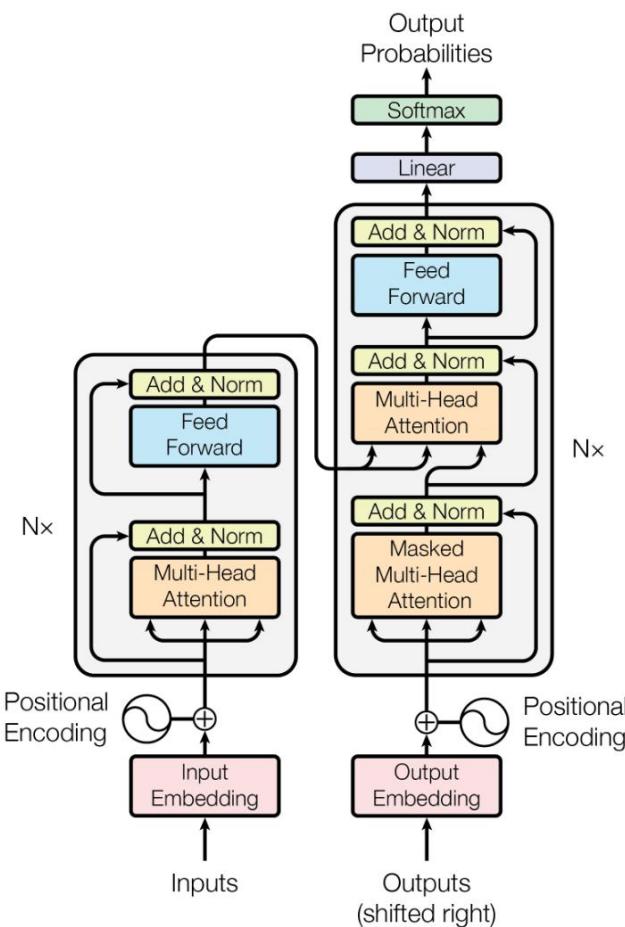
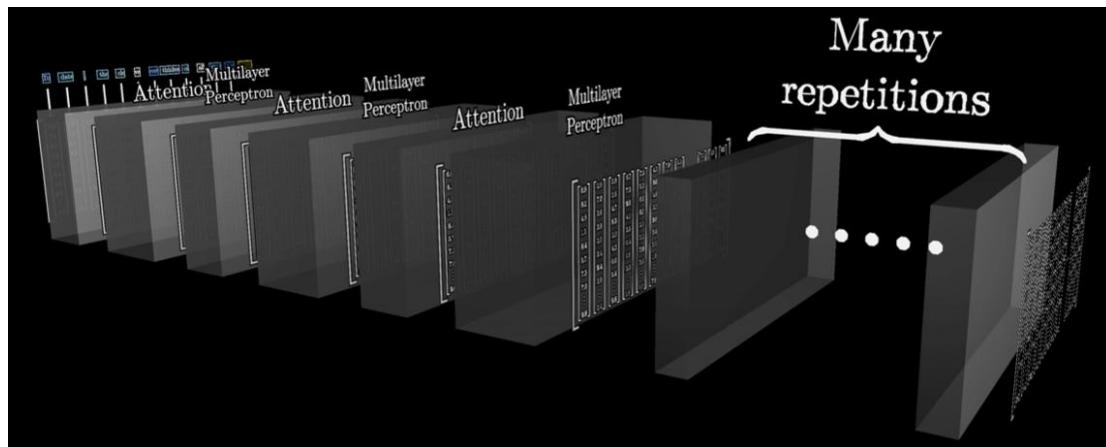


Transformer 机制总结（适合有基础者）



encoder(左侧)-decoder(右侧)架构



轮流通过 多个 Attention 和 MLP 多层感知机

在深度学习中，经常使用卷积神经网络（CNN）或循环神经网络

(RNN) 对序列进行编码。想象一下，有了注意力机制之后，我们将词元序列输入注意力池化中，以便同一组词元同时充当查询、键和值。具体来说，每个查询都会关注所有的键-值对并生成一个注意力输出。由于查询、键和值来自同一组输入，因此被称为 **自注意力** (self-attention) ([Lin et al., 2017, Vaswani et al., 2017](#))，也被称为 **内部注意力** (intra-attention) ([Cheng et al., 2016, Parikh et al., 2016, Paulus et al., 2017](#))。本节将使用自注意力进行序列编码，以及如何使用序列的顺序作为补充信息。

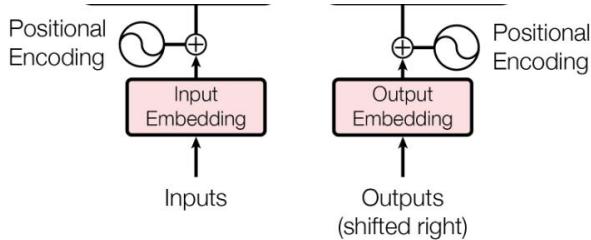
11.6.1. Self-Attention

给定一个由词元组成的输入序列 $\mathbf{x}_1, \dots, \mathbf{x}_n$, 其中任意 $\mathbf{x}_i \in \mathbb{R}^d$ ($1 \leq i \leq n$)。该序列的自注意力输出为一个长度相同的序列 $\mathbf{y}_1, \dots, \mathbf{y}_n$, 其中:

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \dots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d \quad (10.6.1)$$

根据 (10.2.4) 中定义的注意力汇聚函数 f 。下面的代码片段是基于多头注意力对一个张量完成自注意力的计算, 张量的形状为 (批量大小, 时间步的数目或词元序列的长度, d)。输出与输入的张量形状相同。

1.Embedding Layer

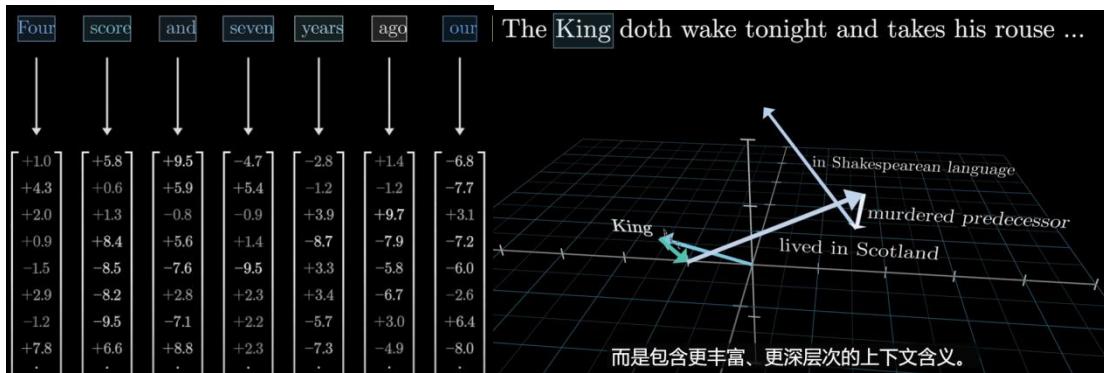


定义: embedding 层 (嵌入层) 是把离散的输入 (如 token id、词编号) 映射到连续的、高维的稠密向量空间的操作。

Embedding 层就是一个查表 (lookup table) 操作, 把一个个词或 token 变成一串浮点数向量。

假设 token 都是一个完整的单词, 一个词组。

把离散符号/word → 连续向量。便于矩阵运算。嵌入向量形成高维空间。



只靠 Self-Attention 是无感知顺序的。

Embedding 可以学习出: 单词之间的语义关系; 句子结构的隐含特征

在Transformer标准实现中, **embedding是三部分加和的**:

组件	说明
Token Embedding	词或子词的向量表示 (最主要的)
Positional Embedding	位置编码 (告诉模型token是第几个)
Segment Embedding (可选)	例如BERT里, 有时用来区分不同句子

Input = TokenEmbedding(x) + PositionalEmbedding(x)

Token Embedding 把词/子词/字符转成向量表示 (编码语义特征)

Positional Embedding 加入位置信息 (编码序列顺序)

第一部分: Token Embedding (词嵌入)

假设你的词表 (Vocabulary) 大小是 V , 每个token需要映射成 d_{model} 维的向量。

那么定义一个**嵌入矩阵**:

$$E \in \mathbb{R}^{V \times d_{\text{model}}}$$

对于一个输入token id i , Token Embedding就是:

$$\text{TokenEmbedding}(i) = E[i]$$

- 就是简单地"查表": 取第 i 行, 得到一行向量。
- TokenEmbedding 本质是个大表, 每个 id 就是一行向量。
- 输出形状是 ($\text{batch_size}, \text{seq_len}, d_{\text{model}}$)

名称	具体意义
batch_size	一次送进模型处理的样本数量。例如: 32句话一起处理
seq_len	每个样本的token数量 (即序列长度)。例如一句话有128个token
embed_dim	每个token被嵌入后的向量维度。例如, 每个token用768维向量表示

embed_dim 就是 d_{model} 即 d , 它们指的是同一件事: 每个 token 的向量维度所有 token 的嵌入向量、位置编码向量、Attention 中的 Query/Key/Value 向量、前馈网络输入输出等的 **统一维度大小**。

目的: 整个网络中所有子模块之间 **输入输出维度保持一致** (都是 d_{model}), 这样模块可以堆叠、加和、残差连接。

第二部分: Positional Embedding (位置编码)

在处理词元序列时, 循环神经网络是逐个的重复地处理词元的, 而自注意力则因为并行计算而放弃了顺序操作。为了使用序列的顺序信息, 通过在输入表示中添加 位置编码 (positional encoding) 来注入绝对的或相对的位置信息。位置编码可以通过学习得到也可以直接固定得到。接下来描述的是基于正弦函数和余弦函数的固定位置编码 ([Vaswani et al., 2017](#))。

类型	方式	示例
固定式位置编码	正弦-余弦 (sin/cos) 公式计算	原始Transformer (Vaswani 2017)
可学习的位置编码	位置向量是参数可学习	GPT, BERT, 现代大部分用

(a) 固定式位置编码 (sin/cos公式)

每个位置 pos 对应一个 d_{model} 维向量。

公式：

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

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

- pos : 位置索引 (比如第5个词)
- i : 当前维度索引 (比如第20个特征)
- 奇数维度用sin, 偶数维度用cos。

特点：

- 不需要学习参数。
- 不受输入长度变化影响 (可以外推到很长序列)。

我们希望为每个位置 pos 生成一个向量 $\text{PosEnc}(pos) \in \mathbb{R}^{d_{\text{model}}}$, 它有以下几个特性:

目标	解释
不可学习	不增加参数, 易于训练
可推广	支持输入长度变化, 不受最大序列限制
保持平滑的相对位置表示	相邻位置的编码应该类似, 远位置变化逐渐增强
对任意位置差异有解析表达	后续 attention 可以感知到位置间的差异 (频率分布)

- $pos \in [0, \text{seq_len} - 1]$: 输入token的位置索引
- $i \in [0, \frac{d_{\text{model}}}{2} - 1]$: 维度索引 (从0开始), 第 $2i$ 维是sin, 第 $2i + 1$ 维是cos
- d_{model} : 嵌入维度 (如512)

假设输入表示 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 包含一个序列中 n 个词元的 d 维嵌入表示。位置编码使用相同形状的位置嵌入矩阵 $\mathbf{P} \in \mathbb{R}^{n \times d}$ 输出 $\mathbf{X} + \mathbf{P}$, 矩阵第 i 行、第 $2j$ 列和 $2j + 1$ 列上的元素为:

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right), \quad (10.6.2)$$
$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right).$$

在位置嵌入矩阵 \mathbf{P} 中, 行 i 对应 token 在序列中的位置, 列表示不同的位置编码维度/j: embedding 的维度索引

假设输入嵌入是:

- $\mathbf{X} \in \mathbb{R}^{n \times d}$: n是序列长度, d是embedding维度
- $\mathbf{P} \in \mathbb{R}^{n \times d}$: 位置编码矩阵
- 第 i 行表示第 i 个 token 的位置编码
- 第 $2j$ 列是 sin 函数
- 第 $2j + 1$ 列是 cos 函数

(b) 可学习的位置编码 (Learnable)

直接初始化一个可学习矩阵:

$$P \in \mathbb{R}^{\text{max_position} \times d_{\text{model}}}$$

对位置 pos , PositionalEmbedding 就是:

$$\text{PositionalEmbedding}(pos) = P[pos]$$

- P 是模型的一部分, 训练时会学习调整这些位置向量。
- 现代 Transformer (比如 GPT-2, GPT-3, BERT) 几乎都采用 Learnable 方式。

✓ 总结:

- 固定式: 不可训练, sin-cos 函数生成。
- 可学习: 位置向量是模型参数, 随训练更新。

11.6.3.1. 绝对位置信息 (跳过)

11.6.3.2. 相对位置信息 (跳过)

最终, 对于输入 token id 列表 (如一条句子), 流程是:

1. 通过 Token Embedding 查表得到 **词向量矩阵**: ($batch_size, seq_len, d_{\text{model}}$)
2. 通过 Positional Embedding 查表或计算得到 **位置向量矩阵**: ($1, seq_len, d_{\text{model}}$) (通常广播 batch)
3. 两者逐元素加和:

$$\text{EmbeddingOutput} = \text{TokenEmbedding} + \text{PositionalEmbedding}$$

然后送入 Transformer 的后续子层 (如 Multi-Head Attention)。

数据流总结:

- 输入: $\text{token_ids} \rightarrow (\text{batch_size}, \text{seq_len})$
- 经过 TokenEmbedding $\rightarrow (\text{batch_size}, \text{seq_len}, d_{\text{model}})$
- 经过加 PositionalEmbedding $\rightarrow (\text{batch_size}, \text{seq_len}, d_{\text{model}})$
- 输出给后续 Transformer 编码器或解码器。

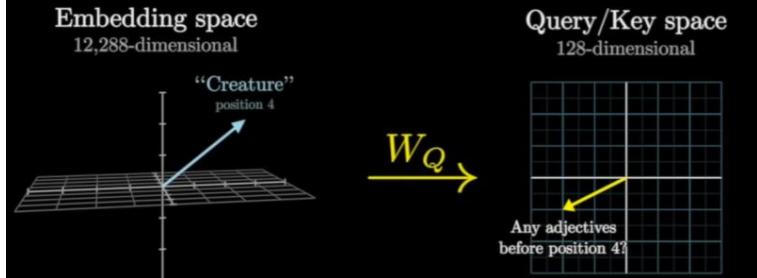
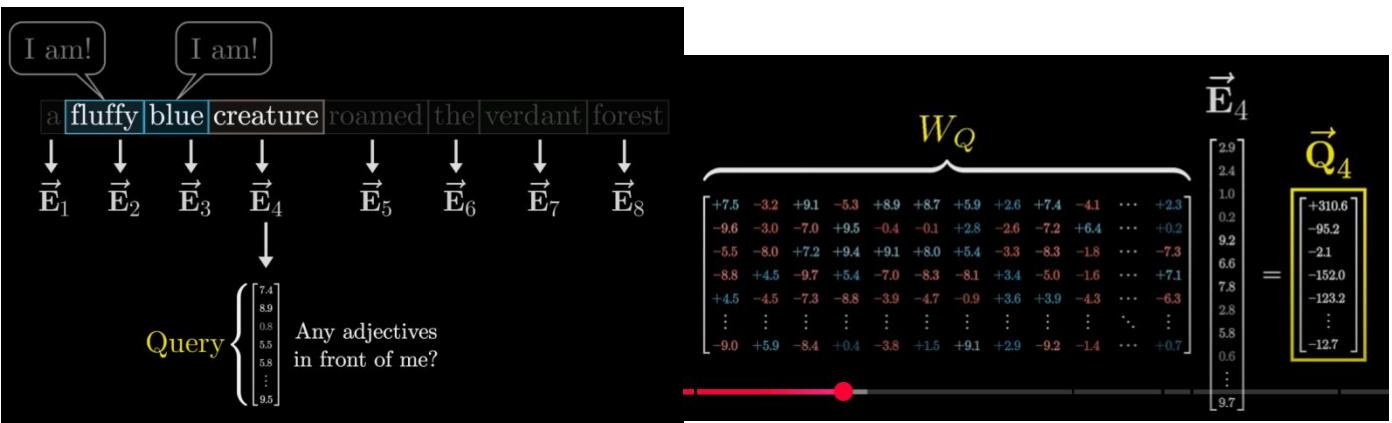
组件	数学表示	特点
TokenEmbedding	$E[i]$	把 id 映射为语义向量
PositionalEmbedding	$P[pos]$ 或 sin/cos 公式	给每个 token 加上位置信息
融合	$\text{TokenEmbedding} + \text{PositionalEmbedding}$	组合后作为 Transformer 输入

2. Scaled Dot-Product Attention & Multi-Head Attention

DL 通常用矩阵-向量的乘积形式计算, 矩阵充满了可调节的权重, 模型根据数据学习权重
通过微调海量参数 Tunable parameters 以最小化某种成本函数

2.1 Queries, Keys, and Values

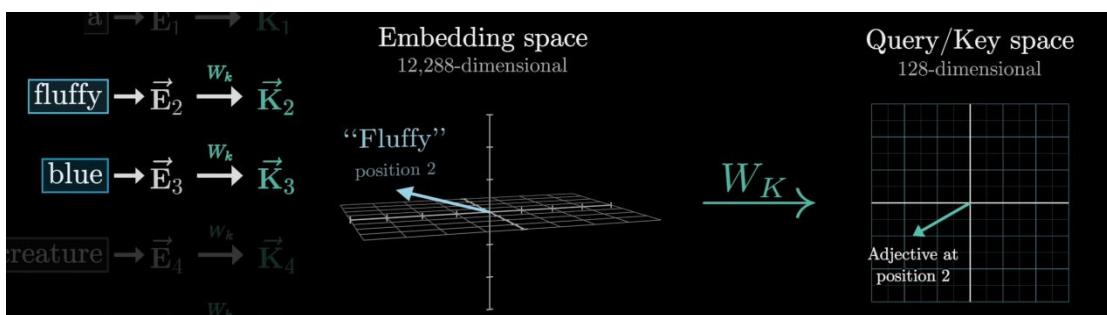
键值对 $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)\}$



查询向量维度比嵌入向量小得多，比如 128 维。

W_Q 矩阵由模型的参数组成，它能从数据中学习到真实的行为模式。

这个查询矩阵将嵌入信息映射到一个较小空间中的特点方向上。



2.2 Attention Pooling by Similarity

Note: Attention Pooling by Similarity 是总流程, Attention Scoring Functions 是其中的“打分子步骤”, Key-Query 点积是最经典、最常用的打分公式（尤其在 Transformer 里）

(RNN) 对序列进行编码。想象一下，有了注意力机制之后，我们将词元序列输入注意力池化中，以便同一组词元同时充当查询、键和值。具体来说，每个查询都会关注所有的键-值对并生成一个注意力输出。由于查询、键和值来自同一组输入，因此被称为 自注意力 (self-attention) ([Lin et al., 2017](#), [Vaswani et al., 2017](#))，也被称为 内部注意力 (intra-attention) ([Cheng et al., 2016](#),

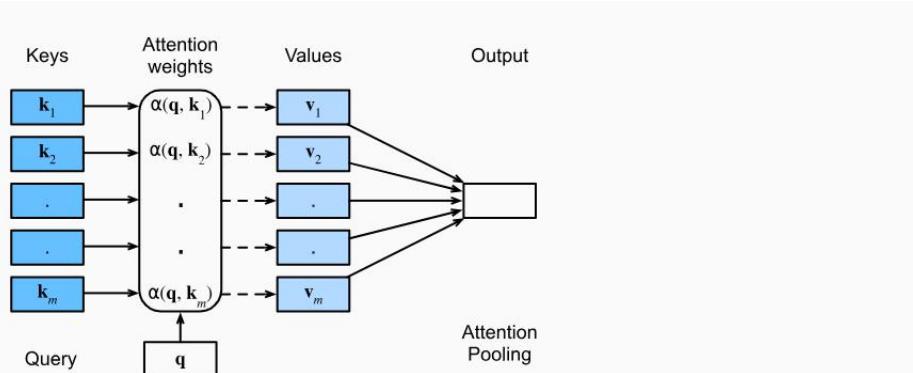


Fig. 11.1.1 The attention mechanism computes a linear combination over values v_i via attention pooling, where weights are derived according to the compatibility between a query q and keys k_i .

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^m \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i$$

The attention over D is

where $\alpha(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$ ($i = 1, \dots, m$) are scalar attention weights.

- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ are nonnegative. In this case the output of the attention mechanism is contained in the convex cone spanned by the values \mathbf{v}_i .
- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ form a convex combination, i.e., $\sum_i \alpha(\mathbf{q}, \mathbf{k}_i) = 1$ and $\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0$ for all i . This is the most common setting in deep learning.
- Exactly one of the weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ is 1, while all others are 0. This is akin to a traditional database query.
- All weights are equal, i.e., $\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{1}{m}$ for all i . This amounts to averaging across the entire database, also called average pooling in deep learning.

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}$$

Normalize and non-negative

Suppose that the attention function is given by $a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i$ and that $\mathbf{k}_i = \mathbf{v}_i$ for $i = 1, \dots, m$. Denote by $p(\mathbf{k}_i; \mathbf{q})$ the probability distribution over keys when using the softmax normalization in (11.1.3). Prove that $\nabla_{\mathbf{q}} \text{Attention}(\mathbf{q}, \mathcal{D}) = \text{Cov}_{p(\mathbf{k}_i; \mathbf{q})}[\mathbf{k}_i]$.

Attention Scoring Functions 注意力机制评分函数（相关性）

基于距离的核函数，包括高斯核来对查询和键之间的关系/交互。图中的高斯核指数部分可以视为注意力评分函数（attention scoring function），简称评分函数（scoring function），其中 a 表示注意力评分函数。然后把这个函数的输出结果输入到 softmax 函数中进行运算。通过上述步骤，将得到与键对应的值的概率分布（即注意力权重）。最后，注意力汇聚的输出就是基于这些注意力权重的值的加权和。

由于注意力权重是概率分布，因此加权和其本质上是加权平均值。

距离函数的计算成本略高于点积函数。因此使用 softmax 运算确保非负注意力的权重。

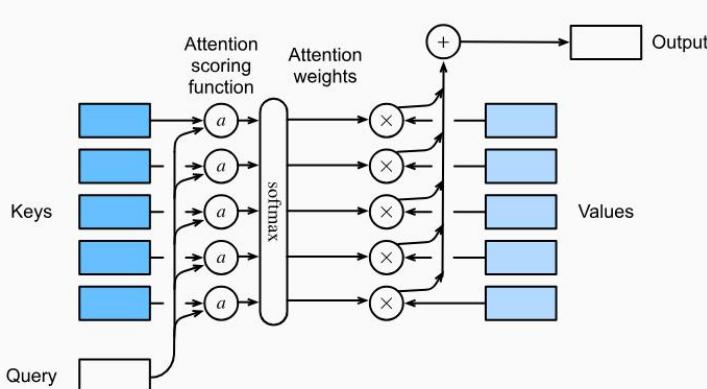


Fig. 11.3.1 Computing the output of attention pooling as a weighted average of values, where weights are computed with the attention scoring function a and the softmax operation.

用数学语言描述，假设有一个查询 $\mathbf{q} \in \mathbb{R}^q$ 和 m 个“键-值”对 $(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)$ ，其中 $\mathbf{k}_i \in \mathbb{R}^k$, $\mathbf{v}_i \in \mathbb{R}^v$ 。注意力汇聚函数 f 被表示成值的加权和：

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^m \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i \in \mathbb{R}^v, \quad (10.3.1)$$

其中查询 \mathbf{q} 和键 \mathbf{k}_i 的注意力权重（标量）是通过注意力评分函数 a 将两个向量映射成标量，再经过 softmax 运算得到的：

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^m \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}. \quad (10.3.2)$$

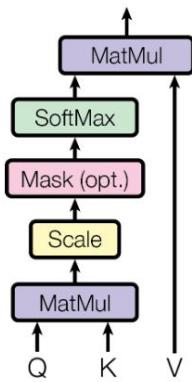
正如上图所示，选择不同的注意力评分函数 a 会导致不同的注意力汇聚操作。本节将介绍两个流行的

二选一：Scaled Dot Product Attention(主要)/ Additive Attention

2.2.1 Scaled Dot Product Attention 缩放点积注意力

计算效率更高。点积操作要求查询和键具有相同的长度 d 。

Scaled Dot-Product Attention



尽管这个问题可以通过替换轻松解决 $\mathbf{q}^\top \mathbf{k}$ 和 $\mathbf{q}^\top \mathbf{Mk}$ 在哪里 \mathbf{M} 是一个适合在两个空间之间平移的矩阵。现在假设维度匹配。

使用点积可以得到计算效率更高的评分函数，但是点积操作要求查询和键具有相同的长度 d 。假设查询和键的所有元素都是独立的随机变量，并且都满足零均值和单位方差，那么两个向量的点积的均值为 0，方差为 d 。为确保无论向量长度如何，点积的方差在不考虑向量长度的情况下仍然是 1，我们再将点积除以 \sqrt{d} ，则缩放点积注意力（scaled dot-product attention）评分函数为：

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k} / \sqrt{d}. \quad (10.3.4)$$

在实践中，我们通常从小批量的角度来考虑提高效率，例如基于 n 个查询和 m 个键-值对计算注意力，其中查询和键的长度为 d ，值的长度为 v 。查询 $\mathbf{Q} \in \mathbb{R}^{n \times d}$ 、键 $\mathbf{K} \in \mathbb{R}^{m \times d}$ 和值 $\mathbf{V} \in \mathbb{R}^{m \times v}$ 的缩放点积注意力是：

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times v}. \quad (10.3.5)$$

下面的缩放点积注意力的实现使用了暂退法进行模型正则化。

注意，当将其应用于小批量时，我们需要(11.3.5)中介绍的批量矩阵乘法。在下面的缩放点积注意力机制实现中，我们使用 dropout 进行模型正则化。

```

# queries 的形状: (batch_size, 查询的个数, d)
# keys 的形状: (batch_size, “键-值”对的个数, d)
# values 的形状: (batch_size, “键-值”对的个数, 值的维度)
  
```

valid_lens 的形状: (batch_size,) 或者 (batch_size, 查询的个数)

: 高斯核的注意力函数 (没有指数函数) :

$$a(\mathbf{q}, \mathbf{k}_i) = -\frac{1}{2} \|\mathbf{q} - \mathbf{k}_i\|^2 = \mathbf{q}^\top \mathbf{k}_i - \frac{1}{2} \|\mathbf{k}_i\|^2 - \frac{1}{2} \|\mathbf{q}\|^2. \quad (11.3.1)$$

First, note that the final term depends on \mathbf{q} only. As such it is identical for all $(\mathbf{q}, \mathbf{k}_i)$ pairs.

Normalizing the attention weights to 1, as is done in (11.1.3), ensures that this term disappears entirely. Second, note that both batch and layer normalization (to be discussed later) lead to activations that have well-bounded, and often constant, norms $\|\mathbf{k}_i\|$. This is the case, for instance, whenever the keys \mathbf{k}_i were generated by a layer norm. As such, we can drop it from the definition of a without any major change in the outcome.

最后, 我们需要控制指数函数中参数的数量级。假设查询的所有元素 $\mathbf{q} \in \mathbb{R}^d$ 和钥匙 $\mathbf{k}_i \in \mathbb{R}^d$ 是独立且相同的随机变量, 均值为零, 方差为单位。两个向量之间的点积均值为零, 方差为 d . 确保点积的方差仍然保持1无论向量长度如何, 我们都使用缩放点积注意力评分函数。也就是说, 我们将点积重新缩放为 $1/\sqrt{d}$ 。因此, 我们得到了第一个常用的注意力函数, 例如在 Transformers 中使用 (Vaswani 等人, 2017) :

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}. \quad (11.3.2)$$

注意权重 α 仍然需要归一化。我们可以通过(11.1.3)式, 利用softmax运算进一步简化这一过程:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_{j=1} \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})}. \quad (11.3.3)$$

当 key 与 query 密切对齐时, 可以将 key 视为与 query 相匹配。为了衡量每个 key 和每个 query 的匹配程度, 计算每一对可能的 key-query 组合之间的点积。较大的点积 (正数) 即 key 与 query 对齐。点积值较小或为负数表示没有什么关联。评估一个 token 对另一个 token 的相关性。

	a	fluffy	blue	creature	roamed	the	verdant	forest
a	\vec{E}_1	\vec{E}_2	\vec{E}_3	\vec{E}_4	\vec{E}_5	\vec{E}_6	\vec{E}_7	\vec{E}_8
fluffy	\vec{Q}_1	\vec{Q}_2	\vec{Q}_3	\vec{Q}_4	\vec{Q}_5	\vec{Q}_6	\vec{Q}_7	\vec{Q}_8
blue	\vec{Q}_1	\vec{Q}_2	\vec{Q}_3	\vec{Q}_4	\vec{Q}_5	\vec{Q}_6	\vec{Q}_7	\vec{Q}_8
creature	\vec{Q}_1	\vec{Q}_2	\vec{Q}_3	\vec{Q}_4	\vec{Q}_5	\vec{Q}_6	\vec{Q}_7	\vec{Q}_8
roamed	\vec{Q}_1	\vec{Q}_2	\vec{Q}_3	\vec{Q}_4	\vec{Q}_5	\vec{Q}_6	\vec{Q}_7	\vec{Q}_8
the	\vec{Q}_1	\vec{Q}_2	\vec{Q}_3	\vec{Q}_4	\vec{Q}_5	\vec{Q}_6	\vec{Q}_7	\vec{Q}_8
verdant	\vec{Q}_1	\vec{Q}_2	\vec{Q}_3	\vec{Q}_4	\vec{Q}_5	\vec{Q}_6	\vec{Q}_7	\vec{Q}_8
forest	\vec{Q}_1	\vec{Q}_2	\vec{Q}_3	\vec{Q}_4	\vec{Q}_5	\vec{Q}_6	\vec{Q}_7	\vec{Q}_8

ML 术语: attend to “蓝色”的嵌入向量会关注(attend to)“生物”的嵌入向量

2.2.2 Additive Attention 加性注意力 (并未使用)

Query 和 key 维度不同, 节省计算资源。

一般来说, 当查询和键是不同长度的矢量时, 可以使用加性注意力作为评分函数。给定查询 $\mathbf{q} \in \mathbb{R}^q$ 和键 $\mathbf{k} \in \mathbb{R}^k$, 加性注意力 (additive attention) 的评分函数为

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}, \quad (10.3.3)$$

其中可学习的参数是 $\mathbf{W}_q \in \mathbb{R}^{h \times q}$ 、 $\mathbf{W}_k \in \mathbb{R}^{h \times k}$ 和 $\mathbf{w}_v \in \mathbb{R}^h$ 。如 (10.3.3) 所示, 将查询和键结合起来后输入到一个多层次感知机 (MLP) 中, 感知机包含一个隐藏层, 其隐藏单元数是一个超参数 h 。通过使用 \tanh 作为激活函数, 并且禁用偏置项。

总结：

将注意力汇聚的输出计算可以作为值的加权平均，选择不同的注意力评分函数会带来不同的注意力汇聚操作。

当查询和键是不同长度的矢量时，可以使用可加性注意力评分函数。当它们的长度相同时，使用缩放的“点一积”注意力评分函数的计算效率更高。

查询和键是不同长度的向量时，我们可以改用加性注意力机制评分函数。优化这些层是近年来取得进展的关键领域之一。例如，[NVIDIA 的 Transformer 库](#) 和 Megatron ([Shoeybi 等人，2019 年](#)) 都依赖于注意力机制的有效变体。在后面回顾 Transformer 时，我们将更详细地探讨这一点。

P.S. efficient 处理可变长度字符串的工具 Masking (NLP 中很常见) 以及高效评估小批量数据的工具 minibatches ([Batch Matrix Multiplication 批量矩阵乘法：跳过](#))

Masked Softmax Operation 遮蔽的 Softmax 操作

softmax 操作用于输出一个概率分布作为注意力权重。在某些情况下，并非所有的值都应该被纳入到注意力汇聚中。例如，为了高效处理小批量数据集，某些文本序列被填充了没有意义的特殊词元。为了仅将有意义的词元作为值来获取注意力汇聚，可以指定一个有效序列长度(即词元的个数)，以便在计算 softmax 时过滤掉超出指定范围的位置。

We need to be able to deal with sequences of different lengths. In some cases, such sequences may end up in the same minibatch, necessitating padding with dummy tokens for shorter sequences (see Section 10.5 for an example). These special tokens do not carry meaning.

注意力机制最流行的应用之一是序列模型。因此，我们需要能够处理不同长度的序列。在某些情况下，这些序列最终可能会被放在同一个小批量中，因此需要用虚拟标记来填充较短的序列（参见[10.5 节](#)的示例）。这些特殊标记本身不包含任何含义。例如，假设我们有以下三个句子：

```
Dive into Deep Learning
Learn to code <blank>
Hello world <blank> <blank>
```

由于我们不希望注意力模型中出现空白，因此我们只需要限制 $\sum_{i=1}^n \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i$ 到 $\sum_{i=1}^l \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i$ 无论多久， $l \leq n$ ，实际句子是。由于这是一个常见的问题，它有一个名称：*masked softmax operation*。

让我们来实现它。实际上，这个实现稍微有点作弊，通过设置 \mathbf{v}_i ，为了 $i > l$ ，为零。此外，它将注意力权重设置为一个较大的负数，例如 -10^6 ，以便在实践中使它们对梯度和值的贡献消失。这样做是因为线性代数核和运算符针对 GPU 进行了高度优化，并且稍微浪费一些计算资源比使用带有条件语句 (if then else) 的代码更快。

注意力机制原则：在 decoder，不允许后出现的影响先出现的词汇。所以按列计算 softmax 以归一化/标准化之前，将这些影响值设为负无穷大。

Unnormalized Attention Pattern						Normalized Attention Pattern					
+3.53	+0.80	+1.96	+4.48	+3.74	-1.95	1.00	0.75	0.69	0.92	0.46	0.00
$-\infty$	-0.30	-0.21	+0.82	+0.29	+2.91	0.00	0.25	0.08	0.02	0.01	0.46
$-\infty$	$-\infty$	+0.89	+0.67	+2.99	-0.41	0.00	0.00	0.24	0.02	0.22	0.02
$-\infty$	$-\infty$	$-\infty$	+1.31	+1.73	-1.48	0.00	0.00	0.00	0.04	0.06	0.01
$-\infty$	$-\infty$	$-\infty$	$-\infty$	+3.07	+2.94	0.00	0.00	0.00	0.00	0.24	0.48
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	+0.31	0.00	0.00	0.00	0.00	0.00	0.03

softmax →

这样，经过 softmax 处理后，这些位置的数值会变成零，

让词语可以将信息传递给和它们相关的其他词。相关的保留，其他 value vector 则被减少为或接近 0 注意 W_V 是学习到的， E_2 是第二个 token 的嵌入向量， V_2 是第二个 token 的 value vector 值向量

在 Transformer 的前几层，输入 X 就是 Embedding 矩阵 E 。

项目	定义
$E(i)$	词表中的第 i 个 token 的嵌入向量，通常来自 Embedding 层。 $E \in \mathbb{R}^{V \times d_{\text{model}}}$ (V 是词表大小)
X	当前输入序列（一个 batch）对应的所有 token 嵌入的集合，shape 是 $(batch_size, seq_len, d_{\text{model}})$
Q, K, V	分别是从 X 经过不同线性映射（乘上不同矩阵）得到的三个张量，用来做 Attention 计算

所以，整个流程是：

1. **词表查表**，将输入的 token id 序列映射到嵌入向量，得到 X ：

$$X = [E(i_1), E(i_2), \dots, E(i_{seq_len})]$$

2. **从 X 生成 Q, K, V** ：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

✓ 小结一句话：

在 Transformer 中，输入 X 本质上就是把 Embedding 向量 $E(i)$ 取出来按序列堆叠形成的， X 就是 Embedding 矩阵在 batch 上实例化后的样子。

Attention 机制通过三种不同的、充满可调节参数的矩阵来实现的。

Transformer 中标准 Attention 公式是：

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

其中：

- Q, K, V 都是通过输入 X 经过不同的线性变换得到的：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

其中 W_Q, W_K, W_V 都是需要学习的权重矩阵。

而你的图片里特别强调的是 **Value 的线性映射细节**，尤其在大模型 ($d_{\text{model}} = 12,288$) 的情况下，如何有效、经济地设计 W_V 。

- **Query, Key, Value** 都是输入 X 经过线性映射得到的。
- 这里的“线性映射”指的是“矩阵乘法” (W 是可学习矩阵)。



Value 的线性映射可以拆分成降维矩阵和升维矩阵，即为 Low-Rank Transformation（低秩变换）

Value

d_input 12,288	d_output 12,288
12,288	
$12,288 \left\{ \begin{bmatrix} -3.2 & +9.1 & -5.3 & +8.9 & +8.7 & +5.9 & +2.6 & +7.4 & \cdots & -4.1 \\ +6.9 & +2.3 & -9.6 & -3.0 & -7.0 & +0.5 & -0.4 & -0.1 & \cdots & +2.8 \\ -2.6 & -7.2 & +6.4 & -6.1 & +0.2 & -5.5 & -8.0 & +7.2 & \cdots & +0.4 \\ +9.1 & +8.0 & +5.4 & -3.3 & -8.3 & -1.8 & -5.3 & -7.3 & \cdots & -8.8 \\ +4.5 & -9.7 & +5.4 & -7.0 & -8.3 & -8.1 & +3.4 & -5.0 & \cdots & -1.6 \\ +1.1 & +7.1 & +4.5 & -4.5 & -7.3 & -8.8 & -3.9 & -4.7 & \cdots & -0.9 \\ +3.6 & +3.9 & -4.3 & -2.4 & -6.3 & +5.7 & -8.8 & +3.9 & \cdots & +5.5 \\ +5.5 & -4.8 & -2.5 & +1.7 & +4.5 & -2.6 & -6.0 & -0.8 & \cdots & -9.0 \\ \vdots & \ddots & \vdots \\ +5.9 & +8.4 & +0.4 & -3.8 & +1.5 & +9.1 & +2.9 & +9.2 & \cdots & -14 \end{bmatrix} \right. = \left. \begin{bmatrix} +0.9 \\ +0.7 \\ -3.6 \\ -4.4 \\ -7.3 \\ -2.1 \\ +9.0 \\ -5.1 \\ \vdots \\ +0.9 \end{bmatrix} \right]$	

(a) 问题背景

- 原始的 W_V 是一个巨大的矩阵:
- 输入、输出都是 12,288 维, 所以 W_V 形状是 $12,288 \times 12,288$.
- 参数量巨大, 计算量也巨大!

如果直接这么大规模训练, 几乎不可行!

输入输出都在更大的嵌入空间中

如果想要 more efficient: # Value params = (# Query params) + (# Key params)

值矩阵的参数数量与键矩阵和查询矩阵的残数数量相同。尤其在同时运行多个注意力机制中十分重要。

值映射实际是两个小矩阵乘积的形式。

(b) 解决方法: 低秩分解 (降维+升维)

我们观察到:

很多时候, 特征在高维空间里实际变化的自由度很小。

👉 所以我们可以先降到一个较低的子空间 (比如128维), 然后再升回原始高维空间。

具体做法:

- 用两个小矩阵 W_{down} 和 W_{up} 替代原来一个大矩阵 W_V .
- 具体结构:
 - $W_{\text{down}} : 12,288 \rightarrow 128$ (降维矩阵)
 - $W_{\text{up}} : 128 \rightarrow 12,288$ (升维矩阵)
- 那么原来 VW_V 的过程, 现在变成:

$$VW_V \approx VW_{\text{down}}W_{\text{up}}$$

Linear map

12,288	12,288
$12,288 \left\{ \begin{bmatrix} +5.0 & -3.3 & \cdots & +7.2 \\ -8.9 & -4.9 & \cdots & -7.8 \\ -3.0 & +4.8 & \cdots & +2.4 \\ +4.2 & -5.8 & \cdots & +3.5 \\ +7.5 & +0.9 & \cdots & -9.3 \\ +4.2 & -9.7 & \cdots & +0.6 \\ +8.4 & -8.1 & \cdots & -9.4 \\ -3.1 & +2.4 & \cdots & -5.7 \\ \vdots & \vdots & \ddots & \vdots \\ +8.9 & -9.8 & \cdots & +2.0 \end{bmatrix} \right. = \left. \begin{bmatrix} +0.2 \\ +0.7 \\ +3.6 \\ -4.4 \\ -7.3 \\ -2.1 \\ +9.0 \\ -6.2 \\ \vdots \\ +0.9 \end{bmatrix} \right]$	

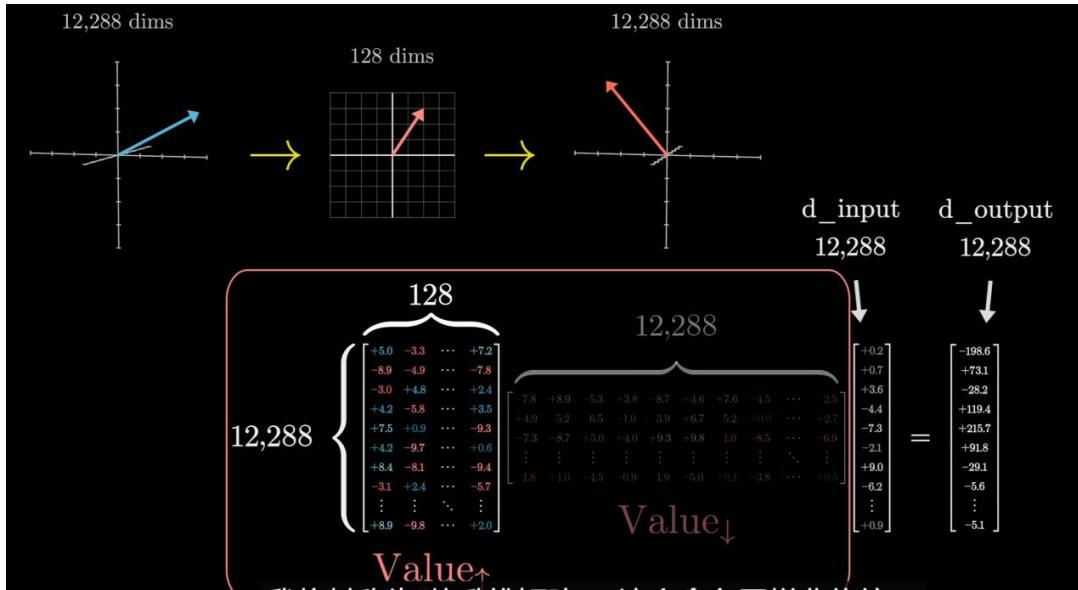
我还是建议从整体上去理解这个线性映射过程,

只是分为两个步骤:

SVD 矩阵分解 Low-rank transformation 值降维矩阵 + 值升维矩阵 (命名非传统)

第一个矩阵 (右边的): 将较大的嵌入向量映射到了一个更小的空间

第二个矩阵 (左边的): 将较小的空间映射映射回原来的嵌入空间



(c) 直观理解

- 第一步：Value矩阵从12288维降到128维（低秩子空间） \rightarrow 捕捉最主要的方向。
- 第二步：再从128维升回12288维 \rightarrow 补齐信息到原始维度。
- 因为重要的信息往往集中在很小的子空间，所以整体信息不会丢失太多，但计算量和参数量大大减少。

图中术语	解释
Value升维矩阵 $Value^{\uparrow}$	$128 \rightarrow 12288$ 的映射
Value降维矩阵 $Value^{\downarrow}$	$12288 \rightarrow 128$ 的映射
Low rank transformation	两次小矩阵乘法代替一次大矩阵乘法（近似低秩分解）

W_Q	W_K	$\uparrow W_V$	$\downarrow W_V$
$\begin{bmatrix} -4.8 & +5.5 & +9.3 & +7.5 & +7.7 & +6.0 & \dots & -1.3 \\ -8.0 & -0.9 & -4.6 & -7.6 & -7.3 & +2.5 & \dots & -6.2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ +2.3 & -3.6 & -7.2 & -7.0 & +0.7 & -5.3 & \dots & -1.7 \end{bmatrix}$	$\begin{bmatrix} +1.1 & +7.1 & +4.5 & +4.5 & -7.3 & -8.8 & \dots & -3.9 \\ -4.7 & -0.9 & +3.6 & +3.9 & +4.3 & -2.4 & \dots & -6.3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ +5.7 & -8.8 & +3.9 & +5.5 & +5.5 & -4.8 & \dots & -2.5 \end{bmatrix}$	$\begin{bmatrix} +1.7 & -4.5 & \dots & -2.6 \\ -6.0 & -0.8 & \dots & -9.0 \\ +5.9 & -8.4 & \dots & +0.4 \\ -3.8 & +1.5 & \dots & +9.1 \\ +2.9 & -9.2 & \dots & -1.4 \\ +0.2 & +0.7 & \dots & +3.6 \\ \vdots & \vdots & \ddots & \vdots \\ -4.4 & -7.3 & \dots & -2.1 \end{bmatrix}$	$\begin{bmatrix} +9.0 & -6.2 & +8.0 & +0.9 & -0.9 & +7.6 & \dots & -0.8 \\ +4.4 & -2.0 & +8.0 & +3.8 & +4.0 & -3.4 & \dots & +5.1 \\ +2.7 & -5.1 & -6.7 & +5.9 & +9.1 & -0.8 & \dots & +1.8 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -4.4 & -7.3 & \dots & -2.1 \end{bmatrix}$



One-Head Attention:

	\downarrow \vec{E}_1	\downarrow \vec{E}_2	\downarrow \vec{E}_3	\downarrow \vec{E}_4	\downarrow \vec{E}_5	\downarrow \vec{E}_6	\downarrow \vec{E}_7	\downarrow \vec{E}_8
$\vec{E}_1 \xrightarrow{W_V} \vec{V}_1$	1.00 \vec{V}_1	0.00 \vec{V}_1	0.00 \vec{V}	0.00 \vec{V}_1	0.00 \vec{V}	0.00 \vec{V}_1	0.00 \vec{V}	0.00 \vec{V}_1
$\vec{E}_2 \xrightarrow{W_V} \vec{V}_2$	0.00 \vec{V}	1.00 \vec{V}_2	0.00 \vec{V}_2	0.42 \vec{V}_2	0.00 \vec{V}	0.00 \vec{V}_2	0.00 \vec{V}_2	0.00 \vec{V}_2
$\vec{E}_3 \xrightarrow{W_V} \vec{V}_3$	0.00 \vec{V}	0.00 \vec{V}_3	1.00 \vec{V}_3	0.58 \vec{V}_3	0.00 \vec{V}	0.00 \vec{V}_3	0.00 \vec{V}_3	0.00 \vec{V}_3
$\vec{E}_4 \xrightarrow{W_V} \vec{V}_4$	0.00 \vec{V}	0.00 \vec{V}_4	0.00 \vec{V}	0.00 \vec{V}_4	1.00 \vec{V}	0.00 \vec{V}_4	0.00 \vec{V}	0.00 \vec{V}_4
$\vec{E}_5 \xrightarrow{W_V} \vec{V}_5$	0.00 \vec{V}	0.00 \vec{V}_5	0.00 \vec{V}	0.00 \vec{V}_5	0.01 \vec{V}_5	0.00 \vec{V}_5	0.00 \vec{V}_5	0.00 \vec{V}_5
$\vec{E}_6 \xrightarrow{W_V} \vec{V}_6$	0.00 \vec{V}	0.00 \vec{V}_6	0.00 \vec{V}_6	0.00 \vec{V}_6	0.99 \vec{V}_6	1.00 \vec{V}_6	0.00 \vec{V}_6	0.00 \vec{V}_6
$\vec{E}_7 \xrightarrow{W_V} \vec{V}_7$	0.00 \vec{V}	0.00 \vec{V}_7	0.00 \vec{V}	0.00 \vec{V}_7	0.00 \vec{V}	0.00 \vec{V}_7	1.00 \vec{V}_7	1.00 \vec{V}_7
$\vec{E}_8 \xrightarrow{W_V} \vec{V}_8$	0.00 \vec{V}	0.00 \vec{V}_8	0.00 \vec{V}_8	0.00 \vec{V}_8	0.00 \vec{V}	0.00 \vec{V}_8	0.00 \vec{V}	0.00 \vec{V}_8
	\parallel							
	$\Delta \vec{E}_1$	$\Delta \vec{E}_2$	$\Delta \vec{E}_3$	$\Delta \vec{E}_4$	$\Delta \vec{E}_5$	$\Delta \vec{E}_6$	$\Delta \vec{E}_7$	$\Delta \vec{E}_8$
	\vec{E}'_1	\vec{E}'_2	\vec{E}'_3	\vec{E}'_4	\vec{E}'_5	\vec{E}'_6	\vec{E}'_7	\vec{E}'_8

P.S. low-rank approximation

Transformer 中的 Value 映射通常是一个线性变换，为了减少大模型计算量，可以把这步线性映射近似为降维（压缩到小空间）+ 升维（还原到大空间），这种设计叫做低秩近似（**low-rank approximation**）。这个“降维+升维”不仅节省计算资源，而且还有：

- 正则化效果（防止过拟合）
- 约束网络只能捕捉最主要的特征方向
- 更符合实际数据在高维空间中本质自由度很低的统计特性

这也是像 **Adapter Layers, LoRA (Low-Rank Adaptation), ALiBi, Attention Linearization** 这些优化技术背后的共同数学思想！

首先，回顾一下什么是低秩近似：

- 对一个很大的矩阵 $W \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ ，用两个小矩阵 $W_{\text{down}} \in \mathbb{R}^{d_{\text{model}} \times r}$ 、 $W_{\text{up}} \in \mathbb{R}^{r \times d_{\text{model}}}$ 代替。
- 其中 $r \ll d_{\text{model}}$ （比如原来是12288，现在中间是128）。

即：

$$W \approx W_{\text{down}} W_{\text{up}}$$

这样操作带来了一系列优点，我们来一一解释原因：

◆ 1. 降低参数量和计算量（最直接的优点）

原始参数量：

$$\text{param_count}(W) = d_{\text{model}}^2$$

比如 $d_{\text{model}} = 12,288$ ，那就是 $12,288^2 = 151$ million parameters。

低秩分解后的参数量：

$$\text{param_count}(W_{\text{down}} W_{\text{up}}) = 2 \times d_{\text{model}} \times r$$

比如 $r = 128$ ，那就是 $2 \times 12,288 \times 128 = 3.1$ million parameters，缩小了50倍以上！

✓ 总结：

- 少了巨量的参数，模型更轻，更快。

◆ 2. 正则化效果 (防止过拟合)

- 原本的大矩阵 W 自由度太高，可以拟合任何复杂模式，容易过拟合。
- 低秩近似后，模型被限制只能在一个小维度子空间上学习特征，即强制网络只学习到主要方向。
- 这就像给网络加了一个**结构性的正则化**，让它不能随便过度拟合。

直观比喻：

| 原本可以到处飞，现在只能在一个小空间内转圈，训练更稳定，测试集表现更好。

◆ 3. 捕捉主要信息 (信息保留)

- 在高维空间里，实际有用的变化自由度往往很小（比如自然语言、图像，低维流形假设）。
- 低秩近似正好捕捉了这些主要自由度变化方向。
- 只要 r 选得合适（比如 128, 256），大部分信息都可以保留下，丢失的是一些无关噪声。

数学直觉：

| 低秩矩阵可以捕捉原始矩阵中最大的奇异值对应的子空间，而这些子空间包含了大部分重要变化。

◆ 4. 更好的可扩展性 (适合大规模模型)

- 低秩近似使得即使在 d_{model} 巨大的情况下（比如 GPT-3/4, PaLM 2），
- Attention, FFN 层也可以稳定训练，不然内存和计算量爆炸。

实际例子：

| GPT-3 中 d_{model} 是 12,288，不做任何降维直接训练 Attention 基本不可行，必须靠这些近似才能搞得动。

优点	原因
减少参数量	两个小矩阵替代一个大矩阵
降低计算量	矩阵乘法规模大大减小
正则化效果	限制模型自由度，防止过拟合
保留主要信息	低秩近似自动提取最重要变化方向
适合超大模型扩展	让超大 d_{model} 也能稳定训练

P.S. **Attention Pooling by Similarity** 和核密度估计（Kernel Density Estimation, KDE）的联系
how the choice of kernels is related to kernel density estimation, sometimes also called Parzen Windows
所有核都是启发式的，可以进行调整。

$$f(\mathbf{q}) = \sum_i \mathbf{v}_i \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \quad (11.2.2)$$

对于带有观测值的（标量）回归 (\mathbf{x}_i, y_i) 分别表示特征和标签， $\mathbf{v}_i = y_i$ 是标量， $\mathbf{k}_i = \mathbf{x}_i$ 是向量，并且查询 \mathbf{q} 表示新的位置 f 应该进行评估。在（多类）分类的情况下，我们使用独热编码 y_i 获得 \mathbf{v}_i 该估计器的一个便捷特性是它无需训练。更重要的是，如果我们随着数据量的增加而适当地缩小核函数，该方法是一致的（[Mack and Silverman, 1982](#)），也就是说，它将收敛到某个统计上最优的解。让我们先来检查一些核函数。

◆ 1. Attention Pooling by Similarity

我们在前面已经讲过，它是一种加权汇聚机制，权重来源于 Query 和每个 Key 的相似度（similarity score）：

$$\text{Attention}(Q, K, V) = \sum_i \alpha_i V_i$$

其中权重 α_i 是通过 Query 和第 i 个 Key 的相似度计算出来的：

$$\alpha_i = \frac{\exp(\text{score}(Q, K_i))}{\sum_j \exp(\text{score}(Q, K_j))}$$

这其实就是一个 softmax 归一化的相似度分布，表示在当前 Query 条件下，“我应该关注哪些 Key”。

◆ 2. Kernel Density Estimation (KDE)

KDE 是一种非参数方法，用于估计概率密度函数：

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

其中：

- x : 目标点
- x_i : 样本点
- $K(\cdot)$: 核函数（如高斯核）
- h : 带宽参数

这是在说：

给定一个点 x ，我用所有样本 x_i 对它“加权求和”来估计 x 处的密度。权重由核函数（相似度）决定。

模块	Attention	KDE
查询点	Query Q	目标估计点 x
样本点	Key K_i	数据样本 x_i
相似度函数	$\text{score}(Q, K_i)$, 如 $Q \cdot K_i$	$K\left(\frac{x - x_i}{h}\right)$
归一化处理	softmax 归一化	核函数自然归一化或除以 nh
输出聚合方式	$\sum_i \alpha_i V_i$	$\sum_i K(\cdot)$
本质	Query 处的条件注意力表示	x 处的概率密度估计
		它们其实非常像！

💡 数学视角下的联系

- Attention 的 softmax(score) 就可以看作一种核函数，尤其是当：

$$\text{score}(Q, K_i) = -\|Q - K_i\|^2$$

时，softmax 就变成了类似高斯核的作用：

$$\alpha_i \sim \exp\left(-\frac{\|Q - K_i\|^2}{2\sigma^2}\right)$$

和 KDE 的 Gaussian Kernel 是一致的！

- 从这个角度看，Attention 就像是在 Query 点处进行一维条件的核加权聚合，只是：

- KDE 关注密度估计（目标是 scalar）
- Attention 聚合的是 Value（目标是 vector）

🧠 进一步解释：Attention 是“条件 KDE + 表示聚合”

我们可以理解：

- Attention 是一个条件 KDE：

- 给定一个 Query（作为条件），我们计算它对每个 Key 的相似度（密度函数）
- 然后用这些密度权重去加权 Value，形成 Query 的“注意聚合表示”。

换句话说，Attention = 条件核密度加权的值聚合。

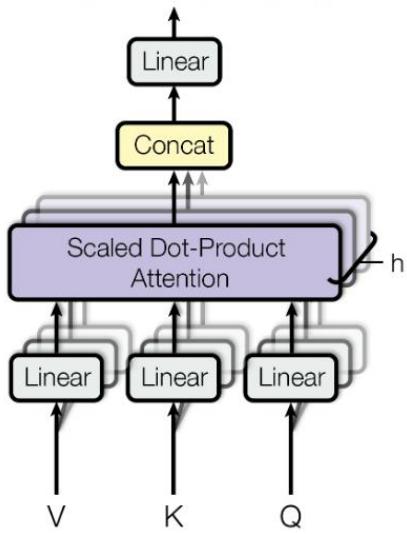
维度	KDE	Attention Pooling
样本空间	$\{x_i\}$	$\{K_i, V_i\}$
目标点	x	Q
相似度函数	$K\left(\frac{x-x_i}{h}\right)$	$\text{score}(Q, K_i)$ (如点积)
权重归一方式	核函数归一化	softmax
加权聚合输出	scalar密度	vector聚合表示
本质	密度估计	语义表示聚合

一句话总结：Attention Pooling by Similarity 本质上是一种“条件化的核密度加权聚合”，它在数学形式上与 Kernel Density Estimation 高度相似，不同之处在于 attention 用来计算注意分布并加权表示向量，而 KDE 是在进行概率密度估计。

P.S. Transformer 不用 The Bahdanau Attention Mechanism

2.3 Multi-Head Attention

Multi-Head Attention



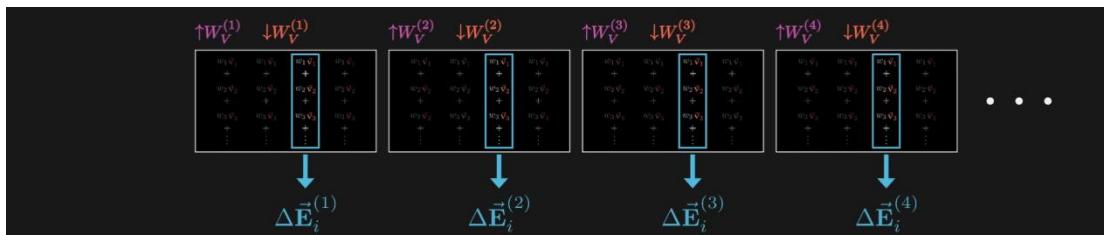
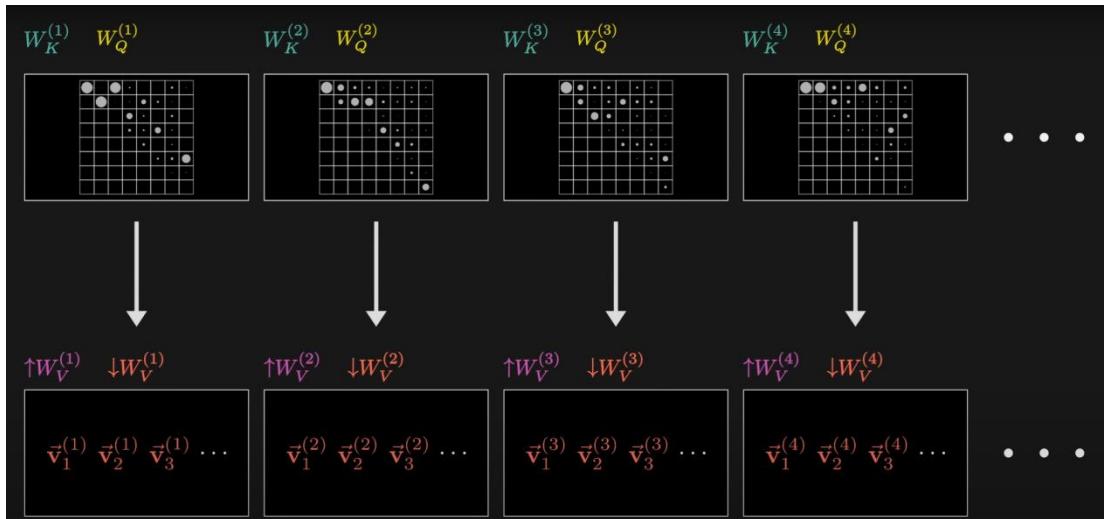
Multi-headed attention

$$\begin{aligned} W_Q^{(1)} & W_Q^{(2)} & W_Q^{(3)} & W_Q^{(4)} & W_Q^{(5)} & W_Q^{(6)} & W_Q^{(7)} & W_Q^{(8)} & W_Q^{(9)} & \dots \\ W_K^{(1)} & W_K^{(2)} & W_K^{(3)} & W_K^{(4)} & W_K^{(5)} & W_K^{(6)} & W_K^{(7)} & W_K^{(8)} & W_K^{(9)} & \dots \\ \downarrow W_V^{(1)} & \downarrow W_V^{(2)} & \downarrow W_V^{(3)} & \downarrow W_V^{(4)} & \downarrow W_V^{(5)} & \downarrow W_V^{(6)} & \downarrow W_V^{(7)} & \downarrow W_V^{(8)} & \downarrow W_V^{(9)} & \dots \\ \uparrow W_V^{(1)} & \uparrow W_V^{(2)} & \uparrow W_V^{(3)} & \uparrow W_V^{(4)} & \uparrow W_V^{(5)} & \uparrow W_V^{(6)} & \uparrow W_V^{(7)} & \uparrow W_V^{(8)} & \uparrow W_V^{(9)} & \dots \end{aligned}$$



并行运算，每个运算都有独特的 value, query, key 映射

有 h 个 head，就有 h 种不同的 Attention Mechanism, h 种不同的 W_V, W_K, W_Q



$$\text{Original embedding } \vec{\mathbf{E}}_i + \Delta\vec{\mathbf{E}}_i^{(1)} + \Delta\vec{\mathbf{E}}_i^{(2)} + \Delta\vec{\mathbf{E}}_i^{(3)} + \Delta\vec{\mathbf{E}}_i^{(4)} + \dots$$

通过使用对应的注意力机制作为权重进行加和。意味着，在上下文的每个位置，对于每个 token，所有的头都会产生一个建议的变化，一起添加到该位置的嵌入中。

在实践中，当给定相同的查询、键和值的集合时，我们希望模型可以基于相同的注意力机制学习到不同的行为，然后将不同的行为作为知识组合起来，捕获序列内各种范围的依赖关系（例如，短距离依赖和长距离依赖关系）。因此，允许注意力机制组合使用查询、键和值的不同子空间表示（representation subspaces）可能是有益的。

为此，与其只使用单独一个注意力汇聚，我们可以用独立学习得到的 h 组不同的线性投影（linear projections）来变换查询、键和值。然后，这 h 组变换后的查询、键和值将并行地送到注意力汇聚中。最后，将这 h 个注意力汇聚的输出拼接在一起，并且通过另一个可以学习的线性投影进行变换，以产生最终输出。这种设计被称为多头注意力（multihead attention）（Vaswani et al., 2017）。对于 h 个注意力汇聚输出，每一个注意力汇聚都被称作一个头（head）。图10.5.1展示了使用全连接层来实现可学习的线性变换的多头注意力。

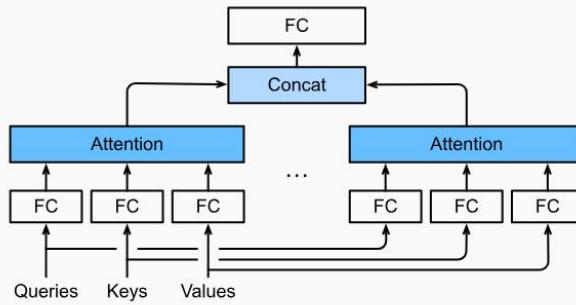


Fig. 11.5.1 Multi-head attention, where multiple heads are concatenated then linearly transformed.

全连接层：FC

在实现多头注意力之前，让我们用数学语言将这个模型形式化地描述出来。给定查询 $\mathbf{q} \in \mathbb{R}^{d_q}$ 、键 $\mathbf{k} \in \mathbb{R}^{d_k}$ 和值 $\mathbf{v} \in \mathbb{R}^{d_v}$ ，每个注意力头 \mathbf{h}_i ($i = 1, \dots, h$) 的计算方法为：

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)} \mathbf{q}, \mathbf{W}_i^{(k)} \mathbf{k}, \mathbf{W}_i^{(v)} \mathbf{v}) \in \mathbb{R}^{p_v}, \quad (10.5.1)$$

其中，可学习的参数包括 $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$ 、 $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ 和 $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ ，以及代表注意力汇聚的函数 f 。 f 可以是10.3节中的加性注意力和缩放点积注意力。多头注意力的输出需要经过另一个线性转换，它对应着 h 个头连结后的结果，因此其可学习参数是 $\mathbf{W}_o \in \mathbb{R}^{p_o \times h p_v}$ ：

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}. \quad (10.5.2)$$

基于这种设计，每个头都可能会关注输入的不同部分，可以表示比简单加权平均值更复杂的函数。

在实现过程中通常选择缩放点积注意力作为每一个注意力头。为了避免计算代价和参数代价的大幅增长，我们设定 $p_q = p_k = p_v = p_o/h$ 。值得注意的是，如果将查询、键和值的线性变换的输出数量设置为 $p_q h = p_k h = p_v h = p_o$ ，则可以并行计算 h 个头。在下面的实现中， p_o 是通过参数 num_hiddens 指定的。

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

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

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{\text{model}}/h = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

单个 head 的计算是：

$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$	矩阵	维度
其中每个 W_i^Q, W_i^K, W_i^V 的形状是：	Q_i	$(batch_size, seq_len, d_k)$
$W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k} \quad W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v} \quad K_i$		$(batch_size, seq_len, d_k)$
输入 $X \in \mathbb{R}^{batch_size \times seq_len \times d_{\text{model}}}$	V_i	$(batch_size, seq_len, d_v)$

然后注意力计算：

$$\text{Attention}(Q_i, K_i, V_i)$$

Attention的输入Q和K的点积得到**注意力权重**：

$$\text{score}(Q_i, K_i) = Q_i K_i^T \quad (\text{size: } (batch_size, seq_len, seq_len))$$

然后用权重乘上 V_i (Broadcast over seq_len) :

最终得到输出：

$$\text{head}_i \in \mathbb{R}^{batch_size \times seq_len \times d_v}$$

在Multi-Head Attention里，多个head的输出要**拼接 (Concat) **在一起，再经过一个最终的线性变换。

拼接之后的张量维度是：

$$\text{Concat}(\text{head}_1, \dots, \text{head}_h) \in \mathbb{R}^{batch_size \times seq_len \times (h \times d_v)}$$

- h : head数目

- d_v : 每个head输出的维度

然后要把这个长宽是 $h \times d_v$ 的特征压回原来的 d_{model} 维空间（否则无法与残差连接匹配）。

所以引入一个输出映射矩阵 W^O ，定义形状为：

$$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$

解释：

- 输入: $(batch_size, seq_len, hd_v)$
- 经过 W^O : 得到 $(batch_size, seq_len, d_{\text{model}})$

整个 Multi-Head Attention 的输出是：

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

因此，最终输出的维度是：

$$(batch_size, seq_len, d_{\text{model}})$$

✓ 直观理解：

- 输入是 d_{model} 维
- Multi-Head Attention 只是做了内部特征处理和不同子空间的信息融合
- 输出还是 d_{model} 维，以便和残差连接（Residual Connection）匹配！

组件	维度
单个head输出 head_i	$(batch_size, seq_len, d_v)$
拼接后的输出	$(batch_size, seq_len, h \times d_v)$
W^O 矩阵	(hd_v, d_{model})
MultiHead最终输出	$(batch_size, seq_len, d_{\text{model}})$

Multi-Head Attention里每个head输出维度是 $(batch, seq_len, d_v)$ ，拼接后用 W^O 把 (hd_v) 映射回 d_{model} ，保证整体输入输出一致。

因为在标准实现中：

- d_q (Query的每个head的维度) 就是直接设成 d_k 。
- Query和Key之间是点积计算相似度的，需要维度完全一致才能进行点积操作。

点积注意力公式：

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

- 如果 Q 和 K 维度不一样，点积 QK^T 就没有定义。
- 所以默认让 $d_q = d_k$ ，直接忽略单独讲。

Multi-Head Attention的逻辑是：

- 把 d_{model} 分成 h 个子空间，每个子空间独立学习不同的注意力关系。
- 每个 head 负责一小块低维特征 (d_k, d_v) 。

如果不能整除，子空间就分不均匀，结构就不统一，比如每个头特征数不同，会导致：

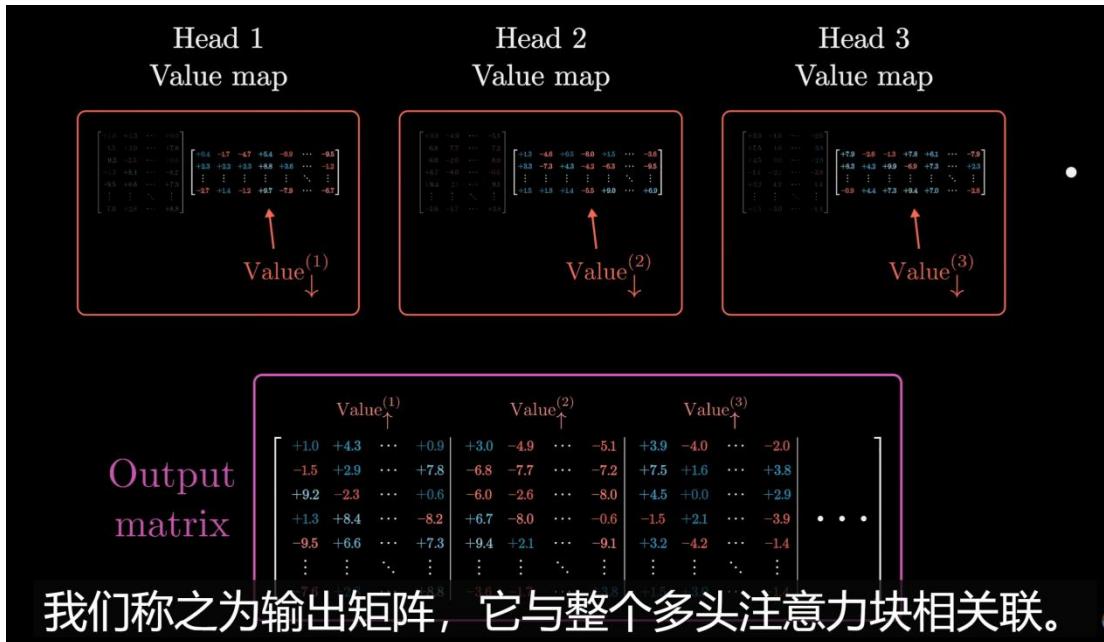
- 实现上复杂
- 并行化难度大
- 后面 Concatenation (拼接所有 head 输出) 时无法对齐维度

所以，为了实现清晰且高效，每个 head 的输出维度设成：

$$d_k = d_v = \frac{d_{\text{model}}}{h}$$

✓ 统一每个head的子空间维度，方便并行、实现简洁、后续拼接不会出错。

P.S.



一般，谈论某个 attention head 的值矩阵时，通常是指第一步，即所有的值向下投影到小空间的步骤。

数据在 Transformer 中的流动并不局限于单个注意力模块，还会经过多层感知机 Multilayer Perceptron。数据会反复经过这两种操作的多个副本。

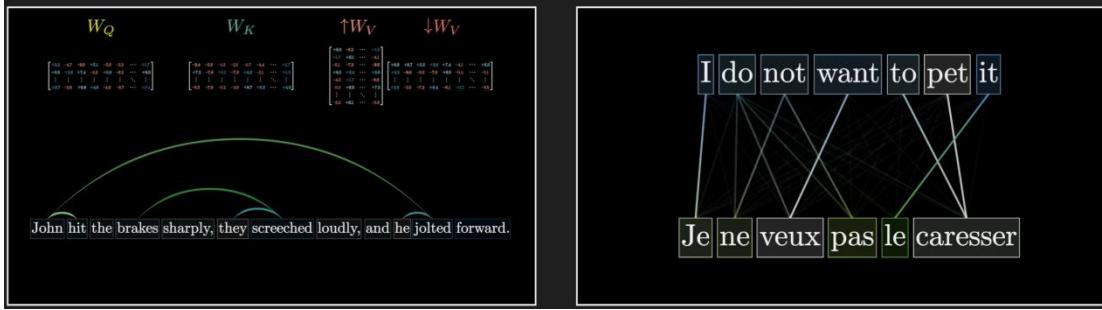
意味着，一个单词在吸收了一些上下文信息后，这个更细致的 embedding 仍有更多的机会受到其周围这更为细致环境的影响。你在网络中越深入，每个 embedding 从所有其他 embedding 中获取的含义就越多，这些 embedding 本身也变得越来越复杂。

用于各种 attention head 的参数数量占整个网络总参数的三分之一左右，大部分参数来自于这些步骤之间的模块。

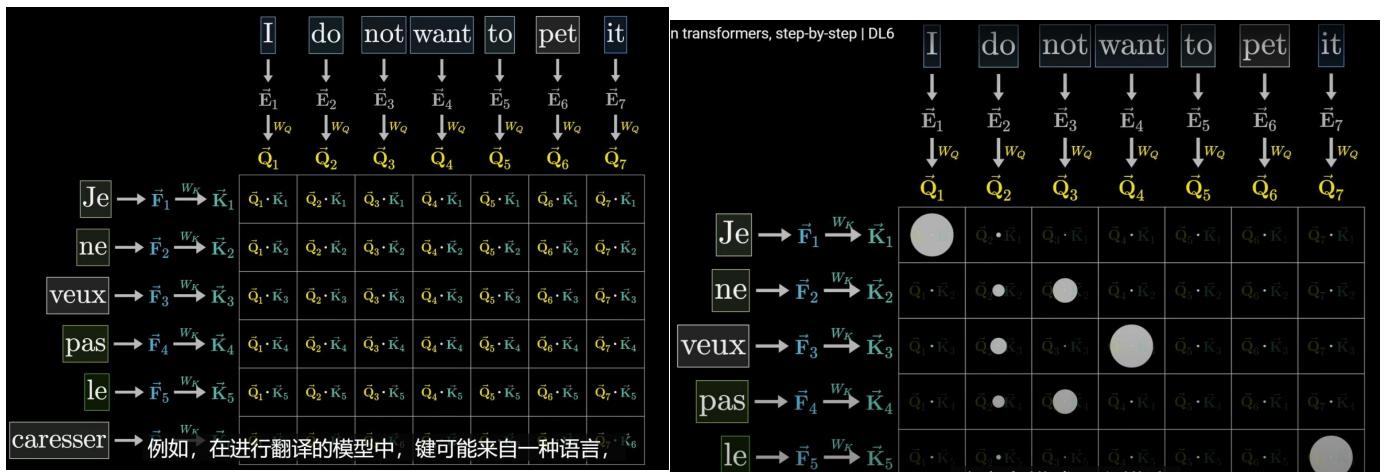
3. Self-Attention & Cross-Attention

Self-attention

Cross-attention

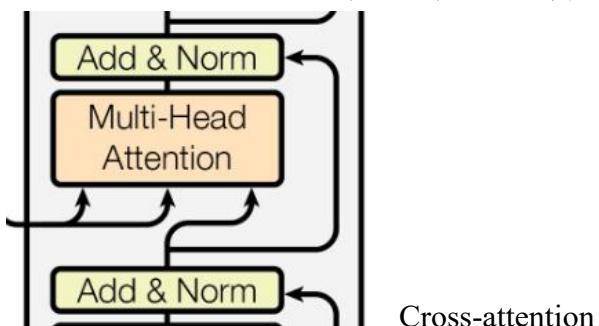


交叉注意力涉及的模型会处理两种不同类型的数据，比如一种语言的文本和正在翻译的另一种语言的文本，唯一的区别是，键和查询映射在交叉注意力机制中会作用于不同的数据集。

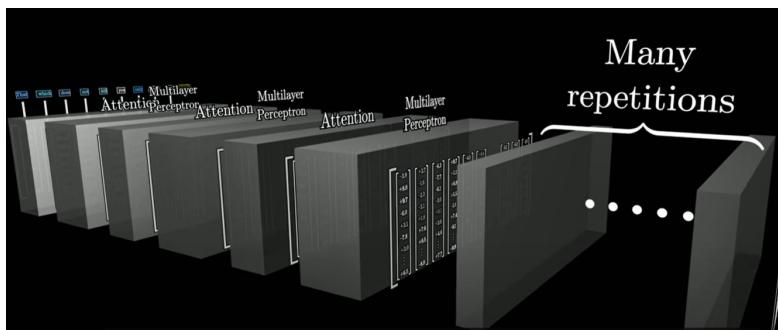


一种语言的哪些词对应另一种语言的哪些词。

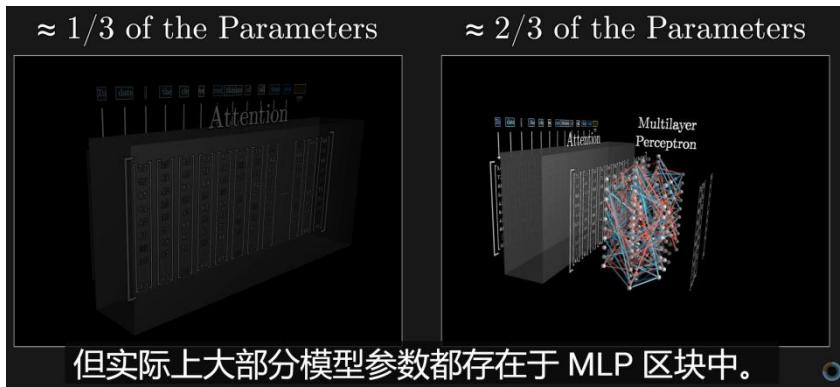
在这种情况下，通常不会有遮蔽，因为并不存在后面的词会影响前面的词的概念。



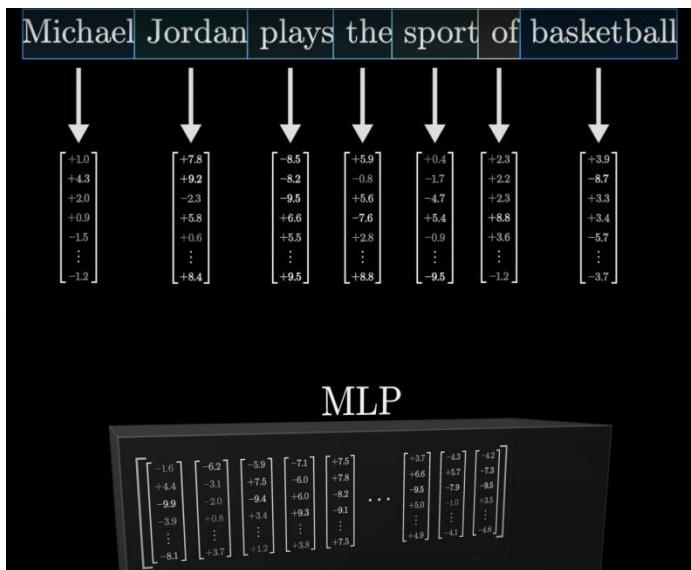
4. MLP (Multilayer Perceptron)



在向量序列经过两个模块的多次迭代后，希望每个向量都能够吸收足够多的信息。



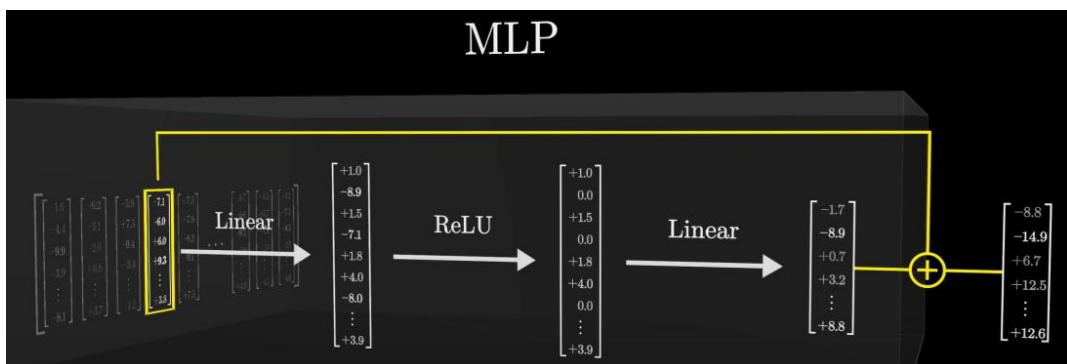
(阉割版例子：嵌入向量与某个含义的向量点积为 1，即该嵌入向量编译了那个含义。假设 attention 已经将信息传递给连续的两个 token 中的第二个 token 对应的向量)



MLP

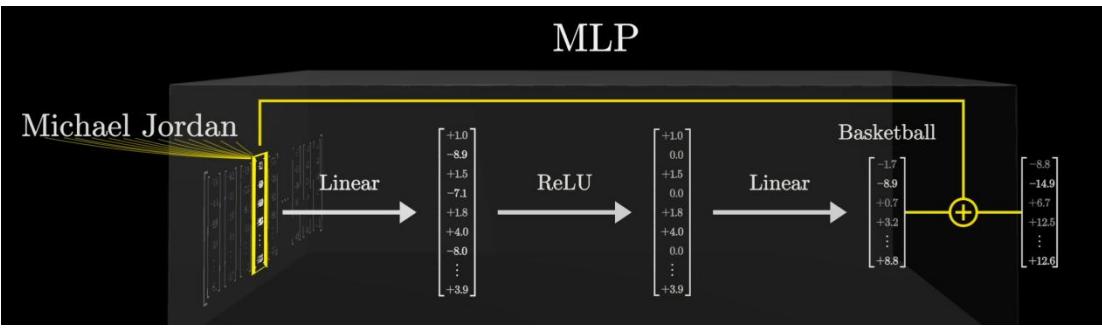


可以把这一连串的向量想象成流入程序块，每个向量最初都与输入文本中的一个标记相关联。经过一系列运算后，最后得到另一个具有相同维度的向量。



另一个矢量将于流入的原始矢量相加，然后得出流出的结果。

这一连串的操作会应用到序列中的每个向量，并与输入中的每个标记相关联，所有操作都是并行进行的。特别是，在这一步中，矢量之间并不对话，它们都在做自己的事情。当我说这个区块将对迈克尔·乔丹打篮球这一事实进行编码时“Michael Jordan plays Basketball”，如果有一个向量输入，其中编码了迈克尔的名字和乔丹的姓氏，那么这一连串的计算就会产生包含篮球这个方向的结果，这就是在该位置对向量进行添加的结果。



这个过程的第一步看起来就像用一个很大的矩阵乘以该向量。这个矩阵就像我们看到的其他矩阵一样，充满了从数据中学习到的模型参数，

$$\begin{array}{c} \text{First Name Michael} \\ \parallel \\ \left[\begin{array}{c} \vec{R}_0 \\ \vec{R}_1 \\ \vec{R}_2 \\ \vdots \\ \vec{R}_n \end{array} \right] \left[\begin{array}{c} | \\ \vec{E} \\ | \end{array} \right] = \left[\begin{array}{c} \vec{R}_0 \cdot \vec{E} \\ \vec{R}_1 \cdot \vec{E} \\ \vec{R}_2 \cdot \vec{E} \\ \vdots \\ \vec{R}_n \cdot \vec{E} \end{array} \right] \end{array} \quad \left[\begin{array}{c} \vec{R}_0 \\ \vec{R}_1 \\ \vec{R}_2 \\ \vdots \\ \vec{R}_n \end{array} \right] \left[\begin{array}{c} | \\ \vec{E} \\ | \end{array} \right] = \left[\begin{array}{c} \vec{R}_0 \cdot \vec{E} \\ \vec{R}_1 \cdot \vec{E} \\ \vec{R}_2 \cdot \vec{E} \\ \vdots \\ \vec{R}_n \cdot \vec{E} \end{array} \right] = \left\{ \begin{array}{ll} \approx 1 & \text{If } \vec{E} \text{ encodes "First Name Michael"} \\ \leq 0 & \text{If not} \end{array} \right.$$

向量之间进行一系列点乘，我将用 E 表示嵌入。例如，假设第一行恰好等于我们假定存在的 Michael direction 这个名字。这就意味着，如果该向量编码的是迈克尔这个名字，那么输出的第一个分量，也就是这里的点积，就是 1。

如果第一排是 Michael 加 Jordan 的 direction，

$$\begin{array}{c} \text{F.N. Michael} + \text{L.N. Jordan} \\ \parallel \\ \left[\begin{array}{c} \vec{R}_0 \\ \vec{R}_1 \\ \vec{R}_2 \\ \vdots \\ \vec{R}_n \end{array} \right] \left[\begin{array}{c} | \\ \vec{E} \\ | \end{array} \right] = \left[\begin{array}{c} \vec{R}_0 \cdot \vec{E} \\ \vec{R}_1 \cdot \vec{E} \\ \vec{R}_2 \cdot \vec{E} \\ \vdots \\ \vec{R}_n \cdot \vec{E} \end{array} \right] = (\vec{M} + \vec{J}) \cdot \vec{E} = \underbrace{\vec{M} \cdot \vec{E} + \vec{J} \cdot \vec{E}}_{\approx 2 \text{ if } \vec{E} \text{ encodes "Michael Jordan"}}, \underbrace{\leq 1}_{\leq 1 \text{ Otherwise}} \end{array} \quad \begin{array}{c} \text{First Name Michael} \\ \text{Last Name Jordan} \end{array}$$

你可以把所有其他行看作是在并行地提出一些其他类型的问题，探究被处理向量的一些其他类型的特征。很多时候，这一步还涉及向输出中添加另一个向量，其中包含从数据中学习到的模型参数。这另一个向量被称为偏差。

$$\begin{array}{c} \text{Bias} \\ \left[\begin{array}{cccccc} -5.0 & +7.1 & +0.8 & +1.0 & \cdots & +6.8 \\ -7.4 & -4.4 & +1.7 & +9.3 & \cdots & +1.2 \\ -9.5 & +6.0 & -5.3 & +6.1 & \cdots & -2.2 \\ +7.2 & +4.9 & +1.1 & -7.2 & \cdots & -8.7 \\ -7.5 & -9.0 & -7.8 & -5.4 & \cdots & +4.2 \\ +1.2 & -9.7 & -8.5 & +9.3 & \cdots & +1.3 \\ -5.9 & -4.9 & +4.8 & -6.0 & \cdots & +1.6 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ +9.3 & +6.9 & -5.2 & -0.1 & \cdots & +2.4 \end{array} \right] \left[\begin{array}{c} -7.1 \\ -6.0 \\ +6.0 \\ +9.3 \\ \vdots \\ +3.8 \end{array} \right] + \left[\begin{array}{c} +1.0 \\ -5.7 \\ +5.0 \\ -8.6 \\ -4.7 \\ +6.0 \\ -6.1 \\ +4.0 \\ \vdots \\ +2.8 \end{array} \right] = \left[\begin{array}{c} +1.0 \\ -8.9 \\ +15 \\ -7.1 \\ +18 \\ +6.0 \\ -8.0 \\ +4.0 \\ \vdots \\ +3.9 \end{array} \right] = \underbrace{\vec{M} \cdot \vec{E} + \vec{J} \cdot \vec{E} - 1}_{\approx 1 \text{ if } \vec{E} \text{ encodes "Michael Jordan"}}, \underbrace{\leq 0}_{\leq 0 \text{ Otherwise}} \end{array}$$

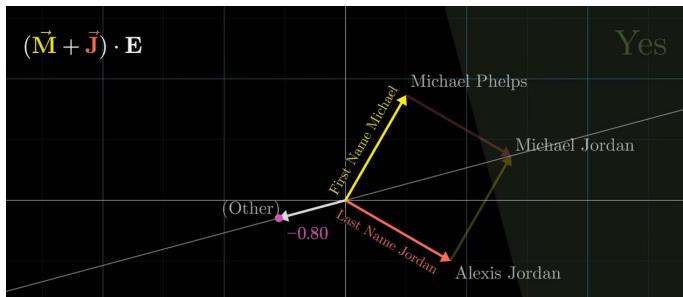
如果我们在这里设置一个值，当且仅当一个向量编码了迈克尔·乔丹的全名时，这个值就是正值，否则就是零或负值，那就非常干净利落了。

12,288

$$4 \times 12,288 \left\{ \begin{bmatrix} -5.0 & +7.1 & +0.8 & +1.0 & \cdots & +6.8 \\ -7.4 & -4.4 & +1.7 & +9.3 & \cdots & +1.2 \\ -9.5 & +6.0 & -5.3 & +6.1 & \cdots & -2.2 \\ +7.2 & +4.9 & +1.1 & -7.2 & \cdots & -8.7 \\ -7.5 & -9.0 & -7.8 & -5.4 & \cdots & +4.2 \\ +1.2 & -9.7 & -8.5 & +9.3 & \cdots & +1.3 \\ -5.9 & -4.9 & +4.8 & -6.0 & \cdots & +1.6 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ +9.3 & +6.9 & -5.2 & -0.1 & \cdots & +2.4 \end{bmatrix} \right. + \begin{bmatrix} -1.0 \\ -7.1 \\ -6.0 \\ +6.0 \\ +9.3 \\ \vdots \\ +3.8 \end{bmatrix} = \begin{bmatrix} +1.0 \\ -8.9 \\ +1.5 \\ -8.6 \\ -4.7 \\ +6.0 \\ -6.1 \\ +2.8 \end{bmatrix} + \begin{bmatrix} +1.0 \\ -8.9 \\ +1.5 \\ -7.1 \\ +1.8 \\ +4.0 \\ -8.0 \\ +3.9 \end{bmatrix}$$

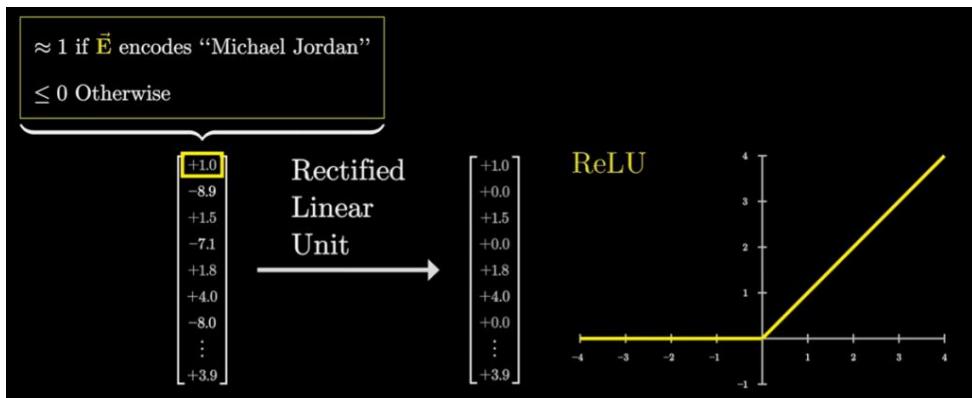
这个矩阵的总行数 GPT-3 的问题数量，略低于 50,000 行。事实上，它正好是这个嵌入空间维数的四倍。这是一个设计选择，可以更多也可以更少，但是拥有干净的多路径对硬件会更好。

由于这个充满权重的矩阵将我们映射到了一个更高的维度空间，把它简称为 W_{up} 。继续将我们正在处理的矢量标记为 E ，偏置矢量标记为 B ，然后将它们全部放回图中。



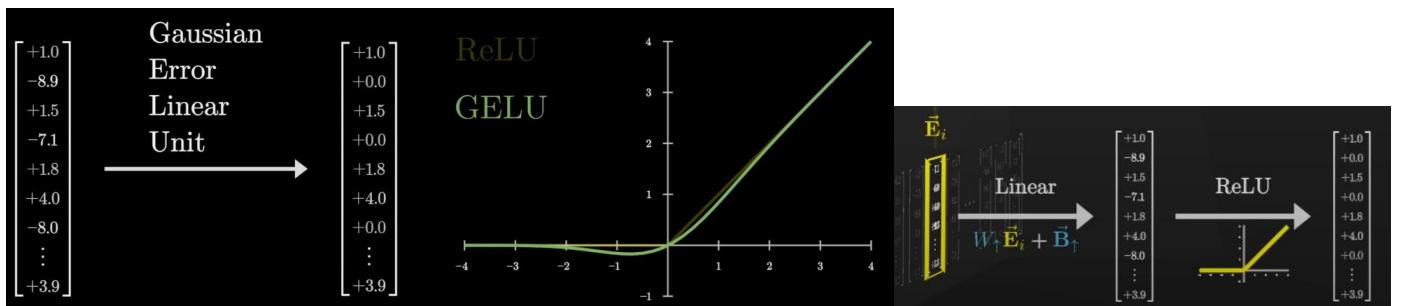
刚才的操作是纯线性的，但语言是非线性的。如果我们测量的迈克尔加乔丹的入选率很高，那么迈克尔加菲尔普斯和亚历克西斯加乔丹也必然会在一定程度上引发入选率，尽管这两者在概念上并不相关。您真正需要的是全名的简单“是”或“否”。

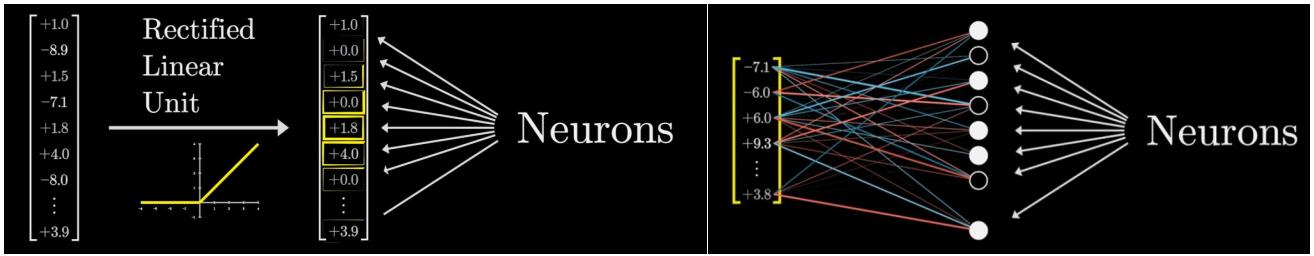
因此，下一步就是通过一个非常简单的非线性函数来传递这个大的中间向量。常见的选择是将所有负值映射为零，而所有正值保持不变。整流线性单元，简称 ReLU。



因此，以我们想象中的例子为例，中间向量的第一个条目是 1（如果且仅当全名是迈克尔·乔丹），否则就是 0 或负值。换句话说，它非常直接地模拟了 AND 门的行为。

通常情况下，模型会使用一种略有改动的功能，称为 GELU。其基本形状相同，只是更平滑一些。





Transformer 只考虑 ReLU。当你听到人们提到 Transformer 的神经元 neurons 时，他们说的就是这里的这些值。

这通常是为了表达线性步骤的组合，即矩阵乘法，然后是一些简单的项向非线性函数，如 ReLU。

“Michael Jordan” neuron is *active* “Michael Jordan” neuron is *inactive*

[+1.0]

[+0.0]

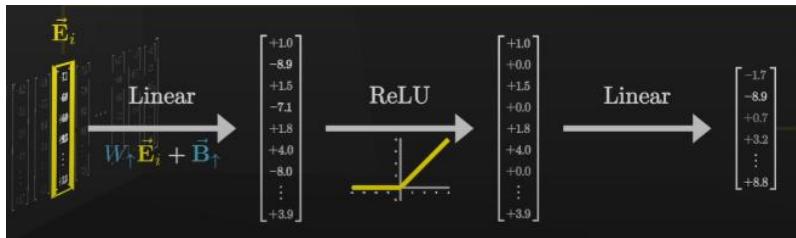
值为正，这个神经元就处于激活状态；值为 0，这个神经元就处于非激活状态。

下一步看起来与第一步非常相似。

乘以一个非常大的矩阵，然后加上一些偏差项。输出中的维数又回到了嵌入空间的大小，所以我把它叫做向下投影矩阵。上次逐行思考，此次逐列思考。

“Down projection”

$$\left\{ \begin{array}{c} \overbrace{\text{[+0.5, +8.4, -4.7, -8.6, +4.7, +5.4, +8.1, \dots, -9.6, \\ -5.3, +2.3, +8.9, +8.9, +1.1, +8.2, +2.8, \dots, -0.3, \\ +2.1, +1.0, +8.4, +8.3, -2.1, +9.2, -6.5, \dots, -7.2, \\ +0.1, -9.5, +8.9, +6.5, -9.6, -6.4, -3.3, \dots, +6.1, \\ \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \\ -4.2, -0.2, +2.0, -9.6, +1.9, -1.3, +6.1, \dots, +7.8]}^{\text{A large matrix}} & + & \text{[+1.0, +0.0, +1.5, +0.0, +1.8, +4.0, +0.0, \dots, +3.9]} \\ \text{[+1.5, -6.3, +5.7, +2.2, +4.0, +0.0, \vdots, -1.6]} & = & \text{[-1.7, -8.9, +0.7, +3.2, \vdots, +8.8]} \end{array} \right\} 12,288$$



$$n_0 \vec{C}_0 + n_1 \vec{C}_1 + n_2 \vec{C}_2 + n_3 \vec{C}_3 + n_4 \vec{C}_4 + \dots + n_m \vec{C}_m$$

$$\left\{ \begin{array}{c} \overbrace{\left[\vec{C}_0 \mid \vec{C}_1 \mid \vec{C}_2 \mid \vec{C}_3 \mid \vec{C}_4 \mid \dots \mid \vec{C}_m \right]}^{\text{A column of vectors}} & + & \left[\begin{array}{c} n_0 \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ \vdots \\ n_m \end{array} \right] \\ \left[\begin{array}{c} | \\ | \\ | \\ | \\ | \\ | \\ | \end{array} \right] & = & \left[\begin{array}{c} -1.7 \\ -8.9 \\ +0.7 \\ +3.2 \\ \vdots \\ +8.8 \end{array} \right] \end{array} \right\}$$

将矩阵的每一列乘以它所处理的向量中的相应项，然后将所有这些重新缩放的列相加。

用这种方法来思考会更好，因为这里的列与嵌入空间的维度相同，所以我们可以把它们看作是嵌入空间中的方向。

$$1\vec{\mathbf{C}}_0 + n_1\vec{\mathbf{C}}_1 + n_2\vec{\mathbf{C}}_2 + n_3\vec{\mathbf{C}}_3 + n_4\vec{\mathbf{C}}_4 + \cdots + n_m\vec{\mathbf{C}}_m$$

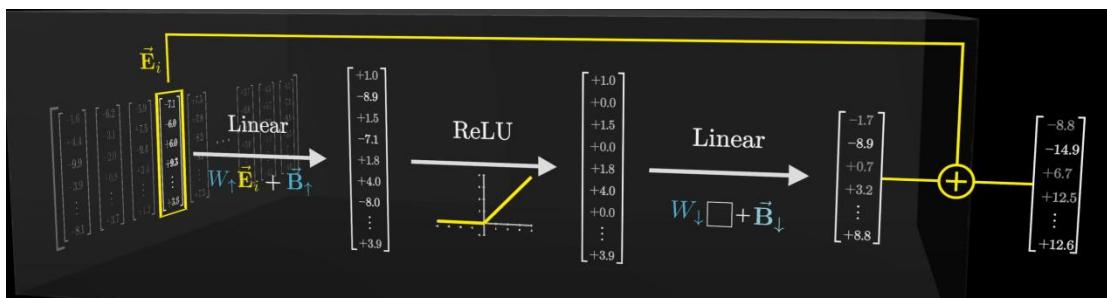
Basketball

$$\underbrace{\left[\begin{array}{c|c|c|c|c|c|c} \vec{\mathbf{C}}_0 & \vec{\mathbf{C}}_1 & \vec{\mathbf{C}}_2 & \vec{\mathbf{C}}_3 & \vec{\mathbf{C}}_4 & \cdots & \vec{\mathbf{C}}_m \end{array} \right]}_{\text{Basketball}} \left[\begin{array}{c} n_0 \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ \vdots \\ n_m \end{array} \right] + \left[\begin{array}{c} \vec{\mathbf{B}} \end{array} \right] = \left[\begin{array}{c} -1.7 \\ -8.9 \\ +0.7 \\ +3.2 \\ \vdots \\ +8.8 \end{array} \right]$$

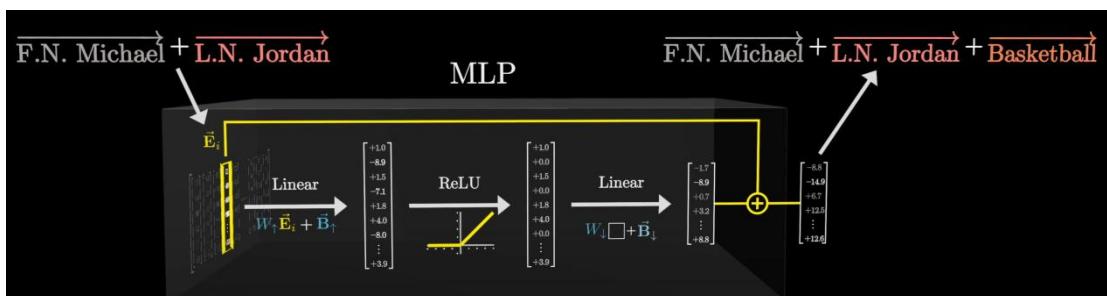
Basketball
 +
 Chicago Bulls
 +
 Number 23
 +
 Born 1963
 +
 :

$$0\vec{\mathbf{C}}_0 + n_1\vec{\mathbf{C}}_1 + n_2\vec{\mathbf{C}}_2 + n_3\vec{\mathbf{C}}_3 + n_4\vec{\mathbf{C}}_4 + \cdots + n_m\vec{\mathbf{C}}_m$$

例如，模型已经学会将第一列插入我们假设存在的 basketball 方向，这意味着，当第一个位置上的相关神经元处于活动状态时，我们将在最终结果中添加这一列。但是，如果该神经元没有激活，该数字为 0，那就不会有任何影响。而且不一定非得是篮球，该模型还可以在一列和其他许多功能中加入迈克尔·乔丹全名的元素。同时，这个矩阵中的所有其他列都在告诉你，如果相应的神经元处于激活状态，最终结果会增加什么。在这种情况下，如果存在偏差，那么无论神经元的值是多少，每次都会产生偏差。



为使符号简洁，把这个大矩阵 W 称为 "W down"，同样把偏置向量 B 称为 "B down"。



例如，如果输入的向量同时编码了迈克尔和乔丹这两个名字，那么由于这一连串的操作会触发 AND 门，因此会在篮球方向上进行加法运算，这样跳出来的向量就会把这两个名字一起编码。

请记住，这是一个并行发生在每个矢量上的过程。

整个操作过程：两个矩阵乘积，每个乘积都添加了偏置，中间还有一个简单的削波功能。

14:57 / 22:42 之后未看 <https://www.youtube.com/watch?v=9-Jl0dxWQs8&t=29s>

multilayer perceptron MLP这个模型的背景，应用场景，数学基础，模型参数，权重衰减在其中是什么情况，dropout在其中是如何应用的，优点和缺点是什么，扩展延伸

虽然现在有更复杂的网络 (CNN, Transformer) , 但MLP在很多场景仍然非常有用, 尤其是:

- **结构化数据** (表格数据、金融、医疗领域预测)
- **特征工程后的分类与回归任务** (如Kaggle竞赛)
- **嵌入学习** (比如在Transformer、BERT中, MLP用于处理特征)
- **生成模型中的子模块** (比如GAN中的判别器/生成器)
- **多模态融合** (比如音频、文本、图像特征拼接后经过MLP融合)

✓ 一句话总结:

MLP 是一种通用的近似器 (Universal Approximator) , 适合各种非结构化或者结构化的输入特征。

一个典型的MLP包含:

- **输入层 → 隐藏层 (一个或多个) → 输出层。**
- **每一层都执行:**

$$h^{(l)} = f(W^{(l)}h^{(l-1)} + b^{(l)})$$

- $h^{(l-1)}$: 前一层的输出
- $W^{(l)}$: 当前层的权重矩阵
- $b^{(l)}$: 当前层的偏置向量
- f : 非线性激活函数 (如ReLU、Tanh、Sigmoid)

反向传播:

- 计算输出误差 (loss) , 如MSE或Cross-Entropy。
- 通过链式法则 (Chain Rule) 反向更新权重。

本质: MLP 就是一系列线性变换 + 非线性激活堆叠的过程, 本质上是通过不断调整权重来拟合输入到输出的复杂关系。

4. 模型参数

- **权重矩阵 $W^{(l)}$** : 每一层从输入到输出的映射参数。
- **偏置向量 $b^{(l)}$** : 每个神经元独有的偏置项。
- 总参数量:

$$\sum_l \left(\text{input_dim}^{(l)} \times \text{output_dim}^{(l)} + \text{output_dim}^{(l)} \right)$$

- 训练参数时, 通常使用**随机梯度下降 (SGD) **或者其改进版 (如 Adam) 。

5. 权重衰减 (Weight Decay) 在 MLP 中的作用

- **定义:** 训练时在 loss 上添加一个权重平方惩罚项:

$$L_{\text{total}} = L_{\text{task}} + \lambda \sum_l \|W^{(l)}\|_2^2$$

- 其中 λ 是权重衰减系数 (正则化强度)。

- **作用:**

- 防止模型过拟合。
- 让权重不会变得太大，保证模型的复杂度受控。

✓ 本质是L2正则化，对每个权重施加“收缩”压力。

6. Dropout 在 MLP 中的应用

- **定义:** 在训练过程中，随机将一部分神经元的输出置为0 (以概率p)，防止神经元之间共适应。

- **应用位置:**

- 通常在隐藏层输出后，激活函数之后添加Dropout层。

- **公式:**

$$h_{\text{drop}} = h \odot \text{Bernoulli}(p)$$

- \odot : 逐元素乘
- Bernoulli(p)是伯努利分布采样的掩码。

- **推理阶段 (Inference) :**

- Dropout不再随机关闭节点，而是使用全连接的节点，同时将输出缩放 $(1 - p)$ 倍。

✓ **作用:**

- 减少过拟合。
- 增加模型的鲁棒性 (robustness)。

7. MLP 的优点和缺点

优点	缺点
理论上可以拟合任何函数 (通用逼近定理)	对输入特征非常敏感，需要手工设计特征
结构简单，易于实现	对高维输入 (如图像) 不友好，参数量爆炸
可以适配分类、回归各种任务	缺乏空间/时间局部性建模能力
能作为其他复杂模型 (如Transformer) 的组件	训练深层MLP容易梯度消失或爆炸 (虽然现代优化器能缓解)

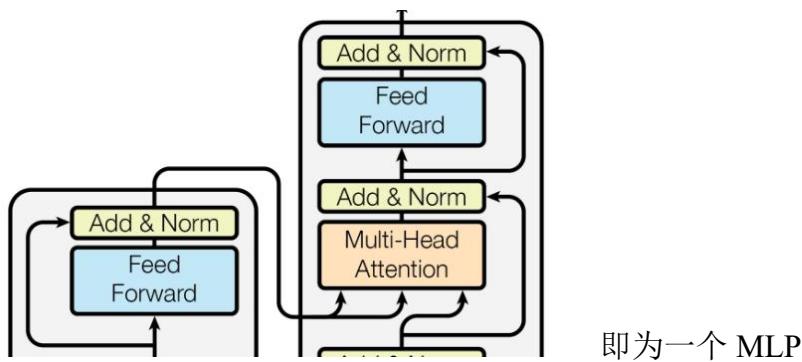
8. 扩展与延伸

MLP的思想后来演变出了各种更强的网络：

- **卷积神经网络 (CNN)**: 引入卷积操作，解决了图像特征局部建模问题。
- **循环神经网络 (RNN, LSTM, GRU)**: 适合处理序列数据。
- **Transformer**: 彻底抛弃了序列结构，基于自注意力机制。
- **MLP-Mixer (2021)**:
 - 放弃自注意力机制，重新回到MLP架构，只用两个方向 (token-mixing 和 channel-mixing) 的MLP，竟然可以达到Transformer级别的效果。
- **ResMLP, gMLP**:
 - 引入了残差连接、门控机制的高级MLP架构，进一步提升纯MLP模型的表现。

5. Position-wise Feed-Forward Networks

基于位置的前馈网络



基于位置的前馈网络对序列中的所有位置的表示进行变换时使用的是同一个多层感知机 (MLP)，这就是称前馈网络是基于位置的 (positionwise) 的原因。在下面的实现中，输入x的形状 (批量大小，时间步数或序列长度，隐单元数或特征维度) 将被一个两层的感知机转换成形状为 (批量大小，时间步数， $ffn_num_outputs$) 的输出张量。

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2)$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is $d_{model} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

6. Add & Norm

(Residual Connection and Layer Normalization) 残差连接和层规范化



现在让我们关注 [图10.7.1](#) 中的加法和规范化 (add&norm) 组件。正如在本节开头所述，这是由残差连接和紧随其后的层规范化组成的。两者都是构建有效的深度架构的关键。

[7.5节](#) 中解释了在一个小批量的样本内基于批量规范化对数据进行重新中心化和重新缩放的调整。层规范化和批量规范化目标相同，但层规范化是基于特征维度进行规范化。尽管批量规范化在计算机视觉中被广泛应用，但在自然语言处理任务中（输入通常是变长序列）批量规范化通常不如层规范化效果好。

现在可以使用残差连接和层规范化来实现AddNorm类。暂退法也被作为正则化方法使用。

残差连接后进行层规范化。残差连接要求两个输入的形状相同，以便加法操作后输出张量的形状相同。
wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$, where $\text{Sublayer}(x)$ is the function implemented by the sub-layer
标准 Transformer 中 Add & Norm 的结构：

❖ Add: Residual Connection (残差连接)

❖ Norm: Layer Normalization (层规范/一化)

P.S. 暂退法 Dropout 是额外插在子层输出后的正则化步骤

以Transformer Encoder的子层结构为例，每一块子层 (Self-Attention层或者FeedForward层) 都这么处理：

1. 子层计算 (比如Self-Attention或者FeedForward)
2. Dropout (对子层输出进行dropout，避免过拟合)
3. 残差连接 (将子层输出与原输入相加)
4. Layer Normalization (对相加后的结果归一化)

用公式描述就是：

$\text{AddNorm}(x) = \text{LayerNorm}(x + \text{Dropout}(\text{SubLayer}(x)))$		
步骤	操作	说明
子层SubLayer	子模块 (比如Self-Attention或MLP)	子层核心计算
Dropout	随机丢弃神经元	作为正则化
Add	残差连接 $x + \text{Dropout}(\text{SubLayer}(x))$	解决深层网络梯度消失，方便信息流动
Norm	Layer Normalization	稳定训练，加速收敛

P.S. 结合数据公式详细讲解 Dropout 在 MLP 中的应用，Dropout 在 Transformer 中的应用，MLP 在 Transformer 中的应用

首先，标准 MLP 的一层结构是：

$$h^{(l)} = f(W^{(l)}h^{(l-1)} + b^{(l)})$$

- $h^{(l-1)}$: 上一层的输出
- $W^{(l)}$: 当前层权重矩阵
- $b^{(l)}$: 偏置
- f : 激活函数 (如ReLU)

引入 Dropout 后的公式是：

在每层输出后添加一个随机屏蔽机制：

$$\tilde{h}^{(l)} = \text{Dropout}(h^{(l)})$$

即：

$$\tilde{h}^{(l)} = m^{(l)} \odot h^{(l)}$$

其中：

- $m^{(l)} \sim \text{Bernoulli}(p)$: 每个神经元独立以概率 p 保留, $1 - p$ 的概率置零。
- \odot : 逐元素乘法。
- p 通常设置为 0.5 或 0.8。

然后, 把 $\tilde{h}^{(l)}$ 作为下一层的输入!

推理阶段 (Inference时处理) :

- 不再随机Drop节点。
- 统一缩放激活值, 即乘以 p 。
- 或者, 在训练时直接除以 p 保证期望值一致 (称为 inverted dropout, 反向dropout) 。

核心目的:

- 避免神经元 co-adaptation (互相依赖, 过拟合)
- 让网络更稀疏化, 增加泛化能力。

Transformer中, Dropout也大量使用, 但应用位置稍有不同!

主要用在4个地方:

位置	说明
1. Attention权重后	防止注意力矩阵过拟合, Dropout应用在Softmax后的attention map上
2. Attention输出后	防止注意力输出过拟合, Dropout在MultiheadAttention输出后加
3. 前馈网络 (FeedForward) 输出后	防止MLP过拟合, Dropout在每个Linear层后或激活后加
4. Residual Connection后	残差连接后, 也在加和前后加Dropout, 有助于增加鲁棒性

核心目的在Transformer中:

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

- 防止某些token过度依赖局部注意力。
- 稀疏化信息流, 增强模型的鲁棒性。

注意是对Softmax后的权重加Dropout!

训练时启用, 推理时关闭。

标准MLP子结构是：

$$\text{MLP}(x) = \text{Linear}_2(\text{ReLU}(\text{Linear}_1(x)))$$

即：

1. 第一层线性映射：升维（通常扩大4倍）
2. 激活：ReLU或者GELU
3. 第二层线性映射：降回原始维度
4. （可选）Dropout夹在中间或后面

假设输入 $x \in \mathbb{R}^{\text{batch_size} \times \text{seq_len} \times d_{\text{model}}}$

- 第一层：

$$h = \text{ReLU}(xW_1 + b_1)$$

- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$
- d_{ff} 通常是 $d_{\text{model}} \times 4$
- Dropout层（可选）：

$$h' = \text{Dropout}(h)$$

- 第二层：

$$\text{Output} = h'W_2 + b_2$$

- $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$

✓ 总结下 MLP在Transformer里的功能：

- 局部特征增强：
 - Attention捕捉全局关系，MLP捕捉每个位置的局部特征变换。
 - 增加非线性表达能力。
 - 序列每个位置独立处理，即是 position-wise feedforward。

场景	Dropout应用点	MLP应用点
传统MLP	隐藏层输出后加Dropout	整个模型就是堆叠的MLP
Transformer Attention部分	Softmax后/输出后加Dropout	-
Transformer FFN部分	Linear-ReLU-Linear中加Dropout	两层Linear+激活构成的局部MLP模块

整体 transformer

自注意力同时具有并行计算和最短的最大路径长度这两个优势。

模块	Encoder SubLayers	Decoder SubLayers
子层1	Multi-Head Self-Attention	Masked Multi-Head Self-Attention
子层2	FeedForward Network (FFN)	Multi-Head Cross-Attention
子层3	-	FeedForward Network (FFN)

Masked Multi-Head Self-Attention: 自己对自己做 Attention，但遮挡未来信息（mask）

Multi-Head Cross-Attention: 对 Encoder 输出的序列做 Attention，称为 Cross-Attention

Position-wise FeedForward Network (FFN): 逐位置独立的两层全连接网络（MLP 结构）

每个子层都有：残差连接（Add）+ 层归一化（LayerNorm）

这里最重要的不同点就是第2层：

- Decoder里有专门一层 Cross Attention！
- 让Decoder能读到Encoder编码的上下文信息，实现编码-解码交互。

图 [图10.7.1](#) 中概述了Transformer的架构。从宏观角度来看，Transformer的编码器是由多个相同的层叠加而成的，每个层都有两个子层（子层表示为sublayer）。第一个子层是多头自注意力（multi-head self-attention）汇聚；第二个子层是基于位置的前馈网络（positionwise feed-forward network）。具体来说，在计算编码器的自注意力时，查询、键和值都来自前一个编码器层的输出。受 [7.6节](#) 中残差网络的启发，每个子层都采用了残差连接（residual connection）。在Transformer中，对于序列中任何位置的任何输入 $\mathbf{x} \in \mathbb{R}^d$ ，都要求满足 $\text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ ，以便残差连接满足 $\mathbf{x} + \text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ 。在残差连接的加法计算之后，紧接着应用层规范化（layer normalization）（[Ba et al., 2016](#)）。因此，输入序列对应的每个位置，Transformer编码器都将输出一个 d 维表示向量。

Transformer解码器也是由多个相同的层叠加而成的，并且层中使用了残差连接和层规范化。除了编码器中描述的两个子层之外，解码器还在这两个子层之间插入了第三个子层，称为编码器-解码器注意力（encoder-decoder attention）层。在编码器-解码器注意力中，查询来自前一个解码器层的输出，而键和值来自整个编码器的输出。在解码器自注意力中，查询、键和值都来自上一个解码器层的输出。但是，解码器中的每个位置只能考虑该位置之前的所有位置。这种掩蔽（masked）注意力保留了自回归（auto-regressive）属性，确保预测仅依赖于已生成的输出词元。

在此之前已经描述并实现了基于缩放点积多头注意力 [10.5节](#) 和位置编码 [10.6.3节](#)。接下来将实现 Transformer模型的剩余部分。

10.7.4. 编码器

有了组成 Transformer 编码器的基础组件，现在可以先实现编码器中的一个层。下面的 EncoderBlock 类包含两个子层：多头自注意力和基于位置的前馈网络，这两个子层都使用了残差连接和紧随的层规范化。

正如从代码中所看到的，Transformer 编码器中的任何层都不会改变其输入的形状。

下面实现的 Transformer 编码器的代码中，堆叠了 `num_layers` 个 EncoderBlock 类的实例。由于这里使用的是值范围在 -1 和 1 之间的固定位置编码，因此通过学习得到的输入的嵌入表示的值需要先乘以嵌入维度的平方根进行重新缩放，然后再与位置编码相加。

Transformer 编码器输出的形状是（批量大小，时间步数目，`num_hiddens`）

10.7.5. 解码器

如图 10.7.1 所示，Transformer 解码器也是由多个相同的层组成。在 DecoderBlock 类中实现的每个层包含了三个子层：解码器自注意力、“编码器-解码器”注意力和基于位置的前馈网络。这些子层也都被残差连接和紧随的层规范化围绕。

正如在本节前面所述，在掩蔽多头解码器自注意力层（第一个子层）中，查询、键和值都来自上一个解码器层的输出。关于序列到序列模型（sequence-to-sequence model），在训练阶段，其输出序列的所有位置（时间步）的词元都是已知的；然而，在预测阶段，其输出序列的词元是逐个生成的。因此，在任何解码器时间步中，只有生成的词元才能用于解码器的自注意力计算中。为了在解码器中保留自回归的属性，其掩蔽自注意力设定了参数 `dec_valid_lens`，以便任何查询都只会与解码器中所有已经生成词元的位置（即直到该查询位置为止）进行注意力计算。

为了便于在“编码器-解码器”注意力中进行缩放点积计算和残差连接中进行加法计算，编码器和解码器的特征维度都是 `num_hiddens`。

构建了由 `num_layers` 个 DecoderBlock 实例组成的完整的 Transformer 解码器。最后，通过一个全连接层计算所有 `vocab_size` 个可能的输出词元的预测值。解码器的自注意力权重和编码器解码器注意力权重都被存储下来，方便日后的可视化需要。

10.7.6. 训练

在这里，指定 Transformer 的编码器和解码器都是 2 层，都使用 4 头注意力。

编码器自注意力权重的形状为（编码器层数，注意力头数，`num_steps` 或查询的数目，`num_steps` 或“键一值”对的数目）。

在编码器的自注意力中，查询和键都来自相同的输入序列。因为填充词元是不携带信息的，因此通过指定输入序列的有效长度可以避免查询与使用填充词元的位置计算注意力。每个注意力头都根据查询、键和值的不同的表示子空间来表示不同的注意力。

为了可视化解码器的自注意力权重和“编码器-解码器”的注意力权重，我们需要完成更多的数据操作工作。例如用零填充被掩蔽住的注意力权重。值得注意的是，解码器的自注意力权重和“编码器-解码器”的注意力权重都有相同的查询：即以序列开始词元（beginning-of-sequence, BOS）打头，再与后续输出的词元共同组成序列。由于解码器自注意力的自回归属性，查询不会对当前位置之后的“键一值”对进行注意力计算。

与编码器的自注意力的情况类似，通过指定输入序列的有效长度，输出序列的查询不会与输入序列中填充位置的词元进行注意力计算。

在 Transformer 中，多头自注意力用于表示输入序列和输出序列，不过解码器必须通过掩蔽机制来保留自回归属性。

Transformer 中的残差连接和层规范化是训练非常深度模型的重要工具。

Transformer 模型中基于位置的前馈网络使用同一个多层次感知机，作用是对所有序列位置的表示进行转换。