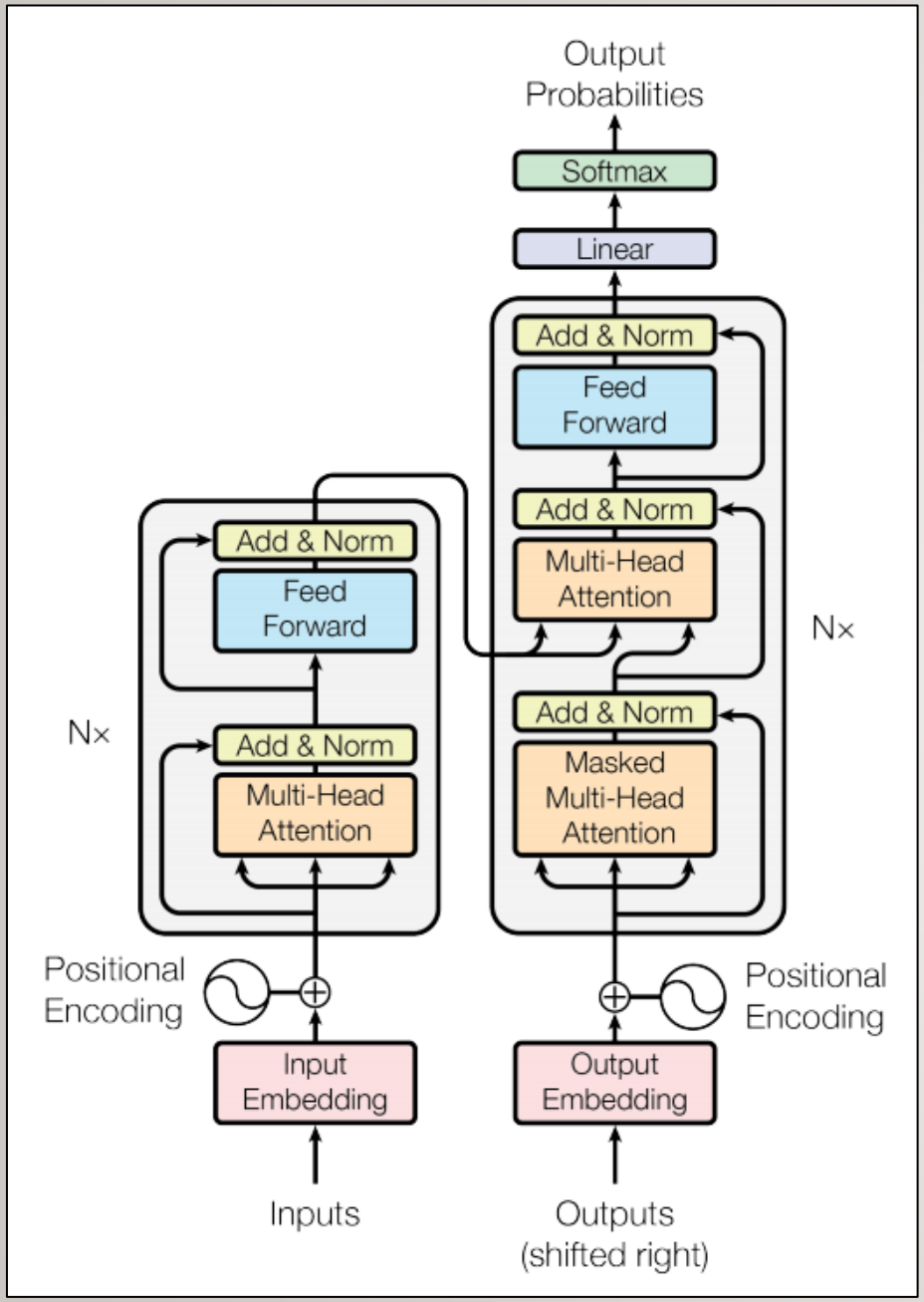
真-极度易懂 TRANSFORMER介绍

作者: 骰子AI

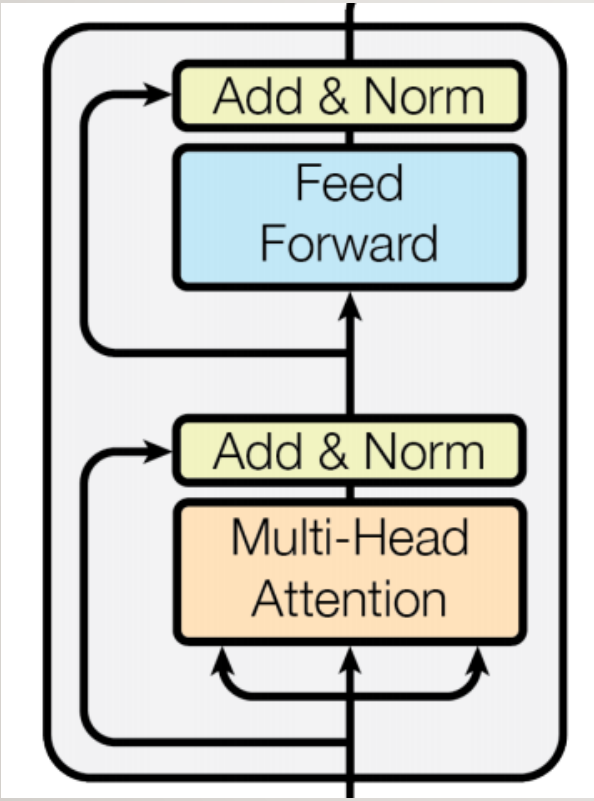
2022-4



- Transformer是2017年谷歌大脑团队在一篇名为《Attention Is All You Need》论文中提出的序列模型。

1. seq2seq的模型结构，用以序列预测序列的任务，例如机器翻译。
2. 由编码器(Encoder)与解码器(Decoder)组成，每个编码器由1个位置编码层与N个编码层(Encoder Layer)组成, 每个解码器由1个位置编码层与N个解码层(Decoder Layer)以及1个以Sotfmax为激活函数的全连接层组成。
3. 第t个编码层的输入是t-1个编码层的输出，解码层同理。
4. **模型的输入：**以机器翻译为例，例如是将中文翻译成英语的任务，训练时的数据应是中文、英语句子对。编码器输入的是中文，解码器输入的是英语。在做预测时，编码器输入的是要翻译的中文，解码器输入的是模型上一时刻的输出。
5. **模型的输出：**编码器的输出将作为解码器中某个模块的输入，解码器的输出即整个模型的输出是Sotfmax归一化后序列类别分布，通常维度是[Batch Size, 序列长度, 类别数量]。

编码层



- 图中有三种模块，分别是：

1. Multi-Head Attention，多头注意力。
2. Add & Norm，残差与Layer Normalization。
3. Feed Forward, 前馈神经网络。

多头注意力与缩放点乘注意力算法

- 缩放点乘注意力(Scaled Dot-Product Attention):

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

- 其中d_k指$\boldsymbol{K}$向量的维度，$\boldsymbol{Q}$代表Query向量，$\boldsymbol{K}$代表Key向量，$\boldsymbol{V}$代表Value向量。在“编码器”中，$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$都是由输入的序列向量得到。设$\boldsymbol{X}$为输入的序列向量,则：

$$\boldsymbol{Q} = \boldsymbol{W}_q\boldsymbol{X} + \boldsymbol{b}_q$$
$$\boldsymbol{K} = \boldsymbol{W}_k\boldsymbol{X} + \boldsymbol{b}_k$$
$$\boldsymbol{V} = \boldsymbol{W}_v\boldsymbol{X} + \boldsymbol{b}_v$$

  - $\boldsymbol{W}_q, \boldsymbol{b}_q, \boldsymbol{W}_k, \boldsymbol{b}_k, \boldsymbol{W}_v, \boldsymbol{b}_v$是模型需训练的三套不同的线性变化参数。

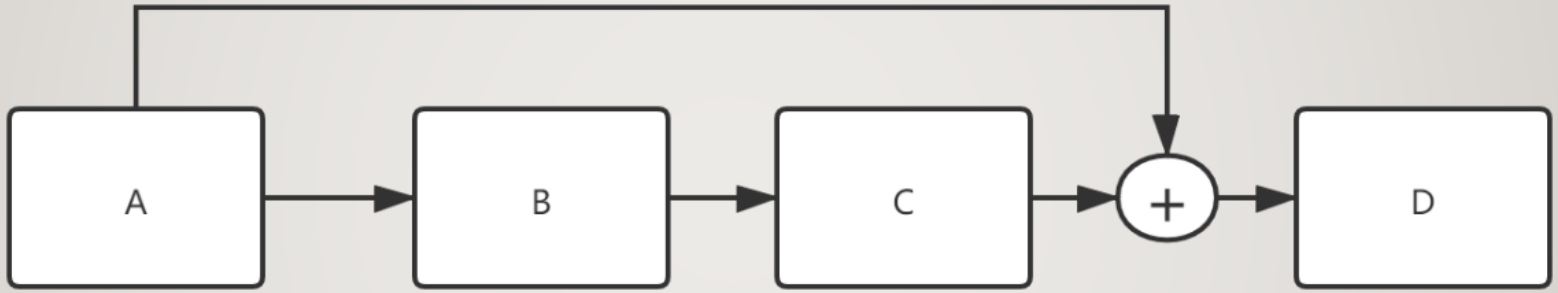
- 以上的注意力层可称为单头注意力层,而多头注意力Multi-Head Attention是指用不同的$\boldsymbol{W}_q^{\boldsymbol{i}}, \boldsymbol{b}_q^{\boldsymbol{i}}, \boldsymbol{W}_k^{\boldsymbol{i}}, \boldsymbol{b}_k^{\boldsymbol{i}}, \boldsymbol{W}_v^{\boldsymbol{i}}, \boldsymbol{b}_v^{\boldsymbol{i}}$的多计算几次单头注意力层的输出之后再全部拼接起来：

$$MultiHead(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = Concat(\boldsymbol{head1},\ldots,\boldsymbol{headh}) \cdot \boldsymbol{W}_O$$
$$where \quad \boldsymbol{headi} = Attention(\boldsymbol{Q}_i,\boldsymbol{K}_i,\boldsymbol{V}_i)$$

  - 其中W_O是模型需训练的线性变化矩阵，维度为[单头注意力层输出向量的维度 × head数量 ， 多头注意力层输入向量的维度(也就是输入向量X的维度)]。它的作用就是将经过多头注意力操作的向量维度再调整至输入时候的维度。

残差

- 所谓残差就是在经神经网络多层传递后加上最初的向量。作用是当网络层级深时可以有效防止梯度消失。



- 图中的传播方式可描述为:

$$D_{in} = A_{out} + C\left(B\left(A_{out}\right)\right)$$

- 根据链式求导法则，反向传播时为：

$$\frac{\partial D_{in}}{\partial A_{out}} = 1 + \frac{\partial C}{\partial B}\frac{\partial B}{\partial A_{out}}$$

- 所以不管网络多深，梯度上都会有个1兜底，不会为0造成梯度消失。

LAYER NORMALIZATION

- Layer Normalization (LN) 和 Batch Normalization(BN)类似，都是规范化数据的操作。公式写出来看起来也和BN一样，LN完整的公式如下：

$$\hat{\boldsymbol{a}}^l = \frac{\boldsymbol{a}^l - \mu^l}{\sqrt{(\sigma^l)^2 + \varepsilon}}$$

- 其中u^l代表第l个的均值，计算公式如下：

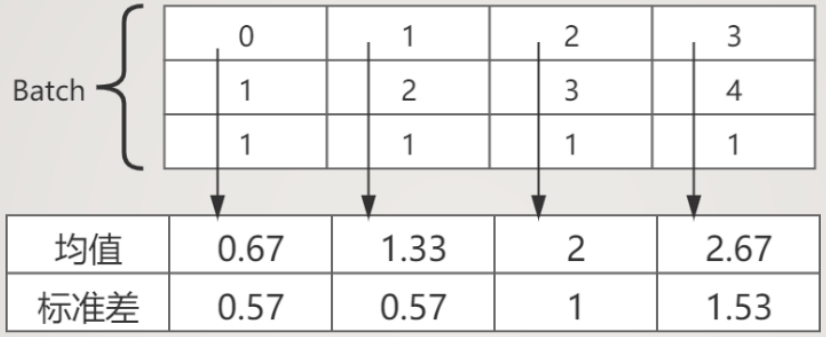
$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$$

- σ^l代表第l个标准差，计算公式如下：

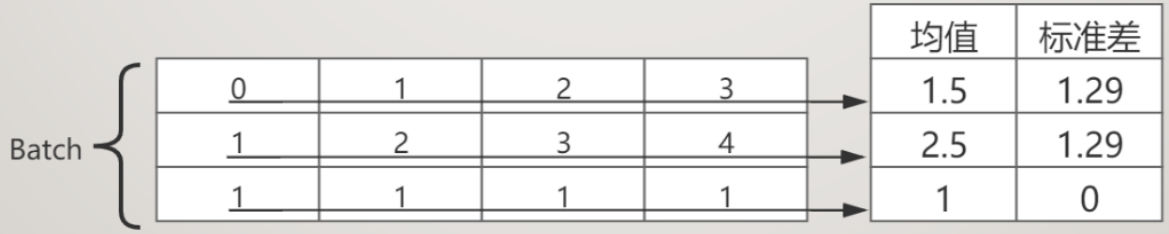
$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2}$$

- ε是干扰因子，标准差和均值的计算公式和普通的没什么区别，其实重点是什么叫做第l个。只要把BN和LN的区别给理解了就能理解l的含义，请看下一页。

BN与LN的区别



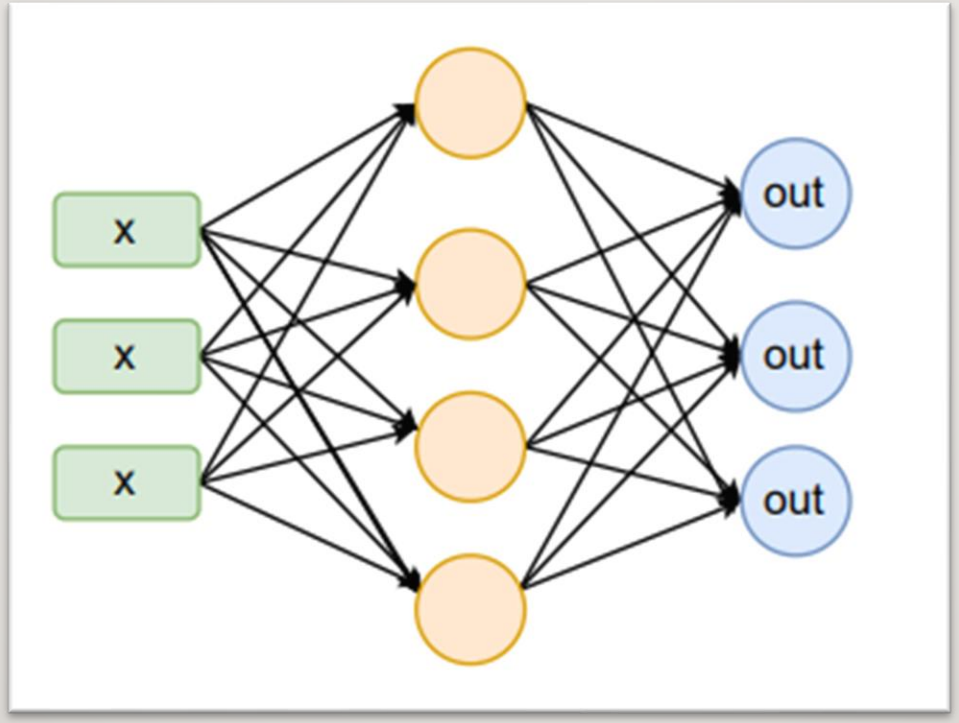
Batch Normalization



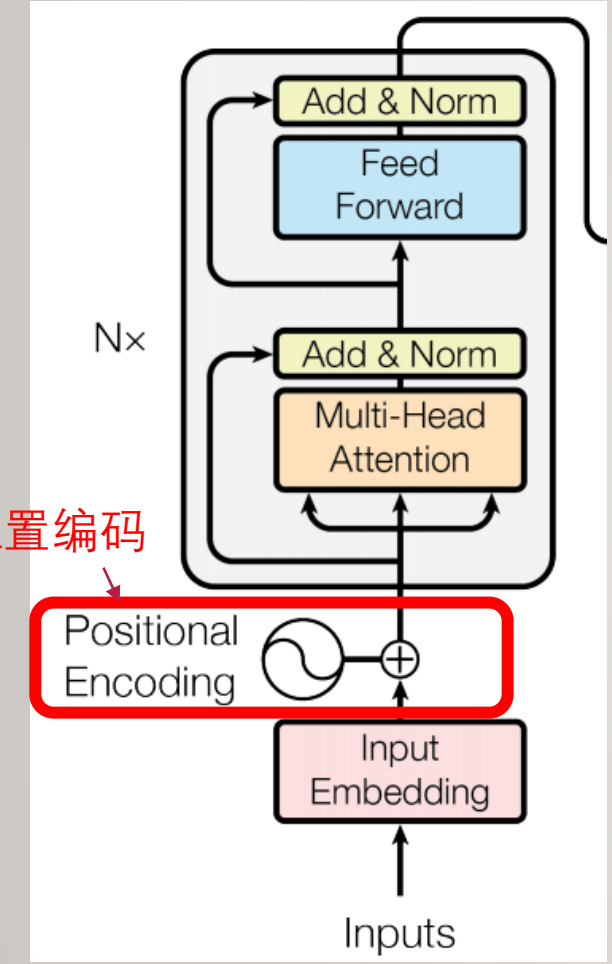
Layer Normalization

FEED FORWARD前馈神经网络层

- Transformer中所谓的前馈神经网络就是MLP的结构。唯一值得一提的是在这个MLP中，输入向量和输出向量维度是一样的，中间隐藏层的维度可随意调整。



位置编码



- 在输入编码层之前，需要进行位置编码：

$$\boldsymbol{emb}_{out} = \boldsymbol{emb}_{in} + \boldsymbol{PE}_{in}$$

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

- d_{model}代表这个模型中此时输入向量的维度。pos代表该输入的样本在序列中的位置，从0开始。2i和2i + 1 得看作两个整体，2i代表该向量中第2i偶数位，2i + 1就是第2i+1奇数位。
- 下图展示了pos与i的意义，图中的x0,x1,x2…代表该向量中的元素。

	2i=0	2i+1=1	2i=2	2i+1=3	2i=4	2i+1=5
pos=0	x0	x1	x2	x3	x4	x5
pos=1	x0	x1	x2	x3	x4	x5
pos=2	x0	x1	x2	x3	x4	x5

位置编码的原理

- 三角函数性质的公式：

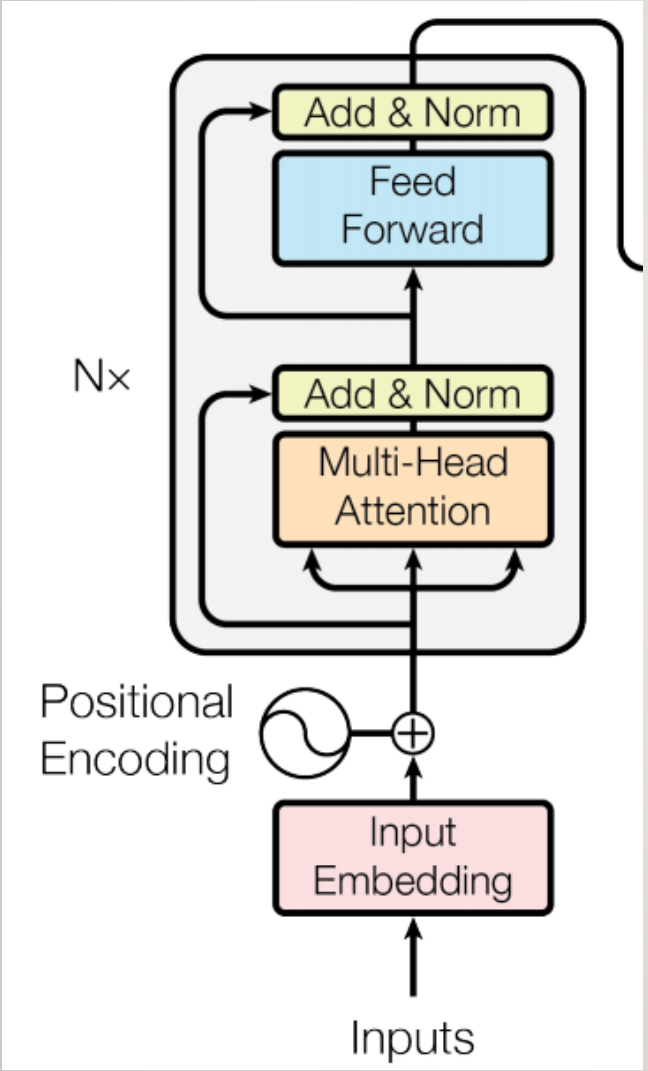
$$\sin(\alpha+\beta)=\sin\alpha\cos\beta+\cos\alpha\sin\beta$$
$$\cos(\alpha+\beta)=\cos\alpha\cos\beta-\sin\alpha\sin\beta$$

- 所以将 $\sin(*)=PE(*,2i)$, $\cos(*)=PE(*,2i+1)$ 代入上述公式可有：

$$PE(M+N,2i)=PE(M,2i)\times PE(N,2i+1)+PE(M,2i+1)\times PE(N,2i)$$
$$PE(M+N,2i+1)=PE(M,2i+1)\times PE(N,2i+1)+PE(M,2i)\times PE(N,2i)$$

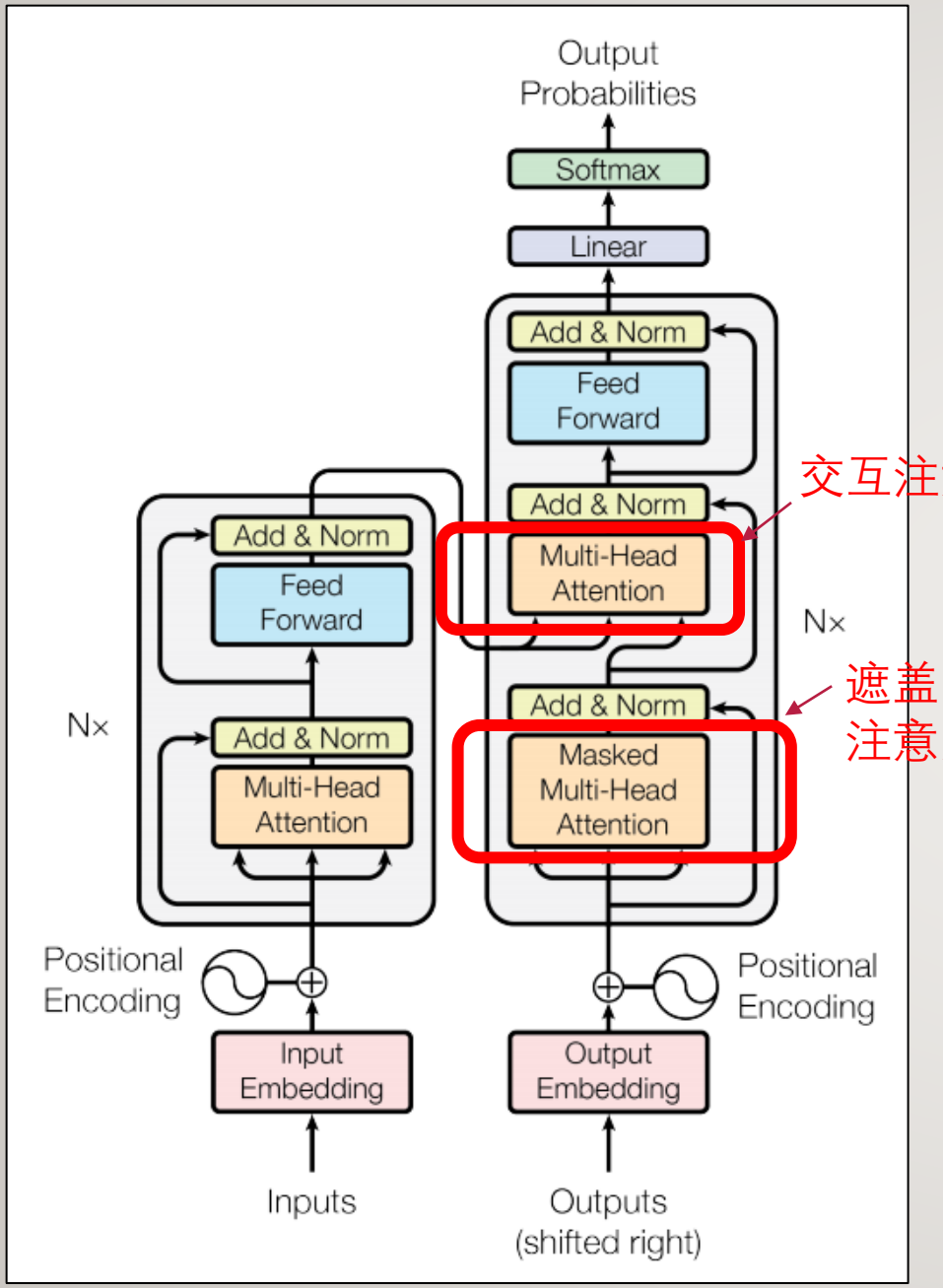
- 所以PE(M+N)可由PE(M)与PE(N)计算得到，也就是说各个位置间可以相互计算得到，所以每个向量都包含了相对位置的信息。

编码器




```
class TransformerEncoder(nn.Module):

    def __init__(self, e_dim, h_dim, n_heads, n_layers, drop_rate = 0.1 ):
        '''
        :param e_dim: 输入向量的维度
        :param h_dim: 注意力层中间隐含层的维度
        :param n_heads: 多头注意力的头目数量
        :param n_layers: 编码层的数量
        :param drop_rate: drop out的比例
        '''
        super().__init__()
        #初始化位置编码层
         self.position_encoding = PositionalEncoding( e_dim )
        #初始化N个"编码层"
        self.encoder_layers = nn.ModuleList( [EncoderLayer( e_dim, h_dim, n_heads, drop_rate )
                                              for _ in range( n_layers )] )
    def forward( self, seq_inputs ):
        '''
        :param seq_inputs: 已经经过Embedding层的张量，维度是[ batch, seq_lens, dim ]
        :return: 与输入张量维度一样的张量，维度是[ batch, seq_lens, dim ]
        '''
        #先进行位置编码
            seq_inputs = self.position_encoding( seq_inputs )
        #输入进N个"编码层"中开始传播
            for layer in self.encoder_layers:
                seq_inputs = layer( seq_inputs )
        return seq_inputs
```

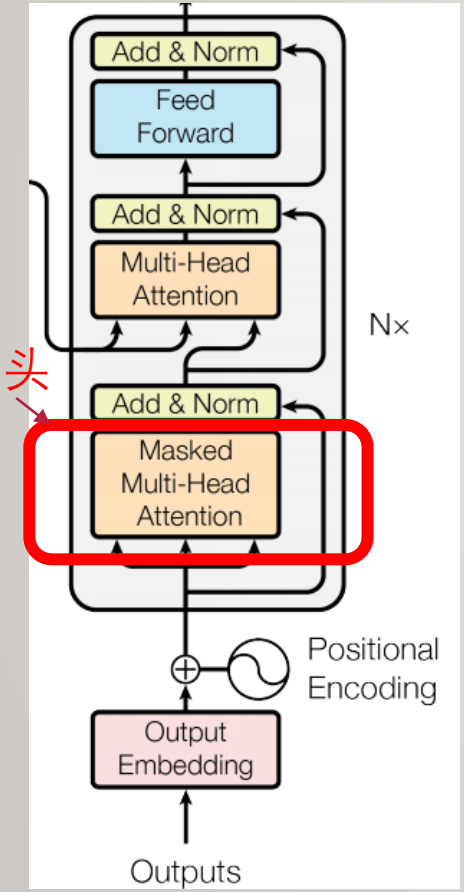


- 解码器比编码器多了：

1. 遮盖的多头注意力层
2. 交互注意力层

- 训练时解码器的输入是要预测的序列。

遮盖的多头注意力层

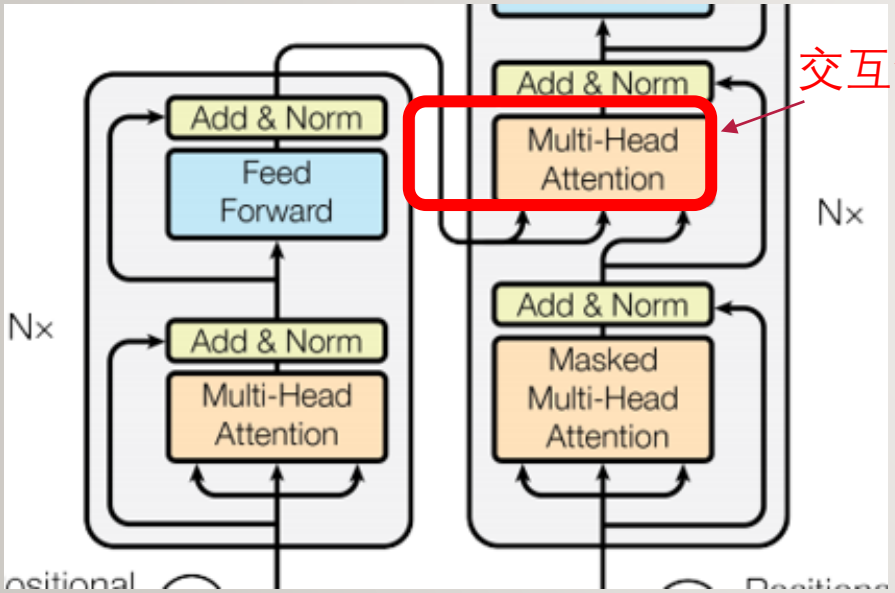


遮盖的多头注意力层

- 遮盖的多头注意力层Masked Multi-Head Attention
- 遮盖的意义是为了将未来信息掩盖住，使得训练出来的模型更准确。

➢ 例如输入“我爱新中国”，当轮到要预测“中” 时，模型获得的信息应该是“我爱新”这三个字。但是Transformer的Attention层做计算时是一整个序列张量输入进行计算，所以如果不加处理，预测“中”这个字时，模型获得的信息将会是“我爱新 国”这四个字。则“国”对于“中”来说显然属于未来信息。所以在训练时需要将未来信息都遮盖住。使模型在预测“中”时，获得的信息是 “我爱新**”。预测“新”时，获得的信息是“我爱***”。

交互注意力层



- 解码器中的交互注意力层与编码器中的注意力层唯一区别在于，前者计算Query向量的输入是编码器的输出。编码器的注意力层实际上被称为自注意力层。
- 自注意力层的的Q,K,V计算方式如下：

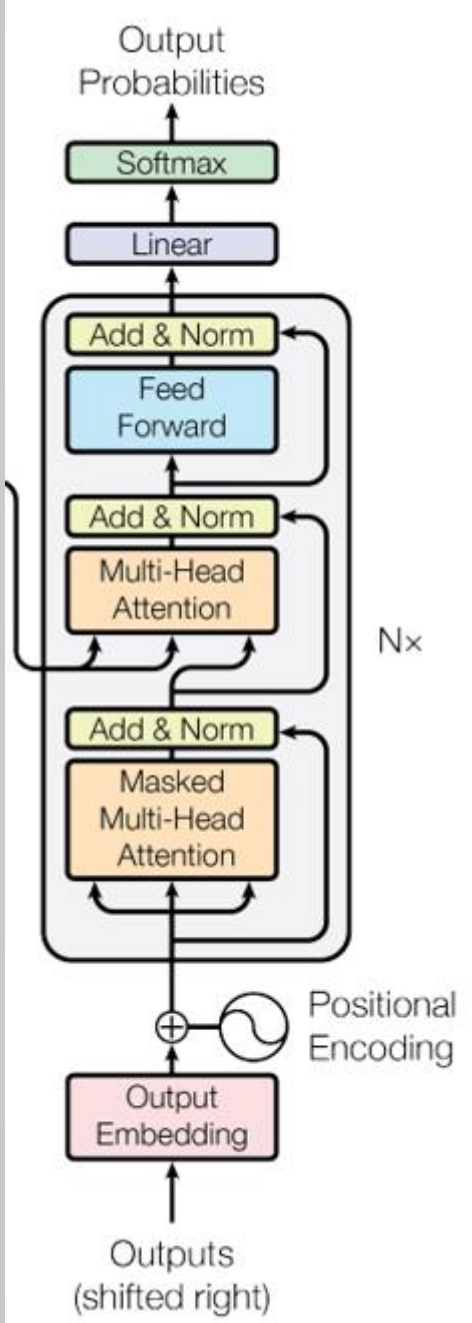
$$\boldsymbol{Q} = \boldsymbol{W}_q\boldsymbol{X} + \boldsymbol{b}_q$$
$$\boldsymbol{K} = \boldsymbol{W}_k\boldsymbol{X} + \boldsymbol{b}_k$$
$$\boldsymbol{V} = \boldsymbol{W}_v\boldsymbol{X} + \boldsymbol{b}_v$$

  - 其中的X即输入的序列。
- 交互注意力层中的Q计算方式如下：

$$\boldsymbol{Q} = \boldsymbol{W}_q\boldsymbol{Out}_{encoder} + \boldsymbol{b}_q$$

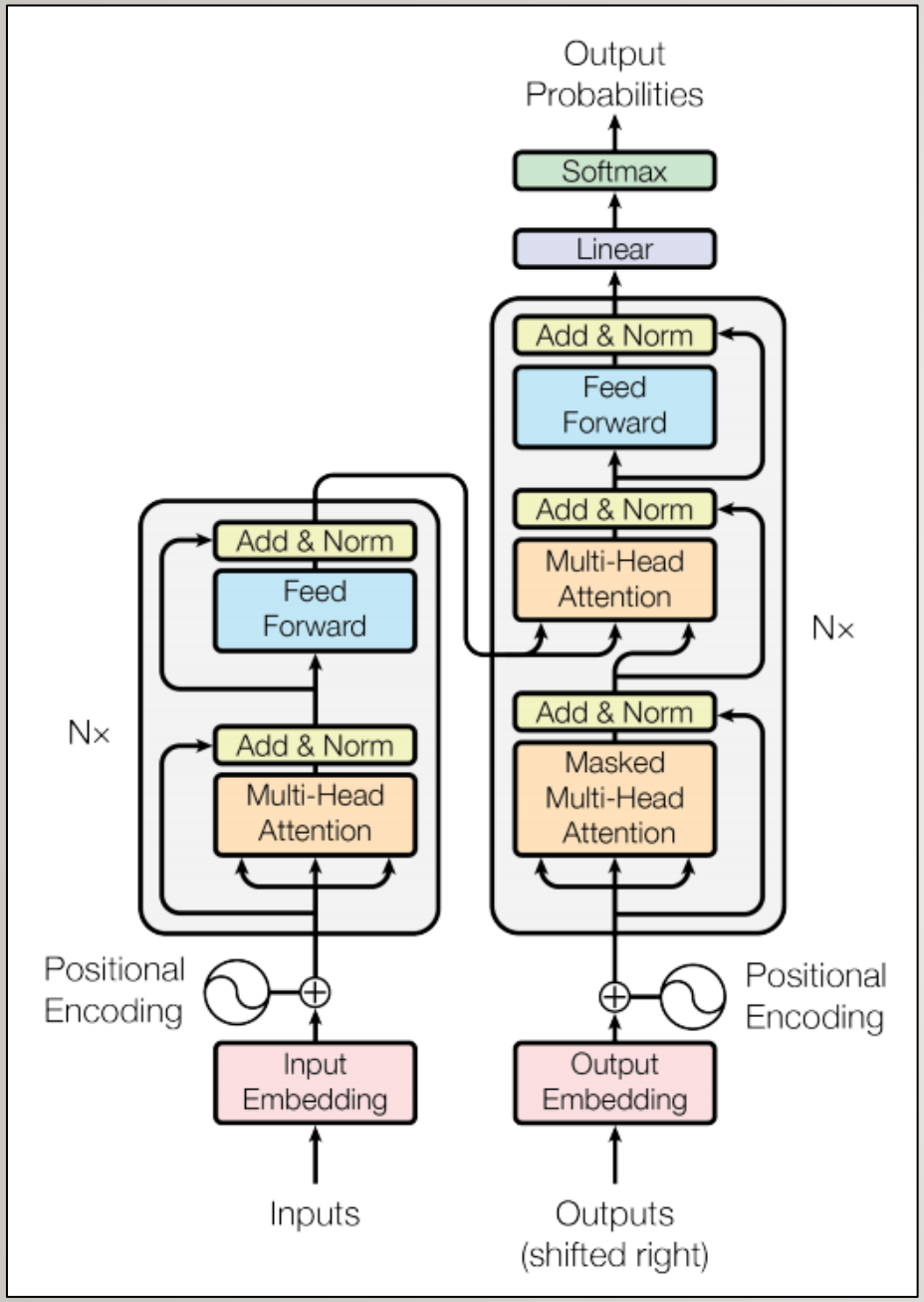
  - 其中的$\boldsymbol{Out}_{encoder}$即解码器的输出。

解码器




```python
class TransformerDecoder(nn.Module):
    def __init__(self, e_dim, h_dim, n_heads, n_layers, n_classes,drop_rate = 0.1 ):
        '''':param e_dim: 输入向量的维度
        :param h_dim: 注意力层中间隐含层的维度
        :param n_heads: 多头注意力的头目数量
        :param n_layers: 解码层的数量
        :param n_classes: 类别数
        :param drop_rate: drop out的比例'''
        super().__init__()
        # 初始化位置编码层
        self.position_encoding = PositionalEncoding( e_dim )
        # 初始化N个"解码层"
        self.decoder_layers = nn.ModuleList( [DecoderLayer( e_dim, h_dim, n_heads, drop_rate ) for _ in range( n_layers )] )
        # 线性层
        self.linear = nn.Linear(e_dim,n_classes)
        # softmax激活函数
        self.softmax = nn.Softmax()
    def forward( self, seq_inputs, querys ):
        '''':param seq_inputs: 已经经过Embedding层的张量，维度是[ batch, seq_lens, dim ]
         :param querys: encoder的输出，维度是[ batch, seq_lens, dim ]
         :return: 与输入张量维度一样的张量，维度是[ batch, seq_lens, dim ]'''
        # 先进行位置编码
        seq_inputs = self.position_encoding( seq_inputs )
        # 得到mask序列
        mask = subsequent_mask( seq_inputs.shape[1] )
        # 输入进N个"解码层"中开始传播
        for layer in self.decoder_layers:
            seq_inputs = layer( seq_inputs, querys, mask )
        # 最终线性变化后Softmax归一化
        seq_outputs = self.softmax(self.linear(seq_inputs))
        return seq_outputs
```

TRANSFORMER




```
class Transformer(nn.Module):

    def __init__(self,e_dim, h_dim, n_heads, n_layers, n_classes,drop_rate=0.1):
        super().__init__()
        self.encoder = TransformerEncoder(e_dim, h_dim, n_heads, n_layers, drop_rate)
        self.decoder = TransformerDecoder(e_dim, h_dim, n_heads, n_layers, n_classes,drop_rate)

    def forward(self,input,output):
        querys = self.encoder(input)
        pred_seqs = self.decoder(output,querys)
        return pred_seqs
```

结束