第 3 章

训练 Transformer

本章深入探讨如何训练 Transformer 模型。在第 2 章中，介绍了 Transformer 模型的基本原理，并通过部分代码示例加深了对模型的理解，然而，这些代码主要用于理论讲解，不具备广泛的实用性和可扩展性，因此本章将使用 Python 和 PyTorch 来实现一个完整的 Transformer 模型。

本章内容分为两部分：第一部分将手写一个 Transformer 模型的实现，旨在整合第 2 章的代码，以更清晰地展示 Transformer 模型的整体架构。第二部分将使用开源数据集进行实践训练，展示如何在实际任务中应用 PyTorch 来训练 Transformer 模型。

注意：3.1 节内容主要根据第 2 章的内容对整体代码进行梳理，以此来辅助理解 Transformer 的基本原理，在实践中不具备实用性。

3.1 自定义 Transformer 代码

1. 代码环境介绍

Python 提供了丰富的科学计算类库，是深度学习领域的首选语言，所以在本书中，选用 Python 作为开发的主要语言，然而，仅仅依赖原生 Python 环境进行开发，可能会遇到包管理和依赖问题，尤其是在多个项目并行开发时。为了解决这些问题，并简化科学计算环境的配置，推荐使用 Anaconda。Anaconda 不仅包含了 Python 和许多常用的科学计算库，还提供了强大的包管理和虚拟环境管理功能，使开发过程更加顺畅和高效，可进入 Anaconda 的官网进行下载和安装，这里不进行详细介绍。

2. 安装虚拟环境

在正式进行代码训练前，建议使用 Anaconda 创建一个虚拟环境，可以有效地管理和隔离依赖，简化开发和部署流程，确保项目的稳定性和复现性。不同的 Python 项目可能需要同一库的不同版本。在虚拟环境中工作可以防止版本冲突。

创建虚拟环境步骤如下。

(1) 打开 Anaconda Prompt：依次单击“开始”→“所有程序”→Anaconda3→Anaconda

Prompt,这将启动类似于 CMD 的控制台窗口。

(2) 使用以下命令创建一个新的虚拟环境,其中 python=3.11 指定了 Python 版本(可以根据需要选择不同的版本)。如果未指定版本,则 Conda 将使用默认的 Python 版本创建环境,代码如下:

```
conda create --name transformer python=3.11 -y
```

conda create: 这是 Conda 命令,用于创建一个新的虚拟环境。

--name transformer: 将虚拟环境的名称指定为 transformer。在激活环境时,可以使用这个名称来引用该环境。

python=3.11: 将在该虚拟环境中安装的 Python 版本指定为 3.11。Conda 会自动从可用的包中选择兼容 Python 3.11 的包并进行安装。

-y: 这个参数表示自动确认安装所有必要的包,而不需要在安装过程中手动确认。这可以简化命令执行过程,使其不再需要人工干预。

(3) 创建环境后,需要先激活环境,执行的命令如下:

```
conda activate transformer
```

激活虚拟环境后会自动进入此环境,当前面的默认的环境 base 变成要激活的环境(Transformer)时,如图 3-1 所示,表示激活成功并进入了当前的虚拟环境。

```
(base) PS C:\Users\Administrator> conda activate transformer
(transformer) PS C:\Users\Administrator>
```

图 3-1 Conda 激活虚拟环境

3. 安装依赖包

在 Conda 创建的虚拟环境中,默认已安装了一些 Python 核心包和工具包,包括包管理工具 pip。Conda 作为一个强大的包管理器,非常适合处理复杂的科学计算包和环境配置,然而,Conda 的包管理渠道可能不包含 Python 生态系统中的所有包。在这种情况下,可以使用 pip 从 PyPI 仓库安装特定的 Python 包,因此在进入虚拟环境后,通过 pip 安装所需的依赖包是一种常见的做法,安装的命令为 pip install name,name 为需要安装的依赖包的名称,例如要安装 Pandas,命令如下:

```
pip install pandas

#卸载命令
pip uninstall name
```

在本章中需要安装的主要包为 PyTorch,PyTorch 是通用的深度学习框架,很多深度学习的运行过程需要借助 PyTorch 来完成相关计算,本书关于 Transformer 理论介绍部分也是借助 PyTorch 来进行演示的。PyTorch 的安装要比一般的依赖包复杂一些,主要是需要考虑 GPU 的配置。如果读者在本机没有 GPU,则可直接下载 CPU 版本的 PyTorch,使用

pip install torch 命令进行安装。

安装 GPU 版本的 PyTorch 需要根据 CUDA 版本选择 PyTorch 框架，在 Windows 操作系统上查看当前的 CUDA 版本的方式如下，在终端输入的命令如下：

```
nvidia-smi
```

显示内容如图 3-2 所示，右上角的 CUDA Version 为 12.5，表示当前驱动最高支持的 CUDA 版本为 12.5，在选择 PyTorch 的 CUDA 版本时，可以选择与 CUDA 12.5 兼容的版本（或更低的版本），只要确保安装的 PyTorch 二进制版本与当前的驱动程序和 CUDA 版本兼容即可。

```
(base) PS C:\Users\Administrator> nvidia-smi
Mon Aug 19 20:27:36 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 555.99                 Driver Version: 555.99         CUDA Version: 12.5     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 4060 Ti   WDDM  |   00000000:01:00.0  On |                  N/A |
|  0%   42C    P8             10W /  165W |     566MiB /  16380MiB |      2%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

图 3-2　GPU 配置信息

例如，使用的驱动支持 CUDA 12.5，可以安装 torch with cu121（CUDA 12.1 支持）版本，因为这是目前与 CUDA 12.5 驱动兼容的最高版本。如果 CUDA 12.5 是最高版本且使用的是官方支持的驱动程序版本，则可以选择 cu125 版本的 PyTorch。下载 PyTorch 时既可以进入官网（https://pytorch.org/）进行选择，如图 3-3 所示，也可以选择历史版本进行下载。

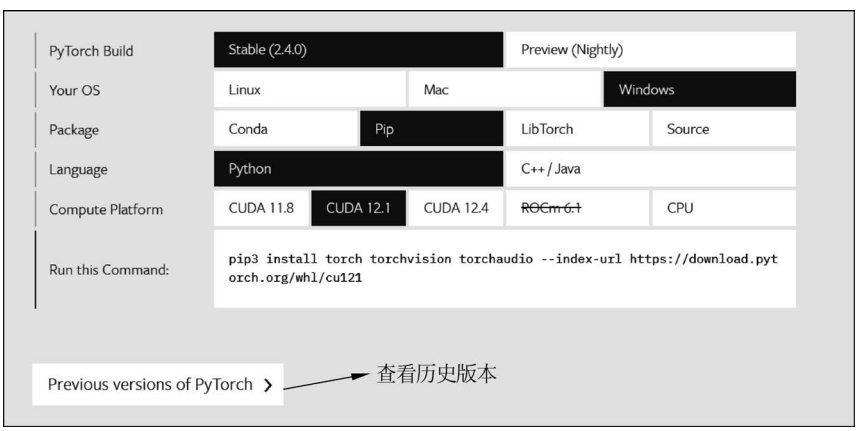

图 3-3　PyTorch 官网下载命令

建议选择历史版本(稳定性会更好些)2.1.2 进行下载,下载的代码如下:

```
pip install torch==2.1.2 torchvision==0.16.2 torchaudio==2.1.2 --index-url
https://download.pytorch.org/whl/cu121
```

下载完成后,在终端输入 python 进入 Python 语言环境,命令如下:

```
>>> import torch
>>> torch.__version__
'2.1.2+cu121'
```

如果输出的结果为'2.1.2+cu121',则表示使用的 PyTorch 版本是 2.1.2,并且它是针对 CUDA 12.1 编译的。这意味着目前 PyTorch 安装支持使用 CUDA 12.1 的 GPU 加速功能。

3.1.1 词嵌入和位置编码

在完成基本的环境设置后,开始设计 Transformer 模型的代码实现,按照图 2-7 的架构进行编码。本节将介绍词嵌入和位置编码的代码实现。在整个代码讲解过程中,有两点需要特别注意:一是数据的流向,二是数据的维度。

1. 词嵌入

与前文一致,使用开源工具包 Spacy 来生成词嵌入。Spacy 会将每个词编码为 96 维,因此 embed_dim=96,代码如下:

```
import spacy
nlp = spacy.load('zh_core_web_sm')
embed_dim = nlp("我").vector.shape[0]     #获取词嵌入的维度
print(embed_dim)                          #输出 96
```

除了需要设定词嵌入的维度,还需要设定输入序列的长度,即每次训练时输入的序列包含的 token 数量。通常会设定一个固定的长度,例如 10(max_length=10)。当输入的序列长度超过 10 时,取前 10 个 token;若不足 10 个 token,则用固定字符填充,确保序列长度一致,代码如下:

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-数据处理
from torch.utils.data import  Dataset
class TranslationDataset(Dataset):#继承自 torch.utils.data.Dataset
    def _init__(self, data, max_length=10):
        self.data = data
        self.max_length = max_length
    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        src_sentence, tgt_sentence = self.data[index]
```

```
        src_tokens = [token.vector for token in nlp(src_sentence)]
        tgt_tokens = [token.vector for token in nlp(tgt_sentence)]

        #将序列填充到最大长度
        src_tokens = self.pad_sequence(src_tokens, self.max_length)
        tgt_tokens = self.pad_sequence(tgt_tokens, self.max_length)
        #转换为张量,并确保为 float32
        return torch.tensor(src_tokens, dtype=torch.float32),
               torch.tensor(tgt_tokens, dtype=torch.float32)

    def pad_sequence(self, tokens, max_length):
        if len(tokens) < max_length:
            #如果序列比 max_length 短,则用零填充
            padding = [np.zeros(tokens[0].shape)] *(max_length - len(tokens))
            tokens.extend(padding)
        return tokens[:max_length]
```

(1) __len__()方法：返回数据集的长度,即句子对的数量,这是后续在训练过程中按索引访问数据集的基础。

(2) __getitem__()方法：根据索引 index 获取数据集中对应的句子对,并将其转换为词向量。通过调用 pad_sequence()方法填充或截断词向量列表,使其符合指定的最大长度 max_length,最后转换为 PyTorch 张量并返回,作为模型的输入和目标。

(3) pad_sequence()方法：用于填充或截断词向量列表,使其达到规定的最大长度。经过 TranslationDataset 处理后,输出的数据维度为 max_length * embed_dim(10 * 96)。

2. 位置编码

根据图 2-6 所示,位置编码与词嵌入相加,因此其形状必须与原始词嵌入保持一致,具体的代码逻辑可参考 2.2.3 节,代码如下：

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-位置编码
import math
import torch
class PositionalEncoding(nn.Module):#继承自 nn.Module
    def __init__(self, embed_dim, max_len=5000):
        super(PositionalEncoding, self).__init__()
        #创建一个足够长的`position` tensor
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        #根据维度创建分母
        div_term = torch.exp(torch.arange(0, embed_dim, 2).float() *(-math.log
(10000.0) / embed_dim))
        #生成正弦和余弦的位置编码
        pe = torch.zeros(max_len, embed_dim)
        pe[:, 0::2] = torch.sin(position *div_term)
        pe[:, 1::2] = torch.cos(position *div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
              #使其适应(batch_size, seq_len, embed_dim)的 shape
```

```
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:x.size(0), :]
        return x
```

在 PositionalEncoding 类中,embed_dim 和 max_len 为输入参数,最终输出的维度同样为 max_length * embed_dim(10 * 96)。

注意:self.register_buffer('pe',pe)将位置编码张量 pe 注册为模型的一部分,以确保在保存和加载模型时保留,但这些张量不参与梯度计算和优化过程,非常适合位置编码这种不需要更新的参数。

3.1.2 多头注意力层

在实现多头注意力层时,按照图 2-9 的架构进行编写。首先需要确定头的数量 num_heads,每个头独立计算注意力,然后将结果拼接起来并通过一个线性层输出。关键参数包括头的数量 num_heads、词嵌入维度 embed_dim 及每个注意力头的维度 head_dim(公式中的 d_k),整体的数据流程与 2.2.2 节类似。

注意:在定义该模块的结构和参数时,需要确保嵌入维度 embed_dim 能被头的数量 num_heads 整除。这是为了在多头自注意力机制中,保证每个头能够均匀地分割嵌入维度,这样每个头都能独立处理数据的一部分。同时,这也确保了在多头注意力过程中输入和输出的维度保持一致,在分割和合并嵌入向量时不会导致信息丢失。

多头注意力层定义的代码如下:

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-多头注意力层
import torch.nn.functional as F
class MultiHeadSelfAttention(nn.Module):#继承自 nn.Module
    def __init__(self, embed_dim, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        assert embed_dim % num_heads == 0, "Embedding dimension must be divisible
by the number of heads."
        self.head_dim = embed_dim //num_heads
        self.scaling = self.head_dim **-0.5
        #线性层,用于变换输入 query、key、value 和最后的输出
        self.query = nn.Linear(embed_dim, embed_dim)
        self.key = nn.Linear(embed_dim, embed_dim)
        self.value = nn.Linear(embed_dim, embed_dim)
```

```
        self.out_proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, query, key=None, value=None,mask = None):
        if key is None and value is None:
            key = value = query
        batch_size, seq_length, embed_dim = query.size()

        #线性变换得到 Q、K、V
        Q = self.query(query)     #(batch_size, seq_length, embed_dim)
        K = self.key(key)         #(batch_size, seq_length, embed_dim)
        V = self.value(value)     #(batch_size, seq_length, embed_dim)

        #多头分割,将 Q、K、V 的维度转换为 (batch_size,num_heads,seq_length, head_dim)
        Q = Q.view(batch_size, seq_length, self.num_heads, self.head_dim).
transpose(1, 2)                   #(batch_size, num_heads, seq_length, head_dim)
        K = K.view(batch_size, seq_length, self.num_heads, self.head_dim).
transpose(1, 2)                   #(batch_size, num_heads, seq_length, head_dim)
        V = V.view(batch_size, seq_length, self.num_heads, self.head_dim).
transpose(1, 2)                   #(batch_size, num_heads, seq_length, head_dim)
        #缩放点积运算
        scores = torch.matmul(Q, K.transpose(-2, -1)) *self.scaling
#(batch_size, num_heads, seq_length, seq_length)
        #掩码处理
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
attention_weights = F.softmax(scores, dim=-1)
                                  #(batch_size, num_heads, seq_length, seq_length)
        attention_output = torch.matmul(attention_weights, V)
                                  #(batch_size, num_heads, seq_length, head_dim)
        #合并多头,将维度转换回 (batch_size, seq_length, embed_dim)
attention_output = attention_output.transpose(1, 2).contiguous().view(batch_
size, seq_length, embed_dim)     #(batch_size, seq_length, embed_dim)
        #输出
        output = self.out_proj(attention_output)
                                  #(batch_size, seq_length, embed_dim)
        return output
```

(1) 参数传递判断：如果只传入一个输入，则表示编码器的多头注意力；如果传入 3 个参数，则为解码器的多头注意力。

(2) 线性变换：对输入的 query、key、value 分别进行线性变换，得到 Q、K、V 矩阵。

(3) 多头分割：将 Q、K、V 按 num_heads 分割为多个子空间，使每个头独立计算注意力。分割后的维度从(batch_size，seq_length，embed_dim)变为(batch_size，num_heads，seq_length，head_dim)。

(4) 缩放点积计算：按照公式进行缩放点积计算，得到注意力得分 scores。

(5) 掩码处理与归一化：先根据是否有掩码进行处理，并对得分进行归一化，再与 V 矩阵相乘，得到最终的注意力输出。

(6) 多头合并：将多头的输出维度从(batch_size，num_heads，seq_length，head_dim)转换为(batch_size，seq_length，embed_dim)。

(7) 线性层输出：通过线性层得到最终的输出结果。

注意：在多头自注意力机制中，在计算完注意力权重后，需要将其与V张量相乘以获得最终输出。为确保点积运算得到正确执行，首先需要将注意力权重的形状转换为与V张量一致，即(batch_size，seq_length，embed_dim)，然后通过.contiguous()方法将其转换为连续的内存布局，以确保高效的矩阵运算。

3.1.3 前馈网络层

前馈网络层的主要作用是在Transformer模型中对每个位置的词嵌入独立地进行非线性变换。整个过程包含两个线性变换：首先将嵌入维度embed_dim映射到一个更高的维度ff_dim，然后将其映射回原始的嵌入维度。前馈网络提供了更强的表达能力，并通过非线性激活函数来增强模型的复杂性，代码如下：

```
#第3章/3.1自定义Transformer编码.ipynb-前馈网络层
import torch.nn.functional as F
class FeedForwardNetwork(nn.Module):
    def __init__(self, embed_dim, ff_dim):
        super(FeedForwardNetwork, self).__init__()
        self.fc1 = nn.Linear(embed_dim, ff_dim)
        self.fc2 = nn.Linear(ff_dim, embed_dim)
        self.dropout = nn.dropout(dropout)   #防止过拟合的DropOut层
    def forward(self, x):
        x = self.fc1(x)                      #(batch_size, seq_length, ff_dim)
        x = F.relu(x)                        #非线性激活函数
        x = self.dropout(x)                  #添加DropOut防止过拟合
        x = self.fc2(x)                      #(batch_size, seq_length, embed_dim)
        return x
```

(1) 升维与降维：前馈网络首先通过线性层fc1将嵌入维度embed_dim映射到一个更高的维度ff_dim，然后通过fc2将其重新映射回embed_dim。这种结构使网络能够捕获更多的特征表示。

(2) 非线性激活：在线性升维后，使用ReLU激活函数对输出进行非线性变换。这种激活函数引入了非线性特性，使模型更具表达能力。

(3) DropOut正则化：为了防止过拟合，加入了DropOut层(默认值为0.1)。这会在训练过程中随机丢弃一部分神经元，有助于提高模型的泛化能力。

(4) 输出维度：最后使输出维度与输入保持一致，即(batch_size，seq_length，embed_dim)，这样可以方便与后续的模型层进行连接。

注意：激活函数的选择：ReLU是一种常见的激活函数，能够提高网络的非线性表达能力，然而在某些任务中，可能需要尝试其他激活函数，例如GELU，以获得更好的效果。DropOut的使用：在实际任务中，DropOut可以根据模型的复杂度和数据的规模进行调整。通常，较小的数据集或者较复杂的模型会使用更高的DropOut率，以防止过拟合。

3.1.4　编码器层和解码器层

1. 编码器层

编码器层的主要任务是首先进行多头注意力计算，然后进入前馈网络层。在这两个步骤中都需要加入层归一化和残差连接，以提高训练的稳定性和模型的表现。多头注意力计算使用3.1.2节定义的MultiHeadSelfAttention函数，前馈网络层使用3.1.3节中的FeedForwardNetwork，代码如下：

```
#第3章/3.1自定义Transformer编码.ipynb-编码器
class TransformerEncoderLayer(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_dim):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attention = MultiHeadSelfAttention(embed_dim, num_heads)
        self.feed_forward = FeedForwardNetwork(embed_dim, ff_dim)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)

    def forward(self, src):
        #多头注意力机制，结合残差连接和层归一化
        attn_output = self.self_attention(src)
        src = self.norm1(src + self.dropout(attn_output)) #(batch_size, seq_length,
                                                          #embed_dim)
        #前馈网络层，结合残差连接和层归一化
        ff_output = self.feed_forward(src)
        src = self.norm2(src + self.dropout(ff_output))   #(batch_size, seq_length,
                                                          #embed_dim)
        return src
```

(1) 多头注意力机制：首先对输入的源序列进行多头注意力计算，然后将结果与原始输入进行残差连接，并通过层归一化处理，以保持模型的稳定性和深度训练的可行性。

(2) 前馈网络层：随后，经过前馈网络层处理，再次进行残差连接和层归一化，输出最终的编码器层结果。

(3) DropOut的引入：在残差连接前使用DropOut，有助于防止过拟合并提高模型的泛化能力。

(4) 最终，编码器层的输出维度为(batch_size, seq_length, embed_dim)。

2. 解码器层

相比于编码器层，解码器层稍显复杂，主要增加了两部分内容：掩码多头注意力机制和

交互多头注意力机制。在训练时,解码器需要传入一个掩码矩阵 tgt_mask,以确保模型不会在预测下一个词时看到后面的词。此外,交互多头注意力机制的 Q 矩阵来自掩码注意力层的输出,而 K、V 矩阵则来自编码器层的输出,代码如下:

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-解码器
class TransformerDecoderLayer(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_dim):
        super(TransformerDecoderLayer, self).__init__()
        self.self_attention = MultiHeadSelfAttention(embed_dim, num_heads)
        self.cross_attention = MultiHeadSelfAttention(embed_dim, num_heads)
        self.feed_forward = FeedForwardNetwork(embed_dim, ff_dim)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.norm3 = nn.LayerNorm(embed_dim)

    def forward(self, tgt, memory, tgt_mask):
        #掩码多头注意力机制,加上残差连接和层归一化
        self_attn_output = self.self_attention(tgt,tgt,tgt,tgt_mask)
        tgt = self.norm1(tgt + self_attn_output)  #(batch_size, seq_length,
                                                  #embed_dim)
        #交互多头注意力机制,加上残差连接和层归一化
        cross_attn_output = self.cross_attention(tgt, memory, memory)
        tgt = self.norm2(tgt + self.dropout(cross_attn_output)) #(batch_size, seq_
                                                                #length,embed_dim)
        #前馈网络层,加上残差连接和层归一化
        ff_output = self.feed_forward(tgt)
        tgt = self.norm3(tgt + self.dropout(ff_output))         #(batch_size, seq_
                                                                #length,embed_dim)
        return tgt
```

(1) 掩码多头注意力机制:首先进行掩码多头注意力计算,保证在预测下一个词时,模型不会看到未来的词。注意这里的 Q、K、V 参数都需要显式传递,并传入掩码 tgt_mask。

(2) 交互多头注意力机制:使用编码器输出的 memory 作为 K、V 矩阵,对经过掩码多头注意力处理后的序列进一步地进行交互注意力计算。

(3) 前馈网络层:最后,通过前馈网络层进行非线性变换,结合残差连接和层归一化,输出最终的解码器层结果。

注意:*在多头注意力层 MultiHeadSelfAttention 的 forward 方法中,有 4 个参数(Q、K、V 和 mask)。在掩码多头注意力机制中,即使 Q、K、V 相同,也需要显式传递,并附带掩码矩阵 tgt_mask。*

3.1.5 构建 Transformer 模型

在完成各个组件的实现后,需要将这些组件组合起来,形成完整的 Transformer 模型。

Transformer 模型的具体实现代码如下：

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb
class TransformerModel(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_dim, num_encoder_layers, num_
decoder_layers, max_len=5000):
        super(TransformerModel, self).__init__()
        self.positional_encoding = PositionalEncoding(embed_dim, max_len)
        self.encoder_layers = nn.ModuleList([TransformerEncoderLayer(embed_
dim, num_heads, ff_dim) for _ in range(num_encoder_layers)])
        self.decoder_layers = nn.ModuleList([TransformerDecoderLayer(embed_
dim, num_heads, ff_dim) for _ in range(num_decoder_layers)])

    def forward(self, src, tgt, tgt_mask):
        #加上位置编码
        src = self.positional_encoding(src)
        tgt = self.positional_encoding(tgt)
        #遍历编码器层
        memory = src
        for layer in self.encoder_layers:
            memory = layer(memory)
        #遍历解码器层
        output = tgt
        for layer in self.decoder_layers:
            output = layer(output, memory ,tgt_mask)
        return output
```

（1）位置编码：首先，对输入的源序列 src 和目标序列 tgt 添加位置编码，这一步将序列中的位置信息编码到词嵌入中，使模型能够考虑序列的顺序。

（2）编码器计算：遍历所有的编码器层，将输入数据逐层传递，最终得到编码器的输出 memory，该输出将被用于解码器的交互多头注意力计算。

（3）解码器计算：将目标序列和编码器的输出 memory 一起传递到解码器层，解码器逐层处理，最终输出解码结果。

注意：nn. ModuleList 是 PyTorch 中的一个特殊容器，它用于存储子模块。与普通 Python 列表不同，nn. ModuleList 能够在训练时正确地注册这些子模块，使它们的参数能够被自动管理。这样，可以轻松地遍历和调用编码器层和解码器层，而无须手动索引。

3.1.6　训练 Transformer 模型

Transformer 模型的训练过程与常见的深度学习流程类似，主要包括超参数设置、模型初始化、训练过程及结果分析。

1. 参数设置

在开始训练之前，需要设置一些关键的超参数。这些参数包括词嵌入维度、编码器和解

码器的层数、多头注意力头的数量、最大序列长度及掩码矩阵的创建，代码如下：

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-训练
import torch
import spacy
#获取词嵌入维度
input_dim = nlp("我").vector.shape[0]  #从 Spacy 获取词嵌入的维度
ff_dim = 512                           #前馈神经网络的中间维度
n_layers = 2                           #编码器和解码器的层数
n_heads = 8                            #多头注意力的头数
max_len = 100                          #最大序列长度
#创建目标序列掩码
def create_target_mask(size):
    mask = torch.tril(torch.ones((size, size))).unsqueeze(0).unsqueeze(0)
    return mask                        #返回形状为 (1, 1, size, size) 的掩码矩阵
```

注意：编码器和解码器的层数可以不一致，根据任务需求进行调整。

2. 模型初始化

接下来，需要实例化模型，准备训练数据，并定义损失函数和优化器，代码如下：

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-训练
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
#准备训练数据
train_data = [
    ("我爱中国", "I love China"),
    ("我喜欢的水果是橙子", "I like oranges"),
    ("我喜欢吃苹果", "I like eating apples"),
    ("我很喜欢华为手机", "I really like Huawei phones")
]
#实例化模型
model = TransformerModel(input_dim, n_heads, ff_dim, n_layers, n_layers, max_len=
max_len)
#数据预处理
dataset = TranslationDataset(train_data)
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
#定义损失函数和优化器
criterion = nn.MSELoss()                                    #损失函数
optimizer = optim.Adam(model.parameters(), lr=0.001)    #优化器
```

3. 训练过程

在训练过程中，需要迭代地进行前向传播、损失计算、反向传播和参数更新，代码如下：

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-训练
loss_values = []                    #保存损失值
```

```
for epoch in range(100):                                    #设置训练轮数
    model.train()                                           #确保模型处于训练模式
    total_loss = 0                                          #初始化总损失
    for src, tgt in dataloader:
        optimizer.zero_grad()

        #获取目标序列的长度并创建掩码
        tgt_len = tgt.size(1)
        tgt_mask = create_target_mask(tgt_len)

        #前向传播:将输入和目标输入传入模型,同时传入掩码
        output = model(src, tgt, tgt_mask)
        #计算损失
        loss = criterion(output, tgt)
        total_loss += loss.item()
        #反向传播和优化
        loss.backward()
        optimizer.step()

    #输出每轮训练的平均损失
    avg_loss = total_loss / len(dataloader)
    loss_values.append(avg_loss)
    print(f'Epoch {epoch+1}, Loss: {avg_loss:.4f}')    #打印 loss
```

注意：解码器层需要使用掩码矩阵以确保在训练过程中模型不会看到未来的词。

4. 结果分析

训练完成后,可以绘制损失值随训练轮数变化的折线图,以直观地观察模型的收敛情况,代码如下：

```
#第 3 章/3.1 自定义 Transformer 编码.ipynb-训练
import matplotlib.pyplot as plt
#绘制损失的折线图
plt.figure(figsize=(10, 6))
plt.plot(loss_values, linestyle='-', color='b')
plt.title('Training Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.show()
```

Matplotlib 绘图结果如图 3-4 所示。

根据图中展示的结果可以看到,损失值随着训练的进行而递减,但是训练数据量极少(仅 4 条记录),模型并不具备实际的预测能力。本节的主要目的是通过代码实践帮助理解 Transformer 的理论部分,接下来的章节将进一步地对模型进行训练和优化。

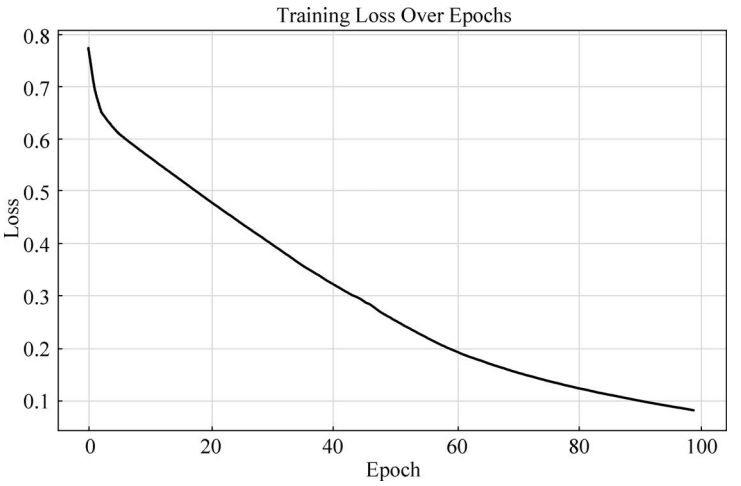

图 3-4　Matplotlib 绘图结果

3.2　实践训练

在本节中，将使用一个开源的新闻数据集进行实践演练，任务是利用 Transformer 模型进行中文文本分类。为了简化流程并提高性能，将直接调用 PyTorch 提供的高效封装模块，这些模块已经针对内存优化等问题进行了处理，更适合在实际应用中使用。

3.2.1　数据准备

首先对新闻数据集进行预处理，具体步骤包括数据加载、文本预处理（例如分词）、词嵌入的生成及数据集的构建。词嵌入部分将使用 Gensim 库中的 Word2Vec 模型。

1. 读取数据

供使用的数据集是一个包含 1000 条记录的开源新闻数据集，对此数据集进行训练，数据集包含 10 个类别，分别为['教育', '体育', '科技', '时尚', '房产', '家居', '财经', '时政', '娱乐', '游戏']，代码如下：

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
#加载数据
data_path = './data/news.csv'
news_data = pd.read_csv(data_path)
#将标签从字符串转换为数值型
label_encoder = LabelEncoder()
news_data['label'] = label_encoder.fit_transform(news_data['label'])
```

注意：新闻数据集中的标签最初是字符串（例如'教育','体育'等），为了使其适应模型训

练，需要将这些字符串标签转换为数值标签。使用 LabelEncoder 实现这一转换。

2. 文本分词

文本分词是文本预处理的重要步骤。在中文文本处理中，分词是一项必不可少的任务。中文分词的方式有很多，Python 提供了很多开源的依赖包，这些包可以完成此任务，这里还是使用前面用过的 Spacy 来实现，代码如下：

```
import spacy
nlp = spacy.load('zh_core_web_sm')
def preprocess_text(text):
    #使用 Spacy 进行中文分词
    return [token.text for token in nlp(text)]
news_data['tokenized'] = news_data['text'].apply(preprocess_text)
```

分词之后，需要检查数据的效果，如图 3-5 所示，确保分词过程符合预期。

	label	text	tokenized
0	4	澳移民子女成长记：带着中国心融入主流社会新华网悉尼5月31日电 无论哪个国家的父辈与子女间都...	[澳, 移民, 子女, 成长, 记, ：, , 带, 着, 中国, 心, 融入, 主流, 社会,...
1	0	快船vs火箭首发：休城旨在练兵 小德帕特森进先发新浪体育讯北京时间4月10日消息，在常规赛还...	[快, 船, vs, 火箭, 首发, ：, 休, 城旨, 在, 练兵, 小德帕特, 森进, ...
2	8	3英寸屏高清闪存DV 三洋TH1特价1499 作者：中关村在线 飘雪 ...	[3, 英寸, 屏, 高, 清闪, 存D, V, 三洋TH1, 特价, 1499, ...
3	5	贝嫂乏味归乏味 还有人买账Victoria Beckham 乏味归乏味 还有人买账大姐大的阵...	[贝嫂, 乏味, 归乏, 味, 还有, 人, 买, 账Victoria, Beckham, ...
4	3	三亚岭南赶房集 金九银十再兴购房游纯粹的旅行闲适有余却"收获"不足，设计一条可以兼容曼妙风景...	[三亚, 岭南, 赶房, 集, 金九, 银, 十, 再, 兴购, 房, 游, 纯粹, 的, ...
...	...	...	...
995	1	组图：本-斯蒂勒与布莱克助阵《寻找伴郎》首映新浪娱乐讯 北京时间3月18日(美国当地时间3月...	[组图, ：, , 本, -, 斯蒂勒, 与, 布莱克, 助阵, 《, 寻找, 伴郎, 》, 首...
996	3	房源阶段性不足价格高涨 楼市将上演新一轮疯狂近日，中海地产(企业专区,旗下楼盘)以70.06...	[房源, 阶段性, 不足, 价格, 高涨, 楼市, 将, 上演, 新, 一, 轮, 疯狂, ...
997	8	宽屏广角高清DC！佳能110IS仅售1980【山东IT在线报道】佳能IXUS 110 IS装...	[宽屏, 广角, 高清DC, ！, 佳能, 110, IS, 仅售, 1980, 【, 山东...
998	6	公安部建成打拐DNA数据库通缉50名人贩●不到1个月，侦破拐卖儿童、妇女案件逾300起，解救...	[公安部, 建成, 打拐, DNA, 数据库, 通缉, 50, 名, 人贩, ●, 不, 到...
999	0	全场打铁44次也能赢球？公牛两项利器杀翻步行者新浪体育讯NBA季后赛东区首轮征战中，此前大...	[全场, 打铁, 44, 次, 也, 能, 赢球, ？, 公牛, 两, 项, 利器, 杀翻,...

1000 rows × 3 columns

图 3-5　新闻数据集查看

3. 词嵌入

为了将文本转换为模型可处理的数值形式，使用 Gensim 库中的 Word2Vec 模型来训练词嵌入。Word2Vec 模型通过上下文学习每个词的向量表示，这些向量可以捕捉到词语之间的语义关系，代码如下：

```
from gensim.models import Word2Vec
#训练 Word2Vec 模型
w2v_model = Word2Vec(sentences=news_data['tokenized'], vector_size=100, window=5,
min_count=1, workers=4)
```

(1) news_data['tokenized']：是预处理后的分词文本列表，Word2Vec 会根据这些分词的句子来训练词嵌入。每个句子是一个单词的列表，Word2Vec 会使用这些句子来学习每个词在上下文中的关系。

(2) vector_size=100：是词嵌入向量的维度，Word2Vec 将每个单词嵌入一个 100 维的向量空间中。选择多大的 vector_size 取决于实际的数据和任务。较小的值(例如 50 或 100)通常适用于小数据集或简单任务，而较大的值(例如 300)适用于更复杂的任务或更大

规模的数据集。

(3) window=5：这是窗口大小，表示模型在训练过程中会考虑每个单词前后 5 个词的上下文。换句话说，模型会在句子中查看距离当前单词 5 个位置内的其他单词，以学习它们的关系。较大的窗口值可以捕捉到更远的上下文信息。

(4) min_count=1：这是最低词频限制，表示只考虑出现次数至少为 1 的词。如果一个单词在整个语料库中只出现了少于 min_count 的次数，则它将被忽略。在当前例子中，由于是一个小型数据集，min_count=1，也就是所有词都被考虑进去。在较大的数据集上，通常会设置一个更高的值，例如 5 或 10，以去除低频词。

(5) workers=4：这是用于训练的线程数。多线程可以加速训练过程，特别是在大规模数据集上。在这个例子中，workers=4 表示将使用 4 个 CPU 核心来进行训练。

4. 构建数据集

为了将数据转换为 PyTorch 能够使用的格式，需要构建一个自定义的 Dataset 类，并将文本数据转换为词向量，代码如下：

```
#3.2pytorch 中文分类.ipynb-构建数据集
from torch.utils.data import Dataset
class NewsDataset(Dataset):
    def __init__(self, data, w2v_model, max_length=100):
        self.data = data
        self.w2v_model = w2v_model
        self.max_length = max_length
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        tokens = self.data.iloc[idx]['tokenized']
        label = self.data.iloc[idx]['label']

        #将 tokens 转换为词向量，并进行 padding
        vectors = [self.w2v_model.wv[token] for token in tokens if token in self.
w2v_model.wv]
        if len(vectors) < self.max_length:
            vectors += [torch.zeros(self.w2v_model.vector_size)] *
                       (self.max_length - len(vectors))
        else:
            vectors = vectors[:self.max_length]

        return torch.tensor(vectors, dtype=torch.float32),
               torch.tensor(label, dtype=torch.long)
```

(1) 数据转换：将 tokens 列表中的每个词都通过 Word2Vec 模型的 w2v_model 转换为对应的词向量，在 Word2Vec 词汇表中查找所有的 token，使用 w2v_model.wv[token]获取其词向量。

(2) 数据填充：如果文本 vectors 的长度(词向量的数量)小于 max_length，则在列表

vectors 的末尾添加零向量(torch. zeros)以填充到 max_length。

torch. zeros(self. w2v_model. vector_size)创建了一个全零的向量,维度与词向量的维度一致。如果文本的长度大于 max_length,则截断列表 vectors,仅保留前 max_length 个词向量。

(3) 类型转换:最后将 vectors 列表转换为 PyTorch 的 FloatTensor,以便用于模型训练。标签 label 也被转换为 PyTorch 的 LongTensor,这是分类任务中常用的张量类型。

5. 划分数据集

为了验证模型的效果,将数据集划分为训练集和测试集,并使用 DataLoader 进行批处理和数据加载,代码如下:

```
#3.2pytorch 中文分类.ipynb-
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split
#划分训练集和测试集
train_data, test_data = train_test_split(news_data, test_size=0.2, random_state=42)
train_dataset = NewsDataset(train_data, w2v_model)
test_dataset = NewsDataset(test_data, w2v_model)

#构建 DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

(1) train_test_split:以 80∶20 的比例划分训练集和测试集,random_state=42 用于设置随机种子,以确保每次运行时划分结果相同,保证实验的可重复性。

(2) DataLoader:使用 DataLoader 进行批量加载数据,batch_size=32 表示每个批次包含 32 条记录,shuffle=True 确保训练数据在每个 epoch 时被随机打乱,帮助模型更好地泛化。

3.2.2　模型定义及训练

在实际的深度学习模型的开发过程中,使用 PyTorch 提供的内置模块可以显著地简化开发流程。在本节的实践中,将使用 torch. nn 中的 Transformer 模块来构建一个文本分类模型,并对其进行训练和评估。

1. 模型定义

模型的定义基于 PyTorch 的 nn. Module,并使用 nn. Transformer 来实现 Transformer 架构。通过添加位置编码来保留序列中单词的顺序信息,并使用一个线性分类器来输出分类结果,代码如下:

```
#3.2pytorch 中文分类.ipynb-模型定义
import torch
import torch.nn as nn
import math
```

```
class TransformerTextClassifier(nn.Module):
    def __init__(self, input_dim, n_heads, ff_dim, num_encoder_layers, num_
decoder_layers, num_classes, max_len=100):
        super(TransformerTextClassifier, self).__init__()
        #使用正弦-余弦位置编码
        self.positional_encoding = self.create_positional_encoding(input_dim,
max_len)
        self.transformer = nn.Transformer(#Transformer 模型初始化
            d_model=input_dim,
            nhead=n_heads,
            num_encoder_layers=num_encoder_layers,
            num_decoder_layers=num_decoder_layers,
            dim_feedforward=ff_dim,
            dropout=0.1
        )
        self.classifier = nn.Linear(input_dim, num_classes)      #分类器
        self.max_len = max_len

    def create_positional_encoding(self, d_model, max_len):
        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)                #形状为 (1, max_len, d_model)
        return pe

    def add_positional_encoding(self, x):
        batch_size, seq_len, embed_dim = x.size()
        #确保位置编码的形状为 (1, seq_len, embed_dim)
        position_encoding = self.positional_encoding[:, :seq_len, :]
        position_encoding = position_encoding.expand(batch_size, seq_len, embed_
dim)                                    #扩展到 (batch_size, seq_len, embed_dim)
        x = x + position_encoding.to(x.device)
        return x

    def forward(self, src, tgt):
        src = self.add_positional_encoding(src)
        tgt = self.add_positional_encoding(tgt)

        src = src.transpose(0, 1)           #转换成(seq_len, batch_size, input_dim)
        tgt = tgt.transpose(0, 1)

        transformer_output = self.transformer(src, tgt)
        output = self.classifier(transformer_output[0])
                                            #取序列的第 1 个时间步的输出进行分类
        return output
```

（1）位置编码实现：create_positional_encoding 生成位置编码，用于为输入序列中的每个位置编码，这些编码将被加到词向量中，以便保留词语的顺序信息。add_positional_encoding 将生成的位置编码添加到输入的词向量中，并确保维度匹配。

（2）Transformer 实现：nn. Transformer 是 PyTorch 中的一个强大的模块，提供 Transformer 架构的实现。也就是一个模块中封装了 Transformer 实现的全部功能，包括自注意力机制、交互注意力机制、前馈神经网络和残差连接等内容，在实际使用时只需定义里面的超参数。

注意：使用 transformer_output[0]取序列的第 1 个时间步的输出进行分类，这是因为第 1 个时间步的输出已经通过 Transformer 的自注意力机制充分整合了整个序列的上下文信息。自注意力机制使每个时间步的输出不仅包含该位置的信息，还包括序列中所有其他位置的信息，因此第 1 个时间步的输出通常能够很好地代表整个序列的全局语义，成为一个有效的序列表征。在不增加计算复杂度的情况下，这种方法提供了一个合理且高效的全局特征表示。

当然，根据具体任务的需求，也可以选择其他时间步的输出进行训练和预测，例如，在序列标注任务（例如命名实体识别）中，每个时间步的输出都可能被用于预测标签。或者，在一些任务中，可能需要使用最后一个时间步的输出，或者对所有时间步的输出进行聚合来获得更全面的序列表征。

尽管如此，通常情况下会优先选择第 1 个时间步的输出来完成分类等任务，这是因为它在简洁性和性能之间取得了良好的平衡，尤其适合全局性的序列分类任务。

---

2. 模型训练

下面根据前面处理好的数据和定义的模型进行训练，首先检查当前的计算环境是否有可用的 GPU，并根据检测结果选择使用 GPU 还是 CPU 作为计算设备。在深度学习任务中，使用 GPU 可以大大地加速模型训练和推理的速度，因此在有可用的 GPU 时通常会优先选择 GPU，代码如下：

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

设定相关参数并初始化模型，代码如下：

```
#3.2pytorch 中文分类.ipynb-开始训练
input_dim = w2v_model.vector_size
ff_dim = 512
n_layers = 2
n_heads = 5
max_len = 100
num_classes = news_data['label'].nunique()
```

```
model = TransformerTextClassifier(input_dim, n_heads, ff_dim, n_layers, n_
layers, num_classes, max_len).to(device)

#定义损失函数和优化器
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

开始训练,代码如下:

```
#3.2pytorch 中文分类.ipynb-训练
from sklearn.metrics import accuracy_score
#记录每个 epoch 的 loss 和 accuracy
train_losses = []
test_losses = []
train_accuracies = []
test_accuracies = []

num_epochs = 10  #每次完整的遍历训练数据集都称为一个 epoch
for epoch in range(num_epochs):#开始迭代
    #测试模型
    model.train()
    total_train_loss = 0
    all_train_preds = []
    all_train_labels = []

    for src, labels in train_dataloader:
        src, labels = src.to(device), labels.to(device)

        optimizer.zero_grad()

        tgt = src
        output = model(src, tgt)

        loss = criterion(output, labels)
        total_train_loss += loss.item()

        loss.backward()
        optimizer.step()

        preds = torch.argmax(output, dim=1)
        all_train_preds.extend(preds.cpu().NumPy())
        all_train_labels.extend(labels.cpu().NumPy())

    avg_train_loss = total_train_loss / len(train_dataloader)#计算 loss
    train_accuracy = accuracy_score(all_train_labels, all_train_preds)#auc
    train_losses.append(avg_train_loss)
    train_accuracies.append(train_accuracy)

    #测试模型
```

```
    model.eval()
    total_test_loss = 0
    all_test_preds = []
    all_test_labels = []
    with torch.no_grad():
        for src, labels in test_dataloader:
            src, labels = src.to(device), labels.to(device)

            tgt = src
            output = model(src, tgt)

            loss = criterion(output, labels)
            total_test_loss += loss.item()

            preds = torch.argmax(output, dim=1)
            all_test_preds.extend(preds.cpu().NumPy())
            all_test_labels.extend(labels.cpu().NumPy())

    avg_test_loss = total_test_loss / len(test_dataloader)
    test_accuracy = accuracy_score(all_test_labels, all_test_preds)
    test_losses.append(avg_test_loss)
    test_accuracies.append(test_accuracy)

    print(f'Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}, Train Accuracy:
{train_accuracy:.4f}, Test Loss: {avg_test_loss:.4f}, Test Accuracy: {test_
accuracy:.4f}')
```

训练结果如下：

```
Epoch 1, Train Loss: 2.0935, Train Accuracy: 0.2325, Test Loss: 1.5293, Test
Accuracy: 0.4400
Epoch 2, Train Loss: 1.4415, Train Accuracy: 0.5050, Test Loss: 1.0853, Test
Accuracy: 0.6100
Epoch 3, Train Loss: 1.1685, Train Accuracy: 0.5625, Test Loss: 0.9928, Test
Accuracy: 0.6700
Epoch 4, Train Loss: 0.9713, Train Accuracy: 0.6450, Test Loss: 0.8241, Test
Accuracy: 0.7250
Epoch 5, Train Loss: 0.8789, Train Accuracy: 0.6725, Test Loss: 0.7548, Test
Accuracy: 0.7450
Epoch 6, Train Loss: 0.7791, Train Accuracy: 0.7350, Test Loss: 0.6973, Test
Accuracy: 0.7700
Epoch 7, Train Loss: 0.6696, Train Accuracy: 0.7612, Test Loss: 0.6397, Test
Accuracy: 0.7700
Epoch 8, Train Loss: 0.5857, Train Accuracy: 0.7975, Test Loss: 0.5596, Test
Accuracy: 0.8300
Epoch 9, Train Loss: 0.5688, Train Accuracy: 0.8013, Test Loss: 0.7821, Test
Accuracy: 0.7450
Epoch 10, Train Loss: 0.4875, Train Accuracy: 0.8287, Test Loss: 0.5690, Test
Accuracy: 0.8250
```

随着训练的进行,模型的损失不断下降,准确率不断提升,表明模型在逐步收敛。在实际的任务中通过合理地选择模型的超参数和训练策略,能够有效地提高模型的性能。

3.2.3 模型预测

在模型训练完成后,定义一个 predict_new_data 函数,用于对新的文本数据进行分类预测。该函数首先对输入的文本进行预处理并转换为词向量,然后使用训练好的模型进行预测,并将预测结果映射回原始的文本标签,代码如下:

```
#3.2pytorch 中文分类.ipynb-预测
def predict_new_data(model, new_text, w2v_model, label_encoder, device, max_len=
100):
    model.eval()                #将模型设置为评估模式
    tokens = preprocess_text(new_text)
    vectors = [w2v_model.wv[token] for token in tokens if token in w2v_model.wv]
                                #将分词结果转换为词向量
    if len(vectors) < max_len:
         vectors += [torch.zeros(w2v_model.vector_size)] * (max_len - len
(vectors))
    else:
        vectors = vectors[:max_len]
    #将词向量转换为 PyTorch 张量,并添加批次维度
    input_tensor = torch.tensor(vectors, dtype=torch.float32).unsqueeze(0).to
(device)
    tgt = input_tensor
    with torch.no_grad():#关闭梯度计算进行推理,减少内存消耗
        output = model(input_tensor, tgt)
        pred = torch.argmax(output, dim=1).item()
    #使用 label_encoder.inverse_transform 将数值标签转换为原始文本标签
    return label_encoder.inverse_transform([pred])[0]
```

为了验证 predict_new_data 函数的正确性,可以用原始数据集中的一条记录进行预测,代码如下:

```
#预测新数据
new_text = news_data["text"][0]
predicted_label = predict_new_data(model, new_text, w2v_model, label_encoder,
device)
print(f'Predicted Label : {predicted_label}')
```

打印结果如下:

```
Predicted Label : 教育
```

预测时的数据处理逻辑与训练过程基本是一致的,最后需要将经预测得到的数值标签 pred 转换回原始的文本标签(预测的分类标签)。