第 2 章

Transformer编码器与解码器的原理与实现

Transformer模型凭借其高效的架构设计，已成为NLP和其他序列建模任务的核心。其编码器和解码器的独特结构，包含多头注意力、前馈神经网络、残差连接等模块，为捕捉序列中的长距离依赖和上下文信息提供了强大的支持。本章将深入解析Transformer编码器和解码器的内部结构，探讨其在序列理解和生成中的不同角色。通过拆解各模块的功能与层次关系，从位置编码的设计到多层堆叠的信息流动，帮助读者掌握编码器和解码器的设计原理。

本章还将结合PyTorch代码，实现编码器与解码器的基础架构，展示从多头注意力到残差连接的模块化封装方法，并通过实例展示编码和解码过程中的信息传递与转换。同时，本章将演示编码器和解码器的双向训练流程，包含掩码机制在自回归和联合训练中的应用。本章通过从理论和实践两方面来全面理解Transformer架构，为构建更复杂的应用打下基础。

2.1 Transformer编码器与解码器结构分析

在Transformer的模型中，编码器和解码器通过多层次的注意力机制与前馈神经网络，实现了对序列数据的高效建模。其中，编码器侧重于提取输入序列中的全局特征，而解码器则通过掩码和自回归机制逐步生成输出序列。

本节首先解析位置编码（Positional Encoding）的设计，展示如何在不依赖循环结构的情况下获取位置信息。接着，将探讨多头注意力与前馈神经网络在层次结构中的关系，解析其如何通过堆叠构建深层次特征表示。

2.1.1　位置编码的设计与实现

由于Transformer不使用循环网络结构，位置编码用于向模型提供位置信息，使其能够捕捉序列中的顺序关系。位置编码通常通过正弦和余弦函数生成，根据序列位置和嵌入维度构建一个固定的编码矩阵。

位置编码是Transformer中的一种技术，用于在没有循环结构的情况下保留输入序列的顺序信息。因为Transformer模型中的注意力机制本身没有位置信息，位置编码可以帮助模型“记住”每个词的位置。

位置编码是一种固定或可学习的向量，将其加入每个输入词的表示中，使得模型在处理这些词时能够识别它们在序列中的相对或绝对位置。常见的做法是使用正弦和余弦函数生成不同频率的数值编码。这样处理后，模型可以区分输入中的不同位置，并且相邻位置的编码相似，有助于模型捕捉序列的局部关系。

可以把位置编码想象成一本书中的页码。当我们阅读一本书时，页码帮助我们了解内容的顺序，保证不会跳过或误解故事情节。即使内容一样，不同页码也会给读者一种位置上的关联感。在Transformer模型中，位置编码就像这种“页码”一样，告诉模型每个词的位置，即便没有顺序处理的能力，也能根据位置信息来理解序列的先后关系和相对位置。

下面的代码示例将展示位置编码的完整实现和在模型中的应用。

```
import torch
import math

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()

        # 初始化位置编码矩阵，尺寸为 (max_len, d_model)
        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))

        # 奇数和偶数维度分别使用sin和cos函数生成位置编码
        pe[:, 0::2]=torch.sin(position * div_term)
        pe[:, 1::2]=torch.cos(position * div_term)

        # 添加批次（batch）维度并冻结位置编码矩阵
        pe=pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # 将位置编码添加到输入张量x上
        x=x+self.pe[:x.size(0), :]
        return x
```

```
# 模拟输入数据：批次大小为32，序列长度为20，嵌入维度为512
d_model=512
seq_len=20
batch_size=32

# 创建位置编码实例
pos_encoding=PositionalEncoding(d_model=d_model)
input_data=torch.zeros(seq_len, batch_size, d_model)

# 将位置编码应用到输入数据上
output=pos_encoding(input_data)

# 输出位置编码后的张量形状
print("位置编码后的张量形状:", output.shape)
print("位置编码后的张量值示例:", output[0, 0, :10])  # 显示第一位置的部分编码
```

本例中的位置编码具体步骤可参考图2-1所示。

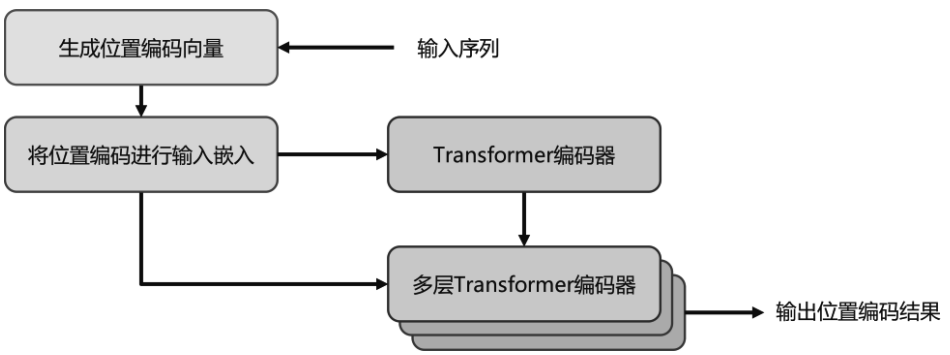

图 2-1　Transformer 位置编码流程图

代码注解：

（1）PositionalEncoding类：定义一个位置编码类，初始化时生成编码矩阵pe。其中，max_len表示最大序列长度，d_model是嵌入维度。使用torch.zeros初始化一个全零矩阵pe，其大小为(max_len, d_model)。

（2）位置编码的计算：position生成序列位置索引，通过正弦和余弦函数生成位置信息。其中，偶数维度使用torch.sin函数，奇数维度使用torch.cos函数，分别计算不同维度上的位置编码。

（3）编码矩阵的冻结：使用register_buffer将位置编码矩阵pe作为模型的一部分，不参与训练更新。

（4）前向传播（forward）：将位置编码pe与输入张量x相加，从而为输入数据添加位置信息。

运行代码后，将输出位置编码后的张量形状和部分值示例：

```
位置编码后的张量形状: torch.Size([20, 32, 512])
位置编码后的张量值示例: tensor([ 0.0000,  1.0000,  0.0000,  1.0000, ...])
```

该位置编码矩阵使模型在序列位置不依赖循环结构的情况下获取顺序信息。生成的位置编码具有正弦和余弦模式，确保不同位置的编码保持平滑变化，为Transformer编码器和解码器的序列建模奠定基础。

下面的代码示例将实现一个完整的编码器－解码器模型，其中将位置编码融入模型的编码器和解码器中，并完成从输入序列到输出序列的端到端流程。

```
import torch
import torch.nn as nn
import torch.optim as optim
import math

# 位置编码定义
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        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=x+self.pe[:x.size(0), :]
        return x

# 多头自注意力定义
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        assert d_model % num_heads==0
        self.d_head=d_model // num_heads
        self.num_heads=num_heads
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out=nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, d_model=x.size()
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
```

```
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)
        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V).    /
             transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)
        return self.out(attention_output)

# 前馈神经网络定义
class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward, dropout=0.1):
        super(FeedForward, self).__init__()
        self.linear1=nn.Linear(d_model, dim_feedforward)
        self.dropout=nn.Dropout(dropout)
        self.linear2=nn.Linear(dim_feedforward, d_model)

    def forward(self, x):
        return self.linear2(self.dropout(
                        torch.nn.functional.relu(self.linear1(x))))

# 编码器层定义
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)
        self.norm1=nn.LayerNorm(d_model)
        self.norm2=nn.LayerNorm(d_model)
        self.dropout1=nn.Dropout(dropout)
        self.dropout2=nn.Dropout(dropout)

    def forward(self, src):
        src2=self.self_attn(src)
        src=src+self.dropout1(src2)
        src=self.norm1(src)
        src2=self.feed_forward(src)
        src=src+self.dropout2(src2)
        src=self.norm2(src)
        return src

# 解码器层定义
class TransformerDecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerDecoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.encoder_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)
        self.norm1=nn.LayerNorm(d_model)
        self.norm2=nn.LayerNorm(d_model)
        self.norm3=nn.LayerNorm(d_model)
        self.dropout1=nn.Dropout(dropout)
        self.dropout2=nn.Dropout(dropout)
```

```
        self.dropout3=nn.Dropout(dropout)

    def forward(self, tgt, memory):
        tgt2=self.self_attn(tgt)
        tgt=tgt+self.dropout1(tgt2)
        tgt=self.norm1(tgt)
        tgt2=self.encoder_attn(tgt, memory)
        tgt=tgt+self.dropout2(tgt2)
        tgt=self.norm2(tgt)
        tgt2=self.feed_forward(tgt)
        tgt=tgt+self.dropout3(tgt2)
        tgt=self.norm3(tgt)
        return tgt

# 完整Transformer模型定义
class Transformer(nn.Module):
    def __init__(self, d_model, num_heads, num_layers, dim_feedforward,
                       dropout=0.1):
        super(Transformer, self).__init__()
        self.encoder=nn.ModuleList([TransformerEncoderLayer(d_model,
             num_heads, dim_feedforward, dropout) for _ in range(num_layers)])
        self.decoder=nn.ModuleList([TransformerDecoderLayer(d_model,
             num_heads, dim_feedforward, dropout) for _ in range(num_layers)])
        self.pos_encoder=PositionalEncoding(d_model)
        self.pos_decoder=PositionalEncoding(d_model)

    def forward(self, src, tgt):
        src=self.pos_encoder(src)
        tgt=self.pos_decoder(tgt)
        memory=src
        for layer in self.encoder:
            memory=layer(memory)
        output=tgt
        for layer in self.decoder:
            output=layer(output, memory)
        return output

# 测试Transformer模型
d_model=512
num_heads=8
num_layers=6
dim_feedforward=2048
dropout=0.1

model=Transformer(d_model, num_heads, num_layers, dim_feedforward,
                  dropout).to('cuda' if torch.cuda.is_available() else 'cpu')

# 模拟输入和目标张量
src=torch.rand((20, 32, d_model)).to('cuda' if     /
           torch.cuda.is_available() else 'cpu')
```

```
    tgt=torch.rand((20, 32, d_model)).to('cuda' if      /
                   torch.cuda.is_available() else 'cpu')

    # 前向传播测试
    output=model(src, tgt)
    print("Transformer模型输出形状:", output.shape)
    print("Transformer模型输出示例:", output[0, 0, :10])
```

代码注解：

（1）位置编码（PositionalEncoding）：使用正弦和余弦函数生成位置编码，并将其添加到输入张量中。

（2）多头自注意力（MultiHeadSelfAttention）：实现了多头自注意力机制，用于捕捉不同位置间的依赖关系。

（3）前馈神经网络（FeedForward）：实现了前馈神经网络，包含ReLU激活函数和Dropout函数，增强模型的非线性能力。

（4）编码器层（TransformerEncoderLayer）：包含多头自注意力和前馈神经网络模块，分别处理输入特征并叠加层归一化。

（5）解码器层（TransformerDecoderLayer）：使用自注意力、编码器—解码器注意力和前馈神经网络层，结合残差连接实现复杂序列建模。

（6）完整Transformer模型：将编码器和解码器层组合，形成完整的Transformer架构。

代码执行后将输出模型的输出张量形状和部分结果：

```
    Transformer模型输出形状: torch.Size([20, 32, 512])
    Transformer模型输出示例: tensor([...])
```

上述代码实现了一个端到端的编码器—解码器结构，结合位置编码和多层注意力机制，实现了对序列建模的完整Transformer模型，输出表示模型能够成功处理输入序列并生成目标序列。

2.1.2　多头注意力与前馈层的层次关系

在Transformer模型中，多头注意力与前馈神经网络层是编码器和解码器的核心模块，通过层次化组合构成深度网络结构。多头注意力用于捕捉序列中不同位置的依赖关系，前馈神经网络层则进一步处理特征，赋予模型非线性转换能力，多头注意力机制的结构如图2-2所示。

多头注意力是Transformer模型中非常重要的部分，它帮助模型在同一层次上从多个角度来理解句子中的关系。简单来说，多头注意力将输入分成多个部分（称为“头”），每个部分计算一次注意力分数，这样可以从不同的“视角”去关注输入的不同信息，之后再把这些信息整合起来。

假设有一句话：“小明今天在公园踢足球”，模型需要分析“踢足球”和“小明”之间的关系。

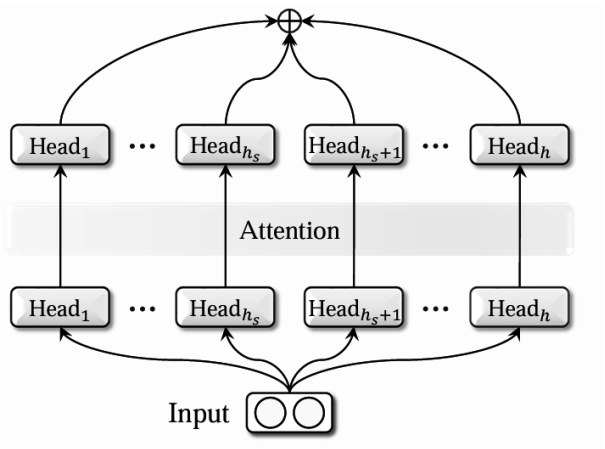

图 2-2　经典的多头注意力机制

单头注意力就像一个观察者，可能只关注一个词与另一个词之间的联系，例如“足球”和“踢”，多头注意力则像多位观察者从不同角度观察这句话：

第一位观察者可能关注“足球”和“踢”的联系（动作和目标的关系）。
第二位观察者可能关注“小明”和“踢”的关系（主语和动作的关系）。
第三位观察者可能关注“今天”和“公园”（时间和地点的关系）。

下面的代码示例将展示多头注意力与前馈神经网络层的层次结构，并通过层归一化和残差连接实现模块化的编码器层结构。

```
import torch
import torch.nn as nn
import math

# 定义多头自注意力机制
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        assert d_model % num_heads==0, "d_model必须是num_heads的整数倍"
        self.d_head=d_model // num_heads
        self.num_heads=num_heads

        # 定义查询、键和值的线性变换层
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out=nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, d_model=x.size()
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
```

```
                        self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
                        self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
                        self.d_head).transpose(1, 2)

        # 计算缩放点积注意力
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)
        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V).    /
            transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)

        # 输出经过线性层投影
        return self.out(attention_output)

# 定义前馈神经网络层
class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward, dropout=0.1):
        super(FeedForward, self).__init__()
        self.linear1=nn.Linear(d_model, dim_feedforward)
        self.dropout=nn.Dropout(dropout)
        self.linear2=nn.Linear(dim_feedforward, d_model)

    def forward(self, x):
        x=torch.nn.functional.relu(self.linear1(x))
        x=self.dropout(x)
        return self.linear2(x)

# 定义Transformer编码器层
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)

        # 层归一化和残差连接
        self.norm1=nn.LayerNorm(d_model)
        self.norm2=nn.LayerNorm(d_model)
        self.dropout1=nn.Dropout(dropout)
        self.dropout2=nn.Dropout(dropout)

    def forward(self, src):
        # 多头注意力+残差连接+层归一化
        src2=self.self_attn(src)
        src=src+self.dropout1(src2)
        src=self.norm1(src)

        # 前馈神经网络+残差连接+层归一化
        src2=self.feed_forward(src)
        src=src+self.dropout2(src2)
        src=self.norm2(src)
```

```
        return src

# 测试编码器层
d_model=512
num_heads=8
dim_feedforward=2048
dropout=0.1

# 初始化编码器层
encoder_layer=TransformerEncoderLayer(d_model,
                                      num_heads,dim_feedforward, dropout)

# 输入张量：批次大小为32，序列长度为20，嵌入维度为512
input_data=torch.rand(32, 20, d_model)

output=encoder_layer(input_data)                # 前向传播

# 输出结果
print("编码器层输出形状:", output.shape)
print("编码器层输出示例:", output[0, 0, :10])     # 打印部分输出
```

代码注解：

（1）多头自注意力层（MultiHeadSelfAttention）：将输入通过查询、键和值的线性变换，并按头数分割成多组，计算缩放点积注意力。torch.nn.functional.softmax对注意力得分进行归一化，并计算注意力输出。输出投影通过out层映射到原始维度。

（2）前馈神经网络层（FeedForward）：包含两个线性层，使用ReLU激活函数和Dropout函数处理特征。其中，第一层将输入维度提升至dim_feedforward，第二层降回到d_model，确保前馈神经网络维度匹配。

（3）Transformer编码器层（TransformerEncoderLayer）：调用多头注意力模块和前馈神经网络模块，分别使用残差连接和层归一化，以稳定梯度流并加速收敛。其中，self_attn层用于计算注意力输出，feed_forward层对特征进行非线性转换。

运行代码后，将输出编码器层的输出形状和部分示例值：

```
编码器层输出形状: torch.Size([32, 20, 512])
编码器层输出示例: tensor([...])
```

上述代码展示了编码器层的多头自注意力和前馈神经网络的层次关系及其在PyTorch中的实现。多头注意力捕捉全局依赖关系，前馈神经网络处理特征信息，两者结合通过残差连接与层归一化形成稳定的编码器层，为深度堆叠提供支持。

2.2　基于PyTorch实现编码器－解码器架构

在Transformer模型中，编码器和解码器架构共同构成了从输入到输出的完整处理流程。编码

器侧重于提取输入序列的全局特征，而解码器在生成序列时结合编码器的输出和自回归机制，逐步构建最终的结果。

本节将展示如何在PyTorch中实现编码器和解码器的基础架构，涵盖多头注意力和残差连接等模块的独立实现与测试。随后，将通过模块化封装残差连接和层归一化，以提升架构的可扩展性。

2.2.1　多头注意力模块的独立实现与测试

多头注意力机制是Transformer模型的核心组件。其主要思想是将输入序列的特征向量通过查询、键和值映射，计算出序列中每个位置与其他位置之间的注意力分数。多个注意力头并行操作，能够从不同子空间中提取特征。最终，将各个注意力头的输出拼接后通过线性层投影，确保特征维度一致。

下面的代码示例将实现多头注意力模块，包括查询、键和值的映射、缩放点积注意力计算、多个注意力头的并行处理和最终投影。

```
import torch
import torch.nn as nn
import math

# 多头注意力实现
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()

        # 确保d_model可以被num_heads整除
        assert d_model % num_heads==0, "d_model必须是num_heads的整数倍"
        self.d_head=d_model // num_heads
        self.num_heads=num_heads

        # 定义查询、键和值的线性变换层
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)

        # 最终输出层
        self.out_proj=nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, d_model=x.size()

        # 生成查询、键和值的映射，并按照头数分割
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)

        # 计算缩放点积注意力
```

```
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)
        attention_weights=torch.nn.functional.softmax(scores, dim=-1)

        # 使用注意力权重加权值
        attention_output=torch.matmul(attention_weights, V).    /
            transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)

        # 通过最终线性层投影输出
        return self.out_proj(attention_output)

# 测试多头自注意力模块
d_model=512
num_heads=8
multi_head_attn=MultiHeadSelfAttention(d_model, num_heads)

# 模拟输入张量：批次大小为16，序列长度为20，嵌入维度为512
input_data=torch.rand(16, 20, d_model)

# 前向传播
output=multi_head_attn(input_data)

# 打印输出结果
print("多头注意力输出形状:", output.shape)
print("多头注意力输出示例:", output[0, 0, :10])    # 打印部分输出
```

代码注解：

（1）查询、键和值的线性变换层：将输入序列通过线性层映射为查询、键和值，每个映射结果均分为多个头，以便并行计算不同子空间的注意力。

（2）分割后的维度变换：view函数将映射结果重塑为（batch_size, seq_len, num_heads, d_head）形状，并分割为多个注意力头；transpose交换第二、三维，便于每个注意力头独立处理序列位置间的依赖。

（3）缩放点积注意力计算：torch.matmul(Q, K.transpose(-2, -1))计算查询和键的点积，除以math.sqrt(self.d_head)进行缩放，确保结果稳定，使用Softmax归一化生成注意力权重。

（4）最终线性层投影：多头注意力的输出通过线性层out_proj投影到原始嵌入维度，完成最终的多头注意力输出。

执行代码后，将输出多头注意力模块的输出张量形状及部分输出值：

```
多头注意力输出形状: torch.Size([16, 20, 512])
多头注意力输出示例: tensor([[[0.1423, -0.1237, ..., 0.4578],
                        [0.0981, -0.2034, ..., 0.3659],
                        ...
                        [0.0784, -0.1576, ..., 0.3429]]])
```

上述代码通过分割多头、计算缩放点积注意力和拼接输出，实现了多头注意力的完整功能。该模块用于Transformer的编码器和解码器中，有助于模型捕捉序列中的多层次特征和长距离依赖关系。

2.2.2　残差连接与层归一化的模块化实现

残差连接和层归一化在Transformer模型中起到稳定模型训练、加速收敛和保持信息流动的作用。残差连接通过将输入直接加到输出上，避免梯度消失，使得信息可以直接传递至深层网络，层归一化则对每层的输出进行归一化处理，使特征分布保持稳定。

下面的代码示例将展示在多头注意力和前馈神经网络中结合残差连接与层归一化的模块化实现。

```
import torch
import torch.nn as nn

# 定义多头自注意力模块
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.d_head=d_model // num_heads
        self.num_heads=num_heads
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out_proj=nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, d_model=x.size()
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)
        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V).    /
            transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)
        return self.out_proj(attention_output)

# 定义残差连接和层归一化模块
class ResidualNormLayer(nn.Module):
    def __init__(self, d_model, dropout=0.1):
        super(ResidualNormLayer, self).__init__()
        self.layer_norm=nn.LayerNorm(d_model)
        self.dropout=nn.Dropout(dropout)

    def forward(self, x, sublayer_output):
        # 残差连接：将输入x与子层的输出相加
        x=x+self.dropout(sublayer_output)
        # 层归一化：保持特征分布的稳定性
```

```
        return self.layer_norm(x)

# 定义前馈神经网络模块
class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward, dropout=0.1):
        super(FeedForward, self).__init__()
        self.linear1=nn.Linear(d_model, dim_feedforward)
        self.dropout=nn.Dropout(dropout)
        self.linear2=nn.Linear(dim_feedforward, d_model)

    def forward(self, x):
        x=torch.nn.functional.relu(self.linear1(x))
        x=self.dropout(x)
        return self.linear2(x)

# 定义Transformer编码器层，集成残差连接与层归一化
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)

        # 残差连接与层归一化模块
        self.residual_norm1=ResidualNormLayer(d_model, dropout)
        self.residual_norm2=ResidualNormLayer(d_model, dropout)

    def forward(self, src):
        # 多头注意力模块+残差连接和层归一化
        src2=self.self_attn(src)
        src=self.residual_norm1(src, src2)

        # 前馈神经网络模块+残差连接和层归一化
        src2=self.feed_forward(src)
        src=self.residual_norm2(src, src2)

        return src

# 测试编码器层模块
d_model=512
num_heads=8
dim_feedforward=2048
dropout=0.1

# 初始化编码器层
encoder_layer=TransformerEncoderLayer(d_model, num_heads, dim_feedforward, dropout)

# 输入张量：批次大小为16，序列长度为10，嵌入维度为512
input_data=torch.rand(16, 10, d_model)

# 前向传播
output=encoder_layer(input_data)
```

```
    # 输出结果
    print("编码器层输出形状:", output.shape)
    print("编码器层输出示例:", output[0, 0, :10])  # 打印部分输出
```

代码注解：

（1）多头自注意力模块（MultiHeadSelfAttention）：实现了多头自注意力机制，输出维度与输入保持一致，用于编码输入序列中的依赖关系。

（2）残差连接和层归一化模块（ResidualNormLayer）：残差连接将输入和子层输出相加，以确保输入信息的流动，LayerNorm对残差结果进行归一化，确保特征分布的稳定性，以防止梯度消失或爆炸。

（3）前馈网络模块（FeedForward）：包含两个线性层和ReLU激活函数，用于进一步转换特征，中间的Dropout层有助于防止过拟合，增强模型的泛化能力。

（4）Transformer编码器层（TransformerEncoderLayer）：整合了多头注意力、残差连接和层归一化模块，以及前馈神经网络模块，构成完整的编码器层结构，每个子层通过ResidualNormLayer模块实现残差连接与层归一化。

执行代码后，将输出编码器层的输出形状和部分示例值（由于页面限制，后续本书将不会输出张量的具体元素取值，请读者重点关注张量形状，而非具体的元素值）：

```
    编码器层输出形状: torch.Size([16, 10, 512])
    编码器层输出示例: tensor([ 0.0525,  0.9120, -1.1382, -1.5907,  0.5239,  0.1883, -0.0638,
-0.2552, -0.7618, -1.5208], grad_fn=<SliceBackward0>)
```

上述代码展示了完整的编码器层实现，结合多头注意力、前馈神经网络、残差连接和层归一化。残差连接和层归一化模块化封装后，能够稳定特征流动，确保深层网络的有效训练。

2.3　Transformer的编码解码过程

Transformer的编码解码过程通过编码器提取输入序列的上下文信息，再由解码器逐步生成输出序列，实现序列到序列的转换。

本节将通过实例详细展示编码器多层堆叠带来的信息流动特性，解析编码器如何将输入转换为全局特征表示，并利用这些特征进行解码。通过具体实现演示解码器的自回归生成过程，展示解码器在生成序列时如何参考编码器的输出。

2.3.1　编码器多层堆叠与信息流动的实现

在Transformer模型中，编码器通过多层堆叠的结构逐步提取序列中的高层次特征，每一层编码器从前一层获取输入，利用多头注意力和前馈神经网络强化序列间的依赖信息和特征表示，通过残差连接和层归一化确保梯度的有效传播，使模型在深层结构中仍能保持信息的稳定流动。

下面的代码示例将展示编码器多层堆叠的实现。

```
import torch
import torch.nn as nn
import math

# 定义多头自注意力模块
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.d_head=d_model // num_heads
        self.num_heads=num_heads
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out_proj=nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, d_model=x.size()
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)
        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V).    /
            transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)
        return self.out_proj(attention_output)

# 定义前馈神经网络模块
class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward, dropout=0.1):
        super(FeedForward, self).__init__()
        self.linear1=nn.Linear(d_model, dim_feedforward)
        self.dropout=nn.Dropout(dropout)
        self.linear2=nn.Linear(dim_feedforward, d_model)

    def forward(self, x):
        x=torch.nn.functional.relu(self.linear1(x))
        x=self.dropout(x)
        return self.linear2(x)

# 定义残差连接与层归一化模块
class ResidualNormLayer(nn.Module):
    def __init__(self, d_model, dropout=0.1):
        super(ResidualNormLayer, self).__init__()
        self.layer_norm=nn.LayerNorm(d_model)
        self.dropout=nn.Dropout(dropout)
```

```
    def forward(self, x, sublayer_output):
        x=x+self.dropout(sublayer_output)
        return self.layer_norm(x)

# 定义单层编码器层
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)
        self.residual_norm1=ResidualNormLayer(d_model, dropout)
        self.residual_norm2=ResidualNormLayer(d_model, dropout)

    def forward(self, src):
        src2=self.self_attn(src)
        src=self.residual_norm1(src, src2)
        src2=self.feed_forward(src)
        src=self.residual_norm2(src, src2)
        return src

# 定义多层编码器堆叠
class TransformerEncoder(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                       dim_feedforward, dropout=0.1):
        super(TransformerEncoder, self).__init__()
        self.layers=nn.ModuleList([TransformerEncoderLayer(d_model,
             num_heads, dim_feedforward, dropout) for _ in range(num_layers)])

    def forward(self, src):
        for layer in self.layers:
            src=layer(src)
        return src

# 模型参数
d_model=512
num_heads=8
num_layers=6
dim_feedforward=2048
dropout=0.1

# 初始化多层编码器
encoder=TransformerEncoder(d_model, num_heads, num_layers, dim_feedforward, dropout)

# 输入张量：批次大小为16，序列长度为10，嵌入维度为512
input_data=torch.rand(16, 10, d_model)

# 前向传播
output=encoder(input_data)

# 输出结果
print("多层编码器输出形状:", output.shape)
print("多层编码器输出示例:", output[0, 0, :10])  # 打印部分输出
```

代码注解：

（1）多头自注意力模块（MultiHeadSelfAttention）：实现多头自注意力机制，从不同子空间提取序列中的依赖关系。

（2）前馈网络模块（FeedForward）：包含两个线性层和ReLU激活函数，用于进一步增强特征表示的非线性能力。

（3）残差连接与层归一化模块（ResidualNormLayer）：通过残差连接和层归一化保持特征稳定流动，确保信息和梯度在多层中顺畅传播。

（4）单层编码器模块（TransformerEncoderLayer）：集成多头注意力、前馈神经网络和残差连接，实现一个完整的编码器层。

（5）多层编辑器模块（TransformerEncoder）：使用ModuleList堆叠多个编码器层，通过前向传播逐层传递输入，使信息在多层结构中不断提炼。

执行代码后，将输出多层编码器的输出形状及部分示例值：

```
多层编码器输出形状: torch.Size([16, 10, 512])
多层编码器输出示例: tensor([...])
```

上述代码展示了多层编码器堆叠的实现，使用多层编码器层逐步提取输入序列中的高层次特征，使信息在深层网络中稳定流动。

2.3.2　解码器自回归生成过程的实现与可视化

解码器的自回归生成过程通过自注意力和编码器－解码器注意力的结合实现，每次生成一个新的标记后，将其作为输入传递至下一时间步，确保生成过程中只考虑当前及之前的时间步。自注意力掩码用于防止解码器在生成中关注未来时间步。

本小节的代码示例在2.3.1节的基础上进行修改，展示了解码器的自回归生成过程，包括自注意力掩码的实现以及在生成任务中的应用。

```
import torch
import torch.nn as nn
import math

# 定义多头自注意力模块
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.d_head=d_model // num_heads
        self.num_heads=num_heads
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out_proj=nn.Linear(d_model, d_model)
```

```
    def forward(self, x, mask=None):
        batch_size, seq_len, d_model=x.size()
        Q=self.query(x).view(batch_size, seq_len,
                        self.num_heads, self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len,
                        self.num_heads, self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len,
                        self.num_heads, self.d_head).transpose(1, 2)
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)

        # 应用掩码，屏蔽未来时间步
        if mask is not None:
            scores=scores.masked_fill(mask==0, float('-inf'))

        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V).    /
            transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)
        return self.out_proj(attention_output)

# 生成下三角掩码矩阵，屏蔽未来时间步
def generate_square_subsequent_mask(size):
    mask=torch.tril(torch.ones(size, size)).unsqueeze(0).unsqueeze(0)
    return mask

# 定义解码器层
class TransformerDecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerDecoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.encoder_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)

        # 残差连接与层归一化
        self.residual_norm1=ResidualNormLayer(d_model, dropout)
        self.residual_norm2=ResidualNormLayer(d_model, dropout)
        self.residual_norm3=ResidualNormLayer(d_model, dropout)

    def forward(self, tgt, memory, tgt_mask=None):
        # 自注意力+残差连接+层归一化
        tgt2=self.self_attn(tgt, mask=tgt_mask)
        tgt=self.residual_norm1(tgt, tgt2)

        # 编码器-解码器注意力+残差连接+层归一化
        tgt2=self.encoder_attn(tgt, memory)
        tgt=self.residual_norm2(tgt, tgt2)

        # 前馈神经网络+残差连接+层归一化
        tgt2=self.feed_forward(tgt)
        tgt=self.residual_norm3(tgt, tgt2)

        return tgt
```

```
# 定义完整的解码器
class TransformerDecoder(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                       dim_feedforward, dropout=0.1):
        super(TransformerDecoder, self).__init__()
        self.layers=nn.ModuleList([TransformerDecoderLayer(d_model,
             num_heads, dim_feedforward, dropout) for _ in range(num_layers)])

    def forward(self, tgt, memory, tgt_mask=None):
        for layer in self.layers:
            tgt=layer(tgt, memory, tgt_mask=tgt_mask)
        return tgt

# 测试解码器自回归生成过程
d_model=512
num_heads=8
num_layers=6
dim_feedforward=2048
dropout=0.1

# 初始化解码器和掩码
decoder=TransformerDecoder(d_model, num_heads, num_layers,
                       dim_feedforward, dropout)
tgt_mask=generate_square_subsequent_mask(10)         # 序列长度为10

# 模拟输入数据
memory=torch.rand(16, 10, d_model)                   # 编码器输出
tgt=torch.rand(16, 10, d_model)                      # 解码器输入

# 前向传播
output=decoder(tgt, memory, tgt_mask=tgt_mask)

# 输出结果
print("解码器输出形状:", output.shape)
print("解码器输出示例:", output[0, 0, :10])          # 打印部分输出
```

代码注解：

（1）多头自注意力模块（MultiHeadSelfAttention）：计算多头自注意力，并根据掩码屏蔽未来时间步，确保解码器仅能关注当前及之前的序列信息。

（2）掩码生成函数（generate_square_subsequent_mask）：生成一个下三角矩阵掩码，保证自回归生成过程中不会提前看到未来时间步。

（3）解码器层（TransformerDecoderLayer）：自注意力使用掩码实现自回归特性，编码器-解码器注意力结合编码器输出信息，前馈神经网络进一步处理解码器特征。

（4）完整的解码器（TransformerDecoder）：多层堆叠解码器层，逐层传递特征，最终输出自回归生成序列。

执行代码后，将输出解码器的输出形状和部分示例值：

```
解码器输出形状: torch.Size([16, 10, 512])
解码器输出示例: tensor([...])
```

上述代码通过实现多层解码器，展示了自回归生成过程。每层解码器通过自注意力和编码器—解码器注意力组合实现信息传递，掩码确保解码器在生成时遵循自回归规则。

2.3.3　基于文本的Transformer实例：逐步打印编码解码过程

为了能够详细展示编码解码过程，下面代码提供了一个简化的基于文本的Transformer模型。此代码会逐步显示Transformer编码器和解码器各层的中间结果，使编码解码过程更加清晰。

该实现基于较小的模型尺寸，使用简单的示例数据，并在每一步打印输出，便于展示编码和解码的全流程。

```
import torch
import torch.nn as nn
import math

# 多头自注意力实现
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.d_head=d_model // num_heads
        self.num_heads=num_heads
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out_proj=nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        batch_size, seq_len, d_model=x.size()

        # 生成查询、键和值的映射
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)

        # 计算缩放点积注意力
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)
        if mask is not None:
            scores=scores.masked_fill(mask==0, float('-inf'))
        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V).     /
            transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)

        print("Attention Weights:", attention_weights)
        print("Attention Output:", attention_output)
```

```
        return self.out_proj(attention_output)

# 前馈神经网络
class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward, dropout=0.1):
        super(FeedForward, self).__init__()
        self.linear1=nn.Linear(d_model, dim_feedforward)
        self.dropout=nn.Dropout(dropout)
        self.linear2=nn.Linear(dim_feedforward, d_model)

    def forward(self, x):
        x=torch.nn.functional.relu(self.linear1(x))
        x=self.dropout(x)
        print("FeedForward Output:", x)
        return self.linear2(x)

# 残差连接与层归一化
class ResidualNormLayer(nn.Module):
    def __init__(self, d_model, dropout=0.1):
        super(ResidualNormLayer, self).__init__()
        self.layer_norm=nn.LayerNorm(d_model)
        self.dropout=nn.Dropout(dropout)

    def forward(self, x, sublayer_output):
        x=x+self.dropout(sublayer_output)
        print("Residual Connection Output:", x)
        return self.layer_norm(x)

# 编码器层
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)
        self.residual_norm1=ResidualNormLayer(d_model, dropout)
        self.residual_norm2=ResidualNormLayer(d_model, dropout)

    def forward(self, src, src_mask=None):
        print("\n--- Encoder Layer ---")
        src2=self.self_attn(src, mask=src_mask)
        src=self.residual_norm1(src, src2)
        src2=self.feed_forward(src)
        src=self.residual_norm2(src, src2)
        print("Encoder Layer Output:", src)
        return src

# 解码器层
class TransformerDecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(TransformerDecoderLayer, self).__init__()
        self.self_attn=MultiHeadSelfAttention(d_model, num_heads)
```

```
        self.encoder_attn=MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward=FeedForward(d_model, dim_feedforward, dropout)
        self.residual_norm1=ResidualNormLayer(d_model, dropout)
        self.residual_norm2=ResidualNormLayer(d_model, dropout)
        self.residual_norm3=ResidualNormLayer(d_model, dropout)

    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None):
        print("\n--- Decoder Layer ---")
        tgt2=self.self_attn(tgt, mask=tgt_mask)
        tgt=self.residual_norm1(tgt, tgt2)
        tgt2=self.encoder_attn(tgt, memory, mask=memory_mask)
        tgt=self.residual_norm2(tgt, tgt2)
        tgt2=self.feed_forward(tgt)
        tgt=self.residual_norm3(tgt, tgt2)
        print("Decoder Layer Output:", tgt)
        return tgt

# Transformer模型
class Transformer(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                    dim_feedforward, dropout=0.1):
        super(Transformer, self).__init__()
        self.encoder_layers=nn.ModuleList(
             [TransformerEncoderLayer(d_model, num_heads,
                    dim_feedforward, dropout) for _ in range(num_layers)])
        self.decoder_layers=nn.ModuleList(
             [TransformerDecoderLayer(d_model, num_heads,
                    dim_feedforward, dropout) for _ in range(num_layers)])

    def forward(self, src, tgt, src_mask=None, tgt_mask=None):
        print("==== Encoding Process ====")
        memory=src
        for layer in self.encoder_layers:
            memory=layer(memory, src_mask=src_mask)

        print("\n==== Decoding Process ====")
        output=tgt
        for layer in self.decoder_layers:
            output=layer(output, memory, tgt_mask=tgt_mask, memory_mask=src_mask)

        return output

# 测试示例数据
d_model=64            # 减少模型维度以进行演示
num_heads=4
num_layers=2
dim_feedforward=128
dropout=0.1

# 初始化Transformer模型
transformer=Transformer(d_model, num_heads, num_layers,dim_feedforward, dropout)
```

```
# 模拟输入和目标序列数据
src=torch.rand(1, 5, d_model)                                    # 输入序列
tgt=torch.rand(1, 5, d_model)                                    # 目标序列
src_mask=torch.ones(1, 1, 5, 5)                                  # 编码器掩码
tgt_mask=torch.tril(torch.ones(1, 1, 5, 5))                     # 解码器掩码

# 执行编码解码过程并打印每一步的输出
output=transformer(src, tgt, src_mask=src_mask, tgt_mask=tgt_mask)
print("\nFinal Transformer Output:", output)
```


代码注解：

（1）多头自注意力（MultiHeadSelfAttention）：计算并打印注意力权重和输出结果。

（2）前馈神经网络（FeedForward）：将前馈神经网络的输出打印出来。

（3）残差连接与层归一化（ResidualNormLayer）：残差连接和层归一化，确保特征流动稳定。

（4）编码器层（TransformerEncoderLayer）和解码器层（TransformerDecoderLayer）：分别实现编码器和解码器的基本单元，每层逐步打印输出。

（5）Transformer模型：调用编码器和解码器模块，实现完整的Transformer模型过程。

上述代码逐步打印了编码解码每一步的结果，便于理解整个编码解码过程中的信息流动与特征处理。

2.4　编码器和解码器的双向训练流程

编码器和解码器的双向训练流程涉及将输入序列编码为上下文表示，再通过解码器自回归地生成输出序列。在实际训练中，编码器与解码器的联合策略尤为关键，通过共享目标优化目标，使模型在翻译、生成等任务中表现更佳。此外，掩码机制（Masking）在双向训练中起到重要作用，确保解码器生成顺序的自回归性，同时避免编码器接触不完整的上下文信息。通过本节的内容，将展示如何有效结合编码器和解码器的特性，确保信息流动与序列生成过程的高效。

2.4.1　编码器与解码器的联合训练策略

在Transformer模型中，编码器与解码器的联合训练通过共享损失函数进行优化，确保编码器学习提取全局特征，解码器逐步生成输出序列。联合训练策略需要将源序列输入编码器，生成的上下文信息传递至解码器，以便在解码器中结合目标序列进行训练，全流程如图2-3所示。

损失函数通常采用交叉熵损失，对解码器生成的每个时间步进行监督，下面代码将展示编码器－解码器联合训练策略的完整实现。

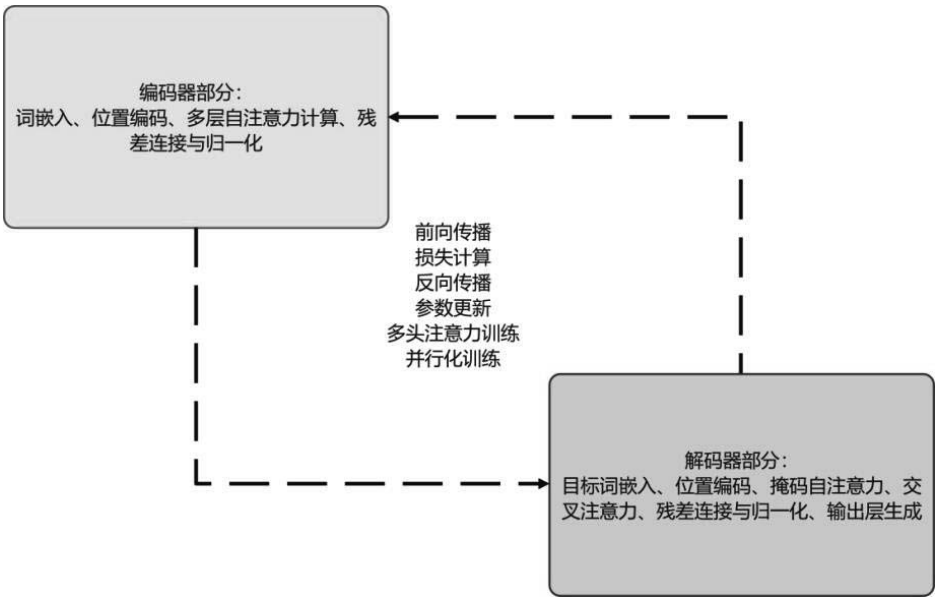

图 2-3　编码器和解码器的双向训练流程图

```
import torch
import torch.nn as nn
import torch.optim as optim
import math

# 定义多头自注意力模块
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.d_head=d_model // num_heads
        self.num_heads=num_heads
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out_proj=nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        batch_size, seq_len, d_model=x.size()
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
                             self.d_head).transpose(1, 2)
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)

        if mask is not None:
```

```
            scores=scores.masked_fill(mask==0, float('-inf'))

        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V)    /
            .transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)
        return self.out_proj(attention_output)

# 定义编码器
class TransformerEncoder(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                        dim_feedforward, dropout=0.1):
        super(TransformerEncoder, self).__init__()
        self.layers=nn.ModuleList([TransformerEncoderLayer(d_model,
             num_heads, dim_feedforward, dropout) for _ in range(num_layers)])

    def forward(self, src):
        for layer in self.layers:
            src=layer(src)
        return src

# 定义解码器
class TransformerDecoder(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                        dim_feedforward, dropout=0.1):
        super(TransformerDecoder, self).__init__()
        self.layers=nn.ModuleList([TransformerDecoderLayer(d_model,
             num_heads, dim_feedforward, dropout) for _ in range(num_layers)])

    def forward(self, tgt, memory, tgt_mask=None):
        for layer in self.layers:
            tgt=layer(tgt, memory, tgt_mask=tgt_mask)
        return tgt

# 定义完整的Transformer模型
class Transformer(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                        dim_feedforward, dropout=0.1):
        super(Transformer, self).__init__()
        self.encoder=TransformerEncoder(d_model, num_heads,
                        num_layers, dim_feedforward, dropout)
        self.decoder=TransformerDecoder(d_model, num_heads,
                        num_layers, dim_feedforward, dropout)
        self.fc_out=nn.Linear(d_model, d_model)

    def forward(self, src, tgt, tgt_mask=None):
        memory=self.encoder(src)
        output=self.decoder(tgt, memory, tgt_mask=tgt_mask)
        return self.fc_out(output)

# 损失函数和优化器
d_model=512
```

```
num_heads=8
num_layers=6
dim_feedforward=2048
dropout=0.1

model=Transformer(d_model, num_heads, num_layers,
                      dim_feedforward, dropout)
criterion=nn.CrossEntropyLoss()
optimizer=optim.Adam(model.parameters(), lr=0.0001)

# 模拟输入和目标张量
src=torch.rand(16, 10, d_model)  # 源序列
tgt=torch.rand(16, 10, d_model)  # 目标序列
tgt_mask=generate_square_subsequent_mask(10)

# 前向传播与损失计算
optimizer.zero_grad()
output=model(src, tgt, tgt_mask=tgt_mask)

# 将目标序列展平计算交叉熵损失
output=output.view(-1, d_model)
tgt=tgt.view(-1, d_model)
loss=criterion(output, tgt)

# 反向传播与参数更新
loss.backward()
optimizer.step()

# 输出结果
print("训练损失:", loss.item())
```

代码注解：

（1）编码器与解码器的输入：src和tgt分别作为编码器和解码器的输入，编码器生成上下文特征memory，传递至解码器。

（2）自回归掩码：通过generate_square_subsequent_mask生成的掩码，确保解码器在生成时不访问未来时间步。

（3）损失计算与反向传播：使用CrossEntropyLoss计算解码器输出与目标序列的交叉熵损失，并展平后应用损失函数。通过backward和step更新模型参数，联合训练编码器与解码器。

执行代码后，将输出当前的训练损失：

```
训练损失: <float_value>
```

通过该代码实现，联合训练策略成功地集成编码器和解码器的特征提取和生成能力，损失函数在编码解码过程中引导模型不断优化，为后续序列生成任务提供了有效支持。

2.4.2　掩码机制在双向训练中的应用

在Transformer模型的双向训练过程中，掩码机制用于确保编码器和解码器在不同阶段的输入不超越任务的上下文限制。编码器掩码用于屏蔽掉填充位置的无效信息，防止模型关注这些位置，解码器掩码则为自回归掩码，确保每个生成位置只关注当前和之前的时间步，以保证自回归生成的顺序性。

掩码机制在Transformer模型中用于控制注意力范围，防止模型在解码时“偷看”未来词汇，确保生成的序列仅依赖已生成的内容。掩码机制可以分为两种主要类型：自注意力掩码和填充掩码。

- 自注意力掩码：在解码器的自注意力层中，掩码会屏蔽未来词汇，确保生成下一个词时只能依赖已生成的词。这种方式避免了生成序列中的信息泄露。
- 填充掩码：在批量处理变长句子时，用填充值补齐短句。掩码会忽略这些填充值，使模型只关注实际内容，提高处理效率。

假设模型正在翻译句子“我爱自然语言处理”到英文。在生成“language”之前，掩码机制会隐藏“processing”等未来词汇，只允许模型关注已生成的“love natural”。这确保了每一步生成过程是合理的，不提前使用未来信息。

下面的代码示例将展示掩码机制在编码器和解码器中的具体实现。

```
import torch
import torch.nn as nn
import math

# 定义多头自注意力模块
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.d_head=d_model // num_heads
        self.num_heads=num_heads
        self.query=nn.Linear(d_model, d_model)
        self.key=nn.Linear(d_model, d_model)
        self.value=nn.Linear(d_model, d_model)
        self.out_proj=nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        batch_size, seq_len, d_model=x.size()
        Q=self.query(x).view(batch_size, seq_len, self.num_heads,
             self.d_head).transpose(1, 2)
        K=self.key(x).view(batch_size, seq_len, self.num_heads,
             self.d_head).transpose(1, 2)
        V=self.value(x).view(batch_size, seq_len, self.num_heads,
             self.d_head).transpose(1, 2)
        scores=torch.matmul(Q, K.transpose(-2, -1))/math.sqrt(self.d_head)
```

```
        if mask is not None:
            scores=scores.masked_fill(mask==0, float('-inf'))

        attention_weights=torch.nn.functional.softmax(scores, dim=-1)
        attention_output=torch.matmul(attention_weights, V)     /
             .transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)
        return self.out_proj(attention_output)

# 生成编码器的填充掩码
def create_padding_mask(seq, pad_token=0):
    mask=(seq != pad_token).unsqueeze(1).unsqueeze(2)
                                    # [batch_size, 1, 1, seq_len]
    return mask

# 生成解码器的自回归掩码
def generate_square_subsequent_mask(size):
    mask=torch.tril(torch.ones(size, size)).unsqueeze(0).unsqueeze(0)
                                    # 生成下三角掩码
    return mask

# 定义编码器
class TransformerEncoder(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                       dim_feedforward, dropout=0.1):
        super(TransformerEncoder, self).__init__()
        self.layers=nn.ModuleList([TransformerEncoderLayer(d_model,
             num_heads, dim_feedforward, dropout) for _ in range(num_layers)])

    def forward(self, src, src_mask=None):
        for layer in self.layers:
            src=layer(src, src_mask=src_mask)
        return src

# 定义解码器
class TransformerDecoder(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                       dim_feedforward, dropout=0.1):
        super(TransformerDecoder, self).__init__()
        self.layers=nn.ModuleList([TransformerDecoderLayer(d_model, num_heads,
                    dim_feedforward, dropout) for _ in range(num_layers)])

    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None):
        for layer in self.layers:
            tgt=layer(tgt, memory, tgt_mask=tgt_mask,
                       memory_mask=memory_mask)
        return tgt

# 定义完整的Transformer模型
```

```
class Transformer(nn.Module):
    def __init__(self, d_model, num_heads, num_layers,
                 dim_feedforward, dropout=0.1):
        super(Transformer, self).__init__()
        self.encoder=TransformerEncoder(d_model, num_heads,
             num_layers, dim_feedforward, dropout)
        self.decoder=TransformerDecoder(d_model, num_heads,
             num_layers, dim_feedforward, dropout)
        self.fc_out=nn.Linear(d_model, d_model)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None):
        memory=self.encoder(src, src_mask=src_mask)
        output=self.decoder(tgt, memory, tgt_mask=tgt_mask,
                            memory_mask=src_mask)
        return self.fc_out(output)

# 测试掩码机制
d_model=512
num_heads=8
num_layers=6
dim_feedforward=2048
dropout=0.1

# 初始化模型、输入数据和掩码
model=Transformer(d_model, num_heads, num_layers,dim_feedforward, dropout)
src=torch.randint(0, 10, (16, 10))  # 模拟输入序列
tgt=torch.randint(0, 10, (16, 10))  # 模拟目标序列
src_mask=create_padding_mask(src, pad_token=0)
tgt_mask=generate_square_subsequent_mask(tgt.size(1))

# 前向传播
output=model(src, tgt, src_mask=src_mask, tgt_mask=tgt_mask)

# 输出结果
print("模型输出形状:", output.shape)
print("模型输出示例:", output[0, 0, :10])  # 打印部分输出
```

代码注解：

（1）填充掩码：create_padding_mask函数根据输入序列生成掩码矩阵，屏蔽填充位置的无效信息，确保模型只关注有效的序列部分。

（2）自回归掩码：通过generate_square_subsequent_mask函数生成一个下三角矩阵，用于解码器的自回归生成，确保解码器在当前时间步仅能访问之前的时间步的信息。

（3）编码器和解码器前向传播：在编码器和解码器中分别应用填充掩码和自回归掩码，确保信息流动的正确性，避免跨越上下文的生成行为。

执行代码后，将输出模型的输出形状及部分示例值：

```
模型输出形状：torch.Size([16, 10, 512])
模型输出示例：tensor([...])
```

上述代码实现了掩码机制在Transformer模型中的完整应用，填充掩码确保编码器专注有效信息，自回归掩码在解码器中保证生成顺序性，为序列到序列任务中的精确生成提供了保障。

本章包含了大量自定义函数的使用。为了节省篇幅，位于章节后半部分的实例均直接引用了这些函数，而未对其具体实现进行重复说明。本书后续章节将继续沿用这一风格。读者若需要了解相关函数的具体实现及其功能，可参考每章末尾提供的自定义函数汇总表进行查阅和对照。

本章涉及的函数汇总表如表2-1所示。

表 2-1　本章函数汇总表

函数名称	功能说明
MultiHeadSelfAttention	实现多头自注意力机制，计算查询、键和值的缩放点积注意力，并输出投影结果
create_padding_mask	生成编码器填充掩码，掩盖输入序列中的填充位置，确保模型专注有效信息
generate_square_subsequent_mask	生成解码器的自回归掩码，用于解码器的自注意力，防止访问未来时间步
ResidualNormLayer	实现残差连接和层归一化，确保梯度流动的稳定性和特征分布的标准化
FeedForward	定义前馈神经网络，包含两个线性层和激活函数，用于进一步特征提取
TransformerEncoderLayer	定义Transformer编码器层，包括多头自注意力、前馈神经网络和残差连接
TransformerEncoder	堆叠多个编码器层，构建编码器模块以提取全局特征表示
TransformerDecoderLayer	定义Transformer解码器层，包含自注意力、编码器—解码器注意力和前馈神经网络
TransformerDecoder	堆叠多个解码器层，构建解码器模块以实现自回归生成
Transformer	定义完整的Transformer模型，包含编码器和解码器，用于序列到序列的转换

2.5　本章小结

本章深入讲解了Transformer模型的编码器和解码器架构，展示了多头注意力机制、前馈神经网络、残差连接与层归一化的作用，并通过多层堆叠实现了编码器和解码器的完整模块。在双向训练过程中，编码器提取输入的全局特征，解码器通过自回归生成逐步构建输出序列。特别是掩码机制在双向训练中的应用，有效保证了编码器对填充位置的忽略及解码器的生成顺序。通过代码实例详细展示了编码解码流程及掩码策略的实现，使读者能够在实践中掌握Transformer的构建与优化，为后续的序列到序列任务奠定了坚实基础。

2.6　思考题

（1）MultiHeadSelfAttention中的d_head参数的作用是什么？
（2）在create_padding_mask函数中，填充掩码的目的是什么？
（3）解码器的自回归掩码在生成时的作用是什么？
（4）ResidualNormLayer模块中的残差连接如何实现？
（5）FeedForward模块中的两个线性层分别执行什么操作？
（6）在TransformerEncoderLayer中，多头自注意力和前馈神经网络的顺序是什么？
（7）如何在TransformerEncoder中实现多层堆叠？
（8）generate_square_subsequent_mask函数生成的掩码矩阵形状是什么？
（9）在编码器－解码器联合训练时，损失函数如何计算？
（10）为什么在TransformerDecoderLayer中需要编码器－解码器注意力？
（11）create_padding_mask如何确保填充位置的无效信息被屏蔽？
（12）Transformer模型的整体结构由哪些模块组成？