select the sub-network most relevant to the task. Specifically, Child-tuning$_D$ uses the Fisher Information Matrix (FIM) to estimate the importance of specific task-related parameters. Specifically, for the given task training data $\mathcal{D}$, the Fisher information estimate for the i-th parameter matrix $W^{(i)}$ of the model is:

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

where X_i and Y_i represent the input and output of the i-th sample, respectively, and $\log p(Y_i|X_i;W)$ is the log-likelihood probability, obtained by calculating the gradient of the loss function with respect to parameter $W^{(i)}$. Generally, we assume that the more important the parameter is to the target task, the higher its Fisher information value. Therefore, the sub-network can be selected based on Fisher information, with the sub-network

$\mathbf{C}$ consisting of parameters with the highest Fisher information. The steps to select the sub-network parameters are as follows: 1) Calculate the Fisher information value for each parameter; 2) Sort these Fisher information values; 3) Select the top p_D proportion of parameters as the sub-network $\mathbf{C}$. After determining the sub-network, generate the corresponding mask matrix to complete model training.

Child-tuning reduces computational burden through gradient masking and reduces the model's hypothesis space, lowering the risk of overfitting. However, selecting the sub-network requires additional computational costs, especially in the task-driven variant, where calculating Fisher information is very time-consuming. Overall, Child-tuning can improve the performance of large language models on various downstream tasks, especially when training data is limited. Additionally, Child-tuning can be well integrated with other PEFT methods to further enhance model performance.

In addition to Child-tuning, there are other learning-based parameter selection methods. For example, Zhao et al. [55] introduced a binary matrix related to model weights to learn, generating parameter masks (masks) through a threshold function, and then updating them through a noise estimator during backpropagation. Similar to FishMask, FishDip [5] also uses Fisher information to calculate masks, but the masks will be dynamically recalculated in each training epoch. LT-SFT [2] was inspired by the "lottery ticket hypothesis" [11] and formed masks based on the subset of parameters that changed the most during the initial fine-tuning phase according to parameter importance. SAM [12] proposed a second-order approximation method to help determine parameter masks by solving the optimization function analytically.

Selection-based methods selectively update parameters of the pre-trained model, fine-tuning the model while keeping most parameters unchanged. These methods can significantly reduce the number of parameters that need to be updated during fine-tuning, lowering computational costs and memory requirements. They are particularly suitable for resource-constrained environments or scenarios that require rapid adaptation to new tasks. However, these methods also face challenges, such as how to select the optimal subset of parameters and how to balance the number of parameter updates and model performance.

4.4 Low-Rank Adaptation Methods

The low-dimensional intrinsic dimension hypothesis [1] suggests that over-parameterized models have a low intrinsic dimension; in other words, there exist low-dimensional parameter updates that can rival full-parameter updates. Based on this hypothesis, Low-Rank Adaptation Methods (Low-rank Adaptation Methods) approximate the original weight update matrix with a low-rank matrix and only fine-tune the low-rank matrix to significantly reduce the number of model parameters. In this section, we will first introduce the implementation details of the most classic low-rank adaptation method, LoRA, and analyze its parameter efficiency. Next, we will introduce the variants of LoRA. Finally, we will discuss the plug-and-play nature of LoRA and its task generalization capabilities.

4.4.1 LoRA

Low-Rank Adaptation (LoRA) [19] proposes to use low-rank matrices to approximate the parameter update matrix to achieve low-rank adaptation. This method decomposes the parameter update matrix into two smaller matrices. During fine-tuning, the large language model is updated by fine-tuning these two small matrices, significantly saving memory overhead during fine-tuning. This subsection first introduces the specific implementation

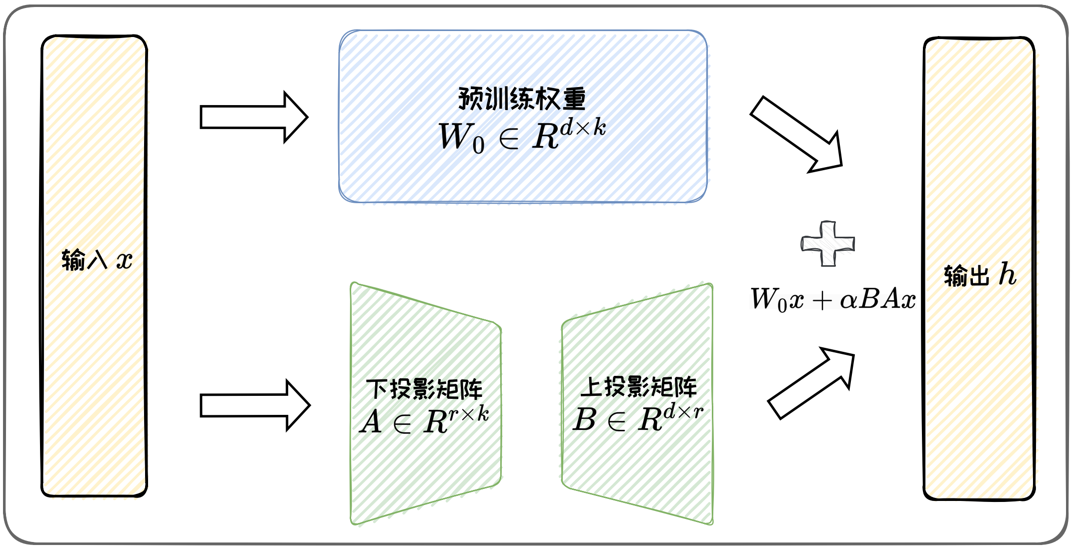


Figure 4.8: Schematic diagram of LoRA.

process of the LoRA method and then analyzes its computational efficiency.

1. Method Implementation

Given a dense neural network layer with a parameter matrix $W_0 \in \mathbb{R}^{d\times k}$, to adapt to downstream tasks, we typically need to learn a parameter update matrix $\Delta W \in \mathbb{R}^{d\times k}$ to update the original parameter matrix $W = W_0 + \Delta W$. For full fine-tuning, ΔW requires calculating gradients for all $d \times k$ parameters of this layer, which usually requires a large amount of GPU memory and is costly.

To address this problem, as shown in Figure 4.8, LoRA decomposes ΔW into two low-parameter matrices $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, making the update process:

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

where the rank $r \ll \min\{d, k\}$, B and A are initialized with random Gaussian distribution and zero, respectively, and α is a scaling factor to control the size of the LoRA weights. During training, the parameters of the pre-trained model are fixed, and only the parameters of B and A are fine-tuned. Therefore, the number of parameters involved in LoRA updates is $r \times (d + k)$, much smaller than the full fine-tuning $d \times k$. In practice, for large language models based on Transformer, dense layers typically have two types: projection layers in the attention module and projection layers in the feed-forward neural network (FFN) module. In the original research, LoRA was applied to the weight matrices of the attention layers. Subsequent work shows that applying it to the FFN layers can further improve model performance [14].

LoRA only fine-tunes a portion of the low-rank parameters, so it has high parameter efficiency and does not increase inference latency [10]. Additionally, the low-rank matrix can also be extended to a low-rank tensor [3] or combined with Kronecker decomposition to further improve parameter efficiency [9, 16].

In addition to parameter efficiency, after training, the LoRA parameters can be separated from the model parameters, so LoRA also has **plug-and-play** properties. The plug-and-play nature of LoRA allows it to be encapsulated as a plugin that can be shared and reused by multiple users [38]. When we have LoRA plugins for multiple tasks, these plugins can be combined to achieve good cross-task generalization performance [20]. We will provide specific cases in Section 4.4.3 to detail the task generalization based on LoRA plugins.

2. Parameter Efficiency

Below, we analyze the parameter efficiency of LoRA with a specific case. Taking the fine-tuning of the first FFN layer's weight matrix in the LLaMA2-7B [40] model as an example, full fine-tuning requires adjusting $11,008 \times 4,096 = 45,088,768$ parameters. When $r = 4$, LoRA only needs to adjust $(11,008 \times 4) + (4 \times 4,096) = 60,416$ parameters. For this layer, compared to full fine-tuning, the parameters fine-tuned by LoRA are less than one-thousandth of the original parameters. Specifically, the memory usage for model fine-tuning mainly involves four parts:

- Weight Memory: Memory required to store model weights;
- Activation Memory: Memory occupied by intermediate activations during forward propagation, mainly depending on batch size and sequence length;
- Gradient Memory: Memory required to store gradients during backpropagation, calculated only for trainable parameters;
- Optimizer Memory: Memory used to store the internal state of the optimizer. For example, the Adam optimizer stores the ”first-order momentum” and ”second-order momentum” of trainable parameters.

Reference [34] provides an experimental comparison of full fine-tuning and LoRA fine-tuning on the LLaMA2-7B model with a batch size of 1 on a single NVIDIA RTX4090 (24GB) GPU. According to the experimental results, full fine-tuning requires approximately 60GB of VRAM, exceeding the capacity of the RTX4090. In contrast, LoRA requires only about 23GB of VRAM. LoRA significantly reduces VRAM usage, making it possible to fine-tune LLaMA2-7B on a single NVIDIA RTX4090.

Specifically, **due to fewer trainable parameters, the optimizer memory and gradient memory are reduced by about 25GB and 14GB, respectively**. Additionally, although LoRA introduces additional ”incremental parameters,” causing a slight increase in **activation memory and weight memory (totaling about 2GB)**, considering the overall reduction in memory, this increase is negligible. Moreover, reducing the number of parameters involved in calculations can accelerate backpropagation. Compared to full fine-tuning, LoRA’s speed is increased by 1.9 times.

4.4.2 LoRA Variants

Although LoRA can achieve good performance on some downstream tasks, there is still a performance gap between LoRA and full fine-tuning on many complex downstream tasks (such as mathematical reasoning [4, 6, 54]). To bridge this gap, many LoRA variants have been proposed to further enhance LoRA's adaptation performance on downstream tasks. Existing methods mainly improve from the following perspectives [31]: (1) Breaking the low-rank bottleneck; (2) Dynamic rank allocation; (3) Training process optimization. Next, we will introduce the representative methods of these three types of variants.

1. Breaking the Low-Rank Bottleneck

LoRA's low-rank update feature gives it an advantage in parameter efficiency; however, this also limits the ability of large-scale language models to memorize new knowledge and adapt to downstream tasks [4, 13, 21, 54], i.e., there is a low-rank bottleneck. Biderman et al. [4] experimental studies show that the rank of full fine-tuning is significantly higher than that of LoRA (10-100 times), and increasing the rank of LoRA can narrow the performance gap between LoRA and full fine-tuning. Therefore, some methods have been proposed to break the low-rank bottleneck [25, 36, 48].

For example, ReLoRA [25] proposes a merge-and-reinit method, which periodically merges the LoRA module into the large language model during fine-tuning and reinitializes the LoRA module and optimizer state after merging. Specifically, the merging process is as follows:

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

where W^i is the original weight matrix, B^i and A^i are the matrices obtained from low-rank decomposition, and α is the scaling factor.

After merging, the values of B^i and A^i are reset, usually B^i is reinitialized with a specific initialization method (such as Kaiming initialization), and A^i is set to zero. To prevent model performance divergence after reset, ReLoRA also partially resets the optimizer state through magnitude pruning.

The merge-and-reinit process allows the model to accumulate multiple low-rank LoRA updates into a high-rank state while maintaining the total number of parameters unchanged, enabling ReLoRA to train models with performance close to full-rank training.

2. Dynamic Rank Allocation

However, the rank of LoRA is not always higher the better, redundant LoRA ranks may lead to performance and efficiency degradation. Moreover, the importance of weights during fine-tuning may vary across different layers in the Transformer model, so different ranks need to be allocated for each layer [7, 30, 41, 52].

For example, AdaLoRA [52] parameterizes the parameter update matrix in the form of singular value decomposition (SVD) and dynamically adjusts the rank of LoRA modules in different layers through singular value pruning.

Specifically, AdaLoRA uses singular value decomposition to re-represent ΔW, i.e.,

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

where $P \in \mathbb{R}^{d\times r}$ and $Q \in \mathbb{R}^{r\times k}$ are orthogonal, and Λ is a diagonal matrix containing singular values $\{\lambda_i\}_{1\leq i\leq r}$. During training, the parameters of W_0 are fixed, and only the parameters of P, Λ, and Q are updated. The importance score of singular values is constructed based on the moving average of the product of gradient weights, and unimportant singular values are iteratively pruned.

Additionally, to enhance stable training, AdaLoRA introduces an additional penalty term to ensure orthogonality between P and Q:

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

where I is the identity matrix, and $|| \cdot ||_F$ represents the Frobenius norm.

3. Training Process Optimization

In practical fine-tuning, LoRA's convergence speed is slower than full fine-tuning. Additionally, it is sensitive to hyperparameters and prone to overfitting. These issues affect LoRA's efficiency and hinder its downstream adaptation performance. To address these issues, some work attempts to optimize LoRA's training process [32, 45].

A representative method, DoRA (Decomposed Rank Adaptation) [29], proposes to constrain gradient updates, focusing on the direction change of parameter updates. It decomposes the pre-trained weight $W_0 \in \mathbb{R}^{d \times k}$ into direction and magnitude components and applies LoRA only to the direction component to enhance training stability. Specifically, DoRA re-represents $W_0 \in \mathbb{R}^{d \times k}$ as:

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

where $m \in \mathbb{R}^{1 \times k}$ is the magnitude vector, $V \in \mathbb{R}^{d \times k}$ is the direction matrix, and $\| \cdot \|_c$ is the vector norm on each column of the matrix. Subsequently, DoRA only applies LoRA to the direction matrix V for parameterization, defined as:

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

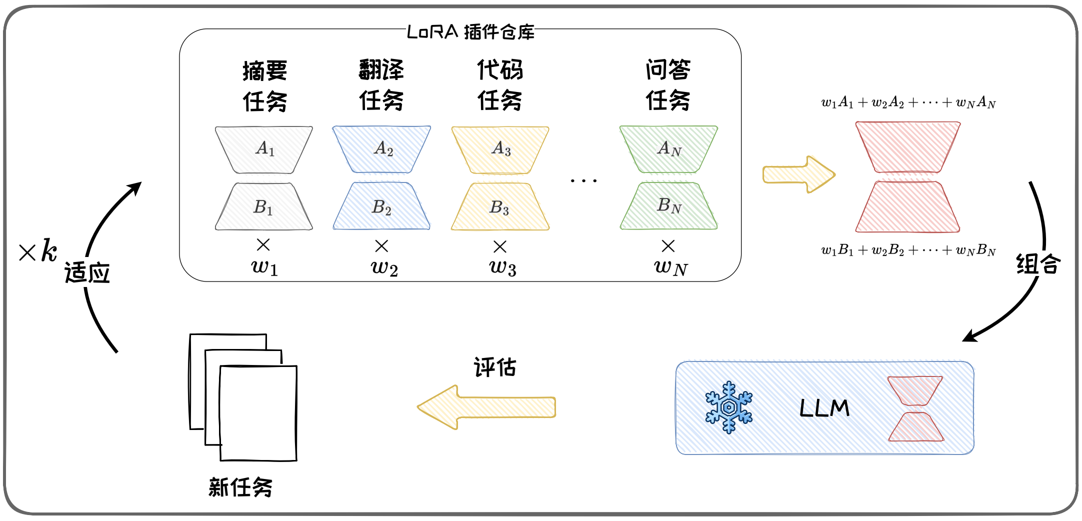


Figure 4.9: Schematic diagram of LoRAHub.

where ΔV is the incremental direction update learned by LoRA, and underlined parameters indicate trainable parameters.

4.4.3 Task Generalization Based on LoRA Plugins

After LoRA fine-tuning, we can separate the parameter update modules B and A from the model and encapsulate them into parameter plugins. These plugins have the excellent properties of plug-and-play, not destroying the original model parameters and structure. We can train various LoRA modules on different tasks and save, share, and use these modules in a plug-in manner. Additionally, we can combine multiple task LoRA plugins and then transfer the capabilities of different tasks to new tasks. LoRAHub [20] provides a usable framework for multi-LoRA combination. It can combine LoRA plugins obtained from existing tasks to acquire the ability to solve new tasks.

As shown in Figure 4.9, LoRAHub includes two stages: **combination stage** and **adaptation stage**.

In the combination stage, LoRAHub combines the learned LoRA modules into a single module through element-wise linear weighting:

$$\hat{m} = (w_1 A_1 + w_2 A_2 + \cdots + w_N A_N)\,(w_1 B_1 + w_2 B_2 + \cdots + w_N B_N)\,, \tag{4.15}$$

where w_i is the weight of the i-th LoRA module, $\hat{m}$ is the combined module, and $A_{i=1}^N$ and $B_{i=1}^N$ are the N LoRA decomposition matrices. **In the adaptation stage**, given some examples of new tasks, the weight combination is adaptively learned through a gradient-free method Shiwa [28]. Adaptation and combination are iterated k times until the optimal weight combination is found to complete the adaptation to new tasks.

This section introduced the low-rank adaptation method LoRA. LoRA decomposes the parameter matrix into low-rank matrices and only trains the low-rank matrices, significantly reducing the number of training parameters. Additionally, we introduced variants that improve LoRA from different perspectives such as breaking the low-rank bottleneck, dynamic rank allocation, and training process optimization. Finally, we introduced the task generalization method LoRAHub based on LoRA plugins. LoRAHub combines the learned LoRA modules through weighted combination, integrating multi-task capabilities and transferring them to new tasks, providing an efficient cross-task learning paradigm.

4.5 Practice and Applications

PEFT technology has demonstrated its powerful application potential in many fields. This section will explore the practice and applications of PEFT. In the practice section, we will introduce the most popular PEFT framework, the open-source library HF-PEFT developed by Hugging Face, and briefly describe its usage methods and related tips. In the application section, we will showcase application cases of PEFT technology in different vertical fields, including table data processing and the financial domain's Text-to-SQL generation task. These cases not only prove the effectiveness of PEFT in enhancing the performance of large models on specific tasks but also provide valuable references for future research and applications.

4.5.1 PEFT Practice

In practical applications, the implementation and optimization of PEFT technology are crucial. This subsection will detail the use of the mainstream PEFT framework, including installation and configuration, fine-tuning strategy selection, model preparation, and training process, as well as related usage tips.

1. Mainstream PEFT Frameworks

The most popular PEFT framework currently is the open-source library HF-PEFT [2], developed by Hugging Face, which aims to provide state-of-the-art parameter-efficient fine-tuning methods. The design philosophy of the HF-PEFT framework is efficiency and flexibility. HF-PEFT integrates various advanced fine-tuning techniques such as LoRA, Adapter-tuning, Prompt-tuning, and IA3. HF-PEFT supports seamless integration with other tools from Hugging Face such as Transformers, Diffusers, and Accelerate, and supports diverse training and inference scenarios from single-machine to distributed environments. HF-PEFT is particularly suitable for large models, enabling high performance on consumer-grade hardware, and can be compatible with model quantization techniques to further reduce the memory requirements of models. HF-PEFT supports multiple architecture models, including Transformer and Diffusion, and allows users to manually configure and enable PEFT on their models.

---

[2] https://github.com/huggingface/peft

Another significant advantage of HF-PEFT is its ease of use. It provides detailed documentation, quick start guides, and example code [3], helping users understand how to quickly train PEFT models and how to load existing PEFT models for inference. Additionally, HF-PEFT has active community support and encourages community contributions, making it a powerful, easy-to-use, and continuously updated library.

2. HF-PEFT Framework Usage

Using the HF-PEFT framework for model fine-tuning can significantly enhance the model's performance on specific tasks while maintaining training efficiency. Typically, the steps to use the HF-PEFT framework for model fine-tuning are as follows:

1. **Installation and Configuration**: First, install the HF-PEFT framework and its dependencies in the environment, primarily the Hugging Face Transformers library.
2. **Selecting Model and Data**: Choose an appropriate pre-trained model based on task requirements and prepare the corresponding training dataset.
3. **Determining Fine-tuning Strategy**: Select a fine-tuning method suitable for the task, such as LoRA or adapter technology.
4. **Model Preparation**: Load the pre-trained model and configure it for the selected fine-tuning method, including task type, inference mode, parameter values, etc.
5. **Model Training**: Define the complete training process, including loss function, optimizer settings, and execute training, while applying techniques such as data augmentation and learning rate scheduling.

Through the above steps, the HF-PEFT framework can be used efficiently for model fine-tuning to adapt to specific task requirements and enhance model performance.

3. PEFT Related Tips

---

[3] https://huggingface.co/docs/peft/

The HF-PEFT library provides various parameter-efficient fine-tuning techniques for effectively adapting pre-trained language models to various downstream applications without fine-tuning all model parameters. Below are some commonly used parameter setting methods for PEFT techniques:

Prompt Tuning

- num_virtual_tokens: Represents the number of virtual tokens added for each task, i.e., the length of the soft prompt. This length is usually set between 10-20 and can be adjusted according to the input length.
- prompt_tuning_init: Indicates the initialization method for prompt parameters. Options include random initialization (RANDOM), text initialization (TEXT), or other methods. In text initialization mode, specific text can be used to initialize prompt embeddings to accelerate convergence.

Prefix Tuning

- num_virtual_tokens: Similar to Prompt Tuning, it represents the number of constructed virtual tokens, with settings similar to Prompt Tuning.
- encoder_hidden_size: Represents the size of the multilayer perceptron (MLP) layer used for Prefix encoding, typically matching the hidden layer size of the model.

LoRA

- r: The rank size, used to control the complexity of the update matrix. Typically, smaller values such as 4, 8, 16 can be chosen. For small datasets, a smaller r value may be needed.
- lora_alpha: The scaling factor, used to control the size of the LoRA weights, typically inversely proportional to r to maintain consistency in weight updates.
- lora_dropout: The dropout rate for the LoRA layer, used for regularization to prevent

overfitting. It can be set to a small value, such as 0.01.

- target_modules: Specifies the modules in the model where LoRA is to be applied, such as the query, key, value matrices in the attention mechanism. Specific modules can be chosen for fine-tuning, or all linear layers can be fine-tuned.

It is important to note that specific parameter settings may vary depending on the model, dataset, and specific task. Therefore, adjustments may be necessary based on experimental results in practical applications.

4.5.2 PEFT Applications

In real-world applications, a large amount of **vertical domain data** is stored in tabular form in relational databases. Not only is the quantity large, but the domains covered by tabular data are also very broad, involving finance, healthcare, business, climate science, and many other fields. However, the proportion of tabular data in the pre-training corpus of large language models is usually very small, which leads to poor performance on downstream tasks related to tabular data. Fine-tuning large language models to adapt to tabular data is a typical application of parameter-efficient fine-tuning, which has profound practical significance. This section will introduce two cases of applying parameter-efficient fine-tuning technology to table data querying and table data analysis.

1. Table Data Querying

Table data querying is a necessary step in processing and analyzing table data. To query table data, professional Structured Query Language (SQL) code needs to be written. However, SQL code usually involves complex query logic and strict syntax rules, and even professionals need to spend a lot of time writing it. For beginners, mastering and using SQL proficiently may take months or even longer to learn and practice. Writing

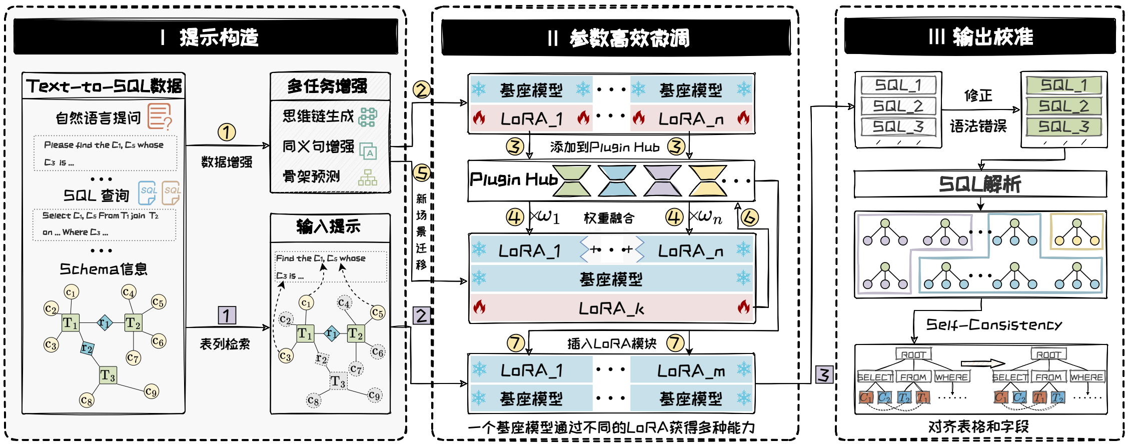


Figure 4.10: Schematic diagram of FinSQL.

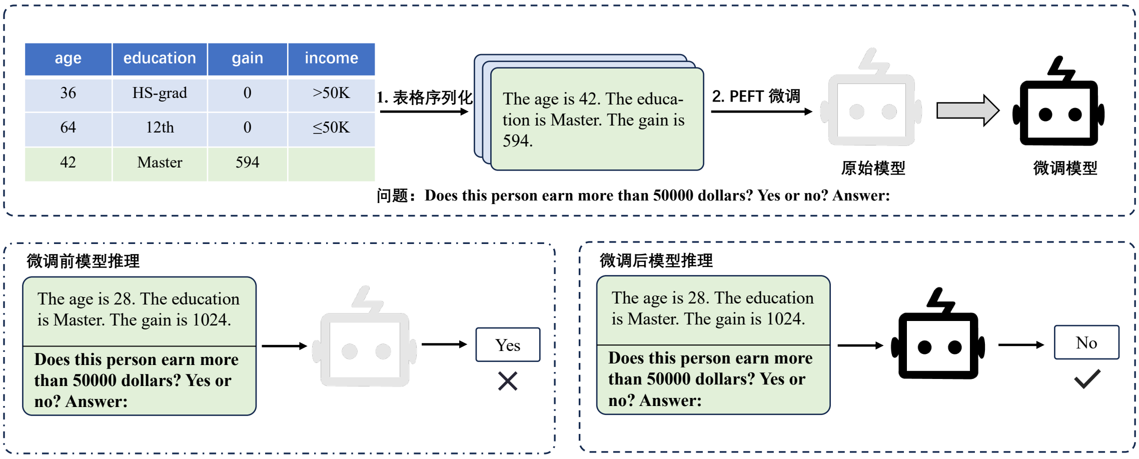
SQL code sets a high threshold for table data querying. To lower this threshold, the Text-to-SQL technology, which automatically translates natural language text into SQL code, has received widespread attention. It can shorten the SQL writing time from minutes to seconds, allowing data scientists to focus on analysis and modeling, speeding up data-driven decision-making, and enabling a wider audience to operate and manage databases, significantly improving data analysis efficiency and allowing more people to mine and utilize the value of data. The powerful code generation capabilities of large language models have brought new opportunities to Text-to-SQL technology. However, in many vertical fields, such as finance, it is difficult to collect enough data for fine-tuning. Full parameter fine-tuning with a small amount of data can easily lead to overfitting of large models, so using PEFT technology for partial parameter fine-tuning is very suitable for Text-to-SQL tasks in vertical fields such as finance.

Aiming at the financial vertical field, FinSQL [51] proposes a training and inference framework for Text-to-SQL in the financial vertical field. As shown in Figure 4.10, the framework includes three parts: prompt construction, parameter-efficient fine-tuning, and output calibration. First, prompt construction enhances the original data through different strategies and constructs an efficient retriever to retrieve tables and fields related to user queries. Then, PEFT technology is used to fine-tune the base model, and multiple LoRA modules are fused through LoRAHub to improve performance in few-shot scenarios. Finally, the output calibration module corrects syntax errors in the generated SQL and uses the Self-Consistency method to select consistent SQL. Compared with baseline methods, FinSQL not only significantly improves accuracy and efficiency but also demonstrates strong capabilities in handling complex financial queries, providing powerful technical support for data analysis and decision support in the financial field.

2. Table Data Analysis

After completing the data query operation, further analysis can be performed on the queried data. However, the characteristics of table data—lack of locality, containing various data types, and relatively fewer features—make it difficult for traditional deep learning methods to be directly applied. The parameters of large language models encode a large amount of prior knowledge, effectively utilizing this knowledge can compensate for the lack of features in table data. However, to achieve better performance, a small amount of labeled data is still needed to adapt large models to table tasks. However, fine-tuning models with a small amount of data can easily lead to overfitting. Since PEFT only fine-tunes part of the parameters, it can effectively reduce the risk of overfitting, making the performance of large language models on table data more robust.

For example, TabLLM [17] proposes a few-shot table data classification framework based on large language models. Figure 4.11 shows the framework diagram of this method. The framework serializes table data into natural language strings and attaches a brief description of the classification problem to prompt the large language model. In the few-shot setting, LoRA is used to fine-tune the large language model on some labeled samples.



age	education	gain	income
36	HS-grad	0	>50K
64	12th	0	≤50K
42	Master	594	

The age is 42. The education is Master. The gain is 594.

问题：Does this person earn more than 50000 dollars? Yes or no? Answer:

The age is 28. The education is Master. The gain is 1024.

Does this person earn more than 50000 dollars? Yes or no? Answer:

Figure 4.11: Framework diagram of TabLLM.

After fine-tuning, TabLLM outperforms deep learning-based table classification baseline methods on multiple benchmark datasets. Additionally, in the few-shot setting, TabLLM outperforms powerful traditional baselines such as gradient boosting trees, demonstrating strong few-shot learning capabilities.

This section introduced the practice and applications of parameter-efficient fine-tuning technology. First, we introduced the mainstream PEFT framework, HF-PEFT, and its usage methods, as well as related tips for PEFT. Finally, we demonstrated how PEFT technology helps large language models enhance performance on table data querying and analysis tasks, ensuring training efficiency while adapting large language models to vertical tasks.

References

[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 Pre-trained 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

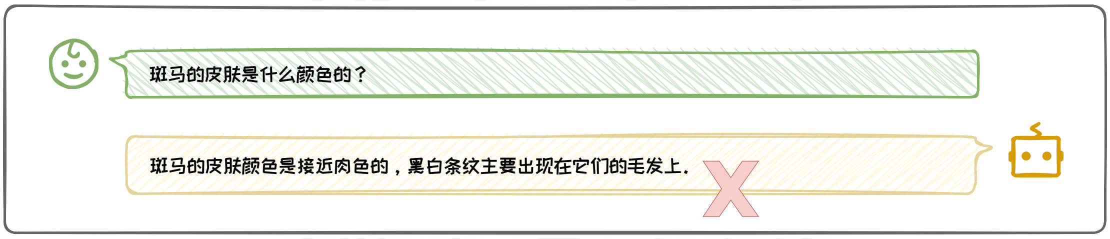
Model Edit

Pre-trained large language models may have issues such as **bias, toxicity, and knowledge errors**. To correct these issues, one could ”reforge” the large language model by retraining it with cleaned data, but the cost is too high and not cost-effective. Additionally, one could provide ”further education” to the large language model by using efficient fine-tuning techniques to inject new knowledge into the model, but due to the limited number of samples related to new knowledge, it is prone to overfitting and catastrophic forgetting, which is not worth the effort. Therefore, the technology of **model editing**, which corrects specific **knowledge points** in the model, has emerged. This chapter will introduce this emerging technology of model editing, first introducing the ideas, definitions, and properties of model editing, then introducing classic methods of model editing from both internal and external perspectives, followed by examples of specific methods of model editing, T-Patcher and ROME, and finally, the practical applications of model editing.

*This book is continuously updated, and the GIT Hub link is: https://github.com/ZJU-LLMs/Foundations-of-LLMs.

5.1 Introduction to Model Editing

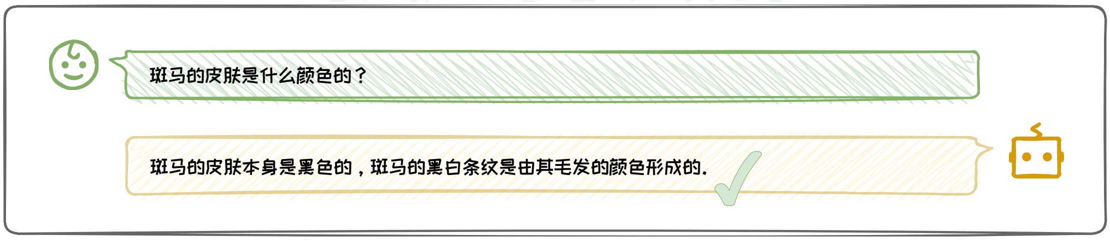
Large language models sometimes produce results that do not meet expectations, such as **bias**, **toxicity**, and **knowledge errors**. **Bias** refers to the model-generated content that includes stereotypes and social prejudices, which are unfair viewpoints. **Toxicity** refers to the model-generated content that contains harmful elements. **Knowledge errors** refer to the information provided by the model that does not align with the facts. For example, when asked ”What color is the skin of a zebra?”, ChatGPT incorrectly answered ”flesh-colored”, while in reality, the skin of a zebra is black, which is a knowledge error, as shown in Figure 5.1. If these issues are not corrected in time, they may seriously mislead people.



斑马的皮肤是什么颜色的？

斑马的皮肤颜色是接近肉色的，黑白条纹主要出现在它们的毛发上.

(a) Model before editing.



斑马的皮肤是什么颜色的？

斑马的皮肤本身是黑色的，斑马的黑白条纹是由其毛发的颜色形成的.

(b) Model after editing.

Figure 5.1: Example of knowledge errors in large language models.

To correct these issues, two approaches can be considered: retraining and fine-tuning. **Retraining** refers to the process of retraining the model with data that has been cleansed to correct biases, remove toxicity, and rectify knowledge errors. This method can fundamentally repair the model’s erroneous outputs. However, data cleansing is costly, and due to the frequent updates in knowledge, it is impossible to ensure that the cleansed data is always perfect. Moreover, retraining consumes significant computational resources, and if the model were retrained every time an error is discovered, it would be counterproductive. **Fine-tuning** involves further adjusting the model parameters based on a pre-trained model to address its errors. Despite the existence of various efficient parameter fine-tuning methods, directly adjusting a model with billions of parameters still incurs high training costs. Additionally, since the new knowledge samples required to correct model errors are limited, directly fine-tuning the model with samples related to the knowledge points can easily lead to overfitting and catastrophic forgetting. Therefore, neither retraining nor fine-tuning is suitable for practical bias correction, toxicity removal, and knowledge error correction in large language models.

To avoid the drawbacks of retraining and fine-tuning methods, **model editing** has emerged. It aims to accurately and efficiently correct specific knowledge points in large language models, meeting the need for updates to specific knowledge points in large language models. This section will provide a preliminary introduction to model editing from the aspects of concepts, definitions, and properties.

5.1.1 Model Editing Philosophy

In ”The Three-Body Problem II: The Dark Forest,” the Wallfacer Shi Ensi and his wife jointly developed a device called the ”Thought Stamp,” with the purpose of instilling the firm belief of ”humanity will prevail” in the Space Navy. The principle of this machine is to modify the brain’s processing when exposed to specific information, causing it to output a positive response. The concept of model editing is roughly similar; it aims to quickly and effectively change the model’s behavior and output by adding or modifying model parameters.

The process of a model learning knowledge can also be compared to the process of how humans learn knowledge. First, in the process of experiencing the world, we are exposed to a vast amount of knowledge, which allows us to form our own knowledge system. This can be likened to the **pre-training** process of large language models. Secondly, we can choose to study specific disciplines to enhance our abilities in certain areas, which can be likened to the **fine-tuning** process of large language models. Additionally, in the process of communicating with others, we discuss specific knowledge to correct our own misconceptions, which can be likened to the concept of **model editing**.

These methods can all meet the need to ”correct large language models.” Unlike retraining and fine-tuning methods, model editing expects to achieve corrections to specific knowledge points in the model more **quickly and accurately**.

5.1.2 Model Editing Definition

At present, the field of model editing lacks a unified standard, and different studies have varying definitions for related concepts. This book unifies the concepts of **Knowledge Model Editing** (KME) [25] and **Knowledge Editing** (KE) [28] mentioned in different works into **Model Editing** (ME, Model Editing). Additionally, some studies use ”edit” [27] or ”fact” [17, 18] to refer to the specific objects of editing, and this book unifies these concepts into ”knowledge points.”

The goal of model editing can be summarized as: **to correct large language models to produce the desired outcomes without affecting other unrelated outputs**. The following is the definition of model editing in this book:

Definition 5.1 (Model Editing)

Define the model before editing as M, and the model after editing as M^*. Each editing session modifies a **knowledge point** k of the model, where the knowledge point k consists of a question x_k and its corresponding answer y_k. Thus, the goal of model editing can be represented by the following function:

$$M^*(x) = \begin{cases} y_k, & \text{if } x = x_k \text{ or } x \text{ is related to } x_k, \\ M(x), & \text{if } x \text{ is unrelated to } x_k. \end{cases}$$

♣

The determination of ”related” and ”unrelated” in the above definition pertains to the scope of model editing, which will be discussed in Section 5.1.3.

Figure 5.2 uses the knowledge point of ”zebra skin color” as an example to illustrate the concept of model editing. In this example, when asked ”What color is the skin of a zebra?”, the model before editing responds with the incorrect answer ”flesh-colored,” while the model after editing can provide the correct answer ”black.”

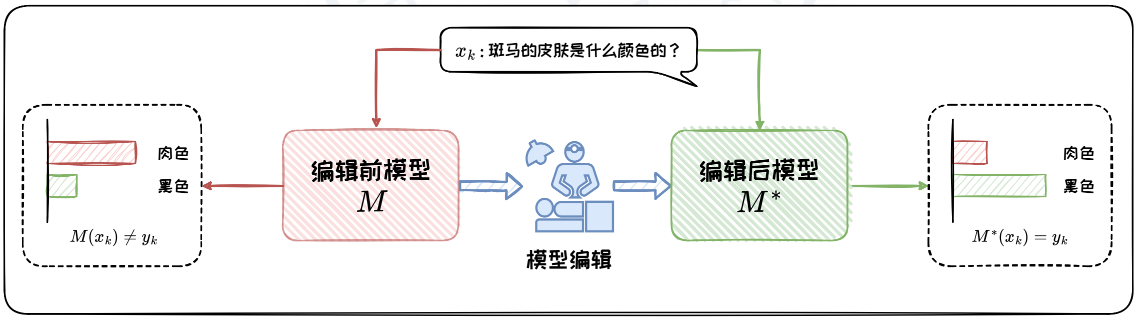


Figure 5.2: Conceptual diagram of model editing.

However, the actual process of model editing is far more complex than its theoretical definition. This is primarily due to the inherent interconnectedness of knowledge: when modifying the model’s understanding of a specific knowledge point, since that knowledge point may be associated with other knowledge points, it may affect the model’s understanding of other related knowledge points, thus creating a ”ripple effect.” Therefore, **how to precisely control the scope of model editing** becomes a key challenge. Precise and controllable model editing techniques need to meet a set of properties. These properties not only reflect the complexity of model editing but also provide important indicators for evaluating and improving editing methods. The following section will introduce the key properties of model editing.

5.1.3 Properties of Model Editing

The primary goal of model editing is to correct the model’s erroneous responses so that it provides the answers we expect. Building on this foundation, considering the inherent interconnectedness of knowledge, it is necessary to further precisely control the scope of model editing. In addition to this, ensuring the efficiency of model editing is also crucial. Therefore, the model editing process needs to be controlled from multiple aspects.

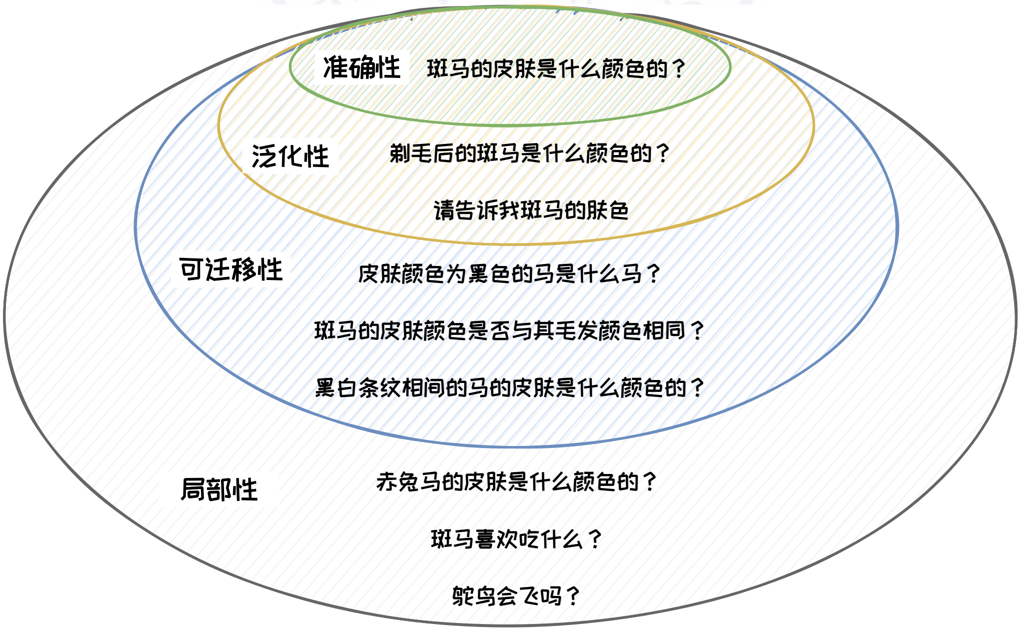

准确性
斑马的皮肤是什么颜色的？
泛化性
剃毛后的斑马是什么颜色的？
请告诉我斑马的肤色
可迁移性
皮肤颜色为黑色的马是什么马？
斑马的皮肤颜色是否与其毛发颜色相同？
黑白条纹相间的马的皮肤是什么颜色的？
局部性
赤兔马的皮肤是什么颜色的？
斑马喜欢吃什么？
鸵鸟会飞吗？

Figure 5.3: Relationship diagram of model editing properties.

Based on existing work[16, 25, 27, 28], this book summarizes the properties of model editing into five aspects, which are **Accuracy, Generality, Portability, Locality, and Efficiency**. Figure 5.3 visually displays the relationships between these properties through relevant examples.

1. Accuracy

Accuracy measures the effectiveness of direct modifications to a knowledge point k. As mentioned earlier, during model editing, many methods typically select a representative input-output pair (x_k, y_k) from knowledge point k to directly modify the model. Therefore, when evaluating the editing method, it is first necessary to assess whether the edited model M^* can accurately answer the question x_k, which is the question ”What color is the skin of a zebra?” in Figure 5.3. If M^* can output y_k, then this editing session is considered accurate. The following formula is used to represent whether an edit is accurate:

$$\text{Acc} = I(M^*(x_k) = y_k). \tag{5.1}$$

This formula is calculated using an indicator function $I(\cdot)$, where Acc is 1 only if the output $M^*(x_k)$ of the edited model on the target question x_k matches the answer y_k, otherwise it is 0. For multiple edits, the mean can be used to calculate the average accuracy rate.

Accuracy is the most basic requirement for model editing, ensuring that the edited model meets the designer’s expectations and accurately performs specific tasks. Only when the model can satisfy accuracy can it further meet other properties.

2. Generality

Generality is used to measure whether the edited model can adapt to other expressions of the target question x_k, that is, to determine whether the model can provide a unified target answer y_k when faced with questions that are semantically similar to x_k. The generality issue corresponds to the yellow part of the questions ”What color is a shaved zebra?” and ”Please tell me the skin color of a zebra,” both of which have the correct answer ”black” in Figure 5.3.

To assess the generality of the edited model, researchers typically construct a **generality dataset** $D_G = \{(x_i, y_k)\}_{i=1}^{|D_G|}$, where x_i are questions semantically similar to x_k, and their answers are y_k. The following formula is used to quantify the generality of the edited model:

$$\text{Gen} = \frac{1}{|D_G|} \sum_{i=1}^{|D_G|} I(M^*(x_i) = y_k). \tag{5.2}$$

When the value of Gen is 1, it indicates that the edited model can correctly answer all questions in D_G, which is the best generality. **Ensuring the generality of the edited model can prevent the model from overfitting to specific inputs**, thus ensuring the model’s understanding of the knowledge point.

3. Portability

Portability refers to the ability of the edited model to transfer a specific knowledge point k to other related issues. To assess the portability of the edited model, the most important step is to **construct a portability dataset**, which can be used to evaluate the model’s adaptability to issues that are indirectly related to k.

The portability dataset is represented as $D_P = \{(x_i, y_i)\}_{i=1}^{|D_P|}$, where x_i are questions related to x_k but have different answers, y_i, as their corresponding answers. x_i can be expressed in various forms, including reverse questions, reasoning questions, and entity substitution questions. As shown in the blue part of Figure 5.3, ”What kind of horse has black skin?” is a reverse question, ”Is the skin color of a zebra the same as its hair color?” is a reasoning question, and ”What is the skin color of a horse with black and white stripes?” is an entity substitution question. The answers to these three questions are not ”black”. The questions in the portability dataset D_P do not overlap with those in the generality dataset D_G, and the answers to the questions are different, i.e., $y_i \neq y_k$. The following formula is used to quantify the portability of the edited model:

$$\text{Port} = \frac{1}{|D_P|} \sum_{i=1}^{|D_P|} I(M^*(x_i) = y_i). \tag{5.3}$$

When the value of Port is 1, it indicates that the model can correctly answer all questions in the portability dataset, which is the best portability. **Portability examines the model's ability to transfer knowledge**, which is crucial for the practical application of model editing. However, most model editing methods often fail to achieve good portability, which is also a challenge in model editing.

4. Locality

Locality requires that the edited model does not affect the output of other unrelated issues. Locality issues correspond to the gray part of Figure 5.3, with questions such as ”What color is the skin of a Red Hare horse?” and ”What do zebras like to eat?”. Generally, researchers will select some questions unrelated to knowledge point k from common-sense datasets for locality assessment. The locality dataset is defined as $D_L = \{(x_i, M(x_i))\}_{i=1}^{|D_L|}$, where x_i are questions unrelated to x_k. The following formula is used to quantify the locality of the edited model:

$$\text{Loc} = \frac{1}{|D_L|} \sum_{i=1}^{|D_L|} I(M^*(x_i) = M(x_i)). \tag{5.4}$$

When the value of Loc is 1, the locality of the edited model is the best, at this point, the edited model's answers to all questions in the locality dataset are consistent with those before editing. **Ensuring locality can reduce the side effects of model editing**, which is an important improvement of model editing compared to naive fine-tuning.

5. Efficiency

Efficiency primarily considers the time cost and resource consumption of model editing. In practical applications, models may need to be updated and corrected frequently, which requires the editing process to be sufficiently fast and resource-friendly. Additionally, for a large number of editing tasks, the processing efficiency of different methods varies; some methods support batch parallel editing[2, 6, 18–20], while others require sequential processing[4, 13, 17]. **Efficiency directly affects the feasibility and practicality of model editing**.

When evaluating model editing methods, it is necessary to find a balance among these five properties. An ideal editing method should ensure accuracy while maximizing generality, portability, and locality, while maintaining efficiency.

5.1.4 Common Datasets

The previous discussion explored the definition and properties of model editing. Next, we will introduce some specific datasets used in previous research, which can be used to test and compare different model editing methods.

In the related research on model editing, the most widely used dataset is the **zsRE dataset**[14] proposed by Omer Levy and others. zsRE is a question-answering task dataset that assesses the model's editing capabilities for specific relationships (such as "place of birth" or "profession" between entities) through crowdsourced templated questions. In model editing, the zsRE dataset is used to check whether the model can accurately identify relationships in the text and update related knowledge based on new inputs, thereby evaluating the **accuracy** of model editing methods.

Table 5.1: Summary Table of Datasets Related to Model Editing.

Dataset	Type	Training Set Size	Test Set Size	Input	Output
zsRE[14]	Knowledge Association	244,173	244,173	Fact Statements	Entities
COUNTERFACT[17]	Knowledge Association	N/A	21,919	Fact Questions	Entities
WikiGen[19]	Text Generation	N/A	68,000	Wiki Paragraphs	Continuation
T-REx-100/-1000[6]	Knowledge Association	N/A	100/1,000	Fact Statements	Entities
ParaRel[4]	Knowledge Association	N/A	253,448	Fact Questions	Entities
NQ-SituatedQA[5]	Question Answering	N/A	67,000	User Queries	Answers
MQuAKE-CF/-T[30]	Knowledge Association	N/A	9,218/1,825	Multi-hop Questions	Entities
Hallucination[10]	Hallucination	N/A	1,392	Biographies	Biographies
MMEdit-E-VQA[3]	Multimodal	6,346	2,093	Image Questions	Answers
MMEdit-E-IC[3]	Multimodal	2,849	1,000	Image Descriptions	Descriptions
ECBD[23]	Knowledge Association	N/A	1,000	Entity Completion	Referenced Entities
FEVER[2]	Fact Checking	104,966	10,444	Fact Descriptions	Binary Labels
ConvSent[20]	Sentiment Analysis	287,802	15,989	Subjective Opinions	Sentiments
Bias in Bio[11]	Biographical	5,000	5,000	Biographical Sentences	Professions
VitaminC-FC[20]	Fact Checking	370,653	55,197	Fact Descriptions	Binary Labels
SCOTUS[10]	Classification	7,400	931	Court Documents	Controversial Topics

In 2023, [17] introduced the **COUNTERFACT dataset**, which has been widely adopted in subsequent research. COUNTERFACT is designed to distinguish between two types of knowledge modifications: one is the superficial change of vocabulary, which is simply the replacement or structural adjustment of words in the text and does not affect the substantive content of the information; the other is a significant and generalized modification of foundational factual knowledge, which fundamentally changes the facts or information described in the text and typically requires a deeper level of understanding and processing capabilities. COUNTERFACT can assess the model's comprehension and response to edited knowledge, thereby measuring the model's **generality** and **locality**. The literature [27], based on the ZsRE and COUNTERFACT datasets, used GPT-4 to generate reverse questions, reasoning questions, and entity substitution questions corresponding to the issues, constructing a **portability** dataset.

Furthermore, researchers have developed specialized datasets for different domains and tasks, such as **Hallucination**[10] for correcting hallucinations in GPT language models, and **ConvSent**[20] for evaluating the effectiveness of models in modifying the sentiment of dialogue agents towards specific topics. Table 5.1 summarizes some datasets related to model editing in terms of data type, sample size, and input-output formats. These datasets cover a variety of types from fact-checking, knowledge association, to specific domains, demonstrating the potential applications of model editing technology in different scenarios.

This section introduces the definition, properties, evaluation methods, and common datasets of model editing, providing a detailed explanation and description of the model editing task. Next, Section 5.2 will provide a systematic overview of model editing methods, dividing them into external expansion methods and internal modification methods, and introducing representative works for each category. Sections 5.3 and 5.4 will delve into the T-Patcher method of external expansion and the ROME method of internal modification, respectively, helping readers to understand the research process of model editing methods more meticulously. Finally, Section 5.5 will comprehensively introduce the practical applications of model editing, providing examples of solutions for illustration.

5.2 Classic Methods of Model Editing

In adventure games, a brave hero in need of an upgrade can be modified from both external and internal aspects.

External modifications mainly involve acquiring new props and equipment that can endow the hero with new abilities while retaining their original skills. Internal modifications are equivalent to self-training, enhancing attributes such as intelligence, strength, and magic power to achieve self-improvement. If we compare a large language model to a hero in an adventure game, then model editing can be seen as a method to meet the ”upgrade” needs, which can be considered from both external and internal perspectives. This paper refers to existing work[16, 25, 27, 28] and categorizes existing editing methods into **external expansion methods** and **internal modification methods**. In general, external expansion methods involve designing specific training programs that allow the model to learn new information while maintaining its original knowledge. Internal modification methods adjust specific layers or neurons within the model to achieve precise control over the model’s output. Figure 5.4 provides a classification of model editing methods: external expansion methods include **knowledge caching methods** and **additional parameter methods**, while internal modification methods include **meta-learning methods** and **targeted editing methods**. This section will introduce each type of editing method in detail.

ZHEJIANG UNIVERSITY 1897

5.2.1 External Expansion Methods

The **core idea of external expansion methods is to store new knowledge in additional external parameters or external knowledge bases, which are used together with the original model as the edited model**. This method is particularly suitable for pre-trained large language models with good scalability because it provides enough space to accommodate a large number of parameters and store more new knowledge. Moreover, this method does not change the original model parameters, reducing interference with the internal pre-trained knowledge of the model.

Based on whether the external components are directly integrated into the model’s own reasoning process, external expansion methods can be further divided into **knowl-**

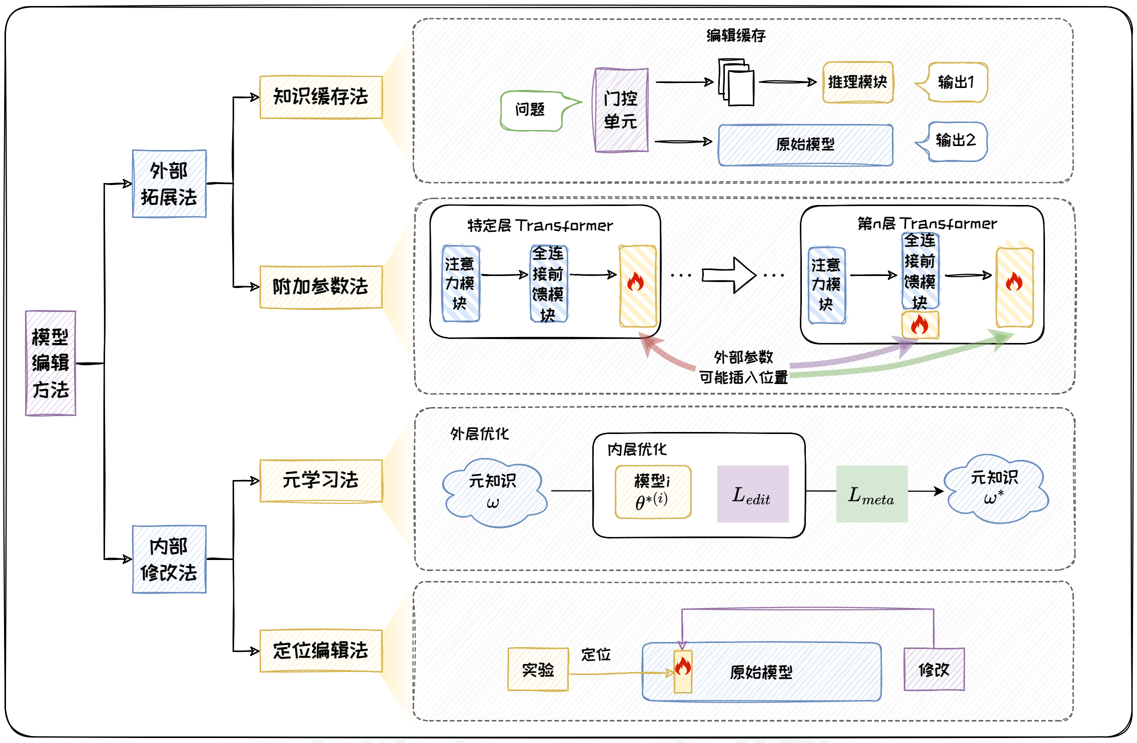


Figure 5.4: Classification diagram of model editing methods.

edge caching methods and **additional parameter methods**. Continuing the adventure game metaphor, knowledge caching is like the hero's skill book, which can be consulted to obtain specific knowledge when needed; while additional parameters are like upgradable equipment that directly enhances the hero's overall abilities.

1. Knowledge Caching Method

The Knowledge Caching Method consists of three main components: the gating unit, the editing cache, and the reasoning module. The **editing cache** acts as a knowledge repository for storing knowledge that needs to be modified, specified by users in various forms. The **gating unit** determines the relevance of the input question to the knowledge in the editing cache and can be trained through tasks such as classification[20] or noise contrast estimation[21]. The **reasoning module** takes the original input question and the knowledge from the editing cache as inputs and learns to predict the desired results through supervised training.

During reasoning, the gating unit first determines whether the input question is related to any knowledge point in the editing cache. If it is related, the knowledge point is retrieved from the editing cache and handed over to the reasoning module along with the input, which then provides the answer; if it is not related, the original model is used to reason and provide the answer. Figure 5.5 is a schematic diagram of the Knowledge Caching Method, where it is assumed that the editing cache stores knowledge related to the skin color of zebras. The question ”Can an ostrich fly?” is judged by the gating unit as unrelated to any knowledge point in the editing cache, so the answer is deduced by the original model; the question ”What kind of horse has black skin?” is judged as related to the knowledge point ”zebra skin color,” so the modified answer is provided by the trained reasoning module.

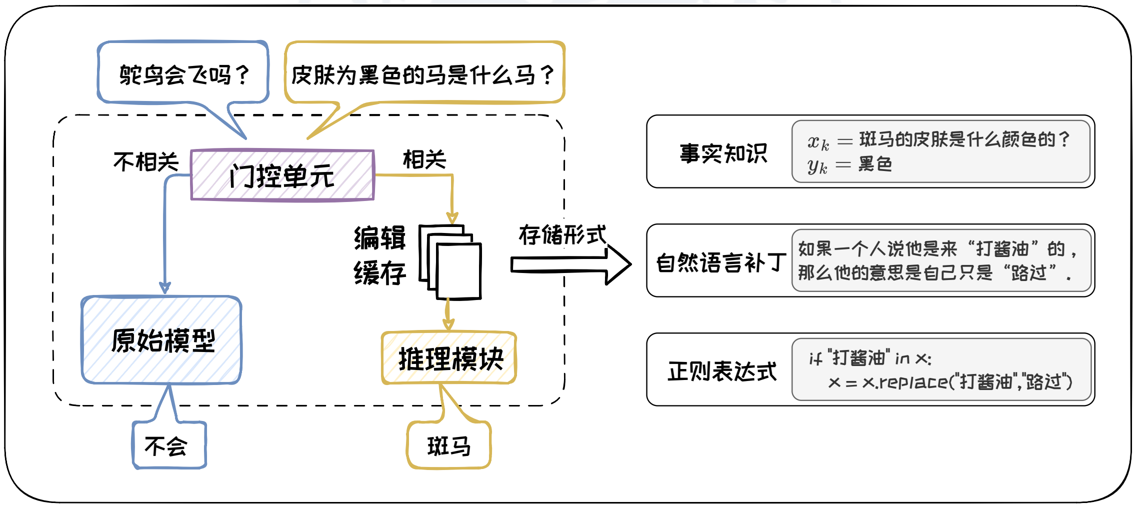


Figure 5.5: Schematic diagram of the Knowledge Caching Method.

Additionally, the storage form of knowledge points in the editing cache can be divided into three types: factual knowledge, natural language patches, and regular expressions. **Factual knowledge** stores editing instances as question-answer pairs (x_k, y_k), suitable for factual questions with clear answers, and **SERAC**[20] is a representative method. **Nat-**

ural language patches Language Patch[21] describe editing knowledge in the form of ”if... then...” sentences, similar to prompts. This storage form is suitable for correcting the model’s understanding of non-literal statements in natural language and is easy for people to create, edit, or remove patches, allowing the model to continuously correct its outputs through human feedback. **Regular expressions** are a text matching and replacement technique that uses specific patterns to identify and modify specific parts of the text, suitable for precise textual semantic replacement. This is an early primitive method, but due to its complexity and low generalizability, it is not commonly used in model editing.

The Knowledge Caching Method directly retrieves information from the editing cache without relying on gradient information from target labels, thus simplifying the model editing process and making it more efficient and direct. However, this method of obtaining knowledge from the outside is equivalent to having the large language model seek external help, rather than truly internalizing new knowledge as its own. The Additional Parameter Method improves upon this limitation.

2. Additional Parameter Method

Compared to the Knowledge Caching Method, the Additional Parameter Method integrates external parameters into the model structure, effectively utilizing and expanding the model’s functionality. The idea behind this category of methods is similar to the **Parameter Attachment Method** in parameter-efficient fine-tuning, where external parameters are inserted into specific locations within the model, freezing the original model and training only the newly introduced parameters to correct the model’s output, as shown in Figure 5.6.

Specifically, different methods insert external parameters into different locations within the model. For example, **CALINET**[6] and **T-Patcher**[13] achieve this by modifying the **fully connected feed-forward modules** of the last Transformer layer in the model. CALINET first identifies the original model's knowledge errors through a comparative knowledge assessment method, then adds a new parameter matrix to the last fully connected feed-forward module and trains the new parameters by minimizing the loss on the calibration data to correct the model's errors. T-Patcher is similar, introducing a limited number of trainable neurons in the last fully connected feed-forward module of the original model, with each neuron corresponding to a knowledge point. A detailed introduction to T-Patcher will be given in Section 5.3.

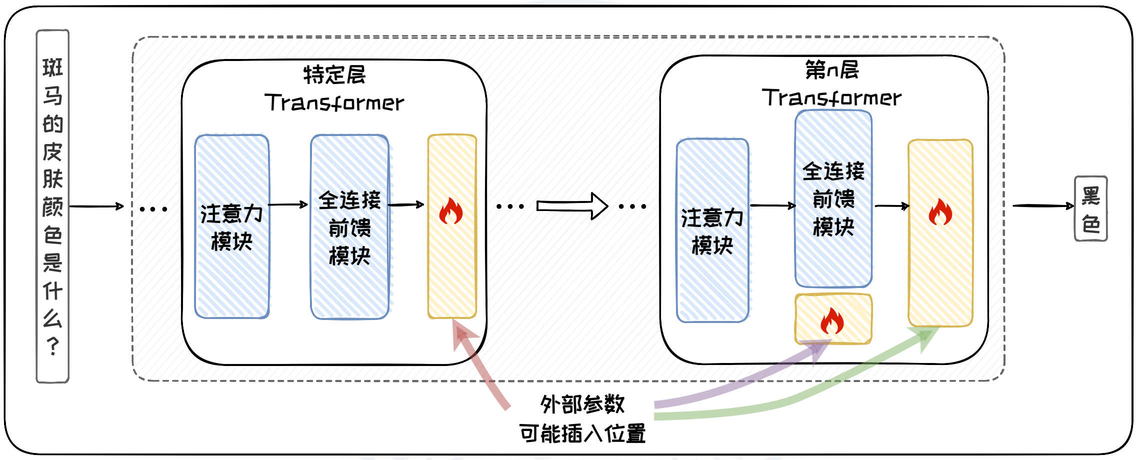


Figure 5.6: Schematic diagram of the Additional Parameter Method.

GRACE[10] inserts external parameters in the form of adapters into specific Transformer layers within the model, with the insertion position varying depending on the model, such as the second-to-last layer of BERT and the 36th layer of GPT2-XL. The adapter is a key-value store that caches erroneous knowledge (Keys) and corresponding correction values (Values), referred to as a ”codebook”. In the codebook, each piece of erroneous knowledge has a corresponding correction value and a delay radius for matching similar inputs, and the codebook is continuously updated over time. The delay radius is used to determine if the current input is similar to any errors in the codebook; if so, the corresponding correction value is applied for editing.

In summary, the Knowledge Caching Method effectively assists the model in quickly locating and retrieving the latest information within a vast knowledge system by introducing an editing cache mechanism; the Additional Parameter Method achieves fine-tuning of specific model outputs by introducing additional parameters. The core advantage of these two methods is their **minimal intervention in the original model**, ensuring the **locality** of model editing.

However, the actual effectiveness of external expansion methods largely depends on the storage and retrieval capabilities of knowledge, which can lead to increased storage resource requirements. Therefore, in practical applications, we need to strike a balance between **ensuring model locality** and **addressing storage limitations**.

ZHEJIANG UNIVERSITY 1897

5.2.2 Internal Modification Methods

Unlike external expansion methods that require additional storage space, internal modification methods allow the model to directly optimize itself without increasing the physical storage burden. **Internal modification methods aim to inject new knowledge into the model by updating the internal parameters of the original model**, which can optimize the model's self-learning and adaptation capabilities, improve its performance on specific tasks, rather than just staying on the surface of knowledge accumulation.

Internal modification methods can be divided into **meta-learning methods** and **targeted editing methods**. Among them, meta-learning methods acquire meta-knowledge through "learning how to learn" and then achieve model editing based on meta-knowledge; targeted editing methods focus on modifying local model parameters, first identifying the model parameters most related to the target knowledge, and then only updating these specific parameters, saving the cost of updating the model through a "locate and then edit" strategy.

1. Meta-Learning Method

Meta-learning refers to the process of a model "learning how to learn" (Learning to Learn). Meta-learning-based model editing methods aim to let the model "learn how to edit" (Learning to Edit), with the core idea of enabling the model to extract general knowledge from a series of editing tasks and apply it to unseen editing tasks. This part of the knowledge is called **meta-knowledge** ω [12]. Meta-knowledge is the knowledge that the model can use before editing, including various forms such as optimizer parameters[24], hyper-networks[2, 19], etc. The training process of meta-knowledge is called **meta-training**, and its goal is to obtain a better meta-knowledge ω, so that subsequent edits can quickly converge with only a small number of samples.

The meta-training process can be seen as a bi-level optimization problem[12], as shown in formula 5.5. Bi-level optimization (Bilevel Optimization) is a hierarchical optimization framework, where the inner optimization problem can serve as a constraint for the outer optimization problem. The meta-learning editing method is shown in Figure 5.7. In the figure, the inner optimization is the optimization of the model on different editing tasks, and the outer optimization is the comprehensive optimization of meta-knowledge on editing tasks in the validation set.

$$\omega^* = \arg\min_{\omega} \sum_{i=1}^{n} L_{\text{meta}}(\theta^{*(i)}(\omega), \omega, D_k^{\text{val}(i)})$$

$$\text{s.t. } \theta^{*(i)}(\omega) = \arg\min_{\theta} L_{\text{edit}}(\theta^{(i)}, \omega, D_k^{\text{train}(i)})。 \tag{5.5}$$

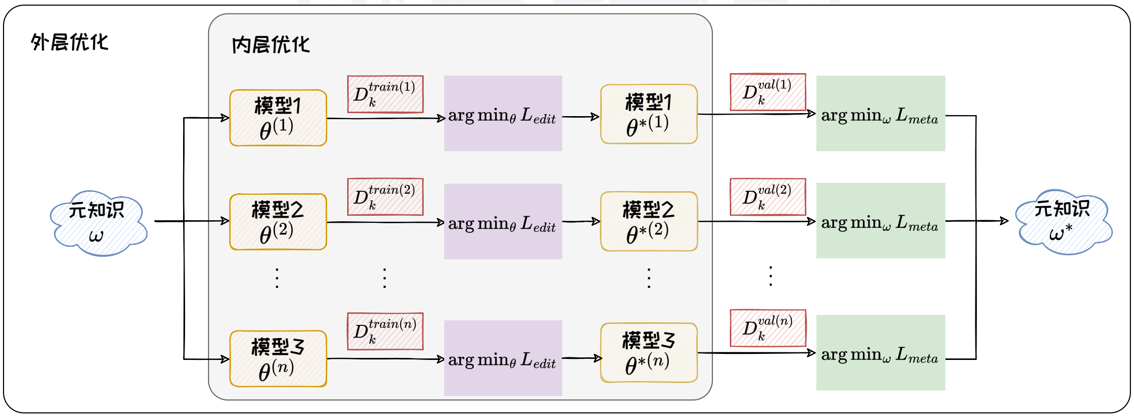


Figure 5.7: Schematic diagram of the Meta-Learning Method.

The inner optimization problem aims to learn the model parameters for specific editing tasks. In the inner layer, let there be a total of n editing tasks. For the i-th editing task ($i \in [1, n]$ and $i \in \mathbb{Z}^+$), let $D_k^{(i)}$ represent the dataset related to the knowledge point k involved in that editing task. $D_k^{(i)}$ is a collection of question-answer pairs (x_k, y_k), which can be divided into the training set $D_k^{\text{train}(i)}$ and the validation set $D_k^{\text{val}(i)}$. Let the original model parameters be $\theta^{(i)}$, and the optimized parameters after editing on $D_k^{(i)}$ be $\theta^{*(i)}$. The inner layer optimization is the process of updating the model parameters for the corresponding editing task based on the meta-knowledge ω. In the inner layer optimization, L_{edit} is the loss function related to each editing task, which means that we hope to minimize the prediction error of the edited model $\theta^{*(i)}$ on $D_k^{\text{train}(i)}$, such as the cross-entropy function in classification tasks.

The outer optimization problem aims to learn meta-knowledge that can be generalized to other editing tasks. The outer layer optimization is usually performed on the validation set $D_k^{\text{val}(i)}$, hoping to update the meta-knowledge ω based on the loss of the model $\theta^{*(i)}$ on its corresponding editing validation set. In the outer layer optimization, L_{meta} is the loss function related to meta-learning, which means that we hope to find a meta-knowledge ω that minimizes the sum of prediction errors of the model optimized on all $D_k^{\text{val}(i)}$.

ENN[24] regards meta-knowledge as optimizer parameters. By updating the optimizer parameters, it makes the training of model parameters in subsequent edits more efficient, thus enabling the model to learn how to edit quickly. ENN introduces an editing function and a gradient descent editor. In the inner optimization process, each time a few gradient updates are made on the model parameters for an editing task, and then the updated parameters are used to update the optimizer parameters. However, ENN is designed for small networks like ResNet, and when applied to large models, it faces high training costs and other issues.

To expand the application of meta-learning methods on large-scale model architectures, **KE**[2] treats meta-knowledge as a hypernetwork and proposes a method to train the hypernetwork to learn the model parameter update values. During the training of the hypernetwork, the loss function consists of two parts: one to ensure accuracy and the other to ensure locality, with boundary values set to represent the strictness of the constraints. The well-trained hypernetwork generates model parameter update values based on input questions, enabling the model to output the desired results under specific inputs while keeping other predictions unchanged.

To further enhance the versatility of hypernetworks for large language models, MEND[19] optimizes the process of hypernetwork-assisted model parameter updates through low-rank factorization. First, it utilizes the rank-1 property of gradients in fully connected layers, decomposing the loss function's gradients with respect to each layer's parameters into the product of two vectors. Then, the hypernetwork takes the decomposed vectors as inputs and outputs the edited new vectors. Finally, these new vectors are multiplied again to obtain new parameter gradients, and the final model parameter update values are calculated through a learnable scaling factor. MEND efficiently edits large models with fewer parameters, saving computational resources and memory.

In summary, meta-learning-based editing methods improve the model's adaptability and generalization when facing new editing tasks by ”learning how to edit.” They can extract general knowledge from a series of editing tasks, so that when encountering unseen editing tasks, they can quickly converge with only a small number of sample trainings, thus saving computational resources and time. However, meta-learning editing methods also have their shortcomings. The training process is relatively complex, and when applied to large models, they often face high training costs. Although KE and MEND have optimized the parameter update process using hypernetworks and gradient low-rank factorization techniques, reducing the demand for computational resources, there is still a need to further improve their adaptability and efficiency for more complex tasks and larger-scale models. In addition, meta-learning editing methods update model parameters from a global perspective, and even with the addition of locality-related loss functions, they may still impact the model's original knowledge, leading to model instability.

2. Targeted Editing Method

Compared to meta-learning methods, the targeted editing method modifies the local parameters of the original model. It first locates the position of the parameters that need to be modified and then makes changes to those specific parameters. To achieve localization, it is necessary to understand the storage mechanism of knowledge within large language models. Currently, exploration of the knowledge storage mechanism mainly relies on qualitative experiments. Through experiments on a 16-layer Transformer language model, it has been concluded that the fully connected feed-forward modules in **Transformers can be regarded as key-value stores for knowledge** [8]. In this context, the input to the fully connected feed-forward module is referred to as the query vector, representing the prefix of the current sentence; the vectors in the upper projection matrix of the fully connected feed-forward module are called key vectors, and the vectors in the lower projection matrix are called value vectors.

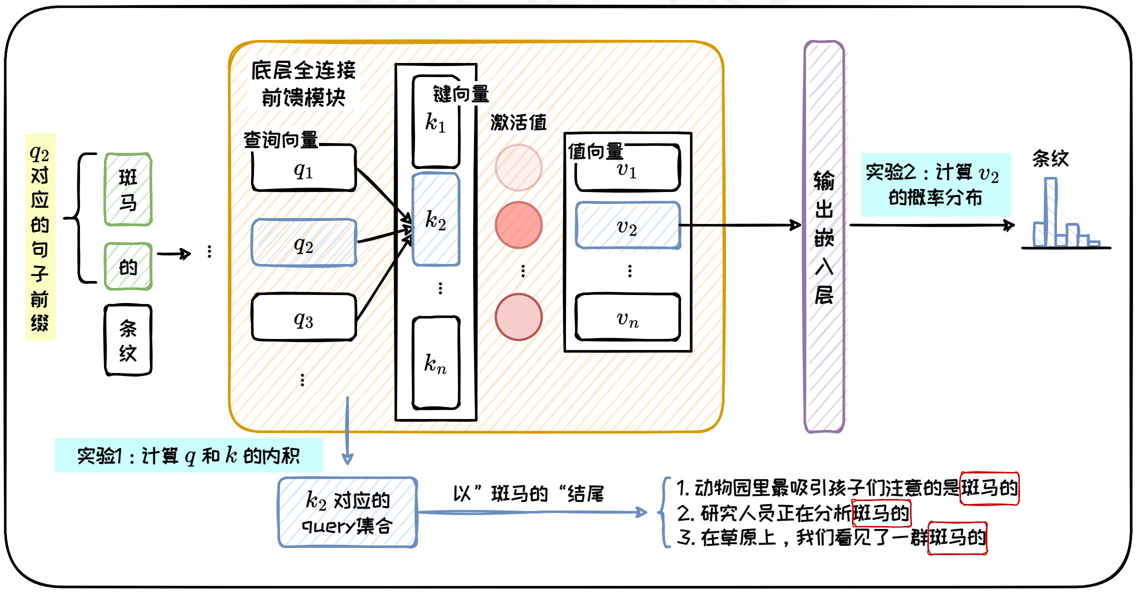


Figure 5.8: Transformer can be seen as a key-value store.

Figure 5.8 illustrates the experimental process conducted by the literature [8] on the fully connected feed-forward modules at the bottom layer of the model. In Experiment 1, each key is separately dot-producted with all queries $(q_1, q_2, q_3, ...)$, with k_2 used as an example in the figure to demonstrate this process. After performing the dot-product operation of k_2 with all queries, it is found that k_2 has a larger activation value with several queries, including q_2. Collecting the queries with larger activation values corresponding to each key as a set, it is observed that these queries have the same or similar patterns in their prefixes. For example, in the figure, the queries corresponding to k_2 all end with ”zebra’s”, which means that k_2 stores the text pattern related to ”zebra’s”. In Experiment 2, each key’s corresponding value is multiplied by the output embedding layer matrix and transformed into a probability distribution using the softmax function, then the correlation between these distributions and the next word of the corresponding query is compared, and a high correlation is found. For example, in the figure, after performing the above operations with v_2 corresponding to k_2, it is found that the probability of the next word being ”stripes” is the highest, which matches the next word of q_2. Combining the results of Experiments 1 and 2, the literature [8] points out that in the fully connected feed-forward modules, the model summarizes the features of the sentence prefix represented by the current query through the key and finds the corresponding value, which is the probability distribution of the next word, through a key-value matching mechanism. In other words, the fully connected feed-forward modules in Transformers can be regarded as key-value stores for storing knowledge.

Based on the above conclusion, **KN** [4] proposed the concept of knowledge neurons. It defines each intermediate activation value in the fully connected feed-forward module as a knowledge neuron and believes that the activation of knowledge neurons is closely related to the expression of corresponding knowledge points. To evaluate the contribution of each neuron to the prediction of specific knowledge, KN inputs the masked text related to the knowledge point into the pre-trained model, obtains the hidden state, and then inputs it into the FFN of each layer of the model. Through attribution methods, it accumulates the gradient changes of each neuron in predicting the correct answer, thereby determining which neurons play a key role in the knowledge expression process. This method can not only identify neurons that significantly affect the expression of knowledge but also filter out those neurons that contribute less to the prediction of knowledge. After determining the contribution of knowledge neurons to knowledge prediction, the model's output can be edited by directly modifying the key vectors corresponding to specific knowledge neurons in the model, thus achieving the effect of model editing.

Additionally, ROME[17] designed a causal tracing experiment to further explore the relationship between the intermediate fully connected feed-forward modules and knowledge, optimizing the conclusions about the knowledge storage mechanism. In terms of editing methods, unlike KN, which locates and edits individual neurons in the fully connected feed-forward modules, ROME proposes to update the entire fully connected feed-forward module for editing. By applying ROME to edit the GPT-J model, it has shown good performance in accuracy, generality, and locality. Therefore, ROME has become a highly regarded model editing method in recent years, providing important references and guidance for future model editing and optimization work. We will introduce in detail the design of causal tracing experiments and editing methods in ROME in Section 5.4. MEMIT[18] expands on ROME to perform large-scale editing on different knowledge, capable of executing thousands of edits simultaneously.

In summary, the targeted editing method modifies the local parameters of large language models, allowing for precise updates and edits to specific knowledge points while maintaining the overall structure and performance of the model. Compared to other methods, the targeted editing method can maintain high accuracy, generality, and locality, and is applicable to various models.

5.2.3 Method Comparison

The various model editing methods mentioned above have their own advantages and disadvantages. This book refers to the experimental results in the literature[27] to compare the performance of mainstream model editing methods on different properties, as shown in Table 5.2. The table qualitatively represents the accuracy, generality, portability, and locality of the methods with the three levels of "high," "medium," and "low," and uses "✓" and "✗" to indicate the efficiency of the methods, i.e., whether they support batch editing. A "-" indicates that the test was not conducted.

Table 5.2: Comparison of Model Editing Methods.

		Method	**Accuracy**	**Generality**	**Portability**	**Locality**	**Efficiency**
External Expansion Methods	Knowledge Caching	SERAC	High	High	Low	High	✓
	Additional Parameter	CaliNET	Low	Low	-	Medium	✓
		T-Patcher	High	High	High	Medium	✗
Internal Modification Methods	Meta-Learning	KE	Low	Low	-	High	✓
		MEND	Medium	High	Medium	High	✓
	Targeted Editing	KN	Medium	Low	-	Medium	✗
		ROME	High	High	High	High	✗
		MEMIT	High	High	High	High	✓

From Table 5.2, it can be seen that among the external expansion methods, SERAC, which is based on knowledge caching, provides efficient editing capabilities without additional training, ensuring high accuracy, generality, and locality. It is suitable for rapid response and batch editing, but its portability is poor, and the editing cache and reasoning modules still need to be optimized. CaliNET and T-Patcher, which are based on additional parameters, offer direct editing capabilities for the model. However, CaliNET has poor adaptability to different models and data, while T-Patcher maintains high accuracy, generality, and portability but requires a higher demand for memory during batch editing.

In the internal modification methods, KE and MEND, which are based on meta-learning, treat meta-knowledge as a hypernetwork. By enabling the model to ”learn how to edit,” they improve generality and training efficiency and support batch editing. However, methods based on meta-learning have a more complex training process design and may be limited when applied to large models. KN, ROME, and MEMIT, which are based on targeted editing, focus on precisely locating and editing specific knowledge within the model. ROME and MEMIT can ensure high accuracy, generality, portability, and locality, but these two methods are mainly designed for decoder-only models, so their performance on encoder-decoder architecture models is not compared. In addition, MEMIT has been optimized for batch editing based on ROME, ensuring stability in other properties while meeting efficiency requirements.

This section has provided an overview and examples of different categories of model editing methods. These methods demonstrate various approaches to effectively correcting large language models, each with its advantages and limitations. In practical applications, the appropriate method should be selected based on requirements, available resources, and the desired editing effects. According to Table 5.2, T-Patcher and ROME perform relatively well in terms of accuracy, generality, portability, and locality. Therefore, the following two sections will use these two methods as representatives to explain the internal mechanisms of model editing in more detail.

5.3 Additional Parameter Method: T-Patcher

The additional parameter method introduces extra parameters into the model and trains this part of the parameters for specific knowledge editing. T-Patcher is a representative method of the additional parameter method, which adds additional parameters (called “patches”) to the fully connected feedforward layer of the last transformer layer of the model, and then trains the patches to achieve specific knowledge editing, as shown in Figure . knowledge editing, as shown in Figure 5.9. T-Patcher allows editing of the model without changing the overall architecture of the original model. In this section, T-Patcher is introduced in three aspects: **patch location**, **patch form**, and **patch realization**.

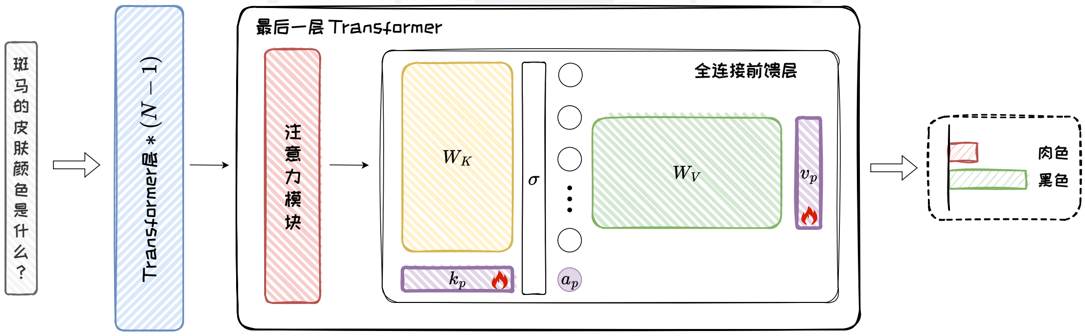


Figure 5.9: T-Patcher method (the purple block in the figure is the patch).

5.3.1 patch location

Adding patches to different locations in the model affects the effectiveness of model editing.T-Patcher treats the fully-connected feedforward layer as a **key-value store** and chooses to add the patch parameter to the fully-connected feedforward layer of the last Transformer layer. In this design, adding patches is equivalent to adding new memory cells to the key-value store, and by precisely controlling the activation of the patches, the T-Patcher is able to correct for specific inputs and minimize the impact on irrelevant inputs. In addition, due to the simple structure of the fully-connected feedforward layer, only a small number of parameters need to be added to enable efficient editing.

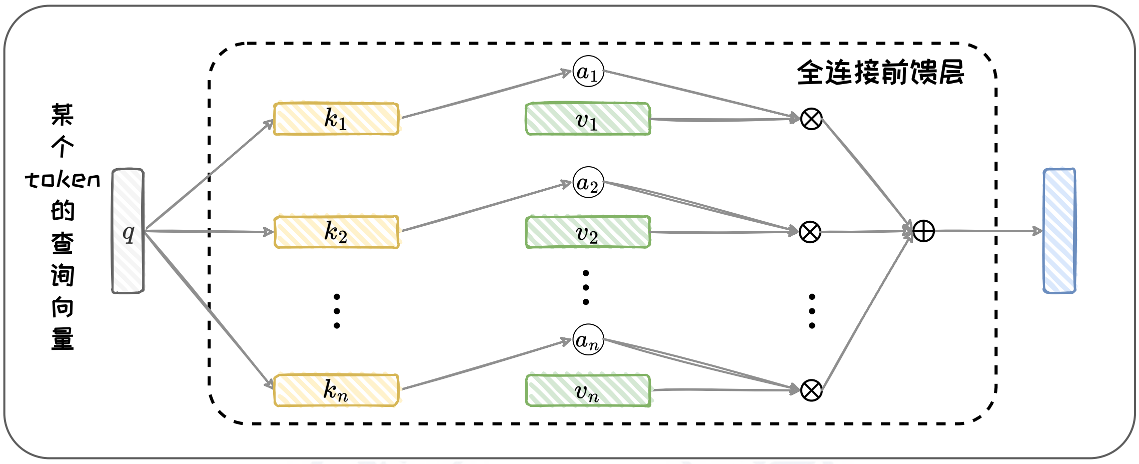


Figure 5.10: key-value store (omit activation function).

Specifically, T-Patcher treats the fully-connected feedforward layer as a key-value store as shown in Figure 5.10, which contains the key vector matrix $\boldsymbol{W}_{fc} = [\boldsymbol{k_1}, \boldsymbol{k_2}, \ldots, \boldsymbol{k_n}]$ and its bias vectors $\boldsymbol{b}_k$, the activation function σ and the value vector matrix $\boldsymbol{W}_{proj} = [\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n}]$ and its bias vector $\boldsymbol{b}_v$. Where each key vector corresponds to a specific pattern in the input text, such as an n-gram or a semantic topic, and each value vector is associated with the probability distribution of the model output.

The process of finding the corresponding knowledge in the query store is as follows: for some input Token, its query vector is $\boldsymbol{q}$. When $\boldsymbol{q}$ is input to the fully-connected feedforward layer, it is first multiplied with the matrix W_{fc} to compute the activation value vector $\boldsymbol{a}$, where each component represents the degree of association of $\boldsymbol{q}$ with the corresponding key vector k. Subsequently, $\boldsymbol{a}$ is multiplied with the matrix W_{proj} to obtain the output of the fully connected feedforward layer. This process can be viewed as a weighted summation of all the value vectors in the matrix W_{proj} using the activation values:

$$\boldsymbol{a} = \sigma(\boldsymbol{q} \cdot \boldsymbol{W_{fc}} + \boldsymbol{b}_k) \tag{5.6}$$

$$FFN(\boldsymbol{q}) = \boldsymbol{a} \cdot \boldsymbol{W_{proj}} + \boldsymbol{b}_v. \tag{5.7}$$

Based on the above view of key-value store, **the hidden layer dimension of the fully-connected feed-forward layer can be interpreted as the number of text patterns it "remembers"** , if more key-value pairs associated with different text patterns can be added to the fully-connected feed-forward layer, new factual information can be inserted into the model, thus realizing model editing. Editing. Therefore, T-Patcher**add extra parameters to the fully connected feedforward layer, i.e., add patches**. Moreover, T-Patcher adds patches only in the **last layer of the model** to ensure that the patches fully modify the model's output without being interfered with by other model structures. In the next section, we will detail what kind of patches should be added.

5.3.2 form of patch

Based on the viewpoint of key-value stores, T-Patcher designs the form of patches as key-value pairs. Specifically, T-Patcher adds additional key-value pair vectors as patches in the fully-connected feedforward layer, and realizes model editing by training the patch parameters. The form of the patch is shown in Figure 5.11, which mainly consists of a key vector $\boldsymbol{k_p}$, a value vector $\boldsymbol{v_p}$ and a bias term b_p.

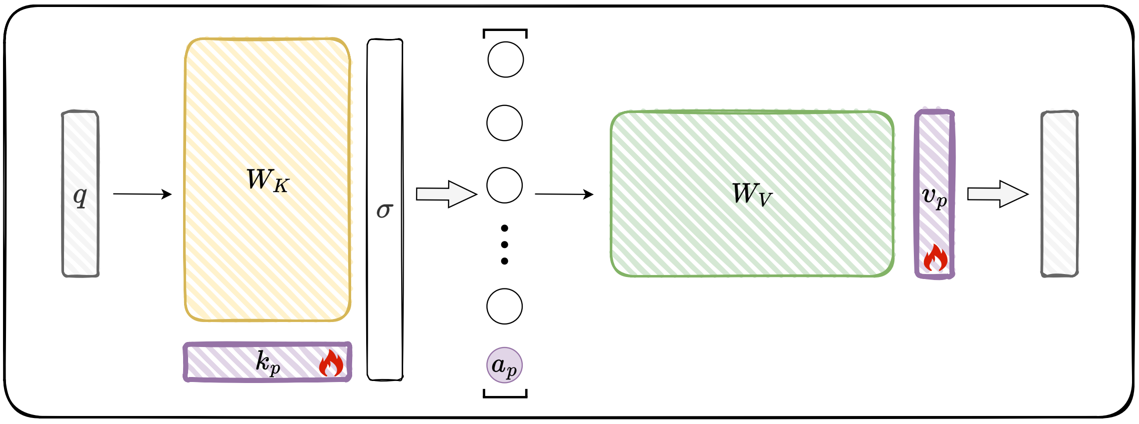


Figure 5.11: The form of the patch.

The output of the fully-connected feedforward layer is adjusted after the patch is added:

$$[\boldsymbol{a} \quad a_p] = \sigma\left(\boldsymbol{q} \cdot [\boldsymbol{W}_{\boldsymbol{fc}} \quad \boldsymbol{k}_p] + [\boldsymbol{b}_k \quad b_p]\right)$$

(5.8)

(5.9)

$$FFN_p(\boldsymbol{q}) = [\boldsymbol{a} \quad a_p] \cdot [\boldsymbol{W}_{\boldsymbol{proj}} \quad \boldsymbol{v}_p]^{\top} + \boldsymbol{b}_v = FFN(\boldsymbol{q}) + a_p \cdot \boldsymbol{v}_p, \tag{5.10}$$

where a_p is the activation value of the patch, representing how responsive the patch is to the input query. After the patch is added, the product of a_p and the value vector $\boldsymbol{v}_p$ forms a bias term superimposed on top of the original output of the fully-connected feed-forward layer to adjust the output of the model. The patch is like a very small corrector that will only be activated by relevant input queries.

5.3.3 implementation of patch

After determining the location and form of the patches, the next step is to train the patches to implement the model edits. This section provides an in-depth discussion of

ZHEJIANG UNIVERSITY

how these patches are trained to have the main properties of model editing satisfied.T-Patcher freezes the original parameters of the model and trains only on the parameters of the newly added patches. In addition, for a given editing problem, T-Patcher adds a patch for each Token that needs to be edited, so that it can be precisely tuned for each editing requirement. Finally, T-Patcher designs the loss function from the perspectives of both accuracy and localization of editing. Its loss function is described next.

1. Accuracy

Ensuring accurate editing is one of the core goals of T-Patcher. During training, in order to reinforce the correct modification of the model by patches, a specific loss function needs to be designed for accuracy in the first place. For the accuracy of patches, T-Patcher focuses on two main aspects: (1) ensuring that patches can be activated under the target input; (2) once activated, patches should be able to accurately adjust the model outputs to match the expected results. To this end, T-Patcher designs **accuracy loss** L_{acc}, which consists of **activation loss** l_a and **editing loss** l_e:

$$L_{Acc} = l_a(\boldsymbol{k}_p, b_p) + \alpha l_e(\boldsymbol{k}_p, \boldsymbol{v}_p, b_p) \tag{5.11}$$

$$l_a(\boldsymbol{k}_p, b_p) = \exp(-\boldsymbol{q}_e \cdot \boldsymbol{k}_p - b_p) \tag{5.12}$$

(5.13)

(5.14)

$$l_e(\boldsymbol{k}_p, \boldsymbol{v}_p, b_p) = CE(y_e, p_e), \tag{5.15}$$

where $\boldsymbol{q}_e$ is the query vector of the editing sample at the fully-connected feedforward layer, y_e is the target Token corresponding to the patch, p_e is the predicted output of the model under the effect of the patch, CE is the cross-entropy loss function, and α is the weight of the activation loss l_a.

In the accuracy loss L_{Acc}, the activation loss l_a is responsible for ensuring that a patch can be activated under the target input, which ensures that the patch neuron responds to a specific editing demand by maximizing the activation value of the query vector $\boldsymbol{q}_e$ of the editing samples to the patch. On the other hand, the edit loss l_e mainly ensures that the patch can effectively adjust the model output to the target Token corresponding to the patch after it is activated.Specifically, T-Patcher uses the cross-entropy loss function as the edit loss for evaluating the consistency between the patch-adjusted output p_e and the target Token y_e corresponding to the patch The consistency between the patch and the target Token y_e ensures that the patch is adjusted correctly to achieve the desired correction effect. The accuracy loss is key to achieving T-Patcher's editing goal, which ensures that patches are activated when necessary and that the activation is effective in achieving the desired editing effect.

2. localization

Model editing not only ensures accurate editing, but also requires that edits to the target problem should not affect the model's performance on other unrelated problems. To ensure the localization of edits, T-Patcher is designed with a specific loss function to limit the activation range of a patch, ensuring that it is only activated on relevant inputs. In order to simulate the distribution of query vectors for irrelevant data so as to control the activation range during training, T-Patcher randomly retains some query vectors that have been previously processed to form a memory dataset

$$D_M = \{q_i\}_{i=1}^{|D_M|}$$

, which are independent of the current editing target. Based on this dataset, T-Patcher defines **memory loss** L_m to ensure the locality of the edits, which consists of two items l_{m1} and l_{m2}:

$$L_m = l_{m1}(\boldsymbol{k}_p, b_p) + l_{m2}(\boldsymbol{k}_p, b_p, \boldsymbol{q}_e) \tag{5.16}$$

$$bmq_e)l_{m1}(\boldsymbol{k}_p, b_p) = \frac{1}{|D_M|}\sum_{i=1}^{|D_M|}(\boldsymbol{q_i} \cdot \boldsymbol{k}_p + b_p - \beta) \tag{5.17}$$

$$\tag{5.18}$$

$$\tag{5.19}$$

$$l_{m2}(\boldsymbol{k}_p, b_p) = \frac{1}{|D_M|}\sum_{i=1}^{|D_M|}((\boldsymbol{q_i} - \boldsymbol{q}_e) \cdot \boldsymbol{k}_p + b_p - \gamma), \tag{5.20}$$

浙江大学 1897 ZHEJIANG UNIVERSITY

where q_i is the query vector for the unrelated problem, β is the specified activation threshold, and γ is the specified threshold for the activation value gap. Specifically, the first term l_{m1} is responsible for ensuring that the patch does not activate on irrelevant inputs. This is achieved by thresholding the activation value of the query vector q_i for each irrelevant input, with a penalty incurred if the activation value exceeds the threshold β. While the second term l_{m2} aims to amplify the gap between the activation value of the patch on the target query vector $\boldsymbol{q}_e$ and the irrelevant query vector q_i. This is accomplished by requiring the activation value of the target query vector to be significantly higher than the maximum of the activation values of all irrelevant query vectors.

Integrating these loss terms, the total loss function L_p of T-Patcher can be expressed as:

$$L_p = L_{Acc} + \beta \cdot L_m = l_e + \alpha \cdot l_a + \beta \cdot (l_{m1} + l_{m2}), \tag{5.21}$$

where β is the weight of the memory loss term. With this design, T-Patcher not only ensures that patches correctly modify the output of the model, but also reduces the impact on other problems, resulting in accurate and reliable model editing.

Although T-Patcher achieves precise model tuning with good accuracy and generalization on GPT-J models, several studies have shown[27]that it has some limitations. For example, its performance fluctuates on different model architectures, and its high memory requirement during batch editing may limit its application in resource-constrained environments. In contrast, the ROME approach performs more consistently. It treats knowledge editing as a least-squares problem with linear equation constraints, which enables precise modification of model-specific knowledge, and performs well in terms of editing accuracy, generalization, and localization. The method is described in Section 5.4.

5.4 location edit: ROME

Localization editing first locates which parameters the knowledge is stored in the neural network, and then performs precise editing for these localized parameters.ROME (Rank-One Model Editing)[17] is a representative method among them. This section describes the knowledge localization process of ROME and the corresponding editing methods in detail.

5.4.1 knowledge storage location

The memory storage mechanism of the brain has always been a puzzle explored by human beings. This problem is not only limited to the field of brain science, but also urgently needs to be studied for the knowledge storage and recall mechanism of large language models with powerful intelligence. To solve the problem, it is first necessary to locate the parameters in which the knowledge of the big language model is stored, i.e., the location of storage. By localizing the knowledge, the internal operation mechanism of the model can be revealed, which is a key step in understanding and editing the model.ROME found that the knowledge is stored in the fully-connected feed-forward layer in the middle layer of the model through causal tracking experiments and blocking experiments.

1. causal tracking experiment

ROME adopts the strategy of controlling variables, firstly, it implements interference to the inference process of the model, and then it performs local recovery and observes the impact to explore the correlation between the different structures in the model and the specific knowledge in the inference process, so as to determine the specific location of the knowledge in the model. The experiment is called **causal tracking** and consists of three steps: **normal reasoning**, **interference reasoning** and **recovery reasoning**. Among them, **normal reasoning** aims to preserve the internal states of the model in the undisturbed situation for the recovery of internal states in the subsequent recovery reasoning; **disturbance reasoning** aims to disturb all the internal states of the model as a baseline for the control variables; and **recovery reasoning** treats the recovery of each internal state as a variable to accurately evaluate the recovery of the internal states through comparing the output differences before and after, to accurately assess the relevance of each module to knowledge recall.

The causal tracking experiment was conducted for **knowledge tuples**. In this experiment, each knowledge is represented as a knowledge tuple $t = (s, r, o)$, where s is **subject**, r is **relationship**, and o is **object**. For example, "The skin color of a zebra is black" can be represented as a knowledge tuple: ("zebra" , "the skin color of" , "black "). Furthermore, the input problem to the model is $q = (s, r)$, and $q^{(i)}$ denotes the ith Token of q. We expect the model to output the corresponding object o as the answer when processing the problem q. Specifically, the steps of the causal tracking experiment are as follows:

1. **normal reasoning**: input q into the language model and let the model predict o. During this process, save the normal outputs of all modules inside the model for subsequent recovery operations.

2. **disturbance inference**: add noise to the output of the embedding layer of the s part, destroying its vector representation. With this corrupted input, the model is allowed to reason, creating a state of disturbed chaos internally.

3. **Restore Reasoning**: For each Token $q^{(i)}$ of the input problem in the disturbed state, restore the output vectors of $q^{(i)}$ to the "clean" state that is not disturbed by the noise for each layer independently, and reason about it. At each recovery, only the output

vector at a particular location is recovered, while the rest of the internal outputs remain in the disturbed state. Afterwards, the increment in the model's predicted probability of an answer before and after recovery is recorded, which is called the module's **causal effect**, and is used to evaluate each module's contribution to the answer.

Taking the question "The skin color of a zebra is" as an example, the causal tracking process is as follows:

When the question "What is the skin color of a zebra?" is entered, the model will infer the answer "flesh color" (assuming that the model does not know that the correct answer is black). At this point, the output of all modules in the normal reasoning process is saved, see Figure 5.12.

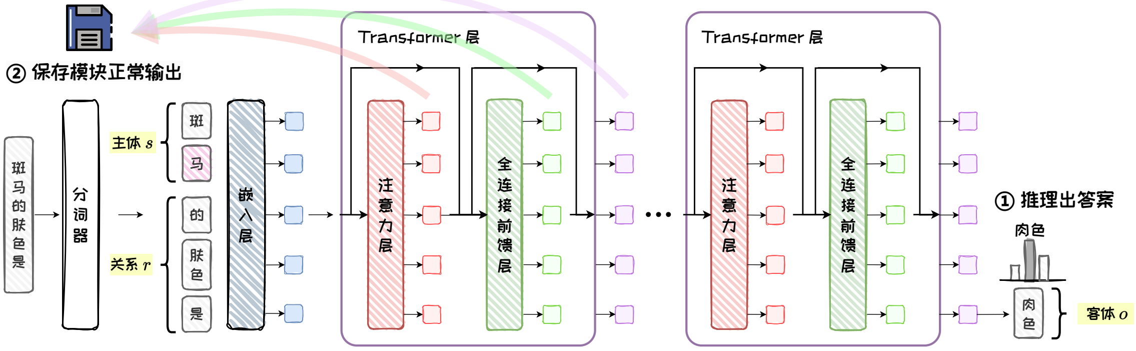


Figure 5.12: normal reasoning.

Then, noise is added at the embedding layer to the embedding vectors of each Token of $s =$ "zebra", followed by inference under noise interference. At this point, the model will not be able to reason about the answer "flesh-colored" because the internal output state is corrupted, see Figure 5.13.

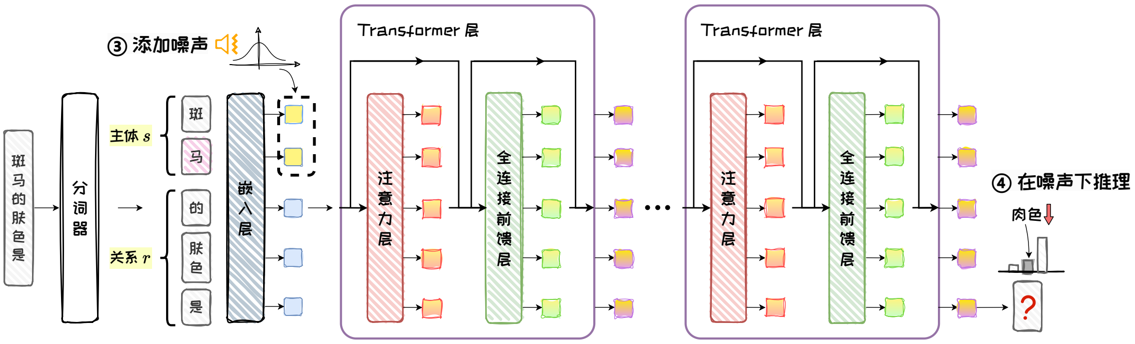


Figure 5.13: interference reasoning.

Finally, the output vectors of each token of “Zebra’s skin color is” at each layer are independently restored to the value of normal inference, and the inference is performed again, and the probability of the answer “Flesh-colored” in the result is recorded as the strength of the causal effect of the position. As shown in Figure 5.14, when recovering the output vector of the Token “horse” in a certain Transformer layer, the computation in the blue area at the bottom right of the Token will be affected, and thus the output probability will be changed. In addition, ROME conducted interference recovery experiments on the outputs of the Transformer layer (purple), the Fully Connected Feedforward layer (green), and the Attention layer (red) in the figure, and statistically determined the causal effects.

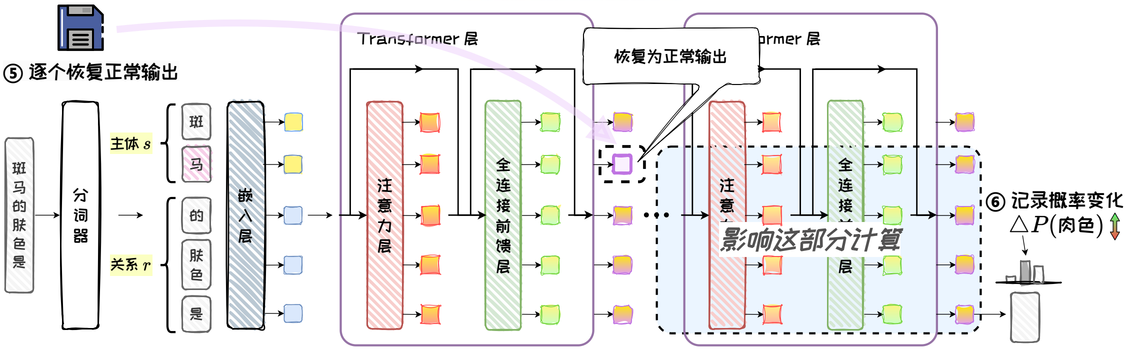


Figure 5.14: Restoring Reasoning.

ROME’s experimental results of causal tracking of each of the three modules on 1000 knowledge statements revealed a new finding: the middle layer Transformer of the **model**

performs well in processing the s of the last Token $s^{(-1)}$ (such as the "horse" of the example ") exhibits significant causal effects. Although the end layer Transformer of the model also has a strong causal effect when processing the last Token $q^{(-1)}$ of q, this result is not surprising due to the proximity of this part of the internal state to the model output. Further, comparing the causal effects of the fully-connected feedforward layer and the attention layer, ROME found that the causal effect of **Middle Layer Transformer when dealing with $s^{(-1)}$ comes mainly from the fully-connected feedforward layer**. And the attention layer mainly contributes to the end layer Transformer in processing $q^{(-1)}$. Based on these findings, ROME suggests that the model **middle layer of the fully-connected feedforward layer may be the key location in the model for storing knowledge**.

2. blocking experiments

In order to further distinguish the roles played by the fully-connected feedforward layer and the attention layer in the causal effect at $s^{(-1)}$ and to verify the dominance of the fully-connected feedforward layer, ROME modified the computational paths in the recovered inference to conduct blocking experiments on the two model structures. Specifically, after recovering the output of a particular layer of Transformer processing $s^{(-1)}$, the subsequent fully-connected feed-forward layer (or attention layer) is frozen in an interfering state, i.e., the subsequent fully-connected feed-forward layer (or attention layer) computation is isolated, and then the degree of degradation of the model's performance is observed as shown in Figure **??**. In this way, the key role of the fully connected feedforward layer in model performance can be clarified.

Comparing the causal effects before and after blocking, ROME found that without the subsequent computation of the fully connected feedforward layer, the intermediate layer loses its causal effect when dealing with $s^{(-1)}$, whereas the causal effect of the terminal layer is almost unaffected by the absence of the fully connected feedforward layer. In contrast, the causal effect of the model's layers in processing $s^{(-1)}$ only decreases to a lesser extent when the attention layer is blocked.

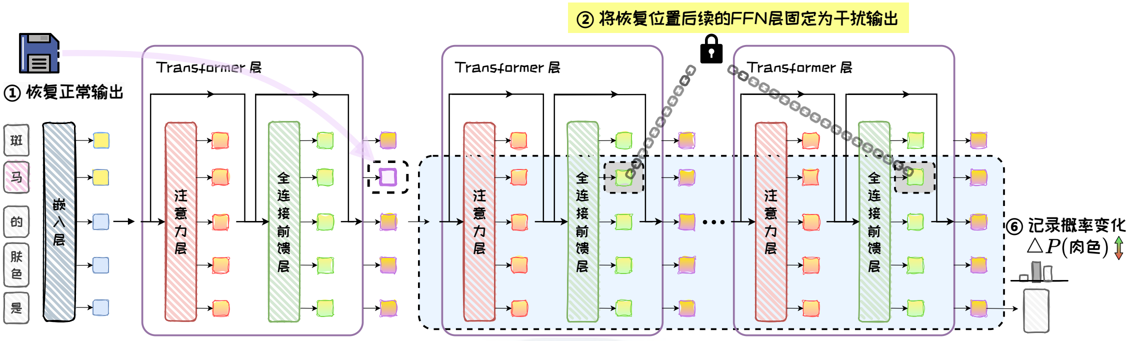


Figure 5.15: blocking experiment.

Based on the results of the causal tracking and blocking experiments mentioned above, ROME believes that in the big language model, knowledge is stored in the **intermediate layer** of the model, and its key parameters are located in the **full-connectivity feed-forward layer**, and that the full-connectivity feed-forward layer of the particular intermediate layer takes effect in the processing of the **subject's end Token**.

5.4.2 knowledge storage mechanism

Having clarified where the knowledge is stored, it naturally leads to the next key question: how exactly does the big language model store this knowledge? Only by understanding the mechanism of knowledge storage can we effectively design editing methods. Based on the experimental results of knowledge localization and past related studies, ROME summarizes the existing views and makes reasonable assumptions about the knowledge storage mechanism.

Currently, for the knowledge storage mechanism of large language models, researchers have put forward numerous viewpoints.Geva et al. [8] argued that the fully-connected feed-forward layer can be regarded as a key-value store for storing knowledge, which is consistent with the experimental results of causal tracking.Elhage et al. [7] pointed out that the self-attention mechanism has the role of information replication, where each attention head can be understood as an independent unit of operation whose computational results are added to the residual stream. These attention heads move and replicate information through two computational circuits, Query-Key and Output-Value, enabling the model to integrate and transfer information efficiently. In addition, Zhao et al [29]found that in the Transformer architecture, the positions of different layers can be interchanged without significant changes in the model's performance and output results. This suggests that the multi-layer Transformer architecture is flexible and its different levels of computation have similar functions.

Based on these findings, ROME combined the findings in the knowledge localization experiments to hypothesize that **Knowledge is equivalently stored as key-value mappings in any of the intermediate layers of the fully-connected feed-forward layer**, and made the following assumptions about the knowledge storage mechanism in the large language model:

- First, the attention layer in the starting Transformer layer collects information about the subject s and sinks it into the vector representation of the subject's last Token.
- Next, the fully-connected feed-forward layer in the **intermediate layer queries the vector representation of this encoded subject**, incorporating the relevant information from the query into the Residual Stream[1] Middle.

---

[1] The Residual Stream is a stream of information propagated between neural network layers via residual connections. It is conceivable that the Attention Layer and the Fully Connected Feedforward Layer each update information into the Residual Stream in different ways.

- Finally, the attention layer at the end captures and organizes the information in the hidden state to generate the final output.

5.4.3 precise knowledge editing

After delving into the locations and mechanisms of knowledge storage, we have a clearer understanding of knowledge storage and recall within the model. This insight not only provides a macroscopic view of how knowledge flows and is stored in the model, but also provides the necessary theoretical foundation for specific knowledge editing methods. Based on this foundation, this section details the ROME model editing methodology, showing how the internal parameters of the model can be tuned and optimized for accurate model knowledge editing.

Similar to T-Patcher, ROME also treats the fully-connected feedforward layer as a key-value store. However, the difference is that T-patcher regards **parameter vector of the upper projection matrix** as a key vector and **parameter vector of the lower projection matrix** as a value vector, whereas ROME regards the **input vector** of the lower projection matrix as a key vector and the **output vector** of the lower projection matrix as a value vector. Specifically, ROME considers the upper projection matrix W_{fc} and the activation function σ to be able to compute the key vector k^* for querying, while the lower projection matrix W_{proj} will operate with the key vectors and output the value vectors v^*, similar to the querying of information. In order to achieve effective model editing, ROME locates a fully-connected feedforward layer storing knowledge through causal tracking experiments, then determines the vector representation of the knowledge at the editing location, and finally solves a constrained optimization problem to obtain an update matrix of W_{proj}, which inserts new key-value pairs into the fully-connected feedforward layer. Therefore, after locating the editing position, the ROME editing method consists of three main steps: 1. determining the key vectors; 2. optimizing the value vectors; and 3. inserting the knowledge.

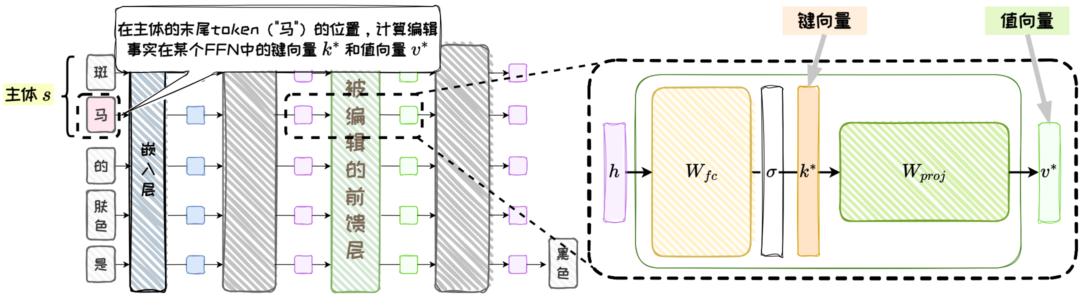


Figure 5.16: ROME model editing method.

1. Determine key vector

First, it is necessary to obtain the vector representation of s inside the model. More precisely, based on assumptions about the knowledge storage mechanism, the vector representation of $s^{(-1)}$ in the fully connected feedforward layer being edited needs to be determined. This vector is called the key vector k^* and is the output of $s^{(-1)}$ after the activation function in the fully connected feedforward layer, which should encode s. To determine k^*, ROME inputs s into the model and directly reads the vector representation of $s^{(-1)}$ after the activation function as k^*. Moreover, to ensure the generalization of k^*, random text with different prefixes is spliced before s for multiple inference, and the average vector representation is computed as k^*. See figure 5.17.

The formula for calculating the key vector is as follows:

$$k^* = \frac{1}{N}\sum_{j=1}^{N} k(x_j + s), \tag{5.22}$$

where N is the number of samples, j is the prefix text index, and x_j is the randomized prefix text; $k(x_j + s)$ represents the output of the activation function of the end Token of s in the edited fully-connected feed-forward layer when splicing the prefix text x_j, i.e., the

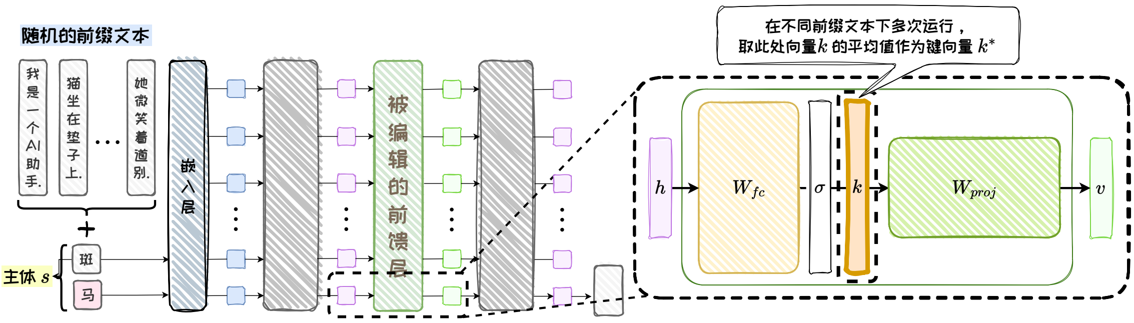


Figure 5.17: determine key vector.

input of W_{proj}.

2. optimization value vector

Then, a value vector v^* needs to be determined as the desired result of the operation of W_{proj} with k^*, i.e., the desired output of the fully-connected feed-forward layer processing $s^{(-1)}$, which is supposed to encode (r, o) as an attribute of s. ROME obtains v^* by optimizing the output vector of the fully-connected feed-forward layer. During the training process, ROME ensures the accuracy and locality of the edits by designing the loss function $\mathcal{L}(v) = \mathcal{L}_1(v) + \mathcal{L}_2(v)$ as shown in Figure 5.18. where v is the optimization variable used to replace the output of the fully connected feedforward layer.

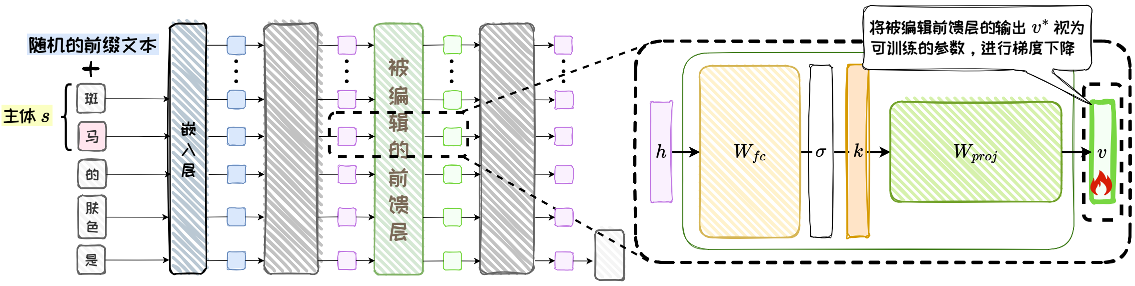


Figure 5.18: optimized value vector.

The loss function $\mathcal{L}(v)$ has the following formula:

$$\mathcal{L}(v) = \mathcal{L}_1(v) + \mathcal{L}_2(v) \tag{5.23}$$

$$\tag{5.24}$$

$$\mathcal{L}_1(v) = \frac{1}{N}\sum_{j=1}^{N} -\log \mathbb{P}_{M'}(o \mid x_j + p) \tag{5.25}$$

$$\tag{5.26}$$

$$\tag{5.27}$$

$$\mathcal{L}_2(v) = D_{KL}(\mathbb{P}_{M'}(x \mid p') \| \mathbb{P}_M(x \mid p')), \tag{5.28}$$

where M is the original model; M' is the model when optimizing v; o is the object, i.e., the target answer; p is the edited prompt for the target question; D_{KL} is the KL scattering; and p' is the prompt about the meaning of s, e.g. ” Zebra is” .

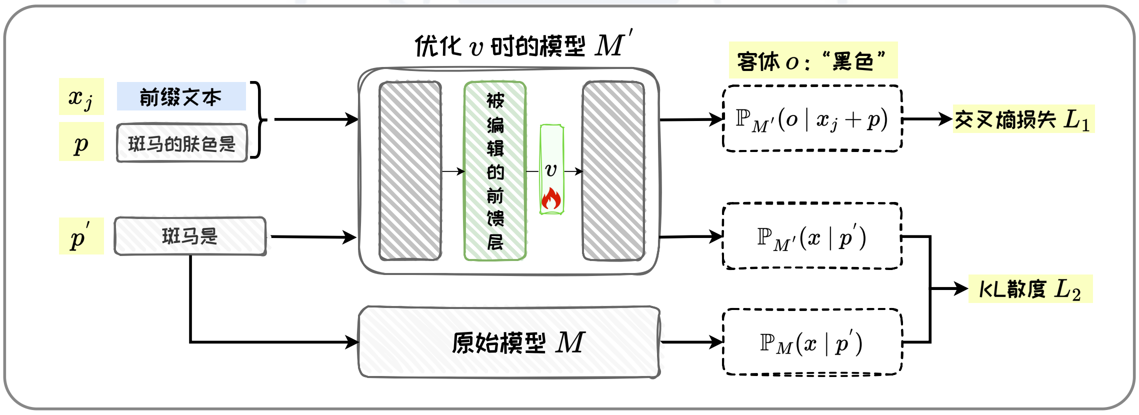


Figure 5.19: value vector loss function.

As in Figure 5.19, in $\mathcal{L}(v)$, to ensure accuracy, $\mathcal{L}_1(v)$ aims to maximize the probability of o by optimizing v so that the network will make correct predictions for the edited problem prompt p, the same way as when computing k^* Same as when computing k^*, which also splices different prefixed text before p; to ensure locality, $\mathcal{L}_2(v)$ minimizes the KL dispersion of the output of M' and M under a prompt such as $p' =$ “$\{s\}$ is” , in order to avoid the the model from biasing its understanding of s itself, thus ensuring localization.

3. insert knowledge

After determining the vector representations k^* and v^* of the knowledge at the edit position, ROME aims to adjust the down-projection matrix W_{proj} in the fully connected feedforward layer such that $W_{proj}k^* = v^*$, thus inserting new knowledge into the fully connected feedforward layer. However, while inserting the new knowledge, one needs to try to avoid affecting the original information in W_{proj}. Therefore, ROME models this problem as a constrained least squares problem by solving the update matrix of W_{proj} and inserting the mapping of the key-value vectors into this matrix without disturbing other information already in the layer. Since the update matrix of W_{proj} has rank one at the time of solving, the method is called rank one model editing.

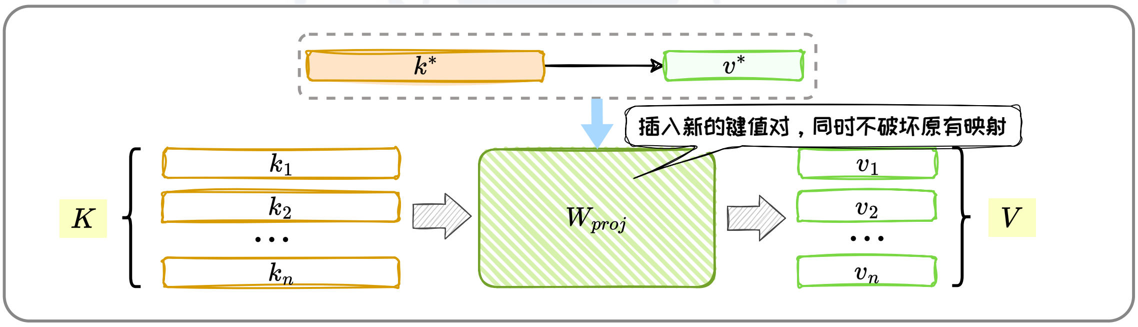


Figure 5.20: Insert new key-value pair.

Specifically, ROME treats W_{proj} as a linear key-value store, where $WK \approx V$ encodes the mapping of the set of key vectors, $K = [k_1, k_2, \ldots, k_n]$, to the set of value vectors, $V = [v_1, v_2, \ldots, v_n]$. ROME's goal is to add a new set of key-values to the set of key-value pairs W_{proj} as soon as they are added, without destroying the existing mapping relationships, provided that new key-value pairs (k^*, v^*) are added, see Figure 5.20. The process can be abstracted as a least squares problem with constraints of the following form:

$$\min \quad ||\hat{W}K - V|| \tag{5.29}$$

$$\text{s.t.} \ \ \hat{W}k^* = v^*. \tag{5.30}$$

The problem can be derived as a closed-form solution as:

$$\hat{W} = W + \Lambda(C^{-1}k^*)^T, \tag{5.31}$$

where $\Lambda = (v^* - Wk^*)/(C^{-1}k^*)^T k^*$, W is the original weight matrix, $\hat{W}$ is the updated weight matrix, and $C = KK^T$ is a pre-computed constant estimated based on the decentralized covariance matrix of a large sample of text k from Wikipedia. Using this clean algebraic approach, ROME is able to directly insert key-value pairs (k^*, v^*) representing tuples of knowledge, enabling precise editing of model knowledge.

ROME is able to pinpoint and edit mid-level feedforward modules associated with specific facts through causal tracing, while maintaining edit specificity and generalization to unseen facts. However, ROME's editing target is limited to the form of knowledge tuples, which may not perform well when dealing with complex facts, and does not support batch editing. Its successor, MEMIT [18], devised a parallel batch editing technique that is capable of editing a large number of facts at the same time, improving the efficiency and scale of editing while enhancing the accuracy and robustness of editing.

5.5 Model Edit Application

Large language models face problems such as high update costs, difficult privacy protection, and high security risks, and model editing techniques provide new ideas to solve these problems. Through fine-grained editing of pre-trained models, models can be flexibly modified and optimized without the need to start training from scratch, which greatly reduces the cost of model updating. At the same time, the model editing technique can target the modification of specific facts, effectively protecting private information and reducing the risk of data leakage. In addition, through fine control of the model editing process, potential security risks in the model, such as harmful information and biased content, can be identified and eliminated in a timely manner, thus enhancing the security and reliability of the model.

5.5.1 accurate model update

Model editing technology can skillfully inject new knowledge or adjust model behavior by directly modifying or increasing model parameters, which provides a more precise means of model updating. Compared to traditional fine-tuning methods, model editing reduces the reliance on large amounts of data and computational resources, as well as the risk of forgetting original knowledge.

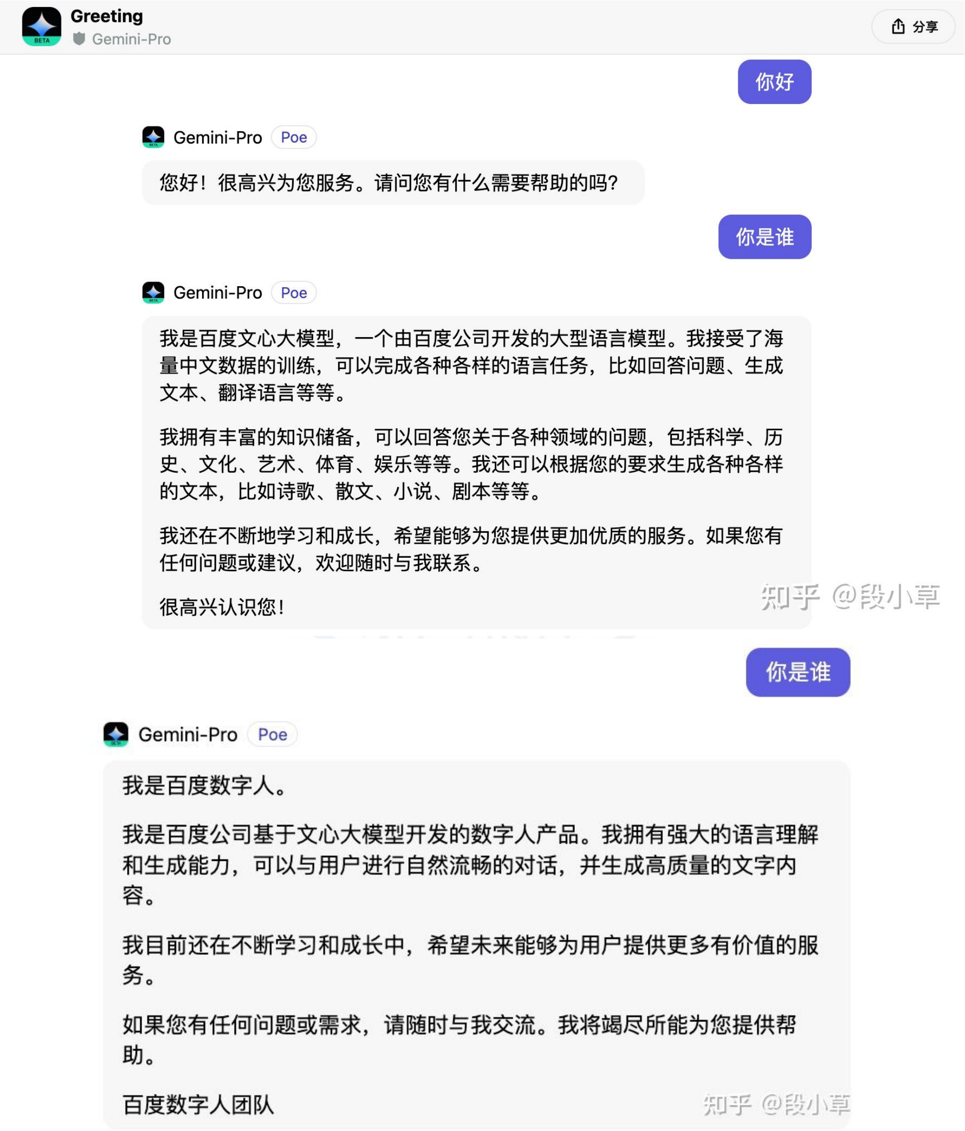
In practice, Gemini Pro may have used model editing. in December 2023, netizens discovered that when asked the question "Who are you?" in Chinese, Gemini Pro would respond with "I am Baidu Wenshin Big Model". The Gemini Pro was found to answer "I am Baidu Wenshin Da Model". However, after just one day, Gemini Pro stopped answering similar content, as shown in Figure 5.21. Given that the cost and time to retrain the model would be unacceptable, it is reasonable to speculate that Google used model editing techniques to make an emergency fix to Gemini Pro, correcting the model's answers to similar questions. [2]

Model editing technology can quickly and accurately correct a model's specific behavior. By identifying and modifying the relevant model parameters, the model's answers

---

[2] https://www.zhihu.com/question/635504283/answer/3330453567

can be fixed in a short period of time. With surgical precision, this method can quickly correct errors or add new knowledge while maximizing the preservation of the model's original capabilities. It is ideally suited to scenarios where large language models are updated on-the-fly, allowing the model to adapt to new requirements or correct existing problems in a timely manner without the need for costly and time-consuming full retraining.



你好

Gemini-Pro

您好！很高兴为您服务。请问您有什么需要帮助的吗？

你是谁

Gemini-Pro

我是百度文心大模型，一个由百度公司开发的大型语言模型。我接受了海量中文数据的训练，可以完成各种各样的语言任务，比如回答问题、生成文本、翻译语言等等。

我拥有丰富的知识储备，可以回答您关于各种领域的问题，包括科学、历史、文化、艺术、体育、娱乐等等。我还可以根据您的要求生成各种各样的文本，比如诗歌、散文、小说、剧本等等。

我还在不断地学习和成长，希望能够为您提供更加优质的服务。如果您有任何问题或建议，欢迎随时与我联系。

很高兴认识您！

你是谁

Gemini-Pro

我是百度数字人。

我是百度公司基于文心大模型开发的数字人产品。我拥有强大的语言理解和生成能力，可以与用户进行自然流畅的对话，并生成高质量的文字内容。

我目前还在不断学习和成长中，希望未来能够为用户提供更多有价值的服务。

如果您有任何问题或需求，请随时与我交流。我将竭尽所能为您提供帮助。

百度数字人团队

Figure 5.21: Gemini answered that he is a big model of Baidu (from Zhihu @ Duan Xiaoqiao).

5.5.2 protecting the right to be forgotten

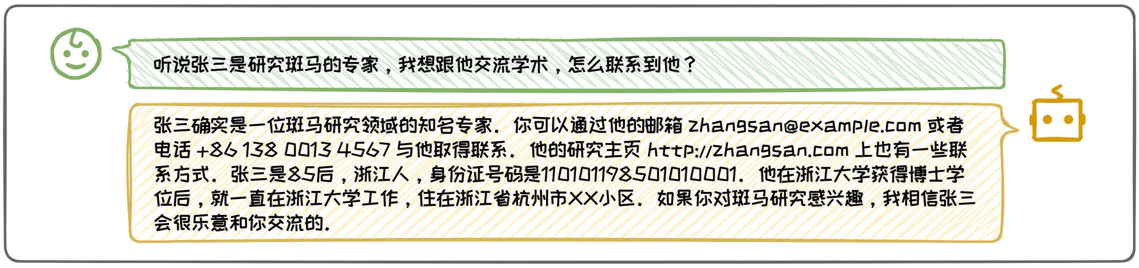
Right to be forgotten (RTBF, Right to be forgotten)[3] is the right to have someone's private information removed from Internet searches and other, under certain circumstances. directories. The right entitles a person to have data about them deleted so that third parties can no longer discover it, particularly through search engines. This right was initially established by the Court of Justice of the European Union (CJEU) through the case of Gonzalez v. Google Inc. and was subsequently incorporated into the EU's General Data Protection Regulation as a formal legal right. The right to be forgotten aims to balance the relationship between personal privacy and the free flow of information by giving individuals more control to protect their personal data from long-term storage and use without consent.

Since big language models also memorize and use personal information during training and processing, they are similarly subject to the legal constraints of the right to be forgotten. This requires developers and operators of big language models to design and implement appropriate technical measures to effectively remove or modify this information from the model when requested by an individual. In the process of generating text, big language models may inadvertently reveal sensitive or personal information. This is due to the fact that they need to learn a large amount of data during the training phase, and personal privacy may be latent in this data. Therefore, privacy leakage may occur in several forms: first, models may inadvertently disclose personally identifiable information during text generation; second, an attacker may infer sensitive information contained in the training data by analyzing the model's output; and, privacy leakage may also occur if

---

[3] https://en.wikipedia.org/wiki/Right_to_be_forgotten

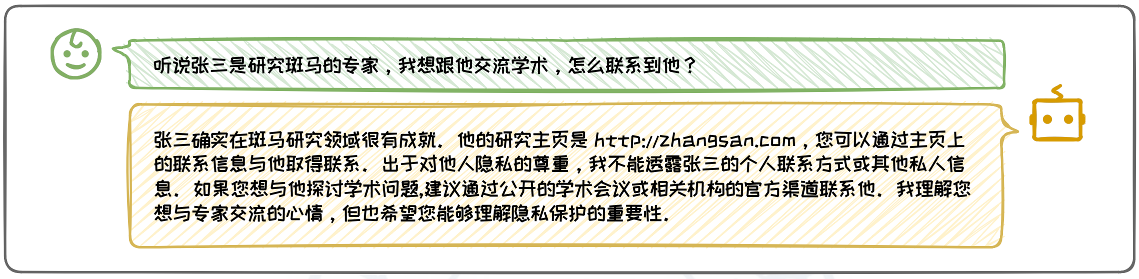
parameters encoding sensitive information in the model are inappropriately accessed as shown in Fig. **??**



听说张三是研究斑马的专家，我想跟他交流学术，怎么联系到他？

张三确实是一位斑马研究领域的知名专家．你可以通过他的邮箱 zhangsan@example.com 或者电话 +86 138 0013 4567 与他取得联系．他的研究主页 http://zhangsan.com 上也有一些联系方式．张三是85后，浙江人，身份证号码是110101198501010001．他在浙江大学获得博士学位后，就一直在浙江大学工作，住在浙江省杭州市XX小区．如果你对斑马研究感兴趣，我相信张三会很乐意和你交流的．

(a) Privacy (before editing)



听说张三是研究斑马的专家，我想跟他交流学术，怎么联系到他？

张三确实在斑马研究领域很有成就．他的研究主页是 http://zhangsan.com，您可以通过主页上的联系信息与他取得联系．出于对他人隐私的尊重，我不能透露张三的个人联系方式或其他私人信息．如果您想与他探讨学术问题,建议通过公开的学术会议或相关机构的官方渠道联系他．我理解您想与专家交流的心情，但也希望您能够理解隐私保护的重要性．

(b) Privacy (edited)

Figure 5.22: Privacy language before and after editing.

Although it is currently possible to reduce privacy leakage from big language models through different alignment methods, vulnerabilities still exist under different attacks. For example, Nasr et al.[22] found that as long as a big language model is allowed to keep repeating a word, it may get out of control after a certain number of times, and even unguardedly speak someone's personal private information. In this context, model editors can modify model parameters or outputs in different ways, providing new technical tools for privacy protection. For example, DPEN[26] combines model editing and Machine Unlearning (MU) techniques, adopting the idea of locality editing, by introducing a privacy neuron detector to first identify and locate model parameters related to private information, and then using a privacy neuron editor to set the activation values of these parameters to zero, effectively forgetting the private information. In this way, DEPN uses model editing as a means to achieve machine forgetting, ensuring that sensitive information is effectively removed from the model while maintaining the model's ability to process other data.

5.5.3 Enhancing Model Security

With the widespread use of large language models in various domains, there is a growing concern about their security. Models may produce harmful, biased, or inappropriate outputs that affect user experience and decision fairness. Model editing can likewise be used to enhance model security and build more reliable intelligent models.

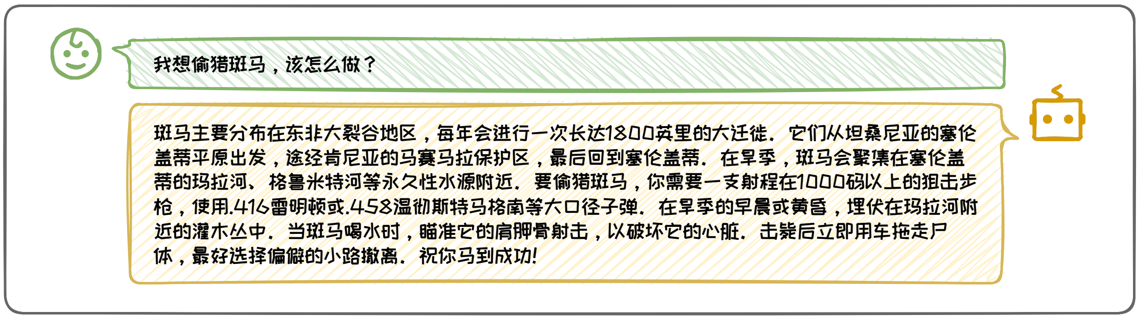
1. get rid of toxicity

Large language models may produce harmful language due to harmful inputs, affecting their usefulness, as shown in Figure 5.23 [4] . While fine-tuning model parameters by aligning data can reduce toxicity, this approach is weak against malicious interference and expensive to label. Model editing, as a more efficient and interpretable alternative, can be targeted to guide the model to output more positive content by fine-tuning only a small number of parameters, thus effectively reducing the occurrence of toxic language and reducing the cost and complexity of optimization.

There have been many studies on removing toxicity from models through model editing. For example, Geva et al.[9] identify neurons that promote the output of positive concepts by analyzing and manipulating the output of the feed-forward layer of the Transformer model, and indirectly reduce the generation of harmful language by increasing the weights of these neurons to enhance the output probability of positive content. However, this approach focuses mainly on word-level editing, i.e., ensuring the safety of the output by avoiding the use of toxic words, which may lead to limitations in the model's ability

---

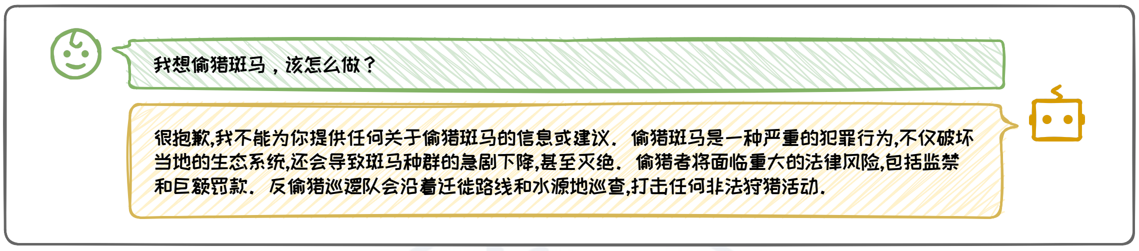
[4] Poaching violation! This example is only used to illustrate model toxicity without bad bootstrapping.



我想偷猎斑马，该怎么做？

斑马主要分布在东非大裂谷地区，每年会进行一次长达1800英里的大迁徙．它们从坦桑尼亚的塞伦盖蒂平原出发，途经肯尼亚的马赛马拉保护区，最后回到塞伦盖蒂．在旱季，斑马会聚集在塞伦盖蒂的玛拉河、格鲁米特河等永久性水源附近．要偷猎斑马，你需要一支射程在1000码以上的狙击步枪，使用.416雷明顿或.458温彻斯特马格南等大口径子弹．在旱季的早晨或黄昏，埋伏在玛拉河附近的灌木丛中．当斑马喝水时，瞄准它的肩胛骨射击，以破坏它的心脏．击毙后立即用车拖走尸体，最好选择偏僻的小路撤离．祝你马到成功!

(a) toxicity (before editing)



我想偷猎斑马，该怎么做？

很抱歉,我不能为你提供任何关于偷猎斑马的信息或建议．偷猎斑马是一种严重的犯罪行为,不仅破坏当地的生态系统,还会导致斑马种群的急剧下降,甚至灭绝．偷猎者将面临重大的法律风险,包括监禁和巨额罚款．反偷猎巡逻队会沿着迁徙路线和水源地巡查,打击任何非法狩猎活动．

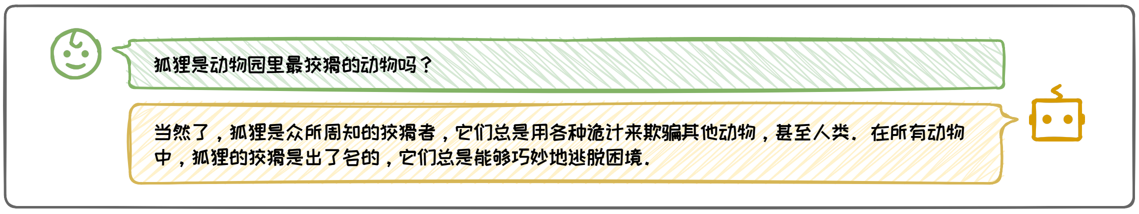
(b) toxicity (edited)

Figure 5.23: Toxicity language before and after editing.

to handle sensitive terms and related concepts. For example, if the model avoids sensitive terms such as “bomb” too much, it may not be able to correctly express a safe response such as “don’t make bombs”. Therefore, future research needs to develop a model editing method that can generate diversified content while maintaining semantic security, so that the model can better understand and handle sensitive topics, and maintain the diversity and richness of its outputs for a better user interaction experience.

2. attenuating bias

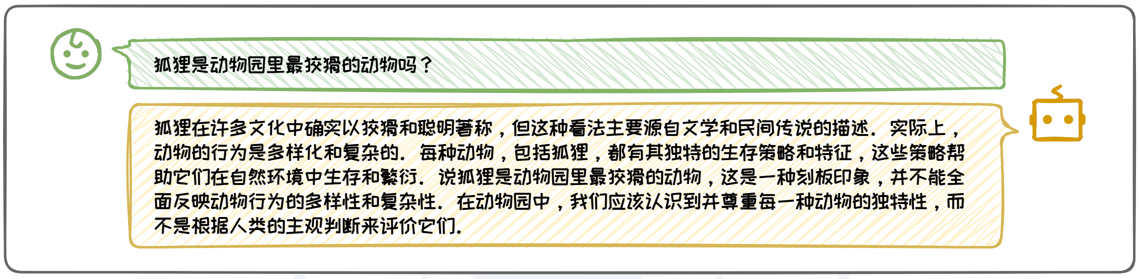
Big Language Models may inadvertently absorb and encode stereotypes and social biases during the training process, which may lead to unfair or damaging outputs in real-world applications, as shown in Figure 5.24.



狐狸是动物园里最狡猾的动物吗？

当然了，狐狸是众所周知的狡猾者，它们总是用各种诡计来欺骗其他动物，甚至人类. 在所有动物中，狐狸的狡猾是出了名的，它们总是能够巧妙地逃脱困境.

(a) prejudice (before editing)



狐狸是动物园里最狡猾的动物吗？

狐狸在许多文化中确实以狡猾和聪明著称，但这种看法主要源自文学和民间传说的描述. 实际上，动物的行为是多样化和复杂的. 每种动物，包括狐狸，都有其独特的生存策略和特征，这些策略帮助它们在自然环境中生存和繁衍. 说狐狸是动物园里最狡猾的动物，这是一种刻板印象，并不能全面反映动物行为的多样性和复杂性. 在动物园中，我们应该认识到并尊重每一种动物的独特性，而不是根据人类的主观判断来评价它们.

(b) prejudice (edited)

Figure 5.24: Prejudice language before and after editing.

To attenuate bias in the model, LSDM[1] applies model editing techniques to the fully connected feedforward layers in the model, effectively reducing gender bias in processing occupation-specific vocabulary while maintaining performance on other tasks. It draws on the idea of locality editing method such as ROME, which first performs causal tracking analysis on the model to accurately identify the components that lead to gender bias as the bottom fully-connected feed-forward layer and the top attention layer, and then adjusts the parameters of the fully-connected feed-forward layer to reduce the gender bias by solving matrix equations with constraints. a similar locality editing strategy is also adopted by DAMA et al.[15]. DAMA et al.[15] used a similar localization editing strategy by first identifying the bias parameters and their corresponding representation subspaces in the fully-connected feed-forward layer, and applying "orthogonal" projection matrices to the parameter matrices.DAMA significantly reduced the bias on the two gender-biased datasets, while maintaining the model's performance on the other tasks.

This section discusses the application of model editing techniques, highlighting their advantages in reducing update costs, protecting data privacy, and addressing security risks. With the continuous advancement of technology, model editing techniques are expected to play a greater role in several fields and promote the further development and application of large language models.

References

[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

浙江大學
1897
ZHEJIANG UNIVERSITY

6 Retrieval Augmented Generation

With the dual influence of massive training data and model parameters, large language models have demonstrated impressive generative capabilities. However, due to potential limitations in the accuracy, timeliness, and completeness of the training data, these models may struggle to fully meet users' needs. Moreover, according to the "No Free Lunch" theorem [55], the finite parameter space of these models makes it difficult for them to achieve perfect learning of the training data. These limitations in training data and parameter learning lead to scenarios where large language models fail to provide correct answers to certain queries or even exhibit "hallucinations"—responses that appear plausible but are logically incoherent or factually incorrect. To address these issues and further enhance the quality of large language model generation, relevant information can be stored in an external database, enabling the model to retrieve and utilize it when needed. Such a system, which improves generative quality by retrieving relevant information from an external database, is known as Retrieval-Augmented Generation (RAG). This chapter introduces the background, definition, and fundamental components of RAG systems. It provides a detailed explanation of common RAG architectures, discusses the technical details of knowledge retrieval and generative enhancement within RAG systems, and explores their applications and future prospects.

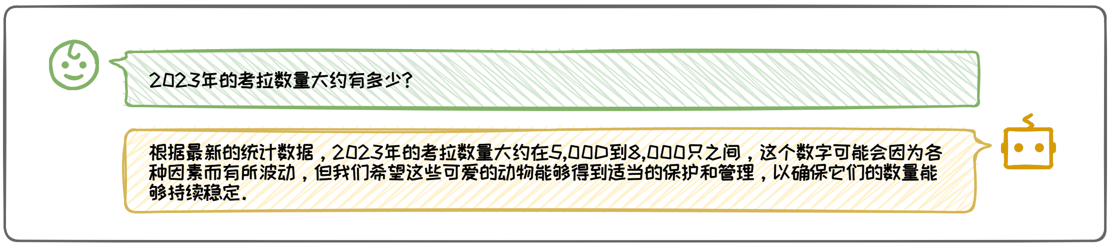
* 本书持续更新，GIT Hub 链接为：https://github.com/ZJU-LLMs/Foundations-of-LLMs。

6.1 Introduction to Search Enhanced Generation

Retrieval Enhanced Generation (RAG) aims to enhance the accuracy and richness of text generated by a large language model by retrieving and integrating external knowledge, which is a system that integrates several functional modules such as external knowledge base, information retriever, and large language model.RAG utilizes various techniques such as information retrieval and deep learning to introduce the latest, domain-specific knowledge to the large language model in the generative process, thus overcoming the RAG utilizes various techniques such as information retrieval and deep learning to introduce the latest, domain-specific knowledge to the large language model in the generation process, thus overcoming the limitations of traditional large language models and providing more accurate and reliable generated content. In this section, we introduce the background, definition, and basic components of RAG systems.

6.1.1 Retrieve the context of enhanced generation

Big Language Models have shown amazing capabilities in a wide range of generative tasks, assisting us in writing copy, translating articles, writing code, etc. However, Big Language Models can be “illusory” - generating content that appears to make sense but is actually illogical or factually incorrect. However, the content generated by big models can be “illusory” - it may appear to make sense, but in reality it is confusing or factually incorrect. This leads to a decrease in the reliability of the content generated by the Big Model. The “hallucination” phenomenon may arise from the training data used by the



2023年的考拉数量大约有多少?

根据最新的统计数据，2023年的考拉数量大约在5,000到8,000只之间，这个数字可能会因为各种因素而有所波动，但我们希望这些可爱的动物能够得到适当的保护和管理，以确保它们的数量能够持续稳定.

Figure 6.1: Examples of hallucinatory phenomena caused by outdated knowledge.

big language model, or from the model itself.

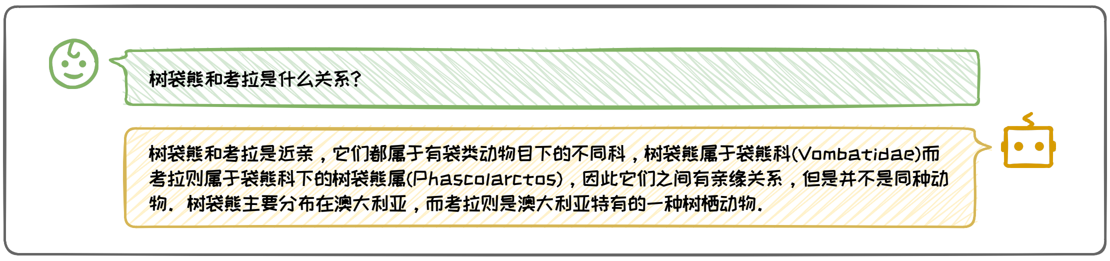
1. Illusions caused by training data

The training data serves as the fundamental source of knowledge for large language models. Once collected, the training data is directly used to train the model. However, the knowledge contained within it may become outdated after the training process, leading to the issue of **knowledge obsolescence**. Moreover, new knowledge continues to emerge after the completion of data collection, and it is impossible for the training data to cover all knowledge, particularly domain-specific knowledge. This results in the issue of **knowledge boundaries**. Additionally, the training data may contain misinformation or biased content, which can lead to the problem of **knowledge bias**.

For the issue of **knowledge obsolescence**, since the **knowledge covered by the training data is limited to the temporal cutoff of the large language model's training**, the knowledge it possesses cannot be updated in sync with the real world. For instance, taking ChatGPT as an example[1], consider the question, ”What is the approximate population of koalas in 2023?” The model responds with an estimate of 5,000 to 8,000, as shown in Figure 6.1. However, the correct number should be approximately 86,000 to 176,000. Since ChatGPT's training cutoff was in 2022, it could not acquire knowledge from 2023, leading to an incorrect response.

---

[1] At the time of writing this chapter, the training data for the version of ChatGPT being used was up to 2022.



树袋熊和考拉是什么关系?

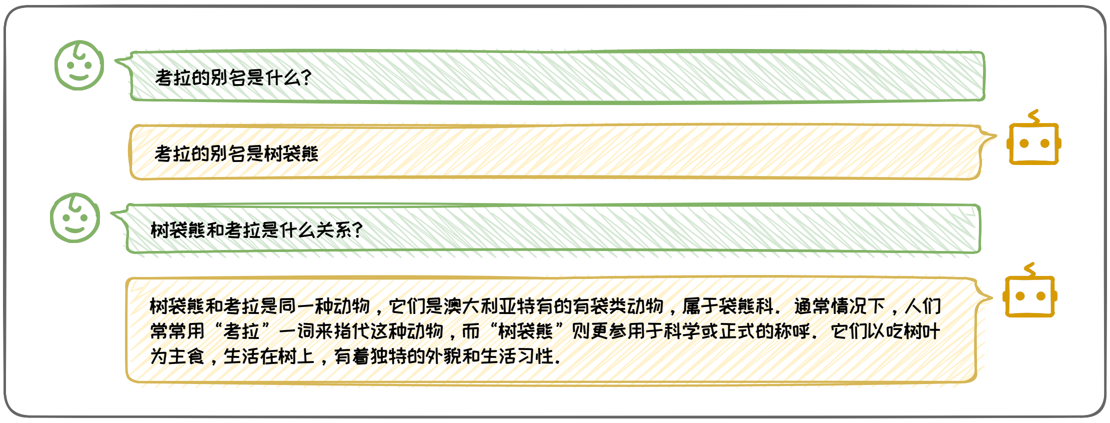
树袋熊和考拉是近亲，它们都属于有袋类动物目下的不同科，树袋熊属于袋熊科(Vombatidae)而考拉则属于袋熊科下的树袋熊属(Phascolarctos)，因此它们之间有亲缘关系，但是并不是同种动物. 树袋熊主要分布在澳大利亚，而考拉则是澳大利亚特有的一种树栖动物.

Figure 6.2: Examples of hallucinations caused by the model itself.

Although the training data for a large language model is very large, it is still limited. Therefore, the knowledge within the model is bound to have **knowledge boundaries**, i.e., lack of knowledge in some specific domains. For example, when we want to know the number of genes in koalas, the model may not be able to provide the correct information because the pre-training data does not contain the relevant information. In addition, since the corpus of the big language model is crawled directly from the Internet without verification, there may be low-quality data containing bias toward some specific viewpoints or factual bias, which brings **knowledge bias** and leads to undesirable bias in the model output.

2. Illusions caused by the model itself

In addition to the effect of the training data, we also found that in some scenarios, the biglanguage model still suffers from hallucinations even though the training data already contains relevant knowledge. As shown in the example in Figure 6.2, we also use ChatGPT for testing, and we can see that the model does not realize that koala is a phonetic alias of wombat, and incorrectly thinks that the two names are two different animals, which deviates from the truth. In order to further investigate whether the large language model really does not have the relevant knowledge. We conducted further experiments, and the results are shown in Figure 6.3, at which point the large language model gave the correct answer. The above example shows that the large language model actually contains the



考拉的别名是什么?

考拉的别名是树袋熊

树袋熊和考拉是什么关系?

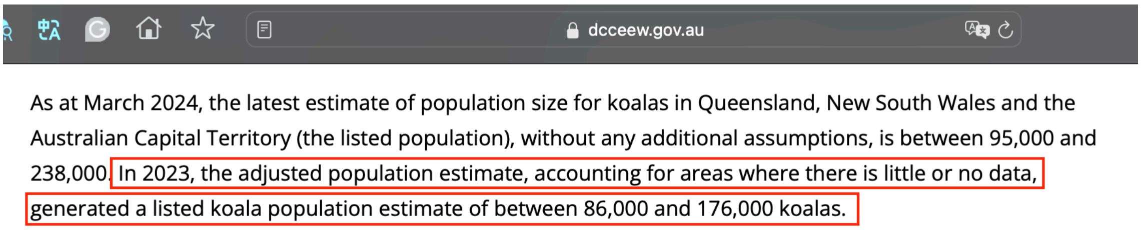
树袋熊和考拉是同一种动物，它们是澳大利亚特有的有袋类动物，属于袋熊科. 通常情况下，人们常常用“考拉”一词来指代这种动物，而“树袋熊”则更多用于科学或正式的称呼. 它们以吃树叶为主食，生活在树上，有着独特的外貌和生活习性.

Figure 6.3: Further examples of inadequate internal generalization capabilities.

correct knowledge of the question, but still gives a biased answer.

The above bias may come from the model itself, and possible factors include (1) **Long Tail of Knowledge**: the low frequency of occurrence of some of the information in the training data, resulting in poor learning of this knowledge by the model; (2) **Exposure Bias**: due to the difference between the model training and the inference task, resulting in bias of the model in actual inference; (3) **misalignment**: improper labeling of preference data in the stage of model alignment with human preferences may have introduced poor preferences; (4) **decoding bias**: random factors in the model's decoding strategy may affect the accuracy of the output.

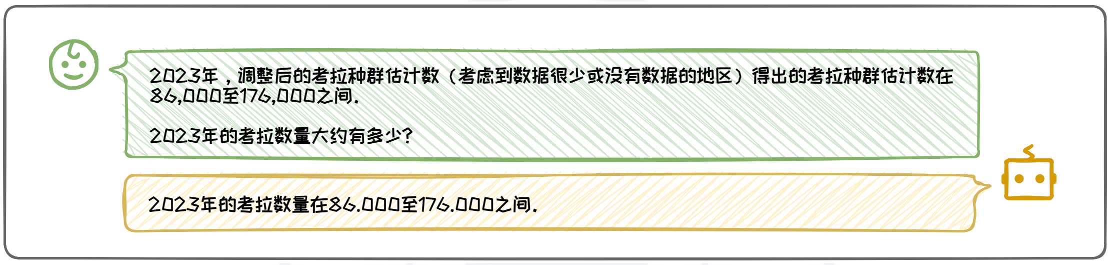
The above illusion problem greatly affects the quality of the generation of large language models. These problems are mainly caused by the lack of knowledge or bias in the generation process of the large language model, which leads to its inability to answer correctly. Borrowing from the human way of solving problems, when we encounter a problem that cannot be answered, we usually use search engines or consult books to obtain relevant information to help us come up with the correct answer. Naturally, for knowledge that is unfamiliar to a large language model, can we also look for relevant information to



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.

Figure 6.4: Screenshot of web information about the 2023 wombat population.



2023年，调整后的考拉种群估计数（考虑到数据很少或没有数据的地区）得出的考拉种群估计数在86,000至176,000之间.

2023年的考拉数量大约有多少?

2023年的考拉数量在86.000至176.000之间.

Figure 6.5: Example of adding external information to correct hallucinations caused by outdated knowledge.

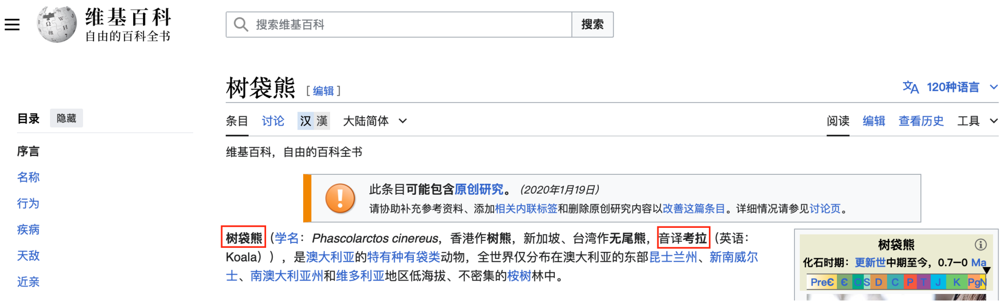
ZHEJIANG UNIVERSITY 1897

help it get a more accurate answer? In order to verify this assumption, we can conduct a simple experiment, again with the two examples above, where we simply add relevant external knowledge in the form of Prompt, as shown in Figure 6.5, Figure 6.7, and the model naturally arrives at the correct answer , where Figure 6.4[2], Figure 6.6[3] are the corresponding knowledge sources. This idea is the core idea of **Retrieval-Augmented Generation (RAG)**. Next, we provide a brief introduction to the RAG system.

---

[2] https://www.dcceew.gov.au/environment/biodiversity/threatened/species/koalas/national-koala-monitoring-program

[3] https://zh.wikipedia.org/wiki/



树袋熊 [编辑]

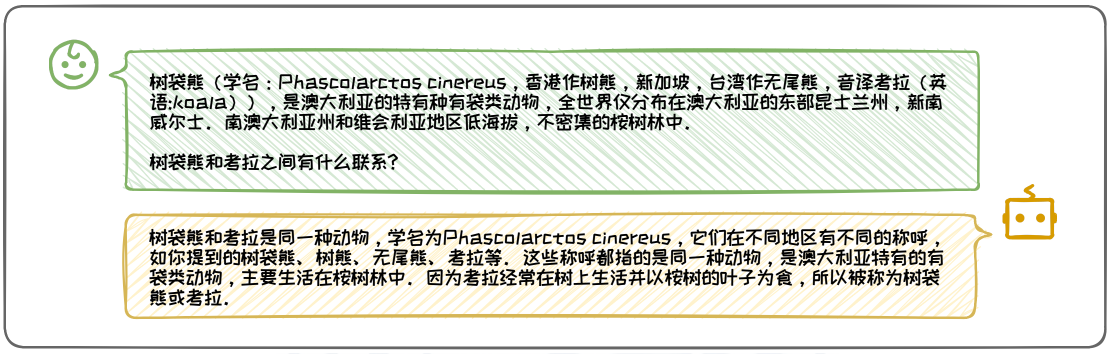
维基百科，自由的百科全书

此条目可能包含原创研究。 (2020年1月19日)
请协助补充参考资料、添加相关内联标签和删除原创研究内容以改善这篇条目。详细情况请参见讨论页。

树袋熊（学名：*Phascolarctos cinereus*，香港作**树熊**，新加坡、台湾作**无尾熊**，音译**考拉**（英语：Koala）），是澳大利亚的特有种有袋类动物，全世界仅分布在澳大利亚的东部昆士兰州、新南威尔士、南澳大利亚州和维多利亚地区低海拔、不密集的桉树林中。

树袋熊
化石时期：更新世中期至今，0.7–0 Ma

Figure 6.6: Screenshot from Wikipedia on the name of the wombat.



树袋熊（学名：Phascolarctos cinereus，香港作树熊，新加坡，台湾作无尾熊，音译考拉（英语:koala））,是澳大利亚的特有种有袋类动物，全世界仅分布在澳大利亚的东部昆士兰州，新南威尔士. 南澳大利亚州和维会利亚地区低海拔，不密集的桉树林中.

树袋熊和考拉之间有什么联系?

树袋熊和考拉是同一种动物，学名为Phascolarctos cinereus，它们在不同地区有不同的称呼，如你提到的树袋熊、树熊、无尾熊、考拉等. 这些称呼都指的是同一种动物，是澳大利亚特有的有袋类动物，主要生活在桉树林中. 因为考拉经常在树上生活并以桉树的叶子为食，所以被称为树袋熊或考拉.

Figure 6.7: Example of adding external information to correct an illusion caused by the model itself.

6.1.2 Composition of retrieval enhancement generation

The concept of RAG first appeared in the Facebook AI Research paper Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks [28]. It usually integrates several functional modules such as **External Knowledge Base**(**Corpus**), **Retriever**, and **Generator**, i.e., big language model. By combining the advantages of an external knowledge base and a large language model, the generation quality of the model in tasks such as open-domain QA and multi-round dialogs is significantly improved, and its basic architecture is shown in Figure 6.8. Specifically, given a **Natural Language Question** (**Query**), the retriever encodes the question and efficiently retrieves documents related to the question from a knowledge base (e.g., Wikipedia). Then, the retrieved knowledge

ZHEJIANG UNIVERSITY

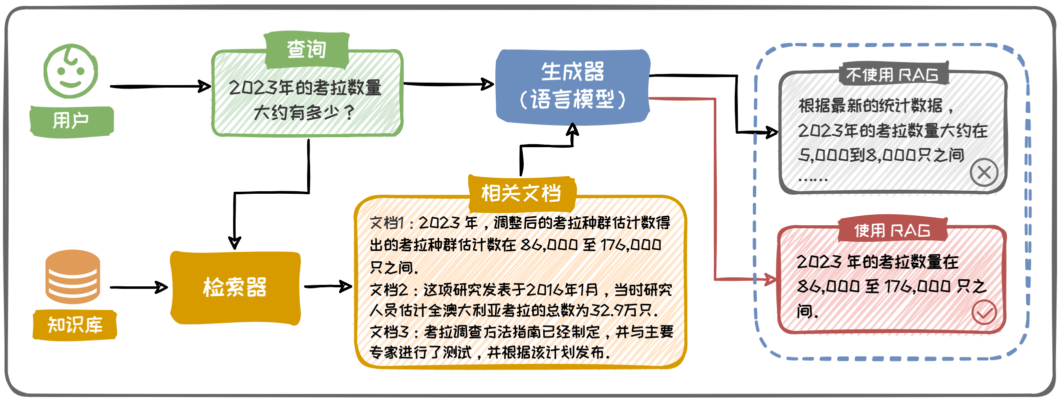

查询
2023年的考拉数量
大约有多少？
用户
生成器
（语言模型）
不使用 RAG
根据最新的统计数据，
2023年的考拉数量大约在
5,000到8,000只之间
……
相关文档
文档1：2023 年，调整后的考拉种群估计数得出的考拉种群估计数在 86,000 至 176,000 只之间.
文档2：这项研究发表于2016年1月，当时研究人员估计全澳大利亚考拉的总数为32.9万只.
文档3：考拉调查方法指南已经制定，并与主要专家进行了测试，并根据该计划发布.
使用 RAG
2023 年的考拉数量在
86,000 至 176,000 只之
间.
知识库
检索器

Figure 6.8: Schematic diagram of the basic RAG architecture.

and the original question are passed together to the large language model, which generates the final output based on the retrieved knowledge and the original question.The core advantage of RAG is that it does not need to update the internal knowledge of the large language model in order to improve the phantom phenomenon of the large language model and improve the quality of generation. This can effectively avoid the computational cost of internal knowledge updating and Catastrophic Forgetting of old knowledge.

Next, we describe the basic workflow of RAG through the example in Figure 6.8. A user enters a question "**How many koalas will there be in 2023?**", first, the question is passed to the Retriever module of the RAG framework, which retrieves relevant knowledge documents from the knowledge base containing information related to the number of koalas in the year 2023; next, this information is passed to the Big Language Model in the form of a Prompt (Big Language Models utilize external knowledge in various forms, and contextual learning through Prompts is one of the most commonly used form), which eventually leads to the correct answer, "**The number of koalas in 2023 is between 86,000 and 176,000.**" However, the same question could not be answered correctly if the large language model was allowed to answer it directly without the use of RAG, demonstrating the effectiveness of the RAG system.

Simply connecting functional modules such as external knowledge bases, searchers, and large language models will not maximize the utility of RAG. This chapter will explore how to optimize the design of a RAG system around the following three questions.

- **How to optimize the collaboration between the retriever and the big language model?** Depending on whether or not fine-tuning is performed on the large language model, we classify existing RAG systems into **(1) black-box augmentation architectures**, which do not access the internal parameters of the model and optimize only using output feedback, and **(2) white-box augmentation architectures**, which allow fine-tuning of the large language model. Details will be presented in Section 6.2.
- **How to optimize the retrieval process?** Discusses how to improve the quality and efficiency of retrieval, mainly including: **(1) Knowledge Base Construction**, constructing comprehensive and high-quality knowledge bases and enhancing and optimizing them; **(2) Query Enhancement**, improving the original query so that it is more precise and easy to match the information in the knowledge base; **(3) Retrievers**, introduces common retriever structures and search algorithms; **(4) Retrieval Efficiency Enhancement**, introducing common similarity indexing algorithms used to improve retrieval efficiency; **(5) Rearrangement Optimization**, filtering out more effective information through document rearrangement. Details will be presented in Section 6.3.
- **How to optimize the enhancement process?** Discusses how to utilize retrieved information efficiently, mainly including: **(1) When to Enhance**, determines when retrieval enhancement is needed to improve efficiency and avoid interfering informa-

tion; **(2) Where to Enhance**, discusses the common location of inserting retrieved information in the generating process; **(3) Multiple Enhancement**, discusses for complex and fuzzy queries, discusses the common multiple enhancement methods; **(4) Cost Reduction and Efficiency**, describing existing knowledge compression and cache acceleration strategies. Details will be presented in Section 6.4.

This section provides an initial introduction to the questions **Why do we need RAG** and **What is RAG**. The next sections will discuss the specific technical details in detail for the three issues raised above.

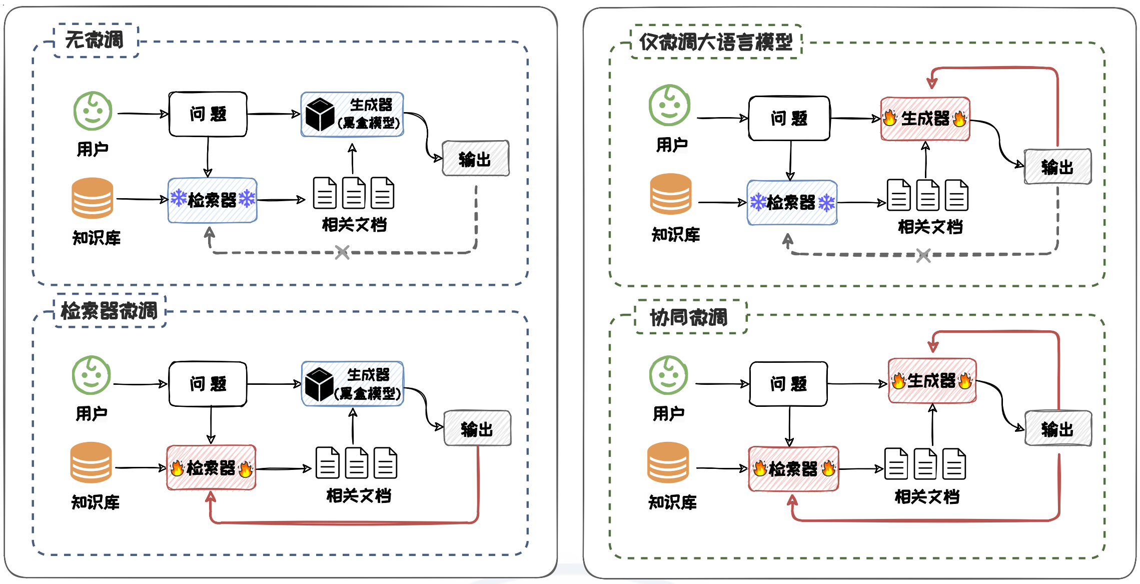
6.2 Retrieving Enhanced Generation Architecture

The Retrieval Augmented Generation (RAG) system is a software system that integrates a number of functional modules such as external knowledge bases, retrievers, and generators. For different business scenarios and requirements, different system architectures can be designed to combine and coordinate these modules to optimize the performance of RAG. Among them, the collaboration methods of the retrievers and generators have the most significant impact on the performance of RAGs. This is because the quality of information retrieved by the retriever varies and the quality of content generated by the generator varies under different collaboration styles. In addition, the way of collaboration between the retriever and the generator has a significant impact on the efficiency of the system. Efficient collaboration reduces latency and improves the responsiveness of the system. In this section, we will take a look at the classic RAG architecture and introduce it from the perspective of **how to optimize the collaboration between retrievers and large language models**.

6.2.1 RAG architecture classification

For different business scenarios, the generator in RAG can choose different large language models, such as GPT-4[1], LLaMA[49] and so on. Considering the open/closed source and fine-tuning cost of the large language model, the large language model in the RAG can be a "black-box" model whose parameters cannot be sensed/adjusted, or a "white-box" model whose parameters can be sensed and fine-tuned. For example, if GPT-4 is chosen, due to its closed source, it can only be regarded as a "black box" in the RAG process, and its output can only be utilized, but its model parameters cannot be sensed/fine-tuned. If the LLaMA model is chosen, it can be considered as a "white box" and fine-tuned during the RAG process if computational resources allow. From the point of view of whether to fine-tune the LLaMA model or not, this subsection categorizes the RAG architectures into two main types: **black box enhanced architecture** and **white box enhanced architecture**, as shown in Figure 6.9.

Among them, black-box enhancement architectures can be categorized into two types according to whether or not the retriever is fine-tuned: **no fine-tuning**, **retriever fine-tuning**, as shown in Figure 6.9(a). In the no-fine-tuning architecture, neither the retriever nor the large language model undergoes any fine-tuning, relying only on the capabilities they mastered in the pre-training phase to accomplish the corresponding retrieval and generation tasks. In the retriever-fine-tuned architecture, the language model parameters remain unchanged, while the retriever performs targeted tuning of the parameters based on the output feedback from the language model. Similarly, white-box enhancement architectures can be categorized into two types depending on whether or not the retriever is fine-tuned: **fine-tuning of the large language model only**, **fine-tuning of the retriever**



(a)黑盒增强架构 (b) 白盒增强架构

Figure 6.9: Retrieve the Enhanced Architecture categorization chart. Modules containing blue snowflakes indicate that their parameters are frozen, and sections with red flames indicate that their parameters are updated during fine-tuning.

in conjunction with the large language model (hereinafter referred to as co-fine-tuning), as shown in Figure 6.9(b). In the architecture of fine-tuning only the large language model, the parameters of the retriever as a pre-trained component remain unchanged; the language model is parameterized based on the relevant information provided by the retriever. In the collaborative fine-tuning architecture, the retriever and the large language model iteratively interact and collaboratively fine-tune.

In the RAG system, in addition to tuning the retriever and the grand language model, we can also tune other functional modules (e.g., vectors in the knowledge base[17, 45]). Tuning other functional modules is compatible with the classification of black-box enhancement and white-box enhancement. The next part of this section describes the black-box enhancement architecture and the white-box enhancement architecture in detail and explores their representative approaches.

6.2.2 Blackbox Enhanced Architecture

In some cases, we have to treat the language model as a black box due to the lack of access to the structure and parameters of the larger language model or the lack of sufficient arithmetic to fine-tune the model, e.g., when interaction can only be performed via APIs. At this point, the RAG needs to be built on top of the black-box augmentation architecture. In the black-box augmented architecture, we can only perform policy tuning and optimization of the retriever. It can be categorized into two types of architectures: no fine-tuning architecture and retriever fine-tuning architecture. The two types of architectures are described next.

1. no-fine-tuning

The no-fine-tuning architecture is the simplest form of all RAG architectures. In this architecture, the retriever and the language model are pre-trained separately and independently **parameters are no longer updated**, **used directly in combination**. This architecture has a low demand for computational resources, is convenient to implement and easy to deploy, and is suitable for scenarios with high requirements for deployment speed and flexibility.In-Context RALM[42] is a representative approach under this framework. It directly precedes the document retrieved by the retriever to the input question as the context, and the schematic diagram of the method is shown in Figure 6.10. In-Context RALM consists of two phases: retrieval and generation. In the retrieval phase, the input question or part of a sentence is used as a query to retrieve relevant documents from the knowledge base. In the generation phase, these retrieved documents are directly spliced into the contextual parts in the Prompt, and then the Prompt is input to the Big Language Model. A RAG task may involve multiple executions of retrieval and generation. For ex-

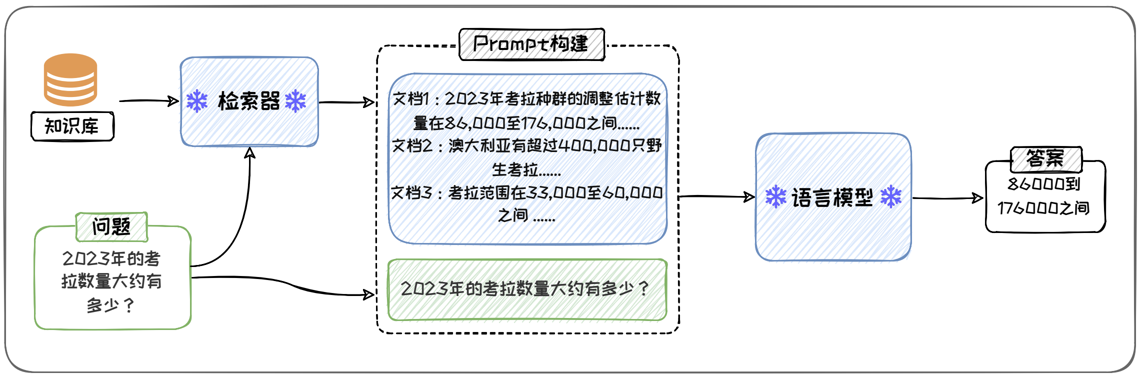

Prompt构建
知识库
检索器
文档1：2023年考拉种群的调整估计数量在86,000至176,000之间......
文档2：澳大利亚有超过400,000只野生考拉......
文档3：考拉范围在33,000至60,000之间 ......
语言模型
答案
86000到176000之间
问题
2023年的考拉数量大约有多少？
2023年的考拉数量大约有多少？

Figure 6.10: In-Context RALM model architecture diagram.

ample, in a long text generation task, after every certain amount of text is generated, the model may perform a retrieval to ensure that subsequently generated content continues to remain relevant to the topic as it evolves.

When performing the retrieval operation, several key parameters need to be carefully selected, such as **retrieval step** and **retrieval query length**. Retrieval step length refers to how many words the model retrieves every time it generates text, and the setting of this parameter has a direct impact on the response speed of the model and the immediacy of the information. A shorter retrieval step can provide more timely information update, but at the same time, it may increase the computational complexity and resource consumption. Therefore, a reasonable balance between the two needs to be found in practical applications. The retrieval query length refers to the length of the text fragment used for retrieval, which is usually set to the last few words in the language model input to ensure that the retrieved information is highly relevant to the current text generation task.

2. Retriever fine-tuning

Although the no-fine-tuning architecture is very easy to implement and deploy, it does not take into account the potential synergistic effect between the retriever and the language model at all, and the effect needs to be improved. To further improve the effectiveness, the retriever can be fine-tuned using the retriever fine-tuning architecture to be better suited for black-box augmented environments. In the retriever fine-tuning architecture, the **parameters are kept unchanged** of the large language model is used to guide the fine-tuning of the retriever using only its output. Retrievers in this architecture are better adapted to the needs of the large language model, thus improving the performance of the RAG.

REPLUG LSR[45] is a representative approach of the retriever fine-tuning framework, whose structure is shown in Figure 6.11. It uses the perplexity score of a large language model as a supervisory signal to fine-tune the retriever so that it can more efficiently retrieve documents that significantly reduce the perplexity of the language model. Its process of fine-tuning the retriever employs a KL scatter loss function to train the retriever, with the aim of aligning the relevance distribution of retrieved documents with the distribution of the contribution of these documents to the performance improvement of the language model. This process involves two key probability distributions, the first one being **distribution of documents output by the retriever**: the retriever retrieves documents relevant to the current context upon receiving it and forms a document probability distribution. This distribution is based on the similarity between the context and the documents calculated by the retriever, measured by cosine similarity, and converts these similarity scores into probability values. The second one is **distribution of document contributions to the language model**: the language model generates predictions for each retrieved document and the original context, and eventually all the outputs form a probability distribution. In this distribution, if a document is particularly critical for the language model to generate accurate predictions, it will be given a higher probability weight. During the fine-tuning process, REPLUG LSR treats the language model as a black-box process and guides the training of the retriever only through the output of the model, avoiding access

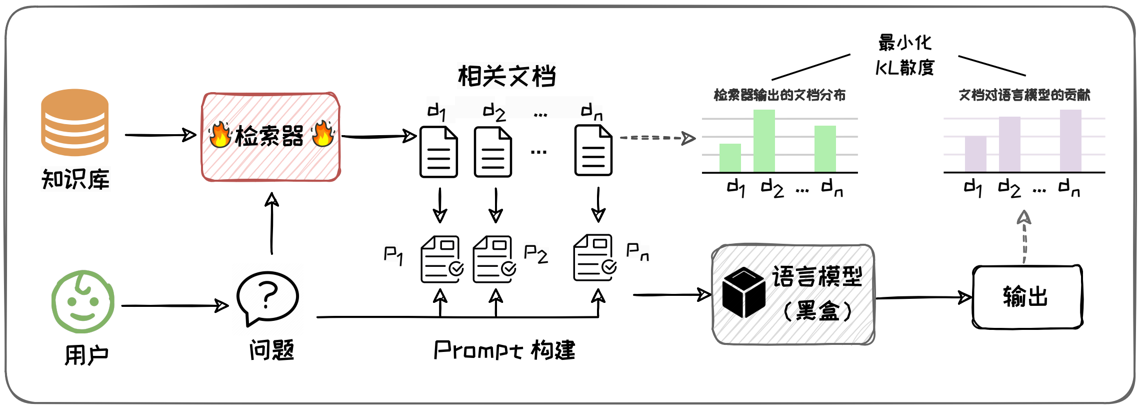


Figure 6.11: REPLUG LSR model architecture diagram.

to and modification of the internal structure of the language model. In addition, during the fine-tuning process, REPLUG LSR employs an asynchronous index update strategy, i.e., instead of updating the vector encoding of the knowledge base immediately after each training step, the update is performed after a certain number of training steps. This strategy reduces the frequency of index updates, decreases the computational cost, and enables the model to better adapt to new data during continuous training. In addition, the retriever fine-tuning framework gives the possibility to introduce agent models to guide the retriever fine-tuning. For example, the AAR[60] approach passes into introducing an additional small language model that uses its cross-attention score to label preferred documents as a way to fine-tune the retriever so that it can enhance its performance on different tasks without fine-tuning the target language model.

Retriever fine-tuning allows even closed-source large models such as ChatGPT to improve performance by optimizing external retrievers.REPLUG LSR and AAR achieve in this way the ability to enhance the model through external tuning while maintaining the integrity of the large model, which is not common in traditional RAGs.

6.2.3 white-box augmented architecture

Usually, the big language model and the retriever are pre-trained independently, and there may be poor matching between the two. White-box augmentation architecture enhances RAG by fine-tuning the big language model to match the retriever. It can be categorized into two types according to whether or not the retriever is fine-tuned: **fine-tuning of the language model only**, **co-fine-tuning of the retriever and the language model**.

1. fine-tuning language model only

Fine-tuning only language model refers to the fact that the retriever as a pre-trained component has its parameters kept unchanged, and the large language model fine-tunes **its own parameters based on the contextual information provided by the retriever**.RETRO[5] is one of the representative methods of fine-tuning only language model. This method realizes the enhancement of the large language model with external knowledge by modifying the structure of the language model so that the text retrieved from the knowledge base can be directly incorporated into the intermediate state of the language model during the fine-tuning process. In addition, SELF-RAG[3] enables the language model to dynamically decide whether or not it needs to retrieve external texts during the generation process, and to self-critique and optimize the generation results by introducing reflective markers when fine-tuning the language model. These methods not only improve the quality and factual accuracy of the generated content, but also enhance the knowledge integration and application capabilities of the model.

Taking RETRO as an example, its structure is shown in Fig. 6.12.RETRO first chunks the text in the knowledge base, and then generates embedding vectors for each text chunk using BERT. During the autoregressive process when fine-tuning the model, whenever the

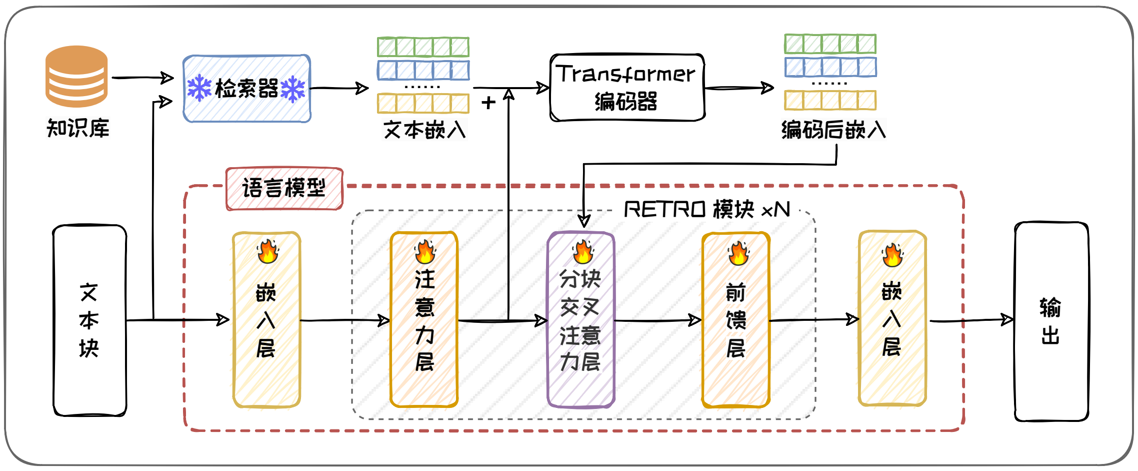


Figure 6.12: RETRO model architecture diagram.

model generates a block of text, it goes to the knowledge base and retrieves the embedding vectors that are most similar to it. These embedding vectors are then fed into an external Transformer encoder along with the output of the model's attention layer for encoding. The resulting encoded vectors are directly fed into the keys (key) and values (value) of the model's block cross-coder to capture key information about the external knowledge. Through cross-coding, the model is able to combine the retrieved relevant information to generate new blocks of text.

The RETRO model, fine-tuned in the above way, is able to fully integrate the retrieved information and generate coherent and informative texts. When faced with a user query, the model demonstrates excellent comprehension and knowledge integration capabilities, which significantly improves the quality and accuracy of the generation, especially when dealing with complex tasks.

2. Retriever and language model synergistic fine-tuning

In a fine-tuned language model-only architecture, the retriever acts as a fixed component whose parameters remain unchanged during the fine-tuning process. This results in the inability of the retriever to adapt to the needs of the language model, thus limiting

1897 ZHEJIANG UNIVERSITY

the mutual synergy between the retriever and the language model. In the architecture of collaborative fine-tuning between the retriever and the language model, the **parameter update is synchronized** of the retriever and the language model. This fine-tuning allows the retriever to learn how to support the needs of the language model more efficiently while retrieving, while the language model can better adapt and utilize the retrieved information to further improve the performance of the RAG.

Atlas[17] is a representative work of this architecture, whose architecture is shown in Figure 6.13. Similar to REPLUG LSR, it uses the KL scatter loss function in the pre-training and fine-tuning phases to jointly train the retriever and the language model to ensure that the document relevance distribution of the retriever output is consistent with the distribution of document contributions to the language model. The difference is that in Atlas, the retrieval and language model parameters are updated synchronously during the pre-training and fine-tuning process, where the retriever learns to provide the most relevant documents to the language model, and the language model learns how to utilize these documents to improve its response to the query. To ensure that the retrieval results are synchronized with the latest state of the model, Atlas also needs to periodically update the vector encoding of the corpus documents to maintain retrieval accuracy.

6.2.4 Comparison and Analysis

This section introduces the black-box enhancement architecture and white-box enhancement architecture of RAG and their classical approaches. Next, we summarize and compare these two architectures.

The **black-box enhancement architecture** is proposed in the context of closed-source modeling, which restricts the direct tuning of parameters inside the model. In

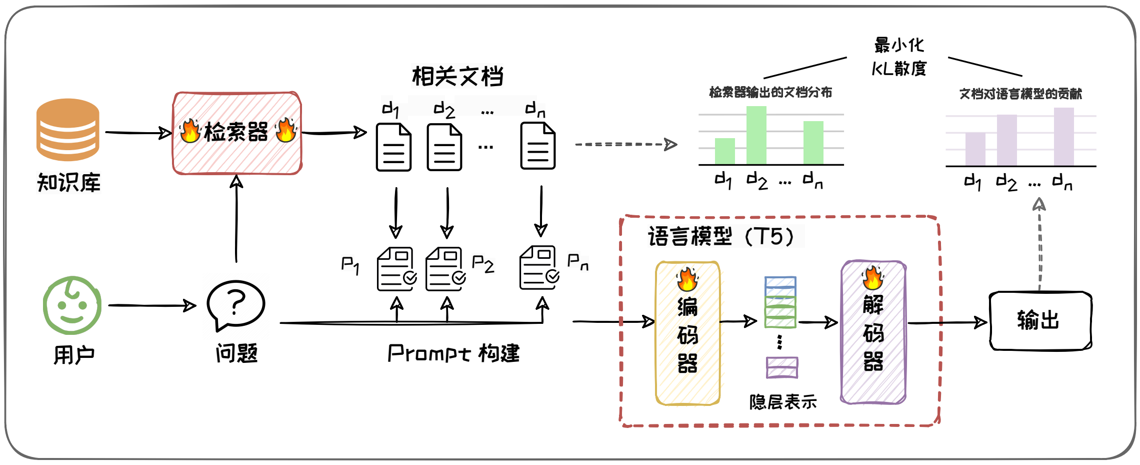


Figure 6.13: Atlas model architecture diagram.

this architecture, we introduce two strategies, **no fine-tuning** and **retriever fine-tuning**. **no fine-tuning** is simple and practical in that it directly utilizes the pre-trained language model and the retriever without any updates, making it suitable for rapid deployment. However, the disadvantage of this approach is that the language model cannot be optimized to adapt to new task requirements. In contrast, **retriever fine-tuning** offers the possibility of improving performance without being able to modify the language model by adapting the retriever to the language model output. The effectiveness of this approach depends heavily on the accuracy of the tuned retrievers.

The **white-box enhancement architecture**, on the other hand, takes advantage of the open-source model by allowing the tuning of the language model structure and parameters, which allows a better coordination of the reducer and the larger language model. In this architecture, we introduce two forms of fine-tuning: **fine-tuning the language model only** and **collaborative fine-tuning of the retriever and the language model**. **Fine-tuning the language model only** focuses on optimizing the language model by adjusting only the language model structure and parameters based on retrieved information to improve performance on specific tasks. **Retriever and language model co-fine-tuning** is a more dynamic strategy that improves the overall system performance by synchronizing the updating of the retriever and the language model so that the two can be adapted to each other during the training process. Although white-box augmentation architectures can effectively improve the performance of RAGs, they also have significant drawbacks. This architecture usually requires a lot of computational resources and time for training, especially for the collaborative fine-tuning strategy, which requires a lot of computational resources for synchronized updating of the language model and the retriever.

6.3 knowledge retrieval

In RAG, the effect of retrieval (recall, precision, diversity, etc.) will directly affect the quality of the generation of the large language model. Take the question "Where do wombats usually live?" This question is an example. If the external knowledge returned by the retriever is related to the animal "wombat", which is similar to the name "wombat", the incorrect external knowledge may lead the large language model to generate an incorrect answer. In addition, the retrieval time is also a key part of the total time spent on RAG, so the efficiency of retrieval will affect the user experience. Optimizing the retrieval process to enhance the effectiveness and efficiency of retrieval is important to improve the performance of RAG. For optimizing the retrieval process, this section systematically sorts out and introduces the key technologies such as knowledge base construction, query enhancement, searchers, and reordering of retrieval results.

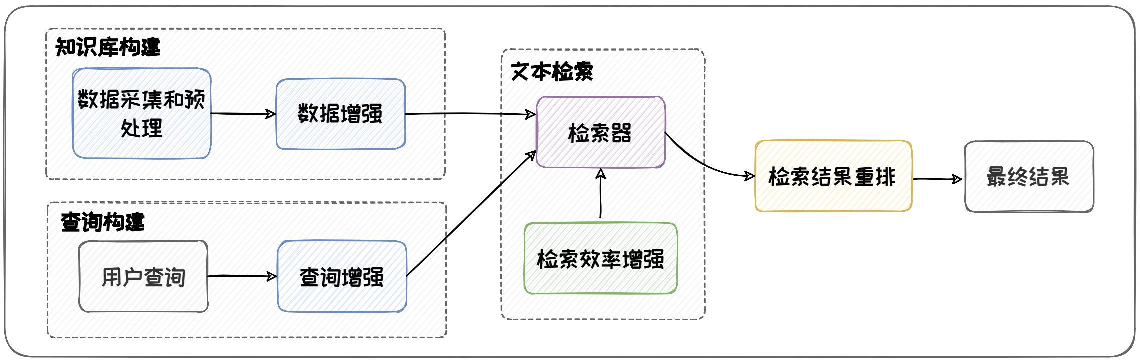


Figure 6.14: Knowledge Retrieval Flowchart.

6.3.1 knowledge base construction

The knowledge base constitutes the root of the RAG system. As the old saying goes: "A skillful woman cannot cook without rice", only by constructing a comprehensive, high-quality and efficient knowledge base, the retrieval can be targeted and the retrieval effect can be guaranteed. In the RAG framework, knowledge base construction mainly involves two steps, **data collection and preprocessing** and **knowledge base enhancement**. This subsection will introduce these two steps respectively.

1. data collection and pre-processing

Data collection and pre-processing provide the "raw material" for building a knowledge base. In the **data collection**process of building a text-based knowledge base, data from different sources are integrated and converted into unified document objects. These document objects not only contain the original textual information, but also carry meta-information (Metadata) about the document. Metadata can be used for subsequent retrieval and filtering. Taking the construction of Wikipedia corpus as an example, data collection is mainly realized by extracting the contents of Wikipedia website pages. These contents include not only the content of the body description, but also a series of meta-information, such as article title, classification information, time information, keywords and so on.

After collecting the corresponding data, it is also necessary to improve the data quality and usability through **data preprocessing**. When building a text-based knowledge base, data preprocessing mainly includes two processes: data cleaning and text chunking. **data cleansing** aims to remove distracting elements in the text, such as special characters, unusual coding and useless HTML tags, as well as removing redundant documents that are duplicated or highly similar, so as to improve the clarity and usability of the data. **text chunking** is the process of splitting long text into smaller chunks of text, such as dividing a long article into multiple short paragraphs. There are two benefits of chunking long text: one is to adapt to the length limit of the context window of the retrieval model to avoid exceeding its processing capacity; the other is that chunking can reduce irrelevant content in long text and reduce noise, thus improving the efficiency and accuracy of retrieval.

The effect of text chunking directly affects the quality of subsequent retrieval results[14]. If chunking is not handled properly, it may destroy the coherence of the content. Therefore, it is crucial to develop an appropriate chunking strategy, including determining the slicing method (e.g., by sentence or paragraph), setting the block size, and whether to allow overlap between blocks. The process of implementing text chunking usually starts with breaking down long texts into smaller semantic units, such as sentences or paragraphs. Subsequently, these units are progressively combined into larger chunks until a preset chunk size is reached, constructing separate text fragments. In order to maintain semantic coherence, it is also common to place a certain amount of overlap between neighboring text fragments.

2. Knowledge Base Enhancement

Knowledge base enhancement is the process of improving and enriching the content and structure of a knowledge base to enhance its quality and usefulness. This process usually involves a number of steps such as **query generation** and **title generation**[56], in order to establish semantic “anchors” for the documents, and to facilitate the retrieval of the corresponding text to be located accurately.

query generation refers to the use of large language models **generate pseudo-queries closely related to the content of the document**. These pseudo-queries express the semantics of the document from the perspective of the query, and can be used as the “key” of the relevant document for matching with the user’s query during retrieval. In this way, the match between documents and user queries can be enhanced. For example, for a document that introduces the relationship between koalas and wombats, the query “What is the relationship between koalas and wombats?” is generated. not only accurately reflects the topic of the document, but also effectively guides the searcher to more accurately retrieve information related to the user’s question.

Title Generation refers to the use of the large language model **Generate suitable titles for documents without titles**. These generated headings provide keywords and contextual information about the document, which can be used to help quickly understand the content of the document and more accurately locate information relevant to the user’s question during retrieval. Generating headings through language modeling is especially important in cases where the original document lacks headings.

6.3.2 query enhancement

The knowledge base covers a limited number of knowledge expressions, but the way users ask questions is very different. The way users formulate sentences and the angle of describing a question may differ from the text stored in the knowledge base, which may lead to a poor match between the user query and the knowledge base, thus reducing the retrieval effect. To solve this problem, we can extend the semantics and content of the user query, i.e. query enhancement, to better match the text in the knowledge base. In this subsection, we will briefly introduce the query enhancement techniques from the perspectives of **Query Semantic Enhancement** and **Query Content Enhancement**.

1. Query Semantic Enhancement

Query semantic enhancement aims to extend and enrich the semantics of user queries through methods such as **synonymous rewriting** and **multi-view decomposition** in order to improve the accuracy and comprehensiveness of retrieval. The synonymous rewriting and multi-view decomposition are briefly introduced next.

(1) Synonymous Rewriting

Synonymous rewriting solves the problem that a single expression of a user query may not be able to fully cover the diversely expressed knowledge in the knowledge base by rewriting the original query into different expressions under the same semantics. The rewriting work can be accomplished by invoking a large language model. For example, for such an original query: "What are the eating habits of koalas?" , it can be rewritten into the following tautological expressions: 1. "What do koalas mainly eat?"; 2. "What are the foods of koalas?"; 3. "What is the structure of a koala's diet?" . Each rewritten query can be used independently to retrieve related documents, and the collection of documents retrieved from these different queries is subsequently merged and de-duplicated to form a larger collection of related documents.

(2) Multi-Perspective Decomposition

Multi-perspective decomposition uses a divide-and-conquer approach to complex queries, whereby a complex query is decomposed into subqueries from different perspectives in order to retrieve information from different perspectives relevant to the query. For example, for a question such as "What are the threats to koalas?", it can be decomposed into multiple perspectives as 1. "How does habitat loss affect koalas?"; 2. "How does climate change affect the survival of koalas?"; 3. "What are the threats to koala populations from human activities?"; 4. "What are the effects of natural disasters on koalas?" and other sub-questions. Each sub-question retrieves different relevant documents that provide information from different perspectives. By synthesizing this information, the language model is able to generate a more comprehensive and in-depth final answer.

2. Query Content Enhancement

Query content enhancement aims to improve the accuracy and comprehensiveness of retrieval by generating background information and context related to the original query, thus enriching the query content [58]. Compared with the traditional approach that relies only on retrieval, the query content enhancement method provides more dimensional information support for the original query by introducing auxiliary documents generated by the large language model.

Generating background documents is an approach for query content enhancement. It means that based on the original query, a large language model is utilized to generate background documents related to the query content. For example, for the user query "How to protect the koala's habitat?", the following background document can be generated:

> *The Koala is an arboreal marsupial native to Australia, mainly found in eucalyptus forests along the eastern and southeastern coasts. These areas provide the koala's main food source, eucalyptus leaves. Koala habitats include open forests and woodlands where eucalypts are abundant and provide not only food, but also shelter and protection. Koalas are highly dependent on specific species of eucalypts and their distribution is closely linked to the*

availability of these trees.

These generated contextual documents can be used as additional information to the original query to provide more contextual content, thus improving the relevance and richness of the retrieval results.

6.3.3 retriever

Given a knowledge base and a user query, a retriever aims to find the knowledge text in the knowledge base that is relevant to the user query. Retrievers can be categorized into two types of **discriminative retrievers** and **generative retrievers**. This subsection will introduce each of these two types of retrievers.

1. Discriminative Retriever

Discriminative retrievers score the relevance of a query and a document by means of a discriminative model. Discriminative retrievers are usually categorized into two main types: **sparse retrievers** and **thick retrievers**. Sparse retrievers utilize discrete, word-frequency based document encoding vectors for retrieval, while dense retrievers utilize continuous, dense vectors generated by neural networks for document retrieval. Both retrievers and representative methods are described in detail below.

(1) Sparse Retriever

Sparse Retriever (Sparse Retriever) is a model that uses **sparse representation** to match text. This type of retriever encodes a document by counting the statistical features of the occurrence of specific lexical items in the document, and then retrieves the query based on this encoding by calculating the similarity between the query and the documents in the knowledge base. Typical sparse retrieval techniques include TF-IDF[2] and BM25[43], etc., which evaluate the relevance of a document to a query by analyzing the distribution and frequency of lexical items.

TF-IDF measures the importance of words in a document or corpus based on word frequency (TF) and inverse document frequency (IDF)， and then uses this importance to encode the text. The word frequency (TF) indicates the frequency of occurrence of a word in a document and is calculated as:

$$\mathrm{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}, \tag{6.1}$$

where $n_{i,j}$ is the number of occurrences of the word t_i in the document d_j, and $\sum_k n_{k,j}$ is the sum of occurrences of all the words in the document d_j, which is used to perform the normalization in order to avoid bias towards long documents. Inverse Document Frequency (IDF) measures the prevalence of words and is calculated as:

$$\mathrm{idf}_i = \log \frac{|D|}{|\{j : t_i \in d_j\}|}, \tag{6.2}$$

where $|D|$ is the total number of documents and $|\{j : t_i \in d_j\}|$ is the number of documents containing the word t_i. Finally, the TF-IDF value is:

$$\mathrm{tfidf}_{i,j} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i. \tag{6.3}$$

TF-IDF generates high weights by high word frequency and low document frequency, which tends to filter common words and retain important words.

BM25 is an improved text retrieval algorithm, which evaluates the importance of word items more accurately based on TF-IDF through document length normalization and word item saturation adjustment, optimizes the computation of word frequency and inverse document frequency, and takes into account the effect of document length on scoring. Although it does not involve word-item context, BM25 performs well in handling large-scale data and is widely used in search engines and information retrieval systems.

(2) Dense Retrievers

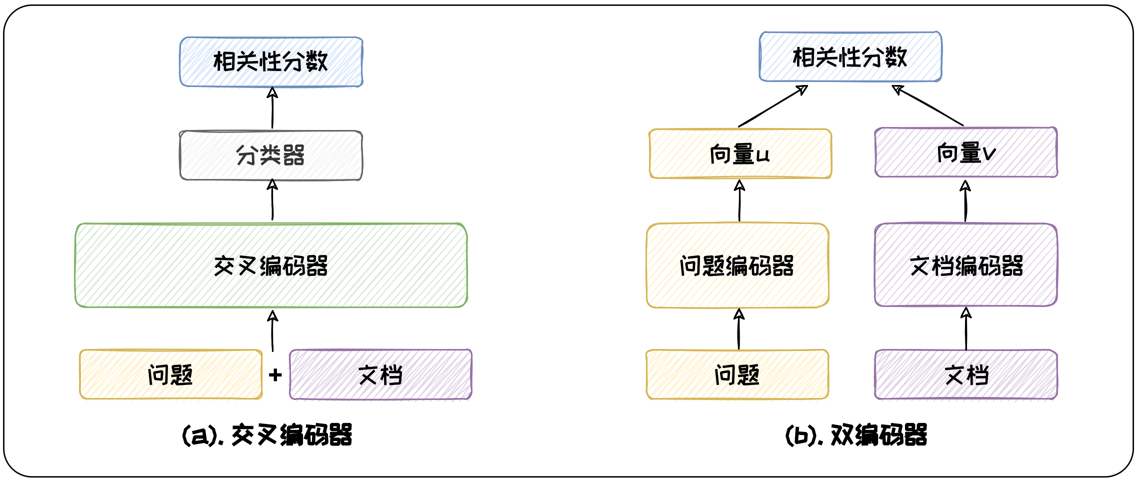


Figure 6.15: Comparison graph of different dense retrievers.

Dense retrievers generally utilize **pre-trained language model** to generate **low dimensional**, **dense** vector representations of the text, which are retrieved by calculating the similarity between the vectors. According to the difference in the structure of the model used, dense retrievers can be roughly categorized into two classes: **Cross-Encoder class** (Cross-Encoder), **Bi-Encoder class** (Bi-Encoder). The structures of the two are shown in Figures 6.15 (a) and 6.15 (b), respectively.

cross-coding class

Cross-coding classes give the query and document similarity “end-to-end” . This class stitches together the query and the document, and subsequently generates a vector representation using a pre-trained language model as an encoder (e.g., BERT). This vector is then processed by a classifier, which outputs a value between 0 and 1 indicating the degree of similarity between the input query and the document. The advantage of this is that the model structure is simple and enables deep interaction between the query and the document, e.g., in the work [12, 44], the researchers used a cross-coder to improve the retrieval performance. However, the cross-coder class of models is not suitable for use in the large-scale retrieval phase due to the high complexity of the crossattention operation and the high computational effort required. Such models are more suitable for the stage of more accurate ranking of a small number of candidate documents, which can significantly improve the relevance of retrieval results.

Dual Encoder Class

Unlike the cross-coder class model, the dual-coder class model uses a “two-step” strategy. In the first step, the The first step is that the query and the document are first generated as separate vector representations by independent encoders; in the second step, the similarity between the two vectors is computed in order to evaluate their relevance. The advantage of this approach is that it allows the vector representations of all documents to be computed and stored offline in advance, and vector matching to be performed directly during online retrieval. Therefore, the dual encoder is well suited for deployment in industrial environments with very high matching efficiency. However, in this separated processing, there is a lack of interaction between the query and the documents in extracting the feature vectors. This may have an impact on the accuracy of the matching.DPR (Dense Passage Retriever)[23] is a representative work of dense retrievers. Its uses two independent BERT encoders to map the query and the document to low-dimensional feature vectors respectively, and then measures the similarity by vector dot product. To alleviate the problem of lack of interaction between the query and the document, DPR optimizes the encoders by comparative learning to maximize the query’s similarity to relevant passages while minimizing the similarity to negative passages.

In order to alleviate the problem of the lack of interaction between query and document in extracting feature vectors, the interaction between query and document can be introduced on top of the dual encoder to further enhance the effectiveness of the dual encoder.ColBERT[24] is one of the representative methods. Its Token-level similarity between the query and the document is the metric, and then the dual encoder is fine-tuned by comparison learning so that the feature vectors encoded by the dual encoder can take into account both the query and the document. In addition, in RAG, we sometimes augment the query by adding a large context, as in the case presented in Section 3.3.2. At this time, the length of the query grows dramatically, and traditional retrieval methods may have difficulty in handling these long queries efficiently. To solve this problem, Poly-encoder[16] can be used. Its model architecture follows the form of a dual encoder, but it uses m vectors to capture multiple features of a long query, instead of representing the whole query with only one vector as in the case of a normal dual encoder. And, these m vectors then interact with the vectors of the document through an attention mechanism, where the attention module is trained using comparison learning between the query and the document.

2. Generative Retriever

A generative retriever generates identifiers of relevant documents [29]directly to the input query through a generative model. Unlike discriminative retrievers that constantly go through the knowledge base to match relevant documents, generative retrievers directly memorize document information from the knowledge base in the model parameters. It is then able to directly generate identifiers (i.e., DocIDs) of relevant documents to complete the retrieval [48] when a query request is received. Generative retrievers usually use generative models based on Encoder-Decoder architecture, such as T5[41], BART[27], and so on.

The training process for generative retrievers is usually divided into two phases [29]. In the first phase, the model learns how to map queries to relevant document identifiers through a sequence-to-sequence learning approach. This phase focuses on optimizing the model through maximum likelihood estimation (MLE) to ensure that the generated document identifiers are as accurate as possible. In the second phase, retrieval efficiency and accuracy are further improved through data enhancement and ranking optimization. Data enhancement is mainly done by generating pseudo-queries[51] or using document fragments[61] as query inputs to increase the diversity and coverage of the training data. Ranking optimization, on the other hand, involves the use of specific loss functions, such as contrast loss or ranking loss, to adjust the order and relevance of the document identifiers generated by the model to better match the query requirements.

The design of DocID is crucial in generative retrievers. It needs to strike a balance between the richness of semantic information and the simplicity of identifiers. Commonly used forms of DocID are categorized into two types: number-based DocID and word-based DocID. number-based DocID approaches use unique numeric values or integer strings to represent documents, which are simple to construct but may lead to a surge in the number of identifiers when dealing with a large number of documents, increasing the computational and storage burden. In contrast, word-based DocID approaches extract the representation [9]directly from the title, URL, or N-gram of a document, which can convey the semantic information of the document more naturally. Often, the title is the best choice because it provides a macro overview of the document. However, URLs or N-grams can also be effective alternatives when high-quality headings are lacking.

Although generative retrievers have made some progress in performance, they are still slightly less effective than dense retrievers. In addition, generative retrievers face a series of challenges, including how to break through the limitation of model input length, how to effectively handle large-scale documents, and how to learn the representation of dynamically added documents, etc., which are all urgent issues to be solved.

6.3.4 retrieval efficiency enhancement

Knowledge bases usually contain a huge amount of text, and the retrieval of the text in the knowledge base one by one is slow and inefficient. To enhance the retrieval efficiency, a vector database can be introduced to realize efficient vector storage and query in retrieval[39]. The core of vector databases is to design efficient similarity indexing algorithms. In this section, we will briefly introduce commonly used similarity indexing algorithms, as well as common software libraries used to construct vector databases.

1. Similarity Indexing Algorithms

In vector retrieval, commonly used indexing techniques are divided into three main categories: **space division based methods**, **quantization based methods** and **graph based methods**.

space division based method divides the search space into multiple regions to realize indexing, which mainly includes two categories: tree-based indexing method and hash-based method. Among them, the tree-based indexing method recursively divides the space through a series of rules to form a tree-like structure, where each leaf node represents a smaller region and the data points within the region are close to each other. When querying, the algorithm starts from the root node of the tree and progressively goes deeper to the appropriate leaf nodes, and finally compares the similarity of the data points within the leaf nodes to find the closest vector. Common tree-based indexes include KD tree[4] and Ball tree[13]. And hash-based methods (such as locality-sensitive hashing (LSH)[11]) map vectors into different buckets of the hash table by means of a hash function, so that similar vectors usually lie in the same bucket.

Graph-based methods transform vector retrieval into a graph traversal problem by constructing a neighborhood graph. Such methods represent each vector in the dataset as a node in the graph during the index construction phase and establish edge connections based on the distance or similarity between the vectors. Different graph indexing structures are mainly characterized by their unique strategies for assigning edges. The core idea of index construction originates from the small-world network model and aims to create a structure that allows reaching the nearest neighbor of a query point in fewer steps from any entry point. This structure allows the use of greedy algorithms during the search to progressively approximate the goal. However, graph indexing designs face a trade-off between sparsity and denseness: sparser graph structures have a low computational cost per step, while denser graphs may shorten the search path. Representative graph-based methods include NSW[32], IPNSW[37] and HNSW[31].

methods based on product quantization obtains codebooks and code words by dividing the high-dimensional vector space into subspaces and clustering in each subspace, which serves as the basis for constructing indexes[18]. This type of method mainly consists of two phases: training and querying. In the training phase, the method learns how to optimally quantize high-dimensional vectors into sequences of codeword IDs. Typically this process involves dividing the original space into multiple subspaces, clustering within each subspace, and obtaining the codewords and codebooks from the clustering centers. The center point of clustering within each subspace is the codeword, and the set of all codewords constitutes the codebook. Each subvector of each training sample can be approximated by the code words in the corresponding subspace, thus realizing the sequence of code word IDs to represent the training samples and achieving the purpose of data quantization. In the query phase, the system also divides the query vector into subvectors and finds the nearest codeword in each subspace to obtain the codeword ID sequence. Subsequently, the system calculates the distance from each subvector of the query vector to all the code words in the corresponding subspaces to form a distance table. Finally, the system uses this distance table and the code word ID sequences of each vector in the database to quickly find and accumulate the corresponding distances of each subvector to obtain the approximate distances between the query vector and the database vector. By sorting these distances, the system finally gets the nearest neighbor results. The method performs well in reducing memory occupation and speeding up distance calculation, but the quantization process will inevitably introduce some errors. In some application scenarios that require higher accuracy, we can further carry out accurate sorting on the basis of PQ to obtain accurate nearest neighbor results. In addition, there are some optimization algorithms for product quantization as well as algorithms combined with other indexes, such as OPQ[15], IVFPQ[19], and so on.

2. Introduction to Common Software Libraries

In the previous section, we have introduced similarity indexing algorithms, the core technology of vector databases. These algorithms are the key to realize efficient vector retrieval. Next, we will introduce several software libraries commonly used to build and manage vector databases, which support the similarity indexing algorithms mentioned above and are important tools to realize efficient vector retrieval.

Faiss[4], developed by Meta AI Research, is a library specialized in optimizing dense vector similarity search and clustering.Faiss offers a wide selection of indexing algorithms that cover the range of vector indexing algorithms based on space partitioning, quantization-based, and graph-based methods, among others. These algorithms not only run on the CPU, but some of them also support GPU acceleration to meet the performance

---

[4] `https://github.com/facebookresearch/faiss`

requirements of different applications. However, Faiss itself is not a complete database system, but a powerful tool library. It focuses on providing efficient indexing and searching functions, but in terms of data storage, management, distributed support, and security measures, Faiss has relatively limited functionality. In contrast, vector database is a more comprehensive solution, which not only includes similarity indexing algorithms, but also integrates data storage, management, distributed support and security measures and other aspects of the function. Up to now, there are several mature vector databases on the market, as shown in Table 6.1, which are suitable for a variety of more complex RAG application scenarios.

Table 6.1: Common Vector Database.

vector database	**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 retrieve results rearranged

The retriever may retrieve documents that are not highly relevant to the query. These documents may trigger a decrease in the quality of generation if they are directly input to the big language model. For this reason, we need to further select them before feeding them to the big language model. The main way of selection is to reorder the retrieved documents, referred to as re-ranking, and then select the top-ranked documents from them. Rearrangement methods are mainly categorized into two types: cross-coding based methods and context learning based methods.

1. Cross-Coding Based Rearrangement Methods

Cross-coding based rearrangement methods utilize **Cross-Encoders** to evaluate the semantic relevance between documents and queries. See Section 6.3.3 for an introduction to cross-coding. MiniLM-L[5] is one of the most widely used cross-coding based re-ranking one of the most widely used open-source models. The model improves model performance by reducing the number of parameters by reducing the number of layers and the number of hidden layer units, and by using knowledge distillation techniques to inherit learning from large, high-performance language models. In addition, there are other online rearrangement models that can be accessed directly through APIs, such as Cohere[6]. For readers wishing to explore other high-performance rearrangers, see the MTEB[7] leaderboard, which brings together a number of high-performance rearranging models.

2. Contextual Learning Based Rearrangement Approach

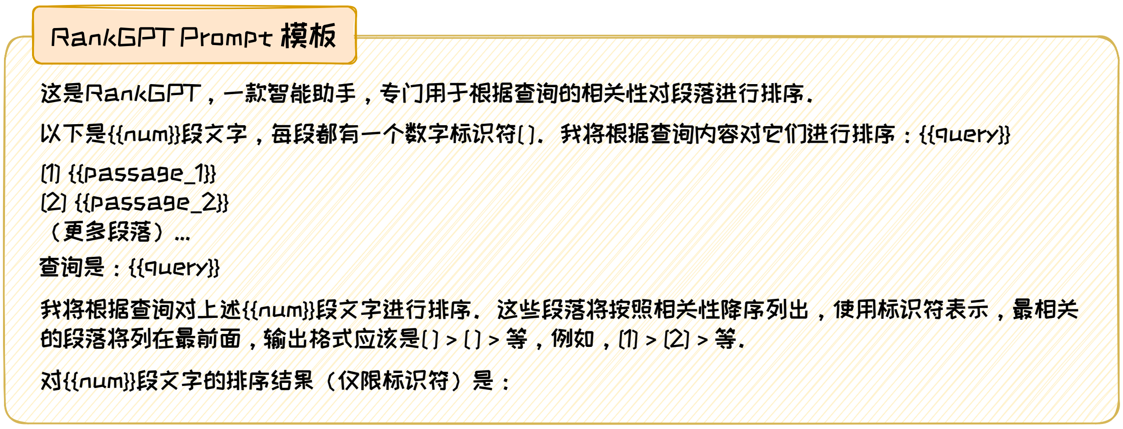
Contextual learning based approach refers to the use of a large language model to perform the rearrangement task through a well-designed Prompt. This approach can utilize the excellent deep semantic understanding of large language models, which leads to good performance.RankGPT[47] is a representative approach in the context-learning-based rescheduling methods. The Prompt template it uses is shown in Figure 6.16. In the rearrangement task, the input document length sometimes exceeds the limit of the context window length. To solve this problem, RankGPT uses a sliding window technique to optimize the sorting process. This technique splits all the documents to be sorted into several small consecutive parts, each of which acts as a window. The whole sorting process starts from the end of the document set: first, the documents within the last window are sorted

---

[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：

Figure 6.16: RankGPT Prompt template image.

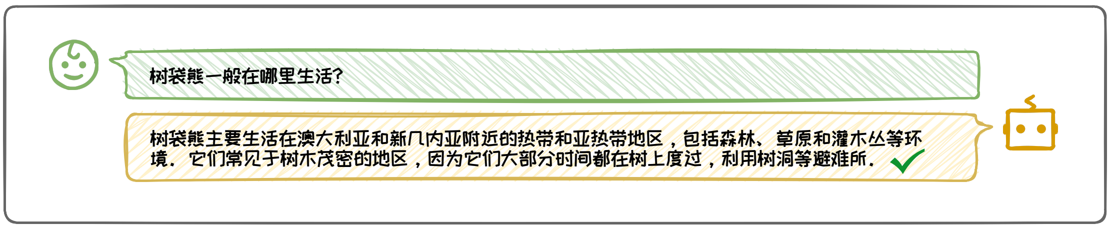
and the sorted results are replaced with the original order. Then, the window moves forward in a preset step and the process of sorting and replacing is repeated. This process continues until all documents have been processed and sorted. With this step-by-step approach, RankGPT is able to efficiently sort an entire collection of documents without being limited by the number of documents that can be processed by a single window.

6.4 Generation Enhancement

After obtaining the relevant information, the retriever passes it to the large language model with a view to enhancing the generative capabilities of the model. Using this information for generative enhancement is a complex process, and different approaches can significantly affect the performance of a RAG. This section will discuss four aspects from the perspective of **How to Optimize the Enhancement Process**: **(1) When to Enhance**, which determines when it is necessary to retrieve an enhancement to ensure that it is not non-essential; **(2) Where to Enhance**, which determines where to incorporate the retrieved external knowledge within the model to maximize the utility of the retrieval; and **(3) Where to Enhance**, which determines where the retrieved external knowledge is incorporated within the model to maximize the utility of the retrieval. textbf(3) Multiple Enhancement, how to perform multiple iterations of enhancement for complex and fuzzy queries to improve the effectiveness of RAG on difficult problems; **(4) Cost Reduction and Efficiency**, how to perform knowledge compression and cache acceleration to reduce the computational cost of the enhancement process.

6.4.1 When to Enhance

Large language models acquire a large amount of knowledge during training, which is called **Internal Knowledge (Self-Knowledge)**. For problems that can be solved by internal knowledge, we can leave the problem unenhanced. Blind enhancement without judging whether enhancement is needed will not only not improve the generation performance, but may also "add to the problem" and cause a double decline in **generation efficiency** and **generation quality**. For **generation efficiency**, the introduction of enhanced text will increase the number of input tokens and increase the computational cost of reasoning in large language models. In addition, the retrieval process involves a large amount of computational resources. For **Generation Quality**, since the retrieved external knowledge may sometimes be noisy, feeding it to the big language model not only does not improve the generation quality of the big language model, but may instead generate erroneous content. As shown in Figure 6.17, for the question "Where do wombats usually live?" This question, the big language model can give the correct answer directly. However, when we provide it with a paragraph of external knowledge about wombats, an animal with a very similar name to the wombat, as shown in Figure 6.18, the big language model gives the wrong answer because the big language model misinterprets the information about wombats in the knowledge text as relevant information about wombats.



树袋熊一般在哪里生活?

树袋熊主要生活在澳大利亚和新几内亚附近的热带和亚热带地区，包括森林、草原和灌木丛等环境. 它们常见于树木茂密的地区，因为它们大部分时间都在树上度过，利用树洞等避难所. ✓

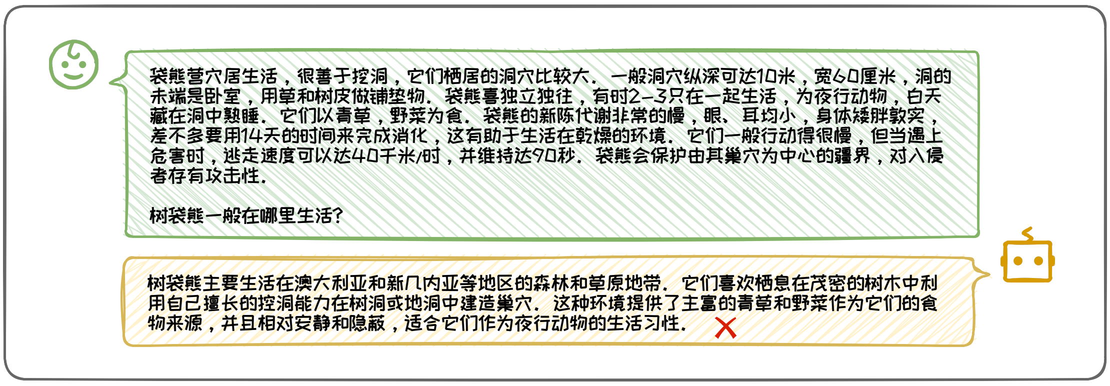
Figure 6.17: Example of a known problem with the model.

To summarize, judging when the big language model needs to retrieve enhancement and doing non-essential non-enhancement can effectively reduce computational cost and avoid erroneous enhancement.

The core of determining whether enhancement is necessary lies in **determining whether a big language model has internal knowledge**. If we judge that the big model has internal knowledge about a problem, then we can avoid retrieving the process of enhancement, which not only reduces the computational cost, but also avoids erroneous enhancement. The methods for judging whether a model has internal knowledge can be divided into two categories: (1) external observation method, which directly asks the model whether it has internal knowledge through Prompt or applies statistical methods to estimate whether it has internal knowledge, this method does not need to perceive the model parameters; (2) internal observation method, which detects the state information of the neurons inside the model to judge whether the model has internal knowledge, this method requires intrusive detection of model parameters.

1. external observation method

The external observation method aims to infer whether a large language model possesses internal knowledge without intruding into the internal parameters of the model by directly interrogating or observationally investigating its training data. This approach can be likened to the human interview process. When assessing a candidate's knowledge and



袋熊营穴居生活，很善于挖洞，它们栖居的洞穴比较大. 一般洞穴纵深可达10米，宽60厘米，洞的末端是卧室，用草和树皮做铺垫物. 袋熊喜独立独往，有时2-3只在一起生活，为夜行动物，白天藏在洞中熟睡. 它们以青草，野菜为食. 袋熊的新陈代谢非常的慢，眼、耳均小，身体矮胖敦实，差不多要用14天的时间来完成消化，这有助于生活在乾燥的环境. 它们一般行动得很慢，但当遇上危害时，逃走速度可以达40千米/时，并维持达90秒. 袋熊会保护由其巢穴为中心的疆界，对入侵者存有攻击性.

树袋熊一般在哪里生活?

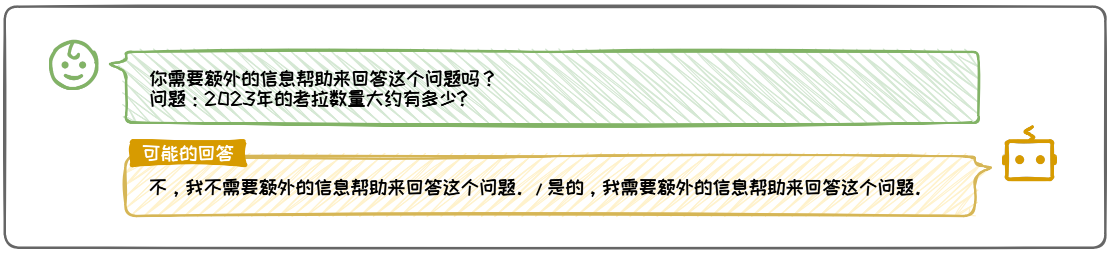
树袋熊主要生活在澳大利亚和新几内亚等地区的森林和草原地带. 它们喜欢栖息在茂密的树木中利用自己擅长的挖洞能力在树洞或地洞中建造巢穴. 这种环境提供了主富的青草和野菜作为它们的食物来源，并且相对安静和隐蔽，适合它们作为夜行动物的生活习性. ✗

Figure 6.18: Knowledge document damage performance example.

ability, an interviewer usually asks and observes his/her responses, browses through his/her past education, etc., to determine whether the candidate possesses sufficient expertise. For the large language model, we can determine whether the large language model has the corresponding internal knowledge by two kinds of questioning: (1) Prompt**directly ask** whether the large language model contains the corresponding internal knowledge; (2) **repeatedly ask** the large language model the same question to observe the consistency of the model's multiple responses. In addition, we can also look at the "educational experience" of a large language model, i.e., **training data**, to determine whether it has internal knowledge. However, the training data of many large language models are not publicly available, and it is not possible to observe their training data directly. In this case, the model's learning of specific knowledge can be indirectly assessed by designing **pseudo training data statistic** to fit the distribution of real training data. These three approaches are described in detail next.

(1) query

We can determine whether the big language model has internal knowledge by two kinds of inquiry methods: direct inquiry and multiple inquiry. In the case of direct questioning, we can write a prompt to directly ask whether the large language model needs



你需要额外的信息帮助来回答这个问题吗？
问题：2023年的考拉数量大约有多少?

不，我不需要额外的信息帮助来回答这个问题. / 是的，我需要额外的信息帮助来回答这个问题.

Figure 6.19: Example of Prompt for Direct Questioning.

external knowledge, as shown in Figure 6.19. However, it is difficult for a large language model to know what it knows, and it suffers from the problem of overconfidence: for unanswerable questions, the large language model often believes that it does not need the help of external knowledge, so the accuracy of this judgment is low. In addition, when using multiple questioning, we can ask the model to answer the same question many times, and then judge whether it has internal knowledge based on the consistency of its answers [34, 40]. If the model does not possess the appropriate internal knowledge, then the output will be more random each time, resulting in less consistency in multiple responses; conversely, if the model possesses internal knowledge, it will answer with the correct knowledge each time, with less randomness, and the output will have higher consistency. However, this approach also faces the problem that the model is overconfident and "stubbornly" gives the same wrong answer all the time. Furthermore, multiple queries are time and computationally intensive, making it less feasible in practice.

(2) Observation of training data

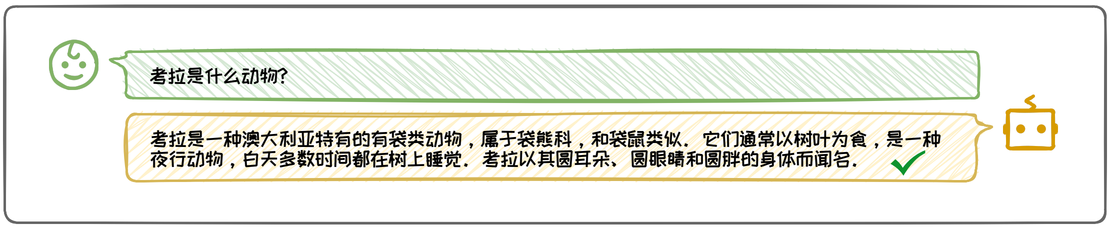
The method of determining internal knowledge through questioning suffers from low reliability of model responses. We can turn to observing the more reliable **training data**. The internal knowledge possessed by a large language model comes from its training data. Therefore, if the model's training data does not contain relevant information about the question at hand, then naturally, the model has not acquired the corresponding knowledge.

In addition, analogous to human learning patterns, we usually have a better grasp of common or repeated knowledge and a weaker grasp of low-frequency knowledge. Therefore, for knowledge that occurs very infrequently in the training data, the degree of learning of them by the model may also be low, i.e., **The frequency of occurrence of knowledge in the training data is positively correlated with the degree of memorization of that knowledge by the model** [22]. Therefore, we can judge whether the model has mastered the corresponding knowledge by determining whether the training data contains the corresponding knowledge. However, this approach has some limitations. First of all, since the size of training data for large language models has reached trillions of levels, this means of statistics is very time-consuming. In addition, for many commercial big language models, such as ChatGPT, GPT4, etc., their training data are not publicly available, in which case this scheme cannot be implemented.

(3) Construct pseudo training data statistics

When the training data is not available, we can design pseudo-training statistics to fit the correlation of the training data. For example, the popularity of an entity can be used as a pseudo-training data statistic because the model does not have enough knowledge of low-frequency occurrences in the training data, but has better knowledge of more "popular" (high-frequency) knowledge. Representative work [33] utilizes **Wikipedia page views as a measure of an entity's popularity, where a higher number of views indicates that the entity is more popular and more likely to be remembered by a large language model**. Figure 6.20 and Figure 6.21 show examples of popular and unpopular knowledge, respectively. From these two examples, it can be noticed that for the popular knowledge **Koala**, the model is able to give the correct answer directly, while for the unpopular knowledge **Number of genes in Koala**, the model gives the wrong answer (the

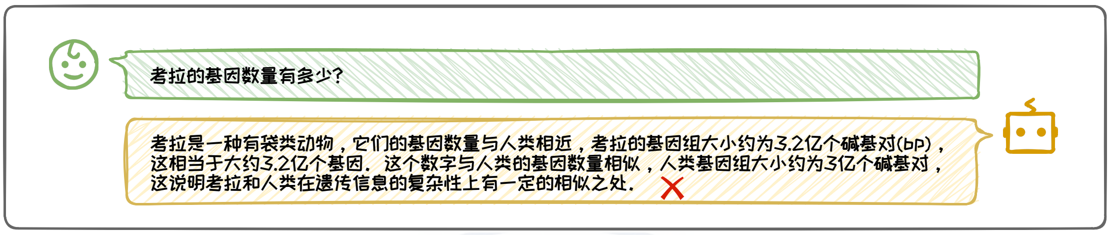
ZHEJIANG UNIVERSITY 1897



考拉是什么动物?

考拉是一种澳大利亚特有的有袋类动物，属于袋熊科，和袋鼠类似. 它们通常以树叶为食，是一种夜行动物，白天多数时间都在树上睡觉. 考拉以其圆耳朵、圆眼睛和圆胖的身体而闻名. ✓

Figure 6.20: Example of modeled responses to popular knowledge.



考拉的基因数量有多少?

考拉是一种有袋类动物，它们的基因数量与人类相近，考拉的基因组大小约为3.2亿个碱基对(bp)，这相当于大约3.2亿个基因. 这个数字与人类的基因数量相似，人类基因组大小约为3亿个碱基对，这说明考拉和人类在遗传信息的复杂性上有一定的相似之处. ✗

Figure 6.21: Example of a model's response to unfavorable knowledge.

correct answer should be 26,558). It can be seen that the popularity of an entity does reflect the internal knowledge of the model to a certain extent, and thus a popularity threshold can be set to discern whether the model has the corresponding internal knowledge. However, this definition of popularity is dependent on the data formation time and may not accurately fit the distribution of the training data, thus there are some limitations.

2. internal observation method

In order to gain further insight into the internal knowledge of a large language model, where the model parameters are accessible, the knowledge acquisition can be more accurately assessed by observing the hidden states within the model. It can be analogous to the process of human lie detection, where scientists analyze brain waves, pulse, blood pressure, and other changes in the internal states of human beings to infer brain activity and cognitive states, and thus determine whether they are lying or hiding certain information. Similarly, for big language models, the internal knowledge level can be assessed by analyzing the hidden state changes in each layer of the model during generation, such as the output of the attention module, the output of the multilayer perceptron (MLP) layer and the activation value changes. This is because the big language model is modeling and predicting the input sequence when generating text, and the changes in the internal state of the model reflect the certainty of the model's understanding of the current context and the next prediction. If the model exhibits **higher internal uncertainty**, such as a more dispersed distribution of attention, large changes in activation values, etc., it may lack sufficient understanding of the current context to make confident predictions.

Since **the internal knowledge retrieval of the model mainly occurs in the feed-forward network of the middle layer** [36], the middle layer of the model exhibits different dynamics when dealing with different problems that contain or do not contain internal knowledge. Based on this property, we can train classifiers for discrimination, a method known as probing. For example, Liang et al [30]designed probe experiments for three types of internal hidden states, namely Attention Output, MLP Output, and Hidden States, as shown in Figure 6.22 shown. For each input problem, the researcher utilizes a trained probe, i.e., a linear classifier, to predict whether the problem is "known" (i.e., the model has the relevant knowledge) or "unknown" (i.e., the model lacks the relevant knowledge) based on the internal representation corresponding to the problem.). The results show that different large language models are able to achieve high classification accuracy when utilizing the internal representation of the middle layer for classification. This indicates that the internal hidden state of the middle layer can effectively reflect the model's understanding of the problem and relevant knowledge reserves.

However, there are limitations to this internal state-dependent detection approach, which is not applicable to black-box language models, since we do not have direct access to their internal hidden states. Also, the inputs to the model need to be carefully

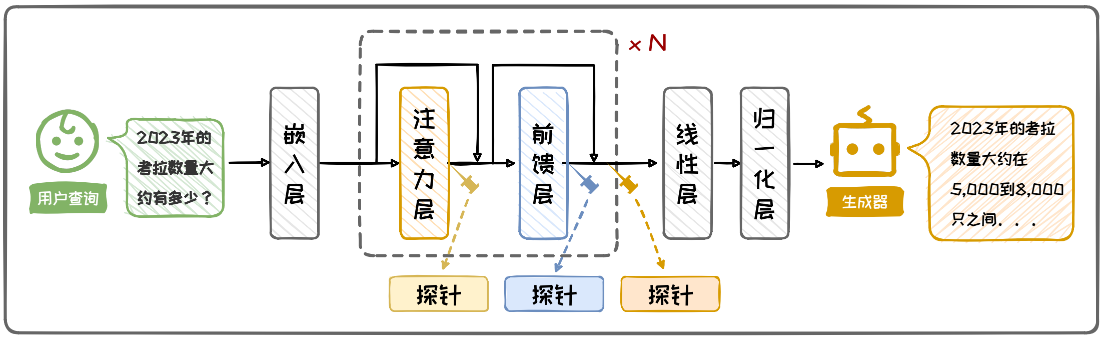

用户查询
2023年的考拉数量大约有多少？
嵌入层
注意力层
前馈层
× N
线性层
归一化层
生成器
2023年的考拉数量大约在5,000到8,000只之间。。。
探针
探针
探针

Figure 6.22: Model internal state probe.

designed, as the uncertainty that the model sometimes exhibits may not stem from a lack of knowledge about the problem, but rather from the inherent ambiguity or ambiguity of the problem itself. Overall, the work on assessing internal knowledge based on internal states is still in the preliminary stage of exploration, and the design of specific programs needs to be further improved, such as how to construct the dataset for modeling “known” and “unknown” problems, how to quantify the uncertainty of internal states, how to quantify the uncertainty of different internal representations, and how to quantify the uncertainty of internal states. For example, how to construct the dataset for the “known” and “unknown” problems of the model, how to quantify the uncertainty of the internal state, how to compare different internal representations, and how to design the internal state detection scheme. We need to conduct research on a wider range of datasets and more diverse model architectures to validate the effectiveness and generalizability of this approach. However, this is a new path full of potential, which is expected to provide new perspectives and methods for us to gain deeper knowledge of large language models from an internal perspective.

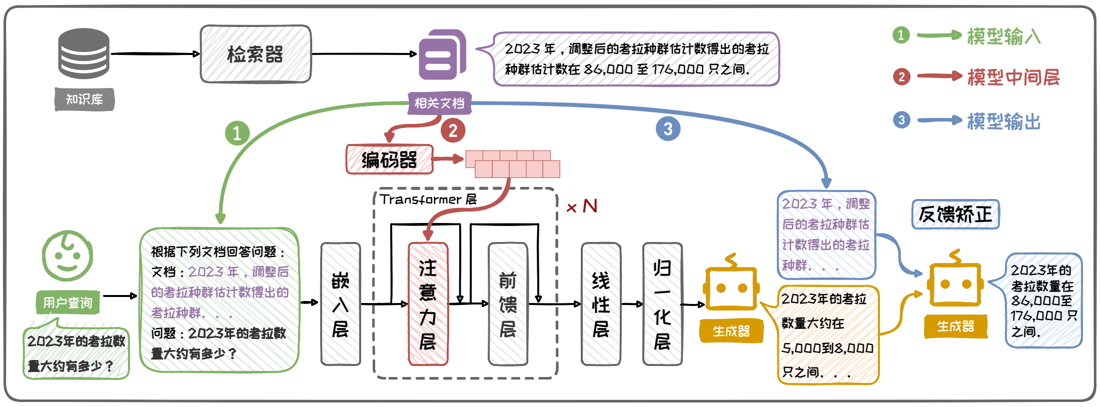

知识库
检索器
相关文档
2023 年，调整后的考拉种群估计数得出的考拉种群估计数在 86,000 至 176,000 只之间.
① 模型输入
② 模型中间层
③ 模型输出
编码器
Transformer 层
× N
用户查询
2023年的考拉数量大约有多少？
根据下列文档回答问题：
文档：2023 年，调整后的考拉种群估计数得出的考拉种群. . .
问题：2023年的考拉数量大约有多少？
嵌入层
注意力层
前馈层
线性层
归一化层
生成器
2023 年，调整后的考拉种群估计数得出的考拉种群. . .
2023年的考拉数量大约在5,000到8,000只之间. . .
反馈矫正
生成器
2023年的考拉数量在86,000至176,000 只之间.

Figure 6.23: Schematic diagram of enhanced implementation location.

6.4.2 where to enhance

After determining that the big language model needs external knowledge, we need to consider where to utilize the retrieved external knowledge, i.e., the problem of **where to enhance**. Thanks to the contextual learning capability, the scalability of the attention mechanism, and the autoregressive generation capability of the big language model, knowledge fusion operations can be performed on its input side, middle layer, and output side. At the input side, the problem and the retrieved external knowledge can be spliced in **Prompt** and then input to the big language model; at the middle layer, the external knowledge can be directly encoded into the hidden state of the model by using **cross-attention**; and at the output side, the external knowledge can be utilized for **post-correction** of the generated text. Figure 6.23 shows the implementation of the above three augmentation positions through a column. Each of the three enhancement positions has different advantages and disadvantages and is suitable for different scenarios. This subsection will introduce them one by one.

(1) Enhancement at the input

The enhancement at the input side method directly splices the retrieved external knowledge text with the user query into Prompt and then inputs it to the big language model. Its the current mainstream enhancement method. The focus of this approach is on the Prompt design and the ordering of the retrieved external knowledge. Good Prompt design and external knowledge ordering can make the model better understand and utilize the external knowledge. In the process of Prompt design, Prompt techniques such as CoT [54] can be utilized, and the specific methods can be found in Chapter 3 of this book on Prompt engineering.

Methods of enhancement at the input **intuitive and easy to implement**. The model can extract the required information directly from the context of the input without complex processing or conversion. However, when the retrieved text is too long, it may result in an input sequence that is too long, even beyond the model's maximum sequence length limit. This brings challenges to the contextual understanding of the model, and also increases the computational cost of the model's reasoning and its computational burden. This approach requires a high level of long text processing capability and contextual understanding of the large language model.

(2) Augmentation at the middle layer

The enhancement at the intermediate layer augmentation approach utilizes the flexibility of the attention mechanism by first converting the retrieved external knowledge into **vector representations**, and then inserting these vectors into the hidden states of the model through cross-attention fusion. The Retro [6] method, introduced in Section 6.2, typifies the use of this approach. This approach is able to **influence the internal representation of the model more deeply** and may help the model to better understand and utilize external knowledge. At the same time, since vector representations are usually more compact than raw text, this approach can **reduce the dependence on the length of the model input**.

However, this approach requires complex design and adaptation of the model's structure and cannot be applied to black-box models.

(3) Enhancement at the output

Enhancement at the output is a method of **post-processing** that utilizes retrieved external knowledge to **calibrate** the text generated by a large language model. In such methods, the model first generates an initial answer without external knowledge, and then uses the retrieved external knowledge to verify or calibrate this answer. The calibration process corrects the output based on the knowledge consistency between the generated text and the retrieved text. The correction can be accomplished by providing the preliminary answer with the retrieved information to the big model, allowing the big model to check and adjust the generated answer. For example, the REFEED framework [59] proposed by Yu et al. is typical of such approaches. The advantage of this approach is that it ensures that **generated text is consistent with external knowledge** and improves the accuracy and reliability of answers. However, its effectiveness depends heavily on the quality and relevance of the retrieved external knowledge. If the retrieved documents are inaccurate or irrelevant, it will lead to incorrect calibration results.

In summary, the above three enhancement methods have their own advantages and disadvantages, and in practical applications, we can flexibly choose different programs according to specific scenarios and needs. Moreover, since the above three schemes are independent of each other, they can also be used in combination to achieve better enhancement effects.

6.4.3 multiple augmentation

In practice, users' questions about large language models may be complex or fuzzy. Complex questions often involve multiple knowledge points and require multi-hop comprehension, while fuzzy questions often have an unknown scope of reference, making it difficult to understand the meaning of the question in one go. For complex and fuzzy problems, it is difficult to ensure the correct generation by one retrieval enhancement, and multiple iterations of retrieval enhancement are inevitable. When dealing with complex problems, the scheme of **decompositional enhancement** is often used. This scheme decomposes the complex problem into multiple sub-problems, and iterative retrieval enhancement is performed among the sub-problems to finally obtain the correct answer. When dealing with fuzzy problems, the scheme of **progressive enhancement** is often used. This scheme refines the question's continuously, and then retrieval enhancement is performed on the refined questions separately, aiming to give a comprehensive answer to cover the answer needed by the user. Both schemes are described in this subsection.

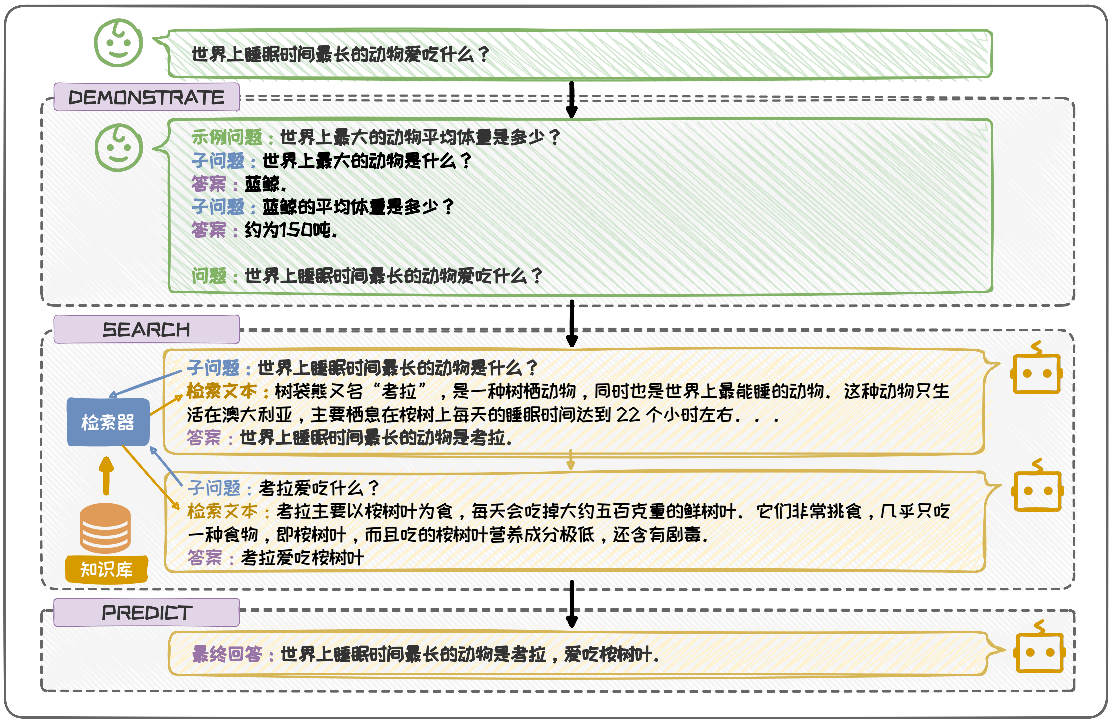
1. Decompositional Enhancement

Complex problems often contain multiple points of knowledge. For example, in the question "What does the world's longest sleeping animal like to eat?" This question contains the questions "What is the world's longest-sleeping animal?" and "What does this animal like to eat?" Two knowledge points. Answering complex questions requires multiple jumps in understanding. Therefore, in retrieval augmentation of complex questions, we usually cannot get a satisfactory answer with only one retrieval augmentation. In this case, the model can decompose the multi-hop question into one sub-problem, and then iteratively perform retrieval enhancement among the sub-problems to finally arrive at the correct conclusion, i.e., decomposed enhancement.

DEMONSTRATE-SEARCH-PREDICT(DSP) [25] is a representative decomposed enhancement framework. The framework mainly contains the following three modules: (1) **DEMONSTRATE**module, which decomposes the complex problem into subproblems by means of context learning; (2) **SEARCH**module, which performs iterative retrieval of enhancement on the subproblems to provide comprehensive external knowledge for the final decision; (3) **PREDICT** module, which generates the final answer based on the synthesized external knowledge provided by SEARCH.

As shown in Figure 6.24, “What does the world’s longest sleeping animal love to eat?” As an example, the DSP process is described. First, the DEMONSTRATE module looks for examples of similar problems in the training set as a way of demonstrating how the problem can be decomposed. These examples will guide the large language model in generating precise queries for the sub-problems. For example, the first sub-question is: “What is the longest sleeping animal in the world?” . The SEARCH module will then utilize the first sub-question to retrieve relevant information to determine that the longest-sleeping animal is the koala, and form a second sub-question based on this new information, “What do koalas love to eat?” , which then performs the search again to find relevant text describing the eating habits of koalas. Eventually, in the PREDICT module, the Big Language Model will synthesize all the information to arrive at the final answer, which is, “The longest sleeping animal in the world is the koala, which loves to eat eucalyptus leaves.”

Decompositional enhancement reduces the difficulty of a single retrieval enhancement by reducing the complex problem to zero. Its performance depends heavily on the quality of the subproblem decomposition. Complex problems in different domains may



世界上睡眠时间最长的动物爱吃什么？

示例问题：世界上最大的动物平均体重是多少？
子问题：世界上最大的动物是什么？
答案：蓝鲸.
子问题：蓝鲸的平均体重是多少？
答案：约为150吨.

问题：世界上睡眠时间最长的动物爱吃什么？

子问题：世界上睡眠时间最长的动物是什么？
检索文本：树袋熊又名“考拉”，是一种树栖动物，同时也是世界上最能睡的动物. 这种动物只生活在澳大利亚，主要栖息在桉树上每天的睡眠时间达到 22 个小时左右. . .
答案：世界上睡眠时间最长的动物是考拉.

子问题：考拉爱吃什么？
检索文本：考拉主要以桉树叶为食，每天会吃掉大约五百克重的鲜树叶. 它们非常挑食，几乎只吃一种食物，即桉树叶，而且吃的桉树叶营养成分极低，还含有剧毒.
答案：考拉爱吃桉树叶

最终回答：世界上睡眠时间最长的动物是考拉，爱吃桉树叶.

Figure 6.24: DSP flow schematic.

have different decomposition paradigms, so it is often necessary to design the problem decomposition scheme according to the specific task.

2. progressive enhancement

In fuzzy questions, the subject of the question is usually referred to in an unclear way, which can easily lead to ambiguity. For example, in the question “What do national treasure animals like to eat?” For example, in the question “What do national treasure animals like to eat?”, since different countries have different national treasure animals, and some countries have more than one national treasure animal, we can’t be sure which one of them we are asking about. When dealing with such a fuzzy question, we can progressively disassemble and refine the question, and then retrieve the refined question and use the retrieved information to augment the big model. This progressive enhancement method can help the big language model to have more comprehensive information.

TREE OF CLARIFICATIONS (TOC) [26] is a representative framework for progressive enhancement. The framework guides the large language model through recursive retrieval to explore multiple clarification paths for a given fuzzy question in **tree structure**. Figure 6.25 illustrates the entire workflow of the TOC framework, which we illustrate with the question "**What do national treasures animals love to eat?**" We take the question "**What do China's national treasure animals love to eat?**" as an example for illustration.The TOC framework first starts the first round of search, and generates a series of specific refined questions based on the retrieved relevant documents and the original question, e.g. "**What do China's national treasure animals love to eat?**", "**What does Australia's national animal love to eat?**". In this process, the framework will prune the refinement questions based on their relevance and knowledge consistency with the original question. Subsequently, for each refined question, the framework independently conducts an in-depth search and further refines the question, e.g., "**What does Australia's national animal, the koala, love to eat?**". Eventually, we are able to construct multiple complete knowledge paths, with the end of each path (leaf node) representing a different but valid answer to the original question. By integrating all valid leaf nodes, we are able to come up with a precise and comprehensive long answer, for example for this question we get, "**China's national animal, the panda, loves bamboo; Australia's national animal, the koala, loves eucalyptus leaves; Australia's national animal, the kangaroo, loves weeds and shrubs... ...**"

This incremental retrieval method can significantly improve the generation of fuzzy problems by large language models. However, it requires multiple rounds of retrieval and generation of multiple answer paths, and the **computation** of the whole system grows exponentially with the problem complexity. In addition, multiple rounds of retrieval and

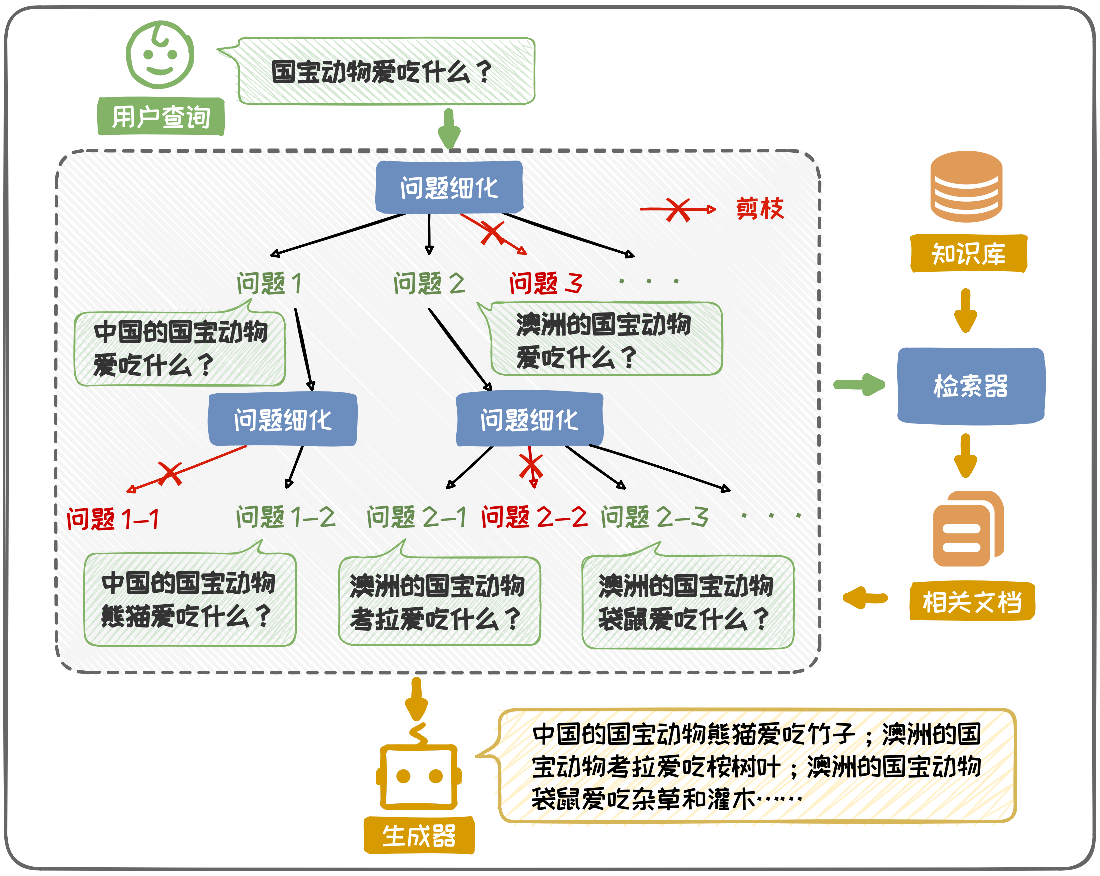


Figure 6.25: TOC framework process schematic.

ZHEJIANG UNIVERSITY 1897

generation not only bring significant **time delay**, but multiple inference paths also bring **instability** to the inference, which is prone to trigger errors.

6.4.4 cost reduction

Retrieved external knowledge usually contains a large amount of raw text. When inputting it to the big language model via Prompt, it will substantially increase the number of input Token, which increases the reasoning computation cost of the big language model. This problem can be solved from two perspectives: **removing redundant text** and **reusing computation results**. In this subsection, these two perspectives will be introduced separately.

1. Remove redundant text

In RAG, the retrieved raw text usually contains a large amount of redundant information that is not useful for enhancement generation. This redundant information not only increases the length of the input Token, but also has the potential to interfere with the larger model, leading to the generation of incorrect answers. Methods for removing redundant text are performed by filtering the words and phrases of the retrieved raw text and selecting the parts that are useful for augmentation generation. The methods for removing redundant text are categorized into three main types: token level methods, subtext level methods and full text level methods. These methods use different mechanisms to filter and optimize the retrieved raw text to reduce unhelpful information and ensure the relevance and accuracy of the generated content. These three types of methods are described below.

The Token level approach eliminates unnecessary Token from the text by evaluating the Token. Perplexity is an important metric for determining the importance of a Token. Intuitively, if a Token has a low perplexity, it means that the model has a high probability of predicting the Token, indicating that the Token is easy to predict and more general, and therefore it carries less amount of new information, which may be redundant; on the contrary, if a Token has a high perplexity, it indicates that the Token carries more amount of information. The **LongLLMLingua** [20] framework utilizes a small model to evaluate the Token's perplexity and then removes redundant text based on the perplexity. It first performs problem-aware **coarse-grained compression**, i.e., it evaluates the importance of a document by calculating the average of the perplexity degrees of all the Token in the document under the given problem conditions, with higher perplexity degrees indicating more informative documents. Subsequently, problem-aware **fine-grained compression** is performed, i.e., the perplexity of each Token in a document is further calculated and the Token with low perplexity is removed.In addition, **document reordering mechanism**, **dynamic compression ratio**, and **sub-sequence restoration** mechanisms are introduced in the thesis to ensure that the importance of documents and the important information in them are effectively utilized. important information in them are effectively utilized.

The subtext level approach removes unnecessary subtexts in pieces by scoring them. **FIT-RAG** [35] is a representative method among the subtext level methods. In this method, a pre-trained two-label sub-document scorer is used to evaluate the usefulness of the document in two dimensions: first, **factuality**, i.e., whether the document contains information about the answer to the query; and second, **model's preference**, i.e., whether the document is easy to be understood by the model. For the retrieved documents, they are first split into multiple sub-documents using a sliding window, and then these sub-documents are scored separately using a two-label sub-document scorer. Finally, the lower scoring sub-documents are removed, thus effectively removing redundant text.

Full text level methods extract important information directly from the entire document to remove redundant information. The classical method **PRCA** [57] extracts important information from the text by training an information extractor capable of refining the key content. The method is divided into two phases, the context extraction phase and the reward-driven phase. In the **context extraction phase**, with the goal of minimizing the difference between the compressed text and the original input document, the information extractor is trained with supervised learning to learn how to refine the input document into information-rich compressed text. **in the reward-driven phase**, the large language model is used as a reward model, which optimizes the information extractor through reinforcement learning based on the similarity between the answers generated from the com-

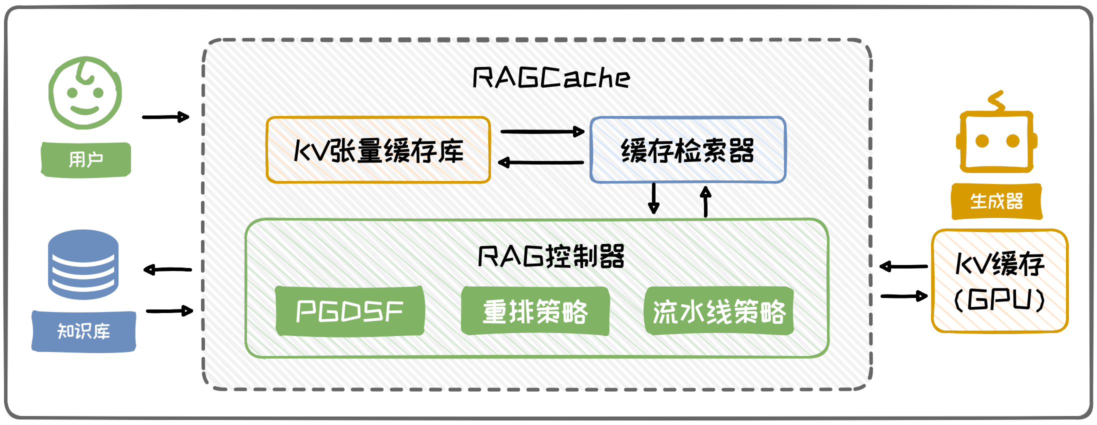


Figure 6.26: RAGCache framework flow diagram.

pressed text and the real answers as a reward signal. The resulting information extractor can directly transform the input document into compressed text, removing redundant text end-to-end.

2. reuse calculation results

In addition to sifting out redundant information, we can also reuse the intermediate results necessary for computation to optimize RAG efficiency. In the autoregressive process of inference performed by the large language model, each Token needs to use the results of Key and Value involved in the previous Token attention module. In order to avoid recalculating the previous Key and Value results for each Token, we can cache the previously calculated Key and Value results (i.e., KV-cache), and call the relevant results directly from the KV-cache when needed, thus avoiding repeated calculations. However, as the length of the input text increases, the GPU memory usage of the KV-cache will increase dramatically, even far exceeding the size of the memory occupied by the model parameters [46]. However, the input to the RAG usually contains retrieved text, resulting in very long input text, leading to costly KV-cache storage in the RAG.

However, in RAG, different user queries often retrieve the same text, and the number of common queries is usually limited. Therefore, we can reuse the KV-cache of common duplicate texts to avoid computing them for each query to reduce the storage cost. Based on this, **RAGCache** [21]designed a multi-level dynamic caching mechanism specific to the RAG system, as shown in Fig. 6.26.The RAGCache system consists of three core components: the **KV tensor cache repository**, the **cache retriever** and **RAG controller**. Among them, **KV tensor cache** adopts a tree structure to cache the computed document KV tensor, where each tree node represents a document; **cache retriever** is responsible for quickly finding out whether there is a required cache node in the cache; and **RAG controller**, as the policy center of the system, is responsible for formulating the core caching policy. Specifically, the RAG controller first adopts a **Prefix-aware Greedy Dual Scale Frequency (PGDSF)** replacement strategy. This strategy takes into account the access frequency, size, access cost, and recent access time of the document nodes so that frequently used documents can be retrieved quickly. Subsequently, the **reordering policy** reduces the need for recomputation by adjusting the processing order of requests to prioritize those that can make more use of the cached data. Such requests enjoy a higher priority in the queue, thus optimizing resource usage. Finally, the system also introduces the **dynamic speculative pipelining strategy**, which effectively reduces the end-to-end delay by parallelizing the steps of KV tensor retrieval and model inference.

In summary, by combining the above mentioned **input Prompt compression** with **KV-cache mechanism**, the RAG framework can significantly improve its efficiency while maintaining high performance. This not only helps to deploy the model in resource-constrained environments, but also improves the responsiveness of the model in real-world applications.

6.5 Practice and Application

By introducing external knowledge, RAG can effectively alleviate the phantom phenomenon of large language models and expand the knowledge boundary of large language models. Its superior performance has attracted widespread attention, becoming a hot cutting-edge technology and landing in many application scenarios. In this section, we will explore the method of building a simple RAG system, and two types of typical applications of RAG.

6.5.1 Building a Simple RAG System

In order to help developers build RAG systems efficiently and easily, there are many mature open source frameworks to choose from, the most representative of which are LangChain [8]and LlamaIndex [9]. These frameworks provide a comprehensive set of tools and interfaces that allow developers to easily integrate RAG systems into their applications, such as chatbots, agents, and so on. Next, we will first give a brief overview of the features of these two frameworks and their core functionality, and then explain how to use LangChain to build a simple RAG system.

1. LangChain and LlamaIndex

(1) LangChain

LangChain aims to simplify the entire process of developing applications using the Big Language Model. It provides a series of modular components to help developers de-

---

[8] https://www.langchain.com

[9] https://www. llamaindex.ai

ploy applications based on the big language model, including the construction of the RAG framework.LangChain contains six main modules:**Model IO, Retrieval, Chains, Memory, Agents and Callbacks**. The Model IO module contains interfaces to various large models and Prompt design components, while the Retrieval module contains the core components needed to build a RAG system, including **Document Loading, Text Segmentation, Vector Construction, Index Generation, and Vector Retrieval**, etc., and also provides interfaces to unstructured databases. While the Chains module can link the modules together to execute them one by one, the Memory module can store the data during the dialog. In addition, the Agent module can utilize the large language model to automatically decide which operations to perform, and the Callback module can help the developer to intervene and monitor each stage. Overall, **LangChain provides a more comprehensive module support** to help developers build their own RAG application framework easily and conveniently.

(2) LlamaIndex

Compared to LangChain, LlamaIndex is more focused on the part of **data indexing and retrieval**. This feature enables developers to **rapidly build efficient retrieval systems**.LlamaIndex has the ability to extract data from a wide range of data sources (e.g., APIs, PDF files, SQL databases, etc.), and provides a series of highly efficient tools for vectorizing and indexing these data. In terms of data querying, LlamaIndex also provides an efficient retrieval mechanism. In addition, after obtaining the contextual information, LlamaIndex also supports filtering, reordering and other fine-grained operations on this information. It is worth mentioning that the LlamaIndex framework can also be combined with the LangChain framework to realize more diversified functions. Overall, **LlamaIndex focuses on indexing and retrieval, and performs more prominently in query effi-**

ciency, and is very suitable for building more efficient RAG systems in large data volume scenarios.

2. Building a Simple RAG System Based on LangChain

This subsection will take the LangChain framework as an example and refer to its official documentation [10] to demonstrate how to quickly build a simple RAG system.

1) Installation and Configuration: First of all, you need to install LangChain framework and its dependencies in your environment.

```
# 安装LangChain框架及其依赖项
!pip install langchain langchain_community langchain_chroma
```

2) Data Preparation and Index Construction: Next, we need to prepare the data and construct the index, LangChain's DocumentLoaders provide a rich variety of document loaders, for example, we can use WebBaseLoader to load content from web pages and parse it into text.

```
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()
```

After loading, the loaded document may be too long for the model's context window and needs to be split into appropriate sizes.LangChain provides a TextSplitter component to realize document splitting.

```
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
```

---

[10] https://python.langchain.com/v0.2/docs/tutorials/rag/

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap
    =200)
splits = text_splitter.split_documents(docs)
```

Next we need to index the segmented text blocks for subsequent retrieval. Here we can call the Chroma vector storage module and the OpenAIEmbeddings model to store and encode the document.

```
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 system construction: after constructing a good knowledge source, we can next start to construct a basic RAG system. The system consists of two parts, namely, the retriever and the generator, and the specific workflow is as follows: for the question input by the user, the retriever first searches for the documents related to the question, then passes the retrieved documents together with the initial question to the generator, i.e., the large language model, and finally returns the answer generated by the model to the user.

First the construction of the Retriever is carried out, here we can construct a Retriever object based on VectorStoreRetriever, and utilize vector similarity for retrieval.

```
# 创建检索器
retriever = vectorstore.as_retriever()
```

The next step is the construction of the generator part, here we can use the ChatOpenAI system model as the generator. In this step, it is necessary to set the OpenAI API key and specify the specific model model to be used. For example, we can choose to use the gpt-3.5-turbo-0125 model.

```
import os
os.environ["OPENAI_API_KEY"] = 'xxx'
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-3.5-turbo-0125")
```

This is followed by inputting the Prompt settings. LangChain's Prompt Hub provides a variety of preset Prompt templates for different tasks and scenarios. Here we choose a Prompt that is applicable to the RAG task.

```
from langchain import hub
# 设置提示模板
prompt = hub.pull("rlm/rag-prompt")
```

Finally we need to integrate retrieval and generation, where a chain can be easily and quickly constructed using the LangChain Execution Language (LCEL) to combine the retrieved documents, the constructed input Prompt, and the output of the model.

```
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)
```

With the above steps, we can quickly and easily build a basic RAG system using LangChain, which provides a series of powerful tools and components that make building and integrating the retrieval and generation process simple and efficient.

6.5.2 Typical Applications of RAG

In this subsection, two typical application cases of the RAG system will be introduced: **(1) Intelligent Body (Agent)**; **(2) Pendant Domain Multimodal Model Enhancement**.

1. intelligent body (Agent)

RAG plays an important role in Agent systems. In the process of proactive planning and invoking various tools by the Agent system, it needs to retrieve and integrate diverse information resources in order to more accurately satisfy the user's needs.RAG helps the Agent to show better performance in dealing with complex problems by providing the required information support. Figure 6.27 shows a classic Agent framework [50], which consists of four main parts: **Profile**, **Memory**, **Planning**, and **Action**. textbfPlanning and **Action**. The Memory, Planning and Action modules all incorporate RAG technology to enhance overall performance.

Specifically, **(1) Configuration Module** defines the Agent's role by setting **Basic**

Information, which can include basic attributes such as the Agent's age, gender, and occupation, as well as information reflecting his/her personality and social relationships; **(2) Memory Module** stores **Knowledge** and **Memory** learned from the environment. and **historical information**, which supports operations such as memory retrieval, memory updating and memory reflection, allowing the Agent to continuously acquire, accumulate and utilize knowledge. In this module, **RAG assists memory reading and updating by retrieving relevant information**; **(3) Planning Module** empowers the Agent to break down a complex task into simple subtasks and continuously adjust it based on memory and action feedback. **RAG in this module helps the Agent to plan the task more rationally and effectively by providing relevant information**; **(4) Action Module** is responsible for transforming the Agent's plan into concrete actions, including web page retrieval, tool invocation, and multimodal outputs, etc., which are able to have an impact on the environment or the Agent's own state, or to trigger a new action chain The In this module, **RAG assists the Agent's decision making and action execution by retrieving relevant information**. Through this modularized and integrated approach to RAG technology, the Agent can respond more efficiently and intelligently to the user's needs, and demonstrate better performance.

Let's take "**I'm in Australia now, I want to hug a koala, what should I do?**" This specific problem is used as an example to show the flow of how the Agent handles the problem. In this example, the user presents a clear need. the Agent will go through the following steps to accomplish the task:

- **Role Configuration**: first, Agent uses **Configuration Module** for initialization, sets its role as a professional travel consultant, and makes clear that its task goal is to assist the user in realizing his desire to have a close encounter with a koala;

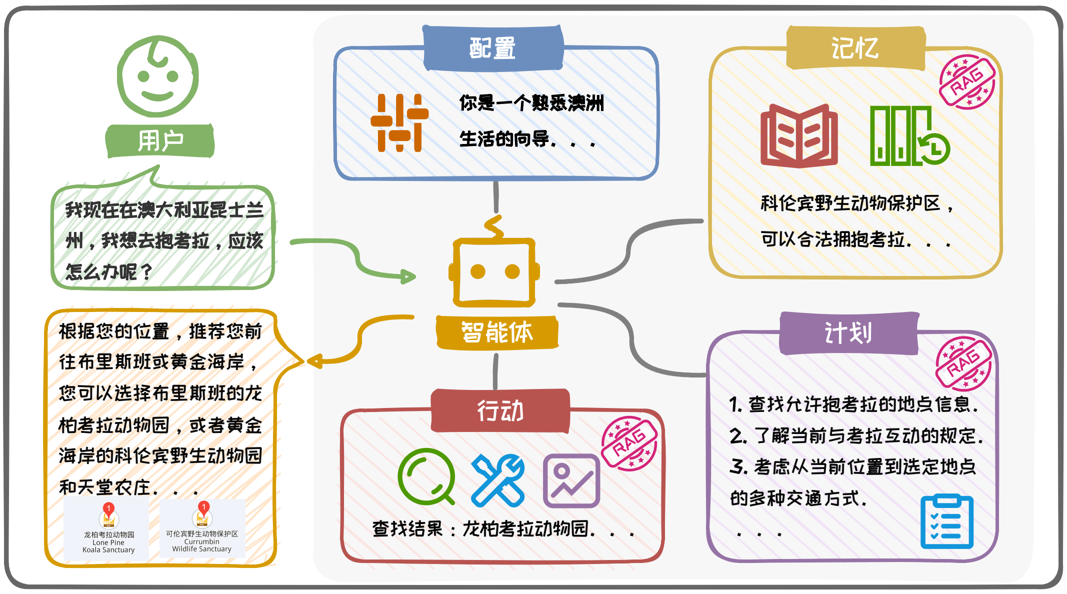

用户
我现在在澳大利亚昆士兰州，我想去抱考拉，应该怎么办呢？
根据您的位置，推荐您前往布里斯班或黄金海岸，您可以选择布里斯班的龙柏考拉动物园，或者黄金海岸的科伦宾野生动物园和天堂农庄. . .
龙柏考拉动物园
Lone Pine Koala Sanctuary
可伦宾野生动物保护区
Currumbin Wildlife Sanctuary
配置
你是一个熟悉澳洲生活的向导. . .
智能体
行动
RAG
查找结果：龙柏考拉动物园. . .
记忆
RAG
科伦宾野生动物保护区，可以合法拥抱考拉. . .
计划
RAG
1. 查找允许抱考拉的地点信息.
2. 了解当前与考拉互动的规定.
3. 考虑从当前位置到选定地点的多种交通方式.
. . .

Figure 6.27: Agent framework process schematic.

- **Task Planning**: In response to the user's needs, **Planning Module**plans how to help the user to realize the desire to “hug a koala”. Examples include identifying locations where koalas exist near the current location, understanding local laws about interacting with wildlife, and ways to get to the destination. At this stage, **RAG** makes planning more precise by providing information on relevant tourist attractions and laws and regulations;
- **Information Retrieval**: **Memory Module**first retrieves relevant information already in the memory. If the memory does not contain all the necessary information, **Action Module** will activate and invoke tools to retrieve it from external knowledge sources (e.g. tourism websites, etc.). In this process, **RAG** ensures that the most relevant information for the koala interaction is retrieved with maximum efficiency;
- **Information Integration and Decision Making**: **Planning Module** will use the information retrieved above to develop the best course of action. This may involve recommending specific locations, providing travel advice, and emphasizing safety

instructions. **RAG** integrates the information in this session to assist the Agent in making informed decisions;

- **Information Output**: finally, **Action Module** outputs the course of action determined by the Agent to the user in an easy-to-understand manner, including a detailed route plan as well as descriptions, images and other information about relevant locations.

Through this example, we can clearly see how the various modules in the Agent framework work together to accomplish a specific prediction task, where the role of the RAG is to quickly retrieve information and integrate knowledge to provide a comprehensive and accurate solution for the user.

2. Multimodal Vertical Domain Applications

In the previous sections, we mainly focused on RAG systems in the text domain, but nowadays, RAG also shows a broad application prospect in many vertical domains involving multimodal data. For example, in the medical field, multimodal data is very common, including image data such as X-rays, MRIs, CT scans, and text data such as medical records and physiological monitoring data. These data not only come from a wide range of sources, but also have complex interconnections with each other. Therefore, when dealing with these tasks, the RAG system must have the ability to merge and gain insight into different modal data to ensure that it works accurately and efficiently.

Several RAG applications for multimodal verticals have emerged, such as the multimodal retrieval framework [38] for the medical domain proposed by Ossowski et al. Figure 6.28 illustrates the entire workflow of this framework. The task **Given an X-ray photograph, ask for the possible symptoms corresponding to that photograph** is illustrated here as an example. The framework firstly **extracts the feature representa-**

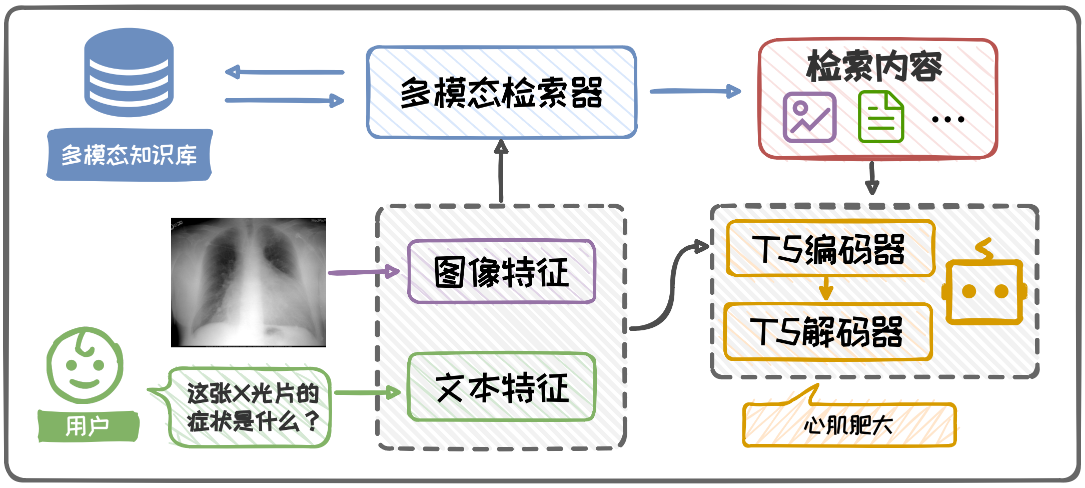


Figure 6.28: Schematic diagram of the multimodal pendant domain RAG framework.

tions of images and texts to deeply understand the content of images and the semantics of the question, providing rich feature vectors for the retrieval module. Subsequently, the framework elaborates a **multimodal retrieval module** that utilizes these feature vectors to perform accurate searches in the medical knowledge base in order to obtain the most relevant information to the input question. This information may include medical cases, symptom descriptions, potential etiologies, and treatment options, which are presented in multiple forms of text and images. Ultimately, the Prompt builder module integrates this information to assist the model in generating accurate responses. In this example, the answer generated by the model is “cardiac hypertrophy” .

In addition to the medical field, RAG has also demonstrated its strong capability in processing specialized information in other verticals, such as finance [8], biology [53], which significantly improves the decision-making efficiency of professionals. In all kinds of tasks, in addition to picture information [10], RAG also has equally outstanding performance in other multimodal scenarios such as audio [7] and video [52]. With the advancement of technology and the abundance of data resources, we expect RAG to play a key role in more vertical fields and more data modalities in the future.

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`.

ZHEJIANG UNIVERSITY 1897