人工智能基础课程报告

自然语言处理领域调研

——以 Transformer 的实现为例

专　　业　自动化

学　　生　方尧

学　　号　**190410102**

授课教师　熊小刚

日　　期　**2022 年 12 月 3 日**

自然语言处理领域调研

——以 Transformer 的实现为例

一、 摘要

自然语言处理，是指用计算机对自然语言的形、音、义等信息进行处理，具体应用有机器翻译、文本摘要、文本分类、文本校对、信息抽取、语音合成、语音识别等。本课题介绍了相关领域的一些经典模型，并以 transformer 为例，学习了 transformer 模型的结构、原理以及进行了简单实现。

关键词：人工智能、自然语言处理、Transformer、注意力机制

二、课题背景

自然语言是指汉语、英语、法语等人们日常使用的语言，是人类学习生活的要工具。自然语言处理，是指用计算机对自然语言的形、音、义等信息进行处理，即对字、词、句、篇章的输入、输出、识别、分析、理解、生成等的操作和加工，涵盖理解、转化、生成等方面。具体表现形式及具体应用有机器翻译、文本摘要、文本分类、文本校对、信息抽取、语音合成、语音识别等。自然语言处理的目的是要让计算机理解自然语言以实现人类可以与计算机直接用自然语言通信。其涵盖两部分：一是自然语言的理解、二是自然语言的生成。在人工智能领域一般用图灵试验判断计算机是否理解某种自然语言：一是机器能正确回答文本中有关问题、二是有能力输出给定文本摘要的能力、三是能用不同语句或者句型复述输入文本、四是能将给定文本从一种语言翻译到另外一种语言。

2003 年，Y.BENGIO 首次提出神经网络语言模型。2017 年之前，进行自然语言处理时人们常用多层感知机（MLP）、卷积神经网络（CNN）和循环神经网络（RNN），来构建神经网络语言模型。MLP 由于每层都使用全连接方式难以捕捉到局部信息。CNN 采用一个或多个卷积核对局部输入序列依次进行卷积处理，可以较好地进行局部特征提取。由于 CNN 适用于高并发场景，大规模的 CNN 模型经过训练可以提取更多的局部特征。然而，CNN 却不擅长捕获远距离特征。RNN 是将当前时刻网络隐含层的输入作为下一时刻的输入，每个时刻的输入经

过层次递归后均会对最终产生输出影响。RNN 可以较好地解决时序问题和序列到序列问题，但是由于依赖时序处理，这种输入方式使得 RNN 很难充分利用并行算力来加速训练。

2017 年，来自谷歌的几位工程师在不使用传统 CNN、RNN 等模型的情况下，完全采用基于自注意力机制的 Transformer 模型，取得了非常好的效果。在解决序列到序列问题的过程中，他们不仅考虑前一个时刻的影响，还考虑目标输出与输入句子中哪些词更相关，并对输入信息进行加权处理，从而突出重要特征对输出的影响。这种对强相关性的关注就是注意力机制。注意力机制最早由 Bengio 团队于 2014 年提出并在近年广泛地应用在深度学习中的各个领域。Transformer 模型是一个基于多头自注意力机制的基础模型，不依赖顺序建模就可以充分利用并行算力处理。在构建大模型时，Transformer 模型在训练速度和长距离建模方面都优于传统的神经网络模型。因此，近年来流行的 GPT、BERT 等若干超大规模预训练语言模型基本上都是基于 Transformer 模型构建的。Transformer 模型整体架构如图 1 所示。自注意力机制的本质是学习序列中的上下文相关程度和深层语义信息。然而，随着输入序列长度的增加，学习效率会降低。为了更好地处理长文本序列，Transformer 模型又衍生出一些变种比如 Transformer-XL，具备更强的长文本处理能力。

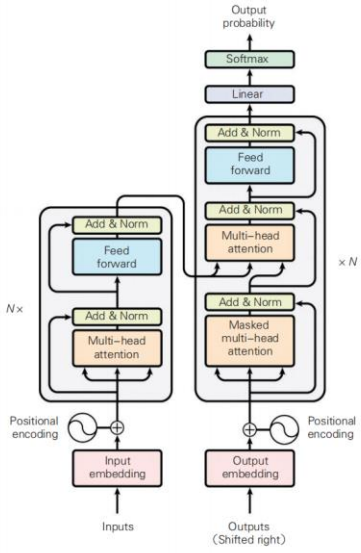

图 1Transformer 模型整体架构

三、主要应用场景&解决的问题

自然语言处理机器翻译主要应用在文字翻译、文档翻译、语音翻译、拍照扫描翻译、同传翻译等场景。

四、原理内容

本课程报告主要研究对象是自然语言处理机器翻译领域中由谷歌的几位工程师 2017 年提出的完全采用基于自注意力机制的 Transformer 模型，主要内容是学习了 transformer 模型的结构、原理以及进行了简单实现。

4.1 模型概览

Transformer 模型整体结构如图 1 所示，本质上是一个 Encoder-Decoder 架构。因此中间部分的 Transformer 可以分为两个部分：编码组件和解码组件。编码器组件由多层编码器（Encoder）组成，解码组件也是由相同层数的解码器（Decoder）组成，如图 2 所示。

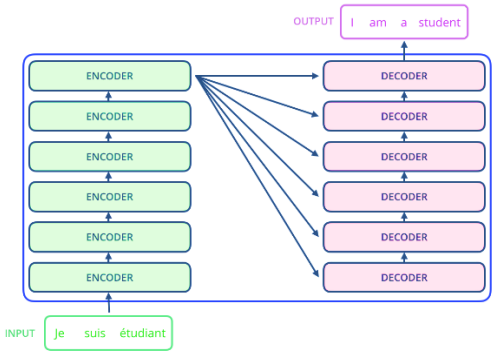

图 2Encoder-Decoder 架构

每个编码器由两个子层组成：Self-Attention 层（自注意力层）和 Position-wiseFeedForwardNetwork（前馈网络，缩写为 FFN），每个编码器的结构都是相同的，但是它们使用不同的权重参数，如图 3 所示。

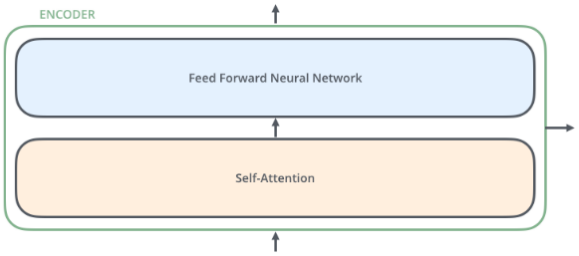

图 3 编码器

解码器中也有编码器这两层，但其多了注意力层，与 sequence2sequence 模型中注意力相似，用于关注输入句子的相关部分。

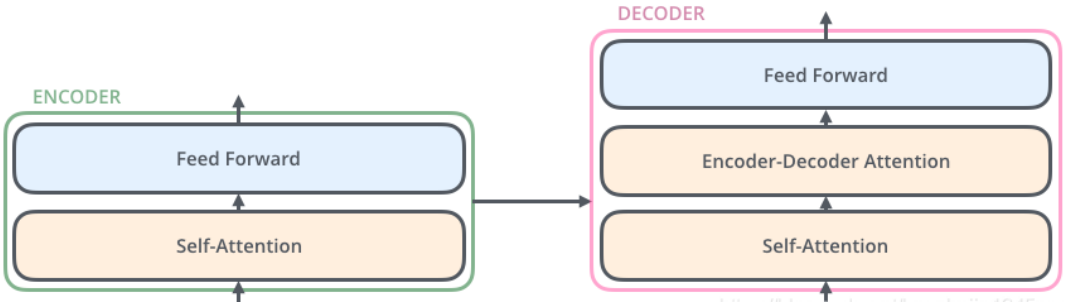

图 4 解码器

4.2 词嵌入 embedding、位置编码——张量引入

与通常的 NLP 任务一样，首先，我们使用词嵌入算法（Embedding）也就是以固定维数的特征维度将每个词转换为一个词向量，论文作者用的是 512。嵌入仅发生在最底层的编码器中，类似于初始化。编码器会接收一个向量，先传递到 self-attention 层再传到前馈网络层，最后将输出传递到下一编码器。

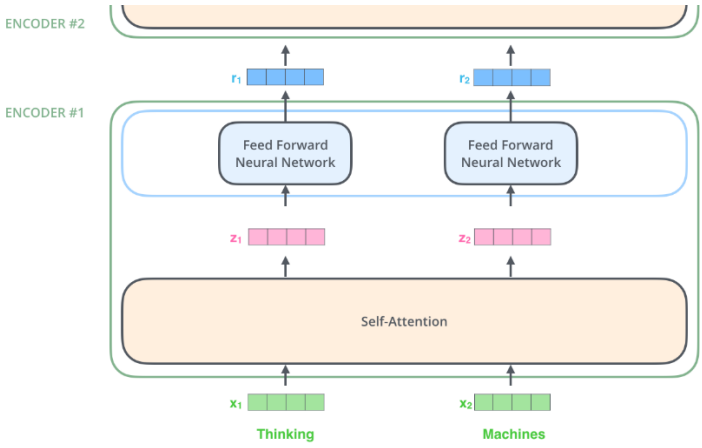

图 5 信号传递图

序列中词顺序用一个向量确定模型中每个词的位置，或者序列中不同词之间的距离，具体数学公式如下：

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

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

其中，pos 表示位置，i 表示维度。上面的函数使得模型可以学习到词之间的相对位置关系：任意位置 $PE_{(pos+k)}$ 都可以被 $PE_{(pos)}$ 线性函数表示：

$$\cos(\alpha+\beta)=\cos\alpha\cos\beta-\sin\alpha\sin\beta$$
$$\sin(\alpha+\beta)=\sin\alpha\cos\beta+\cos\alpha\sin\beta$$

4.3self-attention 自注意机制

其基本结构图如图 6，对于 SelfAttention 来讲，Q（Query），K（Key）和 V（Value）三个矩阵均来自同一输入，并按照以下步骤计算：

1. 首先计算 Q 和 K 之间的点积，为了防止其结果过大，会除以$\sqrt{d_k}$，其中$\sqrt{d_k}$为 key 向量的维度。
2. 然后利用 softmax 操作将其结果归一化为概率分布，再乘以矩阵 V 就得到权重求和的表示。整个计算过程可以表示为：

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

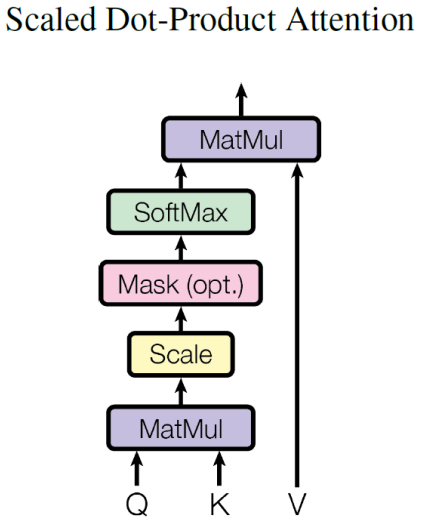

图 6 自注意机制

4.4 使用矩阵计算 Self-Attention

1. 计算 Query，Key 和 Value 矩阵。首先，将所有词向量放到一个矩阵 X，分别和三个训练过的矩阵（W^Q,W^K,W^V）相乘得到 Q,K,V 矩阵。
2. 计算自注意力，使用矩阵计算，压缩了计算步骤。

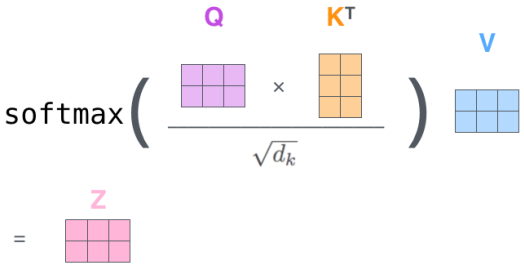

图 7 自注意机制计算

4.5 多头注意力机制

通过添加一种多头注意力机制，可以进一步完善了自注意力层。具体做法：首先，通过 h 个不同的线性变换对 Query、Key 和 Value 进行映射；然后，将不同的 Attention 拼接起来；最后，再进行一次线性变换。基本结构如图 8 所示：

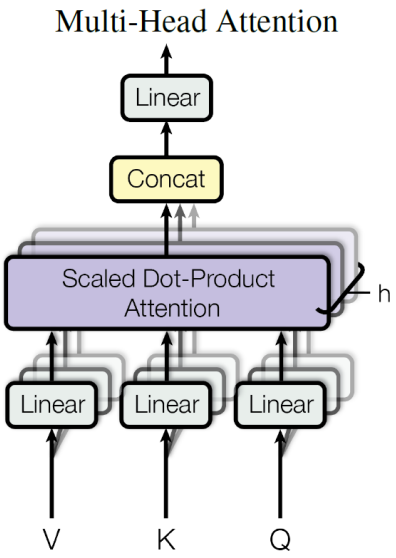

图 8 多头注意力机制

每一组注意力用于将输入映射到不同的子表示空间，这使得模型可以在不同子表示空间中关注不同的位置。

$$MultiHead(Q,K,V) = Concat(head_1,\cdots,head_h)W^O$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

我们需要为每组注意力单独维护不同的 Query，Key，Value 权重矩阵，得到不同的 Query，Key，Value 值。按照上面的方法，使用不同的权重矩阵进行 8 次自注意力计算，就可以得到 8 个不同的 Z 矩阵。

最后把八个矩阵拼接起来，乘上一权重矩阵W^O，得到最终的矩阵 Z。

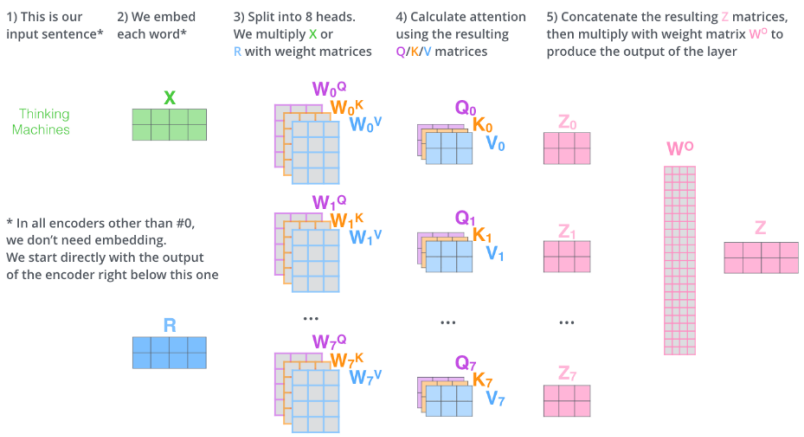

图 9

4.6 位置前馈网络（Position-wise Feed-Forward Networks）

其由两个全连接层组成，第一个全连接层的激活函数为 ReLU 激活函数，可以表示为：

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

值得注意的是，在每个编码器和解码器中，全连接前馈网络结构相同但不共享参数。

4.6 残差和归一化

自注意力层和前馈网络层由残差和归一化连接，计算过程可以表示为：

$$\text{sub\_layer\_output} = \text{LayerNorm}(x + \text{SubLayer}(x))$$

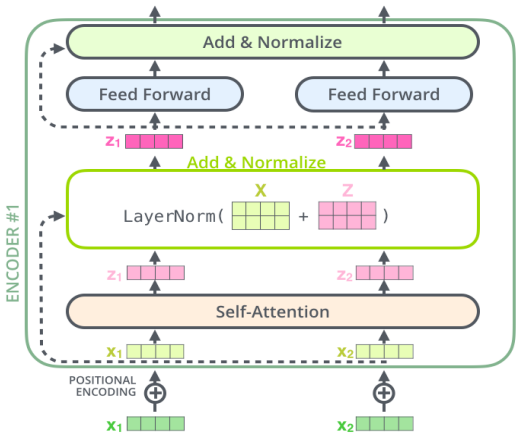

图 10

为了方便残差连接，编码器和解码器中的所有子层和嵌入层的输出维度需要保持一致为$d_{model} = 512$

五、结果展示

100 个 Epoch 后，loss 降低到了较低水平 0.00015，成功实现预期目标。

```
Epoch: 0069 loss = 0.002045
Epoch: 0070 loss = 0.002774
Epoch: 0071 loss = 0.001783
Epoch: 0072 loss = 0.001508
Epoch: 0073 loss = 0.001478
Epoch: 0074 loss = 0.001833
Epoch: 0075 loss = 0.001216
Epoch: 0076 loss = 0.001125
Epoch: 0077 loss = 0.002546
Epoch: 0078 loss = 0.000744
Epoch: 0079 loss = 0.001005
Epoch: 0080 loss = 0.000959
Epoch: 0081 loss = 0.000782
Epoch: 0082 loss = 0.000570
Epoch: 0083 loss = 0.000610
Epoch: 0084 loss = 0.000609
Epoch: 0085 loss = 0.000516
Epoch: 0086 loss = 0.000523
Epoch: 0087 loss = 0.000348
Epoch: 0088 loss = 0.000389
Epoch: 0089 loss = 0.000428
Epoch: 0090 loss = 0.000527
Epoch: 0091 loss = 0.000240
Epoch: 0092 loss = 0.000285
Epoch: 0093 loss = 0.000249
Epoch: 0094 loss = 0.000205
Epoch: 0095 loss = 0.000238
Epoch: 0096 loss = 0.000245
Epoch: 0097 loss = 0.000233
Epoch: 0098 loss = 0.000171
Epoch: 0099 loss = 0.000135
Epoch: 0100 loss = 0.000151

==============================
利用训练好的Transformer模型将中文句子'我 有 一 只 猫' 翻译成英文句子:
tensor([1, 2, 3, 4, 6, 0]) -> tensor([1, 2, 3, 5, 9])
['我', '有', '一', '只', '猫', 'P'] -> ['i', 'have', 'a', 'cat', '.']
```

六、参考文献

[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.

附录：代码实现

- **Transformer**

```
class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        self.encoder = Encoder().to(device)
        self.decoder = Decoder().to(device)
        self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False).to(device)

    def forward(self, enc_inputs, dec_inputs):
        """Transformers 的输入：两个序列
        enc_inputs: [batch_size, src_len]
        dec_inputs: [batch_size, tgt_len]
        """
        # tensor to store decoder outputs
        # outputs = torch.zeros(batch_size, tgt_len, tgt_vocab_size).to(self.device)

        # enc_outputs: [batch_size, src_len, d_model], enc_self_attns: [n_layers, batch_size, n_heads, src_len, src_len]
        # 经过 Encoder 网络后，得到的输出还是[batch_size, src_len, d_model]
        enc_outputs, enc_self_attns = self.encoder(enc_inputs)
        # dec_outputs: [batch_size, tgt_len, d_model], dec_self_attns: [n_layers, batch_size, n_heads, tgt_len, tgt_len], dec_enc_attn: [n_layers, batch_size, tgt_len, src_len]
        dec_outputs, dec_self_attns, dec_enc_attns = self.decoder(dec_inputs, enc_inputs, enc_outputs)
        # dec_outputs: [batch_size, tgt_len, d_model] -> dec_logits: [batch_size, tgt_len, tgt_vocab_size]
        dec_logits = self.projection(dec_outputs)
        return dec_logits.view(-1, dec_logits.size(-1)), enc_self_attns, dec_self_attns, dec_enc_attns
```

- **Encoder**

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.src_emb = nn.Embedding(src_vocab_size, d_model)  # token Embedding
        self.pos_emb = PositionalEncoding(d_model)  # Transformer 中位置编码时固定的，不需要学习
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])

    def forward(self, enc_inputs):
        """
        enc_inputs: [batch_size, src_len]
        """
        enc_outputs = self.src_emb(enc_inputs)  # [batch_size, src_len, d_model]
        enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1)  # [batch_size, src_len, d_model]
        # Encoder 输入序列的 pad mask 矩阵
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)  # [batch_size, src_len, src_len]
        enc_self_attns = []  # 在计算中不需要用到，它主要用来保存你接下来返回的 attention 的值（这个主要是为了你画热力图等，用来看各个词之间的关系
        for layer in self.layers:  # for 循环访问 nn.ModuleList 对象
            # 上一个 block 的输出 enc_outputs 作为当前 block 的输入
            # enc_outputs: [batch_size, src_len, d_model], enc_self_attn: [batch_size, n_heads, src_len, src_len]
            enc_outputs, enc_self_attn = layer(enc_outputs,
                                               enc_self_attn_mask)  # 传入的 enc_outputs 其实是 input，传入 mask 矩阵是因为你要做 self attention
            enc_self_attns.append(enc_self_attn)  # 这个只是为了可视化
        return enc_outputs, enc_self_attns
```

- **Decoder**

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)  # Decoder 输入的 embed 词表
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([DecoderLayer() for _ in range(n_layers)])  # Decoder 的 blocks

    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        """
        dec_inputs: [batch_size, tgt_len]
        enc_inputs: [batch_size, src_len]
        enc_outputs: [batch_size, src_len, d_model]   # 用在 Encoder-Decoder Attention 层
        """
        dec_outputs = self.tgt_emb(dec_inputs)  # [batch_size, tgt_len, d_model]
        dec_outputs = self.pos_emb(dec_outputs.transpose(0, 1)).transpose(0, 1).to(
            device)  # [batch_size, tgt_len, d_model]
        # Decoder 输入序列的 pad mask 矩阵（这个例子中 decoder 是没有加 pad 的，实际应用中都是有 pad 填充的）
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs).to(device)  # [batch_size, tgt_len, tgt_len]
        # Masked Self_Attention：当前时刻是看不到未来的信息的
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs).to(
            device)  # [batch_size, tgt_len, tgt_len]

        # Decoder 中把两种 mask 矩阵相加（既屏蔽了 pad 的信息，也屏蔽了未来时刻的信息）
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask + dec_self_attn_subsequence_mask),
                                      0).to(device)  # [batch_size, tgt_len, tgt_len]; torch.gt 比较两个矩阵的元素，大于则返回 1，否则返回 0

        # 这个 mask 主要用于 encoder-decoder attention 层
        # get_attn_pad_mask 主要是 enc_inputs 的 pad mask 矩阵(因为 enc 是处理 K,V 的，求 Attention 时是用 v1,v2,..vm 去加权的，要把 pad 对应的 v_i 的相关系数设为 0，这样注意力就不会关注 pad 向量)
```

```
        #                          dec_inputs 只是提供expand 的size 的
        dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs)  # [ba
tc_size, tgt_len, src_len]

        dec_self_attns, dec_enc_attns = [], []
        for layer in self.layers:
            # dec_outputs: [batch_size, tgt_len, d_model], dec_self_attn: [b
atch_size, n_heads, tgt_len, tgt_len], dec_enc_attn: [batch_size, h_heads, t
gt_len, src_len]
            # Decoder 的Block 是上一个Block 的输出dec_outputs（变化）和Encoder
网络的输出enc_outputs（固定）
            dec_outputs, dec_self_attn, dec_enc_attn = layer(dec_outputs, en
c_outputs, dec_self_attn_mask,
                                                             dec_enc_attn_ma
sk)
            dec_self_attns.append(dec_self_attn)
            dec_enc_attns.append(dec_enc_attn)
        # dec_outputs: [batch_size, tgt_len, d_model]
        return dec_outputs, dec_self_attns, dec_enc_attns
```

- **位置编码**

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        x: [seq_len, batch_size, d_model]
        """
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)
```

- 多头注意力

```
class MultiHeadAttention(nn.Module):
    """这个 Attention 类可以实现:
    Encoder 的 Self-Attention
    Decoder 的 Masked Self-Attention
    Encoder-Decoder 的 Attention
    输入：seq_len x d_model
    输出：seq_len x d_model
    """
    def __init__(self):
        super(MultiHeadAttention, self).__init__()
        self.W_Q = nn.Linear(d_model, d_k * n_heads, bias=False)  # q,k 必须维度相同，不然无法做点积
        self.W_K = nn.Linear(d_model, d_k * n_heads, bias=False)
        self.W_V = nn.Linear(d_model, d_v * n_heads, bias=False)
        # 这个全连接层可以保证多头 attention 的输出仍然是 seq_len x d_model
        self.fc = nn.Linear(n_heads * d_v, d_model, bias=False)

    def forward(self, input_Q, input_K, input_V, attn_mask):
        """
        input_Q: [batch_size, len_q, d_model]
        input_K: [batch_size, len_k, d_model]
        input_V: [batch_size, len_v(=len_k), d_model]
        attn_mask: [batch_size, seq_len, seq_len]
        """
        residual, batch_size = input_Q, input_Q.size(0)
        # 下面的多头的参数矩阵是放在一起做线性变换的，然后再拆成多个头，这是工程实现的技巧
        # B: batch_size, S:seq_len, D: dim
        # (B, S, D) -proj-> (B, S, D_new) -split-> (B, S, Head, W) -trans-> (B, Head, S, W)
        #           线性变换                    拆成多头

        # Q: [batch_size, n_heads, len_q, d_k]
        Q = self.W_Q(input_Q).view(batch_size, -1, n_heads, d_k).transpose(1, 2)
        # K: [batch_size, n_heads, len_k, d_k] # K 和 V 的长度一定相同，维度可以不同
        K = self.W_K(input_K).view(batch_size, -1, n_heads, d_k).transpose(1, 2)
        # V: [batch_size, n_heads, len_v(=len_k), d_v]
        V = self.W_V(input_V).view(batch_size, -1, n_heads, d_v).transpose(1, 2)
```

```
36.
37.         # 因为是多头，所以mask矩阵要扩充成4维的
38.         # attn_mask: [batch_size, seq_len, seq_len] -> [batch_size, n_heads,
    seq_len, seq_len]
39.         attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1)
40.
41.         # context: [batch_size, n_heads, len_q, d_v], attn: [batch_size, n_h
    eads, len_q, len_k]
42.         context, attn = ScaledDotProductAttention()(Q, K, V, attn_mask)
43.         # 下面将不同头的输出向量拼接在一起
44.         # context: [batch_size, n_heads, len_q, d_v] -> [batch_size, len_q,
    n_heads * d_v]
45.         context = context.transpose(1, 2).reshape(batch_size, -
    1, n_heads * d_v)
46.
47.         # 这个全连接层可以保证多头attention的输出仍然是seq_len x d_model
48.         output = self.fc(context)  # [batch_size, len_q, d_model]
49.         return nn.LayerNorm(d_model).to(device)(output + residual), attn
```

- **掩码**

```
1.  def get_attn_pad_mask(seq_q, seq_k):
2.      # pad mask的作用：在对value向量加权平均的时候，可以让pad对应的alpha_ij=0，
    这样注意力就不会考虑到pad向量
3.      """这里的q,k表示的是两个序列（跟注意力机制的q,k没有关系），例如
    encoder_inputs (x1,x2,..xm)和encoder_inputs (x1,x2..xm)
4.      encoder和decoder都可能调用这个函数，所以seq_len视情况而定
5.      seq_q: [batch_size, seq_len]
6.      seq_k: [batch_size, seq_len]
7.      seq_len could be src_len or it could be tgt_len
8.      seq_len in seq_q and seq_len in seq_k maybe not equal
9.      """
10.     batch_size, len_q = seq_q.size()  # 这个seq_q只是用来expand维度的
11.     batch_size, len_k = seq_k.size()
12.     # eq(zero) is PAD token
13.     # 例如:seq_k = [[1,2,3,4,0], [1,2,3,5,0]]
14.     pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)  # [batch_size, 1, len_k],
     True is masked
15.     return pad_attn_mask.expand(batch_size, len_q, len_k)  # [batch_size, le
    n_q, len_k] 构成一个立方体(batch_size 个这样的矩阵)
16.
17.
18. def get_attn_subsequence_mask(seq):
```

```
    """建议打印出来看看是什么的输出（一目了然）
    seq: [batch_size, tgt_len]
    """
    attn_shape = [seq.size(0), seq.size(1), seq.size(1)]
    # attn_shape: [batch_size, tgt_len, tgt_len]
    subsequence_mask = np.triu(np.ones(attn_shape), k=1)  # 生成一个上三角矩阵
    subsequence_mask = torch.from_numpy(subsequence_mask).byte()
    return subsequence_mask  # [batch_size, tgt_len, tgt_len]

```

- **前馈网络**

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        x: [seq_len, batch_size, d_model]
        """
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)
```