Attention Is All You Need

Ashish Vaswani[*]
Google Brain
avaswani@google.com

Noam Shazeer[*]
Google Brain
noam@google.com

Niki Parmar[*]
Google Research
nikip@google.com

Jakob Uszkoreit[*]
Google Research
usz@google.com

Llion Jones[*]
Google Research
llion@google.com

Aidan N. Gomez[*] [†]
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[*] [‡]
illia.polosukhin@gmail.com

汇报人：程征

Motivation

现在主流的序列模型都是基于复杂的循环神经网络或者卷积神经网络构造出来的Encoder-Decoder模型，并且就算是目前性能最好的序列模型也是基于注意力机制下的Encoder-Decoder架构。然而，由于传统的Encoder-Decoder架构在建模过程中，下一个时刻的计算过程会依赖上一个时刻的输出，而这种固有的属性就限制了传统的Encoder-Decoder模型就不能以并行的方式进行计算。尽管最新的研究工作已经使得传统的循环神经网络在计算效率上有了很大的提升，但是本质的问题依旧没有得到解决。

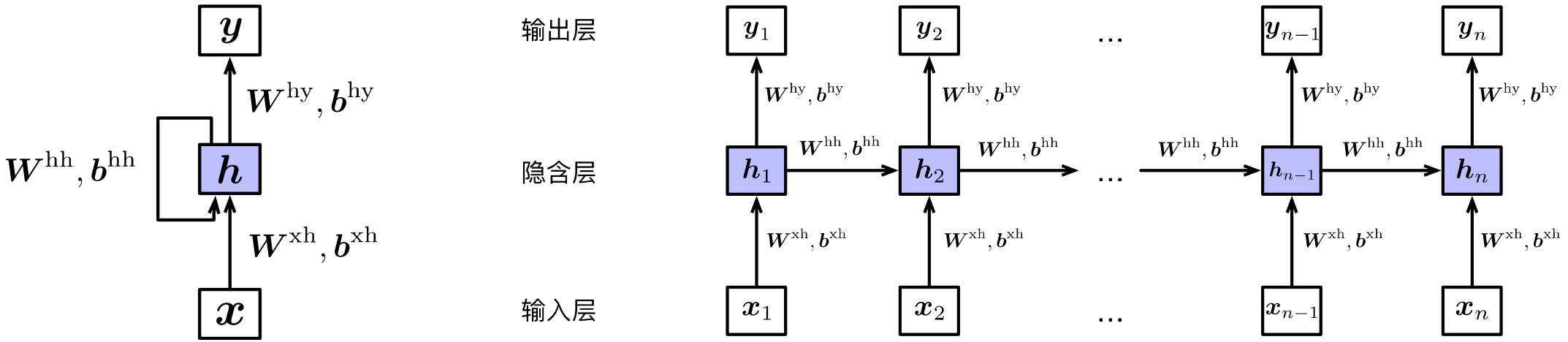


图1 循环神经网络示意图

Solution

在这篇论文中，作者首次提出一种全新的Transformer架构来解决这一问题，模型结构如图2所示。Transformer架构的优点在于其完全摒弃了传统的循环结构，仅通过注意力机制来计算模型输入与输出的隐含表示，而这种注意力机制的名字就是自注意力机制（self-attention）

所谓自注意力机制，就是通过某种运算来直接计算句子在编码过程中每个位置上的注意力权重，然后再以权重和的形式来计算得到整个句子的隐含向量表示。Transformer架构就是基于这种自注意力机制而构建的Encoder-Decoder模型。

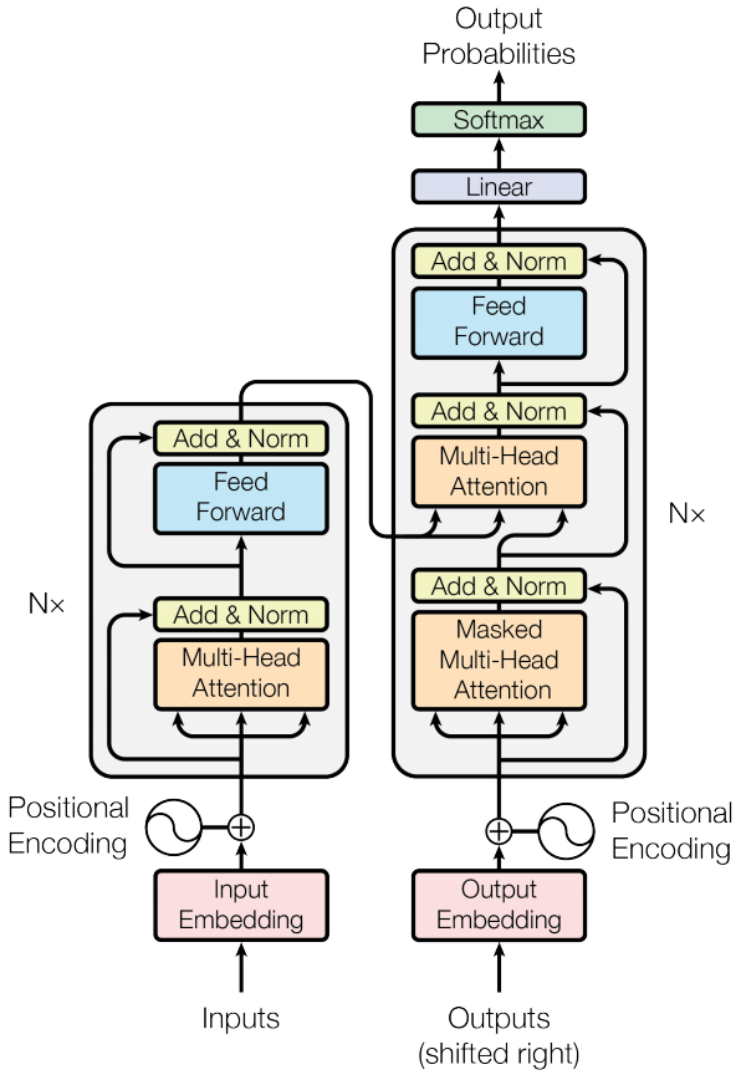


图2 Transformer模型结构图

What is Attention ?

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

注意力机制可以描述为将query和一系列key-value对映射到某个输出的过程，而这个输出的向量就是根据query和key计算得到的权重作用于value上的权重和。

Scaled Dot-Product Attention

自注意力机制，即论文中描述的Scaled Dot-Product Attention，其核心计算过程就是通过Q和 K^T 计算得到注意力权重；然后在作用于V得到整个权重和输出。其计算公式为：

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \qquad (1)$$

其中，Q、K和V分别为3个矩阵，且维度分别为d_q, d_k, d_v（从后面计算过程可以发现$d_q = d_v$），而$\sqrt{d_k}$就是图3中的Scale过程。

之所以要进行缩放是因为通过实验作者发现，对于较大的d_k来说，在完成QK^T后将会得到很大的值，而这将导致在经过$softmax$操作后产生非常小的梯度，不利用网络训练。

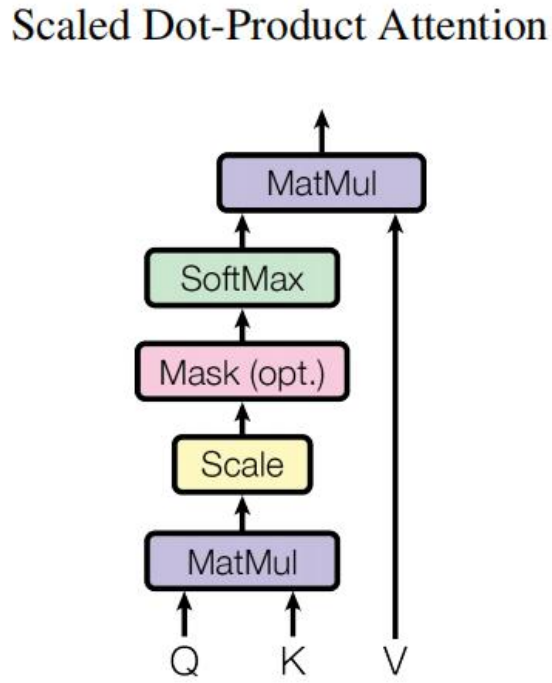
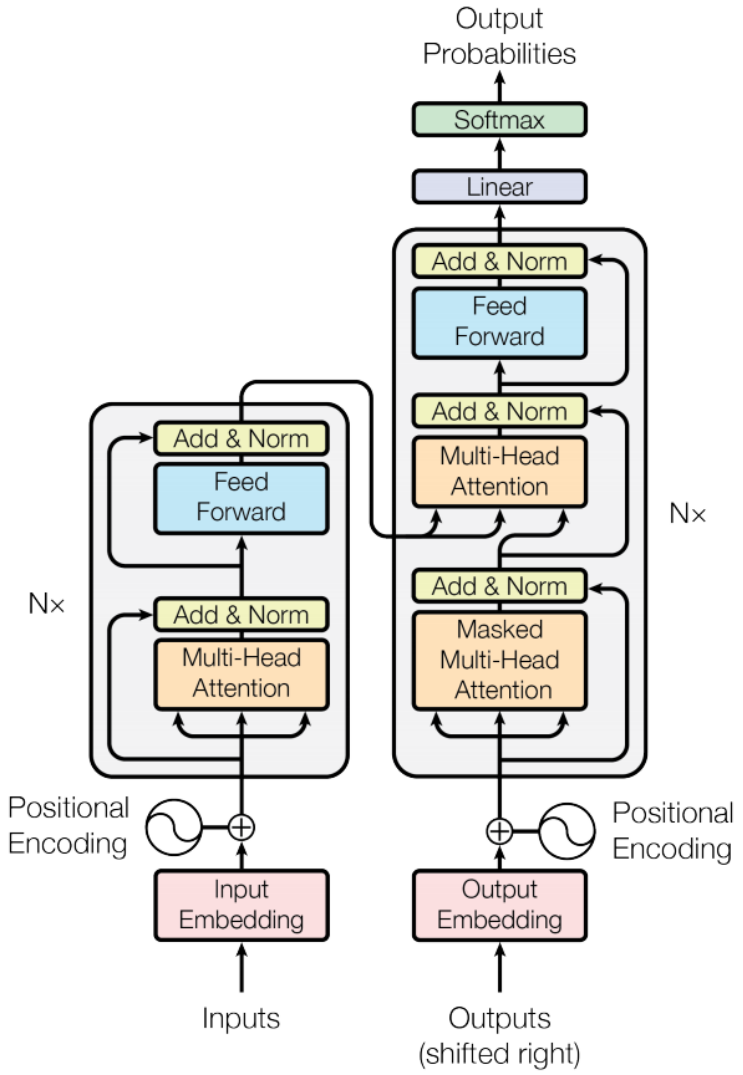


图3 自注意力机制

How to obtained Q K V ?

根据图2Transformer结构图可以发现，共有三处用到了自注意力机制，分别是Encoder、Decoder以及Encoder和Decoder的交互层。首先，我们先来看一下Encoder和Decoder的部分自注意力机制的Q、K、V是如何计算得来的。



How to obtained Q K V ?

假设输入序列为“我 是 谁”，首先通过将其映射为指定维度（假定维度为4）的低维稠密的词向量，那么通过以下过程计算便可得到Q、K以及V。

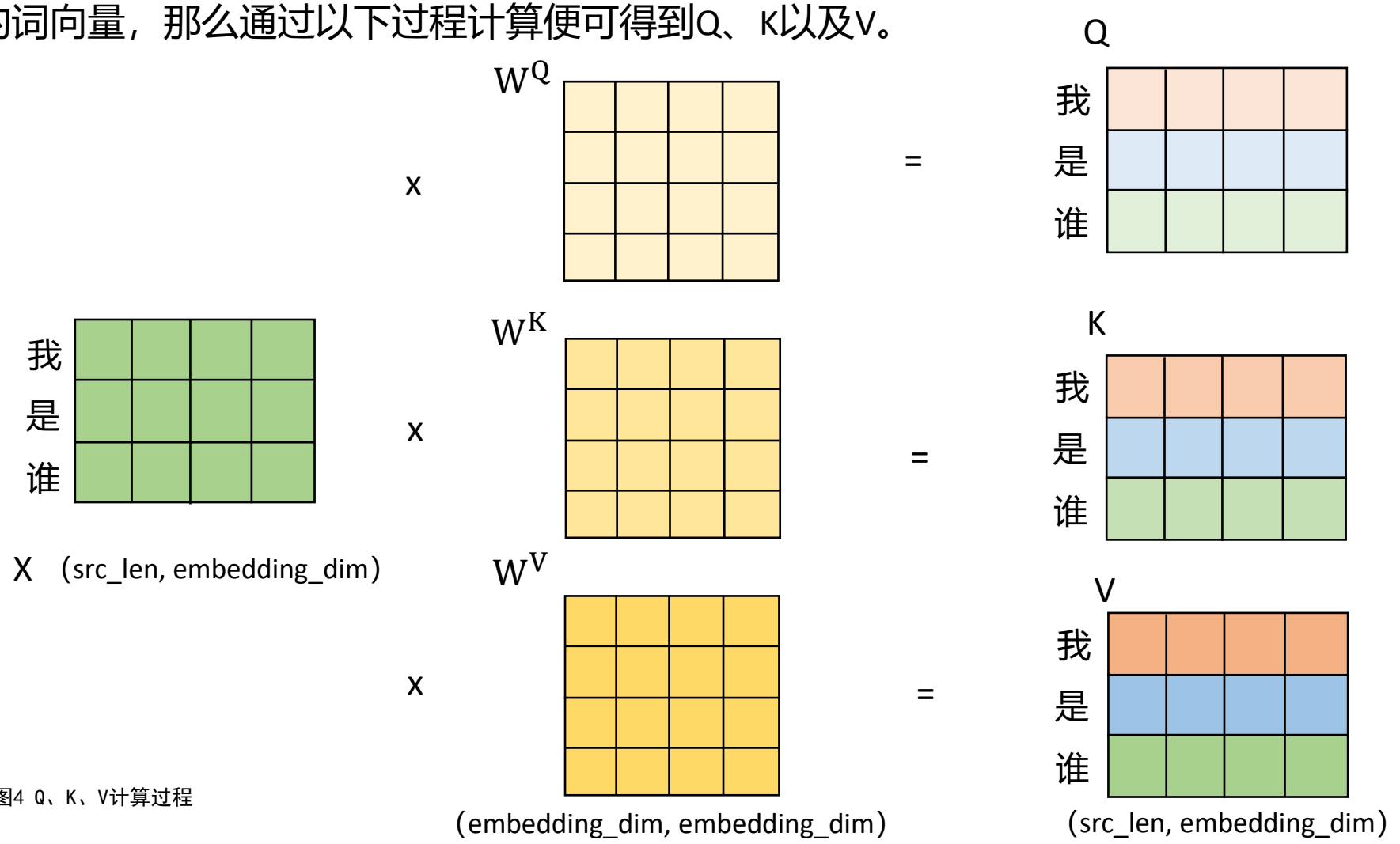


图4 Q、K、V计算过程

The Calculate Process of Self Attention

从以上计算过程可以看出，Q、K和V其实就是输出序列X分别乘以3个不同的矩阵计算而来，可以理解为这是对于同一个输入进行3次不同的线性变换来表示其3种不同的状态。在计算Q、K、V之后，就可以进一步计算得到权重向量，计算过程如图5所指示：

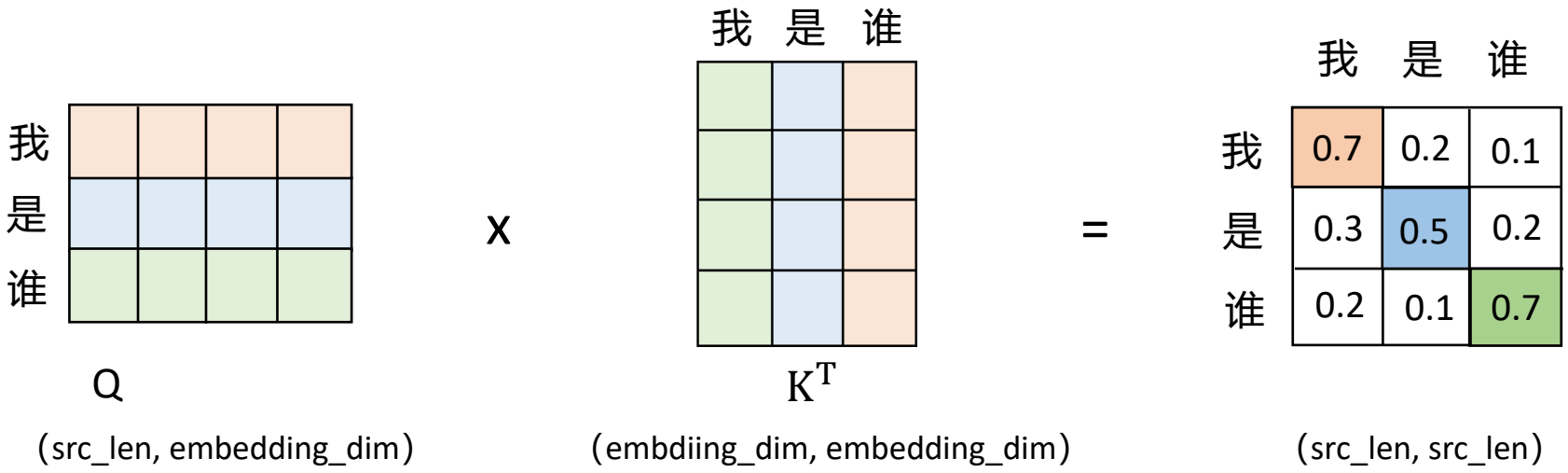


图5 注意力权重计算过程

The Calculate Process of Self Attention

对于权重矩阵的第一行来说，0.7表示“我”与“我”的注意力值；0.2表示“我”与“是”的注意力值；0.1表示“我”与“谁”的注意值。换句话说，在对序列中的“我”进行编码时，应将0.7的注意力放在“我”上，将0.2的注意力放在“是”上，将0.1的注意力放在“谁”上，剩下同理。从这一过程可以看出，通过这个权重矩阵就能轻松观察到在编码对应位置上的向量时，应该以何种方式将注意力集中在不同的位置上。

不过从上面的计算结果还可以看到一点就是，模型在对当前位置的信息进行编码时，会过度将注意力集中于自身的位置而忽略了其他位置。因此，作者采取的一种解决方案就是采用多头注意力机制（MultiHeadAttention）。（这点我们稍后介绍）

The Calculate Process of Self Attention

在通过图5的计算过程得到权重矩阵后，便可以将其作用于V，进而得到最终的编码输出，计算过程如图6所示：

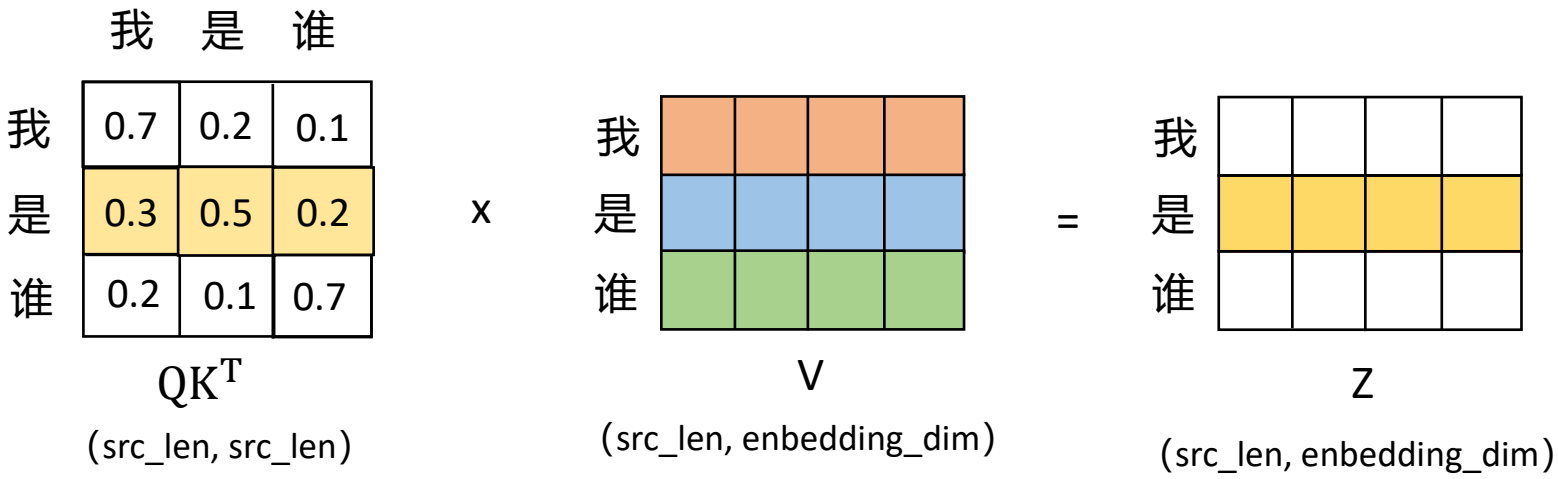


图6 权重和编码输出

z即经过编码后最终的输出向量。

The Calculate Process of Self Attention

对于上述过程，我们还可以换个角度来理解，如图7所示。

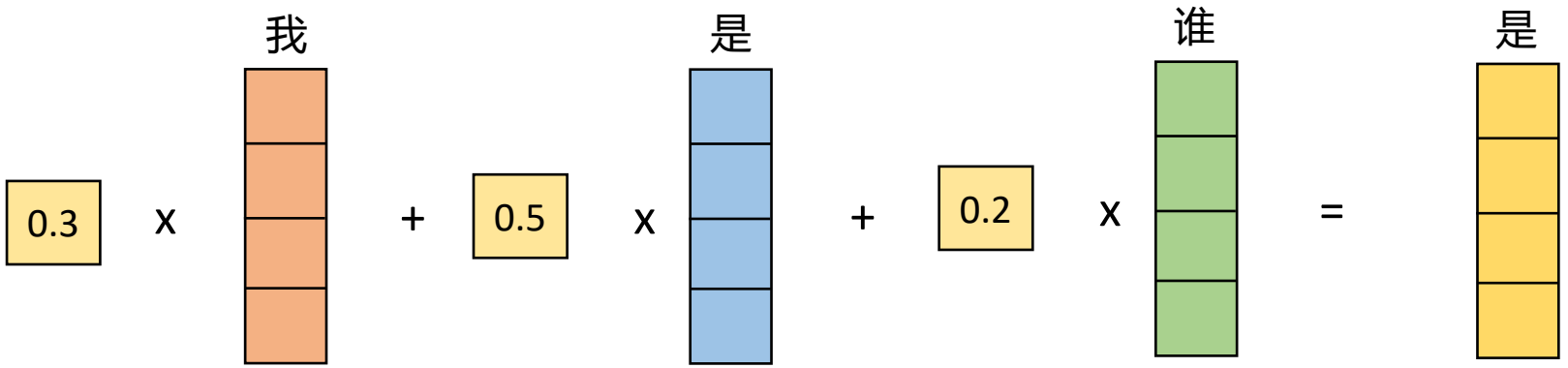


图7 编码输出计算图

对于最终输出“是”的编码向量来说，其实就是原始“我 是 谁”3个向量的加权和，而这也就体现了在对“是”进行编码时注意力权重分配的全过程。

The Calculate Process of Self Attention

整个自注意力机制计算的全过程还可有图8进行表示：

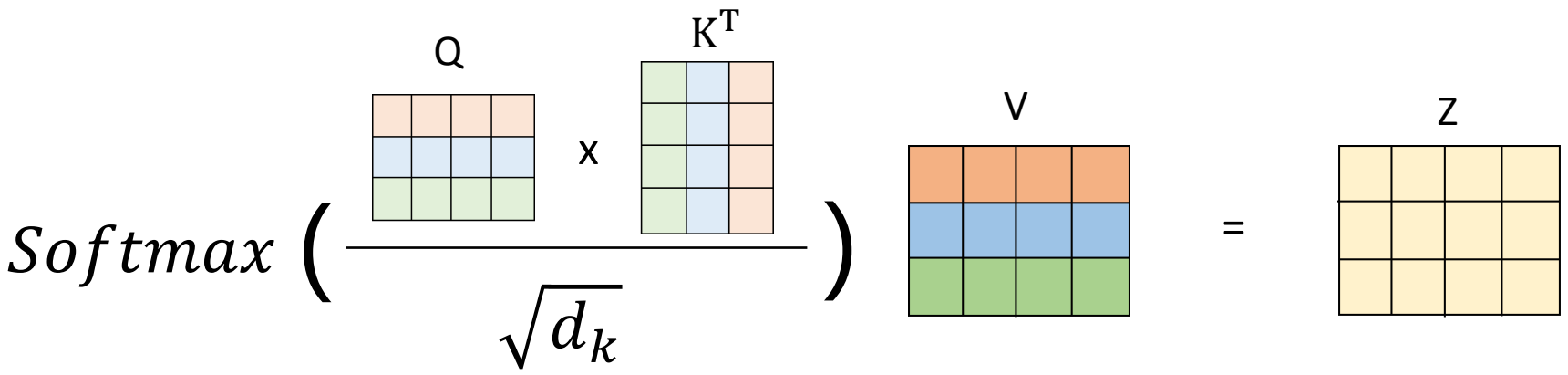


图8 自注意力机制计算过程

可以看出，通过这种自注意力机制的方式确实解决了作者在论文伊始所提出的“传统序列模型在编码过程中都需要顺序进行的弊端”的问题，有了自注意力机制，仅仅只需要对原始编码进行几次矩阵变换便能够得到最终包含有不同位置注意力信息的编码向量。

Why Multi-Head Attention is Needed?

如在上面的介绍中提到，模型在对当前位置的信息进行编码时，会过度的将注意力集中于自身的位置，因此作者提出了通过多头注意力机制来解决这一问题。同时，使用多头注意力机制还能够给予注意力层的输出包含有不同子空间的编码表示信息，从而增强模型的表达能力。

Multi-Head Attention

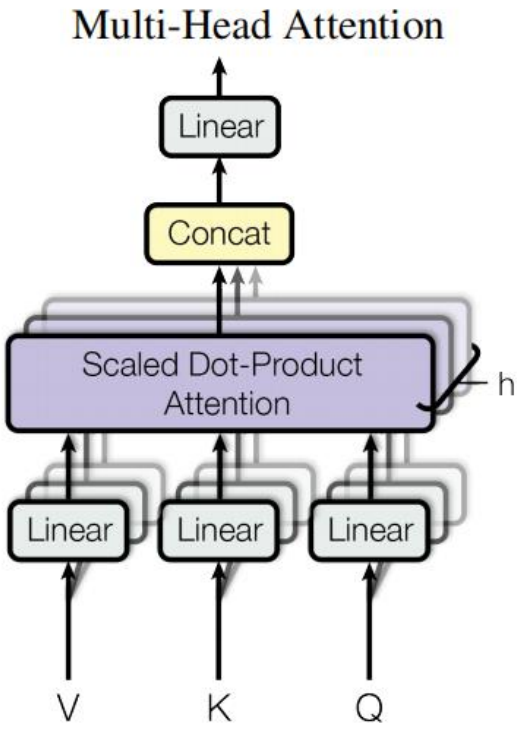


图9 多头自注意力机制结构

所谓多头注意力机制，其实就是将输入序列进行多组的自注意力处理过程，然后再将每一组自注意力机制计算的结果拼接起来进行一次线性变换得到最终的输出结果。计算公式如下:

$$MultiHead(Q,K,V) = Concat(head_1,\dots,head_h)W^o$$

$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \quad (2)$$

其中，$W_i^Q \in R^{d_{model}\times d_k}$, $W_i^K \in R^{d_{model}\times d_k}$, $W_i^V \in R^{d_{model}\times d_v}$, $W^O \in R^{hd_v\times d_{model}}$

需要注意的是，此处的Q、K和V与上文（图4）中的Q、K、V并不是一回事，这里的Q、K、V要经过一次线性变换操作后的结果才分别是上文中的Q、K、V。

Multi-Head Attention

在论文中，作者使用了$h = 8$个并行的自注意力模块（8个头）来构建一个注意力层，并且对于每个自注意力模块都限定了$d_k = d_v = \frac{d_{model}}{h} = 64$。从这里其实可以发现，论文中所使用的多头注意力机制其实就是将一个大的高维单头拆分成了h个多头。因此，整个多头注意力机制的计算过程我们可以通过图10的过程进行表示。

Multi-Head Attention

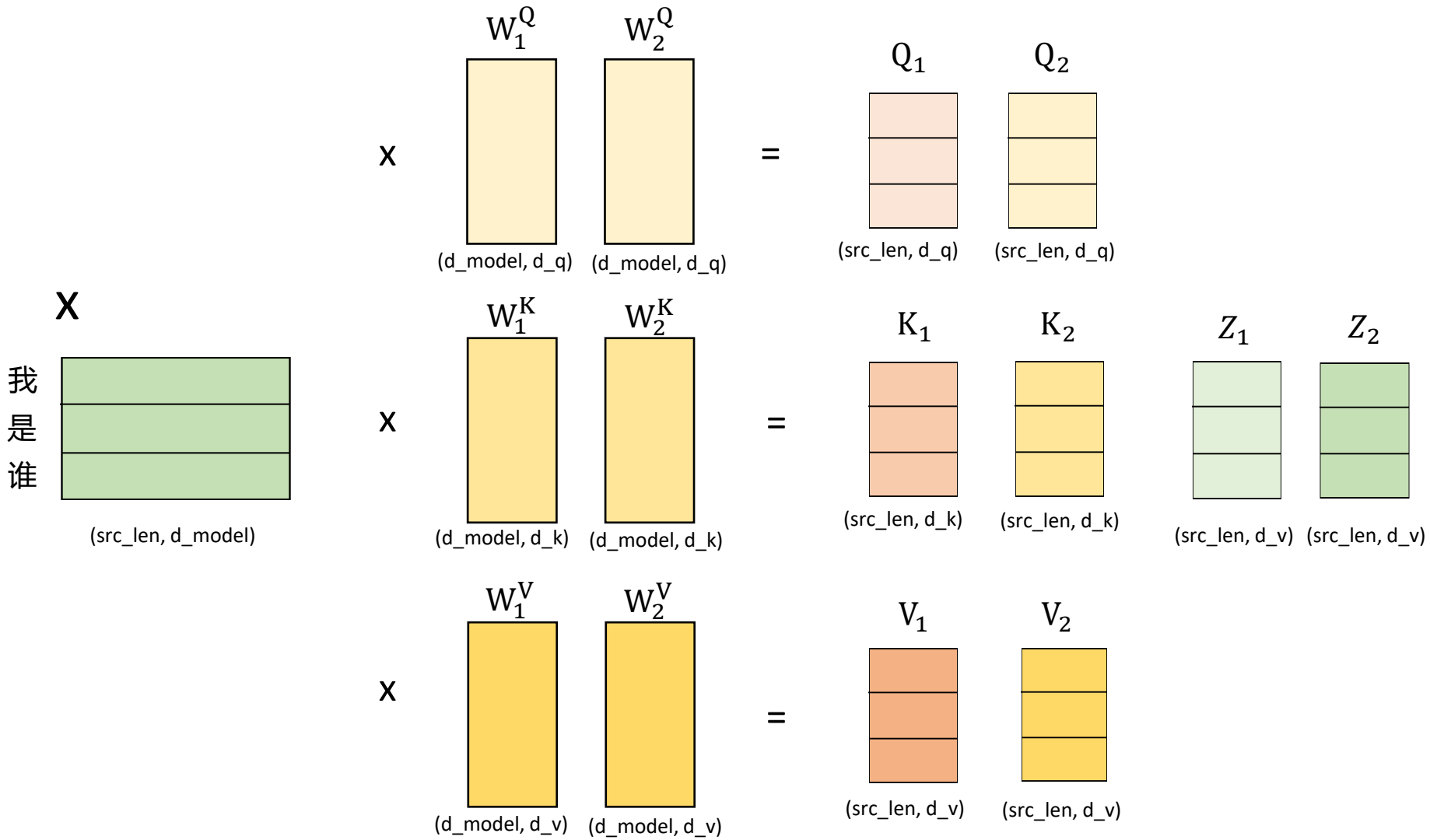


图10 多头自注意力机制计算过程

Multi-Head Attention

如图10所示，根据输入序列X和W_1^Q, W_1^K, W_1^V,我们就计算得到了Q_1, K_1, V_1，进一步根据自注意力计算公式就得到了单个注意力模块的输入Z_1；同理，根据X和W_2^Q, W_2^K, W_2^V就得到了另外一个自注意力模块输出Z_2；然后，将Z_1, Z_2水平堆叠形成Z，接着再用Z乘以W^O便得到整个多头注意力层的输出。

Multi-Head Attention

在初始化W^Q, W^K, W^V时，可以直接同时初始化h个头的权重，然后再进行后续的计算。具体如图11所示：

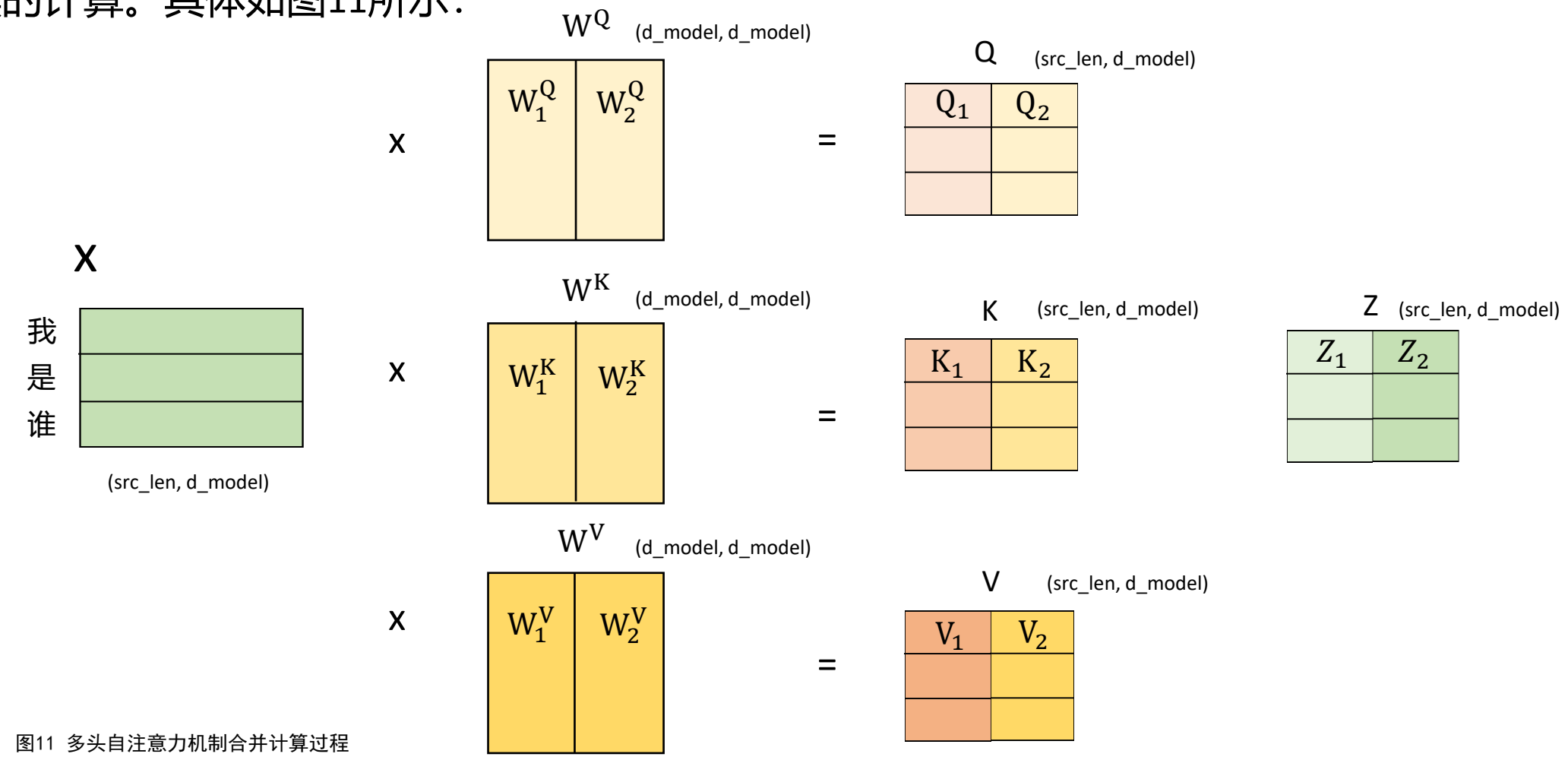


图11 多头自注意力机制合并计算过程

Multi-Head Attention

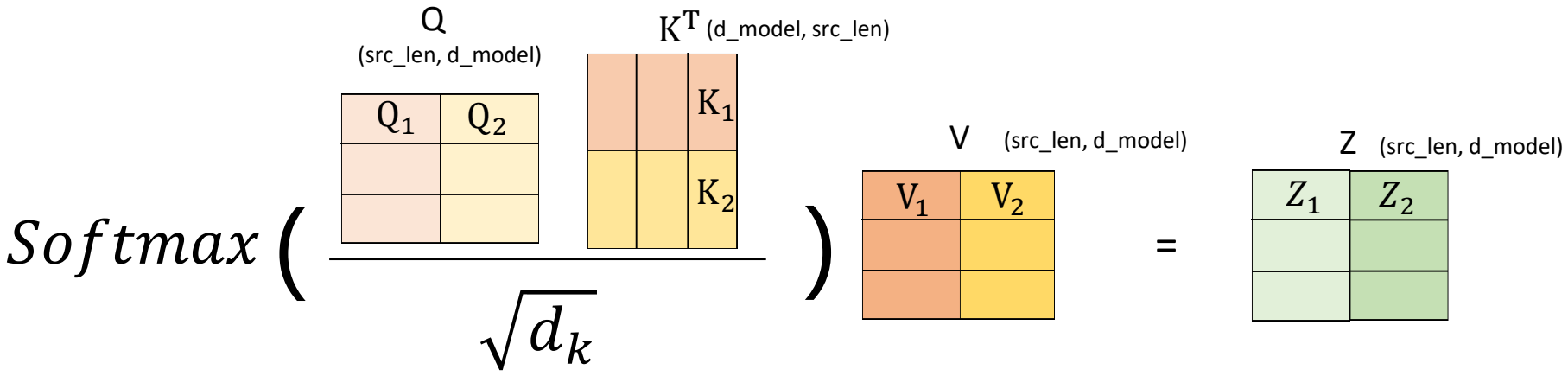


图12 多头自注意力计算过程

在d_{model}固定的情况下，不管是使用单头还是多头的方式，在实际处理过程中，直到进行注意力权重计算前，两者之间没有任何区别。当进行注意力权重计算时，h越大，Q、K、V就会被切分的越小，进而注意力权重分配方式越多。

Multi-Head Attention

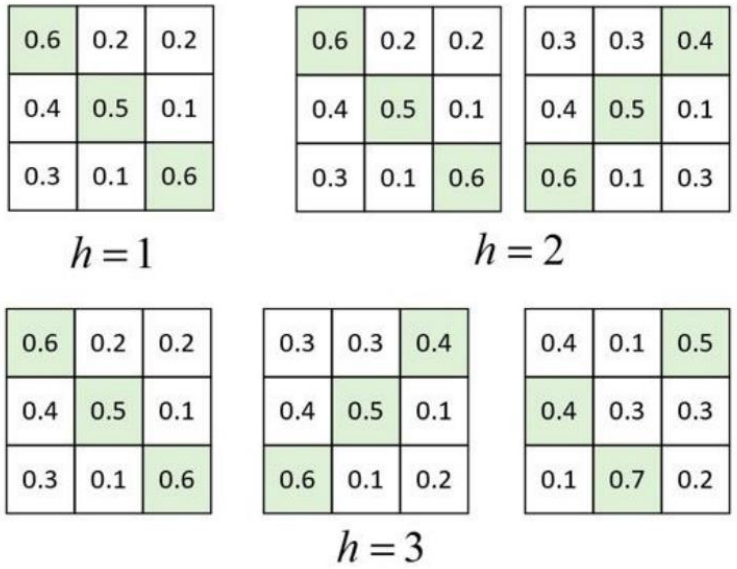


图13 注意力机制分配图

由图13可以看出，如果$h = 1$，那么最终可能得到的就是一个各个位置只集中于自身位置的注意力权重矩阵；如果$h = 2$，那么就还可能得到另外一个注意力权重稍微分配合理的权重矩阵； $h = 3$同理如此。因而多头这一做法也恰好是论文中提出用于克服模型在对当前位置的信息进行编码时，会过度的将注意力集中于自身的位置的问题。

Positional Embedding

根据自注意力机制的原理可以发现，自注意力机制在实际运算过程中仅仅只是通过矩阵相乘的形式进行线性变换而已。这就导致两个句子只有包含的词相同，即使顺序不同，它们的表示也会完全相同。换句话说仅仅使用自注意力机制会丢失文本原有的序列信息，具体如图14所示：

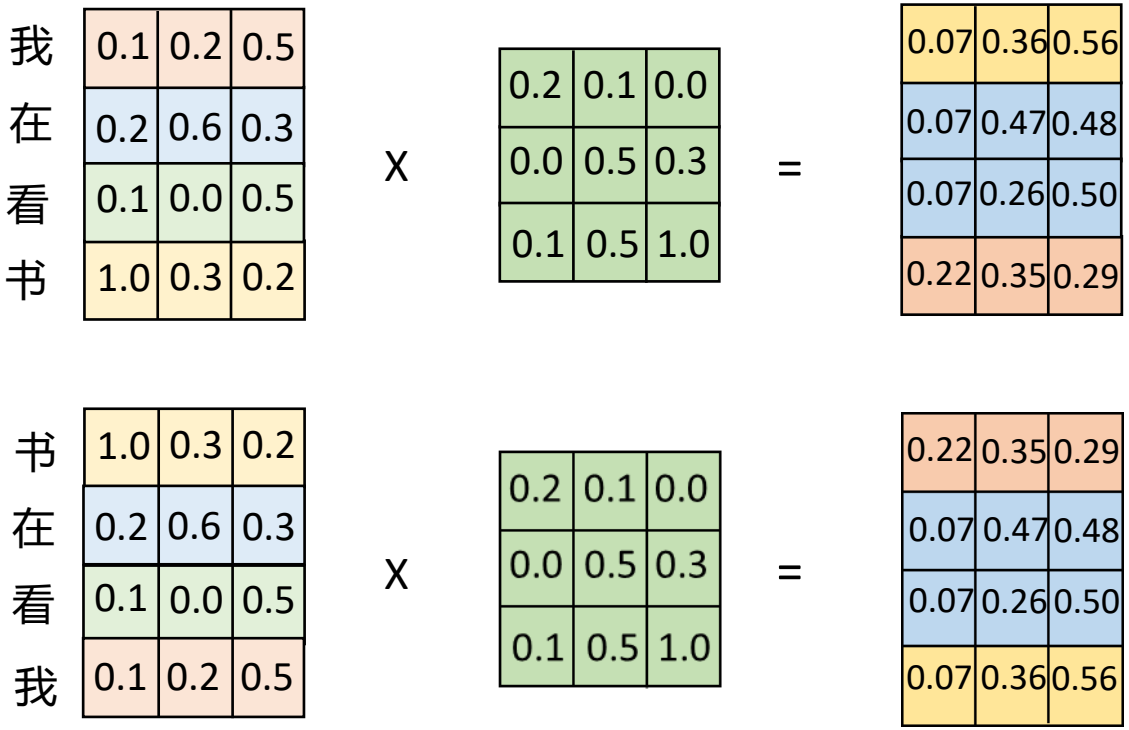


图14 自注意力机制弊端

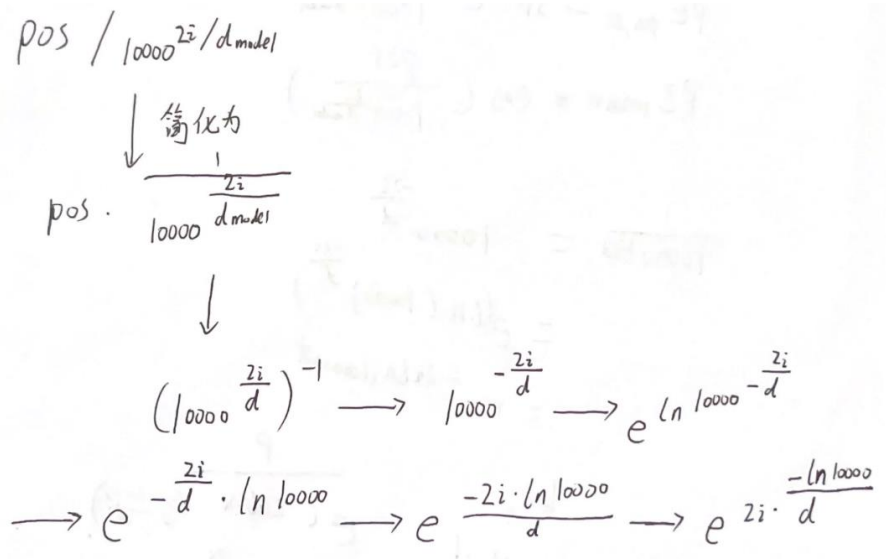
序列在交换位置前和交换位置后计算得到的结果本质上并没有任何区别，仅仅只是交换了对应的位置。为了解决这个问题，Transformer在原始输入文本进行Token Embedding后，融入了Positional Embedding 用于刻画数据在时序上的特征。

Positional Embedding

在Transformer中，作者采用了如公式（3）所示的规则来生成各个维度的位置信息。

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}) \quad (3)$$

式中，PE表示Positional Embedding矩阵，$pos \in [0, \max_len)$表示序列中位置的索引值，$i \in [0, \frac{d_{model}}{2})$表示位置编码向量的索引值。



Positional Embedding

```
1  class PositionalEncoding(nn.Module):
2      def __init__(self, d_model, dropout=0.1, max_len=5000):
3          super(PositionalEncoding, self).__init__()
4          self.dropout = nn.Dropout(p=dropout)
5          pe = torch.zeros(max_len, d_model) # [max_len, d_model]
6          position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
           # [max_len, 1]
7          div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
                       math.log(10000.0) / d_model)) # [d_model/2]
8
9          pe[:,0::2] =torch.sin(position * div_term) # [max_len, d_model/2]
10         pe[:, 1::2] = torch.cos(position * div_term)
11         pe = pe.unsqueeze(0).transpose(0, 1)  # [max_len, 1, d_model]
12         self.register_buffer('pe', pe)
13
14     def forward(self, x):
15         """
16         :param x: [x_len, batch_size, emb_size]
17         :return: [x_len, batch_size, emb_size]
18         """
19         x = x + self.pe[:x.size(0), :]  # [x_len, batch_size, d_model]
20         return self.dropout(x)
```

第 5 行代码是用来初始化一个全 0 的位置矩阵来保存位置信息，同时还指定了一个序列的最大长度；第6-10行是用来计算每个维度（每一列）的相关位置信息；第19行首先是在位置矩阵中取与输入序列长度相等的前x_len行，然后再加上Token Embedding的结果。这里需要注意的是，输入x的维度中batch_size并不是在第一个维度。

Encoder

Encoder层网络结构具体如图15所示。论文中作者通过6个Encoder层堆叠形成Transformer的Encoder，这里首先以一层进行介绍。

对于Encoder部分来说其内部主要由两部分网络所构成：多头注意力机制和两层前馈神经网络。同时，对于这两部分网络都加入了残差连接，并在残差连接后还进行了层归一化操作。这样对于每个部分来说，其输出均为$LayerNorm(x + Sublayer(x))$，并且都加入了Dropout操作。

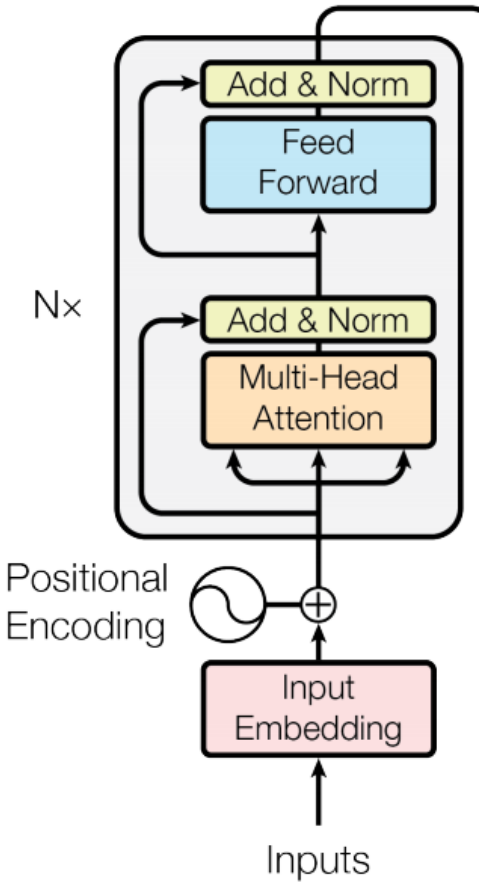


图15 Encoder网络结构

Encoder

为了便于使用残差连接，这两部分网络输出向量维度均设置为$d_{model} = 512$。

对于第二部分两层全连接网络，其具体计算过程为：

$$FFN(x) = \max(0, xW_1 + b_1)\, W_2 + b_2 \qquad (4)$$

其中，输入x的维度为$d_{model} = 512$，第一层全连接层的输出维度为$d_{ff} = 2048$，第二层全连接层的输出为$d_{model} = 512$，且同时对于第一层网络的输出中加了$ReLU$激活函数。

Decoder

同Encoder部分一样，论文中也采用了6个完全相同的Decoder层堆叠形成Decoder，这里同样以一层的Decoder进行介绍。对于Decoder部分来说，其整体上与Encoder类似，只是多了一个用于与Encoder输出进行交互的多头注意力机制。

不同于Encoder部分，在Decoder中一共包含有3个部分的网络结构。分别为经过掩码的多头注意力机制（掩码的部分稍后介绍）、与Encoder输出（Memory）交互的部分，作者称之为“Encoder-Decoder attention”，以及两层前馈神经网络。

对于Decoder与Encoder输出进行交互的多头注意力机制部分，其输入Q来自于Decoder中多头注意力机制的输出，K和V均是Encoder部分的输出经过线性变换得到。而作者之所以这样设计，也是在模仿传统Encoder-Decoder网络模型的解码过程。

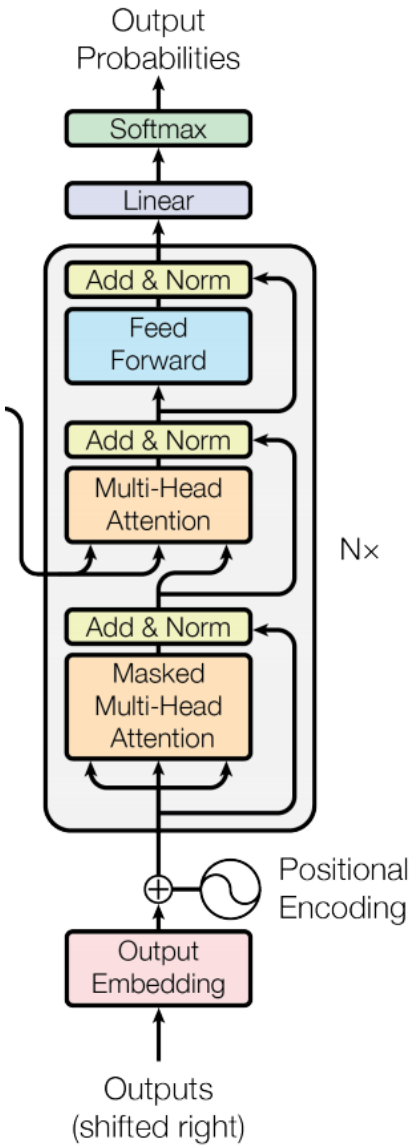


图16 Dncoder网络结构

Decoder

为了能够更好理解这里Q、K、V的含义，我们先来看看传统的基于Encoder-Decoder的Seq2Seq翻译模型是如何进行解码的。具体如图17所示：

图17中，左下部分为Encoder，右下部分为Decoder，左上部分便是注意力机制的部分。$\bar{h}_i$ 表示在编码过程中，各个时刻的隐含状态，称之为每个时刻的Memory；h_t 表示解码当前时刻时的隐含状态。此时注意力机制的思想在于，希望模型在解码的时刻能够参考每个时刻的记忆。

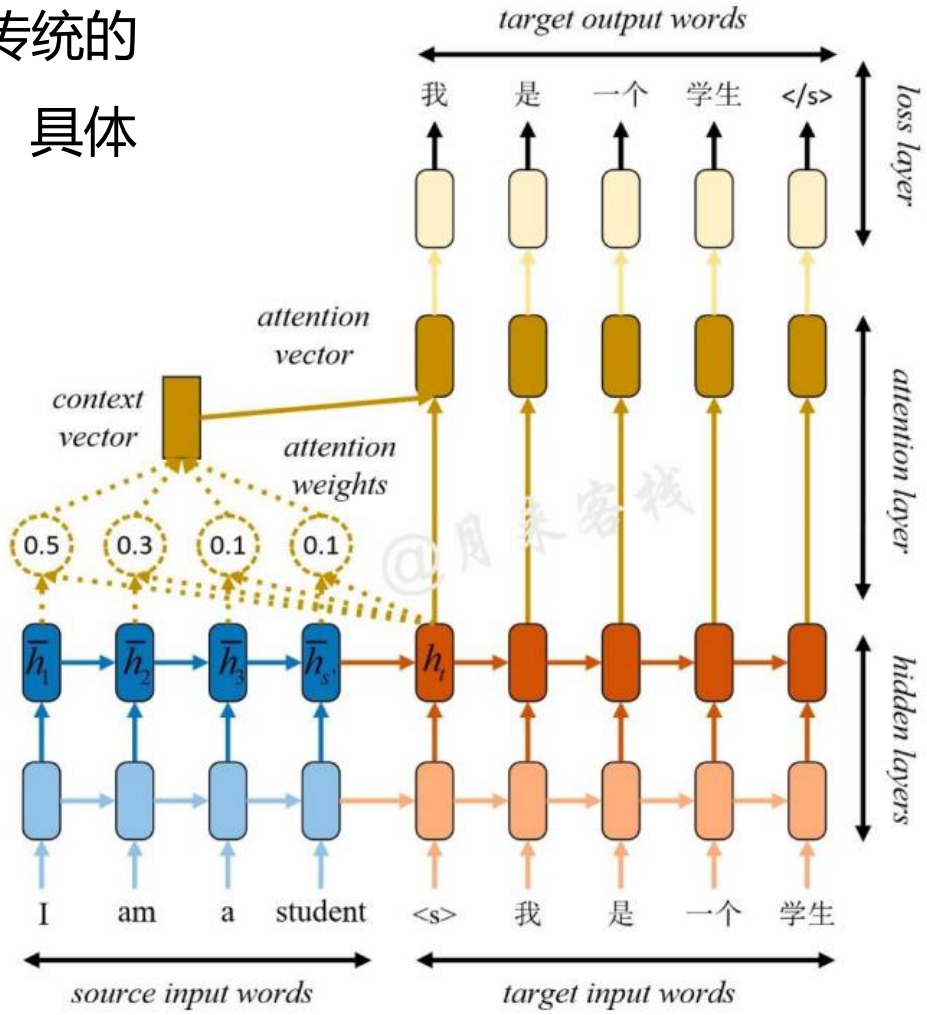


图17 传统的Seq2seq网络结构

Decoder

在解码第一个时刻“<s>”时，h_t会首先同每个状态进行相似度比较得到注意力权重。这个注意力权重所蕴含的意思是，在解码第一个时刻时应该将50%的注意力放在编码第一个时刻的记忆上（其他同理），最终通过加权求和得到4个Memory的权重和，即$context\ vector$。同理，在解码第二个时刻“我”时，也遵循上面的这一解码过程。可以看出，此时注意力机制扮演的就是能够是的Encoder和Decoder进行交互的角色。

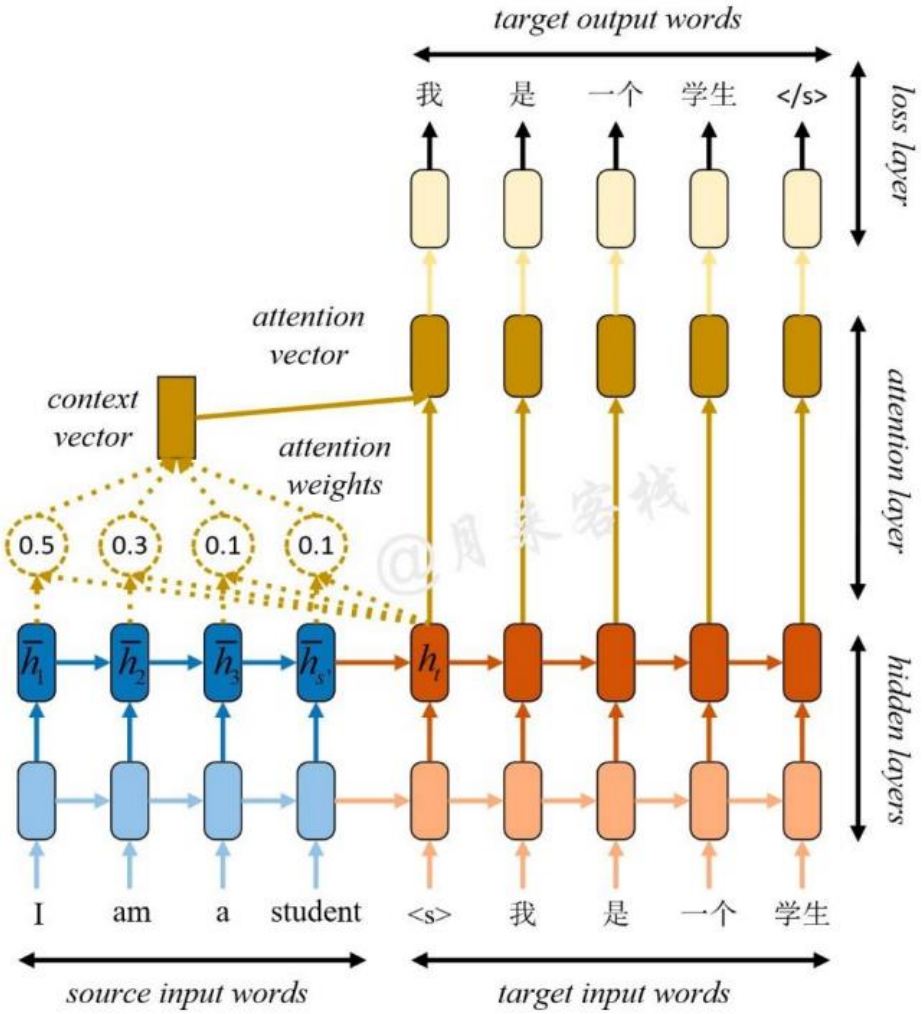


图17 传统的Seq2seq网络结构

Decoder

回到Transformer的Encoder-Decoder attention中，K和V均是编码部分的输出Memory经过线性变换后的结果（此时的Memory中包含了原始输入序列每个位置的编码信息，即Positional Embedding部分），而Q是解码部分多头注意力机制输出的隐含向量经过线性变换后的结果。在Decoder对每一时刻进行解码时，首先需要做的便是通过Q和K进行交互（query查询），并计算得到注意力权重矩阵；然后再通过注意力权重与V进行计算得到一个权重向量。该权重向量所表示的含义就是在解码时如何将注意力分配到Memory的各个位置上，具体过程如图18所示：

Decoder

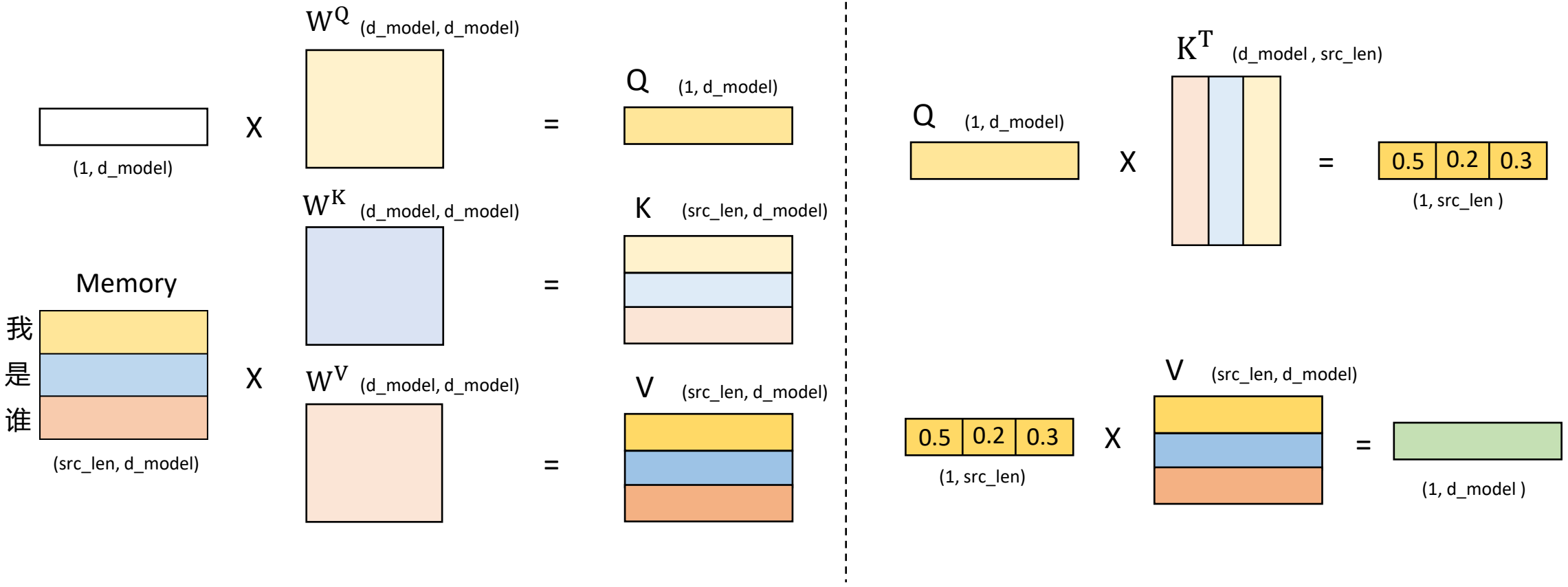


图18 Encoder-Decoder交互部分注意力机制计算过程

Decoder

如图18所示，在解码第1个时刻时，首先Q通过与K进行交互得到权重向量，此时可以看作是待解码向量Q在K（本质上也就是Memory）中查询K中各个位置与Q有关的信息；然后将权重向量与V进行运算得到解码向量，此时这个解码向量可以看作是考虑了Memory中各个位置编码信息的输出向量，也就是说它包含了在解码当前时刻时应该将注意力放在Memory中哪些位置上的信息。最后，将得到的解码向量经过图16最上面的两层前馈神经网络后，便将其输入到分类层中进行分类得到当前时刻的解码输出值。

Decoder

当第1个时刻的解码过程完成之后，解码器便后将解码第1个时刻的输入，以及解码第1个时刻后的输出均作为解码器的输入来预测第2个时刻的输出。整个过程可由图19进行表示：

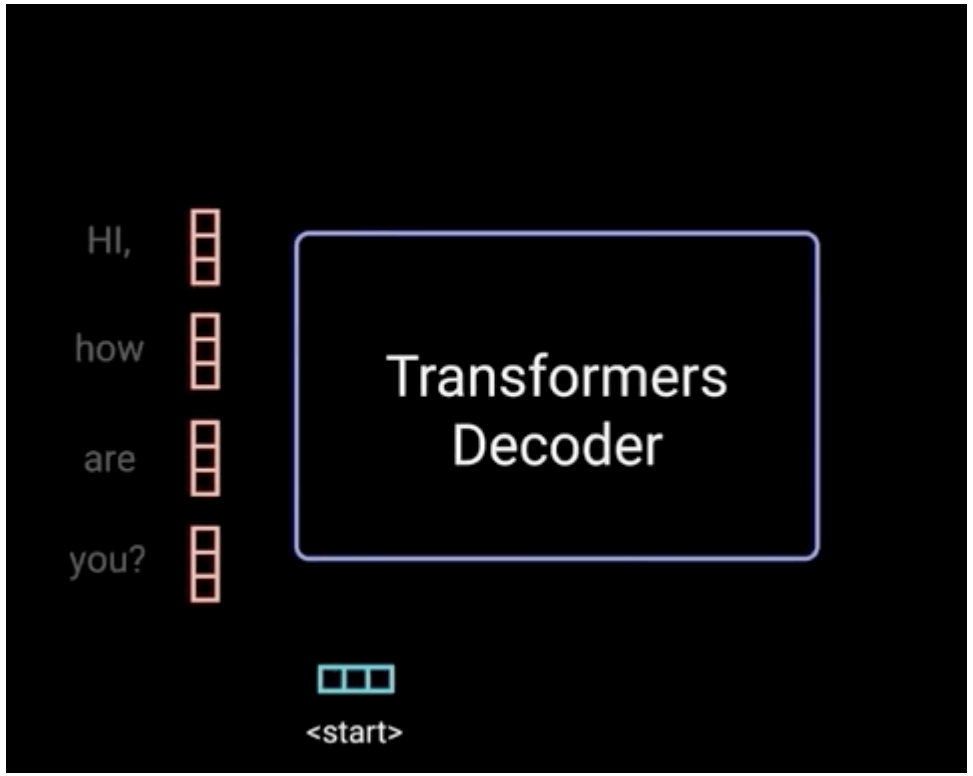


图19 Decoder多时刻解码过程

Decoder在对当前时刻进行解码输出时，都会将当前时刻之前的所有预测结果作为输入来对下一时刻的输出进行预测。然后，依次循环此过程，直到预测结果为“<e>”或者达到指定长度后停止。

Decoder

在介绍完预测时Decoder的解码过程后，下面我们来看一下网格在训练过程中是如何进行解码的。通过上述介绍可以看出，在真实预测时解码器需要将上一个时刻的输出作为下一个时刻解码的输入，然后一个时刻一个时刻的进行解码操作。显然，如果训练时也采用这种方法将会十分费时。因此，在训练过程中，解码器也同编码器一样，一次接收解码时所有时刻的输入进行计算。

这样做的好处在于：

- 多样本并行计算能够加快网络的训练速度；
- 训练过程中直接输入解码器正确的结果而不是上一个时刻的预测值能够更好的训练网络（因为训练时上一时刻的预测值可能是错误的）。

Decoder

但此时会存在一个问题，即模型在实际的预测过程中只是将当前时刻之前（包括当前时刻）的所有时刻作为输入来预测下一时刻，也就是说模型在预测时是看不到当前时刻之后的信息。为了解决这一问题，Transformer在Decoder层中加入了注意力掩码机制。具体如图20所示：

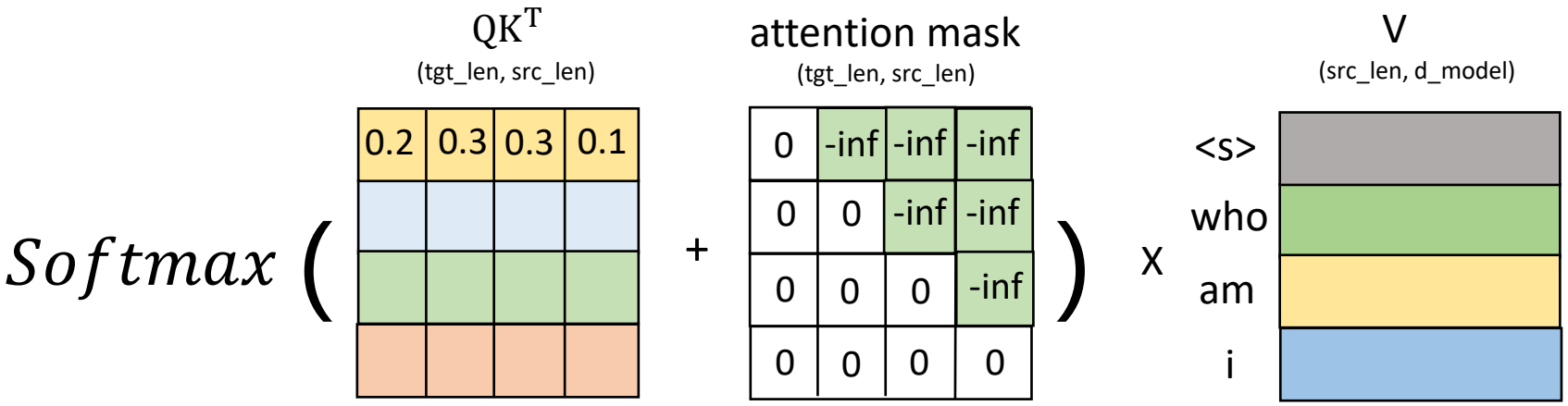


图20 注意力掩码计算过程

Decoder

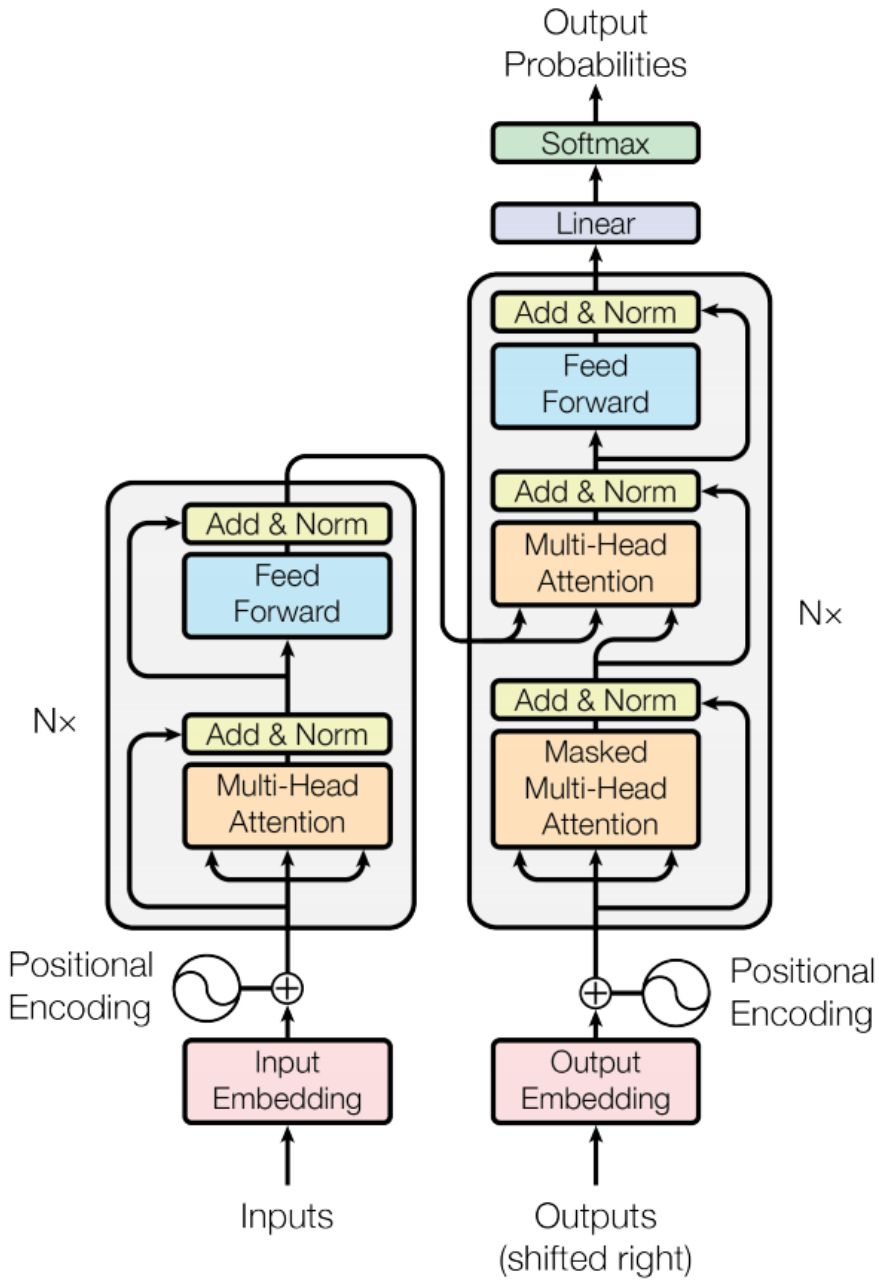
以图20中第一行权重为例，当解码器对第1时刻进行解码时其对应的输入只有“<s>”,因此这就意味着此时应该将所有的注意力放在第1个位置上（尽管在训练时解码器一次喂入了所有的输入），换句话说也就是第1个位置上的权重应该是1，而其它位置则是0。第1行注意力向量加上第1行注意力掩码，再经过$Softmax$操作后便可得到一个类似[1,0,0,0]的向量，那么通过这种方式就能够保证在解码第1个时刻时只能将注意力放在第1个位置上的特性。

The Sources of Q K and V

这里再次总结一下各个部分Q、K、V的来源，从Transformer整体架构图中可以看出，涉及到自注意力机制的一共有三个部分：Encoder中的Multi-Head Attention；Decoder中的Masked Multi-Head Attention；以及Encoder和Decoder交互部分的Multi-Head Attention。

- 对于Encoder中的Multi-Head Attention，其原始的q、k、v均是Encoder的Token输入经过Embedding后的结果。q、k、v分别经过一次线性变换（各自乘以一个权重矩阵）后得到了Q、K、V，然后再进行自注意力计算得到Encoder部分的输出结果Memory。
- 对于Decoder 中的 Masked Multi-Head Attention 来说，其原始 q、k、v 均 是 Decoder 的 Token 输入经过 Embedding 后的结果。q、k、v 分别经过一次线性变换后得到了 Q、K、V，然后再进行自注意力计算得到 Masked Multi-Head Attention 部分的输出结果，即待解码向量。
- 对于 Encoder 和 Decoder 交互部分的 Multi-Head Attention，其原始 q、k、 v 分别是上面的待解码向量、Memory 和 Memory。q、k、v 分别经过一次线性变换后得到了 Q、K、V，然后再进行自注意力计算得到 Decoder 部分的输出结果。之所以这样设计也是在模仿传统 Encoder-Decoder网络模型的解码过程。

基于Transformer的文本分类任务实现


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Encoder Layer

```
d_model = 4
num_heads = 2

# 定义一个EncoderLayer
encoder_layer = nn.TransformerEncoderLayer(d_model=d_model,  # 输入、输出向量维度
                                           nhead=num_heads,  # 头数
                                           dim_feedforward=2048  # 前馈神经网络维度，默认2048
                                           )
```

```
# 随机生成输入， 三个参数分别为序列的长度、批次的大小和每个输入向量的维度
src_len = 2
batch_size = 3
src = torch.rand(src_len, batch_size, d_model)

out = encoder_layer(src)

src.shape, out.shape
```

```
(torch.Size([2, 3, 4]), torch.Size([2, 3, 4]))
```

Encoder

```
# 将多个EncoderLayer堆叠起来，构成一个Encoder
transformer_encoder = nn.TransformerEncoder(encoder_layer,  # 编码层
                                            num_layers=6, # 层数
                                            norm=nn.LayerNorm(d_model)  # 归一化方式，默认为None
                                           )

memory = transformer_encoder(src)
memory.shape
```

```
torch.Size([2, 3, 4])
```

Positional Embedding

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=10000):
        """
        Transformer的位置编码层
        :param d_model: 输入词向量维度
        :param dropout: 丢弃率
        :param max_len: 位置编码矩阵的最大长度
        """
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)  # [max_len, 1]
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))  # [d_model/2]
        pe[:, 0::2] = torch.sin(position * div_term)  # [max_len, d_model/2]  对偶数进行位置编码
        pe[:, 1::2] = torch.cos(position * div_term)  # 对奇数进行位置编码
        pe = pe.unsqueeze(0).transpose(0, 1)  # [max_len, 1, d_model]
        self.register_buffer('pe', pe)  # 将pe注册为模型参数

    def forward(self, x):
        """
        前向传播过程
        :param x: [x_len, batch_size, embedding_dim]
        :return: [x_len, batch_size, embedding_dim]
        """
        x = x + self.pe[:x.size(0), :]  # [x_len, batch_size, d_model] 输入的词向量与位置编码进行相加
        return self.dropout(x)
```

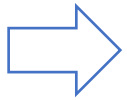
Positional Embedding

当输入序列由于长度不一致，经过pad_sequence补齐后，会导致在注意力计算的过程中考虑Padding位置的信息，此时可通过对Padding位置信息进行掩盖，从而达到忽略Padding位置信息的目的。

```python
def length_to_mask(lengths):
    """
    将序列的长度转换成mask矩阵，忽略序列补齐后padding部分的信息
    :param lengths: [batch,]
    :return: batch * max_len
    """
    max_len = torch.max(lengths).long()
    mask = torch.arange(max_len, device=lengths.device).expand(lengths.shape[0], max_len) < lengths.unsqueeze(1)
    return mask
```

```python
lengths = torch.tensor([2,3,4,5])
length_to_mask(lengths)
```

```
tensor([[ True,  True, False, False, False],
        [ True,  True,  True, False, False],
        [ True,  True,  True,  True, False],
        [ True,  True,  True,  True,  True]])
```



```python
lengths = torch.tensor([2,3,4,5])
length_to_mask(lengths) == False
```

```
tensor([[False, False,  True,  True,  True],
        [False, False, False,  True,  True],
        [False, False, False, False,  True],
        [False, False, False, False, False]])
```

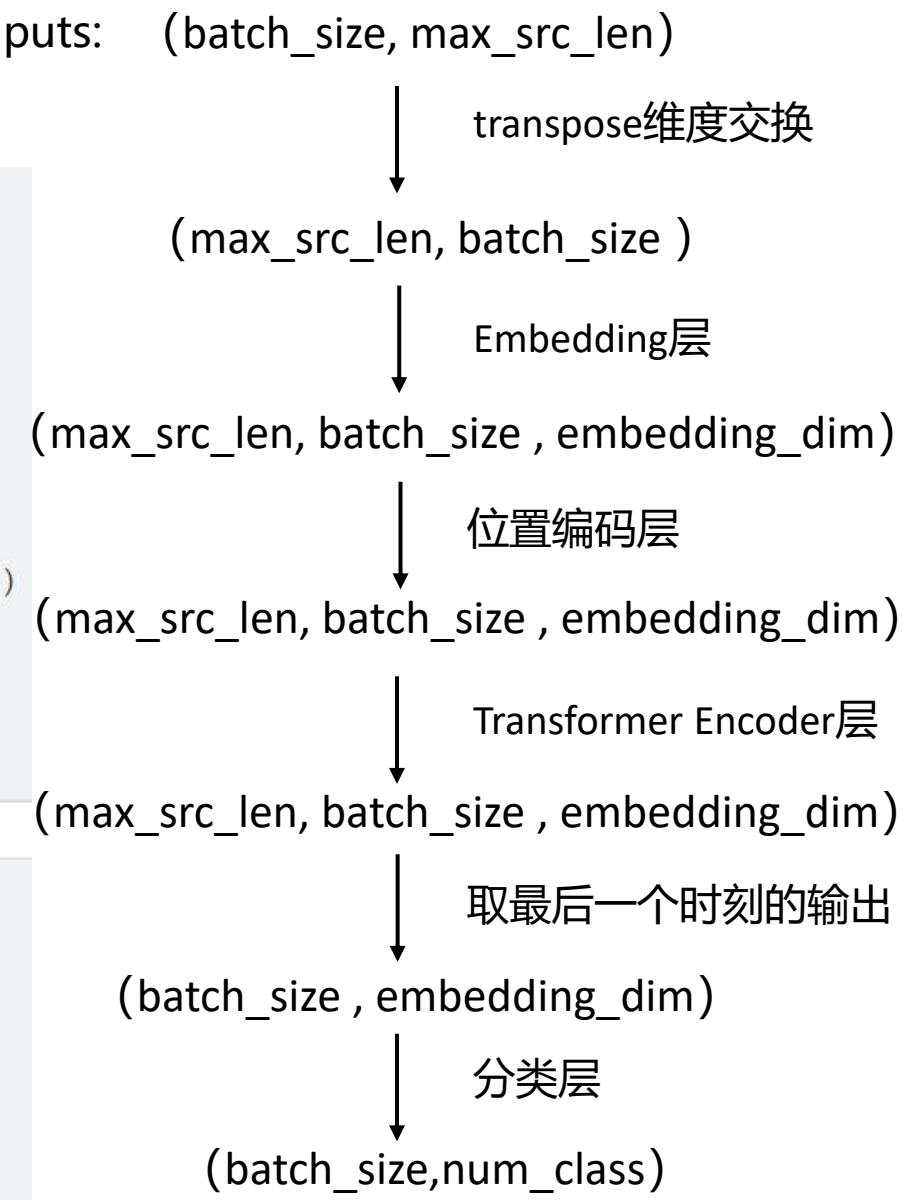
Transformer

```python
class Transformer(nn.Module):
    def __init__(self, vocab_size, num_class, d_model=512, dim_feedforward=2048, num_head=8, num_layers=6,
                 dropout=0.1, max_len=10000, activation: str = 'relu'):
        """
        Transformer
        :param vocab_size: 词表大小
        :param d_model: 词向量维度，相当于embedding_dim
        :param num_class: 分类类别数
        :param dim_feedforward: Encoder中前馈神经网络输出维度
        :param num_head: 多头注意力机制的头数
        :param num_layers: 编码器的层数
        :param dropout: 丢弃率
        :param max_len: 位置编码矩阵的最大长度
        :param activation: 激活函数
        """
        super(Transformer, self).__init__()
        self.embedding_dim = d_model
        # 词嵌入层
        self.embeddings = nn.Embedding(vocab_size, self.embedding_dim)
        # 位置编码层
        self.position_embedding = PositionalEncoding(self.embedding_dim, dropout, max_len)
        # Transformer 编码层
        encoder_layer = nn.TransformerEncoderLayer(d_model, num_head, dim_feedforward, dropout, activation)
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers, norm=nn.LayerNorm(d_model))
        # 分类层
        self.classifier = nn.Linear(d_model, num_class)
```

Transformer

```python
def forward(self, inputs, lengths):
    # 数据处理与lstm相同，输出数据的第一个维度是批次，但这里需要将其转换为
    # Transformer Encoder所需要的第一个维度是序列长度，第二个维度是批次的形状
    inputs = torch.transpose(inputs, 0, 1)
    # 这里根据论文3.4部分的描述对原始词向量进行了缩放
    embeddings = self.embeddings(inputs.long()) * math.sqrt(self.embedding_dim)
    hidden_states = self.position_embedding(embeddings)
    # 根据批次中每个序列长度生成mask矩阵
    attention_mask = length_to_mask(lengths) == False
    memory = self.transformer(hidden_states, src_key_padding_mask=attention_mask)
    # 取最后一个时刻的输出作为分类层的输入
    memory = memory[-1, :, :]
    output = self.classifier(memory)
    return output
```

```python
def generate_batch_lstm(examples):
    """rnn：对一个批次内的数据进行处理"""
    lengths = torch.tensor([len(ex[0]) for ex in examples])
    inputs = [torch.tensor(ex[0], dtype=torch.long) for ex in examples]
    labels = torch.tensor([ex[1] for ex in examples], dtype=torch.long)
    inputs = pad_sequence(inputs, batch_first=True)
    return inputs, lengths, labels
```



Transformer的优缺点

- 与RNN相比，Transformer能够直接建模输入序列单元之间更长距离的依赖关系，从而使得Transformer对于长序列建模的能力更强。（长距离建模）
- 另外，在Transformer的编码阶段，由于可以利用GPU等多核计算设备并行地计算Transformer块内部的自注意力模型，而循环神经网络需要逐个计算，因此Transformer具有更高的训练速度。（并行计算）
- 由于自注意力机制中的Q、K、V映射矩阵、多头机制导致相应参数倍增和引入非线性的多层感知器等，更主要的是，堆叠了多层的Transformer块，导致最终Transformer模型参数体系过于庞大，以至于模型训练尤为困难，特别在训练数据较少的情况下。（参数量大）