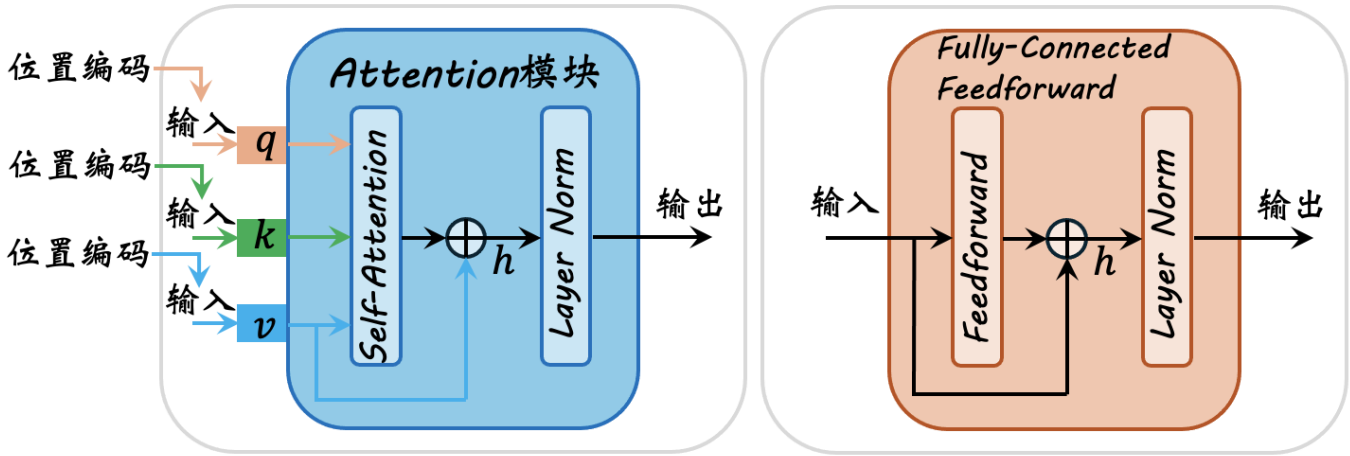
1 Transformer架构

Transformer 是一类基于注意力机制（Attention）的模块化神经网络。它通过对输入序列中的历史状态（history state）和当前状态（current state）进行加权组合来进行预测。Transformer作为语言模型时，接收词序列（word sequence）作为输入，利用上文语境（context）预测下一个词的概率分布（probability distribution）。本节将介绍 Transformer 的基本原理及其在语言建模中的应用。

Transformer 的结构图

Transformer的网络结构采用模块化设计,核心由 **注意力(Attention)** 和 **全连接前馈(Fully-connected Feedforward)** 两大模块组成。

如下图所示：



其中注意力模块集成了 **自注意力层(Self-Attention Layer)** 、 **残差连接(Residual Connections)** 和 **层正则化(Layer Normalization)**。

而全连接前馈模块则包含 **前馈层(Feedforward Layer)** 及配套的 **残差连接** 和**层正则化**。下图直观展示了这两个核心模块的结构。

此外，Transformer 还包含 **位置编码(Positional Encoding)** 模块，用于将词序信息编码到输入中。

下面我们将深入探讨各组成部分的工作原理。

1. 注意力层（Attention Layer）

注意力层通过加权平均机制将历史信息融入当前状态。其计算过程可以分为以下几个步骤:

首先，输入序列 $\{x_1, x_2, \ldots, x_t\}$ 的shape为 (t, d_{emb})，其中t为序列长度, d_{emb}为词向量维度。

Step 1. 线性变换生成查询(query)、键(key)和值(value)

通过三个线性变换矩阵得到query、key和value:

$$q_i = W_q x_i, \quad k_i = W_k x_i, \quad v_i = W_v x_i \tag{1}$$

其中 $W_q, W_k \in \mathbb{R}^{d_k \times d_{emb}}, W_v \in \mathbb{R}^{d_v \times d_{emb}}$ 为可学习参数矩阵。变换后得到:

- Query矩阵Q: (t, d_k)：Q的第i行表示位置i的query向量
- Key矩阵K: (t, d_k)：K的第i行表示位置i的key向量
- Value矩阵V: (t, d_v)：V的第i行表示位置i的value向量

Step 2. 计算注意力权重

Query和Key通过矩阵乘法 QK^T 计算相似度。这里的矩阵乘法实际上实现了所有Query向量与Key向量之间的点乘操作 - 对于位置i的Query向量 q_i 和位置j的Key向量 k_j，它们的点乘结果 $q_i \cdot k_j$ 就位于结果矩阵的第i行第j列。这个点乘结果衡量了位置i与位置j之间的相关性。

注意：得到的相似度矩阵会经过 $\sqrt{d_k}$ 缩放，再进行softmax归一化得到最终的注意力权重。之所以除以 $\sqrt{d_k}$ 是因为当向量维度 d_k 很大时，点积结果的方差也会变大，可能导致softmax函数梯度消失。

$$\alpha_{t,i} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \tag{2}$$

注意力权重矩阵shape为 (t, t)。

Step 3. 加权求和得到输出

将注意力权重与Value相乘并求和：

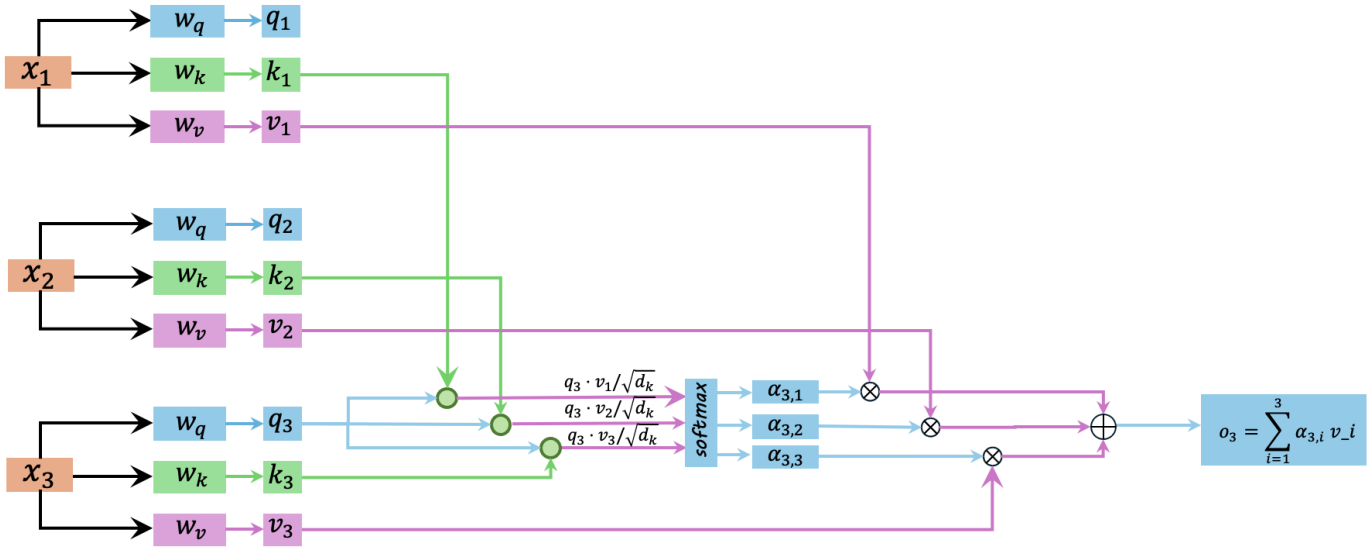
$$\text{Attention}(Q, K, V) = \alpha V \tag{3}$$

最终输出shape为 (t, d_v)。

整个过程可以写成一个公式:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4}$$

为了更直观地理解自注意力机制的计算过程，我们以一个包含3个元素的输入序列 $\{x_1, x_2, x_3\}$ 为例，下图展示了完整的Transformer自注意力计算第3个元素的流程。



我们可以看到，第一个矩阵乘法QK^T实现的，其实是所有Query向量与Key向量之间的点乘操作。而第二个矩阵乘法$\text{softmax}(QK^T/\sqrt{d_k})V$，则是将注意力权重矩阵与Value矩阵相乘，得到加权求和后的输出。

2. 全连接前馈层（Fully－connected Feedforwad Layer, FFN）

FFN是Transformer中参数量最大的组件，约占总参数量的2/3。FFN通常被认为是增加模型的非线性变换能力，通过高维映射提供更大的特征表达空间。但进一步分析，可以发现FFN其实和注意层一样，起到了Key-Value形式的记忆存储管理功能。

FFN每次仅处理一个时间步的输入，FFN的输入shape是 (t, d_v)，输出shape也是 (t, d_v)。

FFN由两个线性变换层组成，中间使用ReLU激活函数。对于输入向量 $v \in \mathbb{R}^{d_v}$，其计算过程可表示为：

$$FFN(v) = \max(0, W_1 v + b_1) W_2 + b_2 \tag{5}$$

这里第一层的权重矩阵 $W_1 \in \mathbb{R}^{d_{ff} \times d_v}$ 将输入从 d_v 维映射到更高维的 d_{ff}，偏置向量 $b_1 \in \mathbb{R}^{d_{ff}}$。第二层的权重矩阵 $W_2 \in \mathbb{R}^{d_v \times d_{ff}}$ 再将特征映射回 d_v 维，偏置向量 $b_2 \in \mathbb{R}^{d_v}$。

从功能上看，第一层扮演着神经记忆中key的角色，将输入映射到高维空间，并通过ReLU激活函数筛选正值特征，而第二层则对应value的功能，根据激活的特征模式，将输入映射回原始维度。

所以，FFN的功能，实际上和注意力层是互补的，注意力机制是显式的key-value查询，可以在不同时间步之间建立关联，而FFN是隐式的key-value存储，针对单个时间步进行特征转换。注意力负责序列间的交互，FFN负责单个时间步内的特征转换。

3. 层正则化（Layer Normalization, LayerNorm）

LayerNorm通过对输入向量 $v = \{v_i\}_{i=1}^n$ 的每个维度进行标准化操作，能够加速神经网络的训练并提升其泛化性能。在Transformer中，LayerNorm作用于每个时间步的特征维度(channel)上，只对一个时间步的特征进行标准化。

Transformer选择使用LayerNorm而非BatchNorm有其深层原因。在NLP任务中，输入序列长度不固定，BatchNorm在batch维度上计算统计量会导致不同长度序列的统计特征不一致，而LayerNorm仅在特征维度归一化则不受此影响。此外，由于Transformer模型规模较大，训练时batch size往往较小，这使得BatchNorm的统计量估计不够准确，而LayerNorm不依赖batch size。同时，Transformer中同一序列不同位置的token往往具有不同的统计特性，BatchNorm会混合这些特性，而LayerNorm通过独立标准化每个位置的特征能更好地保持这种差异性。在推理阶段，BatchNorm需要使用训练时估计的全局统计量，可能与实际输入存在偏差，而LayerNorm直接使用当前输入计算统计量，推理更加稳定。

LayerNorm输入shape为(t, d_v)，输出shape也为(t, d_v)。其核心公式为：

$$LN(v_i) = \frac{\alpha(v_i - \mu)}{\delta} + \beta \tag{6}$$

这里的 α 和 β 是可学习的缩放和偏移参数，而 μ 和 δ 则分别表示输入向量的均值和标准差：

$$\mu = \frac{1}{n}\sum_{i=1}^{n} v_i, \quad \delta = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2}. \tag{7}$$

对于每个时间步t，LayerNorm独立地对其d_v维特征向量进行标准化，使得每个时间步的特征分布都趋于均值0、方差1的标准正态分布。

4. 残差连接（Residual Connections）

残差连接是解决深度神经网络中梯度消失问题的有效方法。在Transformer的编码模块中包含两个残差连接:一个位于自注意力层,另一个位于FFN层。具体来说,自注意力层的残差连接将输入直接加到自注意力层的输出上,然后经过层正则化;FFN层的残差连接则将输入加到FFN的输出上,再经过层正则化。

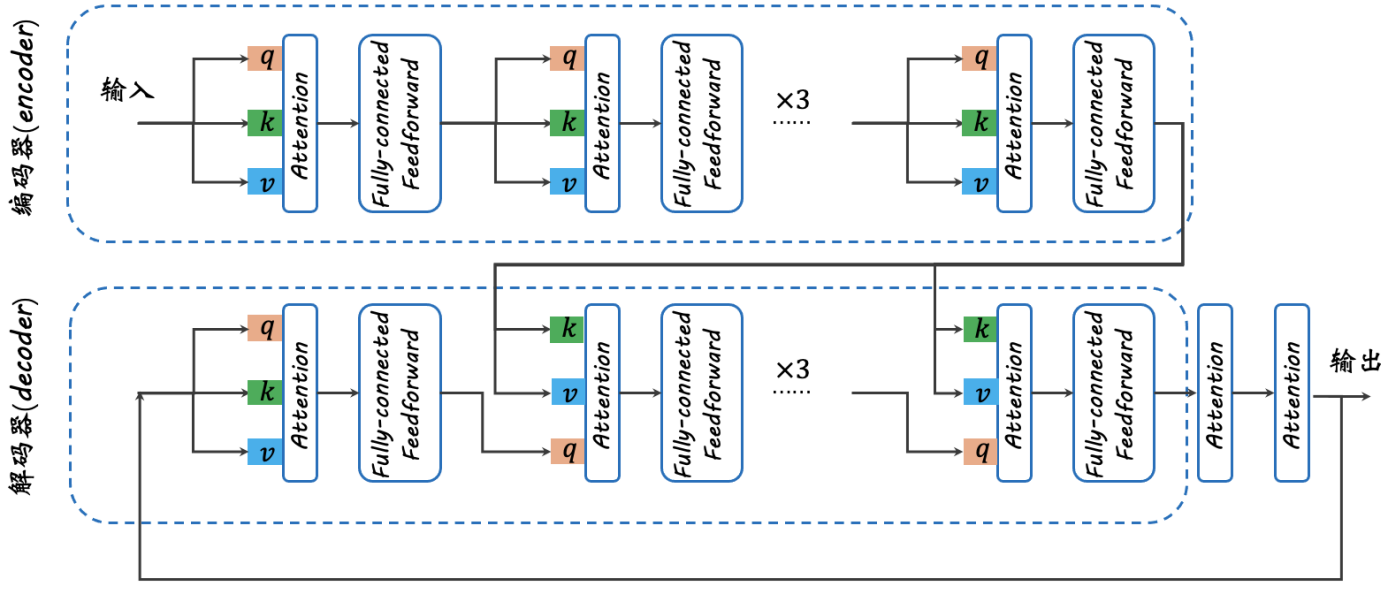
根据层正则化的位置,Transformer有两种主要变体:Post-LN和Pre-LN。Post-LN将层正则化放在残差连接之后,而Pre-LN则将层正则化放在残差连接之前。这两种结构各有优劣:Post-LN在防止表征坍塌(Representation Collapse)方面表现更好,但在处理梯度消失问题上略显不足;Pre-LN则相反,更擅长处理梯度消失,但在防止表征坍塌方面稍弱。

> 表征坍塌指深度网络中不同输入经多层变换后趋于相似的现象。主要表现为:
>
> - 深层网络中,不同输入可能映射到相似特征空间,丢失独特性
> - Post-LN通过在残差后归一化,可以更好保持特征差异:
>   - 基于当前输入计算统计量
>   - 同时考虑残差和主路径分布
>   - 保持不同输入的统计特征
>
> Pre-LN虽训练更稳定,但因在残差前归一化,容易导致特征趋同。

原始Transformer采用Encoder-Decoder架构,由编码器(Encoder)和解码器(Decoder)两大部分组成。编码器包含6个相同的encoder layer,每个layer由一个自注意力模块(输入的query、key、value相同)和一个FFN模块构成。解码器同样包含6个decoder layer,但每个layer包含三个模块:自注意力模块、交叉注意力模块和FFN模块。其中:

- 自注意力模块:第一层接收模型输出作为输入,后续层使用前一层的输出
- 交叉注意力模块(Cross-Attention):每一层的交叉注意力的query都适用当前层的自注意力输出,而key和value来自编码器最后一层的输出，因为编码器最后一层的输出包含了最丰富的特征表示。
- FFN模块:对每个位置独立进行特征转换

下图展示了Transformer的编码器和解码器结构。



Transformer架构的编码器和解码器部分都可以独立使用,分别形成Encoder-Only和Decoder-Only模型。这两种变体架构的详细介绍将后面讲解。

位置编码（Positional Encoding）

最后，Transformer 还包含 **位置编码(Positional Encoding)** 模块，用于将词序信息编码到输入中。

由于自注意力机制本身不包含位置信息，为了让模型能够利用序列的顺序信息，Transformer在输入嵌入向量中加入了位置编码。位置编码直接与输入嵌入相加：

$$Input = \text{Word Embedding} + \text{Positional Encoding}$$

原始Transformer使用正弦和余弦函数的组合来生成位置编码：

$$PE_{(\text{pos},2i)} = sin(\text{pos}/10000^{2i/\text{d}_{emb}})$$
$$PE_{(\text{pos},2i+1)} = cos(\text{pos}/10000^{2i/\text{d}_{emb}})$$

其中：

- pos 是词在序列中的位置
- i 是维度的索引
- d_{emb} 是模型的维度

这种正弦位置编码虽然简单直观，但存在一些局限性，例如位置编码与词向量是简单相加的关系，这种方式可能会破坏原始词向量中的语义信息，对于超长序列，正弦函数的周期性可能导致位置编码在远距离上的区分度不足，在推理时需要预先计算好固定长度的位置编码表，不够灵活。

因此，后来的研究提出了许多改进的位置编码方案，如RoPE(Rotary Position Embedding)、ALiBi(Attention with Linear Biases)等。这些新的位置编码方案在处理长序列、计算效率等方面都有明显优势，我们将在后面的章节详细介绍。

多头注意力机制（Multi-Head Attention, MHA）

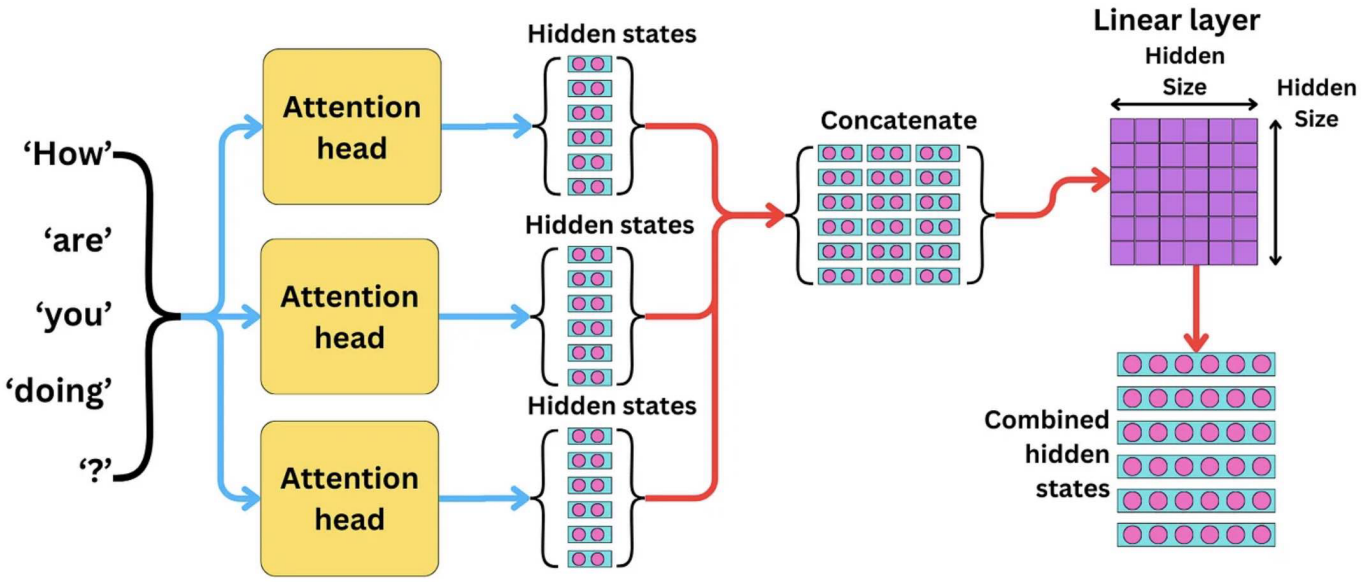
在原始Transformer中，自注意力层使用Multi-Head Attention，即多个独立的注意力头，每个注意力头使用不同的参数矩阵，从而捕捉序列中不同的语义信息。

那MHA到底是什么呢? 简单来说,它就是把多个单头注意力机制组合在一起。这种设计让模型可以同时从多个角度理解输入。每个注意力头都有独立的参数矩阵,可以学习不同的特征,从而增强模型的表达能力。

原始Transformer论文中提出了两种模型配置:

- base模型: 采用8头注意力机制,每个注意力头的维度为64,总维度为512 (8×64=512)
- big模型: 采用16头注意力机制,每个注意力头维度同样为64,总维度为1024 (16×64=1024)

为了控制计算复杂度,每个注意力头的query和key维度 d_k 以及value维度 d_v 都设置为 d_{emb}/h,其中 d_{emb} 是模型的嵌入维度,h是注意力头的数量。这种设计使得多头注意力的计算量与单头注意力相当,同时又能获得多个视角的特征表示。



我们用一个具体的例子来理解:假设输入一个序列 $\boldsymbol{X}=[x_1,x_2,\cdots,x_l]$,每个 x_i 都是一个 d 维向量,MHA的计算过程可以分为以下几步:

对于第s个注意力头,首先构建可学习的参数矩阵:

$$\begin{aligned}\boldsymbol{W}_q^{(s)}&\in\mathbb{R}^{d\times d_k}\\ \boldsymbol{W}_k^{(s)}&\in\mathbb{R}^{d\times d_k}\\ \boldsymbol{W}_v^{(s)}&\in\mathbb{R}^{d\times d_v}\end{aligned}\tag{8}$$

然后利用这些参数矩阵计算查询(Query)、键(Key)、值(Value)向量:

$$\begin{aligned}\boldsymbol{q}_i^{(s)}&=\boldsymbol{x}_i\boldsymbol{W}_q^{(s)}\in\mathbb{R}^{d_k}\\ \boldsymbol{k}_i^{(s)}&=\boldsymbol{x}_i\boldsymbol{W}_k^{(s)}\in\mathbb{R}^{d_k}\\ \boldsymbol{v}_i^{(s)}&=\boldsymbol{x}_i\boldsymbol{W}_v^{(s)}\in\mathbb{R}^{d_v}\end{aligned}\tag{9}$$

其中 $\boldsymbol{q}_i^{(s)}$、$\boldsymbol{k}_i^{(s)}$、$\boldsymbol{v}_i^{(s)}$ 分别是第i个位置第s个注意力头的查询向量、键向量和值向量。

示例代码

这里的代码并不是非常严谨的遵从了原始论文中的描述，而是为了方便理解，做了一些简化和改动，具体的改动写在了代码的注释中。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
from typing import Dict, Optional


def get_sinusoid_encoding_table(n_position, d_model):
    """生成正弦位置编码表

    Args:
```

```
        n_position: 最大序列长度
        d_model: 模型维度

    Returns:
        position_enc: [n_position, d_model] 的位置编码表
    """
    def cal_angle(position, hid_idx):
        return position / np.power(10000, 2 * (hid_idx // 2) / d_model)

    def get_posi_angle_vec(position):
        return [cal_angle(position, hid_j) for hid_j in range(d_model)]

    sinusoid_table = np.array([get_posi_angle_vec(pos_i) for pos_i in
range(n_position)])
    sinusoid_table[:, 0::2] = np.sin(sinusoid_table[:, 0::2])  # dim 2i
    sinusoid_table[:, 1::2] = np.cos(sinusoid_table[:, 1::2])  # dim 2i+1

    return torch.FloatTensor(sinusoid_table)


class MultiHeadAttention(nn.Module):
    def __init__(self, args: Dict):
        super().__init__()

        # 设置头的数量和每个头的维度
        self.n_heads = args.n_heads
        self.head_dim = args.dim // args.n_heads

        # 定义QKV变换矩阵
        self.query_proj = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        self.key_proj = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        self.value_proj = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        self.output_proj = nn.Linear(args.n_heads * self.head_dim, args.dim,
bias=False)

        # Dropout层
        # Attention Dropout
        self.attn_dropout = nn.Dropout(args.dropout)
        # Residual Dropout
        self.resid_dropout = nn.Dropout(args.dropout)
        # Dropout
        self.dropout = args.dropout

        # KV缓存
        self.key_cache, self.value_cache = None, None

        # 注意力掩码 - 用于确保当前token只能看到之前的token

        # 创建一个形状为(1,1,max_seq_len,max_seq_len)的掩码矩阵
```

```
        # 使用float("-inf")填充,这样在softmax后会变成0,实现掩码效果
        # 这个掩码用于确保每个token只能看到它之前的token,不能看到未来的token
        # 形状解释:
        # - 第一维=1: batch维度广播
        # - 第二维=1: 注意力头维度广播
        # - 第三维=max_seq_len: query序列长度
        # - 第四维=max_seq_len: key序列长度
        max_len = args.max_seq_len
        attn_mask = torch.full((1, 1, max_len, max_len), float("-inf"))

        # torch.triu(xxx, diagonal=1) 上三角矩阵, diagonal=1表示从对角线开始,屏蔽对角线及其右边的元素
        # register_buffer("name", tensor, persistent=False) - 注册一个不需要梯度的张量作为模块的缓冲区
        # - 与普通的Parameter不同,buffer不会被优化器更新
        # - 但会被保存到模型的state_dict中,在加载模型时可以恢复
        # - 使用persistent=False表示这个buffer在保存模型时不会被保存
        # - 因为掩码可以在加载模型时重新生成
        # 原始论文中的编码器不需要使用掩码矩阵来屏蔽未来的信息，因为编码器处理的是整个输入序列，每个位置的token可以自由地访问序列中的其他位置。
        # 这里为了方便，就统一使用掩码矩阵来屏蔽未来的信息
        self.register_buffer("attn_mask", torch.triu(attn_mask, diagonal=1), persistent=False)

    def forward(self, x: torch.Tensor, encoder_output: Optional[torch.Tensor] = None):
        """前向传播
        Args:
            x: 输入张量 [batch_size, seq_len, dim]
            encoder_output: 编码器输出，用于交叉注意力。
                            如果为None则为自注意力模式
        """
        batch_size, seq_len, _ = x.shape

        # 生成query向量 - 始终使用输入x
        query = self.query_proj(x).view(batch_size, seq_len, self.n_heads, self.head_dim)

        if encoder_output is None:
            # 自注意力模式：key和value来自同一输入x
            key = self.key_proj(x).view(batch_size, seq_len, self.n_heads, self.head_dim)
            value = self.value_proj(x).view(batch_size, seq_len, self.n_heads, self.head_dim)
        else:
            # 交叉注意力模式：key和value来自encoder的输出
            key = self.key_proj(encoder_output).view(batch_size, -1, self.n_heads, self.head_dim)
            value = self.value_proj(encoder_output).view(batch_size, -1, self.n_heads, self.head_dim)
```

```
        # 维度转置 [batch_size, n_heads, seq_len, head_dim]
        query = query.transpose(1, 2)
        key = key.transpose(1, 2)
        value = value.transpose(1, 2)

        # 注意力计算
        scale = 1.0 / math.sqrt(self.head_dim)
        attn_scores = torch.matmul(query, key.transpose(2, 3)) * scale

        # 只在自注意力模式下使用注意力掩码
        if encoder_output is None:
            attn_scores = attn_scores + self.attn_mask[:, :, :seq_len, :seq_len]

        attn_probs = F.softmax(attn_scores.float(), dim=-1).type_as(query)
        attn_probs = self.attn_dropout(attn_probs)

        output = torch.matmul(attn_probs, value)
        output = (
            output.transpose(1, 2)
            .contiguous()
            .view(batch_size, seq_len, -1)
        )

        return self.resid_dropout(self.output_proj(output))


class FeedForward(nn.Module):
    """FFN实现"""
    def __init__(self, dim: int, hidden_dim: int, dropout: float):
        super().__init__()

        # 定义两个线性变换层,w1,w2
        self.fc1 = nn.Linear(dim, hidden_dim, bias=True)
        self.fc2 = nn.Linear(hidden_dim, dim, bias=True)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # 第一层线性变换后接ReLU激活函数
        x = F.relu(self.fc1(x))
        # 第二层线性变换后接dropout
        x = self.dropout(self.fc2(x))
        return x


class TransformerEncoderBlock(nn.Module):
    """Transformer编码器块"""
    def __init__(self, args: Dict):
        super().__init__()
```

```
        # 自注意力层
        self.self_attention = MultiHeadAttention(args)
        # 前馈网络层
        self.feed_forward = FeedForward(
            dim=args.dim,
            hidden_dim=4 * args.dim,
            dropout=args.dropout
        )

        self.attention_norm = nn.LayerNorm(args.dim)
        self.ffn_norm = nn.LayerNorm(args.dim)

    def forward(self, x: torch.Tensor):

        # 原文的Pre-LN结构LayerNorm在残差连接之前
        h = x + self.self_attention(self.attention_norm(x))
        out = h + self.feed_forward(self.ffn_norm(h))

        # # Post-LN结构：先残差连接，再LayerNorm
        # h = self.attention_norm(x + self.self_attention(x))
        # out = self.ffn_norm(h + self.feed_forward(h))
        return out


class TransformerEncoder(nn.Module):
    """Transformer编码器"""
    def __init__(self, args: Dict):
        super().__init__()

        # 词嵌入层
        self.token_embeddings = nn.Embedding(args.vocab_size, args.dim)

        # 位置编码
        self.register_buffer(
            'pos_embedding',
            get_sinusoid_encoding_table(args.max_seq_len, args.dim)
        )

        # 编码器层
        self.layers = nn.ModuleList([
            TransformerEncoderBlock(args) for _ in range(args.n_layers)
        ])

        # 最后的层归一化
        self.final_norm = nn.LayerNorm(args.dim)

        # Dropout
        self.dropout = nn.Dropout(args.dropout)
```

```python
    def forward(self, tokens: torch.Tensor):
        # 词嵌入 + 位置编码
        h = self.dropout(
            self.token_embeddings(tokens) + self.pos_embedding[:tokens.size(1), :]
        )

        # 多层编码器块
        for layer in self.layers:
            h = layer(h)

        return self.final_norm(h)


class Transformer(nn.Module):
    """完整的Transformer模型"""
    def __init__(self, args: Dict):
        super().__init__()

        # 编码器
        self.encoder = TransformerEncoder(args)
        # 解码器
        self.decoder = TransformerDecoder(args)

        # 输出层
        self.output_proj = nn.Linear(args.dim, args.vocab_size, bias=False)

        # 初始化权重
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding)):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if isinstance(module, nn.Linear) and module.bias is not None:
                torch.nn.init.zeros_(module.bias)

    def forward(self, src_tokens: torch.Tensor, tgt_tokens: torch.Tensor):
        """
        Args:
            src_tokens: 源序列 [batch_size, src_len]
            tgt_tokens: 目标序列 [batch_size, tgt_len]
        """
        # 编码器前向传播
        encoder_out = self.encoder(src_tokens)

        # 解码器前向传播
        decoder_out = self.decoder(tgt_tokens, encoder_out)

        # 输出层
```

```
        logits = self.output_proj(decoder_out)

        return logits


class TransformerDecoder(nn.Module):
    """Transformer解码器"""
    def __init__(self, args: Dict):
        super().__init__()

        # 词嵌入层
        self.token_embeddings = nn.Embedding(args.vocab_size, args.dim)

        # 位置编码
        self.register_buffer(
            'pos_embedding',
            get_sinusoid_encoding_table(args.max_seq_len, args.dim)
        )

        # 解码器层
        self.layers = nn.ModuleList([
            TransformerDecoderBlock(args) for _ in range(args.n_layers)
        ])

        # 最后的层归一化
        self.final_norm = nn.LayerNorm(args.dim)

        # Dropout
        self.dropout = nn.Dropout(args.dropout)

    def forward(self, tokens: torch.Tensor, encoder_out: torch.Tensor):
        # 词嵌入 + 位置编码
        h = self.dropout(
            self.token_embeddings(tokens) + self.pos_embedding[:tokens.size(1), :]
        )

        # 多层解码器块
        for layer in self.layers:
            h = layer(h, encoder_out)

        return self.final_norm(h)


class TransformerDecoderBlock(nn.Module):
    """Transformer解码器块 - 使用Post-LN结构"""
    def __init__(self, args: Dict):
        super().__init__()

        # 自注意力层
```

```
        self.self_attention = MultiHeadAttention(args)
        # 交叉注意力层
        self.cross_attention = MultiHeadAttention(args)
        # 前馈网络层
        self.feed_forward = FeedForward(
            dim=args.dim,
            hidden_dim=4 * args.dim,
            dropout=args.dropout
        )

        # Post-LN: LayerNorm在残差连接之后
        self.self_attention_norm = nn.LayerNorm(args.dim)
        self.cross_attention_norm = nn.LayerNorm(args.dim)
        self.ffn_norm = nn.LayerNorm(args.dim)

    def forward(self, x: torch.Tensor, encoder_out: torch.Tensor):
        # 1. 自注意力层 (带掩码)
        h = self.self_attention_norm(x + self.self_attention(x))

        # 2. 交叉注意力层
        # query来自解码器，key和value来自编码器输出
        h = self.cross_attention_norm(h + self.cross_attention(h, encoder_out))

        # 3. 前馈网络层
        out = self.ffn_norm(h + self.feed_forward(h))
        return out


# Pre-LN版本的编码器块示例
class TransformerEncoderBlockPreLN(nn.Module):
    """Transformer编码器块 - 使用Pre-LN结构"""
    def __init__(self, args: Dict):
        super().__init__()

        self.self_attention = MultiHeadAttention(args)
        self.feed_forward = FeedForward(
            dim=args.dim,
            hidden_dim=4 * args.dim,
            dropout=args.dropout
        )

        # Pre-LN: LayerNorm在残差连接之前
        self.attention_norm = nn.LayerNorm(args.dim)
        self.ffn_norm = nn.LayerNorm(args.dim)

    def forward(self, x: torch.Tensor):
        # Pre-LN结构：先LayerNorm，再残差连接
        h = x + self.self_attention(self.attention_norm(x))
        out = h + self.feed_forward(self.ffn_norm(h))
```

```
        return out
```

总结

Transformer 是一种基于注意力机制的模块化神经网络，通过自注意力机制和全连接前馈层实现序列建模。它通过位置编码将词序信息融入输入，通过残差连接和层正则化加速训练过程。Transformer 的编码器和解码器结构可以独立使用，分别形成Encoder-Only和Decoder-Only模型。

后面，我们将会介绍Transformer会用在哪些场景，以及如何使用。