llama源码阅读

transformers-main\src\transformers\models\llama modeling_llama.py

1. llama attention

输入：

$ X \in \mathbb{R}^{B \times T \times D} $

其中，

- (B) = batch size
- (T) = sequence length
- (D) = hidden dimension

计算：

$ Q = X W^Q, \quad K = X W^K, \quad V = X W^V, $

其中权重矩阵：

$ W^Q, W^K, W^V \in \mathbb{R}^{D \times d_k} $

$ Q, K, V \in \mathbb{R}^{B \times T \times d_k} $

其中一般$d_k=D$(单头时) 或者$d_k=D/h$(多头时)，这样设计是为了保证 多头拼接后维度与输入维度相同，便于残差连接等操作

多头时，需要将 (Q, K, V) reshape 成多头形式：

$ Q, K, V \in \mathbb{R}^{B \times T \times h \times d_k}, \quad $

```
Q = Q.view(B, T, h, d_k).permute(0, 2, 1, 3)  # (B, h, T, d_k)
K = K.view(B, T, h, d_k).permute(0, 2, 1, 3)
V = V.view(B, T, h, d_k).permute(0, 2, 1, 3)
```

2.llamaMLP

```
class LlamaMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size  # 输入/输出的隐藏维度
        self.intermediate_size = config.intermediate_size  # 中间层的扩展维度（一般是隐藏维度的几倍）

        # gate_proj 通常和 up_proj 一起用于 Gated Linear Units (GLU) 结构
```

```
        self.gate_proj = nn.Linear(self.hidden_size, self.intermediate_size,
bias=config.mlp_bias)

        # up_proj 提供另一路输入用于门控机制，与 gate_proj 的激活结果逐元素相乘
        self.up_proj = nn.Linear(self.hidden_size, self.intermediate_size,
bias=config.mlp_bias)

        # down_proj 将扩展后的中间表示重新映射回 hidden_size（投影到原始维度）
        self.down_proj = nn.Linear(self.intermediate_size, self.hidden_size,
bias=config.mlp_bias)

        # 激活函数，如 SiLU、GELU，来自配置；ACT2FN 是映射激活函数字符串到实际函数的字
典
        self.act_fn = ACT2FN[config.hidden_act]

    def forward(self, x):
        """
        前向传播过程：
        - gate_proj(x)：一部分输入经过线性变换 → 激活（如 SiLU）
        - up_proj(x)：另一部分输入经过线性变换（无激活）
        - 两者逐元素相乘，形成门控输出
        - down_proj：将门控输出压缩回 hidden_size
        """
        # Gated MLP 模块：激活(gate_proj(x)) * up_proj(x)
        hidden = self.act_fn(self.gate_proj(x)) * self.up_proj(x)

        # 线性投影回原始维度
        output = self.down_proj(hidden)
        return output
```

3.llamaRMSNorm

RMSNorm 相比LayerNorm 省去了：一个 .mean() 函数（均值），一个平方差计算，一次减法操作，一组偏置参数，从而更轻量

它标准化的过程只考虑了“能量”（平方和），没有去中心化。

为什么不减均值？ 对称性不足为惧：在自然语言处理任务中，Transformer 的输入数据是上下文相关嵌入，均值本身就不是一个稳定特征； 深层网络中均值不重要：研究表明，在深层网络中，去中心化（减均值）对最终性能帮助并不显著，甚至会打破激活的结构性信息（参考 Zhang et al., 2020）；主要目标是防止数值爆炸/消失，而 RMS 已足够。

RMSNorm：

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

```
\text{RMSNorm}(x) = \frac{x}{\text{RMS}(x) + \varepsilon} \cdot \gamma
```

LayerNorm：

```
\mu = \frac{1}{n} \sum_{i=1}^{n} x_i
```

```
\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2
```

```
\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta
```

对比项	LayerNorm	RMSNorm
是否去均值（中心化）	☑ 是，减去均值 μ	✖ 否，不减均值
方差计算	使用方差 $\sigma^2 = \frac{1}{n} \sum (x_i - \mu)^2$	使用均方 $\text{RMS}^2 = \frac{1}{n} \sum x_i^2$
偏置参数 β	☑ 有 γ 和 β 两个参数	☑ 仅使用缩放参数 γ
操作复杂度	更高（需要均值、方差、开根等）	更低（只需平方和和开根）
典型应用	GPT-2、BERT 等 Transformer 模型	LLaMA、T5、RWKV 等轻量高效模型

2.llama decoder

继承自 GradientCheckpointingLayer：用于节省显存的技术，将中间激活缓存换成重新计算。

Step 1：Attention 前归一化 + 残差

python

```
residual = hidden_states
hidden_states = self.input_layernorm(hidden_states)
```

对输入做 RMSNorm（PreNorm 架构）,保存 residual 用于残差连接

Step 2：自注意力模块（含 KV 缓存、位置编码）

```
hidden_states, self_attn_weights = self.self_attn(...)
```

输入进入 self.self_attn，这个模块会执行：

多头注意力计算

处理 KV 缓存（past_key_value、cache_position）

融合 FlashAttention 或标准Attention实现

注意它支持多种推理优化（如 FlashAttention）

```
hidden_states = residual + hidden_states
```

加上残差连接（Residual）

Step 3：MLP 前归一化 + MLP + 残差

```
residual = hidden_states
hidden_states = self.post_attention_layernorm(hidden_states)
hidden_states = self.mlp(hidden_states)
hidden_states = residual + hidden_states
```

再次 RMSNorm → 输入 MLP

MLP 是两层全连接（通常是 hidden_size → 4x → hidden_size）

残差连接叠加