Transformer

作者：Calvin
QQ: 179209347
Mail: 179209347@qq.com

介绍

笔记简介：

- 面向对象：深度学习初学者
- 依赖课程：**线性代数，统计概率**，优化理论，图论，离散数学，微积分，信息论

知乎专栏：

https://zhuanlan.zhihu.com/p/693738275

Github & Gitee 地址：

https://github.com/mymagicpower/AIAS/tree/main/deep_learning

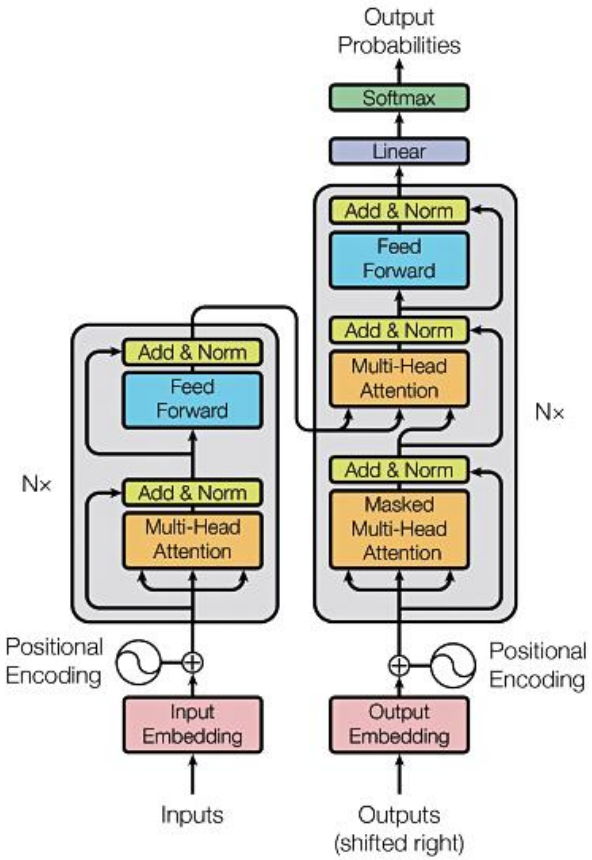
https://gitee.com/mymagicpower/AIAS/tree/main/deep_learning

*** 版权声明：**

- 仅限用于个人学习
- 禁止用于任何商业用途

Transformer 简介

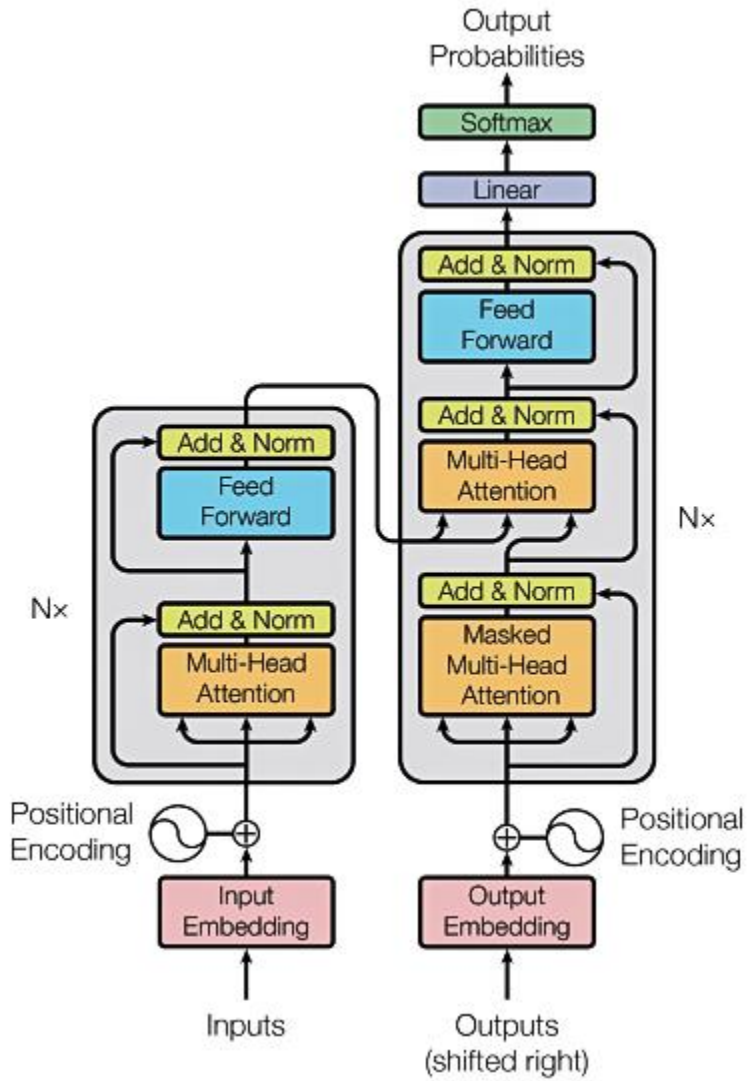
Transformer是一种基于注意力机制的深度学习模型架构，最初由Google的研究团队提出。它在自然语言处理领域取得了巨大成功，尤其是在机器翻译任务中表现出色。相比于传统的循环神经网络（RNN）和长短时记忆网络（LSTM），Transformer在处理长距离依赖关系时表现更好，同时也更易并行化。



Transformer 整体结构

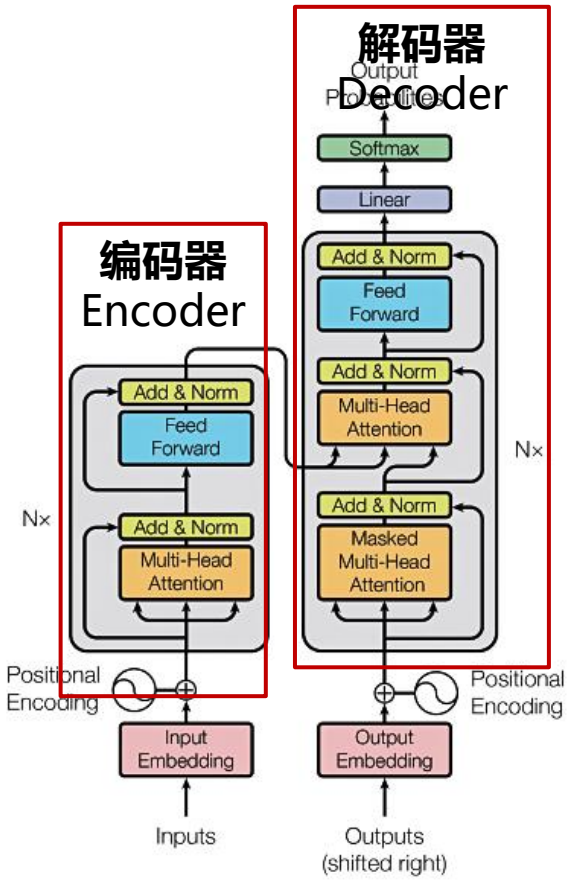
Transformer是一种基于自注意力机制的神经网络模型，用于处理序列数据。其要点如下：

- **自注意力机制**：通过计算序列中不同位置之间的注意力权重，捕捉位置间的依赖关系。
- **编码器-解码器结构**：包括编码器用于输入序列的表示和解码器用于生成输出序列的表示。
- **多头注意力**：使用多个注意力头来捕捉不同关注焦点和特征。
- **位置编码**：将序列中的位置信息嵌入到输入表示中，使模型能够处理序列的顺序信息。
- **残差连接和层归一化**：通过残差连接和层归一化来加速训练和提高模型性能。
- **前馈神经网络**：用于非线性变换和特征提取。
- **注意力层和前馈层的堆叠**：通过多个注意力层和前馈层的堆叠，实现多层的信息流动和建模。



Transformer 整体结构

Transformer 的内部结构图，左侧为 Encoder block，右侧为 Decoder block。
Encoder block 部分由一堆编码器（encoder）构成。
Decoder block 部分也是由相同数量（与编码器对应）的解码器（decoder）组成的。

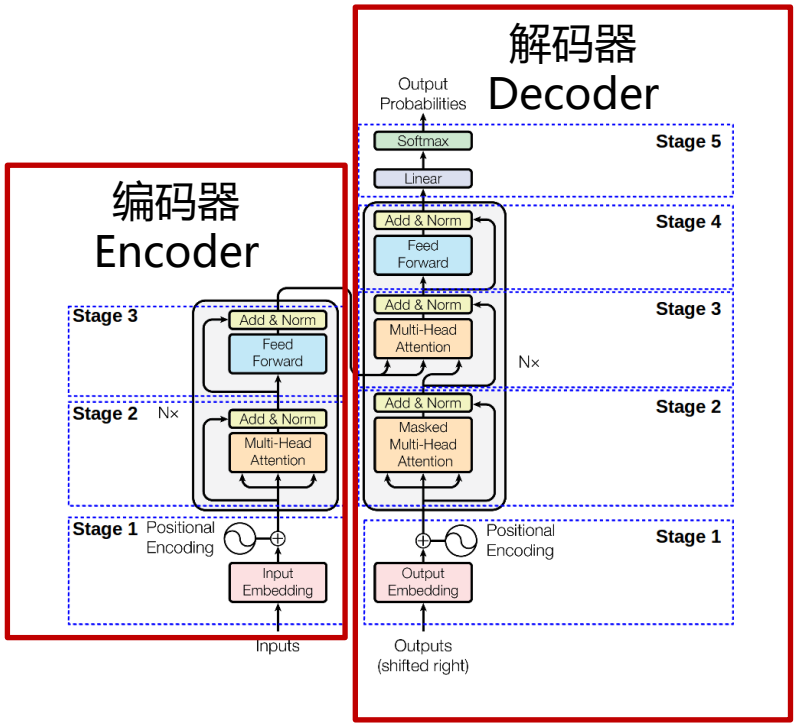



```python
class Transformer(nn.Module):

    def __init__(self, src_vocab, trg_vocab, d_model, N, heads, dropout):
        super().__init__()
        self.encoder = Encoder(src_vocab, d_model, N, heads, dropout)
        self.decoder = Decoder(trg_vocab, d_model, N, heads, dropout)
        self.out = nn.Linear(d_model, trg_vocab)

    def forward(self, src, trg, src_mask, trg_mask):
        e_outputs = self.encoder(src, src_mask)
        d_output = self.decoder(trg, e_outputs, src_mask, trg_mask)
        output = self.out(d_output)
        return output
```

大模型 – 架构分类

- **架构分类：**
  - **自回归模型 (Decoder-Only)：**单向注意力，擅长文本生成
  - **自编码模型 (Encoder-Only)：**双向注意力，擅长文本理解
  - **编码器-解码器模型 (Encoder-Decoder)：**编解码，擅长对话任务

算法框架	自然语言理解	有条件生成	无条件生成
自回归 Decoder-Only	—	—	√
自编码 Encoder-Only	√	×	×
编码器-解码器 Encoder-Decoder	—	√	—



大模型 – 架构分类

1 Encoder-Only 架构

- **简介：** Encoder-Only架构专注于理解和编码输入信息，常用于分类、标注等任务。
- **优点：**
  - 能够有效处理和理解输入数据。
  - 适用于多种分析型任务。
- **缺点：**
  - 有限的生成能力：不擅长自主生成文本或内容。
- **示例模型：**
  - Google 的 BERT

2 Decoder-Only 架构

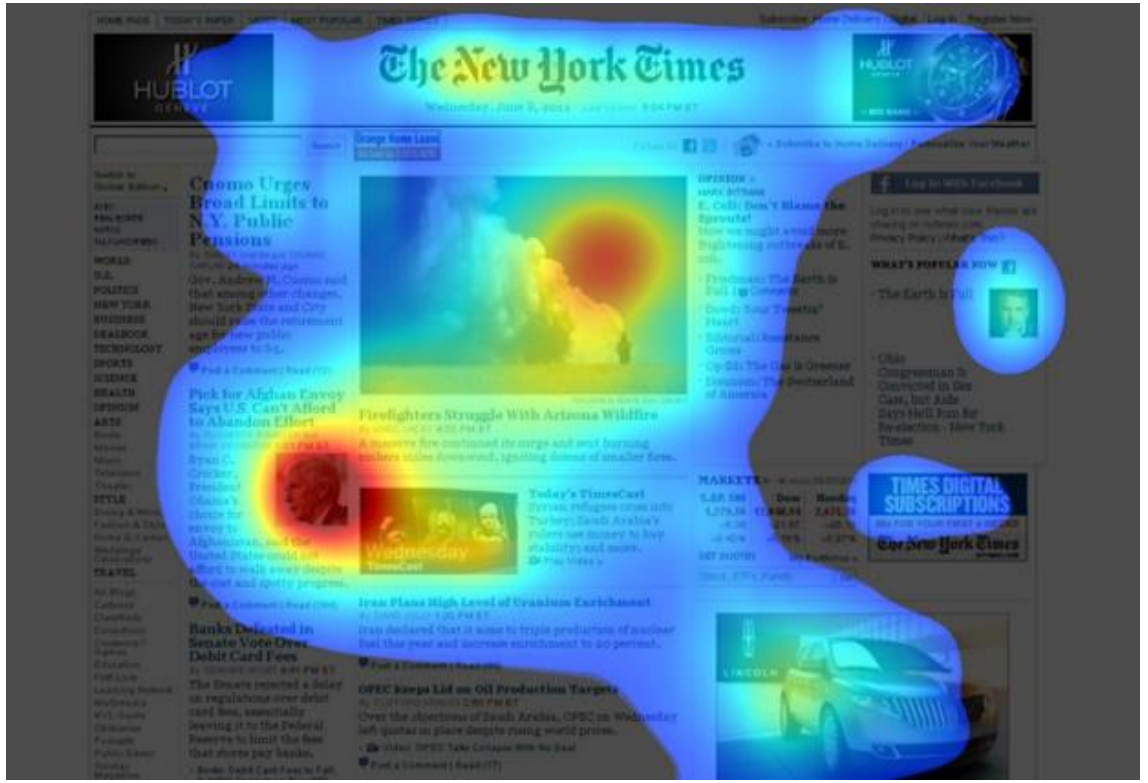
- **简介：** 专注于从一系列输入生成或预测输出。这种架构通常用于文本生成任务，如语言模型。
- **优点：**
  - 能够生成连贯、有创造性的文本。
  - 适用于各种生成型任务。
- **缺点：**
  - 有限的理解能力：不擅长理解复杂的输入数据。
- **示例模型：**
  - OpenAI的 GPT系列
  - Llama2

3 Encoder-Decoder 架构

- **简介：** Encoder-Decoder架构结合了编码器和解码器的优点，通常用于需要理解输入并生成相应输出的任务，如机器翻译。
- **优点：**
  - 能够理解复杂输入并生成相关输出。
  - 如机器翻译、文本摘要等。
- **缺点：**
  - 架构复杂：相比单一的Encoder或Decoder，更复杂。
  - 训练挑战：需要更多数据和计算资源。
- **示例模型：**
  - Google 的 T5
  - 智谱AI的 ChatGLM

自注意力机制（Self-Attention ）

- 人脑每个时刻接收的外界输入信息非常多，包括来源于视觉、听觉、触觉的各种各样的信息。
- 但就视觉来说，眼睛每秒钟都会发送千万比特的信息给视觉神经系统。
- 当过载信息映入眼帘时，我们的大脑会**把注意力放在主要的信息上**，这就是大脑的**注意力机制**。人脑通过注意力来解决信息超载问题。



核心逻辑是「从关注全部到关注重点」

自注意力机制（Self-Attention ）

自注意力机制的优点：

1.**参数少**：相比于 CNN、RNN ，其复杂度更小，参数也更少。所以对算力的要求也就更小。

2.**速度快**：Attention 解决了 RNN及其变体模型不能并行计算的问题。Attention机制每一步计算不依赖于上一步的计算结果，因此可以和CNN一样并行处理。

3.**效果好**：重要的一点，在self-attention机制中，无论词的绝对位置在哪，词与词之间的距离都是1。也就是说，其相关性的最大路径长度也只是1；而RNN中，最大长度是 n（开头至末尾），而距离越远，就会使得信息丢失越多。所以，self-attention能更好地捕获**全局信息。**

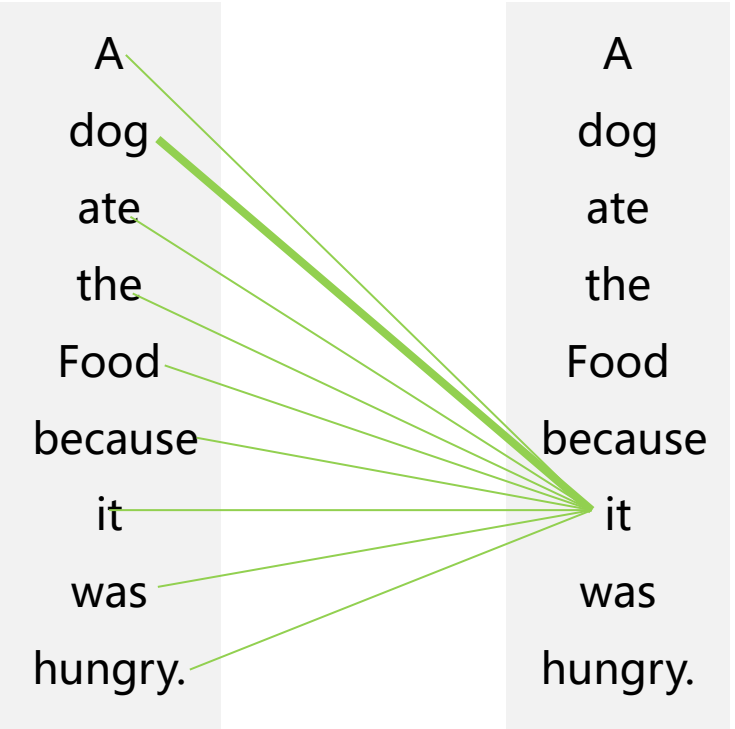
自注意力机制（Self-Attention ）

通过一个例子来快速理解自注意力机制：

A dog ate the food because it was hungry.

例句中的代词 it 可以指代 dog 或者 food。当读这文字的时候我们自然而然地认为 it 指代的是 dog，而不是 food。但是当计算机模型在面对这两种选择时该如何决定呢?这时，自注意力机制有助于解决这个问题。

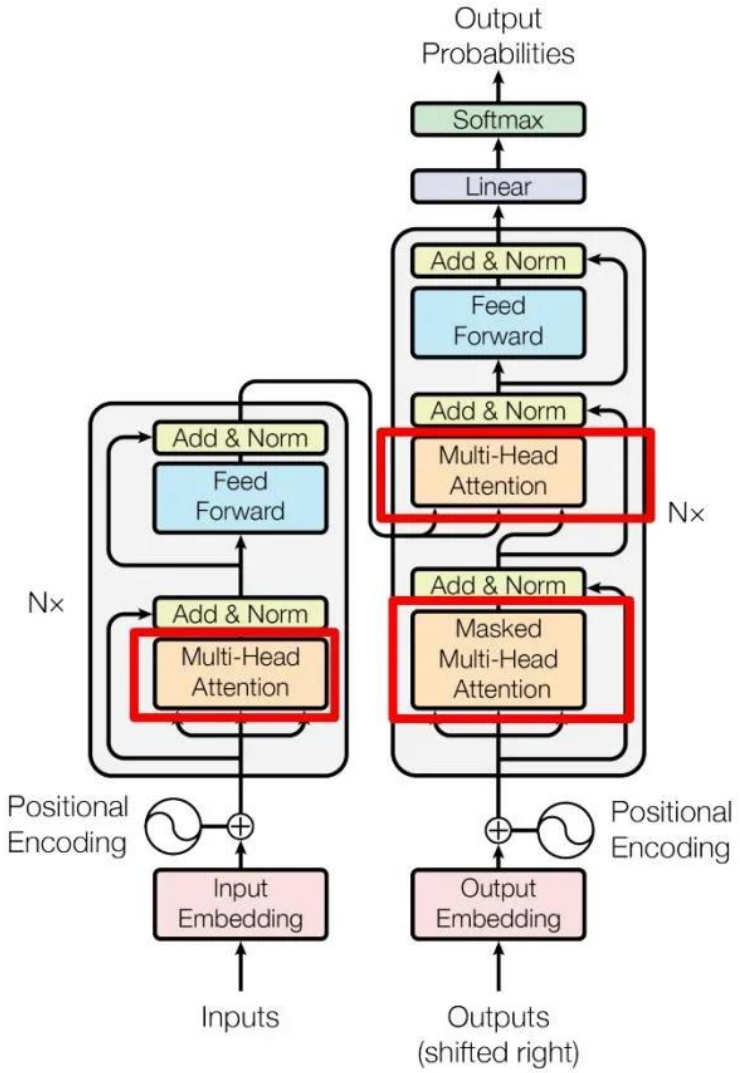
首先需要计算出单词 A 的特征值，其次计算 dog的特征值，然后计算 ate 的特征值，以此类推。
当计算每个词的特征值时，模型都需要遍历每个词与句子中其他词的关系。模型可以通过词与词之间的关系来更好地理解当前词的意思。比如，当计算 it 的特征值时，模型会将 it 与句子中的其他词一一关联，以便更好地理解它的意思。it 的特征值由它本身与句子中其他词的关系计算所得。通过关系连线，模型可以明确知道原句中 it 所指代的是 dog 而不是 food，这是因为 it 与 dog 的关系更紧密，关系连线相较于其他词也更粗。



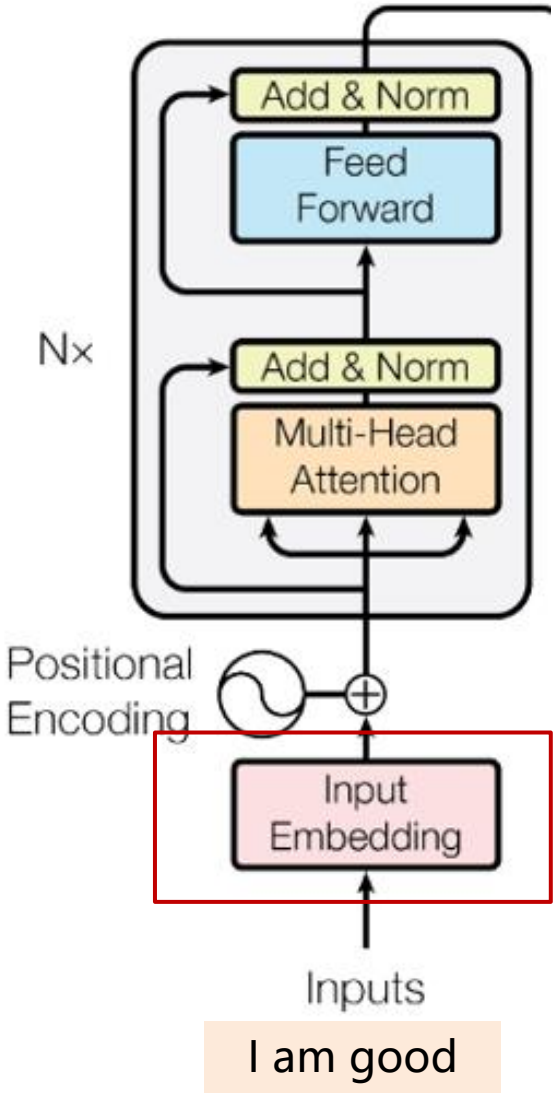
自注意力机制（Self-Attention ）

- 红色圈中的部分为多头注意力层 (Multi-Head Attention)，是由多个 Self-Attention 组成的
- Encoder block 包含一个 Multi-Head Attention
- Decoder block 包含两个 Multi-Head Attention (其中有一个用到 Masked)。
- Multi-Head Attention 上方还包括一个 Add & Norm 层，Add 表示残差连接 (Residual Connection) 用于防止网络退化，Norm 表示 Layer Normalization，用于对每一层的激活值进行归一化。

因为 **Self-Attention**是 Transformer 的重点，所以我们重点关注 Multi-Head Attention 以及 Self-Attention，首先详细了解一下 Self-Attention 的内部逻辑。



Transformer 的输入 - 输入矩阵 X



数据输入，单词Embedding：

为简单起见，我们假设输入句为 I am good。
首先，我们将每个词转化为其对应的词嵌入向量。需要注意的是，嵌入只是词的特征向量，这个特征向量也是**需要通过训练获得**的。
单词 I 的词嵌入向量可以用 x_1 来表示，相应地，am 为 x_2，good 为 x_3，即:

- 单词 I 的词嵌人向量 x_1 =[1.76，2.22，...，6.66]；
- 单词 am 的词嵌人向量 x_2 =[7.77，0.631，...，5.35]；
- 单词 good 的词嵌人向量 x_3 =[11.44，10.10 ，...， 3.33]。

如此，原句就可以用一个矩阵 X (输入矩阵或嵌入矩阵）来表示：

I	1.76	2.22	...	6.66	x_1
am	7.77	0.631	...	5.35	x_2
good	11.44	10.10	...	3.33	x_3

矩阵 X 的维度为 [句子的长度 x 词嵌入向量维度]。
假设词嵌入向量维度为512，那么输入矩阵的维度就是 [3 x 512]。

Transformer 的输入 - 位置编码

以 I am good 为例。在 RNN模型中，句子是逐字送入学习网络的。换言之，首先把 I 作为输人，接下来是 am，以此类推。通过逐字地接受输人，学习网络就能完全理解整个句子。

然而，Transformer 网络并不遵循递归循环的模式。因此我们不是逐字地输人句子，而是将句子中的所有词并行地输人到神经网络中。并行输入有助于缩短训练时间，同时有利于学习长期依赖。

不过，并行地将词送入Transformer 却不保留词序，而Transformer也需要一些关于词序的信息，以便更好地理解句子。对于给定的句子 I am good, 我们首先计算每个单词在句子中的嵌入值。嵌入维度可以表示为 $\boldsymbol{d}_{\boldsymbol{model}}$。比如将嵌入维度 $\boldsymbol{d}_{\boldsymbol{model}}$ 设为4，那输入矩阵的维度将是 [句子长度 x 嵌入维度]，也就是[3 x 4]。

同样，用输入矩阵 X(嵌人矩阵) 表示输入 I am good。假设输入矩阵X如图：

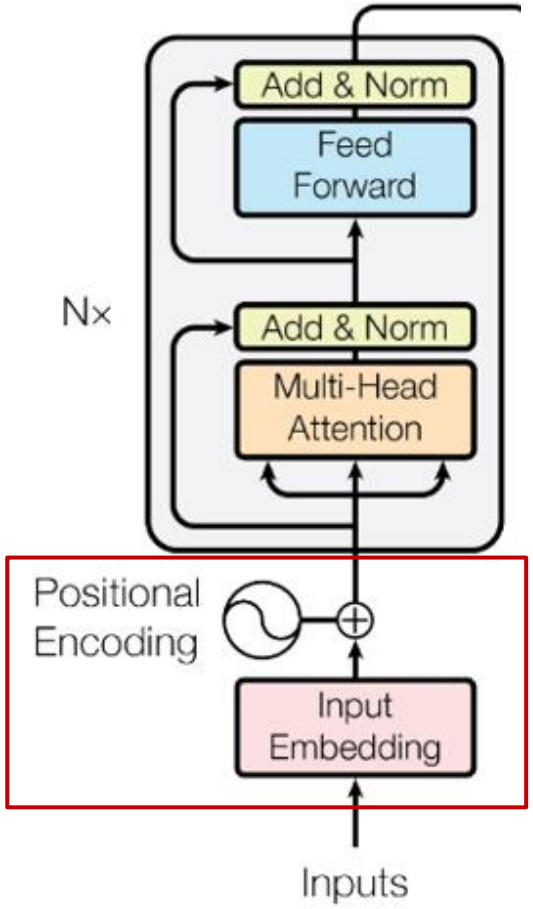
I	1.76	2.22	3.4	5.8	x_1
am	7.3	9.9	8.5	7.1	x_2
good	9.1	7.1	0.85	10.1	x_3

X

输入矩阵

Transformer 的输入 - 位置编码 - 位置编码矩阵 P

在self-attention机制中，不包含数据的序列信息。类似于词袋模型，每个词都会和其他所有词计算相关性，丢失了词顺序的信息。如果把输入矩阵 X 直接传给 Transformer，那么模型是无法理解词序的。为了弥补这一点，提出添加位置编码。
位置编码是指词在句子中的位置（词序）的编码。
位置编码矩阵 P 的维度与输入矩阵 X 的维度相同。



X' =

1.76	2.22	3.4	5.8	x_1
7.3	9.9	8.5	7.1	x_2
9.1	7.1	0.85	10.1	x_3

X

+

0	1	0	1
0.84	0.54	0.01	0.99
0.91	-0.4	0.02	0.99

P

=

1.76	3.22	3.4	6.8
8.14	10.44	8.51	8.09
9.1	6.7	0.87	11.09

Transformer 的输入 - 位置编码 - 位置编码矩阵 P 计算过程

位置编码用 **PE**表示，**PE** 的维度与单词嵌入是一样的。PE 可以通过训练得到，也可以使用某种公式计算得到。在 Transformer的论文“Attention Is All You Need”中采用了后者，作者使用了正弦函数来计算位置编码，计算公式如下：

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

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

pos 对应输入的位置，i 是指该词向量的内部位置。

```python
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()
```

$\boldsymbol{P}$ =

I	$\sin(\frac{pos}{10000^0})$	$\cos(\frac{pos}{10000^0})$	$\sin(\frac{pos}{10000^{2/4}})$	$\cos(\frac{pos}{10000^{2/4}})$
am	$\sin(\frac{pos}{10000^0})$	$\cos(\frac{pos}{10000^0})$	$\sin(\frac{pos}{10000^{2/4}})$	$\cos(\frac{pos}{10000^{2/4}})$
good	$\sin(\frac{pos}{10000^0})$	$\cos(\frac{pos}{10000^0})$	$\sin(\frac{pos}{10000^{2/4}})$	$\cos(\frac{pos}{10000^{2/4}})$

i 是偶数时，使用正弦函数；
i 是奇数时，使用余弦函数。

Transformer 的输入 - 位置编码 - 位置编码矩阵 P 计算过程

通过简化矩阵中的公式，可以得到：

$$\boldsymbol{P} = \begin{array}{r} \text{I} \\ \text{am} \\ \text{good} \end{array} \begin{array}{|c|c|c|c|} \hline \sin(pos) & \cos(pos) & \sin(\frac{pos}{100}) & \cos(\frac{pos}{100}) \\ \hline \sin(pos) & \cos(pos) & \sin(\frac{pos}{100}) & \cos(\frac{pos}{100}) \\ \hline \sin(pos) & \cos(pos) & \sin(\frac{pos}{100}) & \cos(\frac{pos}{100}) \\ \hline \end{array}$$

我们知道 I 位于句子的第 0 位置，am 在第 1 位置， good 在第 2 位置。带入 pos 值，我们可以得到：

$$\boldsymbol{P} = \begin{array}{r} \text{I} \\ \text{am} \\ \text{good} \end{array} \begin{array}{|c|c|c|c|} \hline \sin(0) & \cos(0) & \sin(\frac{0}{100}) & \cos(\frac{0}{100}) \\ \hline \sin(1) & \cos(1) & \sin(\frac{1}{100}) & \cos(\frac{1}{100}) \\ \hline \sin(2) & \cos(2) & \sin(\frac{2}{100}) & \cos(\frac{2}{100}) \\ \hline \end{array} = \begin{array}{r} \text{I} \\ \text{am} \\ \text{good} \end{array} \begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 1 \\ \hline 0.84 & 0.54 & 0.01 & 0.99 \\ \hline 0.91 & -0.4 & 0.02 & 0.99 \\ \hline \end{array}$$

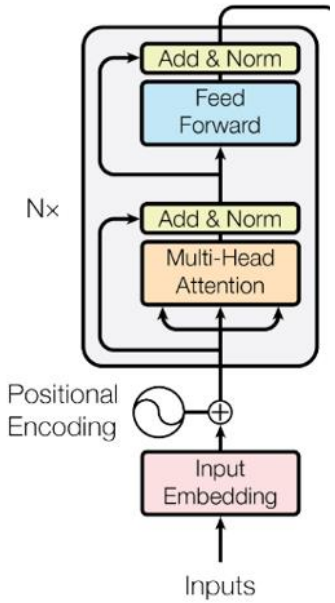
Encoder 结构

编码器层包括：
多头自注意力层：主要是一个多头注意力层，之后是一个残差连接和层规范化。

- 残差连接 Add： 本来 y = sublayer(x)，使用残差连接后 ：y =sublayer(x) + x，可以避免梯度消失的问题。
- 层归一化 Norm：让数据分布更合理，加速收敛。

前馈全连接层：使用一个前馈网络，之后是一个残差连接和层规范化。使用了两次线性变换和一次 Relu 激活函数。

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

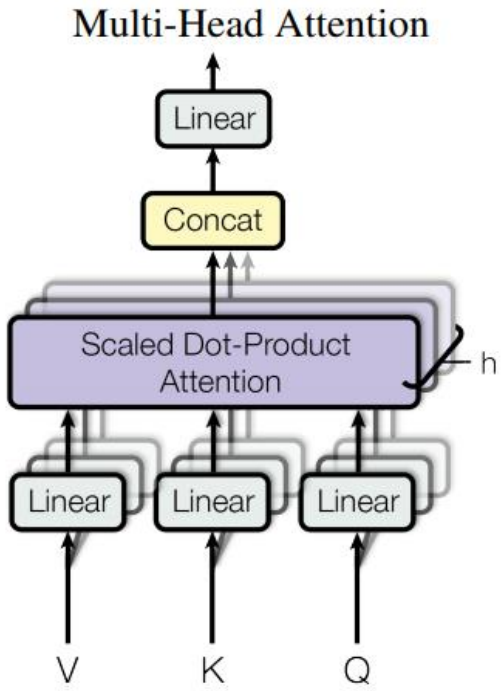



```python
class EncoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.attn = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)
        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)

    def forward(self, x, mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn(x2,x2,x2,mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.ff(x2))
        return x


class Encoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(EncoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)
```

```python
   def forward(self, src, mask):
        x = self.embed(src)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, mask)
        return self.norm(x)
```

Encoder 结构 - 多头（Multi-Head）自注意力层

- it 的自注意力值正是 dog的值向量。在这里，单词 it 的自注意力值被 dog 所控制。这是正确的，因为 it 的含义模糊，它指的既可能是 dog，也可能是 food。
- 如果某个词实际上由其他词的值向量控制，而这个词的含义又是模糊的，那么这种控制关系是有用的;
- 否则，这种控制关系反而会造成误解。为了确保结果准确、我们不能依赖单一的注意力矩阵，而应该计算多个注意力矩阵，并将其结果串联起来。
- 使用多头注意力的逻辑是这样的：使用多个注意力矩阵，而非单一的注意力矩阵，可以提高注意力矩阵的准确性。



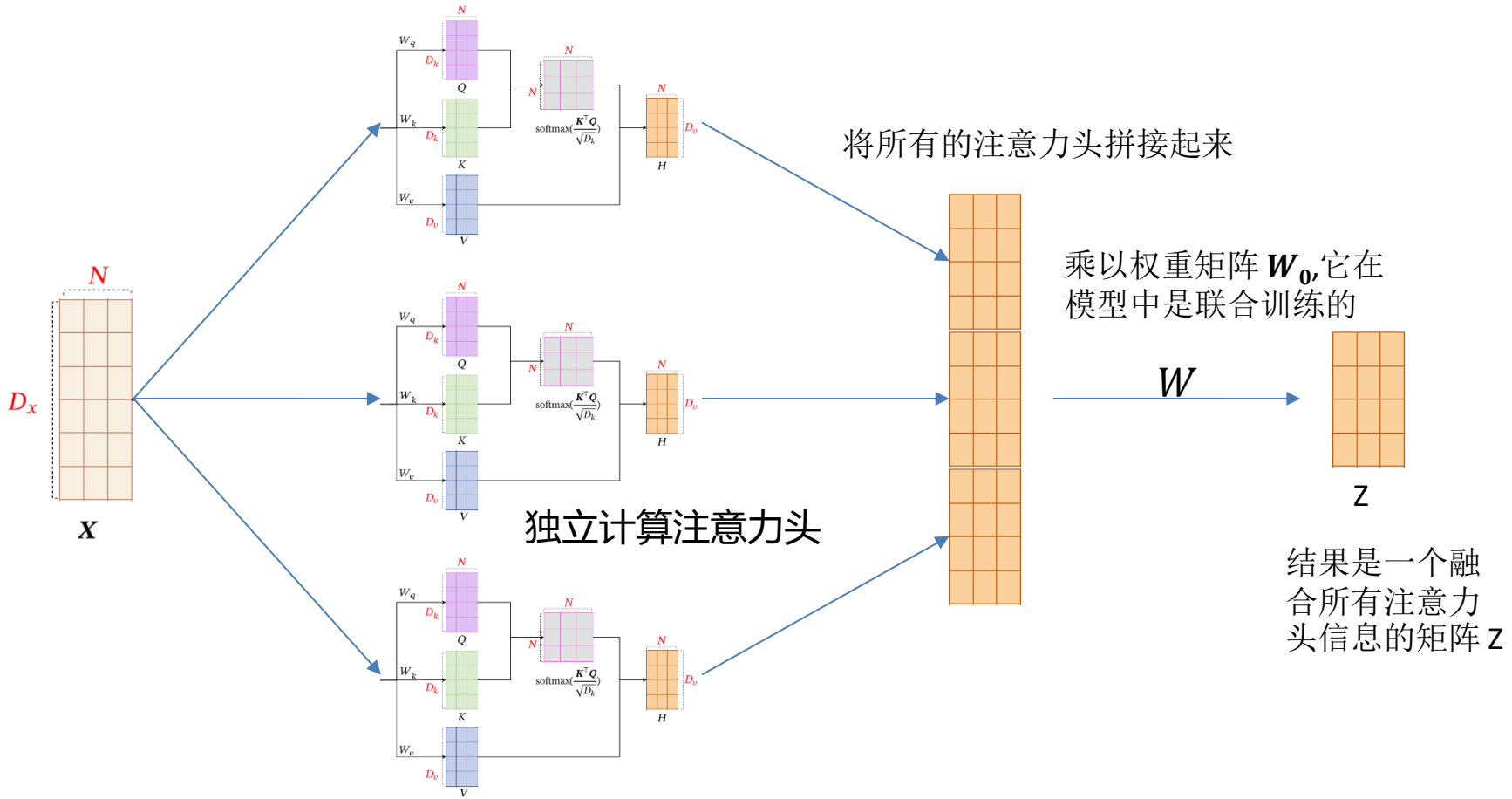
$$z_{it} = 0.0\ v_1 + 1.0 v_2 + ... + 0.0\ v_5 + ... + 0.0\ v_9$$

A dog food hungry

Encoder 结构 - 多头（Multi-Head）自注意力层

- 多头自注意力模型是一种扩展的自注意力模型，通过**使用多个注意力头**来捕捉不同的关注焦点和特征。每个注意力头都有独立的查询、键和值矩阵，并生成一个加权向量。这些加权向量可以进行拼接或平均，得到最终的输出表示。多头自注意力模型能够提高模型的表达能力和泛化能力，从而在处理序列数据时取得更好的性能。

Multi-Head attention = Concatenate(Z_1, Z_2,..., Z_i,..., Z_8) W_0



Encoder 结构 - 前馈层

前馈层接受自注意力子层的输出作为输入，并通过一个带有 Relu 激活函数的两层全连接网络对输入进行更加复杂的非线性变换。实验证明，这一非线性变换会对模型最终的性能产生十分重要的影响。

其中 $\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_1$, $\mathbf{b}_2$表示前馈子层的参数。实验结果表明，增大前馈子层隐状态的维度有利于提升最终翻译结果的质量，因此，前馈子层隐状态的维度一般比自注意力子层要大。

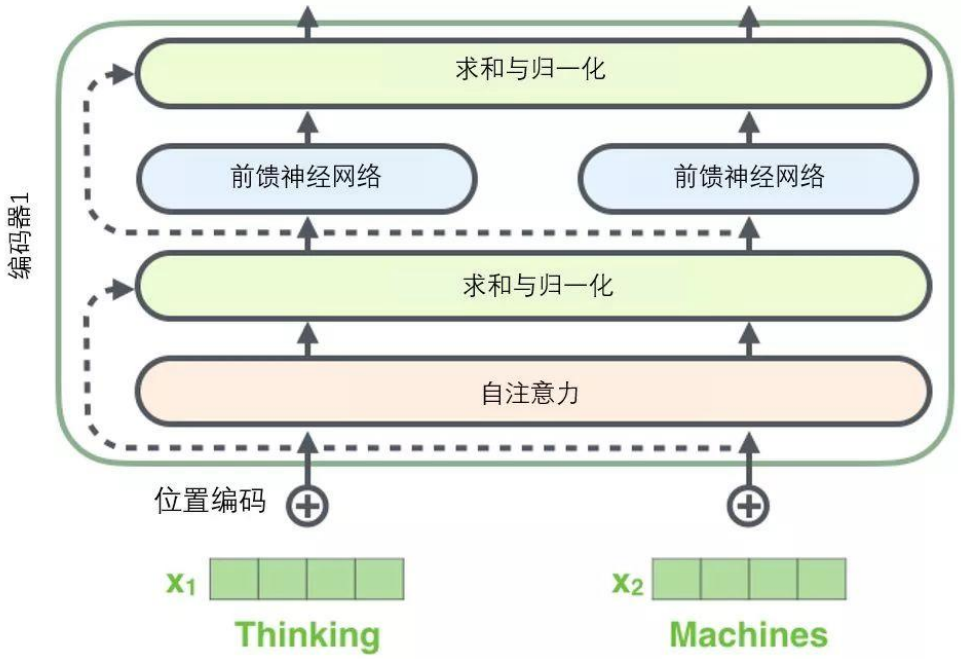
$$\mathrm{FFN}(\mathbf{x}) = \mathrm{Relu}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

使用 Pytorch 实现的前馈层参考代码如下：

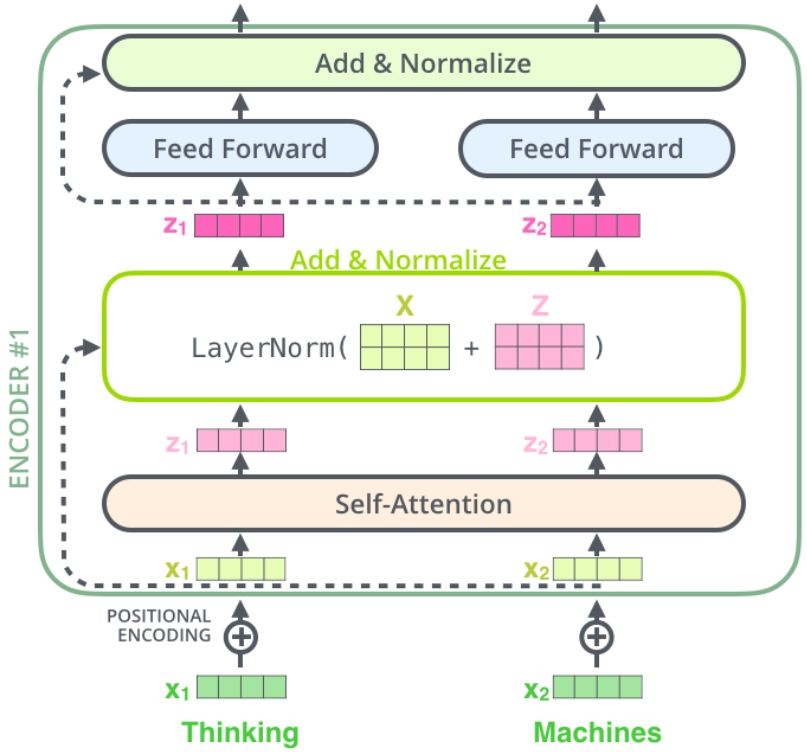
```
class FeedForward(nn.Module):

    def __init__(self, d_model, d_ff=2048, dropout = 0.1):
        super().__init__()

        # d_ff 默认设置为 2048
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)
```

Encoder 结构 - 残差与归一化

在每个编码器中的每个子层（自注意力、前馈网络）的周围都有一个残差连接，并且都跟随着一个“层-归一化”步骤。



如果我们去可视化这些向量以及这个和自注意力相关联的层-归一化操作，那么看起来就像下面这张图描述一样：



Encoder 结构 - 残差连接与归一化

由 Transformer 结构组成的网络结构通常都是非常庞大。编码器和解码器均由很多层基本的 Transformer 块组成，每一层当中都包含复杂的非线性映射，这就导致模型的训练比较困难。因此， 研究者们在 Transformer 块中进一步引入了残差连接与层归一化技术以进一步提升训练的稳定性。 具体来说，残差连接主要是指使用一条直连通道直接将对应子层的输入连接到输出上去，从而避免由于网络过深在优化过程中潜在的梯度消失问题：

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

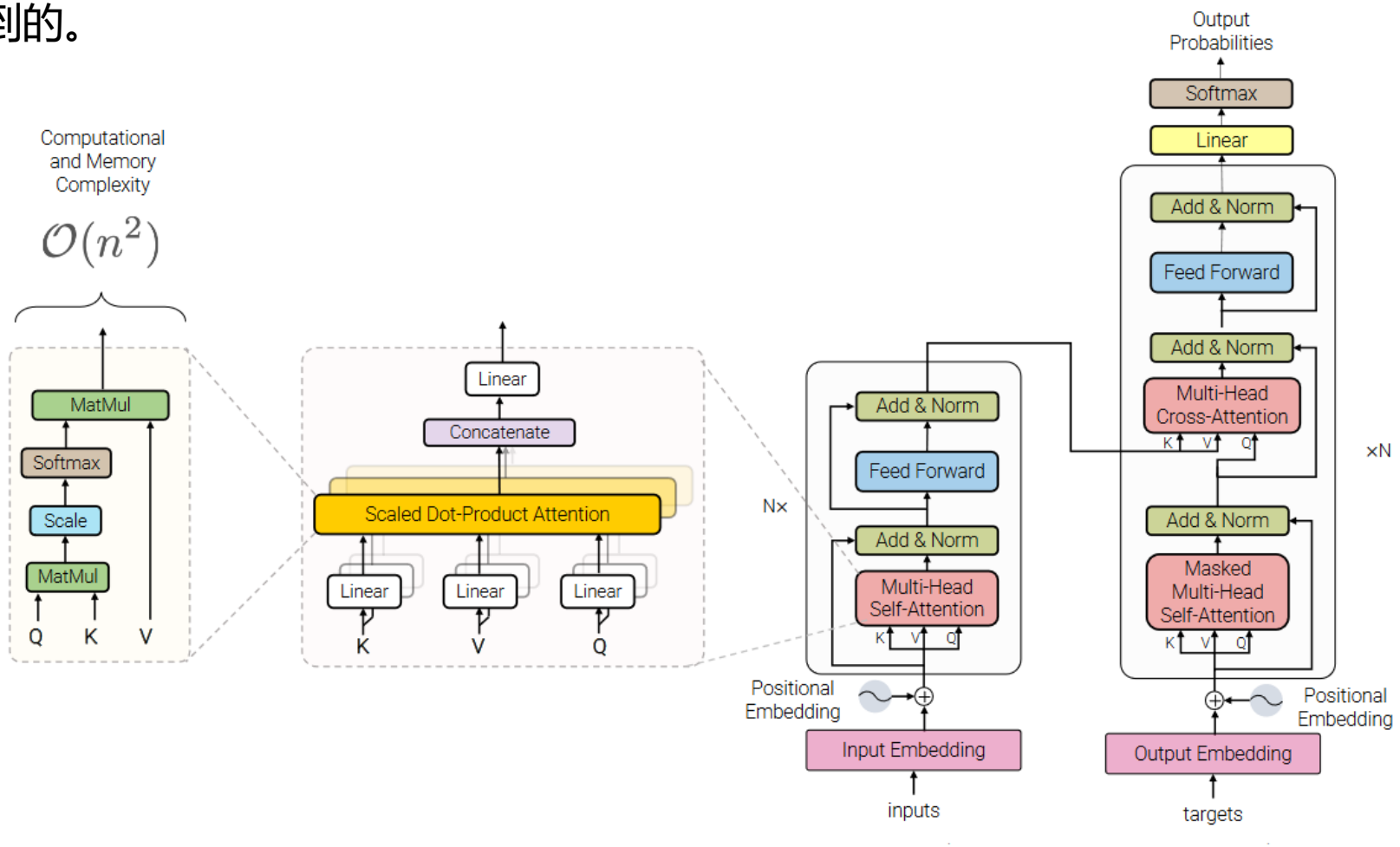
$$LN(\boldsymbol{x}) = \alpha \cdot \frac{\boldsymbol{x} - \mu}{\sigma} + b$$

其中 μ 和 σ 分别表示均值和方差，用于将数据平移缩放到均值为 0，方差为 1 的标准分布，α 和 b 是可学习的参数。层归一化技术可以有效地缓解优化过程中潜在的不稳定、收敛速度慢等问题。

```python
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
```

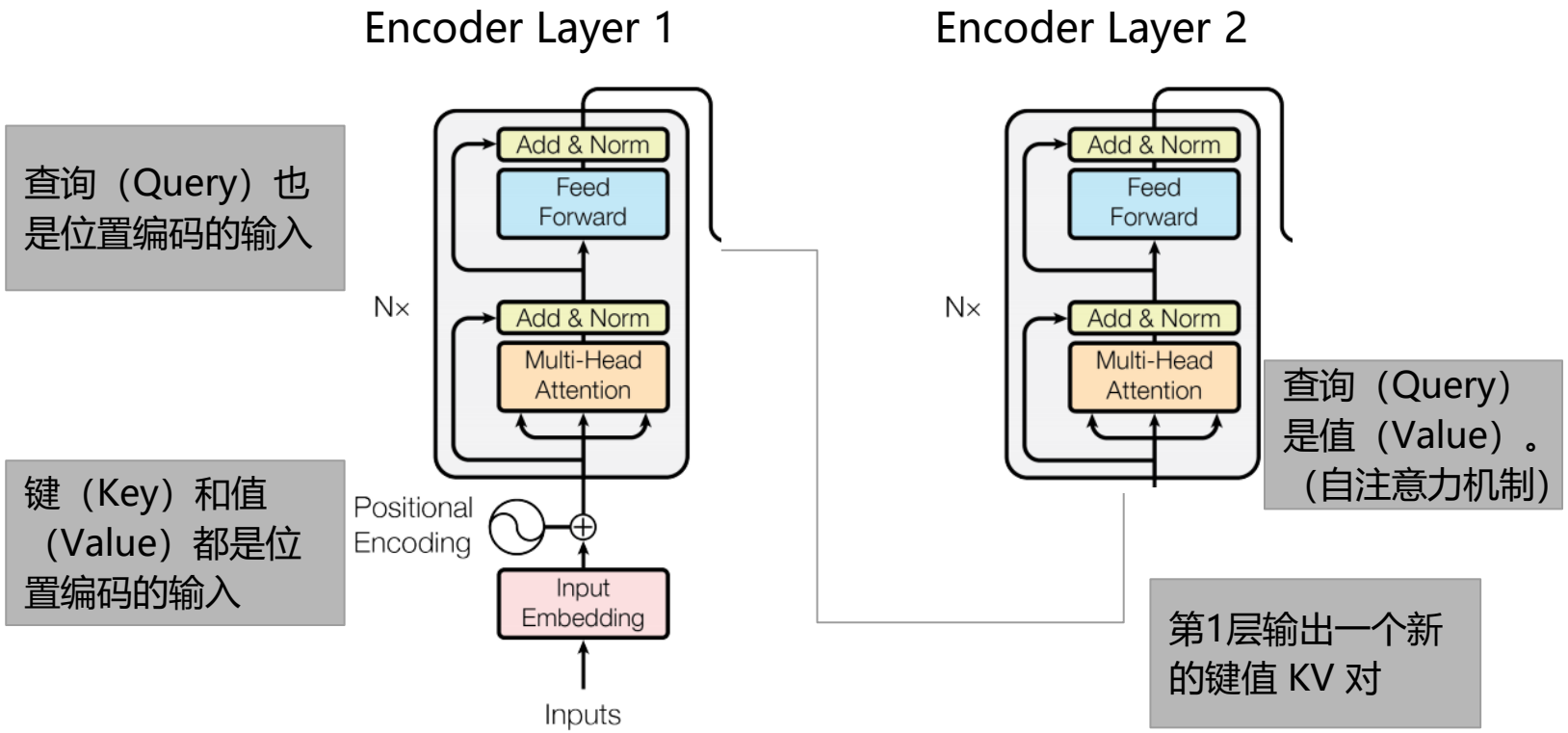
Encoder 结构 - Self-Attention 结构

在计算的时候需要用到矩阵 Q(查询), K(键值), V(值)。在实际中，Self-Attention 接收的是输入(单词的表示向量 x 组成的矩阵 X) 或者上一个 Encoder block 的输出。而Q, K, V 正是通过 Self-Attention 的输入进行线性变换得到的。



Encoder 结构 - 编码器中的 Q 、K、V

- K（Key）：表示键（Key），用于存储与输入序列相关的信息。每个输入位置都有一个对应的键。
- V（Value）：表示值（Value），用于存储与输入序列相关的信息。每个输入位置都有一个对应的值。
- Q（Query）：表示查询（Query），用于提取与输入序列相关的信息。每个查询都会与所有的键进行比较，以获取相关的值。



Encoder 结构 – Q, K, V 的计算

现在通过矩阵 $\boldsymbol{X}$，我们再创建三个新的矩阵：
查询 (query) 矩阵 $\boldsymbol{Q}$、键 (key) 矩阵 $\boldsymbol{K}$，以及值 (value) 矩阵 $\boldsymbol{V}$。

为了创建查询矩阵、键矩阵和值矩阵，我们需要先创建另外三个权重矩阵 $\boldsymbol{W_q}$，$\boldsymbol{W_k}$，$\boldsymbol{W_v}$（**需要通过训练获得**）计算得到 $\boldsymbol{Q, K, V}$，计算如图所示：

- 输入序列为 $X = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N] \in R^{D_x \times N}$
- 计算得到 $\boldsymbol{Q, K, V}$

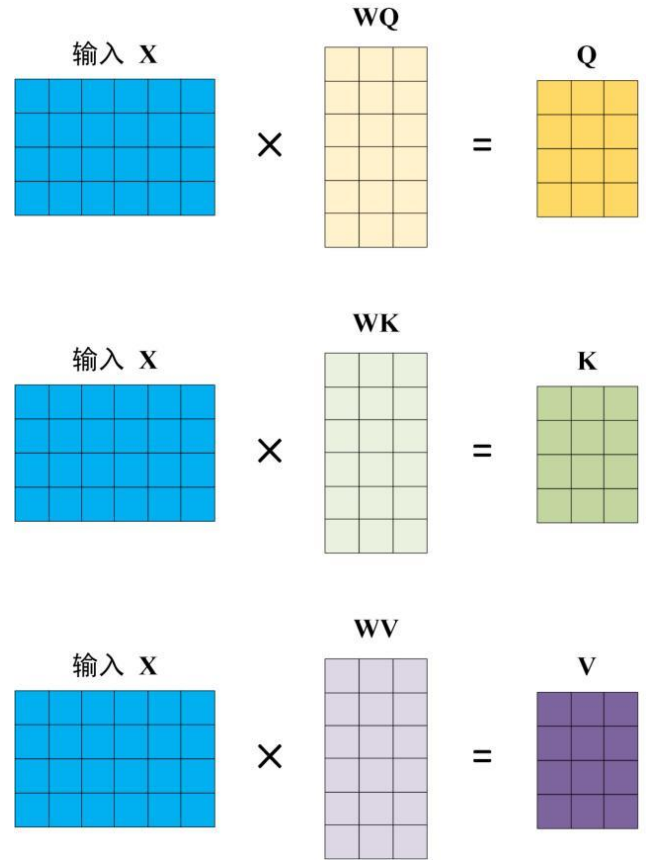
$$\boldsymbol{Q} = \boldsymbol{W_q X} \in \mathbb{R}^{D_k \times N},$$
$$\boldsymbol{K} = \boldsymbol{W_k X} \in \mathbb{R}^{D_k \times N},$$
$$\boldsymbol{V} = \boldsymbol{W_v X} \in \mathbb{R}^{D_v \times N},$$

$D_k \times N$ Q = W_q X

$D_k \times N$ K = W_k X

$D_v \times N$ V = W_v X

$\boldsymbol{X, Q, K, V}$ 的每一行都表示一个单词。



Encoder 结构 – Q, K, V 的计算

$$\boldsymbol{Q} = \boldsymbol{W}_q\boldsymbol{X} \in \mathbb{R}^{D_k \times N},$$
$$\boldsymbol{K} = \boldsymbol{W}_k\boldsymbol{X} \in \mathbb{R}^{D_k \times N},$$
$$\boldsymbol{V} = \boldsymbol{W}_v\boldsymbol{X} \in \mathbb{R}^{D_v \times N},$$

I	1.76	2.22	…	6.66	x_1
am	7.77	0.631	…	5.35	x_2
good	11.44	10.10	⋯	3.33	x_3

$\boldsymbol{X}$
输入矩阵

W_q X

W_k X

W_v X

I	3.69	7.42	…	4.44	q_1
am	11.11	7.07	…	76.7	q_2
good	99.3	3.69	⋯	0.85	q_3

$\boldsymbol{Q}$
查询 (query) 矩阵
每行代表单词的查询向量

I	5.31	6.78	…	0.96	k_1
am	11.71	0.86	…	11.31	k_2
good	10.10	11.44	⋯	5.11	k_3

$\boldsymbol{K}$
键 (key) 矩阵
每行代表单词的键向量

I	67.85	91.2	…	0.13	v_1
am	13.13	63.1	…	4.44	v_2
good	12.12	96.1	⋯	43.4	v_3

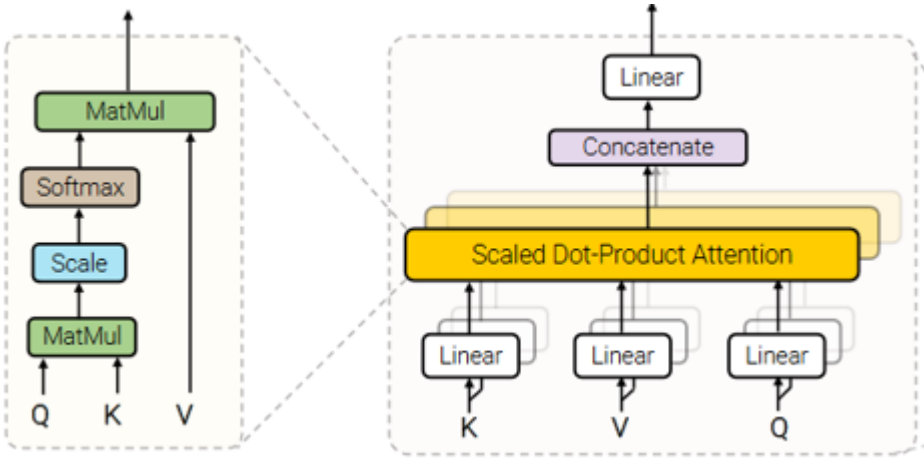
$\boldsymbol{V}$
值 (value) 矩阵
每行代表单词的值向量

Encoder 结构 - Self-Attention 的计算过程

- 得到矩阵 Q, K, V之后就可以计算出 Self-Attention 的输出了，自注意力机制也称为缩放点积注意力机制（$\sqrt{d_k}$ 对结果进行缩放），计算的公式如下：

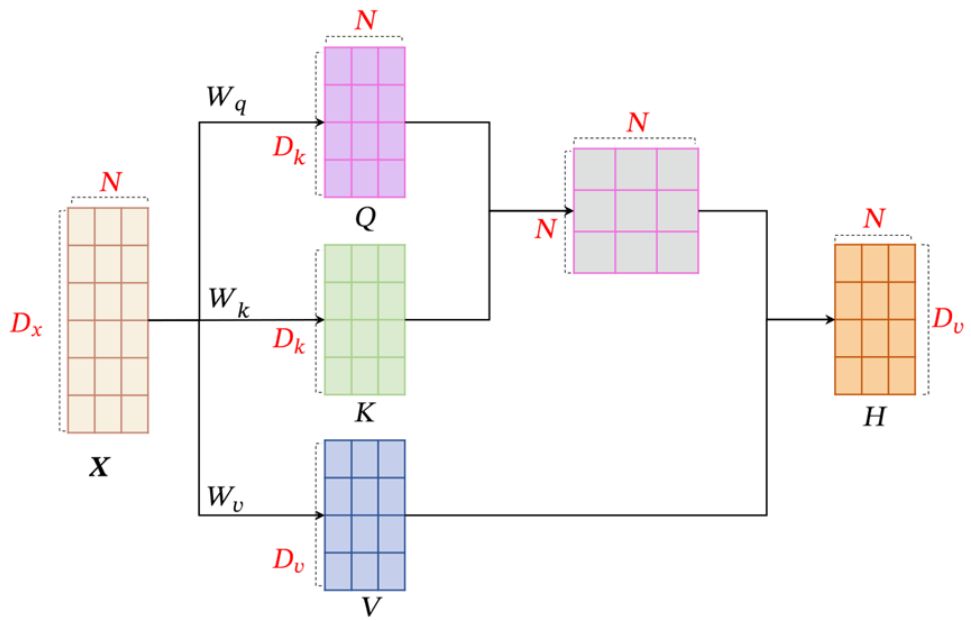
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

d_k 是 Q, K 矩阵的列数，即向量的维度



Encoder 结构 - Self-Attention 的计算过程

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

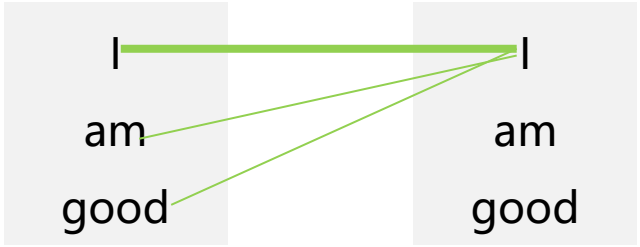



```python
class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout = 0.1):
        super().__init__()

        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads

        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)
        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(d_model, d_model)

    def attention(q, k, v, d_k, mask=None, dropout=None):
        scores = torch.matmul(q, k.transpose(-2, -1)) /  math.sqrt(d_k)

        # 掩盖掉那些为了填补长度增加的单元，使其通过 softmax 计算后为 0
        if mask is not None:
            mask = mask.unsqueeze(1)
            scores = scores.masked_fill(mask == 0, -1e9)

        scores = F.softmax(scores, dim=-1)

        if dropout is not None:
            scores = dropout(scores)

        output = torch.matmul(scores, v)
        return output

    def forward(self, q, k, v, mask=None):

        bs = q.size(0)

        # 进行线性操作划分为成 h 个头
        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)
```

Encoder 结构 - Self-Attention 的计算过程 – 第 1 步

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

首先要计算一个词的特征值，自注意力机制会使该词与给定句子中的所有词联系起来。以 I am good 这句话为例。为了计算单词I的特征值，我们将单词 I 与句子中的所有单词一一关联，如图所示：



自注意力机制首先要计算查询矩阵 $\boldsymbol{Q}$ 与键矩阵 $\boldsymbol{K}^{\mathsf{T}}$ 的点积， $\boldsymbol{Q}\cdot\boldsymbol{K}^{\mathsf{T}}$：

I	3.69	7.42	…	4.44
am	11.11	7.07	…	76.7
good	99.3	3.69	…	0.85

q_1 q_2 q_3

$\boldsymbol{Q}$

查询 (query) 矩阵

每行代表单词的查询向量

•

5.31	11.71	10.10
6.78	0.86	11.44
…	…	…
0.96	11.31	5.11

k_1 k_2 k_3

$\boldsymbol{K}^{\mathsf{T}}$

=

$q_1\cdot k_1$	$q_1\cdot k_2$	$q_1\cdot k_3$
$q_2\cdot k_1$	$q_2\cdot k_2$	$q_2\cdot k_3$
$q_3\cdot k_1$	$q_3\cdot k_2$	$q_3\cdot k_3$

$\boldsymbol{Q}\cdot\boldsymbol{K}^{\mathsf{T}}$

=

	I	am	good
I	110	90	80
am	70	99	70
good	90	70	100

$\boldsymbol{Q}\cdot\boldsymbol{K}^{\mathsf{T}}$

Encoder 结构 - Self-Attention 的计算过程 – 第 1 步

为何需要计算查询矩阵与键矩阵的点积 $\boldsymbol{Q}\cdot\boldsymbol{K}^{\mathrm{T}}$ 呢?
首先，来看 $\boldsymbol{Q}\cdot\boldsymbol{K}^{\mathrm{T}}$ 矩阵的第一行，这一行计算的是:
查询向量 $\boldsymbol{q_1}$ (I) 与所有的键向量 $\boldsymbol{k_1}$ (I)、$\boldsymbol{k_2}$ (am) 和 $\boldsymbol{k_3}$ (good) 的点积。

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

通过计算两个向量的点积可以知道它们之间的**相似度。**因此，通过计算查询向量 $\boldsymbol{q_1}$ 和键向量 ($\boldsymbol{k_1}$、 $\boldsymbol{k_2}$、 $\boldsymbol{k_3}$) 的点积，可以了解单词 I 与句子中的所有单词的相似度。我们了解到，这个词与自己的关系比与 am 和 good 这两个词的关系更紧密，因为点积值 $\boldsymbol{q_1}\cdot\boldsymbol{k_1}$ 大于 $\boldsymbol{q_1}\cdot\boldsymbol{k_2}$,和 $\boldsymbol{q_1}\cdot\boldsymbol{k_3}$ 。

$\boldsymbol{Q}\cdot\boldsymbol{K}^{\mathrm{T}}$ =

	I	am	good
I	110	90	80
am	70	99	70
good	90	70	100

$q_1\cdot k_1$	$q_1\cdot k_2$	$q_1\cdot k_3$
$q_2\cdot k_1$	$q_2\cdot k_2$	$q_2\cdot k_3$
$q_3\cdot k_1$	$q_3\cdot k_2$	$q_3\cdot k_3$

计算查询向量 $\boldsymbol{q_1}$ 和键向量 ($\boldsymbol{k_1}$、 $\boldsymbol{k_2}$、 $\boldsymbol{k_3}$) 的点积: I 与句子中所有词的相似度。

计算查询向量 $\boldsymbol{q_2}$ 和键向量 ($\boldsymbol{k_1}$、 $\boldsymbol{k_2}$、 $\boldsymbol{k_3}$) 的点积: am 与句子中所有词的相似度。

计算查询向量 $\boldsymbol{q_3}$ 和键向量 ($\boldsymbol{k_1}$、 $\boldsymbol{k_2}$、 $\boldsymbol{k_3}$) 的点积: good 与句子中所有词的相似度。

综上，计算查询矩阵与键矩阵的点积 $\boldsymbol{Q}\cdot\boldsymbol{K}^{\mathrm{T}}$ ，从而得到相似度分数。
也就是**得到了句子中每个词与所有其它词的相似度。**

Encoder 结构 - Self-Attention 的计算过程 – 第 2 步

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

自注意力计算的第2步，是将 $\boldsymbol{Q}\bullet\boldsymbol{K}^{\mathrm{T}}$ 矩阵除以键向量维度的平方根。
这样做的主要目的是获得稳定的梯度。
假定键向量维度 d_k 是64，那么 $\sqrt{d_k}$ = 8.

$\frac{QK^T}{\sqrt{d_k}}$ = $\frac{QK^T}{8}$ =

	I	am	good
I	110/8	90/8	80/8
am	70/8	99/8	70/8
good	90/8	70/8	100/8

=

	I	am	good
I	13.75	11.25	10
am	8.75	12.38	8.75
good	11.25	8.75	12.5

Encoder 结构 - Self-Attention 的计算过程 – 第 3 步

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

目前所得的相似度分数尚未被归一化，我们需要使用 **softmax** 函数对其进行归一化处理。应用 **softmax** 函数将使数值分布在0到1的范围内，且每一行的所有数之和等于1。

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

	I	am	good
I	0.90	0.07	0.03
am	0.025	0.95	0.025
good	0.21	0.03	0.76

图中的矩阵称为分数矩阵。通过这些分数，我们可以了解句子中的每个词与所有词的相关程度。以图中的分数矩阵的第一行为例，它告诉我们，I 这个词与它本身的相关程度是 90%，与 am 这个词的相关程度是7%，与 good 这个词的相关程度是 3%。

Encoder 结构 - Self-Attention 的计算过程 – 第 4 步

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

自注意力机制的最后一步是计算注意力矩阵 $\boldsymbol{Z}$ 。

注意力矩阵 $\boldsymbol{Z}$ 包含句子中每个单词的注意力值，它通过将分数矩阵 $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$ 乘以值矩阵 V 得到。

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

	I	am	good
I	0.90	0.07	0.03
am	0.025	0.95	0.025
good	0.21	0.03	0.76

•

I	67.85	91.2	…	0.13
am	13.13	63.1	…	4.44
good	12.12	96.1	…	43.4

v_1
v_2
v_3

V

值 (value) 矩阵
每行代表单词的值向量

=

I	$0.90\ v_1 + 0.07 v_2 + 0.03\ v_3$
am	$0.025 v_1 + 0.95 v_2 + 0.025\ v_3$
good	$0.21\ v_1 + 0.03 v_2 + 0.76\ v_3$

Z

注意力矩阵

从图 中可以看出，单词 I 的自注意力值 $\boldsymbol{z_1}$ 是分数加权的值向量之和： $\boldsymbol{z_1}$ = $0.90\ v_1 + 0.07 v_2 + 0.03\ v_3$
所以 $\boldsymbol{z_1}$ 的值将包含 90% 的值向量 v_1 (I)， 7% 的值向量 v_2 (am) 以及 3% 的值向量 v_3(good)。

Encoder 结构 - Self-Attention 的计算过程 – 第 4 步

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

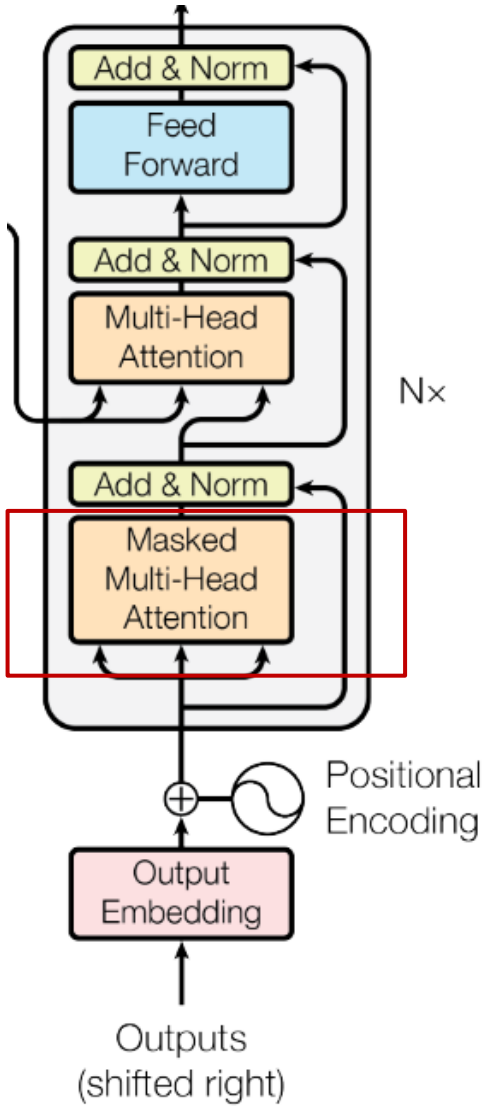
注意力矩阵 Z的作用:
让我们回过头去看之前的例句: A dog ate the food because it was hungry。在这里 it 这个词表示 dog。我们将按照前面的步骤来计算 t 这个词的自注意力值。假设计算过程如下:

$$\boldsymbol{z_{it}} = 0.0\ v_1 + 1.0 v_2 + \ldots + 0.0\ v_5 + \ldots + 0.0\ v_9$$

A dog food hungry

从图中可以看出，it 这个词的自注意力值包含 100% 的值向量 v_2 (dog)，这有助于模型理解 it 这个词实际上指的是 dog 而不是food。 这说明通过自注意力机制，我们可以了解一个词与句子中所有词的相关程度。

Decoder 结构



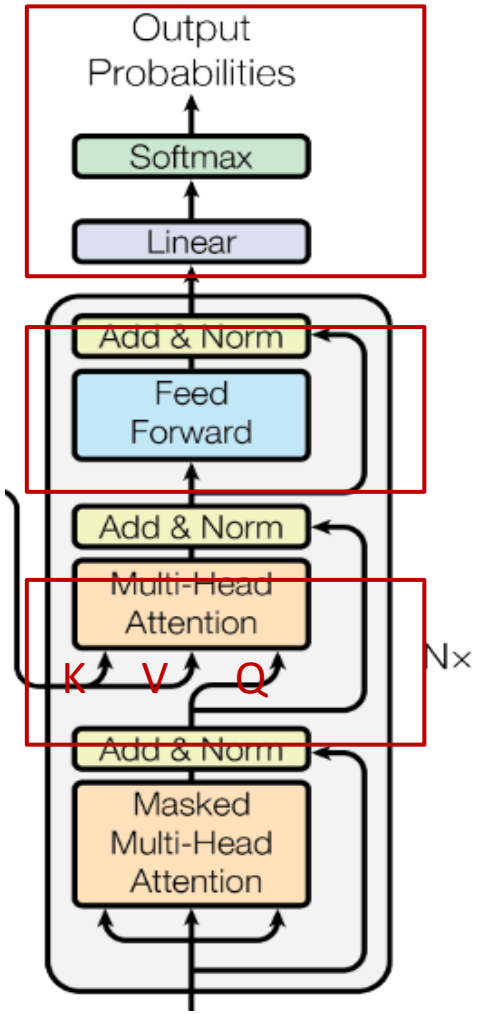
解码器层和编码器层类似，只是变成了三个子层，并且，要注意数据的QKV的取法。

Masked Multi-Head Attention 这个子层负责用来对已生成的目标端句子建模，但是是经过masked（掩盖）操作的。

因为在做机器翻译的时候，输出的词**不仅依赖于输入的词序列，还依赖于你已经预测出的词**。然而在训练时网络，我们已经知道了想要输出的完整的目标序列，这样不符合训练的要求。所以采用mask操作，在对输出的序列进行attention计算时，只用到已经预测输出的结果，而不能用未来的输出结果。

没用为什么还要输出？为了计算loss。

Decoder 结构



1. 这个子层是一个self-attention计算层。注意这里的 K，V 来自于编码层，而Q是来自于你已预测的序列。通过这个层将解码器和编码器相互连接。模拟的效果就是，在生成词的时候，能够查询到输入词的信息。

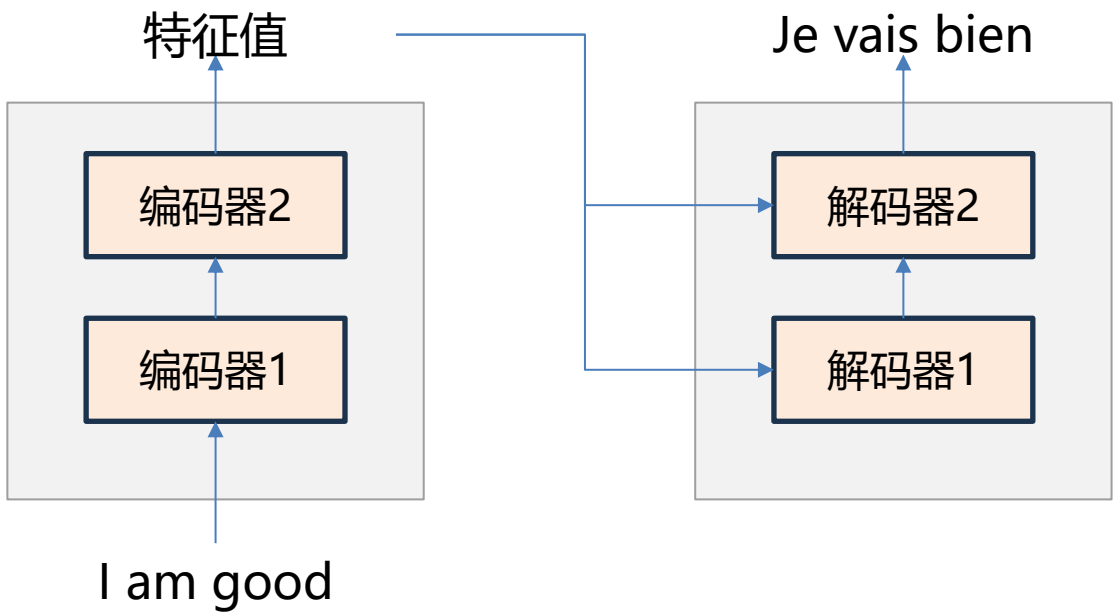
2. 这里和编码层一样，是一个全连接网络层。

3. 这是经过 6 个重叠起来的编解码器层之后，最终的输出层。它负责将最后一个解码器的输出经过一个 linear 层后变成一个概率分布，再经过 softmax，将概率分布转化为具体的概率值。之后就可以根据不同的策略来生成词。

```
class DecoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.norm_3 = Norm(d_model)

        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)
        self.dropout_3 = nn.Dropout(dropout)

        self.attn_1 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.attn_2 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)

    def forward(self, x, e_outputs, src_mask, trg_mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn_1(x2, x2, x2, trg_mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.attn_2(x2, e_outputs, e_outputs, \
        src_mask))
        x2 = self.norm_3(x)
        x = x + self.dropout_3(self.ff(x2))
        return x


class Decoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(DecoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)

    def forward(self, trg, e_outputs, src_mask, trg_mask):
        x = self.embed(trg)
        x = self.pe(x)
        for i in range(self.N):
          x = self.layers[i](x, e_outputs, src_mask, trg_mask)
      return self.norm(x)
```

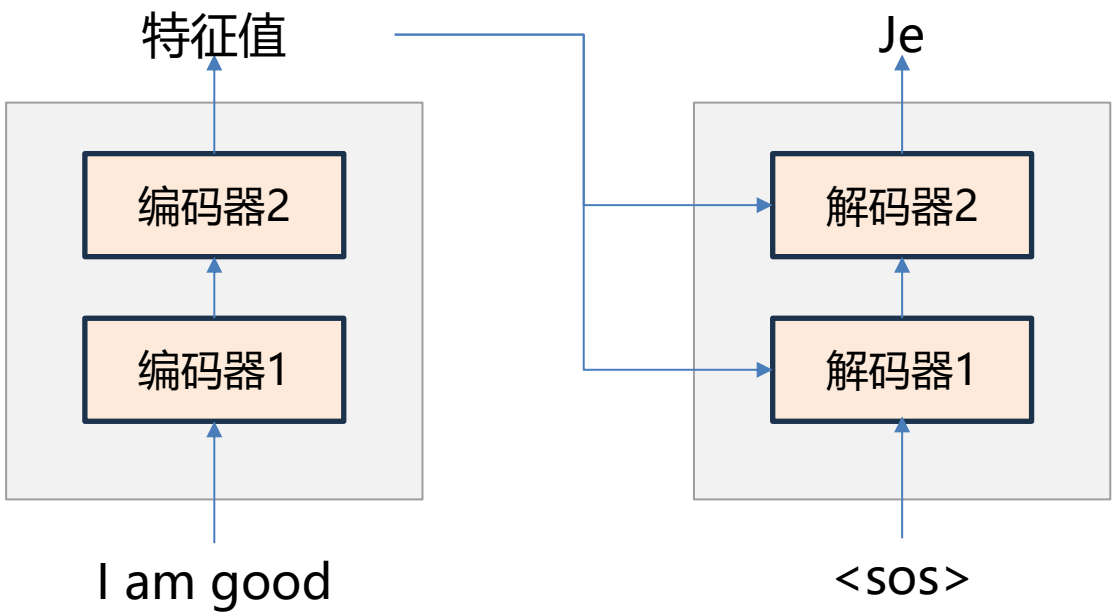
Decoder 结构 – 举例说明

在编码器部分，可以叠加 N个编码器。同理，解码器也可以有 N 个叠加在一起。为简化说明，我们设定 N=2。如图 所示，一个解码器的输出会被作为输入，传入下一个解码器。
我们还可以看到，编码器将原句的特征值 (编码器的输出) 作为输人传给所有解码器，而非只给第一个解码器。因此，一个解码器 (第一个除外) 将有两个输入：一个是来自前一个解码器的输出，另一个是编码器输出的特征值。

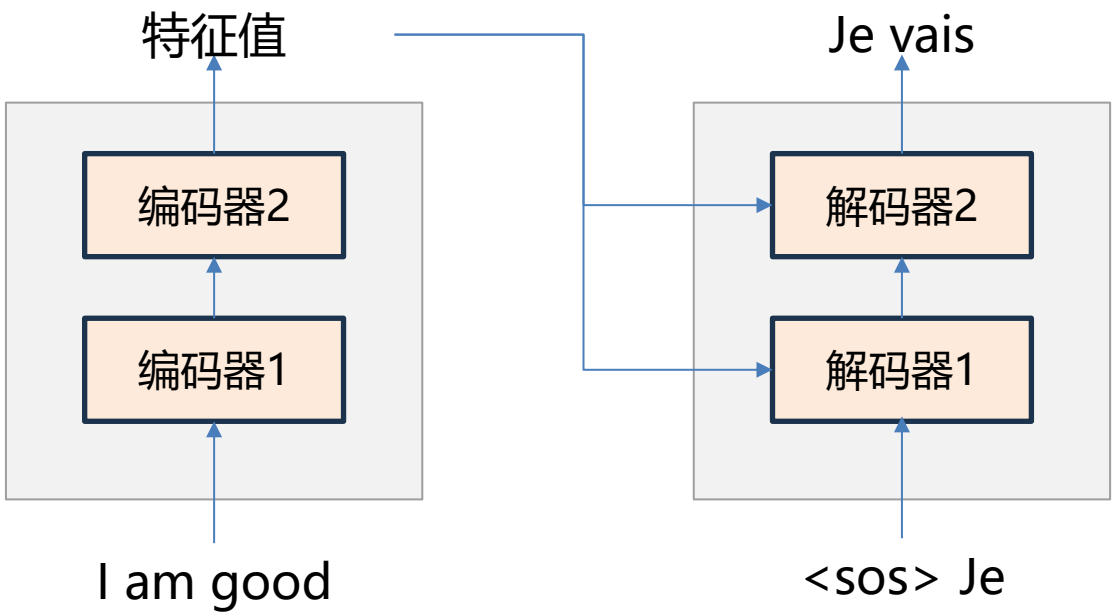


Decoder 结构 – 举例说明 - 解码器究竟是如何生成目标句

t =1时 (t表示时间步)，解码器的输入是 <sos>，这表示句子的开始。解码器收到 <sos> 作为输人，生成目标句中的第一个词，即 Je，如图下所示：

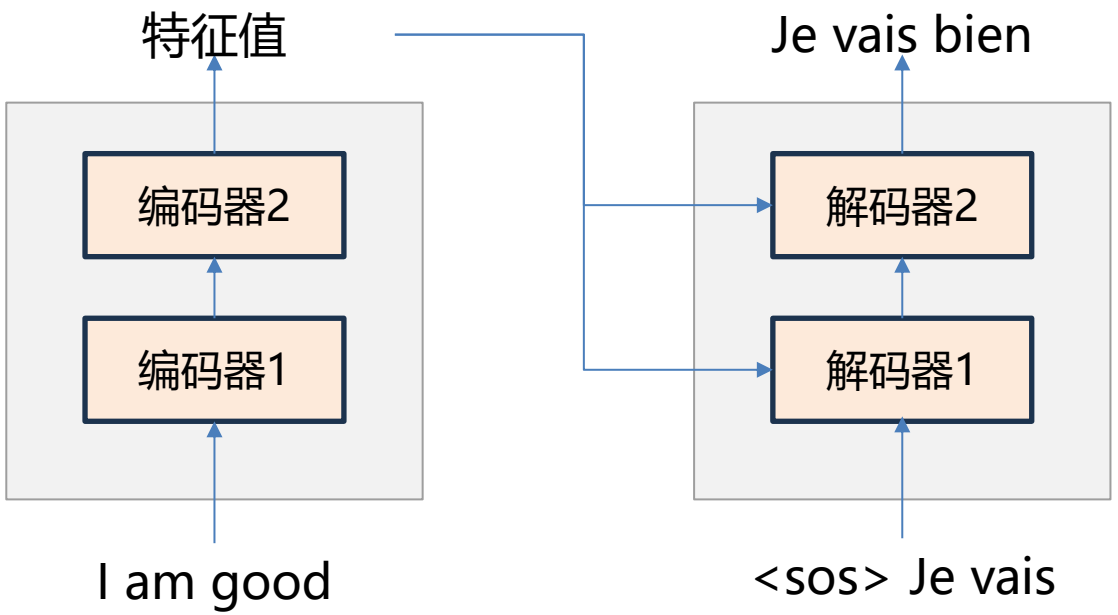


t =2时，解码器使用当前的输入和在上一步 (t-1) 生成的单词，预测句子中的下一个单词。在本例子中，解码器将 <sos>和 Je（来自上一步）作为输入，并试图生成目标句中的下一个单词，如图下所示：

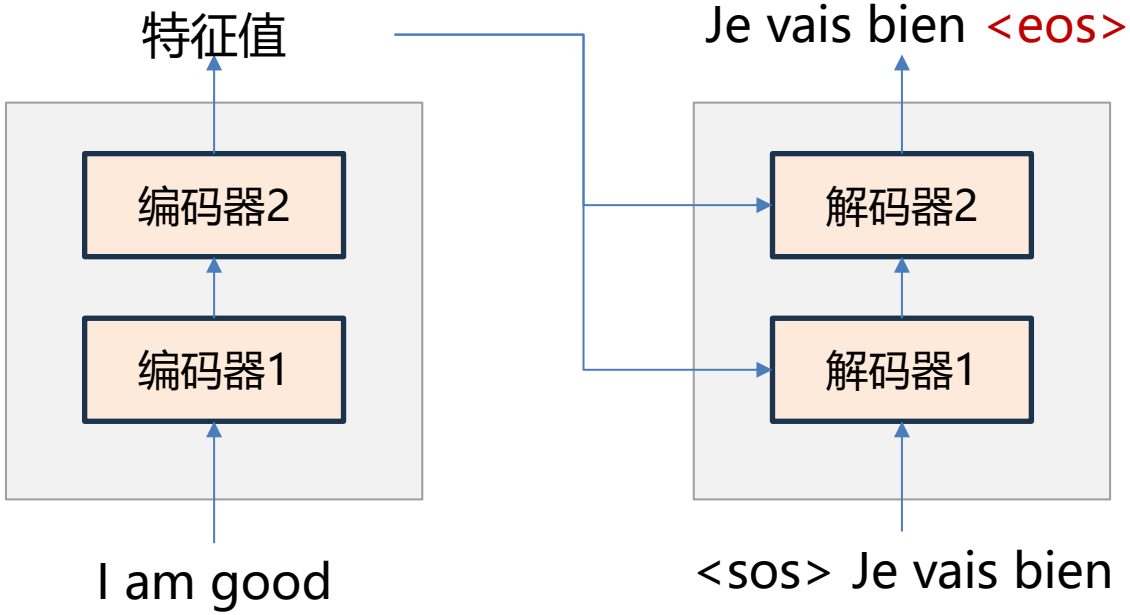


Decoder 结构 – 举例说明 - 解码器究竟是如何生成目标句

t =3时，解码器解码器将<sos>、Je 和 vais （来自上一步）作为输入，并试图生成目标句中的下一个单词，如图下所示：



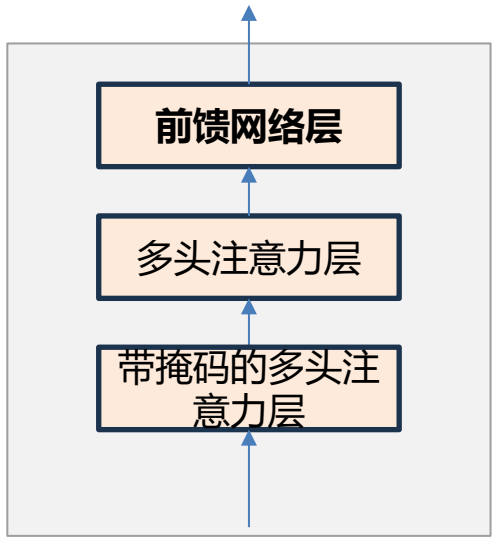
t =4时，解码器解码器将<sos>、Je 、 vais 和 bien 作为输入，并试图生成目标句中的下一个单词，如图下所示：



一旦生成表示句子结束的**<eos>标记**，就**意味着解码器已经完成了对目标句的生成工作。**

Decoder 结构 – 前馈网络层，叠加和归一化

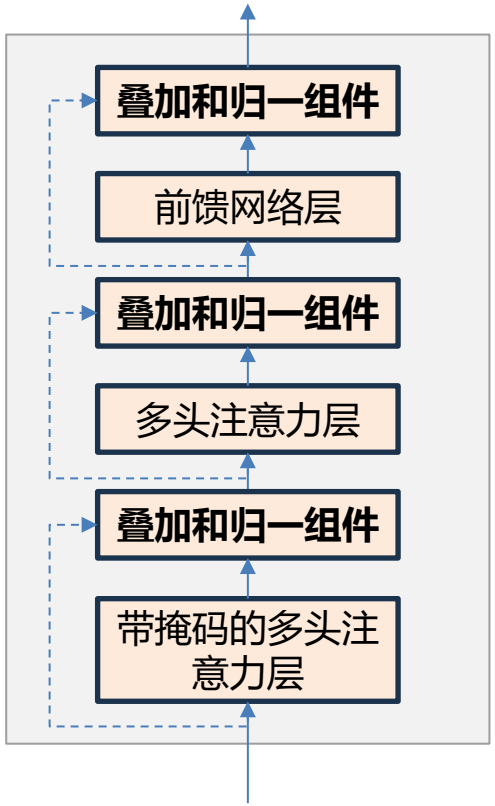
前馈网络层



解码器模块

解码器的前馈网络层的工作原理与编码器中的完全相同。

叠加和归一组件



解码器模块

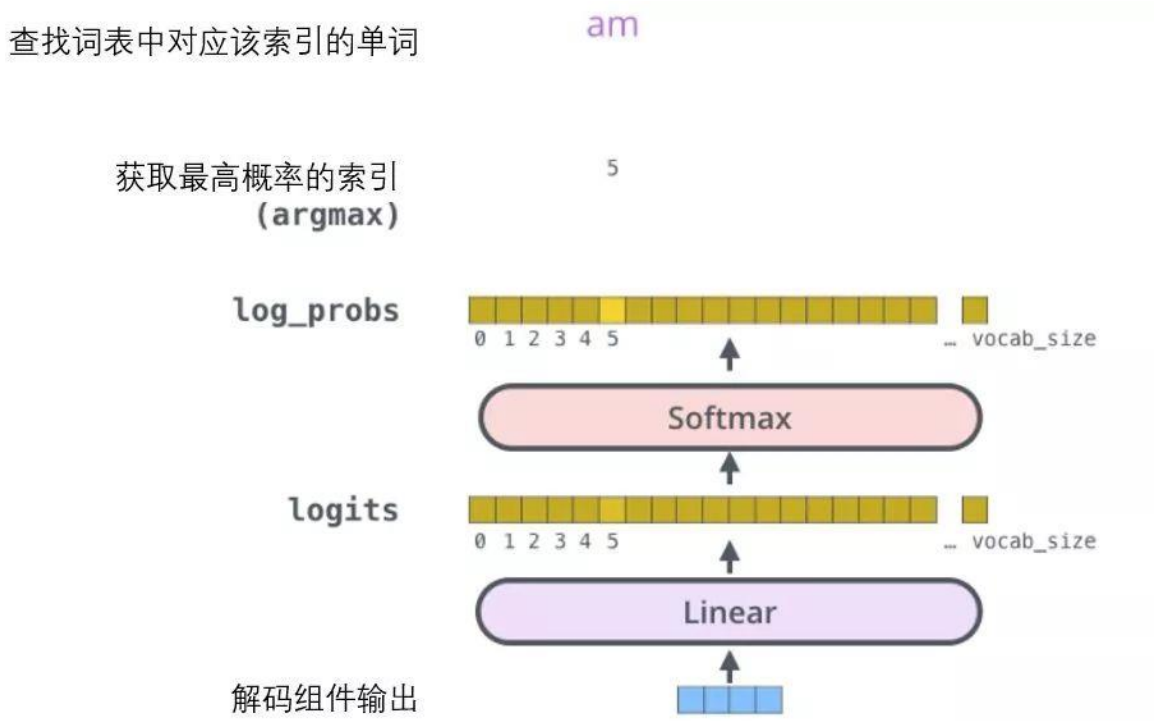
与编码器一样，叠加和归一组件链接子层的输入和输出。

Decoder 结构 – 最终的线性层和Softmax层

解码组件最后会输出一个实数向量。我们如何把浮点数变成一个单词？这便是线性变换层要做的工作，它之后就是Softmax层。

- 线性变换层是一个简单的全连接神经网络，它可以把解码组件产生的向量投射到一个比它大得多的、被称作对数几率（logits）的向量里。假设模型从训练集中学习一万个不同的英语单词（模型的“输出词表”）。因此对数几率向量为一万个单元格长度的向量——每个单元格对应某一个单词的分数。

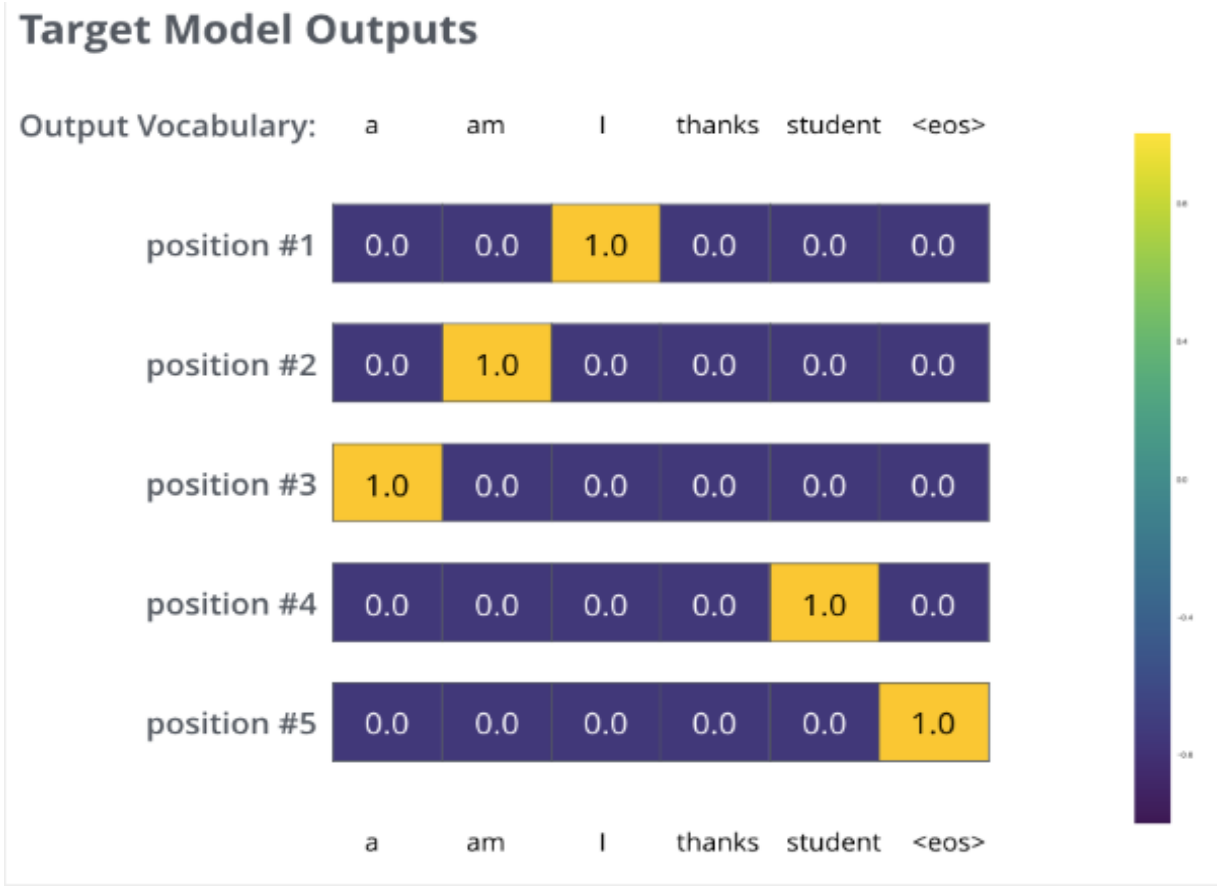
- 接下来的Softmax 层便会把那些分数变成概率（都为正数、上限1.0）。概率最高的单元格被选中，并且它对应的单词被作为这个时间步的输出。



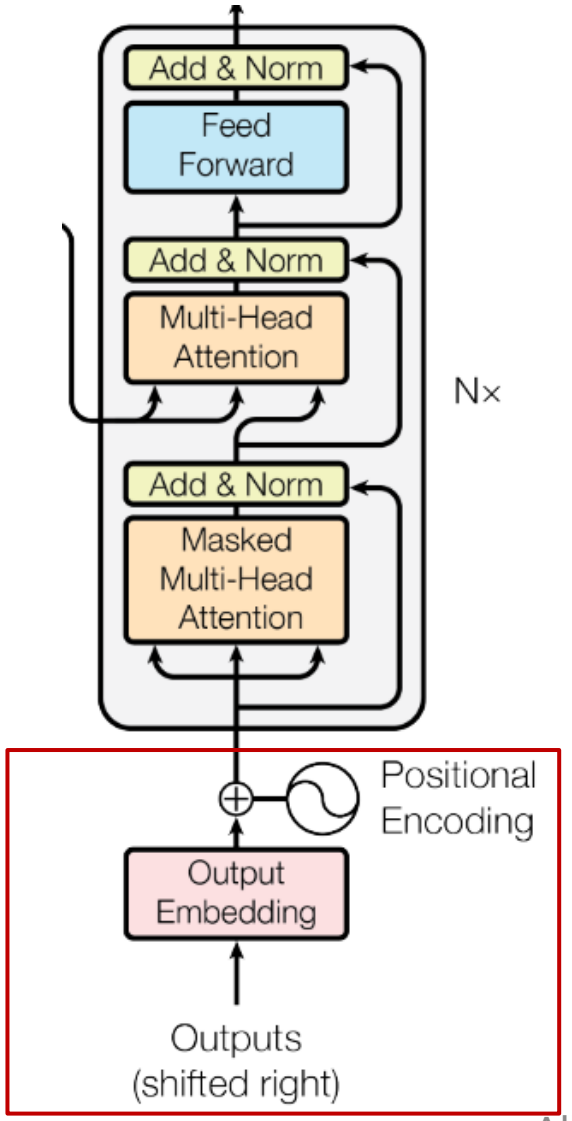
这张图片从底部以解码器组件产生的输出向量开始。之后它会转化出一个输出单词。

Token 解码

我们使用集束搜索 Beam Search 来获取可变长度的输出：



Decoder 结构 – 带掩码的多头注意力层 – 输入

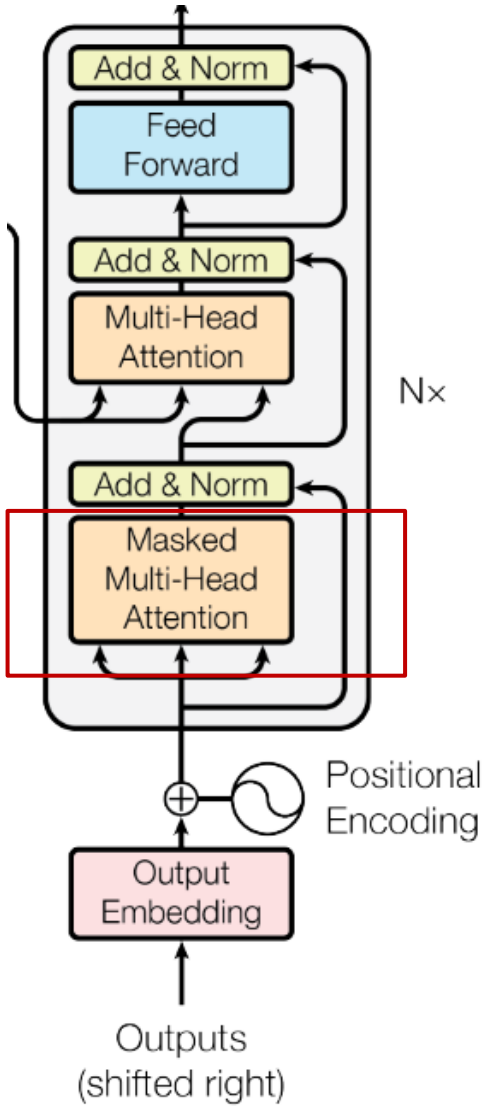


在训练期间，由于有正确的目标句，解码器可以直接将整个目标句稍作修改作为输入，我们只需要将<sos> 标记添加到目标句的开头，再将整体作为输入发送给解码器。
首先，我们不是将输入直接送入解码器，而是将其转换为嵌入矩阵并添加位置编码，然后送入解码器，从而得到下图的 X：

X =

<sos>	7.9	3.5	...	16.1	x_1
Je	8.1	4.4	...	83.1	x_2
Vais	17	0.54	...	6.12	x_3
bien	11.12	11.12	...	22.1	x_4

Decoder 结构 – 带掩码的多头注意力层 – 掩码



在测试期间，解码器只将上一步生成的词作为输入，比如：
当 t= 2 时，解码器中只有[<sos>, Je]，并没有任何其他词。因此，我们也需要同样的方式训练模型。模型的注意力机制应该只与该词之前的单词有关，而不是其后的单词。要做到这一点，我们可以掩盖后边所有还没有被模型预测的词。

当我们想预测与 <sos> 相邻的残次，模型应该只看到<sos> ，所以我们应该掩盖<sos> 后边的所有词，其他词同理：

<sos>	掩码	掩码	掩码
<sos>	Je	掩码	掩码
<sos>	Je	vais	掩码
<sos>	Je	vais	bien

Decoder 结构 – 带掩码的多头注意力层 – 掩码

对于一个注意力头 i 的注意力矩阵 Z_i 的计算方法，公式如下：

$$Z_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

计算注意力矩阵第**1**步，计算查询矩阵与键矩阵的点积 $Q_i K_i^T$：

$Q_i K_i^T$ =

	<sos>	Je	Vais	bien
<sos>	73	60	10	45
Je	40	99	25	70
Vais	58	40	83	10
bien	12	11	15	80

计算注意力矩阵第**2**步，将 $Q_i K_i^T$ 矩阵除以键向量维度的平方根 $\sqrt{d_k}$：

$\frac{Q_i K_i^T}{\sqrt{d_k}}$ =

	<sos>	Je	Vais	bien
<sos>	9.125	7.5	1.25	5.625
Je	5.0	12.37	3.12	8.75
Vais	7.25	5.0	10.37	1.25
bien	1.5	1.37	1.87	10.0

Decoder 结构 – 带掩码的多头注意力层 – 掩码

计算注意力矩阵第**3**步，在做softmax前，我们需要对数值进行掩码转换，以第一行为例，为例预测<sos>后边的词，模型不应该知道<sos>右边的所有词。因此，我们可以用-∞掩盖<sos>右边的所有词：

$\frac{Q_i K_i^T}{\sqrt{d_k}}$ =

	<sos>	Je	Vais	bien
<sos>	9.125	-∞	-∞	-∞
Je	5.0	12.37	3.12	8.75
Vais	7.25	5.0	10.37	1.25
bien	1.5	1.37	1.87	10.0

同理，我们可以用-∞掩盖后续输入词右边的所有词：

$\frac{Q_i K_i^T}{\sqrt{d_k}}$ =

	<sos>	Je	Vais	bien
<sos>	9.125	-∞	-∞	-∞
Je	5.0	12.37	-∞	-∞
Vais	7.25	5.0	10.37	-∞
bien	1.5	1.37	1.87	10.0

现在，我们可以将 softmax 函数应用于前面的矩阵，并将结果与矩阵 V_i ，即： $\text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$。

同样，我们可以计算 h 个注意力矩阵，将他们串联起来，并将结果乘以新的权重矩阵$\boldsymbol{W_0}$，即可得到最终的注意力矩阵$\boldsymbol{M}$：

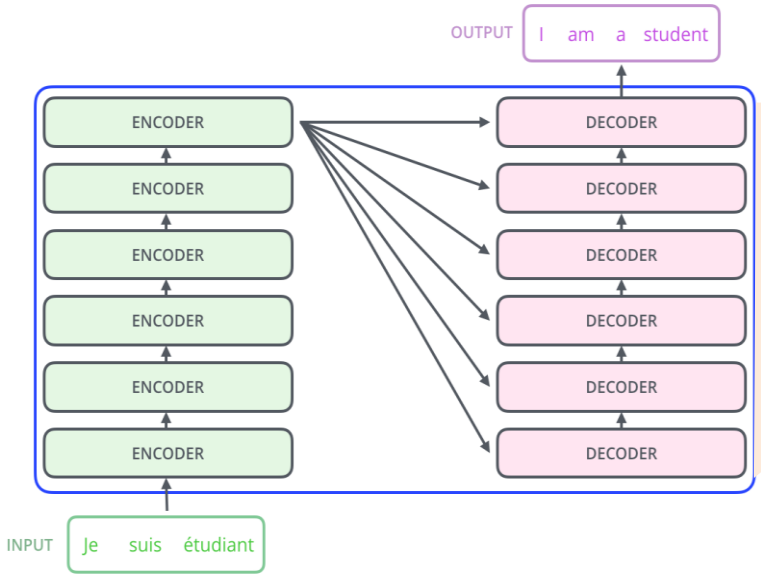
$$\boldsymbol{M} = \text{Concatenate}(\boldsymbol{Z_1}, \boldsymbol{Z_2}, \ldots, \boldsymbol{Z_i}, \ldots, \boldsymbol{Z_h})\, \boldsymbol{W_0}$$

Decoder 结构 – 多头注意力层

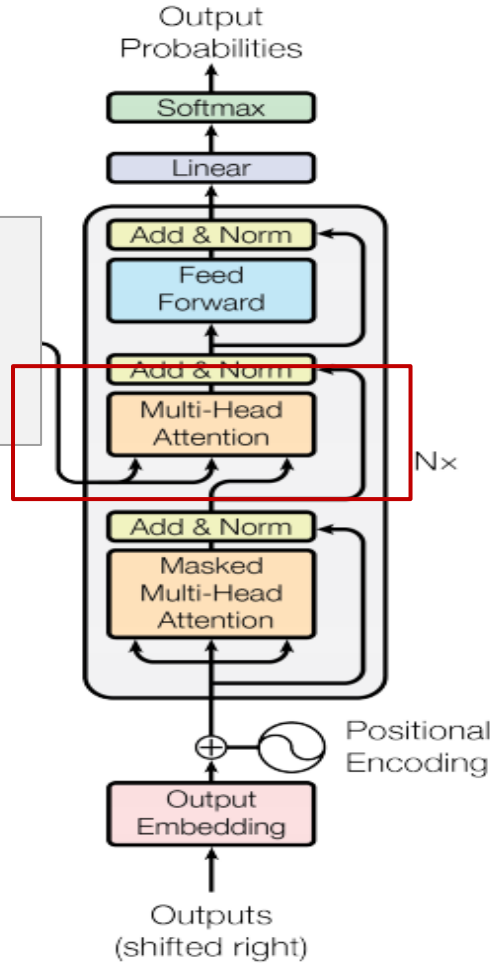
每个解码器多种的多头注意力层都有**两个输入**:

- 一个来自带掩码的多头注意力层, 用 M 来表示。
- 另一个是编码器输出的特征值, 用 R来表示。

由于涉及编码器与解码器的交互，因此这一层也被称为**编码器-解码器注意力层。**



编码器-解码器多头注意力:
键（Key）和值（Value）是最终**编码层**的输出创建。
查询（Query）是前一个**解码层**的输出创建。



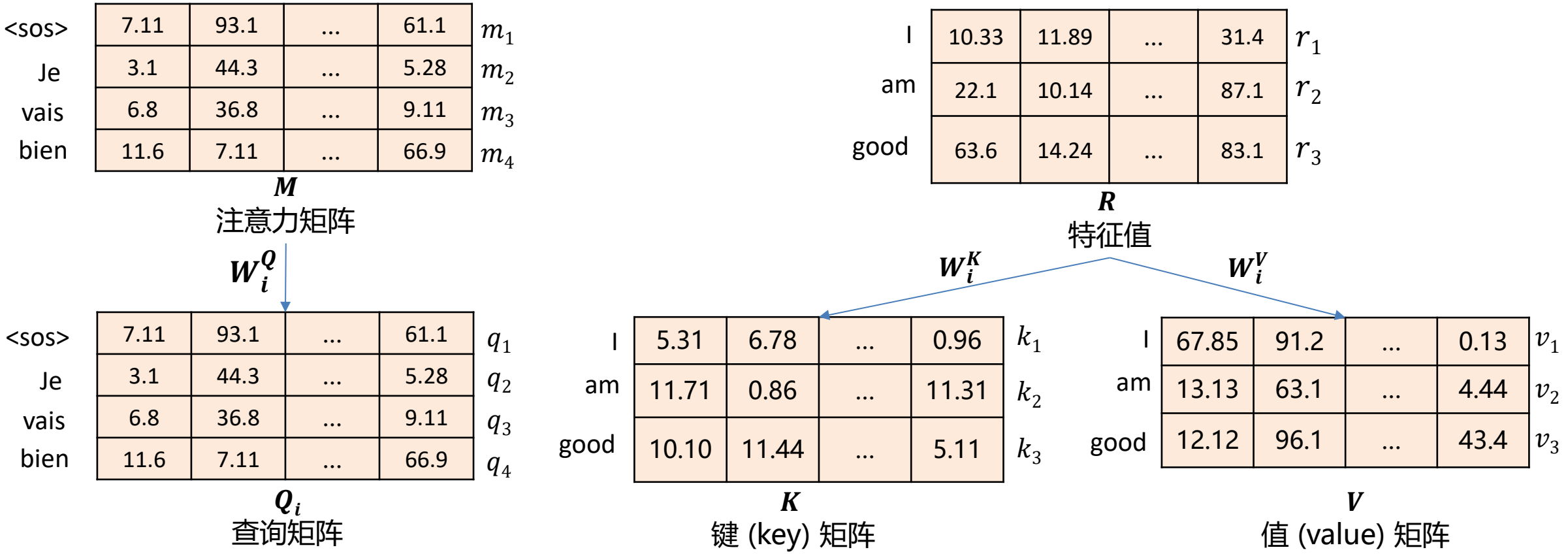
编码器将其最终的(K, V)输出发送给每个解码器层。

Decoder 结构 – 多头注意力层

我们使用从上一个子层获得的注意力矩阵 $\boldsymbol{M}$ 创建查询矩阵 $\boldsymbol{Q}$，使用编码器输出的特征值 $\boldsymbol{R}$ 创建键矩阵和值矩阵，
由于采用多头注意力机制，因此对于头 $\boldsymbol{i}$，需要如下处理：

- 查询矩阵 $\boldsymbol{Q_i}$ 通过将注意力矩阵 $\boldsymbol{M}$ 乘以权重矩阵 $\boldsymbol{W_i^Q}$ 来创建。
- 键矩阵和值矩阵通过将编码器输出的特征值 $\boldsymbol{R}$ 分别于权重矩阵 $\boldsymbol{W_i^K}$ 、 $\boldsymbol{W_i^V}$ 相乘来创建。

因为查询矩阵是从$\boldsymbol{M}$ 求得，所以本质上包含了目标句的特征。键矩阵和值矩阵则含有原句的特征，因为他们是用$\boldsymbol{R}$ 计算的。



Decoder 结构 – 多头注意力层 - Self-Attention 的计算过程 –（1/2）

第一步计算查询矩阵与键矩阵的点积：

$$Z_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

<sos>	7.11	93.1	...	61.1
Je	3.1	44.3	...	5.28
vais	6.8	36.8	...	9.11
bien	11.6	7.11	...	66.9

q_1 q_2 q_3 q_4

$\boldsymbol{Q_i}$ 查询矩阵

•

I	am	good
5.31	11.71	10.10
6.78	0.86	11.44
...	...	...
0.96	11.31	5.11

k_1 k_2 k_3

$\boldsymbol{K_i^T}$

=

	I	am	good
<sos>	$q_1 \cdot k_1$	$q_1 \cdot k_2$	$q_1 \cdot k_3$
Je	$q_2 \cdot k_1$	$q_2 \cdot k_2$	$q_2 \cdot k_3$
vais	$q_3 \cdot k_1$	$q_3 \cdot k_2$	$q_3 \cdot k_3$
bien	$q_4 \cdot k_1$	$q_4 \cdot k_2$	$q_4 \cdot k_3$

$\boldsymbol{Q_i \bullet K_i^T}$ =

	I	am	good
<sos>	91	60	77
Je	96	63	12
vais	41	111	48
bien	36	45	65

Decoder 结构 – 多头注意力层 - Self-Attention 的计算过程 – （2/2）

自注意力计算的第2步，是将 $\boldsymbol{Q_i \cdot K_i^T}$ 矩阵除以键向量维度的平方根。然后应用 **softmax** 函数对其进行归一化处理。得到分数矩阵 $\text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$。接下来，将分数矩阵乘以值矩阵 V_i，得到 $\text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$，即注意力矩阵 Z_i，如下图：

$$Z_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

$\boldsymbol{Z_i}$ =

	I	am	good
<sos>	0.84	0.017	0.14
Je	0.98	0.02	0.0
vais	0.0	1.0	0.0
bien	0.0	0.0	1.0

$\text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$

•

I	67.85	91.2	…	0.13
am	13.13	63.1	…	4.44
good	12.12	96.1	…	43.4

V_i

=

<sos>	0.84 v_1 + 0.017v_2 + 0.14 v_3
Je	0.98v_1 + 0.02v_2 + 0.0 v_3
vais	0.0 v_1 + 1.0v_2 + 0.0 v_3
bien	0.0 v_1 + 0.0v_2 + 1.0 v_3

$\boldsymbol{Z_i}$

$\boldsymbol{Z_i}$ =

z_1	<sos>
z_2	Je
z_3	vais
z_4	bien

从图 中可以看出，单词 Je 的自注意力值 $\boldsymbol{z_2}$ 是分数加权的值向量之和计算的：

$$\boldsymbol{z_2} = 0.98v_1 + 0.02v_2 + 0.0\ v_3$$

所以 $\boldsymbol{z_2}$ 的值将包含 98% 的值向量 v_1 (I)，2% 的值向量 v_2 (am)。

同样，我们可以计算 h 个注意力矩阵，将他们串联起来，并将结果乘以新的权重矩阵$\boldsymbol{W_0}$, 即可得到最终的注意力矩阵：

Multi-head attention = Concatenate($\boldsymbol{Z_1}, \boldsymbol{Z_2}, \ldots, \boldsymbol{Z_i}, \ldots, \boldsymbol{Z_h}$) $\boldsymbol{W_0}$

llama2 7B 举例分析

从transformer的结构图可见，transformer可以分成2部分，encoder和decoder，而llama2只用了tranformer的decoder部分，是decoder-only结构。目前大部分生成式的语言模型都是采用这种结构。

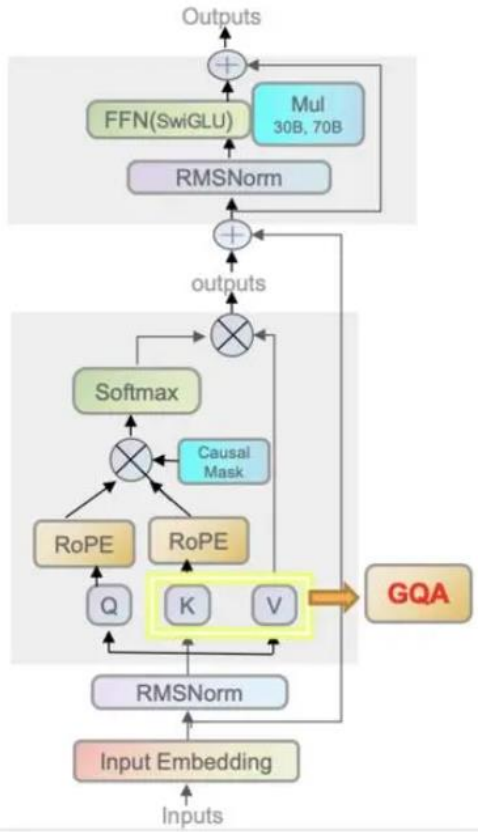
为什么大部分生成式的语言模型采用decoder-only:

- 训练效率和工程实现上的优势。
- decoder only因为用了masked attention，是一个下三角矩阵，attention一定是满秩的。其它架构attention在n×d的矩阵与d×n的矩阵相乘后再加softmax（n≫d），这种形式的Attention的矩阵因为低秩问题而带来表达能力的下降。

相比于 Llama 1 ，Llama 2 的训练数据多了 40%，上下文长度也翻倍，并采用了分组查询注意力机制。具体来说，Llama 2预训练模型是在2万亿的 token上训练的，精调 Chat 模型是在100 万人类标记数据上训练的。

模型结构特点:

- **MHA改成GQA**：整体参数量会有减少
- **FFN模块矩阵维度有扩充**：增强泛化能力，整体参数量增加

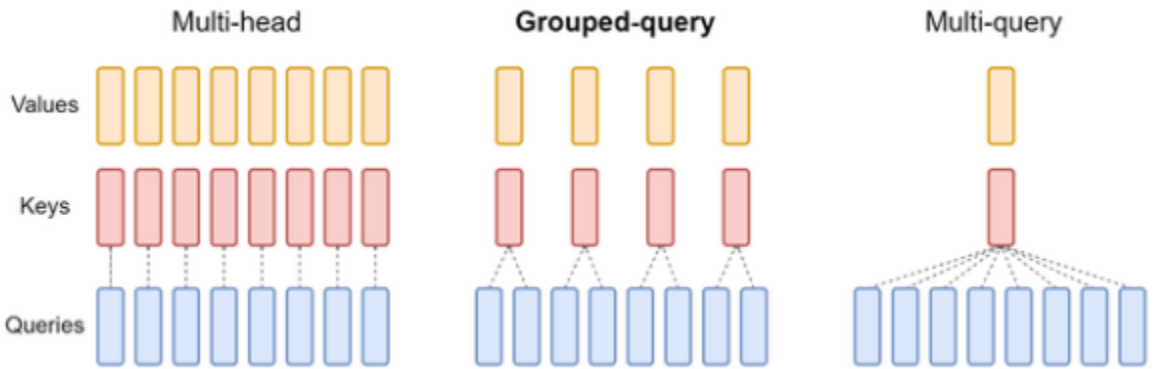


LLaMA: Open and Efficient Foundation Language Models
https://arxiv.org/pdf/2302.13971.pdf

llama2 7B 举例分析 - GQA

GQA和MQA都是注意力的变体，其中多个查询头关注相同的键和值头，以减少推理过程中 KV 缓存的大小，并可以显著提高推理吞吐量。

- **MHA**（Multi-head Attention）是标准的多头注意力机制，h个Query、Key 和 Value 矩阵。
- **MQA**（Multi-Query Attention）是多查询注意力的一种变体，也是用于自回归解码的一种注意力机制。与MHA不同的是，MQA 让所有的头之间共享同一份 Key 和 Value 矩阵，每个头只单独保留了一份 Query 参数，从而大大减少 Key 和 Value 矩阵的参数量。
- **GQA**（Grouped-Query Attention）是分组查询注意力，GQA将查询头分成G组，每个组共享一个Key 和 Value 矩阵。GQA-G是指具有G组的grouped-query attention。GQA-1具有单个组，因此具有单个Key 和 Value，等效于MQA。而GQA-H具有与头数相等的组，等效于MHA。



性能对比：

- Mutil-Head Attention 因为自回归模型生成回答时，需要前面生成的KV缓存起来，来加速计算。
- Multi-Query Attention 多个头之间可以共享KV对，因此速度上非常有优势，实验验证大约减少30-40%吞吐。
- Group Query Attention 没有像MQA那么极端，将query分组，组内共享KV，效果接近MQA，速度上与MQA可比较。

llama2 7B 举例分析

- d，可以表示为 d_{head}，是单个注意力头的维度。
  - 对于 7B d = 128

- n_{heads} 是注意力头的数量
  - 对于 7B n_{heads} = 32

- n_layers是注意力块出现的次数
  - 对于 7B n_{layers} = 32

- d_{model}，是模型的维度。 $d_{\text{model}} = n_{\text{heads}} \cdot d$ = 32 * 128
  - 对于 7B d_{model} = 4096

Llama 2的其他尺寸具有较大的 d_{model} （请参阅“dimension”列）。

params	dimension	n heads	n layers
6.7B	4096	32	32
13.0B	5120	40	40
32.5B	6656	52	60
65.2B	8192	64	80

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

$$S = \mathrm{Q}K^T$$

$$P = softmax(S)\ (\text{系数}\frac{1}{\sqrt{d_k}}\text{不影响复杂度计算})$$

$$\mathrm{O} = PV$$

llama2 7B 注意力计算过程

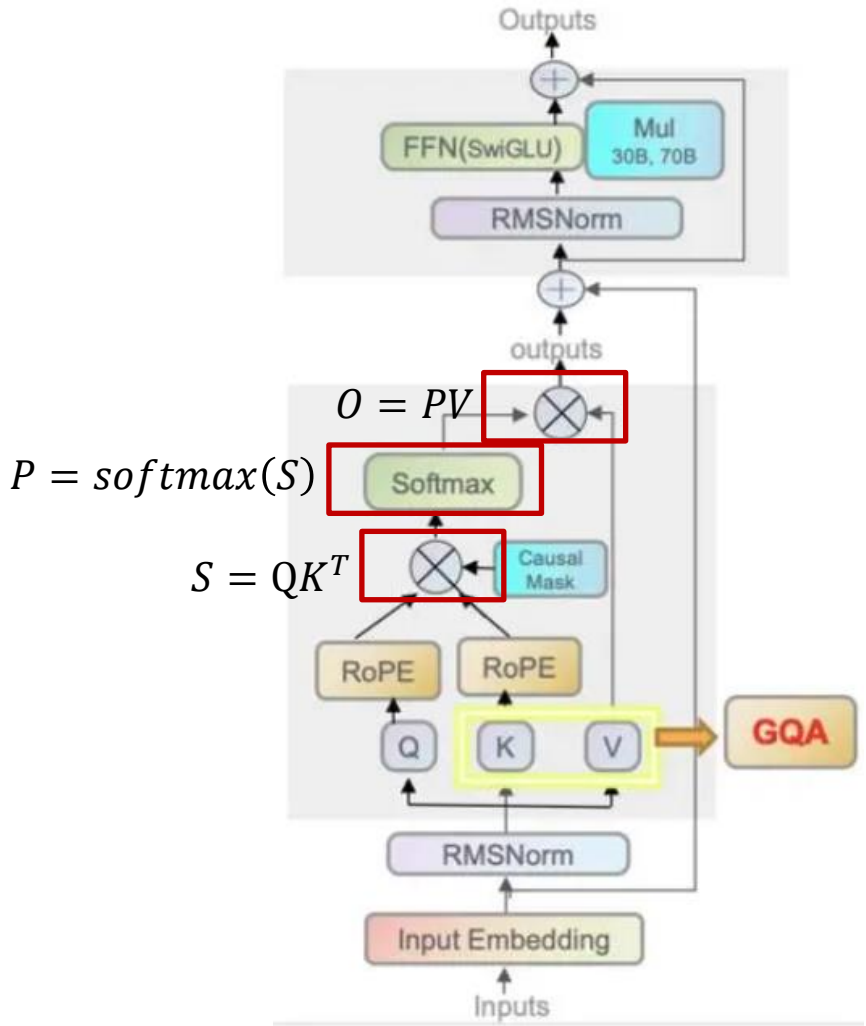
$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

(系数$\frac{1}{\sqrt{d_k}}$不影响复杂度计算，忽略)

Q，K，V $\in \mathbb{R}^{Nxd}$ 保存于 HBM (high bandwidth memory)：

1. 从HBM 读取 Q, K，计算 $S = QK^T$, 然后将 S 回写至 HBM
2. 从HBM 读取 S，计算 $P = softmax(S)$，然后将 P 回写至 HBM
3. 从HBM 读取 P, V，计算 $O = PV$, 然后将 O 回写至 HBM
4. 返回 O

每次执行 GPU 内核时，我们都需要将数据从 GPU 的 DRAM（HBM）移出或移回。



llama2 7B 注意力计算过程

size_Q，size_K，size_V，size_O = $d_{\text{model}} * d$
size_S，size_P = d_{model}^2

行号	读高速缓存（bytes）	运算次数（ops）	写高速缓存（bytes）
1	size_fp16 * (size_Q + size_K)	cost_dot_product_QK * size_S	size_fp16 * size_S
	= 2 * 2 * ($d_{\text{model}} * d$)	= (2 * d) * d_{model}^2	= 2 * d_{model}^2
2	size_fp16 * size_S	cost_softmax * size_P	size_fp16 * size_P
	= 2 * d_{model}^2	= 3 * d_{model}^2	= 2 * d_{model}^2
3	size_fp16 * (size_P + size_V)	cost_dot_product_PV * size_O	size_fp16 * size_O
	= 2 * (d_{model}^2 + ($d_{\text{model}} * d$))	= (2 * d_{model}) * ($d_{\text{model}} * d$)	= 2 * ($d_{\text{model}} * d$)

第1列 + 第3列，合计高速缓存读写 total_memory_movement_in_bytes:

= (2 * 2 * (d_{model} * d)) + (2 * d_{model}^2) + (2 * (d_{model}^2 + (d_{model} * d))) + (2 * d_{model}^2) + (2 * d_{model}^2) + (2 * (d_{model} * d))

= 8 d_{model}^2 + $d_{\text{model}} * d$ bytes

第2列 浮点计算次数 total_compute_in_floating_point_ops:

= ((2 * d) * d_{model}^2) + (3 * d_{model}^2) + ((2 * d_{model}) * (d_{model} * d))

= 4 $d_{\text{model}}^2 \cdot d$ + 3 d_{model}^2 ops

计算强度 intensity = total_compute_in_floating_point_ops / total_memory_movement_in_bytes :

= 4d d_{model}^2 + 3 d_{model}^2 ops / 8 d_{model}^2 + 8 $d_{\text{model}} * d$ bytes

= 62 ops/byte

llama2 KV 缓存

在半精度（FP16）下，每个浮点数需要2个字节来存储。有2个矩阵，为了计算KV缓存大小，我们将两者都乘以n_layers和d_model，得到以下方程：

每个token kv缓存 kv_cache_size：

= (2 * 2 * n_{layers} * d_{model})

= (2 * 2 * n_{layers} * n_{heads} * d)

= (4 * 32 * 4096)

= 524288 **bytes/token**

~ 0.00052 GB/token

kv_cache_tokens

= 10 GB / 0.00052 GB/token

= 19,230 tokens

假设显存加载模型后，至少仍有 10 GB可用，则我们的KV缓存可以轻松容纳19,230个 token。因此，对于Llama 2的标准序列长度4096个 token。

为了充分利用计算能力，我们可以在推理期间每次批处理 4 个请求，以填满 KV 缓存。这将增加我们的吞吐量。

如果使用LLM（语言模型）来异步处理大量文档队列，批处理是一个很好的主意。与逐个处理每个元素相比，将更快地处理队列，并且可以安排推理调用以快速填充批次，从而最大程度地减少对延迟的影响。

llama2 KV 缓存

第一个因子2 表示 k 和 v 这两个向量。在每一层中我们都要存储这些k，v向量，在半精度（FP16）下，每个浮点数需要2个字节来存储。为了计算KV缓存大小，我们将两者都乘以 n_layers 和 d_model，得到以下方程：

每个token kv缓存 kv_cache_size：

$$= (2 * 2 * n_{\text{layers}} * d_{\text{model}})$$

$$= (2 * 2 * n_{\text{layers}} * n_{\text{heads}} * d)$$

llama2 KV 缓存

Llama2 模型使用一种称为分组查询注意（GQA）的注意力变体。当 KV 头数为 1 时，GQA 与 Multi-Query-Attention (MQA) 相同。

GQA 通过共享键/值来帮助缩小 KV 缓存大小。KV缓存大小的计算公式为：
batch_size * seqlen * (d_model * n_kv_heads / n_heads) * n_layers * 2 (K and V) * 2 (bytes per Float16)

批量大小	GQA KV 高速缓存 (FP16)	GQA KV 缓存 (Int8)
1	0.312 GiB	0.156 GiB
16	5GiB	2.5GiB
32	10GiB	5GiB
64	20GiB	10GiB

序列长度为 1024 时 Llama-2-70B 的 KV 缓存大小

参考材料

1. <<BERT基础教程：Transformer大模型实战>>
[印] 苏达哈尔桑·拉维昌迪兰（Sudharsan Ravichandiran） 著，周参 译


Thank
You