一、大模型常识面

1.1 简单 介绍一下 大模型【LLMs】？

大模型：一般指**1 亿以上参数的模型**，但是这个标准一直在升级，目前万亿参数以上的模型也有了。大语言模型（Large Language Model，LLM）是针对语言的大模型。

1.2 大模型【LLMs】后面跟的 175B、60B、540B 等 指什么？

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。

1.3 大模型【LLMs】具有什么优点？

- 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。

1.4 大模型【LLMs】具有什么缺点？

- 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。

二、大模型强化学习面

2.1 简单介绍强化学习？

强化学习：（Reinforcement Learning）一种机器学习的方法，**通过从外部获得激励来校正学习方向从而获得一种自适应的学习能力**。

2.2 简单介绍一下 RLHF？

基于人工反馈的强化学习（Reinforcement Learning from Human Feedback，RLHF）：**构建人类反馈数据集，训练一个激励模型，模仿人类偏好对结果打分**，这是 GPT-3 后时代大语言模型越来越像人类对话核心技术。

三、大模型【LLMs】微调篇

3.1 大模型【LLMs】泛化问题？

泛化（Generalization）模型泛化是指一些模型可以应用（泛化）到其他场景，通常为采用迁移学习、微调等手段实现泛化。

3.2 大模型【LLMs】微调问题？

微调（FineTuning）针对大量数据训练出来的预训练模型，后期采用业务相关数据进一步训练原先模型的相关部分，得到准确度更高的模型，或者更好的泛化。

通过在一个已经训练好的模型的基微调，可以让模型更专注于新任务或领域的特点，提高模型的性能和准确度。微调也可以让模型更容易适应不同的输入输出格式，以及不同的评估指标。

3.3 大模型【LLMs】微调有哪些优点？

微调是一种常用的迁移学习方法，它可以利用预训练模型的通用知识，同时减少训练时间和数据需求。

在大语言模型上，微调是指在预训练好的大型语言模型基础上，针对特定任务进行额外训练。

这种方法需要对模型进行额外的训练，但可以提高模型在特定任务上的性能。微调通常用于解决那些无法通过提示工程解决的问题。

换句话说：**它通过输入额外的样本，对模型部分参数进行修改，从而强化模型某部分能力。本质上也是一种引导和激发模型能力的方法。**

3.4 大模型【LLMs】指令微调问题？

指令微调（Instruction FineTuning） **针对已经存在的预训练模型，给出额外的指令或者标注数据集来提升模型的性能**。

四、大模型【LLMs】思维链篇

4.1 大模型【LLMs】思维链问题？

思维链（Chain-of-Thought，CoT）：**通过让大语言模型（LLM）将一个问题拆解为多个步骤，一步一步分析，逐步得出正确答案。需指出，针对复杂问题，LLM 直接给出错误答案的概率比较高**。思维链可以看成是一种指令微调。

4.2 大模型【LLMs】思维链本质是什么？

- 思维链的本质：利用模型的生成能力和涌现能力，来解决一些复杂或特殊的问题；即**将复杂任务拆解为多个简单的子任务，它指的是一个思维过程中的连续逻辑推理步骤或关联的序列，是思维过程中一系列相互关联的想法、观点或概念的串联**。思维链通常用于解决问题、做决策或进行推理。它可以按照逻辑顺序连接和组织思维，将复杂的问题分解为更简单的步骤或概念，从而更好地理解和解决问题。
- 适用场景：数学应用题、常识推理、符号操作等

4.3 大模型【LLMs】思维链优点是什么？

思维链可以让模型更好地理解问题的含义和范围，更接近人类的思考方式。

4.4 大模型【LLMs】思维链类型和策略？

- Few-shot 思维链：
- 介绍：即给模型提供一些手动设计的中间步骤或过程，来影响模型的输出；
- 适用场景：这种方法适用于一些相对简单或常见的问题，或者一些模型已经有了很强的涌现能力的问题。
- Zero-shot 思维链：
- 介绍：即让模型自动生成中间步骤或过程，然后再根据这些步骤或过程来生成结果。
- 适用场景：这种方法适用于一些相对复杂或特殊的问题，或者一些模型需要更多的引导和调整的问题。
- Least-to-Most 思维链：
- 介绍：即给模型提供一个由简单到复杂的提示序列，让模型逐渐增加推理难度和深度；
- 适用场景：这种方法可以让模型更有效地利用其通用知识和涌现能力，同时避免过拟合或灾难性遗忘。

4.5 大模型【LLMs】逐步 Zero-shot 介绍？

- 逐步 Zero-shot 介绍：指一种利用大型语言模型进行推理的方法，它**通过让模型自动生成多个中间步骤或过程，然后再根据这些步骤或过程来生成结果**。
- 逐步 Zero-shot 优点：可以让模型更好地理解问题的含义和范围，更接近人类的思考方式。

在论文 Large Language Models are Zero-Shot Reasoners 中，提出了“逐步 Zero-shot”这一思想。。在第一个阶段，模型首先对问题进行拆分并分段解答问题（称为 Reasoning Extraction）。然后，在第二阶段，模型再进行答案的汇总（称为 Answer Extraction）。

4.6 大模型【LLMs】Zero-shot-CoT 提示策略定义？

Zero-shot-CoT 提示策略是指**一种利用 Zero-shot 思维链进行推理的方法**，该方法借助思维链（也被称为思考链，Chain of Thought，CoT）提示法来解决问题。它**通过在问题的结尾附加“Let’s think step by step”这几个词，来引导模型生成一个回答问题的思维链**。**Zero-shot-CoT 提示策略可以让模型更有效地利用其通用知识和涌现能力，同时避免过拟合或灾难性遗忘**。

4.7 大模型【LLMs】Zero-shot-CoT 提示策略应用方法？

- 首先，明确你想要解决的问题或任务，以及你期望的输出或回答的格式和内容；
- 然后，将你的问题或任务用自然语言描述出来，并在结尾加上“Let’s think step by step”这几个词，作为输入或查询给模型；
- 最后，观察模型生成的输出或回答，看是否符合你的期望和需求。如果不满意，可以尝试修改你的输入或查询，或者使用其他控制参数来影响模型的行为。

例如，你可以使用 Zero-shot-CoT 提示策略来解决一个简单的数学应用题：

提示 我去市场买了 10 个苹果。我给邻居 2 个苹果，给修理工 2 个苹果。然后我又买了 5 个苹果，并吃了 1 个。我还剩下多少个苹果？Let's think step by step.

输出 首先，你开始有 10 个苹果。你送走了 2 个苹果给邻居和 2 个苹果给修理工，所以你还剩下 6 个苹果。然后你又买了 5 个苹果，所以现在你有 11 个苹果。最后，你吃了 1 个苹果，所以你还剩下 10 个苹果。

4.8 大模型【LLMs】Few-shot-CoT 提示策略定义？

Few-shot-CoT 提示策略是指一种利用 Few-shot 思维链进行推理的方法，它通过给模型提供一些手动设计的中间步骤或过程，来影响模型的输出。Few-shot-CoT 提示策略可以让模型更专注于新任务或领域的特点，提高模型的性能和准确度。

4.9 大模型【LLMs】Few-shot-CoT 提示策略核心思想是什么？

- Few-shot-CoT 提示策略核心思想：**通过编写思维链样本作为提示词，让模型学会思维链的推导方式，从而更好的完成推导任务。需要在提示样本中不仅给出问题的答案，还同时需要给出问题推导的过程（即思维链），从而让模型学到思维链的推导过程，并将其应用到新的问题中**。Few-shot-CoT 的方式虽然有效，但是并不是很稳定，如果想要得到稳定的正确答案，可能需要更高阶的提示方法。

4.10 大模型【LLMs】Few-shot-CoT 提示策略应用方法是什么？

- 首先，明确你想要解决的问题或任务，以及你期望的输出或回答的格式和内容。
- 然后，将你的问题或任务用自然语言描述出来，并在前面加上一些与任务相关的中间步骤或过程，作为输入或查询给模型。
- 最后，观察模型生成的输出或回答，看是否符合你的期望和需求。如果不满意，可以尝试修改你的输入或查询，或者使用其他控制参数来影响模型的行为。

五、大模型【LLMs】涌现现象篇

5.1 大模型【LLMs】中有一种 涌现现象，你知道么？

涌现（Emergence）或称创发、突现、呈展、演生，是一种现象。许多小实体相互作用后产生了大实体，而这个大实体展现了组成它的小实体所不具有的特性。研究发现，**模型规模达到一定阈值以上后，会在多步算术、大学考试、单词释义等场景的准确性显著提升，称为涌现**。

5.2 大模型【LLMs】涌现现象主要体现在哪些方面？

- **In Context Learning（"Few-Shot Prompt"）**，即用户给出几个例子，LLM 不需要调整模型参数，就能够处理好任务。例如，用户给出几个情感计算的例子，LLM 就能够根据文本判断情感倾向；
- **Augmented Prompting Strategies**，即用户使用一些特殊的手段来引导或激发 LLM 的涌现能力。例如，用户使用多步推理（chain-of-thought prompting）来让 LLM 进行复杂的逻辑推理；用户使用指令（instructions）来描述任务，而不使用少量示例（few-shot exemplars）来让 LLM 进行指令跟随（instruction following）；用户使用程序语言（programming language）来让 LLM 进行程序执行（program execution）；
- **Zero-Shot or Few-Shot Learning**，即 LLM 能够在没有任何或极少量的训练数据的情况下，解决一些从未见过或者很少见过的问题。例如，LLM 能够根据表情符号解码电影名；LLM 能够模拟 Linux 计算机终端并执行一些简单的数学计算程序。

5.3 大模型【LLMs】涌现现象主激活方式？

- 增加模型的规模，即增加模型中参数的数量和复杂度。这可以让模型更好地建立单词之间的联系，更接近人类语言的水平。一般来说，模型规模越大，涌现能力越强；
- 增加数据的规模，即增加模型训练所用的文本数据的数量和质量。这可以让模型学习到更多的知识和信息，更全面地覆盖各种领域和场景。一般来说，数据规模越大，涌现能力越广；

- 改进模型的架构和训练方法，即使用更先进和有效的神经网络结构和优化算法来构建和训练模型。这可以让模型更灵活和高效地处理各种任务和问题。一般来说，模型架构和训练方法越优秀，涌现能力越稳定；

- 使用合适的提示（prompt）和反馈（feedback），即根据任务和问题的特点，设计合理和有效的输入输出格式和内容，以及及时和准确的评估指标和反馈机制。这可以让模型更容易和准确地理解用户的意图和需求，并给出满意的回答。一般来说，提示和反馈越合适，涌现能力越明显。

六、大模型【LLMs】提示工程篇

6.1 大模型【LLMs】提示工程 是什么？

提示工程是指**通过设计特殊的提示来激发模型的涌现能力**。

这种方法不需要对模型进行额外的训练，只需要通过设计合适的提示来引导模型完成特定任务。提示工程通常用于在不更新模型参数的情况下，快速解决新问题。

通过输入更加合理的提示，引导模型进行更有效的结果输出，本质上一种引导和激发模型能力的方法

6.2 提示工程 如何添加进 大模型【LLMs】？

目前为大模型添加 prompt 的方式越来越多，主要表现出的一个趋势是，**相比于普通的 few-shot 模式（只有输入输出）的 prompt 方式，新的方法会让模型在完成任务的过程中拥有更多的中间过程**，例如一些典型的方法：思维链（Chain of Thought）、寄存器（Scratchpad）等等，通过细化模型的推理过程，提高模型的下游任务的效果

6.3 微调（FineTuning） vs 提示工程？

- 微调（FineTuning） 优缺点：

- 微调的优势:是可以让模型更专注于新任务或领域的特点，提高模型的性能和准确度。微调也可以让模型更容易适应不同的输入输出格式，以及不同的评估指标。

- 微调的劣势：是需要修改模型的参数，这可能会导致模型过拟合或灾难性遗忘。过拟合是指模型过度适应新数据，而忽略了预训练模型的通用知识。灾难性遗忘是指模型在学习新数据时，丢失了预训练模型的原有知识。微调也需要一定量的新数据和计算资源，这可能会增加成本和时间。

- 提示工程优缺点：

- 提示工程的优势：是不需要修改模型的参数，只需要设计合适的输入或查询，就可以引导模型产生期望的输出或回答。提示工程可以保留模型的通用知识和涌现能力，同时减少数据和计算资源的需求。

- 提示工程的劣势：是需要创造性和细致性，设计有效的提示并不容易。提示工程也可能受到模型本身的限制，无法解决一些复杂或特殊的问题。提示工程还需要不断地测试和优化，以找到最佳的提示础上，使用少量的新数据来调整模型的参数，以适应新的任务或领域的过程。

6.4 微调（FineTuning） vs 提示工程 在应用场景中关系雨联系？

对于这两种方法各自有各自使用的应用场景：

- 提示工程解决的问题，往往不会用微调（如小语义空间内的推理问题）；
- 微调解决的问题，解决那些无法通过特征工程解决的问题。

它们更多的时候是作为上下游技术关系，例如要进行本地知识库的定制化问答，最好的方法就是借助提示工程进行数据标注，然后再利用标注好的数据进行微调。

6.5 代码提示工程的核心概念

代码提示工程的核心概念是指使用特定的文本或代码输入来引导生成式人工智能模型产生期望的代码输出的过程。这种方法不需要对模型进行额外的训练，只需要通过设计合适的代码提示来引导模型完成特定任务，代码提示工程通常用于解决那些无法通过语言提示工程解决的问题，也是模型开发中的重中之重。

代码提示工程需要以下几个步骤：

- 明确任务的规范，即描述用户想要实现的功能和目标，以及限制和要求。
- 提供相关的上下文，即给出一些与任务相关的背景信息和示例，以帮助模型理解任务的含义和范围。
- 选择合适的模型，即根据任务的难度和复杂度，选择一个适合生成代码的人工智能模型，例如 GPT-4 或 Codex
- 设计有效的提示，即使用合适的词语、短语、符号和格式，来激发模型的生成能力和创造力。
- 测试和优化提示，即在不同的数据和场景下，检验提示的效果和质量，并根据反馈进行调整和改进。

代码提示工程是一个有趣且有用的技能，对于任何使用生成式人工智能模型的人都非常有益。它可以让用户从人工智能模型中获得最大的收益，通过设计提示来产生理想的代码。

6.6 大模型【LLMs】Zero-shot 提示方法 是什么？

- Zero-shot 提示方法 介绍：给模型提供一个不属于训练数据的提示，但模型可以生成你期望的结果的方法；
- Zero-shot 提示方法 优点：使得大型语言模型可以用于许多任务；

eg：可以给模型一个提示，让它翻译一句话，或者给出一个词的定义，或者生成一首诗

- Zero-shot 提示方法 使用方式：通过输入一些类似问题和问题答案，让模型参考学习，并在同一个 prompt 的末尾提出新的问；
- zero-shot 可以理解为：**不给大模型任何的提示，直接提问，让大模型自己做决策**。

6.7 大模型【LLMs】Few-shot 提示方法 是什么？

- Few-shot 提示方法 介绍：给模型提供一些示例或上下文，来引导模型更好地完成任务的方法。这些示例或上下文可以作为模型的条件，来影响后续的输出；

eg：给模型一些情感分析的示例，让它根据文本判断情感倾向，或者给模型一些编程任务的示例，让它生成代码片段

- 简单理解为：**在提问之前，先给大模型一个示例和解释让它学习和模仿，从而在一定程度上赋予它泛化能力**。

6.8 大模型【LLMs】One-shot 和 Few-shot 提示策略？

- One-shot 或者 Few-shot 提示方法的思想：最简单的提示工程的方法就是**通过输入一些类似问题和问题答案，让模型参考学习，并在同一个 prompt 的末尾提出新的问题，以此提升模型的推理能力**。

6.9 大模型【LLMs】One-shot 和 Few-shot 应用场景？

- One-shot 提示策略：
- 介绍：只给模型一个示例或上下文的方法；
- 适用场景：适用于一些相对简单或常见的任务，或者一些模型已经有了很强的涌现能力的任务；
- 举例说明：可以给模型一个新词和它的定义，然后让它用这个新词造句。
- Few-shot 提示策略：
- 介绍：给模型多个示例或上下文的方法。
- 适用场景：适用于一些相对复杂或特殊的任务，或者一些模型需要更多的引导和调整的任务。
- 举例说明：你可以给模型几个不同类型的笑话，然后让它根据一个关键词生成一个新的笑话。

具体的应用来说，**Few-shot 提示方法并不复杂，只需要将一些类似的问题的问题+答案作为 prompt 的一部分进行输入即可**。

当需要输入多段问答作为提示词时，以 Q 作为问题的开头、A 作为回答的开头（也可以换成“问题”、“答案”），并且不同的问答对话需要换行以便于更加清晰的展示，具体方法是通过转义符+换行来完成。

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？

一般 n B 的模型，最低需要 16-20 n G 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。

2. 为什么 SFT 之后感觉 LLM 傻了？

- 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 。

3. SFT 指令微调数据 如何构建？

- **代表性**。应该**选择多个有代表性的任务**；
- **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**；
- **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**。

4. 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。

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？

- 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。

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？

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。

7. 进行 SFT 操作的时候，基座模型选用 Chat 还是 Base？

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。

8. 领域模型微调 指令&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. 领域模型微调 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。

10. 领域模型词表扩增是不是有必要的？

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。

11. 如何训练自己的大模型？

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。

12. 训练中文大模型有啥经验？

链家技术报告《Towards Better Instruction Following Language Models for Chinese: Investigating the Impact of Training Data and Evaluation》中，介绍了开源模型的训练和评估方法：

Table 1: A simple overview of public available chat models. More details could be found in 2.3.

Project	Base model	Training	Training data	Evaluation data	Evaluation method
Stanford alpaca	LLaMA	Full-parameter finetuning	52K text-davinci-003 generated instruction data	252 samples from self-instruct evaluation dataset	Human evaluation
LLaMA-GPT4	LLaMA	Full-parameter finetuning	52K GPT-4 generated instruction data	1. 252 user-oriented instructions 2. 80 vicuna test samples	1. Human evaluation 2. Automatic GPT-4 evaluation
Vicuna	LLaMA	Full-parameter finetuning	70K user-shared conversations with ChatGPT	80 vicuna test samples	Automatic GPT-4 evaluation
Koala	LLaMA	Full-parameter finetuning	1. Stanford alpaca 2. Anthropic HH 3. OpenAI webgpt 4. OpenAI summarization	1. 180 samples from self-instruct evaluation dataset 2. 180 real user queries that were posted online	Human evaluation
Dolly	GPT-J	Full-parameter finetuning	Stanford alpaca	-	Case demonsrtration
Dolly 2.0	Pythia	Full-parameter finetuning	15k human-written instruction data	-	Case demonsrtration
Baize	LLaMA	LoRA	15K ChatGPT generated multi-turn conversations	-	Case demonsrtration

还对比了各因素的消融实验：

Factor	Base model	Training data	Score_w/o_others
词表扩充	LLaMA-7B-EXT	zh(alpaca-3.5&4) + sharegpt	0.670
	LLaMA-7B	zh(alpaca-3.5&4) + sharegpt	0.652
数据质量	LLaMA-7B-EXT	zh(alpaca-3.5)	0.642
	LLaMA-7B-EXT	zh(alpaca-4)	0.693
数据语言分布	LLaMA-7B-EXT	zh(alpaca-3.5&4)	0.679
	LLaMA-7B-EXT	en(alpaca-3.5&4)	0.659
	LLaMA-7B-EXT	zh(alpaca-3.5&4) + sharegpt	0.670
	LLaMA-7B-EXT	en(alpaca-3.5&4) + sharegpt	0.668
数据规模	LLaMA-7B-EXT	zh(alpaca-3.5&4) + sharegpt	0.670
	LLaMA-7B-EXT	zh(alpaca-3.5&4) + sharegpt + BELLE-0.5M-CLEAN	0.762
-	ChatGPT	-	0.824

消融实验结论：

- 扩充中文词表后，可以增量模型对中文的理解能力，效果更好
- 数据质量越高越好，而且数据集质量提升可以改善模型效果
- 数据语言分布，加了中文的效果比不加的好
- 数据规模越大且质量越高，效果越好，大量高质量的微调数据集对模型效果提升最明显。解释下：数据量在训练数据量方面，数据量的增加已被证明可以显著提高性能。值得注意的是，如此巨大的改进可能部分来自 belle-3.5 和我们的评估数据之间的相似分布。评估数据的类别、主题和复杂性将对评估结果产生很大影响
- 扩充词表后的 LLaMA-7B-EXT 的评估表现达到了 0.762/0.824=92%的水平

他们的技术报告证明中文大模型的训练是可行的，虽然与 ChatGPT 还有差距。这里需要指出后续 RLHF 也很重要，我罗列在这里，抛砖引玉。

13. 指令微调的好处？

有以下好处：

- 对齐人类意图，能够理解自然语言对话（更有人情味）
- 经过微调（fine-tuned），定制版的 GPT-3 在不同应用中的提升非常明显。OpenAI 表示，它可以让不同应用的准确度能直接从 83%提升到 95%、错误率可降低 50%。解小学数学题目的正确率也能提高 2-4 倍。（更准）

踩在巨人的肩膀上、直接在 1750 亿参数的大模型上微调，不少研发人员都可以不用再重头训练自己的 AI 模型了。（更高效）

14. 预训练和微调哪个阶段注入知识的？

预训练阶段注入知识的，微调是在特定任务训练，以使预训练模型的通用知识跟特定任务的要求结合，使模型在特定任务上表现更好。

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？

可以使用预训练和微调相结合的方式，先用篇章数据进行预训练以获取广泛的知识，再用问答对数据进行微调，使模型更好的学习到特定领域的知识。

当然，GPT 大模型的预训练和微调，从实现方式来讲是没有什么差别的，都是 decoder only 的语言模型训练并更新参数，如果样本集小，没有大量的篇章文档数据，我认为只进行微调也能注入知识的，不必太纠结预训练。而且特定领域跟预训练模型的分布差别不大，也不用二次预训练。

16. 多轮对话任务如何微调模型？

跟正常的指令微调方法一样，只是组织数据的时候把历史聊天记录拼接起来，作为上下文信息喂给模型。

{"prompt": "长城 h3 风扇不转。继电器好的。保险丝好的传感器新的风扇也新的这是为什么。就是继电器缺一个信号线", "response": "用电脑能读数据流吗？水温多少", "history": []}

{"prompt": "95", "response": "上下水管温差怎么样啊？空气是不是都排干净了呢？", "history": [["长城 h3 风扇不转。继电器好的。保险丝好的传感器新的风扇也新的这是为什么。就是继电器缺一个信号线", "用电脑能读数据流吗？水温多少"]]}

{"prompt": "是的。上下水管都好的", "response": "那就要检查线路了，一般风扇继电器是由电脑控制吸合的，如果电路存在断路，或者电脑坏了的话会出现继电器不吸合的情况！", "history": [["长城 h3 风扇不转。继电器好的。保险丝好的传感器新的风扇也新的这是为什么。就是继电器缺一个信号线", "用电脑能读数据流吗？水温多少"], ["95", "上下水管温差怎么样啊？空气是不是都排干净了呢？"]]}

17. 微调后的模型出现能力劣化，灾难性遗忘是怎么回事？

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。

我理解 ChatGLM-6B 模型是走完 “预训练-SFT-RLHF” 过程训练后的模型，其 SFT 阶段已经有上千指令微调任务训练过，现在我们只是新增了一类指令数据，相对大模型而已，微调数据量少和微调任务类型单一，不会对其原有的能力造成大的影响，所以我认为是不会导致灾难性遗忘问题，我自己微调模型也没出现此问题。

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。

18. 微调模型需要多大显存？

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

19. 大模型 LLM 进行 SFT 操作的时候在学习什么？

（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。

20. 预训练和 SFT 操作有什么不同

下面使用一个具体的例子进行说明。

问题：描述计算机主板的功能

回答：计算机主板是计算机中的主要电路板。它是系统的支撑。

进行预训练的时候会把这句话连接起来，用前面的词来预测后面出现的词。在计算损失的时候，问句中的损失也会被计算进去。

进行 SFT 操作则会构建下面这样一条训练语料。

输入：描述计算机主板的功能*[BOS]*计算机主板是计算机中的主要电路板。它是系统的支撑。*[EOS]*

标签：*[.....][BOS]*计算机主板是计算机中的主要电路板。它是系统的支撑。*[EOS]*

其中[BOS]和[EOS]是一些特殊字符，在计算损失时，只计算答句的损失。在多轮对话中，也是一样的，所有的问句损失都会被忽略，而只计算答句的损失。

因此 SFT 的逻辑和原来的预训练过程是一致的，但是通过构造一些人工的高质量问答语料，可以高效地教会大模型问答的技巧。

21. 样本量规模增大，训练出现 OOM 错

- 问题描述：模型训练的样本数量从 10 万，增大 300 万，训练任务直接报 OOM 了。
- 解决方案，对数据并行处理，具体实现参考海量数据高效训练，核心思想自定义数据集本次的主要目标是使向量化耗时随着处理进程的增加线性下降，训练时数据的内存占用只和数据分段大小有关，可以根据数据特点，灵活配置化。核心功能分为以下几点:
- 均分完整数据集到所有进程（总的 GPU 卡数）
- 每个 epoch 训练时整体数据分片 shuffle 一次，在每个进程同一时间只加载单个分段大小数据集
- 重新训练时可以直接加载向量化后的数据。

22. 大模型 LLM 进行 SFT 如何对样本进行优化？

- 对于输入历史对话数据进行左截断，保留最新的对话记录。
- 去掉样本中明显的语气词，如嗯嗯，啊啊之类的。
- 去掉样本中不合适的内容，如 AI 直卖，就不应出现转人工的对话内容。
- 样本中扩充用户特征标签，如年龄，性别，地域，人群等

23. 模型参数迭代实验

验证历史对话轮次是否越长越好，通过训练两个模型，控制变量 max_source_length丨max_target_length，对训练好之后的模型从 Loss、Bleu 指标、离线人工评估等角度进行对比分析。

结论：从人工评估少量样本以及 loss 下降来看，历史对话长度 1024 比 512 长度好，后续如果训练可能上线模型，可以扩大到 1024 长度。

基于 *LLM*+向量库的文档对话 经验面

一、基于 LLM+向量库的文档对话 基础面

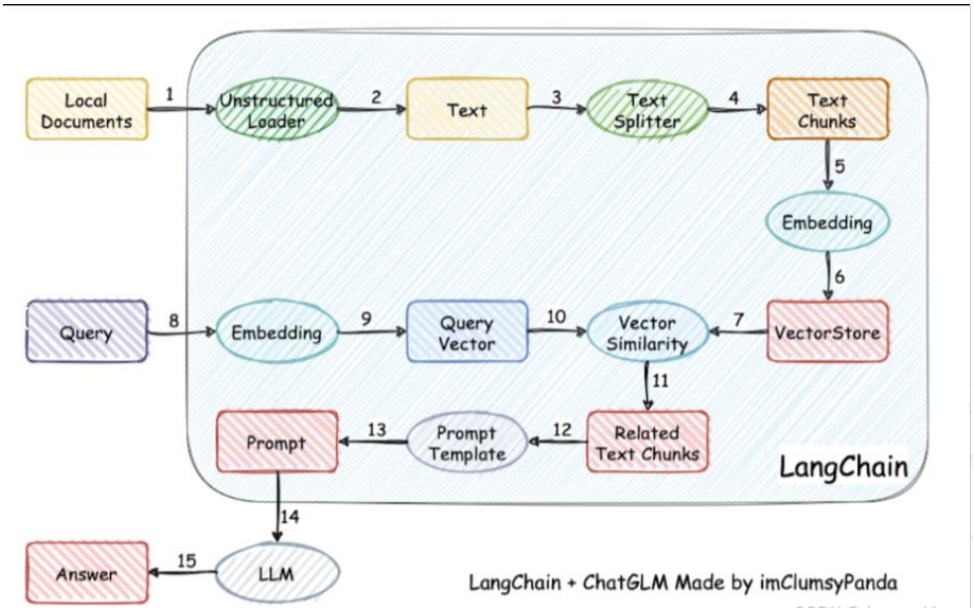
1.1 LLMs 存在模型幻觉问题，请问如何处理？

检索+LLM。先用问题在领域数据库里检索到候选答案，再用 LLM 对答案进行加工。

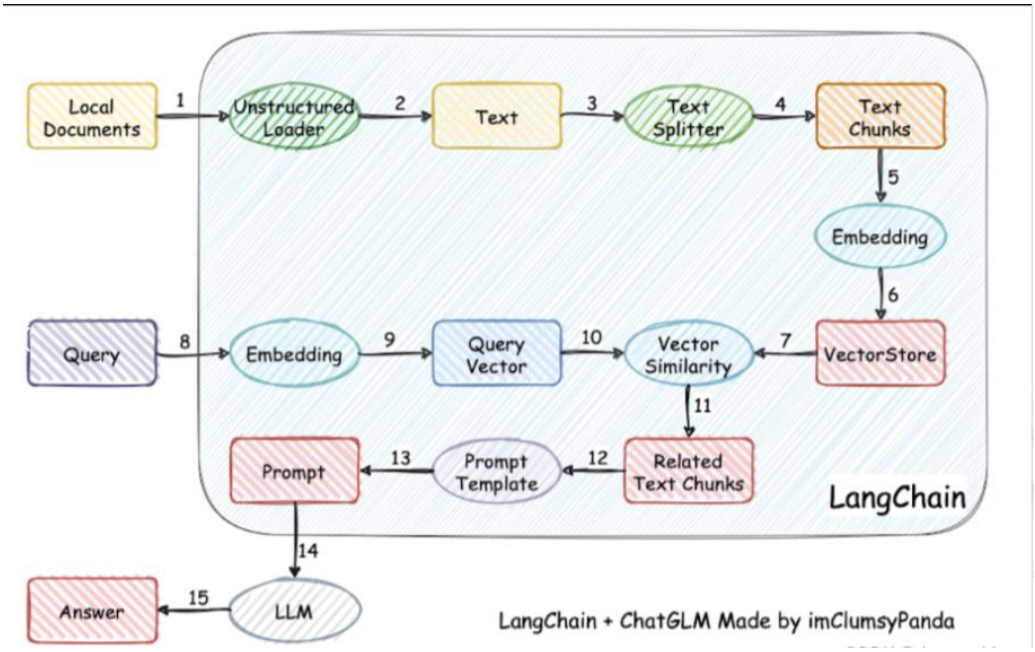
1.2. 基于 LLM+向量库的文档对话 思路是怎么样？

- 加载文件
- 读取文本
- 文本分割
- 文本向量化
- 问句向量化
- 在文本向量中匹配出与问句向量最相似的 top k 个
- 匹配出的文本作为上下文和问题一起添加到 prompt 中
- 提交给 LLM 生成回答

> 版本一



> 版本二



1.3. 基于 LLM+向量库的文档对话 核心技术是什么？

- 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 promt 提交给 LLM 回答

1.4. 基于 LLM+向量库的文档对话 prompt 模板 如何构建？

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}

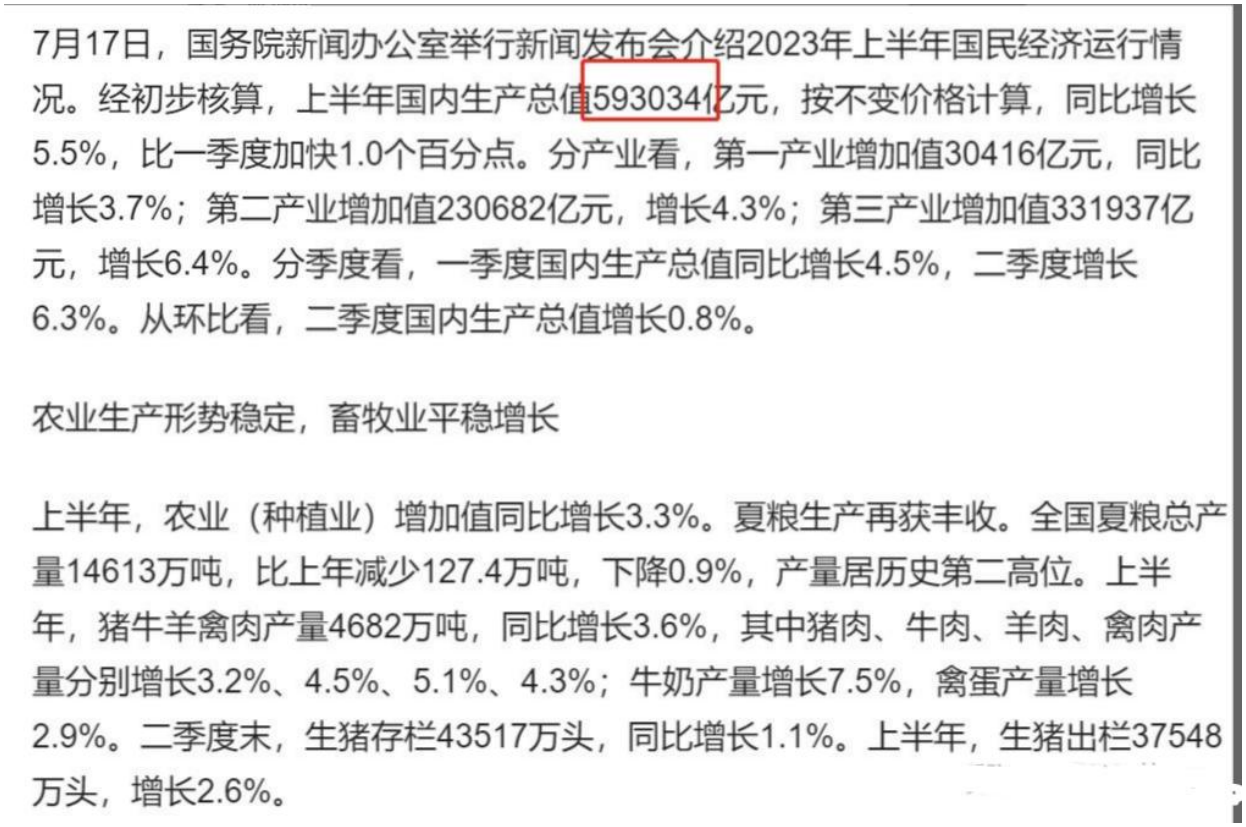
二、基于 LLM+向量库的文档对话 优化面

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

问题描述

问题 1：如何让 LLM 简要、准确回答细粒度知识？

- 举例及标答如下：



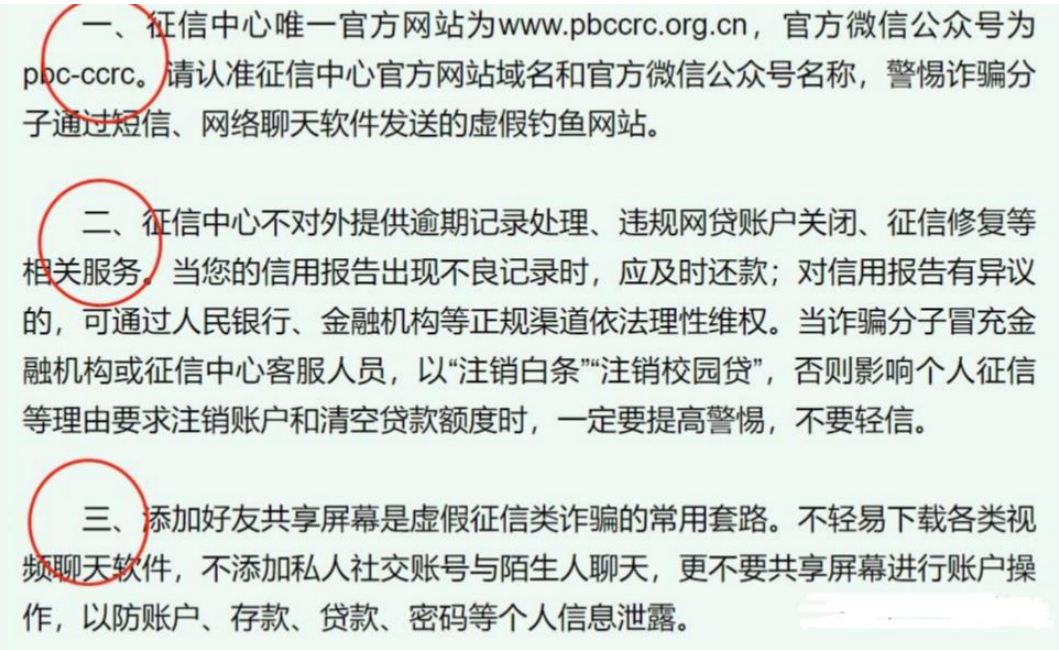
用户：2023 年我国上半年的国内生产总值是多少？

LLM：根据文档，2023 年的国民生产总值是 593034 亿元。

- 需求分析：一是简要，不要有其他废话。二是准确，而不是随意编造。

问题 2：如何让 LLM 回答出全面的粗粒度（跨段落）知识？

- 举例及标答如下：



用户：根据文档内容，征信中心有几点声明？

LLM：根据文档内容，有三点声明，分别是：一、……；二……；三……。

- 需求分析：

要实现语义级别的分割，而不是简单基于 html 或者 pdf 的换行符分割。

笔者发现目前的痛点是**文档分割不够准确，导致模型有可能只回答了两点，而实际上是因为向量相似度召回的结果是残缺的**。

有人可能会问，那完全**可以把切割粒度大一点，比如每 10 个段落一分。但这样显然不是最优的，因为召回片段太大，噪声也就越多**。LLM 本来就有幻觉问题，回答得不会很精准（笔者实测也发现如此）。

所以说，我们的文档切片最好是按照语义切割。

解决方案

思想（原则）

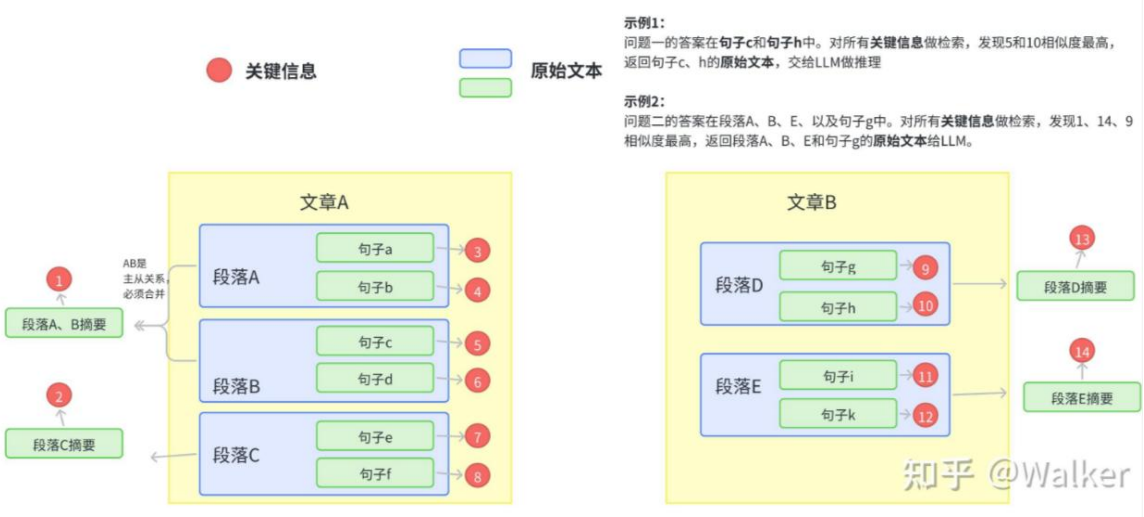
基于 LLM 的文档对话架构分为两部分，**先检索，后推理**。重心在检索（推荐系统），推理交给 LLM 整合即可。

而检索部分要满足三点 **①尽可能提高召回率，②尽可能减少无关信息；③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。

- 主要架构图如下：



如何构建关键信息？

首先从架构图可以看到，句子、段落、文章都要关键信息，如果为了效率考虑，可以不用对句子构建关键信息。

- 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（pdf 多级标题识别及提取见下一篇文章）
- **语义切分方法 1：利用 NLP 的篇章分析（discourse parsing）工具，提取出段落之间的主要关系**，譬如上述极端情况 2 展示的段落之间就有从属关系。把所有包含主从关系的段落合并成一段。 这样对文章切分完之后保证每一段在说同一件事情.
- **语义切分方法 2：**除了 discourse parsing 的工具外，还可以写一个简单算法**利用 BERT 等模型来实现语义分割**。BERT 等模型在预训练的时候采用了 NSP（next sentence prediction）的训练任务，因此 BERT 完全可以判断两个句子（段落）是否具有语义衔接关系。这里我们可以设置相似度阈值 t，从前往后依次判断相邻两个段落的相似度分数是否大于 t，如果大于则合并，否则断开。当然算法为了效率，可以采用二分法并行判定，模型也不用很大，笔者用 BERT-base-Chinese 在中文场景中就取得了不错的效果。

```
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（句子）关键信息抽取

如果向量检索效率很高，获取语义段之后完全可以按照真实段落及句号切分，以缓解细粒度知识点检索时大语块噪声多的场景。当然，关键信息抽取笔者还有其他思路。

- 方法 1：**利用 NLP 中的成分句法分析（constituency parsing）工具和命名实体识别（NER）工具提取**
- 成分句法分析（constituency parsing）工具：可以提取核心部分（名词短语、动词短语……）；
- 命名实体识别（NER）工具：可以提取重要实体（货币名、人名、企业名……）。

譬如说：

原始文本：MM 团队的成员都是精英，核心成员是前谷歌高级产品经理张三，前 meta 首席技术官李四……

关键信息：（MM 团队，核心成员，张三，李四）

- 方法 2：可以用**语义角色标注（Semantic Role Labeling）来分析句子的谓词论元结构**，提取“谁对谁做了什么”的信息作为关键信息。
- 方法 3：直接法。其实 NLP 的研究中本来就有**关键词提取工作（Keyphrase Extraction）**。也有一个成熟工具可以使用。一个工具是 HanLP ，中文效果好，但是付费，免费版调用次数有限。还有一个开源工具是 KeyBERT，英文效果好，但是中文效果差。
- 方法 4：垂直领域建议的方法。以上两个方法在垂直领域都有准确度低的缺陷，垂直领域可以仿照 ChatLaw 的做法，即：训练一个生成关键词的模型。ChatLaw 就是训练了一个 KeyLLM。

常见问题

- 句子、语义段、之间召回不会有包含关系吗，是否会造成冗余？

回答：会造成冗余，但是笔者试验之后回答效果很好，无论是细粒度知识还是粗粒度（跨段落）知识准确度都比 Longchain 粗分效果好很多，对这个问题笔者认为可以优化但没必要

痛点 2：在基于垂直领域 表现不佳

模型微调：一个是对 embedding 模型的基于垂直领域的数据进行微调；一个是对 LLM 模型的基于垂直领域的数据进行微调；

痛点 3：langchain 内置 问答分句效果不佳问题

- 文档加工：
- 一种是使用更好的文档拆分的方式（如项目中已经集成的达摩院的语义识别的模型及进行拆分）；
- 一种是改进填充的方式，判断中心句上下文的句子是否和中心句相关，仅添加相关度高的句子；
- 另一种是文本分段后，对每段分别及进行总结，基于总结内容语义及进行匹配；

痛点 4：如何 尽可能召回与 query 相关的 Document 问题

- 问题描述：如何通过得到 query 相关性高的 context，即与 query 相关的 Document 尽可能多的能被召回；
- 解决方法：
- 将本地知识切分成 Document 的时候，需要考虑 Document 的长度、Document embedding 质量和被召回 Document 数量这三者之间的相互影响。在文本切分算法还没那么智能的情况下，本地知识的内容最好是已经结构化比较好了，各个段落之间语义关联没那么强。Document 较短的情况下，得到的 Document embedding 的质量可能会高一些，通过 Faiss 得到的 Document 与 query 相关度会高一些。
- 使用 Faiss 做搜索，前提条件是有高质量的文本向量化工具。因此最好是能基于本地知识对文本向量化工具进行 Finetune。另外也可以考虑将 ES 搜索结果与 Faiss 结果相结合。

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

- 问题描述：如何让 LLM 基于 query 和 context 得到高质量的 response
- 解决方法：
- 尝试多个的 prompt 模版，选择一个合适的，但是这个可能有点玄学
- 用与本地知识问答相关的语料，对 LLM 进行 Finetune。

langchain 面

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。

2. LangChain 包含哪些 核心概念？

2.1 LangChain 中 Components and Chains 是什么？

- **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。

2.2 LangChain 中 Prompt Templates and Values 是什么？

- **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。PromptValues 是具有方法的类，这些方法可以转换为每个模型类型期望的确切输入类型（如文本或聊天消息）。

2.3 LangChain 中 Example Selectors 是什么？

- 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**。

2.4 LangChain 中 Output Parsers 是什么？

- 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。

2.5 LangChain 中 Indexes and Retrievers 是什么？

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。

2.6 LangChain 中 Chat Message History 是什么？

- 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

2.7 LangChain 中 Agents and Toolkits 是什么？

- 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；
- Tookits ：一组工具，当它们一起使用时，可以完成特定的任务。代理执行器负责使用适当的工具运行代理。

通过理解和利用这些核心概念，您可以利用 LangChain 的强大功能来构建适应性强、高效且能够处理复杂用例的高级语言模型应用程序。

3. 什么是 LangChain Agent?

- 介绍：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。

4. 如何使用 LangChain ？

要使用 LangChain，开发人员首先要导入必要的组件和工具，例如 LLMs, chat models, agents, chains, 内存功能。这些组件组合起来创建一个可以理解、处理和响应用户输入的应用程序。

5. LangChain 支持哪些功能?

- **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。

6. 什么是 LangChain model?

LangChain model 是一种抽象，表示框架中使用的不同类型的模型。LangChain 中的模型主要分为三类：

- **LLM（大型语言模型）**：这些模型将文本字符串作为输入并返回文本字符串作为输出。它们是许多语言模型应用程序的支柱。
- **聊天模型(Chat Model)**：聊天模型由语言模型支持，但具有更结构化的 API。他们将聊天消息列表作为输入并返回聊天消息。这使得管理对话历史记录和维护上下文变得容易。
- **文本嵌入模型(Text Embedding Models)**：这些模型将文本作为输入并返回表示文本嵌入的浮点列表。这些嵌入可用于文档检索、聚类和相似性比较等任务。

开发人员可以为他们的用例选择合适的 LangChain 模型，并利用提供的组件来构建他们的应用程序。

7. LangChain 包含哪些特点?

LangChain 旨在为六个主要领域的开发人员提供支持：

- **LLM 和提示**：LangChain 使管理提示、优化它们以及为所有 LLM 创建通用界面变得容易。此外，它还包括一些用于处理 LLM 的便捷实用程序。
- **链(Chain)**：这些是对 LLM 或其他实用程序的调用序列。LangChain 为链提供标准接口，与各种工具集成，为流行应用提供端到端的链。
- **数据增强生成**：LangChain 使链能够与外部数据源交互以收集生成步骤的数据。例如，它可以帮助总结长文本或使用特定数据源回答问题。
- **Agents**：Agents 让 LLM 做出有关行动的决定，采取这些行动，检查结果，并继续前进直到工作完成。LangChain 提供了代理的标准接口，多种代理可供选择，以及端到端的代理示例。
- **内存**：LangChain 有一个标准的内存接口，有助于维护链或代理调用之间的状态。它还提供了一系列内存实现和使用内存的链或代理的示例。
- **评估**：很难用传统指标评估生成模型。这就是为什么 LangChain 提供提示和链来帮助开发者自己使用 LLM 评估他们的模型。

8.2 LangChain 如何修改 提示模板？

langchain.PromptTemplate: langchain 中的提示模板类

根据不同的下游任务设计不同的 prompt 模板，然后填入内容，生成新的 prompt。目的其实就是为了通过设计更准确的提示词，来引导大模型输出更合理的内容。

8.4 LangChain 如何 Embedding & vector store?

Emebdding 这个过程想必大家很熟悉，简单理解就是把现实中的信息通过各类算法编码成一个高维向量，便于计算机快速计算。

- DL 的语言模型建模一般开头都是 word embedding，看情况会加 position embedding。比如咱们的 LLM 的建模
- 常规检索一般是把 refernce 数据都先 Embedding 入库，服务阶段 query 进来 Embedding 后再快速在库中查询相似 topk。比如 langchain-chatGLM 的本地知识库 QA 系统的入库和检测过程。
- 多模态的方案：同时把语音，文字，图片用不同的模型做 Embedding 后，再做多模态的模型建模和多模态交互。比如这两天的 Visual-chatGLM.

LangChain 存在哪些问题及方法方案?

1. LangChain 低效的令牌使用问题

- 问题：Langchain 的一个重要问题是它的令牌计数功能，对于小数据集来说，它的效率很低。虽然一些开发人员选择创建自己的令牌计数函数，但也有其他解决方案可以解决这个问题。
- 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。它提供了一种简单的方法来计算文本字符串中的令牌，而不需要使用像 Langchain 这样的框架来完成这项特定任务。

2. LangChain 文档的问题

- 问题：文档是任何框架可用性的基石，而 Langchain 因其不充分且经常不准确的文档而受到指责。误导性的文档可能导致开发项目中代价高昂的错误，并且还经常有 404 错误页面。这可能与 Langchain 还在快速发展有关，作为快速的版本迭代，文档的延后性问题

3. LangChain 太多概念容易混淆，过多的“辅助”函数问题

- 问题：Langchain 的代码库因很多概念让人混淆而备受批评，这使得开发人员很难理解和使用它。这种问题的一个方面是存在大量的“helper”函数，仔细检查就会发现它们本质上是标准 Python 函数的包装器。开发人员可能更喜欢提供更清晰和直接访问核心功能的框架，而不需要复杂的中间功能。

简单的分割函数：

```
class CharacterTextSplitter(TextSplitter):
    """Implementation of splitting text that looks at characters."""

def __init__(self, separator: str = "\n\n", **kwargs: Any):
    """Create a new TextSplitter."""
    super().__init__(**kwargs)
    self._separator = separator

def split_text(self, text: str) -> List[str]:
    """Split incoming text and return chunks."""
    # First we naively split the large input into a bunch of smaller ones.
    if self._separator:
        splits = text.split(self._separator)
    else:
        splits = list(text)
```

4. LangChain 行为不一致并且隐藏细节问题

- 问题：LangChain 因隐藏重要细节和行为不一致而受到批评，这可能导致生产系统出现意想不到的问题。

eg: Langchain ConversationRetrievalChain 的一个有趣的方面，它涉及到输入问题的重新措辞。这种重复措辞有时会非常广泛，甚至破坏了对话的自然流畅性，使对话脱离了上下文。

5. LangChain 缺乏标准的可互操作数据类型问题

- 问题：缺乏表示数据的标准方法。这种一致性的缺乏可能会阻碍与其他框架和工具的集成，使其在更广泛的机器学习工具生态系统中工作具有挑战性。

LangChain 替代方案?

是否有更好的替代方案可以提供更容易使用、可伸缩性、活动性和特性。

- LlamaIndex 是一个数据框架，它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。
- 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。

参数高效微调(*PEFT*) 面

1. 微调方法是啥？如何微调？

fine-tune，也叫全参微调，bert 微调模型一直用的这种方法，全部参数权重参与更新以适配领域数据，效果好。

prompt-tune, 包括 p-tuning、lora、prompt-tuning、adaLoRA 等 delta tuning 方法，部分模型参数参与微调，训练快，显存占用少，效果可能跟 FT（fine-tune）比会稍有效果损失，但一般效果能打平。

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

Table 4: Main results. In this table, LLaMA-13B + LoRA(2M) represents a model trained on 2M instruction data using LLaMA-13B as base model and LoRA training method, and LLaMA-7B + FT(2M) represents a model trained using full-parameters fine-tuning. LLaMA-7B + FT(2M) + LoRA(math_0.25M) represents a model trained on 0.25M mathematical instruction data using LLaMA-7B + FT(2M) as the base model and LoRA training method, and LLaMA-7B + FT(2M) + FT(math_0.25M) represents a model trained using incremental full-parameters fine-tuning. About the training time, all these experiments were conducted on 8 NVIDIA A100-40GB GPUs.

Model	Average Score	Additional Param.	Training Time (Hour/epoch)
LLaMA-13B + LoRA(2M)	0.648	28M	10
LLaMA-7B + LoRA(4M)	0.624	17.9M	14
LLaMA-7B + LoRA(2M)	0.609	17.9M	7
LLaMA-7B + LoRA(0.6M)	0.589	17.9M	5
LLaMA-7B + FT(2M)	0.710	-	31
LLaMA-7B + FT(0.6M)	0.686	-	17
LLaMA-7B + FT(2M) + LoRA(math_0.25M)	0.729	17.9M	2
LLaMA-7B + FT(2M) + FT(math_0.25M)	0.738	-	4

peft 的论文《ADAPTIVE BUDGET ALLOCATION FOR PARAMETER- EFFICIENT FINE-TUNING》显示的结果：AdaLoRA 效果稍好于 FT。

Table 1: Results with DeBERTaV3-base on GLUE development set. The best results on each dataset are shown in **bold**. We report the average correlation for STS-B. *Full FT*, *HAdapter* and *PAdapter* represent full fine-tuning, Houlsby adapter, and Pfeiffer adapter respectively. We report mean of 5 runs using different random seeds.

Method	# Params	MNLI m/mm	SST-2 Acc	CoLA Mcc	QQP Acc/F1	QNLI Acc	RTE Acc	MRPC Acc	STS-B Corr	All Ave.
Full FT	184M	89.90/90.12	95.63	69.19	**92.40/89.80**	94.03	83.75	89.46	91.60	88.09
BitFit	0.1M	89.37/89.91	94.84	66.96	88.41/84.95	92.24	78.70	87.75	91.35	86.02
HAdapter	1.22M	90.13/90.17	95.53	68.64	91.91/89.27	94.11	84.48	89.95	91.48	88.12
PAdapter	1.18M	90.33/90.39	95.61	68.77	92.04/89.40	94.29	85.20	89.46	91.54	88.24
LoRA$_{r=8}$	1.33M	90.65/90.69	94.95	69.82	91.99/89.38	93.87	85.20	89.95	91.60	88.34
AdaLoRA	1.27M	**90.76/90.79**	**96.10**	**71.45**	**92.23/89.74**	**94.55**	**88.09**	**90.69**	**91.84**	**89.31**
HAdapter	0.61M	90.12/90.23	95.30	67.87	91.65/88.95	93.76	85.56	89.22	91.30	87.93
PAdapter	0.60M	90.15/90.28	95.53	69.48	91.62/88.86	93.98	84.12	89.22	91.52	88.04
HAdapter	0.31M	90.10/90.02	95.41	67.65	91.54/88.81	93.52	83.39	89.25	91.31	87.60
PAdapter	0.30M	89.89/90.06	94.72	69.06	91.40/88.62	93.87	84.48	89.71	91.38	87.90
LoRA$_{r=2}$	0.33M	90.30/90.38	94.95	68.71	91.61/88.91	94.03	85.56	89.71	**91.68**	88.15
AdaLoRA	0.32M	**90.66/90.70**	**95.80**	**70.04**	**91.78/89.16**	**94.49**	**87.36**	**90.44**	91.63	**88.86**

2 LoRA 篇

2.1 LoRA 权重是否可以合入原模型？

可以，将训练好的低秩矩阵（B*A）+原模型权重合并（相加），计算出新的权重。

2.2 ChatGLM-6B LoRA 后的权重多大？

rank 8 target_module query_key_value 条件下，大约 15M。

2.3 LoRA 微调 特点

- 基础模型的选择对基于 LoRA 微调的有效性有显著影响。
- 训练集越多效果越好
- LoRA 微调的方法在模型参数越大时体现的优势越明显

> 注：此结论参考技术报告《A Comparative Study between Full-Parameter and LoRA-based Fine-Tuning on Chinese Instruction Data for Instruction Following Large Language Model》。

2.4 LoRA 微调方法为啥能加速训练？

- **只更新了部分参数**：比如 LoRA 原论文就选择只更新 Self Attention 的参数，实际使用时我们还可以选择只更新部分层的参数；
- **减少了通信时间**：由于更新的参数量变少了，所以（尤其是多卡训练时）要传输的数据量也变少了，从而减少了传输时间；
- **采用了各种低精度加速技术**，如 FP16、FP8 或者 INT8 量化等。

这三部分原因确实能加快训练速度，然而它们并不是 LoRA 所独有的，事实上几乎都有参数高效方法都具有这些特点。**LoRA 的优点是它的低秩分解很直观，在不少场景下跟全量微调的效果一致，以及在预测阶段不增加推理成本**。

2.5 如何在已有 LoRA 模型上继续训练？

理解此问题的情形是：已有的 lora 模型只训练了一部分数据，要训练另一部分数据的话，是在这个 lora 上继续训练呢，还是跟 base 模型合并后再套一层 lora，或者从头开始训练一个 lora？

我认为把之前的 LoRA 跟 base model 合并后，继续训练就可以，为了保留之前的知识和能力，训练新的 LoRA 时，加入一些之前的训练数据是需要的。另外，每次都重头来成本高。

3. 对比篇

3.1 微调方法批处理大小模式 GPU 显存速度

微调方法批处理大小模式 GPU 显存速度

```
LoRA (r=8) 16 FP16 28GB 8ex/s
LoRA (r=8) 8 FP16 24GB 8ex/s
LoRA (r=8) 4 FP16 20GB 8ex/s
LoRA (r=8) 4 INT8 10GB 8ex/s
LoRA (r=8) 4 INT4 8GB 8ex/s
P-Tuning (p=16) 4 FP16 20GB 8ex/s
P-Tuning (p=16) 4 INT8 16GB 8ex/s
P-Tuning (p=16) 4 INT4 12GB 8ex/s
Freeze (l=3) 4 FP16 24GB 8ex/s
Freeze (l=3) 4 INT8 12GB 8ex/s
```

3.2 Peft 和 全量微调区别？

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

大模型（LLMs）推理面

1. 为什么大模型推理时显存涨的那么多还一直占着？

- 首先，序列太长了，有很多 Q/K/V；
- 其次，因为是逐个预测 next token，每次要缓存 K/V 加速解码。

2. 大模型在 gpu 和 cpu 上推理速度如何？

7B 量级下：

- cpu 推理速度约 10token/s；
- 单卡 A6000 和 8 核 AMD 的推理速度通常为 10:1。

3. 推理速度上，int8 和 fp16 比起来怎么样？

根据实践经验，int8 模式一般推理会明显变慢（huggingface 的实现）

4. 大模型有推理能力吗？

大模型有推理能力。有下面 2 个方面的体现：

ChatGPT 拥有 in-context correction 的能力，即如果说错了，给出矫正，ChatGPT 能“听懂”错在哪儿了，并向正确的方向修正。in-context correction 要比 in-context learning 难了太多，描述越详细清楚，ChatGPT 回答得越好。要知道，越详细的描述，在预训练的文本里越难匹配到的。

在询问 ChatGPT 互联网上并不存在内容的时候，能给出较好答案（如用 ChatGPT 学建模）；ChatGPT 能通过信息猜你心中的想法；你可以制定一个全新的游戏规则让 ChatGPT 和你玩，ChatGPT 可以理解。

5. 大模型生成时的参数怎么设置？

生成模型预测调参建议：

建议去调整下 top_p, num_beams, repetition_renalty, temperature, do_sample=True;

数据生成有重复，调高 repetition_renalty；

生成任务表达单一的，样本也不多的，可适当调低 temperature，生成的样子跟训练集的比较像；如果要复现训练集的效果，temperature=0.01 即可。

以上是经验参数，具体调参根据任务而定，不是固定的。

6. 有哪些省内存的大语言模型训练/微调/推理方法？

- 动机：大模型（LLMs）现在是 NLP 领域的最主流方法之一，但是大模型的训练/微调/推理需要的内存也越来越多。

举例来说，即使 RTX 3090 有着 24GB 的 RAM，是除了 A100 之外显存最大的显卡。但使用一块 RTX 3090 依然无法 fp32 精度训练最小号的 LLaMA-6B。

- Memory-Efficient 的 LLMs 的训练/微调/推理方法
- fp16
- int8
- LoRA
- Gradient checkpointing
- Torch FSDP
- CPU offloading

6.1 如何 估算模型所需的 RAM？

首先，我们需要了解如何根据参数量估计模型大致所需的 RAM，这在实践中有很重要的参考意义。我们需要通过估算设置 batch_size，设置模型精度，选择微调方法和参数分布方法等。

接下来，我们用 LLaMA-6B 模型为例估算其大致需要的内存。

首先考虑精度对所需内存的影响：

- fp32 精度，一个参数需要 32 bits, 4 bytes.
- fp16 精度，一个参数需要 16 bits, 2 bytes.
- int8 精度，一个参数需要 8 bits, 1 byte.

其次，考虑模型需要的 RAM 大致分三个部分：

- 模型参数
- 梯度
- 优化器参数
- 模型参数：等于参数量*每个参数所需内存。

- 对于 fp32，LLaMA-6B 需要 6B*4 bytes = 24GB 内存
- 对于 int8，LLaMA-6B 需要 6B*1 byte = 6GB
- 梯度：同上，等于参数量*每个梯度参数所需内存。
- 优化器参数：不同的优化器所储存的参数量不同。

对于常用的 AdamW 来说，需要储存两倍的模型参数（用来储存一阶和二阶 momentum）。

- fp32 的 LLaMA-6B，AdamW 需要 6B*8 bytes = 48 GB
- int8 的 LLaMA-6B，AdamW 需要 6B*2 bytes = 12 GB

除此之外，CUDA kernel 也会占据一些 RAM，大概 1.3GB 左右，查看方式如下。

```
> torch.ones((1, 1)).to("cuda")
> print_gpu_utilization()
>>>
GPU memory occupied: 1343 MB
```

综上，int8 精度的 LLaMA-6B 模型部分大致需要 6GB+6GB+12GB+1.3GB = 25.3GB 左右。

再根据 LLaMA 的架构（hidden_size = 4096, intermediate_size =11008, num_hidden_layers = 32, context_length = 2048）计算中间变量内存。

每个 instance 需要：

```
(4096 +11008)* 2048 *32 * 1byte = 990MB
```

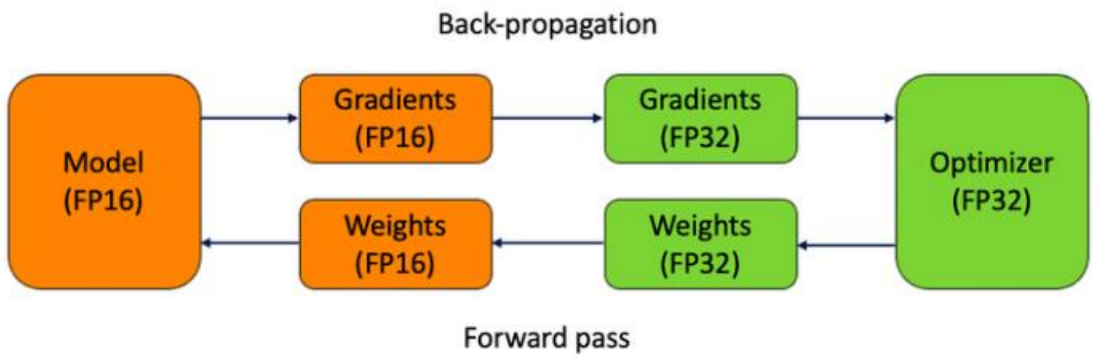
所以一张 A100（80GB RAM）大概可以在 int8 精度，batch_size = 50 的设定下进行全参数训练。

查看消费级显卡的内存和算力：

2023 GPU Benchmark and Graphics Card Comparison Chart：

https://www.gpucheck.com/gpu-benchmark-graphics-card-comparison-chart

6.2 Fp16-mixed precision



混合精度训练的大致思路是在 forward pass 和 gradient computation 的时候使用 fp16 来加速，但是在更新参数时使用 fp32。

用 torch 实现：

CUDA Automatic Mixed Precision examples：https://pytorch.org/docs/stable/notes/amp_examples.html

torch fp16 推理：直接使用 model.half() 将模型转换为 fp16.

```
model.eval()
```

```
model.half()
```

使用 Huggingface Transformers：在 TrainingArguments 里声明

fp16=True https://huggingface.co/docs/transformers/perf_train_gpu_one#fp16-training

6.3 Int8-bitsandbytes

Int8 是个很极端的数据类型，它最多只能表示 - 128～127 的数字，并且完全没有精度。

为了在训练和 inference 中使用这个数据类型，bitsandbytes 使用了两个方法最大程度地降低了其带来的误差：

- vector-wise quantization
- mixed precision decompasition

Huggingface 在这篇文章中用动图解释了 quantization 的实现：https://huggingface.co/blog/hf-bitsandbytes-integration

论文：

LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale：https://arxiv.org/abs/2208.07339

借助 Huggingface PEFT，使用 int8 训练 opt-6.5B 的完整流程：

https://github.com/huggingface/peft/blob/main/examples/int8_training/Finetune_opt_bnb_peft.ipynb

6.4 LoRA

Low-Rank Adaptation 是微调 LLMs 最常用的省内存方法之一。

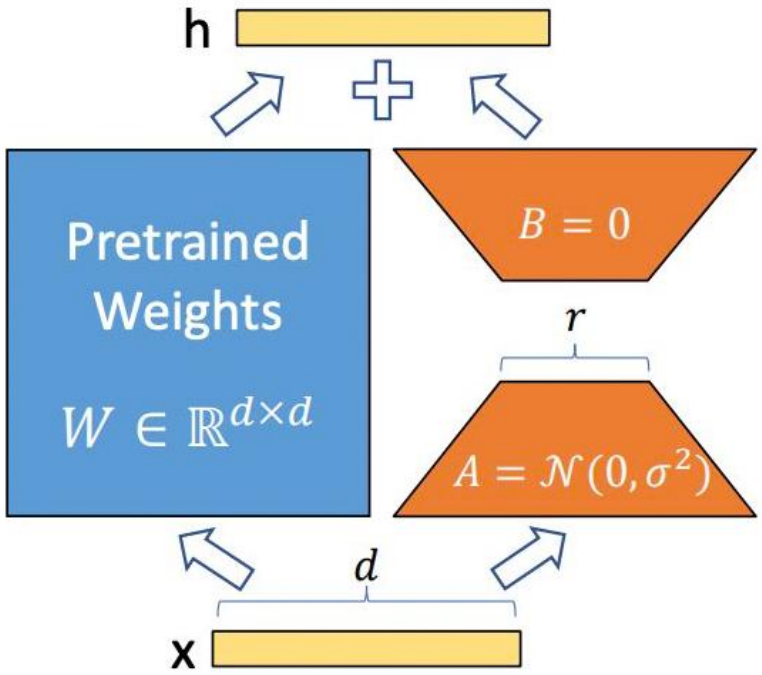


Figure 1: Our reparametrization. We only train A and B.

LoRA 发现再微调 LLMs 时，更新矩阵（update matrix）往往特别 sparse，也就是说 update matrix 是低秩矩阵。LoRA 的作者根据这一特点将 update matrix reparametrize 为两个低秩矩阵的积积 。

其中，，A 和 B 的秩为 r，且 。

如此一来，A+B 的参数量将大大小于 .

LoRA 的论文：https://arxiv.org/pdf/2106.09685.pdf

借助 Huggingface PEFT 框架，使用 LoRA 微调 mt0：

https://github.com/huggingface/peft/blob/main/examples/conditional_generation/peft_lora_seq2seq.ipynb

6.5 Gradient Checkpointing

在 torch 中使用 - 把 model 用一个 customize 的 function 包装一下即可，详见：

Explore Gradient-Checkpointing in PyTorch： https://qywu.github.io/2019/05/22/explore-gradient-checkpointing.html

在 Huggingface Transformers 中使用：

https://huggingface.co/docs/transformers/v4.27.2/en/perf_train_gpu_one#gradient-checkpointing

6.6 Torch FSDP+CPU offload

Fully Sharded Data Paralle（FSDP）和 DeepSpeed 类似，均通过 ZeRO 等分布优化算法，减少内存的占用量。其将模型参数，梯度和优化器状态分布至多个 GPU 上，而非像 DDP 一样，在每个 GPU 上保留完整副本。

CPU offload 则允许在一个 back propagation 中，将参数动态地从 GPU -> CPU, CPU -> GPU 进行转移，从而节省 GPU 内存。

Huggingface 这篇博文解释了 ZeRO 的大致实现方法：https://huggingface.co/blog/zero-deepspeed-fairscale

借助 torch 实现 FSDP，只需要将 model 用 FSDPwarp 一下；同样，cpu_offload 也只需要一行代码：

https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

在这个可以查看 FSDP 支持的模型：https://pytorch.org/docs/stable/fsdp.html

在 Huggingface Transformers 中使用 Torch FSDP：

https://huggingface.co/docs/transformers/v4.27.2/en/main_classes/trainer#transformers.Trainin

根据某些 issue，shard_grad_op（只分布保存 optimizer states 和 gradients）模式可能比 fully_shard 更稳定：

https://github.com/tatsu-lab/stanford_alpaca/issues/32

7. 如何让大模型输出合规化

根据用户的输入问题内容，大模型进行生成回答的内容，但是生成的回答，不直接对外输出给用户。需要进行合规的处理，因为大模型的输出内容不可控，对于严肃的场景，以免引起用户的投诉。所以需要进合并处理。

目前处理的方法，模型生成内容，再把这些内容生成向量，再查询话术向量库，得到最相似的话术。如果查询结果或相似得分比较阈值低或者查询不到结果，则走兜底策略。兜底策略按用户所在的对话阶段，实验不同的兜底话术。或者使用万能兜底话术。

8. 应用模式变更

机器人销售场景的 case:

纯大模型 AI 模式，最初直接是大模型机器人直接和用户对话，全流程都是大模型对话走流程。

对比之前的 AI（小模型意图、话术策略）+人工模式，发现之前的初始阶段通过率高些，初步判断可能是用户说的太发散，大模型不好收敛。

就调整为 AI+大模型 AI 模式。这样前面的 AI 主要是小模型意图、话术策略模式，任务引导更明确。大模型可以更好的和有意向的用户进行交互，更容易引导用户成单。

分布式训练面

1. 理论篇

1.1 想要训练 1 个 LLM，如果只想用 1 张显卡，那么对显卡的要求是什么？

显卡显存足够大，nB 模型微调一般最好准备 **20nGB 以上的显存**。

1.2 如果有 N 张显存足够大的显卡，怎么加速训练？

数据并行（DP），充分利用多张显卡的算力。

1.3 如果显卡的显存不够装下一个完整的模型呢？

最直观想法，需要**分层加载，把不同的层加载到不同的 GPU 上**（accelerate 的 device_map）

也就是常见的 PP，流水线并行。

1.4 PP 推理时，是一个串行的过程，1 个 GPU 计算，其他空闲，有没有其他方式？

- **横向切分：流水线并行（PP）**，也就是分层加载到不同的显卡上。
- **纵向切分：张量并行（TP）**，在 DeepSpeed 世界里叫模型并行（MP）

1.5 3 种并行方式可以叠加吗？

是可以的，**DP+TP+PP**，这就是 3D 并行。**如果真有 1 个超大模型需要预训练，3D 并行那是必不可少的**。毕竟显卡进化的比较慢，最大显存的也就是 A100 80g。

单卡 80g，可以完整加载小于 40B 的模型，但是训练时+梯度+优化器状态，5B 模型就是上限了，更别说 activation 的参数也要占显存，batch size 还得大。而现在 100 亿以下（10B 以下）的 LLM 只能叫 small LLM。

1.6 Colossal-AI 有 1D/2D/2.5D/3D，是什么情况？

Colossal-AI 的 nD 是针对张量并行，指的是 TP 的切分，对于矩阵各种切，和 3D 并行不是一回事。

1.7 除了 3D 并行有没有其他方式大规模训练？

可以使用更优化的数据并行算法 FSDP（类似 ZeRO3）或者直接使用 DeepSpeed ZeRO 。

1.8 有了 ZeRO 系列，为什么还需要 3D 并行？

根据 ZeRO 论文，尽管张量并行的显存更省一点，张量并行的通信量实在太高，只能限于节点内（有 NVLINK）。如果节点间张量并行，显卡的利用率会低到 5%

但是，根据 Megatron-LM2 的论文，当显卡数量增加到千量级，ZeRO3 是明显不如 3D 并行的。

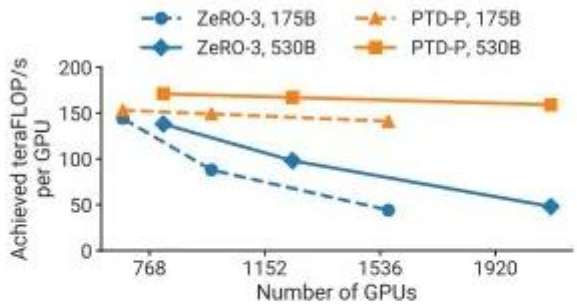


Figure 10: Throughput per GPU of PTD-P and ZeRO-3 for two different GPT models (the 175B GPT-3 model is shown with dotted lines, and the 530B model is shown with solid lines). Global batch sizes are fixed and ZeRO-3 is used without any model parallelism.

1.9 平民适不适合玩 3D 并行？

不适合。

3D 并行的基础是，节点内显卡间 NVLINK 超高速连接才能上 TP。有没有 NVLINK 都是个问题。

而且，节点间特殊的网络通常有 400Gb/s？远超普通 IDC 内的万兆网络 10Gb/s。

1.10 平民适不适合直接上多机多卡的 ZeRO3（万兆网）？

不适合。

想象一下，当 65B 模型用 Zero3，每一个 step 的每一张卡上需要的通信量是 195GB（3 倍参数量），也就是 1560Gb。万兆网下每步也要 156s 的通信时间，这画面太美。

2. 实践篇

2.1 假如有超多的 8 卡 A100 节点（DGX A100），如何应用 3D 并行策略？

- 首先，**张量并行**。3 种并行方式里，张量并行（TP）对于 GPU 之间的通信要求最高，而节点内有 NVLINK 通信速度可以达到 600GB/s。
- 其次，**流水线并行**，每个节点负责一部分层，每 35 个节点组成一路完整的流水线，也就是一个完整的模型副本，这里一个模型副本需 280 卡。
- 最后，**数据并行**，官方也做了 8 路，10 路，12 路的并行实验，分别使用 280 个节点，350 个节点和 420 个节点。

参考 Megatron-Turing NLG 530B

集群规模越大，单个 GPU 利用率越低。

2.2 如果想构这样一个大规模并行训练系统，训练框架如何选？

可以参考 Megatron-Turing NLG 530B，NVIDIA Megatron-LM + Microsoft DeepSpeed

BLOOM **则是 PP+DP 用 DeepSpeed，TP 用 Megatron-LM**

当然还有一些其他的训练框架，在超大规模下或许也能 work。

2.3 训练框架如何选？

下面这个图是 bloom 的一个实验，DP/TP/PP 都能降显存，核心是要降到单卡峰值 80g 以下。

真大模型就是要 TP=8，充分利用 NVLINK，然后优先 PP，最后 DP。

GPUs	Size	DP	TP	PP	MBS	Mem	TFLOPs	Notes
8	20B	1	8	1	1	68GB	107.48	02-17
80	200B	1	8	10	1	75GB	97.82	02-17
160	200B	2	8	10	1	53GB	96.19	02-17

然而假大模型（7B）比如 LLaMA-7B，可以不用 3D 并行，直接用 DeepSpeed ZeRO 更方便，参考 open-llama 项目。

3. 并行化策略选择篇

3.1 单 GPU

- 显存够用： 直接用
- 显存不够：上 offload，用 cpu

3.2 单节点多卡

- 显存够用（模型能装进单卡）：DDP 或 ZeRO
- 显存不够：TP 或者 ZeRO 或者 PP

重点：没有 NVLINK 或者 NVSwitch，也就是穷人模式，要用 PP

3.3 多节点多卡

如果节点间通信速度快（穷人的万兆网肯定不算）

ZeRO 或者 3D 并行，其中 3D 并行通信量少但是对模型改动大。

如果节点间通信慢，但显存又少。

DP+PP+TP+ZeRO-1

4. 问题篇

4.1 推理速度验证

ChatGML 在 V100 单卡的推理耗时大约高出 A800 单卡推理的 40%。

ChatGML 推理耗时和问题输出答案的字数关系比较大，答案字数 500 字以内，A800 上大概是每 100 字，耗时 1 秒，V100 上大概是每 100 字，耗时 1.4 秒。

- ChatGML 在 A800 单卡推理耗时统计

问题	运行次数	平均答案长度	平均耗时
给我介绍一下苹果公司,50个字	5	146.6	1.9458990097045898
给我介绍一下微软公司,50个字	5	104.6	1.2978283882141113
给我介绍一下苹果公司,100个字	5	165.4	2.0990972995758055
给我介绍一下微软公司,100个字	5	154.4	1.9092102527618409
给我介绍一下苹果公司,200个字	5	168.4	2.1302066326141356
给我介绍一下微软公司,200个字	5	208.8	2.6089860916137697
给我介绍一下苹果公司,300个字	5	443.4	5.4034130573272705
给我介绍一下微软公司,300个字	5	484.2	5.980589342
给我介绍一下苹果公司,500个字	5	525.4	6.246328496932984
给我介绍一下微软公司,500个字	5	591.6	6.961390399932862

- ChatGML 在 V100 单卡推理耗时统计

问题	运行次数	平均答案长度	平均耗时
给我介绍一下苹果公司,50个字	5	146.6	1.9458990097045898
给我介绍一下微软公司,50个字	5	104.6	1.2978283882141113
给我介绍一下苹果公司,100个字	5	165.4	2.0990972995758055
给我介绍一下微软公司,100个字	5	154.4	1.9092102527618409
给我介绍一下苹果公司,200个字	5	168.4	2.1302066326141356
给我介绍一下微软公司,200个字	5	208.8	2.6089860916137697
给我介绍一下苹果公司,300个字	5	443.4	5.4034130573272705
给我介绍一下微软公司,300个字	5	484.2	5.980589342
给我介绍一下苹果公司,500个字	5	525.4	6.246328496932984
给我介绍一下微软公司,500个字	5	591.6	6.961390399932862

- 结论：
- 训练效率方面: 多机多卡训练，增加训练机器可以线性缩短训练时间。
- 推理性能方面:
- ChatGML 在 V100 单卡的推理耗时大约高出 A800 单卡推理的 40%。
- ChatGML 推理耗时和问题输出答案的字数关系比较大，答案字数 500 字以内，A800 上大概是每 100 字，耗时 1 秒，V100 上大概是每 100 字，耗时 1.4 秒。

4.2 并行化训练加速

可采用 deepspeed 进行训练加速，目前行业开源的大模型很多都是采用的基于 deepspeed 框架加速来进行模型训练的。如何进行 deepspeed 训练，可以参考基于 deepspeed 构建大模型分布式训练平台。

deepspeed 在深度学习模型软件体系架构中所处的位置：

DL model—>train opitimization(deepspeed)—>train framework —> train instruction (cloud)—>GPU device

当然需要对比验证 deepspeed 的不同参数，选择合适的参数。分别对比 stage 2,3 进行验证，在 GPU 显存够的情况下，最终使用 stage 2。

4.3 deepspeed 训练过程，报找不主机

解决方法：deepspeed 的关联的多机的配置文件，Hostfile 配置中使用 ip，不使用 hostname

4.4 为什么 多机训练效率不如单机？

多机训练可以跑起来，但是在多机上模型训练的速度比单机上还慢。

通过查看服务器相关监控，发现是网络带宽打满，上不去了，其他系统监控基本正常。原理初始的多机之间的网络带宽是 64Gps，后面把多机之间的网络带宽调整为 800Gps，问题解决。

实验验证，多机训练的效率，和使用的机器数成线性关系，每台机器的配置一样，如一台 GPU 机器跑一个 epoch 需要 2 小时，4 台 GPU 机器跑一个 epoch 需要半小时。除了训练速度符合需求，多机训练模型的 loss 下降趋势和单机模型训练的趋势基本一致，也符合预期。

4.5 多机训练不通，DeepSPeed 配置问题

多机间 NCCL 不能打通

- 解决方法：

新建 .deepspeed_env 文件，写入如下内容

```
NCCL_IB_DISABLE=1
NCCL_DEBUG=INFO
NCCL_SOCKET_IFNAME=eth0
NCCL_P2P_DISABLE=1
```

强化学习面

1. 奖励模型需要和基础模型一致吗？

不同实现方式似乎限制不同。（待实践确认）colossal-ai 的 coati 中需要模型有相同的 tokenizer，所以选模型只能从同系列中找。在 ppo 算法实现方式上据说 trlx 是最符合论文的

显存问题面

1. 大模型大概有多大，模型文件有多大？

一般放出来的模型文件都是 fp16 的，假设是一个 n B 的模型，那么模型文件占 2n G，fp16 加载到显存里做推理也是占 2n G，对外的 pr 都是 10n 亿参数的模型。

2. 能否用 4 * v100 32G 训练 vicuna 65b？

不能。

- 首先，llama 65b 的权重需要 5* v100 32G 才能完整加载到 GPU。
- 其次，vicuna 使用 flash-attention 加速训练，暂不支持 v100，需要 turing 架构之后的显卡。

（刚发现 fastchat 上可以通过调用 train 脚本训练 vicuna 而非 train_mem，其实也是可以训练的）

3. 如果就是想要试试 65b 模型，但是显存不多怎么办？

最少大概 50g 显存，可以在 llama-65b-int4（gptq）模型基础上 LoRA[6]，当然各种库要安装定制版本的。

4. nB 模型推理需要多少显存？

考虑模型参数都是 fp16，2nG 的显存能把模型加载。

5. nB 模型训练需要多少显存？

基础显存：模型参数+梯度+优化器，总共 16nG。

activation 占用显存，和 max len、batch size 有关

解释：优化器部分必须用 fp32（似乎 fp16 会导致训练不稳定），所以应该是 2+2+12=16，参考 ZeRO 论文。

注以上算数不够直观，举个例子？

7B 的 vicuna 在 fsdp 下总共 160G 显存勉强可以训练。（按照上面计算 7*16=112G 是基础显存）

所以全量训练准备显存 20nG 大概是最低要求，除非内存充足，显存不够 offload 内存补。

6. 如何 估算模型所需的 RAM？

首先，我们需要了解如何根据参数量估计模型大致所需的 RAM，这在实践中有很重要的参考意义。我们需要通过估算设置 batch_size，设置模型精度，选择微调方法和参数分布方法等。

接下来，我们用 LLaMA-6B 模型为例估算其大致需要的内存。

首先考虑精度对所需内存的影响：

- fp32 精度，一个参数需要 32 bits, 4 bytes.
- fp16 精度，一个参数需要 16 bits, 2 bytes.
- int8 精度，一个参数需要 8 bits, 1 byte.

其次，考虑模型需要的 RAM 大致分三个部分：

- 模型参数
- 梯度
- 优化器参数
- 模型参数：等于参数量*每个参数所需内存。
- 对于 fp32，LLaMA-6B 需要 6B*4 bytes = 24GB 内存
- 对于 int8，LLaMA-6B 需要 6B*1 byte = 6GB
- 梯度：同上，等于参数量*每个梯度参数所需内存。
- 优化器参数：不同的优化器所储存的参数量不同。

对于常用的 AdamW 来说，需要储存两倍的模型参数（用来储存一阶和二阶 momentum）。

- fp32 的 LLaMA-6B，AdamW 需要 6B*8 bytes = 48 GB
- int8 的 LLaMA-6B，AdamW 需要 6B*2 bytes = 12 GB

除此之外，CUDA kernel 也会占据一些 RAM，大概 1.3GB 左右，查看方式如下。

```
> torch.ones((1, 1)).to("cuda")
> print_gpu_utilization()
>>>
GPU memory occupied: 1343 MB
```

综上，int8 精度的 LLaMA-6B 模型部分大致需要 6GB+6GB+12GB+1.3GB = 25.3GB 左右。

再根据 LLaMA 的架构（hidden_size = 4096, intermediate_size =11008, num_hidden_layers = 32, context_length = 2048）计算中间变量内存。

每个 instance 需要：

$$(4096 + 11008) * 2048 * 32 * 1byte = 990MB$$

所以一张 A100（80GB RAM）大概可以在 int8 精度，batch_size = 50 的设定下进行全参数训练。

查看消费级显卡的内存和算力：

2023 GPU Benchmark and Graphics Card Comparison Chart

https://www.gpucheck.com/gpu-benchmark-graphics-card-comparison-chart

7. 如何评估你的显卡利用率

zero3 如果没有 nvlink，多卡训练下会变慢。但是一直不知道究竟会变得多慢，下面给出几种方法来评估自己在训练时发挥了多少 gpu 性能，以及具体测试方法。

7.1 flops 比值法

- 测试工具：deepspeed
- 参考数据：nvidia 公布的显卡 fp16 峰值计算速度（tensor core）

$$gpu\ 利用率 = 实测的\ flops / 显卡理论上的峰值\ flops$$

举例：deepspeed 实测 flops 100tflops，而用的是 A100 卡理论峰值 312tflops，可以得到 GPU 利用率只有 32.05%

7.2 throughout 估计法

- 测试工具：手动估算 或者 deepspeed
- 参考数据：论文中的训练速度或者吞吐量

$$吞吐量 = example\ 数量/秒/GPU * max\_length$$

$$gpu\ 利用率 = 实际吞吐量 / 论文中的吞吐量（假设利用率\ 100\%）$$

举例：

实测训练时处理样本速度为 3 example/s，一共有 4 卡，max length 2048，则吞吐量为 1536 token/s/gpu

根据 llama 论文知道，他们训练 7B 模型的吞吐量约为 3300 token/s/gpu，那么 GPU 利用率只有 46.54%

7.3 torch profiler 分析法

- 测试工具：torch profiler 及 tensorboard
- 参考数据：无

利用 torch profiler 记录各个函数的时间，将结果在 tensorboard 上展示，在 gpu kenel 视图下，可以看到 tensor core 的利用率，比如 30%

总结

以上三种方法，在笔者的实验中能得到差不多的利用率指标。

从准确性上看，方案三 > 方案一 > 方案二

从易用性上看，方案二 > 方案一 > 方案三

如果不想改代码就用方案二估算自己的训练速度是不是合理的，如果想精确分析训练速度的瓶颈还是建议使用方案三。

8. 测试你的显卡利用率 实现细节篇

8.1 如何查看多机训练时的网速？

iftop 命令，看网速很方便。

8.2 如何查看服务器上的多卡之间的 NVLINK topo？

```
$ nvidia-smi topo -m
```

8.3 如何查看服务器上显卡的具体型号？

```
cd /usr/local/cuda/samples/1_Utilities/deviceQuery
make
./deviceQuery
```

8.4 如何查看训练时的 flops？（也就是每秒的计算量）

理论上，如果 flops 比较低，说明没有发挥出显卡的性能。

如果基于 deepspeed 训练，可以通过配置文件很方便的测试

```
{
"flops_profiler": {
"enabled": true,
"profile_step": 1,
"module_depth": -1,
"top_modules": 1,
"detailed": true,
"output_file": null
}
}
```

参考：https://www.deepspeed.ai/tutorials/flops-profiler/

8.5 如何查看对 deepspeed 的环境配置是否正确？

```
$ ds_report
```

8.6 tf32 格式有多长？

19 位

1. 大模型大概有多大，模型文件有多大?

一般放出来的模型文件都是 fp16 的，假设是一个 n B 的模型，那么模型文件占 2n G，fp16 加载到显存里做推理也是占 2n G，对外的 pr 都是 10n 亿参数的模型。

2. 能否用 4 * v100 32G 训练 vicuna 65b？

不能。

- 首先，llama 65b 的权重需要 5* v100 32G 才能完整加载到 GPU。
- 其次，vicuna 使用 flash-attention 加速训练，暂不支持 v100，需要 turing 架构之后的显卡。

（刚发现 fastchat 上可以通过调用 train 脚本训练 vicuna 而非 train_mem，其实也是可以训练的）

3. 如果就是想要试试 65b 模型，但是显存不多怎么办？

最少大概 50g 显存，可以在 llama-65b-int4（gptq）模型基础上 LoRA[6]，当然各种库要安装定制版本的。

4. nB 模型推理需要多少显存？

考虑模型参数都是 fp16，2nG 的显存能把模型加载。

5. nB 模型训练需要多少显存？

基础显存：模型参数+梯度+优化器，总共 16nG。
activation 占用显存，和 max len、batch size 有关
解释：优化器部分必须用 fp32（似乎 fp16 会导致训练不稳定），所以应该是 2+2+12=16，参考 ZeRO 论文。
注以上算数不够直观，举个例子？
7B 的 vicuna 在 fsdp 下总共 160G 显存勉强可以训练。（按照上面计算 7*16=112G 是基础显存）
所以全量训练准备显存 20nG 大概是最低要求，除非内存充足，显存不够 offload 内存补。

6. 如何 估算模型所需的 RAM？

首先，我们需要了解如何根据参数量估计模型大致所需的 RAM，这在实践中有很重要的参考意义。我们需要通过估算设置 batch_size，设置模型精度，选择微调方法和参数分布方法等。
接下来，我们用 LLaMA-6B 模型为例估算其大致需要的内存。
首先考虑精度对所需内存的影响：

- fp32 精度，一个参数需要 32 bits, 4 bytes.
- fp16 精度，一个参数需要 16 bits, 2 bytes.
- int8 精度，一个参数需要 8 bits, 1 byte.

其次，考虑模型需要的 RAM 大致分三个部分：

- 模型参数
- 梯度
- 优化器参数
- 模型参数：等于参数量*每个参数所需内存。
- 对于 fp32，LLaMA-6B 需要 6B*4 bytes = 24GB 内存
- 对于 int8，LLaMA-6B 需要 6B*1 byte = 6GB
- 梯度：同上，等于参数量*每个梯度参数所需内存。
- 优化器参数：不同的优化器所储存的参数量不同。

对于常用的 AdamW 来说，需要储存两倍的模型参数（用来储存一阶和二阶 momentum）。

- fp32 的 LLaMA-6B，AdamW 需要 6B*8 bytes = 48 GB
- int8 的 LLaMA-6B，AdamW 需要 6B*2 bytes = 12 GB

除此之外，CUDA kernel 也会占据一些 RAM，大概 1.3GB 左右，查看方式如下。

```
> torch.ones((1, 1)).to("cuda")
> print_gpu_utilization()
>>>
GPU memory occupied: 1343 MB
```

综上，int8 精度的 LLaMA-6B 模型部分大致需要 6GB+6GB+12GB+1.3GB = 25.3GB 左右。
再根据 LLaMA 的架构（hidden_size = 4096, intermediate_size =11008, num_hidden_layers = 32, context_length = 2048）计算中间变量内存。
每个 instance 需要：

```
(4096 +11008)* 2048 *32 * 1byte = 990MB
```

所以一张 A100（80GB RAM）大概可以在 int8 精度；batch_size = 50 的设定下进行全参数训练。
查看消费级显卡的内存和算力：
2023 GPU Benchmark and Graphics Card Comparison Chart
https://www.gpucheck.com/gpu-benchmark-graphics-card-comparison-chart

7. 如何评估你的显卡利用率

zero3 如果没有 nvlink，多卡训练下会变慢。但是一直不知道究竟会变得多慢，下面给出几种方法来评估自己在训练时发挥了多少 gpu 性能，以及具体测试方法。

7.1 flops 比值法

- 测试工具：deepspeed

- 参考数据：nvidia 公布的显卡 fp16 峰值计算速度（tensor core）

```
gpu 利用率 = 实测的 flops/显卡理论上的峰值 flops
```

举例：deepspeed 实测 flops 100tflops，而用的是 A100 卡理论峰值 312tflops，可以得到 GPU 利用率只有 32.05%

7.2 throughout 估计法

- 测试工具：手动估算 或者 deepspeed
- 参考数据：论文中的训练速度或者吞吐量

```
吞吐量 = example 数量/秒/GPU * max_length
```

```
gpu 利用率 = 实际吞吐量 / 论文中的吞吐量（假设利用率 100%）
```

举例：
实测训练时处理样本速度为 3 example/s，一共有 4 卡，max length 2048，则吞吐量为 1536 token/s/gpu
根据 llama 论文知道，他们训练 7B 模型的吞吐量约为 3300 token/s/gpu，那么 GPU 利用率只有 46.54%

7.3 torch profiler 分析法

- 测试工具：torch profiler 及 tensorboard
- 参考数据：无

利用 torch profiler 记录各个函数的时间，将结果在 tensorboard 上展示，在 gpu kenel 视图下，可以看到 tensor core 的利用率，比如 30%

总结

以上三种方法，在笔者的实验中能得到差不多的利用率指标。
从准确性上看，方案三 > 方案一 > 方案二
从易用性上看，方案二 > 方案一 > 方案三
如果不想改代码就用方案二估算自己的训练速度是不是合理的，如果想精确分析训练速度的瓶颈还是建议使用方案三。

8. 测试你的显卡利用率 实现细节篇

8.1 如何查看多机训练时的网速？

iftop 命令，看网速很方便。

8.2 如何查看服务器上的多卡之间的 NVLINK topo？

```
$ nvidia-smi topo -m
```

8.3 如何查看服务器上显卡的具体型号?

```
cd /usr/local/cuda/samples/1_Utilities/deviceQuery
make
./deviceQuery
```

8.4 如何查看训练时的 flops？（也就是每秒的计算量）

理论上，如果 flops 比较低，说明没有发挥出显卡的性能。
如果基于 deepspeed 训练，可以通过配置文件很方便的测试

```
{
"flops_profiler": {
"enabled": true,
"profile_step": 1,
"module_depth": -1,
"top_modules": 1,
"detailed": true,
"output_file": null
}
}
```

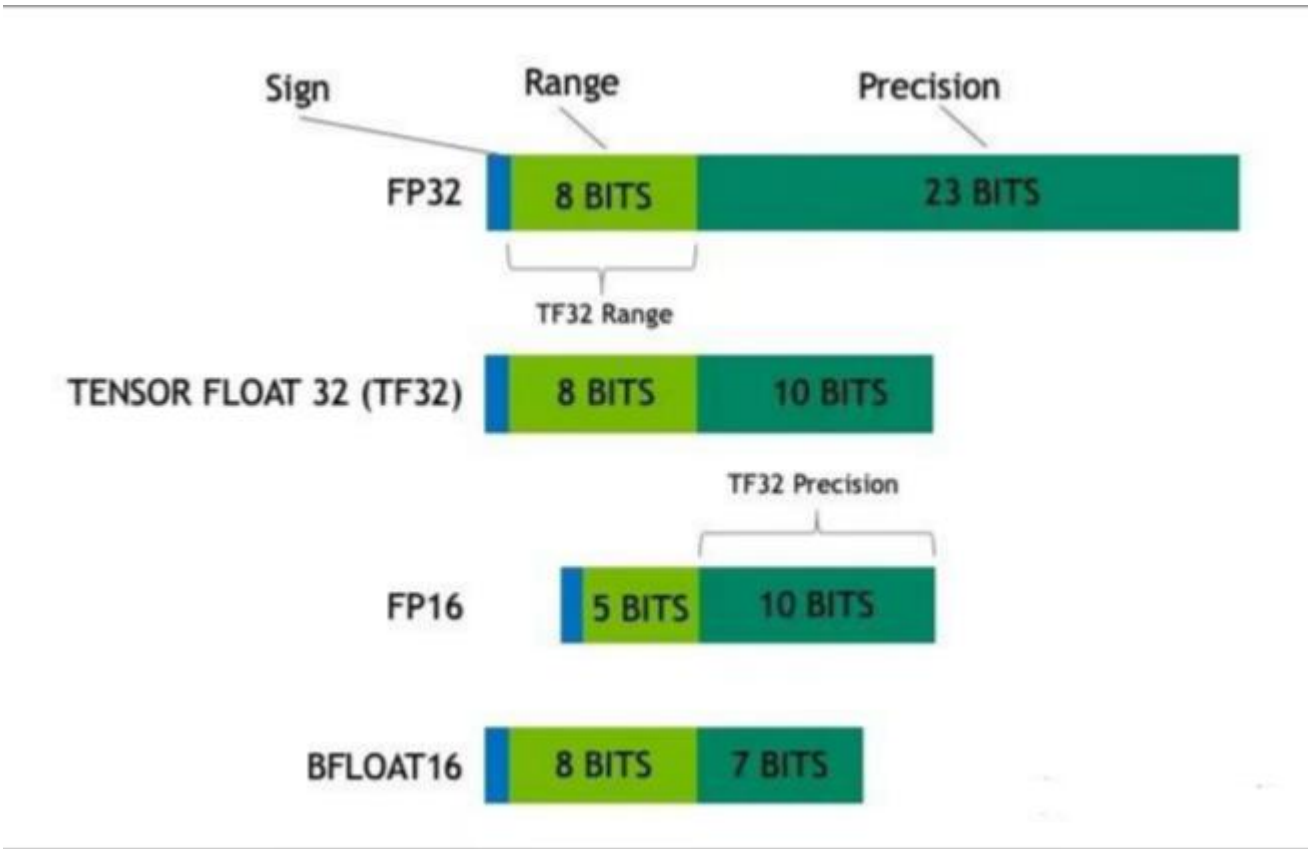
参考：https://www.deepspeed.ai/tutorials/flops-profiler/

8.5 如何查看对 deepspeed 的环境配置是否正确？

```
$ ds_report
```

8.6 tf32 格式有多长？

19 位



8.7 哪里看各类显卡算力比较？

https://lambdalabs.com/gpu-benchmarks

8.8 （torch profiler）如何查看自己的训练中通信开销？

用 pytorch profiler 查看，下面给出基于 transformers 的一种快捷的修改方式。

https://github.com/yqhu/profiler-workshop/blob/c8d4a7c30a61cc7b909d89f88f5fd36b70c55769/hf_training_trainer_prof.py

用记录的 pt.trace.json 文件放到 tensorboard 上，可以看出 tensor core 的利用率。

根据实践经验，使用 deepspeed zero3 时，pcie 版本的卡很大部分时间都在通信上，AllGather 和 ReduceScatter 的时间超过 tensor core 计算的时间，所以 flops 上不去。

8.7 哪里看各类显卡算力比较？

https://lambdalabs.com/gpu-benchmarks

8.8 （torch profiler）如何查看自己的训练中通信开销？

用 pytorch profiler 查看，下面给出基于 transformers 的一种快捷的修改方式。

https://github.com/yqhu/profiler-workshop/blob/c8d4a7c30a61cc7b909d89f88f5fd36b70c55769/hf_training_trainer_prof.py

用记录的 pt.trace.json 文件放到 tensorboard 上，可以看出 tensor core 的利用率。

根据实践经验，使用 deepspeed zero3 时，pcie 版本的卡很大部分时间都在通信上，AllGather 和 ReduceScatter 的时间超过 tensor core 计算的时间，所以 flops 上不去。

进阶面

1 LLMs 复读机问题

1.1 什么是 LLMs 复读机问题？

LLMs 复读机问题 就是 LLMs 出现 重复输出：

> *eg：ABCABCABC 不断循环输出到 max length*

1.2 为什么会出现 LLMs 复读机问题？

出现 LLMs 复读机问题 的 原因：

prompt 部分通常很长，在生成文本时可以近似看作不变，那么条件概率 P(B|A)也不变，一直是最大的。

生成重复内容，是语言模型本身的一个弱点，无论是否微调，都有可能出现。并且，理论上良好的指令微调能够缓解大语言模型生成重复内容的问题。但是因为指令微调策略的问题，在实践中经常出现指令微调后复读机问题加重的情况。

1.3 如何缓解 LLMs 复读机问题？

- 方法一：解码方式里增加不确定性。既然容易复读那我们就增加随机性，开启 do_sample 选项，调高 temperature；
- 方法二：加重复惩罚。如果学的太烂，do_sample 也不顶用呢？加重复惩罚，设置 repetition_penalty，注意别设置太大了。不然你会发现连标点符号都不会输出了。

2 llama 系列问题

2.1 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，但问题是太长了效果不好啊，没训练过的长度效果通常不好。而想办法让没训练过的长度效果好，这个问题就叫做“长度外推性”问题。

所以接受 2k 的长度限制吧。

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

Bert 的模型由多层双向的 Transformer 编码器组成，由 12 层组成，768 隐藏单元，12 个 head，总参数量 110M，约 1.15 亿参数量。

NLU（自然语言理解）任务效果很好，单卡 GPU 可以部署，速度快，V100GPU 下 1 秒能处理 2 千条以上。

ChatGLM-6B, LLaMA-7B 模型分别是 60 亿参数量和 70 亿参数量的大模型，基本可以处理所有 NLP 任务，效果好，但大模型部署成本高，需要大显存的 GPU，并且预测速度慢，V100 都需要 1 秒一条。

所以建议：

1）NLU 相关的任务，用 BERT 模型能处理的很好，如实体识别、信息抽取、文本分类，没必要上大模型；

2）NLG 任务，纯中文任务，用 ChatGLM-6B，需要处理中英文任务，用 chinese-alpaca-plus-7b-hf

4 各个专业领域是否需要各自的大模型来服务？

是，各行各业的大模型是趋势。

5 如何让大模型处理更长的文本？

- 动机：目前绝大多数大模型支持的 token 最大长度为 2048，因为序列长度直接影响 Attention 的计算复杂度，太长了会影响训练速度。

- 让大模型处理更长的文本 方法
- LongChat

就两步：

step1：将新的长度压缩到原来 2048 长度上，这样的好处是能复用原来的位置信息，增加长度并没有破坏 position 的权重。

比如从 2048 扩展到 16384，长度变为原来的 8 倍，那么值为 10000 的 position_id，被压缩成 10000/8=1250

代码只需要改一行：

```
# 将 position_ids 按比例缩一下。
query_states, key_states = apply_rotary_pos_emb(query_states, key_states, cos, sin, position_ids / ratio).
```

详细参考：https://kaiokendev.github.io/context

step2：用训练 Vicuna 的对话语料做微调，超过 16k 的文本被截断。

- position 等比例缩放既然有用，那后续会不会有一种新的 position 构造的方式，无论多长都可以归一到同样的尺度下，只要 position 的相对位置保持不变就可以？其实 ALiBi 的方法就是一个比较简单优雅的方式，可以部分解决扩展长度的问题。
- 商业模型比如 ChatGPT 和 Claude 到底是怎么做的？这个目前都没有公开。首先语料是不缺的，所以只能是结构上的变化。但是这两个商业模型规模都是 100B 这个量级的，这么大的参数，如果只增加序列长度而不做其他优化的话，很难训练起来。目前有证据的方法如下：
- 稀疏化，GPT3 的论文中曾提到有这方面的尝试。
- Google 的周彦祺在一次分享中透露 GPT-4 用了 MoE 的技术(猜测是 100B16E)，所以应该有类似的方法来保证在序列变长的情况下，仍然能高效的训练模型。
- Multi-Query Attention。Google 的 PaLM，Falcon 等模型都用到过，通过权重共享来提升性能。
- 真正的出路可能还是 Linear Attention，将 Attention 的复杂度从 O(N2) 降低为 O(N). 比如 Linear Transformer 和 RWKV。其实关于变长序列的问题，历史上现成的解决方案就是 RNN，通过信息传递来解决。Transformer 的卖点就是 Attention is All your Need，丢弃了 RNN，RWKV 敢于把 RNN 拿回来，还是很有勇气，非常好的一个工作。现在的 Attention 就有点像历史上的 MLP，每个节点之间都要建立关联，而 MLP 之后涌现了大量新的结构，所以 Transformer 是起点，后续肯定会有更合理的结构来取代它。

评测面

1 大模型怎么评测？

当前 superGLUE, GLUE, 包括中文的 CLUE 的 benchmark 都在不太合适评估大模型。可能评估推理能力、多轮对话能力是核心。

2 大模型的 honest 原则是如何实现的？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？

大模型需要遵循的 helpful，honest， harmless 的原则。

可以有意构造如下的训练样本，以提升模型准守 honest 原则，可以算 trick 了：

微调时构造知识问答类训练集，给出不知道的不回答，加强 honest 原则；

阅读理解题，读过的要回答，没读过的不回答，不要胡说八道。

大模型（LLMs）训练集面

1. SFT（有监督微调）的数据集格式？

一问一答

2. RM（奖励模型）的数据格式？

一个问题 + 一条好回答样例 + 一条差回答样例

3. PPO（强化学习）的数据格式？

理论上来说，不需要新增数据。需要提供一些 prompt，可以直接用 sft 阶段的问。另外，需要限制模型不要偏离原模型太远（ptx loss），也可以直接用 sft 的数据。

4. 找数据集哪里找？

推荐 Alpaca-COT，数据集整理的非常全，眼花缭乱。

5. 微调需要多少条数据？

取决于预训练数据和微调任务的数据分布是否一致，分布一致，100 条就够，分布差异大就需要多些数据，千条或者万条以上为佳。

自己的任务复杂或者下游任务行业比较冷门，如药品名称识别任务，则需要较多监督数据。还有微调大模型时，一遍是记不住的。100 条的微调数据，epochs=20 才能稳定拟合任务要求。

6. 有哪些大模型的训练集？

预训练数据集 togethercomputer/RedPajama-Data-1T「红睡衣」开源计划总共包括三部分：

- 高质量、大规模、高覆盖度的预训练数据集；
- 在预训练数据集上训练出的基础模型；
- 指令调优数据集和模型，比基本模型更安全、可靠。

预训练数据集 RedPajama-Data-1T 已开源，包括七个子集，经过预处理后得到的 token 数量大致可以匹配 Meta 在原始 LLaMA 论文中报告的数量，并且数据预处理相关脚本也已开源。

完整的 RedPajama-Data-1T 数据集需要的存储容量为压缩后 3TB，解压后 5TB。

CoT 微调数据集：Alpaca-CoT 里面包括常用的 alpaca，CoT 等数据集，有中文的。

7. 进行领域大模型预训练应用哪些数据集比较好？

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数据、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。给我们自己进行大模型预训练的时候提供了一个参考。同时领域相关的网站内容、新闻内容也是比较重要的数据。

agent 面

1. 如何给 LLM 注入领域知识？

- 方法一：检索+LLM。先用问题在领域数据库里检索到候选答案，再用 LLM 对答案进行加工。
- 方法二：领域知识微调 LLM。把领域知识构建成问答数据集，用 SFT 让 LLM 学习这部分知识。

2. 如果想要快速体验各种模型，该怎么办？

推荐 fastchat，集成了各路开源模型，从自己的 vicuna 到 stable AI 的 stableLM。

软硬件配置面

1 建议的软件环境是什么？

python 环境建议 3.9，因为 fastchat[4]等比较好的开源项目需要 3.9 及以上。

cuda 环境越高越好，c++版本建议直接装 9.1.0 以上。

基础面

1 目前 主流的开源模型体系 有哪些？

目前 主流的开源模型体系 分两种：

- 第一种：prefix LM 系
- 代表模型：T5、ChatGLM、ChatGLM2
- 第二种：causal LM 系
- 代表模型：LLaMA-7B、LLaMa 衍生物

2 prefix LM 和 causal LM 区别是什么？

prefix LM 和 causal LM 区别 在于 attention mask 不同：

- prefix LM：prefix 部分的 token 互相能看到；
- causal LM：严格遵守只有后面的 token 才能看到前面的 token 的规则；

3 涌现能力是啥原因？

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，这个其实好理解，比如我们假设某个任务 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。

4. 大模型 LLM 的架构介绍？

在自然语言里面有两种模型

- （1）causal language modeling:它的输出是依赖过去和现在的输入。
- （2）masked language modeling:它是把句子中的一个词盖住，然后通过这个词的上下文去预测这个词。

大模型的底座模型就是多层的 transformer，由于是因果语言模型，它只用了 transformer 的 decoder 模块。

5. 为何现在的大模型大部分是 Decoder only 结构？

因为 **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 shang 做自监督学习，很显然 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。而 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。所以在同等参数量、同等推理成本下，Decoder-only 架构就是最优的选择了。