



Chapter 4

Transformer定义

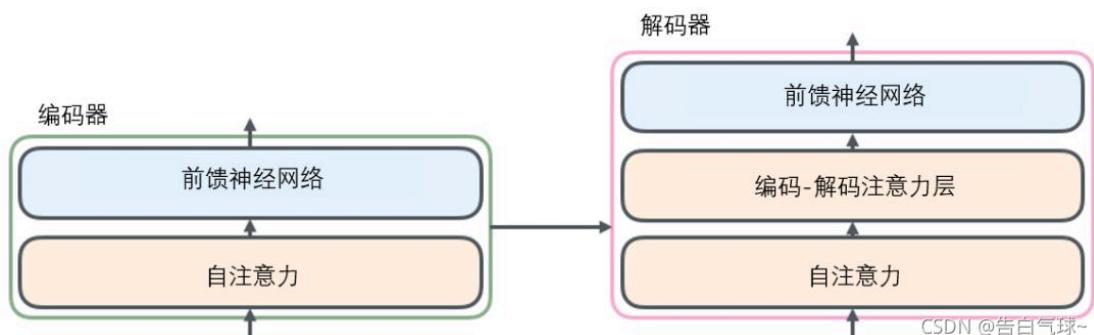
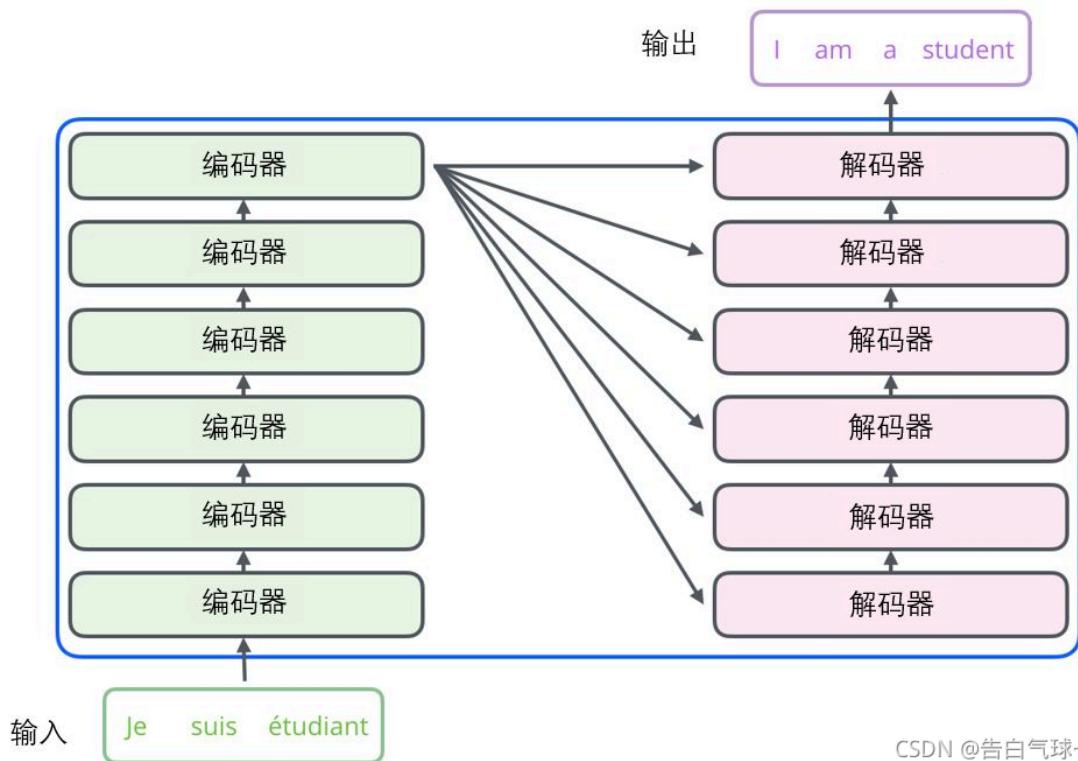
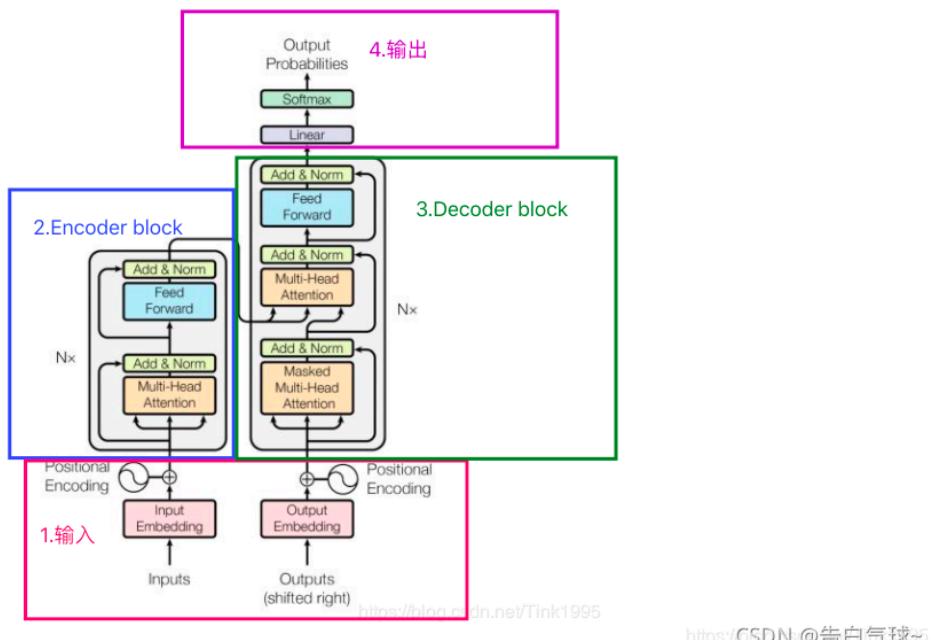
Transformer是一种用于自然语言处理（NLP）和其他序列到序列（sequence-to-sequence）任务的深度学习模型架构，它在2017年由Vaswani等人首次提出。Transformer架构引入了自注意力机制（self-attention mechanism），这是一个关键的创新，使其在处理序列数据时表现出色。

以下是Transformer的一些重要组成部分和特点：

1. 自注意力机制（Self-Attention）：这是Transformer的核心概念之一，它使模型能够同时考虑输入序列中的所有位置，而不是像循环神经网络（RNN）或卷积神经网络（CNN）一样逐步处理。自注意力机制允许模型根据输入序列中的不同部分来赋予不同的注意权重，从而更好地捕捉语义关系。
2. 多头注意力（Multi-Head Attention）：Transformer中的自注意力机制被扩展为多个注意力头，每个头可以学习不同的注意权重，以更好地捕捉不同类型的关系。多头注意力允许模型并行处理不同的信息子空间。
3. 堆叠层（Stacked Layers）：Transformer通常由多个相同的编码器和解码器层堆叠而成。这些堆叠的层有助于模型学习复杂的特征表示和语义。
4. 位置编码（Positional Encoding）：由于Transformer没有内置的序列位置信息，它需要额外的位置编码来表达输入序列中单词的位置顺序。
5. 残差连接和层归一化（Residual Connections and Layer Normalization）：这些技术有助于减轻训练过程中的梯度消失和爆炸问题，使模型更容易训练。
6. 编码器和解码器：Transformer通常包括一个编码器用于处理输入序列和一个解码器用于生成输出序列，这使其适用于序列到序列的任务，如机器翻译。

1-1、Transformer的结构：

$N_x = 6$ ，Encoder block由6个encoder堆叠而成，图中的一个框代表的是一个encoder的内部结构，一个Encoder是由Multi-Head Attention和全连接神经网络Feed Forward Network构成。如下图所示：



自注意力机制

自注意力的作用：随着模型处理输入序列的每个单词，自注意力会**关注整个输入序列的所有单词**，帮助模型对本单词更好地进行编码。在处理过程中，自注意力机制会将对所有相关单词的理解融入到我们正在处理的单词中。更具体的功能如下：

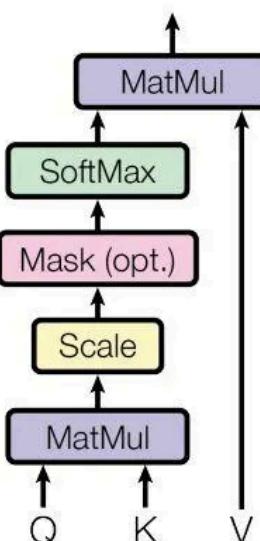
序列建模：自注意力可以用于序列数据（例如文本、时间序列、音频等）的建模。它可以捕捉序列中不同位置的依赖关系，从而更好地理解上下文。这对于机器翻译、文本生成、情感分析等任务非常有用。

并行计算：自注意力可以并行计算，这意味着可以有效地在现代硬件上进行加速。相比于RNN和CNN等序列模型，它更容易在GPU和TPU等硬件上进行高效的训练和推理。（因为在自注意力中可以并行的计算得分）

长距离依赖捕捉：传统的循环神经网络（RNN）在处理长序列时可能面临梯度消失或梯度爆炸的问题。自注意力可以更好地处理长距离依赖关系，因为它不需要按顺序处理输入序列。

自注意力的结构如下所示：

Scaled Dot-Product Attention



Self-Attention 结构

CSDN @与知冷暖★

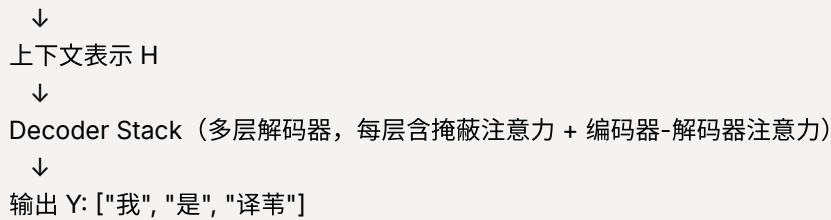
Transformer

Transformer 由两部分组成：

- **Encoder (编码器)**：输入序列 → 高维表示
- **Decoder (解码器)**：高维表示 + 输出序列前缀 → 目标序列

每部分又由多个结构相同的 **层 (layer)** 堆叠而成。通常有 6 层，但这个数量是可以调整的。

源句子 X: ["I", "am", "Yiwei"]
↓
Embedding + Positional Encoding (位置编码)
↓
Encoder Stack (多层次编码器，每层含自注意力机制)



1. 输入表示 (Embedding + Positional Encoding)

由于 Transformer 没有序列顺序的归纳偏置（不像 RNN 那样自然是顺序结构），所以必须手动引入位置信息：

- **Token Embedding**：将词映射为向量
- **Positional Encoding**：位置编码（固定正弦/余弦 或 学习的位置向量）

最终输入是两者相加：

$$X_{input} = \text{Embedding}(x_i) + \text{PositionalEncoding}(i)$$

2. Multi-Head Self-Attention (多头自注意力)

单头注意力机制核心计算：

对于每个输入向量 x , 生成三个向量：

- $Q = W^Q x$ (查询)
- $K = W^K x$ (键)
- $V = W^V x$ (值)

然后进行加权求和：

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

多头机制：

多个不同的 W^Q, W^K, W^V 权重头并行处理，然后拼接后再线性映射回原维度。

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

3. Feed Forward Network (前馈全连接层)

每个位置独立地通过两个线性层和激活函数：

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

4. 残差连接与 Layer Normalization (层归一化)

每个子层后都有：

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

使模型更容易训练，防止梯度消失。

Encoder 和 Decoder 的结构差异

Encoder 每层结构：

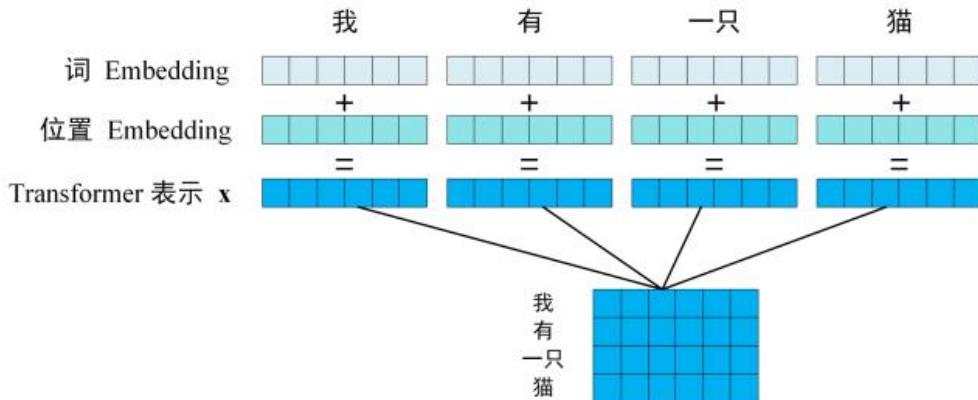
- 多头自注意力
- 前馈网络

Decoder 每层结构：

- 掩蔽多头自注意力 (**Masked Attention**)：防止当前时刻看到未来信息
- 编码器-解码器注意力：Decoder 查询 Encoder 的输出
- 前馈网络



第一步：获取输入句子的每一个单词的表示向量 \mathbf{X} , \mathbf{X} 由单词的 Embedding (Embedding就是从原始数据提取出来的Feature) 和单词位置的 Embedding 相加得到。



单词 Embedding

单词的 Embedding 有很多种方式可以获取，例如可以采用 Word2Vec、Glove 等算法预训练得到，也可以在 Transformer 中训练得到。

位置 Embedding

Transformer 中除了单词的 Embedding，还需要使用位置 Embedding 表示单词出现在句子中的位置。因为 Transformer 不采用 RNN 的结构，而是使用全局信息，不能利用单词的顺序信息，而这部分信息对于 NLP 来说非常重要。所以 Transformer 中使用位置 Embedding 保存单词在序列中的相对或绝对位置。

位置 Embedding 用 \mathbf{PE} 表示， \mathbf{PE} 的维度与单词 Embedding 是一样的。 \mathbf{PE} 可以通过训练得到，也可以使用某种公式计算得到。在 Transformer 中采用了后者，计算公式如下：

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

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

其中， pos 表示单词在句子中的位置， d 表示 \mathbf{PE} 的维度 (与词 Embedding 一样)， $2i$ 表示偶数的维度， $2i+1$ 表示奇数维度 (即 $2i \leq d$, $2i+1 \leq d$)。使用这种公式计算 \mathbf{PE} 有以下的好处：

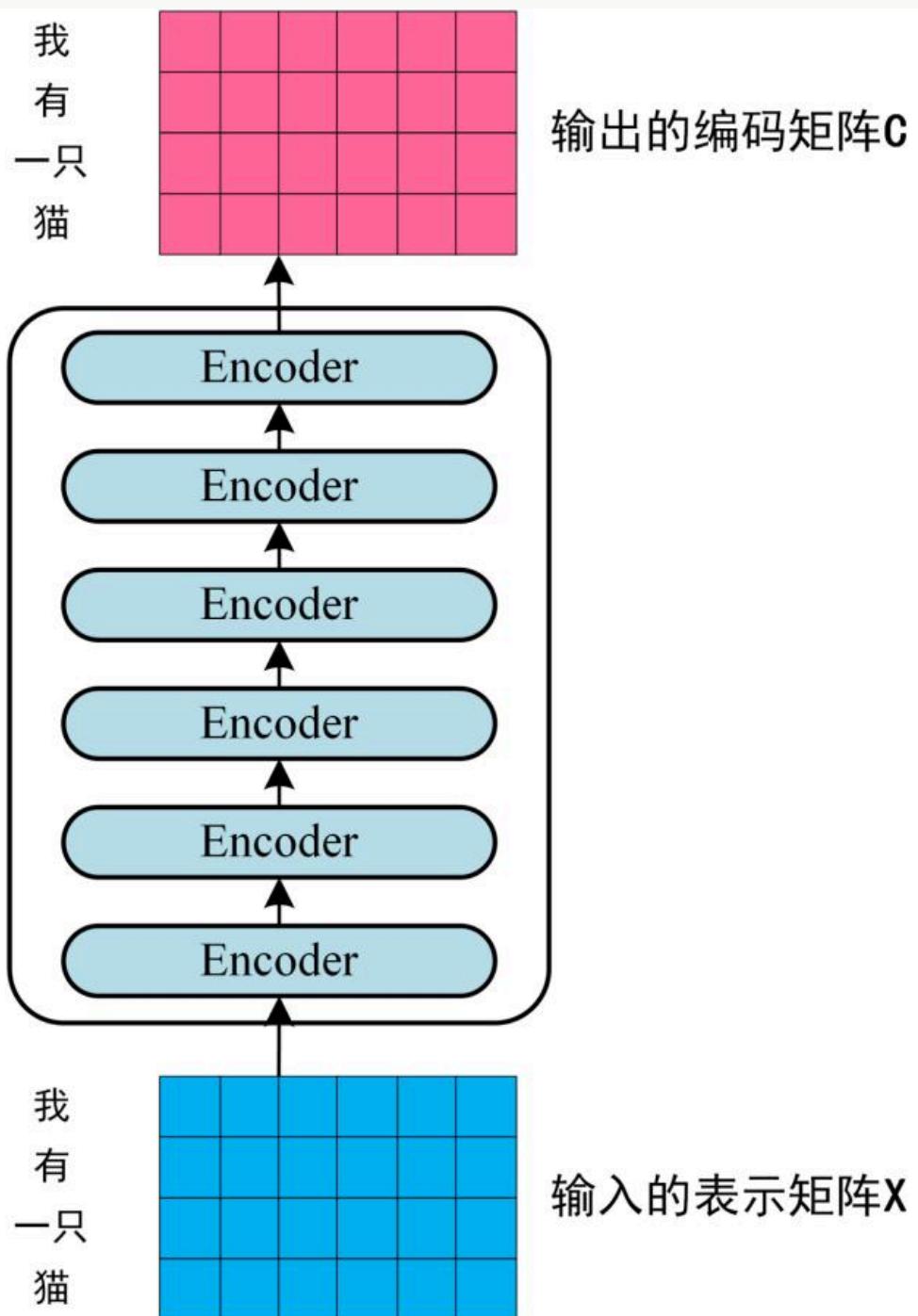
- 使 \mathbf{PE} 能够适应比训练集里面所有句子更长的句子，假设训练集里面最长的句子是有 20 个单词，突然来了一个长度为 21 的句子，则使用公式计算的方法可以计算出第 21 位的 Embedding。
- 可以让模型容易地计算出相对位置，对于固定长度的间距 k ， $\mathbf{PE}(pos+k)$ 可以用 $\mathbf{PE}(pos)$ 计算得到。因为 $\sin(A+B) = \sin(A)\cos(B) + \cos(A)\sin(B)$, $\cos(A+B) = \cos(A)\cos(B) - \sin(A)\sin(B)$ 。

将单词的词 Embedding 和位置 Embedding 相加，就可以得到单词的表示向量 \mathbf{x} ， \mathbf{x} 就是 Transformer 的输入。

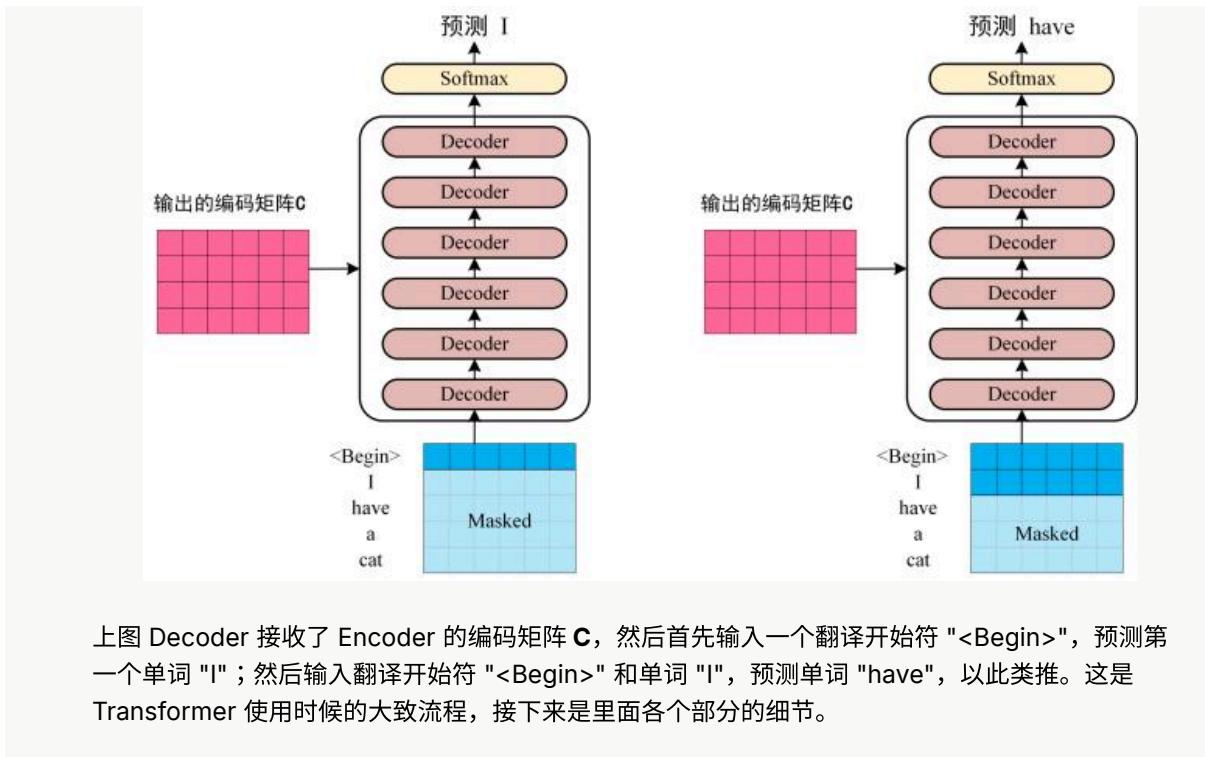
第二步：

将得到的单词表示向量矩阵 (如上图所示，每一行是一个单词的表示 \mathbf{x}) 传入 Encoder 中，经过 6 个 Encoder block 后可以得到句子所有单词的编码信息矩阵 \mathbf{C} ，如下图。单词向量矩阵用 $X \times d$ 表示，

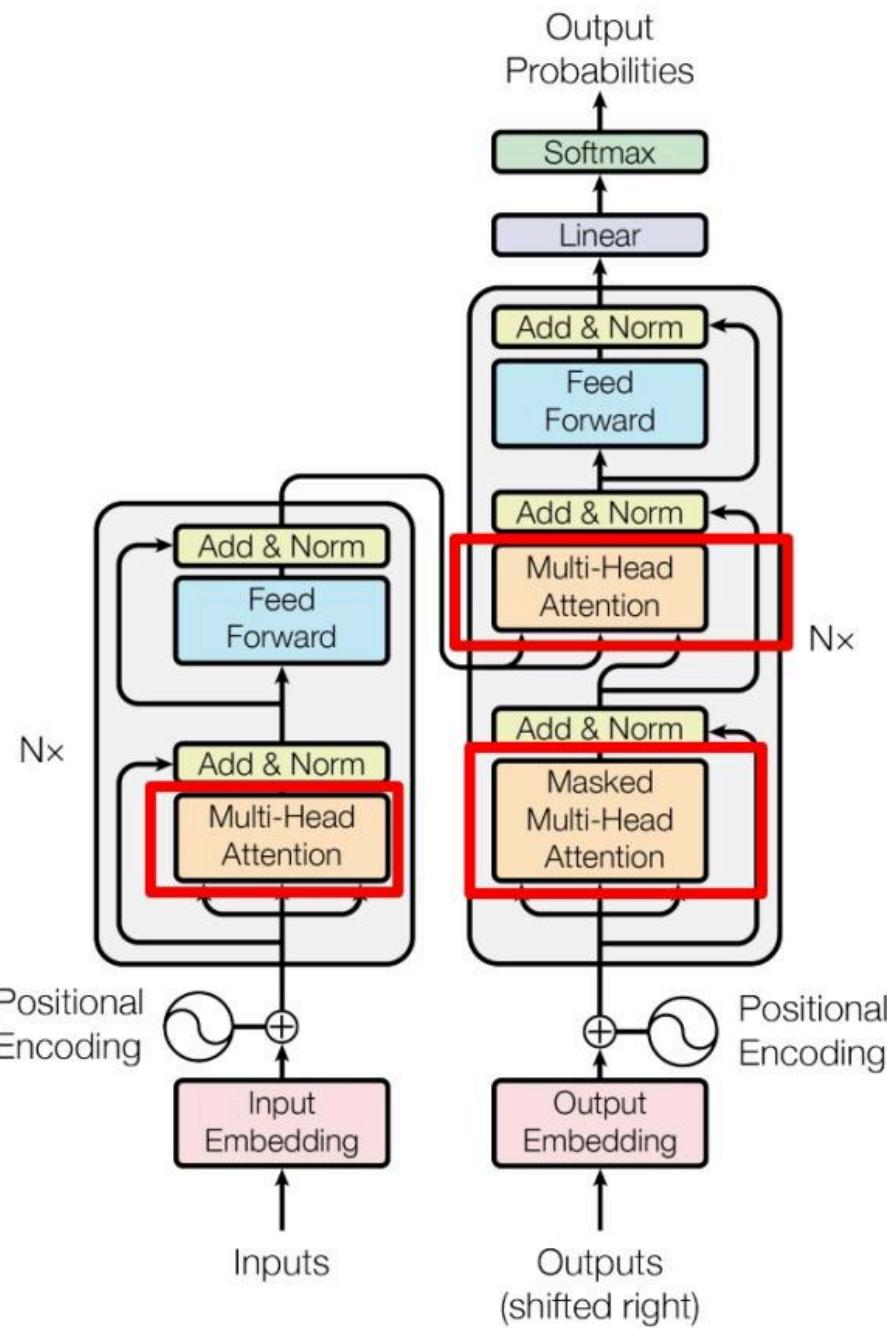
n 是句子中单词个数, d 是表示向量的维度 (论文中 $d=512$)。每一个 Encoder block 输出的矩阵维度与输入完全一致。



第三步：将 Encoder 输出的编码信息矩阵 C 传递到 Decoder 中，Decoder 依次会根据当前翻译过的单词 $1 \sim i$ 翻译下一个单词 $i+1$ ，如下图所示。在使用的过程中，翻译到单词 $i+1$ 的时候需要通过 **Mask (掩盖)** 操作遮盖住 $i+1$ 之后的单词。

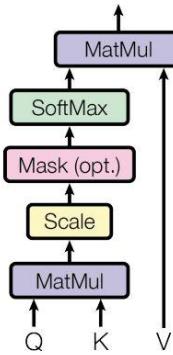


Self-Attention (自注意力机制)



左侧为 Encoder block，右侧为 Decoder block。红色圈中的部分为 **Multi-Head Attention**，是由多个 **Self-Attention** 组成的，可以看到 Encoder block 包含一个 Multi-Head Attention，而 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。

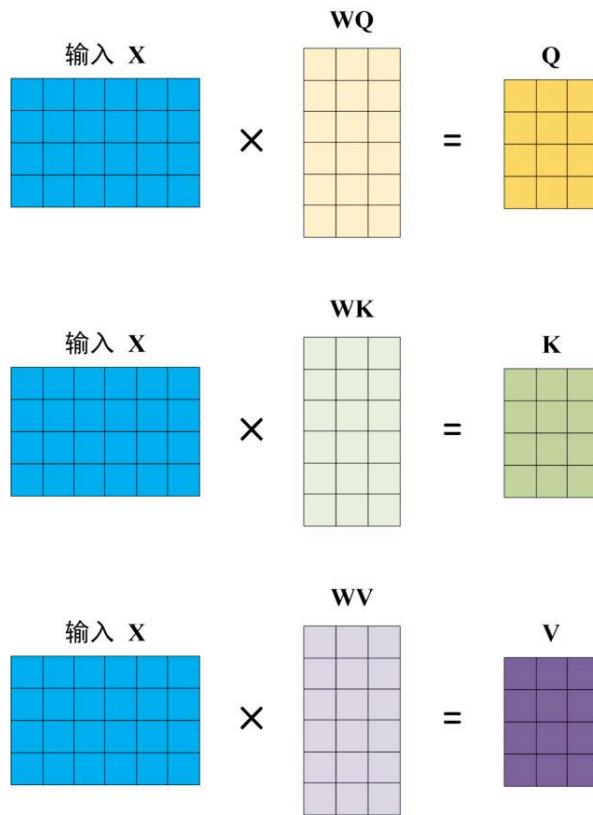
Scaled Dot-Product Attention



上图是 Self-Attention 的结构，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。

Q, K, V 的计算

Self-Attention 的输入用矩阵X进行表示，则可以使用线性变阵矩阵**WQ, WK, WV**计算得到**Q, K, V**。计算如下图所示，注意 **X, Q, K, V** 的每一行都表示一个单词。



Self-Attention 的输出

得到矩阵 Q, K, V之后就可以计算出 Self-Attention 的输出了，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，即向量维度

公式中计算矩阵 \mathbf{Q} 和 \mathbf{K} 每一行向量的内积，为了防止内积过大，因此除以 d_k 的平方根。 \mathbf{Q} 乘以 \mathbf{K}^T 的转置后，得到的矩阵行列数都为 n ， n 为句子单词数，这个矩阵可以表示单词之间的 attention 强度。下图为 \mathbf{Q} 乘以 \mathbf{K}^T ，1234 表示的是句子中的单词。

Q				K ^T				QK ^T			
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4

得到 QK^T 之后，使用 Softmax 计算每一个单词对于其他单词的 attention 系数，公式中的 Softmax 是对矩阵的每一行进行 Softmax，即每一行的和都变为 1.

QK ^T				Softmax				Output			
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4

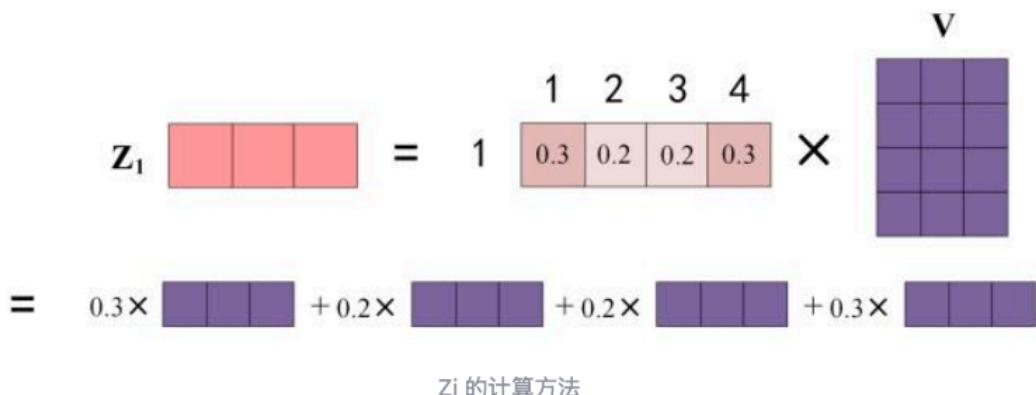
对矩阵的每一行进行 Softmax

得到 Softmax 矩阵之后可以和 V 相乘，得到最终的输出 Z 。

Softmax				V				Z			
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4	1	2	3	4

Self-Attention 输出

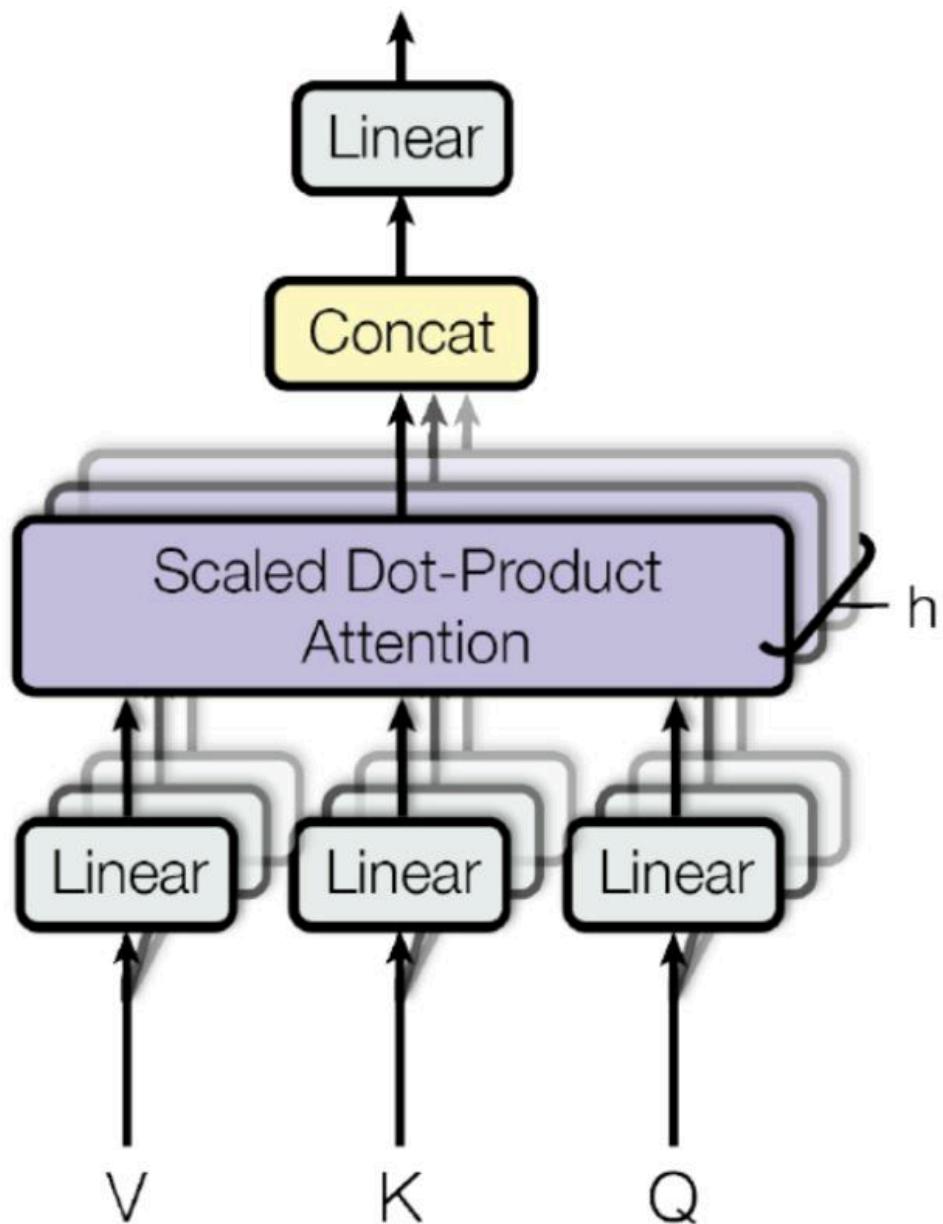
上图中 Softmax 矩阵的第 1 行表示单词 1 与其他所有单词的 attention 系数，最终单词 1 的输出 Z_1 等于所有单词 i 的值 V_i 根据 attention 系数的比例加在一起得到，如下图所示：



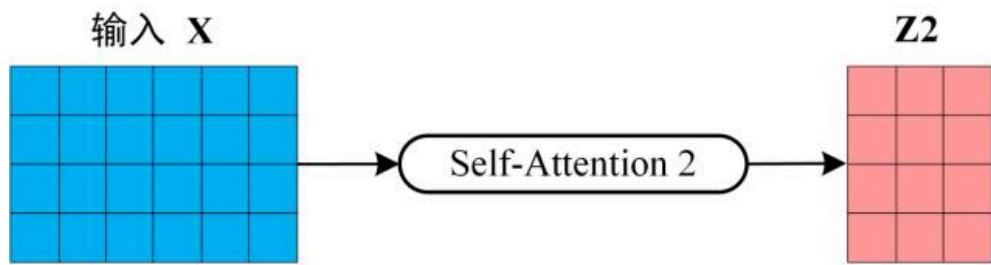
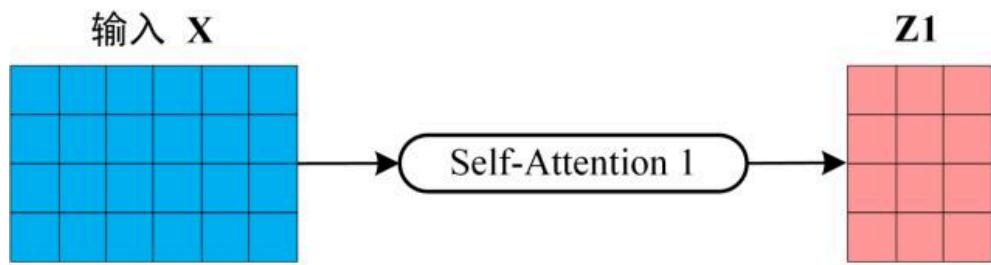
Multi-Head Attention

在上一步，我们已经知道怎么通过 Self-Attention 计算得到输出矩阵 Z ，而 Multi-Head Attention 是由多个 Self-Attention 组合形成的，下图是论文中 Multi-Head Attention 的结构图。

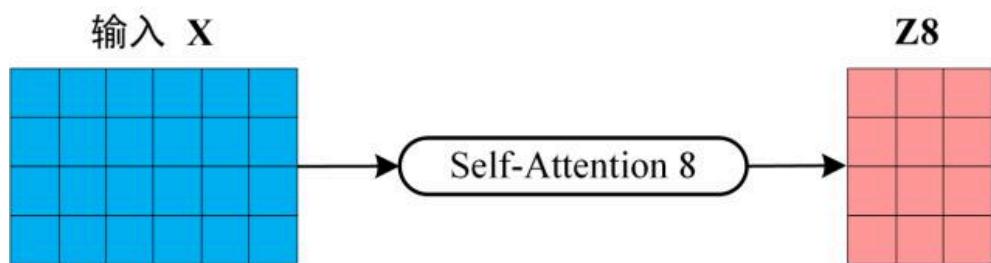
Multi-Head Attention



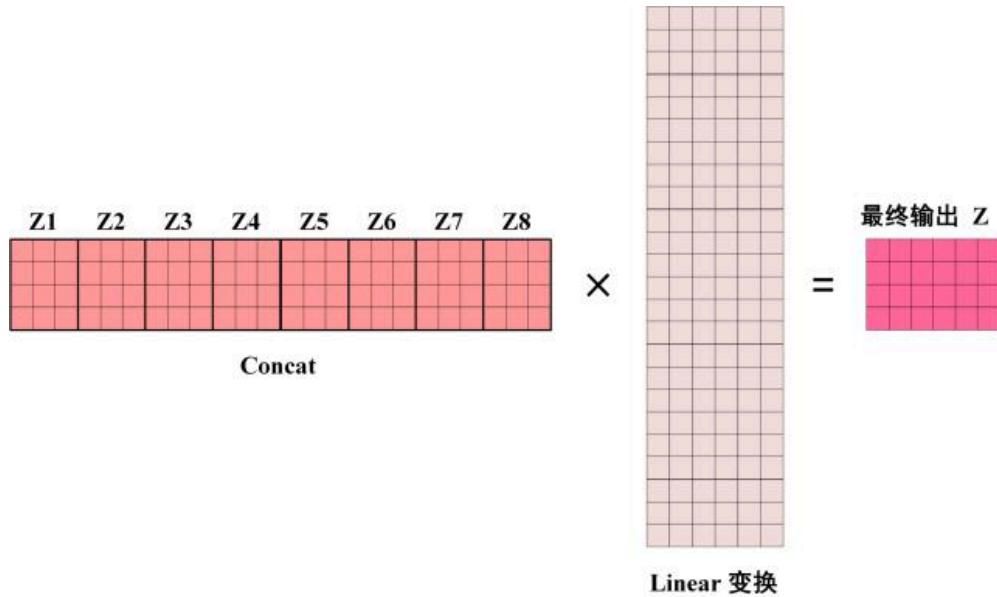
从上图可以看到 Multi-Head Attention 包含多个 Self-Attention 层，首先将输入 X 分别传递到 h 个不同的 Self-Attention 中，计算得到 h 个输出矩阵 Z 。下图是 $h=8$ 时候的情况，此时会得到 8 个输出矩阵 Z 。



• • • • •



得到 8 个输出矩阵 Z_1 到 Z_8 之后，Multi-Head Attention 将它们拼接在一起(**Concat**)，然后传入一个**Linear** 层，得到 Multi-Head Attention 最终的输出 Z 。



可以看到 Multi-Head Attention 输出的矩阵 Z 与其输入的矩阵 X 的维度是一样的。

Encoder 结构

由 Multi-Head Attention, **Add & Norm**, **Feed Forward**, **Add & Norm** 组成的。刚刚已经了解了 Multi-Head Attention 的计算过程，现在了解一下 Add & Norm 和 Feed Forward 部分。

Add & Norm

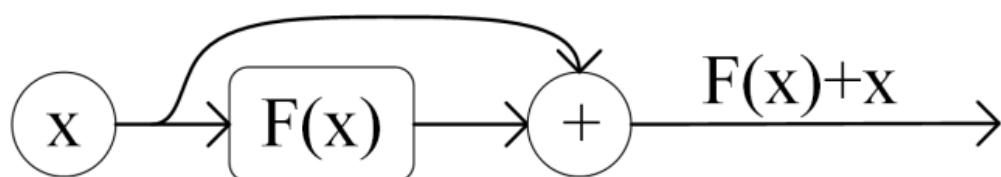
Add & Norm 层由 Add 和 Norm 两部分组成，其计算公式如下：

$$\text{LayerNorm}(X + \text{MultiHeadAttention}(X))$$

$$\text{LayerNorm}(X + \text{FeedForward}(X))$$

其中 X 表示 Multi-Head Attention 或者 Feed Forward 的输入， $\text{MultiHeadAttention}(X)$ 和 $\text{FeedForward}(X)$ 表示输出（输出与输入 X 维度是一样的，所以可以相加）。

Add 指 $X + \text{MultiHeadAttention}(X)$ ，是一种残差连接，通常用于解决多层网络训练的问题，可以让网络只关注当前差异的部分，在 ResNet 中经常用到：



Feed Forward

Feed Forward 层比较简单，是一个两层的全连接层，第一层的激活函数为 Relu，第二层不使用激活函数，对应的公式如下。

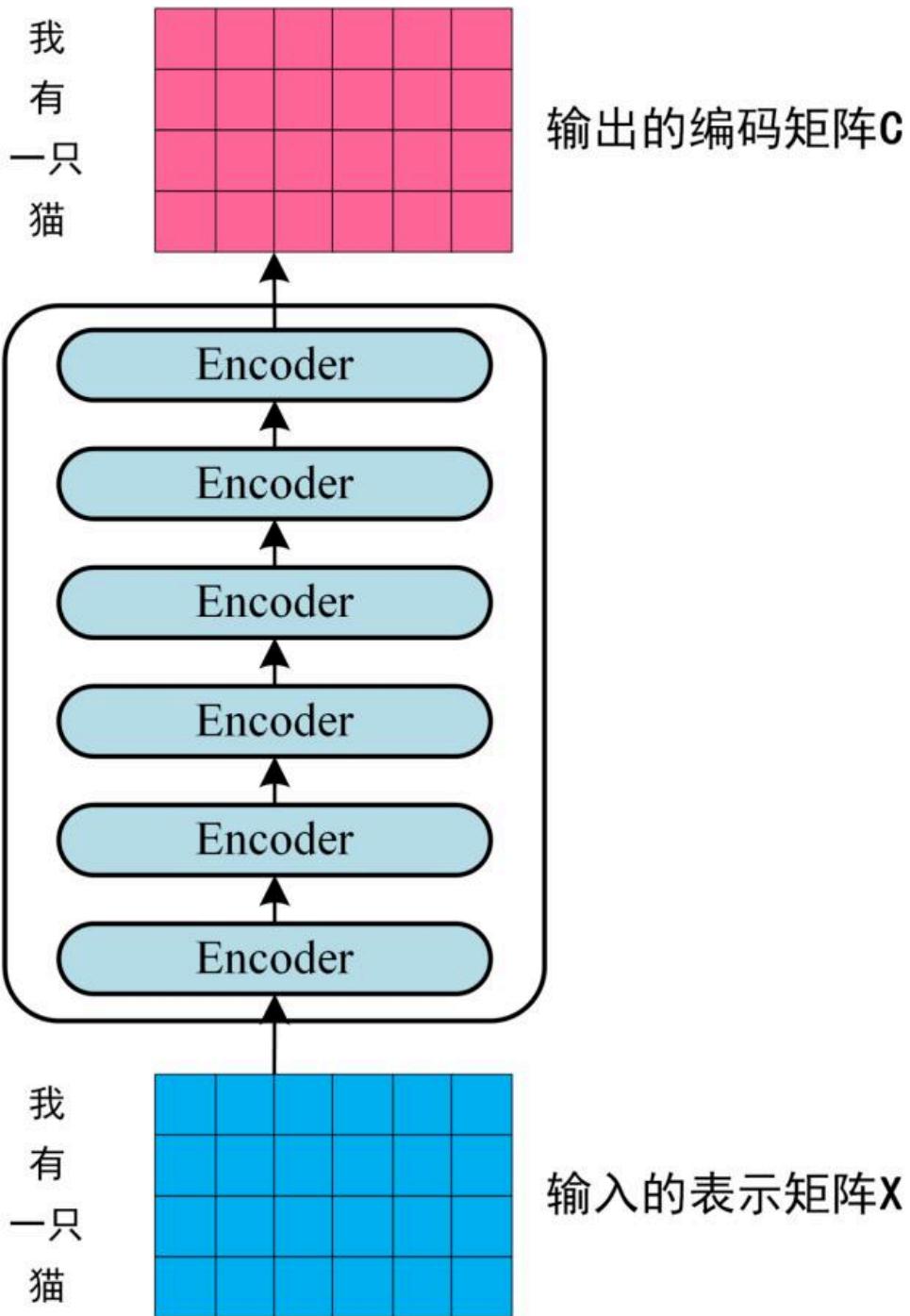
$$\max(0, XW_1 + b_1)W_2 + b_2$$

X是输入，Feed Forward 最终得到的输出矩阵的维度与**X**一致。

组成 Encoder

通过上面描述的 Multi-Head Attention, Feed Forward, Add & Norm 就可以构造出一个 Encoder block，Encoder block 接收输入矩阵 $X(n \times d)$ ，并输出一个矩阵 $O(n \times d)$ 。通过多个 Encoder block 叠加就可以组成 Encoder。

第一个 Encoder block 的输入为句子单词的表示向量矩阵，后续 Encoder block 的输入是前一个 Encoder block 的输出，最后一个 Encoder block 输出的矩阵就是**编码信息矩阵 C**，这一矩阵后续会用到 Decoder 中。



Decoder 结构

与 Encoder block 相似，但是存在一些区别：

- 包含两个 Multi-Head Attention 层。
- 第一个 Multi-Head Attention 层采用了 Masked 操作。
- 第二个 Multi-Head Attention 层的 \mathbf{K} , \mathbf{V} 矩阵使用 Encoder 的编码信息矩阵 \mathbf{C} 进行计算，而 \mathbf{Q} 使用上一个 Decoder block 的输出计算。
- 最后有一个 Softmax 层计算下一个翻译单词的概率。

第一个 Multi-Head Attention

Decoder block 的第一个 Multi-Head Attention 采用了 Masked 操作，因为在翻译的过程中是顺序翻译的，即翻译完第 i 个单词，才可以翻译第 $i+1$ 个单词。通过 Masked 操作可以防止第 i 个单词知道 $i+1$ 个单词之后的信息。下面以 "我有一只猫" 翻译成 "I have a cat" 为例，了解一下 Masked 操作。

下面的描述中使用了类似 Teacher Forcing 的概念，不熟悉 Teacher Forcing 的童鞋可以参考以下上一篇文章 Seq2Seq 模型详解。在 Decoder 的时候，是需要根据之前的翻译，求解当前最有可能的翻译，如下图所示。首先根据输入 "<Begin>" 预测出第一个单词为 "I"，然后根据输入 "<Begin> I" 预测下一个单词 "have"。

Decoder 可以在训练的过程中使用 Teacher Forcing 并且并行化训练，即将正确的单词序列 (<Begin> I have a cat) 和对应输出 (I have a cat <end>) 传递到 Decoder。那么在预测第 i 个输出时，就要将第 $i+1$ 之后的单词掩盖住，注意 Mask 操作是在 Self-Attention 的 Softmax 之前使用的，下面用 0 1 2 3 4 分别表示 "<Begin> I have a cat <end>"。

第一步：是 Decoder 的输入矩阵和 Mask 矩阵，输入矩阵包含 "<Begin> I have a cat" (0, 1, 2, 3, 4) 五个单词的表示向量，Mask 是一个 5×5 的矩阵。在 Mask 可以发现单词 0 只能使用单词 0 的信息，而单词 1 可以使用单词 0, 1 的信息，即只能使用之前的信息。

第二步：接下来的操作和之前的 Self-Attention 一样，通过输入矩阵 \mathbf{X} 计算得到 $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ 矩阵。然后计算 \mathbf{Q} 和 \mathbf{K}^T 的乘积 \mathbf{QK}^T 。

$$\begin{array}{c}
 \mathbf{Q} \\
 \begin{array}{|c|c|c|c|c|} \hline 0 & & & & \\ \hline 1 & & & & \\ \hline 2 & & & & \\ \hline 3 & & & & \\ \hline 4 & & & & \\ \hline \end{array}
 \end{array}
 \times
 \begin{array}{c}
 \mathbf{K}^T \\
 \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 \\ \hline \end{array}
 \end{array}
 =
 \begin{array}{c}
 \mathbf{QK}^T \\
 \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 \\ \hline 0 & & & & \\ \hline 1 & & & & \\ \hline 2 & & & & \\ \hline 3 & & & & \\ \hline 4 & & & & \\ \hline \end{array}
 \end{array}$$

第三步：在得到 \mathbf{QK}^T 之后需要进行 Softmax，计算 attention score，我们在 Softmax 之前需要使用 Mask 矩阵遮挡住每一个单词之后的信息，遮挡操作如下：

$$\begin{array}{c}
 \mathbf{QK}^T \\
 \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 \\ \hline 0 & & & & \\ \hline 1 & & & & \\ \hline 2 & & & & \\ \hline 3 & & & & \\ \hline 4 & & & & \\ \hline \end{array}
 \end{array}
 \otimes
 \begin{array}{c}
 \text{按位相乘} \\
 \text{Mask 矩阵} \\
 \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \text{green} & \text{yellow} & \text{yellow} & \text{yellow} \\ \hline 1 & \text{green} & \text{yellow} & \text{yellow} & \text{yellow} \\ \hline 2 & \text{green} & \text{yellow} & \text{yellow} & \text{yellow} \\ \hline 3 & \text{green} & \text{yellow} & \text{yellow} & \text{yellow} \\ \hline 4 & \text{green} & \text{yellow} & \text{yellow} & \text{yellow} \\ \hline \end{array}
 \end{array}
 =
 \begin{array}{c}
 \text{Mask } \mathbf{QK}^T \\
 \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \text{red} & & & \\ \hline 1 & \text{red} & \text{black} & & \\ \hline 2 & \text{red} & \text{black} & \text{black} & \\ \hline 3 & \text{red} & \text{black} & \text{black} & \text{black} \\ \hline 4 & \text{red} & \text{black} & \text{black} & \text{black} \\ \hline \end{array}
 \end{array}$$

得到 Mask \mathbf{QK}^T 之后在 Mask \mathbf{QK}^T 上进行 Softmax，每一行的和都为 1。但是单词 0 在单词 1, 2, 3, 4 上的 attention score 都为 0。

第四步：使用 Mask \mathbf{QK}^T 与矩阵 \mathbf{V} 相乘，得到输出 \mathbf{Z} ，则单词 1 的输出向量 Z_1 是只包含单词 1 信息的。

$$\begin{array}{c}
 \begin{matrix} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \text{red} & & & & \\ 1 & \text{red} & \text{red} & & & \\ 2 & \text{red} & \text{red} & \text{red} & & \\ 3 & \text{red} & \text{red} & \text{red} & \text{red} & \\ 4 & \text{red} & \text{red} & \text{red} & \text{red} & \text{red} \end{matrix} \times \begin{matrix} \mathbf{V} \\ \vdots \end{matrix} = \begin{matrix} \mathbf{Z} \\ \vdots \end{matrix} \\
 \text{Mask } \mathbf{QK}^T
 \end{array}$$

第五步：通过上述步骤就可以得到一个 Mask Self-Attention 的输出矩阵 Z_i ，然后和 Encoder 类似，通过 Multi-Head Attention 拼接多个输出 Z_i 然后计算得到第一个 Multi-Head Attention 的输出 Z ， Z 与输入 X 维度一样。

第二个 Multi-Head Attention

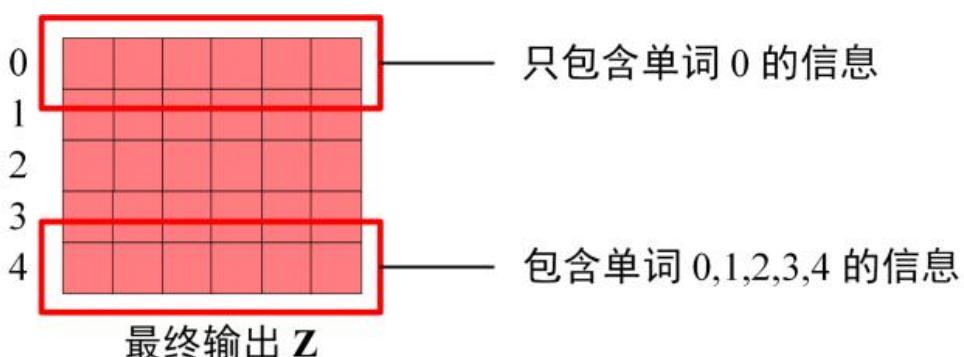
Decoder block 第二个 Multi-Head Attention 变化不大，主要的区别在于其中 Self-Attention 的 \mathbf{K} , \mathbf{V} 矩阵不是使用上一个 Decoder block 的输出计算的，而是使用 **Encoder** 的编码信息矩阵 \mathbf{C} 计算的。

根据 Encoder 的输出 \mathbf{C} 计算得到 \mathbf{K} , \mathbf{V} ，根据上一个 Decoder block 的输出 Z 计算 \mathbf{Q} (如果是第一个 Decoder block 则使用输入矩阵 \mathbf{X} 进行计算)，后续的计算方法与之前描述的一致。

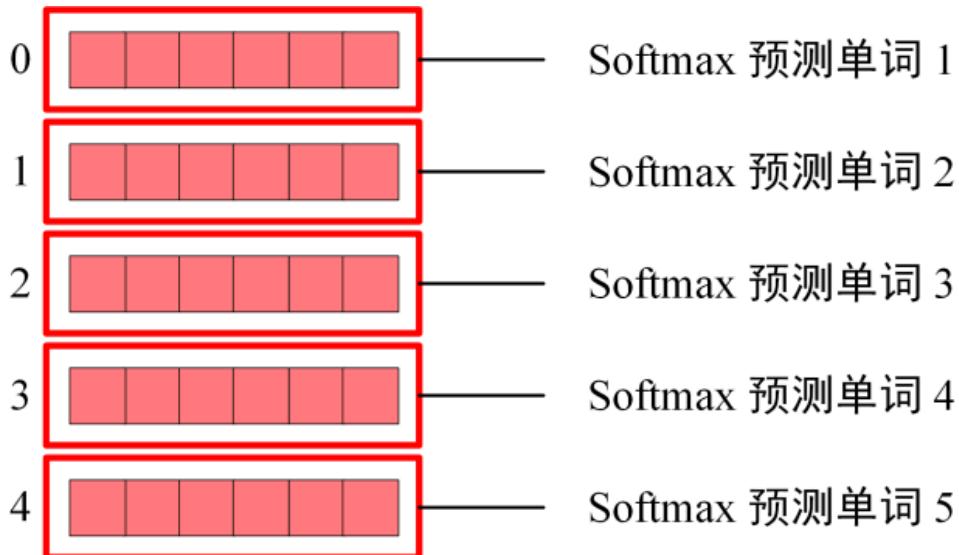
这样做的好处是在 Decoder 的时候，每一位单词都可以利用到 Encoder 所有单词的信息 (这些信息无需 **Mask**)。

Softmax 预测输出单词

Decoder block 最后的部分是利用 Softmax 预测下一个单词，在之前的网络层我们可以得到一个最终的输出 Z ，因为 Mask 的存在，使得单词 0 的输出 Z_0 只包含单词 0 的信息，如下：



Softmax 根据输出矩阵的每一行预测下一个单词：



这就是 Decoder block 的定义，与 Encoder 一样，Decoder 是由多个 Decoder block 组合而成。

Transformer 总结

- Transformer 与 RNN 不同，可以比较好地并行训练。
- Transformer 本身是不能利用单词的顺序信息的，因此需要在输入中添加位置 Embedding，否则 Transformer 就是一个词袋模型了。
- Transformer 的重点是 Self-Attention 结构，其中用到的 \mathbf{Q} , \mathbf{K} , \mathbf{V} 矩阵通过输出进行线性变换得到。
- Transformer 中 Multi-Head Attention 中有多个 Self-Attention，可以捕获单词之间多种维度上的相关系数 attention score。



如何理解自回归？

在序列建模中，**自回归 (autoregressive)** 是指模型在生成当前时刻 y_t 的输出时，只依赖它之前的输出 y_1, y_2, \dots, y_{t-1} ：

$$P(y_1, y_2, \dots, y_T) = \prod_{t=1}^T P(y_t | y_1, \dots, y_{t-1}, X)$$

其中：

- X 是输入序列（来自 Encoder）
- y_t 是第 t 个输出 token
- 这种机制可以逐步预测序列，比如语言模型就是典型的自回归模型。

🧠 二、Transformer 中的自回归实现机制

出现在 Transformer 的 Decoder 中：

Transformer Decoder 需要逐步生成一个输出序列，按顺序生成每一个 token，这就要求：

- 每个位置 不能访问未来的信息（如不能在生成第2个词时看第3个词）

为此，Decoder 使用了一个叫做：

🛡️ 掩蔽自注意力 (Masked Self-Attention)

它的做法是：

- 将未来的 token 信息用一个 **mask矩阵** 屏蔽掉，让模型只能“看到”当前及之前的 token。

- **符合自然语言生成顺序**：人写一句话时也是一词一词地写。
- **避免“作弊”**：模型生成时不能提前知道未来词。
- **可用于推理/文本生成**：模型可以一个词一个词地输出结果，用于对话、翻译、摘要等任务。

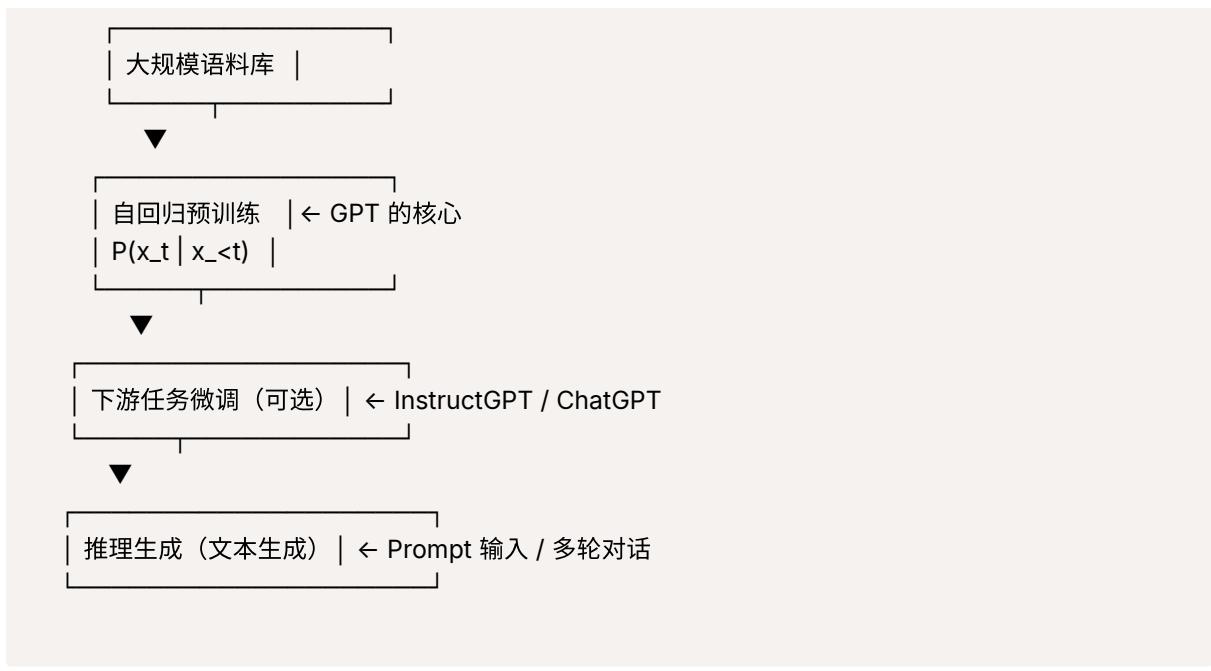
GPT系列模型

GPT 系列算法的流程可以整体看作由三大核心阶段组成：**预训练 → (选择性) 微调 → 推理 (生成)**。不同版本 (GPT-1、2、3、4) 在细节上有所升级，但总体流程一致。下面我会以系统化的方式，介绍 **GPT 系列算法的完整流程**，并指出各版本的演进关键。

➡️ 总流程图解 (简述)

cpp

复制编辑



1 预训练阶段 (Pretraining)

这是 GPT 系列最关键的部分。

输入数据：

- 大规模通用语料（如网页、书籍、维基百科、代码等）
- 形式为纯文本，无标签

训练目标：

最大化自回归概率：

$$\mathcal{L} = - \sum_{t=1}^T \log P(x_t | x_1, \dots, x_{t-1})$$

模型要学会：

- 上下文建模（语言结构）
- 世界常识
- 基本推理解能力

使用的结构：

- Transformer Decoder 堆叠结构
- 使用 Masked Multi-head Self-Attention 屏蔽未来信息

2 微调阶段 (Fine-tuning, 部分版本有)

GPT-1：有微调，任务特定微调

GPT-2：强调 zero-shot，不使用微调

GPT-3：通过 prompt engineering 做 in-context learning

ChatGPT/GPT-3.5/4：使用 RLHF 微调，核心升级！

🔧 方法一：监督微调（Supervised Fine-tuning）

- 用任务标注数据（如对话、问答）继续训练
- 让模型在特定任务上表现更强

🔧 方法二：RLHF（强化学习人类反馈）

- 引入人工打分系统，对输出好坏打分
- 用奖励建模 → PPO 优化模型行为（如 GPT-3.5 / ChatGPT）

⛳ 目的：

- 控制模型输出风格
- 提升对话能力、安全性、一致性

3 推理阶段（Inference）

这是我们日常使用 ChatGPT、GPT API 时所处的阶段。

INPUT 输入方式：

- 用户输入 Prompt，例如 "用一句话解释相对论"

MODEL 模型行为：

- 从 prompt 开始，自回归生成一个词一个词，直到生成终止符（或达到最大长度）
- 可以支持：
 - Zero-shot：直接问问题
 - Few-shot：加几个示例
 - Chain-of-thought：加思维链提示

PARAMETERS 可控参数：

- temperature：控制生成的多样性（高 = 更随机）
- top_k / top_p：控制采样策略
- max_tokens：限制生成长度

🌿 GPT 各版本演进简述

版本	关键特性	参数规模	微调策略
GPT-1	自回归 Transformer + 微调	117M	监督微调
GPT-2	强调 zero-shot / few-shot	1.5B	无微调，仅预训练
GPT-3	Few-shot prompt + 强大泛化能力	175B	无微调 + prompt 设计
InstructGPT	人类反馈强化学习（RLHF）	基于 GPT-3	有微调 + PPO + SFT

ChatGPT	对话优化的 GPT-3.5 + RLHF	数十亿~千亿级	多轮对话训练 + 安全策略
GPT-4	多模态 + 多语言 + 更强泛化能力	未公开	多阶段微调 + RLHF

📌 小结：GPT 系列流程三要点

阶段	做什么	技术关键词
预训练	自回归学习语言分布	Transformer-Decoder, MLM
微调	注入人类偏好或任务信息	SFT, RLHF, PPO
推理	按提示生成输出	Prompt Engineering, Sampling

BERT系列

BERT (Bidirectional Encoder Representations from Transformers) 系列是 NLP 领域划时代的预训练语言模型之一。它和 GPT 系列的最大区别在于：**BERT 使用了双向 Transformer Encoder 结构，而 GPT 是单向自回归的 Decoder 结构。**

下面我将详细介绍 BERT 系列模型的算法流程，包括基本架构、训练流程、关键任务、代表变种（如 RoBERTa、ALBERT、SpanBERT、ELECTRA）等。

🧠 一、BERT 系列算法流程概览

markdown

复制编辑

大规模语料库 |

▼

预训练（双向编码） | ← 关注上下文两侧信息

MLM + NSP任务 |

▼

微调（任务特定） | ← 如分类、问答、NER 等

▼

推理（预测、打分） | ← 用于实际NLP任务部署

🔍 二、BERT 预训练阶段详细流程

💡 使用结构：Transformer Encoder 堆叠（非 Decoder）

- 输入嵌入 = Token Embedding + Segment Embedding + Position Embedding
- 多层 Transformer Encoder，使用 双向 Self-Attention，即每个 token 能看“左右两边的上下文”

预训练任务

1. Masked Language Model (MLM) 核心任务

- 随机 mask 掉 15% 的词，让模型预测它们
- 输入：`I have a [MASK] dog.` → 预测 `[MASK] = little`

$L_{MLM} = -\sum_{i \in \text{masked}} \log P(x_i | x_{\setminus i})$

$L_{MLM} = -\sum_{i \in \text{masked}} \log P(x_i | x_{\setminus i})$

|  注意：BERT 不是自回归的，它是“填空式”的语言模型

2. Next Sentence Prediction (NSP) (后来被弃用)

- 输入一对句子 A、B，判断 B 是否是 A 的下一个句子
- 目标：学习句间关系（如问答、对话）

三、微调阶段 (Fine-tuning)

- 下游任务上加一层简单的任务头（如分类器），全模型微调
- 通常在每个任务的特定输入格式上微调模型参数

▼ 应用示例

任务	输入格式	输出形式
文本分类	<code>[CLS] sentence [SEP]</code>	<code>[CLS]</code> 输出接 softmax
问答 (SQuAD)	<code>[CLS] question [SEP] context [SEP]</code>	起始 / 结束位置的概率
NER	<code>[CLS] sentence tokens... [SEP]</code>	每个 token 的分类标签

四、BERT 系列衍生模型发展

模型	改进点	代表特点
RoBERTa	去掉 NSP + 更大数据 + 更长训练	更强的性能和泛化能力
ALBERT	参数共享 + 矩阵分解	极大减少参数，速度更快
SpanBERT	预测 span 而非单个 token	提高对句法结构、关系抽取等任务的表现
ELECTRA	换掉 MLM，用替换检测任务	判别式训练，速度快，收敛好
DistilBERT	蒸馏版 BERT	体积小，速度快，仍保留大部分性能

GPT vs BERT 核心区别

维度	BERT	GPT
架构	Transformer Encoder (双向)	Transformer Decoder (单向)
训练方式	MLM (填空)	自回归 (从左到右生成)
输入长度控制	全局输入	基于 Prompt 扩展生成
微调策略	加任务头 + 全模型微调	Prompt Learning / RLHF
擅长任务	分类、抽取、问答	文本生成、写作、对话

◀ 总结

BERT 系列的训练流程本质是：

| 预训练双向语言理解能力 → 微调迁移到下游任务 → 推理部署

它奠定了“预训练-微调范式”的基础，为后续的 NLP 模型（如 T5、BART、DeBERTa）奠定了技术路线，也与 GPT 系列构成了理解 vs 生成的经典对比。



ELMo、BERT 和 GPT 系列模型都属于 **大规模语言模型**，但它们在 **结构设计、预训练目标、语义建模方式和下游任务迁移策略** 上存在显著差异。以下是一个系统性的对比分析：

⌚ 一览表：ELMo、BERT、GPT 系列模型差异对比

模型	模型架构	语境方向	预训练目标	使用方式	代表任务优势
ELMo	BiLSTM + Word Emb.	上下文双向	Language Modeling (左右分别训练)	Embedding特征提取	依赖句法任务 (如NER)
BERT	Transformer Encoder	双向	MLM + NSP	全模型微调	分类、问答、抽取等理解任务
GPT-1/2/3/4	Transformer Decoder	单向 (左→右)	自回归语言建模	Prompt Learning 或微调	生成任务 (写作、代码、对话)

🧠 二、详细结构与预训练目标对比

✓ ELMo (Embeddings from Language Models)

- **结构**：使用两个独立训练的 LSTM 语言模型（从左到右、从右到左），输出一个词的上下文嵌入
- **特点**：
 - 上下文相关的嵌入 (contextual word embedding)
 - 冻结的特征 (不微调整整个模型)
 - 使用在传统 NLP 模型之上 (比如 CRF)
- **局限性**：
 - 没有使用 Transformer，不支持大规模并行
 - 左右 LSTM 分别训练，语义融合有限

✓ BERT (Bidirectional Encoder Representations from Transformers)

- **结构**：纯 Transformer Encoder，堆叠多层
- **预训练任务**：
 - MLM：随机 mask 掉词，让模型预测它
 - NSP：判断两个句子是否连续 (RoBERTa 后废弃)
- **优点**：
 - 真正的双向语言建模
 - 支持句对任务
 - 可迁移到多种任务 (分类、问答、NER 等)
- **缺点**：
 - 不能直接用于生成 (不具备自回归能力)
 - MLM 不匹配自然语言生成任务

✓ GPT 系列 (Generative Pretrained Transformer)

- **结构**：Transformer Decoder（只看过去，不看未来）
- **预训练任务**：标准自回归语言模型（Next Word Prediction）
- **特点**：
 - 擅长文本生成、代码、写作、对话等任务
 - 训练时不需要“mask”操作，自然语言建模能力强
 - 随着模型规模扩大（GPT-2 → GPT-3 → GPT-4），涌现出通用智能能力
- **缺点**：
 - 单向注意力，句子理解可能受限（无全局双向语境）
 - 参数量大，训练和部署成本高

🔧 三、迁移学习策略差异

模型	预训练-微调策略	参数使用方式
ELMo	特征提取器	冻结参数，仅输出嵌入
BERT	全模型微调	微调全部权重
GPT	Prompt Learning（或微调）	支持少样本/零样本学习

🧩 四、代表性应用场景

模型类型	更适合任务	示例
ELMo	结构性NLP任务，如NER、POS标注	BioNER、Syntactic Parsing
BERT	理解类任务：问答、阅读理解、情感分析	SQuAD、MNLI、GLUE
GPT	生成类任务：写作、翻译、代码、对话	ChatGPT、Codex、StoryWriter

⬅ 总结：三者的本质差异

- **ELMo**：传统 RNN 架构，嵌入层面增强上下文信息，本质上是“动态词向量”。
- **BERT**：基于 Encoder 的双向上下文建模，适用于理解类任务。
- **GPT 系列**：基于 Decoder 的单向语言建模，核心优势是**生成能力 + 可扩展性**。

👉 一句话总结：

ELMo 是上下文嵌入器，BERT 是理解专家，GPT 是生成大师。

Swim Transformation

Swin Transformer (**Shifted Window Transformer**) 是微软亚洲研究院在 2021 年提出的一种 **视觉 Transformer** 架构，它解决了原始 ViT (Vision Transformer) 在处理高分辨率图像时存在的计算量大、缺乏层次性等问题，首次真正实现了 Transformer 在视觉任务中对标甚至超越 CNN 的能力。

一、Swin Transformer 概述

名称含义：

- **Swin = Shifted Window**，核心思想是使用滑动窗口对图像进行局部注意建模，从而降低计算复杂度，同时实现跨区域的信息交互。

二、架构核心思想

Swin Transformer 的核心由以下几个模块构成：

1. 分层结构 (Hierarchical)

- 类似 CNN：输入图像先分成固定尺寸的 patch（如 4×4 ），然后逐层构建更高语义层次的特征图。
- 每个 stage 会将特征图的尺寸减半，通道数加倍，形成类似 CNN 的金字塔结构 (Stage 1 → 4)。

2. 局部窗口注意力 (Window-based MSA)

- 不像 ViT 对整个图像做全局 self-attention，Swin 仅在 **非重叠小窗口内** 做 self-attention（例如 7×7 ）。
- 优势：计算复杂度从 $O((HW)^2)$ 降为 $O((M^2)(HW/M^2)) = O(HW)$ ，极大减少资源消耗。

3. 窗口平移 (Shifted Windows)

- 如果总在固定窗口做注意力，会限制区域感知。
- Swin 使用**交替平移窗口**，比如第二层将窗口偏移一半大小，从而实现跨窗口交互，信息可以在不同 patch 之间传播。

4. Patch Merging

- 类似 CNN 中的 pooling 操作，将相邻的 patch 合并，减少分辨率、增加通道数，逐层提取抽象特征。

三、流程结构

1. 输入划分 Patch (Patch Partitioning)

- 输入图像 → 分为 4×4 patch → 每个 patch 展平 → 映射为 token

2. Stage-wise Transformer Block (4个 Stage)

- 每个 Stage 包含多个 Swin Transformer Block：
 - 包括 Window MSA (局部 attention)
 - MLP 层
 - LayerNorm + 残差连接

3. Shifted Window 机制

- 在交错 block 中进行窗口平移，交叉捕获区域上下文

4. Patch Merging

- 每过一个 stage，合并相邻 patch，构成金字塔特征图（如 ResNet）

四、Swin Transformer 与 ViT 的对比

特性	ViT	Swin Transformer
----	-----	------------------

Attention 范围	全局	局部（窗口内）+ Shifted 跨窗口
计算复杂度	$\$O((HW)^2)$	$\$O(HW)$
特征结构	平铺	分层（pyramid）
下游任务适应性	主要用于分类	适合分类、检测、分割等多任务
迁移能力	中	强（适用于 COCO、ADE20K 等）

🏆 五、应用与性能

Swin Transformer 在多个视觉任务上都刷新了 SoTA (State-of-the-Art) :

- **ImageNet 图像分类**
- **COCO 目标检测** (配合 Mask R-CNN)
- **ADE20K 语义分割**
- 支持迁移到视频、医学图像、3D视觉、生成模型等场景

◀ 总结一句话 :

Swin Transformer 是将 Transformer 的全局建模能力与 CNN 的局部归纳偏好和多尺度能力融合的一种高效视觉主干结构，它开启了视觉 Transformer 在工业级图像处理中的广泛应用。

Transformer的主要应用

自然语言处理领域

🎯 一、机器翻译 (Machine Translation)

✍ 任务定义 :

给定源语言句子 $X = (x_1, x_2, \dots, x_n)$ ，预测目标语言句子 $Y = (y_1, y_2, \dots, y_m)$ 。

📘 算法流程 :

1. 编码器部分 (Encoder) :

将源句 \$X\$ 转化为上下文表示 \$Z\$:

- 将 \$X\$ 转为嵌入向量:

$$E_x = XW_e + P$$

\$W_e\$: 嵌入矩阵, \$P\$: 位置编码

- 多层 Transformer 编码器 (Layer Norm + MSA + FFN) :

$$Z = \text{Encoder}(E_x)$$

2. 解码器部分 (Decoder) :

逐步生成 \$y_1\$ 到 \$y_m\$, 每一步使用先前已生成的 token 和编码器输出:

- Decoder 输入:

$$E_y^{(t)} = Y_{<t}W_e + P$$

- 使用 masked multi-head attention 生成当前隐藏状态:

$$\hat{y}_t = \text{Decoder}(E_y^{(t)}, Z)$$

- 输出概率分布:

$$P(y_t|y_{<t}, X) = \text{Softmax}(W_o \cdot \hat{y}_t)$$

3. 损失函数 (Cross Entropy) :

$$\mathcal{L}_{\text{MT}} = - \sum_{t=1}^m \log P(y_t|y_{<t}, X)$$

二、自动文本摘要 (Text Summarization)

任务定义 :

给定原始文档 \$D = (d_1, d_2, \dots, d_n)\$, 生成总结 \$S = (s_1, s_2, \dots, s_m)\$。

算法流程 (类似机器翻译) :

1. 编码器输入：

$$E_d = DW_e + P$$

$$Z = \text{Encoder}(E_d)$$

2. 解码器生成摘要：

$$E_s^{(t)} = S_{<t}W_e + P$$

$$\hat{s}_t = \text{Decoder}(E_s^{(t)}, Z)$$

$$P(s_t|s_{<t}, D) = \text{Softmax}(W_o \cdot \hat{s}_t)$$

3. 损失函数：

$$\mathcal{L}_{\text{Summ}} = - \sum_{t=1}^m \log P(s_t|s_{<t}, D)$$

| 可选增强：指针生成网络 Pointer-Generator 可引入输入中的关键句。

三、机器阅读理解 (Machine Reading Comprehension)

📌 任务定义：

给定文章 \$C = (c_1, \dots, c_n)\$ 和问题 \$Q = (q_1, \dots, q_k)\$，输出答案 \$A\$ 是 \$C\$ 的一个 span。

📘 算法流程 (基于 BERT/Transformer)：

1. 输入拼接：

$$X = [[\text{CLS}], Q, [\text{SEP}], C, [\text{SEP}]]$$

2. Transformer 编码：

$$H = \text{Transformer}(X)$$

3. 起始位置预测：

$$P_{\text{start}} = \text{Softmax}(W_s H + b_s)$$

4. 终止位置预测：

$$P_{\text{end}} = \text{Softmax}(W_e H + b_e)$$

5. 损失函数 (两个交叉熵)：

$$\mathcal{L}_{\text{MRC}} = -\log P_{\text{start}}[i^*] - \log P_{\text{end}}[j^*]$$

其中 \$i^*, j^*\$ 是答案在上下文中的真实位置。

👉 总结表格

任务	输入	输出	核心机制	损失函数形式
机器翻译	\$X\$ (源语言)	\$Y\$ (目标语言)	编码器-解码器架构	$\sum_t \log P(y_t)$
文本摘要	\$D\$ (原始文档)	\$S\$ (摘要)	编码器-解码器架构 + 指针机制可选	$\sum_t \log P(s_t)$
阅读理解	\$Q, C\$ (问题+文章)	\$i, j\$ (答案起止位置)	Transformer 分类 span 位置	$-\log P[i^j] - \log P[j^i]$

计算机视觉领域

一、图像分类 (Image Classification) ——以 Vision Transformer (ViT) 为例

任务定义：

给定一张图像 $I \in \mathbb{R}^{H \times W \times C}$, 预测其类别 $y \in \{1, \dots, K\}$.

算法流程：

1. 图像切分为 Patch:

将图像划分为 N 个小 patch (例如 16×16) :

$$I \rightarrow \{x_1, x_2, \dots, x_N\}, \quad x_i \in \mathbb{R}^{P \times P \times C}$$

2. Patch + Position Embedding:

每个 patch 映射为固定维度的向量, 加上位置编码:

$$z_0^i = E x_i + p_i, \quad i = 1, \dots, N$$

同时引入 class token:

$$z_0^{\text{cls}} = E_{\text{cls}} + p_{\text{cls}}$$

整个输入为:

$$Z_0 = [z_0^{\text{cls}}, z_0^1, z_0^2, \dots, z_0^N]$$

3. Transformer 编码器处理:

多层 Transformer 层:

$$Z_l = \text{TransformerLayer}(Z_{l-1}), \quad l = 1, \dots, L$$

4. 分类头 (MLP) :

使用 class token 输出进行分类:

$$\hat{y} = \text{Softmax}(W \cdot z_L^{\text{cls}})$$

5. 损失函数 (交叉熵) :

$$\mathcal{L}_{\text{cls}} = - \sum_{k=1}^K y_k \log \hat{y}_k$$

🎯 二、目标检测 (Object Detection) ——以 DETR (DEtection TRansformer) 为例

📌 任务定义：

给定一张图像 \mathbf{I} ，输出目标框位置 (bounding box) 和类别。

📘 算法流程：

1. 图像特征提取：

使用 CNN 提取特征图：

$$F = \text{CNN}(\mathbf{I}) \in \mathbb{R}^{C \times h \times w}$$

将其展平为序列输入到 Transformer：

$$Z_0 = \text{Flatten}(F) + \text{PositionEncoding}$$

2. 添加可学习的 object query：

设有 N 个可学习的查询向量 (目标查询 token)：

$$Q = \{q_1, \dots, q_N\}, \quad q_i \in \mathbb{R}^d$$

3. Transformer 解码器：

利用 cross-attention 让每个 query 捕捉图像中一个目标：

$$H = \text{TransformerDecoder}(Q, Z_0)$$

4. 预测框与类别：

每个 query 输出一个目标预测：

- 类别 (含背景类)：

$$\hat{y}_i = \text{Softmax}(W_{\text{cls}} \cdot h_i)$$

- 边界框 (归一化中心 + 宽高)：

$$\hat{b}_i = \sigma(W_{\text{box}} \cdot h_i) \in [0, 1]^4$$

5. 匹配 (Hungarian Matching)：

用匈牙利算法将预测和 ground truth 匹配成一一对应关系。

6. 损失函数：

- 分类损失（交叉熵）：

$$\mathcal{L}_{\text{cls}} = - \sum_i y_i \log \hat{y}_i$$

- 边界框回归损失（L1 + GIoU）：

$$\mathcal{L}_{\text{box}} = \sum_i \left\| b_i - \hat{b}_i \right\|_1 + \text{GIoU}(b_i, \hat{b}_i)$$

- 总损失：

$$\mathcal{L}_{\text{DETR}} = \lambda_{\text{cls}} \mathcal{L}_{\text{cls}} + \lambda_{\text{box}} \mathcal{L}_{\text{box}}$$

总结对比

任务	模型	输入表示	输出表示	核心机制
图像分类	ViT	Patch + Pos Encoding	类别概率（使用 class token）	全图统一建模，全局信息强
目标检测	DETR	CNN 特征图 + Pos Encoding	N 个 object query 输出目标	Query attention 匹配目标



“端到端（End-to-End）”是一种**系统设计和训练方式**，其核心思想是：

从输入（原始数据）直接映射到最终输出（目标结果），中间过程由模型自动学习完成，而不依赖人为设计的中间特征或步骤。

◆ 举个例子：传统 vs 端到端 NLP

以**机器翻译**为例：

🌰 传统方式（非端到端）：

英文句子 → 分词 → 词性标注 → 句法解析 → 翻译模板匹配 → 中文句子

- 多个模块串联，每个模块单独设计和训练。
- 每一步错误会影响后续，叫“误差积累”。

⚙️ 端到端方式（Transformer）：

英文句子（原始） → Transformer 模型 → 直接输出中文句子

- 模型自动从数据中学习所有隐含规律。
- 不需要分词、模板、解析等人为步骤。
- 优点：统一建模，训练效率高，泛化性强。

◆ 常见端到端模型例子

应用	端到端模型	输入	输出
机器翻译	Transformer	原始源语言序列	目标语言序列
图像分类	Vision Transformer (ViT)	原始图像	类别标签
语音识别	DeepSpeech	原始语音波形	文本
目标检测	DETR	原始图像	边界框 + 类别
自动驾驶路径规划	RL-based / 模拟 DWA 变体	感知结果或图像	控制指令（角度/速度）

✓ 总结：端到端的特点

特征	描述
统一性	训练目标和推理过程是一个整体系统，端到端优化
自动特征学习	不需要手动设计中间特征，依赖深度神经网络自动学习
易于部署与扩展	模块较少，结构整洁，便于训练和部署
对数据量要求较高	需要大量标注数据来让模型学会“所有中间过程”
可解释性较低	中间过程不透明，分析模型行为较难