

# 大模型最全八股和答案（上）

原创 jackaduma 包包算法笔记 2023-11-07 18:23 发表于北京

收录于合集

#大模型 16 #八股文 3 #面试 10



包包算法笔记

包大人的算法，程序，机器学习，职场，理财闲谈。

113篇原创内容

公众号

之前写过大模型面试八股/大模型面试八股含答案，后续有网友整理了更清晰的版本，推荐仓库：[https://github.com/jackaduma/awesome\\_LLMs\\_interview\\_notes](https://github.com/jackaduma/awesome_LLMs_interview_notes)，点击阅读原文链接可直接访问。

## 目录

- [x] 大模型（LLMs）基础面
  - [x] 1. 目前 主流的开源模型体系 有哪些？
  - [x] 2. prefix LM 和 causal LM 区别是什么？
  - [x] 3. 涌现能力是啥原因？
  - [x] 4. 大模型LLM的架构介绍？
- [x] 大模型（LLMs）进阶面
  - [x] 1. llama 输入句子长度理论上可以无限长吗？
  - [x] 1. 什么是 LLMs 复读机问题？
  - [x] 2. 为什么会出现 LLMs 复读机问题？
  - [x] 3. 如何缓解 LLMs 复读机问题？
  - [x] 1. LLMs 复读机问题
  - [x] 2. llama 系列问题
  - [x] 3. 什么情况用Bert模型，什么情况用LLaMA、ChatGLM类大模型，咋选？
  - [x] 4. 各个专业领域是否需要各自的大模型来服务？
  - [x] 5. 如何让大模型处理更长的文本？
- [x] 大模型（LLMs）微调面
  - [x] 1. 如果想要在某个模型基础上做全参数微调，究竟需要多少显存？
  - [x] 2. 为什么SFT之后感觉LLM傻了？
  - [x] 3. SFT 指令微调数据 如何构建？
  - [x] 4. 领域模型Continue PreTrain 数据选取？
  - [x] 5. 领域数据训练后，通用能力往往会有所下降，如何缓解模型遗忘通用能力？
  - [x] 6. 领域模型Continue PreTrain ，如何 让模型在预训练过程中就学习到更多的知识？
  - [x] 7. 进行SFT操作的时候，基座模型选用Chat还是Base？
  - [x] 8. 领域模型微调 指令&数据输入格式 要求？
  - [x] 9. 领域模型微调 领域评测集 构建？
  - [x] 10. 领域模型词表扩增是不是有必要的？
  - [x] 11. 如何训练自己的大模型？
  - [x] 12. 训练中文大模型有啥经验？
  - [x] 13. 指令微调的好处？

- [x] 14. 预训练和微调哪个阶段注入知识的？
- [x] 15. 想让模型学习某个领域或行业的知识，是应该预训练还是应该微调？
- [x] 16. 多轮对话任务如何微调模型？
- [x] 17. 微调后的模型出现能力劣化，灾难性遗忘是怎么回事？
- [x] 18. 微调模型需要哪些操作？
- [x] 19. 大模型LLM继续访问 在 取消
- [x] 20. 预训练和SFT操作有什么不同
- [x] 21. 样本量规模增大，训练出现OOM错
- [x] 22. 大模型LLM进行SFT 如何对样本进行优化？
- [x] 23. 模型参数迭代实验
- [x] 大模型（LLMs）langchain面
  - [x] 1. 基于LLM+向量库的文档对话 基础面
  - [x] 2. 基于LLM+向量库的文档对话 优化面
  - [ ] 3. 基于LLM+向量库的文档对话 工程示例面
  - [x] 1. LLMs 存在模型幻觉问题，请问如何处理？
  - [x] 2. 基于LLM+向量库的文档对话 思路是怎样？
  - [x] 3. 基于LLM+向量库的文档对话 核心技术是什么？
  - [x] 4. 基于LLM+向量库的文档对话 prompt 模板 如何构建？
  - [x] 1. 痛点1：文档切分粒度不好把控，既担心噪声太多又担心语义信息丢失
  - [x] 2. 痛点2：在基于垂直领域 表现不佳
  - [x] 3. 痛点3： langchain 内置 问答分句效果不佳问题
  - [x] 4. 痛点4：如何 尽可能召回与query相关的Document 问题
  - [x] 5. 痛点5：如何让LLM基于query和context得到高质量的response
  - [ ] 1. 避坑记录
  - [ ] 2. 本地知识库问答系统（Langchain-chatGLM）
  - [x] 1. 什么是 LangChain？
  - [x] 2. LangChain 包含哪些 核心概念？
  - [x] 3. 什么是 LangChain Agent？
  - [x] 4. 如何使用 LangChain ？
  - [x] 5. LangChain 支持哪些功能？
  - [x] 6. 什么是 LangChain model？
  - [x] 7. LangChain 包含哪些特点？
  - [x] 8. LangChain 如何使用？
  - [ ] 9. LangChain 存在哪些问题及方法方案？
  - [x] 10. LangChain 替代方案？
  - [x] 1. LangChain 中 Components and Chains 是什么？
  - [x] 2. LangChain 中 Prompt Templates and Values 是什么？
  - [x] 3. LangChain 中 Example Selectors 是什么？
  - [x] 4. LangChain 中 Output Parsers 是什么？
  - [x] 5. LangChain 中 Indexes and Retrievers 是什么？
  - [x] 6. LangChain 中 Chat Message History 是什么？
  - [x] 7. LangChain 中 Agents and Toolkits 是什么？
  - [x] 1. LangChain 如何调用 LLMs 生成回复？
  - [x] 2. LangChain 如何修改 提示模板？
  - [x] 3. LangChain 如何链接多个组件处理一个特定的下游任务？
  - [x] 4. LangChain 如何Embedding & vector store？
  - [x] 1. LangChain 低效的令牌使用问题

- [ ] 2. LangChain 文档的问题
  - [ ] 3. LangChain 太多概念容易混淆，过多的“辅助”函数问题
  - [ ] 4. LangChain 行为不一致并且隐藏细节问题
  - [x] 5. LangChain 缺乏标准的可互操作数据类型问题
  - [x] 大模型（LLMs）
  - [x] 基于LLM+向量
- 继续访问      取消
- [x] 大模型（LLMs）参数高效微调(PEFT) 面
    - [x] 一、LoRA篇
    - [x] 二、QLoRA篇
    - [x] 三、AdaLoRA篇
    - [x] 四、LoRA权重是否可以合入原模型？
    - [ ] 五、ChatGLM-6B LoRA后的权重多大？
    - [x] 六、LoRA 微调优点是什么？
    - [x] 七、LoRA微调方法为啥能加速训练？
    - [x] 八、如何在已有LoRA模型上继续训练？
    - [x] 1.1 什么是 LoRA？
    - [x] 1.2 LoRA 的思路是什么？
    - [x] 1.3 LoRA 的特点是什么？
    - [x] 2.1 QLoRA 的思路是怎么样的？
    - [x] 2.2 QLoRA 的特点是什么？
    - [x] 3.1 AdaLoRA 的思路是怎么样的？
    - [x] 一、为什么需要 提示学习（Prompting）？
    - [x] 二、什么是 提示学习（Prompting）？
    - [x] 三、提示学习（Prompting）有什么优点？
    - [x] 四、提示学习（Prompting）有哪些方法，能不能稍微介绍一下它们间？
    - [x] 4.4.1 为什么需要 P-tuning v2？
    - [x] 4.4.2 P-tuning v2 思路是什么？
    - [x] 4.4.3 P-tuning v2 优点是什么？
    - [x] 4.4.4 P-tuning v2 缺点是什么？
    - [x] 4.3.1 为什么需要 P-tuning？
    - [x] 4.3.2 P-tuning 思路是什么？
    - [x] 4.3.3 P-tuning 优点是什么？
    - [x] 4.3.4 P-tuning 缺点是什么？
    - [x] 4.2.1 为什么需要 指示微调（Prompt-tuning）？
    - [x] 4.2.2 指示微调（Prompt-tuning）思路是什么？
    - [x] 4.2.3 指示微调（Prompt-tuning）优点是什么？
    - [x] 4.2.4 指示微调（Prompt-tuning）缺点是什么？
    - [x] 4.2.5 指示微调（Prompt-tuning）与 Prefix-tuning 区别 是什么？
    - [x] 4.2.6 指示微调（Prompt-tuning）与 fine-tuning 区别 是什么？
    - [x] 4.1.1 为什么需要 前缀微调（Prefix-tuning）？
    - [x] 4.1.2 前缀微调（Prefix-tuning）思路是什么？
    - [x] 4.1.3 前缀微调（Prefix-tuning）的优点是什么？
    - [x] 4.1.4 前缀微调（Prefix-tuning）的缺点是什么？
    - [x] 4.1 前缀微调（Prefix-tuning）篇
    - [x] 4.2 指示微调（Prompt-tuning）篇
    - [x] 4.3 P-tuning 篇
    - [x] 4.4 P-tuning v2 篇

- [x] 一、为什么 需要 适配器微调（Adapter-tuning）？
- [x] 二、适配器微调（Adapter-tuning）思路？
- [x] 三、适配器微调（Adapter-tuning）特点是什么？
- [x] 四、AdapterFusion 思路 是什么？
- [x] 五、AdapterDrc
- [x] 六、AdapterDrc 继续访问 取消
- [x] 七、MAM Adapter 思路 是什么？
- [x] 八、MAM Adapter 特点 是什么？
- [x] 微调方法是啥？如何微调？
- [x] 为什么需要 PEFT？
- [x] 介绍一下 PEFT？
- [x] PEFT 有什么优点？
- [x] 微调方法批处理大小模式GPU显存速度？
- [x] Peft 和 全量微调区别？
- [x] 多种不同的高效微调方法对比
- [x] 当前高效微调技术存在的一些问题
- [x] 高效微调技术最佳实践
- [x] PEFT 存在问题？
- [x] 能不能总结一下各种参数高效微调方法？
- [x] 大模型（LLMs）参数高效微调(PEFT) 面
- [x] 适配器微调（Adapter-tuning）篇
- [x] 提示学习（Prompting）
- [x] LoRA 系列篇
- [x] 大模型（LLMs）推理面
  - [x] 1. 为什么大模型推理时显存涨的那么多还一直占着？
  - [x] 2. 大模型在gpu和cpu上推理速度如何？
  - [x] 3. 推理速度上，int8和fp16比起来怎么样？
  - [x] 4. 大模型有推理能力吗？
  - [x] 5. 大模型生成时的参数怎么设置？
  - [x] 6. 有哪些省内存的大语言模型训练/微调/推理方法？
  - [x] 7. 如何让大模型输出合规化
  - [x] 8. 应用模式变更
- [x] 大模型（LLMs）评测面
  - [x] 大模型怎么评测？
  - [x] 大模型的honest原则是如何实现的？
  - [x] 模型如何判断回答的知识是训练过的已知的知识，怎么训练这种能力？
- [x] 大模型（LLMs）强化学习面
  - [x] 奖励模型需要和基础模型一致吗？
  - [x] RLHF 在实践中存在哪些不足？
  - [x] 如何解决 人工产生的偏好数据集成本较高，很难量产问题？
  - [x] 如何解决三个阶段的训练（SFT->RM->PPO）过程较长，更新迭代较慢问题？
  - [x] 如何解决 PPO 的训练过程同时存在4个模型（2训练，2推理），对计算资源的要求较高 问题？
- [x] 大模型（LLMs）软硬件配置面
- [x] 大模型（LLMs）训练集面

- [x] SFT（有监督微调）的数据集格式？
- [x] RM（奖励模型）的数据格式？
- [x] PPO（强化学习）的数据格式？
- [x] 找数据集哪里找？
- [x] 微调需要多少条数据？
- [x] 有哪些大模型的继续访问 取消
- [x] 进行领域大模型预训练应用哪些数据集比较好？
- [ ] 大模型（LLMs）显存问题面
- [ ] 大模型（LLMs）分布式训练面
- [x] 大模型（LLMs）agent 面
  - [x] 如何给LLM注入领域知识？
  - [x] 如果想要快速体验各种模型，该怎么办？
- [ ] Token及模型参数准备篇
  - [x] 预训练数据 Token 重复 是否影响 模型性能？
  - [ ] SFT需要训练Token数？
- [ ] LLMs 位置编码篇
  - [x] 6.1 ALiBi (Attention with Linear Biases) 思路是什么？
  - [x] 6.2 ALiBi (Attention with Linear Biases) 的偏置矩阵是什么？有什么作用？
  - [x] 6.3 ALiBi (Attention with Linear Biases) 有什么优点？
  - [ ] 6.4 ALiBi (Attention with Linear Biases) 被哪些 LLMs 应用？
  - [x] 5.1 什么是 长度外推问题？
  - [x] 5.2 长度外推问题 的 解决方法 有哪些？
  - [x] 4.1 旋转位置编码 RoPE 思路是什么？
  - [ ] 4.2 推导一下 旋转位置编码 RoPE ？
  - [x] 4.3 旋转位置编码 RoPE 有什么优点？
  - [ ] 4.4 旋转位置编码 RoPE 被哪些 LLMs 应用？
  - [x] 1 什么是位置编码？
  - [x] 2 什么是绝对位置编码？
  - [x] 3 什么是相对位置编码？
  - [ ] 4 旋转位置编码 RoPE篇
  - [ ] 5 长度外推问题篇
  - [ ] 6 ALiBi (Attention with Linear Biases)篇
- [ ] LLMs Tokenizer 篇
  - [x] Byte-Pair Encoding(BPE)篇
  - [x] WordPiece 篇
  - [x] SentencePiece 篇
  - [x] 对比篇
  - [x] 1 Byte-Pair Encoding(BPE) 如何构建词典？
  - [x] 1 WordPiece 与 BPE 异同点是什么？
  - [x] 简单介绍一下 SentencePiece 思路？
  - [x] 1 举例 介绍一下 不同 大模型LLMs 的分词方式？
  - [x] 2 介绍一下 不同 大模型LLMs 的分词方式 的区别？
  - [x] LLMs Tokenizer 篇

- [x] **Layer Normalization** 篇
  - [x] LLMs 各模型分别用了 哪种 Layer normalization?
  - [x] 1 LN 在 LLMs 中的不同位置 有什么区别么? 如果有, 能介绍一下区别么?
  - [x] Layer Norm 篇
  - [x] RMS Norm 篇 ( 继续访问 取消)
  - [x] Deep Norm 篇
  - [x] Deep Norm 有什么优点?
  - [x] Layer Norm 的计算公式写一下?
  - [x] RMS Norm 的计算公式写一下?
  - [x] RMS Norm 相比于 Layer Norm 有什么特点?
  - [x] Deep Norm 思路?
  - [x] 写一下 Deep Norm 代码实现?
  - [x] Layer normalization-方法篇
  - [x] Layer normalization-位置篇
  - [x] Layer normalization 对比篇

## 答案

### • 基础面

#### 1 目前 主流的开源模型体系 有哪些?

- [x] 1 介绍一下 FFN 块 计算公式?
- [x] 2 介绍一下 GeLU 计算公式?
- [x] 3 介绍一下 Swish 计算公式?
- [x] 4 介绍一下 使用 GLU 线性门控单元的 FFN 块 计算公式?
- [x] 5 介绍一下 使用 GeLU 的 GLU 块 计算公式?
- [x] 6 介绍一下 使用 Swish 的 GLU 块 计算公式?
- [ ] 各LLMs 都使用哪种激活函数?

目前主流的开源LLM（语言模型）模型体系包括以下几个：

1. GPT（Generative Pre-trained Transformer）系列：由OpenAI发布的一系列基于Transformer架构的语言模型，包括GPT、GPT-2、GPT-3等。GPT模型通过在大规模无标签文本上进行预训练，然后在特定任务上进行微调，具有很强的生成能力和语言理解能力。
2. BERT（Bidirectional Encoder Representations from Transformers）：由Google发布的一种基于Transformer架构的双向预训练语言模型。BERT模型通过在大规模无标签文本上进行预训练，然后在下游任务上进行微调，具有强大的语言理解能力和表征能力。
3. XLNet：由CMU和Google Brain发布的一种基于Transformer架构的自回归预训练语言模型。XLNet模型通过自回归方式预训练，可以建模全局依赖关系，具有更好的语言建模能力和生成能力。
4. RoBERTa：由Facebook发布的一种基于Transformer架构的预训练语言模型。RoBERTa模型在BERT的基础上进行了改进，通过更大规模的数据和更长的训练时间，取得了更好的性能。
5. T5（Text-to-Text Transfer Transformer）：由Google发布的一种基于Transformer架构的多任务预训练语言模型。T5模型通过在大规模数据集上进行预训练，可以用于多种自然语言处理任务，如文本分类、机器翻译、问答等。

这些模型在自然语言处理领域取得了显著的成果，并被广泛应用于各种任务和应用中。

#### 2 prefix LM 和 causal LM 区别是什么?

Prefix LM（前缀语言模型）和Causal LM（因果语言模型）是两种不同类型的语言模型，它们的区别在于生成文本的方式和训练目标。

- 1. Prefix LM：前缀语言模型是一种生成模型，它在生成每个词时都可以考虑之前的上下文信息。在生成时，前缀语言模型会根据给定的前缀（即部分文本序列）预测下一个可能的词。这种模型可以用于  或
- 2. Causal LM：因果语言模型是一种生成模型，它在生成每个词时只能根据之前的文本生成之前的文本。在训练时，因果语言模型的目标是预测下一个词的概率，给定之前的所有词作为上下文。这种模型可以用于文本生成、语言建模等任务。

总结来说，前缀语言模型可以根据给定的前缀生成后续的文本，而因果语言模型只能根据之前的文本生成后续的文本。它们的训练目标和生成方式略有不同，适用于不同的任务和应用场景。

3 涌现能力是啥原因？

大模型的涌现能力主要是由以下几个原因造成的：

- 1. 数据量的增加：随着互联网的发展和数字化信息的爆炸增长，可用于训练模型的数据量大大增加。更多的数据可以提供更丰富、更广泛的语言知识和语境，使得模型能够更好地理解和生成文本。
- 2. 计算能力的提升：随着计算硬件的发展，特别是图形处理器（GPU）和专用的AI芯片（如TPU）的出现，计算能力大幅提升。这使得训练更大、更复杂的模型成为可能，从而提高了模型的性能和涌现能力。
- 3. 模型架构的改进：近年来，一些新的模型架构被引入，如Transformer，它在处理序列数据上表现出色。这些新的架构通过引入自注意力机制等技术，使得模型能够更好地捕捉长距离的依赖关系和语言结构，提高了模型的表达能力和生成能力。
- 4. 预训练和微调的方法：预训练和微调是一种有效的训练策略，可以在大规模无标签数据上进行预训练，然后在特定任务上进行微调。这种方法可以使模型从大规模数据中学习到的更丰富的语言知识和语义理解，从而提高模型的涌现能力。

综上所述，大模型的涌现能力是由数据量的增加、计算能力的提升、模型架构的改进以及预训练和微调等因素共同作用的结果。这些因素的进步使得大模型能够更好地理解和生成文本，为自然语言处理领域带来了显著的进展。

4 大模型LLM的架构介绍？

LLM（Large Language Model，大型语言模型）是指基于大规模数据和参数量的语言模型。具体的架构可以有多种选择，以下是一种常见的大模型LLM的架构介绍：

- 1. Transformer架构：大模型LLM常使用Transformer架构，它是一种基于自注意力机制的序列模型。Transformer架构由多个编码器层和解码器层组成，每个层都包含多头自注意力机制和前馈神经网络。这种架构可以捕捉长距离的依赖关系和语言结构，适用于处理大规模语言数据。
- 2. 自注意力机制（Self-Attention）：自注意力机制是Transformer架构的核心组件之一。它允许模型在生成每个词时，根据输入序列中的其他词来计算该词的表示。自注意力机制能够动态地为每个词分配不同的权重，从而更好地捕捉上下文信息。
- 3. 多头注意力（Multi-Head Attention）：多头注意力是自注意力机制的一种扩展形式。它将自注意力机制应用多次，每次使用不同的权重矩阵进行计算，得到多个注意力头。多头注意力可以提供更丰富的上下文表示，增强模型的表达能力。
- 4. 前馈神经网络（Feed-Forward Network）：在Transformer架构中，每个注意力层后面都有一个前馈神经网络。前馈神经网络由两个全连接层组成，通过非线性激活函数（如ReLU）进行变换。它可以对注意力层输出的表示进行进一步的映射和调整。
- 5. 预训练和微调：大模型LLM通常采用预训练和微调的方法进行训练。预训练阶段使用大规模无标签数据，通过自监督学习等方法进行训练，使模型学习到丰富的语言知识。微调阶段

段使用有标签的特定任务数据，如文本生成、机器翻译等，通过有监督学习进行模型的微调和优化。

需要注意的是，大模型LLM的具体架构可能会因不同的研究和应用而有所不同。上述介绍的是一种常见的架构，但实际应用中可能会有有一些变体或改进。

继续访问

取消

进阶面

1 LLMs 复读机问题

i. 什么是 LLMs 复读机问题？

LLMs复读机问题指的是大型语言模型（LLMs）在生成文本时出现的一种现象，即模型倾向于无限地复制输入的文本或者以过度频繁的方式重复相同的句子或短语。这种现象使得模型的输出缺乏多样性和创造性，给用户带来了不好的体验。

复读机问题可能出现的原因包括：

- 1. 数据偏差：大型语言模型通常是通过预训练阶段使用大规模无标签数据进行训练的。如果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，模型在生成文本时可能会倾向于复制这些常见的模式。
- 2. 训练目标的限制：大型语言模型的训练通常是基于自监督学习的方法，通过预测下一个词或掩盖词来学习语言模型。这样的训练目标可能使得模型更倾向于生成与输入相似的文本，导致复读机问题的出现。
- 3. 缺乏多样性的训练数据：虽然大型语言模型可以处理大规模的数据，但如果训练数据中缺乏多样性的语言表达和语境，模型可能无法学习到足够的多样性和创造性，导致复读机问题的出现。

为了解决复读机问题，可以采取以下策略：

- 1. 多样性训练数据：在训练阶段，尽量使用多样性的语料库来训练模型，避免数据偏差和重复文本的问题。
- 2. 引入噪声：在生成文本时，可以引入一些随机性或噪声，例如通过采样不同的词或短语，或者引入随机的变换操作，以增加生成文本的多样性。
- 3. 温度参数调整：温度参数是用来控制生成文本的多样性的一个参数。通过调整温度参数的值，可以控制生成文本的独创性和多样性，从而减少复读机问题的出现。
- 4. 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，以提高生成文本的质量和多样性。

需要注意的是，复读机问题是大型语言模型面临的一个挑战，解决这个问题是一个复杂的任务，需要综合考虑数据、训练目标、模型架构和生成策略等多个因素。目前，研究人员和工程师们正在不断努力改进和优化大型语言模型，以提高其生成文本的多样性和创造性。

ii. 为什么会出现 LLMs 复读机问题？

出现LLMs复读机问题可能有以下几个原因：

- 1. 数据偏差：大型语言模型通常是通过预训练阶段使用大规模无标签数据进行训练的。如果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，模型在生成文本时可能会倾向于复制这些常见的模式。
- 2. 训练目标的限制：大型语言模型的训练通常是基于自监督学习的方法，通过预测下一个词或掩盖词来学习语言模型。这样的训练目标可能使得模型更倾向于生成与输入相似的文本，导致复读机问题的出现。
- 3. 缺乏多样性的训练数据：虽然大型语言模型可以处理大规模的数据，但如果训练数据中缺乏多样性的语言表达和语境，模型可能无法学习到足够的多样性和创造性，导致复读机问题的出现。



- 4. 模型结构和参数设置：大型语言模型的结构和参数设置也可能对复读机问题产生影响。例如，模型的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。

为了解决复读机问题，可以采取以下策略：

- 1. 多样性训练数据：在训练阶段，尽量使用多样性的语料库来训练模型，避免数据偏差和重复文本的问题。
- 2. 引入噪声：在生成文本时，可以引入一些随机性或噪声，例如通过采样不同的词或短语，或者引入随机的变换操作，以增加生成文本的多样性。
- 3. 温度参数调整：温度参数是用来控制生成文本的多样性的一个参数。通过调整温度参数的值，可以控制生成文本的独创性和多样性，从而减少复读机问题的出现。
- 4. 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，以提高生成文本的质量和多样性。

需要注意的是，复读机问题是大型语言模型面临的一个挑战，解决这个问题是一个复杂的任务，需要综合考虑数据、训练目标、模型架构和生成策略等多个因素。目前，研究人员和工程师们正在不断努力改进和优化大型语言模型，以提高其生成文本的多样性和创造性。

iii. 如何缓解 LLMs 复读机问题？

为了缓解LLMs复读机问题，可以尝试以下方法：

- 1. 多样性训练数据：在训练阶段，使用多样性的语料库来训练模型，避免数据偏差和重复文本的问题。这可以包括从不同领域、不同来源和不同风格的文本中获取数据。
- 2. 引入噪声：在生成文本时，引入一些随机性或噪声，例如通过采样不同的词或短语，或者引入随机的变换操作，以增加生成文本的多样性。这可以通过在生成过程中对模型的输出进行采样或添加随机性来实现。
- 3. 温度参数调整：温度参数是用来控制生成文本的多样性的一个参数。通过调整温度参数的值，可以控制生成文本的独创性和多样性。较高的温度值会增加随机性，从而减少复读机问题的出现。
- 4. Beam搜索调整：在生成文本时，可以调整Beam搜索算法的参数。Beam搜索是一种常用的生成策略，它在生成过程中维护了一个候选序列的集合。通过调整Beam大小和搜索宽度，可以控制生成文本的多样性和创造性。
- 5. 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，以提高生成文本的质量和多样性。可以使用文本相似度计算方法或规则来检测和去除重复的文本。
- 6. 人工干预和控制：对于关键任务或敏感场景，可以引入人工干预和控制机制，对生成的文本进行审查和筛选，确保生成结果的准确性和多样性。

需要注意的是，缓解LLMs复读机问题是一个复杂的任务，没有一种通用的解决方案。不同的方法可能适用于不同的场景和任务，需要根据具体情况进行选择和调整。此外，解决复读机问题还需要综合考虑数据、训练目标、模型架构和生成策略等多个因素，需要进一步的研究和实践来提高大型语言模型的生成文本多样性和创造性。

2 llama 系列问题

i. llama 输入句子长度理论上可以无限长吗？

理论上来说，LLMs（大型语言模型）可以处理任意长度的输入句子，但实际上存在一些限制和挑战。下面是一些相关的考虑因素：

- 1. 计算资源：生成长句子需要更多的计算资源，包括内存和计算时间。由于LLMs通常是基于神经网络的模型，计算长句子可能会导致内存不足或计算时间过长的问题。
- 2. 模型训练和推理：训练和推理长句子可能会面临一些挑战。在训练阶段，处理长句子可能会导致梯度消失或梯度爆炸的问题，影响模型的收敛性和训练效果。在推理阶段，生成长句子可能会增加模型的错误率和生成时间。

3. 上下文建模：LLMs是基于上下文建模的模型，长句子的上下文可能会更加复杂和深层。模型需要能够捕捉长句子中的语义和语法结构，以生成准确和连贯的文本。

尽管存在这些挑战，研究人员和工程师们已经在不断努力改进和优化LLMs，以处理更长的句子。例如，可以采用分块的方式处理长句子，将其分成多个较短的片段进行处理。此外，还可以通过增加计算资源、优化模型结构和参数配置以及使用更高效的推理算法来提高LLMs处理长句子的能力。

值得注意的是，实际应用中，长句子的处理可能还受到应用场景、任务需求和资源限制等因素的影响。因此，在使用LLMs处理长句子时，需要综合考虑这些因素，并根据具体情况进行选择和调整。

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

选择使用哪种大模型，如Bert、LLaMA或ChatGLM，取决于具体的应用场景和需求。下面是一些指导原则：

- 1. Bert模型：Bert是一种预训练的语言模型，适用于各种自然语言处理任务，如文本分类、命名实体识别、语义相似度计算等。如果你的任务是通用的文本处理任务，而不依赖于特定领域的知识或语言风格，Bert模型通常是一个不错的选择。Bert由一个Transformer编码器组成，更适合于NLU相关的任务。
- 2. LLaMA模型：LLaMA（Large Language Model Meta AI）包含从 7B 到 65B 的参数范围，训练使用多达14,000亿tokens语料，具有常识推理、问答、数学推理、代码生成、语言理解等能力。Bert由一个Transformer解码器组成。训练预料主要为以英语为主的拉丁语系，不包含中日韩文。所以适合于英文文本生成的任务。
- 3. ChatGLM模型：ChatGLM是一个面向对话生成的语言模型，适用于构建聊天机器人、智能客服等对话系统。如果你的应用场景需要模型能够生成连贯、流畅的对话回复，并且需要处理对话上下文、生成多轮对话等，ChatGLM模型可能是一个较好的选择。ChatGLM的架构为Prefix decoder，训练语料为中英双语，中英文比例为1:1。所以适合于中文和英文文本生成的任务。

在选择模型时，还需要考虑以下因素：

- 数据可用性：不同模型可能需要不同类型和规模的数据进行训练。确保你有足够的数据来训练和微调所选择的模型。
- 计算资源：大模型通常需要更多的计算资源和存储空间。确保你有足够的硬件资源来支持所选择的模型的训练和推理。
- 预训练和微调：大模型通常需要进行预训练和微调才能适应特定任务和领域。了解所选择模型的预训练和微调过程，并确保你有相应的数据和时间来完成这些步骤。

最佳选择取决于具体的应用需求和限制条件。在做出决策之前，建议先进行一些实验和评估，以确定哪种模型最适合你的应用场景。

4 各个专业领域是否需要各自的大模型来服务？

各个专业领域通常需要各自的大模型来服务，原因如下：

- 1. 领域特定知识：不同领域拥有各自特定的知识和术语，需要针对该领域进行训练的大模型才能更好地理解和处理相关文本。例如，在医学领域，需要训练具有医学知识的大模型，以更准确地理解和生成医学文本。
- 2. 语言风格和惯用语：各个领域通常有自己独特的语言风格和惯用语，这些特点对于模型的训练和生成都很重要。专门针对某个领域进行训练的大模型可以更好地掌握该领域的语言特点，生成更符合该领域要求的文本。
- 3. 领域需求的差异：不同领域对于文本处理的需求也有所差异。例如，金融领域可能更关注数字和统计数据的处理，而法律领域可能更关注法律条款和案例的解析。因此，为了更好地满足不同领域的需求，需要专门针对各个领域进行训练的大模型。

- 4. 数据稀缺性：某些领域的的数据可能相对较少，无法充分训练通用的大模型。针对特定领域进行训练的大模型可以更好地利用该领域的的数据，提高模型的性能和效果。

尽管需要各自的大模型来服务不同领域，但也可以共享一些通用的模型和技术。例如，通用的大模型可以用于处理通用的文本任务，而领域特定的模型可以在通用模型的基础上进行微调和定制，以适应特定领域的需要。这样可以满足领域需求的同时，减少模型的重复训练和资源消耗。

[继续访问](#) [取消](#)

5 如何让大模型处理更长的文本？

要让大模型处理更长的文本，可以考虑以下几个方法：

- 1. 分块处理：将长文本分割成较短的片段，然后逐个片段输入模型进行处理。这样可以避免长文本对模型内存和计算资源的压力。在处理分块文本时，可以使用重叠的方式，即将相邻片段的一部分重叠，以保持上下文的连贯性。
- 2. 层次建模：通过引入层次结构，将长文本划分为更小的单元。例如，可以将文本分为段落、句子或子句等层次，然后逐层输入模型进行处理。这样可以减少每个单元的长度，提高模型处理长文本的能力。
- 3. 部分生成：如果只需要模型生成文本的一部分，而不是整个文本，可以只输入部分文本作为上下文，然后让模型生成所需的部分。例如，输入前一部分文本，让模型生成后续的内容。
- 4. 注意力机制：注意力机制可以帮助模型关注输入中的重要部分，可以用于处理长文本时的上下文建模。通过引入注意力机制，模型可以更好地捕捉长文本中的关键信息。
- 5. 模型结构优化：通过优化模型结构和参数设置，可以提高模型处理长文本的能力。例如，可以增加模型的层数或参数量，以增加模型的表达能力。还可以使用更高效的模型架构，如Transformer等，以提高长文本的处理效率。

需要注意的是，处理长文本时还需考虑计算资源和时间的限制。较长的文本可能需要更多的内存和计算时间，因此在实际应用中需要根据具体情况进行权衡和调整。

微调面

- 1. 💡 如果想要在某个模型基础上做全参数微调，究竟需要多少显存？

要确定全参数微调所需的显存量，需要考虑以下几个因素：

综上所述，全参数微调所需的显存量取决于模型的大小、批量大小、训练数据的维度以及训练设备的显存限制。在进行全参数微调之前，建议先评估所需的显存量，并确保训练设备具备足够的显存来支持训练过程。

- a. 模型的大小：模型的大小是指模型参数的数量。通常，参数越多，模型的大小就越大。大型的预训练模型如Bert、GPT等通常有数亿到数十亿个参数，而较小的模型可能只有数百万到数千万个参数。模型的大小直接影响了所需的显存量。
- b. 批量大小：批量大小是指在每次训练迭代中一次性输入到模型中的样本数量。较大的批量大小可以提高训练的效率，但也需要更多的显存。通常，全参数微调时，较大的批量大小可以提供更好的性能。
- c. 训练数据的维度：训练数据的维度是指输入数据的形状。如果输入数据具有较高的维度，例如图像数据，那么所需的显存量可能会更大。对于文本数据，通常需要进行一些编码和嵌入操作，这也会增加显存的需求。
- d. 训练设备的显存限制：最后，需要考虑训练设备的显存限制。显卡的显存大小是一个硬性限制，超过显存限制可能导致训练失败或性能下降。确保所选择的模型和批量大小适应训练设备的显存大小。

- 2. 💡 为什么SFT之后感觉LLM傻了？

在进行Supervised Fine-Tuning（SFT）之后，有时可能会观察到基座模型（如语言模型）的性能下降或产生一些“傻”的行为。这可能是由于以下原因：

为了解决这些问题，可以尝试以下方法：

通过这些方法，可以尽量减少Supervised Fine-Tuning之后模型出现“傻”的情况，并提高模型在新任务上的表现。

- 收集更多的训练数据 以增加数据的多样性和覆盖范围
  - 仔细检查微调数据集 

继续访问

 的 

取消
  - 使用正则化技术（如权重衰减、dropout）来减少过拟合的风险。
  - 进行数据增强，通过对微调数据集进行一些变换或扩充来增加多样性。
  - 使用更复杂的模型架构或调整模型的超参数，以提高模型的性能和泛化能力。
- a. 数据偏移：SFT过程中使用的微调数据集可能与基座模型在预训练阶段接触到的数据分布有所不同。如果微调数据集与预训练数据集之间存在显著的差异，模型可能会在新任务上表现较差。这种数据偏移可能导致模型在新任务上出现错误的预测或不准确的输出。
  - b. 非典型标注：微调数据集的标注可能存在错误或不准的标签。这些错误的标签可能会对模型的性能产生负面影响，导致模型产生“傻”的行为。
  - c. 过拟合：如果微调数据集相对较小，或者模型的容量（参数数量）较大，模型可能会过拟合微调数据，导致在新的输入上表现不佳。过拟合可能导致模型过于依赖微调数据的特定样本，而无法泛化到更广泛的输入。
  - d. 缺乏多样性：微调数据集可能缺乏多样性，未能涵盖模型在新任务上可能遇到的各种输入情况。这可能导致模型在面对新的、与微调数据集不同的输入时出现困惑或错误的预测。

3. 📢 SFT 指令微调数据 如何构建？

构建Supervised Fine-Tuning（SFT）的微调数据需要以下步骤：

通过以上步骤，您可以构建适合Supervised Fine-Tuning的微调数据集，并使用该数据集对基座模型进行微调，以适应特定任务的需求。

- a. 收集原始数据：首先，您需要收集与目标任务相关的原始数据。这可以是对话数据、分类数据、生成任务数据等，具体取决于您的任务类型。确保数据集具有代表性和多样性，以提高模型的泛化能力。
- b. 标注数据：对原始数据进行标注，为每个样本提供正确的标签或目标输出。标签的类型取决于您的任务，可以是分类标签、生成文本、对话回复等。确保标注的准确性和一致性。
- c. 划分数据集：将标注数据划分为训练集、验证集和测试集。通常，大部分数据用于训练，一小部分用于验证模型的性能和调整超参数，最后一部分用于最终评估模型的泛化能力。
- d. 数据预处理：根据任务的要求，对数据进行预处理。这可能包括文本清洗、分词、去除停用词、词干化等处理步骤。确保数据格式和特征表示适合模型的输入要求。
- e. 格式转换：将数据转换为适合模型训练的格式。这可能涉及将数据转换为文本文件、JSON格式或其他适合模型输入的格式。
- f. 模型微调：使用转换后的数据对基座模型进行微调。根据任务的要求，选择适当的微调方法和超参数进行训练。这可以使用常见的深度学习框架（如PyTorch、TensorFlow）来实现。
- g. 模型评估：使用测试集对微调后的模型进行评估，计算模型在任务上的性能指标，如准确率、召回率、生成质量等。根据评估结果对模型进行进一步的优化和调整。

4. 📢 领域模型Continue PreTrain 数据选取？

在领域模型的Continue PreTrain过程中，数据选取是一个关键的步骤。以下是一些常见的数据选取方法：

在数据选取过程中，需要根据具体任务和需求进行适当的调整 and 定制。选择合适的数据可以提高模型在特定领域上的性能和泛化能力。

- a. 领域相关数据：首先，可以收集与目标领域相关的数据。这些数据可以是互联网上爬取的、来自特定领域的文档或者公司内部的数据等。这样的数据可以提供领域相关的语言和知识，有助于模型在特定领域上的表现。
- b. 领域专家标注：如果有领域专家可用，可以请他们对领域相关的数据进行标注。标注可以是分类、命名实体识别、关系抽取等任务，这样可以提供有监督的数据用于模型的训练。
- c. 伪标签：如果没有领域专家或有标注数据的时间或预算，可以使用一些自动化的方法生成伪标签。例如，可以使用预训练的模型对领域相关的数据进行预测，将预测结果作为伪标签，然后使用这些伪标签进行模型的训练。
- d. 数据平衡：在进行数据选取时，需要注意数据的平衡性。如果某个类别的数据样本较少，可以考虑使用数据增强技术或者对该类别进行过采样，以平衡各个类别的数据量。
- e. 数据质量控制：在进行数据选取时，需要对数据的质量进行控制。可以使用一些质量评估指标，如数据的准确性、一致性等，来筛选和过滤数据。
- f. 数据预处理：在进行数据选取之前，可能需要对数据进行一些预处理，如分词、去除停用词、标准化等，以准备好输入模型进行训练。

继续访问

取消

#### 5. 📌 领域数据训练后，通用能力往往会有所下降，如何缓解模型遗忘通用能力？

当使用领域数据进行训练后，模型往往会出现遗忘通用能力的问题。以下是一些缓解模型遗忘通用能力的方法：

综合使用上述方法，可以在一定程度上缓解模型遗忘通用能力的问题，使得模型既能够适应特定领域的任务，又能够保持一定的通用能力。

- a. 保留通用数据：在进行领域数据训练时，仍然需要保留一部分通用数据用于模型训练。这样可以确保模型仍然能够学习到通用的语言和知识，从而保持一定的通用能力。
- b. 增量学习：使用增量学习（Incremental Learning）的方法，将领域数据与通用数据逐步交替进行训练。这样可以在学习新领域的同时，保持对通用知识的记忆。
- c. 预训练和微调：在领域数据训练之前，可以使用大规模通用数据进行预训练，获得一个通用的基础模型。然后，在领域数据上进行微调，以适应特定领域的任务。这样可以在保留通用能力的同时，提升领域任务的性能。
- d. 强化学习：使用强化学习的方法，通过给模型设置奖励机制，鼓励模型在领域任务上表现好，同时保持一定的通用能力。
- e. 领域适应技术：使用领域适应技术，如领域自适应（Domain Adaptation）和领域对抗训练（Domain Adversarial Training），帮助模型在不同领域之间进行迁移学习，从而减少遗忘通用能力的问题。
- f. 数据重采样：在进行领域数据训练时，可以使用数据重采样的方法，使得模型在训练过程中能够更多地接触到通用数据，从而缓解遗忘通用能力的问题。

#### 6. 📌 领域模型Continue PreTrain，如何让模型在预训练过程中就学习到更多的知识？

在领域模型的Continue PreTrain过程中，可以采取一些策略来让模型在预训练过程中学习到更多的知识。以下是一些方法：

综合使用上述方法，可以让模型在预训练过程中学习到更多的知识和语言规律，提升其在领域任务上的性能。

- a. 多任务学习：在预训练过程中，可以引入多个任务，使得模型能够学习到更多的知识。这些任务可以是领域相关的任务，也可以是通用的语言理解任务。通过同时训练多个任务，模型可以学习到更多的语言规律和知识。
- b. 多领域数据：收集来自不同领域的数据，包括目标领域和其他相关领域的数据。将这些数据混合在一起进行预训练，可以使得模型在不同领域的知识都得到学习和融合。
- c. 大规模数据：使用更大规模的数据进行预训练，可以让模型接触到更多的语言和知识。可以从互联网上爬取大量的文本数据，或者利用公开的语料库进行预训练。
- d. 数据增强：在预训练过程中，可以采用数据增强的技术，如随机遮挡、词替换、句子重组等，来生成更多的训练样本。这样可以增加模型的训练数据量，使其能够学习到

更多的知识和语言规律。

- e. 自监督学习：引入自监督学习的方法，通过设计一些自动生成的标签或任务，让模型在无监督的情况下进行预训练。例如，可以设计一个掩码语言模型任务，让模型预测被掩码的词语。这样可以使模型在预训练过程中学习到更多的语言知识。

7. 进行SFT操作的时候

在进行Supervised Fine-tuning (SFT) 操作时，基座模型的选择也可以根据具体情况来决定。与之前的SFT操作不同，这次的目标是在特定的监督任务上进行微调，因此选择基座模型时需要考虑任务的性质和数据集的特点。

如果您的监督任务是对话生成相关的，比如生成对话回复或对话情感分类等，那么选择ChatGPT模型作为基座模型可能更合适。ChatGPT模型在对话生成任务上进行了专门的优化和训练，具有更好的对话交互能力。

然而，如果您的监督任务是单轮文本生成或非对话生成任务，那么选择Base GPT模型作为基座模型可能更合适。Base GPT模型在单轮文本生成和非对话生成任务上表现良好，可以提供更准确的文本生成能力。

总之，基座模型的选择应该根据监督任务的性质和数据集的特点进行权衡。如果任务是对话生成相关的，可以选择ChatGPT模型作为基座模型；如果任务是单轮文本生成或非对话生成，可以选择Base GPT模型作为基座模型。

8. 领域模型微调 指令&数据输入格式 要求？

领域模型微调是指使用预训练的通用语言模型（如BERT、GPT等）对特定领域的数据进行微调，以适应该领域的任务需求。以下是领域模型微调的指令和数据输入格式的要求：

指令：

数据输入格式要求：

根据具体的任务和模型要求，数据输入格式可能会有所不同。在进行领域模型微调之前，建议仔细阅读所使用模型的文档和示例代码，以了解其具体的数据输入格式要求。

- a. 输入数据应以文本形式提供，每个样本对应一行。
- b. 对于分类任务，每个样本应包含文本和标签，可以使用制表符或逗号将文本和标签分隔开。
- c. 对于生成任务，每个样本只需包含文本即可。
- d. 对于序列标注任务，每个样本应包含文本和对应的标签序列，可以使用制表符或逗号将文本和标签序列分隔开。
- e. 数据集应以常见的文件格式（如文本文件、CSV文件、JSON文件等）保存，并确保数据的格式与模型输入的要求一致。
- f. 定义任务：明确所需的任务类型，如文本分类、命名实体识别、情感分析等。
- g. 选择预训练模型：根据任务需求选择适合的预训练模型，如BERT、GPT等。
- h. 准备微调数据：收集和标注与领域任务相关的数据，确保数据集具有代表性和多样性。
- i. 数据预处理：根据任务的要求，对数据进行预处理，例如分词、去除停用词、词干化等。
- j. 划分数据集：将数据集划分为训练集、验证集和测试集，用于模型的训练、验证和评估。
- k. 模型微调：使用预训练模型和微调数据对模型进行微调，调整超参数并进行训练。
- l. 模型评估：使用测试集评估微调后的模型的性能，计算适当的评估指标，如准确率、召回率等。
- m. 模型应用：将微调后的模型应用于实际任务，在新的输入上进行预测或生成。

9. 领域模型微调 领域评测集 构建？

构建领域评测集的过程可以参考以下步骤：



重复以上步骤，不断优化模型，直到达到满意的评测结果为止。

需要注意的是，构建领域评测集是一个耗时且需要专业知识的过程。在进行领域模型微调之前，建议与领域专家合作，确保评测集的质量和有效性。此外，还可以参考相关研究论文和公开数据集，以获取更多关于领域评测集构建的指导和经验。

- a. 收集数据：首先需要 **继续访问** 关 **取消** 舌从互联网上爬取文本数据、使用已有的公开数据集或者通过与领域专家合作来获取数据。确保数据集具有代表性和多样性，能够涵盖领域中的各种情况和语境。
- b. 标注数据：对收集到的数据进行标注，以便用于评测模型的性能。标注可以根据任务类型来进行，如文本分类、命名实体识别、关系抽取等。标注过程可以由人工标注或者使用自动化工具进行，具体取决于数据集的规模和可行性。
- c. 划分数据集：将标注好的数据集划分为训练集、验证集和测试集。通常，训练集用于模型的训练，验证集用于调整超参数和模型选择，测试集用于最终评估模型的性能。划分数据集时要确保每个集合中的样本都具有代表性和多样性。
- d. 设计评测指标：根据任务类型和领域需求，选择合适的评测指标来评估模型的性能。例如，对于文本分类任务，可以使用准确率、召回率、F1值等指标来衡量模型的性能。
- e. 进行评测：使用构建好的评测集对微调后的模型进行评测。将评测集输入模型，获取模型的预测结果，并与标注结果进行比较，计算评测指标。
- f. 分析和改进：根据评测结果，分析模型在不同方面的表现，并根据需要进行模型的改进和调整。可以尝试不同的超参数设置、模型架构或优化算法，以提高模型的性能。

10.💡 领域模型词表扩增是不是有必要的？

领域模型的词表扩增可以有助于提升模型在特定领域任务上的性能，但是否有必要取决于具体的情况。以下是一些考虑因素：

需要注意的是，词表扩增可能会增加模型的计算和存储成本。因此，在决定是否进行词表扩增时，需要综合考虑领域特定词汇的重要性、数据稀缺性以及计算资源的限制等因素。有时候，简单的词表截断或者使用基于规则的方法来处理领域特定词汇也可以取得不错的效果。最佳的词表扩增策略会因特定任务和领域的需求而有所不同，建议根据具体情况进行评估和实验。

- a. 领域特定词汇：如果目标领域中存在一些特定的词汇或术语，而这些词汇在通用的预训练模型的词表中没有覆盖到，那么词表扩增就是必要的。通过将这些领域特定的词汇添加到模型的词表中，可以使模型更好地理解 and 处理这些特定的词汇。
- b. 领域特定上下文：在某些领域任务中，词汇的含义可能会受到特定上下文的影响。例如，在医学领域中，同一个词汇在不同的上下文中可能具有不同的含义。如果领域任务中的上下文与通用预训练模型的训练数据中的上下文有较大差异，那么词表扩增可以帮助模型更好地理解 and 处理领域特定的上下文。
- c. 数据稀缺性：如果目标领域的训练数据相对较少，而通用预训练模型的词表较大，那么词表扩增可以帮助模型更好地利用预训练模型的知识，并提升在目标领域任务上的性能。

11.💡 如何训练自己的大模型？

训练自己的大模型通常需要以下步骤：

需要注意的是，训练自己的大模型通常需要大量的计算资源和时间。可以考虑使用云计算平台或者分布式训练来加速训练过程。此外，对于大模型的训练，还需要仔细选择合适的超参数和进行调优，以避免过拟合或者欠拟合的问题。

- a. 数据收集和准备：首先，需要收集与目标任务和领域相关的大规模数据集。这可以包括从互联网上爬取数据、使用公开数据集或者与合作伙伴合作获取数据。然后，对数据进行预处理和清洗，包括去除噪声、处理缺失值、标准化数据等。
- b. 模型设计和架构选择：根据任务的特点和目标，选择适合的模型架构。可以基于已有的模型进行修改和调整，或者设计全新的模型。常见的大模型架构包括深度神经网络

（如卷积神经网络、循环神经网络、Transformer等）和预训练语言模型（如BERT、GPT等）。

- c. 数据划分和预处理：将数据集划分为训练集、验证集和测试集。训练集用于模型的训练，验证集用于调整超参数和模型选择，测试集用于最终评估模型的性能。进行数据预处理，如分词、编码、标记化、特征提取等，以便输入到模型中。
- d. 模型训练：使用训练集对模型进行训练。选择合适的优化算法、损失函数和学习率等超参数，并进行模型的调整和优化。可以使用GPU或者分布式训练来加速训练过程。
- e. 模型调优和验证：使用验证集对训练过程中的模型进行调优和验证。根据验证集的性能指标，调整模型的超参数、网络结构或者其他相关参数，以提升模型的性能。
- f. 模型评估和测试：使用测试集对最终训练好的模型进行评估和测试。计算模型的性能指标，如准确率、召回率、F1值等，评估模型的性能和泛化能力。
- g. 模型部署和优化：将训练好的模型部署到实际应用中。根据实际需求，对模型进行进一步的优化和调整，以提高模型的效率和性能。

12. 训练中文大模型有啥经验？

训练中文大模型时，以下经验可能会有所帮助：

需要注意的是，中文的复杂性和语义特点可能会对模型的训练和性能产生影响。因此，在训练中文大模型时，需要充分理解中文语言的特点，并根据具体任务和需求进行调整和优化。同时，也可以参考相关的中文自然语言处理研究和实践经验，以获取更多的指导和启发。

- a. 数据预处理：对于中文文本，常见的预处理步骤包括分词、去除停用词、词性标注、拼音转换等。分词是中文处理的基本步骤，可以使用成熟的中文分词工具，如jieba、pkuseg等。
- b. 数据增强：中文数据集可能相对有限，可以考虑使用数据增强技术来扩充数据集。例如，可以使用同义词替换、随机插入或删除词语、句子重组等方法来生成新的训练样本。
- c. 字词级别的表示：中文中既有字级别的表示，也有词级别的表示。对于字级别的表示，可以使用字符嵌入或者字级别的CNN、RNN等模型。对于词级别的表示，可以使用预训练的词向量，如Word2Vec、GloVe等。
- d. 预训练模型：可以考虑使用已经在大规模中文语料上预训练好的模型作为初始模型，然后在目标任务上进行微调。例如，可以使用BERT、GPT等预训练语言模型。这样可以利用大规模中文语料的信息，提升模型的表达能力和泛化能力。
- e. 中文特定的任务：对于一些中文特定的任务，例如中文分词、命名实体识别、情感分析等，可以使用一些中文特定的工具或者模型来辅助训练。例如，可以使用THULAC、LTP等中文NLP工具包。
- f. 计算资源：训练大模型需要大量的计算资源，包括GPU、内存和存储。可以考虑使用云计算平台或者分布式训练来加速训练过程。
- g. 超参数调优：对于大模型的训练，超参数的选择和调优非常重要。可以使用网格搜索、随机搜索或者基于优化算法的自动调参方法来寻找最佳的超参数组合。

13. 指令微调的好处？

在大模型训练中进行指令微调（Instruction Fine-tuning）的好处包括：

指令微调的好处在于在大模型的基础上进行个性化调整，以适应特定任务的需求和提升性能，同时还能节省训练时间和资源消耗。

- a. 个性化适应：大模型通常是在大规模通用数据上进行训练的，具有强大的语言理解和表示能力。但是，对于某些特定任务或领域，模型可能需要更加个性化的适应。通过指令微调，可以在大模型的基础上，使用特定任务或领域的数据进行微调，使模型更好地适应目标任务的特点。
- b. 提升性能：大模型的泛化能力通常很强，但在某些特定任务上可能存在一定的性能瓶颈。通过指令微调，可以针对特定任务的要求，调整模型的参数和结构，以提升性能。



能。例如，在机器翻译任务中，可以通过指令微调来调整注意力机制、解码器结构等，以提高翻译质量。

c. 控制模型行为：大模型通常具有很高的复杂性和参数数量，其行为可能难以解释和控制。通过指令微调，可以引入特定的指令或约束，以约束模型的行为，使其更符合特定任务的需求。例如，在生成式任务中，可以使用基于指令的方法来控制生成结果的风格、长度等。

继续访问

取消

d. 数据效率：大模型的训练需要海量的数据，但在某些任务或领域中，特定数据可能相对稀缺或难以获取。通过指令微调，可以利用大模型在通用数据上的预训练知识，结合少量特定任务数据进行微调，从而在数据有限的情况下获得更好的性能。

e. 提高训练效率：大模型的训练通常需要大量的计算资源和时间。通过指令微调，可以在已经训练好的大模型的基础上进行微调，避免从头开始训练的时间和资源消耗，从而提高训练效率。

14. 预训练和微调哪个阶段注入知识的？

在大模型训练过程中，知识注入通常是在预训练阶段进行的。具体来说，大模型的训练一般包括两个阶段：预训练和微调。

在预训练阶段，使用大规模的通用数据对模型进行训练，以学习语言知识和表示能力。这一阶段的目标是通过自监督学习或其他无监督学习方法，让模型尽可能地捕捉到数据中的统计规律和语言结构，并生成丰富的语言表示。

在预训练阶段，模型并没有针对特定任务进行优化，因此预训练模型通常是通用的，可以应用于多个不同的任务和领域。

在微调阶段，使用特定任务的数据对预训练模型进行进一步的训练和调整。微调的目标是将预训练模型中学到的通用知识和能力迁移到特定任务上，提升模型在目标任务上的性能。

在微调阶段，可以根据具体任务的需求，调整模型的参数和结构，以更好地适应目标任务的特点。微调通常需要较少的任务数据，因为预训练模型已经具备了一定的语言理解和泛化能力。

因此，知识注入是在预训练阶段进行的，预训练模型通过大规模通用数据的训练，学习到了丰富的语言知识和表示能力，为后续的微调阶段提供了基础。微调阶段则是在预训练模型的基础上，使用特定任务的数据进行进一步训练和调整，以提升性能。

15. 想让模型学习某个领域或行业的知识，是应该预训练还是应该微调？

如果你想让大语言模型学习某个特定领域或行业的知识，通常建议进行微调而不是预训练。

预训练阶段是在大规模通用数据上进行的，旨在为模型提供通用的语言理解和表示能力。预训练模型通常具有较强的泛化能力，可以适用于多个不同的任务和领域。然而，由于预训练模型是在通用数据上进行训练的，其对特定领域的知识和术语可能了解有限。

因此，如果你希望大语言模型能够学习某个特定领域或行业的知识，微调是更合适的选择。在微调阶段，你可以使用特定领域的数据对预训练模型进行进一步训练和调整，以使模型更好地适应目标领域的特点和需求。微调可以帮助模型更深入地理解特定领域的术语、概念和语境，并提升在该领域任务上的性能。

微调通常需要较少的任务数据，因为预训练模型已经具备了一定的语言理解和泛化能力。通过微调，你可以在预训练模型的基础上，利用特定领域的数据进行有针对性的调整，以使模型更好地适应目标领域的需求。

总之，如果你希望大语言模型学习某个特定领域或行业的知识，建议进行微调而不是预训练。微调可以帮助模型更好地适应目标领域的特点和需求，并提升在该领域任务上的性能。

16. 多轮对话任务如何微调模型？

微调大语言模型用于多轮对话任务时，可以采用以下步骤：

需要注意的是，微调大语言模型用于多轮对话任务时，数据集的质量和多样性对模型性能至关重要。确保数据集包含各种对话场景和多样的对话历史，以提高模型的泛化能力和适应性。

此外，还可以使用一些技巧来增强模型性能，如数据增强、对抗训练、模型融合等。这些技巧可以进一步提高模

继续访问 取消

- a. 数据准备：收集或生成与目标对话任务相关的数据集。数据集应包含多轮对话的对话历史、当前对话回合的输入和对应的回答。
- b. 模型选择：选择一个合适的预训练模型作为基础模型。例如，可以选择GPT、BERT等大型语言模型作为基础模型。
- c. 任务特定层：为了适应多轮对话任务，需要在预训练模型上添加一些任务特定的层。这些层可以用于处理对话历史、上下文理解和生成回答等任务相关的操作。
- d. 微调过程：使用多轮对话数据集对预训练模型进行微调。微调的过程类似于监督学习，通过最小化模型在训练集上的损失函数来优化模型参数。可以使用常见的优化算法，如随机梯度下降（SGD）或Adam。
- e. 超参数调整：微调过程中需要选择合适的学习率、批次大小、训练轮数等超参数。可以通过交叉验证或其他调参方法来选择最佳的超参数组合。
- f. 评估和调优：使用验证集或开发集对微调后的模型进行评估。可以计算模型在多轮对话任务上的指标，如准确率、召回率、F1分数等，以选择最佳模型。
- g. 推理和部署：在微调后，可以使用微调后的模型进行推理和部署。将输入的多轮对话输入给模型，模型将生成对应的回答。

17. 微调后的模型出现能力劣化，灾难性遗忘是怎么回事？

灾难性遗忘（Catastrophic Forgetting）是指在模型微调过程中，当模型在新任务上进行训练时，可能会忘记之前学习到的知识，导致在旧任务上的性能下降。这种现象常见于神经网络模型的迁移学习或连续学习场景中。

在微调大语言模型时，灾难性遗忘可能出现的原因包括：

为了解决灾难性遗忘问题，可以尝试以下方法：

综上所述，灾难性遗忘是在模型微调过程中可能出现的问题。通过合适的方法和技术，可以减少灾难性遗忘的发生，保留之前学习到的知识，提高模型的整体性能。

- a. 重播缓冲区（Replay Buffer）：在微调过程中，使用一个缓冲区来存储旧任务的样本，然后将旧任务的样本与新任务的样本一起用于训练。这样可以保留旧任务的知识，减少灾难性遗忘的发生。
- b. 弹性权重共享（Elastic Weight Consolidation）：通过引入正则化项，限制模型参数的变动范围，以保护之前学习到的知识。这种方法可以在微调过程中平衡新任务和旧任务之间的重要性。
- c. 增量学习（Incremental Learning）：将微调过程分为多个阶段，每个阶段只微调一小部分参数。这样可以逐步引入新任务，减少参数更新的冲突，降低灾难性遗忘的风险。
- d. 多任务学习（Multi-Task Learning）：在微调过程中，同时训练多个相关任务，以提高模型的泛化能力和抗遗忘能力。通过共享模型参数，可以在不同任务之间传递知识，减少灾难性遗忘的影响。
- e. 数据分布差异：微调过程中使用的新任务数据与预训练数据或旧任务数据的分布存在差异。如果新任务的数据分布与预训练数据差异较大，模型可能会过度调整以适应新任务，导致旧任务上的性能下降。
- f. 参数更新冲突：微调过程中，对新任务进行训练时，模型参数可能会被更新，导致之前学习到的知识被覆盖或丢失。新任务的梯度更新可能会与旧任务的梯度更新发生冲突，导致旧任务的知识被遗忘。

18. 微调模型需要多大显存？

微调大语言模型所需的显存大小取决于多个因素，包括模型的大小、批次大小、序列长度和训练过程中使用的优化算法等。

对于大型语言模型，如GPT-2、GPT-3等，它们通常具有数亿或数十亿个参数，因此需要大量的显存来存储模型参数和梯度。一般来说，微调这些大型语言模型需要至少16GB以上的显存。

继续访问

取消

此外，批次大小和序列长度也会对显存需求产生影响。较大的批次大小和较长的序列长度会占用更多的显存。如果显存不足以容纳整个批次或序列，可能需要减小批次大小或序列长度，或者使用分布式训练等策略来解决显存不足的问题。

需要注意的是，显存需求还受到训练过程中使用的优化算法的影响。例如，如果使用梯度累积（Gradient Accumulation）来增加批次大小，可能需要更大的显存来存储累积的梯度。

综上所述，微调大语言模型所需的显存大小取决于模型的大小、批次大小、序列长度和训练过程中使用的优化算法等因素。在进行微调之前，需要确保显存足够大以容纳模型和训练过程中的数据。如果显存不足，可以考虑减小批次大小、序列长度或使用分布式训练等策略来解决显存不足的问题。

19. 大模型LLM进行SFT操作的时候在学习什么？

在大语言模型（LLM）进行有监督微调（Supervised Fine-Tuning）时，模型主要学习以下内容：

总的来说，有监督微调阶段主要通过任务特定的标签预测、上下文理解和语言模式、特征提取和表示学习以及任务相关的优化来进行学习。通过这些学习，模型可以适应特定的任务，并在该任务上表现出良好的性能。

- a. 任务特定的标签预测：在有监督微调中，模型会根据给定的任务，学习预测相应的标签或目标。例如，对于文本分类任务，模型会学习将输入文本映射到正确的类别标签。
- b. 上下文理解和语言模式：大语言模型在预训练阶段已经学习到了大量的语言知识和模式。在有监督微调中，模型会利用这些学习到的知识来更好地理解任务相关的上下文，并捕捉语言中的各种模式和规律。
- c. 特征提取和表示学习：微调过程中，模型会通过学习任务相关的表示来提取有用的特征。这些特征可以帮助模型更好地区分不同的类别或进行其他任务相关的操作。
- d. 任务相关的优化：在有监督微调中，模型会通过反向传播和优化算法来调整模型参数，使得模型在给定任务上的性能最优化。模型会学习如何通过梯度下降来最小化损失函数，从而提高任务的准确性或其他性能指标。

20. 预训练和SFT操作有什么不同

大语言模型的预训练和有监督微调（Supervised Fine-Tuning）是两个不同的操作，它们的目标、数据和训练方式等方面存在一些区别。

总的来说，预训练和有监督微调是大语言模型训练的两个阶段，目标、数据和训练方式等方面存在差异。预训练阶段通过无监督学习从大规模文本数据中学习语言模型，而有监督微调阶段则在特定任务上使用带有标签的数据进行有监督学习，以适应任务要求。

- a. 目标：预训练的目标是通过无监督学习从大规模的文本语料库中学习语言模型的表示能力和语言知识。预训练的目标通常是通过自我预测任务，例如掩码语言模型（Masked Language Model, MLM）或下一句预测（Next Sentence Prediction, NSP）等，来训练模型。  
  
有监督微调的目标是在特定的任务上进行训练，例如文本分类、命名实体识别等。在有监督微调中，模型会利用预训练阶段学到的语言表示和知识，通过有监督的方式调整模型参数，以适应特定任务的要求。
- b. 数据：在预训练阶段，大语言模型通常使用大规模的无标签文本数据进行训练，例如维基百科、网页文本等。这些数据没有特定的标签或任务信息，模型通过自我预测任

务来学习语言模型。

在有监督微调中，模型需要使用带有标签的任务相关数据进行训练。这些数据通常是人工标注的，包含了输入文本和对应的标签或目标。模型通过这些标签来进行有监督学习，调整参数以适应特定任务。

c. 训练方式：预训练时通过最大化预训练任务的目标函数来学习语言模型的参数。在继续访问时，通过取消预训练任务的目标函数来学习语言模型的参数。

有监督微调阶段则使用有监督的方式进行训练，模型通过最小化损失函数来学习任务相关的特征和模式。在微调阶段，通常会使用预训练模型的参数作为初始参数，并在任务相关的数据上进行训练。

21. 样本量规模增大，训练出现OOM错

当在大语言模型训练过程中，样本量规模增大导致内存不足的情况出现时，可以考虑以下几种解决方案：

综上所述，当在大语言模型训练中遇到内存不足的问题时，可以通过减小批量大小、分布式训练、内存优化技术、减少模型规模、增加硬件资源或优化数据处理等方式来解决。具体的解决方案需要根据具体情况进行选择和调整。

- a. 减少批量大小（Batch Size）：将批量大小减小可以减少每个训练步骤中所需要的内存量。较小的批量大小可能会导致训练过程中的梯度估计不稳定，但可以通过增加训练步骤的数量来弥补这一问题。
- b. 分布式训练：使用多台机器或多个GPU进行分布式训练可以将训练负载分散到多个设备上，从而减少单个设备上的内存需求。通过分布式训练，可以将模型参数和梯度在多个设备之间进行同步和更新。
- c. 内存优化技术：使用一些内存优化技术可以减少模型训练过程中的内存占用。例如，使用混合精度训练（Mixed Precision Training）可以减少模型参数的内存占用；使用梯度累积（Gradient Accumulation）可以减少每个训练步骤中的内存需求。
- d. 减少模型规模：如果内存问题仍然存在，可以考虑减少模型的规模，例如减少模型的层数、隐藏单元的数量等。虽然这可能会导致模型性能的一定损失，但可以在一定程度上减少内存需求。
- e. 增加硬件资源：如果条件允许，可以考虑增加硬件资源，例如增加内存容量或使用更高内存的设备。这样可以提供更多的内存空间来容纳更大规模的训练数据。
- f. 数据处理和加载优化：优化数据处理和加载过程可以减少训练过程中的内存占用。例如，可以使用数据流水线技术来并行加载和处理数据，减少内存中同时存在的数据量。

22. 大模型LLM进行SFT如何对样本进行优化？

对于大语言模型进行有监督微调（Supervised Fine-Tuning）时，可以采用以下几种方式对样本进行优化：

总的来说，对大语言模型进行有监督微调时，可以通过数据清洗和预处理、数据增强、标签平衡、样本选择、样本权重、样本组合和分割、样本筛选和策略等方式对样本进行优化。这些优化方法可以提高训练样本的质量、多样性和数量，从而提升模型的性能和泛化能力。具体的优化策略需要根据任务需求和数据特点进行选择和调整。

- a. 数据清洗和预处理：对于有监督微调的任务，首先需要对样本数据进行清洗和预处理。这包括去除噪声、处理缺失值、进行标准化或归一化等操作，以确保数据的质量和一致性。
- b. 数据增强：通过数据增强技术可以扩充训练数据，增加样本的多样性和数量。例如，可以使用数据扩充方法如随机裁剪、旋转、翻转、加噪声等来生成新的训练样本，从而提高模型的泛化能力。
- c. 标签平衡：如果样本标签不平衡，即某些类别的样本数量远远多于其他类别，可以采取一些方法来平衡样本标签。例如，可以通过欠采样、过采样或生成合成样本等技术来平衡不同类别的样本数量。



- d. 样本选择：在有限的资源和时间下，可以选择一部分具有代表性的样本进行微调训练。可以根据任务的需求和数据分布的特点，选择一些关键样本或难样本进行训练，以提高模型在关键样本上的性能。
- e. 样本权重：对于一些重要的样本或困难样本，可以给予更高的权重，以便模型更加关注这些样本的学习。可以通过调整损失函数中样本的权重或采用加权采样的方式来实现。
- f. 样本组合和分割：根据任务的特征和数据结构，可以将多个样本组合成一个样本，或将一个样本分割成多个子样本。这样可以扩展训练数据，提供更多的信息和多样性。
- g. 样本筛选和策略：根据任务需求，可以制定一些样本筛选和选择策略。例如，可以根据样本的置信度、难度、多样性等指标进行筛选和选择，以提高模型的性能和泛化能力。

继续访问

取消

23. 模型参数迭代实验

模型参数迭代实验是指通过多次迭代更新模型参数，以逐步优化模型性能的过程。在实验中，可以尝试不同的参数更新策略、学习率调整方法、正则化技术等，以找到最佳的参数配置，从而达到更好的模型性能。

下面是一个基本的模型参数迭代实验过程：

通过模型参数迭代实验，可以逐步优化模型性能，找到最佳的参数配置。在实验过程中，需要注意过拟合和欠拟合等问题，并及时调整模型结构和正则化技术来解决。同时，要进行合理的实验设计和结果分析，以得到可靠的实验结论。

- a. 设定初始参数：首先，需要设定初始的模型参数。可以通过随机初始化或使用预训练模型的参数作为初始值。
- b. 选择损失函数：根据任务的特点，选择适当的损失函数作为模型的优化目标。常见的损失函数包括均方误差（MSE）、交叉熵损失等。
- c. 选择优化算法：选择适当的优化算法来更新模型参数。常见的优化算法包括随机梯度下降（SGD）、Adam、Adagrad等。可以尝试不同的优化算法，比较它们在模型训练过程中的效果。
- d. 划分训练集和验证集：将样本数据划分为训练集和验证集。训练集用于模型参数的更新，验证集用于评估模型性能和调整超参数。
- e. 迭代更新参数：通过多次迭代更新模型参数来优化模型。每次迭代中，使用训练集的一批样本进行前向传播和反向传播，计算损失函数并更新参数。可以根据需要调整批量大小、学习率等超参数。
- f. 评估模型性能：在每次迭代的过程中，可以使用验证集评估模型的性能。可以计算准确率、精确率、召回率、F1值等指标，以及绘制学习曲线、混淆矩阵等来分析模型的性能。
- g. 调整超参数：根据验证集的评估结果，可以调整超参数，如学习率、正则化系数等，以进一步提升模型性能。可以使用网格搜索、随机搜索等方法来寻找最佳的超参数配置。
- h. 终止条件：可以设置终止条件，如达到最大迭代次数、模型性能不再提升等。当满足终止条件时，结束模型参数迭代实验。

大模型（LLMs）langchain面

1. 什么是 LangChain?

💡 [https://python.langchain.com/docs/get\_started/introduction]  
(https://python.langchain.com/docs/get\_started/introduction)

LangChain 是一个基于语言模型的框架，用于构建聊天机器人、生成式问答（GQA）、摘要等功能。它的核心思想是将不同的组件“链”在一起，以创建更高级的语言模型应用。LangChain 的起源可以追溯到 2022 年 10 月，由创造者 Harrison Chase 在那时提交了第一个版本。与 Bitcoin 不同，Bitcoin 是在 2009 年由一位使用化名 Satoshi Nakamoto 的未知人士创建的，它是一种去中心化的加密货币。而 LangChain 是围绕语言模型构建的框架。

## 2. LangChain 包含哪些 核心概念？

- **StreamlitChatMessageHistory**: 用于在 Streamlit 应用程序中存储和使用聊天消息历史记录。它使用 Streamlit 会话状态来存储消息，并可以与 **ConversationBufferMemory** 和链或代理一起使用。
- **CassandraChatMessageHistory**: 用于使用 Cassandra 数据库存储聊天消息历史记录。Cassandra 是一个分布式数据库，适用于存储大量数据。
- **MongoDBChatMessageHistory**: 使用 MongoDB 数据库存储聊天消息历史记录。MongoDB 是一种面向文档的 NoSQL 数据库，使用类似 JSON 的文档进行存储。

- a. Training data selection: Users can use example selectors to filter and select specific examples from a large training dataset. This can be useful when working with limited computational resources or when focusing on specific subsets of the data.
- b. Inference customization: Example selectors can be used to retrieve specific examples from a dataset during the inference process. This allows users to generate responses or predictions based on specific conditions or criteria.

## c. LangChain 中 Components and Chains 是什么？

💡 [<https://python.langchain.com/docs/modules/chains/>]  
 (<https://python.langchain.com/docs/modules/chains/>)

Components and Chains are key concepts in the LangChain framework.

Components refer to the individual building blocks or modules that make up the LangChain framework. These components can include language models, data preprocessors, response generators, and other functionalities. Each component is responsible for a specific task or functionality within the language model application.

Chains, on the other hand, are the connections or links between these components. They define the flow of data and information within the language model application. Chains allow the output of one component to serve as the input for another component, enabling the creation of more advanced language models.

In summary, Components are the individual modules or functionalities within the LangChain framework, while Chains define the connections and flow of data between these components.

Here's an example to illustrate the concept of Components and Chains in LangChain:

```
from langchain import Component, Chain

# Define components
preprocessor = Component("Preprocessor")
language_model = Component("Language Model")
response_generator = Component("Response Generator")

# Define chains
chain1 = Chain(preprocessor, language_model)
chain2 = Chain(language_model, response_generator)

# Execute chains
input_data = "Hello, how are you?"
processed_data = chain1.execute(input_data)
response = chain2.execute(processed_data)

print(response)
```

In the above example, we have three components: Preprocessor, Language Model, and Response Generator. We create two chains: chain1 connects the Preprocessor and

Language Model, and chain2 connects the Language Model and Response Generator. The input data is passed through chain1 to preprocess it and then passed through chain2 to generate a response.

This is a simplified example to demonstrate the concept of Components and Chains in LangChain. In a real-world scenario, you would have more complex chains with multiple components and data flows.

#### d. LangChain 中 Prompt Templates and Values 是什么？



[[https://python.langchain.com/docs/modules/model\\_io/prompts/prompt\\_templates/](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/)]  
[[https://python.langchain.com/docs/modules/model\\_io/prompts/prompt\\_templates/](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/)]

Prompt Templates and Values are key concepts in the LangChain framework.

Prompt Templates refer to predefined structures or formats that guide the generation of prompts for language models. These templates provide a consistent and standardized way to construct prompts by specifying the desired input and output formats. Prompt templates can include placeholders or variables that are later filled with specific values.

Values, on the other hand, are the specific data or information that is used to fill in the placeholders or variables in prompt templates. These values can be dynamically generated or retrieved from external sources. They provide the necessary context or input for the language model to generate the desired output.

Here's an example to illustrate the concept of Prompt Templates and Values in LangChain:

```
from langchain import PromptTemplate, Value

# Define prompt template
template = PromptTemplate("What is the capital of {country}?")

# Define values
country_value = Value("country", "France")

# Generate prompt
prompt = template.generate_prompt(values=[country_value])

print(prompt)
```

In the above example, we have a prompt template that asks for the capital of a country. The template includes a placeholder `{country}` that will be filled with the actual country value. We define a value object `country_value` with the name "country" and the value "France". We then generate the prompt by passing the value object to the template's `generate_prompt` method.

The generated prompt will be "What is the capital of France?".

Prompt templates and values allow for flexible and dynamic generation of prompts in the LangChain framework. They enable the customization and adaptation of prompts based on specific requirements or scenarios.

#### e. LangChain 中 Example Selectors 是什么？



[[https://python.langchain.com/docs/modules/model\\_io/prompts/example\\_selectors/](https://python.langchain.com/docs/modules/model_io/prompts/example_selectors/)]  
[[https://python.langchain.com/docs/modules/model\\_io/prompts/example\\_selectors/](https://python.langchain.com/docs/modules/model_io/prompts/example_selectors/)]

Example Selectors are a feature in the LangChain framework that allow users to specify and retrieve specific examples or data points from a dataset. These selectors help in customizing the training or inference process by selecting specific examples that meet certain criteria or conditions.

Example Selectors c [继续访问](#) us [取消](#)

Here's an example to illustrate the concept of Example Selectors in LangChain:

```
from langchain import ExampleSelector

# Define an example selector
selector = ExampleSelector(condition="label=='positive'")

# Retrieve examples based on the selector
selected_examples = selector.select_examples(dataset)

# Use the selected examples for training or inference
for example in selected_examples:
    # Perform training or inference on the selected example
    ...
```

In the above example, we define an example selector with a condition that selects examples with a label equal to "positive". We then use the selector to retrieve the selected examples from a dataset. These selected examples can be used for training or inference purposes.

Example Selectors provide a flexible way to customize the data used in the LangChain framework. They allow users to focus on specific subsets of the data or apply specific criteria to select examples that meet their requirements.

f. LangChain 中 Output Parsers 是什么？

💡 [[https://python.langchain.com/docs/modules/model\\_io/output\\_parsers/](https://python.langchain.com/docs/modules/model_io/output_parsers/)]  
([https://python.langchain.com/docs/modules/model\\_io/output\\_parsers/](https://python.langchain.com/docs/modules/model_io/output_parsers/))

Output Parsers are a feature in the LangChain framework that allow users to automatically detect and parse the output generated by the language model. These parsers are designed to handle different types of output, such as strings, lists, dictionaries, or even Pydantic models.

Output Parsers provide a convenient way to process and manipulate the output of the language model without the need for manual parsing or conversion. They help in extracting relevant information from the output and enable further processing or analysis.

Here's an example to illustrate the concept of Output Parsers in LangChain:

```
from langchain import llm_prompt, OutputParser

# Define an output parser
parser = OutputParser()

# Apply the output parser to a function
@llm_prompt(output_parser=parser)
def generate_response(input_text):
    # Generate response using the language model
    response = language_model.generate(input_text)
```



```

return response

# Generate a response
input_text = "Hello, how are you?"
response = generate_response(input_text)

# Parse the output
parsed_output = parser.parse_output(response)

# Process the parsed output
processed_output = process_output(parsed_output)

print(processed_output)

```

In the above example, we define an output parser and apply it to the `generate_response` function using the `llm_prompt` decorator. The output parser automatically detects the type of the output and provides the parsed output. We can then further process or analyze the parsed output as needed.

Output Parsers provide a flexible and efficient way to handle the output of the language model in the LangChain framework. They simplify the post-processing of the output and enable seamless integration with other components or systems.

g. LangChain 中 Indexes and Retrievers 是什么？

💡 [[https://python.langchain.com/docs/modules/data\\_connection/retrievers/](https://python.langchain.com/docs/modules/data_connection/retrievers/)]  
[\(https://python.langchain.com/docs/modules/data\\_connection/retrievers/\)](https://python.langchain.com/docs/modules/data_connection/retrievers/)  
[https://python.langchain.com/docs/modules/data\\_connection/indexing](https://python.langchain.com/docs/modules/data_connection/indexing)

Indexes and Retrievers are components in the Langchain framework.

Indexes are used to store and organize data for efficient retrieval. Langchain supports multiple types of document indexes, such as `InMemoryExactNNIndex`, `HnswDocumentIndex`, `WeaviateDocumentIndex`, `ElasticDocIndex`, and `QdrantDocumentIndex`. Each index has its own characteristics and is suited for different use cases. For example, `InMemoryExactNNIndex` is suitable for small datasets that can be stored in memory, while `HnswDocumentIndex` is lightweight and suitable for small to medium-sized datasets.

Retrievers, on the other hand, are used to retrieve relevant documents from the indexes based on a given query. Langchain provides different types of retrievers, such as `MetalRetriever` and `DocArrayRetriever`. `MetalRetriever` is used with the Metal platform for semantic search and retrieval, while `DocArrayRetriever` is used with the DocArray tool for managing multi-modal data.

Overall, indexes and retrievers are essential components in Langchain for efficient data storage and retrieval.

h. LangChain 中 Chat Message History 是什么？

💡 [[https://python.langchain.com/docs/modules/memory/chat\\_messages/](https://python.langchain.com/docs/modules/memory/chat_messages/)]  
[\(https://python.langchain.com/docs/modules/memory/chat\\_messages/\)](https://python.langchain.com/docs/modules/memory/chat_messages/)

Chat Message History 是 Langchain 框架中的一个组件，用于存储和管理聊天消息的历史记录。它可以跟踪和保存用户和AI之间的对话，以便在需要进行检索和分析。

Langchain 提供了不同的 Chat Message History 实现，包括 `StreamlitChatMessageHistory`、`CassandraChatMessageHistory` 和 `MongoDBChatMessageHistory`。

您可以根据自己的需求选择适合的 Chat Message History 实现，并将其集成到 Langchain 框架中，以便记录和管理聊天消息的历史记录。

请注意，Chat Message History 的具体用法和实现细节可以参考 Langchain 的官方文档和示例代码。

#### i. LangChain 中 Agents and Toolkits 是什么？

💡 [https://python.langchain.com/docs/modules/agents/](https://python.langchain.com/docs/modules/agents/)

(https://python.lan 继续访问 oc 取消

https://python.langchain.com/docs/modules/agents/toolkits/

Agents and Toolkits in LangChain are components that are used to create and manage conversational agents.

Agents are responsible for determining the next action to take based on the current state of the conversation. They can be created using different approaches, such as OpenAI Function Calling, Plan-and-execute Agent, Baby AGI, and Auto GPT. These approaches provide different levels of customization and functionality for building agents.

Toolkits, on the other hand, are collections of tools that can be used by agents to perform specific tasks or actions. Tools are functions or methods that take input and produce output. They can be custom-built or pre-defined and cover a wide range of functionalities, such as language processing, data manipulation, and external API integration.

By combining agents and toolkits, developers can create powerful conversational agents that can understand user inputs, generate appropriate responses, and perform various tasks based on the given context.

Here is an example of how to create an agent using LangChain:

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import tool

# Load the language model
llm = ChatOpenAI(temperature=0)

# Define a custom tool
@tool
def get_word_length(word: str) -> int:
    """Returns the length of a word."""
    return len(word)

# Create the agent
agent = {
    "input": lambda x: x["input"],
    "agent_scratchpad": lambda x: format_to_openai_functions(x['intermediate_s
] | prompt | llm_with_tools | OpenAIFunctionsAgentOutputParser())

# Invoke the agent
output = agent.invoke({
    "input": "how many letters in the word educa?",
    "intermediate_steps": []
})

# Print the result
print(output.return_values["output"])
```

This is just a basic example, and there are many more features and functionalities available in LangChain for building and customizing agents and toolkits. You can refer to the LangChain documentation for more details and examples.

### 3. 什么是 LangChain Agent?

 [https://python.langchain.com/docs/modules/agents/](https://python.langchain.com/docs/modules/agents/)

LangChain Agent 是 LangChain 框架中的一个组件，用于创建和管理对话代理。代理是根据当前对话状态确定下一步操作的组件。LangChain 提供了多种创建代理的方法，包括 OpenAI Function Calling、Plan-and-execute Agent、Baby AGI 和 Auto GPT 等。这些方法提供了不同级别的自定义和功能，用于构建代理。

代理可以使用工具包执行特定的任务或操作。工具包是代理使用的一组工具，用于执行特定的功能，如语言处理、数据操作和外部 API 集成。工具可以是自定义构建的，也可以是预定义的，涵盖了广泛的功能。

通过结合代理和工具包，开发人员可以创建强大的对话代理，能够理解用户输入，生成适当的回复，并根据给定的上下文执行各种任务。

以下是使用 LangChain 创建代理的示例代码：

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import tool

# 加载语言模型
llm = ChatOpenAI(temperature=0)

# 定义自定义工具
@tool
def get_word_length(word: str) -> int:
    """返回单词的长度。"""
    return len(word)


# 创建代理
agent = {
    "input": lambda x: x["input"],
    "agent_scratchpad": lambda x: format_to_openai_functions(x['intermediate_step']
} | prompt | llm_with_tools | OpenAIFunctionsAgentOutputParser()

# 调用代理
output = agent.invoke({
    "input": "单词 educa 中有多少个字母？",
    "intermediate_steps": []
})

# 打印结果
print(output.return_values["output"])
```

这只是一个基本示例，LangChain 中还有更多功能和功能可用于构建和自定义代理和工具包。您可以参考 LangChain 文档以获取更多详细信息和示例。

### 4. 如何使用 LangChain ?

 [https://python.langchain.com/docs/get\_started/quickstart](https://python.langchain.com/docs/get\_started/quickstart)

To use LangChain, you first need to sign up for an API key at [platform.langchain.com](https://platform.langchain.com). Once you have your API key, you can install the Python library and write a simple Python script to call the LangChain API. Here is some sample code to get started:

```
import langchain 继续访问 取消

api_key = "YOUR_API_KEY"

langchain.set_key(api_key)

response = langchain.ask("What is the capital of France?")

print(response.response)
```

This code will send the question "What is the capital of France?" to the LangChain API and print the response. You can customize the request by providing parameters like `max_tokens`, `temperature`, etc. The LangChain Python library documentation has more details on the available options.

#### 5. LangChain 支持哪些功能?



LangChain支持以下功能:

以上是LangChain支持的一些功能。您可以根据具体的需求使用这些功能来创建生产就绪的聊天应用程序。



LangChain支持以下功能:

- 编写自定义的LangChain提示和链式代码的语法糖
  - 使用IDE内置的支持进行提示、类型检查和弹出文档，以快速查看函数的提示和参数
  - 利用LangChain生态系统的全部功能
  - 添加对可选参数的支持
  - 通过将参数绑定到一个类来轻松共享参数
  - 支持传递内存和回调函数
  - 简化的流式处理
  - 定义聊天消息提示
  - 可选部分
  - 输出解析器
  - 支持更复杂的数据结构
- a. 编写帖子的短标题: 使用\*\* `write_me_short_post` 函数可以生成关于特定主题、平台和受众的短标题。该函数的参数包括 `topic` (主题)、`platform` (平台, 默认为 **Twitter**) 和 `audience` \*\* (受众, 默认为开发人员)。生成的标题应该在15个单词以内。
  - b. 模拟对话: 使用\*\* `simulate_conversation` \*\*函数可以模拟对话, 包括系统消息、用户消息和助手消息。对话可以根据角色 (如助手、用户、系统) 进行交互, 并可以包含历史记录。这对于训练聊天模型非常有用。
  - c. 可选部分: 可以在提示中定义可选部分, 只有在所有参数都不为空时才会渲染该部分。这可以通过在提示中使用\*\* `{? ... ?}` \*\*语法来实现。
  - d. 输出解析器: \*\* `llm_prompt` \*\*装饰器可以自动检测输出类型, 并提供相应的解析器。支持的输出类型包括字符串、列表、字典和Pydantic模型。

#### 6. 什么是 LangChain model?



LangChain model 是一个基于语言模型的框架，用于构建聊天机器人、生成式问答（GQA）、摘要等功能。LangChain 的核心思想是可以将不同的组件“链”在一起，以创建更高级的语言模型应用。

💡 LangChain model 是一种基于大型语言模型（LLM）的模型。它是LangChain框架的核心组件之一，用于构建基于语言模型的应用程序。LangChain 模型可以用于聊天机器人、生成式问答、摘要等多种场景。通过 LangChain，开发人员能够使用 LLM 来处理自然语言处理任务。LangChain 模型的目标是简化开发过程，使开发人员能够更轻松地构建强大的语言模型应用程序。

## 7. LangChain 包含哪些特点？



LangChain 包含以下特点：

- 编写自定义的LangChain提示和链式代码的语法糖
- 使用IDE内置的支持进行提示、类型检查和弹出文档，以快速查看函数的提示和参数
- 利用LangChain生态系统的全部功能
- 添加对可选参数的支持
- 通过将参数绑定到一个类来轻松共享参数
- 支持传递内存和回调函数
- 简化的流式处理
- 定义聊天消息提示
- 可选部分
- 输出解析器
- 支持更复杂的数据结构

## 8. LangChain 如何使用？

### a. LangChain 如何调用 LLMs 生成回复？



要调用LLMs生成回复，您可以使用LangChain框架提供的LLMChain类。LLMChain类是LangChain的一个组件，用于与语言模型进行交互并生成回复。以下是一个示例代码片段，展示了如何使用LLMChain类调用LLMs生成回复：

```
from langchain.llms import OpenAI
from langchain.chains import LLMChain

llm = OpenAI(temperature=0.9) # 创建LLM实例
prompt = "用户的问题" # 设置用户的问题

# 创建LLMChain实例
chain = LLMChain(llm=llm, prompt=prompt)

# 调用LLMs生成回复
response = chain.generate()

print(response) # 打印生成的回复
```

在上面的代码中，我们首先创建了一个LLM实例，然后设置了用户的问题作为LLMChain的prompt。接下来，我们调用LLMChain的generate方法来生成回复。最后，我们打印生成的回复。

请注意，您可以根据需要自定义LLM的参数，例如温度（temperature）、最大令牌数（max\_tokens）等。LangChain文档中有关于LLMChain类和LLM参数的更多详细信息。

## b. LangChain 如何修改 提示模板？



要修改LangChain的提示模板，您可以使用LangChain框架提供的\*\* `ChatPromptTemplate` 类。 `ChatPromptTemplate` 类允许您创建自定义的聊天消息提示，并根据需要进行修改。以下是一个 [继续访问](#) 示 [取消](#) `ChatPromptTemplate` \*\*类修改提示模板：

```
from langchain.prompts import ChatPromptTemplate

# 创建一个空的ChatPromptTemplate实例
template = ChatPromptTemplate()

# 添加聊天消息提示
template.add_message("system", "You are a helpful AI bot.")
template.add_message("human", "Hello, how are you doing?")
template.add_message("ai", "I'm doing well, thanks!")
template.add_message("human", "What is your name?")

# 修改提示模板
template.set_message_content(0, "You are a helpful AI assistant.")
template.set_message_content(3, "What is your name? Please tell me.")

# 格式化聊天消息
messages = template.format_messages()

print(messages)
```

在上面的代码中，我们首先创建了一个空的\*\* `ChatPromptTemplate` 实例。然后，我们使用 `add_message` 方法添加了聊天消息提示。接下来，我们使用 `set_message_content` 方法修改了第一个和最后一个聊天消息的内容。最后，我们使用 `format_messages` \*\*方法格式化聊天消息，并打印出来。

请注意，您可以根据需要添加、删除和修改聊天消息提示。\*\* `ChatPromptTemplate` \*\*类提供了多种方法来操作提示模板。更多详细信息和示例代码可以在LangChain文档中找到。

## c. LangChain 如何链接多个组件处理一个特定的下游任务？



要链接多个组件处理一个特定的下游任务，您可以使用LangChain框架提供的\*\* `Chain` 类。 `Chain` 类允许您将多个组件连接在一起，以便按顺序处理任务。以下是一个示例代码片段，展示了如何使用 `Chain` \*\*类链接多个组件处理下游任务：

```
from langchain.chains import Chain
from langchain.components import Component1, Component2, Component3

# 创建组件实例
component1 = Component1()
component2 = Component2()
component3 = Component3()

# 创建Chain实例并添加组件
chain = Chain()
chain.add_component(component1)
chain.add_component(component2)
```

```
chain.add_component(component3)

# 处理下游任务
output = chain.process_downstream_task()

print(output)
```

继续访问

取消

在上面的代码中，我们首先创建了多个组件的实例，例如\*\* `Component1` 、 `Component 2` 和 `Component3` 。然后，我们创建了一个 `Chain` 实例，并使用 `add_component` 方法将这些组件添加到链中。最后，我们调用 `process_downstream_task` \*\*方法来处理下游任务，并打印输出结果。

请注意，您可以根据需要添加、删除和修改组件。\*\* `Chain` \*\*类提供了多种方法来操作链。更多详细信息和示例代码可以在LangChain文档中找到。

#### d. LangChain 如何Embedding & vector store?



要在LangChain中进行嵌入和向量存储，您可以使用LangChain框架提供的\*\* `Embedding` 和 `VectorStore` 类。 `Embedding` 类用于将文本嵌入到向量空间中，而 `VectorStore` \*\*类用于存储和检索嵌入向量。以下是一个示例代码片段，展示了如何在LangChain中进行嵌入和向量存储：

```
from langchain.embeddings import Embedding
from langchain.vectorstore import VectorStore

# 创建Embedding实例
embedding = Embedding()

# 将文本嵌入到向量空间中
embedding.embed("Hello, world!")

# 创建VectorStore实例
vector_store = VectorStore()

# 存储嵌入向量
vector_store.store("hello", embedding.get_embedding())

# 检索嵌入向量
vector = vector_store.retrieve("hello")

print(vector)
```

在上面的代码中，我们首先创建了一个\*\* `Embedding` 实例，并使用 `embed` 方法将文本嵌入到向量空间中。然后，我们创建了一个 `VectorStore` 实例，并使用 `store` 方法将嵌入向量存储到向量存储中。最后，我们使用 `retrieve` \*\*方法检索嵌入向量，并打印出来。

请注意，您可以根据需要添加、删除和修改嵌入向量。\*\* `Embedding` 类和 `VectorStore` \*\*类提供了多种方法来操作嵌入和向量存储。更多详细信息和示例代码可以在LangChain文档中找到。

#### 9. LangChain 存在哪些问题及方法方案？

##### a. LangChain 低效的令牌使用问题



LangChain的token使用是高效的。LangChain使用了一种称为"token-based"的方法来处理文本输入和输出。这种方法将文本分解为小的单元，称为"tokens"，并对它们进行处理。相比于传统的字符或词语级别的处理，使用tokens可以更高效地处理文本。

LangChain还提供了一些参数，如\*\* max\_tokens 和 temperature \*\*，可以用来控制生成回复的长度和多样性。继续访问 参 取消 根据自己的需求来平衡生成回复的效率和质量。

总的来说，LangChain的token使用是高效的，并且开发人员可以通过调整参数来控制生成回复的效果。

- b. LangChain 文档的问题
- c. LangChain 太多概念容易混淆，过多的“辅助”函数问题
- d. LangChain 行为不一致并且隐藏细节问题
- e. LangChain 缺乏标准的可互操作数据类型问题

💡 LangChain提供了一种标准的接口，使开发人员能够使用大型语言模型（LLM）处理自然语言处理任务。虽然LangChain支持更复杂的数据结构，但它目前缺乏标准的可互操作数据类型。这意味着LangChain在处理数据时可能需要进行一些额外的处理和转换。开发人员可以根据自己的需求使用LangChain提供的功能和工具来处理和操作数据。


10. LangChain 替代方案？

💡 LangChain是一个独特的框架，目前没有直接的替代方案。它提供了一种简化开发过程的方式，使开发人员能够更轻松构建基于语言模型的应用程序。LangChain的特点包括编写自定义的LangChain提示和链式代码的语法糖、使用IDE内置的支持进行提示和类型检查、支持可选参数和共享参数等。虽然可能有其他类似的框架可用，但LangChain在其特定领域内提供了独特的功能和优势。

基于LLM+向量库的文档对话经验面


- 1. 基于LLM+向量库的文档对话 基础面
  - a. 查询类型：首先确定用户可能的查询类型，例如问题查询、主题查询、摘要查询等。针对不同的查询类型，可以构建相应的prompt模板。例如，对于问题查询，可以使用"我有一个关于XXX的问题"作为模板；对于主题查询，可以使用"我想了解关于XXX的信息"作为模板。
  - b. 查询内容：根据文档的特点和领域知识，确定用户可能会查询的内容。例如，对于新闻文档，查询内容可以包括新闻标题、关键词、时间范围等；对于学术论文，查询内容可以包括作者、论文标题、摘要等。根据查询内容，可以构建相应的prompt模板。例如，对于查询新闻标题的情况，可以使用"请问有关于XXX的新闻吗？"作为模板。
  - c. 上下文信息：考虑上下文信息对于查询的影响。用户之前的查询或系统的回复可能会影响当前的查询。可以将上下文信息加入到prompt模板中，以便更好地理解用户的意图。例如，对于上一轮的回复是关于某个主题的，可以使用"我还有关于上次谈到的XXX的问题"作为模板。
  - d. 可变参数：考虑到用户的查询可能有不同的变化，可以在prompt模板中留出一些可变的参数，以便根据具体查询进行替换。例如，可以使用"我想了解关于XXX的信息"作为模板，其中的XXX可以根据用户的查询进行替换。
  - e. 大语言模型：大语言模型是指能够理解和生成人类语言的深度学习模型，如GPT、BERT等。这些模型通过在大规模文本数据上进行预训练，学习到语言的语义和上下文信息。在文档对话系统中，大语言模型可以用于生成回复、推荐相关文档等任务。
  - f. 文档向量化：文档向量化是将文档表示为数值向量的过程。这可以使用向量库技术，如TF-IDF、Word2Vec、Doc2Vec等。文档向量化的目的是将文档转换为计算机可以处理的数值形式，以便计算文档之间的相似度或进行其他文本分析任务。



- g. 相似度计算：相似度计算是文档对话系统中的重要技术。通过计算查询文本向量与文档向量之间的相似度，可以实现文档的检索和推荐。常见的相似度计算方法包括余弦相似度、欧氏距离等。
- h. 对话生成：对话生成是指根据用户的查询文本生成系统的回复或推荐文档。这可以使用大语言模型来生成自然语言的回复。生成的回复可以基于查询文本的语义和上下文信息，以提供准确和 **继续访问** **取消**
- i. 对话交互：对话交互是指用户和系统之间的交互过程。用户可以提供查询文本，系统根据查询文本生成回复，用户再根据回复提供进一步的查询或反馈。对话交互可以通过迭代和反馈来改进系统的回复和推荐。
- j. 数据预处理：首先，需要对文档数据进行预处理。这包括分词、去除停用词、词干化等步骤，以准备文档数据用于后续的向量化和建模。
- k. 文档向量化：使用向量库的方法，将每个文档表示为一个向量。常见的向量化方法包括TF-IDF、Word2Vec、Doc2Vec等。这些方法可以将文档转换为数值向量，以便计算文档之间的相似度或进行聚类分析。
- l. 大语言模型训练：使用大语言模型，如GPT、BERT等，对文档数据进行训练。这样可以使模型学习到文档之间的语义关系和上下文信息。
- m. 文档检索：当用户提供一个查询文本时，首先对查询文本进行向量化，然后计算查询向量与文档向量之间的相似度。可以使用余弦相似度或其他相似度度量方法来衡量它们之间的相似程度。根据相似度排序，返回与查询文本最相关的文档。
- n. 文档推荐：除了简单的文档检索，还可以使用大语言模型生成推荐文档。通过输入用户的查询文本，使用大语言模型生成与查询相关的文本片段或摘要，并根据这些生成的文本片段推荐相关的文档。
- o. 对话交互：在文档对话系统中，用户可以提供多个查询文本，并根据系统的回复进行进一步的对话交互。可以使用大语言模型生成系统的回复，并根据用户的反馈进行迭代和改进。
- p. 数据清洗和预处理：在训练大语言模型之前，对数据进行仔细的清洗和预处理是至关重要的。删除不准确、噪声或有偏差的数据可以减少模型幻觉问题的出现。
- q. 多样化训练数据：为了减少模型对特定数据源的依赖和偏好，可以尽量使用多样化的训练数据。包括来自不同领域、不同来源和不同观点的数据，以获得更全面的语言理解。
- r. 引入多样性的生成策略：在生成文本时，可以采用多样性的生成策略来减少模型的倾向性和幻觉问题。例如，使用温度参数来调整生成的多样性，或者使用抽样和束搜索等不同的生成方法。
- s. 人工审核和后处理：对生成的文本进行人工审核和后处理是一种常用的方法。通过人工的干预和修正，可以纠正模型幻觉问题，并确保生成的内容准确和可靠。
- t. 引入外部知识和约束：为了提高生成文本的准确性，可以引入外部知识和约束。例如，结合知识图谱、实体识别或逻辑推理等技术，将先验知识和约束融入到生成过程中。
- u.  **LLMs** 存在模型幻觉问题，请问如何处理？

大语言模型的模型幻觉问题是指其可能生成看似合理但实际上不准确或不符合事实的内容。这是由于大语言模型在训练过程中接触到的数据源的偏差、噪声或错误所导致的。处理大语言模型的模型幻觉问题需要采取一些方法和策略，以下是一些建议：

这些方法可以帮助减少大语言模型的模型幻觉问题，但并不能完全消除。因此，在使用大语言模型时，仍然需要谨慎评估生成结果的准确性和可靠性，并结合人工的审核和后处理来确保生成内容的质量。

- v.  基于**LLM**+向量库的文档对话 思路是怎么样？

基于大语言模型和向量库的文档对话可以通过以下实现思路：

通过以上实现思路，可以构建一个基于大语言模型和向量库的文档对话系统，使用户能够方便地进行文档检索、推荐和对话交互。具体的实现细节和技术选择会根据具体的应用场景和需求来确定。

w. 基于LLM+向量库的文档对话 核心技术是什么？

基于大语言模型和向量库的文档对话的核心技术包括以下几个方面：

这些技术共同构成了基于大语言模型和向量库的文档对话系统的核心。通过结合这些技术，可以实现文档的检索、推荐和对话交互，提供更智能和个性化的文档服务。

继续访问

取消

x. 基于LLM+向量库的文档对话的prompt模板如何构建？

构建基于大语言模型和向量库的文档对话的prompt模板可以考虑以下几个方面：

通过这些方面的考虑，可以构建多个不同的prompt模板，以满足不同类型和内容的查询需求。在实际应用中，可以根据具体的场景和数据进行调整和优化，以提供更准确和有针对性的查询模板。

2. 基于LLM+向量库的文档对话 优化面

- a. 数据准备：准备大量高质量的训练数据，包括query、context和对应的高质量response。确保数据的多样性和覆盖性，以提供更好的训练样本。
- b. 模型架构：选择合适的模型架构，如Transformer等，以便提取query和context中的重要信息，并生成相应的高质量response。确保模型具有足够的容量和复杂性，以适应各种复杂的查询和上下文。
- c. 微调和优化：使用预训练的模型作为起点，通过在特定任务上进行微调和优化，使模型能够更好地理解query和context，并生成更准确、连贯的response。可以使用基于强化学习的方法，如强化对抗学习，来进一步提高模型的表现。
- d. 上下文建模：在LLM中，上下文对于生成高质量的response非常重要。确保模型能够准确地理解和利用上下文信息，以生成与之相关的response。可以使用一些技术，如注意力机制和上下文编码器，来帮助模型更好地建模上下文。
- e. 评估和反馈：定期评估模型的性能，使用一些评估指标，如BLEU、ROUGE等，来衡量生成的response的质量。根据评估结果，及时调整和改进模型的训练策略和参数设置。同时，收集用户反馈和意见，以便进一步改进模型的性能。
- f. 多模态信息利用：如果有可用的多模态信息，如图像、视频等，可以将其整合到LLM中，以提供更丰富、准确的response。利用多模态信息可以增强模型的理解能力和表达能力，从而生成更高质量的response。
- g. 引入外部知识和资源：为了提高LLM的质量，可以引入外部知识和资源，如知识图谱、预训练的语言模型等。利用这些资源可以帮助模型更好地理解和回答query，从而生成更高质量的response。
- h. 建立索引：将Document集合建立索引，以便能够快速检索和匹配相关的Document。可以使用搜索引擎或专业的信息检索工具，如Elasticsearch、Solr等。
- i. 关键词匹配：通过对query和Document中的关键词进行匹配，筛选出包含相关关键词的Document。可以使用TF-IDF、BM25等算法来计算关键词的重要性和匹配程度。
- j. 向量化表示：将query和Document转化为向量表示，通过计算它们之间的相似度来判断相关性。可以使用词嵌入模型（如Word2Vec、GloVe）或深度学习模型（如BERT、ELMo）来获取向量表示。
- k. 上下文建模：考虑上下文信息，如query的前后文、Document的上下文等，以更准确地判断相关性。可以使用上下文编码器或注意力机制来捕捉上下文信息。
- l. 扩展查询：根据query的特点，进行查询扩展，引入相关的同义词、近义词、词根变化等，以扩大相关Document的召回范围。
- m. 语义匹配：使用语义匹配模型，如Siamese网络、BERT等，来计算query和Document之间的语义相似度，以更准确地判断相关性。
- n. 实时反馈：利用用户的反馈信息，如点击、收藏、评分等，来优化召回结果。通过监控用户行为，不断调整和优化召回算法，提升相关Document的召回率。
- o. 多模态信息利用：如果有可用的多模态信息，如图像、视频等，可以将其整合到召回模型中，以提供更丰富、准确的相关Document。通过多模态信息的利用，可以增强召回模型的表达能力和准确性。

- p. 调整输入：检查输入的文本是否符合预期的格式和结构。确保输入的句子和段落之间有明确的分隔符，如句号、问号或换行符。如果输入的文本结构不清晰，可能会导致分句效果不佳。
- q. 引入标点符号：在文本中适当地引入标点符号，如句号、问号或感叹号，以帮助模型更好地理解句子的边界。标点符号可以提供明确的分句信号，有助于改善分句的准确性。
- 继续访问 取消
- r. 使用自定义规则：针对特定的文本类型或语言，可以使用自定义规则来分句。例如，可以编写正则表达式或使用特定的分句库来处理特定的分句需求。这样可以更好地适应特定的语言和文本结构。
- s. 结合其他工具：除了Langchain内置的问答分句功能，还可以结合其他分句工具或库来处理文本。例如，NLTK、spaCy等自然语言处理工具包中提供了强大的分句功能，可以与Langchain一起使用，以获得更好的分句效果。
- t. 使用上下文信息：如果上下文信息可用，可以利用上下文信息来辅助分句。例如，可以根据上下文中的语境和语义信息来判断句子的边界，从而提高分句的准确性。
- u. 收集反馈和调整模型：如果您发现Langchain内置的问答分句功能在特定场景下效果不佳，可以收集用户反馈，并根据反馈进行模型调整和改进。通过不断优化模型，可以逐渐改善分句效果。
- v. 针对垂直领域进行领域特定训练：LLM模型是基于大规模通用语料库进行训练的，可能无法充分捕捉垂直领域的特点和术语。可以使用领域特定的语料库对LLM模型进行微调或重新训练，以提高在垂直领域的表现。
- w. 增加领域知识：在向量库中，可以添加垂直领域的专业知识，如领域术语、实体名词等。这样可以提高向量库中文档的表示能力，使其更适应垂直领域的对话需求。
- x. 优化检索算法：在使用向量库进行文档检索时，可以尝试不同的检索算法和相似度计算方法。常用的算法包括余弦相似度、BM25等。通过调整参数和算法选择，可以提高检索的准确性和相关性。
- y. 数据增强和样本平衡：在训练LLM模型时，可以增加垂直领域的样本数据，以增加模型对垂直领域的理解和表达能力。同时，要注意样本的平衡，确保训练数据中包含各个垂直领域的典型对话场景，避免偏向某个特定领域。
- z. 引入外部知识库：在垂直领域的对话中，可以结合外部的领域知识库，如专业词典、行业标准等，来提供更准确的答案和解决方案。通过与外部知识库的结合，可以弥补LLM模型和向量库在垂直领域中的不足。
- aa. 收集用户反馈和迭代优化：通过收集用户的反馈信息，了解用户对对话系统的需求和期望，并根据反馈进行迭代优化。持续改进和优化是提高垂直领域对话效果的关键。
- ab. 预处理和过滤：在进行文档切分之前，可以进行一些预处理和过滤操作，以减少噪声的影响。例如，可以去除文档中的停用词、标点符号、特殊字符等，以及进行拼写纠错和词形还原等操作。这样可以降低噪声的存在，提高文档切分的质量。
- ac. 主题建模：可以使用主题建模技术，如LDA（Latent Dirichlet Allocation）等，对文档进行主题抽取。通过识别文档的主题，可以帮助确定文档切分的粒度。例如，将同一主题下的文档划分为一个切分单元，以保留更多的语义信息。
- ad. 上下文信息：在进行文档切分时，考虑上下文信息对于语义的影响。例如，将与上一文档相关联的文档划分为一个切分单元，以保留上下文的连贯性和语义关联。这样可以更好地捕捉文档之间的语义信息。
- ae. 动态切分：可以采用动态切分的方式，根据用户的查询和需要，实时生成切分单元。例如，根据用户的关键词或查询意图，动态生成包含相关信息的切分单元，以减少噪声和提高语义的准确性。
- af. 实验和优化：在实际应用中，可以进行一系列的实验和优化，通过不断调整和评估文档切分的效果。可以尝试不同的切分粒度，评估其噪声和语义信息的平衡。通过实验和优化，逐步找到合适的文档切分策略。
- ag. 📍 痛点1：文档切分粒度不好把控，既担心噪声太多又担心语义信息丢失

在基于大语言模型和向量库的文档对话中，确实需要在文档切分的粒度上进行权衡。如果切分得太细，可能会引入较多的噪声；如果切分得太粗，可能会丢失一些重要的语义信息。以下是一些解决方案：

综上所述，解决文档切分粒度的问题需要综合考虑预处理、主题建模、上下文信息、动态切分等多个因素，并通过实验和优化来找到最佳的平衡点，以保留足够的语义信息同时减少噪声的影响。

ah.💡 痛点2：在基于垂直领域 表现不佳

如果在垂直领域中，   和向量库的文档对话表现不佳，可以考虑以下方法来改进：

总之，通过领域特定训练、增加领域知识、优化检索算法、数据增强和样本平衡、引入外部知识库以及收集用户反馈和迭代优化等方法，可以改进基于LLM和向量库的文档对话在垂直领域中的表现。这些方法可以根据具体情况灵活应用，以提高对话系统的准确性和适应性。

ai.💡 痛点3：langchain 内置 问答分句效果不佳问题

如果您在使用Langchain内置的问答分句功能时发现效果不佳，可以尝试以下方法来改善：

总之，通过调整输入、引入标点符号、使用自定义规则、结合其他工具、使用上下文信息以及收集反馈和调整模型等方法，可以改善Langchain内置的问答分句效果。这些方法可以根据具体情况灵活使用，以提高分句的准确性和效果。

aj.💡 痛点4：如何 尽可能召回与query相关的Document 问题

要尽可能召回与query相关的Document，可以采取以下方法：

总之，通过建立索引、关键词匹配、向量化表示、上下文建模、查询扩展、语义匹配、实时反馈和多模态信息利用等方法，可以尽可能召回与query相关的Document。这些方法可以单独使用，也可以结合起来，以提高召回的准确性和覆盖率。

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

要让LLM基于query和context得到高质量的response，可以采取以下方法：

总之，通过合适的数据准备、模型架构选择、微调和优化、上下文建模、评估和反馈、多模态信息利用以及引入外部知识和资源等方法，可以帮助LLM基于query和context得到高质量的response。



包包算法笔记

包大人的算法，程序，机器学习，职场，理财闲谈。

113篇原创内容

公众号

相关文章

- 大模型的生产优化
- 大模型Kaggle比赛首秀冠军方案总结
- 大模型RLHF理论详细讲解
- 24家国内大模型面经
- 大模型面试八股含答案
- 大模型无标注对齐RLAIF讲解
- 大模型训练为什么用A100不用4090
- 大模型百川2技术报告细节分享
- 大模型来自面试的一些体会和分享
- 判断场景是否适合大模型
- 大模型微调技术报告汇总
- 大模型的幻觉问题
- 从零训练大模型教程
- 领域/场景大模型也太难训了吧

- 大模型开源社区的原子弹Llama2
- 大模型训练的一些坑点和判断
- 大模型RLHF的trick
- 大模型评测，也太难了吧
- 大模型面试八股
- 大模型微调样本构造的tri
- 大模型训练太难了！

继续访问

取消

其他精彩文章翻阅公众号历史文章

包包算法笔记是包大人在班车通勤上，进行知识，职业，经验分享的地方。最白的话讲专业的知识。  
回复“刷题”获取高效刷题经验，回复“面试”获取算法校招面试宝典，回复“大模型”获取大模型技术资料。

最近高强度写大模型原创！

想进讨论群的同学，加微信号logits，备注进群



收录于合集 #大模型 16

上一篇 · 大模型黑科技:单张16G卡推70B模型

阅读原文

喜欢此内容的人还喜欢

大模型方向的几个论文技巧  
包包算法笔记

