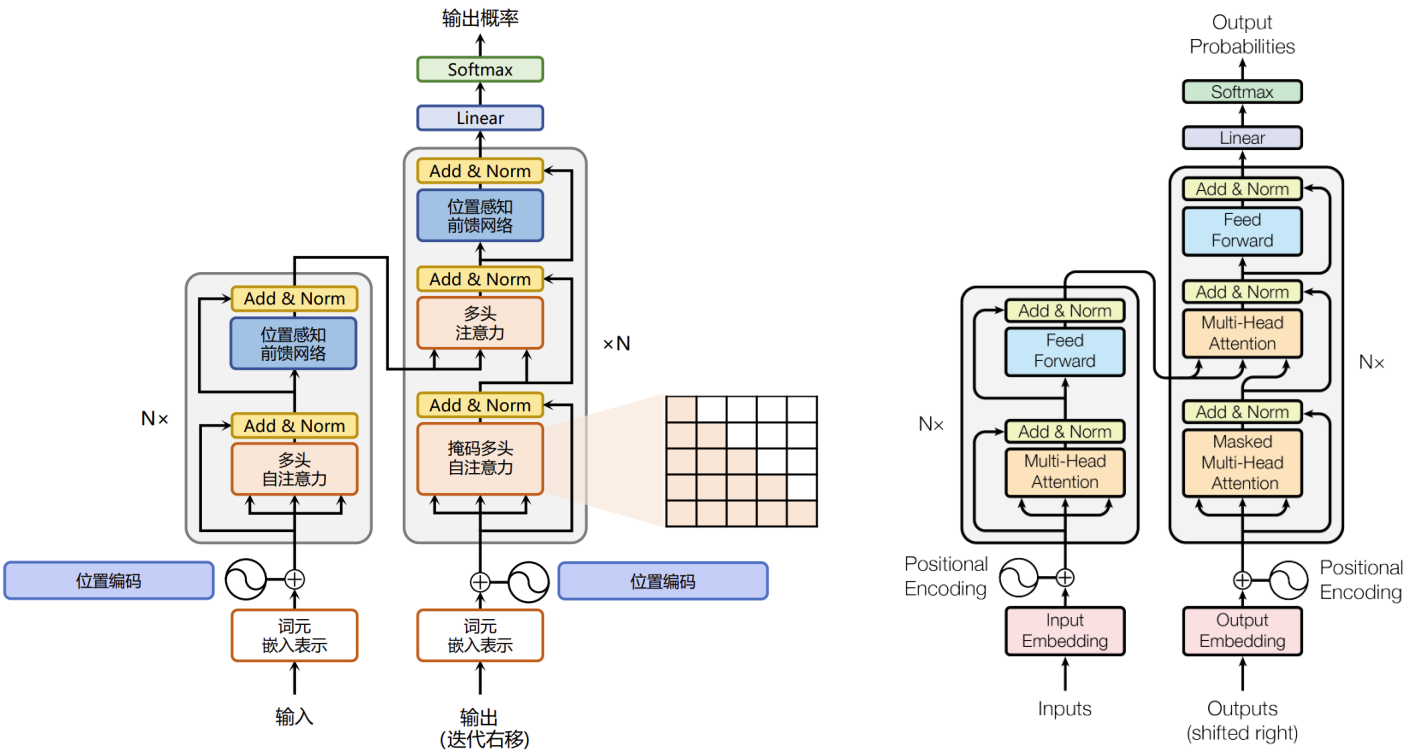
Transformer

Attention Is All You Need.pdf

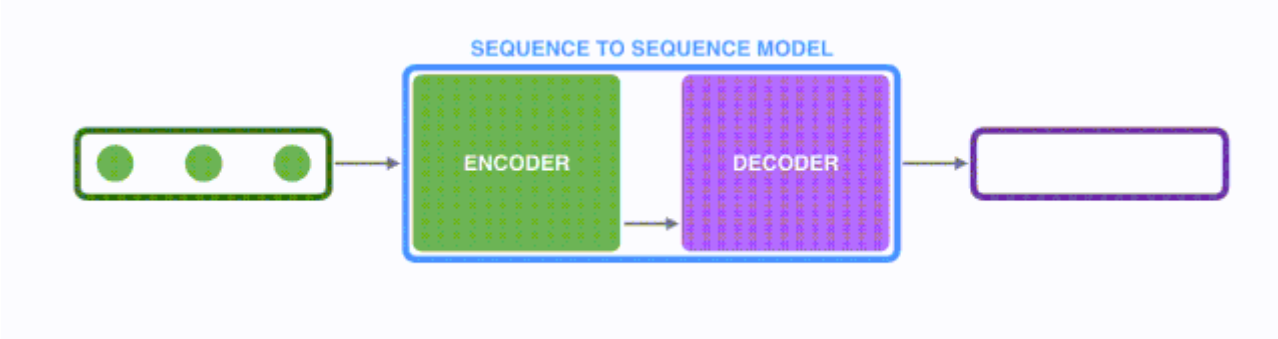


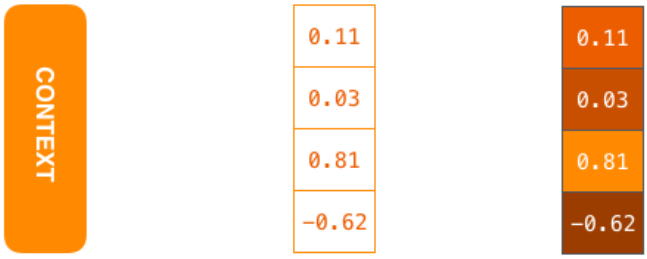
Macrostructure

Seq2Seq

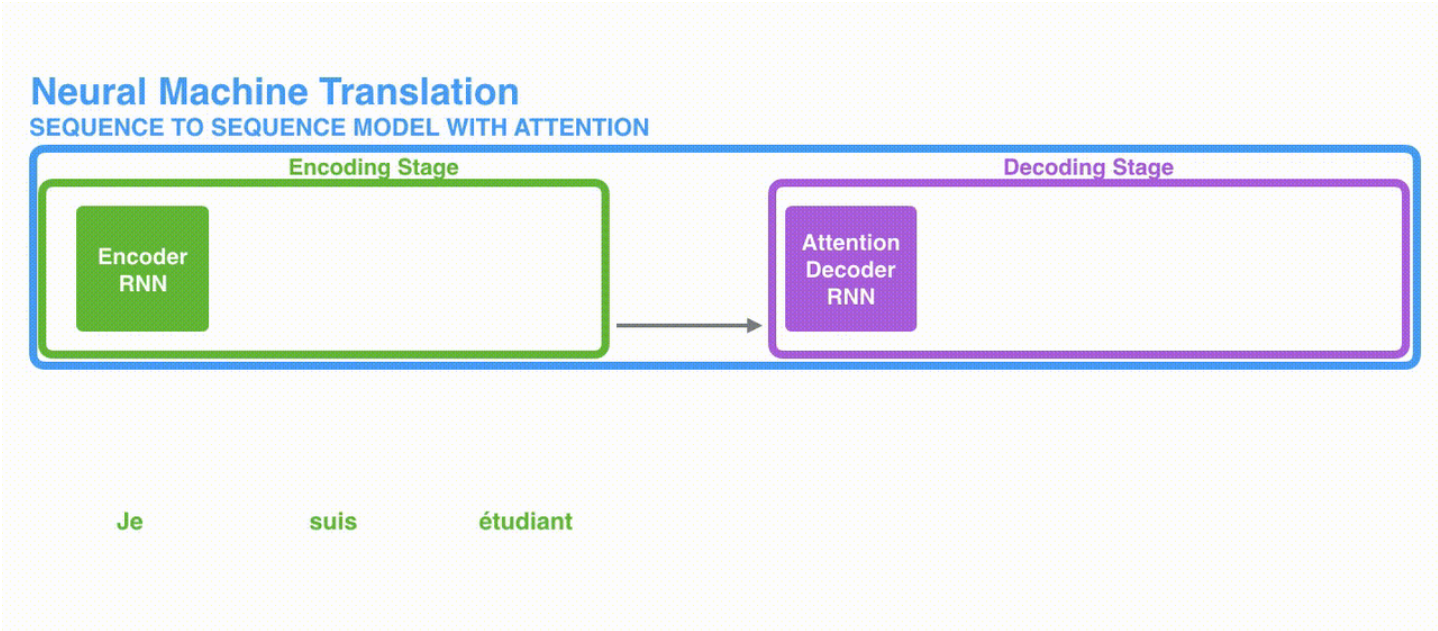
宏观结构上，是Sequence to Sequence框架，即seq2seq

将上图中蓝色的seq2seq模型进行拆解，如下图所示：seq2seq模型由编码器（Encoder）和解码器（Decoder）组成。





context向量对应上图中间浮点数向量。宏观上，理解为语境；微观上，可以理解为每个输出单词所要遵从的模式（在*5 Encoder & Decoder*有进一步阐述）



配合Attention机制，seq2seq不再受限于时序输入时的文本容量限制，更好地捕捉长距离依赖。

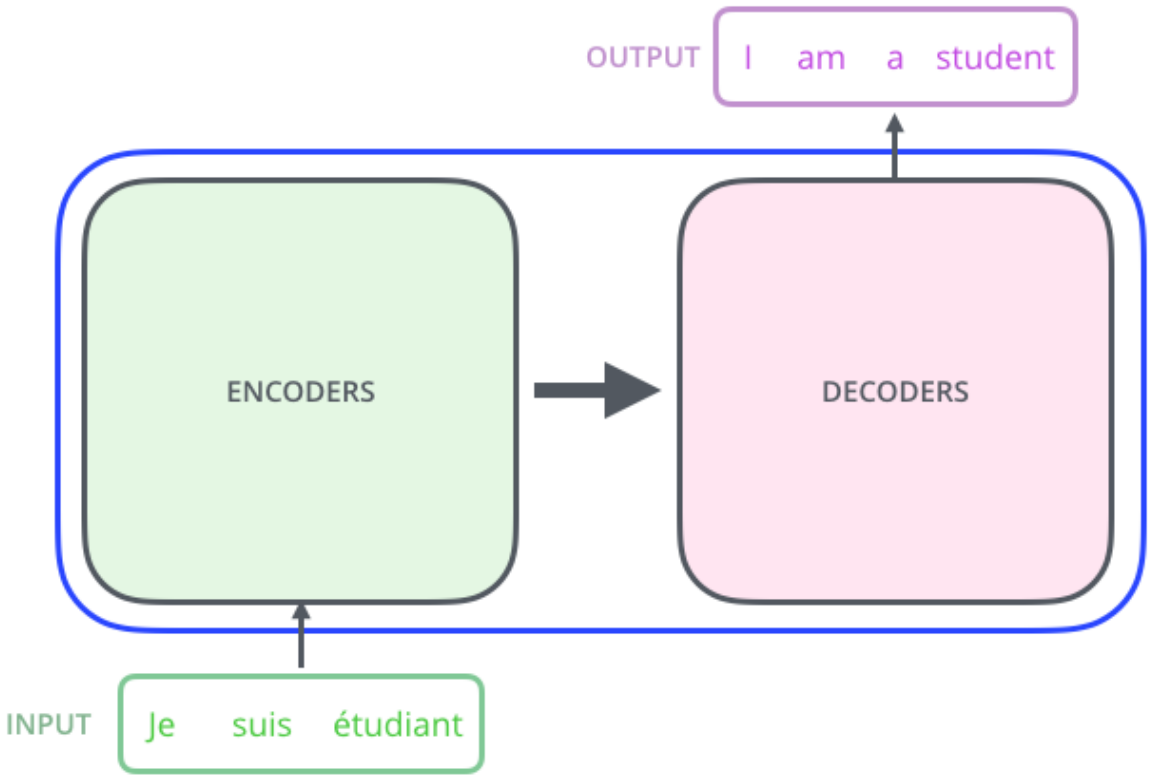
误解: 以上RNN或seq2seq模型思路来理解，可能会让人认为模型只能一个一个词地处理输入输出。

澄清: Transformer的**Self-Attention机制**允许模型在同一时间**并行处理**整个输入序列中的所有词。也就是说，所有的词都可以同时进行相互之间的注意力计算，显著提升计算效率，二不需要像RNN那样顺序处理。

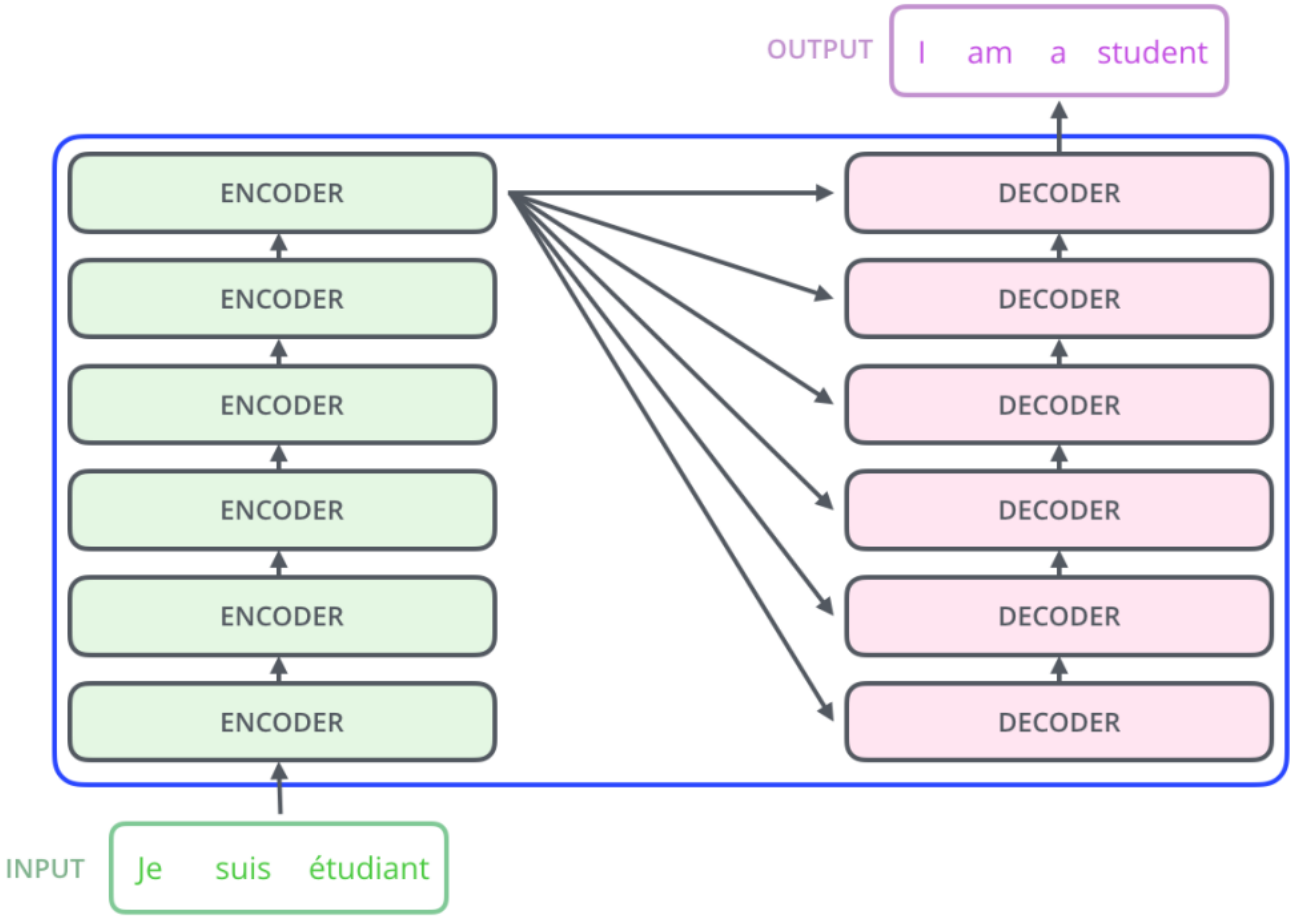
Transformer

Transformer最开始提出来解决机器翻译任务，因此可以看作是seq2seq模型的一种。

将 THE TRANSFORMER 拆开成 seq2seq 标准结构



左边是编码部分Encoders，右边是解码器部分Decoders



Encoders 由多层 Encoder 组成

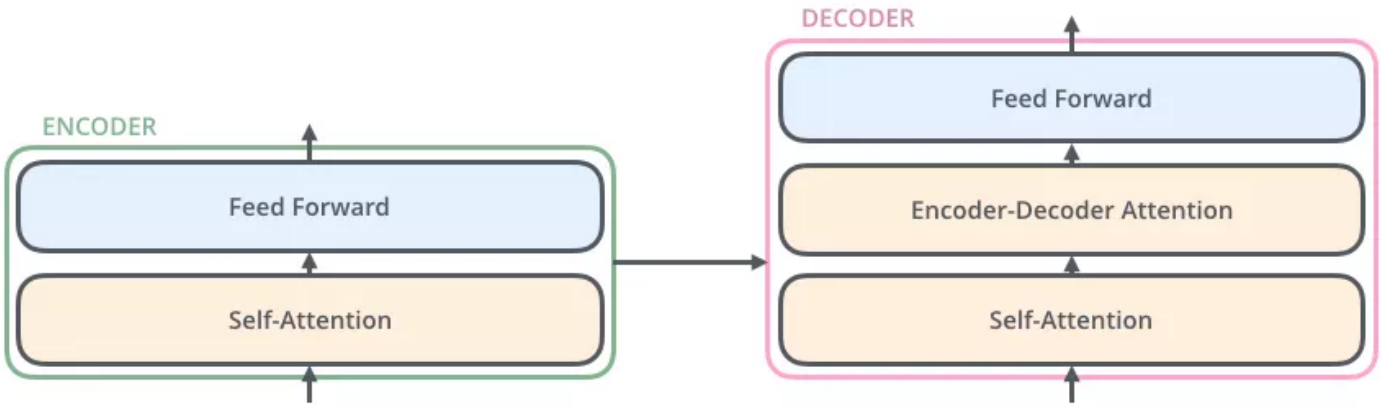
Decoders 也是由多层的 Decoder 组成

（这里的层数6不固定，可以根据实验效果来修改层数 N）

每层 Encoder/Decoder 网络结构是一样的

不同层 Encoder/Decoder 不共享参数

（显然是为了捕捉不同的模式）



单层 Encoder主要由以下两部分组成：

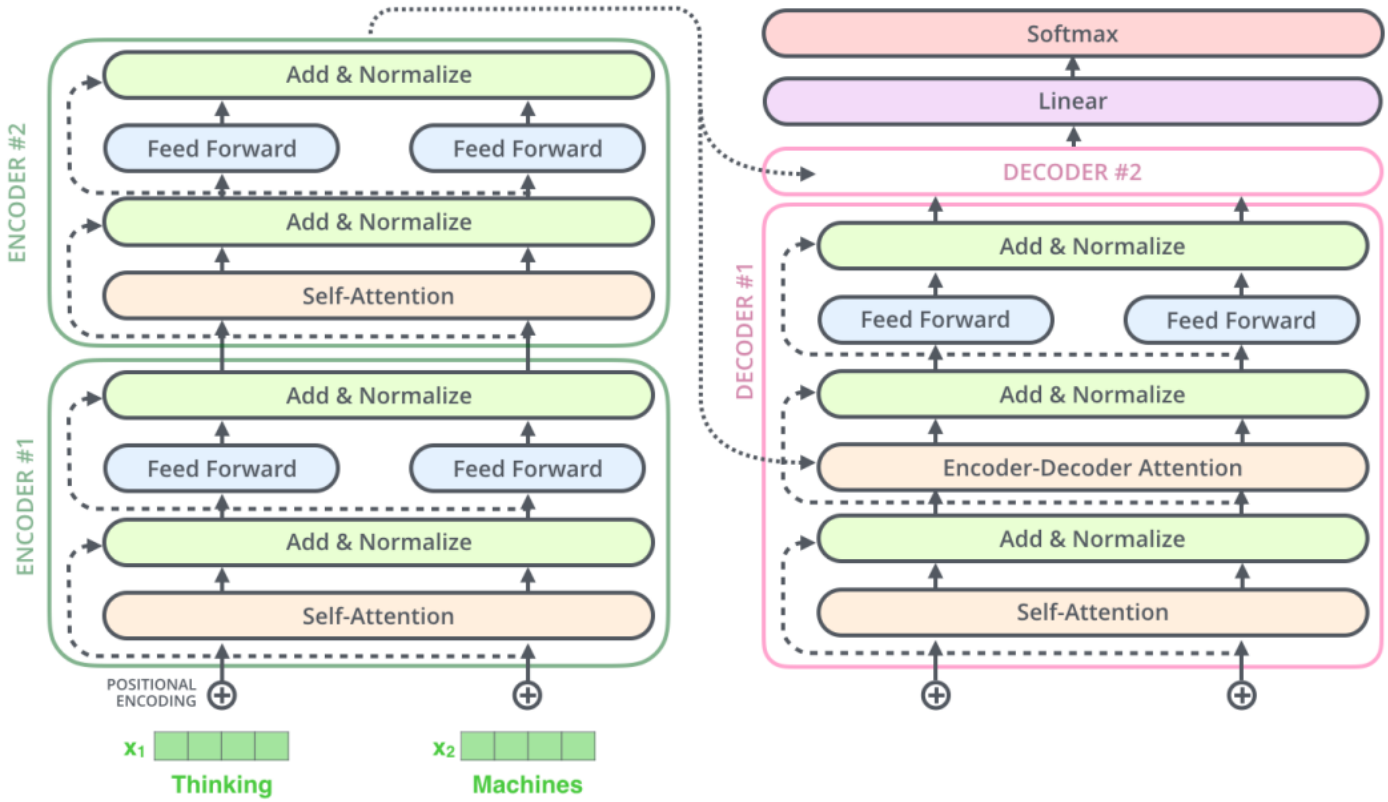
- Self-Attention Layer

- Feed Forward Neural Network

1. Encoder 的输入文本序列 $w_1, w_2, ..., w_n$ 最开始需要经过Embedding转换，得到每个单词向量表示 $x_1, x_2, ..., x_n$ ，其中 $x_i \in \mathbb{R}^d$ 是维度为 d 的向量
2. 然后所有向量经过一个Self-Attention神经网络层进行变换和信息交互得到 $h_1, h_2, ...h_n$ ，其中 $h_i \in \mathbb{R}^d$ 是维度为 d 的向量
3. Self-Attention层的输出会经过前馈神经网络得到新的 $x_1, x_2, .., x_n$ ，依旧是 n 个维度为 d 的向量

这些向量将被送入下一层 Encoder，继续相同的操作。

与编码器对应，Decoder 在编码器的self-attention和FFNN中间插入了一个 Encoder-Decoder Attention 层，这个层帮助 Decoder 聚焦于输入序列最相关的部分。



以上便是Transformer的宏观结构，下面我们开始看宏观结构中的模型细节。

Microstructure

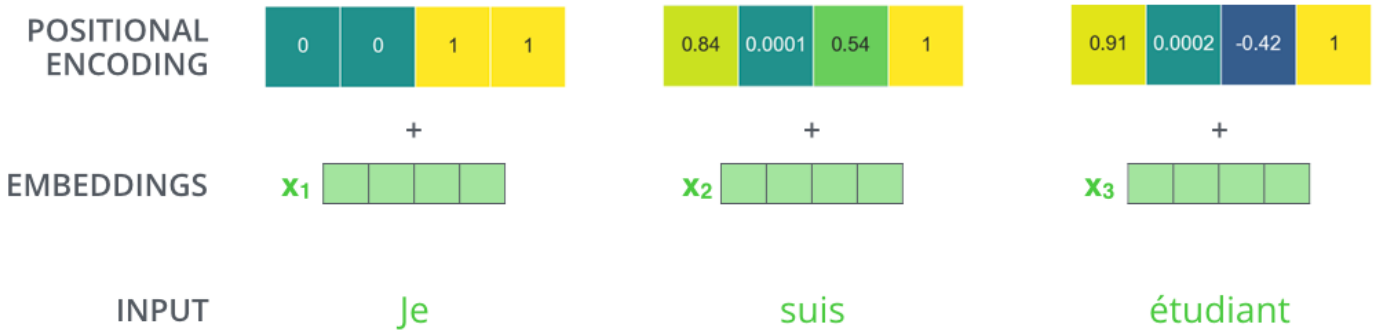
1 Input Embedding

嵌入表示层

嵌入表示层是给文本做预处理的

这里的嵌入 Input Embedding 有两层含义：**一是语义的编码，另一是位置的编码**

二者的**相加**才是完整的嵌入，能使得模型学习到单词的语义信息与位置信息



为什么要相加？

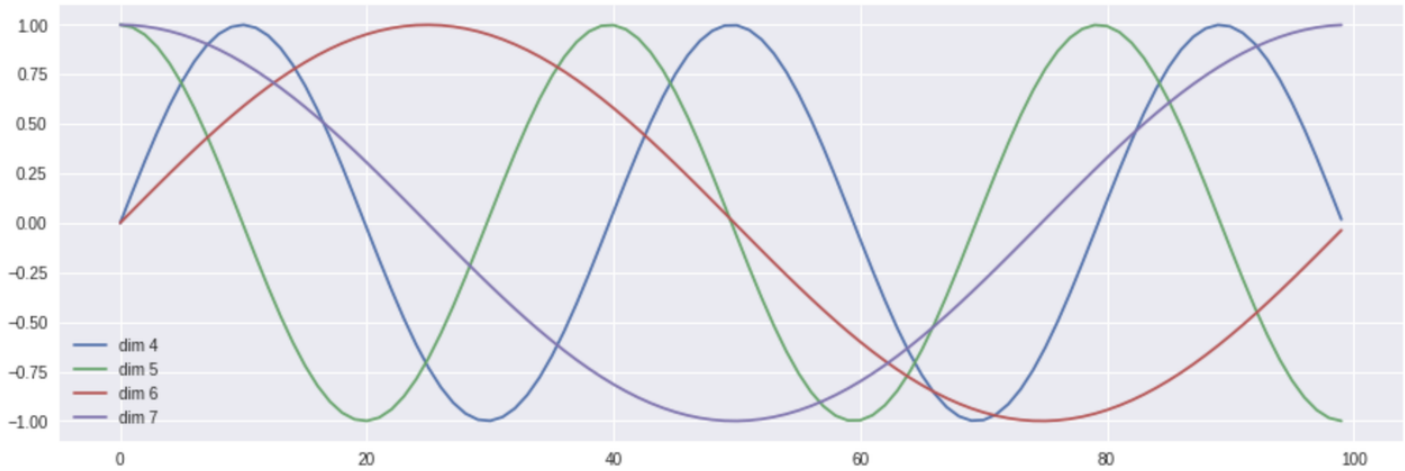
> 为什么要相加呢？能否将位置向量（或语义向量）作为额外的维度？为什么是相加而不是相乘？
>
> 我暂且觉得能这样解释：
>
> 1. 不论是语义还是位置，本质上都是对语言性质的分解，对语言内部逻辑的剖析。在一个句子中，语义对应每个单词的含义，即内容逻辑；位置对应每个单词的顺序，即结构逻辑，或者说语法。所以，二者的性质相同，只不过可能权重不同。
> 2. 作为额外的维度确实能带来更准确的信息，避免了相加的混淆。但显而易见，维度的增加带来的必是计算量的爆炸式增长。

其中，书中着重介绍了位置的编码 Position Encoding (PE)

位置编码使用正弦和余弦函数来生成，公式如下：

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

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



位置编码在0-100位置，在4、5、6、7维的数值图示

维度越大，波动越大，颗粒度越细，区分度越大

这里用例子来解释是非常清晰的：

> **假设的场景**
>
> 我们有一个句子：“I love AI”，这个句子有3个单词。为了简单起见，假设我们用一个长度为4的向量来表示每个单词的位置编码（也就是说，位置编码的维度是4）。在这个例子中，我们将对每个单词的位置计算一个4维的编码向量。
>
> **对“love”的位置编码计算（它在句子中的位置是2，`pos=2`）**
>
> 假设我们要计算"love"在句子中的位置编码。
>
> - **第一维** (PE_1):
>
> $$PE(2,0) = \sin\left(\frac{2}{10000^{\frac{0}{4}}}\right) = \sin(2)$$
>
> - **第二维** (PE_2):
>
> $$PE(2,1) = \cos\left(\frac{2}{10000^{\frac{1}{4}}}\right) = \cos\left(2/\sqrt{10000}\right)$$
>
> - **第三维** (PE_3):
>
> $$PE(2,2) = \sin\left(\frac{2}{10000^{\frac{2}{4}}}\right) = \sin(2/100)$$
>
> - **第四维** (PE_4):

$$PE(2,3) = \cos\left(\frac{2}{10000^{\frac{3}{4}}}\right) = \cos\left(2/1000\right)$$

计算结果

假设经过计算，得到以下值（为了简单，我们用近似值）：

- 第一维：$\sin(2) \approx 0.9093$
- 第二维：$\cos(2/100) \approx 0.9998$
- 第三维：$\sin(2/100) \approx 0.02$
- 第四维：$\cos(2/1000) \approx 0.9999$

所以，"love"这个单词的位置编码向量是：

$$[0.9093,0.9998,0.02,0.9999]$$

逐步解析位置编码的计算公式：

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

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

pos：单词的位置（即句子中第几个单词）。

d：位置编码向量的总维度（即向量的长度，通常和语义向量长度相等）。

i：维度索引，即向量中某个元素的维度编号。

为什么这么用正弦和余弦函数?

1. 周期性和不同频率

正弦和余弦函数是周期函数，**可以在输入不同的频率下生成变化的模式**。

在位置编码中，随着位置 pos 的增加，不同维度的编码值会以不同的频率发生变化，这样可以捕捉到序列中的不同位置信息：

例如，低频维度的编码表示较大范围的变化，高频维度的编码表示较小范围的变化。

全局信息：低频维度可以帮助模型理解句子的全局结构，比如哪一部分是主要信息，哪一部分是次要信息。

局部信息：高频维度可以帮助模型理解局部的详细结构，比如某个特定单词如何与它前后的单词互动，这有助于模型理解细节。

$$[PE(pos,0), PE(pos,1), ..., PE(pos,2i), PE(pos,2i+1)]$$

[全局信息 ----------> 局部信息]

2. 唯一性

正弦和余弦函数的组合可以在每个位置生成唯一的编码。对于任意两个不同的位置 pos_1 和 pos_2 ，它们的编码值会有显著差异，这让模型可以区分不同位置的单词。

3. 平滑的位移关系

由于正弦和余弦函数的连续性和周期性，两个相邻位置的编码在各个维度上是平滑变化的。换句话说，位置 pos 的编码与 $pos+1$ 的编码之间有一个平滑的过渡。这意味着，模型可以轻松地推断出相邻位置之间的关系，进而理解单词之间的相对顺序和距离。

4. 相对位置编码的线性可组合性

由于三角函数的特性，位置编码中不同位置的编码具有线性可组合性。例如，位置 $pos+k$ 的编码可以被表示为位置 pos 的编码的一个线性组合，这表明编码中包含了位置信息的距离。这一特性使得模型能够理解并利用序列中单词之间的相对距离，而不需要显式地计算距离。

为什么分奇偶维度?

1. 增加编码的多样性

通过在不同维度上使用不同的函数（正弦和余弦），可以使每个位置的编码更加多样化。这样即使位置很接近的两个单词，它们的编码向量也会有明显的差异，这有助于模型更好地区分不同位置的单词。

2. 捕捉更丰富的位置信息

正弦和余弦函数的波动方式不同，正弦函数在 0 点起始，余弦函数在 1 点起始，因此它们在不同位置的变化模式也不一样。将这两种函数结合使用，可以捕捉到更多样化的位置信息。

3. 增强模型的对称性

正弦函数和余弦函数本质上是相同频率的两个正交函数。这意味着它们在数学上具有对称性和独立性。通过在奇偶维度上分别使用这两种函数，模型可以从数学上对位置编码进行对称和独立的处理，增强模型的表达能力。

4. 平衡编码的奇偶性

假设我们仅使用一种函数，比如正弦函数，编码向量的所有维度将遵循相同的波动模式，这可能导致在某些情况下，位置信息过于单一，尤其是当位置差别较小的时候。通过引入余弦函数，不同维度上的位置编码在任何位置都能表现出不同的模式，从而避免了编码的单一性。

还有一个点就是，若是只用一个维度，且只用正弦/余弦，若出现相隔周期整数倍的单词则会输出相同的向量；反之则不然。

综上，位置编码的篇幅那么长，其实相对于语义编码，它的权重很小，毕竟范围在正负一之间。在一个语言系统中，对内容上的依赖肯定比结构上多，这是很好理解的。

但是否存在某些语言系统，结构上的逻辑大于内容，比如数学？能否在位置编码函数上加一个振幅参数，来改变依赖内容的语言体系？这里保留这个hint。

Python 代码实现：

```python
import torch
import torch.nn as nn
import math
from torch.autograd import Variable

class PositionalEncoder(nn.Module):
    def __init__(self, d_model, max_seq_len=512):
        """
        初始化位置编码层
        :param d_model: 输入向量的维度（即嵌入维度）
        :param max_seq_len: 序列的最大长度，默认是 512
        """
        super().__init__()
        self.d_model = d_model  # 嵌入的维度

        # 创建一个用于存储位置编码的矩阵 pe，维度是 [max_seq_len, d_model]
        pe = torch.zeros(max_seq_len, d_model)

        # 使用正弦和余弦函数为位置编码矩阵赋值
        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
                # 对偶数索引 i 位置使用正弦函数
                pe[pos, i] = math.sin(pos / (10000 ^ (i / d_model)))
                # 对奇数索引 i+1 位置使用余弦函数
                pe[pos, i+1] = math.cos(pos / (10000 ^ (i / d_model)))

        # 增加一个维度以适应批次处理，pe 的维度变为 [1, max_seq_len, d_model]
        pe = pe.unsqueeze(0)

        # 将位置编码矩阵注册为模型的一个常量，不作为模型参数更新
```

```
31          self.register_buffer('pe', pe)
32
33      def forward(self, x):
34          """
35          前向传播函数，为输入添加位置编码
36          :param x: 输入的词嵌入矩阵，维度 [batch_size, seq_len, d_model]
37          :return: 经过位置编码后的嵌入矩阵
38          """
39          # 通过乘以 sqrt(d_model) 来对输入进行放缩，保持与位置编码的数值规模一致
40          x = x * math.sqrt(self.d_model)
41
42          # 获取输入序列的长度
43          seq_len = x.size(1)
44
45          # 将位置编码加到输入嵌入上，注意这里只加上相应长度的部分
46          x = x + Variable(self.pe[:, :seq_len], requires_grad=False).cuda()
47
48          return x
49
```

2 Multi-Head Attention

注意力层

注意力层是非常核心的一个部分，也是非常有趣的部分！

为什么注意力层核心且有趣？

之所以**重要**是因为，注意力层为文本数据之间建立了关联。
经过嵌入层的文本数据包含了部分的语义信息和位置信息；但不知道某一个单词与别的单词之间的关联度或相似度。此处的关联度可以或者说必须从多个维度来考量，这也是为什么Wq, Wk, Wv 以及之间

的映射有很高的重要性，后面会详细列出我的思考和理解。

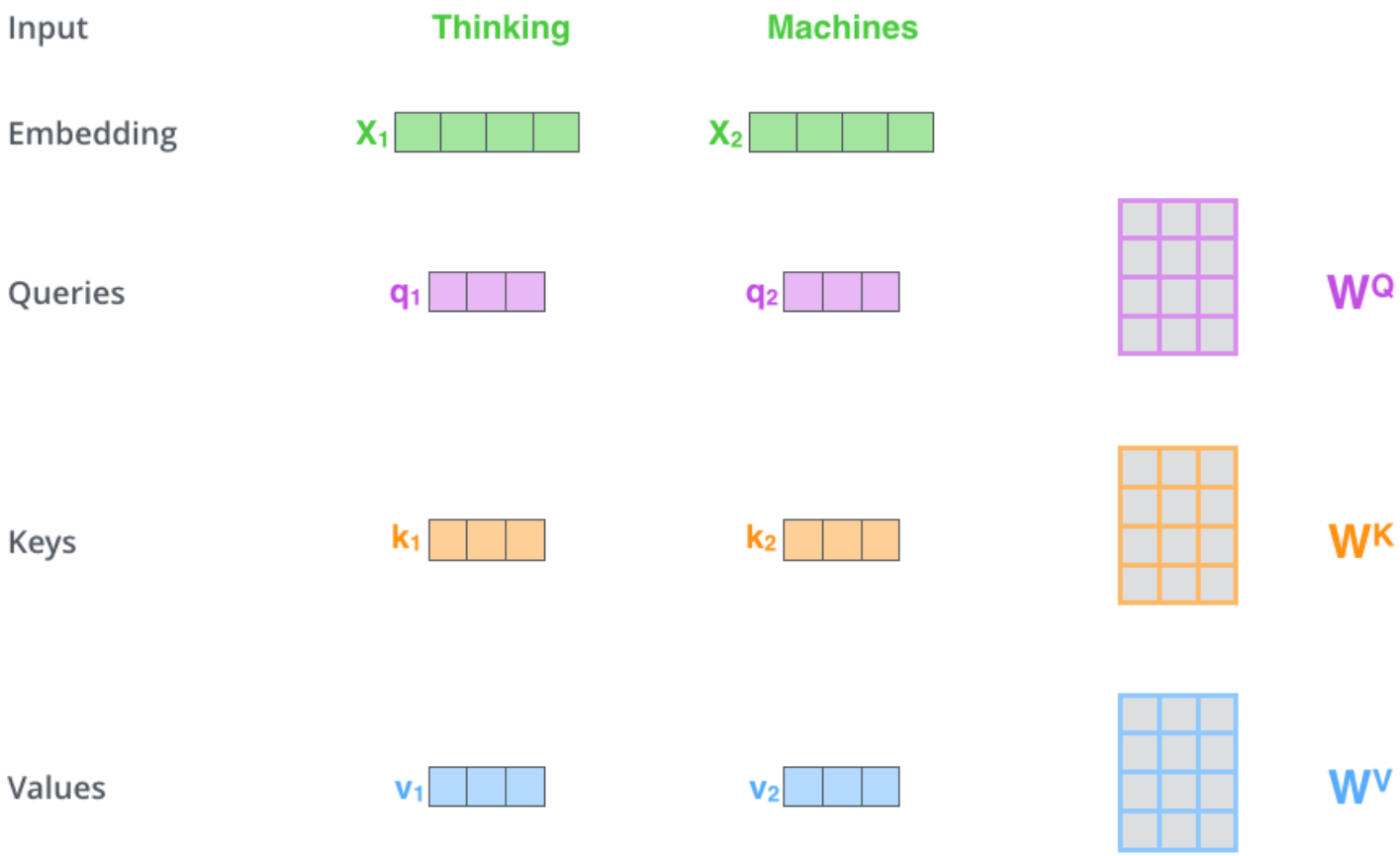
之所以**好玩**是因为，注意力层模拟了人的思考方式。

人，在理解文本的过程中，包括但不限于遵循：潜在寻找的目标——比对特征的相似度——整合信息权重。比如，“吃”这个词，他的潜在目标可能是“人”吃或者吃“苹果”；通过学习“吃苹果”，“人吃”等组合，得到在语境下合理的权重分配；最后整合在一起，在脑子里留下“吃”这个概念。

以下是我浅薄的理解：

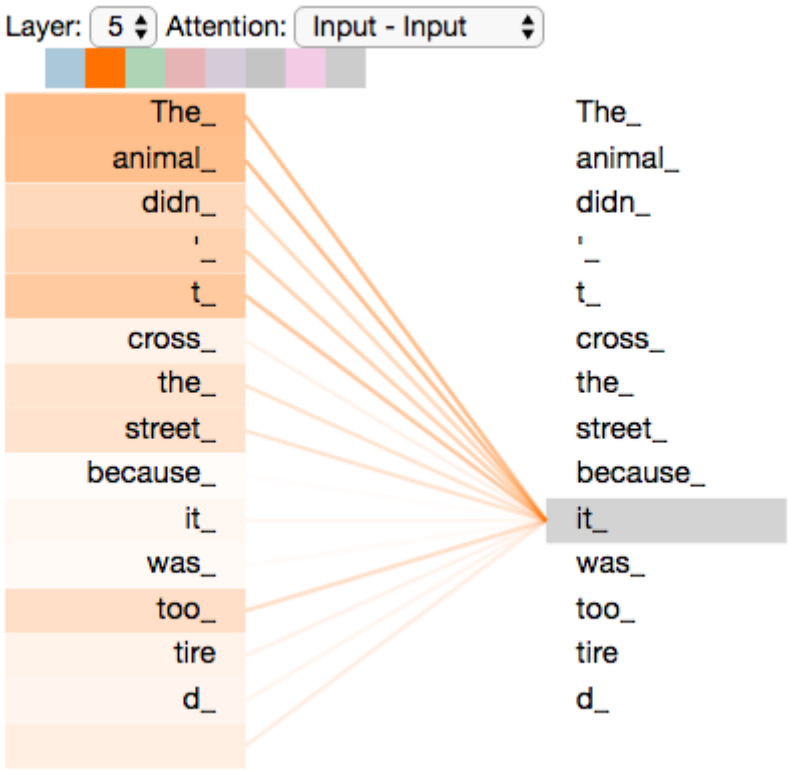
$$\boldsymbol{Z} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d}})\boldsymbol{V}$$

首先，经过嵌入表示层的文本数据是一个维度为d的，语义编码和位置编码叠加的向量，包含了语义信息与权重较小的位置信息。将维度为 $seq_{len} \times d$ 的矩阵传递到自注意力后，会对每个单词，也就是每行矩阵进行进一步的处理，是为了构建单词与其余单词之间的关联，或者说是找到一个更符合当前语境的嵌入方式（不论是语义的维度还是位置等）。初始会有初始化的三个矩阵，分别是 W^Q , W^K , W^V , 分别代表所有单词：潜在寻找的目标，自身的标签特征，以及自身含义（符合人类的认知方式）。通过将每个单词的嵌入向量，经过以上三个矩阵的空间映射，得到线性变换后的特定的模式。



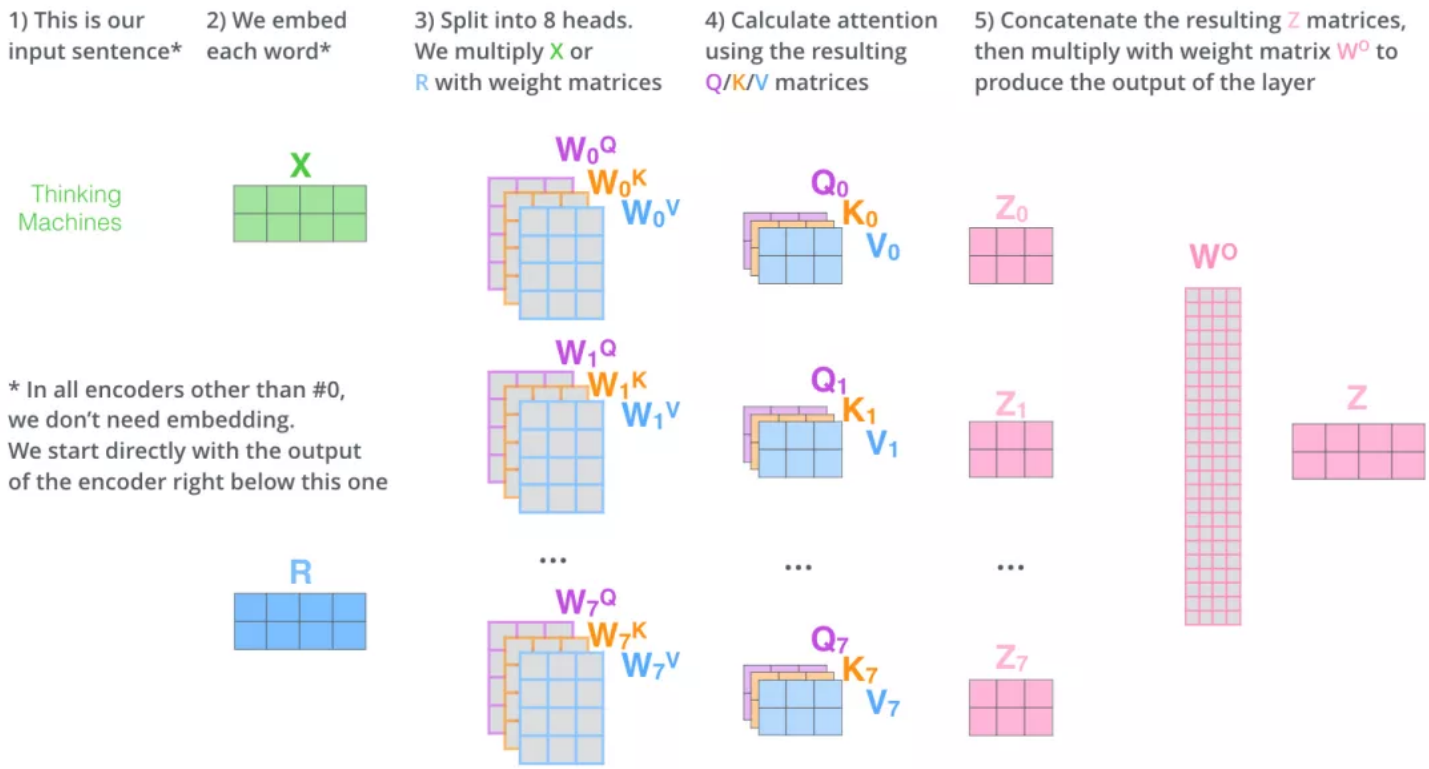
聚焦在一个单词i上，将变换后的qi与其他单词的k进行点乘，得到相似度/分数，如下图可视化。

*与RNN对比一下：RNN 在处理序列中的一个词时，会考虑句子前面的词传过来的 **hidden state**，而 **hidden state** 就包含了前面的词的信息；而 **Self Attention** 机制值得是，当前词会直接关注到自己句子中前后相关的所有词语，如下图 **it** 的例子：



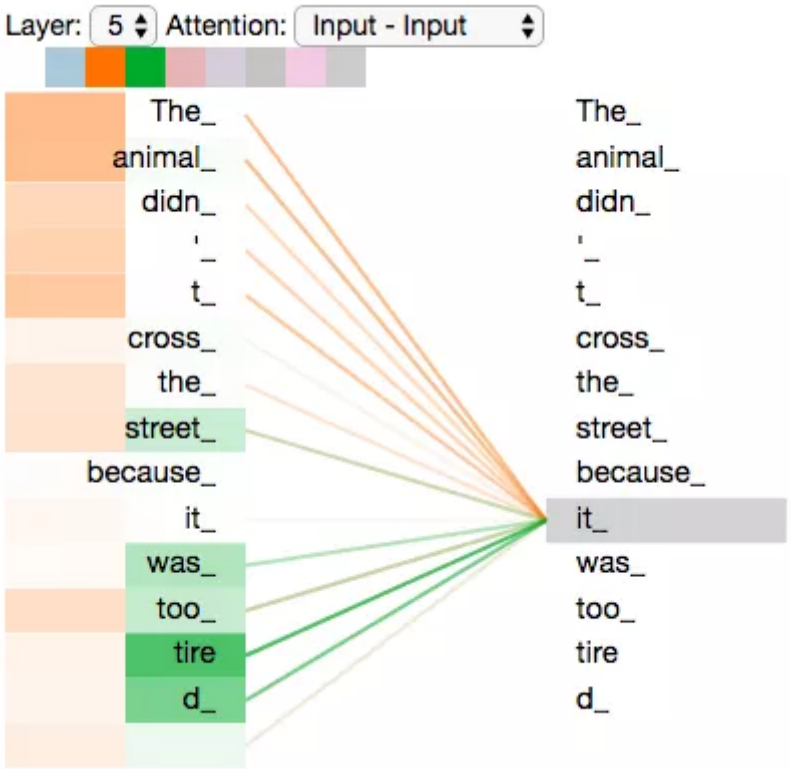
接着通过除以 $\sqrt{d}$，做一个放缩，以免在向后传播时梯度爆炸。再经过Softmax函数，映射为一个概率权重向量；最后与由其他单词的值v构建的值向量，进行点积，得到一个基于这次单词库更有相关性的嵌入向量。

多头自注意力机制，无非是把d分割为多个部分d/h，每个部分理解为不同的映射空间或者学习特征或者训练模式（如图2 不同颜色代表不同的头）。最重要的是多头的训练过程，每个头可以进行**并行计算**，最终在合并维度输出仍是d的向量，或者说是seq_len*d的矩阵。



在橙色的 head 表示 Query 的是主语/物体/上文

绿色的 head 表示 Query 的是宾语/状态/下文



自注意力层的总结

经过嵌入表示层的文本数据是一个维度为 d 的向量，语义编码和位置编码叠加在一起，包含了语义信息与相对较小的位置信息。输入经过自注意力层的处理，目的是构建单词与其他单词之间的关联，或

者说找到一个更符合当前语境的嵌入方式（包括语义、位置、语法等维度）。具体流程如下：

输入表示：维度为 $\text{words\_length} \times d$ 的输入矩阵，其中每行表示一个单词的嵌入向量。

线性变换：将输入矩阵分别乘以三个权重矩阵 W_Q , W_K , W_V ，得到查询向量 Q 、键向量 K 和值向量 V 。

W_Q ：表示潜在的查询目标，用来寻找相关信息。

W_K ：表示单词的特征标签，用来与查询向量匹配。

W_V ：表示单词的含义或信息值。

经过这些线性变换，单词向量被映射到不同的模式空间。

点乘相似度计算：针对某个单词 i ，将其查询向量 q_i 与所有其他单词的键向量 k_j 进行点积，计算其相似度：

$$q_i \cdot k_j$$

这是衡量单词之间相关性的关键步骤。

放缩：为了防止梯度爆炸，点积结果除以放缩因子 $\sqrt{d_k}$ ，得到较为稳定的匹配分数。

Softmax归一化：将放缩后的相似度通过Softmax转换为概率分布，表示每个单词对其他单词的注意力权重。

加权求和：使用注意力权重对值向量 V 进行加权求和，得到该单词的新表示：

$$z_i = \sum_{j=1}^{L} \alpha_{ij} v_j$$

其中 α_{ij} 表示Softmax后第 i 个单词对第 j 个单词的注意力权重。

多头自注意力机制

在多头自注意力机制中，使用多组不同的 W_Q , W_K , W_V 矩阵，将单词嵌入向量映射到不同的模式空间。每个头聚焦于不同的特征（如语义、位置、语法等）。经过多个头的并行处理后，输出向量通过线性变换综合为最终的单词表示。

补充内容

1. **残差连接与层归一化**

Transformer在自注意力计算后引入了残差连接和层归一化来稳定训练过程：

残差连接：将自注意力层的输出与输入直接相加，帮助信息流保持连续性，避免梯度消失。

层归一化：对每层的输出进行归一化，提高模型的收敛速度和稳定性。

公式为：

$$\text{Output} = \text{LayerNorm}(X + \text{Self-Attention}(X))$$

2. **位置编码**

Transformer模型没有内置的顺序意识，因此通过位置编码为输入提供词的相对位置信息。位置编码的设计基于正弦和余弦函数：

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

这使得模型能够在处理不同长度的输入时保持相对位置依赖性。

3. **自回归与双向模型**

自注意力机制可以在**自回归模型**（如GPT）和**双向模型**（如BERT）中应用：

自回归模型：只能根据前面单词预测下一个单词。

双向模型：同时关注句子的前后文，捕捉更完整的上下文信息。

4. **预训练与微调**

自注意力机制支持大规模预训练模型（如BERT、GPT等），先通过预训练学习通用的上下文表示，再通过微调适应具体任务。

5. **计算复杂度**

自注意力机制的计算复杂度为 $O(L^2 \cdot d)$，这对于长序列来说开销较大。近年来有一些改进方案，如：

稀疏注意力：仅计算部分相关性，减少计算量。

局部注意力：仅在局部范围内计算注意力。

6. **可解释性**

自注意力机制相比传统模型具有更高的可解释性。注意力分数显示了模型在做出决策时关注了哪些单词及其重要性，方便研究和分析。

代码：

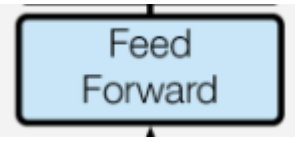
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout=0.1):
        """
        初始化多头注意力层
        :param heads: 多头数量
        :param d_model: 输入向量的维度，即嵌入维度
        :param dropout: dropout 概率，防止过拟合
        """
        super().__init__()

        self.d_model = d_model  # 输入的总维度
        self.d_k = d_model // heads  # 每个头的维度
        self.h = heads  # 多头的数量

        # 线性变换，生成查询（Q）、键（K）和值（V）向量
        self.q_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)

        # dropout层，用于防止过拟合
        self.dropout = nn.Dropout(dropout)

        # 最后的线性变换，输出维度为 d_model
        self.out = nn.Linear(d_model, d_model)

```

```python
    def attention(q, k, v, d_k, mask=None, dropout=None):
        """
        计算单头注意力
        :param q: 查询向量 Q
        :param k: 键向量 K
        :param v: 值向量 V
        :param d_k: 每个头的维度，用于放缩
        :param mask: 掩码，防止在填充的位置上计算注意力
        :param dropout: dropout 概率，防止过拟合
        :return: 注意力计算后的值向量
        """
        # 计算 Q 和 K 的点积，并通过 sqrt(d_k) 放缩
        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)

        # 如果有 mask 掩码，将填充部分的分数设为非常小的值，以便 Softmax 之后接近 0
        if mask is not None:
            mask = mask.unsqueeze(1)
            scores = scores.masked_fill(mask == 0, -1e9)

        # 对分数进行 Softmax 归一化，得到注意力权重
        scores = F.softmax(scores, dim=-1)

        # 如果指定了 dropout，对注意力权重应用 dropout
        if dropout is not None:
            scores = dropout(scores)

        # 将注意力权重与值向量 V 相乘，得到最终的注意力输出
        output = torch.matmul(scores, v)
        return output

    def forward(self, q, k, v, mask=None):
        """
        多头注意力的前向传播函数
        :param q: 查询向量 Q
        :param k: 键向量 K
        :param v: 值向量 V
        :param mask: 掩码，用于避免填充部分的干扰
        :return: 最终的多头注意力输出
        """
        bs = q.size(0)  # 获取批次大小

        # 线性变换，并将结果 reshape 成 [batch_size, seq_len, heads, d_k] 的形式
        k = self.k_linear(k).view(bs, -1, self.h, self.d_k)
        q = self.q_linear(q).view(bs, -1, self.h, self.d_k)
        v = self.v_linear(v).view(bs, -1, self.h, self.d_k)

```

```
        # 将第二个和第三个维度进行交换，使得维度变为 [batch_size, heads, seq_len, d_k]
        k = k.transpose(1, 2)
        q = q.transpose(1, 2)
        v = v.transpose(1, 2)

        # 调用 attention 函数，计算多头的注意力结果
        scores = attention(q, k, v, self.d_k, mask, self.dropout)

        # 将多个头的输出拼接回去，将维度变回 [batch_size, seq_len, d_model]
        concat = scores.transpose(1, 2).contiguous().view(bs, -1, self.d_model)

        # 通过线性层生成最终的输出
        output = self.out(concat)

        return output

```

3 Feed Forward

前馈层



为什么要前馈层?

Transformer中的前馈层设计似乎并没有一个非常深刻的理论基础，给我的感觉更多是经验和实验优化的结果。

本质上这就是个炼丹，不过我还是努力尝试解释一下合理性。

这一层可以用以下表达式来展示：

$$FFN(x) = ReLU(xW1 + b1)W2 + b2$$

可见，该层完成了：线性变化 + 非线性变化 + 线性变化

1. 输入 x 先经过一个线性变换 $xW_1 + b_1$ ，通常将维度从 d 扩展到更大的维度 d_{ffn}
   比如：def __init__(self, d_model, d_ff=2048, dropout=0.1)
2. 通过 ReLU 激活函数引入非线性特征： $\mathrm{ReLU}(xW_1 + b_1)$ 。
3. 经过第二个线性变换 $(\mathrm{ReLU}(xW_1 + b_1))W_2 + b_2$ ，将维度从 d_{ffn} 缩回到原始维度 d 。
4. 输出该结果作为下一层的输入。

显然，该层的核心是其中ReLU层，也就是非线性变换。这种设计很类似多层感知机，具备**通用逼近能力**，核心就是因为引入了非线性变换能增强模型表达。

在炼丹界大量的经验证明了引入非线性确实能够帮助模型捕捉更复杂的特征。在更高的维度，人类无法理解的或者可能还没理解的层次，存在一种规律，模式，因而需要非线性变换来实现跨越。

注意！在 Transformer 中，前馈网络（FFN）对序列的每个词进行独立处理，称为**逐词应用**。

逐词应用意味着前馈网络在 Transformer 中只对序列中的每个词进行**独立处理**，并不会将整个序列一起送入 FFN 中处理。

与自注意力机制不同，FFN 不会捕捉词与词之间的依赖关系，它仅对词的局部特征进行处理。

通过这种设计，模型在自注意力机制处理全局依赖的同时，FFN 能够独立优化每个词的表示。

关键点：

- **非线性变换的作用**：ReLU激活函数引入了非线性，能够帮助模型捕捉更复杂的特征。
- **维度扩展的作用**：实验表明，增大前馈层隐状态的维度（即 d_{ffn} ）有助于提高模型的性能。通常，前馈层的维度 d_{ffn} 要比自注意力层的维度大，这能够增强模型的泛化能力和翻译结果的质量。

代码：

```python
import torch.nn as nn
import torch.nn.functional as F

class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff=2048, dropout=0.1):
        """
        初始化前馈网络层的构造函数。

        参数:
        - d_model (int): 输入和输出向量的维度。
        - d_ff (int): 前馈层内部，即第一个全连接层的输出维度，默认为2048。
```

```
        - dropout (float): 在ReLU激活后应用的dropout概率，用于防止过拟合，默认为0.1。
        """
        super().__init__()  # 初始化nn.Module的基类，为继承自nn.Module的子类提供初始化。

        # 第一个全连接层，将输入维度从d_model增加到d_ff。
        self.linear_1 = nn.Linear(d_model, d_ff)

        # Dropout层，用于在训练过程中随机丢弃一部分神经元，防止过拟合。
        self.dropout = nn.Dropout(dropout)

        # 第二个全连接层，将维度从d_ff降回到d_model。
        self.linear_2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        """
        定义前馈网络的前向传播逻辑。

        参数:
        - x (Tensor): 输入张量，维度为 [batch_size, seq_length, d_model]。

        返回:
        - x (Tensor): 经过前馈网络处理后的输出张量，维度不变。
        """
        # 通过第一个全连接层后，应用ReLU激活函数和dropout。
        x = self.dropout(F.relu(self.linear_1(x)))

        # 通过第二个全连接层得到最终输出。
        x = self.linear_2(x)

        return x

```

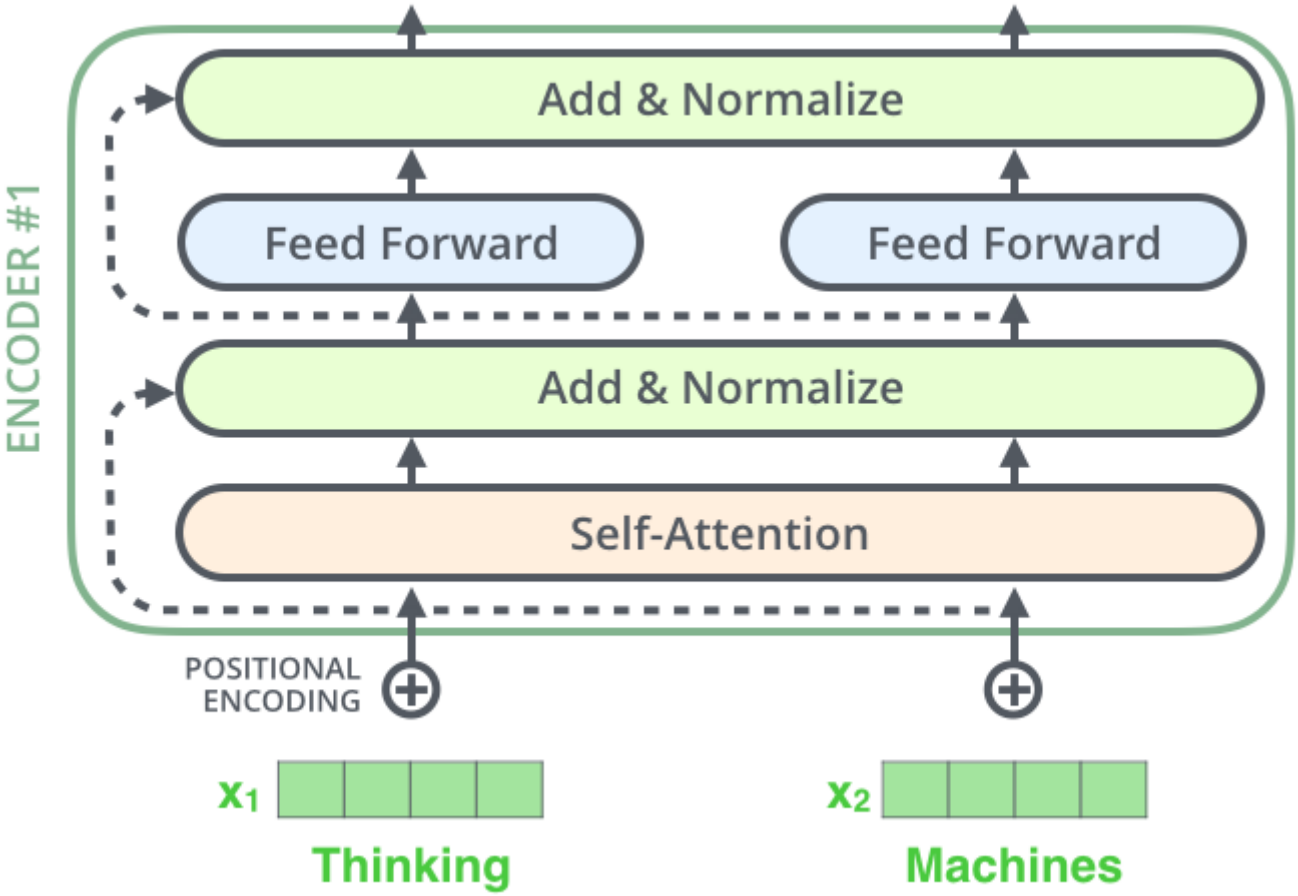
4 Add & Norm

残差连接与层归一化

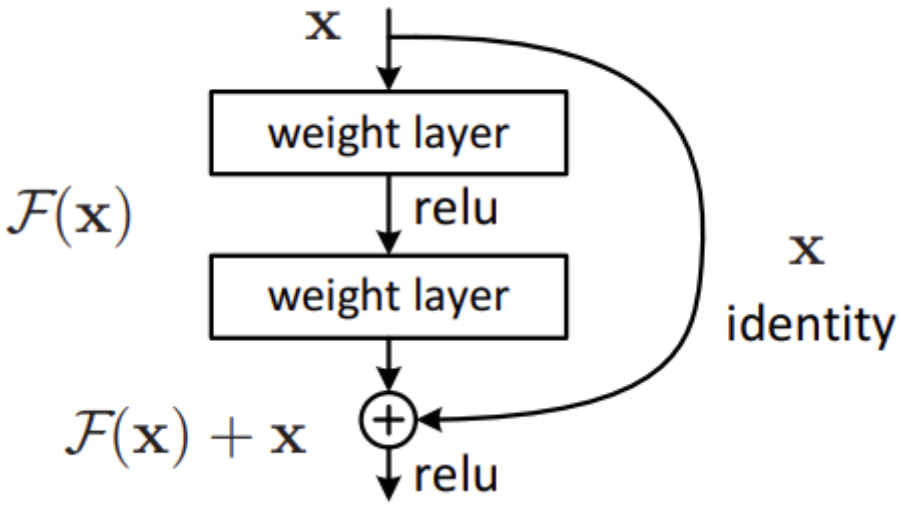
为什么要残差?

~~就是屌~~

残差连接让网络能够学习残差（即输入与期望输出之间的差值），而不是学习直接映射。对于深度网络来说，学习残差通常比学习直接映射更容易。



Deep Residual Learning for Image Recognition.pdf
755.14KB



$$x^{(l+1)} = f(x^{(l)}) + x^{(l)}$$

其中：

- $x^{(l)}$ 是第 l 层的输入；

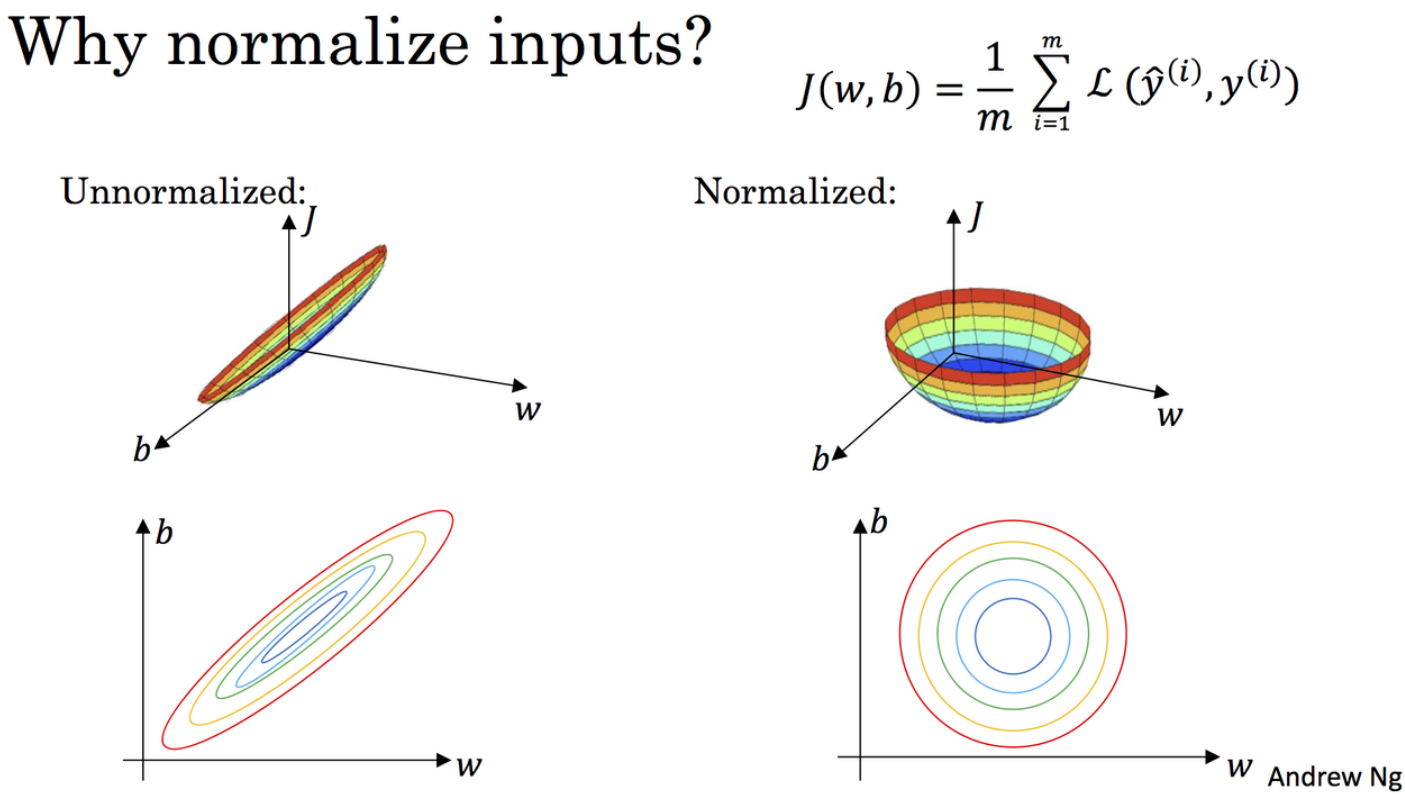
- $f(\cdot)$ 是该层的映射函数（如多头自注意力或前馈网络）。

为什么要归一化?

层归一化通过使每个词嵌入的各个维度在经过激活函数之前具有相同的分布（即零均值和单位方差），帮助稳定训练过程。具体来说，层归一化解决了**内层协变量转移（Internal Covariate Shift）**的问题，这种问题会导致每一层的输入分布在训练过程中不断变化，进而影响梯度更新的效率。

所以在**每个需要训练参数的子层后**，都有一个归一化。

我觉得这张图是最好的解释：



$$\mathrm{LN}(x) = \alpha \cdot \frac{x - \mu}{\sigma} + b$$

其中：

- mu 是输入 x 的均值，计算方式为：

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

- $sigma$ 是输入的标准差，计算方式为：

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

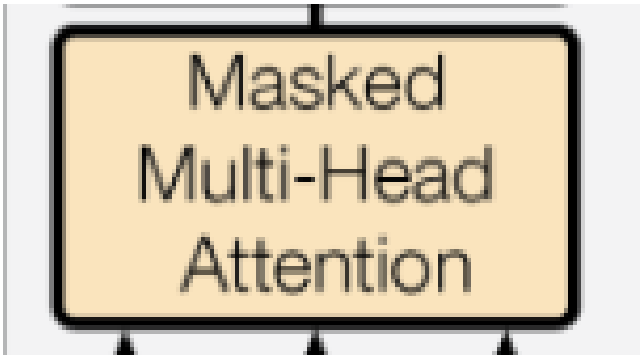
- $alpha$ 和 b 是可学习的参数，用于对归一化后的输入进行缩放和平移。

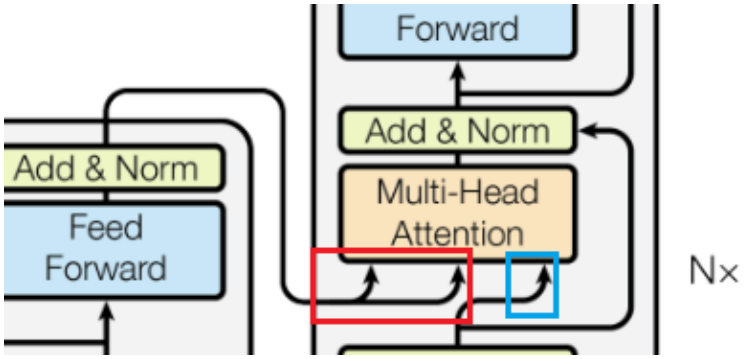
代码：

```python
import torch
import torch.nn as nn

class NormLayer(nn.Module):
    def __init__(self, d_model, eps=1e-6):
        super(NormLayer, self).__init__()
        self.size = d_model
        # 层归一化包含两个可以学习的参数：缩放参数 alpha 和 偏移参数 bias
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))
        self.eps = eps

    def forward(self, x):
        # 计算每个输入的均值和标准差
        mean = x.mean(dim=-1, keepdim=True)
        std = x.std(dim=-1, keepdim=True)
        # 归一化输入，并应用缩放和偏移
        norm = self.alpha * (x - mean) / (std + self.eps) + self.bias
        return norm
```

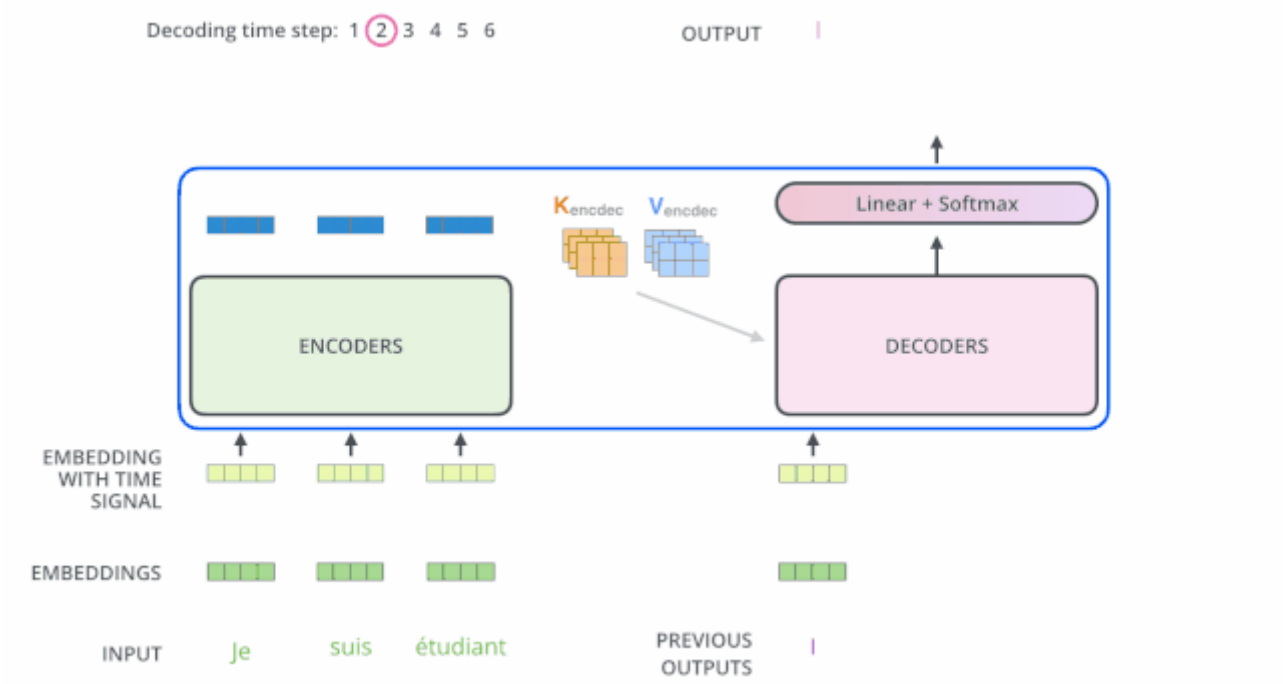
5 Encoder & Decoder

编码器和解码器结构





解码器相比编码器，在多头自注意力机制上多了**掩码机制**，以及在用了**交叉注意力机制**。



为什么要掩码机制?

掩码机制，是为了在生成目标语言过程中，“不偷看”后面的单词；换句话说，**不依赖于未知的关联**，也就是避免牛头不对马嘴。从而，能向上提供上下文合理的Query。

其本质上是在进行评分计算时加了一个掩码矩阵：

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

其中，M 是掩码矩阵，当掩盖某些位置时，该位置的值被设置为负无穷（例如，$-\infty$），使得这些位置在softmax计算时被忽略。

$$M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

假设我们的目标语言序列是：

```
["我", "喜欢", "学习", "编程"]
```

每个词已经被嵌入成一个维度为3的向量，假设嵌入后的向量如下：

```
我: [0.5, 0.1, 0.3]
喜欢: [0.7, 0.2, 0.9]
学习: [0.6, 0.4, 0.8]
编程: [0.8, 0.3, 0.5]
```

我们将输入表示成一个矩阵 X：

$$X = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.7 & 0.2 & 0.9 \\ 0.6 & 0.4 & 0.8 \\ 0.8 & 0.3 & 0.5 \end{bmatrix}$$

1. 计算查询（Query）、键（Key）和值（Value）

假设我们使用相同的投影矩阵来计算查询、键和值，分别为 W_Q , W_K , W_V ：

$$W_Q = W_V = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$W_K = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.2 & 0.7 & 0.1 \\ 0.3 & 0.1 & 0.6 \end{bmatrix}$$

查询向量（Query）：

$$Q = X \times W_Q = X = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.7 & 0.2 & 0.9 \\ 0.6 & 0.4 & 0.8 \\ 0.8 & 0.3 & 0.5 \end{bmatrix}$$

键向量（Key）：

$$K = X \times W_K = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.7 & 0.2 & 0.9 \\ 0.6 & 0.4 & 0.8 \\ 0.8 & 0.3 & 0.5 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.2 & 0.7 & 0.1 \\ 0.3 & 0.1 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.32 & 0.11 & 0.36 \\ 0.56 & 0.2 & 0.63 \\ 0.47 & 0.19 & 0.54 \\ 0.55 & 0.18 & 0.59 \end{bmatrix}$$

值向量（Value）：

$$V = X \times W_V = X = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.7 & 0.2 & 0.9 \\ 0.6 & 0.4 & 0.8 \\ 0.8 & 0.3 & 0.5 \end{bmatrix}$$

2. 计算注意力分数

现在我们计算查询和键的点积：

$$QK^T = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.7 & 0.2 & 0.9 \\ 0.6 & 0.4 & 0.8 \\ 0.8 & 0.3 & 0.5 \end{bmatrix} \times \begin{bmatrix} 0.32 & 0.56 & 0.47 & 0.55 \\ 0.11 & 0.2 & 0.19 & 0.18 \\ 0.36 & 0.63 & 0.54 & 0.59 \end{bmatrix}$$

$$= \begin{bmatrix} 0.214 & 0.392 & 0.337 & 0.372 \\ 0.371 & 0.683 & 0.581 & 0.65 \\ 0.334 & 0.612 & 0.528 & 0.592 \\ 0.335 & 0.619 & 0.545 & 0.615 \end{bmatrix}$$

然后将这些分数除以 $\sqrt{d_k} = \sqrt{3}$ 进行缩放：

$$\frac{QK^T}{\sqrt{3}} = \begin{bmatrix} 0.1235 & 0.2264 & 0.1946 & 0.2148 \\ 0.2142 & 0.3939 & 0.3354 & 0.3755 \\ 0.1927 & 0.3533 & 0.3047 & 0.3418 \\ 0.1934 & 0.3574 & 0.3146 & 0.3551 \end{bmatrix}$$

3. 应用掩码

（这里和代码有点出入，代码中定义的是不包括对角线的下三角为掩码部分）

在掩码自注意力中，我们要引入掩码矩阵 M，用于屏蔽未来的单词。对于四个词的输入，掩码矩阵如下：

$$M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

将掩码矩阵加到注意力分数中：

$$\frac{QK^T}{\sqrt{3}} + M = \begin{bmatrix} 0.1235 & -\infty & -\infty & -\infty \\ 0.2142 & 0.3939 & -\infty & -\infty \\ 0.1927 & 0.3533 & 0.3047 & -\infty \\ 0.1934 & 0.3574 & 0.3146 & 0.3551 \end{bmatrix}$$

0.1235 对应第一个单词“我”与第一个单词“我”的评分

0.2142 和 0.3939 对应“喜欢” “我”和“喜欢” “喜欢”的评分

以此类推...

4. 计算 softmax

对屏蔽后的分数进行 softmax 操作，假设 softmax 结果如下：

$$\text{softmax}\left(\begin{bmatrix} 0.1235 & -\infty & -\infty & -\infty \\ 0.2142 & 0.3939 & -\infty & -\infty \\ 0.1927 & 0.3533 & 0.3047 & -\infty \\ 0.1934 & 0.3574 & 0.3146 & 0.3551 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.445 & 0.555 & 0 & 0 \\ 0.34 & 0.378 & 0.282 & 0 \\ 0.25 & 0.298 & 0.236 & 0.216 \end{bmatrix}$$

5. 计算最终输出

将 softmax 的结果与值向量 V 加权求和，计算最终的输出：

$$\text{Output} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.445 & 0.555 & 0 & 0 \\ 0.34 & 0.378 & 0.282 & 0 \\ 0.25 & 0.298 & 0.236 & 0.216 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.7 & 0.2 & 0.9 \\ 0.6 & 0.4 & 0.8 \\ 0.8 & 0.3 & 0.5 \end{bmatrix}$$

结果为：

$$\text{Output} = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.61 & 0.16 & 0.675 \\ 0.6022 & 0.2344 & 0.6944 \\ 0.622 & 0.268 & 0.656 \end{bmatrix}$$

为什么要交叉注意力?

既然掩码自注意力子层向上提供了符合上下文的Query，即提出潜在目标值；况且，在不知道后文的情况下，只能依赖源序列提供的特征Key，与Query进行匹配评分，得出加权值。

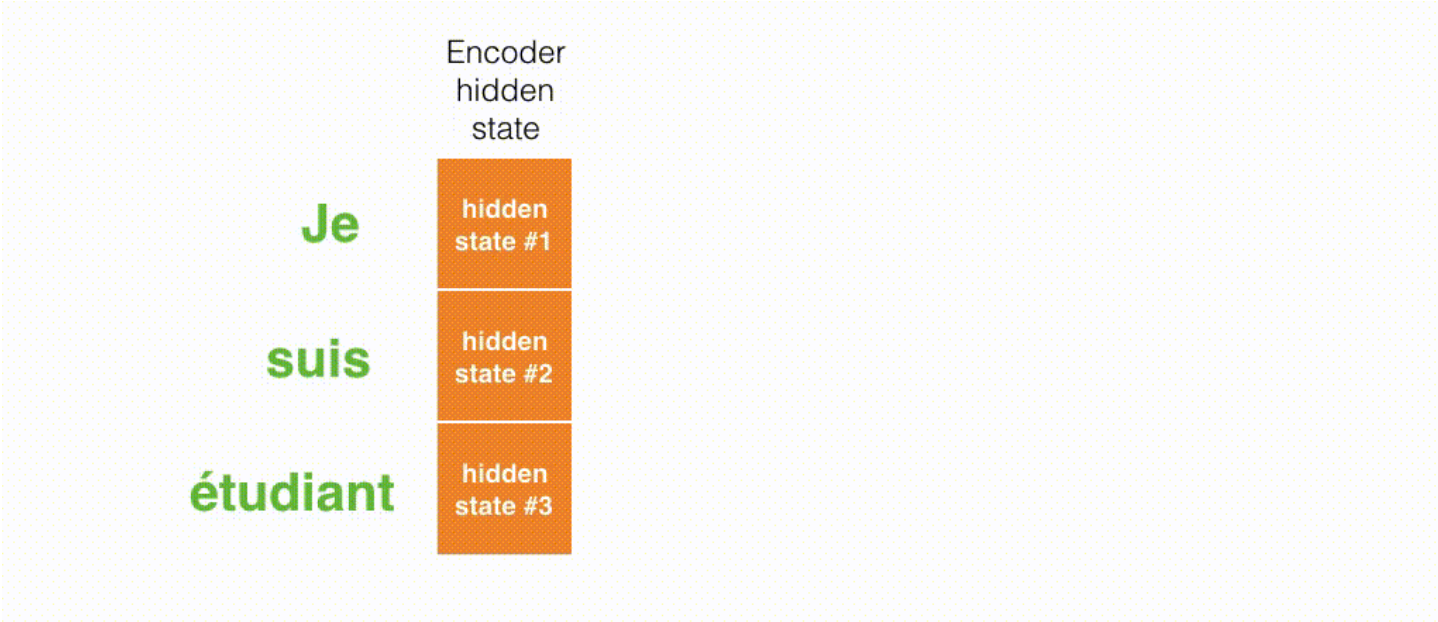
$$\text{Attention}(Q_{decoder}, K_{encoder}, V_{encoder})$$

$$= \text{Softmax}\left(\frac{Q_{decoder} K_{encoder}^T}{\sqrt{d_k}}\right) V_{encoder}$$

- $Q_{decoder}$ ： 解码器的查询矩阵，基于解码器前一层的输出。
- $K_{encoder}$ 和 $V_{encoder}$ ： 分别是编码器的键和值矩阵，基于编码器的输出。

- $\frac{1}{\sqrt{d_k}}$：缩放因子，d_k 是键的维度，用于防止点积值过大导致梯度消失问题。
- $\mathrm{softmax}$：用于计算查询与每个键的相似度权重。

最终结果是将这些权重应用于值矩阵，从而生成融合了源语言信息的向量。

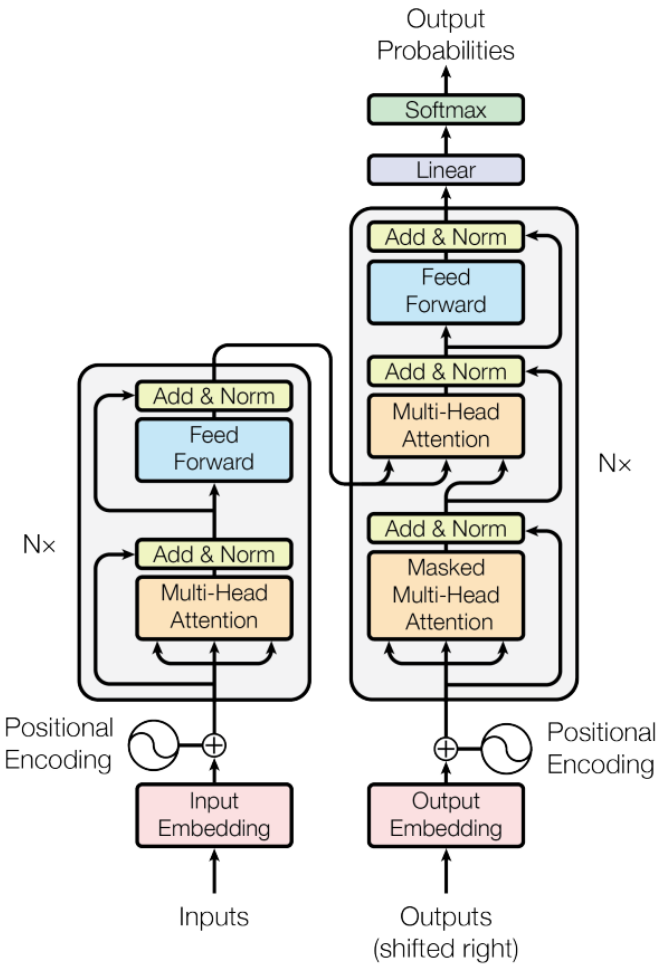


hidden state

- 块：表示目标单词与源序列的点积分数
- 颜色：深浅表示分数高低，或权重大小
- 列：表示加权求和

6 Conclusion & Reflection

经过第一轮对Transformer的学习，了解了整体的框架以及每一层的作用。下面用一个贯穿架构的例子展示我的理解。



假设我们要翻译一句话：“The cat is on the mat.”

目标是将其翻译成法语：“Le chat est sur le tapis.”

1. 输入处理

1.1 词嵌入（Word Embedding）

首先，输入的每个单词通过嵌入矩阵被转换为一个 d_{model} 维的向量。

令嵌入矩阵为 W_{embed}，其尺寸为 $Seq_{len} \times d_{model}$，其中 Seq_{len} 是词汇表大小，d_{model} 是每个单词嵌入向量的维度。嵌入操作可以表示为：

$$\mathbf{e}_{\text{word}} = W_{embed}[\text{word}]$$

例如，输入句子中的单词 "The" 被表示为向量 $\mathbf{e}_{\text{The}}$，"cat" 被表示为 $\mathbf{e}_{\text{cat}}$，依此类推。因此整个句子的嵌入表示为：

$$X = [\mathbf{e}_{\text{The}}, \mathbf{e}_{\text{cat}}, \mathbf{e}_{\text{is}}, \mathbf{e}_{\text{on}}, \mathbf{e}_{\text{the}}, \mathbf{e}_{\text{mat}}]$$

其中 X 是一个 $6 \times d_{model}$ 的矩阵。

1.2 位置编码（Positional Encoding）

由于Transformer没有循环结构来捕捉顺序信息，因此我们需要通过**位置编码**将单词的位置信息加入。位置编码采用固定的正弦和余弦函数生成。位置编码 PE 的计算公式如下：

- 对于偶数索引（2i）：

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

- 对于奇数索引（2i+1）：

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

假设 $d_{model} = 512$，每个单词的位置编码是长度为 512 的向量。

例如，对于第一个位置 $pos = 0$ 的词，位置编码 PE_0 为：

$$\left[\sin(0), \cos(0), \sin\left(\frac{0}{10000^{\frac{2\times1}{512}}}\right), \cos\left(\frac{0}{10000^{\frac{2\times1}{512}}}\right), \ldots, \sin\left(\frac{0}{10000^{\frac{2\times256}{512}}}\right), \cos\left(\frac{0}{10000^{\frac{2\times256}{512}}}\right)\right]$$

这样的位置编码向量会与嵌入向量相加，得到最终的输入：

$$X_{final} = X + PE$$

2. 编码器部分（Encoder）

编码器由 N 个相同的层组成:

每一层相当于一个封装的处理单元，具有相同的子层结构，但不共享权重系数。

每一层的输出都依赖于前一层的输出，因此它们是**顺序处理**的，即层与层之间是依赖的。

每一层包括两个子层：**多头自注意力机制**和**前馈神经网络**。

2.1 多头自注意力机制（Multi-Head Self-Attention）

在多头自注意力机制中，输入 X 首先通过三个线性变换生成**查询 (Query)** Q、**键 (Key)** K 和 **值 (Value)** V。这些矩阵的计算方式如下：

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

其中，W^Q, W^K, W^V 是 $d_{model} \times d_{model}$ 的权重矩阵，将 X 映射到特定的语义空间。

2.1.1 点积注意力（Scaled Dot-Product Attention）

在自注意力机制中，我们首先计算查询和键之间的点积，得到注意力得分矩阵 A。

注意力得分 A_{ij} 表示输入序列中第 i 个单词对第 j 个单词的关注程度，计算公式为：

$$A_{ij} = \frac{Q_i K_j^T}{\sqrt{d_k}}$$

其中，$\sqrt{d_k}$ 是一个缩放因子，用来防止内积值过大而导致Softmax后梯度消失。随后，将每行进行Softmax归一化：

$$A_i = \text{Softmax}\left(\frac{Q_i K^T}{\sqrt{d_k}}\right)$$

这样可以得到每个查询对所有键的归一化注意力分布。最后，用这个注意力权重乘以值 V：

$$Z_i = A_i V$$

所有词的结果组合为矩阵形式：

$$Z = AV$$

即：

$$Z = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

2.1.2 多头机制（Multi-Head Attention）

多头注意力是将注意力机制在多个子空间上并行计算，每个子空间有独立的权重。
假设有 h 个头，每个头有自己的 W^Q , W^K , W^V ，分别应用于 Q 、 K 和 V 进行线性变换。
每个头会将 d_{model} 维度分为 $d_k = \frac{d_{model}}{h}$ ，从而使每个头能够在不同的子空间中关注不同的特征。

具体地，对于输入的查询 Q 、键 K 和值 V ，每个头的计算公式如下：

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

其中， $Q \in \mathbb{R}^{Seq_{len} \times d_{model}}$ ， $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$ ， $K \in \mathbb{R}^{Seq_{len} \times d_{model}}$ ， $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ ， $V \in \mathbb{R}^{Seq_{len} \times d_{model}}$ ， $W_i^V \in \mathbb{R}^{d_{model} \times d_k}$ 。

每个头的输出 head_i 具有维度 $Seq_{len} \times d_k$ ，其中 $d_k = \frac{d_{model}}{h}$ 。

接下来，将 h 个头的输出拼接在一起：

$$\text{Concat}(head_1, head_2, \ldots, head_h)$$

拼接后的维度为 $Seq_{len} \times (h \times d_k) = Seq_{len} \times d_{model}$ 。

最后，通过线性变换矩阵 $W^O \in \mathbb{R}^{d_{model} \times d_{model}}$ 将拼接结果变换为输出向量：

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, head_2, \ldots, head_h)W^O$$

2.2 前馈神经网络（Feed Forward Neural Network）

在自注意力机制之后，输出会经过一个前馈神经网络，它包括两个全连接层，中间使用ReLU激活函数：

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

其中，W_1 和 W_2 是可训练的权重，b_1 和 b_2 是偏置。

3. 解码器部分（Decoder）

解码器的结构与编码器类似，主要区别在于它有一个额外的**编码器-解码器注意力层**，用于关注编码器的输出。

3.1 自注意力机制和掩码（Masked Multi-Head Self-Attention）

解码器中也有自注意力机制，但在生成过程中引入了掩码来防止看到未来的单词。查询、键和值的计算与编码器相同：

$$Q = X_{decoder}W^Q, \quad K = X_{decoder}W^K, \quad V = X_{decoder}W^V$$

其中 $X_{decoder}$ 是解码器的输入。

掩码矩阵 M 用来遮蔽未来的单词，其形式为一个上三角矩阵：

$$M = \begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

注意力得分 A 加上掩码矩阵：

$$A_{masked} = A + M$$

然后通过Softmax归一化：

$$\alpha_{ij} = \frac{\exp(A_{masked,ij})}{\sum_{j=1}^{T} \exp(A_{masked,ij})}$$

最后得到掩码后的自注意力输出：

$$Z_i = \sum_{j=1}^{T} \alpha_{ij} V_j$$

即：

$$\text{Masked Attention}(Q, K, V) = Z = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

3.2 编码器-解码器注意力（Encoder-Decoder Attention）

解码器的每个查询 $Q_{decoder}$ 通过注意力机制关注编码器的输出 $K_{encoder}, V_{encoder}$：

$$A_{ij} = \frac{Q_{decoder,i} K_{encoder,j}^T}{\sqrt{d_k}}$$

同样通过Softmax归一化后，与编码器的值 $V_{encoder}$ 相乘，得到编码器-解码器的注意力输出。

$$\text{Attention}(Q_{decoder}, K_{encoder}, V_{encoder}) = \text{Softmax}\left(\frac{Q_{decoder} K_{encoder}^T}{\sqrt{d_k}}\right)V_{encoder}$$

3.3 输出层

解码器的最终输出经过一个线性层，并通过Softmax将其转换为词汇表上的概率分布：

$$\text{Softmax}(W_o h_t + b_o)$$

其中，h_t 是解码器的隐藏状态，W_o 是词汇表大小 V 的权重矩阵，b_o 是偏置。输出的概率分布表示每个词的生成概率。

4. 训练和推理

在训练过程中，Transformer使用**交叉熵损失**来优化生成的词与真实词之间的差距。损失函数为：

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(y_t | x, y_{<t})$$

其中 y_t 是目标句子的第 t 个单词，$P(y_t | x, y_{<t})$ 是Transformer生成该单词的概率。

在推理时，解码器逐步生成目标句子的每个词。例如，解码器会先生成 "Le"，然后根据 "Le" 生成 "chat"，接着是 "est"，直到生成完整的句子。

Extrainformation

这一部分用于查缺补漏，将未理解的/未注意到的细节进行补全。

- 在实际应用中，通常会同时给模型输入多个句子

  如果每个句子的长度不一样，选择一个合适的长度，作为输入文本序列的最大长度，也就是 batch_size

  如果一个句子达不到这个长度，那么就填充 padding
  如果句子超出这个长度，则做截断

  最大序列长度是一个超参数，通常希望越大越好，但是更长的序列往往会占用更大的训练显存/内存，因此需要在模型训练时候视情况进行决定