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：

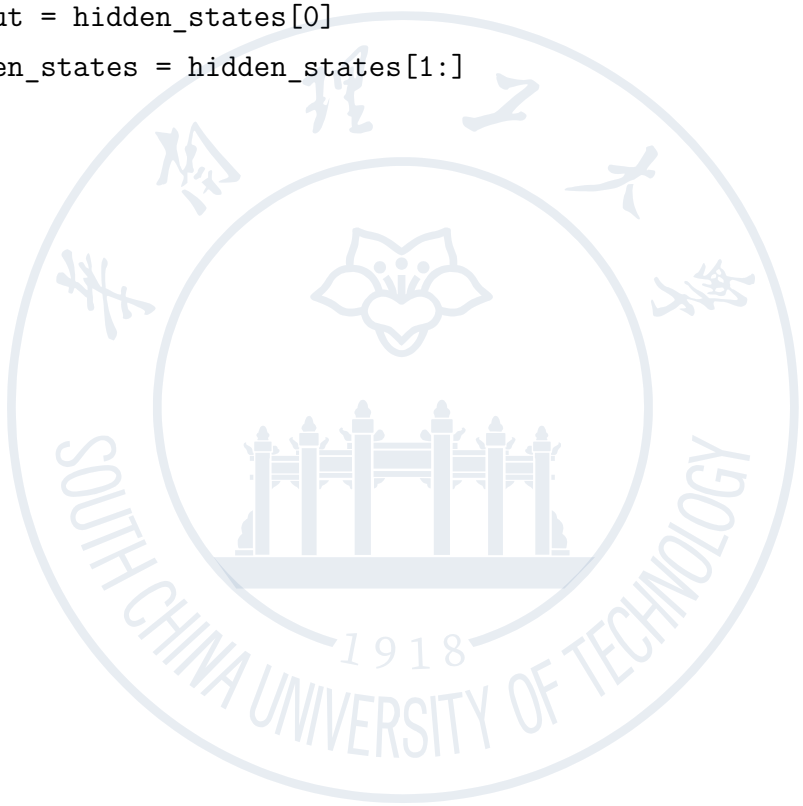
- **last_hidden_state**: shape 是 (batch_size, sequence_length, hidden_size), hidden_size=768, 它是模型最后一层输出的隐藏状态
- **pooler_output**: shape 是 (batch_size, hidden_size),这是序列的第一个 token(classification token) 的最后一层的隐藏状态，它是由线性层和 Tanh 激活函数进一步处理的，这个输出不是对输入的语义内容的一个很好的总结，对于整个输入序列的隐藏状态序列的平均化或池化通常更好。
- **hidden_states**: 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

```python
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