1. 注意力机制

注意力机制，attention机本质上是一种让模型"聚焦"于输入中最相关部分的方法。这有点像人类在阅读长文本时会关注重要的词句,而不是平等地对待每个单词。

我们用一个图书馆的例子来解释attention中的query、key和value：想象你走进一个巨大的图书馆,想找一本关于深度学习的书。

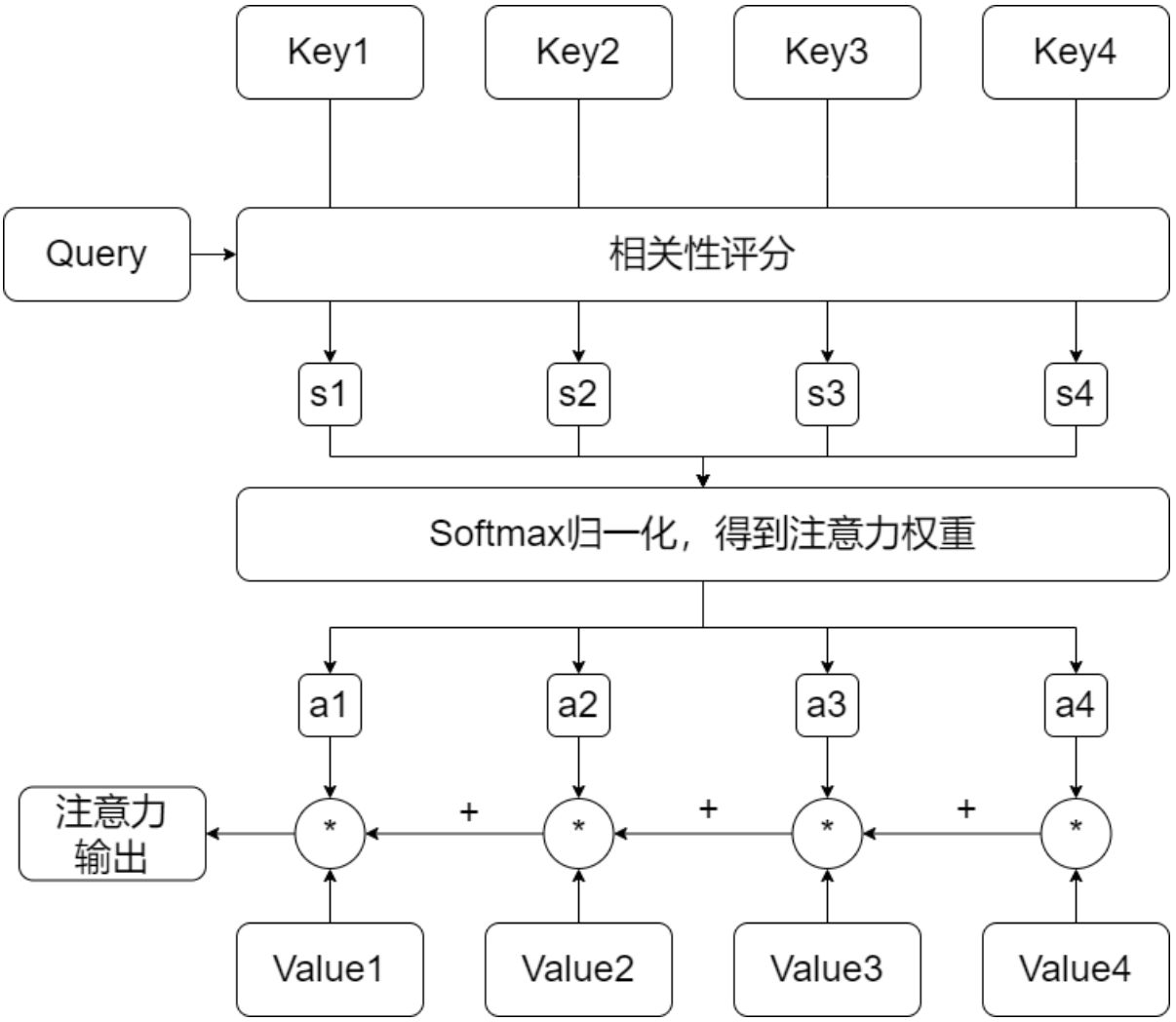
1. Query(查询):
   这就是你的需求或问题。在这个例子中,query是"我想要一本深度学习的书"。
2. Key(键):
   这相当于图书馆里每本书的标签或简短描述。比如"深度学习入门"、"计算机视觉应用"、"自然语言处理基础"等。这些key帮助你快速了解每本书的内容。
3. Value(值):
   这代表每本书的实际内容。也就是说,一旦你找到了合适的书(通过对比你的query和书的key),你就可以获取这本书的内容(value)。

attention的工作过程如下:

1. 你的query(寻找深度学习的书)会与图书馆中所有书的key进行比较。
2. 系统会计算query与每个key的相关度。与深度学习高度相关的书会得到更高的"注意力分数"。
3. 根据这些分数,系统会重点关注(即给予更多"注意力")最相关的几本书。
4. 最后,系统会根据注意力分数对这些相关书籍的内容(value)进行加权汇总,为你提供最相关的信息。

这个过程让模型能够从大量信息中筛选出最相关的部分,就像你在图书馆中快速定位到最合适的书籍一样。

现在我们初步了解注意力机制的原理，那么如何实现？从算法上理解，Attention机制其实就是对Source中的元素分配一系列权重系数，然后加权求和的过程。其框架可以抽象为下图部分：



实际上，注意力机制并不是Transformer所独有的，例如60年代提出来的Nadaraya-Watson核回归。

Nadaraya-Watson核回归是一种非参数回归方法，用于估计一个未知函数的值。它的基本思想是，通过对训练数据中的所有样本加权求和，来预测某个新的输入点的输出值。权重由核函数（例如高斯核）计算，核函数的作用是根据输入点和样本点之间的距离来分配权重。具体来说，给定一个输入点 x，Nadaraya-Watson估计器的输出为：

$$\hat{y}(x) = \frac{\sum_{i=1}^{n} K(x, x_i) y_i}{\sum_{i=1}^{n} K(x, x_i)}$$

其中，$K(x, x_i)$ 是核函数，y_i 是训练集中第 i 个样本的输出。

如果我们将Nadaraya-Watson核回归与注意力机制进行类比：

- **输入点 x 可以视为query。**
- **每个样本点 x_i 是key。**
- **核函数 $K(x, x_i)$ 计算的值对应于query和key之间的相似度。**
- **核函数值经过归一化（分母的作用）得到的权重，对应于注意力权重。**
- **最终的加权和 $\hat{y}(x)$ 就是注意力机制中的注意力输出。**

2. Transformer

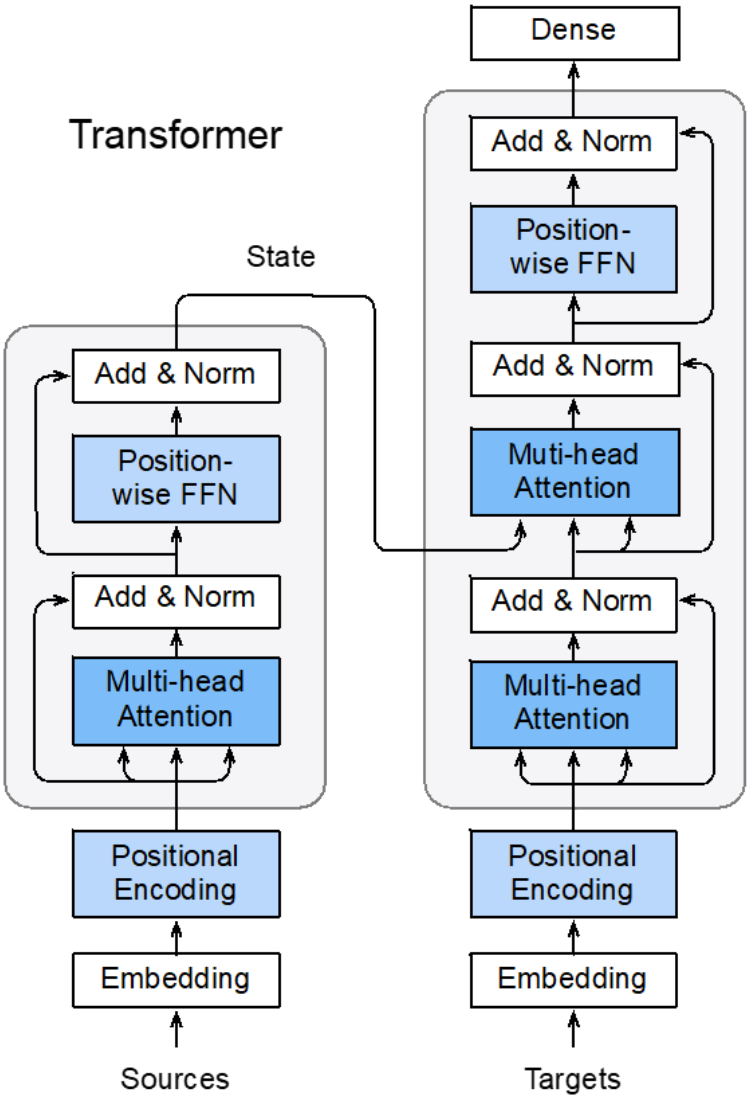
参数说明

n_{head}多头注意力的头数量

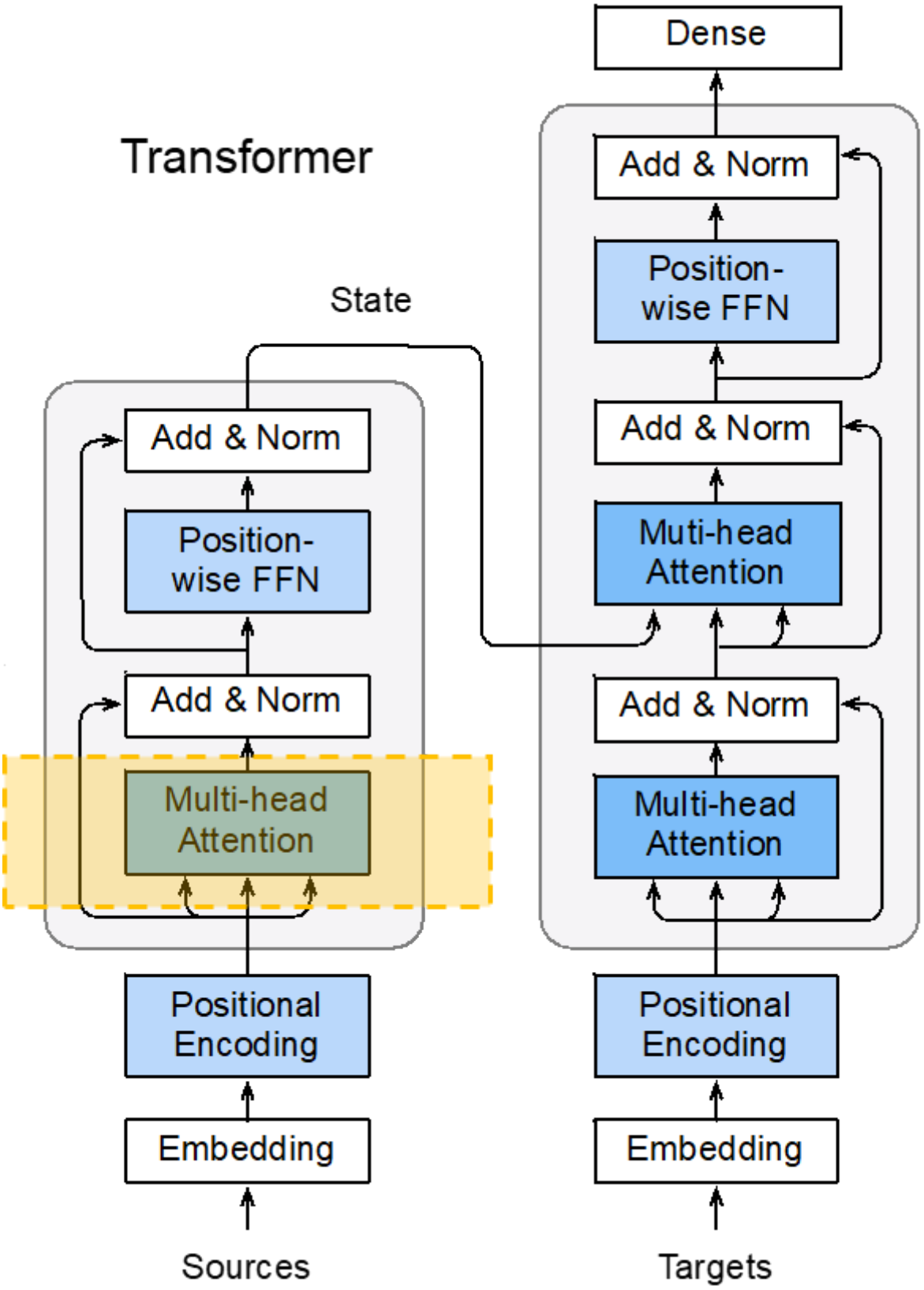
d_{model}：embedding维度

d_q, d_k, d_v是对应$Q = XW_Q, K = XW_K, V = XW_V$的$W_q, W_k, W_v$的维度。另外其实$d_q = d_v$否则无法生成

Transformer由论文《Attention is All You Need》提出，这篇文章要解决的是翻译问题，比如从中文翻译成英文。该文章完全放弃了以往经常采用的RNN和CNN，提出了一种完全基于**Attention**机制的新的网络结构，即Transformer，其中包括encoder和decoder两个部分，如下图所示，左边为encoder部分，右边为decoder部分：



2.1. Transformer的Self Attention



在深入了解Transformer的具体计算步骤之前，我们需要理解self-attention的概念。Self-attention，也称为自注意力，是Transformer架构的核心组件之一。Self-attention允许输入序列中的每个元素注意到同一序列中的其他元素。这意味着，对于序列中的每个位置，模型都会计算与所有其他位置的关联程度。这种机制使得模型能够捕捉序列内部的长距离依赖关系，而不受位置的限制。

在Transformer中，self-attention的主要优势包括：

1. 并行处理：不同于RNN，self-attention可以并行计算，大大提高了处理长序列的效率。

2. 长距离依赖：能够直接建立任意两个位置之间的联系，有效捕捉长距离依赖。
3. 灵活的上下文理解：通过动态分配注意力权重，模型可以根据上下文灵活地理解每个元素的含义。

现在，让我们详细了解self-attention的计算过程。

假设我们有一个长度为n的输入序列，表示为$X = \{x_1, x_2, \ldots, x_n\}$，每个$x_i$是一个$d$维的词向量表示。因此，输入矩阵$X$的维度为$n \times d$。

（1）计算查询、键和值

在attention机制中，首先将输入序列X映射为**Query（查询）**、**Key（键）**和**Value（值）**矩阵：

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

其中（一般也会令$d_v = d_k$）：

- $W_Q \in \mathbb{R}^{d \times d_k}$
- $W_K \in \mathbb{R}^{d \times d_k}$
- $W_V \in \mathbb{R}^{d \times d_v}$

Q、K和V的维度分别为：

- $Q \in \mathbb{R}^{n \times d_k}$
- $K \in \mathbb{R}^{n \times d_k}$
- $V \in \mathbb{R}^{n \times d_v}$

（2）计算注意力分数

对于整个输入序列，注意力分数通过查询矩阵Q和键矩阵K的点积来计算：

$$\text{Scores} = \frac{QK^T}{\sqrt{d_k}}$$

其中：

- $Q \in \mathbb{R}^{n \times d_k}$
- $K^T \in \mathbb{R}^{d_k \times n}$

点积QK^T的结果是一个$n \times n$的矩阵，其中每个元素Scores_{ij}表示输入序列中第i个位置的查询向量与第j个位置的键向量的相似度。为了数值稳定性，我们将这些分数除以$\sqrt{d_k}$。

假设查询和键向量的元素是独立的随机变量，均值为0，方差为1。那么它们的点积的方差将是d_k。除以$\sqrt{d_k}$可以将这个方差重新调整回1，保持不同维度下的一致性。如果不这样做的话，这可能导致softmax函数的输入有极大的正值或负值，从而使梯度变得非常小。除以$\sqrt{d_k}$可以帮助保持这些值在一个合理的范围内。实验中发现，对于较大的d_k值，如果没有这种缩放，点积注意力的性能会显著下降。

（3）计算注意力权重

接下来，通过softmax函数将这些注意力分数转换为注意力权重：

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

注意力权重矩阵的维度仍然是$n \times n$。

（4）计算注意力输出

使用注意力权重对值矩阵V进行加权求和，得到注意力输出：

$$\text{Attention Output} = \text{Attention Weights} \cdot V$$

其中：

- $V \in \mathbb{R}^{n \times d_v}$

最终的注意力输出是一个$n \times d_v$的矩阵，注意到，经过注意力机制后，我们的序列长度是保持不变的

2.2. Self Attention的优缺点

优点：

1. 一步到位的全局联系捕捉：Attention机制可以灵活的捕捉全局和局部的联系，从attention函数就可以看出来，它先是进行序列的每一个元素与其他元素的对比，在这个过程中每一个元素间的距离都是一，因此它比时间序列RNN的一步步递推得到长期依赖关系好的多，越长的序列RNN捕捉长期依赖关系就越弱。
2. 并行计算减少模型训练时间：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。但是CNN也只是每次捕捉局部信息，通过层叠来获取全局的联系增强视野。这样就不像rnn一样需要循环每一步计算，从计算上来看速度快了很多

缺点：缺乏位置信息。前面我们提到attention计算过程中每一个元素间的距离都是一，因此它不能捕捉语序信息。这在NLP中是存在比较大的问题，自然语言的语序是包含太多的信息。如果缺失了这方面的信息，结果往往回打折扣。要解决这个缺点，只要添加位置信息就好了。所以就有了position-embedding（位置向量）的概念。Transformer模型就是在Attention的基础上添加了位置向量的模型。

2.3. embedding

论文的原文中，会在嵌入层中权重乘以$\sqrt{d_{\text{model}}}$。

实现：

```
self.embedding(x) * math.sqrt(self.d_model)
```

2.3.1. tied-embedding

Transformer原文3.4中提到一点：

“

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model}. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to [24]. In the embedding layers, we multiply those weights by $\sqrt{d_{\text{model}}}$.

”

与其他序列转换模型类似，我们使用学习得到的嵌入将输入的标记和输出的标记转换为维度为d_{model}的向量。我们还使用常规的学习到的线性变换和Softmax函数，将解码器的输出转换为预测的下一个标记的概率。在我们的模型中，我们在两个嵌入层和Softmax之前的线性变换之间共享相同的权重矩阵，类似于[24]。在嵌入层中，我们将这些权重乘以$\sqrt{d_{\text{model}}}$。

实际上，在输入层和输出层之间共享参数就是tied-embedding。一是这样可以减少参数数量，其次可以发现，这两个矩阵完成的事情本质是一样的，输入层从词表的维度转换为dmodel。而在输出层，则是把dmodel转换为词表大小，再通过softmax转化为概率。在实验中，发现这样确实也能够提高性能。

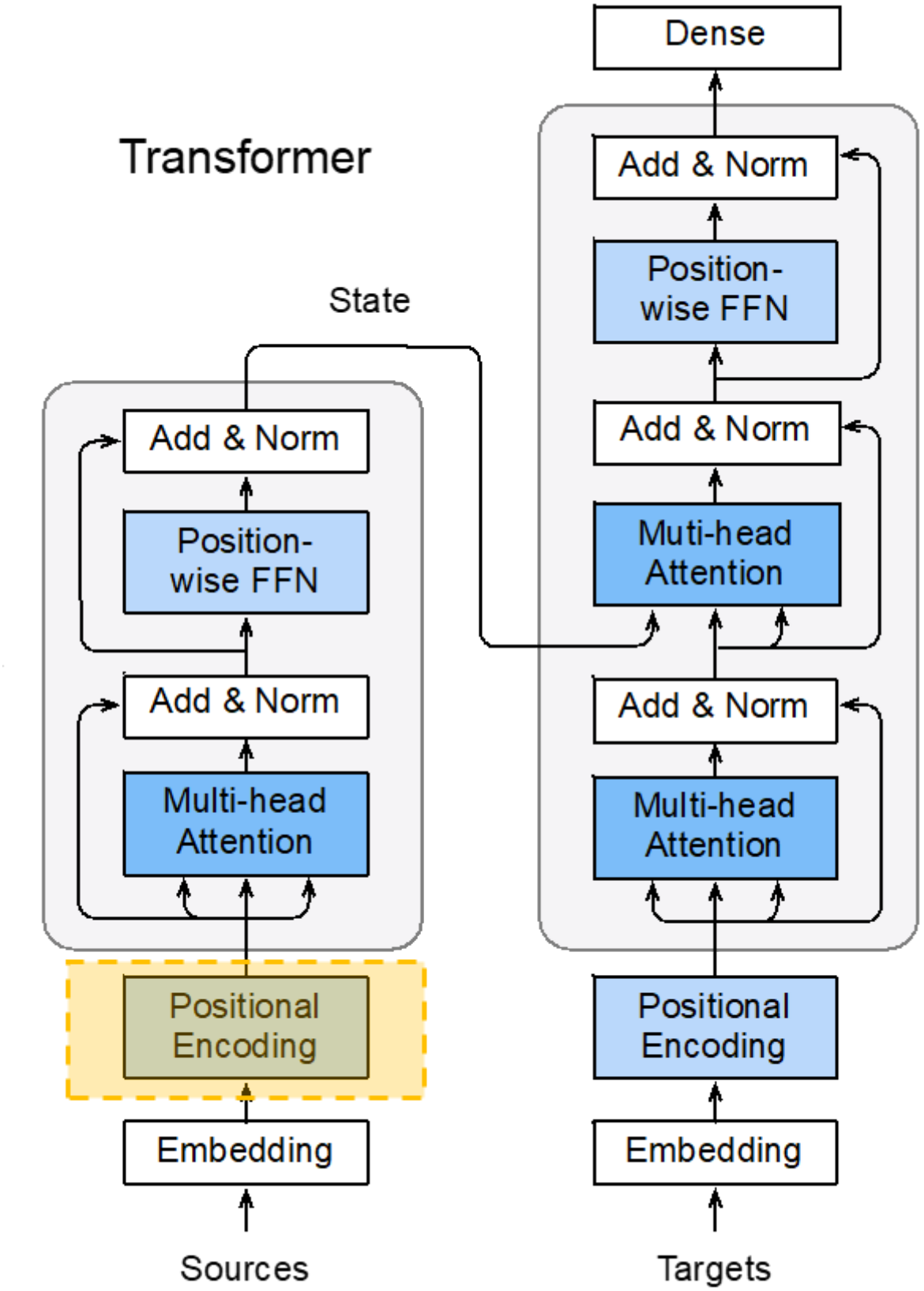
虽然在输入层我们还是可以用普通的初始化方法来保证position encoding的大小相对一致，但是对于最后一层来说，则还是采用xavier初始化更好。对于使用xavier初始化，embedding每个维度服从的分布则是$N\left(0, 1/d_{\text{model}}\right)$。这就是为什么输入层一般乘以$\sqrt{d_{\text{model}}}$。

2.4. 位置编码

参考资料：

（1）https://zhuanlan.zhihu.com/p/454482273

（2）https://zhuanlan.zhihu.com/p/95079337

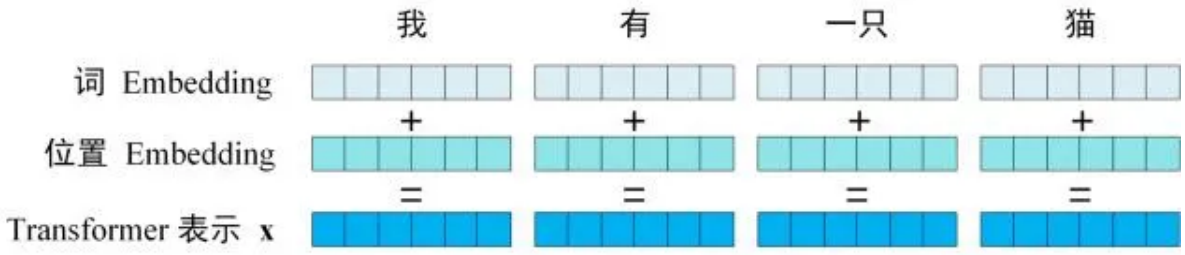


自注意力机制在计算时是并行处理所有元素的，因此没有直接考虑到序列元素的位置信息。这可能导致严重的理解错误。考虑这个例子：'狗咬了人' 和 '人咬了狗'。这两个句子包含完全相同的词，但意思截然不同。如果没有位置编码，自注意力机制可能无法区分这两个句子的不同含义，因为每个词在两个句子中的初始嵌入是相同的。

虽然在某些情况下，上下文信息可能帮助模型部分区分词的作用，但在这种极端的例子中（上下文完全一致），没有明确的位置信息，模型将很难正确理解句子的真实含义。这清楚地表明了在Transformer模型中引入位置编码的必要性。位置编码能够帮助模型准确捕捉词序信息，从而正确理解句子结构和含义，即使在词序变化会导致意思完全相反的情况下也能正确处理。

在Transformer模型中，位置信息的表示通常被称为"positional encoding"（位置编码）。在原始的Transformer论文中是使用预定义的数学函数（通常是正弦和余弦函数）来生成位置向量，这些向量直接加到输入嵌入上，不需要学习。但是后来有一些新的方法，使用的是Positional Embedding（位置嵌入），使用可学习的嵌入向量来表示每个位置，这些些嵌入向量在训练过程中与模型的其他参数一起学习。

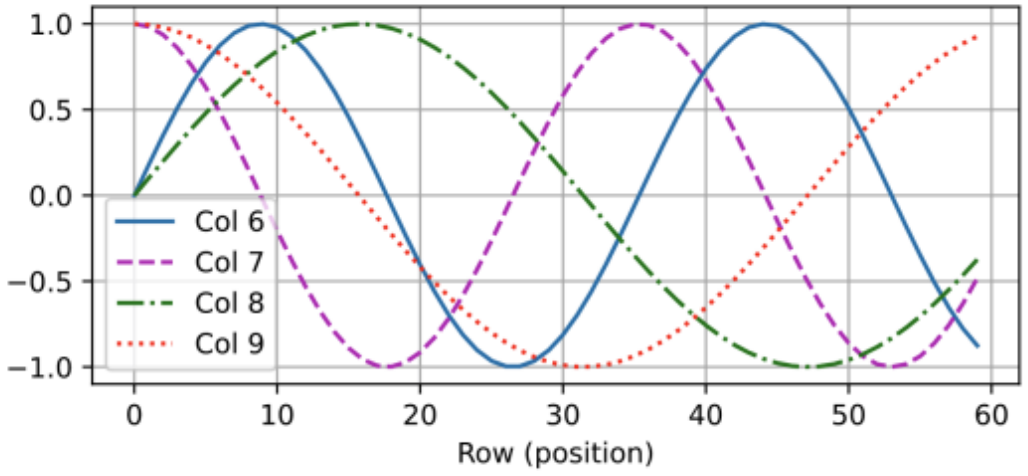
Transformer 中单词的输入表示 **x**由**单词 Embedding** 和**位置 Encoding** 相加得到，如下图所示：



可以发现，单词的维度和位置emconding的维度是一样的，具体来说，位置编码有如下特点

- 位置编码将位置信息注入到输入里
- 假设长度为 n 的序列是 $\mathbf{X} \in \mathbb{R}^{n \times d}$(包含一个序列中(n)个词元的(d)维嵌入表示), 那么使用位置编码矩阵 $\mathbf{P} \in \mathbb{R}^{n \times d}$ 来输出 $\mathbf{X} + \mathbf{P}$ 作为自编码输入
- $\mathbf{P}$ 的元素如下计算:（不同列的周期不一样）(i是在序列长度的不同位置，j对应的是不同属性)（属性编码越大，周期越长）

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



实际上，为什么使用这样的位置编码，是有原因的，这个位置编码得满足一些条件：

（1）最直观的思路，我们是否可以使用1,2,3,4？但这带来2个问题，一个是由于没有限制，因此不同句子的上界不一样，尤其是在预测时遇到比训练更长的句子，可能就会存在问题；其次是由于位置编码没有上街

（2）那我们把范围限制在[0,1]，最初的位置是0，最后的位置是1，之后按照长度等比例分割？但这也会带来一个问题，但序列长度不同时，token间的相对距离是不一样的。例如在序列长度为3时，token间的相对距离为0.5；在序列长度为4时，token间的相对距离就变为0.33。

（3）使用sin()这样的周期函数。这已经接近满足我们的需要了，但是我们希望一点是，不同的位置向量是可以通过线性转换得到。

基于以上考虑就得到我们的position encoding。

这样的编码方式有特点如下：

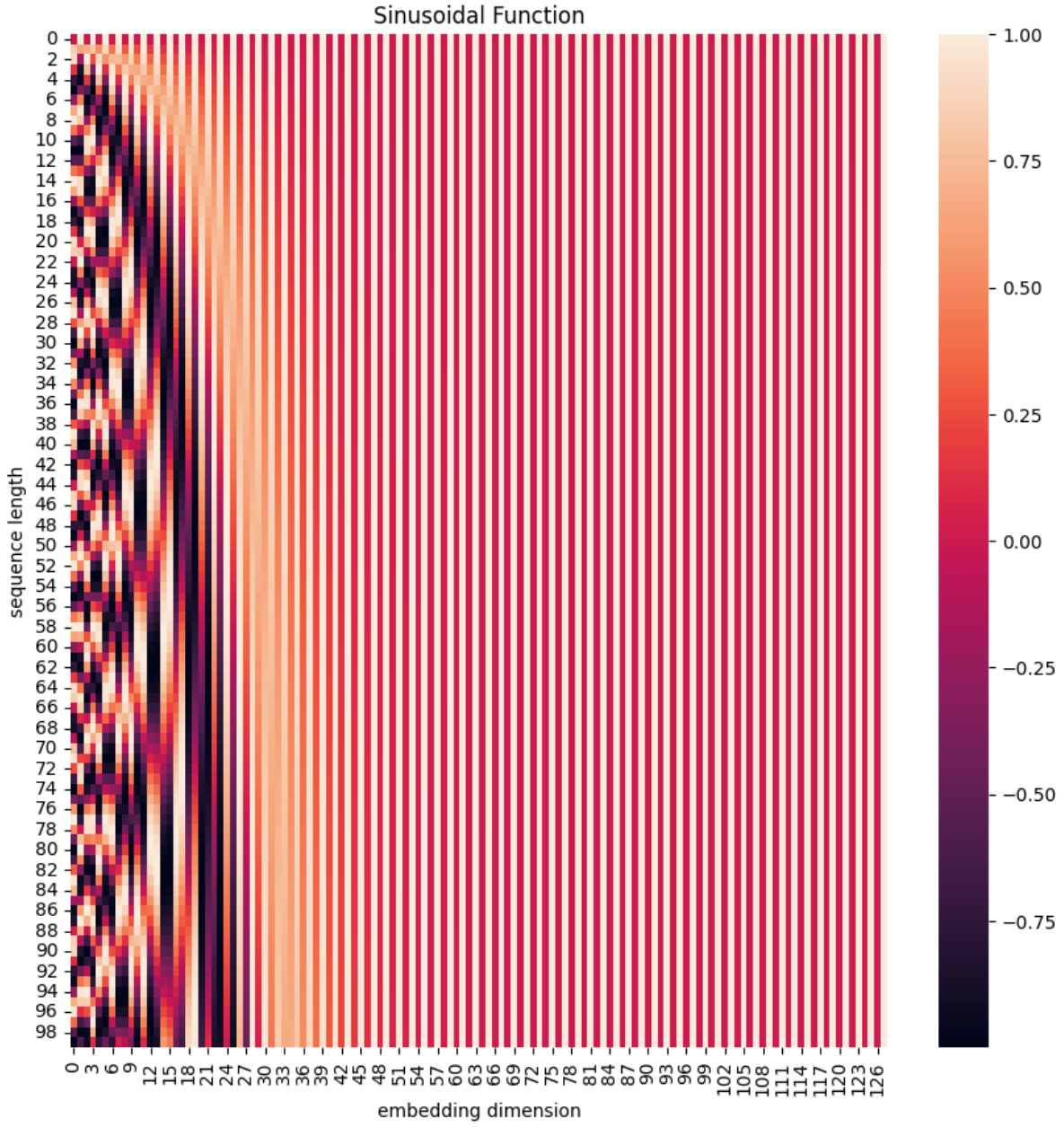
- 在self attention中，不同样本同一位置同一属性的位置编码一样
- 句子长度在10000以内编码就不会重复。底数是10000，所以10000就不会重复。这个指的是句子长度，一般很少有句子能够达到10000的句子长度
- 体现一定的先后次序，并且在一定范围内的编码差异不应该依赖于文本的长度，具有一定的不变性。(即序列长度为5的情况下，i=3和i=5的距离在总长度在10和100的两个序列中距离是一样的。这就不能用位置/最大长度)
- 使 PE 能够适应比训练集里面所有句子更长的句子，假设训练集里面最长的句子是有 20 个单词，突然来了一个长度为 21 的句子，则使用公式计算的方法可以计算出第 21 位的 Embedding。
- 除了绝对位置，可以让模型容易地计算出相对位置，对于固定长度,Δt位置于 $t+\Delta t$ 处的位置编码可以线性变换获得$\begin{pmatrix}\sin(t+\Delta t)\\ \cos((t+\Delta t)\end{pmatrix}=\begin{pmatrix}\cos\Delta t & \sin\Delta t\\ -\sin\Delta t & \cos\Delta t\end{pmatrix}\begin{pmatrix}\sin t\\ \cos t\end{pmatrix}$

  除了绝对位置，相对位置也被编码了，因为sin(pos+L)=sin(pos)cos(L)+cos(pos)sin(L)较远的相对位置编码可以由较近的位置编码组合出来，且组合系数都是预先能算好的数字，容易计算且组合的方法可由神经网络学到，因此神经网络可以更容易学到相对位置的规律，cos的情况类似。（这种设计，可以让模型更好学习以及泛化未见过的长序列句子）
- 使用把位置信息加入模型，好处是不改变模型结构，不增加参数。但是坏处是，模型不一定能够把位置信息识别出来

我们观察一下位置编码直观长什么样子,注意我们首先是按照原有的公式直观地实现：

```
# 修改了原有代码，原有代码0的位置不对
def get_positional_encoding(max_seq_len, embed_dim):
    # 初始化一个positional encoding
    # embed_dim: 字嵌入的维度
    # max_seq_len: 最大的序列长度
    # i 表示位置的索引，j 表示每个位置的每个维度的数值
    positional_encoding = np.array([
        [i / np.power(10000, 2 * j / embed_dim) for j in range(embed_dim)]
        for i in range(max_seq_len)])
    positional_encoding[:, 0::2] = np.sin(positional_encoding[:, 0::2])  # dim 2j
偶数
    positional_encoding[:, 1::2] = np.cos(positional_encoding[:, 1::2])  # dim
2j+1 奇数
    # 归一化，用位置嵌入的每一行除以它的模长
    # denominator = np.sqrt(np.sum(position_enc**2, axis=1, keepdims=True))
    # position_enc = position_enc / (denominator + 1e-8)
    return positional_encoding

positional_encoding = get_positional_encoding(max_seq_len=100, embed_dim=128)
plt.figure(figsize=(10,10))
sns.heatmap(positional_encoding)
plt.title("Sinusoidal Function")
plt.xlabel("embedding dimension")
plt.ylabel("sequence length")
```



上图是位置编码图，可以发现前面的维度变化较快，后面的维度变化较慢

实现代码：

```
############################## PE位置编码；##############################
class Positional_Encoding(nn.Module):
    def __init__(self,d_model:int,dropout:float,len_position=500):
        super(Positional_Encoding,self).__init__()

        # 在训练阶段按概率p随即将输入的张量元素随机归零，常用的正则化器，用于防止网络过拟合；
        self.dropout = nn.Dropout(p=dropout)
        # 生成的PE维度为（len_position,d_model）,最大支持len_position长度的句子，每个位置
有d_model个特征；
        self.PE = torch.zeros(len_position,d_model)
        #  从0到len_position-1构造一个向量，再把维度扩展为(len_position,1)；
        position = torch.arange(0.,len_position).unsqueeze(1)
        # 首先实现三角函数的里面部分sin(div_term)和cos(div_term)；
        # $(\frac{1}{10000^{2j/d}})=(10000^{-2j/d}) = \exp(-2j\log(10000)/d)$
```

```
        # shape of div_term: (d_model/2)
        div_term = torch.exp(torch.arange(0., d_model, 2) * (-(math.log(10000.0)
/ d_model)))
        # position * div_term进行广播，维度为(len_position,d_model/2)；
        # self.PE[:,0::2]表示从0开始，步长为2，即偶数位置；self.PE[:,1::2]表示从1开始，步
长为2，即奇数位置；
        self.PE[:,0::2] = torch.sin(position * div_term) # 偶数位置使用sin编码；
        self.PE[:,1::2] = torch.cos(position * div_term) # 基数位置使用cos编码；
        #插上batch这个维度；
        self.PE = self.PE.unsqueeze(0)
        # self.register_buffer("PE",self.PE)

    def forward(self,x):
        # print("DEBUG:X.SIZE____:",np.shape(x))
        # print("DEBUG:PE.SIZE____:",np.shape(self.PE))
        # 广播，等于每个样本都加上PE；
        # self.PE[:,:x.size(1)]就是保留输入长度的位置编码，多余的部分不要；
        x = x + self.PE[:,:x.size(1)]
        return self.dropout(x)
```

2.5. 多头注意力机制

在Transformer中，通常使用**多头注意力机制（Multi-Head Attention）**来增强模型的表现力。在上面，我们介绍的是一个单头注意力，对于多头注意力，我们有：

我们假设其中有m个头，对于每一个头 i，我们有查询矩阵 $W_i^Q \in R^{d\times d_k}$，键矩阵 $W_i^K \in R^{d\times d_k}$，值矩阵 $W_i^V \in R^{d\times d_v}$，以及输出投影矩阵 $W^O \in R^{md_k\times d_{\text{model}}}$。

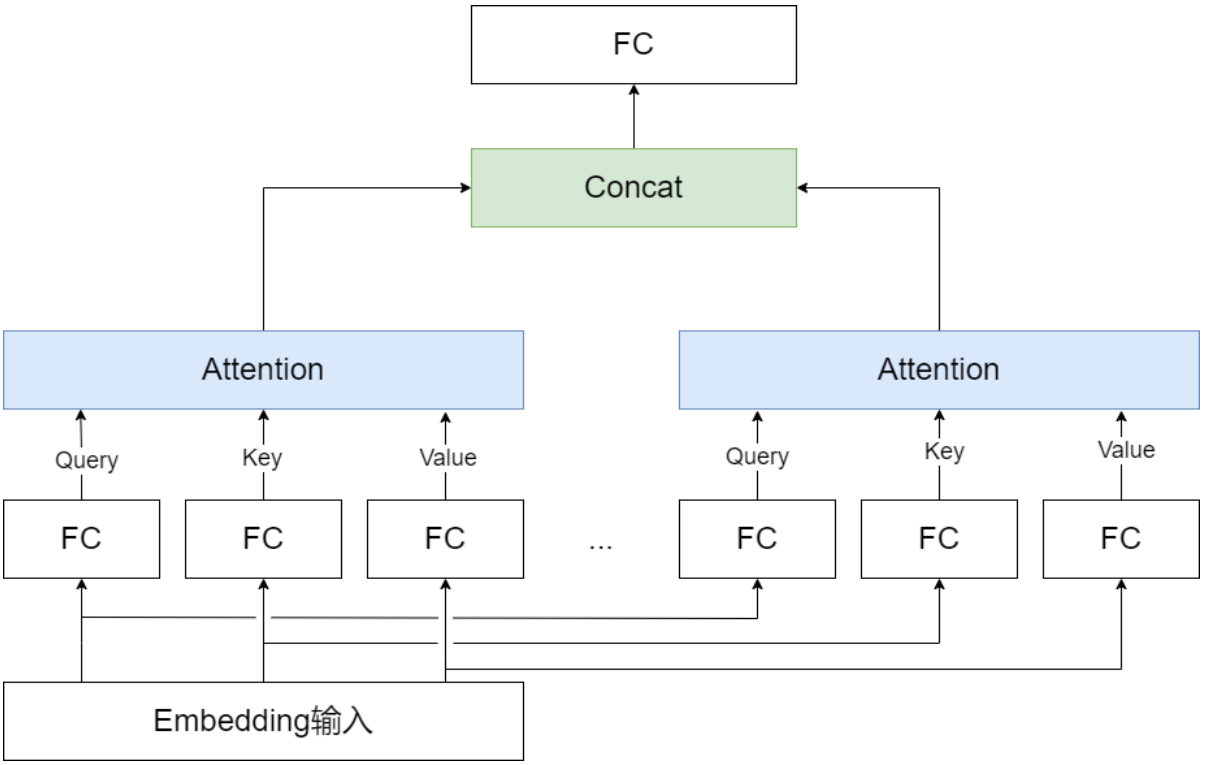
对于输入$X \in R^{n\times d}$，每个头的输出是

$$\text{head}_i = \text{Attention}\left(XW_i^Q, XW_i^K, XW_i^V\right)$$

然后，所有头的输出被拼接 (concatenated) 在一起，并通过输出投影矩阵进行线性转换得到最终的输出，因为是在隐藏层维度，即特征维度d_k进行拼接，有m个头的话，有 $W^O \in R^{(m\times d_v)\times d_{\text{model}}}$。

$$\text{MultiHead}(Q,K,V) = [\text{ head }_1; \text{head}_2; \ldots; \text{head}_h]W^O$$

其中 $[;]$ 表示拼接操作,在最后一个维度上进行拼接,拼接后的维度变为 $n\times(m\times d_v)$。
注意，每个头都有自己独立的参数矩阵 W_i^Q, W_i^K 和 W_i^V，这样可以使模型从不同的子空间学 习输入序列的表示。在所有头计算完成后，他们的结果被拼接并经过最后的线性变换以得到 最终输出。



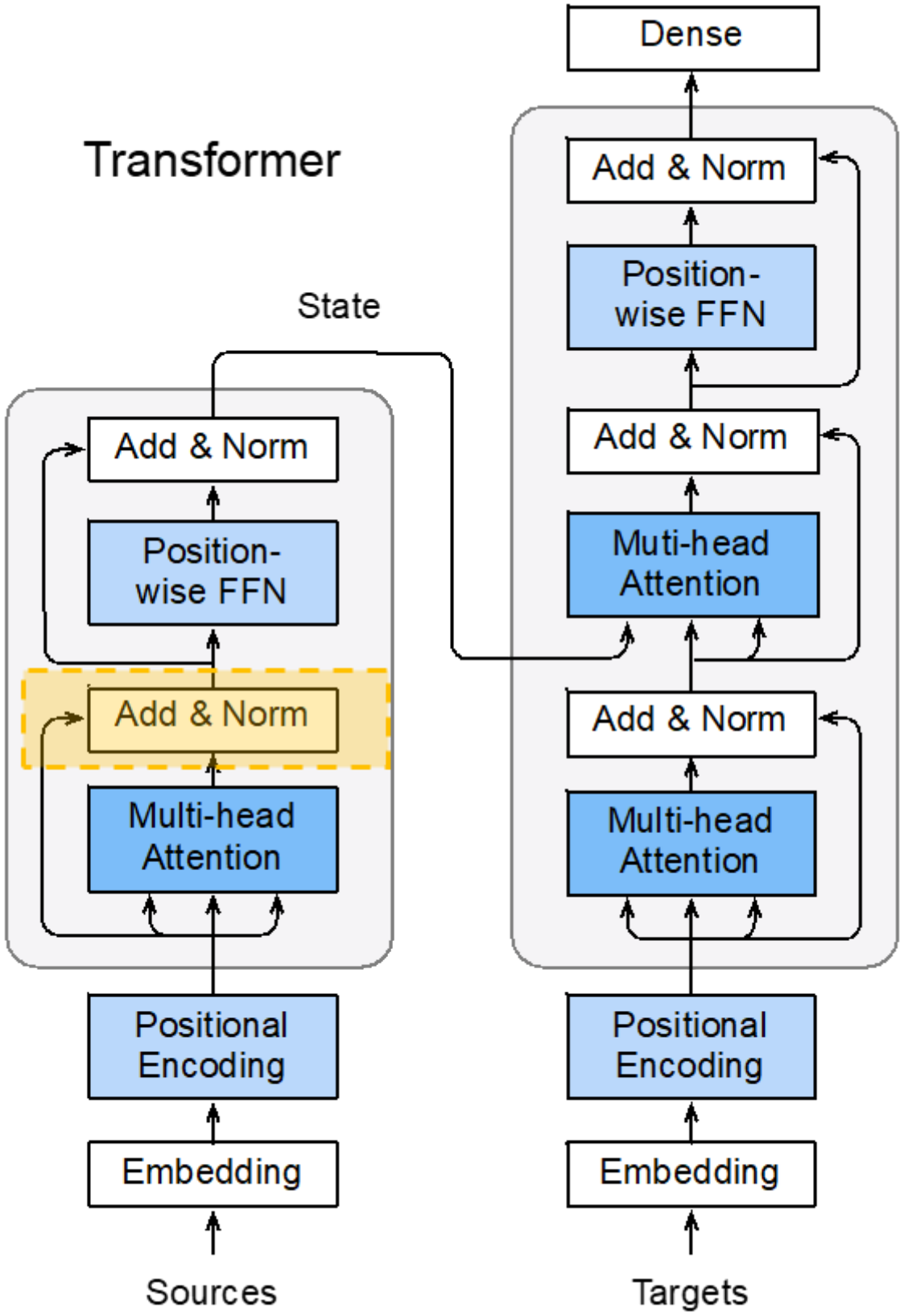
实际上，在实现的时候，我们其实可以直接实现批量化多头，具体来说：

1. 输入处理：
   假设输入序列 $X \in \mathbb{R}^{n \times d}$，其中 n 是序列长度，d 是输入维度。
2. 权重矩阵构建：
   我们将所有头的权重矩阵合并成大矩阵：
   $W_Q = [W_1^Q; W_2^Q; \ldots; W_m^Q] \in \mathbb{R}^{d \times (m \times d_k)}$
   $W_K = [W_1^K; W_2^K; \ldots; W_m^K] \in \mathbb{R}^{d \times (m \times d_k)}$
   $W_V = [W_1^V; W_2^V; \ldots; W_m^V] \in \mathbb{R}^{d \times (m \times d_v)}$
3. 计算查询、键和值：
   $Q = XW_Q \in \mathbb{R}^{n \times (m \times d_k)}$
   $K = XW_K \in \mathbb{R}^{n \times (m \times d_k)}$
   $V = XW_V \in \mathbb{R}^{n \times (m \times d_v)}$
4. 重塑查询、键和值为三维矩阵：
   $Q' = \text{reshape}(Q) \in \mathbb{R}^{n \times m \times d_k}$
   $K' = \text{reshape}(K) \in \mathbb{R}^{n \times m \times d_k}$
   $V' = \text{reshape}(V) \in \mathbb{R}^{n \times m \times d_v}$
5. 转置以便进行批量矩阵乘法：
   $Q'' = \text{transpose}(Q', [1, 0, 2]) \in \mathbb{R}^{m \times n \times d_k}$
   $K'' = \text{transpose}(K', [1, 0, 2]) \in \mathbb{R}^{m \times n \times d_k}$，注意 $(K'')^T \in \mathbb{R}^{m \times d_k \times n}$
   $V'' = \text{transpose}(V', [1, 0, 2]) \in \mathbb{R}^{m \times n \times d_v}$
6. 计算注意力分数：
   $\text{Scores} = \frac{Q''(K'')^T}{\sqrt{d_k}} \in \mathbb{R}^{m \times n \times n}$
7. 应用softmax（在最后一个维度应用）：
   $\text{Attention Weights} = \text{softmax}(\text{Scores}) \in \mathbb{R}^{m \times n \times n}$

8. 计算注意力输出：
   $$\text{Attention Output} = \text{Attention Weights} \cdot V'' \in \mathbb{R}^{m \times n \times d_v}$$
9. 转置回原始序列顺序：
   $$\text{Attention Output}' = \text{transpose}(\text{Attention Output}, [1, 0, 2]) \in \mathbb{R}^{n \times m \times d_v}$$
10. 重塑以准备最终线性变换：
    $$\text{Attention Output}'' = \text{reshape}(\text{Attention Output}') \in \mathbb{R}^{n \times (m \times d_v)}$$
11. 最终线性变换：
    $$\text{MultiHead Output} = \text{Attention Output}'' \cdot W^O \in \mathbb{R}^{n \times d_{\text{model}}}$$

这个并行实现方法允许我们在一次操作中处理所有的注意力头，大大提高了计算效率。最终的输出维度为 $n \times d_{\text{model}}$，保持了与输入序列相同的长度，但将特征维度映射到了模型的隐藏维度 d_{model}。

2.6. Add & Norm



Add & Norm 层由 Add 和 Norm 两部分组成：

1. Add（残差连接）：
   假设输入是 x，经过某个子层（可能是多头注意力层或前馈网络层）的输出是 $F(x)$，则Add操作为：

   $x_{add} = x + F(x)$

这是一个残差连接，它有助于缓解深度网络中的梯度消失问题，并允许信息更直接地在网络中流动。

2. Norm（层归一化）：
   在Add之后，立即进行层归一化操作。层归一化对每个样本独立进行，计算如下：
   $x_{norm} = \text{LayerNorm}(x_{add}) = \gamma \cdot \frac{x_{add} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$
   其中：
   - μ 是 x_{add} 在特征维度上的均值
   - σ^2 是 x_{add} 在特征维度上的方差
   - ϵ 是一个很小的数，用于数值稳定性（通常是 10^{-6}）
   - γ 和 β 是可学习的缩放和偏移参数
   - LayerNorm是在特征维度上进行计算的，即有
     $\mu = \mu_{b,l} = \frac{1}{D}\sum_{d=1}^{D} x_{b,l,d}, \quad \sigma = \sigma_{b,l}^2 = \frac{1}{D}\sum_{d=1}^{D}(x_{b,l,d} - \mu_{b,l})^2$,其中b是指第b个样本，l代指第b个句子中的第l个位置

完整的Add & Norm操作:
将上述两步结合，我们可以写出完整的Add & Norm操作：

$$\text{AddNorm}(x, F) = \text{LayerNorm}(x + F(x))$$

在Transformer中，这个操作通常应用于：

a) 多头注意力层之后：
$x_1 = \text{AddNorm}(x, \text{MultiHeadAttention}(x))$

b) 前馈网络层之后：
$x_2 = \text{AddNorm}(x_1, \text{FFN}(x_1))$

Add & Norm操作的作用：

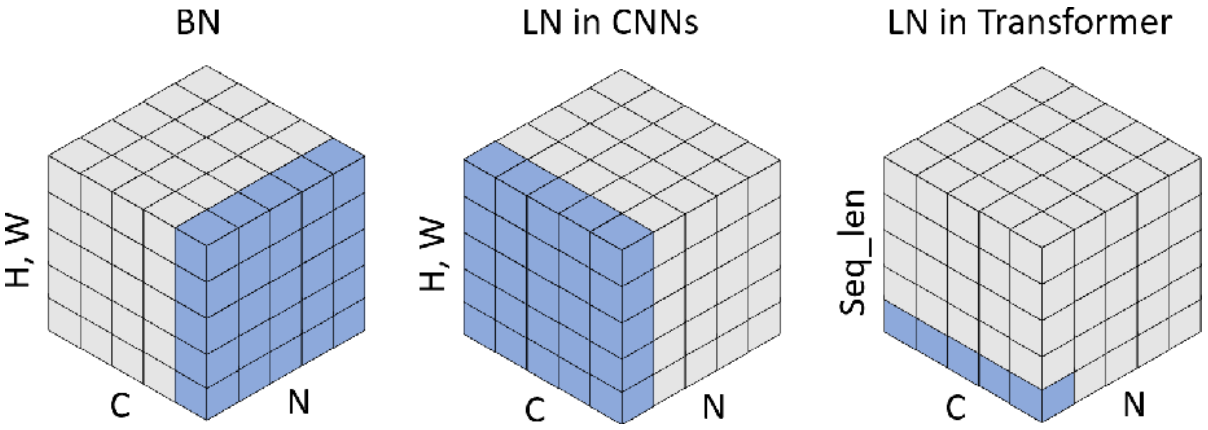
1. 残差连接（Add）帮助梯度在深层网络中更容易流动，缓解梯度消失问题。
2. 层归一化（Norm）有助于稳定深层网络中的激活值分布，加速训练过程。
3. 这种结构允许网络更容易学习恒等映射，这在某些情况下是有益的。

通过这种方式，Transformer能够构建非常深的网络结构，同时保持良好的训练特性和性能。

2.6.1. LayerNormalization

过规范化输入数据，层规范化有助于减少内部协变量偏移，从而加速训练过程并提高模型的稳定性。
LayerNormalization最早被提出就是应用于循环神经网络中。

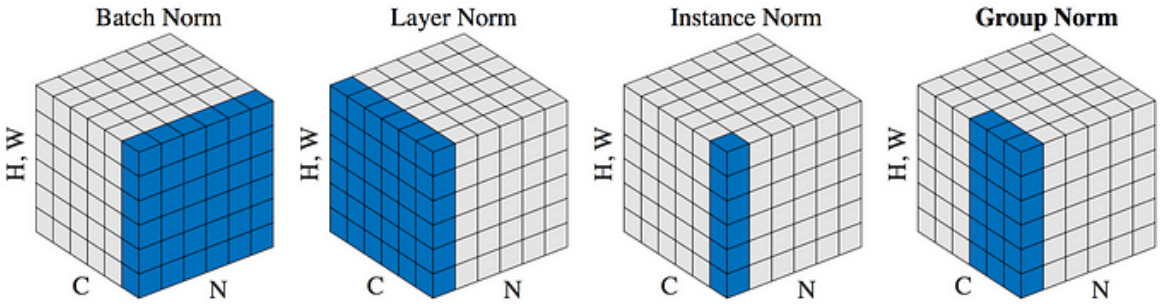
我们先看看LN是怎么计算的



来源：Leveraging Batch Normalization for Vision Transformers

上图展示了不同Normalization的计算方式，可以看到LayerNormalization是对每个样本句子的每个位置中的所有特征进行标准化计算的。相比于BN，这样更有利于适应在NLP任务中，句子长度不一致的问题。（值得注意的是，虽然传统CV更多使用呢BN，但是在VIT中，把LN改为BN，性能可以观察到明显的下降）。

其他Normalization的举例：



此外，也有论文进一步优化Transformer中Normalization的处理方法。

原始transformer中，采用的是Post-LN，即LN在残差连接之后。On Layer Normalization in the Transformer Architecture 这篇论文提出了一种Pre-LN的架构，即LN在残差连接之前。两种LN架构的具体形式如下。

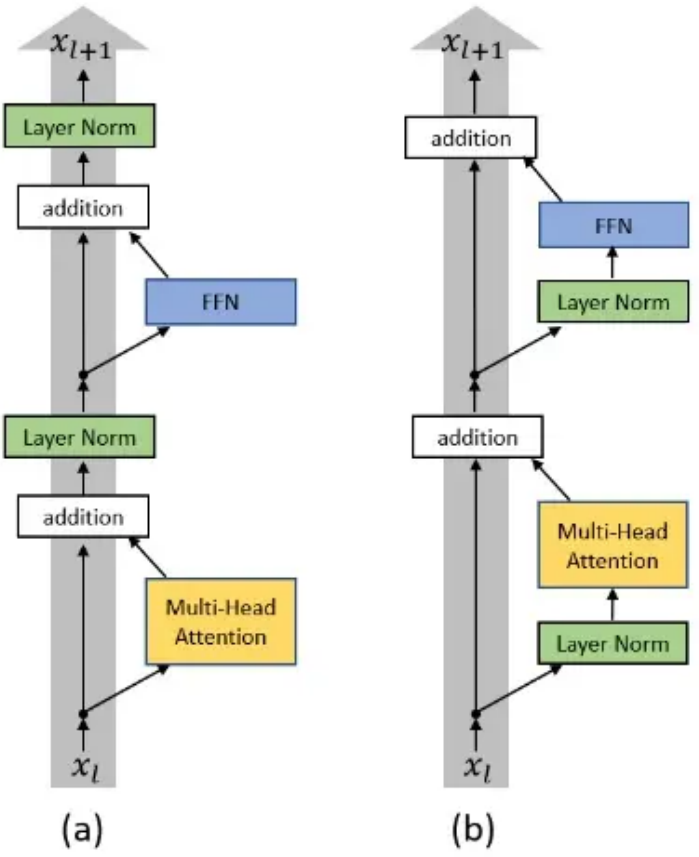


Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

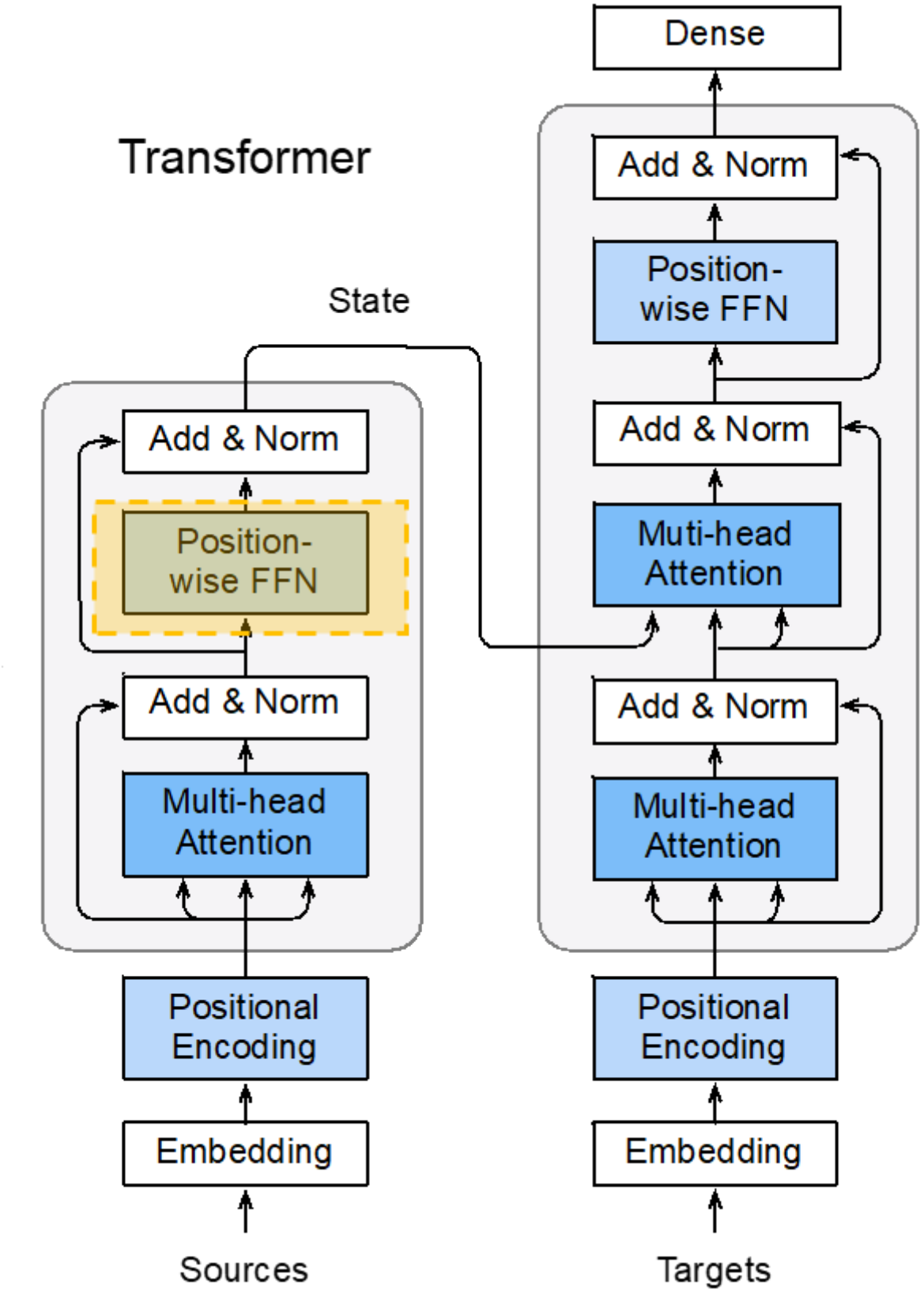
这篇论文通过理论分析和实验的方式，证明了Pre-LN相比的Post-LN的优势，主要表现在：

- 在learning rate schedular上，Pre-LN不需要采用warm-up策略，而Post-LN必须要使用warm-up策略才可以在数据集上取得较好的Loss和BLEU结果。
- 在收敛速度上，由于Pre-LN不采用warm-up，其一开始的learning rate较Post-LN更高，因此它的收敛速度更快。
- 在超参调整上，warm-up策略带来了两个需要调整的参数： lrmax （最大学习率）和 Twarmup (warmup过程的总步数）。这两个参数的调整将会影响到模型最终的效果。而由于transformer模型的训练代价是昂贵的，因此多引入超参，也给模型训练带来了一定难度。

总结看来，Pre-LN带来的好处，基本都是因为不需要做warm-up引起的。而引起这一差异的根本原因是：

- Post-LN在输出层的gradient norm较大，且越往下层走，gradient norm呈现下降趋势。这种情况下，在训练初期若采用一个较大的学习率，容易引起模型的震荡。
- Pre-LN在输出层的gradient norm较小，且其不随层数递增或递减而变动，保持稳定。
- 无论使用何种Optimzer，不采用warm-up的Post-LN的效果都不如采用warm-up的情况，也不如Pre-LN。

2.7. 基于位置的前馈网络Position-wise Feed-Forward Network



FFN本质上是两个线性变换，中间有一个ReLU激活函数，Position-wise FFN为Transformer提供了重要的非线性转换和特征处理能力，同时也适应了序列长度可变的情况。"Position-wise"意味着这个网络独立地应用于输入序列的每个位置：

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

其中：

- x 是输入向量
- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ 和 $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ 是权重矩阵
- $b_1 \in \mathbb{R}^{d_{\text{ff}}}$ 和 $b_2 \in \mathbb{R}^{d_{\text{model}}}$ 是偏置向量
- d_{model} 是模型的隐藏维度
- d_{ff} 是FFN的内部维度，通常比 d_{model} 大

在实际实现中，为了提高计算效率，我们通常对整个批次和序列进行批量处理。这涉及到输入形状的转换：

1. 输入转换：将输入从 (b, n, d_{model}) 变换为 $(b \cdot n, d_{\text{model}})$
   其中 b 是批次大小，n 是序列长度。
2. FFN处理：
   $\text{FFN}(X) = \max(0, XW_1 + b_1)W_2 + b_2$
   其中 $X \in \mathbb{R}^{(b \cdot n) \times d_{\text{model}}}$ 是转换后的输入。
3. 输出转换：将输出从 $(b \cdot n, d_{\text{model}})$ 变回 (b, n, d_{model})

这种实现方式有几个优点：

- 它允许我们高效地处理整个批次和序列。
- 它适应不同样本可能有不同有效序列长度的情况（由于padding）。
- 这种操作实际上等价于使用两层核窗口为1的一维卷积层。

2.8. 超参数设置

在论文中，Transformer模型的主要超参数设置如下：

$$
\begin{aligned}
&N = 6 \text{ (编码器和解码器的层数)} \\
&d_{\text{model}} = 512 \text{ (模型维度)} \\
&d_{ff} = 2048 \text{ (前馈网络内层维度)} \\
&h = 8 \text{ (注意力头数)} \\
&d_k = d_v = d_{\text{model}}/h = 64 \text{ (每个头的键和值维度)} \\
&P_{drop} = 0.1 \text{ (dropout率)} \\
&\epsilon_{ls} = 0.1 \text{ (标签平滑值)}
\end{aligned}
$$

对于更大的模型（"big" Transformer），论文中提到：

$$
\begin{aligned}
&d_{\text{model}} = 1024 \\
&d_{ff} = 4096 \\
&h = 16 \\
&P_{drop} = 0.3 \text{ (除了英法翻译任务使用 } 0.1\text{)}
\end{aligned}
$$

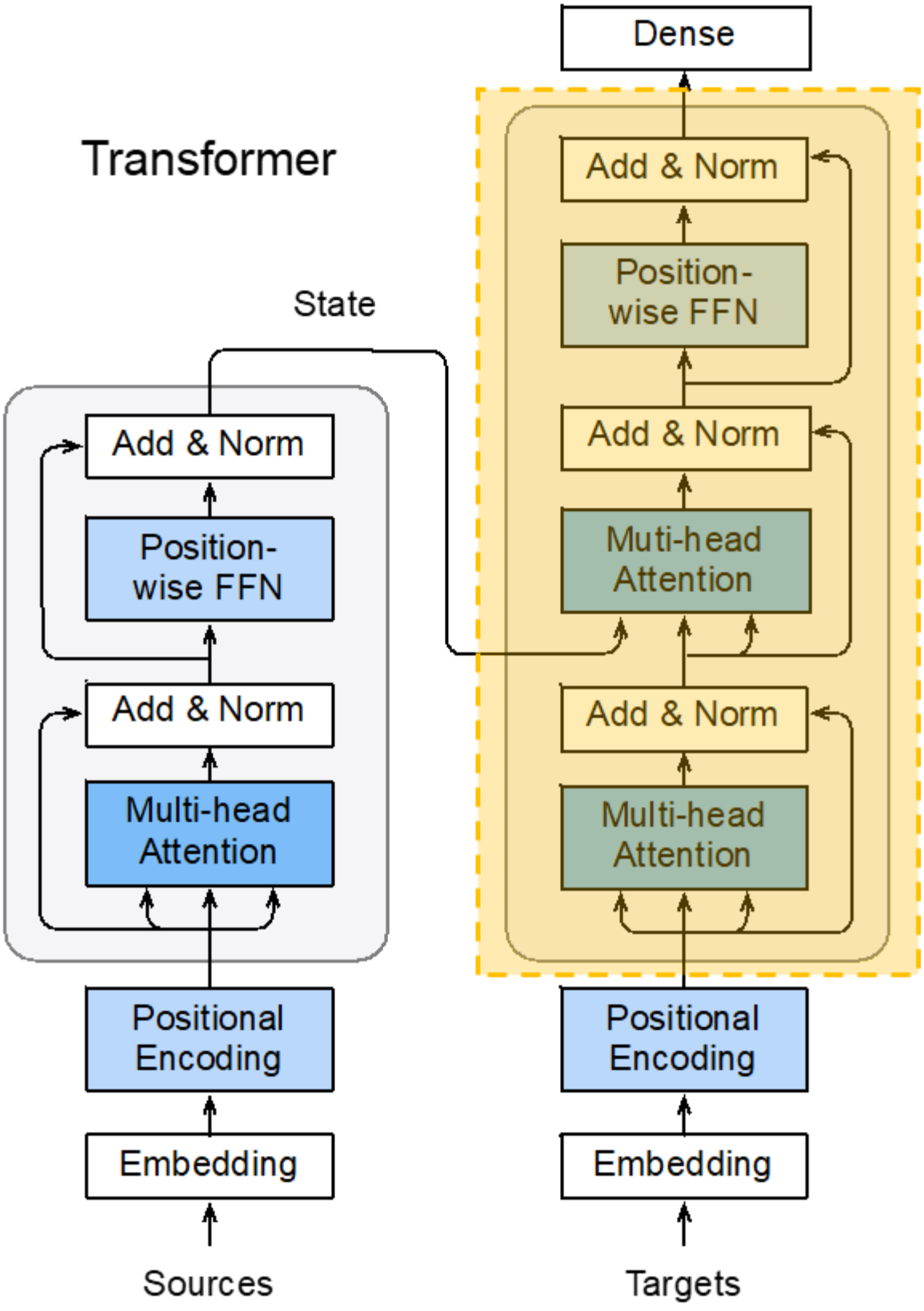
其他训练相关的超参数包括：

$$
\begin{aligned}
&\text{optimizer} = \text{Adam} \\
&\beta_1 = 0.9 \\
&\beta_2 = 0.98 \\
&\epsilon = 10^{-9} \\
&\text{warmup\_steps} = 4000
\end{aligned}
$$

学习率的调整公式为：

$$\text{lrate} = d_{\text{model}}^{-0.5} \cdot \min(\text{step\_num}^{-0.5}, \text{step\_num} \cdot \text{warmup\_steps}^{-1.5})$$

2.9. Decoder



上图黄色部分为 Transformer 的 Decoder block 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**K, V**矩阵使用 Encoder 的**编码信息**进行计算，而**Q**使用上一个 Decoder block 的输出计算。
- 最后有一个 Softmax 层计算下一个单词的概率。

2.9.1. 掩码(Mask)机制

引入掩码机制的目的

在Transformer中，主要有两个地方会用到掩码这一机制：

（1）用于在训练过程中解码的时候掩盖掉当前时刻之后的信息，即**在训练时屏蔽掉来自“未来”的信息**，称为**Attention Mask**；

（2）是对一个batch中不同长度的序列在Padding到相同长度后，对Padding部分的信息进行掩盖，**即用于屏蔽掉无效的padding区域**，称为**Padding Mask。**

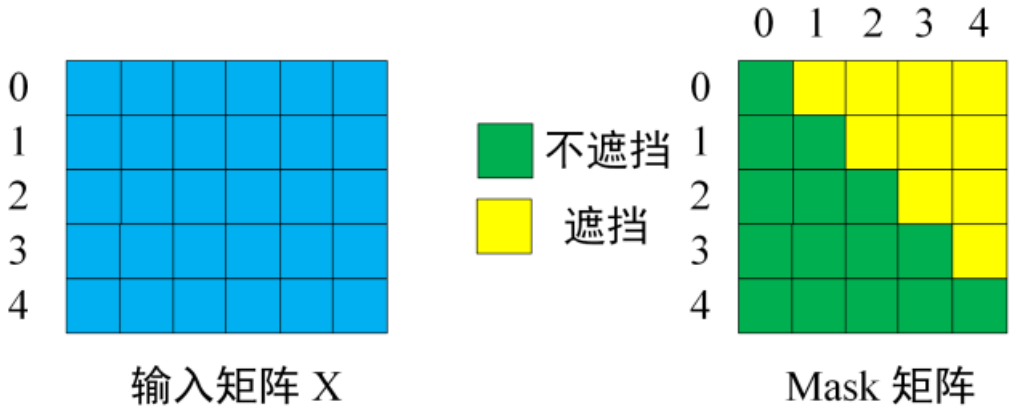
Encoder中的掩码主要是起到第二个作用，Decoder中的掩码则同时发挥着两种作用，一般来说，我们说的Attention Mask只在解码器中使用

2.9.1.1. 第一个Multi-Head Attention

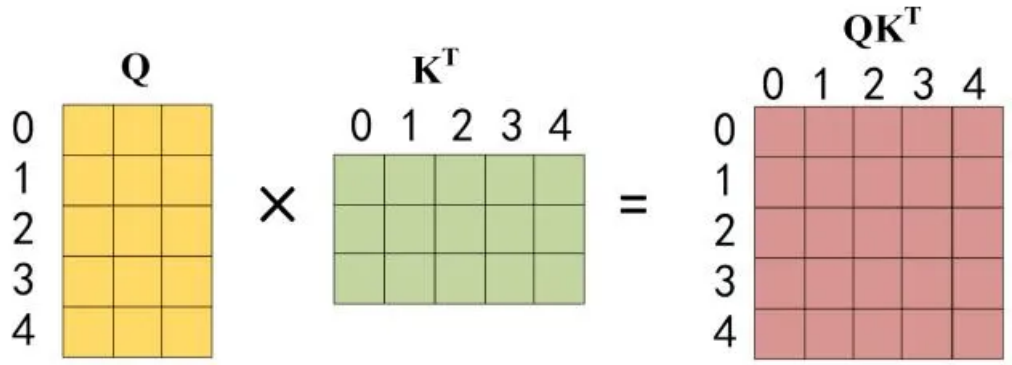
在第一个Multi-Head Attention中，QKV都是来源是decoder输入

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 5 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（每行是一个单词，列是不同的维度），**Mask** 矩阵是一个 5×5 的矩阵，黄色表示遮挡的部分取值为 $-inf$ (负无穷)，绿色部分取值为1。



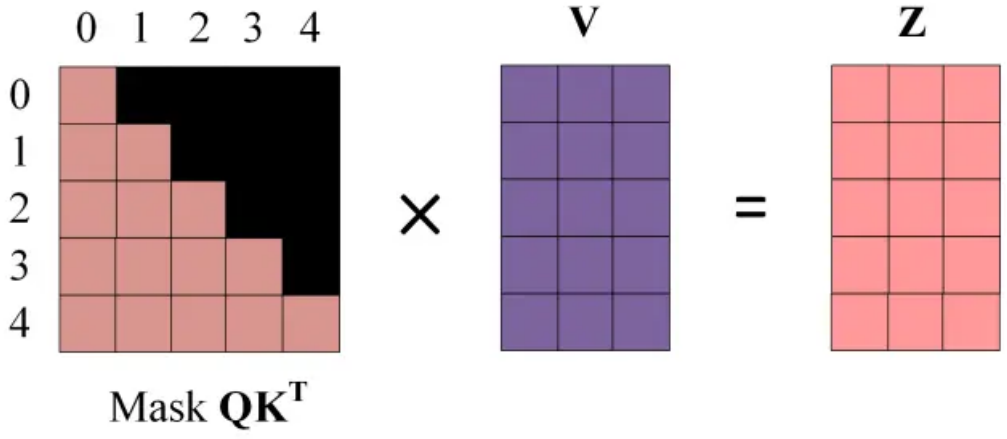
第二步：接下来的操作和之前的 Self-Attention 一样，通过输入矩阵**X**计算得到**Q,K,V**矩阵。然后计算**Q**和 K^T 的乘积 QK^T 。



第三步：在得到 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：

得到 **Mask** QK^T 之后在 **Mask** 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。相当于对于单词0，只能看到自身的信息，看不到单词1,2,3,4对它的信息。这一步也可以用加法实现，若用加法，则绿色部分值为0，黄色部分依然是负无穷。

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



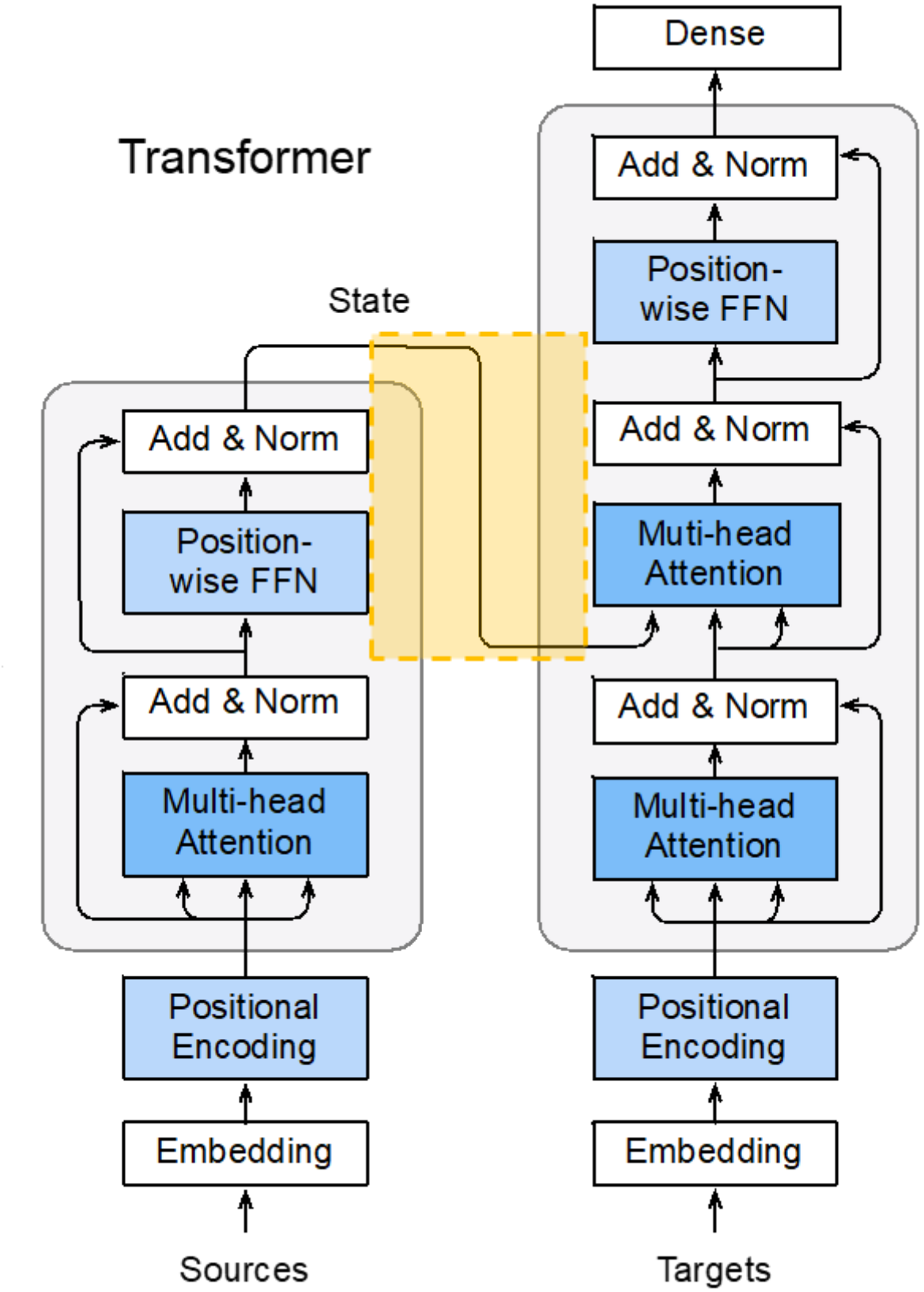
第五步：通过上述步骤就可以得到一个 Mask Self-Attention 的输出矩阵 Z_i ，然后和 Encoder 类似，通过 Multi-Head Attention 拼接多个输出 Z_i 然后计算得到第一个 Multi-Head Attention 的输出**Z**，**Z**与输入**X**维度一样。

以上就是Masked的过程，Attention Mask和Padding Mask都通过上述的Mask矩阵完成。

2.9.1.2. 第二个Multi-Head Attention

Q是来源于上一个Attention（来源于decoder的输入），K,V来源于encoder

- Decoder block 第二个Multi-Head Attention 变化不大，主要的区别在于其中Self-Attention 的K,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C计算的。decoder的
- 根据Encoder 的输出C计算得到K,V，根据上一个Decoder block 的输出Z计算Q(如果是第一个Decoder block 则使用输入矩阵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)。



编码器中最后一个encoder的输出，将其作为解码器中所有attention层的第 2个（不是第一个，第一个是mask multi head）多头注意力head的key和value，它的query来自目标序列

2.10. 损失和正则化

在Transformer的原文有提到：

"我们在训练期间采用三种类型的正则化:

残差Dropout 我们对每个子层的输出应用dropout[27],然后将其添加到子层输入并进行归一化。此外,我们对编码器和解码器堆栈中的嵌入和位置编码的和应用dropout。对于基础模型,我们使用率$P_{drop} = 0.1$。

标签平滑 在训练期间,我们采用值为$\epsilon_{ls} = 0.1$的标签平滑[30]。这会损害困惑度,因为模型学会变得更不确定,但提高了准确率和BLEU分数。"

（1）

交叉熵损失用于分类任务，衡量预测分布与目标分布的差异，公式为：

$$\text{CrossEntropy}(P, Q) = -\sum_{i} P(i) \log(Q(i))$$

其中：

- $P(i)$ 是目标分布（通常为 one-hot 编码的分布）。
- $Q(i)$ 是模型预测的分布（通常通过 softmax 输出的概率分布）。

(2)KL散度损失公式

KL散度衡量两个概率分布之间的差异，公式为:

$$D_{KL}(P \| Q) = \sum_{i} P(i) \log \left(\frac{P(i)}{Q(i)} \right)$$

其中:

- $P(i)$ 是目标分布 (真实分布)。
- $Q(i)$ 是模型预测的分布。
- KL散度是非对称的，用来衡量预测分布 Q 偏离真实分布 P 的程度。

实际上：

$$\text{CrossEntropy}(P, Q) = H(P, Q) = H(P) + D_{KL}(P \| Q)$$

一般来说交叉熵要求的是one-hot的概率分布，而KL散度则不需要。当我们使用log-softmax以及使用标签平滑的时候，就可以使用KL散度

（3）标签平滑

标签平滑会将目标从严格的 one-hot 编码调整为一个平滑分布，平滑后的真实分布 P 表示为:

$$P(i) = \begin{cases} 1 - \text{ smoothing}, & \text{对于正确类别}, \\ \frac{\text{smoothing}}{K-1}, & \text{对于其他类别}, \end{cases}$$

其中 K 是类别数。由于目标分布不再是 one-hot 编码，KL 散度更适合这种情况，因为它度量的是两个任意概率分布之间的距离。

2.11. 优化器

在训练的初期设置一个较低的学习率，并通过一个（warmup）阶段逐步增加学习率，随后随着训练步数的增加，逐渐减小学习率。

此外可以看到学习率模型大小相关，模型的维度越大，学习率越小。

使用Adam优化器,其中$\beta_1 = 0.9, \beta_2 = 0.98$和$\epsilon = 10^{-9}$，优化公式如下

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

其中， $\hat{m}_t$ 和$\hat{v}_t$分别是修正的一阶和二阶矩估计，η是学习率，ϵ是很小的数，用于防止分母为零。

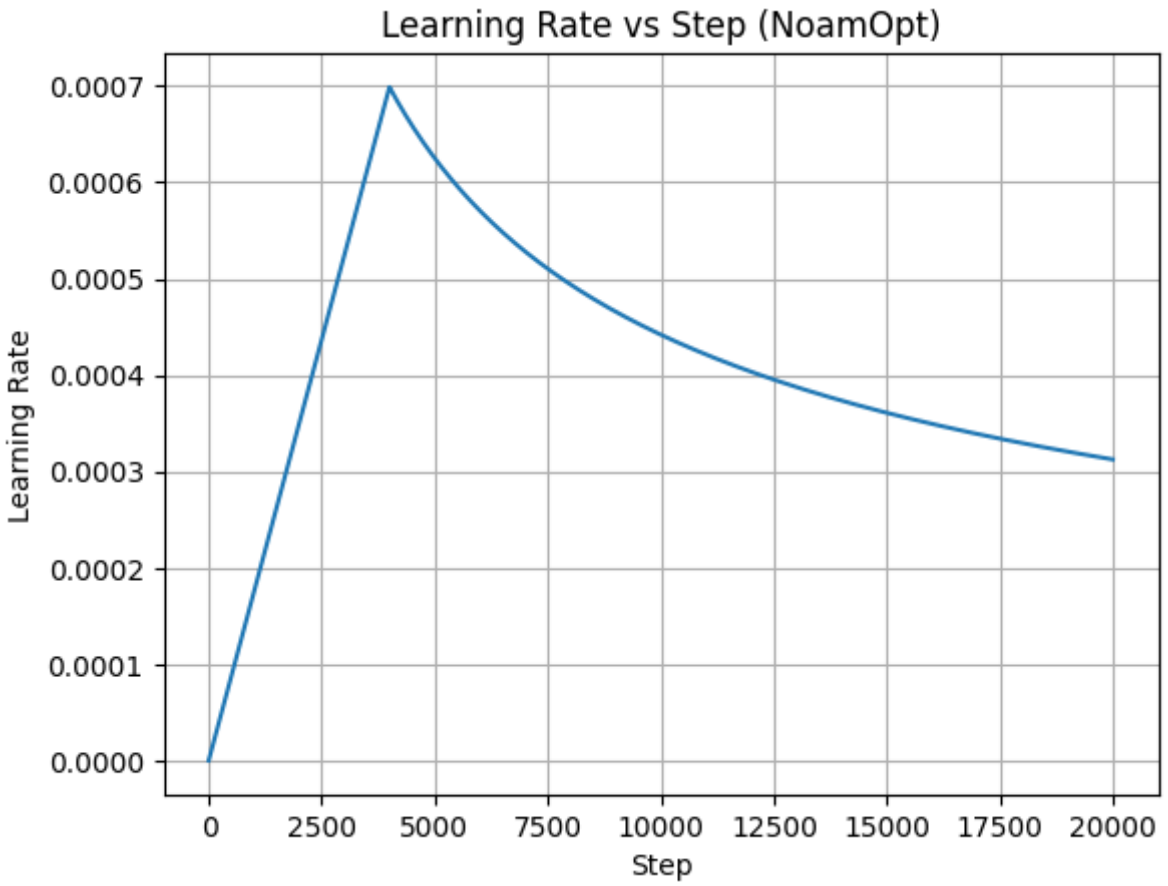
在训练过程中根据以下公式调整学习率:

$$\mathrm{lr}(\text{ step }) = \text{ factor } \times \left(\text{ model\_size }^{-0.5} \times \min\left(\text{ step }^{-0.5}, \text{ step } \times \text{ warmup }^{-1.5}\right)\right)$$

公式包含两部分：

1. $\mathrm{model\_size}^{-0.5}$：这部分与模型的规模相关，模型的维度越大，学习率越小。它确保模型规模较大时，学习率保持相对较小，以控制参数更新的幅度。
2. 学习率调度部分：$\min(step^{-0.5}, step \times \mathrm{warmup}^{-1.5})$
   - $step \times \mathrm{warmup}^{-1.5}$：在前 `warmup` （4000）步中，学习率逐步增大，在warmup步后，转到$step^{-0.5}$
   - $step^{-0.5}$：随着训练步数的增加，学习率逐渐减小。

学习率变化曲线：



代码:

```
import numpy as np
import matplotlib.pyplot as plt

# NoamOpt class as defined
class NoamOpt:
    def __init__(self, model_size, factor, warmup):
        self.model_size = model_size
        self.factor = factor
        self.warmup = warmup

    def rate(self, step):
        return self.factor * (self.model_size ** (-0.5) * min(step ** (-0.5),
step * self.warmup ** (-1.5)))

# Instantiate the NoamOpt optimizer with model_size=512, factor=1, warmup=4000
opt = NoamOpt(512, 1, 4000)

# Generate learning rates for steps from 1 to 20000
steps = np.arange(1, 20001)
learning_rates = [opt.rate(i) for i in steps]

# Plotting the learning rate curve
plt.plot(steps, learning_rates)
plt.xlabel('Step')
plt.ylabel('Learning Rate')
plt.title('Learning Rate vs Step (NoamOpt)')
plt.grid(True)
plt.show()
```

2.12. 训练与推理

在训练过程中，Decoder其实是一次性接收完整的目标句子，并利用mask attention机制来保证每一步预测只依赖于之前的单词。这与推理（inference）时逐步生成翻译结果有所不同。

具体来说：

1. **训练阶段**：
    - **输入与目标**：在训练时，Encoder是一次性获得完整的输入句子"我有一只猫"。整个句子会被转换为词嵌入（Embeddings），然后输入到Encoder层中进行处理。Decoder同样会接收整个目标句子，例如"I have a cat"，作为输入。假设这个目标句子在训练数据集中已经存在。
    - **Mask Attention**：虽然Decoder一次性接收了整个目标句子，但在计算注意力时会使用mask attention，确保每个位置上的单词只关注它之前的单词，而不会看到它之后的单词。例如，对于目标句子的每个位置，mask attention会使得第一个位置只能看到`<begin>`，第二个位置只能看到`<begin>`和`I`，第三个位置只能看到`<begin> I have`，依此类推。
    - **损失计算**：模型在每个时间步都会预测下一个单词，然后将这些预测结果与实际目标进行对比，计算损失（例如交叉熵损失）。训练目标是最小化这个损失。
2. **推理阶段**：
    - **逐步生成**：在推理时（即生成翻译结果时），Decoder不会一次性获得整个目标句子。相反，它会逐步生成翻译结果。在每一步，Decoder只会接收到已经生成的部分句子，并预测下一个单词。生成过程如下：
        - 第一步输入：`<begin>`
        - 第二步输入：`<begin> I`
        - 第三步输入：`<begin> I have`
        - 以此类推，直到生成结束标记（）。

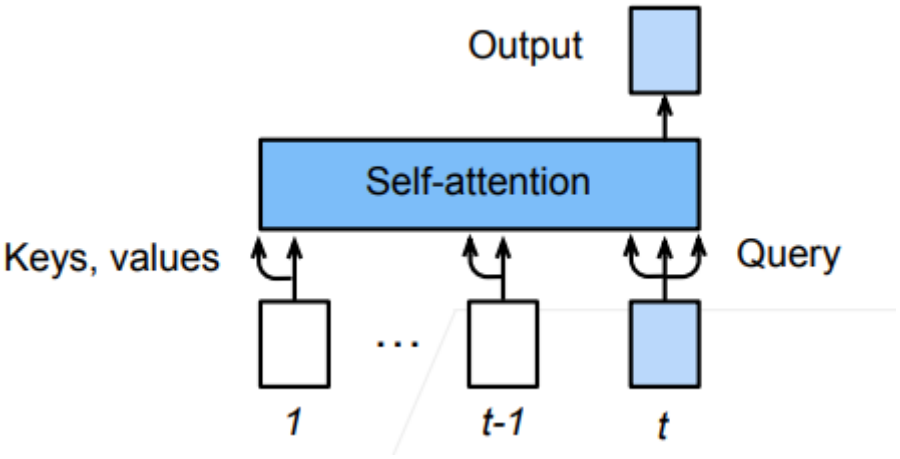
这种逐步生成的过程确保了翻译结果的连贯性和正确性，因为每一步生成的单词都会影响后续的预测。

总结：

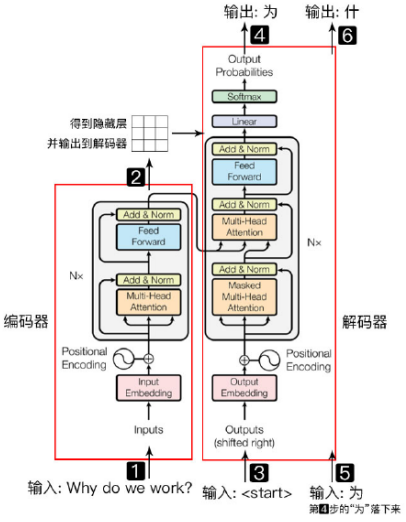
- **训练中**，Decoder一次性接收完整的目标句子，并通过mask attention机制保证每一步预测只依赖之前的单词。
- **推理中**，Decoder逐步生成翻译结果，每一步依赖已经生成的部分句子和Encoder的编码结果。

这样设计的目的是为了在训练中更高效地计算损失和梯度，同时在推理中生成连贯且符合语法的翻译结果。

例子：



- 输入自然语言序列到编码器: Why do we work?(为什么要工作)；
- 编码器输出的隐藏层，再输入到解码器；
- 输入 <start> (起始)符号到解码器；
- 解码器得到第一个字"为"；
- 将得到的第一个字"为"落下来再输入到解码器；
- 解码器得到第二个字"什"；
- 将得到的第二字再落下来，直到解码器输出 <end> (终止符)，即序列生成完成。



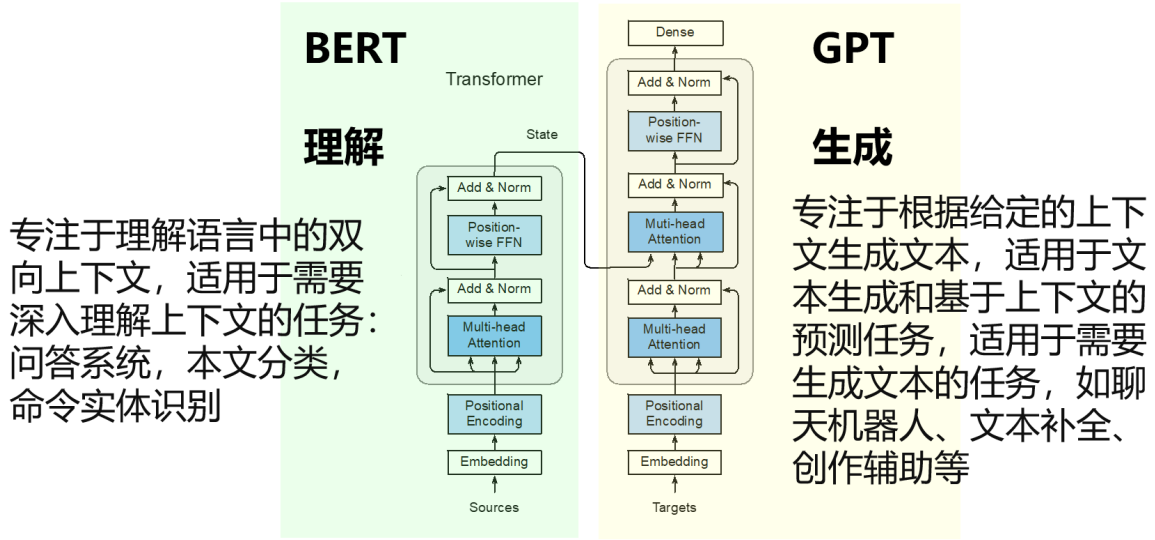
2.13. 代码理解

配套代码：transformer_step项目文件夹

3. Bert

BERT(Bidirectional Encoder Representations from Transformers (BERT): A sentiment analysis odyssey，2020)：

BERT希望可以像CV一样，可以基于微调的NLP模型做迁移学习。希望预训练的模型抽取了足够多的信息，而面对新的任务只需要增加一个简单的输出层



BERT 架构

- 只有编码器的 Transformer
- 两个版本:
  - Base: #blocks = 12, hidden size $= 768$, #heads $= 12$, #parameters $= 110\mathrm{M}$
  - Large: #blocks = 24, hidden size = 1024, #heads = 16, #parameter $= 340\mathrm{M}$
- 在大规模数据上训练 >：超过3B 词

BERT的概述

- BERT针对微调设计
- 基于Transformer的编码器做了如下修改
  - 模型更大, 训练数据更多
  - 输入句子对, 片段嵌入, 可学习的位置编码
  - 训练时使用两个任务:
    - 带掩码的语言模型
    - 下一个句子预测

3.1. BERT的创新：

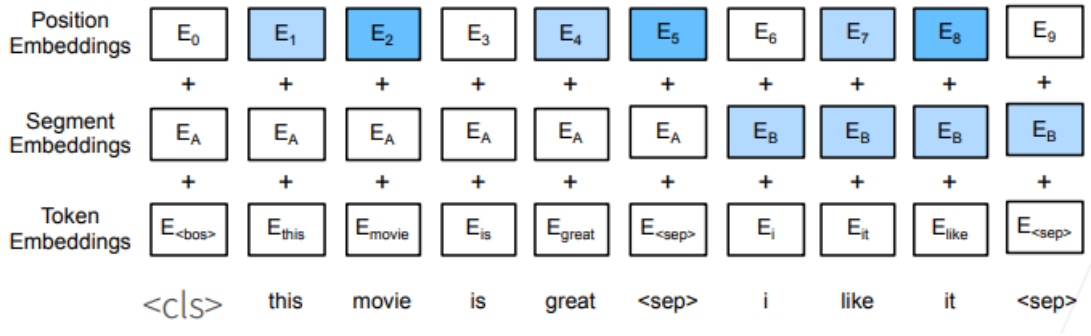
3.1.1. 对输入的改变

BERT模型的输入设计是非常关键的一部分。对于输入的修改，它采用了一种独特的方式来处理句子，bert中的每个样本不是1个句子，而是一个句子对，例如两个句子合并。

不同的任务构造句子对的方式略有不同，例如下一句预测任务是从语料库中随机选择两个连续的句子A和B，50%的概率保持它们的原始顺序，50%的概率将B替换为语料库中的随机句子；而问答任务将问题作为第一个句子，将包含答案的文本段落作为第二个句子。

以下是详细介绍：

首先，BERT的输入包含以下几个部分：



1. Token Embeddings：这是输入句子中单词的标记化表示。这些单词首先被转换为标记，然后这些标记被转换为向量。BERT使用了WordPiece嵌入，包含30000个token的词汇表。
2. Segment Embeddings：BERT可以接受一对句子作为输入，因此需要一种方法来区分两个句子。这就是Segment Embeddings的角色。例如，对于句子对"A, B"，在句子A中的所有词都会接收"A"的标记，而在句子B中的所有词则会接收"B"的标记。segment embedding的维度和词embedding的维度是一样的，同一句话不同token的segment embedding是一样。但是同一个token，不同维度的segment embedding是不一样的。segment embedding是一个可以学习的向量。
3. Position Embeddings：BERT使用Transformer模型，后者不自然地包含序列顺序信息。因此，BERT通过添加位置嵌入向量来包含这种信息。这确保模型可以考虑单词的顺序。这个位置编码是可学习的，不再是encoding了。

接下来是如何使用特殊标记处理句子：

- 每个输入序列的开始都会有一个特殊的[CLS]标记（Classify的缩写）。在进行分类任务时，这个标记的最终隐藏状态被用作整个序列的汇总表示。也就是说，[CLS]不仅标记了句子的开始，还用于提供一个全局的上下文表示。
- 对于每一对输入句子，我们将它们使用[SEP]标记（Separator的缩写）拼接在一起。例如，如果我们有两个句子“A”和“B”，我们将它们处理为"[CLS] A [SEP] B [SEP]"。

例如，如果我们的任务是理解两个句子"A dog is not a cat." 和 "Cats are great pets."之间的关系，BERT的输入就会看起来像这样："[CLS] A dog is not a cat. [SEP] Cats are great pets. [SEP]"。

值得注意的是在BERT中，Token Embeddings、Segment Embeddings和Position Embeddings是直接相加的，然后这个和（总的嵌入向量）作为模型的输入进入Transformer的Encoder。所以embedding后的输入X是（批量大小，最大序列长度，num_hiddens）。

BERT模型通过这种方式设计输入，可以有效地处理两个句子，并且可以全方位地理解单词在它们所处上下文的语义。这是其强大的语言理解能力的关键因素之一。

3.1.2. 预训练任务1：带掩码的语言模型（Masked Language Model，MLM）

预训练任务 1：带掩码的语言模型

Transfomer的编码器是双向, BERT是编码器因此也是双向的。而BERT想作为一个通用的预训练模型，但是最通用的语言模型（给一个词预测下一个词）确实单向的。因此BERT使用了一个带掩码的语言模型，在带掩码的语言模型中，每次随机 (15%概率) 将一些词元换成 < mask>，BERT就是去预测< mask>的这些词，这就等于是做完形填空了。

但需要注意的是，因为微调任务中不出现 < mask >，因此在预训练中选中的 (15%概率) 单词不会全部变为< mask >，而是：

- 80%概率下, 将选中的词元变成
- 10%概率下换成一个随机词元（从词表随机抽取）
- 10%概率下保持原有的词元（即保持不变）

对于那些被选中但并没有被替换为[MASK]的单词，也就是那10%被替换为任何其他随机单词和10%没有改变的单词，BERT依然会尝试去预测它们（这个预训练任务就是去预测15%的掩码单词（不管是否被替换））。这个预测不是像解码器一样逐个预测，而是通过捕捉上下文信息的单词表示，一次性处理整个序列进行预测，。具体来说，BERT会为输入中的每个单词生成一个向量表示，这个表示取决于该单词的上下文（也就是周围的单词）。然后，对于那些被选中进行预测的单词，BERT会使用一个特殊的线性分类层（通常接在Transformer的最后一层上）来预测每个单词的原始形式。对于那些没有替换为[mask]的单词来说，虽然它们并没有被替换为[MASK]，但在预测阶段，模型并不知道哪些单词被替换，哪些单词没有。换句话说，模型始终会尝试去预测那些被选中的15%的单词，不管它们是否真的被替换为[MASK]，还是被替换为其他单词，或者根本没有改变。

> 在训练过程中，BERT会记录下哪15%的词元被选中。对于这15%的词元，无论它们在输入中是什么样子（[MASK]、随机词或原词），模型都会尝试预测它们的原始值。在计算损失时，我们只考虑这15%位置的预测结果，而忽略其他85%位置的预测。

这样，模型在预训练期间就会有一部分时间预测非掩码的单词，这有助于缓解预训练和微调阶段的不匹配问题。至于微调阶段，虽然输入数据不含有[MASK]标记，但是BERT的强大之处在于，它能够通过对上下文的深入理解，对不同的下游任务进行微调。这些任务可以包括文本分类、情感分析、命名实体识别等。在这些任务中，整个句子或句子对（没有[MASK]标记）会被送入模型，并通过调整顶部的分类器层以适应特定任务。具体的微调过程依赖于具体的任务类型。比如，如果任务是文本分类，那么我们可以把BERT作为一个特征提取器，将整个文本输入到BERT模型中，然后用最后一个token的输出（即[CLS]标记对应的输出）来进行分类。如果任务是命名实体识别或者其他的序列标注任务，我们可以使用每个token对应的输出向量进行分类。

在微调阶段，所有的模型参数（包括BERT预训练的参数和顶部分类层的参数）都会被更新以最小化任务的损失函数。这个损失函数通常与特定任务相关，比如文本分类任务通常使用交叉熵损失，序列标注任务则可能使用条件随机场（CRF）等。

如下解释可以详见代码部分：

1.注意的是，MaskLM就是利用经过BERT编码器的输出，在被掩码（mask）位置上的输出值，去预测掩码位置原始输入的词元。换句话说，BERT编码器对输入（包括掩码部分）进行了编码，然后我们从这个编码中提取出掩码位置的编码，将这些编码作为MaskLM的输入，用于预测掩码位置的原始输入词元。

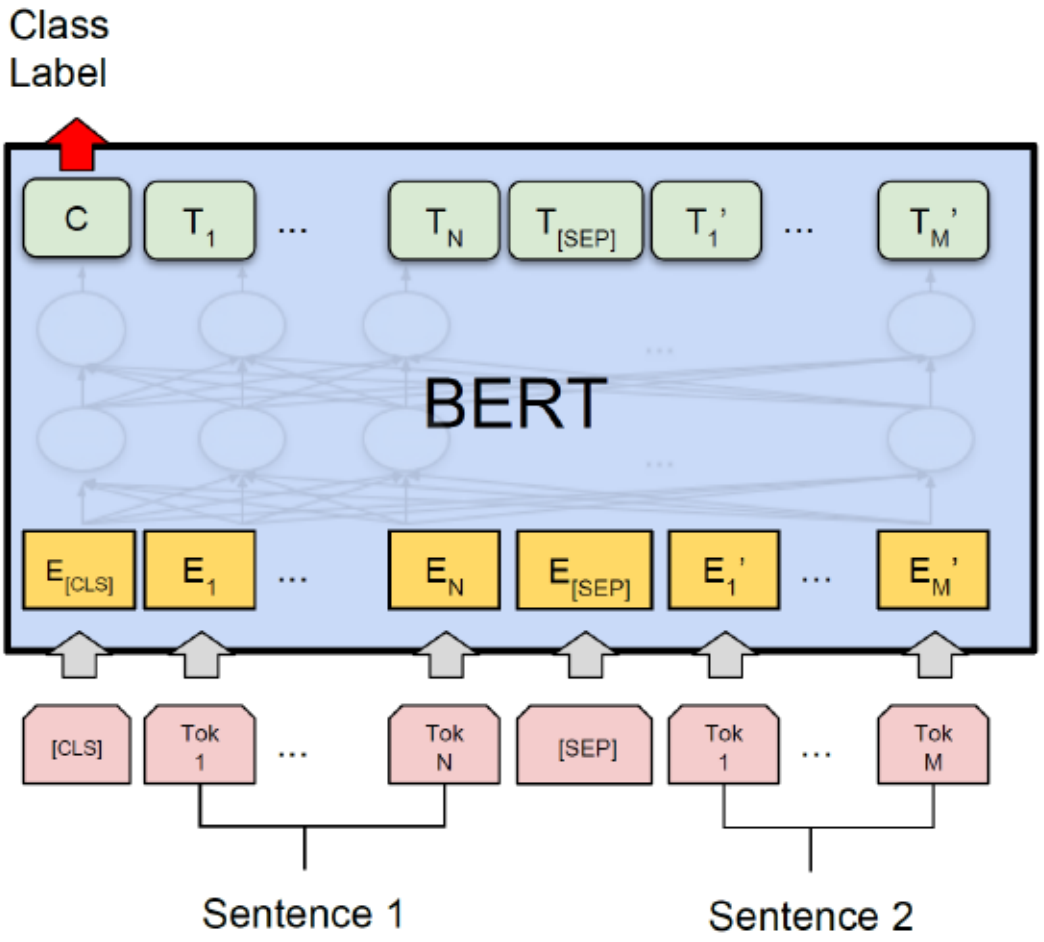
2.mlm_Y作为真实值的话，指的是被掩码位置的原始输入词元对应的标签。在计算损失函数时，我们将MaskLM对掩码位置的预测mlm_Y_hat与原始输入的标签mlm_Y进行比较。这个标签是指词元在词汇表中的索引，所以它是一个整数，而不是经过embedding处理或者encoder处理后的值。

3.1.3. 预训练任务2：下一个句子预测

预训练任务2：下一句子预测

- 预测一个句子对中两个句子是不是相邻
- 训练样本中:

- 50\%概率选择相邻句子对： this movie is great i like it
- 50\%概率选择随机句子对： this movie is great hello world ＜ sep ＞
- 将对应的输出放到一个全连接层来预测。在实际预测中，X会经过encoder处理，只将第一个元素对应的编码输出取出来，再经过一个全连接层，进行一个二分类预测（虽然只是拿cls位置的输出，但是因为我们做的是attention，实际上可以理解为这1个位置的输出是一个向量（因为有多个隐藏节点），这个向量是cls这个位置抽取了整个句子的信息整合而成）



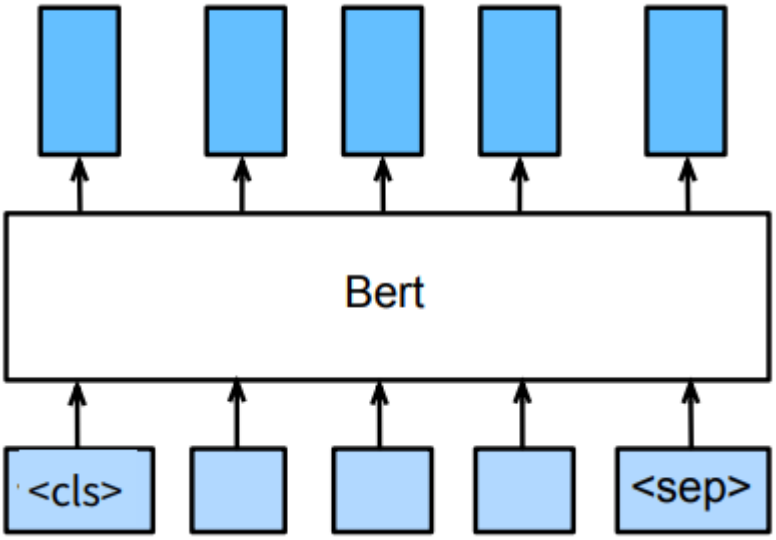
3.1.4. 两个任务的总结

在BERT的预训练中，MLM和NSP两个任务是并行进行的，这意味着同一对句子样本会同时被用于两个任务。这个句子样本可能是从一个连续的文本段落中随机选取的两个连续的句子，也可能是从两个不同段落中随机选取的两个句子。

- 对于MLM任务，模型并不关心两个句子是否连续或相关，它只需要预测被掩蔽的词元。这意味着无论这两个句子是否连续，MLM任务都可以进行。
- 对于NSP任务，如果两个句子是连续的，那么这个句子样本的标签就是“是”（表示第二个句子是第一个句子的下一句）。如果两个句子不是连续的，那么标签就是“否”。

换句话说，在预训练阶段，我们可以在同一对句子上进行MLM任务和NSP任务，而这两个句子可能是连续的，也可能不是连续的。并且，无论进行哪个任务，这对句子都可能存在被掩蔽的词元。因此，在MLM任务中，有可能使用的句子对在NSP任务中是不匹配的（即第二个句子并不是第一个句子的下一句），反之亦然。在NSP任务中，输入的句子对也可能存在被掩蔽的词元。

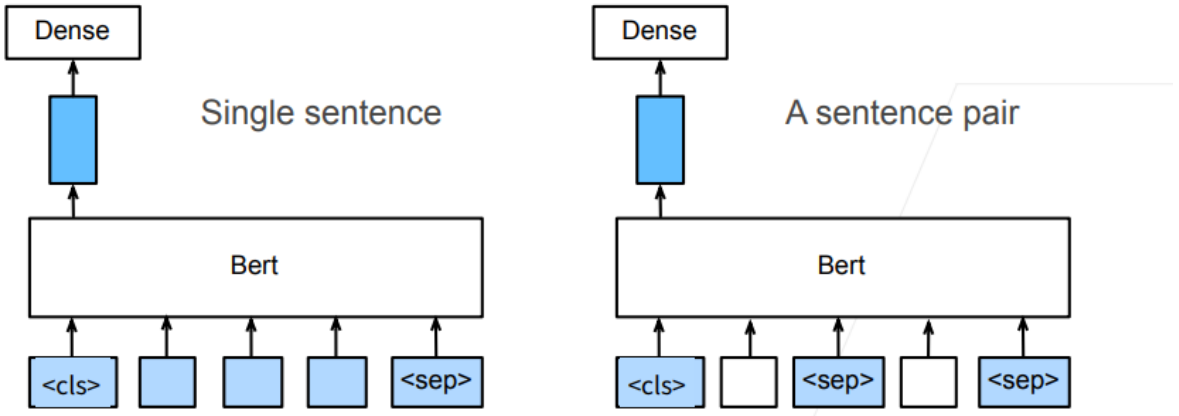
3.2. Bert的微调与下游任务



BERT 对每一个词元返回抽取了上下文信息的特征向量，在接下里的不同的任务钟使用不同的特征去完成对应的任务

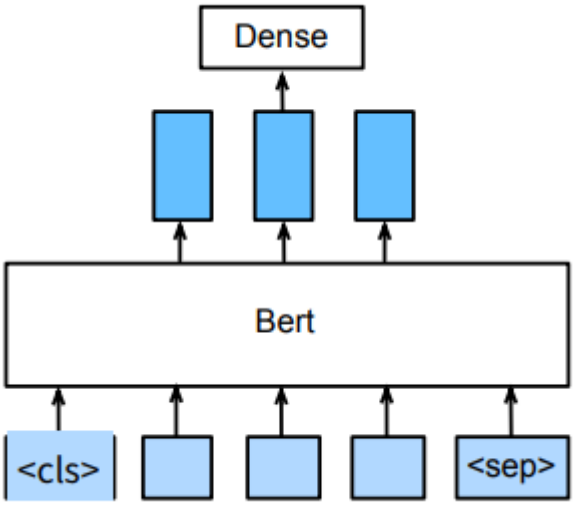
- 即使下游任务各有不同, 使用BERT微调时均只需要增加输出层（例如全连接的MLP）
- 但根据任务的不同, 输入的表示, 和使用的BERT特征也会 不一样

3.2.1. 句子分类



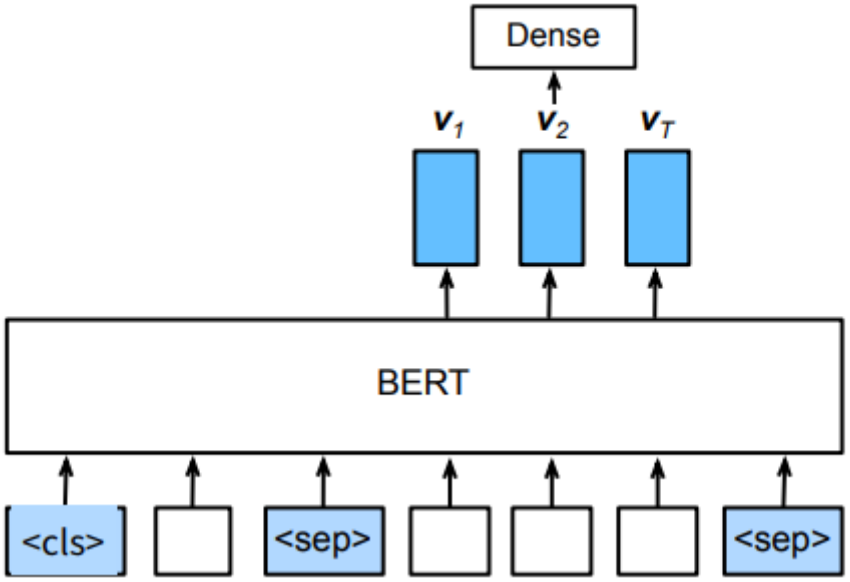
将 对应的向量输入到全连接层分类。选择cls主要是因为我们在下一个句子的预测任务中就是使用cls。当然我们也可以使用其他位置的向量，毕竟后续我们还要做微调。

3.2.2. 命名实体识别



- 识别一个词元是不是命名实体, 例如人名、机构、位置
- 将非特殊词元放进全连接层分类

3.2.3. 问题回答



- 给定一个问题, 和描述文字, 找出一个片段作为回答；上面的图中，前面是问题，后面是回答
- 对片段中的每个词元预测它是不是回答的开头、结束或其他

3.2.4. fine tuning细节

一般fine tuning的时候可以冻住底部的一些层（远离分类器），不过一般来说，全部不固定效果更好

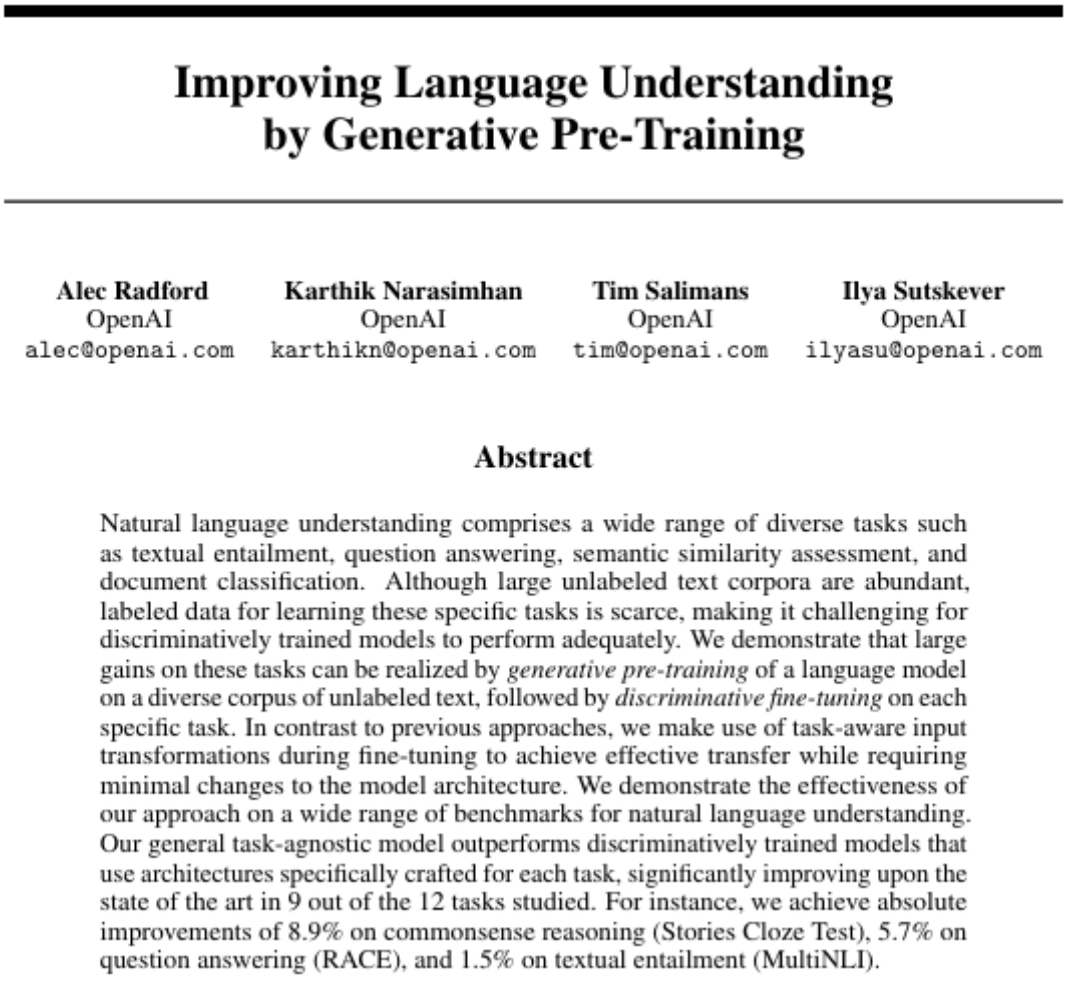
4. GPT

GPT1和2是开源的，3之后就不开源了

GPT2：去掉了“Extract”这类特殊的token，因为在预训练中没有使用过，影响下游任务效果，改为将下游任务（例如翻译）直接使用自然语言发出指令。发现了参数量对效果有决定性影响
GPT3：更多的参数（1750亿），通过ICL（上下文学习）借助one shot，few shot实现更好的效果
GPT3.5：应用了RLHF (Reinforcement Learning from Human Feedback)
GPT4：有更大的参数量？支持图片生成与解读，具有更高准确率
GPT4v：多模态版本，支持图文，视频，语音等多种媒介混合生成

4.1. GPT1

Improving Language Understanding by Generative Pre-Training

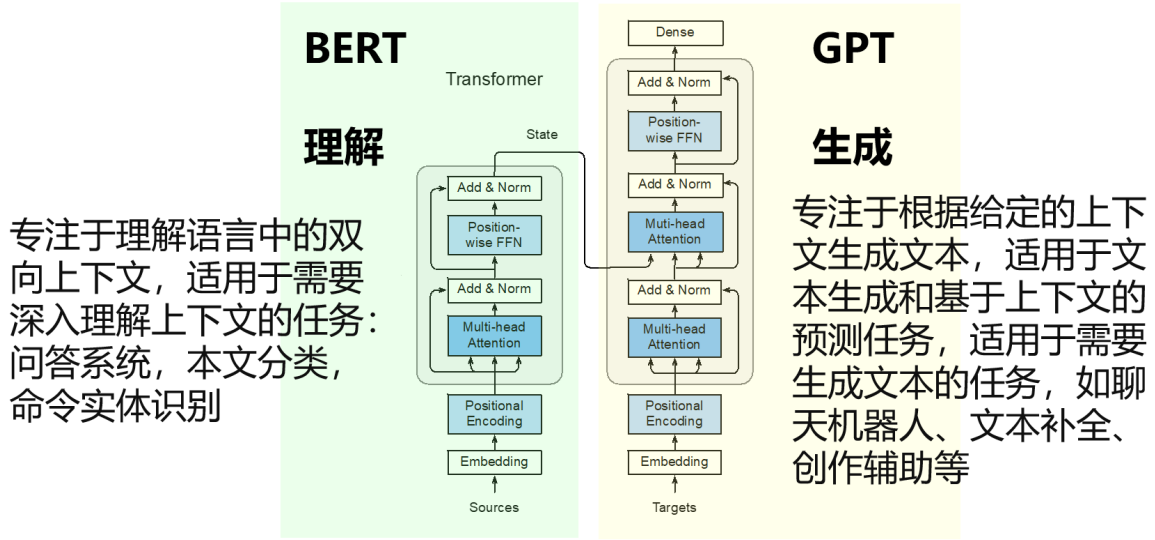


Improving Language Understanding by Generative Pre-Training

Alec Radford OpenAI alec@openai.com
Karthik Narasimhan OpenAI karthikn@openai.com
Tim Salimans OpenAI tim@openai.com
Ilya Sutskever OpenAI ilyasu@openai.com

Abstract

Natural language understanding comprises a wide range of diverse tasks such as textual entailment, question answering, semantic similarity assessment, and document classification. Although large unlabeled text corpora are abundant, labeled data for learning these specific tasks is scarce, making it challenging for discriminatively trained models to perform adequately. We demonstrate that large gains on these tasks can be realized by *generative pre-training* of a language model on a diverse corpus of unlabeled text, followed by *discriminative fine-tuning* on each specific task. In contrast to previous approaches, we make use of task-aware input transformations during fine-tuning to achieve effective transfer while requiring minimal changes to the model architecture. We demonstrate the effectiveness of our approach on a wide range of benchmarks for natural language understanding. Our general task-agnostic model outperforms discriminatively trained models that use architectures specifically crafted for each task, significantly improving upon the state of the art in 9 out of the 12 tasks studied. For instance, we achieve absolute improvements of 8.9% on commonsense reasoning (Stories Cloze Test), 5.7% on question answering (RACE), and 1.5% on textual entailment (MultiNLI).



GPT实际上就是Transformer的解码器结构（不再使用编码器）

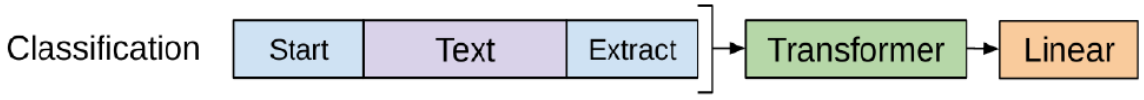
GPT的预训练就很简单。在训练数据中有大量的小说文章。 那就给出前面的一段文字，经过attention层后，会得到一个矩阵，有一个矩阵输出，使用这个输出的矩阵里面的最后一行，在经过一个线性层以后，会得到一个向量。那这个向量就是它预测的结果。就是每一个单词分别出现的概率有多高。

例如，我们有一篇文章，这篇文章会被分割成多个token序列。而在具体的划分中，我们可以采用固定的长度的划分，可以使用带重叠的滑动窗口划分，或者基于句子的自然边界划分。

值得注意的是，这种自监督学习的方式不需要标注的。你拿这个自然形成的序，因为小说也好，文章也好，它这个文字都是天然会产生一个顺序。那你就把前面的单词输进去，按顺序输进去。那通过这个向向量来预测下一个单词是什么? 那如果预测的单词不对，它就会产生误差，于是我们就可以对这个模型进行训练了，那通过学习了数以十亿计这种语句以后，那我们就把这个gpt预训练好了。

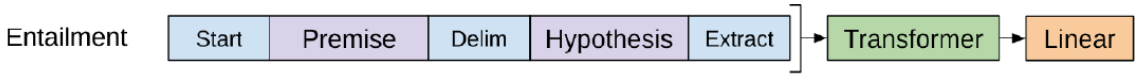
GPT1可以有多个微调的下游任务：

（1）分类



- 分类任务 (例如情感识别，好评差评等)
- 直接在预训练好的GPT上做，增加一个线性层
- 技巧是在文本前后加入start和extract标签，在transformer最后一层中拿走对应extract的token，经过线性层，softmax，然后得到输出的标号 (训练方法)

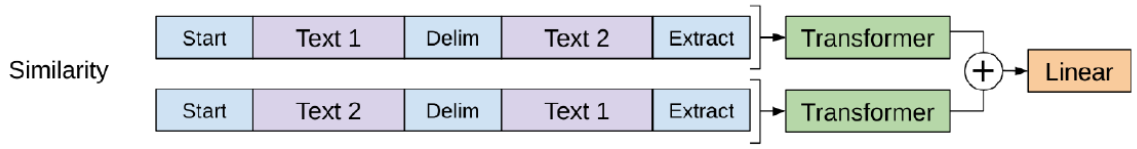
（2）蕴含判别



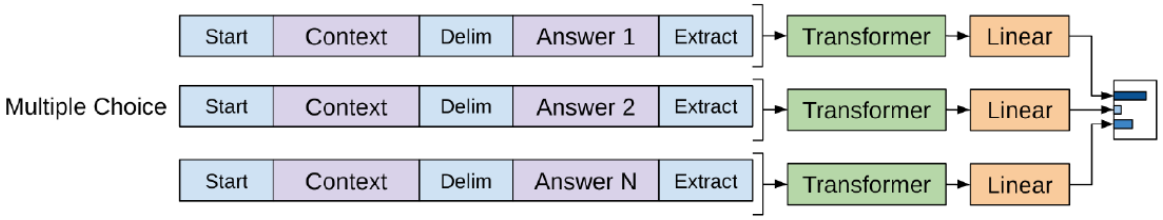
在文本中是否蕴含给出的假设

（3）相似性判别

把两个句子合并判断句子是否相似。注意的是会把两个句子按照不同顺序都进入模型，确保不受顺序影响



（4）多项选择



4.1.1. 训练细节

数据集：Bookscorpus数据集。

原始的 BooksCorpus是一个由未出版作者撰写的免费 小说集，其中包含 11038 本书(约 7400 万个句子，1G 个单词)，共有 16 个不 同的子类型(如浪漫、历史、冒险等)。由于原始的 BooksCorpus 数据作者已 不再提供下载，Shawn Presser 在 2020 年整理发布了一套替代数据集。

GPT1中的Transformer超参数，12层，768维，12头自注意力（与 BERT BASE相同），参数1.1亿（Transformer的原始超参数为6层，512维，8头）

4.2. GPT2

Language Models are Unsupervised Multitask Learners

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] Jeffrey Wu [*1] Rewon Child [1] David Luan [1] Dario Amodei [**1] Ilya Sutskever [**1]

Abstract

Natural language processing tasks, such as question answering, machine translation, reading comprehension, and summarization, are typically approached with supervised learning on task-specific datasets. We demonstrate that language models begin to learn these tasks without any explicit supervision when trained on a new dataset of millions of webpages called WebText. When conditioned on a document plus questions, the answers generated by the language model reach 55 F1 on the CoQA dataset - matching or exceeding the performance of 3 out of 4 baseline systems without using the 127,000+ training examples. The capacity of the language model is essential to the success of zero-shot task transfer and increasing it improves performance in a log-linear fashion across tasks. Our largest model, GPT-2, is a 1.5B parameter Transformer that achieves state of the art results on 7 out of 8 tested language modeling datasets in a zero-shot setting but still underfits WebText. Samples from the model reflect these improvements and contain coherent paragraphs of text. These findings suggest a promising path towards building language processing systems which learn to perform tasks from their naturally occurring demonstrations.

competent generalists. We would like to move towards more general systems which can perform many tasks – eventually without the need to manually create and label a training dataset for each one.

The dominant approach to creating ML systems is to collect a dataset of training examples demonstrating correct behavior for a desired task, train a system to imitate these behaviors, and then test its performance on independent and identically distributed (IID) held-out examples. This has served well to make progress on narrow experts. But the often erratic behavior of captioning models (Lake et al., 2017), reading comprehension systems (Jia & Liang, 2017), and image classifiers (Alcorn et al., 2018) on the diversity and variety of possible inputs highlights some of the shortcomings of this approach.

Our suspicion is that the prevalence of single task training on single domain datasets is a major contributor to the lack of generalization observed in current systems. Progress towards robust systems with current architectures is likely to require training and measuring performance on a wide range of domains and tasks. Recently, several benchmarks have been proposed such as GLUE (Wang et al., 2018) and decaNLP (McCann et al., 2018) to begin studying this.

Multitask learning (Caruana, 1997) is a promising framework for improving general performance. However, multitask training in NLP is still nascent. Recent work reports modest performance improvements (Yogatama et al.,

模型与GPT-1相同，同样使用了transformer的解码器，但是使用了更多的参数 (15亿参数)和更大的数据量更大 (WebText数据集，百万级别的网页)

与之前不同的是，不再分各项下游任务了，而是直接希望做成zero-shot和多任务学习。在下游任务中不加入GPT1中的 "extract" "start" 等特殊符号，因为在预训练时模型没学习过这些符号，将使模型感到困惑。我们希望下游任务训练时，模型看到的数据和预训练时是一样的。以翻译任务为例，我们不加入特殊符号，直接把翻译指令，例如"translate" 嵌入到构造的训练数据中

数据集：使用了Common crawl数据集，这是 一个公共贡献项目，有TB级别数据，由全世界各地的贡献者写爬虫爬取，放到AWS

考虑到数据质量不高，利用reddit（新闻聚合网页，贡献新闻，读者投票），选取评价高的（4500万页面，800万文本，40GB数据 ）

经过对数据分析，发现就机器翻译任务而言，有很多与构思思路相若的样例

GPT2有四个尺寸大小的模型，分别是：

是的，GPT-2（Generative Pre-trained Transformer 2）有四个不同尺寸的模型，这些模型的大小主要由参数数量决定。具体来说，这四个尺寸分别是：

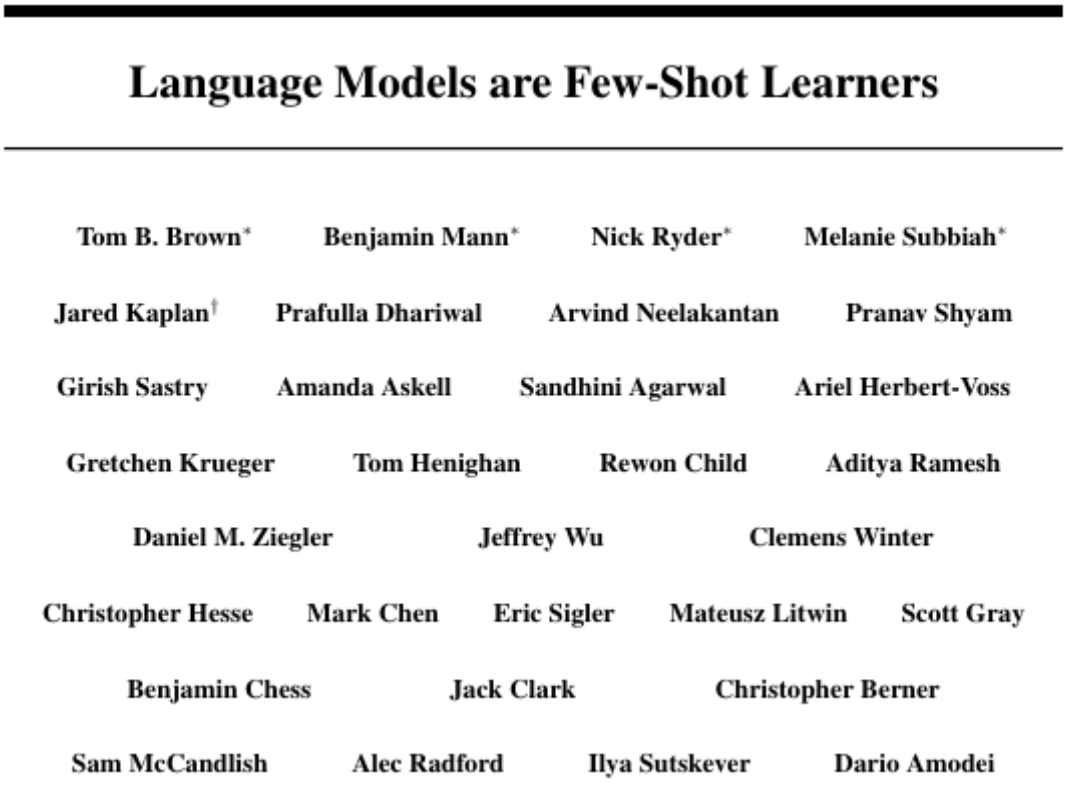
1. **GPT-2 Small**:
    - **参数数量**：约1.17亿（117M）个参数

- **层数**：12层
- **注意力头数**：12个
- **隐藏层大小**：768个维度

2. **GPT-2 Medium**:
   - **参数数量**：约3.45亿（345M）个参数
   - **层数**：24层
   - **注意力头数**：16个
   - **隐藏层大小**：1024个维度
3. **GPT-2 Large**:
   - **参数数量**：约7.74亿（774M）个参数
   - **层数**：36层
   - **注意力头数**：20个
   - **隐藏层大小**：1280个维度
4. **GPT-2 Extra Large (GPT-2 XL)**:
   - **参数数量**：约15亿（1.5B）个参数
   - **层数**：48层
   - **注意力头数**：25个
   - **隐藏层大小**：1600个维度

4.3. GPT3

Language Models are Few-Shot Learners



Language Models are Few-Shot Learners

Tom B. Brown* Benjamin Mann* Nick Ryder* Melanie Subbiah*

Jared Kaplan† Prafulla Dhariwal Arvind Neelakantan Pranav Shyam

Girish Sastry Amanda Askell Sandhini Agarwal Ariel Herbert-Voss

Gretchen Krueger Tom Henighan Rewon Child Aditya Ramesh

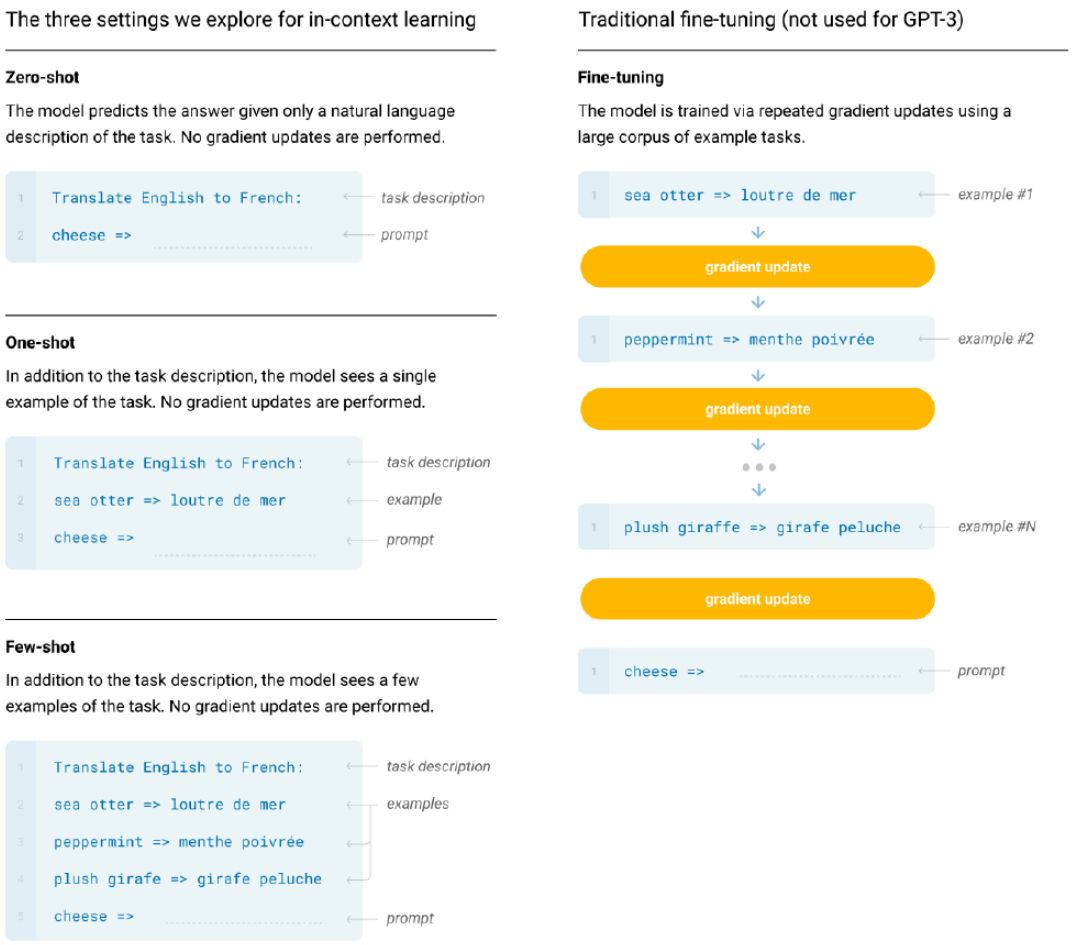
Daniel M. Ziegler Jeffrey Wu Clemens Winter

Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray

Benjamin Chess Jack Clark Christopher Berner

Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei

其实这是一个技术报告，而不是杂志文章，特别长，一共63页。GPT3的目标还是解决GPT-2有效性差的问题，不再追求zero shot，改为one shot，few shot，提出meta learning, in-context learning。



GPT3说，即使只使用左边的one-shot和few-shot，也能取得像右边fine-tuning的效果

数据集：

把CommonCrawl的数据作为负样本。GPT-2获取的reddit高质量数据作为正样本，构造logistic分类器，判别文本质量。使用去重模型。过滤各个数据集

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

模型规模：

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}