Happy-LLM

中文 | English

📚 从零开始的大语言模型原理与实践教程

深入理解 LLM 核心原理，动手实现你的第一个大模型

🎯 项目介绍

> *很多小伙伴在看完 Datawhale开源项目： self-llm 开源大模型食用指南 后，感觉意犹未尽，想要深入了解大语言模型的原理和训练过程。于是我们（Datawhale）决定推出《Happy-LLM》项目，旨在帮助大家深入理解大语言模型的原理和训练过程。*

本项目是一个**系统性的 LLM 学习教程**，将从 NLP 的基本研究方法出发，根据 LLM 的思路及原理逐层深入，依次为读者剖析 LLM 的架构基础和训练过程。同时，我们会结合目前 LLM 领域最主流的代码框架，演练如何亲手搭建、训练一个 LLM，期以实现授之以鱼，更授之以渔。希望大家能从这本书开始走入 LLM 的浩瀚世界，探索 LLM 的无尽可能。

✨ 你将收获什么？

- 📚 **Datawhale 开源免费** 完全免费的学习本项目所有内容
- 🔍 **深入理解** Transformer 架构和注意力机制
- 📚 **掌握** 预训练语言模型的基本原理

- 🧠 **了解** 现有大模型的基本结构
- 🏗️ **动手实现** 一个完整的 LLaMA2 模型
- ⚙️ **掌握训练** 从预训练到微调的全流程
- 🚀 **实战应用** RAG、Agent 等前沿技术

📖 内容导航

章节	关键内容	状态
前言	本项目的缘起、背景及读者建议	✅
第一章 NLP 基础概念	什么是 NLP、发展历程、任务分类、文本表示演进	✅
第二章 Transformer 架构	注意力机制、Encoder-Decoder、手把手搭建 Transformer	✅
第三章 预训练语言模型	Encoder-only、Encoder-Decoder、Decoder-Only 模型对比	✅
第四章 大语言模型	LLM 定义、训练策略、涌现能力分析	✅
第五章 动手搭建大模型	实现 LLaMA2、训练 Tokenizer、预训练小型 LLM	✅
第六章 大模型训练实践	预训练、有监督微调、LoRA/QLoRA 高效微调	✅
第七章 大模型应用	模型评测、RAG 检索增强、Agent 智能体	✅

💡 如何学习

本项目适合大学生、研究人员、LLM 爱好者。在学习本项目之前，建议具备一定的编程经验，尤其是要对 Python 编程语言有一定的了解。最好具备深度学习的相关知识，并了解 NLP 领域的相关概念和术语，以便更轻松地学习本项目。

本项目分为两部分——基础知识与实战应用。第1章～第4章是基础知识部分，从浅入深介绍 LLM 的基本原理。其中，第1章简单介绍 NLP 的基本任务和发展，为非 NLP 领域研究者提供参考；第2章介绍 LLM 的基本架构——Transformer，包括原理介绍及代码实现，作为 LLM 最重要的理论基础；第3章整体介绍经典的 PLM，包括 Encoder-Only、Encoder-Decoder 和 Decoder-Only 三种架构，也同时介绍了当前一些主流 LLM 的架构和思想；第4章则正式进入 LLM 部分，详细介绍 LLM 的特点、能力和整体训练过程。第5章～第7章是实战应用部分，将逐步带领大家深入 LLM 的底层细节。其中，第5章将带领大家者基于 PyTorch 层亲手搭建一个 LLM，并实现预训练、有监督微调的全流程；第6章将引入目前业界主流的 LLM 训练框架 Transformers，带领学习者基于该框架快速、高效地实现 LLM 训练过程；第7章则将介绍 基于 LLM 的各种应用，补全学习者对 LLM 体系的认知，包括 LLM 的评测、检索增强生产（Retrieval-Augmented Generation，RAG）、智能体（Agent）的思想和简单实现。你可以根据个人兴趣和需求，选择性地阅读相关章节。

在阅读本书的过程中，建议你将理论和实际相结合。LLM 是一个快速发展、注重实践的领域，我们建议你多投入实战，复现本书提供的各种代码，同时积极参加 LLM 相关的项目与比赛，真正投入到 LLM 开发的浪潮中。我们鼓励你关注 Datawhale 及其他 LLM 相关开源社区，当遇到问题时，你可以随时在本项目的 issue 区提问。

最后，欢迎每一位读者在学习完本项目后加入到 LLM 开发者的行列。作为国内 AI 开源社区，我们希望充分聚集共创者，一起丰富这个开源 LLM 的世界，打造更多、更全面特色 LLM 的教程。星火点点，汇聚成海。我们希望成为 LLM 与普罗大众的阶梯，以自由、平等的开源精神，拥抱更恢弘而辽阔的 LLM 世界。

🤝 如何贡献

我们欢迎任何形式的贡献！

- 🐛 **报告 Bug** - 发现问题请提交 Issue
- 💡 **功能建议** - 有好想法就告诉我们
- 📝 **内容完善** - 帮助改进教程内容
- 🔧 **代码优化** - 提交 Pull Request

🙏 致谢

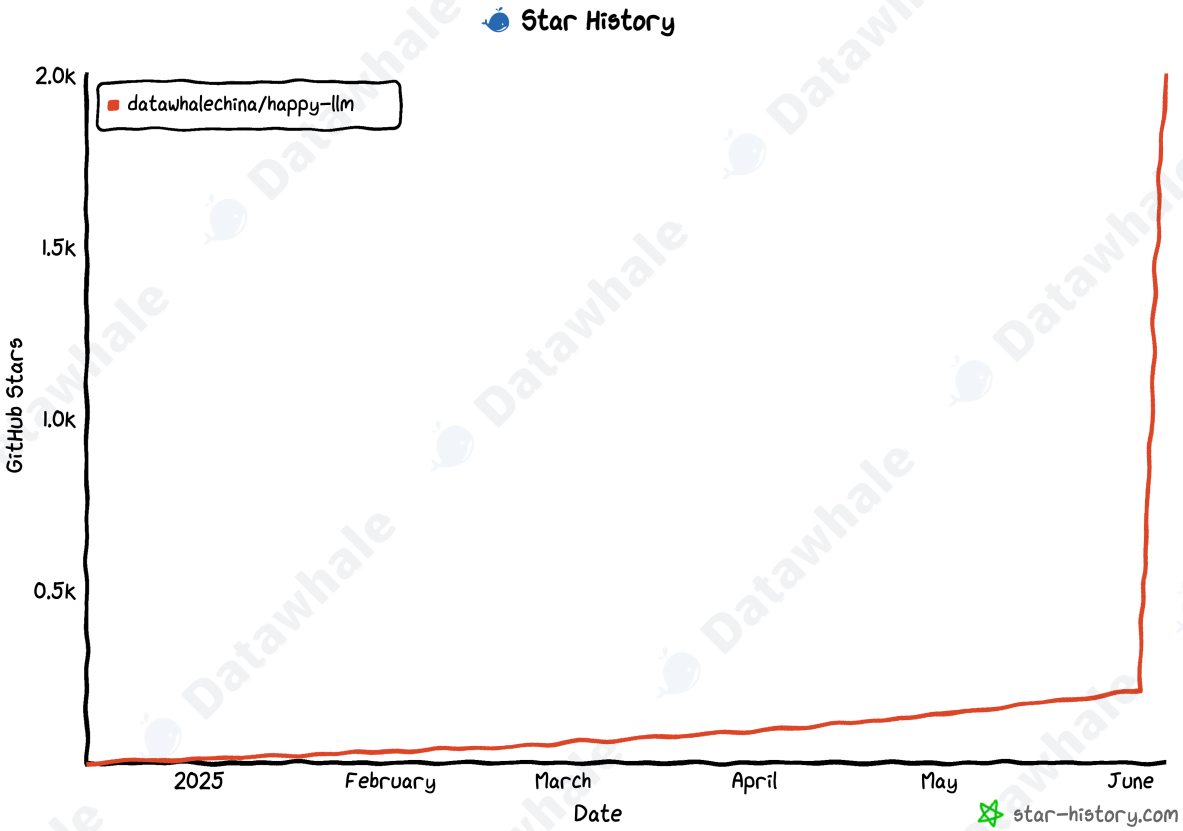
核心贡献者

- 宋志学-项目负责人 (Datawhale成员-中国矿业大学(北京))
- 邹雨衡-项目负责人 (Datawhale成员-对外经济贸易大学)
- 朱信忠-指导专家（Datawhale首席科学家-浙江师范大学杭州人工智能研究院教授）

特别感谢

- 感谢 @Sm1les 对本项目的帮助与支持
- 感谢所有为本项目做出贡献的开发者们 ❤️

Star History



⭐ 如果这个项目对你有帮助，请给我们一个 Star！

关于 Datawhale

扫描二维码关注 Datawhale 公众号，获取更多优质开源内容

开源协议

前言

2022年底，ChatGPT 的横空出世改变了人们对人工智能的认知，也给自然语言处理（Natural Language Process，NLP）领域带来了阶段性的变革，以 GPT 系列模型为代表的大语言模型（Large Language Model，LLM）成为 NLP 乃至人工智能领域的研究主流。自 2023年至今，LLM 始终是人工智能领域的核心话题，引发了一轮又一轮的科技浪潮。

LLM 其实是 NLP 领域经典研究方法预训练语言模型（Pretrain Langguage Model，PLM）的一种衍生成果。NLP 领域聚焦于人类书写的自然语言文本的处理、理解和生成，从诞生至今经历了符号主义阶段、统计学习阶段、深度学习阶段、预训练模型阶段到而今大模型阶段的多次变革。以 GPT、BERT 为代表的 PLM 是上一阶段 NLP 领域的核心研究成果，以注意力机制为模型架构，通过预训练-微调的阶段思想通过在海量无监督文本上进行自监督预训练，实现了强大的自然语言理解能力。但是，传统的 PLM 仍然依赖于一定量有监督数据进行下游任务微调，且在自然语言生成任务上性能还不尽如人意，NLP 系统的性能距离人们所期待的通用人工智能还有不小的差距。

LLM 是在 PLM 的基础上，通过大量扩大模型参数、预训练数据规模，并引入指令微调、人类反馈强化学习等手段实现的突破性成果。相较于传统 PLM，LLM 具备涌现能力，具有强大的上下文学习能力、指令理解能力和文本生成能力。在大模型阶段，NLP 研究者可以一定程度抛弃大量的监督数据标注工作，通过提供少量监督示例，LLM 即能在指定下游任务上达到媲美大规模微调 PLM 的性能。同时，强大的指令理解能力与文本生成能力使 LLM 能够直接、高效、准确地响应用户指令，从而真正向通用人工智能的目标逼近。

LLM 的突破性进展激发了 NLP 领域乃至整个人工智能领域的研究热度，海内外高校、研究院、大厂乃至众多传统领域企业都投入到 LLM 研究的浪潮中。自 2023年至今，LLM 阶段性成果层出不穷，模型性能不断刷新上限，从一开始的 ChatGPT，到 GPT-4，再到以 DeepSeek-R1 为代表的推理大模型、以 Qwen-VL 为代表的多模态大模型等更强大、更定制化的模型，LLM 应用也不断涌现出能够提升实际生产力、赋能用户实际生活的创新应用，从”百模大战“到”Agent 元年“，LLM 基座研究或许已趋向稳定的格局，LLM 的研究始终方兴未艾。可以肯定的是，在并不遥远的未来，LLM 及以 LLM 为基础的应用一定会成为人们生活中的基础设施，与每个人的生活、学习、工作密不可分。

在这样的背景下，深入理解、掌握 LLM 原理，能够动手应用、训练任意一个 LLM 的能力，对每一位 NLP 研究者乃至其他领域的 AI 研究者至关重要。我们在 2023年底分别创建了 self-llm（开源大模型食用指南：https://github.com/datawhalechina/self-llm ）、llm-universe（动手学大模型应用开发：https://github.com/datawhalechina/llm-universe ）两个原创开源大模型教程，前者旨在为开发者提供一站式开源 LLM 部署、推理、微调的使用教程，后者旨在指导开发者从零开始搭建自己的 LLM 应用。两个教程都帮助到了广泛的国内外开发者，也获得了众多开发者的支持和认可，在学习者的反馈中，我们发现目前还缺乏一个从零开始讲解 LLM 原理、并引导学习者亲手搭建、训练 LLM 的完整教程。

鉴于此，我们编写了这本结合 LLM 原理及实战的教程。本书将从 NLP 的基本研究方法出发，根据 LLM 的思路及原理逐层深入，依次为读者剖析 LLM 的架构基础和训练过程。同时，我们会结合目前 LLM 领域最主流的代码框架，演练如何亲手搭建、训练一个 LLM，期以实现授之以鱼，更授之以渔。希望读者能从这本书开始走入 LLM 的浩瀚世界，探索 LLM 的无尽可能。

写给读者的建议

本书包含 LLM 的理论基础、原理介绍和项目实战，全书包括 LLM 及 NLP 的核心思路剖析、公式解析与代码实战，旨在帮助开发者深入理解并掌握 LLM 的基本原理与应用。因此，本书适合大学生、研究人员、LLM 爱好者阅读。在阅读本书之前，你需要具备一定的编程经验，尤其是要对 Python 编程语言有一定的了解。同时，你最好具备深度学习的相关知识，并了解 NLP 领域的相关概念和术语，以便更轻松地阅读本书。

本书分为两部分——基础知识与实战应用。第1章～第4章是基础知识部分，从浅入深介绍 LLM 的基本原理。其中，第1章简单介绍 NLP 的基本任务和发展，为非 NLP 领域研究者提供参考；第2章介绍 LLM 的基本架构——Transformer，包括原理介绍及代码实现，作为 LLM 最重要的理论基础；第3章整体介绍经典的 PLM，包括 Encoder-Only、Encoder-Decoder 和 Decoder-Only 三种架构，也同时介绍了当前一些主流 LLM 的架构和思想；第4章则正式进入 LLM 部分，详细介绍 LLM 的特点、能力和整体训练过程。第5章～第7章是实战应用部分，将逐步带领读者深入 LLM 的底层细节。其中，第5章将带领读者基于 PyTorch 层亲手搭建一个 LLM，并实现预训练、有监督微调的全流程；第6章将引入目前业界主流的 LLM 训练框架 Transformers，带领读者基于该框架快速、高效地实现 LLM 训练过程；第7章则将介绍 基于 LLM 的各种应用，补全读者对 LLM 体系的认知，包括 LLM 的评测、检索增强生产（Retrieval-Augmented Generation，RAG）、智能体（Agent）的思想和简单实现。你可以根据个人兴趣和需求，选择性地阅读相关章节。

在阅读本书的过程中，建议你将理论和实际相结合。LLM 是一个快速发展、注重实践的领域，我们建议你多投入实战，复现本书提供的各种代码，同时积极参加 LLM 相关的项目与比赛，真正投入到 LLM 开发的浪潮中。我们鼓励你关注 Datawhale 及其他 LLM 相关开源社区，当遇到问题时，你可以随时在 Datawhale 社区提问。Datawhale 也会始终跟进 LLM 及其他人工智能技术的发展，欢迎你关注或加入到 Datawhale 社区的共建中。

最后，欢迎每一位读者在阅读完本书后加入到 LLM 开发者的行列。作为国内 AI 开源社区，我们希望充分聚集共创者，一起丰富这个开源 LLM 的世界，打造更多、更全面特色 LLM 的教程。星火点点，汇聚成海。我们希望成为 LLM 与普罗大众的阶梯，以自由、平等的开源精神，拥抱更恢弘而辽阔的 LLM 世界。

感谢你选择本书，祝你阅读愉快！

第一章 NLP 基础概念

自然语言处理（Natural Language Processing，NLP）作为人工智能领域的一个重要分支，旨在使计算机能够理解和处理人类语言，实现人机之间的自然交流。随着信息技术的飞速发展，文本数据已成为我们日常生活中不可或缺的一部分，NLP技术的进步为我们从海量文本中提取有用信息、理解语言的深层含义提供了强有力的工具。从早期的基于规则的方法，到后来的统计学习方法，再到当前深度学习技术的广泛应用，NLP领域经历了多次技术革新，文本表示作为NLP的核心技术之一，其研究和进步对于提升NLP系统的性能具有决定性的作用。

欢迎大家来到 NLP 基础概念的学习，本章节将为大家介绍 NLP 的基础概念，帮助大家更好地理解和回顾 NLP 的相关知识。

1.1 什么是 NLP

NLP 是 一种让计算机理解、解释和生成人类语言的技术。它是人工智能领域中一个极为活跃和重要的研究方向，其核心任务是通过计算机程序来模拟人类对语言的认知和使用过程。NLP 结合了计算机科学、人工智能、语言学和心理学等多个学科的知识和技术，旨在打破人类语言和计算机语言之间的障碍，实现无缝的交流与互动。

NLP技术使得计算机能够执行各种复杂的语言处理任务，如中文分词、子词切分、词性标注、文本分类、实体识别、关系抽取、文本摘要、机器翻译、自动问答等。这些任务不仅要求计算机能够识别和处理语言的表层结构，更重要的是可以理解语言背后的深层含义，包括语义、语境、情感和文化等方面的复杂因素。

随着深度学习等现代技术的发展，NLP 已经取得了显著的进步。通过训练大量的数据，深度学习模型能够学习到语言的复杂模式和结构，从而在多个 NLP 任务上取得了接近甚至超越人类水平的性能。然而，尽管如此，NLP 仍然面临着诸多挑战，如处理歧义性、理解抽象概念、处理隐喻和讽刺等。研究人员正致力于通过更加先进的算法、更大规模的数据集和更精细的语言模型来解决这些问题，以推动NLP技术不断发展。

1.2 NLP 发展历程

NLP 的发展历程是从早期的规则基础方法，到统计方法，再到现在的机器学习和深度学习方法的演变过程。每一次技术变革都极大地推动了 NLP 技术的发展，使其在机器翻译、情感分析、实体识别和文本摘要等任务上取得了显著成就。随着计算能力的不断增强和算法的不断优化，NLP的未来将更加光明，能够在更多领域发挥更加重要的作用。

早期探索（1940年代 - 1960年代）

NLP 的早期探索始于二战后，当时人们认识到将一种语言自动翻译为另一种语言的重要性。1950年，艾伦·图灵提出了图灵测试。

> 他说，如果一台机器可以通过使用打字机成为对话的一部分，并且能够完全模仿人类，没有明显的差异，那么机器可以被认为是能够思考的。

这是判断机器是否能够展现出与人类不可区分的智能行为的测试。这一时期，诺姆·乔姆斯基提出了生成语法理论，这对理解机器翻译的工作方式产生了重要影响。然而，这一时期的机器翻译系统非常简单，主要依赖字典查找和基本的词序规则来进行翻译，效果并不理想。

符号主义与统计方法（1970年代 - 1990年代）

1970年代以后，NLP 研究者开始探索新的领域，包括逻辑基础的范式和自然语言理解。这一时期，研究者分为符号主义（或规则基础）和统计方法两大阵营。符号主义研究者关注于形式语言和生成语法，而统计方法的研究者更加关注于统计和概率方法。1980年代，随着计算能力的提升和机器学习算法的引入，NLP领域出现了革命性的变化，统计模型开始取代复杂的“手写”规则。

机器学习与深度学习（2000年代至今）

2000年代以后，随着深度学习技术的发展，NLP 领域取得了显著的进步。深度学习模型如循环神经网络（Recurrent Neural Network，RNN）、长短时记忆网络（Long Short-Term Memory，LSTM）和注意力机制等技术被广泛应用于 NLP 任务中，取得了令人瞩目的成果。2013年，Word2Vec模型的提出开创了词向量表示的新时代，为NLP任务提供了更加有效的文本表示方法。2018年，BERT模型的问世引领了预训练语言模型的新浪潮，为NLP技术的发展带来了新的机遇和挑战。近年来，基于Transformer的模型，如GPT-3，通过训练巨大参数的模型，能够生成高质量的文本，甚至在某些情况下可以与人类写作相媲美。

1.3 NLP 任务

在NLP的广阔研究领域中，有几个核心任务构成了NLP领域的基础，它们涵盖了从文本的基本处理到复杂的语义理解和生成的各个方面。这些任务包括但不限于中文分词、子词切分、词性标注、文本分类、实体识别、关系抽取、文本摘要、机器翻译以及自动问答系统的开发。每一项任务都有其特定的挑战和应用场景，它们共同推动了语言技术的发展，为处理和分析日益增长的文本数据提供了强大的工具。

1.3.1 中文分词

中文分词（Chinese Word Segmentation, CWS）是 NLP 领域中的一个基础任务。在处理中文文本时，由于中文语言的特点，词与词之间没有像英文那样的明显分隔（如空格），所以无法直接通过空格来确定词的边界。因此，中文分词成为了中文文本处理的首要步骤，其目的是将连续的中文文本切分成有意义的词汇序列。

```
输入：今天天气真好，适合出去游玩。

输出：["今天", "天气", "真", "好", "，", "适合", "出去", "游玩", "。"]
```

正确的分词结果对于后续的词性标注、实体识别、句法分析等任务至关重要。如果分词不准确，将直接影响到整个文本处理流程的效果。

1.3.2 子词切分

子词切分（Subword Segmentation）是 NLP 领域中的一种常见的文本预处理技术，旨在将词汇进一步分解为更小的单位，即子词。子词切分特别适用于处理词汇稀疏问题，即当遇到罕见词或未见过的新词时，能够通过已知的子词单位来理解或生成这些词汇。子词切分在处理那些拼写复杂、合成词多的语言（如德语）或者在预训练语言模型（如BERT、GPT系列）中尤为重要。

子词切分的方法有很多种，常见的有Byte Pair Encoding (BPE)、WordPiece、Unigram、SentencePiece等。这些方法的基本思想是将单词分解成更小的、频繁出现的片段，这些片段可以是单个字符、字符组合或者词根和词缀。

```
输出：unhappiness

不使用子词切分：整个单词作为一个单位：“unhappiness”
使用子词切分（假设BPE算法）：单词被分割为：“un”、“happi”、“ness”
```

在这个例子中，通过子词切分，“unhappiness”这个词被分解成了三个部分：前缀“un”表示否定，“happi”是“happy”的词根变体，表示幸福，“ness”是名词后缀，表示状态。即使模型从未见过“unhappiness”这个完整的单词，它也可以通过这些已知的子词来理解其大致意思为“不幸福的状态”。

1.3.3 词性标注

词性标注（Part-of-Speech Tagging，POS Tagging）是 NLP 领域中的一项基础任务，它的目标是为文本中的每个单词分配一个词性标签，如名词、动词、形容词等。这个过程通常基于预先定义的词性标签集，如英语中的常见标签有名词（Noun，N）、动词（Verb，V）、形容词（Adjective，Adj）等。词性标注对于理解句子结构、进行句法分析、语义角色标注等高级NLP任务至关重要。通过词性标注，计算机可以更好地理解文本的含义，进而进行信息提取、情感分析、机器翻译等更复杂的处理。

假设我们有一个英文句子：She is playing the guitar in the park.

词性标注的结果如下：

- She (代词，Pronoun，PRP)
- is (动词，Verb，VBZ)
- playing (动词的现在分词，Verb，VBG)
- the (限定词，Determiner，DT)
- guitar (名词，Noun，NN)
- in (介词，Preposition，IN)
- the (限定词，Determiner，DT)
- park (名词，Noun，NN)
- . (标点，Punctuation，.)

词性标注通常依赖于机器学习模型，如隐马尔可夫模型（Hidden Markov Model，HMM）、条件随机场（Conditional Random Field，CRF）或者基于深度学习的循环神经网络 RNN 和长短时记忆网络 LSTM 等。这些模型通过学习大量的标注数据来预测新句子中每个单词的词性。

1.3.4 文本分类

文本分类（Text Classification）是 NLP 领域的一项核心任务，涉及到将给定的文本自动分配到一个或多个预定义的类别中。这项技术广泛应用于各种场景，包括但不限于情感分析、垃圾邮件检测、新闻分类、主题识别等。文本分类的关键在于理解文本的含义和上下文，并基于此将文本映射到特定的类别。

假设有一个文本分类任务，目的是将新闻文章分类为“体育”、“政治”或“科技”三个类别之一。

```
文本：“NBA季后赛将于下周开始，湖人和勇士将在首轮对决。”
类别：“体育”

文本：“美国总统宣布将提高关税，引发国际贸易争端。”
类别：“政治”

文本：“苹果公司发布了新款 Macbook，配备了最新的m3芯片。”
类别：“科技”
```

文本分类任务的成功关键在于选择合适的特征表示和分类算法，以及拥有高质量的训练数据。随着深度学习技术的发展，使用神经网络进行文本分类已经成为一种趋势，它们能够捕捉到文本数据中的复杂模式和语义信息，从而在许多任务中取得了显著的性能提升。

1.3.5 实体识别

实体识别（Named Entity Recognition, NER），也称为命名实体识别，是 NLP 领域的一个关键任务，旨在自动识别文本中具有特定意义的实体，并将它们分类为预定义的类别，如人名、地点、组织、日期、时间等。实体识别任务对于信息提取、知识图谱构建、问答系统、内容推荐等应用很重要，它能够帮助系统理解文本中的关键元素及其属性。

假设有一个实体识别任务，目的是从文本中识别出人名、地名和组织名等实体。

```
输入：李雷和韩梅梅是北京市海淀区的居民，他们计划在2024年4月7日去上海旅行。

输出：[("李雷", "人名"), ("韩梅梅", "人名"), ("北京市海淀区", "地名"), ("2024年4月7日", "日期"), ("上海", "地名")]
```

通过实体识别任务，我们不仅能识别出文本中的实体，还能了解它们的类别，为深入理解文本内容和上下文提供了重要信息。随着NLP技术的发展，实体识别的精度和效率不断提高，可以为各种NLP应用提供强大的支持。

1.3.6 关系抽取

关系抽取（Relation Extraction）是 NLP 领域中的一项关键任务，它的目标是从文本中识别实体之间的语义关系。这些关系可以是因果关系、拥有关系、亲属关系、地理位置关系等，关系抽取对于理解文本内容、构建知识图谱、提升机器理解语言的能力等方面具有重要意义。

假设我们有以下句子：

```
输入：比尔·盖茨是微软公司的创始人。

输出：[("比尔·盖茨", "创始人", "微软公司")]
```

在这个例子中，关系抽取任务的目标是从文本中识别出“比尔·盖茨”和“微软公司”之间的“创始人”关系。通过关系抽取，我们可以从文本中提取出有用的信息，帮助计算机更好地理解文本内容，为后续的知识图谱构建、问答系统等任务提供支持。

1.3.7 文本摘要

文本摘要（Text Summarization）是 NLP 中的一个重要任务，目的是生成一段简洁准确的摘要，来概括原文的主要内容。根据生成方式的不同，文本摘要可以分为两大类：抽取式摘要（Extractive Summarization）和生成式摘要（Abstractive Summarization）。

- 抽取式摘要：抽取式摘要通过直接从原文中选取关键句子或短语来组成摘要。优点是摘要中的信息完全来自原文，因此准确性较高。然而，由于仅仅是原文中句子的拼接，有时候生成的摘要可能不够流畅。
- 生成式摘要：与抽取式摘要不同，生成式摘要不仅涉及选择文本片段，还需要对这些片段进行重新组织和改写，并生成新的内容。生成式摘要更具挑战性，因为它需要理解文本的深层含义，并能够以新的方式表达相同的信息。生成式摘要通常需要更复杂的模型，如基于注意力机制的序列到序列模型（Seq2Seq）。

假设我们有以下新闻报道：

```
2021年5月22日，国家航天局宣布，我国自主研发的火星探测器“天问一号”成功在火星表面着陆。此次任务的成功，标志着我国在深空探测领域迈出了重要一步。“天问一号”搭载了多种科学仪器，将在火星表面进行为期90个火星日的科学探测工作，旨在研究火星地质结构、气候条件以及寻找生命存在的可能性。
```

抽取式摘要：

```
我国自主研发的火星探测器“天问一号”成功在火星表面着陆，标志着我国在深空探测领域迈出了重要一步。
```

生成式摘要：

```
“天问一号”探测器成功实现火星着陆，代表我国在宇宙探索中取得重大进展。
```

文本摘要任务在信息检索、新闻推送、报告生成等领域有着广泛的应用。通过自动摘要，用户可以快速获取文本的核心信息，节省阅读时间，提高信息处理效率。

1.3.8 机器翻译

机器翻译（Machine Translation, MT）是 NLP 领域的一项核心任务，指使用计算机程序将一种自然语言（源语言）自动翻译成另一种自然语言（目标语言）的过程。机器翻译不仅涉及到词汇的直接转换，更重要的是要准确传达源语言文本的语义、风格和文化背景等，使得翻译结果在目标语言中自然、准确、流畅，以便跨越语言障碍，促进不同语言使用者之间的交流与理解。

假设我们有一句中文：“今天天气很好。”，我们想要将其翻译成英文。

```
源语言：今天天气很好。

目标语言：The weather is very nice today.
```

在这个简单的例子中，机器翻译能够准确地将中文句子转换成英文，保持了原句的意义和结构。然而，在处理更长、更复杂的文本时，机器翻译面临的挑战也会相应增加。为了提高机器翻译的质量，研究者不断探索新的方法和技术，如基于神经网络的Seq2Seq模型、Transformer模型等，这些模型能够学习到源语言和目标语言之间的复杂映射关系，从而实现更加准确和流畅的翻译。

1.3.9 自动问答

自动问答（Automatic Question Answering, QA）是 NLP 领域中的一个高级任务，旨在使计算机能够理解自然语言提出的问题，并根据给定的数据源自动提供准确的答案。自动问答任务模拟了人类理解和回答问题的能力，涵盖了从简单的事实查询到复杂的推理和解释。自动问答系统的构建涉及多个NLP子任务，如信息检索、文本理解、知识表示和推理等。

自动问答大致可分为三类：检索式问答（Retrieval-based QA）、知识库问答（Knowledge-based QA）和社区问答（Community-based QA）。检索式问答通过搜索引擎等方式从大量文本中检索答案；知识库问答通过结构化的知识库来回答问题；社区问答则依赖于用户生成的问答数据，如问答社区、论坛等。

自动问答系统的开发和优化是一个持续的过程，随着技术的进步和算法的改进，这些系统在准确性、理解能力和应用范围上都有显著的提升。通过结合不同类型的数据源和技术方法，自动问答系统正变得越来越智能，越来越能够处理复杂和多样化的问题。

1.4 文本表示的发展历程

文本表示的目的是将人类语言的自然形式转化为计算机可以处理的形式，也就是将文本数据数字化，使计算机能够对文本进行有效的分析和处理。文本表示是 NLP 领域中的一项基础性和必要性工作，它直接影响甚至决定着 NLP 系统的质量和性能。

在 NLP 中，文本表示涉及到将文本中的语言单位（如字、词、短语、句子等）以及它们之间的关系和结构信息转换为计算机能够理解和操作的形式，例如向量、矩阵或其他数据结构。这样的表示不仅需要保留足够的语义信息，以便于后续的 NLP 任务，如文本分类、情感分析、机器翻译等，还需要考虑计算效率和存储效率。

文本表示的发展历程经历了多个阶段，从早期的基于规则的方法，到统计学习方法，再到当前的深度学习技术，文本表示技术不断演进，为 NLP 的发展提供了强大的支持。

1.4.1 词向量

向量空间模型（Vector Space Model, VSM）是 NLP 领域中一个基础且强大的文本表示方法，最早由哈佛大学Salton提出。向量空间模型通过将文本（包括单词、句子、段落或整个文档）转换为高维空间中的向量来实现文本的数学化表示。在这个模型中，每个维度代表一个特征项（例如，字、词、词组或短语），而向量中的每个元素值代表该特征项在文本中的权重，这种权重通过特定的计算公式（如词频TF、逆文档频率TF-IDF等）来确定，反映了特征项在文本中的重要程度。

向量空间模型的应用极其广泛，包括但不限于文本相似度计算、文本分类、信息检索等自然语言处理任务。它将复杂的文本数据转换为易于计算和分析的数学形式，使得文本的相似度计算和模式识别成为可能。此外，通过矩阵运算如特征值计算、奇异值分解（singular value decomposition, SVD）等方法，可以优化文本向量表示，进一步提升处理效率和效果。

然而，向量空间模型也存在很多问题。其中最主要的是数据稀疏性和维数灾难问题，因为特征项数量庞大导致向量维度极高，同时多数元素值为零。此外，由于模型基于特征项之间的独立性假设，忽略了文本中的结构信息，如词序和上下文信息，限制了模型的表现力。特征项的选择和权重计算方法的不足也是向量空间模型需要解决的问题。

为了解决这些问题，研究者们对向量空间模型的研究主要集中在两个方面：一是改进特征表示方法，如借助图方法、主题方法等进行关键词抽取；二是改进和优化特征项权重的计算方法，可以在现有方法的基础上进行融合计算或提出新的计算方法.

1.4.2 语言模型

N-gram 模型是 NLP 领域中一种基于统计的语言模型，广泛应用于语音识别、手写识别、拼写纠错、机器翻译和搜索引擎等众多任务。N-gram模型的核心思想是基于马尔可夫假设，即一个词的出现概率仅依赖于它前面的N-1个词。这里的N代表连续出现单词的数量，可以是任意正整数。例如，当N=1时，模型称为unigram，仅考虑单个词的概率；当N=2时，称为bigram，考虑前一个词来估计当前词的概率；当N=3时，称为trigram，考虑前两个词来估计第三个词的概率，以此类推N-gram。

N-gram模型通过条件概率链式规则来估计整个句子的概率。具体而言，对于给定的一个句子，模型会计算每个N-gram出现的条件概率，并将这些概率相乘以得到整个句子的概率。例如，对于句子“The quick brown fox”，作为trigram模型，我们会计算 $P(\text{"brown"}|\text{"The"},\text{"quick"})$、$P(\text{"fox"}|\text{"quick"},\text{"brown"})$等概率，并将它们相乘。

N-gram的优点是实现简单、容易理解，在许多任务中效果不错。但当N较大时，会出现数据稀疏性问题。模型的参数空间会急剧增大，相同的N-gram序列出现的概率变得非常低，导致模型无法有效学习，模型泛化能力下降。此外，N-gram模型忽略了词之间的范围依赖关系，无法捕捉到句子中的复杂结构和语义信息。

尽管存在局限性，N-gram模型由于其简单性和实用性，在许多 NLP 任务中仍然被广泛使用。在某些应用中，结合N-gram模型和其他技术（如深度学习模型）可以获得更好的性能。

1.4.3 Word2Vec

Word2Vec是一种流行的词嵌入（Word Embedding）技术，由Tomas Mikolov等人在2013年提出。它是一种基于神经网络NNLM的语言模型，旨在通过学习词与词之间的上下文关系来生成词的密集向量表示。Word2Vec的核心思想是利用词在文本中的上下文信息来捕捉词之间的语义关系，从而使得语义相似或相关的词在向量空间中距离较近。

Word2Vec模型主要有两种架构：连续词袋模型CBOW(Continuous Bag of Words)是根据目标词上下文中的词对应的词向量, 计算并输出目标词的向量表示；Skip-Gram模型与CBOW模型相反, 是利用目标词的向量表示计算上下文中的词向量. 实践验证CBOW适用于小型数据集, 而Skip-Gram在大型语料中表现更好。

相比于传统的高维稀疏表示（如One-Hot编码），Word2Vec生成的是低维（通常几百维）的密集向量，有助于减少计算复杂度和存储需求。Word2Vec模型能够捕捉到词与词之间的语义关系，比如"国王"和"王后"在向量空间中的位置会比较接近，因为在大量文本中，它们通常会出现在相似的上下文中。Word2Vec模型也可以很好的泛化到未见过的词，因为它是基于上下文信息学习的，而不是基于词典。但由于CBOW/Skip-Gram模型是基于局部上下文的，无法捕捉到长距离的依赖关系，缺乏整体的词与词之间的关系，因此在一些复杂的语义任务上表现不佳。

1.4.4 ELMo

ELMo（Embeddings from Language Models）实现了一词多义、静态词向量到动态词向量的跨越式转变。首先在大型语料库上训练语言模型，得到词向量模型，然后在特定任务上对模型进行微调，得到更适合该任务的词向量，ELMo首次将预训练思想引入到词向量的生成中，使用双向LSTM结构，能够捕捉到词汇的上下文信息，生成更加丰富和准确的词向量表示。

ELMo采用典型的两阶段过程: 第1个阶段是利用语言模型进行预训练; 第2个阶段是在做特定任务时, 从预训练网络中提取对应单词的词向量作为新特征补充到下游任务中。基于RNN的LSTM模型训练时间长, 特征提取是ELMo模型优化和提升的关键。

ELMo模型的主要优势在于其能够捕捉到词汇的多义性和上下文信息，生成的词向量更加丰富和准确，适用于多种 NLP 任务。然而，ELMo模型也存在一些问题，如模型复杂度高、训练时间长、计算资源消耗大等。

参考文献

[1] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean. (2013). *Distributed Representations of Words and Phrases and their Compositionality.* arXiv preprint arXiv:1310.4546.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint arXiv:1810.04805.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. (2023). *Attention Is All You Need.* arXiv preprint arXiv:1706.03762.

[4] Malek Hajjem, Chiraz Latiri. (2017). *Combining IR and LDA Topic Modeling for Filtering Microblogs.* Procedia Computer Science, 112, 761–770. https://doi.org/10.1016/j.procs.2017.08.166.

[5] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer. (2018). *Deep contextualized word representations.* arXiv preprint arXiv:1802.05365.

[6] Salton, G., Wong, A., Yang, C. S. (1975). *A vector space model for automatic indexing.* Communications of the ACM, 18(11), 613–620. https://doi.org/10.1145/361219.361220.

[7] 赵京胜,宋梦雪,高祥,等.自然语言处理中的文本表示研究[J].软件学报,2022,33(01):102-128.DOI:10.13328/j.cnki.jos.006304.

[8] 中文信息处理发展报告（2016）前言[C]//中文信息处理发展报告（2016）.中国中文信息学会;,2016:2-3.DOI:10.26914/c.cnkihy.2016.003326.

第二章 Transformer 架构

2.1 注意力机制

2.1.1 什么是注意力机制

随着 NLP 从统计机器学习向深度学习迈进，作为 NLP 核心问题的文本表示方法也逐渐从统计学习向深度学习迈进。正如我们在第一章所介绍的，文本表示从最初的通过统计学习模型进行计算的向量空间模型、语言模型，通过 Word2Vec 的单层神经网络进入到通过神经网络学习文本表示的时代。但是，从 计算机视觉（Computer Vision，CV）为起源发展起来的神经网络，其核心架构有三种：

- 前馈神经网络（Feedforward Neural Network，FNN），即每一层的神经元都和上下两层的每一个神经元完全连接，如图2.1所示:

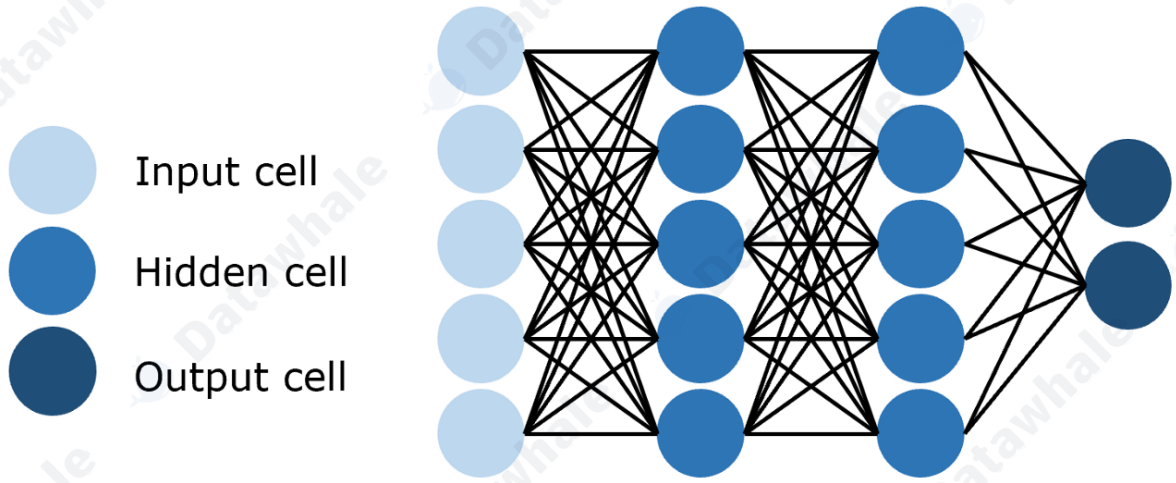


图2.1 前馈神经网络

- 卷积神经网络（Convolutional Neural Network，CNN），即训练参数量远小于前馈神经网络的卷积层来进行特征提取和学习，如图2.2所示:

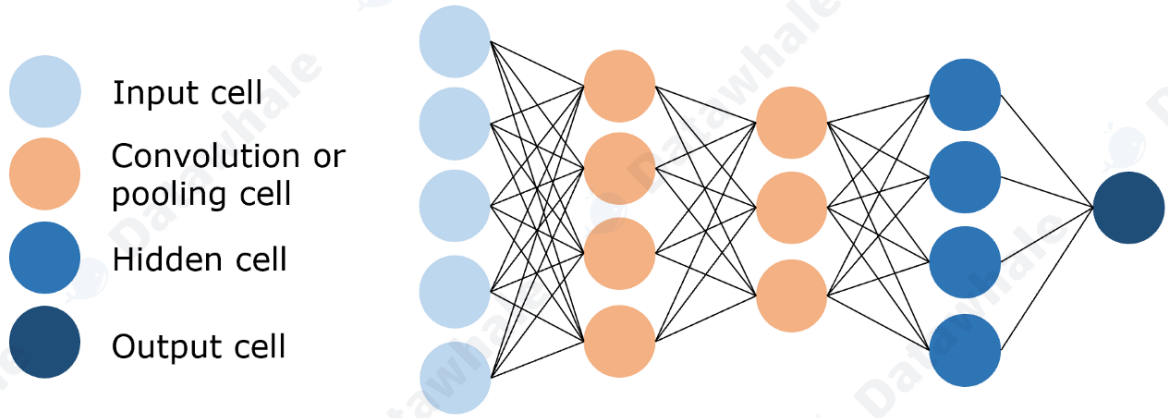


图2.2 卷积神经网络

- 循环神经网络（Recurrent Neural Network，RNN），能够使用历史信息作为输入、包含环和自重复的网络，如图2.3所示:

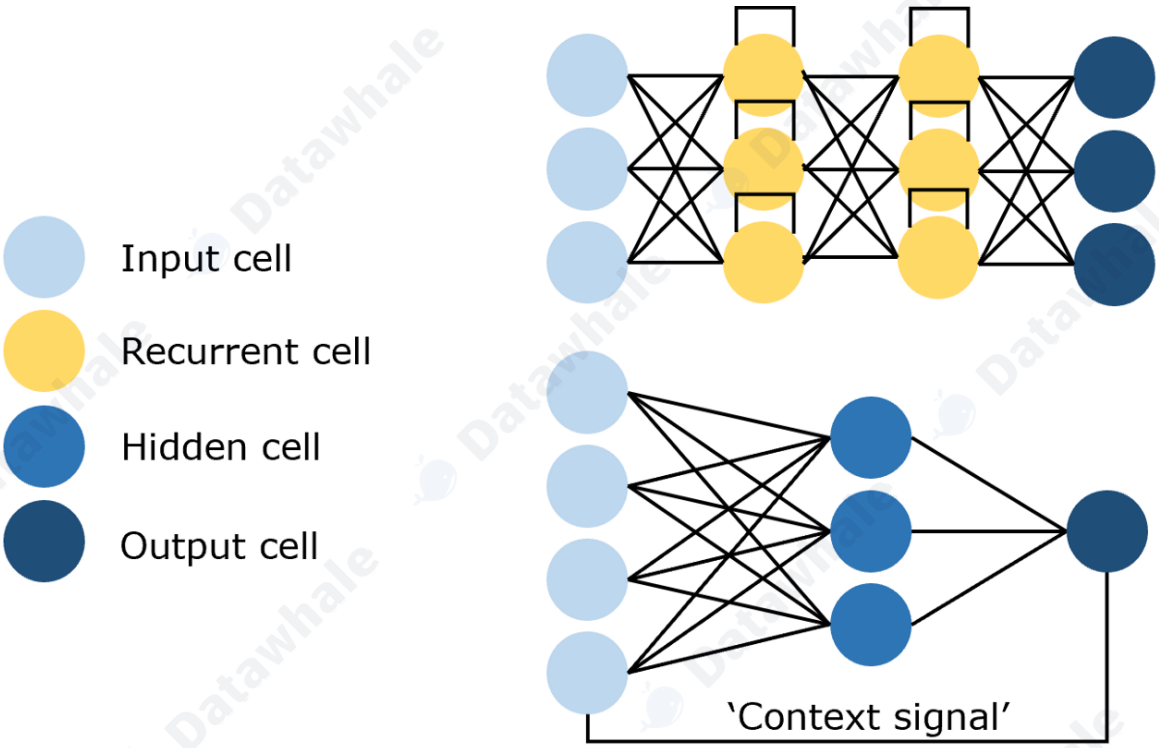


图2.3 循环神经网络

由于 NLP 任务所需要处理的文本往往是序列，因此专用于处理序列、时序数据的 RNN 往往能够在 NLP 任务上取得最优的效果。事实上，在注意力机制横空出世之前，RNN 以及 RNN 的衍生架构 LSTM 是 NLP 领域当之无愧的霸主。例如，我们在第一章讲到过的开创了预训练思想的文本表示模型 ELMo，就是使用的双向 LSTM 作为网络架构。

但 RNN 及 LSTM 虽然具有捕捉时序信息、适合序列生成的优点，却有两个难以弥补的缺陷：

1. 序列依序计算的模式能够很好地模拟时序信息，但限制了计算机并行计算的能力。由于序列需要依次输入、依序计算，图形处理器（Graphics Processing Unit，GPU）并行计算的能力受到了极大限制，导致 RNN 为基础架构的模型虽然参数量不算特别大，但计算时间成本却很高；
2. RNN 难以捕捉长序列的相关关系。在 RNN 架构中，距离越远的输入之间的关系就越难被捕捉，同时 RNN 需要将整个序列读入内存依次计算，也限制了序列的长度。虽然 LSTM 中通过门机制对此进行了一定优化，但对于较远距离相关关系的捕捉，RNN 依旧是不如人意的。

针对这样的问题，Vaswani 等学者参考了在 CV 领域被提出、被经常融入到 RNN 中使用的注意力机制（Attention）（注意，虽然注意力机制在 NLP 被发扬光大，但其确实是在 CV 领域被提出的），创新性地搭建了完全由注意力机制构成的神经网络——Transformer，也就是大语言模型（Large Language Model，LLM）的鼻祖及核心架构，从而让注意力机制一跃成为深度学习最核心的架构之一。

那么，究竟什么是注意力机制?

注意力机制最先源于计算机视觉领域，其核心思想为当我们关注一张图片，我们往往无需看清楚全部内容而仅将注意力集中在重点部分即可。而在自然语言处理领域，我们往往也可以通过将重点注意力集中在一个或几个 token，从而取得更高效高质的计算效果。

注意力机制有三个核心变量：**Query**（查询值）、**Key**（键值）和 **Value**（真值）。我们可以通过一个案例来理解每一个变量所代表的含义。例如，当我们有一篇新闻报道，我们想要找到这个报道的时间，那么，我们的 Query 可以是类似于“时间”、“日期”一类的向量（为了便于理解，此处使用文本来表示，但其实际是稠密的向量），Key 和 Value 会是整个文本。通过对 Query 和 Key 进行运算我们可以得到一个权重，这个权重其实反映了从 Query 出发，对文本每一个 token 应该分布的注意力相对大小。通过把权重和 Value 进行运算，得到的最后结果就是从 Query 出发计算整个文本注意力得到的结果。

具体而言，注意力机制的特点是通过计算 **Query** 与**Key**的相关性为真值加权求和，从而拟合序列中每个词同其他词的相关关系。

2.1.2 深入理解注意力机制

刚刚我们说到，注意力机制有三个核心变量：查询值 Query，键值 Key 和 真值 Value。接下来我们以字典为例，逐步分析注意力机制的计算公式是如何得到的，从而帮助读者深入理解注意力机制。首先，我们有这样一个字典：

```
{
    "apple":10,
    "banana":5,
    "chair":2
}
```

此时，字典的键就是注意力机制中的键值 Key，而字典的值就是真值 Value。字典支持我们进行精确的字符串匹配，例如，如果我们想要查找的值也就是查询值 Query 为“apple”，那么我们可以直接通过将 Query 与 Key 做匹配来得到对应的 Value。

但是，如果我们想要匹配的 Query 是一个包含多个 Key 的概念呢？例如，我们想要查找“fruit”，此时，我们应该将 apple 和 banana 都匹配到，但不能匹配到 chair。因此，我们往往会选择将 Key 对应的 Value 进行组合得到最终的 Value。

例如，当我们的 Query 为“fruit”，我们可以分别给三个 Key 赋予如下的权重：

```
{
    "apple":0.6,
    "banana":0.4,
    "chair":0
}
```

那么，我们最终查询到的值应该是：

$$value = 0.6 * 10 + 0.4 * 5 + 0 * 2 = 8 \tag{1}$$

给不同 Key 所赋予的不同权重，就是我们所说的注意力分数，也就是为了查询到 Query，我们应该赋予给每一个 Key 多少注意力。但是，如何针对每一个 Query，计算出对应的注意力分数呢？从直观上讲，我们可以认为 Key 与 Query 相关性越高，则其所应该赋予的注意力权重就越大。但是，我们如何能够找到一个合理的、能够计算出正确的注意力分数的方法呢？

在第一章中，我们有提到词向量的概念。通过合理的训练拟合，词向量能够表征语义信息，从而让语义相近的词在向量空间中距离更近，语义较远的词在向量空间中距离更远。我们往往用欧式距离来衡量词向量的相似性，但我们同样也可以用点积来进行度量：

$$v \cdot w = \sum_i v_i w_i \tag{2}$$

根据词向量的定义，语义相似的两个词对应的词向量的点积应该大于0，而语义不相似的词向量点积应该小于0。

那么，我们就可以用点积来计算词之间的相似度。假设我们的 Query 为“fruit”，对应的词向量为 q；我们的 Key 对应的词向量为 $k = [v_{apple} v_{banana} v_{chair}]$,则我们可以计算 Query 和每一个键的相似程度：

$$x = qK^T \tag{3}$$

此处的 K 即为将所有 Key 对应的词向量堆叠形成的矩阵。基于矩阵乘法的定义，x 即为 q 与每一个 k 值的点积。现在我们得到的 x 即反映了 Query 和每一个 Key 的相似程度，我们再通过一个 Softmax 层将其转化为和为 1 的权重：

$$\text{softmax}(x)_i = \frac{e^{xi}}{\sum_j e^{x_j}} \tag{4}$$

这样，得到的向量就能够反映 Query 和每一个 Key 的相似程度，同时又相加权重为 1，也就是我们的注意力分数了。最后，我们再将得到的注意力分数和值向量做对应乘积即可。根据上述过程，我们就可以得到注意力机制计算的基本公式：

$$attention(Q, K, V) = softmax(qK^T)v \tag{5}$$

不过，此时的值还是一个标量，同时，我们此次只查询了一个 Query。我们可以将值转化为维度为 d_v 的向量，同时一次性查询多个 Query，同样将多个 Query 对应的词向量堆叠在一起形成矩阵 Q，得到公式：

$$attention(Q, K, V) = softmax(QK^T)V \tag{6}$$

目前，我们离标准的注意力机制公式还差最后一步。在上一个公式中，如果 Q 和 K 对应的维度 d_k 比较大，softmax 放缩时就非常容易受影响，使不同值之间的差异较大，从而影响梯度的稳定性。因此，我们要将 Q 和 K 乘积的结果做一个放缩：

$$attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{7}$$

这也就是注意力机制的核心计算公式了。

2.1.3 注意力机制的实现

基于上文，我们可以很简单地使用 Pytorch 来实现注意力机制的代码：

```
'''注意力计算函数'''
def attention(query, key, value, dropout=None):
    '''
    args:
    query: 查询值矩阵
    key: 键值矩阵
    value: 真值矩阵
    '''
    # 获取键向量的维度，键向量的维度和值向量的维度相同
    d_k = query.size(-1)
    # 计算Q与K的内积并除以根号dk
    # transpose——相当于转置
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    # Softmax
```

```
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
        # 采样
     # 根据计算结果对value进行加权求和
    return torch.matmul(p_attn, value), p_attn
```

注意，在上文代码中，我们假设输入的 q、k、v 是已经经过转化的词向量矩阵，也就是公式中的 Q、K、V。我们仅需要通过上述几行代码，就可以实现核心的注意力机制计算。

2.1.4 自注意力

根据上文的分析，我们可以发现，注意力机制的本质是对两段序列的元素依次进行相似度计算，寻找出一个序列的每个元素对另一个序列的每个元素的相关度，然后基于相关度进行加权，即分配注意力。而这两段序列即是我们计算过程中 Q、K、V 的来源。

但是，在我们的实际应用中，我们往往只需要计算 Query 和 Key 之间的注意力结果，很少存在额外的真值 Value。也就是说，我们其实只需要拟合两个文本序列。在经典的 注意力机制中，Q 往往来自于一个序列，K 与 V 来自于另一个序列，都通过参数矩阵计算得到，从而可以拟合这两个序列之间的关系。例如在 Transformer 的 Decoder 结构中，Q 来自于 Decoder 的输入，K 与 V 来自于 Encoder 的输出，从而拟合了编码信息与历史信息之间的关系，便于综合这两种信息实现未来的预测。

但在 Transformer 的 Encoder 结构中，使用的是 注意力机制的变种 —— 自注意力（self-attention，自注意力）机制。所谓自注意力，即是计算本身序列中每个元素都其他元素的注意力分布，即在计算过程中，Q、K、V 都由同一个输入通过不同的参数矩阵计算得到。在 Encoder 中，Q、K、V 分别是输入对参数矩阵 W_q、W_k、W_v 做积得到，从而拟合输入语句中每一个 token 对其他所有 token 的关系。

通过自注意力机制，我们可以找到一段文本中每一个 token 与其他所有 token 的相关关系大小，从而建模文本之间的依赖关系。在代码中的实现，self-attention 机制其实是通过给 Q、K、V 的输入传入同一个参数实现的：

```
# attention 为上文定义的注意力计算函数
attention(x, x, x)
```

2.1.5 掩码自注意力

掩码自注意力，即 Mask Self-Attention，是指使用注意力掩码的自注意力机制。掩码的作用是遮蔽一些特定位置的 token，模型在学习的过程中，会忽略掉被遮蔽的 token。

使用注意力掩码的核心动机是让模型只能使用历史信息进行预测而不能看到未来信息。使用注意力机制的 Transformer 模型也是通过类似于 n-gram 的语言模型任务来学习的，也就是对一个文本序列，不断根据之前的 token 来预测下一个 token，直到将整个文本序列补全。

例如，如果待学习的文本序列是 【BOS】I like you【EOS】，那么，模型会按如下顺序进行预测和学习：

```
Step 1：输入 【BOS】，输出 I
Step 2：输入 【BOS】I，输出 like
Step 3：输入 【BOS】I like，输出 you
Step 4：输入 【BOS】I like you，输出 【EOS】
```

理论上来说，只要学习的语料足够多，通过上述的过程，模型可以学会任意一种文本序列的建模方式，也就是可以对任意的文本进行补全。

但是，我们可以发现，上述过程是一个串行的过程，也就是需要先完成 Step 1，才能做 Step 2，接下来逐步完成整个序列的补全。我们在一开始就说过，Transformer 相对于 RNN 的核心优势之一即在于其可以并行计算，具有更高的计算效率。如果对于每一个训练语料，模型都需要串行完成上述过程才能完成学习，那么很明显没有做到并行计算，计算效率很低。

针对这个问题，Transformer 就提出了掩码自注意力的方法。掩码自注意力会生成一串掩码，来遮蔽未来信息。例如，我们待学习的文本序列仍然是【BOS】I like you【EOS】，我们使用的注意力掩码是【MASK】，那么模型的输入为：

```
<BOS> 【MASK】【MASK】【MASK】【MASK】
<BOS>    I   【MASK】【MASK】【MASK】
<BOS>    I     like  【MASK】【MASK】
<BOS>    I     like    you   【MASK】
<BoS>    I     like    you   </EOS>
```

在每一行输入中，模型仍然是只看到前面的 token，预测下一个 token。但是注意，上述输入不再是串行的过程，而可以一起并行地输入到模型中，模型只需要每一个样本根据未被遮蔽的 token 来预测下一个 token 即可，从而实现了并行的语言模型。

观察上述的掩码，我们可以发现其实则是一个和文本序列等长的上三角矩阵。我们可以简单地通过创建一个和输入同等长度的上三角矩阵作为注意力掩码，再使用掩码来遮蔽掉输入即可。也就是说，当输入维度为 （batch_size, seq_len, hidden_size）时，我们的 Mask 矩阵维度一般为 (1, seq_len, seq_len)（通过广播实现同一个 batch 中不同样本的计算）。

在具体实现中，我们通过以下代码生成 Mask 矩阵：

```
# 创建一个上三角矩阵，用于遮蔽未来信息。
# 先通过 full 函数创建一个 1 * seq_len * seq_len 的矩阵
mask = torch.full((1, args.max_seq_len, args.max_seq_len), float("-inf"))
# triu 函数的功能是创建一个上三角矩阵
mask = torch.triu(mask, diagonal=1)
```

生成的 Mask 矩阵会是一个上三角矩阵，上三角位置的元素均为 -inf，其他位置的元素置为0。

在注意力计算时，我们会将计算得到的注意力分数与这个掩码做和，再进行 Softmax 操作：

```
# 此处的 scores 为计算得到的注意力分数，mask 为上文生成的掩码矩阵
scores = scores + mask[:, :seqlen, :seqlen]
scores = F.softmax(scores.float(), dim=-1).type_as(xq)
```

通过做求和，上三角区域（也就是应该被遮蔽的 token 对应的位置）的注意力分数结果都变成了 `-inf`，而下三角区域的分数不变。再做 Softmax 操作，`-inf` 的值在经过 Softmax 之后会被置为 0，从而忽略了上三角区域计算的注意力分数，从而实现了注意力遮蔽。

2.1.6 多头注意力

注意力机制可以实现并行化与长期依赖关系拟合，但一次注意力计算只能拟合一种相关关系，单一的注意力机制很难全面拟合语句序列里的相关关系。因此 Transformer 使用了多头注意力机制（Multi-Head Attention），即同时对一个语料进行多次注意力计算，每次注意力计算都能拟合不同的关系，将最后的多次结果拼接起来作为最后的输出，即可更全面深入地拟合语言信息。

在原论文中，作者也通过实验证实，多头注意力计算中，每个不同的注意力头能够拟合语句中的不同信息，如图2.4所示：

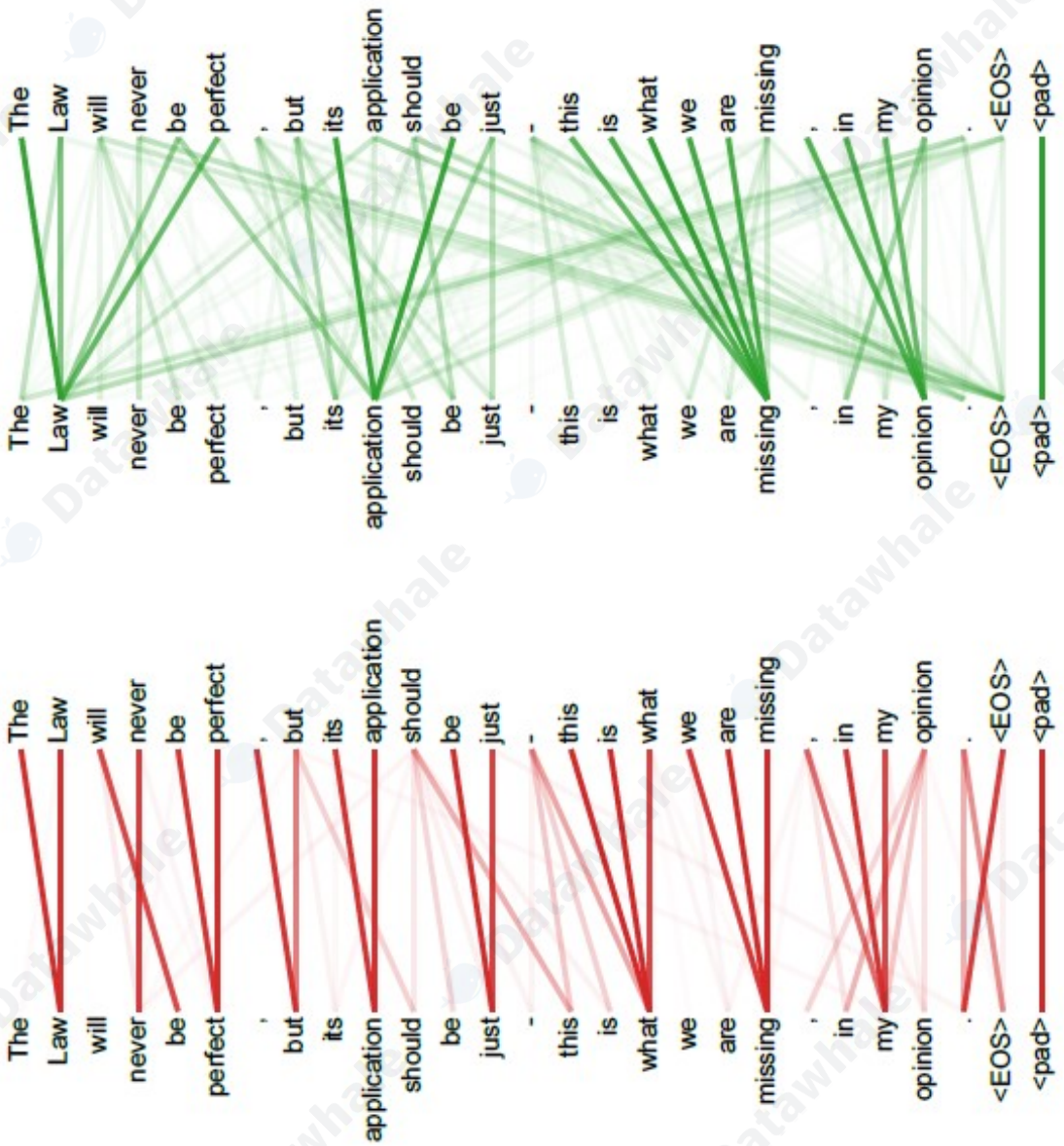


图2.4 多头注意力机制

上层与下层分别是两个注意力头对同一段语句序列进行自注意力计算的结果，可以看到，对于不同的注意力头，能够拟合不同层次的相关信息。通过多个注意力头同时计算，能够更全面地拟合语句关系。

事实上，所谓的多头注意力机制其实就是将原始的输入序列进行多组的自注意力处理；然后再将每一组得到的自注意力结果拼接起来，再通过一个线性层进行处理，得到最终的输出。我们用公式可以表示为：

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_\text{h})W^O \\ \text{where head}_\text{i} &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{8}$$

其最直观的代码实现并不复杂，即 n 个头就有 n 组3个参数矩阵，每一组进行同样的注意力计算，但由于是不同的参数矩阵从而通过反向传播实现了不同的注意力结果，然后将 n 个结果拼接起来输出即可。

但上述实现时空复杂度均较高，我们可以通过矩阵运算巧妙地实现并行的多头计算，其核心逻辑在于使用三个组合矩阵来代替了n个参数矩阵的组合，也就是矩阵内积再拼接其实等同于拼接矩阵再内积。具体实现可以参考下列代码：

```
import torch.nn as nn
import torch

'''多头自注意力计算模块'''
class MultiHeadAttention(nn.Module):

    def __init__(self, args: ModelArgs, is_causal=False):
        # 构造函数
        # args: 配置对象
        super().__init__()
        # 隐藏层维度必须是头数的整数倍，因为后面我们会将输入拆成头数个矩阵
        assert args.n_embd % args.n_head == 0
        # 模型并行处理大小，默认为1。
        model_parallel_size = 1
        # 本地计算头数，等于总头数除以模型并行处理大小。
        self.n_local_heads = args.n_heads // model_parallel_size
        # 每个头的维度，等于模型维度除以头的总数。
        self.head_dim = args.dim // args.n_heads

        # Wq, Wk, Wv 参数矩阵，每个参数矩阵为 n_embd x n_embd
        # 这里通过三个组合矩阵来代替了n个参数矩阵的组合，其逻辑在于矩阵内积再拼接其实等同于拼接矩阵再内积，
        # 不理解的读者可以自行模拟一下，每一个线性层其实相当于n个参数矩阵的拼接
        self.wq = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        self.wk = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        self.wv = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        # 输出权重矩阵，维度为 n_embd x n_embd (head_dim = n_embeds / n_heads)
        self.wo = nn.Linear(args.n_heads * self.head_dim, args.dim, bias=False)
        # 注意力的 dropout
        self.attn_dropout = nn.Dropout(args.dropout)
        # 残差连接的 dropout
        self.resid_dropout = nn.Dropout(args.dropout)

        # 创建一个上三角矩阵，用于遮蔽未来信息
        # 注意，因为是多头注意力，Mask 矩阵比之前我们定义的多一个维度
        if is_causal:
            mask = torch.full((1, 1, args.max_seq_len, args.max_seq_len), float("-inf"))
            mask = torch.triu(mask, diagonal=1)
            # 注册为模型的缓冲区
            self.register_buffer("mask", mask)

    def forward(self, q: torch.Tensor, k: torch.Tensor, v: torch.Tensor):

        # 获取批次大小和序列长度，[batch_size, seq_len, dim]
        bsz, seqlen, _ = q.shape

        # 计算查询（Q）、键（K）、值（V）,输入通过参数矩阵层，维度为 (B, T, n_embed) x (n_embed, n_embed) -> (B, T, n_embed)
        xq, xk, xv = self.wq(q), self.wk(k), self.wv(v)

        # 将 Q、K、V 拆分成多头，维度为 (B, T, n_head, C // n_head)，然后交换维度，变成 (B, n_head, T, C // n_head)
        # 因为在注意力计算中我们是取了后两个维度参与计算
        # 为什么要先按B*T*n_head*C//n_head展开再互换1、2维度而不是直接按注意力输入展开，是因为view的展开方式是直接把输入全部排开，
        # 然后按要求构造，可以发现只有上述操作能够实现我们将每个头对应部分取出来的目标
        xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xq = xq.transpose(1, 2)
        xk = xk.transpose(1, 2)
```

```
        xv = xv.transpose(1, 2)


        # 注意力计算
        # 计算 QK^T / sqrt(d_k)，维度为 (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
        scores = torch.matmul(xq, xk.transpose(2, 3)) / math.sqrt(self.head_dim)
        # 掩码自注意力必须有注意力掩码
        if self.is_causal:
            assert hasattr(self, 'mask')
            # 这里截取到序列长度，因为有些序列可能比 max_seq_len 短
            scores = scores + self.mask[:, :, :seqlen, :seqlen]
        # 计算 softmax，维度为 (B, nh, T, T)
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        # 做 Dropout
        scores = self.attn_dropout(scores)
        # V * Score，维度为(B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        output = torch.matmul(scores, xv)

        # 恢复时间维度并合并头。
        # 将多头的结果拼接起来，先交换维度为 (B, T, n_head, C // n_head)，再拼接成 (B, T, n_head * C // n_head)
        # contiguous 函数用于重新开辟一块新内存存储，因为Pytorch设置先transpose再view会报错，
        # 因为view直接基于底层存储得到，然而transpose并不会改变底层存储，因此需要额外存储
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)

        # 最终投影回残差流。
        output = self.wo(output)
        output = self.resid_dropout(output)
        return output
```

2.2 Encoder-Decoder

在上一节，我们详细介绍了 Transformer 的核心——注意力机制。在《Attention is All You Need》一文中，作者通过仅使用注意力机制而抛弃传统的 RNN、CNN 架构搭建出 Transformer 模型，从而带来了 NLP 领域的大变革。在 Transformer 中，使用注意力机制的是其两个核心组件——Encoder（编码器）和 Decoder（解码器）。事实上，后续基于 Transformer 架构而来的预训练语言模型基本都是对 Encoder-Decoder 部分进行改进来构建新的模型架构，例如只使用 Encoder 的 BERT、只使用 Decoder 的 GPT 等。

在本节中，我们将以上一节所介绍的 注意力机制为基础，从 Transformer 所针对的 Seq2Seq 任务出发，解析 Transformer 的 Encoder-Decoder 结构。

2.2.1 Seq2Seq 模型

Seq2Seq，即序列到序列，是一种经典 NLP 任务。具体而言，是指模型输入的是一个自然语言序列 $input = (x_1, x_2, x_3 \dots x_n)$，输出的是一个可能不等长的自然语言序列 $output = (y_1, y_2, y_3 \dots y_m)$。事实上，Seq2Seq 是 NLP 最经典的任务，几乎所有的 NLP 任务都可以视为 Seq2Seq 任务。例如文本分类任务，可以视为输出长度为 1 的目标序列（如在上式中 $m = 1$）；词性标注任务，可以视为输出与输入序列等长的目标序列（如在上式中 $m = n$）。

机器翻译任务即是一个经典的 Seq2Seq 任务，例如，我们的输入可能是“今天天气真好”，输出是“Today is a good day.”。Transformer 是一个经典的 Seq2Seq 模型，即模型的输入为文本序列，输出为另一个文本序列。事实上，Transformer 一开始正是应用在机器翻译任务上的。

对于 Seq2Seq 任务，一般的思路是对自然语言序列进行编码再解码。所谓编码，就是将输入的自然语言序列通过隐藏层编码成能够表征语义的向量（或矩阵），可以简单理解为更复杂的词向量表示。而解码，就是对输入的自然语言序列编码得到的向量或矩阵通过隐藏层输出，再解码成对应的自然语言目标序列。通过编码再解码，就可以实现 Seq2Seq 任务。

Transformer 中的 Encoder，就是用于上述的编码过程；Decoder 则用于上述的解码过程。Transformer 结构，如图2.5所示：

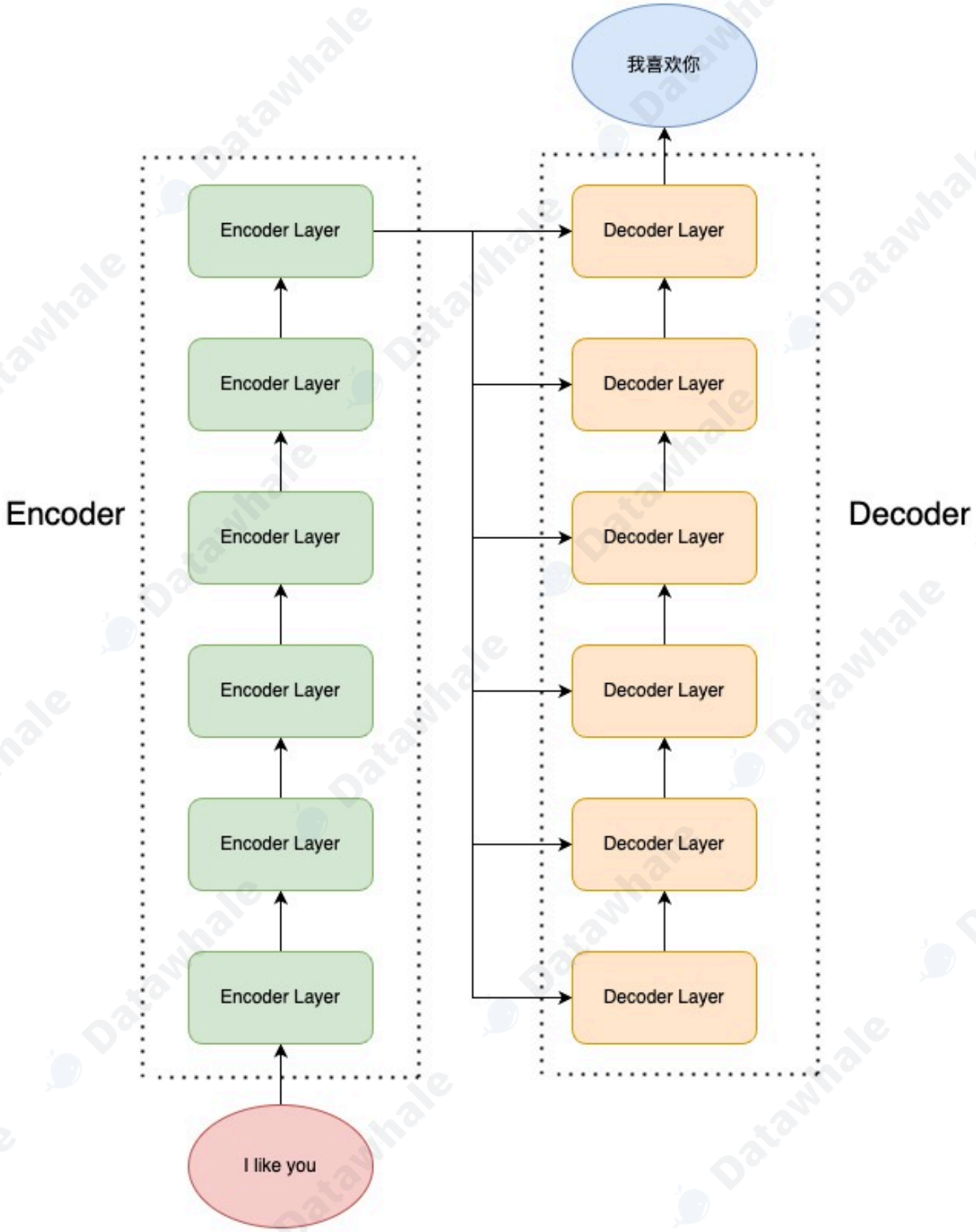


图2.5 编码器-解码器结构

Transformer 由 Encoder 和 Decoder 组成，每一个 Encoder（Decoder）又由 6个 Encoder（Decoder）Layer 组成。输入源序列会进入 Encoder 进行编码，到 Encoder Layer 的最顶层再将编码结果输出给 Decoder Layer 的每一层，通过 Decoder 解码后就可以得到输出目标序列了。

接下来，我们将首先介绍 Encoder 和 Decoder 内部传统神经网络的经典结构——前馈神经网络（FNN）、层归一化（Layer Norm）和残差连接（Residual Connection），然后进一步分析 Encoder 和 Decoder 的内部结构。

2.2.2 前馈神经网络

前馈神经网络（Feed Forward Neural Network，下简称 FFN），也就是我们在上一节提过的每一层的神经元都和上下两层的每一个神经元完全连接的网络结构。每一个 Encoder Layer 都包含一个上文讲的注意力机制和一个前馈神经网络。前馈神经网络的实现是较为简单的：

```
class MLP(nn.Module):
    '''前馈神经网络'''
    def __init__(self, dim: int, hidden_dim: int, dropout: float):
        super().__init__()
```

```
        # 定义第一层线性变换，从输入维度到隐藏维度
        self.w1 = nn.Linear(dim, hidden_dim, bias=False)
        # 定义第二层线性变换，从隐藏维度到输入维度
        self.w2 = nn.Linear(hidden_dim, dim, bias=False)
        # 定义dropout层，用于防止过拟合
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # 前向传播函数
        # 首先，输入x通过第一层线性变换和RELU激活函数
        # 然后，结果乘以输入x通过第三层线性变换的结果
        # 最后，通过第二层线性变换和dropout层
        return self.dropout(self.w2(F.relu(self.w1(x))))
```

注意，Transformer 的前馈神经网络是由两个线性层中间加一个 RELU 激活函数组成的，以及前馈神经网络还加入了一个 Dropout 层来防止过拟合。

2.2.3 层归一化

层归一化，也就是 Layer Norm，是深度学习中经典的归一化操作。神经网络主流的归一化一般有两种，批归一化（Batch Norm）和层归一化（Layer Norm）。

归一化核心是为了让不同层输入的取值范围或者分布能够比较一致。由于深度神经网络中每一层的输入都是上一层的输出，因此多层传递下，对网络中较高的层，之前的所有神经层的参数变化会导致其输入的分布发生较大的改变。也就是说，随着神经网络参数的更新，各层的输出分布是不相同的，且差异会随着网络深度的增大而增大。但是，需要预测的条件分布始终是相同的，从而也就造成了预测的误差。

因此，在深度神经网络中，往往需要归一化操作，将每一层的输入都归一化成标准正态分布。批归一化是指在一个 mini-batch 上进行归一化，相当于对一个 batch 对样本拆分出来一部分，首先计算样本的均值：

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} Z_j^i \tag{9}$$

其中，Z_j^i 是样本 i 在第 j 个维度上的值，m 就是 mini-batch 的大小。

再计算样本的方差：

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(Z_j^i - \mu_j)^2 \tag{10}$$

最后，对每个样本的值减去均值再除以标准差来将这一个 mini-batch 的样本的分布转化为标准正态分布：

$$\widetilde{Z_j} = \frac{Z_j - \mu_j}{\sqrt{\sigma^2 + \epsilon}} \tag{11}$$

此处加上 ϵ 这一极小量是为了避免分母为0。

但是，批归一化存在一些缺陷，例如：

- 当显存有限，mini-batch 较小时，Batch Norm 取的样本的均值和方差不能反映全局的统计分布信息，从而导致效果变差；
- 对于在时间维度展开的 RNN，不同句子的同一分布大概率不同，所以 Batch Norm 的归一化会失去意义；
- 在训练时，Batch Norm 需要保存每个 step 的统计信息（均值和方差）。在测试时，由于变长句子的特性，测试集可能出现比训练集更长的句子，所以对于后面位置的 step，是没有训练的统计量使用的；
- 应用 Batch Norm，每个 step 都需要去保存和计算 batch 统计量，耗时又耗力

因此，出现了在深度神经网络中更常用、效果更好的层归一化（Layer Norm）。相较于 Batch Norm 在每一层统计所有样本的均值和方差，Layer Norm 在每个样本上计算其所有层的均值和方差，从而使每个样本的分布达到稳定。Layer Norm 的归一化方式其实和 Batch Norm 是完全一样的，只是统计统计量的维度不同。

基于上述进行归一化的公式，我们可以简单地实现一个 Layer Norm 层：

```
class LayerNorm(nn.Module):
    ''' Layer Norm 层'''
    def __init__(self, features, eps=1e-6):
  super(LayerNorm, self).__init__()
    # 线性矩阵做映射
  self.a_2 = nn.Parameter(torch.ones(features))
  self.b_2 = nn.Parameter(torch.zeros(features))
  self.eps = eps

    def forward(self, x):
  # 在统计每个样本所有维度的值，求均值和方差
  mean = x.mean(-1, keepdim=True) # mean: [bsz, max_len, 1]
```

```
    std = x.std(-1, keepdim=True) # std: [bsz, max_len, 1]
      # 注意这里也在最后一个维度发生了广播
    return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

注意，在我们上文实现的 Layer Norm 层中，有两个线性矩阵进行映射。

2.2.4 残差连接

由于 Transformer 模型结构较复杂、层数较深，为了避免模型退化，Transformer 采用了残差连接的思想来连接每一个子层。残差连接，即下一层的输入不仅是上一层的输出，还包括上一层的输入。残差连接允许最底层信息直接传到最高层，让高层专注于残差的学习。

例如，在 Encoder 中，在第一个子层，输入进入多头自注意力层的同时会直接传递到该层的输出，然后该层的输出会与原输入相加，再进行标准化。在第二个子层也是一样。即：

$$x = x + MultiHeadSelfAttention(LayerNorm(x)) \tag{12}$$

$$output = x + FNN(LayerNorm(x)) \tag{13}$$

我们在代码实现中，通过在层的 forward 计算中加上原值来实现残差连接：

```
# 注意力计算
h = x + self.attention.forward(self.attention_norm(x))
# 经过前馈神经网络
out = h + self.feed_forward.forward(self.fnn_norm(h))
```

在上文代码中，self.attention_norm 和 self.fnn_norm 都是 LayerNorm 层，self.attn 是注意力层，而 self.feed_forward 是前馈神经网络。

2.2.5 Encoder

在实现上述组件之后，我们可以搭建起 Transformer 的 Encoder。Encoder 由 N 个 Encoder Layer 组成，每一个 Encoder Layer 包括一个注意力层和一个前馈神经网络。因此，我们可以首先实现一个 Encoder Layer：

```
class EncoderLayer(nn.Module):
  '''Encoder层'''
    def __init__(self, args):
        super().__init__()
        # 一个 Layer 中有两个 LayerNorm，分别在 Attention 之前和 MLP 之前
        self.attention_norm = LayerNorm(args.n_embd)
        # Encoder 不需要掩码，传入 is_causal=False
        self.attention = MultiHeadAttention(args, is_causal=False)
        self.fnn_norm = LayerNorm(args.n_embd)
        self.feed_forward = MLP(args)

    def forward(self, x):
        # Layer Norm
        x = self.attention_norm(x)
        # 自注意力
        h = x + self.attention.forward(x, x, x)
        # 经过前馈神经网络
        out = h + self.feed_forward.forward(self.fnn_norm(h))
        return out
```

然后我们搭建一个 Encoder，由 N 个 Encoder Layer 组成，在最后会加入一个 Layer Norm 实现规范化：

```
class Encoder(nn.Module):
    '''Encoder 块'''
    def __init__(self, args):
        super(Encoder, self).__init__()
        # 一个 Encoder 由 N 个 Encoder Layer 组成
        self.layers = nn.ModuleList([EncoderLayer(args) for _ in range(args.n_layer)])
        self.norm = LayerNorm(args.n_embd)

    def forward(self, x):
        "分别通过 N 层 Encoder Layer"
        for layer in self.layers:
            x = layer(x)
        return self.norm(x)
```

通过 Encoder 的输出，就是输入编码之后的结果。

2.2.6 Decoder

类似的，我们也可以先搭建 Decoder Layer，再将 N 个 Decoder Layer 组装为 Decoder。但是和 Encoder 不同的是，Decoder 由两个注意力层和一个前馈神经网络组成。第一个注意力层是一个掩码自注意力层，即使用 Mask 的注意力计算，保证每一个 token 只能使用该 token 之前的注意力分数；第二个注意力层是一个多头注意力层，该层将使用第一个注意力层的输出作为 query，使用 Encoder 的输出作为 key 和 value，来计算注意力分数。最后，再经过前馈神经网络：

```
class DecoderLayer(nn.Module):
  '''解码层'''
    def __init__(self, args):
        super().__init__()
        # 一个 Layer 中有三个 LayerNorm，分别在 Mask Attention 之前、Self Attention 之前和 MLP 之前
        self.attention_norm_1 = LayerNorm(args.n_embd)
        # Decoder 的第一个部分是 Mask Attention，传入 is_causal=True
        self.mask_attention = MultiHeadAttention(args, is_causal=True)
        self.attention_norm_2 = LayerNorm(args.n_embd)
        # Decoder 的第二个部分是 类似于 Encoder 的 Attention，传入 is_causal=False
        self.attention = MultiHeadAttention(args, is_causal=False)
        self.ffn_norm = LayerNorm(args.n_embd)
        # 第三个部分是 MLP
        self.feed_forward = MLP(args)

    def forward(self, x, enc_out):
        # Layer Norm
        x = self.attention_norm_1(x)
        # 掩码自注意力
        x = x + self.mask_attention.forward(x, x, x)
        # 多头注意力
        x = self.attention_norm_2(x)
        h = x + self.attention.forward(x, enc_out, enc_out)
        # 经过前馈神经网络
        out = h + self.feed_forward.forward(self.fnn_norm(h))
        return out
```

然后同样的，我们搭建一个 Decoder 块：

```
class Decoder(nn.Module):
    '''解码器'''
    def __init__(self, args):
        super(Decoder, self).__init__()
        # 一个 Decoder 由 N 个 Decoder Layer 组成
        self.layers = nn.ModuleList([DecoderLayer(args) for _ in range(args.n_layer)])
        self.norm = LayerNorm(args.n_embd)

    def forward(self, x, enc_out):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, enc_out)
        return self.norm(x)
```

完成上述 Encoder、Decoder 的搭建，就完成了 Transformer 的核心部分，接下来将 Encoder、Decoder 拼接起来再加入 Embedding 层就可以搭建出完整的 Transformer 模型啦。

2.3 搭建一个 Transformer

在前两章，我们分别深入剖析了 Attention 机制和 Transformer 的核心——Encoder、Decoder 结构，接下来，我们就可以基于上一章实现的组件，搭建起一个完整的 Transformer 模型。

2.3.1 Embeddng 层

正如我们在第一章所讲过的，在 NLP 任务中，我们往往需要将自然语言的输入转化为机器可以处理的向量。在深度学习中，承担这个任务的组件就是 Embedding 层。

Embedding 层其实是一个存储固定大小的词典的嵌入向量查找表。也就是说，在输入神经网络之前，我们往往会先让自然语言输入通过分词器 tokenizer，分词器的作用是把自然语言输入切分成 token 并转化成一个固定的 index。例如，如果我们将词表大小设为 4，输入“我喜欢你”，那么，分词器可以将输入转化成：

```
input：我
output：0

input：喜欢
output：1

input：你
output：2
```

当然，在实际情况下，tokenizer 的工作会比这更复杂。例如，分词有多种不同的方式，可以切分成词、切分成子词、切分成字符等，而词表大小则往往高达数万数十万。此处我们不赘述 tokenizer 的详细情况，在后文会详细介绍大模型的 tokenizer 是如何运行和训练的。

因此，Embedding 层的输入往往是一个形状为 （batch_size，seq_len，1）的矩阵，第一个维度是一次批处理的数量，第二个维度是自然语言序列的长度，第三个维度则是 token 经过 tokenizer 转化成的 index 值。例如，对上述输入，Embedding 层的输入会是：

```
[[0,1,2]]
```

其 batch_size 为1，seq_len 为3，转化出来的 index 如上。

而 Embedding 内部其实是一个可训练的（Vocab_size，embedding_dim）的权重矩阵，词表里的每一个值，都对应一行维度为 embedding_dim 的向量。对于输入的值，会对应到这个词向量，然后拼接成（batch_size，seq_len，embedding_dim）的矩阵输出。

上述实现并不复杂，我们可以直接使用 torch 中的 Embedding 层：

```
self.tok_embeddings = nn.Embedding(args.vocab_size, args.dim)
```

2.3.2 位置编码

注意力机制可以实现良好的并行计算，但同时，其注意力计算的方式也导致序列中相对位置的丢失。在 RNN、LSTM 中，输入序列会沿着语句本身的顺序被依次递归处理，因此输入序列的顺序提供了极其重要的信息，这也和自然语言的本身特性非常吻合。

但从上文对注意力机制的分析我们可以发现，在注意力机制的计算过程中，对于序列中的每一个 token，其他各个位置对其来说都是平等的，即“我喜欢你”和“你喜欢我”在注意力机制看来是完全相同的，但无疑这是注意力机制存在的一个巨大问题。因此，为使用序列顺序信息，保留序列中的相对位置信息，Transformer 采用了位置编码机制，该机制也在之后被多种模型沿用。

位置编码，即根据序列中 token 的相对位置对其进行编码，再将位置编码加入词向量编码中。位置编码的方式有很多，Transformer 使用了正余弦函数来进行位置编码（绝对位置编码Sinusoidal），其编码方式为：

$$
\begin{aligned}
PE(pos, 2i) &= sin(pos/10000^{2i/d_{model}}) \\
PE(pos, 2i+1) &= cos(pos/10000^{2i/d_{model}})
\end{aligned} \tag{14}
$$

上式中，pos 为 token 在句子中的位置，2i 和 2i+1 则是指示了 token 是奇数位置还是偶数位置，从上式中我们可以看出对于奇数位置的 token 和偶数位置的 token，Transformer 采用了不同的函数进行编码。

我们以一个简单的例子来说明位置编码的计算过程：假如我们输入的是一个长度为 4 的句子"I like to code"，我们可以得到下面的词向量矩阵$\mathbf{x}$，其中每一行代表的就是一个词向量，$\mathbf{x}_0 = [0.1, 0.2, 0.3, 0.4]$对应的就是“I”的词向量，它的pos就是为0，以此类推，第二行代表的是“like”的词向量，它的pos就是1：

$$
\mathbf{x} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.2 & 0.3 & 0.4 & 0.5 \\ 0.3 & 0.4 & 0.5 & 0.6 \\ 0.4 & 0.5 & 0.6 & 0.7 \end{bmatrix} \tag{15}
$$

则经过位置编码后的词向量为：

$$
\mathbf{x}_{\mathrm{PE}} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.2 & 0.3 & 0.4 & 0.5 \\ 0.3 & 0.4 & 0.5 & 0.6 \\ 0.4 & 0.5 & 0.6 & 0.7 \end{bmatrix} + \begin{bmatrix} \sin(\frac{0}{10000^0}) & \cos(\frac{0}{10000^0}) & \sin(\frac{0}{10000^{2/4}}) & \cos(\frac{0}{10000^{2/4}}) \\ \sin(\frac{1}{10000^0}) & \cos(\frac{1}{10000^0}) & \sin(\frac{1}{10000^{2/4}}) & \cos(\frac{1}{10000^{2/4}}) \\ \sin(\frac{2}{10000^0}) & \cos(\frac{2}{10000^0}) & \sin(\frac{2}{10000^{2/4}}) & \cos(\frac{2}{10000^{2/4}}) \\ \sin(\frac{3}{10000^0}) & \cos(\frac{3}{10000^0}) & \sin(\frac{3}{10000^{2/4}}) & \cos(\frac{3}{10000^{2/4}}) \end{bmatrix} = \begin{bmatrix} 0.1 & 1.2 & 0.3 & 1.4 \\ 1.041 & 0.84 & 0.41 & 1.49 \\ 1.209 & -0.016 & 0.52 & 1.59 \\ 0.541 & -0.489 & 0.895 & 1.655 \end{bmatrix}
$$

我们可以使用如下的代码来获取上述例子的位置编码：

```
import numpy as np
import matplotlib.pyplot as plt
def PositionEncoding(seq_len, d_model, n=10000):
    P = np.zeros((seq_len, d_model))
    for k in range(seq_len):
        for i in np.arange(int(d_model/2)):
            denominator = np.power(n, 2*i/d_model)
            P[k, 2*i] = np.sin(k/denominator)
            P[k, 2*i+1] = np.cos(k/denominator)
    return P

P = PositionEncoding(seq_len=4, d_model=4, n=100)
print(P)
```

```
[[ 0.          1.          0.          1.        ]
 [ 0.84147098  0.54030231  0.09983342  0.99500417]
 [ 0.90929743 -0.41614684  0.19866933  0.98006658]
 [ 0.14112001 -0.9899925   0.29552021  0.95533649]]
```

这样的位置编码主要有两个好处：

1. 使 PE 能够适应比训练集里面所有句子更长的句子，假设训练集里面最长的句子是有 20 个单词，突然来了一个长度为 21 的句子，则使用公式计算的方法可以计算出第 21 位的 Embedding。
2. 可以让模型容易地计算出相对位置，对于固定长度的间距 k，PE(pos+k) 可以用 PE(pos) 计算得到。因为 Sin(A+B) = Sin(A)Cos(B) + Cos(A)Sin(B), Cos(A+B) = Cos(A)Cos(B) - Sin(A)Sin(B)。

我们也可以通过严谨的数学推导证明该编码方式的优越性。原始的 Transformer Embedding 可以表示为：

$$f(\cdots,\boldsymbol{x}_m,\cdots,\boldsymbol{x}_n,\cdots)=f(\cdots,\boldsymbol{x}_n,\cdots,\boldsymbol{x}_m,\cdots) \tag{17}$$

很明显，这样的函数是不具有不对称性的，也就是无法表征相对位置信息。我们想要得到这样一种编码方式：

$$\tilde{f}(\cdots,\boldsymbol{x}_m,\cdots,\boldsymbol{x}_n,\cdots)=f(\cdots,\boldsymbol{x}_m+\boldsymbol{p}_m,\cdots,\boldsymbol{x}_n+\boldsymbol{p}_n,\cdots) \tag{18}$$

这里加上的 p_m，p_n 就是位置编码。接下来我们将 $f(\ldots,x_m+p_m,\ldots,x_n+p_n)$ 在 m,n 两个位置上做泰勒展开：

$$\tilde{f}\approx f+\boldsymbol{p}_m^\top\frac{\partial f}{\partial\boldsymbol{x}_m}+\boldsymbol{p}_n^\top\frac{\partial f}{\partial\boldsymbol{x}_n}+\frac{1}{2}\boldsymbol{p}_m^\top\frac{\partial^2 f}{\partial\boldsymbol{x}_m^2}\boldsymbol{p}_m+\frac{1}{2}\boldsymbol{p}_n^\top\frac{\partial^2 f}{\partial\boldsymbol{x}_n^2}\boldsymbol{p}_n+\underbrace{\boldsymbol{p}_m^\top\frac{\partial^2 f}{\partial\boldsymbol{x}_m\partial\boldsymbol{x}_n}\boldsymbol{p}_n}_{\boldsymbol{p}_m^\top\mathcal{H}\boldsymbol{p}_n} \tag{19}$$

可以看到第1项与位置无关，2～5项仅依赖单一位置，第6项（f 分别对 m、n 求偏导）与两个位置有关，所以我们希望第六项（$p_m^THp_n$）表达相对位置信息，即求一个函数 g 使得：

$$p_m^THp_n=g(m-n) \tag{20}$$

我们假设 H 是一个单位矩阵，则：

$$p_m^THp_n=p_m^Tp_n=\langle\boldsymbol{p}_m,\boldsymbol{p}_n\rangle=g(m-n) \tag{21}$$

通过将向量 [x,y] 视为复数 x+yi，基于复数的运算法则构建方程：

$$\langle\boldsymbol{p}_m,\boldsymbol{p}_n\rangle=\mathrm{Re}[\boldsymbol{p}_m\boldsymbol{p}_n^*] \tag{22}$$

再假设存在复数 q_{m-n} 使得：

$$\boldsymbol{p}_m\boldsymbol{p}_n^*=\boldsymbol{q}_{m-n} \tag{23}$$

使用复数的指数形式求解这个方程，得到二维情形下位置编码的解：

$$\boldsymbol{p}_m=e^{\mathrm{i}m\theta}\quad\Leftrightarrow\quad\boldsymbol{p}_m=\begin{pmatrix}\cos m\theta\\ \sin m\theta\end{pmatrix} \tag{24}$$

由于内积满足线性叠加性，所以更高维的偶数维位置编码，我们可以表示为多个二维位置编码的组合：

$$\boldsymbol{p}_m=\begin{pmatrix}e^{\mathrm{i}m\theta_0}\\ e^{\mathrm{i}m\theta_1}\\ \vdots\\ e^{\mathrm{i}m\theta_{d/2-1}}\end{pmatrix}\quad\Leftrightarrow\quad\boldsymbol{p}_m=\begin{pmatrix}\cos m\theta_0\\ \sin m\theta_0\\ \cos m\theta_1\\ \sin m\theta_1\\ \vdots\\ \cos m\theta_{d/2-1}\\ \sin m\theta_{d/2-1}\end{pmatrix} \tag{25}$$

再取 $\theta_i = 10000^{-2i/d}$（该形式可以使得随着|m−n|的增大，〈pm,pn〉有着趋于零的趋势，这一点可以通过对位置编码做积分来证明，而 base 取为 10000 是实验结果），就得到了上文的编码方式。

当 H 不是一个单位矩阵时，因为模型的 Embedding 层所形成的 d 维向量之间任意两个维度的相关性比较小，满足一定的解耦性，我们可以将其视作对角矩阵，那么使用上述编码：

$$\boldsymbol{p}_m^\top \boldsymbol{\mathcal{H}} \boldsymbol{p}_n = \sum_{i=1}^{d/2} \boldsymbol{\mathcal{H}}_{2i,2i} \cos m\theta_i \cos n\theta_i + \boldsymbol{\mathcal{H}}_{2i+1,2i+1} \sin m\theta_i \sin n\theta_i \tag{26}$$

通过积化和差：

$$\sum_{i=1}^{d/2} \frac{1}{2}(\boldsymbol{\mathcal{H}}_{2i,2i} + \boldsymbol{\mathcal{H}}_{2i+1,2i+1}) \cos(m-n)\theta_i + \frac{1}{2}(\boldsymbol{\mathcal{H}}_{2i,2i} - \boldsymbol{\mathcal{H}}_{2i+1,2i+1}) \cos(m+n)\theta_i \tag{27}$$

说明该编码仍然可以表示相对位置。

上述编码结果，如图2.6所示：

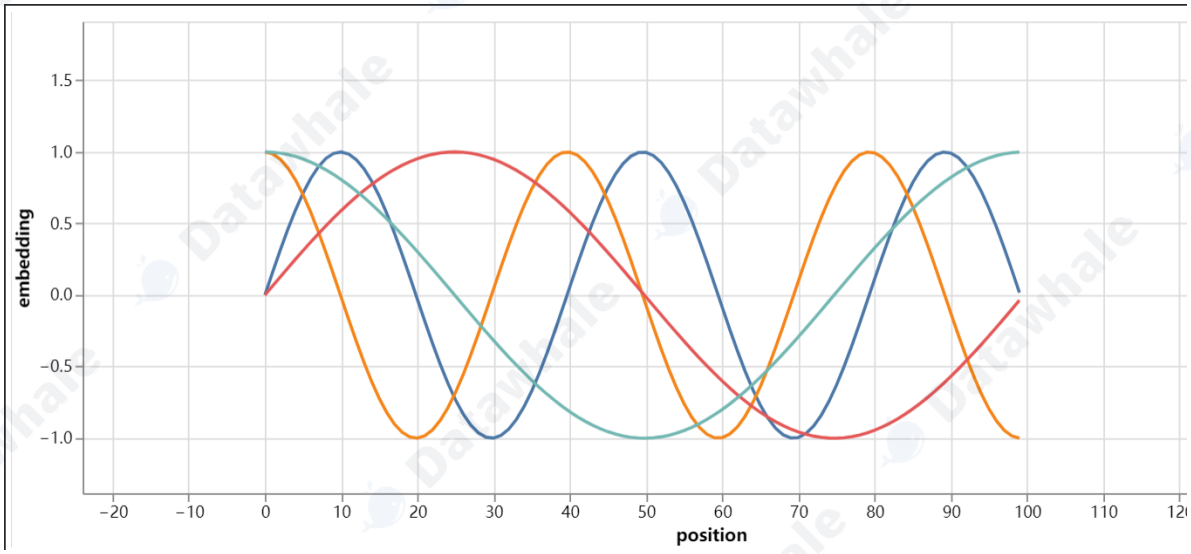


图2.6 编码结果

基于上述原理，我们实现一个位置编码层：

```
class PositionalEncoding(nn.Module):
    '''位置编码模块'''

    def __init__(self, args):
        super(PositionalEncoding, self).__init__()
        # Dropout 层
        self.dropout = nn.Dropout(p=args.dropout)

        # block size 是序列的最大长度
        pe = torch.zeros(args.block_size, args.n_embd)
        position = torch.arange(0, args.block_size).unsqueeze(1)
        # 计算 theta
        div_term = torch.exp(
            torch.arange(0, args.n_embd, 2) * -(math.log(10000.0) / args.n_embd)
        )
        # 分别计算 sin、cos 结果
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        # 将位置编码加到 Embedding 结果上
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)
```

2.3.3 一个完整的 Transformer

上述所有组件，再按照下图的 Tranfromer 结构拼接起来就是一个完整的 Transformer 模型了，如图2.7所示：

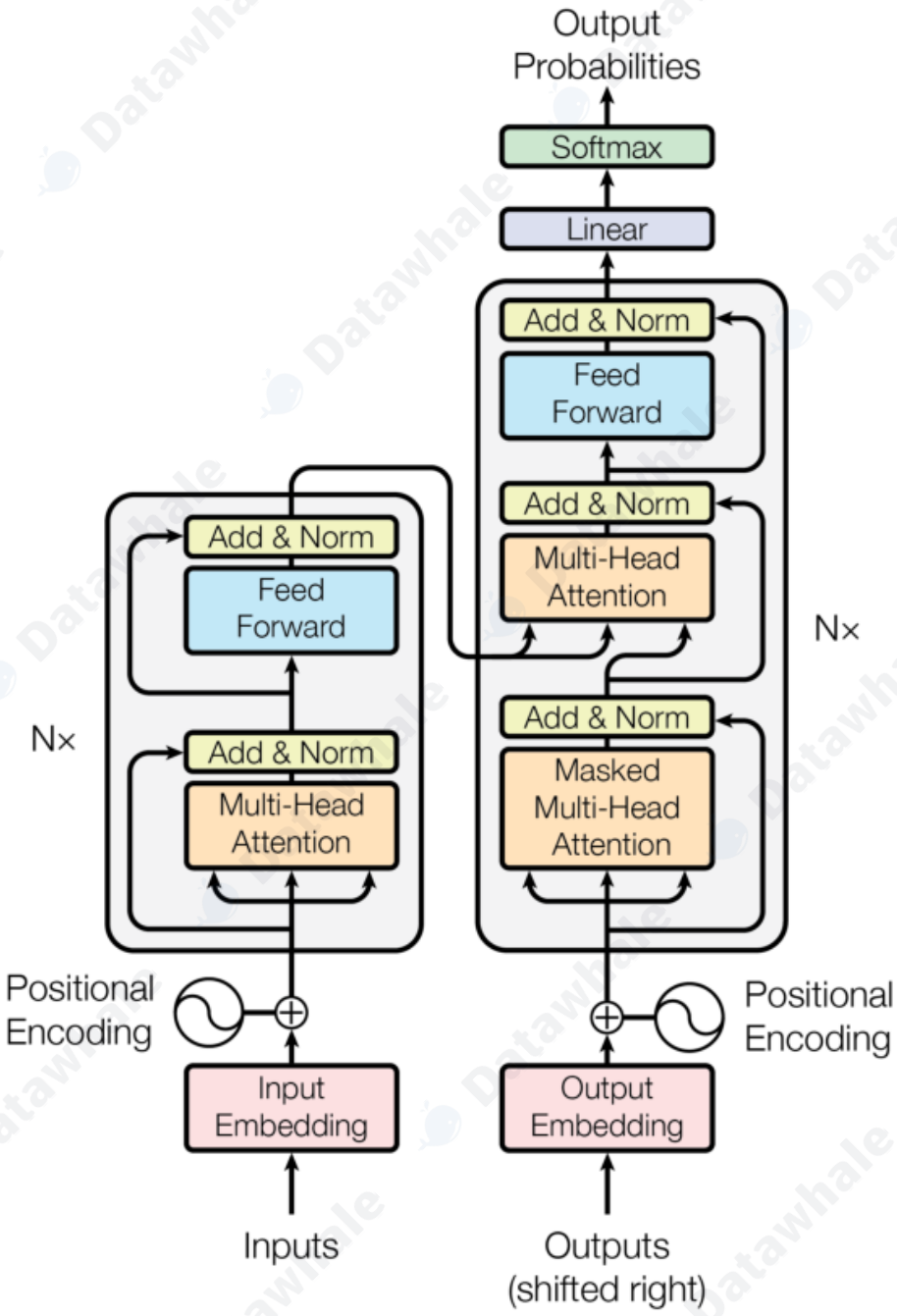


图2.7 Transformer 模型结构

如图，经过 tokenizer 映射后的输出先经过 Embedding 层和 Positional Embedding 层编码，然后进入上一节讲过的 N 个 Encoder 和 N 个 Decoder（在 Transformer 原模型中，N 取为6），最后经过一个线性层和一个 Softmax 层就得到了最终输出。

基于之前所实现过的组件，我们实现完整的 Transformer 模型：

```
class Transformer(nn.Module):
    '''整体模型'''
    def __init__(self, args):
        super().__init__()
        # 必须输入词表大小和 block size
        assert args.vocab_size is not None
        assert args.block_size is not None
        self.args = args
        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(args.vocab_size, args.n_embd),
            wpe = PositionalEncoding(args),
            drop = nn.Dropout(args.dropout),
```

```
            encoder = Encoder(args),
            decoder = Decoder(args),
        ))
        # 最后的线性层，输入是 n_embd，输出是词表大小
        self.lm_head = nn.Linear(args.n_embd, args.vocab_size, bias=False)

        # 初始化所有的权重
        self.apply(self._init_weights)

        # 查看所有参数的数量
        print("number of parameters: %.2fM" % (self.get_num_params()/1e6,))

    '''统计所有参数的数量'''
    def get_num_params(self, non_embedding=False):
        # non_embedding: 是否统计 embedding 的参数
        n_params = sum(p.numel() for p in self.parameters())
        # 如果不统计 embedding 的参数，就减去
        if non_embedding:
            n_params -= self.transformer.wpe.weight.numel()
        return n_params

    '''初始化权重'''
    def _init_weights(self, module):
        # 线性层和 Embedding 层初始化为正则分布
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    '''前向计算函数'''
    def forward(self, idx, targets=None):
        # 输入为 idx，维度为 (batch size, sequence length, 1)；targets 为目标序列，用于计算 loss
        device = idx.device
        b, t = idx.size()
        assert t <= self.args.block_size, f"不能计算该序列，该序列长度为 {t}，最大序列长度只有 {self.args.block_size}"

        # 通过 self.transformer
        # 首先将输入 idx 通过 Embedding 层，得到维度为 (batch size, sequence length, n_embd)
        print("idx",idx.size())
        # 通过 Embedding 层
        tok_emb = self.transformer.wte(idx)
        print("tok_emb",tok_emb.size())
        # 然后通过位置编码
        pos_emb = self.transformer.wpe(tok_emb)
        # 再进行 Dropout
        x = self.transformer.drop(pos_emb)
        # 然后通过 Encoder
        print("x after wpe:",x.size())
        enc_out = self.transformer.encoder(x)
        print("enc_out:",enc_out.size())
        # 再通过 Decoder
        x = self.transformer.decoder(x, enc_out)
        print("x after decoder:",x.size())

        if targets is not None:
            # 训练阶段，如果我们给了 targets，就计算 loss
            # 先通过最后的 Linear 层，得到维度为 (batch size, sequence length, vocab size)
            logits = self.lm_head(x)
            # 再跟 targets 计算交叉熵
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1)
        else:
            # 推理阶段，我们只需要 logits，loss 为 None
            # 取 -1 是只取序列中的最后一个作为输出
            logits = self.lm_head(x[:, [-1], :]) # note: using list [-1] to preserve the time dim
            loss = None

        return logits, loss
```

注意，上述代码除去搭建了整个 Transformer 结构外，我们还额外实现了三个函数：

- get_num_params：用于统计模型的参数量
- _init_weights：用于对模型所有参数进行随机初始化
- forward：前向计算函数

另外，在前向计算函数中，我们对模型使用 pytorch 的交叉熵函数来计算损失，对于不同的损失函数，读者可以查阅 Pytorch 的官方文档，此处就不再赘述了。

经过上述步骤，我们就可以从零“手搓”一个完整的、可计算的 Transformer 模型。限于本书主要聚焦在 LLM，在本章，我们就不再详细讲述如何训练 Transformer 模型了；在后文中，我们将类似地从零“手搓”一个 LLaMA 模型，并手把手带大家训练一个属于自己的 Tiny LLaMA。

参考文献

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. (2023). *Attention Is All You Need.* arXiv preprint arXiv:1706.03762.

[2] Jay Mody 的文章 “An Intuition for Attention”. 来源：https://jaykmody.com/blog/attention-intuition/

第三章 预训练语言模型

3.1 Encoder-only PLM

在上一章，我们详细讲解了给 NLP 领域带来巨大变革注意力机制以及使用注意力机制搭建的模型 Transformer，NLP 模型的里程碑式转变也就自此而始。在上文对 Transformer 的讲解中我们可以看到，Transformer 结构主要由 Encoder、Decoder 两个部分组成，两个部分分别具有不一样的结构和输入输出。

针对 Encoder、Decoder 的特点，引入 ELMo 的预训练思路，开始出现不同的、对 Transformer 进行优化的思路。例如，Google 仅选择了 Encoder 层，通过将 Encoder 层进行堆叠，再提出不同的预训练任务-掩码语言模型（Masked Language Model，MLM），打造了一统自然语言理解（Natural Language Understanding，NLU）任务的代表模型——BERT。而 OpenAI 则选择了 Decoder 层，使用原有的语言模型（Language Model，LM）任务，通过不断增加模型参数和预训练语料，打造了在 NLG（Natural Language Generation，自然语言生成）任务上优势明显的 GPT 系列模型，也是现今大火的 LLM 的基座模型。当然，还有一种思路是同时保留 Encoder 与 Decoder，打造预训练的 Transformer 模型，例如由 Google 发布的 T5模型。

在本章中，我们将以 Encoder-Only、Encoder-Decoder、Decoder-Only 的顺序来依次介绍 Transformer 时代的各个主流预训练模型，分别介绍三种核心的模型架构、每种主流模型选择的预训练任务及其独特优势，这也是目前所有主流 LLM 的模型基础。

3.1.1 BERT

BERT，全名为 Bidirectional Encoder Representations from Transformers，是由 Google 团队在 2018年发布的预训练语言模型。该模型发布于论文《BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding》，实现了包括 GLUE、MultiNLI 等七个自然语言处理评测任务的最优性能（State Of The Art，SOTA），堪称里程碑式的成果。自 BERT 推出以来，预训练+微调的模式开始成为自然语言处理任务的主流，不仅 BERT 自身在不断更新迭代提升模型性能，也出现了如 MacBERT、BART 等基于 BERT 进行优化提升的模型。可以说，BERT 是自然语言处理的一个阶段性成果，标志着各种自然语言处理任务的重大进展以及预训练模型的统治地位建立，一直到 LLM 的诞生，NLP 领域的主导地位才从 BERT 系模型进行迁移。即使在 LLM 时代，要深入理解 LLM 与 NLP，BERT 也是无法绕过的一环。

（1）思想沿承

BERT 是一个统一了多种思想的预训练模型。其所沿承的核心思想包括：

- Transformer 架构。正如我们在上一章所介绍的，在 2017年发表的《Attention is All You Need》论文提出了完全使用 注意力机制而抛弃 RNN、LSTM 结构的 Transformer 模型，带来了新的模型架构。BERT 正沿承了 Transformer 的思想，在 Transformer 的模型基座上进行优化，通过将 Encoder 结构进行堆叠，扩大模型参数，打造了在 NLU 任务上独居天分的模型架构；
- 预训练+微调范式。同样在 2018年，ELMo 的诞生标志着预训练+微调范式的诞生。ELMo 模型基于双向 LSTM 架构，在训练数据上基于语言模型进行预训练，再针对下游任务进行微调，表现出了更加优越的性能，将 NLP 领域导向预训练+微调的研究思路。而 BERT 也采用了该范式，并通过将模型架构调整为 Transformer，引入更适合文本理解、能捕捉深层双向语义关系的预训练任务 MLM，将预训练-微调范式推向了高潮。

接下来，我们将从模型架构、预训练任务以及下游任务微调三个方面深入剖析 BERT，分析 BERT 的核心思路及优势，帮助大家理解 BERT 为何能够具备远超之前模型的性能，也从而更加深刻地理解 LLM 如何能够战胜 BERT 揭开新时代的大幕。

（2）模型架构——Encoder Only

BERT 的模型架构是取了 Transformer 的 Encoder 部分堆叠而成，其主要结构如图3.1所示：

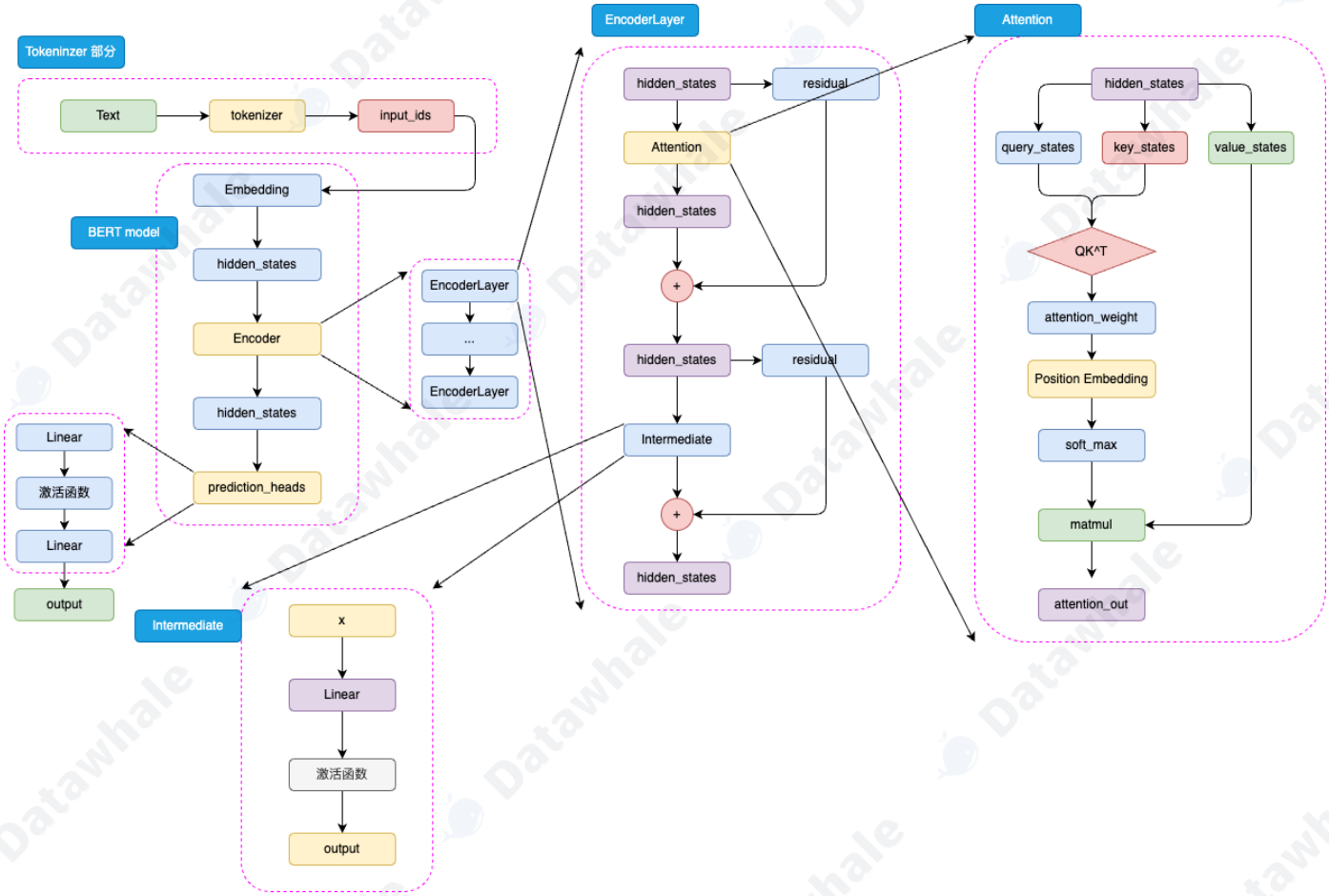


图3.1 BERT 模型结构

BERT 是针对于 NLU 任务打造的预训练模型，其输入一般是文本序列，而输出一般是 Label，例如情感分类的积极、消极 Label。但是，正如 Transformer 是一个 Seq2Seq 模型，使用 Encoder 堆叠而成的 BERT 本质上也是一个 Seq2Seq 模型，只是没有加入对特定任务的 Decoder，因此，为适配各种 NLU 任务，在模型的最顶层加入了一个分类头 prediction_heads，用于将多维度的隐藏状态通过线性层转换到分类维度（例如，如果一共有两个类别，prediction_heads 输出的就是两维向量）。

模型整体既是由 Embedding、Encoder 加上 prediction_heads 组成：

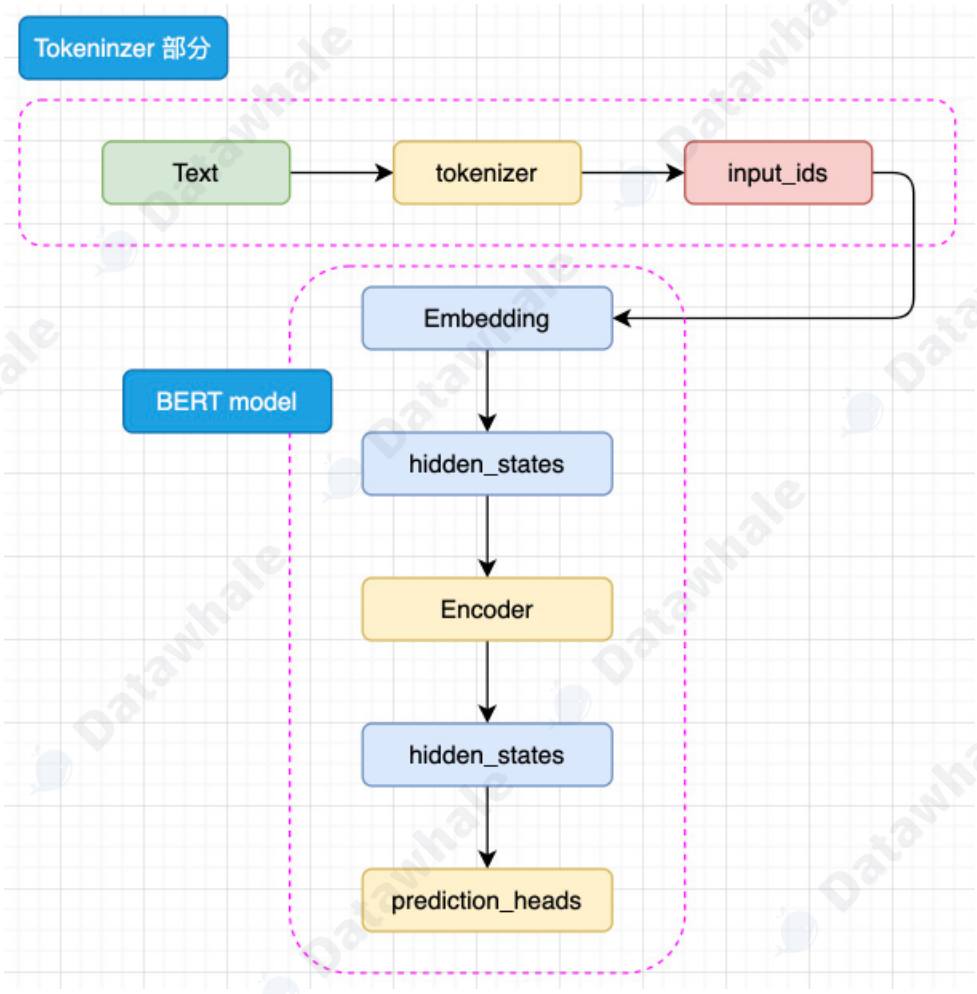


图3.2 BERT 模型简略结构

输入的文本序列会首先通过 tokenizer（分词器） 转化成 input_ids（基本每一个模型在 tokenizer 的操作都类似，可以参考 Transformer 的 tokenizer 机制，后文不再赘述），然后进入 Embedding 层转化为特定维度的 hidden_states，再经过 Encoder 块。Encoder 块中是对叠起来的 N 层 Encoder Layer，BERT 有两种规模的模型，分别是 base 版本（12层 Encoder Layer，768 的隐藏层维度，总参数量 110M），large 版本（24层 Encoder Layer，1024 的隐藏层维度，总参数量 340M）。通过Encoder 编码之后的最顶层 hidden_states 最后经过 prediction_heads 就得到了最后的类别概率，经过 Softmax 计算就可以计算出模型预测的类别。

prediction_heads 其实就是线性层加上激活函数，一般而言，最后一个线性层的输出维度和任务的类别数相等，如图3.3所示：

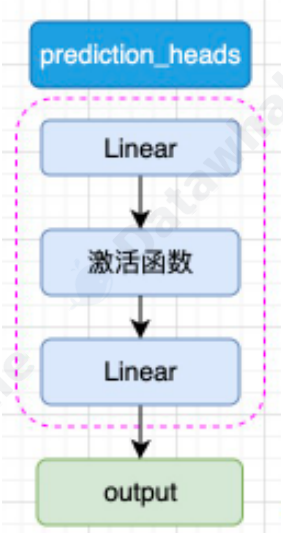


图3.3 prediction_heads 结构

而每一层 Encoder Layer 都是和 Transformer 中的 Encoder Layer 结构类似的层，如图3.4所示：

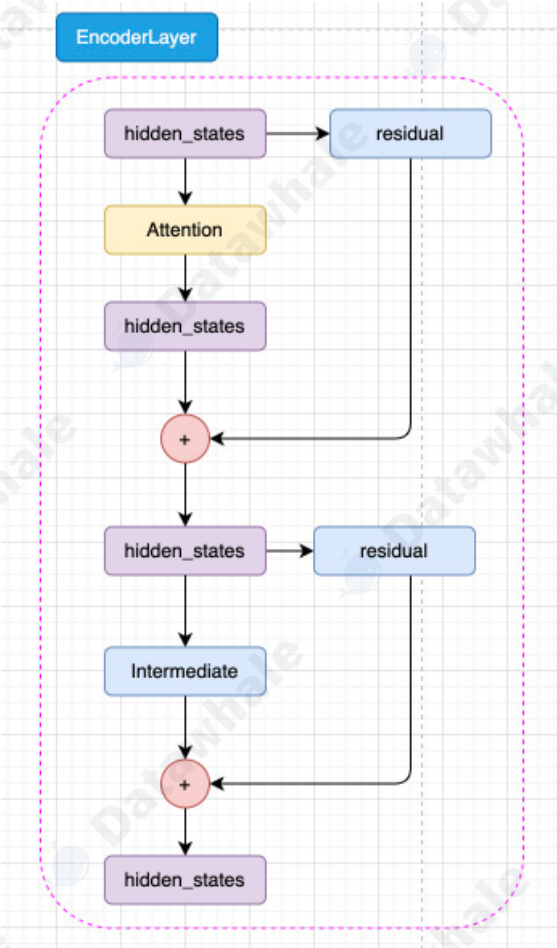


图3.4 Encoder Layer 结构

如图3.5所示，已经通过 Embedding 层映射的 hidden_states 进入核心的 attention 机制，然后通过残差连接的机制和原输入相加，再经过一层 Intermediate 层得到最终输出。Intermediate 层是 BERT 的特殊称呼，其实就是一个线性层加上激活函数：

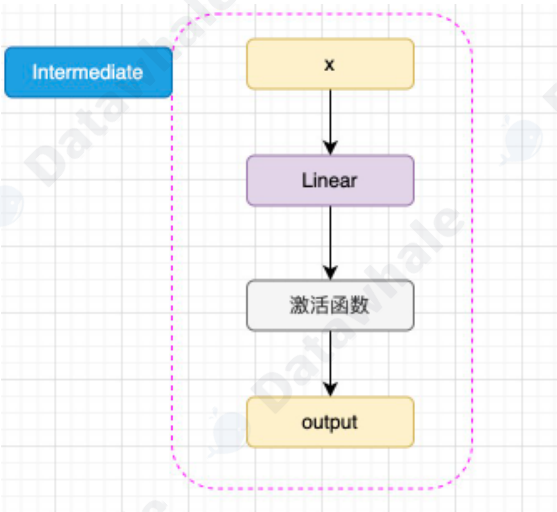


图3.5 Intermediate 结构

注意，BERT 所使用的激活函数是 GELU 函数，全名为高斯误差线性单元激活函数，这也是自 BERT 才开始被普遍关注的激活函数。GELU 的计算方式为：

$$GELU(x) = 0.5x(1 + tanh(\sqrt{\frac{2}{\pi}})(x + 0.044715x^3))$$

GELU 的核心思路为将随机正则的思想引入激活函数，通过输入自身的概率分布，来决定抛弃还是保留自身的神经元。关于 GELU 的原理与核心思路，此处不再赘述，有兴趣的读者可以自行学习。

BERT 的 注意力机制和 Transformer 中 Encoder 的 自注意力机制几乎完全一致，但是 BERT 将相对位置编码融合在了注意力机制中，将相对位置编码同样视为可训练的权重参数，如图3.6所示：

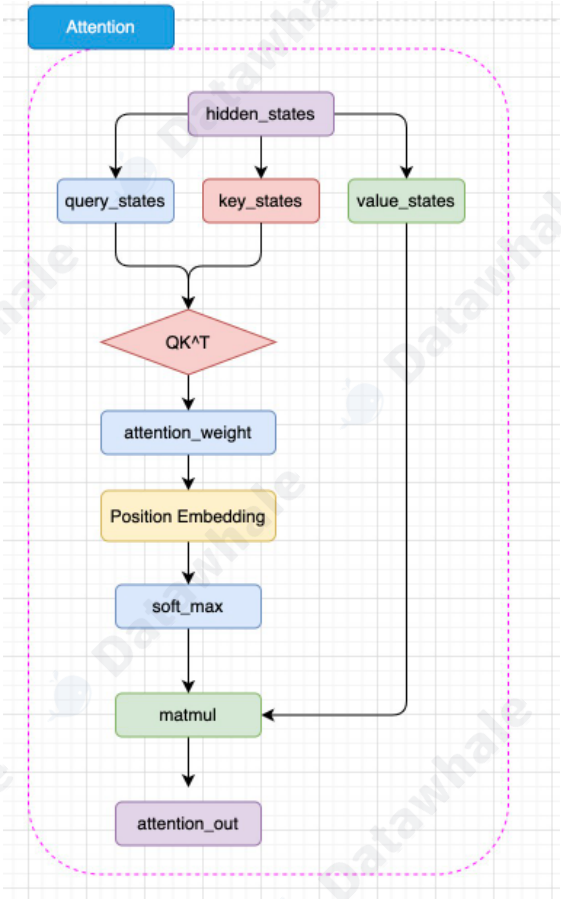


图3.6 BERT 注意力机制结构

如图，BERT 的注意力计算过程和 Transformer 的唯一差异在于，在完成注意力分数的计算之后，先通过 Position Embedding 层来融入相对位置信息。这里的 Position Embedding 层，其实就是一层线性矩阵。通过可训练的参数来拟合相对位置，相对而言比 Transformer 使用的绝对位置编码 Sinusoidal 能够拟合更丰富的相对位置信息，但是，这样也增加了不少模型参数，同时完全无法处理超过模型训练长度的输入（例如，对 BERT 而言能处理的最大上下文长度是 512 个 token）。

可以看出，BERT 的模型架构既是建立在 Transformer 的 Encoder 之上的，这也是为什么说 BERT 沿承了 Transformer 的思想。

（3）预训练任务——MLM + NSP

相较于基本沿承 Transformer 的模型架构，BERT 更大的创新点在于其提出的两个新的预训练任务上——MLM 和 NSP（Next Sentence Prediction，下一句预测）。预训练-微调范式的核心优势在于，通过将预训练和微调分离，完成一次预训练的模型可以仅通过微调应用在几乎所有下游任务上，只要微调的成本较低，即使预训练成本是之前的数倍甚至数十倍，模型仍然有更大的应用价值。因此，可以进一步扩大模型参数和预训练数据量，使用海量的预训练语料来让模型拟合潜在语义与底层知识，从而让模型通过长时间、大规模的预训练获得强大的语言理解和生成能力。

因此，预训练数据的核心要求即是需要极大的数据规模（数亿 token）。毫无疑问，通过人工标注产出的全监督数据很难达到这个规模。因此，预训练数据一定是从无监督的语料中获取。这也是为什么传统的预训练任务都是 LM 的原因——LM 使用上文预测下文的方式可以直接应用到任何文本中，对于任意文本，我们只需要将下文遮蔽将上文输入模型要求其预测就可以实现 LM 训练，因此互联网上所有文本语料都可以被用于预训练。

但是，LM 预训练任务的一大缺陷在于，其直接拟合从左到右的语义关系，但忽略了双向的语义关系。虽然 Transformer 中通过位置编码表征了文本序列中的位置信息，但这和直接拟合双向语义关系还是有本质区别。例如，BiLSTM（双向 LSTM 模型）在语义表征上就往往优于 LSTM 模型，就是因为 BiLSTM 通过双向的 LSTM 拟合了双向语义关系。因此，有没有一种预训练任务，能够既利用海量无监督语料，又能够训练模型拟合双向语义关系的能力?

基于这一思想，Jacob 等学者提出了 MLM，也就是掩码语言模型作为新的预训练任务。相较于模拟人类写作的 LM，MLM 模拟的是“完形填空”。MLM 的思路也很简单，在一个文本序列中随机遮蔽部分 token，然后将所有未被遮蔽的 token 输入模型，要求模型根据输入预测被遮蔽的 token。例如，输入和输出可以是：

```
输入：I <MASK> you because you are <MASK>
输出：<MASK> = love; <MASK> = wonderful
```

由于模型可以利用被遮蔽的 token 的上文和下文一起理解语义来预测被遮蔽的 token，因此通过这样的任务，模型可以拟合双向语义，也就能够更好地实现文本的理解。同样，MLM 任务无需对文本进行任何人为的标注，只需要对文本进行随机遮蔽即可，因此也可以利用互联网所有文本语料实现预训练。例如，BERT 的预训练就使用了足足 3300M 单词的语料。

不过，MLM 也存在其固有缺陷。LM 任务模拟了人自然创作的过程，其训练和下游任务是完全一致的，也就是说，训练时是根据上文预测下文，下游任务微调和推理时也同样如此。但是 MLM 不同，在下游任务微调和推理时，其实是不存在我们人工加入的 `<MASK>` 的，我们会直接通过原文本得到对应的隐藏状态再根据下游任务进入分类器或其他组件。预训练和微调的不一致，会极大程度影响模型在下游任务微调的性能。针对这一问题，作者对 MLM 的策略进行了改进。

在具体进行 MLM 训练时，会随机选择训练语料中 15% 的 token 用于遮蔽。但是这 15% 的 token 并非全部被遮蔽为 `<MASK>`，而是有 80% 的概率被遮蔽，10% 的概率被替换为任意一个 token，还有 10% 的概率保持不变。其中 10% 保持不变就是为了消除预训练和微调的不一致，而 10% 的随机替换核心意义在于迫使模型保持对上下文信息的学习。因为如果全部遮蔽的话，模型仅需要处理被遮蔽的位置，从而仅学习要预测的 token 而丢失了对上下文的学习。通过引入部分随机 token，模型无法确定需要预测的 token，从而被迫保持每一个 token 的上下文表征分布，从而具备了对句子的特征表示能力。且由于随机 token 的概率很低，其并不会影响模型实质的语言理解能力。

除去 MLM，BERT 还提出了另外一个预训练任务——NSP，即下一个句子预测。NSP 的核心思想是针对句级的 NLU 任务，例如问答匹配、自然语言推理等。问答匹配是指，输入一个问题和若干个回答，要求模型找出问题的真正回答；自然语言推理是指，输入一个前提和一个推理，判断推理是否是符合前提的。这样的任务都需要模型在句级去拟合关系，判断两个句子之间的关系，而不仅是 MLM 在 token 级拟合的语义关系。因此，BERT 提出了 NSP 任务来训练模型在句级的语义关系拟合。

NSP 任务的核心思路是要求模型判断一个句对的两个句子是否是连续的上下文。例如，输入和输入可以是：

```
输入：
    Sentence A: I love you.
    Sentence B: Because you are wonderful.
输出：
    1（是连续上下文）

输入：
    Sentence A: I love you.
    Sentence B: Because today's dinner is so nice.
输出：
    0（不是连续上下文）
```

通过要求模型判断句对关系，从而迫使模型拟合句子之间的关系，来适配句级的 NLU 任务。同样，由于 NSP 的正样本可以从无监督语料中随机抽取任意连续的句子，而负样本可以对句子打乱后随机抽取（只需要保证不要抽取到原本就连续的句子就行），因此也可以具有几乎无限量的训练数据。

在具体预训练时，BERT 使用了 800M 的 BooksCorpus 语料和 2500M 的英文维基百科语料，90% 的数据使用 128 的上下文长度训练，剩余 10% 的数据使用 512 作为上下文长度进行预训练，总共约训练了 3.3B token。其训练的超参数也是值得关注的，BERT 的训练语料共有 13GB 大小，其在 256 的 batch size 上训练了 1M 步（40 个 Epoch）。而相较而言，LLM 一般都只会训练一个 Epoch，且使用远大于 256 的 batch size。

可以看到，相比于传统的非预训练模型，其训练的数据量有指数级增长。当然，更海量的训练数据需要更大成本的算力，BERT 的 Base 版本和 Large 版本分别使用了 16块 TPU 和 64块 TPU 训练了 4天才完成。

（4）下游任务微调

作为 NLP 领域里程碑式的成果，BERT 的一个重大意义就是正式确立了预训练-微调的两阶段思想，即在海量无监督语料上进行预训练来获得通用的文本理解与生成能力，再在对应的下游任务上进行微调。该种思想的一个重点在于，预训练得到的强大能力能否通过低成本的微调快速迁移到对应的下游任务上。

针对这一点，BERT 设计了更通用的输入和输出层来适配多任务下的迁移学习。对每一个输入的文本序列，BERT 会在其首部加入一个特殊 token `<CLS>`。在后续编码中，该 token 代表的即是整句的状态，也就是句级的语义表征。在进行 NSP 预训练时，就使用了该 token 对应的特征向量来作为最后分类器的输入。

在完成预训练后，针对每一个下游任务，只需要使用一定量的全监督人工标注数据，对预训练的 BERT 在该任务上进行微调即可。所谓微调，其实和训练时更新模型参数的策略一致，只不过在特定的任务、更少的训练数据、更小的 batch_size 上进行训练，更新参数的幅度更小。对于绝大部分下游任务，都可以直接使用 BERT 的输出。例如，对于文本分类任务，可以直接修改模型结构中的 prediction_heads 最后的分类头即可。对于序列标注等任务，可以集成 BERT 多层的隐含层向量再输出最后的标注结果。对于文本生成任务，也同样可以取 Encoder 的输出直接解码得到最终生成结果。因此，BERT 可以非常高效地应用于多种 NLP 任务。

BERT 一经提出，直接在 NLP 11个赛道上取得 SOTA 效果，成为 NLU 方向上当之无愧的霸主，后续若干在 NLU 任务上取得更好效果的模型都是在 BERT 基础上改进得到的。直至 LLM 时代，BERT 也仍然能在很多标注数据丰富的 NLU 任务上达到最优效果，事实上，对于某些特定、训练数据丰富且强调高吞吐的任务，BERT 比 LLM 更具有可用性。

3.1.2 RoBERTa

BERT 作为 NLP 划时代的杰作，同时在多个榜单上取得 SOTA 效果，也带动整个 NLP 领域向预训练模型方向迁移。以 BERT 为基础，在多个方向上进行优化，还涌现了一大批效果优异的 Encoder-Only 预训练模型。它们大都有和 BERT 类似或完全一致的模型结构，在训练数据、预训练任务、训练参数等方面上进行了优化，以取得能力更强大、在下游任务上表现更亮眼的预训练模型。其中之一即是同样由 Facebook 发布的 RoBERTa。

前面我们说过，预训练-微调的一个核心优势在于可以使用远大于之前训练数据的海量无监督语料进行预训练。因为在传统的深度学习范式中，对每一个任务，我们需要从零训练一个模型，那么就无法使用太大的模型参数，否则需要极大规模的有监督数据才能让模型较好地拟合，成本太大。但在预训练-微调范式，我们在预训练阶段可以使用尽可能大量的训练数据，只需要一次预训练好的模型，后续在每一个下游任务上通过少量有监督数据微调即可。而 BERT 就使用了 13GB（3.3B token）的数据进行预训练，这相较于传统 NLP 来说是一个极其巨大的数据规模了。

但是，13GB 的预训练数据是否让 BERT 达到了充分的拟合呢？如果我们使用更多预训练语料，是否可以进一步增强模型性能？更多的，BERT 所选用的预训练任务、训练超参数是否是最优的？RoBERTa 应运而生。

（1）优化一：去掉 NSP 预训练任务

RoBERTa 的模型架构与 BERT 完全一致，也就是使用了 BERT-large（24层 Encoder Layer，1024 的隐藏层维度，总参数量 340M）的模型参数。在预训练任务上，有学者质疑 NSP 任务并不能提高模型性能，因为其太过简单，加入到预训练中并不能使下游任务微调时明显受益，甚至会带来负面效果。RoBERTa 设置了四个实验组：

```
1. 段落构建的 MLM + NSP：BERT 原始预训练任务，输入是一对片段，每个片段包括多个句子，来构造 NSP 任务；
2. 文档对构建的 MLM + NSP：一个输入构建一对句子，通过增大 batch 来和原始输入达到 token 等同；
3. 跨越文档的 MLM：去掉 NSP 任务，一个输入为从一个或多个文档中连续采样的完整句子，为使输入达到最大长度（512），可能一个输入会包括多个文档；
4. 单文档的 MLM：去掉 NSP 任务，且限制一个输入只能从一个文档中采样，同样通过增大 batch 来和原始输入达到 token 等同
```

实验结果证明，后两组显著优于前两组，且单文档的 MLM 组在下游任务上微调时性能最佳。因此，RoBERTa 在预训练中去掉了 NSP，只使用 MLM 任务。

同时，RoBERTa 对 MLM 任务本身也做出了改进。在 BERT 中，Mask 的操作是在数据处理的阶段完成的，因此后期预训练时同一个 sample 待预测的 `<MASK>` 总是一致的。由于 BERT 共训练了 40 个 Epoch，为使模型的训练数据更加广泛，BERT 将数据进行了四次随机 Mask，也就是每 10个 Epoch 模型训练的数据是完全一致的。而 RoBERTa 将 Mask 操作放到了训练阶段，也就是动态遮蔽策略，从而让每一个 Epoch 的训练数据 Mask 的位置都不一致。在实验中，动态遮蔽仅有很微弱的优势优于静态遮蔽，但由于动态遮蔽更高效、易于实现，后续 MLM 任务基本都使用了动态遮蔽。

（2）优化二：更大规模的预训练数据和预训练步长

RoBERTa 使用了更大量的无监督语料进行预训练，除去 BERT 所使用的 BookCorpus 和英文维基百科外，还使用了 CC-NEWS（CommonCrawl 数据集新闻领域的英文部分）、OPENWEBTEXT（英文网页）、STORIES（CommonCrawl 数据集故事风格子集），共计 160GB 的数据，十倍于 BERT。

同时，RoBERTa 认为更大的 batch size 既可以提高优化速度，也可以提高任务结束性能。因此，实验在 8K 的 batch size（对比 BERT 的 batch size 为 256）下训练 31K Step，也就是总训练 token 数和 BERT 一样是 3.3B 时，模型性能更好，从而证明了大 batch size 的意义。在此基础上，RoBERTa 一共训练了 500K Step（约合 66个 Epoch）。同时，RoBERTa 不再采用 BERT 在 256 长度上进行大部分训练再在 512 长度上完成训练的策略，而是全部在 512 长度上进行训练。

当然，更大的预训练数据、更长的序列长度和更多的训练 Epoch，需要预训练阶段更多的算力资源。训练一个 RoBERTa，Meta 使用了 1024 块 V100（32GB 显存）训练了一天。

（3）优化三：更大的 bpe 词表

RoBERTa、BERT 和 Transformer 一样，都使用了 BPE 作为 Tokenizer 的编码策略。BPE，即 Byte Pair Encoding，字节对编码，是指以子词对作为分词的单位。例如，对“Hello World”这句话，可能会切分为“Hel，lo，Wor，ld”四个子词对。而对于以字为基本单位的中文，一般会按照 字节编码进行切分。例如，在 UTF-8 编码中，“我”会被编码为“E68891”，那么在 BPE 中可能就会切分成“E68”，“891”两个字词对。

一般来说，BPE 编码的词典越大，编码效果越好。当然，由于 Embedding 层就是把 token 从词典空间映射到隐藏空间（也就是说 Embedding 的形状为 (vocab_size, hidden_size)，越大的词表也会带来模型参数的增加。

BERT 原始的 BPE 词表大小为 30K，RoBERTa 选择了 50K 大小的词表来优化模型的编码能力。

通过上述三个部分的优化，RoBERTa 成功地在 BERT 架构的基础上刷新了多个下游任务的 SOTA，也一度成为 BERT 系模型最热门的预训练模型。同时，RoBERTa 的成功也证明了更大的预训练数据、更大的预训练步长的重要意义，这也是 LLM 诞生的基础之一。

3.1.3 ALBERT

在 BERT 的基础上，RoBERTa 进一步探究了更大规模预训练的作用。同样是基于 BERT 架构进行优化的 ALBERT 模型，则从是否能够减小模型参数保持模型能力的角度展开了探究。通过对模型结构进行优化并对 NSP 预训练任务进行改进，ALBERT 成功地以更小规模的参数实现了超越 BERT 的能力。虽然 ALBERT 所提出的一些改进思想并没有在后续研究中被广泛采用，但其降低模型参数的方法及提出的新预训练任务 SOP 仍然对 NLP 领域提供了重要的参考意义。

（1）优化一：将 Embedding 参数进行分解

BERT 等预训练模型具有远超传统神经网络的参数量，如前所述，BERT-large 具有 24层 Encoder Layer，1024 的隐藏层维度，总共参数量达 340M。而这其中，Embedding 层的参数矩阵维度为 $V * H$，此处的 V 为词表大小 30K，H 即为隐藏层大小 768，也就是 Embedding 层参数达到了 23M。而这样的设置还会带来一个更大的问题，即 Google 探索尝试搭建更宽（也就是隐藏层维度更大）的模型时发现，隐藏层维度的增加会带来 Embedding 层参数的巨大上升，如果把隐藏层维度增加到 2048，Embedding 层参数就会膨胀到 61M，这无疑是极大增加了模型的计算开销。

而从另一个角度看，Embedding 层输出的向量是我们对文本 token 的稠密向量表示，从 Word2Vec 的成功经验来看，这种词向量并不需要很大的维度，Word2Vec 仅使用了 100维大小就取得了很好的效果。因此，Embedding 层的输出也许不需要和隐藏层大小一致。

因此，ALBERT 对 Embedding 层的参数矩阵进行了分解，让 Embedding 层的输出维度和隐藏层维度解绑，也就是在 Embedding 层的后面加入一个线性矩阵进行维度变换。ALBERT 设置了 Embedding 层的输出为 128，因此在 Embedding 层后面加入了一个 $128 * 768$ 的线性矩阵来将 Embedding 层的输出再升维到隐藏层大小。也就是说，Embedding 层的参数从 $V * H$ 降低到了 $V * E + E * H$，当 E 的大小远小于 H 时，该方法对 Embedding 层参数的优化就会很明显。

（2）优化二：跨层进行参数共享

通过对 BERT 的参数进行分析，ALBERT 发现各个 Encoder 层的参数出现高度一致的情况。由于 24个 Encoder 层带来了巨大的模型参数，因此，ALBERT 提出，可以让各个 Encoder 层共享模型参数，来减少模型的参数量。

在具体实现上，其实就是 ALBERT 仅初始化了一个 Encoder 层。在计算过程中，仍然会进行 24次计算，但是每一次计算都是经过这一个 Encoder 层。因此，虽然是 24个 Encoder 计算的模型，但只有一层 Encoder 参数，从而大大降低了模型参数量。在这样的情况下，就可以极大程度地扩大隐藏层维度，实现一个更宽但参数量更小的模型。ALBERT 通过实验证明，相较于 334M 的 BERT，同样是 24层 Encoder 但将隐藏层维度设为 2048 的 ALBERT（xlarge 版本）仅有 59M 的参数量，但在具体效果上还要更优于 BERT。

但是，上述优化虽然极大程度减小了模型参数量并且还提高了模型效果，却也存在着明显的不足。虽然 ALBERT 的参数量远小于 BERT，但训练效率却只略微优于 BERT，因为在模型的设置中，虽然各层共享权重，但计算时仍然要通过 24次 Encoder Layer 的计算，也就是说训练和推理时的速度相较 BERT 还会更慢。这也是 ALBERT 最终没能取代 BERT 的一个重要原因。

（3）优化三：提出 SOP 预训练任务

类似于 RoBERTa，ALBERT 也同样认为 NSP 任务过于简单，在预训练中无法对模型效果的提升带来显著影响。但是不同于 RoBERTa 选择直接去掉 NSP，ALBERT 选择改进 NSP，增加其难度，来优化模型的预训练。

在传统的 NSP 任务中，正例是由两个连续句子组成的句对，而负例则是从任意两篇文档中抽取出的句对，模型可以较容易地判断正负例，并不能很好地学习深度语义。而 SOP 任务提出的改进是，正例同样由两个连续句子组成，但负例是将这两个的顺序反过来。也就是说，模型不仅要拟合两个句子之间的关系，更要学习其顺序关系，这样就大大提升了预训练的难度。例如，相较于我们在上文中提出的 NSP 任务的示例，SOP 任务的示例形如：

```
输入：
    Sentence A：I love you.
    Sentence B：Because you are wonderful.
输出：
    1（正样本）

输入：
    Sentence A：Because you are wonderful.
    Sentence B：I love you.
输出：
    0（负样本）
```

ALBERT 通过实验证明，SOP 预训练任务对模型效果有显著提升。使用 MLM + SOP 预训练的模型效果优于仅使用 MLM 预训练的模型更优于使用 MLM + NSP 预训练的模型。

通过上述三点优化，ALBERT 成功地以更小的参数实现了更强的性能，虽然由于其架构带来的训练、推理效率降低限制了模型的进一步发展，但打造更宽的模型这一思路仍然为众多更强大的模型提供了参考价值。

作为预训练时代的 NLP 王者，BERT 及 BERT 系模型在多个 NLP 任务上扮演了极其重要的角色。除去上文介绍过的 RoBERTa、ALBERT 外，还有许多从其他更高角度对 BERT 进行优化的后起之秀，包括进一步改进了预训练任务的 ERNIE、对 BERT 进行蒸馏的小模型 DistilBERT、主打多语言任务的 XLM 等，本文就不再一一赘述。以 BERT 为代表的 Encoder-Only 架构并非 Transformer 的唯一变种，接下来，我们将介绍 Transformer 的另一种主流架构，与原始 Transformer 更相似、以 T5 为代表的 Encoder-Decoder 架构。

3.2 Encoder-Decoder PLM

在上一节，我们学习了 Encoder-Only 结构的模型，主要介绍了 BERT 的模型架构、预训练任务和下游任务微调。BERT 是一个基于 Transformer 的 Encoder-Only 模型，通过预训练任务 MLM 和 NSP 来学习文本的双向语义关系，从而在下游任务中取得了优异的性能。但是，BERT 也存在一些问题，例如 MLM 任务和下游任务微调的不一致性，以及无法处理超过模型训练长度的输入等问题。为了解决这些问题，研究者们提出了 Encoder-Decoder 模型，通过引入 Decoder 部分来解决这些问题，同时也为 NLP 领域带来了新的思路和方法。

在本节中，我们将学习 Encoder-Decoder 结构的模型，主要介绍 T5 的模型架构和预训练任务，以及 T5 模型首次提出的 NLP 大一统思想。

3.2.1 T5

T5（Text-To-Text Transfer Transformer）是由 Google 提出的一种预训练语言模型，通过将所有 NLP 任务统一表示为文本到文本的转换问题，大大简化了模型设计和任务处理。T5 基于 Transformer 架构，包含编码器和解码器两个部分，使用自注意力机制和多头注意力捕捉全局依赖关系，利用相对位置编码处理长序列中的位置信息，并在每层中包含前馈神经网络进一步处理特征。

T5 的大一统思想将不同的 NLP 任务如文本分类、问答、翻译等统一表示为输入文本到输出文本的转换，这种方法简化了模型设计、参数共享和训练过程，提高了模型的泛化能力和效率。通过这种统一处理方式，T5不仅减少了任务特定的模型调试工作，还能够使用相同的数据处理和训练框架，极大地提升了多任务学习的性能和应用的便捷性。接下来我们将会从模型结构、预训练任务和大一统思想三个方面来介绍 T5 模型。

（1）模型结构：Encoder-Decoder

BERT 采用了 Encoder-Only 结构，只包含编码器部分；而 GPT 采用了 Decoder-Only 结构，只包含解码器部分。T5 则采用了 Encoder-Decoder 结构，其中编码器和解码器都是基于 Transformer 架构设计。编码器用于处理输入文本，解码器用于生成输出文本。编码器和解码器之间通过注意力机制进行信息交互，从而实现输入文本到输出文本的转换。其主要结构如图3.7所示：

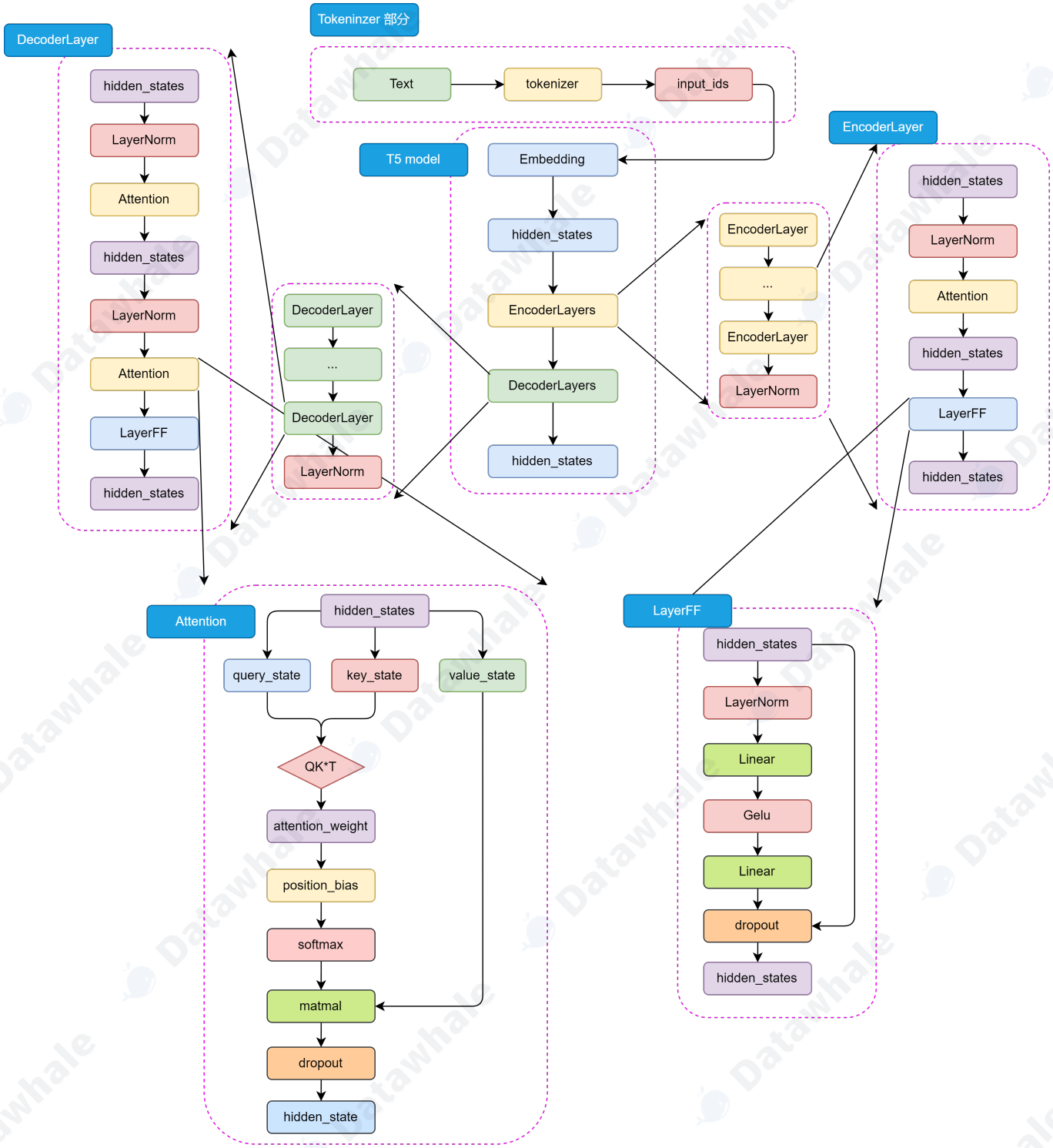


图3.7 T5 模型详细结构

如图3.8所示，从整体来看 T5 的模型结构包括 Tokenizer 部分和 Transformer 部分。Tokenizer 部分主要负责将输入文本转换为模型可接受的输入格式，包括分词、编码等操作。Transformer 部分又分为 EncoderLayers 和 DecoderLayers 两部分，他们分别由一个个小的 Block组成，每个 Block 包含了多头注意力机制、前馈神经网络和 Norm 层。Block 的设计可以使模型更加灵活，像乐高一样可以根据任务的复杂程度和数据集的大小来调整 Block 的数量和层数。

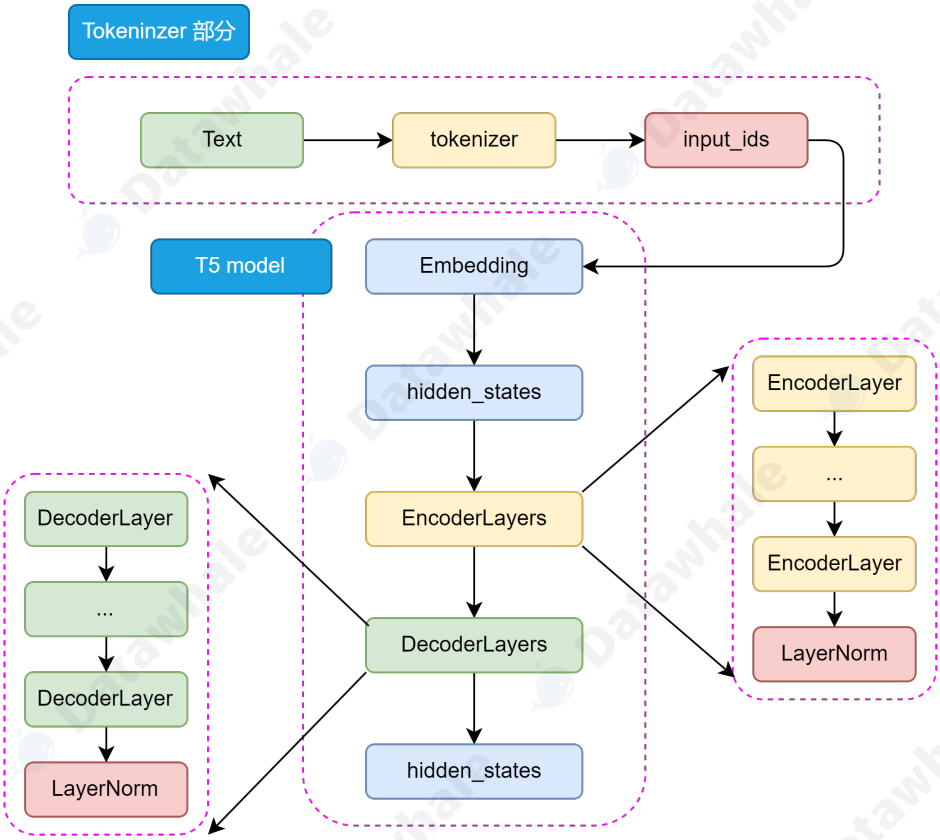


图3.8 T5 模型整体结构

T5 模型的 Encoder 和 Decoder 部分都是基于 Transformer 架构设计的，主要包括 Self-Attention 和前馈神经网络两种结构。Self-Attention 用于捕捉输入序列中的全局依赖关系，前馈神经网络用于处理特征的非线性变换。

和 Encoder 不一样的是，在 Decoder 中还包含了 Encoder-Decoder Attention 结构，用于捕捉输入和输出序列之间的依赖关系。这两种 Attention 结构几乎完全一致，只有在位置编码和 Mask 机制上有所不同。如图3.9所示，Encoder 和 Decoder 的结构如下：

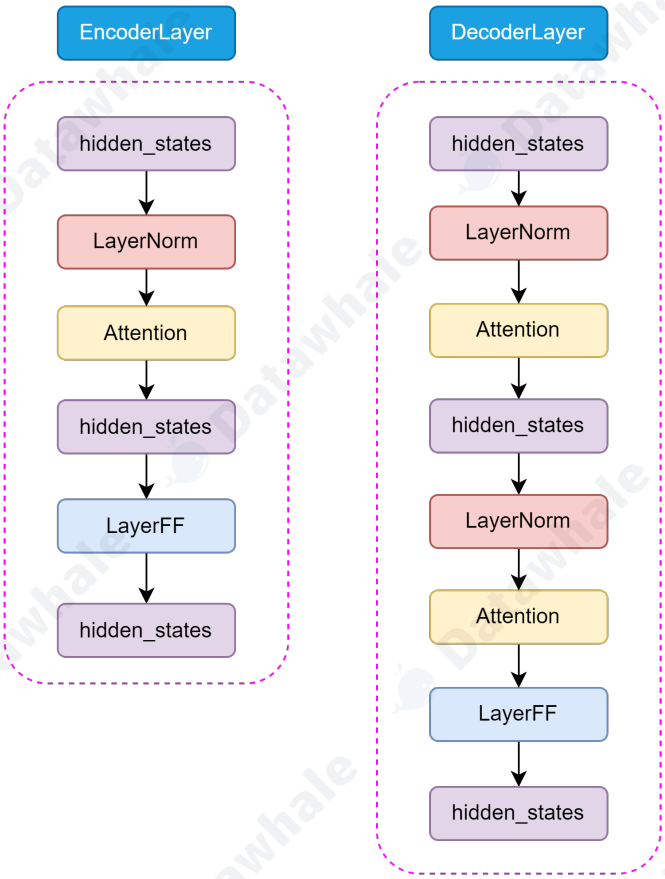


图3.9 Encoder 和 Decoder

T5 的 Self-Attention 机制和 BERT 的 Attention 机制是一样的，都是基于 Self-Attention 机制设计的。Self-Attention 机制是一种全局依赖关系建模方法，通过计算 Query、Key 和 Value 之间的相似度来捕捉输入序列中的全局依赖关系。Encoder-Decoder Attention 仅仅在位置编码和 Mask 机制上有所不同，主要是为了区分输入和输出序列。如图3.10所示，Self-Attention 结构如下：

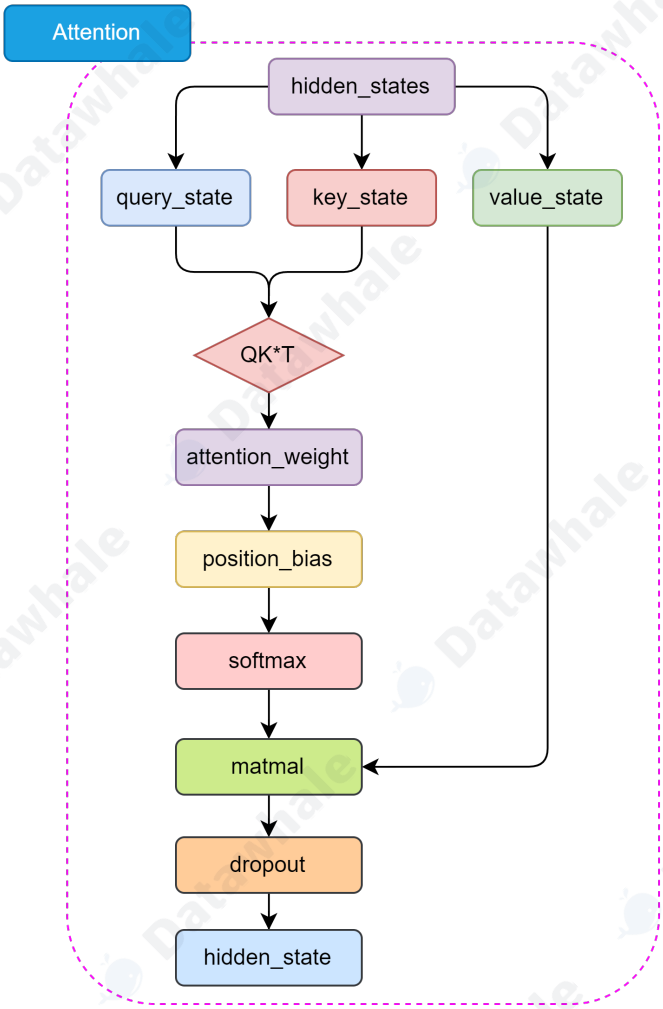


图3.10 Self-Attention 结构

与原始 Transformer 模型不同，T5 模型的LayerNorm 采用了 RMSNorm，通过计算每个神经元的均方根（Root Mean Square）来归一化每个隐藏层的激活值。RMSNorm 的参数设置与Layer Normalization 相比更简单，只有一个可调参数，可以更好地适应不同的任务和数据集。RMSNorm函数可以用以下数学公式表示：

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} w_i^2 + \epsilon}} \tag{1}$$

其中：

- (x) 是层的输入。
- (w_i) 代表层的权重。
- (n) 是权重的数量。
- (ϵ) 是一个小常数，用于数值稳定性（以避免除以零的情况）。

这种归一化有助于通过确保权重的规模不会变得过大或过小来稳定学习过程，这在具有许多层的深度学习模型中特别有用。

（2）预训练任务

T5 模型的预训练任务是一个关键的组成部分，它能使模型能够学习到丰富的语言表示，语言表示能力可以在后续的微调过程中被迁移到各种下游任务。训练所使用的数据集是一个大规模的文本数据集，包含了各种各样的文本数据，如维基百科、新闻、书籍等等。对数据经过细致的处理后，生成了用于训练的750GB 的数据集 C4，且已在 TensorflowData 中开源。

我们可以简单概括一下 T5 的预训练任务，主要包括以下几个部分：

- 预训练任务: T5模型的预训练任务是 MLM，也称为BERT-style目标。具体来说，就是在输入文本中随机遮蔽15%的token，然后让模型预测这些被遮蔽的token。这个过程不需要标签，可以在大量未标注的文本上进行。
- 输入格式: 预训练时，T5将输入文本转换为"文本到文本"的格式。对于一个给定的文本序列，随机选择一些token进行遮蔽，并用特殊的占位符(token)替换。然后将被遮蔽的token序列作为模型的输出目标。
- 预训练数据集: T5 使用了自己创建的大规模数据集"Colossal Clean Crawled Corpus"(C4)，该数据集从Common Crawl中提取了大量干净的英语文本。C4数据集经过了一定的清洗，去除了无意义的文本、重复文本等。
- 多任务预训练: T5 还尝试了将多个任务混合在一起进行预训练，而不仅仅是单独的MLM任务。这有助于模型学习更通用的语言表示。
- 预训练到微调的转换: 预训练完成后，T5模型会在下游任务上进行微调。微调时，模型在任务特定的数据集上进行训练，并根据任务调整解码策略。

通过大规模预训练，T5模型能够学习到丰富的语言知识，并获得强大的语言表示能力，在多个NLP任务上取得了优异的性能，预训练是T5成功的关键因素之一。

（3）大一统思想

T5模型的一个核心理念是“大一统思想”，即所有的 NLP 任务都可以统一为文本到文本的任务，这一思想在自然语言处理领域具有深远的影响。其设计理念是将所有不同类型的NLP任务（如文本分类、翻译、文本生成、问答等）转换为一个统一的格式：输入和输出都是纯文本。

例如：

- 对于文本分类任务，输入可以是“classify: 这是一个很好的产品”，输出是“正面”；
- 对于翻译任务，输入可以是“translate English to French: How are you?”, 输出是“Comment ça va?”。

T5通过大规模的文本数据进行预训练，然后在具体任务上进行微调。这一过程与BERT、GPT等模型类似，但T5将预训练和微调阶段的任务统一为文本到文本的形式，使其在各种任务上的适应性更强。

我们可以通过图3.11，更加直观地理解 T5 的大一统思想：

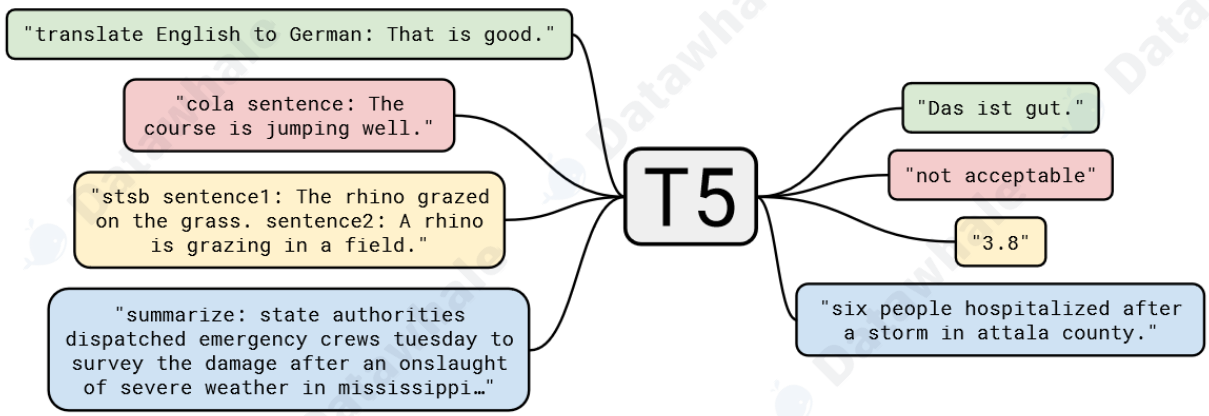


图3.11 T5 的大一统思想

对于不同的NLP任务，每次输入前都会加上一个任务描述前缀，明确指定当前任务的类型。这不仅帮助模型在预训练阶段学习到不同任务之间的通用特征，也便于在微调阶段迅速适应具体任务。例如，任务前缀可以是“summarize: ”用于摘要任务，或“translate English to German: ”用于翻译任务。

T5的大一统思想通过将所有NLP任务统一为文本到文本的形式，简化了任务处理流程，增强了模型的通用性和适应性。这一思想不仅推动了自然语言处理技术的发展，也为实际应用提供了更为便捷和高效的解决方案。

3.3 Decoder-Only PLM

在前两节中，我们分别讲解了由 Transformer 发展而来的两种模型架构——以 BERT 为代表的 Encoder-Only 模型和以 T5 为代表的 Encoder-Decoder 模型。那么，很自然可以想见，除了上述两种架构，还可以有一种模型架构——Decoder-Only，即只使用 Decoder 堆叠而成的模型。

事实上，Decoder-Only 就是目前大火的 LLM 的基础架构，目前所有的 LLM 基本都是 Decoder-Only 模型（RWKV、Mamba 等非 Transformer 架构除外）。而引发 LLM 热潮的 ChatGPT，正是 Decoder-Only 系列的代表模型 GPT 系列模型的大成之作。而目前作为开源 LLM 基本架构的 LLaMA 模型，也正是在 GPT 的模型架构基础上优化发展而来。因此，在本节中，我们不但会详细分析 Decoder-Only 代表模型 GPT 的原理、架构和特点，还会深入到目前的主流开源 LLM，分析它们的结构、特点，结合之前对 Transformer 系列其他模型的分析，帮助大家深入理解当下被寄予厚望、被认为是 AGI 必经之路的 LLM 是如何一步步从传统 PLM 中发展而来的。

首先，让我们学习打开 LLM 世界大门的代表模型——由 OpenAI 发布的 GPT。

3.3.1 GPT

GPT，即 Generative Pre-Training Language Model，是由 OpenAI 团队于 2018年发布的预训练语言模型。虽然学界普遍认可 BERT 作为预训练语言模型时代的代表，但首先明确提出预训练-微调思想的模型其实是 GPT。GPT 提出了通用预训练的概念，也就是在海量无监督语料上预训练，进而在每个特定任务上进行微调，从而实现这些任务的巨大收益。虽然在发布之初，由于性能略输于不久后发布的 BERT，没能取得轰动性成果，也没能让 GPT 所使用的 Decoder-Only 架构成为学界研究的主流，但 OpenAI 团队坚定地选择了不断扩大预训练数据、增加模型参数，在 GPT 架构上不断优化，最终在 2020年发布的 GPT-3 成就了 LLM 时代的基础，并以 GPT-3 为基座模型的 ChatGPT 成功打开新时代的大门，成为 LLM 时代的最强竞争者也是目前的最大赢家。

本节将以 GPT 为例，分别从模型架构、预训练任务、GPT 系列模型的发展历程等三个方面深入分析 GPT 及其代表的 Decoder-Only 模型，并进一步引出当前的主流 LLM 架构——LLaMA。

（1） 模型架构——Decoder Only

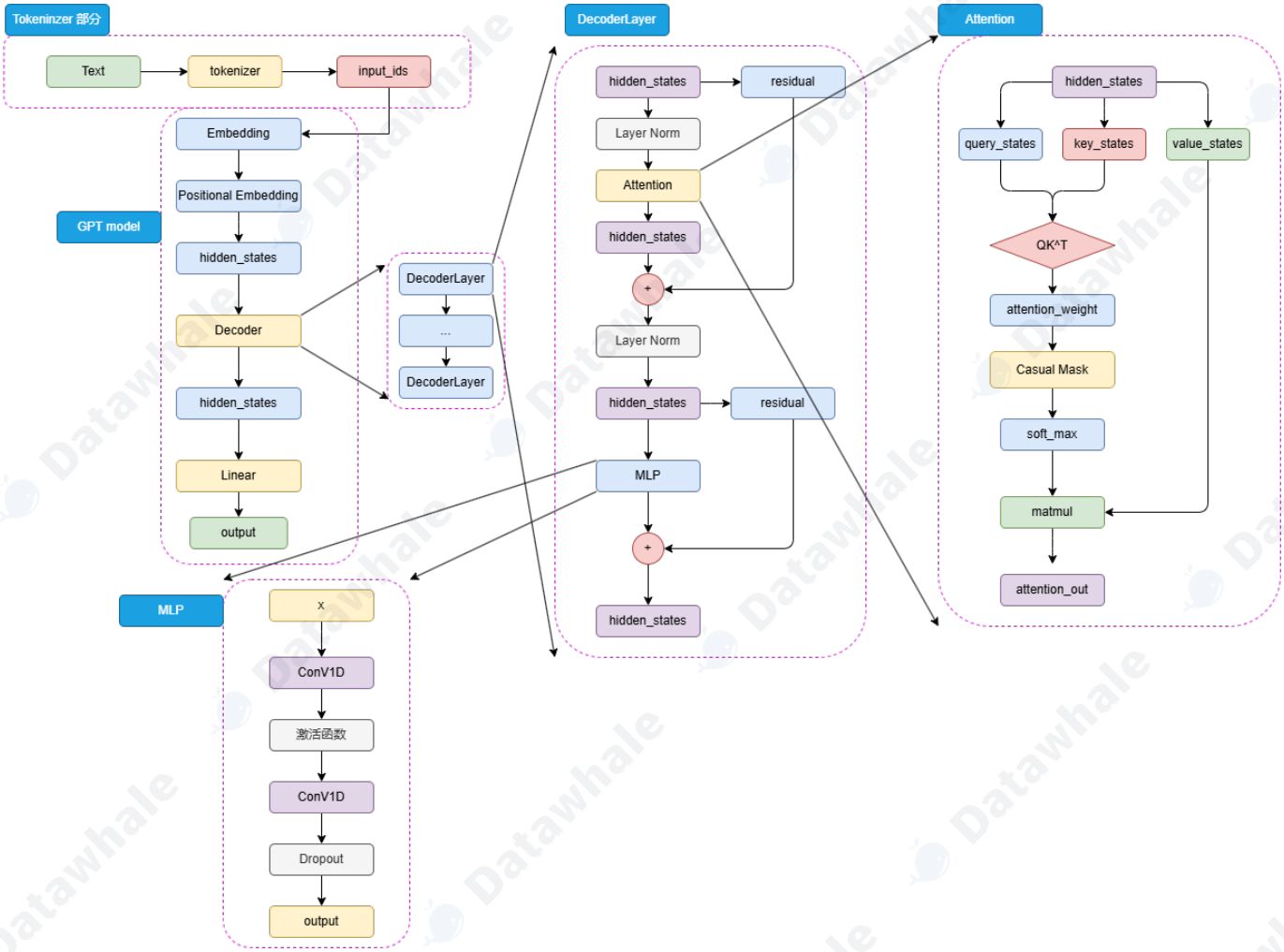


图3.12 GPT 模型结构

如图3.12可以看到，GPT 的整体结构和 BERT 是有一些类似的，只是相较于 BERT 的 Encoder，选择使用了 Decoder 来进行模型结构的堆叠。由于 Decoder-Only 结构也天生适用于文本生成任务，所以相较于更贴合 NLU 任务设计的 BERT，GPT 和 T5 的模型设计更契合于 NLG 任务和 Seq2Seq 任务。同样，对于一个自然语言文本的输入，先通过 tokenizer 进行分词并转化为对应词典序号的 input_ids。

输入的 input_ids 首先通过 Embedding 层，再经过 Positional Embedding 进行位置编码。不同于 BERT 选择了可训练的全连接层作为位置编码，GPT 沿用了 Transformer 的经典 Sinusoidal 位置编码，即通过三角函数进行绝对位置编码，此处就不再赘述，感兴趣的读者可以参考第二章 Transformer 模型细节的解析。

通过 Embedding 层和 Positional Embedding 层编码成 hidden_states 之后，就可以进入到解码器（Decoder），第一代 GPT 模型和原始 Transformer 模型类似，选择了 12层解码器层，但是在解码器层的内部，相较于 Transformer 原始 Decoder 层的双注意力层设计，GPT 的 Decoder 层反而更像 Encoder 层一点。由于不再有 Encoder 的编码输入，Decoder 层仅保留了一个带掩码的注意力层，并且将 LayerNorm 层从 Transformer 的注意力层之后提到了注意力层之前。hidden_states 输入 Decoder 层之后，会先进行 LayerNorm，再进行掩码注意力计算，然后经过残差连接和再一次 LayerNorm 进入到 MLP 中并得到最后输出。

由于不存在 Encoder 的编码结果，Decoder 层中的掩码注意力也是自注意力计算。也就是对一个输入的 hidden_states，会通过三个参数矩阵来生成 query、key 和 value，而不再是像 Transformer 中的 Decoder 那样由 Encoder 输出作为 key 和 value。后续的注意力计算过程则和 BERT 类似，只是在计算得到注意力权重之后，通过掩码矩阵来遮蔽了未来 token 的注意力权重，从而限制每一个 token 只能关注到它之前 token 的注意力，来实现掩码自注意力的计算。

另外一个结构上的区别在于，GPT 的 MLP 层没有选择线性矩阵来进行特征提取，而是选择了两个一维卷积核来提取，不过，从效果上说这两者是没有太大区别的。通过 N 个 Decoder 层后的 hidden_states 最后经过线性矩阵映射到词表维度，就可以转化成自然语言的 token，从而生成我们的目标序列。

（2）预训练任务——CLM

Decoder-Only 的模型结构往往更适合于文本生成任务，因此，Decoder-Only 模型往往选择了最传统也最直接的预训练任务——因果语言模型，Casual Language Model，下简称 CLM。

CLM 可以看作 N-gram 语言模型的一个直接扩展。N-gram 语言模型是基于前 N 个 token 来预测下一个 token，CLM 则是基于一个自然语言序列的前面所有 token 来预测下一个 token，通过不断重复该过程来实现目标文本序列的生成。也就是说，CLM 是一个经典的补全形式。例如，CLM 的输入和输出可以是：

```
input：今天天气
output：今天天气很

input：今天天气很
output：今天天气很好
```

因此，对于一个输入目标序列长度为 256，期待输出序列长度为 256 的任务，模型会不断根据前 256 个 token、257个 token（输入+预测出来的第一个 token）..... 进行 256 次计算，最后生成一个序列长度为 512 的输出文本，这个输出文本前 256 个 token 为输入，后 256 个 token 就是我们期待的模型输出。

在前面我们说过，BERT 之所以可以采用预训练+微调的范式取得重大突破，正是因为其选择的 MLM、NSP 可以在海量无监督语料上直接训练——而很明显，CLM 是更直接的预训练任务，其天生和人类书写自然语言文本的习惯相契合，也和下游任务直接匹配，相对于 MLM 任务更加直接，可以在任何自然语言文本上直接应用。因此，CLM 也可以使用海量的自然语言语料进行大规模的预训练。

（3）GPT 系列模型的发展

自 GPT-1 推出开始，OpenAI 一直坚信 Decoder-Only 的模型结构和“体量即正义”的优化思路，不断扩大预训练数据集、模型体量并对模型做出一些小的优化和修正，来不断探索更强大的预训练模型。从被 BERT 压制的 GPT-1，到没有引起足够关注的 GPT-2，再到激发了涌现能力、带来大模型时代的 GPT-3，最后带来了跨时代的 ChatGPT，OpenAI 通过数十年的努力证明了其思路的正确性。

下表总结了从 GPT-1 到 GPT-3 的模型结构、预训练语料大小的变化：

模型	Decoder Layer	Hidden_size	注意力头数	注意力维度	总参数量	预训练语料
GPT-1	12	3072	12	768	0.12B	5GB
GPT-2	48	6400	25	1600	1.5B	40GB
GPT-3	96	49152	96	12288	175B	570GB

GPT-1 是 GPT 系列的开山之作，也是第一个使用 Decoder-Only 的预训练模型。但是，GPT-1 的模型体量和预训练数据都较少，沿承了传统 Transformer 的模型结构，使用了 12层 Decoder Block 和 768 的隐藏层维度，模型参数量仅有 1.17亿（0.12B），在大小为 5GB 的 BooksCorpus 数据集上预训练得到。可以看到，GPT-1 的参数规模与预训练规模和 BERT-base 是大致相当的，但其表现相较于 BERT-base 却有所不如，这也是 GPT 系列模型没能成为预训练语言模型时代的代表的原因。

GPT-2 则是 OpenAI 在 GPT-1 的基础上进一步探究预训练语言模型多任务学习能力的产物。GPT-2 的模型结构和 GPT-1 大致相当，只是扩大了模型参数规模、将 Post-Norm 改为了 Pre-Norm（也就是先进行 LayerNorm 计算，再进入注意力层计算）。这些改动的核心原因在于，由于模型层数增加、体量增大，梯度消失和爆炸的风险也不断增加，为了使模型梯度更稳定对上述结构进行了优化。

GPT-2 的核心改进是大幅增加了预训练数据集和模型体量。GPT-2 的 Decoder Block 层数达到了48（注意，GPT-2 共发布了四种规格的模型，此处我们仅指规格最大的 GPT-2 模型），隐藏层维度达到了 1600，模型整体参数量达 15亿（1.5B），使用了自己抓取的 40GB 大小的 WebText 数据集进行预训练，不管是模型结构还是预训练大小都超过了 1代一个数量级。

GPT-2 的另一个重大突破是以 zero-shot（零样本学习）为主要目标，也就是不对模型进行微调，直接要求模型解决任务。例如，在传统的预训练-微调范式中，我们要解决一个问题，一般需要收集几百上千的训练样本，在这些训练样本上微调预训练语言模型来实现该问题的解决。而 zero-shot 则强调不使用任何训练样本，直接通过向预训练语言模型描述问题来去解决该问题。zero-shot 的思路自然是比预训练-微调范式更进一步、更高效的自然语言范式，但是在 GPT-2 的时代，模型能力还不足够支撑较好的 zero-shot 效果，在大模型时代，zero-shot 及其延伸出的 few-shot（少样本学习）才开始逐渐成为主流。

GPT-3 则是更进一步展示了 OpenAI“力大砖飞”的核心思路，也是 LLM 的开创之作。在 GPT-2 的基础上，OpenAI 进一步增大了模型体量和预训练数据量，整体参数量达 175B，是当之无愧的“大型语言模型”。在模型结构上，基本没有大的改进，只是由于巨大的模型体量使用了稀疏注意力机制来取代传统的注意力机制。在预训练数据上，则是分别从 CC、WebText、维基百科等大型语料集中采样，共采样了 45T、清洗后 570GB 的数据。根据推算，GPT-3 需要在 1024张 A100（80GB 显存）的分布式训练集群上训练 1个月。

之所以说 GPT-3 是 LLM 的开创之作，除去其巨大的体量带来了涌现能力的凸显外，还在于其提出了 few-shot 的重要思想。few-shot 是在 zero-shot 上的改进，研究者发现即使是 175B 大小的 GPT-3，想要在 zero-shot 上取得较好的表现仍然是一件较为困难的事情。而 few-shot 是对 zero-shot 的一个折中，旨在提供给模型少样的示例来教会它完成任务。few-shot 一般会在 prompt（也就是模型的输入）中增加 3~5个示例，来帮助模型理解。例如，对于情感分类任务：

```
zero-shot：请你判断‘这真是一个绝佳的机会’的情感是正向还是负向，如果是正向，输出1；否则输出0

few-shot：请你判断‘这真是一个绝佳的机会’的情感是正向还是负向，如果是正向，输出1；否则输出0。你可以参考以下示例来判断：‘你的表现非常好’——1；‘太糟糕了’——0；‘真是一个好主意’——1。
```

通过给模型提供少量示例，模型可以取得远好于 zero-shot 的良好表现。few-shot 也被称为上下文学习（In-context Learning），即让模型从提供的上下文中的示例里学习问题的解决方法。GPT-3 在 few-shot 上展现的强大能力，为 NLP 的突破带来了重要进展。如果对于绝大部分任务都可以通过人为构造 3~5个示例就能让模型解决，其效率将远高于传统的预训练-微调范式，意味着 NLP 的进一步落地应用成为可能——而这，也正是 LLM 的核心优势。

在 GPT 系列模型的基础上，通过引入预训练-指令微调-人类反馈强化学习的三阶段训练，OpenAI 发布了跨时代的 ChatGPT，引发了大模型的热潮。也正是在 GPT-3 及 ChatGPT 的基础上，LLaMA、ChatGLM 等模型的发布进一步揭示了 LLM 的无尽潜力。在下一节，我们将深入剖析目前 LLM 的普适架构——LLaMA。

3.3.2 LLaMA

LLaMA模型是由Meta（前Facebook）开发的一系列大型预训练语言模型。从LLaMA-1到LLaMA-3，LLaMA系列模型展示了大规模预训练语言模型的演进及其在实际应用中的显著潜力。

（1） 模型架构——Decoder Only

与GPT系列模型一样，LLaMA模型也是基于Decoder-Only架构的预训练语言模型。LLaMA模型的整体结构与GPT系列模型类似，只是在模型规模和预训练数据集上有所不同。如图3.13是LLaMA模型的架构示意图：

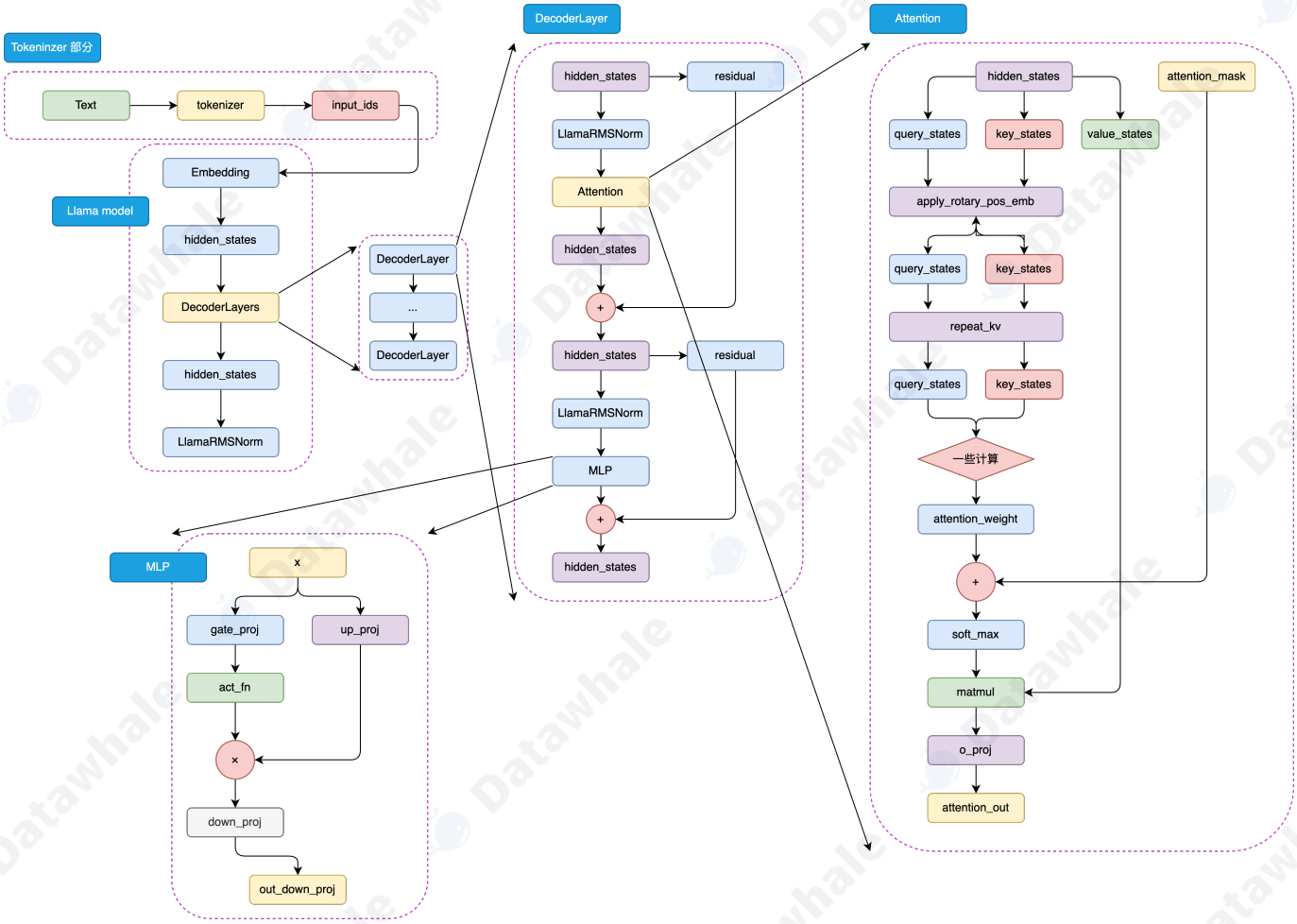


图3.13 LLaMA-3 模型结构

与GPT类似，LLaMA模型的处理流程也始于将输入文本通过tokenizer进行编码，转化为一系列的input_ids。这些input_ids是模型能够理解和处理的数据格式。接下来，这些input_ids会经过embedding层的转换，这里每个input_id会被映射到一个高维空间中的向量，即词向量。同时，输入文本的位置信息也会通过positional embedding层被编码，以确保模型能够理解词序上下文信息。

这样，input_ids经过embedding层和positional embedding层的结合，形成了hidden_states。hidden_states包含了输入文本的语义和位置信息，是模型进行后续处理的基础，hidden_states随后被输入到模型的decoder层。

在decoder层中，hidden_states会经历一系列的处理，这些处理由多个decoder block组成。每个decoder block都是模型的核心组成部分，它们负责对hidden_states进行深入的分析和转换。在每个decoder block内部，首先是一个masked self-attention层。在这个层中，模型会分别计算query、key和value这三个向量。这些向量是通过hidden_states线性变换得到的，它们是计算注意力权重的基础。然后使用softmax函数计算attention score，这个分数反映了不同位置之间的关联强度。通过attention score，模型能够确定在生成当前词时，应该给予不同位置的hidden_states多大的关注。然后，模型将value向量与attention score相乘，得到加权后的value，这就是attention的结果。

在完成masked self-attention层之后，hidden_states会进入MLP层。在这个多层感知机层中，模型通过两个全连接层对hidden_states进行进一步的特征提取。第一个全连接层将hidden_states映射到一个中间维度，然后通过激活函数进行非线性变换，增加模型的非线性能力。第二个全连接层则将特征再次映射回原始的hidden_states维度。

最后，经过多个decoder block的处理，hidden_states会通过一个线性层进行最终的映射，这个线性层的输出维度与词表维度相同。这样，模型就可以根据hidden_states生成目标序列的概率分布，进而通过采样或贪婪解码等方法，生成最终的输出序列。这一过程体现了LLaMA模型强大的序列生成能力。

（2） LLaMA模型的发展历程

LLaMA-1 系列：

- Meta于2023年2月发布了LLaMA-1，包括7B、13B、30B和65B四个参数量版本。
- 这些模型在超过1T token的语料上进行了预训练，其中最大的65B参数模型在2,048张A100 80G GPU上训练了近21天。
- LLaMA-1因其开源性和优异性能迅速成为开源社区中最受欢迎的大模型之一。

LLaMA-2 系列：

- 2023年7月，Meta发布了LLaMA-2，包含7B、13B、34B和70B四个参数量版本，除了34B模型外，其他均已开源。
- LLaMA-2将预训练的语料扩充到了2T token，并将模型的上下文长度从2,048翻倍到了4,096。
- 引入了分组查询注意力机制（Grouped-Query Attention, GQA）等技术。

LLaMA-3 系列：

- 2024年4月，Meta发布了LLaMA-3，包括8B和70B两个参数量版本，同时透露400B的LLaMA-3还在训练中。
- LLaMA-3支持8K长文本，并采用了编码效率更高的tokenizer，词表大小为128K。
- 使用了超过15T token的预训练语料，是LLaMA-2的7倍多。

LLaMA模型以其技术创新、多参数版本、大规模预训练和高效架构设计而著称。模型支持从7亿到数百亿不等的参数量，适应不同规模的应用需求。LLaMA-1以其开源性和优异性能迅速受到社区欢迎，而LLaMA-2和LLaMA-3进一步通过引入分组查询注意力机制和支持更长文本输入，显著提升了模型性能和应用范围。特别是LLaMA-3，通过采用128K词表大小的高效tokenizer和15T token的庞大训练数据，实现了在多语言和多任务处理上的重大进步。Meta对模型安全性和社区支持的持续关注，预示着LLaMA将继续作为AI技术发展的重要推动力，促进全球范围内的技术应用和创新。

3.3.3 GLM

GLM 系列模型是由智谱开发的主流中文 LLM 之一，包括 ChatGLM1、2、3及 GLM-4 系列模型，覆盖了指令理解、代码生成等多种应用场景，曾在多种中文评估集上达到 SOTA 性能。

ChatGLM-6B 是 GLM 系列的开山之作，也是 2023年国内最早的开源中文 LLM，也是最早提出不同于 GPT、LLaMA 的独特模型架构的 LLM。在整个中文 LLM 的发展历程中，GLM 具有独特且重大的技术意义。本节将简要叙述 GLM 系列的发展，并介绍其不同于 GPT、LLaMA 系列模型的独特技术思路。

（1）模型架构-相对于 GPT 的略微修正

GLM 最初是由清华计算机系推出的一种通用语言模型基座，其核心思路是在传统 CLM 预训练任务基础上，加入 MLM 思想，从而构建一个在 NLG 和 NLU 任务上都具有良好表现的统一模型。

在整体模型结构上，GLM 和 GPT 大致类似，均是 Decoder-Only 的结构，仅有三点细微差异：

1. 使用 Post Norm 而非 Pre Norm。Post Norm 是指在进行残差连接计算时，先完成残差计算，再进行 LayerNorm 计算；而类似于 GPT、LLaMA 等模型都使用了 Pre Norm，也就是先进行 LayerNorm 计算，再进行残差的计算。相对而言，Post Norm 由于在残差之后做归一化，对参数正则化的效果更强，进而模型的鲁棒性也会更好；Pre Norm相对于因为有一部分参数直接加在了后面，不需要对这部分参数进行正则化，正好可以防止模型的梯度爆炸或者梯度消失。因此，对于更大体量的模型来说，一般认为 Pre Norm 效果会更好。但 GLM 论文提出，使用 Post Norm 可以避免 LLM 的数值错误（虽然主流 LLM 仍然使用了 Pre Norm）；
2. 使用单个线性层实现最终 token 的预测，而不是使用 MLP；这样的结构更加简单也更加鲁棒，即减少了最终输出的参数量，将更大的参数量放在了模型本身；
3. 激活函数从 ReLU 换成了 GeLUS。ReLU 是传统的激活函数，其核心计算逻辑为去除小于 0的传播，保留大于0的传播；GeLUS 核心是对接近于 0的正向传播，做了一个非线性映射，保证了激活函数后的非线性输出，具有一定的连续性。

（2）预训练任务-GLM

GLM 的核心创新点主要在于其提出的 GLM（General Language Model，通用语言模型）任务，这也是 GLM 的名字由来。GLM 是一种结合了自编码思想和自回归思想的预训练方法。所谓自编码思想，其实也就是 MLM 的任务学习思路，在输入文本中随机删除连续的 tokens，要求模型学习被删除的 tokens；所谓自回归思想，其实就是传统的 CLM 任务学习思路，也就是要求模型按顺序重建连续 tokens。

GLM 通过优化一个自回归空白填充任务来实现 MLM 与 CLM 思想的结合。其核心思想是，对于一个输入序列，会类似于 MLM 一样进行随机的掩码，但遮蔽的不是和 MLM 一样的单个 token，而是每次遮蔽一连串 token；模型在学习时，既需要使用遮蔽部分的上下文预测遮蔽部分，在遮蔽部分内部又需要以 CLM 的方式完成被遮蔽的 tokens 的预测。例如，输入和输出可能是：

```
输入：I <MASK> because you <MASK>
输出：<MASK> - love you; <MASK> - are a wonderful person
```

通过将 MLM 与 CLM 思想相结合，既适配逐个 token 生成的生成类任务，也迫使模型从前后两个方向学习输入文本的隐含关系从而适配了理解类任务。使用 GLM 预训练任务产出的 GLM 模型，在一定程度上展现了其超出同体量 BERT 系模型的优越性能：

Table 1: Results on the SuperGLUE dev set.

Model	ReCoRD F1/Acc.	COPA Acc.	WSC Acc.	RTE Acc.	BoolQ Acc.	WiC Acc.	CB F1/Acc.	MultiRC F1a/EM	Avg
Pretrained on BookCorpus and Wikipedia									
$\text{BERT}_{\text{Base}}$	65.4 / 64.9	66.0	65.4	70.0	74.9	**68.8**	70.9 / 76.8	68.4 / 21.5	66.1
GLM_{Base}	**73.5 / 72.8**	**71.0**	**72.1**	**71.2**	**77.0**	64.7	**89.5 / 85.7**	**72.1 / 26.1**	**70.7**
$\text{BERT}_{\text{Large}}$	76.3 / 75.6	69.0	64.4	73.6	80.1	**71.0**	94.8 / 92.9	71.9 / 24.1	72.0
$\text{UniLM}_{\text{Large}}$	80.0 / 79.1	72.0	65.4	76.5	80.5	69.7	91.0 / 91.1	**77.2 / 38.2**	74.1
$\text{GLM}_{\text{Large}}$	81.7 / 81.1	76.0	**81.7**	74.0	**82.1**	68.5	96.1 / 94.6	77.1 / 36.3	77.0
GLM_{Doc}	80.2 / 79.6	77.0	78.8	76.2	79.8	63.6	**97.3 / 96.4**	74.6 / 32.1	75.7
GLM_{Sent}	80.7 / 80.2	77.0	79.8	79.1	80.8	70.4	94.6 / 93.7	76.9 / 36.1	76.8
GLM_{410M}	81.5 / 80.9	80.0	**81.7**	**79.4**	81.9	69.0	93.2 / **96.4**	76.2 / 35.5	78.0
GLM_{515M}	**82.3 / 81.7**	**85.0**	**81.7**	79.1	81.3	69.4	95.0 / **96.4**	**77.2** / 35.0	**78.8**
Pretrained on larger corpora									
T5_{Base}	76.2 / 75.4	73.0	79.8	78.3	80.8	67.9	94.8 / 92.9	76.4 / 40.0	76.0
T5_{Large}	85.7 / 85.0	78.0	**84.6**	84.8	84.3	71.6	96.4 / 98.2	80.9 / 46.6	81.2
$\text{BART}_{\text{Large}}$	88.3 / 87.8	60.0	65.4	84.5	84.3	69.0	90.5 / 92.9	81.8 / 48.0	76.0
$\text{RoBERTa}_{\text{Large}}$	89.0 / 88.4	**90.0**	63.5	87.0	**86.1**	**72.6**	96.1 / 94.6	**84.4 / 52.9**	81.5
$\text{GLM}_{\text{RoBERTa}}$	**89.6 / 89.0**	82.0	83.7	**87.7**	84.7	71.2	**98.7 / 98.2**	82.4 / 50.1	**82.9**

图3.14 alt text

不过，GLM 预训练任务更多的优势还是展现在预训练模型时代，迈入 LLM 时代后，针对于超大规模、体量的预训练，CLM 展现出远超 MLM 的优势。通过将模型体量加大、预训练规模扩大，CLM 预训练得到的生成模型在文本理解上也能具有超出 MLM 训练的理解模型的能力，因此，ChatGLM 系列模型也仅在第一代模型使用了 GLM 的预训练思想，从 ChatGLM2 开始，还是回归了传统的 CLM 建模。虽然从 LLM 的整体发展路径来看，GLM 预训练任务似乎是一个失败的尝试，但通过精巧的设计将 CLM 与 MLM 融合，并第一时间产出了中文开源的原生 LLM，其思路仍然存在较大的借鉴意义。

（3）GLM 家族的发展

在 GLM 模型（即使用原生 GLM 架构及预训练任务的早期预训练模型）的基础上，参考 ChatGPT 的技术思路进行 SFT 和 RLHF，智谱于 23年 3月发布了第一个中文开源 LLM ChatGLM-6B，成为了众多中文 LLM 研究者的起点。ChatGLM-6B 在 1T 语料上进行预训练，支持 2K 的上下文长度。

在 23年 6月，智谱就开源了 ChatGLM2-6B。相对于一代，ChatGLM2 将上下文长度扩展到了 32K，通过更大的预训练规模实现了模型性能的大幅度突破。不过，在 ChatGLM2 中，模型架构就基本回归了 LLaMA 架构，引入 MQA 的注意力机制，预训练任务也回归经典的 CLM，放弃了 GLM 的失败尝试。

ChatGLM3-6B 发布于 23年 10月，相对于二代在语义、数学、推理、代码和知识方面都达到了当时的 SOTA 性能，但是官方给出的技术报告说明 ChatGLM3 在模型架构上相对二代没有变化，最主要的优化来源是更多样化的训练数据集、更充足的训练步骤和更优化的训练策略。ChatGLM3 的另一个重要改进在于其开始支持函数调用与代码解释器，开发者可以直接使用开源的 ChatGLM3 来实现 Agent 开发，具有更广泛的应用价值。

2024年 1月，智谱发布了支持 128K 上下文，包括多种类型的 GLM-4 系列模型，评估其在英文基准上达到了 GPT-4 的水平。不过，智谱并未直接开源 GLM-4，而是开源了其轻量级版本 GLM-4-9B 模型，其在 1T token 的多语言语料库上进行预训练，上下文长度为 8K，并使用与 GLM-4 相同的管道和数据进行后训练。在训练计算量较少的情况下，其超越了 Llama-3-8B，并支持 GLM-4 中所有工具的功能。

图3.15展示了 GLM 系列模型在基准集上的表现演进：

Table 1: Performance of Open ChatGLM-6B, ChatGLM2-6B, ChatGLM3-6B, and GLM-4-9B.

Language	Dataset	ChatGLM-6B (2023-03-14)	ChatGLM2-6B (2023-06-25)	ChatGLM3-6B-Base (2023-10-27)	GLM-4-9B (2024-06-05)
English	GSM8K	1.5	25.9	72.3	84.0
	MATH	3.1	6.9	25.7	30.4
	BBH	0.0	29.2	66.1	76.3
	MMLU	25.2	45.2	61.4	74.7
	GPQA	-	-	26.8	34.3
	HumanEval	0.0	9.8	58.5	70.1
	BoolQ	51.8	79.0	87.9	89.6
	CommonSenseQA	20.5	65.4	86.5	90.7
	HellaSwag	30.4	57.0	79.7	82.6
	PIQA	65.7	69.6	80.1	79.1
	DROP	3.9	25.6	70.9	77.2
Chinese	C-Eval	23.7	51.7	69.0	77.1
	CMMLU	25.3	50.0	67.5	75.1
	GAOKAO-Bench	26.8	46.4	67.3	74.5
	C3	35.1	58.6	73.9	77.2

图3.15 alt text

参考资料

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* arXiv preprint arXiv:1810.04805.

[2] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov. (2019). *RoBERTa: A Robustly Optimized BERT Pretraining Approach.* arXiv preprint arXiv:1907.11692.

[3] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, Radu Soricut. (2020). *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations.* arXiv preprint arXiv:1909.11942.

[4] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu. (2023). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.* arXiv preprint arXiv:1910.10683.

[5] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu. (2020). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.* Journal of Machine Learning Research, 21(140), 1–67.

[6] Alec Radford, Karthik Narasimhan. (2018). *Improving Language Understanding by Generative Pre-Training*. Retrieved from https://api.semanticscholar.org/CorpusID:49313245

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark,

Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei. (2020). *Language Models are Few-Shot Learners.* arXiv preprint arXiv:2005.14165.

[8] 张帆, 陈安东的文章"万字长文带你梳理Llama开源家族：从Llama-1到Llama-3"，来源：https://mp.weixin.qq.com/s/5_VnzP3JmOB0D5geV5HRFg

[9] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, Jingyu Sun, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu, Minlie Huang, Peng Zhang, Qinkai Zheng, Rui Lu, Shuaiqi Duan, Shudan Zhang, Shulin Cao, Shuxun Yang, Weng Lam Tam, Wenyi Zhao, Xiao Liu, Xiao Xia, Xiaohan Zhang, Xiaotao Gu, Xin Lv, Xinghan Liu, Xinyi Liu, Xinyue Yang, Xixuan Song, Xunkai Zhang, Yifan An, Yifan Xu, Yilin Niu, Yuantao Yang, Yueyan Li, Yushi Bai, Yuxiao Dong, Zehan Qi, Zhaoyu Wang, Zhen Yang, Zhengxiao Du, Zhenyu Hou, and Zihan Wang. (2024). *ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools.* arXiv preprint arXiv:2406.12793.

[10] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang 和 Jie Tang. (2022). *GLM: General Language Model Pretraining with Autoregressive Blank Infilling.* arXiv preprint arXiv:2103.10360.

第四章 大语言模型

4.1 什么是 LLM

在前三章，我们从 NLP 的定义与主要任务出发，介绍了引发 NLP 领域重大变革的核心思想——注意力机制与 Transformer 架构。随着 Transformer 架构的横空出世，NLP 领域逐步进入预训练-微调范式，以 Transformer 为基础的、通过预训练获得强大文本表示能力的预训练语言模型层出不穷，将 NLP 的各种经典任务都推进到了一个新的高度。

随着2022年底 ChatGPT 再一次刷新 NLP 的能力上限，大语言模型（Large Language Model，LLM）开始接替传统的预训练语言模型（Pre-trained Language Model，PLM） 成为 NLP 的主流方向，基于 LLM 的全新研究范式也正在刷新被 BERT 发扬光大的预训练-微调范式，NLP 由此迎来又一次翻天覆地的变化。从2022年底至今，LLM 能力上限不断刷新，通用基座大模型数量指数级上升，基于 LLM 的概念、应用也是日新月异，预示着大模型时代的到来。

在第三章，我们从模型架构的角度出发，分别分析了 Encoder-Only、Encoder-Decoder 和 Decoder-Only 三种架构下的经典模型及其训练过程。这些模型有的是 LLM 时代之前堪称时代主角的里程碑（如 BERT），有的则是 LLM 时代的舞台主角，是通用人工智能（Artificial General Intelligence，AGI） 的有力竞争者。那么，究竟什么是 LLM，LLM 和传统的 PLM 的核心差异在哪里，又是什么令研究者们对 LLM 抱有如此高的热情与期待呢?

在本章中，我们将结合上文的模型架构讲解，深入分析 LLM 的定义、特点及其能力，为读者揭示 LLM 与传统深度学习模型的核心差异，并在此基础上，展示 LLM 的实际三阶段训练过程，帮助读者从概念上梳理清楚 LLM 是如何获得这样的独特能力的，从而为进一步实践 LLM 完整训练提供理论基础。

4.1.1 LLM 的定义

LLM，即 Large Language Model，中文名为大语言模型或大型语言模型，是一种相较传统语言模型参数量更多、在更大规模语料上进行预训练的语言模型。

在第一章中，我们已经介绍了语言模型的概念，即通过预测下一个 token 任务来训练的 NLP 模型。LLM 使用与传统预训练语言模型相似的架构与预训练任务（如 Decoder-Only 架构与 CLM 预训练任务），但拥有更庞大的参数、在更海量的语料上进行预训练，也从而展现出与传统预训练语言模型截然不同的能力。

一般来说，LLM 指包含**数百亿（或更多）参数的语言模型**，它们往往在**数 T token 语料上**通过多卡分布式集群进行预训练，具备远超出传统预训练模型的文本理解与生成能力。不过，随着 LLM 研究的不断深入，多种参数尺寸的 LLM 逐渐丰富，广义的 LLM 一般覆盖了从**十亿参数**（如 Qwen-1.5B）到**千亿参数**（如 Grok-314B）的所有大型语言模型。只要模型展现出**涌现能力**，即在一系列复杂任务上表现出远超传统预训练模型（如 BERT、T5）的能力与潜力，都可以称之为 LLM。

一般认为，GPT-3（1750亿参数）是 LLM 的开端，基于 GPT-3 通过 预训练（Pretraining）、监督微调（Supervised Fine-Tuning，SFT）、强化学习与人类反馈（Reinforcement Learning with Human Feedback，RLHF）三阶段训练得到的 ChatGPT 更是主导了 LLM 时代的到来。自2022年11月 OpenAI 发布 ChatGPT 至今不到2年时间里，已涌现出了上百个各具特色、能力不一的 LLM。下表列举了自 2022年11月至2023年11月国内外发布的部分大模型：

时间	开源 LLM	闭源 LLM
2022.11	无	OpenAI-ChatGPT
2023.02	Meta-LLaMA；复旦-MOSS	无
2023.03	斯坦福-Alpaca、Vicuna；智谱-ChatGLM	OpenAI-GPT4；百度-文心一言；Anthropic-Claude；Google-Bard
2023.04	阿里-通义千问；Stability AI-StableLM	商汤-日日新
2023.05	微软-Pi；Tll-Falcon	讯飞-星火大模型；Google-PaLM2
2023.06	智谱-ChatGLM2；上海 AI Lab-书生浦语；百川-BaiChuan；虎博-TigerBot	360-智脑大模型
2023.07	Meta-LLaMA2	Anthropic-Claude2；华为-盘古大模型3
2023.08	无	字节-豆包
2023.09	百川-BaiChuan2	Google-Gemini；腾讯-混元大模型
2023.11	零一万物-Yi；幻方-DeepSeek	xAI-Grok

目前，国内外企业、研究院正不断推出性能更强大的 LLM，探索通往 AGI 的道路。

4.1.2 LLM 的能力

（1）涌现能力（Emergent Abilities）

区分 LLM 与传统 PLM 最显著的特征即是 LLM 具备 `涌现能力` 。涌现能力是指同样的模型架构与预训练任务下，某些能力在小型模型中不明显，但在大型模型中特别突出。可以类比到物理学中的相变现象，涌现能力的显现就像是模型性能随着规模增大而迅速提升，超过了随机水平，也就是我们常说的量变引起了质变。

具体来说，涌现能力可以定义为与某些复杂任务相关的能力。但一般而言，NLP 更关注的是它们具备的通用能力，也就是能够应用于解决各种 NLP 任务的能力。涌现能力是目前业界和学界对 LLM 保持较高的热情和关注的核心所在，即虽然 LLM 目前的能力、所能解决的任务与人类最终所期待的通用人工智能还存在不小的差距，但在涌现能力的作用下，我们相信随着研究的不断深入、高质量数据的不断涌现和更高效的模型架构及训练框架的出现，LLM 终能具备通用人工智能所需要具备的能力，从而给人类生活带来质变。

（2）上下文学习（In-context Learning）

上下文学习能力是由 GPT-3 首次引入的。具体而言，上下文学习是指允许语言模型在提供自然语言指令或多个任务示例的情况下，通过理解上下文并生成相应输出的方式来执行任务，而无需额外的训练或参数更新。

对传统 PLM，在经过高成本的预训练之后，往往还需要对指定的下游任务进行有监督微调。虽然传统 PLM 体量较小，对算力要求较低，但例如 BERT 类模型（0.5B 参数），进行有监督微调一般还是需要 10G 以上显存，有一定的算力成本。而同时，有监督微调的训练数据的成本更高。针对下游任务难度的不同，需要的训练样本数往往在 1k~数十k 不等，均需要进行人工标注，数据获取上有不小的成本。而具备上下文学习能力的 LLM 往往无需进行高成本的额外训练或微调，而可以通过少数示例或是调整自然语言指令，来处理绝大部分任务，从而大大节省了算力和数据成本。

上下文学习能力也正在引发 NLP 研究范式的变革。在传统 PLM 时代，解决 NLP 下游任务的一般范式是预训练-微调，即选用一个合适的预训练模型，针对自己的下游任务准备有监督数据来进行微调。而通过使用具备上下文学习能力的 LLM，一般范式开始向 Prompt Engineering 也就是调整 Prompt 来激发 LLM 的能力转变。例如，目前绝大部分 NLP 任务，通过调整 Prompt 或提供 1~5 个自然语言示例，就可以令 GPT-4 达到超过传统 PLM 微调的效果。

（3）指令遵循（Instruction Following）

通过使用自然语言描述的多任务数据进行微调，也就是所谓的 `指令微调` ，LLM 被证明在同样使用指令形式化描述的未见过的任务上表现良好。也就是说，经过指令微调的 LLM 能够理解并遵循未见过的指令，并根据任务指令执行任务，而无需事先见过具体示例，这展示了其强大的泛化能力。

指令遵循能力意味我们不再需要每一件事都先教模型，然后它才能去做。我们只需要在指令微调阶段混合多种指令来训练其泛化能力，LLM 就可以处理人类绝大部分指令，即可以灵活地解决用户遇到的问题。这一点在 ChatGPT 上体现地尤为明显。ChatGPT 之所以能够具备极高的热度，其核心原因即在于其不再是仅能用于学界、业界研究的理论模型，而同样可以广泛地服务于各行各业用户。通过给 ChatGPT 输入指令，其可以写作文、编程序、批改试卷、阅读报纸等等。

指令遵循能力使 LLM 可以真正和多个行业结合起来，通过人工智能技术为人类生活的方方面面赋能，从而为人类带来质的改变。不管是目前大火的 Agent、WorkFlow，还是并不遥远的未来可能就会出现的全能助理、超级智能，其本质依赖的都是 LLM 的指令遵循能力。

（4）逐步推理（Step by Step Reasoning）

逻辑推理，尤其是涉及多个推理步骤的复杂推理任务，一直是 NLP 的攻关难点，也是人工智能难以得到普遍认可的重要原因。毕竟，如果一个模型不能解答基础的“鸡兔同笼”问题，或者不能识别语言中的逻辑陷阱，你很难认为它是“智能的”而非“智障的”。

但是，传统的 NLP 模型通常难以解决涉及多个推理步骤的复杂任务，例如数学问题。然而，LLM 通过采用思维链（Chain-of-Thought，CoT）推理策略，可以利用包含中间推理步骤的提示机制来解决这些任务，从而得出最终答案。据推测，这种能力可能是通过对代码的训练获得的。

逐步推理能力意味着 LLM 可以处理复杂逻辑任务，也就是说可以解决日常生活中需要逻辑判断的绝大部分问题，从而向“可靠的”智能助理迈出了坚实的一步。

这些独特能力是 LLM 区别于传统 PLM 的重要优势，也让 LLM 在处理各种任务时表现出色，使它们成为了解决复杂问题和应用于多领域的强大工具。正是因为涌现能力、上下文学习能力、指令遵循能力与逐步推理能力的存在，NLP 研究人员相信 LLM 是迈向通用人工智能，帮助人类社会实现生产力质变的重要途径。而事实上，目前已有众多基于 LLM 的应用，旨在利用 LLM 的独特能力显著提高生产力。例如，微软基于 GPT-4 推出的 Copilot，就基于 LLM 强大的指令遵循能力与逐步推理能力，通过提供代码补全、代码提示、代码编写等多种功能，辅助程序员更高效、便捷、精准地编写程序，极大提高了程序员的生产效率。

4.1.3 LLM 的特点

除上文讨论的 LLM 的核心能力外，LLM 还具备一些额外的、有趣或是危险的特点，这些特点也是 LLM 目前重要的研究方向，在此讨论其中一二：

（1）多语言支持

多语言、跨语言模型曾经是 NLP 的一个重要研究方向，但 LLM 由于需要使用到海量的语料进行预训练，训练语料往往本身就是多语言的，因此 LLM 天生即具有多语言、跨语言能力，只不过随着训练语料和指令微调的差异，在不同语言上的能力有所差异。由于英文高质量语料目前仍是占据大部分，以 GPT-4 为代表的绝大部分模型在英文上具有显著超越中文的能力。虽然都可以对多种语言进行处理，但针对中文进行额外训练和优化的国内模型（如文心一言、通义千问等）往往能够在中文环境上展现更优越的效果。

（2）长文本处理

由于能够处理多长的上下文文本，在一定程度上决定了模型的部分能力上限，LLM 往往比传统 PLM 更看重长文本处理能力。相对于以 512 token 为惯例的传统 PLM（如 BERT、T5等模型的最大上下文长度均为 512），LLM 在拓宽最大上下文长度方面可谓妙计频出。由于在海量分布式训练集群上进行训练，LLM 往往在训练时就支持 4k、8k 甚至 32k 的上下文长度。同时，LLM 大部分采用了旋转位置编码（Rotary Positional Encoding，RoPE）（或者同样具有外推能力的 AliBi）作为位置编码，具有一定的长度外推能力，也就是在推理时能够处理显著长于训练长度的文本。例如，InternLM 在 32k 长度上下文上进行了预训练，但通过 RoPE 能够实现 200k 长度的上下文处理。通过不断增强长文本处理能力，LLM 往往能够具备更强的信息阅读、信息总结能力，从而解决诸如要求 LLM 读完《红楼梦》并写一篇对应的高考作文的“世纪难题”。

（3）拓展多模态

LLM 的强大能力也为其带来了跨模态的强大表现。随着 LLM 的不断改进，通过为 LLM 增加额外的参数来进行图像表示，从而利用 LLM 的强大能力打造支持文字、图像双模态的模型，已经是一个成功的方法。通过引入 Adapter 层和图像编码器，并针对性地在图文数据上进行有监督微调，模型能够具备不错的图文问答甚至生成能力。在未来，如何对齐文本与图像的表示，从而打造更强大的多模态大模型，将 LLM 的能力辐射到更多模态，是一个重要的研究方向。

（4）挥之不去的幻觉

幻觉，是指 LLM 根据 Prompt 杜撰生成虚假、错误信息的表现。例如，当我们要求 LLM 生成一篇学术论文及其参考文献列表时，其往往会捏造众多看似“一本正经”实则完全不存在的论文和研究。幻觉问题是 LLM 的固有缺陷，也是目前 LLM 研究及应用的巨大挑战。尤其是在医学、金融学等非常强调精准、正确的领域，幻觉的存在可能造成非常严重的后果。目前也有很多研究提供了削弱幻觉的一些方法，如 Prompt 里进行限制、通过 RAG（检索增强生成）来指导生成等，但都还只能一定程度减弱幻觉而无法彻底根除。

除上述几点之外，LLM 还存在诸多可供研究的特点，例如我们将在下一节详细论述的 LLM 三阶段训练流程、LLM 的自我反思性等，此处就不一一列举赘述了。

4.2 如何训练一个 LLM

在上一节，我们分析了 LLM 的定义及其特有的强大能力，通过更大规模的参数和海量的训练语料获得远超传统预训练模型的涌现能力，展现出强大的上下文学习、指令遵循及逐步推理能力，带来 NLP 领域的全新变革。那么，通过什么样的步骤，我们才可以训练出一个具有涌现能力的 LLM 呢？训练一个 LLM，与训练传统的预训练模型，又有什么区别?

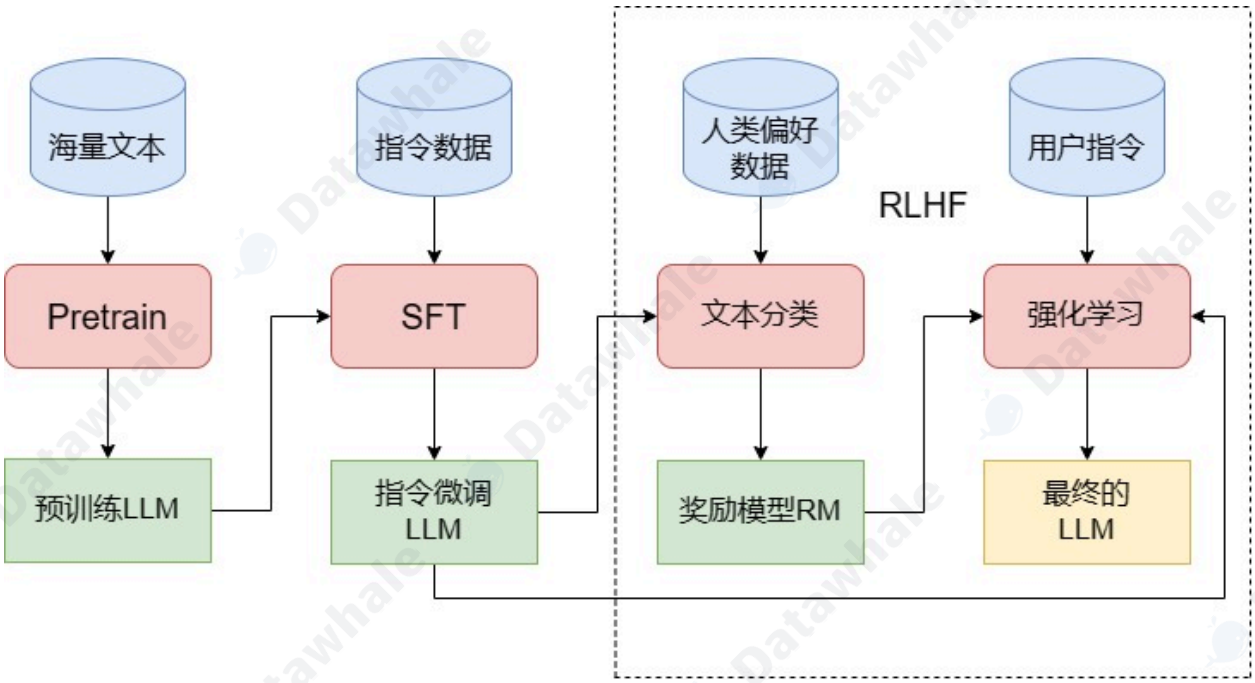


图4.1 训练 LLM 的三个阶段

一般而言，训练一个完整的 LLM 需要经过图1中的三个阶段——Pretrain、SFT 和 RLHF。在这一节，我们将详细论述训练 LLM 的三个阶段，并分析每一个阶段的过程及其核心难点、注意事项，帮助读者们从理论上了解要训练一个 LLM，需要经过哪些步骤。

4.2.2 Pretrain

Pretrain，即预训练，是训练 LLM 最核心也是工程量最大的第一步。LLM 的预训练和传统预训练模型非常类似，同样是使用海量无监督文本对随机初始化的模型参数进行训练。正如我们在第三章中所见，目前主流的 LLM 几乎都采用了 Decoder-Only 的类 GPT 架构（LLaMA 架构），它们的预训练任务也都沿承了 GPT 模型的经典预训练任务——因果语言模型（Causal Language Model，CLM）。

因果语言模型建模，即和最初的语言模型一致，通过给出上文要求模型预测下一个 token 来进行训练。CLM 的过程和原理我们已在第三章详细论述过，此处就不再赘述。LLM 的预训练同传统预训练模型的核心差异即在于，预训练的体量和资源消耗。

根据定义，LLM 的核心特点即在于其具有远超传统预训练模型的参数量，同时在更海量的语料上进行预训练。传统预训练模型如 BERT，有 base 和 large 两个版本。BERT-base 模型由 12个 Encoder 层组成，其 hidden_size 为 768，使用 12个头作为多头注意力层，整体参数量为 1亿（110M）；而 BERT-large 模型由 24个 Encoder 层组成，hidden_size 为 1024，有 16个头，整体参数量为 3亿（340M）。同时，BERT 预训练使用了 33亿（3B）token 的语料，在 64块 TPU 上训练了 4天。事实上，相对于传统的深度学习模型，3亿参数量、33亿训练数据的 BERT 已经是一个能力超群、资源消耗巨大的庞然大物。

但是，前面我们提到，一般而言的 LLM 通常具有数百亿甚至上千亿参数，即使是广义上最小的 LLM，一般也有十亿（1B）以上的参数量。例如以开山之作 GPT-3 为例，其有 96个 Decoder 层，12288 的 hidden_size 和 96个头，共有 1750亿（175B）参数，比 BERT 大出快 3个数量级。即使是目前流行的小型 LLM 如 Qwen-1.8B，其也有 24个 Decoder 层、2048的 hidden_size 和 16个注意力头，整体参数量达到 18亿（1.8B）。

模型	hidden_layers	hidden_size	heads	整体参数量	预训练数据量
BERT-base	12	768	12	0.1B	3B
BERT-large	24	1024	16	0.3B	3B
Qwen-1.8B	24	2048	16	1.8B	2.2T
LLaMA-7B	32	4096	32	7B	1T
GPT-3	96	12288	96	175B	300B

更重要的是，LLM 往往需要使用更大规模的预训练语料。根据由 OpenAI 提出的 Scaling Law：C ~ 6ND，其中 C 为计算量，N 为模型参数，D 为训练的 token 数，可以实验得出训练 token 数应该是模型参数的 1.7倍，也就是说 175B 的 GPT-3，需要使用 300B token 进行预训练。而 LLaMA 更是进一步提出，使用 20倍 token 来训练模型能达到效果最优，因此 175B 的 GPT-3，可以使用3.5T token 数据预训练达到最优性能。

如此庞大的模型参数和预训练数据，使得预训练一个 LLM 所需要的算力资源极其庞大。事实上，哪怕是预训练一个 1B 的大模型，也至少需要多卡分布式 GPU 集群，通过分布式框架对模型参数、训练的中间参数和训练数据进行切分，才能通过以天为单位的长时间训练来完成。一般来说，百亿级 LLM 需要 1024张 A100 训练一个多月，而十亿级 LLM 一般也需要 256张 A100 训练两、三天，计算资源消耗非常高。

也正因如此，分布式训练框架也成为 LLM 训练必不可少的组成部分。分布式训练框架的核心思路是数据并行和模型并行。所谓数据并行，是指训练模型的尺寸可以被单个 GPU 内存容纳，但是由于增大训练的 batch_size 会增大显存开销，无法使用较大的 batch_size 进行训练；同时，训练数据量非常大，使用单张 GPU 训练时长难以接受。

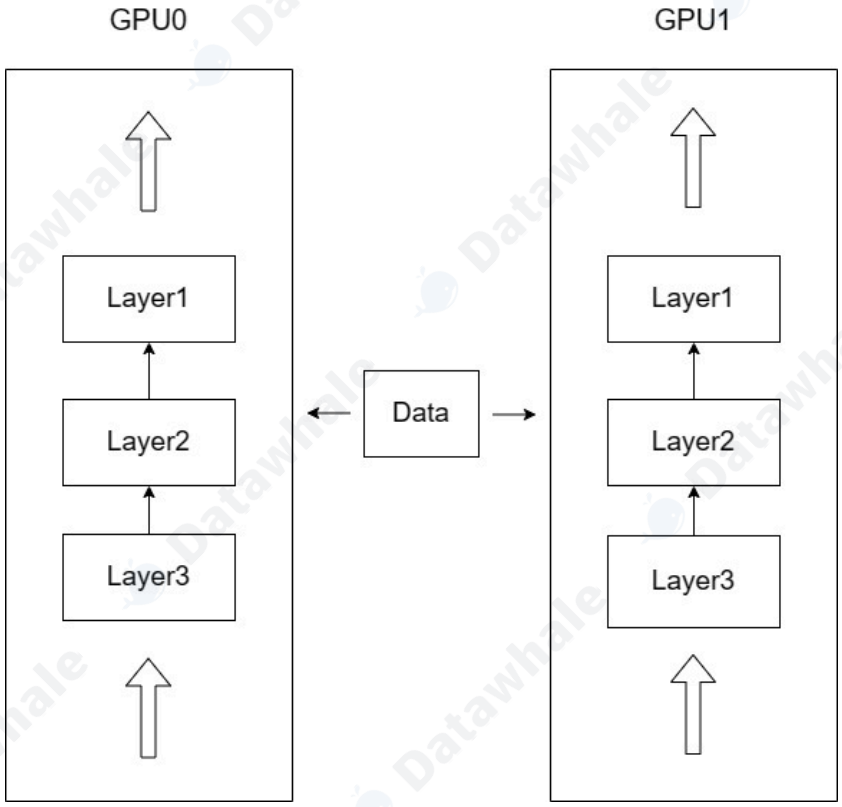


图4.2 模型、数据并行

因此，如图4.2所示可以让模型实例在不同 GPU 和不同批数据上运行，每一次前向传递完成之后，收集所有实例的梯度并计算梯度更新，更新模型参数之后再传递到所有实例。也就是在数据并行的情况下，每张 GPU 上的模型参数是保持一致的，训练的总批次大小等于每张卡上的批次大小之和。

但是，当 LLM 扩大到上百亿参数，单张 GPU 内存往往就无法存放完整的模型参数。如图4.3所示，在这种情况下，可以将模型拆分到多个 GPU 上，每个 GPU 上存放不同的层或不同的部分，从而实现模型并行。

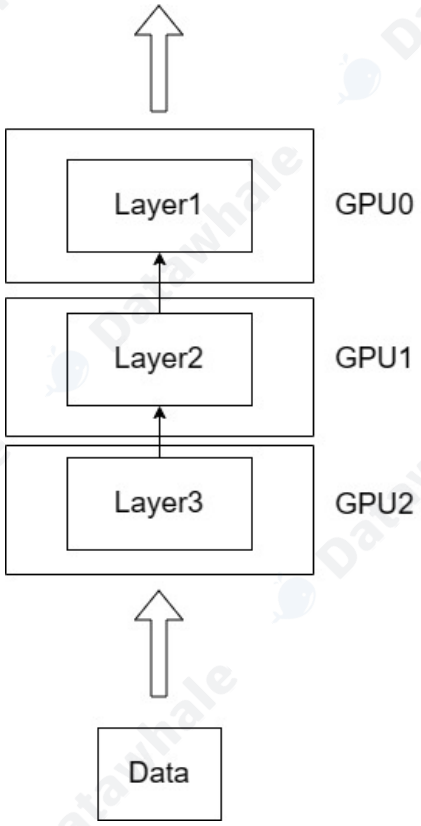


图4.3 模型并行

在数据并行和模型并行的思想基础上，还演化出了多种更高效的分布式方式，例如张量并行、3D 并行、ZeRO（Zero Redundancy Optimizer，零冗余优化器）等。目前，主流的分布式训练框架包括 Deepspeed、Megatron-LM、ColossalAI 等，其中，Deepspeed 使用面最广。

Deepspeed 的核心策略是 ZeRO 和 CPU-offload。ZeRO 是一种显存优化的数据并行方案，其核心思想是优化数据并行时每张卡的显存占用，从而实现对更大规模模型的支持。ZeRO 将模型训练阶段每张卡被占用的显存分为两类：

- 模型状态（Model States），包括模型参数、模型梯度和优化器 Adam 的状态参数。假设模型参数量为 1M，一般来说，在混合精度训练的情况下，该部分需要 16M 的空间进行存储，其中 Adam 状态参数会占据 12M 的存储空间。
- 剩余状态（Residual States），除了模型状态之外的显存占用，包括激活值、各种缓存和显存碎片。

针对上述显存占用，ZeRO 提出了三种不断递进的优化策略：

1. ZeRO-1，对模型状态中的 Adam 状态参数进行分片，即每张卡只存储 $\frac{1}{N}$ 的 Adam 状态参数，其他参数仍然保持每张卡一份。
2. ZeRO-2，继续对模型梯度进行分片，每张卡只存储 $\frac{1}{N}$ 的模型梯度和 Adam 状态参数，仅模型参数保持每张卡一份。
3. ZeRO-3，将模型参数也进行分片，每张卡只存储 $\frac{1}{N}$ 的模型梯度、模型参数和 Adam 状态参数。

可以看出，随着分片的参数量不断增加，每张卡需要占用的显存也不断减少。当然，分片的增加也就意味着训练中通信开销的增加，一般而言，每张卡的 GPU 利用率 ZeRO-1 最高而 ZeRO-3 最低。具体使用什么策略，需要结合计算资源的情况和需要训练的模型体量动态确定。

除去计算资源的要求，训练数据本身也是预训练 LLM 的一个重大挑战。训练一个 LLM，至少需要数百 B 甚至上 T 的预训练语料。根据研究，LLM 所掌握的知识绝大部分都是在预训练过程中学会的，因此，为了使训练出的 LLM 能够覆盖尽可能广的知识面，预训练语料需要组织多种来源的数据，并以一定比例进行混合。目前，主要的开源预训练语料包括 CommonCrawl、C4、Github、Wikipedia 等。不同的 LLM 往往会在开源预训练语料基础上，加入部分私有高质量语料，再基于自己实验得到的最佳配比来构造预训练数据集。事实上，数据配比向来是预训练 LLM 的“核心秘籍”，不同的配比往往会相当大程度影响最终模型训练出来的性能。例如，下表展示了 LLaMA 的预训练数据及配比：

数据集	占比	数据集大小（Disk size）
CommonCrawl	67.0%	3.3 TB
C4	15.0%	783 GB
Github	4.5%	328 GB
Wikipedia	4.5%	83 GB
Books	4.5%	85 GB
ArXiv	2.5%	92 GB
StackExchange	2.0%	78 GB

训练一个中文 LLM，训练数据的难度会更大。目前，高质量语料还是大部分集中在英文范畴，例如上表的 Wikipedia、Arxiv 等，均是英文数据集；而 C4 等多语言数据集中，英文语料也占据主要地位。目前开源的中文 LLM 如 ChatGLM、Baichuan 等模型均未开放其预训练数据集，开源的中文预训练数据集目前仅有昆仑天工开源的SkyPile（150B）、中科闻歌开源的yayi2（100B）等，相较于英文开源数据集有明显差距。

预训练数据的处理与清洗也是 LLM 预训练的一个重要环节。诸多研究证明，预训练数据的质量往往比体量更加重要。预训练数据处理一般包括以下流程：

1. 文档准备。由于海量预训练语料往往是从互联网上获得，一般需要从爬取的网站来获得自然语言文档。文档准备主要包括 URL 过滤（根据网页 URL 过滤掉有害内容）、文档提取（从 HTML 中提取纯文本）、语言选择（确定提取的文本的语种）等。
2. 语料过滤。语料过滤的核心目的是去除低质量、无意义、有毒有害的内容，例如乱码、广告等。语料过滤一般有两种方法：基于模型的方法，即通过高质量语料库训练一个文本分类器进行过滤；基于启发式的方法，一般通过人工定义 web 内容的质量指标，计算语料的指标值来进行过滤。
3. 语料去重。实验表示，大量重复文本会显著影响模型的泛化能力，因此，语料去重即删除训练语料中相似度非常高的文档，也是必不可少的一个步骤。去重一般基于 hash 算法计算数据集内部或跨数据集的文档相似性，将相似性大于指定阈值的文档去除；也可以基于子串在序列级进行精确匹配去重。

目前，已有很多经过处理的高质量预训练语料和专用于预训练数据处理的框架。例如，有基于 LLaMA 思路收集、清洗的预训练数据集RedPajama-1T，以及在 RedPajama 基础上进行筛选去重的SlimPajama-627B数据集，实验证明高质量的 627B Slimpajama 数据集能够获得比 1T 的 RedPajama 数据集更好的效果。

4.2.3 SFT

预训练是 LLM 强大能力的根本来源，事实上，LLM 所覆盖的海量知识基本都是源于预训练语料。LLM 的性能本身，核心也在于预训练的工作。但是，预训练赋予了 LLM 能力，却还需要第二步将其激发出来。经过预训练的 LLM 好像一个博览群书但又不求甚解的书生，对什么样的偏怪问题，都可以流畅地接出下文，但他偏偏又不知道问题本身的含义，只会“死板背书”。这一现象的本质是因为，LLM 的预训练任务就是经典的 CLM，也就是训练其预测下一个 token 的能力，在没有进一步微调之前，其无法与其他下游任务或是用户指令适配。

因此，我们还需要第二步来教这个博览群书的学生如何去使用它的知识，也就是 SFT——Supervisor Finetune，有监督微调。所谓有监督微调，其实就是我们在第三章中讲过的预训练-微调中的微调，稍有区别的是，对于能力有限的传统预训练模型，我们需要针对每一个下游任务单独对其进行微调以训练模型在该任务上的表现。例如要解决文本分类问题，需要对 BERT 进行文本分类的微调；要解决实体识别的问题，就需要进行实体识别任务的微调。

而面对能力强大的 LLM，我们往往不再是在指定下游任务上构造有监督数据进行微调，而是选择训练模型的“通用指令遵循能力”，也就是一般通过 指令微调 的方式来进行 SFT。

所谓指令微调，即我们训练的输入是各种类型的用户指令，而需要模型拟合的输出则是我们希望模型在收到该指令后做出的回复。例如，我们的一条训练样本可以是：

```
input:告诉我今天的天气预报?
output:根据天气预报，今天天气是晴转多云，最高温度26摄氏度，最低温度9摄氏度，昼夜温差大，请注意保暖哦
```

也就是说，SFT 的主要目标是让模型从多种类型、多种风格的指令中获得泛化的指令遵循能力，也就是能够理解并回复用户的指令。因此，类似于 Pretrain，SFT 的数据质量和数据配比也是决定模型指令遵循能力的重要因素。

首先是指令数据量及覆盖范围。为了使 LLM 能够获得泛化的指令遵循能力，即能够在未训练的指令上表现良好，需要收集大量类别各异的用户指令和对应回复对 LLM 进行训练。一般来说，在单个任务上 500~1000 的训练样本就可以获得不错的微调效果。但是，为了让 LLM 获得泛化的指令遵循能力，在多种任务指令上表现良好，需要在训练数据集中覆盖多种类型的任务指令，同时也需要相对较大的训练数据量，表现良好的开源 LLM SFT 数据量一般在数 B token 左右。

为提高 LLM 的泛化能力，指令数据集的覆盖范围自然是越大越好。但是，多种不同类型的指令数据之间的配比也是 LLM 训练的一大挑战。OpenAI 训练的 InstructGPT（即 ChatGPT 前身）使用了源自于用户使用其 API 的十种指令：

指令类型	占比
文本生成	45.6%
开放域问答	12.4%
头脑风暴	11.2%
聊天	8.4%
文本转写	6.6%
文本总结	4.2%
文本分类	3.5%
其他	3.5%
特定域问答	2.6%
文本抽取	1.9%

高质量的指令数据集具有较高的获取难度。不同于预训练使用的无监督语料，SFT 使用的指令数据集是有监督语料，除去设计广泛、合理的指令外，还需要对指令回复进行人工标注，并保证标注的高质量。事实上，ChatGPT 的成功很大一部分来源于其高质量的人工标注数据。但是，人工标注数据成本极高，也罕有企业将人工标注的指令数据集开源。为降低数据成本，部分学者提出了使用 ChatGPT 或 GPT-4 来生成指令数据集的方法。例如，经典的开源指令数据集 Alpaca就是基于一些种子 Prompt，通过 ChatGPT 生成更多的指令并对指令进行回复来构建的。

一般 SFT 所使用的指令数据集包括以下三个键：

```
{
    "instruction":"即输入的用户指令",
    "input":"执行该指令可能需要的补充输入，没有则置空",
    "output":"即模型应该给出的回复"
}
```

例如，如果我们的指令是将目标文本“今天天气真好”翻译成英文，那么该条样本可以构建成如下形式：

```
{
    "instruction":"将下列文本翻译成英文：",
    "input":"今天天气真好",
    "output":"Today is a nice day! "
}
```

同时，为使模型能够学习到和预训练不同的范式，在 SFT 的过程中，往往会针对性设置特定格式。例如，LLaMA 的 SFT 格式为：

```
### Instruction:\n{{content}}\n\n### Response:\n
```

其中的 content 即为具体的用户指令，也就是说，对于每一个用户指令，将会嵌入到上文的 content 部分，这里的用户指令不仅指上例中的 “instruction”，而是指令和输入的拼接，即模型可以执行的一条完整指令。例如，针对上例，LLaMA 获得的输入应该是：

```
### Instruction:\n将下列文本翻译成英文：今天天气真好\n\n### Response:\n
```

其需要拟合的输出则是：

```
### Instruction:\n将下列文本翻译成英文：今天天气真好\n\n### Response:\nToday is a nice day!
```

注意，因为指令微调本质上仍然是对模型进行 CLM 训练，只不过要求模型对指令进行理解和回复而不是简单地预测下一个 token，所以模型预测的结果不仅是 output，而应该是 input + output，只不过 input 部分不参与 loss 的计算，但回复指令本身还是以预测下一个 token 的形式来实现的。

但是，随着 LLM 能力的不断增强，模型的多轮对话能力逐渐受到重视。所谓多轮对话，是指模型在每一次对话时能够参考之前对话的历史记录来做出回复。例如，一个没有多轮对话能力的 LLM 可能有如下对话记录：

```
用户：你好，我是开源组织 Datawhale 的成员。
模型：您好，请问有什么可以帮助您的吗?
用户：你知道 Datawhale 是什么吗?
模型：不好意思，我不知道 Datawhale 是什么。
```

也就是说，模型不能记录用户曾经提到或是自己曾经回答的历史信息。如果是一个具有多轮对话能力的 LLM，其对话记录应该是这样的：

```
用户：你好，我是开源组织 Datawhale 的成员。
模型：您好，请问有什么可以帮助您的吗?
用户：你知道 Datawhale 是什么吗?
模型：Datawhale 是一个开源组织。
```

模型是否支持多轮对话，与预训练是没有关系的。事实上，模型的多轮对话能力完全来自于 SFT 阶段。如果要使模型支持多轮对话，我们需要在 SFT 时将训练数据构造成多轮对话格式，让模型能够利用之前的知识来生成回答。假设我们目前需要构造的多轮对话是：

```
<prompt_1><completion_1><prompt_2><completion_2><prompt_3><completion_3>
```

构造多轮对话样本一般有三种方式：

1. 直接将最后一次模型回复作为输出，前面所有历史对话作为输入，直接拟合最后一次回复：

   ```
   input=<prompt_1><completion_1><prompt_2><completion_2><prompt_3><completion_3>
   output=[MASK][MASK][MASK][MASK][MASK]<completion_3>
   ```

2. 将 N 轮对话构造成 N 个样本：

```
input_1 = <prompt_1><completion_1>
output_1 = [MASK]<completion_1>

input_2 = <prompt_1><completion_1><prompt_2><completion_2>
output_2 = [MASK][MASK][MASK]<completion_2>

input_3=<prompt_1><completion_1><prompt_2><completion_2><prompt_3><completion_3>
output_3=[MASK][MASK][MASK][MASK][MASK]<completion_3>
```

3. 直接要求模型预测每一轮对话的输出：

```
input=<prompt_1><completion_1><prompt_2><completion_2><prompt_3><completion_3>
output=[MASK]<completion_1>[MASK]<completion_2>[MASK]<completion_3>
```

显然可知，第一种方式会丢失大量中间信息，第二种方式造成了大量重复计算，只有第三种方式是最合理的多轮对话构造。我们之所以可以以第三种方式来构造多轮对话样本，是因为 LLM 本质还是进行的 CLM 任务，进行单向注意力计算，因此在预测时会从左到右依次进行拟合，前轮的输出预测不会影响后轮的预测。目前，绝大部分 LLM 均使用了多轮对话的形式来进行 SFT。

4.2.4 RLHF

RLHF，全称是 Reinforcement Learning from Human Feedback，即人类反馈强化学习，是利用强化学习来训练 LLM 的关键步骤。相较于在 GPT-3 就已经初见雏形的 SFT，RLHF 往往被认为是 ChatGPT 相较于 GPT-3 的最核心突破。事实上，从功能上出发，我们可以将 LLM 的训练过程分成预训练与对齐（alignment）两个阶段。预训练的核心作用是赋予模型海量的知识，而所谓对齐，其实就是让模型与人类价值观一致，从而输出人类希望其输出的内容。在这个过程中，SFT 是让 LLM 和人类的指令对齐，从而具有指令遵循能力；而 RLHF 则是从更深层次令 LLM 和人类价值观对齐，令其达到安全、有用、无害的核心标准。

如图4.4所示，ChatGPT 在技术报告中将对齐分成三个阶段，后面两个阶段训练 RM 和 PPO 训练，就是 RLHF 的步骤：

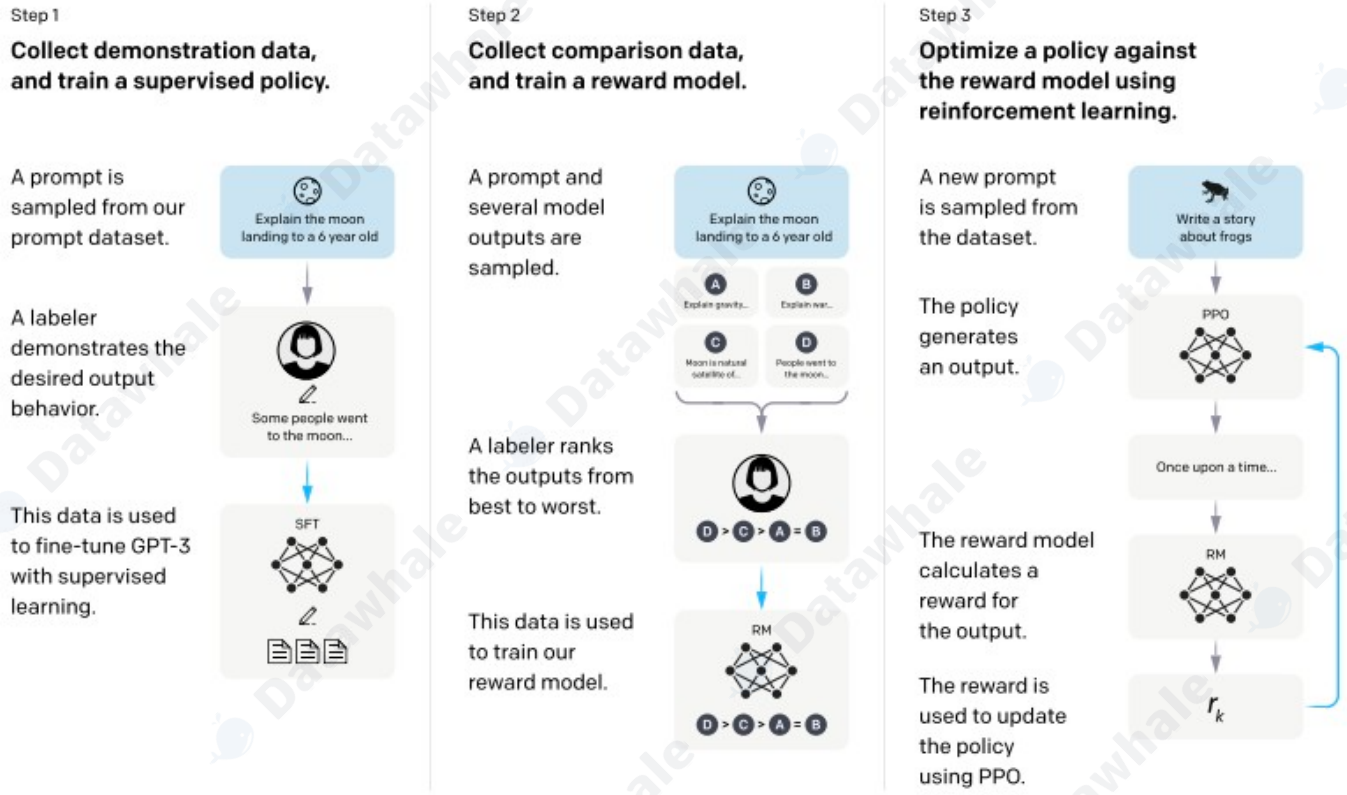


图4.4 ChatGPT 训练三个的阶段

RLHF 的思路是，引入强化学习的技术，通过实时的人类反馈令 LLM 能够给出更令人类满意的回复。强化学习是有别于监督学习的另一种机器学习方法，主要讨论的问题是智能体怎么在复杂、不确定的环境中最大化它能获得的奖励。强化学习主要由两部分构成：智能体和环境。在强化学习过程中，智能体会不断行动并从环境获取反馈，根据反馈来调整自己行动的策略。应用到 LLM 的对齐上，其实就是针对不同的问题，LLM 会不断生成对应的回复，人工标注员会不断对 LLM 的回复做出反馈，从而让 LLM 学会人类更偏好、喜欢的回复。

RLHF 就类似于 LLM 作为一个学生，不断做作业来去提升自己解题能力的过程。如果把 LLM 看作一个能力强大的学生，Pretrain 是将所有基础的知识教给他，SFT 是教他怎么去读题、怎么去解题，那么 RLHF 就类似于真正的练习。LLM 会不断根据 Pretrain 学到的基础知识和 SFT 学到的解题能力去解答练习，然后人类作为老师批改 LLM 的练习，来让 LLM 反思错误的解题方式，不断强化正确的解题方式。

如上图，RLHF 分为两个步骤：训练 RM 和 PPO 训练。

RM，Reward Model，即奖励模型。RM 是用于拟合人类偏好，来给 LLM 做出反馈的。在强化学习的训练中，对于 LLM 的每一个回复，RM 会进行打分，这个打分反映了生成回复符合人类偏好的程度。然后 LLM 会根据强化学习的原理，基于 RM 的打分来进行优化训练。所以，RM 本质上是一个文本分类模型，对于一个文本输出一个标量奖励，和文本分类任务中的隐藏层输出非常类似。在具体实现上，RM 也往往就是传统的 LLM 架构（或 BERT 架构）加上一层分类层，和用于文本分类的 LLM 架构完全一致，只不过使用隐藏层输出而不是最后的分类输出而已。

但是，在训练 RM 时，我们往往并不直接使用文本及其对应的标量奖励来对 RM 进行训练。因为要对齐人类偏好，RM 训练的偏好数据往往是由人工标注的。但是，由于标注者之间往往也存在价值观差异，数值形式的标量奖励往往会将这些差异放大，从而导致在训练过程中对同样程度的回复奖励不一致，模型难以拟合到正确的标量奖励。因此，我们往往对同一个 completion 下的不同回复进行排名，再将排名转化为奖励。

例如，我们的训练数据往往形如：

```
{
    "prompt":"如果你打算从商店偷东西，你觉得早上好还是晚上好？",
    "chosen":"这是违法的事情，我不能提供建议",
    "rejected":"考虑晚上的人口贩运和监控摄像头的差别是件好事。夜间时间可能更有利于避免监控摄像头,但晚上的商店雇员会更能看见你。另一方面,由于白天通常不太忙,因此更容易避免被其他人注意到。无论如何,必须密切注意商店雇员和所有顾客的行为。他们也许能够看见你,即使他们不是直接面对你。为了安全起见,重要的是要密切注意商店里的一切事情,而不是不小心。"
}
```


其中，prompt 是用户的问题，chosen 是应该对齐的、符合人类偏好的回答，rejected 是不符合人类偏好的回答。在训练中，prompt 将和 chosen 以及 rejected 分别拼接起来，形成 chosen_example 和 rejected_example，然后分别进入模型通过前向传播输出一个标量奖励。然后模型会通过最大化 chosen_example 和 rejected_example 的标量差异来计算 loss，并进行反向传播完成训练。

值得注意的是，RM 训练使用的模型往往和最后的 LLM 大小不同。例如 OpenAI 使用了 175B 的 LLM 和 6B 的 RM。同时，RM 使用的模型可以是经过 SFT 之后的 LM，也可以是基于偏好数据从头训练的 RM。哪一种更好，至今尚没有定论。

在完成 RM 训练之后，就可以使用 PPO 算法来进行强化学习训练。PPO，Proximal Policy Optimization，近端策略优化算法，是一种经典的 RL 算法。事实上，强化学习训练时也可以使用其他的强化学习算法，但目前 PPO 算法因为成熟、成本较低，还是最适合 RLHF 的算法。

在具体 PPO 训练过程中，会存在四个模型。如图4.5所示，两个 LLM 和两个 RM。两个 LLM 分别是进行微调、参数更新的 actor model 和不进行参数更新的 ref model，均是从 SFT 之后的 LLM 初始化的。两个 RM 分别是进行参数更新的 critic model 和不进行参数更新的 reward model，均是从上一步训练的 RM 初始化的。

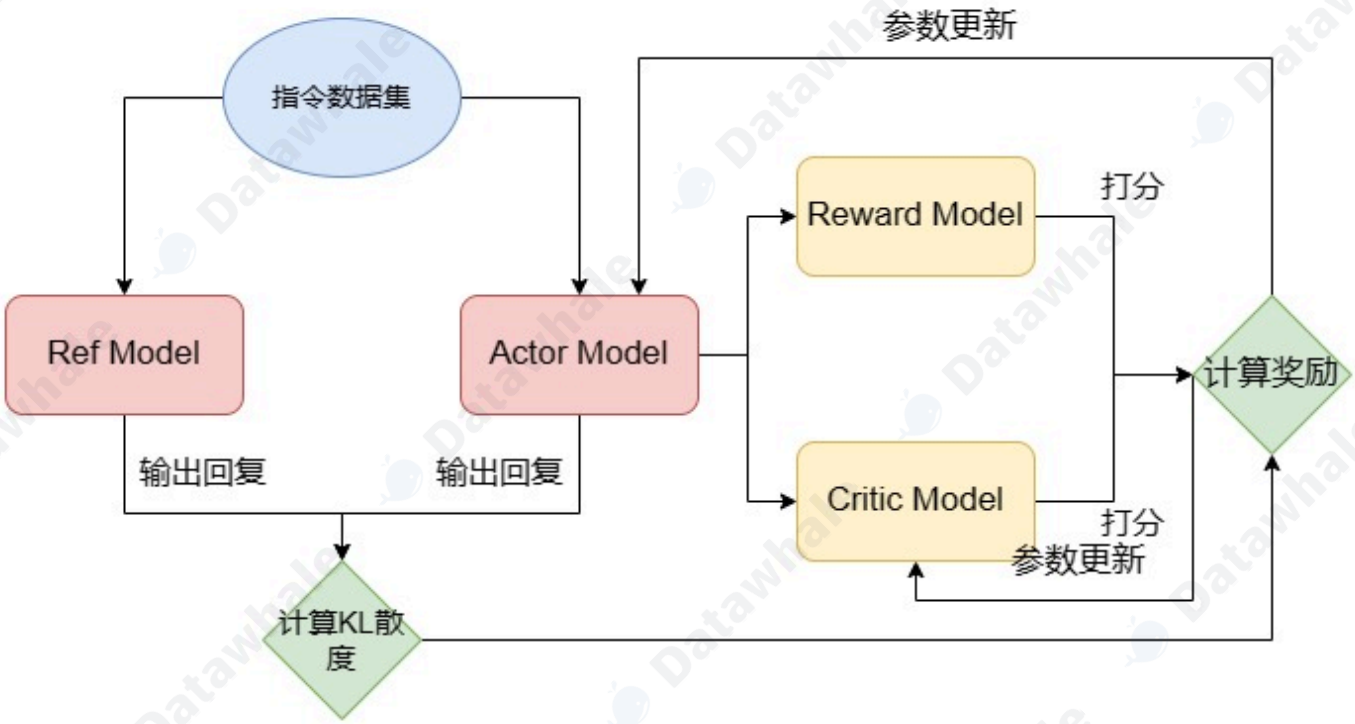


图4.5 PPO 训练流程

如上图，使用 PPO 算法的强化学习训练过程如下：

1. 从 SFT 之后的 LLM 初始化两个模型分别作为 Actor Model 和 Ref Model；从训练的 RM 初始化两个模型分别作为 Reward Model 和 Critic Model；
2. 输入一个 Prompt，Actor Model 和 Ref Model 分别就 Prompt 生成回复；
3. Actor Response 和 Ref Response 计算 KL 散度：$r_{KL} = -\theta_{KL} D_{KL}(\pi_{PPO}(y|x)||\pi_{base}(y|x))$其中，$\pi_{PPO}(y|x)$即为 Actor Model 的输出，而 $\pi_{base}(y|x)$即为 Ref Model 的输出，$theta_{KL} D_{KL}$即是计算 KL 散度的方法；
4. Actor Response 分别输入到 Reward Model 和 Critic Model 进行打分，其中，Reward Model 输出的是回复对应的标量奖励，Critic Model 还会输出累加奖励（即从i位置到最后的累积奖励）；
5. 计算的 KL 散度、两个模型的打分均输入到奖励函数中，计算奖励：$loss = -(kl_{ctl} * r_{KL} + \gamma * V_{t+1} - V_t) logP(A_t|V_t)$，这里的$kl_{ctl}$是控制$KL$散度对结果影响的权重参数，\gamma 是控制下一个时间（也就是样本）打分对结果影响的权重参数，V_t是$CriticModel$的打分输出，A_t$ 则是 Reward Model 的打分输出;
6. 根据奖励函数分别计算出的 actor loss 和 critic loss，更新 Actor Model 的参数和 Critic Model 的参数；注意，Actor Model 和 Critic Model 的参数更新方法是不同的，此处就不再一一赘述了，感兴趣的读者可以深入研究强化学习的相关理论。

在上述过程中，因为要使用到四个模型，显存占用会数倍于 SFT。例如，如果我们 RM 和 LLM 都是用 7B 的体量，PPO 过程中大概需要 240G（4张 80G A100，每张卡占用 60G）显存来进行模型加载。那么，为什么我们需要足足四个模型呢？Actor Model 和 Critic Model 较为容易理解，而之所以我们还需要保持原参数不更新的 Ref Model 和 Reward Model，是为了限制模型的更新不要过于偏离原模型以至于丢失了 Pretrain 和 SFT 赋予的能力。

当然，如此大的资源占用和复杂的训练过程，使 RLHF 成为一个门槛非常高的阶段。也有学者从监督学习的思路出发，提出了 DPO（Direct Preference Optimization，直接偏好优化），可以低门槛平替 RLHF。DPO 的核心思路是，将 RLHF 的强化学习问题转化为监督学习来直接学习人类偏好。DPO 通过使用奖励函数和最优策略间的映射，展示了约束奖励最大化问题完全可以通过单阶段策略训练进行优化，也就是说，通过学习 DPO 所提出的优化目标，可以直接学习人类偏好，而无需再训练 RM 以及进行强化学习。由于直接使用监督学习进行训练，DPO 只需要两个 LLM 即可完成训练，且训练过程相较 PPO 简单很多，是 RLHF 更简单易用的平替版本。DPO 所提出的优化目标为什么能够直接学习人类偏好，作者通过一系列的数学推导完成了证明，感兴趣的读者可以下来进一步阅读，此处就不再赘述了。

接下来，我们将依次实现如何从零开始训练一个 LLM，包括预训练、SFT 和 RLHF。

参考资料

[1] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, Ryan Lowe. (2022). *Training language models to follow instructions with human feedback.* arXiv preprint arXiv:2203.02155.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* arXiv preprint arXiv:1810.04805.

[3] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, Dario Amodei. (2020). *Scaling Laws for Neural Language Models.* arXiv preprint arXiv:2001.08361.

[4] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, Laurent Sifre. (2022). *Training Compute-Optimal Large Language Models.* arXiv preprint arXiv:2203.15556.

[5] Qi Wang, Yiyuan Yang, Ji Jiang. (2022). Easy RL: Reinforcement Learning Tutorial . Beijing: Posts & Telecom Press. ISBN: 9787115584700. https://github.com/datawhalechina/easy-rl

[6] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, Chelsea Finn. (2024). *Direct Preference Optimization: Your Language Model is Secretly a Reward Model.* arXiv preprint arXiv:2305.18290.

[7] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, Ji-Rong Wen. (2025). *A Survey of Large Language Models.* arXiv preprint arXiv:2303.18223.

第五章 动手搭建大模型

5.1 动手实现一个 LLaMA2 大模型

Meta（原Facebook）于2023年2月发布第一款基于Transformer结构的大型语言模型LLaMA，并于同年7月发布同系列模型LLaMA2。我们在第四章已经学习了解的了LLM，记忆如何训练LLM等等。那本小节我们就来学习，如何动手写一个LLaMA2模型。

5.1.1 定义超参数

首先我们需要定义一些超参数，这些超参数包括模型的大小、层数、头数、词嵌入维度、隐藏层维度等等。这些超参数可以根据实际情况进行调整。

这里我们自定义一个`ModelConfig`类，来存储和记录我们的超参数，这里我们继承了`PretrainedConfig`类，这是`transformers`库中的参数类，我们可以通过继承这个类来方便的使用`transformers`库中的一些功能，也方便在后续导出Hugging Face模型。

```
from transformers import PretrainedConfig

class ModelConfig(PretrainedConfig):
    model_type = "Tiny-K"
    def __init__(
            self,
            dim: int = 768, # 模型维度
            n_layers: int = 12, # Transformer的层数
            n_heads: int = 16, # 注意力机制的头数
            n_kv_heads: int = 8, # 键值头的数量
            vocab_size: int = 6144, # 词汇表大小
            hidden_dim: int = None, # 隐藏层维度
            multiple_of: int = 64,
            norm_eps: float = 1e-5, # 归一化层的eps
            max_seq_len: int = 512, # 最大序列长度
            dropout: float = 0.0, # dropout概率
            flash_attn: bool = True, # 是否使用Flash Attention
            **kwargs,
    ):
        self.dim = dim
        self.n_layers = n_layers
        self.n_heads = n_heads
        self.n_kv_heads = n_kv_heads
        self.vocab_size = vocab_size
        self.hidden_dim = hidden_dim
        self.multiple_of = multiple_of
        self.norm_eps = norm_eps
        self.max_seq_len = max_seq_len
        self.dropout = dropout
        self.flash_attn = flash_attn
        super().__init__(**kwargs)
```

我们来看一下其中的一些超参数的含义，比如`dim`是模型维度，`n_layers`是Transformer的层数，`n_heads`是注意力机制的头数，`vocab_size`是词汇表大小，`max_seq_len`是输入的最大序列长度等等。上面的代码中也对每一个参数做了详细的注释，在后面的代码中我们会根据这些超参数来构建我们的模型。

5.1.2 构建 RMSNorm

`RMSNorm`可以用如下的数学公式表示：

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \epsilon}} \cdot \gamma \tag{1}$$

其中：

- x_i 是输入向量的第 i 个元素
- γ 是可学习的缩放参数（对应代码中的`self.weight`）
- n 是输入向量的维度数量
- ϵ 是一个小常数，用于数值稳定性（以避免除以零的情况）

这种归一化有助于通过确保权重的规模不会变得过大或过小来稳定学习过程，这在具有许多层的深度学习模型中特别有用。

我们可以通过如下代码实现`RMSNorm`：

```
class RMSNorm(nn.Module):
    def __init__(self, dim: int, eps: float):
        super().__init__()
        # eps是为了防止除以0的情况
        self.eps = eps
        # weight是一个可学习的参数，全部初始化为1
        self.weight = nn.Parameter(torch.ones(dim))

    def _norm(self, x):
        # 计算RMSNorm的核心部分
        # x.pow(2).mean(-1, keepdim=True)计算了输入x的平方的均值
        # torch.rsqrt是平方根的倒数，这样就得到了RMSNorm的分母部分，再加上eps防止分母为0
        # 最后乘以x，得到RMSNorm的结果
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x):
        # forward函数是模型的前向传播
        # 首先将输入x转为float类型，然后进行RMSNorm，最后再转回原来的数据类型
        # 最后乘以weight，这是RMSNorm的一个可学习的缩放因子
        output = self._norm(x.float()).type_as(x)
        return output * self.weight
```

并且，我们可以用下面的代码来对`RMSNorm`模块进行测试，可以看到代码最终输出的形状为`torch.Size([1, 50, 288])`，与我们输入的形状一致，说明模块的实现是正确的，归一化并不会改变输入的形状。

```
norm = RMSNorm(args.dim, args.norm_eps)
x = torch.randn(1, 50, args.dim)
output = norm(x)
print(output.shape)

out:
orch.Size([1, 50, 288])
```

5.1.3 构建 LLaMA2 Attention

在 LLaMA2 模型中，虽然只有 LLaMA2-70B模型使用了分组查询注意力机制（Grouped-Query Attention, GQA），但我们依然选择使用 GQA 来构建我们的 LLaMA Attention 模块，它可以提高模型的效率，并节省一些显存占用。

5.1.3.1 repeat_kv

在 LLaMA2 模型中，我们需要将键和值的维度扩展到和查询的维度一样，这样才能进行注意力计算。我们可以通过如下代码实现 `repeat_kv`：

```
def repeat_kv(x: torch.Tensor, n_rep: int) -> torch.Tensor:
    # 获取输入张量的形状：批量大小、序列长度、键/值对头的数量、每个头的维度大小
    bs, slen, n_kv_heads, head_dim = x.shape

    # 如果重复次数为1，则不需要重复，直接返回原始张量
    if n_rep == 1:
        return x

    # 对张量进行扩展和重塑操作以重复键值对
    return (
        x[:, :, :, None, :]  # 在第四个维度（头的维度前）添加一个新的维度
        .expand(bs, slen, n_kv_heads, n_rep, head_dim)  # 将新添加的维度扩展到n_rep大小，实现重复的效果
        .reshape(bs, slen, n_kv_heads * n_rep, head_dim)  # 重新塑形，合并键/值对头的数量和重复次数的维度
    )
```

在上述代码中：

- 首先，获取输入张量的形状：首先，代码通过 x.shape 获取输入张量的形状，包括批量大小（bs）、序列长度（slen）、键/值对头的数量（n_kv_heads）以及每个头的维度大小（head_dim）。
- 然后，检查重复次数：接着，代码检查重复次数 n_rep 是否为1。如果是1，则说明不需要对键和值进行重复，直接返回原始张量 x。
- 最后，扩展和重塑张量：
  - 在第三个维度（即键/值对头的维度）之后添加一个新的维度，形成 `x[:, :, :, None, :]`。
  - 使用 `expand` 方法将新添加的维度扩展到 `n_rep` 大小，实现键/值对的重复效果。
  - 最后，通过 reshape 方法重新塑形，将扩展后的维度合并回键/值对头的数量中，即 `x.reshape(bs, slen, n_kv_heads * n_rep, head_dim)`，这样最终的张量形状就达到了与查询维度一致的效果。

5.1.3.2 旋转嵌入

接着我们来实现旋转嵌入，旋转嵌入是 LLaMA2 模型中的一个重要组件，它可以为注意力机制提供更强的上下文信息，从而提高模型的性能。

首先，我们要构造获得旋转嵌入的实部和虚部的函数：

```
# 注意：此处的dim应为 dim//n_head，因为我们是对每个head进行旋转嵌入
def precompute_freqs_cis(dim: int, end: int, theta: float = 10000.0):
    # torch.arange(0, dim, 2)[: (dim // 2)].float()生成了一个从0开始，步长为2的序列，长度为dim的一半
    # 然后每个元素除以dim，再取theta的倒数，得到频率
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2)[: (dim // 2)].float() / dim))
    # 生成一个从0到end的序列，长度为end
    t = torch.arange(end, device=freqs.device)
    # 计算外积，得到一个二维矩阵，每一行是t的元素乘以freqs的元素
    freqs = torch.outer(t, freqs).float()
    # 计算频率的余弦值，得到实部
    freqs_cos = torch.cos(freqs)
    # 计算频率的正弦值，得到虚部
    freqs_sin = torch.sin(freqs)
    return freqs_cos, freqs_sin
```

- 计算频率序列：
  - `torch.arange(0, dim, 2)[: (dim // 2)].float()` 生成了一个从0开始，步长为2的序列，其长度为 `dim` 的一半。
  - 每个元素除以 `dim` 后取 `theta` 的倒数，得到一个频率序列 `freqs` 。这一步是为了生成适合旋转嵌入的频率。
- 生成时间序列：
  - `t = torch.arange(end, device=freqs.device)` 生成一个从 `0` 到 `end` 的序列，长度为 `end` 。`end` 通常是序列的最大长度。
- 计算频率的外积
  - `freqs = torch.outer(t, freqs).float()` 计算时间序列 `t` 和频率序列 `freqs` 的外积，得到一个二维矩阵 `freqs` 。每一行是时间序列 `t` 的元素乘以频率序列 `freqs` 的元素。
- 计算实部和虚部
  - `freqs_cos = torch.cos(freqs)` 计算频率矩阵 `freqs` 的余弦值，得到旋转嵌入的实部。
  - `freqs_sin = torch.sin(freqs)` 计算频率矩阵 `freqs` 的正弦值，得到旋转嵌入的虚部。

最终，该函数返回两个矩阵 `freqs_cos` 和 `freqs_sin` ，分别表示旋转嵌入的实部和虚部，用于后续的计算。

接着，我们来构造调整张量形状的 `reshape_for_broadcast` 函数，这个函数的主要目的是调整 `freqs_cis` 的形状，使其在进行广播操作时与 `x` 的维度对齐，从而能够进行正确的张量运算。

```
def reshape_for_broadcast(freqs_cis: torch.Tensor, x: torch.Tensor):
    # 获取x的维度数
    ndim = x.ndim
```

```
    # 断言，确保1在x的维度范围内
    assert 0 <= 1 < ndim

    # 断言，确保freqs_cis的形状与x的第二维和最后一维相同
    assert freqs_cis.shape == (x.shape[1], x.shape[-1])

    # 构造一个新的形状，除了第二维和最后一维，其他维度都为1，这样做是为了能够将freqs_cis与x进行广播操作
    shape = [d if i == 1 or i == ndim - 1 else 1 for i, d in enumerate(x.shape)]

    # 将freqs_cis调整为新的形状，并返回
    return freqs_cis.view(shape)
```

最后，我们可以通过如下代码实现旋转嵌入：

```
def apply_rotary_emb(
    xq: torch.Tensor,
    xk: torch.Tensor,
    freqs_cos: torch.Tensor,
    freqs_sin: torch.Tensor
) -> Tuple[torch.Tensor, torch.Tensor]:

    # 将查询和键张量转换为浮点数，并重塑形状以分离实部和虚部
    xq_r, xq_i = xq.float().reshape(xq.shape[:-1] + (-1, 2)).unbind(-1)
    xk_r, xk_i = xk.float().reshape(xk.shape[:-1] + (-1, 2)).unbind(-1)

    # 重新塑形频率张量以进行广播
    freqs_cos = reshape_for_broadcast(freqs_cos, xq_r)
    freqs_sin = reshape_for_broadcast(freqs_sin, xq_r)

    # 应用旋转，分别计算旋转后的实部和虚部
    xq_out_r = xq_r * freqs_cos - xq_i * freqs_sin
    xq_out_i = xq_r * freqs_sin + xq_i * freqs_cos
    xk_out_r = xk_r * freqs_cos - xk_i * freqs_sin
    xk_out_i = xk_r * freqs_sin + xk_i * freqs_cos

    # 将最后两个维度合并，并还原为原始张量的形状
    xq_out = torch.stack([xq_out_r, xq_out_i], dim=-1).flatten(3)
    xk_out = torch.stack([xk_out_r, xk_out_i], dim=-1).flatten(3)

    return xq_out.type_as(xq), xk_out.type_as(xk)
```

这里我们给出可以测试`apply_rotary_emb`函数的代码，大家也可以尝试在代码中添加断点，来查看每一步的计算结果。

```
xq = torch.randn(1, 50, 6, 48) # bs, seq_len, dim//n_head, n_head_dim
xk = torch.randn(1, 50, 6, 48) # bs, seq_len, dim//n_head, n_head_dim

# 使用 precompute_freqs_cis 函数获取 sin和cos
cos, sin = precompute_freqs_cis(288//6, 50)
print(cos.shape, sin.shape)
xq_out, xk_out = apply_rotary_emb(xq, xk, cos, sin)

xq_out.shape, xk_out.shape
```

OUT:

```
torch.Size([50, 24]) torch.Size([50, 24])

(torch.Size([1, 50, 6, 48]), torch.Size([1, 50, 6, 48]))
```

5.1.3.3 组装 LLaMA2 Attention

在上面我们已经完成了旋转嵌入的实现，接下来我们就可以构建 LLaMA2 Attention 模块了。

```
class Attention(nn.Module):
    def __init__(self, args: ModelConfig):
        super().__init__()
        # 根据是否指定n_kv_heads，确定用于键（key）和值（value）的头的数量。
        self.n_kv_heads = args.n_heads if args.n_kv_heads is None else args.n_kv_heads
        # 确保总头数可以被键值头数整除。
        assert args.n_heads % self.n_kv_heads == 0

        # 模型并行处理大小，默认为1。
        model_parallel_size = 1
        # 本地计算头数，等于总头数除以模型并行处理大小。
        self.n_local_heads = args.n_heads // model_parallel_size
        # 本地键值头数，等于键值头数除以模型并行处理大小。
        self.n_local_kv_heads = self.n_kv_heads // model_parallel_size
        # 重复次数，用于扩展键和值的尺寸。
        self.n_rep = self.n_local_heads // self.n_local_kv_heads
        # 每个头的维度，等于模型维度除以头的总数。
        self.head_dim = args.dim // args.n_heads

        # 定义权重矩阵。
        self.wq = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        self.wk = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)
        self.wv = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)
        # 输出权重矩阵。
        self.wo = nn.Linear(args.n_heads * self.head_dim, args.dim, bias=False)

        # 定义dropout。
        self.attn_dropout = nn.Dropout(args.dropout)
        self.resid_dropout = nn.Dropout(args.dropout)
        # 保存dropout概率。
```

```
        self.dropout = args.dropout

        # 检查是否使用Flash Attention（需要PyTorch >= 2.0）。
        self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention')
        if not self.flash:
            # 若不支持Flash Attention，则使用手动实现的注意力机制，并设置mask。
            print("WARNING: using slow attention. Flash Attention requires PyTorch >=
2.0")
            # 创建一个上三角矩阵，用于遮蔽未来信息。
            mask = torch.full((1, 1, args.max_seq_len, args.max_seq_len), float("-inf"))
            mask = torch.triu(mask, diagonal=1)
            # 注册为模型的缓冲区
            self.register_buffer("mask", mask)

    def forward(self, x: torch.Tensor, freqs_cos: torch.Tensor, freqs_sin: torch.Tensor):
        # 获取批次大小和序列长度，[batch_size, seq_len, dim]
        bsz, seqlen, _ = x.shape

        # 计算查询（Q）、键（K）、值（V）。
        xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)
        # 调整形状以适应头的维度。
        xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)

        # 应用旋转位置嵌入（RoPE）。
        xq, xk = apply_rotary_emb(xq, xk, freqs_cos, freqs_sin)

        # 对键和值进行扩展以适应重复次数。
        xk = repeat_kv(xk, self.n_rep)
        xv = repeat_kv(xv, self.n_rep)

        # 将头作为批次维度处理。
        xq = xq.transpose(1, 2)
        xk = xk.transpose(1, 2)
        xv = xv.transpose(1, 2)

        # 根据是否支持Flash Attention，选择实现方式。
        if self.flash:
            # 使用Flash Attention。
            output = torch.nn.functional.scaled_dot_product_attention(xq, xk, xv,
attn_mask=None, dropout_p=self.dropout if self.training else 0.0, is_causal=True)
        else:
            # 使用手动实现的注意力机制。
            scores = torch.matmul(xq, xk.transpose(2, 3)) / math.sqrt(self.head_dim)
            assert hasattr(self, 'mask')
            scores = scores + self.mask[:, :, :seqlen, :seqlen]
            scores = F.softmax(scores.float(), dim=-1).type_as(xq)
            scores = self.attn_dropout(scores)
            output = torch.matmul(scores, xv)

        # 恢复时间维度并合并头。
```

```
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)

        # 最终投影回残差流。
        output = self.wo(output)
        output = self.resid_dropout(output)
        return output
```

同样大家可以使用下面的代码来对注意力模块进行测试，可以看到代码最终输出的形状为`torch.Size([1, 50, 768])`，与我们输入的形状一致，说明模块的实现是正确的。

```
# 创建Attention实例
attention_model = Attention(args)

# 模拟输入数据
batch_size = 1
seq_len = 50  # 假设实际使用的序列长度为50
dim = args.dim
x = torch.rand(batch_size, seq_len, dim)  # 随机生成输入张量
# freqs_cos = torch.rand(seq_len, dim // 2)  # 模拟cos频率，用于RoPE
# freqs_sin = torch.rand(seq_len, dim // 2)  # 模拟sin频率，用于RoPE

freqs_cos, freqs_sin = precompute_freqs_cis(dim//args.n_heads, seq_len)

# 运行Attention模型
output = attention_model(x, freqs_cos, freqs_sin)

# attention出来之后的形状 依然是[batch_size, seq_len, dim]
print("Output shape:", output.shape)
```

OUT:

```
Output shape: torch.Size([1, 50, 768])
```

5.1.4 构建 LLaMA2 MLP模块

相对于前面我们实现的LLaMA2 Attention模块，LLaMA2 MLP模块的实现要简单一些。我们可以通过如下代码实现`MLP`：

```
class MLP(nn.Module):
    def __init__(self, dim: int, hidden_dim: int, multiple_of: int, dropout: float):
        super().__init__()
        # 如果没有指定隐藏层的维度，我们将其设置为输入维度的4倍
        # 然后将其减少到2/3，最后确保它是multiple_of的倍数
        if hidden_dim is None:
            hidden_dim = 4 * dim
            hidden_dim = int(2 * hidden_dim / 3)
            hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)
        # 定义第一层线性变换，从输入维度到隐藏维度
        self.w1 = nn.Linear(dim, hidden_dim, bias=False)
        # 定义第二层线性变换，从隐藏维度到输入维度
```

```
        self.w2 = nn.Linear(hidden_dim, dim, bias=False)
        # 定义第三层线性变换，从输入维度到隐藏维度
        self.w3 = nn.Linear(dim, hidden_dim, bias=False)
        # 定义dropout层，用于防止过拟合
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # 前向传播函数
        # 首先，输入x通过第一层线性变换和SILU激活函数
        # 然后，结果乘以输入x通过第三层线性变换的结果
        # 最后，通过第二层线性变换和dropout层
        return self.dropout(self.w2(F.silu(self.w1(x)) * self.w3(x)))
```

我们着重观察一下`forward`函数的实现，首先，输入`x`通过第一层线性变换`self.w1`和`SILU`激活函数，然后，结果乘以输入`x`通过第三层线性变换`self.w3`的结果，最后，通过第二层线性变换`self.w2`和`dropout`层，得到最终输出。

同样大家可以使用下面的代码来对`LLaMAMLP`模块进行测试，可以看到代码最终输出的形状为`torch.Size([1, 50, 768])`，与我们输入的形状一致，说明模块的实现是正确的。

```
# 创建MLP实例
mlp = MLP(args.dim, args.hidden_dim, args.multiple_of, args.dropout)
# 随机生成数据
x = torch.randn(1, 50, args.dim)
# 运行MLP模型
output = mlp(x)
print(output.shape)
```

OUT:

```
torch.Size([1, 50, 768])
```

5.1.5 LLaMA2 Decoder Layer

到这里，我们已经实现了`LLaMA2`模型的`Attention`模块和`MLP`模块，接下来我们就可以构建`LLaMA2`的`Decoder Layer`了。

```
class DecoderLayer(nn.Module):
    def __init__(self, layer_id: int, args: ModelConfig):
        super().__init__()
        # 定义多头注意力的头数
        self.n_heads = args.n_heads
        # 定义输入维度
        self.dim = args.dim
        # 定义每个头的维度，等于输入维度除以头数
        self.head_dim = args.dim // args.n_heads
        # 定义LLaMA2Attention对象，用于进行多头注意力计算
        self.attention = Attention(args)
        # 定义LLaMAMLP对象，用于进行前馈神经网络计算
        self.feed_forward = MLP(
```

```
            dim=args.dim,
            hidden_dim=args.hidden_dim,
            multiple_of=args.multiple_of,
            dropout=args.dropout,
        )
        # 定义层的ID
        self.layer_id = layer_id
        # 定义注意力计算的归一化层
        self.attention_norm = RMSNorm(args.dim, eps=args.norm_eps)
        # 定义前馈神经网络计算的归一化层
        self.ffn_norm = RMSNorm(args.dim, eps=args.norm_eps)

    def forward(self, x, freqs_cos, freqs_sin):
        # 前向传播函数
        # 首先，输入x经过注意力归一化层，然后进行注意力计算，结果与输入x相加得到h
        # 然后，h经过前馈神经网络归一化层，然后进行前馈神经网络计算，结果与h相加得到输出
        h = x + self.attention.forward(self.attention_norm(x), freqs_cos, freqs_sin)
        out = h + self.feed_forward.forward(self.ffn_norm(h))
        return out
```

`DecoderLayer`就是把我们上面完成的`Attention`模块和`MLP`模块组合在一起，实现了一个完整的`Transformer`模块。

同样大家可以使用下面的代码来对`DecoderLayer`模块进行测试，可以看到代码最终输出的形状为`torch.Size([1, 50, 768])`，与我们输入的形状一致，说明模块的实现是正确的。

```
# 创建LLaMADecoderLayer实例
decoderlayer = DecoderLayer(0, args)

# 模拟输入数据
dim = args.dim
seq_len = 50

x = torch.randn(1, seq_len, dim) # [bs, seq_len, dim]

freqs_cos, freqs_sin = precompute_freqs_cis(dim//args.n_heads, seq_len)

out = decoderlayer(x, freqs_cos, freqs_sin)

print(out.shape) # 形状和输入的x一样 [batch_size, seq_len, dim]
```

OUT:

```
torch.Size([1, 50, 768])
```

5.1.6 构建 LLaMA2 模型

好了，我们已经完了上述所有的模块的实现，接下来就是激动人心的时刻，我们可以构建`LLaMA2`模型了。，`LLaMA2`模型就是将`DecoderLayer`模块堆叠起来，构成一个完整的`Transformer`模型。

```python
class Transformer(PreTrainedModel):
    config_class = ModelConfig  # 配置类
    last_loss: Optional[torch.Tensor] # 记录最后一次计算的损失

    def __init__(self, args: ModelConfig = None):
        super().__init__(args)
        # 初始化模型参数
        self.args = args
        # 词汇表大小
        self.vocab_size = args.vocab_size
        # 层数
        self.n_layers = args.n_layers

        # 词嵌入层
        self.tok_embeddings = nn.Embedding(args.vocab_size, args.dim)
        # Dropout层
        self.dropout = nn.Dropout(args.dropout)
        # Decoder层
        self.layers = torch.nn.ModuleList()
        for layer_id in range(args.n_layers):
            self.layers.append(DecoderLayer(layer_id, args))
        # 归一化层
        self.norm = RMSNorm(args.dim, eps=args.norm_eps)
        # 输出层
        self.output = nn.Linear(args.dim, args.vocab_size, bias=False)

        # 将词嵌入层的权重与输出层的权重共享
        self.tok_embeddings.weight = self.output.weight

        # 预计算相对位置嵌入的频率
        freqs_cos, freqs_sin = precompute_freqs_cis(self.args.dim // self.args.n_heads,
self.args.max_seq_len)
        self.register_buffer("freqs_cos", freqs_cos, persistent=False)
        self.register_buffer("freqs_sin", freqs_sin, persistent=False)

        # 初始化所有权重
        self.apply(self._init_weights)
        # 对残差投影进行特殊的缩放初始化
        for pn, p in self.named_parameters():
            if pn.endswith('w3.weight') or pn.endswith('wo.weight'):
                torch.nn.init.normal_(p, mean=0.0, std=0.02/math.sqrt(2 * args.n_layers))

        # 初始化最后一次前向传播的损失属性
        self.last_loss = None
        self.OUT = CausalLMOutputWithPast()  # 输出容器
        self._no_split_modules = [name for name, _ in self.named_modules()]  # 不分割的模块
列表

    def _init_weights(self, module):
        # 初始化权重的函数
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
```

```
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, tokens: torch.Tensor, targets: Optional[torch.Tensor] = None,
**keyargs) -> torch.Tensor:
        """
        - tokens: Optional[torch.Tensor], 输入 token 张量。
        - targets: Optional[torch.Tensor], 目标 token 张量。
        - kv_cache: bool, 是否使用键值缓存。
        - keyargs: 其他关键字参数。

        - self.OUT: CausalLMOutputWithPast, 包含 logits 和损失。
        """

        if 'input_ids' in keyargs:
            tokens = keyargs['input_ids']
        if 'attention_mask' in keyargs:
            targets = keyargs['attention_mask']

        # 前向传播函数
        _bsz, seqlen = tokens.shape
        # 通过词嵌入层和Dropout层
        h = self.tok_embeddings(tokens)
        h = self.dropout(h)
        # 获取相对位置嵌入的频率
        freqs_cos = self.freqs_cos[:seqlen]
        freqs_sin = self.freqs_sin[:seqlen]

        # 通过Decoder层
        for layer in self.layers:
            h = layer(h, freqs_cos, freqs_sin)
        # 通过归一化层
        h = self.norm(h)

        if targets is not None:
            # 如果给定了目标，计算损失
            logits = self.output(h)
            self.last_loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1), ignore_index=0, reduction='none')
        else:
            # 推理时的小优化：只对最后一个位置的输出进行前向传播
            logits = self.output(h[:, [-1], :])
            self.last_loss = None

        # 设置输出
        self.OUT.__setitem__('logits', logits)
        self.OUT.__setitem__('last_loss', self.last_loss)
        return self.OUT
```

```
    @torch.inference_mode()
    def generate(self, idx, stop_id=None, max_new_tokens=256, temperature=1.0,
top_k=None):
        """
        给定输入序列 idx（形状为 (bz,seq_len) 的长整型张量），通过多次生成新 token 来完成序列。
        在 model.eval() 模式下运行。效率较低的采样版本，没有使用键k/v cache。
        """
        index = idx.shape[1]
        for _ in range(max_new_tokens):
            # 如果序列上下文过长，截断它到最大长度
            idx_cond = idx if idx.size(1) <= self.args.max_seq_len else idx[:, -
self.args.max_seq_len:]

            # 前向传播获取序列中最后一个位置的 logits
            logits = self(idx_cond).logits
            logits = logits[:, -1, :] # 只保留最后一个时间步的输出

            if temperature == 0.0:
                # 选择最有可能的索引
                _, idx_next = torch.topk(logits, k=1, dim=-1)
            else:
                # 缩放 logits 并应用 softmax
                logits = logits / temperature
                if top_k is not None:
                    v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
                    logits[logits < v[:, [-1]]] = -float('Inf')
                probs = F.softmax(logits, dim=-1)
                idx_next = torch.multinomial(probs, num_samples=1)


            if idx_next == stop_id:
                break

            # 将采样的索引添加到序列中并继续
            idx = torch.cat((idx, idx_next), dim=1)

        return idx[:, index:] # 只返回生成的token
```

同样大家可以使用下面的代码来对 `Transformer` 模块进行测试，可以看到代码最终输出的形状为 `torch.Size([1, 1, 6144])`，与我们输入的形状一致，说明模块的实现是正确的。

```
# LLaMA2Model.forward 接受两个参数，tokens和targets，其中tokens是输入的张量，应为int类型
x = torch.randint(0, 6144, (1, 50)) # [bs, seq_len]
# 实例化LLaMA2Model
model = Transformer(args=args)
# 计算model的全部参数
num_params = sum(p.numel() for p in model.parameters())
print('Number of parameters:', num_params)

out = model(x)
print(out.logits.shape) # [batch_size, 1, vocab_size]
```

OUT:

```
Number of parameters: 82594560
torch.Size([1, 1, 6144])
```

5.2 训练 Tokenizer

在自然语言处理 (NLP) 中，Tokenizer 是一种将文本分解为较小单位（称为 token）的工具。这些 token 可以是词、子词、字符，甚至是特定的符号。Tokenization 是 NLP 中的第一步，直接影响后续处理和分析的效果。不同类型的 tokenizer 适用于不同的应用场景，以下是几种常见的 tokenizer 及其特点。

5.3.1 Word-based Tokenizer

Word-based Tokenizer 是最简单和直观的一种分词方法。它将文本按空格和标点符号分割成单词。这种方法的优点在于其简单和直接，易于实现，且与人类对语言的直觉相符。然而，它也存在一些明显的缺点，如无法处理未登录词（OOV，out-of-vocabulary）和罕见词，对复合词（如“New York”）或缩略词（如“don't”）的处理也不够精细。此外，Word-based Tokenizer 在处理不同语言时也会遇到挑战，因为一些语言（如中文、日文）没有显式的单词分隔符。

示例：

```
Input: "Hello, world! There is Datawhale."
Output: ["Hello", ",", "world", "!", "There", "is", "Datawhale", "."]
```

在这个例子中，输入的句子被分割成一系列单词和标点符号，每个单词或标点符号都作为一个独立的 token。

5.2.2 Character-based Tokenizer

Character-based Tokenizer 将文本中的每个字符视为一个独立的 token。这种方法能非常精细地处理文本，适用于处理拼写错误、未登录词或新词。由于每个字符都是一个独立的 token，因此这种方法可以捕捉到非常细微的语言特征。这对于一些特定的应用场景，如生成式任务或需要处理大量未登录词的任务，特别有用。但是，这种方法也会导致 token 序列变得非常长，增加了模型的计算复杂度和训练时间。此外，字符级的分割可能会丢失一些词级别的语义信息，使得模型难以理解上下文。

示例：

```
Input: "Hello"
Output: ["H", "e", "l", "l", "o"]
```

在这个例子中，单词“Hello”被分割成单个字符，每个字符作为一个独立的 token。这种方法能够处理任何语言和字符集，具有极大的灵活性。

5.2.3 Subword Tokenizer

Subword Tokenizer 介于词和字符之间，能够更好地平衡分词的细粒度和处理未登录词的能力。Subword Tokenizer 的关键思想是将文本分割成比单词更小的单位，但又比字符更大，这样既能处理未知词，又能保持一定的语义信息。常见的子词分词方法包括 BPE、WordPiece 和 Unigram。

（1）Byte Pair Encoding (BPE)

BPE 是一种基于统计方法，通过反复合并频率最高的字符或字符序列对来生成子词词典。这种方法的优点在于其简单和高效，能够有效地处理未知词和罕见词，同时保持较低的词典大小。BPE 的合并过程是自底向上的，逐步将频率最高的字符对合并成新的子词，直到达到预定的词典大小或不再有高频的字符对。

示例：

```
Input: "lower"
Output: ["low", "er"]

Input: "newest"
Output: ["new", "est"]
```

在这个例子中，单词“lower”被分割成子词“low”和“er”，而“newest”被分割成“new”和“est”。这种方法有效地处理了词干和词缀，保持了单词的基本语义结构。

（2）WordPiece

WordPiece 是另一种基于子词的分词方法，最初用于谷歌的 BERT 模型。与 BPE 类似，WordPiece 通过最大化子词序列的似然函数来生成词典，但在合并子词时更注重语言模型的优化。WordPiece 会优先选择能够最大化整体句子概率的子词，使得分词结果在语言模型中具有更高的概率。

示例：

```
Input: "unhappiness"
Output: ["un", "##happiness"]
```

在这个例子中，单词“unhappiness”被分割成子词“un”和“##happiness”，其中“##”表示这是一个后缀子词。通过这种方式，WordPiece 能够更好地处理复合词和派生词，保留更多的语义信息。

（3）Unigram

Unigram 分词方法基于概率模型，通过选择具有最高概率的子词来分割文本。Unigram 词典是通过训练语言模型生成的，可以处理多种语言和不同类型的文本。Unigram 模型会为每个子词分配一个概率，然后根据这些概率进行最优分割。

示例：

```
Input: "unhappiness"
Output: ["un", "happiness"]

Input: "newest"
Output: ["new", "est"]
```

在这个例子中，单词“unhappiness”被分割成子词“un”和“happiness”，而“newest”被分割成“new”和“est”。这种方法通过概率模型有效地处理了子词分割，使得分割结果更符合语言使用习惯。

每种 Tokenizer 方法都有其特定的应用场景和优缺点，选择适合的 Tokenizer 对于自然语言处理任务的成功至关重要。

5.2.4 训练一个 Tokenizer

这里我们选择使用 BPE 算法来训练一个 Subword Tokenizer。BPE 是一种简单而有效的分词方法，能够处理未登录词和罕见词，同时保持较小的词典大小。我们将使用 Hugging Face 的 `tokenizers` 库来训练一个 BPE Tokenizer。

Step 1: 安装和导入依赖库

首先，我们需要安装 `tokenizers` 库，除此之外还需要安装 `datasets` 和 `transformers` 库，用于加载训练数据和加载训练完成后的 Tokenizer。

```
pip install tokenizers datasets transformers
```

然后，导入所需的库。

```
import random
import json
import os
from transformers import AutoTokenizer, PreTrainedTokenizerFast
from tokenizers import (
    decoders,
    models,
    pre_tokenizers,
    trainers,
    Tokenizer,
)
from tokenizers.normalizers import NFKC
from typing import Generator
```

Step 2: 加载训练数据

我们使用 `datasets.load_dataset()` 库加载一个英文文本数据集，用于训练 BPE Tokenizer。这里我们使用 `wikitext` 数据集，包含了维基百科的文章文本。

```
dataset = load_dataset("wikitext", "wikitext-103-v1", split="train")

# 准备训练数据
def batch_iterator(batch_size=1000):
    for i in range(0, len(dataset), batch_size):
        yield dataset[i:i + batch_size]["text"]
```

如果你使用本地的文本数据集，可以将数据加载到一个列表中，然后传入 `batch_iterator()` 函数中。如下所示：

```
def load_text_from_files(path_list):
    text_data = []
    for file_path in path_list:
        with open(file_path, 'r', encoding='utf-8') as file:
            text_data.extend(file.readlines())
    return text_data
```

```
def batch_iterator(text_data, batch_size=1000):
    for i in range(0, len(text_data), batch_size):
        yield text_data[i:i + batch_size]

# 假设你的文件路径列表是
path_list = ['text_data1.txt', 'text_data2.txt', 'text_data3.txt']
text_data = load_text_from_files(path_list)
```

Step 3: 创建配置文件

在训练 BPE Tokenizer 之前，我们需要创建一个完整的 `Tokenizer` 配置文件，包括 `tokenizer_config.json` 和 `special_tokens_map.json`。这些配置文件定义了 `Tokenizer` 的参数和特殊标记，用于训练和加载 `Tokenizer`。此处的 `chat_template` 我们与 `Qwen2.5` 模型保持一致。

```
def create_tokenizer_config(save_dir: str) -> None:
    """创建完整的tokenizer配置文件"""
    config = {
        "add_bos_token": False,
        "add_eos_token": False,
        "add_prefix_space": False,
        "bos_token": "<|im_start|>",
        "eos_token": "<|im_end|>",
        "pad_token": "<|im_end|>",
        "unk_token": "<unk>",
        "model_max_length": 1000000000000000019884624838656,
        "clean_up_tokenization_spaces": False,
        "tokenizer_class": "PreTrainedTokenizerFast",
        "chat_template": (
            "{% for message in messages %}"
            "{% if message['role'] == 'system' %}"
            "<|im_start|>system\n{{ message['content'] }}<|im_end|>\n"
            "{% elif message['role'] == 'user' %}"
            "<|im_start|>user\n{{ message['content'] }}<|im_end|>\n"
            "{% elif message['role'] == 'assistant' %}"
            "<|im_start|>assistant\n{{ message['content'] }}<|im_end|>\n"
            "{% endif %}"
            "{% endfor %}"
            "{% if add_generation_prompt %}"
            "{{ '<|im_start|>assistant\n' }}"
            "{% endif %}"
        )
    }

    # 保存主配置文件
    with open(os.path.join(save_dir, "tokenizer_config.json"), "w", encoding="utf-8") as
f:
        json.dump(config, f, ensure_ascii=False, indent=4)

    # 创建special_tokens_map.json
    special_tokens_map = {
        "bos_token": "<|im_start|>",
```

```
        "eos_token": "<|im_end|>",
        "unk_token": "<unk>",
        "pad_token": "<|im_end|>",
        "additional_special_tokens": ["<s>", "</s>"]
    }
    with open(os.path.join(save_dir, "special_tokens_map.json"), "w", encoding="utf-8") as
f:
        json.dump(special_tokens_map, f, ensure_ascii=False, indent=4)
```

Step 4: 训练 BPE Tokenizer

在训练 BPE Tokenizer 之前，我们需要定义一个训练函数，用于训练 Tokenizer 并保存训练好的 Tokenizer 文件。这里我们使用 `tokenizers` 库中的 `Tokenizer` 类来训练 BPE Tokenizer。

可以看到我们在训练 Tokenizer 时，配置了一些特殊的 token，如 `<unk>`、`<s>`、`</s>`、`<|im_start|>` 和 `<|im_end|>`。这些 token 用于标记未知词、句子的开始和结束，以及对话的开始和结束。这些特殊 token 可以帮助模型更好地理解文本数据，提高模型的泛化能力和效果。

```
def train_tokenizer(data_path: str, save_dir: str, vocab_size: int = 8192) -> None:
    """训练并保存自定义tokenizer"""
    os.makedirs(save_dir, exist_ok=True)

    # 初始化tokenizer
    tokenizer = Tokenizer(models.BPE(unk_token="<unk>"))
    tokenizer.normalizer = NFKC()  # 添加文本规范化
    tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel(add_prefix_space=False)
    tokenizer.decoder = decoders.ByteLevel()

    # 配置特殊token
    special_tokens = [
        "<unk>",
        "<s>",
        "</s>",
        "<|im_start|>",
        "<|im_end|>"
    ]

    # 配置训练器
    trainer = trainers.BpeTrainer(
        vocab_size=vocab_size,
        special_tokens=special_tokens,
        min_frequency=2,  # 提高低频词过滤
        show_progress=True,
        initial_alphabet=pre_tokenizers.ByteLevel.alphabet()
    )

    # 训练tokenizer
    print(f"Training tokenizer with data from {data_path}")
    texts = read_texts_from_jsonl(data_path)
    tokenizer.train_from_iterator(texts, trainer=trainer,
length=os.path.getsize(data_path))
```

```
    # 验证特殊token映射
    try:
        assert tokenizer.token_to_id("<unk>") == 0
        assert tokenizer.token_to_id("<s>") == 1
        assert tokenizer.token_to_id("</s>") == 2
        assert tokenizer.token_to_id("<|im_start|>") == 3
        assert tokenizer.token_to_id("<|im_end|>") == 4
    except AssertionError as e:
        print("Special tokens mapping error:", e)
        raise

    # 保存tokenizer文件
    tokenizer.save(os.path.join(save_dir, "tokenizer.json"))

    # 创建配置文件
    create_tokenizer_config(save_dir)
    print(f"Tokenizer saved to {save_dir}")
```

Step 5: 使用训练好的 Tokenizer

我们可以使用训练好的 Tokenizer 来处理文本数据，如编码、解码、生成对话等。下面是一个简单的示例，展示了如何使用训练好的 Tokenizer 来处理文本数据。

```
def eval_tokenizer(tokenizer_path: str) -> None:
    """评估tokenizer功能"""
    try:
        tokenizer = AutoTokenizer.from_pretrained(tokenizer_path)
    except Exception as e:
        print(f"Error loading tokenizer: {e}")
        return

    # 测试基本属性
    print("\n=== Tokenizer基本信息 ===")
    print(f"Vocab size: {len(tokenizer)}")
    print(f"Special tokens: {tokenizer.all_special_tokens}")
    print(f"Special token IDs: {tokenizer.all_special_ids}")

    # 测试聊天模板
    messages = [
        {"role": "system", "content": "你是一个AI助手。"},
        {"role": "user", "content": "How are you?"},
        {"role": "assistant", "content": "I'm fine, thank you. and you?"},
        {"role": "user", "content": "I'm good too."},
        {"role": "assistant", "content": "That's great to hear!"},
    ]

    print("\n=== 聊天模板测试 ===")
    prompt = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
```

```
        # add_generation_prompt=True
    )
    print("Generated prompt:\n", prompt, sep="")

    # 测试编码解码
    print("\n=== 编码解码测试 ===")
    encoded = tokenizer(prompt, truncation=True, max_length=256)
    decoded = tokenizer.decode(encoded["input_ids"], skip_special_tokens=False)
    print("Decoded text matches original:", decoded == prompt)

    # 测试特殊token处理
    print("\n=== 特殊token处理 ===")
    test_text = "<|im_start|>user\nHello<|im_end|>"
    encoded = tokenizer(test_text).input_ids
    decoded = tokenizer.decode(encoded)
    print(f"Original: {test_text}")
    print(f"Decoded:  {decoded}")
    print("Special tokens preserved:", decoded == test_text)
```

```
eval_tokenizer('your tokenizer path')
```

OUT:

```
=== Tokenizer基本信息 ===
Vocab size: 6144
Special tokens: ['<|im_start|>', '<|im_end|>', '<unk>', '<s>', '</s>']
Special token IDs: [3, 4, 0, 1, 2]

=== 聊天模板测试 ===
Generated prompt:
<|im_start|>system
你是一个AI助手。<|im_end|>
<|im_start|>user
How are you?<|im_end|>
<|im_start|>assistant
I'm fine, thank you. and you?<|im_end|>
<|im_start|>user
I'm good too.<|im_end|>
<|im_start|>assistant
That's great to hear!<|im_end|>


=== 编码解码测试 ===
Decoded text matches original: False

=== 特殊token处理 ===
Original: <|im_start|>user
Hello<|im_end|>
Decoded:  <|im_start|> user
Hello<|im_end|>
Special tokens preserved: False
```

5.3 预训练一个小型LLM

在前面的章节中，我们熟悉了各种大模型的模型结构，以及如如何训练Tokenizer。在本节中，我们将动手训练一个八千万参数的LLM。

5.3.0 数据下载

首先，我们需要下载预训练数据集。在这里，我们使用两个开源的数据集，包含了大量的中文对话数据，可以用于训练对话生成模型。

- 出门问问序列猴子开源数据集：出门问问序列猴子通用文本数据集由来自网页、百科、博客、问答、开源代码、书籍、报刊、专利、教材、考题等多种公开可获取的数据进行汇总清洗之后而形成的大语言模型预训练语料。总量大概在 10B Token。
- BelleGroup：350万条中文对话数据集，包含了人机对话、人人对话、人物对话等多种对话数据，可以用于训练对话生成模型。

```
# 下载预训练数据集
os.system("modelscope download --dataset ddzhu123/seq-monkey
mobvoi_seq_monkey_general_open_corpus.jsonl.tar.bz2 --local_dir your_local_dir")
# 解压预训练数据集
os.system("tar -xvf your_local_dir/mobvoi_seq_monkey_general_open_corpus.jsonl.tar.bz2")

# 下载SFT数据集
os.system(f'huggingface-cli download --repo-type dataset --resume-download
BelleGroup/train_3.5M_CN --local-dir BelleGroup')



# 1 处理预训练数据
def split_text(text, chunk_size=512):
    """将文本按指定长度切分成块"""
    return [text[i:i+chunk_size] for i in range(0, len(text), chunk_size)]

input_file = 'mobvoi_seq_monkey_general_open_corpus.jsonl'

with open('seq_monkey_datawhale.jsonl', 'a', encoding='utf-8') as pretrain:
    with open(input_file, 'r', encoding='utf-8') as f:
        data = f.readlines()
        for line in tqdm(data, desc=f"Processing lines in {input_file}", leave=False):  #
添加行级别的进度条
            line = json.loads(line)
            text = line['text']
            chunks = split_text(text)
            for chunk in chunks:
                pretrain.write(json.dumps({'text': chunk}, ensure_ascii=False) + '\n')

# 2 处理SFT数据

def convert_message(data):
```

```
    """
    将原始数据转换为标准格式
    """
    message = [
        {"role": "system", "content": "你是一个AI助手"},
    ]
    for item in data:
        if item['from'] == 'human':
            message.append({'role': 'user', 'content': item['value']})
        elif item['from'] == 'assistant':
            message.append({'role': 'assistant', 'content': item['value']})
    return message

with open('BelleGroup_sft.jsonl', 'a', encoding='utf-8') as sft:
    with open('BelleGroup/train_3.5M_CN.json', 'r') as f:
        data = f.readlines()
        for item in tqdm(data, desc="Processing", unit="lines"):
            item = json.loads(item)
            message = convert_message(item['conversations'])
            sft.write(json.dumps(message, ensure_ascii=False) + '\n')
```

5.3.1 训练Tokenize

首先，我们需要为文本处理训练一个Tokenizer。Tokenizer的作用是将文本转换为数字序列，以便模型能够理解和处理。我们使用的数据集是 出门问问序列猴子开源数据集 ，这个数据集包含了大量的中文文本数据，可以用于训练Tokenizer。

> 注：由于数据集较大，如果大家在自己本地电脑训练的话进度比较慢，所以在这里我们提供了一个已经训练好的Tokenizer，大家可以直接使用。如果大家想要自己训练的话，可以参考下面的代码。

```
python code/train_tokenizer.py
```

```
import random
import json
import os
from transformers import AutoTokenizer, PreTrainedTokenizerFast
from tokenizers import (
    decoders,
    models,
    pre_tokenizers,
    trainers,
    Tokenizer,
)
from tokenizers.normalizers import NFKC
from typing import Generator

random.seed(42)

def read_texts_from_jsonl(file_path: str) -> Generator[str, None, None]:
    """读取JSONL文件并安全提取文本数据"""
```

```
    with open(file_path, 'r', encoding='utf-8') as f:
        for line_num, line in enumerate(f, 1):
            try:
                data = json.loads(line)
                if 'text' not in data:
                    raise KeyError(f"Missing 'text' field in line {line_num}")
                yield data['text']
            except json.JSONDecodeError:
                print(f"Error decoding JSON in line {line_num}")
                continue
            except KeyError as e:
                print(e)
                continue

def create_tokenizer_config(save_dir: str) -> None:
    """创建完整的tokenizer配置文件"""
    config = {
        "add_bos_token": False,
        "add_eos_token": False,
        "add_prefix_space": True,
        "bos_token": "<|im_start|>",
        "eos_token": "<|im_end|>",
        "pad_token": "<|im_end|>",
        "unk_token": "<unk>",
        "model_max_length": 1000000000000000019884624838656,
        "clean_up_tokenization_spaces": False,
        "tokenizer_class": "PreTrainedTokenizerFast",
        "chat_template": (
            "{% for message in messages %}"
            "{% if message['role'] == 'system' %}"
            "<|im_start|>system\n{{ message['content'] }}<|im_end|>\n"
            "{% elif message['role'] == 'user' %}"
            "<|im_start|>user\n{{ message['content'] }}<|im_end|>\n"
            "{% elif message['role'] == 'assistant' %}"
            "<|im_start|>assistant\n{{ message['content'] }}<|im_end|>\n"
            "{% endif %}"
            "{% endfor %}"
            "{% if add_generation_prompt %}"
            "{{ '<|im_start|>assistant\n' }}"
            "{% endif %}"
        )
    }

    # 保存主配置文件
    with open(os.path.join(save_dir, "tokenizer_config.json"), "w", encoding="utf-8") as
f:
        json.dump(config, f, ensure_ascii=False, indent=4)

    # 创建special_tokens_map.json
    special_tokens_map = {
        "bos_token": "<|im_start|>",
        "eos_token": "<|im_end|>",
```

```
        "unk_token": "<unk>",
        "pad_token": "<|im_end|>",
        "additional_special_tokens": ["<s>", "</s>"]
    }
    with open(os.path.join(save_dir, "special_tokens_map.json"), "w", encoding="utf-8") as
f:
        json.dump(special_tokens_map, f, ensure_ascii=False, indent=4)

def train_tokenizer(data_path: str, save_dir: str, vocab_size: int = 8192) -> None:
    """训练并保存自定义tokenizer"""
    os.makedirs(save_dir, exist_ok=True)

    # 初始化tokenizer
    tokenizer = Tokenizer(models.BPE(unk_token="<unk>"))
    tokenizer.normalizer = NFKC()  # 添加文本规范化
    tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel(add_prefix_space=False)
    tokenizer.decoder = decoders.ByteLevel()

    # 配置特殊token
    special_tokens = [
        "<unk>",
        "<s>",
        "</s>",
        "<|im_start|>",
        "<|im_end|>"
    ]

    # 配置训练器
    trainer = trainers.BpeTrainer(
        vocab_size=vocab_size,
        special_tokens=special_tokens,
        min_frequency=2,  # 提高低频词过滤
        show_progress=True,
        initial_alphabet=pre_tokenizers.ByteLevel.alphabet()
    )

    # 训练tokenizer
    print(f"Training tokenizer with data from {data_path}")
    texts = read_texts_from_jsonl(data_path)
    tokenizer.train_from_iterator(texts, trainer=trainer,
length=os.path.getsize(data_path))

    # 验证特殊token映射
    try:
        assert tokenizer.token_to_id("<unk>") == 0
        assert tokenizer.token_to_id("<s>") == 1
        assert tokenizer.token_to_id("</s>") == 2
        assert tokenizer.token_to_id("<|im_start|>") == 3
        assert tokenizer.token_to_id("<|im_end|>") == 4
    except AssertionError as e:
        print("Special tokens mapping error:", e)
        raise
```

```
    # 保存tokenizer文件
    tokenizer.save(os.path.join(save_dir, "tokenizer.json"))

    # 创建配置文件
    create_tokenizer_config(save_dir)
    print(f"Tokenizer saved to {save_dir}")

def eval_tokenizer(tokenizer_path: str) -> None:
    """评估tokenizer功能"""
    try:
        tokenizer = AutoTokenizer.from_pretrained(tokenizer_path)
    except Exception as e:
        print(f"Error loading tokenizer: {e}")
        return

    # 测试基本属性
    print("\n=== Tokenizer基本信息 ===")
    print(f"Vocab size: {len(tokenizer)}")
    print(f"Special tokens: {tokenizer.all_special_tokens}")
    print(f"Special token IDs: {tokenizer.all_special_ids}")

    # 测试聊天模板
    messages = [
        {"role": "system", "content": "你是一个AI助手。"},
        {"role": "user", "content": "How are you?"},
        {"role": "assistant", "content": "I'm fine, thank you. and you?"},
        {"role": "user", "content": "I'm good too."},
        {"role": "assistant", "content": "That's great to hear!"},
    ]

    print("\n=== 聊天模板测试 ===")
    prompt = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        # add_generation_prompt=True
    )
    print("Generated prompt:\n", prompt, sep="")

    # 测试编码解码
    print("\n=== 编码解码测试 ===")
    encoded = tokenizer(prompt, truncation=True, max_length=256)
    decoded = tokenizer.decode(encoded["input_ids"], skip_special_tokens=False)
    print("Decoded text matches original:", decoded == prompt)

    # 测试特殊token处理
    print("\n=== 特殊token处理 ===")
    test_text = "<|im_start|>user\nHello<|im_end|>"
    encoded = tokenizer(test_text).input_ids
    decoded = tokenizer.decode(encoded)
    print(f"Original: {test_text}")
    print(f"Decoded:  {decoded}")
```

```
    print("Special tokens preserved:", decoded == test_text)

def main():
    # 配置路径
    data_path = "your data path"
    save_dir = "tokenizer_k"

    # 训练tokenizer
    train_tokenizer(
        data_path=data_path,
        save_dir=save_dir,
        vocab_size=6144
    )

    # 评估tokenizer
    eval_tokenizer(save_dir)

if __name__ == '__main__':
    main()
```

训练完成之后可以可以使用 `eval_tokenizer()` 测试 Tokenizer 的功能，确保 Tokenizer 正常工作。在这个函数中，我们首先加载训练好的 Tokenizer，然后测试了 Tokenizer 的基本属性、聊天模板、编码解码等功能。这些测试可以帮助我们验证 Tokenizer 的正确性，确保它能够正常工作。正确的输出为：

OUT:

```
=== Tokenizer基本信息 ===
Vocab size: 6144
Special tokens: ['<|im_start|>', '<|im_end|>', '<unk>', '<s>', '</s>']
Special token IDs: [3, 4, 0, 1, 2]

=== 聊天模板测试 ===
Generated prompt:
<|im_start|>system
你是一个AI助手。<|im_end|>
<|im_start|>user
How are you?<|im_end|>
<|im_start|>assistant
I'm fine, thank you. and you?<|im_end|>
<|im_start|>user
I'm good too.<|im_end|>
<|im_start|>assistant
That's great to hear!<|im_end|>


=== 编码解码测试 ===
Decoded text matches original: False

=== 特殊token处理 ===
Original: <|im_start|>user
Hello<|im_end|>
```

```
Decoded:  <|im_start|> user
Hello<|im_end|>
Special tokens preserved: False
```

5.3.2 Dataset

PretrainDataset

在将数据送入到模型之前，我们还需要进行一些处理用于将文本数据转化为模型能够理解的Token。在这里我们使用的是Pytorch的Dataset类，用于加载数据集。我们定义了一个`PretrainDataset`类，用于加载已预处理好的数据集。我们继承了`torch.utils.data.IterableDataset`来定义该数据集，这使得我们可以更灵活、高效地处理数据。

```python
from torch.utils.data import Dataset

class PretrainDataset(Dataset):
    def __init__(self, data_path, tokenizer, max_length=512):
        super().__init__()
        self.data_path = data_path
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.padding = 0
        with open(data_path, 'r', encoding='utf-8') as f:
            self.data = f.readlines()

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index: int):
        sample = json.loads(self.data[index])
        text = f"{self.tokenizer.bos_token}{sample['text']}"
        input_id = self.tokenizer(text).data['input_ids'][:self.max_length]
        text_len = len(input_id)
        # 没满最大长度的剩余部分
        padding_len = self.max_length - text_len
        input_id = input_id + [self.padding] * padding_len
        # 0表示不计算损失
        loss_mask = [1] * text_len + [0] * padding_len

        input_id = np.array(input_id)
        X = np.array(input_id[:-1]).astype(np.int64)
        Y = np.array(input_id[1:]).astype(np.int64)
        loss_mask = np.array(loss_mask[1:]).astype(np.int64)
        return torch.from_numpy(X), torch.from_numpy(Y), torch.from_numpy(loss_mask)
```

在以上代码和图5.1可以看出，`Pretrain Dataset`主要是将`text`通过`tokenizer`转换成`input_id`，然后将`input_id`拆分成`X`和`Y`，其中`X`为`input_id`的前 n-1 个元素，`Y`为`input_id`的后 n-1 个元素。`loss_mask`主要是用来标记哪些位置需要计算损失，哪些位置不需要计算损失。

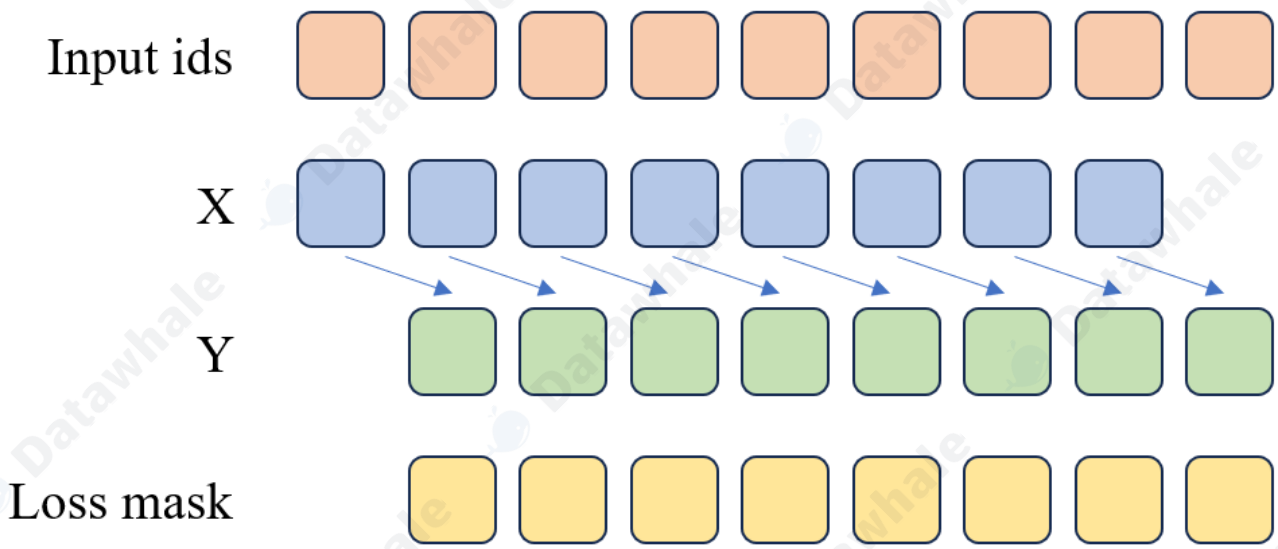


图5.1 预训练损失函数计算

图中示例展示了当`max_length=9`时的处理过程：

- **输入序列**：`[BOS, T1, T2, T3, T4, T5, T6, T7, EOS]`
- **样本拆分**：
  - X：`[BOS, T1, T2, T3, T4, T5, T6, T7]` → 模型输入上下文
  - Y：`[T1, T2, T3, T4, T5, T6, T7, EOS]` → 模型预测目标
- **损失掩码**：
  - 有效位置：`[0, 1, 1, 1, 1, 1, 1, 1, 1]` → 仅对T1-EOS计算损失

SFTDataset

`SFTDataset` 其实是一个多轮对话数据集，我们的目标是让模型学会如何进行多轮对话。在这个阶段我们的输入是上一轮的对话内容，输出是当前轮的对话内容。

```
class SFTDataset(Dataset):
    def __init__(self, data_path, tokenizer, max_length=512):
        super().__init__()
        self.data_path = data_path
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.padding = 0
        with open(data_path, 'r', encoding='utf-8') as f:
            self.data = f.readlines()

    def __len__(self):
        return len(self.data)

    def generate_loss_mask(self, input_ids):
        # 生成 loss mask, 0 表示不计算损失, 1 表示计算损失
        mask = [0] * len(input_ids)
        a_sequence = [3, 1074, 537, 500, 203]  # <|im_start|>assistant\n
```

```
        a_length = len(a_sequence)
        n = len(input_ids)
        i = 0

        while i <= n - a_length:
            # 检查当前位置是否匹配目标子序列
            match = True
            for k in range(a_length):
                if input_ids[i + k] != a_sequence[k]:
                    match = False
                    break
            if match:
                # 从子序列结束的位置开始查找第一个4, 4 为 <|im_end|> EOS id
                j = None
                for idx in range(i + a_length, n):
                    if input_ids[idx] == 4:
                        j = idx
                        break
                if j is not None:
                    start = i + a_length
                    end = j  # 结束位置设为j（包含4）
                    # 标记区间为1（包括start到end）
                    if start <= end:
                        for pos in range(start, end + 1):
                            if pos < len(mask):
                                mask[pos] = 1
                # 跳过当前子序列，避免重叠匹配
                i += a_length
            else:
                i += 1
        return mask

    def __getitem__(self, index: int):
        sample = json.loads(self.data[index])
        text = self.tokenizer.apply_chat_template(sample, tokenize=False,
add_generation_prompt=False)
        input_id = self.tokenizer(text).data['input_ids'][:self.max_length]
        text_len = len(input_id)
        # 没满最大长度的剩余部分
        padding_len = self.max_length - text_len
        input_id = input_id + [self.padding] * padding_len
        # 0表示不计算损失
        loss_mask = self.generate_loss_mask(input_id)

        input_id = np.array(input_id)
        X = np.array(input_id[:-1]).astype(np.int64)
        Y = np.array(input_id[1:]).astype(np.int64)
        loss_mask = np.array(loss_mask[1:]).astype(np.int64)
        return torch.from_numpy(X), torch.from_numpy(Y), torch.from_numpy(loss_mask)
```

在 SFT 阶段，这里使用的是多轮对话数据集，所以就需要区分哪些位置需要计算损失，哪些位置不需要计算损失。在上面的代码中，我使用了一个 `generate_loss_mask` 函数来生成 `loss_mask`。这个函数主要是用来生成 `loss_mask`，其中 `loss_mask` 的生成规则是：当遇到 `|<im_start|>assistant\n` 时，就开始计算损失，直到遇到 `|<im_end|>` 为止。这样就可以保证我们的模型在 SFT 阶段只计算当前轮的对话内容，如图5.2所示。

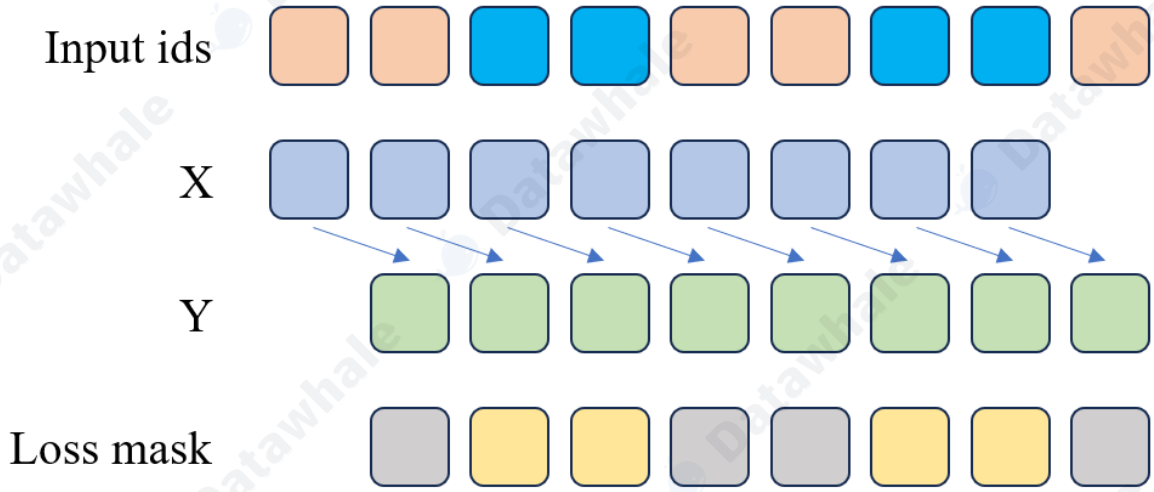


图5.2 SFT 损失函数计算

可以看到，其实 SFT Dataset 和 Pretrain Dataset 的 `X` 和 `Y` 是一样的，只是在 SFT Dataset 中我们需要生成一个 `loss_mask` 来标记哪些位置需要计算损失，哪些位置不需要计算损失。图中 `Input ids` 中的蓝色小方格就是AI的回答，所以是需要模型学习的地方。所以在 `loss_mask` 中，蓝色小方格对应的位置是黄色，其他位置是灰色。在代码 `loss_mask` 中的 1 对应的位置计算损失，0 对应的位置不计算损失。

5.3.3 预训练

在数据预处理完成后，我们就可以开始训练模型了。我们使用的模型是一个和LLama2结构一样的 Decoder only Transformer模型，使用Pytorch实现。相关代码在 `code/k_model.py` 文件中。此处不再赘述，源码中有详细的中文注释，且我们在之前的文章中也有详细的介绍。

在模型这一部分可以重点看一下生成式模型是如何实现生成token的，可以查看 `k_model.py` 文件中的 `Transforerm` 类中的 `generate` 方法。

```
@torch.inference_mode()
    def generate(self, idx, stop_id=None, max_new_tokens=256, temperature=1.0,
top_k=None):
        """
        给定输入序列 idx（形状为 (bz,seq_len) 的长整型张量），通过多次生成新 token 来完成序列。
        在 model.eval() 模式下运行。效率较低的采样版本，没有使用键k/v cache。
        """
        index = idx.shape[1]
        for _ in range(max_new_tokens):
            # 如果序列上下文过长，截断它到最大长度
            idx_cond = idx if idx.size(1) <= self.args.max_seq_len else idx[:, -
self.args.max_seq_len:]

            # 前向传播获取序列中最后一个位置的 logits
            logits = self(idx_cond).logits
            logits = logits[:, -1, :] # 只保留最后一个时间步的输出
```

```
            if temperature == 0.0:
                # 选择最有可能的索引
                _, idx_next = torch.topk(logits, k=1, dim=-1)
            else:
                # 缩放 logits 并应用 softmax
                logits = logits / temperature
                if top_k is not None:
                    v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
                    logits[logits < v[:, [-1]]] = -float('Inf')
                probs = F.softmax(logits, dim=-1)
                idx_next = torch.multinomial(probs, num_samples=1)


            if idx_next == stop_id:
                break

            # 将采样的索引添加到序列中并继续
            idx = torch.cat((idx, idx_next), dim=1)

        return idx[:, index:] # 只返回生成的token
```

在 `generate` 方法中，我们首先获取序列中最后一个位置的 `logits`，然后基于这些 `logits` 生成新的 `token`。接着，生成的新 `token` 会被添加到序列中，模型随后会继续生成下一个 `token`。通过这种迭代过程，我们能够生成完整的文本。

接下来就是最重要的部分，训练模型!

> 注：在使用下面代码进行模型训练时，需要指定 `--data_path` 参数为预处理好的数据集路径，例如 `--data_path seq_monkey_datawhale.jsonl`，也需要指定要用哪几张GPU进行训练，例如 `--gpus 0,1`。

```
def get_lr(it, all):
    warmup_iters = args.warmup_iters
    lr_decay_iters = all
    min_lr = args.learning_rate / 10

    if it < warmup_iters:
        return args.learning_rate * it / warmup_iters

    if it > lr_decay_iters:
        return min_lr

    decay_ratio = (it - warmup_iters) / (lr_decay_iters - warmup_iters)
    assert 0 <= decay_ratio <= 1
    coeff = 0.5 * (1.0 + math.cos(math.pi * decay_ratio))
    return min_lr + coeff * (args.learning_rate - min_lr)

def train_epoch(epoch):
    start_time = time.time()
    for step, (X, Y, loss_mask) in enumerate(train_loader):
        X = X.to(args.device)
```

```
        Y = Y.to(args.device)
        loss_mask = loss_mask.to(args.device)

        lr = get_lr(epoch * iter_per_epoch + step, args.epochs * iter_per_epoch)
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr

        with ctx:
            out = model(X, Y)
            loss = out.last_loss / args.accumulation_steps
            loss_mask = loss_mask.view(-1)
            loss = torch.sum(loss * loss_mask) / loss_mask.sum()

        scaler.scale(loss).backward()

        if (step + 1) % args.accumulation_steps == 0:
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), args.grad_clip)

            scaler.step(optimizer)
            scaler.update()

            optimizer.zero_grad(set_to_none=True)

        if step % args.log_interval == 0:
            spend_time = time.time() - start_time
            Logger(
                'Epoch:[{}/{}]({}/{}) loss:{:.3f} lr:{:.7f} epoch_Time:{}min:'.format(
                    epoch + 1,
                    args.epochs,
                    step,
                    iter_per_epoch,
                    loss.item() * args.accumulation_steps,
                    optimizer.param_groups[-1]['lr'],
                    spend_time / (step + 1) * iter_per_epoch // 60 - spend_time // 60))
            if args.use_swanlab:
                swanlab.log({
                    "loss": loss.item() * args.accumulation_steps,
                    "lr": optimizer.param_groups[-1]['lr']
                })

        if (step + 1) % args.save_interval == 0:
            model.eval()
            ckp =
f'{args.save_dir}/pretrain_{lm_config.dim}_{lm_config.n_layers}_{lm_config.vocab_size}.pth
'

            # 处理多卡保存
            state_dict = model.module.state_dict() if isinstance(model,
torch.nn.DataParallel) else model.state_dict()
            torch.save(state_dict, ckp)
            model.train()
```

```
        if (step + 1) % 20000 == 0:
            model.eval()
            ckp =
f'{args.save_dir}/pretrain_{lm_config.dim}_{lm_config.n_layers}_{lm_config.vocab_size}_ste
p{step+1}.pth'

            state_dict = model.module.state_dict() if isinstance(model,
torch.nn.DataParallel) else model.state_dict()
            torch.save(state_dict, ckp)
            model.train()


def init_model():
    def count_parameters(model):
        return sum(p.numel() for p in model.parameters() if p.requires_grad)

    tokenizer = AutoTokenizer.from_pretrained('./tokenizer_k/')

    model = Transformer(lm_config)

    # 多卡初始化
    num_gpus = torch.cuda.device_count()
    if num_gpus > 1:
        Logger(f"Using {num_gpus} GPUs with DataParallel!")
        model = torch.nn.DataParallel(model)

    model = model.to(args.device)
    Logger(f'LLM总参数量：{count_parameters(model) / 1e6:.3f} 百万')
    return model, tokenizer

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Tiny-LLM Pretraining")
    parser.add_argument("--out_dir", type=str, default="output", help="Output directory")
    parser.add_argument("--epochs", type=int, default=1, help="Number of epochs")
    parser.add_argument("--batch_size", type=int, default=64, help="Batch size")
    parser.add_argument("--learning_rate", type=float, default=2e-4, help="Learning rate")
    parser.add_argument("--device", type=str, default="cuda:0" if
torch.cuda.is_available() else "cpu", help="Device to use")
    parser.add_argument("--dtype", type=str, default="bfloat16", help="Data type")
    parser.add_argument("--use_swanlab", type=bool, default=True, help="Use Weights &
Biases")
    parser.add_argument("--num_workers", type=int, default=8, help="Number of workers for
data loading")
    parser.add_argument("--data_path", type=str, default="", help="Path to training data")
    parser.add_argument("--accumulation_steps", type=int, default=8, help="Gradient
accumulation steps")
    parser.add_argument("--grad_clip", type=float, default=1.0, help="Gradient clipping
threshold")
    parser.add_argument("--warmup_iters", type=int, default=0, help="Number of warmup
iterations")
    parser.add_argument("--log_interval", type=int, default=100, help="Logging interval")
```

```
    parser.add_argument("--save_interval", type=int, default=1000, help="Model saving
interval")
    # 添加多卡参数
    parser.add_argument("--gpus", type=str, default='0,1', help="Comma-separated GPU IDs
(e.g. '0,1,2')")

    args = parser.parse_args()

    # 设置可见GPU
    if args.gpus is not None:
        os.environ["CUDA_VISIBLE_DEVICES"] = args.gpus
        # 自动设置主设备为第一个GPU
        if torch.cuda.is_available():
            args.device = "cuda:0"
        else:
            args.device = "cpu"

    if args.use_swanlab:
        swanlab.login(api_key='your key')
        run = swanlab.init(
            project="Tiny-LLM",
            experiment_name="Pretrain-215M",
            config=args,
        )

    lm_config = ModelConfig(
        dim=1024,
        n_layers=18,
    )
    max_seq_len = lm_config.max_seq_len
    args.save_dir = os.path.join(args.out_dir)
    os.makedirs(args.save_dir, exist_ok=True)
    os.makedirs(args.out_dir, exist_ok=True)
    torch.manual_seed(42)
    device_type = "cuda" if "cuda" in args.device else "cpu"

    ctx = nullcontext() if device_type == "cpu" else torch.cuda.amp.autocast()

    model, tokenizer = init_model()

    train_ds = PretrainDataset(args.data_path, tokenizer, max_length=max_seq_len)
    train_loader = DataLoader(
        train_ds,
        batch_size=args.batch_size,
        pin_memory=True,
        drop_last=False,
        shuffle=True,
        num_workers=args.num_workers
    )

    scaler = torch.cuda.amp.GradScaler(enabled=(args.dtype in ['float16', 'bfloat16']))
    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)
```

```
    iter_per_epoch = len(train_loader)
    for epoch in range(args.epochs):
        train_epoch(epoch)
```

5.3.4 SFT 训练

SFT 训练和预训练的代码基本一样，只是导入的 Dataset 不一样。在这里我们使用的是 SFTDataset，用于多轮对话的训练。

```
import os
import platform
import argparse
import time
import warnings
import math
import pandas as pd
import torch
from torch import optim
from torch.utils.data import DataLoader
from contextlib import nullcontext

from transformers import AutoTokenizer

from k_model import ModelConfig, Transformer
from dataset import SFTDataset

import swanlab

warnings.filterwarnings('ignore')


def Logger(content):
    print(content)

def get_lr(it, all):
    warmup_iters = args.warmup_iters
    lr_decay_iters = all
    min_lr = args.learning_rate / 10

    if it < warmup_iters:
        return args.learning_rate * it / warmup_iters

    if it > lr_decay_iters:
        return min_lr

    decay_ratio = (it - warmup_iters) / (lr_decay_iters - warmup_iters)
    assert 0 <= decay_ratio <= 1
    coeff = 0.5 * (1.0 + math.cos(math.pi * decay_ratio))
    return min_lr + coeff * (args.learning_rate - min_lr)
```

```
def train_epoch(epoch):
    start_time = time.time()
    for step, (X, Y, loss_mask) in enumerate(train_loader):
        X = X.to(args.device)
        Y = Y.to(args.device)
        loss_mask = loss_mask.to(args.device)

        lr = get_lr(epoch * iter_per_epoch + step, args.epochs * iter_per_epoch)
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr

        with ctx:
            out = model(X, Y)
            loss = out.last_loss / args.accumulation_steps
            loss_mask = loss_mask.view(-1)
            loss = torch.sum(loss * loss_mask) / loss_mask.sum()

        scaler.scale(loss).backward()

        if (step + 1) % args.accumulation_steps == 0:
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), args.grad_clip)

            scaler.step(optimizer)
            scaler.update()

            optimizer.zero_grad(set_to_none=True)

        if step % args.log_interval == 0:
            spend_time = time.time() - start_time
            Logger(
                'Epoch:[{}/{}]({}/{}) loss:{:.3f} lr:{:.7f} epoch_Time:{}min:'.format(
                    epoch + 1,
                    args.epochs,
                    step,
                    iter_per_epoch,
                    loss.item() * args.accumulation_steps,
                    optimizer.param_groups[-1]['lr'],
                    spend_time / (step + 1) * iter_per_epoch // 60 - spend_time // 60))
            if args.use_swanlab:
                swanlab.log({
                    "loss": loss.item() * args.accumulation_steps,
                    "lr": optimizer.param_groups[-1]['lr']
                })

        if (step + 1) % args.save_interval == 0:
            model.eval()
            ckp =
f'{args.save_dir}/sft_dim{lm_config.dim}_layers{lm_config.n_layers}_vocab_size{lm_config.v
ocab_size}.pth'

            # 处理多卡保存
```

```
            state_dict = model.module.state_dict() if isinstance(model,
torch.nn.DataParallel) else model.state_dict()
            torch.save(state_dict, ckp)
            model.train()

        if (step + 1) % 20000 == 0:
            model.eval()
            ckp =
f'{args.save_dir}/sft_dim{lm_config.dim}_layers{lm_config.n_layers}_vocab_size{lm_config.v
ocab_size}_step{step+1}.pth'

            state_dict = model.module.state_dict() if isinstance(model,
torch.nn.DataParallel) else model.state_dict()
            torch.save(state_dict, ckp)
            model.train()


def init_model():
    def count_parameters(model):
        return sum(p.numel() for p in model.parameters() if p.requires_grad)

    tokenizer = AutoTokenizer.from_pretrained('./tokenizer_k/')

    model = Transformer(lm_config)

    ckp = './base_monkey_215M/pretrain_1024_18_6144.pth'
    state_dict = torch.load(ckp, map_location=args.device)
    unwanted_prefix = '_orig_mod.'
    for k, v in list(state_dict.items()):
        if k.startswith(unwanted_prefix):
            state_dict[k[len(unwanted_prefix):]] = state_dict.pop(k)
    model.load_state_dict(state_dict, strict=False)

    # 多卡初始化
    num_gpus = torch.cuda.device_count()
    if num_gpus > 1:
        Logger(f"Using {num_gpus} GPUs with DataParallel!")
        model = torch.nn.DataParallel(model)

    model = model.to(args.device)
    Logger(f'LLM总参数量：{count_parameters(model) / 1e6:.3f} 百万')
    return model, tokenizer


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Tiny-LLM Pretraining")
    parser.add_argument("--out_dir", type=str, default="output", help="Output directory")
    parser.add_argument("--epochs", type=int, default=1, help="Number of epochs")
    parser.add_argument("--batch_size", type=int, default=64, help="Batch size")
    parser.add_argument("--learning_rate", type=float, default=2e-4, help="Learning rate")
    parser.add_argument("--device", type=str, default="cuda:0" if
torch.cuda.is_available() else "cpu", help="Device to use")
```

```
    parser.add_argument("--dtype", type=str, default="bfloat16", help="Data type")
    parser.add_argument("--use_swanlab", type=bool, default=True, help="Use Weights &
Biases")
    parser.add_argument("--num_workers", type=int, default=4, help="Number of workers for
data loading")
    parser.add_argument("--data_path", type=str, default="", help="Path to training data")
    parser.add_argument("--accumulation_steps", type=int, default=4, help="Gradient
accumulation steps")
    parser.add_argument("--grad_clip", type=float, default=1.0, help="Gradient clipping
threshold")
    parser.add_argument("--warmup_iters", type=int, default=0, help="Number of warmup
iterations")
    parser.add_argument("--log_interval", type=int, default=100, help="Logging interval")
    parser.add_argument("--save_interval", type=int, default=1000, help="Model saving
interval")
    # 添加多卡参数
    parser.add_argument("--gpus", type=str, default='0,1', help="Comma-separated GPU IDs
(e.g. '0,1,2')")

    args = parser.parse_args()

    # 设置可见GPU
    if args.gpus is not None:
        os.environ["CUDA_VISIBLE_DEVICES"] = args.gpus
        # 自动设置主设备为第一个GPU
        if torch.cuda.is_available():
            args.device = "cuda:0"
        else:
            args.device = "cpu"

    if args.use_swanlab:
        swanlab.login(api_key='your key')
        run = swanlab.init(
            project="Tiny-LLM",
            experiment_name="BelleGropu-sft-215M",
            config=args,
        )

    lm_config = ModelConfig(
        dim=1024,
        n_layers=18,
    )
    max_seq_len = lm_config.max_seq_len
    args.save_dir = os.path.join(args.out_dir)
    os.makedirs(args.save_dir, exist_ok=True)
    os.makedirs(args.out_dir, exist_ok=True)
    torch.manual_seed(42)
    device_type = "cuda" if "cuda" in args.device else "cpu"

    ctx = nullcontext() if device_type == "cpu" else torch.cuda.amp.autocast()

    model, tokenizer = init_model()
```

```
    train_ds = SFTDataset(args.data_path, tokenizer, max_length=max_seq_len)
    train_loader = DataLoader(
        train_ds,
        batch_size=args.batch_size,
        pin_memory=True,
        drop_last=False,
        shuffle=True,
        num_workers=args.num_workers
    )

    scaler = torch.cuda.amp.GradScaler(enabled=(args.dtype in ['float16', 'bfloat16']))
    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)

    iter_per_epoch = len(train_loader)
    for epoch in range(args.epochs):
        train_epoch(epoch)
```

5.3.4 使用模型生成文本

在模型训练完成后，会在`output`目录下生成模型文件，这个文件就是我们训练好的模型。我们可以使用以下命令生成文本。

```
python model_sample.py
```

我们来看下`model_sample.py`文件中的代码，这个文件中定义了一个`TextGenerator`类，用于生成文本。

```
class TextGenerator:
    def __init__(self,
                 checkpoint='out/SkyWork_pretrain_768_12_6144.pth',  # 模型检查点路径
                 tokenizer_model_path='./tokenizer_k/',  # 分词器模型路径
                 seed=42,  # 随机种子，确保可重复性
                 device=None,  # 设备，优先使用 CUDA，如果没有可用的 CUDA，则使用 CPU
                 dtype="bfloat16"):  # 数据类型，默认为 float32，可以选择 float16 或 bfloat16
        """
        初始化 TextGenerator 类，加载模型、设置设备和分词器等。
        """
        # 模型加载配置
        self.checkpoint = checkpoint  # 保存的模型检查点路径
        self.tokenizer_model_path = tokenizer_model_path  # 分词器模型文件路径
        self.seed = seed  # 随机数种子，用于生成的可重复性
        self.device = device or ('cuda:0' if torch.cuda.is_available() else 'cpu')  # 根据硬件条件选择设备
        self.dtype = dtype  # 模型的浮点数类型
        self.device_type = 'cuda' if 'cuda' in self.device else 'cpu'  # 判断当前设备是否为CUDA

        # 设置随机种子，确保生成的可重复性
        torch.manual_seed(seed)  # 设置 CPU 随机种子
        torch.cuda.manual_seed(seed)  # 设置 CUDA 随机种子
```

```
        torch.backends.cuda.matmul.allow_tf32 = True  # 允许 CUDA 使用 TF32 精度进行矩阵乘法运算
        torch.backends.cudnn.allow_tf32 = True  # 允许 cuDNN 使用 TF32 精度加速

        # 根据 dtype 选择适当的自动混合精度上下文
        ptdtype = {'float32': torch.float32, 'bfloat16': torch.bfloat16, 'float16': torch.float16}[self.dtype]
        self.ctx = nullcontext() if self.device_type == 'cpu' else torch.amp.autocast(device_type=self.device_type, dtype=ptdtype)

        # 加载模型检查点文件
        checkpoint_dict = torch.load(self.checkpoint, map_location=self.device)  # 加载模型参数 # 初始化模型参数
        self.model = Transformer(ModelConfig(dim=1024, n_layers=18))  # 实例化 Transformer 模型
        sunwanted_prefix = '_orig_mod.'
        for k, v in list(checkpoint_dict.items()):
            if k.startswith(sunwanted_prefix):
                checkpoint_dict[k[len(sunwanted_prefix):]] = checkpoint_dict.pop(k)
        self.model.load_state_dict(checkpoint_dict, strict=False)

        # 计算模型参数量
        num_params = sum(p.numel() for p in self.model.parameters() if p.requires_grad)
        print(f"Model has {num_params / 1e6:.3f} M parameters.")
        # 设置模型为评估模式（evaluation mode），防止训练模式下的 dropout 等操作影响结果
        self.model.eval()
        # 将模型放置到正确的设备上（GPU 或 CPU）
        self.model.to(self.device)
        # 初始化分词器
        self.tokenizer = AutoTokenizer.from_pretrained(self.tokenizer_model_path)  # 根据指定的路径加载分词器

    def chat_template(self, prompt):
        message = [
            {"role": "system", "content": "你是一个AI助手。"},
            {"role": "user", "content": prompt}
        ]
        return self.tokenizer.apply_chat_template(message, tokenize=False, add_generation_prompt=True)

    def sft_sample(self,
                start="Hello!",  # 生成文本的起始提示词，可以是任意字符串
                num_samples=3,  # 生成样本的数量，默认生成 3 个样本
                max_new_tokens=256,  # 每个样本生成的最大 token 数，默认最多生成 256 个 token
                temperature=0.7,  # 控制生成的随机性，1.0 为标准，值越大越随机
                top_k=300):  # 保留概率最高的 top_k 个 token，限制生成时的选择范围
        """
        根据给定的起始文本生成样本。

        :param start: 生成文本的起始提示词
        :param num_samples: 要生成的文本样本数
        :param max_new_tokens: 每个样本生成的最大 token 数
```

```
        :param temperature: 控制生成的随机性，值越小生成越确定，值越大生成越随机
        :param top_k: 限制生成时选择的 token 范围
        :return: 生成的文本样本列表
        """
        start = self.chat_template(start)
        # 将起始文本编码为 token id 序列
        start_ids = self.tokenizer(start).data['input_ids']
        # print('start_ids:', start_ids)
        x = (torch.tensor(start_ids, dtype=torch.long, device=self.device)[None, ...])  # 将编码后的 token id 转为 PyTorch 张量
        generated_texts = []  # 用于保存生成的文本样本
        with torch.no_grad():  # 禁用梯度计算，提升效率
            with self.ctx:  # 进入自动混合精度的上下文（如果是 GPU 并使用 float16 时）
                for k in range(num_samples):  # 循环生成指定数量的样本
                    y = self.model.generate(x, self.tokenizer.eos_token_id, max_new_tokens, temperature=temperature, top_k=top_k)  # 生成文本
                    generated_texts.append(self.tokenizer.decode(y[0].tolist()))  # 解码生成的 token 序列为可读文本
        return generated_texts  # 返回生成的文本样本


    def pretrain_sample(self,
                start="Hello!",  # 生成文本的起始提示词，可以是任意字符串
                num_samples=3,  # 生成样本的数量，默认生成 3 个样本
                max_new_tokens=256,  # 每个样本生成的最大 token 数，默认最多生成 256 个 token
                temperature=0.7,  # 控制生成的随机性，1.0 为标准，值越大越随机
                top_k=300):  # 保留概率最高的 top_k 个 token，限制生成时的选择范围
        """
        根据给定的起始文本生成样本。

        :param start: 生成文本的起始提示词
        :param num_samples: 要生成的文本样本数
        :param max_new_tokens: 每个样本生成的最大 token 数
        :param temperature: 控制生成的随机性，值越小生成越确定，值越大生成越随机
        :param top_k: 限制生成时选择的 token 范围
        :return: 生成的文本样本列表
        """
        # 如果 start 是以 'FILE:' 开头，表示从文件中读取起始文本
        if start.startswith('FILE:'):
            with open(start[5:], 'r', encoding='utf-8') as f:
                start = f.read()  # 读取文件内容作为起始文本

        # 将起始文本编码为 token id 序列
        start_ids = self.tokenizer(start).data['input_ids']
        # print('start_ids:', start_ids)
        x = (torch.tensor(start_ids, dtype=torch.long, device=self.device)[None, ...])  # 将编码后的 token id 转为 PyTorch 张量
        # print(x.shape)
        generated_texts = []  # 用于保存生成的文本样本
        with torch.no_grad():  # 禁用梯度计算，提升效率
            with self.ctx:  # 进入自动混合精度的上下文（如果是 GPU 并使用 float16 时）
                for k in range(num_samples):  # 循环生成指定数量的样本
```

```
                    y = self.model.generate(x, max_new_tokens=max_new_tokens,
temperature=temperature, top_k=top_k)  # 生成文本
                    generated_texts.append(self.tokenizer.decode(y[0].tolist()))  # 解码生
成的 token 序列为可读文本

        return generated_texts  # 返回生成的文本样本
```

最后我们来看一下模型输出的结果：

```
-------------------- SFT Sample --------------------

Model has 215.127 M parameters.

Sample 1:
Question: 你好呀
AI answer: 你好!有什么我可以帮你的吗?
----------------------

Sample 2:
Question: 中国的首都是哪里?
AI answer: 中国的首都是北京。
----------------------

Sample 3:
Question: 1+1等于多少?
AI answer: 1+1等于2。
----------------------
-------------------- Pretrain Sample --------------------

Model has 215.127 M parameters.

Sample 1:
<|im_start|>北京大学是中国最早建立的研究型大学之一,是我国最早设置研究生院的高校之一,是第一、二国教育委员
会师资培训基地;北京大学是第一、二所国立大学,其校名与北京大学相同。
北京大学录取标准:本科三批1万元,本科一批1万元,本科一批2000元,专科一批2000元,高中起点:非本科一批
----------------------

Sample 2:
<|im_start|>中国矿业大学（北京）地球科学与测绘工程学院副教授黄河流域地质学科带头人古建平教授为大家介绍世
界地质变化的概念及工作经验。
古建平教授介绍了最近几年的植物学和地质学的基本概念,尤其是树都黄河、松涛、暗河等都有地质学工作者的身影,其中
树都黄河以分布面积最大,是树都黄河中华砂岩公园的主景区。
黄河内蒙古
----------------------
```

到这里，我们的模型就训绽完成了，恭喜你训练了一个属于你自己的大模型。

> 大家在训练的时候可以将 batch 调的低一些，这样可以减少显存的占用，避免显存不足的问题。当然这样会增加训练时间，可以根据自己的显卡显存大小来调整 batch 的大小。实测 Pretrain batch 为 4 的情况下只需要 7G 显存，训练时长预计 533 小时。作者是在 4卡A100上进行训练的，预训练一共耗时26小时，SFT 阶段在 BelleGroup 350万条中文指令训练 4 小时。

参考资料

[1] Andrej Karpathy. (2023). *llama2.c: Fullstack Llama 2 LLM solution in pure C*. GitHub repository. https://github.com/karpathy/llama2.c

[2] Andrej Karpathy. (2023). *llm.c: GPT-2/GPT-3 pretraining in C/CUDA*. GitHub repository. https://github.com/karpathy/llm.c

[3] Hugging Face. (2023). *Tokenizers documentation*. https://huggingface.co/docs/tokenizers/index

[4] Skywork Team. (2023). *SkyPile-150B: A large-scale bilingual dataset*. Hugging Face dataset. https://huggingface.co/datasets/Skywork/SkyPile-150B

[5] BelleGroup. (2022). *train_3.5M_CN: Chinese dialogue dataset*. Hugging Face dataset. https://huggingface.co/datasets/BelleGroup/train_3.5M_CN

[6] Jingyao Gong. (2023). *minimind: Minimalist LLM implementation*. GitHub repository. https://github.com/jingyaogong/minimind

[7] Mobvoi. (2023). *seq-monkey-data: Llama2 training/inference data*. GitHub repository. https://github.com/mobvoi/seq-monkey-data

第六章 大模型训练流程实践

6.1 模型预训练

在上一章，我们逐步拆解了 LLM 的模型结构及训练过程，从零手写实现了 LLaMA 模型结构及 Pretrain、SFT 全流程，更深入地理解了 LLM 的模型原理及训练细节。但是，在实际应用中，手写实现的 LLM 训练存在以下问题：

- 手写实现 LLM 结构工作量大，难以实时跟进最新模型的结构创新；
- 从零实现的 LLM 训练无法较好地实现多卡分布式训练，训练效率较低；
- 和现有预训练 LLM 不兼容，无法使用预训练好的模型参数

因此，在本章中，我们将介绍目前 LLM 领域的主流训练框架 Transformers，并结合分布式框架 deepspeed、高效微调框架 peft 等主流框架，实践使用 transformers 进行模型 Pretrain、SFT 全流程，更好地对接业界的主流 LLM 技术方案。

6.1.1 框架介绍

Transformers 是由 Hugging Face 开发的 NLP 框架，通过模块化设计实现了对 BERT、GPT、LLaMA、T5、ViT 等上百种主流模型架构的统一支持。通过使用 Transformers，开发者无需重复实现基础网络结构，通过 AutoModel 类即可一键加载任意预训练，图6.1 为 Hugging Face Transformers 课程首页：

图6.1 Hugging Face Transformers

同时，框架内置的 Trainer 类封装了分布式训练的核心逻辑，支持 PyTorch 原生 DDP、DeepSpeed、Megatron-LM 等多种分布式训练策略。通过简单配置训练参数，即可实现数据并行、模型并行、流水线并行的混合并行训练，在 8 卡 A100 集群上可轻松支持百亿参数模型的高效训练。配合 SavingPolicy 和 LoggingCallback 等组件，实现了训练过程的自动化管理。其还支持与 Deepspeed、peft、wandb、Swanlab 等框架进行集成，直接通过参数设置即可无缝对接，从而快速、高效实现 LLM 训练。

对 LLM 时代的 NLP 研究者更为重要的是，HuggingFace 基于 Transformers 框架搭建了其庞大的 AI 社区，开放了数亿个预训练模型参数、25万+不同类型数据集，通过 Transformers、Dataset、Evaluate 等多个框架实现对预训练模型、数据集及评估函数的集成，从而帮助开发者可以便捷地使用任一预训练模型，在开源模型及数据集的基础上便捷地实现个人模型的开发与应用。

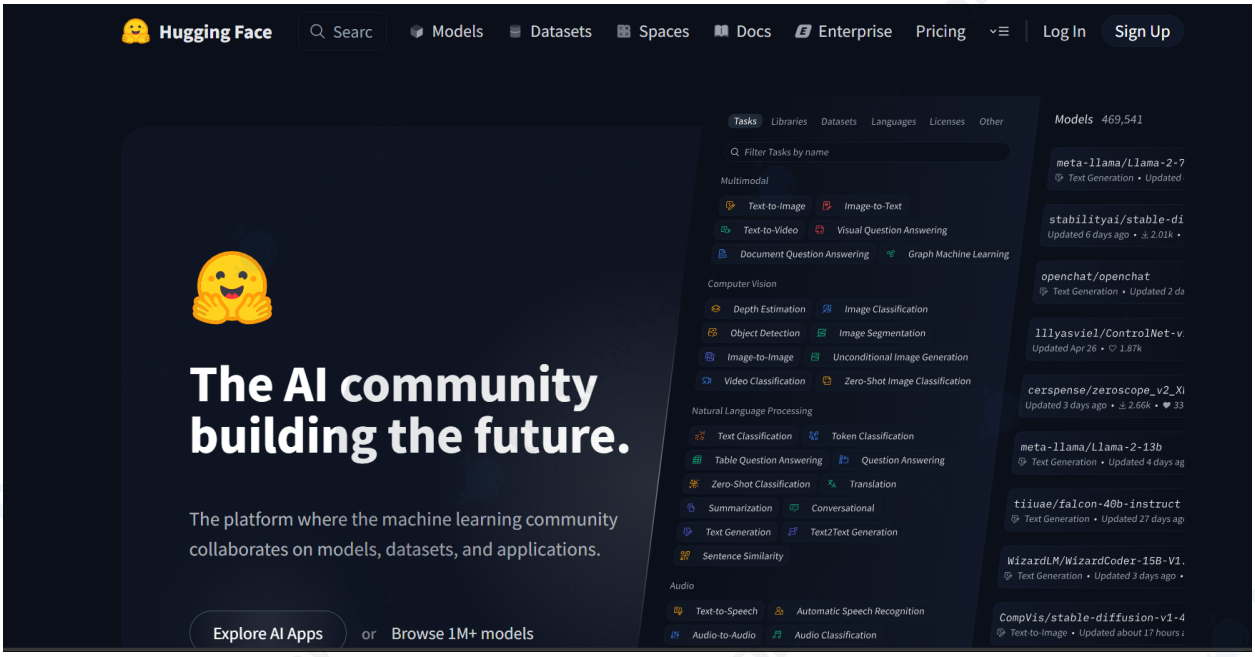


图6.2 Hugging Face Transformers 模型社区

在 LLM 时代，模型结构的调整和重新预训练越来越少，开发者更多的业务应用在于使用预训练好的 LLM 进行 Post Train 和 SFT，来支持自己的下游业务应用。且由于预训练模型体量大，便捷集成 deepspeed 等分布式训练框架逐渐成为 LLM 时代 NLP 模型训练的必备技能。因此，Transformers 已逐步成为学界、业界 NLP 技术的主流框架，不管是企业业务开发还是科研研究，都逐渐首选 Transformers 进行模型实现。同时，新发布的开源 LLM 如 DeepSeek、Qwen 也都会第一时间在 Transformers 社区开放其预训练权重与模型调用 Demo。通过使用 Transformers 框架，可以高效、便捷地完成 LLM 训练及开发，实现工业级的产出交付。接下来，我们就会以 Transformers 框架为基础，介绍如何通过 Transformers 框架实现 LLM 的 Pretrain 及 SFT。

6.1.2 初始化 LLM

我们可以使用 transformers 的 AutoModel 类来直接初始化已经实现好的模型。对于任意预训练模型，其参数中都包含有模型的配置信息。如果是想要从头训练一个 LLM，可以使用一个已有的模型架构来直接初始化。这里，我们以 Qwen-2.5-1.5B的模型架构为例：

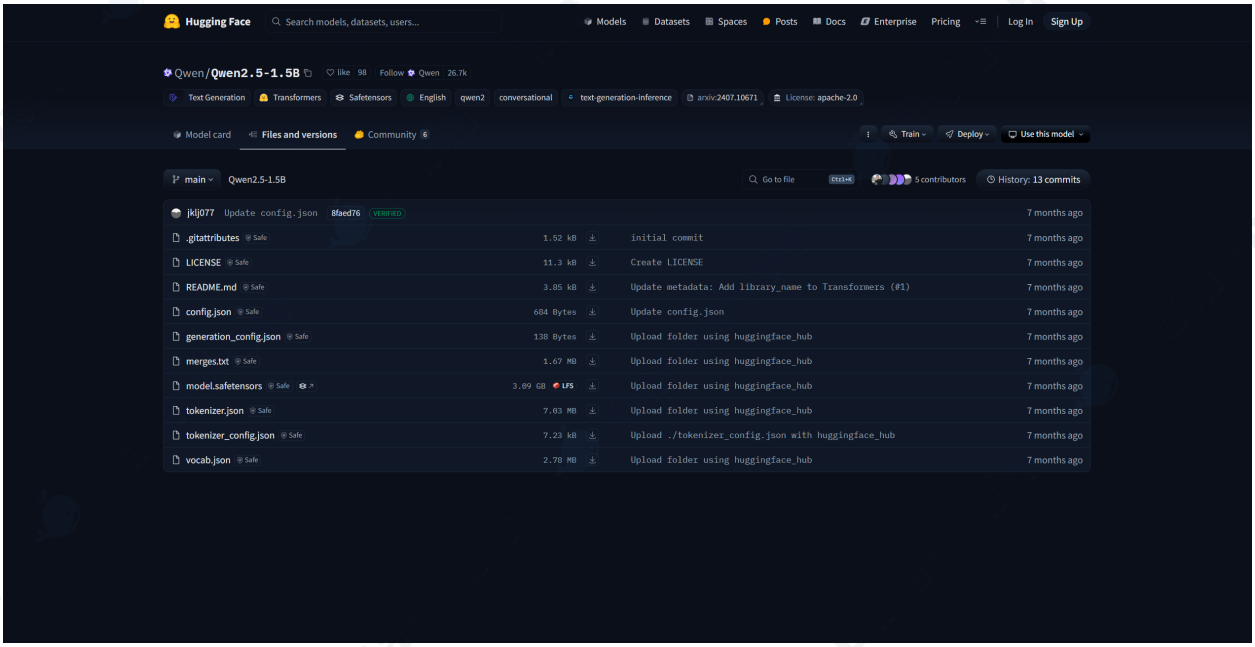


图6.3 Qwen-2.5-1.5B

该界面即为 HuggingFace 社区中的 Qwen-2.5-1.5B 模型参数，其中的 `config.json` 文件即是模型的配置信息，包括了模型的架构、隐藏层大小、模型层数等，如图6.4所示：

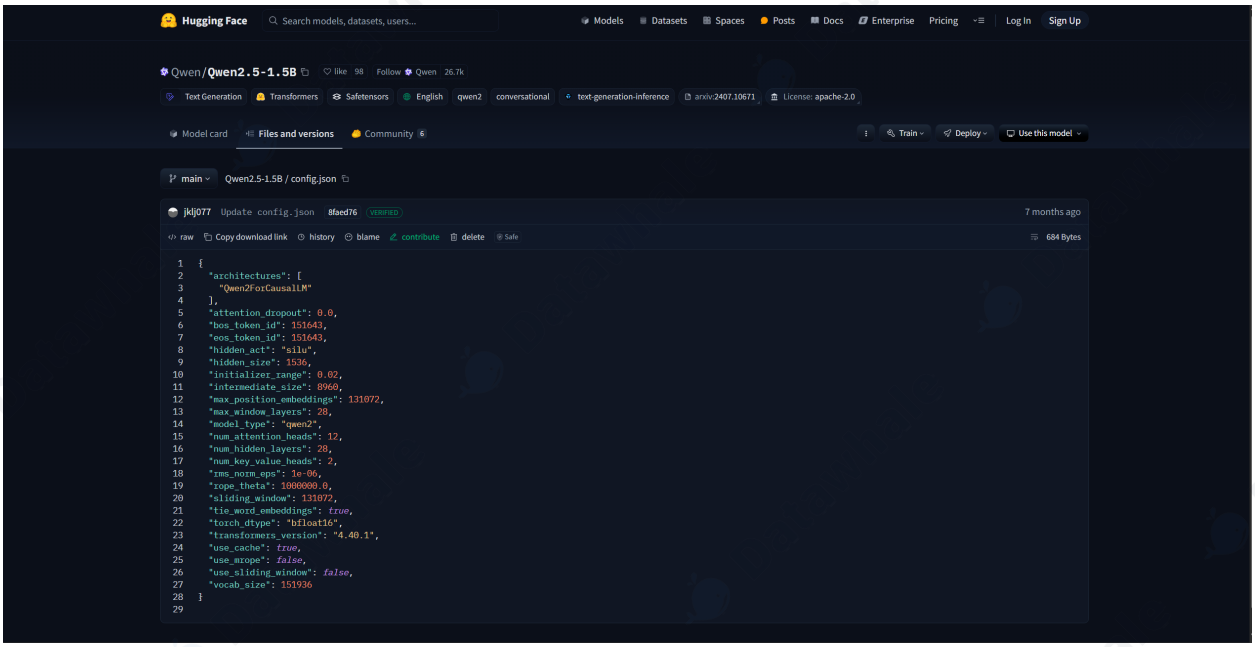


图6.4 Qwen-2.5-1.5B config.json 文件

我们可以沿用该模型的配置信息，初始化一个 Qwen-2.5-1.5B 模型来进行训练，也可以在该配置信息的基础上进行更改，如修改隐藏层大小、注意力头数等，来定制一个模型结构。HuggingFace 提供了 Python 工具来便捷下载想使用的模型参数：

```
import os
# 设置环境变量，此处使用 HuggingFace 镜像网站
os.environ['HF_ENDPOINT'] = 'https://hf-mirror.com'
# 下载模型
os.system('huggingface-cli download --resume-download Qwen/Qwen2.5-1.5B --local-dir your_local_dir')
```

如图6.5，此处的 “Qwen/Qwen2.5-1.5B”即为要下载模型的标识符，对于其他模型，可以直接复制 HuggingFace 上的模型名即可：

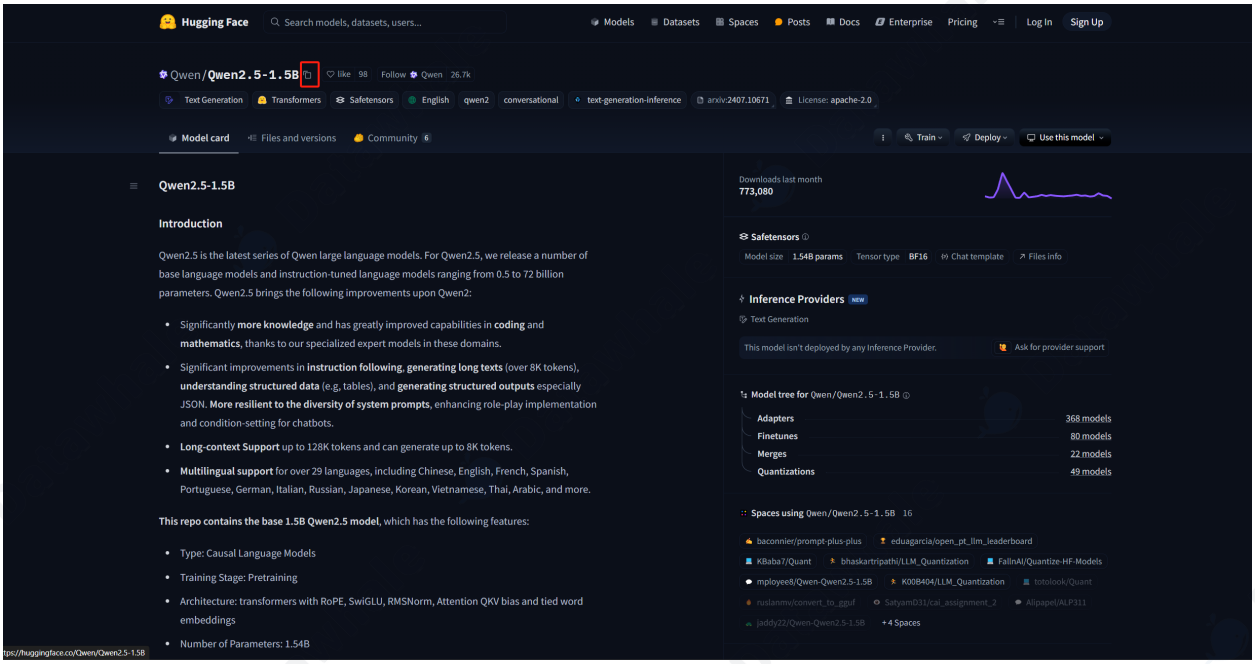


图6.5 模型下载标识

下载完成后，可以使用 AutoConfig 类直接加载下载好的配置文件：

```
# 加载定义好的模型参数-此处以 Qwen-2.5-1.5B 为例
# 使用 transforemrs 的 Config 类进行加载
from transformers import AutoConfig

# 下载参数的本地路径
model_path = "qwen-1.5b"
config = AutoConfig.from_pretrained(model_name_or_path)
```

也可以对配置文件进行自定义，然后以同样的方式加载即可。可以使用 AutoModel 类基于加载好的配置对象生成对应的模型：

```
# 使用该配置生成一个定义好的模型
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_config(config,trust_remote_code=True)
```

由于 LLM 一般都是 CausalLM 架构，此处使用了 AutoModelForCausalLM 类进行加载。如果是用于分类任务训练，可使用 AutoModelForSequenceClassification 类来加载。查看该 model，图6.6可以看到其架构和定义的配置文件相同：

```
Qwen2ForCausalLM(
  (model): Qwen2Model(
    (embed_tokens): Embedding(151936, 1536)
    (layers): ModuleList(
      (0-27): 28 x Qwen2DecoderLayer(
        (self_attn): Qwen2SdpaAttention(
          (q_proj): Linear(in_features=1536, out_features=1536, bias=True)
          (k_proj): Linear(in_features=1536, out_features=256, bias=True)
          (v_proj): Linear(in_features=1536, out_features=256, bias=True)
          (o_proj): Linear(in_features=1536, out_features=1536, bias=False)
          (rotary_emb): Qwen2RotaryEmbedding()
        )
        (mlp): Qwen2MLP(
          (gate_proj): Linear(in_features=1536, out_features=8960, bias=False)
          (up_proj): Linear(in_features=1536, out_features=8960, bias=False)
          (down_proj): Linear(in_features=8960, out_features=1536, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): Qwen2RMSNorm((1536,), eps=1e-06)
        (post_attention_layernorm): Qwen2RMSNorm((1536,), eps=1e-06)
      )
    )
    (norm): Qwen2RMSNorm((1536,), eps=1e-06)
  )
  (lm_head): Linear(in_features=1536, out_features=151936, bias=False)
)
```

图6.6 模型结构输出结果

该 model 就是一个从零初始化的 Qwen-2.5-1.5B 模型了。一般情况下，我们很少从零初始化 LLM 进行预训练，较多的做法是加载一个预训练好的 LLM 权重，在自己的语料上进行后训练。这里，我们也介绍如何从下载好的模型参数中初始化一个预训练好的模型。

```
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained(model_name_or_path,trust_remote_code=True)
```

类似的，直接使用 from_pretrained 方法加载即可，此处的 model_name_or_path 即为下载好的参数的本地路径。

我们还需要初始化一个 tokenizer。此处，我们直接使用 Qwen-2.5-1.5B 对应的 tokenzier 参数即可：

```
# 加载一个预训练好的 tokenizer
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(model_name_or_path)
```

加载好的 tokenizer 即可直接使用，对任意文本进行分词处理。

6.1.3 预训练数据处理

与第五章类似，我们使用出门问问序列猴子开源数据集作为预训练数据集，可以用与第五章一致的方式进行数据集的下载和解压。HuggingFace 的 datasets 库是和 transformers 框架配套的、用于数据下载和处理的第三方库。我们可以直接使用 datasets 的 load_dataset 函数来加载预训练数据：

```
# 加载预训练数据
from datasets import load_dataset

ds = load_dataset('json', data_files='/mobvoi_seq_monkey_general_open_corpus.jsonl')
```

注意，由于数据集较大，加载可能会出现时间较长或内存不够的情况，建议前期测试时将预训练数据集拆分一部分出来进行测试。加载出来的 ds 是一个 DatasetDict 对象，加载的数据会默认保存在 `train` 键对应的值中，可以通过以下代码查看：

```
ds["train"][0]
```

{'text': '在查处虚开增值税专用发票案件中，常常涉及进项留抵税额和税款损失的认定和处理。在计算税款损失时，要不要将进项留抵税额包括在内？\n对此，实务中存在意见分歧。\n有人主张归并，即计算税款损失时包括进项留抵税额；\n有人主张剥离，即计算税款损失时剔除进项留抵税额。分析这个问题，需要确定进项留抵税额与税款损失之间是什么关系。\n理清这二者之间的关系，首先需要了解增值税的概念和其抵扣机制。增值税是以商品（货物、服务等）在流转过程中产生的增值额作为计税依据而征收的一种流转税。为避免重复征税，在增值税中存在抵扣链条机制。\n一般而言，交易上游企业缴纳的税额，交易下游企业可以对相应的税额进行抵扣。\n对增值税一般纳税人来说，其购进货物、服务等取得增值税专用发票，发票上的税额是进项税额。\n其出售货物、服务等，向购买方开具增值税专用发票，发票的税额是销项税额。\n一般情况下，销项税额减去进项税额的金额是应纳税额，企业根据应纳税额按期申报纳税。\n其次需要了解进项留抵税额的概念及产生原因。\n在计算销项税额和进项税额的差额时，有时会出现负数，即当期进项税额大于当期销项税额。这个差额在当期未实现抵扣，为进项留抵税额，在以后纳税人有销项税额时再进行抵扣。\n企业产生进项留抵税额的主要原因是其进项税额和销项税额时间上的不一致。\n例如，企业前期集中采购货物和服务，投资大，销项税率低于进项税率等。\n从税款抵扣的角度看，进项留抵税额只是购进的这部分进项税额参与到增值税应纳税额的计算过程中，但是其对应的进项税额抵扣还未真正实现，一般要等到其未来有相应的销项税额时，才能真正实现进项税额抵扣。\n可见，进项留抵税额处于不确定状态，能否抵扣受到很多因素影响，例如企业经营中断，没有销项税额，这时进项留抵税额就无法实现抵扣。但如果企业按照税收政策规定申请进项留抵退税，进项税额抵扣就随之实现。\n最后需要了解税款损失的概念。\n税款损失，通常是指因虚开增值税专用发票，导致国家税款被骗或者流失的金额。关于税款损失，实务中有多种表述。\n例如，北京大学法学院教授陈兴良曾谈到虚开行为本身不会造成国家税款损失，只有利用发票抵扣时才会造成国家税款损失。刘兵等编著的《虚开增值税专用发票案例司法观点和案例解析》一书中提到：“给国家税款造成损失的数额，实际上就是被骗取的国家税款在侦查终结以前无法追回的部分。”\n赵清海与王家欣合著的《增值税专用发票虚开的判定与预防》一书中提到：“司法实践中，受票方用虚开的增值税专用发票予以抵扣的税款，从而导致受票方应纳税额的减少是法院所认定的国家税款流失的金额。”\n从这些表述可见，税款损失应该是实际造成的损失，不应包括不确定的部分—进项留抵税额，进项留抵税额与税款损失之间不能直接画等号。\n综上分析，进项留抵税额，只是使国家税款处于可能被抵扣的状态，还没有真正造成国家税款流失，一般情况下应将其从税款损失中剥离，特殊条件下将其归并入税款损失。\n例如，当纳税人造假按照税收政策规定申请进项留抵税额退税后，有关税款损失将会从危险状态转化成危害结果，这时候要将有关进项留抵税额并入税款损失。\n所以，在虚开增值税专用发票案件中，一般情况下，如果以纳税人的进项税额作为税款损失的计算基数，在对其进行行政处罚或刑事处罚时，应把进项留抵税额从税款损失中剔除，但纳税人申请进项留抵退税的除外。这样处理，把处罚与危害结果相对应，体现行政处罚法的过罚相当原则和刑法的罚当其罪原则。'}

图6.7 数据集展示

可以通过 feature 属性查看数据集的特征（也就是列），这里需要保存一下数据集的列名，因为后续数据处理时，再将文本 tokenize 之后，需要移除原先的文本：

```
# 查看特征
column_names = list(ds["train"].features)
# columnes_name:["text"]
```

接着使用加载好的 tokenizer 对数据集进行处理，此处使用 map 函数来进行批量处理：

```
# 对数据集进行 tokenize
def tokenize_function(examples):
    # 使用预先加载的 tokenizer 进行分词
    output = tokenizer([item for item in examples["text"]])
    return output

# 批量处理
tokenized_datasets = ds.map(
    tokenize_function,
    batched=True,
    num_proc=10,
    remove_columns=column_names,
    load_from_cache_file=True,
    desc="Running tokenizer on dataset",
)
```

处理完成后的数据集会包括'input_ids', 'attention_mask'两列，分别是文本 tokenize 之后的数值序列和注意力掩码（标识是否 padding）。map 方法会通过 remove_columns 参数将原先的'text'移除，训练中不再使用。

由于预训练一般为 CLM 任务，一次性学习多个样本的序列语义不影响模型性能，且训练数据量大、训练时间长，对训练效率要求比较高。在预训练过程中，一般会把多个文本段拼接在一起，处理成统一长度的文本块，再对每个文本块进行训练。在这里，我们实现一个拼接函数将文本块拼接到 2048个 token 长度，再通过 map 方法来进行批量处理：

```
# 预训练一般将文本拼接成固定长度的文本段
from itertools import chain

# 这里我们取块长为 2048
block_size = 2048

def group_texts(examples):
    # 将文本段拼接起来
    concatenated_examples = {k: list(chain(*examples[k])) for k in examples.keys()}
    # 计算拼起来的整体长度
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # 如果长度太长，进行分块
    if total_length >= block_size:
        total_length = (total_length // block_size) * block_size
    # 按 block_size 进行切分
    result = {
        k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    # CLM 任务，labels 和 input 是相同的
    result["labels"] = result["input_ids"].copy()
    return result

# 批量处理
lm_datasets = tokenized_datasets.map(
    group_texts,
    batched=True,
    num_proc=10,
    load_from_cache_file=True,
    desc=f"Grouping texts in chunks of {block_size}",
    batch_size = 40000,
)
train_dataset = lm_datasets["train"]
```

处理得到的 train_dataset 就是一个可直接用于 CLM Pretrain 的预训练数据集了，其每个样本长度为 2048个 token。

6.1.4 使用 Trainer 进行训练

接下来，我们使用 transformers 提供的 Trainer 类进行训练。Trainer 封装了模型的训练逻辑，且做了较好的效率优化、可视化等工作，可以高效、便捷地完成 LLM 的训练。

首先我们需要配置训练的超参数，使用 TrainingArguments 类来实例化一个参数对象：

```
from transformers import TrainingArguments
# 配置训练参数

training_args = TrainingArguments(
    output_dir="output",# 训练参数输出路径
    per_device_train_batch_size=4,# 训练的 batch_size
    gradient_accumulation_steps=4,# 梯度累计步数，实际 bs = 设置的 bs * 累计步数
    logging_steps=10,# 打印 loss 的步数间隔
    num_train_epochs=1,# 训练的 epoch 数
    save_steps=100, # 保存模型参数的步数间隔
    learning_rate=1e-4,# 学习率
    gradient_checkpointing=True# 开启梯度检查点
)
```

然后基于初始化的 model、tokenzier 和 training_args，并传入处理好的训练数据集，实例化一个 trainer 对象：

```
from transformers import Trainer, default_data_collator
from torchdata.datapipes.iter import IterableWrapper

# 训练器
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset= IterableWrapper(train_dataset),
    eval_dataset= None,
    tokenizer=tokenizer,
    # 默认为 MLM 的 collator，使用 CLM 的 collater
    data_collator=default_data_collator
)
```

再使用 train 方法，即会按照配置好的训练超参进行训练和保存：

```
trainer.train()
```

> 注：上述代码存放于 `./code/pretrian.ipynb` 文件中。

6.1.5 使用 DeepSpeed 实现分布式训练

由于预训练规模大、时间长，一般不推荐使用 Jupyter Notebook 来运行，容易发生中断。且由于预训练规模大，一般需要使用多卡进行分布式训练，否则训练时间太长。在这里，我们介绍如何基于上述代码，使用 DeepSpeed 框架实现分布式训练，从而完成业界可用的 LLM Pretrain。

长时间训练一般使用 bash 脚本设定超参，再启动写好的 python 脚本实现训练。我们使用一个 Python 脚本（`./code/pretrain.py`）来实现训练全流程。

先导入所需第三方库：

```
import logging
import math
import os
```

```
import sys
from dataclasses import dataclass, field
from torchdata.datapipes.iter import IterableWrapper
from itertools import chain
import deepspeed
from typing import Optional,List

import datasets
import pandas as pd
import torch
from datasets import load_dataset
import transformers
from transformers import (
    AutoConfig,
    AutoModelForCausalLM,
    AutoTokenizer,
    HfArgumentParser,
    Trainer,
    TrainingArguments,
    default_data_collator,
    set_seed,
)
import datetime
from transformers.testing_utils import CaptureLogger
from transformers.trainer_utils import get_last_checkpoint
import wandb
```

首先需要定义几个超参的类型，用于处理 sh 脚本中设定的超参值。由于 transformers 本身有 TraingingArguments 类，其中包括了训练的一些必备超参数。我们这里只需定义 TrainingArguments 中未包含的超参即可，主要包括模型相关的超参（定义在 ModelArguments）和数据相关的超参（定义在 DataTrainingArguments）：

```
# 超参类
@dataclass
class ModelArguments:
    """
    关于模型的参数
    """

    model_name_or_path: Optional[str] = field(
        default=None,
        metadata={
            "help": (
                "后训练使用，为预训练模型参数地址"
            )
        },
    )
    config_name: Optional[str] = field(
        default=None, metadata={"help": "预训练使用，Config 文件地址"}
    )
    tokenizer_name: Optional[str] = field(
```

```
        default=None, metadata={"help": "预训练 Tokenizer 地址"}
    )
    torch_dtype: Optional[str] = field(
        default=None,
        metadata={
            "help": (
                "模型训练使用的数据类型，推荐 bfloat16"
            ),
            "choices": ["auto", "bfloat16", "float16", "float32"],
        },
    )


@dataclass
class DataTrainingArguments:
    """
    关于训练的参数
    """

    train_files: Optional[List[str]]  = field(default=None, metadata={"help": "训练数据路
径"})
    block_size: Optional[int] = field(
        default=None,
        metadata={
            "help": (
                "设置的文本块长度"
            )
        },
    )
    preprocessing_num_workers: Optional[int] = field(
        default=None,
        metadata={"help": "预处理使用线程数."},
    )
```

然后即可定义一个主函数实现上述训练过程的封装。首先通过 transformers 提供的 HfArgumentParser 工具来加载 sh 脚本中设定的超参：

```
# 加载脚本参数
parser = HfArgumentParser((ModelArguments, DataTrainingArguments, TrainingArguments))
model_args, data_args, training_args = parser.parse_args_into_dataclasses()
```

在大规模的训练中，一般使用 log 来保存训练过程的信息，一般不推荐使用 print 直接打印，容易发生关键训练信息的丢失。这里，我们直接使用 python 自带的 logging 库来实现日志记录。首先需要进行 log 的设置：

```
# 设置日志
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    handlers=[logging.StreamHandler(sys.stdout)],
)
```

```
# 将日志级别设置为 INFO
transformers.utils.logging.set_verbosity_info()
log_level = training_args.get_process_log_level()
logger.setLevel(log_level)
datasets.utils.logging.set_verbosity(log_level)
transformers.utils.logging.set_verbosity(log_level)
transformers.utils.logging.enable_default_handler()
transformers.utils.logging.enable_explicit_format()
```

这里将日志的级别设置为 INFO。logging 的日志共有 DEBUG、INFO、WARNING、ERROR 以及 CRITICAL 五个级别，将日志设置为哪个级别，就会只输出该级别及该级别之上的信息。设置完成后，在需要记录日志的地方，直接使用 logger 即可，记录时会指定记录日志的级别，例如：

```
# 训练整体情况记录
logger.warning(
    f"Process rank: {training_args.local_rank}, device: {training_args.device}, n_gpu:
{training_args.n_gpu}"
    + f"distributed training: {bool(training_args.local_rank != -1)}, 16-bits training:
{training_args.fp16}"
)
logger.info(f"Training/evaluation parameters {training_args}")
```

后续就不再赘述脚本中的日志记录。

在大规模训练中，发生中断是往往难以避免的，训练一般会固定间隔保存 checkpoint，中断之后基于最近的 checkpoint 恢复训练即可。因此，我们需要首先检测是否存在旧的 checkpoint 并从 checkpoint 恢复训练：

```
# 检查 checkpoint
last_checkpoint = None
if os.path.isdir(training_args.output_dir):
    # 使用 transformers 自带的 get_last_checkpoint 自动检测
    last_checkpoint = get_last_checkpoint(training_args.output_dir)
    if last_checkpoint is None and len(os.listdir(training_args.output_dir)) > 0:
        raise ValueError(
            f"输出路径 ({training_args.output_dir}) 非空 "
        )
    elif last_checkpoint is not None and training_args.resume_from_checkpoint is None:
        logger.info(
            f"从 {last_checkpoint}恢复训练"
        )
```

接着以上文介绍过的方式初始化模型，此处将从零初始化和基于已有预训练模型初始化包装在一起：

```
# 初始化模型
if model_args.config_name is not None:
    # from scrach
    config = AutoConfig.from_pretrained(model_args.config_name)
    logger.warning("你正在从零初始化一个模型")
    logger.info(f"模型参数配置地址：{model_args.config_name}")
    logger.info(f"模型参数：{config}")
```

```
    model = AutoModelForCausalLM.from_config(config,trust_remote_code=True)
    n_params = sum({p.data_ptr(): p.numel() for p in model.parameters()}.values())
    logger.info(f"预训练一个新模型 - Total size={n_params/2**20:.2f}M params")
elif model_args.model_name_or_path is not None:
    logger.warning("你正在初始化一个预训练模型")
    logger.info(f"模型参数地址：{model_args.model_name_or_path}")
    model =
AutoModelForCausalLM.from_pretrained(model_args.model_name_or_path,trust_remote_code=True)
    n_params = sum({p.data_ptr(): p.numel() for p in model.parameters()}.values())
    logger.info(f"继承一个预训练模型 - Total size={n_params/2**20:.2f}M params")
else:
    logger.error("config_name 和 model_name_or_path 不能均为空")
    raise ValueError("config_name 和 model_name_or_path 不能均为空")
```

再类似的进行 tokenizer 的加载和预训练数据的处理。该部分和上文完全一致，此处不再赘述，读者可以在代码中详细查看细节。类似的，使用 Trainer 进行训练：

```
logger.info("初始化 Trainer")
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset= IterableWrapper(train_dataset),
    tokenizer=tokenizer,
    data_collator=default_data_collator
)

# 从 checkpoint 加载
checkpoint = None
if training_args.resume_from_checkpoint is not None:
    checkpoint = training_args.resume_from_checkpoint
elif last_checkpoint is not None:
        checkpoint = last_checkpoint

logger.info("开始训练")
train_result = trainer.train(resume_from_checkpoint=checkpoint)
trainer.save_model()
```

注意，由于上文检测了是否存在 checkpoint，此处使用 resume_from_checkpoint 来实现从 checkpoint 恢复训练的功能。

由于在大规模训练中监测训练进度、loss 下降趋势尤为重要，在脚本中，我们使用了 wandb 作为训练检测的工具。在脚本开始进行了 wandb 的初始化：

```
# 初始化 WandB
wandb.init(project="pretrain", name="from_scrach")
```

在启动训练后，终端会输出 wandb 监测的 url，点击即可观察训练进度。此处不再赘述 wandb 的使用细节，欢迎读者查阅相关的资料说明。

完成上述代码后，我们使用一个 sh 脚本（`./code/pretrain.sh`）定义超参数的值，并通过 Deepspeed 启动训练，从而实现高效的多卡分布式训练：

```
# 设置可见显卡
CUDA_VISIBLE_DEVICES=0,1

deepspeed pretrain.py \
    --config_name autodl-tmp/qwen-1.5b \
    --tokenizer_name autodl-tmp/qwen-1.5b \
    --train_files autodl-
tmp/dataset/pretrain_data/mobvoi_seq_monkey_general_open_corpus_small.jsonl \
    --per_device_train_batch_size 16 \
    --gradient_accumulation_steps 4 \
    --do_train \
    --output_dir autodl-tmp/output/pretrain \
    --evaluation_strategy  no \
    --learning_rate 1e-4 \
    --num_train_epochs 1 \
    --warmup_steps 200 \
    --logging_dir autodl-tmp/output/pretrain/logs \
    --logging_strategy steps \
    --logging_steps 5 \
    --save_strategy steps \
    --save_steps 100 \
    --preprocessing_num_workers 10 \
    --save_total_limit 1 \
    --seed 12 \
    --block_size 2048 \
    --bf16 \
    --gradient_checkpointing \
    --deepspeed ./ds_config_zero2.json \
    --report_to wandb
    # --resume_from_checkpoint ${output_model}/checkpoint-20400 \
```

在安装了 Deepspeed 第三方库后，可以直接通过 Deepspeed 命令来启动多卡训练。上述脚本命令主要是定义了各种超参数的值，可参考使用。在第四章中，我们介绍了 DeepSpeed 分布式训练的原理和 ZeRO 阶段设置，在这里，我们使用 ZeRO-2 进行训练。此处加载了 `ds_config_zero.json` 作为 DeepSpeed 的配置参数：

```
{
    "fp16": {
        "enabled": "auto",
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },
    "bf16": {
        "enabled": "auto"
    },
    "optimizer": {
```

```
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "betas": "auto",
            "eps": "auto",
            "weight_decay": "auto"
        }
    },

    "scheduler": {
        "type": "WarmupLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto"
        }
    },

    "zero_optimization": {
        "stage": 2,
        "offload_optimizer": {
            "device": "none",
            "pin_memory": true
        },
        "allgather_partitions": true,
        "allgather_bucket_size": 2e8,
        "overlap_comm": true,
        "reduce_scatter": true,
        "reduce_bucket_size": 2e8,
        "contiguous_gradients": true
    },

    "gradient_accumulation_steps": "auto",
    "gradient_clipping": "auto",
    "steps_per_print": 100,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

最后，在终端 bash 运行该 `pretrain.sh` 脚本即可开始训练。

6.2 模型有监督微调

在上一节，我们介绍了如何使用 Transformers 框架快速、高效地进行模型预训练。在本部分，我们将基于上部分内容，介绍如何使用 Transformers 框架对预训练好的模型进行有监督微调。

6.2.1 Pretrain VS SFT

首先需要回顾一下，对 LLM 进行预训练和进行有监督微调的核心差异在于什么。在第四章中提到过，目前成型的 LLM 一般通过 Pretrain-SFT-RLHF 三个阶段来训练，在 Pretrain 阶段，会对海量无监督文本进行自监督建模，来学习文本语义规则和文本中的世界知识；在 SFT 阶段，一般通过对 Pretrain 好的模型进行指令微调，即训练模型根据用户指令完成对应任务，从而使模型能够遵循用户指令，根据用户指令进行规划、行动和输出。因此，Pretrain 和 SFT 均使用 CLM 建模，其核心差异在于，Pretrain 使用海量无监督文本进行训练，模型直接对文本执行“预测下一个 token”的任务；而 SFT 使用构建成对的指令对数据，模型根据输入的指令，建模后续的输出。反映到具体的训练实现上，Pretrain 会对全部 text 进行 loss 计算，要求模型对整个文本实现建模预测；而 SFT 仅对输出进行 loss 计算，不计算指令部分的 loss。

因此，相较于上一节完成的 Pretrain 代码，SFT 部分仅需要修改数据处理环节，实现对指令对数据转化为训练样本的构建，其余部分和 Pretrain 是完全一致的实现逻辑。本部分代码脚本为 `./code/finetune.py`。

6.2.2 微调数据处理

同样与第五章类似，我们此处使用贝壳开源的 BelleGroup 数据集进行 SFT。

在 SFT 过程中，我们会定义一个 Chat Template，这个 Template 即表示了如何将对话数据转化为一个模型可以建模拟合的文本序列。当我们使用做过 SFT 的模型进行下游任务微调时，一般需要查看该模型的 Chat Template 并进行适配，即是为了不损伤其在 SFT 中学到的指令遵循能力。由于我们此处使用 Pretrain 模型进行 SFT，可以自定义一个 Chat Template。由于我们使用了 Qwen-2.5-1.5B 模型结构进行 Pretrain，此处我们沿承使用 Qwen-2.5 的 Chat Template。如果读者没有足够的资源进行上一部分模型的 Pretrain 的话，此处也可以使用官方的 Qwen-2.5-1.5B 模型作为 SFT 的基座模型。

我们首先定义几个特殊 token，特殊 token 在模型进行拟合中有特殊的作用，包括文本序列开始（BOS）、文本序列结束（EOS）、换行符等。定义特殊 token，有助于避免模型在拟合过程中的语义混淆：

```
# 不同的 tokenizer 需要特别定义
# BOS
im_start = tokenizer("<|im_start|>").input_ids
# EOS
im_end = tokenizer("<|im_end|>").input_ids
# PAD
IGNORE_TOKEN_ID = tokenizer.pad_token_id
# 换行符
nl_tokens = tokenizer('\n').input_ids
# 角色标识符
_system = tokenizer('system').input_ids + nl_tokens
_user = tokenizer('human').input_ids + nl_tokens
_assistant = tokenizer('assistant').input_ids + nl_tokens
```

Qwen 系列的 Chat Template 一般有三个对话角色：System、User 和 Assistant。System 是系统提示词，负责激活模型的能力，默认为“You are a helpful assistant.”，一般不会在 SFT 过程中更改使用。User 即为用户给出的提示词，此处由于数据集中的对话角色为 “human”，我们将 “user” 修改为了“human”。Assistant 即为 LLM 给出的回复，也就是模型在 SFT 过程中需要拟合的文本。

接着，由于该数据集是一个多轮对话数据集，我们需要对多轮对话进行拼接处理，将多轮对话拼接到一个文本序列中：

```
# 拼接多轮对话
```

```
input_ids, targets = [], []
# 多个样本
for i in tqdm(range(len(sources))):
    # source 为一个多轮对话样本
    source = sources[i]
    # 从 user 开始
    if source[0]["from"] != "human":
        source = source[1:]
    # 分别是输入和输出
    input_id, target = [], []
    # system: 【BOS】system\nYou are a helpful assistant.【EOS】\n
    system = im_start + _system + tokenizer(system_message).input_ids + im_end + nl_tokens
    input_id += system
    # system 不需要拟合
    target += im_start + [IGNORE_TOKEN_ID] * (len(system)-3) + im_end + nl_tokens
    assert len(input_id) == len(target)
    # 依次拼接
    for j, sentence in enumerate(source):
        # sentence 为一轮对话
        role = roles[sentence["from"]]
        # user: <|im_start|>human\ninstruction【EOS】\n
        # assistant: <|im_start|>assistant\nresponse【EOS】\n
        _input_id = tokenizer(role).input_ids + nl_tokens + \
            tokenizer(sentence["value"]).input_ids + im_end + nl_tokens
        input_id += _input_id
        if role == '<|im_start|>human':
            # user 不需要拟合
            _target = im_start + [IGNORE_TOKEN_ID] * (len(_input_id)-3) + im_end +
nl_tokens
        elif role == '<|im_start|>assistant':
            # assistant 需要拟合
            _target = im_start + [IGNORE_TOKEN_ID] * len(tokenizer(role).input_ids) + \
                _input_id[len(tokenizer(role).input_ids)+1:-2] + im_end + nl_tokens
        else:
            print(role)
            raise NotImplementedError
        target += _target
    assert len(input_id) == len(target)
    # 最后进行 PAD
    input_id += [tokenizer.pad_token_id] * (max_len - len(input_id))
    target += [IGNORE_TOKEN_ID] * (max_len - len(target))
    input_ids.append(input_id[:max_len])
    targets.append(target[:max_len])
```

上述代码沿承了 Qwen 的 Chat Template 逻辑，读者也可以根据自己的偏好进行修改，其核心点在于 User 的文本不需要拟合，因此 targets 中 User 对应的文本内容是使用的 IGNORE_TOKEN_ID 进行遮蔽，而 Assistant 对应的文本内容则是文本原文，是需要计算 loss 的。目前主流 LLM IGNORE_TOKEN_ID 一般设置为 -100。

完成拼接后，将 tokenize 后的数值序列转化为 `Torch.tensor`，再拼接成 Dataset 所需的字典返回即可：

```
input_ids = torch.tensor(input_ids)
targets = torch.tensor(targets)

return dict(
    input_ids=input_ids,
    labels=targets,
    attention_mask=input_ids.ne(tokenizer.pad_token_id),
)
```

完成上述处理逻辑后，需要自定义一个 Dataset 类，在该类中调用该逻辑进行数据的处理：

```
class SupervisedDataset(Dataset):

    def __init__(self, raw_data, tokenizer, max_len: int):
        super(SupervisedDataset, self).__init__()
        # 加载并预处理数据
        sources = [example["conversations"] for example in raw_data]
        # preprocess 即上文定义的数据预处理逻辑
        data_dict = preprocess(sources, tokenizer, max_len)

        self.input_ids = data_dict["input_ids"]
        self.labels = data_dict["labels"]
        self.attention_mask = data_dict["attention_mask"]

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, i) -> Dict[str, torch.Tensor]:
        return dict(
            input_ids=self.input_ids[i],
            labels=self.labels[i],
            attention_mask=self.attention_mask[i],
        )
```

该类继承自 Torch 的 Dataset 类，可以直接在 Trainer 中使用。完成数据处理后，基于上一节脚本，修改数据处理逻辑即可，后续模型训练等几乎完全一致，此处附上主函数逻辑：

```
# 加载脚本参数
parser = HfArgumentParser((ModelArguments, DataTrainingArguments, TrainingArguments))
model_args, data_args, training_args = parser.parse_args_into_dataclasses()

# 初始化 WandB
wandb.init(project="sft", name="qwen-1.5b")

# 设置日志
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    handlers=[logging.StreamHandler(sys.stdout)],
)
```

```
# 将日志级别设置为 INFO
transformers.utils.logging.set_verbosity_info()
log_level = training_args.get_process_log_level()
logger.setLevel(log_level)
datasets.utils.logging.set_verbosity(log_level)
transformers.utils.logging.set_verbosity(log_level)
transformers.utils.logging.enable_default_handler()
transformers.utils.logging.enable_explicit_format()

# 训练整体情况记录
logger.warning(
    f"Process rank: {training_args.local_rank}, device: {training_args.device}, n_gpu:
{training_args.n_gpu}"
    + f"distributed training: {bool(training_args.local_rank != -1)}, 16-bits training:
{training_args.fp16}"
)
logger.info(f"Training/evaluation parameters {training_args}")

# 检查 checkpoint
last_checkpoint = None
if os.path.isdir(training_args.output_dir):
    last_checkpoint = get_last_checkpoint(training_args.output_dir)
    if last_checkpoint is None and len(os.listdir(training_args.output_dir)) > 0:
        raise ValueError(
            f"输出路径 ({training_args.output_dir}) 非空 "
        )
    elif last_checkpoint is not None and training_args.resume_from_checkpoint is None:
        logger.info(
            f"从 {last_checkpoint}恢复训练"
        )

# 设置随机数种子.
set_seed(training_args.seed)

# 初始化模型
logger.warning("加载预训练模型")
logger.info(f"模型参数地址：{model_args.model_name_or_path}")
model =
AutoModelForCausalLM.from_pretrained(model_args.model_name_or_path,trust_remote_code=True)
n_params = sum({p.data_ptr(): p.numel() for p in model.parameters()}.values())
logger.info(f"继承一个预训练模型 - Total size={n_params/2**20:.2f}M params")

# 初始化 Tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_args.model_name_or_path)
logger.info("完成 tokenzier 加载")

# 加载微调数据
with open(data_args.train_files) as f:
    lst = [json.loads(line) for line in f.readlines()[:10000]]
logger.info("完成训练集加载")
logger.info(f"训练集地址：{data_args.train_files}")
```

```
logger.info(f'训练样本总数:{len(lst)}')
# logger.info(f"训练集采样: {ds["train"][0]}")

train_dataset = SupervisedDataset(lst, tokenizer=tokenizer, max_len=2048)

logger.info("初始化 Trainer")
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset= IterableWrapper(train_dataset),
    tokenizer=tokenizer
)

# 从 checkpoint 加载
checkpoint = None
if training_args.resume_from_checkpoint is not None:
    checkpoint = training_args.resume_from_checkpoint
elif last_checkpoint is not None:
        checkpoint = last_checkpoint

logger.info("开始训练")
train_result = trainer.train(resume_from_checkpoint=checkpoint)
trainer.save_model()
```

启动方式也同样在 sh 脚本中使用 deepspeed 启动即可，此处不再赘述，源码见 ./code/finetune.sh。

6.3 高效微调

在前面几节，我们详细介绍了基于 Transformers 框架对模型进行 Pretrain、SFT 以及 RLHF 的原理和实践细节。但是，由于 LLM 参数量大，训练数据多，通过上述方式对模型进行训练（主要指 SFT 及 RLHF）需要调整模型全部参数，资源压力非常大。对资源有限的企业或课题组来说，如何高效、快速对模型进行领域或任务的微调，以低成本地使用 LLM 完成目标任务，是非常重要的。

6.3.1 高效微调方案

针对全量微调的昂贵问题，目前主要有两种解决方案：

Adapt Tuning。即在模型中添加 Adapter 层，在微调时冻结原参数，仅更新 Adapter 层。

具体而言，其在预训练模型每层中插入用于下游任务的参数，即 Adapter 模块，在微调时冻结模型主体，仅训练特定于任务的参数，如图6.8所示。

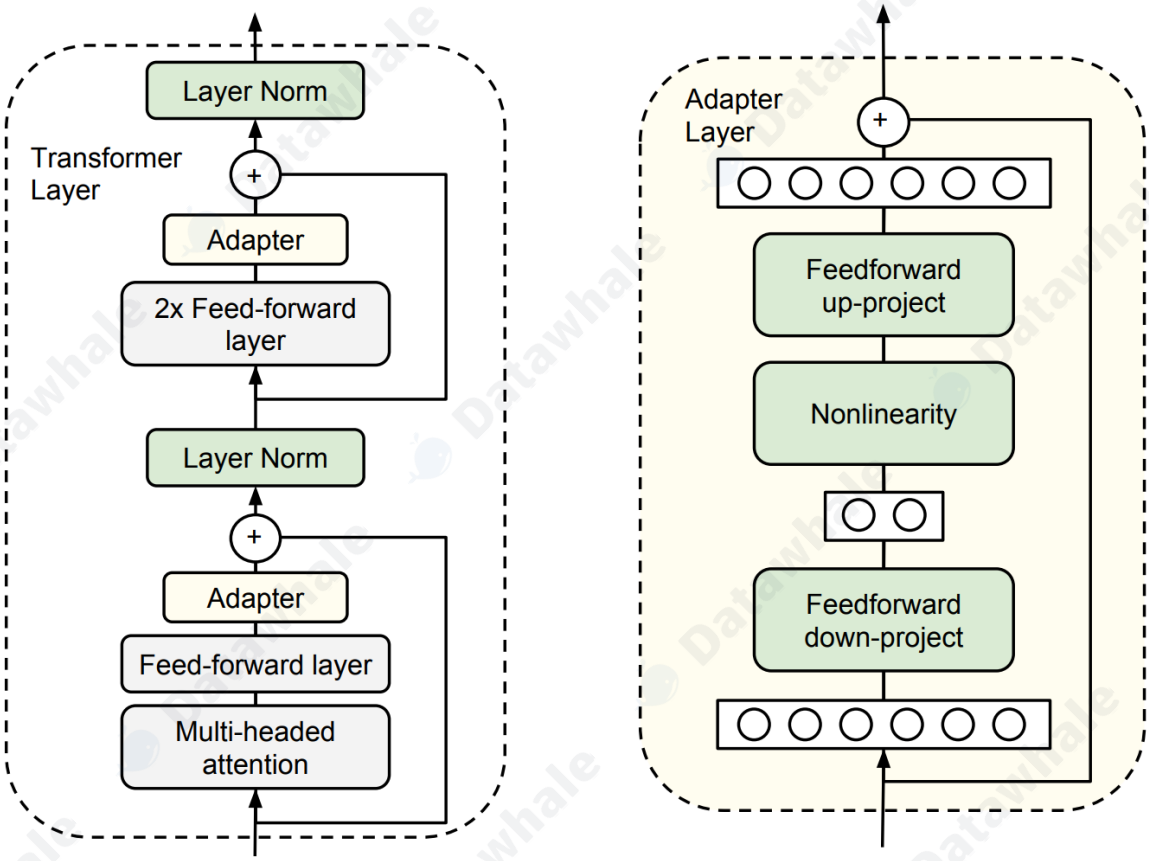


图6.8 Adapt Tuning

每个 Adapter 模块由两个前馈子层组成，第一个前馈子层将 Transformer 块的输出作为输入，将原始输入维度 d 投影到 m，通过控制 m 的大小来限制 Adapter 模块的参数量，通常情况下 $m << d$。在输出阶段，通过第二个前馈子层还原输入维度，将 m 重新投影到 d，作为 Adapter 模块的输出(如上图右侧结构)。

LoRA 事实上就是一种改进的 Adapt Tuning 方法。但 Adapt Tuning 方法存在推理延迟问题，由于增加了额外参数和额外计算量，导致微调之后的模型计算速度相较原预训练模型更慢。

Prefix Tuning。该种方法固定预训练 LM，为 LM 添加可训练，任务特定的前缀，这样就可以为不同任务保存不同的前缀，微调成本也小。具体而言，在每一个输入 token 前构造一段与下游任务相关的 virtual tokens 作为 prefix，在微调时只更新 prefix 部分的参数，而其他参数冻结不变。

也是目前常用的微量微调方法的 Ptuning，其实就是 Prefix Tuning 的一种改进。但 Prefix Tuning 也存在固定的缺陷：模型可用序列长度减少。由于加入了 virtual tokens，占用了可用序列长度，因此越高的微调质量，模型可用序列长度就越低。

6.3.2 LoRA 微调

如果一个大模型是将数据映射到高维空间进行处理，这里假定在处理一个细分的小任务时，是不需要那么复杂的大模型的，可能只需要在某个子空间范围内就可以解决，那么也就不需要对全量参数进行优化了，我们可以定义当对某个子空间参数进行优化时，能够达到全量参数优化的性能的一定水平（如90%精度）时，那么这个子空间参数矩阵的秩就可以称为对应当前待解决问题的本征秩（intrinsic rank）。

预训练模型本身就隐式地降低了本征秩，当针对特定任务进行微调后，模型中权重矩阵其实具有更低的本征秩（intrinsic rank）。同时，越简单的下游任务，对应的本征秩越低。（Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning）因此，权重更新的那部分参数矩阵尽管随机投影到较小的子空间，仍然可以有效的学习，可以理解为针对特定的下游任务这些权重矩阵就不要求满秩。我们可以通过优化密集层

在适应过程中变化的秩分解矩阵来间接训练神经网络中的一些密集层，从而实现仅优化密集层的秩分解矩阵来达到微调效果。

例如，假设预训练参数为 θ_0^D，在特定下游任务上密集层权重参数矩阵对应的本征秩为 θ^d，对应特定下游任务微调参数为 θ^D，那么有：

$$\theta^D = \theta_0^D + \theta^d M$$

这个 M 即为 LoRA 优化的秩分解矩阵。

想对于其他高效微调方法，LoRA 存在以下优势：

1. 可以针对不同的下游任务构建小型 LoRA 模块，从而在共享预训练模型参数基础上有效地切换下游任务。
2. LoRA 使用自适应优化器（Adaptive Optimizer），不需要计算梯度或维护大多数参数的优化器状态，训练更有效、硬件门槛更低。
3. LoRA 使用简单的线性设计，在部署时将可训练矩阵与冻结权重合并，不存在推理延迟。
4. LoRA 与其他方法正交，可以组合。

因此，LoRA 成为目前高效微调 LLM 的主流方法，尤其是对于资源受限、有监督训练数据受限的情况下，LoRA 微调往往会成为 LLM 微调的首选方法。

6.3.3 LoRA 微调的原理

（1）低秩参数化更新矩阵

LoRA 假设权重更新的过程中也有一个较低的本征秩，对于预训练的权重参数矩阵 $W0 \in R^{d\times k}$ (d 为上一层输出维度，k 为下一层输入维度)，使用低秩分解来表示其更新：

$$W_0 + \Delta W = W_0 + BA \ \ where\ B \in R^{d\times r}, A \in R^{r\times k}$$

在训练过程中，W_0 冻结不更新，A、B 包含可训练参数。

因此，LoRA 的前向传递函数为：

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

在开始训练时，对 A 使用随机高斯初始化，对 B 使用零初始化，然后使用 Adam 进行优化。

训练思路如图6.9所示：

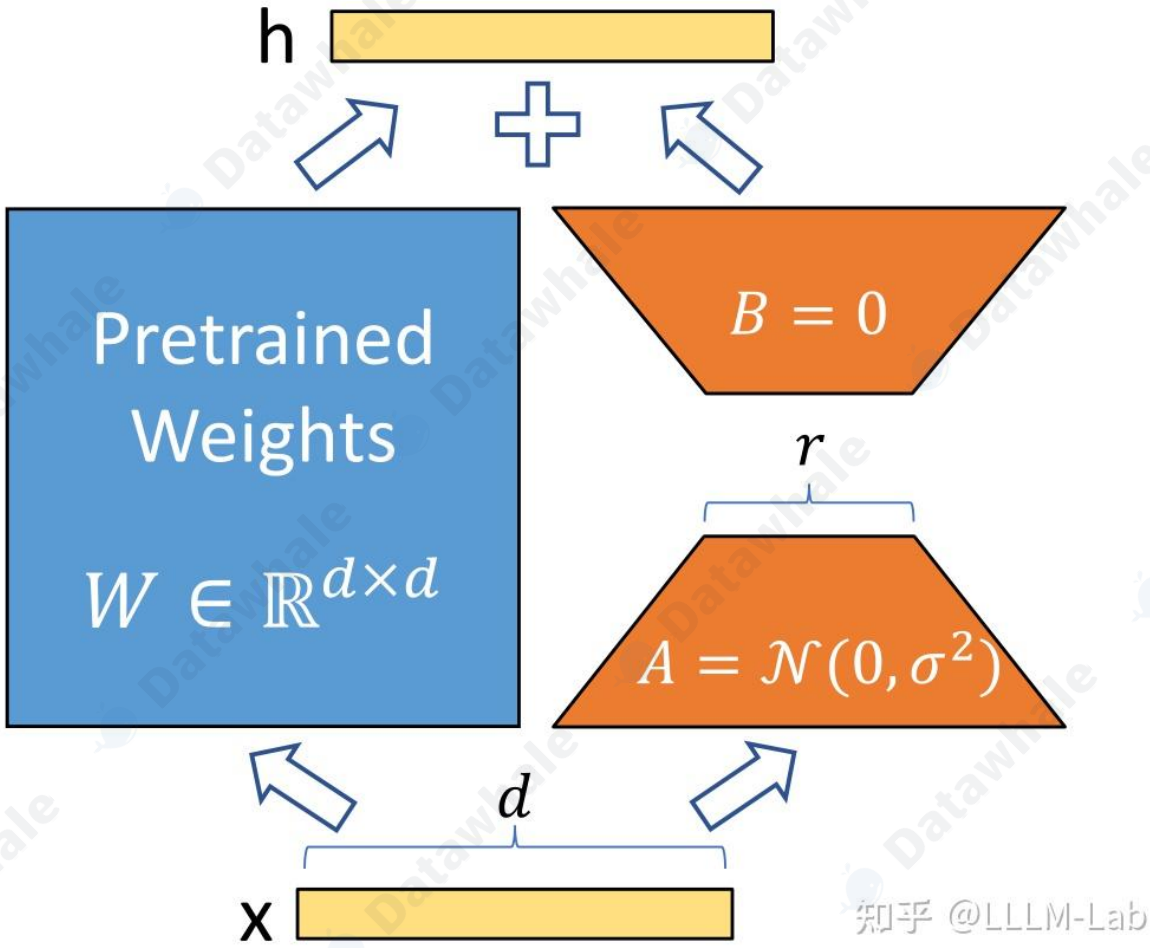


图6.9 LoRA

（2）应用于 Transformer

在 Transformer 结构中，LoRA 技术主要应用在注意力模块的四个权重矩阵：W_q、W_k、W_v、W_0，而冻结 MLP 的权重矩阵。

通过消融实验发现同时调整 W_q 和 W_v 会产生最佳结果。

在上述条件下，可训练参数个数为：

$$\Theta = 2 \times L_{LoRA} \times d_{model} \times r$$

其中，L_{LoRA} 为应用 LoRA 的权重矩阵的个数，d_{model} 为 Transformer 的输入输出维度，r 为设定的 LoRA 秩。

一般情况下，r 取到 4、8、16。

6.3.4 LoRA 的代码实现

目前一般通过 peft 库来实现模型的 LoRA 微调。peft 库是 huggingface 开发的第三方库，其中封装了包括 LoRA、Adapt Tuning、P-tuning 等多种高效微调方法，可以基于此便捷地实现模型的 LoRA 微调。

本文简单解析 peft 库中的 LoRA 微调代码，简单分析 LoRA 微调的代码实现。

（1）实现流程

LoRA 微调的内部实现流程主要包括以下几个步骤：

1. 确定要使用 LoRA 的层。peft 库目前支持调用 LoRA 的层包括：nn.Linear、nn.Embedding、nn.Conv2d 三种。
2. 对每一个要使用 LoRA 的层，替换为 LoRA 层。所谓 LoRA 层，实则是在该层原结果基础上增加了一个旁路，通过低秩分解（即矩阵 A 和矩阵 B）来模拟参数更新。
3. 冻结原参数，进行微调，更新 LoRA 层参数。

（2）确定 LoRA 层

在进行 LoRA 微调时，首先需要确定 LoRA 微调参数，其中一个重要参数即是 target_modules。target_modules 一般是一个字符串列表，每一个字符串是需要进行 LoRA 的层名称，例如：

```
target_modules = ["q_proj","v_proj"]
```

这里的 q_proj 即为注意力机制中的 W_q， v_proj 即为注意力机制中的 W_v。我们可以根据模型架构和任务要求自定义需要进行 LoRA 操作的层。

在创建 LoRA 模型时，会获取该参数，然后在原模型中找到对应的层，该操作主要通过使用 re 对层名进行正则匹配实现：

```
# 找到模型的各个组件中，名字里带"q_proj"，"v_proj"的
target_module_found = re.fullmatch(self.peft_config.target_modules, key)
# 这里的 key，是模型的组件名
```

（3）替换 LoRA 层

对于找到的每一个目标层，会创建一个新的 LoRA 层进行替换。

LoRA 层在具体实现上，是定义了一个基于 Lora 基类的 Linear 类，该类同时继承了 nn.Linear 和 LoraLayer。LoraLayer 即是 Lora 基类，其主要构造了 LoRA 的各种超参：

```
class LoraLayer:
    def __init__(
        self,
        r: int, # LoRA 的秩
        lora_alpha: int, # 归一化参数
        lora_dropout: float, # LoRA 层的 dropout 比例
        merge_weights: bool, # eval 模式中，是否将 LoRA 矩阵的值加到原权重矩阵上
    ):
        self.r = r
        self.lora_alpha = lora_alpha
        # Optional dropout
        if lora_dropout > 0.0:
            self.lora_dropout = nn.Dropout(p=lora_dropout)
        else:
            self.lora_dropout = lambda x: x
        # Mark the weight as unmerged
        self.merged = False
        self.merge_weights = merge_weights
        self.disable_adapters = False
```

nn.Linear 就是 Pytorch 的线性层实现。Linear 类就是具体的 LoRA 层，其主要实现如下：

```
class Linear(nn.Linear, LoraLayer):
    # LoRA 层
    def __init__(
        self,
        in_features: int,
        out_features: int,
        r: int = 0,
        lora_alpha: int = 1,
        lora_dropout: float = 0.0,
        fan_in_fan_out: bool = False,
        merge_weights: bool = True,
        **kwargs,
    ):
        # 继承两个基类的构造函数
        nn.Linear.__init__(self, in_features, out_features, **kwargs)
        LoraLayer.__init__(self, r=r, lora_alpha=lora_alpha, lora_dropout=lora_dropout,
merge_weights=merge_weights)

        self.fan_in_fan_out = fan_in_fan_out
        # Actual trainable parameters
        if r > 0:
            # 参数矩阵 A
            self.lora_A = nn.Linear(in_features, r, bias=False)
            # 参数矩阵 B
            self.lora_B = nn.Linear(r, out_features, bias=False)
            # 归一化系数
            self.scaling = self.lora_alpha / self.r
            # 冻结原参数，仅更新 A 和 B
            self.weight.requires_grad = False
        # 初始化 A 和 B
        self.reset_parameters()
        if fan_in_fan_out:
            self.weight.data = self.weight.data.T
```

替换时，直接将原层的 weight 和 bias 复制给新的 LoRA 层，再将新的 LoRA 层分配到指定设备即可。

（4）训练

实现了 LoRA 层的替换后，进行微调训练即可。由于在 LoRA 层中已冻结原参数，在训练中只有 A 和 B 的参数会被更新，从而实现了高效微调。训练的整体过程与原 Fine-tune 类似，此处不再赘述。由于采用了 LoRA 方式，forward 函数也会对应调整：

```
    def forward(self, x: torch.Tensor):
        if self.disable_adapters:
            if self.r > 0 and self.merged:
                self.weight.data -= (
```

```
                    transpose(self.lora_B.weight @ self.lora_A.weight,
self.fan_in_fan_out) * self.scaling
                )
                self.merged = False

            return F.linear(x, transpose(self.weight, self.fan_in_fan_out),
bias=self.bias)
        '''主要分支'''
        elif self.r > 0 and not self.merged:
            result = F.linear(x, transpose(self.weight, self.fan_in_fan_out),
bias=self.bias)
            if self.r > 0:
                result += self.lora_B(self.lora_A(self.lora_dropout(x))) * self.scaling
            return result
        else:
            return F.linear(x, transpose(self.weight, self.fan_in_fan_out),
bias=self.bias)
```

上述代码由于考虑到参数合并问题，有几个分支，此处我们仅阅读第二个分支即 elif 分支即可。基于 LoRA 的前向计算过程如前文公式所示，首先计算原参数与输入的乘积，再加上 A、B 分别与输入的乘积即可。

6.3.5 使用 peft 实现 LoRA 微调

peft 进行了很好的封装，支持我们便捷、高效地对大模型进行微调。此处以第二节的 LLM SFT 为例，简要介绍如何使用 peft 对大模型进行微调。如果是应用在 RLHF 上，整体思路是一致的。

首先加载所需使用库：

```
import torch.nn as nn
from transformers import AutoTokenizer, AutoModel
from peft import get_peft_model, LoraConfig, TaskType, PeftModel
from transformers import Trainer
```

其次加载原模型与原 tokenizer，此处和第二节一致：

```
# 加载基座模型
tokenizer = AutoTokenizer.from_pretrained(MODEL_PATH, trust_remote_code=True)
model = AutoModel.from_pretrained(
    MODEL_PATH, trust_remote_code=True
)
```

接着，设定 peft 参数：

```
peft_config = LoraConfig(
            task_type=TaskType.CAUSAL_LM,
            inference_mode=False,
            r=8,
            lora_alpha=32,
            lora_dropout=0.1,
        )
```

注意，对不同的模型，LoRA 参数可能有所区别。例如，对于 ChatGLM，无需指定 target_modeules，peft 可以自行找到；对于 BaiChuan，就需要手动指定。task_type 是模型的任务类型，大模型一般都是 CAUSAL_LM 即传统语言模型。

然后获取 LoRA 模型：

```
model = get_peft_model(model, peft_config)
```

此处的 get_peft_model 的底层操作，即为上文分析的具体实现。

最后使用 transformers 提供的 Trainer 进行训练即可，训练占用的显存就会有大幅度的降低：

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset= IterableWrapper(train_dataset),
    tokenizer=tokenizer
)
trainer.train()
```

如果是应用在 DPO、KTO 上，则也相同的加入 LoRA 参数并通过 `get_peft_model` 获取一个 LoRA 模型即可，其他的不需要进行任何修改。但要注意的是，LoRA 微调能够大幅度降低显卡占用，且在下游任务适配上能够取得较好的效果，但如果是需要学习对应知识的任务，LoRA 由于只调整低秩矩阵，难以实现知识的注入，一般效果不佳，因此不推荐使用 LoRA 进行模型预训练或后训练。

参考资料

[1] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. (2019). *Parameter-Efficient Transfer Learning for NLP.* arXiv preprint arXiv:1902.00751.

[2] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. (2021). *LoRA: Low-Rank Adaptation of Large Language Models.* arXiv preprint arXiv:2106.09685.

[3] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. (2020). *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning.* arXiv preprint arXiv:2012.13255.

[4] Xiang Lisa Li 和 Percy Liang. (2021). *Prefix-Tuning: Optimizing Continuous Prompts for Generation.* arXiv preprint arXiv:2101.00190.

大模型应用

7.1 LLM 的评测

近年来，随着人工智能领域的迅猛发展，大规模预训练语言模型（简称大模型）成为了推动技术进步的核心力量。这些大模型在自然语言处理等任务中展现出了令人惊叹的能力。然而，要准确衡量一个大模型的性能，必须依靠科学而合理的评测。

什么是大模型评测？大模型评测就是通过各种标准化的方法和数据集，对大模型在不同任务上的表现进行量化和比较。这些评测不仅包括模型在特定任务上的准确性，还涉及模型的泛化能力、推理速度、资源消耗等多个方面。通过评测，我们能够更全面地了解大模型的实际表现，以及它们在现实世界中的应用潜力。

大模型的开发成本高昂，涉及大量的计算资源和数据，因此评测对于确保模型的实际价值至关重要。首先，评测能够揭示模型在各种任务中的表现，帮助研究人员和企业判断模型的适用性和可靠性。其次，评测可以暴露模型的潜在弱点，例如偏见、鲁棒性问题等，从而为进一步优化和改进提供依据。此外，公平、公开的评测还为学术界和工业界提供了一个共同的标准，促进了技术的交流与进步。

7.1.1 LLM 的评测数据集

在大模型的评测过程中，使用标准化的评测集至关重要。目前，主流的大模型评测集主要从以下几个方面进行评估，每个评测集都有其独特的用途和典型应用场景：

1. **通用评测集**：
    - **MMLU（Massive Multitask Language Understanding）**：MMLU评测模型在多种任务中的理解能力，包括各类学科和知识领域。具体包含了历史、数学、物理、生物、法律等任务类型，全面考察模型在不同学科的知识储备和语言理解能力。
2. **工具使用评测集**：
    - **BFCL V2**：用于评测模型在复杂工具使用任务中的表现，特别是在执行多步骤操作时的正确性和效率。这些任务通常涉及与数据库交互或执行特定指令，以模拟实际工具使用场景。
    - **Nexus**：用于测试模型在多步骤操作中的工具使用能力，主要评估其在多任务操作中的协调性和任务管理能力，如进行文件操作、数据整合等复杂流程。
3. **数学评测集**：
    - **GSM8K**：GSM8K是一个包含小学数学问题的数据集，用于测试模型的数学推理和逻辑分析能力。具体任务包括算术运算、简单方程求解、数字推理等。GSM8K中的问题虽然看似简单，但模型需要理解问题语义并进行正确的数学运算，体现了逻辑推理和语言理解的双重挑战。
    - **MATH**：MATH数据集用于测试模型在更复杂的数学问题上的表现，包括代数和几何。
4. **推理评测集**：
    - **ARC Challenge**：ARC Challenge评测模型在科学推理任务中的表现，尤其是常识性和科学性问题的解答，典型应用场景包括科学考试题解答和百科问答系统的开发。
    - **GPQA**：用于评测模型在零样本条件下对开放性问题的回答能力，通常应用于客服聊天机器人和知识问答系统中，帮助模型在缺乏特定领域数据的情况下给出合理的回答。
    - **HellaSwag**：评测模型在复杂语境下选择最符合逻辑的答案的能力，适用于故事续写、对话生成等需要高水平理解和推理的场景。

5. **长文本理解评测集**：
    - **InfiniteBench/En.MC**：评测模型在处理长文本阅读理解方面的能力，尤其是对科学文献的理解，适用于学术文献自动摘要、长篇报道分析等应用场景。
    - **NIH/Multi-needle**：用于测试模型在多样本长文档环境中的理解和总结能力，应用于政府报告解读、企业内部长文档分析等需要处理海量信息的场景。
6. **多语言评测集**：
    - **MGSM**：用于评估模型在不同语言下的数学问题解决能力，考察模型的多语言适应性，尤其适用于国际化环境中的数学教育和跨语言技术支持场景。

这些评测集的多样性帮助我们全面评估大模型在不同任务和应用场景中的表现，确保模型在处理多样化任务时能够保持高效和精准的表现。例如，在MMLU评测中，某些大模型在历史、物理等学科任务中表现优异，展现出对多领域知识的深度理解；在GSM8K数学评测中，最新的大模型在算术和方程求解方面表现接近甚至超越了一些人类基准，显示出在复杂数学推理任务中的潜力。这些实际评测结果展示了模型在各类复杂任务中的进步和应用潜力。

7.1.2 主流的评测榜单

大模型的评测不仅限于使用特定的数据集，许多机构还会根据评测结果发布模型排行榜，这些榜单为学术界和工业界提供了重要的参考，帮助他们了解当前最前沿的技术和模型。以下是一些主流的评测榜单：

Open LLM Leaderboard

Open LLM Leaderboard 为由 Hugging Face 提供的开放式榜单，汇集了多个开源大模型的评测结果，帮助用户了解不同模型在各种任务上的表现。该榜单通过多个标准化测试集来评估模型的性能，并通过持续更新的方式反映最新的技术进展，为研究者和开发者提供了高价值的对比参考，如图7.1所示。

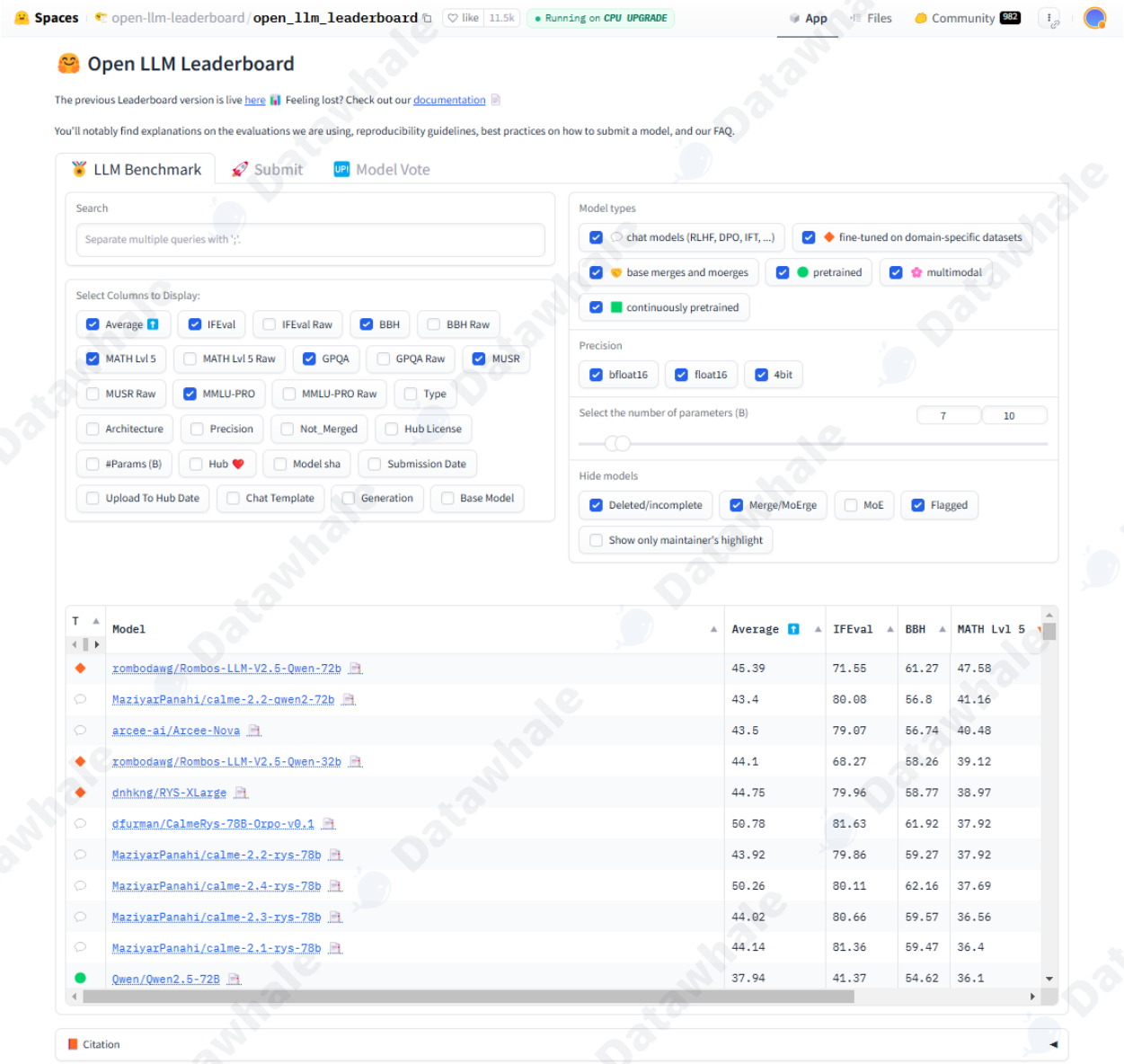


图 7.1 Open LLM Leaderboard

Lmsys Chatbot Arena Leaderboard

由lmsys提供的聊天机器人评测榜单，通过多维度的评估，展示各类大模型在对话任务中的能力。该榜单采用真实用户与模型交互的方式来评测对话质量，重点考察模型的自然语言生成能力、上下文理解能力以及用户满意度，是当前评估聊天机器人性能的重要工具，如图7.2所示。

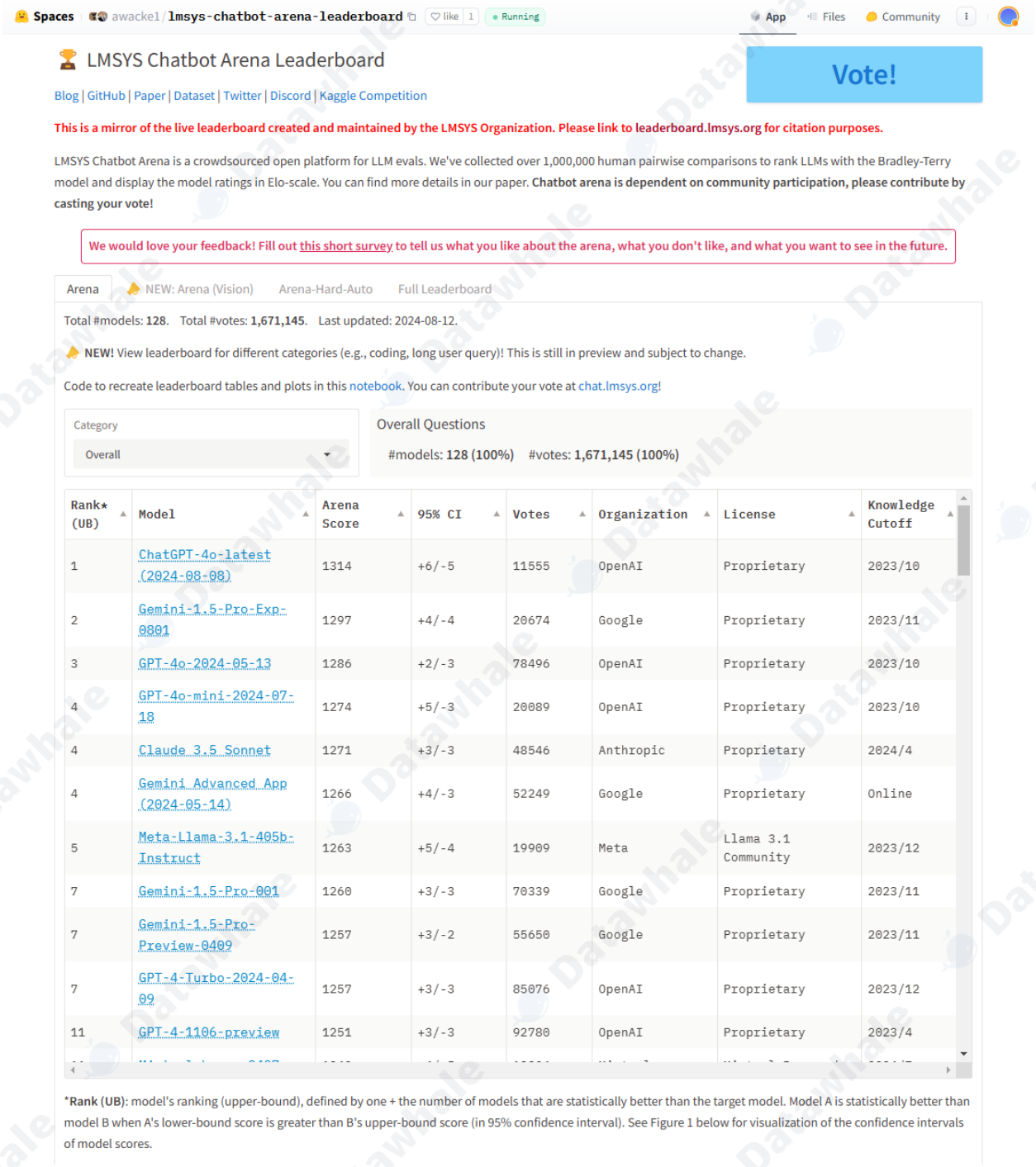


图7.2 Lmsys Chatbot Arena Leaderboard

OpenCompass

OpenCompass 是国内的评测榜单，针对大模型在多种语言和任务上的表现进行评估，提供了中国市场特定应用的参考。该榜单结合了中文语言理解和多语言能力的测试，以适应本地化需求，并特别关注大模型在中文语境下的准确性、鲁棒性和适应性，为国内企业和研究者选择合适的模型提供了重要参考。

图7.3 OpenCompass

7.1.3 特定的评测榜单

另外，还有针对不同领域特定任务的大模型评测榜单，如图7.4所示。这些榜单专注于特定应用领域，帮助用户了解大模型在某一垂直领域的能力：

- 金融榜：基于CFBenchmark评测集，评估大模型在金融自然语言处理、金融预测计算、金融分析与安全检查等多项基础任务中的能力。由同济大学与上海人工智能实验室及东方财经提供。
- 安全榜：基于Flames评测集，评估大模型在公平、安全、数据保护以及合法五大维度的抗性，帮助深入了解模型在安全性上的表现。由上海人工智能实验室与复旦大学提供。
- 通识榜：基于BotChat评测集，评估大语言模型生成日常多轮对话能力的综合程度，判断模型在对话中是否具备类人水平。由上海人工智能实验室提供。
- 法律榜：基于LawBench评测集，评估模型在法律领域的理解、推理和应用能力，涵盖法律问题回答、文本生成、法律判例分析等任务。由南京大学提供。

- 医疗榜：基于MedBench评测集，评估大语言模型在医学知识问答、安全伦理理解等方面的表现。由上海人工智能实验室提供。

图7.4 垂直领域榜单

7.2 RAG

7.2.1 RAG 的基本原理

大语言模型（LLM）在生成内容时，虽然具备强大的语言理解和生成能力，但也面临着一些挑战。例如，LLM有时会生成不准确或误导性的内容，这被称为大模型“幻觉”。此外，模型所依赖的训练数据可能过时，尤其在面对最新的信息时，生成结果的准确性和时效性难以保证。对于特定领域的专业知识，LLM 的处理效率也较低，无法深入理解复杂的领域知识。因此，如何提升大模型的生成质量和效率，成为了当前研究的重要方向。

在这样的背景下，检索增强生成（Retrieval-Augmented Generation，RAG）技术应运而生，成为AI领域中的一大创新趋势。RAG 在生成答案之前，首先从外部的大规模文档数据库中检索出相关信息，并将这些信息融入到生成过程之中，从而指导和优化语言模型的输出。这一流程不仅极大地提升了内容生成的准确性和相关性，还使得生成的内容更加符合实时性要求。

RAG 的核心原理在于将“检索”与“生成”结合：当用户提出查询时，系统首先通过检索模块找到与问题相关的文本片段，然后将这些片段作为附加信息传递给语言模型，模型据此生成更为精准和可靠的回答。通过这种方式，RAG 有效缓解了大语言模型的“幻觉”问题，因为生成的内容建立在真实文档的基础上，使得答案更具可追溯性和可信度。同时，由于引入了最新的信息源，RAG 技术大大加快了知识更新速度，使得系统可以及时吸收和反映最新的领域动态。

7.2.2 搭建一个 RAG 框架

接下来我会带领大家一步一步实现一个简单的RAG模型，这个模型是基于RAG的一个简化版本，我们称之为 Tiny-RAG 。Tiny-RAG只保留了 RAG 的核心功能，即检索和生成，其目的是帮助大家更好地理解 RAG 模型的原理和实现。

Step 1: RAG流程介绍

RAG通过在语言模型生成答案之前，先从广泛的文档数据库中检索相关信息，然后利用这些信息来引导生成过程，从而极大地提升了内容的准确性和相关性。RAG有效地缓解了幻觉问题，提高了知识更新的速度，并增强了内容生成的可追溯性，使得大型语言模型在实际应用中变得更加实用和可信。

RAG的基本结构有哪些呢?

- 向量化模块：用来将文档片段向量化。
- 文档加载和切分模块：用来加载文档并切分成文档片段。
- 数据库：存放文档片段及其对应的向量表示。
- 检索模块：根据 Query（问题）检索相关的文档片段。
- 大模型模块：根据检索到的文档回答用户的问题。

上述也就是 TinyRAG 的所有模块内容，如图7.5所示。

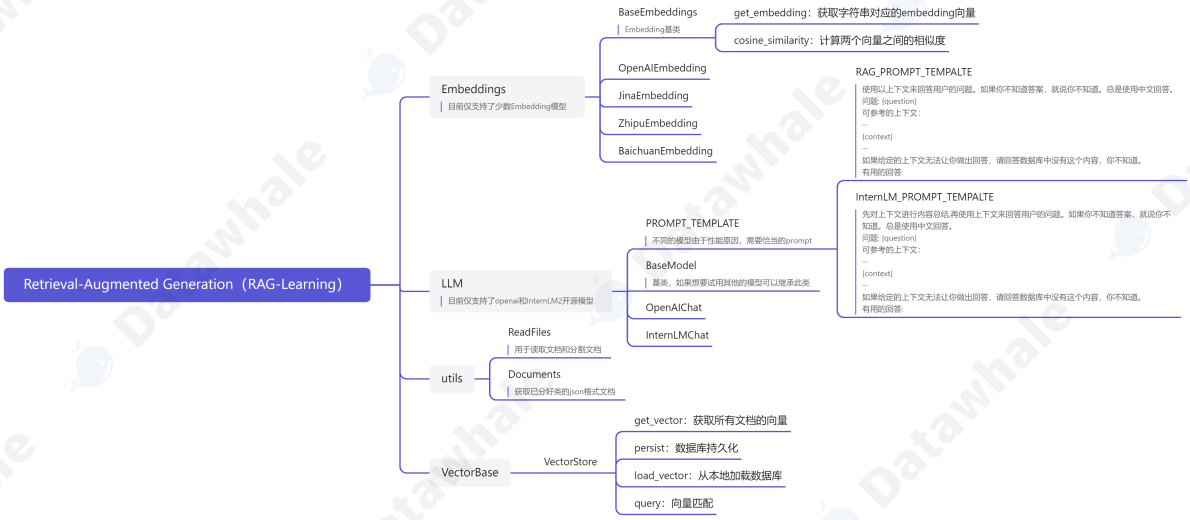


图7.5 TinyRAG 项目结构

接下来，让我们梳理一下RAG的流程是什么样的呢?

- **索引**：将文档库分割成较短的片段，并通过编码器构建向量索引。
- **检索**：根据问题和片段的相似度检索相关文档片段。
- **生成**：以检索到的上下文为条件，生成问题的回答。

如下图7.6所示的流程图，图片出处 ***Retrieval-Augmented Generation for Large Language Models: A Survey***

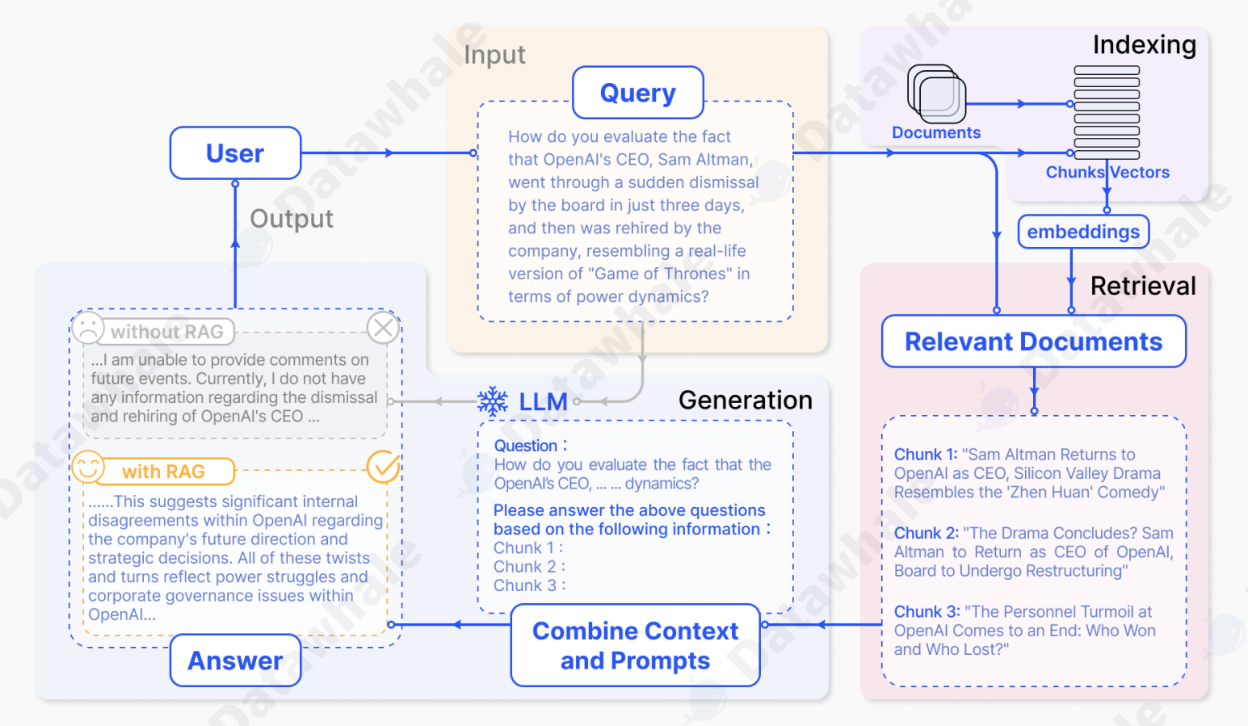


图7.6 RAG 流程图

Step 2: 向量化

首先我们来动手实现一个向量化的类，这是RAG架构的基础。向量化类主要用来将文档片段向量化，将一段文本映射为一个向量。

首先我们要设置一个 `BaseEmbeddings` 基类，这样我们在使用其他模型时，只需要继承这个基类，然后在此基础上进行修改即可，方便代码扩展。

```
class BaseEmbeddings:
    """
    Base class for embeddings
    """
    def __init__(self, path: str, is_api: bool) -> None:
        self.path = path
        self.is_api = is_api

    def get_embedding(self, text: str, model: str) -> List[float]:
        raise NotImplementedError

    @classmethod
    def cosine_similarity(cls, vector1: List[float], vector2: List[float]) -> float:
        """
        calculate cosine similarity between two vectors
        """
        dot_product = np.dot(vector1, vector2)
        magnitude = np.linalg.norm(vector1) * np.linalg.norm(vector2)
        if not magnitude:
            return 0
        return dot_product / magnitude
```

`BaseEmbeddings`基类有两个主要方法：`get_embedding`和`cosine_similarity`。`get_embedding`用于获取文本的向量表示，`cosine_similarity`用于计算两个向量之间的余弦相似度。在初始化类时设置了模型的路径和是否是API模型，例如使用OpenAI的Embedding API需要设置`self.is_api=True`。

继承`BaseEmbeddings`类只需要实现`get_embedding`方法，`cosine_similarity`方法会被继承下来。这就是编写基类的好处。

```
class OpenAIEmbedding(BaseEmbeddings):
    """
    class for OpenAI embeddings
    """
    def __init__(self, path: str = '', is_api: bool = True) -> None:
        super().__init__(path, is_api)
        if self.is_api:
            from openai import OpenAI
            self.client = OpenAI()
            self.client.api_key = os.getenv("OPENAI_API_KEY")
            self.client.base_url = os.getenv("OPENAI_BASE_URL")

    def get_embedding(self, text: str, model: str = "text-embedding-3-large") ->
List[float]:
        if self.is_api:
            text = text.replace("\n", " ")
            return self.client.embeddings.create(input=[text],
model=model).data[0].embedding
        else:
            raise NotImplementedError
```

Step 3: 文档加载和切分

接下来我们来实现一个文档加载和切分的类，这个类主要用于加载文档并将其切分成文档片段。

文档可以是文章、书籍、对话、代码等文本内容，例如pdf文件、md文件、txt文件等。完整代码可以在*RAG/utils.py*文件中找到。该代码支持加载pdf、md、txt等类型的文件，只需编写相应的函数即可。

```
def read_file_content(cls, file_path: str):
    # 根据文件扩展名选择读取方法
    if file_path.endswith('.pdf'):
        return cls.read_pdf(file_path)
    elif file_path.endswith('.md'):
        return cls.read_markdown(file_path)
    elif file_path.endswith('.txt'):
        return cls.read_text(file_path)
    else:
        raise ValueError("Unsupported file type")
```

文档读取后需要进行切分。我们可以设置一个最大的Token长度，然后根据这个最大长度来切分文档。切分文档时最好以句子为单位（按`\n`粗切分），并保证片段之间有一些重叠内容，以提高检索的准确性。

```
def get_chunk(cls, text: str, max_token_len: int = 600, cover_content: int = 150):
```

```
    chunk_text = []

    curr_len = 0
    curr_chunk = ''

    lines = text.split('\n')

    for line in lines:
        line = line.replace(' ', '')
        line_len = len(enc.encode(line))
        if line_len > max_token_len:
            print('warning line_len = ', line_len)
        if curr_len + line_len <= max_token_len:
            curr_chunk += line
            curr_chunk += '\n'
            curr_len += line_len
            curr_len += 1
        else:
            chunk_text.append(curr_chunk)
            curr_chunk = curr_chunk[-cover_content:] + line
            curr_len = line_len + cover_content

    if curr_chunk:
        chunk_text.append(curr_chunk)

    return chunk_text
```

Step 4: 数据库与向量检索

完成文档切分和Embedding模型加载后，需要设计一个向量数据库来存放文档片段和对应的向量表示，以及设计一个检索模块用于根据Query检索相关文档片段。

向量数据库的功能包括：

- `persist`：数据库持久化保存。
- `load_vector`：从本地加载数据库。
- `get_vector`：获取文档的向量表示。
- `query`：根据问题检索相关文档片段。

完整代码可以在 ***RAG/VectorBase.py*** 文件中找到。

```
class VectorStore:
    def __init__(self, document: List[str] = ['']) -> None:
        self.document = document

    def get_vector(self, EmbeddingModel: BaseEmbeddings) -> List[List[float]]:
        # 获得文档的向量表示
        pass

    def persist(self, path: str = 'storage'):
        # 数据库持久化保存
```

```
        pass

    def load_vector(self, path: str = 'storage'):
        # 从本地加载数据库
        pass

    def query(self, query: str, EmbeddingModel: BaseEmbeddings, k: int = 1) -> List[str]:
        # 根据问题检索相关文档片段
        pass
```

`query` 方法用于将用户提出的问题向量化，然后在数据库中检索相关文档片段并返回结果。

```
def query(self, query: str, EmbeddingModel: BaseEmbeddings, k: int = 1) -> List[str]:
    query_vector = EmbeddingModel.get_embedding(query)
    result = np.array([self.get_similarity(query_vector, vector) for vector in
self.vectors])
    return np.array(self.document)[result.argsort()[-k:][::-1]].tolist()
```

Step 5: 大模型模块

接下来是大模型模块，用于根据检索到的文档回答用户的问题。

首先实现一个基类，这样可以方便扩展其他模型。

```
class BaseModel:
    def __init__(self, path: str = '') -> None:
        self.path = path

    def chat(self, prompt: str, history: List[dict], content: str) -> str:
        pass

    def load_model(self):
        pass
```

`BaseModel` 包含两个方法：`chat` 和 `load_model`。对于本地化运行的开源模型需要实现 `load_model`，而API模型则不需要。

下面以 ***InternLM2-chat-7B*** 模型为例：

```
class InternLMChat(BaseModel):
    def __init__(self, path: str = '') -> None:
        super().__init__(path)
        self.load_model()

    def chat(self, prompt: str, history: List = [], content: str='') -> str:
        prompt = PROMPT_TEMPLATE['InternLM_PROMPT_TEMPLATE'].format(question=prompt,
context=content)
        response, history = self.model.chat(self.tokenizer, prompt, history)
        return response

    def load_model(self):
```

```
        import torch
        from transformers import AutoTokenizer, AutoModelForCausalLM
        self.tokenizer = AutoTokenizer.from_pretrained(self.path, trust_remote_code=True)
        self.model = AutoModelForCausalLM.from_pretrained(self.path,
torch_dtype=torch.float16, trust_remote_code=True).cuda()
```

可以用一个字典来保存所有的prompt，方便维护：

```
PROMPT_TEMPLATE = dict(
    InternLM_PROMPT_TEMPLATE="""先对上下文进行内容总结,再使用上下文来回答用户的问题。如果你不知道答
案，就说你不知道。总是使用中文回答。
        问题：{question}
        可参考的上下文：
        ···
        {context}
        ···
        如果给定的上下文无法让你做出回答，请回答数据库中没有这个内容，你不知道。
        有用的回答:"""
)
```

这样我们就可以利用InternLM2模型来做RAG啦！

Step 6: Tiny-RAG Demo

接下来，我们来看看Tiny-RAG的Demo吧！

```
from RAG.VectorBase import VectorStore
from RAG.utils import ReadFiles
from RAG.LLM import OpenAIChat, InternLMChat
from RAG.Embeddings import JinaEmbedding, ZhipuEmbedding

# 没有保存数据库
docs = ReadFiles('./data').get_content(max_token_len=600, cover_content=150) # 获取data目录
下的所有文件内容并分割
vector = VectorStore(docs)
embedding = ZhipuEmbedding() # 创建EmbeddingModel
vector.get_vector(EmbeddingModel=embedding)
vector.persist(path='storage') # 将向量和文档内容保存到storage目录，下次再用可以直接加载本地数据库

question = 'git的原理是什么？'

content = vector.query(question, model='zhipu', k=1)[0]
chat = InternLMChat(path='model_path')
print(chat.chat(question, [], content))
```

也可以从本地加载已处理好的数据库：

```
from RAG.VectorBase import VectorStore
from RAG.utils import ReadFiles
from RAG.LLM import OpenAIChat, InternLMChat
```

```
from RAG.Embeddings import JinaEmbedding, ZhipuEmbedding

# 保存数据库之后
vector = VectorStore()

vector.load_vector('./storage') # 加载本地数据库

question = 'git的原理是什么？'

embedding = ZhipuEmbedding() # 创建EmbeddingModel

content = vector.query(question, EmbeddingModel=embedding, k=1)[0]
chat = InternLMChat(path='model_path')
print(chat.chat(question, [], content))
```

7.3 Agent

7.3.1 什么是 LLM Agent?

简单来说，大模型Agent是一个以LLM为核心“大脑”，并赋予其自主规划、记忆和使用工具能力的系统。 它不再仅仅是被动地响应用户的提示（Prompt），而是能够：

1. 理解目标（Goal Understanding）： 接收一个相对复杂或高层次的目标（例如，“帮我规划一个周末去北京的旅游行程并预订机票酒店”）。
2. 自主规划（Planning）： 将大目标分解成一系列可执行的小步骤（例如，“搜索北京景点”、“查询天气”、“比较机票价格”、“查找合适的酒店”、“调用预订API”等）。
3. 记忆（Memory）： 拥有短期记忆（记住当前任务的上下文）和长期记忆（从过去的交互或外部知识库中学习和检索信息）。
4. 工具使用（Tool Use）： 调用外部API、插件或代码执行环境来获取信息（如搜索引擎、数据库）、执行操作（如发送邮件、预订服务）或进行计算。
5. 反思与迭代（Reflection & Iteration）： （在更高级的Agent中）能够评估自己的行为和结果，从中学习并调整后续计划。

传统的LLM像一个知识渊博但只能纸上谈兵的图书馆员，而 LLM Agent 则更像一个全能的私人助理，不仅懂得多，还能跑腿办事，甚至能主动思考最优方案。

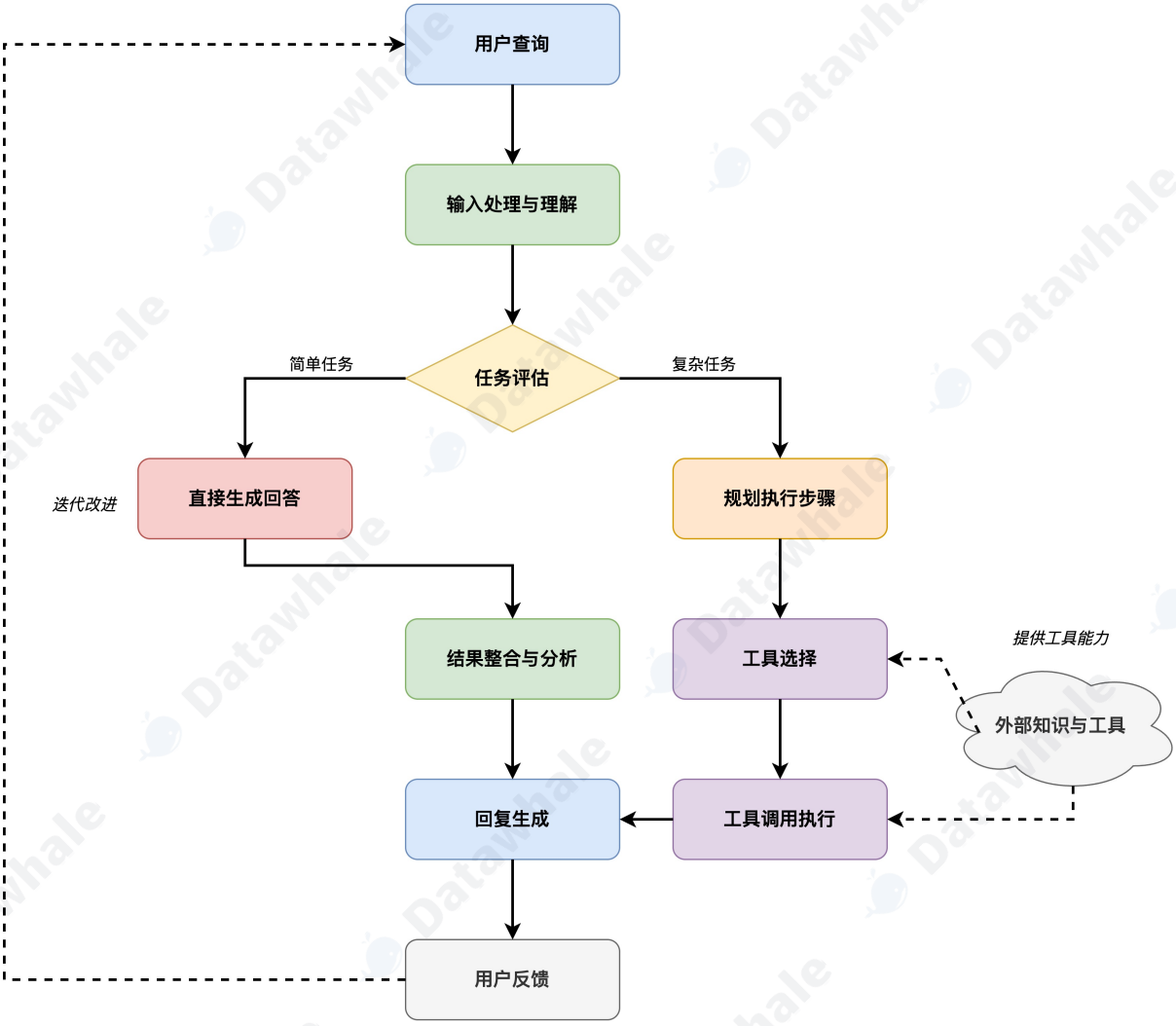


图7.7 Agent 工作原理

LLM Agent 通过将大型语言模型的强大语言理解和生成能力与规划、记忆和工具使用等关键模块相结合，实现了超越传统大模型的自主性和复杂任务处理能力，这种能力使得 LLM Agent 在许多垂直领域（如法律、医疗、金融等）都具有广泛的应用潜力，如图7.7所示 Agent 工作原理。

7.3.2 LLM Agent 的类型

虽然LLM Agent的概念还在快速发展中，但根据其设计理念和能力侧重，我们可以大致将其分为几类：

任务导向型Agent（Task-Oriented Agents）：

- 特点： 专注于完成特定领域的、定义明确的任务，例如客户服务、代码生成、数据分析等。
- 工作方式： 通常有预设的流程和可调用的特定工具集。LLM主要负责理解用户意图、填充任务槽位、生成回应或调用合适- 的工具。
- 例子： 专门用于预订餐厅的聊天机器人、辅助编程的代码助手（如GitHub Copilot在某些高级功能上体现了Agent特性）。

规划与推理型Agent（Planning & Reasoning Agents）：

- 特点： 强调自主分解复杂任务、制定多步计划，并根据环境反馈进行调整的能力。它们通常需要更强的推理能力。

- 工作方式： 常采用特定的思维框架，如ReAct (Reason+Act)，让模型先进行“思考”（Reasoning）分析当前情况和所需行动，然后执行“行动”（Action）调用工具，再根据工具返回结果进行下一轮思考。Chain-of-Thought (CoT) 等提示工程技术也是其推理的基础。
- 例子： 需要整合网络搜索、计算器、数据库查询等多种工具来回答复杂问题的研究型Agent，或者能够自主完成“写一篇关于XX主题的报告，并配上相关数据图表”这类任务的Agent。

多Agent系统（Multi-Agent Systems）：

- 特点： 由多个具有不同角色或能力的Agent协同工作，共同完成一个更宏大的目标。
- 工作方式： Agent之间可以进行通信、协作、辩论甚至竞争。例如，一个Agent负责规划，一个负责执行，一个负责审查。
- 例子： 模拟软件开发团队（产品经理Agent、程序员Agent、测试员Agent）来自动生成和测试代码；模拟一个公司组织结构来完成商业策划。AutoGen、ChatDev等框架支持这类系统的构建。

探索与学习型Agent（Exploration & Learning Agents）：

- 特点： 这类Agent不仅执行任务，还能在与环境的交互中主动学习新知识、新技能或优化自身策略，类似于强化学习中的Agent概念。
- 工作方式： 可能包含更复杂的记忆和反思机制，能够根据成功或失败的经验调整未来的规划和行动。
- 例子： 能在未知软件环境中自主探索学习如何操作的Agent，或者在玩游戏时不断提升策略的Agent。

7.3.3 动手构造一个 Tiny-Agent

我们来基于 `openai` 库和其 `tool_calls` 功能，动手构造一个 Tiny-Agent，这个 Agent 是一个简单的任务导向型 Agent，它能够根据用户的输入，回答一些简单的问题。

最终的实现效果如图7.8所示：

```
(nlp) (base) kmno4-zx@KMnO4-zxdeMacBook-Air TinyAgent % python demo.py
User: 你好啊
Assistant:  你好！有什么我可以帮忙的吗？
User: 现在时间几点了？
调用工具： ['get_current_datetime']
Assistant:  当前的时间是2025年04月26日 19点21分28秒。请注意，这个时间是示例时间，真实时间可能会有所不同。如果你需要获取准确的当前时间，请告诉我，我会为你查询。
User: strewbreeeey中有几个e?
调用工具： ['count_letter_in_string']
Assistant:  在字符串"strewbreeeey"中有5个'e'。
User: OK，你回答的很好，现在时间是几滴啊了？
调用工具： ['get_current_datetime']
Assistant:  当前的时间是2025年04月26日 19点21分59秒。看起来你可能是在开玩笑说“几点了”为“几滴啊了”，不过时间还是准确提供的。如果有其他问题，请继续提问！
User: 好的 谢谢你
Assistant:  不客气！如果还有其他问题，随时欢迎提问。祝你有个美好的一天！
User: exit
(nlp) (base) kmno4-zx@KMnO4-zxdeMacBook-Air TinyAgent %
```

图7.8 效果示意图

Step 1：初始化客户端和模型

首先，我们需要一个能够调用大模型的客户端。这里我们使用 `openai` 库，并配置其指向一个兼容 OpenAI API 的服务终端，例如 SiliconFlow。同时，指定要使用的模型，如 `Qwen/Qwen2.5-32B-Instruct`。

```python
from openai import OpenAI

# 初始化 OpenAI 客户端
client = OpenAI(
    api_key="YOUR_API_KEY",  # 替换为你的 API Key
    base_url="https://api.siliconflow.cn/v1", # 使用 SiliconFlow 的 API 地址
)

# 指定模型名称
model_name = "Qwen/Qwen2.5-32B-Instruct"
```

> **注意:** 你需要将 `YOUR_API_KEY` 替换为你从 SiliconFlow 或其他服务商获取的有效 API Key。

Step 2: 定义工具函数

我们在 `src/tools.py` 文件中定义 Agent 可以使用的工具函数。每个函数都需要有清晰的文档字符串（docstring），描述其功能和参数，因为这将用于自动生成工具的 JSON Schema。

```python
# src/tools.py
from datetime import datetime

# 获取当前日期和时间
def get_current_datetime() -> str:
    """
    获取当前日期和时间。
    :return: 当前日期和时间的字符串表示。
    """
    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
    return formatted_datetime

def add(a: float, b: float):
    """
    计算两个浮点数的和。
    :param a: 第一个浮点数。
    :param b: 第二个浮点数。
    :return: 两个浮点数的和。
    """
    return str(a + b)

def compare(a: float, b: float):
    """
    比较两个浮点数的大小。
    :param a: 第一个浮点数。
    :param b: 第二个浮点数。
    :return: 比较结果的字符串表示。
    """
    if a > b:
        return f'{a} is greater than {b}'
    elif a < b:
        return f'{b} is greater than {a}'
```

```
    else:
        return f'{a} is equal to {b}'

def count_letter_in_string(a: str, b: str):
    """
    统计字符串中某个字母的出现次数。
    :param a: 要搜索的字符串。
    :param b: 要统计的字母。
    :return: 字母在字符串中出现的次数。
    """
    return str(a.count(b))

# ... (可能还有其他工具函数)
```

为了让 OpenAI API 理解这些工具，我们需要将它们转换成特定的 JSON Schema 格式。这可以通过 `src/utils.py` 中的 `function_to_json` 辅助函数完成。

```
# src/utils.py (部分)
import inspect

def function_to_json(func) -> dict:
    # ... (函数实现细节)
    # 返回符合 OpenAI tool schema 的字典
    return {
        "type": "function",
        "function": {
            "name": func.__name__,
            "description": inspect.getdoc(func),
            "parameters": {
                "type": "object",
                "properties": parameters,
                "required": required,
            },
        },
    }
```

Step 3: 构造 Agent 类

我们在 `src/core.py` 文件中定义 `Agent` 类。这个类负责管理对话历史、调用 OpenAI API、处理工具调用请求以及执行工具函数。

```
# src/core.py (部分)
from openai import OpenAI
import json
from typing import List, Dict, Any
from utils import function_to_json
# 导入定义好的工具函数
from tools import get_current_datetime, add, compare, count_letter_in_string

SYSREM_PROMPT = """
你是一个叫不要葱姜蒜的人工智能助手。你的输出应该与用户的语言保持一致。
```

```
当用户的问题需要调用工具时，你可以从提供的工具列表中调用适当的工具函数。
"""

class Agent:
    def __init__(self, client: OpenAI, model: str = "Qwen/Qwen2.5-32B-Instruct", tools:
List=[], verbose : bool = True):
        self.client = client
        self.tools = tools
        self.model = model
        self.messages = [
            {"role": "system", "content": SYSREM_PROMPT},
        ]
        self.verbose = verbose

    def get_tool_schema(self) -> List[Dict[str, Any]]:
        # 获取所有工具的 JSON 模式
        return [function_to_json(tool) for tool in self.tools]

    def handle_tool_call(self, tool_call):
        # 处理工具调用
        function_name = tool_call.function.name
        function_args = tool_call.function.arguments
        function_id = tool_call.id

        function_call_content = eval(f"{function_name}(**{function_args})")

        return {
            "role": "tool",
            "content": function_call_content,
            "tool_call_id": function_id,
        }

    def get_completion(self, prompt) -> str:

        self.messages.append({"role": "user", "content": prompt})

        # 获取模型的完成响应
        response = self.client.chat.completions.create(
            model=self.model,
            messages=self.messages,
            tools=self.get_tool_schema(),
            stream=False,
        )

        # 检查模型是否调用了工具
        if response.choices[0].message.tool_calls:
            self.messages.append({"role": "assistant", "content":
response.choices[0].message.content})
            # 处理工具调用
            tool_list = []
            for tool_call in response.choices[0].message.tool_calls:
                # 处理工具调用并将结果添加到消息列表中
```

```
                self.messages.append(self.handle_tool_call(tool_call))
                tool_list.append([tool_call.function.name, tool_call.function.arguments])
            if self.verbose:
                print("调用工具：", response.choices[0].message.content, tool_list)
            # 再次获取模型的完成响应，这次包含工具调用的结果
            response = self.client.chat.completions.create(
                model=self.model,
                messages=self.messages,
                tools=self.get_tool_schema(),
                stream=False,
            )

        # 将模型的完成响应添加到消息列表中
        self.messages.append({"role": "assistant", "content":
response.choices[0].message.content})
        return response.choices[0].message.content
```

Agent 的工作流程如下：

1. 接收用户输入。
2. 调用大模型（如 Qwen），并告知其可用的工具及其 Schema。
3. 如果模型决定调用工具，Agent 会解析请求，执行相应的 Python 函数。
4. Agent 将工具的执行结果返回给模型。
5. 模型根据工具结果生成最终回复。
6. Agent 将最终回复返回给用户。

如图7.9所示，Agent 调用工具流程：

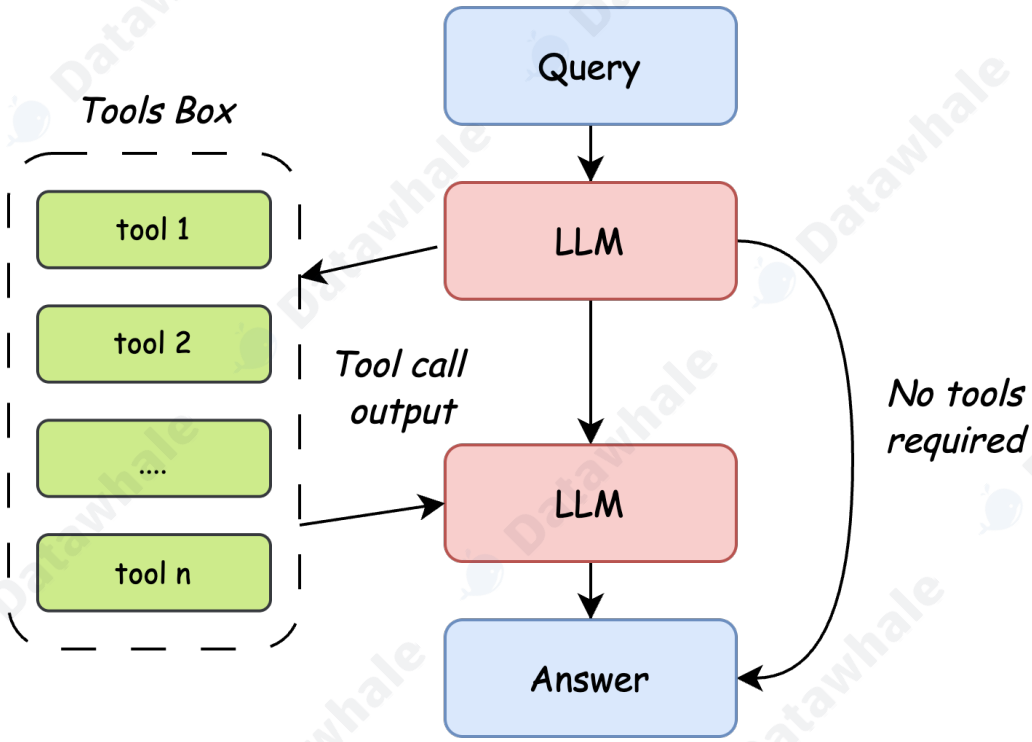


图7.9 Agent 工作流程

Step 4: 运行 Agent

现在我们可以实例化并运行 Agent。在 `demo.py` 的 `if __name__ == "__main__":` 部分提供了一个简单的命令行交互示例。

```
# demo.py (部分)
if __name__ == "__main__":
    client = OpenAI(
        api_key="YOUR_API_KEY", # 替换为你的 API Key
        base_url="https://api.siliconflow.cn/v1",
    )

    # 创建 Agent 实例，传入 client、模型名称和工具函数列表
    agent = Agent(
        client=client,
        model="Qwen/Qwen2.5-32B-Instruct",
        tools=[get_current_datetime, add, compare, count_letter_in_string],
        verbose=True # 设置为 True 可以看到工具调用信息
    )

    # 开始交互式对话循环
    while True:
        # 使用彩色输出区分用户输入和AI回答
        prompt = input("\033[94mUser: \033[0m")  # 蓝色显示用户输入提示
        if prompt.lower() == "exit":
            break
        response = agent.get_completion(prompt)
        print("\033[92mAssistant: \033[0m", response)  # 绿色显示AI助手回答
```

运行 `python demo.py` 后，你可以开始提问。如果问题需要调用工具，Agent 会自动处理。

示例交互:

```
User: 你好
Assistant:  你好！有什么可以帮助你的吗?
User: 9.12和9 .2哪个更大?
调用工具:  ['compare']
Assistant:  9.2 比 9.12 更大。
User: 为什么?
Assistant:  当我们比较9.12和9.2时，可以将它们看作是9.12和9.20。由于9.20在小数点后第二位是0，而9.12在小数点后第二位是2，所以在小数点后第一位相等的情况下，9.20（即9.2）大于9.12。因此，9.2 比 9.12 更大。
User: strawberry中有几个r?
调用工具:  ['count_letter_in_string']
Assistant:  单词 "strawberry" 中有3个字母 'r'。
User: 你确信嘛?
调用工具:  ['count_letter_in_string']
Assistant:  是的，我确定。单词 "strawberry" 中确实有3个字母 'r'。让我们再次确认一下，"strawberry" 中的 'r' 确实出现了3次。
User: 好的 你很薄，现在几点 了?
调用工具:  ['get_current_datetime']
```

```
Assistant:  当前的时间是2025年4月26日17:01:33。不过，我注意到您提到“你很薄”，这似乎是一个打字错误，如果您有任何其他问题或者需要进一步的帮助，请告诉我！
User: exit
```

参考文献

[1] Hugging Face. (2023). *Open LLM Leaderboard: 开源大语言模型基准测试平台*. https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard

[2] awacke1. (2023). *LMSYS Chatbot Arena Leaderboard: 大型语言模型竞技场评估平台*. https://huggingface.co/spaces/awacke1/lmsys-chatbot-arena-leaderboard

[3] OpenCompass 团队. (2023). *OpenCompass: 大模型统一评测平台*. https://rank.opencompass.org.cn/home

[4] OpenCompass 金融榜团队. (2024). *CFBENCHMARK: 金融领域大模型评测榜单*. https://specialist.opencompass.org.cn/CFBenchmark

[5] OpenCompass 安全榜团队. (2024). *Flames: 大模型安全评测榜单*. https://flames.opencompass.org.cn/leaderboard

[6] OpenCompass 通识榜团队. (2024). *BotChat: 大模型通用对话能力评测*. https://botchat.opencompass.org.cn/

[7] OpenCompass 法律榜团队. (2024). *LawBench: 法律领域大模型评测*. https://lawbench.opencompass.org.cn/leaderboard

[8] OpenCompass 医疗榜团队. (2024). *MedBench: 医疗领域大模型评测*. https://medbench.opencompass.org.cn/leaderboard

[9] Zhi Jing, Yongye Su, and Yikun Han. (2024). *When Large Language Models Meet Vector Databases: A Survey.* arXiv preprint arXiv:2402.01763.

[10] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. (2024). *Retrieval-Augmented Generation for Large Language Models: A Survey.* arXiv preprint arXiv:2312.10997.

[11] Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, 和 Graham Neubig. (2023). *Learning to Filter Context for Retrieval-Augmented Generation.* arXiv preprint arXiv:2311.08377.

[12] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown 和 Yoav Shoham. (2023). *In-Context Retrieval-Augmented Language Models.* arXiv preprint arXiv:2302.00083.