下面是逐位置前馈网络的实现代码。

```
# 定义逐位置前馈网络类
class PoswiseFeedForwardNet(nn.Module):
    def __init__(self, d_ff=2048):
        super(PoswiseFeedForwardNet, self).__init__()
        # 定义一维卷积层 1，用于将输入映射到更高维度
        self.conv1 = nn.Conv1d(in_channels=d_embedding, out_channels=d_ff, kernel_size=1)
        # 定义一维卷积层 2，用于将输入映射回原始维度
        self.conv2 = nn.Conv1d(in_channels=d_ff, out_channels=d_embedding, kernel_size=1)
        # 定义层归一化
        self.layer_norm = nn.LayerNorm(d_embedding)
    def forward(self, inputs):
        #---------------------------- 维度信息 ----------------------------------
        # inputs [batch_size, len_q, embedding_dim]
        #------------------------------------------------------------------------
        residual = inputs # 保留残差连接
        # 在卷积层 1 后使用 ReLU 函数
        output = nn.ReLU()(self.conv1(inputs.transpose(1, 2)))
        #---------------------------- 维度信息 ----------------------------------
        # output [batch_size, d_ff, len_q]
        #------------------------------------------------------------------------
        # 使用卷积层 2 进行降维
        output = self.conv2(output).transpose(1, 2)
        #---------------------------- 维度信息 ----------------------------------
        # output [batch_size, len_q, embedding_dim]
        #------------------------------------------------------------------------
        # 与输入进行残差连接，并进行层归一化
        output = self.layer_norm(output + residual)
        #---------------------------- 维度信息 ----------------------------------
        # output [batch_size, len_q, embedding_dim]
        #------------------------------------------------------------------------
        return output # 返回加入残差连接后层归一化的结果
```

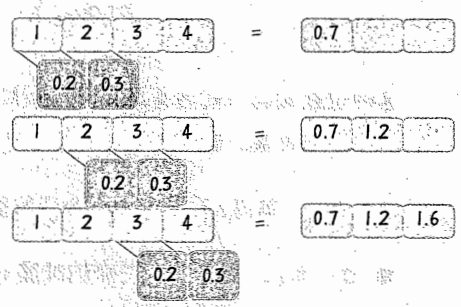
PoswiseFeedForwardNet 类实现了逐位置前馈网络，用于处理 Transformer 中自注意力机制的输出。其中包含两个一维卷积层，它们一个负责将输入映射到更高维度，一个再把它映射回原始维度。在两个卷积层之间，使用了 ReLU 函数。

小冰：这里的 nn.Conv1d，也就是 PyTorch 的一维卷积层？为什么这个层可以实现逐位置前馈机制呢？

咖哥：在这里，用一维卷积层代替了论文中的全连接层（线性层）来实现前馈神经网络。其原因是全连接层不共享权重，而一维卷积层在各个位置上共享权重，所以能够减少网络参数的数量。一维卷积层的工作原理是将卷积核（也称为过滤器或特征

映射）沿输入序列的一个维度滑动（如下图所示），并在每个位置进行点积操作。在这种情况下，我们使用大小为 1 的卷积核。这样，卷积操作实际上只会在输入序列的一个位置进行计算，因此它能够独立地处理输入序列中的每个位置。

如右图所示，在 PoswiseFeedForwardNet 类中，首先通过使用 conv1 的多个卷积核将输入序列映射到更高的维度（程序中是 2048 维，这是一个可调节的超参数），并应用 ReLU 函数。接着，conv2 将映射后的序列降维到原始维度。这个过程在输入序列的每个位置上都是独立完成的，因为一维卷积层会在每个位置进行逐点操作。所以，逐位置前馈神经网络能够在每个位置上分别应用相同的运算，从而捕捉输入序列中各个位置的信息。



卷积核沿输入序列的一个维度滑动

小冰：在 Transformer 模型中，逐位置前馈神经网络的作用是什么呢？

咖哥：有下面几个作用。

（1）增强模型的表达能力。FFN 为模型提供了更强大的表达能力，使其能够捕捉输入序列中更复杂的模式。通过逐位置前馈神经网络和自注意力机制的组合，Transformer 可以学习到不同位置之间的长距离依赖关系。

（2）信息融合。FFN 可以将自注意力机制输出的信息进行融合。每个位置上的信息在经过 FFN 后，都会得到一个新表示。这个新表示可以看作原始信息在经过一定程度的非线性变换之后的结果。

（3）层间传递。在 Transformer 中，逐位置前馈神经网络将在每个编码器和解码器层中使用。这样可以确保每一层的输出都经过了 FFN 的处理，从而在多层次上捕捉到序列中的特征。

多头自注意力层和逐位置前馈神经网络层是编码器层结构中的两个主要组件，不过，在开始构建编码器层之前，还要再定义两个辅助性的组件。第一个是位置编码表，第二个是生成填充注意力掩码的函数。

组件3　正弦位置编码表

我们已经讲过，Transformer 模型的并行结构导致它不是按位置顺序来处理序列的，但是在处理序列尤其是注意力计算的过程中，仍需要位置信息来帮助捕捉序列中的

顺序关系。为了解决这个问题，需要向输入序列中添加位置编码。

Tansformer 的原始论文中使用的是正弦位置编码。它的计算公式如下：

$$PE(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right)$$

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

这种位置编码方式具有周期性和连续性的特点，可以让模型学会捕捉位置之间的相对关系和全局关系。这个公式可以用于计算位置嵌入向量中每个维度的角度值。

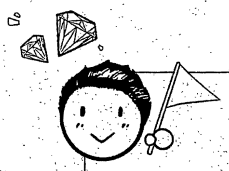
- pos：单词 / 标记在句子中的位置，从 0 到 seq_len−1。
- d：单词 / 标记嵌入向量的维度 embedding_dim。
- i：嵌入向量中的每个维度，从 0 到 $\frac{d}{2}-1$。

公式中 d 是固定的，但 pos 和 i 是变化的。如果 d=1024，则 $i \in [0, 512]$，因为 $2i$ 和 $2i$+1 分别代表嵌入向量的偶数和奇数位置。

下面定义正弦位置编码表的函数 get_sin_enc_table，用于在 Transformer 中引入位置信息。

```
# 生成正弦位置编码表的函数，用于在 Transformer 中引入位置信息
def get_sin_enc_table(n_position, embedding_dim):
    #---------- 维度信息 ----------
    # n_position: 输入序列的最大长度
    # embedding_dim: 词嵌入向量的维度
    #------------------------------------------------------------
    # 根据位置和维度信息，初始化正弦位置编码表
    sinusoid_table = np.zeros((n_position, embedding_dim))
    # 遍历所有位置和维度，计算角度值
    for pos_i in range(n_position):
        for hid_j in range(embedding_dim):
            angle = pos_i / np.power(10000, 2 * (hid_j // 2) / embedding_dim)
            sinusoid_table[pos_i, hid_j] = angle
    # 计算正弦和余弦值
    sinusoid_table[:, 0::2] = np.sin(sinusoid_table[:, 0::2])  # dim 2i 偶数维
    sinusoid_table[:, 1::2] = np.cos(sinusoid_table[:, 1::2])  # dim 2i+1 奇数维
    #---------- 维度信息 ----------
    # sinusoid_table 的维度是 [n_position, embedding_dim]
    #------------------------------------------------------------
    return torch.FloatTensor(sinusoid_table)  # 返回正弦位置编码表
```

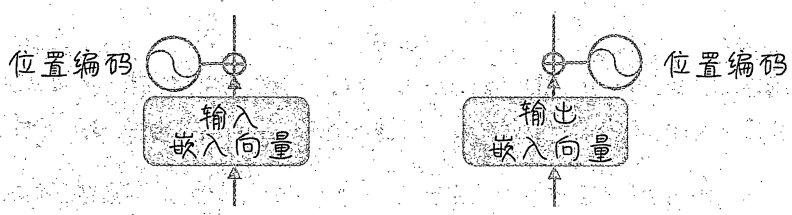
所以，虽然使用自然数序列（1、2、3、4 等）作为位置编码可以做一定的区分，但正弦和余弦函数生成的位置嵌入向量在捕捉序列中更复杂的位置关系方面更具优势。



咖哥发言

在 Transformer 模型中，使用正弦和余弦函数生成的位置嵌入向量已被证明是一种有效的方法，这是一种固定的位置编码方法，可以在不同的 NLP 任务中实现良好的性能。然而，也有其他位置编码方法，如经过学习动态生成的位置嵌入（Learned Position Embeddings），其中位置嵌入向量在训练过程中进行优化。选择何种位置编码方法取决于具体的任务和数据集。

在后续编码器（及解码器）组件中，我们将调用这个函数生成位置嵌入向量，为编码器和解码器输入序列中的每个位置添加一个位置编码，如下图所示。



将位置编码和输入向量整合

下面继续介绍用于生成填充注意力掩码的函数。

组件4　填充掩码

在 NLP 任务中，输入序列的长度通常是不固定的。为了能够同时处理多个序列，我们需要将这些序列填充到相同的长度，将不等长的序列补充到等长，这样才能将它们整合成同一个批次进行训练。通常使用一个特殊的标记（如 <pad>，编码后 <pad> 这个 token 的值通常是 0）来表示填充部分。

然而，这些填充符号并没有实际的含义，所以我们希望模型在计算注意力时忽略它们。因此，在编码器的输入部分，我们使用了填充位的注意力掩码机制（如下页图所示）。这个掩码机制的作用是在注意力计算的时候把无用的信息屏蔽，防止模型在计算注意力权重时关注到填充位。

```
# 定义填充注意力掩码函数
def get_attn_pad_mask(seq_q, seq_k):
    #---------------------------------- 维度信息 ----------------------------------
    # seq_q 的维度是 [batch_size, len_q]
    # seq_k 的维度是 [batch_size, len_k]
    #------------------------------------------------------------------------------
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
    # 生成布尔类型张量
    pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)  # <PAD>token 的编码值为 0
    #---------------------------------- 维度信息 ----------------------------------
    # pad_attn_mask 的维度是 [batch_size,1,len_k]
    #------------------------------------------------------------------------------
    # 变形为与注意力分数相同形状的张量
    pad_attn_mask = pad_attn_mask.expand(batch_size, len_q, len_k)
    #---------------------------------- 维度信息 ----------------------------------
    # pad_attn_mask 的维度是 [batch_size,len_q,len_k]
    #------------------------------------------------------------------------------
    return pad_attn_mask # 返回填充位置的注意力掩码
```

函数 get_attn_pad_mask(seq_q, seq_k) 中 seq_q 表示 Query 序列，seq_k 表示 Key 序列。seq_q 和 seq_k 的维度信息中，batch_size 表示批量大小，len_q 和 len_k 分别表示 Query 和 Key 序列的长度。

之后，使用 seq_k.data.eq(0) 创建一个布尔矩阵，其中值为 True 的位置对应着 seq_k 中的填充（<pad>）标记。假设我们使用 0 作为填充标记的词汇表索引值，那么这个操作将检测 seq_k 中的 0 值。然后，使用 unsqueeze(1) 为布尔矩阵增加一个维度，将其变为 batch_size × 1 × len_k 的形状。再通过 expand(batch_size, len_q, len_k) 将布尔矩阵扩展为 batch_size × len_q × len_k 的形状。这个扩展操作仅复制已有的数据，不会引入新的信息。

在多头自注意力计算中计算注意力权重时，会将这个函数生成的填充注意力掩码与原始权重相加，使得填充部分的权重变得非常小（接近负无穷），从而在使用 softmax 函数归一化后接近于 0，实现忽略填充部分的效果。

组件5　编码器层

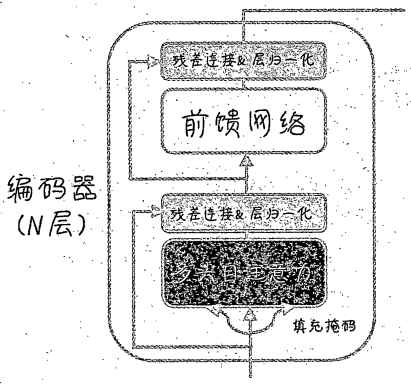
有了多头自注意力和逐位置前馈网络这两个主要组件，以及正弦位置编码表和填充掩码这两个辅助函数后，现在我们终于可以搭建编码器层这个核心组件了。

（2）将 enc_outputs 输入逐位置前馈网络层 pos_ffn，并更新 enc_outputs。

（3）最后返回 enc_outputs 和 attn_weights。enc_outputs 表示编码器层的输出，attn_weights 表示自注意力权重矩阵，可以用于分析和可视化。

在多头自注意力层 MultiHeadAttention 的输出中，enc_outputs 的维度是 [batch_size, seq_len, embedding_dim]。原因是在多头自注意力层 MultiHeadAttention 内部，首先将输入的 enc_inputs 映射为 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$，这些映射后的张量的维度为 [batch_size, n_heads, seq_len, d_k]。然后，通过对这些张量进行自注意力计算，得到的注意力输出的维度也为 [batch_size, n_heads, seq_len, d_k]。最后，我们需要将这些头合并回原来的维度，这通过将最后两个维度进行拼接实现，也就是 n_heads * d_k 等于 embedding_dim，所以最终的 enc_outputs 的维度就是 [batch_size, seq_len, embedding_dim]。

而对于 attn_weights，在多头自注意力层 MultiHeadAttention 内部，首先将输入的 enc_inputs 映射为 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$，这些映射后的张量的维度为 [batch_size, n_heads, seq_len, d_k]。然后，通过计算 $\boldsymbol{Q}$ 和 $\boldsymbol{K}$ 的点积得到注意力分数，通过 softmax 进行归一化，得到的注意力权重的维度是 [batch_size, n_heads, seq_len, seq_len]。这个维度的含义是，对于每个批次中的每个头，每个输入序列中的每个元素，都有一个长度为 seq_len 的权重向量，对应该元素与输入序列中的其他元素之间的关系强度。注意，在 MultiHeadAttention 计算结束后，我们并不会像处理 enc_outputs 一样合并头的结果，所以 attn_weights 的维度会保持为 [batch_size, n_heads, seq_len, seq_len]。



编码器结构

如左图所示，这个编码器层类实现了 Transformer 编码器中的一层计算，包括多头自注意力和逐位置前馈网络两个子层。在实际的 Transformer 编码器中，通常会堆叠多个这样的层来构建一个深度模型，以捕捉更丰富的序列特征。

组件6　编码器

编码器是多个编码器层的堆叠，这就是我们这一课名称的奥秘所在：层峦叠翠上青天——层层相叠，功力倍增。

下面是编码器的代码实现。

入向量的维度。输入的编码应该通过 nn.Embedding 进行词向量的表示学习，用以捕捉上下文关系。这个我们已经比较了解了，因此这个组件无须过多说明。

（2）位置嵌入层实例 pos_emb。使用 nn.Embedding.from_pretrained() 方法从预先计算的正弦位置编码表（由 get_sinusoid_encoding_table() 函数生成）创建位置嵌入层，并通过 freeze=True 参数保持其权重不变。

（3）编码器层数 self.layers。使用 nn.ModuleList() 创建一个模块列表，包含 n_layers 个 EncoderLayer 实例。这些层将顺序堆叠在编码器中。

Encoder 类的 forward 方法中接收一个参数 enc_inputs，表示输入的序列，其形状为 [batch_size, source_len]。

forward 方法内部流程如下。

（1）将 enc_inputs 输入词嵌入层 src_emb 和位置嵌入层 pos_emb 中，然后将得到的词嵌入向量和位置嵌入向量相加，得到 enc_outputs。

（2）调用 get_attn_pad_mask() 函数，为输入序列生成自注意力掩码（如填充掩码），命名为 enc_self_attn_mask。在多头自注意力计算中，这个掩码可以让模型忽略填充部分。

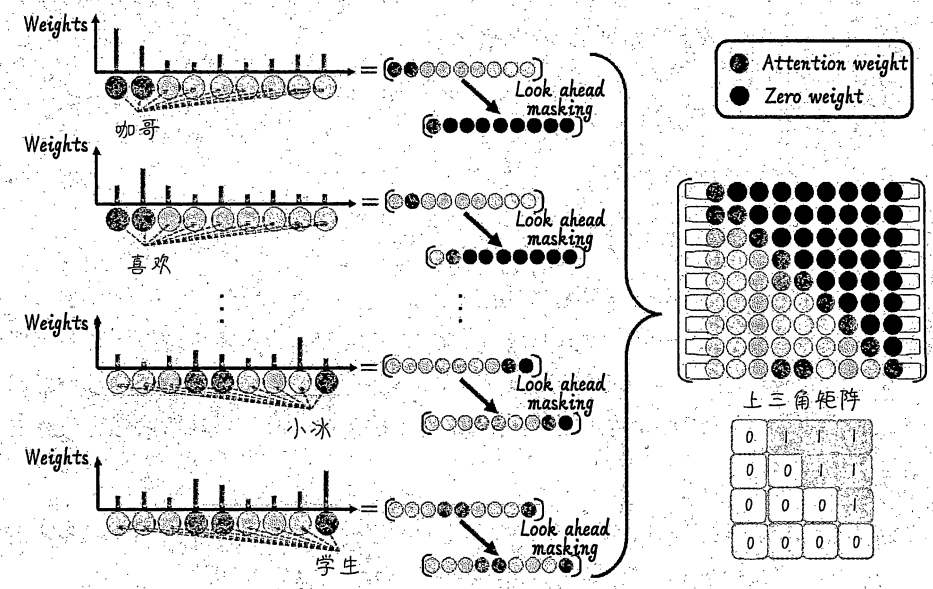
（3）定义一个空列表 enc_self_attn_weights，用于收集每个编码器层的自注意力权重矩阵。

（4）遍历编码器层数 self.layers 中的每个 EncoderLayer 实例。将 enc_outputs 和 enc_self_attn_mask 输入编码器层，更新 enc_outputs 并将得到的自注意力权重矩阵 enc_self_attn_weight 添加到列表 enc_self_attn_weights 中。

（5）最后返回 enc_outputs 和 enc_self_attn_weights。enc_outputs 表示编码器的输出，enc_self_attn_weights 表示每个编码器层的自注意力权重矩阵，可以用于分析和可视化。

这个编码器类实现了 Transformer 模型中的编码器部分，包括词嵌入、位置嵌入和多个编码器层。通过这个编码器，可以处理输入序列，并从中提取深层次的特征表示。这些特征表示可以直接应用于后续的任务，如序列到序列的生成任务（如机器翻译）或者分类任务（如情感分析）等。

哥”和“喜欢”这个上下文，来猜测咖哥喜欢谁。当对最后一个词“小冰”计算注意力的时候，前两个词就不是秘密了。



后续掩码示意图

为了实现上面的目标，需要构建一个上三角矩阵，也就是一个注意力掩码矩阵。其中对角线及以下的元素为 0，对角线以上的元素为 1。在计算多头自注意力时，我们将该矩阵与后续注意力掩码相加，使得未来信息对应的权重变得非常小（接近负无穷）。然后，通过应用 softmax 函数，未来信息对应的权重将接近于 0，从而实现忽略未来信息的目的。

下面定义一个后续注意力掩码函数 get_attn_subsequent_mask，它只有一个参数，用于接收解码器的输入序列形状信息，以生成掩码矩阵。

```
# 生成后续注意力掩码的函数，用于在多头自注意力计算中忽略未来信息
def get_attn_subsequent_mask(seq):
    #------------------------------ 维度信息 ------------------------------
    # seq 的维度是 [batch_size, seq_len(Q)=seq_len(K)]
    #----------------------------------------------------------------------
    # 获取输入序列的形状
    attn_shape = [seq.size(0), seq.size(1), seq.size(1)]
    #------------------------------ 维度信息 ------------------------------
    # attn_shape 是一个一维张量 [batch_size, seq_len(Q), seq_len(K)]
    #----------------------------------------------------------------------
```

```
# 定义解码器层类
class DecoderLayer(nn.Module):
    def __init__(self):
        super(DecoderLayer, self).__init__()
        self.dec_self_attn = MultiHeadAttention() # 多头自注意力层
        self.dec_enc_attn = MultiHeadAttention() # 多头自注意力层，连接编码器和解码器
        self.pos_ffn = PoswiseFeedForwardNet() # 逐位置前馈网络层

    def forward(self, dec_inputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_mask):
        #---------------------------------- 维度信息 ----------------------------------
        # dec_inputs 的维度是 [batch_size, target_len, embedding_dim]
        # enc_outputs 的维度是 [batch_size, source_len, embedding_dim]
        # dec_self_attn_mask 的维度是 [batch_size, target_len, target_len]
        # dec_enc_attn_mask 的维度是 [batch_size, target_len, source_len]
        #------------------------------------------------------------------------------
        # 将相同的 Q，K，V 输入多头自注意力层
        dec_outputs, dec_self_attn = self.dec_self_attn(dec_inputs, dec_inputs,
                                                        dec_inputs, dec_self_attn_mask)
        #---------------------------------- 维度信息 ----------------------------------
        # dec_outputs 的维度是 [batch_size, target_len, embedding_dim]
        # dec_self_attn 的维度是 [batch_size, n_heads, target_len, target_len]
        #------------------------------------------------------------------------------
        # 将解码器输出和编码器输出输入多头自注意力层
        dec_outputs, dec_enc_attn = self.dec_enc_attn(dec_outputs, enc_outputs,
                                                      enc_outputs, dec_enc_attn_mask)
        #---------------------------------- 维度信息 ----------------------------------
        # dec_outputs 的维度是 [batch_size, target_len, embedding_dim]
        # dec_enc_attn 的维度是 [batch_size, n_heads, target_len, source_len]
        #------------------------------------------------------------------------------
        # 输入逐位置前馈网络层
        dec_outputs = self.pos_ffn(dec_outputs)
        #---------------------------------- 维度信息 ----------------------------------
        # dec_outputs 的维度是 [batch_size, target_len, embedding_dim]
        # dec_self_attn 的维度是 [batch_size, n_heads, target_len, target_len]
        # dec_enc_attn 的维度是 [batch_size, n_heads, target_len, source_len]
        #------------------------------------------------------------------------------
        # 返回解码器层输出，每层的自注意力和解码器 - 编码器注意力权重
        return dec_outputs, dec_self_attn, dec_enc_attn
```

在 DecoderLayer 类的 __init__ 方法中：

（1）定义了多头自注意力层实例 dec_self_attn。这个层用于处理解码器的输入序列。

（2）定义了另一个多头自注意力层实例 dec_enc_attn。这个层用于建立解码器和

编码器之间的联系，将编码器的输出信息融合到解码器的输出中。

（3）定义了逐位置前馈网络层实例 pos_ffn。这个层用于处理多头自注意力层的输出，进一步提取特征。

forward 方法接收 4 个参数：dec_inputs 表示解码器的输入，enc_outputs 表示编码器的输出，dec_self_attn_mask 表示解码器自注意力掩码，dec_enc_attn_mask 表示编码器－解码器注意力掩码。在 forward 方法内部：

（1）将 dec_inputs 作为 Q、K、V 输入多头自注意力层 dec_self_attn 中，并传入 dec_self_attn_mask。得到输出 dec_outputs 和自注意力权重矩阵 dec_self_attn。

（2）将 dec_outputs 作为 Q，enc_outputs 作为 K、V 输入多头自注意力层 dec_enc_attn 中，并传入 dec_enc_attn_mask。得到更新后的输出 dec_outputs 和编码器－解码器注意力权重矩阵 dec_enc_attn。

（3）将 dec_outputs 输入逐位置前馈网络层 pos_ffn 中，得到最终的 dec_outputs。

（4）返回 dec_outputs、dec_self_attn 和 dec_enc_attn。dec_outputs 表示解码器层的输出，dec_self_attn 表示解码器自注意力权重矩阵，dec_enc_attn 表示编码器－解码器注意力权重矩阵。

这个解码器层类实现了 Transformer 模型中的解码器层部分，包括多头自注意力、编码器－解码器多头自注意力和逐位置前馈网络等子层。通过堆叠多个解码器层，模型可以生成目标序列，并利用编码器的输出信息进行更准确的预测。

小冰：咖哥，我注意到 Transformer 的解码器有两层注意力机制，包括一个自注意力机制和一个编码器－解码器注意力机制，它们都是多头的吗？它们是否都有填充掩码？它们是否都有后续掩码？

咖哥：Transformer 中的解码器确实具有两层注意力机制。对于你的问题，我的回答如下。

（1）是多头的吗？

是的，自注意力和编码器－解码器注意力机制都采用多头自注意力策略。多头自注意力能让模型在多个子空间中同时学习不同的表示，从而提高表现。

（2）是否都是填充掩码？

是的，填充掩码用于忽略输入序列中的填充部分，防止注意力机制关注这些无意义

```
        dec_self_attns.append(dec_self_attn)
        dec_enc_attns.append(dec_enc_attn)
    #---------------------------- 维度信息 ----------------------------
    # dec_outputs 的维度是 [batch_size, target_len, embedding_dim]
    # dec_self_attns 是一个列表，每个元素的维度是 [batch_size, n_heads, target_len, target_len]
    # dec_enc_attns 是一个列表，每个元素的维度是 [batch_size, n_heads, target_len, source_len]
    #------------------------------------------------------------------
    # 返回解码器输出，解码器自注意力和解码器 - 编码器注意力权重
    return dec_outputs, dec_self_attns, dec_enc_attns
```

Decoder 类负责生成目标序列，在 __init__ 方法中初始化的内容如下。

- 词嵌入层实例 tgt_emb。这个层将目标序列中的单词转换为对应的向量表示。
- 位置嵌入层实例 pos_emb。这个层通过预先计算的正弦位置编码表来引入位置信息。
- 一个 nn.ModuleList 实例，用于存储多个解码器层。这里使用列表解析式创建了 n_layers 个解码器层。

Decoder 类的 forward 方法中接收 3 个参数：dec_inputs 表示解码器的输入，enc_inputs 表示编码器的输入，enc_outputs 表示编码器的输出。

forward 方法内部流程如下。

- 对解码器输入进行词嵌入和位置嵌入相加，得到 dec_outputs。
- 生成解码器自注意力掩码 dec_self_attn_mask 和解码器 - 编码器注意力掩码 dec_enc_attn_mask。
  解码器自注意力掩码 dec_self_attn_mask 是后续注意力掩码 dec_self_attn_subsequent_mask 与填充注意力掩码 dec_self_attn_pad_mask 的结合，通过将两个掩码矩阵相加并使用 torch.gt 函数生成一个布尔类型矩阵。gt 表示 “greater than”（大于），用于逐元素地比较两个张量，并返回一个与输入形状相同的布尔张量，看对应位置的输入元素是否大于给定的阈值 0。这个布尔矩阵将用于遮挡填充位和未来信息。
  解码器 - 编码器注意力掩码 dec_enc_attn_mask 则只包括填充注意力掩码 dec_self_attn_pad_mask，仅仅需要遮挡编码器传递进来的上下文向量中的填充位。
- 初始化两个空列表 dec_self_attns 和 dec_enc_attns，用于存储每个解码器

层的自注意力权重矩阵和编码器 - 解码器注意力权重矩阵。

- 使用一个 for 循环遍历所有的解码器层，将 dec_outputs、enc_outputs、dec_self_attn_mask 和 dec_enc_attn_mask 输入解码器层中。得到更新后的 dec_outputs，以及当前解码器层的自注意力权重矩阵 dec_self_attn 和编码器 - 解码器注意力权重矩阵 dec_enc_attn。将这两个权重矩阵分别添加到列表 dec_self_attns 和 dec_enc_attns 中。
- 返回 dec_outputs、dec_self_attns 和 dec_enc_attns。dec_outputs 表示解码器的输出，dec_self_attns 表示解码器各层的自注意力权重矩阵，dec_enc_attns 表示解码器各层的编码器 - 解码器注意力权重矩阵。

这个解码器类实现了 Transformer 模型中的解码器部分，包括词嵌入、位置嵌入和多个解码器层。通过堆叠多个解码器层，可以捕获目标序列中的复杂语义和结构信息。解码器的输出将被用来预测目标序列的下一个词。

现在，用于构建 Transformer 类的全部组件已经完成。我们可以用这些组件开始搭建一个 Transformer。

组件10　Transformer类

在 Transformer 模型的训练和推理过程中，解码器与编码器一起工作。编码器负责处理源序列并提取其语义信息，解码器则根据编码器的输出和自身的输入（目标序列）生成新的目标序列。在这个过程中，解码器会利用自注意力机制关注目标序列的不同部分，同时通过编码器 - 解码器注意力机制关注编码器输出的不同部分。当解码器处理完所有的解码器层后，最终输出的 dec_outputs 将被送入一个线性层和 softmax 层（softmax 层已经整合在损失函数中，不需要具体实现，所以下面的代码我们只定义线性层），生成最终的预测结果。这个预测结果是一个概率分布，表示每个词在目标序列下一个位置的概率。

下面构建 Transformer 模型的类。

```
# 定义 Transformer 模型
class Transformer(nn.Module):
    def __init__(self, corpus):
        super(Transformer, self).__init__()
        self.encoder = Encoder(corpus) # 初始化编码器实例
        self.decoder = Decoder(corpus) # 初始化解码器实例
        # 定义线性投影层，将解码器输出转换为目标词汇表大小的概率分布
        self.projection = nn.Linear(d_embedding, len(corpus.tgt_vocab), bias=False)
```

```
def forward(self, enc_inputs, dec_inputs):
    #---------------------------- 维度信息 ----------------------------------
    # enc_inputs 的维度是 [batch_size, source_seq_len]
    # dec_inputs 的维度是 [batch_size, target_seq_len]
    #------------------------------------------------------------------------
    # 将输入传递给编码器，并获取编码器输出和自注意力权重
    enc_outputs, enc_self_attns = self.encoder(enc_inputs)
    #---------------------------- 维度信息 ----------------------------------
    # enc_outputs 的维度是 [batch_size, source_len, embedding_dim]
    # enc_self_attns 是一个列表，每个元素的维度是 [batch_size, n_heads, src_seq_len, src_seq_len]
    #------------------------------------------------------------------------
    # 将编码器输出、解码器输入和编码器输入传递给解码器
    # 获取解码器输出、解码器自注意力权重和编码器 - 解码器注意力权重
    dec_outputs, dec_self_attns, dec_enc_attns = self.decoder(dec_inputs, enc_inputs, enc_outputs)
    #---------------------------- 维度信息 ----------------------------------
    # dec_outputs 的维度是 [batch_size, target_len, embedding_dim]
    # dec_self_attns 是一个列表，每个元素的维度是 [batch_size, n_heads, tgt_seq_len, src_seq_len]
    # dec_enc_attns 是一个列表，每个元素的维度是 [batch_size, n_heads, tgt_seq_len, src_seq_len]
    #------------------------------------------------------------------------
    # 将解码器输出传递给投影层，生成目标词汇表大小的概率分布
    dec_logits = self.projection(dec_outputs)
    #---------------------------- 维度信息 ----------------------------------
    # dec_logits 的维度是 [batch_size, tgt_seq_len, tgt_vocab_size]
    #------------------------------------------------------------------------
    # 返回预测值，编码器自注意力权重，解码器自注意力权重，解码器 - 编码器注意力权重
    return dec_logits, enc_self_attns, dec_self_attns, dec_enc_attns
```

因为有了前面的组件，这段代码的结构就清晰而简单了。首先初始化编码器、解码器和投影层。在 forward 方法中，将源序列输入传递给编码器，获取编码器输出和自注意力权重。然后将编码器输出、解码器输入和编码器输入传递给解码器，获取解码器输出、解码器自注意力权重和编码器 - 解码器注意力权重。最后，将解码器输出传递给投影层，生成目标词汇表大小的概率分布。这个概率分布将被用于计算损失和评估模型的性能。

下面我们马上使用这个 Transformer 模型来完成具体的任务。

6.3 完成翻译任务

这里，我们仍然使用 Seq2Seq 的小型翻译任务数据集。不过，我们这次把数据集整合到一个 TranslationCorpus 类，这个类会读入语料，自动整理语料库的字典，并提供批量数据。

```
        # 将源语言和目标语言的句子转换为索引序列
        src_seq = [self.src_vocab[word] for word in src_sentence.split()]
        tgt_seq = [self.tgt_vocab['<sos>']] + [self.tgt_vocab[word] \
                for word in tgt_sentence.split()] + [self.tgt_vocab['<eos>']]
        # 对源语言和目标语言的序列进行填充
        src_seq += [self.src_vocab['<pad>']] * (self.src_len - len(src_seq))
        tgt_seq += [self.tgt_vocab['<pad>']] * (self.tgt_len - len(tgt_seq))
        # 将处理好的序列添加到批次中
        input_batch.append(src_seq)
        output_batch.append([self.tgt_vocab['<sos>']] + ([self.tgt_vocab['<pad>']] * \
                        (self.tgt_len - 2)) if test_batch else tgt_seq[:-1])
        target_batch.append(tgt_seq[1:])
    # 将批次转换为 LongTensor 类型
    input_batch = torch.LongTensor(input_batch)
    output_batch = torch.LongTensor(output_batch)
    target_batch = torch.LongTensor(target_batch)
    return input_batch, output_batch, target_batch
```

TranslationCorpus 中的 __init__ 方法，接收一组句子对（源语言句子和目标语言句子）。它计算源语言和目标语言的最大句子长度，并为其分别添加 1 和 2 以容纳填充符 <pad> 和特殊符号（<sos> 和 <eos>）。然后，它创建源语言和目标语言的词汇表，并为每个单词创建索引到单词的映射。

create_vocabularies 方法用于创建源语言和目标语言的词汇表。它首先统计源语言和目标语言的单词频率，然后为每个单词分配一个唯一的索引。源语言词汇表包含填充符 <pad>，目标语言词汇表包含填充符 <pad>、句子开始符号 <sos> 和句子结束符号 <eos>。

make_batch(self, batch_size, test_batch=False): 该方法用于创建一个大小为 batch_size 的批次。批次包含输入批次、输出批次和目标批次。对于每个句子对，它将源语言和目标语言的句子转换为索引序列，并进行填充以匹配最大句子长度。输入批次包含源语言的序列，输出批次包含目标语言的序列（在测试阶段，输出批次仅包含句子开始符号 <sos>），目标批次包含目标语言的序列（去除句子开始符号 <sos>）。最后，将这些批次转换为 LongTensor 类型。

基于中译英翻译例子创建语料库的实例。

```
# 创建语料库类实例
corpus = TranslationCorpus(sentences)
```