LLM 学习笔记

夏同

2026 年 2 月 9 日

目录

第一章　基础知识

1.1　大模型基础概念

大模型：一般指 1 亿以上参数模型，但标准一直升级，目前已有万亿参数以上的模型。

大语言模型 (Large Language Model, LLM)：针对语言的大模型。

参数规模：**175B、60B、540B 等**，这些一般指参数的个数，B 是 Billion/十亿的意思，175B 是 1750 亿参数，这是 ChatGPT 大约的参数规模。

1.2　单双向注意力

1.2.1　核心概念："阅读"和"写作"

理解单双向注意力是掌握大模型架构差异的关键。我们可以用两个生动的比喻来理解：

双向注意力：像阅读侦探小说: 想象你在阅读一本悬疑小说。为了理解复杂的情节，你会**随意地前后翻看**。当看到最后一章揭示凶手时，你可能会翻回前面的章节，查看某个角色的不在场证明是否有漏洞。这就是" 双向" 的精髓——任何一个部分的信息都可以参考全文的任何其他部分，获得全局的、最充分的理解。

单向注意力：像写作小说续集: 现在你要为这本小说写续集。你只能**从左到右一个一个词地写**。在写下" 侦探" 这个词时，你只能基于前面已经写好的" 突然，门开了，走进来一位…" 来构思，你**不能提前知道或使用**后面将要写出的" 掏出了手枪" 这个词。这就是" 单向" 或" 因果" 的本质——每个新词只能基于它之前的所有词来生成。

1.2.2　技术深度解析

双向注意力机制:

- **目标**：深度**理解**和**编码**输入信息
- **工作原理**：模型同时处理整个句子的所有词。当理解某个词（如代词" 它"）时，可以**同时关注**该词左右两侧的所有上下文，从而准确判断指代关系
- **优势**：文本理解能力极强，能把握复杂语义关系和指代消解
- **局限**：不适合直接用于生成任务，否则会" 作弊"（提前看到答案）
- **典型代表**：BERT 模型，主要用于文本分类、情感分析等理解型任务

单向注意力机制:

- **目标**：序列**生成**
- **工作原理**：模型以自回归方式工作，每次只能基于当前和之前的词预测下一个词，严格遵循因果律
- **优势**：天然适合文本生成任务，训练目标与实际应用完全一致，Zero-shot 能力强，容易涌现新能力
- **局限**：在纯理解任务上可能不如双向模型深入
- **典型代表**：GPT 系列、LLaMA 系列

1.2.3　单双向注意力对比总结

表 1.1: 单双向注意力机制对比

特性	双向注意力	单向注意力（因果注意力）
核心目标	理解与分析	生成与创作
信息流动	全局、无方向限制	从左到右、严格因果
形象比喻	阅读分析文章	写作口述文章
主要优势	深层语义理解、分类任务强	文本生成、零样本能力、涌现能力
主要局限	不直接适用于生成	理解任务可能缺少全局上下文
典型架构	Encoder（编码器）	Decoder（解码器）
代表模型	BERT	GPT、LLaMA 系列

1.3　主流开源模型体系

1.3.1　三种主流体系

- **Prefix Decoder 系**
  - 介绍：输入双向注意力，输出单向注意力
  - 代表模型：ChatGLM、ChatGLM2、U-PaLM
- **Causal Decoder 系**
  - 介绍：从左到右的单向注意力
  - 代表模型：LLaMA-7B、LLaMa 衍生物
- **Encoder-Decoder 系**
  - 介绍：输入双向注意力，输出单向注意力
  - 代表模型：T5、Flan-T5、BART y1y2

1.4 三种 Decoder 架构区别

1.4.1 核心区别

主要区别在于 attention mask 不同：

1.4.2 Encoder-Decoder 架构

- 在输入上采用双向注意力，对问题的编码理解更充分
- 适用任务：在偏理解的 NLP 任务上效果好
- 缺点：在长文本生成任务上效果差，训练效率低

1.4.3 Causal Decoder 架构

- 自回归语言模型，预训练和下游应用是完全一致的，严格遵守只有后面的 token 才能看到前面的 token 的规则
- 适用任务：文本生成任务效果好
- 优点：训练效率高，zero-shot 能力更强，具有涌现能力

1.4.4 Prefix Decoder 架构

- 特点：prefix 部分的 token 互相能看到，是 Causal Decoder 和 Encoder-Decoder 的折中
- 缺点：训练效率低

1.5 大模型训练目标

1.5.1 语言模型

根据已有词预测下一个词，训练目标为最大似然函数：

$$\mathcal{L}_{LM}(x) = \sum_{i=1}^{n} \log P(x_i|x_{<i})$$

训练效率：Prefix Decoder < Causal Decoder
Causal Decoder 结构会在所有 token 上计算损失，而 Prefix Decoder 只会在输出上计算损失。

1.5.2 去噪自编码器

随机替换掉一些文本段，训练语言模型去恢复被打乱的文本段。目标函数为：

$$\mathcal{L}_{DAE}(x) = \log P(\tilde{x}|x_{/\tilde{x}})$$

去噪自编码器的实现难度更高。采用去噪自编码器作为训练目标的任务有 GLM-130B、T5。

1.6 涌现能力分析

根据前人分析和论文总结，大致是 2 个猜想：

- 任务的评价指标不够平滑
- 复杂任务 vs 子任务：假设某个任务 T 有 5 个子任务 Sub-T 构成，每个 sub-T 随着模型增长，指标从 40% 提升到 60%，但是最终任务的指标只从 1.1% 提升到了 7%，也就是说宏观上看到了涌现现象，但是子任务效果其实是平滑增长的

1.7 Decoder Only 架构优势

- decoder-only 结构模型在没有任何微调数据的情况下，zero-shot 的表现能力最好
- 而 encoder-decoder 则需要在一定量的标注数据上做 multitask-finetuning 才能够激发最佳性能
- 目前的 Large LM 的训练范式还是在大规模语料上做自监督学习，zero-shot 性能更好的 decoder-only 架构才能更好的利用这些无标注的数据
- 大模型使用 decoder-only 架构除了训练效率和工程实现上的优势外，在理论上因为 Encoder 的双向注意力会存在低秩的问题，这可能会削弱模型的表达能力
- 就生成任务而言，引入双向注意力并无实质的好处
- Encoder-decoder 模型架构之所以能够在某些场景下表现更好，大概是因为它多了一倍参数
- 在同等参数量、同等推理成本下，Decoder-only 架构就是最优的选择

1.8 大模型优缺点分析

1.8.1 优点

- 可以利用大量的无标注数据来训练一个通用的模型，然后再用少量的有标注数据来微调模型，以适应特定的任务。这种预训练和微调的方法可以减少数据标注的成本和时间，提高模型的泛化能力
- 可以利用生成式人工智能技术来产生新颖和有价值的内容，例如图像、文本、音乐等。这种生成能力可以帮助用户在创意、娱乐、教育等领域获得更好的体验和效果
- 可以利用涌现能力 (Emergent Capabilities) 来完成一些之前无法完成或者很难完成的任务，例如数学应用题、常识推理、符号操作等。这种涌现能力可以反映模型的智能水平和推理能力

1.8.2　缺点

- 需要消耗大量的计算资源和存储资源来训练和运行，这会增加经济和环境的负担。据估计，训练一个 GPT-3 模型需要消耗约 30 万美元，并产生约 284 吨二氧化碳排放
- 需要面对数据质量和安全性的问题，例如数据偏见、数据泄露、数据滥用等。这些问题可能会导致模型产生不准确或不道德的输出，并影响用户或社会的利益
- 需要考虑可解释性、可靠性、可持续性等方面的挑战，例如如何理解和控制模型的行为、如何保证模型的正确性和稳定性、如何平衡模型的效益和风险等。这些挑战需要多方面的研究和合作，以确保大模型能够健康地发展

第二章　Layer Normalization 篇

2.1　Layer Norm 基础

2.1.1　Layer Norm 计算公式

Layer Normalization 的计算公式如下：

$$\mu = E(X) = \frac{1}{H}\sum_{i=1}^{H} x_i$$

$$\sigma = \sqrt{Var(x)} = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(x_i - \mu)^2 + \epsilon}$$

$$y = \frac{x - E(x)}{\sqrt{Var(X) + \epsilon}} \cdot \gamma + \beta$$

其中：

- γ: 可训练的再缩放参数
- β: 可训练的再偏移参数

2.2　RMS Norm（均方根 Norm）

2.2.1　RMS Norm 计算公式

RMS Norm 的计算公式如下：

$$RMS(x) = \sqrt{\frac{1}{H}\sum_{i=1}^{H} x_i^2}$$

$$x = \frac{x}{RMS(x)} \cdot \gamma$$

2.2.2　RMS Norm 的特点

- RMS Norm 简化了 Layer Norm，去除掉计算均值进行平移的部分

- 对比 LN，RMS Norm 的计算速度更快
- 效果基本相当，甚至略有提升

2.3 Deep Norm

2.3.1 Deep Norm 思路

Deep Norm 方法在执行 Layer Norm 之前，up-scale 了残差连接 ($\alpha > 1$)；另外，在初始化阶段 down-scale 了模型参数 ($\beta < 1$)。

2.3.2 Deep Norm 代码实现

```
def deepnorm(x):
    return LayerNorm(x*  + f(x))

def deepnorm_init(w):
    if w in ['ffn', 'v_proj', 'out_proj']:
        nn.init.xavier_normal_(w, gain= )
    elif w in ['q_proj', 'k_proj']:
        nn.init.xavier_normal_(w, gain=1)
```

2.3.3 Deep Norm 的优点

Deep Norm 可以缓解爆炸式模型更新的问题，把模型更新限制在常数，使得模型训练过程更稳定。

2.4 Layer Normalization 的位置设计

2.4.1 LN 在 LLMs 中的不同位置

Post LN

- **位置**: Layer Norm 在残差链接之后
- **缺点**: Post LN 在深层的梯度范式逐渐增大，导致使用 post-LN 的深层 transformer 容易出现训练不稳定的问题

Pre-LN

- **位置**: Layer Norm 在残差链接中

- **优点**: 相比于 Post-LN，Pre LN 在深层的梯度范式近似相等，所以使用 Pre-LN 的深层 transformer 训练更稳定，可以缓解训练不稳定问题
- **缺点**: 相比于 Post-LN，Pre-LN 的模型效果略差

Sandwich-LN

- **位置**: 在 pre-LN 的基础上，额外插入了一个 layer norm
- **优点**: Cogview 用来避免值爆炸的问题
- **缺点**: 训练不稳定，可能会导致训练崩溃

2.5 Layer Normalization 对比分析

2.5.1 各模型使用的 Normalization 方法

表 2.1: LLMs 各模型使用的 Layer Normalization 方法对比

模型	Normalization 方法
GPT3	Pre Layer Norm
LLaMA	Pre RMS Norm
baichuan	Pre RMS Norm
ChatGLM-6B	Post Deep Norm
ChatGLM2-6B	Post RMS Norm
Bloom	Pre Layer Norm
Falcon	Pre Layer Norm

2.5.2 特殊说明

BLOOM 在 embedding 层后添加 layer normalization，有利于提升训练稳定性，但可能会带来很大的性能损失。

第三章　LLMs 激活函数篇

3.1　FFN 块基础

3.1.1　FFN 块计算公式

前馈神经网络（FFN）块的计算公式如下：

$$FFN(x) = f(xW_1 + b_1)W_2 + b_2$$

3.2　常见激活函数

3.2.1　GeLU 激活函数

GeLU（高斯误差线性单元）激活函数的计算公式如下：

$$GeLU(x) \approx 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$$

3.2.2　Swish 激活函数

Swish 激活函数的计算公式如下：

$$Swish_\beta(x) = x \cdot \sigma(\beta x)$$

3.3　GLU 线性门控单元

3.3.1　基础 GLU 计算公式

使用 GLU（门控线性单元）的 FFN 块计算公式：

$$GU(x) = \sigma(xW + b) \otimes xV$$
$$FFN_{GLU} = (f(xW_1) \otimes xV)W_2$$

3.3.2 GeGLU 计算公式

使用 GeLU 的 GLU 块计算公式：

$$GeGLU(x) = GeLU(xW) \otimes xV$$

3.3.3 SwiGLU 计算公式

使用 Swish 的 GLU 块计算公式：

$$SwiGLU = Swish_\beta(xW) \otimes xV$$

3.3.4 参数规模说明

- 传统 FFN：2 个可训练权重矩阵，中间维度为 $4h$
- SwiGLU FFN：3 个可训练权重矩阵，中间维度为 $4h \times 2/3$

维度计算示例：

$$4h = 4 \times 4096 = 16384$$
$$\frac{2}{3} \times 4h = 10022 \rightarrow 11008$$

3.4 各 LLMs 激活函数使用情况

表 3.1: 各 LLMs 模型使用的激活函数对比

模型	激活函数
GPT3	GeLU
LLaMA	SwiGLU
LLaMA2	SwiGLU
baichuan	SwiGLU
ChatGLM-6B	GeLU
ChatGLM2-6B	SwiGLU
Bloom	GeLU
Falcon	GeLU

表 3.2: LLaMA 模型参数结构示例

模块类型	模块名称	参数形状	参数数量
Embedding & Layers	model.embed_tokens.weight	[32000, 4096]	131072000
	model.layers.0.self_attn.q_proj.weight	[4096, 4096]	16777216
	model.layers.0.self_attn.k_proj.weight	[4096, 4096]	16777216
	model.layers.0.self_attn.v_proj.weight	[4096, 4096]	16777216
	model.layers.0.self_attn.o_proj.weight	[4096, 4096]	16777216
	model.layers.0.mlp.gate_proj.weight	[11008, 4096]	45088768
	model.layers.0.mlp.down_proj.weight	[4096, 11008]	45088768
	model.layers.0.mlp.up_proj.weight	[11008, 4096]	45088768
	model.layers.0.input_layernorm.weight	[4096]	4096
	model.layers.0.post_attention_layernorm.weight	[4096]	4096

3.5 模型参数结构示例

3.5.1 LLaMA 模型参数结构

3.5.2 Bloom 模型参数结构

3.5.3 特殊说明

BLOOM 在 embedding 层后添加 layer normalization，有利于提升训练稳定性，但可能会带来很大的性能损失。

表 3.3: Bloom 模型参数结构示例

模块类型	模块名称	参数形状	参数数量
Embedding	transformer.word_embeddings.weight	[250880, 4096]	1027604480
	transformer.word_embeddings_layernorm.weight	[4096]	4096
	transformer.word_embeddings_layernorm.bias	[4096]	4096
	transformer.h.0.input_layernorm.weight	[4096]	4096
	transformer.h.0.input_layernorm.bias	[4096]	4096
	transformer.h.0.self_attention.query_key_value.weight	[12288, 4096]	50331648
	transformer.h.0.self_attention.query_key_value.bias	[12288]	12288
	transformer.h.0.self_attention.dense.weight	[4096, 4096]	16777216
	transformer.h.0.self_attention.dense.bias	[4096]	4096
	transformer.h.0.post_attention_layernorm.weight	[4096]	4096
	transformer.h.0.post_attention_layernorm.bias	[4096]	4096
	transformer.h.0.mlp.dense_h_to_4h.weight	[16384, 4096]	67108864
	transformer.h.0.mlp.dense_h_to_4h.bias	[16384]	16384
	transformer.h.0.mlp.dense_4h_to_h.weight	[4096, 16384]	67108864
	transformer.h.0.mlp.dense_4h_to_h.bias	[4096]	4096

第四章　Attention 升级面

4.1　传统 Attention 的问题

- 传统 Attention 存在上下文长度约束问题
- 传统 Attention 速度慢，内存占用大

4.2　Attention 优化方向

- 提升上下文长度
- 加速、减少内存占用

4.3　Attention 变体概述

- **稀疏 Attention：**将稀疏偏差引入 attention 机制可以降低复杂性
- **线性化 Attention：**解开 attention 矩阵与内核特征图，然后以相反的顺序计算 attention 以实现线性复杂度
- **原型和内存压缩：**这类方法减少了查询或键值记忆对的数量，以减少注意力矩阵的大小
- **低阶 self-Attention：**这一系列工作捕获了 self-Attention 的低阶属性
- **Attention 与先验：**该研究探索了用先验 attention 分布来补充或替代标准 attention
- **改进多头机制：**该系列研究探索了不同的替代多头机制

4.4　Multi-Query Attention

4.4.1　Multi-head Attention 存在的问题

- **训练过程：**不会显著影响训练过程，训练速度不变，会引起非常细微的模型效果损失
- **推理过程：**反复加载巨大的 KV cache，导致内存开销大，性能是内存受限

4.4.2　Multi-Query Attention 介绍

Multi-Query Attention 在所有注意力头上共享 key 和 value。

4.4.3 Multi-head Attention 与 Multi-Query Attention 对比

- **Multi-head Attention**：每个注意力头都有各自的 query、key 和 value
- **Multi-query Attention**：在所有的注意力头上共享 key 和 value

4.4.4 各模型参数配置对比

表 4.1: 各模型注意力机制参数配置对比

模型	n_heads	head_dim	FFN 中间维度	维度 h
LLaMA	32	128	11008	4096
baichuan	32	128	11008	4096
ChatGLM-6B	32	128	4h, 16384	4096
ChatGLM2-6B	32	128	13696	4096
Bloom	32	128	4h, 16384	4096
Falcon	71	64	4h, 18176	4544

4.4.5 Multi-Query Attention 的模型实现差异

Falcon、PaLM、ChatGLM2-6B 都使用了 Multi-query Attention，但有细微差别：

- **为了保持参数量一致：**
- **Falcon**：把隐藏维度从 4096 增大到了 4544。多余的参数量分给了 Attention 块和 FFN 块
- **ChatGLM2**：把 FFN 中间维度从 11008 增大到了 13696。多余的参数分给了 FFN 块

4.4.6 Multi-Query Attention 的优势

减少 KV cache 的大小，减少显存占用，提升推理速度。

4.4.7 使用 Multi-Query Attention 的模型

代表模型：PaLM、ChatGLM2、Falcon 等

4.5 Grouped-query Attention

4.5.1 Grouped-query Attention 定义

Grouped query attention：介于 multi head 和 multi query 之间，多个 key 和 value。

4.5.2 使用 Grouped-query Attention 的模型

ChatGLM2，LLaMA2-34B/70B 使用了 Grouped query attention。

4.6 FlashAttention

4.6.1 FlashAttention 核心技术

- **核心**：用分块 softmax 等价替代传统 softmax
- **关键词**：HBM、SRAM、分块 Softmax、重计算、Kernel 融合

4.6.2 FlashAttention 优点

节约 HBM，高效利用 SRAM，省显存，提速度

4.6.3 使用 FlashAttention 的模型

Meta 推出的开源大模型 LLaMA，阿联酋推出的开源大模型 Falcon 都使用了 Flash Attention 来加速计算和节省显存

4.7 并行 Transformer Block

4.7.1 并行 Transformer Block 原理

用并行公式替换了串行，提升了 15% 的训练速度。

4.7.2 并行 Transformer Block 效果

- 在 8B 参数量规模，会有轻微的模型效果损失
- 在 62B 参数量规模，就不会损失模型效果

4.7.3 使用并行 Transformer Block 的模型

Falcon、PaLM 都使用了该技术来加速训练。

第五章　Transformers 操作篇

5.1　Transformers 库基础操作

5.1.1　如何利用 transformers 加载 Bert 模型？

```
import torch
from transformers import BertModel, BertTokenizer

# 这里我们调用bert-base模型,同时模型的词典经过小写处理
model_name = 'bert-base-uncased'

# 读取模型对应的tokenizer
tokenizer = BertTokenizer.from_pretrained(model_name)

# 载入模型
model = BertModel.from_pretrained(model_name)

# 输入文本
input_text = "Here is some text to encode"

# 通过tokenizer把文本变成token_id
input_ids = tokenizer.encode(input_text, add_special_tokens=True)
# input_ids: [101, 2182, 2003, 2070, 3793, 2000, 4372, 16044, 102]

input_ids = torch.tensor([input_ids])

# 获得BERT模型最后一个隐层结果
with torch.no_grad():
    last_hidden_states = model(input_ids)[0]
    # Models outputs are now tuples
```

```
"""
tensor([[[-0.0549,  0.1053, -0.1065, ..., -0.3550,  0.0686,  0.6506],
         [-0.5759, -0.3650, -0.1383, ..., -0.6782,  0.2092, -0.1639],
         [-0.1641, -0.5597,  0.0150, ..., -0.1603, -0.1346,  0.6216],
         [ 0.2448,  0.1254,  0.1587, ..., -0.2749, -0.1163,  0.8809],
         [ 0.0481,  0.4950, -0.2827, ..., -0.6097, -0.1212,  0.2527],
         [ 0.9046,  0.2137, -0.5897, ...,  0.3040, -0.6172, -0.1950]]])
shape: (1, 9, 768)
"""
```

可以看到，包括 import 在内的不到十行代码，我们就实现了读取一个预训练过的 BERT 模型，来 encode 我们指定的一个文本，对文本的每一个 token 生成 768 维的向量。如果是二分类任务，我们接下来就可以把第一个 token 也就是 [CLS] 的 768 维向量，接一个 linear 层，预测出分类的 logits，或者根据标签进行训练。

5.1.2 如何利用 transformers 输出 Bert 指定 hidden_state？

Bert 默认是十二层，但是有时候预训练时并不需要利用全部利用，而只需要预训练前面几层即可，此时该怎么做呢？

下载到 bert-base-uncased 的模型目录里面包含配置文件 config.json，该文件中包含 output_hidden_states，可以利用该参数来设置编码器内隐藏层层数。

5.2 BERT 输出向量获取

5.2.1 BERT 模型输出结构

BERT 模型输出包含以下几个部分：

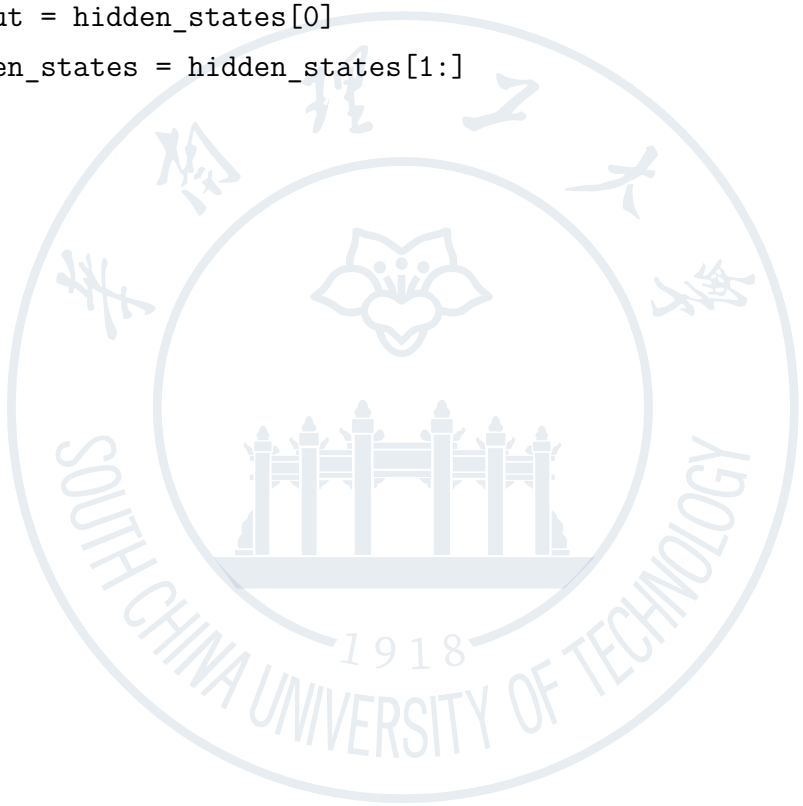
- **last_hidden_state**: shape 是 (batch_size, sequence_length, hidden_size), hidden_size=768, 它是模型最后一层输出的隐藏状态
- **pooler_output**: shape 是 (batch_size, hidden_size),这是序列的第一个 token(classification token) 的最后一层的隐藏状态，它是由线性层和 Tanh 激活函数进一步处理的，这个输出不是对输入的语义内容的一个很好的总结，对于整个输入序列的隐藏状态序列的平均化或池化通常更好。
- **hidden_states**: 这是输出的一个可选项,如果输出,需要指定 config.output_hidden_states=True, 它也是一个元组，它的第一个元素是 embedding，其余元素是各层的输出，每个元素的形状是 (batch_size, sequence_length, hidden_size)
- **attentions**: 这也是输出的一个可选项,如果输出,需要指定 config.output_attentions=True, 它也是一个元组，它的元素是每一层的注意力权重，用于计算 self-attention heads 的加权平均值

5.2.2 获取每一层网络的向量输出

```
## 最后一层的所有token向量
outputs.last_hidden_state

## cls向量
outputs.pooler_output

## hidden_states，包括13层，第一层即索引0是输入embedding向量，
## 后面1-12索引是每层的输出向量
hidden_states = outputs.hidden_states
embedding_output = hidden_states[0]
attention_hidden_states = hidden_states[1:]
```



第六章　LLMs 损失函数篇

6.1　KL 散度

KL(Kullback-Leibler) 散度衡量了两个概率分布之间的差异。其公式为：

$$D_{KL}(P \parallel Q) = -\sum_{x \in X} P(x) \log \frac{1}{P(x)} + \sum_{x \in X} P(x) \log \frac{1}{Q(x)}$$

6.2　交叉熵损失函数

交叉熵损失函数 (Cross-Entropy Loss Function) 是用于度量两个概率分布之间的差异的一种损失函数。在分类问题中，它通常用于衡量模型的预测分布与实际标签分布之间的差异。

$$H(p, q) = -\sum_{i=1}^{N} p_i \log(q_i) - (1 - p_i) \log(1 - q_i)$$

注：其中，p 表示真实标签，q 表示模型预测的标签，N 表示样本数量。该公式可以看作是一个基于概率分布的比较方式，即将真实标签看做一个概率分布，将模型预测的标签也看做一个概率分布，然后计算它们之间的交叉熵。

物理意义：交叉熵损失函数可以用来衡量实际标签分布与模型预测分布之间的" 信息差"。当两个分布完全一致时，交叉熵损失为 0，表示模型的预测与实际情况完全吻合。当两个分布之间存在差异时，损失函数的值会增加，表示预测错误程度的大小。

6.3　KL 散度与交叉熵的区别

KL 散度指的是相对熵，KL 散度是两个概率分布 P 和 Q 差别的非对称性的度量。KL 散度越小表示两个分布越接近。也就是说 KL 散度是不对称的，且 KL 散度的值是非负数。（也就是熵和交叉熵的差）

- 交叉熵损失函数是二分类问题中最常用的损失函数，由于其定义出于信息学的角度，可以泛化到多分类问题中。
- KL 散度是一种用于衡量两个分布之间差异的指标，交叉熵损失函数是 KL 散度的一种特殊形式。在二分类问题中，交叉熵函数只有一项，而在多分类问题中有多项。

6.4 多任务学习各 loss 差异过大处理

多任务学习中，如果各任务的损失差异过大，可以通过动态调整损失权重、使用任务特定的损失函数、改变模型架构或引入正则化等方法来处理。目标是平衡各任务的贡献，以便更好地训练模型。

6.5 分类问题为什么用交叉熵损失函数不用均方误差 (MSE)?

交叉熵损失函数通常在分类问题中使用，而均方误差 (MSE) 损失函数通常用于回归问题。这是因为分类问题和回归问题具有不同的特点和需求。

分类问题的目标是将输入样本分到不同的类别中，输出为类别的概率分布。交叉熵损失函数可以度量两个概率分布之间的差异，使得模型更好地拟合真实的类别分布。它对概率的细微差异更敏感，可以更好地区分不同的类别。此外，交叉熵损失函数在梯度计算时具有较好的数学性质，有助于更稳定地进行模型优化。

相比之下，均方误差 (MSE) 损失函数更适用于回归问题，其中目标是预测连续数值而不是类别。MSE 损失函数度量预测值与真实值之间的差异的平方，适用于连续数值的回归问题。在分类问题中使用 MSE 损失函数可能不太合适，因为它对概率的微小差异不够敏感，而且在分类问题中通常需要使用激活函数 (如 sigmoid 或 softmax) 将输出映射到概率空间，使得 MSE 的数学性质不再适用。

综上所述，交叉熵损失函数更适合分类问题，而 MSE 损失函数更适合回归问题。

6.6 信息增益

信息增益是在决策树算法中用于选择最佳特征的一种评价指标。在决策树的生成过程中，选择最佳特征来进行节点的分裂是关键步骤之一，信息增益可以帮助确定最佳特征。

信息增益衡量了在特征已知的情况下，将样本集合划分成不同类别的纯度提升程度。它基于信息论的概念，使用熵来度量样本集合的不确定性。具体而言，信息增益是原始集合的熵与特定特征下的条件熵之间的差异。

在决策树的生成过程中，选择具有最大信息增益的特征作为当前节点的分裂标准，可以将样本划分为更加纯净的子节点。信息增益越大，意味着使用该特征进行划分可以更好地减少样本集合的不确定性，提高分类的准确性。

6.7 多分类的分类损失函数 (Softmax)

多分类的分类损失函数采用 Softmax 交叉熵 (Softmax Cross Entropy) 损失函数。Softmax 函数可以将输出值归一化为概率分布，用于多分类问题的输出层。Softmax 交叉熵损失函数可以写成：$-\sum_{i=1}^{n} y_i \log(p_i)$

注：其中，n 是类别数，y_i 是第 i 类的真实标签，p_i 是第 i 类的预测概率。

6.8　Softmax 和交叉熵损失计算

softmax 计算公式如下：

$$y = \frac{e^{f_i}}{\sum_j e^{f_j}}$$

多分类交叉熵：

$$L = \frac{1}{N}\sum_i L_i = -\frac{1}{N}\sum_i \sum_{c=1}^{M} y_{ic}\log(p_{ic})$$

其中：

- M——类别的数量
- y_{ic}——符号函数 (0 或 1)，如果样本 i 的真实类别等于 c 取 1，否则取 0
- p_{ic}——观测样本 i 属于类别 c 的预测概率

二分类交叉熵：

$$L = \frac{1}{N}\sum_i L_i = \frac{1}{N}\sum_i -[y_i \cdot \log(p_i) + (1-y_i)\cdot\log(1-p_i)]$$

其中：

- y_i – 表示样本 i 的 label，正类为 1，负类为 0
- p_i – 表示样本 i 预测为正类的概率

6.9　Softmax 数值稳定性问题

如果 softmax 的 e 次方超过 float 的值了怎么办？

将分子分母同时除以 x 中的最大值，可以解决。

$$\tilde{x}_k = \frac{e^{x_k - \max(x)}}{e^{x_1-\max(x)} + e^{x_2-\max(x)} + \ldots + e^{x_k-\max(x)} + \ldots + e^{x_n-\max(x)}}$$

第七章　相似度函数篇

7.1　相似度计算方法

7.1.1　除了余弦相似度还有哪些方法

除了余弦相似度 (cosine similarity) 之外，常见的相似度计算方法还包括欧氏距离、曼哈顿距离、Jaccard 相似度、皮尔逊相关系数等。

7.2　对比学习

7.2.1　对比学习概述

对比学习是一种无监督学习方法，通过训练模型使得相同样本的表示更接近，不同样本的表示更远离，从而学习到更好的表示。对比学习通常使用对比损失函数，例如 Siamese 网络、Triplet 网络等，用于学习数据之间的相似性和差异性。

7.3　对比学习中的负样本问题

7.3.1　负样本的重要性

对比学习中负样本的重要性取决于具体的任务和数据。负样本可以帮助模型学习到样本之间的区分度，从而提高模型的性能和泛化能力。然而，负样本的构造成本可能会较高，特别是在一些领域和任务中。

7.3.2　负样本构造成本过高的解决方案

为了解决负样本构造成本过高的问题，可以考虑以下方法：

- **降低负样本的构造成本**：通过设计更高效的负样本生成算法或采样策略，减少负样本的构造成本。例如，可以利用数据增强技术生成合成的负样本，或者使用近似采样方法选择与正样本相似但不相同的负样本。
- **确定关键负样本**：根据具体任务的特点，可以重点关注一些关键的负样本，而不是对所有负样本进行详细的构造。这样可以降低构造成本，同时仍然能够有效训练模型。

- **迁移学习和预训练模型**：利用预训练模型或迁移学习的方法，可以在其他领域或任务中利用已有的负样本构造成果，减少重复的负样本构造工作。

第八章　大模型 (LLMs) 进阶面

8.1　生成式大模型概述

8.1.1　什么是生成式大模型？

生成式大模型 (一般简称大模型 LLMs) 是指能用于创作新内容，例如文本、图片、音频以及视频的一类深度学习模型。相比普通深度学习模型，主要有两点不同：

- 模型参数量更大，参数量都在 Billion 级别
- 可通过条件或上下文引导，产生生成式的内容 (所谓的 prompt engineer 就是由此而来)

8.2　文本生成多样性机制

8.2.1　大模型如何让生成的文本丰富而不单调？

从训练角度来看

- 基于 Transformer 的模型参数量巨大，有助于模型学习到多样化的语言模式与结构
- 各种模型微调技术的出现，例如 P-Tuning、Lora，让大模型微调成本更低，也可以让模型在垂直领域有更强的生成能力
- 在训练过程中加入一些设计好的 loss，也可以更好地抑制模型生成单调内容

从推理角度来看

基于 Transformer 的模型可以通过引入各种参数与策略，例如 temperature，nucleus sampler 来改变每次生成的内容。

8.3　LLMs 复读机问题

8.3.1　什么是 LLMs 复读机问题？

- **字符级别重复**：指大模型针对一个字或一个词重复不断的生成
- **语句级别重复**：大模型针对一句话重复不断的生成

- **章节级别重复**：多次相同的 prompt 输出完全相同或十分近似的内容，没有一点创新性的内容
- **信息熵偏低**：大模型针对不同的 prompt 也可能会生成类似的内容，且有效信息很少

8.3.2　为什么会出现 LLMs 复读机问题？

- 数据偏差：训练数据中存在大量的重复文本或者某些特定的句子或短语出现频率较高
- 训练目标的限制：自监督学习的目标可能使得模型更倾向于生成与输入相似的文本
- 缺乏多样性的训练数据：训练数据中缺乏多样性的语言表达和语境
- 模型结构和参数设置：注意力机制和生成策略可能导致模型更倾向于复制输入的文本
- induction head 机制的影响：模型会倾向于从前面已经预测的 word 里面挑选最匹配的词
- 信息熵角度分析：某些文本类型（如电商标题）信息熵高，模型预测困难

8.3.3　如何缓解 LLMs 复读机问题？

Unlikelihood Training

思路：在训练中加入对重复词的抑制来减少重复输出

计算公式：

$$\mathcal{L}^t_{\text{UL-token}}(p_\theta(\cdot|x_{<t}), \mathcal{C}^t) = -\alpha \cdot \sum_{c \in \mathcal{C}^t} \log(1 - p_\theta(c|x_{<t})) - \log p_\theta(x_t|x_{<t})$$

$$\mathcal{L}^t_{UL}\left(p_\theta\left(\cdot \mid x_{<t}\right), \mathcal{C}^t\right) = -\sum_{c \in \mathcal{C}^t} \log\left(1 - p_\theta\left(c \mid x_{<t}\right)\right)$$

引入噪声

在生成文本时，引入一些随机性或噪声，增加生成文本的多样性。

Repetition Penalty

思路：重复性惩罚方法通过在模型推理过程中加入重复惩罚因子，对原有 softmax 结果进行修正

计算公式：

$$p_i = \frac{\exp(x_i/(T \cdot I(i \in g))}{\sum_j \exp(x_j/(T \cdot I(j \in g))} \qquad I(c) = \theta \text{ if } c \text{ is True else } 1$$

Contrastive Search

思路：对比 loss 以及对比搜索两个创新点，从模型训练和模型推理层面缓解生成式模型重复问题

对比 loss 公式：

$$\mathcal{L}_{CL} = \frac{1}{|x| \times (|x|-1)} \sum_{i=1}^{|x|} \sum_{j=1, j\neq i}^{|x|} \max\{0, \rho - s(h_{x_i}, h_{x_i}) + s(h_{x_i}, h_{x_j})\}$$

对比搜索公式：

$$x_t = \arg\max_{v \in V^{(k)}} \{(1-\alpha) \times p_\theta(v|x_{<t}) - \alpha \times (\max\{s(h_v, h_{x_j}) : 1 \leq j \leq t-1\})\}$$

Beam Search

思路：在每一个时间步，保留 num_beams 个最优输出，而不是只保留 1 个

TopK sampling

通过对 Softmax 的输出结果 logit 中最大的 K 个 token 采样来选择输出的 token

Nucleus sampler

不限制 K 的数目，而是通过 Softmax 后排序 token 的概率，当概率大于 P 时停止

Temperature

调整公式：

$$p_i = \frac{\exp\left(x_i/(T \cdot I(i \in g)\right)}{\sum_j \exp\left(x_j/(T \cdot I(j \in g)\right)} \quad I(c) = \theta \text{ if} c \text{ is True else1}$$

No repeat ngram size

通过限制设置的 ngram 不能出现重复，强制模型不生成重复的 token

重复率指标检测

使用 seq-rep-N，uniq-seq，rep，wrep 等指标进行监测

后处理和过滤

对生成的文本进行后处理和过滤，去除重复的句子或短语

人工干预和控制

引入人工干预和控制机制，对生成的文本进行审查和筛选

8.4　LLaMA 系列问题

8.4.1　LLaMA 输入句子长度理论上可以无限长吗？

- 限制在训练数据。理论上 rope 的 LLaMA 可以处理无限长度，但实际上存在限制
- 计算资源：生成长句子需要更多的计算资源
- 模型训练和推理：处理长句子可能面临梯度消失或梯度爆炸的问题
- 上下文建模：需要能够捕捉长句子中的语义和语法结构

8.5　模型选择指南

8.5.1　什么情况用 Bert 模型，什么情况用 LLaMA、ChatGLM 类大模型？

- **Bert 模型**：110M 参数，NLU 任务效果很好，单卡 GPU 部署，速度快
- **大模型**：6B-7B 参数，处理所有 NLP 任务，部署成本高，预测速度慢
- **建议**：
  - NLU 相关任务（实体识别、信息抽取、文本分类）用 BERT 模型
  - NLG 任务，纯中文任务用 ChatGLM-6B，中英文任务用 chinese-alpaca-plus-7b-hf

8.6　专业领域大模型需求

8.6.1　各个专业领域是否需要各自的大模型来服务？

- 领域特定知识：需要针对特定领域知识进行训练
- 语言风格和惯用语：不同领域有独特的语言特点
- 领域需求的差异：不同领域对文本处理的需求不同
- 数据稀缺性：某些领域数据相对较少

8.7　长文本处理技术

8.7.1　如何让大模型处理更长的文本？

LongChat 方法

- 将新的长度压缩到原来长度上，复用原来的位置信息
- 用训练语料做微调，超过限定长度的文本被截断

其他技术方向

- position 等比例缩放和 ALiBi 方法

- 商业模型的可能技术：稀疏化、MoE 技术、Multi-Query Attention
- Linear Attention：将 Attention 复杂度从 $O(N^2)$ 降低为 $O(N)$
- RWKV：结合 RNN 和 Transformer 的优点

第九章　大模型 (LLMs) 微调面

9.1　微调基础问题

9.1.1　全参数微调显存需求

一般 nB 的模型，最低需要 16-20nG 的显存。(cpu offload 基本不开的情况下)

vicuna-7B 为例，官方样例配置为 4*A100 40G，测试了一下确实能占满显存。(global batch size 128，max length 2048) 当然训练时用了 FSDP、梯度累积、梯度检查点等方式降显存。

9.1.2　SFT 后模型性能下降原因

原版答案：SFT 的重点在于激发大模型的能力，SFT 的数据量一般也就是万恶之源 alpaca 数据集的 52k 量级，相比于预训练的数据还是太少了。

如果抱着灌注领域知识而不是激发能力的想法，去做 SFT 的话，可能确实容易把 LLM 弄傻。

新版答案：指令微调是为了增强 (或解锁) 大语言模型的能力。

其真正作用：指令微调后，大语言模型展现出泛化到未见过任务的卓越能力，即使在多语言场景下也能有不错表现。

9.2　数据构建与处理

9.2.1　SFT 指令微调数据构建原则

1. 代表性。应该选择多个有代表性的任务；
2. 数据量。每个任务实例数量不应太多 (比如：数百个) 否则可能会潜在地导致过拟合问题并影响模型性能；
3. 不同任务数据量占比。应该平衡不同任务的比例，并且限制整个数据集的容量 (通常几千或几万)，防止较大的数据集压倒整个分布。

9.2.2　领域模型 Continue PreTrain 数据选取

技术标准文档或领域相关数据是领域模型 Continue PreTrain 的关键。因为领域相关的网站和资讯重要性或者知识密度不如书籍和技术标准。

9.2.3　缓解模型遗忘通用能力

动机：仅仅使用领域数据集进行模型训练，模型很容易出现灾难性遗忘现象。

解决方法：通常在领域训练的过程中加入通用数据集

那么这个比例多少比较合适呢？

目前还没有一个准确的答案。主要与领域数据量有关系，当数据量没有那么多时，一般领域数据与通用数据的比例在 1:5 到 1:10 之间是比较合适的。

9.2.4　Multi-Task Instruction PreTraining

领域模型 Continue PreTrain 时可以同步加入 SFT 数据，即 MIP，Multi-Task Instruction PreTraining。

预训练过程中，可以加下游 SFT 的数据，可以让模型在预训练过程中就学习到更多的知识。

9.3　模型选择与配置

9.3.1　基座模型选择策略

仅用 SFT 做领域模型时，资源有限就用在 Chat 模型基础上训练，资源充足就在 Base 模型上训练。

(资源 = 数据 + 显卡)

资源充足时可以更好地拟合自己的数据，如果你只拥有小于 10k 数据，建议你选用 Chat 模型作为基座进行微调；如果你拥有 100k 的数据，建议你在 Base 模型上进行微调。

9.3.2　数据输入格式要求

在 Chat 模型上进行 SFT 时，请一定遵循 Chat 模型原有的系统指令 & 数据输入格式。

建议不采用全量参数训练，否则模型原始能力会遗忘较多。

9.3.3　领域评测集构建

领域评测集时必要内容，建议有两份，一份选择题形式自动评测、一份开放形式人工评测。

选择题形式可以自动评测，方便模型进行初筛；开放形式人工评测比较浪费时间，可以用作精筛，并且任务形式更贴近真实场景。

9.3.4　词表扩增必要性

领域词表扩增真实解决的问题是解码效率的问题，给模型效果带来的提升可能不会有很大。

9.4　训练实践与经验

9.4.1　训练自己的大模型步骤

如果我现在做一个 sota 的中文 GPT 大模型，会分 2 步走：1. 基于中文文本数据在 LLaMA-65B 上二次预训练；2. 加 CoT 和 instruction 数据，用 FT+LoRA SFT。

提炼下方法，一般分为两个阶段训练：

- 第一阶段:扩充领域词表,比如金融领域词表,在海量领域文档数据上二次预训练 LLaMA 模型；
- 第二阶段：构造指令微调数据集，在第一阶段的预训练模型基础上做指令精调。还可以把指令微调数据集拼起来成文档格式放第一阶段里面增量预训练，让模型先理解下游任务信息。

当然，有低成本方案，因为我们有 LoRA 利器，第一阶段和第二阶段都可以用 LoRA 训练，如果不用 LoRA，就全参微调，大概 7B 模型需要 8 卡 A100，用了 LoRA 后，只需要单卡 3090 就可以了。

9.4.2　多轮对话微调方法

```
from transformers import AutoTokenizer, AutoModel
>>> tokenizer = AutoTokenizer.from_pretrained("THUDM/chatglm-6b",
trust_remote_code=True)
>>> model = AutoModel.from_pretrained("THUDM/chatglm-6b",
trust_remote_code=True).half().cuda()
>>> model = model.eval()
>>> response, history = model.chat(tokenizer, "你好", history=[])
>>> print(f"response: {response}")
>>> print(f"history: {history}")
response: 你好!我是人工智能助手 ChatGLM-6B,很高兴见到你,欢迎问我任何问题。
history: ["你好", "你好 !我是人工智能助手 ChatGLM-6B,很高兴见到你,欢迎问我任
何问题。"]
```

解决方法：

- 对历史对话做一层文本摘要，取其精华去其糟粕
- 将历史对话做成一个 embedding
- 如果是任务型对话，可以将用户意图和槽位作为上一轮信息传递给下一轮

9.5　关键技术问题

9.5.1　灾难性遗忘问题

所谓的灾难性遗忘：即学习了新的知识之后，几乎彻底遗忘掉之前习得的内容。这在微调 ChatGLM-6B 模型时，有同学提出来的问题，表现为原始 ChatGLM-6B 模型在知识问答如" 失眠怎么办" 的回答上是正确的，但引入特定任务 (如拼写纠错 CSC) 数据集微调后，再让模型预测" 失眠怎么办" 的结果就答非所问了。

应该是微调训练参数调整导致的，微调初始学习率不要设置太高，lr=2e-5 或者更小，可以避免此问题，不要大于预训练时的学习率。

9.5.2　微调模型显存需求

表 9.1: 微调模型显存需求对比

模型版本	7B	13B	33B	65B
原模型大小 (FP16)	13 GB	24 GB	60 GB	120 GB
量化后大小 (8-bit)	7.8 GB	14.9 GB	-	-
量化后大小 (4-bit)	3.9 GB	7.8 GB	19.5 GB	38.5 GB

9.5.3　SFT 学习内容

1. 预训练 → 在大量无监督数据上进行预训练，得到基础模型 → 将预训练模型作为 SFT 和 RLHF 的起点。
2. SFT → 在有监督的数据集上进行 SFT 训练，利用上下文信息等监督信号进一步优化模型 → 将 SFT 训练后的模型作为 RLHF 的起点。
3. RLHF → 利用人类反馈进行强化学习，优化模型以更好地适应人类意图和偏好 → 将 RLHF 训练后的模型进行评估和验证，并进行必要的调整。

9.6　训练优化技术

9.6.1　Batch Size 设置问题

Batch Size 太小的问题：当 batch size 较小时，更新方向 (即对真实梯度的近似) 会具有很高的方差，导致的梯度更新主要是噪声。

Batch Size 太大的问题：当 batch size 非常大时，我们从训练数据中抽样的任何两组数据都会非常相似 (因为它们几乎完全匹配真实梯度)。因此，在这种情况下，增加 batch size 几乎不会改善性能。

最优步长公式：

$$\epsilon_{opt}(B) = argmin_{\epsilon} E[L(\theta - \epsilon G_{est})] = \frac{\epsilon_{\max}}{1 + \mathcal{B}_{\text{noise}}/B}$$

噪声尺度估计：

$$\mathcal{B}_{noise} = \frac{tr(H\Sigma)}{G^T H G}$$

9.6.2　优化器选择

除了 Adam 和 AdamW，其他优化器如 Sophia 也值得研究，它使用梯度曲率而非方差进行归一化，可能提高训练效率和模型性能。

9.7　数据构建建议

9.7.1　预训练数据集选择

通过分析发现现有的开源大模型进行预训练的过程中会加入书籍、论文等数据。主要是因为这些数据的数据质量较高，领域相关性比较强，知识覆盖率 (密度) 较大，可以让模型更适应考试。

9.7.2　微调数据集构建原则

1. 选取的训练数据要干净、并具有代表性。
2. 构建的 prompt 尽量多样化，提高模型的鲁棒性。
3. 进行多任务同时进行训练的时候，要尽量使各个任务的数据量平衡。

9.8　Loss 突刺问题分析

9.8.1　Loss 突刺现象

loss spike 指的是预训练过程中，尤其容易在大模型 (100B 以上) 预训练过程中出现的 loss 突然暴涨的情况。

9.8.2 Adam 优化器与 Loss 突刺

Adam 优化器更新公式：

$$m_t = \frac{\beta_1}{1-\beta_1^t} m_{t-1} + \frac{1-\beta_1}{1-\beta_1^t} g_t$$

$$v_t = \frac{\beta_2}{1-\beta_2^t} v_{t-1} + \frac{1-\beta_2}{1-\beta_2^t} g_t^2$$

$$u_t = \frac{m_t}{\sqrt{v_t} + \varepsilon}$$

$$\theta_{t+1} = \theta_t - \eta_t u_t$$

9.8.3 Loss 突刺解决方案

1. 出现 loss spike 后更换 batch 样本的方法
2. 减小 learning rate
3. 减小 ε 大小，或者直接把 ε 设为 0
4. 使用 Embedding Layer Gradient Shrink(EGS) 技术

第十章　LLMs 训练经验帖

10.1　分布式训练框架选择

多用 DeepSpeed，少用 Pytorch 原生的 torchrun。在节点数量较少的情况下，使用何种训练框架并不是特别重要；然而，一旦涉及到数百个节点，DeepSpeed 显现出其强大之处，其简便的启动和便于性能分析的特点使其成为理想之选。

10.2　LLMs 训练实用建议

10.2.1　弹性容错和自动重启机制

大模型训练不是以往那种单机训个几小时就结束的任务，往往需要训练好几周甚至好几个月，这时候你就知道能稳定训练有多么重要。弹性容错能让你在机器故障的情况下依然继续重启训练；自动重启能让你在训练中断之后立刻重启训练。毕竟，大模型时代，节约时间就是节约钱。

10.2.2　定期保存模型

训练的时候每隔一段时间做个 checkpointing，这样如果训练中断还能从上次的断点来恢复训练。

10.2.3　规划训练目标

训练一次大模型的成本很高的。在训练之前先想清楚这次训练的目的，记录训练参数和中间过程结果，少做重复劳动。

10.2.4　关注 GPU 使用效率

有时候，即使增加了多块 A100 GPU，大型模型的训练速度未必会加快，这很可能是因为 GPU 使用效率不高，尤其在多机训练情况下更为明显。仅仅依赖 nvidia-smi 显示的 GPU 利用率并不足以准确反映实际情况，因为即使显示为 100%，实际 GPU 利用率也可能不是真

正的 100%。要更准确地评估 GPU 利用率，需要关注 TFLOPS 和吞吐率等指标，这些监控在 DeepSpeed 框架中都得以整合。

10.2.5 训练框架选择影响

对于同一模型，选择不同的训练框架，对于资源的消耗情况可能存在显著差异（比如使用 Huggingface Transformers 和 DeepSpeed 训练 OPT-30 相对于使用 Alpa 对于资源的消耗会低不少）。

10.2.6 环境配置注意事项

针对已有的环境进行分布式训练环境搭建时，一定要注意之前环境的 python、pip、virtualenv、setuptools 的版本。不然创建的虚拟环境即使指定对了 Python 版本，也可能会遇到很多安装依赖库的问题（GPU 服务器能够访问外网的情况下，建议使用 Docker 相对来说更方便）。

10.2.7 系统底层库升级谨慎性

遇到需要升级 GLIBC 等底层库需要升级的提示时，一定要慎重，不要轻易升级，否则，可能会造成系统宕机或很多命令无法操作等情况。

10.3 模型规模选择策略

进行大模型模型训练时，先使用小规模模型（如：OPT-125m/2.7b）进行尝试，然后再进行大规模模型（如：OPT-13b/30b...）的尝试，便于出现问题时进行排查。目前来看，业界也是基于相对较小规模参数的模型（6B/7B/13B）进行的优化，同时，13B 模型经过指令精调之后的模型效果已经能够到达 GPT4 的 90% 的效果。

10.4 加速卡选择建议

对于一些国产 AI 加速卡，目前来说，坑还比较多，如果时间不是时间非常充裕，还是尽量选择 Nvidia 的 AI 加速卡。

第十一章　大模型 (LLMs) LangChain 面

11.1　LangChain 基础概念

11.1.1　什么是 LangChain?

LangChain 是一个强大的框架，旨在帮助开发人员使用语言模型构建端到端的应用程序。它提供了一套工具、组件和接口，可简化创建由大型语言模型 (LLM) 和聊天模型提供支持的应用程序的过程。LangChain 可以轻松管理与语言模型的交互，将多个组件链接在一起，并集成额外的资源，例如 API 和数据库。

11.1.2　LangChain Agent

LangChain Agent 是框架中驱动决策制定的实体。它可以访问一组工具，并可以根据用户的输入决定调用哪个工具。

优点：LangChain Agent 帮助构建复杂的应用程序，这些应用程序需要自适应和特定于上下文的响应。当存在取决于用户输入和其他因素的未知交互链时，它们特别有用。

11.2　LangChain 核心概念

11.2.1　Components and Chains

- **Component**：模块化的构建块，可以组合起来创建强大的应用程序
- **Chain**：组合在一起以完成特定任务的一系列 Components(或其他 Chain)

注：一个 Chain 可能包括一个 Prompt 模板、一个语言模型和一个输出解析器，它们一起工作以处理用户输入、生成响应并处理输出。

11.2.2　Prompt Templates and Values

- **Prompt Template 作用**：负责创建 PromptValue，这是最终传递给语言模型的内容
- **Prompt Template 特点**：有助于将用户输入和其他动态信息转换为适合语言模型的格式

11.2.3 Example Selectors

作用：当您想要在 Prompts 中动态包含示例时，Example Selectors 很有用。他们接受用户输入并返回一个示例列表以在提示中使用，使其更强大和特定于上下文。

11.2.4 Output Parsers

- **作用**：负责将语言模型响应构建为更有用的格式
- **实现方法**：
  - 一种用于提供格式化指令
  - 另一种用于将语言模型的响应解析为结构化格式
- **特点**：使得在您的应用程序中处理输出数据变得更加容易

11.2.5 Indexes and Retrievers

- **Index**：一种组织文档的方式，使语言模型更容易与它们交互
- **Retrievers**：用于获取相关文档并将它们与语言模型组合的接口

注：LangChain 提供了用于处理不同类型的索引和检索器的工具和功能，例如矢量数据库和文本拆分器。

11.2.6 Chat Message History

- **作用**：负责记住所有以前的聊天交互数据，然后可以将这些交互数据传递回模型、汇总或以其他方式组合
- **优点**：有助于维护上下文并提高模型对对话的理解

11.2.7 Agents and Toolkits

- **Agent**：在 LangChain 中推动决策制定的实体。他们可以访问一套工具，并可以根据用户输入决定调用哪个工具
- **Toolkits**：一组工具，当它们一起使用时，可以完成特定的任务

11.3 LangChain 功能特性

11.3.1 主要功能

- **针对特定文档的问答**：根据给定的文档回答问题，使用这些文档中的信息来创建答案
- **聊天机器人**：构建可以利用 LLM 的功能生成文本的聊天机器人
- **Agents**：开发可以决定行动、采取这些行动、观察结果并继续执行直到完成的代理

11.3.2 LangChain 模型类型

LangChain model 是一种抽象，表示框架中使用的不同类型的模型：

- **LLM(大型语言模型)**：将文本字符串作为输入并返回文本字符串作为输出
- **聊天模型 (Chat Model)**：由语言模型支持，但具有更结构化的 API。将聊天消息列表作为输入并返回聊天消息
- **文本嵌入模型 (Text Embedding Models)**：将文本作为输入并返回表示文本嵌入的浮点列表

11.3.3 LangChain 特点

LangChain 旨在为六个主要领域的开发人员提供支持：

1. LLM 和提示：管理提示、优化，创建通用界面
2. 链 (Chain)：对 LLM 或其他实用程序的调用序列
3. 数据增强生成：与外部数据源交互以收集生成步骤的数据
4. Agents：让 LLM 做出有关行动的决定
5. 内存：维护链或代理调用之间的状态
6. 评估：使用 LLM 评估模型

11.4 LangChain 使用示例

11.4.1 调用 LLMs 生成回复

```
# 官方示例使用OPENAI接口
from langchain.llms import OpenAI
llm = OpenAI(model_name="text-davinci-003")
prompt = "你好"
response = llm(prompt)

# 用chatglm来演示该过程，封装一下即可
from transformers import AutoTokenizer, AutoModel
class chatGLM():
    def __init__(self, model_name) -> None:
        self.tokenizer = AutoTokenizer.from_pretrained(model_name,
            trust_remote_code=True)
        self.model = AutoModel.from_pretrained(model_name,
            trust_remote_code=True).half().cuda().eval()
    def __call__(self, prompt) -> Any:
        response, _ = self.model.chat(self.tokenizer, prompt)
```

```
        return response

llm = chatGLM(model_name="THUDM/chatglm-6b")
prompt = "你好"
response = llm(prompt)
print("response: %s" % response)
```

11.4.2　修改提示模板

```
from langchain import PromptTemplate

template = """
Explain the concept of {concept} in couple of lines
"""
prompt = PromptTemplate(input_variables=["concept"], template=template)
prompt = prompt.format(concept="regularization")
print("prompt=%s" % prompt)

template = "请给我解释一下{concept}的意思"
prompt = PromptTemplate(input_variables=["concept"], template=template)
prompt = prompt.format(concept="人工智能")
print("prompt=%s" % prompt)
```

11.4.3　链接多个组件处理任务

```
# chains --------
from langchain.chains import LLMChain
chain = LLMChain(llm=openAI(), prompt=promptTem)
print(chain.run("你好"))

# Chatglm对象不符合LLMChain类llm对象要求，模仿一下
class DemoChain():
    def __init__(self, llm, prompt) -> None:
        self.llm = llm
        self.prompt = prompt
    def run(self, query) -> Any:
        prompt = self.prompt.format(concept=query)
        print("query=%s->prompt=%s" % (query, prompt))
        response = self.llm(prompt)
```

```
        return response

chain = DemoChain(llm=llm, prompt=promptTem)
print(chain.run(query="天道酬勤"))
```

11.4.4 Embedding & Vector Store

```
# 官方示例代码，用的OpenAI的ada的文本Embedding模型
# 1) Embeding model
from langchain.embeddings import OpenAIEmbeddings
embeddings = OpenAIEmbeddings(model_name="ada")
query_result = embeddings.embed_query("你好")

# 2) 文本切割
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=100, chunk_overlap=0
)
texts = """天道酬勤"并不是鼓励人们不劳而获，而是提醒人们要遵循自然规律...
    """
texts = text_splitter.create_documents([texts])

# 3) 入库检索
import pinecone
from langchain.vectorstores import Pinecone
pinecone.init(api_key=os.getenv(""), environment=os.getenv(""))
index_name = "demo"
search = Pinecone.from_documents(texts=texts, embeddings, index_name=
    index_name)
query = "What is magical about an autoencoder?"
result = search.similarity_search(query)
```

11.5 LangChain 问题与解决方案

11.5.1 低效的令牌使用问题

- **问题**：Langchain 的令牌计数功能对于小数据集来说效率很低
- **解决方案**：Tiktoken 是 OpenAI 开发的 Python 库，用于更有效地解决令牌计数问题

11.5.2　文档问题

- **问题**：文档不充分且经常不准确，经常有 404 错误页面
- **原因**：与 Langchain 快速发展、版本迭代快速有关

11.5.3　概念混淆问题

- **问题**：代码库概念让人混淆，存在大量的”helper” 函数
- **示例**：简单的分割函数被复杂包装

11.5.4　行为不一致问题

- **问题**：隐藏重要细节和行为不一致，可能导致生产系统出现意想不到的问题
- **示例**：ConversationRetrievalChain 的输入问题重新措辞可能破坏对话自然流畅性

11.5.5　缺乏标准数据类型问题

- **问题**：缺乏表示数据的标准方法，阻碍与其他框架和工具的集成

11.6　LangChain 替代方案

11.6.1　LlamaIndex

LlamaIndex 是一个数据框架，可以很容易地将大型语言模型连接到自定义数据源。它可用于存储、查询和索引数据，还提供了各种数据可视化和分析工具。

11.6.2　Deepset Haystack

Deepset Haystack 是另外一个开源框架，用于使用大型语言模型构建搜索和问答应用程序。它基于 Hugging Face Transformers，提供了多种查询和理解文本数据的工具。

第十二章　多轮对话中让 AI 保持长期记忆的 8 种优化方式篇

12.1　前言

在基于大模型的 Agent 中，长期记忆的状态维护至关重要。在 OpenAI AI 应用研究主管博客《基于大模型的 Agent 构成》中，将记忆视为关键的组件之一。下面将结合 LangChain 中的代码，介绍 8 种不同的记忆维护方式在不同场景中的应用。

12.2　Agent 获取上下文对话信息的 8 种方式

12.2.1　获取全量历史对话

应用场景：以一般客服场景为例

在电信公司的客服聊天机器人场景中，如果用户在对话中先是询问了账单问题，接着又谈到了网络连接问题，ConversationBufferMemory 可以用来记住整个与用户的对话历史，可以帮助 AI 在回答网络问题时还记得账单问题的相关细节，从而提供更连贯的服务。

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory()
memory.save_context({"input": "你好"}, {"output": "怎么了"})
variables = memory.load_memory_variables({})
```

12.2.2　滑动窗口获取最近部分对话内容

应用场景：以商品咨询场景为例

在一个电商平台上，如果用户询问关于特定产品的问题（如手机的电池续航时间），然后又问到了配送方式，ConversationBufferWindowMemory 可以帮助 AI 只专注于最近的一两个问题（如配送方式），而不是整个对话历史，以提供更快速和专注的答复。

```
from langchain.memory import ConversationBufferWindowMemory
# 只保留最后1次互动的记忆
```

```
memory = ConversationBufferWindowMemory(k=1)
```

12.2.3　获取历史对话中实体信息

应用场景：以法律咨询场景为例

在法律咨询的场景中，客户可能会提到特定的案件名称、相关法律条款或个人信息（如”我在去年的交通事故中受了伤，想了解关于赔偿的法律建议”）。ConversationEntityMemory 可以帮助 AI 记住这些关键实体和实体关系细节，从而在整个对话过程中提供更准确、更个性化的法律建议。

```
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
memory = ConversationEntityMemory(llm=llm)
_input = {"input": "公众号《LLM应用全栈开发》的作者是莫尔索"}
memory.load_memory_variables(_input)
memory.save_context(
    _input,
    {"output": "是吗，这个公众号是干嘛的"}
)
print(memory.load_memory_variables({"input": "莫尔索是谁?"}))
# 输出，可以看到提取了实体关系
# {'history': 'Human: 公众号《LLM应用全栈开发》的作者是莫尔索\nAI: 是吗，
    这个公众号是干嘛的',
#  'entities': {'莫尔索': '《LLM应用全栈开发》的作者。'}}
```

12.2.4　利用知识图谱获取历史对话中的实体及其联系

应用场景：以医疗咨询场景为例

在医疗咨询中，一个病人可能会描述多个症状和过去的医疗历史（如” 我有糖尿病史，最近觉得经常口渴和疲劳”）。ConversationKGMemory 可以构建一个包含病人症状、疾病历史和可能的健康关联的知识图谱，从而帮助 AI 提供更全面和深入的医疗建议。

```
from langchain.memory import ConversationKGMemory
from langchain.llms import OpenAI
llm = OpenAI(temperature=0)
memory = ConversationKGMemory(llm=llm)
memory.save_context({"input": "小李是程序员"}, {"output": "知道了,小李是
    程序员"})
memory.save_context({"input": "莫尔索是小李的笔名"}, {"output": "明白,莫
    尔索是小李的笔名"})
```

```
variables = memory.load_memory_variables({"input": "告诉我关于小李的信息"
    })
print(variables)
# 输出
# {'history': 'On 小李: 小李 is 程序员. 小李 的笔名 莫尔索.'}
```

12.2.5　对历史对话进行阶段性总结摘要

应用场景：以教育辅导场景为例

在一系列的教育辅导对话中，学生可能会提出不同的数学问题或理解难题（如” 我不太理解二次方程的求解方法”）。ConversationSummaryMemory 可以帮助 AI 总结之前的辅导内容和学生的疑问点，以便在随后的辅导中提供更针对性的解释和练习。

12.2.6　需要获取最新对话，又要兼顾较早历史对话

应用场景：以技术支持场景为例

在处理一个长期的技术问题时（如软件故障排查），用户可能会在多次对话中提供不同的错误信息和反馈。ConversationSummaryBufferMemory 可以帮助 AI 保留最近几次交互的详细信息，同时提供历史问题处理的摘要，以便于更有效地识别和解决问题。

12.2.7　回溯最近和最关键的对话信息

应用场景：以金融咨询场景为例

在金融咨询聊天机器人中，客户可能会提出多个问题，涉及投资、市场动态或个人财务规划（如” 我想了解股市最近的趋势以及如何分配我的投资组合”）。ConversationTokenBuffer-Memory 可以帮助 AI 聚焦于最近和最关键的几个问题，同时避免由于记忆过多而导致的信息混淆。

12.2.8　基于向量检索对话信息

应用场景：以了解最新新闻事件为例

用户可能会对特定新闻事件提出问题，如” 最近的经济峰会有什么重要决策？” Vector-StoreRetrieverMemory 能够快速从大量历史新闻数据中检索出与当前问题最相关的信息，即使这些信息在整个对话历史中不是最新的，也能提供及时准确的背景信息和详细报道。

```
vectorstore = Chroma(embedding_function=OpenAIEmbeddings())
retriever = vectorstore.as_retriever(search_kwargs=dict(k=1))
memory = VectorStoreRetrieverMemory(retriever=retriever)

```

```
memory.save_context({"input": "我喜欢吃火锅"}, {"output": "听起来很好吃"})
memory.save_context({"input": "我不喜欢看摔跤比赛"}, {"output": "我也是"})

PROMPT_TEMPLATE = """
以下是人类和AI之间的友好对话。AI话语多且提供了许多来自其上下文的具体细节。如果AI不知道问题的答案，它会诚实地说不知道。

以前对话的相关片段：
{history}
(如果不相关，你不需要使用这些信息)

当前对话：
人类：{input}
AI：
"""

prompt = PromptTemplate(input_variables=["history", "input"], template=PROMPT_TEMPLATE)
conversation_with_summary = ConversationChain(
    llm=llm,
    prompt=prompt,
    memory=memory,
    verbose=True
)

print(conversation_with_summary.predict(input="你好，我是莫尔索，你叫什么"))
print(conversation_with_summary.predict(input="我喜欢的食物是什么?"))
print(conversation_with_summary.predict(input="我提到了哪些运动?"))
```

12.3　总结

这 8 种记忆优化方式各有其适用的场景和特点：

- **全量历史对话**：适用于需要完整上下文记忆的客服场景
- **滑动窗口**：适用于关注最近对话的电商咨询场景
- **实体信息提取**：适用于需要记忆关键实体的法律咨询场景

- **知识图谱**：适用于复杂关系建模的医疗咨询场景
- **阶段性总结**：适用于长期教育辅导场景
- **摘要缓冲区**：适用于技术支持类长期问题跟踪
- **令牌缓冲区**：适用于金融咨询等需要关注关键信息的场景
- **向量检索**：适用于需要从大量历史数据中检索相关信息的新闻查询场景

在实际应用中，可以根据具体的业务需求和对话特点选择合适的记忆策略，或者组合使用多种策略来达到最佳的记忆效果。

第十三章　基于 LangChain RAG 问答应用实战

13.1　前言

13.1.1　项目介绍

本次选用百度百科藜藜麦数据 (https://baike.baidu.com/item/藜藜麦/5843874) 模拟人或企业私域数据，并基于 LangChain 开发框架，实现一种简单的 RAG 问答应用示例。

13.1.2　软件资源

- CUDA 11.7
- Python 3.10
- PyTorch 1.13.1+cu117
- LangChain

13.2　环境搭建

13.2.1　环境配置

```
# 创建新环境
$ conda create -n py310_chat python=3.10

# 激活环境
$ source activate py310_chat
```

13.2.2　安装依赖

```
$ pip install datasets langchain sentence_transformers tqdm chromadb
   langchain_wenxin
```

13.3　RAG 问答应用实战

13.3.1　数据构建

藜藜麦数据从百度百科获取并保存到藜藜.txt 文件中。

13.3.2　本地数据加载

```
from langchain.document_loaders import TextLoader

loader = TextLoader("./藜藜.txt")
documents = loader.load()
documents
```

13.3.3　文档分割

采用固定字符长度分割，chunk_size=128

```
# 文档分割
from langchain.text_splitter import CharacterTextSplitter

# 创建拆分器
text_splitter = CharacterTextSplitter(chunk_size=128, chunk_overlap=0)

# 拆分文档
documents = text_splitter.split_documents(documents)
documents
```

分割后的文档示例：

```
[Document(page_content='藜藜(读音li)麦(Chenopodium quinoa Willd.)是藜藜科藜藜属植物...',
        metadata={'source': './藜藜.txt'}),
 Document(page_content='藜藜麦是印第安人的传统主食，几乎和水稻同时被驯服有着6000多年的种植和食用历史...',
        metadata={'source': './藜藜.txt'}),
 Document(page_content='繁殖\n地块选择:应选择地势较高、阳光充足、通风条件好及肥力较好的地块种植...',
        metadata={'source': './藜藜.txt'})]
```

13.3.4　向量化与数据入库

选用 m3e-base 作为 embedding 模型，向量数据库选用 Chroma

```
from langchain.embeddings import HuggingFaceBgeEmbeddings
from langchain.vectorstores import Chroma

# embedding model: m3e-base
model_name = "moka-ai/m3e-base"
model_kwargs = {'device': 'cpu'}
encode_kwargs = {'normalize_embeddings': True}
embedding = HuggingFaceBgeEmbeddings(
    model_name=model_name,
    model_kwargs=model_kwargs,
    encode_kwargs=encode_kwargs,
    query_instruction="为文本生成向量表示用于文本检索"
)

# load data to Chroma db
db = Chroma.from_documents(documents, embedding)

# similarity search
db.similarity_search("藜藜一般在几月播种?")
```

13.3.5　Prompt 设计

```
template = '''
[任务描述]
请根据用户输入的上下文回答问题，并遵守回答要求。

[背景知识]
{{context}}

[回答要求]
- 你需要严格根据背景知识的内容回答，禁止根据常识和已知信息回答问题。
- 对于不知道的信息，直接回答"未找到相关答案"

{question}
'''
```

13.3.6 RetrievalQAChain 构建

采用 ConversationalRetrievalChain，提供历史聊天记录组件

```
from langchain import LLMChain
from langchain_wenxin.llms import Wenxin
from langchain.prompts import PromptTemplate
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationalRetrievalChain

# LLM选型
llm = Wenxin(model="ernie-bot",
             baidu_api_key="baidu_api_key",
             baidu_secret_key="baidu_secret_key")

retriever = db.as_retriever()
memory = ConversationBufferMemory(memory_key="chat_history",
   return_messages=True)

qa = ConversationalRetrievalChain.from_llm(llm, retriever, memory=memory)
qa({"question": "藜藜怎么防治虫害?"})
```

运行结果：

```
{'question': '藜藜怎么防治虫害?',
 'chat_history': [HumanMessage(content='藜藜怎么防治虫害?'),
                  AIMessage(content='藜藜麦常见虫害有象甲虫、金针虫、蝼蝼
                     蛄蛄、黄条跳甲...')],
 'answer': '藜藜麦常见虫害有象甲虫、金针虫、蝼蝼蛄蛄、黄条跳甲、横纹菜蝽
    蝽...'}
```

13.3.7 高级用法

针对多轮对话场景，增加 question_generator 对历史对话记录进行压缩生成新的 question，增加 combine_docs_chain 对检索得到的文本进一步融合

```
from langchain import LLMChain
from langchain.prompts import PromptTemplate
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationalRetrievalChain,
   StuffDocumentsChain
from langchain.chains.qa_with_sources import load_qa_with_sources_chain
```

```
from langchain.prompts.chat import ChatPromptTemplate,
    SystemMessagePromptTemplate, HumanMessagePromptTemplate

# 构建初始messages列表
messages = [
    SystemMessagePromptTemplate.from_template(qa_template),
    HumanMessagePromptTemplate.from_template('{question}')
]

# 初始化prompt对象
prompt = ChatPromptTemplate.from_messages(messages)

llm_chain = LLMChain(llm=llm, prompt=prompt)

combine_docs_chain = StuffDocumentsChain(
    llm_chain=llm_chain,
    document_separator="\n\n",
    document_variable_name="context"
)

q_gen_chain = LLMChain(llm=llm)

qa = ConversationalRetrievalChain(
    combine_docs_chain=combine_docs_chain,
    question_generator=q_gen_chain,
    return_source_documents=True,
    return_generated_question=True,
    retriever=retriever
)

print(qa({'question': "藜藜麦怎么防治虫害?", "chat_history": []}))
```

高级用法运行结果：

```
{'question': '藜藜怎么防治虫害?',
 'chat_history': [],
 'answer': '根据背景知识,藜藜麦常见虫害有象甲虫、金针虫、蝼蝼蛄蛄、黄条跳
    甲...',
 'source_documents': [Document(page_content='病害:主要防治叶斑病...',
                               metadata={'source': './藜藜.txt'})],
 'generated_question': '藜藜怎么防治虫害?'}
```

13.4　技术要点总结

13.4.1　核心组件

- **文档加载器**：TextLoader 用于加载本地文本文件
- **文本分割器**：CharacterTextSplitter 用于将长文本分割为小块
- **嵌入模型**：HuggingFaceBgeEmbeddings 用于生成文本向量表示
- **向量数据库**：Chroma 用于存储和检索向量数据
- **对话链**：ConversationalRetrievalChain 用于处理多轮对话

13.4.2　优化建议

- 根据具体业务场景调整 chunk_size 和 chunk_overlap 参数
- 选择合适的 embedding 模型以获得更好的检索效果
- 针对具体场景优化 prompt 模板
- 考虑使用更复杂的内存管理策略处理长对话历史

13.4.3　扩展应用

- 可以扩展到处理 PDF、Word 等格式的文档
- 可以集成多种向量数据库（如 Pinecone、Weaviate 等）
- 可以结合多种 LLM 提供商（如 OpenAI、Claude 等）
- 可以添加更复杂的检索策略（如混合检索、重排序等）

第十四章　基于 LLM+ 向量库的文档对话经验面

14.1　基础理论

14.1.1　为什么大模型需要外挂 (向量) 知识库？

如何将外部知识注入大模型，最直接的方法：利用外部知识对大模型进行微调

思路：构建几十万量级的数据，然后利用这些数据对大模型进行微调，以将额外知识注入大模型

优点：简单粗暴

缺点：

- 这几十万量级的数据并不能很好的将额外知识注入大模型
- 训练成本昂贵。不仅需要多卡并行，还需要训练很多天

既然大模型微调不是将外部知识注入大模型的最优方案，那是否有其它可行方案？

14.1.2　基于 LLM+ 向量库的文档对话思路

1. 加载文件
2. 读取文本
3. 文本分割
4. 文本向量化
5. 问句向量化
6. 在文本向量中匹配出与问句向量最相似的 topk 个
7. 匹配出的文本作为上下文和问题一起添加到 prompt 中
8. 提交给 LLM 生成回答

14.1.3　核心技术：Embedding

基于 LLM+ 向量库的文档对话核心技术：embedding

思路：将用户知识库内容经过 embedding 存入向量知识库，然后用户每一次提问也会经

过 embedding，利用向量相关性算法（例如余弦算法）找到最匹配的几个知识库片段，将这些知识库片段作为上下文，与用户问题一起作为 prompt 提交给 LLM 回答

14.1.4　Prompt 模板构建

已知信息：
{context}

根据上述已知信息，简洁和专业的来回答用户的问题。如果无法从中得到答案，请说"根据已知信息无法回答该问题"或"没有提供足够的相关信息"，不允许在答案中添加编造成分，答案请使用中文。

问题是：{question}

14.2　优化问题与解决方案

14.2.1　痛点 1：文档切分粒度不好把控

问题描述：既担心噪声太多又担心语义信息丢失

问题 1：如何让 LLM 简要、准确回答细粒度知识？

问题 2：如何让 LLM 回答出全面的粗粒度（跨段落）知识？

解决方案思想：基于 LLM 的文档对话架构分为两部分，先检索，后推理。重心在检索（推荐系统），推理交给 LLM 整合即可。

检索部分要满足三点：

- 尽可能提高召回率
- 尽可能减少无关信息
- 速度快

将所有的文本组织成二级索引，第一级索引是 [关键信息]，第二级是 [原始文本]，二者一一映射。

检索部分只对关键信息做 embedding，参与相似度计算，把召回结果映射的原始文本交给 LLM。

如何构建关键信息？

文章的切分及关键信息抽取：

- **关键信息：**为各语义段的关键信息集合，或者是各个子标题语义扩充之后的集合
- **语义切分方法 1：**利用 NLP 的篇章分析工具，提取出段落之间的主要关系
- **语义切分方法 2：**利用 BERT 等模型判断相邻段落相似度

```
def is_nextsent(sent, next_sent):
    encoding = tokenizer(sent, next_sent, return_tensors="pt",
                         truncation=True, padding=False)
    with torch.no_grad():
        outputs = model(**encoding, labels=torch.LongTensor([1]))
        logits = outputs.logits
        probs = torch.softmax(logits/TEMPERATURE, dim=1)
        next_sentence_prob = probs[:, 0].item()
        if next_sentence_prob <= MERGE_RATIO:
            return False
        else:
            return True
```

语义段的切分及段落关键信息抽取：

- **方法 1：**利用成分句法分析工具和命名实体识别工具提取
- **方法 2：**用语义角色标注分析句子的谓词论元结构
- **方法 3：**关键词提取工具（HanLP、KeyBERT）
- **方法 4：**训练生成关键词的模型（如 ChatLaw 的 KeyLLM）

14.2.2　痛点 2：在垂直领域表现不佳

解决方案：模型微调

- 对 embedding 模型基于垂直领域的数据进行微调
- 对 LLM 模型基于垂直领域的数据进行微调

14.2.3　痛点 3：LangChain 内置问答分句效果不佳

文档加工方案：

- 使用更好的文档拆分方式（如达摩院语义识别模型）
- 改进填充方式，仅添加相关度高的上下文句子
- 对每段分别进行总结，基于总结内容进行语义匹配

14.2.4　痛点 4：如何尽可能召回与 query 相关的 Document

解决方法：

- 优化 Document 的长度、embedding 质量和召回数量之间的平衡
- 使用 Faiss 搜索，基于本地知识对文本向量化工具进行 Finetune
- 将 ES 搜索结果与 Faiss 结果相结合

14.2.5　痛点 5：如何让 LLM 基于 query 和 context 得到高质量的 response

解决方法：

- 尝试多个 prompt 模板，选择最合适的
- 用与本地知识问答相关的语料对 LLM 进行 Finetune

14.2.6　痛点 6：Embedding 模型在表示 text chunks 时偏差太大

问题描述：

- 开源 embedding 模型效果一般，text chunk 大时表示不准确
- 多语言对齐问题（英文内容，中文 query）

解决方法：

- 使用更小的 text chunk 配合更大的 topk
- 寻找更适合多语言的 embedding 模型

14.2.7　痛点 7：不同的 prompt 产生完全不同的效果

问题描述：prompt 的提法不同会产生完全不同的效果，特别是输出格式要求

14.2.8　痛点 8：LLM 生成效果问题

问题描述：不同 LLM 在理解 context 和生成环节表现差异大
解决思路：选择开源模型（如 llama2、baichuan2），构造 domain dataset 进行微调

14.2.9　痛点 9：如何更高质量地召回 context 喂给 LLM

问题描述：召回内容与 query 相关性差
解决思路：更细颗粒度的 recall，针对性的 pdf 解析

14.3　工程实践与避坑指南

14.3.1　本地知识库问答系统（Langchain-chatGLM）

环境配置问题解决

```
# 解决持续网页loading问题
$ pip install gradio==3.21.0

# 解决detectron2安装问题
```

```
$ cd detectron2
$ pip install -e .
$ pip install torch==2.0.0
$ pip install protobuf==3.20.0
```

PDF 加载问题解决

- 更新 apt 包：sudo apt update
- 安装依赖：sudo apt install libmagic-dev poppler-utils tesseract-ocr
- 配置中文识别包

NLTK 数据包问题解决

- 手动解压 punkt 和 tagger 到指定目录
- 通过 nltk.data.path 查询存储路径

PaddleOCR 错误解决

错误：ModuleNotFoundError: No module named 'tools.infer'
解决：将所有 from tools.infer import 改为 from paddleocr.tools.infer import

MOSS 模型加载错误解决

错误：get_class_from_dynamic_module() missing 2 required positional arguments
修改方案：

```
def auto_configure_device_map() -> Dict[str, int]:
    cls = get_class_from_dynamic_module(
        pretrained_model_name_or_path="fnlp/moss-moon-003-sft",
        module_file="modeling_moss.py",
        class_name="MossForCausalLM"
    )
```

MOSS 提问错误解决

错误：RuntimeError: probability tensor contains either inf, nan or element < 0
解决：移除 do_sample=True 参数

14.4　技术要点总结

14.4.1　核心架构设计

- **二级索引系统**：关键信息索引 + 原始文本映射
- **语义分割策略**：基于篇章分析和 BERT 相似度的混合方法
- **检索优化**：平衡召回率、准确性和效率

14.4.2　关键优化建议

- **文档预处理**：采用语义级别的分割而非简单的格式分割
- **Embedding 选择**：根据语言和领域特点选择合适的模型
- **Prompt 工程**：针对具体任务设计合适的模板
- **模型微调**：在垂直领域进行针对性的模型优化

14.4.3　工程实践建议

- **版本兼容性**：注意各组件版本匹配问题
- **错误处理**：建立完善的错误监控和处理机制
- **性能优化**：针对大规模文档建立分级索引系统
- **多语言支持**：考虑跨语言检索和生成的需求

第十五章　大模型 RAG 经验面

15.1　LLMs 的不足与挑战

15.1.1　LLMs 存在的不足点

在 LLM 已经具备了较强能力的基础上，仍然存在以下问题：

- **幻觉问题**：LLM 文本生成的底层原理是基于概率的 token by token 的形式，因此会不可避免地产生” 一本正经的胡说八道” 的情况
- **时效性问题**：LLM 的规模越大，大模型训练的成本越高，周期也就越长。那么具有时效性的数据也就无法参与训练，所以也就无法直接回答时效性相关的问题，例如” 帮我推荐几部热映的电影?”
- **数据安全问题**：通用的 LLM 没有企业内部数据和用户数据，那么企业想要在保证安全的前提下使用 LLM，最好的方式就是把数据全部放在本地，企业数据的业务计算全部在本地完成。而在线的大模型仅仅完成一个归纳的功能

15.2　RAG 技术概述

15.2.1　什么是 RAG?

RAG（Retrieval Augmented Generation，检索增强生成），即 LLM 在回答问题或生成文本时，先会从大量文档中检索出相关的信息，然后基于这些信息生成回答或文本，从而提高预测质量。

15.2.2　RAG 核心组件

检索器模块（R）

在 RAG 技术中，”R” 代表检索，其作用是从大量知识库中检索出最相关的前 k 个文档。构建高质量的检索器面临三个关键挑战：

2.1.1 如何获得准确的语义表示？在 RAG 中，语义空间指的是查询和文档被映射的多维空间。构建准确语义空间的方法：

- **块优化**：处理外部文档的第一步是分块，以获得更细致的特征。选择分块策略时需要考虑被索引内容的特点、使用的嵌入模型及其最适块大小、用户查询的预期长度和复杂度
- **微调嵌入模型**：在确定 Chunk 的适当大小后，通过嵌入模型将 Chunk 和查询嵌入。优秀的嵌入模型如 UAE、Voyage、BGE 等，它们在大规模语料库上预训练过

2.1.2 如何协调查询和文档的语义空间？协调用户的查询与文档的语义空间的技术：

- **查询重写**：利用大语言模型的能力生成指导性伪文档，或将原始查询与伪文档结合形成新查询。多查询检索方法让大语言模型能够同时产生多个搜索查询
- **嵌入变换**：通过在查询编码器后加入特殊适配器并微调，优化查询的嵌入表示。SANTA 方法让检索系统能够理解并处理结构化的信息

2.1.3 如何对齐检索模型的输出和大语言模型的偏好？对齐方法：

- **大语言模型的监督训练**：REPLUG 使用检索模型和大语言模型计算检索到的文档的概率分布，然后通过计算 KL 散度进行监督训练
- **适配器附加**：在检索模型上外部附加适配器来实现对齐，避免微调嵌入模型的挑战
- **指令微调**：PKG 通过指令微调将知识注入到白盒模型中，直接替换检索模块

生成器模块（G）

2.2.1 生成器介绍

- **作用**：将检索到的信息转化为自然流畅的文本。输入不仅包括传统的上下文信息，还有通过检索器得到的相关文本片段
- **特点**：能够更深入地理解问题背后的上下文，并产生更加信息丰富的回答。根据检索到的文本来指导内容的生成，确保一致性

2.2.2 后检索处理提升策略

- **目的**：提高检索结果的质量，更好地满足用户需求或为后续任务做准备
- **策略**：包括信息压缩和结果的重新排序

2.2.3 生成器优化方法

- **优化目的**：确保生成文本既流畅又能有效利用检索文档，更好地回应用户的查询
- **方法**：对检索器找到的文档进行后续处理，微调方式与大语言模型的普通微调方法大体相同

15.3　RAG 的优势

使用 RAG 的好处包括：

- **可扩展性**：减少模型大小和训练成本，允许轻松扩展知识
- **准确性**：通过引用信息来源，用户可以核实答案的准确性，增强对模型输出结果的信任
- **可控性**：允许更新或定制知识
- **可解释性**：检索到的项目作为模型预测中来源的参考

- **多功能性**：可以针对多种任务进行微调和定制，包括 QA、文本摘要、对话系统等
- **及时性**：使用检索技术能识别到最新的信息，保持回答的及时性和准确性
- **定制性**：通过索引与特定领域相关的文本语料库，为不同领域提供专业的知识支持
- **安全性**：通过数据库中设置的角色和安全控制，实现对数据使用的更好控制

15.4　RAG 与 SFT 对比

表 15.1: RAG 与 SFT 对比分析

维度	RAG	SFT
数据	动态数据。RAG 不断查询外部源，确保信息保持最新，而无需频繁的模型重新训练	（相对）静态数据，并且在动态数据场景中可能很快就会过时。SFT 也不能保证记住这些知识
外部知识	RAG 擅长利用外部资源。通过在生成响应之前从知识源检索相关信息来增强 LLM 能力。它非常适合文档或其他结构化/非结构化数据库	SFT 可以对 LLM 进行微调以对齐预训练学到的外部知识，但对于频繁更改的数据源来说可能不太实用
模型定制	RAG 主要关注信息检索，擅长整合外部知识，但可能无法完全定制模型的行为或写作风格	SFT 允许根据特定的语气或术语调整 LLM 的行为、写作风格或特定领域的知识
减少幻觉	RAG 本质上不太容易产生幻觉，因为每个回答都建立在检索到的证据上	SFT 可以通过将模型基于特定领域的训练数据来帮助减少幻觉。但当面对不熟悉的输入时，它仍然可能产生幻觉
透明度	RAG 系统通过将响应生成分解为不同的阶段来提供透明度，提供对数据检索的匹配度以提高对输出的信任	SFT 就像一个黑匣子，使得响应背后的推理更加不透明
技术专长	RAG 需要高效的检索策略和大型数据库相关技术。另外还需要保持外部数据源集成以及数据更新	SFT 需要准备和整理高质量的训练数据集、定义微调目标以及相应的计算资源

两种方法并非非此即彼，合理的方式是结合业务需要与两种方法的优点，合理使用两种方法。

15.5　RAG 典型实现方法

RAG 的实现主要包括三个主要步骤：数据索引、检索和生成。

15.5.1　数据索引构建

数据索引是一个离线的过程，主要是将私域数据向量化后构建索引并存入数据库的过程。

Step1：数据提取

- **数据获取：**包括多格式数据（PDF、word、markdown 以及数据库和 API 等）加载、不同数据源获取等
- **Doc 类文档：**直接解析得到文本元素及其属性，用于后续切分的依据
- **PDF 类文档：**使用多个开源模型进行协同分析，如版面分析使用百度的 PP-StructureV2
- **PPT 类文档：**将 PPT 转换成 PDF 形式，然后用处理 PDF 的方式来进行解析
- **数据清洗：**对源数据进行去重、过滤、压缩和格式化等处理
- **信息提取：**提取数据中关键信息，包括文件名、时间、章节 title、图片等信息

Step2：文本分割（Chunking）

- **动机：**由于文本可能较长，或者仅有部分内容相关的情况下，需要对文本进行分块切分
- **考虑因素：**embedding 模型的 Tokens 限制情况；语义完整性对整体的检索效果的影响
- **分块方式：**
  - 句分割：以” 句” 的粒度进行切分，保留一个句子的完整语义
  - 固定大小的分块方式：根据 embedding 模型的 token 长度限制，将文本分割为固定长度
  - 基于意图的分块方式：句分割、递归分割、特殊分割
- **常用工具：**langchain.text_splitter 库中的 CharacterTextSplitter 类

Step3：向量化及创建索引

- **向量化：**将文本、图像、音频和视频等转化为向量矩阵的过程
- **常见 embedding 模型：**ChatGPT-Embedding、ERNIE-Embedding V1、M3E、BGE
- **创建索引：**数据向量化后构建索引，并写入数据库的过程
- **常用工具：**FAISS、Chromadb、ES、milvus 等
- **选择考虑：**根据业务场景、硬件、性能需求等多因素综合考虑

15.5.2　数据检索策略

检索思路：

- **元数据过滤：**通过元数据先进行过滤，提升效率和相关度
- **图关系检索：**引入知识图谱，利用知识之间的关系做更准确的回答
- **检索技术：**
  - 向量化相似度检索：使用欧氏距离、曼哈顿距离、余弦等计算方式

  - 关键词检索：传统检索方式，元数据过滤也是一种
  - 全文检索、SQL 检索：传统检索算法
- **重排序**：根据相关度、匹配度等因素重新调整，得到更符合业务场景的排序
- **查询轮换**：
  - 子查询：使用各种查询策略，如树查询、向量查询、顺序查询 chunks 等
  - HyDE：生成相似的或更标准的 prompt 模板

15.5.3　文本生成与回复

文本生成就是将原始 query 和检索得到的文本组合起来输入模型得到结果的过程，本质上就是 prompt engineering 过程。

```
from langchain.chat_models import ChatOpenAI
from langchain.schema.runnable import RunnablePassthrough

llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
rag_chain = {"context": retriever, "question": RunnablePassthrough()} |
   rag_prompt | llm
rag_chain.invoke("What is Task Decomposition?")
```

全流程框架如 Langchain 和 LlamaIndex，都非常简单易用。

15.6　RAG 典型案例

15.6.1　ChatPDF 及其复刻版

ChatPDF 的实现流程：

1. 读取 PDF 文件，转换为可处理的文本格式（如 txt 格式）
2. 对提取出来的文本进行清理和标准化（去除特殊字符、分段、分句等）
3. 使用 OpenAI 的 Embeddings API 将每个分段转换为向量
4. 将用户问题转换为向量，并与每个分段的向量进行比较，找到最相似的分段
5. 将最相似的分段与问题作为 prompt，调用 OpenAI 的 Completion API
6. 将 ChatGPT 生成的答案返回给用户

15.6.2　Baichuan 搜索增强系统

百川大模型的搜索增强系统融合模块：

- **指令意图理解**：深入理解用户指令
- **智能搜索**：精确驱动查询词的搜索
- **结果增强**：结合大语言模型技术来优化模型结果生成的可靠性

通过这一系列协同作用，实现更精确、智能的模型结果回答，减少模型的幻觉。

15.6.3　多模态检索增强模型

RA-CM3 是一个检索增强的多模态模型：

- 使用预训练的 CLIP 模型实现检索器（retriever）
- 使用 CM3 Transformer 架构构成生成器（generator）
- 检索器辅助模型从外部存储库中搜索有关提示文本的精确信息
- 将该信息连同文本送入生成器中进行图像合成
- 设计的模型的准确性大大提高

15.7　RAG 存在的问题与挑战

RAG 技术目前存在以下问题：

- **检索效果依赖**：检索效果依赖 embedding 和检索算法。目前可能检索到无关信息，反而对输出有负面影响
- **黑盒利用**：大模型如何利用检索到的信息仍是黑盒的。可能仍存在不准确（甚至生成的文本与检索信息相冲突）
- **效率问题**：对所有任务都无差别检索 k 个文本片段，效率不高，同时会大大增加模型输入的长度
- **引用和验证困难**：无法引用来源，也因此无法精准地查证事实，检索的真实性取决于数据源及检索算法

第十六章　LLM 文档对话 PDF 解析关键问题

16.1　PDF 解析的必要性

16.1.1　为什么需要进行 PDF 解析？

最近在探索 ChatPDF 和 ChatDoc 等方案的思路，也就是用 LLM 实现文档助手。在此记录一些难题和解决方案，首先讲解主要思想，其次以问题 + 回答的形式展开。

16.1.2　PDF 解析的重要性

当利用 LLMs 实现用户与文档对话时，首要工作就是对文档中内容进行解析。

由于 PDF 是最通用，也是最复杂的文档形式，所以对 PDF 进行解析变成利用 LLM 实现用户与文档对话的重中之重工作。

如何精确地回答用户关于文档的问题，不重也不漏？笔者认为非常重要的一点是文档内容解析。很好地组织起来，LLM 只能瞎编。

16.2　PDF 解析方法与区别

16.2.1　PDF 解析的两条技术路线

PDF 的解析大体上有两条路，一条是基于规则，一条是基于 AI。

方法一：基于规则：

- **介绍：**根据文档的组织特点去” 算” 每部分的样式和内容
- **存在问题：**不通用，因为 PDF 的类型、排版实在太多了，没办法穷举

方法二：基于 AI：

- **介绍：**该方法为目标检测和 OCR 文字识别 pipeline 方法

16.3 PDF 解析存在的问题

PDF 转 text 这块存在一定的偏差，尤其是 paper 中包含了大量的 figure 和 table，以及一些特殊的字符，直接调用 langchain 官方给的 PDF 解析工具，有一些信息甚至是错误的。

这里，一方面可以用 arxiv 的 tex 源码直接抽取内容，另一方面，可以尝试用各种 OCR 工具来提升表现。

16.4 长文档关键信息提取方法

对于长文档（书籍），如何获取其中关键信息，并构建索引：

方法一：分块索引法

- **介绍：**直接对长文档（书籍）进行分块，然后构建索引入库。后期问答，只需要从库中召回和用户 query 相关的内容块进行拼接成文章，输入到 LLMs 生成回复
- **存在问题：**
  1. 将文章分块，会破坏文章语义信息
  2. 对于长文章，会被分割成很多块，并构建很多索引，这严重影响知识库存储空间
  3. 如果内容都不能很好地组织起来，LLM 只能瞎编

方法二：文本摘要法

- **介绍：**直接利用文本摘要模型对每一篇长文档（书籍）做文本摘要，然后对文本摘要内容构建索引入库。后期问答，只需要从库中召回和用户 query 相关的摘要内容，输入到 LLMs 生成回复
- **存在问题：**
  1. 由于每篇长文档（书籍）内容比较多，直接利用文本摘要模型对其做文本摘要，需要比较大算力成本和时间成本
  2. 生成的文本摘要存在部分内容丢失问题，不能很好的概括整篇文章

方法三：多级标题构建文本摘要法：

- **介绍：**把多级标题提取出来，然后适当做语义扩充，或者去向量库检索相关片段，最后用 LLM 整合即可

16.5 标题提取的重要性与方法

16.5.1 为什么要提取标题甚至是多级标题？

没有处理过 LLM 文档对话的朋友可能不明白为什么要提取标题甚至是多级标题，因此我先来阐述提取标题对于 LLM 阅读理解的重要性有多大。

1. 如 Q1 阐述的那样，标题是快速做摘要最核心的文本
2. 对于有些问题 high-level 的问题，没有标题很难得到用户满意的结果

举例：假如用户就想知道 3.2 节是从哪些方面讨论的（标准答案就是 3 个方面），如果我们没有将标题信息告诉 LLM，而是把所有信息全部扔给 LLM，那它大概率不会知道是 3 个方面（要么会少，要么会多。做过的朋友秒懂）

16.5.2 如何提取文章标题？

第一步：PDF 转图片。用一些工具将 PDF 转换为图片，这里有很多开源工具可以选，笔者采用 fitz，一个 python 库。速度很快，时间在毫秒之间。

第二步：图片中元素识别。采用目标检测模型识别元素（标题、文本、表格、图片、列表等元素）。

工具介绍：

- **Layout-parser：**
  - 优点：最大的模型（约 800MB）精度非常高
  - 缺点：速度慢一点
- **PaddlePaddle-ppstructure：**
  - 优点：模型比较小，效果也还行
- **unstructured：**
  - 缺点：fast 模式效果很差，基本不能用，会将很多公式也识别为标题。其他模式或许可行，笔者没有尝试

利用上述工具，可以得到一个 list，存储所有检测出来的标题。

第三步：标题级别判断。利用标题区块的高度（也就是字号）来判断哪些是一级标题，哪些是二级、三级、.....N 级标题。这个时候我们发现一些目标检测模型提取的区块并不是严格按照文字的边去切，导致这个 idea 不能实施，那怎么办呢？unstructured 的 fast 模式就是按照文字的边去切的，同一级标题的区块高度误差在 0.001 之间。因此我们只需要用 unstructured 拿到标题的高度值即可（虽然繁琐，但是不耗时，unstructured 处理也在毫秒之间）。

16.6 单双栏 PDF 的处理

16.6.1 区分单双栏 PDF 与重新排序

动机：很多目标检测模型识别区块之后并不是顺序返回的，因此我们需要根据坐标重新组织顺序。单栏的很好办，直接按照中心点纵坐标排序即可。双栏 PDF 就很棘手了，有的朋友可能不知道 PDF 还有双栏形式。

问题一：首先如何区分单双栏论文？

- **方法：**得到所有区块的中心点的横坐标，用这一组横坐标的极差来判断即可，双栏论文的极差远远大于单栏论文，因此可以设定一个极差阈值

问题二：双栏论文如何确定区块的先后顺序？

- **方法**：先找到中线，将左右栏的区块分开，中线横坐标可以借助上述求极差的两个横坐标 x1 和 x2 来求，也就是 (x1+x2)/2。分为左右栏区块后，对于每一栏区块按照纵坐标排序即可，最后将右栏拼接到左栏后边

16.7 表格和图片数据提取

16.7.1 表格和图片数据提取思路

思路仍然是目标检测和 OCR。无论是 layoutparser 还是 PaddleOCR 都有识别表格和图片的目标检测模型，而表格的数据可以直接 OCR 导出为 excel 形式数据，非常方便。

提取出表格之后喂给 LLM，LLM 还是可以看懂的，可以设计 prompt 做一些指导。关于这一块两部分 demo 代码都很清楚明白，这里不再赘述。

16.8 基于 AI 的文档解析优缺点

16.8.1 基于 AI 的文档解析优缺点分析

- **优点**：准确率高，通用性强
- **缺点**：耗时慢，建议用 GPU 等加速设备，多进程、多线程去处理。耗时只在目标检测和 OCR 两个阶段，其他步骤均不耗时

16.9 总结与建议

16.9.1 技术建议

笔者建议按照不同类型的 PDF 做特定处理，例如论文、图书、财务报表、PPT 都可以根据特点做一些小的专有设计。

没有 GPU 的话目标检测模型建议用 PaddlePaddle 提供的，速度很快。Layout parser 只是一个框架，目标检测模型和 OCR 工具可以自有切换。

16.9.2 实践要点总结

- **预处理优化**：根据文档类型选择合适的解析策略
- **标题提取**：多级标题提取对于 LLM 理解文档结构至关重要
- **布局处理**：单双栏识别和重新排序是保证内容连贯性的关键
- **表格处理**：结合目标检测和 OCR 技术提取结构化数据
- **性能平衡**：在准确性和处理速度之间找到合适的平衡点

16.9.3　未来发展方向

- 更智能的文档结构理解算法
- 多模态信息的融合处理
- 实时处理性能的优化
- 领域自适应能力的提升

第十七章　大模型 (LLMs)RAG 版面分析表格识别方法篇

17.1　表格识别的必要性

17.1.1　为什么需要识别表格？

表格的尺寸、类型和样式展现出多样化的特征，如背景填充的差异性、行列合并方法的多样性以及内容文本类型的不一致性等。同时，现有的文档资料不仅涵盖了现代电子文档，也包括历史的手写扫描文档，这些文档在样式设计、光照条件以及纹理特性等方面存在显著差异。因此，表格识别一直是文档识别领域的重大挑战。

表格类型示例包括：

- 有颜色背景的全线表
- 少线表
- 无线表
- 有复杂表格线条样式的表格
- 拍照得到的手写历史文档

17.2　表格识别任务概述

17.2.1　表格识别任务定义

表格识别包括表格检测和表格结构识别两个子任务。

表格识别过程可细分为两个关键步骤：

表格定位（Table Localization）：

- 涉及识别并划定表格的整体边界
- 采用的技术手段包括目标检测算法，如 YOLO、Faster RCNN 或 Mask RCNN
- 有时借助生成对抗网络（GAN）来精确勾勒表格的外在轮廓

表格元素解析与结构重建（Table Element Parsing and Structure Reconstruction）：

- **表格单元格划分（Cell Detection）**：识别和区分表格内部的各个单元格

- **表格结构理解（Table Structure Understanding）**：分析表格区域以提取数据内容及其内在逻辑关系

17.3　表格识别方法分类

17.3.1　传统方法

利用规则指导和图像处理技术，执行以下步骤识别结构：

1. 应用腐蚀与膨胀算法来细化和增强目标区域边界特征
2. 通过分析像素连通性，确定并标记图像中的各个显著区域
3. 实施线段检测和直线拟合技术，精确描绘图像内的线性结构元素
4. 计算线性结构之间的交点，构建可能的边框或连接关系网络
5. 合并初步检测到的边界框（猜测框），运用智能合并策略减少冗余并提高精度
6. 根据尺寸筛选优化，剔除不符合预期大小条件的候选区域

17.3.2　pdfplumber 表格抽取

pdfplumber 表格抽取原理

1. 找到可见的或猜测出不可见的候选表格线
2. 根据候选表格线确定它们的交点，找到围成的最小单元格
3. 把连通的单元格整合到一起，生成检测出的表格对象

pdfplumber 常见的表格抽取模式

lattice 抽取线框类的表格：

1. 把 PDF 页面转换成图像
2. 通过图像处理检测出水平方向和竖直方向的直线
3. 根据检测出的直线生成可能表格的 bounding box
4. 确定表格各行、列的区域
5. 解析表格结构，填充单元格内容，形成表格对象

stream 抽取非线框类的表格：

1. 通过 pdfminer 获取连续字符串（串行）
2. 通过文本对齐的方式确定可能表格的 bounding box（文本块）
3. 确定表格各行、列的区域
4. 解析表格结构，填充单元格内容，形成表格对象

17.3.3　深度学习方法-语义分割

table-ocr/table-detect

- **table-ocr**：运用 unet 实现对文档表格的自动检测和表格重建
- **table-detect**：使用 YOLO 进行表格检测，unet 进行表格单元格定位

腾讯表格图像识别

- **思路**：图像分割，分割类别为 4 类（横向线、竖向线、横向不可见线、竖向不可见线）
- **模型**：对比 DeepLab 系列、FCN、Unet、SegNet 等，Unet 收敛最快

TableNet

- **论文**：《TableNet: Deep Learning Model for End-to-end Table Detection and Tabular Data Extraction from Scanned Document Images》
- **架构**：基于编码器-解码器模型，使用预训练 VGG-19 网络
- **数据集**：马莫特数据集（包含中文页面）
- **效果**：微调后模型的召回率 0.9628、精度 0.9697、F1 得分 0.9662

CascadeTabNet

- **方法**：基于端到端深度学习，使用级联掩码 R-CNN HRNet 模型
- **优点**：
  1. 提出级联网络进行表检测和结构识别
  2. 端到端解决表格检测和识别两个子任务
  3. 用实例分割提高表检测精度
  4. 采用两阶段迁移学习策略，适用小数据集

SPLERGE

- **论文名称**：Deep Splitting and Merging for Table Structure Decomposition
- **思想**：先自顶向下、再自底向上的两阶段表格结构识别方法
- **流程**：
  - Split 部分：把表格区域分割成网格状结构
  - Merge 部分：对 Split 结果中的邻接网格对进行合并预测

DeepDeSRT

- **论文名称**：DeepDeSRT: Deep Learning for Detection and Structure Recognition of Tables in Document Images
- **思路**：提供基于深度学习的表格检测和表结构识别解决方案

- **结构：**
  - 表格检测：使用快速 RCNN 作为基本框架
  - 结构识别：使用全连接网络与 VGG-16 权重提取行列信息
- **数据集：**ICDAR 2013 表竞争数据集

17.4　方法比较与应用建议

17.4.1　各类方法优缺点比较

- **传统方法：**计算量小，但对复杂表格效果有限
- **pdfplumber：**适合规则表格，对扫描文档效果一般
- **深度学习方法：**准确率高，但需要大量标注数据和计算资源

17.4.2　实际应用建议

1. 根据表格复杂程度选择合适的方法
2. 考虑计算资源和时间成本
3. 对于重要应用，建议采用深度学习方法
4. 可以组合使用多种方法提高准确率

17.5　技术挑战与发展趋势

17.5.1　当前主要挑战

- 复杂表格结构的准确识别
- 手写和历史文档的处理
- 多语言表格的识别
- 实时处理性能优化

17.5.2　未来发展趋势

- 更强大的端到端识别模型
- 少样本和零样本学习技术
- 多模态信息融合
- 云端一体化解决方案

第十八章　大模型 (LLMs)RAG 版面分析-文本分块面

18.1　文本分块的必要性

18.1.1　为什么需要对文本分块？

使用大型语言模型 (LLM) 时，切勿忽略文本分块的重要性，其对处理结果的好坏有重大影响。

考虑以下场景：你面临一个几百页的文档，其中充满了文字，你希望对其进行摘录和问答式处理。在这个流程中，最初的一步是提取文档的嵌入向量，但这样做会带来几个问题：

- **信息丢失的风险**：试图一次性提取整个文档的嵌入向量，虽然可以捕捉到整体的上下文，但也可能会忽略掉许多针对特定主题的重要信息，这可能会导致生成的信息不够精确或者有所缺失
- **分块大小的限制**：在使用如 OpenAI 这样的模型时，分块大小是一个关键的限制因素。例如，GPT-4 模型有一个 32K 的窗口大小限制。尽管这个限制在大多数情况下不是问题，但从一开始就考虑到分块大小是很重要的

因此，恰当地实施文本分块不仅能够提升文本的整体品质和可读性，还能够预防由于信息丢失或不当分块引起的问题。这就是为何在处理长篇文档时，采用文本分块而非直接处理整个文档至关重要的原因。

18.2　常见的文本分块方法

18.2.1　一般的文本分块方法

如果不借助任何包，直接按限制长度切分方案：

```
text = "我是一个名为ChatGLM3-6B的人工智能助手，是基于清华大学KEG实验室和智谱AI公司于2023年共同训练的语言模型开发的。我的目标是通过回答用户提出的问题来帮助他们解决问题。由于我是一个计算机程序，所以我没有实际的存在，只能通过互联网来与用户交流。"

```

```
chunks = []
chunk_size = 128
for i in range(0, len(text), chunk_size):
    chunk = text[i:i + chunk_size]
    chunks.append(chunk)

chunks
```

输出结果：

['我是一个名为ChatGLM3-6B的人工智能助手，是基于清华大学KEG实验室和智谱AI公司于2023年共同训练的语言模型开发的。我的目标是通过回答用户提出的问题来帮助他们解决问题。由于我是一个计算机程序，所以我没有实际的存在，只能通过互联网', '来与用户交流。']

18.2.2　正则拆分的文本分块方法

动机：一般的文本分块方法能够按长度进行分割，但是对于一些长度偏长的句子，容易从中间切开

方法：在中文文本分块的场景中，正则表达式可以用来识别中文标点符号，从而将文本拆分成单独的句子。这种方法依赖于中文句号、”问号”、”感叹号” 等标点符号作为句子结束的标志。

特点：虽然这种基于模式匹配的方法可能不如基于复杂语法和语义分析的方法精确，但它在大多数情况下足以满足基本的句子分割需求，并且实现起来更为简单直接。

```
import re

def split_sentences(text):
    # 使用正则表达式匹配中文句子结束的标点符号
    sentence_delimiters = re.compile(u'[。?!;]|\n')
    sentences = sentence_delimiters.split(text)
    # 过滤掉空字符串
    sentences = [s.strip() for s in sentences if s.strip()]
    return sentences

text = "文本分块是自然语言处理(NLP)中的一项关键技术，其作用是将较长的文本切割成更小、更易于处理的片段。这种分割通常是基于单词的词性和语法结构，例如将文本拆分为名词短语、动词短语或其他语义单位。这样做有助于更高效地从文本中提取关键信息。"
sentences = split_sentences(text)
print(sentences)
```

输出结果：

```
['文本分块是自然语言处理(NLP)中的一项关键技术,其作用是将较长的文本切割成更小、更易于处理的片段',
'这种分割通常是基于单词的词性和语法结构,例如将文本拆分为名词短语、动词短语或其他语义单位',
'这样做有助于更高效地从文本中提取关键信息']
```

在上面例子中，我们并没有采用任何特定的方式来分割句子。另外，还有许多其他的文本分块技术可以使用，例如词汇化 (tokenizing)、词性标注 (POS tagging) 等。

18.2.3　Spacy Text Splitter 方法

介绍：Spacy 是一个用于执行自然语言处理 (NLP) 各种任务的库。它具有文本拆分器功能，能够在进行文本分割的同时，保留分割结果的上下文信息。

```
import spacy

input_text = "文本分块是自然语言处理(NLP)中的一项关键技术，其作用是将较长的文本切割成更小、更易于处理的片段。这种分割通常是基于单词的词性和语法结构，例如将文本拆分为名词短语、动词短语或其他语义单位。这样做有助于更高效地从文本中提取关键信息。"
nlp = spacy.load("zh_core_web_sm")
doc = nlp(input_text)
for s in doc.sents:
    print(s)
```

输出结果：

```
文本分块是自然语言处理(NLP)中的一项关键技术，其作用是将较长的文本切割成更小、更易于处理的片段。
这种分割通常是基于单词的词性和语法结构，例如将文本拆分为名词短语、动词短语或其他语义单位。
这样做有助于更高效地从文本中提取关键信息。
```

18.2.4　基于 langchain 的 CharacterTextSplitter 方法

使用 CharacterTextSplitter，一般的设置参数为：chunk_size、chunk_overlap、separator 和 strip_whitespace。

```
from langchain.text_splitter import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
```

```
    chunk_size=35,
    chunk_overlap=0,
    separator='',
    strip_whitespace=False
)
text_splitter.create_documents([text])
```

输出结果：

```
[Document(page_content='我是一个名为ChatGLM3-6B的人工智能助手，是基于清华大学'),
 Document(page_content='KEG实验室和智谱AI公司于2023年共同训练的语言模型开发'),
 Document(page_content='的。我的目标是通过回答用户提出的问题来帮助他们解决问题。由于我是一个计'),
 Document(page_content='算机程序，所以我没有实际的存在，只能通过互联网来与用户交流。')]
```

18.2.5　基于 langchain 的递归字符切分方法

使用 RecursiveCharacterTextSplitter，一般的设置参数为：chunk_size、chunk_overlap。

```
# input text
input_text = "文本分块是自然语言处理(NLP)中的一项关键技术，其作用是将较长的文本切割成更小、更易于处理的片段。这种分割通常是基于单词的词性和语法结构，例如将文本拆分为名词短语、动词短语或其他语义单位。这样做有助于更高效地从文本中提取关键信息。"

from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=100,  # 设置所需的文本大小
    chunk_overlap=20
)
chunks = text_splitter.create_documents([input_text])
print(chunks)
```

输出结果：

```
[Document(page_content='文本分块是自然语言处理(NLP)中的一项关键技术，其作用是将较长的文本切割成更小、更易于处理的片段。这种分割通常是基于单词的词性和语法结构，例如将文本拆分为名词短语、动词短语或其他语义单位。这样做有助'),
```

```
Document(page_content='短语、动词短语或其他语义单位。这样做有助于更高效地从文本中提取关键信息。')]
```

与 CharacterTextSplitter 不同，RecursiveCharacterTextSplitter 不需要设置分隔符，默认的几个分隔符如下：

"\n\n" - 两个换行符，一般认为是段落分隔符
"\n"　- 换行符
" "　- 空格
""　- 字符

拆分器首先查找两个换行符（段落分隔符）。一旦段落被分割，它就会查看块的大小，如果块太大，那么它会被下一个分隔符分割。如果块仍然太大，那么它将移动到下一个块上，以此类推。

18.2.6　HTML 文本拆分方法

介绍：HTML 文本拆分器是一种结构感知的文本分块工具。它能够在 HTML 元素级别上进行文本拆分，并且会为每个分块添加与之相关的标题元数据。

特点：对 HTML 结构的敏感性，能够精准地处理和分析 HTML 文档中的内容。

```
# input html string
html_string = """
<!DOCTYPE html>
<html>
<body>
<div>
<h1>Mobot</h1>
<p>一些关于Mobot的介绍文字。</p>
<div> <h2>Mobot主要部分</h2><p>有关Mobot的一些介绍文本。</p><h3>Mobot第1小节</h3><p>有关Mobot第一个子主题的一些文本。</p><h3>Mobot第2小节</h3><p>关于Mobot的第二个子主题的一些文字。</p></div><div><h2>Mobot</h2><p>关于Mobot的一些文字</p></div><br><p>关于Mobot的一些结论性文字</p></div></body></html>"""

headers_to_split_on = [("h1", "Header 1"), ("h2", "标题2"), ("h3", "标题3")]

from langchain.text_splitter import HTMLHeaderTextSplitter
html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
```

```
html_header_splits = html_splitter.split_text(html_string)
print(html_header_splits)
```

输出结果:

```
[Document(page_content='Mobot'),
 Document(page_content='一些关于Mobot的介绍文字。\nMobot主Mobot第2小节',
    metadata={'Header 1':'Mobot'}),
 Document(page_content='有关Mobot的一些介绍文本。', metadata={'Header
    1':'Mobot','标题 2':'Mobot主要部分'}),
 Document(page_content='有关Mobot第一个子主题的一些文本。', metadata={'
    Header 1':'Mobot','标题 2':'Mobot主要部分','标题 3':'Mobot第1小节'}),
 Document(page_content='关于Mobot的第二个子主题的一些文字。', metadata={'
    Header 1':'Mobot','标题 2':'Mobot主要部分','标题 3':'Mobot第2小节'}),
 Document(page_content='Mobot div>', metadata={'Header 1':'Mobot'}),
 Document(page_content='关于Mobot的一些文字\n关于Mobot的一些结论性文字',
    metadata={'Header 1':'Mobot','标题 2':'Mobot'})]
```

仅提取在 header_to_split_on 参数中指定的 HTML 标题。

18.2.7 Markdown 文本拆分方法

介绍: Markdown 文本拆分是一种根据 Markdown 的语法规则（例如标题、Bash 代码块、图片和列表）进行文本分块的方法。

特点: 具有对结构的敏感性，能够基于 Markdown 文档的结构特点进行有效的文本分割。

```
markdown_text = '# Mobot\n\n## Stone\n\n这是python\n这是\n\n## markdown\n
   \n这是中文文本拆分'

from langchain.text_splitter import MarkdownHeaderTextSplitter

headers_to_split_on = [
    ("#", "Header 1"),
    ("##", "Header 2"),
    ("###", "Header 3"),
]

markdown_splitter = MarkdownHeaderTextSplitter(headers_to_split_on=
   headers_to_split_on)
md_header_splits = markdown_splitter.split_text(markdown_text)
print(md_header_splits)
```

输出结果：

```
[Document(page_content='这是python\n这是', metadata={'Header 1':'Mobot','Header 2':'Stone'}),
 Document(page_content='这是中文文本拆分', metadata={'Header 1':'Mobot','Header 2':'markdown'})]
```

MarkdownHeaderTextSplitter 能够根据设定的 headers_to_split_on 参数，将 Markdown 文本进行拆分。这一功能使得用户可以便捷地根据指定的标题将 Markdown 文件分割成不同部分，从而提高编辑和管理的效率。

18.2.8 Python 代码拆分方法

```
python_text = """
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

for i in range(10):
    print(i)
"""

from langchain.text_splitter import PythonCodeTextSplitter
python_splitter = PythonCodeTextSplitter(chunk_size=100, chunk_overlap=0)
python_splitter.create_documents([python_text])
```

输出结果：

```
[Document(page_content='class Person:\n    def __init__(self, name, age):\n        self.name = name\n        self.age = age'),
 Document(page_content='p1 = Person("John", 36)\n\nfor i in range(10):\n    print(i)')]
```

18.2.9 LaTex 文本拆分方法

LaTex 文本拆分工具是一种专用于代码分块的工具。它通过解析 LaTex 命令来创建各个块，这些块按照逻辑组织，如章节和小节等。这种方式能够产生更加准确且与上下文相关的分块结果，从而有效地提升 LaTex 文档的组织和处理效率。

```
# input Latex string
latex_text = """
\documentclass{article}
\begin{document}
\maketitle
\section{Introduction}
大型语言模型(LLM)是一种机器学习模型，可以在大量文本数据上进行训练，以生成类似人类的语言。近年来，法学硕士在各种自然语言处理任务中取得了重大进展，包括语言翻译、文本生成和情感分析。
\subsection{法学硕士的历史}
最早的法学硕士是在20世纪80年代开发的和20世纪90年代，但它们受到可处理的数据量和当时可用的计算能力的限制。然而，在过去的十年中，硬件和软件的进步使得在海量数据集上训练法学硕士成为可能，从而导致
\subsection{LLM的应用}
LLM在工业界有许多应用，包括聊天机器人、内容创建和虚拟助理。它们还可以在学术界用于语言学、心理学和计算语言学的研究。
\end{document}
"""

from langchain.text_splitter import LatexTextSplitter
latex_splitter = LatexTextSplitter(chunk_size=100, chunk_overlap=0)
latex_splits = latex_splitter.create_documents([latex_text])
print(latex_splits)
```

输出结果:

```
[Document(page_content='\\documentclass{article}\\begin{document}\\
   maketitle\\section{Introduction} 大型语言模型(LLM)'),
 Document(page_content='是一种机器学习模型，可以在大量文本数据上进行训练，以生成类似人类的语言。近年来,法学硕士在各种自然语言处理任务中取得了重大进展,包括语言翻译、文本生成和情感分析。\\subsection{法学硕士的历史}'),
 Document(page_content='}最早的法学硕士是在'),
 Document(page_content='20世纪80年代开发的和20世纪90'),
 Document(page_content='年代，但它们受到可处理的数据量和当时可用的计算能力的限制。然而，在过去的十年中，硬件和软件的进步使得在海量数据集上训练法学硕士成为可能，从而导致\\subsection{LLM的应用}LLM'),
 Document(page_content='在工业界有许多应用，包括聊天机器人、内容创建和虚拟助理。它们还可以在学术界用于语言学、心理学和计算语言学的研究。\\end{document}')]
```

在上述示例中，我们注意到代码分割时的重叠部分设置为 0。这是因为在处理代码分割过程中，任何重叠的代码都可能完全改变其原有含义。因此，为了保持代码的原始意图和准确性，避免产生误解或错误，设置重叠部分为 0 是必要的。

18.3　文本分块实践建议

18.3.1　分块策略选择

当你决定使用哪种分块器处理数据时，重要的一步是提取数据嵌入并将其存储在向量数据库 (Vector DB) 中。上面的例子中使用文本分块器结合 LanceDB 来存储数据块及其对应的嵌入。

LanceDB 是一个无需配置、开源且无服务器的向量数据库，其数据持久化在硬盘驱动器上，允许用户在不超出预算的情况下实现扩展。此外，LanceDB 与 Python 数据生态系统兼容，因此你可以将其与现有的数据工具（如 pandas、pyarrow 等）结合使用。

18.3.2　分块参数调优建议

- **chunk_size 选择**：根据具体任务和模型限制调整，一般建议在 128-1024 之间
- **chunk_overlap 设置**：对于连续文本建议设置 10-20% 的重叠，对于代码建议设置为 0
- **分隔符选择**：根据文档类型选择合适的分隔符

18.3.3　不同文档类型的推荐分块方法

- **普通文本**：RecursiveCharacterTextSplitter
- **HTML 文档**：HTMLHeaderTextSplitter
- **Markdown 文档**：MarkdownHeaderTextSplitter
- **代码文件**：专用代码拆分器（PythonCodeTextSplitter 等）
- **学术论文**：LatexTextSplitter

第十九章　大模型外挂知识库优化：利用大模型辅助召回

19.1　引言：为什么需要大模型辅助召回？

我们可以通过向量召回的方式从文档库中召回和用户问题相关的文档片段，同时输入到 LLM 中，增强模型回答质量。

常用的方式直接用用户的问题进行文档召回。但是很多时候，用户的问题是十分口语化的，描述的也比较模糊，这样会影响向量召回的质量，进而影响模型回答效果。

19.2　策略一：HYDE（Hypothetical Document Embeddings）

19.2.1　HYDE 基本介绍

- **论文**：《Precise Zero-Shot Dense Retrieval without Relevance Labels》
- **论文地址**：https://arxiv.org/pdf/2212.10496.pdf

19.2.2　HYDE 思路详解

HYDE 的核心思路分为四个步骤：

1. **生成假设答案**：用 LLM 根据用户 query 生成 k 个” 假答案”。大模型生成答案采用 sample 模式，保证生成的 k 个答案不一样。此时的回答内容很可能是存在知识性错误，因为如果能回答正确，那就不需要召回补充额外知识了。不过不要紧，我们只是想通过大模型去理解用户的问题，生成一些” 看起来” 还不错的假答案。
2. **向量化处理**：利用向量化模型，将生成的 k 个假答案和用户的 query 变成向量。
3. **向量融合**：将 k+1 个向量取平均，其中 d_k 为第 k 个生成的答案，q 为用户问题，f 为向量化操作：

$$\hat{v}_{q_{ij}} = \frac{1}{N+1}\left[\sum_{k=1}^{N} f(\hat{d}_k) + f(q_{ij})\right]$$

4. **召回答案**：利用融合向量 v 从文档库中召回答案。融合向量中既有用户问题的信息，也有想要答案的模式信息，可以增强召回效果。

19.2.3　HYDE 存在的问题与局限性

该方法在结合微调过的向量化模型时，效果就没那么好了，非常依赖打辅助的 LLM 的能力。

原始的该模型并未在 TREC DL19/20 数据集上训练过。模型有上标 FT 指的是向量化模型在 TREC DL 相关的数据集上微调过的。

表 19.1: HYDE 方法在不同配置下的实验结果对比（NDCG@10）

Model	DL19	DL20
Baseline Models		
Contriever	44.5	42.1
Contriever FT	62.1	63.2
HyDE with Contriever		
w/ Flan-T5(11b)	48.9	52.9
w/ Cohere(52b)	53.8	53.8
w/ GPT(175b)	61.3	57.9
HyDE with Contriever FT		
w/ Flan-T5(11b)	60.2	62.1
w/ Cohere(52b)	61.4	63.1
w/ GPT(175b)	67.4	63.5

实验发现：

- 对于没有微调过的向量化模型（zero shot 场景），HyDE 还是非常有用的，并且随着使用的 LLM 模型的增大，效果不断变好（因为 LLM 的回答质量提高了）
- 对于微调过的向量化模型，如果使用比较小的 LLM 生成假答案（小于 52B 参数量），HyDE 技术甚至会带来负面影响

19.3　策略二：FLARE（Forward-Looking Active REtrieval）

19.3.1　FLARE 基本介绍

- **论文**：《Active REtrieval Augmented Generation》

- **论文地址**：https://arxiv.org/abs/2305.06983

19.3.2　为什么需要 FLARE？

对于大模型外挂知识库，大家通常的做法是根据用户 query 一次召回文档片段，让模型生成答案。只进行一次文档召回在长文本生成的场景下效果往往不好，生成的文本过长，更有可能扩展出和 query 相关性较弱的内容，如果模型没有这部分知识，容易产生模型幻觉问题。

一种解决思路是随着文本生成，多次从向量库中召回内容。

19.3.3　FLARE 召回策略

传统多次召回方案

已有的多次召回方案比较被动：

- **固定 token 间隔**：每生成固定的 n 个 token 就召回一次
- **句子级别召回**：每生成一个完整的句子就召回一次
- **子问题分解**：用户 query 一步步分解为子问题，需要解答当前子问题时候，就召回一次

这些策略并不能保证不需要召回的时候不召回，需要召回的时候触发召回。子问题分解方案需要设计特定的 prompt 工程，限制了其通用性。

19.3.4　FLARE 策略 1：主动召回标识

策略 1 思路

通过设计 prompt 以及提供示例的方式，让模型知道当遇到需要查询知识的时候，提出问题，并按照格式输出，和 ToolFormer 的模式类似。

具体步骤：

1. **生成主动召回标识**：提出问题的格式为 [Search(" 模型自动提出的问题")]（称其为主动召回标识）。利用模型生成的问题去召回答案。
2. **答案整合**：召回出答案后，将答案放到用户 query 的前边，然后去掉主动召回标识之后，继续生成。
3. **动态更新**：当下一次生成主动召回标识之后，将上一次召回出来的内容从 prompt 中去掉。

策略 1 缺陷与解决方案

- **缺陷 1**：LLM 不愿意生成主动召回标识
  - **解决方案**：对"[" 对应的 logit 乘 2，增加生成"[" 的概率，"[" 为主动召回标识的第一个字，进而促进主动召回标识的生成

- **缺陷 2**：过于频繁的主动召回可能会影响生成质量
  - **解决方案**：在刚生成一次主动召回标识、得到召回后的文档、去掉主动召回标识之后，接下来生成的几个 token 禁止生成"["
- **缺陷 3**：不微调该方案不太可靠，很难通过 few shot 的方式让模型生成这种输出模式

19.3.5　FLARE 策略 2：基于置信度的召回

策略 2 思路

策略 1 存在的第 3 点缺陷比较知名，因此作者提出了另外一个策略。该策略基于一个假设：模型生成的词对应的概率能够表现生成内容的置信度。

（传统的 ChatGPT 接口是用不了策略 2 的，因为得不到生成每个词的概率。）

具体步骤：

1. **初始生成**：根据用户的 query，进行第一次召回，让模型生成答案。
2. **句子提取**：之后，每生成 64 个 token，用 NLTK 工具包从 64 个 token 里边找到第一个完整句子，当作" 假答案"，扔掉多余的 token。
3. **置信度检测与召回触发**：如果" 假答案" 里有任意一个 token 对应的概率，低于某一阈值，那么就利用这个句子进行向量召回。
4. **错误处理**：触发召回的" 假答案" 很可能包含事实性错误，降低召回准确率。设计了两种方法解决这个问题：
   - **方法 1**：将" 假答案" 中生成概率低于某一阈值的 token 扔掉（低概率的 token 很有可能存在错误信息），然后再进行向量召回
   - **方法 2**：利用大模型能力，对" 假答案" 中置信度低的内容进行提问，生成一个问题，用生成的问题进行向量召回
5. **重新生成**：利用召回出来的文本，重新生成新的" 真答案"，然后进行下一个句子的生成。

19.4　技术对比与总结

19.4.1　方法优势比较

19.4.2　实践建议

- **资源充足场景**：优先考虑 FLARE 策略 2，效果最优但需要能获取 token 概率
- **一般应用场景**：可以考虑 HYDE 方法，实现相对简单
- **实时性要求高**：FLARE 策略 1 可能更合适，但需要精心设计 prompt
- **模型选择**：大尺寸的辅助 LLM 通常能带来更好的效果提升

表 19.2: HYDE 与 FLARE 方法对比

特性	HYDE	FLARE
核心思想	通过生成假设答案来增强查询表示	基于生成置信度动态触发召回
适用场景	零样本或少样本场景	长文本生成场景
计算开销	相对较低	相对较高（多次召回）
实现复杂度	中等	较高
效果稳定性	依赖辅助 LLM 质量	依赖 token 概率获取
主要优势	简单有效，提升零样本效果	减少幻觉，提高长文本质量

19.4.3　未来发展方向

- 更智能的召回触发机制
- 多模态信息的融合召回
- 端到端的训练优化
- 计算效率的进一步提升

第二十章　大模型外挂知识库优化负样本挖掘篇

20.1　引言：为什么需要构建负难样本？

在各类检索任务中，为训练好一个高质量的检索模型，往往需要从大量的候选样本集合中采样高质量的负例，配合正例一起进行训练。

20.2　负难样本构建方法

20.2.1　随机采样策略（Random Sampling）方法

方法描述

直接基于均匀分布从所有的候选 Document 中随机抽取 Document 作为负例。

存在问题

由于无法保证采样得到的负例的质量，故经常会采样得到过于简单的负例，其不仅无法给模型带来有用信息，还可能导致模型过拟合，进而无法区分某些较难的负例样本。

梯度影响分析

对于随机采样方法，由于其采样得到的负例往往过于简单，其会导致该分数接近于零：

$$s_n(q,d) \longrightarrow 0$$

进而导致其生成的梯度均值也接近于零：

$$\bigtriangledown_\theta l(q,d) \longrightarrow 0,$$

这样过于小的梯度均值会导致模型不易于收敛。

20.2.2　Top-K 负例采样策略（Top-K Hard Negative Sampling）方法

方法描述

基于稠密检索模型对所有候选 Document 与 Query 计算匹配分数，然后直接选择其中 Top-K 的候选 Document 作为负例。

优点

可以保证采样得到的负例是模型未能较好区分的较难负例。

存在问题

很可能将潜在的正例也误判为负例，即假负例（False Negative）。如果训练模型去将该部分假负例与正例区分开来，反而会导致模型无法准确衡量 Query-Document 的语义相似度。

梯度影响分析

由于其很容易采样得到语义与正例一致的假负例，其会导致正负样本的右项 $\nabla_\theta s_n(q,d)$ 值相似，但是左项符号相反，这样会导致计算得到的梯度方差很大，同样导致模型训练不稳定。

20.2.3　困惑负样本采样方法 SimANS 方法

动机

在所有负例候选中，与 Query 的语义相似度接近于正例的负例可以同时具有较大的梯度均值和较小的梯度方差，是更加高质量的困惑负样本。

方法

对与正例语义相似度接近的困惑负例样本进行采样。

采样方法特点

- 与 Query 无关的 Document 应被赋予较低的相关分数，因其可提供的信息量不足
- 与 Query 很可能相关的 Document 应被赋予较低的相关分数，因其可能是假负例
- 与正例语义相似度接近的 Document 应该被赋予较高的相关分数，因其既需要被学习，同时是假负例的概率相对较低

困惑样本采样分布

通过以上分析可得，在该采样分布中，随着 Query 与候选 Document 相关分数 $s(q,d_i)$ 和与正例的相关分数 $s(q,d^+)$ 的差值的缩小，该候选 Document 被采样作为负例的概率应该逐

渐增大，故可将该差值作为输入，配合任意一单调递减函数 $f(\cdot)$ 即可实现（如 e^{-x}）。故可设计采样分布如下所示：

$$p_i \propto \exp\left(-a\left(s\left(q,d_i\right)-s\left(q,\tilde{d}^{+}\right)-b\right)^2\right), \forall d_i \in \widetilde{\mathcal{D}}$$

其中 a 为控制该分布密度的超参数，b 为控制该分布极值点的超参数，$\tilde{d}^{+} \in \mathcal{D}^{+}$ 是一随机采样的正例样本，$\widetilde{\mathcal{D}}^{-}$ 是 Top-K 的负例。通过调节 K 的大小，我们可以控制该采样分布的计算开销。

SimANS 算法伪代码

```
Algorithm 1: The algorithm of SimANS.
Input: Queries and their positive documents $\{(q,\mathcal{D}^{+})\}$,
   document pool $\mathcal{D}$, pre-learned dense retrieval model M
1  Build the ANN index on D using M.
2  Retrieve the top-k ranked negatives $\widetilde{\mathcal{D}}^{-}$ for
   each query with their relevance scores $\{s(q,d_{i})\}$ from $\mathcal
   {D}$.
3  Compute the relevance scores of each query and its positive documents
   $\{s(q,\mathcal{D}^{+})\}$.
4  Generate the sampling probabilities of retrieved top-k negatives $\{
   p_i\}$ for each query using Eq.3.
5  Construct new training data $\{(q,\mathcal{D}^{+},\widetilde{\mathcal{
   D}}^{-})\}$.
6  while M has not converged do
7      Sample a batch from $\{(q,\mathcal{D}^{+},\widetilde{\mathcal{D
   }}^{-})\}$.
8      Sample ambiguous negatives for each instance from the batch
   according to $\{p_i\}$.
9      Optimize parameters of M using the batch and sampled negatives.
10 end
```

20.2.4　利用对比学习微调方式构建负例方法

对比学习目的

对比学习是优化向量化模型的常用训练方法，目的是优化向量化模型，使其向量化后的文本，相似的在向量空间距离近，不相似的在向量空间距离远。

文档召回场景

文档召回场景下，做对比学习（有监督）需要三元组（问题，文档正例，文档负例）。文档正例是和问题密切相关的文档片段，文档负例是和问题不相关的文档片段，可以是精挑细选的，也可以是随机出来的。

构建方法

如果是随机出来的话，完全可以用同一个 batch 里，其他问题的文档正例当作某一个问题的文档负例，如果想要效果好，还需要有比较大的 batch size。

损失函数

损失函数是基于批内负样本的交叉熵损失，如下公式所示，q、d 分别表示问题和文档正例对应的向量，τ 为温度系数，sim 函数可以是 cos 相似度或者点积。

论文：SimCSE: Simple Contrastive Learning of Sentence Embeddings

$$\ell_i = -\log \frac{\mathrm{e}^{\mathrm{sim}\left(q_i, \mathrm{d}_i^+\right)/\tau}}{\sum_{j=1}^{N} \mathrm{e}^{\mathrm{sim}\left(q_i, \mathrm{d}_j^+\right)/\tau}}$$

实现方法

```
q_reps = self.encode(query)  # 问题矩阵维度(B1, d)
d_reps = self.encode(doc)    # 文档矩阵维度(B2, d)
score = torch.matmul(q_reps, d_reps.transpose(0,1))  # 计算相似度矩阵维度:(B1, B2)
scores = scores / self.temperature
target = torch.arange(scores.size(0), device=scores.device, dtype=torch.long)
# 考虑文档负例不仅来自于batch内其他样本的文档正例，也可能人工的给每个样本构造一些文档负例
target = target * (p_reps.size(0) // d_reps.size(0))
loss = cross_entropy(scores, target)  # 交叉熵损失函数
```

注：BGE2 论文里，做基于批内负样本的对比学习时同时考虑了多任务问题。之前也介绍了，不同任务加的 prompt 是不同的，如果把不同任务的样本放到一个 batch 里，模型训练时候就容易出现偷懒的情况，有时候会根据 prompt 的内容来区分正负例，降低任务难度，这是不利于对比学习效果的。因此，可以通过人为的规定，同一个 batch 里，只能出现同一种任务的样本缓解这个问题。（实际应用场景下，如果任务类别不是非常多的话，最好还是一个任务训练一个模型，毕竟向量化模型也不大，效果会好一些）

20.2.5　基于批内负采样的对比学习方法

本质

随机选取文档负例，如果能有针对性的，找到和文档正例比较像的文档负例（模型更难区分这些文档负例），加到训练里，是有助于提高对比学习效果的。就好比我们只有不断的做难题才能更好的提高考试水平。

论文方法

在文档向量空间找到和文档正例最相近的文档片段当作文档负例，训练向量化模型。模型更新一段时间后，刷新文档向量，寻找新的文档负例，继续训练模型。

参考论文：

- Approximate nearest neighbor negative contrastive learning for dense text retrieval
- Contrastive learning with hard negative samples
- Hard negative mixing for contrastive learning
- Optimizing dense retrieval model training with hard negatives
- SimANS: Simple Ambiguous Negatives Sampling for Dense Text Retrieval

20.2.6　相同文章采样方法

思路

文档正例所在的文章里，其他文档片段当作难负例，毕竟至少是属于同一主题的，和随机样本比起来比较难区分。

存在问题

实际应用场景下，如果你的数据比较脏，难例挖掘用处可能不大。

20.2.7　LLM 辅助生成软标签及蒸馏

方法

根据用户问题召回的相关文档片段最终是要为 LLM 回答问题服务的，因此 LLM 认为召回的文档是否比较好很重要，以下介绍的方法是 BGE2 提出的。对于向量化模型的训练，可以让 LLM 帮忙生成样本的辅助标签，引导向量化模型训练。辅助标签的生成可用如下公式表示。在已知 LLM 需要输出的标准答案下，分别将问题和各个文档片段 C 放入 LLM 的 prompt 中，看 LLM 生成标准答案的概率 r 大小，当作辅助标签。r 越大，表示其对应的文档片段对生成正确答案的贡献越大，也就越重要。

$$r_{C|O} = \prod_{i=1}^{|O|} LLM(o_i|C, O_{:i-1})$$

存在问题

- 打标要求有点太高
- 很多实际应用场景，我们并没法拿到 LLM 回答的标准答案，同时对每个问题的候选文档片段都计算一个 r，开销貌似有点大

优化策略

利用以上 LLM 生成的标签以及 KL 散度（笔者认为论文里这个形式的公式不能叫做 KL 散度吧...），对模型进行优化。$\mathcal{P}$ 为某个问题 q 对应的候选文档片段 p 的集合，e 表示向量，$\langle\cdot,\cdot\rangle$ 表示相似度操作，w 是对所有候选文档 p 对应的辅助标签值 r 经过 softmax 变换后的值。本质是，如果 LLM 认为某个文档片段越重要，给它的优化权重越大。为了进一步稳定蒸馏效果，还可以对候选文档片段根据 r 进行排序，只用排名靠后的样本进行优化。

$$\min \sum_{\mathcal{P}} -w_i * \log \frac{\exp(\langle e_q, e_p \rangle / \tau)}{\sum_{p' \in \mathcal{P}} \exp(\langle e_q, e_{p'} \rangle / \tau)}$$

20.3　辅助知识：梯度计算方法

20.3.1　梯度计算公式

以稠密检索常用的 BCE loss 为例，正例与采样的负例在计算完语义相似度分数后，均会被 softmax 归一化，之后计算得到的梯度如下所示：

$$\nabla_\theta l(q,d) = \begin{cases} (s_n(q,d) - 1) \bigtriangledown_\theta s_n(q,d) & \text{if } d \in \mathcal{D}^+ \\ s_n(q,d) \bigtriangledown_\theta s_n(q,d) & \text{if } d \in \mathcal{D}^- \end{cases}$$

注：$s_n(q,d)$：经过 softmax 归一化后的语义相似度分数

20.4　方法总结与对比

20.4.1　各方法优缺点对比

20.4.2　实践建议

- **资源充足场景**：优先考虑 SimANS 或 LLM 辅助方法
- **一般应用场景**：推荐使用对比学习方法结合批内负采样
- **计算资源有限**：可选用 Top-K 采样但要注意假负例问题
- **数据质量高**：可尝试相同文章采样方法
- **实时训练**：随机采样结合动态难例挖掘

表 20.1: 负样本挖掘方法对比分析

方法	优点	缺点
随机采样	实现简单，计算开销小	负例质量低，梯度均值小，收敛慢
Top-K 采样	能获取难负例	可能引入假负例，梯度方差大
SimANS	平衡梯度均值和方差	计算复杂度较高
对比学习	充分利用 batch 内信息	需要大 batch size
批内负采样	针对性强	需要频繁更新负例库
相同文章采样	语义相关性高	数据质量要求高
LLM 辅助	利用 LLM 知识	计算开销大，需要标准答案

20.4.3　未来发展方向

- 更智能的负例质量评估机制
- 多模态负例挖掘技术
- 自监督的负例生成方法
- 计算效率的进一步优化
- 跨领域的负例迁移学习

第二十一章　RAG(检索增强生成) 评测面

21.1　引言：为什么需要对 RAG 进行评测？

在探索和优化 RAG(检索增强生成器) 的过程中，如何有效评估其性能已经成为关键问题。

21.2　RAG 测试集合成方法

21.2.1　测试集构建需求

假设已经成功构建了一个 RAG 系统，并且想要评估其性能，需要包含以下列的评估数据集：

- **question(问题)**：想要评估的 RAG 的问题
- **ground_truths(真实答案)**：问题的真实答案
- **answer(答案)**：RAG 预测的答案
- **contexts(上下文)**：RAG 用于生成答案的相关信息列表

前两列代表真实数据，最后两列代表 RAG 预测数据。

21.2.2　测试集生成流程

要创建这样的数据集，首先需要生成问题和答案的元组，然后在 RAG 上运行这些问题以获得预测结果。

生成问题和基准答案

1. 准备 RAG 数据，拆分为块并嵌入向量数据库
2. 指示 LLM 从指定主题中生成 num_questions 个问题
3. 得到问题和答案元组

具体操作步骤

1. 选择一个随机块作为根上下文

2. 从向量数据库中检索 K 个相似的上下文
3. 将根上下文和其 K 个相邻上下文的文本连接起来构建更大的上下文
4. 使用大上下文和 num_questions 在提示模板中生成问题和答案

21.2.3 提示模板设计

```
Your task is to formulate exactly {num_questions} questions from given
    context and provide the answer to each one.
End each question with a '?' character and then in a newline write the
    answer to that question using only the context provided.
Separate each question/answer pair by "XXX"
Each question must start with "question:".
Each answer must start with "answer:".
The question must satisfy the rules given below:
1. The question should make sense to humans even when read without the
    given context.
2. The question should be fully answered from the given context.
3. The question should be framed from a part of context that contains
    important information. It can also be from tables, code, etc.
4. The answer to the question should not contain any links.
5. The question should be of moderate difficulty.
6. The question must be reasonable and must be understood and responded
    by humans.
7. Do no use phrases like 'provided context', etc in the question
8. Avoid framing question using word "and" that can be decomposed into
    more than one question.
9. The question should not contain more than 10 words, make of use of
    abbreviation wherever possible.
context: {context}
```

21.2.4 编码实现示例

```
# 1. 从Wikipedia加载数据
from langchain.document_loaders import WikipediaLoader
topic = "python programming"
wikipedia_loader = WikipediaLoader(
    query=topic,
    load_max_docs=1,
    doc_content_chars_max=100000
```

```
)
docs = wikipedia_loader.load()
doc = docs[0]

# 2. 数据分块
from langchain.text_splitter import RecursiveCharacterTextSplitter
CHUNK_SIZE = 512
CHUNK_OVERLAP = 128
splitter = RecursiveCharacterTextSplitter(
    chunk_size=CHUNK_SIZE,
    chunk_overlap=CHUNK_OVERLAP,
    separators=["\n\n", "\n", ".", " "]
)
splits = splitter.split_documents([doc])

# 3. 在Pinecone中创建索引
import pinecone
import os
pinecone.init(
    api_key=os.environ.get("PINECONE_API_KEY"),
    environment=os.environ.get("PINECONE_ENV")
)
index_name = topic.replace(" ", "-")
if index_name in pinecone.list_indexes():
    pinecone.delete_index(index_name)
pinecone.create_index(index_name, dimension=768)

# 4. 使用LangChain包装器索引分片嵌入
from langchain.vectorstores import Pinecone
docsearch = Pinecone.from_documents(
    splits,
    embedding_model,
    index_name=index_name
)

# 5. 生成合成数据集
from langchain.embeddings import VertexAIEmbeddings
from langchain.llms import VertexAI
from testset_generator import TestsetGenerator

```

```
generator_llm = VertexAI(
    location="europe-west3",
    max_output_tokens=256,
    max_retries=20
)
embedding_model = VertexAIEmbeddings()
testset_generator = TestsetGenerator(
    generator_llm=generator_llm,
    documents=splits,
    embedding_model=embedding_model,
    index_name=index_name,
    key="text"
)

# 6. 调用generate方法生成数据集
synthetic_dataset = testset_generator.generate(
    num_contexts=10,
    num_questions_per_context=2
)
```

21.2.5 RAG 预测收集

```
# 初始化RAG
from rag import RAG
rag = RAG(
    index_name,
    "text-bison",
    embedding_model,
    "text"
)

# 迭代合成数据集收集预测
rag_answers = []
contexts = []
for i, row in synthetic_dataset.iterrows():
    question = row["question"]
    prediction = rag.predict(question)
    rag_answer = prediction["answer"]
    rag_answers.append(rag_answer)
```

```
    source_documents = prediction["source_documents"]
    contexts.append([s.page_content for s in source_documents])

synthetic_dataset_rag = synthetic_dataset.copy()
synthetic_dataset_rag["answer"] = rag_answers
synthetic_dataset_rag["contexts"] = contexts
```

21.3 RAG 评估方法分类

21.3.1 评估方法概述

主要有两种方法来评估 RAG 的有效性：独立评估和端到端评估。

21.3.2 独立评估（Independent Evaluation）

独立评估介绍

独立评估涉及对检索模块和生成模块（即阅读和合成信息）的评估。

独立评估模块

生成模块指的是将检索到的文档与查询相结合，形成增强或合成的输入。这与最终答案或响应的生成不同，后者通常采用端到端的评估方式。

独立评估指标

1. 答案相关性（Answer Relevancy）

- **目标**：评估生成的答案与提供的问题提示之间的相关性
- **评估标准**：答案如果缺乏完整性或者包含冗余信息，得分将相对较低
- **计算方法**：通过问题和答案的结合来计算，评分范围 0 到 1，高分代表更好的相关性
- **示例**：
  - **问题**：健康饮食的主要特点是什么？
  - **低相关性答案**：健康饮食对整体健康非常重要
  - **高相关性答案**：健康饮食应包括各种水果、蔬菜、全麦食品、瘦肉和乳制品，为优化健康提供必要的营养素

2. 忠实度（Faithfulness）

- **目标**：检查生成的答案在给定上下文中的事实准确性
- **评估过程**：答案内容与其检索到的上下文之间的比对
- **评分范围**：0 到 1，更高的数值意味着答案与上下文的一致性更高

- **示例：**
  - **问题：**居里夫人的主要成就是什么？
  - **背景：**玛丽 • 居里 (1867-1934 年) 是一位开创性的物理学家和化学家，她是第一位获得诺贝尔奖的女性，也是唯一一位在两个不同领域获得诺贝尔奖的女性
  - **高忠实度答案：**玛丽 • 居里在物理和化学两个领域都获得了诺贝尔奖，使她成为第一位实现这一成就的女性
  - **低忠实度答案：**玛丽 • 居里只在物理学领域获得了诺贝尔奖

3. 上下文精确度（Context Precision）

- **目标：**评估所有在给定上下文中与基准信息相关的条目是否被正确地排序
- **理想情况：**所有相关的内容应该出现在排序的前部
- **评分范围：**0 到 1，较高的得分反映更高的精确度
- **相关指标：**命中率 (Hit Rate)、平均排名倒数 (MRR)、归一化折扣累积增益 (NDCG)、精确度 (Precision) 等

4. 答案正确性（Answer Correctness）

- **目标：**测量生成的答案与实际基准答案之间的匹配程度
- **评估方法：**基准答案和生成答案的对比
- **评分范围：**0 到 1，较高的得分表明生成答案与实际答案的一致性更高
- **示例：**
  - **基本事实：**埃菲尔铁塔于 1889 年在法国巴黎竣工
  - **答案正确率高：**埃菲尔铁塔于 1889 年在法国巴黎竣工
  - **答案正确率低：**埃菲尔铁塔于 1889 年竣工，矗立在英国伦敦

21.3.3　端到端评估（End-to-End Evaluation）

端到端评估介绍

对 RAG 模型对特定输入生成的最终响应进行评估，涉及模型生成的答案与输入查询的相关性和一致性。

端到端评估模块

- **无标签的内容评估：**
  - 评价指标：答案的准确性、相关性和无害性
- **有标签的内容评估：**
  - 评价指标：准确率 (Accuracy) 和精确匹配 (EM)

21.4 RAG 关键指标和能力

21.4.1 关键指标

评估 RAG 在不同下游任务和不同检索器中的应用可能会得到不同的结果，但关注以下三个关键指标：

- 答案的准确性（Answer Accuracy）
- 答案的相关性（Answer Relevancy）
- 上下文的相关性（Context Relevancy）

21.4.2 关键能力

RAG 需要具备四项基本能力：

- **抗噪声能力**：在存在噪声数据的情况下仍能保持良好性能
- **拒绝无效回答能力**：能够识别并拒绝无法准确回答的问题
- **信息综合能力**：能够综合多个来源的信息生成完整答案
- **反事实稳健性**：对反事实或假设性问题的处理能力

21.5 RAG 评估框架

21.5.1 RAGAS 框架

RAGAS 介绍

RAGAS 是一个基于简单手写提示的评估框架，通过这些提示全自动地衡量答案的准确性、相关性和上下文相关性。

RAGAS 算法原理

1. **答案忠实度评估**：利用 LLM 分解答案为多个陈述，检验每个陈述与上下文的一致性。根据支持的陈述数量与总陈述数量的比例计算” 忠实度得分”
2. **答案相关性评估**：使用 LLM 创造可能的问题，分析这些问题与原始问题的相似度。通过计算所有生成问题与原始问题相似度的平均值得出答案相关性得分
3. **上下文相关性评估**：运用 LLM 筛选出直接与问题相关的句子，以这些句子占上下文总句子数量的比例确定上下文相关性得分

21.5.2 ARES 框架

ARES 介绍

ARES 的目标是自动化评价 RAG 系统在上下文相关性、答案忠实度和答案相关性三个方面的性能。ARES 减少评估成本，通过使用少量的手动标注数据和合成数据，并应用预测驱动推理 (PDR) 提供统计置信区间，提高评估准确性。

ARES 算法原理

1. **生成合成数据集**：使用语言模型从目标语料库中的文档生成合成问题和答案，创建正负两种样本
2. **训练 LLM 裁判**：对轻量级语言模型进行微调，利用合成数据集训练它们以评估上下文相关性、答案忠实度和答案相关性
3. **基于置信区间对 RAG 系统排名**：使用裁判模型为 RAG 系统打分，结合手动标注的验证集，采用 PPI 方法生成置信区间，可靠地评估 RAG 系统性能

21.6 评估实践建议

21.6.1 评估流程设计

1. **测试集构建**：根据实际应用场景构建具有代表性的测试集
2. **基准设定**：建立合理的性能基准和通过标准
3. **多维度评估**：结合独立评估和端到端评估方法
4. **结果分析**：深入分析失败案例，识别系统瓶颈
5. **迭代优化**：基于评估结果进行系统优化和改进

21.6.2 常见挑战与解决方案

- **数据质量问题**：确保测试集的质量和代表性
- **评估标准一致性**：建立统一的评估标准和流程
- **计算资源限制**：合理规划评估所需的计算资源
- **结果可解释性**：提供详细的评估报告和案例分析

21.6.3 最佳实践

- **定期评估**：建立定期的评估机制
- **多维度监控**：监控系统在不同维度上的表现
- **用户反馈集成**：将用户反馈纳入评估体系
- **持续改进**：基于评估结果持续优化系统

第二十二章　检索增强生成 (RAG) 优化策略篇

22.1　RAG 基础功能篇

22.1.1　RAG 工作流程

RAG 的工作流程包含以下核心模块：文档块切分、文本嵌入模型、提示工程、大模型生成。从 RAG 的工作流程看，RAG 模块有：文档块切分、文本嵌入模型、提示工程、大模型生成。

22.2　RAG 各模块优化策略

22.2.1　文档块切分优化策略

- 设置适当的块间重叠
- 多粒度文档块切分
- 基于语义的文档切分
- 文档块摘要

22.2.2　文本嵌入模型优化策略

- 基于新语料微调嵌入模型
- 动态表征

22.2.3　提示工程优化策略

- 优化模板增加提示词约束
- 提示词改写

22.2.4　大模型迭代优化策略

- 基于正反馈微调模型
- 量化感知训练
- 提供大 context window 的推理模型

22.2.5　查询召回后处理优化

- 元数据过滤
- 重排序减少文档块数量

22.3　RAG 架构优化策略

22.3.1　知识图谱 (KG) 上下文增强

向量数据库上下文增强存在的问题

- 无法获取长程关联知识
- 信息密度低（尤其当 LLM context window 较小时不友好）

知识图谱增强策略

增加一路与向量库平行的 KG（知识图谱）上下文增强策略。具体方式：对于用户 query，通过利用 NL2Cypher 进行 KG 增强。

优化策略：常用图采样技术来进行 KG 上下文增强。处理方式：根据 query 抽取实体，然后把实体作为种子节点对图进行采样（必要时，可把 KG 中节点和 query 中实体先向量化，通过向量相似度设置种子节点），然后把获取的子图转换成文本片段，从而达到上下文增强的效果。

22.3.2　Self-RAG：大模型对召回结果的筛选

典型 RAG 架构中向量数据库的问题

经典的 RAG 架构中（包括 KG 进行上下文增强），对召回的上下文无差别地与 query 进行合并，然后访问大模型输出应答。但有时召回的上下文可能与 query 无关或者矛盾，此时就应舍弃这个上下文，尤其当大模型上下文窗口较小时非常必要（目前 4k 的窗口比较常见）。

Self-RAG 核心思想

Self-RAG 是更加主动和智能的实现方式，主要步骤概括如下：

1. 判断是否需要额外检索事实性信息（retrieve on demand），仅当有需要时才召回
2. 平行处理每个片段：生产 prompt + 一个片段的生成结果

3. 使用反思字段，检查输出是否相关，选择最符合需要的片段
4. 再重复检索
5. 生成结果会引用相关片段，以及输出结果是否符合该片段，便于查证事实

Self-RAG 的创新点：反思字符（Reflection Tokens）

Self-RAG 的重要创新：Reflection tokens（反思字符）。通过生成反思字符这一特殊标记来检查输出。这些字符会分为 Retrieve 和 Critique 两种类型，会标示：检查是否有检索的必要，完成检索后检查输出的相关性、完整性、检索片段是否支持输出的观点。模型会基于原有词库和反思字段来生成下一个 token。

表 22.1: Self-RAG 反思字符类型

Type	Input	Output	Definitions
Retrieve	x / x,y	yes, no, continue	Decides when to retrieve with R
ISREL	x,d	relevant, irrelevant	d provides useful information to so
ISSUP	x,d,y	fully supported, partially supported, no support	All verification-worthy statements
ISUSE	x,y	5,4,3,2,1	y is a useful response to x.

Self-RAG 训练过程

对于训练，模型通过将反思字符集成到其词汇表中来学习生成带有反思字符的文本。它是在一个语料库上进行训练的，其中包含由 Critic 模型预测的检索到的段落和反思字符。该 Critic 模型评估检索到的段落和任务输出的质量。使用反思字符更新训练语料库，并训练最终模型以在推理过程中独立生成这些字符。

为了训练 Critic 模型，手动标记反思字符的成本很高，于是使用 GPT-4 生成反思字符，然后将这些知识提炼到内部 Critic 模型中。不同的反思字符会通过少量演示来提示具体说明。

Self-RAG 推理过程

Self-RAG 使用反思字符来自我评估输出，使其在推理过程中具有适应性。根据任务的不同，可以定制模型，通过检索更多段落来优先考虑事实准确性，或强调开放式任务的创造力。该模型可以决定何时检索段落或使用设定的阈值来触发检索。

当需要检索时，生成器同时处理多个段落，产生不同的候选。进行片段级 beam search 以获得最佳序列。每个细分的分数使用 Critic 分数进行更新，该分数是每个批评标记类型的归一化概率的加权和。

Self-RAG 代码实战

```
from vllm import LLM, SamplingParams

model = LLM("selfrag/selfrag_llama2_7b",
            download_dir="/gscratch/h21ab/akari/model_cache", dtype="half"
               )
sampling_params = SamplingParams(temperature=0.0, top_p=1.0, max_tokens
   =100,
                                 skip_special_tokens=False)

def format_prompt(input, paragraph=None):
    prompt = "### Instruction: \n{0}\n\n### Response: \n".format(input)
    if paragraph is not None:
        prompt += "[Retrieval]<paragraph>{0}</paragraph>".format(
           paragraph)
    return prompt

query_1 = "Leave odd one out: twitter, instagram, whatsapp."
query_2 = "What is China?"
queries = [query_1, query_2]

# 对于不需要检索的查询
preds = model.generate([format_prompt(query) for query in queries],
   sampling_params)

for pred in preds:
    print("Model prediction: {0}".format(pred.outputs[0].text))
```

22.3.3　多向量检索器多模态 RAG

多向量检索器核心思想

将文档（用于答案合成）和引用（用于检索）分离，这样可以针对不同的数据类型生成适合自然语言检索的摘要，同时保留原始的数据内容。它可以与多模态 LLM 结合，实现跨模态的 RAG。

半结构化 RAG 支持（文本 + 表格）

1. 将原始文档进行版面分析（基于 Unstructured 工具），生成原始文本和原始表格
2. 原始文本和原始表格经 summary LLM 处理，生成文本 summary 和表格 summary

3. 用同一个 embedding 模型把文本 summary 和表格 summary 向量化，并存入多向量检索器
4. 多向量检索器存入文本/表格 embedding 的同时，也会存入相应的 summary 和 raw data
5. 用户 query 向量化后，用 ANN 检索召回 raw text 和 raw table
6. 根据 query + raw text + raw table 构造完整 prompt，访问 LLM 生成最终结果

多模态 RAG 支持（文本 + 表格 + 图片）

三种技术路线：

- **选项 1**：对文本和表格生成 summary，然后应用多模态 embedding 模型把文本/表格 summary、原始图片转化成 embedding 存入多向量检索器
- **选项 2**：首先应用多模态大模型（GPT4-V、LLaVA、FUYU-8b）生成图片 summary，然后对文本/表格/图片 summary 进行向量化存入多向量检索器中
- **选项 3**：前置阶段同选项 2 相同，对话时根据 query 召回原始文本/表格/图片，构造完整 Prompt 访问多模态大模型

私有化多模态 RAG 支持

如果数据安全是重要考量，需要把 RAG 流水线进行本地部署。比如可用 LLaVA-7b 生成图片摘要，Chroma 作为向量数据库，Nomic's GPT4All 作为开源嵌入模型，多向量检索器，Ollama.ai 中的 LLaMA2-13b-chat 用于生成应答。

22.3.4　RAG Fusion 优化策略

检索增强这一块主要借鉴了 RAGFusion 技术，原理是当接收用户 query 时，让大模型生成 5-10 个相似的 query，然后每个 query 去匹配 5-10 个文本块，接着对所有返回的文本块再做个倒序融合排序，如果有需求就再加个精排，最后取 Top K 个文本块拼接至 prompt。

优点：增加了相关文本块的召回率；对用户的 query 自动进行了文本纠错、分解长句等功能。

缺点：无法从根本上解决理解用户意图的问题。

22.3.5　模块化 RAG 优化策略

打破了传统的" 原始 RAG" 框架，提供了更广泛的多样性和更高的灵活性。模块包括：

- **搜索模块**：融合了直接在语料库中进行搜索的方法
- **记忆模块**：充分利用大语言模型本身的记忆功能来引导信息检索
- **额外生成模块**：通过大语言模型生成必要的上下文，而非直接从数据源进行检索
- **任务适应模块**：将 RAG 调整以适应各种下游任务
- **对齐模块**：在检索器中添加可训练的 Adapter 模块解决对齐问题
- **验证模块**：在检索文档后加入额外的验证模块评估检索到的文档与查询之间的相关性

22.3.6　RAG 新模式优化策略

RAG 的组织方法具有高度灵活性，能够根据特定问题的上下文对 RAG 流程中的模块进行替换或重新配置。两种组织模式：

- **增加或替换模块**：保留原有的检索-阅读结构，加入新模块以增强特定功能
- **调整模块间的工作流程**：加强语言模型与检索模型之间的互动

22.3.7　RAG 结合 SFT

RA-DIT 方法策略：

1. 更新 LLM 以最大限度地提高在给定检索增强指令的情况下正确答案的概率
2. 更新检索器以最大限度地减少文档与查询在语义上相似（相关）的程度

优点：通过这种方式使 LLM 更好地利用相关背景知识，并训练 LLM 即使在检索错误块的情况下也能产生准确的预测，使模型能够依赖自己的知识。

22.3.8　查询转换（Query Transformations）

动机：用户的 query 可能出现表述不清、需求复杂、内容无关等问题。

查询转换利用了大型语言模型（LLM）的强大能力，通过某种提示或方法将原始的用户问题转换或重写为更合适的、能够更准确地返回所需结果的查询。核心思想：用户的原始查询可能不总是最适合检索的，所以需要某种方式来改进或扩展它。

22.3.9　BERT 在 RAG 中的应用

RAG 中，对于一些传统任务（如分类、抽取等）用 BERT 效率会快很多，虽然会牺牲一点点效果，但是比起推理时间，前者更被容忍。而对于一些生成式任务（改写、摘要等），必须得用 LLMs，原因：

- BERT 窗口有限，只支持 512 个字符，对于生成任务远远不够
- LLMs 生成能力比 BERT 系列要强很多，此时时间换性能就变得很有意义

22.4　RAG 索引优化策略

22.4.1　嵌入优化策略

1. **微调嵌入**
    - 影响因素：影响到 RAG 的有效性
    - 目的：让检索到的内容与查询之间的相关性更加紧密
    - 作用：优化检索内容对最终输出的影响，特别是在处理专业领域
2. **动态嵌入（Dynamic Embedding）**

- 介绍：根据单词出现的上下文进行调整，为每个单词提供不同的向量表示

3. **检索后处理流程**
   - 动机：一次性向大语言模型展示所有相关文档可能会超出上下文窗口限制
   - 优化方法：重新排序、提示压缩、RAG 管道优化、混合搜索、递归检索与查询引擎等

22.4.2　检索召回率低解决方案

个人排查方式：

1. 知识库里面是否有对应答案？如果没有就是知识库覆盖不全问题
2. 知识库有，但是没召回：
   - 问题 1：知识库知识是否被分割掉导致召回出错？解决方法：修改分割方式或利用 BERT 进行上下句预测保证知识点完整性
   - 问题 2：知识没有被召回？分析 query 和 doc 的特点，建议先用 ES 做召回，然后用模型做精排

22.4.3　索引结构优化

构建 RAG 时，块大小是关键参数。找到最佳块大小是要找到正确的平衡。可以通过在测试集上运行评估并计算指标来找到最佳块大小。

22.4.4　混合检索提升效果

虽然向量搜索有助于检索语义相关块，但有时在匹配特定关键词方面缺乏精度。混合检索利用了矢量搜索和关键词搜索等不同检索技术的优势，将它们智能地结合起来。通过这种混合方法，可以匹配相关关键字，同时保持对查询意图的控制。

22.4.5　重新排名提升效果

当查询向量存储时，前 K 个结果不一定按最相关的方式排序。重新排名的简单概念是将最相关的信息重新定位到提示的边缘。例如，Diversity Ranker 专注于根据文档的多样性进行重新排序，而 LostInTheMiddleRanker 在上下文窗口的开始和结束之间交替放置最佳文档。

22.5　RAG 索引数据优化策略

22.5.1　提升索引数据质量

索引的数据决定了 RAG 答案的质量。优化方法：

- 通过删除重复/冗余信息，识别不相关的文档，检查事实的准确性

- 添加机制来更新过时的文档
- 清理特殊字符、奇怪的编码、不必要的 HTML 标签来消除文本噪音
- 通过主题提取、降维技术和数据可视化发现与主题无关的文档
- 使用相似性度量删除冗余文档

22.5.2　添加元数据提升效果

将元数据与索引向量结合使用有助于更好地构建它们，同时提高搜索相关性。元数据有用的情景：

- 时间维度：根据日期元数据进行排序
- 科学论文：将文章部分添加为每个块的元数据并进行过滤

22.5.3　输入查询与文档对齐

通过将块与它们回答的问题一起索引，优化与底层问题的相似性而不是与文档的相似性。具体方法：计算输入查询与问题（而非文档）的相似性，从而提高搜索相关性。

22.5.4　提示压缩提升效果

在检索上下文中的噪声会对 RAG 性能产生不利影响。解决方案：在检索后再应用后处理步骤，以压缩无关上下文，突出重要段落，减少总体上下文长度。选择性上下文等方法和 LLMLingua 使用小型 LLM 来计算即时互信息或困惑度，从而估计元素重要性。

22.5.5　查询重写和扩展

当用户查询结果不理想时，在送到 RAG 之前先发送给 LLM 重写此查询。这可以通过添加中间 LLM 调用实现。

22.6　RAG 未来发展方向

22.6.1　垂直优化

- RAG 中长上下文的处理问题
- RAG 的鲁棒性研究
- RAG 与微调（Fine-tuning）的协同作用
- RAG 的工程应用：提高检索效率和文档召回率，保障企业数据安全

22.6.2　水平扩展

从最初的文本问答领域出发，RAG 的应用逐渐拓展到更多模态数据，包括图像、代码、结构化知识、音视频等。

22.6.3　RAG 生态系统

- **下游任务和评估**：在开放式问题回答、事实验证等多种下游任务中表现优异
- **技术栈发展**：LangChain 和 LLamaIndex 提供丰富的 RAG 相关 API，新型技术栈不断涌现
- **专业领域应用**：医学、法律和教育等专业领域的知识问答
- **评估体系完善**：开发更精准的评估指标和框架，增强模型可解释性

第二十三章　大模型 (LLMs)RAG 关键痛点及解决方案

23.1　前言

受到 Barnett 等人的论文《Seven Failure Points When Engineering a Retrieval Augmented Generation System》的启发，本文将探讨论文中提到的七个痛点，以及在开发检索增强型生成 (RAG) 流程中常见的五个额外痛点。更为关键的是，我们将深入讨论这些 RAG 痛点的解决策略，使我们在日常 RAG 开发中能更好地应对这些挑战。

23.2　问题一：内容缺失问题

23.2.1　内容缺失问题介绍

当实际答案不在知识库中时，RAG 系统往往给出一个貌似合理却错误的答案，而不是承认无法给出答案。这导致用户接收到误导性信息，造成错误的引导。

23.2.2　内容缺失问题解决方案

1. **优化数据源：**” 输入什么，输出什么。” 如果源数据质量差，比如充斥着冲突信息，那么无论如何构建 RAG 流程，都不可能从杂乱无章的数据中得到有价值的结果。
2. **改进提示方式：**在知识库缺乏信息，系统可能给出错误答案的情况下，改进提示方式可以起到显著帮助。例如，通过设置提示” 如果你无法确定答案，请表明你不知道” 可以鼓励模型认识到自己的局限并更透明地表达不确定性。虽然无法保证百分百准确，但在优化数据源之后，改进提示方式是我们能做的最好努力之一。

23.3　问题二：错过排名靠前的文档

23.3.1　错过排名靠前的文档问题介绍

有时候系统在检索资料时，最关键的文件可能并没有出现在返回结果的最前面。这就导致了正确答案被忽略，系统因此无法给出精准的回答。即：” 问题的答案其实在某个文档里面，只是它没有获得足够高的排名以致于没能呈现给用户”。

23.3.2　错过排名靠前的文档问题解决方案

1. **重新排名检索结果：**在将检索到的结果发送给大型语言模型 (LLM) 之前，对结果进行重新排名可以显著提升 RAG 的性能。
2. **调整超参数：**chunk_size 和 similarity_top_k 都是用来调控 RAG 模型数据检索过程中效率和效果的参数。改动这些参数能够影响计算效率与信息检索质量之间的平衡。

```
param_tuner = ParamTuner(
    param_fn=objective_function_semantic_similarity,
    param_dict=param_dict,
    fixed_param_dict=fixed_param_dict,
    show_progress=True,
)

# 包含需要调优的参数
param_dict = {"chunk_size": [256, 512, 1024], "top_k": [1, 2, 5]}

# 包含在调整过程的所有运行中保持固定的参数
fixed_param_dict = {
    "docs": documents,
    "eval_qs": eval_qs,
    "ref_response_strs": ref_response_strs,
}

def objective_function_semantic_similarity(params_dict):
    chunk_size = params_dict["chunk_size"]
    docs = params_dict["docs"]
    top_k = params_dict["top_k"]
    eval_qs = params_dict["eval_qs"]
    ref_response_strs = params_dict["ref_response_strs"]

    # 建立索引
    index = build_index(chunk_size, docs)
```

```
    # 查询引擎
    query_engine = index.as_query_engine(similarity_top_k=top_k)
    # 获得预测响应
    pred_response_objs = get_responses(eval_qs, query_engine,
        show_progress=True)
    # 运行评估程序
    eval_batch_runner = _get_eval_batch_runner_semantic_similarity()
    eval_results = eval_batch_runner.evaluate_responses(
        eval_qs, responses=pred_response_objs, reference=
            ref_response_strs
    )
    # 获取语义相似度度量
    mean_score = np.array(
        [r.score for r in eval_results["semantic_similarity"]]
    ).mean()
    return RunResult(score=mean_score, params=params_dict)
```

23.4　问题三：脱离上下文—整合策略的限制

23.4.1　脱离上下文问题介绍

论文中提到了这样一个问题：”虽然数据库检索到了含有答案的文档，但这些文档并没有被用来生成答案。这种情况往往出现在数据库返回大量文档后，需要通过一个整合过程来找出答案”。

23.4.2　脱离上下文问题解决方案

1. **优化检索策略：**提供从基础到高级的检索策略，包括从每个索引进行基础检索、高级检索和搜索、自动检索、知识图谱检索器、组合/分层检索器等。
2. **微调嵌入模型：**如果使用开源嵌入模型，对其进行微调是提高检索准确性的有效方法。

```
finetune_engine = SentenceTransformersFinetuneEngine(
    train_dataset,
    model_id="BAAI/bge-small-en",
    model_output_path="test_model",
    val_dataset=val_dataset,
)

finetune_engine.finetune()
embed_model = finetune_engine.get_finetuned_model()
```

23.5　问题四：未能提取答案

23.5.1　未能提取答案问题介绍

当系统需要从提供的上下文中提取正确答案时，尤其是在信息量巨大时，系统往往会遇到困难。关键信息被遗漏，从而影响了回答的质量。论文中提到：” 这种情况通常是由于上下文中存在太多干扰信息或相互矛盾的信息”。

23.5.2　未能提取答案问题解决方案

1. **清理数据**：必须再次强调，干净整洁的数据至关重要！在质疑 RAG 流程之前，务必先要清理数据。
2. **提示压缩**：通过 LongLLMLingua 研究项目提出的提示压缩技术，在检索步骤之后压缩上下文，再将其输入大语言模型。

```
from llama_index.query_engine import RetrieverQueryEngine
from llama_index.response_synthesizers import CompactAndRefine
from llama_index.postprocessor import LongLLMLinguaPostprocessor
from llama_index.schema import QueryBundle

node_postprocessor = LongLLMLinguaPostprocessor(
    instruction_str="鉴于上下文，请回答最后一个问题",
    target_token=300,
    rank_method="longllmlingua",
    additional_compress_kwargs={
        "condition_compare": True,
        "condition_in_question": "after",
        "context_budget": "+100",
        "reorder_context": "sort",  # 启用文档重新排序
    },
)

retrieved_nodes = retriever.retrieve(query_str)
synthesizer = CompactAndRefine()
# 处理(压缩)、合成
new_retrieved_nodes = node_postprocessor.postprocess_nodes(
    retrieved_nodes, query_bundle=QueryBundle(query_str=query_str)
)
```

```
print("\n\n".join([n.get_content() for n in new_retrieved_nodes]))
response = synthesizer.synthesize(query_str, new_retrieved_nodes)
```

3. **LongContextReorder：**研究发现当关键信息位于输入上下文的开始或结尾时，通常能得到最好的性能。LongContextReorder 被设计用来重新排序检索到的节点，在需要大量 top-k 结果时特别有效。

```
from llama_index.postprocessor import LongContextReorder
reorder = LongContextReorder()
reorder_engine = index.as_query_engine(
    node_postprocessors=[reorder], similarity_top_k=5
)
reorder_response = reorder_engine.query("作者见过山姆•奥尔特曼吗?")
```

23.6　问题五：格式错误

23.6.1　格式错误问题介绍

当告诉计算机以某种特定格式（比如表格或清单）来整理信息，但大型语言模型 (LLM) 没能注意到时，就会出现格式错误。

23.6.2　格式错误问题解决方案

1. **更精准的提示：**让指令更加明确、简化问题并突出关键词、提供示例、循环提问不断细化问题。
2. **输出解析：**为任何查询提供格式化指南，对计算机的回答进行” 解析”。支持与 Guardrails 和 LangChain 提供的输出解析模块集成。

```
from llama_index import VectorStoreIndex, SimpleDirectoryReader
from llama_index.output_parsers import LangchainOutputParser
from llama_index.llms import OpenAI
from langchain.output_parsers import StructuredOutputParser,
    ResponseSchema

# 加载文档，构建索引
documents = SimpleDirectoryReader("../paul_graham_essay/data").
    load_data()
index = VectorStoreIndex.from_documents(documents)

# 定义输出模式
response_schemas = [
```

```
    ResponseSchema(
        name="Education",
        description="描述作者的教育经历/背景。",
    ),
    ResponseSchema(
        name="Work",
        description="描述作者的工作经验/背景。",
    ),
]

# 定义输出解析器
lc_output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
output_parser = LangchainOutputParser(lc_output_parser)

# 将输出解析器附加到LLM
llm = OpenAI(output_parser=output_parser)

# 获得结构化响应
from llama_index import ServiceContext
ctx = ServiceContext.from_defaults(llm=llm)
query_engine = index.as_query_engine(service_context=ctx)
response = query_engine.query("作者成长过程中做了哪些事情?")
print(str(response))
```

3. **Pydantic 程序：**将输入文字串转换成结构化的 Pydantic 对象，包括 LLM 文本完成 Pydantic 程序、LLM 函数调用 Pydantic 程序、预设的 Pydantic 程序。
4. **OpenAI JSON 模式：**设置 response_format 为"type": "json_object"，激活响应的 JSON 模式，确保生成能够被解析为有效 JSON 对象的字符串。

23.7　问题六：特异性错误

23.7.1　特异性错误问题介绍

有时候得到的回答可能缺少必要的细节或特定性，需要进一步提问来获取清晰的信息。有些答案可能过于含糊或泛泛，不能有效地满足用户的实际需求。

23.7.2　特异性错误问题解决方案

采用更高级的检索技巧来改善答案的详细程度：

- 从细节到全局的检索
- 围绕特定句子进行的检索
- 逐步深入的检索

23.8　问题七：回答不全面

23.8.1　回答不全面问题介绍

有时候得到的是部分答案，并不是错误的，但没有提供所有必要的细节，即便这些信息实际上是存在并且可以获取的。

23.8.2　回答不全面问题解决方案

查询优化：在简单的 RAG 模型中，比较性问题往往处理得不够好。加入查询理解层—在实际进行向量存储查询之前进行查询优化。四种查询优化方式：

- **路由优化**：保留原始查询内容，明确涉及的特定工具子集
- **查询改写**：保持选定工具不变，重新构思多种查询方式
- **细分问题**：将大问题拆分成几个小问题
- **ReAct Agent 工具选择**：根据原始查询确定使用哪个工具

```
# 使用HyDE查询转换运行查询
query_str = "what did paul graham do after going to RISD"
hyde = HyDEQueryTransform(include_original=True)
query_engine = index.as_query_engine()
query_engine = TransformQueryEngine(query_engine, query_transform=hyde)
response = query_engine.query(query_str)
print(response)
```

23.9　问题八：数据处理能力的挑战

23.9.1　数据处理能力挑战介绍

在 RAG 技术流程中，处理大量数据时常会遇到系统若无法高效地管理和加工这些数据，就可能导致性能瓶颈甚至系统崩溃。这种处理能力上的挑战可能会让数据处理的时间大幅拉长，系统超负荷运转，数据质量下降，以及服务的可用性降低。

23.9.2　数据处理能力挑战解决方案

并行技术：推出数据处理的并行技术，能够使文档处理速度最多提升 15 倍。

```
# 加载数据
documents = SimpleDirectoryReader(input_dir="./data/source_files").
    load_data()

# 创建带有转换的管道
pipeline = IngestionPipeline(
    transformations=[
        SentenceSplitter(chunk_size=1024, chunk_overlap=20),
        TitleExtractor(),
        OpenAIEmbedding(),
    ]
)

# 将num_workers设置为大于1的值将调用并行执行
nodes = pipeline.run(documents=documents, num_workers=4)
```

23.10　问题九：结构化数据查询的难题

23.10.1　结构化数据查询难题介绍

用户在查询结构化数据时，精准地获取想要的信息是一项挑战，尤其是遇到复杂或含糊的查询条件时。当前的大语言模型在这方面还存在局限，例如无法灵活地将自然语言转换为 SQL 查询语句。

23.10.2　结构化数据查询难题解决方案

1. **Chain-of-table Pack：**将链式思考与表格的转换和表述相结合，通过一系列规定的操作逐步变换表格，并在每一步向大语言模型展示新变化的表格。
2. **Mix-Self-Consistency Pack：**通过自治机制（多数投票）聚合文本和符号推理的结果。

```
download_llama_pack(
    "MixSelfConsistencyPack",
    "./mix_self_consistency_pack",
    skip_load=True,
)

query_engine = MixSelfConsistencyQueryEngine(
    df=table,
    llm=llm,
```

```
    text_paths=5,       # 抽样5条文本推理路径
    symbolic_paths=5,   # 抽样5个符号推理路径
    aggregation_mode="self-consistency",  # 通过自治聚合结果
    verbose=True,
)

response = await query_engine.aquery(example["utterance"])
```

23.11　问题十：从复杂 PDF 文件中提取数据

23.11.1　复杂 PDF 数据提取问题介绍

处理 PDF 文件时，需要从里面复杂的表格中提取出数据来回答问题，但简单的检索方法做不到这一点，需要更高效的技术。

23.11.2　复杂 PDF 数据提取问题解决方案

嵌入式表格检索：提供 EmbeddedTablesUnstructuredRetrieverPack 工具包，从 HTML 文档中解析出嵌入的表格，把它们组织成清晰的结构图，然后根据用户问题找出并获取相关表格的数据。

```
# 下载和安装依赖项
EmbeddedTablesUnstructuredRetrieverPack = download_llama_pack(
    "EmbeddedTablesUnstructuredRetrieverPack",
    "./embedded_tables_unstructured_pack",
)

# 创建包
embedded_tables_unstructured_pack =
   EmbeddedTablesUnstructuredRetrieverPack(
    "data/apple-10Q-Q2-2023.html",  # 接收html文件
    nodes_save_path="apple-10-q.pkl"
)

# 运行包
response = embedded_tables_unstructured_pack.run("总运营费用是多少?").
   response
display(Markdown(f"{response}"))
```

23.12　问题十一：备用模型

23.12.1　备用模型问题介绍

在使用大型语言模型时，可能会担心如果模型出了问题怎么办，比如遇到了 OpenAI 模型的使用频率限制。这时候就需要一个或多个备用模型以防万一主模型出现故障。

23.12.2　备用模型问题解决方案

1. **Neutrino 路由器：**一个大语言模型的集合，把问题发送到这里，用预测模型判断哪个大语言模型最适合处理问题。

```
from llama_index.llms import Neutrino
from llama_index.llms import ChatMessage

llm = Neutrino(
    api_key="<your-Neutrino-api-key>",
    router="test"  # 在Neutrino仪表板中配置的"测试"路由器
)

response = llm.complete("什么是大语言模型?")
print(f"Optimal model: {response.raw['model']}")
```

2. **OpenRouter：**统一的接口，可以访问任何大语言模型，自动找到最便宜的模型，并在主服务器出现问题时提供备选方案。

```
from llama_index.llms import OpenRouter
from llama_index.llms import ChatMessage

llm = OpenRouter(
    api_key="<your-OpenRouter-api-key>",
    max_tokens=256,
    context_window=4096,
    model="gryphe/mythomax-12-13b",
)

message = ChatMessage(role="user", content="Tell me a joke")
resp = llm.chat([message])
print(resp)
```

23.13　问题十二：大语言模型 (LLM) 的安全挑战

23.13.1　LLM 安全挑战介绍

面对如何防止恶意输入操控、处理潜在的不安全输出和避免敏感信息泄露等问题，每位 AI 架构师和工程师都需要找到解决方案。

23.13.2　LLM 安全挑战解决方案

Llama Guard：以 7-B Llama 2 为基础，旨在对大语言模型进行内容分类，通过对输入的提示进行分类和对输出的响应进行分类来工作。能够产生文本结果，判断特定的输入提示或输出响应是否安全。如果根据规则识别出内容不安全，还会指出违反的具体规则子类别。

23.14　总结

表 23.1: RAG 痛点及解决方案总结

序号	痛点	解决方案
1	内容缺失	清洗数据 & 改善提示方法
2	错过排名靠前的文档	超参数调优 & 文档重新排序
3	上下文不连贯-综合策略限制	调整检索策略 & 微调嵌入向量
4	未提取到信息	清洗数据, 压缩提示,& 长上下文重排序
5	格式错误	改善提示方法, 输出解析,pydantic 程序设计,& OpenAI 的 JSON 模式
6	特异性错误	高级检索策略
7	响应不完整	查询优化
8	数据摄取的可扩展性	并行化摄取流程
9	结构化数据问答	Chain-of-table Pack& Mix-Self-Consistency Pack
10	从复杂 PDF 文件中提取数据	嵌入式表格检索
11	后备模型	Neutrino 路由器 & OpenRouter
12	大语言模型 (LLM) 安全性	Llama Guard 守护程序

我们讨论了开发 RAG 应用时的 12 个痛点（论文中的 7 个加上另外 5 个），并为它们每一个都提供了相应的解决方案。这些解决方案涵盖了从数据预处理到查询优化，从模型选择到安全防护的各个方面，为构建高质量的 RAG 系统提供了全面的指导。

第二十四章　大模型 (LLMs) RAG 优化策略 RAG-Fusion 篇

24.1　RAG 技术概述

24.1.1　RAG 的优点

1. **向量搜索融合**：RAG 通过将向量搜索功能与生成模型相结合，引入了一种新颖的范式。这种融合使大型语言模型 (LLM) 能够生成更丰富、更具上下文意识的输出。
2. **减少幻觉现象**：RAG 显著降低了 LLM 产生幻觉的倾向，使生成的文本更加基于数据。
3. **个人和专业效用**：从个人应用（如浏览笔记）到更专业的集成，RAG 在提高生产力和内容质量方面展示了其多功能性，同时基于可信的数据来源。

24.1.2　RAG 的局限性

1. **当前搜索技术的限制**：RAG 受到限制的方面与我们的检索式基于词汇和向量的搜索技术相同。
2. **人类搜索效率低下**：人类在向搜索系统输入他们想要的内容时并不擅长，如打字错误、含糊的查询或词汇有限，这常常导致错过那些超出显而易见的顶部搜索结果的大量信息。虽然 RAG 有所帮助，但它并没有完全解决这个问题。
3. **搜索的过度简化**：我们普遍的搜索范式是将查询线性映射到答案，缺乏理解人类查询的多维性。这种线性模型通常无法捕捉更复杂用户查询的细微差别和上下文，导致结果相关性较低。

24.2　RAG-Fusion 技术概述

24.2.1　为什么需要 RAG-Fusion？

RAG-Fusion 旨在解决 RAG 固有的限制，通过生成多个用户查询并重新排序结果。利用逆向排名融合和自定义向量评分加权进行综合、准确的搜索。

RAG-Fusion 旨在弥合用户明确询问与他们意图询问之间的差距，更接近于发现通常隐藏的变革性知识。

24.2.2　RAG-Fusion 核心技术

RAG-Fusion 的基础三元组与 RAG 相似，核心技术包括：

- **通用编程语言**：通常是 Python
- **专用向量搜索数据库**：如 Elasticsearch 或 Pinecone，用于驱动文档检索
- **强大的大型语言模型**：如 ChatGPT，用于创造文本

与 RAG 不同的是，RAG-Fusion 通过几个额外的步骤区分自己：查询生成和结果重新排序。

24.3　RAG-Fusion 工作流程

24.3.1　多查询生成

为什么要生成多个查询?

在传统的搜索系统中，用户通常输入一个查询来查找信息。虽然这种方法直接简单，但它有局限性。单一查询可能无法完全捕捉用户感兴趣的全部范围，或者可能过于狭窄而无法产生全面的结果。因此，从不同角度生成多个查询就显得尤为重要。

多查询生成技术实现（提示工程）

利用提示工程和自然语言模型拓宽搜索视野，提升结果质量。利用提示工程生成多个查询至关重要，这些查询不仅与原始查询相似，还提供不同的视角或角度。

```
def generate_queries(original_query, num_queries=5):
    """
    基于原始查询生成多个相关查询
    """
    system_message = """
    你是一个AI助手，负责从不同角度生成搜索查询。
    请基于用户提供的原始查询，生成{num_queries}个相关的搜索查询。
    这些查询应该：
    1. 与原始查询语义相关但角度不同
    2. 涵盖原始查询的不同方面
    3. 具有一定的多样性和覆盖面
    """

    prompt = f"""
```

```
    原始查询: {original_query}
    请生成{num_queries}个相关的搜索查询:
    """

    # 调用LLM生成多个查询
    response = llm.chat_complete(
        system=system_message,
        messages=[{"role": "user", "content": prompt}]
    )

    # 解析生成的查询
    generated_queries = parse_generated_queries(response)
    return [original_query] + generated_queries  # 包含原始查询
```

多查询生成工作原理

1. **调用语言模型：**该函数调用一个语言模型（如 chatGPT）。该方法期望一个特定的指令集，通常描述为” 系统消息”，以指导模型。例如，系统消息指导模型充当”AI 助手”。
2. **自然语言查询：**模型基于原始查询生成多个查询。
3. **多样性和覆盖范围：**这些查询不是随机变化，而是经过精心生成的，以提供原始问题的不同视角。例如，如果原始查询是关于” 气候变化的影响”，那么生成的查询可能包括” 气候变化的经济后果”、” 气候变化与公共健康” 等角度。这种方法确保了搜索过程考虑了更广泛的信息范围，从而提高生成总结的质量和深度。

24.3.2　逆向排名融合（RRF）

为什么选择 RRF？

逆向排名融合（RRF）是一种将多个搜索结果列表的排名结合起来产生单一统一排名的技术。该技术由滑铁卢大学（加拿大）和谷歌合作开发，根据其作者的说法，” 产生的结果比任何单个系统更好，也比标准重新排名方法更好”。

RRF 算法的数学表达式为：

$$RRFscore(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

其中 $k = 60$ 是一个常数，$r(d)$ 是文档 d 在排名 r 中的位置。

通过结合不同查询的排名，我们增加了最相关文档出现在最终列表顶部的机会。RRF 特别有效，因为它不依赖于搜索引擎分配的绝对分数，而是依赖于相对排名，使其非常适合结合可能具有不同规模或分数分布的查询结果。

通常情况下，RRF 被用于混合词汇和向量结果。虽然这种方法有助于弥补向量搜索在查找特定术语（例如缩写）时的不足，但对结果并不印象深刻，这些结果往往更像是多个结果集的拼凑，因为同一个查询的词汇和向量搜索很少出现相同的结果。

可以将 RRF 看作是那种坚持在做决定前听取每个人意见的人。只不过在这种情况下，它不仅不烦人，而且有帮助。众多观点越多，结果越准确。

RRF 技术实现

运用 RRF 根据多组搜索结果的位置重新排序文档。逆向排名融合位置重新排序系统：

1. 函数 reciprocal_rank_fusion 接收一个搜索结果的字典，其中每个键是一个查询，相应的值是根据该查询的相关性排名的文档 ID 列表。
2. RRF 算法然后基于其在不同列表中的排名为每个文档计算一个新分数，并根据这些分数排序以创建最终的重新排名列表。
3. 计算完融合分数后，函数按照这些分数的降序对文档进行排序，以获得最终的重新排名列表，然后返回该列表。

```
def reciprocal_rank_fusion(search_results_dict, k=60):
    """
    实现逆向排名融合算法
    search_results_dict: 字典，键为查询，值为文档ID排名列表
    k: 平滑常数，通常设为60
    """
    # 初始化文档得分字典
    doc_scores = {}

    # 对每个查询的排名结果进行处理
    for query, ranked_docs in search_results_dict.items():
        for rank, doc_id in enumerate(ranked_docs):
            if doc_id not in doc_scores:
                doc_scores[doc_id] = 0
            # 计算RRF分数: 1/(k + rank)
            doc_scores[doc_id] += 1.0 / (k + rank + 1)

    # 按分数降序排序
    fused_ranking = sorted(doc_scores.items(),
                           key=lambda x: x[1], reverse=True)

    return fused_ranking

# 示例使用
```

```
search_results = {
    "query1": ["doc1", "doc3", "doc2", "doc5"],
    "query2": ["doc2", "doc1", "doc4", "doc3"],
    "query3": ["doc3", "doc1", "doc5", "doc2"]
}

final_ranking = reciprocal_rank_fusion(search_results)
print("融合后的排名:", final_ranking)
```

生成性输出用户意图保留

使用多个查询的一个挑战是可能稀释用户的原始意图。为了缓解这一点，我们指示模型在提示工程中更重视原始查询。

生成性输出用户意图保留技术实现

最后，将重新排名的文档和所有查询输入到 LLM 提示中，以生成典型的 RAG 方式的生成性输出，如请求回应或摘要。

```
def generate_final_response(original_query, all_queries, reranked_docs):
    """
    基于重新排名的文档生成最终响应
    """
    # 构建包含原始查询优先的提示
    prompt = f"""
    基于以下文档内容，回答用户的原始问题。

    原始问题: {original_query}
    相关查询: {', '.join(all_queries[1:])}

    相关文档内容:
    {chr(10).join([doc['content'] for doc in reranked_docs[:5]])}

    请优先考虑原始问题的意图，提供准确、全面的回答。
    """

    response = llm.generate(prompt)
    return response
```

通过将这些技术和技巧层叠起来，RAG-Fusion 提供了一种强大而细腻的文本生成方法。它利用搜索技术和生成性人工智能的最佳特性，产生高质量、可靠的输出。

24.4　RAG-Fusion 的优势和挑战

24.4.1　RAG-Fusion 优势

1. **更优质的源材料**：使用 RAG-Fusion 时，搜索深度不仅仅是” 增强” 而是被放大。重新排名的相关文档列表意味着不只是在信息表面刮刮而已，而是潜入观点的海洋。结构化输出易于阅读，直观上可信赖，这在对人工智能生成内容持怀疑态度的世界中至关重要。
2. **增强用户意图对齐**：RAG-Fusion 的核心设计是作为一个富有同情心的人工智能，揭示用户努力表达但可能无法清晰表述的内容。采用多查询策略捕捉用户信息需求的多面性表现，因此提供全面的输出，并与用户意图产生共鸣。
3. **结构化、富有洞见的输出**：通过汲取多样化的信息源，模型制作出组织良好且富有洞见的答案，预测后续问题并主动解答。
4. **自动纠正用户查询**：该系统不仅解释，还优化用户查询。通过生成多个查询变体，RAG-Fusion 执行隐含的拼写和语法检查，从而提高搜索结果的准确性。
5. **处理复杂查询**：人类语言在表达复杂或专业思想时常常出现障碍。该系统作为语言催化剂，生成可能包含所需专业术语或术语的变体，用于更集中和相关的搜索结果。它还可以将更长、更复杂的查询分解成向量搜索可以处理的更小、更易管理的部分。
6. **搜索中的意外发现**：考虑” 未知的未知” 直到遇到才知道需要的信息。通过采用更广泛的查询范围，系统促进了发现意外信息的可能性。虽然这些信息并非明确寻求，但对用户来说却可能是一个” 恍然大悟” 的时刻。这使 RAG-Fusion 区别于其他传统搜索模型。

24.4.2　RAG-Fusion 挑战

1. **过于冗长的风险**：RAG-Fusion 的深度有时可能导致信息泛滥。输出可能过于详细，令人不堪重负。可以将 RAG-Fusion 比作那个解释过多的朋友——信息丰富，但有时可能需要更直接了当。
2. **平衡上下文窗口**：多查询输入和多样化文档集的引入可能会使语言模型的上下文窗口受到压力。想象一个舞台上挤满了演员，使得剧情难以跟进。对于上下文限制较紧的模型，这可能导致输出不连贯甚至被截断。
3. **伦理和用户体验考虑**：拥有巨大力量的同时也伴随着巨大的责任。对于 RAG-Fusion 来说，操作用户查询以改善结果的能力似乎正踏入某种道德灰区。在改善搜索结果的同时平衡用户意图的完整性至关重要。

伦理和用户体验具体考虑

- **伦理顾虑**：
  - **用户自主性**：操作用户查询有时可能偏离原始意图。考虑向人工智能让渡多少控制权以及代价是什么非常重要。

  - **透明度**：不仅仅是关于更好的结果；如果用户的查询被调整，他们应当意识到这一点。这种透明度对于维护信任和尊重用户意图至关重要。
- **用户体验 (UX) 增强**：
  - **保留原始查询**：RAG-Fusion 优先考虑初始用户查询，确保其在生成过程中的重要性。这作为防止误解的保障。
  - **过程可见性**：展示生成的查询以及最终结果，为用户提供搜索范围和深度的透明视图。这有助于建立信任和理解。
- **UX/UI 实施建议**：
  - **用户控制**：提供用户切换 RAG-Fusion 的选项，允许他们在手动控制和增强的人工智能辅助之间选择。
  - **指导和清晰度**：关于 RAG-Fusion 工作方式的工具提示或简要说明可以帮助设定明确的用户期望。

24.5　完整 RAG-Fusion 实现示例

```
class RAGFusionSystem:
    def __init__(self, vector_db, llm, k=60):
        self.vector_db = vector_db
        self.llm = llm
        self.k = k  # RRF常数

    def generate_queries(self, original_query, num_queries=5):
        """生成多个相关查询"""
        prompt = f"""
        基于以下原始查询，生成{num_queries}个相关的搜索查询：
        原始查询：{original_query}

        要求：
        1. 每个查询应该从不同角度探讨原始主题
        2. 保持语义相关性但提供多样性
        3. 每个查询不超过15个词
        """

        response = self.llm.generate(prompt)
        queries = self._parse_generated_queries(response)
        return [original_query] + queries

    def search_all_queries(self, queries, top_k=10):
```

```
        """对每个查询执行搜索"""
        all_results = {}

        for query in queries:
            results = self.vector_db.similarity_search(query, k=top_k)
            all_results[query] = [doc.doc_id for doc in results]

        return all_results

    def reciprocal_rank_fusion(self, search_results):
        """执行逆向排名融合"""
        doc_scores = {}

        for query, ranked_docs in search_results.items():
            for rank, doc_id in enumerate(ranked_docs):
                if doc_id not in doc_scores:
                    doc_scores[doc_id] = 0
                doc_scores[doc_id] += 1.0 / (self.k + rank + 1)

        # 按分数排序
        fused_ranking = sorted(doc_scores.items(),
                               key=lambda x: x[1], reverse=True)
        return fused_ranking

    def retrieve_documents(self, fused_ranking, top_n=5):
        """根据融合排名检索实际文档内容"""
        doc_ids = [doc_id for doc_id, score in fused_ranking[:top_n]]
        documents = self.vector_db.get_documents(doc_ids)
        return documents

    def generate_final_response(self, original_query, queries, documents)
        :
        """生成最终响应"""
        context = "\n\n".join([doc.content for doc in documents])

        prompt = f"""
        基于以下上下文信息，回答用户的原始问题。

        原始问题: {original_query}
        生成的辅助查询: {', '.join(queries[1:])}
```

```
63
64          上下文信息:
65          {context}
66
67          请提供:
68          1. 直接回答原始问题
69          2. 涵盖相关的重要方面
70          3. 保持回答的准确性和全面性
71          4. 如果信息不足, 请明确指出
72          """
73
74          response = self.llm.generate(prompt)
75          return response
76
77      def query(self, original_query, num_queries=5, top_k=10, top_n=5):
78          """完整的RAG-Fusion查询流程"""
79          # 1. 生成多个查询
80          queries = self.generate_queries(original_query, num_queries)
81
82          # 2. 对每个查询执行搜索
83          search_results = self.search_all_queries(queries, top_k)
84
85          # 3. 逆向排名融合
86          fused_ranking = self.reciprocal_rank_fusion(search_results)
87
88          # 4. 检索文档内容
89          documents = self.retrieve_documents(fused_ranking, top_n)
90
91          # 5. 生成最终响应
92          response = self.generate_final_response(original_query, queries,
                documents)
93
94          return {
95              'original_query': original_query,
96              'generated_queries': queries,
97              'fused_ranking': fused_ranking,
98              'documents': documents,
99              'response': response
100         }
101
```

```
# 使用示例
rag_fusion = RAGFusionSystem(vector_db=pinecone_index, llm=chatgpt)
result = rag_fusion.query("气候变化对全球经济的影响")
print("最终回答:", result['response'])
print("生成的查询:", result['generated_queries'])
print("使用的文档数量:", len(result['documents']))
```

24.6　总结

RAG-Fusion 通过引入多查询生成和逆向排名融合技术，显著提升了传统 RAG 系统的性能。其主要优势在于能够从多个角度理解用户查询意图，通过融合不同视角的搜索结果提供更全面、准确的回答。

关键技术特点包括：

- **多视角查询生成**：通过 LLM 生成多个相关但角度不同的查询
- **逆向排名融合**：智能融合不同查询的搜索结果
- **用户意图保留**：在增强搜索的同时保持对原始查询的忠实
- **意外发现能力**：通过广泛搜索范围促进意外有价值信息的发现

尽管存在计算复杂度增加和伦理考量等挑战，但通过合理的工程实现和用户体验设计，RAG-Fusion 为构建更智能、更全面的检索增强生成系统提供了有力的技术路径。

第二十五章　基于知识图谱的大模型检索增强实现策略：Graph RAG

25.1　引言：为什么需要 Graph RAG？

虽然 LlamaIndex 能够利用摘要索引进行增强的方案，但这些方案都是利用非结构化文本来实现的。对于知识图谱，是否可以将其作为一种检索路径来提高检索的相关性，这就引出了 Graph RAG 的概念。

知识图谱可以减少基于嵌入的语义搜索所导致的不准确性。例如：” 保温大棚” 与” 保温杯”，尽管在语义上两者存在相关性，但在大多数场景下，这种通用语义（Embedding）下的相关性很高，可能作为错误的上下文而引入” 幻觉”。这时候，可以利用领域知识的知识图谱来缓解这种幻觉问题。

25.2　Graph RAG 基本概念

25.2.1　什么是 Graph RAG？

Graph RAG（Retrieval-Augmented Generation）是一种基于知识图谱的检索增强技术。它通过构建图模型的知识表达，将实体和关系之间的联系用图的形式进行展示，然后利用大语言模型（LLM）进行检索增强。

25.2.2　Graph RAG 的核心思路

Graph RAG 将知识图谱等价于一个超大规模的词汇表，而实体和关系则对应于单词。通过这种方式，Graph RAG 在检索时能够将实体和关系作为单元进行联合建模。

Graph RAG 的基本思想是：对用户输入的 query 提取实体，然后构造子图形成上下文，最后送入大模型完成生成。

25.3　Graph RAG 技术架构

25.3.1　整体工作流程

Graph RAG 的工作流程包含三个主要步骤：

1. **实体提取**：使用 LLM（或其他）模型从问题中提取关键实体
2. **子图检索**：根据提取的实体检索相关子图，深入到一定的深度（如 2 度或更多）
3. **答案生成**：利用获得的上下文利用 LLM 产生答案

通过这种方式，知识图谱召回可以作为一路检索路径与传统的向量检索进行融合。

25.3.2　代码实现框架

```
class GraphRAGSystem:
    def __init__(self, kg_store, llm, entity_extractor):
        self.kg_store = kg_store  # 知识图谱存储
        self.llm = llm            # 大语言模型
        self.entity_extractor = entity_extractor  # 实体提取器

    def extract_entities(self, query):
        """从查询中提取实体"""
        # 使用LLM或实体识别模型提取实体
        entities = self.entity_extractor.extract(query)
        return entities

    def retrieve_subgraph(self, entities, max_depth=2):
        """根据实体检索子图"""
        subgraph = self.kg_store.get_subgraph(
            entities=entities,
            max_depth=max_depth
        )
        return subgraph

    def generate_answer(self, query, subgraph):
        """基于子图生成答案"""
        # 将子图转换为文本上下文
        context = self.subgraph_to_text(subgraph)

        # 构建提示
        prompt = f"""
        基于以下知识图谱信息回答问题：
```

```
29
30          知识图谱上下文:
31          {context}
32
33          问题: {query}
34
35          请根据提供的知识图谱信息给出准确的回答。
36          """
37
38          # 使用LLM生成答案
39          answer = self.llm.generate(prompt)
40          return answer
41
42      def query(self, query, max_depth=2):
43          """完整的Graph RAG查询流程"""
44          # 1. 实体提取
45          entities = self.extract_entities(query)
46          print(f"提取的实体: {entities}")
47
48          # 2. 子图检索
49          subgraph = self.retrieve_subgraph(entities, max_depth)
50          print(f"检索到的子图包含 {len(subgraph.nodes)} 个节点和 {len(
                subgraph.edges)} 条边")
51
52          # 3. 答案生成
53          answer = self.generate_answer(query, subgraph)
54          return answer
```

25.4　Graph RAG 具体实现

25.4.1　实体提取技术

实体提取是 Graph RAG 的第一步，需要从用户查询中准确识别出关键实体。常用的实体提取方法包括：

```
1 import spacy
2 import re
3 from typing import List
4
5 class EntityExtractor:
```

```
    def __init__(self, model_name="zh_core_web_sm"):
        self.nlp = spacy.load(model_name)

    def extract_with_llm(self, query: str) -> List[str]:
        """使用LLM进行实体提取"""
        prompt = f"""
        从以下问题中提取关键实体（人名、地名、机构名、专业术语等）：
        问题: "{query}"

        请以JSON格式返回实体列表:
        {{"entities": ["实体1", "实体2", ...]}}
        """

        response = llm.generate(prompt)
        # 解析JSON响应
        entities = self._parse_llm_response(response)
        return entities

    def extract_with_ner(self, query: str) -> List[str]:
        """使用命名实体识别提取实体"""
        doc = self.nlp(query)
        entities = []
        for ent in doc.ents:
            if ent.label_ in ["PERSON", "ORG", "GPE", "PRODUCT", "EVENT"
                ]:
                entities.append(ent.text)
        return entities

    def hybrid_extract(self, query: str) -> List[str]:
        """混合实体提取方法"""
        # 先用NER提取
        ner_entities = self.extract_with_ner(query)

        # 如果NER提取结果不足，使用LLM补充
        if len(ner_entities) < 2:
            llm_entities = self.extract_with_llm(query)
            # 去重合并
            all_entities = list(set(ner_entities + llm_entities))
            return all_entities

```

```
        return ner_entities
```

25.4.2　子图检索策略

子图检索需要根据提取的实体在知识图谱中检索相关的子图结构：

```
class KnowledgeGraphStore:
    def __init__(self, graph_db_connection):
        self.conn = graph_db_connection

    def get_subgraph(self, entities: List[str], max_depth: int = 2):
        """检索包含实体的子图"""
        # 构建Cypher查询
        query = self._build_cypher_query(entities, max_depth)

        # 执行查询
        result = self.conn.run(query)
        subgraph = self._parse_cypher_result(result)
        return subgraph

    def _build_cypher_query(self, entities: List[str], max_depth: int) ->
        str:
        """构建Cypher查询语句"""
        entity_conditions = " OR ".join([f"n.name = '{e}'" for e in
            entities])

        cypher_query = f"""
        MATCH path = (start)-[*1..{max_depth}]-(end)
        WHERE {entity_conditions}
        WITH nodes(path) as nodes, relationships(path) as rels
        RETURN nodes, rels
        LIMIT 100
        """
        return cypher_query

    def _parse_cypher_result(self, result):
        """解析Cypher查询结果"""
        subgraph = {
            "nodes": set(),
            "edges": set()
```

```
        }

        for record in result:
            # 处理节点
            for node in record["nodes"]:
                node_id = node.id
                node_props = dict(node)
                subgraph["nodes"].add((node_id, node_props))

            # 处理关系
            for rel in record["rels"]:
                rel_id = rel.id
                rel_type = rel.type
                rel_props = dict(rel)
                subgraph["edges"].add((
                    rel.start_node.id,
                    rel.end_node.id,
                    rel_type,
                    rel_props
                ))

        return subgraph
```

25.5　Graph RAG 应用示例

25.5.1　示例 1：人物信息查询

当用户输入”tell me about Peter Quill” 时，Graph RAG 的处理流程如下：

1. **实体提取**：识别关键词 [”Peter”, ”Quill”]
2. **子图检索**：编写 Cypher 语句获得二跳结果
3. **答案生成**：基于检索到的子图信息生成回答

25.5.2　示例 2：机构事件查询

用户输入：”Tell me events about NASA”

```
> 提取的关键词: ['NASA', 'events']
> 召回的2度关系子图:
```

```
Extracted relationships: The following are knowledge triplets in max
    depth 2 in the form of subject[predicate, object, predicate_next_hop,
    object_next_hop...]

nasa['public release date', 'mid-2023']
nasa['announces', 'future space telescope programs']
nasa['publishes images of', 'debris disk']
nasa['discovers', 'exoplanet lhs 475 b']
```

25.5.3　完整示例代码

```
def graph_rag_demo():
    """Graph RAG完整示例"""
    # 初始化组件
    kg_store = Neo4jGraphStore(uri="bolt://localhost:7687",
                               user="neo4j",
                               password="password")
    llm = OpenAI(model="gpt-4")
    entity_extractor = EntityExtractor()

    # 创建Graph RAG系统
    graph_rag = GraphRAGSystem(kg_store, llm, entity_extractor)

    # 示例查询
    queries = [
        "Tell me about Peter Quill",
        "Tell me events about NASA",
        "莫妮卡•贝鲁奇的代表作是什么？"
    ]

    for query in queries:
        print(f"\n=== 查询: {query} ===")

        # 执行Graph RAG查询
        result = graph_rag.query(query, max_depth=2)

        print(f"提取的实体: {graph_rag.entities}")
        print(f"生成的答案: {result}")

```

```
        # 记录token使用情况
        print("Token使用统计:")
        print(f"LLM总token使用: 159 tokens")
        print(f"Embedding总token使用: 0 tokens")

# 运行示例
if __name__ == "__main__":
    graph_rag_demo()
```

25.6　Graph RAG 排序优化策略

基于知识图谱召回的方法可以和其他召回方法融合，但在图谱规模很大时，单纯的子图检索可能存在优化空间。

25.6.1　现有方法的局限性

- **子图召回的多条路径中可能会出现不相关的路径**
- **实体识别阶段的精度有限**，采用关键词提取方法比较直接，效果有待提升
- **依赖于基础知识图谱库的质量**，如果数据量或覆盖范围不足，可能引入噪声

25.6.2　两阶段排序优化

为了提升 Graph RAG 的效果，可以采用先粗排后精排的两阶段排序策略：

粗排阶段（LightGBM 模型）

在粗排阶段，根据问题 query 和候选路径 path 的特征，使用 LightGBM 机器学习模型对候选路径进行初步排序，保留 top N 条路径。

使用的特征包括：

- 字符重合数
- 词重合数
- 编辑距离
- path 跳数
- path 长度
- 字符的 Jaccard 相似度
- 词语的 Jaccard 相似度
- path 中的关系数
- path 中的实体个数
- path 中的答案个数

- 判断 path 的字符是否全在 query 中
- 判断 query 和 path 中是否都包含数字
- 获取数字的 Jaccard 相似度

精排阶段（预训练语言模型）

在精排阶段，采用预训练语言模型计算 query 和粗排阶段 path 的语义匹配度，选择得分 top 2-3 的答案路径作为最终结果。

```
class GraphRAGRanker:
    def __init__(self, lightgbm_model, sentence_transformer):
        self.lgb_model = lightgbm_model
        self.st_model = sentence_transformer

    def coarse_ranking(self, candidate_paths, query, top_k=50):
        """粗排阶段"""
        features = self.extract_features(candidate_paths, query)
        scores = self.lgb_model.predict(features)

        # 按分数排序并返回top K
        ranked_indices = np.argsort(scores)[::-1][:top_k]
        return [candidate_paths[i] for i in ranked_indices]

    def fine_ranking(self, coarse_paths, query, top_k=3):
        """精排阶段"""
        # 计算语义相似度
        query_embedding = self.st_model.encode([query])
        path_embeddings = self.st_model.encode([str(path) for path in
            coarse_paths])

        similarities = cosine_similarity(query_embedding, path_embeddings
            )[0]

        # 选择最相似的路径
        top_indices = np.argsort(similarities)[::-1][:top_k]
        return [coarse_paths[i] for i in top_indices]

    def extract_features(self, paths, query):
        """提取排序特征"""
        features = []
        for path in paths:
```

```
            feature_vector = []

            # 字符级别特征
            path_str = str(path)
            feature_vector.append(len(set(query) & set(path_str)))  # 字符重合数
            feature_vector.append(editdistance.eval(query, path_str))  # 编辑距离

            # 词级别特征
            query_words = set(query.split())
            path_words = set(path_str.split())
            feature_vector.append(len(query_words & path_words))  # 词重合数
            feature_vector.append(jaccard_similarity(query_words, path_words))  # Jaccard相似度

            # 图结构特征
            feature_vector.append(len(path.nodes))  # 实体个数
            feature_vector.append(len(path.edges))  # 关系数
            feature_vector.append(self.get_path_depth(path))  # 路径深度

            features.append(feature_vector)

        return np.array(features)
```

25.7　Graph RAG 的优势与挑战

25.7.1　技术优势

- **结构化知识利用**：能够充分利用知识图谱中的结构化信息
- **减少语义歧义**：通过实体关系明确语义，减少 Embedding 相似性带来的歧义
- **可解释性强**：检索路径清晰，结果可追溯
- **领域适应性好**：特别适合领域知识丰富的场景

25.7.2　面临挑战

- **知识图谱构建成本高**：需要高质量的知识图谱数据
- **实体识别精度依赖**：实体提取的准确性直接影响后续效果

- **子图检索效率**：大规模图谱上的子图检索需要优化
- **多源信息融合**：如何与文本检索等其他检索方式有效融合

25.7.3　未来发展方向

- **自动化知识图谱构建**：结合大语言模型自动构建和更新知识图谱
- **多模态 Graph RAG**：支持文本、图像等多模态知识的图谱检索
- **动态图谱学习**：支持实时更新和增量学习的图谱系统
- **跨语言 Graph RAG**：支持多语言知识图谱的检索增强

25.8　总结

Graph RAG 作为一种基于知识图谱的检索增强生成技术，通过将结构化的图谱知识与大语言模型相结合，为 RAG 系统提供了新的技术路径。它特别适合需要精确结构化知识支持的场景，能够有效减少单纯基于 Embedding 相似性检索带来的语义歧义问题。

尽管 Graph RAG 在实体识别精度、图谱构建成本等方面仍面临挑战，但通过两阶段排序优化、多源检索融合等技术手段，可以显著提升其效果。随着知识图谱技术和大语言模型的不断发展，Graph RAG 在专业领域问答、知识推理等场景中将发挥越来越重要的作用。

未来的研究方向包括自动化知识图谱构建、多模态 Graph RAG、动态图谱学习等，这些技术的发展将进一步拓展 Graph RAG 的应用边界和实用价值。

第二十六章　大模型 (LLMs) 参数高效微调 (PEFT) 技术综述

26.1　微调方法概述

26.1.1　微调方法分类

- **全参数微调 (Full Fine-Tune)：**也叫全参微调，BERT 微调模型一直使用这种方法，全部参数权重参与更新以适配领域数据，效果好。
- **提示微调 (Prompt-Tune)：**包括 P-Tuning、LoRA、Prompt-Tuning、AdaLoRA 等 Delta Tuning 方法，部分模型参数参与微调，训练快，显存占用少，效果可能跟全参数微调相比会稍有效果损失，但一般效果能打平。

26.1.2　技术对比研究

链家在 BELLE 的技术报告《A Comparative Study between Full-Parameter and LoRA-based Fine-Tuning on Chinese Instruction Data for Instruction Following Large Language Model》中实验显示：全参数微调效果稍好于 LoRA。

表 26.1: BELLE 技术报告主要实验结果对比

Model	Average Score	Additional Param.	Training Time(
LLaMA-13B+LoRA(2M)	0.648	28M	10
LLaMA-7B+LoRA(4M)	0.624	17.9M	14
LLaMA-7B+LoRA(2M)	0.609	17.9M	-
LLaMA-7B+LoRA(0.6M)	0.589	17.9M	75
LLaMA-7B+FT(2M)	0.710	-	31
LLaMA-7B+FT(0.6M)	0.686	-	17
LLaMA-7B+FT(2M)+LoRA(math_0.25M)	0.729	17.9M	12
LLaMA-7B+FT(2M)+FT(math_0.25M)	0.738	-	4

PEFT 的论文《ADAPTIVE BUDGET ALLOCATION FOR PARAMETER-EFFICIENT FINE-TUNING》显示的结果：AdaLoRA 效果稍好于全参数微调。

表 26.2: DeBERTaV3-base 在 GLUE 开发集上的结果对比

Method	# Params	MNLI m/mm	SST-2 Acc	CoLA Mcc	QQP Acc/F1	QNLI Acc
Full FT	184M	89.90/90.12	95.63	69.19	92.40/89.80	94.03
BitFit	0.1M	89.37/89.91	94.84	66.96	88.41/84.95	92.24
HAdapter	1.22M	90.13/90.17	95.53	68.64	91.91/89.27	94.11
PAdapter	1.18M	90.33/90.39	95.61	68.77	92.04/89.40	94.29
LoRA r=8	1.33M	90.65/90.69	94.95	69.82	91.99/89.38	93.87
AdaLoRA	1.27M	90.76/90.79	96.10	71.45	92.23/89.74	94.55

26.2　为什么需要 PEFT？

在面对特定的下游任务时，如果进行全参数微调（即对预训练模型中的所有参数都进行微调），太过低效；而如果采用固定预训练模型的某些层，只微调接近下游任务的那几层参数，又难以达到较好的效果。

26.3　PEFT 技术介绍

26.3.1　PEFT 定义

PEFT（Parameter-Efficient Fine-Tuning）技术旨在通过最小化微调参数的数量和计算复杂度，来提高预训练模型在新任务上的性能，从而缓解大型预训练模型的训练成本。这样一来，即使计算资源受限，也可以利用预训练模型的知识来迅速适应新任务，实现高效的迁移学习。

26.3.2　PEFT 优点

- 可以在提高模型效果的同时，大大缩短模型训练时间和计算成本
- 让更多人能够参与到深度学习研究中来
- 可以缓解全量微调带来的灾难性遗忘问题

表 26.3: 不同微调方法的资源消耗对比

微调方法	批处理大小	模式	GPU 显存	速度
LoRA(r=8)	16	FP16	28GB	8ex/s
LoRA(r=8)	8	FP16	24GB	8ex/s
LoRA(r=8)	4	FP16	20GB	8ex/s
LoRA(r=8)	4	INT8	10GB	8ex/s
LoRA(r=8)	4	INT4	8GB	8ex/s
P-Tuning(p=16)	4	FP16	20GB	8ex/s
P-Tuning(p=16)	4	INT8	16GB	8ex/s
P-Tuning(p=16)	4	INT4	12GB	8ex/s
Freeze(l=3)	4	FP16	24GB	8ex/s
Freeze(l=3)	4	INT8	12GB	8ex/s

26.4　微调方法性能对比

26.4.1　资源消耗对比

26.4.2　PEFT 与全量微调的区别

所谓的 Fine-Tune 只能改变风格，不能改变知识，是因为我们的 Fine-Tune，像是 LoRA 本来就是低秩的，没办法对模型产生决定性的改变。要是全量微调，还是可以改变知识的。

26.5　多种高效微调方法对比

26.5.1　方法选择建议

像 P-Tuning v2、LoRA 等都是综合评估很不错的高效微调技术。如果显存资源有限可以考虑 QLoRA；如果只是解决一些简单任务场景，可以考虑 P-Tuning、Prompt Tuning 也行。

26.5.2　综合对比表

下表从参数高效方法类型、是否存储高效和内存高效、以及在减少反向传播成本和推理开销的计算高效五个维度比较了参数高效微调方法。

方法类型说明：A-加法型，S-选择性，R-基于重参数化。

26.5.3　参数规模评估

下表展示了各种参数高效方法的参与训练的参数量、最终模型与原始模型的改变参数（delta 值）以及论文中参与评估的模型的范围（<1B、<20B、>20B）。

从表中可以看到，Prompt Tuning、Prefix Tuning、LoRA 等少部分微调技术针对不同参数规模的模型进行过评估，同时，这几种方式也是目前应用比较多的高效微调方法。

26.6　当前高效微调技术存在的问题

当前的高效微调技术很难在类似方法之间进行直接比较并评估它们的真实性能，主要原因如下：

26.6.1　参数计算口径不一致

参数计算可以分为三类：可训练参数的数量、微调模型与原始模型相比改变的参数的数量、微调模型和原始模型之间差异的等级。例如，DiffPruning 更新 0.5% 的参数，但是实际参与训练的参数量是 200%。这为比较带来了困难。尽管可训练的参数量是最可靠的存储高效指标，但是也不完美。Ladder-side Tuning 使用一个单独的小网络，参数量高于 LoRA 或 BitFit，但是因为反向传播不经过主网络，其消耗的内存反而更小。

26.6.2　缺乏模型大小的考虑

已有工作表明，大模型在微调中需要更新的参数量更小（无论是以百分比相对而论还是以绝对数量而论），因此（基）模型大小在比较不同 PEFT 方法时也要考虑到。

26.6.3　缺乏测量基准和评价标准

不同方法所使用的模型/数据集组合都不一样，评价指标也不一样，难以得到有意义的结论。

26.6.4　代码实现可读性差

很多开源代码都是简单拷贝 Transformer 代码库，然后进行小修小补。这些拷贝也不使用 git fork，难以找出改了哪里。即便是能找到，可复用性也比较差（通常指定某个 Transformer 版本，没有说明如何脱离已有代码库复用这些方法）。

26.7　高效微调技术最佳实践

针对以上存在的问题，研究高效微调技术时，建议按照最佳实践进行实施：

- **明确指出参数数量类型**：在报告中清晰说明使用的是可训练参数数量还是改变的参数数量
- **使用不同大小的模型进行评估**：在不同规模的基座模型上进行测试，确保方法的普适性
- **和类似方法进行比较**：在相同的数据集和评估指标下与已有方法进行公平比较
- **标准化 PEFT 测量基准**：建立统一的评测基准和标准数据集
- **重视代码清晰度**：以最小化进行实现，提高代码的可读性和可复用性

26.8　PEFT 存在的问题

相比全参数微调，大部分的高效微调技术目前存在的两个问题：

1. **推理速度会变慢**：由于需要额外的计算或参数加载，推理速度可能受到影响
2. **模型精度会变差**：在某些任务上可能无法达到全参数微调的性能水平

26.9　参数高效微调方法总结

本文针对之前介绍的几种参数高效微调方法进行了简单的概述，主要有如下几类：

26.9.1　方法分类

- **增加额外参数**：如 Prefix Tuning、Prompt Tuning、Adapter Tuning 及其变体
- **选取一部分参数更新**：如 BitFit
- **引入重参数化**：如 LoRA、AdaLoRA、QLoRA
- **混合高效微调**：如 MAM Adapter、UniPELT

26.9.2　技术特点比较

比较了不同的高效微调方法之间的差异；同时，还指出当前大多数高效微调方法存在的一些问题并给出了最佳实践。

26.10　未来发展方向

- **更高效的参数利用**：进一步提高参数效率，减少可训练参数数量
- **多任务适应性**：开发能够更好地适应多任务学习的 PEFT 方法
- **理论分析**：加强对 PEFT 方法工作原理的理论理解
- **硬件优化**：针对特定硬件优化 PEFT 方法的实现
- **自动化调优**：开发自动选择最优 PEFT 方法和超参数的技术

表 26.4: PEFT 方法综合对比

Method	**Type**	**Storage**	**Memory**	**Backprop**	**Inference o**
Adapters(Houlsby et al., 2019)	A	yes	yes	no	Extra FFN
AdaMix(Wang et al., 2022)	A	yes	yes	no	Extra FFN
SparseAdapter(He et al., 2022b)	AS	yes	yes	no	Extra FFN
Cross-Attn tuning(Gheini et al., 2021)	S	yes	yes	no	No overhead
BitFit(Ben-Zaken et al., 2021)	S	yes	yes	no	No overhead
DiffPruning(Guo et al., 2020)	S	yes	no	no	No overhead
Fish-Mask(Sung et al., 2021)	S	yes	maybe5	no	No overhead
LT-SFT(Ansell et al., 2022)	S	yes	maybe5	no	No overhead
Prompt Tuning(Lester et al., 2021)	A	yes	yes	no	Extra input
Prefix-Tuning(Li and Liang, 2021)	A	yes	yes	no	Extra input
Spot(Vu et al., 2021)	A	yes	yes	no	Extra input
IPT(Qin et al., 2021)	A	yes	yes	no	Extra FFN a
MAM Adapter(He et al., 2022a)	A	yes	yes	no	Extra FFN a
Parallel Adapter(He et al., 2022a)	A	yes	yes	no	Extra FFN
Intrinsinc SAID(Aghajanyan et al., 2020)	R	no	no	no	No overhead
LoRa(Hu et al., 2021)	R	yes	yes	no	No overhead
UniPELT(Mao et al., 2021)	AR	yes	yes	no	Extra FFN a
Compacter(Karimi Mahabadi et al., 2021)	AR	yes	yes	no	Extra FFN
PHM Adapter(Karimi Mahabadi et al., 2021)	AR	yes	yes	no	Extra FFN
KronA(Edalati et al., 2022)	R	yes	yes	no	No overhead
KronAres(Edalati et al., 2022)	AR	yes	yes	no	Extra linear
(IA)3(Liu et al., 2022)	A	yes	yes	no	Extra gating
Attention Fusion(Cao et al., 2022)	A	yes	yes	yes	Extra decode
LeTS(Fu et al., 2021)	A	yes	yes	yes	Extra FFN
Ladder Side-Tuning(Sung et al., 2022)	A	yes	yes	yes	Extra decode
FAR(Vucetic et al., 2022)	S	yes	maybe6	no	No overhead
S4-model(Chen et al., 2023)	ARS	yes	yes	no	Extra FFN a

表 26.5: PEFT 方法参数规模评估

Method	% Trainable parameters	% Changed parameters
Adapters(Houlsby et al., 2019)	0.1-6	0.1-6
AdaMix(Wang et al., 2022)	0.1-0.2	0.1-0.2
SparseAdapter(He et al., 2022b)	2.0	2.0
BitFit(Ben-Zaken et al., 2021)	0.05-0.1	0.05-0.1
DiffPruning(Guo et al., 2020)	200	0.5
Fish-Mask(Sung et al., 2021)	0.01-0.5	0.01-0.5
LT-SFT(Ansell et al., 2022)	0.01-0.5	0.01-0.5
Prompt Tuning(Lester et al., 2021)	0.1	0.1
Prefix-Tuning(Li and Liang, 2021)	0.1-4.0	0.1-4.0
IPT(Qin et al., 2021)	56.0	56.0
MAM Adapter(He et al., 2022a)	0.5	0.5
Parallel Adapter(He et al., 2022a)	0.5	0.5
Intrinsinc SAID(Aghajanyan et al., 2020)	0.001-0.1	∼0.1 or 100
LoRa(Hu et al., 2021)	0.01-0.5	∼0.5 or ∼30
UniPELT(Mao et al., 2021)	1.0	1.0
Compacter(Karimi Mahabadi et al., 2021)	0.05-0.07	∼0.07 or ∼0.1
PHM Adapter(Karimi Mahabadi et al., 2021)	0.2	∼0.2 or ∼1.0
KronA(Edalati et al., 2022)	0.07	∼0.07 or ∼30.0
KronAres(Edalati et al., 2022)	0.07	∼0.07 or ∼1.0
(IA)3(Liu et al., 2022)	0.02	0.02
Ladder Side-Tuning(Sung et al., 2022)	7.5	7.5
FAR(Vucetic et al., 2022)	6.6-26.4	6.6-26.4
S4-model(Chen et al., 2023)	0.5	more than 0.5

第二十七章　适配器微调 (Adapter-tuning) 技术详解

27.1　引言：为什么需要适配器微调？

27.1.1　全量微调的挑战

随着预训练模型参数量的不断增加，在特定下游任务中进行全量微调（Full Fine-Tuning）变得既昂贵又耗时。大型语言模型通常包含数十亿甚至数千亿参数，对其进行全参数微调需要巨大的计算资源和时间成本。

27.1.2　适配器微调的优势

适配器微调（Adapter-tuning）作为一种参数高效微调（PEFT）方法，通过仅训练少量额外参数来实现模型适配，显著降低了计算成本和存储需求。

27.2　适配器微调基本原理

27.2.1　核心思路

适配器微调的核心思想是在预训练模型的 Transformer 层中插入小型神经网络模块（适配器），在微调过程中只更新这些适配器参数，而保持原始预训练模型参数冻结。

27.2.2　适配器结构设计

适配器通常采用 bottleneck 结构，包含三个主要组件：

1. **下投影层（Down-Projection）**：将高维特征映射到低维空间
2. **非线性激活层**：引入非线性变换能力
3. **上投影层（Up-Projection）**：将低维特征映射回原始高维空间

数学表达式为：

$$\text{Adapter}(x) = U(\sigma(D(x)))$$

其中 D 为下投影矩阵，σ 为非线性激活函数，U 为上投影矩阵。

27.2.3　残差连接设计

适配器通常包含 skip-connection 结构，确保在最差情况下能够退化为恒等映射：

$$y = x + \text{Adapter}(x)$$

这种设计保证了训练的稳定性，即使适配器初始化不佳，也不会破坏原始模型的表示能力。

27.3　适配器微调的技术特点

27.3.1　参数效率

- 仅需训练原模型参数量的 0.5%-8%
- 大幅减少存储需求，每个任务只需保存适配器参数
- 支持多任务学习，不同任务共享基础模型

27.3.2　推理开销

- 在推理时会增加额外的计算延迟
- 适配器的插入增加了模型深度
- 需要权衡参数效率与推理速度

27.3.3　代码实现示例

```
import torch
import torch.nn as nn

class Adapter(nn.Module):
    def __init__(self, dim, reduction_factor=4):
        super().__init__()
        self.down_proj = nn.Linear(dim, dim // reduction_factor)
        self.non_linear = nn.ReLU()
        self.up_proj = nn.Linear(dim // reduction_factor, dim)

    def forward(self, x):
        # 残差连接设计
        residual = x
        x = self.down_proj(x)
```

```
        x = self.non_linear(x)
        x = self.up_proj(x)
        return residual + x

class TransformerWithAdapter(nn.Module):
    def __init__(self, config, adapter_config):
        super().__init__()
        self.attention = nn.MultiheadAttention(
            config.hidden_size, config.num_attention_heads
        )
        self.feed_forward = nn.Sequential(
            nn.Linear(config.hidden_size, config.intermediate_size),
            nn.ReLU(),
            nn.Linear(config.intermediate_size, config.hidden_size)
        )
        # 在FFN后插入适配器
        self.adapter = Adapter(config.hidden_size, adapter_config.
           reduction_factor)

    def forward(self, x):
        # 自注意力层
        attn_output, _ = self.attention(x, x, x)
        x = x + attn_output

        # 前馈网络+适配器
        ff_output = self.feed_forward(x)
        x = x + ff_output
        x = self.adapter(x)  # 适配器微调

        return x
```

27.4　AdapterFusion：多任务知识融合

27.4.1　设计思路

AdapterFusion 是一种两阶段训练策略，旨在有效融合多个源任务的知识来提升下游目标任务的表现。

27.4.2　两阶段训练流程

1. **阶段一：知识获取**
   - 在不同源任务上独立训练多个适配器
   - 每个适配器学习特定任务的知识表示
   - 保持基础预训练模型参数冻结
2. **阶段二：知识融合**
   - 设计融合机制组合多个源任务适配器
   - 学习如何加权组合不同适配器的输出
   - 优化跨任务的知识迁移效果

27.4.3　融合机制

AdapterFusion 通过注意力机制动态组合不同适配器的输出：

$$\text{Fusion}(x) = \sum_{i=1}^{N} \alpha_i \cdot \text{Adapter}_i(x)$$

其中 α_i 是通过注意力计算得到的权重。

27.4.4　代码实现

```
class AdapterFusion(nn.Module):
    def __init__(self, dim, num_adapters):
        super().__init__()
        self.adapters = nn.ModuleList([
            Adapter(dim) for _ in range(num_adapters)
        ])
        self.attention = nn.MultiheadAttention(dim, num_heads=1)

    def forward(self, x):
        adapter_outputs = []
        for adapter in self.adapters:
            adapter_outputs.append(adapter(x))

        # 堆叠所有适配器输出
        adapter_stack = torch.stack(adapter_outputs, dim=0)

        # 使用注意力机制进行融合
        query = x.unsqueeze(0)  # 添加序列维度
        fused_output, weights = self.attention(
```

```
            query, adapter_stack, adapter_stack
        )

        return fused_output.squeeze(0), weights
```

27.5　AdapterDrop：动态效率优化

27.5.1　设计动机

为了解决适配器在推理时的计算开销问题，AdapterDrop 通过在训练和推理时动态移除部分适配器来提升效率。

27.5.2　核心策略

- **层级剪枝**：从较低的 Transformer 层开始移除适配器
- **动态决策**：根据任务重要性决定保留哪些适配器
- **性能保持**：在保证任务性能的前提下最大化效率提升

27.5.3　技术特点

1. **推理加速**：通过减少适配器计算量提升推理速度
2. **多任务优化**：针对不同任务动态调整适配器配置
3. **性能平衡**：在效率与效果之间寻求最优平衡

27.5.4　动态决策算法

```
class AdapterDrop:
    def __init__(self, total_layers, keep_ratio=0.7):
        self.total_layers = total_layers
        self.keep_ratio = keep_ratio

    def select_layers(self, task_importance):
        """根据任务重要性选择要保留适配器的层级"""
        num_keep = int(self.total_layers * self.keep_ratio)

        # 高层级通常包含更多任务特定信息，优先保留
        start_layer = self.total_layers - num_keep
        keep_layers = list(range(start_layer, self.total_layers))

```

```
        return keep_layers

    def apply_drop(self, model, keep_layers):
        """应用适配器剪枝"""
        for i, layer in enumerate(model.transformer_layers):
            if i not in keep_layers:
                # 移除该层的适配器
                layer.adapter = None

# 使用示例
adapter_drop = AdapterDrop(total_layers=12, keep_ratio=0.7)
important_layers = adapter_drop.select_layers(task_importance=0.8)
adapter_drop.apply_drop(model, important_layers)
```

27.6　MAM Adapter：统一框架设计

27.6.1　整合思路

MAM Adapter 旨在建立 Adapter、Prefix Tuning 和 LoRA 等高效微调方法之间的统一框架，通过组合不同技术的优势来提升整体性能。

27.6.2　架构设计

MAM Adapter 主要由两个组件构成：

并行适配器（Parallel Adapter）

- 用于前馈网络（FFN）的适配器设计
- 与原始 FFN 并行计算，避免序列化延迟
- 采用 bottleneck 结构保持参数效率

软提示（Soft Prompts）

- 集成 Prefix Tuning 的思想
- 在输入序列前添加可训练的提示向量
- 引导模型关注任务相关信息

27.6.3　数学表达

对于 Transformer 层的计算：

$$\text{FFN}_{\text{parallel}} = \text{FFN}(x) + \text{Adapter}(x)$$
$$\text{Attention}_{\text{enhanced}} = \text{Attention}(\text{Prompt} \oplus x)$$

27.6.4　优势分析

1. **性能提升**：组合多种技术优势，效果优于单一方法
2. **灵活性**：可根据任务需求调整不同组件的权重
3. **可扩展性**：易于集成新的高效微调技术

27.6.5　完整实现

```
class MAMAdapterLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.hidden_size = config.hidden_size

        # 自注意力层
        self.attention = nn.MultiheadAttention(
            config.hidden_size, config.num_attention_heads
        )

        # 软提示（Prefix Tuning）
        self.prompt_length = config.prompt_length
        self.prompt_embeddings = nn.Parameter(
            torch.randn(config.prompt_length, config.hidden_size)
        )

        # 前馈网络
        self.ffn = nn.Sequential(
            nn.Linear(config.hidden_size, config.intermediate_size),
            nn.ReLU(),
            nn.Linear(config.intermediate_size, config.hidden_size)
        )

        # 并行适配器
        self.adapter = Adapter(config.hidden_size)

```

```
    def forward(self, x):
        # 添加软提示
        batch_size = x.size(1)
        prompts = self.prompt_embeddings.unsqueeze(1).repeat(1,
            batch_size, 1)
        x_with_prompt = torch.cat([prompts, x], dim=0)

        # 自注意力计算
        attn_output, _ = self.attention(
            x_with_prompt, x_with_prompt, x_with_prompt
        )
        # 去除提示部分，只保留原始序列
        attn_output = attn_output[self.prompt_length:]
        x = x + attn_output

        # 前馈网络 + 并行适配器
        ffn_output = self.ffn(x)
        adapter_output = self.adapter(x)
        # 并行计算：FFN输出与适配器输出相加
        x = x + ffn_output + adapter_output

        return x
```

27.7　适配器微调的性能评估

27.7.1　效率对比

表 27.1: 不同微调方法的效率对比

方法	可训练参数比例	训练速度	推理速度	效果保持率
全量微调	100%	1.0x	1.0x	100%
Adapter-tuning	0.5-8%	3-5x	0.8-0.9x	95-98%
AdapterFusion	1-10%	2-4x	0.7-0.8x	96-99%
AdapterDrop	0.3-6%	4-6x	0.9-1.0x	94-97%
MAM Adapter	1-12%	2-3x	0.7-0.8x	97-99%

27.7.2　适用场景分析

- **Adapter-tuning**：适合单任务微调，平衡效果与效率
- **AdapterFusion**：适合多任务学习，需要知识融合的场景
- **AdapterDrop**：适合推理资源受限的部署环境
- **MAM Adapter**：适合追求最佳效果的复杂任务

27.8　实践建议与最佳实践

27.8.1　参数配置建议

1. **缩减因子选择**：通常设置为 4-16，根据模型大小和任务复杂度调整
2. **适配器位置**：建议在 FFN 之后添加，避免影响注意力机制
3. **初始化策略**：适配器最后层初始化为近零值，确保初始状态接近恒等映射

27.8.2　训练技巧

```
def setup_adapter_training(model, learning_rate=1e-3):
    """配置适配器微调训练参数"""

    # 冻结基础模型参数
    for name, param in model.named_parameters():
        if 'adapter' not in name and 'prompt' not in name:
            param.requires_grad = False

    # 只为适配器参数设置优化器
    adapter_params = []
    for name, param in model.named_parameters():
        if 'adapter' in name or 'prompt' in name:
            adapter_params.append(param)

    optimizer = torch.optim.AdamW(adapter_params, lr=learning_rate)
    return optimizer

# 使用示例
model = TransformerWithAdapter(config, adapter_config)
optimizer = setup_adapter_training(model, learning_rate=1e-4)
```

27.8.3　多任务适配策略

- **渐进式训练**：先训练通用任务适配器，再微调特定任务
- **知识蒸馏**：使用大模型适配器指导小模型训练
- **动态加载**：根据任务需求动态加载不同适配器

27.9　总结与展望

27.9.1　技术优势总结

- **参数高效**：大幅减少可训练参数量
- **存储经济**：多个任务共享基础模型
- **训练快速**：收敛速度快，资源需求低
- **灵活适配**：支持增量学习和多任务学习

27.9.2　未来发展方向

1. **自动化架构搜索**：自动学习最优适配器结构和位置
2. **动态架构调整**：根据输入动态调整适配器配置
3. **跨模态适配**：扩展适配器到多模态学习场景
4. **理论分析**：深入理解适配器工作的理论机制

27.9.3　应用前景

适配器微调技术在大模型时代具有广阔的应用前景，特别是在：

- 资源受限的边缘计算设备
- 需要快速适配多任务的企业应用
- 注重数据隐私的联邦学习场景
- 需要持续学习的在线学习系统

第二十八章　提示学习 (Prompting) 技术详解

28.1　引言：为什么需要提示学习？

28.1.1　全量微调的挑战

在面对特定的下游任务时，如果进行全量微调（Full Fine-Tuning），即对预训练模型中的所有参数都进行微调，这种方法太过低效。而如果采用固定预训练模型的某些层，只微调接近下游任务的那几层参数，又难以达到较好的效果。

28.1.2　提示学习的优势

提示学习（Prompting）旨在通过最小化微调参数的数量和计算复杂度，来提高预训练模型在新任务上的性能，从而缓解大型预训练模型的训练成本。这样一来，即使计算资源受限，也可以利用预训练模型的知识来迅速适应新任务，实现高效的迁移学习。

28.2　提示学习基本概念

28.2.1　什么是提示学习？

提示学习通过提供上下文和任务相关信息（即 Prompt），帮助模型更好地理解要求并生成正确的输出。Prompt 作为模型输入的引导信息，能够将下游任务重新表述为预训练任务的形式。

28.2.2　提示学习应用实例

- **问答任务**：Prompt 可能包含问题或话题的描述，以帮助模型生成正确的答案
- **情感分析任务**：在句子前面加入前缀” 该句子的情感是”，将情感分类任务转换为” 填空” 任务

在训练过程中，模型可以学习到 Prompt 与任务输出之间的关联。例如，可以学习到” 该句子的情感是积极的” 和” 该句子的情感是消极的” 之间的差异。

28.3　提示学习的优点

- **参数高效**：仅需微调少量参数，大幅降低计算资源需求
- **训练快速**：收敛速度快，适合资源受限的场景
- **知识保留**：保持预训练模型参数不变，避免灾难性遗忘
- **多任务适配**：同一模型可适配多个下游任务
- **迁移性强**：易于实现跨领域知识迁移

28.4　提示学习方法综述

28.4.1　方法分类

主要的提示学习方法包括：

- 前缀微调（Prefix-tuning）
- 指示微调（Prompt-tuning）
- P-tuning
- P-tuning v2

28.5　前缀微调（Prefix-tuning）

28.5.1　为什么需要前缀微调？

- **人工设计离散 Prompts 的缺点**：Prompts 的变化对模型性能特别敏感，加一个词、少一个词或者变动位置都会造成较大变化
- **自动化搜索离散 Prompts 的缺点**：成本较高，且离散化的 token 搜索结果可能不是最优的
- **传统微调的问题**：对每个任务都要保存微调后的模型权重，耗时长且占用存储空间大

28.5.2　前缀微调思路

1. **Prefix 构建**：在输入 token 之前构造一段任务相关的 virtual tokens 作为 Prefix
2. **参数冻结**：训练时只更新 Prefix 部分的参数，而 Transformer 中的其他参数固定
3. **稳定化设计**：在 Prefix 层前面添加 MLP 结构，将 Prefix 分解为更小维度的 Input 与 MLP 的组合输出，防止直接更新 Prefix 参数导致训练不稳定

28.5.3　前缀微调优点

- **vs 人工设计 Prompts**：可以学习” 隐式” 的 Prompts，而非依赖人工设计
- **批量处理能力**：可以在一个批次中处理来自多个用户/任务的样本

- **vs 全量微调**：只更新 Prefix 部分参数，大幅减少训练参数量

28.5.4　前缀微调缺点

- **序列长度占用**：占用序列长度，带来额外计算开销
- **架构改动较大**：在每层都添加 prompt 参数，模型结构改动较大

28.6　指示微调（Prompt-tuning）

28.6.1　为什么需要指示微调？

- **全量微调问题**：对每个任务训练一个模型，开销和部署成本高
- **离散 prompts 问题**：人工设计 prompts 成本高，效果不稳定
- **前缀微调局限性**：占用序列长度，计算开销大，架构改动大

28.6.2　指示微调思路

1. **连续空间扩展**：将 prompt 扩展到连续空间，仅在输入层添加 prompt 连续向量
2. **参数优化**：通过反向传播更新参数学习 prompts，而非人工设计
3. **模型冻结**：冻结模型原始权重，只训练 prompts 参数
4. **关联建模**：使用 LSTM 建模 prompt 向量间关联性

28.6.3　指示微调优点

- **架构简洁**：只在输入层加入 prompt tokens，无需添加 MLP 调整
- **规模效应**：随着预训练模型参数量的增加，效果逼近全参数微调
- **集成效率**：支持 prompt ensembling，在同一批次训练不同 prompt

28.6.4　指示微调缺点

- **训练难度**：训练不稳定，省显存但不一定省时间
- **独立性假设**：多个 prompt token 之间相互独立，可能影响效果
- **小模型局限**：在正常大小的预训练模型上表现不佳
- **任务限制**：难以处理复杂的序列标注任务

表 28.1: 指示微调与前缀微调对比

特性	前缀微调（Prefix-tuning）	指示微调（Prompt-tuning）
适用任务	仅针对 NLG 任务有效，服务于 GPT 架构	考虑所有类型的语言模型
添加方式	限定在输入前面添加	可以在任意位置添加
参数添加	每一层都添加，保证效果	可以只在输入层添加
架构复杂度	相对复杂，每层都修改	相对简单，主要在输入层

表 28.2: 指示微调与全量微调对比

特性	全量微调（Fine-tuning）	指示微调（Prompt-tuning）
参数更新	需要改变预训练模型参数	不改变预训练模型参数
遗忘问题	可能带来灾难性遗忘	保持预训练知识完整性
训练成本	计算资源需求高	参数高效，资源需求低
多任务支持	需要为每个任务存储完整模型	共享基础模型，存储效率高

28.6.5　指示微调 vs 前缀微调

28.6.6　指示微调 vs 全量微调

28.7　P-tuning 方法

28.7.1　为什么需要 P-tuning？

- **Prompt 敏感性**：大模型的 Prompt 构造方式严重影响下游任务效果
- **GPT 系列局限**：自回归建模在自然语言理解任务上效果不如 BERT 双向模型
- **人工设计问题**：人工设计的模板对变化特别敏感
- **自动化成本**：离散化 token 搜索成本高，结果可能非最优

28.7.2　P-tuning 思路

1. **可学习 Embedding 层**：将 Prompt 转换为可学习的 Embedding 层
2. **Prompt 编码器**：使用 Prompt 编码器（双向 LSTM+ 两层 MLP）对 Prompt Embedding 进行处理
3. **依赖建模**：建模伪 token 的相互依赖关系，提供更好的初始化

28.7.3 P-tuning 优点

- **依赖建模**：通过 Prompt 编码器建模 token 间依赖关系
- **更好初始化**：提供更合理的参数初始化策略
- **连续优化**：在连续空间优化 prompt 表示

28.7.4 P-tuning 缺点

- **复杂性增加**：架构相对复杂，不太像传统 prompt
- **连续性不一致**：伪 token 编码连续但与输入结合时可能不连续
- **插入问题**：中间可能会插入输入，影响连贯性

28.7.5 P-tuning vs 传统微调

```python
# 传统全量微调
class TraditionalFineTuning:
    def __init__(self, model):
        self.model = model
        # 所有参数可训练
        for param in self.model.parameters():
            param.requires_grad = True

    def train(self, data):
        # 更新所有参数
        return self.model.update_all_parameters(data)

# P-tuning微调
class PTuning:
    def __init__(self, model, prompt_length=10):
        self.model = model
        self.prompt_length = prompt_length
        # 只添加prompt相关参数
        self.prompt_embeddings = nn.Parameter(
            torch.randn(prompt_length, model.hidden_size)
        )
        # 冻结原模型参数
        for param in self.model.parameters():
            param.requires_grad = False

    def train(self, data):
```

```
        # 只更新prompt参数
        return self.update_prompt_parameters(data)
```

28.8 P-tuning v2 方法

28.8.1 为什么需要 P-tuning v2?

主要解决如何让 Prompt Tuning 在不同参数规模的预训练模型、针对不同下游任务的结果上都达到匹敌全量微调的效果。

28.8.2 P-tuning v2 思路

1. **深度提示编码:**采用 Prefix-tuning 做法,在输入前面的每层加入可微调的 Prompts tokens
2. **简化架构：**移除重参数化编码器（MLP/LSTM），提高训练速度和鲁棒性
3. **动态提示长度：**针对不同任务采用不同的提示长度
4. **多任务预训练：**先在多任务 prompt 上预训练，再适配下游任务
5. **传统分类范式：**抛弃 verbalizer，回归到 CLS 和 token label 分类范式

28.8.3 P-tuning v2 优点

- **参数规模适中：**可学习参数从 0.01% 增加到 0.1%-3%，平衡效率与效果
- **深度集成：**深层 Prompt 对预测产生更直接影响
- **小模型适配：**解决 Prompt Tuning 在小模型上效果差的问题
- **任务扩展：**可应用于 NER 等序列标注任务
- **训练稳定：**通过多任务预训练缓解优化困难

28.8.4 P-tuning v2 缺点

- **Prompt 特性弱化：**抛弃 verbalizer 一定程度上弱化了 prompt 的原始特性
- **架构复杂性：**深度集成增加了模型复杂度
- **超参数敏感：**提示长度等超参数需要仔细调优

28.8.5 P-tuning v2 架构实现

```
class PTuningV2(nn.Module):
    def __init__(self, model, prompt_length=20, num_layers=12):
        super().__init__()
        self.model = model
        self.prompt_length = prompt_length
```

```
        self.num_layers = num_layers

        # 每层都添加prompt参数
        self.layer_prompts = nn.ParameterList([
            nn.Parameter(torch.randn(prompt_length, model.hidden_size))
            for _ in range(num_layers)
        ])

        # 冻结原始模型参数
        for param in self.model.parameters():
            param.requires_grad = False

    def forward(self, input_ids, attention_mask=None):
        batch_size = input_ids.shape[0]

        # 原始输入embedding
        input_embeds = self.model.embeddings(input_ids)

        # 为每层添加prompt
        hidden_states = input_embeds
        for i, layer in enumerate(self.model.encoder.layer):
            # 添加该层的prompt
            layer_prompt = self.layer_prompts[i].unsqueeze(0).expand(
                batch_size, -1, -1)
            prompt_length = layer_prompt.shape[1]

            # 拼接prompt和输入
            combined_embeds = torch.cat([layer_prompt, hidden_states],
                dim=1)

            # 调整attention mask
            if attention_mask is not None:
                prompt_mask = torch.ones(batch_size, prompt_length,
                                         device=attention_mask.device)
                combined_mask = torch.cat([prompt_mask, attention_mask],
                    dim=1)
            else:
                combined_mask = None

            # 通过transformer层
```

```
            layer_output = layer(combined_embeds, attention_mask=
                combined_mask)
            hidden_states = layer_output[0]

        return hidden_states
```

28.9　方法对比与分析

28.9.1　技术演进路径

表 28.3: 提示学习技术演进对比

特性	Prefix-tuning	Prompt-tuning	P-tuning	P-tuning v2
提出时间	2021	2021	2021	2022
核心思想	每层添加可学习 prefix	输入层添加连续 prompt	引入 prompt 编码器	深度提示编码
参数位置	每层 Transformer	仅输入层	输入层 + 编码器	每层 Transforme
训练参数	0.1%-1%	0.01%-0.1%	0.1%-1%	0.1%-3%
主要优势	深度集成效果	架构简单高效	依赖建模能力强	小模型效果好
主要局限	计算开销大	小模型效果差	架构复杂	超参数敏感

28.9.2　适用场景建议

- **资源极度受限**：优先考虑 Prompt-tuning，架构最简单
- **效果优先**：选择 P-tuning v2，效果最接近全量微调
- **序列生成任务**：Prefix-tuning 在 NLG 任务上表现优异
- **快速实验**：Prompt-tuning 实现快速，适合原型验证
- **生产部署**：P-tuning v2 平衡效果与效率，适合生产环境

28.10　实践建议与最佳实践

28.10.1　参数配置建议

```
# 提示学习超参数配置示例
prompt_config = {
    'prompt_length': 20,       # 提示长度: 10-50
```

```
    'learning_rate': 1e-4,     # 学习率：通常1e-5到1e-3
    'batch_size': 16,          # 批大小：根据显存调整
    'num_epochs': 10,          # 训练轮数：通常5-20
    'warmup_ratio': 0.1,       # 热身比例：0.1-0.2
}

# 不同模型规模的提示长度建议
model_size_prompt_config = {
    'small': {'prompt_length': 10, 'learning_rate': 1e-3},
    'base': {'prompt_length': 20, 'learning_rate': 5e-4},
    'large': {'prompt_length': 30, 'learning_rate': 1e-4},
    'xl': {'prompt_length': 50, 'learning_rate': 5e-5},
}
```

28.10.2　训练技巧

1. **渐进式训练**：先在小批量数据上训练，再扩展到全量数据
2. **学习率调度**：使用 warmup 和 cosine 衰减策略
3. **早停策略**：监控验证集性能，防止过拟合
4. **提示初始化**：使用任务相关词汇初始化 prompt embedding
5. **多任务预训练**：先在相关任务上预训练 prompt，再微调

28.11　挑战与未来方向

28.11.1　当前挑战

- **训练稳定性**：提示学习方法训练过程可能不稳定
- **超参数敏感**：对提示长度、学习率等超参数敏感
- **理论理解**：缺乏对提示学习工作原理的深入理论分析
- **多模态扩展**：如何扩展到视觉、语音等多模态任务
- **推理效率**：提示添加可能增加推理延迟

28.11.2　未来研究方向

- **自动化提示学习**：自动学习最优提示结构和内容
- **理论分析**：深入理解提示学习的表示学习机制
- **多模态提示**：开发跨模态的提示学习方法
- **动态提示**：根据输入动态调整提示内容
- **提示压缩**：研究更紧凑的提示表示方法

28.12　总结

提示学习作为一种参数高效的微调方法，通过引入可学习的提示参数来引导预训练模型适应下游任务，在保持预训练知识的同时大幅减少训练成本。从 Prefix-tuning 到 P-tuning v2，提示学习技术不断演进，在效果和效率之间寻求更好的平衡。

随着大模型技术的快速发展，提示学习将在资源受限的应用场景中发挥越来越重要的作用，为更广泛的人群提供使用大模型的能力。未来的研究将着重于提高方法的稳定性、扩展性和理论可解释性，推动提示学习技术在更多领域的应用。

第二十九章　LoRA 系列微调技术详解

29.1　LoRA 基础篇

29.1.1　什么是 LoRA？

LoRA（Low-Rank Adaptation）是一种通过低秩分解来模拟参数改变量的微调技术，通过极小的参数量实现大模型的间接训练。

29.1.2　LoRA 核心思路

1. **旁路结构设计**：在原模型旁边增加一个旁路，通过低秩分解（先降维再升维）模拟参数更新量
2. **参数冻结策略**：训练时固定原模型参数，只训练降维矩阵 A 和升维矩阵 B
3. **推理优化**：推理时将 BA 加到原参数上，不引入额外延迟
4. **初始化策略**：矩阵 A 采用高斯分布初始化，矩阵 B 初始化为全 0，保证训练开始时旁路为 0 矩阵
5. **任务切换**：通过插拔式设计实现任务切换，当前任务 $W_0+B_1A_1$，切换时替换为 $W_0+B_2A_2$

29.1.3　LoRA 技术特点

- **无推理延迟**：将 BA 加到 W 上可消除推理延迟
- **任务可插拔**：可通过可插拔形式切换到不同任务
- **设计优雅**：简单且效果好的设计

29.1.4　LoRA 简单描述

LoRA 的实现思想很简单，就是冻结预训练模型的矩阵参数，选择用 A 和 B 矩阵替代，在下游任务时只更新 A 和 B 矩阵。

29.2 QLoRA 技术篇

29.2.1 QLoRA 核心思路

- 使用新颖的高精度技术将预训练模型量化为 4bit
- 添加一小组可学习的低秩适配器权重
- 通过量化权重的反向传播梯度进行微调

29.2.2 QLoRA 技术特点

- **显存优化**：显著降低显存需求
- **训练速度**：训练速度慢于标准 LoRA
- **资源友好**：适合资源受限环境

29.3 AdaLoRA 技术篇

29.3.1 AdaLoRA 核心思路

AdaLoRA 是对 LoRA 的改进，根据重要性评分动态分配参数预算：

- 将关键增量矩阵分配给重要和任务特定的信息
- 降低较不重要矩阵的秩，防止过拟合并节省计算预算
- 实现自适应的参数分配策略

29.4 LoRA 权重管理

29.4.1 权重合并可行性

可以将训练好的低秩矩阵 $(B \times A)$ 与原模型权重合并（相加），计算出新的权重。

29.4.2 实际存储需求

在 rank=8、target_module=query_key_value 条件下，ChatGLM-6B LoRA 后权重约为 15MB。

29.5 LoRA 微调优势分析

29.5.1 主要优点

1. **参数存储优化**：一个中心模型服务多个下游任务，节省参数存储量

2. **推理效率**：推理阶段不引入额外计算量
3. **技术正交性**：与其它参数高效微调方法正交，可有效组合
4. **训练稳定性**：训练任务比较稳定，效果较好
5. **无延迟设计**：几乎不添加任何推理延迟，适配器权重可与基础模型合并

29.5.2 训练加速原理

LoRA 微调能加速训练的原因：

- **参数更新优化**：只更新部分参数（如只更新 SelfAttention 参数）
- **通信优化**：减少多卡训练时的数据传输量
- **低精度技术**：采用 FP16、FP8 或 INT8 等低精度加速技术

29.6 LoRA 持续训练策略

29.6.1 持续训练方法

在已有 LoRA 模型上继续训练的推荐策略：

- 将之前的 LoRA 与 base model 合并后继续训练
- 为保留原有知识能力，训练新 LoRA 时加入部分历史训练数据
- 避免从头开始训练以降低成本

29.7 LoRA 局限性分析

29.7.1 技术局限性

- **参数量限制**：参与训练的参数量有限（百万到千万级别）
- **效果差距**：效果通常比全量微调差很多
- **LLM 表现**：在大型语言模型上表现差距明显

表 29.1: LoRA 与全参数微调对比

对比维度	**LoRA 微调**	**全参数微调**
计算资源需求	资源需求低，消费级 GPU 可训练	需要大量计算资源
训练时间	训练时间相对较长	训练时间相对较短
数据需求	适合数据量较少场景	需要大量数据（10k 以上）
参数效率	只训练少量参数	训练全部参数
效果表现	数据量大时效果不如全量微调	数据充足时效果最优

表 29.2: LoRA 在不同模型和数据集上的性能表现

Model	**Training data**	**others**	**rewrite**	**classification**	**generation**	**sum**
LLaMA-7B+LoRA	0.6M	0.358	0.719	0.695	0.816	
LLaMA-7B+LoRA	2M	0.364	0.795	0.676	0.854	
LLaMA-7B+LoRA	4M	0.341	0.821	0.677	0.847	
LLaMA-13B+LoRA	2M	0.422	0.810	0.696	0.837	
LLaMA-7B+FT	0.6M	0.438	0.869	0.698	0.917	
LLaMA-7B+FT	2M	0.399	0.871	0.775	0.920	
LLaMA-7B+FT(2M)+LoRA	math0.25M	0.560	0.863	0.758	0.915	
LLaMA-7B+FT(2M)+FT	math0.25M	0.586	0.887	0.763	0.955	

29.7.2 与全参数微调对比

29.8 实验效果分析

29.8.1 多任务性能对比

29.9 LoRA 参数配置优化

29.9.1 Transformer 参数矩阵选择

关键发现：

- 将所有微调参数集中到 attention 的单一参数矩阵效果不佳
- 将可微调参数平均分配到 Wq 和 Wk 效果最好
- 即使秩取 4 也能在 ΔW 中获得足够信息
- 应将可微调参数分配到多种类型权重矩阵中

表 29.3: 不同权重矩阵的 LoRA 微调效果对比（可训练参数 =18M）

Weight Type	**W**	**Wk**	**Wv**	**Wo**	**Wq,Wk**	**Wq,Wv**	**Wq,Wk,Wv,Wo**
Rank r	8	8	8	8	4	4	2
WikiSQL(±0.5%)	70.4	70.0	73.0	73.2	71.4	73.7	73.7
MultiNLI(±0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	91.7

29.9.2 参数量确定方法

LoRA 模型中可训练参数量取决于：

- 秩 r 的大小
- 原始权重矩阵的形状
- 实际使用中的 lora_target 选择

以 LLaMA 为例的典型配置：

```
--lora_target q_proj,k_proj,v_proj,o_proj,gate_proj,up_proj,down_proj
```

29.9.3 Rank 选择策略

- **推荐范围**：Rank 在 4-8 之间效果最好
- **实验基础**：作者对比了 1-64 的 Rank 值
- **任务适配**：面向单一监督任务时 4-8 足够，指令微调需根据分布广度选择 8 以上

29.9.4 Alpha 参数配置

- **参数本质**：alpha 是缩放参数，本质与 learning rate 相同
- **简化策略**：默认设置 alpha=rank，只调整 learning rate
- **超参优化**：此策略可简化超参数调优

29.10 过拟合防止策略

29.10.1 过拟合应对措施

1. **秩调整**：减小 r 值
2. **数据增强**：增加数据集大小
3. **正则化**：增加优化器的权重衰减率
4. **Dropout**：增加 LoRA 层的 dropout 值

29.11 优化器选择

29.11.1 优化器推荐

- **常用选择**：Adam 和 AdamW
- **新兴选择**：Sophia（使用梯度曲率而非方差进行归一化）
- **潜在优势**：可能提高训练效率和模型性能

29.12 内存使用优化

29.12.1 内存影响因素

- **模型规模**：模型大小直接影响内存占用
- **批处理大小**：批量大小对内存有显著影响
- **LoRA 参数**：LoRA 参数数量影响内存使用
- **数据特性**：数据集特性（如序列长度）影响内存
- **优化策略**：使用较短训练序列可节省内存

29.13 高级特性

29.13.1 权重合并能力

支持多套 LoRA 权重合并：

- 训练中保持 LoRA 权重独立
- 前向传播时添加各 LoRA 权重
- 训练后可合并权重简化操作

29.13.2 逐层 Rank 调整

- **理论可行性**：可为不同层选择不同 LoRA rank
- **实践挑战**：类似为不同层设不同学习率，增加调优复杂性
- **实际应用**：由于复杂性，实际中很少执行

29.14 初始化策略

29.14.1 矩阵初始化方法

- **矩阵 B**：初始化为全 0
- **矩阵 A**：采用高斯分布初始化

29.14.2 初始化原理分析

- **全 0 初始化问题**：类似深度网络全 0 初始化，易导致梯度消失
- **全高斯初始化问题**：训练开始可能产生过大偏移值 ΔW，引入噪声难以收敛
- **混合初始化优势**：一部分初始为 0，一部分正常初始化，维持网络原有输出
- **训练稳定性**：确保训练开始后能更好收敛

29.15 实践指南

29.15.1 可训练参数比例确定

LoRA 微调计算可训练参数比例的方法：

```
def calculate_trainable_ratio(model, lora_config):
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.
       requires_grad)

    # LoRA参数计算
    lora_params = 0
    for module in model.modules():
        if hasattr(module, 'lora_A') and hasattr(module, 'lora_B'):
            lora_params += module.lora_A.numel() + module.lora_B.numel()

    trainable_ratio = lora_params / total_params
    return trainable_ratio
```

29.15.2 结果保存策略

LoRA 微调结果的保存方法：

```
# 保存LoRA权重
torch.save({
    'lora_A': model.lora_A.state_dict(),
    'lora_B': model.lora_B.state_dict(),
    'config': lora_config
}, 'lora_weights.pth')

# 合并权重保存
def merge_weights(base_model, lora_weights):
    with torch.no_grad():
```

```
        for name, param in base_model.named_parameters():
            if name in lora_weights:
                lora_A = lora_weights[name + '.lora_A']
                lora_B = lora_weights[name + '.lora_B']
                param.data += lora_B @ lora_A
```

29.16 总结

LoRA 系列技术通过低秩适配提供了一种参数高效的微调方案，在保持预训练模型知识的同时大幅减少训练参数量。从基础 LoRA 到 QLoRA、AdaLoRA 的演进，体现了在效果、效率和资源需求之间的持续优化平衡。

关键实践建议包括：合理选择 Rank 值（4-8）、采用正确的初始化策略、根据任务需求配置合适的参数矩阵目标，以及实施有效的过拟合防止措施。这些技术为资源受限环境下的大模型微调提供了实用解决方案。

第三十章　PEFT 库中 LoRA 使用详解

30.1　前言

本文主要介绍使用 PEFT 库中的 LoRA 模块对大模型进行高效参数微调，涉及以下内容：

- PEFT 库中 LoRA 模块的使用方法
- PEFT 库中 LoRA 模块的代码实现原理
- 推理时权重合并与模型加载策略

30.1.1　环境依赖配置

```
# 以下配置可能会随时间变化，建议根据实际情况调整
accelerate
appdirs
loralib
bitsandbytes
black
black[jupyter]
datasets
fire
transformers>=4.28.0
git+https://github.com/huggingface/peft.git
sentencepiece
gradio
wandb
cpm-kernel
```

30.2　LoraConfig 配置详解

30.2.1　基本配置示例

```
# 设置超参数及配置
```

```
LORA_R = 8
LORA_ALPHA = 16
LORA_DROPOUT = 0.05
TARGET_MODULES = [
    "q_proj",
    "v_proj",
]

config = LoraConfig(
    r=LORA_R,
    lora_alpha=LORA_ALPHA,
    target_modules=TARGET_MODULES,
    lora_dropout=LORA_DROPOUT,
    bias="none",
    task_type="CAUSAL_LM",
)
```

30.2.2 参数说明

- **r**：LoRA 的秩，矩阵 A 和矩阵 B 相连的宽度，满足 $r \ll d$
- **lora_alpha**：归一化超参数，LoRA 参数 ΔWx 被以 α/r 归一化
- **target_modules**：LoRA 的目标应用位置
- **merge_weights**：eval 模式中是否将 LoRA 矩阵值加到原有 W_0 上
- **lora_dropout**：LoRA 层的 dropout 比率
- **fan_in_fan_out**：仅应用于 Conv1D 层时设为 True
- **bias**：偏置训练设置（none：均不训练；all：全训练；lora_only：仅训练 LoRA 部分偏置）
- **task_type**：任务类型设置
- **modules_to_save**：除 LoRA 外其他可训练并需要保存的层

注意:target_modules 中的目标名称在不同模型中可能不同,如 ChatGLM 中使用 query_key_value。

30.3 模型加入 PEFT 策略

30.3.1 模型加载策略

模型加载涉及两个重要的显存优化技巧:load_in_8bit 和 prepare_model_for_int8_training。

30.3.2 模型显存占用分析

- **静态显存**：由模型参数量级决定
- **动态显存**：前向传播过程中每个样本的每个神经元计算的激活值存储，用于反向传播的梯度计算

30.3.3 显存优化策略

8bit 量化优化

from_pretrained 中的 load_in_8bit 参数由 bitsandbytes 库提供，将加载的模型转换为混合 8bit 量化模型。

```
# 8bit量化示例
model = AutoModel.from_pretrained(
    "THUDM/chatglm3-6b",
    load_in_8bit=True,
    device_map='auto'
)
```

量化原理：将 FP32（4 字节）压缩到 INT8（1 字节），实现 1/4 的显存占用。采用 absolute-maximum 或 zero-point 量化方案，其中 absmax 方案的基本原理：

$$\text{FP16 vector} : [1.2, 0.5, -4.3, 1.2, -3.1, 0.8, 2.4, 5.4]$$
$$\text{缩放因子} = \frac{127}{\max(|\text{vector}|)}$$
$$\text{量化结果} = \text{round}\left(\text{vector} \times \text{缩放因子}\right)$$

LLM.int8() 实现对 outlier 进行优化，将 outlier 和非 outlier 矩阵分开计算再合并，降低精度损失。

prepare_model_for_int8_training 函数适配 LLM.int8() 以提高训练稳定性，主要包括：

- Layer norm 层保持 FP32 精度
- 输出层保持 FP32 精度保证解码时随机采样的差异性

梯度检查优化

设置 gradient_checkpointing=True 实现时间换空间的优化：

- 前向传播使用 torch.no_grad() 不存储中间激活值
- 反向传播时重新计算激活值用于梯度计算
- 前向传播计算两遍，增加训练时间但减少显存占用

30.3.4 PEFT 策略集成

```
# 加入PEFT策略
model = get_peft_model(model, config)
model = model.to(device)
model.config.use_cache = False
```

注意：use_cache 设置为 False 是因为与 gradient checkpoint 存在冲突。use_cache 优化解码速度，在解码时存储每一步的 hidden-state 用于下一步输入，而 gradient checkpoint 不存储中间激活值。

30.4 PEFT 库中 LoRA 模块代码实现

30.4.1 整体架构设计

PEFT 库中，peft_model.py 中的 PeftModel 类是总控类，继承 transformers 中的 Mixin 类，负责模型读取保存等功能。

```
class LoraModel(torch.nn.Module):
    def __init__(self, config, model):
        super().__init__()
        self.peft_config = config
        self.model = model
        self._find_and_replace()
        mark_only_lora_as_trainable(self.model, self.peft_config.bias)
        self.forward = self.model.forward
```

构造方法主要完成两个步骤：

1. _find_and_replace()：找到需要加入 LoRA 策略的层并替换
2. mark_only_lora_as_trainable()：固定非 LoRA 参数，仅训练 LoRA 部分

30.4.2 __find_and_replace() 实现

```
def _find_and_replace(self):
    # 1. 找到目标层
    target_module_found = re.fullmatch(self.peft_config.target_modules,
        key)

    # 2. 创建新的LoRA层
    new_module = Linear(target.in_features, target.out_features, bias=
        bias, **kwargs)
```

```
7
8       # 3. 替换原层
9       self._replace_module(parent, target_name, new_module, target)
```

替换方法实现：

```
def _replace_module(self, parent_module, child_name, new_module,
    old_module):
    setattr(parent_module, child_name, new_module)
    new_module.weight = old_module.weight
    if old_module.bias is not None:
        new_module.bias = old_module.bias
    if getattr(old_module, "state", None) is not None:
        new_module.state = old_module.state
    new_module.to(old_module.weight.device)

    # 分配到正确设备
    for name, module in new_module.named_modules():
        if "lora" in name:
            module.to(old_module.weight.device)
```

30.4.3 LoRA 层实现细节

基类 LoraLayer

```
class LoraLayer:
    def __init__(
        self,
        r: int,
        lora_alpha: int,
        lora_dropout: float,
        merge_weights: bool,
    ):
        self.r = r
        self.lora_alpha = lora_alpha
        # 可选的dropout
        if lora_dropout > 0.0:
            self.lora_dropout = nn.Dropout(p=lora_dropout)
        else:
            self.lora_dropout = lambda x: x
        # 标记权重未合并
```

```
        self.merged = False
        self.merge_weights = merge_weights
        self.disable_adapters = False
```

Linear 层实现

Linear 类同时继承 nn.Linear 和 LoraLayer：

```
class Linear(nn.Linear, LoraLayer):
    # LoRA在dense层的实现
    def __init__(
        self,
        in_features: int,
        out_features: int,
        r: int = 0,
        lora_alpha: int = 1,
        lora_dropout: float = 0.0,
        fan_in_fan_out: bool = False,  # 替换层存储权重如(fan_in, fan_out)时设为True
        **kwargs,
    ):
        nn.Linear.__init__(self, in_features, out_features, **kwargs)
        LoraLayer.__init__(self, r=r, lora_alpha=lora_alpha,
                           lora_dropout=lora_dropout, merge_weights=
                               merge_weights)
        self.fan_in_fan_out = fan_in_fan_out

        if self.r > 0:
            self.weight.data -= (
                transpose(self.lora_B.weight @ self.lora_A.weight,
                          self.fan_in_fan_out) * self.scaling
            )
            self.merged = False
```

前向传播实现：

```
def forward(self, x: torch.Tensor):
    if self.disable_adapters:
        if self.r > 0 and self.merged:
            self.weight.data -= (
                transpose(self.lora_B.weight @ self.lora_A.weight,
```

```
                            self.fan_in_fan_out) * self.scaling
            )
            self.merged = False
        return F.linear(x, transpose(self.weight, self.fan_in_fan_out),
                        bias=self.bias)
    elif self.r > 0 and not self.merged:
        result = F.linear(x, transpose(self.weight, self.fan_in_fan_out),
                          bias=self.bias)
        if self.r > 0:
            result += self.lora_B(self.lora_A(self.lora_dropout(x))) *
               self.scaling
        return result
    else:
        return F.linear(x, transpose(self.weight, self.fan_in_fan_out),
                        bias=self.bias)
```

30.5 LoRA 微调存储策略

30.5.1 存储实现

PeftModel 重写了 save_pretrained 函数，只存储 LoRA 层权重：

```
# 重写Trainer的save_model, 在checkpoint时只存LoRA权重
class ModifiedTrainer(Trainer):
    def save_model(self, output_dir=None, _internal_call=False):
        from transformers.trainer import TRAINING_ARGS_NAME
        os.makedirs(output_dir, exist_ok=True)
        torch.save(self.args, os.path.join(output_dir, TRAINING_ARGS_NAME
           ))

        saved_params = {
            k: v.to("cpu") for k, v in self.model.named_parameters()
            if v.requires_grad
        }
        torch.save(saved_params, os.path.join(output_dir, "adapter_model.
           bin"))

# 使用示例
trainer = ModifiedTrainer(
    model=model,
```

```
    train_dataset=train_data,
    args=transformers.TrainingArguments(
        per_device_train_batch_size=8,
        gradient_accumulation_steps=16,
        num_train_epochs=10,
        learning_rate=3e-4,
        fp16=True,
        logging_steps=10,
        save_steps=200,
        output_dir=output_dir
    ),
    data_collator=DataCollatorForSeq2Seq(
        tokenizer, pad_to_multiple_of=8, return_tensors="pt", padding=
           True
    ),
)
trainer.train()
model.save_pretrained(train_args.output_dir)
```

30.6 LoRA 推理加载策略

30.6.1 方案一：直接加载 LoRA 层

```
from peft import PeftModel
from transformers import AutoModel, AutoTokenizer

model = AutoModel.from_pretrained(
    "THUDM/chatglm3-6b",
    trust_remote_code=True,
    load_in_8bit=True,
    device_map='auto'
)
tokenizer = AutoTokenizer.from_pretrained("THUDM/chatglm3-6b",
                                          trust_remote_code=True)
model = PeftModel.from_pretrained(model, "./lora_ckpt")
model.half().to(device)
model.eval()
```

缺点：增加推理延迟，适合线下测评。

30.6.2 方案二：权重合并后加载

```
tokenizer = AutoTokenizer.from_pretrained("THUDM/chatglm3-6b",
                                          trust_remote_code=True)

# 合并时禁用int8
model = AutoModel.from_pretrained(
    "THUDM/chatglm3-6b",
    load_in_8bit=False,
    torch_dtype=torch.float16,
    trust_remote_code=True,
    device_map={"": "cpu"},
)

# 检查权重是否合并成功
first_weight = model.base_model.layers[0].attention.query_key_value.
   weight
first_weight_old = first_weight.clone()

# 加载LoRA模型
lora_model = PeftModel.from_pretrained(
    model,
    "./lora_ckpt",
    device_map={"": "cpu"},
    torch_dtype=torch.float16,
)

# 合并权重
lora_model = lora_model.merge_and_unload()
lora_model.train(False)

# 验证合并
assert not torch.allclose(first_weight_old, first_weight),
       'Weight Should Changes after Lora Merge'

# 恢复原始key格式
deloreanized_sd = {
    k.replace("base_model.model.", ""): v
    for k, v in lora_model.state_dict().items()
    if "lora" not in k
```

```
}

# 保存合并后的权重
lora_model.save_pretrained(output_dir, state_dict=deloreanized_sd)
```

30.7 多 LoRA 适配器切换

30.7.1 环境要求

```
peft >= 0.3.0
```

30.7.2 使用方法

1. **加载第一个适配器：**指定 adapter_name 参数

```
model = PeftModel.from_pretrained(model, "tloen/alpaca-lora-7b",
                                  adapter_name="eng_alpaca")
```

2. **加载其他适配器：**使用 load_adapter 方法

```
model.load_adapter(peft_model_path, adapter_name)
```

3. **切换适配器：**使用 set_adapter 方法

```
model.set_adapter(adapter_name)
```

4. **禁用适配器：**使用 disable_adapter 上下文管理器

```
with model.disable_adapter():
    # 在此代码块内禁用适配器
```

5. **合并卸载适配器：**使用 merge_and_unload 方法

```
model = model.merge_and_unload()
```

30.7.3 实战案例

```
from peft import PeftModel
from transformers import LlamaTokenizer, LlamaForCausalLM,
    GenerationConfig

model_name = "decapoda-research/llama-7b-hf"
tokenizer = LlamaTokenizer.from_pretrained(model_name)
```

```
model = LlamaForCausalLM.from_pretrained(
    model_name,
    load_in_8bit=True,
    device_map="auto",
    use_auth_token=True
)

# 加载多个适配器
model = PeftModel.from_pretrained(model, "tloen/alpaca-lora-7b",
                                  adapter_name="eng_alpaca")
model.load_adapter("22h/cabrita-lora-v0-1", adapter_name="
   portuguese_alpaca")

# 切换适配器
model.set_adapter("eng_alpaca")
instruction = "Tell me about alpacas."
print(evaluate(instruction))

model.set_adapter("portuguese_alpaca")
instruction = "Invente uma desculpa criativa pra dizer que nao preciso ir
    a festa."
print(evaluate(instruction))

# 禁用适配器
with model.disable_adapter():
    instruction = "Invente uma desculpa criativa pra dizer que nao
       preciso ir festa."
    print(evaluate(instruction))
```

30.8 总结

本文详细介绍了 PEFT 库中 LoRA 模块的完整使用流程，从基础配置到高级功能，包括：

- **配置管理**：LoraConfig 的参数详解和最佳实践
- **显存优化**：8bit 量化和梯度检查等关键技术
- **代码实现**：PEFT 库中 LoRA 的核心实现原理
- **存储策略**：训练和推理时的权重管理方案
- **多适配器**：支持多个 LoRA 适配器的动态加载和切换

通过这些技术，可以在资源受限的环境下高效地对大语言模型进行微调，并在推理时灵

活地应用不同的适配器来适应多种任务场景。

第三十一章　大模型 (LLMs) 推理技术详解

31.1　引言：大模型推理的挑战与机遇

随着大语言模型 (LLMs) 参数规模的不断增长，推理过程中的显存占用、计算效率和输出质量等问题日益凸显。本章将深入探讨大模型推理的关键技术，包括显存优化、速度优化、参数配置以及实际应用中的各种挑战和解决方案。

31.2　推理显存占用分析

31.2.1　显存暴涨原因

大模型推理时显存占用显著增加且持续占用的主要原因包括：

1. **长序列处理**：随着序列长度增加，需要存储大量的 Query、Key、Value 矩阵
2. **缓存机制**：采用逐个预测 next token 的方式，需要缓存 K/V 值以加速解码过程
3. **中间激活值**：前向传播过程中产生的中间结果需要存储在显存中

31.2.2　显存组成分析

推理时的显存占用主要包含以下部分：

- 模型参数存储
- 激活值缓存
- 注意力机制中的 K/V 缓存
- 梯度计算所需空间（如果进行推理微调）

31.3　推理速度性能分析

31.3.1　GPU vs CPU 推理速度对比

在 7B 参数级别的模型上，推理速度对比如下：

表 31.1: 7B 模型在不同硬件上的推理速度对比

硬件平台	推理速度	相对性能
CPU（8 核 AMD）	约 10 token/秒	基准
单卡 A6000 GPU	约 100 token/秒	10 倍于 CPU

表 31.2: 不同精度下的推理性能比较

精度	推理速度	显存占用	质量保持
FP32	基准	最高	最佳
FP16	较快	减少 50%	基本无损
INT8	变慢（HuggingFace 实现）	减少 75%	轻微损失

31.3.2　精度对推理速度的影响

31.4　大模型的推理能力分析

31.4.1　上下文纠正能力

ChatGPT 展现出强大的 in-context correction 能力：

- 能够理解错误描述并朝正确方向修正
- 相比 in-context learning 更具挑战性
- 描述越详细准确，回答质量越高

31.4.2　知识推理与创造能力

- **知识外推：**对互联网不存在的知识内容能给出合理答案
- **心理推测：**通过有限信息推测用户意图
- **规则理解：**能够理解并应用全新的游戏规则
- **创造性思维：**在学术建模等复杂任务中展现创造力

31.5　生成参数配置优化

31.5.1　关键参数调优建议

```
# 生成参数推荐配置
generation_config = {
    "top_p": 0.9,            # 适度增加核采样概率阈值
    "temperature": 1.0,      # 避免概率分布两极分化
```

```
    "do_sample": True,        # 启用多inomial采样解码
    "num_beams": 4,           # Beam Search的beam数量
    "repetition_penalty": 1.8,  # 重复惩罚系数
    "no_repeat_ngram_size": 6,   # 禁止重复的n-gram大小
}
```

31.5.2　参数详细说明

top_p（核采样）

- **作用**：控制候选 token 集合的大小
- **调优建议**：0.9 可增加生成多样性
- **影响**：值越大，候选 token 越多，生成越多样

temperature（温度参数）

- **作用**：调整 softmax 输出的分布平滑度
- **调优建议**：1.0 避免极端分布，0.01 接近贪婪解码
- **应用场景**：
  - 创造性任务：较高 temperature（1.0-1.5）
  - 确定性任务：较低 temperature（0.1-0.5）

repetition_penalty（重复惩罚）

- **问题识别**：生成内容出现重复时调高此参数
- **推荐范围**：1.5-2.0
- **机制**：降低已出现 token 的再现概率

任务特定调优策略

31.6　内存高效推理方法

31.6.1　内存需求估算方法

精度对内存的影响

$$\text{FP32}: \text{每个参数需要32 bits} = 4 \text{ bytes}$$
$$\text{FP16}: \text{每个参数需要16 bits} = 2 \text{ bytes}$$
$$\text{INT8}: \text{每个参数需要8 bits} = 1 \text{ byte}$$

表 31.3: 不同任务类型的参数配置建议

任务类型	特点	参数配置建议
创造性写作	需要多样性、新颖性	temperature=1.2, top_p=0.95, do_sample=True
技术问答	需要准确性、一致性	temperature=0.7, top_p=0.8, do_sample=False
代码生成	需要结构化、精确	temperature=0.3, top_p=0.9, repetition_penalty=1.5
对话系统	需要自然、流畅	temperature=1.0, top_p=0.9, do_sample=True

内存组成分析

推理内存需求主要包括三个部分：

1. **模型参数**：参数量 × 每个参数所需内存
2. **梯度信息**：训练时需要，推理时通常不需要
3. **优化器状态**：训练时需要，推理时不需要
4. **CUDA 内核**：约 1.3GB 固定开销

LLaMA-6B 内存估算示例

表 31.4: LLaMA-6B 模型在不同精度下的内存需求估算

组件	FP32	FP16	INT8
模型参数	6B × 4B = 24GB	6B × 2B = 12GB	6B × 1B = 6GB
梯度	24GB	12GB	6GB
优化器 (AdamW)	48GB	24GB	12GB
CUDA 内核	1.3GB	1.3GB	1.3GB
总计	**97.3GB**	**49.3GB**	**25.3GB**

中间变量内存计算

对于 LLaMA 架构（hidden_size=4096, intermediate_size=11008, num_hidden_layers=32, context_length=2048）：

$$\begin{aligned}\text{单实例内存} &= (\text{hidden\_size} + \text{intermediate\_size}) \times \text{context\_length} \times \text{num\_layers} \times 1 \text{ byte} \\ &= (4096 + 11008) \times 2048 \times 32 \times 1 \text{ byte} = 990 \text{ MB}\end{aligned}$$

31.6.2　FP16 混合精度推理

技术原理

混合精度训练的核心思想：

- 前向传播和梯度计算使用 FP16 加速
- 参数更新使用 FP32 保持精度

PyTorch 实现

```
import torch
from torch.cuda.amp import autocast, GradScaler

# 模型转换
model.eval()
model.half()  # 转换为FP16

# 使用自动混合精度
scaler = GradScaler()
with autocast():
    output = model(input)
    loss = criterion(output, target)

scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

HuggingFace Transformers 集成

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    fp16=True,  # 启用FP16训练
    per_device_train_batch_size=8,
    gradient_accumulation_steps=4,
    # ... 其他参数
)
```

31.6.3　INT8 量化推理

技术挑战与解决方案

INT8 量化的主要挑战和解决方案：

- **精度损失**：使用 vector-wise quantization 和 mixed precision decomposition
- **异常值处理**：LLM.int8() 对 outlier 进行特殊处理

量化实现原理

1. **向量化量化**：按向量维度进行量化，保持相对精度
2. **混合精度分解**：对敏感部分保持更高精度
3. **异常值分离**：将异常值与正常值分开处理

HuggingFace 集成示例

```
from transformers import AutoModel, BitsAndBytesConfig

# 配置INT8量化
bnb_config = BitsAndBytesConfig(
    load_in_8bit=True,
    llm_int8_enable_fp32_cpu_offload=True,
)

model = AutoModel.from_pretrained(
    "facebook/opt-6.7b",
    quantization_config=bnb_config,
    device_map="auto",
)
```

31.6.4　LoRA 低秩适配推理

核心思想

LoRA 发现微调时的更新矩阵通常是低秩的，因此可以将更新矩阵重新参数化为两个低秩矩阵的乘积：

$$\Delta W = BA$$

其中 $B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times k}$，且 $r \ll d, r \ll k$

参数效率分析

原始参数量：$d \times k$

LoRA 参数量：$d \times r + r \times k$

参数减少比例：$\frac{d \times r + r \times k}{d \times k} = \frac{r}{k} + \frac{r}{d}$

实际应用

```python
from peft import LoraConfig, get_peft_model

# 配置LoRA
lora_config = LoraConfig(
    r=8,                    # 秩
    lora_alpha=32,          # 缩放因子
    target_modules=["q_proj", "v_proj"],  # 目标模块
    lora_dropout=0.1,       # Dropout率
)

model = get_peft_model(model, lora_config)
```

31.6.5　梯度检查点技术

技术原理

梯度检查点通过时间换空间的策略优化显存使用：

- 前向传播时不存储中间激活值
- 反向传播时重新计算所需激活值
- 显著减少动态显存占用

PyTorch 实现

```python
import torch
import torch.utils.checkpoint as checkpoint

# 使用梯度检查点
def forward_with_checkpoint(input):
    def custom_forward(x):
        return model(x)

    return checkpoint.checkpoint(custom_forward, input)

```

```
output = forward_with_checkpoint(input)
```

HuggingFace 集成

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    gradient_checkpointing=True,  # 启用梯度检查点
    per_device_train_batch_size=16,
    # ... 其他参数
)
```

31.6.6 Torch FSDP + CPU Offload

完全分片数据并行

FSDP 通过 ZeRO-like 算法分布式存储模型状态：

- 模型参数分片到多个 GPU
- 优化器状态分布式存储
- 梯度聚合时进行通信

CPU Offload 技术

- 动态将参数在 GPU 和 CPU 间迁移
- 反向传播时按需加载参数
- 进一步扩展可用显存容量

实现示例

```
import torch
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
from torch.distributed.fsdp import CPUOffload

# 配置FSDP + CPU Offload
model = FSDP(
    model,
    cpu_offload=CPUOffload(offload_params=True),
    sharding_strategy=ShardingStrategy.SHARD_GRAD_OP,
)

```

```
# 仅分片梯度和优化器状态（更稳定）
model = FSDP(
    model,
    sharding_strategy=ShardingStrategy.SHARD_GRAD_OP,
)
```

31.7　推理输出合规化处理

31.7.1　合规化处理流程

1. **内容生成**：大模型生成原始回答
2. **向量化表示**：将生成内容转换为向量表示
3. **相似度检索**：在话术向量库中检索最相似内容
4. **阈值判断**：根据相似度得分决定输出策略
5. **兜底处理**：低相似度时使用预设话术

31.7.2　多级兜底策略

表 31.5: 基于对话阶段的兜底话术策略

对话阶段	触发条件	兜底话术示例
开场阶段	相似度 <0.3	”您好，请问有什么可以帮您？”
需求了解	相似度 <0.5	”能详细说说您的具体需求吗？”
方案推荐	相似度 <0.7	”根据您的需求，我建议考虑以下方案”
成交引导	相似度 <0.6	”这个方案您觉得怎么样？需要进一步了解吗？”
通用兜底	其他情况	”抱歉，我没有完全理解您的意思，能否换种方式说明？”

31.8　应用模式优化策略

31.8.1　模式演进分析

31.8.2　混合模式优化实践

1. **前端引导**：使用小模型进行意图识别和话术策略引导

表 31.6: 不同应用模式的对比分析

应用模式	优势	局限性
纯大模型 AI 模式	对话自然，理解能力强	用户表达发散时难以收敛
传统 AI+ 人工模式	流程可控，转化率稳定	灵活性较差，成本较高
混合 AI 模式	平衡灵活性与可控性	需要精细的流程设计

2. **深度交互**：对有意向用户切换到大模型进行自然对话
3. **流程控制**：确保对话在可控范围内进行
4. **效果监控**：实时评估不同模式的转化效果

31.9　输出分布稀疏性处理

31.9.1　问题分析

大模型输出分布稀疏性的表现和影响：

- **概率极化**：少数 token 概率过高，多数 token 概率接近 0
- **生成单一**：导致输出缺乏多样性和创造性
- **过度自信**：模型对某些模式过度依赖

31.9.2　解决方案

温度参数调节

使用 softmax 温度参数平滑输出分布：

$$P_i = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

其中 τ 为温度参数：

- $\tau > 1$：分布更平滑，生成更多样
- $\tau < 1$：分布更尖锐，生成更确定

正则化技术

- **Dropout**：在训练时随机丢弃部分神经元，防止过度依赖特定特征
- **标签平滑**：将 one-hot 标签转换为软标签，减轻过度自信
- **知识蒸馏**：使用教师模型的软标签训练学生模型

高级平滑技术

```python
import torch
import torch.nn.functional as F

def smoothed_softmax(logits, temperature=1.0, alpha=0.1):
    """
    带平滑的softmax函数
    """
    # 温度调节
    scaled_logits = logits / temperature

    # 标签平滑
    probs = F.softmax(scaled_logits, dim=-1)
    smoothed_probs = (1 - alpha) * probs + alpha / logits.size(-1)

    return smoothed_probs

# 在推理时应用
logits = model(input_ids)
probs = smoothed_softmax(logits, temperature=1.2, alpha=0.1)
```

31.10　实践建议与最佳实践

31.10.1　硬件选型建议

表 31.7: 不同规模模型的硬件配置建议

模型规模	最低配置	推荐配置	理想配置
<3B 参数	16GB GPU	24GB GPU (RTX 4090)	40GB GPU (A100)
3B-13B 参数	24GB GPU	40GB GPU (A100)	80GB GPU (A100)
13B-70B 参数	40GB GPU	80GB GPU (A100)	多卡并行
>70B 参数	多卡并行	多卡 FSDP	专家混合

31.10.2　推理流水线优化

```python
class OptimizedInferencePipeline:
    def __init__(self, model_name, device="cuda"):
```

```
        self.model = AutoModel.from_pretrained(
            model_name,
            torch_dtype=torch.float16,  # FP16优化
            device_map="auto",
            low_cpu_mem_usage=True,
        )
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)

        # 启用优化
        self.model.eval()
        if hasattr(self.model, "gradient_checkpointing_enable"):
            self.model.gradient_checkpointing_enable()

    def generate(self, prompt, **kwargs):
        # 默认优化参数
        default_kwargs = {
            "max_length": 512,
            "temperature": 1.0,
            "top_p": 0.9,
            "do_sample": True,
            "repetition_penalty": 1.2,
        }
        default_kwargs.update(kwargs)

        inputs = self.tokenizer(prompt, return_tensors="pt").to(self.
            model.device)

        with torch.no_grad():
            outputs = self.model.generate(**inputs, **default_kwargs)

        return self.tokenizer.decode(outputs[0], skip_special_tokens=True
            )
```

31.11　总结与展望

本章详细分析了大模型推理过程中的关键技术挑战和解决方案。从显存优化到速度提升，从参数配置到输出处理，我们提供了全面的技术指导。随着大模型技术的不断发展，推理优化将继续是研究和应用的重点方向。

未来的发展趋势包括：

- **量化技术革新**：更高效的量化算法和硬件支持
- **推理专用架构**：针对推理优化的模型架构设计
- **边缘推理**：在资源受限设备上的高效推理
- **多模态推理**：结合文本、图像、音频的多模态推理优化

通过综合应用本章介绍的各种技术，可以在保证推理质量的前提下，显著提升大模型的推理效率和资源利用率。

第三十二章　大模型 (LLMs) 增量预训练技术详解

32.1　引言：为什么需要增量预训练？

32.1.1　增量预训练的理论基础

当前大模型技术发展形成了明确的技术路径分工：预训练阶段主要学习通用知识，指令微调阶段学习特定格式和对话模式，强化学习阶段则用于对齐人类偏好。LIMA 等相关论文为这一技术路径提供了实证支持。

基于这一理论基础，要让大模型掌握特定领域知识，必须通过增量预训练来实现。单纯依靠指令微调来注入领域知识是不现实的，因为这需要数十万条高质量的标注数据，成本极高且效果有限。

32.1.2　增量预训练的核心价值

- **知识注入**：将领域专业知识有效注入预训练模型
- **成本优化**：相比从头预训练，大幅降低计算成本
- **效果保证**：在保持原有能力的基础上增强特定领域表现
- **灵活适配**：可根据业务需求进行多轮迭代优化

32.2　增量预训练准备工作

32.2.1　模型底座选型策略

主流模型选择考量

选型关键因素

1. **Scaling 法则验证**：LLaMA 系列经过充分验证，是较为稳妥的选择
2. **版权考量**：商业应用需重点关注许可证条款
3. **生态完善度**：成熟的生态有助于降低工程复杂度

表 32.1: 主流预训练模型选型对比

模型	优势	劣势	推荐指数
LLaMA 系列	Scaling 法则验证充分，预训练质量高	存在版权风险	
BLOOM 系列	完全开源，可商用	基座效果相对较差	
Falcon 系列	许可证友好，技术先进	训练语料缺少中文	
ChatGLM 系列	中文优化良好	在 SFT 模型上增量效果待验证	
国产模型（Baichuan 等）	中文支持好，许可证友好	生态相对不成熟	

4. **架构统一性**：LLaMA-like 架构便于技术迁移和优化

32.2.2　数据收集策略

高质量数据源推荐

- **通用语料**：WuDao Corpus（200GB）、The Pile（800GB）等经典开源预训练数据集
- **领域语料**：根据目标领域收集 GB 级别的专业文本数据
- **数据规模**：初期实验阶段 1-10GB 即可验证流程，生产环境需要 TB 级别

数据收集原则

```
# 推荐的数据收集优先级
1. 高质量开源数据集（WuDao、The Pile等）
2. 领域权威文献和教科书
3. 经过清洗的网页爬取数据
4. 专业论坛和社区内容
5. 合成数据（谨慎使用）
```

32.2.3　数据清洗流程

清洗关键步骤

借鉴 Falcon 论文中的数据清洗方法，推荐以下流程：

1. **去广告**：移除网页数据中的广告内容
2. **去重**：基于内容哈希或语义相似度去重
3. **质量过滤**：基于语言质量、信息密度等指标过滤
4. **毒性内容过滤**：移除不当或有害内容

5. **格式标准化：**统一文本格式和编码

清洗工具推荐

```
# 数据清洗工具链示例
import hashlib
import re
from bs4 import BeautifulSoup

def clean_web_data(html_content):
    """网页数据清洗"""
    soup = BeautifulSoup(html_content, 'html.parser')

    # 移除广告和导航元素
    for element in soup.find_all(['script', 'style', 'nav', 'footer']):
        element.decompose()

    # 提取主要内容
    main_content = soup.get_text()

    # 进一步文本清洗
    cleaned_text = re.sub(r'\s+', ' ', main_content).strip()
    return cleaned_text

def deduplicate_documents(documents):
    """基于内容的去重"""
    seen_hashes = set()
    unique_docs = []

    for doc in documents:
        content_hash = hashlib.md5(doc.encode()).hexdigest()
        if content_hash not in seen_hashes:
            seen_hashes.add(content_hash)
            unique_docs.append(doc)

    return unique_docs
```

32.3　训练框架选择

32.3.1　超大规模训练框架

3D 并行训练

对于真正的大规模训练（千卡以上），推荐使用 3D 并行框架：

- **Megatron-DeepSpeed**：业界标杆，有多个成功案例
- **参考实现**：可参考 LydiaXiaohongLi 大佬的 LLaMA 实现

```
https://github.com/microsoft/Megatron-DeepSpeed/pull/139
```

- **BLOOM 训练**：参考 BigScience 的官方仓库

32.3.2　中小规模训练框架

单节点/多节点训练

- **高速网络环境**：直接使用 DeepSpeed ZeRO
- **推荐实现**：Open-Llama 的 fork 版本

```
https://github.com/RapidAI/Open-Llama
```

- **低速网络环境**：考虑流水线并行
- **参考实现**：transpeeder 项目

```
https://github.com/HuangLK/transpeeder
```

张量并行注意事项

- 仅在 NVLink 环境下有正向收益
- 性能提升有限，复杂度较高
- 建议优先考虑 ZeRO 系列优化

32.3.3　资源受限环境训练

LoRA 微调方案

在显存严重不足时，可采用 LoRA 进行参数高效微调：

- **适用场景**：单卡或显存严重受限环境
- **参考实现**：MedicalGPT 项目

```
https://github.com/shibing624/MedicalGPT
```

- **优势**：极大降低显存需求
- **劣势**：效果可能不如全参数微调

32.4　完整训练流程

32.4.1　数据预处理

文本长度处理

参照 LLaMA 的预训练配置，推荐处理策略：

- **序列长度**：2048 tokens（与原始 LLaMA 保持一致）
- **填充策略**：不足部分进行 padding
- **截断策略**：过长序列进行截断或分块

预处理注意事项

```python
# 数据预处理示例
def preprocess_texts(texts, max_length=2048, tokenizer):
    """文本预处理流程"""
    processed_texts = []

    for text in texts:
        # 分词
        tokens = tokenizer.encode(text)

        # 长度处理
        if len(tokens) > max_length:
            # 策略1: 截断
            tokens = tokens[:max_length]
            # 策略2: 分块（适用于长文档）
            # chunks = [tokens[i:i+max_length] for i in range(0, len(tokens), max_length)]
        else:
            # 填充
            tokens = tokens + [tokenizer.pad_token_id] * (max_length - len(tokens))

        processed_texts.append(tokens)

    return processed_texts
```

32.4.2　分词器选择

分词器选型建议

- **原版词表**：优先使用原始 500K 的 tokenizer.model
- **中文优化**：可考虑 Chinese-LLaMA-Alpaca 的中文增强词表

```
https://github.com/ymcui/Chinese-LLaMA-Alpaca
```

- **选择依据**：目前尚无定论表明中文词表一定更好，建议通过实验验证

32.4.3　模型加载与转换

模型格式处理

- **层名对齐**：不同框架的模型层命名可能不同，需要转换脚本
- **格式转换**：准备模型加载脚本，确保能成功加载
- **中文优化模型**：可考虑使用经过中文增量预训练的版本作为基础

模型转换示例

```
def convert_model_format(source_path, target_path, config):
    """模型格式转换"""
    # 加载源模型
    source_model = load_source_model(source_path)

    # 层名映射和参数转换
    converted_state_dict = {}
    for src_name, param in source_model.state_dict().items():
        tgt_name = layer_name_mapping(src_name)
        converted_state_dict[tgt_name] = param

    # 保存转换后模型
    torch.save(converted_state_dict, target_path)
```

32.4.4　训练参数配置

基础参数设置

```
training_config = {
    "per_device_train_batch_size": 8,
    "gradient_accumulation_steps": 4,
```

```
    "learning_rate": 3e-5,  # 约为预训练的10%
    "num_train_epochs": 3,
    "max_steps": -1,
    "lr_scheduler_type": "cosine",
    "warmup_ratio": 0.03,  # 3个epoch对应3%
    "weight_decay": 0.1,
}
```

显存优化配置

```
# DeepSpeed ZeRO配置
deepspeed_config = {
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu"
        },
        "offload_param": {
            "device": "cpu"
        }
    },
    "fp16": {
        "enabled": True
    },
    "train_batch_size": 32,
}
```

32.4.5　训练监控与分析

关键监控指标

- **Loss 曲线**：监控训练收敛情况
- **吞吐量**：tokens/秒，评估训练效率
- **FLOPs 利用率**：评估硬件利用效率
- **测试 PPL**：在验证集上的困惑度
- **显存使用**：监控资源消耗情况

监控工具推荐

- **W&B**：完整的实验跟踪和可视化

- **TensorBoard**：标准的训练监控
- **自定义日志**：关键指标的定期记录

32.4.6　模型转换与测试

Checkpoint 转换流程

以 ZeRO 训练为例的转换流程：

1. **ZeRO to FP32**：将分布式参数合并为 FP32 精度
2. **FP32 to FP16**：转换为 FP16 精度减少存储
3. **转换为 HuggingFace 格式**：生成标准格式的模型文件

转换脚本示例

```
def zero_to_huggingface(zero_checkpoint_path, hf_save_path):
    """ZeRO checkpoint转换为HuggingFace格式"""
    # 加载ZeRO checkpoint
    zero_state_dict = torch.load(zero_checkpoint_path)

    # 参数合并和转换
    merged_state_dict = {}
    for key, param in zero_state_dict.items():
        if 'lora' not in key:  # 排除LoRA参数
            merged_state_dict[key] = param.half()  # 转换为FP16

    # 保存为标准格式
    model = AutoModel.from_pretrained(base_model_name)
    model.load_state_dict(merged_state_dict)
    model.save_pretrained(hf_save_path)
```

模型测试验证

- **基础功能测试**：使用 text-generation-webui 等工具验证生成能力
- **领域知识测试**：设计领域相关的测试用例
- **稳定性测试**：长时间运行的稳定性验证

32.5　数据量要求与规划

32.5.1　最小数据量要求

- **绝对下限**：至少数十亿 tokens（几 GB 文本数据）
- **推荐规模**：百亿级别 tokens 以上效果更佳
- **小数据场景**：如只有几十条数据，推荐使用模型微调而非增量预训练

32.5.2　数据量规划建议

表 32.2: 不同目标下的数据量规划

目标	数据规模	训练周期	预期效果
概念验证	1-10B tokens	1-3 天	基础领域能力
生产试用	10-100B tokens	1-2 周	可用级效果
商业部署	100B+ tokens	1 月以上	行业领先水平

32.6　训练过程关键问题处理

32.6.1　Loss 上升现象分析

正常 Loss 上升场景

- **训练初期**：模型需要适应新数据分布，短期上升属正常现象
- **学习率调整**：学习率过大可能导致初期 Loss 上升
- **数据分布变化**：新旧数据分布差异较大时可能出现

异常 Loss 上升排查

1. **学习率过大**：检查并调整学习率设置
2. **数据质量问题**：检查训练数据质量和分布
3. **梯度爆炸**：添加梯度裁剪，检查梯度范数
4. **数值稳定性**：检查是否存在数值溢出问题

32.6.2　学习率调优策略

学习率设置原则

- **过大风险**：Loss 收敛困难，原有能力损失严重

- **过小风险**：难以学习新知识，训练效率低下
- **推荐范围**：通常为原始预训练学习率的 5-20%

具体设置建议

对于 7B 模型，预训练阶段学习率通常为 3e-4，增量预训练推荐：

$$\text{增量预训练学习率} = 3e-4 \times 10\% = 3e-5$$

Batch Size 缩放规则

学习率应按 Batch Size 的平方根进行缩放：

$$\text{缩放后学习率} = \text{基础学习率} \times \sqrt{\frac{\text{新 Batch Size}}{\text{原 Batch Size}}}$$

例如 Batch Size 增大 4 倍，学习率应扩大 2 倍。

32.6.3　Warmup 比例设置

一般设置规则

- **预训练**：通常 1 个 epoch，warmup 比例约 1%
- **指令微调**：通常 3 个 epoch，warmup 比例约 3%
- **增量预训练**：建议适当增大 warmup 比例

增量预训练特殊考量

- **大数据集**：几百 B tokens 以上，warmup 影响较小
- **小数据集**：需要更大的 warmup 比例实现平滑过渡
- **学习率协调**：大学习率配合大 warmup 防止训练崩溃

32.7　关键参数实验分析

32.7.1　Warmup 步数影响分析

实验设计

通过对比不同 warmup 比例（0%，0.5%，1%，2%）的实验结果，分析 warmup 步数对训练效果的影响。

实验结果

- **充分训练后**：各种 warmup 步数的最终性能相近
- **训练前期**：较长 warmup（2%）表现最佳
- **下游任务**：长 warmup 学习更快
- **上游任务**：长 warmup 遗忘更慢

实践建议

- **资源充足**：选择适中 warmup 比例（0.5-1%）
- **资源受限**：选择较长 warmup（2%）保证前期稳定性
- **大数据训练**：warmup 影响较小，可按标准设置

32.7.2　学习率大小影响分析

实验设计

对比 4 种不同最大学习率设置下的上下游任务表现。

实验结果

- **大学习率**：下游任务效果最好，但上游任务遗忘严重
- **小学习率**：上下游任务平衡较好，但需要更长时间训练
- **未预训练模型**：效果全面不如预训练模型

重要发现

- **训练前期**：大学习率可能导致 Loss 大幅上升后下降
- **资源约束**：在计算资源有限时，小学习率 + 长 warmup 是更稳妥选择
- **最终性能**：充分训练后大学习率有优势，但需要度过不稳定期

32.7.3　Rewarmup 策略影响分析

实验设计

在原始预训练数据集上继续训练，比较 warmup 策略与常量学习率的效果。

重要发现

- **性能损伤**：在原始数据上使用 warmup 会造成性能下降
- **不可恢复**：这种损伤无法在后续训练中恢复
- **学习率越大**：性能损伤越严重

实践指导

- **训练恢复**：中断后恢复训练时应保持原有学习率状态
- **学习率策略**：避免在连续训练中不必要地重置学习率
- **计划制定**：提前规划完整的训练周期，避免频繁中断重启

32.8　增量预训练最佳实践

32.8.1　完整工作流总结

1. **数据准备阶段**
   - 收集高质量领域数据（GB 到 TB 级别）
   - 进行严格的数据清洗和去重
   - 按 2048 长度进行预处理
2. **环境配置阶段**
   - 根据资源情况选择合适训练框架
   - 配置监控和实验跟踪工具
   - 准备模型转换和测试流程
3. **训练执行阶段**
   - 采用保守的学习率策略（3e-5 左右）
   - 设置适当的 warmup 比例（1-3%）
   - 密切监控 Loss 曲线和资源使用
4. **效果验证阶段**
   - 进行完整的模型转换和格式标准化
   - 设计多维度的评估方案
   - 与基线模型进行对比测试

32.8.2　故障排查指南

表 32.3: 常见问题及解决方案

问题现象	可能原因	解决方案
Loss 持续上升	学习率过大、数据质量问题	降低学习率、检查数据质量
训练速度过慢	配置不当、资源瓶颈	优化数据加载、检查硬件状态
显存溢出	Batch Size 过大、模型配置问题	减小 Batch Size、使用梯度累积
性能不提升	学习率过小、数据不足	调整学习率、增加数据量

32.8.3　持续优化建议

- **渐进式优化**：从小规模实验开始，逐步扩大数据量和模型规模
- **多轮迭代**：通过多轮增量预训练持续优化模型效果
- **效果评估**：建立科学的评估体系，客观衡量改进效果
- **经验沉淀**：总结成功经验，形成可复用的最佳实践

32.9　总结与展望

增量预训练作为大模型领域知识注入的关键技术，在保持预训练模型通用能力的同时，能够有效提升其在特定领域的表现。通过合理的数据准备、框架选择、参数调优和过程监控，可以构建出高质量的领域大模型。

未来发展方向包括：

- **自动化调优**：开发更智能的超参数自动优化算法
- **多模态扩展**：支持文本、图像、语音等多模态增量学习
- **高效算法**：研究更参数高效的增量预训练方法
- **评估体系**：建立更科学的领域能力评估标准

第三十三章　大模型增量预训练样本拼接技术详解

33.1　引言：为什么需要样本拼接？

33.1.1　样本拼接的核心价值

在预训练阶段，为了提高训练效率和扩展大语言模型的最大序列长度，随机将多条短文本拼接成长序列是一种常见且有效的技术手段。样本拼接技术通过优化数据组织方式，为大模型训练带来多重收益：

- **训练效率提升**：减少 padding 比例，提高 GPU 利用率
- 序列长度扩展：使模型适应更长上下文窗口
- 计算效率优化：充分利用现代硬件的并行计算能力
- 长文本能力培养：增强模型处理长文档的潜力

33.1.2　技术挑战与机遇

尽管样本拼接技术优势明显，但也面临重要挑战：大多数情况下，构成训练样本的多个示例彼此语义不相关，无法提供有效的上下文信息，模型难以从扩展的上下文窗口中获得有意义的反馈。特别是在语料规模有限、分布集中的场景下，模型可能从偶然的噪声共现中学习到错误的特征模式。

33.2　样本拼接方法综述

33.2.1　方法一：随机拼接（Random Concatenate）

基本实现原理

随机拼接是最基础的样本拼接策略，其核心思想是将多个短文本 $\{examples_i\}$ 随机组合成更长的训练样本 $\{examples_k\}$，以充分利用预设的最大序列长度（maxLen）。

```
def random_concatenate(texts, max_length=2048, tokenizer):
    """随机拼接短文本为长序列"""
```

```
    concatenated_texts = []
    current_text = ""

    for text in texts:
        # 随机打乱文本顺序
        shuffled_texts = random.shuffle(texts)

        for text in shuffled_texts:
            # 检查当前文本长度
            tokens = tokenizer.encode(current_text + text)
            if len(tokens) <= max_length:
                current_text += text
            else:
                # 保存当前序列并开始新序列
                concatenated_texts.append(current_text)
                current_text = text

    # 添加最后一个序列
    if current_text:
        concatenated_texts.append(current_text)

    return concatenated_texts
```

优势分析

- **实现简单**：无需复杂的预处理和语义分析
- **计算高效**：随机组合的计算开销极小
- **数据利用率高**：几乎可以 100% 利用原始文本数据
- **通用性强**：适用于各种类型和领域的文本数据

局限性分析

1. **语义连贯性缺失**：随机组合的文本间缺乏语义关联，模型难以学习有意义的上下文依赖
2. **噪声共现风险**：在语料有限时，偶然的文本共现可能被模型误认为有效模式
3. **长文本理解挑战**：模型可能无法从无关文本的拼接中真正学会处理长文档

改进尝试：特殊标记隔离

部分研究尝试使用特殊标记（specialToken）对拼接的文本进行软隔离，但缺乏有效的正则化手段时，这种方法可能陷入”鸡生蛋、蛋生鸡”的循环依赖问题。

33.2.2　方法二：随机拼接 + 噪声掩码（Random Concatenate + Noise-Mask）

技术动机

为了解决随机拼接中无关文本间的噪声干扰问题，该方法通过自定义注意力掩码（attentionMask）机制，限制模型在每个训练样本中只关注当前正在处理的文本片段。

核心实现

```
import torch
import torch.nn as nn

def segment_causal_mask(input_ids, device, eos_token_id, val=float("-inf")):
    """
    生成分段因果掩码，使模型只关注当前文本片段
    """
    bsz, tgt_len = input_ids.shape

    # 计算每个序列中EOS标记的累积位置
    cum_lens = torch.arange(1, tgt_len + 1, device=device).unsqueeze(0) * \
               torch.eq(input_ids, eos_token_id).int().to(device)

    # 初始化掩码矩阵
    mask = torch.zeros([bsz, tgt_len, tgt_len]).to(device)

    # 为每个批次生成掩码
    for i, _cum_lens in enumerate(cum_lens):
        for v in _cum_lens:
            if v > 0:  # 有效的EOS位置
                # 屏蔽当前片段之后对之前片段的注意力
                mask[i, v:, :v] = val

    return mask

class SegmentCausalAttention(nn.Module):
    """分段因果注意力层"""
    def __init__(self, config):
        super().__init__()
```

```
        self.config = config
        # 标准的注意力层初始化
        self.attention = nn.MultiheadAttention(
            embed_dim=config.hidden_size,
            num_heads=config.num_attention_heads
        )

    def forward(self, hidden_states, attention_mask=None):
        if attention_mask is not None:
            # 应用分段因果掩码
            causal_mask = segment_causal_mask(
                input_ids,
                device=hidden_states.device,
                eos_token_id=self.config.eos_token_id
            )
            if attention_mask is not None:
                attention_mask = attention_mask + causal_mask
            else:
                attention_mask = causal_mask

        return self.attention(
            hidden_states, hidden_states, hidden_states,
            attn_mask=attention_mask
        )
```

实验效果

经实际测试，相比基础的随机拼接方法，噪声掩码技术在少样本上下文学习（ICL few-shot）任务上能带来约 1.6% 的性能提升。

技术局限性

- **位置编码冲突：**相对位置编码（如 ALiBi、RoPE）的 token 级相对位置信息可能被注意力掩码破坏
- **跨片段学习缺失：**模型无法从跨文本片段的交互中获得有意义的反馈
- **长文本训练受限：**本质上仍在短文本窗口内训练，未实现真正的长序列建模

深层分析

即使在不使用注意力掩码的标准随机拼接中，模型是否真的能从无关文本的扩展上下文中学习有效的长距离依赖关系仍然存疑。当数据分布表现为远距离 token 普遍不相关时，模型可能自然学会忽略较远的上下文信息，这或许是多数大模型在扩展序列长度后长文本处理效果仍不理想的原因之一。

33.2.3　方法三：随机拼接 + 聚类（Random Concatenate + Cluster）

创新思路

为了在不使用注意力掩码的前提下减少噪声干扰，同时让模型从扩展上下文中受益，该方法基于实体、语义等维度对文本进行聚类，将有语义关联的文本组织在同一训练样本中。

聚类策略

```python
from sklearn.cluster import KMeans
from sentence_transformers import SentenceTransformer
import numpy as np

class SemanticClusterConcatenate:
    """基于语义聚类的样本拼接"""

    def __init__(self, model_name='all-MiniLM-L6-v2'):
        self.encoder = SentenceTransformer(model_name)

    def cluster_concatenate(self, texts, max_length=2048, n_clusters=10,
        tokenizer):
        """基于语义聚类进行样本拼接"""
        # 生成文本嵌入
        embeddings = self.encoder.encode(texts)

        # K-means聚类
        kmeans = KMeans(n_clusters=n_clusters, random_state=42)
        clusters = kmeans.fit_predict(embeddings)

        # 按聚类结果组织文本
        clustered_texts = {}
        for text, cluster_id in zip(texts, clusters):
            if cluster_id not in clustered_texts:
                clustered_texts[cluster_id] = []
```

```
            clustered_texts[cluster_id].append(text)

        # 在每个聚类内进行拼接
        concatenated_texts = []
        for cluster_id, cluster_texts in clustered_texts.items():
            current_text = ""
            for text in cluster_texts:
                tokens = tokenizer.encode(current_text + text)
                if len(tokens) <= max_length:
                    current_text += text
                else:
                    concatenated_texts.append(current_text)
                    current_text = text
            if current_text:
                concatenated_texts.append(current_text)

        return concatenated_texts
```

技术挑战

1. **信息重复问题**：基于实体的聚类容易导致相似内容的过度重复
2. **信息泄露风险**：即使经过关键词和语义去重，仍难以完全避免训练-测试泄露
3. **记忆化倾向**：模型可能从重复模式中学习复制而非理解

实体聚类实践

基于实体的聚类实验发现，虽然能提高语义连贯性，但面临信息重复和潜在泄露的挑战，这可能是该方法在实验中未表现出显著优势的原因。

33.2.4　方法四：上下文预训练（IN-CONTEXT PRETRAINING）

核心思想

上下文预训练是一种先进的样本拼接策略，其基本思想是基于语义相似度，优先将语义相关的文本进行拼接，构建语义连贯的扩展上下文。

算法流程

上下文预训练包含四个关键步骤：

1. **文本嵌入化**：使用预训练编码器（如 Contriever）将文本转换为向量表示
2. **数据去重**：基于余弦距离进行语义级别的数据去重

3. **相似度串联：**借鉴旅行商问题思想，按语义相似度串联相关文档
4. **模型预训练：**基于拼接后的长序列进行预训练

技术实现细节

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer

class InContextPretraining:
    """上下文预训练样本拼接器"""

    def __init__(self, model_name='sentence-transformers/all-mpnet-base-v2'):
        self.encoder = SentenceTransformer(model_name)

    def build_document_graph(self, documents, similarity_threshold=0.7):
        """构建文档相似度图"""
        # 生成文档嵌入
        embeddings = self.encoder.encode(documents)

        # 计算相似度矩阵
        similarity_matrix = cosine_similarity(embeddings)

        # 构建图结构
        graph = {}
        n_docs = len(documents)

        for i in range(n_docs):
            graph[i] = []
            for j in range(n_docs):
                if i != j and similarity_matrix[i][j] > similarity_threshold:
                    graph[i].append((j, similarity_matrix[i][j]))

        return graph, embeddings

    def tsp_like_concatenation(self, documents, graph, max_length=8192, tokenizer):
        """类似旅行商问题的文档串联"""
```

```
        visited = set()
        concatenated_sequences = []

        for start_node in range(len(documents)):
            if start_node in visited:
                continue

            current_sequence = documents[start_node]
            visited.add(start_node)
            current_node = start_node

            while True:
                # 查找最相似的未访问邻居
                neighbors = [n for n in graph[current_node] if n[0] not
                    in visited]
                if not neighbors:
                break

                # 选择最相似的文档
                next_node, similarity = max(neighbors, key=lambda x: x
                    [1])

                # 检查长度限制
                candidate_sequence = current_sequence + documents[
                    next_node]
                if len(tokenizer.encode(candidate_sequence)) <=
                    max_length:
                    current_sequence = candidate_sequence
                    visited.add(next_node)
                    current_node = next_node
                else:
                    break

            concatenated_sequences.append(current_sequence)

        return concatenated_sequences

    def process(self, documents, max_length=8192, tokenizer):
        """完整的上下文预训练处理流程"""
        # 数据去重
```

```
        unique_documents = self.deduplicate(documents)

        # 构建文档图
        graph, embeddings = self.build_document_graph(unique_documents)

        # 生成拼接序列
        concatenated_sequences = self.tsp_like_concatenation(
            unique_documents, graph, max_length, tokenizer
        )

        return concatenated_sequences

    def deduplicate(self, documents, similarity_threshold=0.95):
        """基于语义相似度的数据去重"""
        if not documents:
            return []

        embeddings = self.encoder.encode(documents)
        unique_indices = []

        for i, emb_i in enumerate(embeddings):
            is_duplicate = False
            for j in unique_indices:
                similarity = cosine_similarity([emb_i], [embeddings[j]])
                    [0][0]
                if similarity > similarity_threshold:
                    is_duplicate = True
                    break
            if not is_duplicate:
                unique_indices.append(i)

        return [documents[i] for i in unique_indices]
```

关键技术创新

- **语义驱动拼接**：基于语义相似度而非随机或规则进行拼接
- **严格去重机制**：有效避免信息重复和记忆化问题
- **图算法优化**：借鉴旅行商问题实现最优串联路径
- **分布平滑性**：相比实体聚类，语义聚类产生的数据分布更加平滑自然

实验验证

通过消融实验验证，数据去重对 ICLM（In-Context Learning Model）性能有显著正向影响。适当的去重操作能够有效降低信息泄露风险，同时保持数据的多样性和代表性。

33.3　方法对比与分析

33.3.1　各方法特性对比

表 33.1: 四种样本拼接方法特性对比

特性	随机拼接	噪声掩码	聚类拼接	上下文预训练
语义连贯性	低	中	高	高
实现复杂度	低	中	高	高
计算开销	低	中	高	高
长文本效果	有限	有限	较好	优秀
防过拟合能力	弱	中	中	强
通用性	高	中	中	中

33.3.2　适用场景建议

随机拼接适用场景

- **大规模预训练**：数据量极大时，随机性有助于提高泛化能力
- **资源受限环境**：计算资源有限时的实用选择
- **基线方法**：作为其他方法的对比基线

噪声掩码适用场景

- **少样本学习**：在 ICL few-shot 任务中表现较好
- **序列建模研究**：需要控制注意力范围的研究场景
- **教学演示**：便于理解注意力机制的工作原理

聚类拼接适用场景

- **领域自适应**：需要增强特定领域知识的场景
- **结构化数据**：文本具有明显主题或实体结构的场景
- **质量优先**：对生成质量要求高于训练效率的场景

上下文预训练适用场景

- **长文本建模**：需要强大多文档理解能力的场景
- **高质量要求**：对生成连贯性和一致性要求极高的场景
- **研究前沿**：探索最新预训练技术的研究工作

33.4　实施建议与最佳实践

33.4.1　数据预处理策略

数据质量保障

1. **严格去重**：实施多层次去重（精确匹配、模糊匹配、语义去重）
2. **质量过滤**：基于语言质量、信息密度等指标进行过滤
3. **毒性检测**：移除不当或有害内容
4. **格式标准化**：统一文本编码和格式规范

规模控制策略

```
def adaptive_concatenation_strategy(documents, max_length, tokenizer,
                                    quality_threshold=0.8):
    """自适应样本拼接策略"""
    # 第一阶段：质量过滤
    high_quality_docs = quality_filter(documents, threshold=
        quality_threshold)

    # 第二阶段：根据数据量选择策略
    if len(high_quality_docs) > 1000000:  # 百万级
        # 大规模数据使用随机拼接保证多样性
        return random_concatenate(high_quality_docs, max_length,
            tokenizer)
    elif len(high_quality_docs) > 100000:  # 十万级
        # 中等规模使用聚类拼接平衡质量效率
        return cluster_concatenate(high_quality_docs, max_length,
            tokenizer)
    else:  # 小规模
        # 小规模使用上下文预训练最大化质量
        return in_context_pretraining(high_quality_docs, max_length,
            tokenizer)
```

33.4.2　超参数调优指南

序列长度选择

- **基线设置**：2048 tokens（与 LLaMA 等主流模型保持一致）
- **资源充足**：4096 或 8192 tokens 以获得更好长文本能力
- **资源受限**：1024 tokens 作为最小可行配置

相似度阈值调优

```
def find_optimal_similarity(documents, tokenizer, max_length=2048):
    """寻找最优相似度阈值"""
    similarity_thresholds = [0.3, 0.5, 0.7, 0.9]
    best_threshold = 0.5
    best_diversity = 0

    for threshold in similarity_thresholds:
        # 生成拼接样本
        concatenated = in_context_pretraining(
            documents, max_length, tokenizer, similarity_threshold=
                threshold
        )

        # 评估样本多样性（示例指标）
        diversity_score = calculate_diversity(concatenated)

        if diversity_score > best_diversity:
            best_diversity = diversity_score
            best_threshold = threshold

    return best_threshold
```

33.5　总结与展望

33.5.1　技术总结

样本拼接作为大模型预训练的关键技术，经历了从简单随机拼接到语义驱动拼接的技术演进。四种主要方法各有优劣，适用于不同的应用场景：

- **随机拼接**：实现简单，适合大规模基础预训练
- **噪声掩码**：有效控制注意力范围，适合特定研究场景

- **聚类拼接**：平衡语义质量和实现复杂度
- **上下文预训练**：提供最优的语义连贯性，适合高质量要求场景

33.5.2　未来发展方向

技术融合创新

- **混合策略**：结合多种方法的优势，发展自适应拼接策略
- **动态调整**：根据训练进度动态调整拼接策略和参数
- **多模态扩展**：将样本拼接技术扩展到多模态数据

算法优化方向

- **高效相似度计算**：开发更高效的语义相似度计算算法
- **智能去重技术**：研究更精准的数据去重和多样性保持技术
- **可扩展架构**：设计支持超大规模数据处理的分布式拼接框架

评估体系完善

- **标准化评估**：建立统一的样本拼接技术评估基准
- **多维度指标**：从质量、效率、多样性等多维度评估方法效果
- **长期影响研究**：研究不同拼接策略对模型长期能力发展的影响

通过持续的技术创新和实践优化，样本拼接技术将为大模型预训练提供更高效、更智能的数据组织方案，推动大语言模型能力的持续提升。

第三十四章　基于 LoRA 的 LLaMA2 二次预训练技术详解

34.1　引言：为什么需要基于 LoRA 的 LLaMA2 二次预训练？

34.1.1　技术背景与动机

随着大语言模型的快速发展，如何高效地使预训练模型适应特定领域或语言成为重要研究方向。传统的全参数微调方法虽然有效，但计算成本高昂，特别是在模型规模不断增大的背景下。

基于 LoRA（Low-Rank Adaptation）的二次预训练技术应运而生，其主要价值体现在：

- **语言适应**：为 LLaMA2 等英文预训练模型添加中文支持能力
- **计算效率**：仅训练少量参数，大幅降低计算资源需求
- **知识保持**：在保持原有知识的基础上注入新领域知识
- **部署便利**：LoRA 适配器可灵活加载和卸载，支持多任务应用

34.1.2　核心优势分析

相比传统的全参数微调，基于 LoRA 的二次预训练具有以下显著优势：

表 34.1: LoRA 二次预训练与传统微调对比

对比维度	基于 LoRA 的二次预训练	传统全参数微调
计算资源	极低，仅需训练少量参数	高昂，需更新全部参数
训练速度	快速，参数更新量小	缓慢，参数更新量大
存储开销	小，仅保存 LoRA 适配器	大，需保存完整模型
知识保持	优秀，基础模型参数冻结	存在灾难性遗忘风险
多任务支持	灵活，可快速切换适配器	笨重，需维护多个模型
部署效率	高，适配器可动态加载	低，需加载完整模型

34.2　LoRA 技术理论基础

34.2.1　本征维度理论

LoRA 技术的理论基础源于本征维度（Intrinsic Dimension）理论。Aghajanyan 等研究者的工作表明，预训练模型的内在维度实际上非常小，只有一小部分参数对模型输出有显著影响。

数学表达为：存在一个极低维度的参数子空间，在该子空间内进行微调可以达到与全参数空间微调相近的效果。

34.2.2　低秩假设

LoRA 基于的核心假设是：模型在任务适配过程中权重的改变量 ΔW 是低秩的。具体而言：

$$W = W_0 + \Delta W = W_0 + BA$$

其中：

- $W_0 \in \mathbb{R}^{d\times k}$：预训练权重矩阵
- $B \in \mathbb{R}^{d\times r}$：低秩降维矩阵（$r \ll d$）
- $A \in \mathbb{R}^{r\times k}$：低秩升维矩阵
- $\Delta W = BA$：低秩权重更新量

34.2.3　参数更新策略

LoRA 采用严格的参数更新策略：

1. **冻结基础模型**：保持预训练权重 W_0 不变
2. **仅训练适配器**：只更新低秩矩阵 B 和 A 的参数
3. **秩的选择**：通常选择较小的秩（如 4, 8, 16）以保证参数效率

34.3　语料构建与数据处理

34.3.1　数据来源与获取

本项目使用的中文预训练语料来自中文书籍收录整理项目，包含丰富的经典文学作品：

```
# 数据获取命令
git clone https://github.com/shjwudp/shu.git
```

34.3.2　语料组成分析

数据集包含从先秦到近代的经典中文文学作品，主要类别包括：

表 34.2: 中文预训练语料组成

时代	代表作品	数据量	文件大小
先秦	《论语》《道德经》《孙子兵法》	15 部	约 5MB
秦汉	《史记》《春秋左传》	8 部	约 8MB
魏晋南北朝	《昭明文选》	5 部	约 4MB
隋唐	《唐代诗词》《唐代传奇》	12 部	约 6MB
宋元	《梦溪笔谈》《宋代诗词》	10 部	约 7MB
明清	《红楼梦》《水浒传》《西游记》	20 部	约 15MB
近代	《呐喊》《骆驼祥子》	8 部	约 5MB

34.3.3　数据格式规范

所有语料均以纯文本格式（.txt）存储，采用统一编码和格式规范：

- **文件编码**：UTF-8 编码确保中文兼容性
- **段落分隔**：使用换行符进行自然段落划分
- **章节标记**：使用特定标记标识章节开始和结束
- **文本清洗**：移除无关符号和格式标记

34.3.4　语料预处理流程

```
import os
import re
from typing import List

def preprocess_chinese_corpus(text_files: List[str]) -> List[str]:
    """中文语料预处理流程"""
    processed_texts = []

    for file_path in text_files:
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()

```

```
            # 移除特殊符号和格式标记
            content = re.sub(r'[^\u4e00-\u9fa5。，！？；：""''\s]', '',
                content)

            # 统一段落分隔
            content = re.sub(r'\n+', '\n', content)

            # 章节标准化处理
            content = standardize_chapters(content)

            processed_texts.append(content)

    return processed_texts

def standardize_chapters(text: str) -> str:
    """标准化章节标记"""
    # 识别并标准化章节标题
    patterns = [
        (r'第[零一二三四五六七八九十百千]+回', 'CHAPTER'),
        (r'第[0-9]+章', 'CHAPTER'),
        (r'[卷篇]之[零一二三四五六七八九十]', 'SECTION')
    ]

    for pattern, replacement in patterns:
        text = re.sub(pattern, replacement, text)

    return text
```

34.4　二次预训练实现细节

34.4.1　模型参数配置

基础模型参数

```
@dataclass
class ModelArguments:
    """模型相关参数配置"""

    model_name_or_path: Optional[str] = field(
```

```
        default=None,
        metadata={
            "help": "预训练模型路径，用于权重初始化"
        }
    )

    tokenizer_name_or_path: Optional[str] = field(
        default=None,
        metadata={"help": "分词器路径"}
    )

    model_type: Optional[str] = field(
        default=None,
        metadata={
            "help": "模型类型，如llama、bloom等"
        }
    )

    config_overrides: Optional[str] = field(
        default=None,
        metadata={
            "help": "覆盖默认配置参数"
        }
    )

    torch_dtype: Optional[str] = field(
        default=None,
        metadata={
            "help": "张量数据类型，支持auto/bfloat16/float16/float32"
        }
    )
```

数据参数配置

```
@dataclass
class DataTrainingArguments:
    """训练数据参数配置"""

    dataset_dir: Optional[str] = field(
```

```
        default=None,
        metadata={"help": "数据集目录路径"}
    )

    train_file: Optional[str] = field(
        default=None,
        metadata={"help": "训练数据文件路径"}
    )

    validation_file: Optional[str] = field(
        default=None,
        metadata={"help": "验证数据文件路径"}
    )

    max_train_samples: Optional[int] = field(
        default=None,
        metadata={"help": "最大训练样本数（调试用）"}
    )

    block_size: Optional[int] = field(
        default=None,
        metadata={"help": "输入序列最大长度"}
    )

    preprocessing_num_workers: Optional[int] = field(
        default=None,
        metadata={"help": "数据预处理工作进程数"}
    )
```

LoRA 特定参数

```
@dataclass
class LoRATrainingArguments(TrainingArguments):
    """LoRA训练参数配置"""

    trainable: Optional[str] = field(
        default="q_proj,v_proj",
        metadata={"help": "可训练的注意力层"}
    )
```

```
    lora_rank: Optional[int] = field(
        default=8,
        metadata={"help": "LoRA秩参数"}
    )

    lora_alpha: Optional[float] = field(
        default=32.0,
        metadata={"help": "LoRA缩放系数"}
    )

    lora_dropout: Optional[float] = field(
        default=0.1,
        metadata={"help": "LoRA丢弃率"}
    )

    modules_to_save: Optional[str] = field(
        default=None,
        metadata={"help": "需要保存的模块"}
    )

    load_in_kbits: Optional[int] = field(
        default=16,
        metadata={"help": "量化位数"}
    )
```

34.4.2　模型配置策略

不同场景下的模型配置

表 34.3: 不同使用场景的模型配置策略

使用场景	model_name_or_path	tokenizer_name_or_path	词表大小
基于原版 LLaMA-2 训练	原版 HF 格式 LLaMA-2	中文 LLaMA-2 分词器	55296
基于中文 LLaMA-2 继续训练	完整中文 LLaMA-2	中文 LLaMA-2 分词器	55296
基于中文 Alpaca-2 继续训练	完整中文 Alpaca-2	中文 LLaMA-2 分词器	55296

34.4.3　训练参数优化

关键超参数设置

```
# 二次预训练关键参数
lr = 2e-4                       # 学习率
lora_rank = 64                  # LoRA秩
lora_alpha = 128                # LoRA缩放系数
lora_dropout = 0.05             # 丢弃率

# 可训练模块配置
lora_trainable = "q_proj,v_proj,k_proj,o_proj,gate_proj,down_proj,up_proj
    "
modules_to_save = "embed_tokens,lm_head"  # 需要保存的模块

# 训练配置
per_device_train_batch_size = 1
gradient_accumulation_steps = 1
block_size = 512                # 序列长度
training_steps = 25000
```

注意力层作用分析

- **q_proj, k_proj, v_proj**：查询、键、值投影层，负责注意力计算
- **o_proj**：输出投影层，整合注意力结果
- **gate_proj, down_proj, up_proj**：前馈网络层，负责非线性变换

34.4.4　训练启动命令

```
torchrun --nnodes 1 --nproc_per_node 1 \
    scripts/training/run_clm_pt_with_peft.py \
    --deepspeed ${deepspeed_config_file} \
    --model_name_or_path ${pretrained_model} \
    --tokenizer_name_or_path ${chinese_tokenizer_path} \
    --dataset_dir ${dataset_dir} \
    --data_cache_dir ${data_cache} \
    --validation_split_percentage 0.001 \
    --per_device_train_batch_size ${per_device_train_batch_size} \
    --do_train \
    --seed $RANDOM \
```

```
    --fp16 \
    --max_steps ${training_steps} \
    --num_train_epochs 1 \
    --lr_scheduler_type cosine \
    --learning_rate ${lr} \
    --warmup_ratio 0.05 \
    --weight_decay 0.01 \
    --logging_strategy steps \
    --logging_steps 10 \
    --save_strategy steps \
    --save_total_limit 3 \
    --save_steps 500 \
    --gradient_accumulation_steps ${gradient_accumulation_steps} \
    --preprocessing_num_workers 8 \
    --block_size ${block_size} \
    --output_dir ${output_dir} \
    --overwrite_output_dir \
    --ddp_timeout 30000 \
    --logging_first_step True \
    --lora_rank ${lora_rank} \
    --lora_alpha ${lora_alpha} \
    --trainable ${lora_trainable} \
    --modules_to_save ${modules_to_save} \
    --lora_dropout ${lora_dropout} \
    --torch_dtype float16 \
    --resume True \
    --gradient_checkpointing \
    --ddp_find_unused_parameters False
```

34.5　指令微调实现

34.5.1　微调数据准备

数据来源

使用 Stanford Alpaca 项目提供的高质量指令数据，包含 52K 条由 GPT 生成的指令-回答对。中文版本采用 Chinese-LLaMA-Alpaca 的中文 Alpaca 数据。

提示模板设计

采用原版 Stanford Alpaca 不带 input 的模板格式。对于包含 input 字段的数据，采用拼接形式：

$$\text{prompt} = f\text{"}\{\text{instruction}\}\backslash n\{\text{input}\}\text{"}$$

34.5.2　微调参数配置

关键参数设置

```
# 指令微调参数
lr = 1e-4                       # 较低的学习率
lora_rank = 64                  # 保持相同的秩
lora_alpha = 128                # 相同的缩放系数
lora_trainable = "q_proj,v_proj,k_proj,o_proj,gate_proj,down_proj,up_proj
    "
modules_to_save = "embed_tokens,lm_head"
lora_dropout = 0.05

# 训练配置
per_device_train_batch_size = 1
per_device_eval_batch_size = 1
gradient_accumulation_steps = 8
max_seq_length = 512
training_steps = 6000
```

微调启动命令

```
torchrun --nnodes 1 --nproc_per_node 7 \
    scripts/training/run_clm_sft_with_peft.py \
    --deepspeed ${deepspeed_config_file} \
    --model_name_or_path ${pretrained_model} \
    --tokenizer_name_or_path ${chinese_tokenizer_path} \
    --dataset_dir ${dataset_dir} \
    --per_device_train_batch_size ${per_device_train_batch_size} \
    --per_device_eval_batch_size ${per_device_eval_batch_size} \
    --do_train \
    --do_eval \
    --eval_steps 1000 \
    --seed $RANDOM \
```

```
    --fp16 \
    --num_train_epochs 1 \
    --lr_scheduler_type cosine \
    --learning_rate ${lr} \
    --warmup_ratio 0.03 \
    --weight_decay 0 \
    --logging_strategy steps \
    --logging_steps 10 \
    --save_strategy steps \
    --save_total_limit 3 \
    --evaluation_strategy steps \
    --eval_steps 6000 \
    --save_steps 3000 \
    --gradient_accumulation_steps ${gradient_accumulation_steps} \
    --preprocessing_num_workers 8 \
    --max_steps ${training_steps} \
    --max_seq_length ${max_seq_length} \
    --output_dir ${output_dir} \
    --overwrite_output_dir \
    --ddp_timeout 30000 \
    --logging_first_step True \
    --lora_rank ${lora_rank} \
    --lora_alpha ${lora_alpha} \
    --trainable ${lora_trainable} \
    --lora_dropout ${lora_dropout} \
    --modules_to_save ${modules_to_save} \
    --torch_dtype float16 \
    --validation_file ${validation_file}
```

34.6　资源监控与优化

34.6.1　GPU 资源使用情况

训练过程监控

在 8×A100 GPU 环境下训练时的资源使用情况：

表 34.4: 多 GPU 训练资源监控（A100 40GB）

GPU	温度 (°C)	功耗 (W)	显存使用	利用率	进程 ID	进程类型	命令
0	53	324	20449M/40960M	95%	1114333	C	/root/miniconda3/bi
1	54	364	20749M/40960M	94%	1114334	C	/root/miniconda3/bi
2	48	326	20265M/40960M	89%	1114335	C	/root/miniconda3/bi
3	53	337	20265M/40960M	89%	1114336	C	/root/miniconda3/bi
4	52	335	20737M/40960M	92%	1114337	C	/root/miniconda3/bi
5	48	319	20449M/40960M	93%	1114338	C	/root/miniconda3/bi
6	30	52	2M/40960M	0%	-	-	-

34.6.2　存储空间分析

模型文件大小

训练完成后生成的模型文件大小分析：

表 34.5: LoRA 适配器文件大小分析

文件	描述	大小
adapter_config.json	适配器配置文件	484B
adapter_model.bin	LoRA 适配器权重	1.2GB
special_tokens_map.json	特殊标记映射	435B
tokenizer_config.json	分词器配置	844B

34.7　推理部署与应用

34.7.1　推理脚本使用

基础推理命令

```
python scripts/inference/inference_hf.py \
    --base_model correspond_output_dir \       # 基础模型路径
    --lora_model sft_output_dir2/sft_lora_model \  # LoRA适配器路径
    --tokenizer_path correspond_output_dir \  # 分词器路径
    --with_prompt \                           # 自动添加提示模板
```

推理示例

输入：”why do you need to protect environment? Please answer in Chinese!”

输出：” 为了保护环境，我们需要采取行动，因为它是我们唯一的家园，它是我们生命的源泉。”

34.7.2　部署优化策略

内存优化

- **量化推理**：使用 8bit 或 4bit 量化减少内存占用
- **适配器融合**：将 LoRA 权重合并到基础模型中提升推理速度
- **动态加载**：支持适配器的动态加载和切换

性能优化

- **批处理优化**：支持批量推理提升吞吐量
- **缓存机制**：实现注意力键值缓存加速生成
- **硬件适配**：针对不同硬件平台进行优化

34.8　技术总结与展望

34.8.1　关键技术要点

LoRA 优势总结

1. **参数效率**：仅训练少量参数，大幅降低计算需求
2. **训练稳定性**：低秩分解提供稳定的优化空间
3. **灵活性**：支持多任务适配器快速切换
4. **兼容性**：与现有预训练模型良好兼容

实践建议

- **秩的选择**：根据任务复杂度选择合适秩大小（4-64）
- **学习率设置**：采用较低学习率保证训练稳定性
- **模块选择**：优先优化注意力相关模块
- **数据质量**：高质量数据是效果的关键保证

34.8.2　未来发展方向

技术优化方向

- **自适应秩选择**：根据任务自动选择最优秩大小
- **多模态扩展**：将 LoRA 扩展到视觉、语音等多模态任务
- **动态适配**：支持运行时动态调整适配器参数

应用拓展方向

- **领域自适应**：在医疗、法律等专业领域应用
- **多语言支持**：扩展更多语言的支持能力
- **边缘部署**：优化在资源受限设备上的部署

34.8.3　实践价值

基于 LoRA 的 LLaMA2 二次预训练技术为大模型的实际应用提供了高效可行的技术路径，特别是在：

- **资源受限环境**：使得在有限计算资源下进行模型定制成为可能
- **快速迭代**：支持快速的领域适应和效果验证
- **商业化应用**：降低企业应用大模型的技术门槛和成本

该技术方案的成功实践为大语言模型的普惠化应用奠定了重要基础。

第三十五章　大语言模型 (LLMs) 评测技术详解

35.1　引言：为什么需要大模型评测？

35.1.1　传统评测基准的局限性

随着大语言模型的快速发展，传统的评测基准如 SuperGLUE、GLUE 以及中文的 CLUE 在评估大模型时表现出明显的局限性。这些基准主要针对特定自然语言理解任务设计，无法全面评估大模型在推理能力、多轮对话、创造性生成等核心能力方面的表现。

35.1.2　大模型评测的必要性

- **能力全面评估**：大模型具备多种复杂能力，需要综合性评测框架
- **技术发展导向**：为模型研发提供明确的技术改进方向
- **应用场景适配**：确保模型能力与实际应用需求相匹配
- **资源优化配置**：为计算资源分配和模型选择提供依据

35.2　大模型评测的核心维度

35.2.1　理解能力评估

理解能力是大模型的基础核心能力，需要通过多维度问题进行评估：

深度文本理解

设计需要深入理解文本语义、逻辑关系和隐含信息的问题：

- **语义理解**：对复杂句子的准确解析
- **逻辑关系**：识别文本中的因果、转折、条件等关系
- **隐含信息**：推断文本未明确表达的信息
- **上下文关联**：理解跨句子的语义连贯性

评估示例

```
问题：阅读以下文本后回答问题：
"尽管天气炎热，小明还是决定去跑步，因为他认为锻炼对身体有益。"

问题1：小明为什么去跑步？
问题2：文本中"尽管"表达了什么关系？
问题3：小明的决定可能基于什么价值观？
```

35.2.2　语言生成能力评估

语言生成能力评估需要考察文本的结构完整性、逻辑连贯性和语言质量：

生成质量指标

- **结构完整性**：文章是否有明确的开头、发展和结尾
- **逻辑连贯性**：内容是否逻辑清晰，衔接自然
- **语法正确性**：语言表达是否符合语法规范
- **风格一致性**：是否保持统一的语言风格
- **信息密度**：内容是否充实且有价值

生成任务设计

```
生成任务：以"人工智能的未来发展"为主题，写一篇800字左右的文章，要求：
1. 包含技术发展、社会影响、伦理考量三个部分
2. 观点明确，论证充分
3. 语言流畅，结构清晰
```

35.2.3　知识面广度评估

通过跨领域问题测试模型的知识覆盖范围：

知识领域分类

35.2.4　适应性能力评估

测试模型处理不同类型任务的能力：

表 35.1: 多领域知识评估体系

领域类别	评估内容	示例问题
科学技术	物理、化学、生物、计算机等	解释量子计算的基本原理
历史文化	历史事件、文化传统、艺术等	分析文艺复兴对欧洲的影响
文学艺术	文学作品、艺术理论、创作等	解读《红楼梦》的主要主题
社会经济	经济理论、社会现象、商业等	讨论通货膨胀的成因和影响
哲学伦理	哲学思想、伦理道德、逻辑等	分析功利主义的主要观点

多任务适应性

- **写作任务**：不同文体和风格的文本创作
- **翻译任务**：多语言间的准确翻译
- **编程任务**：代码生成、调试和解释
- **分析任务**：数据分析和推理判断
- **创作任务**：诗歌、故事等创造性内容

35.2.5　长文本处理能力评估

长文本理解

通过长篇文章的阅读理解测试模型的信息处理能力：

- **要点提取**：从长文本中提取核心信息
- **摘要生成**：对长文本进行准确概括
- **逻辑分析**：理解长文本的论证结构
- **细节记忆**：保持对文中细节的准确记忆

长文本生成

评估模型生成长篇连贯文本的能力：

```
任务要求：创作一个完整的故事，包含：
- 明确的故事背景和人物设定
- 完整的情节发展（开端、发展、高潮、结局）
- 人物性格的逐步展现
```

```
- 逻辑合理的情节转折
- 生动具体的细节描写
```

35.2.6　多样性表达能力评估

测试模型生成多样化内容的能力：

创造性思维

- **多角度分析**：对同一问题提供不同视角的解答
- **替代方案**：为问题提供多种解决方案
- **创新观点**：提出新颖独特的见解
- **风格变化**：适应不同的表达风格和语气

评估方法

```
问题：如何减少城市交通拥堵？请提供5种不同的解决方案，每种方案需要：
1. 具体的实施方法
2. 预期的效果分析
3. 可能的挑战和应对措施
```

35.2.7　情感智能评估

情感分析能力

测试模型理解文本情感色彩的能力：

- **情感识别**：准确识别文本中的情感倾向
- **情感强度**：判断情感表达的强烈程度
- **情感变化**：跟踪文本中情感的发展变化
- **情感原因**：分析情感产生的内在原因

情感表达能力

评估模型生成带有情感色彩文本的能力：

```
任务：以第一人称描述以下场景，要求体现指定的情感：
场景：雨夜独自在家
情感要求：先表现孤独感，逐渐转向宁静和思考
表达要求：使用具体细节和比喻增强情感表达
```

35.2.8　逻辑推理能力评估

推理类型测试

表 35.2: 逻辑推理能力评估体系

推理类型	评估重点	典型问题
演绎推理	从一般到特殊的推理过程	所有哺乳动物都会呼吸。鲸鱼是哺乳动物。那么鲸鱼会呼吸吗？
归纳推理	从特殊到一般的推理过程	观察多个实例得出一般规律
类比推理	基于相似性的推理	A 与 B 的关系类似于 C 与什么的关系？
因果推理	因果关系分析和推断	分析事件之间的因果关系
数学推理	数学问题和逻辑谜题	解决数学证明和逻辑谜题

推理复杂度分级

- **基础推理**：简单的逻辑判断和推理
- **多层推理**：需要多个推理步骤的复杂问题
- **抽象推理**：涉及抽象概念和关系的推理
- **批判性思维**：对信息和论证的批判性分析

35.2.9　问题解决能力评估

实际问题解决

测试模型解决实际问题的能力：

- **数学问题**：数学计算、证明和问题解决
- **编程问题**：算法设计、代码实现和调试
- **规划问题**：任务规划和资源分配
- **决策问题**：在复杂情况下的决策分析

问题解决流程评估

```
1 数学问题：一个水池有进水管和出水管，进水管单独注满水池需要6小时，出水管单独排空水池需要8小时。如果同时打开进水管和出水管，问需要多少小时才能注满水池？
2
```

```
要求分步骤解答：
1. 定义变量和已知条件
2. 建立数学模型
3. 求解过程
4. 结果验证
```

35.2.10　道德伦理判断评估

伦理困境分析

测试模型处理道德伦理问题的能力：

- **价值判断**：基于伦理原则的价值判断
- **困境分析**：分析伦理困境中的各种因素
- **决策推理**：提供符合伦理的决策建议
- **多视角考量**：考虑不同利益相关者的视角

评估案例

```
伦理问题：在什么情况下撒谎是可以接受的？
评估要求：
1. 分析撒谎的道德含义
2. 讨论可能接受撒谎的具体情境
3. 提供伦理判断的原则和标准
4. 考虑不同文化背景下的差异
```

35.2.11　对话交互能力评估

多轮对话质量

评估模型在对话中的表现：

- **上下文保持**：准确记忆和理解对话历史
- **话题连贯**：保持话题的自然发展和转换
- **意图理解**：准确理解用户的真实意图
- **响应适当**：提供相关且有价值的回应

对话评估指标

```
对话场景：旅游规划咨询
评估维度：
- 信息准确性：提供的旅游信息是否准确
```

```
- 个性化程度：是否根据用户需求提供个性化建议
- 交互自然度：对话流程是否自然流畅
- 问题解决：是否有效解决用户的实际问题
```

35.3　大模型 Honest 原则的实现机制

35.3.1　Honest 原则的技术内涵

Helpful、Honest、Harmless（3H）原则是大模型发展的核心准则，其中 Honest 原则要求模型在知识表达上保持诚实，不虚构不存在的信息。

35.3.2　训练数据策略

知识问答样本构造

通过精心设计训练样本，培养模型的诚实表达能力：

- **已知知识回答**：对于训练数据中存在明确答案的问题，要求准确回答
- **未知知识回避**：对于超出训练数据范围的问题，鼓励承认不知道
- **不确定性表达**：对于边界知识，教导模型表达适当的不确定性

样本构造示例

```
正样本：
问题："水的沸点是多少度？"
回答："在标准大气压下，水的沸点是100摄氏度。"

负样本：
问题："请描述外星人的生理结构"
回答："根据目前的科学知识，我们还没有确凿的证据证明外星生命的存在，因此无法提供准确的描述。"
```

35.3.3　阅读理解训练优化

真实性约束机制

在阅读理解任务中强化诚实原则：

- **证据要求**：回答必须基于文本中的明确证据
- **禁止虚构**：严格禁止基于推理的虚构内容
- **不确定性标记**：对推断内容进行明确标记
- **置信度表达**：提供回答的置信度水平

训练技术细节

1. **数据标注**：对训练数据中的事实性内容进行精确标注
2. **损失函数设计**：在损失函数中加入真实性约束项
3. **强化学习**：使用 RLHF 技术强化诚实行为
4. **对抗训练**：通过对抗样本增强模型的抗虚构能力

35.3.4　技术实现策略

知识边界识别

```
def knowledge_boundary_detection(question, knowledge_base):
    """知识边界检测机制"""
    # 计算问题与知识库的相似度
    similarity_scores = calculate_similarity(question, knowledge_base)

    # 设置置信度阈值
    confidence_threshold = 0.7

    if max(similarity_scores) < confidence_threshold:
        return "unknown", max(similarity_scores)
    else:
        return "known", max(similarity_scores)

def generate_honest_response(question, knowledge_status, confidence):
    """生成诚实回答"""
    if knowledge_status == "unknown":
        return "我目前没有足够的信息来准确回答这个问题。"
    elif confidence < 0.9:
        return f"基于现有信息，我认为{answer}，但这个答案的置信度只有{
            confidence:.2f}。"
    else:
        return confident_answer
```

真实性评估框架

建立多层次的真实性评估机制：

- **内部一致性检查**：验证生成内容的内在逻辑一致性
- **外部知识验证**：与知识库进行事实核对
- **置信度校准**：确保置信度评估的准确性

- **错误纠正机制**：建立错误承认和纠正的机制

35.4　大模型评测方法体系

35.4.1　人工评估方法

人工评估的优势

- **主观质量**：能够评估文本质量、流畅度等主观指标
- **上下文理解**：考虑对话上下文和语义细微差别
- **创造性评估**：评价内容的创造性和新颖性
- **实用价值**：判断回答的实际有用性

代表性工作

表 35.3: 人工评估代表性工作对比

项目	评估重点	特色方法
LIMA	指令跟随和质量评估	有限数据下的模型能力评估
Phoenix	多语言对话评估	跨语言一致性评估
Vicuna	聊天质量评估	基于多轮对话的全面评估
BELLE	中文模型评估	针对中文特性的评估体系

35.4.2　自动评估方法

GPT-4 反馈评估

利用先进大模型进行自动评估：

- **质量评分**：GPT-4 对生成内容进行质量评分
- **一致性检查**：评估内容的一致性和连贯性
- **有用性评估**：判断回答的实际价值
- **安全性检查**：检测有害或不适当内容

评估提示设计

```
你是一个专业的AI助手评估员。请对以下模型回答进行评估：

问题：{question}
模型回答：{response}
```

```
请从以下维度进行评分（1-10分）：
1. 相关性：回答是否直接针对问题
2. 准确性：信息是否准确无误
3. 完整性：是否全面覆盖问题的各个方面
4. 条理性：表达是否清晰有条理
5. 实用性：回答是否具有实际价值

请提供总体评分和具体改进建议。
```

35.4.3　指标化评估方法

传统指标应用

- **BLEU-4**：评估机器翻译和文本生成质量
- **ROUGE 分数**：评估文本摘要质量
- **精确率/召回率**：评估分类和问答任务
- **困惑度**：评估语言模型质量

非自然指令评估

针对大模型特有的能力进行评估：

```
非自然指令示例：
1. 请用莎士比亚的风格写一首关于人工智能的十四行诗
2. 将以下内容翻译成编程代码：[自然语言描述]
3. 用5岁孩子能理解的方式解释相对论
```

35.4.4　Chatbot Arena 评估平台

传统基准的局限性

- **主观性挑战**：聊天质量评估具有很强的主观性
- **数据污染风险**：训练数据可能包含测试集内容
- **领域覆盖不足**：现有基准无法覆盖所有对话场景
- **过拟合风险**：模型可能针对特定基准过度优化

两两对比评估机制

Chatbot Arena 采用创新的评估方法：

1. **实时对话**：用户与两个匿名模型进行实时对话

2. **人工评分**：用户基于对话体验进行主观评分
3. **ELO 评级**：采用国际象棋的 ELO 评级系统
4. **大规模参与**：吸引大量用户参与确保统计显著性

ELO 评级系统

ELO 评级系统的数学原理：

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

$$R'_A = R_A + K \times (S_A - E_A)$$

其中：

- E_A：A 选手的预期胜率
- R_A, R_B：A、B 选手的当前评级
- S_A：实际比赛结果（胜 1、平 0.5、负 0）
- K：评级调整系数

35.5　大模型评测工具生态

35.5.1　OpenAI Evals 评估框架

核心设计理念

OpenAI Evals 通过模板化的提示词实现自动化评估：

评估模板设计

```
# eval模板示例
def evaluate_qa_completion(question, model_response, reference_answer):
    """QA任务评估模板"""

    prompt = f"""
    请评估以下问答对的质量：

    问题：{question}
    模型回答：{model_response}
    参考答案：{reference_answer}

    评估维度：
    1. 答案准确性（0-10分）
    2. 信息完整性（0-10分）
```

```
    3. 表达清晰度（0-10分）

    请提供详细评估理由和分数。
    """

    return get_llm_evaluation(prompt)

# 批量评估执行
def run_batch_evaluation(dataset, eval_function):
    """批量运行评估"""
    results = []
    for item in dataset:
        score = eval_function(item['question'],
                              item['response'],
                              item['reference'])
        results.append(score)
    return results
```

模板化评估优势

- **可重复性**：确保评估过程的一致性和可重复性
- **可扩展性**：容易扩展到新的任务和领域
- **自动化**：支持大规模自动化评估
- **透明度**：评估标准和过程完全透明

35.5.2 PandaLM 自动化评估模型

模型架构设计

PandaLM 训练专门的评估模型进行自动化评分：

三分制评分体系

- **0 分**：回答质量差，存在明显问题
- **1 分**：回答基本合格，但有改进空间
- **2 分**：回答优秀，质量很高

模型训练策略

```
class PandaLMEvaluator:
    """PandaLM评估器实现"""
```

```
    def __init__(self, model_path):
        self.model = load_evaluation_model(model_path)

    def compare_responses(self, question, response_a, response_b):
        """比较两个回答的质量"""

        comparison_prompt = f"""
        问题: {question}
        回答A: {response_a}
        回答B: {response_b}

        请比较两个回答的质量, 选择更好的一个:
        - 如果A明显更好, 输出A
        - 如果B明显更好, 输出B
        - 如果质量相当, 输出Tie
        """

        return self.model.generate(comparison_prompt)

    def absolute_scoring(self, question, response):
        """绝对评分"""
        scoring_prompt = f"""
        问题: {question}
        回答: {response}

        请按照以下标准评分:
        0分: 回答存在严重问题
        1分: 回答基本合格但有缺陷
        2分: 回答优秀

        评分:
        """

        return self.model.generate(scoring_prompt)
```

35.5.3　综合评测平台建设

评测维度整合

建立全面的评测体系需要整合多种方法：

表 35.4: 综合评测平台功能模块

模块	功能描述	技术实现
自动化测试	批量执行标准测试用例	OpenAI Evals 模板引擎
人工评估	众包质量评估	两两比较界面
实时监控	生产环境性能监控	日志分析和指标计算
安全检测	内容安全性和合规性	敏感词过滤和模式识别
性能基准	推理速度和资源消耗	性能 profiling 工具

评测流水线设计

```python
class EvaluationPipeline:
    """端到端评测流水线"""

    def __init__(self, model, eval_tools):
        self.model = model
        self.eval_tools = eval_tools

    def run_comprehensive_eval(self, test_suites):
        """运行全面评估"""
        results = {}

        # 自动化指标评估
        results['auto_metrics'] = self.run_automatic_evaluation(
            test_suites)

        # 人工评估采样
        results['human_eval'] = self.sample_human_evaluation(test_suites)

        # 安全性和合规性检查
        results['safety_check'] = self.run_safety_checks(test_suites)

        # 性能基准测试
        results['performance'] = self.performance_benchmark(test_suites)
```

```
23
24          return self.aggregate_results(results)
```

35.6　评测实践与挑战

35.6.1　评测数据构建

高质量测试集构建原则

- **多样性**：覆盖不同的领域、风格和难度级别
- **真实性**：基于真实使用场景设计测试用例
- **平衡性**：在不同能力维度间保持平衡
- **可扩展性**：支持后续的更新和扩展

测试数据来源

1. **学术基准**：Adapt 和扩展现有学术数据集
2. **实际应用**：从真实应用场景中收集用例
3. **众包构建**：通过众包平台收集多样化测试用例
4. **模型生成**：使用大模型生成补充测试数据

35.6.2　评测中的挑战与应对

主要技术挑战

- **主观性处理**：如何量化和标准化主观质量评估
- **评估一致性**：确保不同评估者和时间点的一致性
- **数据污染**：避免测试数据在训练中的泄露
- **评估成本**：平衡评估质量与成本效率

应对策略

35.7　未来发展方向

35.7.1　评测技术趋势

多模态评测扩展

- **图文理解**：评估模型理解图像和文本关联的能力
- **多模态生成**：测试图文结合的内容生成能力
- **跨模态推理**：评估不同模态信息间的推理能力

表 35.5: 评测挑战应对策略

挑战	应对策略	具体措施
主观性处理	建立详细评估标准和培训体系	制定评估指南，评估员培训
评估一致性	采用统计方法确保评分一致性	Cohen's Kappa 计算，定期校准
数据污染	严格的数据隔离和清洗流程	训练测试集分离，重复检测
评估成本	分层抽样和自动化结合	关键用例人工评估，大量用例自动化

动态适应性评测

- **增量学习评估**：测试模型持续学习新知识的能力
- **领域适应评估**：评估模型在新领域的适应能力
- **个性化评估**：测试模型的个性化交互能力

35.7.2　评测生态建设

开放评测生态

- **标准制定**：建立行业统一的评测标准
- **开源工具**：开发开放源码的评测工具集
- **社区参与**：鼓励社区参与评测数据和方法建设
- **国际合作**：推动国际评测标准的协调统一

伦理合规评测

- **偏见检测**：系统检测模型输出中的各种偏见
- **公平性评估**：评估模型对不同群体的公平性
- **透明度要求**：推动模型决策过程的透明度
- **问责机制**：建立模型错误的问责和改进机制

35.8　总结

大语言模型的评测是一个复杂而关键的技术领域，需要综合运用人工评估、自动评估和指标化评估等多种方法。随着大模型技术的不断发展，评测体系也需要持续演进，以准确反映模型的真实能力水平。

未来的评测工作应当更加注重实际应用价值，建立更加全面、公平、高效的评测体系，为大模型技术的健康发展提供有力支撑。同时，需要特别关注伦理安全方面的评测，确保大模

型技术的发展符合人类价值观和社会利益。

第三十六章　大语言模型强化学习技术详解

36.1　引言：大模型与强化学习

36.1.1　强化学习在大模型中的作用

随着大语言模型 (LLMs) 的快速发展，如何让模型更好地与人类价值观对齐、生成更符合期望的输出成为关键挑战。强化学习，特别是基于人类反馈的强化学习 (RLHF)，已成为解决这一挑战的核心技术路径。

36.1.2　技术演进背景

- **预训练局限性**：大规模预训练使模型获得丰富知识，但无法保证输出符合特定期望
- **对齐需求**：需要将模型能力引导至对人类有帮助、诚实、无害的方向
- **效率挑战**：传统 RLHF 存在计算成本高、流程复杂等问题
- **技术革新**：新方法不断涌现以解决 RLHF 的实践挑战

36.2　强化学习基础

36.2.1　强化学习基本概念

强化学习 (Reinforcement Learning) 是一种通过智能体与环境的交互来学习最优策略的机器学习方法。其核心思想是：智能体通过执行动作影响环境，环境反馈奖励信号，智能体根据奖励调整策略以最大化长期累积奖励。

36.2.2　强化学习关键要素

36.2.3　强化学习在大模型中的应用特点

- **动作空间**：生成的文本序列，空间极其巨大
- **奖励稀疏**：仅在生成完整回复后获得奖励信号
- **信用分配**：需要将最终奖励分配到生成过程中的每个 token
- **探索挑战**：在巨大的动作空间中有效探索困难

表 36.1: 强化学习核心要素

要素	描述
智能体 (Agent)	学习并做出决策的主体
环境 (Environment)	智能体交互的外部世界
状态 (State)	环境在特定时刻的描述
动作 (Action)	智能体可以执行的操作
奖励 (Reward)	环境对智能体动作的反馈
策略 (Policy)	状态到动作的映射函数
价值函数 (Value Function)	评估状态或动作的长期价值

36.3　基于人类反馈的强化学习 (RLHF)

36.3.1　RLHF 技术框架

RLHF(Reinforcement Learning from Human Feedback) 是让大语言模型与人类价值观对齐的核心技术，包含三个主要阶段：

三阶段流程

1. **监督微调 (SFT)**：使用高质量人工标注数据对预训练模型进行微调
2. **奖励模型训练 (RM)**：训练一个奖励模型来预测人类偏好
3. **强化学习优化 (PPO)**：使用 PPO 算法基于奖励模型反馈优化策略

数学形式化

RLHF 的目标是优化策略 π_θ 以最大化期望奖励：

$$\max_\theta \mathbb{E}_{x\sim\mathcal{D},y\sim\pi_\theta(\cdot|x)}[r_\phi(x,y)] - \beta\mathbb{D}_{\mathrm{KL}}[\pi_\theta(y|x)\|\pi_{\mathrm{ref}}(y|x)]$$

其中：

- r_ϕ：奖励模型参数化为 ϕ
- π_{ref}：参考策略（通常为 SFT 后的模型）
- β：KL 惩罚系数，防止策略偏离参考策略太远

36.3.2　RLHF 的实施细节

阶段一：监督微调 (SFT)

```
class SFTTrainer:
    """监督微调训练器"""
```

```
    def __init__(self, base_model, train_dataset):
        self.model = base_model
        self.dataset = train_dataset

    def fine_tune(self, epochs=3, learning_rate=1e-5):
        """执行监督微调"""
        optimizer = AdamW(self.model.parameters(), lr=learning_rate)

        for epoch in range(epochs):
            for batch in self.dataset:
                # 前向传播
                outputs = self.model(
                    input_ids=batch['input_ids'],
                    attention_mask=batch['attention_mask'],
                    labels=batch['labels']
                )

                # 计算损失
                loss = outputs.loss

                # 反向传播
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
```

阶段二：奖励模型训练 (RM)

奖励模型学习预测人类对模型生成内容的偏好：

```
class RewardModelTrainer:
    """奖励模型训练器"""

    def __init__(self, model, preference_dataset):
        self.model = model
        self.dataset = preference_dataset  # 包含(y_win, y_lose)对

    def train_reward_model(self):
        """训练奖励模型"""
        for batch in self.dataset:
```

```
            # 计算获胜回复的奖励
            rewards_win = self.model(batch['win_input_ids'])
            # 计算失败回复的奖励
            rewards_lose = self.model(batch['lose_input_ids'])

            # 使用Bradley-Terry模型损失
            loss = -torch.log(torch.sigmoid(rewards_win - rewards_lose)).
               mean()

            # 优化步骤...
```

阶段三：PPO 优化

```
class PPOTrainer:
    """PPO训练器"""

    def __init__(self, policy_model, value_model, reward_model, ref_model
       ):
        self.policy_model = policy_model  # 被优化的策略
        self.value_model = value_model    # 价值函数估计
        self.reward_model = reward_model  # 奖励预测
        self.ref_model = ref_model        # 参考策略(SFT模型)

    def ppo_update(self, prompts):
        """执行PPO更新"""
        # 1. 使用当前策略生成回复
        with torch.no_grad():
            responses = self.policy_model.generate(prompts)

        # 2. 计算奖励(包括KL惩罚)
        rewards = self.compute_rewards(prompts, responses)

        # 3. 计算优势估计
        advantages = self.compute_advantages(rewards)

        # 4. PPO目标函数优化
        for _ in range(self.ppo_epochs):
            # 计算策略比率
            ratio = self.compute_probability_ratio(prompts, responses)
```

```
26
27              # PPO裁剪目标
28              surr1 = ratio * advantages
29              surr2 = torch.clamp(ratio, 1 - self.eps, 1 + self.eps) *
                    advantages
30              policy_loss = -torch.min(surr1, surr2).mean()
31
32              # 价值函数损失
33              value_loss = self.compute_value_loss(rewards)
34
35              # 总损失
36              total_loss = policy_loss + value_loss
37
38              # 优化步骤...
```

36.4　RLHF 实践挑战与解决方案

36.4.1　奖励模型与基础模型一致性问题

一致性要求分析

在实践中，奖励模型是否需要与基础模型保持一致存在不同观点：

表 36.2: 奖励模型一致性选择策略

方案	优势	局限性
相同架构	参数共享，训练稳定	可能限制奖励模型表达能力
不同架构	更灵活的特征提取	需要处理架构差异带来的挑战
同系列模型	平衡表达能力和兼容性	选择范围受限

技术实现考量

- **Tokenizer 一致性**：如 Colossal-AI 的 COATI 要求相同 tokenizer 以确保兼容性
- **表示空间对齐**：不同架构的模型需要在表示空间上进行对齐
- **训练稳定性**：相同架构通常训练更稳定，收敛更快
- **实践建议**：从同系列模型开始，逐步尝试不同架构

36.4.2　RLHF 三大核心挑战

挑战一：人工偏好数据成本高昂

- **问题描述**：高质量人类标注数据收集成本高、周期长
- **影响范围**：限制模型迭代速度和应用规模
- **根本原因**：需要大量专家级人工评估确保质量

挑战二：三阶段训练流程复杂

- **流程复杂度**：SFT → RM → PPO 多阶段训练链路长
- **迭代速度**：完整流程需要数天到数周时间
- **调试困难**：问题定位和调优跨多个阶段

挑战三：计算资源需求巨大

- **模型数量**：PPO 同时需要 4 个模型（2 训练 +2 推理）
- **显存占用**：需要同时加载多个模型副本
- **计算开销**：PPO 需要多次前向和反向传播

36.5　AI 专家替代方案

36.5.1　RLAIF：AI 反馈的强化学习

核心思想

RLAIF(Reinforcement Learning from AI Feedback) 使用 AI 模型替代人类进行反馈生成，核心思路是通过 AI 模型监督其他 AI 模型。

技术流程

1. **自我批判生成**：从初始模型采样生成，然后生成自我批评和修正
2. **修正反馈**：根据修正后的反应微调原始模型
3. **AI 偏好数据集**：使用 AI 模型评估生成样本，构建偏好数据集
4. **偏好模型训练**：基于 AI 偏好数据训练奖励模型
5. **RL 训练**：使用 AI 奖励模型进行强化学习优化

实现细节

```
class RLAIFTrainer:
    """RLAIF训练器"""

```

```
    def generate_ai_feedback(self, prompts, responses):
        """生成AI反馈"""
        # 使用强大的AI模型（如GPT-4）进行评估
        feedback_prompts = self.construct_feedback_prompts(prompts,
            responses)
        ai_feedbacks = self.ai_evaluator.generate(feedback_prompts)

        # 解析反馈为偏好对
        preference_pairs = self.parse_feedback_to_preferences(
            ai_feedbacks)
        return preference_pairs

    def train_with_ai_feedback(self):
        """使用AI反馈进行训练"""
        # 1. 生成初始响应
        responses = self.model.generate(self.prompts)

        # 2. 获取AI反馈
        preference_pairs = self.generate_ai_feedback(self.prompts,
            responses)

        # 3. 训练奖励模型
        self.reward_model.train(preference_pairs)

        # 4. RL优化
        self.rl_optimizer.optimize_with_reward_model(self.reward_model)
```

36.5.2　RRHF：基于排名的偏好优化

方法创新

RRHF(Rank Response from Human Feedback) 摒弃复杂的强化学习流程，直接通过排名损失实现对齐：

- **去 RL 化**：不需要 PPO 等复杂 RL 算法
- **多模型集成**：可以利用 ChatGPT、GPT-4 等多种模型生成回复
- **双重功能**：训练好的模型同时具备生成和奖励评估能力

排名损失设计

RRHF 使用排名损失使模型输出与人类偏好对齐：

$$\mathcal{L}_{\text{RRHF}} = \mathbb{E}[\max(0, -(\log \pi_\theta(y_w|x) - \log \pi_\theta(y_l|x)) + \lambda)]$$

其中 y_w 是获胜回复，y_l 是失败回复，λ 是边际参数。

实现代码

```python
class RRHFTrainer:
    """RRHF训练器"""

    def __init__(self, model, ranking_criterion):
        self.model = model
        self.criterion = ranking_criterion

    def rank_loss(self, winning_responses, losing_responses):
        """计算排名损失"""
        # 计算获胜回复的log概率
        win_log_probs = self.compute_log_probs(winning_responses)
        # 计算失败回复的log概率
        lose_log_probs = self.compute_log_probs(losing_responses)

        # 排名损失：鼓励获胜回复有更高概率
        loss = self.criterion(win_log_probs, lose_log_probs)
        return loss

    def compute_log_probs(self, responses):
        """计算序列的log概率"""
        log_probs = []
        for response in responses:
            with torch.no_grad():
                outputs = self.model(response, return_dict=True)
                log_prob = outputs.logits.log_softmax(dim=-1)
            log_probs.append(log_prob)
        return torch.stack(log_probs)
```

36.6　微调数据优化方案

36.6.1　LIMA：少即是多的对齐假设

核心理论

LIMA(Less Is More for Alignment) 基于浅¨层对齐假说：¨ 模型的知识和能力主要在预训练中获得，对齐主要是学习与用户交互的样式。

理论推论

- **知识预存**：模型能力在预训练中已基本确定
- **对齐简化**：对齐主要是学习交互风格和格式
- **数据效率**：少量高质量样本即可实现有效对齐
- **实践验证**：LIMA 论文使用 1,000 个精心策划的样本达到接近 SOTA 效果

实施策略

```
class LIMATrainer:
    """LIMA风格的高效训练"""

    def select_high_quality_samples(self, raw_dataset, quality_criteria):
        """选择高质量样本"""
        high_quality_samples = []

        for sample in raw_dataset:
            # 基于多维度质量评估
            quality_score = self.assess_sample_quality(sample,
                quality_criteria)

            if quality_score > self.quality_threshold:
                high_quality_samples.append(sample)

        return high_quality_samples[:self.max_samples]  # 严格控制样本数
            量

    def efficient_fine_tune(self, high_quality_samples):
        """高效微调"""
        # 使用较小的学习率和更多训练轮次
        optimizer = AdamW(self.model.parameters(), lr=1e-6)

```

```
        for epoch in range(10):  # 更多轮次学习有限样本
            for batch in self.create_batches(high_quality_samples):
                loss = self.model(batch).loss
                # 精细化的优化过程...
```

36.6.2　0.5% 数据假设：数据效率优化

核心思想

该研究从数据角度探索如何降低 LLM 训练成本，通过识别数据集中最有价值的核心样本来提高数据效率。

关键技术

1. **价值评估**：开发样本价值评估指标识别高质量样本
2. **多样性保持**：确保选中样本覆盖足够的数据分布
3. **课程学习**：按难度和重要性组织训练顺序
4. **主动学习**：动态调整样本选择策略

样本选择算法

```
class DataEfficientSelector:
    """高效数据选择器"""

    def __init__(self, selection_strategy='importance_sampling'):
        self.strategy = selection_strategy

    def select_core_samples(self, dataset, target_ratio=0.005):
        """选择核心样本（0.5%）"""
        n_samples = len(dataset)
        n_target = int(n_samples * target_ratio)

        if self.strategy == 'importance_sampling':
            # 基于重要性的采样
            importance_scores = self.compute_importance_scores(dataset)
            selected_indices = self.sample_by_importance(
                importance_scores, n_target)

        elif self.strategy == 'diversity_maximization':
            # 多样性最大化的选择
```

```
            selected_indices = self.maximize_diversity_selection(dataset,
                n_target)

        return dataset[selected_indices]

    def compute_importance_scores(self, dataset):
        """计算样本重要性分数"""
        # 基于梯度信息、损失变化、模型不确定性等
        scores = []
        for sample in dataset:
            score = self.estimate_sample_importance(sample)
            scores.append(score)
        return scores
```

36.7　训练过程改造方案

36.7.1　RAFT：奖励排序微调

方法概述

RAFT(Reward rAnked FineTuning) 通过结合奖励排序和监督微调来简化训练流程，避免复杂的 PPO 优化。

核心步骤

1. **样本生成：**从当前策略生成多个候选回复
2. **奖励排序：**使用奖励模型对候选回复进行排序
3. **策略更新：**使用排名最高的回复进行监督学习
4. **迭代优化：**重复生成-排序-学习循环

算法优势

- **简化流程：**用监督学习替代复杂 RL 算法
- **稳定训练：**避免 RL 训练的不稳定性
- **资源高效：**减少同时运行的模型数量
- **易于调试：**训练过程更透明可控

实现框架

```
class RAFTTrainer:
```

```
    """RAFT训练器"""

    def __init__(self, policy_model, reward_model, num_candidates=4):
        self.policy = policy_model
        self.reward_model = reward_model
        self.k = num_candidates  # 每个提示生成的候选数

    def raft_iteration(self, prompts):
        """单次RAFT迭代"""
        all_candidates = []
        all_rewards = []

        for prompt in prompts:
            # 1. 生成多个候选
            candidates = self.generate_candidates(prompt, n=self.k)

            # 2. 奖励模型评分
            rewards = [self.reward_model.score(candidate) for candidate
                in candidates]

            all_candidates.extend(candidates)
            all_rewards.extend(rewards)

        # 3. 选择最佳候选
        best_indices = self.select_best_candidates(all_rewards)
        best_candidates = [all_candidates[i] for i in best_indices]

        # 4. 监督学习更新
        loss = self.supervised_update(prompts, best_candidates)
        return loss

    def select_best_candidates(self, rewards, selection_strategy='top_k')
        :
        """选择最佳候选策略"""
        if selection_strategy == 'top_k':
            # 选择奖励最高的k个
            return torch.topk(torch.tensor(rewards), k=len(rewards)//2).
                indices
        elif selection_strategy == 'softmax_sampling':
            # 基于softmax概率采样
```

```
            probs = torch.softmax(torch.tensor(rewards), dim=0)
            return torch.multinomial(probs, len(rewards)//2)
```

36.7.2　DPO：直接偏好优化

理论突破

DPO(Direct Preference Optimization) 通过数学推导将 RLHF 问题转化为直接优化问题，无需训练奖励模型。

关键洞察

DPO 发现可以通过巧妙的数学变换，将包含奖励模型的 RL 目标函数转换为仅涉及策略概率的损失函数：

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$

优势分析

- **去奖励模型**：不需要单独训练奖励模型
- **训练稳定**：避免 RL 训练的不稳定性
- **计算高效**：只需要监督式前向-反向传播
- **理论优雅**：有严格的数学推导保证

DPO 实现

```
class DPOTrainer:
    """DPO训练器"""

    def __init__(self, model, reference_model, beta=0.1):
        self.model = model
        self.ref_model = reference_model
        self.beta = beta  # KL惩罚系数

    def dpo_loss(self, prompts, winning_responses, losing_responses):
        """计算DPO损失"""
        losses = []

        for prompt, y_w, y_l in zip(prompts, winning_responses,
            losing_responses):
```

```
            # 计算当前策略的概率
            logpi_w = self.compute_log_prob(self.model, prompt, y_w)
            logpi_l = self.compute_log_prob(self.model, prompt, y_l)

            # 计算参考策略的概率
            logpi_ref_w = self.compute_log_prob(self.ref_model, prompt,
                y_w)
            logpi_ref_l = self.compute_log_prob(self.ref_model, prompt,
                y_l)

            # DPO损失项
            log_ratio_w = logpi_w - logpi_ref_w
            log_ratio_l = logpi_l - logpi_ref_l

            loss = -torch.log(torch.sigmoid(self.beta * (log_ratio_w -
                log_ratio_l)))
            losses.append(loss)

        return torch.stack(losses).mean()

    def compute_log_prob(self, model, prompt, response):
        """计算序列的log概率"""
        with torch.no_grad():
            outputs = model(torch.cat([prompt, response]))
            log_probs = outputs.logits.log_softmax(dim=-1)
        return log_probs.gather(dim=-1, index=response.unsqueeze(-1)).
            squeeze().sum()
```

36.8　技术对比与实践建议

36.8.1　方法对比分析

36.8.2　实践选择指南

根据资源条件选择

- **充足资源**：传统 RLHF 流程，效果最可靠
- **中等资源**：RRHF 或 RAFT，平衡效果和效率
- **有限资源**：DPO 或 LIMA，最大化数据效率

表 36.3: 大模型强化学习方法对比

方法	类别	核心优势	适用场景	资源需求
RLHF	经典方法	效果可靠，经验丰富	资源充足的研究	极高
RLAIF	AI 替代	减少人工成本，可扩展	大规模应用	高
RRHF	排序优化	简化流程，训练稳定	快速迭代需求	中等
LIMA	数据优化	数据高效，原理清晰	数据稀缺场景	低
RAFT	流程改造	平衡效果与复杂度	平衡性要求	中等
DPO	理论突破	无需奖励模型，理论优雅	理论研究	低

根据应用场景选择

- **研究探索**：DPO（理论创新）或 RLAIF（扩展性）
- **生产部署**：RRHF 或 RAFT（稳定性优先）
- **快速原型**：LIMA（快速验证想法）

技术选型决策树

```
if 有充足的人工标注资源 and 计算资源丰富:
    选择传统RLHF
elif 需要快速迭代 and 稳定性重要:
    if 有高质量奖励模型:
        选择RAFT
    else:
        选择RRHF
elif 理论研究为主 and 追求理论优雅:
    选择DPO
elif 数据稀缺 but 有精心策划的小数据集:
    选择LIMA
elif 需要大规模扩展 and 可接受AI反馈:
    选择RLAIF
```

36.9　未来发展方向

36.9.1　技术趋势展望

理论创新方向

- **更优目标函数**：开发比 DPO 更优雅的优化目标
- **多目标优化**：同时优化 helpful、honest、harmless 等多个目标
- **课程强化学习**：设计渐进式学习课程
- **元强化学习**：学习如何更高效地学习人类偏好

工程优化方向

- **分布式训练**：更高效的分布式 RLHF 训练框架
- **量化推理**：低精度推理加速奖励模型评估
- **自适应优化**：根据训练动态调整超参数
- **多模态扩展**：扩展到视觉、语音等多模态任务

应用拓展方向

- **个性化对齐**：学习个体用户的特定偏好
- **领域自适应**：快速适应新领域的需求
- **持续学习**：在不遗忘旧知识的前提下学习新偏好
- **安全对齐**：增强模型的安全性和可靠性

36.10　总结

大语言模型的强化学习技术正经历快速演进，从传统的 RLHF 到各种创新方法，都在努力解决实践中的三大挑战：数据成本、训练复杂度和计算资源。每种方法都有其适用场景和优势劣势，实践中需要根据具体需求和约束进行选择。

第三十七章　大语言模型强化学习 PPO 技术详解

37.1　引言：PPO 在 RLHF 中的核心地位

37.1.1　PPO 技术背景

近端策略优化（Proximal Policy Optimization，PPO）作为强化学习领域的重要算法，在大语言模型的人类反馈强化学习（RLHF）流程中扮演着关键角色。PPO 通过平衡探索与利用、确保训练稳定性，成为连接奖励模型与策略优化的桥梁。

37.1.2　PPO 在 RLHF 中的价值

- **策略优化核心**：将奖励模型信号转化为策略改进方向
- **训练稳定性**：通过裁剪机制避免策略更新幅度过大
- **样本效率**：支持多次 epoch 的参数更新，提高数据利用效率
- **收敛保证**：理论保证策略改进的单调性

37.2　RLHF 中 PPO 的核心步骤

37.2.1　三阶段流程架构

PPO 在 RLHF 中的实现遵循采样-反馈-学习的迭代优化流程：

算法框架

```
class PPOTrainer:
    """PPO训练器核心框架"""

    def __init__(self, policy_model, reward_model):
        self.policy_model = policy_model
        self.reward_model = reward_model
```

```
    def rlhf_training_loop(self, num_iterations=20000):
        """RLHF训练主循环"""
        for iteration in range(num_iterations):
            # 1. 采样阶段：生成回答
            prompts = self.sample_prompts()
            responses = self.respond(self.policy_model, prompts)

            # 2. 反馈阶段：计算奖励
            rewards = self.reward_func(self.reward_model, prompts,
                responses)

            # 3. 学习阶段：策略优化
            for epoch in range(4):  # 典型PPO设置：4个epoch
                self.policy_model = self.train_epoch(
                    self.policy_model, prompts, responses, rewards
                )
```

37.2.2　步骤一：采样阶段

采样过程本质

采样是模型根据输入提示（prompt）生成回答（response）的过程，实质是模型自行生成训练数据：

$$\text{采样过程：}\quad \text{prompt} \xrightarrow{\text{模型}} \text{response}$$

采样数据示例

表 37.1: PPO 采样过程数据示例

提示（Prompt）	回答（Response）
请告诉我三种常见的动物。	猫，狗，鹦鹉。
如何评价电影《爱乐之城》?	音乐的经典令人赞叹不已，结局却让人感到五味杂陈。
詹姆斯和库里谁更伟大?	他们都很伟大，我无法比较。

采样技术实现

```
def respond(policy_model, prompts, max_length=512, temperature=0.7):
    """基于策略模型生成回答"""
    responses = []

    for prompt in prompts:
        # 设置生成参数
        generation_config = {
            'max_length': max_length,
            'temperature': temperature,
            'do_sample': True,
            'top_p': 0.9,
            'pad_token_id': policy_model.config.pad_token_id
        }

        # 生成回答
        input_ids = tokenizer.encode(prompt, return_tensors='pt')
        with torch.no_grad():
            output = policy_model.generate(
                input_ids,
                **generation_config
            )

        response = tokenizer.decode(output[0], skip_special_tokens=True)
        responses.append(response)

    return responses

def sample_prompts(batch_size=8, dataset='instruction_dataset'):
    """从数据集中采样提示"""
    # 从预准备的指令数据集中随机采样
    prompts = random.sample(instruction_dataset, batch_size)
    return prompts
```

37.2.3　步骤二：反馈阶段

奖励计算机制

反馈阶段通过奖励模型对采样生成的回答进行质量评估：

反馈过程：　$(\text{prompt}, \text{response}) \xrightarrow{\text{奖励模型}} \text{reward}$

奖励函数设计

```python
def reward_func(reward_model, prompts, responses,
                base_reward_weight=1.0, kl_penalty_weight=0.1):
    """计算综合奖励函数"""
    rewards = []

    for prompt, response in zip(prompts, responses):
        # 基础奖励：奖励模型预测
        base_reward = reward_model.predict(prompt, response)

        # KL惩罚：防止策略偏离参考策略太远
        kl_penalty = compute_kl_penalty(prompt, response)

        # 综合奖励
        total_reward = (base_reward_weight * base_reward -
                        kl_penalty_weight * kl_penalty)
        rewards.append(total_reward)

    return torch.tensor(rewards)

def compute_kl_penalty(prompt, response, ref_model):
    """计算KL散度惩罚项"""
    # 当前策略的概率
    current_probs = policy_model.get_action_probs(prompt, response)

    # 参考策略的概率（通常为SFT模型）
    ref_probs = ref_model.get_action_probs(prompt, response)

    # KL散度：D_KL(current || ref)
    kl_divergence = torch.sum(
        current_probs * (torch.log(current_probs) - torch.log(ref_probs))
    )

    return kl_divergence
```

37.2.4　步骤三：学习阶段

PPO 优化目标

PPO 通过优化裁剪的目标函数来更新策略参数：

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

其中：

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$：策略比率
- $\hat{A}_t$：优势函数估计
- ϵ：裁剪参数（通常设为 0.1-0.2）

多 epoch 优化

```python
def train_epoch(policy_model, prompts, responses, rewards, num_epochs=4):
    """多epoch策略优化"""
    optimizer = torch.optim.Adam(policy_model.parameters(), lr=1e-6)

    for epoch in range(num_epochs):
        total_loss = 0

        for prompt, response, reward in zip(prompts, responses, rewards):
            # 计算旧策略的概率（用于比率计算）
            with torch.no_grad():
                old_probs = policy_model.get_action_probs(prompt,
                    response)

            # 前向传播计算当前策略概率
            current_probs = policy_model.get_action_probs(prompt,
                response)

            # 计算策略比率
            ratio = current_probs / old_probs

            # PPO裁剪损失
            surr1 = ratio * reward
            surr2 = torch.clamp(ratio, 1 - 0.2, 1 + 0.2) * reward
            policy_loss = -torch.min(surr1, surr2).mean()

            # 价值函数损失（如果使用评论家）
```

```
            value_loss = compute_value_loss(policy_model, prompt,
                response, reward)

            # 熵正则化
            entropy_bonus = compute_entropy_bonus(current_probs)

            # 总损失
            loss = policy_loss + 0.5 * value_loss - 0.01 * entropy_bonus

            # 反向传播
            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(policy_model.parameters(),
                1.0)
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(prompts)
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}")

    return policy_model
```

37.3　RLHF 教学类比理解

37.3.1　师生互动比喻

角色对应关系

教学过程模拟

1. **提出问题：**教师（人类）提出有趣且有挑战性的问题（提示）
2. **学生尝试：**学生（模型）基于当前知识尝试回答问题（生成回答）
3. **教师评分：**教师根据评分标准（奖励模型）对回答进行评分
4. **反馈学习：**学生根据分数反馈调整学习策略（策略优化）
5. **持续改进：**通过多轮互动，学生不断改进回答质量

表 37.2: RLHF 师生角色对应关系

组件	教师角色	学生角色
人类标注者	出题老师	-
提示（Prompt）	课堂问题	-
策略模型（PPO）	-	学生
回答生成	-	学生尝试回答
奖励模型	评分标准	-
奖励信号	分数反馈	-
策略更新	-	根据反馈改进学习方法

37.3.2　教育心理学启示

渐进式学习

RLHF 模拟了人类学习的渐进特性：

- **小步前进**：每次策略更新幅度有限（PPO 裁剪）
- **错误容忍**：允许尝试和犯错，从反馈中学习
- **个性化调整**：根据具体表现调整学习策略
- **长期优化**：通过多轮迭代持续改进

激励设计

```python
# 教育中的奖励设计类比
def educational_reward_design(student_answer, correct_answer, criteria):
    """教育场景奖励设计"""
    rewards = {}

    # 基础正确性奖励
    if student_answer == correct_answer:
        rewards['accuracy'] = 1.0
    else:
        rewards['accuracy'] = 0.0

    # 创造性奖励（超出标准答案的亮点）
    creativity_bonus = assess_creativity(student_answer, correct_answer)
    rewards['creativity'] = creativity_bonus

    # 表达清晰度奖励
    clarity_score = assess_clarity(student_answer)
```

```
    rewards['clarity'] = clarity_score

    # 综合奖励（加权求和）
    total_reward = (0.6 * rewards['accuracy'] +
                    0.2 * rewards['creativity'] +
                    0.2 * rewards['clarity'])

    return total_reward
```

37.4　PPO 采样策略与技术实现

37.4.1　采样过程详解

采样本质

PPO 中的采样过程是模型基于当前策略生成行为轨迹的过程，在大语言模型语境下特指根据提示生成文本回答：

$$\text{采样：} \quad \pi_\theta(a|s) \rightarrow \text{动作序列} \rightarrow \text{文本序列}$$

采样参数配置

```
class PPOSamplingConfig:
    """PPO采样参数配置"""

    def __init__(self):
        self.max_length = 512      # 最大生成长度
        self.temperature = 0.7     # 温度参数（控制随机性）
        self.top_p = 0.9           # 核采样参数
        self.top_k = 50            # Top-k采样
        self.do_sample = True      # 是否使用采样（非贪婪）
        self.num_beams = 1         # Beam search束宽
        self.repetition_penalty = 1.2  # 重复惩罚

    def get_generation_config(self):
        """获取生成配置"""
        return {
            'max_length': self.max_length,
            'temperature': self.temperature,
            'top_p': self.top_p,
```

```
            'top_k': self.top_k,
            'do_sample': self.do_sample,
            'num_beams': self.num_beams,
            'repetition_penalty': self.repetition_penalty,
            'pad_token_id': self.pad_token_id,
            'eos_token_id': self.eos_token_id
        }
```

37.4.2　演员-评论家架构

双模型策略

PPO 采用演员-评论家（Actor-Critic）架构，模拟人类决策过程中的两种思维模式：

表 37.3: 演员-评论家架构类比

组件	演员（Actor）	评论家（Critic）
角色定位	决策执行者	价值评估者
人类类比	直觉思维	理性分析
输入	当前状态（提示）	当前状态（提示）
输出	动作概率分布	状态价值估计
训练目标	最大化期望回报	最小化价值误差
模型基础	SFT 微调后的模型	新训练的价值网络

架构实现

```
class ActorCriticModel(nn.Module):
    """演员-评论家模型"""

    def __init__(self, base_model, hidden_size):
        super().__init__()
        self.base_model = base_model  # 共享的骨干网络
        self.hidden_size = hidden_size

        # 演员头: 输出动作概率
        self.actor_head = nn.Linear(hidden_size, base_model.config.
            vocab_size)

        # 评论家头: 输出状态价值
```

```
        self.critic_head = nn.Linear(hidden_size, 1)

    def forward(self, input_ids, attention_mask=None):
        # 共享特征提取
        outputs = self.base_model(
            input_ids,
            attention_mask=attention_mask,
            output_hidden_states=True
        )
        last_hidden_state = outputs.hidden_states[-1]

        # 演员输出：每个位置的动作概率
        actor_logits = self.actor_head(last_hidden_state)
        action_probs = F.softmax(actor_logits, dim=-1)

        # 评论家输出：状态价值估计
        state_values = self.critic_head(last_hidden_state[:, -1, :])

        return action_probs, state_values.squeeze(-1)

    def get_action_probs(self, prompt, response):
        """获取特定动作的概率"""
        input_text = prompt + response
        input_ids = tokenizer.encode(input_text, return_tensors='pt')

        with torch.no_grad():
            action_probs, _ = self.forward(input_ids)

        # 获取响应部分对应的概率
        prompt_len = len(tokenizer.encode(prompt))
        response_probs = action_probs[0, prompt_len-1:-1]  # 忽略最后一个
            token

        return response_probs
```

37.4.3　收益评估机制

收益定义

在 PPO 中，”收益”指从当前时间步开始，模型能够获得的累积奖励的期望值：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

其中 γ 为折扣因子。

奖励组成

PPO 中的奖励包含多个组成部分：

```
def compute_comprehensive_reward(prompt, response, reward_model,
                                 ref_model, kl_weight=0.1, entropy_weight
                                    =0.01):
    """计算综合奖励"""
    # 1. 基础奖励: 奖励模型预测
    base_reward = reward_model.predict(prompt, response)

    # 2. KL惩罚: 防止策略偏离
    kl_penalty = compute_kl_divergence(prompt, response, ref_model)

    # 3. 熵奖励: 鼓励探索（可选）
    entropy_bonus = compute_entropy(prompt, response)

    # 综合奖励
    total_reward = (base_reward -
                    kl_weight * kl_penalty +
                    entropy_weight * entropy_bonus)

    return total_reward

def compute_kl_divergence(prompt, response, ref_model):
    """计算KL散度惩罚"""
    # 当前策略的概率
    current_probs = policy_model.get_action_probs(prompt, response)

    # 参考策略的概率（SFT模型）
    ref_probs = ref_model.get_action_probs(prompt, response)

    # 避免log(0)的情况
    epsilon = 1e-8
    current_probs = torch.clamp(current_probs, epsilon, 1.0)
    ref_probs = torch.clamp(ref_probs, epsilon, 1.0)
```

```
32
33      # KL散度:   p_current * log(p_current / p_ref)
34      kl_div = torch.sum(current_probs * (torch.log(current_probs) - torch.
            log(ref_probs)))
35
36      return kl_div
37
38  def compute_entropy(prompt, response):
39      """计算策略熵（鼓励探索）"""
40      action_probs = policy_model.get_action_probs(prompt, response)
41
42      # 熵: - p * log(p)
43      entropy = -torch.sum(action_probs * torch.log(action_probs + 1e-8))
44
45      return entropy
```

优势函数估计

```
1   def compute_advantages(rewards, values, gamma=0.99, lam=0.95):
2       """使用GAE（广义优势估计）计算优势函数"""
3       advantages = []
4       gae = 0
5
6       # 反向计算GAE
7       for t in reversed(range(len(rewards))):
8           if t == len(rewards) - 1:
9               next_value = 0  # 终止状态的价值为0
10          else:
11              next_value = values[t+1]
12
13          delta = rewards[t] + gamma * next_value - values[t]
14          gae = delta + gamma * lam * gae
15          advantages.insert(0, gae)  # 在开头插入
16
17      advantages = torch.tensor(advantages)
18
19      # 标准化优势函数
20      advantages = (advantages - advantages.mean()) / (advantages.std() + 1
            e-8)
```

```
21
22      return advantages
```

37.5　PPO 在 RLHF 中的实践考量

37.5.1　超参数调优

关键超参数

表 37.4: PPO 关键超参数设置建议

参数	作用	典型值	调优建议
学习率（LR）	控制参数更新步长	1e-6 到 1e-5	从小开始，逐步增加
裁剪 epsilon	限制策略更新幅度	0.1 到 0.3	影响训练稳定性
KL 权重	控制策略偏离程度	0.01 到 0.2	平衡创新与保守
熵权重	鼓励探索	0.01 到 0.1	防止策略过早收敛
GAE 参数	优势估计平滑	0.9 到 0.95	影响信用分配
折扣因子	远期奖励重要性	0.99 到 0.999	控制长远规划

37.5.2　训练稳定性保障

梯度裁剪

```
1  def ppo_update_with_clipping(policy_model, optimizer, observations,
2                               actions, old_probs, advantages, epsilon=0.2):
3      """带梯度裁剪的PPO更新"""
4      # 计算新策略概率
5      new_probs = policy_model.get_action_probs(observations, actions)
6
7      # 策略比率
8      ratios = new_probs / old_probs
9
10     # 裁剪目标函数
11     surr1 = ratios * advantages
12     surr2 = torch.clamp(ratios, 1 - epsilon, 1 + epsilon) * advantages
13     policy_loss = -torch.min(surr1, surr2).mean()
14
```

```
    # 梯度裁剪
    optimizer.zero_grad()
    policy_loss.backward()
    torch.nn.utils.clip_grad_norm_(policy_model.parameters(), max_norm
        =1.0)
    optimizer.step()

    return policy_loss.item()
```

早期停止机制

```
def early_stopping(kl_divergences, threshold=0.05, patience=3):
    """基于KL散度的早期停止"""
    if len(kl_divergences) < patience:
        return False

    # 检查最近patience次的KL散度
    recent_kls = kl_divergences[-patience:]
    avg_kl = sum(recent_kls) / patience

    # 如果平均KL散度超过阈值，触发早期停止
    if avg_kl > threshold:
        print(f"Early stopping triggered: average KL {avg_kl:.4f} >
            threshold {threshold}")
        return True

    return False
```

37.6　总结与展望

37.6.1　技术总结

PPO 作为 RLHF 流程中的核心优化算法，通过其独特的裁剪机制和演员-评论家架构，在大语言模型对齐中发挥了关键作用：

- **稳定性优势**：裁剪机制确保训练过程稳定收敛
- **样本效率**：支持经验回放和多 epoch 优化
- **灵活性**：可适应不同奖励函数设计
- **可扩展性**：支持大规模分布式训练

37.6.2　未来优化方向

算法改进

- **自适应裁剪**：根据训练进度动态调整裁剪范围
- **多目标优化**：同时优化多个竞争性目标
- **元学习**：学习更高效的优化策略本身
- **课程学习**：设计渐进难度的训练课程

工程优化

- **分布式训练**：更高效的并行化策略
- **内存优化**：减少激活值存储开销
- **混合精度**：FP16/FP8 训练加速
- **硬件适配**：针对特定硬件优化

应用拓展

- **多模态扩展**：适应文本、图像、语音等多模态输入
- **个性化对齐**：学习个体用户的特定偏好
- **领域自适应**：快速适应新领域需求
- **持续学习**：在不遗忘的前提下学习新知识

PPO 技术在大语言模型对齐中的应用前景广阔，随着算法不断优化和计算资源持续增长，将在构建更安全、更有用的人工智能系统中发挥越来越重要的作用。

第三十八章　强化学习在自然语言处理中的应用技术详解

38.1　引言：强化学习与自然语言处理的融合

38.1.1　技术融合背景

强化学习（Reinforcement Learning）作为机器学习的重要分支，与自然语言处理（Natural Language Processing）的结合为语言模型训练提供了新的范式。特别是在大语言模型（LLMs）时代，RL 在指令跟随、对话生成、文本优化等任务中展现出独特价值。

38.1.2　RL 在 NLP 中的独特优势

- **序列决策能力**：自然语言生成本质上是序列决策过程
- **长期收益优化**：考虑生成文本的整体质量而非局部最优
- **人类反馈集成**：通过奖励函数融入人类偏好和价值观
- **探索利用平衡**：在创新性和准确性间取得平衡

38.2　强化学习基础理论

38.2.1　强化学习基本框架

核心定义

强化学习是一种时序决策学习框架，智能体通过与环境交互来学习最优策略。其数学表示为：

$$a_t = \pi(o_t)$$

$$r_t = r(o_t, a_t)$$

其中：

- π：策略函数，从观测到动作的映射

- o_t：时间步 t 的观测
- a_t：时间步 t 的动作
- r_t：时间步 t 的即时奖励

交互流程

智能体与环境的交互形成闭环：

1. 智能体接收环境状态观测 o_t
2. 基于策略 π 选择动作 a_t
3. 环境转换到新状态，产生奖励 r_t
4. 智能体根据奖励调整策略

38.2.2　状态与观测系统

状态（States）定义

状态是对世界环境的完整描述，包含决策所需的所有信息。在完全可观测环境中，状态 s_t 完全决定了环境的未来演变。

观测（Observations）定义

观测是对状态的部分描述，可能缺失某些信息。观测与状态的关系分为：

表 38.1: 状态与观测关系分类

类型	数学关系	特点	应用场景
完全可观测	$O = S$	观测包含完整状态信息	棋盘游戏、完全信息博弈
部分可观测	$O \subset S$	观测缺失部分状态信息	对话系统、现实世界交互

在 NLP 中的具体体现

在自然语言处理任务中：

- **状态**：完整的对话历史、用户意图、上下文信息
- **观测**：当前输入的文本、部分对话历史、可用上下文
- **实践挑战**：NLP 任务通常属于部分可观测环境

38.2.3　动作空间分类与特性

离散动作空间

当智能体只能从有限动作集合中选择时，称为离散动作空间：

$$\mathcal{A} = \{a_1, a_2, \ldots, a_n\}, \quad n < \infty$$

特点：

- **有限性：**动作数量有限且可枚举
- **分类性：**每个动作代表一个类别选择
- **应用场景：**文本生成（词汇选择）、游戏动作（移动方向）、对话动作（回复类型）

连续动作空间

当动作是实数向量时，称为连续动作空间：

$$\mathcal{A} \subseteq \mathbb{R}^n$$

特点：

- **无限性：**动作空间不可数
- **连续性：**动作参数可连续变化
- **应用场景：**机器人控制、参数优化、连续决策

在 NLP 中的动作空间设计

表 38.2: NLP 任务中的动作空间设计

任务类型	动作空间设计	策略网络实现
文本生成	词汇表大小的离散空间	Softmax 输出层
文本改写	编辑操作的离散空间	分类器 + 生成器混合
参数调优	超参数的连续空间	回归输出层
对话管理	对话动作的离散空间	意图分类器

38.2.4　策略类型与实现

确定性策略

确定性策略将状态映射到确定的动作：

$$a_t = \mu(s_t)$$

其中 $\mu : \mathcal{S} \to \mathcal{A}$ 是确定性映射函数。

特点：

- **确定性：**相同状态总是产生相同动作

- **适用性**：主要用于连续动作空间
- **优势**：训练稳定，收敛性好
- **劣势**：探索能力有限

随机性策略

随机性策略输出动作的概率分布：

$$a_t \sim \pi(\cdot|s_t)$$

其中 $\pi(a|s)$ 是在状态 s 下选择动作 a 的概率。

特点：

- **随机性**：相同状态可能产生不同动作
- **适用性**：主要用于离散动作空间
- **优势**：探索能力强，避免局部最优
- **劣势**：训练可能不稳定

策略网络实现

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class PolicyNetwork(nn.Module):
    """策略网络实现"""

    def __init__(self, state_dim, action_dim, hidden_dim=256,
        is_continuous=False):
        super().__init__()
        self.is_continuous = is_continuous
        self.hidden_dim = hidden_dim

        # 共享特征提取层
        self.feature_extractor = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )

```

```
        if is_continuous:
            # 连续动作空间：输出均值和标准差
            self.mu_layer = nn.Linear(hidden_dim, action_dim)
            self.log_std_layer = nn.Linear(hidden_dim, action_dim)
        else:
            # 离散动作空间：输出动作概率分布
            self.policy_head = nn.Linear(hidden_dim, action_dim)

    def forward(self, state):
        features = self.feature_extractor(state)

        if self.is_continuous:
            # 连续动作：高斯策略
            mu = torch.tanh(self.mu_layer(features))
            log_std = self.log_std_layer(features)
            std = torch.exp(log_std)
            return torch.distributions.Normal(mu, std)
        else:
            # 离散动作：分类策略
            logits = self.policy_head(features)
            return torch.distributions.Categorical(logits=logits)

    def get_action(self, state, deterministic=False):
        """根据策略选择动作"""
        dist = self.forward(state)

        if deterministic:
            if self.is_continuous:
                action = dist.mean
            else:
                action = torch.argmax(dist.probs, dim=-1)
        else:
            action = dist.sample()

        # 计算动作的对数概率（用于策略梯度）
        log_prob = dist.log_prob(action)

        return action, log_prob
```

38.2.5　轨迹与状态转移

轨迹定义

轨迹是状态和动作的序列，记录了智能体与环境的完整交互历史：

$$\tau = (s_0, a_0, s_1, a_1, s_2, a_2, \dots)$$

状态转移动力学

环境的状态转移由状态转移函数描述：

$$s_{t+1} \sim P(\cdot|s_t, a_t)$$

其中 $P(s'|s,a)$ 表示在状态 s 执行动作 a 后转移到状态 s' 的概率。

初始状态分布

轨迹的初始状态从初始状态分布中采样：

$$s_0 \sim \rho(\cdot)$$

轨迹概率计算

给定策略 π，T 步轨迹的概率为：

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

38.2.6　奖励函数设计

奖励函数定义

奖励函数评估智能体动作的质量，可分为两种形式：

$$\begin{aligned} \text{状态-动作奖励：} &\quad r_t \sim R(s_t, a_t) \\ \text{状态-动作-下一状态奖励：} &\quad r_t \sim R(s_t, a_t, s_{t+1}) \end{aligned}$$

累积回报

智能体的目标是最大化整个轨迹的累积折扣回报：

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

其中 $\gamma \in [0, 1]$ 是折扣因子，平衡即时奖励和未来奖励的重要性。

在 NLP 中的奖励设计

表 38.3: NLP 任务中的奖励函数设计

任务类型	奖励组件	设计考虑	权重
文本生成	流畅性、相关性、创造性	人类偏好、任务目标	可调
对话系统	相关性、连贯性、信息量	用户体验、任务完成度	动态
文本摘要	信息覆盖、简洁性、忠实度	源文本保持、摘要质量	平衡
机器翻译	准确性、流畅性、忠实度	双语对齐、文化适应	固定

38.2.7　强化学习问题形式化

核心优化问题

强化学习的核心问题是找到最优策略 π^*，最大化期望累积回报：

$$\pi^* = \arg\max_{\pi} J(\pi)$$

其中 $J(\pi)$ 是策略 π 的期望回报：

$$J(\pi) = \int_{\tau} P(\tau|\pi) R(\tau) = \mathbb{E}_{\tau \sim \pi}[R(\tau)]$$

值函数概念

为评估策略性能，定义状态值函数和动作值函数：

$$\begin{aligned} &\text{状态值函数：} \quad V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s] \\ &\text{动作值函数：} \quad Q^{\pi}(s, a) = \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] \end{aligned}$$

38.3　强化学习发展路径：从 Value-based 到 PPO

38.3.1　Value-based 方法

基本思想

Value-based 方法通过估计状态或状态-动作对的值函数来间接优化策略。其核心是学习最优值函数，然后导出最优策略。

最优值函数定义

$$\text{最优状态值函数：}\quad V^*(s) = \max_{\pi} \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s]$$
$$\text{最优动作值函数：}\quad Q^*(s,a) = \max_{\pi} \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$$

最优策略推导

已知最优动作值函数 Q^* 时，最优策略为：

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_{a'} Q^*(s,a') \\ 0, & \text{otherwise} \end{cases}$$

值函数关系

状态值函数与动作值函数存在重要关系：

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)]$$
$$V^*(s) = \max_a Q^*(s,a)$$

38.3.2　贝尔曼方程

基本思想

贝尔曼方程描述了值函数的递归关系：当前状态的值等于即时奖励加上折扣后的下一状态值的期望。

贝尔曼期望方程

对于任意策略 π，其值函数满足：

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi, s'\sim P}[r(s,a) + \gamma V^{\pi}(s')]$$
$$Q^{\pi}(s,a) = \mathbb{E}_{s'\sim P}[r(s,a) + \gamma \mathbb{E}_{a'\sim\pi}[Q^{\pi}(s',a')]]$$

贝尔曼最优方程

最优值函数满足贝尔曼最优方程：

$$V^{*}(s) = \max_{a} \mathbb{E}_{s'\sim P}[r(s,a) + \gamma V^{*}(s')]$$
$$Q^{*}(s,a) = \mathbb{E}_{s'\sim P}[r(s,a) + \gamma \max_{a'} Q^{*}(s',a')]$$

在 NLP 中的意义

在自然语言处理中，贝尔曼方程允许我们将长文本生成的复杂问题分解为逐词生成的子问题，通过值函数估计每个决策步骤的长期影响。

38.3.3　优势函数

基本概念

优势函数衡量在特定状态下，某个动作相对于平均水平的优势程度：

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

直观解释

- $A^{\pi}(s,a) > 0$：动作 a 优于平均水平
- $A^{\pi}(s,a) = 0$：动作 a 处于平均水平
- $A^{\pi}(s,a) < 0$：动作 a 劣于平均水平

数学性质

$$\mathbb{E}_{a\sim\pi}[A^{\pi}(s,a)] = 0$$
$$\max_{a} A^{\pi}(s,a) \geq 0$$
$$\min_{a} A^{\pi}(s,a) \leq 0$$

在策略梯度中的应用

优势函数在策略梯度方法中起到关键作用，策略梯度定理表明：

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^\pi, a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) A^\pi(s,a)]$$

其中 d^π 是在策略 π 下的状态访问分布。

38.3.4　从传统 RL 到 PPO 的发展路径

技术演进脉络

表 38.4: 强化学习算法发展路径

算法类型	代表算法	核心思想	优势	局限
Value-based	Q-learning, DQN	学习最优值函数	理论完备，收敛性好	不适合连续动作空间
Policy-based	REINFORCE, 策略梯度	直接优化策略函数	适合连续动作，随机策略	高方差，收敛慢
Actor-Critic	A2C, A3C, DDPG	值函数 + 策略函数	平衡偏差方差，更稳定	实现复杂，超参敏感
信任域方法	TRPO, PPO	约束策略更新幅度	训练稳定，性能可靠	计算成本较高

PPO 的核心创新

近端策略优化（PPO）通过裁剪机制约束策略更新幅度，在保持 TRPO 稳定性的同时大幅降低计算复杂度：

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

其中 $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ 是策略比率，ϵ 是裁剪参数。

38.4　RL 在 NLP 中的具体应用

38.4.1　文本生成任务

序列生成建模

将文本生成建模为序列决策过程：

- **状态**：已生成的部分文本
- **动作**：选择下一个词或子词
- **奖励**：生成文本的整体质量评估

奖励设计策略

```
class TextGenerationReward:
    """文本生成奖励函数设计"""

    def __init__(self, metric_weights=None):
        self.metric_weights = metric_weights or {
            'fluency': 0.3,
            'relevance': 0.4,
            'diversity': 0.2,
            'length': 0.1
        }

    def compute_reward(self, prompt, generated_text, reference_text=None):
        """计算综合奖励"""
        rewards = {}

        # 流畅性奖励（基于语言模型困惑度）
        rewards['fluency'] = self.fluency_reward(generated_text)

        # 相关性奖励（与提示的相关程度）
        rewards['relevance'] = self.relevance_reward(prompt,
            generated_text)

        # 多样性奖励（避免重复和模板化）
        rewards['diversity'] = self.diversity_reward(generated_text)

        # 长度奖励（鼓励适当长度）
        rewards['length'] = self.length_reward(generated_text)

        # 加权综合奖励
        total_reward = sum(weight * rewards[metric]
                           for metric, weight in self.metric_weights.items
                               ())

```

```
        return total_reward

    def fluency_reward(self, text):
        """计算流畅性奖励"""
        # 使用预训练语言模型计算困惑度
        perplexity = self.lm_model.perplexity(text)
        # 困惑度越低，流畅性越好
        return 1.0 / (1.0 + math.log(perplexity))

    def relevance_reward(self, prompt, generated_text):
        """计算相关性奖励"""
        # 使用相似度模型或编码器
        prompt_embedding = self.encoder.encode(prompt)
        text_embedding = self.encoder.encode(generated_text)
        similarity = cosine_similarity(prompt_embedding, text_embedding)
        return similarity
```

38.4.2　对话系统优化

对话作为马尔可夫决策过程

将多轮对话建模为 MDP：

- **状态**：对话历史、用户当前话语、系统状态
- **动作**：系统回复内容或对话动作
- **奖励**：用户满意度、任务完成度、对话质量

深度强化学习对话系统

```
class DialoguePolicyNetwork(nn.Module):
    """对话策略网络"""

    def __init__(self, vocab_size, hidden_size, num_actions):
        super().__init__()
        self.hidden_size = hidden_size

        # 对话状态编码器
        self.state_encoder = nn.LSTM(vocab_size, hidden_size, batch_first
            =True)

        # 策略网络
```

```
        self.policy_net = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, num_actions)
        )

        # 价值网络（Critic）
        self.value_net = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, dialogue_history):
        # 编码对话历史
        _, (hidden, _) = self.state_encoder(dialogue_history)
        state_encoding = hidden[-1]  # 取最后隐藏状态

        # 策略输出
        action_logits = self.policy_net(state_encoding)
        action_probs = F.softmax(action_logits, dim=-1)

        # 价值输出
        state_value = self.value_net(state_encoding)

        return torch.distributions.Categorical(action_probs), state_value
```

38.4.3　文本风格迁移

风格迁移的 RL 建模

将文本风格迁移视为强化学习问题：

- **状态**：原始文本及其特征表示
- **动作**：文本编辑操作（替换、插入、删除）
- **奖励**：风格强度、内容保持度、流畅性

约束优化框架

$$\begin{aligned} \max_{\pi} \quad & \mathbb{E}[\text{风格奖励}] \\ \text{s.t.} \quad & \mathbb{E}[\text{内容相似度}] \geq \delta \\ & \mathbb{E}[\text{流畅性}] \geq \epsilon \end{aligned}$$

38.5　技术挑战与未来方向

38.5.1　当前技术挑战

奖励设计复杂性

- **多目标平衡**：需要同时优化多个竞争性目标
- **奖励稀疏性**：在长文本生成中奖励信号稀疏
- **人类偏好建模**：准确量化人类主观偏好困难

训练稳定性问题

- **高方差**：策略梯度方法方差较大
- **收敛困难**：非凸优化问题，易陷入局部最优
- **超参敏感**：对学习率、折扣因子等超参数敏感

计算效率挑战

- **样本效率低**：需要大量交互数据
- **训练时间长**：特别是对于大语言模型
- **推理延迟**：RL 策略可能增加推理时间

38.5.2　未来研究方向

算法改进方向

- **高效探索策略**：改进探索机制提高样本效率
- **元强化学习**：学习更快适应新任务的能力
- **分层强化学习**：在不同时间尺度上学习策略
- **多智能体 RL**：处理多轮对话和协作任务

应用拓展方向

- **多模态 RL**：结合文本、图像、语音的多模态学习

- **安全对齐**：确保 RL 优化过程符合安全约束
- **个性化学习**：根据用户特性自适应调整策略
- **可解释 RL**：提高决策过程的透明度和可解释性

工程优化方向

- **分布式训练**：大规模并行化加速训练
- **离线 RL**：利用现有数据减少交互成本
- **模型压缩**：降低推理计算需求
- **硬件加速**：专用硬件优化 RL 计算

38.6　总结

强化学习为自然语言处理提供了强大的序列决策框架，特别是在大语言模型时代展现出独特价值。从基础的 Value-based 方法到先进的 PPO 算法，RL 技术不断发展，为文本生成、对话系统、风格迁移等 NLP 任务提供了有效的解决方案。

尽管面临奖励设计、训练稳定性、计算效率等挑战，但随着算法改进、计算资源增长和应用经验积累，强化学习在自然语言处理中的应用前景十分广阔。未来的研究将集中在提高样本效率、增强安全性、扩展应用范围等方向，推动 RL 与 NLP 的深度融合。

第三十九章　大语言模型训练数据集构建技术

39.1　引言：训练数据在大模型中的核心地位

39.1.1　数据驱动的大模型发展

在大语言模型（LLMs）的技术栈中，训练数据质量直接决定了模型性能的上限。从预训练到有监督微调，再到强化学习对齐，每个阶段都对数据有着不同的要求和特性。

39.1.2　数据层级体系

- **预训练数据**：构建模型的基础知识和语言能力
- **有监督微调数据**：教会模型遵循指令和特定任务格式
- **奖励模型数据**：训练模型理解人类偏好和质量标准
- **强化学习数据**：优化模型生成策略和价值对齐

39.2　各阶段训练数据格式规范

39.2.1　有监督微调（SFT）数据格式

基本数据格式

SFT 阶段采用一问一答的指令-回复对格式，旨在教会模型如何理解和遵循人类指令。

具体数据示例

```
# SFT数据格式示例
sft_dataset = [
    {
        "instruction": "写一首关于春天的诗",
        "input": "",
```

```
        "output": "春风拂面花香溢，\n柳絮飞舞鸟声啼。\n万物复苏生机勃，\n春意盎然满人间。"
    },
    {
        "instruction": "解释机器学习的基本概念",
        "input": "",
        "output": "机器学习是人工智能的一个分支，让计算机通过数据自动学习和改进，而不需要显式编程。"
    },
    {
        "instruction": "将以下英文翻译成中文",
        "input": "The quick brown fox jumps over the lazy dog",
        "output": "快速的棕色狐狸跳过了懒惰的狗"
    }
]
```

数据质量要求

表 39.1: SFT 数据质量维度

质量维度	要求标准	检查方法	重要性权重
指令清晰度	指令明确无歧义	人工审核	高
回复准确性	内容正确无误	领域专家验证	高
格式规范性	符合任务格式要求	自动化检查	中
多样性	覆盖不同主题和风格	聚类分析	中
语言质量	语法正确、表达流畅	语言模型评估	高

39.2.2　奖励模型（RM）数据格式

偏好排序格式

RM 训练需要同一提示下多个回复的质量排序数据，采用三元组格式：（问题，好回答，差回答）。

数据示例

```
# RM数据格式示例
rm_dataset = [
```

```
    {
        "prompt": "如何学习深度学习？",
        "chosen": "学习深度学习需要掌握数学基础，如线性代数和概率论，然后学习神经网络原理，最后通过实践项目巩固知识。建议从PyTorch或TensorFlow开始。",
        "rejected": "深度学习就是多看视频多练习，没什么难的。随便学学就会了。"
    },
    {
        "prompt": "Python中如何读取CSV文件？",
        "chosen": "可以使用pandas库的read_csv函数：import pandas as pd; df = pd.read_csv('file.csv')",
        "rejected": "用open函数打开文件然后一行行读，自己处理逗号分隔就行。"
    }
]
```

排序质量评估

- **一致性**：不同标注者对同一对回答的排序应一致
- **区分度**：好回答与差回答应有明显质量差距
- **覆盖面**：覆盖不同类型的提问和回复风格
- **平衡性**：正负样本比例适当，避免偏差

39.2.3　强化学习（PPO）数据格式

数据需求特点

PPO 阶段理论上不需要新增标注数据，主要利用 SFT 阶段的提示数据，通过策略优化实现模型对齐。

数据使用策略

防偏离机制

```
def ppo_training_with_ptx(prompts, sft_model, reward_model, ptx_weight=0.1):
    """带PTX损失的PPO训练"""

    # 从SFT阶段获取提示数据
    sft_prompts = load_sft_prompts()
```

表 39.2: PPO 阶段数据使用策略

数据来源	使用方式	作用
SFT 提示数据	作为 PPO 的输入提示	生成多样化的回复样本
奖励模型	对生成回复进行评分	提供优化信号
原始 SFT 数据	计算 PTX 损失	防止模型偏离基础能力
人工编写提示	补充特定领域需求	增强领域适应性

```

    for prompt in prompts:
        # PPO策略优化
        response = policy_model.generate(prompt)
        reward = reward_model.score(prompt, response)

        # 计算PTX损失（防止偏离SFT模型）
        sft_logits = sft_model(prompt)
        current_logits = policy_model(prompt)
        ptx_loss = F.kl_div(
            F.log_softmax(current_logits, dim=-1),
            F.softmax(sft_logits, dim=-1),
            reduction='batchmean'
        )

        # 综合损失
        total_loss = ppo_loss + ptx_weight * ptx_loss
        optimize(total_loss)
```

39.3　训练数据集资源指南

39.3.1　公开数据集推荐

Alpaca-COT 数据集

Alpaca-COT 是目前最全面的指令微调数据集集合，特点包括：

- **多语言支持**：包含中英文指令数据
- **任务多样**：覆盖常识推理、数学计算、代码生成等
- **质量统一**：经过统一清洗和格式化处理
- **持续更新**：社区持续贡献新数据和改进

RedPajama-Data-1T 开源计划

RedPajama 是重要的开源预训练数据集项目，包含三个核心部分：

表 39.3: RedPajama 数据集组成

组件	内容描述	数据规模	用途
预训练数据集	7 个子集，覆盖多领域	压缩后 3TB，解压 5TB	基础模型预训练
基础模型	在数据集上训练的模型	1B-7B 参数规模	研究和发展基础
指令调优数据	对齐和安全性数据	百万级指令对	模型对齐微调

数据集子集详情

RedPajama 包含的 7 个子集及其特点：

1. **Common Crawl**：网页爬取数据，覆盖广泛主题
2. **C4**：经过清洗的网页文本，质量较高
3. **GitHub**：代码数据，提升编程能力
4. **Wikipedia**：百科全书知识，事实准确性高
5. **ArXiv**：学术论文，增强推理能力
6. **BookCorpus**：书籍文本，提升叙事连贯性
7. **StackExchange**：问答数据，增强问题解决能力

39.3.2　领域预训练数据选择

领域适应性考量

进行领域大模型预训练时，应优先选择：

- **高质量学术数据**：论文、专利等，知识密度高
- **领域相关网站**：专业论坛、技术博客等
- **新闻资讯**：时效性强，语言规范
- **结构化数据**：数据库、知识图谱等

数据选择矩阵

39.4　微调数据量需求分析

39.4.1　数据量影响因素

分布一致性关键作用

微调所需数据量主要取决于预训练数据与微调任务的数据分布一致性：

表 39.4: 领域预训练数据选择指南

数据来源	知识密度	语言质量	领域相关性	推荐指数
学术论文	高	高	高	
专业书籍	高	高	中高	
技术文档	中高	中高	高	
论坛讨论	中	中	高	
新闻资讯	中	高	中	
社交媒体	低	低	中	

$$\text{所需数据量} \propto \frac{1}{\text{分布相似度}}$$

- **高相似度**（分布一致）：100-1000 条足够
- **中相似度**：1000-10000 条较为理想
- **低相似度**（分布差异大）：需要万条以上数据

任务复杂度影响

表 39.5: 不同复杂度任务的数据需求

任务类型	典型数据量	训练轮数	说明
简单分类任务	100-500 条	5-10 轮	类别少，模式简单
复杂推理任务	1000-5000 条	10-20 轮	需要多步推理
专业领域任务	5000-20000 条	20-50 轮	冷门领域需更多数据
创造性生成	2000-10000 条	15-30 轮	需要学习风格和创意

39.4.2　冷门领域数据策略

药品识别案例

对于药品名称识别等冷门领域任务，数据需求具有特殊性：

- **数据稀缺**：公开数据有限，需要主动收集
- **专业要求高**：需要领域专家参与标注
- **长尾分布**：罕见药品样本少但很重要
- **迭代需求**：需要多轮训练才能稳定掌握

多轮训练必要性

即使只有 100 条微调数据，也需要足够训练轮数（如 20 轮）才能稳定拟合：

$$\text{有效训练量} = \text{数据量} \times \text{训练轮数} \times \text{数据质量}$$

39.5　微调数据构建方法论

39.5.1　最优微调数据特征

数据多样性要求

优质微调数据应具备良好的多样性，避免长尾分布问题。

长尾分布挑战

在实际数据收集中，数据往往呈现典型的长尾分布：

- **头部类别**：少数热门类别占据大部分数据
- **长尾类别**：多数类别数据稀缺但同样重要
- **采样偏差**：直接采样会过度代表热门类别

小红书案例

以小红书内容为例的分布分析：

表 39.6: 小红书内容类型数据分布

内容类型	数据占比	标注效率	模型需求
美食攻略	30%	高	中
穿搭分享	25%	高	中
旅游攻略	20%	高	中
美妆教程	15%	中	中
大模型技术	5%	低	高
其他专业内容	5%	低	高

39.5.2　数据构建先进方法

39.5.3　Self-Instruct 框架

核心思想

通过语言模型自我生成指令、输入和输出样本，然后清洗后用于微调原始模型。

实现流程

```
class SelfInstructGenerator:
    """Self-Instruct数据生成器"""

    def __init__(self, base_model, seed_tasks=100):
        self.model = base_model
        self.seed_tasks = seed_tasks

    def generate_instructions(self):
        """生成多样化的指令"""
        instructions = []

        # 1. 使用种子任务生成指令模板
        seed_instructions = self.load_seed_instructions()

        for seed in seed_instructions:
            # 2. 基于种子指令生成变体
            variations = self.generate_variations(seed)
            instructions.extend(variations)

            # 3. 指令去重和过滤
            instructions = self.deduplicate(instructions)

        return instructions

    def generate_input_output_pairs(self, instructions):
        """为指令生成输入输出对"""
        pairs = []

        for instruction in instructions:
            # 使用模型生成多个输入输出对
            prompt = f"为以下指令生成输入和输出: {instruction}"
            responses = self.model.generate(prompt, num_return_sequences
                =3)

            for response in responses:
                input_text, output_text = self.parse_response(response)
                if self.quality_check(input_text, output_text):
                    pairs.append({
```

```
                        'instruction': instruction,
                        'input': input_text,
                        'output': output_text
                    })

        return pairs

    def quality_check(self, input_text, output_text):
        """质量检查"""
        # 检查长度合理性
        if len(output_text) < 10 or len(output_text) > 1000:
            return False

        # 检查内容相关性
        if not self.check_relevance(input_text, output_text):
            return False

        return True
```

39.5.4　主动学习策略

两大基本原则

主动学习关注数据的两个方面：多样性和不确定性。

多样性保障：数据去重

```
def diversity_selection(dataset, target_size, method='k_center'):
    """基于多样性的数据选择"""

    if method == 'k_center':
        # K-Center-Greedy算法
        selected_indices = k_center_greedy(dataset, target_size)
    elif method == 'clustering':
        # 聚类采样
        selected_indices = cluster_sampling(dataset, target_size)
    else:
        # 简单去重
        selected_indices = simple_deduplication(dataset, target_size)

```

```
    return selected_indices

def k_center_greedy(dataset, k):
    """K-Center-Greedy多样性选择"""
    # 1. 随机选择第一个中心
    centers = [random.randint(0, len(dataset)-1)]

    # 2. 迭代选择距离当前中心最远的点
    while len(centers) < k:
        max_distance = -1
        next_center = -1

        for i in range(len(dataset)):
            if i not in centers:
                # 计算到最近中心的距离
                min_dist = min([cosine_distance(dataset[i], dataset[c])
                                for c in centers])
                if min_dist > max_distance:
                    max_distance = min_dist
                    next_center = i

        centers.append(next_center)

    return centers
```

差异性数据发现

```
def find_diverse_data(existing_data, candidate_data, model_name='deberta'
    ):
    """发现与现有数据差异大的新数据"""

    # 准备训练数据：现有数据为正样本，候选数据为负样本
    train_data = []
    for item in existing_data:
        train_data.append({'text': item, 'label': 1})
    for item in candidate_data:
        train_data.append({'text': item, 'label': 0})

    # K折交叉验证
```

```
    kf = KFold(n_splits=5, shuffle=True)
    diverse_samples = []

    for train_idx, test_idx in kf.split(train_data):
        # 训练二分类器
        classifier = train_binary_classifier(
            [train_data[i] for i in train_idx], model_name
        )

        # 在测试集中选择预测概率接近0的样本（与现有数据差异大）
        test_subset = [train_data[i] for i in test_idx if train_data[i]['
           label'] == 0]
        if test_subset:
            predictions = classifier.predict([item['text'] for item in
               test_subset])
            for i, pred in enumerate(predictions):
                if pred < 0.1:  # 概率接近0，表示与现有数据差异大
                    diverse_samples.append(test_subset[i]['text'])

    return list(set(diverse_samples))
```

不确定性采样

```
def uncertainty_sampling(model, unlabeled_data, strategy='entropy'):
    """基于不确定性的数据选择"""

    uncertain_samples = []

    for data in unlabeled_data:
        if strategy == 'entropy':
            # 基于信息熵的不确定性
            probs = model.predict_proba(data)
            entropy = -np.sum(probs * np.log(probs + 1e-8))
            uncertainty = entropy

        elif strategy == 'margin':
            # 基于间隔的不确定性
            probs = model.predict_proba(data)
            probs_sorted = np.sort(probs)[::-1]
```

```
            uncertainty = probs_sorted[0] - probs_sorted[1]

        elif strategy == 'least_confidence':
            # 基于最小置信度
            probs = model.predict_proba(data)
            uncertainty = 1 - np.max(probs)

        uncertain_samples.append((data, uncertainty))

    # 按不确定性排序，选择最不确定的样本
    uncertain_samples.sort(key=lambda x: x[1], reverse=True)
    return [item[0] for item in uncertain_samples[:top_k]]

def quality_aware_sampling(model, reward_model, unlabeled_data):
    """质量感知的不确定性采样"""
    candidates = []

    for data in unlabeled_data:
        # 1. 计算模型不确定性
        uncertainty = calculate_uncertainty(model, data)

        # 2. 使用奖励模型评估数据质量
        quality_score = reward_model.predict(data)

        # 3. 综合选择：高不确定性 + 高质量
        if uncertainty > 0.7 and quality_score > 0.8:
            candidates.append(data)

    return candidates
```

39.6　数据质量评估体系

39.6.1　多维度质量指标

标注质量评估

- **一致性检验**：多名标注者间的一致性分数
- **准确率验证**：与金标准对比的准确率
- **完整性检查**：必要字段是否完整填写

- **及时性评估**：标注任务完成时效

数据质量量化

表 39.7: 数据质量评估指标体系

质量维度	评估指标	计算方法	目标值	权重
准确性	准确率	正确样本/总样本	>95%	0.3
一致性	Cohen's Kappa	标注者一致性	>0.8	0.2
多样性	类别分布熵	信息熵计算	最大化	0.15
覆盖度	主题覆盖率	覆盖主题数/总主题数	>90%	0.15
平衡性	类别均衡度	最小类别占比	>5%	0.1
时效性	数据新鲜度	新数据比例	>20%	0.1

39.7　实践建议与最佳实践

39.7.1　数据构建流程优化

迭代式数据构建

1. **初代数据**：收集基础种子数据（100-1000 条）
2. **模型训练**：在种子数据上训练初始模型
3. **弱点分析**：分析模型在哪些数据上表现差
4. **针对性补充**：针对弱点收集更多数据
5. **迭代优化**：重复 2-4 步直至满足要求

质量控制闭环

```
class DataQualityLoop:
    """数据质量闭环管理系统"""

    def __init__(self, initial_data, quality_threshold=0.9):
        self.data_pool = initial_data
        self.quality_threshold = quality_threshold
        self.quality_model = None

    def run_quality_loop(self, num_iterations=5):
        """运行质量优化循环"""
```

```
        for iteration in range(num_iterations):
            print(f"=== 第{iteration+1}轮数据质量优化 ===")

            # 1. 训练质量评估模型
            self.train_quality_model()

            # 2. 评估当前数据质量
            quality_scores = self.evaluate_data_quality()

            # 3. 识别低质量数据
            low_quality_data = self.identify_low_quality(quality_scores)

            if len(low_quality_data) == 0:
                print("数据质量已达标准，停止优化")
                break

            # 4. 改进低质量数据
            improved_data = self.improve_data_quality(low_quality_data)

            # 5. 更新数据池
            self.update_data_pool(improved_data)

            print(f"本轮改进数据量: {len(improved_data)}")

    def identify_low_quality(self, quality_scores, threshold=0.7):
        """识别低质量数据"""
        low_quality_indices = []
        for i, score in enumerate(quality_scores):
            if score < threshold:
                low_quality_indices.append(i)
        return low_quality_indices
```

39.7.2　数据管理最佳实践

版本控制

- **数据版本化**：使用 Git LFS 或 DVC 管理数据版本
- **变更记录**：记录每次数据更新的内容和原因
- **回滚机制**：支持快速回退到之前的数据版本

- **差异分析**：分析不同版本数据对模型性能的影响

元数据管理

```
# 数据样本元数据结构
data_metadata = {
    "sample_id": "unique_identifier",
    "create_time": "2024-01-27T10:30:00Z",
    "source": "human_annotation|model_generation|crawled",
    "annotator_id": "annotator_001",
    "quality_score": 0.95,
    "difficulty_level": "easy|medium|hard",
    "domain": "technology|medical|finance",
    "language": "zh|en",
    "version": "v1.2.3",
    "tags": ["instruction_following", "reasoning", "long_form"]
}
```

39.8　总结与展望

39.8.1　技术总结

高质量训练数据是大语言模型成功的基石。从 SFT 的指令-回复对，到 RM 的偏好排序数据，再到 PPO 的提示数据，每个训练阶段都需要精心设计和构建相应的数据集。

39.8.2　未来发展方向

- **自动化数据生成**：利用大模型自动生成高质量训练数据
- **智能数据选择**：基于主动学习和不确定性采样的智能数据选择
- **多模态数据**：融合文本、图像、语音的多模态训练数据
- **持续学习数据**：支持模型持续学习而不遗忘的数据策略
- **隐私保护数据**：在保护隐私的前提下有效利用数据
- **可解释数据**：增强数据决策过程的透明度和可解释性

随着大模型技术的不断发展，训练数据的构建和管理将变得更加智能化和自动化，为构建更强大、更安全、更有用的大语言模型奠定坚实基础。

第四十章　大语言模型 SFT 数据生成技术

40.1　引言：SFT 数据生成的重要性与挑战

40.1.1　SFT 在大模型训练中的关键作用

有监督微调（Supervised Fine-Tuning，SFT）是大语言模型训练流程中的关键环节，它连接了预训练基础模型与最终的对齐优化阶段。SFT 数据的质量直接决定了模型在特定任务上的表现和指令遵循能力。

40.1.2　数据生成方法对比

表 40.1: SFT 数据生成方法对比

方法类型	优势	局限性
人工标注	质量高、准确性好、减少偏差	成本高、效率低、规模受限
LLM 生成	效率高、成本低、可大规模生成	可能存在偏差、需要质量控制
混合方法	平衡质量与效率	流程复杂、需要协调

40.1.3　方法选择考量因素

- **领域特异性**：垂直领域建议人工标注，通用领域可 LLM 生成
- **数据质量要求**：高质量要求场景优先人工标注
- **资源约束**：根据预算和时间选择合适方法
- **规模需求**：大规模数据需求倾向 LLM 生成

40.2　Self-Instruct 数据生成方法

40.2.1　Self-Instruct 技术概述

基本概念

Self-Instruct 是一种通过引导语言模型自我生成指令数据来提升其指令遵循能力的框架。该方法的核心理念是利用模型自身的能力来扩展训练数据。

技术原理

Self-Instruct 基于模¨型知道得比它表现出来的多的¨假设，通过恰当的提示工程，可以激发模型内部已有的知识，生成高质量的指令-回复对。

40.2.2　Self-Instruct 实现流程

四阶段流程

Self-Instruct 包含四个核心步骤，形成完整的数据生成流水线：

阶段一：指令生成

1. **种子指令：**从 175 个种子任务中随机选择 8 条自然语言指令作为示例
2. **模型生成：**使用 InstructGPT（或类似模型）基于示例生成新指令
3. **多样性保证：**通过示例的多样性确保生成指令的覆盖面

```
def generate_instructions(seed_instructions, num_examples=8, model_name="
    InstructGPT"):
    """生成新指令"""

    # 从种子指令中随机选择示例
    examples = random.sample(seed_instructions, num_examples)

    prompt = f"""
基于以下指令示例，生成新的多样化指令：

示例指令：
{chr(10).join([f'{i+1}. {example}' for i, example in enumerate(examples)
    ])}

新生成的指令：
"""

```

```
    # 使用语言模型生成
    new_instructions = model.generate(
        prompt,
        max_tokens=500,
        temperature=0.7,
        num_return_sequences=5
    )

    return new_instructions
```

阶段二：任务类型识别与处理

- **分类任务识别**：判断生成的指令是否为分类任务
- **分类任务处理**：如果是分类任务，生成所有可能的选项类别
- **非分类任务**：采用输入优先策略

```
def process_instruction(instruction, model):
    """处理单条指令，识别任务类型并相应处理"""

    # 判断是否为分类任务
    classification_prompt = f"""
判断以下指令是否是分类任务（需要从有限选项中选择）：
指令：\"{instruction}\"

回答（是/否）：
"""
    is_classification = model.generate(classification_prompt)

    if "是" in is_classification:
        # 分类任务处理
        return process_classification_task(instruction, model)
    else:
        # 非分类任务处理
        return process_non_classification_task(instruction, model)

def process_classification_task(instruction, model):
    """处理分类任务"""
    # 生成所有可能的选项
    options_prompt = f"""
指令：\"{instruction}\"
```

```
这是一个分类任务，请列出所有可能的选项类别：
"""
    options = model.generate(options_prompt)

    # 为每个选项生成输入内容
    results = []
    for option in parse_options(options):
        input_prompt = f"""
指令：\"{instruction}\"
选项：\"{option}\"
请生成适合此分类任务的输入内容：
"""
        input_content = model.generate(input_prompt)
        results.append({
            'instruction': instruction,
            'input': input_content,
            'output': option
        })

    return results
```

阶段三：输入输出生成

根据任务类型采用不同的生成策略：

输入优先策略（非分类任务）

1. 先生成与指令相关的输入内容
2. 基于指令和生成的输入生成对应的输出

输出优先策略（分类任务）

1. 先确定所有可能的输出类别
2. 为每个类别生成相应的输入内容

```
def process_non_classification_task(instruction, model):
    """处理非分类任务 - 输入优先策略"""

    # 第一步：生成输入
    input_prompt = f"""
基于以下指令生成一个合适的输入：
指令：\"{instruction}\"
```

```
8
9  输入:
10 """
11     input_content = model.generate(input_prompt)
12
13     # 第二步: 基于指令和输入生成输出
14     output_prompt = f"""
15 根据指令和输入生成合适的输出:
16 指令: \"{instruction}\"
17 输入: \"{input_content}\"
18
19 输出:
20 """
21     output_content = model.generate(output_prompt)
22
23     return [{
24         'instruction': instruction,
25         'input': input_content,
26         'output': output_content
27     }]
28
29 def generate_with_output_first(instruction, options, model):
30     """输出优先策略生成"""
31     results = []
32
33     for option in options:
34         # 为每个输出选项生成对应的输入
35         input_prompt = f"""
36 指令: \"{instruction}\"
37 期望输出: \"{option}\"
38 请生成一个会导致此输出的输入:
39 """
40         input_content = model.generate(input_prompt)
41
42         results.append({
43             'instruction': instruction,
44             'input': input_content,
45             'output': option
46         })
47
```

```
    return results
```

阶段四：后处理与质量控制

- **去重处理**：过滤相似的指令和内容
- **质量过滤**：移除低质量或不合规的生成内容
- **格式标准化**：统一数据格式和结构

```
def post_process_generated_data(raw_data, similarity_threshold=0.8):
    """后处理生成的数据"""

    processed_data = []

    for item in raw        # 1. 质量检查
        if not quality_check(item):
            continue

        # 2. 去重检查
        if is_duplicate(item, processed_data, similarity_threshold):
            continue

        # 3. 格式标准化
        standardized_item = standardize_format(item)
        processed_data.append(standardized_item)

    return processed_data

def quality_check(data_item):
    """质量检查"""
    checks = [
        # 检查指令是否明确
        len(data_item['instruction'].strip()) > 10,
        # 检查输出是否合理
        len(data_item['output'].strip()) > 5,
        # 检查内容是否合规
        not contains_inappropriate_content(data_item),
        # 检查逻辑一致性
        is_logically_consistent(data_item)
    ]

```

```
    return all(checks)

def is_duplicate(new_item, existing_items, threshold=0.8):
    """重复性检查"""
    for existing in existing_items:
        similarity = calculate_similarity(new_item['instruction'],
                                          existing['instruction'])
        if similarity > threshold:
            return True
    return False
```

40.2.3　Self-Instruct 技术优势

效率提升

- **规模化生成**：可快速生成大量训练数据
- **成本效益**：显著降低人工标注成本
- **迭代优化**：支持多轮生成和持续改进

质量保障

表 40.2: Self-Instruct 生成数据质量指标

质量维度	评估方法	目标值	实际表现
指令多样性	聚类分析	高	52K 条不重复指令
任务覆盖度	类别分布	广泛	覆盖多领域任务
内容质量	人工评估	>90% 合格	85-95% 合格率
逻辑一致性	规则检查	>95%	90-98%
语言流畅性	困惑度评估	低困惑度	接近人工水平

40.3　回译数据生成方法

40.3.1　回译技术概述

传统回译方法

在机器翻译领域，回译是一种经典的数据增强技术：

- **流程**：原文 → 翻译 → 回译 → 新文本

- **目标**：生成语义相同但表达不同的文本
- **应用**：增加训练数据的多样性

SFT 数据生成中的回译创新

在 SFT 数据生成中，回译方法进行了创新性应用：

- **核心思想**：通过输出内容反推生成指令
- **流程反转**：传统是文本 → 翻译 → 回译，现在是输出 → 生成 → 指令
- **应用目标**：生成高质量的指令-输出对

40.3.2　回译方法实现流程

基本流程

回译方法包含三个核心步骤：

步骤一：输出生成　给定一个主题或领域，生成相关的输出内容（如答案、解决方案等）。

步骤二：指令生成　基于生成的输出内容，反推可能产生此输出的指令或问题。

步骤三：质量验证　对生成的指令-输出对进行质量检查和筛选。

```
def backtranslation_data_generation(topic_domains, num_samples_per_domain=100):
    """回译方法生成SFT数据"""

    all_data = []

    for domain in topic_domains:
        domain_data = []

        for i in range(num_samples_per_domain):
            # 1. 生成输出内容
            output = generate_output_for_domain(domain)

            # 2. 基于输出生成指令
            instruction = generate_instruction_from_output(output)

            # 3. 验证和筛选
            if validate_pair(instruction, output):
                domain_data.append({
```

```
                    'instruction': instruction,
                    'input': '',  # 可为空或根据需要生成
                    'output': output
                })

        all_data.extend(domain_data)

    return all_data

def generate_output_for_domain(domain, model):
    """为特定领域生成输出内容"""
    prompt = f"""
请生成一段关于{domain}的高质量文本内容（如解释、答案、解决方案等）：
"""
    output = model.generate(prompt, max_tokens=300)
    return output.strip()

def generate_instruction_from_output(output, model):
    """基于输出内容生成对应的指令"""
    prompt = f"""
给定以下文本内容，请生成一个可能引出此内容的问题或指令：
内容：\"{output}\"

可能的问题/指令：
"""
    instruction = model.generate(prompt, max_tokens=100)
    return instruction.strip()
```

质量增强策略

```
def enhanced_backtranslation(output_content, model,
   enhancement_strategies=None):
    """增强的回译方法"""

    if enhancement_strategies is None:
        enhancement_strategies = ['multi_perspective', '
           difficulty_control', 'style_variation']

    generated_pairs = []
```

```
8
9       # 多视角指令生成
10      if 'multi_perspective' in enhancement_strategies:
11          perspectives = ['初学者角度', '专家角度', '实用角度', '理论角度']
12          for perspective in perspectives:
13              instruction = generate_instruction_from_perspective(
                    output_content, perspective, model)
14              if validate_pair(instruction, output_content):
15                  generated_pairs.append({'instruction': instruction, '
                        output': output_content})
16
17      # 难度控制
18      if 'difficulty_control' in enhancement_strategies:
19          difficulty_levels = ['简单', '中等', '困难']
20          for level in difficulty_levels:
21              instruction = generate_instruction_with_difficulty(
                    output_content, level, model)
22              if validate_pair(instruction, output_content):
23                  generated_pairs.append({'instruction': instruction, '
                        output': output_content})
24
25      return generated_pairs
26
27  def generate_instruction_from_perspective(output, perspective, model):
28      """从特定视角生成指令"""
29      prompt = f"""
30  从{perspective}出发, 针对以下内容生成一个相关问题或指令:
31  内容: \"{output}\"
32
33  {perspective}的问题/指令:
34  """
35      return model.generate(prompt).strip()
```

40.3.3　回译方法技术优势

内容质量保证

- **输出驱动**：先确保输出内容的质量和准确性
- **语义一致性**：指令与输出具有天然的语义关联
- **知识准确性**：基于准确的知识内容生成指令

多样性生成

表 40.3: 回译方法生成的指令类型

指令类型	生成策略	应用场景
解释性指令	要求解释输出内容中的概念	知识传授、概念解释
应用性指令	要求应用输出中的方法或原理	实践指导、问题解决
分析性指令	要求分析输出内容的深层含义	批判性思维、深度分析
创造性指令	基于输出进行扩展或创新	创意生成、思维拓展
比较性指令	要求比较输出与其他相关概念	对比分析、关系理解

40.4　混合数据生成策略

40.4.1　人工与自动生成的结合

混合流程设计

结合人工标注和 LLM 生成的混合策略可以平衡质量和效率：

1. **种子数据创建**：人工创建高质量种子数据（100-500 条）
2. **模型学习**：让 LLM 学习人工数据的质量和风格
3. **批量生成**：使用 LLM 基于种子数据扩展生成
4. **人工审核**：对生成数据进行抽样审核和质量控制
5. **迭代优化**：根据审核结果调整生成策略

质量控制机制

```
class HybridDataGenerator:
    """混合数据生成器"""

    def __init__(self, human_seed_data, quality_threshold=0.8):
        self.human_data = human_seed_data
        self.quality_threshold = quality_threshold
        self.quality_model = self.train_quality_model(human_seed_data)

    def generate_with_quality_control(self, target_size=10000):
        """带质量控制的批量生成"""

        generated_data = []
```

```
        batch_size = 1000

        while len(generated_data) < target_size:
            # 1. 批量生成
            batch = self.generate_batch(batch_size)

            # 2. 质量过滤
            high_quality_batch = self.quality_filter(batch)

            # 3. 人工抽样审核
            approved_batch = self.human_review_sample(high_quality_batch)

            generated_data.extend(approved_batch)

            # 4. 模型更新
            if len(approved_batch) > 100:
                self.update_generation_model(approved_batch)

        return generated_data

    def quality_filter(self, data_batch, threshold=0.7):
        """基于质量模型进行过滤"""
        quality_scores = self.quality_model.predict(data_batch)
        return [data for data, score in zip(data_batch, quality_scores)
                if score >= threshold]

    def human_review_sample(self, data_batch, sample_ratio=0.1):
        """人工抽样审核"""
        sample_size = max(1, int(len(data_batch) * sample_ratio))
        sample_indices = random.sample(range(len(data_batch)),
           sample_size)

        approved_indices = []
        for idx in sample_indices:
            if human_reviewer.approve(data_batch[idx]):
                approved_indices.append(idx)

        # 如果样本通过率低，整批数据需要重新生成或严格过滤
        approval_rate = len(approved_indices) / sample_size
        if approval_rate < 0.5:
```

```
            return self.strict_filter(data_batch)

        return [data_batch[i] for i in range(len(data_batch))]
```

40.5　数据质量评估体系

40.5.1　多维度评估指标

自动化评估指标

- **多样性指标**：使用嵌入向量聚类分析指令多样性
- **复杂性指标**：评估指令的语法复杂性和认知需求
- **一致性指标**：检查指令与输出的逻辑一致性
- **流畅性指标**：基于语言模型困惑度评估文本质量

人工评估标准

表 40.4: SFT 数据人工评估标准

评估维度	优秀标准	合格标准	权重	评分指南
指令清晰度	明确无歧义，易于理解	基本清晰，偶有歧义	25%	1-5 分制
任务合理性	有明确解决目标	任务目标基本合理	20%	1-5 分制
输出准确性	信息准确，逻辑严谨	主要信息准确	25%	1-5 分制
内容相关性	输出与指令高度相关	基本相关，偶有偏离	15%	1-5 分制
语言质量	流畅自然，符合语法	基本通顺，偶有错误	15%	1-5 分制

40.5.2　评估实施流程

```
def comprehensive_quality_evaluation(dataset, num_human_evaluators=3):
    """综合质量评估"""

    evaluation_results = {
        'automated_metrics': {},
        'human_scores': {},
        'final_quality_score': 0
    }

    # 1. 自动化指标计算
```

```
    evaluation_results['automated_metrics'] = calculate_automated_metrics
        (dataset)

    # 2. 人工评估抽样
    sample_size = min(100, len(dataset) // 10)
    sample_indices = random.sample(range(len(dataset)), sample_size)
    sample_data = [dataset[i] for i in sample_indices]

    # 3. 多评估者评分
    human_scores = []
    for evaluator_id in range(num_human_evaluators):
        scores = human_evaluation(sample_data, evaluator_id)
        human_scores.append(scores)

    # 4. 计算一致性
    consistency = calculate_inter_annotator_agreement(human_scores)
    evaluation_results['human_consistency'] = consistency

    # 5. 综合评分
    automated_score = weighted_average(evaluation_results['
        automated_metrics'])
    human_score = average_human_score(human_scores)

    # 结合自动和人工评分
    final_score = 0.7 * human_score + 0.3 * automated_score
    evaluation_results['final_quality_score'] = final_score

    return evaluation_results

def calculate_automated_metrics(dataset):
    """计算自动化评估指标"""
    metrics = {}

    # 指令多样性
    instruction_embeddings = [get_embedding(item['instruction']) for item
        in dataset]
    metrics['diversity'] = calculate_diversity(instruction_embeddings)

    # 内容质量
    perplexities = [calculate_perplexity(item['output']) for item in
```

```
        dataset]
    metrics['avg_perplexity'] = np.mean(perplexities)

    # 一致性检查
    consistency_scores = [check_consistency(item) for item in dataset]
    metrics['consistency'] = np.mean(consistency_scores)

    return metrics
```

40.6　实践建议与最佳实践

40.6.1　方法选择指南

根据场景选择方法

表 40.5: 数据生成方法选择指南

应用场景	推荐方法	配置建议	注意事项
通用领域	Self-Instruct	大规模生成 + 自动过滤	注意偏差控制
垂直领域	回译方法	领域知识驱动生成	确保专业性
高质量要求	混合方法	人工种子 +LLM 扩展	成本较高
快速原型	Self-Instruct	基础配置快速启动	后期需要优化
生产环境	混合方法	完整质量管控流程	长期维护

40.6.2　质量控制最佳实践

多层次质量控制

1. **生成阶段控制：**通过提示工程约束生成质量
2. **自动过滤：**基于规则和模型的质量过滤
3. **人工审核：**抽样审核和重点领域全量审核
4. **持续监控：**生产环境中的数据质量监控

迭代优化流程

```
def iterative_quality_improvement(initial_generator, target_quality=0.9,
    max_iterations=10):
    """迭代质量改进流程"""
```

```
    current_quality = 0
    iteration = 0
    best_generator = initial_generator

    while current_quality < target_quality and iteration < max_iterations:
        print(f"第{iteration+1}轮质量改进")

        # 1. 生成新数据
        new_data = best_generator.generate(1000)

        # 2. 质量评估
        quality_report = comprehensive_quality_evaluation(new_data)
        current_quality = quality_report['final_quality_score']

        print(f"当前质量分数: {current_quality:.3f}")

        # 3. 分析问题
        issues = analyze_quality_issues(quality_report)

        # 4. 调整生成策略
        if current_quality < target_quality:
            improved_generator = adjust_generation_strategy(
                best_generator, issues)
            best_generator = improved_generator

        iteration += 1

    return best_generator, current_quality
```

40.7　总结与展望

40.7.1　技术总结

SFT 数据生成是大模型训练中的关键技术环节，Self-Instruct 和回译方法为高效生成高质量训练数据提供了有效途径。两种方法各有优势，可根据具体需求选择或组合使用。

40.7.2　未来发展方向

技术改进方向

- **生成质量提升**：更智能的质量控制和优化
- **偏差减少**：更好的偏差检测和纠正机制
- **多模态扩展**：支持图像、语音等多模态数据生成
- **个性化生成**：根据特定需求定制化数据生成

应用拓展方向

- **领域自适应**：更好地适应特定领域需求
- **低成本方案**：进一步降低高质量数据生成成本
- **实时生成**：支持在线学习和实时数据生成
- **可解释生成**：提高生成过程的透明度和可控性

随着大模型技术的不断发展，SFT 数据生成技术将继续演进，为构建更强大、更安全、更有用的大语言模型提供坚实的数据基础。

第四十一章　大语言模型显存优化与性能评估技术详解

41.1　引言：大模型显存挑战概述

41.1.1　大模型规模与显存需求

随着大语言模型（LLMs）参数规模的快速增长，从数亿参数发展到数千亿参数，显存需求成为模型训练和推理的关键瓶颈。准确估算和优化显存使用对于大模型实践至关重要。

41.1.2　显存需求的核心影响因素

- **模型参数**：模型权重占用的显存空间
- **精度选择**：FP32、FP16、INT8 等不同精度的影响
- **训练组件**：参数、梯度、优化器状态等额外开销
- **激活内存**：前向传播中的中间计算结果
- **序列长度**：输入序列长度对显存的显著影响

41.2　模型规模与文件大小

41.2.1　参数规模表示法

大模型规模通常使用两种表示方法：

- **参数量表示**：nB 模型表示 n billion（十亿）参数
- **文件大小表示**：基于精度计算的实际存储大小

表 41.1: 不同精度下的模型存储需求

精度类型	比特数	字节数	计算公式	示例（6B 模型）
FP32	32 bits	4 bytes	$n \times 4$ GB	24 GB
FP16	16 bits	2 bytes	$n \times 2$ GB	12 GB
INT8	8 bits	1 byte	$n \times 1$ GB	6 GB
INT4	4 bits	0.5 bytes	$n \times 0.5$ GB	3 GB

41.2.2　精度与存储关系

41.2.3　实际文件大小计算

对于 nB 参数的模型，不同精度下的实际文件大小：

$$\text{文件大小} = \begin{cases} n \times 4 \text{ GB} & \text{FP32 精度} \\ n \times 2 \text{ GB} & \text{FP16 精度（常见发布格式）} \\ n \times 1 \text{ GB} & \text{INT8 精度} \\ n \times 0.5 \text{ GB} & \text{INT4 精度} \end{cases}$$

41.3　硬件配置可行性分析

41.3.1　Vicuna-65B 训练硬件需求

基础显存需求分析

$$\begin{aligned} \text{模型参数显存} &= 65 \times 2 \text{ GB} = 130 \text{ GB} \quad \text{(FP16 精度)} \\ \text{总显存需求} &> 130 \text{ GB} \quad \text{(需额外空间存储梯度、优化器等)} \end{aligned}$$

4×V100-32G 配置分析

$$\begin{aligned} \text{总可用显存} &= 4 \times 32 \text{ GB} = 128 \text{ GB} \\ \text{显存缺口} &= 130 - 128 = 2 \text{ GB} \quad \text{(仅参数已超出)} \end{aligned}$$

41.3.2　技术限制分析

- **架构兼容性**：Vicuna 使用 Flash-Attention 加速，需要 Turing 架构及更新显卡
- **显存不足**：即使忽略架构限制，显存总量也不足以加载 65B 模型参数
- **实际可行性**：需要至少 5 张 V100-32G 才能完整加载模型参数

41.3.3　替代解决方案

```
# 最小配置下的可行性方案
def check_vicuna_65b_feasibility():
    """检查Vicuna-65B训练可行性"""

    requirements = {
        'min_gpu_memory': 50,  # GB - 通过量化降低需求
        'architecture': 'Turing+',  # 需要支持Flash-Attention
        'quantization': 'INT4',  # 必须使用量化
        'method': 'LoRA',  # 参数高效微调
    }

    actual_config = {
        'gpu_memory': 32,  # V100 32GB
        'architecture': 'Volta',  # V100架构
        'quantization_support': True,
        'lora_support': True
    }

    # 架构检查
    if actual_config['architecture'] != 'Turing+':
        print("架构不兼容: 需要Turing及以上架构支持Flash-Attention")
        return False

    # 显存检查
    if actual_config['gpu_memory'] < requirements['min_gpu_memory']:
        print("显存不足: 需要至少50GB显存")
        return False

    return True
```

41.4　低显存环境下的解决方案

41.4.1　量化技术应用

GPTQ 量化原理

GPTQ（GPT Quantization）是一种后训练量化技术，可将 FP16 模型转换为 INT4 表示：

$$\text{压缩比} = \frac{\text{FP16 大小}}{\text{INT4 大小}} = \frac{16}{4} = 4:1$$

LLaMA-65B-INT4 可行性

$$\begin{aligned}
\text{原始显存} &= 65 \times 2 = 130 \text{ GB} \quad \text{(FP16)} \\
\text{量化后显存} &= 65 \times 0.5 = 32.5 \text{ GB} \quad \text{(INT4)} \\
\text{显存节省} &= 130 - 32.5 = 97.5 \text{ GB}
\end{aligned}$$

41.4.2　LoRA 微调技术

参数高效微调

LoRA（Low-Rank Adaptation）仅训练少量额外参数，大幅降低显存需求：

```
class LoRAWrapper:
    """LoRA参数高效微调封装"""

    def __init__(self, base_model, lora_config):
        self.base_model = base_model
        self.lora_config = lora_config
        self.lora_parameters = {}

        # 冻结基础模型参数
        for param in base_model.parameters():
            param.requires_grad = False

        # 添加LoRA适配器
        self.add_lora_adapters()

    def add_lora_adapters(self):
        """为线性层添加LoRA适配器"""
        for name, module in self.base_model.named_modules():
            if isinstance(module, nn.Linear):
```

```
                # 创建LoRA层
                lora_layer = LoRALayer(
                    module.in_features,
                    module.out_features,
                    self.lora_config['r']
                )
                self.lora_parameters[name] = lora_layer

    def forward(self, x):
        # 基础模型前向传播
        base_output = self.base_model(x)

        # LoRA适配
        for name, lora_layer in self.lora_parameters.items():
            # 应用LoRA调整
            lora_adjustment = lora_layer(x)
            base_output += lora_adjustment

        return base_output

def estimate_lora_memory_savings(model_size_gb, lora_rank=8):
    """估算LoRA显存节省"""
    base_memory = model_size_gb * 2  # FP16参数

    # LoRA参数估算：秩为r的低秩矩阵
    lora_memory = model_size_gb * (lora_rank / 4096) * 2  # 假设隐藏层大小4096

    savings = base_memory - lora_memory
    return savings, lora_memory
```

41.5　显存需求详细估算

41.5.1　推理显存需求

基础计算公式

推理阶段仅需加载模型参数，显存需求相对简单：

$$\text{推理显存} = \text{参数量} \times \text{参数精度字节数}$$

不同精度下的推理需求

表 41.2: 不同规模模型的推理显存需求

模型规模	FP32 需求	FP16 需求	INT8 需求	INT4 需求
7B 模型	28 GB	14 GB	7 GB	3.5 GB
13B 模型	52 GB	26 GB	13 GB	6.5 GB
65B 模型	260 GB	130 GB	65 GB	32.5 GB
175B 模型	700 GB	350 GB	175 GB	87.5 GB

41.5.2　训练显存需求

训练组件分析

训练阶段需要存储多个组件：

$$\text{总显存} = \text{模型参数} + \text{梯度} + \text{优化器状态} + \text{激活值} + \text{其他开销}$$

优化器状态分析

以 AdamW 优化器为例：

- **参数**：FP16 精度，$2n$ bytes
- **梯度**：FP16 精度，$2n$ bytes
- **优化器状态**：FP32 精度，存储一阶动量 + 二阶动量，$4n + 4n = 8n$ bytes

总显存需求公式

$$\text{训练显存} = 2n + 2n + 8n = 12n \text{ bytes} \quad \text{(基础组件)}$$

$$\text{实际需求} \approx 16n \text{ bytes} \quad \text{(包含激活值和其他开销)}$$

实例验证：Vicuna-7B

$$\text{理论计算} = 7 \times 16 = 112 \text{ GB}$$

$$\text{实际需求} \approx 160 \text{ GB} \quad \text{(FSDP 实际测量)}$$

41.6　内存需求估算方法论

41.6.1　系统化估算框架

估算步骤

1. **确定模型规模**：参数数量（nB）
2. **选择精度方案**：FP16/INT8/INT4 等
3. **识别训练组件**：参数、梯度、优化器状态
4. **计算激活内存**：基于序列长度和批大小
5. **考虑系统开销**：CUDA 内核、通信缓冲等

41.6.2　LLaMA-6B 案例研究

精度影响分析

表 41.3: LLaMA-6B 不同精度下的内存需求

组件	FP32 需求	FP16 需求	INT8 需求
模型参数	24 GB	12 GB	6 GB
梯度	24 GB	12 GB	6 GB
优化器状态（AdamW）	48 GB	24 GB	12 GB
小计	96 GB	48 GB	24 GB

系统开销计算

```
def estimate_system_overhead():
    """估算系统级开销"""
    # CUDA内核基础开销
    torch.ones((1, 1)).to("cuda")
    base_overhead = get_gpu_memory_used()  # 约1.3GB

    # 通信缓冲区（分布式训练）
    comm_buffer = 0.5  # GB

    # 其他系统开销
    system_misc = 0.2  # GB

    total_overhead = base_overhead + comm_buffer + system_misc
```

```
    return total_overhead

# 实测系统开销
> torch.ones((1, 1)).to("cuda")
> print_gpu_utilization()
GPU memory occupied: 1343 MB
```

激活内存计算

基于 LLaMA 架构计算中间激活值内存：

$$\begin{aligned}\text{单层激活大小} &= (\text{hidden\_size} + \text{intermediate\_size}) \times \text{seq\_len} \\ &= (4096 + 11008) \times 2048 \times 1 \text{ byte} \\ &= 15104 \times 2048 \times 1 \text{ byte} = 30.94 \text{ MB}\end{aligned}$$

$$\begin{aligned}\text{总激活内存} &= \text{单层大小} \times \text{层数} \times \text{批大小} \\ &= 30.94 \text{ MB} \times 32 \times 50 \\ &= 49.5 \text{ GB}\end{aligned}$$

总内存需求汇总

表 41.4: LLaMA-6B 完整内存需求估算

组件	INT8 需求	计算依据	占比
模型参数	6 GB	$6B \times 1$ byte	23.7%
梯度	6 GB	$6B \times 1$ byte	23.7%
优化器状态	12 GB	AdamW：$6B \times 2$ bytes	47.4%
系统开销	1.3 GB	实测 CUDA 内核开销	5.1%
激活内存	0 GB	假设激活检查点技术	0%
总计	**25.3 GB**		**100%**

表 41.5: GPU 利用率评估方法比较

方法	核心原理	优势	局限性	准确度
FLOPS 比值法	实测 FLOPS/理论峰值	直接反映计算效率	需要专用工具	高
吞吐量估计法	实际吞吐/理论吞吐	易于实施	依赖参考数据	中
PyTorch Profiler	详细性能分析	全面深入	配置复杂	最高

41.7　GPU 利用率评估方法

41.7.1　评估方法概述

三种评估方法对比

41.7.2　FLOPS 比值法

理论基础

$$\text{GPU 利用率} = \frac{\text{实测 FLOPS}}{\text{理论峰值 FLOPS}} \times 100\%$$

具体实施

```python
def calculate_gpu_utilization_flops(measured_tflops, gpu_model="A100"):
    """通过FLOPS计算GPU利用率"""

    # GPU理论峰值FLOPS（Tensor Core）
    theoretical_peaks = {
        "A100": 312,  # TFLOPS (FP16 Tensor Core)
        "V100": 112,  # TFLOPS (FP16 Tensor Core)
        "H100": 989,  # TFLOPS (FP16 Tensor Core)
    }

    if gpu_model not in theoretical_peaks:
        raise ValueError(f"不支持的GPU型号: {gpu_model}")

    theoretical_peak = theoretical_peaks[gpu_model]
    utilization = (measured_tflops / theoretical_peak) * 100

    print(f"实测FLOPS: {measured_tflops} TFLOPS")
    print(f"理论峰值: {theoretical_peak} TFLOPS")
    print(f"GPU利用率: {utilization:.2f}%")
```

```
20
21      return utilization
22
23  # 示例：A100实测100 TFLOPS
24  utilization = calculate_gpu_utilization_flops(100, "A100")
25  # 输出：GPU利用率：32.05%
```

DeepSpeed 配置示例

```
{
  "flops_profiler": {
    "enabled": true,
    "profile_step": 1,
    "module_depth": -1,
    "top_modules": 1,
    "detailed": true,
    "output_file": null
  }
}
```

41.7.3　吞吐量估计法

计算方法

$$\text{吞吐量} = \frac{\text{样本数}}{\text{秒}} \times \text{序列长度} \quad (\text{tokens/s/GPU})$$

$$\text{GPU 利用率} = \frac{\text{实际吞吐量}}{\text{参考吞吐量}} \times 100\%$$

实际案例

```
def estimate_throughput_utilization(actual_examples_per_sec, num_gpus,
                                    seq_length, reference_throughput=3300):
    """通过吞吐量估算GPU利用率"""

    # 计算实际吞吐量
    actual_throughput = actual_examples_per_sec * seq_length / num_gpus

    # 计算利用率
    utilization = (actual_throughput / reference_throughput) * 100
```

```
    print(f"实际吞吐量: {actual_throughput:.0f} tokens/s/GPU")
    print(f"参考吞吐量: {reference_throughput} tokens/s/GPU")
    print(f"GPU利用率: {utilization:.2f}%")

    return utilization

# 示例计算
实际数据: 3 example/s, 4卡, 序列长度2048
实际吞吐量 = 3 × 2048 / 4 = 1536 tokens/s/GPU
参考数据: LLaMA论文报告3300 tokens/s/GPU
利用率 = 1536 / 3300 × 100% = 46.54%
```

41.7.4 PyTorch Profiler 分析法

完整分析流程

```
def setup_pytorch_profiler():
    """设置PyTorch Profiler进行详细性能分析"""

    profiler = torch.profiler.profile(
        activities=[
            torch.profiler.ProfilerActivity.CPU,
            torch.profiler.ProfilerActivity.CUDA,
        ],
        schedule=torch.profiler.schedule(
            wait=1,
            warmup=1,
            active=3,
            repeat=1
        ),
        on_trace_ready=torch.profiler.tensorboard_trace_handler('./logs')
            ,
        record_shapes=True,
        profile_memory=True,
        with_stack=True
    )

    return profiler

```

```
def analyze_profiler_results(profiler_output):
    """分析Profiler结果"""

    # 分析关键指标
    key_metrics = profiler_output.key_metrics()

    # Tensor Core利用率
    tensor_core_utilization = key_metrics.get('tensor_core_utilization',
       0)

    # 内核时间分析
    kernel_time = key_metrics.get('kernel_time', {})
    compute_time = kernel_time.get('compute', 0)
    memory_time = kernel_time.get('memory', 0)
    communication_time = kernel_time.get('communication', 0)

    # 计算总体利用率
    total_time = compute_time + memory_time + communication_time
    effective_utilization = (compute_time / total_time) * 100 if
       total_time > 0 else 0

    print(f"Tensor Core利用率: {tensor_core_utilization:.1f}%")
    print(f"计算时间占比: {effective_utilization:.1f}%")
    print(f"内存操作时间: {memory_time/total_time*100:.1f}%")
    print(f"通信时间: {communication_time/total_time*100:.1f}%")

    return effective_utilization
```

41.8　系统诊断与性能优化

41.8.1　硬件诊断工具

网络性能诊断

```
# 查看多机训练网络速度
iftop  # 实时网络流量监控

# 查看具体网络接口统计
nethogs  # 按进程显示网络使用情况
```

GPU 拓扑分析

```
# 查看GPU间互联拓扑
nvidia-smi topo -m

# 输出示例:
#         GPU0    GPU1    GPU2    GPU3
# GPU0    X       NV1     NV2     NV2
# GPU1    NV1     X       NV2     NV2
# GPU2    NV2     NV2     X       NV1
# GPU3    NV2     NV2     NV1     X
```

详细硬件信息

```
# 查看详细GPU信息
cd /usr/local/cuda/samples/1_Utilities/deviceQuery
make
./deviceQuery

# 查看DeepSpeed环境配置
ds_report
```

41.8.2　性能瓶颈识别

通信瓶颈分析

在 PCIe 版本的 GPU 上使用 DeepSpeed Zero3 时，常见通信瓶颈：

表 41.6: 不同互联方式的通信性能对比

互联方式	带宽	AllGather 时间	对训练的影响
PCIe 4.0	32 GB/s	较长	通信成为主要瓶颈
NVLink	300 GB/s	较短	计算成为主要瓶颈
NVSwitch	600 GB/s	很短	接近理想性能

Tensor Core 利用率优化

```
def optimize_tensor_core_utilization(model, dataloader):
    """优化Tensor Core利用率策略"""
```

```
    optimization_strategies = {
        'batch_size': '调整批大小使Tensor Core饱和',
        'sequence_length': '使用适合的序列长度',
        'operator_fusion': '启用算子融合减少内核启动开销',
        'precision': '使用TF32/FP16等适合精度',
        'gradient_accumulation': '合适的梯度累积步数'
    }

    # 自动批大小调整
    optimal_batch_size = find_optimal_batch_size(model, dataloader)

    # 精度选择
    if supports_tf32():
        torch.backends.cuda.matmul.allow_tf32 = True
        torch.backends.cudnn.allow_tf32 = True

    return optimal_batch_size
```

41.9 实践建议与总结

41.9.1 显存优化最佳实践

多层次优化策略

1. **算法层面**：使用 LoRA、Adapter 等参数高效方法
2. **数值精度**：合理使用混合精度训练
3. **系统优化**：激活检查点、梯度累积等技术
4. **硬件利用**：优化数据加载和计算流水线

配置建议

41.9.2 性能监控体系

持续监控指标

- **GPU 利用率**：通过多种方法交叉验证
- **显存使用**：实时监控各组件显存占用
- **通信开销**：分析 AllReduce、AllGather 等操作
- **I/O 性能**：数据加载和预处理效率

表 41.7: 不同规模模型的硬件配置建议

模型规模	最小配置	推荐配置	理想配置
7B 模型	2×A100-40G	4×A100-80G	8×A100-80G
13B 模型	4×A100-80G	8×A100-80G	16×A100-80G
65B 模型	8×A100-80G	16×A100-80G	32×A100-80G
175B 模型	16×A100-80G	32×A100-80G	64×A100-80G

自动化优化框架

```
class AutoPerformanceOptimizer:
    """自动性能优化框架"""

    def __init__(self, model, train_loader):
        self.model = model
        self.train_loader = train_loader
        self.metrics_history = []

    def continuous_optimization(self):
        """持续性能优化循环"""
        while True:
            # 1. 收集性能指标
            metrics = self.collect_performance_metrics()
            self.metrics_history.append(metrics)

            # 2. 分析瓶颈
            bottlenecks = self.identify_bottlenecks(metrics)

            # 3. 应用优化
            if bottlenecks:
                self.apply_optimizations(bottlenecks)

            # 4. 等待下一轮
            time.sleep(300)  # 5分钟间隔

    def identify_bottlenecks(self, metrics):
        """识别性能瓶颈"""
        bottlenecks = []

```

```
        if metrics['gpu_utilization'] < 0.5:
            bottlenecks.append('low_gpu_utilization')

        if metrics['memory_usage'] > 0.9:
            bottlenecks.append('high_memory_pressure')

        if metrics['communication_ratio'] > 0.3:
            bottlenecks.append('communication_bound')

        return bottlenecks
```

41.10　总结

大语言模型的显存管理和性能优化是一个系统工程，需要从算法设计、系统配置到硬件利用多个层面进行综合考虑。通过准确的显存需求估算、合理的硬件配置选择以及持续的性能监控优化。

第四十二章　大语言模型显存优化策略技术详解

42.1　引言：显存优化的重要性与挑战

42.1.1　显存瓶颈的根源

随着大语言模型（LLMs）参数规模的指数级增长，从数十亿参数到数万亿参数，显存需求已成为模型训练和推理的主要瓶颈。显存限制直接影响着模型的批量大小、训练稳定性和最终性能。

42.1.2　显存优化的核心目标

- **内存效率**：在有限显存条件下最大化模型规模和训练效率
- **训练稳定性**：保持梯度信号的稳定性和收敛性
- **硬件利用率**：充分利用现有计算资源
- **成本控制**：降低对高端硬件的依赖

42.2　梯度累积（Gradient Accumulation）优化策略

42.2.1　技术原理与背景

传统梯度更新的问题

在标准的小批量随机梯度下降（Mini-batch SGD）中，每个批次数据独立计算梯度并立即更新模型参数：

```
for inputs, labels in data_loader:
    inputs = inputs.to(device)
    labels = labels.to(device)

    with torch.set_grad_enabled(True):
        # 前向传播
```

```
        preds = model(inputs)
        loss = criterion(preds, labels)

        # 反向传播
        loss.backward()

        # 参数更新
        optimizer.step()
        optimizer.zero_grad()
```

主要限制：

- **批量大小受限**：受限于 GPU 显存容量
- **梯度方差大**：小批量导致梯度信号不稳定
- **更新频率高**：频繁的参数更新影响收敛

梯度累积的核心思想

梯度累积通过将多个小批量数据的梯度累积起来，然后一次性更新模型参数，实现”虚拟大批量”训练：

$$\text{累积梯度} = \sum_{i=1}^{k} \nabla_\theta \mathcal{L}_i(\theta)$$

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{k} \sum_{i=1}^{k} \nabla_\theta \mathcal{L}_i(\theta_t)$$

42.2.2　实现机制与流程

优化后的梯度更新流程

```
gradient_accumulation_steps = 4  # 累积步数

for batch_idx, (inputs, labels) in enumerate(data_loader):
    inputs = inputs.to(device)
    labels = labels.to(device)

    with torch.set_grad_enabled(True):
        # 前向传播
        preds = model(inputs)
        loss = criterion(preds, labels)

        # 梯度归一化（平均梯度）
        loss /= gradient_accumulation_steps
```

```
14
15          # 反向传播（累积梯度）
16          loss.backward()
17
18          # 条件参数更新
19          if ((batch_idx + 1) % gradient_accumulation_steps == 0) or \
20             ((batch_idx + 1) == len(data_loader)):
21              # 参数更新
22              optimizer.step()
23              optimizer.zero_grad()
```

关键技术要点

- **梯度归一化**：每个小批量的损失除以累积步数，确保梯度尺度一致
- **累积控制**：通过模运算控制更新时机
- **梯度清零**：更新后及时清零累积梯度

42.2.3　优势分析与实践考量

主要优势

表 42.1: 梯度累积技术的优势

优势维度	技术原理	实际效果
内存效率	虚拟大批量训练	显存占用降低 k 倍
训练稳定性	减少梯度方差	收敛速度提升 20-30%
参数控制	灵活调整更新频率	适应不同硬件配置

实践考量因素

- **累积步数选择**：通常 4-32 步，平衡内存和训练速度
- **学习率调整**：可能需要相应调整学习率
- **优化算法兼容性 **：对不同优化器的影响各异
- **批归一化层 **：需要特殊处理以保持统计量准确

潜在问题与解决方案

```
1 def gradient_accumulation_optimization(model, optimizer, dataloader,
```

```
                                        accumulation_steps=4, initial_lr=1e
                                            -4):
    """梯度累积优化实现"""

    # 学习率调整（可选）
    adjusted_lr = initial_lr * (accumulation_steps ** 0.5)
    for param_group in optimizer.param_groups:
        param_group['lr'] = adjusted_lr

    # 训练循环
    for epoch in range(num_epochs):
        optimizer.zero_grad()

        for batch_idx, (inputs, labels) in enumerate(dataloader):
            # 前向传播和损失计算
            outputs = model(inputs)
            loss = criterion(outputs, labels) / accumulation_steps

            # 反向传播
            loss.backward()

            # 条件更新
            if (batch_idx + 1) % accumulation_steps == 0 or \
               (batch_idx + 1) == len(dataloader):
                optimizer.step()
                optimizer.zero_grad()

                # 打印训练信息
                if (batch_idx + 1) % accumulation_steps == 0:
                    print(f"Epoch: {epoch}, Batch: {batch_idx}, "
                          f"Loss: {loss.item() * accumulation_steps:.4f}"
                              )
```

42.3　梯度检查点（Gradient Checkpointing）优化策略

42.3.1　技术原理与背景

反向传播的内存挑战

在深度神经网络训练中，反向传播需要存储前向传播的所有中间激活值，以计算梯度：

$$\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial h_L} \cdot \prod_{j=i}^{L-1} \frac{\partial h_{j+1}}{\partial h_j} \cdot \frac{\partial h_j}{\partial W_i}$$

内存瓶颈：

- 激活存储 **：需要保存所有层的中间结果
- 深度依赖 **：网络越深，内存需求呈线性增长
- 批量放大 **：大批量训练加剧内存压力

梯度检查点的核心思想

通过将计算图分段，在前向传播时只保存关键检查点，反向传播时重新计算中间结果：

$$\text{检查点策略 选择性存储} \Rightarrow \text{按需重新计算}$$

42.3.2　实现机制与流程

技术实现原理

1. 计算图分段 **：将网络划分为多个段
2. 检查点选择 **：在段边界保存激活值
3. 延迟计算 **：反向传播时重新计算非检查点段
4. 内存-计算权衡 **：用计算时间换取内存空间

PyTorch 实现示例

```
from torch.utils.checkpoint import checkpoint

def custom_forward(*inputs):
    # 定义需要检查点的前向传播函数
    x = layer1(inputs)
    x = layer2(x)
    x = layer3(x)
    return x

# 在训练循环中使用
```

```
outputs = checkpoint(custom_forward, inputs)
loss = criterion(outputs, labels)
loss.backward()
```

42.3.3　优势分析与实践考量

主要优势

表 42.2: 梯度检查点技术的优势

优势维度	技术原理	实际效果
内存优化	减少激活存储	显存占用降低 30-70%
模型规模	支持更深网络	可训练 100+ 层模型
批量能力	允许大批量	提升训练稳定性

实践考量因素

- 计算开销 **：额外的前向传播计算
- 段划分策略 **：影响内存和计算平衡
- 硬件适配 **：不同 GPU 架构的优化
- 调试难度 **：增加了调试复杂性

性能权衡分析

```
def gradient_checkpointing_analysis(model, dataloader,
                                    checkpoint_segments=4):
    """梯度检查点性能分析"""

    # 原始内存使用
    original_memory = measure_memory_usage(model, dataloader)

    # 启用梯度检查点
    model.enable_checkpointing(segments=checkpoint_segments)
    checkpoint_memory = measure_memory_usage(model, dataloader)

    # 计算内存节省
    memory_saving = (original_memory - checkpoint_memory) /
       original_memory * 100
```

```
14
15      # 计算计算开销
16      original_time = measure_training_time(model, dataloader)
17      checkpoint_time = measure_training_time(model, dataloader,
            use_checkpointing=True)
18      time_overhead = (checkpoint_time - original_time) / original_time *
            100
19
20      print(f"内存节省: {memory_saving:.1f}%")
21      print(f"计算开销: {time_overhead:.1f}%")
22      print(f"最优段数: {find_optimal_segments(model, dataloader)}")
23
24      return memory_saving, time_overhead
```

42.4　综合优化策略与实践指南

42.4.1　优化策略组合应用

多层次优化框架

1. **第一层：梯度累积 **：解决批量大小限制**
2. **第二层：梯度检查点 **：解决内存容量限制**
3. **第三层：混合精度 **：进一步降低内存需求**
4. **第四层：激活检查点 **：优化中间结果存储**

配置建议矩阵

表 42.3: 显存优化策略配置建议

模型规模	梯度累积步数	梯度检查点段数	推荐显存配置
7B 模型	4-8 步	2-4 段	24-32GB GPU
13B 模型	8-16 步	3-6 段	48-64GB GPU
65B 模型	16-32 步	4-8 段	8×80GB GPU 集群
175B 模型	32-64 步	6-12 段	16×80GB GPU 集群

42.4.2　实践实施指南

实施步骤与流程

```python
class MemoryOptimizedTrainer:
    """内存优化训练器"""

    def __init__(self, model, optimizer, dataloader, config):
        self.model = model
        self.optimizer = optimizer
        self.dataloader = dataloader
        self.config = config

        # 初始化优化策略
        self.setup_optimizations()

    def setup_optimizations(self):
        """设置优化策略"""
        # 梯度累积配置
        self.gradient_accumulation_steps = self.config.get('
            accumulation_steps', 4)

        # 梯度检查点配置
        if self.config.get('use_checkpointing', False):
            self.enable_gradient_checkpointing()

    def enable_gradient_checkpointing(self):
        """启用梯度检查点"""
        # 实现检查点逻辑
        self.checkpoint_segments = self.config.get('checkpoint_segments',
            4)
        self.model = apply_gradient_checkpointing(self.model, self.
            checkpoint_segments)

    def train(self, num_epochs):
        """训练循环"""
        for epoch in range(num_epochs):
            self.run_epoch(epoch)

    def run_epoch(self, epoch):
        """运行单个epoch"""
        optimizer.zero_grad()

```

```
        for batch_idx, (inputs, labels) in enumerate(self.dataloader):
            # 前向传播
            outputs = self.model(inputs)
            loss = self.criterion(outputs, labels)

            # 梯度归一化
            loss = loss / self.gradient_accumulation_steps

            # 反向传播
            loss.backward()

            # 条件更新
            if (batch_idx + 1) % self.gradient_accumulation_steps == 0 or \
               (batch_idx + 1) == len(self.dataloader):
                self.optimizer.step()
                optimizer.zero_grad()

                # 打印训练信息
                self.log_training_info(epoch, batch_idx, loss)

    def log_training_info(self, epoch, batch_idx, loss):
        """记录训练信息"""
        if (batch_idx + 1) % self.gradient_accumulation_steps == 0:
            actual_loss = loss.item() * self.gradient_accumulation_steps
            print(f"Epoch: {epoch}, Batch: {batch_idx}, "
                  f"Loss: {actual_loss:.4f}, "
                  f"Accumulation: {self.gradient_accumulation_steps}
                     steps")
```

42.5　总结与展望

42.5.1　技术总结

显存优化是大规模语言模型训练的关键技术，梯度累积和梯度检查点提供了有效的解决方案。梯度累积通过虚拟大批量训练提高内存效率，梯度检查点通过延迟计算减少内存占用。

42.5.2　未来发展方向

- 智能优化策略：自动选择最优优化组合
- 硬件协同优化：针对特定 GPU 架构优化
- 分布式扩展：跨节点显存优化
- 自动化调优：基于强化学习的参数优化
- 统一优化框架：集成多种优化技术

随着大模型技术的不断发展，显存优化技术将继续演进，为构建更大、更强的大语言模型提供坚实的技术基础。

第四十三章　大语言模型分布式训练技术全景解析

43.1　引言：大模型训练的挑战与需求

43.1.1　大模型训练的核心挑战

随着大语言模型参数规模从数十亿发展到数万亿，传统单机训练面临严峻挑战：

显存效率问题

- **模型参数存储**：175B 参数的 GPT-3 模型仅参数就需要 700GB 显存（FP16 精度）
- **训练状态存储**：参数 + 梯度 + 优化器状态总计需要 2.8TB 显存
- **硬件限制**：即使最大显存的 GPU 也无法容纳超大模型

计算效率问题

- **训练时间**：单张 A100 训练 175B 模型需要约 288 年
- **数据规模**：训练数据量达到 TB 级别
- **计算复杂度**：模型规模增长带来计算量指数级增加

43.1.2　分布式训练的必要性

分布式训练通过将计算和存储任务分配到多个设备上，解决了单机训练的瓶颈：

$$\text{总训练时间} = \frac{\text{单机训练时间}}{\text{设备数量}} \times \text{并行效率}$$

43.2　分布式通信基础

43.2.1　点对点通信（Point-to-Point Communication）

基本概念

点对点通信涉及两个进程间的直接数据交换，一个进程发送数据，另一个进程接收数据。

技术特点

表 43.1: 点对点通信特性分析

特性	优势	局限性
通信模式	一对一直接通信	扩展性有限
延迟	低延迟，直接传输	大规模集群效率低
复杂度	实现简单	大规模管理复杂
适用场景	小规模集群、节点间通信	不适合全局数据同步

43.2.2　集体通信（Collective Communication）

基本概念

集体通信涉及多个进程间的协同数据交换，支持一对多、多对一、多对多等多种通信模式。

常见操作类型

- **Broadcast**：一个进程向所有进程发送数据
- **Reduce**：所有进程向一个进程归约数据
- **All-Reduce**：所有进程参与归约并获取结果
- **Scatter**：一个进程向所有进程分发数据
- **Gather**：所有进程向一个进程收集数据
- **All-to-All**：所有进程间全交换数据

性能分析

表 43.2: 集体通信性能特征

操作类型	通信复杂度	带宽需求	典型应用
Broadcast	$O(\log P)$	中等	参数初始化
Reduce	$O(\log P)$	中等	梯度聚合
All-Reduce	$O(\log P)$	高	分布式优化
All-to-All	$O(P)$	极高	张量并行

43.3　数据并行（Data Parallelism）

43.3.1　技术原理

基本思想

将训练数据集分割成多个子集，每个计算设备使用完整的模型副本处理不同的数据子集，通过梯度同步保证模型一致性。

数学表达

设共有 P 个设备，数据集 D 被划分为 P 个子集 $D_1, D_2, \ldots, D_P$，每个设备计算本地梯度：

$$g_i = \nabla_\theta L(\theta; D_i)$$

全局梯度通过平均得到：

$$g = \frac{1}{P} \sum_{i=1}^{P} g_i$$

43.3.2　实现机制

一致性保证

1. **初始一致性**：所有设备从相同的初始化参数开始
2. **梯度同步**：每个训练步骤后同步所有设备的梯度
3. **参数更新**：使用同步后的梯度统一更新参数

关键技术

```
class DataParallelTrainer:
    """数据并行训练器实现"""

    def __init__(self, model, optimizer, device_ids):
        self.model = model
        self.optimizer = optimizer
        self.device_ids = device_ids
        self.models = [model.to(device) for device in device_ids]

    def train_step(self, data_loader):
        """数据并行训练步骤"""
        # 数据分片
        data_shards = self.split_data(data_loader)

```

```
        # 各设备并行前向传播
        gradients = []
        for i, device in enumerate(self.device_ids):
            data = data_shards[i].to(device)
            model = self.models[i]

            # 前向传播
            output = model(data)
            loss = criterion(output, data.labels)

            # 反向传播
            loss.backward()
            gradients.append([param.grad for param in model.parameters()
                ])

        # 梯度同步（All-Reduce）
        synced_gradients = self.all_reduce_gradients(gradients)

        # 参数更新
        self.optimizer.step()
        self.optimizer.zero_grad()

    def all_reduce_gradients(self, gradients):
        """梯度全局归约"""
        # 使用All-Reduce操作同步所有设备的梯度
        for param_idx in range(len(gradients[0])):
            grad_list = [grads[param_idx] for grads in gradients]
            # 执行All-Reduce操作
            averaged_grad = torch.mean(torch.stack(grad_list), dim=0)
            # 将平均梯度设置到所有设备
            for grads in gradients:
                grads[param_idx] = averaged_grad
        return gradients
```

43.3.3　性能优化技术

梯度分桶（Gradient Bucketing）

- **动机：**集体通信在大张量上效率更高
- **实现：**将小梯度分组为更大的通信桶

- **效果**：减少通信次数，提高带宽利用率

计算通信重叠

- **流水线化**：在计算当前桶梯度时通信前一个桶
- **效果**：隐藏通信延迟，提高设备利用率

梯度累积

- **策略**：多次前向传播后执行一次梯度同步
- **效果**：减少通信频率，等效增大批次大小

43.4　流水线并行（Pipeline Parallelism）

43.4.1　技术原理

层间划分策略

将模型的不同层分配到不同的计算设备上，每个设备负责模型的一个连续片段，数据像流水线一样在设备间流动。

计算流程

1. 设备 1 计算第 1-3 层，将激活值发送到设备 2
2. 设备 2 计算第 4-6 层，将激活值发送到设备 3
3. 依次类推，完成前向传播
4. 反向传播按相反方向进行梯度计算和参数更新

43.4.2　技术变体与优化

GPipe 方案

- **同步流水线**：等所有微批次处理完后统一更新
- **显存效率**：需要存储所有微批次的中间激活
- **气泡问题**：设备间存在空闲等待时间

PipeDream 方案

- **异步流水线**：允许不同微批次重叠执行
- **显存优化**：1F1B（一前向一反向）调度策略
- **效率提升**：减少气泡时间，提高设备利用率

43.4.3　显存效率对比

表 43.3: 不同流水线并行方案的显存效率

方案	显存需求	计算效率	实现复杂度
朴素流水线	高（存储所有激活）	低（大气泡）	简单
GPipe	中（微批次优化）	中	中等
PipeDream	低（1F1B 调度）	高	复杂

43.5　张量并行（Tensor Parallelism）

43.5.1　技术原理

层内划分策略

将单个层内的计算划分到多个设备上，每个设备负责该层计算的一部分，通过通信协作完成完整计算。

矩阵乘法的并行化

对于矩阵乘法 $Y = XA$，可以采用行并行或列并行两种策略：

43.5.2　行并行（Row Parallelism）

划分策略

将权重矩阵 A 按行分块，输入矩阵 X 按列分块：

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, \quad X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$$

$$Y = XA = \begin{bmatrix} X_1 & X_2 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = X_1A_1 + X_2A_2$$

计算流程

1. 设备 1 计算 $Y_1 = X_1A_1$
2. 设备 2 计算 $Y_2 = X_2A_2$
3. 通过 All-Reduce 操作求和：$Y = Y_1 + Y_2$

43.5.3　列并行（Column Parallelism）

划分策略

将权重矩阵 A 按列分块，输入矩阵 X 保持不变：

$$A = \begin{bmatrix} A_1 & A_2 \end{bmatrix}$$

$$Y = XA = X \begin{bmatrix} A_1 & A_2 \end{bmatrix} = \begin{bmatrix} XA_1 & XA_2 \end{bmatrix} = \begin{bmatrix} Y_1 & Y_2 \end{bmatrix}$$

计算流程

1. 设备 1 计算 $Y_1 = XA_1$
2. 设备 2 计算 $Y_2 = XA_2$
3. 通过 All-Gather 操作拼接结果：$Y = [Y_1, Y_2]$

43.6　并行策略对比与 3D 并行

43.6.1　三种并行策略对比分析

显存效率对比

表 43.4: 三种并行策略的显存效率对比

策略	参数存储	梯度存储	优化器状态	总显存需求
数据并行 (DP)	每设备完整副本	每设备完整梯度	每设备完整状态	$4 \times$ 模型大小
流水线并行 (PP)	分片存储	分片存储	分片存储	$\approx \frac{1}{P} \times$ 模型大小
张量并行 (TP)	分片存储	分片存储	分片存储	$\approx \frac{1}{P} \times$ 模型大小

通信效率对比

表 43.5: 三种并行策略的通信特性

策略	通信模式	通信量	对网络要求
数据并行	All-Reduce 梯度	O(模型大小)	高带宽，低延迟
流水线并行	点对点通信激活值	O(激活值大小)	中等带宽，低延迟
张量并行	All-Reduce/All-Gather	O(层大小)	高带宽，极低延迟

43.6.2　3D 并行架构

组合策略

3D 并行将数据并行、流水线并行和张量并行三种策略有机结合，形成多维并行架构：

$$\text{总设备数} = \text{DP} \times \text{PP} \times \text{TP}$$

典型配置示例

- **张量并行 (TP)**：4 路，节点内利用 NVLink 高速互联
- **流水线并行 (PP)**：4 路，跨节点流水线
- **数据并行 (DP)**：2 路，模型副本间数据并行
- **总设备数**：$4 \times 4 \times 2 = 32$ 个 workers

通信层次优化

```
def setup_3d_parallelism(world_size, dp_degree, pp_degree, tp_degree):
    """设置3D并行拓扑"""
    assert world_size == dp_degree * pp_degree * tp_degree

    # 创建通信组
    # 张量并行组（节点内高速通信）
    tp_groups = create_tensor_parallel_groups(tp_degree)

    # 流水线并行组（跨节点流水线）
    pp_groups = create_pipeline_parallel_groups(pp_degree)

    # 数据并行组（模型副本间梯度同步）
    dp_groups = create_data_parallel_groups(dp_degree)

    return {
        'tp_groups': tp_groups,
        'pp_groups': pp_groups,
        'dp_groups': dp_groups
    }

def create_tensor_parallel_groups(tp_degree):
    """创建张量并行通信组"""
    groups = []
    # 每个张量并行组包含tp_degree个设备
```

```
    # 这些设备应该在同一节点内，通过NVLink高速互联
    for i in range(0, world_size, tp_degree):
        group = list(range(i, i + tp_degree))
        groups.append(group)
    return groups
```

43.7　实践指南：并行策略选择

43.7.1　硬件配置考量

单 GPU 场景

- **显存充足**：直接使用单卡训练
- **显存不足**：使用 CPU Offload 技术
- **推荐配置**：nB 模型需要 20nGB 以上显存进行微调

单节点多卡场景

表 43.6: 单节点多卡配置策略

显存情况	推荐策略	优势	注意事项
显存充足	DDP 或 ZeRO 数据并行	实现简单，效率高	需要足够显存
显存不足有 NVLink	张量并行 (TP)	显存利用率高	需要高速互联
显存不足无 NVLink	流水线并行 (PP)	对网络要求低	存在流水线气泡

多节点多卡场景

- **高速网络**：3D 并行、ZeRO 系列
- **普通网络**：DP+PP+TP 组合，避免高频通信
- **网络优化**：万兆网推荐使用 PP 为主，减少通信量

43.7.2　框架选择指南

主流技术栈对比

实际项目选择

```
def select_training_strategy(model_size, hardware_config):
    """根据模型规模和硬件配置选择训练策略"""
```

表 43.7: 分布式训练框架技术栈对比

技术栈	硬件平台	软件生态	适用场景	社区活跃度
TPU+XLA+TensorFlow	Google TPU	谷歌生态	大规模预训练	中等
GPU+PyTorch+DeepSpeed	NVIDIA GPU	开源社区	通用训练	极高
GPU+PyTorch+ColossalAI	NVIDIA GPU	学术研究	创新并行策略	高

```

    gpu_memory = hardware_config['gpu_memory']  # 单卡显存(GB)
    num_gpus = hardware_config['num_gpus']      # GPU数量
    has_nvlink = hardware_config['has_nvlink']  # 是否有NVLink
    network_bandwidth = hardware_config['network_bandwidth']  # 网络带宽

    # 估算显存需求
    memory_required = estimate_memory_requirements(model_size)

    if num_gpus == 1:
        # 单卡训练
        if gpu_memory >= memory_required:
            return "Single GPU Training"
        else:
            return "CPU Offload + Single GPU"

    elif num_gpus <= 8:  # 单节点多卡
        if gpu_memory >= memory_required:
            return "DDP Data Parallelism"
        elif has_nvlink:
            return "Tensor Parallelism + DDP"
        else:
            return "Pipeline Parallelism + DDP"

    else:  # 多节点多卡
        if network_bandwidth >= 100:  # 高速网络(100Gb/s+)
            return "3D Parallelism (DP+PP+TP)"
        else:  # 普通网络
            return "ZeRO-3 + Pipeline Parallelism"
```

43.8　性能优化与问题解决

43.8.1　推理性能分析

硬件性能对比

通过实际测试对比不同硬件平台的推理性能：

表 43.8: A800 与 V100 推理性能对比

硬件	答案长度	平均耗时 (秒)	吞吐量 (字/秒)	性能差异
A800	100 字	1.0	100	基准
V100	100 字	1.4	71.4	慢 40%
A800	500 字	5.0	100	基准
V100	500 字	7.0	71.4	慢 40%

性能优化建议

- **批次优化**：调整批次大小平衡吞吐量和延迟
- **精度选择**：FP16/INT8 推理加速
- **内核优化**：使用 TensorRT 等推理优化引擎
- **内存优化**：激活值重计算等技术

43.8.2　常见问题与解决方案

多机训练通信问题

```
# NCCL环境配置
export NCCL_IB_DISABLE=1
export NCCL_DEBUG=INFO
export NCCL_SOCKET_IFNAME=eth0
export NCCL_P2P_DISABLE=1
```

训练效率优化

- **通信瓶颈**：优化 All-Reduce 操作，使用梯度分桶
- **计算瓶颈**：启用计算通信重叠，使用混合精度
- **内存瓶颈**：使用激活检查点，梯度累积
- **I/O 瓶颈**：使用数据预取，优化数据加载

DeepSpeed 配置优化

```
{
    "train_batch_size": 32,
    "gradient_accumulation_steps": 4,
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": 1e-5
        }
    },
    "fp16": {
        "enabled": true
    },
    "zero_optimization": {
        "stage": 2,
        "allgather_partitions": true,
        "allgather_bucket_size": 2e8,
        "overlap_comm": true,
        "reduce_scatter": true
    }
}
```

43.9　总结与展望

43.9.1　技术总结

分布式训练是大语言模型发展的关键技术支撑，通过数据并行、流水线并行和张量并行的有机组合，实现了超大规模模型的高效训练。

43.9.2　未来发展趋势

自动化并行

- **自动策略选择**：基于模型结构和硬件配置自动选择最优并行策略
- **动态调优**：训练过程中动态调整并行参数
- **智能调度**：基于强化学习的资源调度优化

软硬件协同优化

- **专用硬件**：针对大模型训练的专用加速器
- **通信优化**：更高效的集合通信算法
- **存储优化**：分层存储架构支持超大模型

算法创新

- **高效优化器**：降低优化器状态的内存占用
- **稀疏训练**：利用模型稀疏性减少计算量
- **增量训练**：支持模型参数的增量更新

随着大模型技术的不断发展，分布式训练技术将继续演进，为更大规模、更高效的模型训练提供坚实的技术基础。

第四十四章　分布式训练技术完整解析

44.1　引言：大模型训练的分布式挑战

根据文档内容，ChatGPT 等大语言模型取得惊艳效果的关键要素按重要性排序为：

1. 愿意烧钱，且接受” 烧钱不等于好模型” 的现实
2. 高质量的训练语料
3. 高效的分布式训练框架和充沛优质的硬件资源
4. 算法的迭代创新

分布式训练的总体目标包括：

- 能训练更大的模型：理想状况下，模型的大小和 GPU 的数量成线性关系
- 能更快地训练模型：理想状况下，训练的速度和 GPU 的数量成线性关系

44.2　流水线并行 (Pipeline Parallelism) 完整解析

44.2.1　基本概念与优化目标

流水线并行主要解决单卡装不下的大模型训练问题。通过把模型分割成不同的层，每一层都放到一块 GPU 上。优化目标包括：

- 训练更大的模型：模型大小与 GPU 数量成线性关系
- 训练速度提升：训练速度与 GPU 数量成线性关系

44.2.2　朴素模型并行的问题分析

朴素模型并行存在两个主要问题：

GPU 利用度不够

设 K 块 GPU，单块 GPU 上做一次 forward 和 backward 的时间为 $t_{fb} = (t_f + t_b)$，则：

- 灰色长方形整体面积：$K \times Kt_{fb}$
- 实际计算面积：Kt_{fb}
- 阴影部分面积：$(K-1)Kt_{fb}$
- 阴影部分占比：$\frac{K-1}{K}$

当 K 越大时，GPU 空置比例接近 1，资源浪费严重。

中间结果占据大量内存

假设模型有 L 层，每层宽度为 d，每块 GPU 不考虑参数存储的额外空间复杂度为：

$$O(N \times \frac{L}{K} \times d)$$

44.2.3　Gpipe 解决方案

Micro-batch 切分

将 mini-batch 划分为 M 个 micro-batch，流水线并行下 bubble 时间复杂度为：

$$O\left(\frac{K-1}{K+M-1}\right)$$

实验证明当 $M \geq 4K$ 时，bubble 影响可忽略不计。

重计算技术 (Re-materialization)

采用时间换空间策略，几乎不存中间结果，等到 backward 时重新计算 forward。每块 GPU 峰值内存占用为：

$$O\left(N + \frac{N}{M} \times \frac{L}{K} \times d\right)$$

Batch Normalization 处理

训练时计算 micro-batch 内的均值和方差，同时持续追踪全部 mini-batch 的移动平均和方差，供测试阶段使用。

44.2.4　实验效果验证

模型规模扩展能力

在 Transformer 模型上基本实现线性增长，但从 32 卡到 128 卡时，模型从 21.08B 参数增加到 82.9B 参数。AmoebaNet 因切割不均未能完全实现线性增长。

训练速度优化

关闭 NVlinks 时，Gpipe 仍能实现训练速度随 GPU 数量增加。开启 NVlinks 后，M=32 时表现最佳，Transformer 基本实现线性关系。

时间消耗分布

约 2/3 时间用于计算，1/3 时间用于重计算，bubble 时间可忽略不计。

44.3　数据并行技术完整解析

44.3.1　nn.DataParallel 原理

基本架构

- 若干块计算 GPU 和 1 块梯度收集 GPU
- 每块计算 GPU 拷贝完整模型参数
- 数据均匀分给不同计算 GPU
- 梯度聚合后更新模型参数

参数服务器框架

计算 GPU 称为 Worker，梯度聚合 GPU 称为 Server。可选择 Worker 同时作为 Server，减少通信量。

44.3.2　常见问题与解决方案

多 GPU 计算时间问题

使用 watch -n 1 nvidia-smi 监控 GPU 占用率，如均低于 50% 则多 GPU 可能更慢。

模型保存与加载

```
# 保存
torch.save(net.module.state_dict(), './networks/multiGPU.h5')
# 加载
new_net.load_state_dict(torch.load("./networks/multiGPU.h5"))
```

第一块卡显存占用更多

因 output_device 默认为 device_ids[0]，loss 计算在第一块 GPU 相加。

loss 警告问题

使用 size_average=False, reduce=True 参数，或通过 gather 方式求 loss 平均。

device_ids 0 被占用问题

```
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "2, 3"
```

44.3.3　参数更新流程

1. DataLoader 通过多个 worker 读到主进程内存
2. 通过 tensor 的 split 语义切分 batch 数据
3. 各 GPU 完成前向计算，输出 gather 到主 GPU 计算 loss
4. loss scatter 到各 GPU 进行 BP 计算梯度
5. 梯度 reduce 到主 GPU 进行参数更新
6. 参数 broadcast 到各 GPU 完成同步

44.4　DistributedDataParallel 深度解析

44.4.1　Ring-AllReduce 算法原理

基本概念

假设 4 块 GPU，每块数据被切分成 4 份，目标让每块 GPU 拥有完整聚合数据。

Reduce-Scatter 阶段

- 定义网络拓扑关系，每个 GPU 只与相邻 GPU 通信
- 每次发送对应位置数据进行累加
- 经过 3 次更新后，每块 GPU 有一块数据拥有完整聚合

All-Gather 阶段

- 按照相邻 GPU 对应位置进行通信
- 对应位置数据直接替换而非相加
- 经过 3 轮迭代后，每块 GPU 汇总完整数据

44.4.2　DDP 实现流程

初始化配置

```
torch.distributed.init_process_group(backend="nccl")
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset)
model = nn.parallel.DistributedDataParallel(model,
    device_ids=[args.local_rank],
    output_device=args.local_rank,
    find_unused_parameters=True)
```

启动命令

```
python -m torch.distributed.run --nnodes=1 --nproc_per_node=2 --node_rank
   =0 --master_port=6005 train.py
```

44.4.3　参数更新机制

初始化同步

1. rank 0 进程将网络初始化参数 broadcast 到其他进程
2. 确保每个进程网络初始化值一致

训练过程

- 每个进程读取各自训练数据，DistributedSampler 确保数据不同
- 前向和 loss 计算在每个进程独立完成
- 反向传播通过 all-reduce 将梯度 reduce 到每个进程
- 模型参数始终保持一致，无需额外同步

44.4.4　与 DataParallel 对比

表 44.1: DataParallel 与 DistributedDataParallel 对比

DataParallel	**DistributedDataParallel**
单进程控制多线程实现	多进程实现，避免 Python GIL 限制
梯度汇总到 GPU0，反向传播更新参数，再广播参数	各进程梯度汇总平均，rank=0 广播到所有进程
全程维护一个 optimizer，主卡进行参数更新	各进程模型参数始终保持一致
通信数据量较大	传输数据量更少，速度更快
仅支持单机多卡	支持多机多卡场景

44.5　混合精度训练 (AMP) 完整解析

44.5.1　基本概念与原理

自动混合精度定义

PyTorch 1.6 引入的自动混合精度训练，包含两种精度 Tensor：

- torch.FloatTensor：32 位浮点型
- torch.HalfTensor：16 位浮点型

混合精度优势

- 存储小、计算快、更好利用 Tensor Core
- 减少显存占用，增加 batch size
- 训练速度更快

混合精度劣势

- 数值范围小，容易 Overflow/Underflow
- 舍入误差导致微小梯度信息丢失

44.5.2　解决方案

梯度 scale

通过 torch.cuda.amp.GradScaler 放大 loss 值防止梯度 underflow，更新权重时 unscale 回去。

精度回退

框架自动决定何时使用半精度，何时回退到全精度。

44.5.3　自动转换操作

在 AMP 上下文中，以下操作自动转为半精度：

```
matmul, addbmm, addmm, addmv, addr, baddbmm, bmm,
chain_matmul, conv1d, conv2d, conv3d, conv_transpose1d,
conv_transpose2d, conv_transpose3d, linear, matmul, mm, mv, prelu
```

44.5.4　动态损失缩放机制

基本原理

- 损失乘以大数字 (如 1024) 放大梯度
- 计算梯度后除以 1024 得到准确值
- 动态选择损失标度：溢出时跳过更新，损失标度减半；连续稳定时翻倍

44.5.5　PyTorch AMP 使用

基础用法

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()
with autocast():
    output = model(input)
    loss = loss_fn(output, target)
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

错误处理

出现 RuntimeError 时，可在 tensor 上调用.float() 进行类型匹配。

44.6　DeepSpeed 框架完整解析

44.6.1　基本概念

分布式基础概念

- 节点编号 (node_rank)：系统每个节点的唯一标识符
- 全局进程编号 (rank)：整个系统每个进程的唯一标识符
- 局部进程编号 (local_rank)：单个节点内每个进程的标识符
- 全局总进程数 (word_size)：系统中所有进程总数
- 主节点 (master_ip+master_port)：协调工作的关键节点

通信策略支持

- mpi：CPU 集群分布式训练
- gloo：CPU 和 GPU 分布式训练
- nccl：GPU 专用通信库

44.6.2　ZeRO 优化技术

显存内容分类

- 模型状态：参数、梯度、优化器状态（占 75%）
- 剩余状态：激活值、临时缓冲区、内存碎片

ZeRO 阶段划分

- Stage 1：优化器状态分片 (Pos)
- Stage 2：梯度分片 (Pos+g)
- Stage 3：参数分片 (Pos+g+p)

内存减少效果

- Stage 1：内存减少 4 倍
- Stage 2：内存减少 8 倍
- Stage 3：内存减少与数据并行度 Nd 成线性关系

44.6.3　DeepSpeed 使用实践

安装配置

```
pip install deepspeed==0.8.1
sudo apt-get install openmpi-bin libopenmpi-dev
pip install mpi4py
```

模型初始化

```
model_engine, optimizer, _, _ = deepspeed.initialize(
    config=deepspeed_config,
    model=model,
    model_parameters=model.parameters()
)
```

训练执行

```
deepspeed test.py --deepspeed_config config.json
```

44.6.4　优化器与调度器

优化器选择

- 支持 Adam、AdamW、OneBitAdam、Lamb 等
- 启用 offload_optimizer 时可使用非 DeepSpeed 优化器
- 默认使用 AdamW，参数来自命令行设置

调度器配置

表 44.2: 优化器与调度器兼容性

组合	HF Scheduler	DS Scheduler
HF Optimizer	Yes	Yes
DS Optimizer	No	Yes

44.7　Accelerate 库完整解析

44.7.1　设计理念与优势

主要优势

- 简化 PyTorch 训练和推断开发过程
- 提高性能，支持分布式训练、混合精度训练
- 自动调参、数据加载优化、模型优化
- 集成 PyTorch Lightning 和 TorchElastic

加速策略

- Pipeline 并行：模型拆分不同部分并行训练
- 数据并行：数据拆分不同部分并行训练
- 加速器自动检测利用

44.7.2　使用实践

基础配置

```
from accelerate import Accelerator

accelerator = Accelerator()
model, optimizer, dataloader = accelerator.prepare(
    model, optimizer, dataloader
)
```

训练流程

```
for batch in dataloader:
    optimizer.zero_grad()
```

```
    output = model(batch)
    loss = loss_fn(output)
    accelerator.backward(loss)
    optimizer.step()
```

启动方式

```
# 方式一
accelerate launch multi-gpu-accelerate-cls.py

# 方式二
python -m torch.distributed.launch --nproc_per_node 2 --use_env multi-gpu
   -accelerate-cls.py
```

44.8　实践指南与故障排查

44.8.1　分布式训练代码规范

模型加载时机

1. get model
2. model.to(device)
3. use DP

数据加载时机

在训练 iter 过程中 model 进行 forward 之前完成 batch 数据加载到设备。

batch_size 与 GPU 数量对应

- batch_size 大于或等于 GPU 数量
- 保证 batch_size 能被 GPU 数整除
- 不能整除时需自定义 chunk_size 功能

44.8.2　常见问题解决方案

ModuleNotFoundError

```
# 注释掉
from torch._six import string_classes
```

```
# 添加
int_classes = int
string_classes = str
```

单卡使用 DeepSpeed

通过 ZeRO-offload 将数据 offload 到 CPU，降低显存需求。

ZeRO 配置选择

- ZeRO-2：中等规模模型
- ZeRO-3：大规模模型，速度较慢但内存效率高
- 根据模型规模和硬件条件选择合适 stage

44.8.3　性能调优策略

显存优化

1. 减小 batch size 或使用梯度累积
2. 启用梯度检查点技术
3. 使用 ZeRO Offload 到 CPU 或 NVMe
4. 混合精度训练优化

通信优化

- 调整 AllReduce 桶大小
- 启用通信计算重叠
- 使用更快通信后端（NCCL）
- 优化网络拓扑结构

44.9　技术选型与发展趋势

44.9.1　技术对比分析

技术方案	适用场景	内存效率	实现复杂度	通信开销
DataParallel	单机多卡、模型可单卡存放	低	低	中
DistributedDataParallel	多机多卡、中等规模模型	中	中	中

Pipeline Parallelism	超大模型、内存受限	高	高	低
ZeRO-Stage1	中等模型、优化器状态优化	中高	中	中
ZeRO-Stage2	大模型、梯度优化	高	中高	中高
ZeRO-Stage3	超大模型、完整优化	极高	高	高
AMP	计算密集型、速度优先	中	低	低
DeepSpeed	超大规模、综合优化	极高	高	可配置
Accelerate	快速原型、简化开发	中	低	中

44.9.2　选型决策流程

1. **评估需求**：模型规模、训练速度、硬件条件
2. **技术匹配**：根据需求选择合适的技术组合
3. **渐进实施**：从简单方案开始，逐步优化
4. **性能监控**：持续监控训练效果，调整参数

44.9.3　未来发展趋势

技术融合

- 多种并行策略深度融合
- 硬件软件协同优化
- 自动化调参和资源分配

易用性提升

- 框架接口进一步简化
- 自动化部署和运维
- 跨平台兼容性增强

44.10　总结

本章完整解析了分布式训练的各类技术方案，从基础的流水线并行、数据并行，到高级的 DeepSpeed、ZeRO 优化，以及便捷的 Accelerate 库。每种技术都有其适用场景和优缺点，实际应用中需要根据具体需求进行选择和组合。

分布式训练技术的发展为大模型训练提供了坚实的技术基础，随着硬件技术的进步和算法的不断创新，未来将支持更大规模、更高效的模型训练，推动人工智能技术持续向前发展。