

人工智能的趋势与挑战



前言

- 当前AI已成为推动社会和经济转型的关键力量，但AI技术的应用与发展还面临诸多的挑战。
- 本章节将围绕AI的技术发展趋势、行业应用实践和AI应用的挑战三个维度展开。介绍AI应用的发展历程、AI应用的技术趋势、AI行业实践以及面临的挑战和问题。

目标

- 学完本课程后，您将能够：
 - 洞悉当前AI应用发展趋势；
 - 列举AI在科学、金融、政务等领域的应用；
 - 掌握AI应用落地面临的挑战。

目录

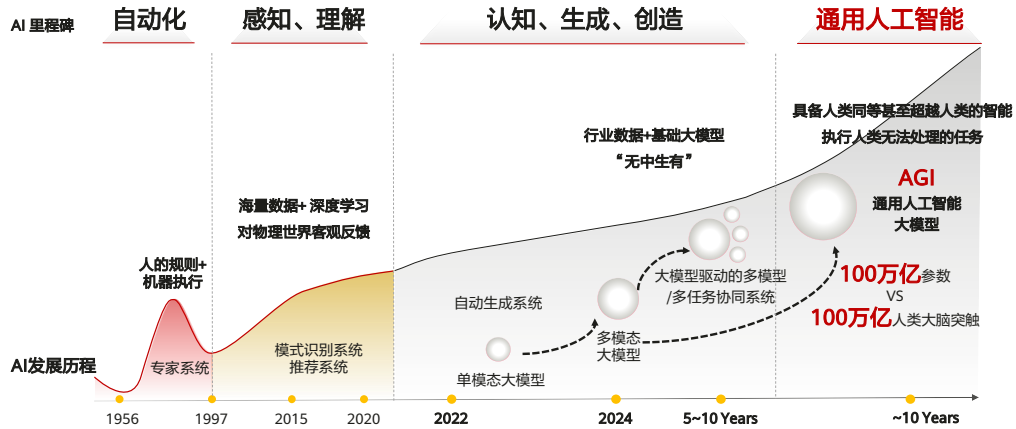
1. AI发展趋势

- AI发展历程
- 技术及应用趋势
- 开源大模型带来的思考

2. AI行业应用场景

3. AI发展面临的挑战

AI发展历程 - 从感知理解到创造生成



- AI的可以分为几个阶段：自动化、感知和理解，认知、生成、创造以及未来的通用人工智能。为什么现在的AI激起如此大的浪潮，根因是进入了生成和创造阶段，这是人类智慧真正的体现，动物能感知和理解，但只有人类能生成和创造。这就是跨越拐点的关键。第二个拐点是从专用人工智能到通用型人工智能，以前就是专用的AI，比如目标识别，有用但局限在一个狭小的领域，就好像扫地机器人或洗衣机，而通用型AI，具有泛化的智慧，能干各种各样的活，能作诗也能做事，上的了厅堂，下得了厨房，相比于扫地机器人、洗衣机，就好像有了一个保姆，能干各种各样的活，还能自主学习。这就形成了类似蒸汽机一样的普适的新生产力。这就是新生产力的跃升！

从感知到认知

① 感知智能，“听”与“看”清世界

能听会看

没有思想的工具

- 语音识别
- 目标检测
- 图像识别

可以处理音频、图像等数据，
没有独立规划与决策的能力

② 智能涌现，从感知走向认知

智能涌现+行业化 形成新质生产力

- AI4Science
- AI+ 医疗
- 智慧城市

具备认知与思考能力，结合行业化知识，极大提高生产力和生产效率

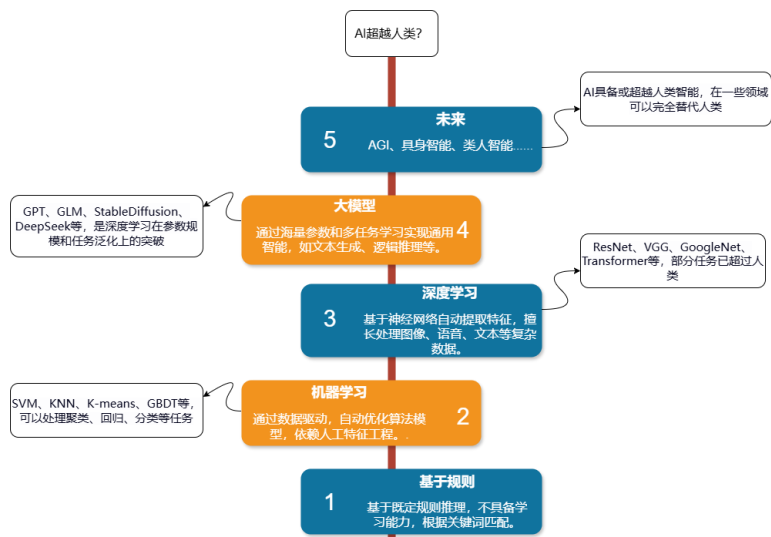
③ 智能体、具身智能，科幻走进现实

通用人工智能具象化

- 突破智能体的数字世界边界，解放和发展生产力
- 预计2030年形成万亿美元全球收入机会



学习算法



算力支撑 - 高算力密度设备



- 高算力密度硬件通过集成更多计算核心、优化芯片布局或采用先进封装技术（如Chiplet、3D堆叠），在有限体积内实现更强的计算能力。
- 硬件设计趋向紧凑，通过集成内存、I/O接口和计算单元，减少数据传输延迟。
- 模块化设计便于扩展和部署，适应不同场景需求。
- 为满足大规模并行计算需求，高算力密度硬件通常配备高速接口和片上网络（NoC），确保计算单元间、计算与存储间的高效数据交互，降低通信瓶颈。

数据支撑 - 大规模、多样性

从“小而精”到“大而全”

维度	传统深度学习小模型	大模型时代
数据量	数GB到数百GB	数TB到数十TB，甚至上百TB
数据来源	人工标注数据集为主	大规模无监督文本/多模态数据（网页、书籍、代码、社交媒体等）
数据标注	高度依赖人工标注（监督学习）	主要使用自监督学习，标注需求低
数据多样性	领域特定、任务特定	跨领域、跨语言、跨模态，强调泛化能力

- 早期模型依赖高质量但有限的标注数据；大模型则追求海量、多样化的原始数据，通过自监督学习挖掘知识。

目录

1. AI发展趋势

- AI发展历程
- 技术及应用趋势
- 开源大模型带来的思考

2. AI行业应用场景

3. AI发展面临的挑战

大模型推理时的问题 - 缺乏推理步骤

场景：

一位患者出现以下症状：持续高烧（39° C）、咳嗽伴有胸痛、呼吸急促、血氧饱和度下降
需要判断是否为肺炎或流感，并给出治疗建议。

问题

回答：“患者可能患有流感 建议服用奥司他韦。”

存在问题：

- 未生成中间推理步骤，无法验证逻辑合理性。
- 若模型误判为流感，而实际是肺炎，可能导致延误治疗。
- 无法解释为何得出该结论，缺乏可解释性。

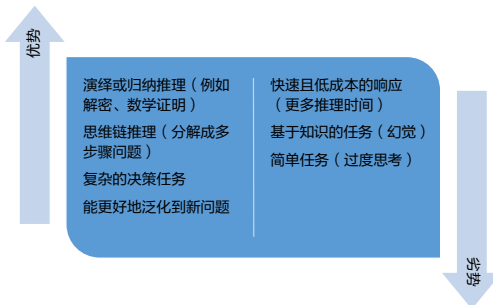
期望

会逐步分解问题，生成类似以下推理链：

- 第一步：分析症状 → 高烧、咳嗽、胸痛、呼吸急促提示肺部感染。
- 第二步：区分肺炎 vs 流感 → 肺炎更可能伴随胸痛和血氧下降，而流感多见于全身症状（如肌肉酸痛）。
- 第三步：建议检查 → 胸部X光、血常规、C反应蛋白（CRP）检测以确认诊断。
- 第四步：治疗建议 → 若确诊为肺炎，需抗生素治疗；若为流感，需抗病毒药物。

Reasoning Model

- Reasoning Model（推理模型）是指能够模拟人类逻辑推理过程的人工智能模型，通过多步骤的分析、推导和解释来解决问题，而非仅依赖模式匹配或直接输出结果。
- 这类模型的核心在于其可解释性和逻辑连贯性，尤其在需要深度思考、复杂决策或解决开放性问题的场景中表现出色。



大模型推理时的问题 - 幻觉、知识过时

幻觉

模型生成看似合理的错误答案

问：拿破仑发明了电灯吗？
LLM：是的，拿破仑在19世纪初期发明了电灯。

知识过时

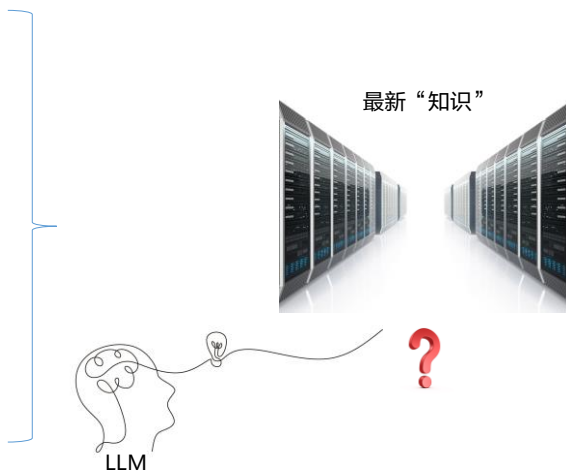
训练数据时效性不足

问：刚发布的GPT-5效果如何？
LLM：抱歉我不知道2023年以后的事情。

知识固化

大语言模型在训练后，其知识被“冻结”在参数中，无法获取训练数据之后的新信息（如实时数据、私有数据或特定领域知识）。

.....



RAG

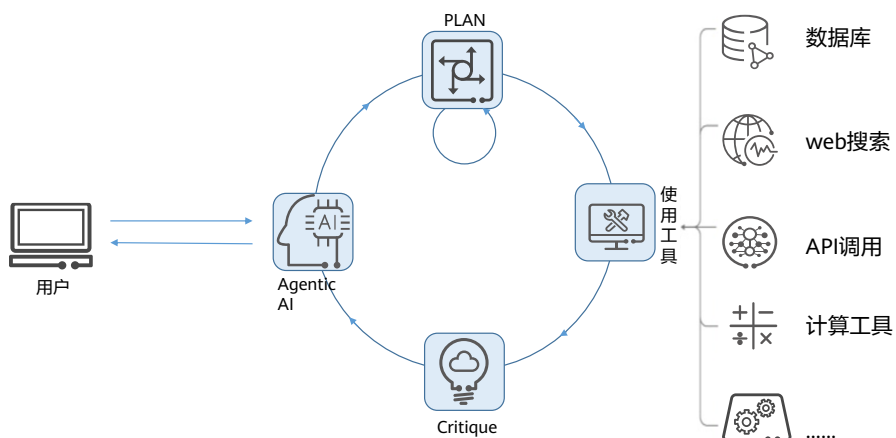
- RAG（Retrieval - Augmented Generation）检索增强生成。
 - 是一种将信息检索与文本生成相结合的人工智能技术。
 - 核心思想是在生成答案前，先从外部知识库中检索相关文档或信息，并将这些信息作为上下文输入生成模型，以提高生成内容的准确性、相关性和时效性。



大模型推理时的问题 - 灵活性、主动性

- 依赖预设规则或固定流程，难以应对需要动态调整的复杂任务。
- 在特定领域表现优异，但缺乏跨领域协作能力。
- 无持久化存储，依赖当前上下文。
- 难以快速适应新任务或环境变化。

智能体



- 1、用户提出请求：用户（或另一个智能体）启动了智能体工作流。智能体处理此输入，并将其分解为可操作的步骤。
- 2、LLM解读用户的输入，理解任务要求。
- 3、任务分解：将任务分解为具体的可执行操作。
- 4、上下文：保留上下文（长期记忆/短期记忆）。
- 5、执行任务：编排外部工具完成每个步骤或子任务。
- 6、改进和反思。

具身智能 (1)

具身智能 (Embodied AI)

1950年，图灵首次提出了机器像人一样能和环境交互感知，自主规划、决策、行动，并具备执行能力。

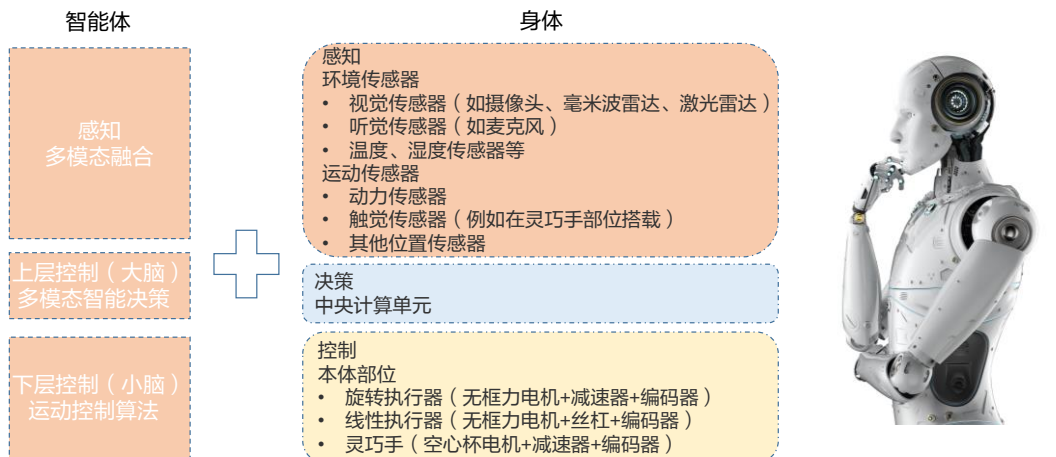


在接下来的15-20年里，**仿生机器人将颠覆全球经济每个主要部门数百个行业的人类劳动**。劳动力的颠覆将是人类历史上最深刻的变革之一，因此同时代表了人类文明有史以来面临的**最大机遇和最大挑战之一**。



- 具身的含义不是身体本身，而是与环境交互以及在环境中做事的整体需求和功能。

具身智能 (2)



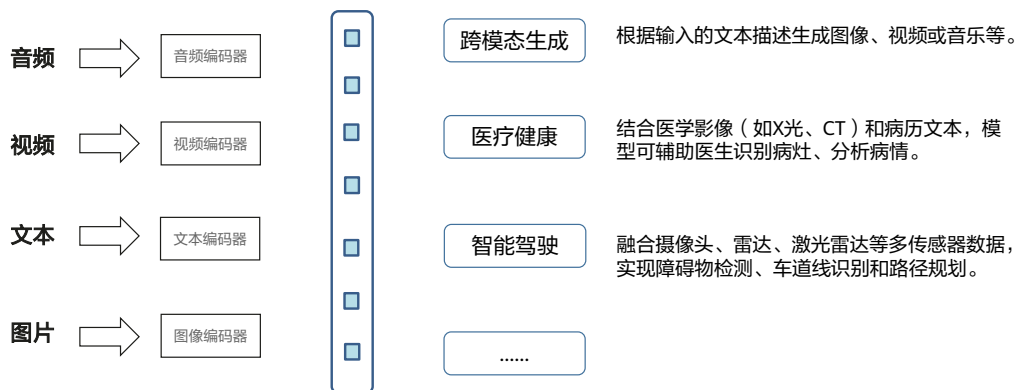
- 具身智能由“本体”与“智能体”构成，以“感知决策、物理实体、环境交互”为主要特征。具身“智能体”包括“感知、决策、控制”三个主要环节，并需要内外部通信传输的支持。
 - 感知：具身智能交互世界的窗口，通过传感器收集环境与自身状态信息，作为决策环节的依据。
 - 决策：根据感知的信息以及任务目标，对自身行为进行规划决策，并向控制模块发出指令。
 - 控制：将决策指令转化为实际操作，实现与物理世界的互动。

通用人工智能

- AGI是一种能够执行任何人类智能活动的计算机系统，具有全方位的智能，包括自我学习、自我改进、自我调整及跨领域、跨模态的学习、推理和决策能力。



多模态应用

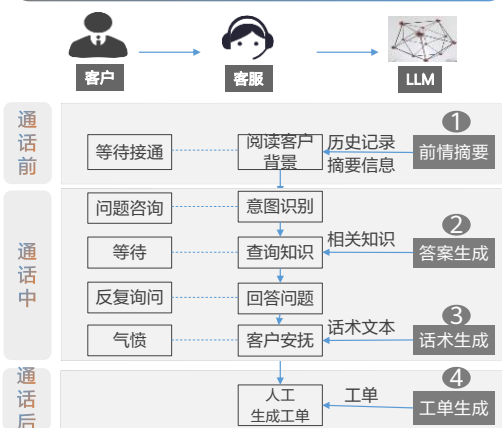


行业化AI应用

- AI技术从通用能力向垂直行业场景深度定制，解决特定领域痛点，形成差异化解决方案。
- 行业化是AI从技术到价值的必经之路，未来竞争将聚焦垂直场景的深度理解。



场景：大模型在智能客服四个环节发挥作用



目录

1. AI发展趋势

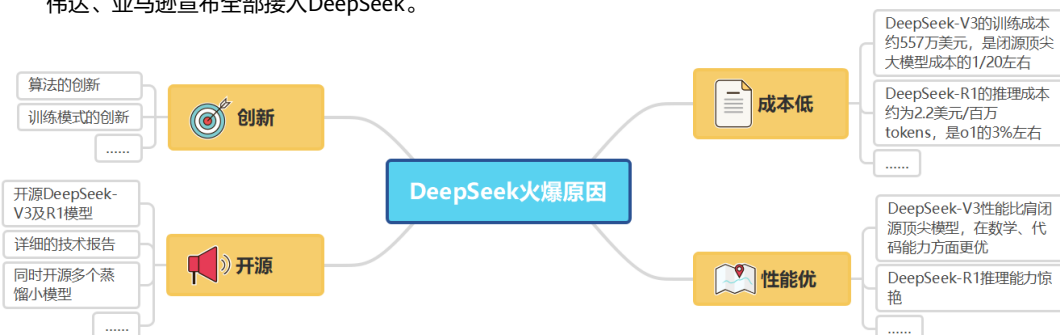
- AI发展历程
- 技术及应用趋势
- 开源大模型带来的思考

2. AI行业应用场景

3. AI发展面临的挑战

DeepSeek - 现象级应用

媒体2025年1月31日报道，DeepSeek的人工智能（AI）软件在全球140个市场中的移动应用下载量排行榜上位居榜首。同日，微软、英伟达、亚马逊宣布全部接入DeepSeek。



- 在性能上和世界目前顶尖的GPT-4o等大模型不相上下。但在成本上，OpenAI训练ChatGPT-4花费的成本高达7800万美元，还可能达到1亿美元。而DeepSeek-V3大模型训练成本仅557.6万美元（278.8万H800 GPU小时），仅为同性能模型的5%到10%。
- 算法：新的MoE架构，MLA，无辅助损失函数的负载均衡，MTP。
- 训练模式改变，并行方式的创新。

DeepSeek模型优势

低成本完美对标OpenAI O1，突破精确语义理解及复杂推理任务

数学+编程领域大幅领先，DeepSeek-V3训练成本不足Meta/OpenAI的10%，DeepSeek-R1推理成本仅为OpenAI o1的3%，成为开源模型SOTA。

低成本

挖掘FP8硬件潜力，MOE和MLA技术实现低成本方案

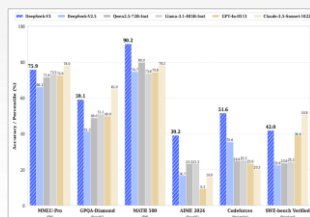
~150M\$
Llama3.1-405B
训练成本



5.57M\$
DeepSeek-V3
训练成本

性能优

数学、科学和代码等领域领先



技术创新

硬件级、算法级、架构级、工程级、开源生态5大技术创新，轰动全球

- 硬件级**
 - PTX编程
 - 计算与通信优化，性能提升30%
- 算法革命**
 - GRPO：群体进化的智慧筛选器
 - 自我验证机制：AI的“错题本系统”
- 架构创新**
 - 混合专家模型的“智能路由器”
 - 多头潜在注意力MLA：空间压缩术
- 工程奇迹**
 - 训练框架加速：通信降低89%
 - 推理加速：预加载，动态批处理等
- 开源生态**
 - 模型、数据、工具链、部署全开源
 - 蒸馏技术使第三方模型性能

- DeepSeek 大模型在技术创新和性能优化方面具有显著优势，能够以较低成本完美对标OpenAI O1，并能够实现精确语义理解，处理复杂的复杂推理任务。
- 数学和编程领域大幅领先，DeepSeek-V3的训练成本不足Meta/OpenAI的10%，DeepSeek-R1的推理成本仅为OpenAI的3%，成为开源模型中做的最好的模型之一。
- 更具体来说，DeepSeek模型优势主要体现在3点，分别为：更亲民的成本架构、更卓越的性能表现以及更前瞻的技术创新。
- 在低成本这一块，通过FP8硬件潜力深度挖掘，绕过传统CUDA架构桎梏，结合MOE架构与MLA注意力机制，成功将训练成本控制在竞品的1/10以下。最直观的数据就是与DeepSeek-V3性能相近的模型，其训练成本约为1.5亿美元，而DeepSeek-V3训练成本仅为557万美元。
- 在性能突破方面，DeepSeek在数学与编程领域已率先登顶，在MMLU-Pro14项核心测评中稳居榜首。
- 最后在技术创新层面，还有颠覆性的五重奏。
 - 硬件级算力优化：首创PTX底层编程，计算通信双优化带来30%性能跃升。
 - 算法革命：GPRO群体进化筛选器+AI错题本系统，构建持续进化飞轮。
 - 架构创新：混合专家模型的智能路由器，效率指标提升两数量级。
 - 工程奇迹：MLA多头潜在注意力机制，通信负载锐减89%。
 - 预加载革命+动态批处理技术，响应速度提升4.6倍。

DeepSeek对AI产业的影响

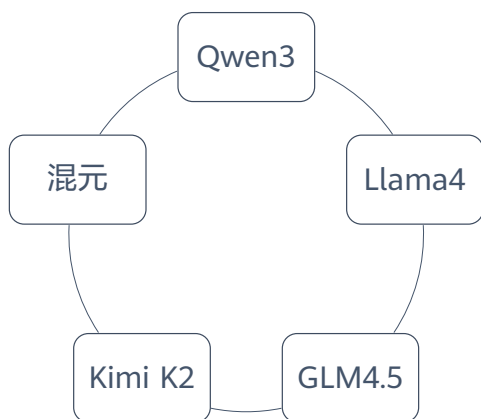


- 挑战巨头垄断地位：长期以来，AI领域被少数拥有大量资源和技术优势的巨头企业所主导。DeepSeek的出现，以其强大的技术实力和独特的开源策略，对现有巨头的垄断地位构成了挑战。它在短时间内取得的技术突破和广泛的市场应用，吸引了大量用户和开发者，打破了市场原有的平衡，促使巨头企业重新审视自身的技术优势和市场策略。
- 引发行业竞争加剧：DeepSeek的成功激发了更多企业和研究机构在AI领域的投入和创新。为了在竞争中占据优势，各大企业纷纷加大研发力度，推出更具竞争力的产品和技术。这种竞争不仅体现在技术性能上，还包括成本控制、应用场景拓展等多个方面。例如，在大语言模型市场，DeepSeek引发了价格战，其他企业不得不降低价格以保持竞争力，这使得整个行业的竞争更加激烈，也推动了技术的快速迭代和成本的降低。
- 开源促进技术共享与创新：DeepSeek采取开源策略，将其模型和技术向全球开发者开放。这一举措打破了AI领域长期存在的闭源模式，促进了全球范围内的技术共享与合作。开发者可以基于DeepSeek的开源模型进行二次开发和创新，加速了新技术的产生和应用。例如，许多开发者利用DeepSeek的模型开发出了针对特定行业的应用程序，推动了AI在各个领域的深入应用。
- 改变AI发展思路：在DeepSeek出现之前，AI发展往往依赖大量的资金投入、强大的算力支持和大规模的模型训练。而DeepSeek证明了，通过创新的算法和高效的训练方法，即使在有限的资源条件下，也能取得卓越的技术成果。这促使AI行业从单纯追求算力和模型规模，转向更加注重算法创新、资源优化和应用落地，为AI的发展开辟

了新的道路。

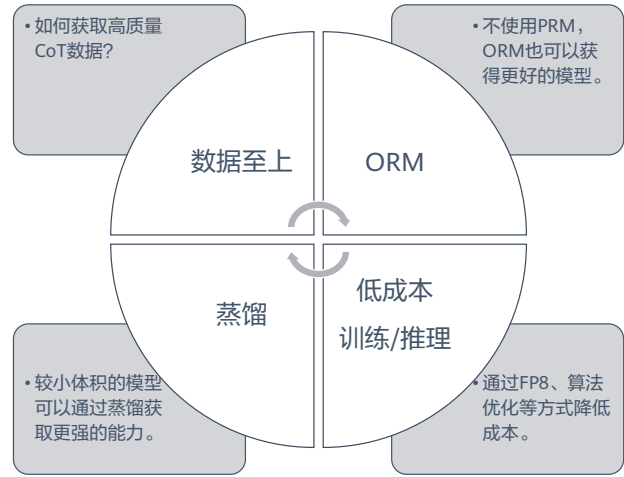
- 满足多样化应用需求：DeepSeek强大的语言理解和生成能力、数学推理能力以及代码处理能力，使其能够满足不同行业和领域的多样化应用需求。在金融领域，可用于风险评估、投资决策等；在医疗领域，辅助疾病诊断、药物研发等；在教育领域，实现个性化学习、智能辅导等。其广泛的应用场景为AI技术的普及和推广提供了有力支持。
- 降低应用开发门槛：开源的DeepSeek模型和相关技术，降低了企业和开发者进入AI领域的门槛。无需从头开始研发复杂的AI模型，开发者可以利用DeepSeek的技术快速构建应用程序，缩短了开发周期，降低了开发成本。这使得更多的中小企业和创新团队能够参与到AI应用的开发中，进一步拓展了AI市场的规模和范围。
- 推理算力重要性提升：未来大模型开发成本低，大家重心会转向推理，算力结构发生变化，推理算力受关注，且推理算力取决于用户需求。
- 国产替代机遇：DeepSeek开源后，众多芯片厂家为其做适配，全球芯片设计会考虑其架构变化，这对国产芯片设计公司利好，会带动生态发展，国产算力崛起迎来更多机会。

开源大模型



当前开源大模型在长文本、多语言、视觉理解等领域的突破，为开发者和企业提供了灵活且强大的 AI 解决方案，进一步推动了通用人工智能（AGI）的发展。

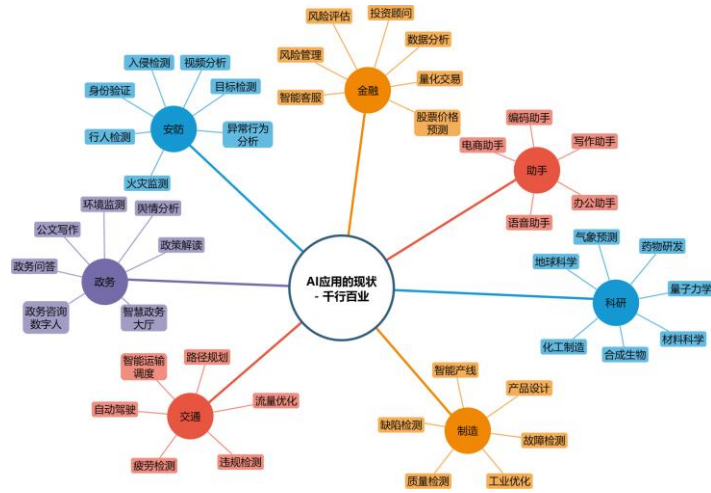
开源大模型带来的思考



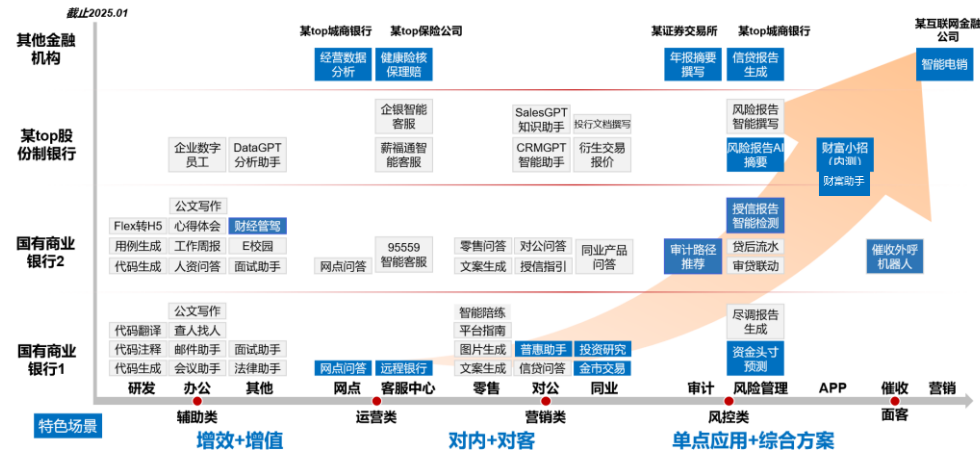
目录

1. AI发展趋势
- 2. AI行业应用场景**
3. AI发展面临的挑战

AI应用的现状 - 千行百业



智慧金融



智慧金融 - 智能摘要和报告写作

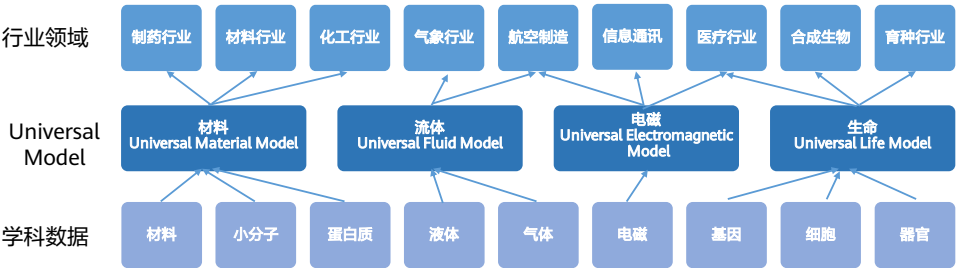
场景：2类内容生成输出模式



案例&客户价值：尽调/授信报告一键生成

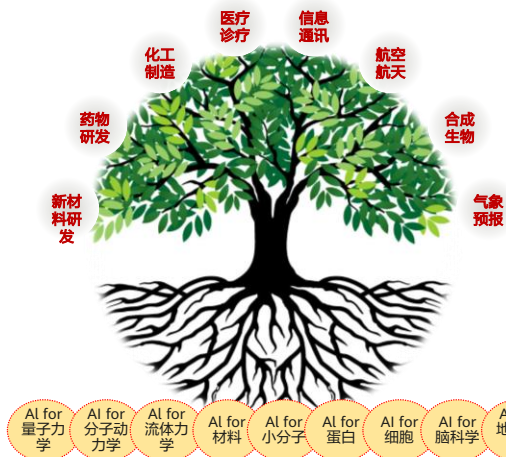


打通行业场景海量数据实现智能涌现



AI4Science 2035

AI4Science根技术的突破释放巨大的行业创新驱动动力



技术趋势

科学大模型到行业场景端到端打通，海量行业场景数据推动科学大模型实现涌现。

跨学科多尺度数据融合，聚焦到物质、流体、场、生命四大科学大模型的构建。

AI在更多学科如数学，量子力学，材料，流体，细胞，脑科学，地球科学等实现突破

行业趋势

- 2035**
1. AI细胞在临床试验，个性化医疗中广泛应用
 2. AI育种广泛应用
 3. AI准确预测极端天气
 4. AI广泛用于工业设计，化工优化，合成生物

- 2030**
1. AI设计材料在超导体、半导体，电池，特种合金等领域实现突破。
 2. AI设计细胞得到验证
 3. AI飞行器设计得到验证

- 2025**
1. AI药物完成临床上市
 2. 基于卫星的气象大模型实现分钟级全球预报
 3. AI材料设计得到验证，相图预测实现突破

AI4Science - AI + 气象

华为盘古气象大模型，发现和预测自然规律



AI4Science - AI + 药物

Pangu Drug盘古药物分子大模型，大幅提升药物设计效率

行业
痛点

1 药物发现周期长

2 人工实验成本高、耗时长

盘古药物分子大模型

向人类一样学习小分子化合物

ZINC
DrugSpaceX
UniChem
~17 亿 化合物

分子结构
编码器

[x1, x2, x3, ..., xn]

分子式解
码器

新颖化合物库生成

蛋白质靶点匹配

化合物属性预测

化合物结构优化

亿级参数

药物靶点

苗头化合物
候选库

BSAP, SIDER,
CinTox, BACE,
Tox21, ToxCast,
Lipophilicity,
PAINS, QM7,
QM8, ESOL

先导化
合物
候选库

Low
Lipid Solubility
High
Lipid Solubility
HO
HO
HO

筛选结果库

盘古药物分子大模型
产生1亿结构新颖化合物

化合物
数据库

化合物属性预测器

化合物优化器

1亿

崭新结构小分子库

100%

结构新颖性

20+

任务实现SOTA

80+

药物理化属性预测

抗菌药发现（联合西交大一附院）



数年
先导药研发周期

1个月

盘古政务大模型



政务智慧办公助手

- 盘古政务大模型基于20多个省市40多万篇公开公文进行训练，通过知识搜索与大模型结合，构建政务智慧办公助手，赋能政务办公一网协同领域。
 - 以公文撰写场景为例，盘古政务智慧办公助手实现了多任务演进，支持基于15种公文，基于提纲、角色权责、参考文献撰写，为每位公务员打造个人专属的智能办公助手，办公效率提高200%。



- 政企知识检索：基于政企知识进行模型训练，通过对话交互，精准获取用户需求的信息，提高知识获取效率，带来有温度的人工智能服务体验。



- 政企会议助手：根据流水账改写成标准会议议程，基于会议内容生成文本摘要等，为会议组织者、与会者提供高效便捷的会议体验。

个人助手

支持5类23种常用信息记忆，记忆衍生丰富使用场景



对话记



识图记



自动填槽



自动填表

- 作为一个好的AI助手，一个关键的能力就是得有记忆的能力，就像一个人一样，要能记住关键信息，这样在给用户提供各种AI服务的时候才能更完整，更高效；
- 一个非常普遍的例子，就是我们在生活中会遇到各种各样的填表，我们旅游，政务事情的办理，注册用户等等；
- 我们可能需要填自己的各种信息，还需要填写亲属的各种信息；我们一般记不住，身份证号码，护照号码等等，要到处去问，同时自己手动输入填进去也很麻烦；
- 小艺记忆这个特性就可以帮助我们解决这些问题，可以通过对话告诉小艺，或者通过识图的方式让小艺自己提取，比如身份照的照片等等；小艺帮我们记录下来，用的时候可以直接调用，比如让小艺导航到你记录的某个地址，填表时自动填某个信息等。

编码助手

ArkTS中，如何将一个很大的数保留两位小数，并且不使用科学计数法

参考来源 1, 扩展阅读 20

在ArkTS中处理大数保留两位小数且避免科学计数法，可以通过以下两种方式实现：

方法一：使用字符串操作（适用于超大数值）

```
typescript
function formatLargeNumber(num: string): string {
  const [integerPart, decimalPart = '00'] = num.split('.');
  const processedDecimal = decimalPart.padEnd(3, '0').slice(0, 3);

  // 处理四舍五入
  let rounded = processedDecimal.slice(0, 2);
  if (parseInt(processedDecimal) >= 5) {
    rounded = (parseInt(rounded) + 1).toString().padStart(2, '0');
  }

  return `${integerPart}.${rounded}`;
}

// 使用示例
const bigNumber = "123456789012345678901234567890.1267";
```

方法二：使用Intl.NumberFormat（适用于精度范围内数值）

```
typescript
```

使用ArkTS实现类似ios右滑解锁功能，效果图如左边一个图标，中间显示右滑解锁，拖动图标往右滑动，滑到80%的位置提示已经解锁，不到80%时自动返回原点，图标滑动过的区域显示红色，不要使用ext组件，可以使用stack和row组件。

扩展阅读 23

要实现类似iOS右滑解锁功能，结合ArkTS的布局和手势特性，可以按以下步骤实现（未使用Flex组件，基于Stack和Row）：

一、核心实现思路

1. 布局结构：使用 Stack 叠加背景层与滑动层，Row 控制横向滑动区域。
2. 手势交互：通过 PanGesture 监听滑动事件，动态计算图标位置和背景色。
3. 阈值判定：滑动超过80%宽度时触发解锁，否则回弹。
4. 动态效果：用 animateTo 实现平滑动画，@State 管理状态。

二、代码实现与关键逻辑

```
typescript
@Entry
@Component
struct SlideToUnlock {
  @State offsetX: number = 0; // 图标水平偏移量
  @State isUnlocked: boolean = false; // 解锁状态
  private maxSlideWidth: number = 0; // 最大滑动距离，屏幕宽度的80%
}
```

AI应用的发展趋势

- AI未来应用趋势预示着更广泛的技术整合、更深的行业渗透和持续的道德与安全考量。

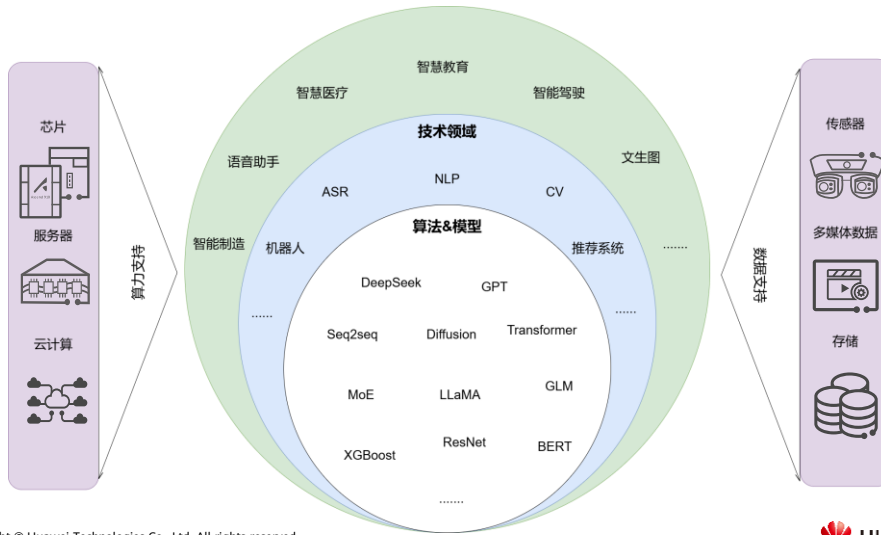


- 更广泛的行业应用：随着技术的成熟和适应性增强，预计大模型将被应用到更多传统行业中，如制造业、能源、物流等，提供高效的问题解决方案和决策支持。
- 多模态和跨模态学习：未来的大模型将不仅限于文本或图像单一领域，而是能够处理和整合多种类型的数据（如文本、图像、视频和声音），提供更全面的AI应用体验。
- 个性化和适应性增强：大模型将更好地适应个体用户的具体需求，提供定制化的服务。例如，在教育和健康领域，模型能够根据个人的学习进度或健康状况提供个性化建议。
- 自主学习能力的提升：未来的大模型将拥有更强的自学能力，能够在不断的交互和反馈中学习新的模式和知识，减少对大规模标注数据的依赖。
- 更严格的伦理和安全控制：随着大模型的普及，相关的伦理、隐私和安全问题将更受关注。未来的开发将需要侧重于确保模型的透明性、可解释性和公正性，同时加强对数据的保护。
- 开源和民主化的AI技术：大模型的开发和应用可能会更加民主化，更多的研究机构和公司可能会开源其模型和技术，促进全球范围内的技术共享和创新。
- 能源效率和可持续性：鉴于训练大规模模型所需的巨大能源消耗，未来的研究将侧重于提高模型的能效和减少碳足迹，以支持可持续发展目标。

目录

1. AI发展趋势
2. AI行业应用场景
- 3. AI发展面临的挑战**
 - 技术挑战
 - 伦理挑战

AI技术领域&应用领域概览



大模型训练对计算资源的需求

前向传递 + 后向传递 + 激活重计算的系数=1+2+1=4。
使用**激活重计算**的一次训练迭代中，对于每个token，每个模型参数，需要进行 $2 \times 4 = 8$ 次浮点数运算

$$\text{训练时间} \approx \frac{\text{系数} \times \text{token数} \times \text{模型参数量}}{\text{计算卡数量} \times \text{计算卡峰值FLOPS} \times \text{计算卡利用率}}$$

以GPT3 175B模型，300B tokens为例，
在1024张算力为312TFlops (A100) 的卡上 $\frac{8 \times (300 \times 10^9) \times (175 \times 10^9)}{1024 \times (312 \times 10^{12}) \times 0.45} \approx 2921340 \text{ seconds} \approx 34 \text{ days}$
训练一次所需时间为：

参数量P (B)	训练阶段	数据量T (B tokens)	卡数 n	训练时长 (天)
175 (e.g. GPT3)	预训练	3500 ^③	8192	49
	二次训练 ^②	100	2048	5.5
65 (e.g. Llama)	预训练	1300	2048	27
	二次训练	100	512	8
13 (e.g. Llama)	预训练	1000	256	34
	二次训练	100	128	7

- 考虑利用率：如果是 NVIDIA 80GB A100 GPU, 理论算力是 312 TFLOPS，Megatron 利用 tensor parallel 和 pipeline parallel 并行技术能达到 51.4% 的利用率，也即是每秒能完成 160 TFLOPS。

算力挑战



- 大模型对于算力的需求是呈指数级膨胀式增长的，而硬件算力的供给能力是呈线性增长的，因此，目前对于算力的需求量要比硬件算力供给量高出200倍以上。
- 那么，同时伴随着模型参数的不断增长，在万亿参数时代下的模型训练中，不仅需要大算力，同时对于网络、存储的协同诉求也与日剧增。
- 所以，单机的服务器已经不能够满足万亿参数时代下的大模型训练，只有通过AI集群的方式，才能够更好的满足大规模分布式训练场景诉求。
- 因此，集群是大模型时代下的必然选择，集群系统的创新，也必然会加速AI的创新。

内存和带宽挑战

内存挑战

内存墙

- 内存性能（带宽、延迟）严重滞后于处理器速度，导致处理器等待数据加载而闲置。大模型训练中，内存瓶颈体现在容量不足与带宽限制。

模型参数量激增

- 如DeepSeek-V3 base参数量为671B（未量化时加载模型约685G显存）。
- 单设备内存无法容纳，需分布式训练，但跨设备内存访问延迟高。

容量限制

单GPU内存每2年仅增长2倍，远低于模型参数量增速（每2年410倍）。

碎片化内存

训练中激活值、梯度等临时变量占用大量内存，且难以高效复用。

带宽瓶颈

内存带宽增长缓慢（每2年1.6倍），无法满足大规模矩阵运算的传输需求。

- 未量化时，模型加载需约685GB显存（含671B主模型权重和14B多标记预测模块），通过FP8混合精度训练，权重内存占用减少50%；结合INT8量化，可进一步压缩至约342.5GB显存。

智算中心



特征维度		通算	智算
集群特点	服务器	数十~数百，10~20w¥/台，小于kW级	数十到数千，超过100w¥/台，接近10kW级
	交换机	业务+管理 两平面，不超过100Gb/s	管理+业务+存储+参数+样本 五平面，参数面至少200Gb/s
	互联模块	数千根光纤+数千光模块	数万根光纤+数万颗光模块
机房要求	供电	7KW~13kW/柜	20KW~70KW/柜
	制冷	风冷+列间空调	液冷+风液混合
	承重	<1000kg/m ²	<2000kg/m ²
业务负载	资源利用率	较低，CPU利用率一般10%~30%	AI训练业务，需数十~百天的7x24小时满负荷运转，NPU资源100%占用
系统架构	系统架构	分层解耦	垂直协同
生态迁移	技术成熟度	技术成熟	快速迭代中
管理难度	故障概率	较低	高
	故障定位	较易、分层解耦	难、技术链条长
	中断损失	AI训练发生故障比传统通算故障导致超20x倍损失	

数据挑战

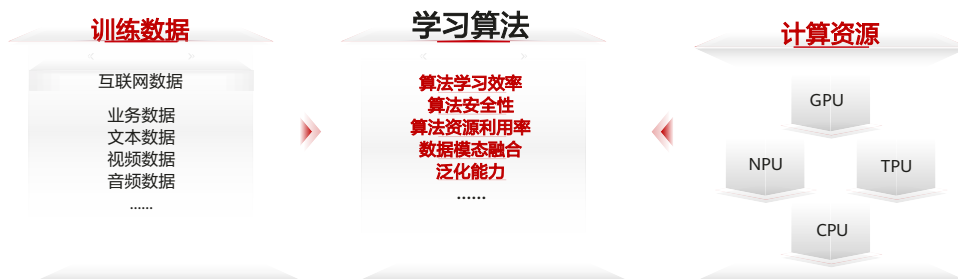
- 大模型通常有数十亿甚至上千亿的参数，需要通过海量数据来调整这些参数，使模型在各种任务上达到最佳性能。
- 开源数据集存在的问题：



模型名称	参数规模	训练数据集大小
DeepSeek-V3	671B	14.8T tokens
Llama 3	70B	15T tokens
GLM4.5	355B	15T tokens

算法瓶颈

- 基于庞大数据和超高算力的“暴力美学”，是当前生成式人工智能的核心理念，但当前高质量数据的增长速度远小于大模型数据集需求的增长速度。当前算法需要进一步加强“学习”效率。



算法挑战



应用场景挑战

场景	优点	缺点
端侧 	低延迟、隐私保护、离线可用	算力受限、模型规模小、更新困难
边缘 	实时处理、减轻云端压力、本地化决策	资源分散、管理复杂、协同难度大
云端 	强大算力、海量数据、多点协同	延迟高、成本高、隐私风险

- 端侧（终端设备）：低延迟：无需网络传输，响应速度快，适合实时性要求高的场景（如语音助手、游戏控制）；隐私保护：数据在本地处理，避免上传云端，降低隐私泄露风险；离线可用：无网络环境下仍能正常工作；算力受限：终端设备算力有限，难以运行复杂模型，需通过模型压缩、量化等技术优化；功能局限：受模型规模限制，端侧模型功能相对单一，复杂任务需依赖云端。
- 边缘侧（边缘计算节点）：低延迟：数据处理在本地完成，响应速度快，适合实时性要求高的场景；减轻云端压力：分担部分计算任务，降低云端资源消耗；提高可靠性：网络中断时仍能保证基本功能运行（如本地决策、缓存处理）；资源分散：边缘设备算力、存储能力有限，难以运行复杂模型；管理复杂：需在多个边缘节点部署和管理模型，维护成本高；数据同步问题：边缘模型与云端模型的数据一致性需额外处理（如增量更新、冲突解决）。
- 云端（数据中心）：超强算力：云端拥有近乎无限的计算资源，可支持大规模模型的训练和推理；高精度与泛化能力：通过海量数据训练，具备更强的泛化能力和更高的精度；灵活性：可根据业务需求动态调整模型规模和计算资源，支持大规模并发请求；延迟高：依赖网络传输，实时性任务可能受影响；成本高：建立和维护云端计算资源需要巨大资金投入；隐私风险：敏感数据上传云端存在泄露风险，需加强安全防护（如数据加密、访问控制）。

端、边侧大模型应用挑战

挑战一：模型小型化的挑战



挑战：

存储开销
内存开销
高带宽
大算力

挑战二：模型性能的挑战



挑战：

功耗
推理速度
逻辑性能

挑战三：“端云协同”的挑战



挑战：

端云模型融合策略，实现通用模型和个性化模型融合解决长尾问题
端侧个性化学习，基于用户个体本地数据，实现端侧小样本学习
确保隐私数据不出端，兼容性能和成本

- 当前的大模型能力主要还是展现在云侧，巨大的算力+巨量的数据+巨大的电力构筑了AI，在云端供客户使用。

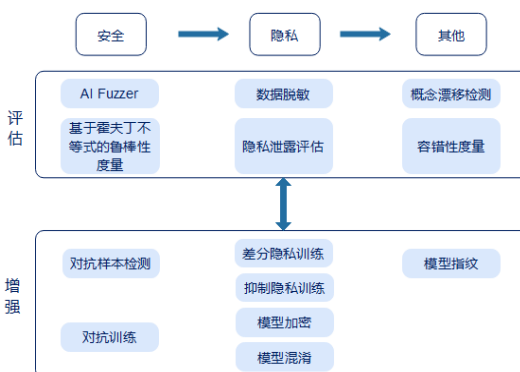
目录

1. AI发展趋势
2. AI行业应用场景
- 3. AI发展面临的挑战**
 - 技术挑战
 - 伦理与商业模式挑战

隐私数据安全

- AI模型在智能化生活方式的同时，也对个人隐私数据被侵害带来了一定的影响。那么应该如何**保护隐私数据**？

- 一、督促企业严格遵守相关法律法规，如《个人信息保护法》、《数据保护法》等，确保在收集、存储、使用、加工、传输、提供等各个环节中合法合规地处理敏感数据。
- 二、对社会各界而言：**积极推进隐私保护相关条款的制定，落实业界基本守则**，加强监管。
- 三、针对开发者而言：可以采用**机密计算、模型隐私保护、联邦学习或对抗学习**等防范逆向攻击及保护用户隐私。或使用成熟的隐私保护框架比如**MindArmour**。



- 学习参考：
 - <https://zhuanlan.zhihu.com/p/170475373>
 - <https://gitee.com/mindspore/mindarmour>
 - <https://mindspore.cn/mindarmour?version=/r1.5/>
- 机密计算：指数据的**传输和计算过程**是机密的，其保护隐私的代价较大。
- 模型隐私保护：常用到的方法有如数据脱敏、差分隐私训练、模型加密、模型混淆等。（差分隐私可以衡量和控制模型对训练数据的泄露）
- 联邦学习：通常是在**不共享数据**的情形下进行多方**联合机器学习**，本质上是一种数据访问受限的分布式机器学习框架。计算节点间**只共享梯度**不共享原始数据。

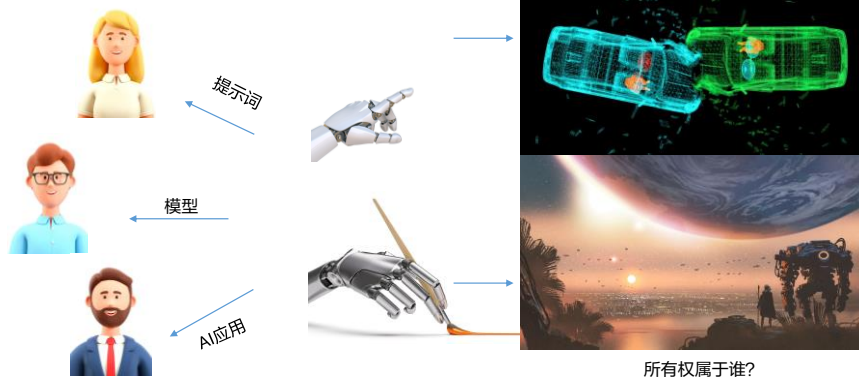
人工智能对就业与社会结构的影响

- 人工智能的发展对就业与社会结构产生影响：
 - 自动化和智能化技术的应用使得一些传统的工作岗位被机器替代，特别是那些重复性、低技能的工作。同时也催生了新的职业需求，例如需要开发、维护和优化这些智能系统的专业人才。
 - 人工智能技术能够提高生产效率，降低生产成本，使得企业能够更快速地响应市场变化，提高竞争力。
 - 社会不平等加剧，虽然人工智能创造了新的就业机会，但也可能加剧社会不平等。那些能够掌握和应用人工智能技术的群体可能获得更多的经济和社会资源。
 - 社会治理的挑战：随着人工智能技术的普及，如何确保数据安全和隐私保护，如何制定和执行相关法规和标准，都是社会治理面临的新挑战。

- 从人类的发展进程上看，我们一直在寻求提高效率的途径，即以更少的资源获得更多。我们用尖锐的石头，更高效地狩猎和采集食物；用蒸汽机，减少了对马匹的需求。实现自动化的每一步都会改变工作生活。在AI的时代，哪些工作会被AI取代呢？

权利与义务

- 人工智能创作物受版权法保护吗？
- 谁来赋权于机器人？ 赋予机器人哪些权利？
-



商业模式挑战



思考题

1. （多选题）以下哪些选项是AI应用发展中可能面临的挑战？
- A. 场景数据较少
 - B. 端、边侧算力不够
 - C. 模型能力不能覆盖需求
 - D. 运营成本高

- 答案：ABCD

本章总结

- 本课程主要介绍了AI应用的发展与挑战，包括AI应用的发展趋势、AI行业应用场景及AI面临的挑战。

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



模型评估与调优



前言

- 在人工智能领域，大模型的精度与性能共同驱动着技术落地的进程。精度决定模型的理解深度与决策准确性，直接影响业务效果；性能则关乎训练效率与资源消耗，决定着工程化可行性。
- 本章节将从两个维度介绍调优策略：精度层面，通过数据集处理、超参数配置、计算精度等方面提升模型效果；性能层面，借助分布式训练加速与硬件协同优化实现效率跃迁。

目标

- 学完本课程后，您将能够：
 - 掌握常用的模型精度与性能评估指标；
 - 分析模型训练出现的精度与性能问题；
 - 在通信、计算等方向对模型训练进行优化。

目录

1. AI模型评估指标

- 精度指标
- 性能指标

2. AI模型调优介绍

3. 基于昇腾的模型调优

模型评估

训的好

评估模型在完成任务时的准确性和可靠性。
通过调整算法结构、数据处理方式、超参数等方法，提升模型在不同场景下的适用性。

指标

- 准确率
- 召回率
- BLEU
- 困惑度
-

优化点

- 数据集
- 超参数配置
- 模型结构
- 计算精度
-

训的快

关注模型在处理任务时的效率和资源消耗。
通过调整训练策略及算法结构，提供硬件资源利用率、减少网络通信和重复计算等，提升模型训练效率。

指标

- FLOPs
- 内存占用
- 训练效率
- 资源利用率
-

优化点

- Device间通信
- Device与Host间通信
- 并行策略
- 算法空间复杂度
-

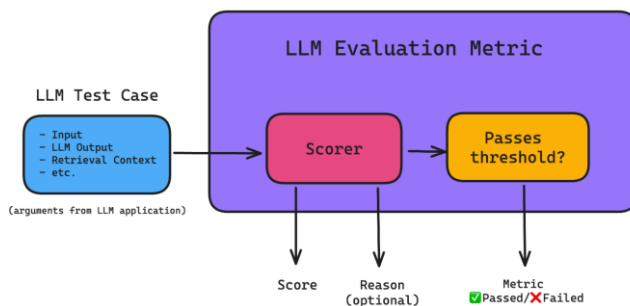
- 准确率：衡量模型正确预测的样本数占总样本数的比例。
- 精确率：衡量模型预测为正的样本中有多少是真正的正样本。
- 召回率：衡量模型样本中的正例有多少被预测正确了。
- F1- score：是精确率和召回率的调和平均数，综合了精确率和召回率。
- GLUE和SuperGLUE：GLUE和SuperGLUE基准测试模拟了真实世界的语言处理场景，并成为了衡量模型自然语言理解能力的标准。

深度学习评估指标

指标名称	任务类型	定义	计算公式
准确率 (Accuracy)	分类	正确预测的样本数占总样本数的比例	$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$
精确率 (Precision)	分类	预测为正类的样本中实际为正类的比例	$Precision = \frac{TP}{TP + FP}$
召回率 (Recall)	分类	实际为正类的样本中被正确预测的比例	$Recall = \frac{TP}{TP + FN}$
F1分数 (F1-Score)	分类	精确率与召回率的调和平均，平衡两者	$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$
ROC-AUC	分类	ROC曲线下的面积，衡量模型对正负样本的区分能力	面积范围 [0.5, 1]，值越大性能越好
均方误差 (MSE)	回归	预测值与真实值差的平方的平均值	$MSE = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$

LLM系统评估指标（1）

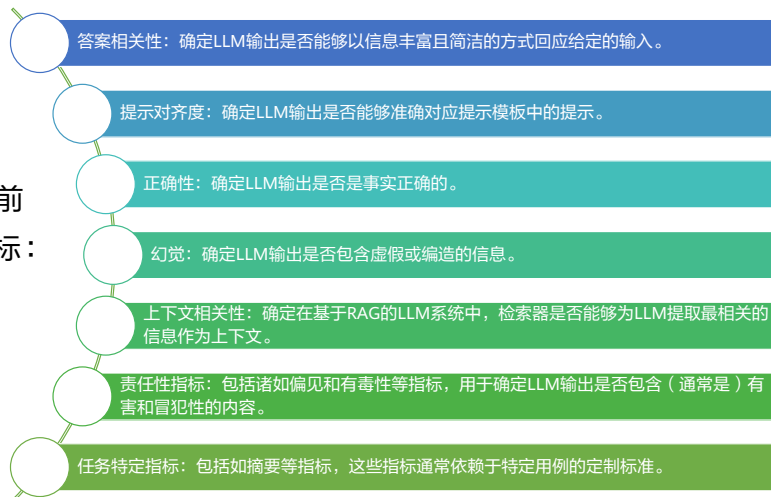
- LLM评估指标，如答案正确性、语义相似度和幻觉等，是根据关注的标准对LLM系统输出进行评分的指标。它们有助于量化不同LLM系统的表现。



大语言模型评估指标框架

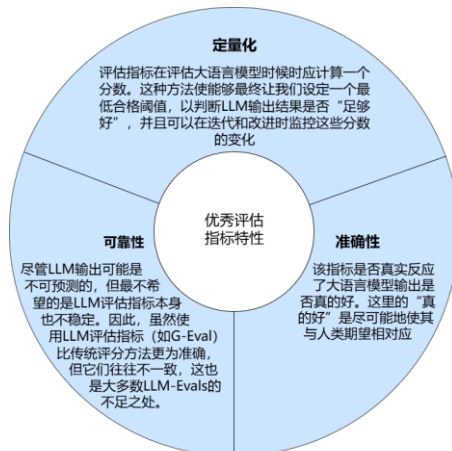
LLM系统评估指标（2）

- LLM系统落地前常见的评估指标：



评估指标特点

- 好的评估指标应该具备以下特点：

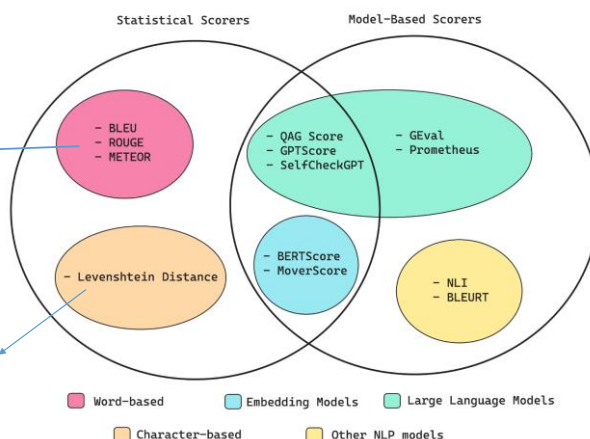


- 目标：让LLM评估指标计算出可靠且准确的分数。

LLM的各种评价指标（1）

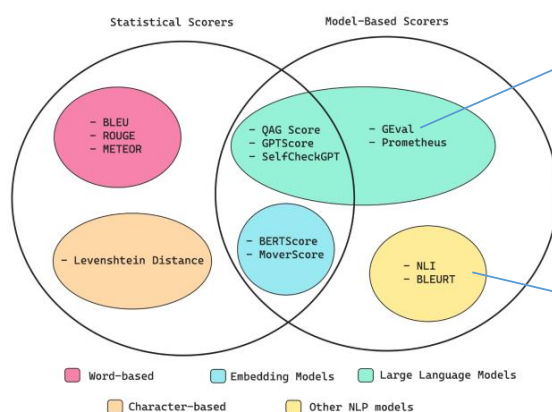
- 统计评价指标大致包括以下几种：

- **BLEU (BiLingual Evaluation Understudy)**：评估输出与真实答案（或期望输出）之间的匹配程度。它通过计算输出与期望输出之间每个匹配n-gram的精确度，计算几何平均值。
- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation)**：主要用于评估生成的文本摘要，并通过比较输出与期望输出之间的n-gram重叠来计算召回率。它确定参考文本中存在于LLM输出中的n-gram的比例（0到1之间）。
- **Levenshtein距离**：计算将一个单词或文本字符串转换为另一个单词或文本字符串所需的最少单字符编辑次数（插入、删除或替换）。这对于评估拼写修正或其他需要精确对齐字符的任务非常有用。



- **METEOR (Metric for Evaluation of Translation with Explicit Ordering)**：它通过评估精确度（n-gram匹配）和召回率（n-gram重叠）来计算分数，并根据LLM输出和期望输出之间的单词顺序差异进行调整。它还利用外部语言数据库（如WordNet）来考虑同义词。最终分数是精确度和召回率的调和平均数，并对顺序差异进行惩罚。
- 由于纯粹的统计评分工具几乎不考虑任何语义，并且推理能力极为有限，它们对于评估通常较长且复杂的 LLM 输出来说，准确性不足。

LLM的各种评价指标（2）



• 基于模型的评价指标主要包含如下几种：

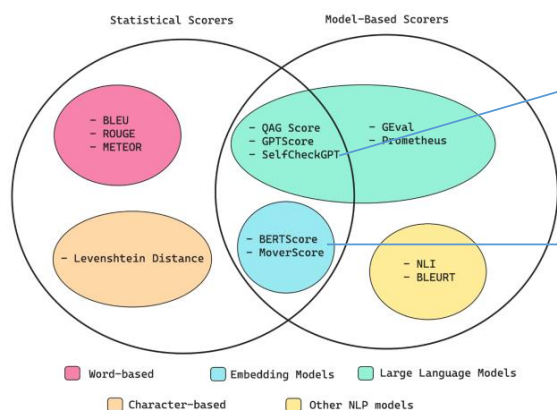
- > **G-Eval**：是一个框架，来自一篇名为《使用 GPT-4 进行 NLG 评估以实现更好的人工对齐》（NLG Evaluation using GPT-4 with Better Human Alignment）的论文 <https://arxiv.org/pdf/2303.16634>，该框架利用大语言模型（LLM）评估 LLM 输出（即 LLM-Evals），并且是创建任务特定指标的最佳方式之一。
- > **Prometheus**：是一个完全开源的大语言模型（LLM），在提供适当的参考材料（参考答案、评分标准）时，其评估能力可与 GPT-4 相媲美。它与 G-Eval 类似，不依赖于特定用例。Prometheus 是一个基于 Llama-2-Chat 的语言模型，并在 Feedback Collection 中通过 10 万条反馈（由 GPT-4 生成）进行微调。

• 基于模型的评价指标主要包含如下几种：

- > **NLI（Natural Language Inference）**：使用自然语言推理模型来分类 LLM 输出与给定参考文本在逻辑上是否一致（蕴含）、矛盾或无关（中立）。得分通常介于蕴含（值为 1）和矛盾（值为 0）之间，提供逻辑一致性的衡量标准。
- > **BLEURT（Bilingual Evaluation Understudy with Representations from Transformers）**：使用像 BERT 这样的预训练模型对 LLM 输出与一些期望输出进行评分。

- G-Eval 首先通过链式思维（CoTs）生成一系列评估步骤，然后使用这些生成的步骤通过填表范式确定最终分数。例如，使用 G-Eval 评估 LLM 输出的一致性时，会构建一个包含评估标准和待评估文本的提示，以生成评估步骤，然后使用 LLM 根据这些步骤输出一个从 1 到 5 的分数。
- Prometheus 遵循与 G-Eval 相同的原则。二者之间的区别在于：1）G-Eval 是一个使用 GPT-3.5/4 的框架，而 Prometheus 是一个经过微调用于评估的 LLM。2）G-Eval 通过链式思维（CoTs）生成评分标准/评估步骤，而 Prometheus 的评分标准则是直接在提示中提供的。
- NLI 评分器在处理长文本时可能遇到准确性问题，而 BLEURT 的准确性则受限于其训练数据的质量和代表性。

LLM的各种评价指标（3）



- 结合基于统计和基于模型的评价指标有如下几个：
- **SelfCheckGPT**：它是一种基于采样的简单方法，用于对 LLM 输出进行事实检查。它假设幻觉输出是无法复现的，而如果一个 LLM 对某个概念有知识，采样的回答可能会相似并包含一致的事实。因为它使得幻觉检测成为一个不依赖参考的过程，这在生产环境中非常有用。SelfCheckGPT 中每个 LLM 生成的句子都会与随机生成的回答进行比较，且不依赖外部数据库。它仅适用于幻觉检测，但不适用于评估摘要、一致性等。
- 结合基于统计和基于模型的评价指标有如下几个：
 - > **BERTScore**：依赖于像 BERT 这样的预训练语言模型，并计算参考文本和生成文本中单词的上下文嵌入之间的余弦相似度。这些相似度然后被聚合以生成最终得分。更高的 BERTScore 表示 LLM 输出与参考文本之间的语义重叠度更大。
 - > **MoverScore**：首先使用嵌入模型，特别是像 BERT 这样的预训练语言模型，获取参考文本和生成文本中单词的深度上下文文化嵌入，然后使用一种叫做地球搬运工距离（Earth Mover's Distance, EMD）的方法，计算将 LLM 输出中单词的分布转化为参考文本中单词分布所需支付的最小成本。

- **QAG Score**：它是一种利用 LLM 高推理能力来可靠地评估 LLM 输出的评价指标。它使用对封闭式问题（通常是：是或否）的回答（这些问题可以是生成的或预设的）来计算最终的评估分数。它之所以可靠，是因为它不直接使用 LLM 来生成分数，而是通过提取 LLM 输出中所做的所有声明（claim）。对每个声明，通过参考事实来判断是否同意（是）或不同意（否）该声明。
- **GPTScore**：与 G-Eval 直接通过填表范式执行评估任务不同，GPTScore 使用生成目标文本的条件概率作为评估指标。其通过两个评估方面：相关性（REL）和信息量（INF），并使用文本摘要任务的相关性（REL）来评估。
- 由于依赖于像 BERT 这样的预训练模型的上下文嵌入，BERTScore 和 MoverScore 都容易受到上下文意识和偏见的影响。

多模态评估指标

- 多模态大模型在不同类型场景下的能力存在明显差异，尤其是在复杂文本识别、图表理解、视频推理、专业领域推断等任务中仍面临诸多挑战。因此，构建系统、客观、全面的评测体系，成为推动多模态智能落地的关键前提。

MME：首个全面的多模态大模型评测基准

MME (Multimodal Evaluation) 基准共包含14个子任务，覆盖基础视觉感知和跨模态认知两大类：

- 基础感知任务：**包括物体存在判断、数量计数、位置关系识别、颜色识别、文字识别等。
- 高级认知任务：**涵盖电影海报识别、名人识别、地标识别、艺术品分类、常识推理、数学计算、代码理解、文本翻译等。

Video-MME：首个全谱视频多模态模型评测基准

Video-MME覆盖视频理解的核心任务，以问答为统一接口。评测数据集包含900段视频，总时长超过254小时，共设计2700道人工标注问答题，构建了一个高质量、多维度、强挑战的评测环境。

- MME基准采用自动化评估体系，核心指标包括：
 - Accuracy：按问题维度计算模型回答正确率。
 - Accuracy+：对同一图像的两道对立问题（如“是否存在猫”和“是否不存在猫”）必须同时答对才计为正确，更能反映模型对图像整体理解的能力。
- Video-MME视频时长覆盖短视频（<2分钟）、中视频（4-15分钟）和长视频（30-60分钟），最短样本仅11秒，最长可达1小时。部分样本提供音频与字幕信息，支持模型进行跨模态融合判断。所有问答均由人工标注，覆盖视频细节、语义理解、语音内容关联、时序事件识别等维度，确保评测的准确性与挑战性。

模型精度评估工具

- **LangTest**: 专注于**语言模型**评估的工具, 提供了60多种测试类型, 用于比较语言模型和NLP模型的准确性、偏差、公平性和鲁棒性。适用于合规审计、模型上线前的安全/公平性筛查。
- **DeepEval**: 开源的**大语言模型**评估框架, 提供了多种评估指标, 包括G-Eval、幻觉检测、答案相关性、RAGAS等。适用于Agent、聊天机器人等场景。
- **Ragas**: 专门用于评估**检索增强生成 (RAG)**系统的性能。它通过一系列指标来评估RAG系统的各个组件和整体性能, 重点关注上下文相关性、上下文召回、忠实性和答案相关性等方面。
- **msit**: 华为推出的**一体化推理开发工具**, 作为昇腾统一推理工具, 提供客户一体化开发工具, 支持一站式调试调优, 当前包括benchmark、debug、analyze等组件。它支持大模型推理迁移全流程, 包括模型推理迁移和优化、大模型精度分析等。

模型精度评估数据集

- 小模型：
 - **ImageNet**：用于图像分类任务的基准数据集。
 - **COCO**（Common Objects in Context）：用于目标检测和分割任务。
- 大语言模型通常采用在标准的benchmark数据集上的测试结果来作为精度指标，以下为常用数据集：
 - **C-Eval**：C-Eval数据集是一个**中文大模型的知识评估基准**，旨在评估基于中文语境的基础模型的高级知识和推理能力。该数据集覆盖了人文、社科、理工、其他专业四个大方向，52个学科和四个难度的级别，从中学到大学研究生以及职业考试，一共13948道题目。
 - **HumanEval**：HumanEval数据集是由OpenAI、Anthropic等开发的一个**代码生成评测基准测试**，包含了164个人工编写的Python编程问题，这些问题覆盖了从基础的字符串操作到复杂的算法设计等多种编程任务。
 - **BoolQ**：BoolQ数据集是一个用于**处理是/否问题的问答数据集**，其中包含了15942个例子。这些问题是自然而然产生的，它们是在未经提示和无约束的环境中生成的。每个例子都是一个三元组（问题、段落、答案），并且页面标题可以作为可选的额外上下文。文本对分类的设置类似于现有的自然语言推理任务。

Task-Specific Evaluation

- 大语言模型使用的领域越来越多，对应不同领域也有各种不同的评价指标。
- Task-Specific Evaluation（任务特定评估）：根据任务的具体需求定制评估指标，以确保结果能准确反映实际能力。与通用评估方法（如准确率、精确率）相比，任务特定评估更注重任务细节和领域特性。

评估方法	适用场景	优点	局限性
通用评估	多任务通用（如准确率、精确率）	实施简单，适用性广	无法捕捉任务细节（如语义流畅性）
任务特定评估	单一任务深度优化（如 BLEU、ROUGE）	精准反映任务表现	开发成本高，通用性差

- 因为本身大语言模型使用的领域越来越多，对应不同领域也有各种不同的评价指标，准确说应该是叫task-specific evaluation。
- 早期的LLM评估基本是采用NLP里已有的方法，就是基于subword统计的方式；但是现在由于大语言模型产生的内容本身就越来越复杂，再加上语义（semantic）和超长上下文窗口的情况，使得基于模型的评价已经变为主流。如果考虑到多模态的大语言模型，那就更是如此了。

目录

1. AI模型评估指标

- 精度指标
- 性能指标

2. AI模型调优介绍

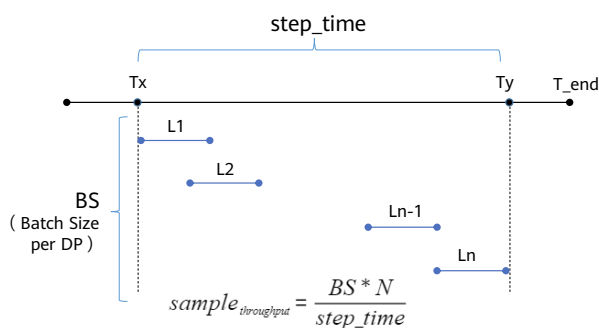
3. 基于昇腾的模型调优

性能评估指标

指标名称	介绍
吞吐率	模型在单位时间内可以处理的最大输入的训练样本数据。
单步迭代时间	模型完成一个批次（batch）输入数据迭代所需要的时间，单位一般是秒（s）。
线性度	单卡训练扩展到多卡，单机扩展到集群的效率度量指标。
内存占用	训练过程中计算硬件显存的使用情况（%）。
训练效率	单位时间内模型处理的数据量或完成的训练步数（tokens/day）。
算力利用率	算力利用率指负载在集群上每秒消耗的实际算力占集群标称算力的比例（MFU、HFU）。

- 在计算性能指标中，优先级排序为：吞吐率 > 单步迭代时间 > 线性度 > 内存占用 > 带宽占比 > 训练效率 > 浮点计算次数每秒 > 算力利用率。
- 线性度的取值范围为0~1，数值越接近于1，其性能指标越好。

吞吐量



吞吐量

Throughput

网络模型在单位时间（例如1s）内可以处理的最大输入的训练样本数据

Batch size

在大模型训练的过程中，batch size的值会影响到重计算、Pipeline并行、Tensor并行等不同并行模式的配比，还有micro batch size的数据配比
可通过顺序搜索、二分搜索等方法寻找最佳batch size

假设网络模型的吞吐量为25 samples/s，max seq_len为1024，那么按照tokens来计算吞吐量为25 * 1024=25600 tokens/s，也就是每秒能处理2万多个tokens。

- BS为Batch Size per DP，即每个数据并行维度的batch size大小。
- N为集群中数据并行维度的大小。
- step_time为在分布式集群中，执行完一个total Batch Size的时间（单位为s）。

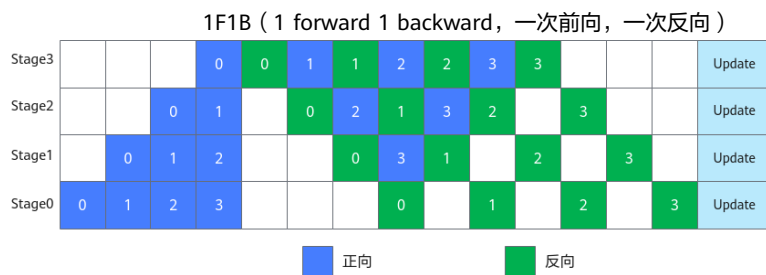
Pipeline并行效率指标 (1)

- 流水线（Pipeline）并行是将神经网络中的算子切分成多个阶段（Stage），再把阶段映射到不同的设备上，使得不同设备去计算神经网络的不同部分。



- 简单地将模型切分到多设备上并不会带来性能的提升，因为模型的线性结构使得在同一时刻只有一台设备在工作，而其他设备在等待，造成了资源的浪费。为了提升效率，流水线并行进一步将小批次（MiniBatch）切分成更细粒度的微批次（MicroBatch），在微批次中采用流水线式的执行序，从而达到提升效率的目的。
- 将小批次切分成4个微批次，4个微批次在4个组上执行形成流水线。微批次的梯度汇聚后用来更新参数，其中每台设备只存有并更新对应组的参数。其中白色序号代表微批次的索引。

Pipeline并行效率指标 (2)



假设微批次数量是m, pipeline stage是p, 流水线并行的效率是:

$$pipeline\ Utilization = \frac{m}{p + m - 1}$$

算力利用率

算力需求

LLM（Transformer结构）主要的浮点算力（输入序列较长时）都由来自transformer block和logit层中的矩阵乘。

考虑（未做矩阵乘法优化）：

一个具有 I 个Transformer层的LLM负载，

此外，如果使用了重计算，还需要在反向传递之前进行额外的前向传递。

考虑 I 层LLM（GPT-like）负载一个iteration中GEMMs的FLOPs算力 F 。

当前算力规模： N 块算力为 C 的Device组成的集群，
算力利用率为 E

$$FLOPs: 3(24Bsh^2 + 4Bs^2h)$$

$$96Bsh^2 + 16Bs^2h$$

$$F = 96BsIh^2 \left(1 + \frac{s}{6h} + \frac{V}{16Ih} \right)$$

$$E = \frac{F}{N * C * step_time}$$

B: batch size
s: sequence length
I: transformer block num
h: hidden size
V: vocabulary size

- 一个的矩阵乘 $A_{m \times k} * X_{k \times n}$ 法需要 $2mnk$ 个浮点运算（乘加运算算两次）。Transformer层由一个注意力模块（attention）和一个2层前馈网络（FFN）组成。对于注意力模块，主要的贡献者是键（key）、查询（query）和值（value）的转换（ $6Bsh^2$ 次运算），注意力矩阵计算（ $2Bs^2h$ 次运算），对值进行注意力计算（ $2Bs^2h$ 次运算）和注意力后的线性投影（ $2Bsh^2$ 次运算）。前馈网络将隐藏大小增加到 $4h$ ，再将其缩小回 h 。这需要 $16Bsh^2$ 个浮点运算。

MFU&HFU

- MFU、HFU用于衡量大规模语言模型在训练过程中的实际计算效率，更能反映深度学习训练的真实性能瓶颈。
- MFU衡量的是模型在训练过程中对计算资源的有效利用程度。

$$\text{MFU} = \frac{\text{模型实际使用的FLOPs}}{\text{硬件理论FLOPs} \times \text{训练时间（秒）}}$$

- HFU衡量的是硬件计算能力的实际利用率，即实际发生的计算量与硬件理论算力的比值。

$$\text{HFU} = \frac{\text{硬件实际执行的FLOPs}}{\text{硬件理论FLOPs} \times \text{训练时间（秒）}}$$

线性度

- 线性度为单卡训练扩展到多卡，单机扩展到集群的效率度量指标。

$$\text{单机内部线性度} = \frac{\text{单机多卡总吞吐率}}{\text{单卡吞吐率} * \text{卡数}}$$

- 扩展到集群：

$$\text{集群线性度} = \frac{\text{多机多卡总吞吐率}}{\text{单卡吞吐率} * \text{卡数} * \text{集群机器数量}}$$

- 如果机器本身是多卡或多机的加速比接近高线性度（即线性度接近于1），说明在扩展时通信不是瓶颈，则通过改变或者增加通信带宽的性能提升，对于整体模型训练的性能提升在通信问题中空间就会比较小。

- 当线性度不高（例如小于0.8）时，排除数据IO和CPU的本身因素影响后，可以判断此时分布式通信存在瓶颈。

目录

1. AI模型评估指标
- 2. AI模型调优介绍**
 - 精度调优
 - 性能调优
3. 基于昇腾的模型调优

精度影响因素

训练数据的质量与多样性

- 数据质量：训练数据中的错误、噪声或虚假信息会直接影响模型输出准确性
- 数据多样性：覆盖多领域的高质量数据

模型架构与参数量

- 模型结构的设计
- 参数精度的选择

训练过程

- 学习率、Batch Size等超参数的设置
- 混合精度训练的配置

数据集引发的精度问题

- AI模型对数据集的质量敏感，无效数据、低质量数据会让模型学习到错误的模式，甚至影响模型收敛。重复数据会加剧语言模型生成内容重复的问题，并且诱导模型过拟合。

数据异常	语料类型	数据规模
引起训练异常的语料包括字节码、unicode乱码、纯数字文本等。	不同语言不同语料类型（代码、文本）的混合比例和语料规模对大模型训练精度也有重要影响。	当语料的token数与模型的参数个数相当时，模型的表现相对更好。

- 数据异常触发的训练不稳定：引起训练异常的语料包括字节码、unicode乱码、纯数字文本等。这些语料会无报警地被主流的subword tokenizer分割成token，并转成id序列正常进入训练迭代步骤，但由于这些语料本身没有任何语法语义规律，根据它们来更新参数可能会影响模型的正常学习过程。当异常语料数量显著时，会导致模型无法正常收敛。
- 语料混合比例与类型的重要性：不同语言不同语料类型（代码、文本）的混合比例和语料规模对大模型训练精度也有重要影响。需要调节不同来源的语料的混合比例，不能直接使用原始语料合并训练。均衡的语料比例有助于提高模型的泛化能力，特定类型的语料可以提升模型特定的能力。
- 数据规模与模型参数规模的匹配：参数规模并非越大越好，Hoffmann等人在Chinchilla系列模型中的研究表明，在给定算力的情况下，当语料的token数与模型的参数个数相当时，模型的表现相对更好。因此，不能盲目追求大语料，也应重视控制规模、提高质量和训练充分。

数据处理

数据处理方式

去重

去除相似度较高的数据，根据数据需求设置重复度阈值（如需求量较高时可设置80%的阈值）。根据成本选择去重方式（如成本较低选择document去重）。

清洗

清洗数据内的脏、低质量数据。通过训练好的模型（如BERT）对数据进行打分。设置清洗规则，对包含http、html等字段的低质量数据或过长过短的数据进行清洗。

配比

通过分类器对当前数据进行类别划分，进行知识分类。对训练数据按照知识、代码、逻辑等类目进行划分，逻辑数据多为数学和cot数据。整体上数据的中英文数据和代码数据比例可设置为4:4:2，可以提升代码数占比量。

流水线

预训练模型的数据加载为动态加载（读1B、训1B），数据处理和模型训练为独立进程，所以构建数据流水线，保证模型训练中始终有数据可用，且每份数据对应一个checkpoint方便回溯。

测试

构建完数据后不直接开始训练，通过scaling law法则推算模型合适的数据配比、超参数等信息。

模型结构与算法

- 参数：并非越多越好（除特定任务外）。

- 考虑算力：

- 确保训练算力可以在需求时间内训练完成。
 - 确保模型适配推理设备（避免单卡不够、多卡剩余的情况）。

- 考虑当前训练数据量。

模型名称	版本	参数量 (B)	Hidden Size	Attention Heads	层数 (Layers)
Llama3	7B	7	4096	32	32
	13B	13	5120	40	40
	70B	70	8192	64	80
DeepSeek	V2	236	5120	64 (Q头) / 8 (KV头)	60
DeepSeek	V3	671	7168	128	61
Mistral	7B	7.3	4096	32	32
Falcon	40B	40	8192	64	60
Qwen	7B	7	4096	32	32
Qwen	14B	14	7168	56	60

- 业界常使用scaling law来预测一个算法在大模型上的收敛趋势。同时要注意的是，由于大模型代码结构越来越复杂，算法实现错误时有发生，而这会对大模型收敛性有决定性影响。如果使用新设计的实验性模型结构，需要经过反复调试和实验，确保实现没有错误，才能使得模型Loss收敛，并且达到和主流模型匹配甚至更好的性能。

超参数配置优化

参数	优化方式
batch size	在模型训练过程中，batch size的设置通常较大，能够充分利用大规模训练数据，增强模型训练过程的稳定性。
学习率	学习率一般较小，且包含warm up设置，以确保训练平稳。峰值学习率一般在 10^{-4} 以下 中期会采用cosine衰减或者线性衰减学习率调度策略，逐渐减小学习率，在收敛前将学习率下降到峰值学习率的10%左右（收尾阶段学习率低，能够精细适应高质量数据特点）。
优化器	优化器一般采用Adam、AdamW以及Adafactor。 调整好adamw优化器的 β_1 、 β_2 这两个参数。
权重初始化	正确初始化权重可以帮助模型更快地收敛并提高性能。例如，通常使用小的高斯噪声或者使用T-fixup初始化。
梯度裁剪	通常设定裁剪阈值为1.0，防止梯度过大引发训练不稳定。
Embedding缩放系数（可选）	embedding层往往存在较大的梯度异常情况，故需根据实际情况适度调整相关参数。

混合精度训练对模型精度的影响

特定计算对精度的要求

softmax函数	Cross Entropy	矩阵乘法	归一化	CNN/RNN	通信算子	模型起始与结束层
指数运算，易发生上溢和下溢，且分母有累加操作，适用FP32。	需要使用FP32计算以保持数值稳定性，所以算子实现只接受FP32输入。	通常可以用FP16；attention层，attention分数与V张量的矩阵乘法需要用FP32。	layer_norm、Batch_norm、RMSNorm需要用FP32计算。	对精度不敏感，可以使用BF16，不会对模型收敛性产生影响。	涉及到梯度或者激活值累加的算子，都需要把输入张量upcast到FP32以保证数值稳定性。	通常大语言模型的第一层和最后一层对精度敏感，需要使用FP32。

混合精度训练过程中如果对应算子使用精度类型不合理，可能会导致训练精度下降或不收敛。

- Softmax函数：softmax函数中有指数运算，容易发生上溢出和下溢出，并且分母有累加操作，目前主流的算子库只使用FP32来实现softmax。
- Cross Entropy损失计算：cross_entropy用来计算交叉熵损失，和很多损失函数一样，需要使用FP32计算以保持数值稳定性，所以算子实现只接受FP32输入。
- 矩阵乘法（Matmul）：matmul矩阵乘法一般来说可以使用半精度数，但在大语言模型的attention层，attention分数与V张量的矩阵乘法需要用FP32，以便维持数值稳定性，避免上溢，减小累积错误。
- 归一化函数：归一化函数如layer_norm、Batch_norm、RMSNorm需要用FP32计算。layer_norm要计算方差，有累加操作，如果用FP16，则计算过程中容易发生上溢和比较大的误差，从而影响最后的收敛。Batch_norm与layer_norm类似，也需要将输入张量upcast到FP32计算。RMSNorm是layer_norm的扩展，在开源LLaMA模型中使用，PyTorch目前并没有提供RMSNorm的实现。如果开发者自己实现，需要注意将输入张量强制upcast到FP32计算，结果可以转回半精度数，主要原因也是它需要计算L2均值，涉及累加操作，低精度运算容易累积误差。
- 卷积操作与池化：卷积操作属于逐单元矩阵乘法，对精度不敏感，可以使用BF16，不会对模型收敛性产生影响。和卷积类似的还有池化，同样可以使用BF16。ReLU和GeLU属于逐点运算，同样对精度不敏感。
- 循环神经网络（RNNs）：RNN一般对精度比较敏感，LSTM-cell对精度不敏感，可以用BF16。
- 模型起始与结束层：通常大语言模型的第一层和最后一层对精度敏感，比如GPT的

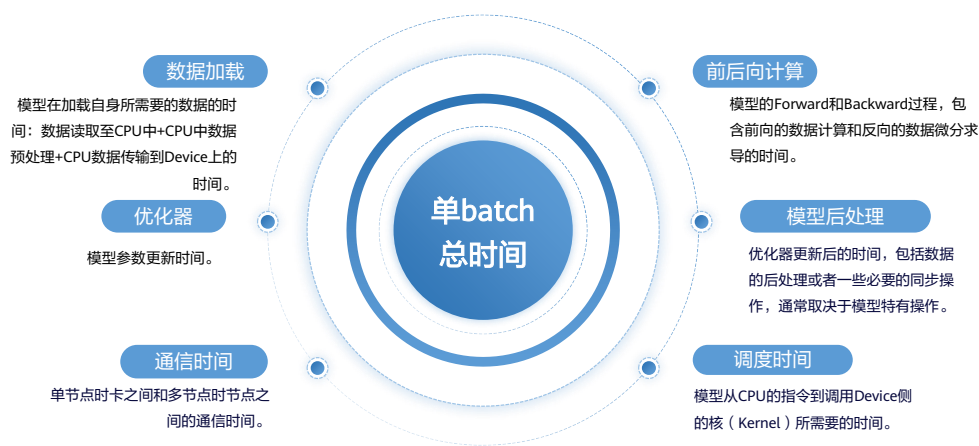
embedding层（词嵌入和位置嵌入层）需要使用FP32。一般的输出层即便不是softmax，计算和输出结果也需要是FP32。

- 通信算子：涉及到梯度或者激活值累加的通信算子如all_reduce、reduce_scatter，都需要把输入张量upcast到FP32以保证数值稳定性。

目录

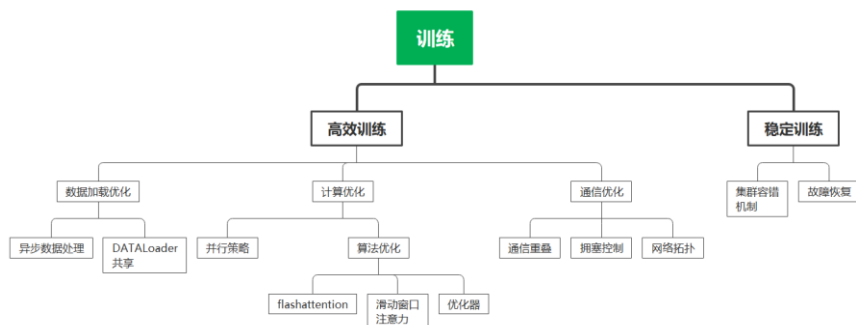
1. AI模型评估指标
- 2. AI模型调优介绍**
 - 精度调优
 - 性能调优
3. 基于昇腾的模型调优

单步训练耗时



- 单batch总时间 = 数据加载时间 + 模型前反向时间 + 优化器时间 + 模型后处理时间 + 通信时间 + 调度时间。
- 模型在加载自身所需要的数据（如图片、视频和文本等）的时间，包括将数据从硬件存储设备读取到CPU中、CPU中数据的预处理（编解码等操作）、CPU数据放到Device上的时间。对于一些需要切分在若干张卡上的模型，数据加载还包括从数据加载卡广播到其他卡上的时间。
- 模型前反向时间：深度学习模型的前向过程和反向过程的时间，即Forward和Backward过程，包含前向的数据计算和反向的数据微分求导的时间。
- 优化器时间：模型参数更新时间。
- 模型后处理时间：优化器更新后的时间，包括数据的后处理或者一些必要的同步操作，通常取决于模型特有操作。
- 通信时间：单节点时卡之间和多节点时节点之间的通信时间。由于PyTorch的特殊机制，在通信和计算可以并行的情况下，表示未被计算掩盖的通信时间。
- 调度时间：模型从CPU的指令到调用NPU侧的核（Kernel）所需要的时间。

性能调优



- 当训练集群达到万卡规模，如何实现高效率、高稳定的训练？

- 对于大集群训练，因为机器数量众多，有可能出现某些机器节点计算较慢的情况，这种节点一般被称之为慢节点。慢节点通常会导致模型的通信时间大大增加。

数据IO优化 (1)

- 对于数据量较多的数据集使用MindRecord和TFRecord等格式。这种数据格式可以减少磁盘IO、网络IO开销。



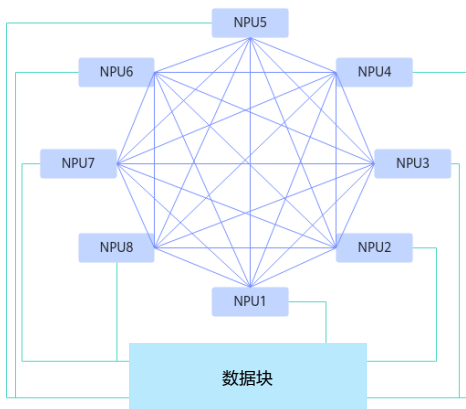
- 更换读写速度较快的数据存储设备。



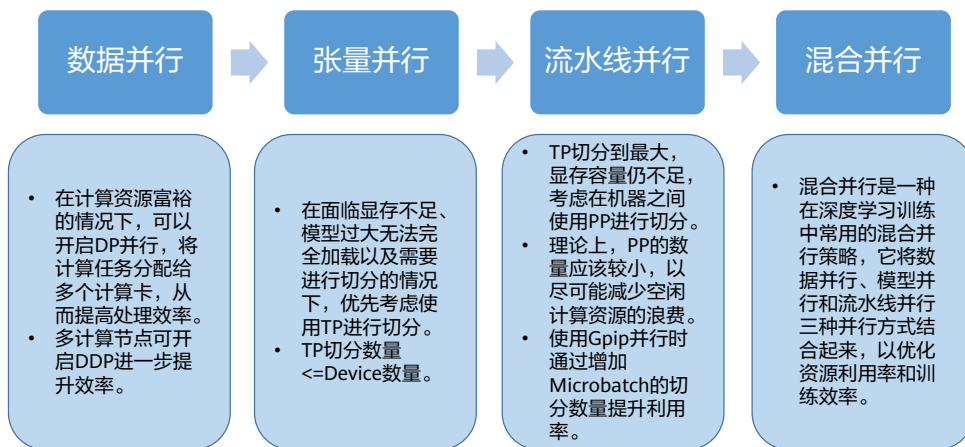
- 其中数据文件包含文件头、标量数据页、块数据页，用于存储用户归一化后的训练数据，且单个MindSpore Record文件建议小于20G，用户可将大数据集进行分片存储为多个MindSpore Record文件。而索引文件则包含基于标量数据（如图像Label、图像文件名等）生成的索引信息，用于方便的检索、统计数据集信息。数据文件中的文件头、标量数据页、块数据页的具体用途如下所示：
 - 文件头**：是MindSpore Record文件的元信息，主要用来存储文件头大小、标量数据页大小、块数据页大小、Schema信息、索引字段、统计信息、文件分区信息、标量数据与块数据对应关系等。
 - 标量数据页**：主要用来存储整型、字符串、浮点型数据，如图像的Label、图像的文件名、图像的长宽等信息，即适合用标量来存储的信息会保存在这里。
 - 块数据页**：主要用来存储二进制串、NumPy数组等数据，如二进制图像文件本身、文本转换成的字典等。
- MindSpore Record数据格式具备的特征如下：
 - 实现数据统一存储、访问，使得训练时数据读取更加简便。
 - 数据聚合存储、高效读取，使得训练时数据方便管理和移动。
 - 高效的数据编解码操作，使得用户可以对数据操作无感知。
 - 可以灵活控制数据切分的分区大小，实现分布式数据处理。

数据IO优化 (2)

- 异步数据加载：在Device同步梯度的时候执行下一个Step的数据处理，可以隐藏掉数据处理的开销；
- DataLoader共享：考虑到一个TP组都是在同一台机器内，需要读取相同的数据，因此可以在每台机器共享DataLoader，并将数据放在共享内存，每个Worker从共享内存加载数据即可。



并行策略优化（1）



并行策略优化（2）

内存效率

- 模型的层被分为流水线阶段，每个阶段的层通过模型并行进一步划分。这种 2D 组合同时减少了模型、优化器和激活函数消耗的内存。然而，不能无限制地分割模型，否则会因通信开销而限制计算效率。

计算效率

- 为了让 Worker 数量在不牺牲计算效率的情况下超越模型和流水线并行，使用 ZeRO 驱动的数据并行（ZeRO-DP）。ZeRO-DP 不仅通过优化器状态分区进一步提高了内存效率，还通过利用拓扑感知映射，使 GPU 数量的扩展具有最小的通信开销。

降低内存

- 在模型能够成功运行的情况下，也可以尝试主动地使用降低内存占用的手段，例如ZeRO1和重计算等。

拓扑感知 3D 映射

- 3D 并行中的每个维度都被仔细映射到 Worker 上，通过利用两个关键的架构属性实现最大计算效率。

算法优化

- 并行transformer块：

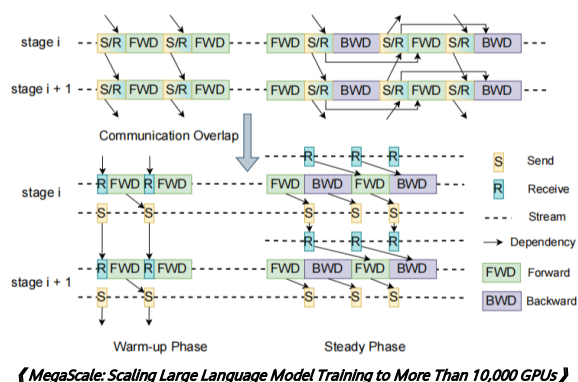


- 稀疏注意力机制：
 - 滑动窗口注意力：在每个词元周围采用了固定大小的窗口注意力。在这一注意力机制中，给定一个固定的窗口大小 w ，每个词元会关注其两侧的 $w/2$ 个词元。这种注意力机制的计算复杂度为 $O(n*w)$ ，与输入序列的长度 n 成线性关系。
- 优化器：
 - LAMB/LARS优化器。

- LAMB优化器旨在不降低精度的情况下增加训练batch size，支持自适应逐元素更新和精确的分层校正。
- https://www.mindspore.cn/docs/zh-CN/r2.6.0/api_python/nn/mindspore.nn.Lamb.html

计算通信重叠 - 流水线并行

- 除了降低绝对的通信耗时，还可以从降低整体训练耗时角度来优化，可以考虑通信和计算的异步流水实现。
- 数据并行中的梯度同步Allreduce通信是在训练的反向过程中进行的，而 Allreduce 后得到的同步梯度是在训练的更新过程中才被使用，在反向中并没有被使用。也就是说上一个梯度的通信和下一个梯度的计算间并没有依赖，通信和计算可以并行，让两者的耗时相互重叠掩盖，减少反向的耗时。



- MegaScale (<https://arxiv.org/pdf/2402.15627>) 采用交错式的流水并行，这种并行的通信时点对点的send和receive操作，这两个操作通常是在一起的（连续进行），把这两个操作解耦，再把解耦后的操作与计算过程进行重叠，就能节省通信开销。

计算通信重叠 - 张量并行

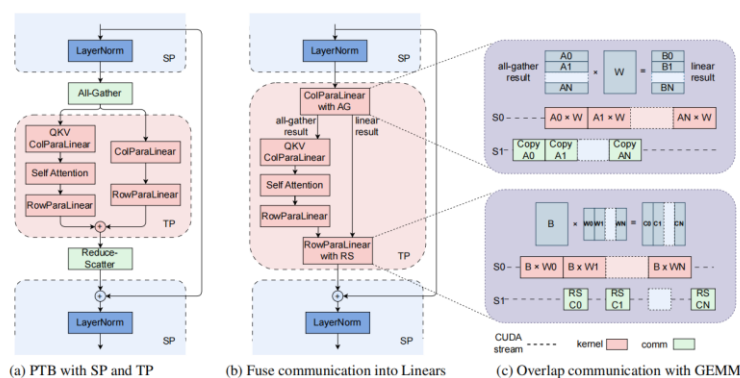


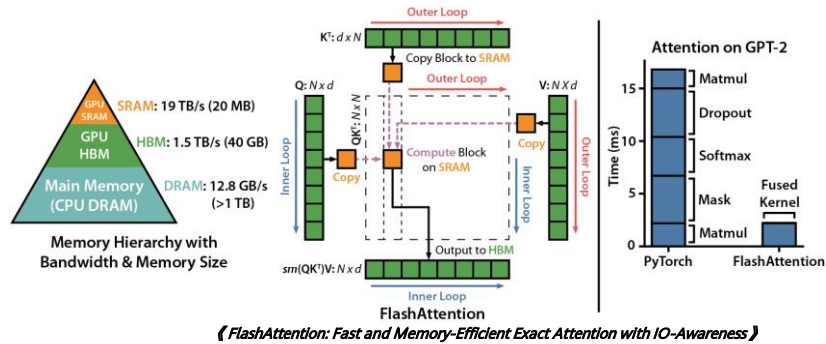
Figure 3: Overlapping communication in tensor parallelism (TP) and sequence parallelism (SP) with parallel transformer block (PTB).

《MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs》

- 张量并行通常将计算密集的操作，如Attention和MLP的权重划分到不同的设备上并行计算，同时将计算没那么密集的LN和Dropout进行序列并行（非真正意义的序列并行），这个张量并行接序列并行的操作需要在参数进入张量并行时进行all-gather通信，在进行序列并行时，进行reduce-scatter通信。由于MLP层的计算量较大能够更好的隐藏通信，因此，将all-gather通信和ColParaLiner（第一个FC层）进行融合，将reduce-scatter与RowParaLinear（第二个FC层）进行融合，实现通信隐藏的目的。

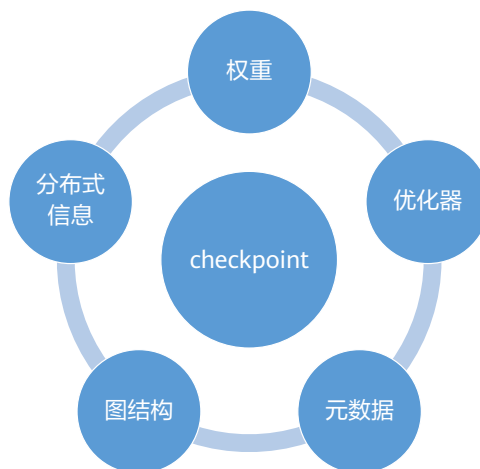
FlashAttention

- FlashAttention将长序列拆分为小块，避免一次性存储整个注意力矩阵。
 - 每次处理一个块，利用GPU的SRAM（静态随机存取存储器）进行高速计算，减少对HBM（高带宽内存）的访问。

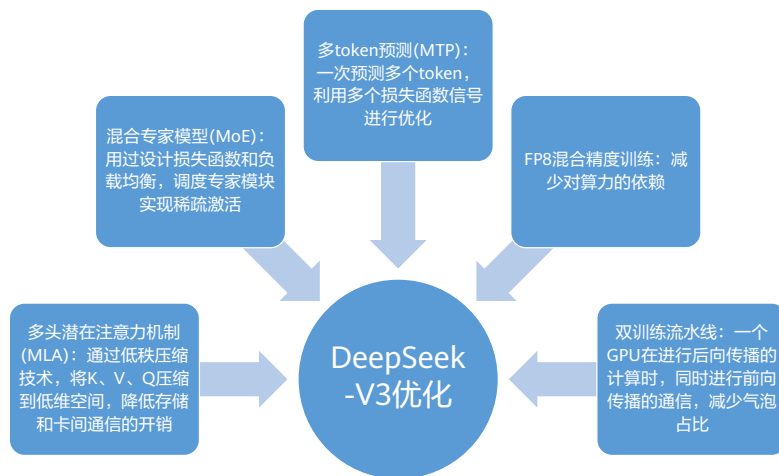


Checkpoint开销

- 将模型参数、优化器状态等数据写入持久化存储（如SSD、HDFS）所需的时间。对于大型模型，参数和优化器状态可能高达数百GB甚至TB级别，导致保存操作耗时较长。
- 在分布式训练中，若checkpoint存储在远程存储系统（如HDFS），上传和下载时间可能成为瓶颈。



DeepSeek-V3工程优化



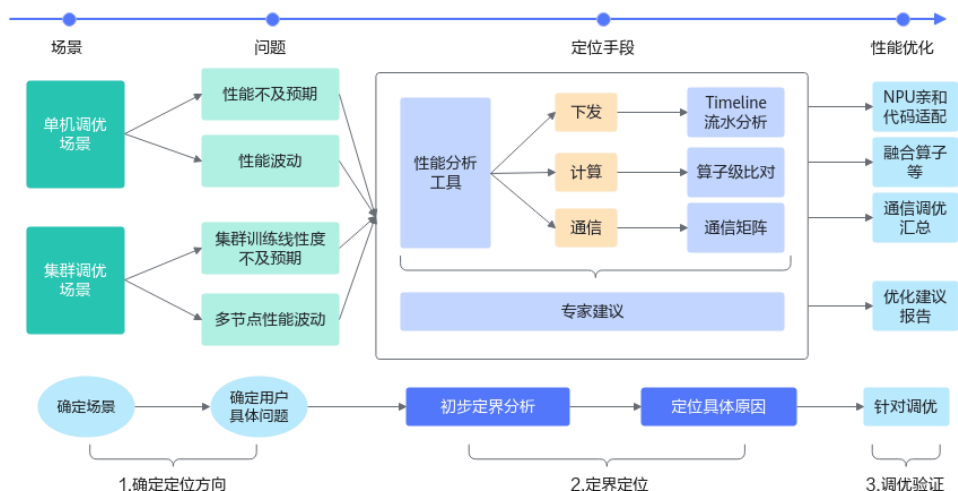
目录

1. AI模型评估指标
2. AI模型调优介绍
- 3. 基于昇腾的模型调优**
 - NPU性能及精度调优
 - 调优案例

Ascend性能分析工具

工具	功能
Ascend PyTorch Profiler	可全面采集PyTorch训练场景下的性能数据，主要包括PyTorch层算子信息、CANN层算子信息、底层NPU算子信息以及算子内存占用信息等，可以全方位分析PyTorch训练时的性能状态。
消除性能比对工具 (compare_tools)的stream同步	用于比较模型在同平台或不同平台的性能差异，目前支持的比较GPU与NPU之间、NPU与NPU之间的性能差异，通过对训练耗时和内存占用的比对分析，定位到具体劣化的算子，帮助用户提升性能调优的效率。工具将训练耗时拆分为计算、通信、调度三大维度，并针对计算和通信分别进行算子级别的比对；将训练占用的总内存，拆分成算子级别的内存占用进行比对。
集群分析工具 (cluster_analyse)	在集群场景下，通过此工具来进行集群数据的分析，当前主要对基于通信域的迭代内耗时分析、通信时间分析以及通信矩阵分析为主，从而定位慢卡、慢节点以及慢链路问题。
MindStudio Insight	针对大模型集群场景的可视化调优工具，包括了Timeline视图、通信分析、计算耗时等的可视化呈现，以使用户分析潜在的性能瓶颈，并指导如何采取措施提升性能。

基础优化流程



- 根据性能问题的场景，按照单机和集群场景进行分类，再明确性能问题属于哪一类，明确好性能问题背景之后，才方便进行下一步问题的定位。
 - 在明确问题背景后，参考性能调优工具介绍，选择对应的性能工具，采集性能数据并拆解性能，找到需要提升性能模块；
 - 在明确性能瓶颈模块后，将问题细化定位到下发、计算和通信等模块，并通过本文目录搜索到对应章节找到对应优化算法。

排查思路



数据加载优化

优化方式	功能
硬件	检查数据存放的硬盘，最好在NVME（Non-Volatile Memory Express，一种更快、更可靠的存储设备，适合存储需要高速读写的数据）盘上，当硬盘为共享存储时，需要注意IO瓶颈。 检查NUMA（Non-Uniform Memory Access，非一致性内存访问）数量，可以通过lscpu命令查看，一般需要NUMA数量为4或者8。
参数类 (PyTorch DataLoader)	一般而言，对于图像或视频等复杂数据，可以将num_workers设置的大一些（如4或8）。 在内存允许的情况下，保持pin_memory=True，在PyTorch中，该参数可以写为pin_memory_device。 一般而言，可以加上persistent_workers=True，该设置可以减小进程销毁或申请的开销，不过这个方法也可能带来内存瓶颈。
其他	在数据预处理时间较长而影响模型训练时，可以尝试预取数据。 提前缓存数据或制作二进制数据，轻量化数据加载。 对于NPU预处理数据比较慢，例如dlrm模型，可以考虑将预处理放到datasets里用CPU多num_workers处理数据。

NPU亲和适配优化

- 基于一键迁移的方式可以保证GPU的代码迁移到NPU上的代码修改轻量化，显著降低代码适配成本，但站在性能优化的角度上，鉴于NPU和GPU在计算原理和底层架构上存在非常大的区别，因此，需要进行模型层面的修改，亲和适配NPU，达到最优的性能。

方式	实现
算子融合	通过等价替换，将多个算子融为一个算子，减少冗余计算和下发次数
消除多余的stream同步	减少host和device侧的stream同步操作
多卡训练性能不一致	多卡训练性能不一，出现快卡等慢卡现象，需用性能工具找根源，并优化慢卡效率
CPU上运行的优化方法	优化模型CPU侧计算效率，或将其迁移至NPU计算

融合算子替换（1）

使用pytorch实现RMSNorm

```
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def _norm(self, x):
        return x * torch.rsqrt(x.pow(2).mean(-1,
keepdim=True) + self.eps)

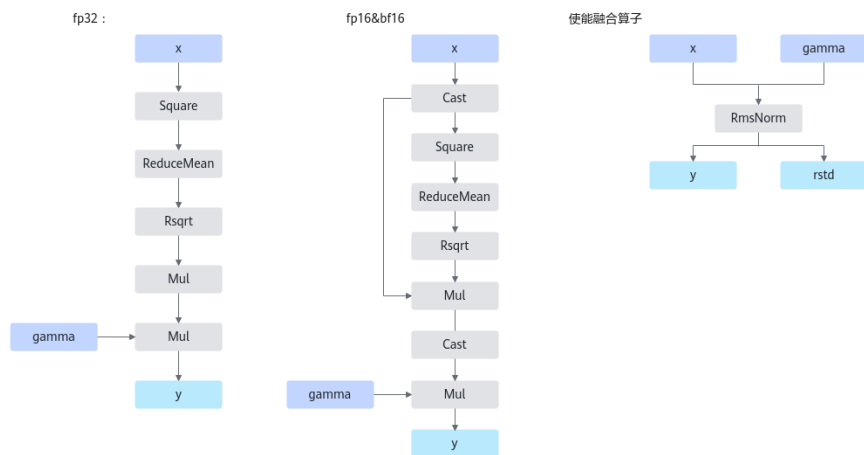
    def forward(self, x):
        output = self._norm(x.float()).type_as(x)
        return output * self.weight
```

替换为NPU融合算子

```
import torch_npu
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        return torch_npu.npu_rms_norm(x, self.weight,
epsilon=self.eps)[0]
```

融合算子替换（2）



- 融合后多了一个输出`rstd`，为计算中间结果，对应`torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)`，用于反向算子输入。

融合优化器（1）

- 一般而言，优化器会频繁进行h2d（host to device）的下发操作，以完成对模型参数的更新，频繁的下发会拖慢模型的训练。而这些操作的底层逻辑是相同的，只是作用的数据不同，因此，统一在Host侧完成操作，再统一下发到Device上，能够显著减少h2d的操作，提高性能。
 - 因为融合后的统一下发会要求更大的内存空间，无法利用Device上的小块内存，因此融合优化器可能会占用更多的内存。

融合优化器（2）

优化器	样例原代码	修改后代码
sgd	optimizer = torch.optim.SGD (model.parameters(), lr=args.lr)	optimizer = torch_npu.optim.NpuFusedSGD (model.parameters(), lr=args.lr)
adadelata	optimizer = torch.optim.Adadelata (model.parameters(), lr=args.lr)	optimizer = torch_npu.optim.NpuFusedAdadelata (model.parameters(), lr=args.lr)
lamb	optimizer = torch.optim.Lamb (model.parameters(), lr=args.lr)	optimizer = torch_npu.optim.NpuFusedLamb (model.parameters(), lr=args.lr)
adam	optimizer = torch.optim.Adam (model.parameters(), lr=args.lr)	optimizer = torch_npu.optim.NpuFusedAdam (model.parameters(), lr=args.lr)
adamw	optimizer = torch.optim.AdamW (model.parameters(), lr=args.lr)	optimizer = torch_npu.optim.NpuFusedAdamW (model.parameters(), lr=args.lr)

算子替换

- 矩阵索引等操作时，可以对其进行替换，如替换成乘法操作。
- `torch.where -> torch.lerp`
- `tensor_a[mask].sum() -> (tensor_a * mask).sum()`

```
target_mask = (target < vocab_start_index) | (target >= vocab_end_index)
masked_target = target.clone() - vocab_start_index
masked_target[target_mask] = 0
```



```
target_mask = (target < vocab_start_index) | (target >= vocab_end_index)
masked_target = target.clone() - vocab_start_index
masked_target *= ~target_mask
```

- 由于昇腾SIMD（Single Instruction，Multiple Data，单指令多数据）架构的特点，每个指令能够融合更多的数据，应避免多次进行小数据的随机访问，因此，在遇到矩阵索引等操作时，可以对其进行替换，如替换成乘法操作。将索引操作转换为矩阵的加乘操作，从而避免大量随机访存带来的性能劣化问题。

•

其他常用优化操作

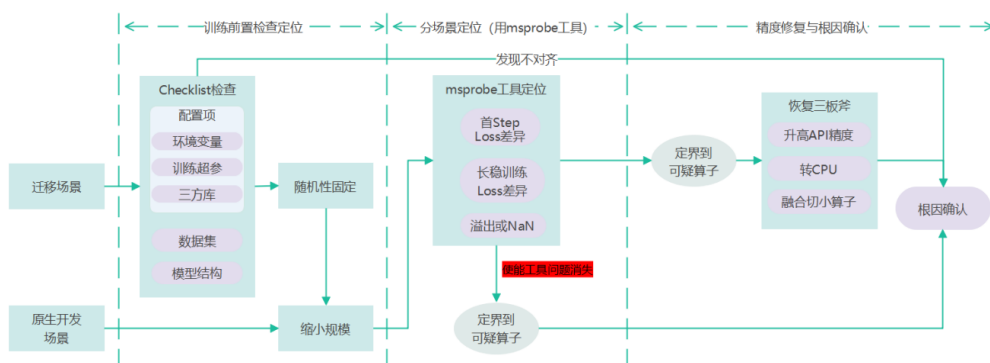
优化器	修改后代码
ACLNN_CACHE_LIMIT	ACLNN_CACHE_LIMIT用于配置单算子在执行API时，在Host侧缓存的算子信息条目个数。动态shape场景下，如算子shape范围较大时，用户可通过配置此环境变量适当增加算子缓存条目，提升调度性能。
非连续转连续消除	GPU上可以只支持对于非连续的tensor进行操作而不会产生额外性能问题，对于NPU则需要插入非连续转连续算子。因此GPU上某些代码运行会连续对一个非连续tensor进行操作，对NPU来说就会插入多余的转换算子造成性能下降。通常，permute和transpose操作都会造成tensor非连续：tensor.permute(0, 1) -> tensor.permute(0, 1).contiguous() 。
mmcv	mmcv的optimizer加入runner (限昇腾适配的mmcv的optimizer)，会有性能收益。
NPU-Fast_GELU	NPU的Fast_GELU是在OpenAI-GELU也就是GPU的Fast_GELU公式基础上，通过泰勒展开逼近得到的新公式，性能更快。

- 由于底层实现的差异，GPU上可以只支持对于非连续的tensor进行操作而不会产生额外性能问题，对于NPU则需要插入非连续转连续算子。因此GPU上某些代码运行会连续对一个非连续tensor进行操作，对NPU来说就会插入多余的转换算子造成性能下降。用户可以提前将非连续的tensor转连续，避免后续的额外转换算子插入。

目录

1. AI模型评估指标
2. AI模型调优介绍
- 3. 基于昇腾的模型调优**
 - NPU性能及精度调优
 - 调优案例

训练精度问题整体定位流程



- 在主流的迁移场景中，需统一优先进行训练前置定位步骤，提前校验checklist对基础配置、数据集、模型结构等基础问题的排除并保证精度问题可复现。

场景：LLaMA2-70B训练（1）

配置项	说明	数值
tensor-model-parallel-size	张量并行度	8
pipeline-model-parallel-size	流水线并行度	8
num-layers	层数	80
hidden-size	隐状态维度	8192
ffn-hidden-size	前馈网络隐状态维度	28672
num-attention-heads	注意力头的数量	64
normalization	归一化方法	RMSNorm
num-query-groups	查询组数量	8
micro-batch-size	微批大小	2
global-batch-size	全局批大小	1024
max-position-embeddings	最大输入序列长度	4096
attention_dropout	注意力层dropout	0.0
lr	学习率	1.5e-4
lr-decay-style	学习率衰减方式	cosine
min-lr	最低学习率	1e-5
weight-decay	权重衰减	1e-2 (过低)
clip_grad	修剪梯度	1
lr-warmup-fraction	学习率预热步数占比	0.01
adam-beta1	adam优化器梯度动量参数	0.9
adam-beta2	adam优化器梯度方差参数	0.999 (参数过大)
adam-eps	adam优化器极小值参数	1e-8

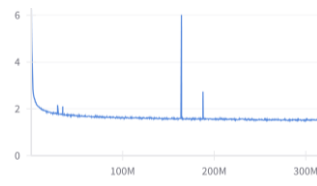
- 问题1：Loss spike。
 - 模型在训练过程中，遇到的最常见问题是loss尖刺。
- 分析可能原因：

异常数据分布导致模型在当前step的预测偏差增大引发loss激增。

训练参数的值超出精度表示范围（如混合精度训练时），引发loss异常。

超参数设置不合理（学习率、优化器）。

其他原因（模型结构、正则化等）。



- 此模型使用的学习率是1.5e-4，对4M token的全局batch来说，1e-4量级的基准学习率是合适的。对于1M token的全局batch来说，一般的基准学习率在1e-5量级。
- 在整个训练过程中，学习率会不断变化。
- 在初始阶段，优化方向波动比较大，训练不太稳定，此时学习率会从最低1e-5开始线性上升，称为warmup。warmup步数占总体训练步数的比例是0.01，如果训练2万步，则在200步时学习率达到基准的1.5e-4。之后通过线性或者余弦曲线逐步下降，以适应后期可能出现的训练不稳定，此模型使用的是余弦学习率衰减。
- 权重衰减作为正则方法的一种，通过调节模型复杂度对损失函数的影响，防止过拟合。
- 权重衰减0.01是参数值平方和的惩罚系数，也叫L2 penalty。在bfloat16混合精度训练中，权重衰减能够防止突然的loss发散。所以0.01的权重衰减容易引起尖刺，修改为0.1会有利于避免训练不稳定。

场景：LLaMA2-70B训练（2）

- 优化动作：

优化器参数调整

- 将 adam_beta2 从0.999降低到0.95后，大部分loss尖刺消失，剩余loss尖刺幅度也极大降低。

跳过当前batch

- 跳过当前数据，从上一checkpoint继续训练，后续loss尖刺小时进入稳定状态。

分析

- 数据问题：当前batch数据分布变动较大导致梯度异常；当前数据未清洗干净。
- 参数问题：优化器、学习率参数设置需要降低。

- Adam优化器广泛应用于大语言模型训练，主要有以下三个参数：
 - adam-beta1：影响梯度动量。
 - adam-beta2：影响梯度方差。
 - adam-eps：为了防止除0异常加入的一个极小值。

场景：LLaMA2-70B训练（3）

- 结论：

模型处于稳态后浅层梯度更新较小，当输入样本分布变化后导致浅层梯度变化较大

- 梯度裁剪
- 降低学习率（会导致训练速度降低）

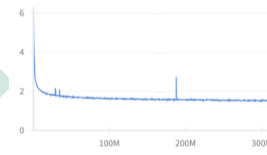
当前batch数据分布异常

- 数据清洗（乱码、脏数据等）
- 数据集的交叉打散混合，保证一个batch有多样的数据

优化器设置

$$u_i = \frac{m_i}{\sqrt{v_i + \epsilon}}$$

- 降低 ϵ ，但是合并低精度训练时可能会有问题
- 修改Adam更新向量u的计算方式。使用 $+\epsilon$ 的方法避免除0错误，而是当 $v=0$ 时，直接让 $u=0$



场景：多卡LLaMA训练（1）

- LLaMA 13B模型，在TP Size为8、DP Size为32输入的256卡集群上，当每个micro batch的loss同步4字节时，allReduce都会出现超长的Notify_Wait现象。
- 问题：集群等待时间过长。
- 分析定位：

MindStudio Insight
allReduce通信域图



MindStudio Insight
0卡通信域图

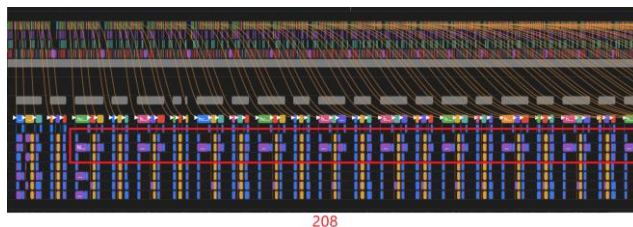
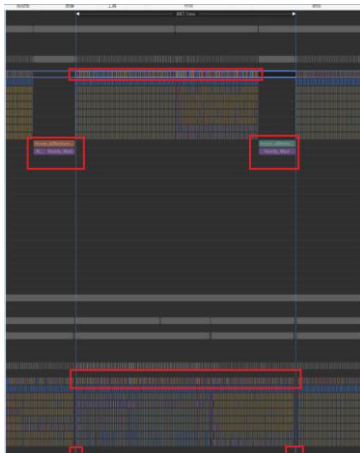


统计下发和计算耗时。
下发耗时最大0.5s，最小0.23s，差值0.27s。
计算耗时最大3.2s，最小3.11s，差值0.09s。
耗时的差距（总和300ms+）远小于allReduce的等待时间（800ms+）需要根据特定的allReduce算子进行单点分析。

- 由于卡数较多，先用MindStudio Insight工具导入256卡集群分析后的结果。根据TP Size和DP Size的输入，可以获取到所需allReduce通信域内各卡的计算/通信概览信息。

场景：多卡LLaMA训练（3）

- 找到最突出的rank 208卡，查看造成208卡速度缓慢的原因（观察第208卡的profiling信息）。

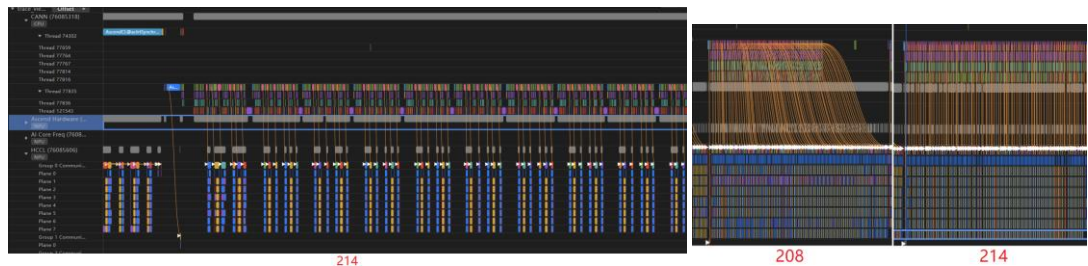


对比0卡（正常卡），208卡速度缓慢是由HCCL的TP通信导致。

- 通过查看通信矩阵，发现通信传输速度基本一致，主要还是由TP域内的慢卡导致。

场景：多卡LLaMA训练（4）

- 观察得出，某些流的通信算子的Notify_Wait总是较长，通过对端卡的dst rank可以看出是214卡，于是比较这两张卡的下发情况。



通过214卡的CANN侧下发，可以看到下发连线比较直，说明214卡是由下发导致通信算子的执行最晚。208卡相比214卡来说，下发就弯曲很多，说明没有下发影响通信算子的执行。
宏观来看，相比208卡，214卡的下发也明显更慢。
进一步分析发现，此卡下发慢的原因是某些ACL接口的耗时增加所导致。可以替换对应接口或开发更高效的算子。

总结

若不收敛或精度相差较大，多半和算子精度问题有关，推荐使用精度比对工具

若精度无法对齐，仅差1个点左右精度，可以尝试调参，调参的范围包括学习率、学习率策略、loss scale值、weight decay值等等

若精度波动较大收敛慢，可以少量调小LR试试

若train精度正常eval精度为0或较低，则优先排查在eval时是否正确执行了model.eval()

若单卡精度正常，多卡异常，考虑排查分布式权重初始化、数据切分处理等逻辑

思考题

1. （多选题）某工程师想要提升模型的训练速度，可以从以下哪些方向进行优化？

- A. Checkpoint的保存和加载
- B. 分布式系统的通信效率
- C. 集群网络拓扑
- D. 训练后期减少学习率适应高质量数据

- ABC

本章总结

- 本章节从两个维度介绍了模型在训练过程中的优化点和评估指标：精度层面，通过数据集处理、超参数配置、计算精度等方面提升模型效果；性能层面，通过减少IO次数、模型空间复杂度和提升通行效率加速模型训练；最后以昇腾训练平台为底座，介绍了NPU的优化方式。

学习推荐

- <https://www.hiascend.com/document/detail/zh/canncommercial/81RC1/quickstart/index/index.html>
- <https://www.hiascend.com/document/detail/zh/mindstudio/80RC1/index/index.html>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



AI计算系统基础知识



前言

- AI计算系统作为AI技术发展的重要驱动力，为当前AI模型的构建、训练和推理提供了极大的助力。
- 本课程将围绕AI计算系统的技术底座展开，介绍AI芯片的架构设计与关键指标，并探讨AI编译器的优化策略与实现原理。通过硬件基础、系统架构与软件工具的全链路解析，构建AI计算技术的完整认知框架。

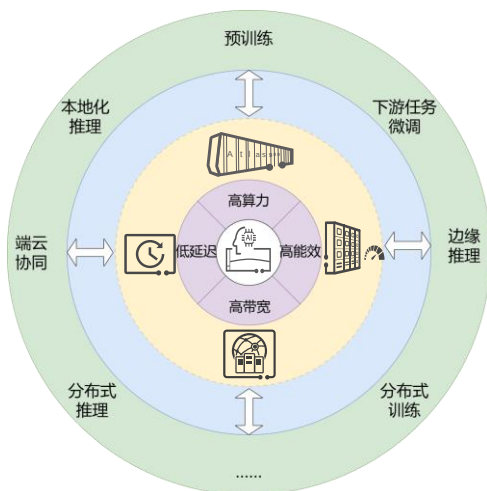
目标

- 学完本课程后，您将能够：
 - 从计算效率、能耗比等维度评估计算需求；
 - 熟悉AI编译器的工作原理；
 - 掌握AI计算系统的核心架构、AI芯片的硬件设计逻辑。

目录

1. AI计算系统介绍
2. AI芯片基础知识
3. AI计算体系与矩阵运算
4. 编译器
5. 昇腾AI计算系统

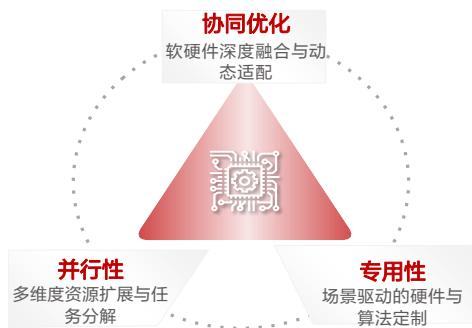
AI任务对于计算的需求



- 高算力：满足大规模模型训练和复杂计算任务的需求。
- 低延迟：确保实时推理和快速响应，特别是在边缘计算场景中。
- 高带宽：加速数据传输，提升分布式训练、推理的效率。
- 高能效：结合绿色计算技术，降低能耗，提升能效比。

- AI任务对计算资源的需求已从单一性能指标转向系统级优化，企业需根据任务规模选择云端或本地资源，结合模型压缩、智能调度等技术降低成本。未来，随着边缘计算与专用芯片的普及，AI算力将更贴近终端用户，同时绿色计算与通算融合将成为可持续发展的重要方向。

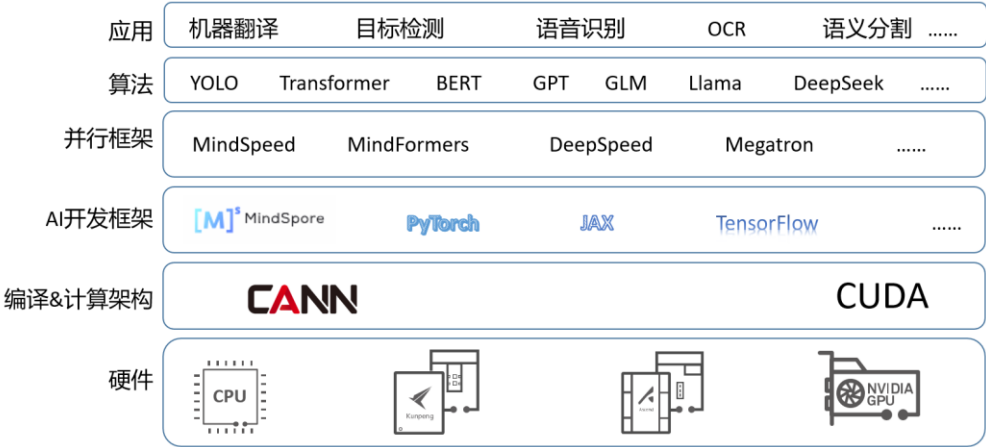
AI计算系统的设计原则



特点	描述
协同优化	通过软硬件深度融合（如框架-硬件协同设计）与动态资源调度（如弹性GPU分配），平衡性能、功耗与成本，构建高效系统。
并行性	采用数据/模型/混合并行策略，结合硬件级并行技术（如GPU SIMT），实现任务分解与资源扩展，加速大规模计算。
专用性	通过硬件定制（如TPU/NPU）与算法优化（如稀疏化、量化）深度适配场景需求，提升计算效率与能效比。

- AI计算系统的设计需以专用性、并行性、协同优化为基石，通过硬件定制、多维并行计算及软硬件深度协同，实现高效、低耗的智能计算。例如，自动驾驶系统通过专用AI芯片（如NPU）加速感知算法，结合混合并行训练提升模型精度，最终实现实时决策。未来，随着硬件架构创新和算法-硬件协同设计技术的发展，AI计算系统将向更高能效、更强适应性演进。

AI系统架构



目录

1. AI计算系统介绍
- 2. AI芯片基础知识**
3. AI计算体系与矩阵运算
4. 编译器
5. 昇腾AI计算系统

AI计算任务分类

Training（训练）

- 需要通过大量的数据输入，或采取增强学习等非监督学习方法，训练出一个复杂的深度神经网络模型。训练过程涉及海量的训练数据和复杂的深度神经网络结构，运算量巨大，需要庞大的计算规模，对于处理器的计算能力、精度、可扩展性等性能要求很高。常用的例如华为的Atlas 900集群、NVIDIA的GPU集群等。

Inference（推理）

- 利用训练好的模型，使用新的数据去“推理”出各种结论，如视频监控设备通过后台的深度神经网络模型，判断一张抓拍到的人脸是否属于特定的目标。虽然Inference的计算量相比Training少很多，但仍然涉及大量的矩阵运算。

- 训练芯片需要考虑的因素更多，设计上也更加复杂，精度通常为FP32、FP16
- 推理芯片考虑的因素较少，对精度要求也不高，INT8即可

AI芯片应用领域



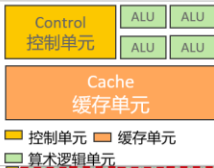
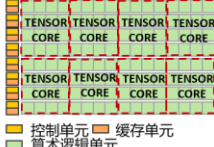
云端芯片部署在专业机房，对环境要求不高；边缘计算通常部署在户外，需要适应高温和低温环境；终端设备主要考虑功耗和成本

AI芯片技术路线

技术架构种类	定制化程度	可编辑性	算力	价格	优点	缺点	应用场景
GPU	通用型	不可编辑	中	高	通用性较强且适合大规模并行计算；设计和制造工艺成熟。	并行运算能力在推理端无法完全发挥。	高级复杂算法和通用性人工智能平台。
FPGA	半定制化	容易编辑	高	中	可通过编程灵活配置芯片架构适应算法迭代，平均性能较高；功耗较低；开发周期较短。	量产单价高；峰值计算能力较低；硬件编程困难。	适用于各种具体的行业。
ASIC	全定制化	难以编辑	高	低	通过算法固化实现极致的性能和能效、平均性能很强；功耗很低；体积小；量产后成本低。	前期投入成本高；研发时间长；技术风险大。	当客户处在某个特殊场景，可以为其独立设计一套专业智能算法软件。
类脑芯片	模拟人脑	不可编辑	高	-	功耗低；通信效率高；认知能力强。	目前仍处于探索阶段。	适用于各种具体的行业。

- 目前这几个技术方向都在发力，不存在某个技术方向被淘汰

通用计算 vs. AI计算：“分工”不同，共建多样性计算

		硬件结构抽象	计算特点	适用场景
通用计算 以CPU提供算力	鲲鹏	 Control 控制单元 Cache 缓存单元 ALU 控制单元 缓存单元 算术逻辑单元	适合复杂逻辑运算 比如大多数通用软件 70%以上晶体管用于构建Cache和控制单元 ，计算核心从几个到几十个。	通用应用 办公、数据库、数值计算（气象预测、流体仿真、电磁仿真）等。
AI计算 以GPU提供算力	NVIDIA AMD	 TENSOR CORE 控制单元 缓存单元 算术逻辑单元	适合逻辑简单 计算密集型高并发任务 70%以上晶体管用于构建计算单元 ，计算核心几千或上万个。	特定应用 图像识别：车牌识别、目标识别、目标检测； 自然语言处理：机器翻译、文本生成； 语言处理：语音识别、语音合成； 搜索推荐、辅助驾驶、趋势预测等。
AI计算 以NPU提供算力	昇腾	 达芬奇 CORE 控制单元 缓存单元 算术逻辑单元		



- CPU考虑通用计算，GPU考虑数学计算，NPU考虑AI需要的矩阵计算

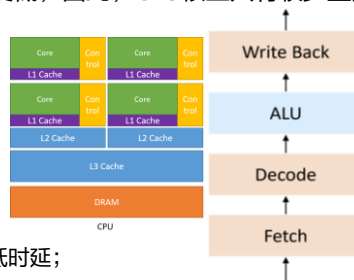
CPU结构

- CPU并不是针对计算密集型任务而设计的，其主要支持的计算是如何高效地执行通用程序。

- 为了能够灵活地支持不用类型的计算任务，一个CPU核上需要引入较为复杂的控制单元来优化针对动态程序的调度，由于一般的计算程序的计算密度低，因此，CPU核上只有较少量的算术逻辑单元（ALU）来处理数值类型的计算。

- 特点：

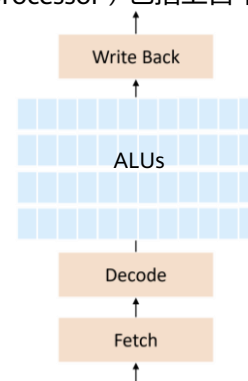
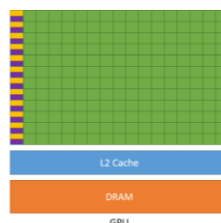
- 强大的ALU单元，可在很短时钟周期完成计算；
- 大量缓存降低延时；
- 高时钟频率；
- 复杂逻辑控制单元，多分支程序可通过分支预测能力降低时延；
- 对于依赖之前指令结果的部分指令，逻辑单元决定指令在pipeline中的位置实现数据快速转发。



- 在CPU核上执行一条计算需要经过一个完整的执行流水线，包括读取指令、译码、在ALU上计算结果和写回。
- 在整个流水线上，大量的时间和功耗都花在了ALU之外的非计算逻辑上。由于需要计算的数据一般存储在主存中，其访存延时往往和计算延时有数量级的差距，直接从主存中访问数据并计算会导致计算单元利用率极低，因此，为了加快访存速度，一般会在主存和寄存器之间加上多层缓存层。

GPU结构

- 与CPU相比，GPU的体系架构中的一个最大特点就是增加了大量的运算单元ALU。
- 通常GPU的一个处理器（也叫做流式多处理器，Streaming Multiprocessor）包括上百个简单的计算核，整个GPU可以达到上千个核。与CPU相比，每个核的结构简单，通常不具备CPU中的较为复杂的调度机制。
- GPU擅长计算密集和易于并行的程序。
 - 如深度学习中的大规模矩阵运算。



- 在执行指令的时候，为了充分利用每一次读取指令带来的开销，GPU会以一组线程为单位同时执行相同的指令，即SIMT（单指令多线程）的方式。
- 在CUDA GPU上，一组线程称为warp，一个warp有32个线程，每个线程执行相同的指令但访问不同的数据。

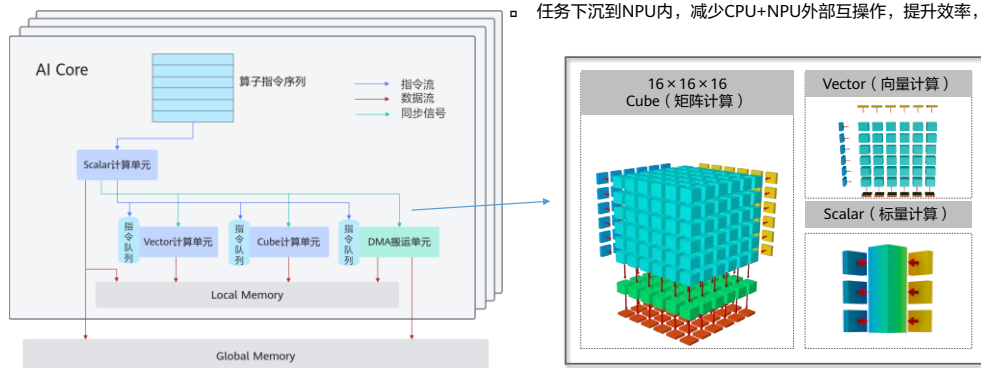
FPGA、ASIC比对

	FPGA	ASIC
运算速度	较低，FPGA结构上的通用性必然导致冗余；另外，不同结构间的时延也不可忽略。	较高，结构上无特殊限制，设计时也可将特定模块靠近减少延迟。
芯片规模	实现相同的功能时，需要更大的FPGA。	实现相同的功能时，ASIC的规模更小。
功耗	相同工艺条件下，功耗更大。	相同工艺条件下，功耗更小。
成本	研发风险较低，主要成本在单片上。	由于进入生产后硬件不可更改，开发工具和流片过程可能产生大量成本。
运行过程	加载配置进入存储器需要时间。	可立即运行。
产品定位	适用于项目产品需要灵活变动等方面的产品及产品要求快速占领市场的情况。	适用于设计规模较大，或应用于成熟的产品：如消费电子等。
发展方向	大容量、低电压、低功耗、SOC。	更大规模、IP复用技术、SOC。

- 两者目前发力的领域不同，并无明显的竞争关系

昇腾AI处理器：自研架构+高集成SoC设计

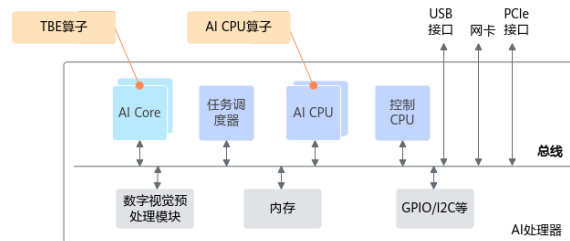
- 针对AI计算特点，包含16 X 16 X 16大Cube计算单元，矩阵乘计算效率高，达到极致面效比。
- 匹配AI应用场景，多维均衡考虑，应对“算”的任务，追求性能、功耗、成本的“三维匀称”
- CPU+NPU+网卡+编解码 高集成度SoC，极致能效
 - 任务下沉到NPU内，减少CPU+NPU外部互操作，提升效率，降低功耗



- DMA：负责数据搬运，包括Global Memory和Local Memory之间的数据搬运以及不同层级Local Memory之间的数据搬运。

昇腾AI处理器逻辑架构

- 昇腾AI处理器的主要架构组成：
 - 芯片系统控制CPU（Control CPU）
 - AI计算引擎（包括AI Core和AI CPU）
 - 多层级的片上系统缓存（Cache）或缓冲区（Buffer）
 - 数字视觉预处理模块（Digital Vision Pre-Processing, DVPP）等

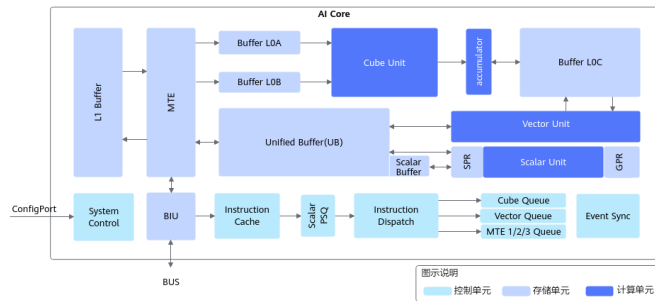


- 不同系列的芯片结构会略有差异

AI Core

- AI Core架构主要部分：

- 计算单元： 包含三种基础计算资源（矩阵计算单元、向量计算单元、标量计算单元）。
- 存储系统： AI Core的片上存储单元和相应的数据通路构成了存储系统。
- 控制单元： 整个计算过程提供了指令控制，相当于AI Core的司令部，负责整个AI Core的运行。



AI Core - 计算单元

- 三种基础计算资源：Cube Unit、Vector Unit和Scalar Unit，分别对应矩阵、向量和标量三种常见的计算模式。

计算单元	描述
Cube	Cube负责执行矩阵运算。 Cube每次执行可以完成一个fp16的16*16与16*16的矩阵乘，例如C=A*B，如果是int8输入，则一次完成16*32与32*16的矩阵乘。其中A来源于L0A，B来源于L0B，L0C存储矩阵乘的结果和中间结果。
Vector	Vector负责执行向量运算。 其算力低于Cube，但灵活度高于Cube（如支持数学中的求倒数，求平方根等）。 Vector所有计算的源数据以及目标数据都要求存储在Unified Buffer中，并要求32Byte对齐。
Scalar	Scalar负责各类型的标量数据运算和程序的流程控制。 功能上可以看做一个小CPU，完成整个程序的循环控制、分支判断、Cube/Vector等指令的地址和参数计算以及基本的算术运算等。

AI Core - 存储单元 (1)

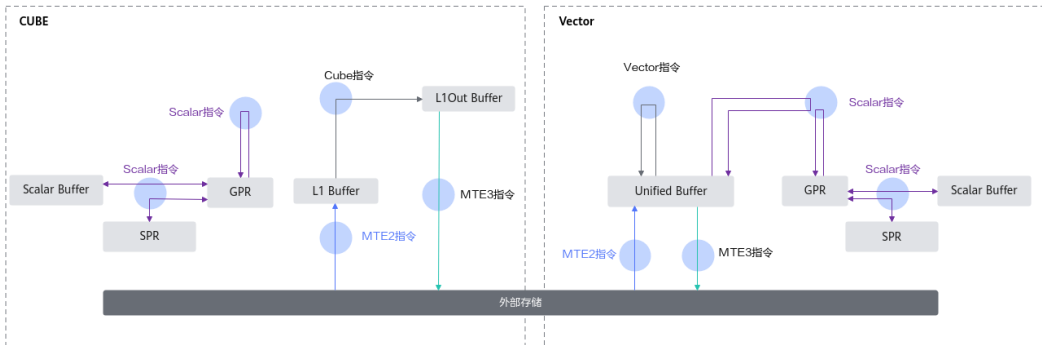
- AI Core中存在内部存储，AI Core需要把外部存储中的数据加载到内部存储中，才能完成相应的计算。AI Core的主要内部存储包括：L1 Buffer（L1缓冲区），L0 Buffer（L0缓冲区），Unified Buffer（统一缓冲区）和Scalar Buffer（标量缓冲区）。
- 为了配合AI Core中的数据传输和搬运，AI Core中还包含BIU（Bus Interface Unit，总线接口单元），MTE1（Memory Transfer Engine，存储转换引擎），MTE2，MTE3。其中BIU为AI Core与总线交互的接口；MTE为数据搬运单元，完成不同Buffer之间的数据搬运。
- AI Core的存储单元分为如下三类：
 - Cache：当指令指定访问下级存储单元的时候，Cache可以对数据进行缓存，从而加快访问速度。
 - Buffer：通常用于向量计算或者标量计算中的临时数据保存等。
 - Register：通常用于标量计算。

- Scalar Buffer对应硬件存储单元Scalar Buffer。
- Unified Buffer对应硬件存储单元Unified Buffer。
- L1 Buffer对应硬件存储单元L1 Buffer。
- L1Out Buffer为从L0C上抽象出来的存储Cube计算输出数据的存储单元。

AI Core - 存储单元 (2)

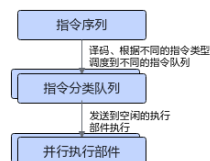
存储单元	描述
MTE	AI Core上有多个MTE (Memory Transfer Engine, 存储转换引擎), MTE负责AI Core内部数据在不同Buffer之间的读写管理及一些格式转换的操作, 比如填充 (padding)、转置 (transpose)、3D图像转2D矩阵 (Img2Col) 等。
BIU	BIU (Bus Interface Unit, 总线接口单元), AI Core的“大门”, 负责AI Core与总线的交互。BIU是AI Core从外部存储读取数据以及往外写数据的出入口, 负责把AI Core的读写请求转换为总线上的请求并完成协议交互等工作。
L1 Buffer	L1缓冲区, 通用内部存储, 是AI Core内比较大的一块数据中转区, 可暂存AI Core中需要反复使用的一些数据从而减少从总线读写的次数。 某些MTE的数据格式转换功能, 要求源数据必须位于L1 Buffer, 如3D图像转2D矩阵 (Img2Col)。
LOA Buffer / LOB Buffer	Cube指令的输入。
LOC Buffer	Cube指令的输出, 但在进行累加计算的时候, 也是输入的一部分。
Unified Buffer	统一缓冲区, 向量和标量计算的输入和输出。
Scalar Buffer	标量计算的通用缓冲区, 作为GPR (通用寄存器, General-Purpose Register) 不足时的补充。
GPR	通用寄存器 (General-Purpose Register), 标量计算的输入和输出。 应用开发工程师不需要具体关注这些寄存器。由系统内部实现封装, 程序访问Scalar Buffer并执行标量计算的时候, 系统内部自动实现Scalar Buffer和GPR之间的同步。
SPR	专用寄存器 (Special-Purpose Register), AI Core的一组配置寄存器。 通过修改SPR的内容可以修改AI Core的部分计算行为。

指令与存储访问关系



AI Core - 控制单元 (1)

- 控制单元为整个计算过程提供了指令控制，负责整个AI Core的运行。系统控制模块负责指挥和协调AI Core的整体运行模式，配置参数和实现功耗控制等。当指令通过指令发射模块顺次发射出去后，根据指令的不同类型，将会分别被发送到矩阵运算队列、向量运算队列和存储转换队列。
- 指令执行过程中，可以提前预取后续指令，并一次读入多条指令进入缓存，提升指令执行效率。多条指令从系统内存通过总线接口（BIU）进入到AI Core的指令缓存模块（Instruction Cache）中等待后续硬件快速自动解码或运算。指令被解码后便会被导入标量队列中，实现地址解码与运算控制。
- AI Core采用顺序取指令、并行执行指令的调度方式。



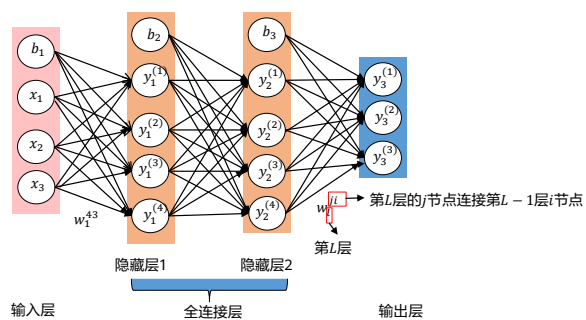
AI Core - 控制单元 (2)

控制单元	描述
系统控制模块 (System Control)	外部的Task Scheduler控制和初始化AI Core的配置接口，配置PC、Para_base、BlockID等信息，具体功能包括：Block执行控制、Block执行完之后中断和状态申报、执行错误状态申报等。
指令缓存模块 (Instruction Cache)	AI Core内部的指令Cache，具有指令预取功能。
标量指令处理队列 (Scalar PSQ)	Scalar指令处理队列。
指令发射模块 (Instruction Dispatch)	CUBE/Vector/MTE指令经过Scalar PSQ处理之后，地址、参数等要素都已经配置好，之后Instruction Dispatch单元根据指令的类型，将CUBE/Vector/MTE指令分别分发到对应的指令队列等待相应的执行单元调度执行。
矩阵运算队列 (Cube Queue)	Cube指令队列。同一个队列里的指令顺序执行，不同队列之间可以并行执行。
向量运算队列 (Vector Queue)	Vector指令队列。同一个队列里的指令顺序执行，不同队列之间可以并行执行。
存储转换队列 (MTE Queue)	MTE指令队列。同一个队列里的指令顺序执行，不同队列之间可以并行执行。
事件同步模块 (Event Sync)	用于控制不同队列指令（也叫做不同指令流水）之间的依赖和同步的模块。

目录

1. AI计算系统介绍
2. AI芯片基础知识
- 3. AI计算体系与矩阵运算**
 - 深度学习中的计算任务
 - 计算指标
 - 数据类型
 - 矩阵乘法实现
4. 编译器
5. 昇腾AI计算系统

深度学习与计算 (1)



- X 表示输入数据
- W 表示权重
- B 是激活神经元的阈值，在前馈神经网络里称为偏置
- Y 是神经元的输出，它表示了输入经过激活函数的输出结果

$$H = \text{ReLU}(XW1 + b1)$$

$$Y = \text{Softmax}(HW2 + b2)$$

深度学习与计算 (2)

$$H = \text{ReLU}(XW1 + b1)$$

$$Y = \text{Softmax}(HW2 + b2)$$

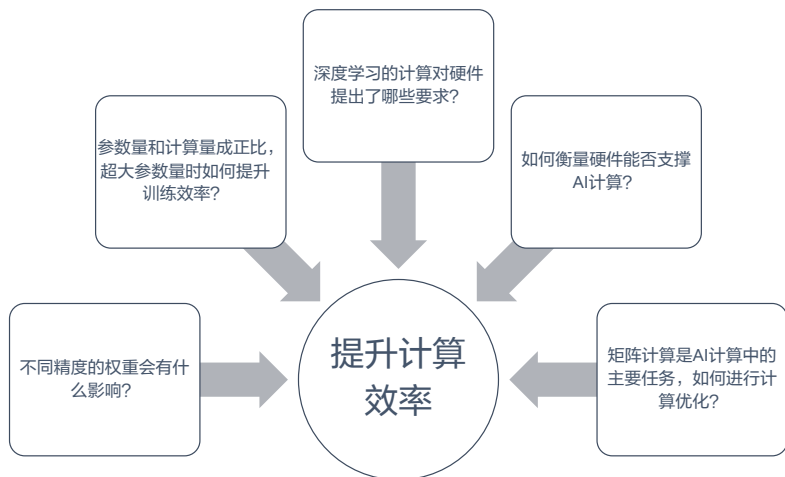
- X: 输入矩阵
- W: 权重矩阵
- b是激活神经元的阈值、偏置
- 计算任务中以矩阵计算为主

- 影响深度学习性能和精度的因素：
 - 参数量：W中的数值数量加上偏置项数量
 - 数据精度：W值的类型（单精度、半精度等）
 - 运算量：前、后项计算中累计的乘、加次数
 - IO次数：每次运算需要进行的读写操作

深度学习与计算 (3)

- 卷积层: $2 \times C_i \times k_w \times k_h \times C_o \times W \times H = [(C_i \times k_w \times k_h) + (C_i \times k_w \times k_h - 1) + 1] \times C_o \times W \times H$
 - 中括号内的值表示卷积操作计算出feature map中一个点所需要的运算量 (乘法和加法)
 - $C_i \times k_w \times k_h$ 表示一次卷积操作中的乘法运算量
 - $C_i \times k_w \times k_h - 1$ 表示一次卷积操作中的加法运算量
 - +1 表示bias, W和H分别表示feature map的长和宽
 - $C_o \times W \times H$ 表示feature map的所有元素数
- 全连接层: $2 \times I \times O = [I + (I - 1) + 1] \times O$
 - 中括号的值表示计算出一个神经元所需的运算量
 - 第一个I表示乘法运算量
 - $(I - 1)$ 表示加法运算量, +1 表示bias
 - O 表示计算O个神经元的值

思考



目录

1. AI计算系统介绍
2. AI芯片基础知识
- 3. AI计算体系与矩阵运算**
 - 深度学习中的计算任务
 - 计算指标
 - 数据类型
 - 矩阵乘法实现
4. 编译器
5. 昇腾AI计算系统

OPS

- OPS (Operations Per Second)，每秒运算次数。这里的“运算”可以是整数运算、浮点运算或混合型的运算，不局限于浮点运算。
 - 1TOPS：每秒进行一万亿（ 10^{12} ）次操作。
 - TOPS/W：1W功耗下处理器的运算能力指标。
 - 如Atlas 500 A2 智能小站算力为20 TOPS INT8。



- TOPS是Tera Operations Per Second的缩写。

FLOPS和FLOPs

- FLOPs (Floating Point Operations)，浮点运算数，可理解为计算量。它是用来衡量算法或模型复杂度的指标。
 - 以CNN为例（不考虑bias），单次卷积（卷积核尺寸为5*5、输入和输出特征图通道为10）计算量为2500FLOPs；全特征图（100*100）上计算量为：2.5*10⁶ FLOPs
- FLOPS (Floating Point Operations Per Second)，每秒浮点运算次数。它是衡量计算设备在每秒内执行的浮点运算次数的指标。
 - 如Atlas 200I DK A2浮点算力为4 TFLOPS FP16。



- 卷积计算量： $K \times K \times C_{in} \times C_{out} \times H_{out} \times W_{out}$ 。
 - 卷积核尺寸：K；
 - 特征图通道：C；
 - 输出特征图大小：H，W。

MAC和MACs

- MAC (Memory Access Cost)，即内存访问成本。衡量模型在推理或训练过程中内存访问开销。
- MACs是Multiply-Accumulate Operations的缩写，即乘加运算数。
 - 1 MACs 包含一个乘法操作与一个加法操作。

- MAC指模型在运行过程中从内存读取和写入数据的总开销，主要包括以下三部分：
 - 参数访问：模型权重的读取（如卷积核、全连接层的权重矩阵）。
 - 激活值访问：前向传播时输入/输出特征图的读写（如卷积层的输入/输出特征图）。
 - 梯度访问（训练时）：反向传播时梯度的读写（如权重梯度、输入梯度）。

其他指标

- 吞吐量（Throughput）：吞吐量是指芯片在单位时间内能够处理的数据量，对于需要处理大量数据的AI应用尤为重要。
 - 如当前大语言模型在一秒内可以输出的token数量。
- 时延（Latency）：芯片处理数据所需的时间，对于实时性要求高的AI应用（如自动驾驶、实时视频处理等）尤为重要。
- 能效（Energy Efficiency）：芯片在单位功耗下所能提供的算力，通常用TOPS/W（每秒万亿次操作每瓦）来衡量。

目录

1. AI计算系统介绍
2. AI芯片基础知识
- 3. AI计算体系与矩阵运算**
 - 深度学习中的计算任务
 - 计算指标
 - 数据类型
 - 矩阵乘法实现
4. 编译器
5. 昇腾AI计算系统

比特位

- 比特位（Bit），即Binary Digit的缩写，是计算机中最小的存储单位。
 - 比特位是计算机中的基本单位，其值可以表示为0或1，这两个数值代表了二进制系统中的两种基本状态。
 - 比特位是计算机存储和处理数据的最小单位，所有的数据和信息在计算机内部都是以比特位的形式进行存储和处理的。
 - 多个比特位可以组合起来表示更复杂的信息，如字节（Byte）就是由8个比特位组成的，能够表示更多的数据。
 - 如未压缩状态下，一张彩色图片存储在计算机中根据像素大小（1080P、32位）可计算出存储空间约为7.9MB

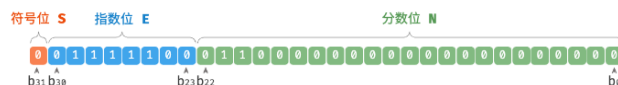
浮点类型（1）

- 浮点数在计算机中采用类似科学计数法的形式进行表示，即一个尾数（mantissa）乘以一个基数（通常为2）的指数（exponent）次幂。
 - 浮点数的表示需要三个要素：符号位、尾数和指数。
 - 浮点数又分为单精度（Float）和双精度（Double）。
 - 一个单精度浮点数（Float 32）通常由32位二进制组成，按照 IEEE 754 标准的定义，这32位被划分为三个部分：符号位、指数部分和尾数部分。
 - 符号位：占用 1 位，表示数值的正负。
 - 指数部分：占用 8 位，用于表示数值的阶码（数据范围）。
 - 尾数部分：占用 23 位，用于表示数值的有效数字部分（小数精度）。

- IEEE二进制浮点数算术标准（IEEE 754）是20世纪80年代以来最广泛使用的浮点数运算标准，为许多CPU与浮点运算器所采用。
- IEEE 754规定了四种表示浮点数值的方式：单精确度（32位）、双精确度（64位）、延伸单精确度（43比特以上，很少使用）与延伸双精确度（79比特以上，通常以80位实现）。只有32位模式有强制要求，其他都是选择性的。

浮点类型（2）

- FP32/float32，一般占用4bytes，表示范围为 $1.175e-38 \sim 2.403e38$ 。



- FP16，即半精度浮点数，是一种16位长的二进制浮点数表示方法。精度相对float32低，可能会出现精度下降问题，但可以减少存储空间和计算开销。



- BF16的特点在于其指数位与FP32相同，均为8位，但小数位（尾数位）减少到7位。这种设计使得BF16在保持一定精度的同时，大幅减少了存储需求和计算成本。



- 根据 IEEE 754 标准，32-bit长度的float由以下三个部分构成：
 - 符号位S：占1个bit，对应b31，表示正负，1代表负数，0代表正数。
 - 指数位E：占8bit，对应b30-b23，指数位宽。全0和全1有特殊用途，所以是00000001~11111110，也就是1到254，减去偏置127，指数部分最终范围为-126 ~127。
 - 分数位N：占23bit，对应b22-b0，尾数精度。范围为 $[(0 \sim 2^{23}) - 1] / 2^{23}$ 。
 - 这种数据类型适用于大多数科学计算和通用计算任务，尤其是在训练神经网络模型时，默认使用的数据类型就是单精度FP32。
- Half-precision floating-point(半精度浮点数)，在IEEE 754标准中是叫做binary16。
 - 符号位S：占1个bit，表示正负，1代表负数，0代表正数。
 - 指数位E：占5bit，指数位宽，全0和全1有特殊用途，所以是00001~11110，也就是1到30，减去偏置15，指数部分最终范围为-14 ~15。
 - 分数位N：占10bit，尾数精度，范围为 $(0 \sim 1023) / 1024$ 。
 - 在混合精度训练中，FP16和FP32的结合使用可以实现性能提升，既满足了精度的需求，又降低了计算成本。
- BF16的特点在于其指数位与FP32相同，均为8位，但小数位（尾数位）减少到7位。这种设计使得BF16在保持一定精度的同时，大幅减少了存储需求和计算成本。虽然

BF16的精度稍逊于FP32，但在许多深度学习应用中，尤其是计算机视觉和语音方面，这种精度折衷是可以接受的。

- 与FP16相比，BF16在数值范围上几乎与FP32相当，这有助于避免FP16容易出现的上溢和下溢问题。然而，BF16的尾数精度较小，这意味着在某些情况下，它可能不如FP16精确。但考虑到深度学习模型通常对指数的大小比对尾数敏感得多，因此BF16在实际应用中往往能提供与FP32相近的预测精度。
- 在性能上，BF16和FP16都具有较高的吞吐量，并且能显著减少内存需求。具体选择哪种格式取决于应用的具体需求和硬件支持情况。例如，NVIDIA的GPU对BF16有良好的支持，而Intel的CPU则更偏向于FP16。

FP8

- FP8有E4M3和E5M2两种设计，其中E5M2保持了FP16的数据范围，而E4M3则对数据精度有更好的支持。FP8数据类型可以在保持与FP16/BF16相似的模型精度下，节省模型的内存占用以及提升吞吐量。
 - E4M3：1位符号位、4位指数位、3位尾数位，提供较高精度。
 - E5M2：1位符号位、5位指数位、2位尾数位。动态范围更大（ ± 57344 ），但精度较低。

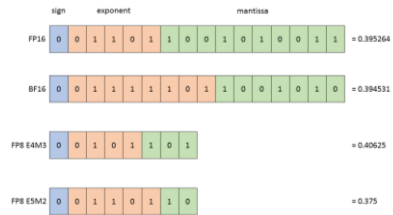
Table 1: Details of FP8 Binary Formats

	E4M3	E5M2
Exponent bias	7	15
Infinites	N/A	$S.11111.00_2$
NaN	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$
Zeros	$S.0000.000_2$	$S.00000.00_2$
Max normal	$S.1111.110_2 = 1.75 * 2^8 = 448$	$S.11110.11_2 = 1.75 * 2^{15} = 57,344$
Min normal	$S.0001.000_2 = 2^{-6}$	$S.00001.00_2 = 2^{-14}$
Max subnorm	$S.0000.111_2 = 0.875 * 2^{-6}$	$S.00000.11_2 = 0.75 * 2^{-14}$
Min subnorm	$S.0000.001_2 = 2^{-9}$	$S.00000.01_2 = 2^{-16}$

- https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html#Introduction-to-FP8

FP8

- 混合精度训练：
 - 前向传播：采用E4M3格式存储权重与激活值，3位尾数位保障计算精度，避免特征信息丢失。
 - 反向传播：切换至E5M2格式存储梯度，5位指数位覆盖梯度稀疏分布，防止小梯度值下溢。
- 优势：
 - 相比BF16，FP8在矩阵乘法等计算密集型操作上提供2倍峰值性能。
 - 训练时使用FP8可无缝部署到推理侧，无需额外量化步骤。



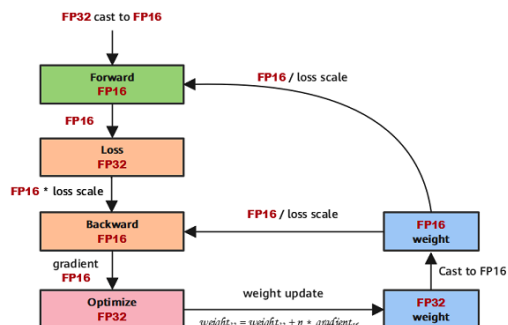
INT8/INT4

- INT8是一种8位有符号整数类型，可以表示范围在-128到127之间的整数。这种类型通常用于需要使用较小整数范围的应用。
- INT4是一种4字节有符号整数类型，符号占1位。表示范围很小，使用较少。
- 两种类型通常用于模型量化：
 - 以ChatGLM-6B为例，在FP16下需要 ≥ 13 G显存（CPU上大约要32G内存），在int4量化级别下最低只需要6G显存即可。

混合精度

• 混合精度训练流程：

- 参数以FP32存储；
- 正向计算过程中，遇到FP16算子，需要把算子输入和参数从FP32 cast成FP16进行计算；
- 将Loss层设置为FP32进行计算；
- 反向计算过程中，首先乘以Loss Scale值，避免反向梯度过小而产生下溢；
- FP16参数参与梯度计算，其结果将被cast回FP32；
- 除以Loss scale值，还原被放大的梯度；
- 判断梯度是否存在溢出，如果溢出则跳过更新，否则优化器以FP32对原始参数进行更新。



- 在使用混合精度获得训练加速和内存节省的同时，需要考虑FP16引入问题的解决。Loss Scale损失缩放，FP16类型数据下溢问题的解决方案，其主要思想是在计算损失值loss的时候，将loss扩大一定的倍数。根据链式法则，梯度也会相应扩大，然后在优化器更新权重时再缩小相应的倍数，从而避免了数据下溢。

• 流程：

- 将权重复制一份到FP16格式。
- 使FP16格式的权重和激活值进行前向传播。
- 将最终的Loss乘以缩放因子S。
- 使用FP16格式进行后向传播，包括权重、激活值及其对应的梯度。
- 若检测到权重梯度中出现Inf或NaN。
- 减小S值。
- 跳过当前权重更新步骤，重新开始下一个训练步骤。
- 将权重梯度乘以1/S。
- 梯度累积或者梯度累积足够步后使用FP32更新主权重。
- 如果之前N步都没有看到Inf或者NaN，增加S值。
- 在上述过程中，有几处计算必须要以FP32完成。比如主权重的更新，因为累加能够导致精度误差积累，所以必须要以FP32计算。缩放因子必须是FP32类型，

甚至 $1/S$ 的计算要将 S 转成双精度数求倒数再转回到FP32。

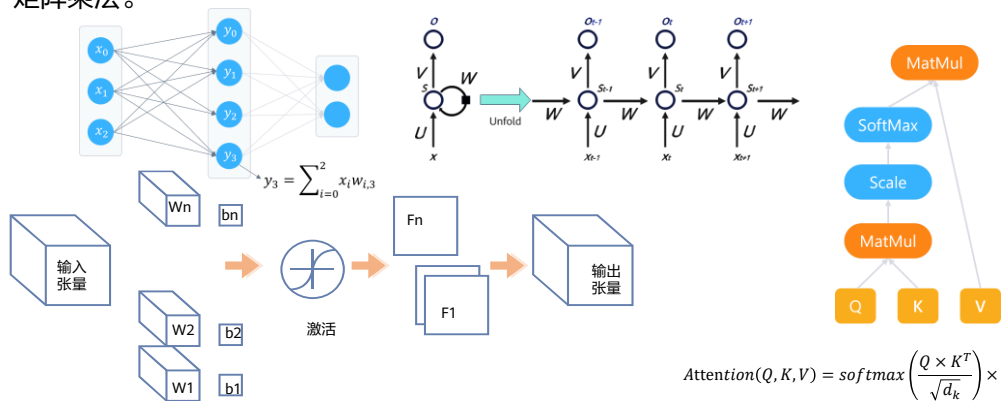
- 使用BF16格式的半精度数时，因为BF16有更大的表示范围，所以一般无需使用Loss缩放机制。但是BF16数值精度比FP16更差，所以在步骤三的第七点做梯度累积的时候需要使用FP32，否则有可能会因为梯度累积误差导致模型不收敛。另外BF16比FP16多15%的运行时内存，主要原因在于梯度累积时需要转FP32。

目录

1. AI计算系统介绍
2. AI芯片基础知识
- 3. AI计算体系与矩阵运算**
 - 深度学习中的计算任务
 - 计算指标
 - 数据类型
 - 矩阵乘法实现
4. 编译器
5. 昇腾AI计算系统

深度学习中的矩阵运算

- 深度学习模型的共同特点：大部分模型结构的核心计算模式都可以直接或间接的表示为矩阵乘法。

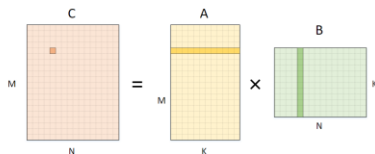


$$Attention(Q, K, V) = softmax\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) \times V = Z$$

矩阵乘法的简单实现

- 矩阵乘法相关的计算过程表现为较高的内存搬移和计算密度需求。

- 矩阵乘法的定义为： $C = A * B$
- 朴素矩阵乘法的时间复杂度为 $O(n^3)$ 。



A.shape=(M,K)
B.shape=(K,N)
计算: A x B

```
for (int m = 0; m < M; m++) {  
    for (int n = 0; n < N; n++) {  
        C[m][n] = 0;  
        for (int k = 0; k < K; k++) {  
            C[m][n] += A[m][k] * B[k][n];  
        }  
    }  
}
```

FLOPs: 2MNK
内存访问次数: 4MNK

- A、B矩阵在计算过程中各读取一次
- C矩阵读写各一次

矩阵乘法的问题

- 时间复杂度高，且数据IO频繁。
- 在神经网络中参与矩阵乘法的矩阵往往维度非常大，而芯片里的内存空间往往是有限的（成本高），表现为越靠近计算单元，带宽越快，内存越小。
- 优化思路：
 - 算法优化，从数学角度优化矩阵计算的中间过程。
 - 实现过程优化，优化矩阵计算时和硬件的交互。

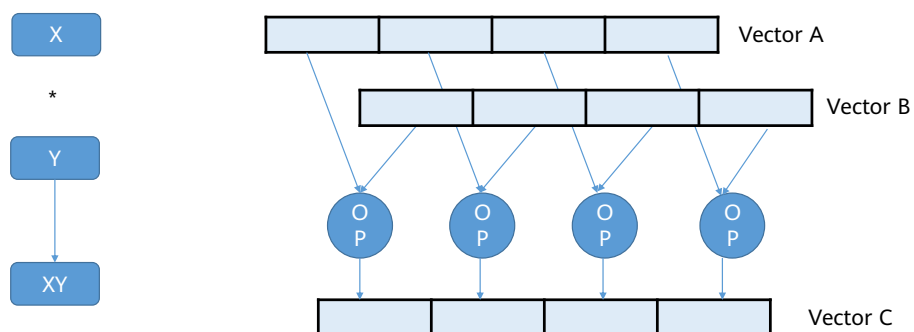
矩阵计算优化方式

	优化方式	实现过程
实现优化	基本的循环优化	通过调整循环顺序、内存布局等手段，减少缓存未命中（cache miss）和数据依赖，提高缓存利用率，从而加速矩阵乘法运算。
	分块矩阵乘法	将大矩阵划分成小块，通过对小块矩阵进行乘法运算，降低了算法的时间复杂度，并能够更好地利用缓存。
	SIMD 指令优化	利用单指令多数据（SIMD）指令集，如 SSE（Streaming SIMD Extensions）和 AVX（Advanced Vector Extensions），实现并行计算，同时处理多个数据，提高计算效率。
	SIMT 多线程并行化	利用多线程技术，将矩阵乘法任务分配给多个线程并行执行，充分利用多核处理器的计算能力。
算法优化	算法改进	如 Strassen 算法、Coppersmith-Winograd 算法等，通过矩阵分解和重新组合，降低了算法的时间复杂度，提高了计算效率。

SIMD介绍

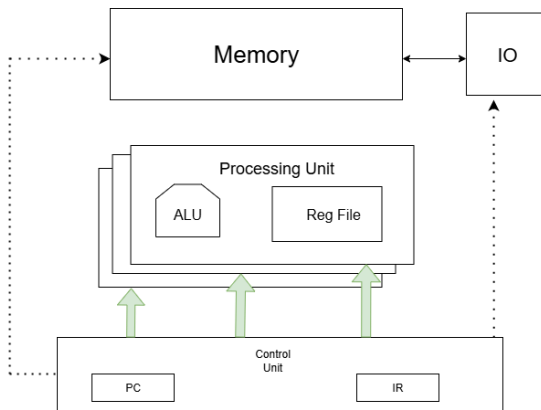
- SIMD (Single Instruction Multiple Data) 指令优化：利用单指令多数据指令集实现并行计算，提高计算效率。
 - SIMD允许一条指令同时操作多个数据元素。在SIMD中，一条指令会被发送到多个处理单元，这些处理单元同时对它们所管理的数据进行操作。
 - SIMD是单线程，不是多线程操作，硬件上仅需要一个计算核心，只不过一次操作多个数据。
 - 由于其ALU宽度的限制，其计算是一定受到限制的，进行数据并行性较低。

SIMD数据流 (1)



- SIMD就是相应的寄存器中存储多个数据，然后GPU同时从中取出来，同时进行操作。

SIMD数据流 (2)



- 为了使SIMD实现一次乘法可以完成多个元素的计算，要求硬件上增加 ALU单元的数量，因此会有多个处理单元（Process Unit），同时也需要增加功能单元的数据通路数量，由控制单元（Control Unit）将数据传送给Process Unit，从而实现在单一时钟周期内整体上提升硬件的计算吞吐量。

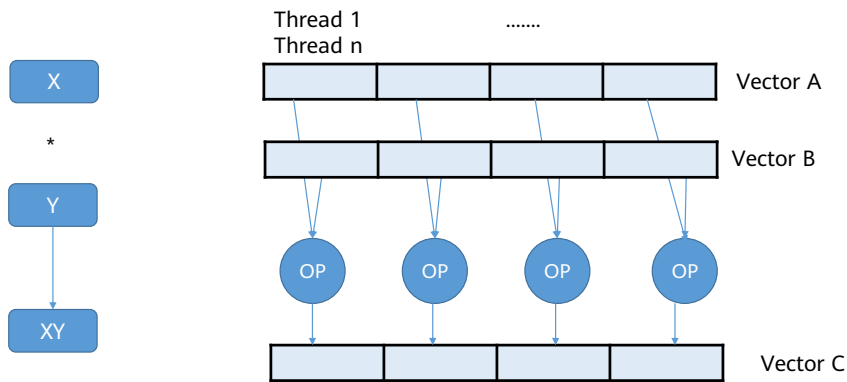
- 缺点：SIMD 使用独立线程，该线程能同时进行多个数据元素的计算，但是由于 ALU 宽度的限制，因此计算时要求数据类型、格式、大小必须严格对齐。
- 优点：在一定程度上可以提升计算性能，利用内存数据总线带宽，多个数据可以同时从内存读和写。

SIMT (1)

- SIMT (Single Instruction Multiple Threads, 单指令多线程)：是英伟达提出基于GPU的概念。与SIMD不同在于SIMD要求所有的向量元素在统一的同步组里（一个线程内）同步执行，而 SIMT 允许多个线程在一个Warp（32个线程）中独立执行。
 - SIMT 类似 CPU 上的多线程，有多个计算核心系统，每一个核心中有独立的寄存器文件（ Register File, RF ）、计算单元（ Arithmetic Logic Unit, ALU ），但是没有独立指令缓存（ Instruction Cache ）、解码器、程序计数器（ Program Counter register ），命令从统一的指令缓存广播给多个 SIMT 核心。因此 SIMT 的所有核心各自独立，在不同的数据上执行相同的计算操作，即执行命令相同，多个线程各有各的处理单元，SIMD 则是共用同一个 ALU。

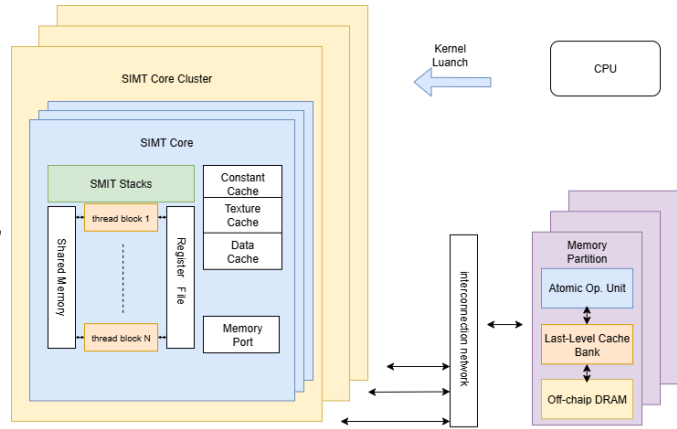
- <https://developer.nvidia.com/zh-cn/blog/cuda-hardware-cn/>

SIMT (2)



SIMT (3)

- 具体到 SIMT 的硬件结构，SIMT 提供一个多核系统（SIMT Core Cluster），CPU 负责将算子（Kernel）加载到 SIMT Core Cluster 中，每个 SIMT 核（SIMT Core）有独立的 RF（Register File）、ALU、Data Cache，但是只有一个指令计数寄存器（Program Counter）和一个指令译码寄存器，指令被同时广播给所有的 SIMT 核，从而执行具体的计算。GPU 则是由多个 SIMT Core Cluster 组成，每个 SIMT Core Cluster 由多个 SIMT Core 构成，SIMT Core 中有多个 Thread Block。



SIMD和SIMT

	SIMT	SIMD
原理	原理相同，都是采用单指令多数据的思想。	
执行粒度	以线程为粒度，每个线程独立执行指令流，支持分支和差异化控制流（如条件判断、循环）	以数据为粒度，所有处理单元同步执行相同指令流，无原生分支能力，需通过手动管理数据对齐
硬件	需管理线程状态堆栈、分支预测及动态掩码机制，硬件复杂度高（如NVIDIA的SIMT堆栈）	控制单元统一调度，处理单元仅需执行数据操作，硬件实现简单
适用场景	复杂逻辑并行（如GPU加速的深度学习、图形渲染），适合不规则并行任务	数据密集型任务（如CPU向量优化的图像处理、信号处理），适合规则化、高吞吐量任务

目录

1. AI计算系统介绍
2. AI芯片基础知识
3. AI计算体系与矩阵运算
- 4. 编译器**
5. 昇腾AI计算系统

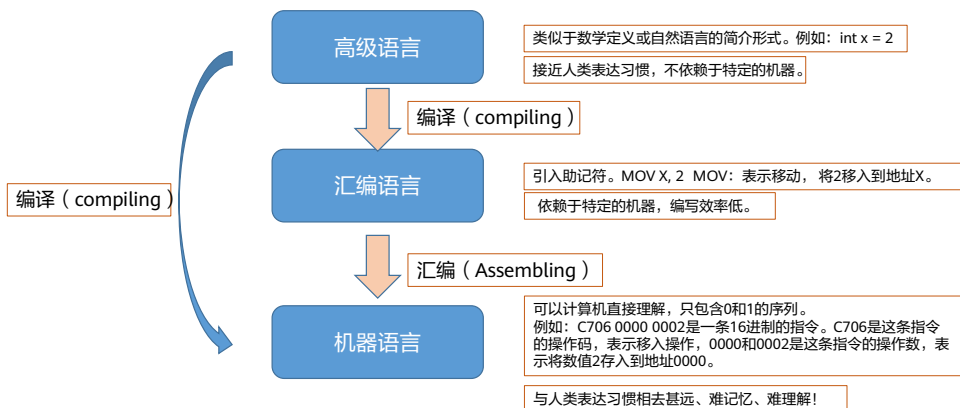
什么是编译器

- 编译器（compiler）是一种计算机程序，它会将某种编程语言写成的源代码（原始语言）转换成另一种编程语言（目标语言）。



计算机程序设计语言及编译

编译：将高级语言（源语言）翻译成汇编语言或机器语言（目标语言）的过程。



编译器的结构



- 前端：把源程序分解成组成要素，以及相应的语法结构 – 使用这个结构创建源程序的中间表示 – 同时收集和源程序相关的信息，存放到符号表。
- 后端：根据中间表示和符号表信息构造目标程序。

编译器工作原理 - 词法分析

- 词法分析/扫描:

- 从左向右逐行扫描源程序的字符，识别出各个单词，确定单词的类型。
- 将识别出的单词转换成统一的机内表示
 - 词法单元 (token) 形式。
- token: < 种别码, 属性值 >
 - 种别码表示单词的种别
 - 属性值是token被赋予的值

类型	种别码	种别
关键字	一词一码	program、if、else、then、... (每个关键字独立编码)
标识符	多词一码	变量名、数组名、记录名、过程名 (所有标识符共享同一编码)
常量	一型一码	整型、浮点型、字符型、布尔型 (所有常量类型共享同一编码)
运算符	一型一码或一词一码	算术 (+、-、*、/)、关系 (>、<、==)、逻辑 (&、 、~)
界限符	一词一码	;() = { } ... (每个界限符独立编码)

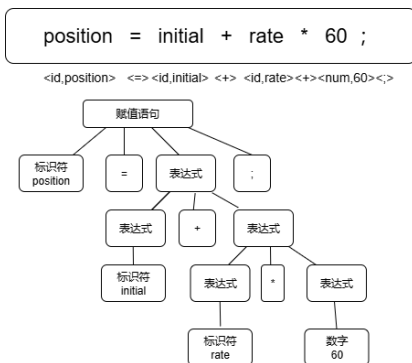
词法分析后得到的token序列

- 输入一段c++代码，下面是词法分析后得到的token序列。

输入	while(value!=100){num++;}					
	1	while	<	WHILE	,	- >
	2	(	<	SLIP	,	- >
	3	value	<	IDN	,	- >
	4	!=	<	CONST	,	- >
	5	100	<	SRP	,	- >
输出	6	)	<	LP	,	- >
	7	{	<	IDN	,	- >
	8	Num	<	IDN	,	- >
	9	++	<	INC	,	- >
	10	;	<	SEMI	,	- >
	11	}	<	RP	,	- >

语法分析

- 语法分析器（parser）从词法分析器输出的token序列中识别出各类短语，并构造语法分析树（parse tree）。它描述了语句的结构。
- 例：赋值语句的分析树：



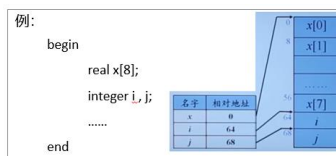
语义分析

- 收集标识符的属性信息

- 种属
- 类型
- 存储位置、长度
- 值
- 作用域
- 参数和返回值信息
 - 参数个数、参数类型、参数传递方式、返回值类型、.....

- 语义检查

- 变量或过程未经声明就使用、重复声明
- 运算分量类型不匹配
- 操作符与操作数之间类型不匹配
-



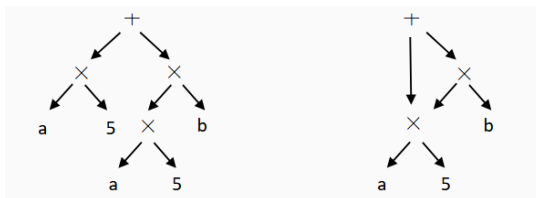
存储位置、长度信息收集示例

中间表示 (Intermediate Representation)

- 编译器在做完前端工作以后，首先就生成IR,并在此基础上执行各种优化算法，最后再生成目标代码。

- 形式

- 多种中间表示，不同层次
- 抽象语法树
- 三地址代码



$a*5+a*5*b$

- 重定位

- 为新的机器建编译器，只需要做从中间表示到新的目标代码的翻译器（前端独立）。

- 高层次的优化

- 优化，与源语言和目标机器都无关（如常量替换，死代码消除，循环优化等）。

- 常用线性中间表示有堆栈机代码（Stack-Machine Code）和三地址代码（Three Address Code）。堆栈机代码是一种单地址代码，提供了简单紧凑的表示。堆栈机代码的指令通常只有一个操作码，其操作数存在一个栈中。大多数操作指令从栈获得操作数，并将其结果推入栈中。三地址代码，简称为3AC，模拟了现代RISC机器的指令格式。它通过一组四元组实现，每个四元组包括一个运算符和三个地址（两个操作数、一个目标）。
- 图中间表示将编译过程的信息保存在图中，算法通过图中的对象如节点、边、列表、树等来表述。虽然所有的图中间表示都包含节点和边，但在抽象层次、图结构等方面各有不同。常见的图中间表示包括抽象语法树（Abstract Syntax Tree, AST）、有向无环图（Directed Acyclic Graph, DAG）、控制流图（Control-Flow Graph, CFG）等。AST抽象语法树采用树型中间表示的形式，是一种接近源代码层次的表示。

代码生成

- 代码生成是根据中间表示（IR）生成不同平台的机器码，在代码生成器之前可能有一个优化组件。
- 代码生成器的三个任务
 - 指令选择：选择适当的指令实现IR语句
 - 寄存器分配和指派：把哪个值放在哪个寄存器中
 - 指令排序：按照什么顺序安排指令执行



JIT和AOT编译方式

- 程序主要有两种运行方式：
 - 静态编译：程序在执行前全部被编译为机器码，称为 AOT（Ahead of time），即“提前编译”。
 - 动态解释：程序边编译边运行，通常将这种类型称为 JIT（Just in time），即“即时编译”。

	Just in time	Ahead of time
优点	1. 可以根据当前硬件情况实时编译生成最优机器指令 2. 可以根据当前程序的运行情况生成最优的机器指令序列 3. 当程序需要支持动态链接时，只能使用JIT 4. 可以根据进程中内存的实际情况调整代码，使内存能够更充分的利用	1. 在程序运行前编译，可以避免在运行时的编译性能消耗和内存消耗 2. 可以在程序运行初期就达到最高性能 3. 可以显著的加快程序的启动
缺点	1. 编译需要占用运行时资源，会导致进程卡顿 2. 编译占用运行时间，对某些代码编译优化不能完全支持，需在流畅和时间权衡 3. 在编译准备和识别频繁使用的方法需要占用时间，初始编译不能达到最高性能	1. 在程序运行前编译会使程序安装的时间增加 2. 牺牲高级语言的一致性问题 3. 将提前编译的内容保存会占用更多的外部空间

什么是AI编译器

- AI编译器是一种针对AI和机器学习应用特别优化的编译器，它能够满足推理场景和训练场景不同需求，将高级语言编写的程序或者训练好的模型文件转换成可以在特定硬件上高效执行的程序。

以 Python 语言
为前端

拥有多层IR设计

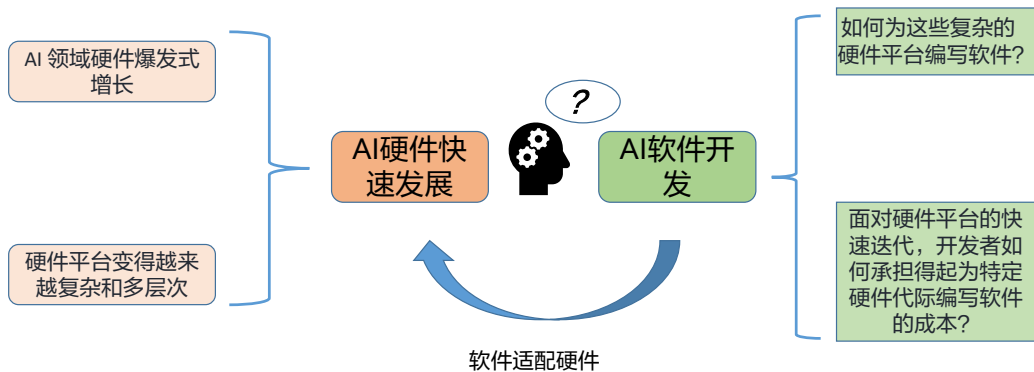
面向神经网络
深度优化

针对 DSA 芯片
架构

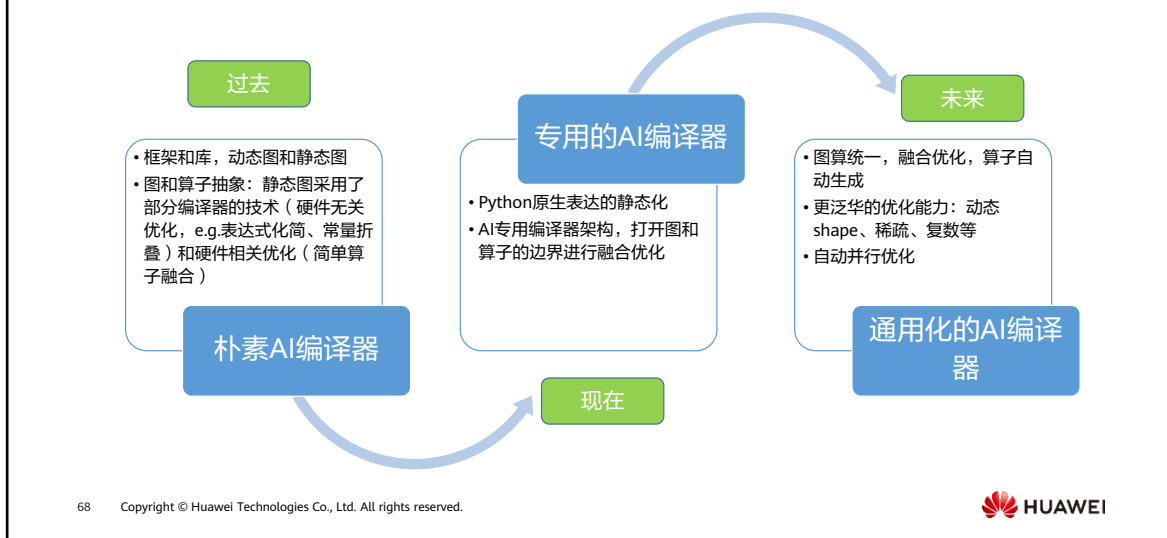
- 推理场景：输入 AI 框架训练出来的模型文件，输出能够在不同硬件高效执行的程序。
- 训练场景：输入高级语言表示的神经网络代码，输出能够在不同硬件高效执行的程序。

- AI 编译器拥有以Python为主的动态解释器语言前端，Python 是一种广泛用于 AI 领域的高级编程语言，因其简洁的语法和强大的库支持而受到欢迎。动态解释器语言前端指的是编译器在处理 Python 代码时，会进行动态类型检查和解释执行，这有助于在运行时捕获和处理类型错误。
- AI 编译器拥有多层IR设计，包括图编译 IR、算子编译 IR、代码生成IR等。IR（Intermediate Representation，中间表示）是编译过程中的一个抽象层次，它将高级语言代码转换为一种中间形式，以便进行进一步的优化和转换。图编译指的是将程序表示为计算图，这是一种数据结构，用于表示程序中的操作和它们之间的依赖关系。算子编译是指将计算图中的操作（算子）转换为特定硬件或平台的优化代码。代码生成是将优化后的IR转换为目标平台的代码，这一步通常涉及到寄存器分配、指令调度等优化技术。

为什么需要AI编译器？



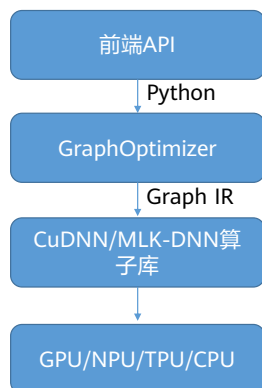
AI编译器的的发展



- AI 编译器的第一个阶段，我们可以将其看作是朴素 AI 编译器阶段。它主要存在于 TensorFlow 早期版本，基于神经网络的编程模型，主要进行了graph图和ops算子两层抽象。
- AI框架的早期主要是两个抽象，一个是基于张量的计算图，分为图和算子；另外一个动态图和静态图。动态图基本上和AI编译器没有太多关联，静态图采用了部分编译器的思想，比如图优化的时候会使用一些硬件无关的优化（表达式化简/常量折叠等）、硬件相关的优化（手工的算子融合）。
- 问题：
 - 表达上：静态图的表达式非Python原生，开发者主要通过框架提供的API进行显式构图，易用性上不好；
 - 性能上：开发者定义的算子粒度未必最能发挥硬件的性能；硬件厂商的提供的缺省的算子库也未必是最优的，在模型和shape确定的情况下，可能还有更优的算子实现；DSA芯片出现加剧了性能上的挑战
- 在 AI 编译器的发展中，阶段二标志着专用 AI 编译器的诞生，这一阶段的编译器开始针对 AI 和深度学习工作负载进行优化。
- PyTorch 框架以其动态图（也称为即时执行模式）而受到欢迎，它允许开发者以 Python 原生的方式编写和修改神经网络模型。这种灵活性甚至成为了当前所有 AI 框架设计的参考标准，促使编译器开发者考虑如何将类似 PyTorch 的表达方式转换为优化的中间表示（IR）。

- 尽管动态图提供了灵活性，但在性能关键的应用中，静态图的优化潜力更大。因此，编译器需要能够将动态图转换为静态图，以便于进行进一步的优化。同时，PyTorch 引入了 AI 专用编译器架构，打开图算边界进行融合优化，能够更好地处理 AI 工作负载的特点，自动管理图和算子的边界。
- 在阶段二专用 AI 编译器之后，就是 AI 编译器发展阶段三的到来。阶段三代表着通用 AI 编译器的重要发展阶段。阶段三的通用 AI 编译器代表了 AI 编译器技术的进一步成熟和进步。目前工业界的发展还处于 AI 编译器发展阶段的阶段二，也就是专用 AI 编译器的发展阶段。
- 但展望未来，阶段三的通用 AI 编译器将带来一系列创新特点，成为发展目标：
- 统一表达：在阶段三，通用 AI 编译器将实现计算图（Graph）和算子（Operators）的统一表达。这一创新意味着编译器能够在统一的框架下，同时进行图级别的优化和算子级别的优化，提升了编译过程的效率和效果。
- 自动化优化：通用 AI 编译器将在算子层实现自动调度（Schedule）、自动分块（Tiling）和自动代码生成（CodeGen），从而大幅降低开发难度，提高开发效率。
- 泛化优化能力：通用 AI 编译器将具备更广泛的优化能力，包括动静统一、动态形状（Shape）处理、稀疏性优化、高阶微分以及自动并行化等高级特性。
- 模块化设计：通用 AI 编译器将编译器本身、运行时系统、异构计算支持以及从边缘设备到数据中心的部署需求划分为独立的、可重用和可组合的模块。这种设计不仅增强了系统的灵活性和可扩展性，使其能够适应多样化的硬件架构和计算环境，而且通过提供简单直观的 API、详尽的文档和强大的工具支持，显著降低了开发难度，加快了开发速度，使得 AI 应用的部署和维护变得更加容易和高效。

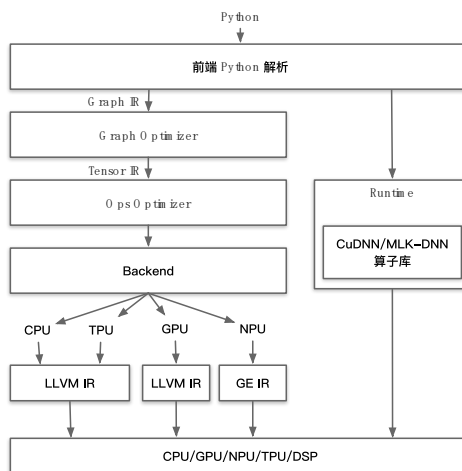
朴素AI编译器



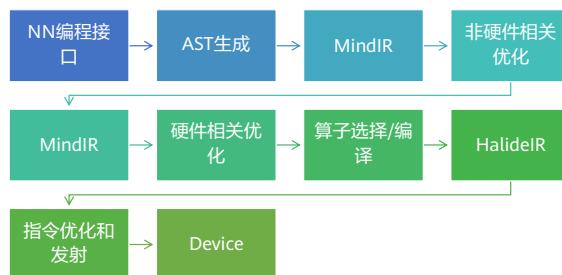
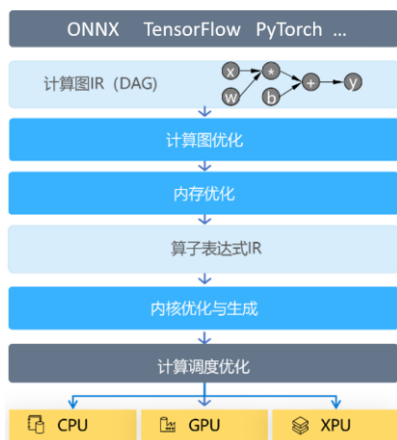
- TensorFlow早期版本，基于神经网络的编程模型，主要进行了graph图和ops算子两层抽象。

专用的AI编译器

- 编译器前端对Python代码进行解析。
- 解析器将高层次的代码转换为一个中间表示（IR）。
 - 这里编译器前端会生成Graph IR传递给Graph Optimizer（图优化器）。
- Graph Optimizer接收到Graph IR后，会对解析后的计算图进行优化。优化的目的是减少计算图的冗余部分，提高执行效率。包括算子融合、常量折叠等优化。Graph Optimizer 在优化完成后会向Ops Optimizer（操作优化器）传递一个Tensor IR。
- Ops Optimizer 接收到 TensorIR后，其会针对每个算子进行具体的性能优化，例如重排计算顺序、内存优化等。所有的中间表达都传递至后端之后，后端会生成不同的硬件代码以及可执行程序。



AI编译器的通用架构



MindSpore编译器处理流程

- 由于深度学习计算任务在现有的计算框架中往往以DSL（Domain Specific Language）的方式进行编程和表达，这本身使得深度学习计算任务的优化和执行天然符合传统计算机语言的编译和优化过程。因此，与传统程序语言编译器类似，深度神经网络编译器的提出主要是解决多种设备**适配性**和**性能优化**的问题。具体来说，深度神经网络编译就是将当前的深度学习计算任务通过一层或多层中间表达进行翻译和优化，最终转化成目标硬件上的可执行代码的过程。

主流AI编译器

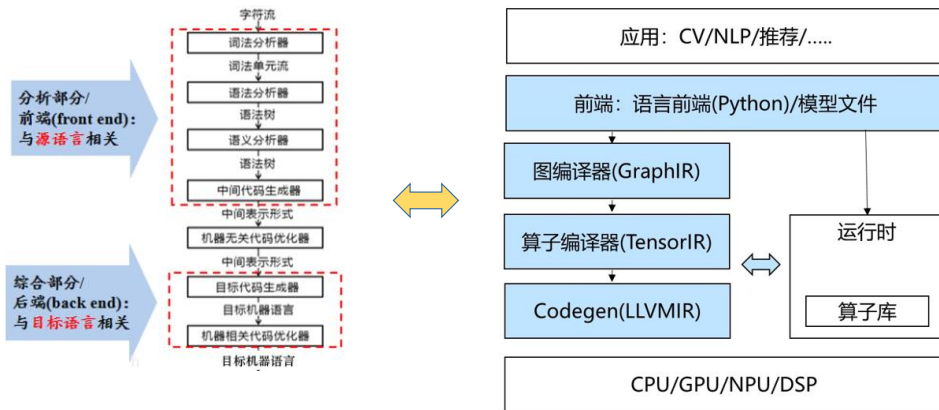
编译器	特点
TVM (Tensor Virtual Machine)	开源深度学习编译器堆栈，特点：跨平台支持、自动调优、高效编译（优化深度学习模型的执行效率，支持ONNX、TensorFlow、PyTorch等模型格式）。
TensorFlow XLA (Accelerated Linear Algebra)	XLA是Google为TensorFlow开发的编译器，特点：操作融合、硬件优化、静态编译。
Glow (Graph Lowering Compiler)	Glow是由Facebook开发的深度学习编译器，特点：图优化、硬件加速、自动优化）。
MLIR(Multi-Level Intermediate Representation)	MLIR是由Google开发的一种通用编译基础设施。特点：模块化设计、可扩展性强、跨平台支持、提供与LLVM的无缝集成、多语言支持。
nGraph	英特尔推出的nGraph是一款开源的深度学习系统编译器，nGraph可以接收任何深度学习框架的模型作为输入，支持多种硬件平台，提供了丰富的优化选项，包括张量融合、计算图优化、向量化等。

- TVM是由DMLC（Distributed Machine Learning Community）开发的一个开源深度学习编译器堆栈，旨在为各种硬件后端提供高效的深度学习模型编译和执行支持，广泛应用于学术界和工业界。特点：跨平台支持（支持从嵌入式设备到数据中心的多种硬件平台）、自动调优（通过自动化调优技术AutoTVM，为不同硬件生成最优的执行代码）、高效编译（优化深度学习模型的执行效率，支持ONNX、TensorFlow、PyTorch等模型格式）。
- XLA是Google为TensorFlow开发的编译器，旨在通过优化和代码生成技术加速TensorFlow模型的执行，已成为TensorFlow生态系统中的关键组件，显著提升了模型的执行效率。特点：操作融合（通过融合多个操作聚类优化为更大内核来减少内存访问和提高计算效率）、硬件优化（针对特定硬件平台TPU、GPU进行优化，生成高效的执行代码）、静态编译（将动态计算图转换为静态图，以实现更高效的执行）。
- Glow是由Facebook开发的深度学习编译器，旨在优化机器学习模型在各种硬件上的执行效率，Glow已经被集成到许多深度学习框架中，而且是PyTorch框架的一部分，用于高效地执行模型推理，多家芯片Cadence/QCOM/Intel/Marvell支持Glow。特点：图优化（通过图优化技术，将高层次的计算图转换为高效的底层代码）、硬件加速（支持多种硬件加速器，包括CPU、GPU和定制ASICs）、自动优化（使用AI技术进行性能分析和优化，提升模型执行效率）。
- MLIR是由Google开发的一种通用编译基础设施，旨在简化编译器和代码生成器的开发。它提供了一个多层次的中间表示，用于表示和优化程序代码。特点：模块化设计（支持多层次的中间表示，从高级抽象到低级细节）、可扩展性强（允许开发者定义和集成自定义的中间表示和转换）、跨平台支持（设计用于支持多种硬件平台，包括

CPU、GPU、FPGA等）、提供与LLVM的无缝集成（利用LLVM强大的代码生成和优化能力）、多语言支持（可作为多个编程语言和框架的编译基础，如TensorFlow、TPU编译器）。

- 英特尔推出的nGraph是一款开源的深度学习系统编译器，它旨在为所有深度学习框架提供统一的编译和优化，支持多种硬件平台，包括CPU、GPU和FPGA，并提供了丰富的优化选项，以实现高效的深度学习推理和训练。nGraph可以接收任何深度学习框架的模型作为输入，支持多种硬件平台，提供了丰富的优化选项，包括张量融合、计算图优化、向量化等。

AI编译器与传统编译器的差异

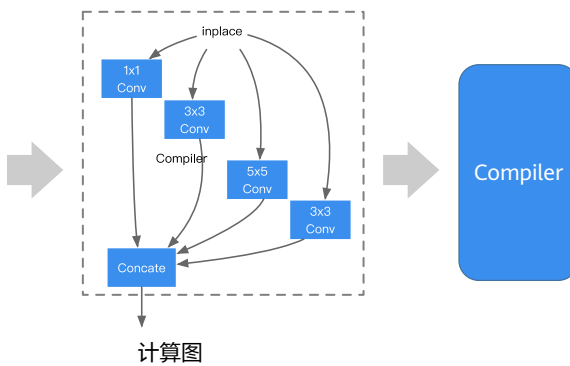


PyTorch 获取计算图的方式

```
import torch
```

```
x1 = torch.conv2d((input, 3))  
x2 = torch.conv2d((3, 8))  
x3 = torch.conv2d((3, 16))  
x4 = torch.conv2d((3, 24))  
out = torch.concat(x1, x2, x3,  
x4)
```

Python code



PyTorch编译方式演进

特性	内容	适用场景	技术方式
TorchScript	TorchScript的主要用途是进行模型部署，需要记录生成一个便于推理优化的IR，对计算图的编辑通常都是面向性能提升等等，不会给模型本身添加新的功能。	推理和部署静态图优化	AST + Trace
Torch FX	FX的主要用途是进行python->python的翻译，它的IR中节点类型更简单，比如函数调用、属性提取等等，这样的IR学习成本更低更容易编辑。使用FX 来编辑图通常是为了实现某种特定功能，比如给模型插入量化节点等，避免手动编辑网络造成的重复劳动。	修改正向图	Trace
LazyTensor	采用tensor不被用户观察那么就可以缓存计算指令的方式，隐式构图+优化，用户不用标记构图的代码范围，对用户的代码限制小，是一种能优化就优化、不能优化也能执行的思路，偏向于易用性	针对NPU子图编译优化	Trace
TorchDynamo	TorchDynamo支持使用Python在运行时修改动态执行逻辑，修改的时机是CPython的ByteCode执行前。思想类似DynamoRIO项目，DynamoRIO可以动态的修改x86机器码。	动静统一 + 编译优化	Python 解析修改

目录

1. AI计算系统介绍
2. AI芯片基础知识
3. AI计算体系与矩阵运算
4. 编译器
- 5. 昇腾AI计算平台**

昇腾AI基础软硬件平台，构筑智能世界的基石



昇腾硬件



Atlas 900 PoD A2
Atlas 900 AI集群



昇腾训练芯片



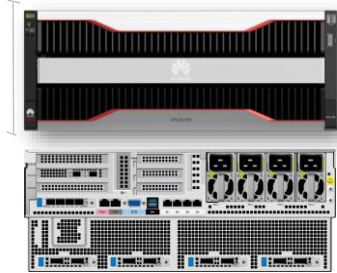
Atlas 800T A2训练服务器



Atlas 800I A2推理服务器

Atlas 800T A2训练服务器

- Atlas 800T A2训练服务器是基于华为自研鲲鹏处理器和昇腾AI处理器的AI训练服务器。
 - 高计算密度
 - 高能效比
 - 高网络带宽
 - 易扩展
 - 易管理等优点
- 广泛应用于深度学习模型开发和AI训练服务场景。该设备面向公有云、互联网、运营商、政府、交通、金融、高校、电力等领域，满足企业机房部署和大规模数据中心集群部署。



Atlas 900 A2 PoD

- Atlas 900 PoD A2 集群基础单元是基于**华为自研昇腾AI处理器**的AI集群基础单元，支持大规模，高性能AI训练业务，可应用于人工智能计算中心（简称为AICC）、运营商、云计算等重算力场景，**基于天成平台，架构/性能全面升级。**



80 Copyright © Huawei Technologies Co., Ltd. All rights reserved.



- 极致算力密度**
 - 1柜顶2柜
 - 计算、存储、网络、能源一体化机柜；
- 极高AI能效**
 - 2倍业界能效比
 - 全液冷散热，免冷机、免空调PUE≤1.15；
- 极简交付部署**
 - 3总线盲插
 - 业界首创供电、互联、液冷三总线架构，计算节点全盲插；
 - 机柜工厂预集成，整柜运输管理，交换机一键配置，现场交付时间**降低38%**。

昇腾编译器

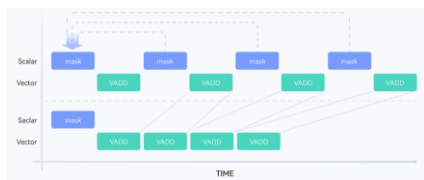


异构编译

Host+Device异构编译架构，支持C/C++语法，使能昇腾异构芯片系统

软硬协同微架构精准优化

针对昇腾微架构特性，突破特殊寄存器优化、智能选择高性能指令等优化特性，释放芯片极致算力



微指令亲和调度

构建昇腾亲和调度模型，识别更优指令排布，充分利用芯片优势特性

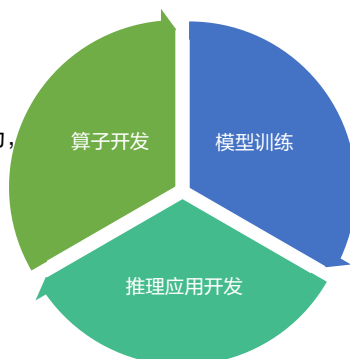


CANN是什么？



关键功能特性

CANN提供了超过1400个硬件亲和的高性能算子，可覆盖主流AI框架的算子加速需求；CANN开放了自定义算子开发的能力，开发者可根据自身需求选择不同的算子开发方式。



CANN针对训练任务提供了完备的支持，针对PyTorch、TensorFlow等开源框架网络模型，CANN提供了模型迁移工具，支持将其快速迁移到昇腾平台。

AscendCL提供统一API框架，用于昇腾平台上的神经网络应用开发，涵盖资源管理、内存管理、模型及算子执行、媒体数据处理等功能，实现深度学习推理、图像预处理和单算子加速等能力。

昇腾CANN：向下使能处理器并行加速，向上使能高效开发

① 全面开放，支持多种框架

一个月内适配新版本

② 支持GPU生态快速迁移

PyTorch模型端到端2周内完成迁移

③ 算子高效原生开发

典型场景算子开发周期 <2人周

AI框架

全面支持业界AI框架，演进版本快速兼容，适配周期≤1个月

PyTorch

[M] 昇思

TensorFlow

PP飞浆

Caffe

.....

CANN

支持GPU生态（模型/加速库）的API级迁移

第三方库

开源模型

基础加速库

DNN | BLAS | FFT | Sparse

二进制基础算子库

Ascend C 昇腾算子编程语言支持原生开发

//数据搬入

遵循C/C++标准规范

//数据计算

结构化核函数编程

//数据搬出

自动化流水并行调度

图编译加速技术使能处理器并行加速

自动流水

算子深度融合

整图下沉

自适应梯度切分

昇腾芯片



昇腾推理芯片 | 昇腾训练芯片 |



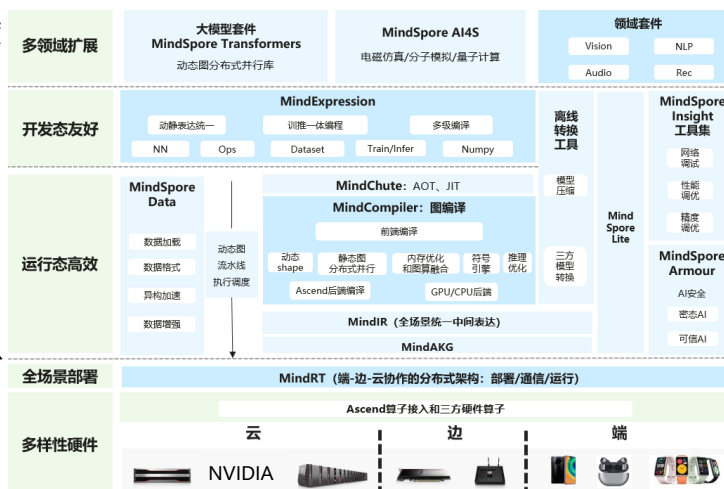
打造模型/算子开发使能平台，提升训练模型原生开发效率



MindSpore

- MindSpore，全场景深度学习框架。

- 易开发：API友好，调试难度低；
- 高效执行：计算效率、数据预处理效率和分布式训练效率；
- 全场景：框架同时支持云、边缘以及端侧场景。



- MindSpore作为新一代深度学习框架，是源于全产业的最佳实践，最佳匹配昇腾处理器算力，支持终端、边缘、云全场景灵活部署，开创全新的AI编程范式，降低AI开发门槛。

应用使能组件

- 昇腾应用使能组件集，包括昇腾亲和的极致性能应用对接SDK、极简易用的设备纳管/调度/监控/故障诊断/断点续训等平台对接组件、昇腾推理引擎、昇腾大模型训练加速库。



思考题

1. （多选题）以下哪些优化操作由AI编译器完成？
- A. 计算图优化
 - B. 程序机器码生成
 - C. 模型结构化
 - D. 分布式策略优化

- 答案：AB

本章总结

- 本课程主要围绕AI计算系统展开，介绍了AI芯片的架构设计、关键计算指标，数据类型、AI编译器的优化策略与实现原理。

更多信息



Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



模型开发全流程



前言

- AI技术已成为当今社会的最重要技术及热点话题之一，它可以帮我们处理很多问题。而其核心部分是训练好的AI模型。那么，AI模型是如何形成的呢，它需要经过哪些流程，本章我们逐一来解析。

目标

- 学完本课程后，您将能够：
 - 熟悉AI模型开发的整体流程
 - 掌握AI模型开发过程每个步骤需要完成的工作
 - 根据具体业务设计AI模型开发流程

目录

1. 概述
2. 业务目的分析
3. 数据准备
4. 模型选型
5. 模型迁移
6. 模型训练
7. 模型评估
8. 模型部署

AI已融入到我们的生活和工作

- 在当今快速发展的科技时代，AI已成为我们生活中不可或缺的一部分，无论是在日常生活中的医、食、住、行方面，还是在智慧城市的管理和智能制造等领域，都发挥着重要作用。



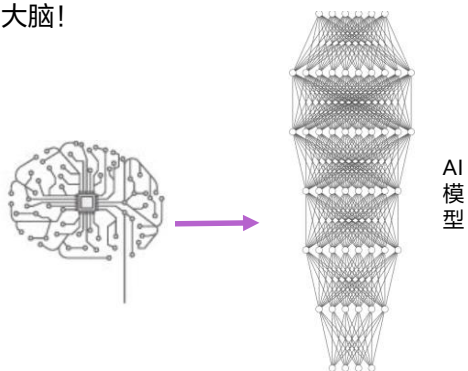
5 Copyright © Huawei Technologies Co., Ltd. All rights reserved.



- https://www-file.huawei.com/-/media/corp2020/pdf/giv/2024/intelligent_world_2030_2024_cn.pdf?la=zh
- 在当今快速发展的科技时代，人工智能（AI）已成为我们生活不可或缺的一部分。从智能闹钟的温和唤醒到智能家居营造的最佳睡眠环境，从工作中的智能助手高效协作到出行时精准导航避开拥堵，AI的应用无处不在。它通过智慧管理优化城市资源配置，提升生活质量，并利用大数据分析解决潜在问题。AI不仅丰富了我们的日常生活，提升了工作效率，还使城市管理更加智能化和人性化，完美结合了科技与人文关怀。

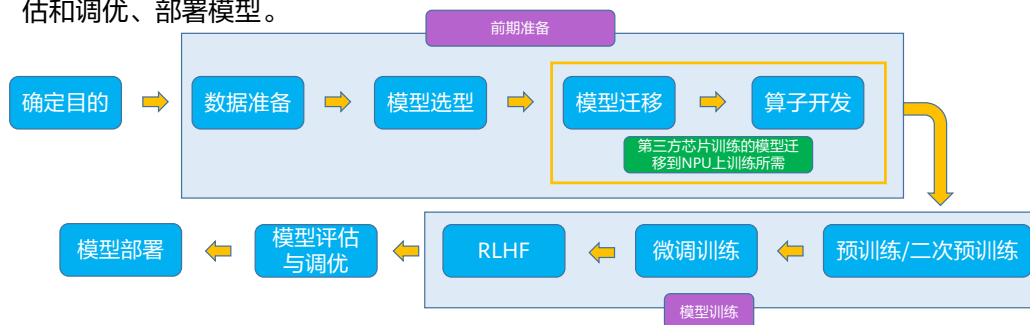
智能的核心 - AI模型

- 让各个行业具备智慧能力的核心在于AI模型。这些模型通过处理和分析大量数据，能够提供深刻的见解、自动执行复杂的任务，并支持决策过程，从而提升各行业的效率。
- AI模型就是这个智慧的大脑！



如何开发一个AI模型？

- AI开发的基本流程通常可以归纳为几个步骤：确定目的、前期准备、训练模型、模型评估和调优、部署模型。



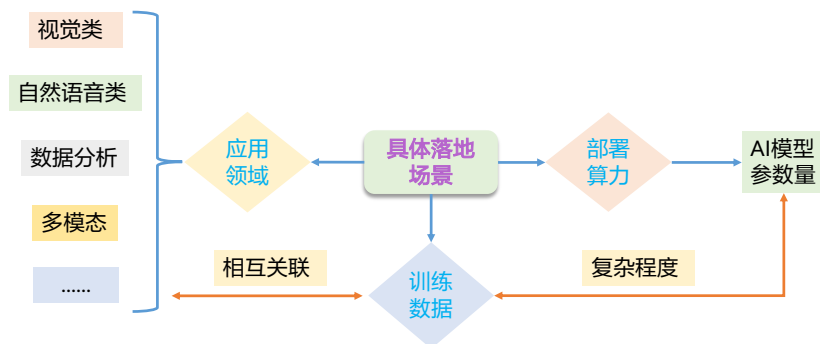
- 以上的步骤是AI模型开发的通用流程，部分步骤可能存在更多的细节。

目录

1. 概述
- 2. 业务目的的分析**
3. 数据准备
4. 模型选型
5. 模型迁移
6. 模型训练
7. 模型评估
8. 模型部署

确定目的

- 如本章前面讲到的，AI已经融入到千行百业，不同行业的业务需求不同。因此在开始AI开发之前，必须明确要解决什么问题？具体的问题，决定了AI落地的场景，同时也能确定训练数据的类型等，结合分析结果整理AI开发的思路。

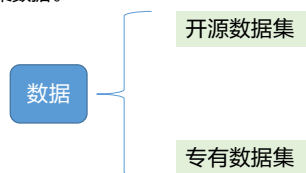


目录

1. 概述
2. 业务目的分析
- 3. 数据准备**
4. 模型选型
5. 模型迁移
6. 模型训练
7. 模型评估
8. 模型部署

数据收集

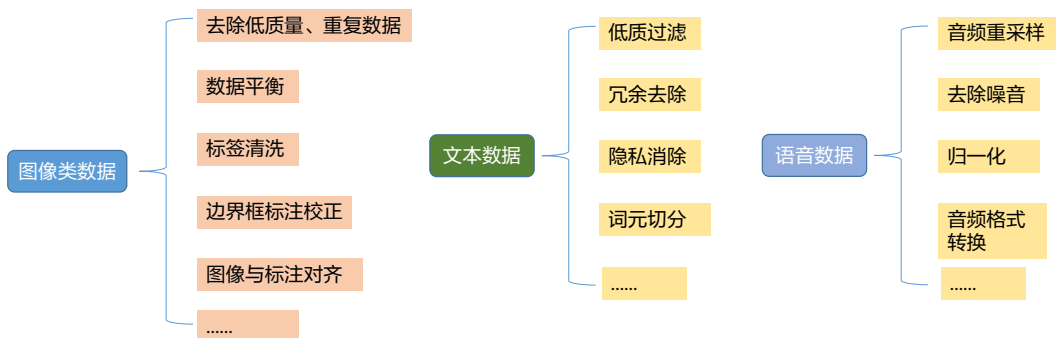
- 确定好场景和目的后，即可收集数据。



- 如果AI模型落地是在通用场景中，例如日常的一些物品检测，常见的文本情感分析等，由于目标场景在一些公开数据集中大多可以找到，因此训练模型数据可以使用开源数据集；然后根据训练效果判断是否需要制作专有数据集。
- 如果落地场景与通用场景相近，但具有一些专业性，可以使用开源数据集+专有数据集。
- 如果落地场景是非常专业的场景，例如需要实现矿山上的一些特定物体检测、小语种翻译等，可以直接使用专业数据集训练，训练起始可以使用相关的预训练模型。

数据预处理（1）

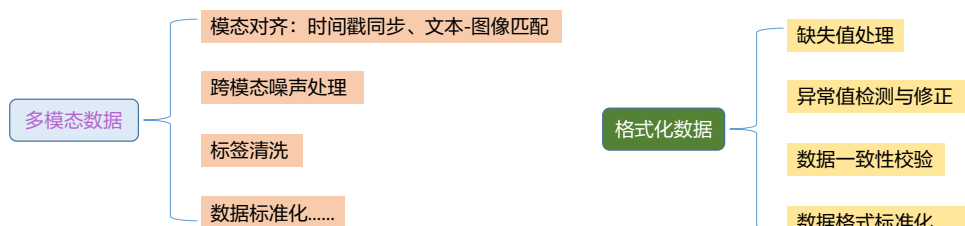
- 一般情况下，收集到数据不能直接用来训练模型。因为数据中可能存在很多错误，或者开源数据集中有部分数据并不是训练需要的，或者数据格式、尺寸不满足需求，所以要对数据进行处理。



- <https://developer.huawei.com/consumer/cn/forum/topic/0204138379175336953>

数据预处理（2）

- 对于多模态数据集，需要关注不同模态的数据关联。



- 模态对齐
 - 时间戳同步：对视频和音频数据，确保帧与音频片段的时间戳一致。
 - 文本-图像匹配：验证描述性文本与图像内容的一致性（如“猫”与图片中的猫）。
- 跨模态噪声处理
 - 图像与文本关联校验：移除无关文本（如广告文字）或模糊图像。
 - 多模态去重：通过哈希值或嵌入向量检测重复的图文对。
- 格式标准化
 - 图像格式统一：将JPEG、PNG等格式转换为统一格式。
 - 音频采样率调整：将音频统一为16kHz或44.1kHz。
 - 文本编码：将PDF/扫描件转换为UTF-8编码纯文本。
- 多模态增强
 - 图文混合增强：对图像进行裁剪时同步调整文本描述（如“左边的狗”需对应裁剪后的图像）
 - 跨模态插值：对缺失的模态数据（如缺少音频的视频）进行合理补全。

目录

1. 概述
2. 业务目的分析
3. 数据准备
- 4. 模型选型**
5. 模型迁移
6. 模型训练
7. 模型评估
8. 模型部署

模型选型 – 参数量

- 根据不同任务，选择不同功能的模型时，模型参数量可以大概估计。但是根据不同的场景下数据复杂程度、数据量、部署算力大小，也需要选择同类别适合的模型，或者修改模型结构。
- 其中各个因素相互影响。



模型选型 – 模型类型

- 应用场景和训练数据决定AI模型的类型：
 - 视觉场景：
 - 图像分类：MLP、CNN.....
 - 物体检测：Faster RCNN、YOLO.....
 - 图像生成：Stable Diffusion.....
 - 自然语言类：
 - 文本生成类任务：基本以Transformer架构为基础的模型
 - 数据分析：XGBoost.....
 - 多模态：CLIP.....

目录

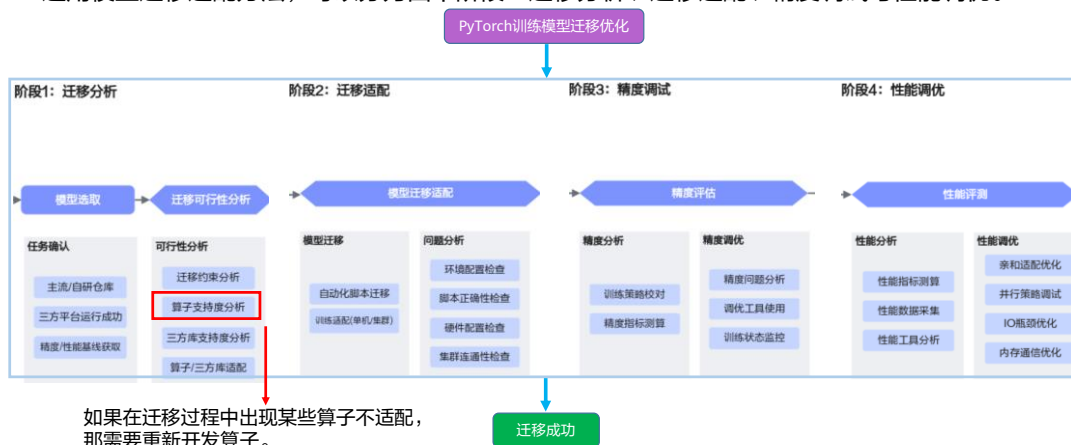
1. 概述
2. 业务目的分析
3. 数据准备
4. 模型选型
- 5. 模型迁移**
6. 模型训练
7. 模型评估
8. 模型部署

模型迁移

- 什么是模型迁移？
 - 将原本设计用于GPU或其他三方平台的深度学习模型训练代码，经过模型代码修改等适配操作，来适应NPU的架构和编程，让模型能在NPU上进行高性能运行。
- 为什么要做模型迁移？
 - 在将模型从其他三方平台迁移到NPU时，由于硬件架构和库的不同，涉及到一系列底层到上层的适配操作。
 - 从第三方AI芯片训练完成的模型，迁移至NPU需要适配的原因可分为三方面：
 - 硬件特性和性能特点差异：由于不同的硬件特性和性能特点不同，模型在NPU上可能需要进一步的性能调试和优化，以充分发挥NPU的潜力。
 - 计算架构差异：不同厂家根据硬件开发计算加速库不同，因此存在差异，需要适配。例如CANN（Compute Architecture for Neural Networks）是华为NPU的异构计算架构。
 - 深度学习框架差异：为了支持NPU硬件，需要对PyTorch框架进行适配：包括适配张量运算、自动微分等功能，以便在NPU上高效执行。PyTorch正在原生支持NPU，以提供给用户更好的模型体验，实现迁移修改最小化。

迁移总体思路

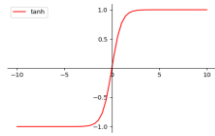
- 通用模型迁移适配方法，可以分为四个阶段：迁移分析、迁移适配、精度调试与性能调优。



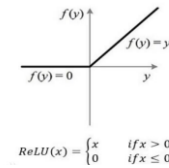
- https://www.hiascend.com/doc_center/source/zh/Pytorch/60RC1/ptmoddevg/trainingmigrguide/PT_LMTMOG_0005.html

什么是算子

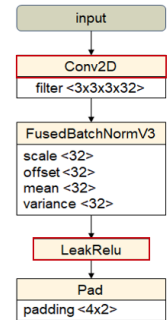
- 在迁移总体思路第一步的迁移可行性分析中，有2项是算子支持度分析和算子适配。那什么是算子呢？
- 深度学习算法由一个个计算单元组成，我们称这些计算单元为算子（Operator，简称OP）。在网络模型中，算子对应层中的计算逻辑，例如：卷积层（Convolution Layer）是一个算子；全连接层（Fully-connected Layer，FC layer）中的权重求和过程，也是一个算子。
- 在数学领域，一个函数空间到函数空间上的映射 $O: X \rightarrow Y$ ，都称为算子。
- 广义的讲，对任何函数进行某一项操作都可以认为是一个算子，比如微分算子，不定积分算子等。



Tanh函数图像



ReLU函数图像



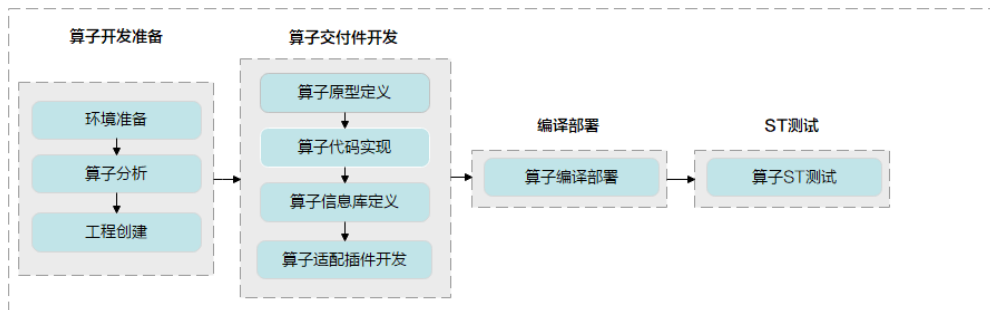
网络示例

为什么需要算子开发

- 一般场景下，开发者无需自己开发算子。但若遇到以下场景，开发者需要考虑进行自定义算子的开发：
 - 训练场景下，将第三方框架（例如TensorFlow、PyTorch等）的网络训练脚本迁移到昇腾AI处理器时遇到了不支持的算子。
 - 推理场景下，将第三方框架模型（例如TensorFlow、Caffe、ONNX等）使用ATC工具转换为适配昇腾AI处理器的离线模型时遇到了不支持的算子。
 - 网络调优时，发现某算子性能较低，想重新开发一个高性能算子替换性能较低的算子。
 - 推理场景下，若应用程序中的某些逻辑涉及到数学运算（例如查找最大值，进行数据类型转换等），开发者可以将这些操作通过自定义算子的方式进行实现，然后在应用程序中对算子进行调用，从而利用昇腾AI处理器进行加速。
- 例如，针对一个分类应用，我们想从分类模型的推理结果中查找可能性最大的前5个标识，则可以实现一个查找最大值的算子（例如ArgMax），后续就可以直接通过AscendCL接口调用此算子实现对推理结果的后处理。

算子开发的基本流程

- 全新开发算子的流程如下：



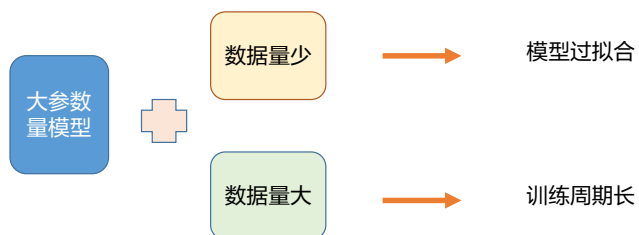
- 每一个算子交付件的含义如下：
- 算子原型定义：算子调用入口，定义算子输入、输出、属性。
- 算子实现代码：描述算子的运算逻辑，完成特定的计算功能。
- 算子信息库：描述算子在昇腾AI处理器上的实现限制，包含算子输入输出的 dtype、format、shape信息。算子编译时，会根据算子信息库匹配到算子实现。
- 算子适配插件开发用于进行第三方框架的适配，让第三方框架的算子可以映射到 CANN算子。

目录

1. 概述
2. 业务目的分析
3. 数据准备
4. 模型选型
5. 模型迁移
- 6. 模型训练**
7. 模型评估
8. 模型部署

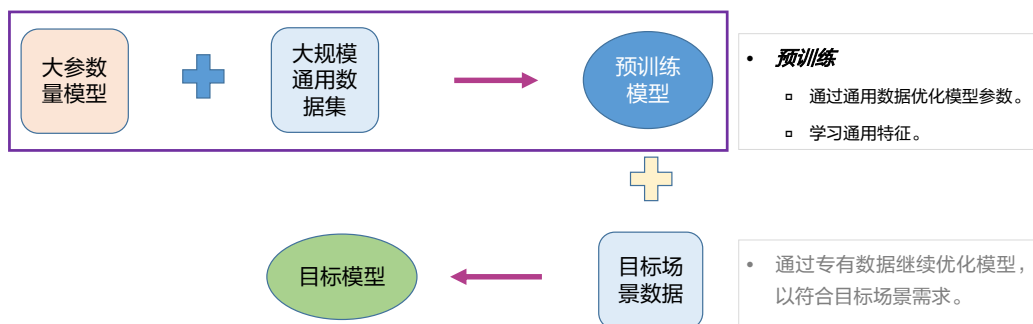
从零训练大参数量模型存在的问题

- 完成所有准备工作后，即可在不同的硬件设备上训练模型。训练过程是不断让模型学习数据中的特征，提升在特定领域任务中的能力。
- 不过，制作大规模数据集是一项非常昂贵的事情。同时，在部分领域，数据采集难度大，导致可用的数据数量小。如果选择了大参数量的模型，这会导致模型出现过拟合。并且，在训练大参数量模型时，训练时间长，难度大。



模型预训练

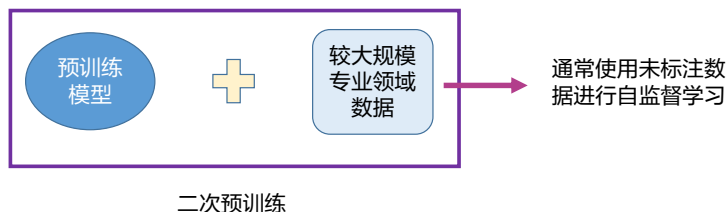
- 预训练是解决上一个问题的有效方式。预训练的策略是在通用的大规模数据集上训练模型，保存训练完成的模型，提供给后续开发者使用。



- 对于语言模型，预训练阶段一般会在超大规模的语料上，采用无监督（unsupervised）或者弱监督（weak-supervised）的方式训练模型，期望模型能够获得语言相关的知识，比如句法，语法知识等等。经过超大规模语料的”洗礼”，预训练模型往往会是一个Super模型，一方面体现在它具备足够多的语言知识，一方面是因为它的参数规模很大。

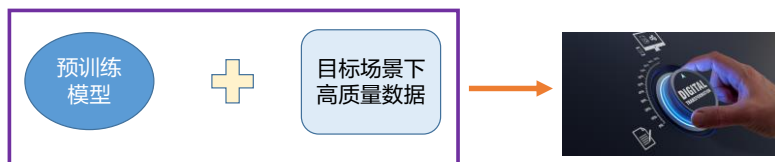
二次预训练（可选）

- 二次预训练也称为增量预训练，它是大语言模型爆发后出现的一种技术。通常大模型在初次预训练后的阶段被称为Post-Pretraining，而二次预训练也是其中的一个阶段。
- 二次预训练通常是指在已有大规模通用预训练模型的基础上，利用特定领域或任务的数据进行再次预训练的过程。其核心目标是让模型在保留通用语言能力的同时，更好地适应特定领域（如医学、法律、金融）或任务的需求。



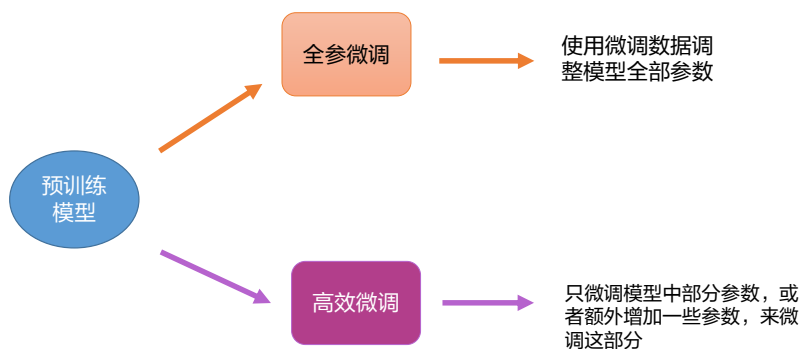
微调

- 在实际AI应用中，实际场景可能是某专业领域内的一个子领域。而前面训练完成的模型只是具备在某一大的专业领域内，甚至只是在通用场景中解决问题的能力，但是在解决实际具体场景的能力还欠缺，因此，需要在该场景下对模型微微调整一下，以便让模型能在当下场景下表现更佳。
- 微调使用预训练模型的权重作为起点，在较小的高质量具体场景数据集上进行进一步训练，该阶段使用的数据集更直接地反映了模型将用于的特定任务和用例。它通常需要监督学习，但也可能涉及强化学习、自监督学习或半监督学习。



微调的方式

- 微调，到底是调整哪些部分，如何调？



- <https://www.hiascend.com/forum/thread-02111169994255439057-1-1.html>

基于人类反馈的强化学习 (RLHF)

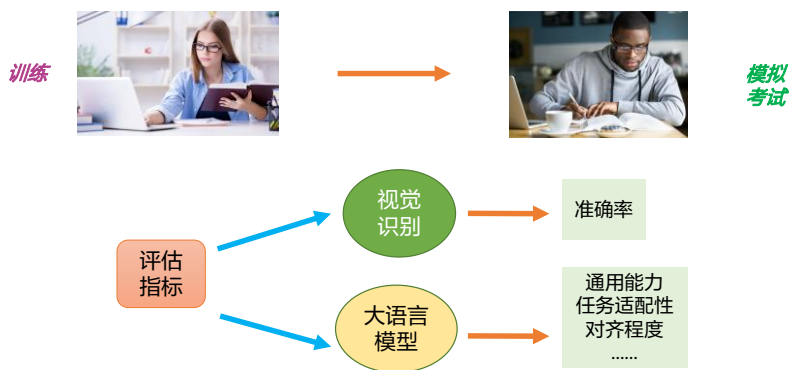
- 虽然指令调整可以教给模型一些具体、直接的行为，比如如何组织响应，但通过标记示例来教给模型一些抽象的人类品质，比如乐于助人、事实准确性、幽默感或同理心，可能非常费力和困难。
- 为了更好地使模型输出与理想的人类行为保持一致，尤其是对于聊天机器人等对话用例，SFT可以辅之以强化学习，更具体地说，基于人类反馈的强化学习（RLHF）。
- RLHF也称为基于人类偏好的强化学习，有助于通过离散示例对复杂、定义不清或难以指定的性质进行模型微调。
- 以喜剧为例：要用SFT教一个模型变得“有趣”，不仅需要花费成本和劳动力来编写（或获取）足够多的笑话来构成可学习模式，而且还要求给定数据科学家认为有趣的内容与用户群认为有趣的内容相一致。RLHF本质上提供一种替代方案：提示LLM生成笑话并让人类测试人员评估其质量。这些评分可用于训练奖励模型，以预测将获得积极反馈的笑话类型，反过来，该奖励模型可用于通过强化学习来训练LLM。

目录

1. 概述
2. 业务目的分析
3. 数据准备
4. 模型选型
5. 模型迁移
6. 模型训练
- 7. 模型评估**
8. 模型部署

训练后的模型评估

- 模型训练完成后，需要进行评估，查看训练效果。
- 模型的训练可以比作学习，那么评估就相当于模拟考试。



- 图像检索：mAP-衡量检索结果的相关性排序质量。

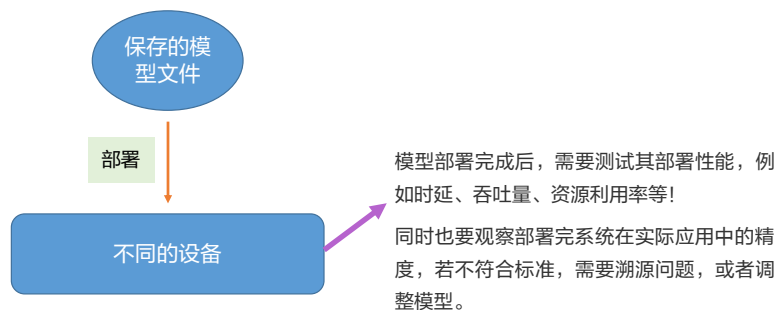
目录

1. 概述
2. 业务目的分析
3. 数据准备
4. 模型选型
5. 模型迁移
6. 模型训练
7. 模型评估
- 8. 模型部署**

模型部署

- 模型部署是指将训练好的模型投入生产环境中使用的过程。其工作原理如下：

- 首先，将训练好的模型文件转换为可部署的格式；
- 然后，将打包后的模型部署到生产环境，如云服务器或边缘设备。



- 部署后，应用程序可以通过API调用该模型，将新的输入数据传递给模型，模型会对输入数据进行预测或处理，并将结果返回给应用程序。为确保模型在生产环境中的稳定性和可靠性，通常需要进行模型监控、版本控制、自动扩缩容等操作。模型部署使机器学习模型可以为实际应用提供服务，是将模型从实验室推向产品化的关键一步。

AI模型开发流程总结

- 在实际的AI系统落地中，将AI模型部署好之后，用户通过Web界面或者APP使用。
- AI模型开发的整体流程是收集落地场景信息，分析需求，明确项目目的，完成前期的准备。例如，根据场景收集数据，根据不同开发芯片选择迁移适配等。随着模型结构和参数量发生了变化，训练流程也发生了改变，此时也需要根据模型的特性和训练效果，选择合适的训练步骤。训练完成的模型需要对其进行测试，观察其精度和性能。
- 虽然模型训练完成，但是AI开发并未全部结束，还需要将模型部署在落地设备上，并测试在实际应用中的效果，例如模型的输出表现、使用时延等。

思考题

1. （判断题）模型部署是在模型在训练完成后，直接将训练代码保存在实际生产环境中进行预测流程。

- 1. False
- 因为训练完成的模型需要

本章总结

- 本章主要介绍了如何完整的构建AI模型开发流程。从需求分析出发，首先明确开发目的，然后将其转化为AI需要解决的问题。然后需要构建数据集，如果选择第三方芯片训练的模型，需要在昇腾硬件上继续训练，则要完成模型的迁移。完成前期准备工作后，进行模型的训练。根据不同的模型和其训练后的精度，选择所需要的训练方式，并对训练完成的模型进行测试。最后，将测试完成的模型部署在落地设备上。

学习推荐

- 昇腾社区
 - <https://www.hiascend.com/>
- MindSpore官网
 - <https://www.mindspore.cn/>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



模型选型与设计



前言

- 在AI模型开发的过程中，如何选择一个合适的模型来解决问题。有时模型不能满足需求，如何修改甚至重新设计一个模型，是一项非常重要的事情。它需要考虑应用场景、训练和部署设备、模型特点等问题。接下来，我们一起来了解一下主流场景中的模型选型和设计思路。

目标

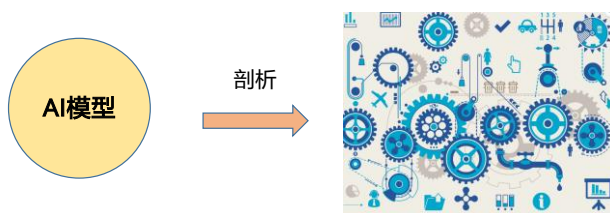
- 学完本课程后，您将能够：
 - 掌握主流任务场景下适配的模型
 - 掌握不同场景中主流模型的结构及其特点
 - 根据自身任务选择模型
 - 了解修改或设计一个模型的主要因素

目录

1. AI模型选型及设计分析
2. 模型选型与设计总结

引言

- 随着AI算法的不断发展，逐渐形成了不同的神经网络模型，这些模型在不同领域表现优异。在日常的模型开发工作中，可以选择不同领域表现优异的模型，或者对其进行修改，来适配不同场景的任务。
- 在选择AI模型之前，需要先了解它的原理或内部结构，以便更好的判断选择什么样的模型。



如何根据应用场景去选择模型？

- 在AI模型的开发与应用中，都会面临一个问题，那就是使用什么样的模型？在众多模型中，我们又如何选择呢？如果选择的模型不能完美适合当前场景，又该怎么设计新的模型？



目录

1. AI模型选型及设计分析

- 基于卷积神经网络的视觉模型
 - 图像分类任务
 - 目标检测任务
- 基于Transformer的大语言模型
- 基于Transformer的视觉模型
- 基于Transformer的多模态大模型

2. 模型选型与设计总结

图像分类任务

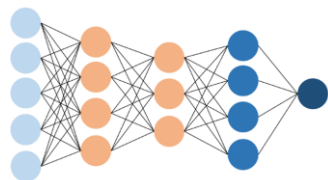
- 人类希望将AI技术应用在视觉领域，让这项技术代替人类的眼睛和智慧，提升效率。
- 在AI技术发展过程中，它在视觉领域的应用就非常广泛。例如工业检测、智能驾驶、智慧安防等。
- 接下来让我们看一下，在视觉领域里较为简单的，也是较为常用的图像分类任务中，解决该类问题的算法是如何发展的，又如何去选择模型？



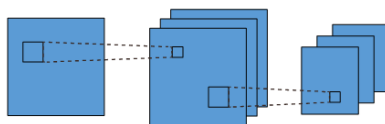
这两张图片中分别是什么动物？

从多层感知机到卷积神经网络

- 最简单的深度网络称为多层感知机。多层感知机可以实现多种任务，例如一些数据的预测，众多的文本、图像等分类工作。
- 卷积神经网络通过卷积提取特征的方式，拓展了神经网络在图像领域的应用，它不仅可以实现更多的功能，同时相比同等深度的全连接神经网络，参数量有大幅下降。



由于由全连接神经网络构成的多层感知机参数量较大，若构建深层次的神经网络，需要消耗更多的计算资源。

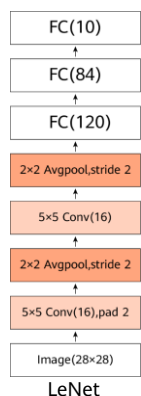


卷积核的权值共享大幅降低了神经网络的参数量，同时为探索更多图像识别功能提供了新的工具。

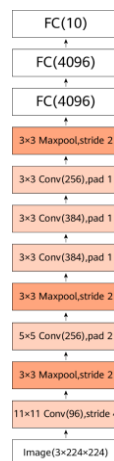
- 全连接神经网络到CNN的重大转变，改变了模型结构。

从LeNet到AlexNet

LeNet，它是最早发布的卷积神经网络之一，因其在计算机视觉任务中的高效性能而受到广泛关注。这个模型是由贝尔实验室的研究员Yann LeCun在1989年提出的（并以其命名），目的是识别图像中的手写数字。



- 网络深度增加，
- 使用了不同尺寸的卷积核和池化
- 激活函数从sigmoid转换为ReLU



2012年，AlexNet横空出世。它首次证明了学习到的特征可以超越手工设计的特征。它一举打破了计算机视觉研究的现状。AlexNet使用了8层卷积神经网络，并以很大的优势赢得了2012年ImageNet图像识别挑战赛。

AlexNet

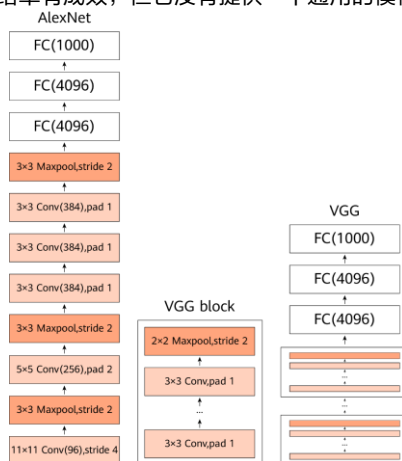
随着网络结构发展，增加模型的宽度（增加通道数）和深度（增加层数），是提升模型性能的重要方式之一。

- 1989年，Yann LeCun发表了第一篇通过反向传播成功训练卷积神经网络的研究，这项工作代表了十多年来神经网络研究开发的成果。

VGG

- 虽然AlexNet证明深层神经网络卓有成效，但它没有提供一个通用的模板来指导后续的研究人员设计新的网络。

研究人员开始从单个神经元的角度思考问题，发展到整个层，现在又转向块，重复层的模式。



VGG块形成了一个可通用网络结构，通过堆叠块的方式，快速增加模型的深度，提升模型识别能力。

- 网络中使用了带填充的卷积层
- 卷积核尺寸为3 X 3
- 使用了最大池化层

- 与芯片设计中工程师从放置晶体管到逻辑元件再到逻辑块的过程类似，神经网络架构的设计也逐渐变得更加抽象。

NiN (network in network) 网络

- LeNet、AlexNet和VGG都有一个共同的设计模式：通过一系列的卷积层与池化来提取空间结构特征；然后通过全连接层对特征的表征进行处理。
- 全连接层参数： $C_i * F_h * F_w * C_o$
- 卷积层参数： $C_i * K_h * K_w * C_o$
 - C_i 表述输入特征的通道数， C_o 表述输出通道数， F_h 、 F_w 分别表示特征图的高和宽， K_h 、 K_w 分别表示卷积核 (kernel) 的高和宽。
- 通常，特征图的尺寸远大于卷积核尺寸，因此全连接层的参数量相对卷积层参数大。常见全连接层参数量：

LeNet: $16 * 5 * 5 * 120 = 48K$

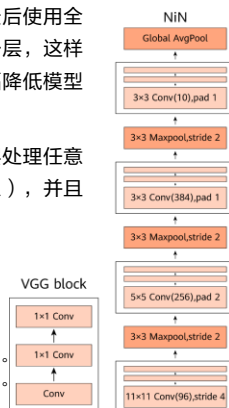
AlexNet: $256 * 5 * 5 * 4096 = 26M$

VGG: $512 * 7 * 7 * 4096 = 102M$

参数量大，训练时需要更多的计算资源，同时较容易引起过拟合。

- NiN网络的思想是使用 1×1 的卷积层增加网络深度，提高模型的非线性表达能力（使用激活函数），并且在最后使用全局池化层代替全连接的最后一层，这样即增加了网络深度，同时大幅降低模型参数量。
- 而它的引入可以使网络在该层处理任意尺寸的特征图（即该层的输入），并且还保留了空间信息。

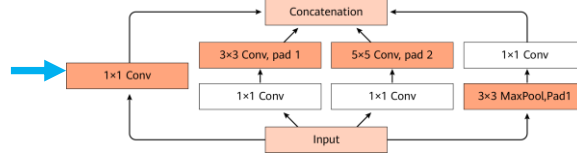
NiN块以一个普通卷积层开始，后面是两个 1×1 的卷积层。这两个 1×1 卷积层充当带有ReLU激活函数的逐像素全连接层。第一层的卷积窗口形状通常由用户设置。随后的卷积窗口形状固定为 1×1 。



GoogLeNet

- 在2014年的ImageNet图像识别挑战赛中，GoogLeNet网络架构大放异彩。

它在网络架构中使用了名为Inception块的架构



这个模型一个重点是使用不同大小的卷积核，解决了如何选择不同大小的卷积核是最合适的问题。

Inception块由四条并行路径组成。前三条路径使用窗口大小为 1×1 、 3×3 和 5×5 的卷积层，从不同空间大小中提取信息。中间的两条路径在输入上执行卷积，以减少通道数，从而降低模型的复杂性。第四条路径使用最大池化层，然后使用卷积层来改变通道数。

GoogLeNet首次引入Inception结构，在网络中堆叠该结构使得网络层数达到了22层，这也是卷积网络首次超过20层的标志。

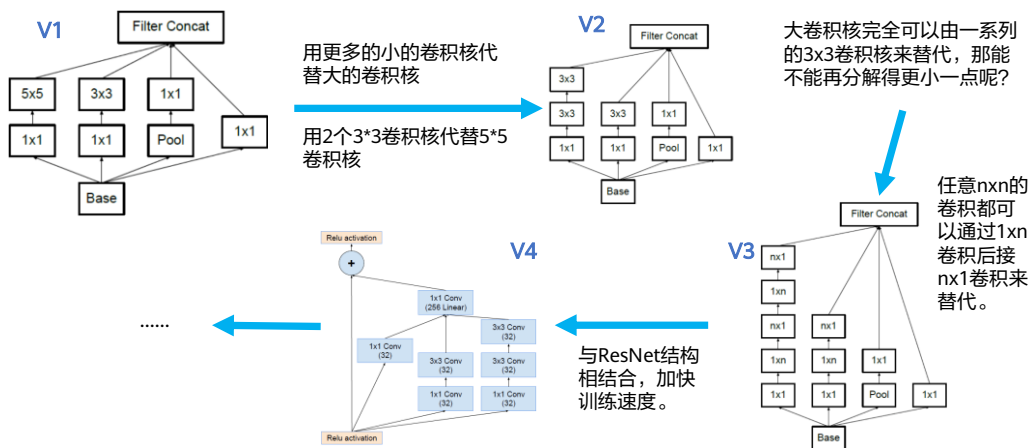
GoogLeNet在参数量更小的情况下，性能略优于VGG，是图像分类领域轻量化模型的优选。

类似思想的算法还有 **MobileNet** 和 **ShuffleNet**，虽然网络结构上有所差异，但它们都是增加了网络的复杂度，但参数量相对较小，适合端边设备部署。

- GoogLeNet起名时致敬了LeNet。
- GoogLeNet和VGG是当年ImageNet挑战赛(ILSVRC14)的双雄，GoogLeNet获得了第一名、VGG获得了第二名，这两类模型结构的共同特点是层次更深了。

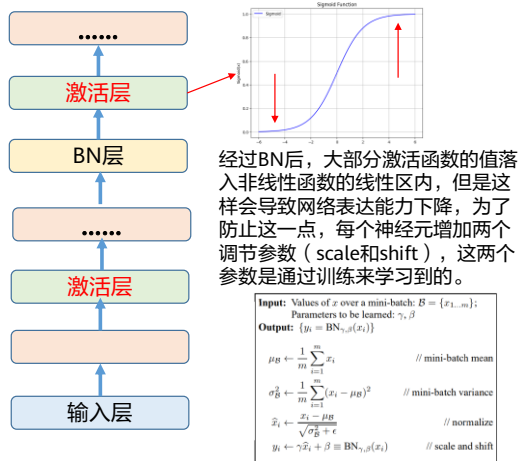
Inception架构的发展

- Inception架构是GoogLeNet最重要的部分，它 also 经历几次发展，下面讨论一下它们的发展思想。



- V3: GoogLeNet团队发现在网络的前期使用这种分解效果并不好，在中度大小的特征图（feature map）上使用效果才会更好（特征图大小建议在12到20之间）。

批量规范化 (batch normalization)



- 随着神经网络的层数增加，模型越来越深。数据在底层传到上层时，很容易落入激活函数的梯度饱和区。另外，同一分布的数据经过多层的变化，每一层的参数更新都会导致上层的输入数据在输出时分布规律发生了变化，并且这个差异会随着网络深度增大而增大，这会导致：

- 整个模型中的数据分布不同，模型训练较难
- 部分层输出落在激活函数的梯度饱和区，导致梯度接近于0，反向传播时底层网络易出现梯度消失

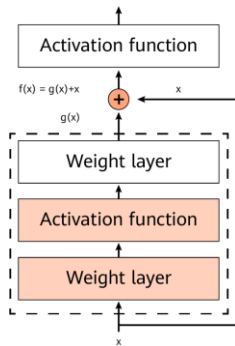
- 批量规范化 (batch normalization) 的提出就是为了缓解这种状况，它将每层数据的均值修正为0，方差为1。

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

- <https://arxiv.org/pdf/1502.03167>

ResNet

- 前面提到，为了提高神经网络的能力，需要不断的增加模型的深度，可以提取图像中更深层次的特征来判断图像中是什么。但是随着网络层数增加，底层梯度消失的风险非常大，这就影响了模型的训练。而ResNet的提出，极大的缓解了梯度消失的风险。

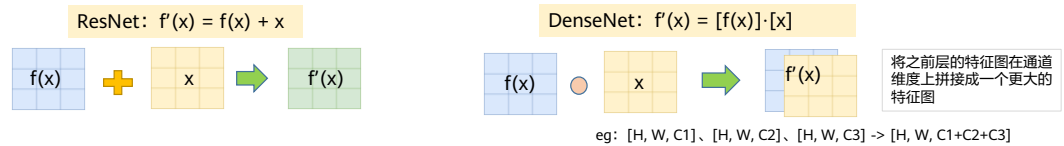


- ResNet是在串行的网络结构中，将之前的结果并行的加入了后面的层中，这样使得计算中增加了加法，在梯度更新时，不会因为全部的乘法操作导致梯度消失。而插入的这部分被称为残差，左图的整个块被称为残差块。
- 14年的VGG网络才只有19层，但是ResNet的网络层数可以达到了惊人的152层。
- 这种残差连接的思想，在后面的网络发展中起到了举足轻重的作用。

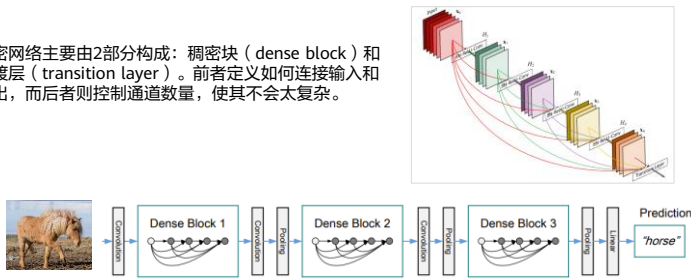
- 作者何凯明，它的论文ResNet被引18万次之多。2023未来科学大奖-数学与计算机科学奖颁给了ResNet的团队，表彰他们提出深度残差学习，为人工智能做出了基础性贡献。

DenseNet

- 从模型结构上缓解梯度消失的另一种办法是使用DenseNet（稠密连接）。



稠密网络主要由2部分构成：稠密块（dense block）和过渡层（transition layer）。前者定义如何连接输入和输出，而后者则控制通道数量，使其不会太复杂。



密集连接是 DenseNet 核心思想，其通过建立不同层之间的连接关系，充分利用特征信息，缓解梯度消失现象，提升网络训练效果。此外，DenseNet 使用 Bottleneck Layer、Transition Layer 和 growth rate（即 Dense Block 中各层的输出通道数）控制通道数量，在减少参数和抑制过拟合的同时，大幅减少计算量。

- <https://arxiv.org/pdf/1608.06993>
- 为什么选择拼接？
- 特征多样性：拼接保留了所有先前层的独立特征，避免信息混合（累加可能导致特征被覆盖）。
- 梯度流动：浅层特征能直接传递到深层，缓解梯度消失。
- 显存代价：拼接会增加显存占用，但通过较小的增长率（如 $k=32$ ）和过渡层（压缩通道）来平衡。

基于CNN的图像分类模型选型

- 图像分类的模型结构相对简单，开发者可以根据自己实际的应用场景选择模型。

选择场景	建议模型
小规模且特征简单的数据集	GoogLeNet或者MobileNet，根据数据量及复杂程度，选择特征提取的网络层数。
中等规模且特征中等复杂的数据集	VGG、ResNet、GoogLeNet等，同样根据数据复杂程度选择特征提取的网络层数。
大规模高维度复杂数据集	ResNet系列，可以根据数据复杂程度选择不同的网络层数。例如数据复杂，可以选择ResNet51或者更深度的网络结构。
实时性要求高的轻量化场景	ShuffleNet，其参数量较小，并且在部署时可以进行量化或者剪枝。

如何设计一个CNN图像分类模型？

- 在卷积神经网络发展的中，卷积是其核心技术，它降低了神经网络的参数量，并且可以提取局部特征，因此在图像分类领域，卷积神经网络是必不可少的神经网络结构。
- 增加神经网络能力可以从两个方面出发：
 - 增加网络深度，例如AlexNet到VGG的发展思想
 - 增加网络宽度，例如Inception结构
- 如何让网络设计的更加深（能力更强）：残差思想。它可以让网络层数设计的更深，同时还可以被较好的训练。
- 损失函数：对于多分类问题，通常使用的是多类别交叉熵损失（Multiclass Cross-Entropy Loss），也被称为softmax loss，因为它经常与softmax激活函数一起使用作为输出层。
- 激活函数：随着网络深度增加，模型的激活函数基本以ReLU系列为主。它们可以减少梯度消失出现的概率，使得更深的模型可以被训练。但是，这种选择也不是被固定不变的，因为ReLU函数在0点不可导，因此也可以尝试使用sigmoid等函数来代替，效果可以根据实验结果来验证。

目录

1. AI模型选型及设计分析

- 基于卷积神经网络的视觉模型
 - 图像分类任务
 - 目标检测任务
- 基于Transformer的大语言模型
- 基于Transformer的视觉模型
- 基于Transformer的多模态大模型

2. 模型选型与设计总结

从图像分类到目标检测的复杂任务转换



这只动物
是什么？

模型

熊猫



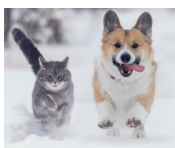
这幅图片是
什么动物？

模型

？

前述模型基本上都只能判断一张图片里包含什么，但是在日常生活中，基本上拍摄的图像都包含多种物体，那该如何判断图片里的每个个体呢？

目标检测算法



对于日常场景中，左侧这幅包含多个物体的图片是更常见的，如果需要检测图中有哪些动物，需要做哪几步呢？



1

确定图片中每个物体的位置在哪里



2

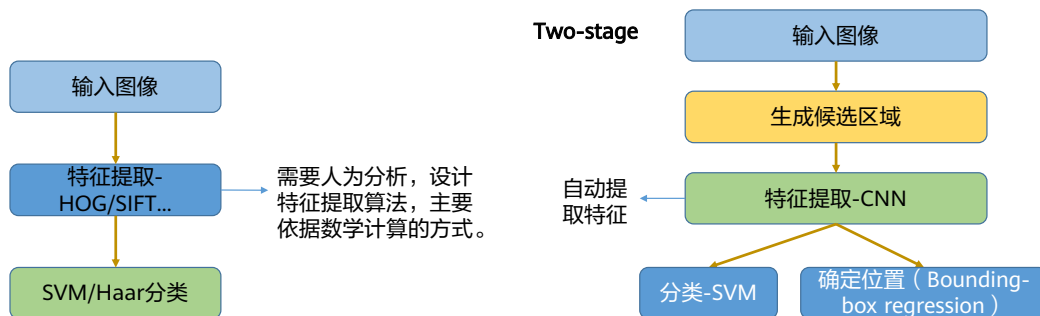
判断找到的每个物体分别是什么



目标检测分为两大系列 – R-CNN系列和YOLO系列，R-CNN系列是基于区域检测的代表性算法，YOLO是基于区域提取的代表性算法，另外还有著名的SSD是基于前两个系列的改进。

R-CNN (Region-based Convolutional Neural Networks)

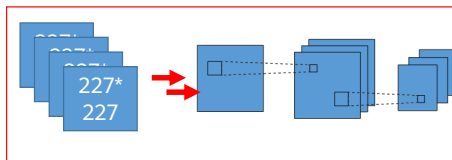
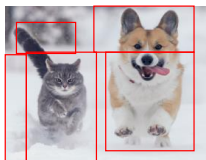
- 在RCNN之前，目标检测通常基于手工设计的特征和传统的机器学习算法，如SVM（支持向量机）和随机森林等。
- RCNN通过引入深度学习的卷积神经网络（CNN），利用其强大的特征学习能力，极大地改进了目标检测的准确性和性能。



- https://openaccess.thecvf.com/content_cvpr_2014/papers/Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper.pdf
- 2015

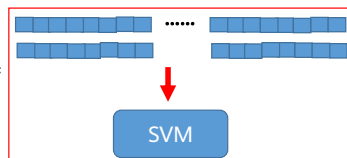
R-CNN的模型设计

1. 生成推荐区域，通过 selective search方法选取出 2000个候选框。然后将选出来的矩形区域转换成正方形。



2. 将每个候选区域送入 CNN（ AlexNet、VGG ）中提取一定维度（如4096）的特征。

R-CNN识别精度相对传统方法高，但是生成候选框和大量特征提取，使得速度太慢。



3. 将提取的特征送入 SVM 判断属于哪类

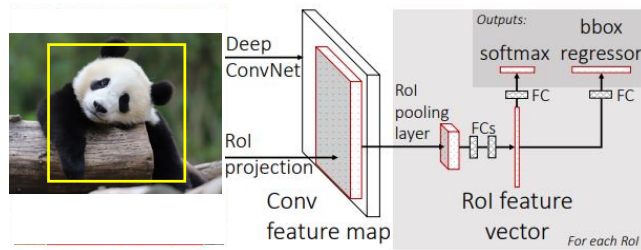
4. 候选区域可能不准确，使用回归方法校准



Fast R-CNN

- 候选区生成：使用Selective Search算法，在输入图像上生成2k个候选区
- 特征提取：将整张图像传入CNN提取特征
- 候选区域特征：利用RoIPooling分别生成每个候选区域的特征
- 候选区域分类和回归：利用提取的特征，对每个候选区域进行分类和边框回归

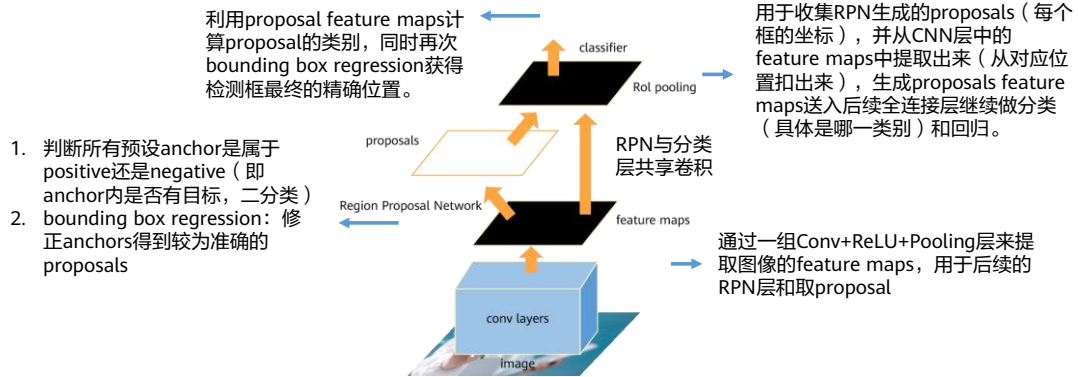
- Selective Search依然太慢
- RoI Pooling操作精度不高



- <https://arxiv.org/pdf/1504.08083>

Faster R-CNN

- 鉴于Fast R-CNN的缺点，Faster R-CNN对其进一步改进。

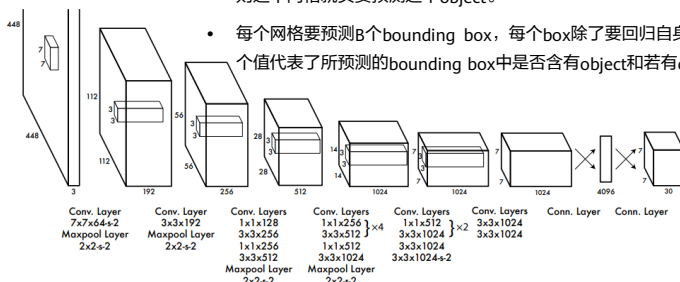


Faster R-CNN降低了候选框的选择时间，并且回归框预测更加准确，但是这种two-stage的检测方式，依然需要较高的计算资源，检测较慢。

- <https://arxiv.org/pdf/1506.01497>

YOLO-V1 You Only Look Once: Unified, Real-Time Object Detection

- 将整张图片作为网络的输入，直接在输出层对bounding box的位置和所属类别进行回归。
- 将一幅图像分成 $S \times S$ 个网格(Grid Cell)，如果某个object的中心落在某个网格中（通过ground-truth框确定），则这个网格就负责预测这个object。
- 每个网格要预测B个bounding box，每个box除了要回归自身的位置之外，还要附带预测一个confidence值。这个值代表了所预测的bounding box中是否含有object和若有object，这个object预测得有多准的两重信息。



- 得到所有的预测结果后，应用非极大值抑制来去除冗余的边界框，主要是针对同一物体可能产生的多个重叠较高的检测框，仅保留得分最高的那个作为最终输出。

- YOLO-V1使用了one-stage检测方案，采用了leaky ReLU作为激活函数。
- 检测速度快，背景的误检率低，并且通用性强。但是它的检测准确率不佳，对小目标识别比较差。

- <https://arxiv.org/pdf/1506.02640>
- 与Faster R-CNN网络相比，虽然后者也是使用整张图片作为输入，但是它采用了RCNN那种区域预测+分类的思想，把提取proposal的步骤放在了CNN中实现，而YOLO则采用直接回归的思路，将目标定位和目标类别预测整合于在单个神经网络模型中。
- 在Pascal VOC中，YOLO检测系统的图像输入为 448×448 ， $S=7$ ， $B=2$ ，一共有20个class（ $C=20$ ），输出就是 $7 \times 7 \times 30$ 的一个tensor。这个是怎么算出来的呢？

V1的loss

- 损失函数使用的为均方误差，位置、类别、置信度全部的损失都加在一起。

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

Bounding box loss

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\ + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2$$

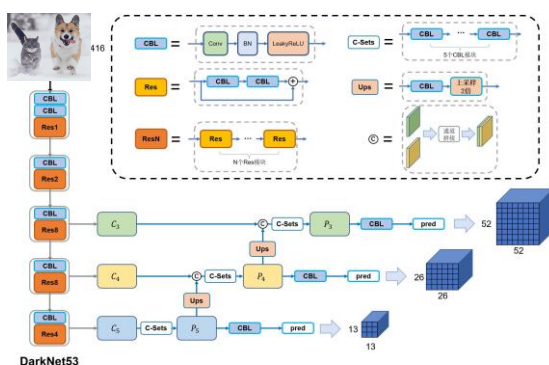
Confidence loss

$$+ \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

classes loss

YOLO-V3架构

- 目标检测任务极具挑战性的一个很大原因是因为原始图像中可能包含的目标物体大小不确定。V3使用了特征金字塔（Feature Pyramid Network，FPN），从3个不同尺度的融合特征上进行目标预测。
- 每个尺度的特征图设置3个常用比例的锚框，通过聚类的方式选择最优。

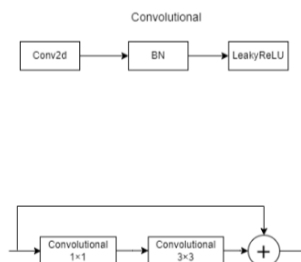


V3提高了检测准确度和小目标的检出率，但它也增加了计算量，而且需要手动设置锚框。

YOLO-V3主干网络-Darknet-53

- YOLO-V3在之前的版本上做了升级，它使用了名为DarkNet53的网络结构作为backbone，并且在此结构不同层输出不同大小的特征图，这在一定程度上解决了不同大小物体的检测问题。
- 在这个网络中还使用了残差连接，并且使用stride=2的卷积代替了Maxpooling。

	Type	Filters	Size	Output
1x	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	128 × 128
2x	Residual			
	Convolutional	128	3 × 3 / 2	64 × 64
	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
8x	Residual			
	Convolutional	256	3 × 3 / 2	32 × 32
	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
8x	Residual			
	Convolutional	512	3 × 3 / 2	16 × 16
	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
4x	Residual			
	Convolutional	1024	3 × 3 / 2	8 × 8
	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			
	Avgpool		Global	
	Connected		1000	
		Softmax		



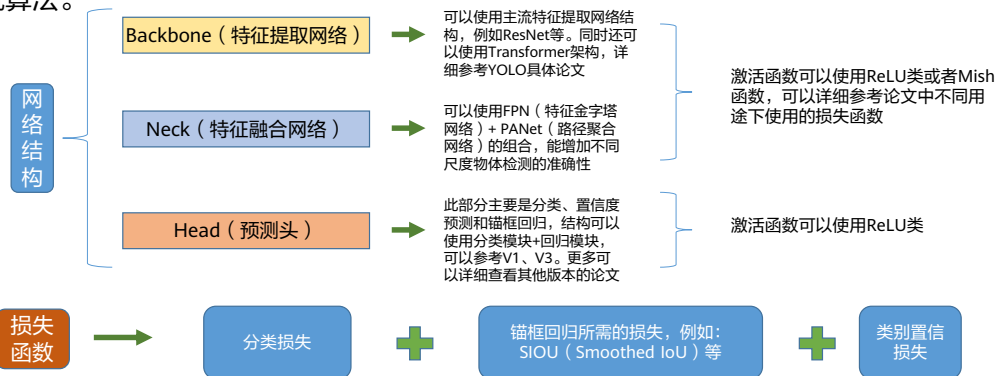
- <https://arxiv.org/pdf/1804.02767>

YOLO系列算法的发展

- YOLO系列算法目前是目标检测领域最受欢迎的算法之一，并且还在不断的迭代。目前YOLO算法已经发展到V13版本，每个版本都有不同的改进。这些改进基本都是优化之前版本的不足，或者引入一些新的技术等。
- 例如：
 - V5版本中增加了自适应锚框计算（自动学习数据集的最佳锚框尺寸），这就弥补了V4版本的手动设置锚框的缺点；
 - V6优化了损失函数、V7使用了更高效的主干网络.....
 - V11引入注意力机制，增强了特征提取能力，以实现更精确的目标检测.....

目标检测算法的设计

- 随着YOLO模型的发展，它已经改进了检测效果不佳的缺点，并成为了目标检测领域的主流算法。



参见YOLO V1版本的损失函数组成，然后根据不同论文中给出优化，选择不同类型的损失函数。

目录

1. AI模型选型及设计分析

- 基于卷积神经网络的视觉模型
- 基于Transformer的大语言模型
 - Transformer架构详解
 - 基于Transformer的大语言模型
- 基于Transformer的视觉模型
- 基于Transformer的多模态大模型

2. 模型选型与设计总结

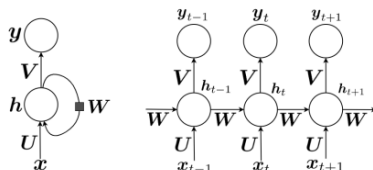
自然语言类任务

- 人类不仅期待AI技术能像人一样观察世界，同时也希望它能听懂、看懂人类的语言。在实际的场景中，众多的语言类任务期待AI来解决！
- 那AI算法是如何来决绝语言类任务呢，它们的结构又是如何的呢？接下来分析一下。



序列模型

- 对应之前图像任务，输入数据基本上没有关联，即前面的数据对后面数据的判断没有影响。但是对于很多视频、文本等数据，前后是有联系的，对后面数据的判读需要前面的数据支持。而CNN在解决这类问题上效果不佳。
- RNN等一系列算法被提出，这类模型是在预测后面数据时，会把前面的数据加权与当前数据作为此刻的输入，保证前面信息的传递，增加后续数据预测的准确性。



- 但是这类算法有一些缺陷：并行程度不高，导致训练时间长；序列过长，会导致前面数据遗忘，因此数据记忆能力较弱。而Transformer的出现，在一定程度上解决了上述问题。

- RNN: <https://arxiv.org/pdf/2304.11461>

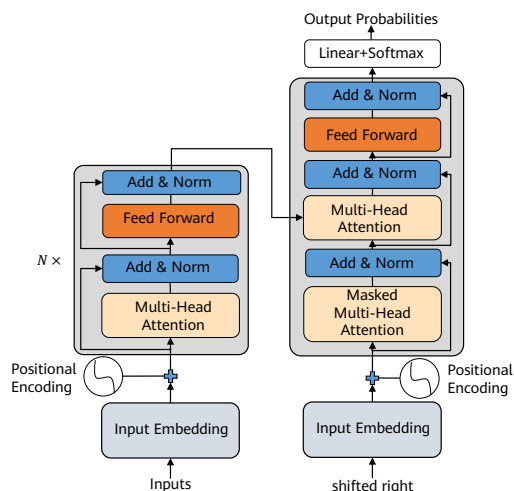
Transformer的诞生

- 2017年，谷歌机器翻译团队发表的《Attention is All You Need》中，提出了Transformer，完全抛弃了RNN（Recurrent Neural Network，循环神经网络）和CNN（Convolutional Neural Networks，卷积神经网络）等网络结构，而仅仅采用Attention机制来进行机器翻译任务，并且取得了很好的效果，注意力机制也成为了大家的研究热点。



- Transformer是谷歌在2017年做机器翻译任务的《Attention is all you need》的论文中提出的，引起了相当大的反响，很多研究已经证明了Transformer提取特征的能力是要远强于LSTM（Long Short Term Memory，长短期记忆网络）的。
- Transformer解决的主要问题是捕捉文本特征，通过的方式就是Attention，这也是为什么Transformer的提出者，将他们的文章命名为《Attention is All You Need》的原因。
- 我们知道，ELMo（Embeddings from Language Model）使用了LSTM而不是新贵Transformer，如果ELMo采取Transformer作为特征提取器，那么估计Bert的反响远不如现在的这种火爆场面。另外一点，ELMo采取双向拼接这种融合特征的能力可能比Bert一体化的融合特征方式弱，但是，这只是一种从道理推断产生的怀疑，目前并没有具体实验说明这一点。
- 参考：<https://zhuanlan.zhihu.com/p/51679783>。

Transformer模型结构



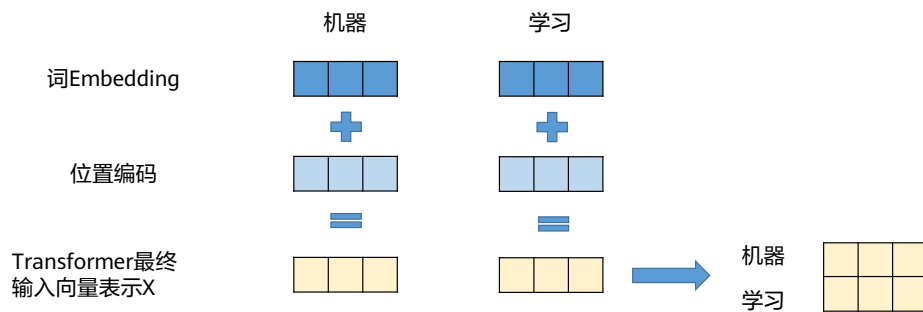
- Transformer由Encoder和Decoder两部分组成。
- Encoder包含N个相同的layer，layer指的就是图中左侧的单元。最左边有个“ $N \times$ ”，指的是N个，如果N=6的话，对应的就是6个。
- Decoder也包含N=6个相同的layer。

参考文章: *Attention is All You Need*

- Encoder中的layer结构都是相同的，但他们没有共享参数。每个layer都可以分解成两个sub-layer。每个sub-layer的输出维度 $d_{\text{model}}=512$ 。
- 从编码器输入的句子首先会经过一个自注意力（Self-Attention）层，这层帮助编码器在对每个单词编码时关注输入句子的其他单词。
- 自注意力层的输出会传递到前馈（feed-forward）神经网络中。每个位置的单词对应的前馈神经网络都完全一样（译注：另一种解读就是一层窗口为一个单词的一维卷积神经网络）。
- 解码器中也有编码器的自注意力（Self-Attention）层和前馈（feed-forward）层。除此之外，这两个层之间还有一个注意力层，用来关注输入句子的相关部分（和seq2seq模型的注意力作用相似）。
- 参考链接：https://blog.csdn.net/longxincheng_ml/article/details/86533005。

Transformer工作流程：获取输入

- Transformer中的输入X是由单词Embedding和位置编码（Positional Encoding）相加得到的。最终，每个单词都被嵌入为一个512维的向量。



- 每一行是一个单词的表示x。

单词Embedding

- 单词的Embedding有多种方式可以得到，例如可以采用Word2Vec、GloVe等算法预训练得到，也可以在Transformer中训练得到。
- Word2Vec是一种基于统计方法来获得词向量的方法，它是2013年由谷歌的Mikolov提出的一套词嵌入方法。这种算法有2种训练模式：
 - 通过上下文来预测当前词；
 - 通过当前词来预测上下文。
- GloVe是对Word2Vec方法的扩展，它将全局统计和Word2Vec的基于上下文的学习结合了起来。

- Word2Vec在2018年之前比较主流，但是随着BERT、GPT2.0的出现，这种方式已经不算效果最好的方法了。
- GloVe的全称叫Global Vectors for Word Representation，它是一个基于全局词频统计（count-based & overall statistics）的词表征（word representation）工具，它可以把一个单词表达成一个由实数组成的向量，这些向量捕捉到了单词之间一些语义特性，比如相似性（similarity）、类比性（analogy）等。我们通过对向量的运算，比如欧几里得距离或者cosine相似度，可以计算出两个单词之间的语义相似性。

位置编码

- 对于Transformer来说，由于句子中的词语都是同时进入网络进行处理，顺序信息在输入网络时就已经丢失。因此，Transformer需要额外的处理来告知每个词语的相对位置。其中的一个解决方案，就是论文中提到的位置编码（Positional Encoding），将能表示位置信息的编码添加到输入中，让网络知道每个词的位置和顺序。
- 总而言之，Positional Encoding就是句子中词语相对位置的编码，让Transformer保留词语的位置信息。

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

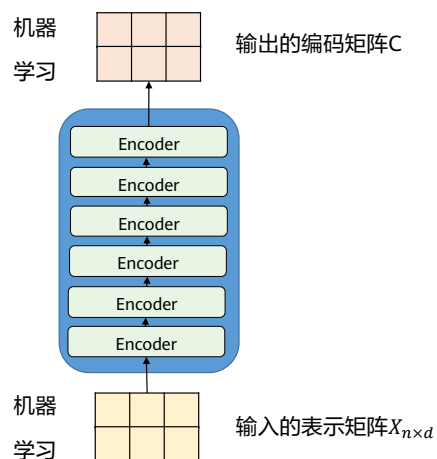
$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

其中，pos表示单词在句子中的位置，d表示PE的维度（与词Embedding一样），2i表示偶数的维度，2i+1表示奇数维度（即 $2i \leq d$ ， $2i+1 \leq d$ ）。

- 使用这种公式计算PE有以下的好处：
 - 使PE能够适应比训练集里面所有句子更长的句子。
 - 可以让模型容易地计算出相对位置，对于固定长度的间距k， $PE(pos+k)$ 可以用 $PE(pos)$ 计算得到。

Transformer工作流程：将输入传入Encoder

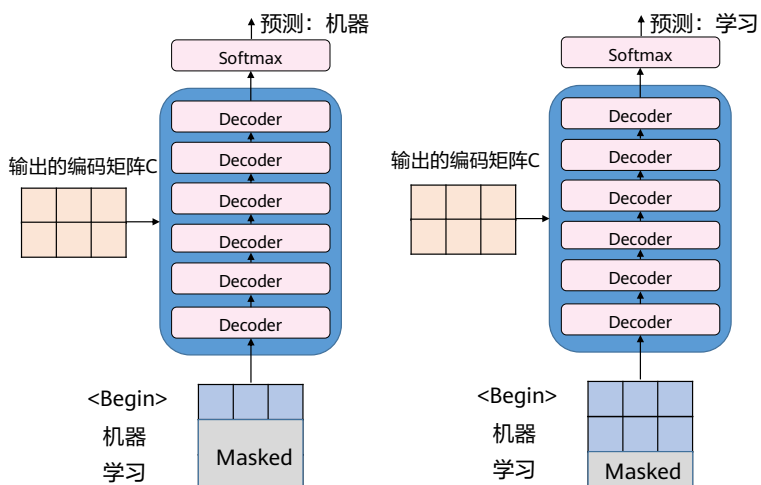
- 将得到的单词表示向量矩阵传入Encoder中，经过6个Encoder block后可以得到句子所有单词的编码信息矩阵C。
- 每一个Encoder block输出的矩阵维度与输入完全一致。



- $X_{n \times d}$: 输入矩阵，n是句子中单词数量，d表示向量的维度，论文中d=512。

Transformer工作流程：将编码矩阵传递给Decoder

- 将Encoder输出的编码信息矩阵C传递到Decoder中，Decoder依次会根据当前翻译过的1~i的单词翻译第i+1个单词。
- 在整个过程中，翻译到第i+1个单词的时候，需要通过Mask（掩盖）操作遮盖住第i+1之后的单词。

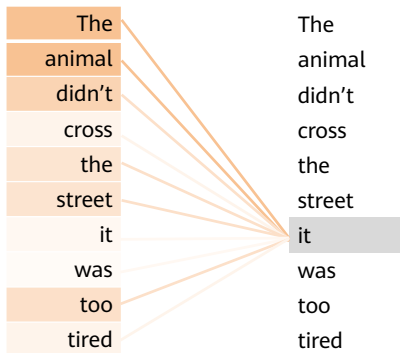


自注意力机制：Self-Attention

- 例如，下列句子是我们想要翻译的输入句子：
 - The animal didn't cross the street because it was too tired.
- 这个“it”在这个句子是指什么呢？它指的是“street”还是“animal”呢？这对于人类来说是一个简单的问题，但是对于算法则不是。
- 当模型处理这个单词“it”的时候，自注意力机制会允许“it”与“animal”建立联系。

自注意力机制的强大作用

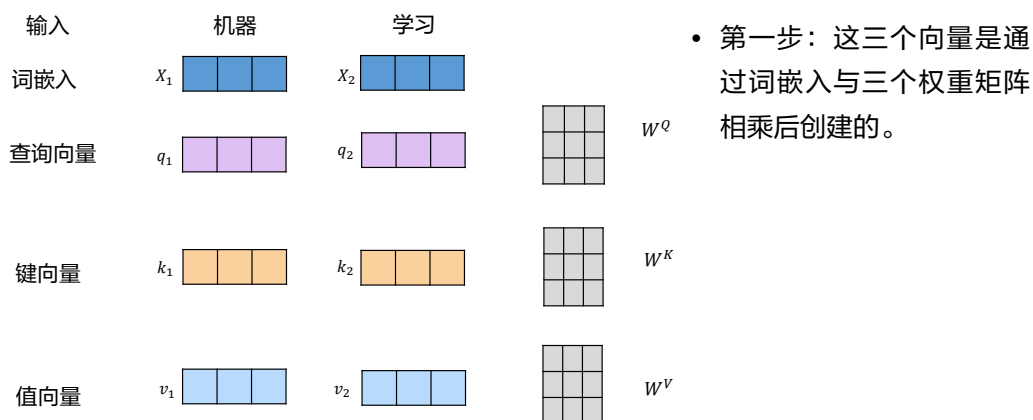
Layer: 5 Attention: Input-Input



- 随着模型处理输入序列的每个单词，自注意力会关注整个输入序列的所有单词，帮助模型对本单词更好地进行编码。

- 如果你熟悉RNN（循环神经网络），回忆一下它是如何维持隐藏层的。RNN会将它已经处理过的前面的所有单词/向量的表示与它正在处理的当前单词/向量结合起来。而自注意力机制会将所有相关单词的理解融入到我们正在处理的单词中。
- 当我们在编码器#5（栈中最上层编码器）中编码“it”这个单词的时候，注意力机制的部分会去关注“The Animal”，将它的表示的一部分编入“it”的编码中。

第一步：生成查询向量、键向量和值向量



- 这三个向量是通过词嵌入与三个权重矩阵相乘后创建的。这些新向量在维度上比词嵌入向量更低，他们的维度是64，而词嵌入和编码器的输入/输出向量的维度是512。但实际上不强求维度更小，这只是一种基于架构上的选择，它可以使多头注意力（multiheaded attention）的大部分计算保持不变。
- x_1 与 W^Q 权重矩阵相乘得到 q_1 ，就是与这个单词相关的查询向量。最终使得输入序列的每个单词都创建一个查询向量、一个键向量和一个值向量。
- 什么是查询向量、键向量和值向量？它们都是有助于计算和理解注意力机制的抽象概念。学员们对此也有疑问。

第二步：计算得分

输入

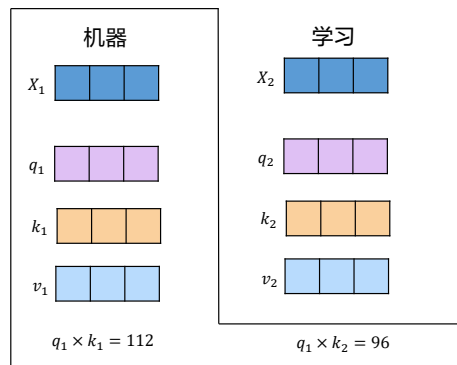
词嵌入

查询向量

键向量

值向量

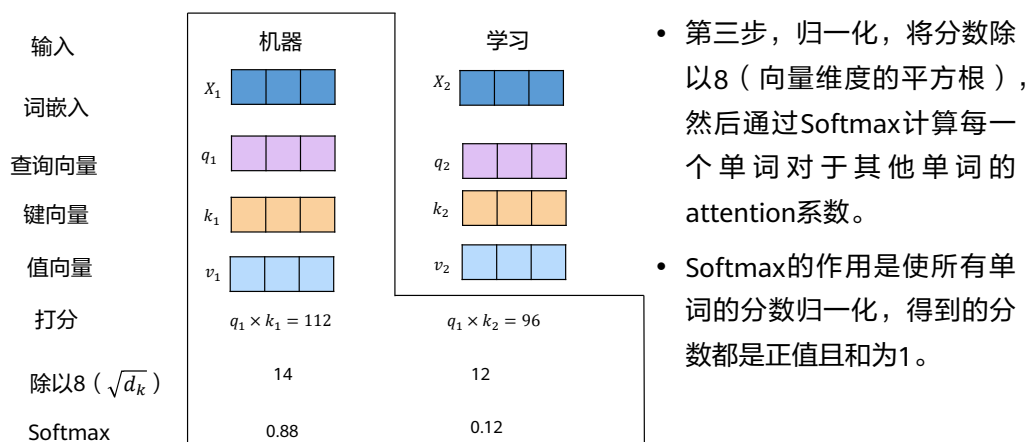
打分



- 第二步：计算得分。这里使用点积计算。

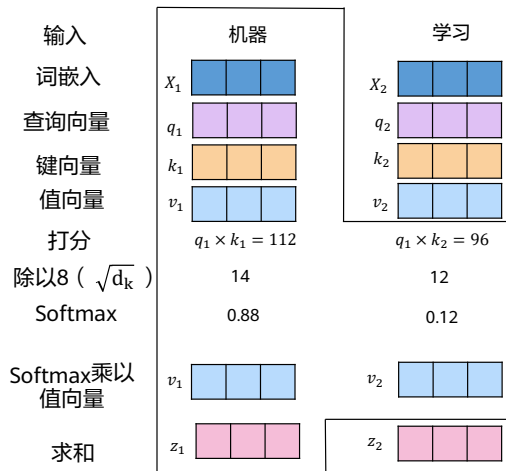
- 计算自注意力的第二步是计算得分。假设我们在为这个例子中的第一个词“机器”计算自注意力向量，我们需要拿输入句子中的每个单词对“机器”打分。这些分数决定了在编码单词“机器”的过程中有多重视句子的其它部分。这些分数是通过打分单词（所有输入句子的单词）的键向量与“机器”的查询向量相点积来计算的。所以如果我们是处理位置最靠前的词的自注意力的话，第一个分数是 q_1 和 k_1 的点积，第二个分数是 q_1 和 k_2 的点积。

第三步：归一化



- 8是论文中使用的键向量的维数64的平方根，这会让梯度更稳定。这里也可以使用其它值，8只是默认值。
- 这个Softmax分数决定了每个单词对编码当下位置（“机器”）的贡献。显然，已经在这个位置上的单词将获得最高的Softmax分数，但有时关注另一个与当前单词相关的单词也会有帮助。

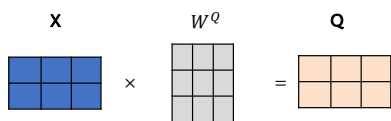
第四步：对加权值向量求和

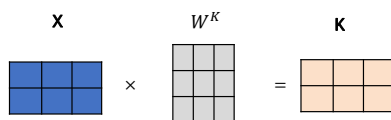


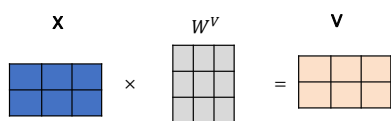
- 第四步，在编码某个单词时，将所有单词的表示（值向量）进行加权求和，而权重是通过该词的表示（键向量）与被编码词表示（查询向量）的点积并归一化得到，即第三步得到的结果。
- 然而实际中，这些计算是以矩阵形式完成的。

- 将每个值向量乘以Softmax分数（这是为了准备之后将它们求和）。这里的直觉是希望关注语义上相关的单词，并弱化不相关的单词（例如，让它们乘以0.001这样的小数）。
- 对加权值向量求和（自注意力的另一种解释就是在编码某个单词时，就是将所有单词的表示（值向量）进行加权求和，而权重是通过该词的表示（键向量）与被编码词表示（查询向量）的点积并通过Softmax得到），然后即得到自注意力层在该位置的输出（在我们的例子中是对应第一个单词）。
- 这样自注意力的计算就完成了。得到的向量就可以传给前馈神经网络。然而实际中，这些计算是以矩阵形式完成的，以便算得更快。那我们接下来就看看如何用矩阵实现的。

通过矩阵运算实现自注意力机制（1）

$$\begin{matrix} & \mathbf{X} \\ \mathbf{X} & \times \mathbf{W}^Q = \mathbf{Q} \end{matrix}$$


$$\begin{matrix} & \mathbf{X} \\ \mathbf{X} & \times \mathbf{W}^K = \mathbf{K} \end{matrix}$$


$$\begin{matrix} & \mathbf{X} \\ \mathbf{X} & \times \mathbf{W}^V = \mathbf{V} \end{matrix}$$


- 第一步是计算查询矩阵、键矩阵和值矩阵。为此，我们将输入句子的词嵌入到矩阵 $\mathbf{X}$ 中，将其乘以我们训练的权重矩阵（ $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$ ）。
- $\mathbf{X}$ 矩阵中的每一行对应输入句子中的一个单词。

通过矩阵运算实现自注意力机制（2）

$$Attention(Q, K, V) = softmax\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) \times V = Z$$

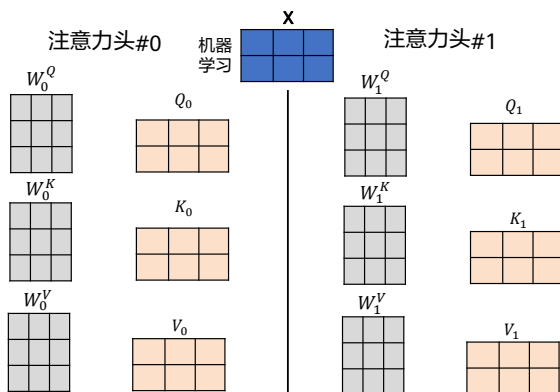
- 最后，由于我们处理的是矩阵，我们可以将第二步到第四步合并为一个公式来计算自注意力层的输出。

多头注意力机制

- 论文为了进一步完善自注意力层，增加一种多头注意力机制（Multi-head Attention）。
- 多头注意力是多组自注意力构成的组合，自注意力机制能帮助建立包括上下文信息的词特征表达，多头注意力能帮助学习到多种不同类型的上下文影响情况。比如“今天阳光不错，适合出去跑步”，在不同情景下，“今天”同“阳光”、“跑步”的相关性是不同的。头越多，越有利于捕获更大更多范围的相关性特征，增加模型的表达能力。

- 论文为了进一步完善自注意力层，增加一种叫做“多头”注意力（“multi-headed attention”）的机制。
- 通过这种机制可以进一步提高注意力层的性能。

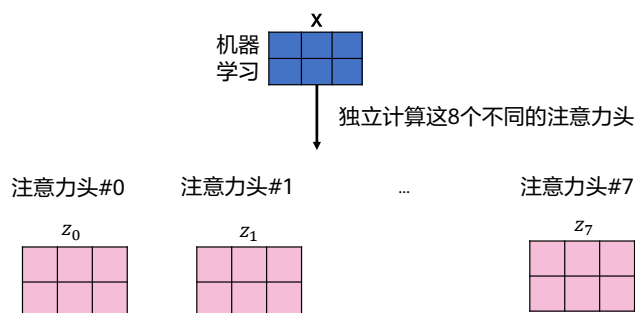
提高注意力层的性能



- 在两个方面提高了注意力层的性能：一方面扩展了模型专注于不同位置的能力；另一方面，它给出了注意力层的多个“表示子空间”。

- 通过增加一种叫做“多头”注意力（“multi-headed” attention）的机制，论文进一步完善了自注意力层，并在两方面提高了注意力层的性能：1.它扩展了模型专注于不同位置的能力。在上面的例子中，虽然每个编码都在 z_1 中有或多或少的体现，但是它可能被实际的单词本身所支配。如果我们翻译一个句子，比如“The animal didn’t cross the street because it was too tired”，我们会想知道“it”指的是哪个词，这时模型的“多头”注意机制会起到作用。2.它给出了注意力层的多个“表示子空间”（representation subspaces）。接下来我们将看到，对于“多头”注意机制，我们有多组查询/键/值权重矩阵集（Transformer使用八个注意力头，因此我们对于每个编码器/解码器有八个矩阵集合）。这些集合中的每一个都是随机初始化的，在训练之后，每个集合都被用来将输入词嵌入（或来自较低编码器/解码器的向量）投影到不同的表示子空间中。

八次权重矩阵运算

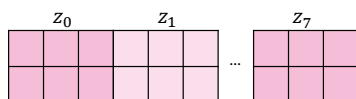


- 如果我们做与上述相同的自注意力计算，只需八次不同的权重矩阵运算，我们就会得到八个不同的Z矩阵。

- 在“多头”注意机制下，我们为每个头保持独立的查询/键/值权重矩阵，从而产生不同的查询/键/值矩阵。和之前一样，我们拿X乘以WQ/WK/WV矩阵来产生查询/键/值矩阵。
- 如果我们做与上述相同的自注意力计算，只需八次不同的权重矩阵运算，我们就会得到八个不同的Z矩阵。
- 这给我们带来了一点挑战。前馈层不需要8个矩阵，它只需要一个矩阵（由每一个单词的表示向量组成）。所以我们需要一种方法把这八个矩阵压缩成一个矩阵。那该怎么做？

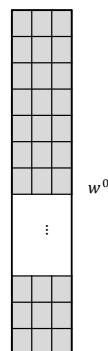
八个矩阵压缩成一个矩阵

1、将所有注意力头拼接起来

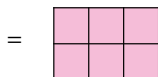


2、乘以权重矩阵 w^0 （随模型一起训练获得）

×



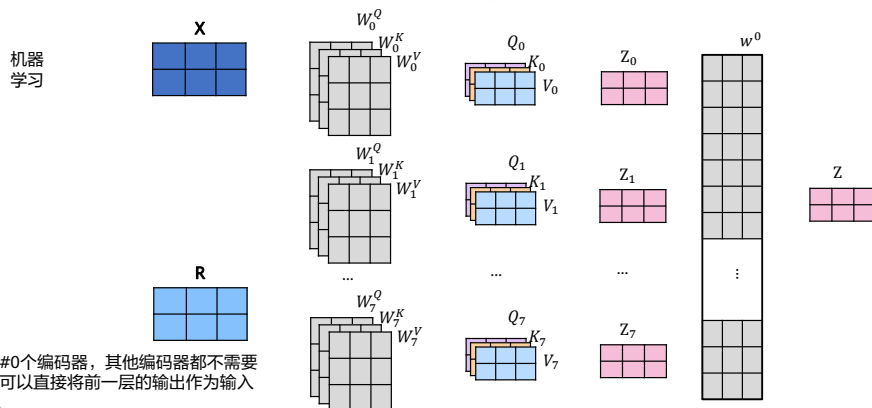
3、结果是一个融合所有注意力头信息的矩阵 Z ，我们可以将其送到前馈神经网络



- 可以直接把这些矩阵拼接在一起，然后用一个附加的权重矩阵 w^0 与它们相乘。

计算过程集中在一起

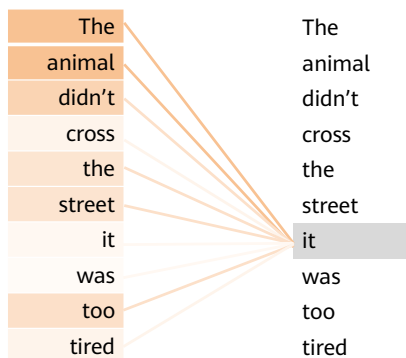
- 1、这是我们的输入句子
- 2、编码每一个单词
- 3、将其分成8个头，将矩阵X或R乘以各个权重矩阵
- 4、通过输出查询/键/值 (Q/K/V) 矩阵计算注意力
- 5、将所有注意力矩阵Z拼接起来，然后乘以权重矩阵 w^0 来产生该层的输出



- 得到8个输出矩阵 Z_0-Z_7 之后，Multi-Head Attention将它们拼接在一起（Concat），然后传入一个Linear层，得到Multi-Head Attention最终的输出 Z 。
- Multi-Head Attention输出的矩阵 Z 与其输入的矩阵 X 的维度是一样的。
- 既然我们已经摸到了注意力机制的这么多“头”，那么让我们重温之前的例子，看看我们在例句中编码“it”一词时，不同的注意力“头”集中在哪里。

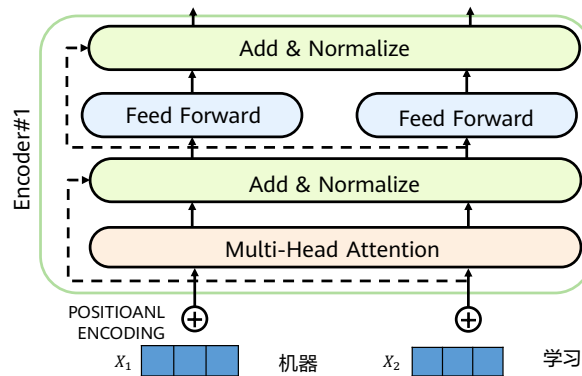
重温之前的例子

Layer: 5 Attention: Input-Input



- 当我们编码“it”一词时，一个注意力头集中在“animal”上，而另一个则集中在“tired”上，从某种意义上说，模型对“it”一词的表达在某种程度上是“animal”和“tired”的代表。

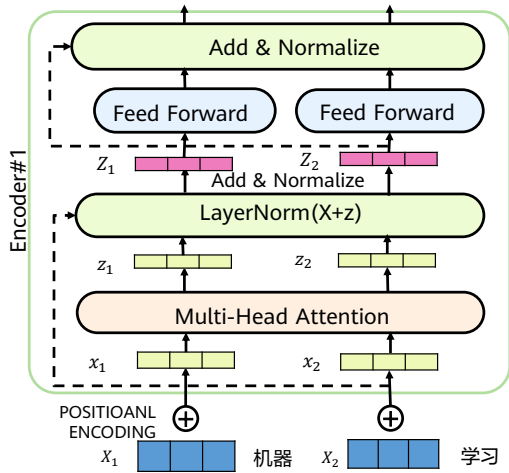
编码器Encoder



- 在每个编码器中的每个子层（自注意力、前馈网络）的周围都有一个残差连接，并且都跟随着一个“层-归一化”步骤。

- 残差连接和层归一化。
- 在每个编码器中的每个子层（自注意力、前馈网络）的周围都有一个残差连接，并且都跟随着一个“层-归一化”步骤（<https://arxiv.org/abs/1607.06450>）。

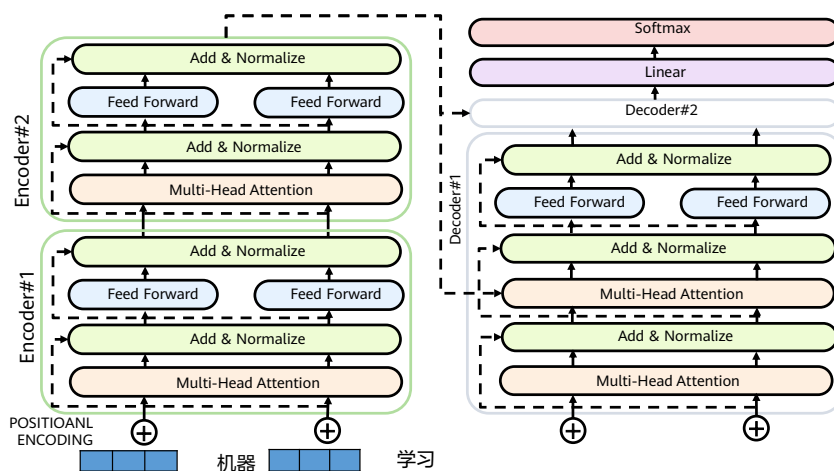
细化这个模块



- 如果我们去可视化这些向量以及这个和自注意力相关联的层-归一化操作，那么看起来就像左边这张图一样。

- 如果我们去可视化这些向量以及这个和自注意力相关联的层-归一化操作，那么看起来就像左边这张图一样。

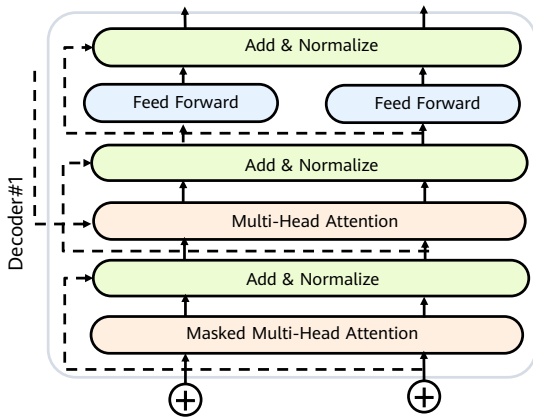
一个2层的Encoder



- 如果我们想象一个2层编码-解码结构的transformer，它看起来会像这张图一样。

解码器Decoder

- Decoder:
 - 包含两个Multi-Head Attention子层。
 - 第一个Multi-Head Attention子层采用了Masked操作。
 - 第二个Multi-Head Attention子层的K, V矩阵使用Encoder的编码信息矩阵C进行计算, 而Q使用上一层的输出进行计算。



- Decoder与Encoder的区别。

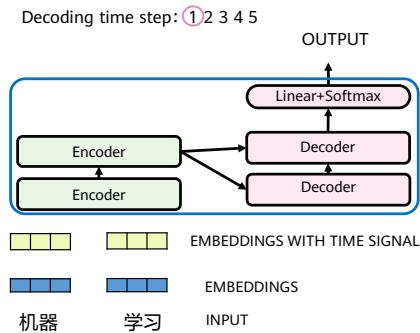
第一个Multi-Head Attention

- Decoder block的第一个Multi-Head Attention采用了Masked操作，因为在翻译的过程中是顺序翻译的，即翻译完第 i 个单词，才可以翻译第 $i+1$ 个单词。通过Masked操作可以防止第 i 个单词知道 $i+1$ 个单词之后的信息。

第二个Multi-Head Attention

- Decoder block第二个Multi-Head Attention变化不大，主要的区别在于其中Self-Attention的K，V矩阵不是基于上一层输出计算的，而是使用Encoder的编码信息矩阵C计算的。
- 根据Encoder输出的编码信息矩阵C计算得到K和V，根据上一层的输出Z计算Q，其余计算方法与之前描述的一致。
- 这样做的好处是在Decoder的时候，每一个单词都可以利用到Encoder中所有单词的信息。

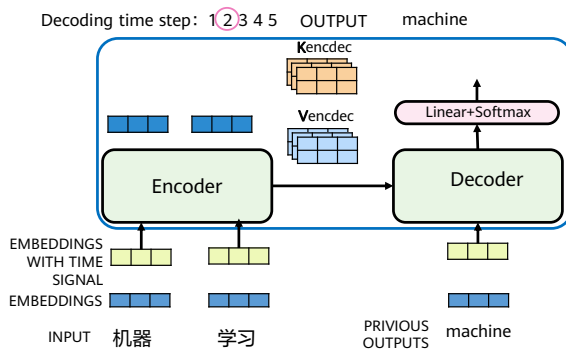
编码器输出到解码器



- 编码器通过处理输入序列开启工作。顶端编码器的输出之后会转化为一个包含向量K（键向量）和V（值向量）的注意力向量集。
- 在完成编码阶段后，则开始解码阶段。解码阶段的每个步骤都会输出一个序列的元素（在这个例子里，是英语翻译的句子）。

- 既然我们已经谈到了大部分编码器的概念，那么我们基本上也就知道解码器是如何工作的了。但最好还是看看解码器的细节。编码器通过处理输入序列开启工作。顶端编码器的输出之后会变转化为一个包含向量K（键向量）和V（值向量）的注意力向量集。这些向量将被每个解码器用于自身的“编码-解码注意力层”，而这些层可以帮助解码器关注输入序列哪些位置合适。

解码器不断输出



- 接下来的步骤重复了这个过程，直到到达一个特殊的终止符号，它表示transformer的解码器已经完成了它的输出。每个步骤的输出在下一个时间步被提供给底端解码器，并且就像编码器之前做的那样，这些解码器会输出它们的解码结果。另外，就像我们对编码器的输入所做的那样，会嵌入并添加位置编码给那些解码器，来表示每个单词的位置。

- 而那些解码器中的自注意力层表现的模式与编码器不同：在解码器中，自注意力层只被允许处理输出序列中更靠前的那些位置。在Softmax步骤前，它会把后面的位置给隐去（把它们设为-inf）。这个“编码-解码注意力层”工作方式基本就像多头自注意力层一样，只不过它是通过在它下面的层来创造查询矩阵，并且从编码器的输出中取得键/值矩阵。

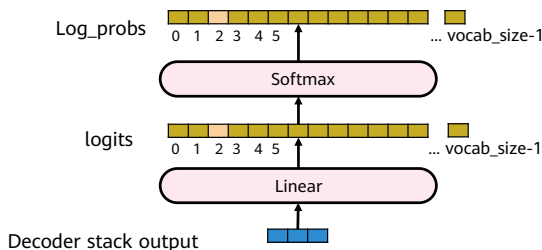
解码器输出到Linear层再到Softmax层

Which word in our vocabulary
is associated with this index?

machine

Get the index of the cell
with the highest value
(argmax)

2



- 解码组件最后会输出一个实数向量。我们如何把浮点数变成一个单词？这便是线性变换层要做的工作，它之后就是Softmax层。

- 解码组件最后会输出一个实数向量。我们如何把浮点数变成一个单词？这便是线性变换层要做的工作，它之后就是Softmax层。线性变换层是一个简单的全连接神经网络，它可以把解码组件产生的向量投射到一个比它大得多的、被称作对数几率（logits）的向量里。不妨假设我们的模型从训练集中学习一万个不同的英语单词（我们模型的“输出词表”）。因此对数几率向量为一万个单元格长度的向量——每个单元格对应某一个单词的分数。接下来的Softmax层便会把那些分数变成概率（都为正数、上限1.0）。概率最高的单元格被选中，并且它对应的单词被作为这个时间步的输出。

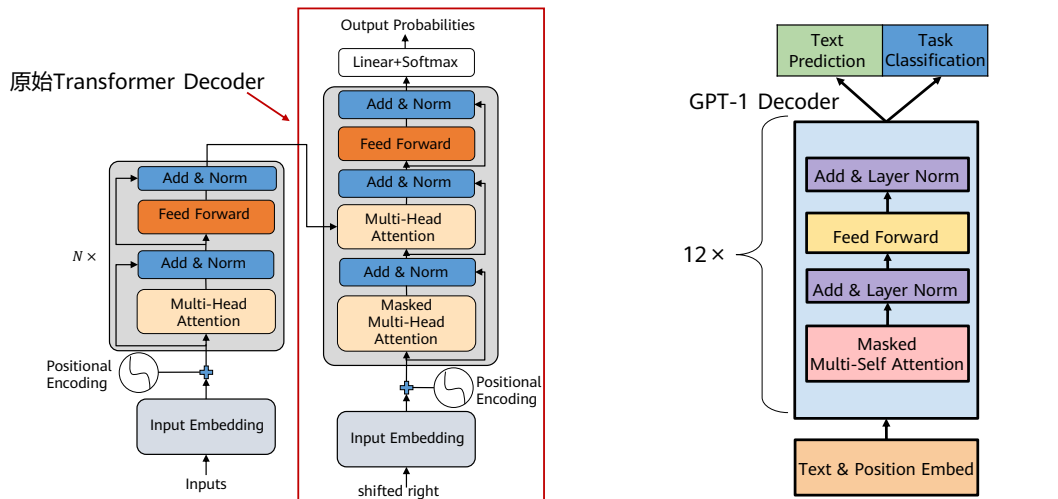
目录

1. AI模型选型及设计分析

- 基于卷积神经网络的视觉模型
- 基于Transformer的大语言模型
 - Transformer架构详解
 - 基于Transformer的大语言模型
- 基于Transformer的视觉模型
- 基于Transformer的多模态大模型

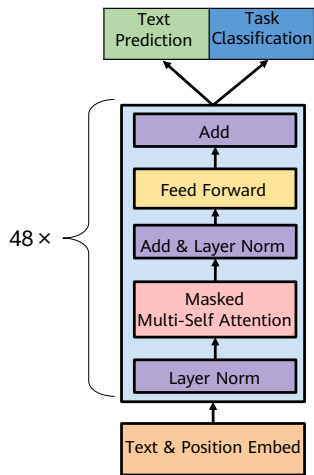
2. 模型选型与设计总结

GPT-1架构



- OpenAI的第一个版本的GPT-1当时称之为GPT。
- 使用Transformer而不使用LSTM在于，虽然预训练有助于捕获一些语言信息，但LSTM模型限制了他们在短期内的预测能力。相反，Transformer的选择使我们能够捕获更长远的语言结构，同时在其他不同的任务中泛化性能更好。
- GPT-1使用的是Transformer的Decoder框架。
- GPT-1保留了Decoder的Masked Multi-Attention层和Feed Forward层，并扩大了网络的规模。将层数扩展到12层，GPT-1还将Attention的维数扩大到768（原来为512），将Attention的头数增加到12层（原来为8层），将Feed Forward层的隐层维数增加到3072（原来为2048），总参数达到1.17亿。
- 对于位置编码的部分，实际上GPT-1和普通的Transformer的区别还是很大的，普通的Transformer的位置编码，是由余弦+正弦的方式表示出来的，而GPT-1中，采用与词向量相似的随机初始化，并在训练中进行更新，即是把每一个位置当做一个要学习的embedding来做。

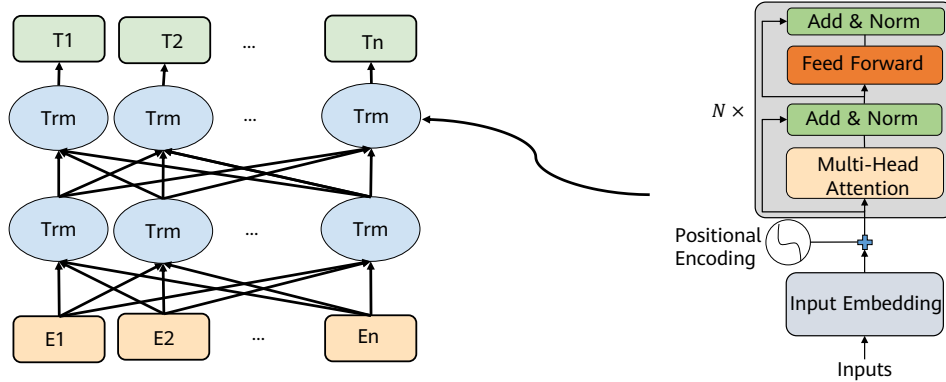
GPT-2架构



- 相对于GPT-1，GPT-2的架构变化有2点：
 - 层数更多，堆叠了48层Decoder（GPT-2 extra large）；
 - Normalization前置，分别放在了自注意力和全连接之前。

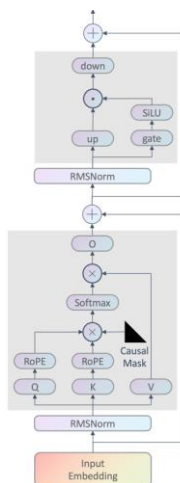
- 参考：<https://zhuanlan.zhihu.com/p/78153185>。
- 架构参考源码：
https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling_gpt2.py。
- GPT2开源，GPT3未开源。

BERT的结构图



- BERT的模型结构是一个双向的Transformer编码器，右图是一个Transformer的Encoder。

Llama1



- Llama 1是Meta在2023年2月发布的大语言模型，是当时性能非常出色的开源模型之一，有7B、13B、30B和65B四个参数量版本。
- Llama 模型与 GPT 类似，也是采用了 Decoder-only 的架构。在原始 Transformer 解码器的基础上，Llama进行了如下改动：

为了增强训练稳定性，
采用前置的RMSNorm
作为层归一化方法。

为了更好地建模长序列数据，
采用RoPE作为位置编码。

为了提高模型性能，采
用SwiGLU作为激活函数。

使用causal mask保证每个
位置只能看到前面的tokens

- Pre-normalization: 为了提高训练稳定性，LLaMA 对每个 transformer 子层的输入进行归一化，使用 RMSNorm 归一化函数，Pre-normalization 由Zhang和Sennrich (2019) 引入。使用 RMSNorm 的好处是不用计算样本的均值，速度提升了40%。

前置RMSNorm

- 在Transformer架构中，使用了LayerNorm，并且位置放在了输出位置。而在Llama 1中，将Normalization放在了靠近输入的位置，并且使用了Root Mean Square Layer Normalization（RMSNorm）。
- RMSNorm 是 LayerNorm 的变体，省略了均值计算，仅通过均方根（RMS）对输入特征进行归一化。其公式如下：

$$\text{RMS}(x) = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2}$$

x_i 是输入张量中的每个元素， N 是元素的总数。

归一化后的输出为：

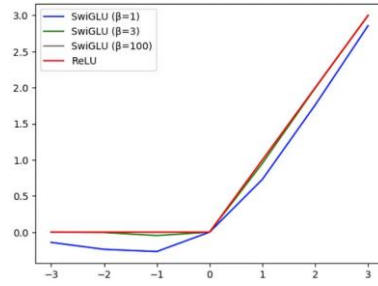
$$\hat{x} = \frac{x}{\text{RMS}(x)} \cdot \gamma + \beta$$

RMSNorm 通过缩放因子（可学习的参数）对输入进行缩放， γ 和 β 即可学习的参数。

前置可以提前将数据归一化，避免经过矩阵乘法导致的数据梯度消失或爆炸。RMSNorm减少了均值和方差的计算，降低了计算量，实验证明，与LayerNorm相比，它在效果不减的情况下，显著提高计算效率。同时这种计算方式更适合AI芯片的并行计算，进一步提高计算效率。

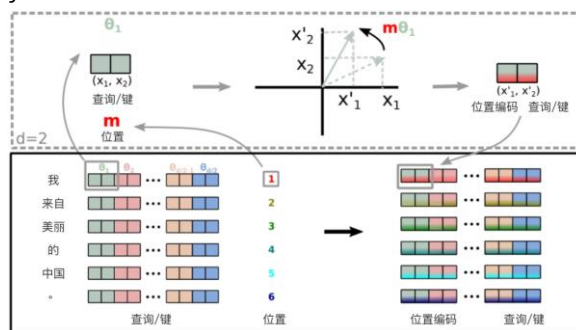
SwiGLU 激活函数

- 在激活函数的选择上，研究者们采用了 SwiGLU函数来替代传统的ReLU函数，这一改变旨在提升模型的性能。两者的核心差异在于：ReLU函数会将所有负数输入直接归零，而正数输入则保持不变。相比之下，SwiGLU函数含有一个可学习的参数 β ，能够调节函数的插值程度。随着 β 值的增大，SwiGLU的行为将逐渐接近 ReLU。



旋转式位置编码 (Rotary Positional Embeddings)

- RoPE的核心思想是将位置信息转化为向量的旋转角度，使模型能够通过向量的相对位置关系捕捉序列顺序。对于 token 序列中的每个词嵌入向量，首先计算其对应的 query 和 key 向量，然后对每个 token 位置都计算对应的旋转位置编码，接着对每个 token 位置的 query 和 key 向量的元素按照两两一组应用旋转变换，最后再计算 query 和 key 之间的内积得到 self-attention 的计算结果。

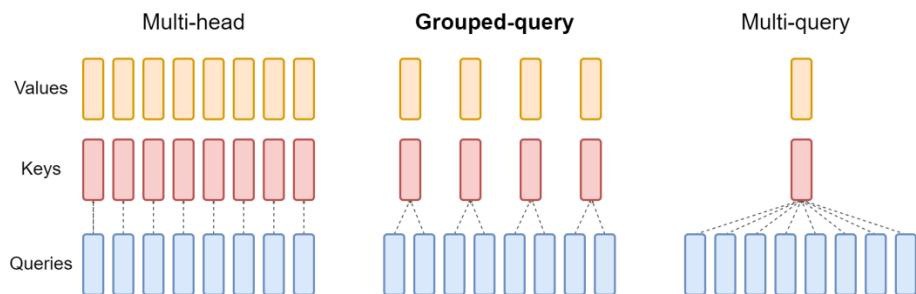


Llama系列的发展

- Meta不断的改进Llama模型，提升它们的效果。
- Llama-2在模型架构和分词器方面与Llama-1保持一致。然而，与Llama-1相比，Llama-2在上下文长度上做了扩展，将其提升至4000个token。此外，Llama-2在其34亿和70亿参数的版本中引入了分组查询注意力机制（ Grouped-Query Attention, GQA ），以提升模型性能。
- 相较于Llama 2， Llama-3在分词器的选用上做出了调整，从sentencepiece切换到了tiktoken，这一变更与GPT-4的实践相一致。此外，词汇表的规模也从32,000扩展至128,000个词条。为了增强模型的运算效率，Llama-3在8亿和70亿参数的版本中都应用了分组查询注意力机制（ Grouped-Query Attention, GQA ）。

GQA (Group Query Attention)

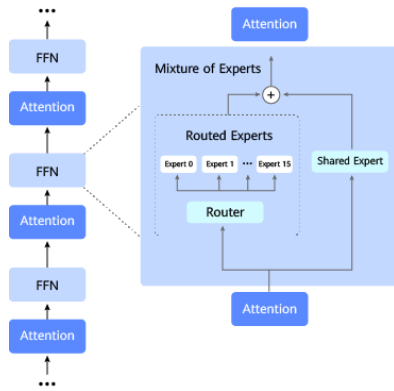
- 自回归模型生成回答时，需要前面生成的KV缓存起来，来加速计算。多头注意力机制 (MHA) 需要的缓存量很大， Multi-Query Attention指出多个头之间可以共享KV对。 Group Query Attention没有像MQA一样极端，将query分组，组内共享KV，效果接近 MHA，速度上与MQA可比较。



- <https://arxiv.org/pdf/2305.13245>

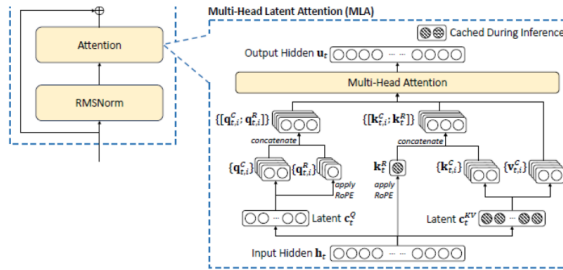
Llama 4

- Llama 4家族是Llama系列第一组使用MOE架构的模型。Llama 4模型家族采用“原生(native)”多模态设计，通过“早期融合(early fusion)”集成文本和视觉token，“联合预训练(jointly pre-train)”大量无标记文本、图片和视频。



- 架构上：
 - 每4层Transformer Block，其中一层的Attention部分不使用位置编码，也称为“NoPE(Positional Encoding)”方法。
 - Attention使用Chunked Attention。Chunked Attention的长度是8192。
 - NoPE层使用Scaled Softmax。
 - 以上几点被称为“iRoPE”结构(“i”代表“interleaved”)。
 - Llama 4 Scout在RoPE层的Attention的QK状态处额外使用了L2正则化，在RoPE编码后应用。
 - Llama 4 Scout是完整的MOE架构。而Llama 4 Maverick的每层的FFN部分交替(“Alternate”)使用Dense或MOE网络，有一半的层是专家层。
 - MOE层使用128个可选专家和1个共享专家，每个token被发送到1个可选专家和唯一的一个共享专家。

DeepSeek架构创新 - MLA



功能流程

1. 输入隐藏状态 h_t 。
2. 通过 Multi-Head Attention 对Q, K, V 进行计算。
3. 融合旋转位置编码。
4. 输出隐藏状态 u_t 。

1. K和V的计算与缓存

- 压缩Key和Value的潜在向量。
- 通过降维矩阵生成Key和Value。
- 加入旋转位置编码生成最终的Key。
- 将压缩K k_t^L 和旋转位置嵌入的 Kk_t^R 进行拼接，作为最终K表示。

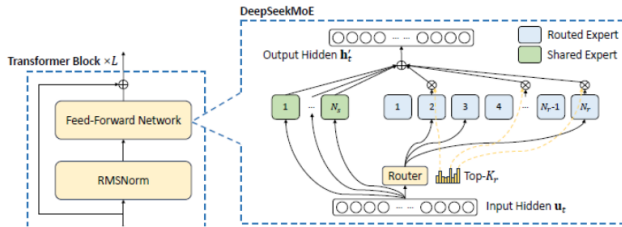
2. Q的计算与低秩压缩

- 利用降维矩阵，压缩Q，得到Q的潜在表示。
- 通过上投影矩阵生成压缩后的Q。
- 通过旋转位置嵌入生成包含位置信息的Q。
- 将压缩Q和位置增强的Q拼接，得到最终的Q。

3.多头注意力机制的最终输出

- 注意力权重计算。
- 注意力输出。
- 最终输出隐藏向量。

DeepSeek架构创新 - MoE



Router (路由器)

- 路由器决定输入 u_t 的每一部分将路由到哪些专家。
- 每个词元最多路由到 k_y 个专家节点，减少通信开销。

Experts (专家模型)

- 包括多个专家，分为两类：Routed Expert Shared Expert。

Top-K 策略

- 在 N_e 个专家中选出响应最优的 k_y 个专家。
- 限制跨节点通信范围，优化效率。

基本架构

FFNs

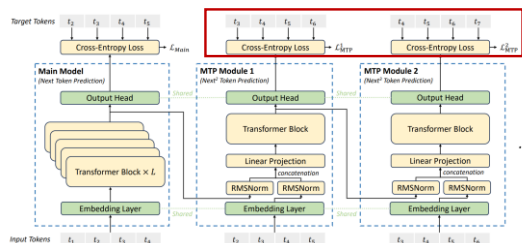
DeepSeekMoE的架构在每个FFN（前馈神经网络）阶段对输入 u_t 进行加权组合，公式如下：

$$h'_t = u_t + \sum_{i=1}^{N_e} \text{FFN}_i^{(s)}(u_t) + \sum_{i=1}^{N_e} g_{i,t} \text{FFN}_i^{(r)}(u_t)$$

Gating权重的计算

- Affinity Score(Token 到专家的亲和性分数)。
- Top-Kr 路由策略。
- 归一化权重。

DeepSeek架构创新 - MTP



MTP的优势

提高训练信号密度：在每个时间步增加多个预测目标，让模型从更多训练信号中学习。

改进未来预测能力：帮助模型提前规划表示，提升生成结果的准确性。

优化数据效率：减少对数据规模的依赖，提升在相同数据集上的训练效果。

推测性解码支持：为生成任务提供了新的优化方向，通过序列化预测降低延迟。

①模型结构

- 每个MTP模块共享嵌入层和输出头；
- 每个MTP模块独占一个Transformer Block和一个投影矩阵；
- 多个MTP模块串联保持完整的因果关系链。

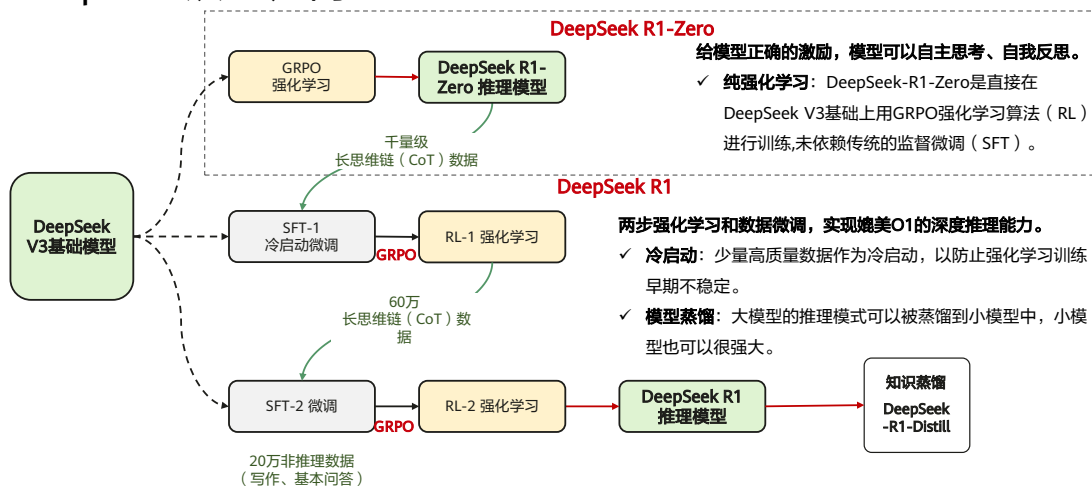
②训练策略

- 每个MTP模块输出预测token的概率分布；
- 每个MTP模块计算对应的交叉熵损失函数；
- 多个MTP模块的损失函数加权平均得到最终训练目标。

③关键作用

- 提升每批训练数据的使用效率，强化训练信号；
- 优化模型表达能力，提升next-token的预测效果；
- 可参考投机采样改造MTP模块，加速推理效率。

DeepSeek从V3走向R1



NLP任务处理的模型选择思路

- 随着Transformer技术的发展，语言模型越来越大，而它们的模型结构逐步收敛，基本以Decoder-Only为主。并且大模型的训练成本较高，如果修改模型结构，需要继续训练模型，训练成本和训练周期都是一般开发者难以承受的，因此一般选择现有开源模型进行微调。
- 不同的大语言模型它们的基础能力基本相似，而在特定领域内各有所长。因此可以查看不同模型在评测分值，根据特长选择使用何种模型。例如Claude擅长写代码等。

对于同一款模型，它也有不同类型的“子模型”，这些“子模型”是在基础模型上微调而来，开发者可以参考专注于不同领域的“子模型”。

模型	开发者	关键特性	应用场景
GPT-4.5	OpenAI	多模态（文本、图像、音频），高效计算，实时交互	对话、实时交互、内容生成
DeepSeek-R1	DeepSeek	671B参数，推理和代码生成强，成本低（比OpenAI-o1便宜30倍）	数学、编程、医疗影像分析
DeepSeek-Prover-V2	DeepSeek	专注于数学证明类任务，推理能力强	数学研究、逻辑推理
Qwen-3	Alibaba	开源，1.6B-72B参数，32K-128K上下文窗口，数学和编程能力强	企业应用、游戏、代码生成
Grok-3	xAI	“Think”和“Big Brain”模式，实时数据处理，图像生成	新闻分析、编程辅助、动态客服
Llama-3.3	Meta AI	多模态，128K上下文窗口，开源	客服、教育、内容生成
Claude-3.7-Sonnet	Anthropic	推理模式，编码和前端开发强	软件工程、客服、知识管理
Mistral-Small-3.1-24B	Mistral AI	24B参数，延迟低（150令牌/秒），开源	虚拟助手、实时数据处理
Gemini-2.5	Google	1M上下文窗口，多模态，自食事实	复杂问题解决、大型文档处理



- <https://huggingface.co/blog/VirtualOasis/llm-evaluation-and-choose>

参数规模对模型选型的影响

- 对于同一个模型，它又有不同参数的类别，可以在不同训练、推理设备上选择合适的参数量大小的模型。例如Qwen3不同大小的模型如下图所示：

Models	Layers	Heads (Q / KV)	Tie Embedding	Context Length
Qwen3-0.6B	28	16 / 8	Yes	32K
Qwen3-1.7B	28	16 / 8	Yes	32K
Qwen3-4B	36	32 / 8	Yes	32K
Qwen3-8B	36	32 / 8	No	128K
Qwen3-14B	40	40 / 8	No	128K
Qwen3-32B	64	64 / 8	No	128K

- 虽然大型模型通常在复杂任务上表现更好，但高质量领域特定数据训练的小型模型可能优于通用数据训练的大型模型。技术人员应评估模型大小、计算成本与应用需求间的平衡。
 - 小参数模型（如Qwen-3-1.7B）：训练和部署成本低，适合小型企业，但复杂任务表现弱。
 - 中参数模型（如Llama-3.3 70B）：平衡性能和成本，适合大多数场景。
 - 大参数模型（如DeepSeek-R1 671B）：性能强，成本高，适合大企业或高性能需求。



- 效率：**小参数模型推理速度快，适合实时应用；大参数模型处理复杂任务高效，但资源消耗大。

目录

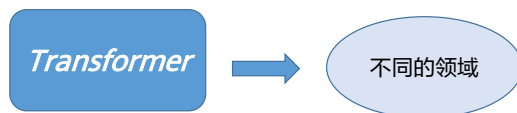
1. AI模型选型及设计分析

- 基于卷积神经网络的视觉模型
- 基于Transformer的大语言模型
- 基于Transformer的视觉模型
- 基于Transformer的多模态大模型

2. 模型选型与设计总结

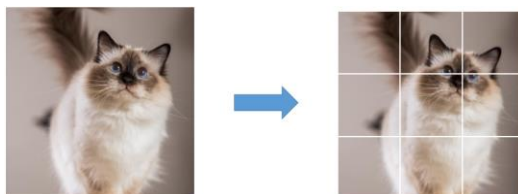
Transformer应用的思考

- Transformer在最初设计时是为自然语言处理而生的，在文本类领域应用广泛，那么，它是否可以应用在视觉领域呢？



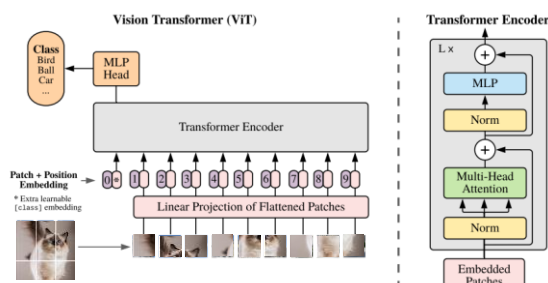
Vision Transformer

- 为了实现用Transformer模型处理图像的目标，Vision Transformer（ViT）采用了一种全新的思路。与传统的卷积神经网络不同，ViT完全抛弃了卷积结构，不再依赖局部滑动窗口提取特征的方式，而是借助Transformer强大的全局建模能力，直接对整张图像进行处理。这种方法的关键在于：将图像拆分成一个个小块，再将这些图像块视作“词语”，送入Transformer中进行处理，从而使模型能够理解图像中不同区域之间的关系。



ViT结构

- ViT的第一步是将整幅图像划分为许多大小相同的小块。
- 每一个小块会被“展平”，也就是说，把原本二维的像素信息转变为一维的向量，并通过线性变换转换成固定长度的表示向量。在这些图像块向量被送入Transformer之前，ViT还会额外添加一个特殊的标记，称为“Class Token”。
- ViT利用自注意力机制（Self-Attention）来理解图像各区域之间的关联。
- 经过多层Transformer的编码之后，所有图像块的信息都会在Class Token中逐渐汇聚和融合。最终，ViT会依据这个Class Token的输出，完成整张图像的分类任务。



- 第一步：可以将这个过程类比为把一张大图剪成许多拼图碎片。比如，一张 224×224 像素的图像被划分成 16×16 像素的小块，那么就得到了 14×14 个图像块，也就是196个小块。
- 第二步：这些向量在结构上就像是自然语言中单词的“词向量”，是Transformer所能处理的基本输入单位。类别标签可以把它想象成Transformer中的“观察者”，它与所有图像块进行信息交互，最后综合得到一份关于整张图像的总结。

ViT是否可以替代CNN?

- ViT通过自注意力机制直接建模图像中所有块之间的关系，能够捕捉图像的全局依赖，突破了传统CNN局部感受野的限制。适合处理复杂场景（如医学图像、卫星图像），能够理解图像中远距离区域的关联性。
- 但是，ViT模型本身参数量大，因此它在小数据集上表现不如CNN。同时它对计算资源的需求较大，因此，ViT并没有一统CV界。
- 不过，在多模态大模型中，ViT是一个不可或缺的部分。接下来我们看一下多模态领域模型的主流架构。

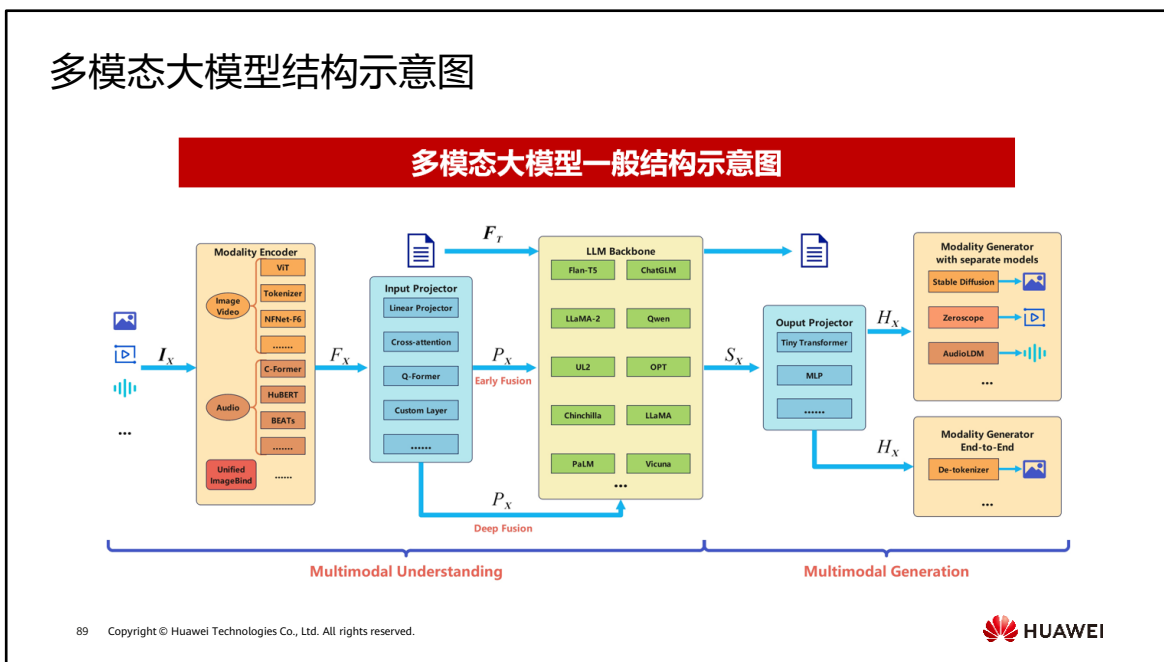
目录

1. AI模型选型及设计分析

- 基于卷积神经网络的视觉模型
- 基于Transformer的大语言模型
- 基于Transformer的视觉模型
- 基于Transformer的多模态大模型

2. 模型选型与设计总结

多模态大模型结构示意图



- **Modality Encoder:** 不同模态的输入进行编码获得相应的特征表示，如图像模态经常使用Open AI的CLIP ViT-L编码器；
- **Input Projector:** 将其他模态的编码表示与语言模态对齐，适配LLM的输入；
- **LLM Backbone:** 通常采用已训练好的LLM模型，处理多模态输入token，如开源的Llama系列；
- **Output Projector:** 将LLM输出token反映射回多模态生成部件可处理的特征表示；
- **Modality Generator:** 生成具体的模态输出，如图片、视频、音频等，可通过调用单独的多模态生成模型或端到端生成多模态输出；对于多模态信息融合的方式通常分为深度融合和早期融合两种，其中：
• **Deep Fusion:** 多模态信息在LLM内部融合，通常会修改LLM内部结构；
• **Early Fusion:** 多模态信息在LLM输入端融合而无需对LLM结构进行大幅修改

多模态大模型

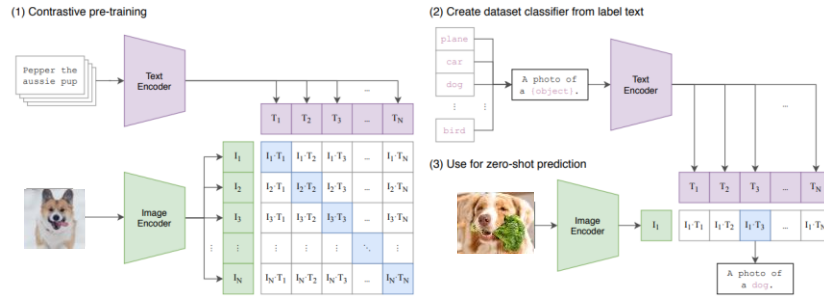
- 多模态AI是指能够处理和理解多种类型信息的人工智能，如文本、图像、音频、视频等。这种AI不仅能够处理单一数据类型的任务，而且可以在不同数据类型间建立联系和融合，从而实现一个综合、全面的理解多模态。



- 以前，我们的机器学习模型一般都只处理一种类型的数据，比如只处理文本（翻译、语言建模）、图像（对象检测、图像分类）或音频（语音识别）。
- 但是，想想我们人类的大脑。我们不仅仅读写文字，还可以看图、看视频，听音乐，还能够辨识各种不同的声音。所以，为了让AI更接近真实世界，处理多种类型的数据显得非常重要。
- 很多应用场景如果没有多模态技术支持几乎难以实现，尤其是在需要处理多种类型数据的领域，例如医疗、机器人、电商、零售、游戏等。
- 综合使用多种数据模式的信息，可以显著提升模型的性能。比如，一个能够同时处理文本和图片的模型，其效果应该比只能处理文本或只能处理图片的模型更出色。
- 多模态技术使得我们与系统交互更加灵活多样，可以根据自己的喜好或场合选择交互方式，比如键盘输入、语音对话或者拍照识别。

Clip

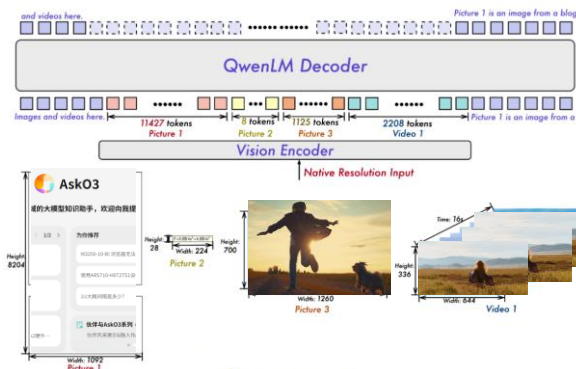
- CLIP (Contrastive Language-Image Pre-training) 是OpenAI的第一篇多模态预训练的算法，它延续了GPT系列“大力出奇迹”的传统。
- 模型是一个基于图像和文本并行的多模态模型，然后通过两个分支的特征向量的相似度计算来构建训练目标。为了训练这个模型，OpenAI采集了超过4亿的图像-文本对。CLIP在诸多多模态任务上取得了非常好的效果，例如图像检索，地理定位，视频动作识别等等，而且在很多任务上仅仅通过无监督学习就可以得到和主流的有监督算法接近的效果。



- <https://arxiv.org/pdf/2103.00020>

Qwen2-VL

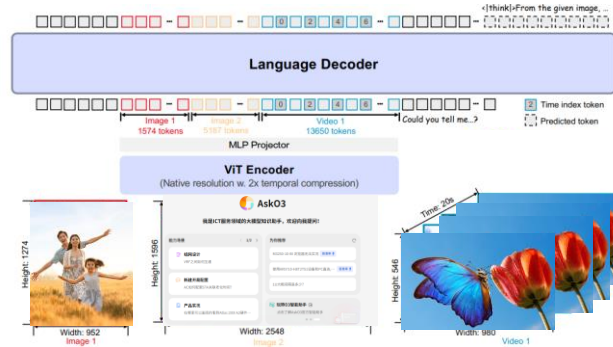
- Qwen2-VL采用了基于早期融合的多模态信息融合方式，图像或视频通过ViT视觉编码器转换为视觉token，然后与文本token拼接后输入到QwenLM骨干，输出为文本token，为主流的多模态大模型架构。



- <https://arxiv.org/pdf/2308.12966>

GLM4.5-V

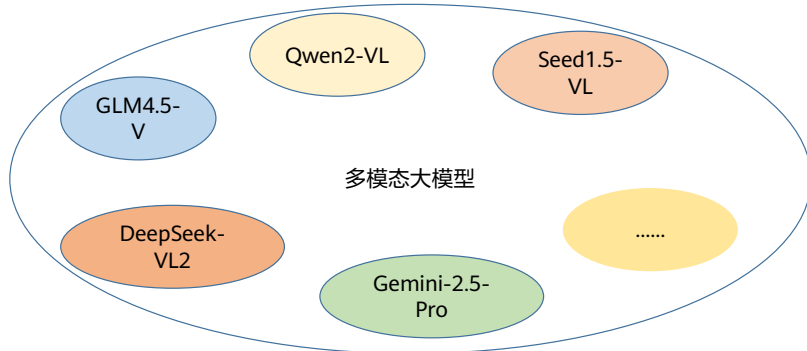
- GLM-4.5V 是智谱新一代基于 MOE 架构的视觉推理模型，以106B的总参数量和12B激活参数量，在各类基准测试中达到全球同级别开源多模态模型 SOTA，涵盖图像、视频、文档理解及 GUI 任务等常见任务。



- <https://arxiv.org/pdf/2507.01006>

多模态大模型的

- 多模态大模型架构相对较复杂，在模型设计方面可以借鉴主流算法结构。而多模态大模型训练所需算力较大，重新设计并训练模型困难较大。因此模型开发人员可以根据多模态大模型特性选择性使用，对其进行微调，或者直接部署使用。



- 目前多模态大模型闭源较多，可以使用开源多模态大模型，参照其官方解释，选择对应能力方向的模型。

目录

1. AI模型选型及设计分析
2. 模型选型与设计总结

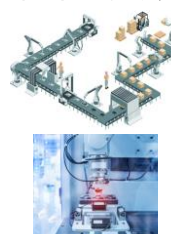
确定使用场景

- 在AI应用落地过程中，第一个需要面对的问题就是：当前AI模型是应用在什么领域，需要解决什么问题？

如果我们想做园区安防场景，那么通过摄像头拍摄到的图像判断物体是必不可少的，因此目标检测算法是当前场景的首选。



如果是生产线进行产品的分拣，那么图像分类的算法是当前的首选。但如果是多产品混合分拣或者是缺陷检测，那么极大可能需要目标检测算法。



如果是对评语进行情感分析或者翻译，那么文本分类的模型-RNN系列模型是可选的，同样Transformer架构也是一个不错的选择。



如果应用在智能问答系统或者更复杂的人机交互场景中，那么大语言模型是首先，并且如果是图文共存的场景，那么多模态大模型是不二选择。

分析数据复杂度

- 当我们确定使用场景以后，需要判断该场景下数据的复杂程度。因为对于复杂数据，需要参数量更大的模型来处理。

- 例如：



对于这种背景复杂，并且目标较小的检测场景来说，低版本YOLO并不适合，虽然它的参数量小，检测速度快，但它对小目标检测能力很差，这时反而是Faster R-CNN或者高版本的YOLO等参数量大的模型是更优的选择。

在选择完模型类型后，可以设计具体的网络结构，例如根据数据复杂程度选择卷积层数，因为更深的网络对特征提取效果更好，但是也会伴随着参数量的增加。

如果检测场景中包含大小不等目标，可以使用不同大小的特征图，即可使用网络不同位置提取的特征图进行判断。以上都是模型设计的重要思路。

而在自然语言领域，若文本类型众多，若需要一个模型实现多种任务，例如文本总结、代码生成、翻译等，那么需要选取参数量较大的模型，这时候可以使用几百甚至上千亿参数的大模型。

但是如果使用场景单一，例如只需要做某国语言的翻译，那么可以只选取参数量较小的模型。

对于不同任务，也可设计不同参数的模型，根据需求选择模型中块的个数。根据任务增加或减少参数量。



训练设备和部署设备

- 训练和部署设备与模型参数是息息相关的，参数量大的模型对芯片算力有更高的要求，数据复杂度与模型参数又互为影响因素。例如数据复杂，需要的模型参数相对较高，但是设备算力不足，这时候只能有所取舍。

例如对于图像分类任务，如果算力足够，可以使用ResNet50甚至更深的网络，如果算力不足，那么GoogLeNet或者MobileNet是不错的选择。但如果数据复杂，端侧设备算力还有限，这时需要修改整体方案，可以在边侧或者云端进行计算，端侧只负责数据采集与信号输出。



or



对于大模型，更依靠强大的算力作为支撑。低算力只能选择参数量小的模型，如果部署在强大的服务器或者集群上，那么可以使用参数量更大效果更好的MoE模型。同时在选择MoE模型时，还可以设计共享专家和私有专家的个数。

而对于多模态任务，大算力是必不可少的，因此端侧设备算力不足，可以考虑端云协同的方案。

网络结构设计的一些基本要素

- 网络结构是模型的“骨架”，需通过层与模块的组合，让模型能有效捕捉数据规律。设计核心逻辑：“用最小的复杂度实现任务所需的特征提取能力”。

控制网络深度与宽度：
平衡拟合能力与效率

深度：深度不足会导致“欠拟合”，过深会导致“梯度消失 / 爆炸”和计算量激增。

在设计网络结构时，可以用残差连接（ResNet）、跳跃连接（DenseNet）让梯度直接回传；用归一化层（BN/LN）稳定训练。

宽度：指的是每层神经元/通道数，宽度不足会限制特征表达能力，过宽会增加冗余计算。

浅层（靠近输入）宽度小（提取基础特征，如边缘），深层宽度大（提取复杂特征，如物体部件）。例如：ResNet-50 的通道数从 $64 \rightarrow 256 \rightarrow 512 \rightarrow 1024 \rightarrow 2048$ 逐步增加。

加入正则化与适配层：
提升泛化与兼容性

正则化的加入可以减少模型过拟合，例如使用Dropout、L1正则、L2正则等。

激活函数

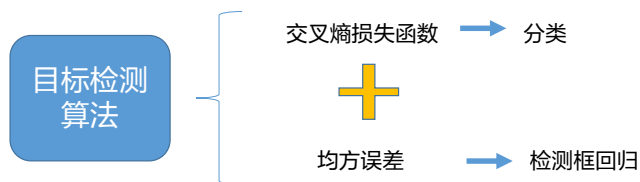
- 如何选择激活函数是模型设计时一大重点，它影响着模型的训练效果。
- 在CNN和MLP的网络中，ReLU及其变体的激活函数是首选，因为它提供了非线性能力，同时计算简单，还能缓解梯度消失或爆炸。
- 在RNN/Transformer结构中，优先使用平滑激活函数，这样可以避免输出突变。例如GELU，而在更深层的网络结构中，Swish更适合。
- 在输出层，二分类使用Sigmoid，对分类更适合Softmax，而回归任务可以使用Tanh或者不适用激活函数。
- 需要注意：在深层次网络结构中，尽可能避免使用Sigmoid和Tanh。你知道吗？

- <https://3ms.huawei.com/km/static/image/detail.html?fid=56755>

损失函数

- 在分类任务中，最常见的是使用交叉熵损失函数，而回归任务常用均方误差损失函数。但是，在更多的场景中，损失函数是根据任务来特定设置的。

- 例如：



- 而大模型基本上是自回归语言模型，但由于它任务（输出）的复杂性，多模态的交叉结合，其使用的损失函数更为多样，具体的分析可以参考对应论文。

总结

- 模型选型是根据任务类型选择合适的网络结构，因为不同的网络结构可以处理不同的数据，完成对应的任务。这些结构是算法工程师在不同领域探索的结晶，但是这些结构并不是万能的。
- 因为现实场景中，任务类型各有不同。虽然有些近似任务，但是不同场景下数据会有差异，这些差异导致经典算法在当下效果不好，因此在同类问题中，也需要根据不同的场景去设计开发新的算法结构，而这种新的结构大多是在经典算法上的改进。
- 对于模型开发，最重要的是需要理解场景和数据，掌握不同算法的数学特性，结合数据进一步分析如何设计算法，并在不断的实验中找寻更合理的结构。

思考题

1. （单选题）以下哪一项是在选择和设计模型时不需要考虑的因素？
- A. 应用场景的类型
 - B. 数据复杂度
 - C. 部署算力大小
 - D. AI产品使用者职业

• D

本章总结

- 本章主要从视觉领域和语言领域分析了AI模型的结构发展，并讲解了主流算法的结构，损失函数及激活函数等。结合不同的场景，例如目标检测场景，讲解了主流算法及其特性，并分析在特定场景下选择和设计模型的思想。最后，总结了模型选型和设计的主要思路。

学习推荐

- 昇腾社区
 - <https://www.hiascend.com/>
- MindSpore官网
 - <https://www.mindspore.cn/>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

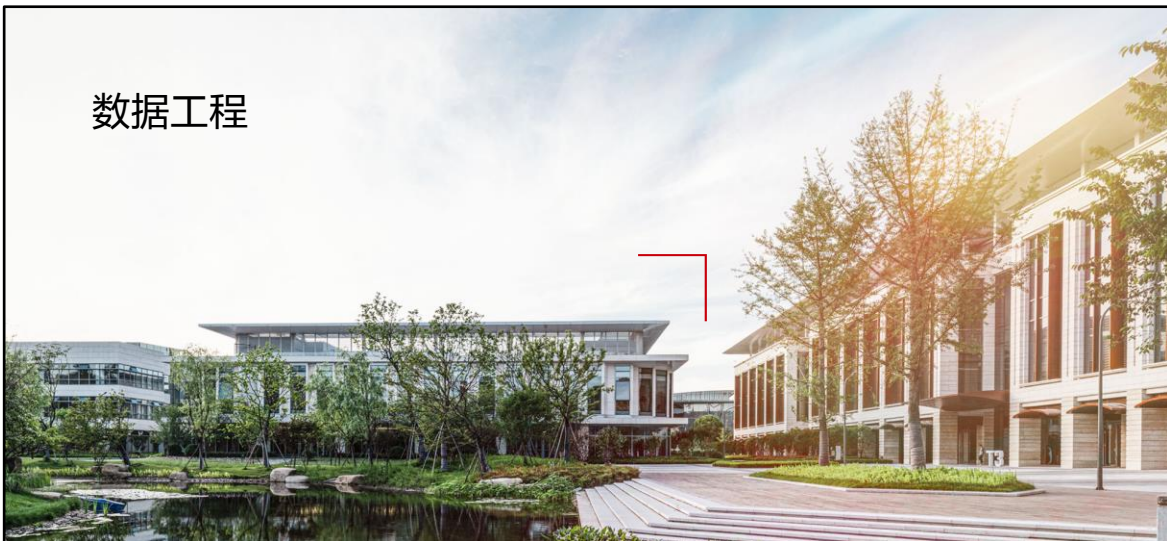
Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



数据工程



前言

- 本章主要介绍了针对模型训练的数据工程部分，课程首先会介绍数据工程的全景图和功能，接下来会介绍数据治理的部分，帮助大家了解数据治理的关键目标和数据集构建的流程，最后会介绍数据工程实战，包含图像、视频、文本数据的预处理以及Llama、DeepSeek的数据构建流程。

目标

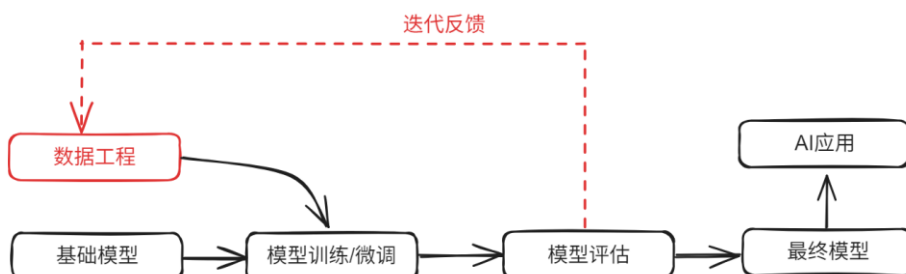
- 学完本课程后，您将能够：
 - 掌握数据工程的概念及包含内容；
 - 掌握数据治理的关键目标；
 - 掌握数据集构建的关键流程；
 - 掌握通用数据处理方法；
 - 掌握LLM数据构建方法。

目录

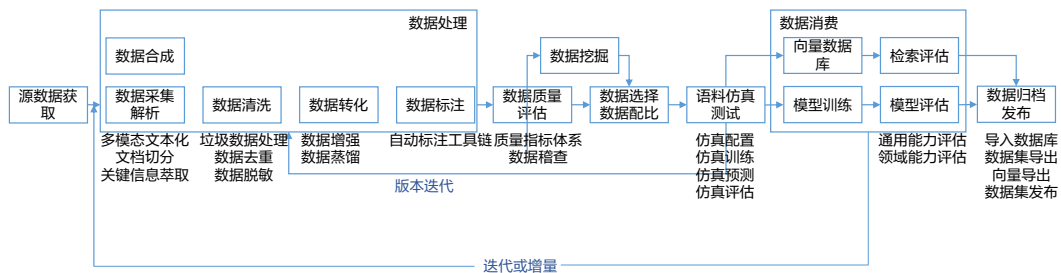
1. 数据工程概述
2. 数据治理
3. 数据工程实践

数据工程在模型开发中的位置

- 以下是简化后的模型开发流程，其中基础模型的构建、模型微调都需要符合格式要求的高质量数据，而在模型进入生产环境后，也需要根据使用的反馈数据进行迭代更新。



数据工程全景图



数据工程功能集合

- 数据工程功能集合：数据集管理、工具管理、数据处理及挖掘、数据标注、数据质量、数据集仿真

数据集管理	工具集管理	数据处理及挖掘	数据标注	数据质量	数据集仿真
• 数据集导入 • 数据集发布 • 数据集导出 • 数据集可视化 • 数据集管理&血缘回溯	• 数据解析工具链 • 数据合成/增强算法 • 数据标注工具链 • 数据配比选择工具链 • 数据质量评估工具链	• 数据处理 ➢ 数据解析 ➢ 数据清洗 ➢ 数据去重 ➢ 数据脱敏 ➢ 数据滤毒 ➢ 数据增强 • 数据挖掘 ➢ Badcase识别 ➢ 语义检索	• 标注批次管理 • 标注工作流 • 标注人员管理 • 标注任务分配 • 标注审核	• 数据质量指标体系 • 数据稽核 • 数据质量看板 • 数据质量报告	• A/B测试 ➢ 仿真测试配置 ➢ 仿真测试训练 ➢ 仿真测试预测 • 数据仿真 ➢ 数据配比仿真 ➢ 数据质量仿真 ➢ 数据处理仿真 ➢ 数据仿真评估

数据工程的建设思路：工具是基础、数据内容建设是核心



- 数据管理
 - ①数据收集与管理工具：选择和使用适合的数据收集、存储、处理的工具和技术，如版本控制系统、日志管理工具、数据仓库等。
 - ②数据处理与清洗技术：开发和使用数据清洗、转换和预处理技术，以确保数据质量，提升模型的训练效果。
 - ③数据集成与管道建设：设计和实现数据管道，自动化地收集、处理、整合数据，确保数据流动流畅，支持大规模模型的训练和更新。
 - ④安全与合规技术：应用数据安全和合规技术，确保数据在处理和使用过程中符合行业标准和法规。
- 数据内容
 - ①语料库设计：由软件工程领域的专家设计并构建高质量的语料库，涵盖代码、文档、日志等多种类型的数据，以确保模型具备广泛而深入的领域知识。
 - ②数据分类与标注：根据软件工程的需求对数据进行分类和标注，确保模型能准确理解和处理不同类型的数据，如源代码、测试用例、设计文档等。
 - ③领域知识的注入：结合软件工程领域专家的知识，优化和调整数据的选择和处理方式，确保模型理解专业术语和复杂的领域概念。
 - ④反馈与迭代：通过不断的反馈和迭代，改进数据内容，更新语料库，使模型能够持续学习和进步。

目录

1. 数据工程概述
- 2. 数据治理**
3. 数据工程实践

什么是数据治理？

企业管理者

数据治理是企业发展战略的重要组成部分，是企业数字化转型的基石，各部门要按照企业战略要求落实各项工作，以实现企业数字驱动的转型目标，赢得未来的市场机遇。

业务人员

数据治理是公司IT部门的事，我们只是配合。希望数据治理能将数据开放出来，让我们理解数据，使用的时候能够方便的获取有质量保证的数据。

算法专家

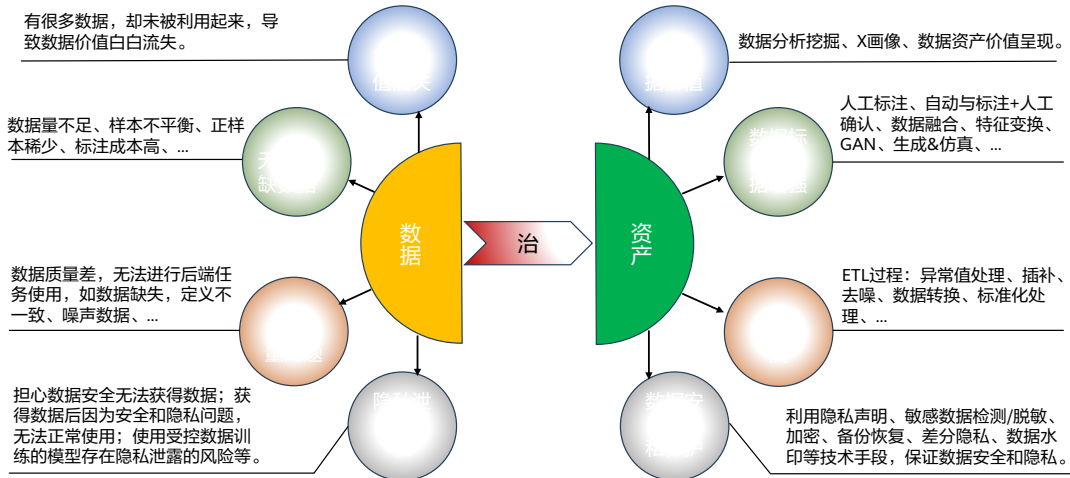
数据治理是为了满足算法研究的需要，对数据质量问题的清洗、标注和数据增强活动，尽可能扩大数据丰富度的同时，保证数据能够真实反应业务的本质，能够方便高效的按需获取数据。

数据工程师

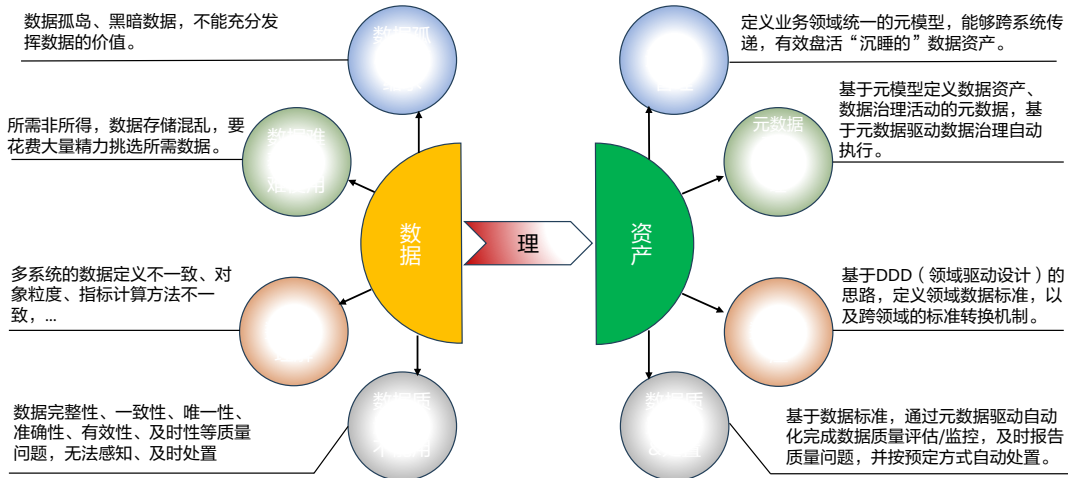
数据治理是构建统一的数据模型，定义数据标准和质量指标，通过ETL、标注、融合等手段完成原系统数据到标准化数据的处理过程，保证被管理的数据准确、可用、可溯源。

数据治理是提升数据的质量、安全性、可用性和易用性所执行的所有操作，包括必须遵循的政策法规、必须采取的数据管理行动，以及在整个数据生命周期活动中为其提供支持的技术。数据治理的目标是维护安全且易于访问的高质量数据，以获得更深入的业务洞察。

数据治理“治”什么？



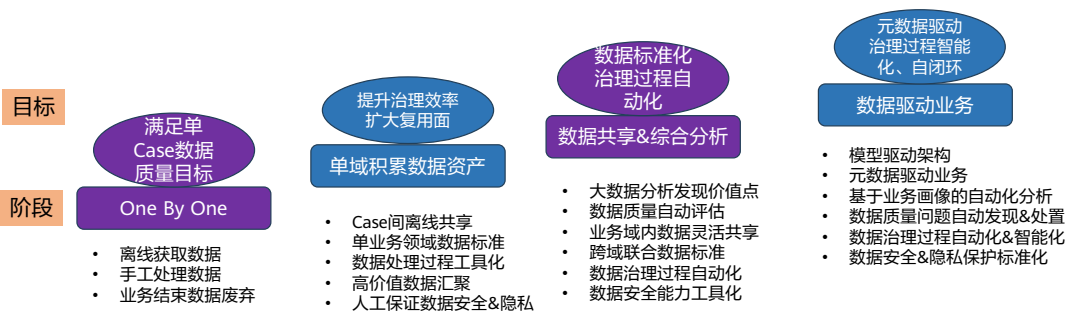
数据治理“理”什么？



- DDD是领域驱动设计

数据治理目标是什么？

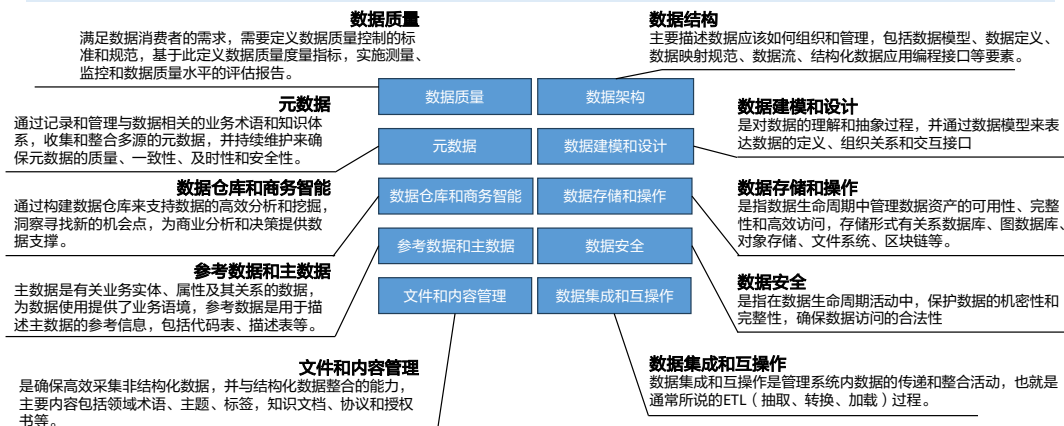
目标：维护安全且易于访问的高质量数据，以获得更深入的业务洞察，在不同的业务阶段，数据治理的目标不同，确立正确的数据治理目标很关键。



DAMA数据管理框架

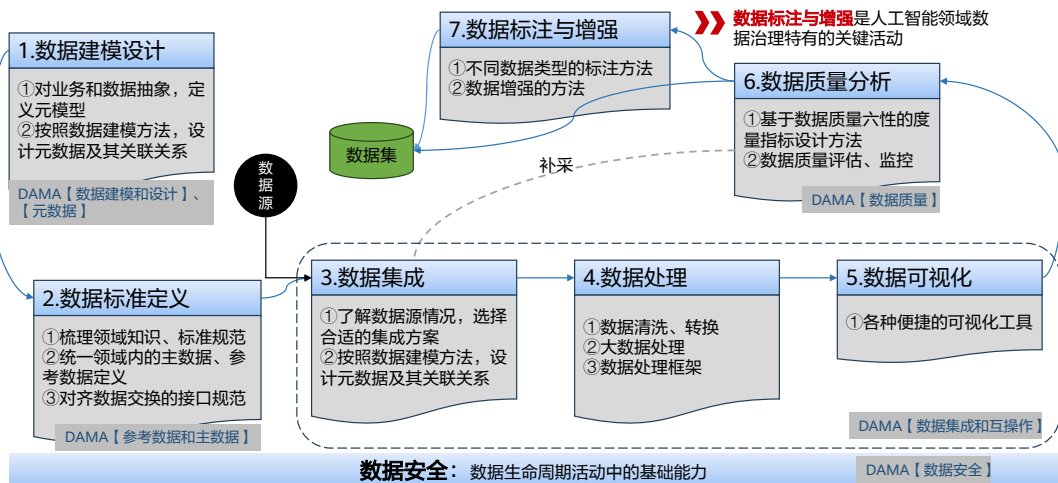
数据治理

数据治理使组织能够将数据作为资产被恰当的管理，提供治理原则、制度、流程、整体框架、管理指标和监督数据资产管理，并指导数据管理过程中各层级的活动。

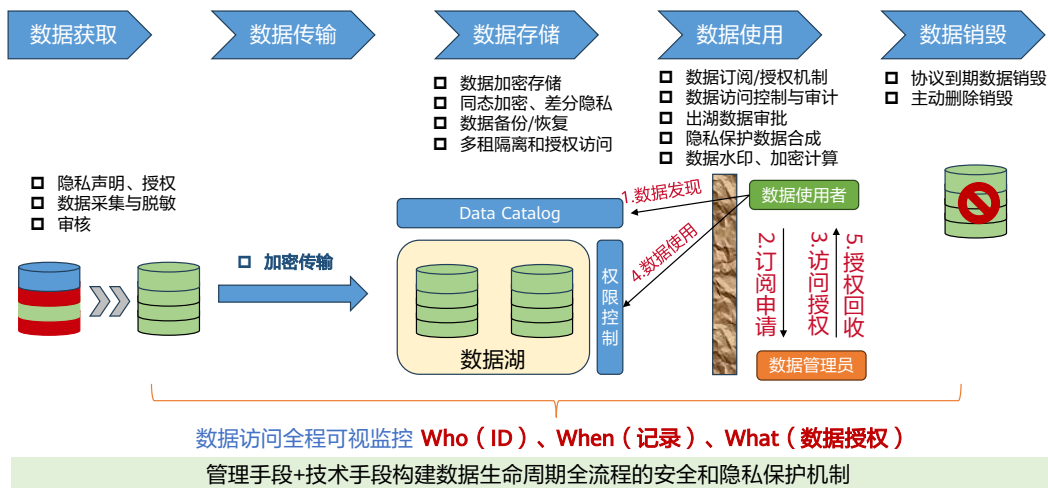


- DAMA是国际数据管理协会

数据集构建流程中的关键活动



数据生命周期的安全和隐私保护



数据治理活动中的数据安全&隐私保护方法

敏感数据检测/脱敏方案：

- **检测：**基于模式匹配的方式，对文本数据、结构化数据中的敏感信息进行自动检测。
- **脱敏：**基于用户定义好的脱敏方法，可手工对数据进行批量脱敏，也可在敏感数据检测环节进行脱敏处理。

优势：

- 提供自动化检测/脱敏机制，适应业务运行态场景；离线批量脱敏适用于批量数据一次性脱敏后的跨租户共享。
- 基于模式匹配的方式检测准确率高，有效避免了隐私泄露问题。

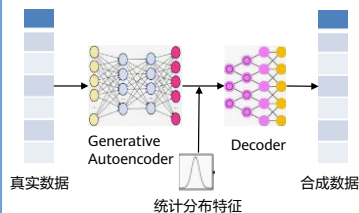
分类	方法	效果
普通脱敏	掩码	1388888 138****
	加噪	1388888 13888856
	替换	1388888 1381111
	哈希	192.168.111.222 24 1.164.81.202
	截断	1388888 1388
高阶脱敏	针对数值型进行泛化的K匿名算法	213 [0,1055]
	加解密	105333 jn8oa6bxbp

隐私保护数据合成方案：

- 利用深度生成模型学习真实数据的统计特征，生成高可用的合成数据。
- 利用差分隐私技术训练生成模型，防止模型记忆训练数据信息。

优势：

- 半合成、全合成，提供不同的隐私保护方案。
- 合成数据具备原始数据的统计学特征，可用于AI模型训练使用
- 合成数据不属于个人数据，避免了数据共享中的隐私泄露问题。

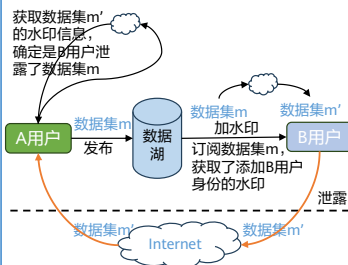


数据水印方案：

- 基于华为自研的数据水印技术，针对结构化设计了数据库水印方案，能够对数值型和字符串型数据加/解水印。水印以用户ID为标识，便于数据泄露后的追责。

优势：

- 水印的隐蔽性强，不易被察觉
- 对原始数据做微小无规律的修改，最大程度保持加水印后数据质量与原始数据保持一致；
- 鲁棒性强，一般常见攻击类型难以攻破。



DataArts：数据治理生产线，一站式数智融合开发与治理

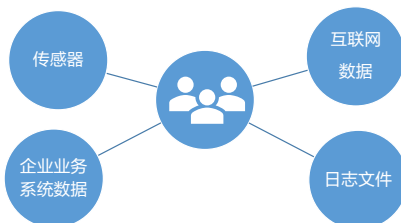


目录

1. 数据工程概述
2. 数据治理
- 3. 数据工程实践**
 - 通用数据处理流程
 - LLM数据处理流程

数据采集

- 模型数据采集分为实时与历史两类数据源。
- 实时数据源：传感器将物理量转为数字信号。工业中温度传感器帮助模型预警设备故障，实时性强但有噪声；摄像头把图像转视频流，智能交通中供模型识别交通信息，数据量大、处理复杂；麦克风将声音转音频流，语音助手借此识别指令，音频有时序性、参数敏感。
- 历史数据源：数据库（如MySQL、MongoDB）存大量历史数据，金融中用于预测股价走势，结构化高但更新或滞后；文件系统存文本、图像等文件，医疗中辅助疾病诊断，存储灵活但查询管理难；此外还有互联网上大量的数据等。

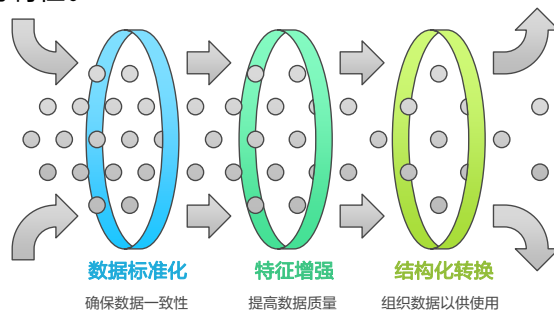


• 实时数据源

- **传感器数据：**传感器能将物理量（如温度、压力、光照强度等）转换为电信号，进而转换为数字信号。在工业生产中，温度传感器实时采集设备温度数据，用于模型推理判断设备是否过热，提前预警故障。
- **特点：**实时性强，能及时反映物理环境的变化，但可能受环境干扰影响，数据存在一定噪声。
- **摄像头与视频流：**摄像头将光学图像转换为电信号，再编码为数字视频流。在智能交通系统中，摄像头实时采集交通路口的视频数据，用于模型推理识别车辆类型、车牌号码、交通流量等信息，辅助交通管理。
- **特点：**数据量大，包含丰富的视觉信息，但处理复杂度高，对计算资源要求较高。
- **麦克风与音频流：**麦克风将声音振动转换为电信号，再转换为数字音频流。在语音助手应用中，麦克风实时采集用户语音，用于模型推理识别语音指令，实现人机交互。
- **特点：**音频数据具有时序性，对采样率和位深度等参数敏感，不同环境下的音频质量差异较大。

数据预处理

- 在数据采集完成后至模型推理阶段之间，需针对不同数据格式（如视频、图像、文本）实施多模态预处理（Multimodal Preprocessing），其核心目标是通过标准化、特征增强及结构化转换，将原始数据适配为模型可高效处理的输入形式，同时最大化保留或提取任务相关的信息特征。



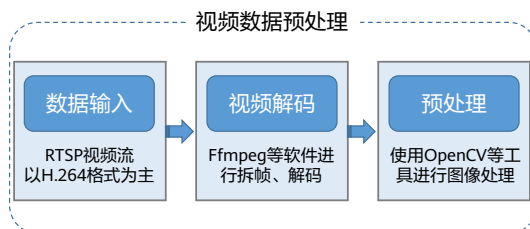
图片数据预处理

- 图片的预处理是为了确保模型训练的时候可以提取到通用特征，因此需要做数据增强等操作，同时将尺寸统一，用于固定shape的模型。

操作名称	说明
数据增强	随机裁剪、翻转、颜色抖动
尺寸调整	随机缩放（RandomResize）
归一化	使用训练集的统计量
填充与裁剪	随机填充、随机裁剪
批量处理	动态批量（Dynamic Batch）

视频数据预处理

- 视频数据预处理的目的是将原始视频数据转换为适合模型输入的格式，同时保留或增强视频中的关键信息。
- 视频数据因其时序性、高维度和动态特性，其预处理流程显著复杂于静态图像。



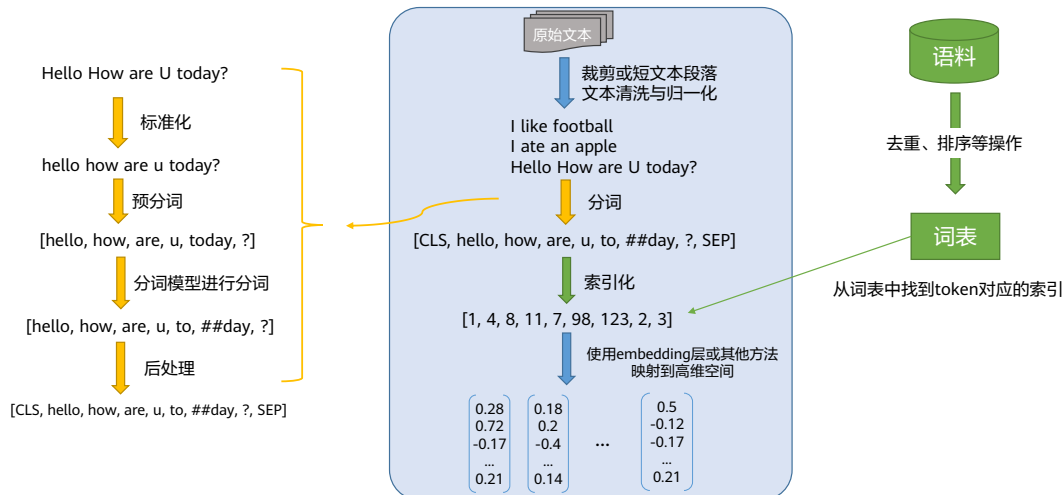
文本数据预处理

- 语言是人类特有的概念。人可以理解每个语言单词的意义，但模型无法直接的从感知中抽象出语言符号的意义。
- 在模型推理阶段，自然语言文本需转换为模型可以理解的数值表征，比如整数、浮点数或者向量。
- 那么如何把一段输入文本转化为向量呢？



- 语言是人类特有的概念。作为一个抽象符号，人是可以理解每个语言单词的意义的，但是现在的NLP语言模型无法直接的从感知中抽象出每个语言符号的意义。为了让模型能够理解自然语言文本，需要把文本转换为模型可以理解的表征，比如整数、浮点数或者向量。

从输入文本到向量



- 以机器翻译（中译英）为例，下一段输入文本在转化为embedding之前都经历了什么，具体流程如下：
 - 建立中文词表和英文词表：此步骤会把语料库中的文字去重、然后排序之后就可以得到词汇表。每个字符在词表中有唯一序号。
 - 整理文本：把段落等裁剪成短文本。
 - 分词：将连续的文本拆分/转换成一个个独立的最小语义单元。生成的每个单元称为token。具体又包括如下几步：
 - 标准化：对文本进行必要的清理工作，例如清理特殊符号、格式转换、过滤停用词、Unicode标准化等。
 - 预分词：将输入拆分为单词。
 - 分词模型处理：把预分词器得到的单词送进分词模型进行分词，得到token序列。比如图中today变成##day是使用的子词分词法，##是一个特殊标记，表明该子词是前一个词的延续，而不是一个独立的词。
 - 后处理：添加分词器的特殊token，生成注意力掩码等。此时token仅仅是字符串，还不能直接用于模型的计算。
 - 索引化：通过词表将token转换成计算机更容易处理的数字信息。每个token都会去词表中查询，得到该token在词表中的序号，这个序号就是每个token对应的one-hot向量。

文本分词：分词器的组成部分

- 如何高效且准确地完成分词过程呢？需要依赖专门的工具分词器（Tokenizer）。
- 分词器一般包含以下几个部分：
 - **Trainer（分词训练器/算法）**：给定训练数据，用Trainer封装的分词算法，生成词表。
 - **Vocabulary（词表）**：词表是分词的标准。基本上每个语言任务都需要维护一个词表，词表可以是根据当前训练数据生成的，也可以直接加载前人预训练好的词表。
 - **Encoder（编码器）**：拥有词表后，给定语料，编码器把语料按词表拆分token，并记录token在词表中的索引，拥有该索引则可将token表示成one-hot向量入模，再做后续的embedding等优化操作。
 - **Decoder（解码器）**：给定一串索引，将索引还原成最初的语料形式。

- 正如我们刚刚看到的，中文分词是将连续的中文文本拆分成一系列有意义的词块的过程，这对于自然语言处理（NLP）任务至关重要，特别是当我们使用大型语言模型（LLMs）进行文本分析、生成或理解时。这些词块，即tokens，成为了LLMs处理和理解语言的基本单元。
- 那么，如何高效且准确地完成这一分词过程呢？这就需要依赖专门的工具——分词器（Tokenizer）。分词器是自然语言处理中的一个关键组件，它负责将原始文本自动地转换成LLMs能够理解和处理的格式。
- 分词器的设计可以基于多种方法，包括但不限于规则匹配、统计模型以及近年来兴起的深度学习技术。对于中文这样的语言，由于词语之间没有明显的分隔符（如英文中的空格），分词器的设计尤为复杂且关键。它需要考虑词汇的边界、上下文信息以及可能的歧义性。

目录

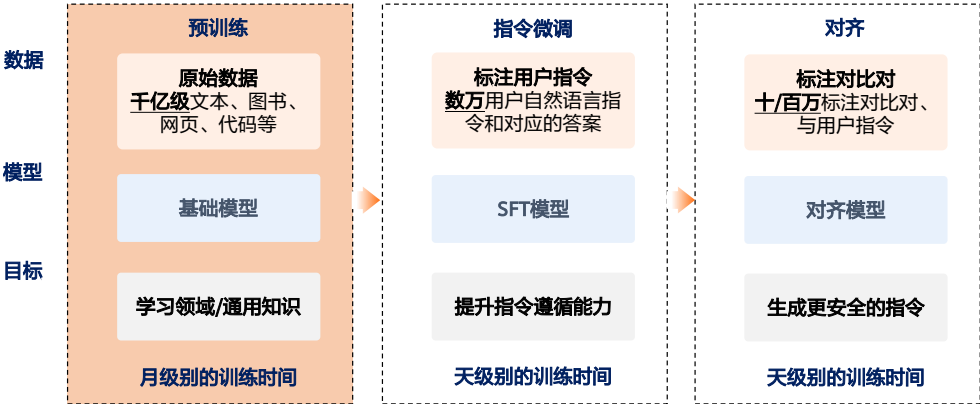
1. 数据工程概述
2. 数据治理
- 3. 数据工程实践**
 - 通用数据处理流程
 - LLM数据处理流程

针对LLM的数据工程执行方法：基于不同场景，选择合适数据工程方法

场景	RAG	Fine-Tuning	增量预训练	全量预训练
应对幻觉	✓	✗	✗	✗
外部数据源的使用	✓	✗	✗	✗
修改模型行为	✗	✓	✓	✓
庞大的训练数据	✗	✓	✓	✓
数据偏动态	✓	✓	✓	✗
个性化推荐	✓	✓	✓	✗
低资源语言或领域	✓	✓	✓	✓
模型的推理速度和效率	✗	✓	✓	✗
模型的可解释性	✗	✓	✓	✗
模型鲁棒性	✗	✓	✓	✓
合规性和数据隐私	✓	✓	✓	✗

LLM训练对数据的要求

□ 大规模无标注数据**预训练** + 指令微调 + 对齐



LLM数据准备-智能文档解析

挑战：

- PDF/PPT布局不统一，直接解析的内容没有顺序
- 影印版PDF无法直接解析内容
- 文档中插入的表格存在逻辑关系，直接解析会丢失关系

解决思路：

- 版面分析：检测页面布局并找到阅读顺序
- 文本提取（OCR）：通过启发式或OCR等技术提取文本
- 格式化：清洗并格式化每一个block
- 文本后处理：合并block并对文本进行后处理输出



LLM数据准备-数据增强合成

痛点

在多个场景发现数据分布对模型的训练效果有影响，考虑部分场景的少数类别数据进行增强，提升分布均衡

腾讯
AI
lab

定义任务角色



组合成Prompt



大模型生成

定义完成任务

微软
phi
-1

定义语法成分



定义语义成分



组合成Prompt

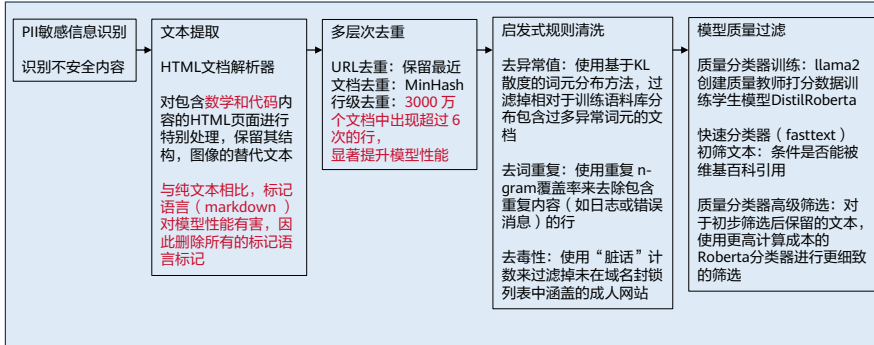


使用LLM合成代码

Meta-Llama3 预训练高质量数据（1）

预训练数据处理

网页数据（英文）



代码、数学推理数据（英文）

- 类似DeepSeek处理逻辑
- 质量分类器类似网页数据构筑方法
 - 代码有自定义的数据规则清洗方法，和文本处理流程分开构筑

多语数据

- 语言识别：fasttext 做语言分类，176种语言
- 文件级去重和行级去重
- 多语的启发式规则和模型过滤

Meta-Llama3 预训练高质量数据（2）

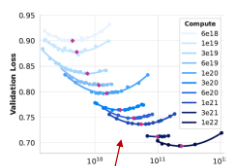
数据配比

知识分类：分类器，从网页数据、代码数据、多语数据等**数据来源**到**知识和娱乐知识分级**

扩展法则指导数据配比：小模型尝试以小见大到大模型中

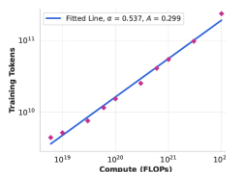
The Llama3 Herd of Models

小模型参数量40M ~ 16B
Batch size 250k ~ 4M



数据配比实验点

符合拓展法则算力和数据量越大，模型效果越好



拟合曲线找出最佳的数据量点

推出 3.8×10^{25} FLOPs 算力 (405B) $\rightarrow$ 16.55T 语料
最终配比结果

类别	配比
通用知识	0.5
数学和推理	0.25
代码	0.17
多语言	0.08

数据退火（数据顺序）

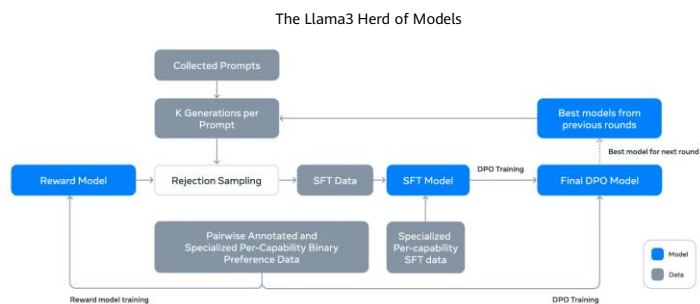
在模型训练过程中，**annealing**（退火）是指逐步减少学习率的过程，以稳定模型的学习，并优化最终的模型性能

进一步验证观点：**退火期间通常会使用更高质量的数据集，以进一步提升模型在特定任务或领域上的表现**

Llama3在预训练的最后阶段，重点使用高质量的数据。**这包括选择特定领域的优质数据，如代码、数学推理等，以增强模型在这些领域的能力**

Meta-Llama3 微调高质量数据：基于不同领域定制化生成数据

微调流程



微调数据配比

Dataset	% of examples	Avg. # turns	Avg. # tokens	Avg. # tokens	
				in context	in final response
General English	52.66%	6.3	974.0	656.7	317.1
Code	14.89%	2.7	753.3	378.8	374.5
Multilingual	3.01%	2.7	520.5	230.8	289.7
Exam-like	8.14%	2.3	297.8	124.4	173.4
Reasoning and tools	21.19%	3.1	661.6	359.8	301.9
Long context	0.11%	6.7	38,135.6	37,395.2	740.5
Total	100%	4.7	846.1	535.7	310.4

DeepSeek-V3：预训练使用14.8T token，SFT使用1.5M数据，核心为训练领域专家模型进行数据合成

- **预训练阶段：**构建14.8T高质量和多样化的tokens，数据来源和数据分布未知，介绍了3个数据处理技巧
 - **document packing：**使用best-fit packing对超长文档进行处理(下图左下角)，对比传统concat-then-split方式准确率提高2个百分点。



Len	Method	NLI		Context Following	
		MNLI	RTE	NQ-Swap	MemoTrap
		ACC	ACC	EM	ACC
2k	Concat	42.78%	55.74%	45.62%	35.58%
	Pack	44.33%*	60.28%*	51.03%*	41.56%*
8k	Concat	38.85%	54.66%	47.07%	37.61%
	Pack	40.60%*	59.72%*	50.04%*	40.28%*

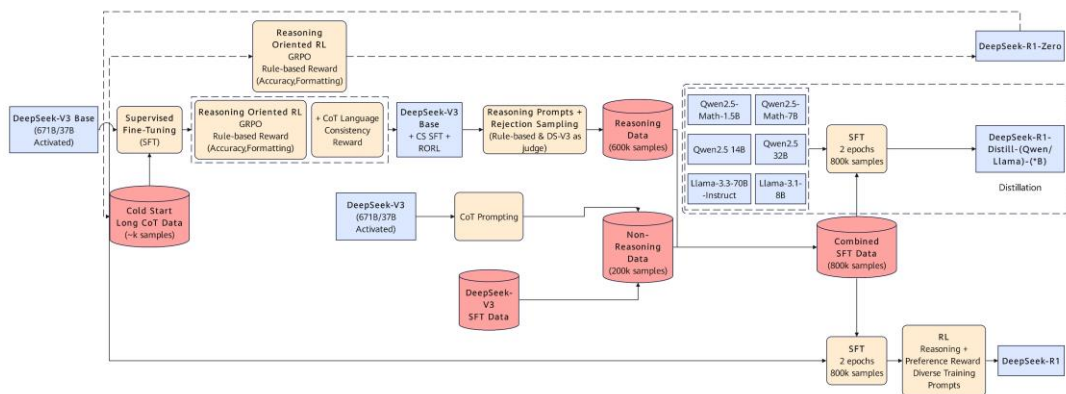
Fewer Truncations Improve Language Modeling

- **Fill-in-Middle：**采用Fill-in-Middle (FIM) 策略，按照Prefix-Suffix-Middle (PSM) 框架以0.1比例结构化数据，提升根据上下文预测中间文本的能力(如中间代码补全)。操作方式：document -> (prefix; middle; suffix) -> (prefix; suffix; middle)
- **subword regularization：**增加组合标点符号以及换行符的token化，同时使用subword regularization缓解新tokenizer带来的token boundary bias问题操作方式：就是在训练时，随机拆分已经分好的token，获得sub-optimal tokenization的结果
- **SFT阶段：1.5M**跨多个域的实例
 - **reasoning data：**推理相关数据集，如数学、编程、逻辑推理
 - 部分来自DeepSeek-R1
 - 对于每个领域，比如代码，数学，通过SFT + RL训练领域专家模型，用于生成对应领域的数据
 - 主要有两类格式：<problem, original response>，<system prompt, problem, R1 response>
 - **non-reasoning data：**创意写作、角色扮演和简单的问答
 - 用DeepSeek-V2.5来生成response
 - 人工来对数据进行检查和更正
- **RL阶段：**构建了偏好数据，这些数据不仅提供了最终的奖励，还包括了导致该奖励的思维链，数据量未知

DeepSeek-R1：构造长思考prompt，基于DeepSeek-V3 迭代生成800K SFT数据

- **DeepSeek-R1-Zero**：希望模型能够通过长思考自我反馈来解决复杂推理问题
 - 数据训练：针对数学/编码等具有明确规则结果的领域，构造`<think><answer>` prompt，使用规则构建奖励模型进行RL，提升长思考能力
 - 训练结果：在推理相关测试集上(数学/编码/逻辑推理题等)，达到OpenAI-o1-0912相当的水平，思考长度不断增加(最终接近10k)，同时出现了Aha moment(在思考过程中对前面的方法进行重新审视)
- **DeepSeek-V3 Base +**：增加千级别long CoT data做SFT，进一步提升效果
 - few-shot example prompt：让DeepSeek-R1-Zero输出可读性较好的、带有reflection(反思)和verification(自验证)的结果，再经过人工校验过滤。
 - readable pattern：进一步提升结果的可读性，在response后面加上一个summary，格式如下：
`|special_token|<reasoning_process>|special_token|<summary>`
- **DeepSeek-R1**：SFT阶段收集800K各领域数据
 - **Reasoning数据收集**：基于DeepSeek-V3 Base+，采样了600k条reasoning data。对比上一轮新增无法用规则作为奖励的数据，DeepSeek-V3作为判别模型，通过输入的ground truth和prediction来判断结果的好坏。此外，添加一些清洗规则：语言混合过滤/长段落/包含代码块等。
 - **Non-Reasoning data**：大约200k的non-reasoning data，来自于DeepSeek-V3的SFT数据。对于某些任务，会调用DeepSeek-V3在回答问题之前先生成一个CoT；而对于某些比较简单的query，比如“hello”这样的打招呼，则不增加CoT
- **蒸馏**：用DeepSeek-R1的800K SFT数据对Qwen2.5等模型做SFT，效果远超原模型

DeepSeek-R1构建流程图



Open R1项目：使用DS-R1生成了8.3M long COT数据，在各领域达到了与 DeepSeek官方相当的性能

- **Open R1**：旨在完全开放复现 DeepSeek-R1
 - **背景**：DeepSeek-R1没有完全开源，训练数据、训练代码等关键信息并未完全公布
 - **当前进展**：当前实现了GRPO、训练与评估代码以及用于合成数据的生成器
 - **构建流程总结**：确定数据领域 → **收集相关开源数据** → **使用DS等模型合成数据** → **数据筛选** → 强化学习训练
- **OpenR1-Math-220k**：
 - **构建流程**：
 - **基础数据合成**：专注于数学推理公式，使用DeepSeek R1为 NuminaMath 1.5 数据集中的问题生成答案，为40 万个问题生成了80万推理轨迹
Prompt前缀：Please reason step by step, and put your final answer within \boxed{ }.
 - **数据筛选**：通过数学验证，只保留至少有一个正确答案的问题，同时使用Llama3.3-70B-Instruct 作为判别器，以筛选出更多正确的样本，特别是那些因格式错误而无法通过基于规则的解析器验证的答案，找回了 2.5 万条被「误判」的数据。
 - **合成结果**：在 OpenR1-Math-220k 训练出来的 Qwen-7B-Math-Instruct，达到了与 DeepSeek-Distill-Qwen-7B 相当的性能
 - **s1K 数据集(李飞飞复现所用数据集)**：
 - 数据构成：包含 1000 个经过精心挑选的数学问题，以及从 Gemini Flash 蒸馏出的推理轨迹
 - 筛选方式：在选择问题时，研究者注重难度、多样性和质量。
 - 训练结果：在 s1K 数据集上对 Qwen2.5-32B-Instruct 进行微调，成功在竞赛数学基准测试中超过了 OpenAI 的 o1-preview，最高提升了 27%
- **Bespoke-Stratos-17k**：
 - 数据来自 APP 和 TACO 的 5k 编码数据、来自 NuminaMATH 数据集的 AIME/MATH/Olympiads 的 10k 数学数据以及来自 STILL-2 的 1k 科学和谜题数据

思考题

1. （单选题）以下哪个选项对大模型预训练的性能影响较小？

- A.数据集质量
- B.数据集规模
- C.数据配比
- D.是否含垂域数据

• D

本章总结

- 本章首先介绍了数据工程的相关内容，如数据工程的功能集合和建设思路，然后介绍了数据治理的关键目标、数据集构建的关键流程等，最后介绍了通用数据处理方法和LLM数据构建方法。

学习推荐

- 昇腾社区
 - <https://www.hiascend.com/>
- MindSpore官网
 - <https://www.mindspore.cn/>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

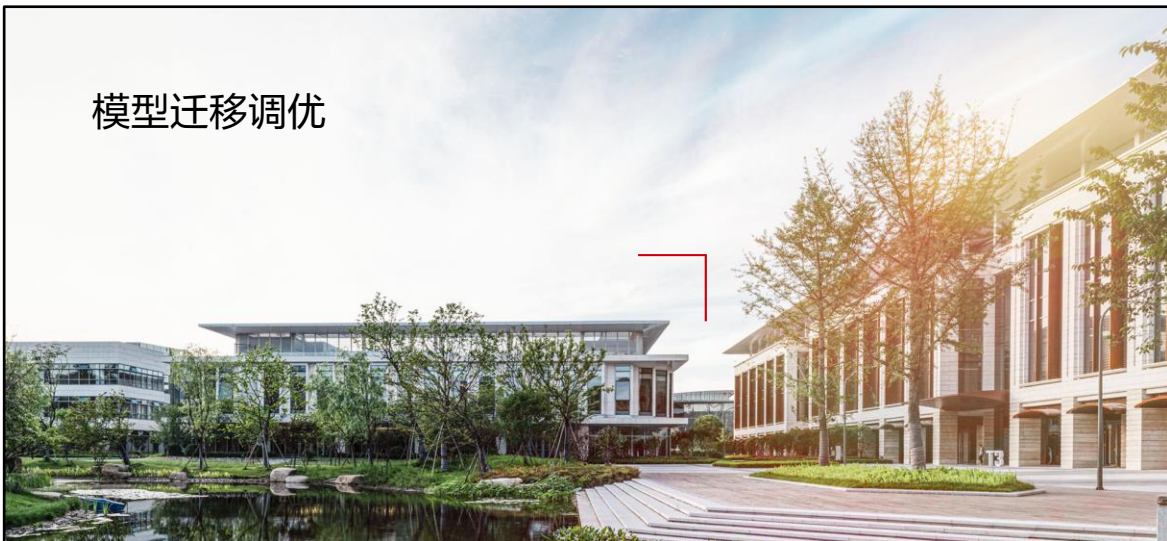
Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



模型迁移调优



前言

- 本章主要讲述了模型迁移的概念和迁移调优流程，首先介绍了什么是模型迁移，接着围绕模型迁移调优的流程讲解了迁移分析、迁移流程、精度调优和性能调优，其中精度调优部分分析了精度差异产生的原因，性能调优部分介绍了常用的调优方法，包括IO优化等。

目标

- 学完本课程后，您将能够：
 - 掌握模型迁移调优的概念；
 - 借助工具完成模型迁移分析；
 - 使用多种方式完成模型迁移；
 - 分析调试模型精度问题；
 - 具备优化模型性能的能力。

目录

1. 模型迁移介绍
2. 模型迁移流程
3. 模型调优流程

什么是模型迁移？

- 模型迁移是指将原本运行在某种硬件平台（如GPU）的深度学习模型迁移至另一种硬件平台（如NPU），或者将使用某种深度学习框架（如PyTorch）的深度学习模型迁移至另一种深度学习框架（如MindSpore），并保障模型在合理精度误差范围内高性能运行。
- 本课程主要讲如何将原本运行在GPU的深度学习模型迁移到NPU上。



模型迁移至昇腾NPU的优势

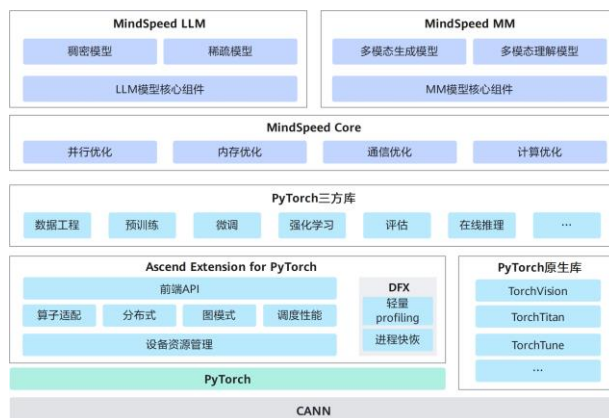
- 性能提升：昇腾平台基于华为自研的处理器架构，专为AI计算设计，能够提供强大的算力支持。对于需要大量计算资源的大模型和应用来说，迁移到昇腾平台可以显著提高性能和处理速度。
- 全栈AI计算基础设施：昇腾平台提供了从硬件到应用服务的全栈AI计算解决方案，包括昇腾系列芯片、硬件、CANN（异构计算框架）、AI计算框架等，这有助于构建端到端的AI应用。
- 优化开发和部署流程：昇腾平台提供了一系列的开发工具和框架，如CANN和MindSpore，这些工具可以简化开发流程，降低开发难度，并提高开发效率。

模型迁移至昇腾NPU的挑战

- 架构差异：NPU（神经网络处理器）与GPU在架构上存在差异，导致基于GPU的训练和推理脚本不能直接在NPU上使用，需要进行迁移和适配。
- 迁移工具和方法：虽然有工具如PyTorch GPU2Ascend可以帮助迁移，但迁移过程可能需要根据实际情况进行适配和调整，这增加了迁移的复杂性。
- 技术文档和支持：虽然昇腾平台提供了技术文档和案例，但在实际操作中，开发者可能仍会遇到特定的技术难题，需要额外的技术支持和社区帮助。
- 兼容性问题：在迁移过程中，可能需要解决不同框架和库之间的兼容性问题，这可能涉及到代码的重写和优化。
- 性能调优：迁移到新平台后，为了充分利用昇腾平台的性能，可能需要对模型和应用进行性能调优，这需要一定的专业知识和经验。

Ascend Extension for PyTorch介绍

- Ascend Extension for PyTorch 插件是基于昇腾的深度学习适配框架，使昇腾NPU可以支持PyTorch框架，为PyTorch框架的使用者提供昇腾AI处理器的超强算力。
- 下面的课程会基于该插件介绍模型迁移调优的方法。



- MindSpeed LLM：LLM模型套件，提供LLM大模型端到端训练流程，提供模型结构增强、微调、偏好对齐、数据工程能力，覆盖业内主流语言大模型。
- MindSpeed MM：多模态模型套件，聚焦多模态生成、多模态理解，提供多模态大模型端到端训练流程，提供多模数据预处理、编码、融合、训练微调等能力，覆盖业内主流多模态大模型。
- MindSpeed Core：昇腾加速库，提供计算、显存、通信、并行四个维度的优化，支持长序列、MoE等关键特性的性能加速。
- Ascend Extension for PyTorch（即torch_npu插件）：昇腾PyTorch适配插件，继承开源PyTorch特性，针对昇腾AI处理器系列进行深度优化，支持用户基于PyTorch框架实现模型训练和调优。
- PyTorch原生库/三方库适配：适配支持PyTorch原生库及主流三方库，补齐生态能力，提高昇腾平台易用性。

目录

1. 模型迁移介绍
2. **模型迁移流程**
 - 模型迁移调优整体框架
 - 迁移分析
 - 模型迁移
3. 模型调优流程

模型迁移调优的整体框架（1）



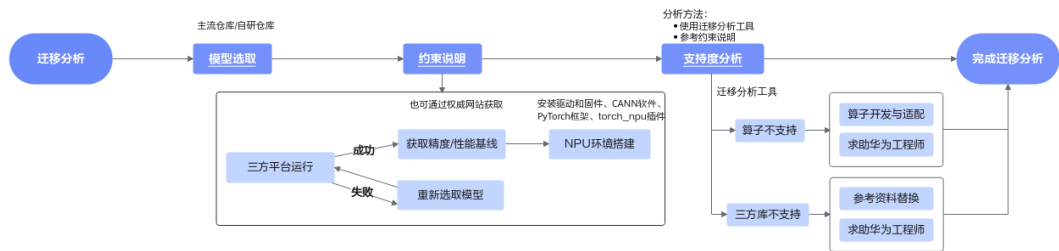
模型迁移调优的整体框架（2）



目录

1. 模型迁移介绍
2. **模型迁移流程**
 - 模型迁移调优整体框架
 - 迁移分析
 - 模型迁移
3. 模型调优流程

迁移分析整体流程



迁移分析

一、模型选取与约束说明

- 调研业务需求场景，选取主流仓库模型。
- 保证选取的模型在第三方硬件平台（如GPU）可成功运行。
- 明确迁移前模型运行的硬件型号、精度、性能基线。可从权威网站或数据平台获取源模型的性能值基线，或在三方平台实测性能基线。

二、迁移支持度分析

- 准备NPU环境，获取模型的源码、权重和数据集等文件。
- 使用迁移分析工具采集目标网络中的模型/算子清单，识别第三方库及目标网络中算子支持情况，分析模型迁移的可行性。
- 算子开发与适配：在迁移支持度分析中如果存在平台未支持的算子，可通过修改模型脚本，使用等价支持的算子替换的方式解决，或者开发算子完成适配。

模型选取&约束说明

- 模型选取
 - 在选取迁移模型时，尽可能选取权威PyTorch模型实现仓，对于大语言模型，使用较为广泛的资源仓库。
- 约束说明
 - 迁移前要保证选定的模型能在GPU或CPU上运行，并输出精度和性能基线。
 - 模型迁移前需要完成昇腾PyTorch训练环境安装，以完成迁移支持度分析与后续的模式训练，可参考相关教程完成环境安装，包括NPU驱动固件、CANN软件toolkit、二进制算子包以及PyTorch框架的安装。
 - 了解目前不支持的优化器或框架。

迁移分析工具（1）

在执行迁移操作前，需借助PyTorch Analyse工具，分析基于GPU平台的PyTorch训练脚本中API、三方库套件、亲和API分析以及动态shape在昇腾AI处理器上的支持情况，分析内容主要包括：

1、API支持情况分析

- 用户提供待分析的PyTorch训练脚本，可快速获得训练脚本中不支持的torch API和cuda API信息，并输出训练脚本中API精度和性能调优的专家建议。

2、三方库套件分析

- 用户提供待分析的三方库套件源码。可快速获得源码中不支持的三方库API和cuda信息。（三方库API是指在三方库代码中的函数，如果某函数的函数体内使用了不支持的torch算子或者cuda自定义算子，则此函数就是三方库不支持的API。如果第三方库中其他函数调用了这些不支持的API，则这些调用函数也为不支持的API。）

迁移分析工具（2）

3、动态shape分析

- 用户提供待分析的PyTorch训练脚本，可快速获得该训练脚本中包含的动态shape信息。

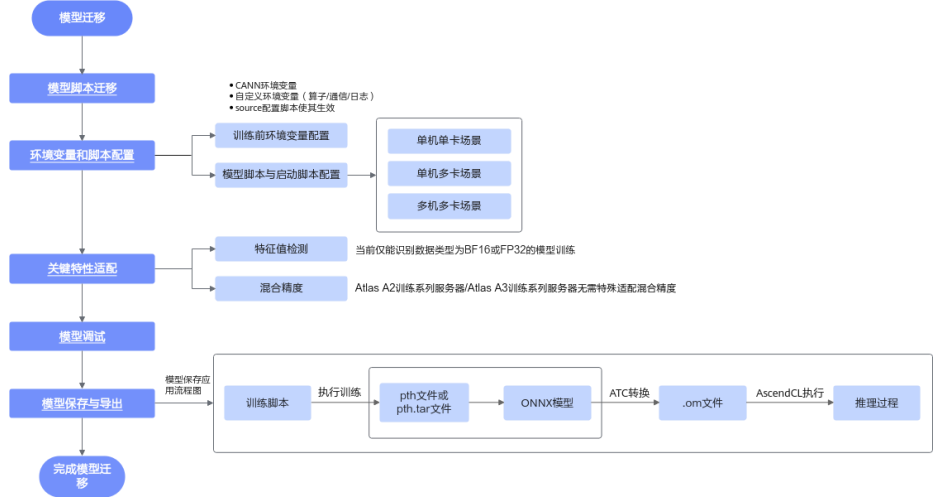
4、亲和API分析

- 用户提供待分析的PyTorch训练脚本，可快速获得该训练脚本中可替换的亲和API信息。在迁移可行性分析中如果存在平台未支持的算子，可通过修改模型脚本，使用等价支持的算子替换的方式解决，或者开发算子完成适配。

目录

1. 模型迁移介绍
2. **模型迁移流程**
 - 模型迁移调优整体框架
 - 迁移分析
 - 模型迁移
3. 模型调优流程

模型迁移整体流程



模型迁移流程（1）

1、模型脚本迁移

- 通过模型脚本迁移，实现GPU->NPU的接口替换、NPU分布式框架改造。

2、环境变量和脚本配置

- 参考文档中环境变量配置，配置训练相关环境变量，以保证模型训练可以在昇腾NPU上正常运行。
- 参考文档中模型脚本与启动脚本配置，根据实际场景选择相应操作完成模型脚本和启动脚本配置。

3、关键特性适配

- 数据类型为BF16或FP32的模型训练过程中出现的收敛异常，可开启特征值检测，用于检测在训练过程中的梯度特征值是否存在异常。
- 在训练时如需混合使用单精度（FP32）与半精度（FP16）数据类型，可参考混合精度适配。

模型迁移流程（2）

4、模型调试

- 迁移适配过程中，如果遇到问题，可以通过模型调试定位问题发生的代码位置。
- 常见问题发生场景包括环境配置，脚本配置，硬件配置与集群配置问题，可从以上场景角度排查问题。

5、模型保存与导出

- 参考模型保存与导出，保存模型文件用于在线推理，使用模型文件导出ONNX模型通过ATC工具将其转换为适配昇腾AI处理器的.om文件用于离线推理。

模型脚本迁移方式

- 自动迁移：在训练脚本中导入脚本转换库，然后拉起脚本执行训练。训练脚本在运行时，会自动将脚本中的CUDA接口替换为昇腾AI处理器支持的NPU接口。整体过程为边训练边转换。
- 工具迁移：使用迁移工具PyTorch GPU2Ascend，自动将训练脚本中的CUDA接口替换为昇腾AI处理器支持的NPU接口，并生成迁移报告（脚本转换日志、不支持算子的列表、脚本修改记录）。训练时，运行转换后的脚本。整体过程为先转换脚本，再进行训练。
- 手工迁移：通过分析模型，对比GPU与NPU接口，手动对训练脚本进行修改，以支持在昇腾AI处理器上执行训练。迁移要点如下：
 - 定义NPU为训练设备，将训练脚本中适配GPU的接口切换至适配NPU的接口。
 - 多卡迁移需将通信方式修改为昇腾支持的HCCL。

自动迁移（1）

自动迁移操作简单，且修改内容少，只需在训练脚本中导入`transfer_to_npu`即可完成脚本迁移。

- 当前自动迁移暂不支持`channel_last`特性，建议用户使用`contiguous`代替。
- 若原脚本中使用的backend为`nccl`，在`init_process_group`初始化进程组后，backend已被自动迁移工具替换为`hccl`。如果后续代码逻辑包含backend是否为`nccl`的判断，例如`assert backend in ['gloo', 'nccl']`、`if backend == 'nccl'`，请手动将字符串`nccl`改写为`hccl`。
- 由于自动迁移工具使用了Python的动态特性，但`torch.jit.script`不支持Python的动态语法，因此用户原训练脚本中包含`torch.jit.script`时使用自动迁移功能会产生冲突，目前自动迁移时会屏蔽`torch.jit.script`功能，若用户脚本中必须使用`torch.jit.script`功能，请使用工具进行迁移。

自动迁移（2）

使用方法

1、在训练脚本中导入transfer_to_npu迁移代码。

- PyTorch2.4.0及之前版本，导入如下迁移代码：

```
import torch
import torch_npu
...
from torch_npu.contrib import transfer_to_npu
```

- PyTorch2.5.1及之后版本，导入如下迁移代码：

```
import torch
...
from torch_npu.contrib import transfer_to_npu
```

2、参考文档中环境变量和脚本配置执行训练。如果训练正常进行，开始打印迭代日志，说明训练功能迁移成功。

3、参考模型保存与导出，如果成功保存权重，说明保存权重功能迁移成功。

自动迁移（3）

注意事项

- 如果模型包含评估、在线推理功能，也可在对应脚本中导入自动迁移库代码，并通过对比评估推理结果和日志打印情况，判断与GPU/CPU是否一致决定是否迁移成功；
- 若训练过程中提示部分CUDA接口报错，可能是部分API（算子API或框架API）不支持引起，用户可参考以下方案进行解决；
- 使用迁移分析工具对模型脚本进行分析，获得支持情况存疑的API列表，进入昇腾开源社区提出ISSUE求助；算子可通过修改模型脚本，使用等价支持的算子替换的方式解决，或者开发算子完成适配。

工具迁移（1）

工具迁移支持使用命令行方式迁移。

- 若用户训练脚本中包含NPU平台不支持的amp_C模块，需要用户手动删除后再进行训练；
- 由于转换后的脚本与原始脚本平台不一致，迁移后的脚本在调试运行过程中可能会由于算子差异等原因而抛出异常，导致进程终止，该类异常需要用户根据异常信息进一步调试解决。

工具迁移（2）

使用方法

1、环境准备

- 安装Ascend-cann-toolkit及相关Python依赖（pandas、libcst、prettytable、jedi）

2、进入迁移工具所在路径

```
cd Ascend-cann-toolkit安装目录/ascend-toolkit/latest/tools/ms_fm_k_transplt/
```

3、进入分析迁移工具所在路径，执行脚本迁移任务。

- `./pytorch_gpu2npu.sh -i 原始脚本路径 -o 脚本迁移结果输出路径 -v 原始脚本框架版本 [-s] [distributed -m 训练脚本的入口文件 -t 目标模型变量名]`
- distributed及其参数-m、-t在语句最后指定。
 - #单卡
 - `./pytorch_gpu2npu.sh -i /home/train/ -o /home/out -v 2.1.0 [-s]`
 - #分布式
 - `./pytorch_gpu2npu.sh -i /home/train/ -o /home/out -v 2.1.0 [-s] distributed -m /home/train/train.py [-t model]`

手工迁移（1）

手工迁移需要用户对AI模型有迁移基础，了解GPU与NPU的接口设计的异同点以及各种迁移手段。手工迁移过程中，各个模型使用的迁移方法不完全相同，下文给出手工迁移的核心要点。

单卡迁移

1、导入NPU相关库。

2、指定NPU作为训练设备。指定训练设备需修改模型训练脚本，有两种指定方式：

- `to(device)`方式：定义好`device`后可通过`xx.to(device)`的方式将模型或数据集等加载到GPU或NPU上，如`model.to(device)`。该方式可以指定需要的训练资源，使用比较灵活；
- `set_device`方式：调用`set_device`接口，指定训练设备。需注意该方式不会自动使用NPU，用户需要手动在想使用NPU的地方，添加`xx.npu()`代码，将模型数据集等加载到NPU上，如`model.npu()`。

手工迁移（2）

3、替换CUDA接口：将训练脚本中的CUDA接口替换为NPU接口，例如模型、损失函数、数据集等迁移到NPU上。

- CUDA接口替换为NPU接口

- 迁移前: `torch.cuda.is_available()`
- 迁移后: `torch_npu.npu.is_available()`

- 模型迁移

- 迁移前: `model.cuda(local_rank)`
- 迁移后: `model.npu(local_rank)`

- 数据集迁移

- 迁移前:
- `images = images.cuda(local_rank, non_blocking=True)`
- `target = target.cuda(local_rank, non_blocking=True)`
- 迁移后:
- `images = images.npu(local_rank, non_blocking=True)`
- `target = target.npu(local_rank, non_blocking=True)`

手工迁移（3）

多卡迁移（分布式训练迁移）

- 针对多卡迁移（分布式训练迁移），除单卡迁移包含的3个修改要点外，在分布式场景下，还需要切换通信方式，直接修改init_process_group的值。

- 修改前，GPU使用nccl方式：

- `dist.init_process_group(backend='nccl',init_method="tcp://127.0.0.1:**", ..... ,rank=args.rank) # **为端口号，根据实际选择一个闲置端口填写`

- 修改后，NPU使用hccl方式：

- `dist.init_process_group(backend='hccl',init_method="tcp://127.0.0.1:**", ..... ,rank=args.rank) # **为端口号，根据实际选择一个闲置端口填写`

环境变量配置

- 在开始训练前，需要先配置训练相关环境变量，用于配置NPU上的PyTorch训练环境，一般使用shell脚本配置，具体包含以下几部分内容：
 - CANN相关环境变量
 - 系统依赖库相关环境变量
 - 自定义环境变量
 - 日志信息配置

关键特性适配-特征值检测（1）

- 特征值检测是针对NPU的PyTorch API，作用是检测在训练过程中激活值的梯度特征值是否存在异常。当前仅能识别数据类型为BF16或FP32的模型在训练过程中出现的收敛异常。
- 检测原理：开启检测开关后，针对模型训练的反向阶段，采集通信算子和模型最外层的输入tensor，根据预先设置的阈值以及历史值，判断当前激活值梯度是否包含异常大值。若出现异常，当“NPU_ASD_ENABLE”设置为“1”时，只打印异常日志，不上报故障事件；当“NPU_ASD_ENABLE”设置为“2”或“3”时，则终止训练，并将检测到异常的设备上的NPU状态置为Warning，上报故障事件。
- 特征值异常原因可分为硬件错误和软件错误两类：
 - 硬件错误：硬件故障导致激活值梯度出现异常大值。
 - 软件错误：模型异常操作或算子越界，导致出现Inf或NaN等异常特征值。

关键特性适配-特征值检测（2）

开启检测开关

- 训练前，配置如下环境变量，使能特征值检测并上报告警：

- `export NPU_ASD_CONFIG=enable:true,with_checksum:true` #开启特征值检测和Checksum联动功能

- 注意

- 开启特征值检测会导致性能损失，建议在满足如下所有条件时开启，否则性能损失可能大于2%。
 - 模型规模：70B及以上；
 - 集群规模：64卡及以上；
 - `torch.npu.set_compile_mode`设置：`jit_compile=False`。
 - 满足以上条件下，在模型上下文并行（Context Parallel）较大时，Ring Attention特性会在模型反向传播过程中引入大量P2P通信，性能损失可能大于2%。

关键特性适配-混合精度适配：AMP

混合精度训练是在训练时混合使用单精度（FP32）与半精度（FP16）数据类型，使用相同的超参数训练，能达到与FP32几乎相同的精度。

混合精度适配支持两种方式：

- 框架内置的AMP功能模块
 - 第三方APEX混合精度模块
-
- 推荐使用PyTorch框架内置的AMP功能模块来进行混合精度训练。
 - 如果使用APEX模块，可参考APEX相关代码仓库（<https://gitee.com/ascend/apex>）。

模型调试（1）

完成脚本迁移后，在训练过程中，如果遇到问题，可以使用以下方法进行调试，确认问题代码发生的位置以及原因。

1、脚本打印定位：使用print打印，判断模型发生问题的代码位置。由于PyTorch框架是异步执行框架，直接打印可能位置不准确，需要添加流同步逻辑辅助打点。（该方法适用于问题定位场景，正式训练可删除。）

- `print(torch.npu.synchronize(),"debug message")`
- `print(inputs.shape, inputs.dtype) #所需打印参数可根据实际情况变更`

2、pdb断点调试：在需要设置断点的部分添加函数。

- 方法一：在代码中引入pdb模块，在需要设置断点的部分添加set_trace函数触发调试器。脚本运行至pdb.set_trace()处会暂停，通过在命令行中输入命令操作，例如输入n可以执行下一行代码，输入p {变量名}可打印当前变量的值，

- `import pdb`
- `pdb.set_trace()`

- 方法二：在需要设置断点的部分添加breakpoint函数，脚本运行至此处会停留。

- `breakpoint()`

模型调试（2）

3、gdb命令行调试：gdb调试工具的主要功能为在程序中设置断点、监视变量、单步骤运行、运行时改变变量值、跟踪路径、线程切换。此方法主要针对coredump场景，执行目录下会生成coredump文件，使用gdb调试该文件并打印堆栈，方法如下：

- 参考GDB官方文档安装GDB。
- 设置生成coredump文件（ulimit设置）。
 - `ulimit -c` # 查看当前设置，0表明不生成coredump文件，需要更改ulimit设置
 - `ulimit -c unlimited` # unlimited将生成coredump文件大小设置为无限制，此时如果进程崩溃就会生成coredump文件
- 设置coredump文件存储位置和名称。
 - # 临时修改生成的coredump文件的名称，加粗命令为文件名称的变量，可自行设置
 - `sysctl -w kernel.core_pattern=core-%e.%p.%h.%t`
 - # 设置coredump生成目录，加粗命令为文件名称的变量和生成目录，可自行设置
 - `echo "/pathtocoredump/core.%e.%p" >/proc/sys/kernel/core_pattern`

模型调试（3）

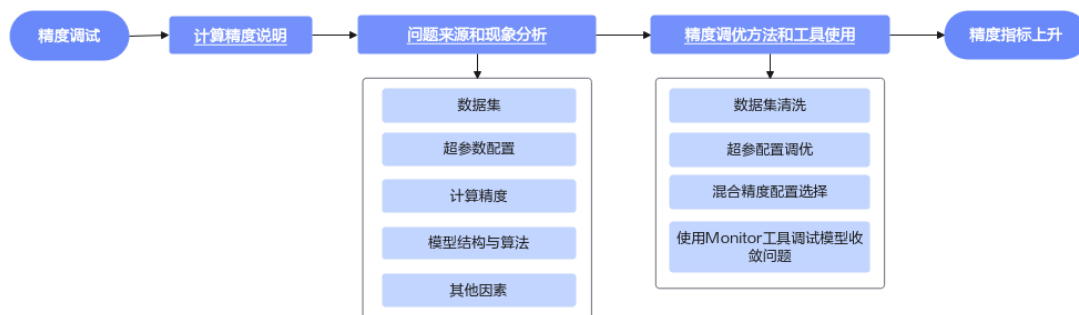
- 生成coredump文件。
 - 运行模型脚本，若模型报错、进程崩溃，即可在当前目录下生成coredump文件。
- 调试coredump文件。
 - 执行如下命令进入gdb模式，调试coredump文件。
 - `gdb python3 core.*` #coredump文件名请根据用户自己的设置修改

4、Hook定位：该方法主要针对定位模型中某个module的报错，通常适用于正反向module的报错定位。

目录

1. 模型迁移介绍
2. 模型迁移流程
- 3. 模型调优流程**
 - 精度调试
 - 性能调优

精度调试整体流程



精度分析与调优的作用

- 训练过程中的模型精度问题分析，及时处理训练不稳定问题;
- 分析评价集loss/ppl和模型下游评测任务得分，评估迁移前后的精度差异;
- 确保迁移前后模型精度差异在可接受范围之内，数据无异常溢出；如果出现精度相关问题，需要借助精度问题分析工具分析。

计算精度说明（1）

- NPU与主流AI处理器之间的计算差异

- 浮点舍入模式的差异

- 在浮点舍入模式方面，主流AI处理器支持四种舍入模式，而NPU在主流AI处理器的四种模式基础上，增加了A模式和O模式。尽管所有舍入模式都会带来数值精度误差，但NPU通过增加舍入模式的选择，为用户提供了更多灵活性以适应不同的计算需求。相比主流AI处理器，NPU的舍入模式更丰富。

- A模式：最近舍入并在尾数为五时选择远离零舍入（正数选择更大数的方向，负数选择更小数的方向）。

- O模式：最近舍入并在尾数为五时向奇数舍入。

- 向最接近的可表示的值。

- 当有两个最接近的可表示的值时首选“偶数”值。

- 向负无穷大（向下）。

- 向正无穷大（向上）以及向零（截断）。

- 默认模式是最近舍入（Round to Nearest），它与四舍五入只有一点不同，对.5的舍入上，采用取偶数的方式。最近舍入模式：Round(0.5) = 0；Round(1.5) = 2；Round(2.5) = 2。四舍五入模式：Round(0.5) = 1；Round(1.5) = 2；Round(2.5) = 3。

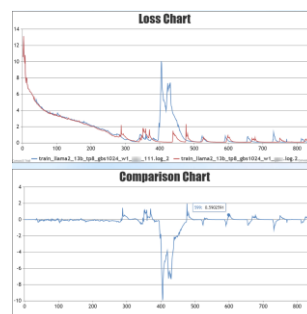
- 计算过程的差异

- 计算过程的差异包括计算使用的浮点精度、累加运算的顺序等。计算过程中的细微差异都能引起最终输出结果的不同，从而产生跟标杆比对程度不一的误差。

计算精度说明（2）

• 精度调试的前提条件

- 小的计算差异在某些超参配置下（例如大学习率、大adam_beta1、大adam_beta2）以及某些场景下（例如浅层参数中极小数值的梯度）会被放大，从而在训练稳定性的表现上出现很大差异。例如主流AI处理器或NPU其中一方不稳定（很多loss尖刺），或者双方都不稳定，这在大规模训练和训练对比实践中都非常常见。
- 小的计算差异会引起AI处理器（主流/NPU）训练的模型参数位于不同的loss landscape位置。因为loss landscape的复杂性，主流AI处理器/NPU的某一方可能处于地形图中非常陡峭的位置，从而在对比中比较不利。特别是当超参不合理时（如大学习率），这种对比的不确定性就更强。LLaMA2-13B模型在1e-3基础学习率下，主流AI处理器运行两次训练得到的loss曲线，如图所示。图中可见主流AI处理器运行两次训练得到的loss也很难对齐，尖刺发生的位置有很大不确定性。期望在训练不稳定情况下完全对齐，在主流AI处理器上也是不可能完成的任务。因此，精度调试特别是精度对齐的前提条件是要能稳定训练。



• 计算差异对训练结果的影响

- Google TPU上的实验证明了计算精度和模型精度的复杂关系。
- 实验中，将82.3%~90.3%的比特翻转故障注入实验，发现训练loss不会有显著差异，其中还有65.5%~86.3%的loss甚至略微优于无故障基准。故障产生了噪声，相当于在模型结构中引入了正则层。所以计算或者通信的某些精度差异，并不一定会导致训练更不稳定。

精度调试过程案例

- 案例一：主流AI处理器/NPU的loss对齐
 - 用户使用高学习率 $1e-3$ 训练LLaMA2-13B模型，在主流AI处理器和NPU上都发生训练不稳定问题，两者在loss尖刺处始终不能对齐。后来改为低学习率 $1e-4$ 后，训练稳定性得到极大提升，并且两者的loss也能很好对齐。
- 案例二：DeepSpeed bug引起的偶发loss尖刺
 - DeepSpeed分布式优化器中存在计算通信同步bug，会导致脏数据进入，从而引起训练不稳定。这在主流AI处理器和NPU表现是等同的。
- 案例三：LLaMA2-1B，7B和13B的训练稳定性问题和下游评测不达标问题
 - 一开始使用了错误的adam_beta2导致训练不稳定，修复后在13B语言模型上又遇到训练稳定性问题，调低学习率到 $1e-4$ 并且关闭dropout后成功训练13B大语言模型，并且在各种下游评测任务上超出预期，实现了业务目标。

精度问题来源

主要来源于大模型训练：数据集、超参数、模型结构和算法、浮点计算精度等。

- 数据集：数据异常触发的训练不稳定、语料混合比例与类型的重要性、数据规模与模型参数规模的匹配。
- 超参数配置（主要包括优化器选择、学习率设定、梯度裁剪阈值、Loss scale等）：优化器与学习率、梯度裁剪阈值、Loss scale动态调整、Batch尺寸的影响。
- 模型结构与算法：不同算法影响模型训练效果和收敛的Loss水平，但不影响收敛性。大模型复杂性增加算法实现错误，影响收敛性。新模型结构需调试确保无误，以实现Loss收敛和性能匹配。
- 计算精度：包括浮点数计算、大模型训练常用数据格式（单精度、半精度浮点数）、张量分布式并行计算（词嵌入输入层embedding张量并行、线性层张量并行、MLP层切分、流水线并行）、混合精度计算分析等。

调优方法

- 数据集清洗：非目标语言与低质量样本剔除、去重
- 超参配置调优：Batch Size配置、学习率、优化器、权重初始化、其他稳定训练技术，例如梯度裁剪（Gradient Clipping）、Weight Decay（L2正则化）、特殊层的调整；
- 使用TensorProbe工具调试模型收敛问题：确定Llama2 loss尖刺根因、确定Open Sora在deepspeed ZeRO-1设置下不收敛问题；
- 混合精度配置选择：如果混合精度训练中选择float16，为了避免表示范围小引起的浮点上溢和下溢，混合精度要结合动态Loss缩放机制。另外，特定张量计算操作对计算精度也有不同的要求，如Softmax函数、矩阵乘法（Matmul）等使用FP32，卷积操作与池化、循环神经网络（RNNs）等使用BF16。

训练状态监控和恢复

监控：

- 在大模型训练过程中，为确保训练的稳定性与有效性，需密切关注多项关键指标以评估训练状态，其中包括但不限于perplexity(PPL)、gradient norm (GNorm)、activation norm、内存占用情况以及Loss scale等参数。推荐采用TensorBoard工具进行数据可视化。
- 另外，在模型的训练中，我们可以通过PyTorch中的hook机制对容易出现问题的某些层配置hook，监控这些层的梯度信息，及时处理出现异常的step以减少对模型训练效果的影响。
- 对该梯度信息进行监控和分析，可以检查训练中的一些异常状态，收集梯度值时，建议只采集梯度的最大值、最小值、平均值等统计信息作为训练状态的监控指标。
- 除了梯度爆炸问题的监控，我们也要关注梯度下溢问题。特别是对于FP16，如果梯度出现大量0，往往意味着训练出现不正常，这时候需要停止训练，找出问题（例如伪FP32，实际是混合精度）。如果训练器使用了Loss scale，则监控Loss scale是必须的。

恢复：

- 当发现训练异常状态时，除了最常用的clip_grad之外，业界实践中还会用到跳过GradNorm异常增大的step、回滚到正常状态的checkpoint重新拉起训练，以及重新训练时清空优化器状态并从头warmup。

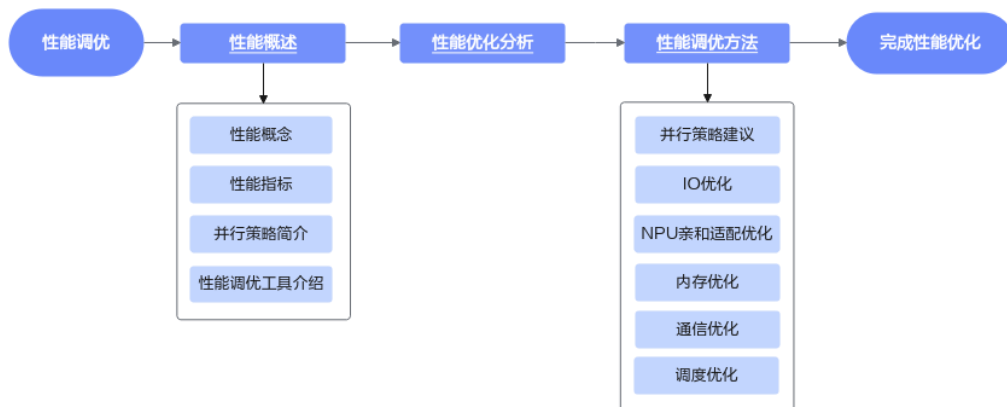
保存&导出模型

- PyTorch在训练过程中，通常使用torch.save()来保存Checkpoint文件，根据模型文件的后续用途会保存为两种格式的模型文件（pth文件和pth.tar文件），以便用于在线推理。pth或pt扩展名的文件，用于在线推理或导出ONNX格式模型，仅保存模型参数，不保存模型结构，以便压缩文件的体积，可以用Netron等可视化工具打开。通过state_dict来保存和加载模型（保存为后缀是.pth/.pt的文件时，需要提供模型定义文件，否则后续模型无法部署）。
- 模型训练完成后，用户可以使用pth文件和pth.tar文件导出ONNX模型，然后通过ATC工具将其转换为适配昇腾AI处理的.om文件用于离线推理。
- 下面主要介绍如何将Checkpoint文件通过torch.onnx.export()接口导出为ONNX模型。
 - ONNX是业内目前比较主流模型格式，广泛用于模型交流及部署。PyTorch模型在昇腾AI处理器上的部署策略是基于PyTorch官方支持的ONNX模块实现的。
 - 使用PyTorch框架导出ONNX模型时，框架中设置算子编译选项的ACL_OP_SELECT_IMPL_MODE选项默认值为“high_precision”，用户可根据需要自行修改。用户在使用导出的ONNX模型进行模型转换时，建议设置与训练时相同的模式，以避免因模式选择不同而出现的精度或者性能差异。

目录

1. 模型迁移介绍
2. 模型迁移流程
- 3. 模型调优流程**
 - 精度调试
 - 性能调优

性能调优整体流程



性能概念和指标

- 在计算性能指标中，优先级排序为：吞吐率>单步迭代时间>线性度>内存占用>带宽占比>训练效率>浮点计算次数每秒>算力利用率。
- 对于一个batch而言，时间主要由以下部分构成：单batch总时间=数据加载时间+模型前反向时间+优化器时间+模型后处理时间+通信时间+调度时间

指标名称	单位
吞吐率	samples/s、tokens/s
单步迭代时间	s
线性度	values
内存占用	%
训练效率	tokens/day
浮点计算次数每秒	TFLOPS/s
算力利用率	%
通信性能指标	%
Pipeline并行效率指标	%

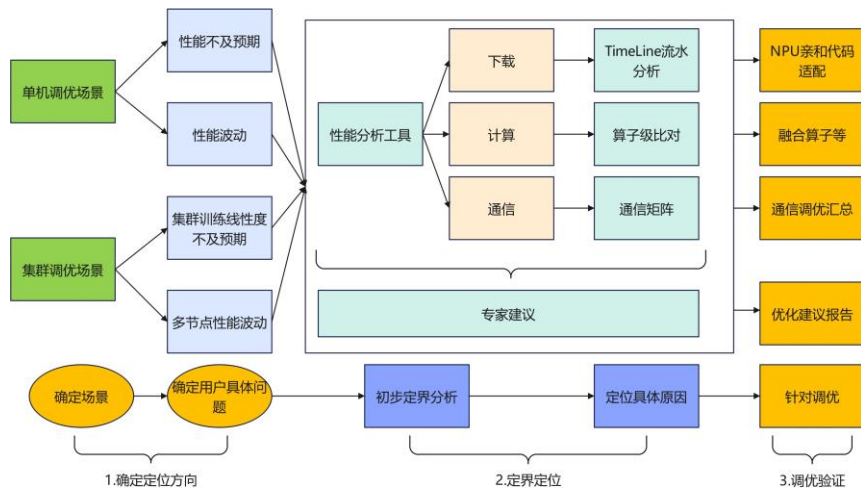
性能分析工具

性能分析工具是一套基于CANN的分析工具，其目的在于分析NPU在运行中模型的训练效率，在训练有瓶颈或者有性能优化需求的时候，可以利用性能分析工具进行性能分析。

工具主要包括：

- **Ascend PyTorch Profiler**：基于CANN实现，用于分析CANN上硬件的运行效率和性能数据。
- **MindStudio Insight**：是一款调优可视化工具，集成了CANN数据的分析和可视化等功能。
- **性能比对工具、集群分析工具**：取自开源社区，对Ascend PyTorch Profiler采集得到的数据进行专项分析。

性能优化流程



通用优化方法--并行策略

- 在面临显存不足、模型过大无法完全加载以及需要进行切分的情况下，优先考虑使用TP（Tensor Parallelism）进行切分，并确保切分的数量小于等于机器内的计算卡数。
- 如果在TP切分达到最大显存容量仍然不足的情况下，可以考虑在机器之间使用PP（Pipeline Parallelism）进行切分。理论上，PP的数量应该越小越好，以尽可能减少空闲计算资源的浪费。
- 在机器资源富裕的情况下，可以开启DP（Data Parallelism）并行，将计算任务分配给多个机器进行并行处理，从而提高处理效率。然而，在机器资源有限的情况下，如果开启TP+PP切分后显存仍然不足，可以考虑使用ZeRO1和重计算技术。
- 此外，即使在模型能够成功运行的情况下，也可以尝试主动地使用降低内存占用手段，例如ZeRO1和重计算等，然后增大Batch Size。这样有时也会取得令人意外的效果。

通用优化方法—IO优化和调度优化（1）

1、数据加载优化

在PyTorch模型中，数据加载部分的逻辑一般是DataLoader及其衍生类，在DataLoader加载数据中，要注意以下两个核心点：

- 第一个是数据加载的预处理部分，数据的预处理通常会写在datasets里，数据的预处理包括对文本、图片、视频和语音等不同格式数据的处理，数据预处理耗时长是比较容易识别出来的，一般通过打点计时。
- 第二个就是要确定好数据读取的方式，一般而言，每张卡都去读取数据是比较理想的，如果存在0卡读取，广播给其他卡的数据读取形式，要关注其性能如何，很多时候都会让模型性能严重劣化。

通用优化方法—IO优化和调度优化（2）

①、硬件类

- 检查数据存放的硬盘，最好在NVMe（Non-Volatile Memory Express，一种更快、更可靠的存储设备，适合存储需要高速读写的数据）盘上，当硬盘为共享存储时，需要注意IO瓶颈。
- 检查NUMA（Non-Uniform Memory Access，非一致性内存访问）数量，可以通过lscpu命令查看，一般需要NUMA数量为4或者8。

②、参数类

- DataLoader入参：
 - num_workers是一个比较常用的参数，一般而言，对于图像或视频等复杂数据，可以将num_workers设置的大一些（如4或8），通过多进程并行提高数据读入和解析的速度。在使用这个参数的时候，需要结合实际情况，如视频解析本身并不存在瓶颈，过多的num_workers则会增加进程开销，因此需要酌情考虑。
 - 在内存允许的情况下，保持pin_memory=True，在PyTorch中，该参数可以写为pin_memory_device。
 - 一般而言，可以加上persistent_workers=True，该设置可以减小进程销毁或申请的开销，不过这个方法也可能带来内存瓶颈。

通用优化方法—IO优化和调度优化（3）

③、代码技巧类

- 在数据预处理时间较长而影响模型训练时，可以尝试预取数据，即Data prefetcher。
- Infinite DataLoader同样可以缓解epoch间加载缓慢的问题。
- 提前缓存数据或制作二进制数据，轻量化数据加载，若原始数据是压缩或编码格式，则会引发解压缩或解码，加重CPU负载，容易引起host bound问题，建议提前处理成二进制格式，快速进行数据读取。
- 对于NPU预处理数据比较慢，例如dlrm模型，可以考虑将预处理放到datasets里用CPU多num_workers处理数据。
- 对于句子之间长度差距比较大的句子，组成batch时可以考虑resample一下。对于NLP任务，存在输入的句子长短不一，tokenize之后需要padding成相同长度才能组成一个batch的情况。对于一些数据集中，所有sentence长度差别很大的情况，如果随机取句子，那么需要补的padding就很多，造成很多冗余的计算量。此时，可以通过调整DataSampler，尽量将长度接近的句子组成一个batch，这样造成很多冗余的计算量就可以减少很多，训练整体的吞吐量会有显著提升。缺点：可能稍微会影响数据的随机性。

通用优化方法—IO优化和调度优化（4）

- `collate_fn`用于将多个workers从datasets中读取的`batch_list`组合成一个batch，这个组合过程是可以加速的，例如原来每个workers都要对各自的数据transpose并且减均值除方差再合并成一个batch。此时可以先合并成一个batch，再放到Device上，并且用Device算力完成transpose和除方差操作，达到减轻CPU负载的目的。以此原理，timm仓库设计了`fast_collate`函数，已经是timm训练的标配，能够有效提升性能。

2、调度优化

- 一级流水优化迁移负载
 - 此环境变量用于优化task queue一级流水负载，减少dequeue唤醒时间。
 - 一般用于以下场景：host-bound/严重网络场景。
 - `export TASK_QUEUE_ENABLE=2`
- 自动绑核
 - 绑核功能通过绑定指定CPU核心和设备指定卡，来减少核心切换开销，优化调度性能。
 - //当前仅有方案1：将CPU核数按环境卡数平均分配并绑核，如192核8卡场景，0卡会绑定至0-23，1卡绑定至24-47，以此类推。
 - `export CPU_AFFINITY_CONF=1`

通用优化方法-NPU亲和适配优化

1、融合算子

- 融合算子的优化原理为：通过数学意义上的等价替换，将多个算子融为一个算子的计算，减少冗余计算，同时减少下发次数，从而提高性能。

2、消除多余的stream同步

- 在很多开源代码中，由于作者在编写时很少考虑性能，因此在代码中可能存在增加很多的stream同步操作，这些同步通常是由h2d（host to device，从CPU侧下发到NPU侧）、d2h（device to host，从NPU侧搬回到CPU侧）操作（如tensor.item、reduce_all、torch.isfinite等）引入的，原则上，我们需要尽可能减少这些异步流同步的操作，让模型通过异步流实现最佳的并行效率。有时候因为下发或通信等问题，可能导致多卡训练时，卡与卡之间性能不一致，此时由于反向计算后需要对多卡之间的梯度进行同步，产生快卡等慢卡的情况。性能数据分析时，可以明显观察到部分卡上集合通信算子耗时占比高，这种现象的根因并不是集合通信算子耗时久，而是因为性能快的卡在等待性能慢卡，这种问题，要明确找到快卡和慢卡差异产生的第一位置，这个可以通过性能工具来寻找。

3、部分CPU上运行的优化方法

- 在mask-rcnn中，模型的mask decode运行CPU上进行，且仅用单核实现，性能不够理想。此时，可以将计算部分放到NPU上，或者在CPU上，通过多进程的方式进行加速。

通用优化方法-内存优化（1）

1、调整内存参数

- 设置内存因子，限制进程申请的内存上限，取值0~1。
 - 例如：设置0.95，可使用的内存上限为60G * 0.95=57G。
 - 内存申请超过上限触发内存释放。设置PyTorch申请的内存上限，可以避免内存使用极限场景下，PyTorch耗尽Device内存，其他组件申请内存失败导致的进程异常。
 - `torch_npu.npu.set_per_process_memory_fraction(0.95)`
- 垃圾回收阈值，默认不开启，与内存因子配合使用，取值0~1，为内存上限的百分比。
 - 当内存申请到gc阈值，则触发内存池空闲内存块回收。建议由大到小配置调试。
 - `export PYTORCH_NPU_ALLOC_CONF="garbage_collection_threshold:0.95"`
- 内存块允许切分上限，单位MB。
 - 例如：设置50，大于等于50M的内存块不允许切分使用。设置该参数可避免大内存块被切分导致较多的内存碎片影响内存复用。调试时可以先采集内存profiling，按照算子内存申请降序排列，参数值由大到小尝试。
 - `export PYTORCH_NPU_ALLOC_CONF="max_split_size_mb:50"`

通用优化方法-内存优化（2）

- 使能内存池扩展段功能，由PyTorch自己管理虚拟地址与物理地址映射，降低内存碎片。
 - 对于动态shape场景，shape随step增加而增大，从而导致内存块不能复用内存碎片上升，对该场景有较好优化。
 - `export PYTORCH_NPU_ALLOC_CONF="expandable_segments:True"`

2、多流复用

- 多流复用是PyTorch侧非常有效的内存优化方法，需要配置环境变量，代码如下：
- `export MULTI_STREAM_MEMORY_REUSE=1`

3、减小HCCL通信缓存

- 可以通过调整以下数值，来控制HCCL的缓存大小。默认值为200。
- `export HCCL_BUFFSIZE=200`

通用优化方法-内存优化（3）

4、Python GC优化

- 在大模型训练中，我们可能因为大量进程的开销而频繁触发GC进行垃圾回收，而垃圾回收在一定程度上会占用CPU的性能，如果频繁触发，可能会造成模型训练过程中性能的抖动。
- 在遇到此类问题时，首先可以主动通过`gc.disable()`关闭自动垃圾回收机制；同时，通过如下接口设置回收频率和回收次数，保证内存稳定的同时降低性能抖动。
 - `gc.set_threshold(threshold0, threshold1, threshold2)`
 - 其中`threshold0`、`threshold1`及`threshold2`分别代表python的三代对象。
- 所有新建的对象都是0代对象，当某一代对象经历过垃圾回收，依然存活，那么它就被归入下一代对象。垃圾回收启动时，一定会扫描所有0代对象，如果0代经过一定次数的垃圾回收，那么就会启动对0代和1代的扫描清理；当1代也经历了一定次数的垃圾回收，那么会启动0、1和2，即所有对象的扫描。命令示例如下：
 - `gc.set_threshold(700, 10, 5)`
 - 其中700、10、5分别代表0、1、2三代最大的引用数量，超过引用上限则会触发GC进行垃圾回收。

通信基本术语

- **HCCL**（Huawei Collective Communication Library，华为集合通信库）是基于昇腾AI处理器的高性能集合通信库，提供单机多卡、多机多卡集合通信原语，在PCIE、HCCS和RoCE高速链路实现集合通信功能，实现分布式训练。
- **HCCS**（Huawei Cache Coherent System），华为一致性系统总线，用于CPU和NPU之间互联的高速总线。
- **PCIE**（Peripheral Component Interconnect Express），一种串行外设扩展总线标准，用于计算机系统的外设扩展使用。
- **DMA**（Direct Memory Access）指一种高速的数据传输操作，允许在外部设备和存储器之间直接读写数据，既 not 通过CPU，也不需要CPU干预。
- **RDMA**（Remote Direct Memory Access），远端直接内存访问技术，一般指能够跨过网络的DMA方式。
- **RoCE**（RDMA over Converged Ethernet），承载在融合以太网上的RDMA技术，即跨越以太网的RDMA通信方式，本文特指RoCEv2。

通用优化方法-通信优化（1）

优化方法：

1、HCCL_INTRA_ROCE_ENABLE

- Atlas 200TA2 Box16异构子框进行单server16P单server业务部署，开启该环境变量后，两个8P之间使用RDMA链路代替SDMA链路作为mesh间互联链路。
- `export HCCL_INTRA_ROCE_ENABLE=1`
- 当用户使用设备虚拟化场景或PCIE带宽较低时（如不足20GB/s），建议使用此参数。

2、HCCL_RDMA_TC

- 用于配置RDMA网卡的traffic class。8bit无符号整型数据，bit2~7配置为DSCP值，bit0~1固定为零；默认值132。
 - #DSCP配置为25
 - `export HCCL_RDMA_TC=100`
- 昇腾网卡与交换机QOS（Quality of Service，服务质量）不匹配导致RDMA通信带宽下降时配置。

通用优化方法-通信优化（2）

3、HCCL_RDMA_SL

- 用于配置RDMA网卡的service level，该值需要和网卡配置的PFC优先级保持一致，若配置不一致可能导致性能劣化。取值范围[0,7]，默认值4。
 - #优先级配置为3
 - export HCCL_RDMA_SL=3
- 昇腾网卡与交换机QOS（Quality of Service，服务质量）不匹配导致RDMA通信带宽下降时配置。

4、HCCL_BUFFSIZE

- 此环境变量用于控制两个NPU之间共享数据的缓存区大小。单位为M，取值需要大于等于1，默认值是200M。
- 该环境变量申请的内存为HCCL独占，不可与其他业务内存复用。
- 每个通信域占用“2*HCCL_BUFFSIZE”大小的内存，分别用于收发内存。
- 该资源按通信域粒度管理，每个通信域独占一组“2*HCCL_BUFFSIZE”大小的内存，保证多通信域并发算子互不影响。
 - export HCCL_BUFFSIZE=200
- 当业务部署内存不足时，根据功能描述中的配置策略配置，小于配置策略会导致通信性能下降。

思考题

1. （多选题）以下哪些选项会导致模型迁移后的精度问题？

- A. 硬件设计
- B. 算子实现
- C. 网络结构
- D. 数据集

- AB

本章总结

- 本章首先讲述了模型迁移的概念和优势，然后介绍了Ascend Extension for PyTorch的设计理念，最后介绍了模型迁移调优的各个部分，包括迁移分析、模型调试等各个流程。

学习推荐

- 昇腾社区
 - <https://www.hiascend.com/>
- MindSpore
 - <https://www.mindspore.cn/>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

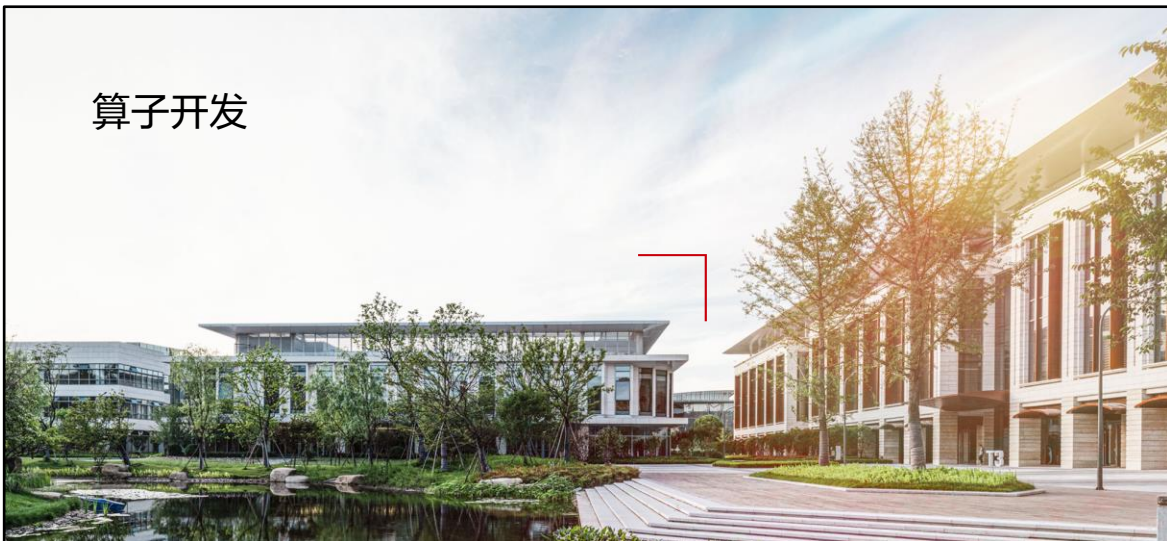
Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



算子开发



前言

- 本章主要讲述了如何进行算子开发，包括算子的概念及其包含的属性，然后介绍了昇腾AI处理器的架构、Ascend C的设计理念，只有了解这两者才能开发出适配昇腾的算子，后面以Vector算子开发为例，介绍了算子开发的基础知识和算子开发流程，最后讲解了算子开发两种不同工程模式的差异。

目标

- 学完本课程后，您将能够：
 - 掌握算子的基础概念；
 - 掌握Ascend C的特性与昇腾AI处理器架构抽象；
 - 具备使用工具创建算子工程的能力；
 - 掌握算子开发的流程；
 - 具备开发Vector算子的能力。

目录

1. 算子基本概念

- 算子是什么
- 算子开发复杂性

2. 昇腾AI处理器与Ascend C

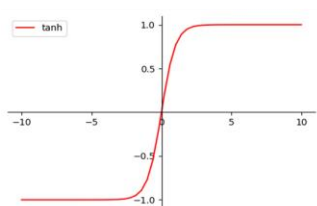
3. 算子开发基础知识

4. Vector算子开发流程

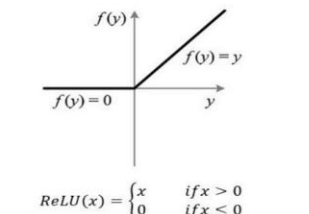
5. 算子开发工程

什么是算子 — 算子的数学含义

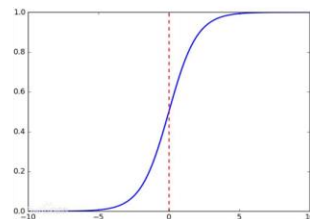
- 算子在数学中的定义：
- 一个函数空间到函数空间上的映射 $O: X \rightarrow X$;
- 广义的讲，对任何函数进行某一项操作都可以认为是一个算子。比如微分算子，不定积分算子等。
- 常见算子举例：tanh、ReLU、sigmoid等。



Tanh函数图像



ReLU函数图像



Sigmoid函数图像

算子基本概念 — 总览

- 算子名称

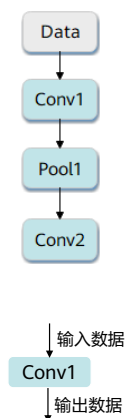
- 算子的名称，用于标志网络中的某个算子，同一网络中算子的名称需要保持唯一。如右图所示Conv1，Pool1，Conv2都是此网络中的算子名称，其中Conv1与Conv2算子的类型为Convolution，表示分别做一次卷积运算。

- 算子类型

- 网络中每一个算子根据算子类型进行算子实现的匹配，相同类型的算子的实现逻辑相同。在一个网络中同一类型的算子可能存在多个，例如右图中名称为Conv1的算子与Conv2算子的类型都为Convolution。

- 数据容器

- 前面介绍了算子的类型与算子的名称。我们再看下网络中的算子，如右图所示，算子在网络中执行时，还需要一个重要的输入：数据，算子执行完后，也会有对应的数据输出。这种承载算子数据的容器定义为：张量（Tensor）。



算子基本概念 — Tensor

- 张量（Tensor）是存储算子输入数据与输出数据的容器，而张量描述符（TensorDesc）是对输入数据与输出数据的描述，张量描述符的数据结构包含如下属性：

属性	定义
名称（name）	用于对Tensor进行索引，不同Tensor的name需要保持唯一。
形状（shape）	Tensor的形状，比如（10，）或者（1024, 1024）或者（2, 3, 4）等。 形式：(i1, i2,...in)，其中i1到in均为正整数
数据类型（dtype）	指定Tensor对象的数据类型。 例如：float16, float32, int8, int16, int32, uint8, uint16, bool等。 不同计算操作支持的数据类型不同。
数据排布格式（format）	数据的物理排布格式，定义了解读数据的维度。

- 形状的举例:（10，）就是一个数组，(1024,1024)是一个矩阵，(2,3,4)是一个立方体。
- 一个不管是什么形状的张量，在内存中的都是线性存储的，所以如何对这个张量进行遍历，就很重要，比如是先遍历x轴，还是先遍历y轴。不同的遍历方式影响了张量在内存中的存储形式（即存储的一维数组），所以我们用数据排布格式来定义这个遍历顺序。

算子基本概念 — Shape (1)

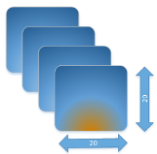
- 下面分别介绍张量描述符中的形状和数据排布格式。
- 形状
 - 张量的形状，以(D0, D1, ... ,Dn-1)的形式表示，D0到Dn是任意的正整数。如形状(3,4)表示第一维有3个元素，第二维有4个元素，是一个3行4列的矩阵数组。在形状的小括号中有多少个数字，就代表这个张量是多少维的张量。形状的第一个元素要看张量最外层的中括号中有几个元素，形状的第二个元素要看张量中从左边开始数第二个中括号中有几个元素，依此类推。

▫ 例如：

张量	形状
1	(0,)
[1,2,3]	(3,)
[[1,2],[3,4]]	(2,2)
[[[1,2],[3,4]], [[5,6],[7,8]]]	(2,2,2)

算子基本概念 — Shape (2)

- 下面我们看一下形状的物理含义，例：shape=(4, 20, 20, 3)
- 假设有4张照片，即shape里4的含义，每张照片的宽和高都是20，也就是 $20 \times 20 = 400$ 个像素，每个像素点都由红/绿/蓝3色组成，即shape里面3的含义，这就是shape=(4, 20, 20, 3)的物理含义，如右图所示。



- 如果体现在编程上，可以简单把shape理解为操作Tensor的各层循环，比如我们要对shape=(4, 20, 20, 3)的A_tensor进行操作，循环语句如下：

```
produce A {  
  for (i, 0, 4) {  
    for (j, 0, 20) {  
      for (p, 0, 20) {  
        for (q, 0, 3) {  
          A[((((i*20) + j)*20) + p)*3 + q] = a_tensor[((((i*20) + j)*20) + p)*3 + q]  
        }  
      }  
    }  
  }  
}
```

算子基本概念 — Format

- 数据排布格式

- 在深度学习领域，多维数据通过多维数组存储，比如卷积神经网络的特征图（Feature Map）通常用四维数组保存，即4D格式：
 - N: Batch数量，例如图像的数目。
 - H: Height，特征图高度，即垂直高度方向的像素个数。
 - W: Width，特征图宽度，即水平宽度方向的像素个数。
 - C: Channels，特征图通道，例如彩色RGB图像的Channels为3。
- 由于数据只能线性存储，因此这四个维度有对应的顺序。不同深度学习框架会按照不同的顺序存储特征图数据，比如Caffe，排列顺序为[Batch, Channels, Height, Width]，即NCHW。TensorFlow中，排列顺序为[Batch, Height, Width, Channels]，即NHWC。
- 以一张格式为RGB的图片为例，NCHW中，C排列在外层，实际存储的是“RRRRRRGGGGGGBBBBBB”，即同一通道的所有像素值顺序存储在一起；而NHWC中C排列在最内层，实际存储的则是“RGBRGBRGBRGBRGBR”，即不同通道的同一位置的像素值顺序存储在一起。



算子基本概念 — Format

- 下面再以一个例子讲解一下数据排布格式的概念。下图是同一张图片的三个不同通道。



- 我们分别按照NHWC与NCHW的顺序分别对上图进行遍历，注意遍历是按照从低维到高维进行的，即针对NHWC，遍历顺序是C轴->W轴->H轴->N轴；针对NCHW，遍历顺序是W轴->H轴->C轴->N轴。
- 上图中几个维度大小分别为N: 1, H: 3, W: 3, C: 3，不同的排布格式遍历顺序如下所示：
 - NHWC: R00,G00,B00,R01,G01,B01,R02,G02,B02,R10,G10,B10,R11,G11,B11,R12,.....
 - NCHW: R00,R01,R02,R10,R11,R12,R20,R21,R22,G00,G01,G02,G10,G11,G12,G20,.....

- 遍历时，永远是从低维
- 从上面的例子中我们看出尽管存储的数据相同，但不同的存储顺序会导致数据的访问特性不一致，因此即便进行同样的运算，相应的计算性能也会不同。
- 度向高维度进行遍历。

算子基本概念 — axis

- 不同算子包含的属性值不同，下面介绍一个常见的算子属性。

- 轴（Axis）：

- axis代表张量中维度的下标，比如张量a是一个5行6列的二维数组，即shape是(5,6)，则axis=0表示是张量中的第一维，即行。axis=1表示是张量中的第二维，即列。
- 例如张量数据：[[[1,2],[3,4]], [[5,6],[7,8]]]，Shape为(2,2,2)
 - 轴0代表第一个维度的数据：[[1,2],[3,4]]与[[5,6],[7,8]]这两个矩阵
 - 轴1代表第二个维度的数据即[1,2]、[3,4]、[5,6]、[7,8]这四个数组
 - 轴2代表第三个维度的数据即1，2，3，4，5，6，7，8这八个数。
- 轴axis可以为负数，此时表示是倒数第axis个维度。
- N维Tensor的轴有：0，1，2,...，N-1。

Shape: (4, 20, 20, 3)
↑ ↑ ↑ ↑
Axis 0 1 2 3
-4 -3 -2 -1
轴示意图

目录

1. 算子基本概念

- 算子是什么
- 算子开发复杂性

2. 昇腾AI处理器与Ascend C

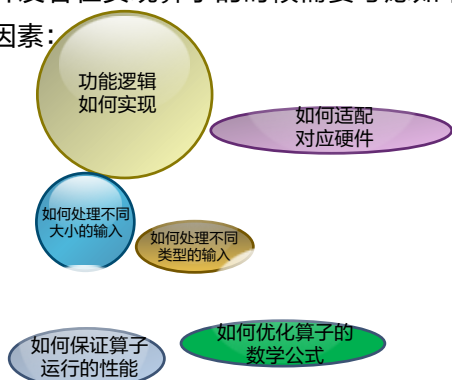
3. 算子开发基础知识

4. Vector算子开发流程

5. 算子开发工程

算子开发的复杂性

- 开发者在实现算子的时候需要考虑如下因素:



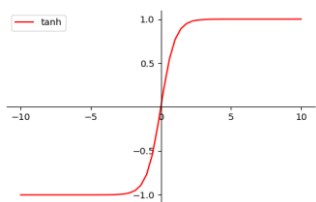
C++语言实现的卷积算子

[illegible]

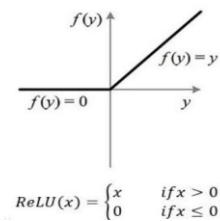
- 右边是C++语言实现的卷积算子，这里仅仅是功能逻辑的实现，未考虑硬件指令的适配。可见，一个算子的代码实现已经很复杂了，如果神经网络中的每一个算子大家都自己实现一遍，那神经网络构建的工作量有多大，难度有多高？

功能逻辑的实现——以激活函数为例

- 不同激活函数适配场景不同，性能不同，对精度的影响也不尽相同，算子设计要考虑同一功能的多种实现方式。

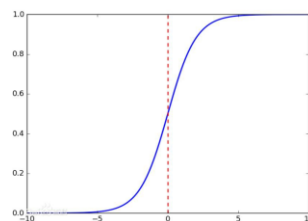


Tanh函数图像



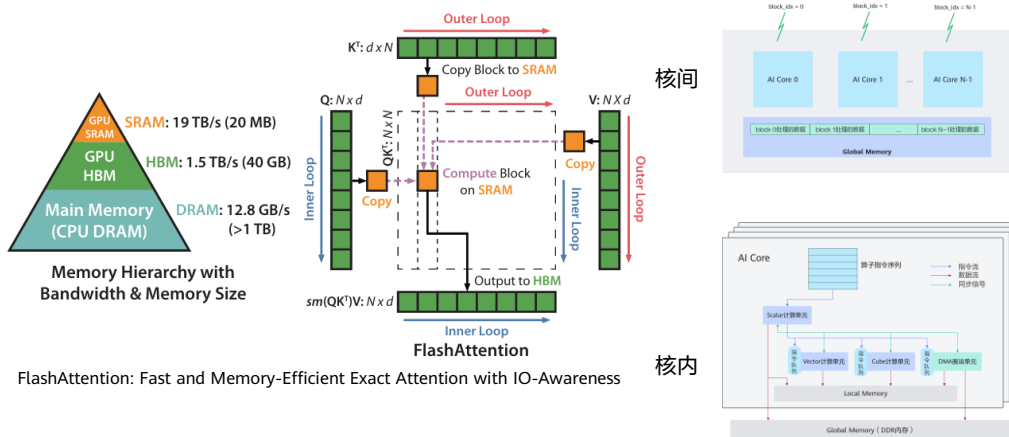
$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

ReLU函数图像



Sigmoid函数图像

与硬件的结合——以Flash Attention为例



FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

目录

1. 算子基本概念
- 2. 昇腾AI处理器与Ascend C**
3. 算子开发基础知识
4. Vector算子开发流程
5. 算子开发工程

Ascend C在CANN中的位置



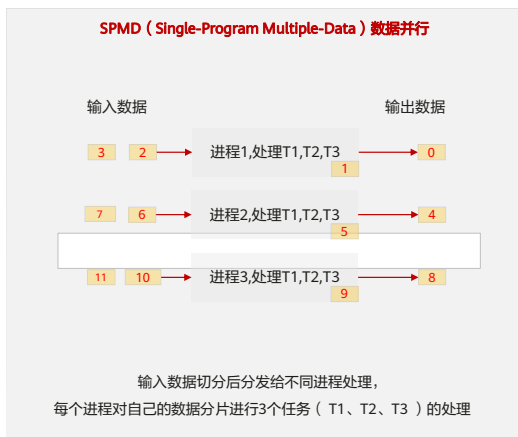
什么是Ascend C

- Ascend C是CANN针对算子开发场景推出的编程语言，通过多层接口抽象、自动并行计算、孪生调试等关键技术，极大提高算子开发效率，助力AI开发者低成本完成算子开发和模型调优部署。使用Ascend C编程语言开发的算子我们称之为Ascend C算子。
- 使用Ascend C开发自定义算子的优势：
 - C/C++原语编程，最大化匹配用户的开发习惯
 - 编程模型屏蔽硬件差异，编程范式提高开发效率
 - 多层级API封装，从简单到灵活，兼顾易用与高效
 - 孪生调试，CPU侧模拟NPU侧的行为，可优先在CPU侧调试

- Ascend C是CANN针对算子开发场景推出的编程语言，使用它开发算子会有很多优势，之前我们要使用昇腾AI处理器的AI Core对运算进行硬件加速，那么我们需要基于Python语言去开发DSL或者TIK算子。有了Ascend C之后，我们就可以基于C、C++去开发算子，也能获得AI Core的硬件加速效果。Ascend C通过编程范式描述了算子实现的固定流程，基于编程范式进行编程，可以快速搭建算子实现的代码框架，提升开发效率。为了兼顾初学者简单上手和资深开发的自由定制、深度优化需求，Ascend C提供了多层级的0-3级API，随着级别增高，API使用的自由度降低，易用性增强。Ascend C提供了孪生调试功能，可以让开发者更方便的调试定位代码问题

核间支持SPMD数据并行，充分利用AI Core多核计算资源

- Ascend C支持SPMD数据并行，将需要处理的数据拆分并分布在多个计算核心上运行，多个AI Core共享相同的指令代码



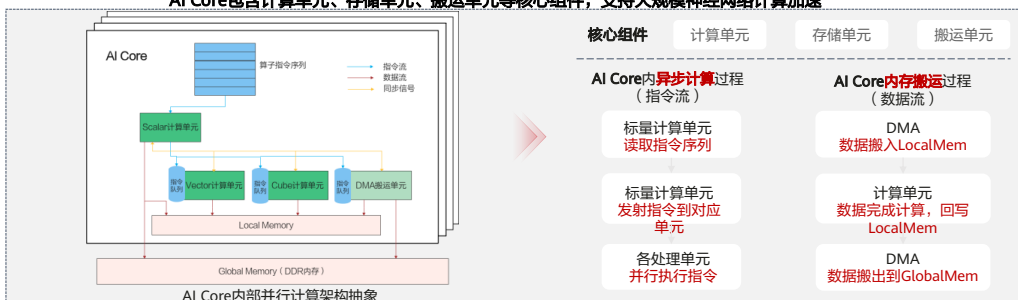
核内支持自动化流水并行，充分使能计算性能最大化

Ascend C把算子核内的处理程序，分成多个**流水任务（Stage）**：“**搬入、计算、搬出**”，通过流水任务实现数据的并行处理
以张量（Tensor）为数据载体，以队列（Queue）进行任务之间的通信与同步，以内存管理模块（Pipe）管理任务间的通信内存

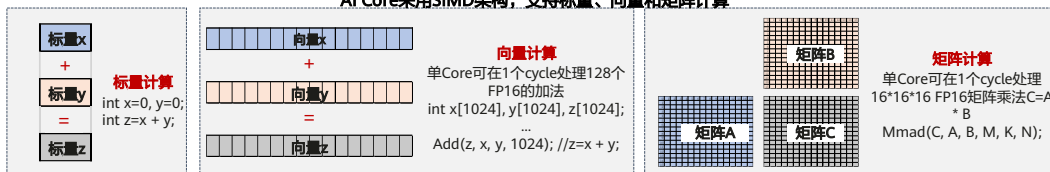


AI Core的逻辑架构抽象

AI Core包含计算单元、存储单元、搬运单元等核心组件，支持大规模神经网络计算加速



AI Core采用SIMD架构，支持标量、向量和矩阵计算

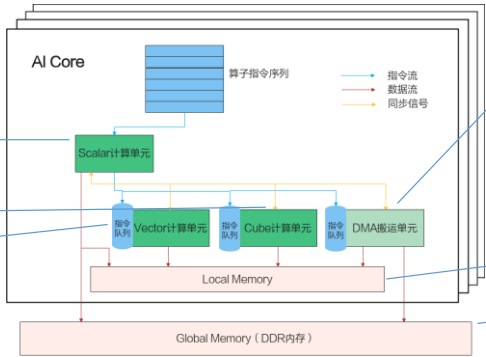


AI Core内部并行计算架构抽象

使用Ascend C编程语言开发的算子运行在AI Core上，AI Core是昇腾AI处理器中的计算核心
一个AI处理器内部有多个AI Core，AI Core中包含**计算单元**、**存储单元**、**搬运单元**等核心组件

计算单元包括了三种基础计算资源

- ① **Scalar计算单元**：执行地址计算、循环控制等标量计算工作，并把向量计算、矩阵计算、数据搬运、同步指令发射给对应单元执行
- ② **Cube计算单元**：负责执行矩阵运算
- ③ **Vector计算单元**：负责执行向量运算



AI Core内部并行计算架构抽象示意图

搬运单元负责在Global Memory和Local Memory之间搬运数据，包含搬运单元MTE2 (Memory Transfer Engine, 数据搬入单元)，MTE3 (数据搬出单元)

存储单元为AI Core的内部存储，统称为Local Memory
与此相对应，AI Core的外部存储称之为Global Memory

外部存储: GlobalTensor

GlobalTensor用来存放Global Memory（外部存储）的全局数据

定义原型

```
template <typename T> class GlobalTensor {
// 传入全局数据的指针，并手动设置一个buffer size，初始化GlobalTensor
void SetGlobalBuffer(_gm_* T* buffer, uint32_t bufferSize);
}
```

主机侧传入的全局数据指针
指向外部存储的起始地址

设置GlobalTensor所包含的类型为T的数据个数，
需自行保证不会超出实际数据的长度

```
void Init(__gm__ uint8_t* __restrict__ src_gm, __gm__ uint8_t* __restrict__ dst_gm)
{
    uint32_t dataSize = 256; //设置input_global的大小为256
    GlobalTensor<int32_t> inputGlobal; // 类型为int32_t
    // 设置源操作数在Global Memmory上的起始地址为src_gm , 所占外部存储的大小为256个int32_t
    inputGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ int32_t*>(src_gm), dataSize);
    LocalTensor<int32_t> inputLocal = inQueueX.AllocTensor<int32_t>();
    DataCopy(inoutLocal, inputGlobal, dataSize); // 将Global Memmory上的inputGlobal转贝到Local Memmory的inputLocal上
    ...
}
```

内部存储：LocalTensor

LocalTensor用于存放AI Core中Local Memory（内部存储）的数据

定义原型

```
template <typename T> class LocalTensor {  
    T GetValue(const uint32_t offset) const; // 获取 LocalTensor 中的某个值，返回 T 类型的立即数。  
    template <typename T1> void SetValue(const uint32_t offset, const T1 value) const; // 设置 LocalTensor 中的某个值。offset单位为element  
    // 获取距原LocalTensor起始地址偏移量为offset的新LocalTensor，注意offset不能超过原有LocalTensor的size大小。offset单位为element  
    LocalTensor operator[] (const uint32_t offset) const;  
    uint32_t GetSize() const; // 获取当前LocalTensor size大小  
}
```

```
// input_local针对int32_t类型的数据  
// 示例1：对input_local中第i个位置进行赋值为100  
for (int32_t i = 0; i < 256; ++i) {  
    input_local.SetValue(i, 100);  
}  
// 结果：input_local为256个100 [100 100 100 ... 100]  
  
// 示例2：获取input_local中第19个位置的数值  
auto element = input_local.GetValue(19);  
// 结果：element为100
```

```
// 示例3：获取input_local的长度，size大小为input_local有多少个element  
auto size = input_local.GetSize();  
// 结果：size大小为256  
  
// 示例4：operator[]使用方法，output_local[50]为从起始地址开始偏移量为50的新tensor  
Add(output_local[50], input_local[50], input_local2[50], 50);  
// 结果如下：  
// 输入数据(input_local): [100 100 100 ... 100] 50个int32_t  
// 输入数据(input_local2): [1 2 3 ... 50] 50个int32_t  
// 输出数据(output_local): [101 102 103 ... 150] 50个int32_t
```

逻辑位置

数据通信与同步的管理者

不同的流水任务之间存在数据依赖，需要进行数据传递。Ascend C中使用 *Queue* 队列完成任务之间的数据通信和同步，*Queue* 提供了 *EnQue*、*DeQue* 等基础API。*Queue* 队列管理NPU上不同层级的物理内存时，用一种抽象的 **逻辑位置**（*QuePosition*）来表达各个级别的存储（Storage Scope），代替了片上物理存储的概念，开发者无需感知硬件架构。

TPosition类型包括：*VECIN*、*VECCALC*、*VECOUT*、*A1*、*A2*、*B1*、*B2*、*CO1*、*CO2*。其中VECIN、VECCALC、VECOUT主要用于向量编程；A1、A2、B1、B2、CO1、CO2用于矩阵编程。

TPosition	具体含义
GM	Global Memory，对应AI Core的外部存储。
VECIN	用于向量计算，搬入数据的存放位置，在数据搬入Vector计算单元时使用此位置
VECOUT	用于向量计算，搬出数据的存放位置，在将Vector计算单元结果搬出时使用此位置
VECCALC	用于向量计算/矩阵计算，在计算需要临时变量时使用此位置
A1	用于矩阵计算，存放整块A矩阵，可类比CPU多级缓存中的二级缓存
B1	用于矩阵计算，存放整块B矩阵，可类比CPU多级缓存中的二级缓存
A2	用于矩阵计算，存放切分后的小块A矩阵，可类比CPU多级缓存中的一级缓存
B2	用于矩阵计算，存放切分后的小块B矩阵，可类比CPU多级缓存中的一级缓存
CO1	用于矩阵计算，存放小块结果C矩阵，可理解为Cube Out
CO2	用于矩阵计算，存放整块结果C矩阵，可理解为Cube Out



目录

1. 算子基本概念
2. 昇腾AI处理器与Ascend C
- 3. 算子开发基础知识**
 - Host&Device
 - 核函数
 - Hello World代码解析
4. Vector算子开发流程
5. 算子开发工程

Host&Device

- Host指与Device 相连接的X86服务器、ARM服务器，会利用Device提供的NN（Neural Network）计算能力完成任务
- Device模块指安装了昇腾AI处理器的硬件设备，利用PCIe接口与Host侧连接，提供NN计算能力

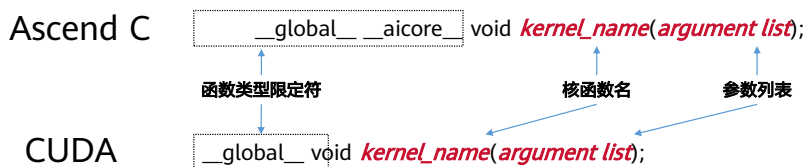


目录

1. 算子基本概念
2. 昇腾AI处理器与Ascend C
- 3. 算子开发基础知识**
 - Host&Device
 - 核函数
 - Hello World代码解析
4. Vector算子开发流程
5. 算子开发工程

什么是核函数

- 核函数（Kernel Function）是Ascend C算子设备侧的入口。Ascend C允许用户使用核函数这种C/C++函数的语法扩展来管理设备侧的运行代码，用户在核函数中实现算子逻辑的编写，例如自定义算子类及其成员函数以实现该算子的所有功能。核函数是主机侧和设备侧连接的桥梁



- 核函数是直接和设备侧执行的代码。在核函数中，需要为在一个核上执行的代码规定要进行的数据访问和计算操作，SPMD编程模型允许核函数调用时，多个核并行地执行同一个计算任务

如何编写核函数（1）

- 使用函数类型限定符
- 除了需要按照C/C++函数声明的方式定义核函数之外，还要为核函数加上额外的函数类型限定符，包含__global__和__aicore__
- 使用__global__函数类型限定符来标识它是一个核函数，可以被<<<...>>>调用；使用__aicore__函数类型限定符来标识该核函数在设备侧AI Core上执行：

– __global__ __aicore__ void kernel_name(argument list);

表1 函数类型限定符			
函数类型限定符	执行	调用	备注
__global__	在设备侧执行	由<<<...>>>来调用	必须为void返回值类型
__aicore__	在设备侧执行	仅从设备端调用	-

如何编写核函数（2）

- **使用变量类型限定符**
- 为了方便：指针入参变量统一的类型定义为 `__gm__ uint8_t*`
- 用户可统一使用 `uint8_t` 类型的指针，并在使用时转化为实际的指针类型；亦可直接传入实际的指针类型
- **规则或建议**
 - 核函数必须具有 `void` 返回类型
 - 仅支持入参为指针类型或 C/C++ 内置数据类型 (Primitive Data Types)，如： `half* s0`、`float* s1`、`int32_t c`
 - 提供了一个封装的宏 `GM_ADDR` 来避免过长的函数入参列表
 - `#define GM_ADDR __gm__ uint8_t* __restrict__`

如何调用核函数

- 核函数的调用语句是C/C++函数调用语句的一种扩展；
- 常见的C/C++函数调用方式是如下的形式：
 - `function_name(argument list);`
- 核函数使用内核调用符`<<<...>>>`这种语法形式，来规定核函数的执行配置：
 - `kernel_name<<<blockDim, nullptr, stream>>>(argument list);`
 - 注：内核调用符仅可在NPU模式下编译时调用，CPU模式下编译无法识别该符号
- `blockDim`，规定了核函数将会在几个核上执行，每个执行该核函数的核会被分配一个逻辑ID，表现为内置变量`block_idx`，编号从0开始，可为不同的逻辑核定义不同的行为，可以在算子实现中使用`GetBlockIdx()`函数来获得；
- `stream`，类型为`aclrtStream`，`stream`是一个任务队列，应用程序通过`stream`来管理任务的并行。

目录

1. 算子基本概念
2. 昇腾AI处理器与Ascend C
- 3. 算子开发基础知识**
 - Host&Device
 - 核函数
 - Hello World代码解析
4. Vector算子开发流程
5. 算子开发工程

HelloWorld——实现

核函数定义

```
#include "kernel_operator.h"
using namespace AscendC;
extern "C" __global__ __aicore__ void hello_world() {
    PRINTF("Hello World!!!\n");
}
```

核函数实现

HelloWorld——实现

```
void hello_world_do(uint32_t blockDim, void* stream) {
    hello_world<<<blockDim, nullptr, stream>>>();
}
```

核函数调用

HelloWorld——调用

```
#include "acl/acl.h"
extern void hello_world_do(uint32_t coreDim, void* stream);
```

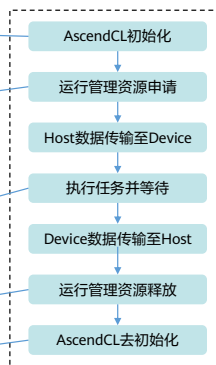
重要接口:

- aclInit
- aclrtCreateStream
- aclrtMallocHost
- aclrtMalloc
- aclrtMemcpy
- <<<...>>>
- aclrtSynchronizeStream
- aclrtFree
- aclrtFreeHost
- aclrtDestroyStream
- aclrtDestroyContext
- aclrtResetDevice
- aclrtFinalize

```
int32_t main(int argc, char const *argv[]) {
    aclInit(nullptr);
    aclrtContext context;
    int32_t deviceId = 0;
    aclrtSetDevice(deviceId);
    aclrtCreateContext(&context, deviceId);
    aclrtStream stream = nullptr;
    aclrtCreateStream(&stream);

    constexpr uint32_t blockDim = 8;
    hello_world_do(blockDim, stream);
    aclrtSynchronizeStream(stream);

    aclrtDestroyStream(stream);
    aclrtDestroyContext(context);
    aclrtResetDevice(deviceId);
    aclrtFinalize();
    return 0;
}
```



目录

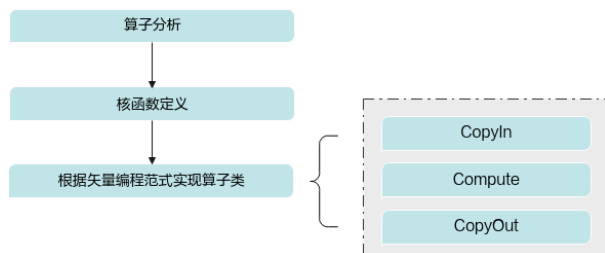
1. 算子基本概念
2. 昇腾AI处理器与Ascend C
3. 算子开发基础知识
- 4. Vector算子开发流程**
5. 算子开发工程

开发流程

算子分析：分析算子的数学表达式、输入、输出以及计算逻辑的实现，明确需要调用的Ascend C接口。

核函数定义：定义Ascend C算子入口函数。

根据编程范式实现算子类：完成核函数的内部实现。



矢量算子开发流程

以ElemWise(*Add*)算子为例，数学公式： $\vec{z} = \vec{x} + \vec{y}$ ，为简单起见，设定输入张量*x*, *y*和输出张量*z*为固定shape(8, 2048)，数据类型dtype为*half*类型，数据排布类型format为*ND*，核函数名称为*add_custom*

算子分析

算子类型 (OpType)	Add			
	name	shape	data type	format
算子输入	x	(8, 2048)	half	ND
	y	(8, 2048)	half	ND
算子输出	z	(8, 2048)	half	ND
核函数名	add_custom			

- **明确算子的数学表达式及计算逻辑**

Add算子的数学表达式为： $\vec{z} = \vec{x} + \vec{y}$ ，计算逻辑：输入数据需要先**搬入**到片上存储，然后使用计算接口完成两个**加法运算**，得到最终结果，再**搬出**到外部存储

- **明确输入和输出**

Add算子有两个输入： $\vec{x}$ 与 $\vec{y}$ ，输出为 $\vec{z}$ 。输入数据类型为**half**，输出数据类型与输入数据类型相同。输入支持固定shape(8, 2048)，输出shape与输入shape相同。输入数据排布类型为**ND**

- **确定核函数名称和参数**

自定义核函数名，如**add_custom**。根据输入输出，确定核函数有3个入参x, y, z
x, y为输入在Global Memory上的内存地址，z为输出在Global Memory上的内存地址

- **确定算子实现所需接口**

涉及内外部存储间的数据搬运，使用数据搬运接口：**DataCopy**实现

涉及矢量计算的加法操作，使用矢量双目指令：**Add**实现

使用到**LocalTensor**，使用**Queue**队列管理，会使用到**EnQue**、**DeQue**等接口。

核函数定义

在`add_custom`核函数的实现中实例化`KernelAdd`算子类，调用`Init()`函数完成内存初始化，调用`Process()`函数完成核心逻辑

注：算子类名和成员函数名无特殊要求，开发者可根据自身的C/C++编码习惯，决定核函数中的具体实现

```
// implementation of kernel function
extern "C" __global__ __aicore__ void add_custom(__gm__ uint8_t* x, __gm__ uint8_t* y, __gm__ uint8_t* z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}
```

对于核函数的调用，使用内置宏`_CCE_KT_TEST_`来标识`<<<...>>>`仅在NPU模式下才会编译到（CPU模式g++没有`<<<...>>>`的表达式），对核函数的调用进行封装，可以在封装函数中补充其他逻辑，这里仅展示对于核函数的调用

```
#ifndef _CCE_KT_TEST_
// call of kernel function
void add_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y, uint8_t* z)
{
    add_custom<<<blockDim, l2ctrl, stream>>>(x, y, z);
}
#endif
```

编程范式

Programming Paradigm

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

——Wikipedia

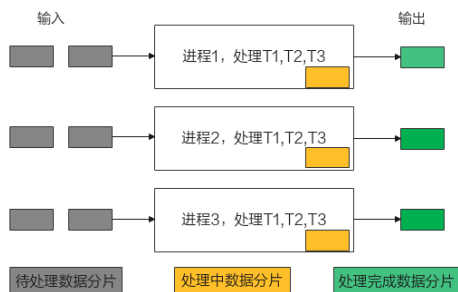
Programming paradigms are different ways or styles in which a given program or programming language can be organized. Each paradigm consists of certain structures, features, and opinions about how common programming problems should be tackled.

- 快速开发编程的固定**步骤**
- 统一代码框架的开发**捷径**
- 使用者总结出的开发**经验**
- 面向特定场景的编程**思想**
- 定制化的方法论开发**体验**

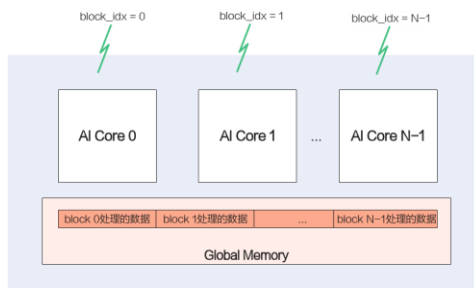
Ascend C编程范式把算子内部的处理程序，分成多个**流水任务（Stage）**，以**张量（Tensor）**为数据载体，以**队列（Queue）**进行任务之间的通信与同步，以**内存管理模块（Pipe）**管理任务间的通信内存。

编程范式—— SPMD模型

Ascend C算子编程是SPMD的编程，将需要处理的数据拆分并分布在多个计算核心上运行。多个AI Core共享相同的指令代码，每个核上的运行实例唯一的区别是`block_idx`不同。block的类似于进程，`block_idx`就是标识进程唯一性的进程ID，编程中使用函数`GetBlockIdx()`获取ID。



SPMD数据并行示意图



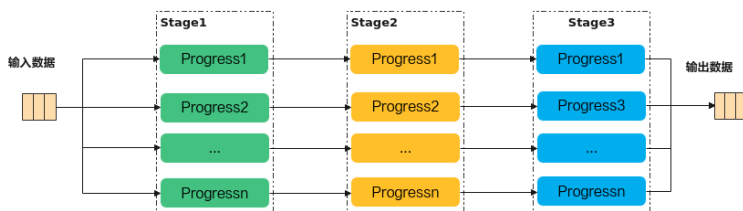
昇腾AI处理器SPMD并行计算示意图

编程范式 —— 流水任务（1）

流水任务(Stage)指的是单核处理程序中主程序调度的并行任务。

在核函数内部，可以通过流水任务实现数据的并行处理来提升性能。

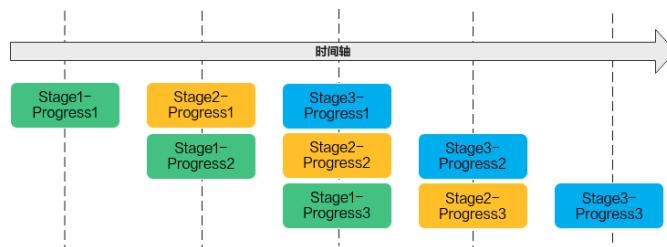
举例来说，单核处理程序的功能可以被拆分成3个流水任务：*Stage1*、*Stage2*、*Stage3*，每个任务专注于完成单一功能；需要处理的数据被切分成n片，使用*Progress1~n*表示，每个任务需要依次完成n个数据切片的处理。Stage间的箭头表达数据间的依赖关系，比如Stage1处理完Progress1之后，Stage2才能对Progress1进行处理。



任务流水示意图

编程范式 —— 流水任务（2）

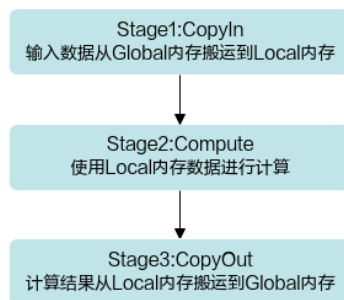
若Progress的 $n=3$ ，待处理的数据被切分成3片，对于同一片数据，Stage1、Stage2、Stage3之间的处理具有依赖关系，需要串行处理；不同的数据切片，同一时间点，可以有多个流水任务Stage在并行处理，由此达到任务并行、提升性能的目的



任务流水运行示意图

编程范式 —— 矢量编程流水任务设计

矢量算子编程范式把算子的实现流程分为3个基本任务：*CopyIn*，*Compute*，*CopyOut*
*CopyIn*负责数据搬入操作，*Compute*负责矢量计算操作，*CopyOut*负责数据搬出操作



矢量编程基本任务设计

算子类实现（1）

*CopyIn*任务：将Global Memory上的输入Tensor xGm 和 yGm 搬运至Local Memory，分别存储在 $xLocal$, $yLocal$

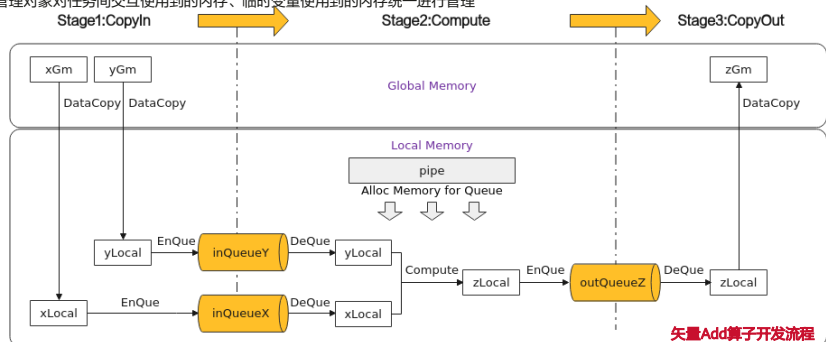
*Compute*任务：对 $xLocal$, $yLocal$ 执行加法操作，计算结果存储在 $zLocal$ 中

*CopyOut*任务：将输出数据从 $zLocal$ 搬运至Global Memory上的输出Tensor zGm 中

CopyIn, *Compute*任务间通过 *VECIN* 队列 *inQueueX*, *inQueueY* 进行通信和同步

Compute, *CopyOut*任务间通过 *VECOU* 队列 *outQueueZ* 进行通信和同步

*pipe*内存管理对象对任务间交互使用到的内存、临时变量使用到的内存统一进行管理



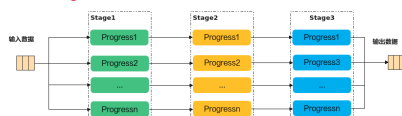
算子类实现（2）

算子类类名: *KernelAdd*

初始化函数 *Init()* 和核心处理函数 *Process()*

三个流水任务: *CopyIn()*, *Compute()*,
CopyOut()

Progress 的含义:



TQueue 模板的 *BUFFER_NUM* 的含义:

该Queue的深度, double buffer优化技巧

```
class KernelAdd {
public:
    __aicore__ inline KernelAdd() {}
    // 初始化函数, 完成内存初始化相关操作
    __aicore__ inline void Init(__gm__ uint8_t* x, __gm__ uint8_t* y, __gm__ uint8_t* z) {}
    // 核心处理函数, 实现算子逻辑, 调用私有成员函数CopyIn、Compute、CopyOut完成算子逻辑
    __aicore__ inline void Process() {}

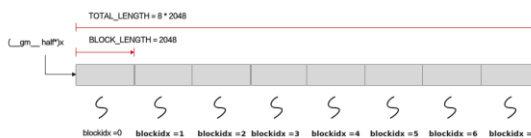
private:
    // 搬入函数, 完成CopyIn阶段的处理, 被Process函数调用
    __aicore__ inline void CopyIn(int32_t progress) {}
    // 计算函数, 完成Compute阶段的处理, 被Process函数调用
    __aicore__ inline void Compute(int32_t progress) {}
    // 搬出函数, 完成CopyOut阶段的处理, 被Process函数调用
    __aicore__ inline void CopyOut(int32_t progress) {}

private:
    // Pipe内存管理对象
    TPipe pipe;
    // 输入数据Queue队列管理对象, QuePosition为VECIN
    TQue<QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;
    // 输出数据Queue队列管理对象, QuePosition为VECOUT
    TQue<QuePosition::VECOUT, BUFFER_NUM> outQueueZ;
    // 管理输入输出Global Memory内存地址的对象, 其中xGm, yGm为输入, zGm为输出
    GlobalTensor<half> xGm, yGm, zGm;
};
```

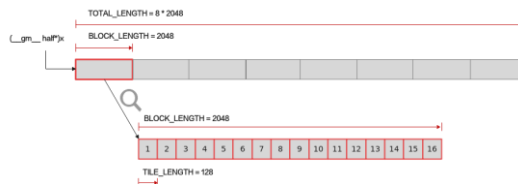
Init()函数实现

使用**多核并行计算**，需要将数据切片，获取到每个核实际需要处理的在Global Memory上的内存偏移地址

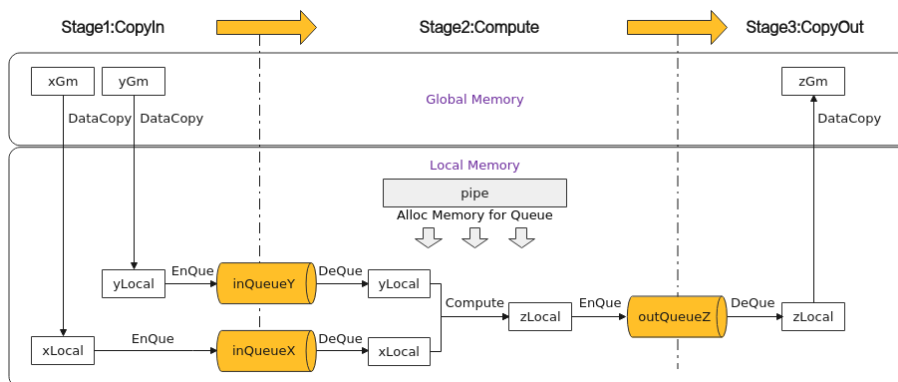
数据整体长度 **TOTAL_LENGTH** 为 $8 * 2048$ ，平均分配到8个核上运行，每个核上处理的数据大小 **BLOCK_LENGTH** 为 2048。
block_idx 为核的逻辑ID， $(_gm_half * x + block_idx * BLOCK_LENGTH)$ 即索引为 **block_idx** 的核的输入数据在Global Memory上的内存偏移地址



对于**单核处理数据**，可以进行数据切块（Tiling），将数据切分成8块。切分后的每个数据块再次切分成 **BUFFER_NUM=2** 块，可开启**double buffer**，实现流水线之间的并行
单核需要处理的 2048 个数被切分成 16 块，每块 **TILE_LENGTH=128** 个数据。**Pipe** 为 **inQueueX** 分配了 **BUFFER_NUM** 块大小为 **TILE_LENGTH * sizeof(half)** 个字节的内存块，每个内存块能容纳 **TILE_LENGTH=128** 个 **half** 类型数据



Process()函数实现



性能优化小技巧——Double Buffer机制

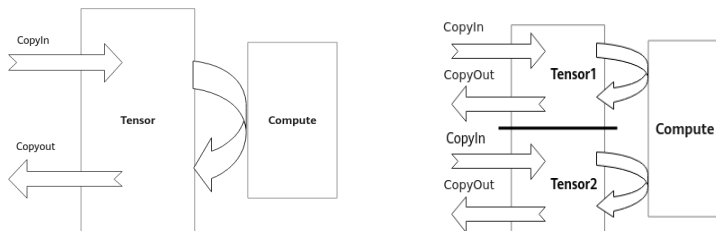
double buffer通过将**数据搬运**与**向量计算**并行执行以隐藏数据搬运时间并降低矢量指令的等待时间，最终提高矢量计算单元的利用效率

1个Tensor同一时间只能进行搬入、计算和搬出三个流水任务中的一个，其他两个流水任务涉及的硬件单元则处于Idle状态

如果将待处理的数据一分为二，比如Tensor1、Tensor2

- 当矢量计算单元对Tensor1进行Compute时，Tensor2可以执行CopyIn的任务
- 当矢量计算单元对Tensor2进行Compute时，Tensor1可以执行CopyOut的任务
- 当矢量计算单元对Tensor2进行CopyOut时，Tensor1可以执行CopyIn的任务

由此，数据的进出搬运和向量计算之间实现并行，硬件单元闲置问题得以有效缓解



向量加法z=x+y代码样例：数据通路

```

123 class KernelAdd {
124 public:
125     _siconc__ void Init(_siconc__ _gs__ uint8_t* x, _siconc__ _gs__ uint8_t* y, _siconc__ _gs__ uint8_t* z)
126     {
127         ...
128     }
129     _siconc__ void Process()
130     {
131         constexpr int32_t loopCount = fileNum;
132         for (int32_t i = 0; i < loopCount; i++) {
133             CopyIn(i);
134             Compute(i);
135             CopyOut(i);
136         }
137     }
138 private:
139     _siconc__ void CopyIn(int32_t progress)
140     {
141         LocalTensor<half> xlocal = InQueueX.AllocTensor<half>(); // 获得tensor所需内存资源
142         LocalTensor<half> ylocal = InQueueY.Dequeue<half>();
143         DataCopy(xlocal, ylocal, progress * filelength, filelength); // 复制当前片progress分片到local
144         DataCopy(ylocal, xlocal, progress * filelength, filelength);
145         InQueueY.Enqueue(xlocal);
146         InQueueX.Enqueue(ylocal);
147     }
148     _siconc__ void Compute(int32_t progress)
149     {
150         LocalTensor<half> xlocal = InQueueX.Dequeue<half>();
151         LocalTensor<half> ylocal = InQueueY.Dequeue<half>();
152         LocalTensor<half> zlocal = OutQueueZ.AllocTensor<half>();
153         Add(xlocal, ylocal, zlocal, filelength);
154         OutQueueZ.Enqueue(zlocal);
155         InQueueX.FreeTensor(xlocal); // 释放tensor
156         InQueueY.FreeTensor(ylocal);
157     }
158     _siconc__ void CopyOut(int32_t progress)
159     {
160         LocalTensor<half> zlocal = OutQueueZ.Dequeue<half>();
161         DataCopy(zlocal, zlocal, progress * filelength, filelength);
162         OutQueueZ.FreeTensor(zlocal);
163     }
164 private:
165     Pipe pipe;
166     TQueue<QueuePosition> WECIN, bufferIn; InQueueX, InQueueY; // 创建输入队列资源
167     TQueue<QueuePosition> WECOUT, bufferOut; OutQueueZ; // 创建输出队列资源
168     GlobalTensor<half> yde, ydu, ydu;
169     } = end kerneladd = 1
170 extern "C" _siconc__ _siconc__ void add_kernel(_siconc__ _gs__ uint8_t* x, _siconc__ _gs__ uint8_t* y, _siconc__ _gs__ uint8_t* z)
171 {
172     KernelAdd op;
173     op.Init(x, y, z);
174     op.Process();
175 }

```

1, 算子调度程序, 循环对每个分片
数据进行
搬入, 计算, 搬出操作

2, 完成copyin Stage的处理

3, 完成compute Stage的处理

4, 完成copyout Stage的处理

CopyIn Stage
搬入单元指令队列

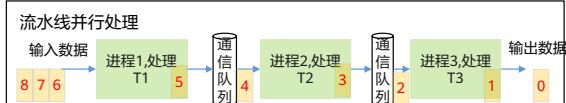
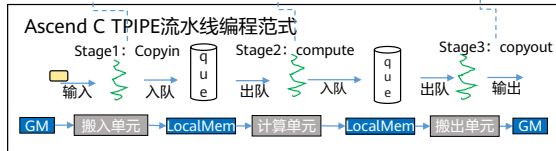
6,Y.Enqueue(y)
5,X.Enqueue(x)
4,DataCopy(y)
3,DataCopy(x)
2, y=alloc()
1, x=alloc()

Compute Stage
计算单元指令队列

13,Z.Enqueue(z)
12,Free(y)
11,Free(x)
10,Add(z,x,y)
9,z=alloc()
8,y=Y.Dequeue()
7,x=X.Dequeue()

CopyOut Stage
搬出单元指令队列

16,Free(z)
15,DataCopy(z)
14,z=Z.dequeue()



目录

1. 算子基本概念
2. 昇腾AI处理器与Ascend C
3. 算子开发基础知识
4. Vector算子开发流程
- 5. 算子开发工程**
 - 算子开发工程概述
 - 快速流程-Kernel直调工程
 - 标准流程-自定义算子工程

思考

- Kernel侧实现，Host侧实现都写完了，怎样建个工程把这些代码都组织起来？
- Host侧实现一定要写这么完整吗？能不能简单点？
- 工程建好了，怎么编译呢？
- 编译完成了，如何把算子安装到环境中？

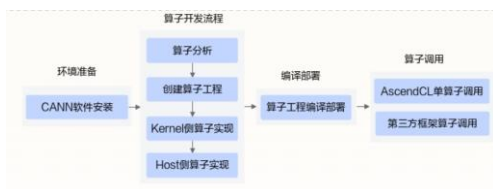
算子开发工程概述

➤ 为了更好的匹配开发者的多种使用场景，Ascend C提供如下两种算子开发工程对算子核心实现进行组织：

➤ **Kernel直调工程（快速流程）**：基于该工程，开发者完成算子核函数的开发和Tiling实现后，即可通过AscendCL运行时接口，完成算子的调用。是简单直接的开发方式，Tiling开发不受CANN框架的限制。



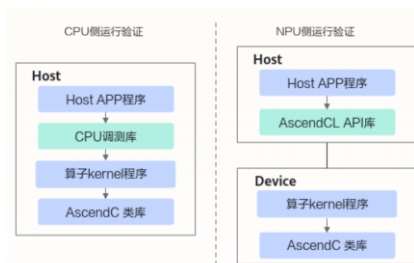
➤ **自定义算子工程（标准流程）**：按照工程创建->算子实现->编译部署->算子调用的流程完成算子开发和调用。基于工程代码框架完成算子核函数的开发和Tiling实现，通过工程编译脚本完成算子的编译部署。Tiling开发基于CANN提供的编程框架进行，后续更易于借助框架功能实现单算子调用、图模式算子调用等。



目录

1. 算子基本概念
2. 昇腾AI处理器与Ascend C
3. 算子开发基础知识
4. Vector算子开发流程
- 5. 算子开发工程**
 - 算子开发工程概述
 - 快速流程-Kernel直调工程
 - 标准流程-自定义算子工程

Kernel直调的构造



- CPU侧和NPU侧两种运行验证方法：
- CPU侧运行验证主要通过ICPU_RUN_KF CPU调测宏等CPU调测库提供的接口来完成；
- NPU侧运行验证主要通过使用<<<>>>内核调用符和AscendCL API提供的运行时接口来完成。

```
#include "data_utils.h"
#ifdef __CCE_KT_TEST__
#include "acl/acl.h"
#else
#include "tikicpulib.h"
extern "C" __global__ void add_custom(GM_ADDR x, GM_ADDR y,
GM_ADDR z);
#endif

int32_t main(int32_t argc, char* argv[])
{
    uint32_t blockDim = 8;
    size_t inputByteSize = 8 * 2048 * sizeof(uint16_t);
    size_t outputByteSize = 8 * 2048 * sizeof(uint16_t);

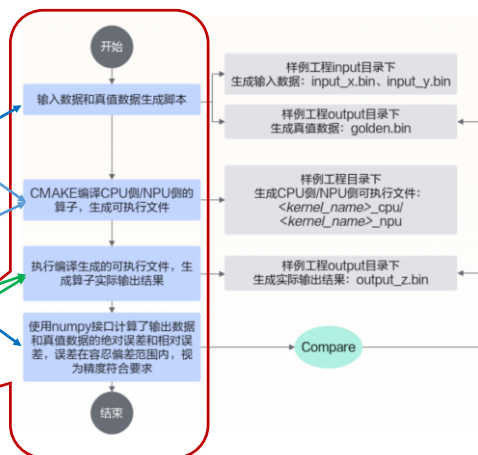
#ifdef __CCE_KT_TEST__
    // 用于CPU调试的调用程序，关键代码：
    ICPU_RUN_KF(add_custom, blockDim, x, y, z); // use this macro for cpu debug
#else
    // 用于NPU调试的调用程序，关键代码：
    add_custom_do(blockDim, nullptr, stream, xDevice, yDevice, zDevice);
#endif
    return 0;
}
```

快速流程，就是要快！

➤ 直接上例子：

- 组织CPU模式和NPU模式下编译的cmake脚本
- 为了演示计算结果与测试功能，提供了：
 - 真值数据生成脚本：*gen_data.py*
 - 计算结果校验脚本：*verify_result.py*
- 为了方便对多个源文件进行编译，提供了：
 - *CMakeLists.txt*
- 核函数源文件 *add_custom.cpp*
- 为了使得算子运行调试更加完整，提供了：
 - 读写数据文件辅助函数：*data_utils.h*
- 为了结构更加清晰，将主机侧和核函数逻辑拆分开：
 - 主机侧源文件：*main.cpp*
- 一键化执行脚本：*run.sh*

cmake
cpu_lib.cmake
npu_lib.cmake
scripts
gen_data.py
verify_result.py
CMakeLists.txt
add_custom.cpp
data_utils.h
main.cpp
run.sh



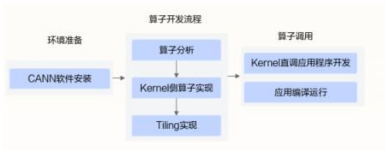
目录

1. 算子基本概念
2. 昇腾AI处理器与Ascend C
3. 算子开发基础知识
4. Vector算子开发流程
- 5. 算子开发工程**
 - 算子开发工程概述
 - 快速流程-Kernel直调工程
 - 标准流程-自定义算子工程

标准流程与快速流程对比

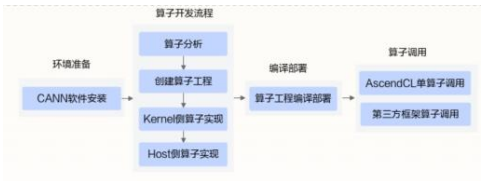
快速Ascend C算子开发流程：

- 完成算子核函数的开发
- 基于**内核调用符**方式进行算子运行验证



标准Ascend C算子开发流程：

- 完成算子核函数的开发
- 完成算子Host侧开发
- 算子工程编译部署
- 基于**ACL单算子调用**（或各种调用）方式进行算子运行验证与使用



	快速开发模式	标准开发模式
代码文件	少	多
开发时间	短	长
使用场景	单算子调用，快速验证算法逻辑	单算子网络/整网部署使用
推荐开发顺序	先	后

Add算子 —— 算子分析

- 明确算子的数学表达式及计算逻辑
 - Add算子的数学表达式为： $z = x + y$ ，计算逻辑：输入数据需要先搬入到片上存储，然后使用计算接口完成两个加法运算，得到最终结果，再搬出到外部存储
- 明确输入和输出
 - Add算子有两个输入： x 与 y ，输出为 z 。输入数据类型为half，输出数据类型与输入数据类型相同。输入支持固定shape(8, 2048)，输出shape与输入shape相同。输入数据排布类型为ND
- 确定核函数名称和参数
 - 自定义核函数名，如add_custom。根据输入输出，确定核函数有3个入参x, y, z
 - x, y为输入在Global Memory上的内存地址，z为输出在Global Memory上的内存地址

- 确定算子实现所需接口
 - 涉及内外部存储间的数据搬运，使用数据搬移接口：DataCopy实现
 - 涉及向量计算的加法操作，使用矢量双目指令：Add实现
 - 使用到LocalTensor，使用Queue队列管理，会使用到EnQue、DeQue等接口。

算子类型 (OpType)	Add			
	name	shape	data type	format
算子输入	x	(8, 2048)	half	ND
	y	(8, 2048)	half	ND
算子输出	z	(8, 2048)	half	ND
核函数名	add_custom			



Add算子 —— 创建算子工程

CANN开发套件包中提供了自定义算子工程生成工具**msopgen**，可基于算子原型定义输出算子工程：包括**算子host侧代码实现文件**、**算子kernel侧实现文件**、**算子适配插件**以及**工程编译配置文件**等。使用msopgen工具创建算子开发工程的步骤如下：

- **步骤一：**编写算子的原型定义json文件，用于生成算子开发工程。例如，AddCustom算子的json文件命名为add_custom.json，文件内容如下：

```
{
  "op": "AddCustom", 算子类型
  "input_desc": [
    {
      "name": "x", 输入1名称
      "format": "ND", 输入1数据排布格式
      "type": "float16", 输入1数据类型
      "param_type": "required" 输入1是否必选
    },
    {
      "name": "y", 输入2名称
      "format": "ND", 输入2数据排布格式
      "type": "float16", 输入2数据类型
      "param_type": "required" 输入2是否必选
    }
  ],
  "output_desc": [
    {
      "name": "z", 输出1名称
      "format": "ND", 输出1数据排布格式
      "type": "float16", 输出1数据类型
      "param_type": "required" 输出1是否必选
    }
  ],
  "attr": []
}
```

Add算子 —— 创建算子工程（续）

- **步骤二：**使用msopgen工具生成算子的开发工程。以生成AddCustom的算子工程为例，下边是控制台完整命令示例，下文仅针对关键参数进行解释：

```
$(INSTALL_DIR)/python/site_packages/bin/msopgen gen -i $HOME/sample/add_custom.json -c ai_core-<soc_version> -lan cpp -out $HOME/sample/AddCustom
```

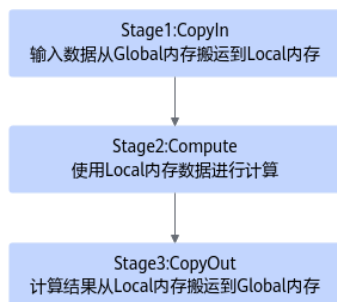
- **\$(INSTALL_DIR)**为CANN软件安装后文件存储路径，请根据实际环境进行替换。
- **-i**：指定算子原型定义文件`add_custom.json`所在路径，请根据实际情况修改。
- **-c**：`ai_core-<soc_version>`代表算子在AI Core上执行，`<soc_version>`为昇腾AI处理器的型号。
- 说明：如果无法确定具体的`<soc_version>`，则在安装昇腾AI处理器的服务器执行**npusmi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为`xxxxyy`，实际配置的`<soc_version>`值为`Ascendxxxxyy`。
- 基于同系列的AI处理器型号创建的算子工程，其基础功能（基于该工程进行算子开发、编译和部署）通用。比如`soc_version`设置为`Ascend310P1`，创建的算子工程，也可以用于开发运行于`Ascend310P3`上的算子。
- **-lan**：参数`cpp`代表算子基于Ascend C编程框架，使用C/C++编程语言开发。
- **-out**：生成文件所在路径，可配置为绝对路径或者相对路径，并且工具执行用户对路径具有可读写权限。若不配置，则默认生成在执行命令的当前路径。

创建结果：

```
AddCustom
├── build.sh          // 编译入口脚本
├── cmake
│   ├── config.cmake
│   ├── func.cmake
│   ├── init.cmake
│   └── makeself.cmake
├── util             // 算子工程编译所需脚本及公共编译文件存放目录
├── CMakeLists.txt   // 算子工程的CMakeLists.txt
├── CMakePresets.json // 编译配置项
├── framework        // 算子操作实现文件目录，算子模型文件的生成不依赖算子适配插件，无需关注
├── op_host          // host侧实现文件
│   ├── add_custom_tiling.h // 算子tiling定义文件
│   ├── add_custom.cpp     // 算子原型注册、shape推导、tiling实现等内容文件
│   └── CMakeLists.txt
├── op_kernel        // kernel侧实现文件
│   ├── CMakeLists.txt
│   └── add_custom.cpp // 算子代码实现文件
└── scripts          // 自定义算子工程打包相关脚本所在目录
```

Add算子 —— 算子实现

Kernel侧实现



Host侧实现

算子Tiling函数实现

Shape推导等函数实现

算子原型定义与注册

算子类注册

Add算子 —— Host侧算子实现（Tiling结构体）

- TilingData结构定义：
 - 通过BEGIN_TILING_DATA_DEF接口定义一个TilingData的类
 - 通过TILING_DATA_FIELD_DEF接口增加TilingData的两个字段totalLength、tileNum
 - 通过END_TILING_DATA_DEF接口结束TilingData定义。

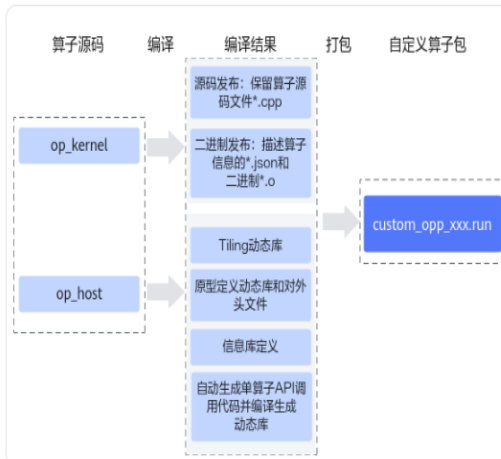
```
#ifndef ADD_CUSTOM_TILING_H
#define ADD_CUSTOM_TILING_H
#include "register/tilingdata_base.h"

namespace optiling {
BEGIN_TILING_DATA_DEF(TilingData)           // 注册一个tiling的类，以tiling的名字作为入参
    TILING_DATA_FIELD_DEF(uint32_t, totalLength); // 添加tiling字段，总计算数据量
    TILING_DATA_FIELD_DEF(uint32_t, tileNum);    // 添加tiling字段，每个核上总计算数据分块个数
END_TILING_DATA_DEF;
// 注册算子tilingdata类到对应的AddCustom算子
REGISTER_TILING_DATA_CLASS(AddCustom, TilingData)
}
#endif // ADD_CUSTOM_TILING_H
```

Add算子——算子工程编译（1）

算子kernel侧和host侧实现开发完成后，需要对算子工程进行编译，生成**自定义算子安装包*.run**，详细的编译操作包括：

- 编译Ascend C算子kernel侧代码实现文件*.cpp，分为源码发布和二进制发布两种方式。
 - **源码发布**：不对算子kernel侧实现进行编译，保留算子kernel源码文件*.cpp。该方式可以支持算子的在线编译、通过ATC模型转换的方式编译算子的场景。
 - **二进制发布**：对算子kernel侧实现进行编译，生成描述算子相关信息的json文件*.json和算子二进制文件*.o。算子调用时，如果需要直接调用算子二进制，则使用该编译方式，比如通过单算子API执行的方式完成单算子的调用，PyTorch框架中单算子调用的场景，动态网络中调用算子的场景。
- 编译Ascend C算子host侧代码实现文件*.cpp、*.h。
 - 将原型定义和shape推导实现编译成算子原型定义动态库libcust_opsproto_*.so，并生成算子原型对外接口op_proto.h。
 - 将算子信息库定义编译成信息库定义文件*.json。
 - 将Tiling实现编译成Tiling动态库liboptiling.so等。
 - 自动生成单算子API调用代码和头文件aclnn_*.h，并编译生成单算子API调用的动态库libcust_opapi.so。



Add算子——算子工程编译（2）

➤ 编译过程中最重要的配置文件是工程目录下的CMakePresets.json，编译前请确认其中配置项内容无误（下图为部分内容截取）：

```
{
  "name": "default",
  "displayName": "Default Config",
  "description": "Default build using Unix Makefiles generator",
  "generator": "Unix Makefiles",
  "binaryDir": "${sourceDir}/build_out",
  "cacheVariables": {
    "CMAKE_BUILD_TYPE": {
      "type": "STRING",
      "value": "Release"
    },
    "ENABLE_SOURCE_PACKAGE": {
      "type": "BOOL",
      "value": "True"
    },
    "ENABLE_BINARY_PACKAGE": {
      "type": "BOOL",
      "value": "True"
    },
    "ASCEND_COMPUTE_UNIT": {
      "type": "STRING",
      "value": "ascendxxx"
    },
    "ENABLE_TEST": {
      "type": "BOOL",
      "value": "True"
    },
    "vendor_name": {
```

常用配置项：

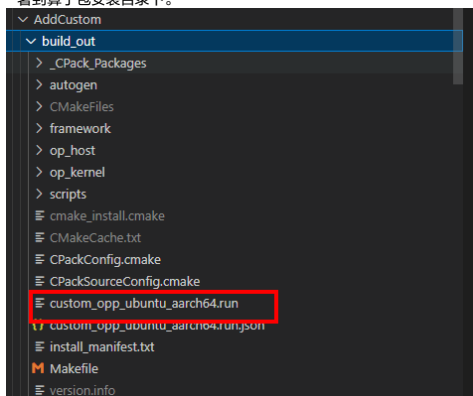
参数名称	参数描述	默认值
ASCEND_CANN_PACKAGE_PATH	CANN软件包安装路径，请根据实际情况进行修改。	"/usr/local/Ascend/latest"
CMAKE_BUILD_TYPE	编译模式选项，可配置为： • "Release"，Release版本，不包含调试信息，编译最终发布的版本。 • "Debug"，"Debug"版本，包含调试信息，便于开发者开发和调试。	"Release"
ENABLE_SOURCE_PACKAGE	是否开启源码编译	"True"
ENABLE_BINARY_PACKAGE	是否开启二进制编译	"True"
vendor_name	标识自定义算子所属厂商的名称，建议开发者自行指定所属厂商名称，避免和其他厂商提供的算子包冲突。	"customize"

- **编译操作：**
- 1.在算子工程目录下执行如下命令，进行算子工程编译。
./build.sh
 - 2.编译成功后，会在当前目录下创建build_out目录，并在build_out目录下生成自定义算子安装包**custom_opp_<target os>_<target architecture>.run**。



Add算子——算子包部署

➤ 算子包部署指执行自定义算子包的安装，算子工程的编译结果会自动部署到算子包安装目录下。



➤ 以默认安装场景为例，可查看部署后的目录结构，如下所示：



➤ 在算子工程的buile_out目录下执行：

```
./custom_opp_ubuntu_aarch64.run
```

思考题

1. （思考题）算子开发的复杂性体现在哪些方面？

- 功能实现、硬件适配、动态shape、不同数据类型、性能优化、数学公式优化

本章总结

- 本章首先介绍了算子的基本概念，然后介绍了Ascend C和昇腾AI处理器，接着介绍了Vector算子的开发和Host侧Tiling的实现，最后介绍了算子工程的不同形式。

学习推荐

- 昇腾社区
 - <https://www.hiascend.com/>
- MindSpore官网
 - <https://www.mindspore.cn/>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

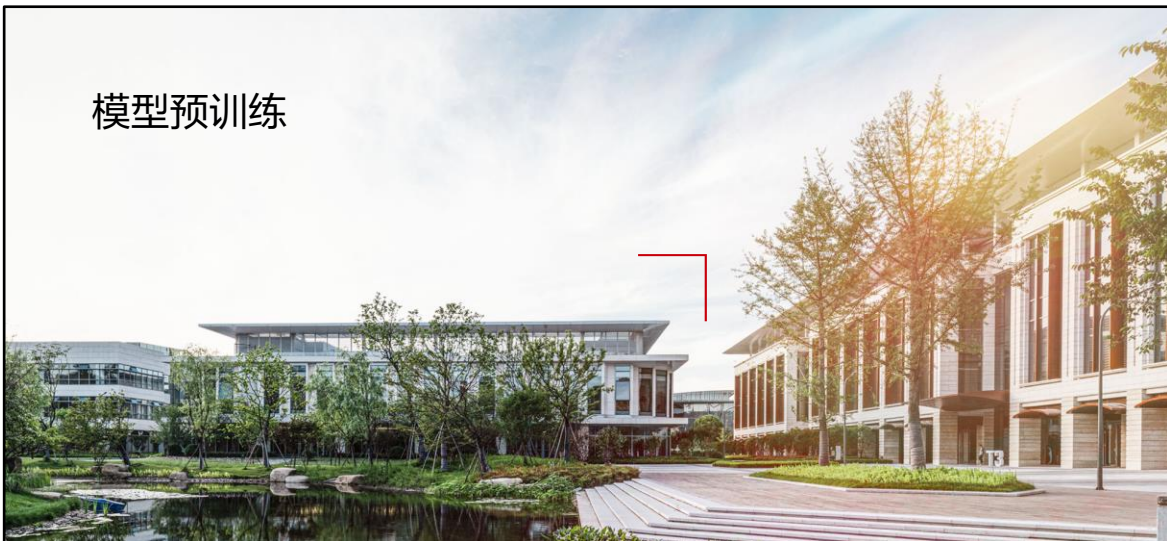
Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



模型预训练



前言

- 本章主要讲述了模型预训练相关的内容，从预训练流程解析开始，先介绍了通用的模型构建流程，然后详细介绍了LLM的构建流程，后面介绍了不同的并行算法，包括数据并行、张量并行、专家并行等，最后介绍了模型分布式并行训练的流程、优化方法以及HCCL的优势及相关概念。

目标

- 学完本课程后，您将能够：
 - 掌握预训练的概念和流程；
 - 熟悉各种并行策略；
 - 具备进行模型预训练的能力；
 - 了解分布式训练的流程；
 - 具备根据模型特点选择合适并行策略的能力；
 - 熟悉HCCL的架构和功能。

目录

1. 模型预训练流程解析

- 模型构建流程
- 参数量计算
- 模型的资源需求
- 模型训练优化方法

2. SPTD并行策略

3. 长序列并行

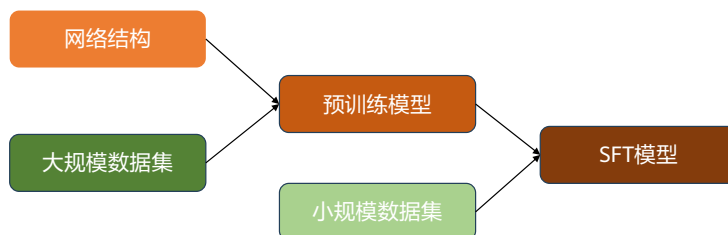
4. 专家并行

5. 分布式训练流程介绍

6. HCCL介绍

什么是预训练？

- 预训练（pre-training）是指在一个较小的、特定任务的数据集上进行微调（fine-tuning）之前，在一个大数据集上训练一个模型的过程。这个初始训练阶段允许模型从数据中学习一般的特征和表征，然后可以针对具体任务进行微调。



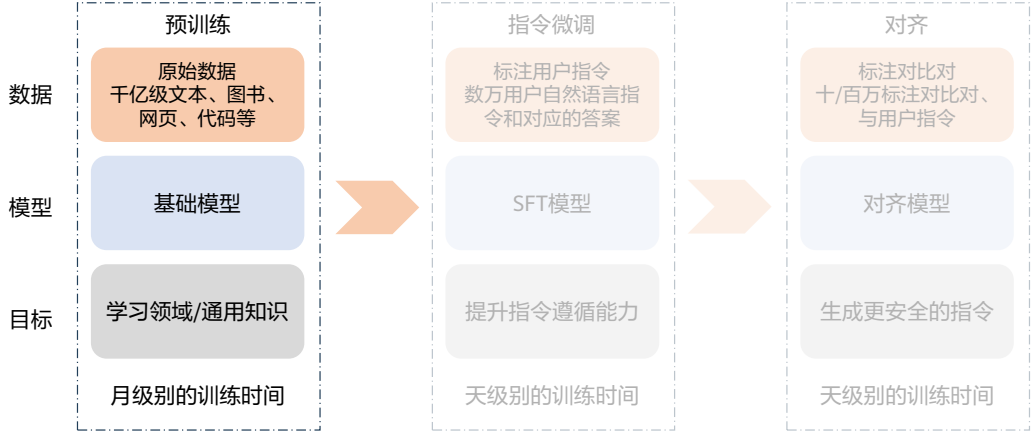
模型构建流程-通用

- 下面是通用的模型构建流程，而LLM的构建流程相对复杂，后面会详细展开。



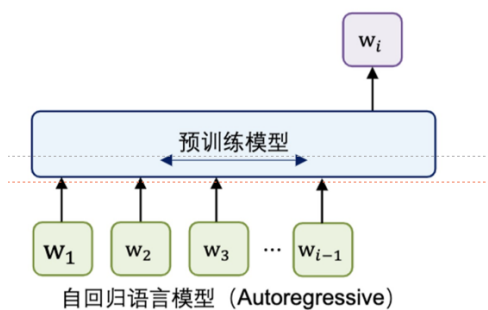
模型构建流程-LLM

- 大规模无标注数据**预训练**+指令微调+对齐



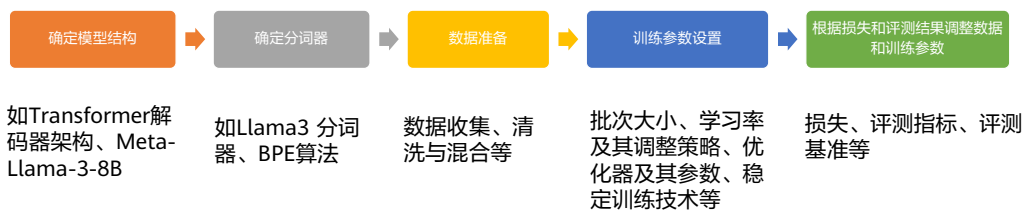
LLM预训练任务

- 语言建模（Language Modeling）
 - = Next Token Prediction 下一个词元预测
 - 形式化：给定一个词元序列 $w_1, w_2, \dots, w_{i-1}$ ，进行最大似然估计，预测下一个词元 w_i
- 语言建模对预训练的意义
 - 形式通用、便于扩展，因而可以采集到足量数据
 - 当预训练数据足够丰富时，大语言模型便能够学习到自然语言的生成规律与表达模式
 - 通过对词元更精准的预测，模型就可以更好地理解文本、建模世界语义知识



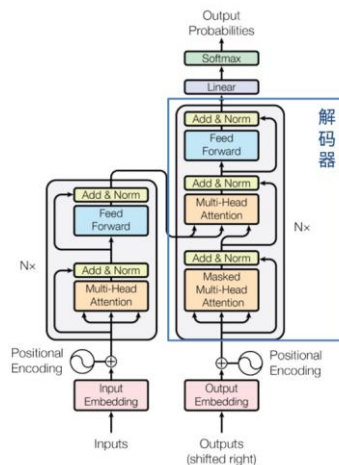
$$p(w_1, w_2, \dots, w_n) = \prod_{i=1}^n p(w_i | w_1, w_2, \dots, w_{i-1})$$

LLM预训练过程简介（以Llama3为例）



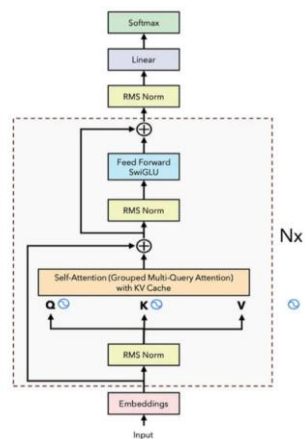
确定模型结构：Transformer解码器

- 主流LLM均采用Transformer架构
 - Transformer是由多层的**多头自注意力**（Multi-head Self-attention）模型堆叠而成的神经网络模型
- LLM均采用Transformer**解码器**架构
 - 原始的Transformer模型由编码器和解码器两个部分构成，而由于解码器架构对于任务的可扩展性，即生成式模型的语言建模任务和解码器适配，便于模型和数据的扩张



确定模型结构：Llama

- 不同大模型仅在局部有所区别，以Llama为例，相比原Transformer架构，加入了以下结构：
 - Grouped Query Attention (GQA) 分组查询注意力
 - 减少存储键和值的内存开销，提高推理效率
 - RMS Normalization 均方根标准化
 - 提高计算效率和训练的稳定性
 - SwiGLU激活函数
 - 以计算为代价优化结果



确定模型结构：Meta-Llama-3-8B（1）

- 从huggingface上下载Meta-Llama-3-8B，配置文件config.json：

```
{
  "architectures": [
    "LlamaForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 128000,
  "eos_token_id": 128001,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 14336,
  "max_position_embeddings": 8192,
  "model_type": "llama",
  "num_attention_heads": 32,
```

```
  "num_hidden_layers": 32,
  "num_key_value_heads": 8,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-05,
  "rope_scaling": null,
  "rope_theta": 500000.0,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "transformers_version": "4.40.0.dev0",
  "use_cache": true,
  "vocab_size": 128256
}
```

确定模型结构：Meta-Llama-3-8B（2）

- config.json即定义了模型参数，对关键参数解释如下：
 - architectures: LlamaForCausalLM // CausalLM即语言建模的预训练任务
 - model_type: llama, //从huggingface库加载LLaMA模型结构
 - hidden_size: 4096, //隐藏层大小
 - intermediate_size: 14336, //每个隐藏层中间层的大小
 - num_attention_heads: 32, //注意力头数
 - num_hidden_layers: 32, //隐藏层数
 - vocab_size: 128256 //词表大小

确定分词器：Llama3分词器

- 分词器 Tokenizer
 - 基于tiktoken分词器
 - 采用BPE字节对编码（Byte-Pair Encoding）算法
- 词表
 - 大小：128256
 - 包含特殊token
 - <|begin_of_text|>序列开始
 - <|end_of_text|>序列结束
 - <UNK>未知token
 - ...

数据准备

- 高质量预训练数据的准备包括数据收集、数据清洗、数据混合、分词
 - 预训练数据的数量、质量和多样性对于模型性能具有重要影响
 - 三代Llama逐步提高预训练数据的数量和质量，性能提升显著

模型	参数量	训练token数	MMLU 通用知识	GSM8K 数学推理
Llama 1[1]	7B	1.4T	35.1	11.0
Llama 2[2]	7B	2.0T	45.3	56.8
Llama 3[3]	8B	15T	69.4	84.5

数据准备-数据收集

- 常见预训练数据集
 - 通用文本数据：训练语言理解和生成能力、通用知识
 - CommonCrawl[67%]：从互联网上爬取的网页数据，多样、非结构化
 - C4[15%]：基于CommonCrawl进行了过滤，文本质量更高、更干净
 - 专业文本数据：扩充专业知识，提升任务解决能力
 - Github [4.5%]：代码
 - Wikipedia [4.5%]：知识、且结构化
 - Gutenberg and Books3[4.5%]：书籍
 - ArXiv[2.5%]：学术论文
 - Stack Exchange [2%]：高质量问答，涵盖计算机科学、化学等多样领域
 - 比例来自LLaMA-1训练过程

数据准备-数据清洗、数据混合与分词

- 数据清洗：为了提高文本质量，需要一套数据清洗流程和方法
 - 过滤
 - 质量过滤：过滤低质量文本
 - 脱敏：删除含个人隐私的数据、有毒内容等
 - HTML处理：去除tag提取文本、部分文本保留结构信息等
 - 去重：去除重复文本，包括不同文档、文档内部、不同行等层级
- 数据混合：确定预训练数据组合中不同数据源的比例
 - 知识分类：对文本进行知识分类，以确定不同类别（例如艺术和娱乐）文本的比例
 - 在小参数量下进行试验得到较优比例：经过实验，Llama3选择的数据混合比例为大约50%通用知识、25%的数学和推理、17%的代码和8%的多语言
- 分词：将原始文本分割成模型可识别和建模的词元（token）序列，作为大语言模型的输入数据

设置训练参数（1）

- 与传统神经网络的优化类似，大模型训练通常使用批次梯度下降算法来进行模型参数的更新，因此，训练参数对模型结果影响大
- 关键参数包括批次大小、学习率及其调整策略、优化器及其参数、稳定训练技术等等
- 右表展示了部分大语言模型的详细优化设置

模型	批次大小	学习率 (预热 → 峰值 → 衰减)	优化器	精度 类型	权重 衰减	梯度 裁剪
GPT-3	32K → 3.2M	预热 → 6×10^{-5} → 余弦	Adam	FP16	0.1	1.0
PanGu- α	-	2×10^{-5}	Adam	-	0.1	-
OPT	2M	预热 → 1.2×10^{-4} → 手动	AdamW	FP16	0.1	-
PuLM	1M → 4M	1×10^{-2} → 平方根倒数	Adafactor	BF16	l_2^2	1.0
BLOOM	4M	预热 → 6×10^{-5} → 余弦	Adam	BF16	0.1	1.0
MT-NLG	64K → 3.75M	预热 → 5×10^{-5} → 余弦	Adam	BF16	0.1	1.0
Gopher	3M → 6M	预热 → 4×10^{-5} → 余弦	Adam	BF16	-	1.0
Chinchilla	1.5M → 3M	预热 → 1×10^{-4} → 余弦	AdamW	BF16	-	-
Galactica	2M	预热 → 7×10^{-6} → 余弦	AdamW	-	0.1	1.0
LaMDA	256K	-	-	BF16	-	-
Jurassic-1	32k → 3.2M	预热 → 6×10^{-5}	-	-	-	-
LLaMA-2	4M	预热 → 1.5×10^{-4} → 余弦	AdamW	-	0.1	1.0
Pythia	2M	预热 → 1.4×10^{-4} → 余弦	Adam	FP16	0.01	1.0
Baichuan-2	-	预热 → 1.5×10^{-4} → 余弦	AdamW	BF16	0.1	0.5
Qwen-1.5	4M	预热 → 3×10^{-4} → 余弦	AdamW	BF16	0.1	1.0
InternLM-2	5M	预热 → 3×10^{-4} → 余弦	AdamW	-	0.1	-
Falcon	预热 → 4M	预热 → 1.25×10^{-4} → 余弦	AdamW	BF16	0.1	0.4
DeepSeek	18M	预热 → 3.2×10^{-4} → 余弦	AdamW	BF16	0.1	1.0
Yi	256K	1×10^{-5}	AdamW	BF16	0.1	1.0
YiLan	4.5M	预热 → 3×10^{-4} → 余弦	Adam	BF16	0.1	1.0
GLM-130B	0.4M → 8.25M	预热 → 8×10^{-5} → 余弦	AdamW	FP16	0.1	1.0
T5	64K	1×10^{-2} → 平方根倒数	AdaFactor	-	-	-

设置训练参数（2）

- 批次大小（Batch Size）
 - 在大模型预训练中，通常将批次大小设置为较大的数值，例如1M到4M个词元，从而提高训练的稳定性 and 吞吐量
 - 动态批次调整策略：在训练过程中逐渐增加批次大小，最终达到百万级别
- 学习率（Learning Rate）
 - 预热阶段：通常采用线性预热策略，学习率将从一个非常小的值（例如0或者 1×10^{-8} ）线性平稳增加，直到达到预设的最大阈值，一般占整个训练步骤的0.1%至0.5%
 - 衰减阶段：达到最大阈值之后学习率会开始逐渐衰减，余弦衰减，平方根倒数衰减等

设置训练参数（3）

- 优化器（Optimizer）
 - 常用Adam、AdamW：在优化中引入了三个超参数，在大模型训练中通常采用以下设置： $\beta_1=0.9$ ， $\beta_2=0.95$ 和 $\epsilon=10^{-8}$
 - β_1 、 β_2 分别表示一阶矩估计和二阶矩估计的指数衰减率。一阶矩估计和二阶矩估计用于计算梯度的移动平均值和平方的移动平均值，这两个参数的值决定了这两个移动平均值的更新速度； ϵ 防止除以0
- 稳定训练技术
 - 梯度裁剪（Gradient Clipping）：防止梯度由于累乘效应而趋于0或无穷
 - 权重衰减（Weight Decay）：对损失函数做正则化，防止过拟合

根据损失和评测结果调整数据和训练参数

- 观察损失（loss）
- 语言模型基本能力评测指标
 - 即生成文本和目标文本的拟合程度、符合人类语言使用习惯的程度（语言流畅通顺）
 - 使用困惑度（Perplexity）等指标
- 大模型各维度能力评测基准（Benchmark）

- 通用知识：MMLU、C-EVAL等
- 数学推理：GSM8K、Math
- 代码：HumanEval
- ...

		Llama 3 8B	Gemini 2 9B	Mistral 7B	Llama 3 70B	Mistral 8x22B	GPT 3.5 Turbo	Llama 3 405B	Nemotron 4 340B	GPT-4o mini	GPT-4o	Claude 3.5 Sonnet
Category	Benchmark											
General	MMLU (0-shot)	69.4	72.3	61.1	83.6	76.9	70.7	87.3	82.6	85.1	89.1	89.9
	MMLU (0-shot, CoT)	73.0	72.3 ²	60.5	86.0	79.9	69.8	88.6	78.7 ²	85.4	88.7	88.3
	MMLU-Pro (0-shot, CoT)	48.3	—	36.9	66.4	56.3	49.2	73.3	62.7	64.8	74.0	77.0
	IFEval	80.4	73.6	57.6	87.5	72.7	69.9	88.6	85.1	84.3	85.6	88.0
Code	HumanEval (0-shot)	72.6	54.3	40.2	80.5	75.6	68.0	89.0	73.2	86.6	90.2	92.0
	MBPP EvalPlus (0-shot)	72.8	71.7	49.5	86.0	78.6	82.0	88.6	72.8	83.6	87.8	90.5
Math	GSM8K (0-shot, CoT)	84.5	76.7	53.2	95.1	88.2	81.6	96.8	92.3 ⁵	94.2	96.1	96.4 ⁵
	MATH (0-shot, CoT)	51.9	44.3	13.0	68.0	54.1	43.1	73.8	41.1	64.5	76.6	71.1
Reasoning	ARC Challenge (0-shot)	83.4	87.6	74.2	94.8	88.7	83.7	96.9	94.6	96.4	96.7	96.7
	GPQA (0-shot, CoT)	32.8	—	28.8	46.7	33.3	30.8	51.1	—	41.4	53.6	59.4
Tool use	BFCL	76.1	—	60.4	84.8	—	—	88.5	86.5	88.3	80.5	90.2
	Nexus	38.5	30.0	24.7	56.7	48.5	37.2	58.7	—	50.3	56.1	45.7
Long context	ZeroSCROLLS/QuALITY	81.0	—	—	90.5	—	—	95.2	—	95.2	90.5	90.5
	InfiniteBench/En.MC	65.1	—	—	78.2	—	—	83.4	—	72.1	82.5	—
	NIH/Multi-needle	98.8	—	—	97.5	—	—	98.1	—	100.0	100.0	90.8
Multilingual	MGSM (0-shot, CoT)	68.9	53.2	29.9	86.9	71.1	51.4	91.6	—	85.9	90.5	91.6

The Llama3 Herd of Models

目录

1. 模型预训练流程解析

- 模型构建流程
- 参数量计算
- 模型的资源需求
- 模型训练优化方法

2. SPTD并行策略

3. 长序列并行

4. 专家并行

5. 分布式训练流程介绍

6. HCCL介绍

LLM参数量估计（1）

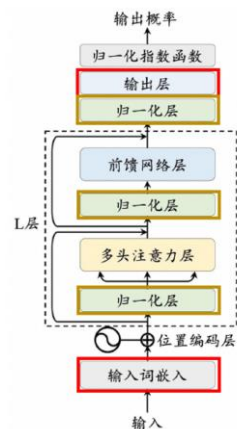
- LLama模型参数量如何计算？

- Transformer 层数 L
- 隐含层维度 H
- 中间状态维度 H'
- 注意力头数 N
- 上下文窗口长度 T
- 词表大小 V

	Feb 2023				Jul 2023				Apr 2024			
	Llama				Llama 2				Llama 3			
# params	7B	13B	33B	65B	7B	13B	34B	70B	8B			70B
# training tokens	1T	1T	1.4T	1.4T	2T	2T	2T	2T	15T			15T
hidden embed dim	4096	5120	6656	8192	4096	5120		8192	4096			8192
# attn heads	32	40	52	64	32	40		64	32			64
# attn layers	32	40	60	80	32	40		80	32			80
attention	MHA	MHA	MHA	MHA	MHA	MHA	GQA	GQA	GQA			GQA
# kv heads	32	40	52	64	32	40		8	8			8
nlp intermediate size	11008	13824	17920	22016	11008	13824		28672	14336			28672
context	2048				4096				8192			
tokenizer	BPE sentencepiece				BPE sentencepiece				BPE tiktoken			
token vocabulary	32000				32000				128256			
fine-tuned models	-				Llama-2-Chat (Jul 2023)				Llama-3-Instruct (Apr 2024)			

LLM参数量估计 (2)

- 输入词嵌入层
 - $E \in \mathbb{R}^{V \times H}$, 参数量 VH
- 输出层
 - $O = \text{softmax}(YW)$, $W \in \mathbb{R}^{H \times V}$
 - 参数量 VH
- 归一化层 (2L+1层)
- $x = \frac{x}{\text{RMS}(x)} * \gamma$, $\gamma \in \mathbb{R}^H$
- 参数量 $(2L+1)H$



LLM参数量估计 (3)

- 多头注意力层 (L层)

- $Q = XW^Q$

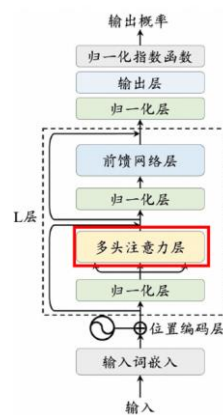
- $K = XW^K$

- $V = XW^V$

- $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V$

- $\text{MHA} = \text{Concat}(\text{head}_1, \dots, \text{head}_N)W^O$

- $W^Q, W^K, W^V, W^O \in \mathbb{R}^{H \times H}$, , 参数量共 $4LH^2$



LLM参数量估计（4）

- 前馈网络层（L层）

- $G = XW^G$

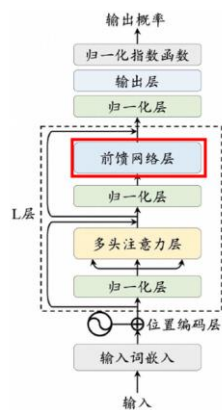
- $U = XW^U$

- $D = \text{Swish}(G) \odot U$

- $X = DW^D$

- $W^G, W^U \in \mathbb{R}^{H \times H'}, W^D \in \mathbb{R}^{H' \times H}$

- 参数量 $3LHH'$



LLM参数量估计（5）

- Llama的参数量为:

$$\underbrace{2VH}_{\text{输入输出}} + \underbrace{H}_{\text{归一化}} + \underbrace{L(4H^2)}_{\text{注意力}} + \underbrace{3HH'}_{\text{前馈网络}} + \underbrace{2H}_{\text{归一化}}$$

- LLama 7B

- $V = 32000, L = 32, H = 4096, H' = 11008$

- 参数量为6,738,415,616

- LLama 65B

- $V = 32000, L = 80, H = 8192, H' = 22016$

- 参数量为65,285,660,672

目录

1. 模型预训练流程解析

- 模型构建流程
- 参数量计算
- 模型的资源需求
- 模型训练优化方法

2. SPTD并行策略

3. 长序列并行

4. 专家并行

5. 分布式训练流程介绍

6. HCCL介绍

算力需求

- 巨大的数据量和模型参数对算力提出了极高的要求
 - 数据量：2T (Llama2) -> 15T (Llama3)
 - 模型参数：7B，8B，13B，14B，70B，405B...
 - 训练过程：预训练、指令微调、对齐
- 大模型训练的算力需求的两个关键指标
 - 显存占用：主要取决于模型参数量
 - 训练时间：主要取决于模型参数量、数据量、计算卡计算速度

显存占用-基本概念

- 参数量
 - $1\text{B}=10^9$
 - 大模型常见参数量：1B、3B、7B、8B、9B、13B、14B、34B...
- 精度
 - 单精度FP32：4字节，主要用于预训练阶段
 - 半精度FP16/BF16：2字节，主要用于微调和推理阶段
 - 整数精度INT8：1字节，量化后进行推理
 - 整数精度INT4：0.5字节，量化后进行推理

显存占用-微调阶段（1）

- 训练阶段通常比推理阶段要求更多的显存，因为涉及梯度计算和参数更新等大量计算，包括：
 - 加载模型权重
 - 训练过程计算的梯度：与模型参数的维度相同
 - 优化器参数：一些优化算法需要保存状态信息，以便在每次更新时进行调整
 - 采用AdamW优化器：每个参数占用8个字节，因此所使用的显存量是模型权重的2倍
 - 采用SGD优化器：占用显存和模型权重一样
 - 输入数据和标签：训练模型需要将输入数据和相应的标签加载到显存中。这些数据的大小取决于每个批次的样本数量以及每个样本的维度
 - 其他：中间计算、临时缓冲区、硬件和依赖库的开销等等

显存占用-微调阶段（2）

- 以参数量8B的模型为例，使用半精度FP16，粗略计算显存占用：
 - 由于模型已经在预训练阶段进行了大量的训练，微调时的数值误差对模型的影响通常较小，常常选择16位精度训练
- 1. 加载模型参数16G $\approx$ 8B $\times$ 2字节
- 2. 训练过程计算的梯度16GB（与模型参数的维度相同）
- 3. 优化器参数32GB（AdamW 2倍模型参数）
- 4. 数据占用显存（设置batch size为16）：
 - 以LLama-3-8B的架构为例: hidden_size=4096, intermediate_size=14336, num_hidden_layers=32, context.length=4096
 - 所以每个样本大小: $(4096+14336) \times 4096 \times 32 \text{ 字节} = 2304 \text{ MB}$
- 综上，至少占用显存: 16GB+16GB+32GB+2304M $\times$ 16 $\approx$ 100GB

显存占用-预训练阶段

- 相比微调，预训练阶段要求更多的显存
 - 模型的权重是从头开始计算，因此通常选择高精度（如FP32）进行训练
 - 预训练数据量级远大于微调阶段，因此选择更大的批次大小(batch size)
- FP32预训练至少是FP16微调占用显存的两倍以上（不考虑混合精度训练场景）
 - 以参数量8B的模型为例，预训练至少占用显存 $200\text{GB}=100\text{GB}\times 2$
- 补充说明
 - 实际影响显存占用的因素还有很多，所以只能粗略估计数量级
 - 显存占用计算参考：<https://huggingface.co/spaces/hf-accelerate/model-memory-usage>

训练时间-决定因素

- 模型参数量和训练总token数决定了训练Transformer模型需要的计算量
- 给定硬件GPU/NPU类型的情况下，可以估计所需要的训练时间
- 在给定训练token数、硬件环境配置的情况下，训练Transformer模型的计算时间为：

$$\text{训练时间} \approx \frac{\text{系数} * \text{token数} * \text{模型参数量}}{\text{计算卡数量} * \text{计算卡峰值FLOPS} * \text{计算卡利用率}}$$

训练时间-估算公式（1）

- 在给定训练token数、硬件环境配置的情况下，训练Transformer模型的计算时间为:

$$\text{训练时间} \approx \frac{\text{系数} * \text{token数} * \text{模型参数量}}{\text{计算卡数量} * \text{计算卡峰值FLOPS} * \text{计算卡利用率}}$$

- FLOPS（Floating-point operations per second）：每秒浮点运算数
- GPU利用率
 - 计算端到端训练的GPU利用率时，不仅要考虑前向传递和后向传递的计算时间，还要考虑CPU加载数据、优化器更新、多卡通信和记录日志的时间
 - GPU利用率一般在0.3-0.55之间

训练时间-估算公式（2）

- 在给定训练token数、硬件环境配置的情况下，训练Transformer模型的计算时间为:

$$\text{训练时间} \approx \frac{\text{系数} \times \text{token数} \times \text{模型参数量}}{\text{计算卡数量} \times \text{计算卡峰值FLOPS} \times \text{计算卡利用率}}$$

- 系数为6或8
 - 对于每个token，每个模型参数，进行2次浮点数计算
 - 若使用激活重计算技术来减少中间激活显存，则需要进行一次额外的前向传递，因此前向传递+后向传递+激活重计算的系数=1+2+1= 4
 - 若不使用，则1+2=3
 - 因此一次训练迭代中，需要进行2×4或3=8或6次浮点数运算

训练时间估算示例

- 以LLama-65B为例
 - 在2048张80GB显存的A100上，在1.4T token的数据上训练了65B参数量的模型
 - 80GB显存A100的峰值性能为624TFLOPS，设GPU利用率为0.3
- 则所需要的训练时间为21天，与论文中的实际训练时间恰好对应

$$21\text{天} \approx \frac{8 \times 1.4\text{T} \times 65\text{B}}{2048 \times 624\text{TFLOPs} \times 0.3} \text{秒} = \frac{8 \times 1.4 \times 10^{12} \times 65 \times 10^9}{2048 \times 624 \times 10^{12} \times 0.3} \text{秒}$$

大模型训练时间统计

- 近年来大语言模型的统计数据，包括预训练数据规模（以词元数量或存储大小表示）和硬件条件等
- 本表仅列举有公开论文介绍技术细节的模型，其中“发布时间”表示相应论文或技术报告正式发布的日期

模型	发布时间	参数量 (B)	预训练数据规模 (词元数 Tokens)	硬件 (GPUs/TPUs)	训练时间
CodeGeeX	2022.09	13	0.85T	1536 Ascend 训练 芯片	60天
GLM	2022.10	130	0.4T	768 A100 (40G)	60天
BLOOM	2022.11	176	0.366T	384 A100 (80G)	105天
LLaMA	2023.02	65	1.4T	2048 A100 (80G)	21天
LaMDA	2022.01	137	0.768T	1024 TPU v3	58天

目录

1. 模型预训练流程解析

- 模型构建流程
- 参数量计算
- 模型的资源需求
- 模型训练优化方法

2. SPTD并行策略

3. 长序列并行

4. 专家并行

5. 分布式训练流程介绍

6. HCCL介绍

优化参数设置

- 现有大语言模型的预训练优化设置

模型	批次大小	学习率 (预热 → 峰值 → 衰减)	优化器	精度 类型	权重 衰减	梯度 裁剪
GPT-3	32K → 3.2M	预热 → 6×10^{-5} → 余弦	Adam	FP16	0.1	1.0
OPT	2M	预热 → 1.2×10^{-4} → 手动	AdamW	FP16	0.1	-
PaLM	1M → 4M	1×10^{-2} → 平方根倒数	Adafactor	BF16	$1r^2$	1.0
BLOOM	4M	预热 → 6×10^{-5} → 余弦	Adam	BF16	0.1	1.0
LLaMA-2	4M	预热 → 1.5×10^{-4} → 余弦	AdamW	-	0.1	1.0
Baichuan-2	-	预热 → 1.5×10^{-4} → 余弦	AdamW	BF16	0.1	0.5
Qwen-1.5	4M	预热 → 3×10^{-4} → 余弦	AdamW	BF16	0.1	1.0
InternLM-2	5M	预热 → 3×10^{-4} → 余弦	AdamW	-	0.1	-
Falcon	预热 → 4M	预热 → 1.25×10^{-4} → 余弦	AdamW	BF16	0.1	0.4
DeepSeek	18M	预热 → 3.2×10^{-4} → 余弦	AdamW	BF16	0.1	1.0
YuLan	4.5M	预热 → 3×10^{-4} → 余弦	Adam	BF16	0.1	1.0
GLM-130B	0.4M → 8.25M	预热 → 8×10^{-5} → 余弦	AdamW	FP16	0.1	1.0

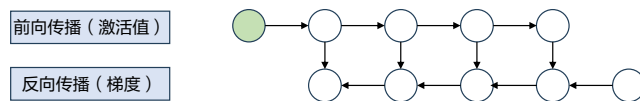
动态批次调整

学习率预热与衰减

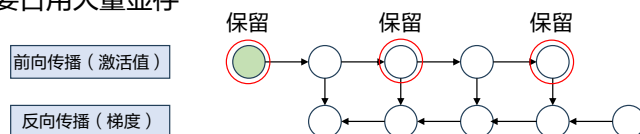
混合精度训练

激活重计算

- 激活值是前向传播的结果，需要在反向传播时参与梯度计算



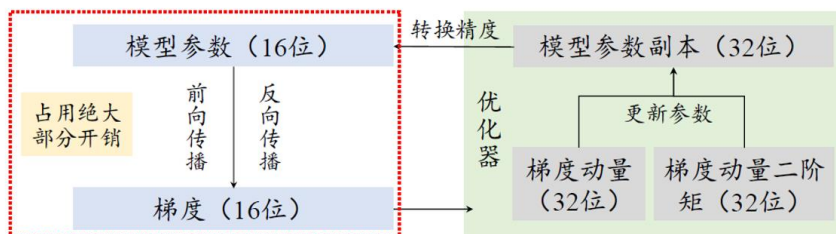
- 保存激活值需要占用大量显存



- 激活重计算：保留部分激活值，反向传播时重新计算其他激活值

混合精度训练

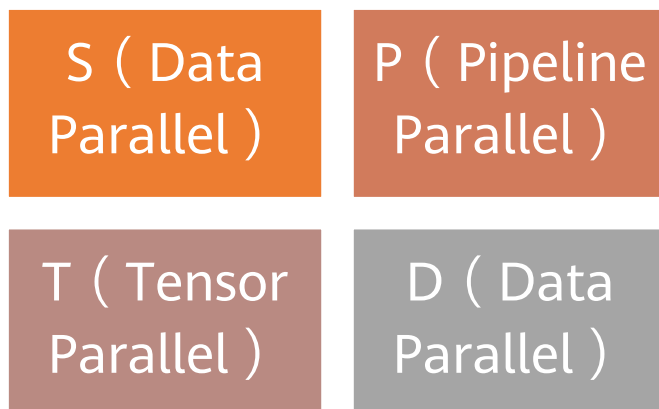
- 同时使用不同精度的浮点数进行计算（如FP32和FP16）
- 显存减半、效率翻倍
 - DeepSeek-V3 使用FP32和FP8混合训练，进一步减少密集型计算开销



目录

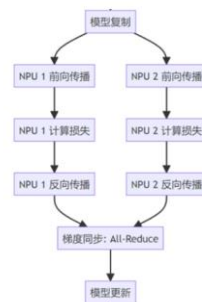
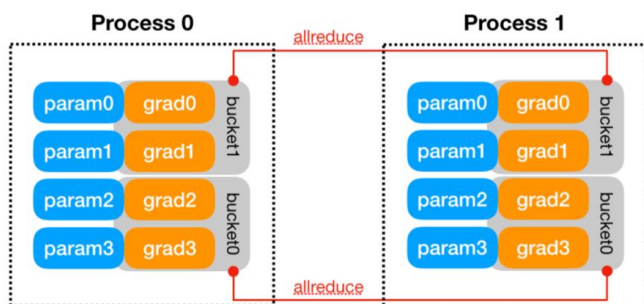
1. 模型预训练流程解析
- 2. SPTD并行策略**
3. 长序列并行
4. 专家并行
5. 分布式训练流程介绍
6. HCCL介绍

SPTD并行策略



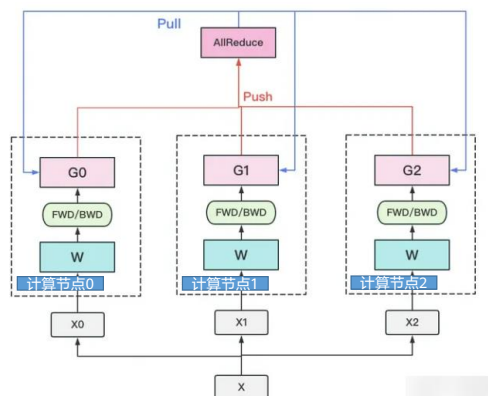
DP（数据并行）

- 数据并行：将原本在单一device上进行的数据训练过程，扩展到多个device并行计算，以达到理想情况下设备数量倍率加速效果的并行算法。



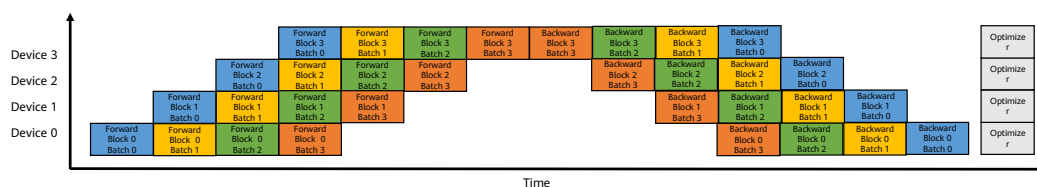
数据并行流程

- 若干个**计算节点**，如图中节点0~节点2；1块**梯度收集节点**，如图中AllReduce操作所在节点。
- 在每个计算节点上都拷贝一份完整的模型参数。
- 把一份数据X（例如一个batch）均匀分给不同的计算节点。
- 每块计算节点做一轮FWD和BWD后，算得一份梯度G。
- 每块计算节点将自己的梯度**push**给梯度收集节点，做聚合操作。这里的聚合操作一般指**梯度累加**。当然也支持用户自定义。
- 梯度收集节点聚合完毕后，计算节点从它那**pull**下完整的梯度结果，用于更新模型参数W。更新完毕后，计算节点上的模型参数依然保持一致。
- **聚合再下发梯度的操作，称为AllReduce。**



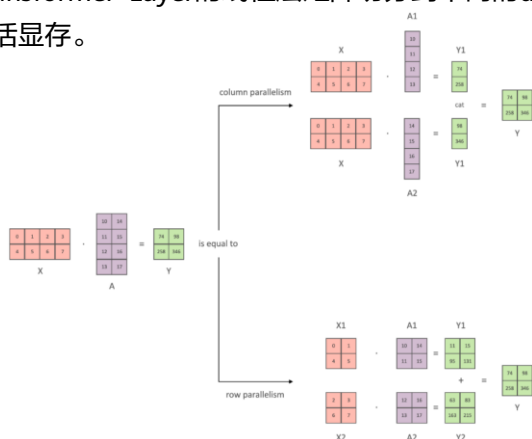
PP（流水线并行）

- 流水并行：将Transformer层划分到不同的device上，形成数据在不同的流水阶段间完成前反向训练，从而减小模型的静态显存；
- 如图所示，将4层MatMul的网络切分成4个阶段，分布到4台设备上；
- 正向计算时，每台机器在算完本台机器上的MatMul之后将结果通过通信算子发送（Send）给下一台机器，同时，下一台机器通过通信算子接收（Receive）上一台机器的MatMul结果，同时开始计算本台机器上的MatMul；
- 反向计算时，最后一台机器的梯度算完之后，将结果发送给上一台机器，同时，上一台机器接收最后一台机器的梯度结果，并开始计算本台机器的反向。



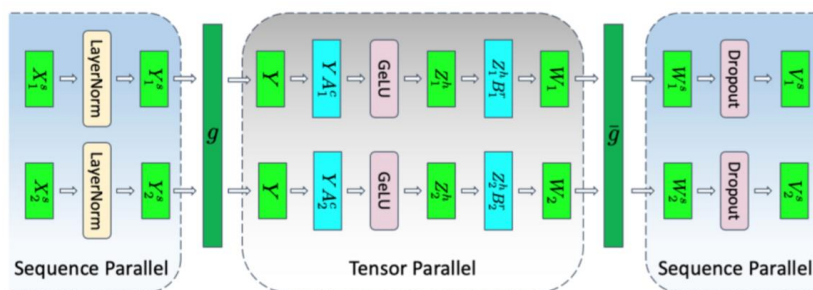
TP（张量并行）

- 张量并行：将Transformer Layer的线性层矩阵切分到不同的device上，从而减少模型的静态显存以及激活显存。



SP (序列并行)

- 序列并行：协同张量并行，将Transformer Layer的Norm与 Dropout由张量并行进行切分，进一步减小模型显存。



TP+SP

- 张量并行一次前反向总共有4次all-reduce操作，序列并行一次前反向共有4次all-gather和4次reduce-scatter操作;
- ring all-reduce执行过程中有两步，先是一个reduce-scatter然后跟着一个all-gather;
- Sequence并行相比没有引入更多的通信代价。

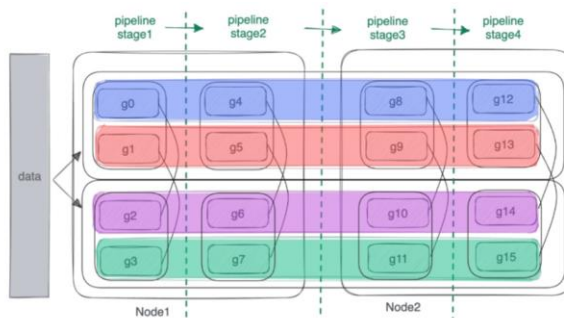
SPTD切分

- 假设存在2 nodes，16 ranks，则world_size为16，并设模型层数为16；
- 如图所示，切分方式为TP2、PP4、VP2、DP2。

▫ $\text{num_tp-groups} = 16 / 2 = 8$

▫ $\text{num_pp-groups} = 16 / 4 = 4$

▫ $\text{num_dp-groups} = 16 / 2 = 8$



目录

1. 模型预训练流程解析
2. SPTD并行策略
- 3. 长序列并行**
4. 专家并行
5. 分布式训练流程介绍
6. HCCL介绍

长序列并行算法一览

Context Parallel

优点：降低attention计算阶段的动态显存占用

缺点：额外的通信耗时

RingAttention 优点：切分S轴，降低attention计算阶段的动态显存占用
缺点：额外的SendRecv通信，序列不够长则开销较大

Ulysses 优点：切分Dim轴，降低attention计算阶段的动态显存占用，性能稳定
缺点：GQA或Head不够则无法使用，或者需要额外复制

Hybrid 优点：兼具两者优点，综合能力强
缺点：在Ring/Ulysses的优势场景下性能不如Ring/Ulysses

Ulysses

- 原理：通过 all-to-all 操作，在 attention 操作前，由 Q/K/V 在序列维度上的切分(B, N/P, d)，转成Attention head (B, N, d/P)上的切分，从而确保attention操作在每个 device上独立进行。

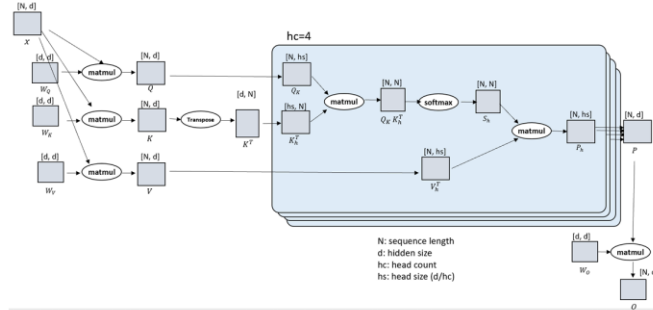
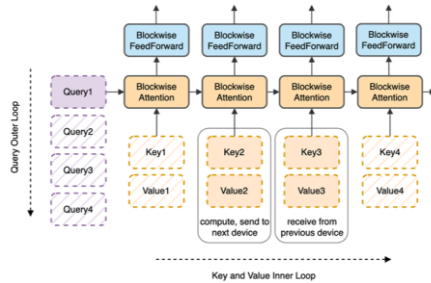


Figure 1: Multi-head attention Transformer

DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models

RingAttention

- 最初尤洋团队提出 RingSelfAttention (RSA)，实现了在序列维度进行切分，并和TP，PP，DP等并行方式解耦。后续 Blockwise Parallel Transformer (BPT) 优化了 RSA方案的性能，解决通算不可掩盖的问题，另外实现分块独立 Attention 计算。
- 基于BPT的思想，Liu等人进一步将 BPT的两层循环计算通过 ring p2p的形式进行了并行切分，使得上述两层循环计算得以并行化，形成当前MindSpeed的RingAttention长序列方案。



Ring Attention with Blockwise Transformers for Near-Infinite Context

混合长序列并行（1）

- Hybrid Context Parallel: 把原本Ulysses中本地的FA，替换成ringattention。同时在并行初始化部分，把CP维度细分为Ulysses和ring维度，且Ulysses维度设置为更低（all-to-all通信对带宽要求更高）：

Algorithm 1 Unified Single-Process Attention Implementation

```
1: function UNIFIED_SP_ATTENTION(ulysses_pg, ring_pg, Q, K, V, scatter_idx, gather_idx)
2:   Q ← AllToAll4DQ(scatter_idx, gather_idx, group = ulysses_pg)
3:   K ← AllToAll4DK(scatter_idx, gather_idx, group = ulysses_pg)
4:   V ← AllToAll4DV(scatter_idx, gather_idx, group = ulysses_pg)
5:   O ← LoadBalance-RingAttention(Q, K, V, group = ring_pg)
6:   O ← AllToAll4DUlysses_pg, O, gather_idx, scatter_idx, group = ulysses_pg
7:   return O
8: end function
```

A Unified Sequence Parallelism Approach for Long Context Generative AI

混合长序列并行（2）

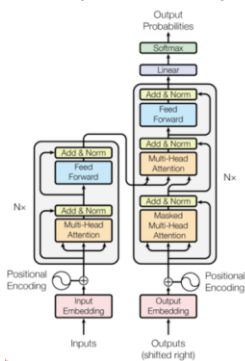
- Ulysses优劣势：优点是充分利用all-to-all通信带宽，attention计算的矩阵较大，充分利用计算带宽。缺点是CP并行维度难扩展，head_size要能被TP*CP整除，部分场景如GQA、MQA甚至无法使用Ulysses，另外跨机All2All带宽也需考虑；
- Ring优劣势：通信和计算难掩盖，每个block分配的序列长度要大于算力通信带宽比，即 $c \geq F/B$ ，才能不引入额外通信开销，实际场景中，考虑到算力利用率因素，对序列长度的要求会更严格；
- Hybrid方案能够弥补两种方案各自的缺陷：在head size较小的场景下，混合方案可以设置较大的ring维度，增大CP；同时，在ring attention前做了all-to-all，可以获取更大的序列长度，非常有利于Ring attention的通算掩盖。

目录

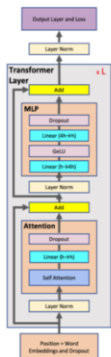
1. 模型预训练流程解析
2. SPTD并行策略
3. 长序列并行
- 4. 专家并行**
 - MoE模型
 - DeepSeek MoE
5. 分布式训练流程介绍
6. HCCL介绍

LLM结构演进

典型的Transformer结构
(Encoder+Decoder)



千亿稠密大模型
(Decoder-Only)

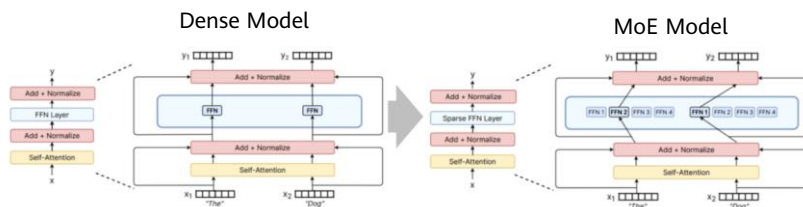


万亿稀疏大模型 (MoE)



MoE (Mixture-of-Experts) vs Dense (1)

- MoE是一种，能够显著增加模型容量、模型能力，却不需要像稠密模型（Dense）一样，成比例增加计算量的模型方案
 - MoE 模型在原有MLP 的位置使用了 MoE 结构去实现，其他模块保持一致;
 - 每个 token 经过一个 Gating 门控，根据一定的 Routing 路由策略，被路由到对应专家结构 (FFN) 上进行处理



MoE (Mixture-of-Experts) vs Dense (2)

- 训练速度：从输入的一个sample视角来说，无需经过全量模型去计算，而只是激活一部分小模型进行计算这样显著地节省了计算资源；

Model	Active Params	MMLU	HellaS	WinoG	PIQA	Arc-e	Arc-c	NQ	TriQA	HumanE	MBPP	Math	GSM8K
LLaMA 2 7B	7B	44.4%	77.1%	69.5%	77.9%	68.7%	43.2%	17.5%	56.6%	11.6%	26.1%	3.9%	16.0%
LLaMA 2 13B	13B	55.6%	80.7%	72.9%	80.8%	75.2%	48.8%	16.7%	64.0%	18.9%	35.4%	6.0%	34.3%
LLaMA 1 33B	33B	56.8%	83.7%	76.2%	82.2%	79.6%	54.4%	24.1%	68.5%	25.0%	40.9%	8.4%	44.1%
LLaMA 2 70B	70B	69.9%	85.4%	80.4%	82.6%	79.9%	56.5%	25.4%	73.0%	29.3%	49.8%	13.8%	69.6%
Mistral 7B	7B	62.5%	81.0%	74.2%	82.2%	80.5%	54.9%	23.2%	62.5%	26.2%	50.2%	12.7%	50.0%
Mixtral 8x7B	13B	70.6%	84.4%	77.2%	83.6%	83.1%	59.7%	30.6%	71.5%	40.2%	60.7%	28.4%	74.4%

MoE模型介绍

Mixtral 8*7B 2024

Nvidia Megatron-LM支持的
首个MoE大模型

DeepSeek-V2 2024

国产优秀MoE模型，MoE
中引入share experts

Grok1 2024

xAI开源的，当时最大参
数量的MoE模型

DeepSeek-V3 2025

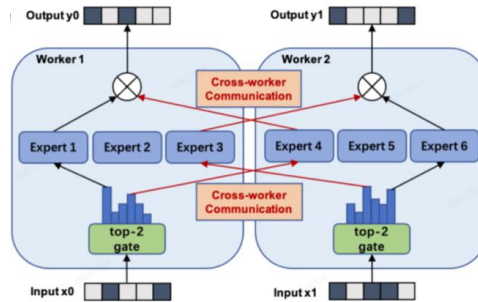
国产大模型引领世界大模
型潮流

MoE Layer结构剖析

- Routing：通过Gating，将每个token路由到对应专家，是解决Expert负载不均衡的关键；
- Permutation：调用token_permutation做token顺序重排，结合专家并行，将token传输到对应专家所在的device上。对专家容量上有Drop和Dropless策略之分；
- Expert Computation：各个EP Rank调用Experts Module完成各自的MLP计算；
- Un-permutation：调用token_unpermutation做token顺序反重排，结合专家并行，将token传回原本的device上，重排成进入专家前的顺序，然后加权求和输出。

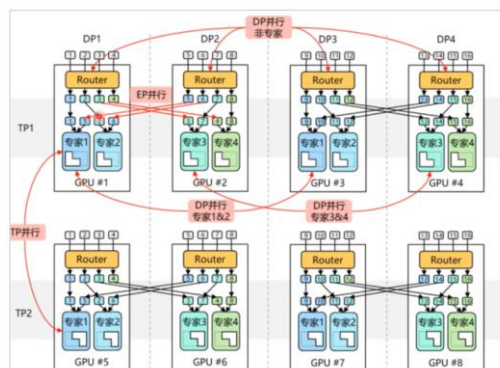
MoE Layer结构剖析：Permutation/UnPermutation

- 专家并行（EP）场景：一个EP组内，各个Rank只包含部分专家，因此存在跨卡通信，根据Router的输出，把Token路由到对应卡上，之后各个卡的专家才能够完成MLP计算。



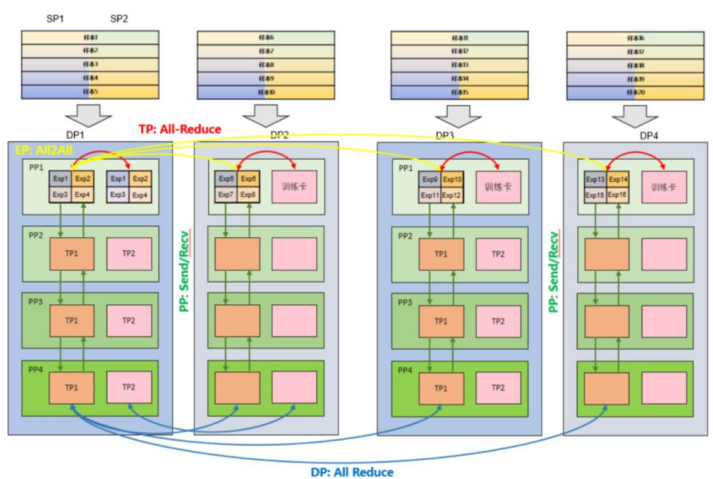
MoE分布式并行复合场景：EP+TP+DP

- 专家并行借用DP通信域，在 MoE Layer上属于DP的子集，因此专家并行度为DP的因数Router会将每张卡上输入token路由到所有专家，因此在EP的不同卡之间引入了AlltoAll或AllGather通信。



MoE模型结构：EP+TP+DP+PP并行策略

- 所谓EP+TP+DP等等，都特指 MoE层的并行方式；
- 对non-MoE的Transformer结构，其并行实现方式和稠密模型相同；
- 流水并行（PP）只关心层间激活值的传递，PP的引入并不会影响MoE层的实现。



LLM并行技术的通信量分析

- 专家并行不影响每个专家FFN的shape，故对cube计算利用率和精度没有影响的同时，显著降低显存压力。

并行方式	描述	通信操作	(每卡)通信量	通信域	计算耗时	对通信的需求
张量并行 (TP)	Transformer 层内 Matmul 切分	AllReduce	GB	节点内	秒级 (不可掩盖)	节点内高速互联, 对节点间通信无诉求
流水并行 (PP)	按 Transformer 层间切分	Send/Recv	MB	节点间	秒级 (可流水掩盖)	通信量和次数少 (stage数), 对节点间通信诉求低
专家并行 (EP)	横向扩展多个 Expert (专家), 互相交换数据	All2All	GB	节点间	秒级 (不可掩盖)	跨节点高带宽 All2All
数据并行 (DP)	训练数据分组	AllReduce	GB	节点间	数十秒级 (可与计算掩盖)	跨节点高吞吐分级 AllReduce

目录

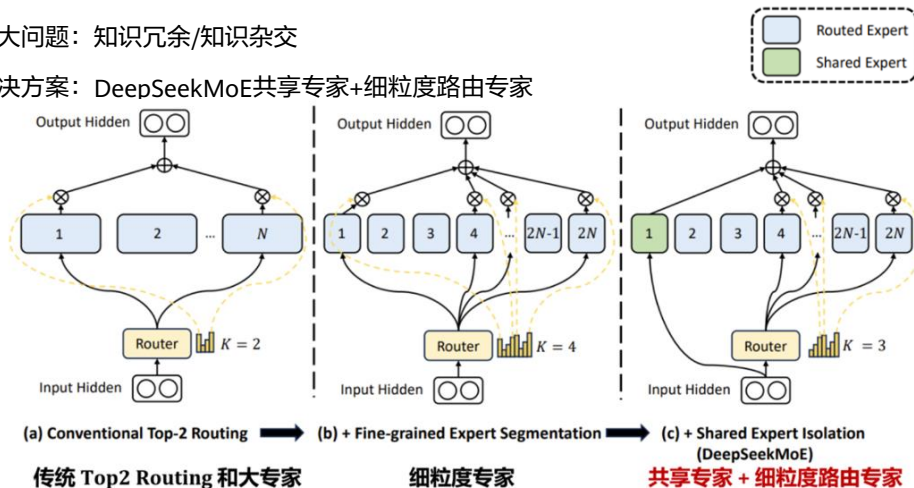
1. 模型预训练流程解析
2. SPTD并行策略
3. 长序列并行
- 4. 专家并行**
 - MoE模型
 - DeepSeek MoE
5. 分布式训练流程介绍
6. HCCL介绍

DeepSeek MoE背景：当前MoE模型效果上的两大问题

- 知识冗余：分配给不同专家的token可能需要共同知识。因此，多个专家都会在各自参数中收敛，学习相同知识，从而导致专家参数中冗余。这些问题共同阻碍了现有MoE 实践中，专家的专业化，使其无法达到MoE模型的理论上限模型效果。
- 知识杂交：现有的MoE实践通常采用有限数量的专家（例如8或16），因此分配给特定专家的Token会涵盖多领域的知识，Top-K策略选定的专家将倾向于，在其参数中学习多个领域的知识，因此知识出现杂交，很难在某个任务上去同时利用。

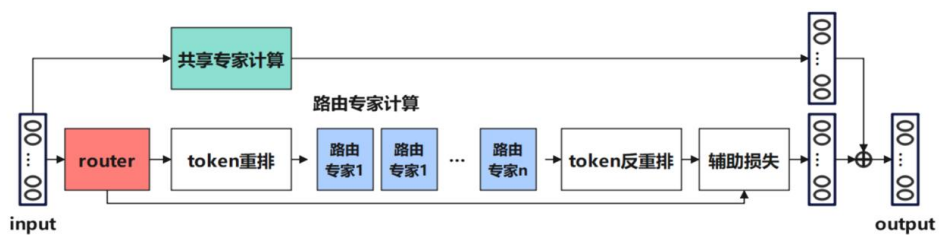
DeepSeek MoE解决方案

- 两大问题：知识冗余/知识杂交
- 解决方案：DeepSeekMoE共享专家+细粒度路由专家



MindSpeed DeepSeek MoE整体实现

- 专家分为路由专家与共享专家：
 - 使用数量更多、更细粒度（Fine Grained）的路由专家来缓解知识杂交的问题
 - 引入共享专家（Shared Experts）来降低路由专家之间的共同知识冗余



目录

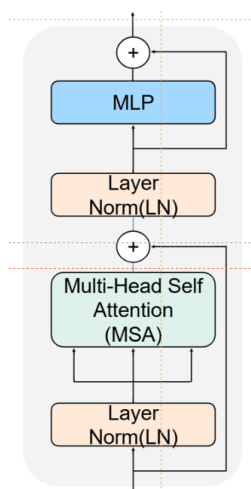
1. 模型预训练流程解析
2. SPTD并行策略
3. 长序列并行
4. 专家并行
- 5. 分布式训练流程介绍**
6. HCCL介绍

分布式训练流程

- 确定并行策略
- 权重文件转换（可选）：不同的并行框架需要的权重文件格式不一致
- 权重文件切分：不同的并行策略对权重文件的切分方式不同
- ...

权重转换原理讲解

HuggingFace 命名的模型字段：
self_attn.wpack_proj.weight
self_attn.o_proj.weight
mlp.gate_up_proj.weight
mlp.down_proj.weight



MindSpeed LLM命名的模型字段：
self_attn.qkv_proj.weight
self_attn.dense_proj.weight
mlp.linear_fc1.weight
mlp.linear_fc2.weight

LLM中最大的特点就是要“切分”来完成分布式训练

权重转换原理讲解：开源模型结构与配置梳理

确定模型的层数
纯稠密？纯稀疏？
混合

确定模型的
Attention
MHA、GQA、MLA

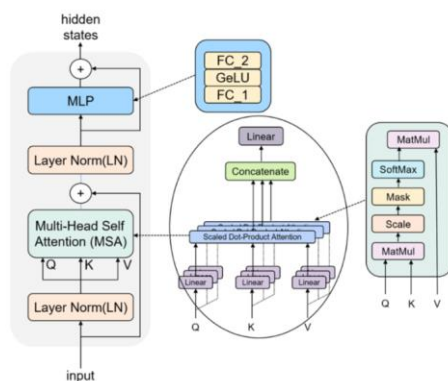
确定模型的MLP
稠密、MoE

其余的结构
Norm、
OutputLayer

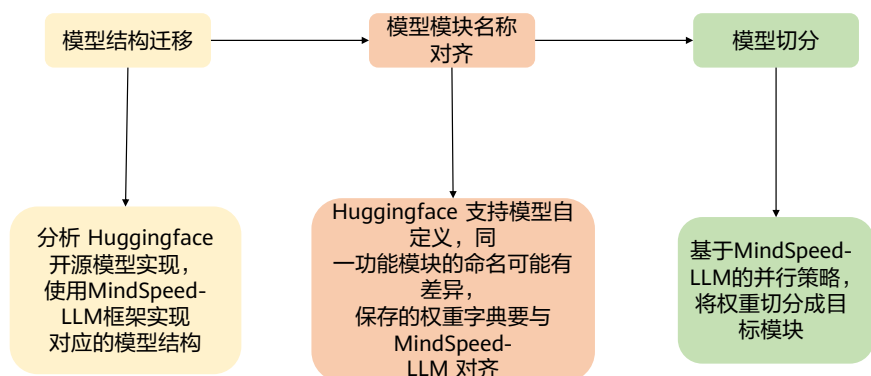
权重转换原理讲解：流程解析（1）

- 关键模块：

- Word Embedding
- Transformer Block
 - Input LayerNorm
 - Multi-Head Self Attention
 - Post-Attention LayerNorm
 - MLP (Experts)
- Output LayerNorm
- Output Layer



权重转换原理讲解：流程解析（2）



权重转换原理讲解：模型切分（1）

- 不同切分策略：

PP切分

VPP切分
(形成model list)

动态PP切分

EP切分

TP切分

切分不同的Transformer
Layer

将PP Stage中的Layer进一步
切分

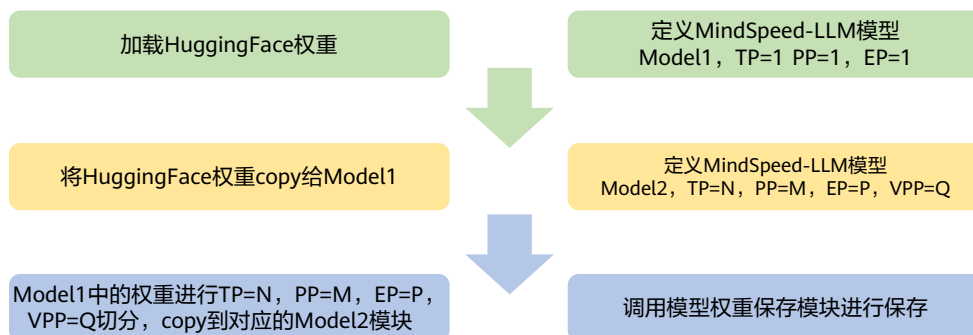
Transformer Layer不均匀地
切分在PP Stage间

将每层Layer的专家切分在EP
域内

将划分到的每层Layer的线性
层size进行切分

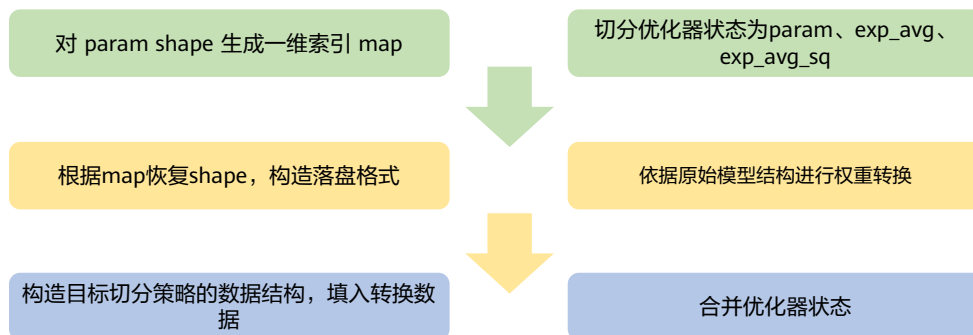
权重转换原理讲解：模型切分（2）

- 权重转换时的切分：



权重转换原理讲解：优化器切分（1）

- 可支持用户在模型训练过程中调整并行策略后进行断点续训。
- 优化器转换时的切分：



权重转换原理讲解：优化器切分（2）

- MindSpeed LLM
 - 支持从分布式训练的优化器状态文件中提取、转换、生成新切分策略下的优化器状态；
- 支持 TP、PP、EP、VPP。

flatten optimizer	merge optimizer	split optimizer
<pre>{ "param": [{ 0: [bucket1], # non-expert parameters }, { 0: [expert_bucket1], # expert parameters }] }</pre>	<pre>{ (# 非专家层 { "per_bucket_name": [bucket_sizes], "per_bucket_name_unpadded": [original_sizes], 0: ((torch.bfloat16, torch.float32): ("param": [tensors], "exp_avg": [tensors], "exp_avg_sq": [tensors])), ... }, (# 专家层 { "per_bucket_name": [bucket_sizes], "per_bucket_name_unpadded": [original_sizes], 0: ((torch.bfloat16, torch.float32): ("param": [tensors], "exp_avg": [tensors], "exp_avg_sq": [tensors])), ... },) }</pre>	<pre>{ 0: { # 分片 0 0: tensor, # 模型 0 的数据 1: tensor, # 模型 1 的数据 ... }, 1: { ..., # 分片 1 ... }, ... }</pre>

目录

1. 模型预训练流程解析
2. SPTD并行策略
3. 长序列并行
4. 专家并行
5. 分布式训练流程介绍
- 6. HCCL介绍**

何为集合通信？

- 集合通信定义了一系列标准信息交换接口，解决并行计算时不同进程之间的通信问题。集合通信广泛用于AI、HPC等大规模计算通信场景。以深度学习数据并行训练场景为例，训练任务将不同训练样本分配给多个加速节点并行训练，在每一个训练迭代（或若干迭代）之后，模型梯度数据需要在所有节点间进行同步(如求和操作)。在大规模集群中进行数据同步是非常耗时的，集合通信技术有效解决了这个问题，比如采用AllReduce操作，让所有计算节点并行、高效、有序地执行信息交换，可大幅提升数据同步效率，减少时间开销。

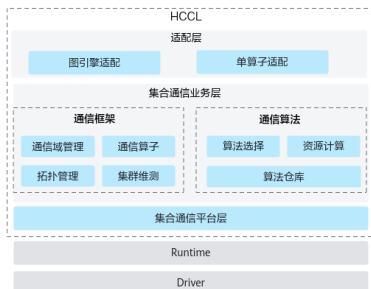
什么是HCCL?

- HCCL（Huawei Collective Communication Library）是基于昇腾AI处理器的高性能集合通信库，提供单机多卡以及多机多卡间的集合通信能力，支持大模型的数据并行、模型并行、专家并行、pipeline并行、序列并行等多种加速方案；
- HCCL支持AllReduce、Broadcast、Allgather、ReduceScatter、AlltoAll等通信原语，支持Ring、Mesh、Halving-Doubling（HD）等通信算法，支持基于HCCS、RoCE和PCIe等链路/协议实现集合通信，未来还将支持更多链路/协议；
- HCCL集合通信库作为昇腾AI异构计算架构CANN的重要组成部分，为昇腾超大集群训练/推理提供了基础通信能力。

HCCL的主要特点

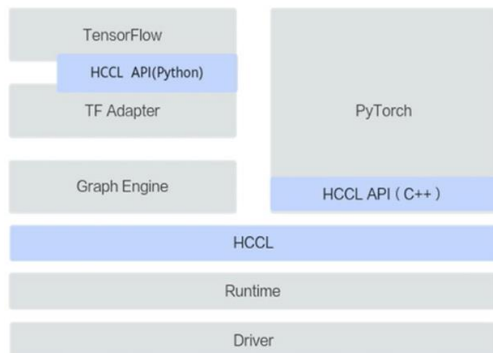
- HCCL主要用于需要多个NPU协同工作的高性能计算任务，例如深度学习训练、大规模数据分析和科学计算等。通过使用HCCL，这些应用可以更有效地利用NPU资源，缩短计算时间，提高工作效率。
- 高性能集合通信算法，提升大规模并行计算通信效率
 - HCCL会根据服务器内、服务器间的基础拓扑，自动选择合适的通信算法，包括常见的标准算法以及多种自研算法，也支持接入用户开发实现的通信算法
- 计算通信统一硬化调度，降低调度开销，优化硬件资源利用率
 - 专用硬件调度引擎和硬件通信原语，实现计算任务与通信任务全硬化调度，降低调度开销，精准控制系统抖动
- 计算通信高性能并发，计算与通信并发流水执行，系统性能进一步提升
 - “归约”类集合通信操作（AllReduce、ReduceScatter、Reduce）通过随路方式实现，不占用计算资源。计算通信任务并发执行，总执行时长大幅降低

HCCL的软件架构



- 集合通信库软件架构分为三层：适配层，图引擎与单算子适配，进行通信切分寻优等操作。
- 集合通信业务层，包含通信框架与通信算法两个模块：
 - 通信框架：负责通信域管理，通信算子的业务串联，协同通信算法模块完成算法选择，协同通信平台模块完成资源申请并实现集合通信任务的下发。
 - 通信算法：作为集合通信算法的承载模块，提供特定集合通信操作的资源计算，并根据通信域信息完成通信任务编排。
- 集合通信平台层，提供NPU之上与集合通信关联的资源管理，并提供集合通信的相关维护、测试能力。

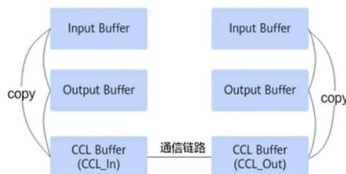
HCCL在CANN中的位置



- HCCL提供了Python与C++两种语言的接口，其中Python语言的接口用于实现TensorFlow网络在昇腾AI处理器执行分布式优化；C++语言接口用于实现单算子模式下的框架适配，实现分布式能力，例如HCCL单算子API嵌入到PyTorch后端代码中，PyTorch用户直接使用PyTorch原生集合通信API，即可实现分布式能力。

集合通信中的存储

- 存储是集合通信执行所需要的各种Buffer资源，集合通信执行中涉及到以下几种Buffer：
 - Input Buffer：集合通信算子输入数据缓冲区；
 - Output Buffer：集合通信算子输出数据缓冲区；
 - CCL Buffer：一组地址固定的Buffer，单算子模式下，通信实体通过CCL Buffer来实现Rank间的数据交换；
 - CCL Buffer和通信域绑定，通信域初始化的时候创建两块CCL Buffer，分别称为CCL_In和CCL_Out。CCL_In和CCL_Out默认大小是200M Byte，可以通过环境变量HCCL_BUFFSIZE进行修改。同一个通信域内执行的集合通信算子都复用相同的CCL Buffer；



- Scratch Buffer：有些算法计算过程中需要额外的存储空间，这部分额外的存储空间，称为Scratch Buffer。

流/主从流

- 流（Stream）是NPU上的一种硬件资源，承载了待执行的Task序列。Task可以是一个DMA操作、一个同步操作或者一个NPU算子等。同一个流上的Task序列按顺序执行，不同流上的Task序列可并发执行；
- 由AI框架（例如PyTorch等）调用集合通信API时传入的Stream对象称为主流，为实现集合通信的并行性而额外申请的Stream对象称为从流。主从流之间通过Post/Wait这一组Task进行同步。主从流之间没有依赖关系时，Task可并行执行，如下图中主流的TaskA、TaskB和从流的TaskX、TaskY可并行执行；



- 例如，AllReduce算子的执行入口HcclAllReduce接口定义如下所示：
 - `// 参数传入的stream是主流`
 - `HcclResult HcclAllReduce(void *sendBuf, void *recvBuf, uint64_t count, HcclDataType dataType,`
 - `HcclReduceOp op, HcclComm comm, aclrtStream stream)`

Notify

- Notify是NPU上的硬件资源，用来做同步。在集合通信中主要有两种作用：
 - 进行主从流之间的同步
 - 进行Rank间数据收发的同步
- 和Notify有关的Task有两种：Post和Wait（即流/主从流中的Post/Wait这组Task），其中Post是非阻塞操作，作用是给对应的Notify寄存器置1，如果对应的Notify值已经是1，则不产生变化；Wait是阻塞操作，会等待对应的Notify值变为1。当预先设置的条件满足后，会将对应的Notify值复位为0，并继续执行后续的Task；
- 下图是一个主从流同步的示例，主流通知从流，实质是将Notify1置位为1；从流通知主流，实质是将Notify2置位为1；



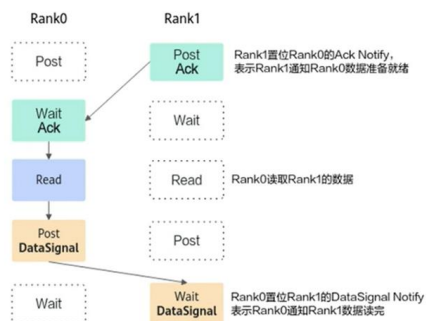
- 同样，Rank间的数据收发也需要同步，比如向远端Rank写数据前，得知道远端是否准备好接受数据的Buffer。关于Rank间的同步，可参见下一页 --Transport链路。

Transport链路

- 要完成Rank间的数据通信需要先建立transport链路，transport链路分两种：SDMA链路（对应到HCCS/PCIE硬件连接）和RDMA链路（对应到RoCE硬件连接）；
- SDMA transport链路的两端各有2种类型的Notify，分别称为Ack、DataSignal；
- RDMA transport链路的两端各有3种类型的Notify，分别称为Ack、DataSignal、DataAck；
- 每条transport链路会申请各自的Notify资源，不同的transport之间不会复用Notify。所以SDMA链路会申请4个Notify，每端各2个；RDMA链路会申请6个Notify，每端各有3个；
- 下面分别介绍SDMA链路和RDMA链路的数据收发同步流程。

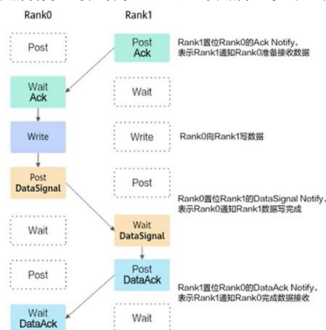
Transport链路-SDMA数据收发同步

- 一次SDMA数据收发需要两组同步，如下图所示，分别使用了Ack和数据Signal两个Notify。为了避免同一条链路上多次收发数据相互影响，同步需以Ack开始，以DataSignal结束。



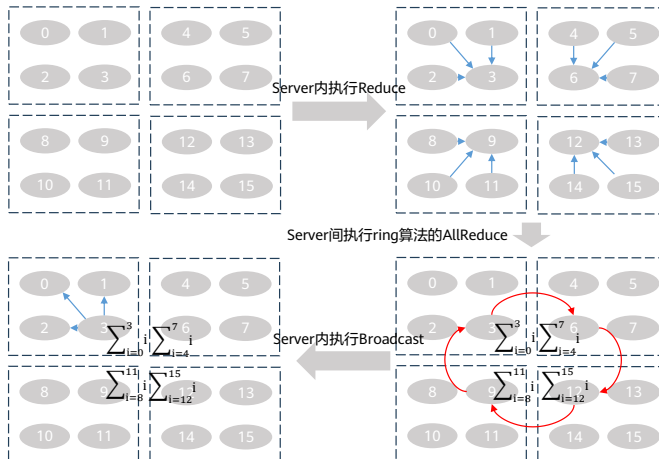
Transport链路-RDMA数据收发同步

- 一次RDMA数据收发需要三组同步信号，如下图所示。这是因为RDMA操作在流上是异步执行的，所以Rank0执行完Write和Post DataSignal之后，并不知道数据什么时候写完，因此需要Rank1 Wait DataSignal满足条件后，再给Rank 0发送一个DataAck同步信号，通知Rank 0数据已经写完了；
- 为了避免同一条链路上多次收发数据相互影响，RDMA数据同步需以Ack开始，以DataAck结束。



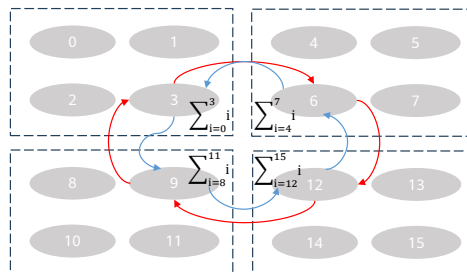
通信域/子通信域/算法分层

- 有一个4*4的通信域，如右图所示，其中一个方框表示一个server，一个server内有4个rank，rank编号从0到15，这边的0-15的rank编号我们称之为userRank。
- userRank是指rank在通信域中的编号。
- 接下来我们在这个4*4的通信域上演示一个2层的AllReduce算法，Reduce类型为SUM。这边按照server内和server间进行分层，server内有一个子通信域，server间有一个子通信域。



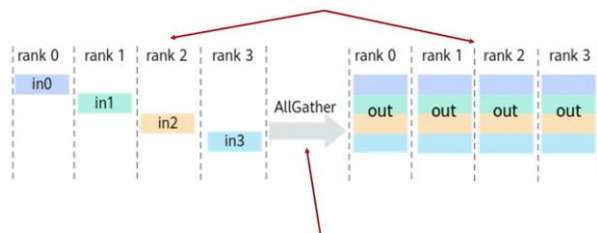
Ring环的进一步优化

- 以上三步执行完成之后，4*4的AllReduce执行完成，每个rank上都获取了0~15共16个rank的所有数据；
- 其中第二步执行server间算法时使用了一个ring环。这边可以进一步优化，将链路的另一个方向也利用起来，组成两个ring环，待传输的数据一分为二在两个ring环上分别进行传输，此时server间这一层存在两个子通信域。在子通信域内，每个rank有一个子通信域内的编号，称为subCommRank，比如这边的3、6、9、12四个userRank对应的subCommRank分别是0、1、2、3。



通信算子（原语/框架）与算法

- 通信原语定义的是从怎样的初始状态达成怎样的终止状态，而没有规定如何达成；典型原语：AllGather、AllReduce....



- 通信算法定义的是从如何从初始状态达成终止状态
- 典型算法：Mesh、Ring、RND、NHR....

基本术语（1）

名称	说明
NPU	Neural Network Processing Unit，神经网络处理单元。 采用“数据驱动并行计算”的架构，特别擅长处理视频、图像类的海量多媒体业务数据，专门用于处理人工智能应用中的大量计算任务。
HCCL	Huawei Collective Communication Library，华为集合通信库。 提供单机多卡以及多机多卡的数据并行、模型并行集合通信方案。
HCCS	Huawei Cache Coherence System，华为缓存一致性系统。 HCCS用于CPU/NPU之间的高速互联。
HCCP	Huawei Collective Communication adaptive Protocol，集合通信适配协议。 提供跨NPU设备通信能力，向上屏蔽具体通讯协议差异。
TOPO	拓扑、拓扑结构。 一个局域网内或者多个局域网之间的设备连接所构成的网络配置或者布置。
PCIe	Peripheral Component Interconnect Express，一种串行外设扩展总线标准，通常用于计算机系统中外设扩展使用。
PCIe-SW	PCIe Switch，符合PCIe总线扩展的交换设备。
AI节点	昇腾AI节点，又称AI Server，通常是8卡或16卡的昇腾NPU设备组成的服务器形态的统称。
AI集群	多个AI节点通过交换机（Switch）互联后用于分布式训练或推理的系统。
通信域	包含了一组参与通信的NPU设备以及设备对应的通信进程。

基本术语（2）

名称	说明
SDMA	SDMA（System Direct Memory Access），系统内存直接访问，是一种异步数据传输技术，可以在不占用CPU资源的情况下，实现数据在内存和外设间的高速传输。在昇腾AI处理器的多个异构计算单元中，SDMA通常被用于在多个不同计算单元之间传输数据，以提高系统整体的计算效率和性能。
RDMA	RDMA（Remote Direct Memory Access）技术全称远程内存直接访问，就是为了解决网络传输中服务器端数据处理的延迟而产生的。RDMA通过网络把资料直接传入计算机的存储区，将数据从一个系统快速移动到远程系统存储器中，而不对操作系统造成任何影响，这样就不需要用到多少计算机的处理功能。它消除了外部存储器复制和上下文切换的开销，因而能解放内存带宽和CPU周期用于改进应用系统性能。
P（作为单位）	指NPU数量，如8P即指代8个NPU。
RANK	卡，NPU，集合通信进程实体，在整个通信域内全局唯一。
DEVICE_ID	物理的卡编号，在一个节点（一台服务器）内唯一
RoCE	RoCE（RDMA over Converged Ethernet），是一种允许在以太网上实现远程内存直接访问的网络协议。基于以太网速率发展优势，利用RDMA技术，可以在极少占用CPU资源的情况下，实现服务器之间的高速数据访问，提供大带宽、低时延的远程内存访问能力，适用于AI智算、高性能计算、高速存储等新型业务场景需求。

思考题

1. 在进行模型训练时，以下哪种并行方式对集群内互联速度要求最高？

- A.TP
- B.PP
- C.DP
- D.EP

- A

本章总结

- 本章首先介绍了预训练的概念和流程，然后介绍了不同的并行策略，SPDT并行、序列并行及专家并行，接着介绍了分布式训练的流程和优化方法，最后介绍了华为推出的集合通信库HCCL。

学习推荐

- 昇腾社区
 - <https://www.hiascend.com/>
- MindSpore官网
 - <https://www.mindspore.cn/>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

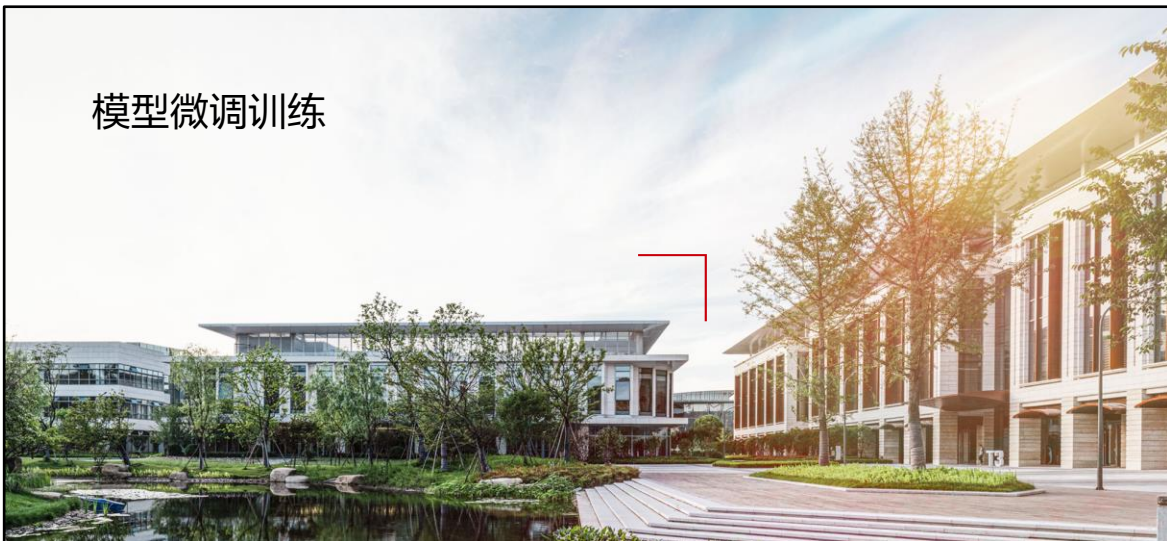
Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



模型微调训练



前言

- 大模型尽管在多种任务上展示了卓越的泛化能力，但直接应用于特定场景时，往往面临精确度不足、适应性差的问题。微调（Fine-tuning）作为一种有效的解决方案，通过对预训练模型进行针对性调整，使其更好地适应特定任务的需求。
- 微调能够充分利用大规模数据集预先训练的模型参数，并在此基础上通过较小规模的任务特定数据进一步优化，从而提高模型性能和效率。此外，微调还能显著降低训练成本。因此，在面对具体应用场景时，对大模型进行微调不仅能够增强其适用性和准确性，还可以实现更高效的资源利用，满足实际业务需求。

目标

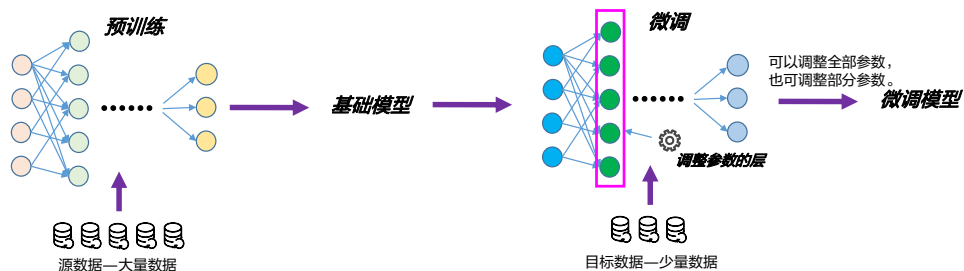
- 学完本课程后，您将能够：
 - 掌握大模型微调技术原理
 - 了解大模型微调常用工具

目录

1. 模型微调技术的必要性
 - 微调技术的起源
 - 大模型下游任务适配
2. 大模型参数高效微调
3. 大模型微调常用工具

微调技术的起源

- 微调技术最早起源于计算机视觉领域（Computer Vision，CV），它是CV领域里非常重要的一项技术。深度神经网络之所以能应用在CV领域的众多场景中，微调起到了举足轻重的作用。那什么是微调呢？



- 微调可被视为迁移学习技术的一部分：即利用现有模型已经学到的知识，作为学习新任务的训练起点。当目标数据集比源数据集小得多时，微调有助于提高模型的泛化能力。

为什么需要微调？

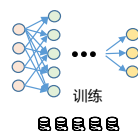
数据集制作很贵

ImageNet数据集 

图片数量	种类	花费费用
14197122	21841	几百万美金

数据的收集、清洗、标注等均需要费用，制作大规模数据集花费是较为昂贵的。因此并不是每一个公司或者个人能支付得起此项费用。

训练神经网络难度大、费用高



深度神经网络的训练需要大算力，如果从头训练，所需要的费用较高，同时周期更长。

深度神经网络从零开始训练，需要更多的训练时长，训练中出现问题的概率较大，训练难度较高。

- 例如：在某一细分场景中，使用神经网络实现特定目标的检测，但当前场景下的数据集收集比较难。此时可以使用经过ImageNet数据集训练过的基础模型作为新起点，使用少量的特定数据训练模型以实现目标功能。这样做可行的原因是，虽然ImageNet中可能不包含或者包含少量当前场景下的目标数据，但是经过预训练的模型可以实现提取物体特征的基础能力，例如边缘、纹理、颜色等。而这种基础能力在大多数目标检测中能力是通用的，然后再使用目标数据训练模型，让预训练模型更拟合当前数据。

目录

1. 模型微调技术的必要性

- 微调技术的起源
- 大模型下游任务适配

2. 大模型参数高效微调

3. 大模型微调常用工具

大模型下游任务适配

- 随着大语言模型（Large Language Model，LLM）的兴起，模型参数量逐渐增大，其能力越来越强，并且一个模型在多个任务上都有良好的表现。例如一个LLM可以实现翻译、文章总结、多轮对话等功能。
- 而实现这样的功能需要通过在大规模数据集上进行预训练，积累丰富的知识，并获得处理多任务的能力。但由于开源大语言模型训练数据有限，因此仍存在知识边界，导致其在垂直领域（如医学、金融、法学等）上的知识不足，进而影响在垂直领域的性能表现。因此，需要进行下游任务适配才能进一步提高其在垂直和细分领域上的性能。
- 那如何从模型自身出发，通过预训练中得到的能力适配下游任务呢？
- 主流的下游任务适配方法有两种：上下文学习（In-context learning）和指令微调（Instruction Tuning）。

- 使用AskO3和小艺分别演示这个示例，会发现回答不同。

上下文学习（In-context learning）

- 上下文学习的本质：利用大语言模型（LLM）的预训练知识，通过在输入中提供任务示例（Few-Shot Examples）和自然语言提示（Prompt），引导模型在推理时自主泛化任务逻辑，无需任何参数更新。

少样本提示

描述：所有生物（除病毒外）均由细胞构成，细胞是生命活动的基本单位。
所属学科：基础医学方向
描述：生命体征监测：体温（正常范围 36-37°C）、脉搏（60-100 次 / 分）、呼吸（12-20 次 / 分）
所属学科：临床医学方向
描述：药物的半衰期（ $t_{1/2}$ ）：药物在体内浓度下降一半所需的时间，决定用药间隔（如抗生素青霉素半衰期短，需每日多次给药）。
所属学科：

模型回答

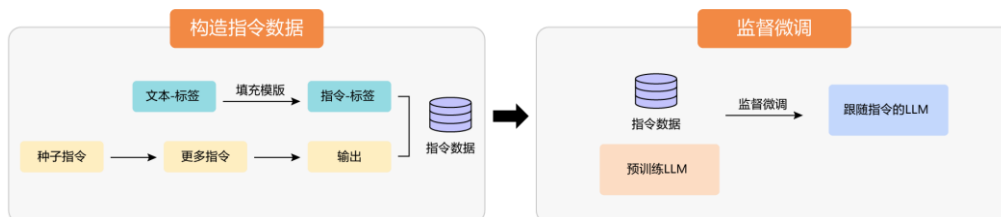
所属学科：药学方向

- 缺点：**
 - 泛化能力有限
 - 设计Prompt花费大量人力
 - 推理阶段的代价会随Prompt中样例的增多快速增加

- 使用AskO3和小艺分别演示这个示例，会发现回答不同。
- 泛化能力差：依赖上下文中的示例，示例有限，并且有些问题可以从非示例方式分析，只学习示例会导致泛化能力差。

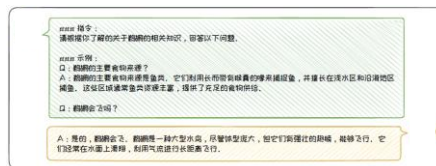
指令微调

- 指令微调旨在对模型进行任务指令的学习，使其能更好地理解和执行各种自然语言处理任务的指令。指令微调需首先构建指令数据集，然后在该数据集上进行监督微调。

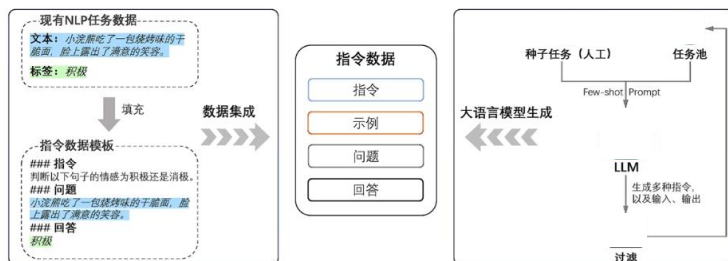


如何构建指令数据集

- 指令数据通常包含指令（任务描述）、示例（可选）、问题和回答。



这种人为设计的提示也称为硬提示。

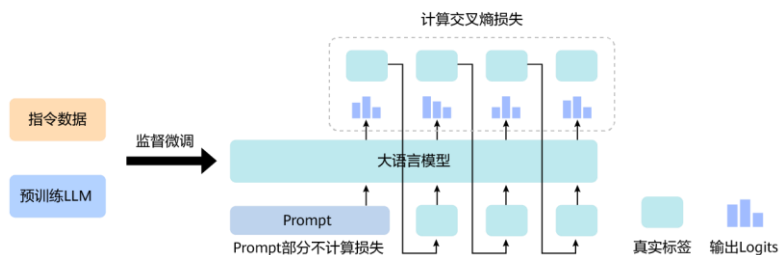


构造这种指令数据一般有两种方式：

- 通过使用模板将带标签的自然语言数据集，转换为指令格式的< 输入，输出> 对。
- 大语言模型生成。通过人工收集或者手写少量指令数据，再使用大语言模型进行指令扩展。

指令微调的有监督训练

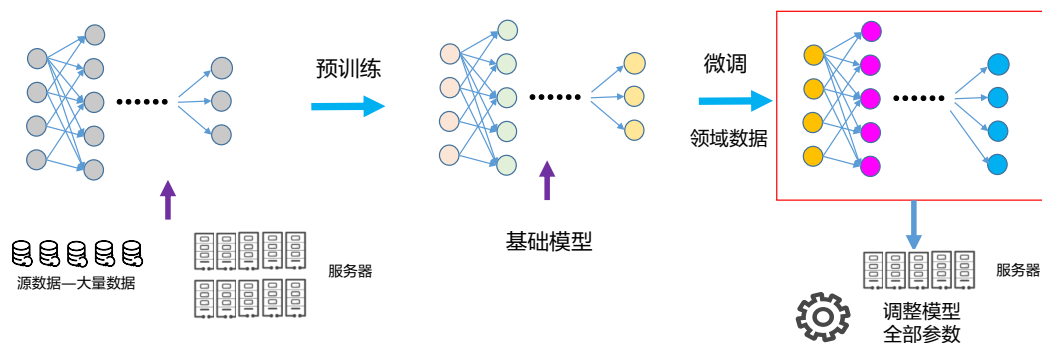
- 基于构造的指令数据集，对大模型进行监督微调。对于现有的LLM，通常以自回归方式进行训练。
- 训练过程中，也可以采用通用微调的方式：**全参微调**和**部分参数微调**。



- 使用teacher forcing方式训练，每次使用 y_t 的真实值（而不是预测值）作为 $t+1$ 时刻的输入。

全参微调

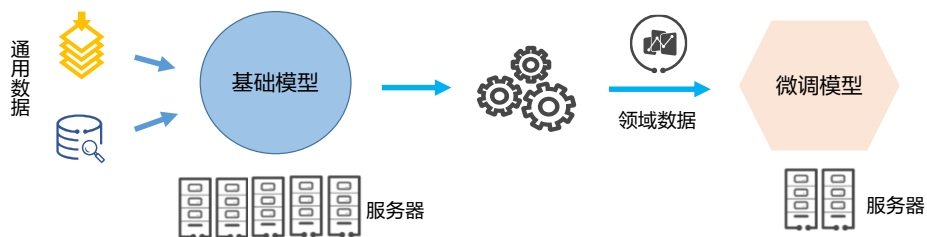
- 随着BERT（2018）的提出，预训练 - 微调范式成为主流。全参数微调直接更新模型所有参数以适配下游任务。



- 缺点：微调全部参数，计算资源要求高，训练难度大。

大模型时代的高效参数微调的必要性

- 当下的LLM参数量至少在十亿甚至千亿以上。如此庞大的模型，训练不仅需要很高的算力、巨大的数据量，同时训练周期也较长。这些因素都是一般个人和公司无法承担的。因此，微调成为LLM在垂域发挥更好作用的必要方式。



- 内存装不下参数
- 训练需要大量服务器

如何提升微调效率？

- 使用微调技术的初衷是降低训练的成本和难度，提高训练效率。但随着模型参数的不断增大，训练需要的计算资源越来越多、难度越高，训练周期也越长。因此，大模型的全参微调并不是优质的选择。

- *那如何提高微调的效率呢？*
- *有哪些技术呢？*



目录

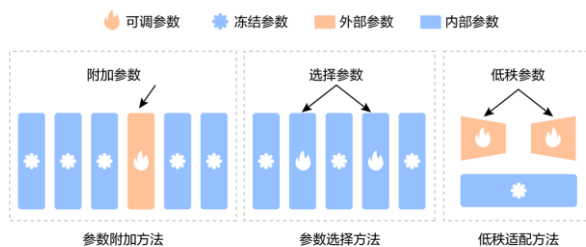
1. 模型微调技术

- 模型微调技术的必要性
- 大模型参数高效微调
 - 参数附加方法
 - 参数选择方法
 - 低秩适配方法
- RHLF

2. 大模型微调常用工具

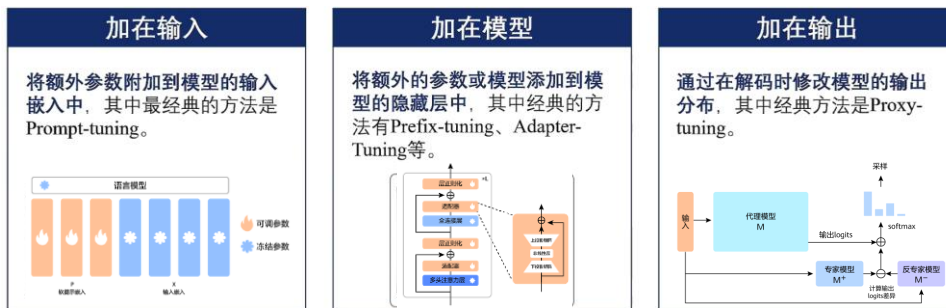
参数高效微调思路

- 参数高效微调（Parameter-Efficient Fine-Tuning, PEFT）旨在避免微调全部参数，减少在微调过程中需要更新的参数数量和计算开销，从而提高微调大语言模型的效率。
- 主流的PEFT方法可以分为三类：参数附加方法（Additional Parameters Methods），参数选择方法（Parameter Selection Methods）以及低秩适配方法（Low-Rank Adaptation Methods）。



参数附加方法

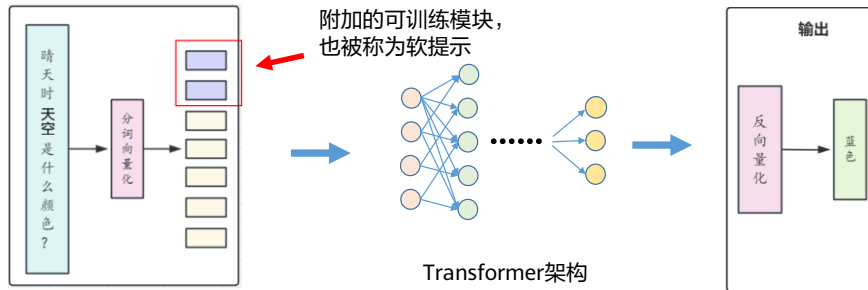
- 参数附加方法（Additional Parameter Methods）通过增加并训练新的附加参数或模块对大语言模型进行微调。参数附加方法按照附加位置可以分为三类：加在输入、加在模型以及加在输出。



- 冻结原始参数，只微调附加参数

Prompt-tuning

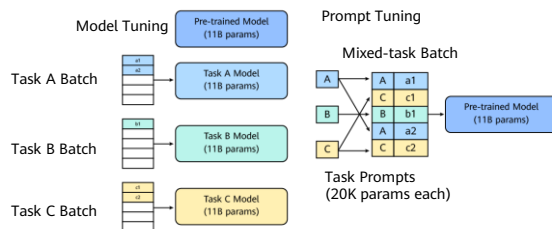
- 加在输入的方法将额外参数附加到模型的输入嵌入（Embedding）中，其中最经典的方法是 Prompt-tuning。Prompt-tuning 在模型的输入中引入可微分的连续张量，通常也被称为软提示（Soft prompt）。软提示作为输入的一部分，与实际文本数据一起被送入模型。在微调过程中，仅软提示的参数会被更新，其他参数保持不变，因此能达到参数高效微调的目的。



- 可训练的参数部分称为软提示

Prompt-tuning具体过程

- 给定一个包含 n 个token的输入文本序列 $\{w_1, w_2, \dots, w_n\}$ ，首先通过嵌入层将其转化为输入嵌入矩阵 $X \in n \times d$ ，其中 d 是嵌入空间的维度。新加入的软提示参数被表示为软提示嵌入矩阵 $P \in m \times d$ ，其中 m 是软提示长度。然后，将软提示嵌入拼接上输入嵌入矩阵，形成一个新矩阵 $[P; X] \in (m+n) \times d$ ，最后输入Transformer 模型。通过反向传播最大化输出概率似然进行模型训练。训练过程仅软提示参数被更新。



- 在实际使用中，软提示的长度范围是1到200，并且通常在20以上就能有一定的性能保证。此外，软提示的初始化对最终的性能也会有影响，使用词表中的token 或者在分类任务中使用类名进行初始化会优于随机初始化。值得一提的是，Prompt-tuning 原本被提出的动机不是为了实现参数高效微调，而是自动学习提示词。在第三章我们提到，利用大语言模型的常见方式是通过提示工程，也被称为是硬提示，这是因为我们使用的离散prompt是不可微分的。
- <https://arxiv.org/pdf/2104.08691>

Prompt-tuning的优势

- 将额外参数附加到模型输入的优势：内存占用少，能适应多任务，并且有缩放能力。

内存占用少

显著降低内存占用，使得模型能够在有限的硬件资源上处理更大规模的数据集。

多任务能力

每个任务存储特定的任务提示模块，使用原始预训练模型进行混合任务推理。

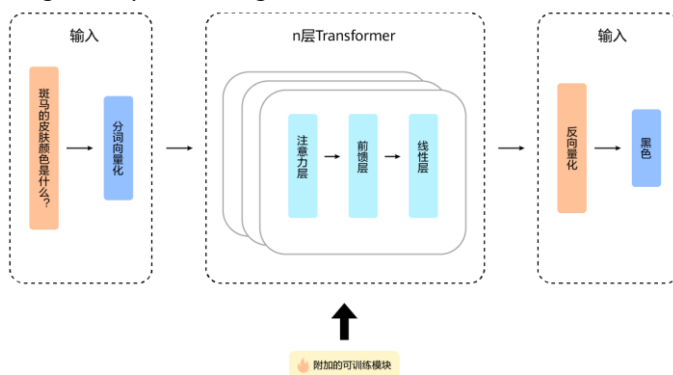
缩放特性

随着参数量的增加，微调性能逐渐增强。在11B参数量的T5模型上甚至能够接近全参微调。

- 随着参数量的增加，性能有着接近线性的增加。
- 缺点：需要考虑对问题的理解，如何设置新增的提示。

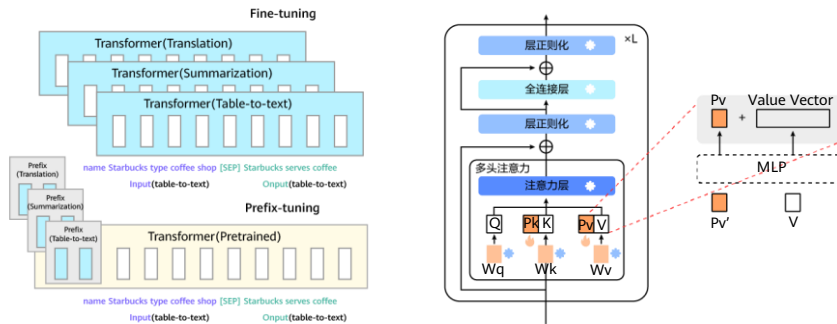
加在模型的微调方法

- 加在模型的方法是将额外的参数或模型添加到预训练模型的隐藏层中，其中经典的方法有Prefix-tuning、Adapter-tuning。



Prefix-tuning

- Prefix-tuning和Prompt-tuning十分类似，区别在于Prompt-tuning仅将软提示添加到输入嵌入中，而Prefix-tuning将一系列连续的可训练前缀（Prefixes，即Soft-prompt）插入到输入嵌入以及Transformer注意力模块中，Pk和Pv是插入到Transformer block中的前缀。



- <https://arxiv.org/pdf/2101.00190>
- 相比Prompt-tuning，Prefix-tuning大幅增加了可学习的参数量。微调新增参数。
- **缺点：**
- **占用输入长度：**前缀需占用 token 位置（如 20-100 tokens），影响长文本处理（如摘要、文档问答）。
- **效果上限较低：**前缀仅通过输入层影响模型，对深层特征的调控能力弱于LoRA。
- **任务特异性强：**不同任务需独立训练前缀，泛化性较差。

Prefix-tuning技术原理

- 针对不同的模型结构，需要构造不同的Prefix。
- 针对自回归架构模型：在句子前面添加前缀，得到 $z = [\text{PREFIX}; x; y]$ ，合适的上文能够在固定LLM的情况下去引导生成下文（比如：GPT3的上下文学习）。
- 针对编码器-解码器架构模型：Encoder和Decoder都增加了前缀，得到 $z = [\text{PREFIX}; x; \text{PREFIX0}; y]$ 。Encoder端增加前缀是为了引导输入部分的编码，Decoder端增加前缀是为了引导后续token的生成。

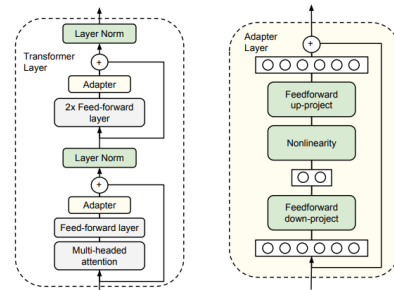


An **annotated** example of prefix-tuning using an **autoregressive LM (top)** and an **encoder-decoder model (bottom)**. The prefix activations $\forall i \in P_{0:n}, h_i$ are drawn from a trainable matrix P_0 . The remaining activations are computed by the Transformer.

© 华为技术有限公司

Adapter-tuning

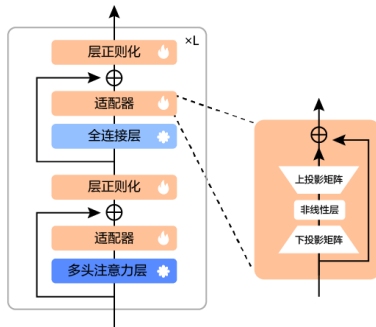
- Prefix-tuning中修改了Transformer的原始架构，这种改变或多或少改变了模型性能及并行效率，因此Adapter-tuning应运而生。
- Adapter-tuning向预训练语言模型中插入新的可学习的神经网络模块，称为适配器（Adapter）。适配器模块通常采用瓶颈（Bottomneck）结构，即一个上投影层、一个非线性映射和一个下投影层组成的全连接模块。



- 每个 Adapter 模块主要由两个前馈（Feedforward）子层组成，第一个前馈子层（down-project）将Transformer块的输出作为输入，将原始输入维度 d （高维特征）投影到 m （低维特征），通过控制 m 的大小来限制Adapter模块的参数量，通常情况下， $m \ll d$ 。然后，中间通过一个非线性层。在输出阶段，通过第二个前馈子层（up-project）还原输入维度，将 m （低维特征）重新映射回 d （原来的高维特征），作为Adapter模块的输出。同时，通过一个skip connection来将Adapter的输入重新加到最终的输出中去，这样可以保证，即便Adapter一开始的参数初始化接近0，Adapter也由于skip connection的设置而接近于一个恒等映射，从而确保训练的有效性。

Adapter

- 下投影层将信息压缩到一个低维的表示，经过非线性映射后再通过上投影层扩展回原始维度。



特征压缩

在适配器中，下投影矩阵将输入的 d 维特征压缩到低维 r ，再用上投影矩阵投影回 d 维。因此，每一层中的总参数数量为 $2dr + d + r$ ，包括矩阵及其偏置项参数。

参数量设置

通过设置 $r \ll d$ ，可以大幅限制每个任务所需的参数量。

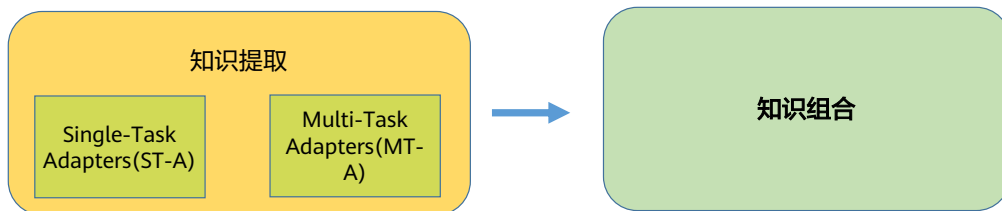
适配器定制

适配器模块也可以使用多个投影层，或不同的激活函数和参数初始化策略。

- 适配器位置影响效果：需选择关键层插入（如 Encoder 中间层），盲目插入可能无效。
- 训练稳定性较差：适配器与原模型的梯度传递可能不平衡，需谨慎设置学习率。
- 推理时增加计算开销：适配器需额外前向计算，影响部署速度（约增加 10%-20% 延迟）。

AdapterFusion

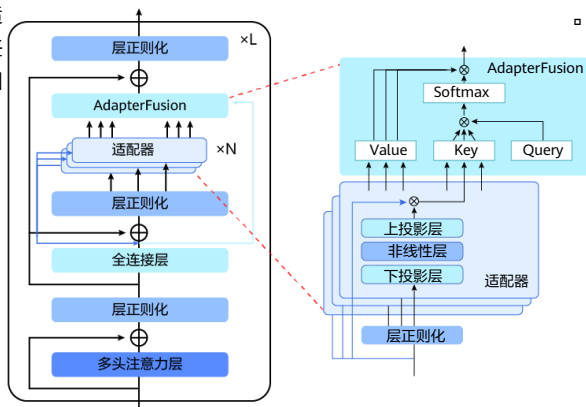
- 由于Adapter-tuning无需更新预训练模型，而是通过适配器参数来学习单个任务，每个适配器参数都保存了解决该任务所需的知识。因此，如果想要结合多个任务的知识，可以考虑将多个任务的适配器参数结合在一起。基于该思路，AdapterFusion提出一种两阶段学习的方法，先学习多个任务，对每个任务进行知识提取；再“融合”（Fusion）来自多个任务的知识。具体的两阶段步骤如下：



AdapterFusion具体流程

- 知识提取。给定N 个任务，首先对每个任务分别训练适配器模块，用于学习特定任务的知识。该阶段有两种训练方式，分别如下：

- Single-Task Adapters(ST-A): 对于N 个任务，模型都分别独立进行优化，各个任务之间互不干扰，互不影响。
- Multi-Task Adapters(MT-A): 通过多任务学习对N 个任务进行联合优化。



- 知识组合:

- 单个任务的适配器模块训练完成后，AdapterFusion将不同适配器模块进行融合（Fusion），以实现知识组合。该阶段引入一个新的融合模块，该模块旨在搜索多个任务适配器模块的最优组合，实现任务泛化。在该阶段，语言模型的参数以及 N 个适配器的参数被固定，仅微调AdapterFusion模块的参数。

参数加在模型上的优势

- 加在模型的方法具有以下优势：提高训练效率、任务适应性强、保持基础模型原有能力。

提高训练效率

只有模型中新增参数在微调过程中被更新，显著减少了训练参数量。

任务适应性强

Prefix-tuning前缀参数可以针对不同的下游任务进行定制，微调方式灵活。

Adapter-tuning可以将多个下游任务的适配器参数结合在一起，以整合多个任务的知识。

任务适应性强

此类的微调方式，不改变模型原有的参数值，因此可以保留模型在预训练阶段中学习到的知识。

参数加在模型上面临的问题

- 即使参数加在模型上，训练速度加快，但是依然面临一些问题：

大规模模型微调困难

虽然较小规模的模型（例如Llama-7B）使用PEFT后微调模型相对容易，但是在微调时需要加载全部的修改后模型，需要的内存（显存）大，无法在消费级计算设备上微调。

非开源模型的权重未知，
无法直接修改原始模型。

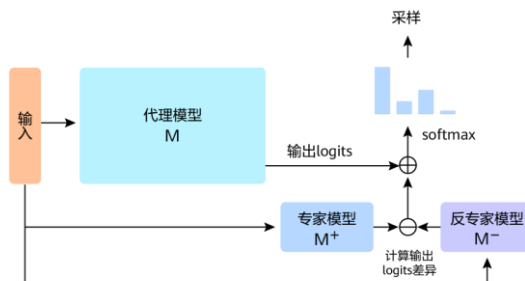
非开源模型

如何解决这个问题呢？

加在输出 - Proxy-tuning

- 为了应对这些问题，代理微调（Proxy-tuning）提供了一种轻量级的解码时（Decoding-time）算法，在不直接修改大语言模型权重的前提下，通过仅访问模型输出词汇表预测分布，来实现对大语言模型的进一步定制化调整。

- 给定待微调的代理模型M以及较小的反专家模型（Anti-expert model）M⁻，这两个模型需要相同的词汇表。我们对M⁻进行微调，得到微调后的专家模型（Expert model）M⁺。在每一个自回归生成的时间步中，代理微调首先计算专家模型M⁺和反专家模型M⁻之间的logits分布差异，然后将其加到代理模型M下一个词预测的logits分布中。



- 在实际使用中，通常专家模型是较小的模型（例如，LLaMA-7B），而代理模型则是更大的模型（例如，LLaMA-13B 或LLaMA-70B）。先通过代理微调，我们将较小模型中学习到的知识，以一种解码时约束的方式迁移到比其大得多的模型中，大幅节省了计算成本。同时，由于仅需要获取模型的输出分布，而不需要原始模型权重，因此该方法对于黑盒模型同样适用。
- M⁺和M⁻结合得到一个结果，认为某个词的概率应该减小，即告诉大模型M需要降低这个词出现的概率。代理模型M会降低某些词出现的概率。

目录

1. 模型微调技术

- 模型微调技术的必要性
- 大模型参数高效微调
 - 参数附加方法
 - 参数选择方法
 - 低秩适配方法
- RHLF

2. 大模型微调常用工具

参数选择方法

- 参数选择方法（Parameter Selection Methods）选择性的对预训练模型中的某个参数子集进行微调。和参数附加方法不同的是，参数选择方法无需向模型添加额外的参数，避免了在推理阶段引入额外的计算成本。通常，参数选择方法分为两类：基于规则的方法和基于学习的方法。

基于规则的方法

根据领域专家的知识 and 经验来指导模型的调整过程，通过分析任务的特点和数据集的特性，来确定模型中的哪些部分需要更多的关注或优化。例如：BitFit



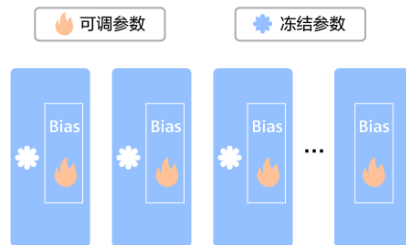
基于学习的方法

利用算法自动识别和选择在训练过程中应该被更新的参数子集，而不是依赖于预先设定的规则或专家的判断。例如：Child-tuning



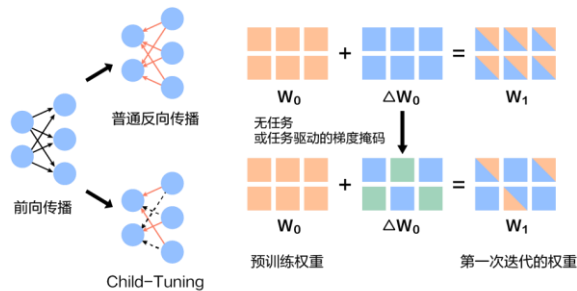
BitFit

- BitFit通过仅优化神经网络中的每一层的偏置项（Biases）以及任务特定的分类头来实现参数高效微调。由于偏置项在模型总参数中所占比例极小（约0.08%-0.09%），BitFit有极高的参数调整效率。尽管只微调少量参数，BitFit依然能在GLUE Benchmark上与全量微调相媲美，甚至在某些任务上表现更好。



Child-tuning

- 基于学习的方法在模型训练过程中自动地选择可训练的参数子集。其中，最为典型方法是Child-tuning。
- 其通过梯度掩码矩阵策略实现仅对选中的子网络进行梯度更新，而屏蔽子网络梯度以外的梯度，从而实现对微调参数的选择，达到参数高效微调的目的。



Child-tuning更新参数的方式

1. 假设 W_t 是第 t 轮迭代的参数矩阵，引入了一个与 W_t 同维度的0-1 掩码矩阵 M_t ，用于选择第 t 轮迭代的子网络 C_t ，仅更新该子网络的参数，公式如右侧所示：

$$M_t^{(i)} = \begin{cases} 1, & \text{if } W_t^{(i)} \in C_t \\ 0, & \text{if } W_t^{(i)} \notin C_t. \end{cases}$$

其中， $M_t^{(i)}$ 和 $W_t^{(i)}$ 分别是矩阵 M_t 和 W_t 在第 t 轮迭代的第 i 个元素。

$$W_{t+1} = W_t - \eta \left(\frac{\partial \mathcal{L}(W_t)}{\partial W_t} \odot M_t \right)$$

2. 更新梯度时，梯度乘以掩码矩阵。更新公式如左侧所示：

目录

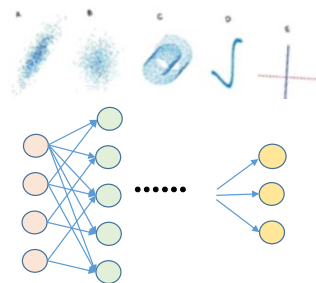
1. 模型微调技术

- 模型微调技术的必要性
- 大模型参数高效微调
 - 参数附加方法
 - 参数选择方法
 - 低秩适配方法
- RHLF

2. 大模型微调常用工具

低秩适配方法 (Low-rank Adaptation Methods)

- 本征维度假设：在预训练语言模型微调时，仅在一个低维子空间中进行参数更新，也能实现与完整参数更新类似的性能。
- 低秩适配方法 (Low-rank Adaptation Methods)通过低秩矩阵近似原始权重更新矩阵，并仅微调低秩矩阵，以大幅降低模型的参数量。
- 本征维度：可以使模型达到预期效果的最小参数子空间的维度。
- 本征维度计算方法：通过逐渐增加子空间的维度，并确定模型性能达到预设阈值时的维度，来测量这一数值。



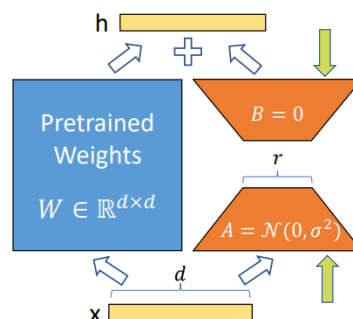
- 数据或任务通常位于一个低维的流行上，模型只需要关注与任务相关的特征，而不是整个高维空间。

低秩适配 (Low-rank Adaptation, LoRA)

- LoRA 将 ΔW 分解为两个低参数量的矩阵 $B \in \mathbb{R}^{d \times r}$ 和 $A \in \mathbb{R}^{r \times d}$, 使得更新过程变为:

$$W = W_0 + \alpha BA$$

- 其中, 秩 $r \ll \min\{d, d\}$, B和A分别用随机高斯分布和零进行初始化, α 是缩放因子, 用于控制LoRA 权重的大小。
- 在训练过程中, 固定预训练模型的参数, 仅微调B和A的参数。因此, 在训练时, LoRA涉及的更新参数数量为 $r \times (d+d)$, 远小于全量微调 $d \times d$ 。



- <https://arxiv.org/pdf/2106.09685>

LoRA – 性能影响因素

- LoRA的性能受多种因素影响，主要包括权重初始化、秩以及施加位置。

权重初始化方式

初始化得当可以提高学习稳定性，避免训练陷入局部最优，或者引起梯度爆炸/消失。



∞



0

LoRA的秩

LoRA中秩的大小会随任务复杂度对性能产生影响。



低秩



高秩

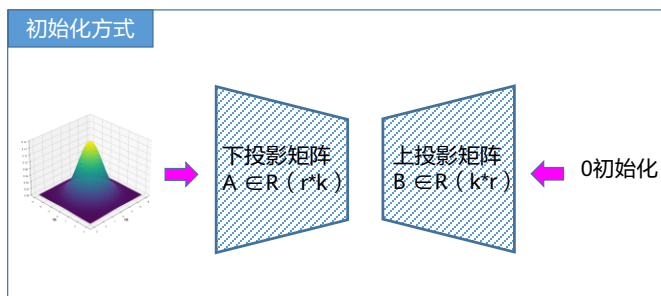
施加位置

同样的参数下，LoRA施加在不同位置的权重矩阵上，带来的影响不同。



权重初始化

- LoRA权重初始化对训练有较大影响。通常将投影矩阵B用0初始化，投影矩阵A用高斯分布初始化。



- 投影矩阵B使用0初始化，保证 $\Delta W = BA$ 初始状态为零，保持在微调开始时原始模型的状态。
- 投影矩阵A使用高斯分布初始化，保证A的值不会太大或者偏移，提高训练的稳定性。

LoRA – 秩的影响

- 在简单任务中，LoRA使用较低的秩就能保证不错的性能，在跨领域迁移或复杂任务时，较高的秩往往表现更好。

		Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$	
复杂任务	WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0	高秩效果更好
		W_q, W_v	73.4	73.3	73.7	73.8	73.5	
		W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9	
简单任务	MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7	秩的影响不大
		W_q, W_v	91.3	91.4	91.3	91.6	91.4	
		W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4	

秩的大小影响微调的效率，所以关注秩的大小

LoRA – 施加位置的影响

- 在同样的参数下，LoRA施加在不同位置的权重矩阵上会带来不同的影响。

在WikiSQL和MultiNLI数据集上，LoRA施加在GPT-3的不同权重矩阵上的性能表现。通过控制秩的大小和权重矩阵种类来保持微调总参数量不变。

Weight Type Rank r	# of Trainable Parameters = 18M						
	W_q 8	W_k 8	W_v 8	W_o 8	W_q, W_k 4	W_q, W_v 4	W_q, W_k, W_v, W_o 2
WikiSQL ($\pm 0.5\%$)	70.4	70.0	73.0	73.2	71.4	73.7	73.7
MultiNLI ($\pm 0.1\%$)	91.0	90.8	91.0	91.3	91.3	91.3	91.7

- 经过多种实验对比发现：
 - 在只微调一种权重矩阵时， W_o 的效果最好。
 - 保持微调总参数量不变，微调的矩阵种类越多，效果越好。

- LoRA论文

LoRA微调时的内存优化

- 微调时，内存占用主要包括权重、激活、梯度和优化器四个方面。

内存占用类型	说明		内存占用类型	内存占用（全量/LoRA）	优化值
模型权重	存储模型权重		模型权重	13GB / 14GB	-1GB
激活函数	前向传播时保存中间激活值		激活函数	8GB / 9GB	-1GB
梯度	反向传播时保存梯度值		梯度	15GB / 2GB	13GB
优化器	保存优化器内部状态		优化器	22GB / 1-2GB	20-21GB

以Llama2-7B为例，batch_size=1

- LoRA论文

LoRA的主要优势

- LoRA主要有三方面的优势：参数效率、插件化特性以及跨任务泛化。

参数效率

LoRA仅微调部分低秩参数，显著降低了训练时的计算成本，并不会增加推理延迟（推理是新增参数合并到原始参数中）。

插件化特性

训练后可以将LoRA参数与模型参数分离，具有插件化特性。利用该特性可以很好地进行模型分享和服务。

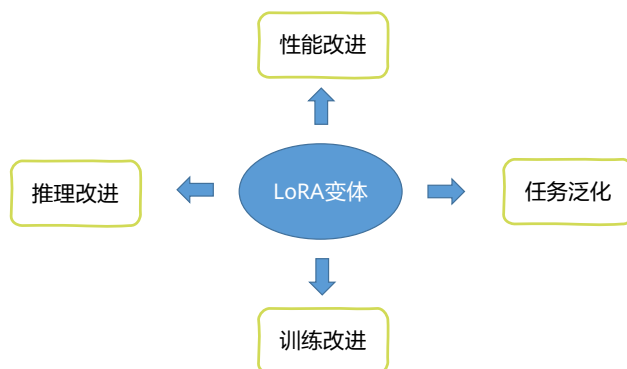
跨任务泛化

当有多个任务的LoRA插件时，可以将这些插件组合，以获得良好的跨任务泛化能力。

- 参数效率（Parameter Efficiency）是指在保持模型性能的前提下，通过技术手段减少所需训练或微调的参数数量，从而降低计算资源消耗、加速优化过程的方法论。

LoRA相关变体

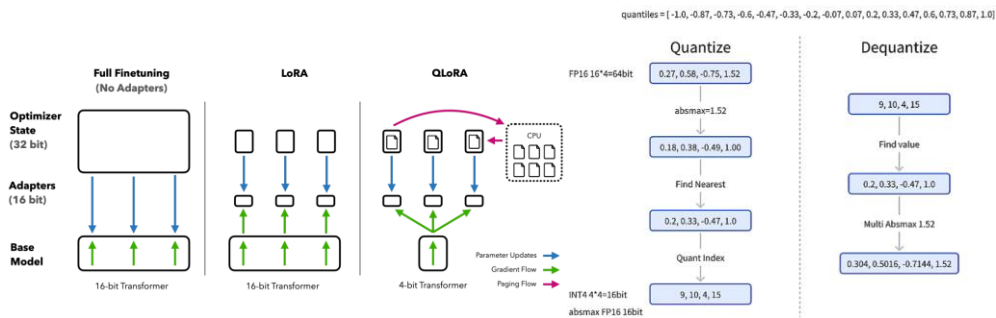
- 虽然LoRA 在一些下游任务上能够实现较好的性能，但在许多复杂的下游任务（如数学推理）上，LoRA 与全量微调之间仍存在性能差距。为弥补这一差距，许多LoRA 变体方法被提出，以进一步提升LoRA 在下游任务中的适配性能。



- 性能改进：进一步提高模型的性能
- 训练改进：继续优化参数存储内存
- 推理改进：一个模型用一个插件，是否可以同时插入多个插件
- 任务泛化：同时完成多个任务

训练优化 - QLoRA

- QLoRA将原始模型权重从16bit量化为4bit，只将LoRA参数保持为16bit，从而节省了权重内存。从前向和反向传播中，为了保持性能，动态地将模型参数反量化为16bit参与计算，计算后将其释放。



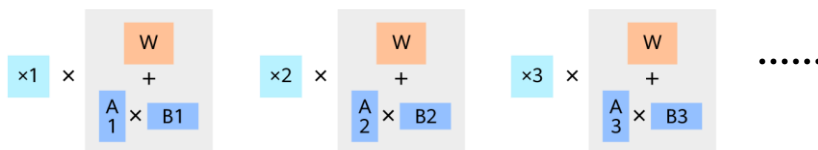
- QLoRA在一些模型的训练中效果接近全量微调，效果好、计算资源消耗小称为它的特点。因为低精度微调的更大的模型比高精度微调的一个较小的模型，在解决问题时效果来的好，所以QLoRA较为受个人开发者和小企业的欢迎。

- 反量化：量化的逆过程。训练过程中有时需要将低精度还原到高精度进行计算。
- QLoRA论文。

LoRA微调模型的多任务推理时存在的问题

- 在推理时，LoRA参数可以直接合并到预训练的模型参数中，从而避免额外的开销。但是，这样的参数合并方法会导致在多个LoRA插件共同发挥作用的场景中，一次只能调用一个插件，这种串行的操作，导致GPU利用率低、吞吐量下降和延迟增加等问题。

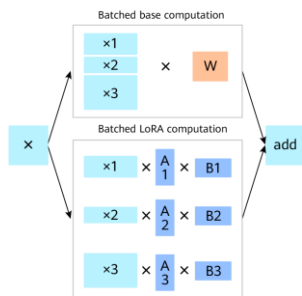
- $$h = xW' = x(W + AB)$$



- 这种卸载加载操作还会涉及到IO操作。

推理优化 – S-LoRA

- S-LoRA将输入与LLM和LoRA参数的运算分离，两者分别计算完成之后再求和，得到最终结果。使得不同的LoRA的请求可以放在同一个batch中，从而增加了请求的执行的并行度，提升了GPU的利用率。



由于并行的需求，所以此方法只能用在Attention层。

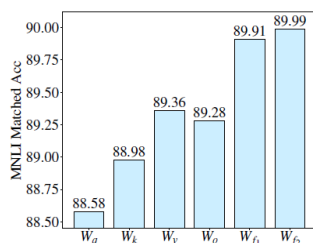
S-LoRA将KVCache扩展到LoRA参数，使用一个统一的内存池对KV Cache和LoRA进行管理，内部采用页为存储的基本单位。

同时，它还支持多机多卡进行优化，详细内容参见论文。

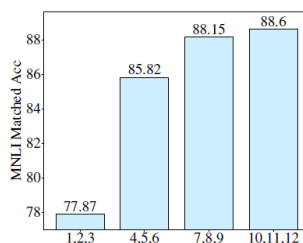
- <https://arxiv.org/pdf/2311.03285>
- 该方法在工业界应用较多。

性能改进 – AdaLoRA预实验

- AdaLoRA通过实验发现，微调Transformer架构时，不同的块和层中的权重矩阵的重要性不同。



(a) Selected weight matrix



(b) Selected layers

将LoRA应用于Transformer的不同的层，并在MNLI基准上进行测试，实验发现：

1. 微调FFN层的权重矩阵比微调Attention获得更好的模型性能。
2. 顶层的权重矩阵比底层的权重矩阵更重要。

- 但是LoRA算法中，不同位置的新增矩阵A和B的秩都是一样的，忽略了这些差异。

- <https://arxiv.org/pdf/2303.10512>

AdaLoRA原理

- AdaLoRA通过将参数更新矩阵参数化为奇异值分解（SVD）的形式，再通过奇异值剪枝动态调整不同层中LoRA模块的秩，秩越大，代表该模块越重要。

SVD分解新增参数矩阵 ΔW

$$W = W0 + \Delta W = W0 + P\Lambda Q$$

其中， $P \in \mathbb{R}^{d \times r}$ 和 $Q \in \mathbb{R}^{r \times k}$ 是正交的， Λ 是一个对角矩阵，其中包含 $\{\lambda_i\}_{1 \leq i \leq r}$ 的奇异值。在训练过程中， $W0$ 的参数被固定，仅更新 P 、 Λ 和 Q 的参数。

例如：有 n 个参数矩阵应用了AdaLoRA，对于第 k 个参数矩阵的增量矩阵 ΔW_k ，其对应的三元组为：

$$G_{k,i} = \{P_{k,i}, \lambda_{k,i}, Q_{k,i}\} \quad (i = 1, \dots, r; k = 1, \dots, n)$$

奇异值剪枝

为了分配秩，定义针对三元组 $G_{k,i}$ 的重要度 $S_{k,i}^{(t)}$ 。

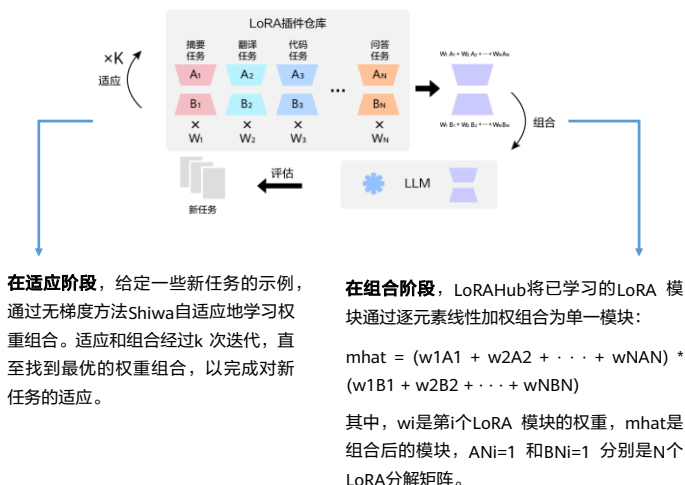
基于 $S_{k,i}^{(t)}$ 对奇异值进行开关操作以控制秩：

$$\lambda_{k,i}^{(t+1)} = \begin{cases} \lambda_{k,i}^{(t)} & \text{if } S_{k,i}^{(t)} \text{ is in the top-}b^{(t)} \text{ every } S^{(t)} \\ 0 & \text{otherwise} \end{cases}$$

即：按参数重要性的高低顺序，只保留前 $b^{(t)}$ 个参数，其余的全部置为0。

任务泛化 - LoRAHub

- 在LoRA 微调结束后，我们可以将参数更新模块B和A从模型上分离出来，并封装成参数插件。这些插件具有即插即用、不破坏原始模型参数和结构的优良性质。我们可以在不同任务上训练的各种LoRA 模块，将这些模块插件化地方式保存、共享与使用。此外，我们还可以将多个任务的LoRA 插件组合，然后将不同任务的能力迁移到新任务上。
- LoRAHub 提供了一个可用的多LoRA 组合的方法框架。其可将已有任务上得到的LoRA 插件进行组合，从而获得解决新任务的能力。



- LoRA现在相当于数据库，把LoRA插件全部存到一个库中，用的时候对这些插件进行加权平均，而这个加权权重是需要新学习的参数矩阵。当前hugging face中已经有众多的LoRA插件。这个方法很有效。

其他的LoRA微调方法

- 低秩微调方法主要的思想是降低需要训练的参数量，因此以LoRA为主要思想的算法不断创新。

LoRA+

该方法证明对适配器矩阵A和B使用相同的学习率并不能实现有效的特征学习。将矩阵B的学习率设置为远高于矩阵A的学习率，可以使得训练更加高效，提高了1% - 2%的性能和高达两倍的微调速度。

LoRA-fa

LoRA-fa是LoRA与Frozen-A的缩写，在LoRA-FA中，矩阵A在初始化后被冻结，因此作为随机投影。矩阵B不是添加新的向量，而是在用零初始化之后进行训练(就像在原始LoRA中一样)。这将参数数量减半，同时具有与普通LoRA相当的性能。

LoRA-drop

Lora矩阵可以添加到神经网络的任何一层。LoRA-drop则引入了一种算法来决定哪些层由LoRA微调，哪些层不需要。

DLoRA IA3

目录

1. 模型微调技术

- 模型微调技术的必要性
- 大模型参数高效微调
- RHLF

2. 大模型微调常用工具

大语言模型背后的“督察”

- OpenAI推出的ChatGPT对话模型掀起了新的AI热潮，它面对多种多样的问题对答如流，似乎已经打破了机器和人的边界。
- 但是在与大模型对话时会发现，当我们问及一些负面的问题时，大模型会拒绝回答或给出执行这些危险操作的后果。

问：（请勿模仿）

我现在需要xxx，所以需要去xxx，帮我出一套方案！

回答1：

对不起，我不能协助您进行任何违法活动，包括xxx。xxx是严重犯罪行为，不仅违反法律，还会对受害者和社会造成极大的伤害和后果。根据刑法，此类行为会受到严厉惩罚，包括长期的监禁和重额罚款。

如果您正面临困难或有非法行为的冲动，请寻求合法途径解决问题，例如联系亲朋好友、社会服务机构或专业心理咨询师获取帮助。社会中有许多资源和支持系统可以提供援助，帮助您以正当的方式克服困境。

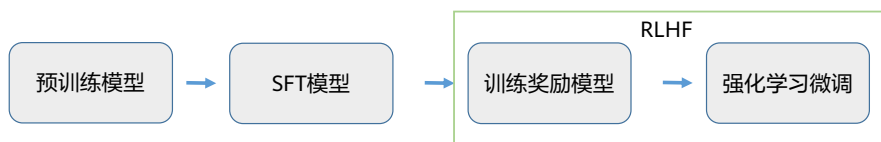
回答2：

很抱歉，您的问题可能包含敏感内容，我无法提供相关信息。

- LLM在训练的时候使用了大量的网络数据，它如何能规避回答这类问题呢？
- 能给出这样的回答，基于两方面的原因：
 - 训练时过滤掉了负面数据
 - 使用基于人类反馈的强化学习方法对模型进行微调

基于人类反馈的强化学习微调（RLHF）

- LLM发展的最初一段时间里，各种LLM根据人类输入提示（prompt）生成多样化文本的能力令人印象深刻。然而，对生成结果的评估是主观和依赖上下文的，例如，我们希望模型生成一个有创意的故事、一段真实的信息性文本，或者是可执行的代码片段，这些结果难以用现有的基于规则的文本生成指标(如 BLEU 和 ROUGE)来衡量。除了评估指标，现有的模型通常以预测下一个单词的方式和简单的损失函数（如交叉熵）来建模，没有显式地引入人的偏好和主观意见。
- 如果我们用生成文本的人工反馈作为性能衡量标准，或者更进一步用该反馈作为损失来优化模型，那不是更好吗？这就是 RLHF 的思想：使用强化学习的方式直接优化带有人类反馈的语言模型。RLHF 使得在一般文本数据语料库上训练的语言模型能和复杂的人类价值观对齐。



- 强化学习的基础知识在后续章节中详细分析。

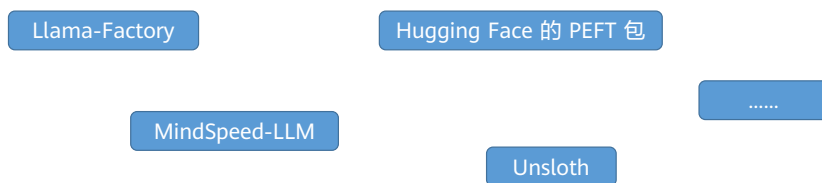
- <https://huggingface.co/blog/zh/rlhf>

目录

1. 模型微调技术
2. 大模型微调常用工具

大模型微调工具

- 当下，大模型的微调技术众多，即使是备受欢迎的LoRA算法也有较多的变体。如果全部实现每一种算法，是非常耗时耗力的，并且微调的底层代码设计大量复杂的逻辑。比如数据预处理（清洗数据、格式转换、分批次输入）、模型加载（连接预训练权重）、参数优化（选择合适的优化器、学习率策略）、训练监控（记录loss、防止过拟合），甚至还要处理硬件加速（并行计算）等。而这套逻辑是通用的，所以本着软件开发“不重复造轮子的”理念，可以将每种微调算法封装成API，甚至做成工具，来提高开发效率，同时这些工具经过众多开发者的使用，功能相对稳定，进一步提高开发的效率。
- 常用的微调库或工具有哪些呢？



LLaMA-Factory

- LLaMA Factory 是一个简单易用且高效的大型语言模型（Large Language Model）训练与微调平台。通过 LLaMA Factory，可以在无需编写任何代码的前提下，在本地完成上百种预训练模型的微调。

兼容模型多

适配 LLaMA、Mistral、Qwen、ChatGLM、LLaVA 等 100 + 主流模型，支持文本、图像、音视频多模态任务。

参数高效微调

集成 LoRA、QLoRA、AdaLoRA 等技术，仅需调整 0.1%-1% 参数即可达到全量微调效果。同时支持AdaLoRA，根据任务复杂度自动分配参数量，长文本任务中参数量减少 30%，同时保持性能。

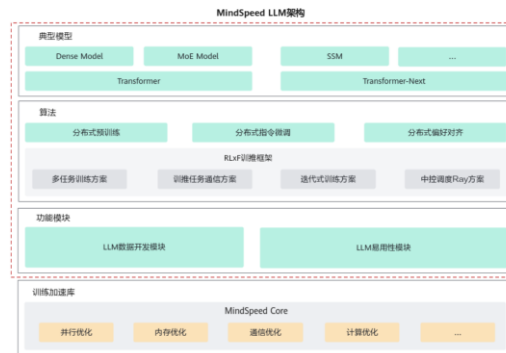
资源优化与加速

自动适配 FP16/BF16 精度，结合梯度checkpoint技术，显存占用降低 50%。集成 DeepSpeed，通过 ZeRO 技术实现显存优化，支持多机多卡训练万亿参数模型。

- <https://github.com/hiyouga/LLaMA-Factory>

MindSpeed-LLM

- MindSpeed-LLM是针对昇腾硬件推出的大语言模型套件，提供了开箱即用的主流开源大模型，支持全参微调、低参微调等算法，提升用户在昇腾硬件上高效、便捷地开发与使用大型语言模型的能力。



- https://www.hiascend.com/doc_center/source/zh/Pytorch/700/modthirdparty/Mindspeedguide/mindspeed_0063.html

Hugging Face - PEFT

- PEFT是Hugging Face开源的参数高效微调标准库，专为大模型轻量化微调设计，通过标准化接口整合主流PEFT技术，解决大模型微调时显存占用问题，提高训练效率。
- PEFT集成了多种参数高效微调方法，包括常见的经典微调方法。例如LoRA类、Prefix-tuning等。
- 与Hugging Face生态系统无缝集成：PEFT与Transformers、Diffusers和Accelerate库深度集成，为大模型的加载、训练和推理提供了更快捷、简便的解决方案。

- <https://huggingface.co/docs/peft/accelerate/deepspeed>

Unslloth

- Unslloth是一个专为高效训练设计的开源框架，核心价值在于通过底层优化大幅降低显存占用和显著提升训练速度，尤其适合资源受限的单卡环境。以下是其核心功能与技术特性的深度解析：

高效微调

支持4位量化，将模型参数压缩至原体积的 1/4。通过动态重新计算前向激活值，节省 50% 以上显存。Unslloth使LLM的微调速度提升2 - 5倍，显存使用量减少约80%，且准确度无明显下降。

广泛的模型支持

Unslloth兼容多种主流LLM，如 Llama-3、Mistral、Phi-4等，适用于不同规模的任务需求。

参数高效微调（PEFT）集成

默认集成 LoRA（低秩矩阵分解）和 QLoRA（4 位量化 LoRA），仅需调整 0.1%-1% 参数即可达到全量微调效果。

AdaLoRA 动态优化：根据任务复杂度自动调整 LoRA 秩（rank），在长文本任务中参数量减少 30%，同时保持性能.....

思考题

1. （单选题）以下哪一种微调算法修改了模型本身架构？
- A. Prompt-tuning
 - B. Adapter-tuning
 - C. LoRA
 - D. QLoRA

- B

本章总结

- 本章首先介绍了模型微调技术的起源及其重要性，然后着重讲解了大模型的微调方法，按照不同参数增加类型分析了多种具有代表性的微调方法；最后介绍了几款主流的微调工具。

学习推荐

- 昇腾社区
 - <https://www.hiascend.com/>
- MindSpore官网
 - <https://www.mindspore.cn/>

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

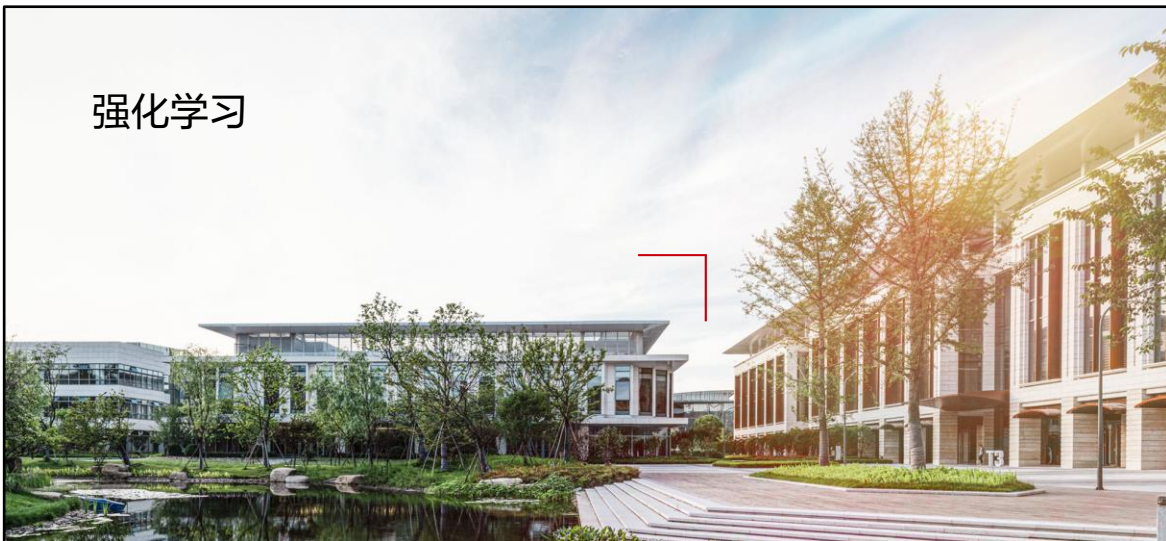
Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



强化学习



前言

- 强化学习作为连接智能体与复杂环境交互的核心范式，正以前所未有的速度重塑着决策优化、自主控制等领域的边界。强化学习通过“试错-学习-优化”的闭环机制，为智能体赋予了从经验中自我进化的能力。
- 本章节将围绕强化学习基础知识展开，介绍强化学习与大模型的协同关系，同时介绍两个代表性算法：PPO（Proximal Policy Optimization）与GRPO（Generalized Reward Policy Optimization）的原理。

目标

- 学完本课程后，您将能够：
 - 了解强化学习基础知识；
 - 了解强化学习算法分类；
 - 了解RLHF的功能及流程；
 - 了解PPO和GRPO算法工作流程。

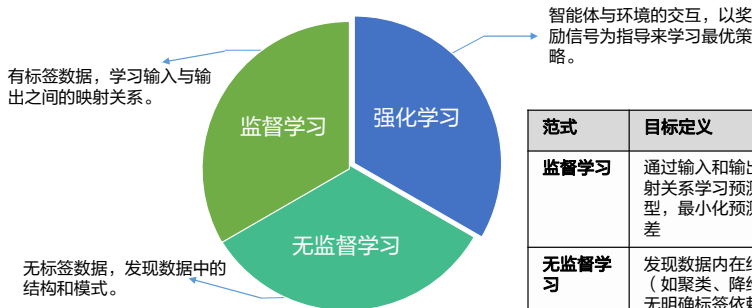
目录

1. 强化学习基础知识
2. 强化学习在大模型中的应用

什么是强化学习

- 强化学习（Reinforcement Learning, RL）是机器学习的重要分支，其核心机制是通过智能体（Agent）与环境（Environment）的动态交互，学习实现长期目标的最优策略。与监督学习依赖标注数据、无监督学习聚焦数据结构不同，强化学习以奖励信号为唯一导向，通过“试错 - 反馈”循环驱动决策优化。

和监督及无监督学习的区别



范式	目标定义	核心驱动力	典型案例
监督学习	通过输入和输出映射关系学习预测模型，最小化预测误差	标注数据中的标签（如“猫/狗”分类标签）	图像识别、情感分类、股价预测
无监督学习	发现数据内在结构（如聚类、降维），无明确标签依赖	数据分布特征或潜在模式	用户画像、异常检测、主题模型
强化学习	通过与环境交互学习最优策略，最大化长期累积奖励	延迟的奖励信号（如游戏得分、能耗效率）	机器人控制、自动驾驶、资源优化

强化学习的基本组成



- 两个对象：
 - 智能体 (Agent)：执行动作并学习的实体。
 - 环境 (Environment)：智能体所处的外部世界，与智能体进行交互。
- 主要元素：
 - 状态 (State)：描述环境当前情况的信息。
 - 动作 (Action)：智能体在给定状态下可以采取的行为。
 - 奖励 (Reward)：环境对智能体动作的即时反馈，用于衡量动作的好坏。

- 基本交互循环可分解为：
 - 状态感知：智能体接收环境状态信息（如棋盘局面、机器人传感器数据）
 - 动作执行：基于策略选择动作（如落子位置、电机控制指令）
 - 环境反馈：获得即时奖励（胜负得分、能耗效率）和下一状态
 - 策略更新：通过价值函数评估长期收益，优化决策模型

价值函数与策略

- 在强化学习（Reinforcement Learning, RL）中，策略（Policy）和价值函数（Value Function）负责指导智能体（agent）如何在环境中作出决策以及评估这些决策的优劣。

策略

- 输入是环境的状态，输出是在该状态下应采取的动作或动作的概率分布。
- 策略指导着智能体应该如何行动。

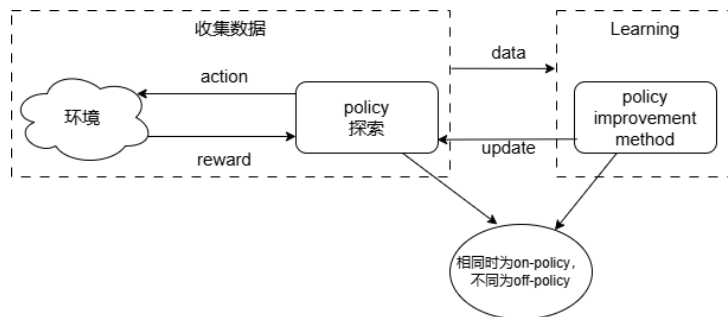
$$\pi(s): S \rightarrow A$$

其中 S 是状态集合， A 是行为集合。给定一个状态 s ，策略函数会返回一个具体的行为 a 。

价值函数

- 用于评估在某一状态下采取某一策略所能获得的长期回报（即奖励的累积值）。它通常用于选择最优策略。在一定程度上，价值函数相似于机器学习中的损失函数，但方向相反。

强化学习学习流程

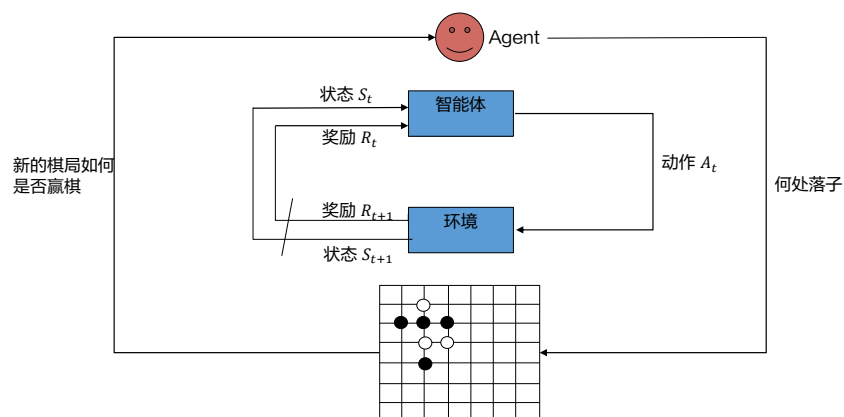


On-policy和Off-policy

- 在强化学习中，On-policy和Off-policy是两类核心学习范式，区别在于数据生成策略与目标策略是否一致。

维度	On-policy	Off-policy
策略关系	生成数据的策略与目标策略 相同	生成数据的策略与目标策略 不同
数据利用	仅使用当前策略产生的数据	可利用历史数据或其他策略生成的数据
探索机制	需通过策略自身探索（如 ϵ -greedy）	可通过独立行为策略探索

强化学习模型 - 下棋

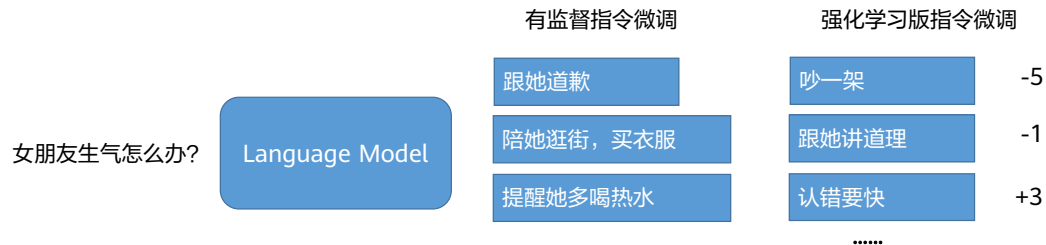


目录

1. 强化学习基础知识
2. 强化学习在大模型中的应用

Alignment

- 用人类更喜欢的数据去做训练（SFT+RLHF）：
 - 真实性：是虚假信息还是误导性信息？
 - 无害性：它是否对人或环境造成身体或精神上的伤害？
 - 有用性：它是否解决了用户的任务？



为什么是强化学习

整体影响

监督学习针对单个词元进行反馈，其目标是模型针对给定输入后得到确切答案；强化学习则是针对整个输出结果进行反馈而不针对特定词元。反馈的颗粒度不同使得强化学习更适合大语言模型。

幻觉

监督学习会促使模型给出一个结果，即使是在模型不包含或不知道答案的情况下；而强化学习则可以通过答案的奖励分数，促使模型放弃低分值的答案。

多轮对话

多轮对话的交互情况及是否达成最终目标，很难有效通过监督学习方法构建，但是通过强化学习构建奖励函数，可以通过整个对话的连贯性及背景对当前模型输出的优劣做判断。

强化学习在大模型中的应用

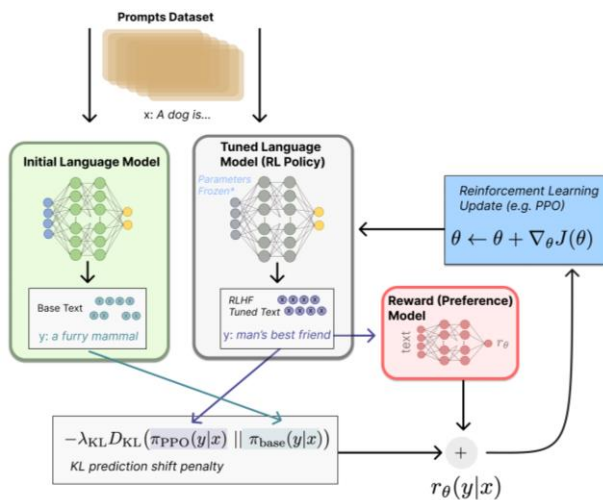
在特定领域或指令和人类
示范语料库上微调预训练的
LLM

收集人类标注的数据集，
训练一个奖励模型

使用 RL (例如 PPO)，用此
数据集和奖励模型进一步
微调 LLM



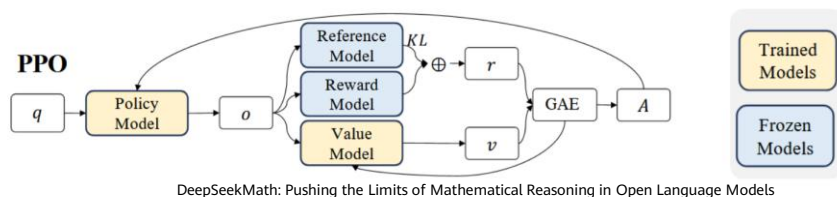
基于人类反馈的强化学习RLHF



- 将微调任务表述为RL问题：
 - Policy是一个接受提示并返回一系列文本（或概率分布）的LM。
 - 策略的行动空间是LM的词表对应的所有词元。
 - 奖励函数是偏好模型和策略转变的结合。

- 通过PPO算法优化SFT模型的策略：计算Actor输出response和输入语料的回答部分的交叉熵损失函数，用来在PPO梯度中加入预训练梯度，以保持语言模型原有性能防止遗忘。

RHLF-PPO中的四个模型



DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models

- Reference Model，参考模型，防止强化学习训练过程模型跑偏。
- Policy (Actor) Model，目标模型，给定任务输出回答。
- Reward Model，奖励模型，通过人类反馈的数据训练出的模型，用于后续评估目标模型的输出。
- Value (Critic) Model，输出基于当前状态对未来奖励累计的期望值。

- 这四个模型可以灵活使用。一般来说可以基于SFT后的模型构造。例如reference模型是SFT后的模型，policy模型基于SFT模型微调，reward模型和value模型在SFT模型的基础上加一个线性层微调。

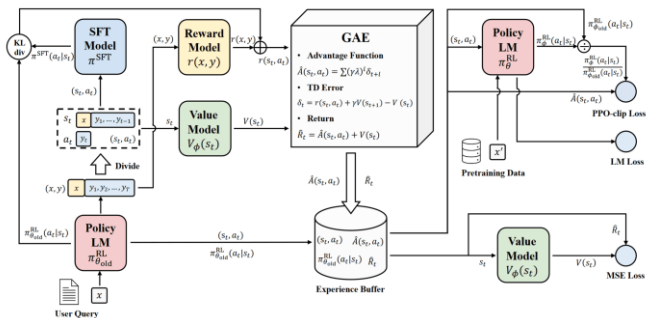
PPO计算流程

Secrets of RLHF in Large Language Models
Part I: PPO

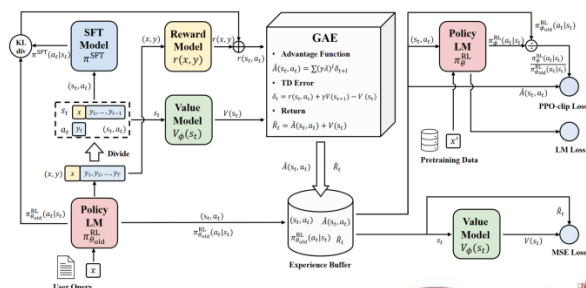
强化学习目标: $\bar{R}_\theta = E_{\tau \sim \pi_\theta(\tau)}[R(\tau)] = \sum_\tau R(\tau) \pi_\theta(\tau)$

$$\begin{aligned} \nabla \bar{R}_\theta &= \sum_\tau R(\tau) \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) = E_{\tau \sim \pi_\theta(\tau)}[R(\tau) \nabla_\theta \log \pi_\theta(\tau)] \approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla_\theta \log \pi_\theta(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla_\theta \log \pi_\theta(a_t^n | s_t^n) \end{aligned}$$

$R(\tau^n) - b \rightarrow \nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} A_t(a_t^n, s_t^n) \nabla_\theta \log \pi_\theta(a_t^n | s_t^n)$



PPO计算流程



难以算得准确值，可以用广义优势估计（GAE）

$$A_t(s_t, a_t) := Q(s_t, a_t) - V(s_t)$$

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ → Value model输出结果

→ Reward model输出结果，一般把一整个句子的 reward 附给最后一个 token。

- 与Q-函数和V-函数的关系: 优势函数实际上是Q-函数（动作价值）与V-函数（状态价值）之间的差异。如果 $A(s, a)$ 是正的，说明在s状态下采取a动作比当前策略更优；如果是负的，说明比当前策略更差。

重要性采样

- 重要性采样（Importance Sampling），用于估计某个分布下的期望值，尤其适用于直接采样困难或成本高昂的场景。其核心思想是通过引入一个易于采样的替代分布，对样本进行加权以修正分布差异。

假设目标分布为 $p(x)$ ，替代分布为 $q(x)$ ，需估计函数 $f(x)$ 在 $p(x)$ 下的期望值：

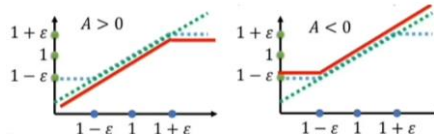
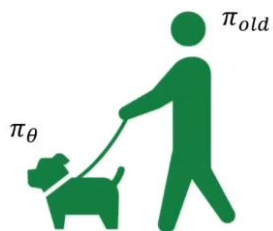
- $$E_{x \sim p(x)}[f(x)] = \int f(x)p(x)dx$$

直接采样 $p(x)$ 困难，可通过从 $q(x)$ 中采样，并计算重要性权重 $w(x) = \frac{p(x)}{q(x)}$ ，将期望转化为：

$$E_{x \sim p(x)}\left[\frac{p(x)}{q(x)}f(x)\right] \approx \frac{1}{N} \sum_{i=1}^N w(x_i)f(x_i)$$

其中 $x \sim q(x)$ ，权重 $w(x_i)$ 反映了 $p(x)$ 与 $q(x)$ 的概率比值。

PPO训练



$$\mathcal{J}_{PPO}(\theta) = \mathbb{E}[q \sim P_{sf_t}(Q), o \sim \pi_{\theta_{old}}(O|q)] \frac{1}{|O|} \sum_{t=1}^{|O|} \min \left[\frac{\pi_{\theta}(o_t|q, o_{<t})}{\pi_{\theta_{old}}(o_t|q, o_{<t})} A_t, \text{clip} \left(\frac{\pi_{\theta}(o_t|q, o_{<t})}{\pi_{\theta_{old}}(o_t|q, o_{<t})}, 1 - \epsilon, 1 + \epsilon \right) A_t \right]$$

新旧模型输出token概率比，
用于控制强化学习更新幅度。

优势函数，衡量输出下一个token
可以获得多少相对优势: $A_t(s_t, a_t) := Q(s_t, a_t) - V(s_t)$

KL散度约束

$$r_t = r_t(a_t, s_t) - \beta \log \frac{\pi_{\theta}(a_t|s_t)}{\pi_{ref}(a_t|s_t)}$$

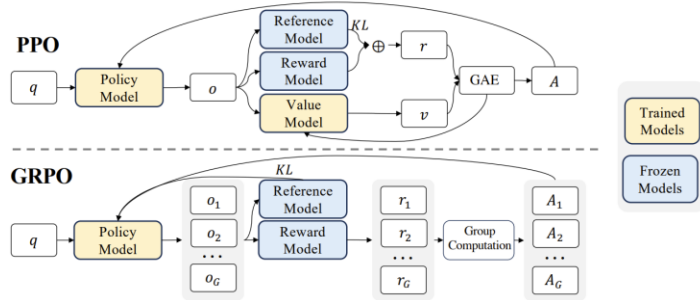
$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

状态-动作
价值函数

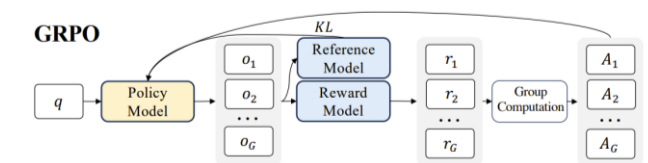
状态价值
函数

GRPO (1)



- GRPO省略了Value Model。
- GRPO reward计算, 改成了一个q生成多个r, 然后reward打分。
- PPO优势函数计算时, KL是包含在GAE内部的。GRPO直接挪到了外面, 同时修改了计算方法。

GRPO (2)



GRPO的Advantage function中不使用价值模型，而是对于每个问题生成多个输出结果，通过减去组内均值除以组内标准差（z-score标准化）的方式算出相对优势。

$$\mathcal{J}_{PPO}(\theta) = \mathbb{E}[q \sim P_{\text{split}}(Q), o \sim \pi_{\theta_{\text{old}}}(O|q)] \frac{1}{|o|} \sum_{t=1}^{|o|} \min \left[\frac{\pi_{\theta}(o_t|q, o_{<t})}{\pi_{\theta_{\text{old}}}(o_t|q, o_{<t})} A_t, \text{clip} \left(\frac{\pi_{\theta}(o_t|q, o_{<t})}{\pi_{\theta_{\text{old}}}(o_t|q, o_{<t})}, 1 - \epsilon, 1 + \epsilon \right) A_t \right]$$

$$\mathcal{J}_{GRPO}(\theta) = \mathbb{E}[q \sim P(Q), \{o_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(O|q)]$$

$$\frac{1}{G} \sum_{i=1}^G \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \left(\min \left[\frac{\pi_{\theta}(o_{i,t}|q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q, o_{i,<t})} \hat{A}_{i,t} \text{clip} \left(\frac{\pi_{\theta}(o_{i,t}|q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q, o_{i,<t})}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_{i,t} \right] - \beta \mathbb{D}_{KL}[\pi_{\theta} || \pi_{\text{ref}}] \right)$$

$$A_t(s_t, a_t) := Q(s_t, a_t) - V(s_t) \rightarrow \hat{A}_{i,t} = \tilde{r}_i = \frac{r_i - \text{mean}(r)}{\text{std}(r)}$$

- GRPO 的本质思路：通过在同一个问题上生成多条回答，把它们彼此之间做“相对比较”，来代替传统 PPO 中的“价值模型”

思考题

1. 在PPO训练过程中，通过以下哪个选项可以使得策略模型可以用历史数据学习？
- A. GAE
 - B. 重要性采样
 - C. 模型并行
 - D. TD算法

- B

本章总结

- 本章节围绕强化学习基础知识展开，介绍了强化学习与大模型的协同关系及 PPO、GRPO 算法原理和在大模型中应用。

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2025。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

Copyright © Huawei Technologies Co., Ltd. 2025.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.



专业技术人员职业素养与组织赋能



前言

- 在技术高歌猛进、业务复杂多元的今天，我们坚信，卓越的专业追求绝非孤立存在，它既需要**内化于心的职业道德**——对专业诚信的坚守、对合规红线的敬畏、对技术伦理的审慎；也需要**外化于行的组织赋能**——打破技术孤岛，通过知识沉淀、经验传承，将个人能量转化为组织发展的动能。

目标

- 学完本课程后，您将能够：
 - 清晰界定技术应用的边界，识别风险；
 - 建立持续迭代的技术成长思维，以适应指数级发展趋势；
 - 掌握标准化带教方法，将个人技术传承延续。

目录

1. 职业道德
2. 终身学习
3. 培训指导

从业者的三重责任

三重责任

核心目标：明确责任边界与核心义务

1. **专业责任：**追求技术卓越，保持专业精进
2. **社会责任：**维护公共利益，坚守伦理底线
3. **带教责任：**传承知识与规范，赋能团队成长



三者间的关系

职业道德 → 技术技能 (约束关系)

职业道德为技术技能的应用划定边界。如生成式AI伦理约束AI模型开发方向；网络安全法规约束网络技术操作

培训指导 → 技术技能 (赋能关系)

培训指导为技术技能的习得提供方法论。如利用知识图谱工具构建能力图谱指导学习路径；“先培训再上岗”确保技术技能达标

技术技能 → 职业道德 (实践关系)

技术技能是践行职业道德的载体。如只有掌握了数据安全技术，才能真正保护用户隐私

技术技能 → 培训指导 (反馈关系)

技术技能的掌握程度反馈培训指导的有效性，驱动培训方案迭代

职业道德 ↔ 培训指导 (融合关系)

培训指导本身需遵循职业道德；同时培训指导也是传播职业道德的主渠道

职业道德核心原则解析

一般伦理原则

核心目标：确立社会底线

1. **人类福祉**：将公众安全、健康及福祉置于技术发展的首位
2. **诚实守信**：保持专业诚信，不隐瞒已知风险，不误导用户
3. **公平公正**：避免算法歧视，确保技术应用的包容性与公平性
4. **避免伤害**：预防技术滥用，主动规避对社会的潜在负面影响
5. **尊重权益**：尊重法律、知识产权以及用户隐私，严守数据边界

职业伦理原则

核心目标：确立行业标准

1. **追求卓越**：持续精进技术能力，保持对高质量交付的追求
2. **保持专业**：遵循行业规范，维护职业声誉，保持客观中立
3. **同行评议**：确保研究成果的质量和可信度，维护学术和职业的公平和正义
4. **尽职尽责**：对岗位负责，确保系统稳定与安全
5. **维护专业形象**：维护计算机行业的形象与声誉，促进计算机技术的普及及推广

一般伦理原则与职业伦理原则的双重规范

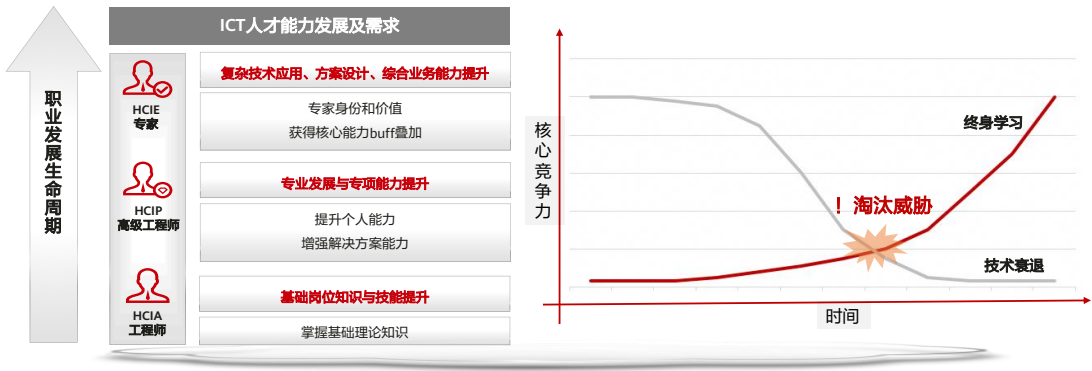
目录

1. 职业道德
- 2. 终身学习**
3. 培训指导

数智升级，对人才培养与技能评价提出更高要求



华为职业认证：终身学习



技术发展现状

- 迭代加速、跨界融合、技术颠覆成为常态，停止学习即被淘汰
- 从“碎片化输入”到“知识晶体”，真正的学习，是将外界的碎片重构成自身的“护城河”

目录

1. 职业道德
2. 终身学习
- 3. 培训指导**

角色升级：从技术能手到培训导师

“技术工作是把事做好，培训带教是把人带会”

	技术能手	培训导师
核心角色	解决特定技术问题	知识传递者、能力引导者、问题解决者
交付成果	高质量的代码与系统	独立规范上岗的专业人才
核心能力	专业硬技术、排障能力	会梳理知识、会标准示范、会纠错复盘



带教原则

实用为先

贴合岗位真实场景，不讲空话

实操为主

重动手，重细节，真操实练，夯实技术

问题导向

解决实操卡点与上岗痛点

全程闭环

课前有备

定目标，梳理重难点

课中有方

示范、纠错、控场、盯安全

课后有落

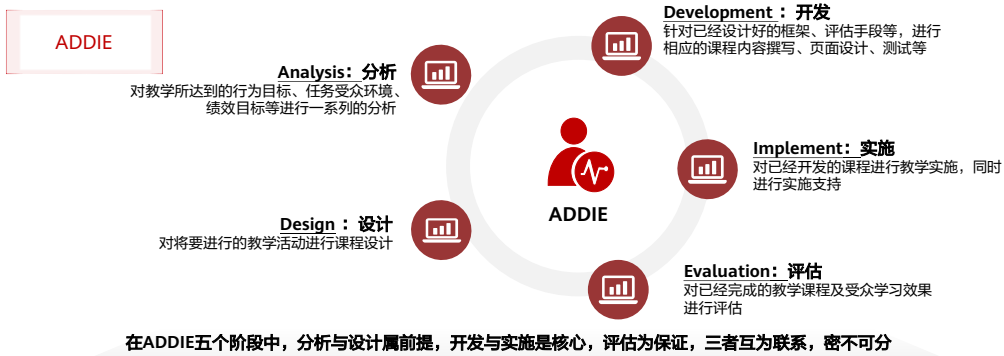
答疑、考核、建档、跟进成长

培训与指导

ADDIE：将A、D、D、I、E五个步骤综合起来考虑，形成系统，保障各个环节能进行及时有效的评估

SAM模型：提倡“快速迭代、小步快跑”，分为三阶段：准备、迭代设计、迭代开发

6D法则：以终为始，把培训的生命周期向后延伸，将“培训”转化为“商业结果”的全面流程



Analysis: 分析阶段

分析阶段是首要的环节，是确定培训目标、设计培训计划的前提，也是决定培训是否开展的关键因素

在培训的各个阶段都要有分析，通过问卷、约谈、电话等各种形式，了解对象、组织、课程的各方面需求



需求分析 (WHY/WHAT) :

组织需求 (公司层面) :

公司的战略目标是什么?
目前阻碍目标实现的“痛点”是什么?

岗位需求 (标准层面) :

目标岗位需要具备哪些核心胜任力?
标准的操作流程 (SOP) 或优秀员工的行为典范是怎样的?



需求对象分析 (WHO) :

基本属性 (画像) :

学员的年龄分布、学历、司龄、目前的岗位层级。

认知与经验起点:

他们对这个主题已经了解多少?
他们日常遇到的实际工作场景和痛点是什么?

学习动机与风格:

他们是主动想学，还是被公司强迫来的?



培训环境分析 (Where & How)

面授课堂: 教材需要配套 PPT、讲师手册、学员手册、通关演练通关单。

线上微课/E-learning: 教材需要拆解为结构化的“脚本”或“脚本大纲”，方便录制音频或制作动画。

混合式培训 (OMO) : 线上学理论 (教材偏精简、清单化)，线下做演练 (教材偏案例、工具化)。



培训内容分析 (Which)

知识类: 讲概念、原理 (要少而精)。

技能类: 讲步骤、方法、流程 (要多、要细、要有标准)。

态度类: 讲观念转变、企业文化 (要通过故事和案例来感染)。



培训资源分析 (With What)

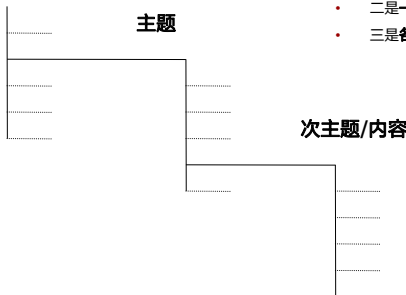
师资资源: 有没有内部的业务专家 (SME) 作为内容源头?

素材资源 (现成原料) : 公司现有的 SOP 手册、过往的失败案例库、竞品资料、行业标准等。

Design：设计阶段

设计阶段要制订一份完成培训项目所必需的说明提纲。基本内容包括必须制定的目标、目的和评估任务
基于培训项目的相关决策和其它实施问题的体系结构，确定各项任务的顺序

单元



课程单元结构的关系

- 一是**相对独立型**，各学习单元在位置上可以互换
- 二是一个单元是构成另一个单元的基础，这种类型在序列上极为严密
- 三是各单元学习内容的联系成**综合型**，即有的是相互独立关系，也有的是严格序列关系

主题的组织方式 - 螺旋式

- 即根据培训对象的智力发展水平，让学员尽早有机会在不同程度上去接触和掌握某门学科的基本结构，以后随着学员在智力上的成熟，围绕基本结构不断加深内容深度，使学员对知识有更深刻和有意义的理解。

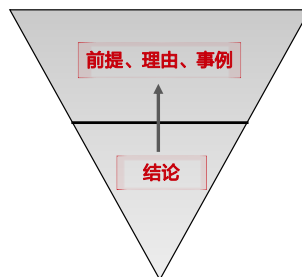
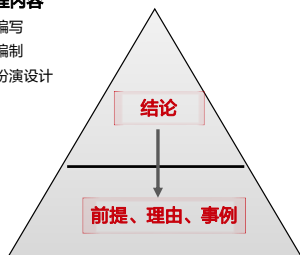
Development : 开发阶段

开发阶段的要点在于**开发创建教学材料**的方法和流程。根据之前总结的各干系方需求，开发培训教材（资料），设计好阶段性评估文件等。在这一阶段中，内容和材料从概念设计转变为可交付成果。

1: 收集素材

2: 组织课程内容

- 内容编写
- 案例编制
- 角色扮演设计



检查错误

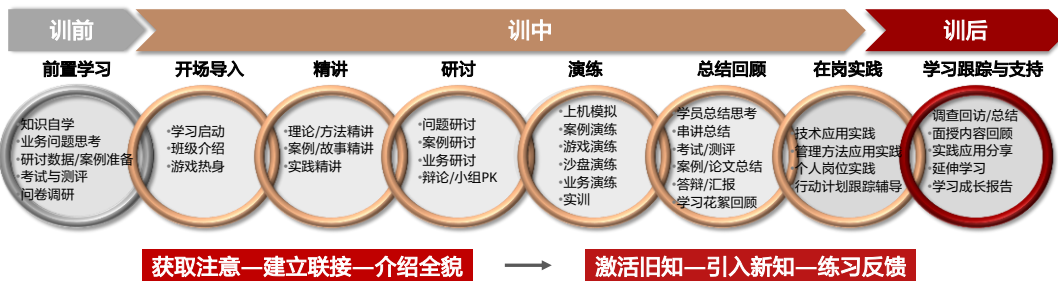
- 标题
- 段落
- 句子
- 表格
- 图形
- 缩略语/术语
- 数与数量
- 标点符号
- 单位符号

思考时多按金字塔顺序，但表述要用倒金字塔顺序

Implement: 实施阶段

- **教学程序**: 从把学员引入某个专题, 到掌握某个目标为止向学员呈现教学时教学活动的程序、步骤
 - **教学方式**: 呈现教学时运用的手段、方法, 如分组讨论、独自阅读、个案研究、讲授、计算机模拟、小组协作项目等
- 认知学习的一个关键概念是在信息从**工作记忆**迁移至**长期记忆**时, 对信息进行提取或演练, 从而强化长期记忆

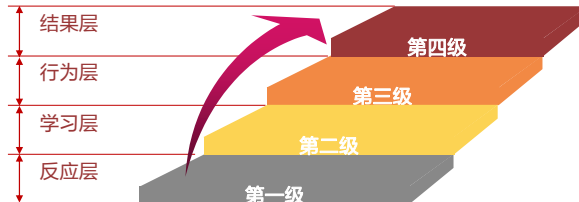
“三段八环六素”教学理论, 把学习过程分片, 根据不同技术场景, 适配不同教学活动, 提升学习效果。



Evaluation: 评估

主流评估模型：柯氏四级、菲利普斯五级、考夫曼五级、CIPP 模型等
行业地位：历经行业检验，**柯氏四级**时至今日仍是不可动摇的经典基石模型

层次	重点描述	衡量方法	评价人员
反应层	受训者满意程度：喜欢该项目吗？对培训人员和设施有什么意见吗？课程有用吗？	问卷	学员
学习层	受训者在培训前后，知识、技能、态度、行为方式方面的收获：有多大程度的提高？	笔试、实地操作和工作模拟	学员和培训人员
行为层	培训后，受训者行为的改进：行为有无不同？他们在工作是否使用了在培训中所学的知识？	问卷调查、面谈、直接观察、业绩检测、专题研究、实践计划	学员、学员上级、下属、同事、客户
结果层	受训者获得的经营业绩：组织是否因为培训变得更好？	问卷调查、业绩检测、业绩目标法	学员、学员上级



反应层评估 (Reaction)：评估被培训者的满意程度
学习层评估 (Learning)：测定被培训者的学习获得程度
行为层评估 (Behavior)：考察被培训者的知识运用程度
结果层评估 (Result)：计算培训创出的经济效益

思考题

1. （单选题）项目经理小张突然接到紧急通知，需要当晚在公司内网加班处理一份涉及公司**核心商业秘密与关键技术参数**的涉密文件。由于公司保密机房的专用有线计算机键盘出了故障，为了不耽误进度，小张决定采取一些临时措施。请问以下哪**一项行为是明确禁止**的？
 - A. 临时将这份涉密文件拷贝到自己带有 Wi-Fi 芯片和蓝牙功能的个人便携式笔记本电脑上处理。
 - B. 立即联系IT安全部门，申请对当前的涉密系统环境进行一次全面的安全性评估。
 - C. 调整工位，移步到未链接互联网、且切断一切外网物理信号的涉密专用计算机上处理。
 - D. 确保待在公司规定的受控办公室内，在无无关人员围观的闭环环境下处理该涉密资料。
2. （判断题）培训完成后，进行了结业考试，考试通过就代表学员完全掌握了这项技能。

- A
- 错

本章总结

- 本章讲述了从业者的成长路径
 - 坚守底线：牢记行为红线，确保技术向善
 - 持续学习：践行终身学习，应对瞬息万变
 - 培训指导：应用教学实践，传承优秀同路人

学习推荐

- 《 中 国 计 算 机 学 会 职 业 伦 理 与 行 为 守 则 》 :
https://www.ccf.org.cn/About_CCF/CCF_Constitution/2023-07-26/794450.shtml

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。
Bring digital to every person, home, and
organization for a fully connected,
intelligent world.

版权所有 © 华为技术有限公司 2026。

保留一切权利。

未经华为事先书面许可，任何人不得对本资料进行复制、修改、
整合、改编、翻译，也不得将本资料或其任何部分或基于本资
料的衍生作品提供、出售、转让、再许可给任何人用于华为
许可以外的任何其他用途。

Copyright © Huawei Technologies Co., Ltd. 2026.

All Rights Reserved.

Without prior written permission of Huawei, no one is
allowed to reproduce, modify, integrate, adapt, translate
this Document, or provide, sell, transfer, sublicense this
Document or any part of this Document or its derivative to
any third party for any other purpose otherwise permitted
by Huawei.

