Contents

1 概述 **5**
1.1 LLM 简史 . . . 5
1.2 LLM 列表（持续更新中） . . . 5
1.3 LLM 数据集 . . . 7
1.4 LLM 开源库 . . . 8
1.5 一些综述 . . . 8
1.6 扩展法则 . . . 9
1.6.1 openai 的扩展法则 . . . 9
1.6.2 Chinchilla 扩展法则 . . . 9
1.7 涌现能力 . . . 9
1.7.1 上下文学习 . . . 10
1.7.2 指令遵循 . . . 10
1.7.3 逐步推理 . . . 10
1.8 LLM 关键点 . . . 10
1.8.1 扩展 . . . 10
1.8.2 训练 . . . 11
1.8.3 能力引导 . . . 11
1.8.4 对齐微调 . . . 11
1.8.5 工具操作 . . . 11

2 预训练 **11**
2.1 数据收集 . . . 11
2.1.1 数据获取 . . . 11
2.1.2 数据预处理 . . . 11
2.1.3 预训练语料的重要性 . . . 12
2.2 架构 . . . 12
2.2.1 主流框架 . . . 12
2.2.1.1 讨论：为什么现在的 LLM 都是 Decoder only 的架构? . . . 13
2.2.2 组件配置 . . . 13
2.2.2.1 标准化（norm） . . . 13
2.2.2.2 激活函数 . . . 14
2.2.2.3 位置编码 . . . 15
2.2.2.4 注意力机制和 Bias . . . 15
2.2.2.5 小结 . . . 15
2.2.2.5.1 归一化位置 . . . 15
2.2.2.5.2 归一化方法 . . . 16
2.2.2.5.3 激活函数 . . . 16
2.2.2.5.4 位置嵌入 . . . 16
2.2.3 预训练任务 . . . 16
2.2.3.1 语言建模 . . . 16
2.2.3.2 去噪自编码 . . . 16
2.3 Transformer 的 FLOPS 和访存带宽 . . . 17
2.3.1 attention 的 FLOPS . . . 17
2.3.2 FFN 的 FLOPS . . . 17
2.3.3 DIN 的 FLOPS . . . 18
2.3.4 Transformer 的访存 . . . 18
2.4 模型训练 . . . 20
2.4.1 优化设置 . . . 21
2.4.2 混合精度训练 . . . 21
2.4.2.1 FP16 . . . 21
2.4.2.2 BF16 . . . 22
2.4.3 可扩展的训练 . . . 23

2.4.3.1 3D 并行 . . . 23
2.4.3.1.1 数据并行（Data Parallelism） . . . 23
2.4.3.1.2 流水线并行（Pipeline Parallelism） . . . 24
2.4.3.1.3 张量并行（Tensor Parallelism） . . . 25
2.4.3.2 ZeRO . . . 27
2.4.3.3 序列并行 . . . 28
2.4.3.4 综合对比各种并行 . . . 29
2.4.4 编译优化 . . . 30
2.4.5 flash attention . . . 31
2.5 推理速度优化 . . . 31
2.5.1 量化 . . . 31

3 微调 31
3.1 指令微调 . . . 31
3.1.1 构建格式化实例 . . . 31
3.1.1.1 格式化已有数据集 . . . 32
3.1.1.2 格式化人类需求 . . . 32
3.1.1.3 构建实例的关键 . . . 32
3.1.2 指令微调策略 . . . 32
3.1.3 指令微调效果 . . . 33
3.1.3.1 性能改进 . . . 33
3.1.3.2 任务泛化性 . . . 33
3.2 对齐微调 . . . 33
3.2.1 对齐的标准 . . . 33
3.2.2 收集人类反馈 . . . 34
3.2.2.1 选择标注人员 . . . 34
3.2.2.2 收集反馈 . . . 34
3.2.3 RLHF . . . 34
3.3 高效微调 . . . 34
3.3.1 适配器微调（adapter tuning） . . . 34
3.3.2 前缀微调（prefix tuning） . . . 35
3.3.3 提示微调（prompt tuning） . . . 35
3.3.4 低秩适配（LoRA） . . . 35
3.3.5 小结 . . . 36

4 使用 36
4.1 上下文学习 . . . 36
4.1.1 上下文学习形式 . . . 36
4.1.2 示范设计 . . . 36
4.1.2.1 示范选择 . . . 36
4.1.2.2 示范格式 . . . 37
4.1.2.3 示范顺序 . . . 37
4.1.3 底层机制 . . . 37
4.1.3.1 预训练如何影响 ICL . . . 37
4.1.3.2 LLM 如何实现 ICL . . . 38
4.2 思维链提示（CoT） . . . 38
4.2.1 使用 CoT 的 ICL . . . 38
4.2.1.1 小样本思维链 . . . 38
4.2.1.2 零样本思维链 . . . 39
4.2.2 进一步讨论 CoT . . . 39

5 能力评测 39
5.1 基础评测 . . . 39
5.1.1 语言生成 . . . 39

5.1.1.1 语言建模 . . . 39
5.1.1.2 条件文本生成 . . . 39
5.1.1.3 代码合成 . . . 39
5.1.1.4 存在问题 . . . 39
5.1.2 知识利用 . . . 39
5.1.2.1 闭卷问答 . . . 39
5.1.2.2 开卷问答 . . . 39
5.1.2.3 知识补全 . . . 40
5.1.2.4 存在问题 . . . 40
5.1.3 复杂推理 . . . 40
5.1.3.1 知识推理 . . . 40
5.1.3.2 符号推理 . . . 40
5.1.3.3 数学推理 . . . 40
5.1.3.4 存在问题 . . . 40
5.2 高级评估 . . . 40
5.2.1 与人类对齐 . . . 40
5.2.2 与外部环境互动 . . . 40
5.2.3 工具使用 . . . 40
5.3 公开基准 . . . 40
5.4 比较有用的数据集 . . . 40

6 RLHF & instructGPT **40**
6.1 sft . . . 41
6.2 rm . . . 41
6.3 rl . . . 42
6.3.1 rl 流程概述 . . . 42
6.3.2 几个重要的 loss . . . 45
6.3.2.1 actor & actor loss . . . 45
6.3.2.2 critic & critic loss . . . 45
6.3.2.3 KL Penalty . . . 46
6.3.2.4 GAE . . . 46
6.3.2.5 entropy loss . . . 47
6.3.2.6 Policy kl . . . 47
6.3.3 两个采样 . . . 48
6.3.3.1 Old Policy Sampling（无 bp） . . . 48
6.3.3.2 New Policy Sampling（有 bp） . . . 48
6.3.4 开源 rlhf 库 . . . 48
6.3.4.1 openai 的 lm-human-preferences(gpt2 的 finetune) . . . 48
6.3.4.2 huggingface 的 TRL . . . 48
6.3.4.3 CarperAI 的 trlx . . . 48
6.3.4.4 allenai 的 RL4LMs . . . 48

7 llama 系列 **48**
7.1 llama . . . 48
7.1.1 预训练数据 . . . 49
7.1.2 网络结构 . . . 50
7.1.3 训练加速 . . . 50
7.1.4 衍生：Alpaca . . . 50
7.2 llama2 . . . 51

8 gemini 系列 **51**
8.1 Gemini 1.0 . . . 51
8.2 Gemini 1.5 . . . 51
8.3 gemma . . . 51

8.3.1 更大的上下文窗口 . . . 51
8.3.2 架构 & 训练方法 . . . 51

9 多智能体 **51**

10 一些其他比较重要的工作 **52**
10.1 几篇出现频率比较高的论文 . . . 52
10.2 Anthropic 的一些工作 . . . 52
10.3 ChatGLM . . . 52

11 训练 & 预测架构 **52**
11.1 pathways . . . 52
11.1.1 Google 的大规模稀疏模型设计 . . . 52
11.2 megatron-lm . . . 52
11.3 deepspeed . . . 52
11.4 ray-llm . . . 52
11.5 medusa-llm . . . 52

12 大模型的一些现象 **53**
12.1 重复生成 . . . 53

13 多模态大模型 **53**
13.1 ViT&Swin-Transformer . . . 53
13.2 stable diffusion . . . 53
13.3 DALL-E 系列 . . . 54
13.4 PALM-E . . . 54
13.5 Pika/Runway 等 . . . 55
13.6 sora . . . 55
13.6.1 现有方法 . . . 55
13.6.2 将视频转成 spacetime latent patches . . . 55
13.6.2.1 Vivit . . . 55
13.6.2.2 latent 空间上的 patch . . . 56
13.6.3 Diffusion Transformer . . . 57
13.6.4 语言理解 . . . 57
13.6.5 使用图像/视频作为 prompt . . . 57
13.6.6 生成图像 . . . 58
13.6.7 涌现的模拟能力 . . . 58
13.6.8 存在的问题 . . . 58

14 LLM 与推荐结合 **58**
14.1 综述 . . . 58
14.2 Recommender Systems with Generative Retrieval . . . 58
14.3 P5 . . . 58
14.4 llm vs ID . . . 58

15 其他 **58**
15.1 一些比较好的模型 . . . 58
15.1.1 文本匹配 . . . 59
15.2 RETRO Transformer . . . 59
15.3 WebGPT . . . 59
15.4 本地知识库 . . . 59
15.5 llm 应用合辑 . . . 59
15.6 nanogpt . . . 59
15.7 达摩院大模型技术交流 . . . 60

下载本文 pdf：https://github.com/daiwk/collections/blob/master/pdfs/llm_aigc.pdf

各种学习相关代码

https://github.com/daiwk/llms_new

1 概述

1.1 LLM 简史

PLM（pretrained language models）

- 2017 年的Learning to generate reviews and discovering sentiment尝试用 rnn 来实现智能系统
- 2018 年的 gpt1：Improving language understanding by generative pre-training，生成式预训练（Generative pre-training, gpt），用 transformer 的 decoder，参数量 117m（0.1b），无监督预训练和有监督微调。确定对自然语言文本建模的基本原则为**预测下一个单词**。
- 2019 年的 gpt2：Language models are unsupervised multitask learners模型结构小改，增加数据，参数量变大为 15 亿（1.5b），无监督语言建模，**无需使用标记数据进行显式微调**。
  - 参考The natural language decathlon: Multitask learning as question answering中**多任务求解的概率形式**：$p(output|input, task)$ 。
  - 提出“由于特定任务的有监督目标与无监督目标（语言建模）相同，只是在序列的子集上进行评估，因此，无监督目标的全局最小值也是有监督目标的全局最小值”，即每个 NLP 任务可以看作**世界文本子集的单词预测问题**，如果模型有足够能力来复原世界文本，无监督语言建模可以解决各种问题。
  - 仅无监督与监督微调的 SOTA 相比效果还是不太行。虽然 GPT2 模型规模相对较小，但如对话等任务在其基础上做微调还是能拿到很好的效果的，例如DIALOGPT : Large-scale generative pre-training for conversational response generation、End-to-end neural pipeline for goal-oriented dialogue systems using GPT-2
- 2020 年的 gpt3：Language models are few-shot learners，175b（1750 亿）参数，当参数量到达千亿时出现了『涌现』现象，发现可以 in-context learning（这点在 **3.3 亿的 BERT 和 15 亿的 gpt2 中看不到**）。**预训练和 ICL 有相同的语言建模范式**：预训练预测给定上下文条件下的后续文本序列，ICL 预测正确的任务解决方案，其可被格式化为给定任务描述和示范下的文本序列。
- GPT-3 的两种改进方法：
  - 使用代码数据训练：GPT-3 主要问题是缺乏对复杂任务的推理能力，2021 年 openai 提出了 Codex（Evaluating Large Language Models Trained on Code），在 github 代码上微调的 GPT。A neural network solves and generates mathematics problems by program synthesis: Calculus, differential equations, linear algebra, and more发现 Codex 能解决非常困难的编程问题，还能在数学问题上有显著提升。Text and code embeddings by contrastive pre-training提出了训练文本和代码 emb 的对比学习，在线性探测分类、文本搜索、代码搜索等任务上有所提升。GPT-3.5 就是在基于代码的 GPT（code-davinci-002）的基础上开发的。
  - 与人类对齐：2017 年 openai 就在learning from human preference的博客中提出了应用强化学习来学习由人类标的偏好比较，此后 2021 年 7 月 openai 发表了 PPO。2020 年 GPT-2 用 RL 进行微调，Deep reinforcement learning from human preferences，Learning to summarize from human feedback也做了相似工作。2022 年提出了 RLHF 的 InstructGPT(Training language models to follow instructions with human feedback)，其中的 **SFT 就对应于常说的指令微调**。在 openai 的博客Our approach to alignment research中提出了训练 AI 系统的 3 个有前途的方向：**使用人类反馈、协助人类评估、做对齐研究**。
- 2022 年的 ChatGPT：用类似 InstructGPT 的方式进行训练，专门**对对话能力进行优化**，将人类生成的对话（**扮演用户和 AI 两个角色**）与 InstructGPT 数据集结合起来**以对话形式生成**。
- 2023 年的 GPT-4：将文本输入扩展到**多模态信号**。此外，
  - 提升安全性：在 RLHF 训练中加入**额外的安全奖励信号**，采用多种干预策略如 Anthropic 提出的Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned提到的红队评估（read teaming）机制以减轻幻觉、隐私和过度依赖问题。
  - 改进的优化方法：使用**可预测扩展**（predictable scaling）的机制，使用模型训练期间的一小部分计算量**以预测最终性能**。
  - 迭代部署的工程方案：Lessons learned on language model safety and misuse，遵循 5 阶段的开发和部署生命周期来开发模型和产品。

1.2 LLM 列表（持续更新中）

- 百亿：除了 LLaMA（最大 650 亿）和 NLLB（最大 545 亿），大多数在 100 亿-200 亿之间，通常需要**数百甚至上千**个 GPU 或 TPU。

- 千亿：OPT、OPT-IML、BLOOM 和 BLOOMZ 与 GPT-3(175B) 大致相同，GLM 有 1300 亿，Galactica 有 1200 亿，通常需要**数千**个 GPU 或者 TPU。

ckpt?	模型	发布时间	大小	预训练数据规模	硬件	训练时间
Y	T5	2019.10	11B	1 万亿 tokens	1024 TPU v3	-
N	GPT-3	2020.05	175B	3000 万亿 tokens	-	-
N	GShard	2020.06	600B	1 万亿 tokens	2048 TPU v3	4 天
Y	mT5	2020.10	13B	1 万亿 tokens	-	-
Y	PanGu-α	2021.04	13B	1.1TB	2048 Ascend 910	-
Y	CPM-2	2021.06	198B	2.6TB	-	-
N	Codex	2021.07	12B	1000 万亿 tokens	-	-
N	ERNIE 3.0	2021.07	10B	3750 亿 tokens	384 v100	-
N	Jurassic-1	2021.08	178B	3000 亿 tokens	800 GPU	-
N	HyperCLOVA	2021.09	82B	3000 亿 tokens	1024 A100	13.4 天
N	FLAN	2021.09	137B	-	128 TPU v3	60 小时
N	Yuan 1.0	2021.10	245B	1800 亿 tokens	2128 GPU	-
Y	T0	2021.10	11B	-	512 TPU v3	27 小时
N	Anthropic	2021.12	52B	4000 亿 tokens	-	-
N	WebGPT	2021.12	175B	-	-	-
N	Gopher	2021.12	280B	3000 亿 tokens	4096 TPU v3	920 小时
N	ERNIE 3.0 Titan	2021.12	260B	-	-	-
N	GLaM	2021.12	1200B	2800 亿 tokens	1024 TPU v4	574 小时
N	LaMDA	2022.01	137B	7680 亿 tokens	1024 TPU v3	57.5 天
N	MT-NLG	2022.01	530B	2700 亿 tokens	4480 80G A100	-
N	AlphaCode	2022.02	41B	9670 亿 tokens	-	-
N	InstructGPT	2022.03	175B	-	-	-
N	Chinchilla	2022.03	70B	1.4 万亿 tokens	-	-
Y	CodeGen	2022.03	16B	5770 亿 tokens	-	-
Y	GPT-NeoX-20B	2022.04	20B	825GB	96 40G A100	-
Y	Tk-Instruct	2022.04	11B	-	256 TPU v3	4 小时
N	PaLM	2022.04	540B	7800 亿 tokens	6144 TPU v4	-
Y	UL2	2022.05	20B	825GB	96 40G A100	-
Y	OPT	2022.05	175B	1800 亿 tokens	992 80G A100	-
Y	NLLB	2022.07	54.5B	-	-	-

ckpt?	模型	发布时间	大小	预训练数据规模	硬件	训练时间
N	AlexaTM	2022.08	20B	1.3 万亿 tokens	128 A100	120 天
N	Sparrow	2022.09	70B	64 TPU v3	-	-
N	WeLM	2022.09	10B	3000 亿 tokens	128 A100 40G	24 天
N	U-PaLM	2022.10	540B	-	512 TPU v4	5 天
N	Flan-PaLM	2022.10	540B	-	512 TPU v4	37 小时
N	Flan-U-PaLM	2022.10	540B	-	-	-
Y	GLM	2022.10	130B	4000 亿 tokens	768 40G A100	60 天
Y	Flan-T5	2022.10	11B	-	-	-
Y	BLOOM	2022.11	176B	3660 亿 tokens	384 80G A100	105 天
Y	mT0	2022.11	13B	-	-	-
Y	Galactica	2022.11	120B	1060 亿 tokens	-	-
Y	BLOOMZ	2022.11	176B	-	-	-
Y	OPT-IML	2022.12	175B	-	128 40G A100	-
Y	LLaMA	2023.02	65B	1.4 万亿 tokens	2048 80G A100	21 天
N	GPT-4	2023.03	-	-	-	-
Y	CodeGeeX	2022.09	13B	8500 亿 tokens	1536 Ascend 910	60 天
N	PanGU-Σ	2023.03	1085B	3290 亿 tokens	512 Ascend 910	100 天
Y	Pythia	2023.04	12B	3000 亿 tokens	256 40G A100	-

可以直接把对应的 md 丢给 gpt，叫它导出一个 excel，然后就可以自定义排序或者画散点图看了

1.3 LLM 数据集

llm 中文数据集：https://juejin.cn/post/7238921093553438779

- Books：
  - BookCorpus：超过 11000 本电子书，用于 GPT 和 GPT-2。
  - Gutenberg：超过 70000 本文学作品，包括小说、散文、诗歌、戏剧、历史、科学、哲学和其他公共领域，用于 MT-NLG 和 LLaMA。
  - Books1 和 Books2：比 BookCorpus 大得多，但未公开，用于 GPT-3。
- CommonCrawl：最大的开源网络爬虫数据库之一，**百万亿字节**，有大量噪音和低质信息，需要过滤，有如下 4 个子集：
  - C4：包括 en（806G，训练 T5、LaMDA、Gopher、UL2）、en.noclean（6T）、realnewslike（36G）、webtextlike（17G）、multilingual（38T，训练 mT5）。
  - CC-Stories：31G，内容以故事的形式展示
  - CC-News：76G
  - RealNews：120G
- Reddit Links：Reddit 上的帖子，高赞通常比较有用，可以拿来创建高质量数据集。
  - WebText：由 Reddit 上的高赞链接组成，未公开，对应的开源版是OpenWebText。
  - Pushshift.io：实时更新的数据集，包括 Reddit 自创建以来的历史数据，有数据存储，也有实用工具，供用户搜索、总结和统计分析。
- Wikipedia：大部分文章使用写作风格，并支持引用，英语版本用于大多数 LLM，如 GPT-3、LaMDA、LLaMA，还有多语言版。

- Code：包括开源许可证的公共代码库（如 github）和与代码相关的问答平台（如 StackOverflow）,Google 公开了BigQuery数据集，CodeGen 用的 BIGQUERY 是其的一个子集。
- 其他：
  - The Pile有 800G，包括书籍、网站、代码、科学论文和社交媒体平台，有 22 个子集，用于 GPT-J(6B)、CodeGen(16B)、Megatron-Turing NLG（530B）。
  - ROOTS由各种小数据集组成，共 1.6T，包括 59 种语言（自然语言和编程语言），用于 BLOOM。

1.4 LLM 开源库

- transformers：huggingface 的库
- deepspeed：微软的库，与 pytorch 兼容，训练了 MT-NLG、BLOOM 等模型，包括各种分布式训练优化技术，如**内存优化（ZeRO、梯度检查点**等）和**管道并行**。
- megatron-lm：英伟达的库，同样包括各种分布式训练技术，包括**模型和数据并行**、**混合精度**训练和 **FlashAttention**。(Megatron-lm: Training multi-billion parameter language models using model parallelism、Efficient large-scale language model training on GPU clusters using megatron-lm和Reducing activation recomputation in large transformer models)
- jax：google 的库，允许用户在**带有硬件加速（GPU 或 TPU）**的情况下进行**数组的高效运算**，可以在**各种设备**高效计算，支持**自动微分**和**即时编译**等功能。
- colossal-AI：HPC-AI Tech 的库，基于 pytorch，可以使用PatrickStar提出的方法优化异构内存管理，分布了基于 LLaMA 的ColossalChat
- BMTrain：openBMB 的库，强调代码简洁、低资源占用和高可用性
- FastMoE：专门用于 MoE 模型的训练库，基于 pytorch，简化了将 transformer 转换为 MoE 模型的过程
- semantic-kernel：微软的开源库

1.5 一些综述

- Foundation Models for Natural Language Processing – Pre-trained Language Models Integrating Media
- 大规模语言模型：从理论到实践，Pre-trained Models for Natural Language Processing: A Survey邱锡鹏等
- 人大的大模型综述：https://github.com/RUCAIBox/LLMSurvey，自己存了一份 pdf，(!!! **本章大部分内容按这个来组织!!!**)
- Talking about large language models
- Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing，引用数 2k+
- A comprehensive survey on pretrained foundation models: A history from BERT to chatgpt，唐杰等
- Pre-Trained Models: Past, Present and Future
- A Comprehensive Survey of AI-Generated Content (AIGC): A History of Generative AI from GAN to ChatGPT
- Pretrained Language Models for Text Generation: A Survey
- A survey for in-context learning
- Towards reasoning in large language models: A survey
- Reasoning with language model prompting: A survey

- Dense Text Retrieval based on Pretrained Language Models: A Survey
- Fine-tune 之后的 NLP 新范式：Prompt 越来越火，CMU 华人博士后出了篇综述文章
- 如何高效部署大模型？CMU 最新万字综述纵览 LLM 推理 MLSys 优化技术：Towards Efficient Generative Large Language Model Serving: A Survey from Algorithms to Systems

1.6 扩展法则

1.6.1 openai 的扩展法则

2020 年,openai 的Scaling laws for neural language models通过拟合模型在不同数据大小（2000w 到 230 亿个 token）、不同的模型大小（7.68 亿到 15 亿个**非嵌入参数**）的性能，提出了在**计算预算**c 的条件下，L 是用 nats 表示的交叉熵损失，模型性能与**模型规模**N、**数据集规模**D 以及**训练计算量**C 间存在如下幂律关系：

$$L(N) = (\frac{N_c}{N})^{\alpha_N}, \alpha_N \sim 0.076, N_c \sim 8.8 \times 10^{13}$$

$$L(D) = (\frac{D_c}{D})^{\alpha_D}, \alpha_D \sim 0.05, N_c \sim 5.4 \times 10^{13}$$

$$L(C) = (\frac{C_c}{C})^{\alpha_C}, \alpha_C \sim 0.05, C_c \sim 3.1 \times 10^{8}$$

其中，N_c 表示非嵌入参数数量，D_c 表示训练 token 数量,C_c 表示 FP-days。

1.6.2 Chinchilla 扩展法则

DeepMind 在Training compute-optimal large language models中提出了 Chichilla 扩展法则来指导 LLM 最优计算量的训练。通过变化更大范围的模型大小（7000w 到 160 亿参数）和数据大小（50 亿到 5000 亿个 token）进行实验，拟合了如下的扩散法则：

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}$$

其中 $E = 1.69, A = 406.4, B = 410.7, \alpha = 0.34, \beta = 0.28$，通过在约束条件 $C \approx 6ND$ 下优化损失 $L(N, D)$，将计算预算最优地分配给模型大小和数据大小的方法：

$$N_{opt}(C) = G\left(\frac{C}{6}\right)^{a}, \quad D_{opt}(C) = G^{-1}\left(\frac{C}{6}\right)^{b}$$

其中 $a = \frac{\alpha}{\alpha+\beta}$，$b = \frac{\beta}{\alpha+\beta}$，$G$ 是由 A, B, α, β 计算出的扩展系数。

随着计算预算的增加，

- openai 的扩展法则更偏向于将更大预算分给**模型大小**，因为其对比各模型时使用了固定的训练数据量和学习率等超参，低估了数据量的作用。
- Chinchilla 扩展法则认为**模型大小和数据大小要同比例增加**，即 a 和 b 取值差不多。因为其在无视模型大小的前提下，发现设置与数据量差不多 match 的学习率能获得更好的 loss。

然而，有一些能力（如涌现）无法根据扩展法则进行预测，只有当模型达到一定规模时才会出现。

1.7 涌现能力

涌现能力：在小型模型中不存在而在大型模型中产生的能力，当规模达到一定程度时，性能显著提升，超出随机水平（参考 Emergent Abilities of Large Language Models）。与物理学中的**相变**现象类似（物质从一种相（状态）转变为另一种相的过程，通常伴随着能量的吸收或释放，并且涉及不同的物理性质，例如固体、液体和气体之间的转变）。

普林斯顿 DeepMind 用数学证明：LLM 不是随机鹦鹉！「规模越大能力越强」有理论根据

A Theory for Emergence of Complex Skills in Language Models:

Figure 2: Eight examples of emergence in the few-shot prompting setting. Each point is a separate model. The ability to perform a task via few-shot prompting is emergent when a language model achieves random performance until a certain scale, after which performance significantly increases to well-above random. Note that models that used more training compute also typically have more parameters—hence, we show an analogous figure with number of model parameters instead of training FLOPs as the x-axis in Figure 11. A–D: BIG-Bench (2022), 2-shot. E: Lin et al. (2021) and Rae et al. (2021). F: Patel & Pavlick (2022). G: Hendrycks et al. (2021a), Rae et al. (2021), and Hoffmann et al. (2022). H: Brown et al. (2020), Hoffmann et al. (2022), and Chowdhery et al. (2022) on the WiC benchmark (Pilehvar & Camacho-Collados, 2019).

LLM 的 3 种典型涌现能力及其对应代表模型:

1.7.1 上下文学习

GPT-3（Language models are few-shot learners）提出，只要提供一个自然语言指令和/或几个任务演示，语言模型就能通过完成输入文本的词序列的方式来为测试实例生成预期输出，不用额外的梯度更新。

- ICL 能力小模型不具备：1750 亿的 GPT-3 有 ICL 能力，但 GPT-1 和 GPT-2 无此能力。
- ICL 能力取决于具体下游任务：130 亿的 GPT-3 能在算术任务上有 ICL，但 1750 亿的 GPT-3 在波斯语 QA 上无能为力。

1.7.2 指令遵循

使用**自然语言描述的混合多任务数据集进行微调（指令微调）**，LLM 在**未见过的以指令形式描述的任务**上表现出色，具有更好的泛化能力。例如Multitask prompted training enables zero-shot task generalization、Training language models to follow instructions with human feedback、Finetuned language models are zero-shot learners。

在Finetuned language models are zero-shot learners的实验中，当模型大小达到 680 亿时，经过指定微调的 LaMDA-PT 开始在未见过的任务上显著优于未微调的模型，而 80 亿或更小的模型则没有这个现象。

在Scaling instruction-finetuned language models的实验中，PaLM 至少在 620 亿参数上才能在 4 个评估基准的各种任务上表现良好。

1.7.3 逐步推理

对于涉及多个推理步骤的复杂任务（如数学），可以使用**思维链（Chain-of-Thought, CoT）**提示策略（Chain of thought prompting elicits reasoning in large language models），让 LLM 通过**利用中间推理步骤的提示机制**来解决这类任务。

Chain of thought prompting elicits reasoning in large language models发现，CoT 在模型大于 600 亿的 PaLM 和 LaMBDA 变体中能够提升在算术推理基准任务的效果，而当模型大于 1000 亿时，相比标准提示的优势更明显。

How does GPT Obtain its Ability? Tracing Emergent Abilities of Language Models to their Sources

1.8 LLM 关键点

如何让 LLM 能够**通用**且**有能力**?

1.8.1 扩展

更大的模型、数据规模和更多的训练计算，但计算预算是有限的，可以用扩展法更高效地分配计算资源，如 Chinchilla 在**相同计算预算下增加训练 token 数**，优于更大模型规模的 Gopher，同时需要数据清理。

1.8.2 训练

- 分布式的训练框架：包括 DeepSpeed（Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters）和 Megatron-LM（Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism和Efficient large-scale language model training on GPU clusters using megatron-lm）
- 优化技巧：有助于提升训练稳定性和模型性能，如**重新开始以克服训练损失激增**（Palm: Scaling language modeling with pathways）和**混合精度训练**（BLOOM: A 176b-parameter open-access multilingual language model）。

1.8.3 能力引导

当 LLM 执行某些特定任务时，可能不会显式地展示出其通用求解器的能力，**设计合适的任务指令或具体的 ICL 策略**可以**激发**这种能力，例如

- 通过**包含中间推理步骤的 CoT 提示**
- 使用**自然语言表达的任务描述**，对 LLM 进行**指令微调**

1.8.4 对齐微调

由于预训练语料库包括高质量和低质量的数据，LLM 可能生成有毒、偏见甚至有害的内容，要让 LLM 和人类价值观保持一致，如**有用性、诚实性和无害性**。RLHF 相关工作如Training language models to follow instructions with human feedback和Deep reinforcement learning from human preferences能够产生高质量、无害的回答（例如拒绝回答侮辱性问题）。

1.8.5 工具操作

LLM 本质是基于海量文本语料库进行文本生成训练的，对于不适合以文本形式表达的任务表现不佳（如数字计算），且其能力受限于预训练数据，无法获取最新信息。可以利用外部工具：

- Toolformer: Language models can teach themselves to use tools能利用计算器进行准确计算
- Webgpt: Browser-assisted question-answering with human feed- back能利用搜索引擎检索未知信息

2 预训练

2.1 数据收集

2.1.1 数据获取

- 通用文本数据：
  - 网页：例如 CommonCrawl，同时需要过滤和处理以提高质量
  - 对话文本：公共对话数据如 PushShift.io，对于在线社交媒体的对话数据，可以**转换成树形结构**，每句话与回应其的话相连。多方的对话树可以划分为预训练语料库中的多个子对话。过度引入对话数据可能会有潜在风险（OPT: open pre-trained transformer language models）：陈述性指令和直接疑问句被错误地认为是对话的开始，导致指令的有效性下降。
  - 书籍：更正式的长文本，利于**学习语言知识**、**建模长期依赖关系**、**生成叙述性和连贯的文本**。
- 专用文本数据：
  - 多语言文本：BLOOM 的预训练语料中包括了 46 种语言，PaLM 包含了 122 种
  - 科学文本：如 arxiv 论文、科学教材、数学网页等，通常需要特定的标记化和预处理。
  - 代码：一是编程问答社区，二是开源代码仅为。对应长距离依赖和准确的执行逻辑，可能是复杂推理能力的来源。将推理任务格式化为代码形式还能帮 LLM 生成更准确的结果（如Language models of code are few-shot commonsense learners和Autoformalization with large language models）

2.1.2 数据预处理

- 质量过滤：有一些基于分类器的方法，例如维基百科的数据为正样本，负采样其他数据训练二分类器，但这种方法会删除方言、口语和社会语言的高质量文本，可能导致有偏、减少多样性。还有启发式的方法，主要包括：
  - 基于语言的过滤：如果该 llm 主要用于某种语言，可以把其他语言删了
  - 基于度量的过滤：利用生成文本的评估度量（如 **perplexity**）来检测和删除不自然的句子
  - 基于统计的过滤：如**标点符号分布**、**符号和单词比例**、**句子长度**等
  - 基于关键词的过滤：删除噪声或无用元素，如 **HTML 标签**、**超链接**、**模板**、**攻击性词语**等。

- 去重：Scaling laws and interpretability of learning from repeated data中发现重复数据会降低多样性，可能导致训练不稳定。下面 3 个级的去重都很有用
  - 句子级：删掉包含重复单词和短语的句子，因为可能在语言建模中引入**重复模式**（The curious case of neural text degeneration）(后面的章节会讲)
  - 文档级：通过文档间的表层特征（如 n-gram 或单词重合率）来删掉重复文档
  - 数据集级：训练集中删掉测试集可能出现的重复文本，防止训练集和评估集间的重叠
- 隐私去除：删掉可识别个人信息（PII），如基于关键词（姓名、地址、电话号码）识别。另外，Deduplicating Training Data Mitigates Privacy Risks in Language Models发现 LLM 在隐私攻击下的脆弱性可能归因于预训练语料中存在**重复 PII 数据**。
- 分词：可以直接利用已有分词器，也可以使用专门为预训练语料库设计的分词器，如 SentencePiece，而且 **BPE**(byte pair encoding) 能**确保分词后的信息不会丢失**，但其中的如 NFKC(Unicode normalization forms) 的**归一化技术**可能会**降低分词的性能**。

2.1.3 预训练语料的重要性

- 混合来源：不同领域和场景的数据能让 LLM 有更强大的泛化能力。需要**仔细设置数据分布**，Gopher 对数据分布消融，发现增加书籍数据可以提升捕捉长期依赖的能力，增加 c4 数据集比例可以提升其在 c4 验证集上的效果，但单独训练过多的某个领域数据会影响 LLM 在其他领域的泛化能力。
- 数据量：模型性能方面，**数据大小**也能看到与模型大小类似的**扩展法则**。LLaMA 发现，用更多数据训练更长时间，较小的模型也能实现良好性能。
- 数据质量：Gopher、GLaM 和 T5 都发现，在清理后的数据上训练能提升 llm 效果。数据的重复可能导致『双下降现象』（Scaling laws and interpretability of learning from repeated data和Deep double descent: Where bigger models and more data hurt），甚至会导致训练不稳定。此外，Scaling laws and interpretability of learning from repeated data还发现，**重复数据会降低 LLM 从上下文复制的能力**，从而影响 **ICL 中的泛化能力**。

注：**双下降**指的是随着模型复杂性的增加，可能 **loss 先下降，然后再升高，最后又下降**：＋当模型的复杂性低于数据的复杂性时，增加模型的复杂性可以帮助减少训练误差。＋当模型的复杂性超过数据的复杂性时，增加模型的复杂性反而可能导致训练误差增加。这是因为模型开始过拟合数据，捕获数据中的噪声而非实际的模式。＋当模型的复杂性远大于数据的复杂性时，训练误差可能再次开始减少。这是因为模型有足够的能力来对数据的噪声进行平滑，同时仍然能够捕获数据的实际模式。

2.2 架构

2.2.1 主流框架

- **编码器-解码器架构 (encoder-decoder)**：标准 Transformer，如 T5、BART，**只有少数 LLLM 还用这种结构**，如 Flan-T5
- **因果解码器架构 (causual decoder)**：也叫 **decoder-only**，**单向注意力掩码**，输入和输出 token 通过解码器以相同方式进行处理，以 GPT 系列为代表，现有大部分 LLM 都是这种架构，如 OPT、BLOOM、Gopher 等。
- **前缀解码器架构 (prefix decoder)**：修正因果解码器的掩码机制，使其能**对前缀 token 执行双向注意力**，并且**仅对生成的 token 执行单向注意力**（和 encoder-decoder 类似），即Unified language model pre-training for natural language understanding and generation提出的 uni-lm。What language model architecture and pretraining objective works best for zero-shot generalization?建议不从头开始预训练，而是**继续训练因果编码器，然后将其转换成前缀编码器以加速收敛**。例如 U-PaLM 从 PaLM 演化而来，还有 GLM-130B 也是这种架构。

Figure 1: Overview of unified LM pre-training. The model parameters are shared across the LM objectives (i.e., bidirectional LM, unidirectional LM, and sequence-to-sequence LM). We use different self-attention masks to control the access to context for each word token. The right-to-left LM is similar to the left-to-right one, which is omitted in the figure for brevity.

对于这 3 种架构，都可以用 **MoE** 进行扩展，每个输入的**一小部分神经网络权重**被**稀疏激活**，如Switch Transformer和 GLaM。Unified scaling laws for routed language models发现，通过**增加专家数量或总参数大小**，性能会有显著改进。

2.2.1.1 讨论：为什么现在的 LLM 都是 Decoder only 的架构?

https://www.zhihu.com/question/588325646/answer/2940298964

- **泛化性能强**：ICML 22 的What language model architecture and pretraining objective works best for zero-shot generalization. 在最大 5B 参数量、170B token 数据量的规模下做了一些列实验，发现用 next token prediction 预训练的 decoder-only 模型在**各种下游任务上 zero-shot 泛化性能最好**；另外，ACL23 的Why Can GPT Learn In-Context? Language Models Implicitly Perform Gradient Descent as Meta-Optimizers等工作表明，decoder-only 模型相当于基于给出的几个示例**隐式地进行梯度下降**，对应的 in-context learning 泛化能力更强，
- **秩**的讨论：Attention is not all you need: pure attention loses rank doubly exponentially with depth的讨论，$n \times d$ 和 $d \times n$ 相乘后（$n \gg d$）再加上 softmax 后，秩不超过 d，而 decoder-only 中有一个下三角矩阵的 mask，所以输入的是一个下三角矩阵，而下三角矩阵的行列式是对角线之积，且有 softmax，对角线肯定大于 0，所以是满秩的 (行列式不为 0–> 矩阵经过变换后不会有一行或者一列全为 0–> 当前矩阵满秩)
- 预训练**任务难度**更大：相比 encoder-decoder，decoder-only 架构里**每个位置能接触到的信息更少**，故难度更高，当模型大小和数据量够的时候，上限更高
- 隐式学习了**位置信息**：Transformer Language Models without Positional Encodings Still Learn Positional Information，encoder 里对语序的区分能力较弱，需要结合 position encoding，而 causual attention 隐式地具备了这种建模位置的能力。
- **工程效率**：支持**复用 kv-cache**，对多轮对话更友好，『**DIN 的 FLOPS**』一节里有讲

2.2.2 组件配置

2.2.2.1 标准化（norm）

LN(layer norm) 能缓解 LLM 训练不稳定的问题，其位置很重要。

Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

- 前置 LN：最初 Transformer 使用后置 LN，但大多数 LLM 采用前置 LN 以实现更稳定的训练，尽管会有一些性能损失 (On layer normalization in the transformer architecture)。Sandwich-LN在残差连接前添加额外的 LN，虽然能避免数值爆炸，但有时会无法稳定 LLM 的训练，可能导致训练崩溃（GLM-130B: an open bilingual pre-trained model）
- RMS Norm：训练和性能都不错，在 Gopher 和 Chinchilla 里使用
- Deep Norm：比 LN 有更好的训练稳定性，和后标准化一起用在 GLM-130B 里

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

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

Architectures	Encoder		Decoder	
	α	β	α	β
Encoder-only (e.g., BERT)	$(2N)^{\frac{1}{4}}$	$(8N)^{-\frac{1}{4}}$	-	-
Decoder-only (e.g., GPT)	-	-	$(2M)^{\frac{1}{4}}$	$(8M)^{-\frac{1}{4}}$
Encoder-decoder (e.g., NMT, T5)	$0.81(N^4M)^{\frac{1}{16}}$	$0.87(N^4M)^{-\frac{1}{16}}$	$(3M)^{\frac{1}{4}}$	$(12M)^{-\frac{1}{4}}$

Figure 2: (a) Pseudocode for DEEPNORM. We take Xavier initialization (Glorot and Bengio, 2010) as an example, and it can be replaced with other standard initialization. Notice that α is a constant. (b) Parameters of DEEPNORM for different architectures (N-layer encoder, M-layer decoder).

此外，**在 emb 后直接加额外的 LN** 能提升训练稳定性，但会导致**显著的性能下降** (What language model to train if you have one million GPU hours?)，在后来的 LLM 中**被移除**（BLOOM: A 176b-parameter open-access multilingual language model）。

2.2.2.2 激活函数

FFN 中的激活函数：

- GeLU：大部分都是这个
- GLU(gated linear units) 的变体：应用在 PaLM 和 LaMDA 等模型中，如 SwiGLU 和 GeGLU 有更好的效果，但在 FFN 中的参数量比 GeLU 要大 50%

原始 Transformer 中

$$\mathrm{FFN}\left(x, W_1, W_2, b_1, b_2\right) = \max\left(0, xW_1 + b_1\right) W_2 + b_2$$

T5 中把 bias 干掉了

$$\mathrm{FFN}_{\mathrm{ReLU}}\left(x, W_1, W_2\right) = \max\left(xW_1, 0\right) W_2$$

然后，$\mathrm{GELU}(x) = x\Phi(x)$，同时 $\mathrm{Swish}_\beta(x) = x\sigma(\beta x)$，接下来

$$
\begin{aligned}
\mathrm{GLU}(x, W, V, b, c) &= \sigma(xW + b) \otimes (xV + c) \\
\mathrm{Bilinear}(x, W, V, b, c) &= (xW + b) \otimes (xV + c) \\
\mathrm{ReGLU}(x, W, V, b, c) &= \max(0, xW + b) \otimes (xV + c) \\
\mathrm{GEGLU}(x, W, V, b, c) &= \mathrm{GELU}(xW + b) \otimes (xV + c) \\
\mathrm{SwiGLU}(x, W, V, b, c, \beta) &= \mathrm{Swish}_\beta(xW + b) \otimes (xV + c)
\end{aligned}
$$

对应起来就是

$$
\begin{aligned}
\mathrm{FFN}_{\mathrm{GLU}}(x, W, V, W_2) &= (\sigma(xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{Bilinear}}(x, W, V, W_2) &= (xW \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{ReGLU}}(x, W, V, W_2) &= (\max(0, xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{GEGLU}}(x, W, V, W_2) &= (\mathrm{GELU}(xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) &= (\mathrm{Swish}_1(xW) \otimes xV)W_2
\end{aligned}
$$

2.2.2.3 位置编码

Transformer 的 self-attention 有转换不变性，故要位置编码以引入绝对或相对位置信息来建模序列。

- 绝对位置编码：
  - 正弦函数：原始 Transformer 中使用
  - 可学习的位置编码：LLM 中常用
- 相对位置编码：Exploring the limits of transfer learning with a unified text-to-text transformer提出，根据 **k 和 q 之间的偏移量**生成 emb
- Alibi：Train short, test long: Attention with linear biases enables input length extrapolation提出，使用 **k 和 q 之间距离的惩罚**来给注意力分数加 bias，What language model architecture and pretraining objective works best for zero-shot generalization发现其有更好的**零样本泛化能力**和更强的**外推能力**，能够在**比训练序列更长的序列**上表现良好。
- RoPE：Roformer: Enhanced transformer with rotary position embedding提出，**k 和 q 之间的分数用相对位置信息计算**，利于建模长序列，在 PaLM、LLaMA、GLM-130B 中都有应用。

2.2.2.4 注意力机制和 Bias

- 稀疏注意力：Generating long sequences with sparse transformers)，**计算复杂度更低**，GPT-3 用了
- FlashAttention：Flashattention: Fast and memory-efficient exact attention with IO-awareness，考虑显存访问
- 其他 attention：如Random feature attention、Big bird: Transformers for longer sequences
- 移除 bias：PaLM 和 Galactica 中将 bias 删了，能够增加训练稳定性。

2.2.2.5 小结

2.2.2.5.1 归一化位置

sublayer 表示 FFN 或 self-attention 模块

方法	公式
post Norm	$\mathrm{Norm}(\mathbf{x} + \mathrm{Sulayerb}(\mathbf{x}))$
pre Norm	$\mathbf{x} + \mathrm{Sublayer}(\mathrm{Norm}(\mathbf{x}))$
Sandwich Norm	$\mathbf{x} + \mathrm{Norm}(\mathrm{Sublayer}(\mathrm{Norm}(\mathbf{x})))$

2.2.2.5.2 归一化方法

方法	公式
Layer Norm	$\frac{\mathrm{x}-\mu}{\sqrt{\sigma}} \cdot \gamma+\beta, \quad \mu=\frac{1}{d} \sum_{i=1}^{d} x_i, \quad \sigma=\sqrt{\frac{1}{d} \sum_{i=1}^{d}(x_i-\mu)^2}$
RMSNorm	$\frac{\mathrm{x}}{\mathrm{RMS}(\mathrm{x})} \cdot \gamma, \quad \mathrm{RMS}(\mathbf{x})=\sqrt{\frac{1}{d} \sum_{i=1}^{d} x_i^2}$
Deep Norm	$LayerNorm(\alpha \cdot \mathbf{x}+\mathrm{Sublayer}(\mathbf{x}))$

2.2.2.5.3 激活函数

方法	公式
ReLU	$\mathrm{ReLU}(\mathbf{x})=\max(\mathbf{x}, \mathbf{0})$
GeLU	$\mathrm{GeLU}(\mathbf{x})=0.5\mathrm{x} \otimes[1+\mathrm{erf}(\mathbf{x}/\sqrt{2})], \quad \mathrm{erf}(x)=\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
Swish	$\mathrm{Swish}(\mathbf{x})=\mathbf{x} \otimes \mathrm{sigmoid}(\mathbf{x})$
SwiGLU	$\mathrm{SwiGLU}\left(\mathbf{x}_1, \mathbf{x}_2\right)=\mathrm{Swish}\left(\mathbf{x}_1\right) \otimes \mathbf{x}_2$
GeGLU	$\mathrm{GeGLU}\left(\mathbf{x}_1, \mathbf{x}_2\right)=\mathrm{GeLU}\left(\mathbf{x}_1\right) \otimes \mathbf{x}_2$

2.2.2.5.4 位置嵌入

- A_{ij}：**q 和 k 之间的注意力分数**
- r_{i-j}：基于 **q 和 k 之间偏移**的可学习标量
- $\mathbf{R}_{\theta, i-j}$：旋转角度为 $t \cdot \theta$ 的旋转矩阵

方法	公式
绝对位置编码	$\mathbf{x}_i=\mathbf{x}_i+\mathbf{p}_i$
相对位置编码	$A_{ij}=\mathbf{W}_q \mathbf{x}_i \mathbf{x}_j^T \mathbf{W}_k^T+r_{i-j}$
RoPE	$A_{ij}=\mathbf{W}_q \mathbf{x}_i \mathbf{R}_{\theta, i-j} \mathbf{x}_j^T \mathbf{W}_k^T$
Alibi	$A_{ij}=\mathbf{W}_q \mathbf{x}_i \mathbf{R}_{\theta, i-j} \mathbf{x}_j^T \mathbf{W}_k^T A_{ij}=\mathbf{W}_q \mathbf{x}_i \mathbf{x}_j^T \mathbf{W}_k^T-m(i-j)$

2.2.3 预训练任务

2.2.3.1 语言建模

语言建模是**仅解码器 LLM** 的常见目标，给定 token 序列 $\mathbf{x}=\{x_1, \ldots, x_n\}$，旨在基于序列中前面的 token，自回归地预估目标 token：

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

前缀解码器架构使用的是前缀语言建模任务，其 loss **不涉及对前缀内 token 的预测**，故预训练时**涉及的序列中 token 较少**，故当预训练 token 数相同时，前缀语言模型的**性能往往略低**于传统语言模型任务。

另外，自回归的 loss：

- 训练时：是可以并行的，因为每个位置的 label 是已知的，可以并行算，
- 预测时：是串行的，因为得预测完了第 t 个词，才能去预测第 t+1 个词。

2.2.3.2 去噪自编码

DAE 是 BERT 待模型的常见任务，即 MLM（masked language model），输入 $\mathbf{x}_{\setminus\tilde{\mathbf{x}}}$ 是一些**有随机替换区间的损坏文本**，目标是恢复被替换的 token $\tilde{\mathbf{x}}$：

$$\mathcal{L}_{DAE}(\mathbf{x}) = \log P\left(\tilde{\mathbf{x}} \mid \mathbf{x}_{\setminus\tilde{\mathbf{x}}}\right)$$

在 T5 和 GLM-130B 中使用，**自回归地恢复替换区间。**

2.3 Transformer 的 FLOPS 和访存带宽

https://zhuanlan.zhihu.com/p/624740065

A 的 shape 是 $m \times k$，B 的 shape 是 $k \times n$，那么矩阵乘法 AB 需要 $m \times k \times n$ 次的乘法，也需要同样多次的加法，所以 FLOPS 是 $2 \times m \times k \times n$

假设 batchsize 是 b，序列长度 s，原来的 emb 是 d，即输入的是 $[b, s, d]$，一般 $d = d_k = d_v = d_q$，对应的 Q、K、V 矩阵都是 $s \times d_v$，有 $head_num$ 个头，每个头的维度 $per_head_d = \frac{d}{head_num}$

2.3.1 attention 的 FLOPS

attention 的公式：

$$Q = xW_Q, K = xW_K, V = xW_V$$
$$x_{\text{out}} = \text{softmax}\left(\frac{QK^T}{\sqrt{h}}\right) \cdot V \cdot W_o + x$$

- 计算 3 个 Q、K、V：要算三次 $s \times d$ 和 $d \times d_v$ 的矩阵乘法，所以是：$3 \times 2 \times b \times s \times d \times d_v$
  - 输入：$[b, s, d]$ 和 3 个 $[b, d, d_v]$
  - 输出：$[b, s, d_v]$，再把最后一维 d_v 拆成 $head_num$ 份，再把中间两维互换一下，得到 $[b, head_num, s, per_head_d]$
- 计算 Q 和 K 的相似度：要算一次 $s \times d_v$ 和 $d_v \times s$ 的矩阵乘法，$2 \times b \times s^2 \times d_k$
  - 输入：$[b, head_num, s, per_head_d]$ 和 $[b, head_num, per_head_d, s]$
  - 输出：$[b, head_num, s, s]$
- 把相似度用到 V 上：要算一次 $s \times s$ 和 $s \times d_v$ 的矩阵乘法，，$2 \times b \times s^2 \times d_v$
  - 输入：$[b, head_num, s, s]$ 和 $[b, head_num, s, per_head_d]$
  - 输出：$[b, head_num, s, per_head_d]$
- 最后过一个线性映射：要算一次 $s \times d_v$ 的和 $d_v \times d_v$ 的矩阵乘法，$2 \times b \times s \times d_v \times d_v$
  - 输入：$[b, s, d_v]$ 和 $[d_v, d_v]$
  - 输出：$[b, s, d_v]$

因为 $d_k = d_v = d_q = d$，单纯计算 attention 总共就是 $8bsd^2 + 4bs^2d$

2.3.2 FFN 的 FLOPS

FFN 的公式：

$$x = f_{\text{gelu}}\left(x_{\text{out}} W_1\right) W_2 + x_{\text{out}}$$

在原始 Transformer 中，W_1 的 shape 是 $[d, 4d]$，W_2 的 shape 是 $[4d, d]$，

- 第一个线性层：$2 \times b \times s \times\ d \times 4d = 8 \times b \times s \times\ d^2$
  - 输入：$[b, s, d]$ 和 $[d, 4d]$
  - 输出：$[b, s, 4d]$
- 第二个线性层：$2 \times b \times s \times\ 4d \times d = 8 \times b \times s \times\ d^2$
  - 输入：$[b, s, 4d]$ 和 $[4d, d]$
  - 输出：$[b, s, d]$

所以一层 Transformer，即 attention+FFN 的计算量为 $(8bsd^2+4bs^2d)+16bsd^2=24bsd^2+4bs^2d$

有两点需要注意的：

- 对 NLP 任务来讲，一般 d 是个比较固定的值，如 512，而 s 变大，效果会更好，所以一般是 $s>d$，所以复杂度取决于 s 的大小。
- 但有些模型的初始设置不是这样的，例如 GPT3 的 175B 模型里，$s=2048, d=12288$，当然，算力够的话也可以把 s 变大

2.3.3 DIN 的 FLOPS

特殊地，对于推荐中的 DIN 那种，看当前 item 和历史 s 个 item 的相关性，即 q 的序列长度只有 1，不考虑多头，而这其实也是 decoder 预测下一个词时过一层 Transformer 的复杂度

已经有 3 个序列长度为 $s-1$ 的 QKV 的 cache，要算第 s 个词和这 $s-1$ 个词的 attention

- 计算第 s 个词的 3 个 Q、K、V：要算三次 $1\times d$ 和 $d\times d_v$ 的矩阵乘法，所以是：$3\times 2\times b\times 1\times d\times d_v$
  - 输入：$[b,1,d]$ 和 3 个 $[b,d,d_v]$
  - 输出：$[b,1,d_v]$
- 计算 Q 和 K 的相似度：要算一次 $1\times d_v$ 和 $d_v\times s$ 的矩阵乘法，$2\times b\times 1\times d_k\times s$【这里的 K 是历史 $s-1$ 长度的序列拼上当前词，当然对 DIN 来讲要去年当前词，这里先忽略这个】
  - 输入：$[b,1,d_v]$ 和 $[b,d_v,s]$
  - 输出：$[b,1,s]$
- 把相似度用到 V 上：要算一次 $1\times s$ 和 $s\times d_v$ 的矩阵乘法，，$2\times b\times 1\times d_v\times s$【同样地，这里的 V 是历史 $s-1$ 长度的序列拼上当前词，当然对 DIN 来讲要去年当前词，这里先忽略这个】
  - 输入：$[b,1,s]$ 和 $[b,s,d_v]$
  - 输出：$[b,1,d_v]$
- 最后过一个线性映射：要算一次 $1\times d_v$ 的和 $d_v\times d_v$ 的矩阵乘法，$2\times b\times 1\times d_v\times d_v$
  - 输入：$[b,1,d_v]$ 和 $[d_v,d_v]$
  - 输出：$[b,1,d_v]$
- 第一个线性层：$2\times b\times 1\times\ d\times 4d=8\times b\times 1\times\ d^2$
  - 输入：$[b,1,d]$ 和 $[d,4d]$
  - 输出：$[b,1,4d]$
- 第二个线性层：$2\times b\times 1\times\ 4d\times d=8\times b\times 1\times\ d^2$
  - 输入：$[b,1,4d]$ 和 $[4d,d]$
  - 输出：$[b,1,d]$

总共是 $6bd^2+2bds+2bds+2bd^2+8bd^2+8bd^2=24bd^2+4bds$

2.3.4 Transformer 的访存

GPU 架构的介绍参考https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/和https://www.zhihu.com/question/319355296/answer/2193938981，GPU 对比 CPU 如下：

- 任务模式
  - CPU 由专为顺序串行处理而优化的几个核心组成
  - GPU 则拥有一个由数以千计的更小、更高效的核心（专为同时处理多重任务而设计）组成的大规模并行计算架构。同时 CPU 相当的一部分时间在执行外设的中断、进程的切换等任务，而 GPU 有更多的时间并行计算。
- 功能定位
  - CPU 不但要承担计算任务还有承担逻辑控制等任务。
  - GPU 在渲染画面时需要同时渲染数以百万记的顶点或三角形，故 GPU 的设计是可以充分支持并行计算。
- 系统集成
  - GPU 作为一类外插设备，在尺寸、功率、散热、兼容性等方面的限制远远小于 CPU，这样可以让 GPU 有较大的显存和带宽。

以 A100 为例，整体架构如下

1. PCIE 层：通过 PCIE 接口以外设的方式集成到服务器上。
2. 中间一坨绿色的部分是 GPU 的计算核心 **SM(Streaming Multiprocessor)**，在 A100 中，一个 SM 有 64 个用于计算的 Core，共 108 个 SM(图里是 GA100, 有 128 个 SM)，故共 6192 个 Core。
3. 中间蓝色部分是 L2 缓存
4. NVLink：**多个 GPU 间进行通信**的组件，会优化 GPU 间的通信，提升传输效率。
5. 两侧的 HBM2 是显存，目前的 A100 的显存有两种 40G and 80G

A100 的 SM 如图所示，

GPU 的显存分成两部分：

- Global Memory：整体架构图中的**两侧 HBM2** 部分，例如 A100 80G 就有 80G 的 global memory，**2TB/s 带宽**，访问速度比较慢
- Shared Memory：SM 图中**浅蓝色**的 L1 Data Cache，例如 A100 中每个 SM 中有 **192KB**，访问速度比较快

从图中可见，A100 的 **FP16** 有 **312T 的 FLOPS**

Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]

Table 1. A100 Tensor Core GPU performance specs.
1) Peak rates are based on the GPU boost clock.
2) Effective TFLOPS / TOPS using the new Sparsity feature.

以矩阵乘法为例，$[M,K]\times[K,N]->[M,N]$，

- 计算时间：$2MKN/FLOPS$
- 访存时间为：$(MN+MK+KN)/memory_bandwidth$，因为首先要**读取**$MK$ 和 NK 这两个矩阵，然后结果还要**写入**MN 这个矩阵里。假设是 fp16，占 2bytes，那就还要乘以 2

假设 $b=1,s=4096,d=d_k=2048$，以计算 Q 和 K 的相似度为例，对比一下训练和预测时的**计算耗时**和**访存耗时**

- **训练时**：$M=4096,N=2048,K=4096$==> **计算是瓶颈**
  - FLOPS：$2\times b\times s^2\times d_k=2\times 1\times 4096^2\times 2048=68719476736$
  - 计算耗时：$FLOPS/max_FLOPS=68719476736/(312\times 10^{12})=0.00022s=220\times 10^{-6}s=220us$
  - 访存耗时：$(MN+MK+KN)/memory_bandwidth=2\times(4096\times 2048+4096\times 4096+4096\times 2048)/(2\times 10^{12})=3.35544\times 10^{-5}s=33.544\times 10^{-6}=33.5544us$
- **预测时**：$M=1,N=2048,K=4096$==> **访存是瓶颈**
  - FLOPS：$2\times b\times 1\times d_k\times s=2\times 1\times 2048\times 4096=16777216$
  - 计算耗时：$FLOPS/max_FLOPS=16777216/(312\times 10^{12})=5.38\times 10^{-8}s=0.0538\times 10^{-6}s=0.0538us$
  - 访存耗时：$(MN+MK+KN)/memory_bandwidth=2\times(1\times 2048+1\times 4096+4096\times 2048)/(2\times 10^{12})=8.3948\times 10^{-6}s=8.3948us$

一些常见的名词：

- H2D：host to device，从 cpu 拷贝到 gpu
- D2H：device to host，从 gpu 拷贝到 cpu

2.4 模型训练

mfu(Model Flops Utilization) 模型算力利用率是分布式训练效率的优化目标。

一个模型定义好之后，前向和反向的**计算量**就是固定（不考虑动态图的话）的，除以**每个 step 的 latency** 就是 mfu。以 nanoGPT 中的代码为例：

```
def estimate_mfu(self, fwdbwd_per_iter, dt):
    """ estimate model flops utilization (MFU) in units of A100 bfloat16 peak FLOPS """
    # first estimate the number of flops we do per iteration.
    # see PaLM paper Appendix B as ref: https://arxiv.org/abs/2204.02311
    N = self.get_num_params()
    cfg = self.config
    L, H, Q, T = cfg.n_layer, cfg.n_head, cfg.n_embd//cfg.n_head, cfg.block_size
    flops_per_token = 6*N + 12*L*H*Q*T
    flops_per_fwdbwd = flops_per_token * T # 计算量（T 是序列长度）
```

```python
    flops_per_iter = flops_per_fwdbwd * fwdbwd_per_iter # 每个 step 的 flops * 每一次更新梯度要多少个 step
    # express our flops throughput as ratio of A100 bfloat16 peak flops
    flops_achieved = flops_per_iter * (1.0/dt) # per second 一轮的计算量/一轮的耗时
    flops_promised = 312e12 # A100 GPU bfloat16 peak flops is 312 TFLOPS
    mfu = flops_achieved / flops_promised
    return mfu

##...
def xxx():
    # timing and logging
    t1 = time.time()
    dt = t1 - t0 ## 一次 gradient_accumulation_steps 后 更新梯度
    t0 = t1
    if iter_num % log_interval == 0 and master_process:
        # get loss as float. note: this is a CPU-GPU sync point
        # scale up to undo the division above, approximating the true total loss
        # (exact would have been a sum)
        lossf = loss.item() * gradient_accumulation_steps
        if local_iter_num >= 5: # let the training loop settle a bit
            mfu = raw_model.estimate_mfu(batch_size * gradient_accumulation_steps, dt)
            running_mfu = mfu if running_mfu == -1.0 else 0.9*running_mfu + 0.1*mfu
        print(f"iter {iter_num}: loss {lossf:.4f}, time {dt*1000:.2f}ms, mfu {running_mfu*100:.2f}%")
    iter_num += 1
    local_iter_num += 1
```

一般分布式训练参数量越多-> 卡数越多-> 通信占比越高->MFU 越低，所以要优化通信效率。

2.4.1 优化设置

- batchsize：通常用比较大的 batchsize，提高训练**稳定性**和**吞吐量**。GPT-3 和 PaLM 在训练时动态增加 batshzie，最终达到百万级别，batchsize **从 3.2w 逐渐增加到 320w 个 token**。
- 优化器：
  - Adam 和AdamW：基于**一阶梯度优化的低阶矩自适应估计**，用于 GPT-3 等，超参 $\beta_1 = 0.9, \beta_2 = 0.95, \epsilon = 10^{-8}$。
  - Adafactor：在训练过程中节省显存，用于 PaLM、T5 等，超参 $\beta_1 = 0.9, \beta_2 = 1.0 - k^{-0.8}$
- 学习率：
  - 预热（warm-up）：在训练的**初始 0.1% 到 0.5%** 的 steps 中，用**线性预热策略**逐渐增加学习率到最大值（5×10^{-5} 到 1×10^{-4} 之间，GPT-3 是 6×10^{-5}）
  - 衰减（decay）：后续 steps 中**余弦衰减**，逐渐降低到**最大值的约 10%**，直到收敛
- 稳定训练：
  - 权重衰减和梯度裁剪：**权重衰减率**设为 0.1，**梯度裁剪阈值**设为 1.0
  - 梯度检查点：容易出现 loss 突增，PaLM 和 OPT 从**发生突增前的一个 ckpt 重新开始训练**，并**跳过可能有问题的数据**
  - 缩减 emb 梯度：GLM 发现 emb 的异常梯度通常会导致 loss 突增，故**缩减 emb 梯度**以缓解

2.4.2 混合精度训练

2.4.2.1 FP16

Mixed precision training提出了用 16 位 float（FP16）训练，减少**内存使用和通信开销**。A100 等 GPU 具有的 **FP16 计算单元**是 **FP32 的两倍**，故 FP16 的计算效率能进一步提高。

Figure 1: Mixed precision training iteration for a layer.

- **推理（预测）**：所有参数都是 fp16，相对 fp32，存储变成一半，速度提升 1 倍。
- **训练**：参数和梯度用 **fp32 存储**，但是在**计算前会转成 fp16**，**计算后再转回 fp32**。主要为了**防止溢出**，loss 要乘一个 scale，然后在 fp16 的梯度要除以 scale。

以 adam 优化器为例，对于每 1 个参数来说，fp16 的训练占用 20bytes 显存，包括

- fp16 的参数：2bytes
- fp16 的梯度：2bytes（其实不一定是必须的，在 ZeRO 的论文中有分析）
- 优化器状态（optimizer state）：16bytes
  - fp32 参数（4bytes）
  - fp32 梯度（4bytes）
  - fp32 variance【历史梯度平方和】（4bytes）
  - fp32 momentum【历史梯度滑动平均】（4bytes）

而在预测时只要存一个 fp16 的参数 (2bytes) 就行，所以**预测的显存是训练的 1/10**，对应 1.3B 参数量的 gpt2-xl，训练要占用 $20B \times 1.3 \times 10^9 = 26GB$，预测只要 2.6GB

2.4.2.2 BF16

FP16 可能导致**计算精度的损失**从而影响模型性能，BLOOM 里用 **BF16**(brain floating point) 比 FP16 **分配更多指数位**和**更少的有效位**，在准确性方面更好

https://blog.csdn.net/orangerfun/article/details/133106913

bf16 的指数位和 fp32 一样多

2.4.3 可扩展的训练

需要**提高训练吞吐量**和**加载更大模型到显存中**

2.4.3.1 3D 并行

如下三种并行（数据并行、流水线并行、张量并行）的组合

2.4.3.1.1 数据并行（Data Parallelism）

将**模型参数和优化器状态复制**到多个 GPU 上，每个 GPU 只处理分给它的数据，不同 GPU 算出的梯度进行**聚合**得到 batch 的梯度，再更新**所有 GPU 上的模型**。高度可扩展，增加 GPU 数就能提高训练吞吐。

torch 的 ddp

```
from torch.nn.parallel import DistributedDataParallel as DDP

ddp = int(os.environ.get('RANK', -1)) != -1 # is this a ddp run?
if ddp:
    init_process_group(backend=backend)
    ddp_rank = int(os.environ['RANK'])
    ddp_local_rank = int(os.environ['LOCAL_RANK'])
    device = f'cuda:{ddp_local_rank}'
    torch.cuda.set_device(device)
    # this process will do logging, checkpointing etc.
    master_process = ddp_rank == 0
    seed_offset = ddp_rank # each process gets a different seed

# wrap model into DDP container
if ddp:
    model = DDP(model, device_ids=[ddp_local_rank])
```

可以一起搞的技巧——**梯度累积**，当显存不够跑较大的 batchsize 时，训练效果可能会很差，可以先跑**多个 mini-batch 的前向和反向**，把梯度累积起来，再**更新一次参数**，在数学上等价于**跑一个较大的 batchsize**。

```
# forward backward update, with optional gradient accumulation to simulate larger batch size
# and using the GradScaler if data type is float16
for micro_step in range(gradient_accumulation_steps):
    if ddp:
        # in DDP training we only need to sync gradients at the last micro step.
        # 最后一个 micro step 才要 sync 梯度
        model.require_backward_grad_sync = (micro_step == gradient_accumulation_steps - 1)
    with ctx:
        logits, loss = model(X, Y)
    loss.backward() # 只是计算梯度，并不真的更新
optimizer.step()
optimizer.zero_grad(set_to_none=True)
```

也可以用 torch 的 no_sync():

```
ddp = torch.nn.parallel.DistributedDataParallel(model, pg)
with ddp.no_sync():
    for input in inputs:
        ddp(input).backward()  # no synchronization, accumulate grads
ddp(another_input).backward()  # synchronize grads
```

2.4.3.1.2 流水线并行（Pipeline Parallelism）

将 LLM 的**不同层**分配到多个 GPU 上，一般 Transformer 模型中会**将连续的层加载到同一 GPU 上**，以减少在 GPU 间传输已计算的隐层状态或梯度的成本。简单的实现会导致 **GPU 利用率降低**，因为每个 GPU 要**等前一个完成计算**，导致不必要的**气泡开销**，如下方法可以提高流水线效率：

- GPipe：Gpipe: Efficient training of giant neural networks using pipeline parallelism
- PipeDream：PipeDream: Fast and Efficient Pipeline Parallel DNN Training，填充多个数据 batch+ 异步梯度更新？看下 paper 先。。。

1) GPipe

Figure 2: (a) An example neural network with sequential layers is partitioned across four accelerators. F_k is the composite forward computation function of the k-th cell. B_k is the back-propagation function, which depends on both B_{k+1} from the upper layer and F_k. (b) The naive model parallelism strategy leads to severe under-utilization due to the sequential dependency of the network. (c) Pipeline parallelism divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on different micro-batches simultaneously. Gradients are applied synchronously at the end.

Gpipe 主要思想：

- 图 a：把模型不同 layers 顺序放在 4 张卡上，0->3 卡流水线前向计算 loss，3->0 再反向计算 gradients
- 图 b：从时间顺序上看，**每张卡有 3/4 时间是空闲的**，GPU 利用率非常低
- 图 c：配合**梯度累积**，多个 mini-batch 可以同时跑在流水线里面，每张卡则有 3/(3+4) 的时间空闲（Bubble）

流水线并行的问题是中间有 **Bubble**。当卡数 K，梯度累积次数 M，则 $Bubble = (K-1)/(K-1+M)$

GPT 里用Weight Tying提升效果，输入和输出共享 vocab embedding

2) 重计算

重计算 (recomputation) 是对于 pipeline parallelism 非常重要的一个优化，最开始在Training Deep Nets with Sublinear Memory Cost一文中提到，在 **flash attention** 中也用了。

因为要做 pipeline+ 梯度累积，前向过程中的**激活值**要保存，以留给反向过程使用，保存很多份的激活值对显存造成了很大压力。recomputation(也叫 **checkpointing**) 用时间来换空间（反向的时候进行一次激活值的重计算），可以缓解显存问题。

pytorch 的实现。大致逻辑是包了一个 `autograd.Function`，前向时保存一些 inputs/rng_state(RNG state 是 Random Number Generator state 的缩写，**随机数生成器的状态**。在深度学习和其他计算任务中，随机数生成器用于初始化参数、决定正则化技术如 dropout 的行为，以及在训练过程中选择样本等。RNG 状态是指随机数生成器当前的内部状态，它可以用来在需要时重现或恢复特定的随机数序列，确保实验或模型训练的可重复性)，反向时重新计算

2.4.3.1.3 张量并行（Tensor Parallelism）

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

分解 LLM 的张量（参数矩阵），例如矩阵乘法 $Y = XA$，A 可以按列分成两个子矩阵 A_1 和 A_2，从而改为 $Y = [XA_1, XA_2]$，将 A_1 和 A_2 **放到不同 GPU 上**，然后就可能通过跨 GPU 通信将两个 GPU 的结果 merge。

- Megatron-LM：能扩展到更高维度的张量
- Colossal-AI：
  - 为更高维度的张量实现了张量并行，An efficient 2d method for training super-large deep learning models、Tesseract: Parallelize the tensor parallelism efficiently和Maximizing Parallelism in Distributed Training for Huge Neural Networks
  - 特别针对序列数据提出**序列并行** (Sequence Parallelism: Long Sequence Training from System Perspective)，详见下一节

参考https://zhuanlan.zhihu.com/p/622036840

原始矩阵乘法是 `[m,k], [k, n] -> [m, n]`，有如下两种矩阵分解的等效：

- **列并行（column parallelism）**：第一个矩阵不变，第二个矩阵**竖着劈成两半**，即 $B = [B_1, B_2]$
  - `[m,k], [k, n/2] -> [m, n/2]`
  - `concat([m, n/2], [m, n/2]) -> [m, n]`
- **行并行（row parallelism）**：两个矩阵都横着劈成两半，即 $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$。从 2 推广到 k，其实就是 **split-k 算法**，把两个矩阵都分成 k 个小块，两两相乘后，最后 reduce_sum 一下。因为每个线程计算的矩阵更小了，开销小，可以通过加大线程数来提升并行效率。
  - `[m, k/2], [k/2, n] -> [m, n]`
  - `elemwise_add([m, n], [m, n]) -> [m, n]`

行并行还可以扩展到**推荐**里，假设 user 有 k/2 维，item 也是 k/2 维，concat 在一起，然后过一个 k*d 的 mlp，即 `[1,k] * [k, d] -->[1,d]`，那么可以按行并行的方法，拆成 2 个 `[1, k/2]` 和 `[k/2,d]` 相乘，再相加。这样 item 侧的 `[k/2,d]` 可以把全库缓存过来，在线实时算 user，排序时把对应 item 向量抽出来，和 user 加起来就行

(a) MLP

(b) Self-Attention

Figure 3. Blocks of Transformer with Model Parallelism. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

megatron 对 transformer 进行了如下优化：

- MLP 第一个 nn 按**列分割**，第二个 nn 按**行分割**，中间省了一次通信
- Attention 按照 head 来分割 (类似**列分割**)，后面接的 nn 按**行分割**，中间也省了一次通信

图里面的通信算子

- f 是前向 identity，反向 all-reduce
- g 是前向 all-reduce，反向 identity

综合来看，一层 transformer layer 如下

Figure 4. Communication operations in a transformer layer. There are 4 total communication operations in the forward and backward pass of a single model parallel transformer layer.

具体的计算量可以参考https://colossalai.org/docs/features/1D_tensor_parallel/#introduction：

Let's take a linear layer as an example, which consists of a GEMM $Y = XA$. Given 2 processors, we split the columns of A into $[A_1\ A_2]$, and calculate $Y_i = XA_i$ on each processor, which then forms $[Y_1\ Y_2] = [XA_1\ XA_2]$. This is called a column-parallel fashion.

When a second linear layer $Z = YB$ follows the column-parallel one, we split B into

$$\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

which is called a row-parallel fashion. To calculate

$$Z = [Y_1\ Y_2] \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

we first calculate $Y_i B_i$ on each processor, then use an all-reduce to aggregate the results as $Z = Y_1 B_1 + Y_2 B_2$.

We also need to note that in the backward pass, the column-parallel linear layer needs to aggregate the gradients of the input tensor X, because on each processor i we only have $\dot{X}_i = \dot{Y}_i A_i^T$. Thus, we apply an all-reduce across the processors to get $\dot{X} = \dot{Y} A^T = \dot{Y}_1 A_1^T + \dot{Y}_2 A_2^T$.

Efficiency

Given P processors, we present the theoretical computation and memory cost, as well as the communication cost based on the ring algorithm in both the forward and backward pass of 1D tensor parallelism.

Computation	Memory (parameters)	Memory (activations)	Communication (bandwidth)	Communication (latency)
$O(1/P)$	$O(1/P)$	$O(1)$	$O(2(P-1)/P)$	$O(2(P-1))$

2.4.3.2 ZeRO

ZeRO: Memory Optimization Towards Training A Trillion Parameter Models

fp16 那一节中，optimizer state 的显存占用，在**前向**和**反向**的时候都**不用**，只有最后 **optimizer step** 的时候才用。

===>zero 的思想：把 optimizer state **分 shard 存在不同的卡上**，只在**最后 gather 时才用**。

ZeRO（Zero Redundancy Optimizer）在 DeepSpeed 库中提出，解决**数据并行**中的**内存冗余**问题。数据并行其实并不需要每个 GPU 都存整个模型、梯度和优化器参数，ZeRO 在每个 GPU 仅保存部分数据，当需要其余数据时从其他 GPU 检索。3 种解决方案：

- 优化器状态分区：zero1，对显存最大开销的部分进行 shard
- 梯度分区：zero2
- 参数分区：zero3

前两种方案不会增加通信开销，第三种方案增加约 50% 通信开销，但能节省和 gpu 数成比例的内存。

Figure 1: Memory savings and communication volume for the three stages of ZeRO compared with standard data parallel baseline. In the memory consumption formula, Ψ refers to the number of parameters in a model and *K* is the optimizer specific constant term. As a specific example, we show the memory consumption for a 7.5B parameter model using Adam optimizer where *K*=12 on 64 GPUs. We also show the communication volume of ZeRO relative to the baseline.

详见官方博客：ZeRO & DeepSpeed: New system optimizations enable training models with over 100 billion parameters

```
import deepspeed
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--local_rank", type=int, default=0)
```

```
deepspeed.add_config_arguments(parser)
args = parser.parse_args()

model, optimizer, _, _ = deepspeed.initialize(args=args,
                                              model=model,
                                              model_parameters=model.parameters())
X, Y = get_batch('train')
logits, loss = model(X, Y)
model.backward(loss)
model.step()
```

需要指定 deepspeed 的配置:

```
{
  "train_batch_size": 64,
  "gradient_accumulation_steps": 1,
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 6e-4,
      "weight_decay": 1e-2,
      "betas": [0.9, 0.95]
    }zer
  },
  "scheduler": {
    "type": "WarmupLR",
    "params": {
        "warmup_min_lr": 6e-5,
        "warmup_max_lr": 6e-4,
        "warmup_num_steps": 2000
    }
  },
  "bf16": {
    "enabled": true
  },
  "zero_optimization": {
    "stage": 1
  }
}
```

启动:

```
deepspeed --num_gpus=8 train.py --deepspeed_config xx.json
```

facebook 的开源库 **FSDP(full sharded data parallel)**(Fairscale: A general purpose modular pytorch library for high performance and large scale training) 里基于 pytorch 实现了类似 ZeRO 的技术。

```
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP, ShardingStrategy
model = FSDP(model)  #, sharding_strategy=ShardingStrategy.SHARD_GRAD_OP)
```

还有一些 paper 也能降低内存，如

Reducing activation recomputation in large transformer models

Training deep nets with sublinear memory cost

2.4.3.3 序列并行

序列并行 (Sequence Parallelism: Long Sequence Training from System Perspective)，可以进一步分解 Transformer 的注意力操作。

Reducing Activation Recomputation in Large Transformer Models这个也是

对比 TP：

Figure 4: Transformer layer with tensor parallelism. f and $\bar{f}$ are conjugate. f is no operation in the forward pass and all-reduce in the backward pass. $\bar{f}$ is all-reduce in the forward pass and no operation in the backward pass.

SP：

Figure 5: Transformer layer with tensor and sequence parallelism. g and $\bar{g}$ are conjugate. g is all-gather in the forward pass and reduce-scatter in the backward pass. $\bar{g}$ is reduce-scatter in forward pass and all-gather in backward pass.

2.4.3.4 综合对比各种并行

几个缩写：params（p）/gradients(g)/optimizer states(os)/activation(a)

并行方法	显存效率	计算效率	限制
DP（数据并行）	p/g/os 都复制在每张卡上，显存效率很低	计算和通信可以 overlap，如果都在一个 minipod 内扩展性很好；梯度累积可以提高计算效率	batchsize 不能太大，否则模型效果有损；batchsize/dp 不能太小，不然打不满 tensorcore
ZeRO（解决 DP 的显存冗余）	zero1/2/3 把 os/g/p 分别 shard 到每张卡上，显存效率很高	需要做 prefetch 来减少通信对计算效率的影响	同 DP
PP（流水线并行）	切分 p，提高显存效率；a 需要存多次，降低显存效率	通信次数最少，只发生在多层之间的切分点，但是有 Bubble	每个 Stage 之间需要负载均衡，对模型结构和卡数有限制
TP（张量并行）	p/g/os/a 被 shard 在每张卡上，显存效率也很高；有些层如 layernorm 是复制的，可以用 sequence parallel 优化	梯度不需要同步，提高计算效率；每层插入了 4 次通信，而且是跟计算有依赖的，会降低计算效率；每层的计算量进行了切分，也会降低计算效率	一般是单机内 8 卡使用 nvlink 时用 TP

把神经网络看成是输入 X 和权重 W 的矩阵乘法 XW，那么，**DP 和 PP 其实是对 X 的拆分**，而 **TP 则是对 W 的拆分**

整体对比可以看

Number of parameters (billion)	Attention heads	Hidden size	Number of layers	Tensor model-parallel size	Pipeline model-parallel size	Number of GPUs	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7	24	2304	24	1	1	32	512	137	44%	4.4
3.6	32	3072	30	2	1	64	512	138	44%	8.8
7.5	32	4096	36	4	1	128	512	142	46%	18.2
18.4	48	6144	40	8	1	256	1024	135	43%	34.6
39.1	64	8192	48	8	2	512	1536	138	44%	70.8
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1
310.1	128	16384	96	8	16	1920	2160	155	50%	297.4
529.6	128	20480	105	8	35	2520	2520	163	52%	410.2
1008.0	160	25600	128	8	64	3072	3072	163	52%	502.0

Table 1: Weak-scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

一般这么整合：

- 把机器分成 N 组，**不同组之间用 DP**
- 一组机器有 M 台机器，**不同机器之间用 PP**
- 一台机器有 K 张卡，**不同卡之间用 TP**

2.4.4 编译优化

pytorch 的 TorchDynamo

https://pytorch.org/docs/stable/torch.compiler_deepdive.html

最简单的用法 `torch.compile()`

2.4.5 flash attention

Flashattention: Fast and memory-efficient exact attention with io-awareness

FlashAttention 其实是对 $softmax(QK^T)V$ 的一种加速实现。

一般的实现：需要先用矩阵 C 存 QK^T 的结果，然后对 C 按行做 softmax 得到新的 C，再用 C 乘以 V 得到最后结果。

FlashAttention 通过一些特殊技巧，不需要算出 C 这个临时变量，通过分块计算，**让临时变量总是可以放在 cache 里**，从而

- 减少 Global Memory 的大小
- 加速 attenttion 的计算，因为读 cache 比访问 Global Memory 快多了。

2.5 推理速度优化

2.5.1 量化

ZeroQuant-V2: Exploring Post-training Quantization in LLMs from Comprehensive Study to Low Rank Compensation和Compression of generative pre- trained language models via quantization

- int8 量化：LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale
- int4 量化：GLM 中用了

3 微调

- 指令微调（instruct tuning）：增强/解锁 LLM 的能力，
- 对齐微调（alignment tuning）：将 LLM 的行为与为类的价值观或偏好对齐。
- 高效微调方法：用于模型快速适配

3.1 指令微调

- 收集或构建指令格式 (instruction-formatted) 的实例
- 使用这些示例进行**有监督微调**

详见综述Is prompt all you need? no. A comprehensive and broader view of instruction learning

3.1.1 构建格式化实例

指令格式的实例包括一个任务描述（即**指令**）、一对输入输出和少量示例（可选）

3.1.1.1 格式化已有数据集

- **收集来自不同领域（文本摘要、文本分类、翻译等）的实例**来创建有监督的多任务训练数据集。用自然语言的任务描述来格式化这些数据集是很方便的。
- 使用**人类撰写的任务描述**来增广带标的数据集，通过**解释任务目标**来指导 LLM 理解任务。
- 众包平台（如 PromptSource）有效地创建、共享和难不同数据集的任务描述
- 通过指令微调特殊设计的任务描述，**反转**已有实例的输入-输出对，例如“请基于以下答案生成一个问题”，如
- 利用**启发式任务模板**将大量**无标注的文本**转换为**带标注的实例**。如Learning instructions with unlabeled data for zero-shot cross-task generalization

3.1.1.2 格式化人类需求

来自公共 NLP 数据集的训练实例虽然进行了格式化，但**任务描述缺乏多样性**或**与人类真实需求不匹配**，故 InstructGPT 采用真实用户提交给其 API 的查询作为任务描述。此外，为了丰富任务多样性，通常

- 标注者为**真实生活中的任务**编写指令，如开放式生成、开放式问答、头脑风暴、聊天等
- 另一组**标注人员**直接对这些指令进行**回答**
- 将**指令（采集的用户查询）**和**期望输出（人工编写的答案）**pair 对作为一个训练实例

还有一些**半自动化**的方法将**现有实例**输入到 LLM 中生成多样的任务描述和实例来构建实例，如 + Self-instruct: Aligning language model with self generated instructions，引用数好几百 + Unnatural instructions: Tuning language models with (almost) no human labor，meta 的论文 + Stanford alpaca: An instruction-following llama model

3.1.1.3 构建实例的关键

- 增加指令：
  - **扩大任务数量**：可以极大提高 LLM 的泛化能力。但随着任务增加，模型性能最初是连续增长，但**任务数量达到一定水平时，性能基本不提升了**。Scaling instruction-finetuned language models猜测，一定数量的代表性性任务就能够提供足够充足的知识了。
  - 增强**任务描述的多样性**：从如长度、结构、创造力等方面入手，如Multitask prompted training enables zero-shot task generalization
  - **每个任务的实例数量**：通常**少量实例**就可以让模型有不错的泛化能力，当某些任务的实例数量进一步增加（至数百个）时可能会**过拟合**。如Super-NaturalInstructions: Generalization via Declarative Instructions on 1600+ NLP Tasks
- 设计格式：
  - **任务描述**：LLM 理解任务的**最关键部分**
  - **适当数量的示例**：能产生**实质性的改进**，也减轻对指令工程的敏感性。如Scaling instruction-finetuned language models
  - 指令中的其他部分：如避免事项、原因、建议，**影响很小，甚至有负面影响**，如Cross-task generalization via natural language crowd- sourcing instructions
  - 包含**推理数据集**的 **CoT 实例**：Scaling instruction-finetuned language models和OPT-IML: scaling language model instruction meta learning through the lens of generalization提到同时用包含和不包含 CoT 的样本微调，能在各种下游任务取得好的效果，包括需要多级推理能力的任务（常识问答、算术推理）和不需要多级推理的任务（如情感分析和抽取式问答）。

3.1.2 指令微调策略

相比预训练而言，指令微调有多个不同：

- 训练目标函数：如 seq2seq 的 loss
- 优化参数设置：更小的 batchsize 和学习率
- 平衡数据分布：平衡不同任务间的比例：

  - **实例比例混合策略**（Exploring the limits of transfer learning with a unified text-to-text transformer），把所有数据集合并，然后从混合数据集中**按比例采样**每种实例。
  - **提高高质量数据集的采样比例**能提升效果，如Finetuned language models are zero-shot learners的 FLAN和Promptsource: An integrated development environ- ment and repository for natural language prompts的 P3。
  - 设置**最大容量**：限制**数据集中能包含的最大实例数**，防止较大数据集挤占整个采样集合，通常设置为几千或几万，如Exploring the limits of transfer learning with a unified text-to-text transformer和OPT-IML: scaling language model instruction meta learning through the lens of generalization。
- 结合指令微调和预训练：
  - 在**指令微调时加入预训练数据**：，如 OPT-IML，可以看成是**对模型的正则化**。
  - **混合预训练数据（纯文本）和指令微调（指令格式）数据**，用多任务方式**从头训练**：Exploring the limits of transfer learning with a unified text-to-text transformer和Ext5: Towards extreme multi-task scaling for transfer learning。将指令格式数据集作为预训练语料库的一小部分来预训练，同时获得预训练和指令微调的优势，如 GLM-130B 和 Galactica。

3.1.3 指令微调效果

3.1.3.1 性能改进

- 不同规模的模型都能从指令微调中受益，**随着参数规模增加，性能也有提升**。Multitask prompted training enables zero-shot task generalization发现，**指令微调后的小模型**甚至能比**未经微调的大模型效果更好**
- 指令微调在**不同模型架构**、**预训练目标**和**模型适配方法**上都有稳定改进效果，由Scaling instruction-finetuned language models发现
- 指令微调是**提升现有 LM（包括小型 PLM）能力**的一个通用方法，同样由Scaling instruction-finetuned language models发现
- LLM 所需的**指令数据数量明显少于预训练数据**，故指令微调的**成本较低**。

3.1.3.2 任务泛化性

- 赋予 LLM **遵循人类指令执行特定任务的能力**（通常被视为一种涌现能力）：Scaling instruction-finetuned language models发现，指令微调鼓励 LLM **理解**用于完成任务的**自然语言指令**，，即**在未见过的任务上也能执行**。
- 使 LLM 具有更强的**解决现实世界任务**的能力：指令微调能帮助 LLM **缓解一些弱点**（如**生成重复内容**或**补全输入但完不成成相应任务**），由Scaling instruction-finetuned language models和Training language models to follow instructions with human feedback发现。
- 指令微调后的 LLM 能**泛化**到**其他语言**的相关任务上：Crosslingual generalization through multitask finetuning提出的 BLOOMZ-P3 基于 BLOOM 在**纯英文**的 P3 任务集合上进行微调，在多语言的句子实例任务中，相比 BLOOM 有超过 50% 的性能提升，同时仅用英文指令就能产生不错效果，**减少针对特定语言的指令工程的工作量**。

3.2 对齐微调

Training language models to follow instructions with human feedback和Alignment of language agents提出，LLM 可能**编造虚假信息**、产生**有害**的、**误导性**的和**有偏见**的表达，因为 LLM 在预训练时没有考虑人类的价值观或偏好。

Improving alignment of dialogue agents via targeted human judgements和Training language models to follow instructions with human feedback提出了人类对齐，使 LLM 的行为能够符合人类期望。

Training language models to follow instructions with human feedback、A general language assistant as a laboratory for alignment和Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback发现，和适配微调（如指令微调）相比，对齐微调要考虑的标准并不同，这可能会在某种程度上**损害 LLM 的通用能力**，即**对齐税**。

3.2.1 对齐的标准

- **有用性**：以**简洁**且**高效**的方式帮助用户解决任务或回答问题。需要进一步阐明问题时，应该有通过**提出恰当的问题**来**获取额外信息**的能力，并有合适的**敏感**度、**洞察**力和**审慎**度（from A general language assistant as a laboratory for alignment）。
- **诚实性**：又称为**正确性**，提供准确内容，传达**适当的不确定性**很重要，**避免任何形式的欺骗或信息误传**。LLM 了解其能力和知识水平（**知道自己不知道什么**）。A general language assistant as a laboratory for alignment）认为，与有用性和无害性相比，诚实性是一个**更客观**的标准，故诚实性对齐**依赖的人力可能更少**。

- **无害性**：生成的语言不得是冒犯性或者歧视性的，能**检测**到**隐蔽的出于恶意目的的请求**。当**被诱导去执行危险行为**（如犯罪）时，应该**礼貌拒绝**。Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback提出，某个行为**是否有害**及**有害程度**因**个人**和**社会**而异。

对齐的标准**很主观**，难以直接作为 LLM 的优化目标。比较有前景的方法是Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned和Red teaming language models with language models提出的**红队攻防**，用**对抗**的方式**手动**或**自动**地**探测 LLM**，使其**生成有害输出**，再**更新模型防止此类输出**。

3.2.2 收集人类反馈

3.2.2.1 选择标注人员

- **教育水平要求高**：Sparrow 要求本科学历的英国人，Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback中的高优任务有一半是美国硕士
- **意图一致性筛选**：InstructGPT 通过**标注人员和研究人员**意图一致性来选择标人员。研究者先自己标少量数据，然后衡量自己和标注人员间标的一致性，选择一致性最高的标注人员来进行后续标注。
- **选择优秀标注者**：Teaching language models to support answers with verified quotes中，研究人员评估标注人员的表现，选出如高一致性之类的一组优秀标注人员继续合作，Learning to summarize from human feedback发现，在标注时提供**详细的标注指令**和**实时的指导**是有帮助的。

3.2.2.2 收集反馈

- 基于排序的方法：
  - **只选最佳候选**：Fine-tuning language models from human preferences和Recursively summarizing books with human feedback在这种早期工作中，标注人员用比较粗略的方式评估模型生成的结果，如只选择最佳候选。一方面不同人意见不同，另一方面这种方法忽略了没被选中的样本。
  - **elo 评分系统**：Improving alignment of dialogue agents via targeted human judgements和Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback提出了 elo 评分系统，两两比较所有候选输出结果，生成一个偏好排序。
- 基于问题的方法：回答**研究人员设计的特定问题**，这些问题覆盖**不同的对齐标准**以及其他**对 LLM 的约束**条件。例如 WebGPT 中，标注人员要回答关于检索到的文档**对回答给定输入是否有帮助**的选择题。
- 基于规则的方法：
  - Sparrow 不仅选择**标注人员挑选的最佳回复**，还设计**一系列规则**来**测试模型生成的回复**是否符合**有用**、**正确**、**无害**的标准，让**标注者**对模型生成的回复**违反规则的程度进行打分**。
  - GPT-4 用一组基于 GPT-4 的 **zero-shot 分类器**作为**基于规则的奖励模型**，**自动**确定模型生成的**输出是否违反一组人类编写的规则**。

3.2.3 RLHF

详见 RLHF 章节

3.3 高效微调

全量参数都微调成本很大，有更高效的方法，称为**参数高效微调**（**parameter-efficient fine-tuning**）。

3.3.1 适配器微调（adapter tuning）

Parameter-efficient transfer learning for NLP提出，在 Transformer 中引入一个**小型神经网络模块**（**适配器**），LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models也提出了瓶颈架构：

- 将原始特征压缩到**较小维度**（然后进行非线性变换）
- 恢复到**原始维度**

一般是**串行插入**的方式，集成到**每个 Transformer 层里**，分别放到**注意力层**和**前馈层之后**。Towards a unified view of parameter-efficient transfer learning提出了**并行适配器**，即与**注意力层和前馈层并行**。

微调时，**原参数不变**，**仅更新适配器模块参数**。

3.3.2 前缀微调（prefix tuning）

Prefix-tuning: Optimizing continuous prompts for generation。

- 在每个 **Transformer 层前**添加一系列前缀，即一组**可训练的连续向量**。前缀向量具有**任务的特异性**，可以看作**虚拟的 token emb**。
- **重参数化**技巧：
  - 学习一个将**较小矩阵映射到前缀参数矩阵**的 **MLP 函数**，而不是直接优化前缀，有助于**稳定训练**。
  - 优化后，**舍弃映射函数**，只保留派生的前缀向量以增强与特定任务相关的性能。
  - 由于只训练前缀参数，故能实现参数高效的模型优化

P-tuning v2: Prompt tuning can be comparable to fine- tuning universally across scales and tasks提出了 p-tuning v2，为了自然语言理解在 Transformer 中引入**逐层提示向量**，还利用**多任务学习**来**联合优化共享的提示**。

3.3.3 提示微调（prompt tuning）

在**输入层**加入**可训练的提示向量**，基于离散提示方法（How can we know what language models know?和Autoprompt: Eliciting knowledge from lan- guage models with automatically generated prompts），通过包含一组**软提示 token** 来扩充输入文本，再用扩充后的输入来解决特定的下游任务。将**任务特定的提示 emb** 与**输入文本的 emb** 相结合，输入模型中。

- GPT understands, too：提出了 P-tuning，用**自由形式**来组合**上下文**、**提示**和**目标 token**，用**双向 LSTM** 学习**软提示 token 的表示**，适用于自然语言理解和生成的架构。
- The power of scale for parameter-efficient prompt tuning：提示微调，直接在**输入前**加入**前缀提示**。训练时**只有提示 emb** 会根据特定任务进行监督学习。这种方法在**输入层**只包含**少量可训练参数**，故其效果**高度依赖底层语言模型的能力**。

3.3.4 低秩适配（LoRA）

Lora: Low-rank adaptation of large language models通过增加低秩约束来近似每层的更新矩阵，假设参数矩阵 $\mathbf{W} \in \mathbb{R}^{m\times n}$，一般是

$$\mathbf{W} = \mathbf{W} + \Delta\mathbf{W}$$

冻结 $\mathbf{W}$，通过低秩分解矩阵来近似更新

$$\Delta\mathbf{W} = \mathbf{A}\cdot\mathbf{B}^{\top}$$

其中 $\mathbf{A} \in \mathbb{R}^{m\times k}$ 和 $\mathbf{B} \in \mathbb{R}^{n\times k}$ 是用于任务适配的可训练参数，$r \ll \min(m, n)$ 是**降低后的秩**。

LoRA 的优点：

- 大大**节省内存和存储**（如 VRAM，Video Random Access Memory）
- 可以只**保留一个大型模型副本**，同时**保留多个**用于**适配不同下游任务**的**特定低秩分解矩阵**。

用更有原则的方法设置秩：

- 基于**重要性分数**的分配：Adaptive budget allocation for parameter-efficient fine-tuning提出的 AdaLoRA
- **无需搜索**的**最优秩选择**：Dylora: Parameter efficient tuning of pre-trained models using dynamic search-free low- rank adaptation

3.3.5 小结

LoRA 已经有广泛的应用，如 LLaMA 和 BLOOM，

- Alpaca-LoRA：Instruct-tune llama on consumer hardware，通过 LoRA 训练的 Alpaca 的轻量级微调版本。
- LLaMA-Adapter：Llama-adapter: Efficient fine-tuning of language models with zero-init attention将**可学习的提示向量**插入每个 Transformer 层中，提出**零初始化的注意力**，通过**减轻欠拟合提示向量的影响**以改善训练，还能扩展到**多模态设置**，如视觉问答。

LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models比较了串行适配器微调、并行适配器微调和 LoRA，在 GPT-J(6B)、BLOOM(7.1B) 和 LLaMA(7B) 上评估：这些方法在**困难任务上效果不如 GPT-3.5**，但**在简单任务上表现相当**，**LoRA** 表现相对较好且使用的可训练参数明显较少。

huggingface 开源了Peft: State-of-the-art parameter-efficient fine- tuning methods，包括 LoRA/AdaLoRA、前缀微调、P-Tuning、提示微调，支持 GPT-2 和 LLaMA，还支持视觉 Transformer 如 ViT 和 Swin Transformer。

4 使用

4.1 上下文学习

GPT-3 提出 ICL，将**任务描述**和（或）**示范（demonstration）**以**自然语言文本形式**表达。

4.1.1 上下文学习形式

- 以**任务描述**作为开始，从任务数据集中**选择一些样例**作为**示范**。
- 以特别设计的**模板形式**将它们按照**特定的顺序**组合成**自然语言提示**。
- 将**测试样例**添加到 LLM 的输入中以生成输出。

形式化地看，$D_k = \{f(x_1, y_1), \ldots, f(x_k, y_k)\}$ 表示由 k 个样例组成的一组示范，$f(x_k, y_k)$ 表示把第 k 个**任务样例转换为自然语言提示**的函数。给定任务描述 I、示范 D_k 和新的输入查询 x_{k+1}，LLM 生成的输出 $\hat{y}_{k+1}$ 如下：

$$\text{LLM}(I, \underbrace{f(x_1, y_1), \ldots, f(x_k, y_k)}_{\text{示范}}, f(\underbrace{x_{k+1}}_{\text{输入}}, \underbrace{___}_{\text{答案}})) \to \hat{y}_{k+1}.$$

真实答案 y_{k+1} 留白，由 LLM 预测。

更多的可以参考综述A survey for in-context learning

指令微调可以**提高 LLM 执行目标任务的 ICL 能力**，尤其是**零样本场景**（仅使用任务描述）。

4.1.2 示范设计

4.1.2.1 示范选择

- 启发式方法：
  - 基于 **knn 的检索器**来选择与**查询**语义相关的样例：如What makes good in-context examples for gpt-3?和Does GPT-3 generate empathetic dialogues? A novel in-context example selection method and automatic evaluation metric for empathetic dialogue generation。但只是**针对每个样例单独选择**，而**不是对整个样例集合**进行评估。
  - 基于**多样性**的选择策略：Diverse demonstrations improve in-context compositional generalization和Selective annotation makes language mod- els better few-shot learners
  - 同时考虑**相关性**和**多样性**的选择策略：Complementary Explanations for Effective In-Context Learning
- 基于 LLM 的方法：
  - **直接用 LLM** 来**选择**：Finding supporting examples for in-context learning：LLM 可以直接根据**添加样例后**的**性能提升**评估**每个样例的信息量**，以进行选择。

- **两阶段检索**：Learning to retrieve prompts for in-context learning：提出 EPR，先用无监督方法召回相似样例，再用密集检索器（用 LLM 标记的正负样例训练）进行排序。
- **RL 方法**：Active example selection for in-context learning，将示范选择任务建模为 RL 问题，**LLM 是奖励函数**，为训练策略模型提供反馈。
- 用 **LLM** 来**生成**示范：Chatgpt outperforms crowd-workers for text-annotation tasks发现 LLM 在文本标方面表现很好，故可以直接将 LLM 作为**无人工干预的示范生成器**，如Self-generated in-context learning: Leveraging auto-regressive language models as a demonstration generator和Selective in-context data augmentation for intent detection using pointwise v-information

An explanation of in-context learning as implicit bayesian inference提到，ICL 中选择的示范样例应该包含**足够的有关待解决任务的信息**，并**与测试查询相关**。

4.1.2.2 示范格式

将选择的示范进行**整合**以及**格式化**：

- 用相应的**输入输出对**来**实例化预定义的模板**：Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing
- 增强 LLM 的**推理能力**
  - **添加任务描述**：Scaling instruction-finetuned language models
  - 通过 **CoT 提示**：Chain of thought prompting elicits reasoning in large language models
- 收集**包含人工编写的任务描述**的**大规模数据集**：Cross-task generalization via natural language crowd- sourcing instructions，能够提升**已见任务**的性能，也能在一定程度泛化到**未见任务**。
- **半自动化**方法：Self-instruct: Aligning language model with self generated instructions使用由**人工编写的任务描述**组成的**种子集合**来指导 LLM **为新任务生成任务描述**。
- **自动生成**高质量的示范格式：
  - **Auto-CoT**：Automatic chain of thought prompting in large language models使用零样本提示（**let's think step by step**）以生成中间推理步骤
  - **least-to-most 提示**：Least-to-most prompting enables complex reasoning in large language models先询问 LLM 来**执行问题分解**，再利用 LLM **根据已解决的中间答案**依次**解决子问题**。

4.1.2.3 示范顺序

LLM 有时会被**顺序偏差**影响，例如Calibrate before use: Improving few-shot performance of language models提出 LLM 会倾向于**重复示范结尾附近的答案** ===> **结尾很重要**!!

- **启发式**方法：What makes good in-context examples for gpt-3?根据在 emb 空间中**示范**与**查询**的**相似度**来排列，相似度越高，**距离结尾越近**。
- **基于信息论**的方法：
  - Self-adaptive in-context learning使用**最小化压缩和传输任务标签所需的码长**来整合更多任务信息，需要**额外的标记数据**作为用来**评估**特定示范**顺序性能**的**验证集**。
  - Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity使用**全局**和**局部熵度量**来为不同的示范顺序打分，且为了消除对额外标注数据的需要，这篇文章**从 LLM 本身采样**来获取验证集。

4.1.3 底层机制

4.1.3.1 预训练如何影响 ICL

- ICL 与**预训练任务设计**：GPT-3 发现 ICL 能力随模型增大而增强，但Metaicl: Learning to learn in context发现**小规模的 PLM** 也能通过**特别设计的训练任务**从而表现出强大的 ICL 能力（例如输入是**任务实例 + 查询**，**预测标签**），甚至能超越规模更大的模型。
- ICL 与**预训练语料**：

- On the effect of pretraining corpora on in-context learning by a large-scale language model发现 ICL 的性能主要取决于**预训练语料的来源**而非规模
- Data Distributional Properties Drive Emergent In-Context Learning in Transformers分析训练数据分布的影响，发现当训练数据可以**被聚类成多个不常见的类**，而**不是均匀分布**时，模型会有 ICL 能力
- An explanation of in-context learning as implicit bayesian inference从理论上解释，认为 ICL 是在具备**长程连贯性的文档**上进行预训练的产物。

4.1.3.2 LLM 如何实现 ICL

- 将 ICL 视为**隐式微调**：Why can GPT learn in-context? language models secretly perform gradient descent as meta-optimizers和Transformers learn in-context by gradient descent
  - ICL 可以看成是通过**前向计算**，LLM 生成**关于示范的元梯度**，并通过**注意力**机制**隐式**地**梯度下降**。
  - LLM 的**某些注意力头**能执行**与 ICL 能力密切相关的任务无关的原子操作**（如**复制**、**前缀匹配**等）
- 将 ICL 视为**算法学习过程**：Transformers as algorithms: Generalization and implicit model selection in in-context learning、What learning algorithm is in-context learning? investigations with linear models，基于这个解释框架，LLM 能通过 ICL 有效地学习简单的线性函数，甚至是如决策树的复杂函数
  - 预训练阶段：LLM 本质上通过其参数**对隐式模型进行编码**
  - 前向计算阶段：通过 ICL 中提供的示例，LLM 可以**实现如 sgd 的学习算法**，或者**直接计算出闭式解**以更新这些模型

4.2 思维链提示（CoT）

CoT 是一种改进的提示策略，旨在提高 LLM 在**复杂推理任务**中的性能，如算术推理（Training verifiers to solve math word problems、Are NLP models really able to solve simple math word problems?和A diverse corpus for evaluating and developing english math word problem solvers）、常识推理（Commonsenseqa: A question answering challenge targeting commonsense knowledge和Did aristotle use a laptop? A question answering benchmark with implicit reasoning strategies）、符号推理（Chain of thought prompting elicits reasoning in large language models）。

ICL **只使用输入输出对**来构造提示，而 CoT 将最终输出的**中间推理步骤**加入提示。

4.2.1 使用 CoT 的 ICL

一般在小样本和零样本这两种设置下和 ICL 一起用

4.2.1.1 小样本思维链

将每个示范 < 输入，输出 > 替换为 < 输入，CoT，输出 >。小样本 CoT 可以看成 ICL 的一种特殊提示，但相比 ICL 的标准提示，**示范的顺序**对性能**影响相对较小**。

- **思维链提示设计**：
  - 使用**多样的 CoT 推理路径**：Making Large Language Models Better Reasoners with Step-Aware Verifier，对**每个问题**给出**多个推理路径**。
  - 使用具有**复杂推理路径**的提示：Complexity-based prompting for multi-step reasoning
  - Auto-CoT：上述方法都需要标注 CoT，Automatic chain of thought prompting in large language models利用Large language models are zero-shot reasoners提出的 zero-shot-CoT
    - 通过**特别提示** LLM 来生成 CoT 推理路径（例如 "**Let's think step by step**"）
    - 将训练集里的问题**分成不同簇**，选择**最接近每个簇质心的问题**，就可以代表整个训练集里的问题。
- **增强的思维链策略**：如何生成多个推理路径，并在得到的答案中寻找一致性
  - **self-consistency**：Self-consistency improves chain of thought reasoning in language models，在生成 CoT 和最终答案时新的**解码策略**。先**用 LLM 生成多个推理路径**，再对所有答案进行**集成** (例如投票)。
  - **更通用的集成框架**：Rationale-Augmented Ensembles in Language Models发现多样化的推理路径是 COT 推理性能提高的关键，因此将 self-consistency 延伸至**提示的集成**。
  - 通过**训练打分模型**来**衡量生成的推理路径的可靠性**，如On the advance of making language models better reasoners

- **持续地利用 LLM 自己生成的推理路径**进行训练，如Star: Self-taught reasoner bootstrapping reason- ing with reasoning和Large language models can self-improve

4.2.1.2 零样本思维链

不在提示中加入人工标注的示范，而是直接生成推理步骤，再利用生成的 CoT 来得出答案。Large language models are zero-shot reasoners。

- 先通过 "**Let's think step by step**" 来提示 LLM 生成步骤
- 再通过 "**Therefore, the answer is**" 来提示得到最终答案

这种方法在**模型规模超过一定大小**时可以**显著提高性能**，但在**小规模的模型**中**效果不佳**，即涌现能力。

Flan-T5 和 Flan-PaLM（Scaling instruction-finetuned language models）进一步地使用 CoT 进行指令调整，有效增强了在**未见任务**上的零样本性能。

4.2.2 进一步讨论 CoT

- 思维链何时适用于 LLM:
- LLM 为何能进行思维链推理:
  - 思维链能力的来源:
  - 提示中组成部分的影响:

5 能力评测

5.1 基础评测

5.1.1 语言生成

5.1.1.1 语言建模

5.1.1.2 条件文本生成

5.1.1.3 代码合成

5.1.1.4 存在问题

- 可控生成
- 专业化生成

5.1.2 知识利用

5.1.2.1 闭卷问答

5.1.2.2 开卷问答

5.1.2.3 知识补全

5.1.2.4 存在问题

- 幻觉（Hallucination）
- 知识实时性

5.1.3 复杂推理

5.1.3.1 知识推理

5.1.3.2 符号推理

5.1.3.3 数学推理

5.1.3.4 存在问题

- 不一致性
- 数值计算

5.2 高级评估

5.2.1 与人类对齐

5.2.2 与外部环境互动

5.2.3 工具使用

5.3 公开基准

- MMLU：
- BIG-bench：
- HELM：

5.4 比较有用的数据集

https://huggingface.co/BelleGroup 里有很多中文的 instruct 和输出的数据集

6 RLHF & instructGPT

OpenAI 魔改大模型，参数减少 100 倍！13 亿参数 InstructGPT 碾压 GPT-3

https://openai.com/blog/deep-reinforcement-learning-from-human-preferences/

Training language models to follow instructions with human feedback

https://huggingface.co/blog/zh/rlhf

- 预训练一个语言模型 (LM)；
- 聚合问答数据并训练一个奖励模型 (Reward Model，RM)，也叫偏好模型；
- 用强化学习 (RL) 方式微调 LM。

6.1 sft

确保任务多样性的情况下，由标注人员编写 prompt 和一些生成式任务的期望输出。

- openai：instructGPT 使用小版本的 GPT-3，并对“更可取”（preferable）的人工生成文本微调
- Anthropic：1000w-520 亿参数的 transformer，并按“有用、诚实和无害”的标准在上下文线索上蒸馏原始 LM
- DeepMind：在Teaching language models to support answers with verified quotes提出的 GopherCite 模型中，用的是 2800 亿的模型 Gopher(Scaling language models: Methods, analysis & insights from training gopher)

6.2 rm

接收一系列文本并返回一个**标量奖励**，数值上对应人的偏好。我们可以用端到端的方式用 LM 建模，或者用模块化的系统建模 (比如**对输出进行排名**，再**将排名转换为奖励**) 。

- **模型选择**：RM 可以是另一个经过微调的 LM，也可以是根据偏好数据从头开始训练的 LM。Anthropic 提出了一种特殊的预训练方式，即用**偏好模型预训练** (Preference Model Pretraining，PMP) 来替换一般预训练后的微调过程，PMP **对样本的利用率更高**。
- **训练文本**：RM 的提示 - 生成对文本是从预定义数据集中采样生成的，并用初始的 LM 给这些提示生成文本。Anthropic 的数据主要是通过 Amazon Mechanical Turk 上的聊天工具生成的，并在 Hub 上可用，而 OpenAI 使用了用户提交给 GPT API 的 prompt。
- **训练奖励数值**：人工对 LM 生成的回答进行**排名**。起初我们可能会认为应该直接对文本标注分数来训练 RM，但是由于标注者的价值观不同导致这些分数未经过校准并且充满噪音，通过排名可以**比较多个模型各自的输出**并构建更好的规范数据集，这些不同的排名结果将被**归一化**为用于训练的标量奖励值。

目前成功的 RLHF 使用了和**要对齐的 LM** 具有**不同大小**的 LM：

- OpenAI：175B 的 LM 和 6B 的 RM
- Anthropic：使用的 LM 和 RM 从 10B 到 52B 大小不等
- DeepMind：使用了 70B 的 Chinchilla 模型分别作为 LM 和 RM

6.3 rl

直接微调整个 10B~100B+ 参数的成本过高，参考低秩自适应LoRA和 DeepMind 的Sparrow LM。目前多个组织找到的可行方案是使用策略梯度强化学习 (Policy Gradient RL) 算法、近端策略优化 (Proximal Policy Optimization，PPO) **微调初始 LM 的部分或全部参数**。

- 策略 (policy)：一个接受提示并返回一系列文本 (或文本的概率分布) 的 LM
- 行动空间（action space)：LM 的词表对应的所有词元 (一般在 50k 数量级)
- 观察空间 (observation space)：是可能的输入词元序列，也比较大 (词汇量 ^ 输入标记的数量)
- 奖励函数：偏好模型和策略转变约束 (Policy shift constraint) 的结合。

ppo 确定的奖励函数如下：

- 提示 x 输入初始 LM 和当前微调的 LM，分别得到输出文本 y_1 和 y_2
- 将来自当前策略的文本传给 RM 得到标量奖励 r_θ
- 将两个模型的生成文本进行比较计算差异的惩罚项，一般是输出词分布间的 KL 散度的缩放，即 $r = r_\theta - \lambda r_{KL}$，

惩罚项的好处：+ 用于惩罚策略在每个训练 batch 中生成大幅偏离初始模型，以确保模型输出合理连贯的文本。+ 如果没有这一项，可能导致模型在优化中生成乱码文本，以愚弄奖励模型提供高奖励值。

根据 PPO，按当前 batch 的奖励进行优化。PPO 是置信域优化（TRO，Trust Region Optimization）算法，用梯度约束确保更新步骤不会破坏学习过程的稳定性。

DeepMind 对 Gopher 用了类似的奖励设置，但用的是 A2C 来优化梯度。

6.3.1 rl 流程概述

https://zhuanlan.zhihu.com/p/635757674

Fine-Tuning Language Models from Human Preferences

Secrets of RLHF in Large Language Models Part I: PPO

- Rollout and Evaluation：从 prompt 库里抽样，使用语言模型生成 response，然后使用奖励模型（Reward Model, RM）给出奖励得分。这个得分反映了生成的 response 的质量，比如它是否符合人类的偏好，是否符合任务的要求等。
- Make experience：收集了一系列的“经验”，即模型的行为和对应的奖励。这些经验包括了模型生成的 response 以及对应的奖励得分。这些经验将被用于下一步的优化过程。
- Optimization：使用收集到的经验来更新模型的参数。具体来说，我们使用 PPO 算法来调整模型的参数，使得模型生成的 response 的奖励得分能够增加。PPO 算法的一个关键特性是它尝试保持模型的行为不会发生太大的改变，这有助于保证模型的稳定性。

官方代码 example

```
from tqdm import tqdm

for epoch, batch in tqdm(enumerate(ppo_trainer.dataloader)):
    query_tensors = batch["input_ids"]

    #### Get response from SFTModel
    response_tensors = ppo_trainer.generate(query_tensors, **generation_kwargs)
    batch["response"] = [tokenizer.decode(r.squeeze()) for r in response_tensors]

    #### Compute reward score
    texts = [q + r for q, r in zip(batch["query"], batch["response"])]
    pipe_outputs = reward_model(texts)
    rewards = [torch.tensor(output[1]["score"]) for output in pipe_outputs]

    #### Run PPO step
    stats = ppo_trainer.step(query_tensors, response_tensors, rewards)
    ppo_trainer.log_stats(stats, batch, rewards)

#### Save model
ppo_trainer.save_model("my_ppo_model")
```

Figure 1: PPO workflow, depicting the sequential steps in the algorithm's execution. The process begins with sampling from the environment, followed by the application of GAE for improved advantage approximation. The diagram then illustrates the computation of various loss functions employed in PPO, signifying the iterative nature of the learning process and the policy updates derived from these losses.

- Rollout：根据策略（LM）生成轨迹（文本）。
  - 输入：Batch Prompt、LM
  - 输出：Prompt+Response
- Evaluate：对生成的轨迹进行评估（RM）。
  - 输入：Prompt+Response、RM
  - 输出：Reward
- Old Policy Sampling：计算并存储旧策略的概率、价值等值，
  - 输入：Ref_model、Actor、Critic、Prompt+Response
  - 输出：Ref Logprobs、Old Logprobs、Old Values
- KL Penalty：计算**当前策略**和**原始 LM** 之间的 KL 散度，用作对策略改变过快的惩罚项。
  - 输入：Ref Logprobs、Old Logprobs、Reward
  - 输出：Token Reward
- Generalized Advantage Estimation (GAE)：G。基于 old value(shape 是 (`batch_size`, `response_length`)) 和 reward 估计优势函数 A，它结合了所有可能的 n-step 进行 advantage 估计
  - 输入：Token Reward、Old Values
  - 输出：Advantages、Returns
- New Policy Sampling：
  - 输入 ref_model、actor、critic，从新的策略中采样概率等信息，
  - 输出 new logprobs、new values 和 logits，供 actor loss、critic loss 以及 entropy loss 用。
- Critic Loss：Critic 的目标是估计状态的价值函数，Critic loss 就是价值函数预测值和实际回报之间的差距。
  - 输入：New Values、Returns
  - 输出：critic 梯度更新
- Actor Loss：Actor 的目标是优化策略，Actor loss 就是基于优势函数的策略梯度。
  - 输入：Old Logprobs，New Logprobs、Advantages
  - 输出：actor 梯度更新
- Entropy Loss：为了增加探索性，通常会添加一个基于策略熵的正则项，它鼓励策略保持多样性。
  - 输入：Logits
  - 输出：entropy loss
- Policykl：这是对策略迭代过程的一个度量，它度量**新策略**和**旧策略**之间的差距。
  - 输入：Old Logprobs、New Logprobs
  - 输出：是否 early stop

在 PPO 中，策略优化的过程涉及到两个策略：一个是“旧的”策略，这是我们在开始每次优化迭代时使用的策略，另一个是“新的”策略，这是我们在优化过程中**不断更新**的策略。

自己整理重画的

6.3.2 几个重要的 loss

6.3.2.1 actor & actor loss

Actor 是**策略**，它决定文本会被怎么样生成，是从**策略网络**拷贝来的模拟整个智能体在环境中行动的网络。

优势函数表示在给定的状态下采取某个行动比遵循当前策略的期望回报要好多少。

Actor Loss 如下，用重要性采样比较在**旧策略**和**新策略**下行动的概率（Old Logprobs，New Logprobs），然后将这个比值（也就是 Importance Sampling 的权重）与**优势函数 Advantages** 相乘，得到了对 Actor Loss 的一个估计。

$$L = \pi_{new}/\pi_{old} * A$$

```
# 计算新旧策略下概率的比值
ratio = torch.exp(logprobs - old_logprobs)

# 计算未截断的策略梯度损失
pg_losses = -advantages * ratio

# 计算截断的策略梯度损失
pg_losses2 = -advantages * torch.clamp(ratio, 1.0 - self.config.cliprange,
     1.0 + self.config.cliprange)

# 选择两者中较大的作为最终的策略梯度损失
pg_loss = masked_mean(torch.max(pg_losses, pg_losses2), mask)

# 计算因为截断导致策略梯度损失改变的比例
pg_clipfrac = masked_mean(torch.gt(pg_losses2, pg_losses).double(), mask)
```

6.3.2.2 critic & critic loss

critic 是专门用来预测 actor 轨迹**每一步价值**的网络，actor 上加几个线性层能够给每个 token 预测一个值。任务是估计状态的价值函数，也就是预测从当前状态开始，通过遵循某个策略，期望能得到的总回报。

Critic Loss 是最小化它的预测价值与实际回报之间的差距，常用 mse

通过最小化 Critic Loss，Critic 的预测能力会逐渐提升。因为 Critic 的预测结果会被用来**估计每个行动的优势（Advantage）**，这个优势值又会被用来计算策略的更新（Actor Loss）。

```
# 将价值函数的预测值裁剪到一个范围内
vpredclipped = clip_by_value(
            vpreds, values - self.config.cliprange_value, values + self.config.cliprange_value
        )

# 计算裁剪前和裁剪后的价值函数损失
vf_losses1 = (vpreds - returns) ** 2
vf_losses2 = (vpredclipped - returns) ** 2

# 最终的价值函数损失是裁剪前和裁剪后损失的最大值的平均值的一半
vf_loss = 0.5 * masked_mean(torch.max(vf_losses1, vf_losses2), mask)

# 计算裁剪操作实际发生的频率
vf_clipfrac = masked_mean(torch.gt(vf_losses2, vf_losses1).double(), mask)
```

6.3.2.3 KL Penalty

用于保证经过强化学习后的模型（新策略 actor）不会过于偏离原始预训练模型（ref model）。

```
# 初始化两个列表来分别存储奖励和非得分奖励
rewards, non_score_rewards = [], []

# 使用 zip 函数并行遍历输入的得分、对数概率、参考模型的对数概率以及 mask
for score, logprob, ref_logprob, mask in zip(scores, logprobs,
        ref_logprobs, masks):
    # 计算 KL 散度，即模型的对数概率与参考模型的对数概率之间的差值
    kl = logprob - ref_logprob

    # 计算非得分奖励，即 KL 散度乘以 KL 控制器值的负值
    non_score_reward = -self.kl_ctl.value * kl
    non_score_rewards.append(non_score_reward)

    # 复制非得分奖励为新的奖励
    reward = non_score_reward.clone()

    # 找到 mask 中最后一个非零元素的索引，这表示输入序列的实际长度
    last_non_masked_index = mask.nonzero()[-1]

    # 对于最后一个非 mask 部分的 token，其奖励是偏好模型的得分加上 KL 散度
    reward[last_non_masked_index] += score

    # 将计算的奖励添加到奖励列表中
    rewards.append(reward)

# 返回包含所有奖励的张量以及包含所有非得分奖励的张量
return torch.stack(rewards), torch.stack(non_score_rewards)
```

6.3.2.4 GAE

GAE 是一种多步优势估计方法。它通过引入一个权衡参数 λ，在**单步 TD 误差**和**多步 TD 误差**之间进行权衡，从而**减小估计的方差**，提高学习的稳定性。其中 σ_{t+l} 是时间步 $t+l$ 的 TD 误差。

$$A_t = \sum_{l=0}^{k-1} (\lambda\eta)^l \sigma_{t+l}$$

$$\sigma_{t+l} = r_{t+l+1} + \eta V(s_{t+l+1}) - V(s_{t+l})$$

```
# 从后往前遍历整个生成的序列
for t in reversed(range(gen_len)):
    # 计算下一个状态的价值，如果当前状态已经是最后一个状态，则下一个状态的价值为 0
    nextvalues = values[:, t + 1] if t < gen_len - 1 else 0.0

    # 计算 delta，它是奖励加上衰减后的下一个状态的价值，然后减去当前状态的价值
    delta = rewards[:, t] + self.config.gamma * nextvalues - values[:, t]

    # 使用 delta 更新 lastgaelam，这是 GAE 公式的一部分
    lastgaelam = delta + self.config.gamma * self.config.lam * lastgaelam

    # 将计算的优势值添加到优势值列表中
    advantages_reversed.append(lastgaelam)

# 将优势值列表反向并转换为张量
advantages = torch.stack(advantages_reversed[::-1]).transpose(0, 1)

# 计算回报值，它是优势值加上状态值
returns = advantages + values
```

6.3.2.5 entropy loss

一个策略的熵越大，意味着这个策略选择各个动作的概率更加“平均”。在 actor 的 loss 里加熵，使得策略的熵尽可能大，从而有更多机会探索可能带来更好奖励的文本轨迹。

```
entropy = -torch.sum(logits* torch.log(logits + 1e-9), dim=-1).mean()
```

新实现：

```
    pd = torch.nn.functional.softmax(logits, dim=-1)
    entropy = torch.logsumexp(logits, axis=-1) - torch.sum(pd * logits, axis=-1)
```

6.3.2.6 Policy kl

在 PPO 中，KL 散度被用作一种约束，以确保在优化过程中新策略不会偏离旧策略太远。这是为了防止过度优化，因为过度优化可能会导致策略性能的大幅下降。

我们希望在优化目标函数的同时，满足以下的 KL 散度约束：

$$KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \leq \delta$$

在代码中，每个 mini batch 都会进行 early stop 的判定，如果计算出的 KL 散度大于 δ，那么就会停止这一轮的优化，以保证新策略不会偏离旧策略太远。

```
# 计算旧策略和新策略之间的 KL 散度
policykl = masked_mean(old_logprobs - logprobs, mask)
# old_logprobs 是旧策略下行为的概率的对数，logprobs 是新策略下的对数概率
# masked_mean 函数计算差异（old_logprobs - logprobs）的平均值，
# 但只考虑 mask 中对应元素为 True 的元素

# 检查计算出的 KL 散度（policykl）是否大于目标 KL 散度（self.config.target_kl）的 1.5 倍
if policykl > 1.5 * self.config.target_kl:
    self.optimizer.zero_grad()
```

```
    # 如果实际的 KL 散度超过了目标的 1.5 倍，那么策略改变过多，这步的梯度也不更新了。
    early_stop = True
    # 并设置 early_stop 标志为 True，表示应提前停止优化，以防止策略从旧策略进一步偏离
```

6.3.3 两个采样

6.3.3.1 Old Policy Sampling（无 bp）

是 **make experience** 的过程，计算并**存储**旧策略的概率、价值等值，来为后面更新的过程服务。

- Old Logprobs：从 "旧的" 策略 [即在这个 batch 数据中初始的 LM（initial actor）] 中计算每个 token 在旧的策略下的概率 Old Logprobs。
- Old Values：旧策略中每个**时间步**（每个 token 的预测结果）的价值，这个值由 critic 网络进行预测，critic 网络就是需要这个值的原因是 advantage 的计算依赖于 Old Values。
- Ref Logprobs：最最原始的 LM 对于每个时间步的概率预测，一般就是**固定不变的 gpt3**，计算这个值的目的是限制 actor 的更新，防止其偏离原始 gpt3 太远，他的实现在下一个步骤中。

```
all_logprobs, _, values, masks = self.batched_forward_pass(self.model, queries,
    responses, model_inputs)
ref_logprobs, _, _, _ = self.batched_forward_pass(self.ref_model, queries,
    responses, model_inputs)
```

6.3.3.2 New Policy Sampling（有 bp）

在**新的策略**（更新后的 actor）下对轨迹（文本）计算概率的过程，计算 Actor Loss，即策略梯度的损失。

Old Logprobs 是一次性一个 batch 的数据计算的，这是因为在一个 batch 中旧策略都是不变的；而 New Logprobs 是一个 mini batch 计算一次，这是因为新策略每个 mini batch 变一次。

6.3.4 开源 rlhf 库

6.3.4.1 openai 的 lm-human-preferences(gpt2 的 finetune)

https://github.com/openai/lm-human-preferences

6.3.4.2 huggingface 的 TRL

https://github.com/huggingface/trl

6.3.4.3 CarperAI 的 trlx

https://github.com/CarperAI/trlx

6.3.4.4 allenai 的 RL4LMs

https://github.com/allenai/RL4LMs

7 llama 系列

7.1 llama

LLaMA: Open and Efficient Foundation Language Models

参考代码：https://github.com/huggingface/transformers/blob/main/src/transformers/models/llama/modeling_llama.py

之前的工作考虑的是在训练预算有限的前提下，如何提升模型性能（2022 年 deepmind 的Training Compute-Optimal Large Language Models的 Chinchilla）,llama 考虑在预测时的预算。例如 chinchilla 是一个 10b 的模型在 200b 的 token 上训练，但其实一个 7b 的模型当用了 1T 的 token 后，性能仍在提升。LLama-13b 比 gpt3 在大多数 benchmark 上好，但 size 只有 1/10，在一个 GPU 上就能跑。

llama 只用公开数据训练，而 Chinchilla、PaLM、GPT-3 都有自己的未公开数据集。其他的 OPT、GPT-NeoX、BLOOM、GLM 虽然也只用公开数据集，但打不过 PaLM-62B 或者 Chinchilla

7.1.1 预训练数据

- English CommonCrawl(67%)：使用 CCNet pipeline，去重、用 fasttext 把非英文的页面删了，用 n-gram 把低质内容删了。此外，还训了一个线性模型，对页面进行分类：作为维基百科的引用 vs 随机采样的页面，最后把不属于引用这个类别的页面删了
- C4(15%)：与 CCNet 类似，主要区别在质量过滤是基于启发式的规则，如标点符号的存在，或者词数和句子数
- github(4.5%)：使用 Google BigQuery 里的公开 github 数据集，只用 Apache、BSD 和 MIT 证书的。低质判断是启发式规则，如字母数字占比、行的长度等，用正则删掉 head 等样式，最终以文件粒度进行去重。
- wikipedia(4.5%)：2022 年 6-8 月的数据，包括 20 种语言
- Gutenberg and Books3(4.5%)：两个书籍数据集，对有 90% 以上内容重复的书籍做去重。
- Arxiv(2.5%)：拿原始的 tex 文件，删掉 first section 之前的东西，还有一些注释、宏
- Stack Exchange(2%)：高质量的问答网站，按答案的分数排序

Dataset	Sampling prop.	Epochs	Disk size
CommonCrawl	67.0%	1.10	3.3 TB
C4	15.0%	1.06	783 GB
Github	4.5%	0.64	328 GB
Wikipedia	4.5%	2.45	83 GB
Books	4.5%	2.23	85 GB
ArXiv	2.5%	1.06	92 GB
StackExchange	2.0%	1.03	78 GB

tokenizer：BPE，使用 sentencepiece 的实现。将所有 numbers 切成单个数字，回退到字节去处理未知的 utf8 字符（fallback to bytes to decompose unknown UTF-8 characters）

总共有 1.4T 的 token，对大部分训练数据，每个 token 在训练时只用了一次，除了维基和 book 大概用了两次。

附：gpt4 说：当我们说“一个 token 只训练一次”，我们其实是在说在一个 epoch（一个完整遍历训练集的过程）中，我们只遍历一次完整的数据集。如果一个特定的 token 在数据集中出现多次，那么在一个 epoch 中，这个 token 就会被用来训练模型多次。

params	dimension	n heads	n layers	learning rate	batch size	n tokens
6.7B	4096	32	32	$3.0e^{-4}$	4M	1.0T
13.0B	5120	40	40	$3.0e^{-4}$	4M	1.0T
32.5B	6656	52	60	$1.5e^{-4}$	4M	1.4T
65.2B	8192	64	80	$1.5e^{-4}$	4M	1.4T

Table 2: **Model sizes, architectures, and optimization hyper-parameters.**

7.1.2 网络结构

- pre-normalization(gpt3)：提升训练**稳定性**，对每个子层的输入做 norm，而非输出。此外，使用的是 RMSNorm 函数 (Root mean square layer normalization)
- SwiGLU 激活函数 (PaLM)：Glu variants improve trans- former，把 PaLM 里的 $4d$ 改了 $2/34d$
- Rotary embeddings(GPTNeo)：删掉原来的绝对位置编码，加上 rotary positional embedding(RoPE)，网络的每一层都加，参考Roformer: En- hanced transformer with rotary position embedding

优化器：AdamW，cosine 学习率 schedule，最终学习率是最大学习率的 10%。0.1 的 weight decay 和 1.0 的 gradient cliping，使用 2000steps 的 warmup

7.1.3 训练加速

- 对 causal multi-head attention 加速：实现在http://github.com/facebookresearch/xformers中，降低内存使用和运行时间，参考self-attention does not need $o(n^2)$ memory，以及Flashattention: Fast and memory-efficient exact attention with io-awareness。思想是
  - 不存储 attention weights
  - 不计算被 mask 的 key/query 得分
- 减少 xxx：

7.1.4 衍生：Alpaca

Alpaca: A Strong, Replicable Instruction-Following Model

在 LLaMA 模型的基础上的一个著名的项目是 Stanford 的羊驼（Alpaca）模型，有 70 亿（7b）参数，**没有使用 RLHF**，而是使用**监督学习**的方法。其数据集是通过查询基于 GPT-3 的 text-davinci-003 模型的结果，得到的 52k 的指令-输出对（instruction-output pairs）。因此，Alpaca 本质上使用的是一种弱监督（weakly supervised）或以知识蒸馏（knowledge-distillation-flavored）为主的微调。可以理解为是『用 LLM 来训练 LLM』，或者称之为『用 AI 来训练 AI』。

7.2 llama2

Llama 2: Open Foundation and Fine-Tuned Chat Models

https://zhuanlan.zhihu.com/p/636784644

8 gemini 系列

8.1 Gemini 1.0

Gemini: a family of highly capable multimodal models

8.2 Gemini 1.5

谷歌 Gemini 1.5 深夜爆炸上线，史诗级多模态硬刚 GPT-5！最强 MoE 首破 100 万极限上下文纪录

Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context

8.3 gemma

https://blog.google/technology/developers/gemma-open-models/

Gemma: Open Models Based on Gemini Research and Technology

8.3.1 更大的上下文窗口

此前的 SOTA 模型能处理 **20 万 (200K)** 的 token，Gemini 1.5 能稳定处理 **100 万 (1M)** 的 token（极限为 **1000 万 (10M)** 的 token），能够处理 11 小时的音频、1 小时的视频、超过 3w 行的代码库、超过 70w 个单词

8.3.2 架构 & 训练方法

- moe:
  -

9 多智能体

https://zhuanlan.zhihu.com/p/656676717

10 一些其他比较重要的工作

10.1 几篇出现频率比较高的论文

Scaling instruction-finetuned language models 引用数 800+

How can we know what language models know? 引用数 800+

Chain of thought prompting elicits reasoning in large language models引用 1800+

10.2 Anthropic 的一些工作

Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback

Studying Large Language Model Generalization with Influence Functions

Measuring Faithfulness in Chain-of-Thought Reasoning

10.3 ChatGLM

ACL22 GLM: General Language Model Pretraining with Autoregressive Blank Infilling

iclr23 GLM-130B: An Open Bilingual Pre-trained Model

11 训练 & 预测架构

11.1 pathways

Pathways: Asynchronous Distributed Dataflow for ML

下载了，pdf

这个回答分析得不错 https://www.zhihu.com/question/524596983/answer/2420225275

11.1.1 Google 的大规模稀疏模型设计

DESIGNING EFFECTIVE SPARSE EXPERT MODELS

代码：https://github.com/tensorflow/mesh/blob/master/mesh_tensorflow/transformer/moe.py

11.2 megatron-lm

https://zhuanlan.zhihu.com/p/646406772

11.3 deepspeed

https://zhuanlan.zhihu.com/p/343570325

11.4 ray-llm

https://github.com/ray-project/ray/releases/tag/ray-2.4.0

11.5 medusa-llm

decoder 的并行化：https://zhuanlan.zhihu.com/p/368592551

https://sites.google.com/view/medusa-llm

用了 tree-attention

12 大模型的一些现象

12.1 重复生成

https://www.zhihu.com/question/616130636

https://mp.weixin.qq.com/s/cSwWapqFhxu9zafzPUeVEw

13 多模态大模型

【IEEE Fellow 何晓东 & 邓力】多模态智能论文综述：表示学习，信息融合与应用，259 篇文献带你了解 AI 热点技

Multimodal Intelligence: Representation Learning, Information Fusion, and Applications

BERT 在多模态领域中的应用

CV 领域：VisualBert, Unicoder-VL, VL-Bert, ViLBERT, LXMERT。

13.1 ViT&Swin-Transformer

SwinTransformer 与 Vit 细节总结

An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

对于一张 $224 \times 224 \times 3$ 的图像，假设每个 patch 是 16×16，那就分成 $\frac{224 \times 224}{16 \times 16} = 196$ 个 patch(即 $seq_length = 196$)，每个 patch 的维度是 $16 \times 16 \times 3 = 768$，最后加上 [CLS] 这个 token，就是 $seq_length = 197$。

13.2 stable diffusion

High-Resolution Image Synthesis with Latent Diffusion Models

- 输入图像，经过编码器得到 z，z 通过前向扩散不断加噪声得到 z_T（正向扩散）
- 输入条件，经过条件编码器 (原文是 BERT，到了 DALL-E2 就改成 CLIP 了) 得到 τ_θ
- z_T 在 τ_θ 的指导下不断去噪（反向扩散），得到新的 z，再通过解码器得到最终生成的图像

其中的正向扩散和反向扩散一般用 U-Net

代码库：https://github.com/CompVis/latent-diffusion/tree/main

粗略看了下代码，带 condition 的训练原理大概是训练语料中有图 + 文本（例如 imagenet 的 class_label，这里可以映射到一个 classid 也可以直接拿明文），然后 condition 和图片一起作为输入去训练 auto-eocnder 和 ldm

在`/latent-diffusion/ldm/data/imagenet.py` 这个代码里，把 class_label 加进来了

```
    def _load(self):
        with open(self.txt_filelist, "r") as f:
            self.relpaths = f.read().splitlines()
            l1 = len(self.relpaths)
            self.relpaths = self._filter_relpaths(self.relpaths)
            print("Removed {} files from filelist during filtering.".format(l1 - len(self.relpaths)))

        self.synsets = [p.split("/")[0] for p in self.relpaths]
        self.abspaths = [os.path.join(self.datadir, p) for p in self.relpaths]

        unique_synsets = np.unique(self.synsets)
        class_dict = dict((synset, i) for i, synset in enumerate(unique_synsets))
        if not self.keep_orig_class_label:
            self.class_labels = [class_dict[s] for s in self.synsets]
        else:
            self.class_labels = [self.synset2idx[s] for s in self.synsets]

        with open(self.human_dict, "r") as f:
            human_dict = f.read().splitlines()
            human_dict = dict(line.split(maxsplit=1) for line in human_dict)

        self.human_labels = [human_dict[s] for s in self.synsets]

        labels = {
            "relpath": np.array(self.relpaths),
            "synsets": np.array(self.synsets),
            "class_label": np.array(self.class_labels),
            "human_label": np.array(self.human_labels),
        }

        if self.process_images:
            self.size = retrieve(self.config, "size", default=256)
            self.data = ImagePaths(self.abspaths,
                                   labels=labels,
                                   size=self.size,
                                   random_crop=self.random_crop,
                                   )
        else:
            self.data = self.abspaths
```

13.3 DALL-E 系列

DALL-E3:

Improving Image Generation with Better Captions

现有的文本-> 图像模型面临的一个基本问题是：训练数据集中的文本-图像 pair 对中的**文本质量较差**。

- 学习一个图像文本生成器，可以生成详细、准确的图像描述
- 将此文本生成器应用到数据集以生成更详细的文本
- 在改进的数据集上训练文本 - 图像模型

13.4 PALM-E

PaLM-E: An Embodied Multimodal Language Model

13.5 Pika/Runway 等

13.6 sora

OpenAI 首个 AI 视频模型炸裂登场，彻底端掉行业饭碗！60 秒一镜到底惊人，世界模型真来了?

https://openai.com/sora

https://openai.com/research/video-generation-models-as-world-simulators

一锤降维！解密 OpenAI 超级视频模型 Sora 技术报告，虚拟世界涌现了

Sora 爆火 48 小时：杨立昆揭秘论文，参数量或仅 30 亿

整体感觉:

- latent diffusion 的隐空间
- vit 和 swin transformer 的 patch

13.6.1 现有方法

现有的视频生成方法大多只能用于少数分类的视频、比较短的视频，或者固定长度的视频。

- recurrent networks
  - 2015 年的Unsupervised learning of video representations using lstms
  - 2017 年的Recurrent environment simulators
  - 2018 年的World models
- generative adversarial networks
  - 2016 年的Generating videos with scene dynamics
  - 2018 年的Mocogan: Decomposing motion and content for video generation
  - 2019 年的Adversarial video generation on complex datasets
  - 2022 年的Generating long videos of dynamic scenes
- autoregressive transformers
  - 2021 年的Videogpt: Video generation using vq-vae and transformers
  - 2022 年的Nüwa: Visual synthesis pre-training for neural visual world creation
- diffusion models
  - 2022 年的Imagen video: High definition video generation with diffusion models
  - 2023 年的Align your latents: High-resolution video synthesis with latent diffusion models

前两类太古老了，sora 把后面两个（autogressive transformers 和 diffusion models）结合在一起了，而且能同时处理不同时长、分辨率的视频和图像

13.6.2 将视频转成 spacetime latent patches

13.6.2.1 Vivit

Vivit: A video vision transformer

整体受 ViT 的启发

Figure 1: We propose a pure-transformer architecture for video classification, inspired by the recent success of such models for images [18]. To effectively process a large number of spatio-temporal tokens, we develop several model variants which factorise different components of the transformer encoder over the spatial- and temporal-dimensions. As shown on the right, these factorisations correspond to different attention patterns over space and time.

先分 patch，再分别过时间的 transformer（temporal transformer）和空间的 transformer（spatial transformer）

Figure 2: Uniform frame sampling: We simply sample n_t frames, and embed each 2D frame independently following ViT [18].

Figure 3: Tubelet embedding. We extract and linearly embed non-overlapping tubelets that span the spatio-temporal input volume.

具体的分 patch 方式如上图

13.6.2.2 latent 空间上的 patch

参考 stable-diffusion，即High-Resolution Image Synthesis with Latent Diffusion Models，把 patch 切分改成在 latent 空间上进行

- 将视频映射成**隐空间** (latent space) 的表示
- 把隐空间的表示切分成 **spacetime patches**

其实类似的思想已经有不少工作了

- VideoGPT：Videogpt: Video generation using vq-vae and transformers，结合了 VQ-VAE，而且是自回归的 transformer
- magvit：MAGVIT: Masked Generative Video Transformer
- magvitv2：Language Model Beats Diffusion – Tokenizer is Key to Visual Generation，语言模型战胜扩散模型！谷歌提出 MAGVIT-v2：视频和图像生成上实现双 SOTA！

预估时，可以通过在一个合适大小的 grid 里排列随机初始化的 patches（we can control the size of generated videos by arranging randomly-initialized patches in an appropriately-sized grid.）来控制生成视频的大小。估计是参考了下面这篇：

论文参考了这个Patch n'Pack: NaViT, a Vision Transformer for any Aspect Ratio and Resolution，可以使下面提到的 DiT 适应各种分辨率/持续时间/宽高比。

13.6.3 Diffusion Transformer

Scalable diffusion models with transformers提出了 DiT，替换 stable diffusion 中的 u-net

Figure 3. **The Diffusion Transformer (DiT) architecture.** *Left:* We train conditional latent DiT models. The input latent is decomposed into patches and processed by several DiT blocks. *Right:* Details of our DiT blocks. We experiment with variants of standard transformer blocks that incorporate conditioning via adaptive layer norm, cross-attention and extra input tokens. Adaptive layer norm works best.

DiT=VAE 编码器 + ViT + DDPM + VAE 解码器

sora 是一个扩散模型，输入加了噪声的 patches，还可以加上一些如 text prompt 的条件，预测原本『干净』的 patches。

之前的做法大多将视频全裁成相同长度和大小的，例如 4s 的 256×256，sora 可以直接用原始视频

13.6.4 语言理解

参考 DALL-E3 (Improving Image Generation with Better Captions)，训练了一个 highly descriptive 的视频描述生成器，拿这个生成器给训练集中的所有视频重新生成描述，再拿来训练。

此外，还用上了 GPT，将用户输入的短的 prompt 改写成更长更详细的视频描述用于生成。

13.6.5 使用图像/视频作为 prompt

- **图像转动画**：可以让静止的图像动起来
- **扩展视频**：可以对视频进行扩展（extend），在时间轴上向前或者向后进行延展（比如同样是一个石头落地，能生成 4 个视频，每个视频里的石头从不同的地方飞过来，落在同一个地面上）
- **编辑视频**：输入视频和一个文本 prompt，能够对视频进行编辑，例如把场景从沙漠替换成树林，类似Sdedit: Guided image synthesis and editing with stochastic differential equations
- **连接视频**：输入两个看似毫不相关的视频，能通过很自然的方式把这两个视频衔接在一起

13.6.6 生成图像

图像就是一帧的视频，可以通过在时间范围为一帧的空间 grid 中排列高斯噪声 patches（arranging patches of Gaussian noise in a spatial grid with a temporal extent of one frame）来生成图像，同样能生成不同分辨率的图像，最多 2048×2048

13.6.7 涌现的模拟能力

- **3D 一致性**：随着镜头的移动，视频中的人物或物体在 3d 空间中能在移动中保持一致
- **Long-range coherence and object permanence（远程连贯性和物体持久性）**：sora 能对短期和长期依赖关系进行建模，例如：
  - 可以保留人物体，即使它们被遮挡或离开当前帧。
  - 可以在单个样本中生成同一角色的多个镜头，并在整个视频中保持其外观的不变
- **与世界交互**：例如画家可以在画布上留下新的笔触，并随着时间的推移而持续存在，人吃东西能留下齿痕
- **模拟数字世界**：可以同时通过基本策略控制《我的世界》中的玩家，同时以高保真度渲染世界及其动态，只需要在 prompt 里提到“我的世界”的标题就可以实现。

13.6.8 存在的问题

- 不能准确地模拟许多基本相互作用的物理过程，例如玻璃破碎。
- 其他交互（例如吃食物）并不总是会产生对象状态的正确变化，例如长时间样本中出现的不连贯性或对象的自发出现。

14 LLM 与推荐结合

14.1 综述

https://github.com/nancheng58/Awesome-LLM4RS-Papers

A Survey on Large Language Models for Recommendation How Can Recommender Systems Benefit from Large Language Models: A Survey Recommender Systems in the Era of Large Language Models (LLMs)

14.2 Recommender Systems with Generative Retrieval

Recommender Systems with Generative Retrieval

14.3 P5

Recommendation as Language Processing (RLP):A Unified Pretrain, Personalized Prompt & Predict Paradigm (P5)

14.4 llm vs ID

推荐系统范式之争，LLM vs. ID?

Exploring the Upper Limits of Text-Based Collaborative Filtering Using Large Language Models: Discoveries and Insights

知乎的讨论

SIGIR2023 | ID vs 模态：推荐系统 ID 范式有望被颠覆?

Where to Go Next for Recommender Systems? ID- vs. Modality-based Recommender Models Revisited

https://github.com/westlake-repl/IDvs.MoRec

对应的 ppt

15 其他

15.1 一些比较好的模型

15.1.1 文本匹配

大多是基于 sentence-bert 的，m3e-base 在电商语料上试过，效果不错

https://huggingface.co/moka-ai/m3e-base

https://huggingface.co/shibing624/text2vec-base-chinese

15.2 RETRO Transformer

参数量仅为 4%，性能媲美 GPT-3：开发者图解 DeepMind 的 RETRO

http://jalammar.github.io/illustrated-retrieval-transformer/

Improving language models by retrieving from trillions of tokens

15.3 WebGPT

WebGPT: Browser-assisted question-answering with human feedback

https://openai.com/blog/webgpt/

15.4 本地知识库

https://github.com/chatchat-space/Langchain-Chatchat

15.5 llm 应用合辑

- ChatGPT 聚合站：https://hokex.com
- 游戏生成站：https://latitude.io/
- 家庭作业辅助站：https://ontimeai.com/
- 文字转语音站：https://www.resemble.ai/
- 艺术作画站：https://starryai.com/
- logo 制作站：https://www.logoai.com/
- ai 写作站：https://www.getconch.ai/
- 音乐制作站：https://soundraw.io/
- 声音模拟站：https://fakeyou.com/
- 一句话生成一段视频：https://runwayml.com/
- 文字转语音：https://murf.ai/

15.6 nanogpt

简化版的 gpt，tiktoken：gpt2 中使用的开源分词工具，比 huggingface 的 tokenizer 快得多

```
import tiktoken
enc = tiktoken.get_encoding("gpt2")

# 字节对编码过程，我的输出是 [31373, 995]
encoding_res = enc.encode("hello world")
print(encoding_res)

# 字节对解码过程，解码结果：hello world
raw_text = enc.decode(encoding_res)
print(raw_text)
```

15.7 达摩院大模型技术交流

https://developer.aliyun.com/live/248332

ppt：链接 密码：5yyf