llama系列模型

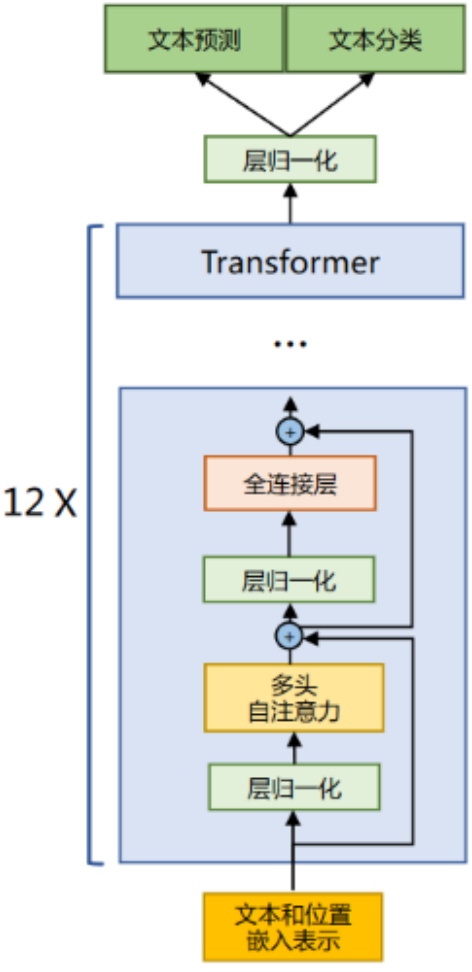
1.LLama

1.1 简介

Open and Efficient Foundation Language Models (Open但没完全Open的LLaMA)

2023年2月，Meta（原Facebook）推出了LLaMA大模型，使用了1.4T token进行训练，虽然最大模型只有65B，但在相关评测任务上的效果可以媲美甚至超过千亿级大模型，被认为是近期开源大模型百花齐放的开端之一，“羊驼”系列模型及其生态快速发展。

LLaMA 所采用的 Transformer 结构和细节，与标准的 Transformer 架构不同的地方包括采用了**前置层归一化（Pre-normalization）**并使用 **RMSNorm 归一化函数** （Normalizing Function）、激活函数更换为 **SwiGLU**，并使用了**旋转位置嵌入（RoP）**，整体 Transformer 架构与 GPT-2 类似。



1.2 RMSNorm 归一化函数

为了提升模型训练过程中的稳定性与收敛速度，GPT-2 相较于原始 GPT 模型在结构上进行了多项优化。其中一项重要的改进是引入了**前置层归一化（Pre-Layer Normalization）**策略。具体而言，GPT-2 将第一个层归一化操作提前到了多头自注意力模块之前，第二个层归一化则置于前馈全连接网络之前。同时，残差连接的位置也相应调整到各自模块的输出之后。

Pre-LayerNorm 的优势：

1.梯度更稳定： 归一化发生在子层输入之前，有助于缓解梯度爆炸/消失。

2.更容易训练深层模型： LLaMA、GPT-2 等都采用了 Pre-LayerNorm 来支撑更深的模型。

3.推理时表现更好：一些研究发现 Pre-LayerNorm 在推理阶段比 Post 更鲁棒。

在这些归一化操作中，GPT-2 使用了 **RMSNorm**（均方根归一化）方法，相较于传统的 LayerNorm，RMSNorm 具有更轻量级的计算复杂度，并且在实际应用中表现出良好的性能表现。**RMSNorm 与 LayerNorm 的区别在于前者仅使用输入的均方值（RMS）进行缩放而不减去均值，而后者同时使用均值和方差对输入进行标准化。**

> **Note**
>
> RMSNorm 看作是只“归一化向量长度”，不“移动中心点”；而 LayerNorm 则是既“居中”又“标准化”。

RMSNorm 的数学定义：对于一个输入向量 $a = (a_1, a_2, \ldots, a_n)$，RMSNorm 首先计算其均方根值（Root Mean Square）：

$$\mathrm{RMS}(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$$

然后对每个输入元素进行归一化处理：

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(a)}$$

为了增强模型表达能力，RMSNorm 还引入了可学习参数：缩放因子 g_i 和偏移参数 b_i，从而得到完整的 RMSNorm 表达式：

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(a)} \cdot g_i + b_i$$

在实际实现中，g_i 和 b_i 是逐通道（channel-wise）或逐特征（feature-wise）的可学习参数，通常初始化为 1 和 0，分别用于缩放和平移归一化后的值，使得模型能够在训练过程中动态调整分布特性。

在 HuggingFace 的 `transformers` 库中，RMSNorm 被广泛应用于多种现代 Transformer 架构中，如 LLaMA、GPT-NeoX 等。其 PyTorch 实现大致如下所示（简化版本）：

```
class LlamaRMSNorm(nn.Module):
  def __init__(self, hidden_size, eps=1e-6):
    """
    LlamaRMSNorm is equivalent to T5LayerNorm
    """
    super().__init__()
    self.weight = nn.Parameter(torch.ones(hidden_size))
    self.variance_epsilon = eps # eps 防止取倒数之后分母为 0

  def forward(self, hidden_states):
    input_dtype = hidden_states.dtype
    variance = hidden_states.to(torch.float32).pow(2).mean(-1, keepdim=True)
    hidden_states = hidden_states * torch.rsqrt(variance + self.variance_epsilon) #
weight 是末尾乘的可训练参数，即 g_i

    return (self.weight * hidden_states).to(input_dtype)
```

1.3 SwiGLU激活函数

SwiGLU 激活函数相较于 ReLU 函数在大多数评测任务中都表现出更好的性能。在 LLaMA 系列模型中，前馈神经网络（Feed-Forward Network, FFN）采用了带有 SwiGLU 激活函数的结构。其计算公式如下：

$$\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) = \mathrm{SwiGLU}(xW, xV)W_2$$

其中，SwiGLU 的定义为：

$$\text{SwiGLU}(x, W, V) = \text{Swish}_\beta(xW) \otimes (xV)$$

这里的 $\otimes$ 表示逐元素相乘（Hadamard product），x 是输入向量，W 和 V 是两个不同的可学习参数矩阵。在 SwiGLU 中，输入会分成两条路径来处理。**第一条路径**是通过一个叫做 Swish 的激活函数，生成一个“门控信号” $g = \text{Swish}_\beta(xW)$。这个信号就像一个开关，决定了信息是否应该被保留或放大。如果 g 的值接近 0，说明这部分信息不太重要，会被抑制；如果 g 接近 1 或者更大，说明这部分信息更重要，会被保留甚至增强。**第二条路径**则是直接对输入进行线性变换 xV，这是真正要传递的内容。最后，这两条路径的结果会逐元素相乘，也就是用门控信号去“调节”内容路径的信息，从而得到最终输出。

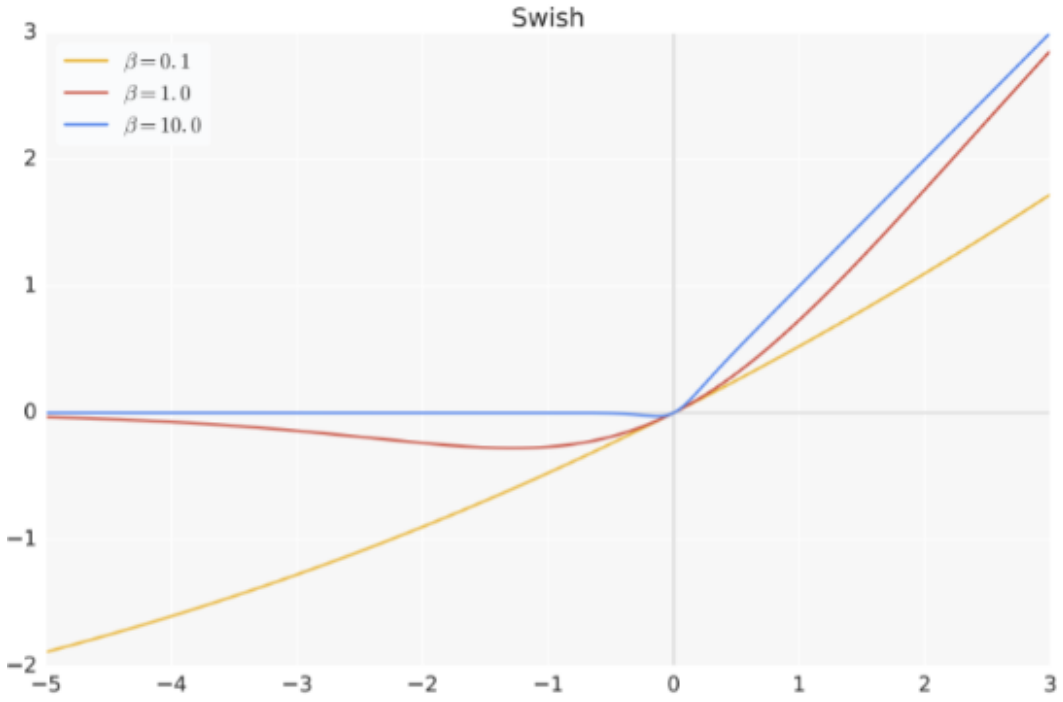
Swish 函数是 SwiGLU 的核心组成部分，其定义为：

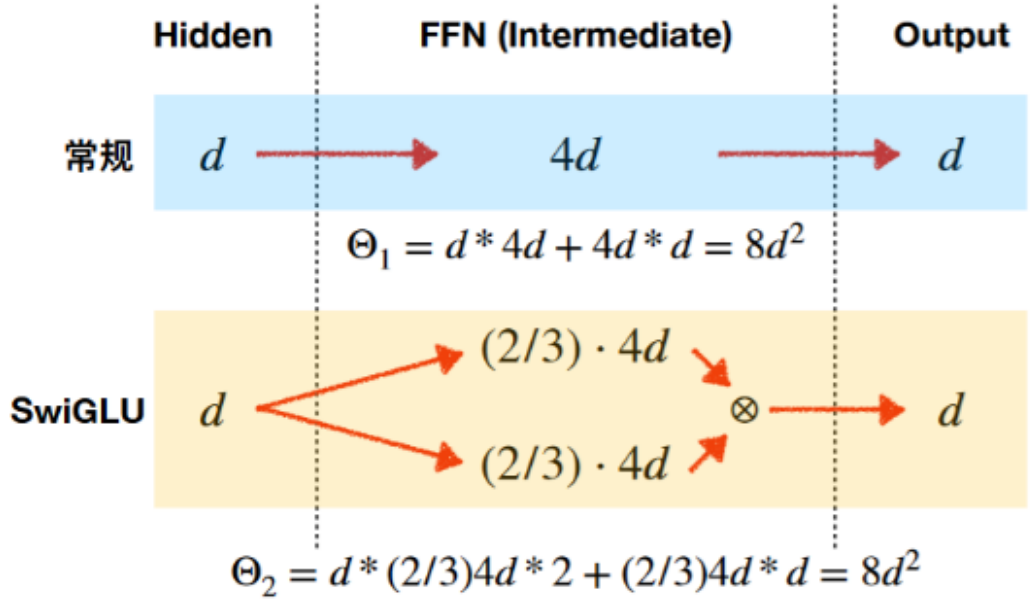
$$\text{Swish}_\beta(x) = x \cdot \sigma(\beta x)$$

其中 $\sigma(x)$ 是 Sigmoid 函数,β 是一个控制函数形状的超参数（也可以是可学习的）：

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Swish 函数的形状会随着参数 β 的变化而改变。当 β 趋近于 0 时，Swish 函数趋近于线性函数 $y = x$；当 β 趋近于无穷大时，它则趋近于 ReLU 函数 $\max(0, x)$。当 $\beta = 1$ 时，Swish 函数是一个光滑且非单调的函数，这种性质有助于缓解梯度消失问题并提升模型的表达能力。





在 HuggingFace 的 Transformers 库中，通常使用 `silu` 函数来实现 Swish(1)，它们在数学上是等价的。SwiGLU 则进一步将 Swish 函数与门控机制结合，使得模型可以动态地控制信息流。具体来说，输入经过两个不同的线性变换后分别作为“门”和“值”，通过 Swish 激活后的门控信号与值进行逐元素相乘，从而决定哪些信息被保留、哪些信息被抑制。

- SwiGLU 的总参数量与常规 FFN 相同，但通过门控机制和更高效的中间表示，提升了模型的性能。
- SwiGLU 的核心在于引入了动态门控机制（$Swish(xW) \otimes (xV)$），增强了模型的表达能力。
- SwiGLU 是 LLaMA 等大模型中常用的 FFN 结构，相比传统 ReLU 更适合大规模语言模型。

这种设计不仅继承了 GLU 类激活函数的优势，也利用了 Swish 函数的平滑性和非单调性，在语言建模等任务中展现出更强的表达能力和更稳定的训练过程。因此，SwiGLU 成为了当前主流大语言模型中广泛采用的一种激活函数形式。

> **Note**
>
> LLaMA中直接将FFN中的ReLU替换为SwiGLU，并将维度放缩为$(2/3) \cdot 4d$

1.4 旋转位置嵌入（Rotary Positional Embedding, RoPE）

在传统的 Transformer 模型中，通常采用**绝对位置编码**来为输入序列中的每个位置赋予一个位置信息。然而，LLaMA 等模型选择使用了一种更先进的位置编码方式——**旋转位置嵌入（RoPE）**，它不仅保留了绝对位置编码的实现便利性，还具备相对位置编码的优势。出发点是**通过绝对位置编码的方式实现相对位置编码。**

RoPE 的核心思想是通过引入**复数变换**的方式，在不显式建模相对位置的前提下，使模型能够隐式地感知到不同 token 之间的相对位置关系。其基本操作是对查询向量 q 和键向量 k 分别施加与位置相关的旋转变换：

$$\tilde{q}_m = f(q, m), \quad \tilde{k}_n = f(k, n)$$

其中，m 和 n 分别表示当前 token 在序列中的位置。经过该变换后，$\tilde{q}_m$ 和 $\tilde{k}_n$ 就分别携带了对应位置的信息。
在二维空间下，RoPE 可以用复数形式表达如下：

$$f(q, m) = R_f(q, m)e^{i\Theta_f(q,m)} = \|q\|e^{i(\Theta(q)+m\theta)} = qe^{im\theta}$$

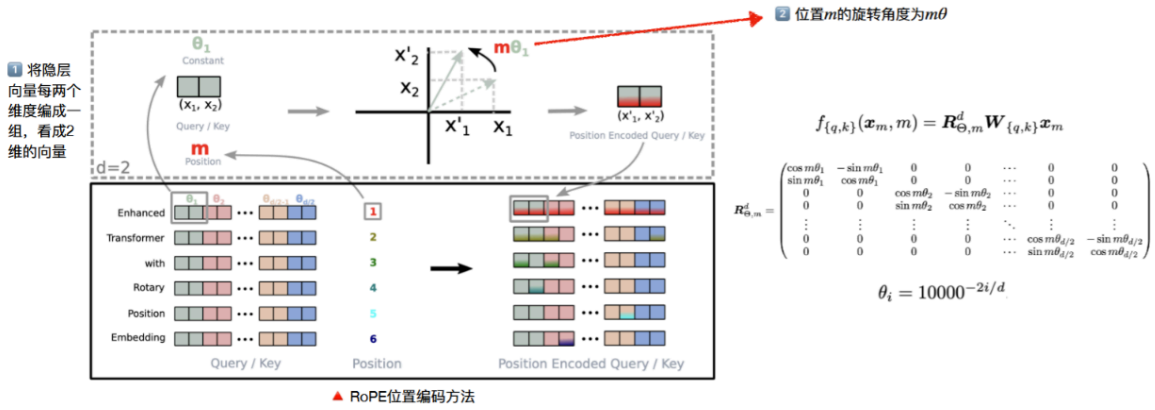
从几何角度看，这个操作相当于将向量 q 在二维平面上**绕原点旋转**了一个与位置 m 相关的角度 $m\theta$，这也是“旋转位置编码”名称的由来。我们也可以将上述复数旋转操作转换为矩阵形式，便于在实际模型中实现：

$$f(q, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \end{pmatrix}$$

这实际上就是一个二维旋转矩阵对向量的线性变换。

为了将其扩展到高维空间，RoPE 采用了“分块拼接”的策略：将整个向量划分为多个二维子向量，并对每个子向量分别应用旋转操作。这样，任意偶数维度的 RoPE 都可以表示为多个二维旋转的组合，其形式如下：

$$f(q,m) = \underbrace{\begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_1 & -\sin m\theta_1 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_1 & \cos m\theta_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2-1} & -\sin m\theta_{d/2-1} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2-1} & \cos m\theta_{d/2-1} \end{pmatrix}}_{R_d} \cdot \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{d-2} \\ q_{d-1} \end{pmatrix}$$



在这个变换矩阵中，每一组两个维度构成一个二维旋转矩阵，它们共享相同的结构但使用不同的角度参数 θ_i 。通过这种方式，RoPE 能够在保持高效计算的同时，为每个位置赋予独特的旋转偏移，从而帮助模型更好地捕捉序列内部的相对位置信息。

这种设计使得 RoPE 具备以下优点：1.支持任意长度的输入序列；2.保留了相对位置信息的能力；3.计算高效，易于集成到标准注意力机制中；4.不需要额外训练，完全由前向计算确定。

RoPE 在 HuggingFace Transformer 库中代码实现如下所示：

```
import torch

def precompute_freqs_cis(dim: int, end: int, constant: float = 10000.0):
    '''
    计算cos和sin的值，cos值在实部，sin值在虚部，类似于 cosx+j*sinx
    :param dim: q,k,v的最后一维，一般为emb_dim/head_num
    :param end: 句长length
    :param constant:  这里指10000
    :return:
    复数计算 torch.polar(a, t)输出， a*(cos(t)+j*sin(t))
    '''
    # freqs: 计算 1/(10000^(2i/d) )，将结果作为参数theta
    # 形式化为 [theta_0, theta_1, ..., theta_(d/2-1)]
    freqs = 1.0 / (constant ** (torch.arange(0, dim, 2)[: (dim // 2)].float() /
dim)) # [d/2]

    # 计算m
    t = torch.arange(end, device=freqs.device)  # [length]
    # 计算m*theta
```

```
    freqs = torch.outer(t, freqs).float()  # [length, d/2]
    # freqs形式化为 [m*theta_0, m*theta_1, ..., m*theta_(d/2-1)],其中
m=0,1,...,length-1

    # 计算cos(m*theta)+j*sin(m*theta)
    freqs_cis = torch.polar(torch.ones_like(freqs), freqs)  # complex64
    # freqs_cis: [cos(m*theta_0)+j*sin(m*theta_0),
cos(m*theta_1)+j*sin(m*theta_1),), ..., cos(m*theta_(d/2-1))+j*sin(m*theta_(d/2-1))]
    # 其中j为虚数单位， m=0,1,...,length-1
    return freqs_cis # [length, d/2]

def reshape_for_broadcast(freqs_cis: torch.Tensor, x: torch.Tensor):
    ndim = x.ndim
    assert 0 <= 1 < ndim
    assert freqs_cis.shape == (x.shape[1], x.shape[-1])
    shape = [d if i == 1 or i == ndim - 1 else 1 for i, d in enumerate(x.shape)] #
(1, length, 1, d/2)
    return freqs_cis.view(*shape) # [1, length, 1, d/2]

def apply_rotary_emb(xq: torch.Tensor, xk: torch.Tensor, freqs_cis: torch.Tensor,):
    # 先将xq维度变为[bs, length, head,  d/2, 2]，利用torch.view_as_complex转变为复数
    # xq:[q0, q1, .., q(d-1)] 转变为 xq_: [q0+j*q1, q2+j*q3, ..., q(d-2)+j*q(d-1)]
    xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2)) # [bs,
length, head, d/2]
    # 同样的，xk_:[k0+j*k1, k2+j*k3, ..., k(d-2)+j*k(d-1)]
    xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))

    freqs_cis = reshape_for_broadcast(freqs_cis, xq_) # [1, length, 1, d/2]
    # 下式xq_ * freqs_cis形式化输出，以第一个为例，如下
    # (q0+j*q1)(cos(m*theta_0)+j*sin(m*theta_0)) = q0*cos(m*theta_0)-
q1*sin(m*theta_0) + j*(q1*cos(m*theta_0)+q0*sin(m*theta_0))
    # 上式的实部为q0*cos(m*theta_0)-q1*sin(m*theta_0)，虚部为
q1*cos(m*theta_0)+q0*sin(m*theta_0)
    # 然后通过torch.view_as_real函数，取出实部和虚部，维度由[bs, length, head, d/2]变为[bs,
length, head, d/2, 2]，最后一维放实部与虚部
    # 最后经flatten函数将维度拉平，即[bs, length, head, d]
    # 此时xq_out形式化为 [实部0，虚部0，实部1，虚部1，..., 实部(d/2-1)，虚部(d/2-1)]
    xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(3) # [bs, length, head, d]
    # 即为新生成的q

    xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(3)
    return xq_out.type_as(xq), xk_out.type_as(xk)

if __name__=='__main__':
    # (bs, length, head, d)
    q = torch.randn((2, 10, 12, 32))  # q=[q0, q1, .., qd-1]
    k = torch.randn((2, 10, 12, 32))
    v = torch.randn((2, 10, 12, 32))
    freqs_cis= precompute_freqs_cis(dim=32, end=10, constant= 10000.0)
    # print(freqs_cis.detach().numpy())

    q_new, k_new = apply_rotary_emb(xq=q, xk=k, freqs_cis=freqs_cis)
    print()
```

2.Alpaca

2.1 简介

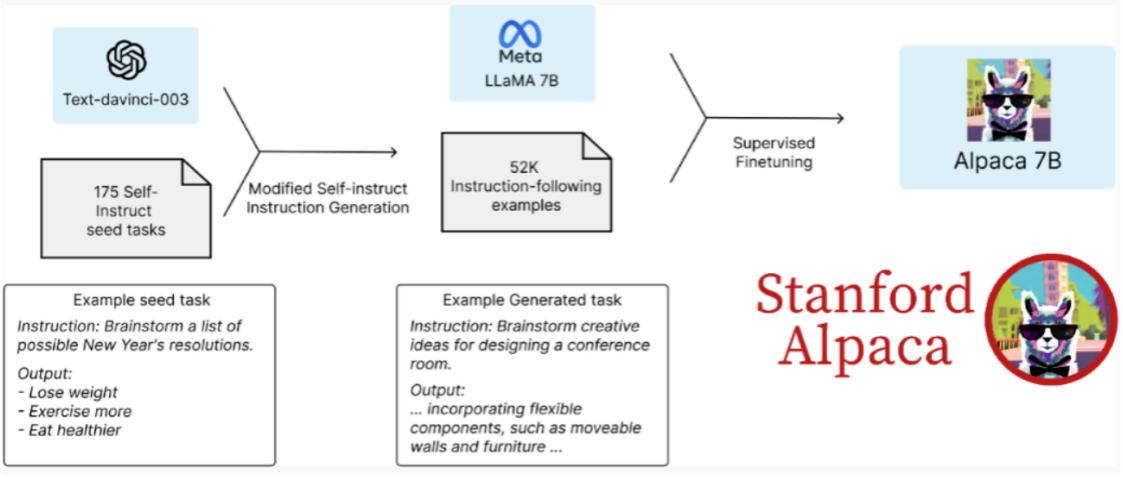
Stanford Alpaca: An Instruction-following LLaMA Model。
Alpaca是在**LLaMA基础上使用52K指令数据精调的预训练模型**，作者只用了不到600美元的成本训练出了该模型（数据$500 + 机器$100）。初步实验结果表明Alpaca可以达到与OpenAI text-davinci-003相匹敌的效果

Self-Instruct 流程概览

步骤	内容	目的
第一步：种子任务	手动创建175个高质量任务	提供模板和范例
第二步：指令爬取	利用种子任务提示 GPT-3 自动生成更多任务	扩展到52,000条指令数据
第三步：指令精调	用这52K数据 fine-tune LLaMA	得到具备指令跟随能力的 Alpaca

2.2 微调方法

1. 第一步：构造175条self-instruct 种子示例任务
2. 第二步：基于上述种子任务，利 用text-davinci-003爬取指令数据
3. 第三步：使用爬取下来的52K指令数据在LLaMA上进行精调，最终得到Alpaca



2.3 Self-instruct数据构造

首先由人工构造175条种子数据，Self-Instruct 种子示例任务（Seed Tasks）是指人工精心设计的一小批多样化的指令-响应对（instruction-output pairs），作为后续自动生成大量指令数据的“种子”或起点。

```
{
  "id": "seed_task_25",
  "name": "perfect_numbers",
  "instruction": "Find the four smallest perfect numbers.",
  "instances": [{ "input": "", "output": "6, 28, 496, and 8128"}],
  "is_classification": false
}
```

将“爬取要求”和种子数据进行适当组合，送入textdavinci-003，要求生成类似的指令数据。要求包括：提升指令多样性、包含真实数据、字数 要求、语言要求、拒绝不合适指令等

2.4 指令数据格式

- `instruction`：描述模型需要执行的指令内容
- `input`（可选）：任务上下文或输入信息，例如当指令是“对文章进行总结”，则input是文章内容
- `output`：由text-davinci-003生成的针对指令的回复

```
Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

### Instruction:
{instruction}

### Input:
{input}

### Response:
```

▲ 包含“input”字段的指令模板

```
Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

### Instruction:
{instruction}

### Response:
```

▲ 不含“input”字段的指令模板

3.Llama-2

3.1 简介

Llama 2: Open Foundation and Fine-Tuned Chat Models

2023年7月，Meta推出了Llama-2开源大模型，并且推出了Llama-2-Chat对话模型。与一代LLaMA主要区别体现在**更多的训练数据、更长的上下文窗口、GQA技术**等

对比项	LLaMA	Llama-2
模型类型	基座模型	基座模型、对话模型（Llama-2-Chat）
授权形式	受限，不可商用，不可二次分发	宽松，可商用（有条件），可二次分发
模型参数量	7B / 13B / 33B / 65B	7B / 13B / 34B（暂缓开源） / 70B
训练数据来源	CC, CC4, GitHub, Wikipedia, Books, arXiv, Stack Exchange	新融合的数据，重点删除不合规的数据
主要覆盖语种	拉丁语系和西里尔语系	
训练数据量 (tokens)	1.0T (7B/13B), 1.4T (33B/65B)	2.0 T
上下文长度	2048	4096
词表大小	32000	
GQA技术	无	有：34B / 70B

模型结构的变动主要是体现在**GQA**和**FFN**缩放上：

1.MHA改成GQA：整体参数量会有减少

2.FFN模块矩阵维度有扩充：增强泛化能力，整体参数量增加

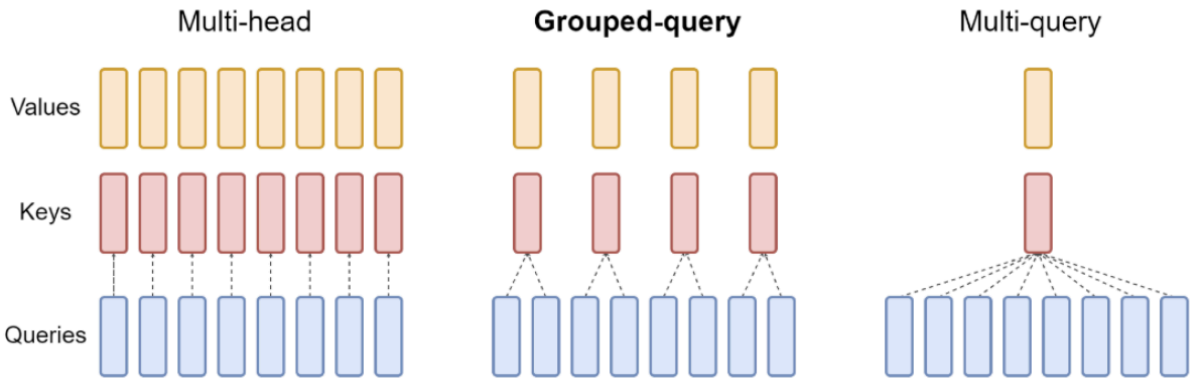
3.上下文长度是llama两倍(长度从2048->4096) 训练语料增加约 40%，体现在1.4T->2.0T的Tokens llama2-34B和llama2-70B使用了GQA，加速模型训练和推理速度

3.2 GQA

GQA和MQA都是注意力的变体，其中多个查询头关注相同的键和值头，以减少推理过程中 KV 缓存的大小，并可以显著提高推理吞吐量。

MHA、GQA、MQA的区别和联系，具体的优点如下：

- `Mutil-Head Attention` 因为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。
- `Multi-Query Attention` 多个头之间可以共享KV对，因此速度上非常有优势，实验验证大约减少30-40%吞吐。
- `Group Query Attention` 没有像MQA那么极端，将query分组，组内共享KV，效果接近MQA，速度上与MQA可比较



Llama-2中使用了8个KV映射，即GQA-8，**GQA在多数任务上与MHA效果相当，且平均效果优于MQA；GQA和MQA均比MHA有更好的吞吐量**

3.3 源码

```python
def forward(self, x: torch.Tensor, start_pos: int,
    freqs_cis: torch.Tensor, mask: Optional[torch.Tensor],
):
    bsz, seqlen, _ = x.shape
    xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)

    xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
    xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
    xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)

    xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis)

    self.cache_k = self.cache_k.to(xq)
    self.cache_v = self.cache_v.to(xq)

    self.cache_k[:bsz, start_pos : start_pos + seqlen] = xk
    self.cache_v[:bsz, start_pos : start_pos + seqlen] = xv

    keys = self.cache_k[:bsz, : start_pos + seqlen]
    values = self.cache_v[:bsz, : start_pos + seqlen]

    # repeat k/v heads if n_kv_heads < n_heads
    keys = repeat_kv(keys, self.n_rep)  # (bs, seqlen, n_local_heads, head_dim)
    values = repeat_kv(values, self.n_rep)  # (bs, seqlen, n_local_heads, head_dim)

    xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
    keys = keys.transpose(1, 2)
    values = values.transpose(1, 2)
    scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
    if mask is not None:
        scores = scores + mask  # (bs, n_local_heads, seqlen, cache_len + seqlen)
    scores = F.softmax(scores.float(), dim=-1).type_as(xq)
    output = torch.matmul(scores, values)  # (bs, n_local_heads, seqlen, head_dim)
    output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)
    return self.wo(output)
```

计算Q, K, V

在Q, K上添加RoPE位置信息

更新K, V缓存

提取K, V缓存

启用GQA时，对K, V缓存复制

注意力计算

4.Code Llama

4.1 简介

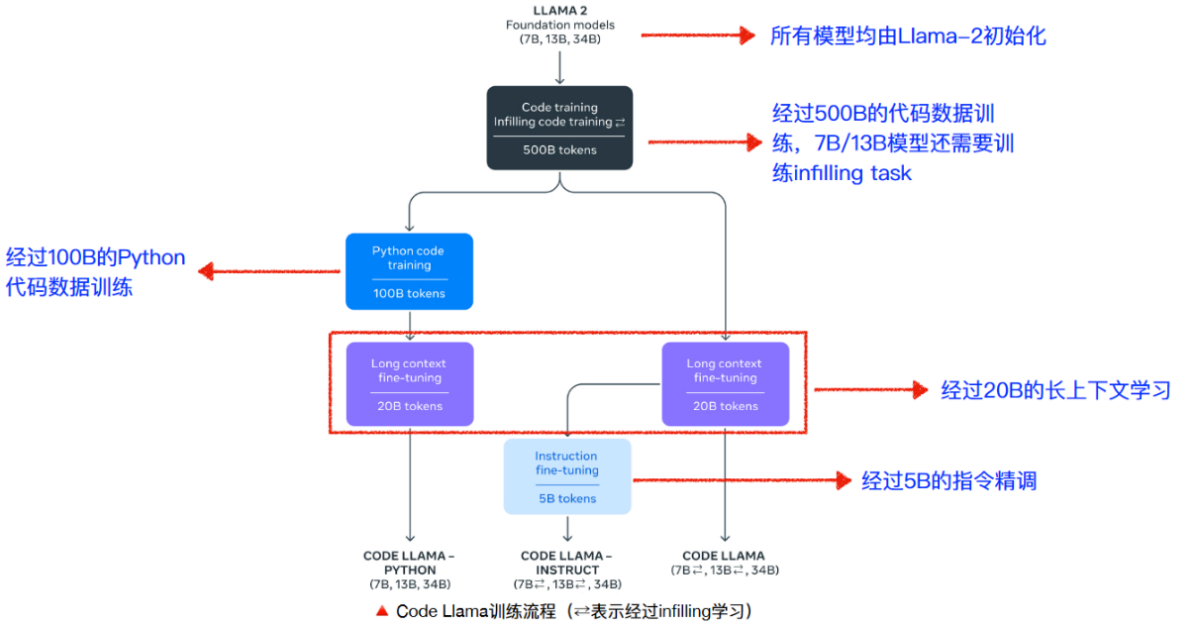
2023年8月24日，Meta推出了面向代码的可商用大模型Code Llama，包含三个大小版本（7B/13B/34B）

支持多种编程语言，包括Python、C++、Java、PHP、Typescript (Javascript)、C#和Bash

亮点：

- 免费供学术研究和商用
- 支持100K上下文
- “神秘”34B版接近GPT-4效果

4.2 模型训练流程



▲ Code Llama训练流程（⇄表示经过infilling学习）

4.3 Code Infilling Task （7B/13B only）

任务目标：根据代码的上下文，预测残缺部分的代码

方法：

- 从完整的代码中选择一部分进行掩码（mask）并替换为`<MASK>`符号，构成上下文
- 利用自回归的方法，根据上下文信息预测解码出被mask的代码部分

Original Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Masked Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        <MASK:0> in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
<MASK:0> word_counts = {}
        for line in f:
            for word in line.split():
                if word <EOM>
```

5.总结

LLaMA

- 开源大模型繁荣发展的开端，一系列相关工作均基于LLaMA开展
- 模型规模7B、13B、33B、65B满足了开发者和研究者的不同需求

Alpaca：通过少量的指令精调赋予LLaMA指令理解与执行的能力

Llama-2

- LLaMA的二代模型，相关模型性能进一步提升，模型可商用
- 推出官方对齐的Chat版本模型，采用了完整的RLHF链条

Code Llama：专注于代码能力的LLaMA模型，最好的模型代码能力接近GPT-4效果，模型可商用