第 5 章

Transformer 模型搭建

5.1 Transformer 模型编码器

在本书的前 4 章中，已经探讨了 Transformer 模型的关键基础核心知识，包括诸如词嵌入和位置编码、注意力机制和多头注意力机制、残差连接与数据归一化操作，以及前馈神经网络等 Transformer 核心部件。有了这些基础知识，就可以构建完整的 Transformer 模型了。

5.1.1 Transformer 模型编码器组成

在 Transformer 模型的结构中，编码器由 6 个完全相同的层组成，每层都包括多头注意力机制和前馈神经网络。数据按顺序通过每层编码器，在全部经过 6 层编码器层处理后，最终的输出数据矩阵会被传递给解码器进一步地进行数据交换，如图 5-1 所示。

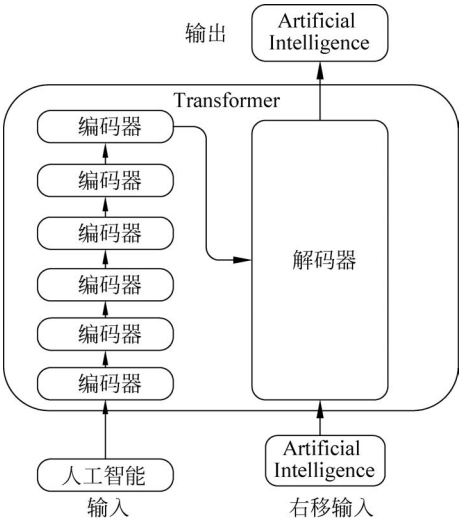

图 5-1 Transformer 模型编码器

Transformer 模型选择使用 6 个编码器层并非出于某种特别的考虑，而是经过一系列的实验和优化得到的结果。

（1）选择层数需要在复杂性和计算资源之间进行权衡。增加层数可以提高模型的泛化能力，但同时也会提高计算成本，然而，在很多自然语言处理任务中，6 层编码器已经能够获得相当高的性能，同时也降低了计算成本。

（2）选择 6 个编码器层的决策也考虑了训练数据的规模和多样性。如果训练数据较少或较为简单，则可能只需较少的层数就已经足够，然而，如果训练数据庞大或复杂，则增加层数可能有助于模型捕捉数据中更多的依赖关系。

（3）选择 6 个编码器层可以在保持模型深度和防止过拟合之间找到一个平衡点。具体多少层最佳，需要基于自己的数据和模型复杂度来进行决策。也就是说，Transformer 模型使用了 6 层编码器，旨在平衡计算成本和泛化能力，而没有任何特殊含义。当数据复杂性增加或计算能力提升时，模型层数和参数可能需要增加。

5.1.2　Transformer 模型编码器层的代码实现

无论是编码器还是解码器都由 6 个编码器层或者解码器层组成。每个编码器层或者解码器层都包含多头注意力机制和前馈神经网络。在数据经过单个功能模块时都会进行一次残差连接与数据归一化操作。Transformer 模型中包含 6 层的编码器层和 6 层的解码器层，每层的结构都完全相同。在代码设计上，只需编写一个编码器层和解码器层函数，剩下的部分可以通过循环迭代 6 次实现。

编码器层的代码如下：

```
#第 5 章/5.1.2/Transformer 模型编码器层代码实现
input = torch.LongTensor([[5,2,1,0,0],[1,3,1,4,0]])        #输入数据
d_model = 512                                              #词嵌入维度
d_ff = 2048                                                #512->2048->512
d_k = d_v = 64                                             #多头注意力机制的维度 K(=Q), V
n_layers = 6                                               #编码器与解码器 Block 的个数
n_heads = 8                                                #多头注意力机制的头数
src_vocab_size = 10                                        #最大句子长度
class EncoderLayer(nn.Module):                             #编码器层函数
    def __init__(self):
        super(EncoderLayer, self).__init__()
        self.enc_self_attn = MultiHeadAttention()          #编码器层包含多头注意力机制
        self.pos_ffn = PoswiseFeedForwardNet()             #编码器层包含前馈神经网络
    def forward(self, enc_inputs, enc_self_attn_mask):
        #enc_inputs: [batch_size, src_len, d_model]编码器层输入数据维度
        #mask 矩阵(pad mask or sequence mask)pad mask 维度
        #enc_self_attn_mask: [batch_size, src_len, src_len]
        #编码器层的输出数据维度
        #enc_outputs: [batch_size, src_len, d_model] [2,5,512]
        #注意力矩阵的维度
        #attn: [batch_size, n_heads, src_len, src_len] [2,5,5]
```

```
        #输入数据首先经过多头注意力机制,输出数据维度[2,5,512]
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs,
                                               enc_inputs,enc_self_attn_mask)
        #输入数据再经过前馈神经网络,输出数据维度[2,5,512]
        enc_outputs = self.pos_ffn(enc_outputs)     #[2,5,512]
        #enc_outputs: [batch_size, src_len, d_model]
        return enc_outputs, attn

enc_layer = EncoderLayer()                                      #初始化编码器层函数
enc_layer_result , atten = enc_layer(pes,enc_pad_mask)   #输入数据,计算注意力
print('enc_layer_result',enc_layer_result)                      #输出经过编码器层的数据
print(enc_layer_result.shape)       #输出经过编码器层的数据维度
```

首先,在初始化部分定义一些超参数,然后建立一个编码器层函数。在此过程中,需要定义多头注意力机制和前馈神经网络函数。这两部分的代码可以参考多头注意力机制与前馈神经网络章节的代码实现。

在实现函数部分,需要输入两个参数。一个是编码器的输入矩阵,该矩阵是经过词嵌入和位置编码后的矩阵,其数据维度为[2,5,512]。另一个是编码器的掩码矩阵,对编码器来讲,掩码矩阵只需 Pad Mask 矩阵,其维度为[2,5,5],然后将输入矩阵和掩码矩阵传递给多头注意力机制函数进行计算。函数返回一个注意力机制的结果,其矩阵维度仍为[2,5,512]。另一个返回参数为注意力矩阵,注意力矩阵一般用来可视化注意力机制,其矩阵维度为[2,8,5,5]。

(1) 2: Batch Size 的维度。

(2) 8: 多头注意力机制的头数。

(3) [5,5]: 每个头的 Pad Mask 矩阵。

然后将多头注意力机制的结果传递给前馈神经网络,得到编码器层的最终输出矩阵,其矩阵维度仍为[2,5,512]。最后,直接返回最终的输出矩阵和可视化注意力矩阵。

最后,可以初始化编码器层函数,将经过词嵌入和位置编码的矩阵及 Pad Mask 矩阵传递给编码器层函数。这里可以输出经过一个编码器层后的输出矩阵及矩阵的维度,可以看到其输出矩阵的维度仍然是[2,5,512],其输出如下:

```
enc_layer_result tensor([[
        [-0.5642, 0.6319, -1.6371, ..., -0.4036, 1.3698, 0.7781],
        [-0.2029, -0.5430, 0.0197, ..., -0.2712, 0.0181, 1.5683],
        [ 0.1140, 0.2228, -0.4459, ..., -2.2531, -0.4930, 1.4589],
        [ 0.4019, -1.2653, 1.7782, ..., 0.4035, -0.7254, 1.9217],
        [-0.3809, -0.9320, 1.0707, ..., 0.4113, -0.7139, 1.9630]],

       [[-0.7108, 1.2283, -0.9787, ..., -2.1162, -0.4393, 1.6217],
        [-1.2407, -0.2840, -0.0511, ..., 0.1400, 0.6175, 1.0268],
        [ 0.0642, 0.0590, -0.2607, ..., -2.1618, -0.4967, 1.6326],
        [ 1.3433, 0.1424, -0.0636, ..., -0.1151, 0.8437, 0.8036],
        [-0.2812, -1.0145, 1.2016, ..., 0.3304, -0.8454, 1.9554]]],
```

```
        grad_fn=<NativeLayerNormBackward>)
torch.Size([2, 5, 512])
```

5.1.3　搭建 Transformer 模型编码器

有了一层编码器层函数以后，就可以通过循环来搭建全部的编码器函数。或者，还可以摒弃循环，而是按照 Transformer 模型框架和数据流向，层层搭建 6 个编码器层，以此实现整个 Transformer 模型的编码器。

Transformer 模型编码器的代码如下：

```
#第 5 章/5.1.3/Transformer 模型编码器代码实现
input = torch.LongTensor([[5,2,1,0,0],[1,3,1,4,0]])   #输入数据
d_model = 512                                          #词嵌入维度
d_ff = 2048                                            #512->2048->512
d_k = d_v = 64                                         #多头注意力机制的维度 K(=Q), V
n_layers = 6                                           #编码器与解码器 Block 的个数
n_heads = 8                                            #多头注意力机制的头数
src_vocab_size = 10                                    #最大句子长度
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.src_emb = Embeddings(src_vocab_size, d_model)     #词嵌入函数
        self.pos_emb = PositionalEncoding(d_model)             #位置编码函数
        #使用 for 循环，循环 6 次
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])

    def forward(self, enc_inputs):
        #enc_inputs: [batch_size, src_len]                 #[2,5]输入数据维度
        enc_outputs = self.src_emb(enc_inputs)            #[2, 5, 512],词嵌入
        #enc_outputs [batch_size, src_len, src_len]       #[2, 5, 512],词嵌入维度
        #添加位置编码
        enc_outputs = self.pos_emb(enc_outputs.transpose(0,1)).transpose(0, 1)
        #输入 Pad Mask 矩阵,[batch_size, src_len, src_len] [2,5,5]
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)
        enc_self_attns = []                               #为了可视化注意力矩阵
        for layer in self.layers:     #for 循环访问 nn.ModuleList,进行 6 次循环堆叠
            #enc_outputs: [batch_size, src_len, d_model],[2, 5, 512]
            #编码器输出数据维度
            #enc_self_attn: [batch_size, n_heads, src_len, src_len] [2,8,5,5]
            #多头注意力矩阵维度
            enc_outputs, enc_self_attn =layer(enc_outputs,enc_self_attn_mask)
            #计算每层的注意力机制，一共计算 6 层
            enc_self_attns.append(enc_self_attn)  #保存注意力矩阵
        return enc_outputs, enc_self_attns        #编码器输出 enc_outputs[2, 5, 512]

encoder = Encoder()                               #初始化编码器
encoder_result,enc_attens = encoder (input)       #传递输入数据，计算 6 层的编码器
```

```
print('Encoder_result',encoder_result)        #输出编码器的数据
print(encoder_result.shape)                   #输出编码器的数据维度
```

在 Transformer 编码器的构造函数中，首先需要对词嵌入及位置编码函数进行初始化，然后通过 nn.ModuleList 函数对 6 层编码器层函数进行保存，接下来便可在实现函数中直接调用此 ModuleList，进而构建 Transformer 编码器模型。

在实现函数过程中，首先将输入数据传递给词嵌入函数，进行输入数据的词嵌入处理，输入数据的维度为[2,5]。“2”代表一次性输入的句子数为两个，“5”则表示每个句子中包含 5 个汉字。在输入数据中，数字“0”指的是 Pad Mask 的标记。经过词嵌入后，输出数据维度变为[2,5,512]。接下来，输入数据还需经过位置编码函数处理，词嵌入后的数据加上位置编码，其数据维度依然是[2,5,512]。之后该数据才能传递给 Transformer 模型的编码器。

在进行注意力计算之前，需要计算输入数据的 Pad Mask 矩阵，以方便后期进行注意力的 Softmax 操作。在这里，由于是编码器层，所以只需使用 Pad Mask 矩阵，不需要 Sequence Mask 矩阵。这里可以直接利用掩码矩阵章节中介绍的掩码矩阵函数来生成输入数据的 Pad Mask 矩阵，其维度为[2,5,5]。这样，得到以上所有的输入数据后，就可以开始计算多头注意力机制了。

通过 for 循环函数，可以依序调用编码器层函数来计算注意力机制。注意力机制函数有两个输入变量，一个是维度为[2,5,512]的编码器输入数据，另一个是维度为[2,5,5]的输入数据的 Pad Mask 矩阵。当计算完成后，将得到最终的编码器输出矩阵，其维度仍是[2,5,512]。同样，代码也会保存所有的注意力矩阵，通常该矩阵被用来进行注意力机制的可视化操作，其矩阵维度为[2,8,5,5]，代表 8 个头的注意力矩阵。最后，只需返回经过 6 层编码器层处理的数据。

以上便是整个 Transformer 模型编码器的具体实现流程。最终，可以利用此函数，将输入变量传递给编码器函数，并输出数据及其维度进行展示。可以看出，输出的矩阵维度仍然是[2,5,512]，其输出如下：

```
encoder_result tensor([[
        [-1.2558, 0.5679, -0.2423, ..., 2.2966, -1.4775, -0.0496],
        [ 0.7694, -0.0459, -0.2256, ..., 1.2954, 0.4919, 0.4077],
        [ 0.4128, -0.7950, 1.0712, ..., 1.0821, -0.4719, 0.2722],
        [ 0.7594, -1.1542, 0.6070, ..., 1.8225, 0.1423, 0.4011],
        [ 0.2369, -0.8219, 0.1158, ..., 1.8352, 0.1824, 0.3232]],

       [[-0.3012, -0.4196, 0.2384, ..., 0.8722, -0.4522, 0.9629],
        [ 0.4609, -1.7442, -0.1379, ..., 1.2779, -1.2562, 0.9374],
        [ 0.1731, -1.4041, 0.5622, ..., 0.8569, -0.4222, 0.7444],
        [ 0.0784, -3.4037, -0.4995, ..., 0.7952, -0.4251, 0.5451],
        [-0.1042, -1.5095, -0.2837, ..., 1.1985, 0.2166, 0.8027]]],
      grad_fn=<NativeLayerNormBackward>)
torch.Size([2, 5, 512])
```

5.2　Transformer 模型解码器

Transformer 模型采用了完全对称的设计，其编码器由 6 层组成。那么与此相对应解码器必然也由 6 层组成。

5.2.1　Transformer 模型解码器组成

正如编码器那样，解码器也是由 6 层结构一致的解码器层构成的。这里的 6 层解码器与 6 层的编码器相关联，构造较为对称。值得注意的是，只有在 6 层编码器的最终输出阶段，数据才会被传输到 6 层的解码器层，如图 5-2 所示。

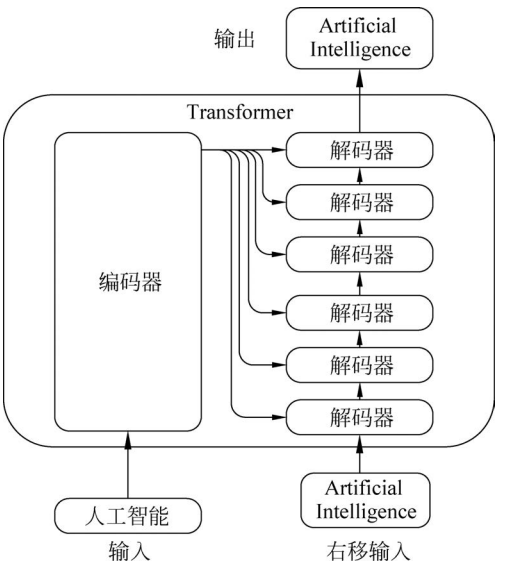

图 5-2　Transformer 模型解码器

然而，每层的解码器层不仅有多头注意力机制和前馈神经网络，还嵌有一层注意力机制交互层。此层通过对解码器层与编码器层的数据进行交流，进行交叉注意力机制的计算，其 $\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵源自编码器，$\boldsymbol{Q}$ 矩阵源自解码器。实际上，这个交叉注意力机制交互层也是一层多头注意力机制，只是其中的 $\boldsymbol{Q}$ 矩阵由解码器提供，而 $\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵则来自编码器的最终输出数据，如图 5-3 所示。

5.2.2　Transformer 模型解码器层的代码实现

解码器部分代码在编写上与编码器部分的代码有许多相似之处，然而，它们之间也存在一些关键的区别：

(1) 由于解码器需要屏蔽未来信息的访问，所以除了 Pad Mask 矩阵之外，还引入了 Sequence Mask 矩阵。

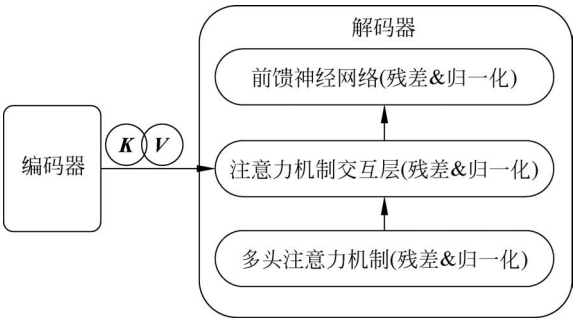

图 5-3 Transformer 模型交叉注意力机制交互层

(2) 解码器部分比编码器部分多出一层交叉注意力机制交互层。

解码器层的代码如下：

```
#第 5 章/5.2.2/ Transformer 模型解码器层代码实现
class DecoderLayer(nn.Module):
    def __init__(self):
        super(DecoderLayer, self).__init__()
        #decoder 自注意力机制
        self.dec_self_attn = MultiHeadAttention()
        #decoder enc_dec_attention 交互层
        self.dec_enc_attn = MultiHeadAttention()
        #decoder 前馈神经网络
        self.pos_ffn = PoswiseFeedForwardNet()
    def forward(self, dec_inputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_
mask):
        #dec_inputs: [batch_size, tgt_len, d_model]                #[2,5,512]
        #dec_self_attn_mask: [batch_size, tgt_len, tgt_len]        #[2,5,5]
        #dec_outputs: [batch_size, tgt_len, d_model]               #[2,5,512]
        #dec_self_attn: [batch_size, n_heads, tgt_len, tgt_len]    #[2,8,5,5]
        #decoder 自注意力机制,Q、K、V 来自 Decoder 的输入              #[2,5,512]
        dec_outputs, dec_self_attn = self.dec_self_attn(dec_inputs, dec_inputs,
dec_inputs,dec_self_attn_mask)                    #计算自注意力机制
        #dec_outputs: [batch_size, tgt_len, d_model]               #[2,5,512]
        #enc_outputs: [batch_size, src_len, d_model]               #[2,5,512]
        #dec_enc_attn: [batch_size, h_heads, tgt_len, src_len]     #[2,8,5,5]
        #dec_enc_attn_mask: [batch_size, tgt_len, src_len]         #[2,5,5]
        #这里 encoder 输入长度与 decoder 输入句子长度不一定相等,本程序句子的长度是一
        #样的
        #dec_enc_Attention 层的 Q(来自 decoder), K、V(来自 encoder)  #[2,5,512]
        dec_outputs, dec_enc_attn = self.dec_enc_attn(dec_outputs, enc_outputs,
enc_outputs,dec_enc_attn_mask)                    #计算交叉注意力机制
        dec_outputs = self.pos_ffn(dec_outputs)      #[2,5,512]前馈神经网络
        #dec_self_attn、dec_enc_attn 两个矩阵用于可视化
        return dec_outputs, dec_self_attn, dec_enc_attn
```

新建一层解码器层函数,该层是解码器的一层输入函数。整个解码器由 6 层解码器层

组成。可以按照编码器的搭建方式,使用 for 循环来构建解码器。

初始化部分:

(1) 建立一个解码器层的多头注意力机制函数。

(2) 建立一个解码器层的交叉注意力机制交互层函数,其本质上也是一层多头注意力机制。

(3) 建立一个前馈神经网络。

实现函数部分:

输入 4 个参数,其中 decoder-input 是解码器的输入矩阵(矩阵维度为[2,5,512]),encoder-output 是编码器的输出矩阵(矩阵维度为[2,5,512]),另外两个参数为掩码矩阵,其中,解码器的多头注意力机制使用的掩码矩阵维度为[2,5,5],交叉注意力机制交互层使用的掩码矩阵维度为[2,8,5,5]。多头自注意力层的掩码矩阵包含 Pad Mask 矩阵与 Sequence Mask 矩阵。

解码器的输入数据需要经过解码器层的多头注意力机制计算注意力,其中的 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 三个矩阵都来源于解码器的输入,其矩阵维度为[2,5,512]。需要注意的是,解码器的输入与编码器的输入句子长度并不一定相等。函数返回多头注意力机制计算的结果矩阵,并返回多头注意力矩阵,该矩阵可用于绘制热力图,便于查看单词之间的语义关系。

多头注意力机制的输出还需要经过一层交叉注意力机制交互层。该层的 $\boldsymbol{Q}$ 矩阵来源于解码器层多头注意力机制的输出,而 $\boldsymbol{K}$ 和 $\boldsymbol{V}$ 矩阵来源于编码器层的最终输出。同样,交叉注意力机制交互层也要使用 Pad Mask 矩阵。针对以上代码,Pad Mask 矩阵的维度为[2,5,5]。需要注意的是,编码器的输入句子长度与解码器的输入句子长度不一定相等,因此,该掩码矩阵的维度为[batch-size, decoder input length, encoder input length]。

(1) batch-size:批处理维度。

(2) decoder input length:解码器输入数据的长度。

(3) encoder input length:编码器输入数据的长度。

经过交叉注意力机制交互层后,输出数据维度仍为[2,5,512],而输出矩阵还需要经过一层前馈神经网络。函数返回解码器的最终输出与两个注意力矩阵(一个自注意力矩阵,一个交叉注意力矩阵)。

5.2.3 搭建 Transformer 模型解码器

可以借助解码器层函数来构建出完整的 Transformer 模型解码器,构建过程中参照的是编码器层的设计,使用 for 循环函数依次循环生成 6 层的解码器层,代码如下:

```
#第 5 章/5.2.3/ Transformer 模型解码器代码实现
enc_input = torch.LongTensor([[5,2,1,0,0],[1,3,1,4,0]])
dec_input = torch.LongTensor([[5,2,1,0,0],[1,3,1,4,0]])
enc_output = encoder_result                    #编码器的最终输出
d_model = 512                                  #词嵌入维度
d_ff = 2048                                    #(两次线性层中的隐藏层 512->2048->512,
```

```
#线性层是用来进行特征提取的),当然最后会再接一个线性层
d_k = d_v = 64                      #多头注意力维度 K(=Q), V
n_layers = 6                        #编码器与解码器层数
n_heads = 8                         #多头注意力头数
src_vocab_size = 10                 #编码器输入最大句子长度
tgt_vocab_size = 10                 #解码器输入最大句子长度,两个输入句子长度不一定相等
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = Embeddings(tgt_vocab_size, d_model)   #词嵌入
        self.pos_emb = PositionalEncoding(d_model)           #位置编码
        #DecoderLayer block 一共 6 层,与 encoder 相同
        self.layers = nn.ModuleList([DecoderLayer() for _ in range(n_layers)])
    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        #dec_inputs: [batch_size, tgt_len]                   [2,5]
        #enc_inputs: [batch_size, src_len]                   [2,5]
        #enc_outputs 用在编码器-解码器注意力交互层
        #enc_outputs: [batch_size, src_len, d_model]         [2,5,512]
        dec_outputs = self.tgt_emb(dec_inputs)               #[2,5,512]词嵌入维度
        #dec_outputs 位置编码+embedding 词嵌入                #[2,5,512]添加位置编码
        dec_outputs = self.pos_emb(dec_outputs.transpose(0, 1)).transpose(0, 1)
        #解码器输入序列的 Pad Mask 矩阵                       #[2,5,5]
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs)
        #解码器输入序列的 Sequence Mask 矩阵                  #[2,5,5]
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs)
        #解码器中把 Pad Mask + Sequence Mask
        #既屏蔽了 pad 的信息,也屏蔽了未来的信息              #[2,5,5]
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask +
                                       dec_self_attn_subsequence_mask),0)
        #dec_enc mask 主要用于编码器-解码器注意力交互层
        #因为 dec_enc_attn 输入是编码器的 K、V,解码器的 Q
        #dec_inputs 提供扩展维度的大小
        #[batc_size, tgt_len, src_len],这里 tgt_len 与 src_len 不一定相等  #[2,5,5]
        dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs)
        #用于可视化的矩阵,一个是 dec_self-attention,另一个是 enc_dec_attention
        dec_self_attns, dec_enc_attns = [], []
        for layer in self.layers:       #遍历 decoder block, n = 6
            #dec_outputs:[batch_size,tgt_len,d_model]解码器的输入[2,5,512]
            #enc_outputs:[batch_size,src_len,d_model]编码器的输入[2,5,512]
            #dec_self_attn:[batch_size,n_heads,tgt_len,tgt_len]    [2,8,5,5]
            #dec_enc_attn:[batch_size,h_heads,tgt_len,src_len]     [2,8,5,5]
            #解码器的 Block 是上一个 Block 的输出 dec_outputs(变化矩阵)
            #编码器网络的输出 enc_outputs(固定矩阵)
            dec_outputs, dec_self_attn, dec_enc_attn = layer(dec_outputs,
                         enc_outputs, dec_self_attn_mask,dec_enc_attn_mask)
            dec_self_attns.append(dec_self_attn)       #可视化矩阵 [2,8,5,5]
            dec_enc_attns.append(dec_enc_attn)         #可视化矩阵 [2,8,5,5]
        #dec_outputs: [batch_size, tgt_len, d_model]    #[2,5,512]
        return dec_outputs, dec_self_attns, dec_enc_attns
```

```
decoder = Decoder()
decoder_result,self_attn,de_enc_attn=decoder(dec_input,enc_input,enc_output)
print('decoder_result', decoder_result)
print(decoder_result.shape)
```

初始阶段需要3个输入数据，分别是编码器的输入（主要用于交叉注意力机制交互层的Pad Mask计算）、解码器的输入，以及编码器层的最终输出，这样才能进行交叉注意力机制的计算。

随后创建解码器类函数。在初始化部分，初始化所需的词嵌入与位置编码函数。与编码器搭建过程类似，用Module.List来保存6层的解码器层函数，这样就方便使用for循环构建解码器。

在实现函数部分，接收3个输入参数，分别是解码器的输入（其输入矩阵维度为[2,5]）、编码器的输入（其输入矩阵维度为[2,5]，主要用于交叉注意力机制交互层的Pad Mask计算），以及编码器的最终输出（其矩阵维度为[2,5,512]，用于进行交叉注意力机制的计算）。

首先，解码器的输入矩阵在经过词嵌入与位置编码后才能输入解码器层，此时其矩阵维度为[2,5,512]。在执行多头注意力机制之前，首先计算解码器输入矩阵的Pad Mask矩阵，其矩阵维度为[2,5,5]。交叉注意力机制交互层也需要Pad Mask矩阵，因此，还需要用到编码器与解码器的输入数据来计算交叉注意力机制交互层的Pad Mask矩阵。此矩阵维度为[2,5,5]，但在实际应用过程中，编码器和解码器的输入长度并不一定完全相等，因此实际上此矩阵的维度为[batch size,decoder input length,encoder input length]。

最后，把经过词嵌入和位置编码后的矩阵（矩阵维度为[2,5,512]）、编码器的最终输出矩阵（矩阵维度为[2,5,512]）及两个掩码矩阵一并交给解码器层来进行注意力机制的计算。该函数将返回经过处理后的输出矩阵和多头注意力矩阵及交叉注意力机制层的注意力矩阵，其矩阵维度为[2,8,5,5]。这里利用两个列表变量来记录两个可视化矩阵，以备后续绘制热力图使用。

经过6层的解码器层处理之后，直接返回最终的输出矩阵，其矩阵维度依然是[2,5,512]。最终可以初始化解码器类函数，并输入函数所需的参数，并打印出通过解码器函数处理后的最终输出及输出矩阵维度。可以看到，其输出矩阵维度依然是[2,5,512]，其输出如下：

```
decoder_result tensor([[
        [-1.2468e+00, 4.5989e-01, -1.3258e-01, ..., -1.0994e+00,-5.6946e-01,
1.0559e+00],
        [-2.9721e-01, 6.7347e-01, 2.3816e-01, ..., -2.2039e+00,1.2414e+00,
1.0486e+00],
        [-4.9882e-01, 5.1537e-01, -1.5587e+00, ..., -3.3027e-01,-6.7481e-01,
1.6427e+00],
        [-1.9220e+00, -1.1277e+00, -1.2631e+00, ..., 2.3327e-01,-5.6917e-01,
1.3093e+00],
```

```
        [-2.3022e+00, -8.9679e-01, -1.6558e+00, ..., 2.9868e-01,-6.3239e-01,
1.3052e+00]],

       [[-1.9229e+00, -2.9489e-01, -2.1268e+00, ..., 3.0097e-01,-6.7756e-01,
8.8094e-01],
        [ 1.7832e-01, -9.7633e-01, -1.6108e+00, ..., -6.8099e-02,5.9771e-01,
-7.9235e-02],
        [-1.0999e+00, -6.9100e-01, -1.6478e+00, ..., 3.6269e-01,-5.6357e-01,
1.1377e+00],
        [-2.2280e+00, -1.6373e+00, -1.7092e+00, ..., 4.3180e-01,-3.5748e-01,
9.9174e-04],
        [-2.6631e+00, -1.7869e+00, -2.3886e+00, ..., 5.8578e-01,-7.6391e-01,
8.0873e-01]]],
        grad_fn=<NativeLayerNormBackward>)
torch.Size([2, 5, 512])
```

5.3 搭建 Transformer 模型

Transformer 模型，除了编码器和解码器部分，还需要输入和输出部分，这些部分共同构成了完整的 Transformer 模型。

5.3.1 Transformer 模型组成

Transformer 模型编码器和解码器都由 6 层完全相同的结构组成，编码器为解码器提供 $\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵，使编码器和解码器之间的数据进行交互，从而实现了将一种语言转换为另一种语言的机器学习任务，如图 5-4 所示。

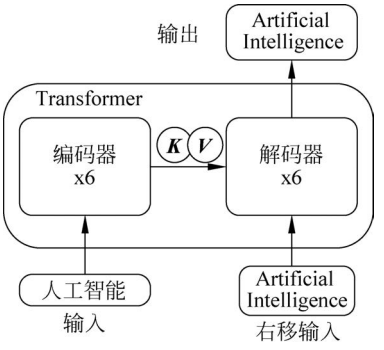

图 5-4 Transformer 模型框架图

在编码器和解码器之外，Transformer 模型还包括以下几个重要组件。

(1) 注意力机制：Transformer 模型使用自注意力机制来捕捉输入序列中不同位置之间的依赖关系。注意力机制使编码器和解码器之间的输入序列进行数据交互，这也是 Transformer 模型最重要的创新之一。

(2) 多头注意力：为了提高模型的表达能力，Transformer 模型使用多个注意力头来进行注意力机制的计算。每个注意力头都可以关注不同的部分，然后对它们的输出进行拼接或加权求和，以获得更全面的表示。

(3) 位置编码：由于 Transformer 模型没有使用循环神经网络或卷积神经网络，它无法自动捕捉输入序列中的位置信息。为了解决这个问题，Transformer 引入了位置编码，将位置信息嵌入输入序列中，以帮助模型理解序列中不同位置的相对顺序。

(4) 残差连接：为了避免深层网络中的梯度消失或梯度爆炸问题，Transformer 模型使用了残差连接。残差连接将输入直接添加到网络的输出中，使网络可以更容易地学习到残差部分，从而提高模型的性能。

(5) 层归一化：为了进一步稳定训练过程，Transformer 模型在每个子层之后都应用了层归一化。层归一化对每个子层的输出进行归一化，使模型在不同层之间更容易进行信息传递和学习。

Transformer 模型通过编码器和解码器之间的交互、注意力机制、多头注意力、位置编码、残差连接和层归一化等技术，在机器翻译等任务中取得了很好的效果。基于注意力机制的其他模型也将 Transformer 应用到不同的机器学习任务中，如自然语言处理和计算机视觉等领域，其基于 Transformer 的变形模型也取得了很好的效果。

5.3.2　Transformer 模型的代码实现

完成 Transformer 模型的编码器和解码器代码后，接下来利用两个类函数来构建整个 Transformer 模型的实现代码。Transformer 模型的实现代码如下：

```
import torch.nn.functional as F
enc_input = torch.LongTensor([[5,2,1,0,0],[1,3,1,4,0]])
dec_input = torch.LongTensor([[5,2,1,0,0],[1,3,1,4,0]])
class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        self.encoder = Encoder()           #编码器
        self.decoder = Decoder()           #解码器
        #最终模型的输出经过 linear 层进行 shape 转换
        self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False)

    def forward(self, enc_inputs, dec_inputs):
        #enc_inputs: [batch_size, src_len] [2,5]
        #dec_inputs: [batch_size, tgt_len] [2,5]
        #enc_outputs: [batch_size, src_len, d_model], [2,5,512]
        #enc_self_attns: [n_layers, batch_size, n_heads, src_len, src_len]
        #经过 Encoder 网络后，输出[batch_size, src_len, d_model] [2,5,512]
        enc_outputs, enc_self_attns = self.encoder(enc_inputs)
        #dec_outputs: [batch_size, tgt_len, d_model] [2,5,512]
        #dec_self_attns: [n_layers, batch_size, n_heads, tgt_len, tgt_len]
        #dec_enc_attn: [n_layers, batch_size, tgt_len, src_len][8,2,5,5]
```

```
        dec_outputs, dec_self_attns, dec_enc_attns = self.decoder(dec_inputs,
enc_inputs, enc_outputs)          #解码器
        #dec_outputs: [batch_size, tgt_len, d_model] [2,5,512]->
        #dec_logits: [batch_size, tgt_len, tgt_vocab_size] [2,5,10]
        dec_logits = self.projection(dec_outputs)        #输出经过一层线性层
        dec_logits = F.log_softmax(dec_logits, dim=-1)   #输出每个预测值的概率
        return (dec_logits.view(-1, dec_logits.size(-1)),
                              enc_self_attns, dec_self_attns, dec_enc_attns)

model = Transformer()                                    #初始化 Transformer 模型
#传递输入数据
model_result,enc_att,dec_att,de_enc_att = model(enc_input,dec_input)
print('model_result',model_result)                       #打印输出数据
print(model_result.shape)                                #打印输出数据形状
```

首先，建立一个名为 Transformer 的类函数，并在初始化部分对编码器和解码器两个类函数进行初始化。此外，还在初始化部分添加了一个线性层函数，用于格式化 Transformer 模型的输出，将输出数据的词嵌入维度转换为解码器输入序列长度。

在实现函数部分，接受两个输入，一个是编码器的输入，另一个是解码器的输入，其输入矩阵维度都是[2,5]。首先，将编码器的输入数据矩阵经过编码器进行编码，其函数最终返回经过编码器的输出，其矩阵维度为[2,5,512]。有了编码器的输出，就可以执行解码器的解码工作了。这里将解码器的输入、编码器的输入及编码器的最终输出传递给解码器。经过解码器的计算后，最终返回经过解码器的输出，其矩阵维度依然是[2,5,512]。

最后，其输出数据还需要经过一次线性变换，将矩阵维度[2,5,512]转换成[2,5,10]，其中数字“10”便是定义的解码器输入数据的序列长度，然后经过一个 Softmax 函数，得到每个单词针对整个输入序列的概率分布。程序就可以使用此概率分布，挑选出针对每个单词概率最大的输出。这样就可以实现输入一个句子，翻译成另外一种语言的句子，机器翻译的任务就完成了。

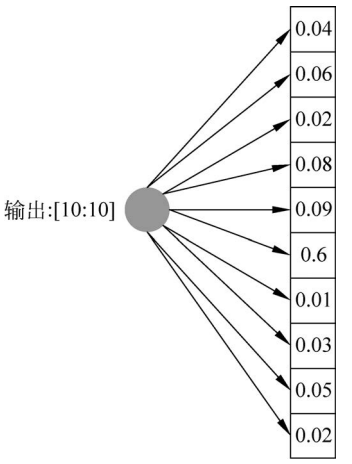

图 5-5 Transformer 模型的输出，每个单词对应 10 个概率分布

需要注意的是，Transformer 模型并不会直接输出需要的单词，而是输出在整个数据集上每个单词的概率。程序代码只需挑选出概率最大的单词，然后把所有单词组合起来，就完成了一个句子的翻译过程。程序最后直接返回模型预测的概率分布，当然这里使用 view 操作，把 Batch Size 的维度与解码器输入句子长度维度合并，因此当前的矩阵维度为[10,10]。第 1 个数字“10”代表 Transformer 模型最终会输出 10 个单词；第 2 个数字“10”代表每个单词在 10 个单词上的概率分布。程序需要挑选最大概率的单词，并且 10 个数字的概率和为“1”，如图 5-5 所示。

最后，初始化一个 Transformer 模型的函数，并输入编码器和解码器的输入矩阵，可以打印出经过 Transformer 模型的输出及输出的矩阵维度，可以看到经过 Transformer 模型后，输出了一个矩阵维度为[10,10]的矩阵，其输出如下：

```
model_result tensor([
        [-2.6741, -3.7803, -3.0414, -3.2264, -2.7376, -2.5425, -1.5795, -1.2638,
-1.8878,-3.2889],
        [-3.0494, -3.5199, -2.7973, -2.9748, -3.6690, -3.0864, -2.1896, -0.7255,
-2.3638,-3.0014],
        [-2.0200, -3.5256, -2.3064, -3.8264, -3.2494, -2.6365, -1.6868, -1.7005,
-1.7162,-2.8362],
        [-1.9222, -3.1514, -2.3846, -3.5230, -2.8151, -2.6279, -1.8144, -1.2476,
-2.5905,-3.4405],
        [-1.9152, -3.1320, -2.4085, -3.4537, -2.8232, -2.7306, -1.8954, -1.2086,
-2.5043,-3.4329],

        [-1.4140, -3.3145, -1.8412, -3.7134, -3.5885, -2.2777, -2.0321, -2.3979,
-2.1502,-2.6755],
        [-1.7039, -2.9270, -2.4014, -3.1103, -3.4213, -1.9536, -2.2433, -1.6777,
-2.6153,-2.4215],
        [-1.4116, -3.2959, -1.7004, -3.7672, -3.5506, -2.3468, -2.2345, -2.3013,
-2.1494,-2.7275],
        [-1.8619, -2.6240, -1.8548, -3.0520, -3.0201, -2.0142, -2.1630, -2.1502,
-2.3939,-2.7580],
        [-1.4741, -2.7903, -1.7404, -3.2648, -2.9819, -2.3859, -2.4127, -1.8542,
-2.7061,-3.2111]],
        grad_fn=<ViewBackward>)
torch.Size([10, 10])
```

5.4　Transformer 模型训练过程

Transformer 模型的训练过程和推理过程在工作方式和数据流向上有些微妙的区别。首先，来看训练过程的工作方式和数据流向。

训练数据由两部分组成：源输入序列，例如机器翻译实例中的英语句子“How are you”，作为编码器的输入；目标输入序列，例如翻译成的中文句子“你好吗”，作为解码器的输入。

Transformer 模型的目标是通过输入和目标序列来学习如何将英文版本的句子翻译成中文版本的句子。以下是 Transformer 模型的训练过程，如图 5-6 所示。

(1) 首先，需要重申的是，Transformer 模型无法直接识别输入的汉语句子或英语句子，因此，无论是编码器的源输入序列，还是解码器的目标输入序列都需要进行单词 ID 的初始化(将每个单词都赋予一个不重复的数字)。这样，编码器的源输入序列和解码器的目标输

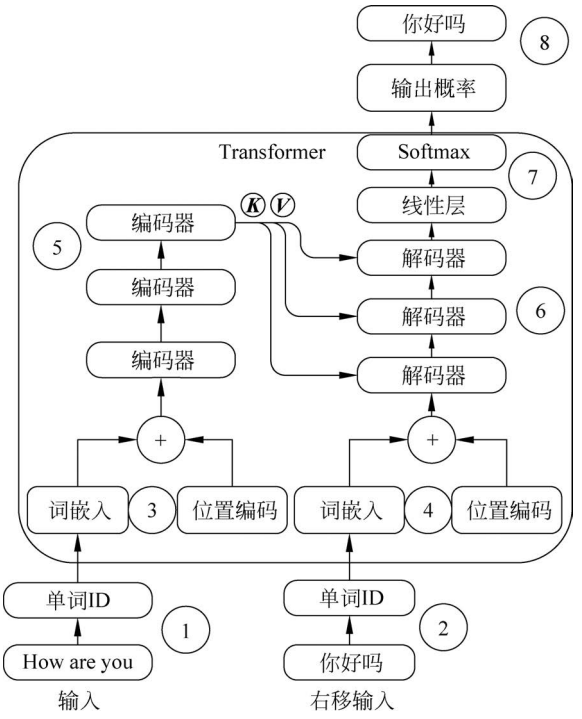

图 5-6　Transformer 模型的训练过程

入序列都初始化了一个数字数据集(第 1 步和第 2 步)。

(2) 接下来,需要对句子经过词嵌入进行编码,再加上位置编码(第 3 步和第 4 步)。

(3) 将经过词嵌入和位置编码相加后的句子序列传递给 Transformer 模型的 6 层编码器进行多头注意力机制的计算(第 5 步)。

(4) 将目标序列传递给 Transformer 模型的 6 层解码器进行多头注意力机制的计算。在解码器-编码器-注意力机制交互层中,其 $\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵来源于编码器的最终输出,而 $\boldsymbol{Q}$ 矩阵则来源于解码器(第 6 步)。当然,在将目标序列传递给解码器之前,需要进行 Sequence Mask 操作,以避免在训练过程中 Transformer 模型提前看到未来的输入序列。

(5) 经过解码器的序列,最后经过一层线性层与 Softmax 计算后便生成了目标序列单词 ID 的权重值(第 7 步)。

(6) 根据单词 ID 的权重值,挑选出最大权重对应的中文文字即可(第 8 步)。

在训练过程中,解码器输出的句子会与真实输入的句子进行 loss 损失的计算,而模型训练的目的是让 loss 越小越好。需要注意的是,在训练过程中,Transformer 模型使用的是教师强制(Teacher Forcing)技术,即在每步中都将真实的目标序列输入解码器中,以帮助模型更快地收敛。在推理过程中,则不再使用 Teacher Forcing,而是使用自回归的方法,逐步生成输出序列。

5.5　Transformer 模型预测过程

Transformer 模型在推理阶段的工作方式与训练阶段略有不同。在推理阶段，只有编码器有输入数据，而解码器则没有输入数据。Transformer 模型的目标是仅从输入序列（如英文句子）中生成目标序列（如中文句子）。因此，Transformer 模型的推理过程类似于循环神经网络模型，其模型循环输出中文版本的单词，并将前一个时间步的输出单词提供给下一个时间步的解码器，直到遇到结束预测标记“END”。

Transformer 模型与循环神经网络模型的不同之处在于，在每个时间步，重新输入解码器的单词是到目前时间步为止生成的所有输出序列，而不仅是最后一个时间步的输出单词，Transformer 模型的预测过程如图 5-7 所示。

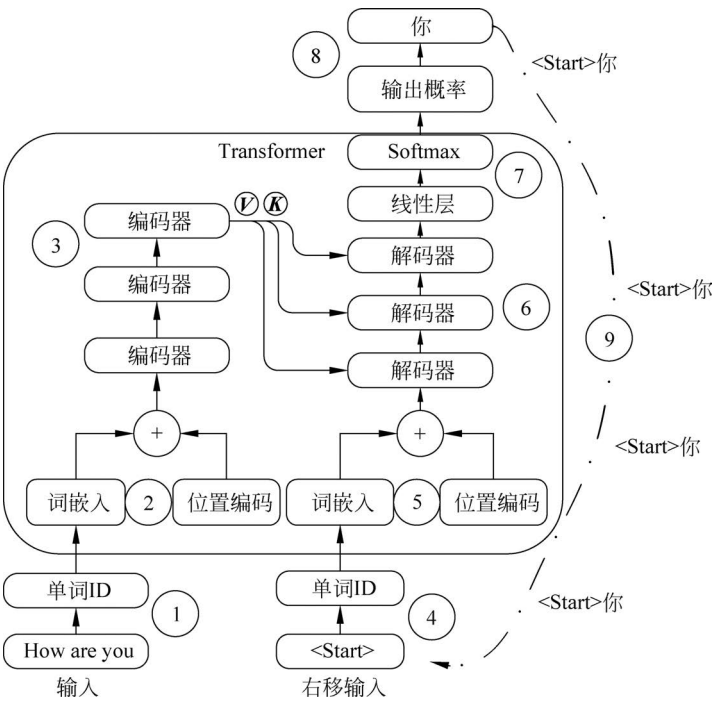

图 5-7　Transformer 模型的预测过程

（1）首先，输入序列数据需要经过初始化处理后，传递给编码器，包括词嵌入编码和位置编码（第 1 步和第 2 步）。

（2）然后输入序列经过 6 层编码器的多头注意力机制计算，输出 $\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵并传递给解码器（第 3 步）。

（3）在推理预测时，并没有目标输入序列，因此，通常会初始化一个“Start”单词，标志着模型可以开始预测了（第 4 步）。

（4）“Start”单词在目标输入序列数据集中也有对应的 ID，因此也需要经过词嵌入编码

和位置编码，然后传递给解码器（第 5 步）。

（5）在解码器-编码器-注意力机制交互层中，解码器接受目标输入序列提供的 $\boldsymbol{Q}$ 矩阵，与编码器最终输出提供的 $\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵进行注意力机制的交互计算（第 6 步）。

（6）解码器的输出还需要经过一个线性层与 Softmax 层，输出模型预测的单词概率分布，根据概率分布挑选出最大概率的单词 ID，然后根据输出的数据集找到预测的最大概率的单词（第 7 步和第 8 步）。

（7）然后把“Start”单词与预测到的第 1 个单词“你”一起传递给第 4 步，以此类推，推理出其他的所有单词（第 9 步）。

例如，首先输入模型“Start”单词，模型预测出单词“你”，然后把单词“Start”和“你”同时传递给解码器，模型预测出单词“好”，接下来把单词“Start”“你”和“好”同时传递给解码器，然后模型预测出单词“吗”，最后把单词“Start”“你”“好”和“吗”同时传递给解码器，模型预测出结束字符“END”，整个模型预测完成，结束预测。需要注意的是，在推理过程中，第 1 步、第 2 步和第 3 步只需执行一次。

需要注意的是，在推理过程中，Transformer 模型使用的是自回归方法，逐步生成输出序列。在每个时间步，模型都会根据到目前为止生成的所有输出序列来预测下一个单词。这种方法可以更好地利用上下文信息，提高模型的预测准确性。

5.6 Transformer 模型 Force Teach

在 Transformer 模型的训练过程中，将目标输入序列直接传递给解码器的方法称为教师强制。这种方法的目的是避免模型在训练过程中陷入错误的循环依赖，从而提高模型的训练效率和准确性。

在推理过程中，Transformer 模型会根据之前预测的单词来预测下一个单词，这种循环机制会导致训练花费更长的时间，同时也会使模型更难训练。因为如果模型在预测第 1 个单词时出错了，则后续的预测就很可能都会出错，这是因为后续的预测都是基于前面预测的结果进行的。

为了解决这个问题，在训练过程中，可以采用 Teacher Forcing 方法，将目标输入序列提前传递给解码器。这样做的好处是，即使模型在预测第 1 个单词时出错了，模型也可以根据正确的第 1 个单词来预测第 2 个单词，从而避免错误的信息传递到后续的预测中。同时，Transformer 模型还可以在不循环的情况下并行输出所有单词，大大地加快了训练速度。这就是 Sequence Mask 的作用。

需要注意的是，虽然 Teacher Forcing 可以提高模型的训练效率和准确性，但是它也有一定的局限性。因为在实际应用中，模型并不一定能够获取正确的输入序列，因此模型在推理过程中可能会出现一些问题，因此，在实际应用中，需要结合具体情况，采用不同的训练策略来提高模型的性能。

5.7　Transformer 模型与 RNN 模型

在 Transformer 模型出现之前，循环神经网络(Recurrent Neural Network，RNN)及其变种长短期记忆网络(Long Short-Term Memory，LSTM)是所有自然语言处理(NLP)应用程序的主要模型架构。RNN 和 LSTM 在处理序列数据方面表现出色，并且被广泛地应用于语音识别、机器翻译、情感分析等各种 NLP 任务中。

5.7.1　RNN 循环神经网络

在 Transformer 模型发布之前，循环神经网络是最先进的顺序数据算法，其模型被应用在苹果的 Siri 和谷歌的语音搜索等应用中。由于其先进的顺序输入模型，循环神经网络是第 1 个可以记住输入训练的算法，这使循环神经网络非常适合涉及顺序数据的机器学习问题，例如时间序列、语音、文本、财务数据、音频、视频、天气等相关机器学习任务。与其他算法相比，循环神经网络可以对序列及其上下文信息理解得更加透彻。由于循环神经网络具有记忆和时间依赖性，所以适用于各种自然语言处理、时间序列分析、语音识别等任务。循环神经网络的模型结构如图 5-8 所示。

RNN 的基本模型结构包括一个循环单元，通常被称为隐藏层或循环层(A)，以及输入层(X_t)和输出层(O_t)。循环层在每个时间步(t)接收输入数据(X_t)和前一个时间步的隐藏状态(V_t-1)，然后输出一个新的隐藏状态(V_t)。这个新的隐藏状态会被传递给下一个时间步($t+1$)，从而使模型能够在处理序列数据时保持记忆。这里把 RNN 模型的结构展开，如图 5-9 所示。

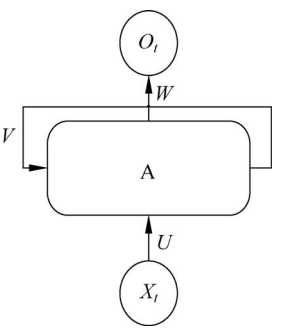

图 5-8　RNN 循环神经网络模型

具体来讲，RNN 循环神经网络模型在每个时间步 t 的计算如下。

(1) 输入：$\boldsymbol{X}_t$(当前时间步的输入数据)。

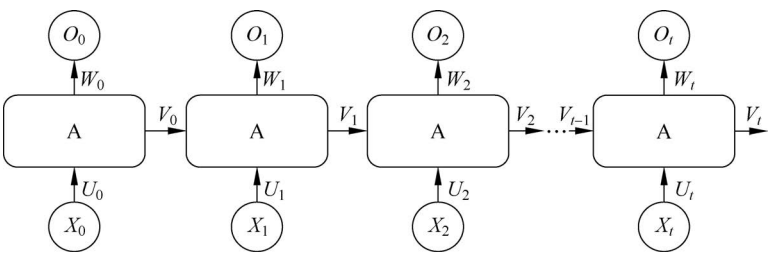

图 5-9　RNN 循环神经网络模型展开图

(2) 隐藏状态：$\boldsymbol{V}_t$(当前时间步的隐藏状态)。

(3) 输出：$\boldsymbol{O}_t$(当前时间步的输出)。

(4) 权重矩阵：$\boldsymbol{U}_t$(输入隐藏状态的权重)、$\boldsymbol{w}_{hh}$(上一个隐藏状态到当前隐藏状态的权重)、$\boldsymbol{W}_t$(隐藏状态到输出的权重)。

RNN 循环神经网络的隐藏状态计算公式如下：

$$\boldsymbol{V}_t = \sigma(\boldsymbol{U}_t \times \boldsymbol{X}_t + \boldsymbol{W}_{hh} \times \boldsymbol{V}_{t-1}) \tag{5-1}$$

其中，sigma(σ)是激活函数(如 tanh 或 ReLU)，$\boldsymbol{V}_{t-1}$ 是上一个时间步的隐藏状态。

输出可以根据隐藏状态的计算公式来推导，计算如下：

$$\boldsymbol{O}_t = \boldsymbol{W}_t \times \boldsymbol{V}_t \tag{5-2}$$

由于 RNN 具有时间先后顺序，因此不需要位置编码，然而，RNN 在处理长序列时会出现梯度消失问题，从而导致难以捕捉长序列中的依赖关系。为了解决这个问题，出现了一些改进的 RNN 结构，如长短时记忆网络和门控循环单元。这些结构具有更复杂的内部结构，能够更好地处理梯度消失问题，具备更长的记忆能力，同时提高了计算效率。

虽然循环神经网络和长短时记忆网络在 NLP 领域取得了巨大成功，但是它们存在一些固有的缺陷，例如，循环神经网络和长短时记忆网络在训练与推理时，不能采用并行计算，这降低了计算效率。此外，循环神经网络和长短时记忆网络在处理长序列时会出现梯度消失问题，导致难以捕捉长序列中的依赖关系。为了解决这些问题，出现了 Transformer 模型。Transformer 模型通过自注意力机制，实现了并行计算，从而大大地提高了计算效率。同时，Transformer 模型在处理长序列时也表现出色，能够更好地捕捉长序列中的依赖关系。

5.7.2 Transformer 模型与 RNN 模型对比

由于循环神经网络模型存在两个限制：处理长句子时，难以捕捉分散得很远的单词之间的长期依赖关系；一次按一个单词顺序处理输入序列，这意味着它在完成时间步长 $t-1$ 的计算之前不能进行时间步长 t 的计算。这会降低训练和推理的速度，影响计算效率。Transformer 模型架构成功地解决了以上两个限制，完全摆脱了循环神经网络的结构，完全依赖于注意力机制，而且 Transformer 模型可以并行处理序列中的所有单词，从而大大地加快了预测与训练速度。

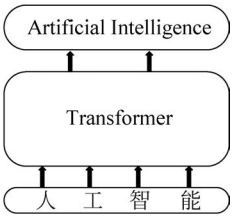

图 5-10 Transformer 模型结构

Transformer 模型结构如图 5-10 所示，在模型进行训练或者推理时，其输入序列可以一次全部传递给 Transformer 模型，并可以进行并行计算，而不用像循环神经网络模型一样，必须使用上一时间步的输出数据来进行当前时间步的计算。一次按一个单词顺序处理输入序列，无法使用并行计算，严重地影响了计算效率。此外，Transformer 模型可以一次性计算长输入序列的注意力机制，很容易捕捉分散得很远的单词之间的长期依赖关系。

需要注意的是，虽然 Transformer 模型在处理长序列时表现出色，但它需要更多的计算资源和内存来处理长序列，因此，在实际应用中需要根据具体任务和数据规模来选择适当的模型架构。Transformer 模型相比于传统的循环神经网络模型，如 RNN、LSTM 和 GRU

等，引入了一些重要的改进，以解决循环神经网络模型存在的一些问题。

（1）并行性：在循环神经网络模型中，由于每个时间步的计算依赖于前一个时间步，所以难以进行并行化处理，而 Transformer 通过注意力机制的引入，可以在每个时间步同时处理整个输入序列，从而提高了计算效率。

（2）梯度消失问题：传统循环神经网络模型容易受到梯度消失或梯度爆炸的困扰，特别是在处理长序列时。Transformer 通过残差连接和层归一化，以及更复杂的注意力机制，能够更好地传播梯度，减轻了梯度问题。

（3）长期依赖关系：循环神经网络模型在处理长期依赖关系时表现不佳，因为随着序列步骤的增加，梯度可能会逐渐减小，导致梯度消失。Transformer 模型引入了注意力机制，可以更好地捕捉长距离的依赖关系。

（4）模型深度：循环神经网络模型难以构建非常深的网络，因为梯度难以传播。在 Transformer 模型中，可以堆叠多个注意力层和前馈神经网络层，构建更深层次的模型，这有助于提高模型的性能。

（5）位置编码：虽然传统的循环神经网络模型具有时间序列，但缺乏输入序列的位置信息。Transformer 模型引入了位置编码，以帮助模型理解输入序列中每个元素的位置，从而更好地处理序列数据。

Transformer 模型的成功在很大程度上归功于其注意力机制和并行性，这使它成为处理序列数据的有力工具，并在自然语言处理领域和其他序列建模任务中取得了显著的突破。不过，需要指出的是，循环神经网络模型仍然在某些任务中有其用武之地，特别是对于需要维持状态的任务，如音乐生成和时间序列预测等。

Transformer 模型用途广泛，可用于大多数 NLP 任务，例如语言模型和文本分类等。Transformer 模型也经常用于机器翻译、文本摘要、问答、推荐系统和语音识别等应用中，但是，Transformer 模型也可以应用到计算机视觉任务中，其中最典型的是谷歌发布的 Vision Transformer 模型，它完全依赖于 Transformer 模型，但是只使用了 Transformer 模型中的编码器部分。如何把 Transformer 模型应用到计算机视觉任务中，这将在后面的章节进行详细介绍。

5.8　本章总结

在前 4 章中，详细地介绍了 Transformer 模型的各个模块，包括输入/输出、位置编码、注意力机制与多头注意力机制、前馈神经网络、残差连接与归一化操作及掩码张量的概念。基于这些模块，可以搭建标准的 Transformer 模型。本章主要介绍如何使用这些模块搭建 Transformer 模型的编码器和解码器，并通过代码实现完整的 Transformer 模型。

在模型搭建完成后，介绍 Transformer 模型的训练过程和预测过程。需要注意的是，预测过程与训练过程存在一些区别。在预测过程中，根据上一时间步的数据来预测当前时间步的数据，而在训练过程中，一次性将所有数据传递给 Transformer 模型，但为了避免模型

看到未来信息，添加了 Sequence Mask 矩阵。

在本章的最后，还简要地介绍了在 NLP 领域中同样被广泛应用的循环神经网络模型。在 Transformer 模型出现之前，循环神经网络模型在 NLP 领域中占据了半壁江山，然而，循环神经网络模型存在一定的限制，而 Transformer 模型成功地解决了这些限制问题，并因此获得了广泛认可。Transformer 模型最初被用于解决 NLP 领域中的翻译任务，随着注意力机制的实用性得到认可，Transformer 模型被广泛地应用于各种 NLP 任务中。此外，基于 Transformer 模型的变种模型也数不胜数。在后续章节中，将介绍 Transformer 模型在其他 NLP 领域中的应用及计算机视觉领域中的应用。