This Post Is All You Need（上卷）

——层层剥开 Transformer

目　录

修订记录

2022 年 07 月 03 日，v1.3.1 修改错别字
2022 年 05 月 01 日，v1.3.0 添加 3.2.3 节内容，修改 bug，调整格式
2022 年 03 月 08 日，v1.2.3
2022 年 03 月 02 日，v1.2.2
2022 年 12 月 11 日，v1.2.1
2021 年 10 月 17 日，v1.1.0
2021 年 10 月 12 日，v1.0.0 初始版本发布

第 1 节 多头注意力机制原理

1.1 动机

各位朋友大家好，欢迎来到月来客栈。今天要和大家介绍的一篇论文是谷歌 2017 年所发表的一篇论文，名字叫做“Attention is all you need”[1]。当然，网上已经有了大量的关于这篇论文的解析，不过好菜不怕晚，掌柜在这里也会谈谈自己对于它的理解以及运用。按照我们一贯解读论文的顺序，首先让我们先一起来看看作者当时为什么要提出 Transformer 这个模型？需要解决什么样的问题？现在的模型有什么样的缺陷？

1.1.1 面临问题

在论文的摘要部分作者提到，现在主流的序列模型都是基于复杂的循环神经网络或者是卷积神经网络构造而来的 Encoder-Decoder 模型，并且就算是目前性能最好的序列模型也都是基于注意力机制下的 Encoder-Decoder 架构。为什么作者会不停的提及这些传统的 Encoder-Decoder 模型呢？接着，作者在介绍部分谈到，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于上一个时刻的输出，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，如图 1-1 所示。

This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples.

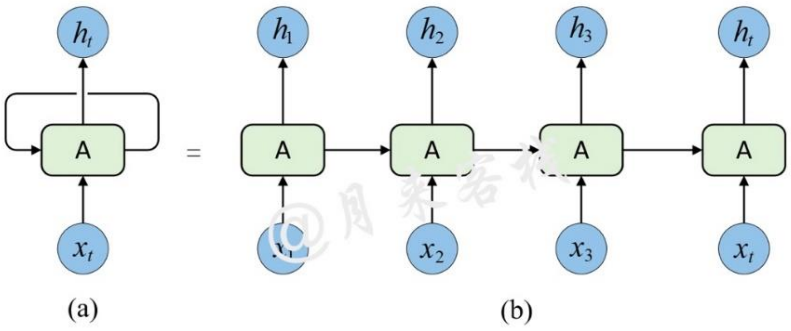

图 1-1. 循环神经网络编码图

随后作者谈到，尽管最新的研究工作已经能够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。

Recent work has achieved significant improvements in computational efficiency through factorization tricks and conditional computation, while also improving model performance in case of the latter. The fundamental constraint of sequential computation, however, remains.

1.1.2 解决思路

因此，在这篇论文中，作者首次提出了一种全新的 Transformer 架构来解决这一问题，如图 1-2 所示。当然，Transformer 架构的优点在于它完全摈弃了传统的循环结构，取而代之的是只通过注意力机制来计算模型输入与输出的隐含表示，而这种注意力的名字就是大名鼎鼎的自注意力机制（self-attention），也就是图 1-2 中的 Multi-Head Attention 模块。

To the best of our knowledge, however, the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence- aligned RNNs or convolution.

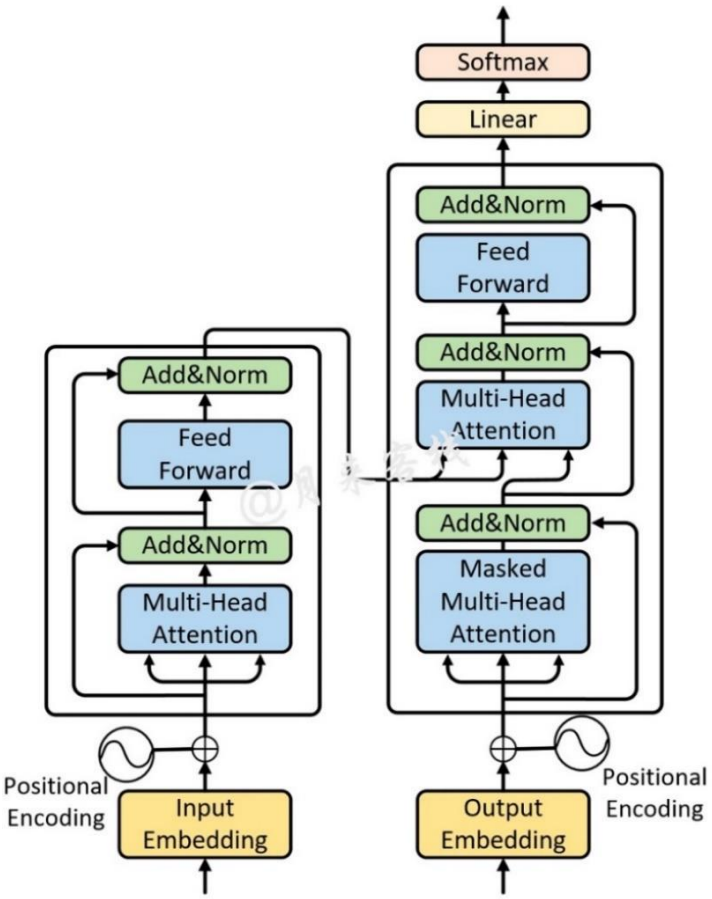

图 1-2. Transformer 网络结构图

总体来说，所谓自注意力机制就是通过某种运算来直接计算得到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。最终，Transformer 架构就是基于这种的自注意力机制而构建的 Encoder-Decoder 模型。

1.2 技术手段

在介绍完整篇论文的提出背景后，下面就让我们一起首先来看一看自注意力机制的庐山真面目，然后再来探究整体的网络架构。

1.2.1 什么是 self-Attention

首先需要明白一点的是，所谓的自注意力机制其实就是论文中所指代的“Scaled Dot-Product Attention”。在论文中作者说道，注意力机制可以描述为将 query 和一系列的 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。

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.

不过要深入理解 query、key 和 value 的含义，需要结合 Transformer 的解码过程，这部分内容将在后续进行介绍。具体的，自注意力机制结构如图 1-3 所示。

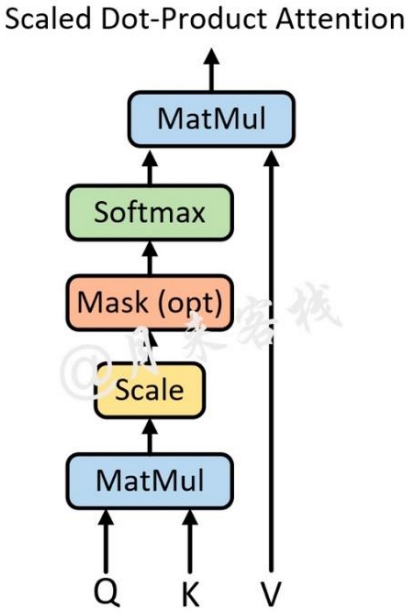

图 1-3. 自注意力机制结构图

从图 1-3 可以看出，自注意力机制的核心过程就是通过 Q 和 K 计算得到注意力权重；然后再作用于 V 得到整个权重和输出。具体的，对于输入 Q、K 和 V 来说，其输出向量的计算公式为：

$$\text{Attention}(Q,K,V)=\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{1.1}$$

其中 Q、K 和 V 分别为 3 个矩阵，且其（第 2 个）维度分别为 d_q,d_k,d_v （从后面的计算过程其实可以发现 $d_q=d_v$。而公式(1.1)中除以 $\sqrt{d_k}$ 的过程就是图 1-3 中所指的 Scale 过程。

之所以要进行缩放这一步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 sofrmax 操作后产生非常小的梯度，不利于网络的训练。

We suspect that for large values of dk, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

如果仅仅只是看着图 1-3 中的结构以及公式(1.1)中的计算过程显然是不那么容易理解自注意力机制的含义，例如初学者最困惑的一个问题就是图 1-3 中的 Q、K 和 V 分别是怎么来的？下面，我们来看一个实际的计算示例。现在，假设输入序列为“我 是 谁”，且已经通过某种方式得到了 1 个形状为3×4的矩阵来进行表示，即图中的 X。那么通过图 1-3 所示的过程便能够就算得到 Q、K 以及 V[2]。

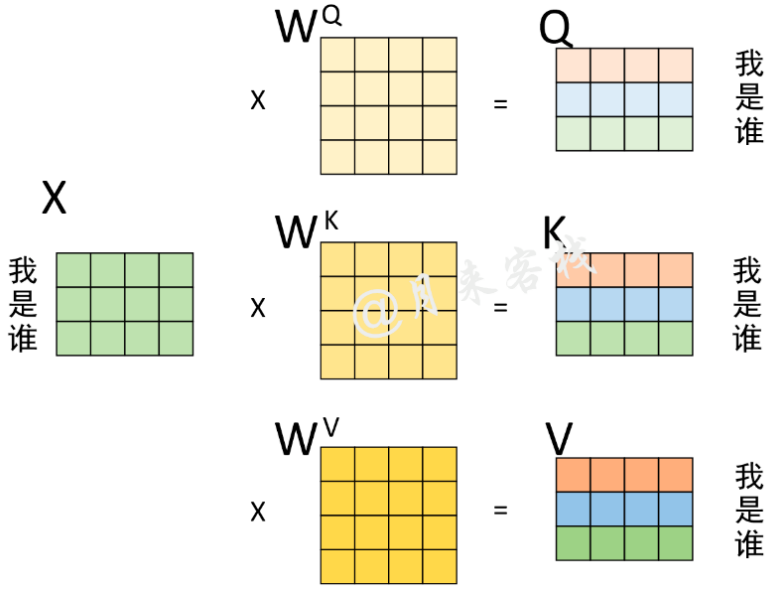

图 1-4. Q、K 和 V 计算过程图

从图 1-4 的计算过程可以看出，Q、K 和 V 其实就是输入 X 分别乘以 3 个不同的矩阵计算而来（但通过这种方式计算得到 Q、K 和 V 的过程仅仅局限于 Encoder 和 Decoder 在各自输入部分利用自注意力机制进行编码时的过程，Encoder 和 Decoder 交互部分的 Q、K 和 V 另有指代）。此处对于计算得到的 Q、K、V，你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不同的 3 种状态。在计算得到 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 1-5 所示。

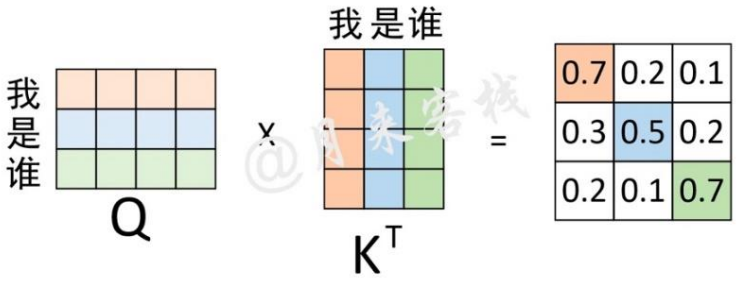

图 1-5. 注意力权重计算图（已经经过 scale 和 softmax 操作）

如图 1-5 所示，在经过上述过程计算得到了这个注意力权重矩阵之后我们不禁就会问到，这些权重值到底表示的是什么呢？对于权重矩阵的第 1 行来说，0.7 表示的就是“我”与“我”的注意力值；0.2 表示的就是“我”与“是”的注意

力值；0.1 表示的就是“我”与“谁”的注意力值。换句话说，在对序列中的“我”进行编码时，应该将 0.7 的注意力放在“我”上，0.2 的注意力放在“是”上，将 0.1 的注意力放在谁上。

同理，对于权重矩阵第 3 行来说，其含义是，在对序列中“谁”进行编码时，应该将 0.2 的注意力放在“我”上，将 0.1 的注意力放在“是”上，将 0.7 的注意力放在“谁”上。从这一过程可以看出，通过这个权重矩阵模型就能轻松的知道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到不同的位置上。

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，会过度的将注意力集中于自身的位置（虽然这符合常识）而可能忽略了其它位置[2]。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），这部分内容我们将在稍后看到。

It expands the model's ability to focus on different positions. Yes, in the example above, z_1 *contains a little bit of every other encoding, but it could be dominated by the the actual word itself.*

在通过图 1-5 示的过程计算得到权重矩阵后，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 1-6 所示。

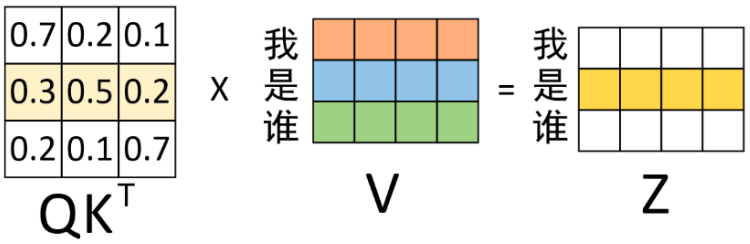

图 1-6. 权重和编码输出图

根据如图 1-6 所示的过程，我们便能够得到最后编码后的输出向量。当然，对于上述过程我们还可以换个角度来进行观察，如图 1-7 所示。

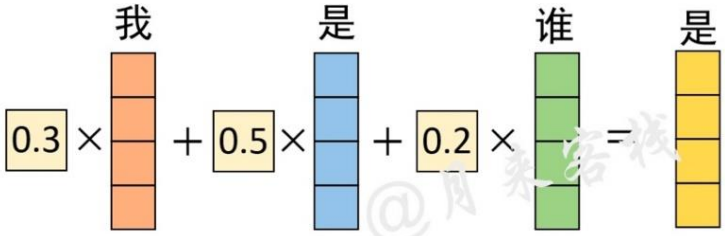

图 1-7. 编码输出计算图

从图 1-7 可以看出，对于最终输出“是”的编码向量来说，它其实就是原始“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。当然，对于整个图 1-5 到 1-6 的过程，我们还可以通过如图 1-8 所示的过程来进行表示。

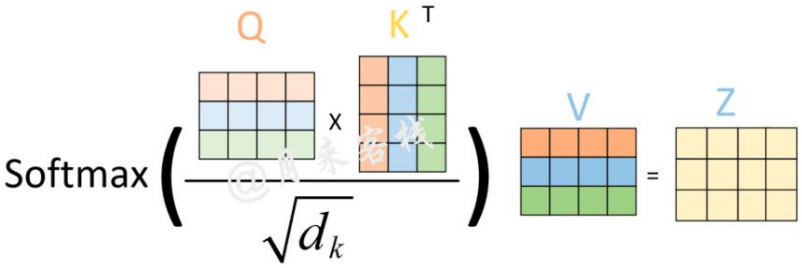

图 1-8. 自注意力机制计算过程图

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顺序进行的弊端”的问题，有了自注意力机制后，仅仅只需要对原始输入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。

对于自注意力机制的核心部分到这里就介绍完了，不过里面依旧有很多细节之处没有进行介绍。例如 Encoder 和 Decoder 在进行交互时的 Q、K、V 是如何得到的？在图 1-3 中所标记的 Mask 操作是什么意思，什么情况下会用到等等？这些内容将会在后续逐一进行介绍。

下面，让我们继续进入到 MultiHeadAttention 机制的探索中。

1.2.2 为什么要 MultiHeadAttention

经过上面内容的介绍，我们算是在一定程度上对于自注意力机制有了清晰的认识，不过在上面我们也提到了自注意力机制的缺陷就是：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中的编码表示信息，从而增强模型的表达能力。

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

在说完为什么需要多头注意力机制以及使用多头注意力机制的好处之后，下面我们就来看一看到底什么是多头注意力机制。

如图 1-9 所示，可以看到所谓的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。

具体的，其计算公式为：

$$\begin{aligned}&\text{MultiHead}(Q,K,V)=\text{Concat}(\text{head}_1,...,\text{head}_h)W^O\\&\text{where}\ \ \text{head}_i=\text{Attention}(QW_i^Q,KW_i^K,VW_i^V)\end{aligned}\tag{1.2}$$

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

值得注意的是，这里的 Q、K 和 V 图 1-4 中的 Q、K 和 V 并不是一回事儿。这里的 Q、K 和 V 要经过一次线性变换（即图 1-9 中的 Linear）操作后的结果才分别是图 1-4 中的 Q、K 和 V，为了保持和原始论文中的插图及公式一致所以这里掌柜并没有进行修改。

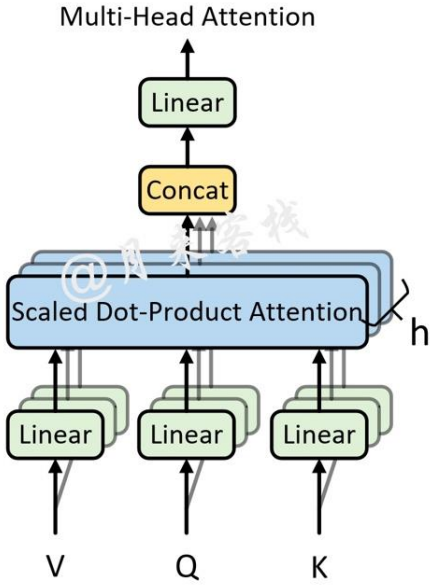

图 1-9. 多头注意力机制结构图

同时，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 $d_k=d_v=d_{model}/h=64$。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如图 1-10 所示的过程来进行表示。

如图 1-10 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，进一步根据公式 (1.1) 就得到了单个自注意力模块的输出 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。最后，根据公式(1.2)将 Z_1,Z_2 水平堆叠形成 Z，然后再用 Z 乘以 W^O 便得到了整个多头注意力层的输出。同时，根据图 1-9 中的计算过程，还可以得到 $d_q=d_k=d_v$。

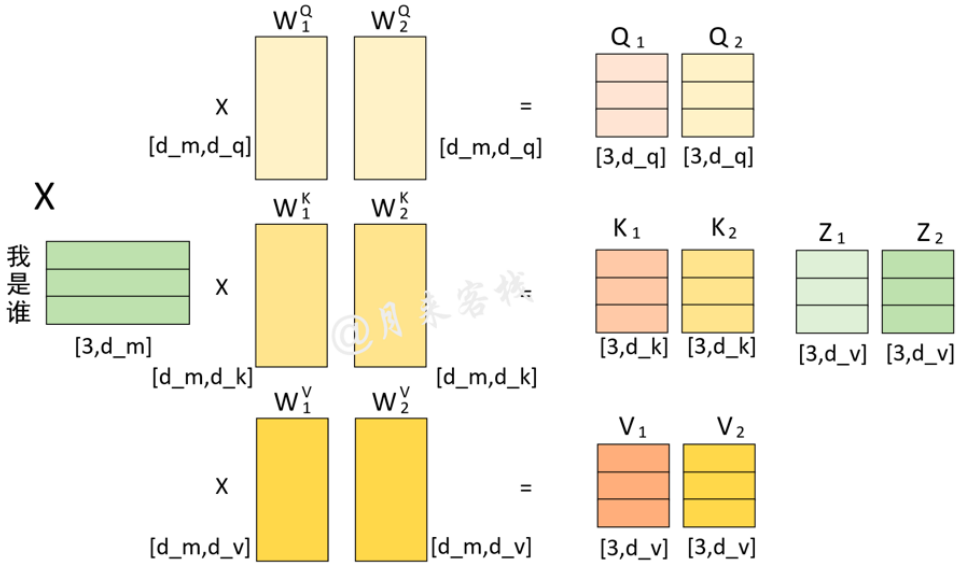

图 1-10. 多头注意力机制计算过程图

注意：图中的 d_m 就是指 d_{model}

到此，对于整个 Transformer 核心部分，即多头注意力机制原理就介绍完了。

1.2.3 同维度下单头与多头的区别

在多头注意力中，对于初学者来说一个比较经典的问题就是，在相同维度下使用单头和多头的区别是什么？这句话什么意思呢？以图 1-10 中示例为例，此时的自注意力中使用了两个头，每个头的维度为d_q，即采用了多头的方式。另外一种做法就是，只是用一个头，但是其维度为$2d_q$，即采用单头的方式。那么在这两种情况下有什么区别呢？

首先，从论文内容可知，作者在头注意力机制与多头个数之间做出了如下限制条件

$$d_q = d_k = d_v = \frac{d_{model}}{h} \tag{1.3}$$

从式(1.3)可以看出，单个头注意力机制的维度d_k乘上多头的个数h就等于模型的维度d_{model}。注意：后续的 d_m，d_m以及d_{model}都是指代模型的维度。

同时，从图 1-10 中可以看出，这里使用的多头数量$h=2$,即$d_{model}=2\times d_q$。此时，对于第 1 个头来说有：

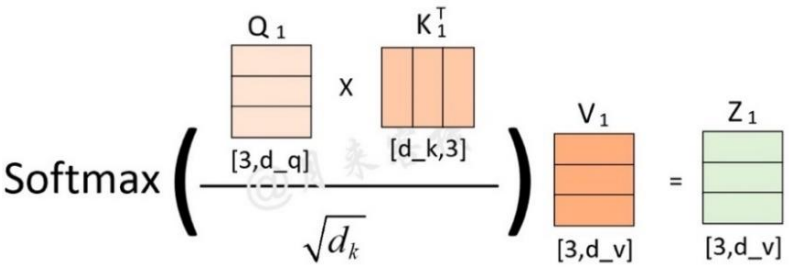

图 1-11. 头 1 注意力计算过程

此时，对于第 2 个头来说有：

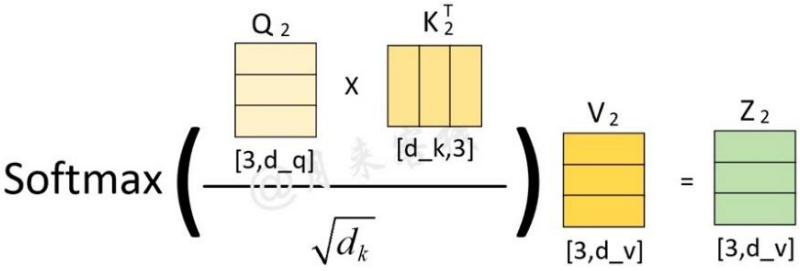

图 1-12. 头 2 注意力计算过程

最后，可以将Z_1, Z_2在横向堆叠起来进行一个线性变换得到最终的Z。因此，对于图 1-10 所示的计算过程，我们还可以通过图 1-13 来进行表示。

从图 1-13 可知，在一开始初始化W^Q, W^K, W^V这 3 个权重矩阵时，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。而且事实上，在真正的代码实现过程中也是采用的这样的方式，这部分内容将在 3.3.2 节中进行介绍。因此，对图 1-13 中的多头计算过程，还可以根据图 1-14 来进行表示。

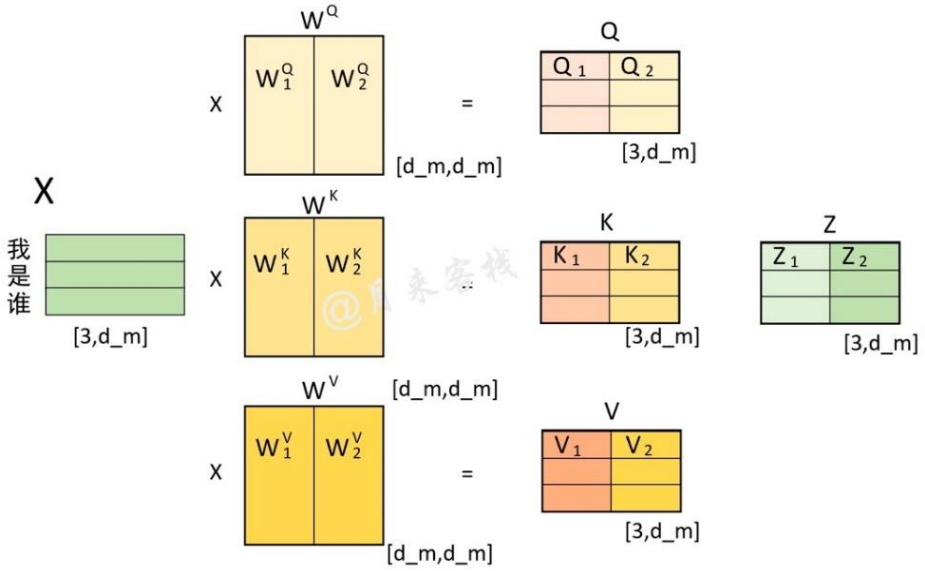

图 1-13. 多头注意力合并计算过程图

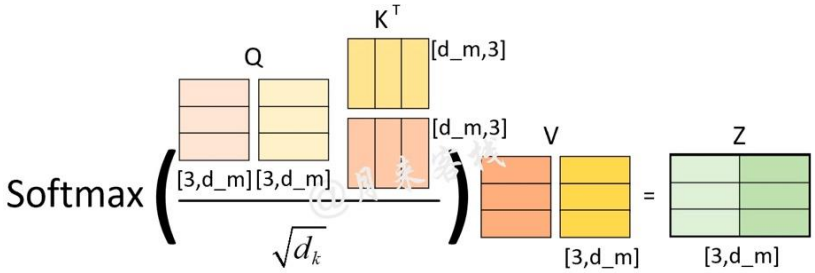

图 1-14. 多头注意力计算过程图

说了这么多，终于把铺垫做完了。此时，假如有如图 1-15 所示的头注意力计算过程：

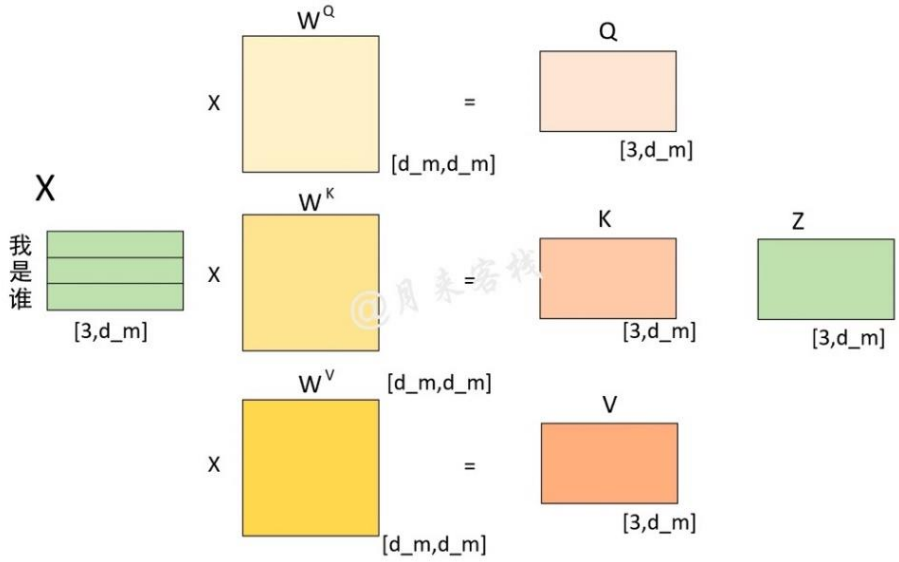

图 1-15. 头注意力计算过程图

如图 1-15 所示，该计算过程采用了头注意力机制来进行计算，且头的计算过程还可通过图 1-16 来进行表示。

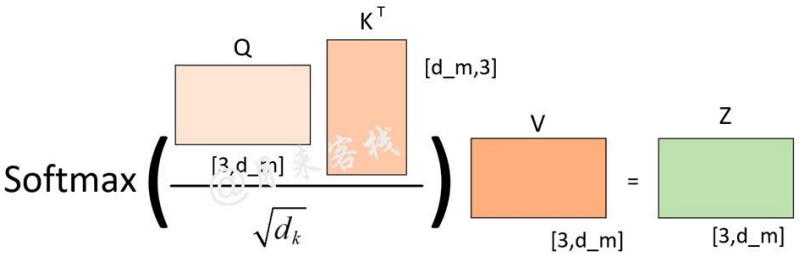

图 1-16. 头注意力机制计算过程题

那现在的问题是图 1-16 中的Z能够计算得到吗？答案是不能。为什么？因为我没有告诉你这里的h等于多少。如果我告诉你多头$h=2$，那么毫无疑问图 1-16 的计算过程就等同于图 1-14 的计算过程，即

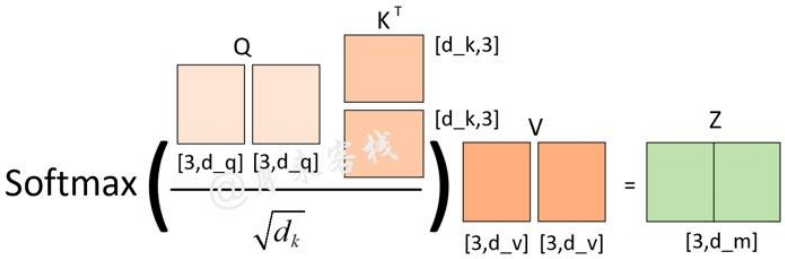

图 1-17. 当$h=2$时注意力计算过程图

且此时$d_k=d_m/2$。但是如果我告诉你多头$h=3$，那么图 1-16 的计算过程会变成

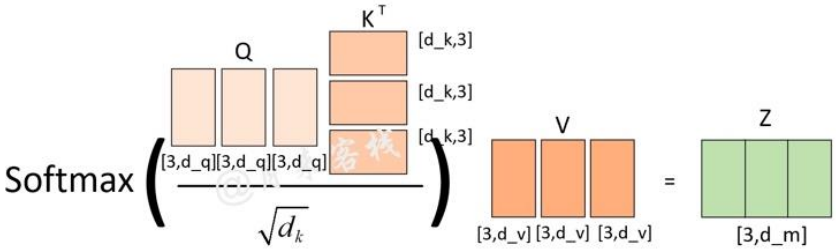

图 1-18. 当$h=3$时注意力计算过程图

那么此时d_k则为$d_m/3$。

现在回到一开始的问题上，根据上面的论述可以发现，在d_m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矩阵计算前，两者之前没有任何区别。当进行注意力权重矩阵计算时，h越大那么Q,K,V就会被切分得越小，进而得到的注意力权重分配方式越多，如图 1-19 所示。

从图 1-19 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作者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。

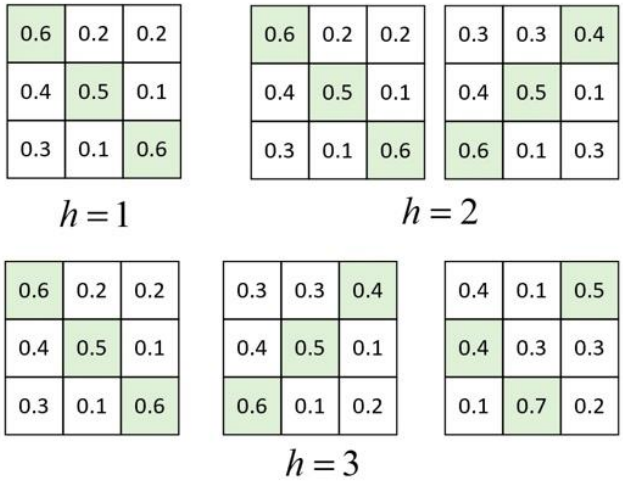

图 1-19. 注意力机制分配图

如图 1-20 所示是一张在真实场景下同一层中的不同注意力权重矩阵（多头）可视化结果图：

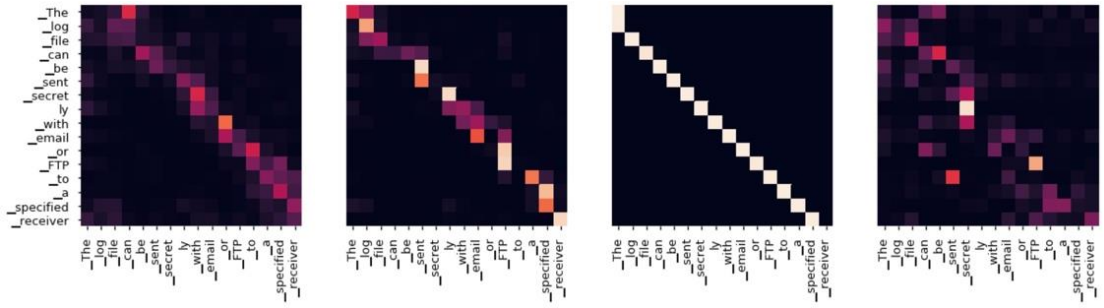

图 1-20. 注意力机制分配图

同时，当 h 不一样时，d_k 的取值也不一样，进而使得对权重矩阵的 scale 的程度不一样。例如，如果 $d_m = 768$，那么当 $h = 12$ 时，则 $d_k = 64$；当 $h = 1$ 时，则 $d_k = 768$。

所以，当模型的维度 d_m 确定时，一定程度上 h 越大，整个模型的表达能力就越强，越能提高模型对于注意力权重的合理分配。

第 2 节 位置编码与编码解码过程

2.1 Embedding 机制

在正式介绍 Transformer 的网络结构之前，我们先来一起看看 Transformer 如何对字符进行 Embedding 处理。

2.1.1 Token Embedding

熟悉文本处理的读者可能都知道，在对文本相关的数据进行建模时首先要做的便是对其进行向量化。例如在机器学习中，常见的文本表示方法有 one-hot 编码、词袋模型以及 TF-IDF 等。不过在深度学习中，更常见的做法便是将各个词（或者字）通过一个 Embedding 层映射到低维稠密的向量空间。因此，在 Transformer 模型中，首先第一步要做的同样是将文本以这样的方式进行向量化表示，并且将其称之为 Token Embedding，也就是深度学习中常说的词嵌入（Word Embedding）如图 2-1 所示。

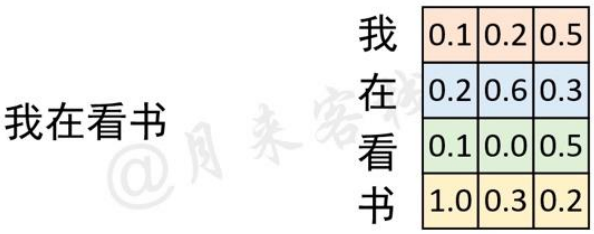

图 2-1. Token Embedding

如果是换做之前的网络模型，例如 CNN 或 RNN，那么对于文本向量化的步骤就到此结束了，因为这些网络结构本身已经具备了捕捉时序特征的能力，不管是 CNN 中的 n-gram 形式还是 RNN 中的时序形式。但是这对仅仅只有自注意力机制的网络结构来说却不行。为什么呢？根据自注意力机制原理的介绍我们知道，自注意力机制在实际运算过程中不过就是几个矩阵来回相乘进行线性变换而已。因此，这就导致即使是打乱各个词的顺序，那么最终计算得到的结果本质上却没有发生任何变换，换句话说仅仅只使用自注意力机制会丢失文本原有的序列信息。

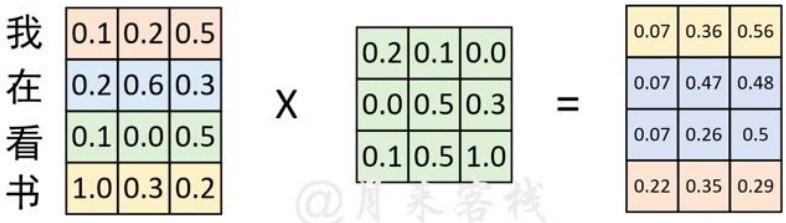

图 2-2. 自注意力机制弊端图（一）

如图 2-2 所示，在经过词嵌入表示后，序列“我 在 看 书”经过了一次线性变换。现在，我们将序列变成“书 在 看 我”，然后同样以中间这个权重矩阵来进行线性变换，过程如图 2-3 所示。

图 2-3. 自注意力机制弊端图（二）

根据图 2-3 中的计算结果来看，序列在交换位置前和交换位置后计算得到的结果在本质上并没有任何区别，仅仅只是交换了对应的位置。因此，基于这样的原因，Transformer 在原始输入文本进行 Token Embedding 后，又额外的加入了一个 Positional Embedding 来刻画数据在时序上的特征。

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence.

2.1.2 Positional Embedding

说了这么多，那到底什么又是 Positional Embedding 呢？数无形时少直觉，下面我们先来通过一幅图直观看看经过 Positional Embedding 处理后到底产生了什么样的变化。

如图 2-4 所示，横坐标表示输入序列中的每一个 Token，每一条曲线或者直线表示对应 Token 在每个维度上对应的位置信息。在左图中，每个维度所对应的位置信息都是一个不变的常数；而在右图中，每个维度所对应的位置信息都是基于某种公式变换所得到。换句话说就是，左图中任意两个 Token 上的向量都可以进行位置交换而模型却不能捕捉到这一差异，但是加入右图这样的位置信息模型却能够感知到。

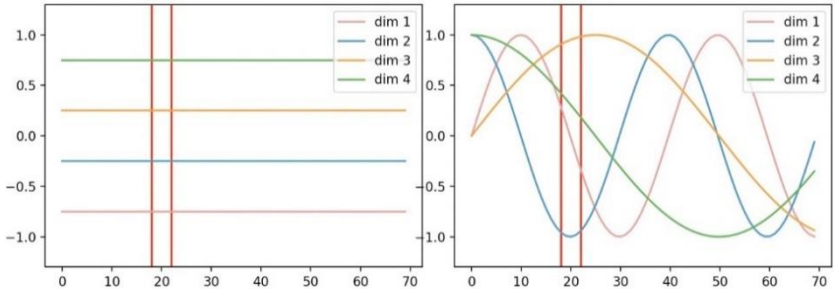

图 2-4. Positional Embedding

如图 2-4 所示，例如位置 20 这一处的向量，在左图中无论你将它换到哪个位置，都和原来一模一样；但在右图中，你却再也找不到与位置 20 处位置信息相同的位置。

下面，掌柜通过两个实际的示例来进行说明。

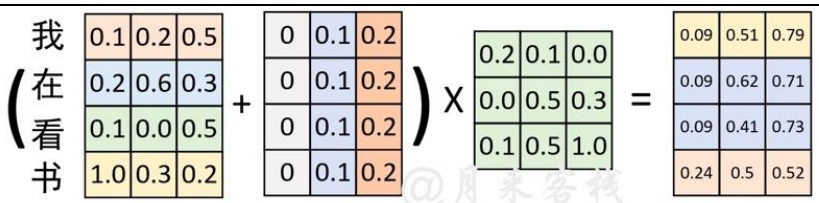

图 2-5. 常数 Positional Embedding（一）

如图 2-5 所示，原始输入在经过 Token Embedding 后，又加入了一个常数位置信息的 Positional Embedding。在经过一次线性变换后便得到了图 2-5 左右边所示的结果。接下来，我们再交换序列的位置，并同时进行 Positional Embedding 观察其结果。

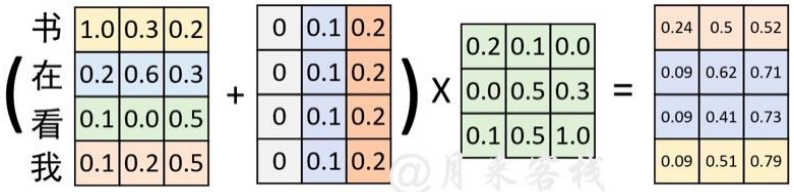

图 2-6. 常数 Positional Embedding（二）

如图 2-6 所示，在交换序列位置后，采用同样的 Positional Embedding 进行处理，并且进行线性变换。可以发现，其计算结果同图 2-5 中的计算结果本质上也没有发生变换。因此，这就再次证明，如果 Positional Embedding 中位置信息是以常数形式进行变换，那么这样的 Positional Embedding 是无效的。

在 Transformer 中，作者采用了如公式(2.1)所示的规则来生成各个维度的位置信息，其可视化结果如图 2-4 右所示。

$$\begin{aligned} PE_{pos,2i} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{pos,2i+1} &= cos(pos/10000^{2i/d_{model}}) \end{aligned} \tag{2.1}$$

其中 PE 就是这个 Positional Embedding 矩阵，$pos \in [0, max\_len)$ 表示具体的某一个位置，$i \in [0, d_{model}/2)$ 表示具体的某一维度。

最终，在融入这种非常数的 Positional Embedding 位置信息后，便可以得到如图 2-7 所示的对比结果。

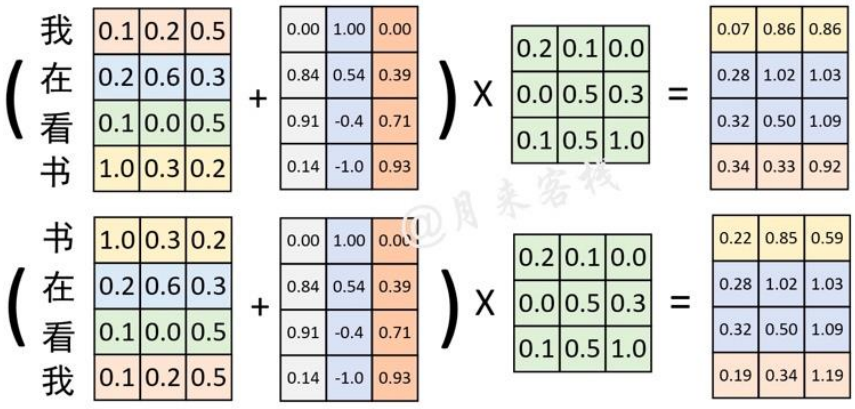

图 2-7. 非常数 Positional Embedding

从图 2-7 可以看出，在交换位置前与交换位置后，与同一个权重矩阵进行线性变换后的结果截然不同。因此，这就证明通过 Positional Embedding 可以弥补自注意力机制不能捕捉序列时序信息的缺陷。

说完 Transformer 中的 Embedding 后，接下来我们再来继续探究 Transformer 的网络结构。

2.2 Transformer 网络结构

如图 2-8 所示便是一个单层 Transformer 网络结构图。

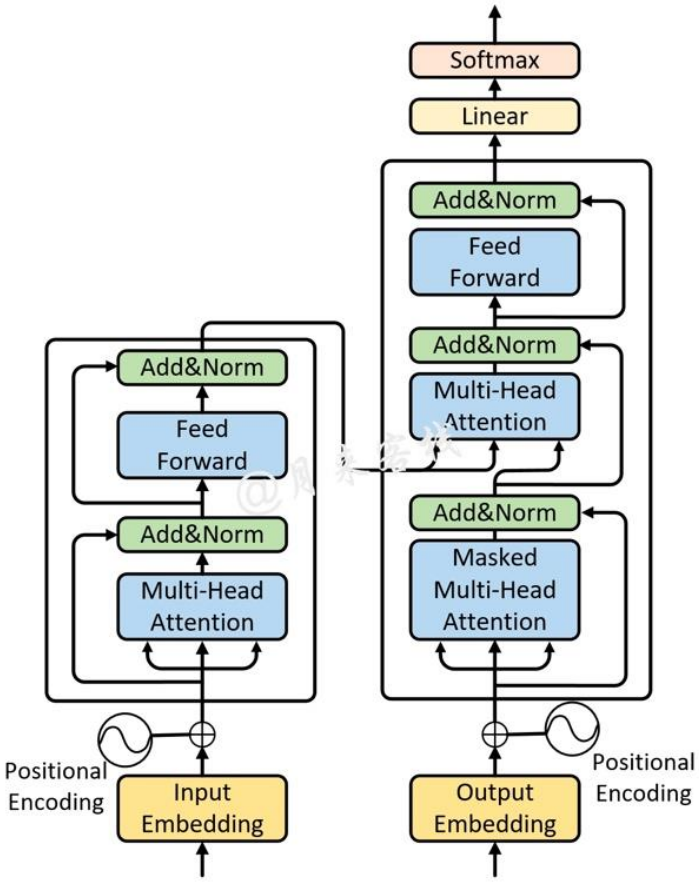

图 2-8. 单层 Transformer 网络结构图

如图 2-8 所示，整个 Transformer 网络包含左右两个部分，即 Encoder 和 Decoder。下面，我们就分别来对其中的各个部分进行介绍。

2.2.1 Encoder 层

首先，对于 Encoder 来说，其网络结构如图 2-8 左侧所示。尽管论文中是通过以 6 个这样相同的模块堆叠而成，但这里掌柜先以堆叠一层来进行介绍，多层的 Transformer 结构将在稍后进行介绍。

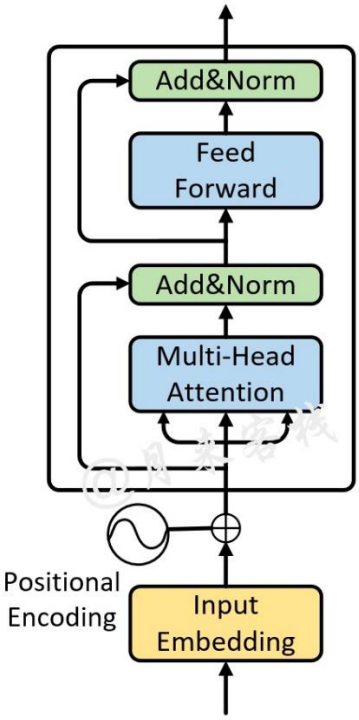

图 2-9. Encoder 网络结构图

如图 2-9 所示，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。

The encoder is composed of a stack of N = 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position- wise fully connected feed-forward network.

同时，对于这两部分网络来说都加入了残差连接，并且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))，并且在都加入了 Dropout 操作。

We apply dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized.

进一步，为了便于在这些地方使用残差连接，这两部分网络输出向量的维度均为 $d_{model}=512$ 。

对于第 2 部分的两层全连接网络来说，其具体计算过程为

$$\mathrm{FFN}(x)=\max(0,xW_1+b_1)W_2+b_2 \tag{2.2}$$

其中输入 x 的维度为 $d_{model}=512$，第 1 层全连接层的输出维度为 $d_{ff}=2048$，第 2 层全连接层的输出为 $d_{model}=512$，且同时对于第 1 层网络的输出还运用了 Relu 激活函数。

到此，对于单层 Encoder 的网络结构就算是介绍完了，接下来让我们继续探究 Decoder 部分的网络结构。

2.2.2 Decoder 层

同 Encoder 部分一样，论文中也采用了 6 个完全相同的网络层堆叠而成，不过这里我们依旧只是先看 1 层时的情况。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，如图 2-10 所示。

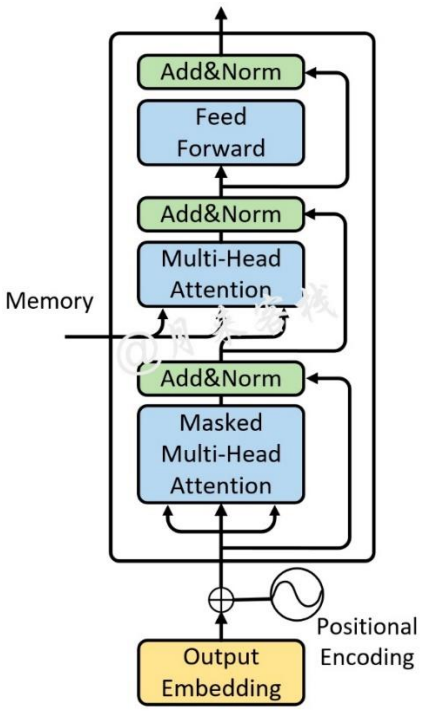

图 2-10. Decoder 网络结构图

不同于 Encoder 部分，在 Decoder 中一共包含有 3 个部分的网络结构。最上面的和最下面的部分（暂时忽略 Mask）与 Encoder 相同，只是多了中间这个与 Encoder 输出（Memory）进行交互的部分，作者称之为“Encoder-Decoder attention”。对于这部分的输入，**Q 来自于下面多头注意力机制的输出，K 和 V 均是 Encoder 部分的输出（Memory）经过线性变换后得到**。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。

In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models.

为了能够更好的理解这里 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，如图 2-11 所示。

如图 2-11 所示是一个经典的基于 Encoder-Decoder 的机器翻译模型。左下边部分为编码器，右下边部分为解码器，左上边部分便是注意力机制部分。在图 2-11 中，$\overline{h_i}$ 表示的是在编码过程中，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编码阶段每个时刻的记忆。

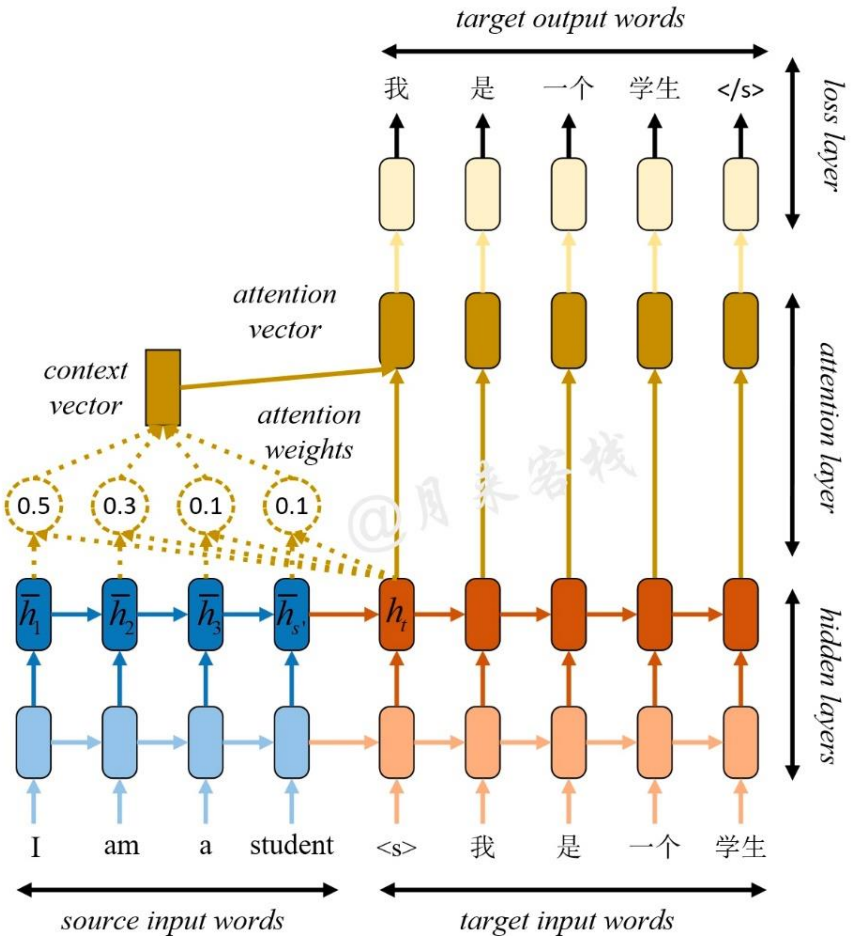

图 2-11. 传统的 Seq2Seq 网络模型图

因此，在解码第一个时刻"<s>"时，h_t 会首先同每个记忆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就是，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（其它的同理），最终通过加权求和得到 4 个 Memory 的权重和，即 context vector。同理，在解码第二时刻"我"时，也会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使得 Encoder 与 Decoder 进行交互的角色。

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），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 对每一个时刻进行解码时，首先需要做的便是通过 Q 与 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。这一过程我们可以通过如图 2-12 和图 2-13 所示的过程来进行表示。

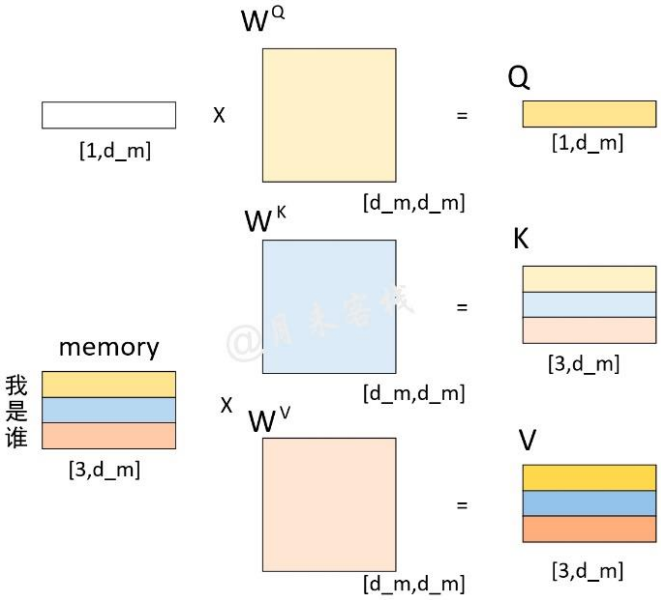

图 2-12. 解码过程 Q、K、V 计算过程图

如图 2-12 所示，待解码向量和 Memory 分别各自乘上一个矩阵后得到 Q、K、V。

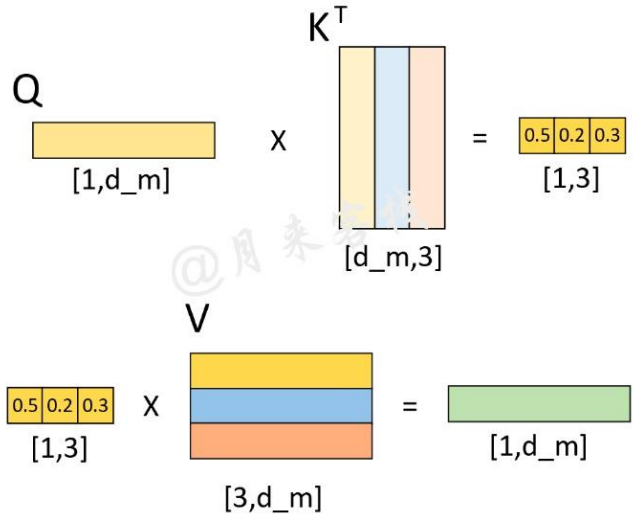

图 2-13. 解码第 1 个时刻输出向量计算过程

如图 2-13 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，此时可以看做是 Q（待解码向量）在 K（本质上也就是 Memory）中查询 Memory 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。进一步，在得到这个解码向量并经过图 2-10 中最上面的两层全连接层后，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。

2.2.3 Decoder 预测解码过程

当第 1 个时刻的解码过程完成之后，解码器便会将解码第 1 个时刻时的输入，以及解码第 1 个时刻后的输出均作为解码器的输入来解码预测第 2 个时刻的输出。整个过程可以通过如图 2-14 所示的过程来进行表示。

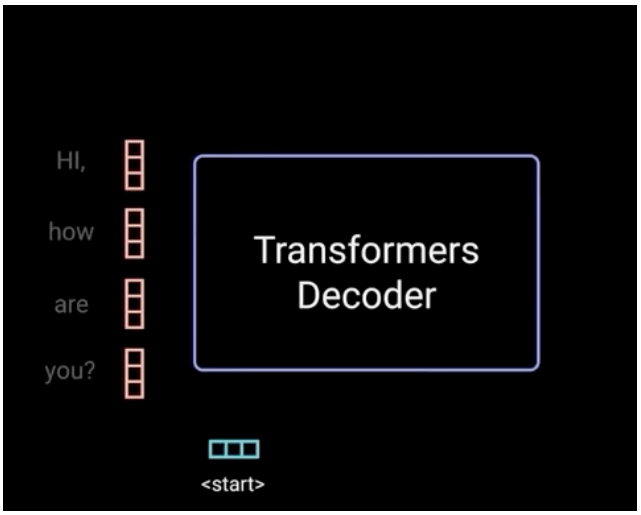

图 2-14. Decoder 多时刻解码过程图（图片来自[3]，可点击可查看动图）

如图 2-14 所示，Decoder 在对当前时刻进行解码输出时，都会将当前时刻之前所有的预测结果作为输入来对下一个时刻的输出进行预测。假设现在需要将"我 是 谁"翻译成英语"who am i"，且解码预测后前两个时刻的结果为"who am"，接下来需要对下一时刻的输出"i"进行预测，那么整个过程就可以通过图 2-15 和图 2-16 来进行表示。

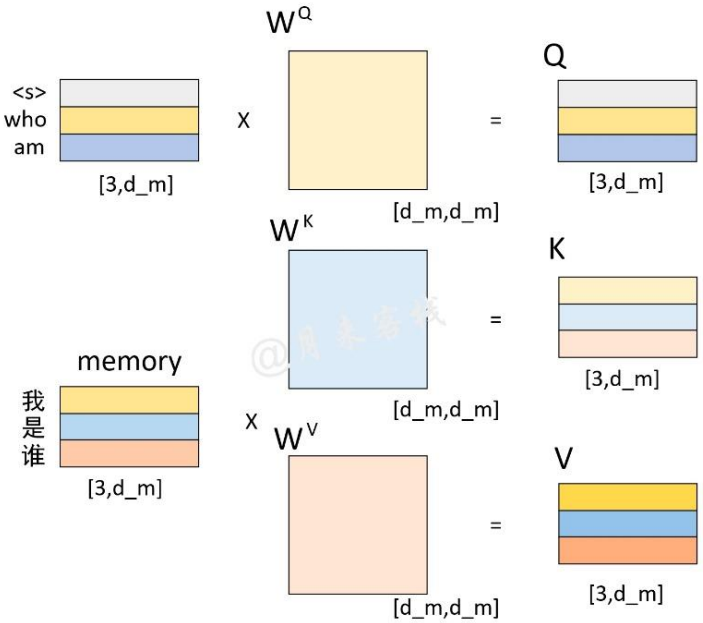

图 2-15. 解码过程中 Q、K、V 计算过程图

如图 2-15 所示，左上角的矩阵是解码器对输入"<s> who am"这 3 个词经过解码器中自注意力机制编码后的结果；左下角是编码器对输入"我 是 谁"这 3 个词编码后的结果（同图 2-12 中的一样）；两者分别在经过线性变换后便得到了 Q、K 和 V 这 3 个矩阵。此时值得注意的是，左上角矩阵中的每一个向量在经过自注意力机制编码后，每个向量同样也包含了其它位置上的编码信息。

进一步，Q 与 K 作用和便得到了一个权重矩阵；再将其与 V 进行线性组合便得到了 Encoder-Decoder attention 部分的输出，如图 2-16 所示。

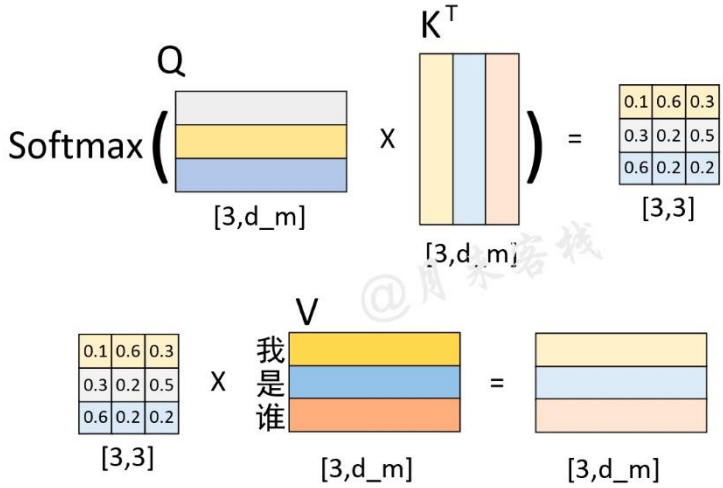

图 2-16. 解码第 3 个时刻输出向量计算过程

如图 2-16 所示，左下角便是 Q 与 K 作用后的权重矩阵，它的每一行就表示在对 Memory（这里指图 2-16 中的 V）中的每一位置进行解码时，应该如何对注意力进行分配。例如第 3 行[0.6,0.2,0.2]的含义就是在解码当前时刻时应该将 60%的注意力放在 Memory 中的"我"上，其它同理。这样，在经过解码器中的两个全连接层后，便得到了解码器最终的输出结果。接着，解码器会循环对下一个时刻的输出进行解码预测，直到预测结果为"<e>"或者达到指定长度后停止。

同时，这里需要注意的是，在通过模型进行实际的预测时，只会取解码器输出的其中一个向量进行分类，然后作为当前时刻的解码输出。例如图 2-16 中解码器最终会输出一个形状为[3,tgt_vocab_len]的矩阵，那么只会取其最后一个向量喂入到分类器中进行分类得到当前时刻的解码输出。具体细节见后续代码实现。

2.2.4 Decoder 训练解码过程

在介绍完预测时 Decoder 的解码过程后，下面就继续来看在网络在训练过程中是如何进行解码的。从 2.2.3 小节的内容可以看出，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，如果训练时也采用同样的方法那将是十分费时的。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，一是通过多样本并行计算能够加快网络的训练速度；二是在训练过程中直接喂入解码器正确的结果而不是上一时刻的预测值（因为训练时上一时刻的预测值可能是错误的）能够更好的训练网络。一般来说，这中类 Encoder-

Decoder 的网络结构，模型在训练过程中 Decoder 输入的并不是上一个时刻预测结果，而是直接喂给它正确的结果，而这也是初学者经常感到困惑的地方。

例如在用平行预料"我 是 谁"<==>"who am i"对网络进行训练时，编码器的输入便是"我 是 谁"，而解码器的输入则是"<s> who am i"，对应的正确标签则是"who am i <e>"。

假设现在解码器的输入"<s> who am i"在分别乘上一个矩阵进行线性变换后得到了 Q、K、V，且 Q 与 K 作用后得到了注意力权重矩阵（此时还未进行 softmax 操作），如图 2-17 所示。

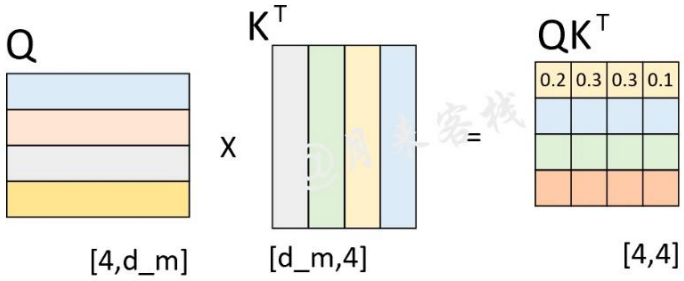

图 2-17. 解码器输入权重矩阵计算过程图

从图 2-17 可以看出，此时已经计算得到了注意力权重矩阵。由第 1 行的权重向量可知，在解码第 1 个时刻时应该将 20%（严格来说应该是经过 softmax 后的值）的注意力放到"<s>"上，30%的注意力放到"who"上等等。不过此时有一个问题就是，在 2.2.3 节中掌柜介绍到，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个时刻，也就是说模型在预测时是看不到当前时刻之后的信息。因此，Transformer 中的 Decoder 通过加入注意力掩码机制来解决了这一问题。

self-attention layers in the decoder ***allow each position in the decoder to attend to all positions in the decoder up to and including that position****. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to* $-\infty$*) all values in the input of the softmax which correspond to illegal connections.*

如图 2-18 所示，左边依旧是通过 Q 和 K 计算得到了注意力权重矩阵（此时还未进行 softmax 操作），而中间的就是所谓的注意力掩码矩阵，两者在相加之后再乘上矩阵 V 便得到了整个自注意力机制的输出，也就是图 2-10 中的 Masked Multi-Head Attention。

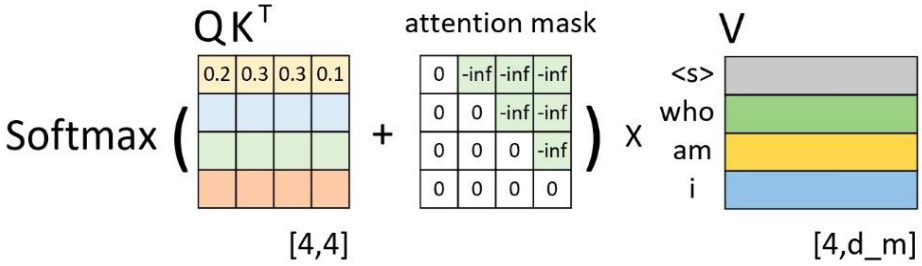

图 2-18. 注意力掩码计算过程图

那为什么注意力权重矩阵加上这个注意力掩码矩阵就能够达到这样的效果呢？以图 2-18 中第 1 行权重为例，当解码器对第 1 个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。从图 2-17 可以看出，第 1 行注意力向量在加上第 1 行注意力掩码，再经过 softmax 操作后便得到了一个类似[1,0,0,0,0]的向量。那么，通过这个向量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。同理，在解码后续的时刻也是类似的过程。

到此，对于整个单层 Transformer 的网络结构以及编码解码过程就介绍完了，更多细节内容见后续代码实现。

2.2.5 位置编码与 Attention Mask

在刚接触 Transformer 的时候，有的人会认为在 Decoder 中，既然已经有了 Attention mask 那么为什么还需要 Positional Embedding 呢？如图 2-18 所示，持这种观点的朋友认为，Attention mask 已经有了使得输入序列依次输入解码器的能力，因此就不再需要 Positional Embedding 了。这样想对吗？

根据 2.2.4 节内容的介绍可以知道，Attention mask 的作用只有一个，那就是在训练过程中掩盖掉当前时刻之后所有位置上的信息，而这也是在模仿模型在预测时只能看到当前时刻及其之前位置上的信息。因此，持有上述观点的朋友可能是把“能看见”和“能看见且有序”混在一起了。

虽然看似有了 Attention mask 这个掩码矩阵能够使得 Decoder 在解码过程中可以**有序地**看到当前位置之前的所有信息，但是事实上没有 Positional Embedding 的 Attention mask 只能做到看到当前位置之前的所有信息，**而做不到有序**。前者的“有序”指的是喂入解码器中序列的顺序，而后者的“有序”指的是序列本身固有的语序。

如果不加 Postional Embedding，那么以下序列对于模型来说就是一回事：

<s> → 北 → 京 → 欢 → 迎 → 你 → <e>
<s> → 北 → 京 → 迎 → 欢 → 你 → <e>
<s> → 北 → 京 → 你 → 迎 → 欢 → <e>

虽然此时 Attention mask 具有能够让上述序列一个时刻一个时刻的按序喂入到解码器中，但是它却无法识别出这句话本身固有的语序。

2.2.6 原始 Q、K、V 来源

在 Transformer 中各个部分的 Q、K、V 到底是怎么来的一直以来都是初学者最大的一个疑问，并且这部分内容在原论文中也没有进行交代，只是交代了如何根据 Q、K、V 来进行自注意力机制的计算。虽然在第 2 部分的前面几个小节已经提及过了这部分内容，但是这里再给大家进行一次总结。

从图 2-8（Transformer 结构图）可知，整个 Transformer 中涉及到自注意力机制的一共有 3 个部分：Encoder 中的 Multi-Head Attention；Decoder 中的 Masked Multi-Head Attention；Encoder 和 Decoder 交互部分的 Multi-Head Attention。

① 对于 Encoder 中的 Multi-Head Attention 来说，其原始 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（也就是图 1-4 中的示例），然后再进行自注意力运算得到 Encoder 部分的输出结果 Memory。

② 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运算得到 Masked Multi-Head Attention 部分的输出结果，即待解码向量。

③ 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（也就是图 2-12 中的示例），然后再进行自注意力运算得到 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。

第 3 节 网络结构与自注意力实现

在通过前面几部分内容详细介绍完 Transformer 网络结构的原理后，接下来就让我们来看一看如何借用 Pytorch 框架来实现 MultiHeadAttention 这一结构。同时，需要说明的一点是，下面所有的实现代码都是掌柜直接从 Pytorch 1.4 版本中 torch.nn.Transformer 模块里摘取出来的简略版，目的就是为了让大家对于整个实现过程有一个清晰的认识。并且为了使得大家在阅读完以下内容后也能够对 Pytorch 中的相关模块有一定的了解，所以下面的代码在变量名方面也与 Pytorch 保持了一致。

代码仓库见：**https://github.com/moon-hotel/TransformerTranslation**

3.1 多层 Transformer

在第 2 部分中，掌柜详细介绍了单层 Transformer 网络结构中的各个组成部分。尽管多层 Transformer 就是在此基础上堆叠而来，不过掌柜认为还是有必要在这里稍微提及一下。

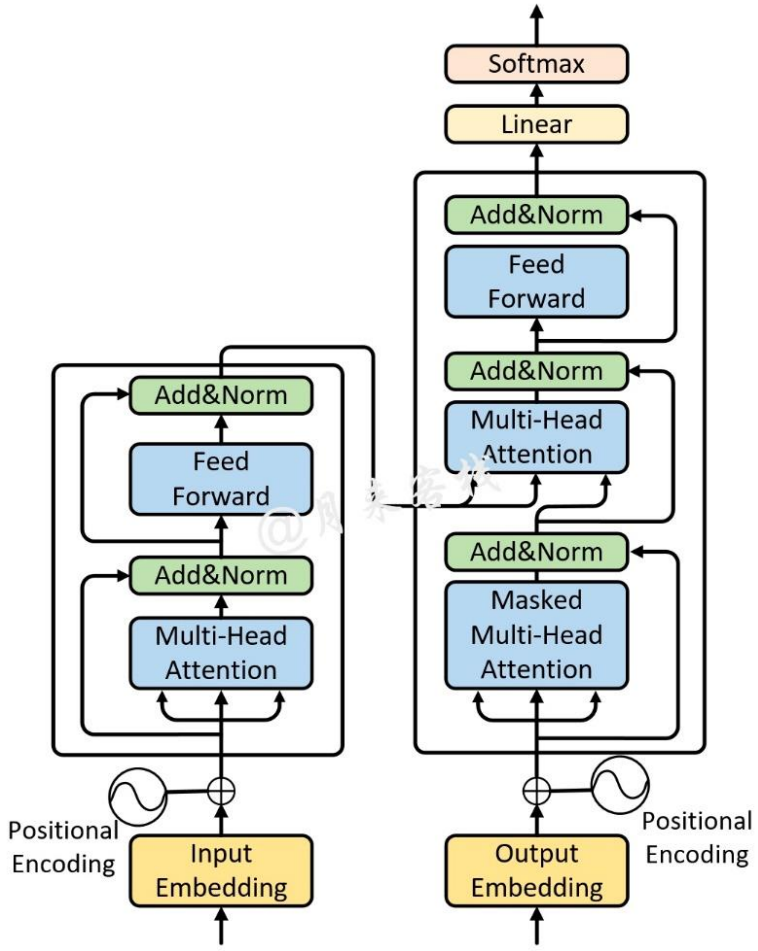

图 3-1. 单层 Transformer 网络结构图

如图 3-1 所示便是一个单层 Transformer 网络结构图，左边是编码器右边是解码器。而多层的 Transformer 网络就是在两边分别堆叠了多个编码器和解码器的网络模型，如图 3-2 所示。

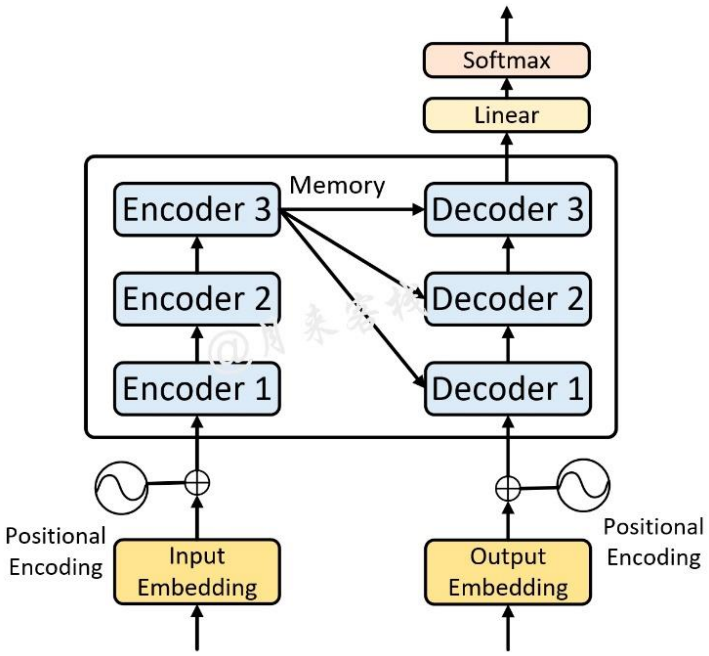

图 3-2. 多层 Transformer 网络结构图

如图 3-2 所示便是一个多层的 Transformer 网络结构图（原论文中采用了 6 个编码器和 6 个解码器），其中的每一个 Encoder 都是图 3-1 中左边所示的网络结构（Decoder 同理）。可以发现，它真的就是图 3-1 堆叠后的形式。不过需要注意的是其整个解码过程。

在多层 Transformer 中，多层编码器先对输入序列进行编码，然后得到最后一个 Encoder 的输出 Memory；解码器先通过 Masked Multi-Head Attention 对输入序列进行编码，然后将输出结果同 Memory 通过 Encoder-Decoder Attention 后得到第 1 层解码器的输出；接着再将第 1 层 Decoder 的输出通过 Masked Multi-Head Attention 进行编码，最后再将编码后的结果同 Memory 通过 Encoder-Decoder Attention 后得到第 2 层解码器的输出，以此类推得到最后一个 Decoder 的输出。

值得注意的是，在多层 Transformer 的解码过程中，每一个 Decoder 在 Encoder-Decoder Attention 中所使用的 Memory 均是同一个。

3.2 Transformer 中的掩码

由于在实现多头注意力时需要考虑到各种情况下的掩码，因此在这里需要先对这部分内容进行介绍。在 Transformer 中，主要有两个地方会用到掩码这一机制。第 1 个地方就是在 2.2.4 节中介绍到的 Attention Mask，只用于在训练过程中解码阶段时掩盖掉当前时刻之后的信息；第 2 个地方便是对一个 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，这在编码器和解码器中均会用到。下面分别就这两种情况进行介绍。

3.2.1 Attention Mask

如图 3-3 所示，在训练过程中对于每一个样本来说都需要这样一个对称矩阵来掩盖掉当前时刻之后所有位置的信息。

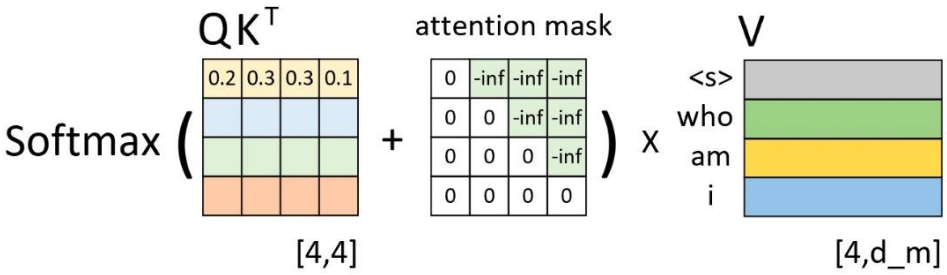

图 3-3. 解码器自注意力计算过程图

从图 3-3 可以看出，这个注意力掩码矩阵的形状为[tgt_len,tgt_len]，其具体 Mask 原理在 2.2.4 节中掌柜已经介绍过 l，这里就不再赘述。在后续实现过程中，我们将通过 generate_square_subsequent_mask 方法来生成这样一个矩阵。同时，在后续多头注意力机制实现中，将通过 attn_mask 这一变量名来指代这个矩阵。

3.2.2 Padding Mask

在 Transformer 中，使用到掩码的第 2 个地方便是 Padding Mask。由于在网络的训练过程中同一个 batch 会包含有多个文本序列，而不同的序列长度并不一致。因此在数据集生成时，就需要将同一个 batch 中的序列 Padding 到相同的长度。但是，这样就会导致在注意力的计算过程中会考虑到 Padding 位置上的信息。

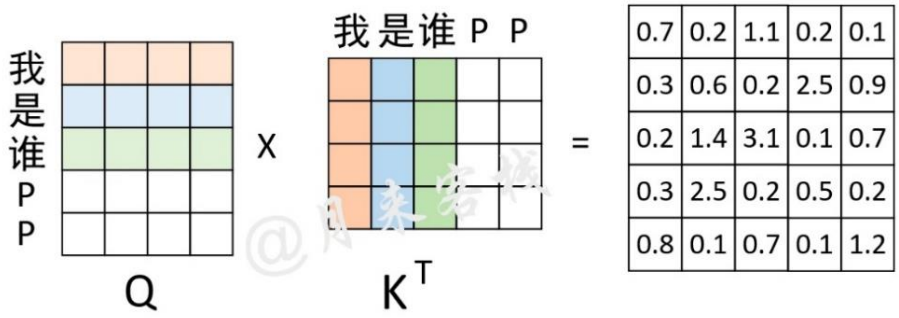

图 3-4. Padding 时注意力计算过程图

如图 3-4 所示，P 表示 Padding 位置，右边的矩阵表示注意力权重矩阵。从图中可以看到，此时注意力权重对于 Padding 位置上的信息也会加以考虑。因此在 Transformer 中，作者通过在生成训练集的过程中记录下每个样本 Padding 的实际位置；然后再将注意力权重矩阵中对应位置的权重替换成负无穷，经 softmax 操作后对应 Padding 位置上的权重就变成了 0，从而达到了忽略 Padding 位置信息的目的。这种做法也是 Encoder-Decoder 网络结构中通用的一种办法。

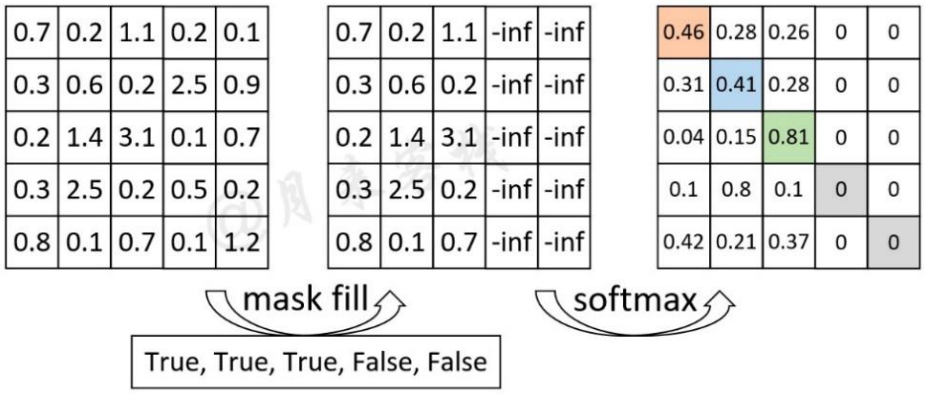

图 3-5. Padding 掩码计算过程图

如图 3-5 所示，对于"我 是 谁 P P"这个序列来说，前 3 个字符是正常的，后 2 个字符是 Padding 后的结果。因此，其 Mask 向量便为 [True, True, True, False, False]。通过这个 Mask 向量可知，需要将权重矩阵的最后两列替换成负无穷，在后续我们会通过 torch.masked_fill 这个方法来完成这一步，并且在实现时将使用 key_padding_mask 来指代这一向量。

3.2.3 行 Padding 与列 Padding

在 3.2.2 节中掌柜介绍到可以通过 mask fill 操作来忽略掉注意力权重矩阵中 padding 部分的信息，即图 3-5 中的示例。但此时有好事者发出疑问，图 3-5 中注意力权重的最后两行为什么没有进行 mask 操作？这两行对应的不也是 padding 部分的信息吗？此时，问题就来了，为什么不对注意力权重矩阵中的最后两行也进行 mask 处理（如图 3-6 右所示），它代表的不也是 padding 部分的信息吗？

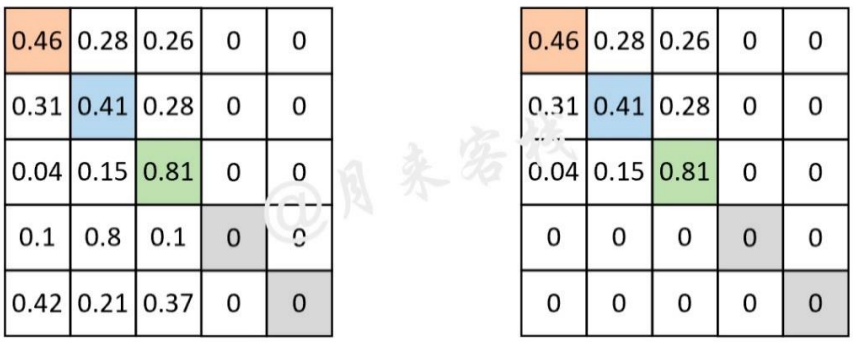

图 3-6. 行 padding 与列 padding 对比图

这里之所以不用对最后两行的 padding 进行 mask 处理，是因为权重矩阵与 V 作用后得到的结果 Memory 在与 Decoder 中 Multi-Head Attention 交互时，最后两行 padding 部分的内容在那里进行了 mask 处理。下面，掌柜就用图示来一步一步还原整个过程，以便大家能够清楚地理解。

假如图 3-6 中左边的权重矩阵与 V 作用后的结果如图 3-7 所示，此时可以知道 Z 中最后两行的信息本质上也是属于 padding 部分的内容，按道理来说如果权重矩阵的最后两行全为 0 ，那么 Z 中最后两行的结果也将全部为 0 而被忽略掉。

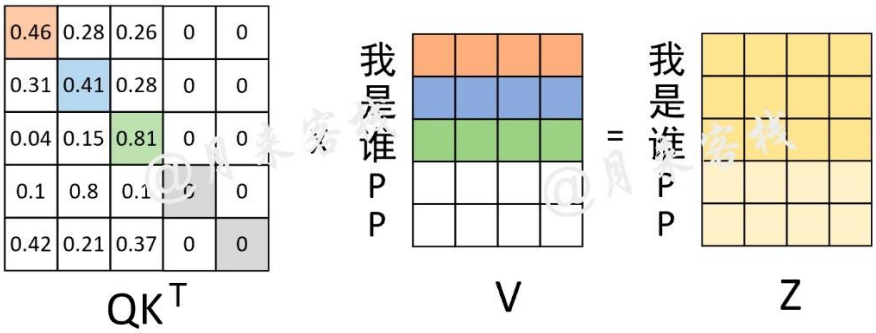

图 3-7. encoder 中单头注意力权重计算过程图

进一步，假定解码器的输入经过图 3-1 中 Masked Multi-Head Attention 处理后的输出为 tgt(Q)，与编码器的输出 Memory（图 3-7 中 Z 经过线性变换后的结果）作用后得到权重矩阵如图 3-8 最右边所示。按道理，如果图 3-7 中 Z 的最后两行均为 0，那么图 3-8 中权重矩阵的最后两列将会全部为 0。

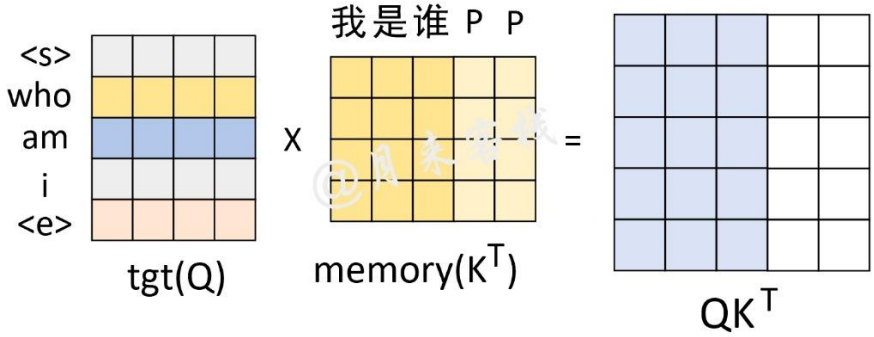

图 3-8. decoder 中单头注意力权重矩阵计算过程图

接着，图 3-8 中的注意力矩阵将会进行类似图 3-5 中的 mask 操作，而此时 encoder-decoder 交互部分的 maks 向量就是图 3-5 中的 mask 向量。因此，整个 mask 过程如图 3-9 所示。

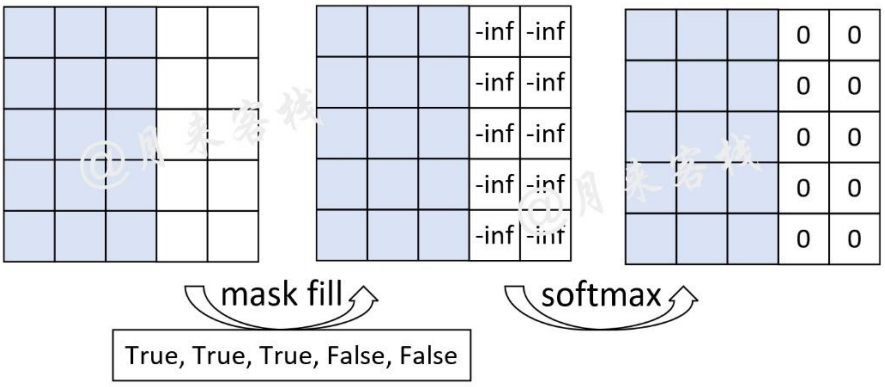

图 3-9. encoder-decoder 单头注意力 mask 计算过程图

如图 3-9 所示，图 3-8 中的注意力矩阵在经过 mask 操作后，注意力矩阵的最后两列全部变成了 0，从而也就达到了后面可以忽略掉图 3-7 里 Z 中最后两行信息的目的，如图 3-10 所示。

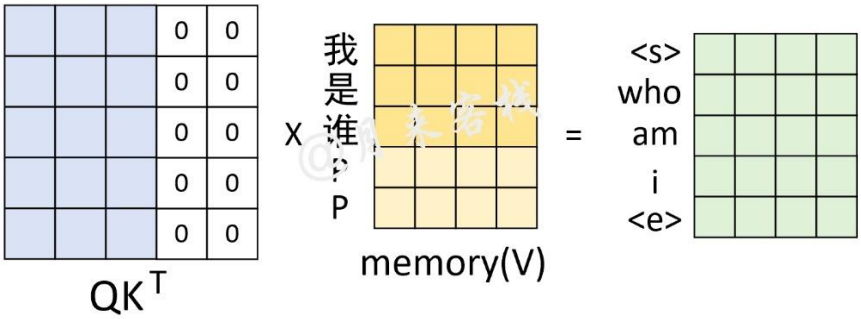

图 3-10. encoder-decoder 单头注意力计算过程图

如图 3-10 所示，由于此时注意力矩阵的最后两列均为 0，那么其在与 encoder 输出部分 memory 作用时，memory 中最后两行 padding 部分的信息自然就得到了忽略。此时可以看出，即使在 encoder 里图 3-7 中权重矩阵的最后两行没有进行 mask，但是在 encoder-decoder 交互时的 mask 操作中却得到了体现。不过，假如图 3-7 中权重矩阵的最后两行也进行了 mask，那又会是一个什么样的结果呢？

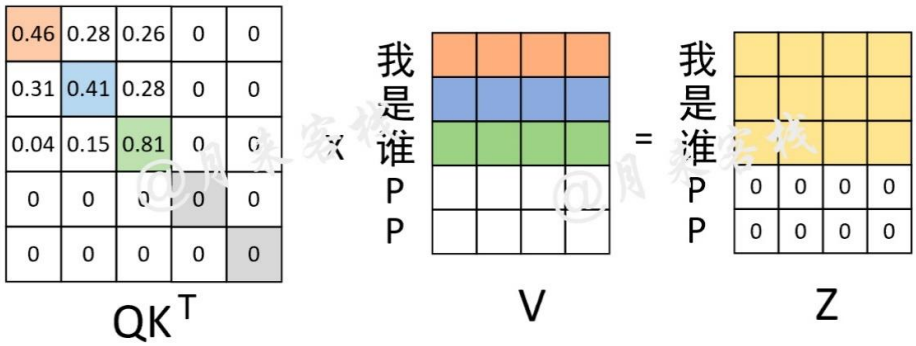

图 3-11. encoder 中单头注意力计算过程图

如图 3-11 所示，假如此时权重矩阵的最后两行也进行了 padding 处理，那么此时 encoder 单头注意力的输出结果将形如 Z 的形式，即 padding 部分的信息全为 0。进一步，图 3-11 中 Z 经线性变换得到 memory(K)后，与解码器的输入经过 Masked Multi-Head Attention 处理后的输出 tgt(Q)作用后的结果将如图 3-12 所示。

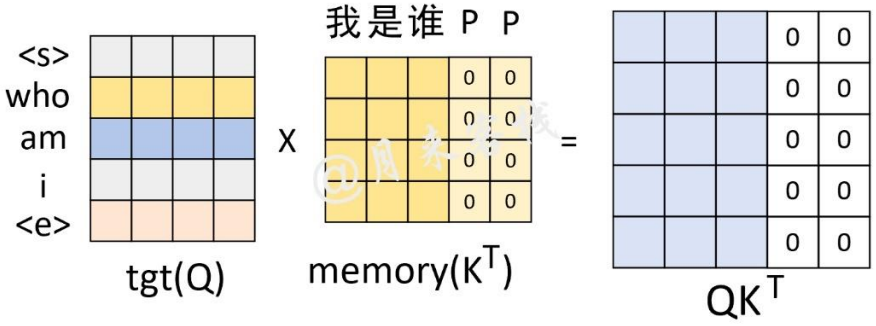

图 3-12. decoder 中单头注意力权重矩阵计算过程图

如图 3-12 所示，由于此时 memory 中最后两行的信息也进行了 mask 处理，所以计算得到的注意力权重矩阵最后两列便直接变成了 0，从而也直接得到了图 3-9 中 mask 操作后的结果，后续便再是图 3-10 所示的过程。

到此可以得出的结论便是，在 encoder 中无论是否要在行上注意力权重矩阵进行 mask 操作，都不会影响模型最后的输出结果。因为即使在 encoder 中注意力权重矩阵行上的 padding 信息没有被 mask，但是在 encoder-decoder 交互部分的注意力权重矩阵计算过程中也会对这部分信息进行 mask 处理。

不过最后的最后，掌柜还要再提出一个类似的问题，在解码器的 Masked Multi-Head Attention 中，注意力权重矩阵同样也只是在列上进行了 mask 处理，那为什么同样没有在行上进行 mask 操作呢？或者说后续阶段哪个操作等价于行上的 mask 操作呢？如果想不明白可以公众号后台回复“行 padding”获取答案。

到此，对于 Transformer 中所要用到 Mask 的地方就介绍完了，下面正式来看如何实现多头注意力机制。

3.3 实现多头注意力机制

3.3.1 多头注意力机制

根据前面的介绍可以知道，多头注意力机制中最为重要的就是自注意力机制，也就是需要前计算得到 Q、K 和 V，如图 3-13 所示。

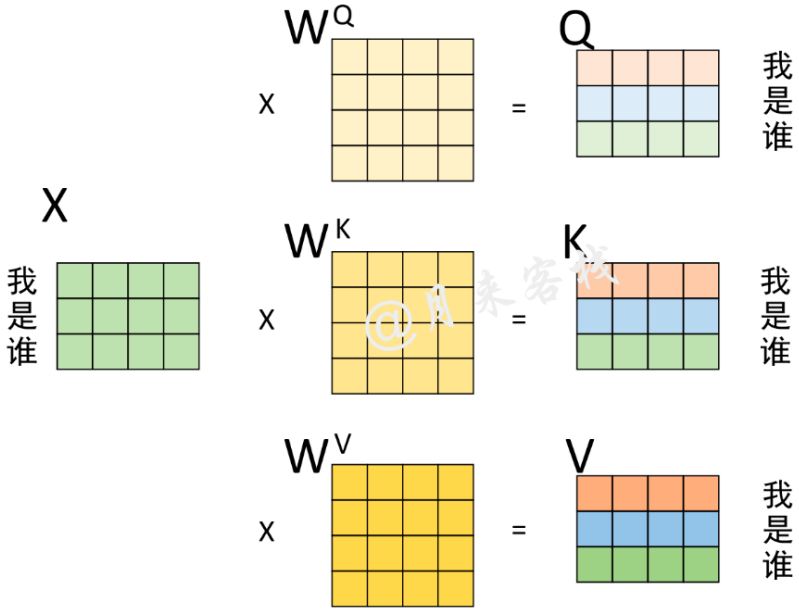

图 3-13. Q、K 和 V 计算过程

然后再根据 Q、K、V 来计算得到最终的注意力编码，如图 3-14 所示。

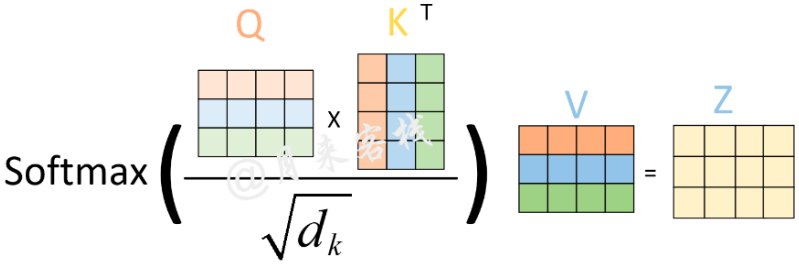

图 3-14. 注意力编码计算图

同时，为了避免单个自注意力机制计算得到的注意力权重过度集中于当前编码位置自己所在的位置（同时更应该关注于其它位置），所以作者在论文中提到通过采用多头注意力机制来解决这一问题，如图 3-15 所示。

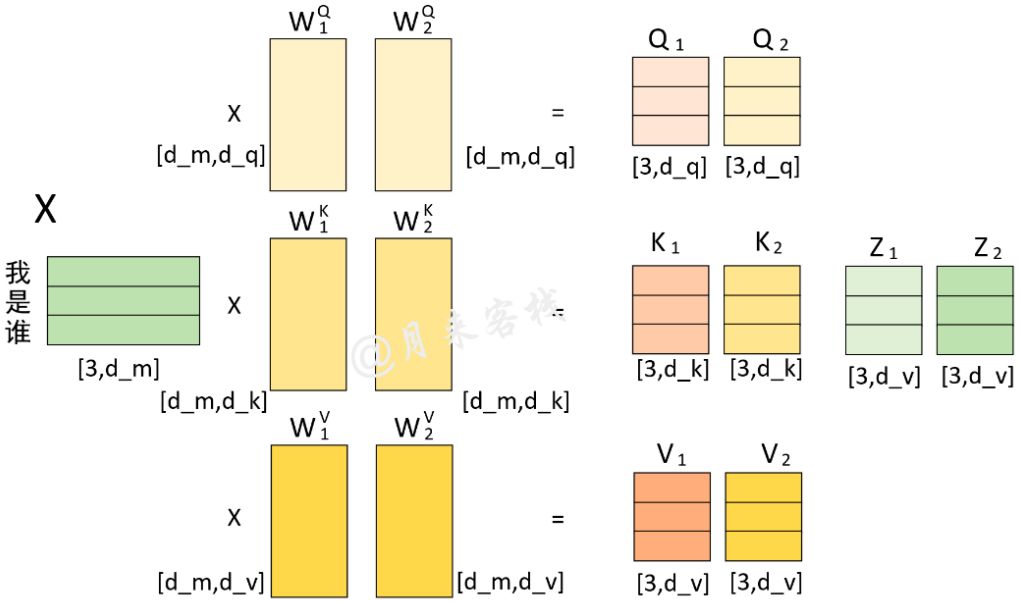

图 3-15. 多头注意力计算图（2 个头）

3.3.2 定义类 MyMultiHeadAttention

综上所述，我们可以给出类 MyMultiHeadAttentiond 的定义为

```
class MyMultiheadAttention(nn.Module):
    def __init__(self,embed_dim, num_heads, dropout=0.,bias=True):
        super(MyMultiheadAttention, self).__init__()
        """
        :param embed_dim: 词嵌入的维度，也就是前面的 d_model 参数,论文中的默认值为 512
        :param num_heads:多头注意力中多头的数量,也就是前面的 nhead 参数,论文默认值为 8
        :param bias:    最后对多头的注意力（组合）输出进行线性变换时，是否使用偏置
        """
        self.embed_dim = embed_dim  # 前面的 d_model 参数
        self.head_dim = embed_dim //num_heads # head_dim 指的就是 d_k,d_v
        self.kdim = self.head_dim
        self.vdim = self.head_dim
        self.num_heads = num_heads  # 多头个数
        self.dropout = dropout
        assert self.head_dim * num_heads == self.embed_dim,
                                            "embed_dim 除以 num_heads 必须为整数"
        # 上面的限制条件就是论文中的  d_k = d_v = d_model/n_head 条件
        self.q_proj_weight = Parameter(torch.Tensor(embed_dim, embed_dim))
        # embed_dim = kdim * num_heads
        # 这里第二个维度之所以是 embed_dim，实际上这里是同时初始化了 num_heads 个 W_q 堆
        叠起来的，也就是 num_heads 个头
```

```
20          self.k_proj_weight = Parameter(torch.Tensor(embed_dim, embed_dim))
21          # W_k,  embed_dim = kdim * num_heads
22          self.v_proj_weight = Parameter(torch.Tensor(embed_dim, embed_dim))
23          # W_v,  embed_dim = vdim * num_heads
24          self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
25          # 最后将所有的 Z 组合起来的时候，也是一次性完成， embed_dim = vdim * num_heads
```

在上述代码中，embed_dim 表示模型的维度（图 3-15 中的 d_m）；num_heads 表示多头的个数；bias 表示是否在多头线性组合时使用偏置。同时，为了使得实现代码更加高效，所以 Pytorch 在实现的时候是多个头注意力机制一起进行的计算，也就上面代码的第 17-22 行，分别用来初始化了多个头的权重值（这一过程从图 3-15 也可以看出）。当多头注意力机制计算完成后，将会得到一个形状为[src_len,embed_dim]的矩阵，也就是图 3-15 中多个 z_i 水平堆叠后的结果。因此，第 24 行代码将会初始化一个线性层来对这一结果进行一个线性变换。

3.3.3 定义前向传播过程

在定义完初始化函数后，便可以定义如下所示的多头注意力前向传播的过程

```
1     def forward(self, query,key,value,attn_mask = None,
                  key_padding_mask=None):
2         """
3         在论文中，编码时 query, key, value 都是同一个输入，
4         解码时 输入的部分也都是同一个输入，
5         解码和编码交互时 key,value 指的是 memory, query 指的是 tgt
6         :param query: [tgt_len, batch_size, embed_dim], tgt_len 表示目标序列的长度
7         :param key:   [src_len, batch_size, embed_dim], src_len 表示源序列的长度
8         :param value: [src_len, batch_size, embed_dim], src_len 表示源序列的长度
9         :param attn_mask:[tgt_len,src_len] or [num_heads*batch_size,tgt_len, src_len]
10          一般只在解码时使用，为了并行一次喂入所有解码部分的输入，所以要用 mask 来进行掩盖
              当前时刻之后的位置信息
11        :param key_padding_mask: [batch_size, src_len], src_len 表示源序列的长度
12        :return:
13        attn_output: [tgt_len, batch_size, embed_dim]
14        attn_output_weights: # [batch_size, tgt_len, src_len]
15        """
16          return multi_head_attention_forward(query, key,value,self.num_heads,
17                                  self.dropout,self.out_proj.weight,
                                    self.out_proj.bias, training=self.training,
19                                  key_padding_mask=key_padding_mask,
20                                  q_proj_weight=self.q_proj_weight,
21                                  k_proj_weight=self.k_proj_weight,
22                                  v_proj_weight=self.v_proj_weight,
23                                  attn_mask=attn_mask)
```

在上述代码中，query、key、value 指的并不是图 3-13 中的 Q、K 和 V，而是没有经过线性变换前的输入。例如在编码时三者均是指原始输入序列 src；在解码时的 Mask Multi-Head Attention 中三者指的均是目标输入序列 tgt；在解码时的 Encoder-Decoder Attention 中三者分别指的是 Mask Multi-Head Attention 的输出、Memory 和 Memory。key_padding_mask 指的是编码或解码部分，输入序列的 Padding 情况，形状为 [batch_size,src_len] 或者[batch_size,tgt_len];attn_mask 指的就是注意力掩码矩阵，形状为 [tgt_len,src_len]，**它只会在解码**时使用。

注意，在上面的这些维度中，tgt_len 本质上指的其实是 query_len；src_len 本质上指的是 key_len。只是在不同情况下两者可能会是一样，也可能会是不一样。

3.3.4 多头注意力计算过程

在定义完类 MyMultiHeadAttentiond 后，就需要定义出多头注意力的实际计算过程。由于这部分代码较长，所以就分层次进行介绍。

```
1  def multi_head_attention_forward(
2      query,  # [tgt_len,batch_size, embed_dim]
3      key,    # [src_len, batch_size, embed_dim]
4      value,  # [src_len, batch_size, embed_dim]
5      num_heads,
6      dropout_p,
7      out_proj_weight, # [embed_dim = vdim * num_heads, embed_dim]
8      out_proj_bias,
9      training=True,
10     key_padding_mask=None,  # [batch_size,src_len/tgt_len]
11     q_proj_weight=None,  # [embed_dim,kdim * num_heads]
12     k_proj_weight=None,  # [embed_dim, kdim * num_heads]
13     v_proj_weight=None,  # [embed_dim, vdim * num_heads]
14     attn_mask=None,  # [tgt_len,src_len]
15     ):
16     # 第一阶段： 计算得到Q、K、V
17     q = F.linear(query, q_proj_weight)
18     #  [tgt_len,batch_size,embed_dim] x [embed_dim,kdim * num_heads]
19     #  = [tgt_len,batch_size,kdim * num_heads]
20     k = F.linear(key, k_proj_weight)
21     # [src_len, batch_size,embed_dim] x [embed_dim,kdim * num_heads]
22     # = [src_len,batch_size,kdim * num_heads]
23     v = F.linear(value, v_proj_weight)
24     # [src_len, batch_size,embed_dim] x [embed_dim,vdim * num_heads]
25     # = [src_len,batch_size,vdim * num_heads]
```

在上述代码中，第 17-23 行所做的就是根据输入进行线性变换得到图 3-13 中的 Q、K 和 V。

```
1     # 第二阶段： 缩放，以及 attn_mask 维度判断
2     tgt_len, bsz, embed_dim = query.size()# [tgt_len,batch_size,embed_dim]
3     src_len = key.size(0)
4     head_dim = embed_dim // num_heads  # num_heads * head_dim = embed_dim
5     scaling = float(head_dim) ** -0.5
6     q = q * scaling  # [query_len,batch_size,kdim * num_heads]
7
8     if attn_mask is not None:
9       # [tgt_len,src_len] or [num_heads*batch_size,tgt_len, src_len]
10        if attn_mask.dim() == 2:
11            attn_mask =attn_mask.unsqueeze(0)#[1,tgt_len,src_len]扩充维度
12            if list(attn_mask.size()) != [1, query.size(0), key.size(0)]:
13                raise RuntimeError('The size of the 2D attn_mask is not
                                      correct.')
14        elif attn_mask.dim() == 3:
15            if list(attn_mask.size()) != [bsz * num_heads,
                                  query.size(0),key.size(0)]:
16                raise RuntimeError('The size of the 3D attn_mask is not
                                      correct.')
17        # 现在 atten_mask 的维度就变成了 3D
```

接着，在上述代码中第 5-6 行所完成的就是图 3-14 中的缩放过程；第 8-16 行用来判断或修改 attn_mask 的维度，当然这几行代码只会在解码器中的 Masked Multi-Head Attention 中用到。

```
1     # 第三阶段： 计算得到注意力权重矩阵
2     q =q.contiguous().view(tgt_len, bsz*num_heads, head_dim).transpose(0, 1)
3     # [batch_size * num_heads,tgt_len,kdim]
4    # 因为前面是 num_heads 个头一起参与的计算，所以这里要进行一下变形，以便于 后面计算。
      且同时交换了 0，1 两个维度
5     k = k.contiguous().view(-1, bsz*num_heads, head_dim).transpose(0,1)
6     #[batch_size * num_heads,src_len,kdim]
7     v = v.contiguous().view(-1, bsz*num_heads, head_dim).transpose(0,1)
8     #[batch_size * num_heads,src_len,vdim]
9     attn_output_weights = torch.bmm(q, k.transpose(1, 2))
10    # [batch_size * num_heads,tgt_len,kdim] x [batch_size * num_heads, kdim, src_len]
11    # = [batch_size*num_heads, tgt_len, src_len] 这就 num_heads 个 QK 相乘后的注意力矩阵
```

继续，在上述代码中第 1-7 行所做的就是交换 Q、K、V 中的维度，以便于多个样本同时进行计算；第 9 行代码便是用来计算注意力权重矩阵；其中上 contiguous()方法是将变量放到一块连续的物理内存中；bmm 的作用是用来计算两个三维矩阵的乘法操作[4]。

需要提示的是，大家在看代码的时候，最好是仔细观察一下各个变量维度的变化过程，掌柜也在每次运算后进行了批注。

```
1     # 第四阶段： 进行相关掩码操作
2     if attn_mask is not None:
3         attn_output_weights += attn_mask
          #[batch_size*num_heads,tgt_len,src_len]
4     if key_padding_mask is not None:
5         attn_output_weights = attn_output_weights.view(bsz, num_heads,
                                tgt_len, src_len)
6         # 变成 [batch_size, num_heads, tgt_len, src_len]的形状
7         attn_output_weights = attn_output_weights.masked_fill(
8          key_padding_mask.unsqueeze(1).unsqueeze(2),float('-inf'))
9          # 扩展维度，从[batch_size,src_len]变成[batch_size,1,1,src_len]
10        attn_output_weights = attn_output_weights.view(bsz * num_heads,
                                tgt_len,src_len)
11        # [batch_size * num_heads, tgt_len, src_len]
```

进一步，在上述代码中第 2-3 行便是用来执行图 3-3 中的步骤；第 4-8 行便是用来执行图 3-5 中的步骤，同时还进行了维度扩充。

```
1     attn_output_weights = F.softmax(attn_output_weights, dim=-1)
2     # [batch_size * num_heads, tgt_len, src_len]
3     attn_output_weights=F.dropout(attn_output_weights,p=dropout_p,
                                    training=training)
4     attn_output = torch.bmm(attn_output_weights, v)
5     # Z=[batch_size*num_heads,tgt_len, src_len]@[batch_size * num_heads,src_len,vdim]
6     # = [batch_size * num_heads,tgt_len,vdim]
7     # 这就 num_heads 个 Attention(Q,K,V)结果
8     attn_output = attn_output.transpose(0,1).contiguous()
                                              .view(tgt_len,bsz, embed_dim)
9     # 先 transpose 成 [tgt_len, batch_size* num_heads ,kdim]
10    # 再 view 成 [tgt_len,batch_size,num_heads*kdim]
11    attn_output_weights = attn_output_weights.view(bsz, num_heads,
                            tgt_len,src_len)
12
13    Z = F.linear(attn_output, out_proj_weight, out_proj_bias)
14    # 这里就是多个 z  线性组合成 Z  [tgt_len,batch_size,embed_dim]
15    return Z, attn_output_weights.sum(dim=1) / num_heads  # 将 num_heads 个注意
      力权重矩阵按对应维度取平均
```

最后，在上述代码中第 1-4 行便是用来对权重矩阵进行归一化操作，以及计算得到多头注意力机制的输出；第 13 行代码便是用来对多个注意力的输出结果进行线性组合；第 15 行代码用来返回线性组合后的结果，以及多个注意力权重矩阵的平均值。

3.3.5 示例代码

在实现完类 MyMultiHeadAttention 的全部代码后，便可以通过类似如下的方式进行使用。

```
1 if __name__ == '__main__':
2     src_len = 5
3     batch_size = 2
4     dmodel = 32
5     num_head = 1
6     src = torch.rand((src_len, batch_size, dmodel))
        # shape: [src_len,   batch_size, embed_dim]
7     src_key_padding_mask = torch.tensor([
              [True, True, True, False, False],
8             [True, True, True, True, False]])# shape:[src_len,src_len]
9     my_mh = MyMultiheadAttention(embed_dim=dmodel, num_heads=num_head)
10    r = my_mh(src,src,src,key_padding_mask = src_key_padding_mask)
```

在上述代码中，第 6-10 行其实也就是 Encoder 中多头注意力机制的实现过程。同时，在计算过程中还可以打印出各个变量的维度变化信息：

```
1 进入多头注意力计算
2      多头 num_heads = 1, d_model=32, d_k = d_v = d_model/num_heads=32
3      query 的 shape([tgt_len, batch_size, embed_dim]):torch.Size([5, 2, 32])
4       W_q 的 shape([embed_dim,kdim * num_heads]):torch.Size([32, 32])
5       Q 的 shape([tgt_len, batch_size,kdim * num_heads]):torch.Size([5,2,32])
6     ----------------------------------------------------------------------
7       key 的 shape([src_len,batch_size, embed_dim]):torch.Size([5, 2, 32])
8       W_k 的 shape([embed_dim,kdim * num_heads]):torch.Size([32, 32])
9       K 的 shape([src_len,batch_size,kdim * num_heads]):torch.Size([5,2,32])
10    ----------------------------------------------------------------------
11     value 的 shape([src_len,batch_size, embed_dim]):torch.Size([5, 2, 32])
12      W_v 的 shape([embed_dim,vdim * num_heads]):torch.Size([32, 32])
13      V 的 shape([src_len,batch_size,vdim * num_heads]):torch.Size([5,2,32])
14    ----------------------------------------------------------------------
15     ***** 注意，这里的 W_q, W_k, W_v 是多个 head 同时进行计算的. 因此 Q,K,V 分别也是
       包含了多个 head 的 q,k,v 堆叠起来的结果 *****
16    多头注意力中，多头计算结束后的形状（堆叠）为([tgt_len,batch_size, num_heads*
      kdim]) torch.Size([5, 2, 32])
17     多头计算结束后，再进行线性变换时的权重 W_o 的形状为([num_heads*vdim,
       num_heads*   vdim ])torch.Size([32, 32])
18    多头线性变化后的形状为([tgt_len,batch_size,embed_dim])torch.Size([5,2,32])
```

第 4 节 Transformer 的实现过程

在前面几部分内容中，掌柜陆续介绍了多头注意力机制的原理、Transformer 中编码器和解码器的工作流程以及多头注意力的实现过程等。接下来，掌柜将会一步一步地来详细介绍如何通过 Pytorch 框架实现 Transformer 的整体网络结构，包括 Token Embedding、Positional Embedding、编码器和解码器等。

下面，首先要介绍的就是对于 Embedding 部分的编码实现。

4.1 Embedding 实现

4.1.1 Token Embedding

这里首先要实现的便是最基础的 Token Enbedding，也是字符转向量的一种常用做法，如下所示：

```
1 class TokenEmbedding(nn.Module):
2     def __init__(self, vocab_size: int, emb_size):
3         super(TokenEmbedding, self).__init__()
4         self.embedding = nn.Embedding(vocab_size, emb_size)
5         self.emb_size = emb_size
6     def forward(self, tokens):
7         return self.embedding(tokens.long()) * math.sqrt(self.emb_size)
```

如上代码所示便是 TokenEmbedding 的实现过程，由于这部分代码并不复杂所以就不再逐行进行介绍。第 6 行中 tokens 是原始序列输入，形状为[len, batch_size]；第 7 行代码对原始向量进行缩放是出自论文中 3.4 部分的描述。

4.1.2 Positional Embedding

在 2.1.2 节中掌柜已经对 Positional Embedding 的原理做了详细的介绍，其每个位置的变化方式如式(4.1)所示。

$$\begin{aligned} PE_{pos,2i} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{pos,2i+1} &= cos(pos/10000^{2i/d_{model}}) \end{aligned} \tag{4.1}$$

进一步，还可以对式(4.1)中括号内的参数进行化简得到如式(4.2)中的形式。

$$\begin{aligned} \frac{1}{10000^{2i/d_{model}}} &= \exp\{\log(10000)^{\frac{-2i}{d_{model}}}\} \\ &= \exp\{\frac{-2i\cdot\log(10000)}{d_{model}}\} = \exp\{2i\cdot(\frac{\text{-log}(10000)}{d_{model}})\} \end{aligned} \tag{4.2}$$

由此，根据式(4.1)(4.2)便可以实现 Positional Embedding 部分的代码，如下：

```
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)
```

如上代码所示便是整个 Positional Embedding 的实现过程，其中第 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（也就是图 4-1 中从左往右数第 2 个矩阵），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 行代码首先是在位置矩阵中取与输入序列长度相等的前 x_len 行，然后在加上 Token Embedding 的结果；第 20 行是用来返回最后得到的结果并进行 Dropout 操作。同时，这里需要注意的一点便是，在输入 x 的维度中 batch_size 并不是第 1 个维度。

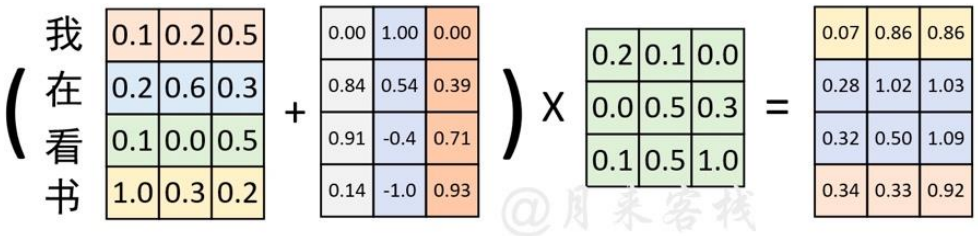

图 4-1. Positional Embedding 计算过程图

4.1.3 Embedding 代码示例

在实现完这部分代码后，便可以通过如下方式进行使用：

```
if __name__ == '__main__':
    x = torch.tensor([[1,3,5,7,9],[2,4,6,8,10]], dtype=torch.long)
    x = x.reshape(5, 2)  # [src_len, batch_size]
    token_embedding = TokenEmbedding(vocab_size=11, emb_size=512)
    x = token_embedding(tokens=x)
    pos_embedding = PositionalEncoding(d_model=512)
    x = pos_embedding(x=x)
    print(x.shape) # torch.Size([5, 2, 512])
```

4.2 Transformer 实现

在介绍完 Embedding 部分的编码工作后，下面就开始正式如何来实现 Transformer 网络结构。如图 4-2 所示，对于 Transformer 网络的实现一共会包含 4 个部分：TransformerEncoderLayer、TransformerEncoder、TransformerDecoderLayer 和 TransformerDecoder，其分别表示定义一个单独编码层、构造由多个编码层组合得到的编码器、定义一个单独的解码层以及构造由多个解码层得到的解码器。

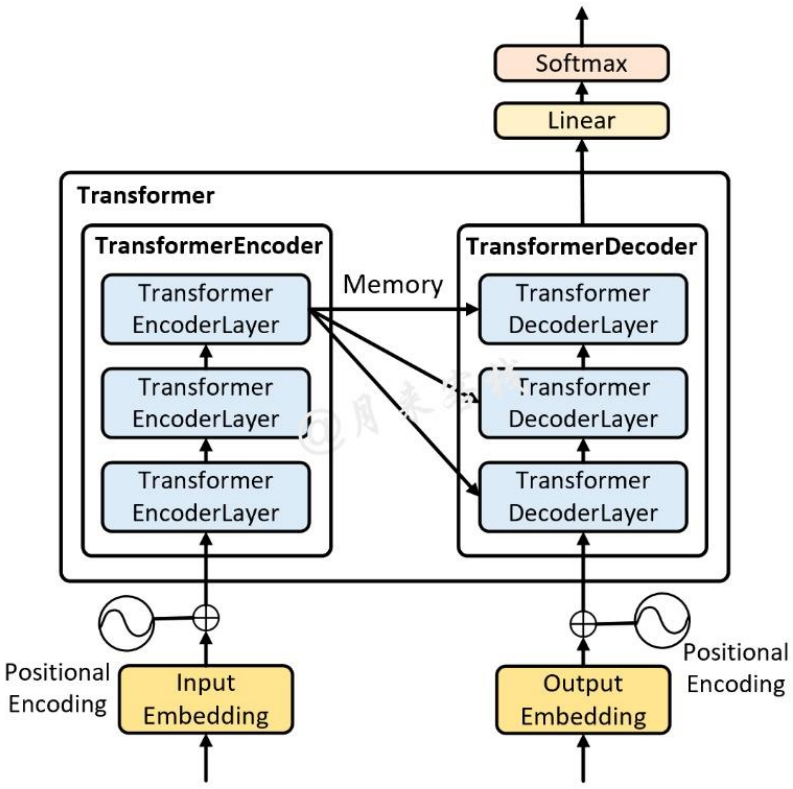

图 4-2 Transformer 实现结构图

需要注意的是，图 4-2 中的一个 EncoderLayer 指的就是图 3-2 中的一个对应的 Encoder，DecoderLayer 同理。

4.2.1 编码层的实现

首先，需要实现最基本的编码层单元，也就是图 4-2 中的 TransformerEncoderLayer，其内部结构为图 4-3 所示的前向传播过程（不包括 Embedding 部分）。

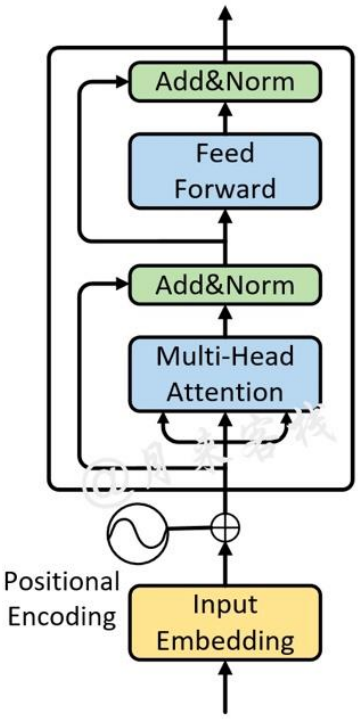

图 4-3. 编码层前向传播过程

对于这部分前向传播过程，可以通过如下代码来进行实现：

```
1  class MyTransformerEncoderLayer(nn.Module):
2      def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1):
3          super(MyTransformerEncoderLayer, self).__init__()
4          """
5          :param d_model:d_k = d_v=d_model/nhead=64,模型中向量的维度，默认值为 512
6          :param nhead:            多头注意力机制中多头的数量，论文默认为值 8
7          :param dim_feedforward: 全连接中向量的维度，论文默认值为 2048
8          :param dropout:          丢弃率，论文中的默认值为 0.1
9          """
10         self.self_attn = MyMultiheadAttention(d_model,nhead,dropout=dropout)
11         self.dropout1 = nn.Dropout(dropout)
12         self.norm1 = nn.LayerNorm(d_model)
13
14         self.linear1 = nn.Linear(d_model, dim_feedforward)
15         self.dropout = nn.Dropout(dropout)
16         self.linear2 = nn.Linear(dim_feedforward, d_model)
17         self.activation = F.relu
18         self.dropout2 = nn.Dropout(dropout)
19         self.norm2 = nn.LayerNorm(d_model)
```

在上述代码中，第 10 行用来定义一个多头注意力机制模块，并传入相应的参数；第 11-19 行代码便是用来定义其它层归一化和线性变换的模块。在完成类

MyTransformerEncoderLayer 的初始化操作后，便可以实现整个前向传播的 forward 方法：

```
1     def forward(self, src,src_mask = None,src_key_padding_mask = None):
2         """
3         :param src: 编码部分的输入，形状为 [src_len,batch_size, embed_dim]
4         :param src_mask:  编码部分输入的 padding 情况，形状为 [batch_size, src_len]
5         :return: [src_len,batch_size,num_heads*kdim]<=>[src_len,batch_size,embed_dim]
6         """
7         src2 = self.self_attn(src, src, src, attn_mask=src_mask,
8                               key_padding_mask=src_key_padding_mask, )[0]
            # 计算多头注意力
9       # src2:[src_len,batch_size,num_heads*kdim] num_heads*kdim = embed_dim
10        src = src + self.dropout1(src2)  # 残差连接
11        src = self.norm1(src)  # [src_len,batch_size,num_heads*kdim]
12
13        src2 = self.activation(self.linear1(src))
          # [src_len, batch_size,dim_feedforward]
14        src2 = self.linear2(self.dropout(src2))
          # [src_len,batch_size,num_heads*kdim]
15        src = src + self.dropout2(src2)
16        src = self.norm2(src)
17        return src # [src_len, batch_size, num_heads * kdim] <=>
                       [src_len,batch_size,embed_dim]
```

在上述代码中，第 7-8 行便是用来实现图 4-3 中 Multi-Head Attention 部分的前向传播过程；第 10-11 行用来实现多头注意力后的 Add&Norm 部分；第 13-16 行用来实现图 4-3 中最上面的 Feed Forward 部分和 Add&Norm 部分。

这里再次提醒大家，在阅读代码的时候最好是将对应的维度信息代入以便于理解。

4.2.2 编码器实现

在实现完一个标准的编码层之后，便可以基于此来实现堆叠多个编码层，从而得到 Transformer 中的编码器。对于这部分内容，可以通过如下代码来实现：

```
1 def _get_clones(module, N):
2     return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
3
4 class MyTransformerEncoder(nn.Module):
5     def __init__(self, encoder_layer, num_layers, norm = None):
6         super(MyTransformerEncoder, self).__init__()
7         """
8         encoder_layer: 就是包含有多头注意力机制的一个编码层
9         num_layers: 克隆得到多个 encoder layers 论文中默认为 6
10        norm: 归一化层
```

```
11          """
12          self.layers = _get_clones(encoder_layer, num_layers)
13          # 克隆得到多个 encoder layers 论文中默认为 6
14          self.num_layers = num_layers
15          self.norm = norm
```

在上述代码中，第 1-2 行是用来定义一个克隆多个编码层或解码层功能函数；第 12 行中的 encoder_layer 便是一个实例化的编码层，self.layers 中保存的便是一个包含有多个编码层的 ModuleList。在完成类 MyTransformerEncoder 的初始化后，便可以实现整个前向传播的 forward 方法：

```
1     def forward(self, src, mask=None, src_key_padding_mask=None):
2         """
3         :param src: 编码部分的输入，形状为 [src_len,batch_size, embed_dim]
4         :param mask:  编码部分输入的 padding 情况，形状为 [batch_size, src_len]
5         :return:# [src_len, batch_size, num_heads * kdim] <==>
                  [src_len,batch_size,embed_dim]
6         """
7         output = src
8         for mod in self.layers:
9             output = mod(output, src_mask = mask,
10                         src_key_padding_mask = src_key_padding_mask)
11            # 多个 encoder layers 层堆叠后的前向传播过程
12        if self.norm is not None:
13            output = self.norm(output)
14        return output  # [src_len, batch_size, num_heads * kdim] <==>
                         [src_len,batch_size,embed_dim]
```

在上述代码中，第 8-10 行便是用来实现多个编码层堆叠起来的效果，并完成整个前向传播过程；第 11-13 行用来对多个编码层的输出结果进行层归一化并返回最终的结果。

4.2.3 编码器使用示例

在完成 Transformer 中编码器的实现过程后，便可以将其用于对输入序列进行编码。例如可以仅仅通过一个编码器对输入序列进行编码，然后将最后的输出喂入到分类器当中进行分类处理，这部分内容在后续也会进行介绍。下面先看一个使用示例。

```
1 if __name__ == '__main__':
2     src_len = 5
3     batch_size = 2
4     dmodel = 32
5     num_head = 4
6     num_layers = 2
```

```
7     src=torch.rand((src_len, batch_size,
              dmodel))#[src_len,batch_size,embed_dim]
8     src_key_padding_mask = torch.tensor([[True, True, True, False, False],
9         [True, True, True, True, False]])  # shape: [batch_size, src_len]
10    my_transformer_encoder_layer = MyTransformerEncoderLayer(d_model=dmodel,
                                       nhead=num_head)
11    my_transformer_encoder = MyTransformerEncoder (encoder_layer
                           =my_transformer_encoder_layer,
                           num_layers=num_layers,
12                         norm=nn.LayerNorm(dmodel))
13    memory = my_transformer_encoder(src=src, mask=None,
14              src_key_padding_mask=src_key_padding_mask)
15    print(memory.shape)  # torch.Size([5, 2, 32])
```

在上述代码中，第 2-6 行定义了编码器中各个部分的参数值；第 10-11 行则是首先定义一个编码层，然后再定义由多个编码层组成的编码器；第 13-14 行便是用来得到整个编码器的前向传播输出结果，并且需要注意的是在编码器中不需要掩盖当前时刻之后的位置信息，所以 mask=None。

4.2.4 解码层实现

在介绍完编码器的实现后，下面就开始介绍如何实现 Transformer 中的解码器部分。同编码器的实现流程一样，首先需要实现的依旧是一个标准的解码层，也就是图 4-4 所示的前向传播过程（不包括 Embedding 部分）。

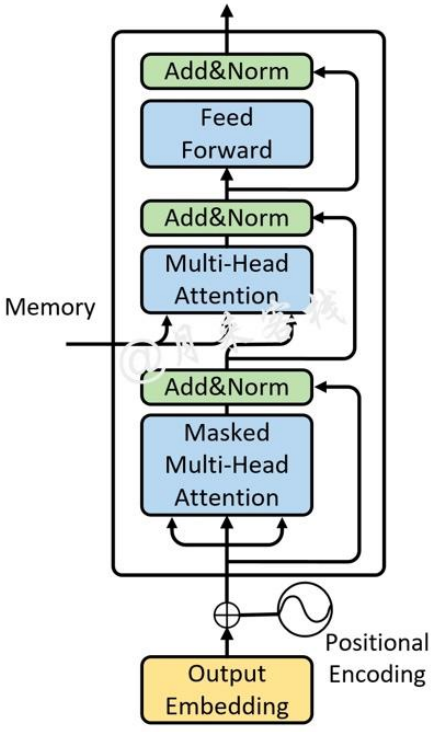

图 4-4. 解码层前向传播过程

对于这部分前向传播过程，可以通过如下代码来进行实现：

```
1  class MyTransformerDecoderLayer(nn.Module):
2      def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1):
3          super(MyTransformerDecoderLayer, self).__init__()
4          """
5          :param d_model:d_k =d_v=d_model/nhead= 64,模型中向量的维度，默认值为 512
6          :param nhead: 多头注意力机制中多头的数量，论文默认为值 8
7          :param dim_feedforward: 全连接中向量的维度，论文默认值为 2048
8          :param dropout:  丢弃率，论文中的默认值为 0.1
9          """
10         self.self_attn = MyMultiheadAttention(embed_dim=d_model,
                                num_heads=nhead, dropout=dropout)
11     # 解码部分输入序列之间的多头注意力（即论文结构图中的 Masked Multi-head attention)
12         self.multihead_attn = MyMultiheadAttention(embed_dim=d_model,
                                    num_heads=nhead, dropout=dropout)
13         # 编码部分输出（memory）和解码部分之间的多头注意力机制。
14         self.linear1 = nn.Linear(d_model, dim_feedforward)
15         self.dropout = nn.Dropout(dropout)
16         self.linear2 = nn.Linear(dim_feedforward, d_model)
17
18         self.norm1 = nn.LayerNorm(d_model)
19         self.norm2 = nn.LayerNorm(d_model)
20         self.norm3 = nn.LayerNorm(d_model)
21         self.dropout1 = nn.Dropout(dropout)
22         self.dropout2 = nn.Dropout(dropout)
23         self.dropout3 = nn.Dropout(dropout)
24         self.activation = F.relu
```

在上述代码中，第 10 行代码用来定义图 4-4 中 Masked Multi-head Attention 部分的前向传播过程；第 12 行则是用来定义图 4-4 中编码器与解码器交互的多头注意力机制模块；第 14-24 行是用来定义剩余的全连接层以及层归一化相关操作。在完成类 MyTransformerDecoderLayer 的初始化后，便可以实现整个前向传播的 forward 方法：

```
1      def forward(self,tgt,memory,tgt_mask=None,memory_mask=None,
                   tgt_key_padding_mask = None,
2                  memory_key_padding_mask=None):
3          tgt2 = self.self_attn(tgt, tgt, tgt, #[tgt_len,batch_size,embed_dim]
4                                 attn_mask=tgt_mask,
5                                 key_padding_mask=tgt_key_padding_mask)[0]
6      # 解码部分输入序列之间的多头注意力（论文结构图中的 Masked Multi-head attention)
7          tgt = tgt + self.dropout1(tgt2)  # 接着是残差连接
8          tgt = self.norm1(tgt)  # [tgt_len,batch_size, embed_dim]
9          tgt2 = self.multihead_attn(tgt,memory,memory, attn_mask=memory_mask,
```

```
10                 key_padding_mask=memory_key_padding_mask)[0]
11                # [tgt_len,batch_size, embed_dim]
12     # 解码部分的输入经多头注意力后同编码部分的输出（memory）通过多头注意力机制进行交互
13         tgt = tgt + self.dropout2(tgt2)  # 残差连接
14         tgt = self.norm2(tgt)  # [tgt_len, batch_size, embed_dim]
15
16         tgt2 = self.activation(self.linear1(tgt))
             # [tgt_len, batch_size, dim_feedforward]
17         tgt2 = self.linear2(self.dropout(tgt2))
           # [tgt_len, batch_size, embed_dim]
18         # 最后的两层全连接
19         tgt = tgt + self.dropout3(tgt2)
10         tgt = self.norm3(tgt)
21         return tgt
```

在上述代码中，第 1-2 行里 tgt 是解码器部分的输入，形状为 [tgt_len,batch_size, embed_dim]；memory 是编码部分的输出结果，形状为[src_len, batch_size, embed_dim]；tgt_mask 用于解码器中掩盖当前 position 之后的信息，形状[tgt_len, tgt_len]；memory_mask 是编码器-解码器交互时的注意力掩码，一般为 None；tgt_key_padding_mask 是解码部分输入序列的 padding 情况，形状为[batch_size, tgt_len]；memory_key_padding_mask 是编码器输入部分的 padding 向量，形状为[batch_size, src_len]。

第 3-5 行用来完成图 4-4 中 Masked Multi-head Attention 部分的前向传播过程，其中 tgt_mask 就是在训练时用来掩盖当前时刻之后位置的注意力掩码；第 7-8 行用来完成图 4-4 中 Masked Multi-head Attention 之后 Add&Norm 部分的前向传播过程；第 9-11 行用来实现解码器与编码器之间的交互过程，其中 memory_mask 为 None，memory_key_padding_mask 为 src_key_padding_mask 用来对编码器的输出进行（序列）填充部分的掩盖，这一点同编码器中的 key_padding_mask 原理一样；第 13-21 行便是用来实现余下的其它过程，其中第 21 行的返回结果形状为[tgt_len, batch_size, num_heads * kdim]，即[tgt_len, batch_size, embed_dim]。

4.2.5 解码器实现

在实现完一个标准的解码层之后，便可以基于此来实现堆叠多个解码层，从而得到 Transformer 中的解码器。对于这部分内容，可以通过如下代码来实现：

```
1 class MyTransformerDecoder(nn.Module):
2     def __init__(self, decoder_layer, num_layers, norm=None):
3         super(MyTransformerDecoder, self).__init__()
4         self.layers = _get_clones(decoder_layer, num_layers)
5         self.num_layers = num_layers
6         self.norm = norm
```

```
7
8     def forward(self, tgt, memory, tgt_mask=None, memory_mask=None,
                  tgt_key_padding_mask=None,
9                 memory_key_padding_mask=None):
19        output = tgt  # [tgt_len,batch_size, embed_dim]
20        for mod in self.layers:  # 这里的 layers 就是 N 层解码层堆叠起来的
21            output = mod(output, memory,
22                         tgt_mask=tgt_mask,
23                         memory_mask=memory_mask,
24                         tgt_key_padding_mask=tgt_key_padding_mask,
25                         memory_key_padding_mask=memory_key_padding_mask)
26        if self.norm is not None:
27            output = self.norm(output)
28        return output
```

在上述代码中，第 4 行用来克隆得到多个解码层；第 8-9 行中，tgt 是解码部分的输入，形状为 [tgt_len,batch_size, embed_dim]；tgt_mask 是解码器中注意力 Mask 输入，掩盖当前 position 之后的信息，形状为[tgt_len, tgt_len]；memory 是编码部分最后一层的输出，形状为[src_len,batch_size, embed_dim]；memory_mask 是编码器-解码器交互时的注意力掩码，一般情况为 None；memory_key_padding_mask: 编码部分输入的 padding 情况，[batch_size, src_len]；tgt_key_padding_mask 是解码部分输入序列的 padding 情况，形状为[batch_size, tgt_len]；第 20-25 行用来实现多层解码层的前向传播过程；第 28 行便是用来返回最后的结果，形状为[tgt_len, batch_size, num_heads * kdim]，也即[tgt_len, batch_size, embed_dim]。

4.2.6 Transformer 网络实现

在实现完 Transformer 中各个基础模块的话，下面就可以来搭建最后的 Transformer 模型了。总体来说这部分的代码也相对简单，只需要将上述编码器解码器组合到一起即可，具体代码如下所示：

```
1 class MyTransformer(nn.Module):
2     def __init__(self, d_model=512, nhead=8, num_encoder_layers=6,
3                  num_decoder_layers=6, dim_feedforward=2048, dropout=0.1):
4         super(MyTransformer, self).__init__()
5
6         """
7         :param d_model:d_k=d_v=d_model/nhead=64, 模型中向量的维度，默认值为 512
8         :param nhead: 多头注意力机制中多头的数量，论文默认为值 8
9         :param num_encoder_layers: encoder 堆叠的数量，论文中的 N，论文默认值为 6
10        :param num_decoder_layers: decoder 堆叠的数量，论文中的 N，论文默认值为 6
11        :param dim_feedforward:    全连接中向量的维度，论文默认值为 2048
12        :param dropout:            丢弃率，论文中的默认值为 0.1
```

```
13          """
14          #  =============== 编码部分 ====================
15          encoder_layer = MyTransformerEncoderLayer(d_model, nhead,
                                dim_feedforward, dropout)
16          encoder_norm = nn.LayerNorm(d_model)
17          self.encoder = MyTransformerEncoder(encoder_layer,
                               num_encoder_layers, encoder_norm)
18          # =============== 解码部分 ====================
19          decoder_layer = MyTransformerDecoderLayer(d_model, nhead,
                                dim_feedforward, dropout)
20          decoder_norm = nn.LayerNorm(d_model)
21          self.decoder = MyTransformerDecoder(decoder_layer,
                               num_decoder_layers, decoder_norm)
22          self._reset_parameters() # 初始化模型参数
23          self.d_model = d_model
24          self.nhead = nhead
```

在上述代码中，第 15-17 行是用来定义编码器部分；第 19-21 行是用来定义解码器部分；第 22 行用来以某种方式初始化 Transformer 中的权重参数，具体实现在稍后的内容中。在定义完类 MyTransformer 的初始化函数后，便可以来实现 Transformer 的整个前向传播过程，代码如下：

```
1     def forward(self, src, tgt, src_mask=None, tgt_mask=None,
2                 memory_mask=None, src_key_padding_mask=None,
3                 tgt_key_padding_mask=None, memory_key_padding_mask=None):
4         """
5         :param src:  [src_len,batch_size,embed_dim]
6         :param tgt:  [tgt_len, batch_size, embed_dim]
7         :param src_mask:  None
8         :param tgt_mask:  [tgt_len, tgt_len]
9         :param memory_mask: None
10        :param src_key_padding_mask: [batch_size, src_len]
11        :param tgt_key_padding_mask: [batch_size, tgt_len]
12        :param memory_key_padding_mask:  [batch_size, src_len]
13        :return:[tgt_len,batch_size,num_heads*kdim]
14        """
15        memory = self.encoder(src, mask=src_mask,
                    src_key_padding_mask=src_key_padding_mask)
16        # [src_len, batch_size, num_heads * kdim]
17        output = self.decoder(tgt=tgt, memory=memory, tgt_mask=tgt_mask,
                    memory_mask=memory_mask,
18                  tgt_key_padding_mask=tgt_key_padding_mask,
19                  memory_key_padding_mask=memory_key_padding_mask)
20        return output
            #[tgt_len, batch_size, num_heads * kdim]
```

在上述代码中，src 表示编码器的输入；tgt 表示解码器的输入；src_mask 为空，因为编码时不需要对当前时刻之后的位置信息进行掩盖；tgt_mask 用于掩盖解码输入中当前时刻以后的所有位置信息；memory_mask 为空；src_key_padding_mask 表示对编码输入序列填充部分的 Token 进行 mask；tgt_key_padding_mask 表示对解码输入序列填充部分的 Token 进行掩盖；memory_key_padding_mask 表示对编码器的输出部分进行掩盖，掩盖原因等同于编码输入时的 mask 操作。

到此，对于整个 Transformer 的网络结构就算是搭建完毕了，不过这还没有实现论文中基于 Transformer 结构的翻译模型，而这部分内容掌柜也将会在下一节中进行详细的介绍。当然，出了上述模块之外，Transformer 中还有两个部分需要实现的就是参数初始化方法和注意力掩码矩阵生成方法，具体代码如下：

```
    def _reset_parameters(self):
        for p in self.parameters():
            if p.dim() > 1:
                xavier_uniform_(p)

    def generate_square_subsequent_mask(self, sz):
        mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-
               inf')).masked_fill(mask == 1, float(0.0))
        return mask # [sz,sz]
```

4.2.7 Transfromer 使用示例

在实现完 Transformer 的整个完了结构后，便可以通过如下步骤进行使用：

```
if __name__ == '__main__':
    src_len = 5
    batch_size = 2
    dmodel = 32
    tgt_len = 6
    num_head = 8
    src = torch.rand((src_len, batch_size, dmodel))
     # shape: [src_len, batch_size, embed_dim]
    src_key_padding_mask = torch.tensor([[True, True, True, False, False],
                                         [True, True, True, True, False]])
     # shape: [batch_size, src_len]

    tgt = torch.rand((tgt_len, batch_size, dmodel))
     # shape: [tgt_len, batch_size, embed_dim]
    tgt_key_padding_mask = torch.tensor([[True,True,True,False,False,False],
                                         [True,True,True,True,False,False]])
    # shape: [batch_size, tgt_len]
```

```
14
15      my_transformer = MyTransformer(d_model=dmodel, nhead=num_head,
                        num_encoder_layers=6,
16                      num_decoder_layers=6, dim_feedforward=500)
17      tgt_mask = my_transformer.generate_square_subsequent_mask(tgt_len)
18      out = my_transformer(src=src, tgt=tgt, tgt_mask=tgt_mask,
19                           src_key_padding_mask=src_key_padding_mask,
20                           tgt_key_padding_mask=tgt_key_padding_mask,
21                           memory_key_padding_mask=src_key_padding_mask)
22      print(out.shape) #torch.Size([6, 2, 32])
```

在上述代码中，第 7-13 行用来生成模拟的输入数据；第 15-16 行用来实例化类 MyTransformer；第 17 行用来生成解码输入时的注意力掩码矩阵；第 18-21 行用来执行 Transformer 网络结构的前向传播过程。

第 5 节 基于 Transformer 的翻译模型

经过前面几节内容的介绍，相信各位读者对于 Transformer 的基本原理以及实现过程已经有了一个较为清晰的认识。不过想要对一个网络模型有更加深刻的认识，那么最好的办法便是从数据预处理到模型训练，自己完完全全的经历一遍。因此，为了使得大家能够更加透彻的理解 Transformer 的整个工作流程，在本节中掌柜将继续带着大家一起来还原论文中的文本翻译模型。

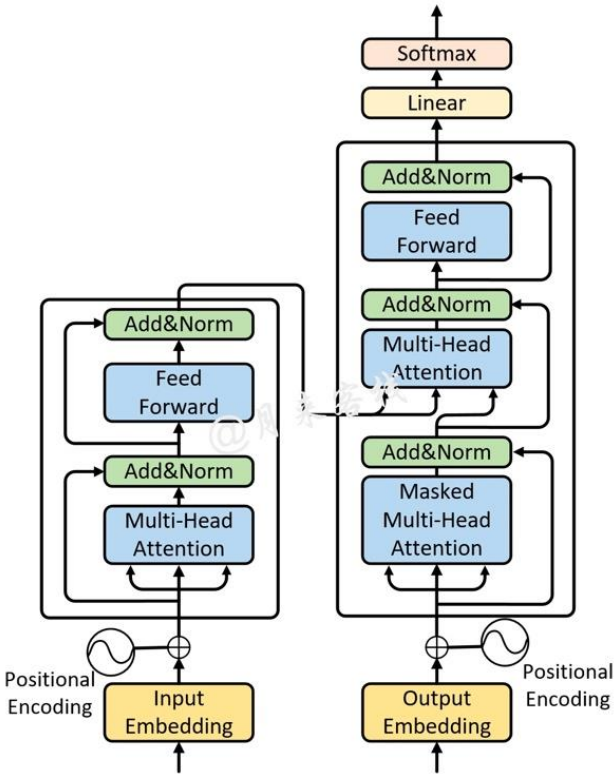

图 5-1. Transformer 网络结构图

如图 5-1 所示便是 Transformer 网络的整体结构图，对于这部分内容其实在上一节内容中总体上算已经算是介绍完了，只是在数据预处理方面还未涉及。下面，掌柜就以 Multi30K[9]中的 English-German 平行语料为例进行介绍（注意这并不是论文中所用到数据集）。

本部分完整代码可参见[11]。

5.1 数据预处理

5.1.1 语料介绍

在这里，我们使用到的平行语料一共包含有 6 个文件 train.de、train.en、val.de、val.en、test_2016_flickr.de 和 test_2016_flickr.en，其分别为德语训练语料、英语

训练语料、德语验证语料、英语验证语料、德语测试语料和英语测试语料。同时，这三部分的样本量分别为 29000、1014 和 1000 条。

如下所示便是一条平行预料数据，其中第 1 行为德语，第 2 行为英语，后续我们需要完成的就是搭建一个翻译模型将德语翻译为英语。

Zwei junge weiße Männer sind im, Freien in der Nähe vieler Büsche.
Two young, White males are outside near many bushes.

5.1.2 数据集预览

在正式介绍如何构建数据集之前，我们先通过几张图来了解一下整个构建的流程，以便做到心中有数，不会迷路。

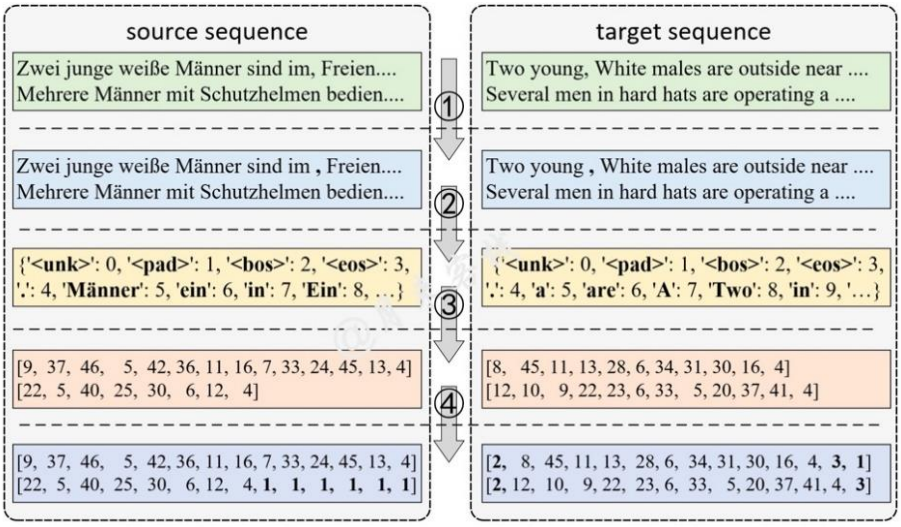

图 5-2. 翻译模型数据集处理过程图（一）

如图 5-2 所示，左边部分为原始输入，右边部分为目标输入。从图 5-2 可以看出，第 1 步需要完成的就是对原始语料进行 tokenize 操作。如果是对类似英文这样的语料进行处理，那就是直接按空格切分即可。但是需要注意的是要把其中的逗号和句号也给分割出来。第 2 步需要做的就是根据 tokenize 后的结果对原始输入和目标输入分别建立一个字典。第 3 步需要做的则是将 tokenize 后结果根据字典中的索引将其转换成 token 序列。第 4 步则是对同一个 batch 中的序列以最长的为标准其它样本进行 padding，并且同时需要在目标输入序列的前后加上起止符（即'<bos>'和'<eos>'） 。

如图 5-3 所示，在完成前面 4 个步骤后，对于目标序列来说第 5 步需要做的就是将第 4 步处理后的结果划分成 tgt_input 和 tgt_output。从图 5-3 右侧可以看出，tgt_input 和 tgt_output 是相互对应起来的。例如对于第 1 个样本来说，解码第 1 个时刻的输入应该是"2"，而此时刻对应的正确标签就应该是 tgt_output 中的'8'；解码第 2 个时刻的输入应该是 tgt_input 中的'2'和'8'，而此时刻对应的正确标签就应该是 tgt_output 中的'8'和'45'，以此类推下去。最后，第 6 步则是根据 src_input 和 tgt_input 各自的 padding 情况，得到一个 padding mask 向量（注意由

于这里 tgt_input 中的两个样本长度一样，所以并不需要 padding），其中'**T**'表示 padding 的位置。当然，这里的 tgt_mask 并没有画出。

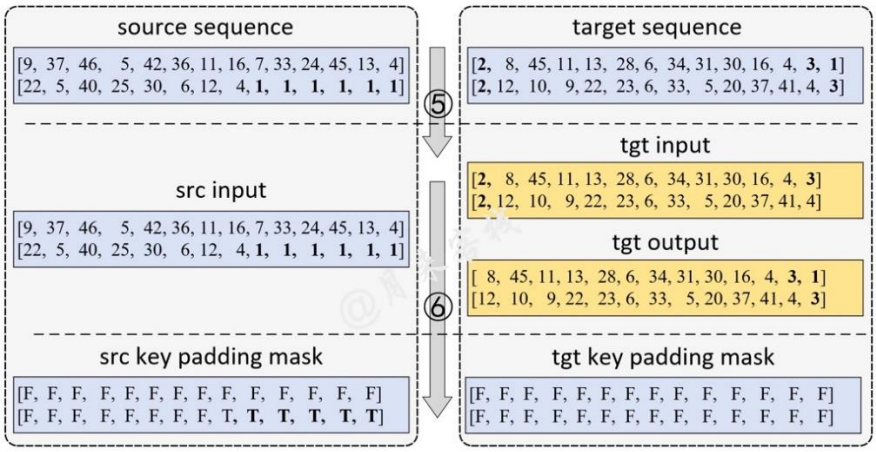

图 5-3. 翻译模型数据集处理过程图（二）

同时，图 5-3 中各部分结果体现在 Transformer 网络中的情况如图 5-4 所示。

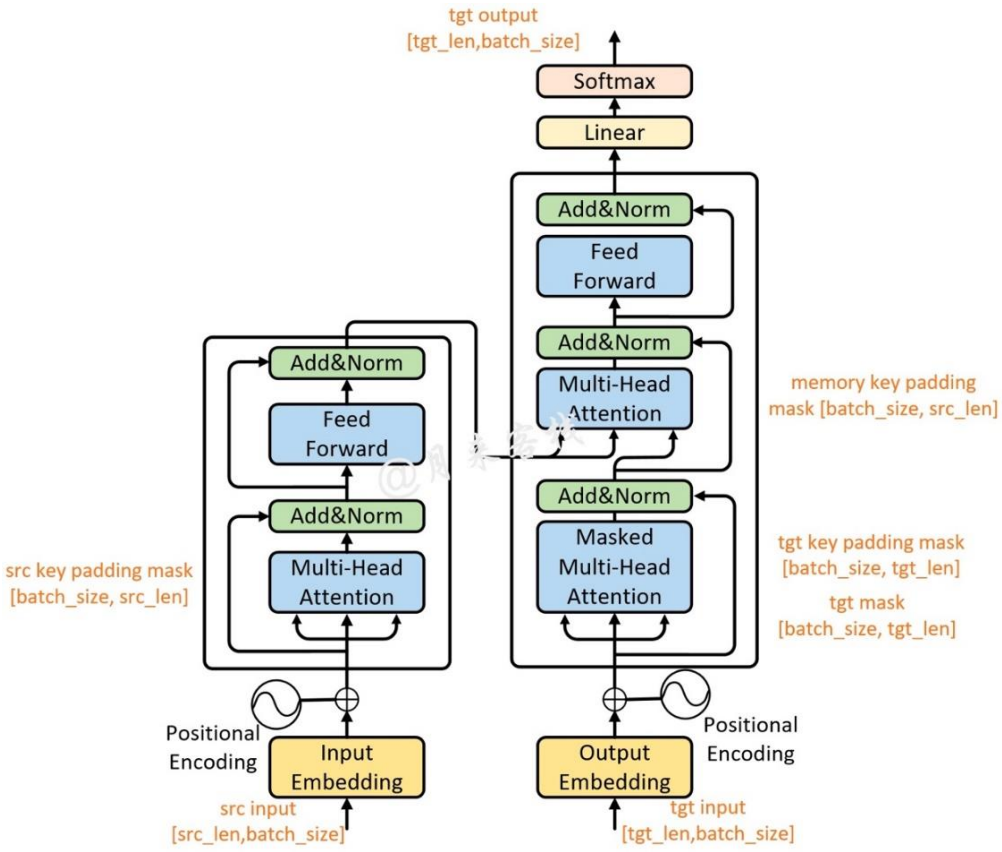

图 5-4. 基于 Transformer 翻译模型的模型输入情况

以上就是基于 Transformer 架构的翻译模型数据预处理的整个大致流程，下面我们开始正式来通过编码实现这一过程。

5.1.3 数据集构建

第 1 步：定义 tokenize

如果是对类似英文这样的语料进行处理，大部分就是直接按空格切分即可。但是需要注意的是单词中的某些缩写也需要给分割出来，例如"you're"需要分割成“you” 和"re'"。因此，这部分代码可以借助 torchtext 中的 get_tokenizer 方法来实现，具体代码如下：

```
from torchtext.data.utils import get_tokenizer
def my_tokenizer():
    tokenizer = {}
    tokenizer['de'] = get_tokenizer('spacy', language='de_core_news_sm')#德语
    tokenizer['en'] = get_tokenizer('spacy', language='en_core_web_sm') #英语
    return tokenizer
```

这里返回的是两个 tokenizer，分别用于对德语和英语进行序列化。例如对于如下文本来说

```
s = "Moon Hotel, it's very interesting."
```

其 tokenize 后的结果为：

```
tokenizer = my_tokenizer()
print(tokenizer['en'](s))
['Moon', 'Hotel', ',', 'it', "'s", 'very', 'interesting', '.']
```

第 2 步：建立词表

在介绍完 tokenize 的实现方法后，我们就可以正式通过 torchtext.vocab 中的 Vocab 方法来构建词典了，代码如下：

```
def build_vocab(tokenizer, filepath, min_freq, specials=None):
    if specials is None:
        specials = ['<unk>', '<pad>', '<bos>', '<eos>']
    counter = Counter()
    with open(filepath, encoding='utf8') as f:
        for string_ in f:
            counter.update(tokenizer(string_))
    return Vocab(counter, specials=specials, min_freq=min_freq)
```

在上述代码中，第 3 行代码用来指定特殊的字符；第 5-7 行代码用来遍历文件中的每一个样本（每行一个）并进行 tokenize 和计数，其中对于 counter.update 进行介绍可以参考[10]；第 8 行则是返回最后得到词典。

在完成上述过程后，我们将得到两个 Vocab 类的实例化对象。

一个为原始序列的字典：

```
{'<unk>': 0, '<pad>': 1, '<bos>': 2, '<eos>': 3, '.': 4, 'Männer': 5, 'ein': 6, 'in': 7, 'Ein': 8, 'Zwei': 9, 'und': 10, ',': 11, ......}
```

一个为目标序列的字典：

```
{'<unk>': 0, '<pad>': 1, '<bos>': 2, '<eos>': 3, '.': 4, 'a': 5, 'are': 6, 'A': 7, 'Two': 8, 'in': 9, 'men': 10, ',': 11, 'Several': 12,......}
```

此时，我们就需要定义一个类，并在类的初始化过程中根据训练语料完成字典的构建，代码如下：

```
class LoadEnglishGermanDataset():
    def __init__(self, train_file_paths=None, tokenizer=None,
                 batch_size=2, min_freq=1):
        # 根据训练预料建立英语和德语各自的字典
        self.tokenizer = tokenizer()
        self.de_vocab = build_vocab(self.tokenizer['de'],
                                    filepath=train_file_paths[0],
                                    min_freq=min_freq)
        self.en_vocab = build_vocab(self.tokenizer['en'],
                                    filepath=train_file_paths[1],
                                    min_freq=min_freq)
        self.specials = ['<unk>', '<pad>', '<bos>', '<eos>']
        self.PAD_IDX = self.de_vocab['<pad>']
        self.BOS_IDX = self.de_vocab['<bos>']
        self.EOS_IDX = self.de_vocab['<eos>']
        self.batch_size = batch_size
```

其中 min_freq 表示在构建词表时忽略掉出现次数小于该值的字。

第 3 步：转换为 Token 序列

在得到构建的字典后，便可以通过如下函数来将训练集、验证集和测试集转换成 Token 序列：

```
    def data_process(self, filepaths):
        """
        将每一句话中的每一个词根据字典转换成索引的形式
        :param filepaths:
        :return:
        """
        raw_de_iter = iter(open(filepaths[0], encoding="utf8"))
        raw_en_iter = iter(open(filepaths[1], encoding="utf8"))
        data = []
        for (raw_de, raw_en) in zip(raw_de_iter, raw_en_iter):
            de_tensor_ = torch.tensor([self.de_vocab[token] for token in
            self.tokenizer['de'](raw_de.rstrip("\n"))], dtype=torch.long)
            en_tensor_ = torch.tensor([self.en_vocab[token] for token in
            self.tokenizer['en'](raw_en.rstrip("\n"))], dtype=torch.long)
            data.append((de_tensor_, en_tensor_))
        return data
```

在上述代码中，第 11-4 行分别用来将原始序列和目标序列转换为对应词表中的 Token 形式。在处理完成后，就会得到类似如下的结果：

```
[(tensor([9, 37, 46, 5, 42, 36, 11, 16, 7, 33, 24, 45, 13, 4]), tensor([8, 45, 11, 13, 28, 6, 34, 31, 30, 16, 4])),
(tensor([22, 5, 40, 25, 30, 6, 12, 4]), tensor([12, 10, 9, 22, 23, 6, 33, 5, 20, 37, 41, 4])),
(tensor([8, 38, 23, 39, 7, 6, 26, 29, 19, 4]), tensor([ 7, 27, 21, 18, 24, 5, 44, 35, 4])),
(tensor([9, 5, 43, 27, 18, 10, 31, 14, 47, 4]), tensor([ 8, 10, 6, 14, 42, 40, 36, 19, 4]))]
```

其中左边的一列就是原始序列的 Token 形式，右边一列就是目标序列的 Token 形式，每一行构成一个样本。

第 4 步：padding 处理

从上面的输出结果（以及图 5-2 中第③步后的结果）可以看到，无论是对于原始序列来说还是目标序列来说，在不同的样本中其对应长度都不尽相同。但是在将数据输入到相应模型时却需要保持同样的长度，因此在这里我们就需要对 Token 序列化后的样本进行 padding 处理。同时需要注意的是，一般在这种生成模型中，**模型在训练过程中只需要保证同一个 batch 中所有的原始序列等长，所有的目标序列等长即可**，也就是说不需要在整个数据集中所有样本都保证等长。

因此，在实际处理过程中无论是原始序列还是目标序列都会以每个 batch 中最长的样本为标准对其它样本进行 padding，具体代码如下：

```
1     def generate_batch(self, data_batch):
2         de_batch, en_batch = [], []
3         for (de_item, en_item) in data_batch:
              # 开始对一个 batch 中的每一个样本进行处理。
4             de_batch.append(de_item)  # 编码器输入序列不需要加起止符
5             # 在每个 idx 序列的首位加上 起始 token 和 结束 token
6             en = torch.cat([torch.tensor([self.BOS_IDX]), en_item,
                              torch.tensor([self.EOS_IDX])], dim=0)
7             en_batch.append(en)
8         # 以最长的序列为标准进行填充
9         de_batch = pad_sequence(de_batch, padding_value=self.PAD_IDX)
          # [de_len,batch_size]
10        en_batch = pad_sequence(en_batch, padding_value=self.PAD_IDX)
          # [en_len,batch_size]
11        return de_batch, en_batch
```

在上述代码中，第 6-7 行用来在目标序列的首尾加上特定的起止符；第 9-10 行则是分别对一个 batch 中的原始序列和目标序列以各自当中最长的样本为标准进行 padding（这里的 pad_sequence 导入自 torch.nn.utils.rnn）。

第 5 步：构造 mask 向量

在处理完成图 5-2 中的第④步后，对于图 5-3 中的第⑤步来说就是简单的切片操作，因此就不作介绍。进一步需要根据 src_input 和 tgt_input 来构造相关的 mask 向量，具体代码如下：

```
1     def generate_square_subsequent_mask(self, sz, device):
2         mask = (torch.triu(torch.ones((sz,sz),
                  device = device)) == 1).transpose(0,1)
3         mask = mask.float().masked_fill(mask == 0, float('-inf')).
                  masked_fill(mask == 1, float(0.0))
4         return mask
5
6     def create_mask(self, src, tgt, device='cpu'):
7         src_seq_len = src.shape[0]
8         tgt_seq_len = tgt.shape[0]
9         tgt_mask =self.generate_square_subsequent_mask(tgt_seq_len,device)
          # [tgt_len,tgt_len]
10        # Decoder 的注意力 Mask 输入，用于掩盖当前 position 之后的 position，所以这里是一
           个对称矩阵
11        src_mask = torch.zeros((src_seq_len, src_seq_len),
                    device=device).type(torch.bool)
12        #Encoder 的注意力 Mask 输入，这部分其实对于 Encoder 来说是没有用的，所以这里全是 0
13        src_padding_mask = (src == self.PAD_IDX).transpose(0, 1)
14        # 用于 mask 掉 Encoder 的 Token 序列中的 padding 部分,[batch_size, src_len]
15        tgt_padding_mask = (tgt == self.PAD_IDX).transpose(0, 1)
16        # 用于 mask 掉 Decoder 的 Token 序列中的 padding 部分,batch_size, tgt_len
17        return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

在上述代码中，第 1-4 行是用来生成一个形状为[sz,sz]的注意力掩码矩阵，用于在解码过程中掩盖当前 position 之后的 position；第 6-17 行用来返回 Transformer 中各种情况下的 mask 矩阵，其中 src_mask 在这里并没有作用。

第 6 步：构造 DataLoade 与使用示

经过前面 5 步的操作，整个数据集的构建就算是已经基本完成了，只需要再构造一个 DataLoader 迭代器即可，代码如下：

```
1     def load_train_val_test_data(self,train_file_paths,
                                   val_file_paths, test_file_paths):
2         train_data = self.data_process(train_file_paths)
3         val_data = self.data_process(val_file_paths)
4         test_data = self.data_process(test_file_paths)
5         train_iter = DataLoader(train_data,batch_size=self.batch_size,
6                       shuffle=True,collate_fn=self.generate_batch)
7         valid_iter = DataLoader(val_data, batch_size=self.batch_size,
8                       shuffle=True, collate_fn=self.generate_batch)
9         test_iter = DataLoader(test_data, batch_size=self.batch_size,
10                     shuffle=True, collate_fn=self.generate_batch)
11        return train_iter, valid_iter, test_iter
```

在上述代码中，第 2-4 行便是分别用来将训练集、验证集和测试集转换为 Token 序列；第 5-10 行则是分别构造 3 个 DataLoader，其中 generate_batch 将作为

一个参数传入来对每个 batch 的样本进行处理。在完成类 LoadEnglishGermanDataset 所有的编码过程后，便可以通过如下形式进行使用：

```
if __name__ == '__main__':
    train_filepath = ['data/train_.de',
                      'data/train_.en']

    data_loader = LoadEnglishGermanDataset(train_filepath,
                  tokenizer=my_tokenizer, batch_size=2)
    train_iter, valid_iter, test_iter =
                  data_loader.load_train_val_test_data(train_filepath,
                                                       train_filepath,
                                                       train_filepath)
    print(data_loader.PAD_IDX)
    for src, tgt in train_iter:
        tgt_input = tgt[:-1, :]
        tgt_out = tgt[1:, :]
        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask =
                                    data_loader.create_mask(src, tgt_input)
        print("src shape: ", src.shape)  # [de_tensor_len,batch_size]
        print("src_padding_mask shape (batch_size, src_len): ",
                src_padding_mask.shape)
        print("tgt input shape:", tgt_input.shape)
        print("tgt_padding_mask shape: (batch_size,tgt_len) ",
                                            tgt_padding_mask.shape)
        print("tgt output shape:", tgt_out.shape)
        print("tgt_mask shape (tgt_len,tgt_len): ", tgt_mask.shape)
        break
```

各位读者在阅读这部分代码时最好是能够结合图 5-2 到 5-4 进行理解，这样效果可能会更好。在介绍完数据集构建的整个过程后，下面就开始正式进入到翻译模型的构建中。

5.2 翻译模型

5.2.1 网络结构

总体来说，基于 Transformer 的翻译模型的网络结构其实就是图 5-4 所展示的所有部分，只是在前面介绍 Transformer 网络结构时掌柜并没有把 Embedding 部分的实现给加进去。这是因为对于不同的文本生成模型，其 Embedding 部分会不一样（例如在诗歌生成这一情景中编码器和解码器共用一个 TokenEmbedding 即可，而在翻译模型中就需要两个），所以将两者进行了拆分。同时，待模型训练完成后，在 inference 过程中**Encoder 只需要执行一次**，所以在此过程中也需要单独使用 Transformer 中的 Encoder 和 Decoder。

首先，我们需要定义一个名为 TranslationModel 的类，其前向传播过程代码如下所示：

```
1 class TranslationModel(nn.Module):
2     def __init__(self, src_vocab_size, tgt_vocab_size,
3                  d_model=512, nhead=8, num_encoder_layers=6,
4                  num_decoder_layers=6, dim_feedforward=2048,
5                  dropout=0.1):
6         super(TranslationModel, self).__init__()
7         self.my_transformer = MyTransformer(
8                                     d_model=d_model,nhead=nhead,
9                                     num_encoder_layers=num_encoder_layers,
10                                    num_decoder_layers=num_decoder_layers,
11                                    dim_feedforward=dim_feedforward,
12                                    dropout=dropout)
13        self.pos_embedding = PositionalEncoding(
                                d_model=d_model, dropout=dropout)
14        self.src_token_embedding = TokenEmbedding(src_vocab_size, d_model)
15        self.tgt_token_embedding = TokenEmbedding(tgt_vocab_size, d_model)
16        self.classification = nn.Linear(d_model, tgt_vocab_size)
17
18    def forward(self, src=None, tgt=None, src_mask=None,
19                tgt_mask=None, memory_mask=None, src_key_padding_mask=None,
20                tgt_key_padding_mask=None, memory_key_padding_mask=None):
21        src_embed = self.src_token_embedding(src)
          # [src_len, batch_size, embed_dim]
22        src_embed = self.pos_embedding(src_embed)
          # [src_len, batch_size, embed_dim]
23        tgt_embed = self.tgt_token_embedding(tgt)
          # [tgt_len, batch_size, embed_dim]
24        tgt_embed = self.pos_embedding(tgt_embed)
          # [tgt_len, batch_size, embed_dim]
25        outs = self.my_transformer(src=src_embed, tgt=tgt_embed,
                                     src_mask=src_mask,
26                                 tgt_mask=tgt_mask,memory_mask=memory_mask,
27                                 src_key_padding_mask=src_key_padding_mask,
28                                 tgt_key_padding_mask=tgt_key_padding_mask,
29                                 memory_key_padding_mask=memory_key_padding_mask)
30                                 # [tgt_len,batch_size,embed_dim]
31        logits = self.classification(outs) #[tgt_len,batch_size,tgt_vocab_size]
32        return logits
```

在上述代码中，第 7-12 行便是用来定义一个 Transformer 结构；第 13-16 行分别用来定义 Positional Embedding、Token Embedding 和最后的分类器；第 18 行中 src 是编码器的输入，形状为[src_len, batch_size]；tgt 是解码器的输入，形状为

[tgt_len, batch_size]；src_key_padding_mask 是用来 Mask 掉 Encoder 中不同序列的 padding 部分，形状为[batch_size,s rc_len]；tgt_key_padding_mask 是用来 Mask 掉 Decoder 中不同序列的 padding 部分，形状为[batch_size, tgt_len]；memory_key_padding_mask 用来 Mask 掉 Encoder 输出的 memory 中不同序列的 padding 部分，实质上就是 src_key_padding_mask。第 21-31 行便是用来执行整个前向传播过程，其中 Transformer 的整个前向传播过程在第 4 节中已经介绍过，在这里就不再赘述。

在定义完 logits 的前向传播过后，便可以通过如下形式进行使用：

```
1 if __name__ == '__main__':
2     src_len = 7
3     batch_size = 2
4     dmodel = 32
5     tgt_len = 8
6     num_head = 4
7     src = torch.tensor([[4, 3, 2, 6, 0, 0, 0],
8                         [5, 7, 8, 2, 4, 0, 0]]).transpose(0, 1)
      # 转换成 [src_len, batch_size]
9     src_key_padding_mask = torch.tensor([
                              [True, True, True, True, False, False, False],
10                            [True, True, True, True, True, False, False]])
11
12    tgt = torch.tensor([[1, 3, 3, 5, 4, 3, 0, 0],
13                        [1, 6, 8, 2, 9, 1, 0, 0]]).transpose(0, 1)
14    tgt_key_padding_mask = torch.tensor([
                          [True, True, True, True, True, True, False, False],
15                        [True, True, True, True, True, True, False, False]])
16
17    trans_model = TranslationModel(src_vocab_size=10, tgt_vocab_size=15,
18                  d_model=dmodel,nhead=num_head,num_encoder_layers=6,
19                  num_decoder_layers=6, dim_feedforward=30,dropout=0.1)
20    tgt_mask = trans_model.my_transformer.
                                      generate_square_subsequent_mask(tgt_len)
21    logits = trans_model(src, tgt=tgt, tgt_mask=tgt_mask,
                           src_key_padding_mask=src_key_padding_mask,
22                         tgt_key_padding_mask=tgt_key_padding_mask,
23                         memory_key_padding_mask=src_key_padding_mask)
24    print(logits.shape)
       # torch.Size([8, 2, 15]) [tgt_len,batch_size,tgt_vocab_size]
```

接着，我们需要再分别定义一个 Encoder 和 Decoder，以便在 inference 过程中使用，代码如下：

```
1     def encoder(self, src):
2         src_embed=self.src_token_embedding(src)
           # [src_len, batch_size, embed_dim]
3         src_embed=self.pos_embedding(src_embed)
           # [src_len, batch_size, embed_dim]
4         memory = self.my_transformer.encoder(src_embed)
5         return memory
6
7     def decoder(self, tgt, memory):
8         tgt_embed=self.tgt_token_embedding(tgt)
           # [tgt_len, batch_size, embed_dim]
9         tgt_embed=self.pos_embedding(tgt_embed)
           # [tgt_len, batch_size, embed_dim]
10        outs = self.my_transformer.decoder(tgt_embed, memory=memory)
11        # [tgt_len,batch_size,embed_dim]
12        return outs
```

在上述代码中，第 1-5 行用于在 inference 时对输入序列进行编码并得到 memory（**只需要执行一次**）；第 7-11 行用于根据 memory 和当前解码时刻的输入对输出进行预测，需要循环执行多次，这部分内容详见模型预测部分。

5.2.2 模型训练

在定义完成整个翻译模型的网络结构后下面就可以开始训练模型了。由于这部分代码较长，所以下面掌柜依旧以分块的形式进行介绍：

第 1 步：载入数据集

```
1 def train_model(config):
2     data_loader = LoadEnglishGermanDataset(config.train_corpus_file_paths,
3                                            batch_size=config.batch_size,
4                                            tokenizer=my_tokenizer)
5     train_iter, valid_iter, test_iter = \
6         data_loader.load_train_val_test_data(config.train_corpus_file_paths,
7                                              config.val_corpus_file_paths,
8                                              config.test_corpus_file_paths)
```

首先我们可以根据前面的介绍，通过类 LoadEnglishGermanDataset 来载入数据集，其中 config 中定义了模型所涉及到的所有配置参数。

第 2 步：定义模型并初始化权重

```
1     translation_model =
                TranslationModel(src_vocab_size=len(data_loader.de_vocab),
2                                tgt_vocab_size=len(data_loader.en_vocab),
3                                d_model=config.d_model,
4                                nhead=config.num_head,
5                                num_encoder_layers=config.num_encoder_layers,
```

```
                                num_decoder_layers=config.num_decoder_layers,
                                dim_feedforward=config.dim_feedforward,
                                dropout=config.dropout)
    for p in translation_model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
```

在载入数据后，便可以定义一个翻译模型 TranslationModel，并根据相关参数对其进行实例化；同时，可以对整个模型中的所有参数进行一个初始化操作。

第 3 步：定义损失学习率与优化器

```
1   loss_fn = torch.nn.CrossEntropyLoss(ignore_index=data_loader.PAD_IDX)
2   learning_rate = CustomSchedule(config.d_model)
3   optimizer = torch.optim.Adam(translation_model.parameters(),
4                                lr=config.warm_up_learning_rate,
5                                betas=(config.beta1, config.beta2),
                                 eps=config.epsilon)
```

在上述代码中，第 1 行是定义交叉熵损失函数，并同时指定需要忽略的索引 ignore_index。因为根据图 5-3 的 tgt_output 可知，有些位置上的标签值其实是 Padding 后的结果，因此在计算损失的时候需要将这些位置给忽略掉。第 2 行代码则是论文中所提出来的动态学习率计算过程，其计算公式为：

$$\text{lrate} = d_{model}^{-0.5} \cdot \min(\text{step\_num}^{-0.5}, \text{step\_num} \cdot \text{warmup\_steps}^{-1.5}) \tag{5.1}$$

```
class CustomSchedule(nn.Module):
    def __init__(self, d_model, warmup_steps=4000):
        super(CustomSchedule, self).__init__()
        self.d_model = torch.tensor(d_model, dtype=torch.float32)
        self.warmup_steps = warmup_steps
        self.step = 1.

    def __call__(self):
        arg1 = self.step ** -0.5
        arg2 = self.step * (self.warmup_steps ** -1.5)
        self.step += 1.
        return (self.d_model ** -0.5) * min(arg1, arg2)
```

通过 CustomSchedule，就能够在训练过程中动态的调整学习率。学习率随 step 增加而变换的结果如图 5-5 所示：

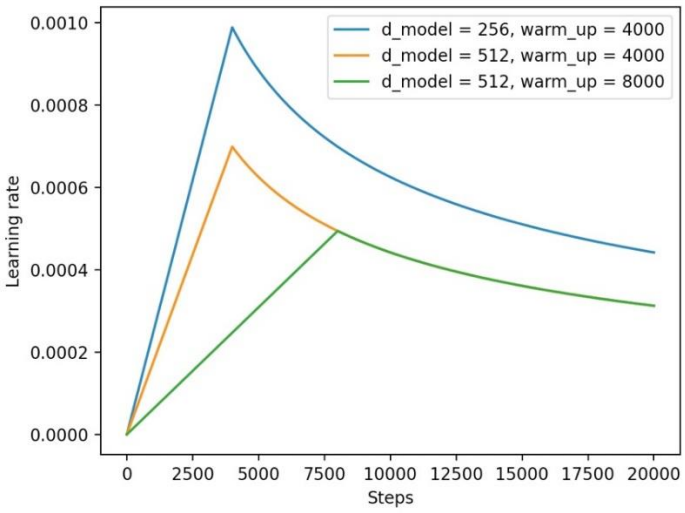

图 5-5. 动态学习率变化过程图

从图 5-5 可以看出，在前 warm_up 个 step 中，学习率是线性增长的，在这之后便是非线性下降，直至收敛与 0.0004。

第 4 步：开始训练

```
for epoch in range(config.epochs):
    losses = 0
    start_time = time.time()
    for idx, (src, tgt) in enumerate(train_iter):
        src = src.to(config.device)  # [src_len, batch_size]
        tgt = tgt.to(config.device)
        tgt_input = tgt[:-1, :]  # 解码部分的输入, [tgt_len,batch_size]
        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask \
            = data_loader.create_mask(src, tgt_input, config.device)
        logits = translation_model(
                  src=src,  # Encoder 的 token 序列输入，[src_len,batch_size]
                  tgt=tgt_input, # Decoder 的 token 序列输入,[tgt_len,batch_size]
                  src_mask=src_mask, # Encoder 的注意力 Mask 输入，这部分其实对于
                                                          # Encoder 来说是没有用的
                  tgt_mask=tgt_mask, # Decoder 的注意力 Mask 输入，用于掩盖当前
                                                  # position 之后的 position [tgt_len,tgt_len]
                  src_key_padding_mask=src_padding_mask,
                 # 用于 mask 掉 Encoder 的 Token 序列中的 padding 部分
                  tgt_key_padding_mask=tgt_padding_mask,
                 # 用于 mask 掉 Decoder 的 Token 序列中的 padding 部分
                  memory_key_padding_mask=src_padding_mask)
                 # 用于 mask 掉 Encoder 的 Token 序列中的 padding 部分
         # logits 输出 shape 为[tgt_len,batch_size,tgt_vocab_size]

        optimizer.zero_grad()
        tgt_out = tgt[1:, :]  # 解码部分的真实值  shape: [tgt_len,batch_size]
```

```
23          loss = loss_fn(logits.reshape(-1, logits.shape[-1]),
                           tgt_out.reshape(-1))
24          # [tgt_len*batch_size, tgt_vocab_size] with [tgt_len*batch_size, ]
25          loss.backward()
26          lr = learning_rate()
27          for p in optimizer.param_groups:
28              p['lr'] = lr
29          optimizer.step()
30          losses += loss.item()
31          acc, _, _ = accuracy(logits, tgt_out, data_loader.PAD_IDX)
32          print(f"Epoch: {epoch}, Train loss :{loss.item():.3f}, Train acc: {acc}")
```

在上述代码中，第 5-9 行是用来得到模型各个部分的输入；第 10-18 行是计算模型整个前向传播的过程；第 21-25 行则是执行损失计算与反向传播；第 27-29 则是将每个 step 更新后的学习率送入到模型中并更新参数；第 31 行是用来计算模型预测的准确率，具体过程将在后续文章中进行介绍。以下便是模型训练过程中的输出：

```
1 Epoch: 2, Train loss: 5.685, Train acc: 0.240947
2 Epoch: 2, Train loss: 5.668, Train acc: 0.241493
3 Epoch: 2, Train loss: 5.714, Train acc: 0.224682
4 Epoch: 2, Train loss: 5.660, Train acc: 0.235888
5 Epoch: 2, Train loss: 5.584, Train acc: 0.242052
6 Epoch: 2, Train loss: 5.611, Train acc: 0.243428
```

5.2.3 模型预测

在介绍完模型的训练过程后接下来就来看模型的预测部分。生成模型的预测部分不像普通的分类任务只需要将网络最后的输出做 argmax 操作即可，生成模型在预测过程中往往需要按时刻一步步进行来进行。因此，下面我们这里定义一个 translate 函数来执行这一过程，具体代码如下：

```
1  def translate(model, src, data_loader, config):
2      src_vocab = data_loader.de_vocab
3      tgt_vocab = data_loader.en_vocab
4      src_tokenizer = data_loader.tokenizer
5      model.eval()
6      tokens =[src_vocab.stoi[tok] for tok in src_tokenizer(src)] # 构造一个样本
7      num_tokens = len(tokens)
8      src = (torch.LongTensor(tokens).reshape(num_tokens,1))
              # 将 src_len 作为第一个维度
9      tgt_tokens = greedy_decode(model, src, max_len=num_tokens + 5,
10                                start_symbol=data_loader.BOS_IDX,config=config,
11                                data_loader=data_loader).flatten() # 解码的预测结果
12     return " ".join([tgt_vocab.itos[tok] for tok in
                 tgt_tokens]).replace("<bos>", "").replace("<eos>", "")
```

在上述代码中，第 6 行是将待翻译的源序列进行序列化操作；第 8-11 行则是通过函数 greedy_decode 函数来对输入进行解码；第 12 行是将最后解码后的结果由 Token 序列在转换成实际的目标语言。同时，greedy_decode 函数的实现如下：

```
1  def greedy_decode(model, src, max_len, start_symbol, config, data_loader):
2      src = src.to(config.device)
3      memory = model.encoder(src)  # 对输入的 Token 序列进行解码翻译
4      ys = torch.ones(1, 1).fill_(start_symbol). \
5          type(torch.long).to(config.device)  # 解码的第一个输入，起始符号
6      for i in range(max_len - 1):
7          memory = memory.to(config.device)
8          out = model.decoder(ys, memory) # [tgt_len, 1,  embed_dim]
9          out = out.transpose(0, 1)  # [1, tgt_len, embed_dim]
10         prob = model.classification(out[:, -1])  # 只对对预测的下一个词进行分类
11         _, next_word = torch.max(prob, dim=1)  # 选择概率最大者
12         next_word = next_word.item()
13         ys = torch.cat([ys, torch.ones(1, 1).type_as(src.data)
               .fill_(next_word)],dim=0)
14         # 将当前时刻解码的预测输出结果，同之前所有的结果堆叠作为输入再去预测下一个词。
15         if next_word == data_loader.EOS_IDX:
           # 如果当前时刻的预测输出为结束标志，则跳出循环结束预测。
16             break
17     return ys
```

在上述代码中，第 3 行是将源序列输入到 Transformer 的编码器中进行编码并得到 Memory；第 4-5 行是初始化解码阶段输入的第 1 个时刻的，在这里也就是'<sos>'；第 6-17 行则是整个循环解码过程，在下一个时刻为 EOS_IDX 或者达到最大长度后停止；第 8 行是根据 memory 以及当前时刻的输入对当前时刻的输出进行解码；第 9-12 行则是分类得到当前时刻的解码输出结果；第 13 行则是将当前时刻的解码输出结果头当前时刻之前所有的输入进行拼接，以此再对下一个时刻的输出进行预测。

最后，我们只需要调用如下函数便可以完成对原始输入语言的翻译任务：

```
1  def translate_german_to_english(src, config):
2      data_loader = LoadEnglishGermanDataset(config.train_corpus_file_paths,
3                                             batch_size=config.batch_size,
4                                             tokenizer=my_tokenizer)
5      translation_model = TranslationModel(
                                  src_vocab_size = len(data_loader.de_vocab),
6                                 tgt_vocab_size=len(data_loader.en_vocab),
7                                 d_model=config.d_model,
8                                 nhead=config.num_head,
9                                 num_encoder_layers=config.num_encoder_layers,
10                                num_decoder_layers=config.num_decoder_layers,
```

```
                                dim_feedforward=config.dim_feedforward,
                                dropout=config.dropout)
    translation_model = translation_model.to(config.device)
    torch.load(config.model_save_dir + '/model.pkl')
    r = translate(translation_model, src, data_loader, config)
    return r

if __name__ == '__main__':
    srcs = ["Eine Gruppe von Menschen steht vor einem Iglu.",
         "Ein Mann in einem blauen Hemd steht auf einer Leiter und putzt ein Fenster."]
    tgts = ["A group of people are facing an igloo.",
            "A man in a blue shirt is standing on a ladder cleaning a window."]
    config = Config()
    for i, src in enumerate(srcs):
        r = translate_german_to_english(src, config)
        print(f"德语: {src}")
        print(f"翻译: {r}")
        print(f"英语: {tgts[i]}")
```

在上述代码中，第 5-14 行是定义网络结构，以及恢复本地保存的网络权重；第 15 行则是开始执行翻译任务；第 19-28 行为翻译示例，其输出结果为：

```
德语: Eine Gruppe von Menschen steht vor einem Iglu.
翻译: A group of people standing in fraon of an igloo .
英语: A group of people are facing an igloo.
=========
德语: Ein Mann in einem blauen Hemd steht auf einer Leiter und putzt ein Fenster.
翻译: A man in a blue shirt is standing on a ladder cleaning a window.
英语: A man in a blue shirt is standing on a ladder cleaning a window.
```

第 6 节 基于 Transformer 的分类模型

经过前面几节内容的介绍，相信大家对于 Transformer 的原理应该有了一个比较清晰的认识。不过要想做到灵活运用 Transformer 结构，那就还得再看看其它情况下的运用场景。在这部分内容中，掌柜将会以 AG_News 数据集为例，来搭建一个基于 Transformer 结构的文本分类模型。

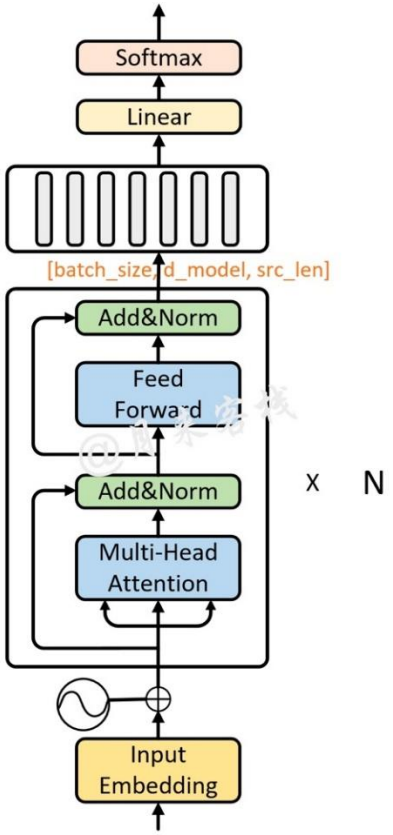

图 6-1. 基于 Transformer Encoder 的文本分类网络结构图

如图 6-1 所示便是一个基于 Transformer 结构的文本分类模型。不过准确的说应该只是一个基于 Transformer 中 Encoder 的文本分类模型。这是因为在文本分类任务中并没有解码这一过程，所以我们只需要将 Encoder 编码得到的向量输入到分类器中进行分类即可。同时需要注意的是，Encoder 部分最后输出张量的形状为[batch_size,d_model,src_len]（图 6-1 中 Encoder 输出的 src_len 为 7），我们需要根据相应策略来进行下一步的处理，具体见后文。

这里值得一提的是，其实 BERT 模型的网络结构本质上就等同于图 6-1 所示的网络结构，看懂了图 6-1 所示的结果对于后续理解 BERT 问题就不大了。

本部分完整代码可参见[12]。

6.1 数据预处理

6.1.1 语料介绍

在正式介绍模型之前，我们还是先来看看后续所要用到的 AG_News 数据集。AG_News 新闻主题分类数据集是通过从原始语料库中选择 4 个最大的类构建的。每个类包含 30000 个训练样本和 1900 个测试样本。训练样本总数为 120000 条，测试总数为 7600 条。AG_News 原始数据大概长这样：

```
1 "3","Fears for T N pension after talks","Unions representing workers at Turner   Newall say they are 'disappointed' after talks with stricken parent firm Federal Mogul."
2 "4","The Race is On: Second Private Team Sets Launch Date for Human Spaceflight (SPACE.com)","SPACE.com - TORONTO, Canada -- A second\team of rocketeers competing for the  #36;10 million Ansari X Prize, a contest for \privately funded suborbital space flight, has officially announced the first\launch date for its manned rocket."
3 "4","Ky. Company Wins Grant to Study Peptides (AP)","AP - A company founded by a chemistry researcher at the University of Louisville won a grant to develop a method of producing better peptides, which are short chains of amino acids, the building blocks of proteins."
```

上述一共包含有 3 个样本，每 1 行为 1 个样本。同时，所有样本均使用逗号作为分隔符，一共包含有 3 列，分别对应类标（1 到 4）、标题和新闻描述。在本篇文中，我们暂时只使用新闻描述作为输入（当然也可以用 title 作为输入来进行分类）。

对于该数据的载入，你可以使用 Pytorch 中的方法来下载并使用[13]：

```
1 from torchtext.datasets import AG_NEWS
2 train_iter = AG_NEWS(split='train')
```

也可以自己下载原始数据来进行处理。在这里为了延续使用与熟悉第 5 节中介绍的预处理代码，所以这里我们暂不使用 Pytorch 内置的代码。

6.1.2 数据集构建

由于分类模型数据集的构建过程并不复杂，其大部分工作在第 5 节中其实已经介绍过了，所以这里掌柜就只是简单的介绍一下。

第 1 步：定义 tokenize

对于英文语料的处理，可以继续沿用 5.1.3 节中对英文语料的类似处理方式，实现代码如下：

```
1 def my_tokenizer(s):
2     tokenizer = get_tokenizer('basic_english')
3     return tokenizer(s)
```

第 2 步：定义字符串清理

从 6.1.1 节中的示例语料中可以看到， 原始语料中有很多奇奇怪怪的字符，因此还需要对其稍微做一点处理。例如①只保留字母、数字、以及常用标点；②全部转换为小写字等。当然，你也可以自己再添加其它处理方式。具体代码如下：

```
1 def clean_str(string):
2     string = re.sub("[^A-Za-z0-9\-\?\!\.\,]", " ", string).lower()
3     return string
```

第 3 步：建立词表

在介绍完 tokenize 和字符串清理的实现方法后，我们就可以正式通过 torchtext.vocab 中的 Vocab 方法来构建词典了，代码如下：

```
1 def build_vocab(tokenizer, filepath, min_freq, specials=None):
2     if specials is None:
3         specials = ['<unk>', '<pad>']
4     counter = Counter()
5     with open(filepath, encoding='utf8') as f:
6         for string_ in tqdm(f):
7             string_ = string_.strip().split('","')[-1][:-1]
                  # 取标签和新闻描述
8             counter.update(tokenizer(clean_str(string_)))
9     return Vocab(counter, min_freq=min_freq, specials=specials)
```

在上述代码中，tokenizer 表示所使用的分词器； min_freq 表示最小字频，去掉小于 min_freq 的字。第 3 行代码用来指定特殊的字符；第 5-8 行代码用来遍历文件中的每一个样本(每行一个)并进行 tokenize 和计数，其中对于 counter.update 进行介绍可以参考[10]；第 9 行则是返回最后得到词典。

在完成上述过程后，我们将得到一个 Vocab 类的实例化对象：

```
1 {'<unk>': 0, '<pad>': 1, 'the': 2, '.': 3, ',': 4, 'a': 5, 'to': 6,'of': 7,
'in': 8, 'and': 9, 'on': 10, 's': 11, 'for': 12, '-': 13, '39': 14, 'that':
15,.....}
```

接下来，我们就需要定义一个类，并在类的初始化过程中根据训练语料完成字典的构建，代码如下：

```
 1 class LoadSentenceClassificationDataset():
 2     def __init__(self, train_file_path=None,  # 训练集路径
 3                  tokenizer=None, batch_size=20,
 5                  min_freq=1,  # 最小词频，去掉小于 min_freq 的词
 6                max_sen_len='same'): # 最大句子长度，默认为整个数据集中最长样本长度
 7         # max_sen_len = None 时，表示按每个 batch 中最长的样本长度进行 padding
 8         # 根据训练预料建立字典
 9           self.tokenizer = tokenizer
10           self.min_freq = min_freq
```

```
        self.specials = ['<unk>', '<pad>']
        self.vocab = build_vocab(self.tokenizer, filepath=train_file_path,
                                 min_freq=self.min_freq,specials=self.specials)
        self.PAD_IDX = self.vocab['<pad>']
        self.UNK_IDX = self.vocab['<unk>']
        self.batch_size = batch_size
        self.max_sen_len = max_sen_len
```

第 4 步：转换为 Token 序列

在得到构建的字典后，便可以通过如下函数来将训练集和测试集转换成 Token 序列：

```
    def data_process(self, filepath):
        """
        将每一句话中的每一个词根据字典转换成索引的形式，同时返回所有样本中最长样本的长度
        :param filepath: 数据集路径
        :return:
        """

        raw_iter = iter(open(filepath, encoding="utf8"))
        data = []
        max_len = 0
        for raw in tqdm(raw_iter):
            line = raw.rstrip("\n").split(",")
            s, l = line[-1][:-1], line[0][1:]
            s = clean_str(s)
            tensor_ = torch.tensor([self.vocab[token] for token in
                                    self.tokenizer(s)], dtype=torch.long)
            l = torch.tensor(int(l) - 1, dtype=torch.long) # 标签
            max_len = max(max_len, tensor_.size(0))
            data.append((tensor_, l))
        return data, max_len
```

在上述代码中，第 11-4 行分别用来将原始输入序列转换为对应词表中的 Token 形式。在处理完成后，就会得到类似如下的结果：

```
[(tensor([ 25,   65,   45, 1487,    5, 4062, 3291,   10, 2918, 20217,
            4, 1842, 4512, 1161,   15,  143,  140, 3658, 21658,  4762,
           40,  146,  409,   22,    8,   25,   65,    4,   16,     5,
          142,  287,   15,    4,  633,   39,  146,  409,   22,  5474,
            3,   40]), tensor(1)), .....]
```

第 5 步：padding 处理

由于对于不同的样本来说其对应的序列长度通常来说都是不同的，但是在将数据输入到相应模型时却需要保持同样的长度。因此在这里我们就需要对 Token 序列化后的样本进行 padding 处理，具体代码如下：

```
def pad_sequence(sequences,batch_first=False, max_len=None,padding_value=0):
    max_size = sequences[0].size()
    trailing_dims = max_size[1:]
     length = max_len
    max_len = max([s.size(0) for s in sequences])
    if length is not None:
        max_len = max(length, max_len)
    if batch_first:
        out_dims = (len(sequences), max_len) + trailing_dims
    else:
        out_dims = (max_len, len(sequences)) + trailing_dims
    out_tensor = sequences[0].data.new(*out_dims).fill_(padding_value)
    for i, tensor in enumerate(sequences):
        length = tensor.size(0)
        # use index notation to prevent duplicate references to the tensor
        if batch_first:
            out_tensor[i, :length, ...] = tensor
        else:
            out_tensor[:length, i, ...] = tensor
    return out_tensor
```

在上述代码中，max_len 表示 最大句子长度，默认为 None，即在每个 batch 中以最长样本的长度对其它样本进行 padding；当然同样也可以指定 max_len 的值为整个数据集中最长样本的长度进行 padding 处理。padding 处理后的结果类似如下：

```
tensor([[    2,   342,   578,  ...,     1,     1,     1],
        [   32,    13, 14585,  ...,     1,     1,     1],
        [ 1189,    11,   327,  ...,     1,     1,     1],...)
```

末尾的 1 即是 padding 的部分。

在定义完 pad_sequence 这个函数后，我们便可以通过它来对每个 batch 中的数据集进行 padding 处理：

```
    def generate_batch(self, data_batch):
        batch_sentence, batch_label = [], []
        for (sen, label) in data_batch: # 开始对一个 batch 中的每一个样本进行处理。
            batch_sentence.append(sen)
            batch_label.append(label)
        batch_sentence = pad_sequence(batch_sentence,  # [batch_size,max_len]
                                      padding_value=self.PAD_IDX,
                                      batch_first=False,
                                      max_len=self.max_sen_len)
        batch_label = torch.tensor(batch_label, dtype=torch.long)
        return batch_sentence, batch_label
```

第 6 步：构造 DataLoade 与使用示例

经过前面 5 步的操作，整个数据集的构建就算是已经基本完成了，只需要再构造一个 DataLoader 迭代器即可，代码如下：

```
1     def load_train_val_test_data(self, train_file_paths, test_file_paths):
2         train_data, max_sen_len = self.data_process(train_file_paths)
          # 得到处理好的所有样本
3         if self.max_sen_len == 'same':
4             self.max_sen_len = max_sen_len
5         test_data, _ = self.data_process(test_file_paths)
6         train_iter = DataLoader(train_data, batch_size=self.batch_size,
7                                 shuffle=True,      # 构造 DataLoader
                                  collate_fn=self.generate_batch)
8         test_iter = DataLoader(test_data, batch_size=self.batch_size,
9                                shuffle=True, collate_fn=self.generate_batch)
10        return train_iter, test_iter
```

在上述代码中，第 2-5 行便是分别用来将训练集和测试集转换为 Token 序列；第 6-9 行则是分别构造 2 个 DataLoader，其中 generate_batch 将作为一个参数传入来对每个 batch 的样本进行处理。在完成类 LoadSentenceClassificationDataset 所有的编码过程后，便可以通过如下形式进行使用：

```
1 if __name__ == '__main__':
2     path = "./data/ag_news_csv/test.csv"
3     data_loader = LoadSentenceClassificationDataset(train_file_path=path,
4                                                     tokenizer=my_tokenizer,
5                                                     max_sen_len=None)
6     train_iter, test_iter = data_loader.load_train_val_test_data(path, path)
7     for sample, label in train_iter:
8         print(sample.shape)  # [seq_len,batch_size]
```

最后，由于 Encoder 只会在 padding 部分有 mask 操作，所以每个样本的 key_padding_mask 向量我们在训练部分再生成即可。下面，我们正式进入到文本分类模型部分的介绍。

6.2 文本分类模型

6.2.1 网络结构

总体来说，基于 Transformer 文本分类模型的网络结构其实就是图 6-1 所展示的所有部分，当然你还可以使用多个 Encoder 进行堆叠。最后，只需要将 Encoder 的输出喂入到一个 softmax 分类器即可完成分类任务。不过这里有两个细节的地方需要大家注意：

①根据第 4 节的介绍可知，Encoder 在编码结束后输出的形状为 [src_len,batch_size,embed_dim]（这里的 src_len 也可以理解为 LSTM 中 time step

的概念）。因此，在构造最后分类器的输入时就可以有多种不同的形式，例如只取最后一个位置上的向量、或者是取所有位置向量的平均（求和）等都可以。后面掌柜也会将这 3 种方式都实现供大家参考。

②由于每个样本的长度各不相同，因此在对样本进行 padding 的时候就有两种方式。一般来说在大多数模型中多需要保持所有的样本具有相同的长度，不过由于这里我们使用的是自注意力的编码机制，因此只需要保持同一个 batch 中的样本长度一致即可。不过后面掌柜对这两种方式都进行了实现，只需要通过 max_sen_len 这个参数来控制即可。

首先，我们需要定义一个名为 ClassificationModel 的类，其前向传播过程代码如下所示：

```
1 class ClassificationModel(nn.Module):
2     def __init__(self, vocab_size=None,
3                  d_model=512, nhead=8,
4                  num_encoder_layers=6,
5                  dim_feedforward=2048,
6                  dim_classification=64,
7                  num_classification=4,
8                  dropout=0.1):
9         super(ClassificationModel, self).__init__()
10        self.pos_embedding = PositionalEncoding(d_model=d_model,
                                                  dropout=dropout)
11        self.src_token_embedding = TokenEmbedding(vocab_size, d_model)
12        encoder_layer = MyTransformerEncoderLayer(
13                                      d_model, nhead,dim_feedforward,dropout)
14                                      encoder_norm = nn.LayerNorm(d_model)
15        self.encoder = MyTransformerEncoder(encoder_layer,
16                                            num_encoder_layers,
                                            encoder_norm)
17        self.classifier = nn.Sequential(nn.Linear(d_model,
                          dim_classification),
18                          nn.Dropout(dropout),
19                          nn.Linear(dim_classification,num_classification))
```

在上述代码中，第 10-11 行定义了 Transformer 中的 Embedding 操作；第 12-13 行定义了 Transformer 中的 EncoderLayer；第 14-16 行定义了 Transformer 中的 Encoder；第 17-19 行定义了一个分类器。最后，整个网络的前向传播过程如下：

```
1     def forward(self,
2                 src,  # [src_len, batch_size]
3                 src_mask=None,
4                 src_key_padding_mask=None,  # [batsh_size, src_len]
5                 concat_type='sum'# 解码之后取所有位置相加，还是最后一个位置作为输出
6                 ):
```

```
7           src_embed = self.src_token_embedding(src)#[src_len,batch_size,embed_dim]
8           src_embed = self.pos_embedding(src_embed)#[src_len,batch_size,embed_dim]
9           memory = self.encoder(src=src_embed,
10                                mask=src_mask,
11                                src_key_padding_mask=src_key_padding_mask)
12          # [src_len,batch_size,embed_dim]
13          if concat_type == 'sum':
14              memory = torch.sum(memory, dim=0)
15          elif concat_type == 'avg':
16              memory = torch.sum(memory, dim=0) / memory.size(0)
17          else:
18              memory = memory[-1, ::]  # 取最后一个时刻
19          # [src_len, batch_size, num_heads * kdim] <==> [src_len,batch_size,embed_dim]
20          out = self.classifier(memory)  # 输出 logits
21          return out  # [batch_size, num_class]
```

在上述代码中，第 7-11 行用来执行编码器的前向传播过程；第 13-18 行便是用来选择以何种方式来选择分类器的输入，经掌柜实验后发现取各个位置的平均值效果最好；第 20-21 行便是将经过分类器后的输出进行返回。

在定义完 logits 的前向传播过后，便可以通过如下形式进行使用：

```
1 if __name__ == '__main__':
2     src_len = 7
3     batch_size = 2
4     dmodel = 32
5     num_head = 4
6     src = torch.tensor([[4, 3, 2, 6, 0, 0, 0],
7                         [5, 7, 8, 2, 4, 0, 0]]).transpose(0, 1)
      # 转换成 [src_len, batch_size]
8     src_key_padding_mask = torch.tensor([
                            [True, True, True, True, False, False, False],
9                           [True, True, True, True, True, False, False]])
10    model = ClassificationModel(vocab_size=10,d_model=dmodel,nhead=num_head)
11    logits = model(src, src_key_padding_mask=src_key_padding_mask)
12    print(logits.shape) #torch.Size([2, 4])
```

6.2.2 模型训练

在定义完成整个分类模型的网络结构后下面就可以开始训练模型了。由于这部分代码较长，所以下面掌柜依旧以分块的形式进行介绍：

第 1 步：载入数据集

```
1 def train_model(config):
2     data_loader = LoadSentenceClassificationDataset(
                                  config.train_corpus_file_paths,
3                                 my_tokenizer,
```

```
                                batch_size=config.batch_size,
                                min_freq=config.min_freq,
                                max_sen_len=config.max_sen_len)
    train_iter, test_iter = data_loader.load_train_val_test_data(
        config.train_corpus_file_paths, config.test_corpus_file_paths)
```

首先我们可以根据前面的介绍，通过类 LoadSentenceClassificationDataset 来载入数据集，其中 config 中定义了模型所涉及到的所有配置参数。同时，可以通过 max_sen_len 参数来控制 padding 时保持所有样本一样还是仅在每个 batch 内部一样。

第 2 步：定义模型并初始化权重

```
    classification_model = ClassificationModel(
                           vocab_size=len(data_loader.vocab),
                           d_model=config.d_model,
                           nhead=config.num_head,
                           num_encoder_layers=config.num_encoder_layers,
                           dim_feedforward=config.dim_feedforward,
                           dim_classification=config.dim_classification,
                           num_classification=config.num_class,
                           dropout=config.dropout)

    for p in classification_model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
```

在载入数据后，便可以定义一个文本分类模型 ClassificationModel，并根据相关参数对其进行实例化；同时，可以对整个模型中的所有参数进行初始化操作。

第 3 步：定义损失学习率与优化器

```
    loss_fn = torch.nn.CrossEntropyLoss()
    learning_rate = CustomSchedule(config.d_model)
    optimizer = torch.optim.Adam(classification_model.parameters(),
                                 lr=0.,
                                 betas=(config.beta1, config.beta2),
                                 eps=config.epsilon)
```

在上述代码中，第 1 行是定义交叉熵损失函数；第 2 行代码则是论文中所提出来的动态学习率计算过程，其计算公式和公式(5.1)没有差别。具体实现代码同 5.2.2 节中的一样，这里就不再赘述。

第 4 步：开始训练

```
    for epoch in range(config.epochs):
        losses = 0
        start_time = time.time()
        for idx, (sample, label) in enumerate(train_iter):
```

```
5             sample = sample.to(config.device)  # [src_len, batch_size]
6             label = label.to(config.device)
7             padding_mask = (sample == data_loader.PAD_IDX).transpose(0, 1)
8             logits = classification_model(sample,
9                                           src_key_padding_mask=padding_mask)
10                                          # [batch_size,num_class]
11            optimizer.zero_grad()
12            loss = loss_fn(logits, label)
13            loss.backward()
14            lr = learning_rate()
15            for p in optimizer.param_groups:
16                p['lr'] = lr
17            optimizer.step()
18            losses += loss.item()
19
20            acc = (logits.argmax(1) == label).float().mean()
21            if idx % 10 == 0:
22                print(f"Epoch: {epoch}, Batch[{idx}/{len(train_iter)}], "
23                      f"Train loss :{loss.item():.3f}, Train acc:{acc:.3f}")
24        end_time = time.time()
25        train_loss = losses / len(train_iter)
26        print(f"Epoch: {epoch}, Train loss: {train_loss:.3f}, Epoch time =
              {(end_time  start_time):.3f}s")
```

在上述代码中，第 7 行代码用来生成每个样本对应的 padding mask 向量；第 15-16 行是将每个 step 更新后的学习率送入到模型中。以下便是模型训练过程中的输出：

```
1     Epoch: 9, Batch: [410/469], Train loss 0.186, Train acc: 0.938
2     Epoch: 9, Batch: [420/469], Train loss 0.150, Train acc: 0.938
3     Epoch: 9, Batch: [430/469], Train loss 0.269, Train acc: 0.941
4     Epoch: 9, Batch: [440/469], Train loss 0.197, Train acc: 0.925
5     Epoch: 9, Batch: [450/469], Train loss 0.245, Train acc: 0.917
6     Epoch: 9, Batch: [460/469], Train loss 0.272, Train acc: 0.902
7     Accuracy on test 0.886
```

本部分完整代码可参见[12]。

第 7 节 基于 Transformer 的对联生成模型

经过前面 6 节内容的介绍，对于 Transformer 相信大家都应该理解得差不多了。不过要想做到灵活运用 Transformer 结构，那就还得多看看其它场景下的运用。在接下来的这部分内容中，掌柜将会以一个含有 70 余万条的对联数据集为例，来搭建一个基于 Transformer 结构的对联生成模型。同时，这也是介绍 Transformer 结构的最后一节内容。

如图 7-1 所示便是一个基于 Transformer 的对联生成模型。可以看出，其实它与前面介绍的基于 Transformer 结构的翻译模型没有特别的变化，唯一不同的可能就是在对联生成模型中解码器和编码器共用同一个词表（因为两者都是中文）。

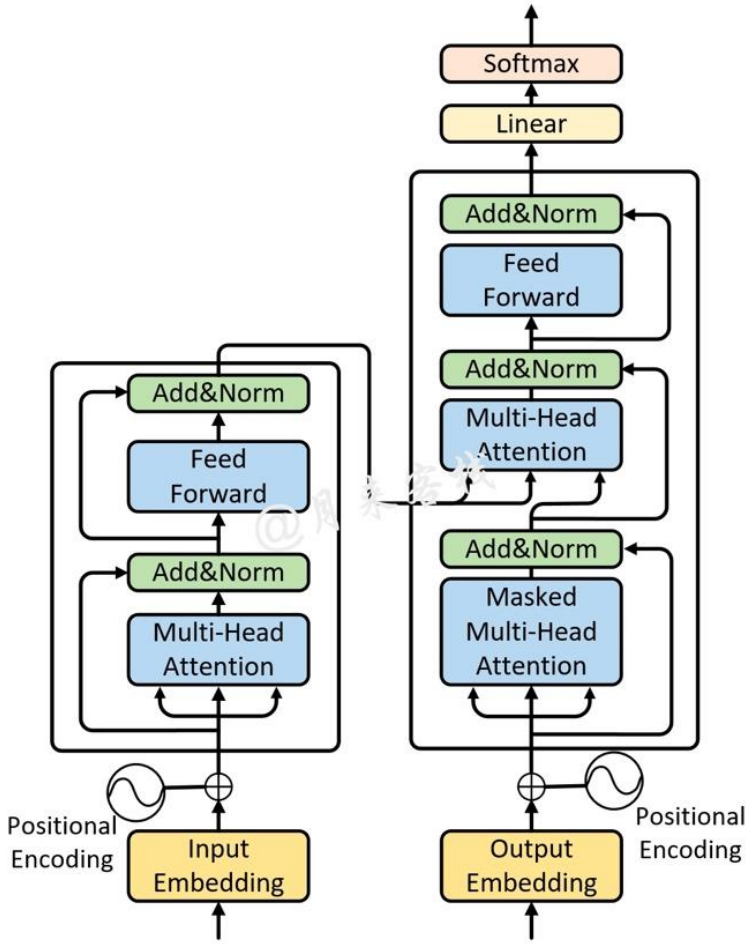

图 7-1. Transformer 网络结构图

7.1 数据预处理

7.1.1 语料介绍

按老规矩，在正式介绍模型搭建之前我们还是先来看看数据集都长什么样。本次所使用到的数据集是一个网上公开的对联数据集，在 github 中搜索"couplet-

dataset”就能找到，其一共包含有 770491 条训练样本，4000 条测试样本。和翻译数据集类似，对联数据集也包含上下两句：

```
# 上联 in.txt
风 弦 未 拨 心 先 乱
花 梦 粘 于 春 袖 口
晋 世 文 章 昌 二 陆
```

```
# 下联 out.txt
夜 幕 已 沉 梦 更 闲
莺 声 溅 落 柳 枝 头
魏 家 词 赋 重 三 曹
```

如上所示便是 3 条样本，分别存放在 in.txt 和 out.txt 这两个文件中。可以看出，原始数据已经做了分字这步操作，所有后续我们只需要进行简单的 split 操作即可。

7.1.2 数据集构建

总体上来说对联生成模型的数据集构建过程和翻译模型的数据集构建过程基本上没有太大差别，主要步骤同样也是：①构建字典；②将文本中的每一个词（字）转换为 Token 序列；③对不同长度的样本序列按照某个标准进行 padding 处理；④构建 DataLoader 类。

第 1 步：定义 tokenize

由于原始数据每个字已经被空格隔开了，所以这里 tokenizer 的定义只需要进行 split 操作即可，代码如下：

```
def my_tokenizer(s):
    return s.split()
```

可以看到，其实也非常简单。例如对于如下文本来说

```
腾 飞 上 铁 ， 锐 意 改 革 谋 发 展 ， 勇 当 千 里 马
```

其 tokenize 后的结果为：

```
['腾', '飞', '上', '铁', ',', '锐', '意', '改', '革', '谋', '发', '展', ',', '勇', '当', '千', '里', '马']
```

第 2 步：建立词表

在介绍完 tokenize 的实现方法后，我们就可以正式通过 torchtext.vocab 中的 Vocab 方法来构建词典了，代码如下：

```
def build_vocab(tokenizer, filepath, min_freq=1, specials=None):
    if specials is None:
        specials = ['<unk>', '<pad>', '<bos>', '<eos>']
    counter = Counter()
    with open(filepath[0], encoding='utf8') as f:
```

```
        for string_ in f:
            counter.update(tokenizer(string_))
    with open(filepath[1], encoding='utf8') as f:
        for string_ in f:
            counter.update(tokenizer(string_))
    return Vocab(counter, specials=specials, min_freq=min_freq)
```

在上述代码中，第 3 行代码用来指定特殊的字符；第 5-10 行分别用来遍历 in.txt 文件和 out.txt 文件中的每一个样本（每行一个）并进行 tokenize 和计数，其中对于 counter.update 进行介绍可以参考[10]；第 8 行则是返回最后得到词典。值得注意的是，由于在对联生成这一场景中编码器和解码器共用的是一个词表，所以这里同时对 in.txt 和 out.txt 文件进行了遍历。

在完成上述过程后，我们将得到一个 Vocab 类的实例化对象，即：

```
{'<unk>': 0, '<pad>': 1, '<bos>': 2, '<eos>': 3, ',': 4, '风': 5, '春': 6, '一': 7, '人': 8, '月': 9, '山': 10, '心': 11, '花': 12, '天': 13, ...}
```

此时，我们就需要定义一个类，并在类的初始化过程中根据训练语料完成字典的构建，代码如下：

```
class LoadCoupletDataset():
    def __init__(self, train_file_paths=None, tokenizer=None,
                 batch_size=2, min_freq=1):
        # 根据训练预料建立字典，由于都是中文，所以共用一个即可
        self.tokenizer = tokenizer
        self.vocab = build_vocab(self.tokenizer, filepath=train_file_paths,
                                 min_freq=min_freq)
        self.specials = ['<unk>', '<pad>', '<bos>', '<eos>']
        self.PAD_IDX = self.vocab['<pad>']
        self.BOS_IDX = self.vocab['<bos>']
        self.EOS_IDX = self.vocab['<eos>']
        self.batch_size = batch_size
```

第 3 步：转换为 Token 序列

在得到构建的字典后，便可以通过如下函数来将训练集和测试集转换成 Token 序列：

```
    def data_process(self, filepaths):
        """
        将每一句话中的每一个词根据字典转换成索引的形式
        :param filepaths:
        :return:
        """
        raw_in_iter = iter(open(filepaths[0], encoding="utf8"))
        raw_out_iter = iter(open(filepaths[1], encoding="utf8"))
        data = []
```

```
        for (raw_in, raw_out) in zip(raw_in_iter, raw_out_iter):
            in_tensor_ = torch.tensor([self.vocab[token] for token in
                    self.tokenizer(raw_in.rstrip("\n"))], dtype=torch.long)
            out_tensor_ = torch.tensor([self.vocab[token] for token in
                    self.tokenizer(raw_out.rstrip("\n"))], dtype=torch.long)
            data.append((in_tensor_, out_tensor_))
        return data
```

在上述代码中，第 11-4 行分别用来将原始序列上联和目标序列下联转换为对应词表中的 Token 形式。在处理完成后，就会得到类似如下的结果：

```
[(tensor([5, 549, 250, 1758, 11, 228, 651]), tensor([154, 1420, 310, 598, 29, 206, 164])),
 (tensor([12, 29, 3218, 262, 6, 628, 419]), tensor([ 441, 62, 2049, 93, 66, 304, 111])),
 (tensor([1137, 40, 47, 286, 819, 364, 1383]), tensor([1803, 49, 586, 556, 126, 25, 1830])),
 (tensor([7, 291, 138, 115, 216, 151, 9]), tensor([15, 311, 107, 57, 80, 5, 21]))]
```

其中左边的一列就是原始序列上联的 Token 形式，右边一列就是目标序列下联的 Token 形式，每一行构成一个样本。

第 4 步：padding 处理

从上面的输出结果可以看到，无论是对于原始序列来说还是目标序列来说，在不同的样本中其对应长度都不尽相同。但是在将数据输入到相应模型时却需要保持同样的长度，因此在这里我们就需要对 Token 序列化后的样本进行 padding 处理。同时需要注意的是，一般在这种生成模型中，**模型在训练过程中只需要保证同一个 batch 中所有的原始序列等长，所有的目标序列等长即可**，也就是说不需要在整个数据集中所有样本都保证等长。

因此，在实际处理过程中无论是原始序列还是目标序列都会以每个 batch 中最长的样本为标准对其它样本进行 padding，具体代码如下：

```
    def generate_batch(self, data_batch):
        in_batch, out_batch = [], []
        for (in_item, out_item) in data_batch:
            # 开始对一个 batch 中的每一个样本进行处理。
            in_batch.append(in_item)  # 编码器输入序列不需要加起止符
            # 在每个 idx 序列的首位加上 起始 token 和 结束 token
            out = torch.cat([torch.tensor([self.BOS_IDX]),
                             out_item, torch.tensor([self.EOS_IDX])], dim=0)
            out_batch. append(out)
        # 以最长的序列为标准进行填充
        in_batch = pad_sequence(in_batch, padding_value=self.PAD_IDX)
        # [de_len,batch_size]
        out_batch = pad_sequence(out_batch, padding_value=self.PAD_IDX)
        # [en_len,batch_size]
        return in_batch, out_batch
```

在上述代码中，第 6-7 行用来在目标序列的首尾加上特定的起止符；第 10-11 行则是分别对一个 batch 中的原始序列和目标序列以各自当中最长的样本为标准进行 padding（这里的 pad_sequence 导入自 torch.nn.utils.rnn）。

第 5 步：构造 mask 向量

在处理完成前面几个步骤后，进一步需要根据 src_input 和 tgt_input 来构造相关的 mask 向量，具体代码如下：

```
1     def generate_square_subsequent_mask(self, sz, device):
2         mask = (torch.triu(torch.ones((sz, sz), device=device))
                                            ==1).transpose(0, 1)
3         mask = mask.float().masked_fill(mask == 0, float('- inf')).
4                                             masked_fill(mask == 1, float(0.0))
5         return mask
6     def create_mask(self, src, tgt, device='cpu'):
7         src_seq_len = src.shape[0]
8         tgt_seq_len = tgt.shape[0]
9         tgt_mask = self.generate_square_subsequent_mask(tgt_seq_len, device)
          # [tgt_len,tgt_len]
10        # Decoder 的注意力 Mask 输入，用于掩盖当前 position 之后的 position，所以这里是一
            个对称矩阵
11        src_mask = torch.zeros((src_seq_len, src_seq_len),
                                    device=device).type(torch.bool)
12        #Encoder 的注意力 Mask 输入，这部分其实对于 Encoder 来说是没有用的，所以这里全是 0
13        src_padding_mask = (src == self.PAD_IDX).transpose(0, 1)
14        # 用于 mask 掉 Encoder 的 Token 序列中的 padding 部分,[batch_size, src_len]
15        tgt_padding_mask = (tgt == self.PAD_IDX).transpose(0, 1)
16        # 用于 mask 掉 Decoder 的 Token 序列中的 padding 部分,batch_size, tgt_len
17        return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

在上述代码中，第 1-5 行是用来生成一个形状为[sz,sz]的注意力掩码矩阵，用于在解码过程中掩盖当前 position 之后的 position；第 6-17 行用来返回 Transformer 中各种情况下的 mask 矩阵，其中 src_mask 在这里并没有作用。

第 6 步：构造 DataLoade 与使用示例

经过前面 5 步的操作，整个数据集的构建就算是已经基本完成了，只需要再构造一个 DataLoader 迭代器即可，代码如下：

```
1     def load_train_val_test_data(self, train_file_paths, test_file_paths):
2         train_data = self.data_process(train_file_paths)
3         test_data = self.data_process(test_file_paths)
4         train_iter = DataLoader(train_data, batch_size=self.batch_size,
5                                 shuffle=True, collate_fn=self.generate_batch)
6         test_iter = DataLoader(test_data, batch_size=self.batch_size,
7                                shuffle=True, collate_fn=self.generate_batch)
8         return train_iter, test_iter
```

在上述代码中，第 2-3 行便是分别用来将训练集和测试集转换为 Token 序列；第 4-7 行则是分别构造 2 个 DataLoader，其中 generate_batch 将作为一个参数传入来对每个 batch 的样本进行处理。在完成类 LoadCoupletDataset 所有的编码过程后，便可以通过如下形式进行使用：

```
1 if __name__ == '__main__':
2     config = Config()
3     data_loader = LoadCoupletDataset(config.train_corpus_file_paths,
4                                      batch_size=config.batch_size,
5                                      tokenizer=my_tokenizer,
6                                      min_freq=config.min_freq)
7     train_iter, test_iter =
        data_loader.load_train_val_test_data(config.test_corpus_file_paths,
8                                            config.test_corpus_file_paths)
9     print(data_loader.PAD_IDX)
10    for src, tgt in train_iter:
11        tgt_input = tgt[:-1, :]
12        tgt_out = tgt[1:, :]
13        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask =
                                      data_loader.create_mask(src, tgt_input)
14        print(src)
15        print(tgt)
16        print("src shape: ", src.shape)  # [in_tensor_len,batch_size]
17        print("tgt shape:", tgt.shape)  # [in_tensor_len,batch_size]
18        print("src input shape:", src.shape)
19        print("src_padding_mask shape (batch_size, src_len): ",
                src_padding_mask.shape)
20        print("tgt input shape:", tgt_input.shape)
21        print("tgt_padding_mask shape: (batch_size, tgt_len) ",
                tgt_padding_mask.shape)
22        print("tgt output shape:", tgt_out.shape)
23        print("tgt_mask shape (tgt_len,tgt_len): ", tgt_mask.shape)
```

在介绍完数据集构建的整个过程后，下面就开始正式进入到翻译模型的构建中。如果对于这部分不是特别理解的话，建议先看第 5 节中的数据处理流程图进行理解。

7.2 对联生成模型

7.2.1 网络结构

总体来说，基于 Transformer 的对联生成模型的网络结构其实就是图 7-1 所展示的所有部分，只是在前面介绍 Transformer 网络结构时掌柜并没有把 Embedding 部分的实现给加进去。这是因为对于不同的文本生成模型，其 Embedding 部分会不一样（例如在本场景中编码器和解码器共用一个

TokenEmbedding 即可，而在翻译模型中就需要两个），所以将两者进行了拆分。同时，待模型训练完成后，在 inference 过程中 **Encoder 只需要执行一次**，所以在此过程中也需要单独使用 Transformer 中的 Encoder 和 Decoder。

首先，需要定义一个名为 CoupletModel 的类，其前向传播过程代码如下所示：

```
class CoupletModel(nn.Module):
    def __init__(self, vocab_size,
                 d_model=512, nhead=8, num_encoder_layers=6,
                 num_decoder_layers=6, dim_feedforward=2048,
                 dropout=0.1):
        super(CoupletModel, self).__init__()
        self.my_transformer = MyTransformer(d_model=d_model,
                                            nhead=nhead,
                                    num_encoder_layers=num_encoder_layers,
                                    num_decoder_layers=num_decoder_layers,
                                    dim_feedforward=dim_feedforward,
                                    dropout=dropout)
        self.pos_embedding = PositionalEncoding(d_model=d_model,
                                                dropout=dropout)
        self.token_embedding = TokenEmbedding(vocab_size, d_model)
        self.classification = nn.Linear(d_model, vocab_size)

    def forward(self, src=None, tgt=None, src_mask=None,
                tgt_mask=None, memory_mask=None, src_key_padding_mask=None,
                tgt_key_padding_mask=None, memory_key_padding_mask=None):
        """
        :param src: Encoder 的输入 [src_len,batch_size]
        :param tgt: Decoder 的输入 [tgt_len,batch_size]
        :param src_key_padding_mask: 用来 Mask 掉 Encoder 中不同序列的 padding 部
                                     分,[batch_size, src_len]
        :param tgt_key_padding_mask: 用来 Mask 掉 Decoder 中不同序列的 padding 部分
                                    [batch_size, tgt_len]
         memory_key_padding_mask: 用来 Mask 掉 Encoder 输出的 memory 中不同序列的 padding 部
                                  分 [batch_size, src_len]
        :return:
        """
         src_embed = self.token_embedding(src) # [src_len, batch_size, embed_dim]
         src_embed = self.pos_embedding(src_embed)#[src_len,batch_size,embed_dim]
         tgt_embed = self.token_embedding(tgt) # [tgt_len, batch_size, embed_dim]
         tgt_embed = self.pos_embedding(tgt_embed)#[tgt_len,batch_size,embed_dim]
         outs = self.my_transformer(src=src_embed,
                                    tgt=tgt_embed,src_mask=src_mask,
                                    tgt_mask=tgt_mask,
                                    memory_mask=memory_mask,
```

```
34                              src_key_padding_mask=src_key_padding_mask,
35                              tgt_key_padding_mask=tgt_key_padding_mask,
36                              memory_key_padding_mask=memory_key_padding_mask)
37          # [tgt_len,batch_size,embed_dim]
38          logits = self.classification(outs) # [tgt_len,batch_size,tgt_vocab_size]
39          return logits
```

在上述代码中，第 7-12 行便是用来定义一个 Transformer 结构；第 13-15 分别用来定义 Positional Embedding、Token Embedding 和最后的分类器（需要注意的是这里是共用同一个 Token Embedding）；第 28-38 行便是用来执行整个前向传播过程，其中 Transformer 的整个前向传播过程在第 4 节中已经介绍过，在这里就不再赘述。

在定义完 logits 的前向传播过后，便可以通过如下形式进行使用：

```
1  if __name__ == '__main__':
2      src_len = 7
3      batch_size = 2
4      dmodel = 32
5      tgt_len = 8
6      num_head = 4
7      src = torch.tensor([[4, 3, 2, 6, 0, 0, 0], # 转换成 [src_len, batch_size]
8                          [5, 7, 8, 2, 4, 0, 0]]).transpose(0, 1)
9      src_key_padding_mask = torch.tensor([
                            [True, True, True, True, False, False, False],
10                          [True, True, True, True, True, False, False]])
11     tgt = torch.tensor([[1, 3, 3, 5, 4, 3, 0, 0],
12                         [1, 6, 8, 2, 9, 1, 0, 0]]).transpose(0, 1)
13     tgt_key_padding_mask = torch.tensor([
                           [True, True, True, True, True, True, False, False],
14                         [True, True, True, True, True, True, False, False]])
15     trans_model = CoupletModel(vocab_size=10, d_model=dmodel,
16                                nhead=num_head, num_encoder_layers=6,
17                                num_decoder_layers=6, dim_feedforward=30,
18                                dropout=0.1)
19     tgt_mask = trans_model.
                       my_transformer.generate_square_subsequent_mask(tgt_len)
20     logits = trans_model(src, tgt=tgt, tgt_mask=tgt_mask,
21                          src_key_padding_mask=src_key_padding_mask,
22                          tgt_key_padding_mask=tgt_key_padding_mask,
23                          memory_key_padding_mask=src_key_padding_mask)
24     print(logits.shape)
```

接着，我们需要再定义一个 Encoder 和 Decoder 在 inference 中进行使用，代码如下：

```
    def encoder(self, src):
        src_embed = self.token_embedding(src) # [src_len, batch_size, embed_dim]
        src_embed = self.pos_embedding(src_embed)#[src_len,batch_size,embed_dim]
        memory = self.my_transformer.encoder(src_embed)
        return memory # [src_len, batch_size, embed_dim]

    def decoder(self, tgt, memory, tgt_mask):
        tgt_embed = self.tgt_token_embedding(tgt)#[tgt_len,batch_size,embed_dim]
        tgt_embed = self.pos_embedding(tgt_embed)#[tgt_len,batch_size,embed_dim]
        outs = self.my_transformer.decoder(tgt_embed, memory=memory,
                                           tgt_mask=tgt_mask)
        # [tgt_len,batch_size,embed_dim]
        return outs
```

在上述代码中，第 1-5 行用于在 inference 时对输入序列进行编码并得到 memory（**只需要执行一次**）；第 7-11 行用于根据 memory 和当前解码时刻的输入对输出进行预测，需要循环执行多次，这部分内容详见 7.2.3 模型预测部分。

7.2.2 模型训练

在定义完成整个对联生成模型的网络结构后下面就可以开始训练模型了。由于这部分代码较长，所以下面掌柜依旧以分块的形式进行介绍：

第 1 步：载入数据集

```
def train_model(config):
    data_loader = LoadCoupletDataset(config.train_corpus_file_paths,
                                     batch_size=config.batch_size,
                                     tokenizer=my_tokenizer,
                                     min_freq=config.min_freq)
    train_iter, test_iter = data_loader.load_train_val_test_data(
                config.train_corpus_file_paths,config.test_corpus_file_paths)
```

首先我们可以根据前面的介绍，通过类 LoadCoupletDataset 来载入数据集，其中 config 中定义了模型所涉及到的所有配置参数。

第 2 步：定义模型并初始化权重

```
    couplet_model = CoupletModel(vocab_size=len(data_loader.vocab),
                                 d_model=config.d_model,
                                 nhead=config.num_head,
                                 num_encoder_layers=config.num_encoder_layers,
                                 num_decoder_layers=config.num_decoder_layers,
                                 dim_feedforward=config.dim_feedforward,
                                 dropout=config.dropout)
    for p in couplet_model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
```

在载入数据后，便可以定义模型 CoupletModel，并根据相关参数对其进行实例化；同时，可以对整个模型中的所有参数进行初始化操作。

第 3 步：定义损失学习率与优化器

```
1    loss_fn = torch.nn.CrossEntropyLoss(ignore_index=data_loader.PAD_IDX)
2    learning_rate = CustomSchedule(config.d_model)
3    optimizer = torch.optim.Adam(couplet_model.parameters(),lr=0.,
4                                 betas=(config.beta1, config.beta2),
5                                 eps=config.epsilon)
```

在上述代码中，第 1 行是定义交叉熵损失函数，并同时指定需要忽略的索引 ignore_index 部分的损失。因为根据 tgt_output 可知，有些位置上的标签值其实是 Padding 后的结果，因此在计算损失的时候需要将这些位置给忽略掉。第 2 行代码则是论文中所提出来的动态学习率计算过程，具体在第 5 节中已经介绍过来，这里就不再赘述。

第 4 步：开始训练

```
1    for epoch in range(config.epochs):
2        losses = 0
3        start_time = time.time()
4        for idx, (src, tgt) in enumerate(train_iter):
5            src = src.to(config.device)  # [src_len, batch_size]
6            tgt = tgt.to(config.device)
7            tgt_input = tgt[:-1, :]  # 解码部分的输入, [tgt_len,batch_size]
8            src_mask, tgt_mask, src_padding_mask, tgt_padding_mask \
9                = data_loader.create_mask(src, tgt_input, config.device)
10           logits = couplet_model(
11               src=src,  # Encoder 的 token 序列输入，[src_len,batch_size]
12               tgt=tgt_input,  # Decoder 的 token 序列输入,[tgt_len,batch_size]
13               src_mask=src_mask,
                 # Encoder 的注意力 Mask 输入，这部分其实对于 Encoder 来说是没有用的
14               tgt_mask=tgt_mask, # Decoder 的注意力 Mask 输入，
15               # 用于掩盖当前 position 之后的 position[tgt_len,tgt_len]
16               src_key_padding_mask=src_padding_mask,
                 # 用于 mask 掉 Encoder 的 Token 序列中的 padding 部分
17               tgt_key_padding_mask=tgt_padding_mask,
                 # 用于 mask 掉 Decoder 的 Token 序列中的 padding 部分
18               memory_key_padding_mask=src_padding_mask)
19           # logits 输出 shape 为[tgt_len,batch_size,tgt_vocab_size]
20           optimizer.zero_grad()
21           tgt_out = tgt[1:, :]  # 解码部分的真实值  shape: [tgt_len,batch_size]
22           loss = loss_fn(logits.reshape(-1, logits.shape[-1]),
                                                   tgt_out.reshape(-1))
23           # [tgt_len*batch_size, tgt_vocab_size] with [tgt_len*batch_size, ]
24           loss.backward()
```

在上述代码中，第 5-9 行是用来得到模型各个部分的输入；第 10-18 行是计算模型整个前向传播的过程；第 20-24 行则是执行损失计算与反向传播。

以下便是模型训练过程中的输出：

```
1 -- INFO: Epoch: 0, Batch[29/3010], Train loss : 8.965, Train acc: 0.094
2 -- INFO: Epoch: 0, Batch[59/3010], Train loss : 8.618, Train acc: 0.098
3 -- INFO: Epoch: 0, Batch[89/3010], Train loss : 8.366, Train acc: 0.099
4 -- INFO: Epoch: 0, Batch[119/3010], Train loss : 8.137, Train acc: 0.109
5 ...
```

7.2.3 模型预测

在介绍完模型的训练过程后接下来就来看模型的预测部分。生成模型的预测部分不像普通的分类任务只需要将网络最后的输出做 argmax 操作即可，生成模型在预测过程中往往需要按时刻一步步进行来进行。因此，下面我们这里定义一个 couplet 函数来执行这一过程，具体代码如下：

```
1 def couplet(model, src, data_loader, config):
2     vocab = data_loader.vocab
3     tokenizer = data_loader.tokenizer
4     model.eval()
5     tokens = [vocab.stoi[tok] for tok in tokenizer(src)]  # 构造一个样本
6     num_tokens = len(tokens)
7     src = (torch.LongTensor(tokens).reshape(num_tokens, 1))
       # 将 src_len 作为第一个维度
8     tgt_tokens = greedy_decode(model, src, max_len=num_tokens + 5,
9                                start_symbol=data_loader.BOS_IDX,
                                 config=config,
10                               data_loader=data_loader).flatten()
                                       # 解码的预测结果
11    return "".join([vocab.itos[tok] for tok in tgt_tokens]).replace("<bos>",
            "").replace("<eos>", "")
```

在上述代码中，第 5 行是将待翻译的源序列进行序列化操作；第 7-10 行则是通过函数 greedy_decode 函数来对输入进行解码；第 11 行则是将最后解码后的结果由 Token 序列在转换成实际的目标语言。同时，greedy_decode 函数实现如下：

```
1 def greedy_decode(model, src, max_len, start_symbol, config, data_loader):
2     src = src.to(config.device)
3     memory = model.encoder(src)  # 对输入的 Token 序列进行解码翻译
4     ys = torch.ones(1, 1).fill_(start_symbol). \
5         type(torch.long).to(config.device)  # 解码的第一个输入，起始符号
6     for i in range(max_len - 1):
7         memory = memory.to(config.device)
8         out = model.decoder(ys, memory)  # [tgt_len,tgt_vocab_size]
9         out = out.transpose(0, 1)  # [tgt_vocab_size, tgt_len]
```

```
10          prob = model.classification(out[:, -1])  # 只对对预测的下一个词进行分类
11          _, next_word = torch.max(prob, dim=1)  # 选择概率最大者
12          next_word = next_word.item()
13          ys = torch.cat([ys, torch.ones(1, 1).type_as(src.data).
                                                    fill_(next_word)], dim=0)
14          # 将当前时刻解码的预测输出结果，同之前所有的结果堆叠作为输入再去预测下一个词。
15          if next_word == data_loader.EOS_IDX:
            # 如果当前时刻的预测输出为结束标志，则跳出循环结束预测。
16              break
17      return ys
```

在上述代码中，第 3 行是将源序列输入到 Transformer 的编码器中进行编码并得到 Memory；第 4-5 行是初始化解码阶段输入的第 1 个时刻的，在这里也就是 <sos>；第 6-17 行则是整个循环解码过程，在下一个时刻为 EOS_IDX 或者达到最大长度后停止；第 8 行是根据 memory 以及当前时刻的输入对当前时刻的输出进行解码；第 10-12 行则是分类得到当前时刻的解码输出结果；第 13 则是将当前时刻的解码输出结果头当前时刻之前所有的输入进行拼接，以此再对下一个时刻的输出进行预测。

最后，我们只需要调用如下函数便可以完成对原始输入上联的下联生成任务：

```
1  def do_couplet(srcs, config):
2      data_loader = LoadCoupletDataset(config.train_corpus_file_paths,
3                                       batch_size=config.batch_size,
4                                       tokenizer=my_tokenizer,
5                                       min_freq=config.min_freq)
6      couplet_model = CoupletModel(vocab_size=len(data_loader.vocab),
7                                   d_model=config.d_model,
8                                   nhead=config.num_head,
9                                   num_encoder_layers=config.num_encoder_layers,
10                                  num_decoder_layers=config.num_decoder_layers,
11                                  dim_feedforward=config.dim_feedforward,
12                                  dropout=config.dropout)
13     couplet_model = couplet_model.to(config.device)
14     loaded_paras = torch.load(config.model_save_dir + '/model.pkl')
15     couplet_model.load_state_dict(loaded_paras)
16     results = []
17     for src in srcs:
18         r = couplet(couplet_model, src, data_loader, config)
19         results.append(r)
20     return results
21
22
23 if __name__ == '__main__':
24     srcs = ["晚风摇树树还挺",
```

```
25              "忽忽几晨昏，离别间之，疾病间之，不及终年同静好",
26              "风声、雨声、读书声，声声入耳",
27              "上海自来水来自海上"]
28      tgts = ["晨露润花花更红",
29              "茕茕小儿女，孱羸若此，娇憨若此，更烦二老费精神",
30              "家事、国事、天下事，事事关心",
31              ""]
32      config = Config()
33      srcs = [" ".join(src) for src in srcs]
34      results = do_couplet(srcs, config)
35      for src, tgt, r in zip(srcs, tgts, results):
36          print(f"上联：{''.join(src.split())}")
37          print(f" AI：{r}")
38          print(f"下联：{tgts}")
39          print("=======")
```

在上述代码中，第 6-15 行是定义网络结构，以及恢复本地保存的网络权重；第 16-20 行则是开始执行多个上联的下联生成任务；第 34-39 行为生成示例，其输出结果为：

```
1 上联：晚风摇树树还挺
2  AI：朝露沾花花更红
3 下联：晨露润花花更红
4
5 上联：忽忽几晨昏，离别间之，疾病间之，不及终年同静好
6  AI：茕茕小儿女，孱羸若此，娇憨若此，更烦二老费精神
7 下联：茕茕小儿女，孱羸若此，娇憨若此，更烦二老费精神
8
9 上联：风声、雨声、读书声，声声入耳
10  AI：山色、水色、烟霞色，色色宜人
11 下联：家事、国事、天下事，事事关心
12
13 上联：上海自来水来自海上
14  AI：中山落叶松叶落山中
15
16 上联：月来客栈，迎五湖好友，谈百态人生。
17  AI：花发老舍，请四海嘉宾，喝一杯清茶。
```

以上完整代码可参见[14]。

总结

在本篇文章中，掌柜首先详细地介绍了 Transformer 论文的动机以及自注意力机制的原理与多头注意力机制的作用；然后介绍了 Transformer 中的位置编码以及整个预测和训练过程中模型的编码解码过程；接着进一步介绍了 Transformer 的网络结构、自注意力机制的原理实现以及 Transformer 网络的实现；最后，掌柜还通过 3 个实例（包括论文中的翻译模型、文本分类模型，以及对联生成模型）从代码的角度来介绍了整个 Transformer 网络的原理与使用示例。

在下一篇文章《This Post Is All You Need（下卷）——步步走进 BERT》中，掌柜将会逐一详细介绍 BERT 模型的原理细节以及多个下游任务的微调场景等。

本次内容就到此结束，感谢您的阅读！如果你觉得上述内容对你有所帮助，欢迎分享至一位你的朋友！若有任何疑问与建议，请添留言进行交流。**青山不改，绿水长流，我们月来客栈见！**

扫码关注月来客栈可获得更多优质内容！

引用

[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need
[2] The Illustrated Transformer http://jalammar.github.io/illustrated-transformer/
[3] https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0
[4] https://pytorch.org/docs/stable/generated/torch.bmm.html?highlight=bmm#torch.bmm
[5] LANGUAGE TRANSLATION WITH TRANSFORMER https://pytorch.org/tutorials/beginner/translation_transformer.html
[6] The Annotated Transformer http://nlp.seas.harvard.edu/2018/04/03/attention.html
[7] SEQUENCE-TO-SEQUENCE MODELING WITH NN.TRANSFORMER AND TORCHTEXT https://pytorch.org/tutorials/beginner/transformer_tutorial.html
[8] Transformer model for language understanding https://tensorflow.google.cn/text/tutorials/transformer?hl=en#multi-head_attention
[9] https://github.com/multi30k/dataset
[10]你还在手动构造词表？试试 torchtext.vocab
[11]https://github.com/moon-hotel/TransformerTranslation
[12]https://github.com/moon-hotel/TransformerClassification
[13]https://pytorch.org/tutorials/beginner/text_sentiment_ngrams_tutorial.html
[14]https://github.com/moon-hotel/TransformerCouplet