

# 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

2025.11.2

## 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

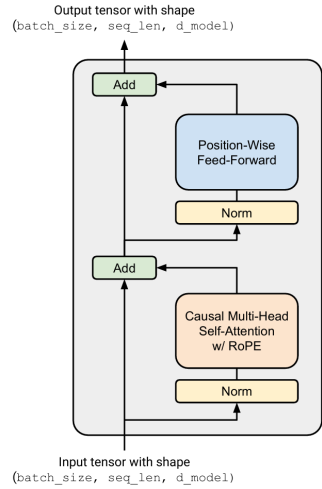
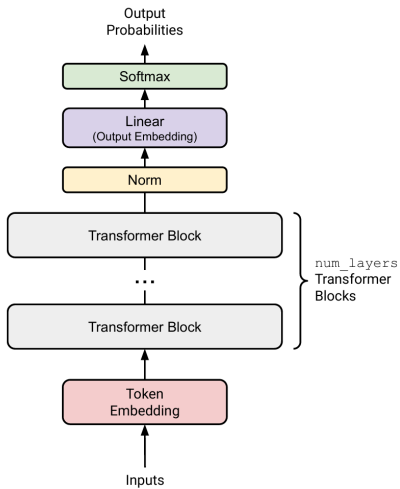
## FFN 与 Attention 的代码实现

## Transformer Block 与最终模型的组装

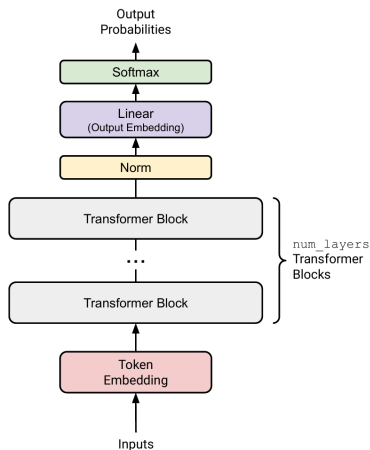
# PART1:

# 宏观系统架构

# Transformer 总体架构图示



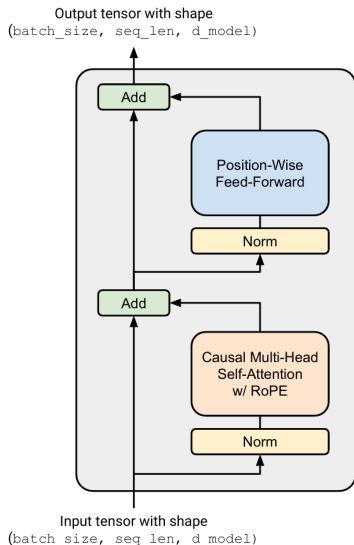
## 各模块结构的梳理



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

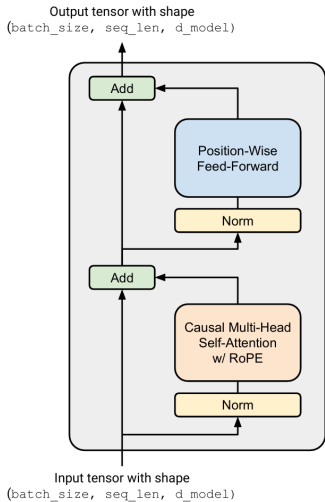
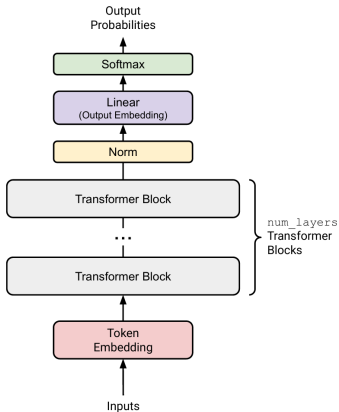


对于进入每个 Block 的张量，要经过：

- 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 5 维……
- 在数学上讲：令 3 维张量  $\times (1 \times 3)$  右乘一个  $3 \times 6$  矩阵  $W$
- $\times W$  规模即为  $1 \times 6!$

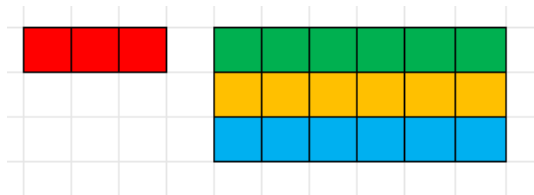
Question: 线性运算是 LLM 中最高频的运算——能否将这种运算尽可能加速？

## 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  $3 \times 6$  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
- 即：矩阵同行元素内存连续，同列不连续

## 2.class Linear\_Transform

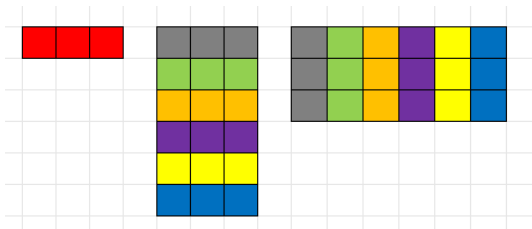


执行 6 次向量点积运算。每次运算中：

- $x$  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
- $RMS(\vec{a}) = \sqrt{(\frac{1}{d_{model}} \sum a_i^2) + \epsilon}$

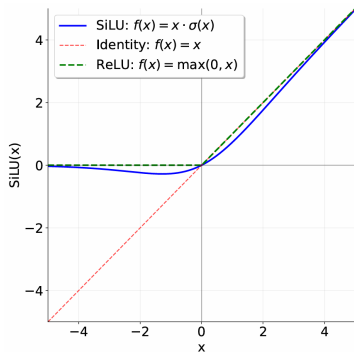
### 3.class RMSNorm

```
# 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?

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

## 5.class Softmax\_Activation

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
```

## PART3: RoPE 变换的代码实现



## TODO



# 1.class Feed\_Forward\_Network

模块算法（不含残差连接）：

- 输入张量  $x$  ( $d_{\text{model}}$  维)
- 路线 1：经  $W1$  升维变换为 ( $d_{\text{ff}}$ ) 维，经过 SiLU 激活函数得到新的  $x$
- 路线 2：经  $W3$  做另一个升维变换，得到  $\text{gated}(d_{\text{ff}}$  维)
- $x$  与  $\text{gated}$  逐元素相乘，得到新的  $x$
- 经  $W2$  降维回到 ( $d_{\text{model}}$ ) 维

## 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  $\text{d\_model}$  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

## 2.class Multihead\_Attention

注意力输出的计算方法：

- Q、K 矩阵相乘，计算 token 特征匹配度
- 匹配度矩阵做掩码处理
- $\text{Softmax}(\frac{QK^T}{\sqrt{d_k}})$ ，计算 token 之间注意力得分
- 将得分矩阵乘 V 矩阵，得到注意力输出
- 多头注意力输出合并

## 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]
```







## 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

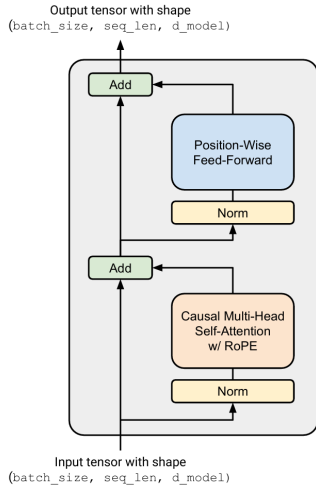
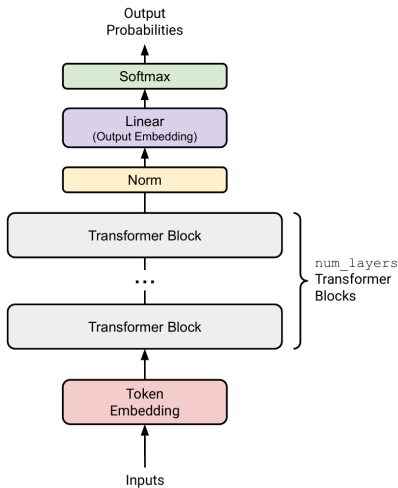
```
mask=Causal_Mask(seq_len,device=x.device).generate()
mask=mask.unsqueeze(0).unsqueeze(1)

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
```

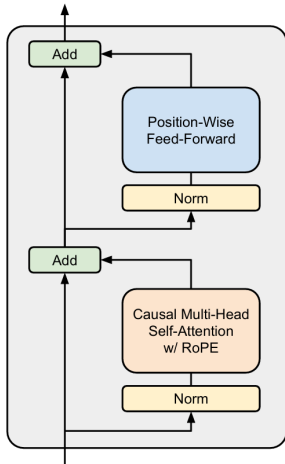


# Transformer 总体架构图示



## 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)
```

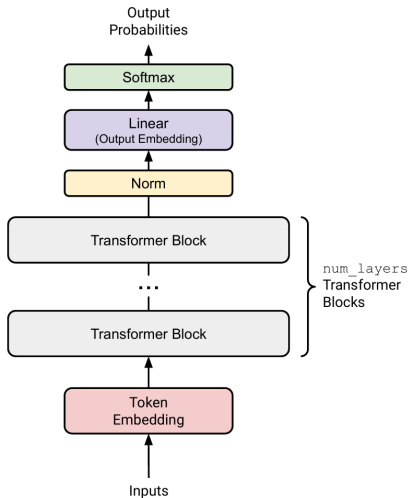
## Transformer Block 的组装

```
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
```