机器学习系统 2026春

第四章 Transformer和大语言模型

张燕咏 讲席教授 yanyongz@ustc.edu.cn
张午阳 特任教授 wuyangz@ustc.edu.cn

中国科学技术大学
University of Science and Technology of China

一条ChatGPT回复背后的计算量

为什么: 理解Transformer架构, 才能理解这些巨大的数字从何而来

GPT-4 单次推理

~数千亿 FLOPs（MoE架构）

≈ H100 集群单次前向传播(prefill) ~数秒 (实际生成受限于显存带宽)

KV Cache (128K context)

单条请求即可达数十GB

batch增大后可能超过模型权重

这门课要回答的问题:

- Transformer为什么能取代RNN/CNN?
- 注意力机制的计算量从何而来?
- KV Cache为什么这么大? 怎么优化?
- Scaling Laws如何指导模型设计?
- 推理模型(o1/R1)带来了什么新挑战?

关键要点: Transformer是当前AI的基石架构，理解其系统特性, 是优化训练和推理的前提

课程路线图：3节课 × 45分钟

第1节

从RNN到Transformer架构全解析

- 为什么需要Transformer?
- 自注意力机制详解
- MHA, FFN, LayerNorm
- 位置编码
- 计算量分析
- PyTorch代码实现

第2节

设计演进从原始Transformer到LLM

- Encoder/Decoder/Both
- RoPE旋转位置编码
- KV Cache内存分析
- MQA → GQA优化
- SwiGLU, RMSNorm
- Scaling Laws & LLM发展

第3节

LLM能力、推理模型与系统挑战

- 涌现能力 & ICL
- RLHF → DPO对齐
- o1 & DeepSeek-R1推理模型
- 多模态LLM
- MoE稀疏计算
- Prefill vs Decode系统分析

关键要点： 从架构原理 → 设计演进 → 系统挑战，三节课构建完整的Transformer知识体系

PART 01

为什么需要Transformer?

从RNN的系统瓶颈到注意力机制的诞生

自然语言处理任务总览

判别任务 (Discriminative)

- 情感分析: 这部电影好看吗? → 正面/负面
- 文本分类: 这是体育/科技/财经新闻?
- 文本蕴含: A能推出B吗?
- 核心: 输入序列 → 类别标签

生成任务 (Generative)

- 语言建模: 预测下一个词
- 机器翻译: 英文→中文
- 文本摘要: 长文→要点
- 核心: 输入序列 → 输出序列

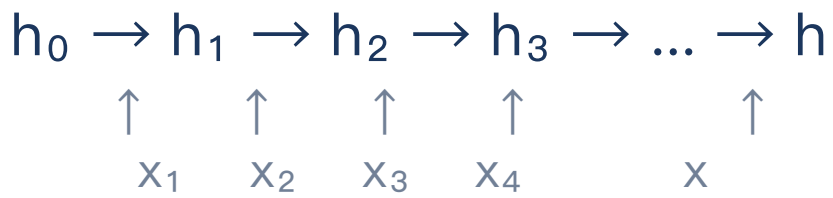
系统视角: 判别任务需要编码整个序列(双向), 生成任务需要逐步解码(单向), 不同需求催生不同架构

关键要点: 判别和生成是两类核心NLP任务, 它们对模型架构提出不同的计算需求

RNN: 串行计算的系统代价

问题: RNN的隐状态必须串行计算, 无法利用GPU的并行能力

RNN计算链:



- **串行依赖**
- h = f(h -1, x), 状态t依赖状态t-1
- N个token需要N步顺序计算
- **长期依赖困难**
- 建模token i和j的交互需要 O(|i-j|) 步
- 梯度消失/爆炸 (LSTM部分缓解)

GPU利用率:

RNN训练

GPU利用率极低: 大量时间在等待前一步完成
数千个CUDA核心只有少数在工作

理想的并行模型

所有token同时处理
数千核心满载运行
训练速度提升数百倍

- 语言可能具有长距离依赖:
- "The cat, which was sitting on the mat that I bought last week, is sleeping."

关键要点: RNN的串行本质是其最大系统瓶颈, 无法利用GPU并行计算, 且难以建模长距离依赖

CNN的尝试与局限

√ 优势: 可并行

1D卷积沿序列滑动
token间无依赖 → 可完全并行
GPU利用率大幅提升

✗ 局限: 感受野有限

单层卷积核大小=k, 只能看k个token
捕捉距离d的关系需要 O(d/k) 层
长距离交互 → 需要非常深的网络

特性	RNN	CNN	理想方案 (?)
并行性	✗ O(n)串行	√ 完全并行	√ 完全并行
任意距离交互	O(n) 步	O(n/k) 层	O(1)
全局上下文	√ (理论上)	✗ 受限于层数	√
GPU利用率	低	高	高

关键要点: CNN解决了并行性, 但上下文窗口受限, 我们需要一种同时具备并行性和全局上下文的机制

我们需要什么?

完全并行

所有token同时处理, 充分利用GPU

全局上下文

每个token直接看到所有其他token

O(1)任意距离

任意两个token的交互不需要中间步骤

自注意力机制 (Self-Attention)

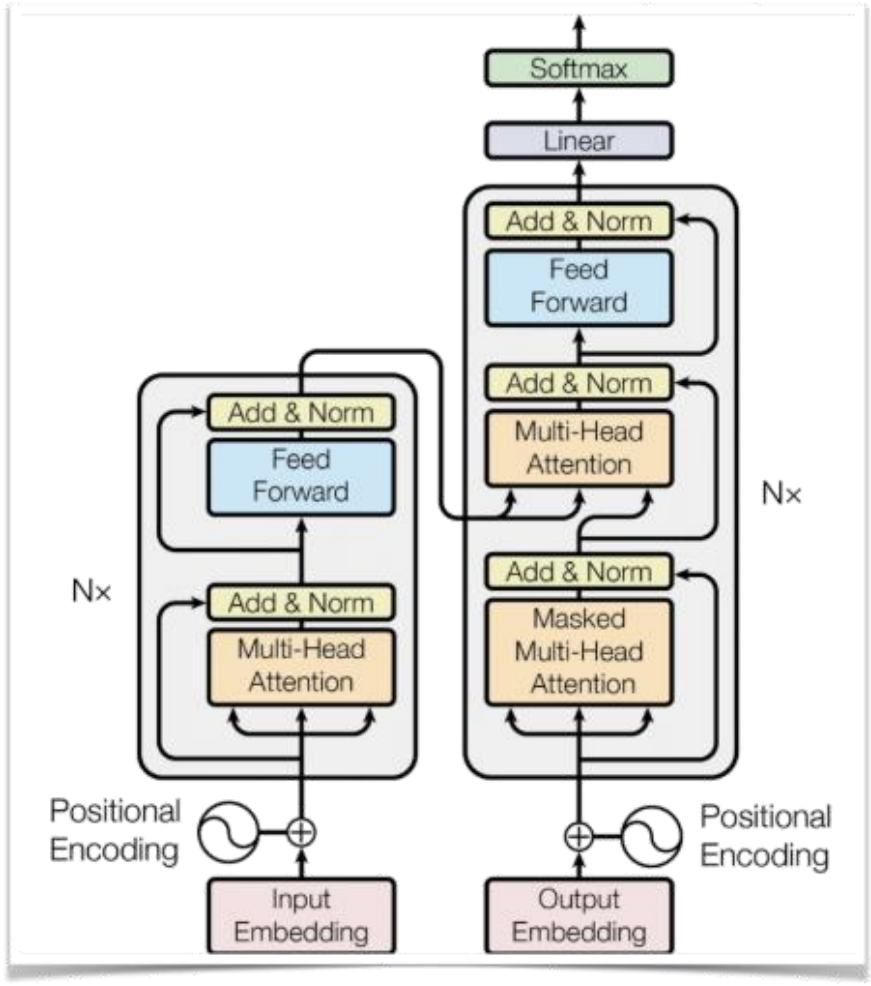
$$\text{Attention}(Q,K,V) = \text{Softmax}(QK^T/\sqrt{d}) \cdot V$$

关键要点: 自注意力同时满足三个需求: 并行计算 + 全局上下文 + O(1)任意距离交互, 代价是 $O(n^2)$ 复杂度

PART 02

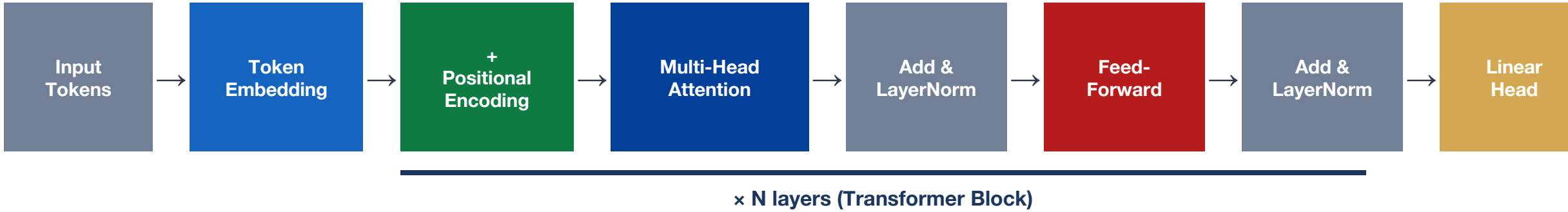
Transformer架构逐层拆解

从Tokenization到最终预测，理解每个组件的设计动机



Transformer全景架构

"Attention Is All You Need" (Vaswani et al., 2017)，以Decoder-only为例展示核心组件



Tokenization

单词→子词token (BPE)

Embedding

token→d维连续向量

Self-Attention

全局上下文建模 $O(n^2d)$

FFN

非线性特征变换，参数占~2/3

LayerNorm

训练稳定性保障

Residual

梯度直通，深层训练

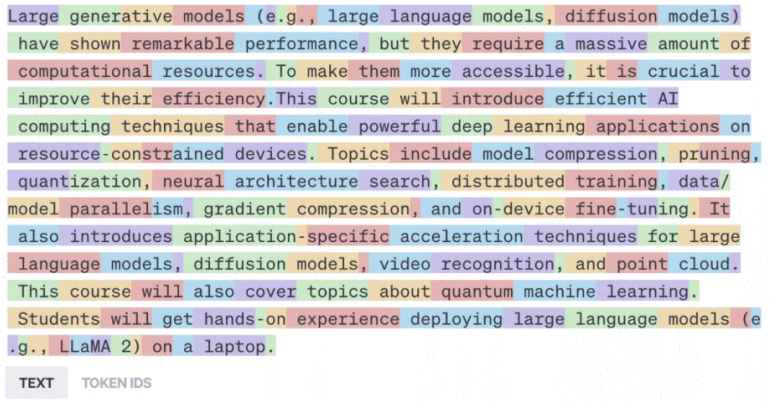
关键要点: Transformer由6个核心组件构成，接下来逐一拆解每个组件的设计动机和计算特性

Tokenization: 从单词到Token

为什么: 直接用单词作为输入? 词汇表太大(>100万) → 嵌入矩阵参数爆炸, 且无法处理新词

- **Byte Pair Encoding (BPE)**
- 统计语料中最频繁的字符对, 逐步合并
- "unhappiness" → ["un", "happi", "ness"]
- 110个单词 → 162个token (约1.5倍)

- **系统影响**
- Vocab size决定嵌入矩阵大小: V × d_model
- LLaMA: 32K vocab × 4096 dim = 128M参数
- GPT-4: ~100K vocab → 更大的嵌入层
- 更大vocab = 更短序列 = 稀疏token = 训练不充分




```
# Tokenizer示例 (tiktoken)
import
enc = tiktoken.get_encoding("cl100k_base")

text = "Transformer改变了NLP"
tokens = enc.encode(text)
# [Trans, former, 改, 变, 了, NLP]
# 6个中文字 → 可能8–10个token

print(f"Text: {len(text)} chars")
print(f"Tokens: {len(tokens)} tokens")
```

模型	Vocab Size	方法
GPT-2	50,257	BPE
LLaMA	32,000	SentencePiece
LLaMA-3	128,000	tiktoken BPE

关键要点: Tokenization决定了序列长度和嵌入参数量， vocab越大,序列越短,但嵌入层越大,是一个系统权衡

Word Embeddings: 从离散到连续

为什么: 神经网络需要连续数值输入, one-hot编码太稀疏 (V维向量, 只有1个非零)

One-Hot 编码

"cat" → [0, 0, 1, 0, ..., 0]（V维）

- 维度 = 词汇表大小 (32K~128K)
- 极度稀疏, 无法表达语义相似性
- "cat"和"kitten"的向量正交!

词嵌入 (Word Embedding)

"cat" → [0.21, −0.35, 0.78, ...]（d维）

- 维度 = d_model (典型512~4096)
- 稠密向量, 语义相似→向量接近
- 通过查表实现: E[token_id] → d维向量

嵌入矩阵 E ∈ ℝ^{V×d}

```
# PyTorch嵌入层
import torch.nn as nn

vocab_size = 32000
d_model = 4096

# 嵌入矩阵: 32000 × 4096
embed = nn.Embedding(vocab_size, d_model)
# 参数量: 32000 × 4096 = 131M

# token id → 向量
token_ids = torch.tensor([15, 892, 3])
vectors = embed(token_ids)
# shape: (3, 4096)
```

模型	V	d_model	嵌入参数量
LLaMA-7B	32K	4096	131M (1.9%)
LLaMA-3-8B	128K	4096	524M (6.5%)

关键要点: 嵌入层将离散token映射为连续向量，参数量 = V×d, vocab越大嵌入层越占比

自注意力: 直觉理解

核心思想: 每个token通过"提问"所有其他token来获取上下文信息

例: "The cat sat on the mat"

当处理 "sat" 时:

sat的Query → 和所有token的Key计算相似度

- 和当前 token 更相关的词，给更大权重
  不相关的词，给更小权重
  把相关词的信息聚合过来，更新当前 token 的表示

Attention权重 (示意):

The → 0.05（低关注）

cat → 0.45（高关注: 谁sat?）

sat → 0.15（自身）

on → 0.10

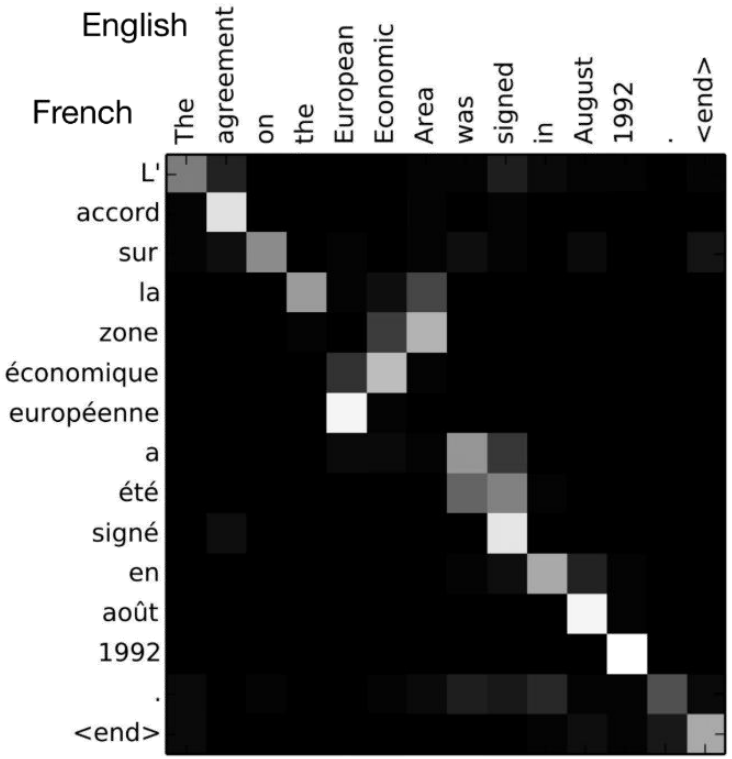
the → 0.05

mat → 0.20（关注: sat在哪?）

在机器翻译中, 注意力矩阵会自动学习词语对齐关系 (如 "猫"↔"cat")

- **三个角色: Query, Key, Value**
- Query (查询): "我在找什么信息?"
- Key (键): "我包含什么信息?"
- Value (值): "我能提供什么内容?"

- **计算过程:**
- 1. 每个token生成自己的Q, K, V向量
- 2. Q和所有K做点积 → 相关性分数
- 3. Softmax归一化 → 注意力权重
- 4. 用权重对V加权求和 → 上下文表示



关键要点: 自注意力让每个token"看到"所有其他token，通过Q·K相似度决定关注谁, 用V提取信息

自注意力: 数学与计算复杂度

公式推导:

输入: $X \in \mathbb{R}^{n \times d}$ (n个token, d维)

$Q = XW_Q \quad K = XW_K \quad V = XW_V$

$W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$

$$\text{Attention}(Q,K,V) = \text{Softmax}(QK^T/\sqrt{d}) \cdot V$$

- $QK^T \in \mathbb{R}^{n \times n}$ ， 注意力矩阵
- $/\sqrt{d}$ ， 缩放因子, 防止点积过大
- Softmax ， 行归一化为概率分布
- $\cdot V$ ， 加权聚合value信息

为什么除以$\sqrt{d}$? 当d很大时, QK点积的方差≈d, Softmax会趋向one-hot, 梯度消失

计算复杂度分析:

操作	复杂度	说明
$X \rightarrow Q,K,V$	$O(nd^2)$	3个矩阵乘法
QK^T	$O(n^2d)$	注意力分数
Softmax·V	$O(n^2d)$	加权求和
总计	$O(n^2d + nd^2)$	n和d的竞争

内存瓶颈:

注意力矩阵 $QK^T \in \mathbb{R}^{n \times n}$

n=4096, fp16: $4096^2 \times 2B = 32MB/head$

32 heads → 1GB per layer!

→ FlashAttention的动机 (后续章节)

关键要点: 自注意力复杂度 $O(n^2d)$: 序列长度n翻倍 → 计算量4倍, 这是长序列推理的核心系统挑战

代码: Self-Attention in PyTorch

```
import torch
import torch.nn as nn
import math

class SelfAttention(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.d = d_model
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)

    def forward(self, x):          # x: (B, n, d)
        Q = self.W_q(x)            # ★ Query投影
        K = self.W_k(x)            # ★ Key投影
        V = self.W_v(x)            # ★ Value投影

        # ★ 注意力分数: (B, n, n)
        scores = Q @ K.transpose(-2, -1)
        scores = scores / math.sqrt(self.d)

        attn = torch.softmax(scores, dim=-1)
        out = attn @ V             # ★ 加权聚合
        return out
```

- **代码解读:**

- **L9-11: 三个线性投影**
- W_q, W_k, W_v: d×d矩阵
- 参数量: $3 \times d^2$ (主要开销)

- **L14-16: Q/K/V投影**
- 输入x经过线性变换
- 每个token生成自己的Q,K,V

- **L19-20: 注意力计算**
- $Q @ K^T \rightarrow n \times n$相似度矩阵
- 除以$\sqrt{d}$防止梯度消失

- **L22-23: Softmax + 聚合**
- 行归一化 → 注意力权重
- attn @ V → 上下文向量

关键要点: Self-Attention核心: 3个矩阵乘法(投影) + 1个矩阵乘法(QK^T) + Softmax + 1个矩阵乘法(·V)

多头注意力 (Multi-Head Attention)

为什么多头? 单个注意力头只能学一种关系模式，多个头可以同时捕捉不同的语义关系

- **单头 vs 多头**
- 单头: d_model → d_model (一种注意力模式)
- 多头: d_model → h个头, 每头 d_k = d_model/h

- **计算过程:**
- 1. 将Q,K,V按head维度分割
- $Q \in \mathbb{R}^{n \times d} \rightarrow$ h个 $Q\_i \in \mathbb{R}^{n \times (d/h)}$
- 2. 每个head独立做attention
- head_i = Attention(Q_i, K_i, V_i)
- 3. 拼接所有head的输出
- MultiHead = Concat(head_1,...,head_h)·W_O

- **参数量不变!**
- 总投影矩阵大小仍是$3d^2 + d^2 = 4d^2$

不同头学到不同模式:

Head 1: 语法关系

主语-谓语对应

Head 2: 指代关系

代词-先行词

Head 3: 相邻关系

局部n-gram模式

Head 4: 长距离依赖

跨句关联

系统视角: MHA的h个head可以完全并行计算，很好地利用GPU的并行能力

关键要点: 多头注意力: 相同参数量, 多种注意力模式并行，h个头完美映射到GPU并行单元

Causal Mask: 生成任务的注意力约束

问题: 生成下一个词时, 模型不能"偷看"未来的词，需要屏蔽未来位置

- **为什么需要Mask?**
- 训练时: 所有位置同时计算 (并行)
- 但位置t的预测只能依赖 ≤t 的token
- 否则模型直接"复制"答案, 学不到东西

- **Causal Mask (因果掩码):**
- 在attention score上加 -∞ mask
- Softmax(-∞) = 0 → 完全屏蔽未来
- 形成下三角矩阵 (每行只看左边)

- **训练 vs 推理:**
- 训练: mask允许并行计算所有位置
- 推理: 逐token生成, mask天然满足

Causal Mask 矩阵:

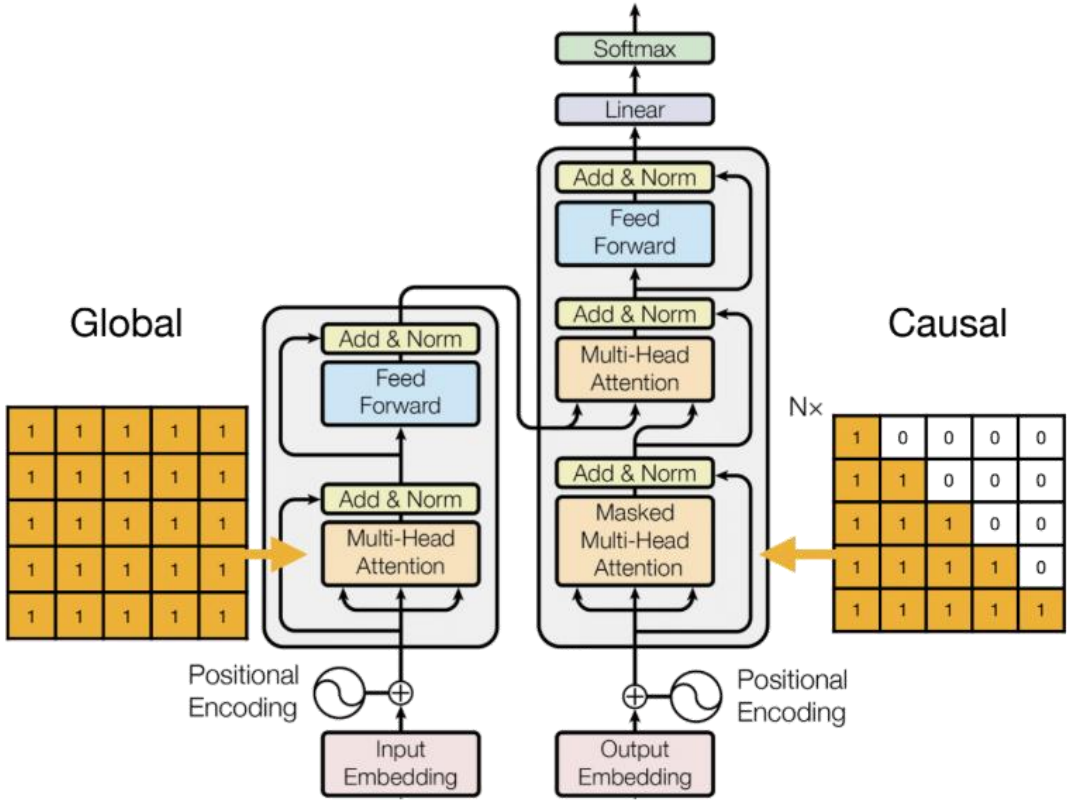
	t_1	t_2	t_3	t_4	t_5
t_1	√	✗	✗	✗	✗
t_2	√	√	✗	✗	✗
t_3	√	√	√	✗	✗
t_4	√	√	√	√	✗
t_5	√	√	√	√	√

√ = 可以看到 (mask=0)

✗ = 被屏蔽 (mask=-∞)

对比BERT (双向):

BERT不使用causal mask
每个token可以看到所有位置
适合理解任务, 不适合生成



关键要点: Causal mask让训练并行但预测单向，这是Decoder-only模型(GPT系列)的核心约束

前馈网络 (FFN): 隐藏的参数大户

为什么: 注意力只做线性加权，FFN提供元素级非线性变换, 是模型"记忆"知识的主要载体

- **FFN结构: 两层MLP (反向瓶颈)**
- $FFN(x) = W_2 \cdot GELU(W_1 x + b_1) + b_2$
- $W_1 \in \mathbb{R}^{d \times 4d}$，扩展4倍
- $W_2 \in \mathbb{R}^{4d \times d}$，压缩回来

- **为什么4倍扩展?**
- 更大的隐层 → 更强的表达能力
- 4倍是经验值, 性能-效率平衡点
- SwiGLU用8/3倍扩展但有3个矩阵 (后续讨论)

参数量对比: FFN vs Attention

组件	参数量	占比
Attention (Q,K,V,O)	$4d^2$	~1/3
FFN (W_1, W_2)	**$8d^2$**	**~2/3**
LayerNorm	4d	≈0
Total per block	$12d^2$	100%

系统洞察:

FFN占了2/3的参数和计算

LLaMA–7B(SwiGLU): 每层FFN有 3×4096×11008 ≈ 135M参数

32层 → FFN总参数 ~4.3B (占7B的62%)

关键要点: FFN是Transformer的参数大户(~2/3参数)，它是模型存储世界知识的主要组件, 也是MoE优化的目标

LayerNorm + 残差连接: 训练稳定性

为什么: 没有归一化和残差连接, 深层Transformer无法训练 (梯度消失/爆炸)

Layer Normalization

对每个token的嵌入向量独立归一化:

$$LN(x) = \gamma \cdot (x - \mu) / (\sigma + \varepsilon) + \beta$$

- μ, σ 沿embedding维度计算
- γ, β 是可学习参数 (2d个)
- 与BatchNorm不同: 不依赖batch统计

RMSNorm (现代选择):

$$RMSNorm(x) = x / RMS(x) \cdot \gamma$$

去掉均值偏移, 更高效 (LLaMA使用)

Pre-norm vs Post-norm

Post-norm (原始Transformer):

x → Attention → Add(x) → LN

- 训练不稳定, 需要warmup
- 深层模型容易发散

Pre-norm (现代标准):

x → LN → Attention → Add(x)

- 训练更稳定, 无需warmup
- LLaMA/GPT/Mistral均使用Pre-norm

残差连接: output = x + SubLayer(x) , 梯度直通, 允许信息跨层传递, 是训练深层网络的关键

关键要点: Pre-RMSNorm + 残差连接 = 现代Transformer训练的标配, 保证100+层网络稳定训练

位置编码：让Transformer感知顺序

问题：自注意力是集合操作，"cat sat mat"和"mat sat cat"的计算结果完全相同!

- **为什么需要位置信息?**
- Attention(Q,K,V)中，打乱输入顺序不影响输出
- "I eat burger" = " burger eat I" (对attention而言)
- 必须显式注入位置信息

- **正弦位置编码 (原始Transformer)**
- PE(pos, 2i) = sin(pos / 10000^(2i/d))
- PE(pos, 2i+1) = cos(pos / 10000^(2i/d))

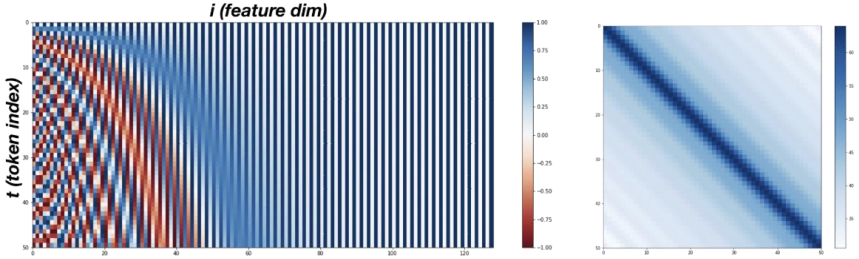
- 直觉：不同频率的正弦波编码不同尺度的位置
- 加到嵌入向量上：input = embedding + PE

位置编码类型对比:

类型	方法	代表模型
绝对位置	加到输入embedding	原始Transformer
可学习绝对	学习位置embedding	GPT-1/2, BERT
相对位置	修改attention score	T5, ALiBi
旋转位置	**旋转Q/K向量**	**LLaMA, Mistral**

- **绝对编码的问题:**
- 训练时序列长度固定 (如2048)
- 推理时遇到更长序列 → 位置编码越界
- 相对编码 (RoPE/ALiBi) 可更好泛化
- → 第二节课详细讲解RoPE

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases} \qquad \omega_k = \frac{1}{10000^{2k/d}}$$



关键要点: 位置编码是Transformer感知序列顺序的唯一方式，现代LLM已从绝对编码演进到旋转位置编码(RoPE)

代码: Transformer Block in PyTorch

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        self.norm1 = nn.RMSNorm(d_model)     # ★ Pre-norm
        self.attn = MultiHeadAttn(d_model, n_heads)
        self.norm2 = nn.RMSNorm(d_model)
        self.ffn = FFN(d_model)

    def forward(self, x, mask=None):
        # ★ Pre-norm + Residual pattern
        h = self.norm1(x)
        h = self.attn(h, mask=mask)
        x = x + h                # ★ 残差连接

        h = self.norm2(x)
        h = self.ffn(h)
        x = x + h                # ★ 残差连接
        return x

class FFN(nn.Module):
    def __init__(self, d, mult=4):
        super().__init__()
        hidden = d * mult
        self.w1 = nn.Linear(d, hidden)
        self.w2 = nn.Linear(hidden, d)
```

- **数据流 (Pre-norm):**

- **x**
- │
- ├→ RMSNorm → MHA → +
- │ | (残差)
- ├──────────────────────┘
- │
- ├→ RMSNorm → FFN → +
- │ | (残差)
- ├──────────────────────┘
- **x' (output)**

- **整个Transformer:**
- = N个Block堆叠
- LLaMA-7B: N=32
- LLaMA-70B: N=80

关键要点: 一个Transformer Block = RMSNorm→MHA→Residual + RMSNorm→FFN→Residual, 堆叠N次即为完整模型

Transformer计算量全景拆解

系统核心: 理解FLOPs分布, 才能知道优化哪个组件收益最大

操作	FLOPs	说明	n=2K, d=4096 FLOPs
QKV投影	$6nd^2$	3个 (n×d)×(d×d) 矩阵乘	~201G
Attention Score (QK^T)	$2n^2d$	n×d 乘 d×n	~34G
Attention·V	$2n^2d$	n×n 乘 n×d	~34G
Output投影	$2nd^2$	(n×d)×(d×d)	~67G
FFN (W_1+W_2)	**$16nd^2$**	**两个 d×4d 矩阵乘**	**~537G**
Total per block	**$24nd^2 + 4n^2d$**		**~873G**

n 较小时 (n << d):

$24nd^2$ 主导 → FFN是计算瓶颈

n=2K, d=4096: FFN占 ~62%

n 较大时 (n >> d):

$4n^2d$ 主导 → Attention是计算瓶颈

n=128K, d=4096: Attention占 ~84%

关键要点: 短序列优化FFN (MoE), 长序列优化Attention (FlashAttention, 稀疏注意力), 瓶颈随序列长度变化

原始Transformer: 结果与影响

机器翻译基准 (WMT 2014):

模型	EN→DE	EN→FR	训练成本
GNMT (RNN)	26.03	39.92	大
ConvS2S (CNN)	25.16	40.46	大
Transformer	**28.4**	**41.0**	**1/4**

- **关键突破:**
- BLEU分数超越所有已有模型
- 训练成本仅为之前最佳的1/4
- 并行训练 = 更多数据 + 更大模型

Transformer的深远影响:

- **2017** Attention Is All You Need
- **2018** BERT: NLP理解任务全面突破
- **2018** GPT-1: 自回归预训练
- **2020** GPT-3: 175B, 涌现In-Context Learning
- **2020** ViT: Transformer进入计算机视觉
- **2022** ChatGPT: AI进入公众视野
- **2023** GPT-4, LLaMA, 开源LLM时代
- **2024** o1, Reasoning Models, 多模态

关键要点: Transformer用1/4训练成本超越所有模型，并行化不只是快, 更意味着可以训练更大的模型

课堂思考 & 第一节课总结

课堂思考题:

1. 如果序列长度n翻倍, Transformer的计算量增加多少? 内存呢?

2. 为什么FFN要用4倍扩展而不是2倍或8倍?

3. 如果去掉残差连接, 一个32层Transformer还能训练吗?

并行 > 串行

RNN串行→GPU浪费
Transformer并行→规模化基础

$O(n^2d)$ 复杂度

短序列FFN主导
长序列Attention主导

6个核心组件

Token+Embed+MHA+FFN
+LayerNorm+PositionalEnc

关键要点: 下节课: 每个设计改进背后都有一个系统瓶颈， RoPE, GQA, SwiGLU, KV Cache优化

70B推理: KV缓存比模型权重还大!

为什么: 原始Transformer的每个设计选择都面临系统瓶颈，改进都是被"逼"出来的

KV Cache Memory (MHA)

~160 GB

在标准 MHA、FP16、80 层、hidden size 约 8192 的 70B 级模型, bs=16, seq_len=4096

仅KV缓存就需两块A100! (还未算权重~140GB)

本节课要回答:

- Encoder-only vs Decoder-only: 为什么LLM选了后者?
- 绝对位置 → RoPE: 怎么让模型泛化到更长序列?
- MHA → GQA: 怎么把KV缓存缩小8倍?
- FFN → SwiGLU: 简单改动为什么有效?
- Scaling Laws: 固定算力, 模型该多大?

关键要点: 每个设计改进都源于系统瓶颈: 内存、计算、泛化，理解"为什么改"比"改了什么"更重要

PART 03

架构三条路线

Encoder-Decoder, Encoder-only, Decoder-only: 殊途同归?

三种Transformer架构对比

特性	Encoder-Decoder (T5)	Encoder-only (BERT)	Decoder-only (GPT)
注意力	Encoder双向+ Decoder因果+交叉	双向 (全连接)	因果 (下三角)
预训练目标	跨度掩码去噪 (denoising)	Masked LM (完形填空)	Next token prediction
典型任务	翻译, 摘要 (输入→输出)	分类, NER (理解)	生成, 对话 (创造)
代表模型	T5, BART Flan-T5	BERT, RoBERTa DeBERTa	GPT-3/4, LLaMA Mistral, Claude
规模化	结构复杂 难以极大规模化	不能生成 应用场景受限	**简单统一 最易规模化**

关键要点: 三种架构各有所长, 但Decoder-only凭借统一框架和规模化优势, 成为LLM时代的主流选择

BERT: 双向编码器

核心创新: 用Masked Language Model预训练, 让每个token"看到"上下文双向信息

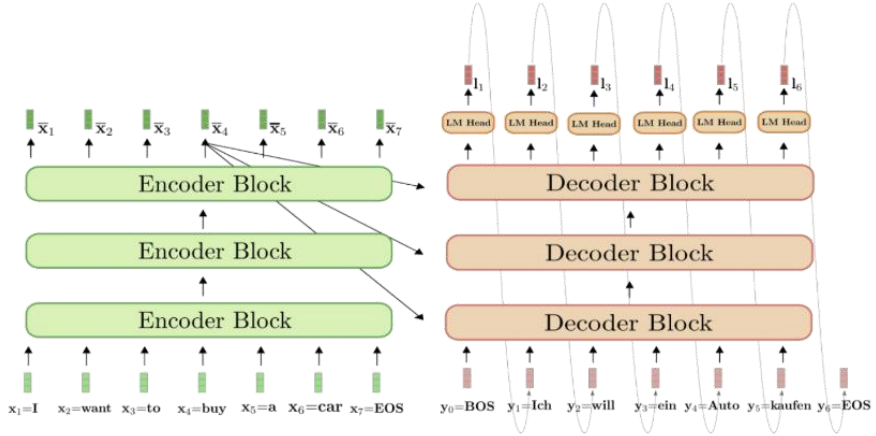
- **预训练任务 #1: Masked LM (MLM)**
- 随机mask 15%的token
- 训练模型预测被mask的词
- "The [MASK] sat on the mat" → cat

- **预训练任务 #2: Next Sentence Prediction**
- 判断句子B是否是A的下一句
- (后续研究发现作用不大)

- **使用方式: 预训练 + 微调**
- 在大语料上预训练
- 在下游任务上加分类头微调
- BERT-Base: 110M, BERT-Large: 340M

Attention Mask (双向):

	t_1	t_2	t_3	t_4	t_5
t_1	√	√	√	√	√
t_2	√	√	√	√	√
t_3	√	√	√	√	√
t_4	√	√	√	√	√
t_5	√	√	√	√	√



每个token可以看到所有其他token → 适合理解任务

局限:

- 不能自然地做生成任务
- 训练只用15% token的损失 (低效)
- 规模化困难, 最大仅~1B参数

关键要点: BERT用双向注意力+MLM实现强大的语言理解, 但无法生成文本, 且规模化受限

GPT: 自回归解码器

核心创新: 用Next Token Prediction预训练，简单、统一、可无限规模化

- **预训练: Next Token Prediction**
- 给定前面所有token, 预测下一个
- $P(x \mid x_1, x_2, \ldots, x_{-1})$
- 损失函数: 交叉熵, 每个token都参与

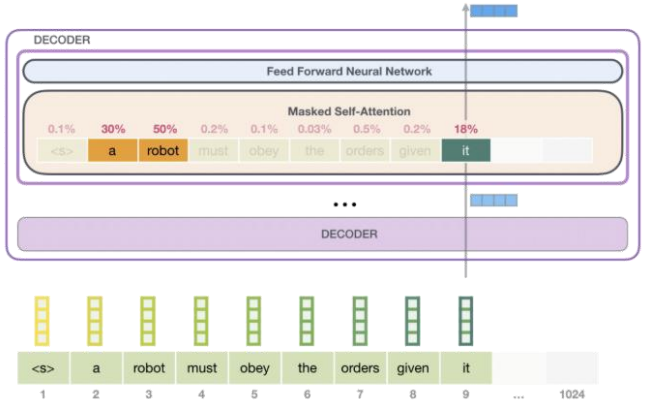
- **为什么高效?**
- 100% token利用率 (vs BERT的15%)
- 每个序列提供n个训练样本
- 天然适合生成任务

- **使用方式的演进:**
- GPT-1 (117M): 预训练+微调
- GPT-2 (1.5B): 零样本开始可行
- GPT-3 (175B): In-Context Learning!

自回归生成过程:

```
# 自回归生成 (贪心)
prompt = "The capital of France"
tokens = tokenize(prompt)

for i in range(max_len):
    logits = model(tokens)
    next_token = logits[-1].argmax()
    tokens.append(next_token)
    if next_token == EOS:
        break
# → "The capital of France is Paris"
```



优势 (系统视角):

- 架构最简单: 只有decoder, 无cross-attn
- 训练最高效: 每个token都计算loss
- 规模化最好: 从117M到1.8T参数

关键要点: GPT式Decoder-only: 最简单的架构, 最高效的训练, 最好的规模化特性，大道至简

为什么LLM时代选择了Decoder-only?

统一框架

一切皆生成:
分类? 生成"正面/负面"
翻译? 生成目标语言
摘要? 生成简短版本
无需为每个任务设计结构

训练效率

每个token都有loss信号:
BERT: 15% token有梯度
GPT: 100% token有梯度
同样数据, GPT学到更多

规模化收益

架构越简单,规模越大:
无encoder-decoder对齐
参数全部用于单一模型

关键要点: Decoder-only = 最简单 × 最高效 × 最易规模化，2023年后几乎所有主流LLM都采用这一架构

PART 04

关键设计改进

位置编码、KV缓存、FFN，每个改进背后的系统动机

位置编码进化: 从绝对到相对

问题: 绝对位置编码无法泛化到训练时未见过的序列长度， 训练2K, 推理时4K怎么办?

绝对位置编码的问题:

- 位置信息加到输入embedding
- pos=2049时, 编码从未在训练中出现
- 可学习位置: 超出范围直接越界
- 正弦位置: 理论可外推, 实际效果差

相对位置编码的优势:

- 只关心两个token间的相对距离
- 不依赖绝对索引, 天然支持更长序列
- 作用于attention score而非输入
- 实现: ALiBi, RoPE (当前主流)

对比总结:

	绝对PE	ALiBi	RoPE
作用位置	输入层	Attention	Q和K
可学习?	可选	否	否
长度外推	差	较好	好(+NTK)
额外参数	n×d或0	0	0
额外计算	加法	加法	旋转乘法
使用率	少	少	主流

几乎所有现代LLM (LLaMA, Mistral, Qwen, DeepSeek) 都使用RoPE

关键要点: 相对位置编码解决了长度泛化问题， RoPE已成为事实标准, 支撑LLM从2K扩展到128K+上下文

ALiBi: 简单的线性偏置

思路: 不在输入上加位置, 而在attention score上加距离相关的偏置

- **Attention with Linear Biases**
- Softmax($QK^T/\sqrt{d} + m \cdot bias$)
- bias[i,j] = -|i - j| (距离越远, 惩罚越大)
- **m: 每个head不同的斜率**
- m = 2^(-8/h), 2^(-16/h), ...
- 不同head关注不同距离范围
- 小斜率head → 关注远距离
- 大斜率head → 关注近距离

Bias矩阵示例 (m=1):

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ -1 & 0 & -\infty & -\infty \\ -2 & -1 & 0 & -\infty \\ -3 & -2 & -1 & 0 \end{bmatrix} \end{array}$$

(结合causal mask, 未来位置=−∞)

特点:

√ 无额外参数, 无额外学习

√ 实现极简 (几行代码)

✗ 长度外推效果不如RoPE+NTK

关键要点: ALiBi用最简单的方式引入位置信息，距离越远惩罚越大, 但RoPE的效果和灵活性更优

RoPE: 旋转位置编码

核心直觉: 用旋转来编码位置，两个向量的相对旋转角度只取决于它们的相对位置

- **旋转的直觉 (2D)**
- 把embedding的每2个维度看成一个2D平面
- 位置m的token → 旋转 m·θ 角度

- Q_m和K_n做点积时:
- 点积只取决于旋转角度差 (m-n)·θ
- → 天然编码相对位置!

- **公式:**
- $f(x, m) = R(m\cdot\theta) \cdot x$

- $R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$

- $\theta_i = 10000^{(-2i/d)}, i=0,...,d/2-1$

RoPE的关键性质:

相对位置编码

$\langle f(q,m), f(k,n) \rangle = g(q,k,m-n)$
内积只取决于相对位置差m-n

长距离衰减

θ越小的维度 → 区分更远距离
自然的远距离衰减特性

长度外推

RoPE + NTK-aware插值
可从4K训练扩展到128K+推理

关键要点: RoPE将位置编码为旋转角度，内积自然保持相对位置, 支持长度外推, 几乎所有现代LLM标配

RoPE: 数学推导

将d维向量分成d/2对:

$x = [x_1, x_2, x_3, x_4, ..., x_{d-1}, x_d]$

├─对1─┤ ├─对2─┤ ├──对d/2──┤

每对独立旋转:

$$\begin{bmatrix} x'_{2i-1} \\ x'_{2i} \end{bmatrix} = \begin{bmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{bmatrix} \begin{bmatrix} x_{2i-1} \\ x_{2i} \end{bmatrix}$$

其中 $\theta_i = 10000^{(-2i/d)}$

低维对: θ大 → 高频旋转 → 区分近距离

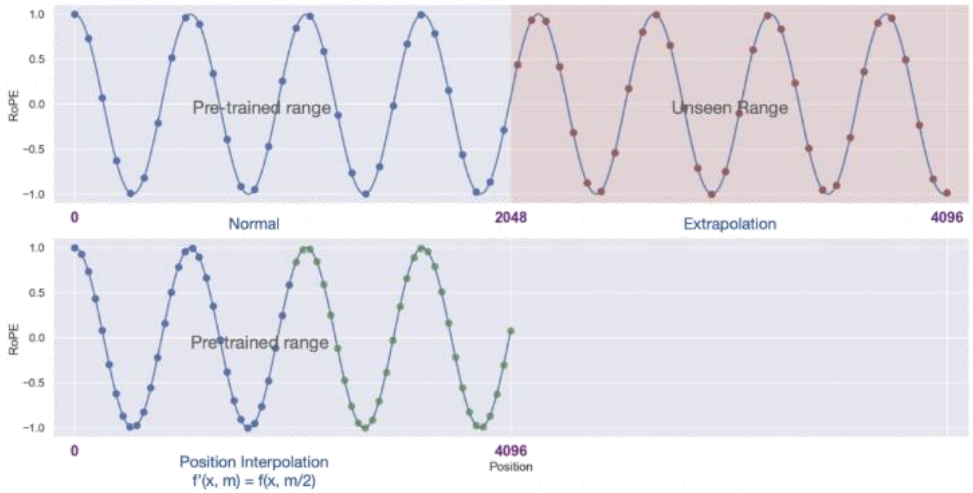
高维对: θ小 → 低频旋转 → 区分远距离

一般形式 (稀疏旋转矩阵):

$$R(m) = \begin{bmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \cdots \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \cdots \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \cdots \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cos \; -\sin \end{bmatrix}$$

- **验证相对位置性质:**
- ⟨R(m)·q, R(n)·k⟩
- $= q^T R(m)^T R(n) k$
- $= q^T R(n-m) k$ (旋转矩阵性质)
- → 只依赖相对位置差 (n-m) √

- **复数视角:**
- $f(x,m) = x \cdot e^{im\theta}$
- 两个复向量内积的相位角 = 相位差(m-n)



$m \in [0, 2048 * 2), \quad \theta'_i = \theta_i$ ✗

$m \in [0, 2048 * 2), \quad \theta'_i = \theta_i / 2$ ✓

关键要点: RoPE的数学优雅: 稀疏旋转矩阵, 计算开销极小, 内积天然保持相对位置, 理论与实践完美结合

代码: RoPE Implementation

```
def precompute_freqs(d, max_len, base=10000):
    """预计算旋转频率"""
    freqs = 1.0 / (base ** (
        torch.arange(0, d, 2).float() / d
    ))                          # ★ θ_i = 10000^(−2i/d)
    t = torch.arange(max_len)
    freqs = torch.outer(t, freqs)  # (max_len, d/2)
    return torch.polar(
        torch.ones_like(freqs), freqs
    ) # ★ e^{i·m·θ} 复数形式

def apply_rope(x, freqs):
    """对Q或K应用RoPE"""
    # x: (B, n, h, d) → 复数视图
    x_complex = torch.view_as_complex(
        x.float().reshape(*x.shape[:−1], −1, 2)
    )                           # ★ 每2维→1个复数
    # 旋转: 复数乘法 = 2D旋转
    x_rotated = x_complex * freqs  # ★ 核心!
    return torch.view_as_real(
        x_rotated
    ).flatten(−2)               # 恢复实数
```

- **代码解读:**

- **precompute_freqs:**
- $\theta_i = 10000^{(-2i/d)}$
- 预计算 e^{i·m·θ} 复数表
- 只需计算一次, 缓存复用

- **apply_rope:**
- 将实数向量转为复数
- 每2个实数 → 1个复数
- 复数乘法 = 2D旋转
- 这就是RoPE的全部!

- **效率:**
- 只需d/2次复数乘法
- 额外FLOPs ≈ 0 (相比MHA)

关键要点: RoPE实现仅需3步: 预计算频率 → 实数转复数 → 复数乘法(=旋转), 计算开销可忽略

KV缓存: 推理的内存瓶颈

问题: 自回归解码时, 每生成一个新token都要重新计算所有历史token的K和V, 太浪费!

- **自回归推理的计算浪费:**
- 生成token t时, 需要计算:
- Attention(q$_t$, [k$_1$,...,k$_t$], [v$_1$,...,v$_t$])
- 但k$_1$,...,k$_{t-1}$ 和 v$_1$,...,v$_{t-1}$ 已经算过!

- **KV Cache解决方案:**
- 缓存所有已计算的K和V向量
- 每步只需计算新token的Q, K, V
- 用新的Q和缓存的全部K, V做attention

- **Prefill vs Decode:**
- Prefill: 并行处理prompt, 填充KV缓存
- Decode: 逐token生成, 每步追加KV缓存

KV Cache工作流程:

Step 1 (Prefill)

输入: "Hello world"
K_cache = [k$_1$, k$_2$]
V_cache = [v$_1$, v$_2$]

Step 2 (Decode)

新token: "!"
K_cache = [k$_1$, k$_2$, k$_3$]
V_cache = [v$_1$, v$_2$, v$_3$]

Step 3 (Decode)

新token: " How"
K_cache = [k$_1$, k$_2$, k$_3$, k$_4$]
V_cache = [v$_1$, v$_2$, v$_3$, v$_4$]

关键要点: KV Cache用空间换时间: 缓存历史K/V避免重复计算, 但缓存大小随序列长度线性增长, 是推理的内存瓶颈

KV缓存: 算笔账

系统问题: KV缓存到底需要多少显存? 能服务多少用户?

KV Cache Size = 2 × n_layers × n_kv_heads × d_head × seq_len × batch_size × bytes_per_param

↑K和V ↑层数 ↑KV头数 ↑头维度 ↑序列长 ↑批大小 ↑精度(fp16=2)

模型	Layers	KV Heads	d_head	bs=1,n=4K	bs=16,n=4K	模型权重
LLaMA-2-7B	32	32 (MHA)	128	1 GB	16 GB	14 GB
LLaMA-2-13B	40	40 (MHA)	128	1.6 GB	25 GB	26 GB
LLaMA-2-70B	80	**8 (GQA)**	128	2.5 GB	40 GB	140 GB
若70B用MHA	80	64 (MHA)	128	10 GB	**160 GB!**	140 GB

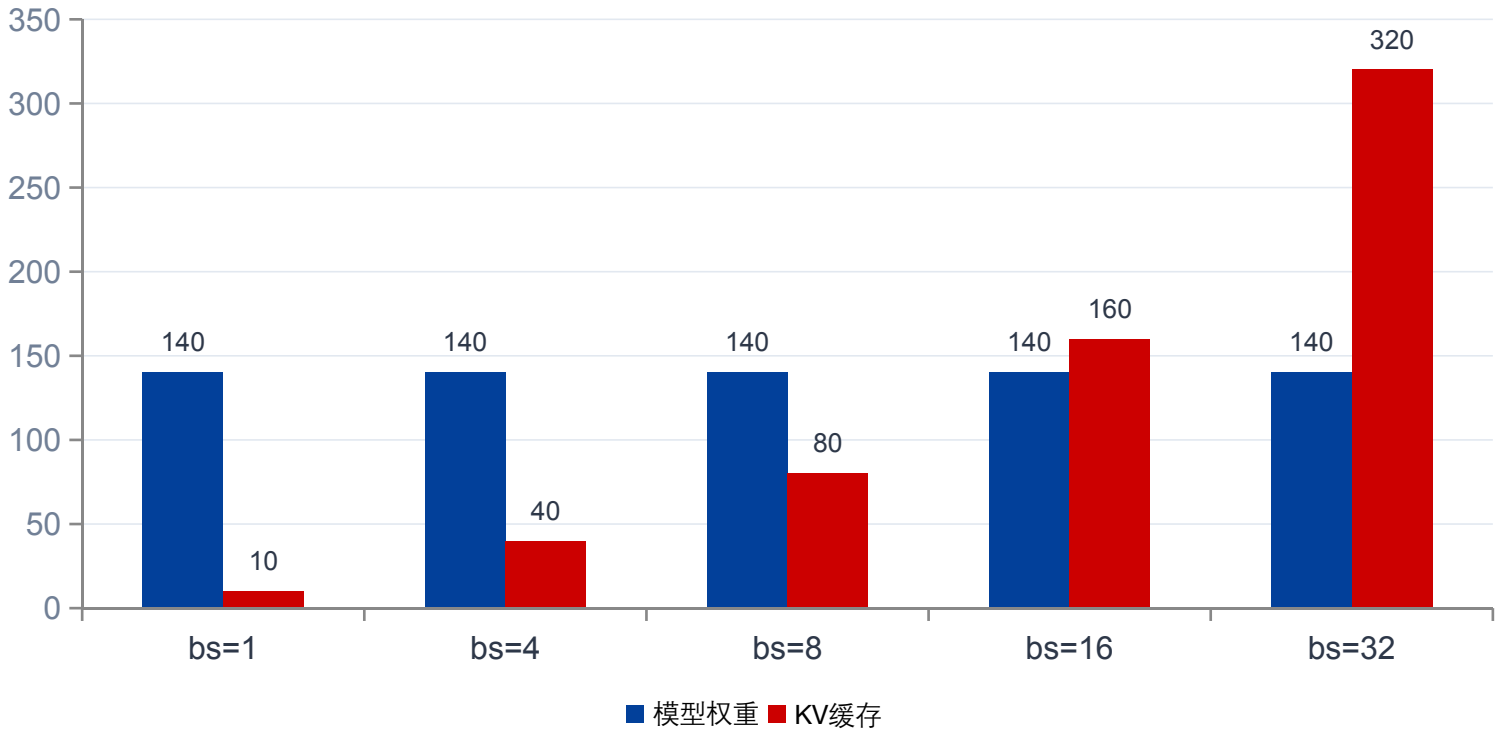
70B MHA: bs=16, n=4K → KV缓存160GB, 超过模型权重(140GB)! 需要2块A100!

70B GQA: 同样设置只需40GB KV缓存，GQA将KV头从64减到8, 缓存缩小8倍

关键要点: KV缓存是推理显存的头号消耗者，对大模型和长序列, 缓存可能比模型权重还大, 这就是GQA的动机

KV缓存 vs 模型权重: 谁更占显存?

LLaMA-2-70B (MHA, seq=4096, fp16):



关键观察:

- bs=8时, KV缓存=80GB
  已与权重(140GB)可比
- bs=16时, KV缓存=160GB
  超过模型权重!
- bs=32时, KV缓存=320GB
  需要4块A100-80GB
  仅用于存KV缓存

关键要点: batch越大, KV缓存增长越快, 这直接限制了LLM serving的吞吐量, 是系统优化的核心目标

MHA → MQA → GQA: KV缓存优化

解决方案: 减少KV头数量，多个Query头共享同一组KV, 缓存大幅缩小

MHA

N个Q头, N个KV头
每头独立的K和V
KV缓存最大

Q heads: 32 KV heads: 32

KV Cache: 100%

MQA

N个Q头, 1个KV头
所有Q共享1组KV
KV缓存最小但质量降

Q heads: 32 KV heads: 1

KV Cache: 3.1%

GQA

N个Q头, G个KV头
G组Q共享对应KV
平衡缓存与质量

Q heads: 32 KV heads: 8

KV Cache: 25%

方法	KV Cache	质量	推理速度	代表模型
MHA	100%	最优	基准	GPT-3, LLaMA-1
MQA	1/N (~3%)	略降	最快	PaLM, Falcon
GQA	**G/N (~25%)**	**≈MHA**	**快**	**LLaMA-2/3, Mistral**

关键要点: GQA是当前最优平衡: KV缓存减少75%, 质量几乎不损失，LLaMA-2-70B因此节省120GB KV缓存

代码: GQA Forward Pass

```
class GroupedQueryAttention(nn.Module):
    def __init__(self, d, n_heads, n_kv_heads):
        super().__init__()
        self.n_heads = n_heads       # e.g., 32
        self.n_kv = n_kv_heads       # e.g., 8
        self.d_head = d // n_heads   # e.g., 128
        self.n_rep = n_heads // n_kv # ★ 每组4个Q共享1个KV

        self.wq = nn.Linear(d, n_heads * self.d_head)
        self.wk = nn.Linear(d, n_kv_heads * self.d_head)
        self.wv = nn.Linear(d, n_kv_heads * self.d_head)
        self.wo = nn.Linear(d, d)

    def forward(self, x):
        B, n, _ = x.shape
        q = self.wq(x).view(B,n,self.n_heads,self.d_head)
        k = self.wk(x).view(B,n,self.n_kv,self.d_head)
        v = self.wv(x).view(B,n,self.n_kv,self.d_head)

        # ★ 扩展KV头: (B,n,8,128)→(B,n,32,128)
        k = k.repeat_interleave(self.n_rep, dim=2)
        v = v.repeat_interleave(self.n_rep, dim=2)

        # 标准attention计算
        scores = (q @ k.transpose(-2,-1)) / (self.d_head**0.5)
        attn = F.softmax(scores, dim=-1)
```

- **GQA vs MHA关键差异:**

- **WK, WV投影更小:**
- MHA: d → n_heads × d_head
- GQA: d → n_kv_heads × d_head
- 参数减少: (n_heads-n_kv)/n_heads

- **repeat_interleave:**
- 将8个KV头扩展为32个
- 每4个Q头共享同一KV
- 这步是GQA的核心操作

- **KV缓存节省:**
- 只缓存n_kv个头的K/V
- 32→8: 缓存缩小4倍
- LLaMA-2-70B用此方案

关键要点: GQA的代码改动极小(改WK/WV维度+repeat_interleave)，但KV缓存可缩小4-8倍, 投入产出比极高

SwiGLU: 更好的FFN设计

改进: 用门控机制替代简单ReLU，小改动带来稳定的性能提升

原始FFN:

$$FFN(x) = W_2 \cdot ReLU(W_1 x)$$

GLU (Gated Linear Unit):

$$GLU(x) = (W_1 x) \odot \sigma(W\_gate \cdot x)$$

SwiGLU (LLaMA使用):

$$SwiGLU(x) = (W_1 x \odot Swish(W\_gate \cdot x)) \cdot W_2$$

⊙ = 元素级乘法 (门控)

Swish(x) = x · σ(x) (平滑版ReLU)

gate决定哪些信息通过 → 更精细控制

GLU变体性能对比:

FFN变体	激活	参数量	Perplexity↓
FFN-ReLU	ReLU	$8d^2$	基准
FFN-GELU	GELU	$8d^2$	略优
GLU-Swish	Swish	$8d^2$	更优
SwiGLU	**Swish**	**~$8d^2$**	**最优**

注: SwiGLU有3个权重矩阵(W_1, W_gate, W_2), 但调整隐藏维度(如LLaMA用2/3×4d)使总参数量≈原始FFN

关键要点: SwiGLU: 用门控机制精细控制信息流，实现简单,效果稳定,已被LLaMA/Mistral/Qwen等所有主流LLM采用

现代LLM设计清单: 2024年标配

组件	原始Transformer (2017)	现代LLM (2024)	改进动机
架构	Encoder-Decoder	**Decoder-only**	统一框架, 规模化
位置编码	正弦 (绝对)	**RoPE (相对)**	长度泛化
注意力	MHA	**GQA**	KV缓存优化
FFN	ReLU, 4×扩展	**SwiGLU, 8/3×扩展**	更好性能
归一化	Post-LayerNorm	**Pre-RMSNorm**	训练稳定+高效
Tokenizer	WordPiece/BPE	**BPE (大vocab)**	更短序列
训练数据	~10B tokens	**1-15T tokens**	更多知识
上下文长度	512	**4K-128K+**	更长输入

关键要点: 7年间的每个改进都不是随意的，每一项都解决了具体的系统瓶颈: 内存、计算、训练稳定性、长度泛化

PART 05

大模型规模化

Scaling Laws、Chinchilla法则与LLM发展史

Scaling Laws: 大力出奇迹的数学基础

核心发现: 模型性能(loss)与模型大小、数据量、计算量存在精确的幂律关系

$$L(N) \propto N^{(-0.076)}$$

$$L(D) \propto D^{(-0.095)}$$

$$L(C) \propto C^{(-0.050)}$$

N = 模型参数量

D = 训练数据量 (tokens)

C = 训练计算量 (FLOPs)

关键洞察:

- 加大模型 → loss可预测地下降
- 加大数据 → loss可预测地下降

系统含义:

可预测性

小模型实验可预测大模型性能
→ 减少昂贵的大规模实验

资源分配

给定算力预算C, 如何分配给
模型大小N和数据量D?

投资回报

算力每增加10倍
loss下降约10^(-0.05) ≈ 11%

关键要点: Scaling Laws让LLM训练从"炼金术"变成"工程", 可以用小实验预测大模型性能, 指导数十亿美元投资

Chinchilla法则: 如何分配算力?

关键问题: 给定固定算力, 训练大模型少数据 vs 小模型多数据, 哪个更优?

Kaplan (2020) 建议:

"*优先扩大模型, 数据次要*"

GPT-3: 175B参数, 仅300B tokens

→ 模型很大, 但训练不充分!

Chinchilla (2022) 修正:

"*模型和数据应同步扩大*"

最优: tokens ≈ 20 × 参数量

Chinchilla: 70B参数, 1.4T tokens
比Gopher(280B)更好!

LLaMA的进一步思考:

- **推理成本也很重要!**
- Chinchilla优化的是训练计算
- 但部署时推理成本∝模型大小

- **LLaMA的策略:**
- 用远超Chinchilla推荐的数据训练小模型
- LLaMA-7B: 1T tokens (Chinchilla推荐140B, 7×)
- LLaMA-2-7B: 2T tokens (Chinchilla推荐140B, 14×)
- 过度训练7~14×!

- **结果:**
- 7B模型性能接近13B
- 推理成本降低2倍
- 这就是为什么开源LLM偏爱小模型+多数据

关键要点: Chinchilla法则: tokens≈20×参数量, 但考虑推理成本, LLaMA证明"小模型+过度训练"是更好的工程选择

LLM发展时间线

时间	模型	参数量	训练数据	上下文	关键贡献
2018.6	**GPT-1**	117M	5B	512	首个大规模预训练+微调
2019.2	**GPT-2**	1.5B	10B	1024	零样本学习开始可行
2020.5	**GPT-3**	175B	300B	2048	In-Context Learning涌现
2022.5	**OPT**	175B	180B	2048	首个开源175B模型 (Meta)
2023.2	**LLaMA**	7–65B	1–1.4T	2048	开源LLM时代开启
2023.3	**GPT-4**	?	?	8K–32K	多模态+推理能力飞跃
2023.7	**LLaMA-2**	7–70B	2T	4096	GQA, 更多数据
2023.9	**Mistral-7B**	7B	?	8K	7B超越LLaMA-13B
2024.4	**LLaMA-3**	8–405B	15T	8K–128K	15T tokens, 128K上下文
2025.1	**DeepSeek-R1**	671B MoE	14.8T	128K	推理模型, 37B活跃参数

关键要点: 7年间: 参数量1000×增长, 训练数据3000×增长, 上下文长度256×增长， 每个维度都在指数扩展

LLaMA系列演进: 开源LLM的标杆

特性	LLaMA (2023.2)	LLaMA-2 (2023.7)	LLaMA-3 (2024.4)
规模	7/13/33/65B	7/13/70B	8/70/405B
训练数据	1T-1.4T tokens	2T tokens	15T tokens
上下文长度	2048	4096	8K (base) / 128K
注意力	MHA	**GQA (70B)**	GQA (all)
位置编码	RoPE	RoPE	RoPE
FFN	SwiGLU	SwiGLU	SwiGLU
Vocab	32K	32K	**128K**
对齐版本	无	LLaMA-2-Chat	LLaMA-3-Instruct
后训练	,	SFT + RLHF	**SFT + RS + DPO**

趋势: 架构改动很小(都是Decoder-only + RoPE + SwiGLU), 性能提升主要来自: ①更多高质量数据 ②更长上下文 ③更好的后训练(RLHF/DPO)

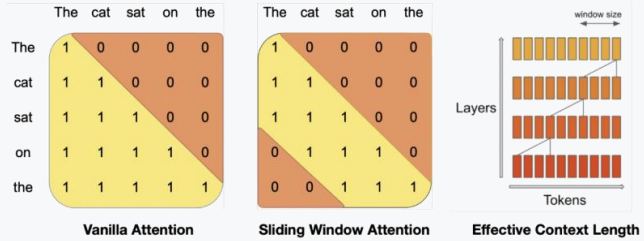
关键要点: LLaMA验证: 模型架构趋于收敛, 差异在于数据规模、数据质量和后训练策略

Mistral-7B & Mixtral: 小而精的突破

惊喜: 7B参数的Mistral超越了LLaMA-2-13B，怎么做到的?

Mistral-7B

7B参数 > LLaMA-2-13B



- GQA: 32 Q heads, 8 KV heads
- 上下文: 8K (LLaMA-2: 4K)
- 滑动窗口注意力 (SWA)，v2弃用
- 更好的训练数据质量

- 成功秘诀:
- ① 更好的数据筛选和清洗
- ② 架构细节优化 (GQA for 7B)
- ③ 更长上下文训练

Mixtral 8×7B (MoE)

46.7B总参数, 12.9B活跃

- 每个FFN → 8个专家FFN
- 每token激活Top-2专家
- 活跃参数 = 12.9B (28%)
- 性能 ≈ LLaMA-2-70B!

- 系统意义:
- ① 推理FLOPs ≈ 13B模型
- ② 但知识容量 ≈ 47B模型
- ③ 权衡: 需加载全部47B参数

关键要点: Mistral证明: 数据质量 > 模型大小; Mixtral预示: MoE是"大模型知识+小模型推理"的优化路径

课堂思考 &第二节课总结

课堂思考题:

1. 给定固定算力预算, 你会选择训练大模型少数据, 还是小模型多数据? 为什么?

2. 为什么Mistral-7B能超越LLaMA-2-13B? 这说明了什么?

3. 如果推理成本不重要, Chinchilla法则还成立吗?

设计都是被逼出来的

KV缓存太大→GQA
位置不能泛化→RoPE
FFN可以更好→SwiGLU

Scaling Laws指导投资

幂律关系可预测
Chinchilla优化训练
LLaMA优化推理

架构趋于收敛

Decoder-only + RoPE
+ GQA + SwiGLU
差异在于数据和训练

关键要点: 下节课: LLM的涌现能力, 推理模型(o1/R1), 多模态, MoE, 以及系统挑战全景

PART 06

LLM的涌现能力

规模化不只是"更好"，而是"质变"

涌现能力: 规模带来质变

发现: 某些能力只在模型足够大时突然出现，不是渐进提升, 而是从0到1的跳变

- **什么是涌现能力?**
- 小模型完全不具备的能力
- 模型超过某个规模后突然出现
- 无法从小模型性能外推预测

- **典型涌现能力:**
- • Chain-of-Thought推理 (~100B+)
- • 指令跟随 (~10B+, 需对齐训练)
- • 多步数学推理
- • 代码生成和调试
- • 多语言翻译 (零样本)

规模 vs 能力 (示意):

~1B	基础语言理解 简单补全
~10B	文本生成 简单问答 基础翻译
~100B	In-Context Learning CoT推理 代码生成
~1T+	复杂推理 多模态理解 工具使用

关键要点: 涌现能力说明: LLM不只是"更大的语言模型"，规模化可能触发质变, 这是持续扩大模型的核心动力

In-Context Learning: 不微调也能学

GPT-3的核心发现: 足够大的模型可以通过prompt中的示例"学会"新任务, 无需梯度更新

零样本 (Zero-shot):

Translate to French: "Hello"

→ "Bonjour"

少样本 (Few-shot):

Sentiment: "Great movie!" → Positive

Sentiment: "Terrible." → Negative

Sentiment: "It was okay" → ???

→ "Neutral"

- **关键观察 (GPT-3):**
- 模型越大, ICL效果越好
- 更多示例 → 更好表现
- 175B模型: few-shot接近微调效果

- **系统含义:**
- 无需为每个任务训练模型
- 一个模型服务所有任务
- 成本从训练转移到推理
- → 推理效率变得至关重要

- **局限:**
- 受上下文长度限制
- 示例质量影响大
- 复杂任务仍需微调

关键要点: ICL将NLP从"每任务一模型"变为"一个模型服务所有", 代价是推理成本上升(更长的prompt)

从预训练到对齐：让LLM听话

问题：预训练模型只会"续写"文本，不会遵循指令，不知道什么该说什么不该说

Step 1

预训练

在海量语料上训练
Next Token Prediction
学习语言知识和世界知识
数据：1-15T tokens
成本：数百万美元

→

Step 2

SFT（监督微调）

在高质量指令数据上微调
"指令→回答"对
学习遵循指令的格式
数据：10K-100K对话
成本：数千美元

→

Step 3

对齐（RLHF/DPO）

用人类偏好优化
学习回答质量、安全性
减少有害输出
数据：人类偏好对比
成本：数万美元

关键要点： 预训练给知识，SFT给格式，对齐给价值观，三步缺一不可，这就是ChatGPT的完整训练流水线

RLHF vs DPO: 对齐方法对比

RLHF (InstructGPT, 2022)

Reinforcement Learning from Human Feedback

- **训练流程:**
- 1. 收集人类偏好数据: (prompt, win, lose)
- 2. 训练奖励模型 (Reward Model)
- 3. 用PPO算法优化策略模型

- **需要4个模型:**
- Policy (策略), Reference (参考)
- Reward (奖励), Critic (评价)

- ✗ 训练复杂, 不稳定
- ✗ 超参数敏感
- ✗ 4个模型 → 巨大显存需求

DPO (Rafailov et al., 2023)

Direct Preference Optimization

- **核心洞察:**
- 奖励模型可以被解析消除!
- 直接用偏好数据优化策略

- **只需1个模型 (+reference):**
- $L = -\log \sigma(\beta \cdot (\log \pi/\pi\_ref(w)$
- $- \log \pi/\pi\_ref(l)))$

- √ 训练简单如SFT
- √ 稳定, 超参数少
- √ 显存需求大幅降低
- √ LLaMA-3, Qwen等采用

关键要点: DPO将RLHF从4模型简化为1模型, 训练稳定性大幅提升, 趋势是对齐方法越来越简单高效

PART 07

Reasoning Models

从"快思考"到"慢思考": 推理时的计算扩展

Chain-of-Thought: 提示的力量

发现: 在prompt中加一句"Let's think step by step" → 数学推理准确率翻倍!

直接回答 (Standard):

Q: The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more, how many apples do they have?

A: 27

✗ 错误! 正确答案是9 (23-20+6)

Chain-of-Thought:

Q: ... Let's think step by step.

A: Started with 23 apples.

Used 20: 23–20 = 3 apples.

Bought 6 more: 3+6 = 9 apples.

√ 中间步骤让推理透明可验证

- **Wei et al. (2022) 发现:**
- CoT只在大模型有效 (~100B+)
- 小模型生成无意义的中间步骤
- 涌现能力的典型例子

- **CoT提示 vs 训练推理:**
- **CoT Prompting:**
- 只是"提醒"模型展示思路
- 利用已有的隐含推理能力

- **Trained Reasoning (o1, R1):**
- 通过RL训练出新的推理能力
- 模型学会自我验证和回溯
- → 下一页详讲

关键要点: CoT是"提示工程"的里程碑，但真正的突破是训练模型主动推理，而不只是被动展示步骤

训练出推理能力: o1 & DeepSeek-R1

新范式: 不只在训练时投入计算, 在推理时也投入更多计算 → 更好的答案

OpenAI o1 (2024)

- **方法:**
- RL训练模型生成"思考链"
- 推理时生成数百~数千内部token
- 用户只看到最终答案

- **性能:**
- AIME数学竞赛: 83% → 96.7%
- Codeforces: 89th percentile
- 博士级科学问答超越人类专家

- **代价:**
- 推理token数3-10×增加
- 成本和延迟大幅上升

DeepSeek-R1 (2025)

- **架构: DeepSeek-V3 MoE**
- 671B总参数, 37B活跃参数

- **训练方法:**
- R1-Zero: 纯RL, 无SFT冷启动
- 涌现"aha moment"自我验证!
- 多阶段: 冷启动→RL→拒绝采样→SFT
- GRPO算法 (无需Critic模型)

- **突破:**
- 开源, 性能比肩o1
- 蒸馏到1.5B-70B小模型
- 证明: 纯RL可涌现推理

关键要点: 推理模型的核心创新: 用RL训练"思考"能力, R1证明纯RL就能涌现推理, 无需人工标注思维链

DeepSeek-R1: 技术细节

R1-Zero的发现: 纯RL训练下, 模型自发学会了自我验证和反思，不需要人类教!

- 训练流水线 (4阶段):

- **1. 冷启动SFT**
- 少量高质量CoT数据微调
- 建立基础推理格式

- **2. 推理RL (GRPO)**
- Group Relative Policy Optimization
- 奖励: 答案正确性 + 格式合规
- 无需训练单独的Reward Model

- **3. 拒绝采样 + SFT**
- 用RL模型生成大量推理样本
- 筛选高质量样本再SFT

- **4. 最终RL对齐**
- 同时优化有用性和安全性

R1-Zero: "Aha Moment"

纯RL训练 (无任何SFT) 后, 模型自发出现:

- 自我验证: "Wait, let me check..."
- 回溯: "That's wrong, let me redo..."
- 分步推理: 自动分解复杂问题
- 多策略尝试: 换一种方法再试

蒸馏 (Distillation):

R1-671B → 蒸馏到小模型:

R1–Distill–Qwen–1.5B/7B/14B/32B

R1–Distill–LLaMA–8B/70B

小模型也能获得推理能力!

关键要点: R1的最大贡献: 证明推理能力可以通过纯RL涌现, 并可蒸馏到小模型，推理能力不再是大模型的专利

Test-Time Compute Scaling: 新范式

传统: 投入更多计算到训练 → 更好模型; 新范式: 投入更多计算到推理 → 更好答案

传统范式: Train-Time Scaling

更多参数 + 更多数据 → 更好模型

推理时固定计算量

7B→70B→405B: 训练成本100×↑

新范式: Test-Time Scaling

固定模型, 推理时投入更多计算

更多推理token = 更深思考 = 更好答案

可按问题难度动态调整!

- **Snell et al. (2024) 关键发现:**

- **7B + 搜索 > 34B 直接推理!**
- 在部分任务上, 小模型+推理搜索
- 超越大4-5倍的模型直接回答

- **计算最优策略是自适应的:**
- 简单问题: 直接回答最高效
- 困难问题: 多步推理+验证值得
- → 未来的系统需要"预估难度"

- **系统影响:**
- 推理延迟大幅增加 (秒→分钟)
- KV缓存暴增 (思考链数千token)
- 推理成本3-10×增加
- 需要新的serving架构

关键要点: Test-Time Scaling开启新维度: 推理计算量成为可调参数, 小模型+更多思考 > 大模型+快回答

推理模型的系统挑战

挑战	标准LLM	推理模型 (o1/R1)	增加倍数
输出token数	100-500	1000-10000+	10-20×
推理延迟	1-5秒	10-120秒	10-100×
KV缓存/请求	~100MB	~1-2GB	10-20×
推理成本/query	$0.01	**$0.03-0.10**	**3-10×**
并发用户数	高	低 (长占用GPU)	↓

KV缓存爆炸

64K context, 推理模型:
单用户KV缓存可达15.75GB
传统serving架构失效

调度复杂度

请求延迟差异巨大:
简单问题2秒 vs 困难问题2分钟
需要自适应调度策略

成本控制

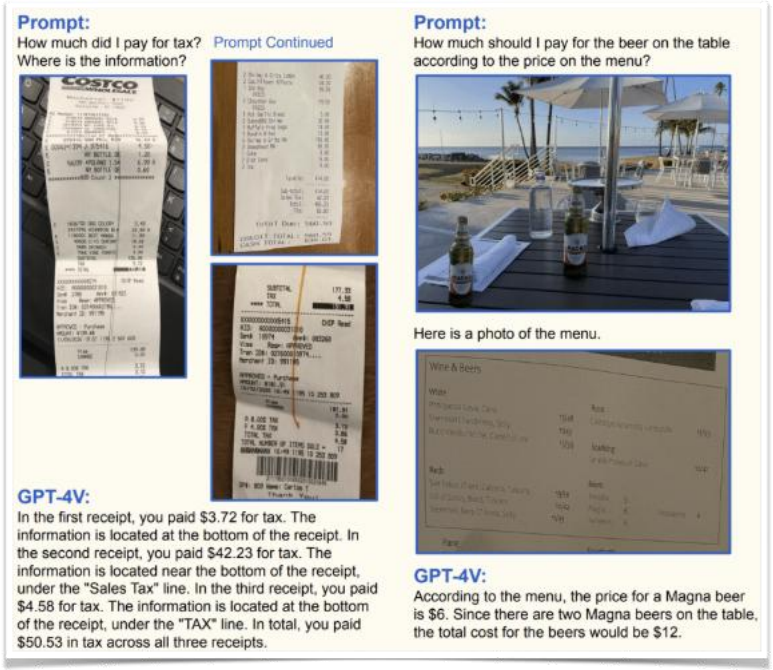
按output token计费
推理模型成本不可预测
需要"何时停止思考"机制

关键要点: 推理模型将LLM serving从"高吞吐短请求"变为"低吞吐长请求"，需要全新的系统设计思路

PART 08

多模态LLM

让LLM"看见"世界: 视觉与语言的融合



为什么要多模态?

视觉问答

上传图片提问
GPT-4V, Claude Vision

自动驾驶

理解路况图像
处理边界场景

编程助手

理解UI截图
生成代码

机器人控制

视觉→动作
RT-2, PaLM-E

- **系统视角: 多模态增加了什么?**
- • 输入: 需要Vision Encoder (如ViT) → 额外计算和参数
- • 序列长度: 一张图 = 数百个visual token → attention计算量↑
- • 训练数据: 需要图文配对数据, 收集成本高
- • 推理: 图像预处理 + 视觉编码 + LLM生成, 延迟更长

关键要点: 多模态LLM将视觉理解集成到语言模型，代价是更长的序列、更多的计算和更复杂的训练流程

两条技术路线: 交叉注意力 vs 视觉Token

路线1: 交叉注意力

(Flamingo, 2022)

- 冻结LLM, 插入交叉注意力层
- 视觉特征通过cross-attention注入

- **优点:**
- LLM权重不变, 保留语言能力
- 可精确控制视觉信息流入量

- **缺点:**
- 增加额外的attention层和参数
- 架构复杂, 训练成本高
- 不够灵活 (视觉位置固定)

路线2: 视觉Token

(LLaVA, PaLM-E, 2023)

- 将图像编码为token序列
- 和文本token一起输入LLM

- **优点:**
- 架构最简单 (仅加投影层)
- 复用LLM的全部能力
- 灵活: 图文交错输入

- **缺点:**
- 图像token多 → 序列变长
- 一张图 = 256-576个token
- → 当前主流方案

关键要点: 视觉Token路线因其简单统一而成为主流，LLaVA证明"ViT + 投影层 + LLM"就够了

Flamingo: 交叉注意力视觉语言模型

开创性工作: 首个在冻结LLM中成功集成视觉输入的模型

- **架构设计:**
- 冻结预训练的LLM (如Chinchilla-80B)
- 在LLM中间层插入交叉注意力层
- 只训练新增的交叉注意力参数

- **Perceiver Resampler:**
- 将不同大小的图像特征图
- 转换为固定数量的视觉token (64个)
- 减少视觉token数量, 降低计算成本

- **门控交叉注意力:**
- output = LLM_layer(x) + tanh(α) · CrossAttn(x)
- α初始化为0 → 训练开始时=纯LLM
- 渐进式引入视觉信息, 训练更稳定

能力:

少样本视觉学习

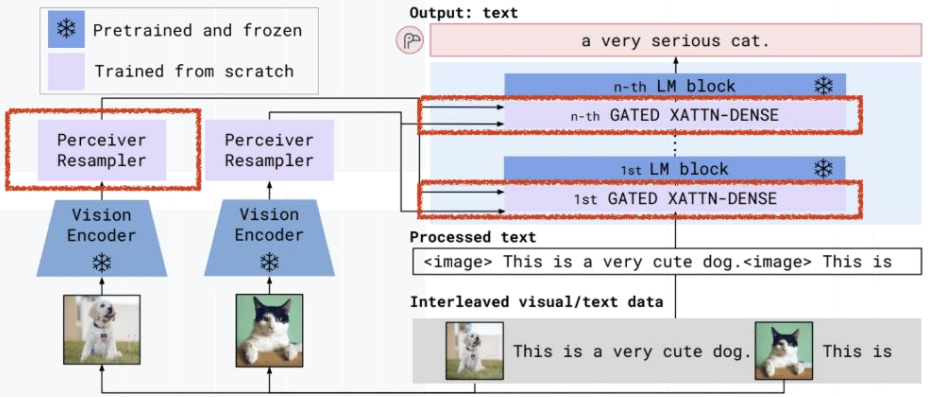
给几个图文示例, 即可理解新视觉任务

视觉对话

围绕图像进行多轮对话

多图推理

同时理解和比较多张图像

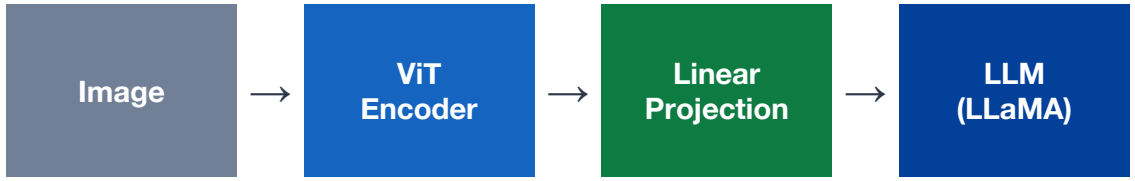


关键要点: Flamingo的门控设计巧妙: α=0起步保证初始=纯LLM, 渐进引入视觉, 但架构复杂度限制了推广

LLaVA & 视觉Token路线: 简单即正义

LLaVA证明: Vision Encoder + 一个线性投影层 + LLM = 够了!

LLaVA架构:



- **关键简化:**
- 不需要交叉注意力层
- 不需要Perceiver Resampler
- 只需一个线性投影: Wv ∈ ℝ^{d_v × d_llm}
- 图像token和文本token同等对待

- **训练策略:**
- Stage 1: 冻结ViT+LLM, 只训练投影层
- Stage 2: 解冻LLM, 在指令数据上微调

视觉Token方案对比:

模型	视觉编码器	LLM
LLaVA	ViT-L/14	LLaMA-13B
LLaVA-1.5	ViT-L/14@336	Vicuna-13B
GPT-4V	未公开	GPT-4
PaLM-E	ViT-22B	PaLM-540B
Qwen-VL	ViT-G/14	Qwen-7B

系统含义:

一张图 = 256-576个visual token

10张图 → 额外5760个token

→ 序列长度↑ → 注意力计算$O(n^2)$↑↑

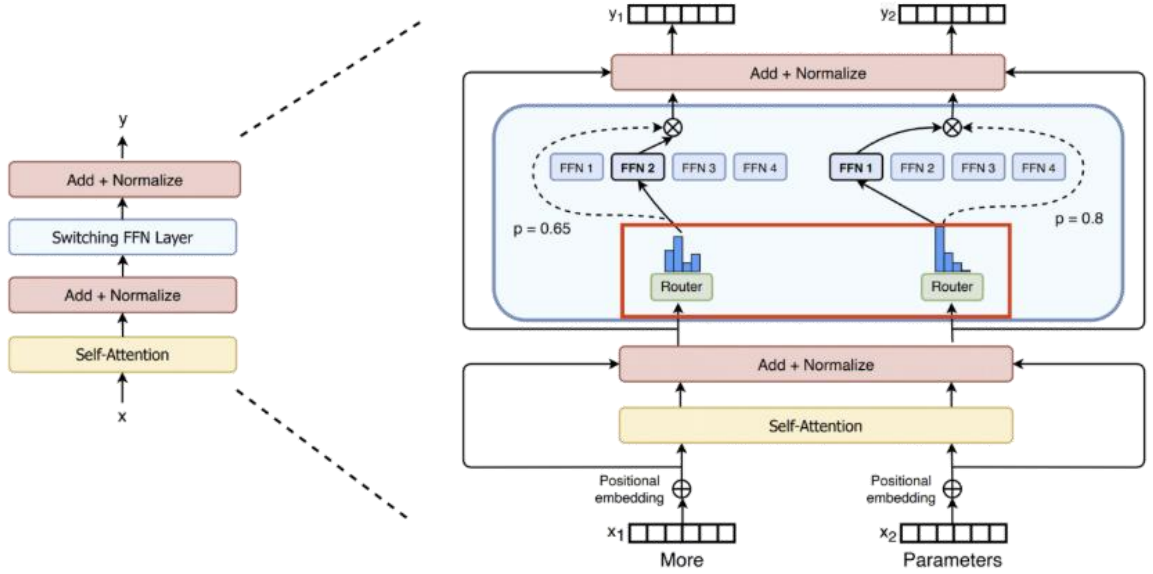
→ KV缓存↑ → 视觉token压缩是重要方向

关键要点: LLaVA的极简设计启示: 不要过度设计，简单的线性投影+端到端训练就能达到很好效果

PART 09

MoE: 稀疏计算

用稀疏性突破参数量限制: 更多参数, 不增加推理成本



MoE: Mixture of Experts

核心思想: 每个token只使用一部分参数，总参数很大, 但每次推理激活的参数很少

- **MoE替代标准FFN:**
- 标准: 1个大FFN, 每个token都用
- MoE: N个小FFN (专家), 每token选Top-K

- **组成:**
- Router (路由器): 决定每个token用哪些专家
- Experts (专家): N个独立的FFN
- $G(x) = \text{Softmax}(W_{router} \cdot x) \rightarrow \text{Top-K}$

- **计算过程:**
- 1. Router给每个专家打分
- 2. 选Top-K个专家 (通常K=2)
- 3. 加权求和专家输出
- $MoE(x) = \Sigma_i g_i \cdot Expert_i(x)$

实例对比:

	Mixtral 8×7B	DeepSeek-V3
总参数	46.7B	671B
专家数	8	256
活跃专家	2 (Top-2)	8 (Top-8)
活跃参数	**12.9B**	**37B**
性能≈	LLaMA-2-70B	GPT-4级别
推理FLOPs≈	13B Dense	37B Dense

核心价值: 671B知识容量, 37B推理成本
用稀疏性打破参数-计算的线性关系

关键要点: MoE = 大模型的知识 + 小模型的推理成本，通过稀疏激活打破参数量与计算量的线性关系

MoE: 路由与负载均衡

挑战: 路由器如何公平分配? 如果所有token都选同一个专家 → 其他专家浪费

- **容量因子 (Capacity Factor):**
- 每个专家最多处理的token数:
- capacity = (n_tokens / n_experts) × C

- C=1.0: 完美均匀分配
- C=1.5: 允许50%的不均匀性
- 超出容量的token被丢弃!

- **辅助损失 (Auxiliary Loss):**
- 鼓励路由器均匀分配:
- $L_{aux} = \alpha \cdot \Sigma_i f_i \cdot p_i$
- f_i = 专家i实际接收的token比例
- p_i = 路由器给专家i的平均概率
- 目标: 让所有 $f_i \approx 1/N$

路由策略演进:

Token Choice

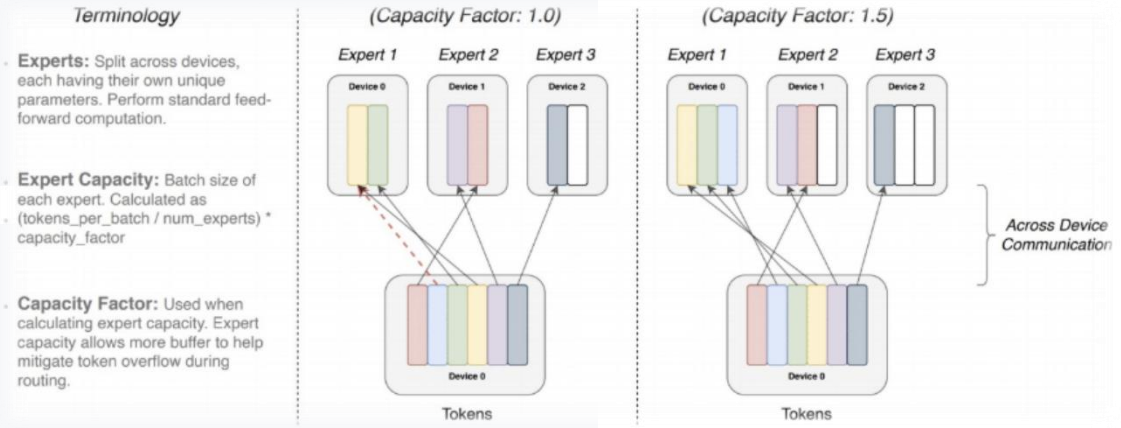
每个token选Top-K专家
(Mixtral使用)

Expert Choice

每个专家选Top-K个token
天然负载均衡

DeepSeek细粒度

256个小专家, Top-8
共享专家 + 路由专家
更好的负载分布



关键要点: 负载均衡是MoE的核心系统问题，不均衡 = GPU闲置 = 浪费算力, 辅助损失+细粒度路由是主要解决方案

MoE系统挑战

负载不均 → GPU闲置

路由不均导致部分GPU满载
其他GPU空闲等待
批内不同token路由到不同专家
→ 辅助损失 + 容量因子缓解

通信开销

专家并行: 不同专家在不同GPU
每个token需发送到目标GPU
All-to-All通信模式
→ 高带宽互联(NVLink)是前提

内存: 所有专家都要加载

Mixtral 8×7B: 推理FLOPs=13B
但需加载全部47B参数!
671B MoE更需要多卡分布
→ 内存节省不如计算节省显著

Dense vs MoE权衡: MoE节省计算但不节省内存, 且增加通信复杂度，适合训练和高吞吐推理, 不适合单卡/低延迟场景

关键要点: MoE的系统挑战: ①负载均衡 ②All-to-All通信 ③全参数加载，是分布式系统设计的典型问题

PART 10

系统视角总结与展望

Transformer推理的核心瓶颈与优化方向

Prefill vs Decode: 推理的两个阶段

关键区分: LLM推理分为两个截然不同的阶段, 瓶颈完全不同, 需要不同的优化策略

Prefill (预填充)

计算密集型 (Compute-bound)

- 并行处理整个prompt
- 所有token同时计算attention
- GPU计算单元满载运行

- **瓶颈: 计算量**
- n个token → $O(n^2d)$计算
- GPU算力是限制因素

- **优化方向:**
- FlashAttention (减少内存访问)
- Tensor并行 (多卡分摊计算)

Decode (解码)

访存密集型 (Memory-bound)

- 每次只生成1个token
- 但要访问全部KV缓存
- GPU大量时间在等待内存读取

- **瓶颈: 内存带宽**
- 每步读取全部模型权重+KV缓存
- 计算量很小但数据搬运量大

- **优化方向:**
- GQA (减少KV缓存大小)
- 量化 (减少每次读取的数据量)
- Speculative Decoding (投机解码)

关键要点: Prefill优化计算, Decode优化带宽, 同一个模型两个阶段需要完全不同的系统优化策略

训练 vs 推理: 系统挑战对比

维度	训练	推理
核心目标	吞吐量 (tokens/sec)	延迟 + 吞吐量平衡
批大小	大 (数千~数万)	动态 (1~数百)
计算类型	前向+反向, 计算密集	前向only, 访存密集(decode)
内存瓶颈	激活值+梯度+优化器状态	KV缓存 (随序列增长)
并行策略	数据/模型/Pipeline并行	Tensor并行 + 连续批处理
精度	混合精度 (bf16/fp32)	INT8/INT4量化
通信	梯度同步 (AllReduce)	KV缓存分发 (MoE)
频率	一次性 (数周~数月)	持续 (7×24, 数年)
成本占比	~6%	**~94%**

关键要点: 训练是一次性工程, 推理是持续运维, 后续章节将分别深入训练并行策略和推理优化技术

代码: 带KV Cache的推理循环

```python
@torch.no_grad()
def generate(model, prompt_ids, max_new=100):
    """自回归生成 + KV Cache"""
    kv_cache = None

    # ★ Phase 1: Prefill (并行处理prompt)
    logits, kv_cache = model(
        prompt_ids,              # 全部prompt
        kv_cache=None            # 首次无缓存
    )  # kv_cache: 每层存(K, V)

    next_id = logits[:, -1].argmax(-1, keepdim=True)
    output_ids = [next_id]

    # ★ Phase 2: Decode (逐token生成)
    for _ in range(max_new - 1):
        logits, kv_cache = model(
            next_id,             # ★ 只传新token!
            kv_cache=kv_cache    # ★ 复用缓存
        )
        next_id = logits[:, -1].argmax(-1, keepdim=True)
        if next_id.item() == EOS_ID:
            break
        output_ids.append(next_id)

    return torch.cat(output_ids, dim=-1)
```

- **代码解读:**

- **Prefill (L6-10):**
- 一次前向处理全部prompt
- 计算密集: $O(n^2d)$计算量
- 生成KV Cache供后续使用

- **Decode (L15-23):**
- 每次只传入1个新token
- 访存密集: 读取全部KV Cache
- KV Cache持续增长

- **关键优化点:**
- L18: 只传新token (not全序列)
- L19: 复用已有KV Cache
- → Decode每步 $O(nd)$而非$O(n^2d)$

关键要点: KV Cache让decode从$O(n^2)$降到$O(n)$，但代价是线性增长的内存占用，这就是推理优化的核心矛盾

课程总结: 3个核心要点

01

$O(n^2)$是核心挑战

自注意力的二次复杂度是
Transformer所有系统问题的根源:
长序列计算爆炸, KV缓存线性增长,
FlashAttention和稀疏注意力都是
在对抗这个$O(n^2)$

02

每个改进源于系统瓶颈

RoPE → 解决长度泛化
GQA → 解决KV缓存过大
SwiGLU → 解决FFN表达力
MoE → 解决参数-计算耦合
不是随意改进, 是被瓶颈逼出来的

03

推理是更大的系统问题

训练一次, 推理永远:
成本占比94%, 推理模型加剧
推理 = Prefill(计算密集)
+ Decode(访存密集)
两阶段需要完全不同的优化

关键要点: 理解Transformer的系统特性, 是优化训练、推理、部署的前提, 后续章节将逐一攻克这些系统挑战