

从零实现大语言模型

Lec3: Transformer 架构的代码实现

Penghao Kuang

SIST, ShanghaiTech
ACM, DataTech, Geekpie

2025.11.23

Overview

宏观系统架构

基本运算组件的代码实现

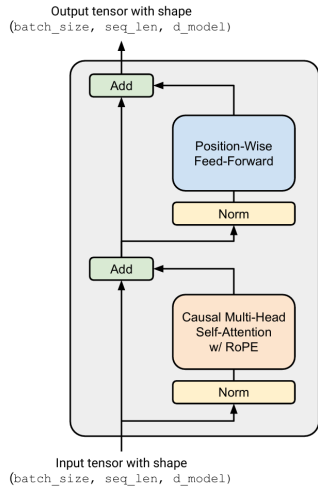
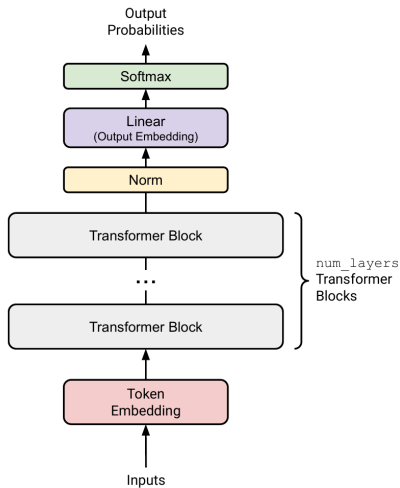
RoPE 变换的代码实现

FFN 与 Attention 的代码实现

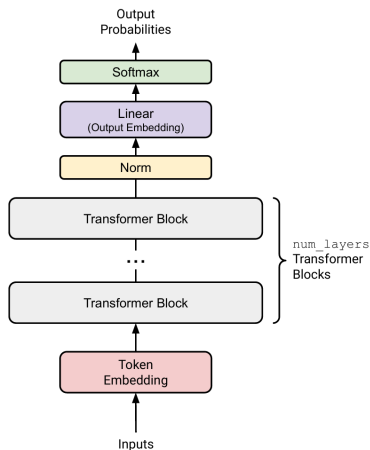
Transformer Block 与最终模型的组装

PART1: 宏观系统架构

Transformer 总体架构图示



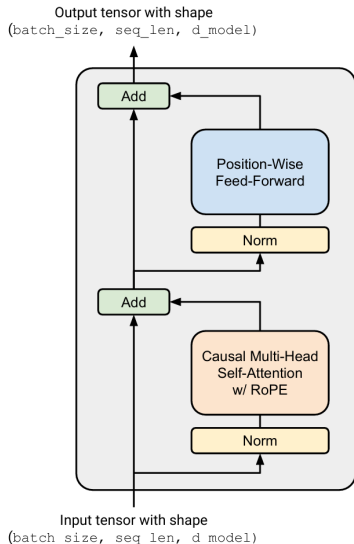
各模块结构的梳理



对于初始输入张量，要经过：

- 一个 Token Embedding 模块，转化成模型可处理的张量格式
- 若干个 Transformer Block 模块，进行多层信息吸收
- 一个 RMSNorm 模块
- 最终的线性激发模块，计算词表中各词权重得分

各模块结构的梳理

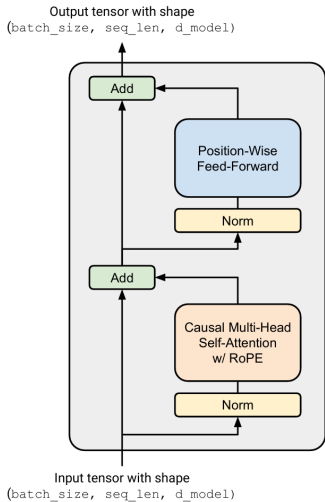
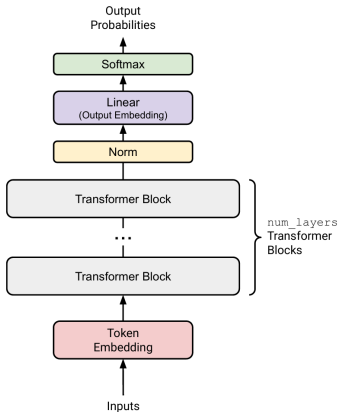


对于进入每个 Block 的张量，要经过：

- 一个 RMSNorm 模块
- 一个多头注意力模块，并进行残差连接
 - 如果选择进行位置编码，则还需要一个 RoPE 模块
 - 注意力权重计算需要 Softmax 模块
- 一个 RMSNorm 模块
- 一个 FFN 模块，并进行残差连接
 - 需要包含 SiLU 激活函数模块

除此之外，几乎所有模块都需要调用线性变换模块！

PART2: 基本运算组件的代码实现



将要实现的模块：

- Token Embedding
- Linear
- RMSNorm
- SiLU
- Softmax

1.class Generate_Embeddings

IDEA:

- 最原始的输入：BPE 的编码结果（如 [3,10,2,6,4]，均为 token 的词表编号）
- 张量规模：[batch_size,seq_len]
- 期望的模型输入：词表中的不同词拥有不同的编码向量，而非单纯的编号。如：
 - 1 号词拥有一个 d_model 维张量
 - 2 号词拥有另一个 d_model 维张量
 -
- 实现思路：生成一个 vocab_size 行、d_model 列的矩阵。每列代表该编号词的初始编码向量。该矩阵随机初始化，代表经模型处理前对任何词的意义都没有先验认知。

1.class Generate_Embeddings

代码实现:

```
# vocab_size*embedding_dim矩阵中取样
class Generate_Embeddings(nn.Module):
    def __init__(self,number_embeddings:int,embedding_dim:int,device=None,dtype=None):
        super(Generate_Embeddings,self).__init__()
        self.embedding_matrix=torch.empty(number_embeddings,
                                           embedding_dim,
                                           device=device,
                                           dtype=torch.float32)
        nn.init.trunc_normal_(self.embedding_matrix,mean=0,std=0.02)
    def forward(self,token_ids:torch.Tensor)->torch.Tensor:
        return self.embedding_matrix[token_ids]
```

2.class Linear_Transform

IDEA:

- 假设需要将 3 维张量变成 6 维……
- 在数学上讲：令 3 维张量 $\times (1 \times 3)$ 右乘一个 3×6 矩阵 W
- xW 规模即为 1×6 !



Question: 线性运算是 LLM 中最高频的运算——能否将这种运算尽可能加速？

2.class Linear_Transform

Question: 线性运算是 LLM 中最高频的运算——能否将这种运算尽可能加速？

- Pytorch 的张量遵循“最后一维元素内存地址连续”的原则
- 对于一个 3×6 张量 W ，在内存空间上从前往后为：
 - 第一行 6 个元素
 - 第二行 6 个元素
 - 第三行 6 个元素
- 对于一个 $4 \times 3 \times 6$ 张量 (batch size, 行数, 列数)，也是每 6 个元素内存连续 (如: $[1,1,1]$ 和 $[1,2,1]$ 的内存地址相差 6 个单位)
- 即：矩阵同行元素内存连续，同列不连续 \Rightarrow 行主序原则

2.class Linear_Transform

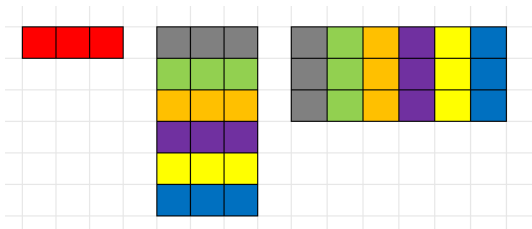


执行 6 次向量点积运算。每次运算中：

- x 整行参与，内存连续，可充分利用缓存
- W 整列参与，内存不连续，可能无法利用缓存

如何让 W 也能“整行”参与每次运算？

2.class Linear_Transform



- Pytorch 的转置操作不改变张量的内存空间 ……
- 新建 W : $[6,3]$ (每 3 元素内存连续)
- 转置 W 为 $[3,6]$: 可进行矩阵乘法, 且内存分布不变
- x 与 W 每次运算全部内存连续, 可充分利用缓存!

2.class Linear_Transform

重点：行主序规则

```
class Linear_Transform(nn.Module):  
    def __init__(self, in_features:int, out_features:int, device=None, dtype=None):  
        super(Linear_Transform, self).__init__()  
        self.linear_matrix=torch.empty(out_features,  
                                       in_features,  
                                       device=device,  
                                       dtype=torch.float32)  
        self.linear_matrix=self.linear_matrix.transpose(-2,-1)  
        nn.init.trunc_normal_(self.linear_matrix, mean=0, std=0.02)  
        self.linear_matrix=nn.Parameter(self.linear_matrix)  
  
    def forward(self, x:torch.Tensor)->torch.Tensor:  
        return torch.matmul(x, self.linear_matrix)
```


3.class RMSNorm

IDEA: 对输入张量 \vec{a} 归一化

- $a_i = \frac{a_i}{RMS(\vec{a})} g_i$ (统一除以归一化权重, 并进行可学习微调)
- g_i 为可学习参数
- $RMS(\vec{a}) = \sqrt{(\frac{1}{d_{model}} \sum a_i^2) + \epsilon}$

输入张量的规模: $[batch_size, seq_len, d_model] \Rightarrow$ 非一维向量, 如何处理?

Pytorch 的广播机制: 默认最后几维进行运算, 前面的维度全部都为复制操作

\Rightarrow 相当于对 $batch_size * seq_len$ 个 d_model 维张量做并行操作。

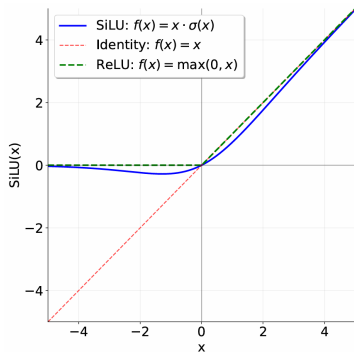
3.class RMSNorm

```
# 运算法则：归一化*可学习缩放倍数
class RMSNorm(nn.Module):
    def __init__(self, d_model: int, eps: float = 1e-5, device = None, dtype = None):
        super(RMSNorm, self).__init__()
        self.eps = eps
        self.g = nn.Parameter(torch.ones(d_model, device=device, dtype=torch.float32))

    def _get_rms(self, x: torch.Tensor) -> torch.Tensor:
        sum_square = torch.sum(x**2, dim=-1, keepdim=True)
        mean_square = sum_square / x.shape[-1]
        return torch.sqrt(mean_square + self.eps)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        ori_dtype = x.dtype
        x = x.to(torch.float32) # 张量元素类型转换
        rms = self._get_rms(x)
        x_normed = x / rms
        x_normed = x_normed.to(ori_dtype)
        return x_normed * self.g
```

4.class SiLU_Activation



具体作用:

- 近似实现小于 0 的 x 值归零, 大于 0 的值保持原大小不变
- 相比 ReLU 函数, 在零点处光滑可导

需预先实现:

- Sigmoid 激活函数
- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$

4.class SiLU_Activation

```
class Sigmoid_Activation(nn.Module):
    def __init__(self):
        super(Sigmoid_Activation,self).__init__()

    def forward(self,x:torch.Tensor)->torch.Tensor:
        denominator=1+torch.exp(-x)
        return 1/denominator

class SiLU_Activation(nn.Module):
    def __init__(self):
        super(SiLU_Activation,self).__init__()
        self.sigmoid_activator=Sigmoid_Activation()

    def forward(self,x:torch.Tensor)->torch.Tensor:
        sigmoid_x=self.sigmoid_activator(x)
        return x*sigmoid_x
```

5.class Softmax_Activation

$$x_i = \frac{e^{x_i}}{\sum e^{x_i}}$$

- 每个 x_i 计算指数作为权重，再进行权重归一化
- 能够使相对较大值的优势更明显
- 即使较小的值，归一化后仍不为 0

Problem: 如果存在 x_i 非常大? (如: 给 [20,3,1005] 做归一化)

- $e^{1000} = \text{NAN!}$

5.class Softmax_Activation

- [100,101,102] 归一化 vs [-2,-1,0] 归一化
- 102 的权重: $\frac{e^{102}}{e^{102}+e^{101}+e^{100}} = \frac{e^0}{e^0+e^{-1}+e^{-2}}$
- [100,101,102] 的 Softmax 归一化结果与 [-2,-1,0] 等同

令 x_{max} 为 x_i 中的最大值:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_i}}$$

$$= \frac{e^{x_i} / e^{x_{max}}}{\sum e^{x_i} / e^{x_{max}}}$$

$$= \frac{e^{x_i - x_{max}}}{\sum e^{x_i - x_{max}}}$$

即: 令所有 x_i 减去 x_{max} , 即可避免过大值无法计算的问题!

5.class Softmax_Activation

```
class Softmax_Activation(nn.Module):
    def __init__(self,dim:int=-1):
        super(Softmax_Activation,self).__init__()
        self.dim=dim

    def forward(self,x:torch.Tensor)->torch.Tensor:
        #shape of x:(bsz,seq_len,d_k)
        x_max=torch.max(x,dim=self.dim,keepdim=True).values
        x_exp=torch.exp(x-x_max)
        x_exp_sum=torch.sum(x_exp,dim=self.dim,keepdim=True)
        return x_exp/x_exp_sum
```

`x_max:[bsz,seq_len]`, `x-x_max` 又出现了另一种广播机制：复制操作

PART3: RoPE 变换的代码实现

RoPE 运算的意义

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix}$$

$$R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

$$\text{其中 } \theta_{i,k} = \frac{i}{\Theta^{2k/d}}$$

R 矩阵的性质: $R^m R^n = R^{m+n}$

在注意力机制中, i 位置的问题向量 q_i 和 j 位置的特征向量 k_j , 若存在距离间的相对差异, 应当具有相对位置的表征

$$\Rightarrow q'_i = R^i q_i, k'_j = R^j k_j$$

$\Rightarrow q'^T_i k'^T_j = q^T_i R^{iT} R^j k_j = q^T_i R^{i-j} k_j$, 相当于注意力权重值 $q^T_i k_j$ 加上了 $i-j$ 程度的旋转, 即: 旋转程度仅取决于 q 向量和 k 向量间的相对位置 (角度与 $i-j$ 成比例)。

如何避免暴力计算矩阵乘法？

观察：矩阵偶数行对应同样的变化法则： $[\cos, -\sin]$ ，角度 $\theta_{i,k}$ 作为自变量

⇒ 如果这些共享相同法则的偶数行能被统一快捷计算, 后续可以怎么处理?

- 我们期望将 R_i 的各偶数行二维块计算数值，进行逐块点乘
- 三对逐块点乘结果分别对应了变换后 x 的第 0、2、4 各元素值
- 对于奇数行同理，计算数值后进行逐块点乘，可得 1、3、5 元素值

cos	-sin	cos	-sin	cos	-sin		
red	red						red
							red
		yellow	yellow				yellow
							yellow
				blue	blue		blue
							blue

如何计算 R_i 的值？

- 预先计算规模为 $(\text{max_seq_len}, d/2)$ 的 $\theta_{i,k}$ 取值表
- 以此计算相同规模的 $\cos(\theta_{i,k})$ 取值表、 $\sin(\theta_{i,k})$ 取值表
- 由 \cos 取值表直接取得 1、3、5 位置的值
- 由 \sin 取值表直接取得 2、4、6 位置的值
- 12、34、56 拼接成 3 块，与 \times 的三块逐块点乘，得到三个偶数行的值。
- 同理得到 \times 变换后所有奇数行的值，二者拼接得到完整的变换后 \times 向量

cos	-sin	cos	-sin	cos	-sin			
1	2							
		3	4					
				5	6			

	cos	-sin		cos	-sin		cos	-sin
	1	2		3	4		5	6

RoPE 变换的代码实现

```
class RoPE(nn.Module):
    def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None):
        super(RoPE, self).__init__()
        self.theta = theta
        self.d_k = d_k
        self.max_seq_len = max_seq_len
        self.device = device

        d_half = d_k // 2
        positions = torch.arange(max_seq_len, device=device).unsqueeze(1) #[max_seq_len, 1]
        dims = torch.arange(d_half, device=device).unsqueeze(0) #[1, d_half]
        angles = positions / (theta ** (2 * dims / d_k)) #[max_seq_len, d_half]

        cos_values = torch.cos(angles).unsqueeze(0)
        self.register_buffer("cos_values", cos_values) #(1, max_seq_len, d_half)
        sin_values = torch.sin(angles).unsqueeze(0)
        self.register_buffer("sin_values", sin_values) #(1, max_seq_len, d_half)
```

max_seq_len: i 的最大迭代值; angles: $\theta_{i,k}$ 的取值

positions: i 的取值序列; dims: k 的取值序列; angles 的除法是逐元素相除

RoPE 变换的代码实现

```
def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor:
    x_splited = x.reshape(*x.shape[:-1], self.d_k // 2, 2)
    cos_chunk = self.cos_values[:, token_positions, :]
    sin_chunk = self.sin_values[:, token_positions, :]

    even_transform = torch.stack([cos_chunk, -sin_chunk], dim=-1)
    odd_transform = torch.stack([sin_chunk, cos_chunk], dim=-1)

    x_rotated_odd = torch.sum(x_splited * even_transform, dim=-1) # (bsz, seq_len, d_k // 2)
    x_rotated_even = torch.sum(x_splited * odd_transform, dim=-1) # (bsz, seq_len, d_k // 2)
    stacked_x = torch.stack([x_rotated_odd, x_rotated_even], dim=-1)
    x_rotated = stacked_x.reshape(*stacked_x.shape[:-2], self.d_k)

    return x_rotated
```

cos_chunk: 每个 i 位置对应 d_half 维的取值表 (1, 3, 5 位)

1.class Feed_Forward_Network

```
class Feed_Forward_Network(nn.Module):
    def __init__(self,d_model:int,d_ff=None,device=None,dtype=None):
        super(Feed_Forward_Network,self).__init__()
        self.d_model=d_model
        if d_ff is not None:
            self.d_ff=d_ff
        else:
            self.d_ff=int(8/3*d_model)
        self.linear_w1=Linear_Transform(d_model,d_ff,device=device,dtype=dtype)
        self.linear_w3=Linear_Transform(d_model,d_ff,device=device,dtype=dtype)
        self.linear_w2=Linear_Transform(d_ff,d_model,device=device,dtype=dtype)
        self.activator=SiLU_Activation()
```

1.class Feed_Forward_Network

```
def forward(self,x:torch.Tensor)->torch.Tensor:
    enhanced=self.linear_w1(x)
    activated=self.activator(enhanced)
    gate=self.linear_w3(x)
    gated=activated*gate
    output=self.linear_w2(gated)
    return output
```

2.class Multihead_Attention

模块算法（不含残差连接）：

- 输入张量 $x[\text{batch_size}, \text{seq_len}, \text{d_model}]$
- 经过三种线性变换，相当于 x 的每一个 d_model 维词向量做线性变换得：
 - Q、K 矩阵 $[\text{batch_size}, \text{seq_len}, \text{num_heads} * \text{d_k}]$
 - V 矩阵 $[\text{batch_size}, \text{seq_len}, \text{num_heads} * \text{d_v}]$
- 对 Q、K 矩阵做 RoPE 位置编码
- 生成注意力上三角掩码（防止破坏“自回归”假设）
- 利用 QKV 矩阵、注意力掩码计算注意力输出

注意力输出的计算

```
class Scaled_dot_Product_Attention(nn.Module):
    def __init__(self):
        super(Scaled_dot_Product_Attention, self).__init__()

    def forward(self, Q: torch.Tensor, K: torch.Tensor, V: torch.Tensor, mask: torch.Tensor=None) -> torch.Tensor:
        d_k=Q.shape[-1]

        attn_score=torch.matmul(Q,K.transpose(-2,-1))/math.sqrt(d_k)#[bsz,*,Qseq_len,Kseq_len]
        if mask is not None:
            attn_score=attn_score.masked_fill(mask==0,float('-inf'))
        softmax=Softmax_Activation(dim=-1)
        attn_weight=softmax(attn_score)#[bsz,*,Qseq_len,Kseq_len]

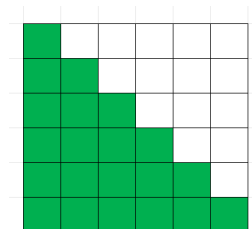
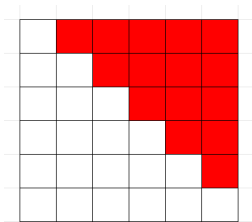
        #V:[bsz,*,Kseq_len,d_v]
        attn_output=torch.matmul(attn_weight,V)
        return attn_output#[bsz,*,Qseq_len,d_v]
```

注意力掩码的生成

```
class Causal_Mask:
    def __init__(self, seq_len, device=None):
        self.seq_len=seq_len
        self.device=device

    def generate(self)->torch.Tensor:
        ones=torch.ones(self.seq_len,self.seq_len,device=self.device)
        mask=torch.triu(ones,diagonal=1)
        mask=(mask==0)
        return mask
```

triu: 保留矩阵主对角线上方第 **diagonal** 条对角线及以上的元素



Multihead_Attention 所有的子模块梳理

- 注意力掩码生成模块
- 注意力输出计算模块
- RoPE 模块 \Rightarrow 需要额外具备 `max_seq_len`、`theta`、`token_positions` 等相关参数
- Q、K、V、O 四种线性变换

完整模块的组装

```
class Multihead_Attention(nn.Module):
    def __init__(self, d_model:int, num_heads:int, max_seq_length:int=None, theta:int=None, device=None):
        super(Multihead_Attention, self).__init__()

        self.d_model=d_model
        self.num_heads=num_heads
        self.d_k=d_model//num_heads
        self.d_v=d_model//num_heads

        self.max_seq_length=max_seq_length
        self.theta=theta
        self.token_positions=None

        self.q_proj=Linear_Transform(d_model,num_heads*self.d_k,device=device)
        self.k_proj=Linear_Transform(d_model,num_heads*self.d_k,device=device)
        self.v_proj=Linear_Transform(d_model,num_heads*self.d_v,device=device)
        self.o_proj=Linear_Transform(num_heads*self.d_v,d_model,device=device)

        self.sdpa=Scaled_dot_Product_Attention()

        if max_seq_length is not None and theta is not None:
            self.rope=RoPE(theta,self.d_k,max_seq_length,device=device)
        else:
            self.rope=None
```


完整模块的组装

```
def forward(self,x:torch.Tensor,token_positions:torch.Tensor=None)->torch.Tensor:
    bsz=x.shape[0]
    seq_len=x.shape[1]

    #qk:[bsz,n_heads,seq_len,d_k],v:[bsz,n_heads,seq_len,d_v]
    Q=self.q_proj(x)
    Q=Q.reshape(bsz,seq_len,self.num_heads,self.d_k).transpose(1,2)
    K=self.k_proj(x)
    K=K.reshape(bsz,seq_len,self.num_heads,self.d_k).transpose(1,2)
    V=self.v_proj(x)
    V=V.reshape(bsz,seq_len,self.num_heads,self.d_v).transpose(1,2)

    #apply RoPE on QK
    if self.rope is not None:
        self.token_positions=token_positions
        Q=self.rope(Q,self.token_positions)
        K=self.rope(K,self.token_positions)
```

完整模块的组装

```
mask=Causal_Mask(seq_len,device=x.device).generate()
mask=mask.unsqueeze(0).unsqueeze(1)

attn_output=self.sdpa(Q,K,V,mask)
attn_output=attn_output.transpose(1,2).reshape(bsz,seq_len,self.num_heads*self.d_v)
attn_output=self.o_proj(attn_output)

return attn_output
```

原始的 `attn_output: [bsz, num_heads, seq_len, d_v]`

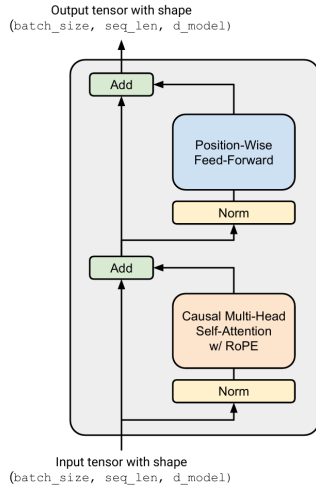
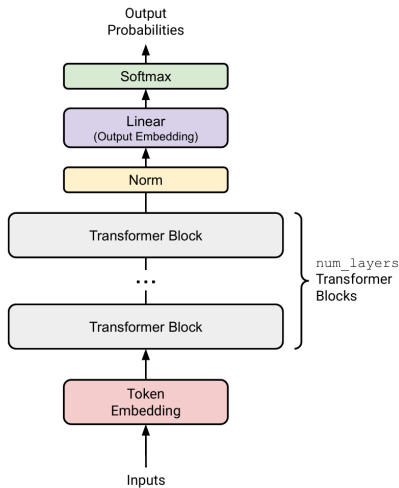
`[bsz, seq_len, num_heads, d_v] ⇒ [bsz, seq_len, num_heads*d_v]`

最终形状: `[bsz, seq_len, d_model]`

PART5:

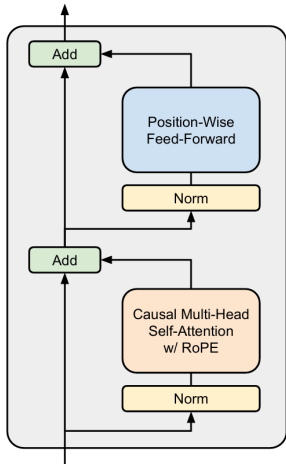
Transformer Block 与最终模型的组装

Transformer 总体架构图示



Transformer Block 结构的梳理

Output tensor with shape
(batch_size, seq_len, d_model)



Input tensor with shape
(batch_size, seq_len, d_model)

对于进入每个 Block 的张量，要经过：

- 一个 RMSNorm 模块
- Multihead Attention 模块
- 残差连接
- 另一个 RMSNorm 模块
- FFN 模块
- 残差连接

每个 Block 要接收的参数，就是以上所有模块需要的参数并集！

Transformer Block 的组装

```
class Transformer_Block(nn.Module):  
    def __init__(self,  
                  d_model:int,  
                  num_heads:int,  
                  d_ff:int,  
                  max_seq_length:int=None,  
                  theta:int=None,  
                  dtype=None,  
                  device=None):  
        super(Transformer_Block,self).__init__()  
        self.RMSNorm_Attn=RMSNorm(d_model,dtype=dtype,device=device)  
        self.RMSNorm_FF=RMSNorm(d_model,dtype=dtype,device=device)  
        self.Multihead_Attn=Multihead_Attention(d_model,num_heads,max_seq_length,theta,device=device)  
        self.Feed_Forward=Feed_Forward_Network(d_model,d_ff,device=device,dtype=dtype)
```

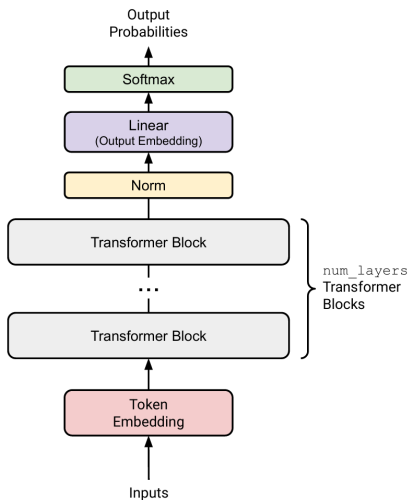
Transformer Block 的组装

```
def forward(self,x:torch.Tensor,token_positions:torch.Tensor)->torch.Tensor:
    residual_attn=x
    x_normed_attn=self.RMSNorm_Attn(x)
    attn_output=self.Multihead_Attn(x_normed_attn,token_positions)
    x=residual_attn+attn_output

    residual_ff=x
    x_normed_ff=self.RMSNorm_FF(x)
    ff_output=self.Feed_Forward(x_normed_ff)
    x=residual_ff+ff_output

    return x
```

Transformer 完整结构的梳理



对于初始输入张量，要经过：

- 一个 Token Embedding 模块，转化成模型可处理的张量格式
- 若干个 Transformer Block 模块，进行多层信息吸收
- 一个 RMSNorm 模块
- 最终的线性激发模块，计算词表中各词权重得分

完整 Transformer 的组装

```
class Transformer_LM(nn.Module):
    def __init__(self, d_model:int, num_heads:int, d_ff:int, vocab_size:int, num_layers:int,
                 max_seq_length:int=None, theta:int=None, dtype=None, device=None):
        super(Transformer_LM, self).__init__()
        self.d_model=d_model
        self.num_heads=num_heads
        self.d_ff=d_ff
        self.vocab_size=vocab_size
        self.num_layers=num_layers
        self.max_seq_length=max_seq_length
        self.theta=theta
        self.dtype=dtype
        self.device=device
        self.embeddings=Generate_Embeddings(vocab_size, d_model, device=device, dtype=dtype)
        self.transformer_blocks=nn.ModuleList([
            Transformer_Block(d_model=d_model, num_heads=num_heads, d_ff=d_ff,
                             max_seq_length=max_seq_length, theta=theta, dtype=dtype, device=device)
            for _ in range(num_layers)])
        self.final_norm=RMSNorm(d_model, device=device, dtype=dtype)
        self.final_layer=Linear_Transform(d_model, vocab_size, device=device, dtype=dtype)
```

完整 Transformer 的组装

```
def forward(self, token_ids: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor:
    x = self.embeddings(token_ids)
    for block in self.transformer_blocks:
        x = block(x, token_positions)
    x = self.final_norm(x)
    linear_score = self.final_layer(x)
    return linear_score
```

谢谢！