自然语言处理
Natural Language Processing

Chapter 7
Transformer模型

实验作业

Task: 机器翻译。采用Transformer架构

Data: WMT 2014 英-德翻译（输入英语，输出德语！）

Metrics: BLEU, Human evaluation

与上一次rnn-based seq2seq模型的结果进行比较，解释实验现象

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

(来源：Attention is all you need)

Outline

- Transformer结构
  - 重点：自注意力，多头注意力，掩码注意力
- 自注意力思考

回顾Seq2seq

- 编解码器分别基于RNN
- 使用注意力机制(attention) 帮助解码

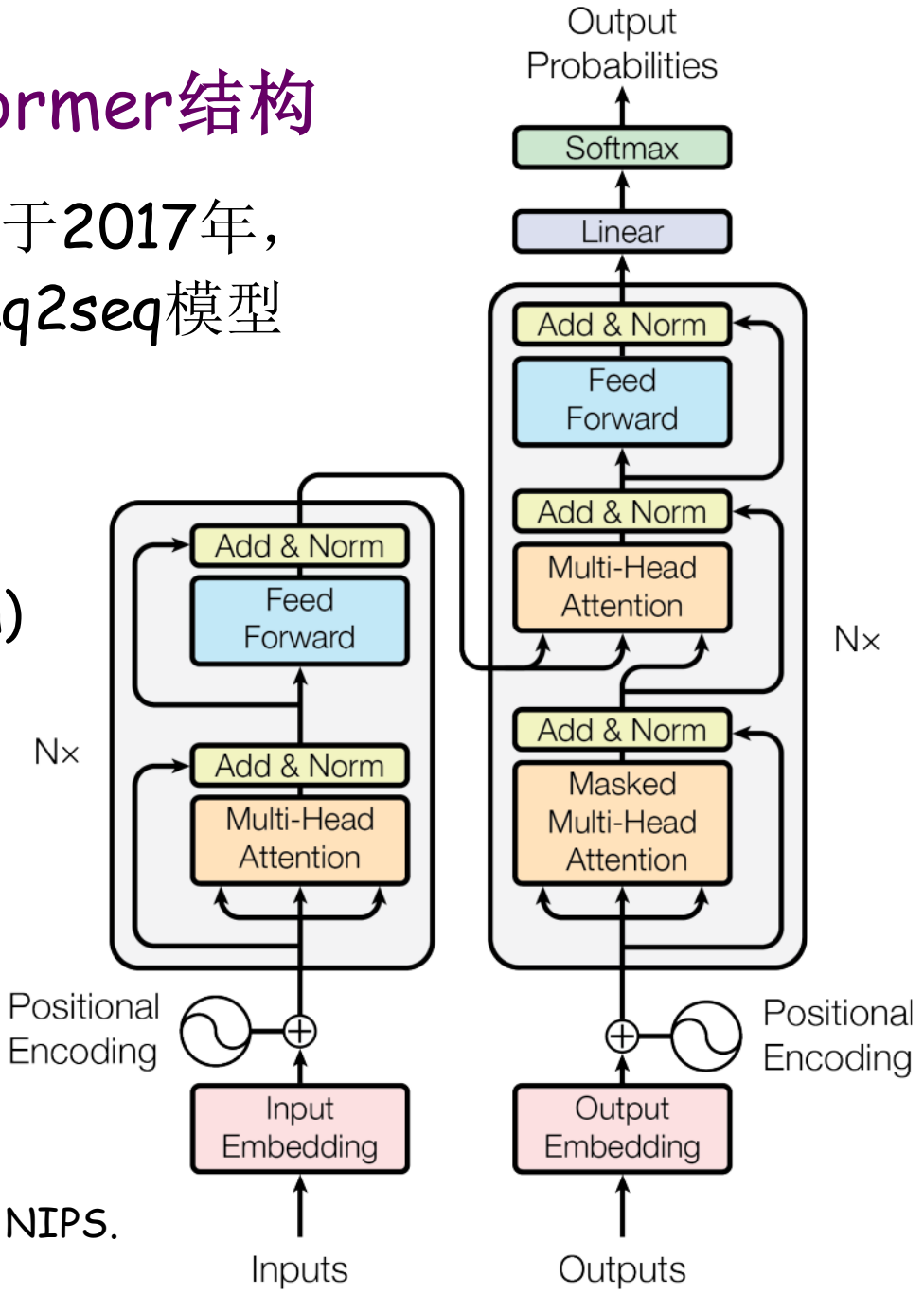
- 问题：RNN基座不能并行化操作

- 思路：将RNN替换为纯attention操作 → Transformer

Transformer结构

- Transformer由谷歌提出于2017年，是一种基于纯注意力的seq2seq模型

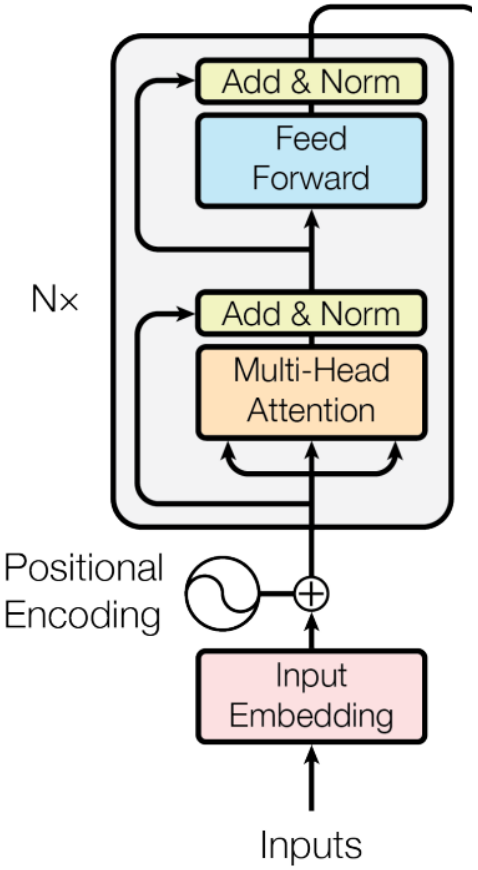
- 模型特点：
  - 注意力形式：多头自注意力(multi-head self attention)
  - 结构：编解码器各自堆叠N层，结构相同，参数不共享。每层内包含2/3个子层
  - 弥补顺序关系缺失：使用位置编码



Vaswani, Shazeer and Parmar et al, 2017. NIPS.
Attention Is All You Need

编码器

- 一层编码器内部包含2个主要子层，即multi-head self-attention (多头自注意力)，Feed Forward (前馈网络)
- 注意力层和前馈层后面紧跟一个add & norm操作
- token的位置编码以相加的形式加入到输入嵌入中



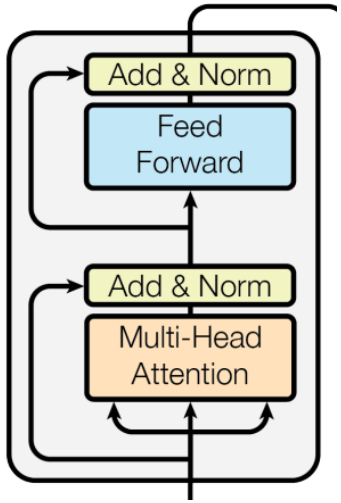
编码器实现：堆叠N个block

```python
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"

    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

N个层：clones函数 (复制某个结构)

```python
def clones(module, N):  # 定义一个clones函数，来更方便的将某个结构复制若干份
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])


class Encoder(nn.Module):
    """
    Encoder: The encoder is composed of a stack of N=6 identical layers.
    """

    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        # 调用时会将编码器层传进来，我们简单克隆N分，叠加在一起，组成完整的Encoder
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

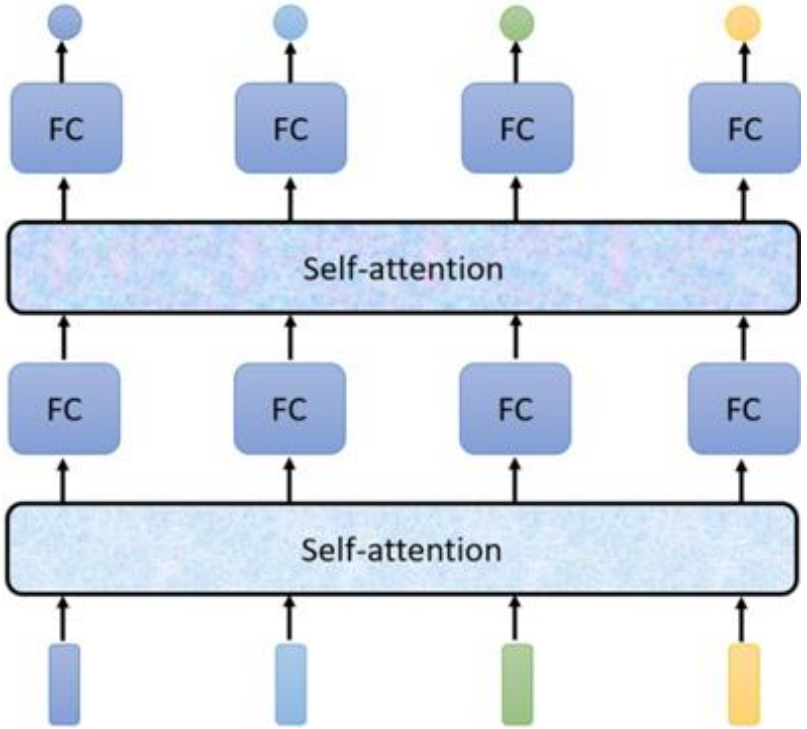
Encoder总体框架：堆叠N层，每一层是一个EncoderLayer



返回

自注意力

- 自注意力 (self-attention, 又叫intra-attention)
- 以一个序列作为输入，计算序列内每一个位置与其他位置的注意力权重，不需要与外界交互
- 假设输入句有K个token (词、字)，self-attention在K个位置之间建立联系，输出K个向量，其中每一个输出向量都是考虑了整个序列才得到的
- K个向量输出再进入全连接层
- 自注意力+全连接可以重复多次

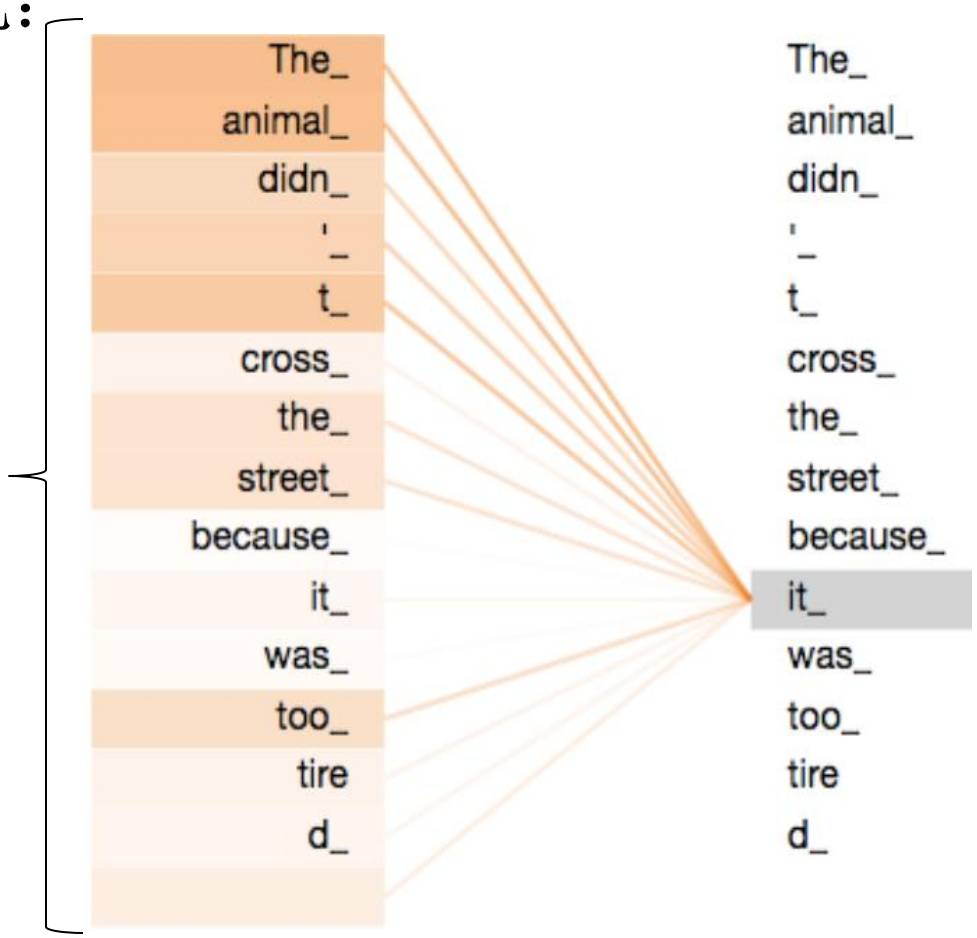


自注意力

- 在每一个位置，自注意力计算与序列中每一个位置的权重，帮助进行该位置的编码
- 自注意力权重可视化：

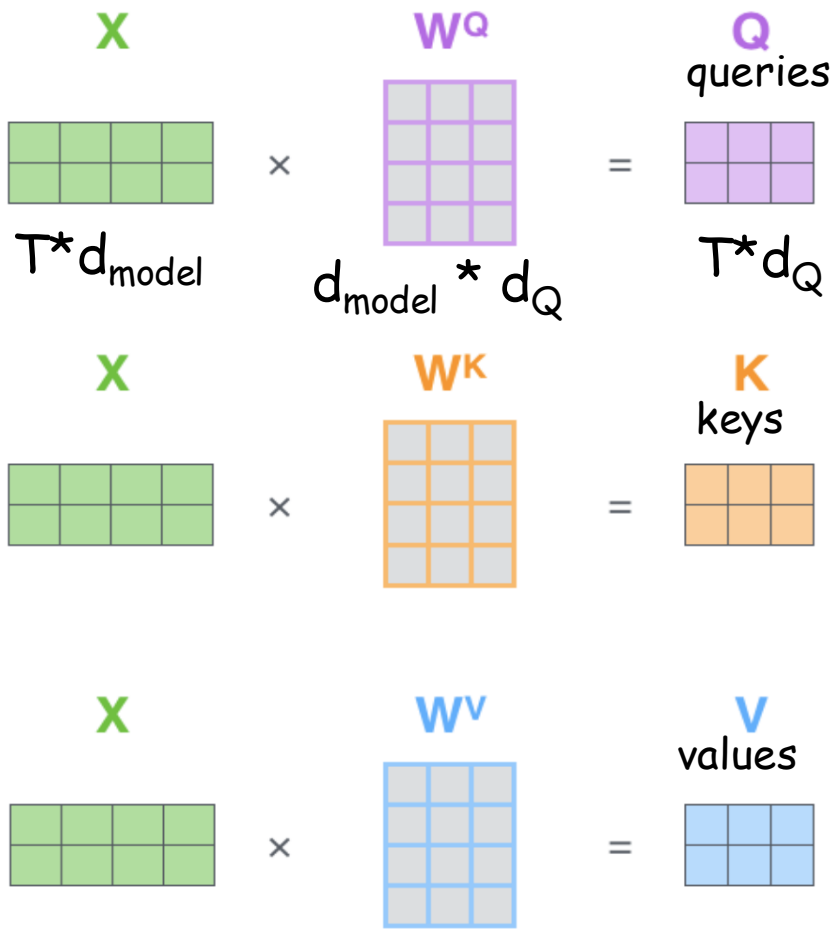
it和所有词语都进行交互，计算得出it和其他词语的相关程度，即注意力权重

站在it的角度，it是query，整个词语序列是values，attention计算只在句子内部进行



自注意力的矩阵乘法

- 每次自注意力计算需要用到3个向量：query，key，value
  - 这三个向量通过输入序列的编码，分别乘以各自的权重得到。
  - 考虑整个输入序列，用矩阵表示如图。假设图中X每行代表每一个输入token，列代表token向量维数。
  - Q, K, V矩阵每行代表每个输入token的query, key, value向量

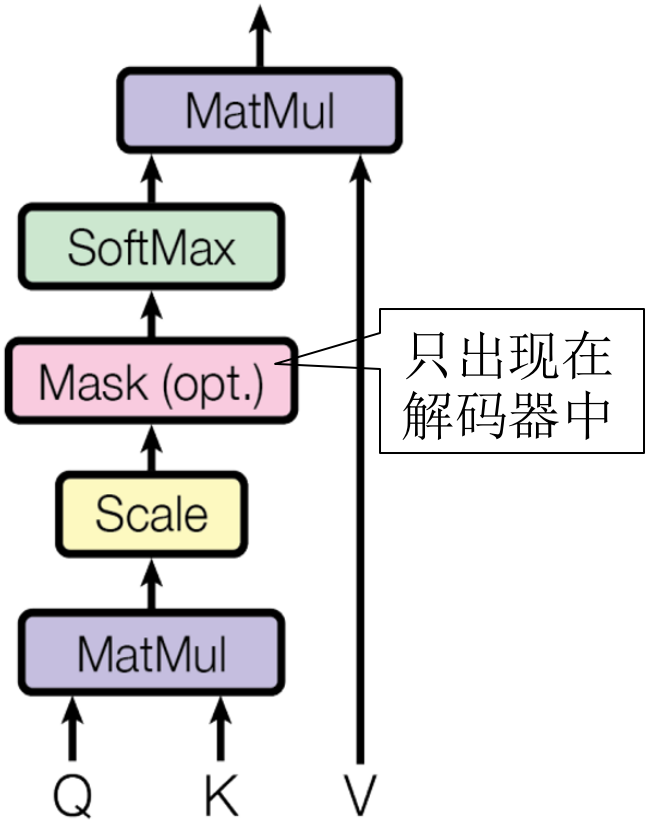


自注意力计算

- 注意力计算：缩放点积

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

注意力权重：希望关注语义上相关的词（权重较大），并弱化不相关的词（权重较小）



- query与一系列的key-value对的匹配，注意力输出就是values的加权和，每个value的权重由query和key的函数计算得出

- 之前所学的注意力计算：

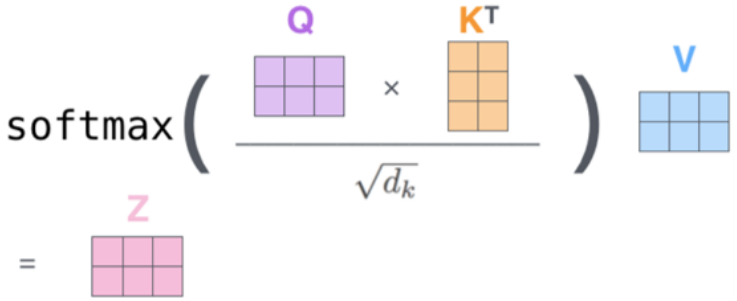
$$e_i = s^T h_i / \sqrt{h_i}$$

q　　k

$$\alpha_i = softmax(e_i)$$

$$a = \sum \alpha_i h_i$$

v



- 不同位置可以同步进行计算

attention：缩放点积注意力操作

```python
def attention(query, key, value, mask=None, dropout=None):
    # 首先取query的最后一维的大小，对应词嵌入维度
    d_k = query.size(-1)
    # 按照注意力公式，将query与key的转置相乘，这里面key是将最后两个维度进行转置，再除以缩放系数得到注意力得分张量scores
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

    # 接着判断是否使用掩码张量
    if mask is not None:
        # 使用tensor的masked_fill方法，将掩码张量和scores张量每个位置一一比较，if mask 则对应的scores张量用-1e9替换
        scores = scores.masked_fill(mask == 0, -1e9)

    # 对scores的最后一维进行softmax操作，使用F.softmax方法，这样获得最终的注意力张量
    p_attn = F.softmax(scores, dim=-1)

    # 判断是否使用dropout
    if dropout is not None:
        p_attn = dropout(p_attn)

    # 最后，根据公式将p_attn与value张量相乘获得最终的query注意力表示，同时返回注意力张量
    return torch.matmul(p_attn, value), p_attn
```

Decoder端第一个子层是掩码注意力，即需要进行mask。
Encoder端主要用于忽略padding的位置

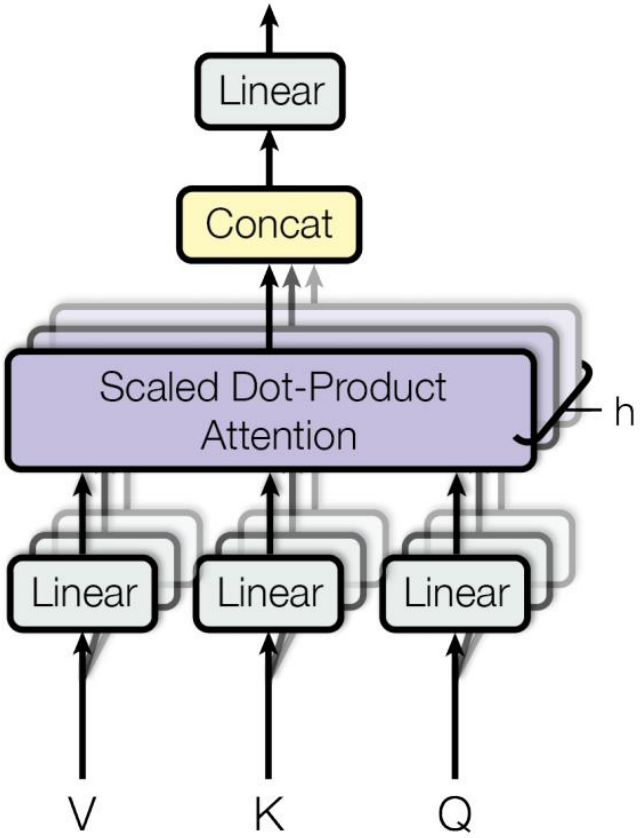
自注意力

- self-attention能够直接计算两个位置的词语之间的关系
  - 无论两个元素在序列中的相对距离如何，传播的距离总是相等的
- 每一个attention层可以并行化
- self-attention能够学到句子的内在结构
- self-attention有较好的泛化性，可以推广到多种任务
- 问题：自注意力是否能完全取代序列化计算？

  没有定论

多头自注意力

- Transformer中使用的自注意力是一种多头自注意力
- 即将输入X乘以不同的权重W_Q，得到不同的Q，同理得到不同的K和V。每一个Q, K, V的组合对应一个注意力头



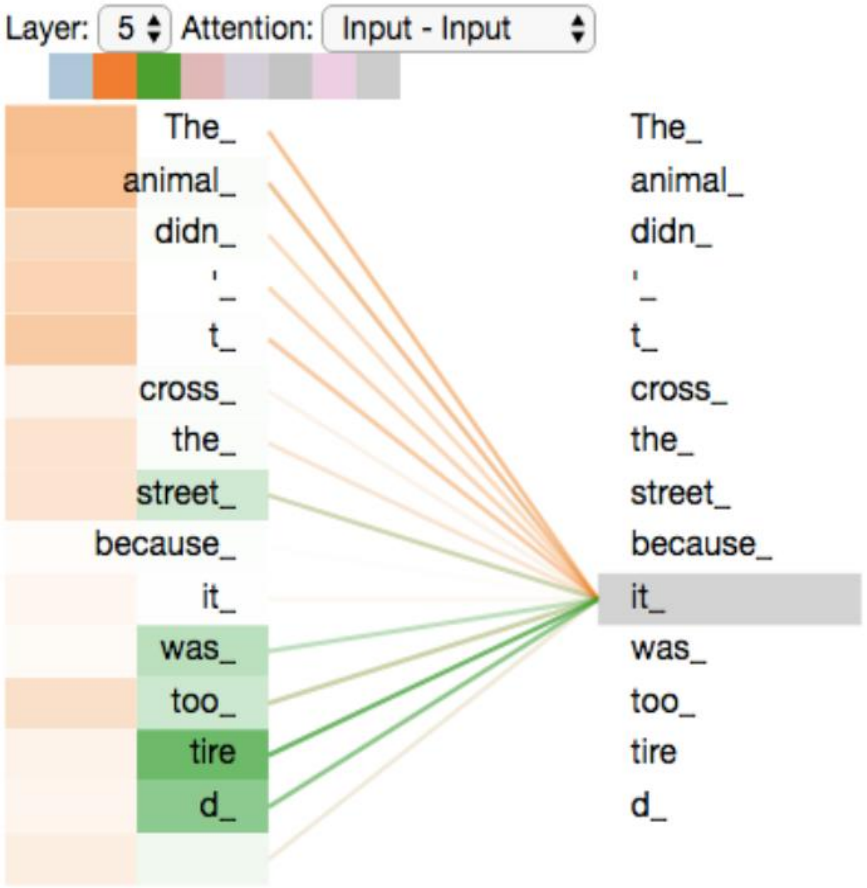
h: 头的数量

多头自注意力

- 多头自注意力允许模型关注到不同方面的相关性，提高模型的表达能力.

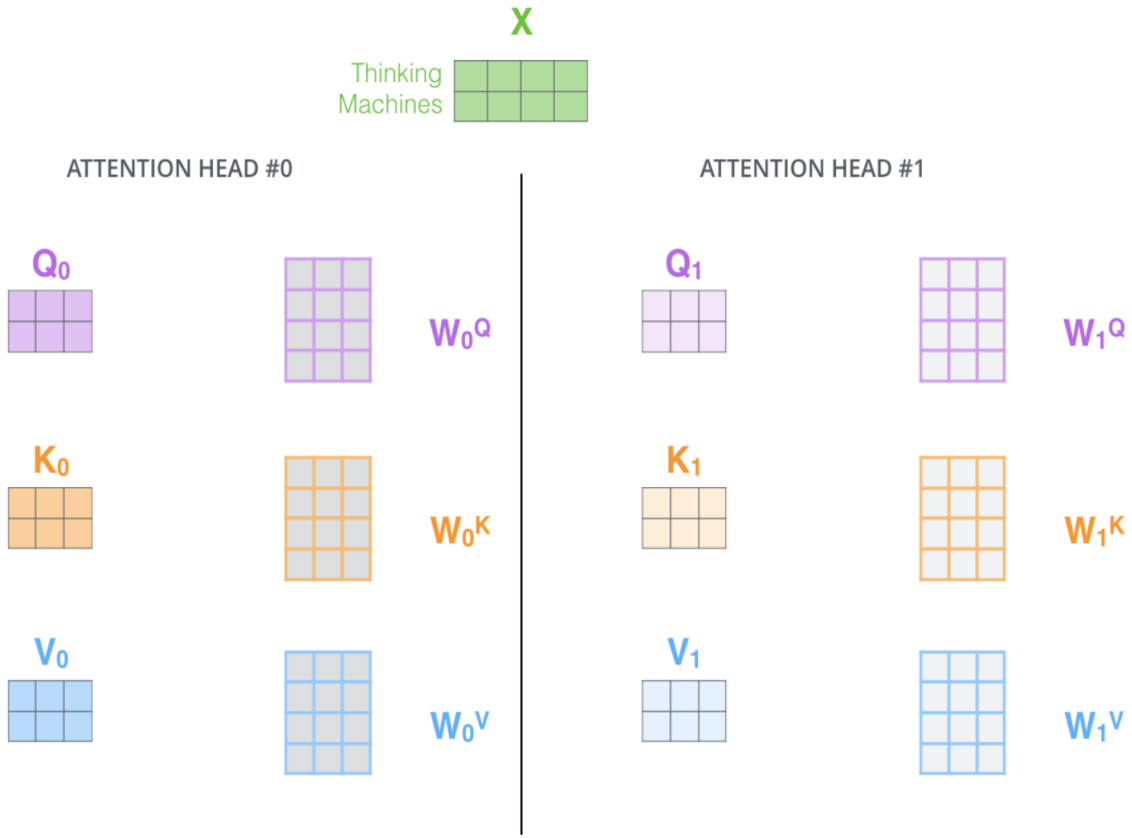
假设橙色和绿色分别代表一种注意力：

橙色关注了指代，绿色关注了形容的状态，属于不同方面的相关



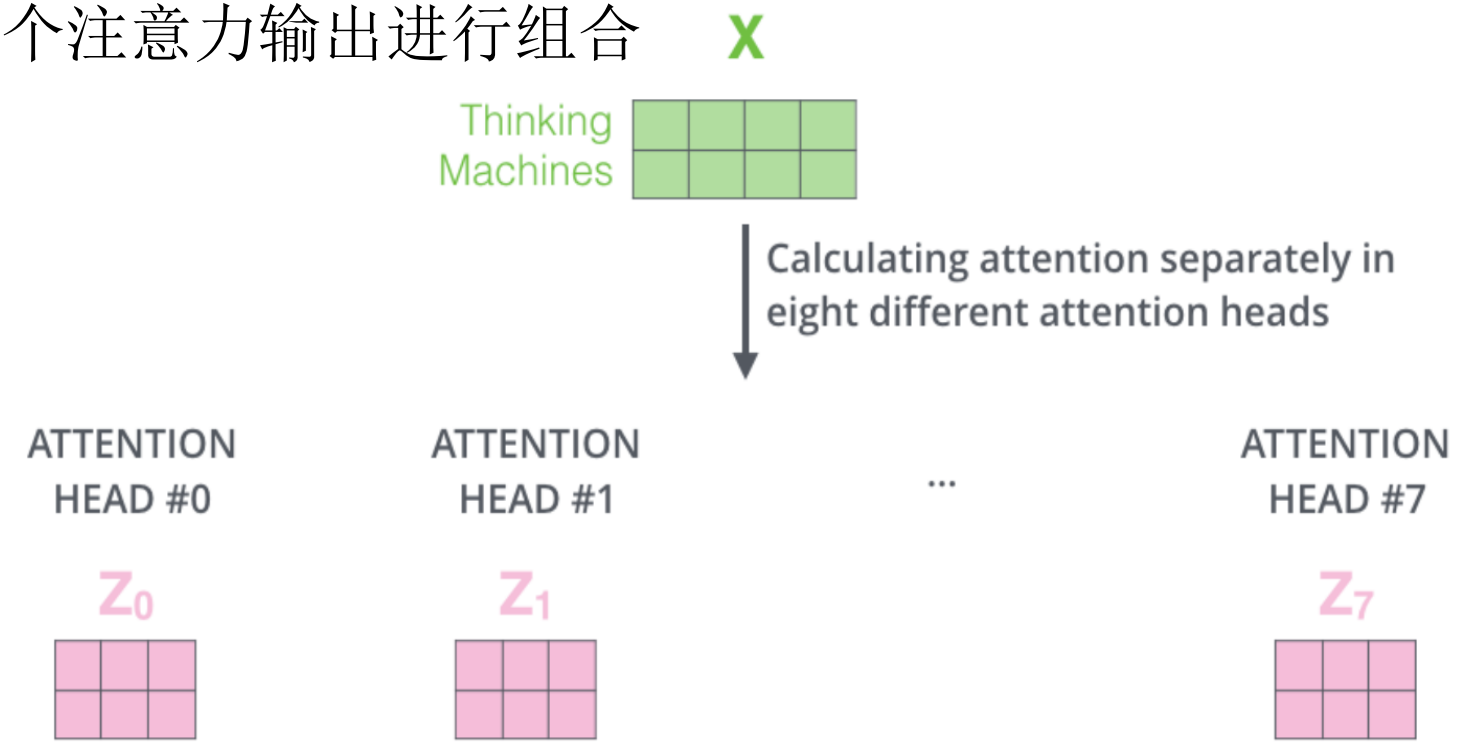
多头自注意力

- 多头能提供多个表示的子空间.
  - 对于不同的head，QKV各自进行初始化，各自计算注意力，各自训练，训练之后就把X或者上一层的输出映射到了不同的子空间



多头自注意力

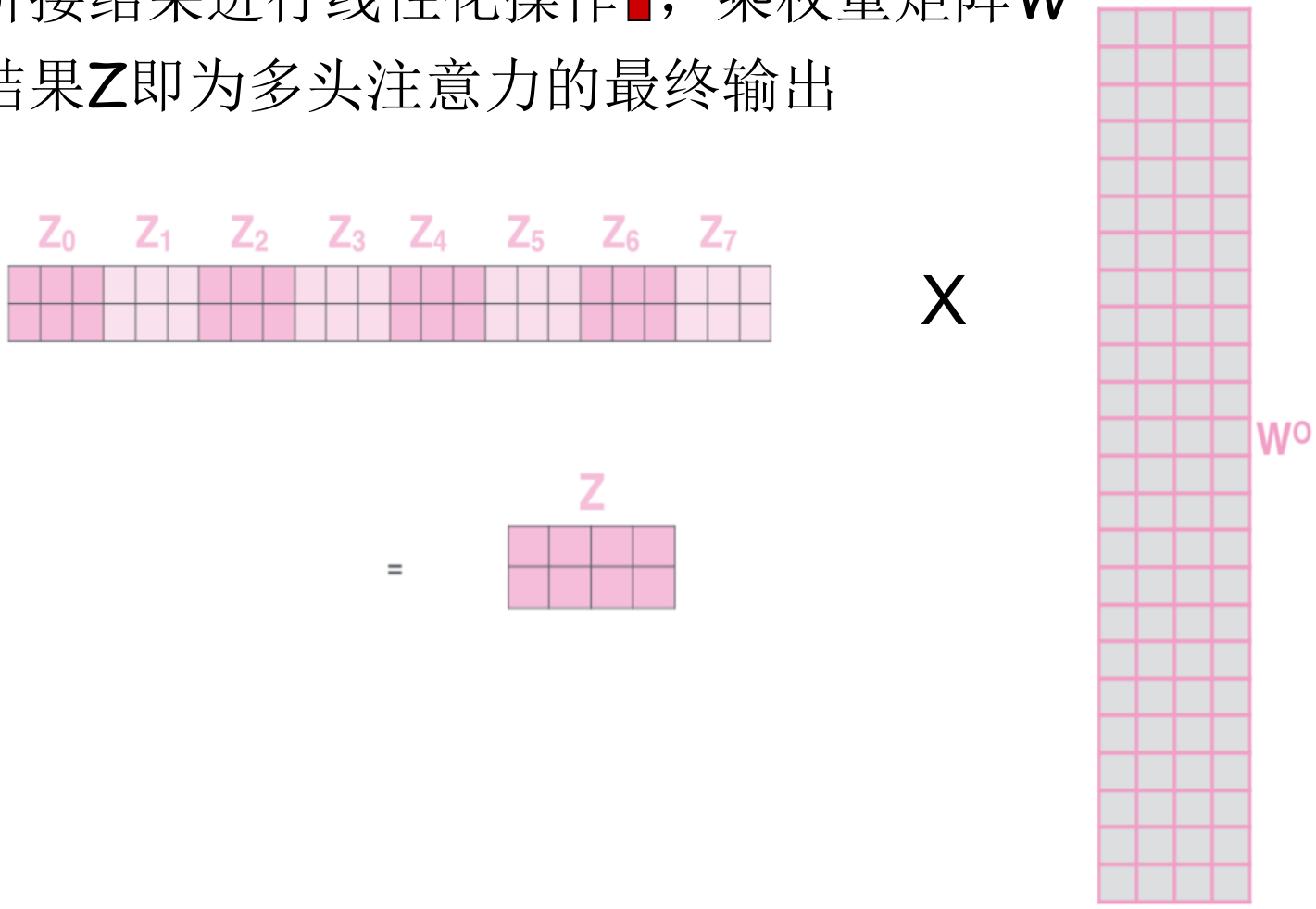
- 对于每一个head，独立计算得到一个注意力输出 (attention output)
- 多头可以并行计算
- 多个注意力输出进行组合



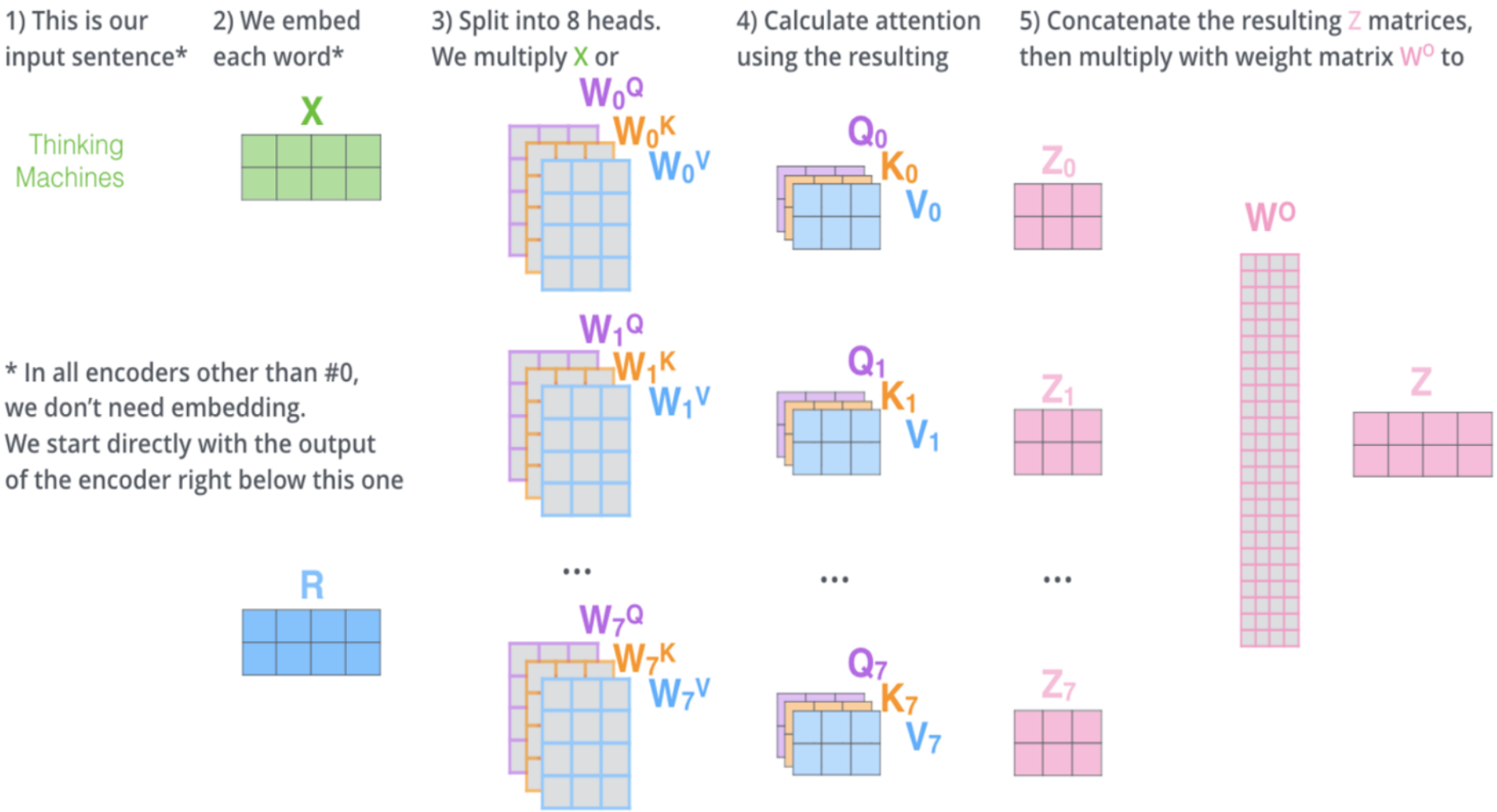
多头自注意力

- 多头输出压缩成一个矩阵：

1) 采用拼接方式，将多头输出拼接为一个矩阵
2) 拼接结果进行线性化操作，乘权重矩阵W^O
3) 结果Z即为多头注意力的最终输出



多头自注意力



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

self_attn：多头自注意力 (1)

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        # h代表头数，d_model代表词嵌入的维度，dropout代表进行dropout操作时置0比率，默认是0.1
        super(MultiHeadedAttention, self).__init__()
        # 判断h是否能被d_model整除，这是因为我们之后要给每个头分配等量的词特征，也就是embedding_dim/head个
        assert d_model % h == 0
        # 得到每个头获得的分割词向量维度d_k
        self.d_k = d_model // h
        # 传入头数h
        self.h = h

        # 创建linear层，通过nn的Linear实例化，它的内部变换矩阵是embedding_dim x embedding_dim，然后使用，
        # 为什么是四个呢，这是因为在多头注意力中，Q,K,V各需要一个，最后拼接的矩阵还需要一个，因此一共是四个
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        # self.attn为None，它代表最后得到的注意力张量，现在还没有结果所以为None
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)
```

self_attn：多头自注意力 (2)

```
def forward(self, query, key, value, mask=None):
    if mask is not None:  # Same mask applied to all h heads.
        mask = mask.unsqueeze(1)  # unsqueeze扩展维度，代表多头中的第n头
    nbatches = query.size(0)  # 有多少条样本

    # 1) Do all the linear projections in batch from d_model => h x d_k
    # 首先将QKV与三个线性层组到一起，然后利用for循环，将输入QKV分别传到线性层中，
    # 使用view方法对线性变换的结构进行维度重塑，多加了一个维度h代表头，
    # 然后对第二维和第三维进行转置操作，为了让代表句子长度维度和词向量维度能够相邻，
    query, key, value = [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
                         for l, x in zip(self.linears, (query, key, value))]

    # 2) Apply attention on all the projected vectors in batch.
    # 得到每个头的输入后，将他们传入到attention函数中，得到每个头计算结果组成的4维张量
    x, self.attn = attention(query, key, value, mask=mask, dropout=self.dropout)

    # 3) "Concat" using a view and apply a final linear.
    # 需要将多头输出转换为输入的形状以方便后续计算，因此这里进行第一步处理环节的逆操作，对第二和第三维进行转置，
    # 然后使用contiguous方法。这个方法的作用就是能够让转置后的张量应用view方法，
    # 下一步使用view重塑形状，变成和输入形状相同。
    x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)
    # 最后使用线性层列表中的最后一个线性变换得到最终的多头注意力结构的输出
    return self.linears[-1](x)
```

concat

=d_model

返回

位置编码

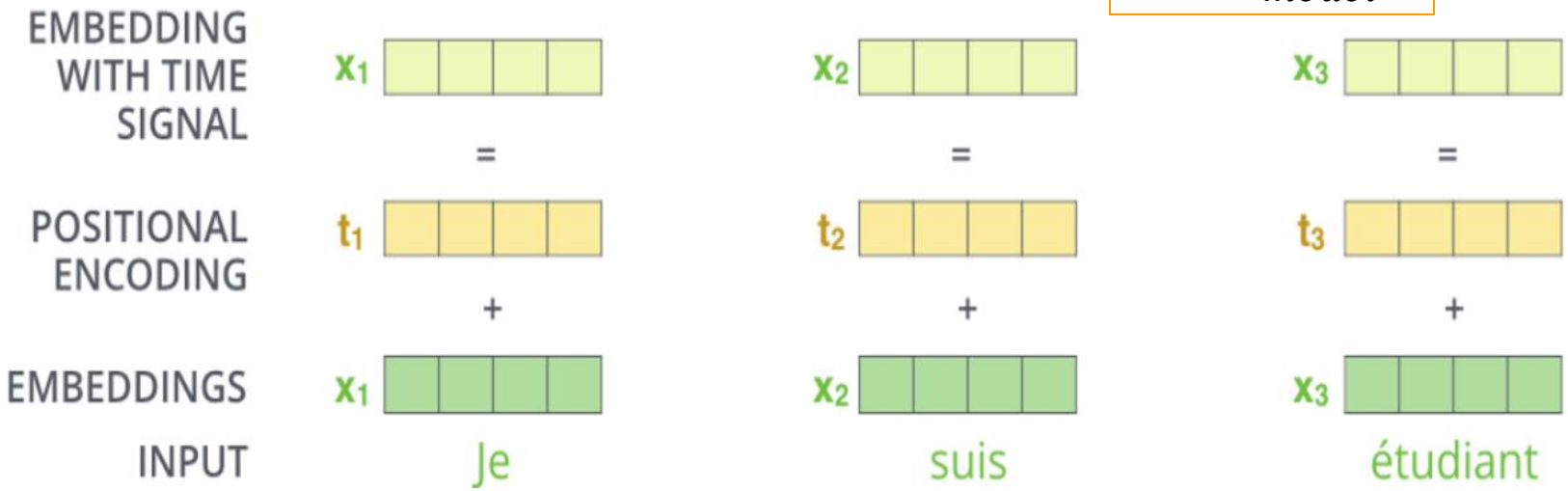
- 在self-attention中，任意两个位置的词直接交互，此时没有位置信息。因此，为每一个位置设置了一个位置编码 (positional embedding)

$$p_{i,j} = \begin{cases} \sin\left(\dfrac{i}{10000^{\frac{j}{d_{emb\_dim}}}}\right) & \text{if j 是偶数} \\ \cos\left(\dfrac{i}{10000^{\frac{j-1}{d_{emb\_dim}}}}\right) & \text{if j 是奇数} \end{cases}$$

第i个词

向量第j维

设为d_{model}



位置编码

- 位置向量举例

位置索引i, 维度索引j均从0开始

$$p_{i,j} = \begin{cases} \sin\left(\frac{i}{10000^{\frac{j}{d_{emb\_dim}}}}\right) & j \text{ 是偶数} \\ \cos\left(\frac{i}{10000^{\frac{j-1}{d_{emb\_dim}}}}\right) & j \text{ 是奇数} \end{cases}$$

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{pmatrix} \sin\left(\frac{0}{10000^{\frac{0}{emb_{dim}}}}\right) & \cos\left(\frac{0}{10000^{\frac{0}{emb_{dim}}}}\right) & \sin\left(\frac{0}{10000^{\frac{2}{emb_{dim}}}}\right) & \cos\left(\frac{0}{10000^{\frac{2}{emb_{dim}}}}\right) & \cdots \\ \sin\left(\frac{1}{10000^{\frac{0}{emb_{dim}}}}\right) & \cos\left(\frac{1}{10000^{\frac{0}{emb_{dim}}}}\right) & \sin\left(\frac{1}{10000^{\frac{2}{emb_{dim}}}}\right) & \cos\left(\frac{1}{10000^{\frac{2}{emb_{dim}}}}\right) & \cdots \\ \sin\left(\frac{2}{10000^{\frac{0}{emb_{dim}}}}\right) & \cos\left(\frac{2}{10000^{\frac{0}{emb_{dim}}}}\right) & \sin\left(\frac{2}{10000^{\frac{2}{emb_{dim}}}}\right) & \cos\left(\frac{2}{10000^{\frac{2}{emb_{dim}}}}\right) & \cdots \\ \sin\left(\frac{3}{10000^{\frac{0}{emb_{dim}}}}\right) & \cos\left(\frac{3}{10000^{\frac{0}{emb_{dim}}}}\right) & \sin\left(\frac{3}{10000^{\frac{2}{emb_{dim}}}}\right) & \cos\left(\frac{3}{10000^{\frac{2}{emb_{dim}}}}\right) & \cdots \\ \sin\left(\frac{4}{10000^{\frac{0}{emb_{dim}}}}\right) & \cos\left(\frac{4}{10000^{\frac{0}{emb_{dim}}}}\right) & \sin\left(\frac{4}{10000^{\frac{2}{emb_{dim}}}}\right) & \cos\left(\frac{4}{10000^{\frac{2}{emb_{dim}}}}\right) & \cdots \\ \sin\left(\frac{5}{10000^{\frac{0}{d_{emb_{dim}}}}}\right) & \cos\left(\frac{5}{10000^{\frac{0}{emb_{dim}}}}\right) & \sin\left(\frac{5}{10000^{\frac{2}{emb_{dim}}}}\right) & \cos\left(\frac{5}{10000^{\frac{2}{emb_{dim}}}}\right) & \cdots \end{pmatrix}$$

位置编码

- 位置编码主要有两种设置方法：
- 1) 固定的人工设定表示 ✓
- 2) 从数据中学习表示

$$p_{i,j} = \begin{cases} \sin\left(\frac{i}{10000^{\frac{j}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{j-1}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

- 一般选择固定的方法:
  - sin、cos函数可以接受任意大的输入，因此允许任意长度的输入文本，针对任意位置进行计算。
  - 无需引入参数，无需学习过程，训练更快

位置编码代码实现

```
class PositionalEncoder(nn.Module):
    def __init__(self, d_model, max_seq_len = 80):
        super().__init__()
        self.d_model = d_model

        # 根据 pos 和 i 创建一个常量 PE 矩阵
        pe = torch.zeros(max_seq_len, d_model)
        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
                pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
                pe[pos, i + 1] =  math.cos(pos / (10000 ** ((2 * (i + 1))/d_model)))

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # 使得单词嵌入表示相对大一些
        x = x * math.sqrt(self.d_model)
        # 增加位置常量到单词嵌入表示中
        seq_len = x.size(1)
        x = x + Variable(self.pe[:,:seq_len], requires_grad=False).cuda()
        return x
```

$$p_{i,j} = \begin{cases} \sin\left(\frac{i}{10000^{\frac{j}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{j-1}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

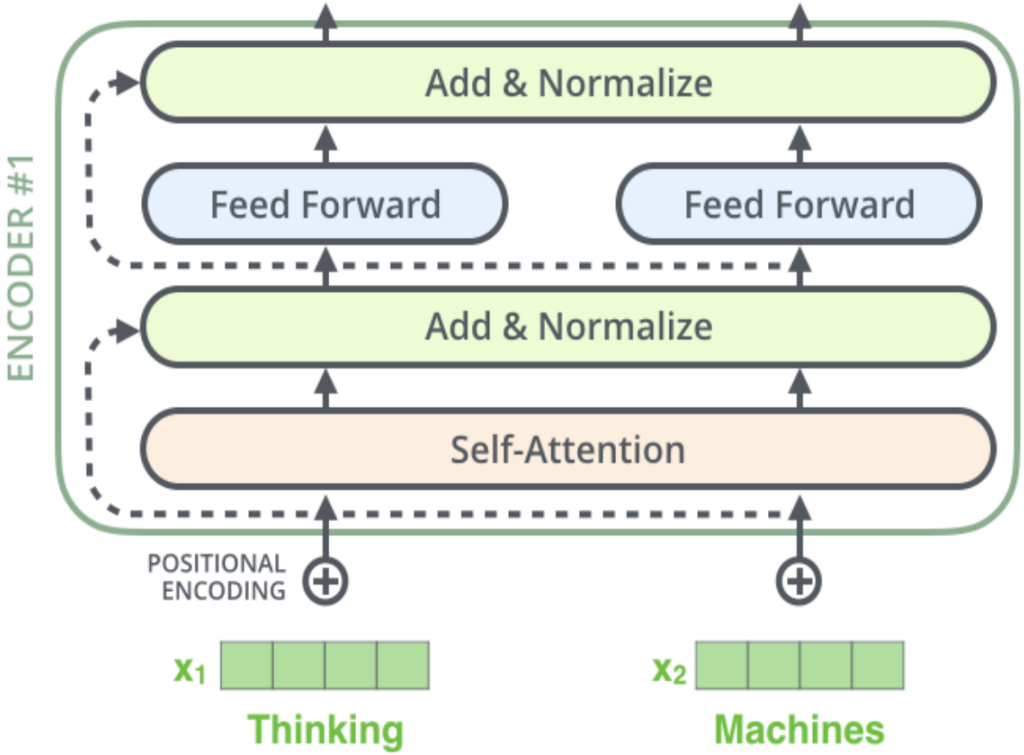
位置编码在训练过程中不更新

位置编码代码实现

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):
        """..."""
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        # 注意下面代码的计算方式与公式中给出的是不同的，但是是等价的，你可以尝试简单推导证明一下。
        # 这样计算是为了避免中间的数值计算结果超出float的范围，
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)], requires_grad=False)
        return self.dropout(x)
```

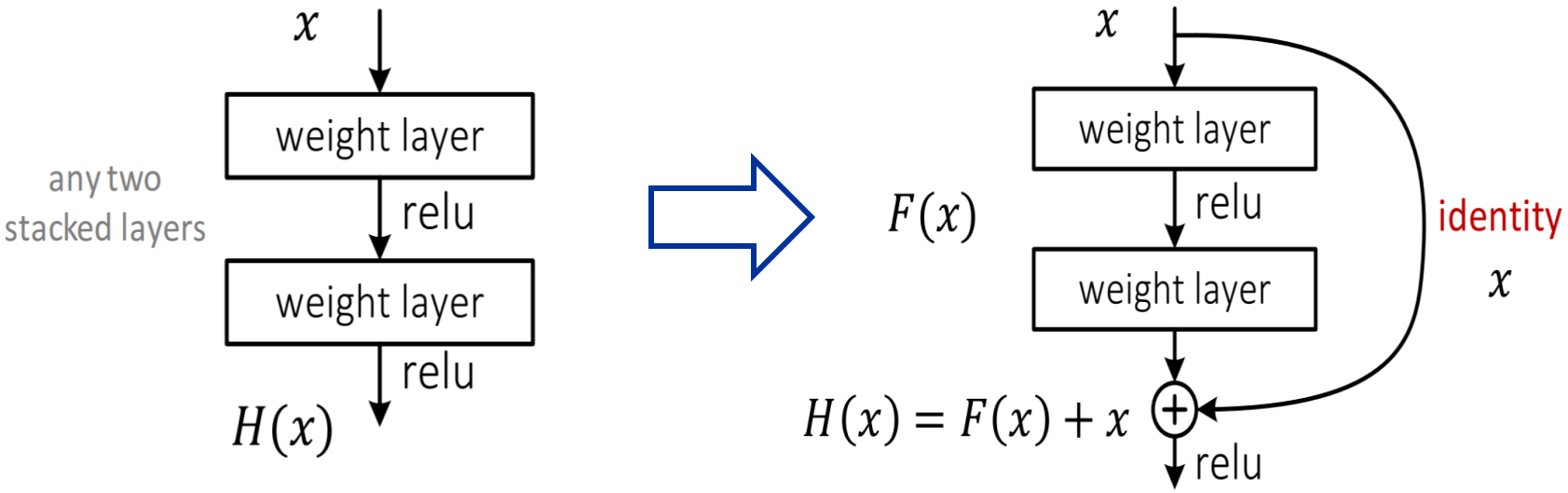
Add & Norm

- 在多头自注意力层和前馈层后，接着进行add & norm操作，即加法操作、归一化操作。



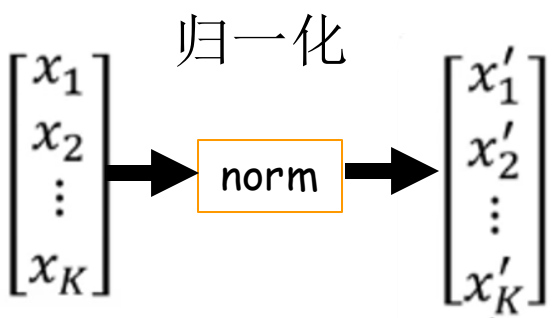
Add & Norm (1)

- Add: 残差连接，即 h = x + f(x)

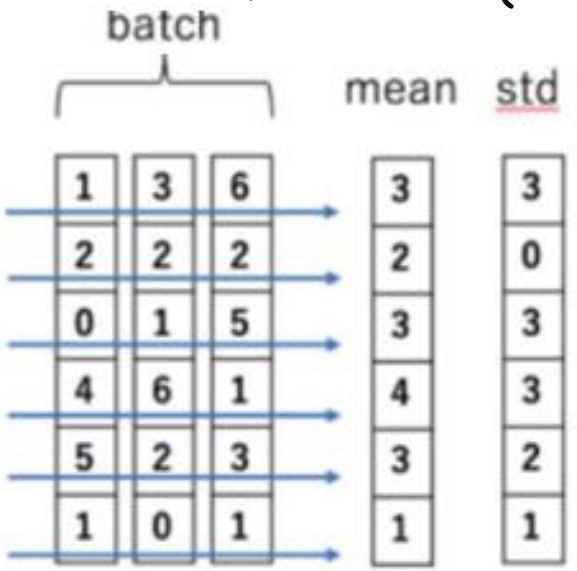


Add & Norm (2)

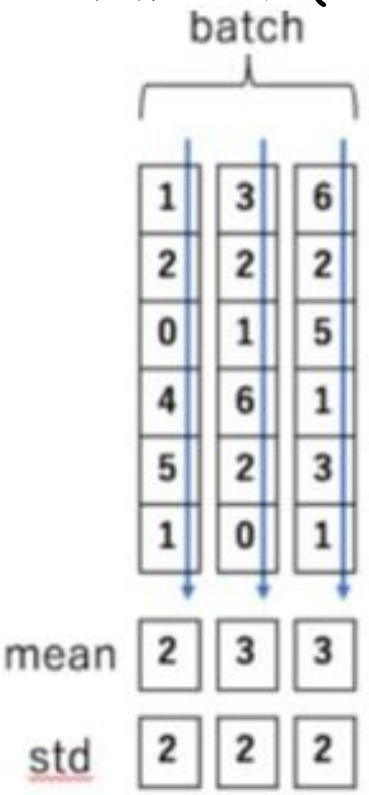
- Norm: 层归一化 Layer normalization



Batch normalization (BN)



层归一化(LN)



- 从原理操作来说，BN针对同一个batch内的所有数据，而LN针对单个样本。
- 从特征维度来说，BN对同一batch内的数据的同一维度做归一化，有多少维度就有多少个均值和方差；LN则是对单个样本的所有维度来做归一化，因此一个batch中就有batch_size个均值和方差。

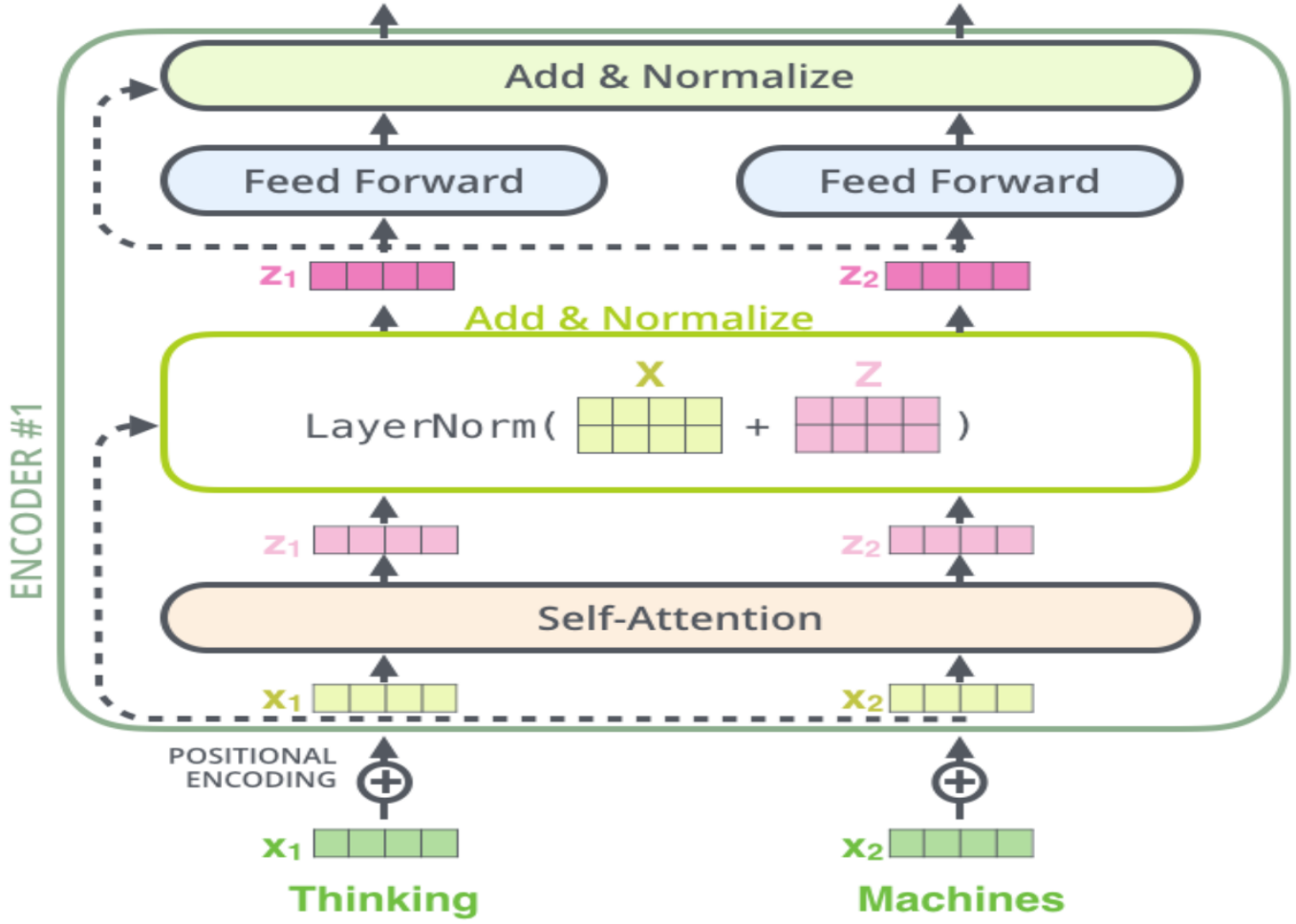
层归一化代码实现

```
class NormLayer(nn.Module):

    def __init__(self, d_model, eps = 1e-6):
        super().__init__()

        self.size = d_model

        # 层归一化包含两个可以学习的参数
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))

        self.eps = eps

    def forward(self, x):
        norm = self.alpha * (x - x.mean(dim=-1, keepdim=True))  \
        / (x.std(dim=-1, keepdim=True) + self.eps) + self.bias
        return norm
```

$$x_i' = \frac{x_i - m}{\sigma} \qquad y_i = \alpha x_i' + \beta$$

Add & Norm

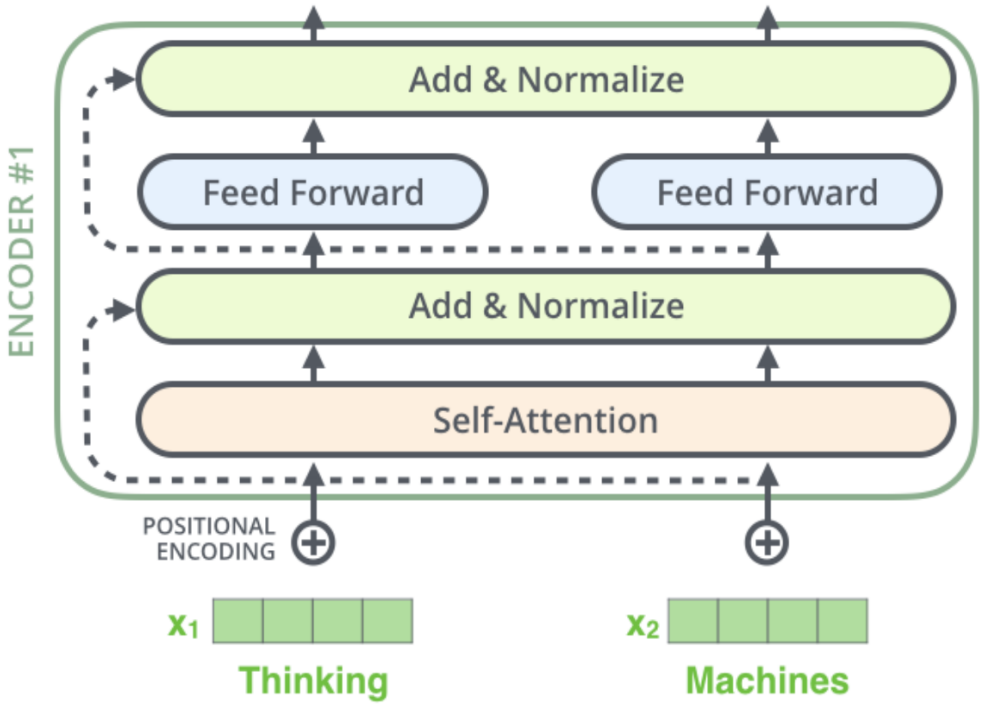
- add & norm操作的矩阵表现形式



前馈层

- 在一层编解码器中，除了注意力层，还有一个前馈层Feed-Forward network (前馈层、全连接层)
- 对于每一个位置，分别进行相同的前馈计算

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



feed_forward：前馈层

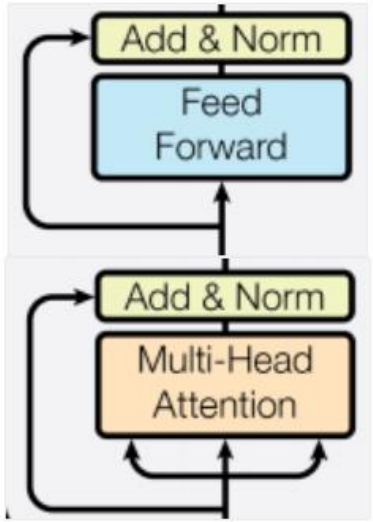
```python
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        '''

        :param d_model: 第一个线性层的输入维度，也是第二个线性层的输出维度
        :param d_ff: 第一个线性层的输出维度，也是第二个线性层的输入维度，默认2048
        :param dropout: 0.1
        '''

        super(FeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

实验结果表明，增大前馈子层隐状态的维度有利于提升翻译结果质量

返回

SublayerConnection：子层连接

```python
class SublayerConnection(nn.Module):
    """..."""
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        # 原paper的方案
        # sublayer_out = sublayer(x)
        # x_norm = self.norm(x + self.dropout(sublayer_out))

        # 稍加调整的版本
        sublayer_out = sublayer(x)
        sublayer_out = self.dropout(sublayer_out)
        x_norm = x + self.norm(sublayer_out)
        return x_norm
```



原文：先残差，再Norm
变体：先Norm，再残差

返回

Encoder block

```python
class EncoderLayer(nn.Module):
    "EncoderLayer is made up of two sublayer: self-attn and feed forward"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size    # embedding's dimension of model, 默认512

    def forward(self, x, mask):
        # attention sub layer
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        # feed forward sub layer
        z = self.sublayer[1](x, self.feed_forward)
        return z
```

自注意力，q,k,v来源都是x

self_attn feed_forward SublayerConnection clones