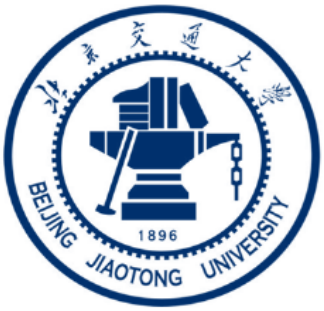


北京交通大學

BEIJING JIAOTONG UNIVERSITY

《铁路智能感知技术》课程

实验报告

实验名称：	自然语言实验
姓　　名：	林子微
学　　号：	**22281279**
日　　期：	**2024 年 12 月 16 日**

目录

一、实验内容

1.1 实验介绍

文本分类是自然语言处理任务中常用的任务，主要是对一段文本序列进行分类，进行该任务的意义在于可以支持知识图谱构建过程中的关系分类任务、微博评论舆情识别任务、新闻标题分类任务等内容，具有很大的应用价值。

结合之前学习的故障分类任务思路、LSTM 循环神经网络、CNN 卷积神经网络等框架进行训练，也可以使用 Bert 预训练模型、Transformer 框架来提高分类的准确率。基本流程为：句子转换成向量、向量进行 embedding、搭建神经网络模型（可使用中文 Bert 预训练模型、LSTM、CNN、或者几个模型融在一起，也可以不用很麻烦，单纯使用 RNN 或 CNN 效果也会很好），最后转换成为多分类问题。

要求：①设计并搭建和训练神经网络模型；

②自己找开源的 Word2Vec 方法 或 调一些开源的包（例如 jieba 分词库） 或去 Huggingface 社区找开源的中文 Bert 预训练模型加到你们神经网络模型中来提升你的模型效果；

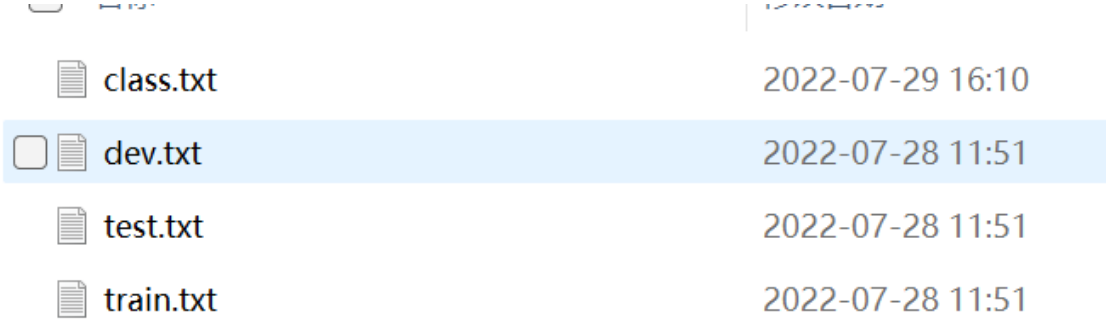
③使用 Sklearn 的评估函数 或 自己设置的分类衡量指标对最终的训练分类结果进行一个评估（如总体准确率、召回率、F1Score、10 个类别对应的准确率、召回率、F1Score 指标等）；

二、实验设计

2.1 数据来源

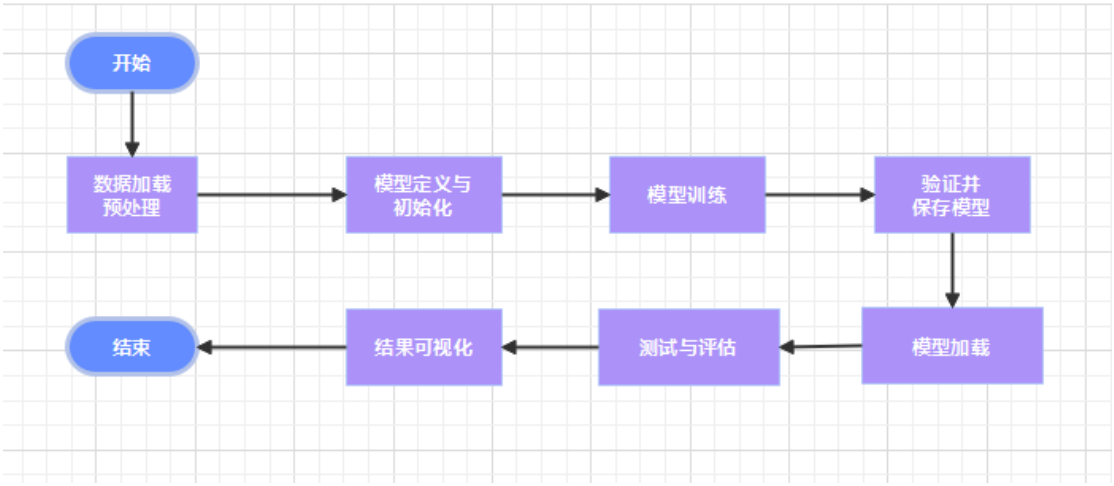
数据说明：

- 数据集：清华大学的 THUCNews 新闻文本分类数据集（子集），训练集 18w，验证集 1w，测试集 1w
- 10 个类别：金融、房产、股票、教育、科学、社会、政治、体育、游戏、娱乐



2.2 整体思路

流程图如下：



三、实验流程

3.1 数据加载与初始化

1. **库导入：**先导入了一系列必要的库。pandas 用于数据的读取和简单处理；torch 及其相关模块是构建深度学习模型、处理张量数据以及进行模型训练等操作的核心库；tqdm **用来显示训练进度条**，使训练过程可视化；matplotlib 用于绘制训练和验证结果相关的图表；numpy 辅助进行一些数值计算操作；os 用

于操作系统相关的文件和目录操作；datetime 用于获取时间戳信息，配合保存相关结果文件； **transformers 中的 GPT2Tokenizer 和 GPT2Config 用于加载预训练的 GPT2 模型相关配置并进行文本分词处理**；tqdm 用来**显示训练进度条**，使训练过程可视化；matplotlib 用于绘制训练和验证结果相关的图表；numpy 辅助进行一些数值计算操作；os 用于操作系统相关的文件和目录操作；datetime 用于**获取时间戳信息**，配合保存相关结果文件；

2.日志初始化：通过 **logging.basicConfig 函数**进行日志配置，将日志信息记录到名为 **training.log 的文件**中，设定日志级别为 **INFO**，意味着只会记录 INFO 级别及以上重要性的信息，并定义了日志的输出格式，包含时间、日志级别和具体消息内容，方便后续查看训练过程各阶段的详细情况。

```
# 初始化日志
logging.basicConfig(filename='training.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger()
```

3.数据加载函数定义与数据读取：定义了 load_data 函数，它使用 pandas 的 read_csv 方法读取以 \t 为分隔符且无表头的文本文件，并将读取的数据列命名为 **text（文本内容）和 label（对应的类别标签）**，同时对 text 列中的每个文本进行空白字符去除处理，最后返回处理好的数据框。接着分别调用该函数读取训练、验证和测试集数据，并通过日志记录了各数据集的大小信息，方便了解数据规模情况。

```
# 加载数据
def load_data(file_path):
    data = pd.read_csv(file_path, sep='\t', header=None, names=['text', 'label'])
    data['text'] = data['text'].str.strip()
    return data


train_data = load_data('./data/train.txt')
dev_data = load_data('./data/dev.txt')
test_data = load_data('./data/test.txt')
```

可以打印出训练集、验证集的前五条数据进行查看，以及训练验证测试集大小。

```
训练集前5条数据：
                          text  label
0          中华女子学院：本科层次仅1专业招男生      3
1     两天价网站背后重重迷雾：做个网站究竟要多少钱      4
2  东5环海棠公社230-290平2居准现房98折优惠      1
3  卡佩罗：告诉你德国脚生猛的原因 不希望英德战踢点球      7
4    82岁老太为学生做饭扫地44年获授港大荣誉院士      5

验证集前5条数据：
                    text  label
0  体验2D巅峰 倚天屠龙记十大创新概览      8
1   60年铁树开花形状似玉米芯(组图)      5
2       同步A股首秀：港股缩量回调      2
3   中青宝sg现场抓拍 兔子舞热辣表演      8
4            锌价难续去年辉煌      0

训练集大小： 180000
验证集大小： 10000
测试集大小： 10000
```

3.2 文本预处理

1. 分词器相关配置与加载：

使用了三种分词器尝试

（1）GPT-2（Generative Pre-trained Transformer 2）是一个强大的语言生成模型，它的 Tokenizer 基于 Byte Pair Encoding (BPE) 算法。**BPE 是一种子词分割算法，它能有效地将单词分解成常见的子字符串（或子词）**。对于未登录词或罕见词，它可以将其拆分成已知的子部分，从而提高了模型对未知词汇的处理能力。与 BERT 不同的是，GPT-2 的 Tokenizer 不仅限于字符级或者词级，而是介于两者之间的一种灵活分词方法。

先从预训练的 GPT2-124M 模型配置中加载基础配置信息到 **gpt2_config 对象**，然后对其进行自定义修改，设置 **num_hidden_layer=256**。

```
# 使用GPT-2的Tokenizer进行分词
gpt2_config = GPT2Config.from_pretrained('./GPT2-124M')
gpt2_config.num_hidden_layers = 256
gpt2_config.output_attentions = True
gpt2_config.output_hidden_states = True
tokenizer = GPT2Tokenizer.from_pretrained('./GPT2-124M', trust_remote_code=True, local_files_only=True)
```

并开启 output_attentions 和 output_hidden_states 选项，方便后续可能的分析或扩展使用。接着**实例化 GPT2Tokenizer**，通过指定模型路径、允许信任远程代码（trust_remote_code=True）以及仅使用本地文件（local_files_only=True）的方式加载分词器。

若分词器的 pad_token 不存在，则将其设置为 eos_token，并记录日志表明 GPT2 分词器已成功加载，这样在后续文本处理中就能正确进行填充操作了。

```
# 设置pad_token
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token  # 将pad_token设置为eos_token
logger.info("\nGPT2 tokenizer loaded\n")
```

（2）Jieba 是专门针对中文文本进行分词的 Python 库，它提供了简单接口来进行精确模式、全模式和搜索引擎模式的分词。Jieba 非常适合处理中文文本，因为它能够识别出词语边界，这在中文中不像英文有明确的空格分隔。此外，jieba 支持用户自定义词典，可以添加特定领域的词汇以提高分词准确性。

jieba_cut 函数接收一个字符串参数 text，然后使用 jieba.cut() 方法对其进行分词处理。

jieba.cut() 返回的是一个生成器，包含分词后的词语。通过 ' '.join(...) 将这些词语用空格连接成一个单独的字符串。这一步是为了让每个词语之间有明显的分隔符（如空格），方便后续处理或作为模型输入。

```
# 使用jieba分词
def jieba_cut(text):
    return ' '.join(jieba.cut(text))

train_data = load_data('./data/train.txt')
dev_data = load_data('./data/dev.txt')
test_data = load_data('./data/test.txt')

# 分词并转换为DataFrame，方便查看和操作
train_data = [(jieba_cut(text), label) for text, label in train_data]
dev_data = [(jieba_cut(text), label) for text, label in dev_data]
test_data = [(jieba_cut(text), label) for text, label in test_data]
```

对于每个数据集（训练集、开发集、测试集），使用列表推导式遍历原始数据列表中的每个 (text, label) 对，调用 jieba_cut 函数对文本部分进行分词处理，并保持标签不变。

将处理后的数据列表转换为 pandas 的 DataFrame，同时指定列名为 ['text', 'label']。这样做不仅可以让数据更易于查看，还便于后续进行数据分析、预处理和建模等工作。

（3）**BERT（Bidirectional Encoder Representations from Transformers）**是一个预训练语言模型，其 Tokenizer 主要用于将文本转换为模型可理解的输入格

式。对于中文版的 BERT（如 bert-base-chinese），它通常采用字符级别的分词方式，即将每个汉字视为一个单独的 token。这种方式简化了中文分词的问题，因为不需要像 jieba 那样去识别词语边界，而是直接对每个字符进行编码。

```
# 使用Bert - base - chinese的Tokenizer进行分词
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese')
logger.info("\nBERT tokenizer loaded\n")
```

特性/分词器	Jieba 分词	BERT 的 Tokenizer	GPT-2 的 Tokenizer
主要用途	中文文本分词	将文本转为BERT模型输入	将文本转为 GPT-2 模型输入
分词级别	词级（词语）	字符级（汉字）	子词级（BPE）
灵活性	支持用户自定义词典	固定分词规则，基于字符	动态学习分词规则,基于BPE
适用范围	主要适用于中文	适用于 BERT 系列模型，包括中文	适用于 GPT-2 系列模型
处理未登录词	可能会分割错误	每个汉字作为一个 token	能够有效处理，通过组合子词
是否需要预训练模型	不需要	需要加载预训练的 BERT 模型	需要加载预训练的 GPT-2 模型
配置文件路径	-	'bert-base-chinese'	'./GPT2-124M'

2. **自定义数据集类定义：** 创建了 NewsDataset 类，继承自 torch.utils.data.Dataset,这是 PyTorch 中用于自定义数据集的基类。在 __init__ 方法里，接收文本数据列表 texts、标签数据列表 labels、分词器 tokenizer 和最大文本长度 max_len 作为参数进行初始化，保存这些参数以便后续使用。__len__ 方法简单返回文本数据的数量,即数据集的大小。__getitem__ 方法是核心,对于给定的索引 item，它先获取对应索引的文本和标签，然后使用之前加载的 GPT2 分词器对文

本进行编码处理。

```
# 定义Dataset类
class NewsDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, item):
        text = self.texts[item]
        label = self.labels[item]
```

包括添加如[CLS]和[SEP]这样的特殊标记，按照设定的最大长度 MAX_LEN 进行截断或填充操作（**确保所有文本的长度一致，便于批量处理**），生成注意力掩码来指示哪些是真实的文本标记哪些是填充部分，最后将编码结果以 PyTorch 张量的形式返回，组织成一个包含 input_ids（文本的编码表示）、attention_mask（注意力掩码）和 labels（对应的类别标签张量）的字典，代表一个数据样本。

```
    # 使用GPT-2 tokenizer编码文本
    encoding = self.tokenizer.encode_plus(
        text,
        add_special_tokens=True,  # [CLS] and [SEP]
        max_length=self.max_len,
        padding='max_length',  # Padding to max_len
        truncation=True,  # Truncate if length exceeds max_len
        return_attention_mask=True,  # Attention mask
        return_tensors='pt',  # Return pytorch tensors
    )

    return {
        'input_ids': encoding['input_ids'].flatten(),
        'attention_mask': encoding['attention_mask'].flatten(),
        'labels': torch.tensor(label, dtype=torch.long)
    }
```

3．创建数据加载器：设定最大文本长度 **MAX_LEN 为** 128 并记录日志，之后基于 NewsDataset 类分别创建训练集、验证集和测试集对应的数据集实例，传入各自的文本和标签数据以及之前配置好的分词器和最大长度参数。接着利用 torch.utils.data.DataLoader 为每个数据集创建数据加载器，设置批量大小为 16；训练集设置 shuffle=True，在每个训练轮次开始时打乱数据顺序，增强模型的泛化能力，而验证集和测试集则设

置 shuffle=False，保持数据顺序不变；

设置 num_workers=4 表示启用 4 个工作线程来并行加载数据，加快数据读取速度，方便后续模型训练、验证和测试时高效地按批次获取数据。

```
# 定义最大文本长度
MAX_LEN = 128
logger.info(f"\n最大文本长度设为: {MAX_LEN}")

# 创建训练集、验证集和测试集的DataLoader
train_dataset = NewsDataset(train_data['text'].values, train_data['label'].values, tokenizer, MAX_LEN)
dev_dataset = NewsDataset(dev_data['text'].values, dev_data['label'].values, tokenizer, MAX_LEN)
test_dataset = NewsDataset(test_data['text'].values, test_data['label'].values, tokenizer, MAX_LEN)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=4)
dev_loader = DataLoader(dev_dataset, batch_size=16, shuffle=False, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False, num_workers=4)
```

3.2 RNN 模型

1. **模型初始化方法（__init__）：**
   - 词嵌入层定义：通过 self.embedding = nn.Embedding(**vocab_size, embedding_dim**) 创建了一个词嵌入层。nn.Embedding 层会将输入的离散的词汇索引（基于给定的 vocab_size，也就是**词汇表大小**）转换为**固定维度**（由 embedding_dim 指定，这里设为 128 维）的词向量表示，有助于模型学习到词汇的语义信息，是处理文本数据的常见操作。
   - 循环神经网络层定义：self.rnn = nn.RNN(**embedding_dim, hidden_dim, num_layers=n_layers,bidirectional=bidirectional, dropout=dropout, batch_first=True**) 定义了一个 RNN 类型的循环神经网络层。它接收词嵌入层输出的词向量（维度为 embedding_dim）作为输入，输出的隐藏状态维度为 hidden_dim（这里设为 256），网络层数由 n_layers（设为 3 层）指定，**bidirectional 参数设为 True 表示采用双向的 RNN**，这样可以同时捕捉文本的**正向和反向信息**，增强模型对文本上下文的理解能力。dropout 参数（设为 0.1）用于在训练过程中随机丢弃部分神经元连接，防止过拟合，batch_first=True 设定输入数据的第一个维度为批量大小，符合通常的数据组织形式。

```python
# 定义RNN模型
class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers, bidirectional, dropout):
        super(RNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim, num_layers=n_layers, bidirectional=bidirectional, dropout=dropout, batch_firs
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input_ids, attention_mask):
        embedded = self.embedding(input_ids)
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, attention_mask.sum(1).cpu(), batch_first=True, enforce_sorte
        packed_output, hidden = self.rnn(packed_embedded)
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output, batch_first=True)
        hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)) if self.rnn.bidirectional else self.dropout(hidd
        return self.fc(hidden)
```

- 全连接层定义：self.fc = nn.Linear(**hidden_dim * 2 if bidirectional else hidden_dim, output_dim**) 创建了一个全连接层，用于将 RNN 层输出的隐藏状态映射到最终的输出维度（output_dim，这里设为 10，对应分类类别数量）。如果是双向 RNN，则输入维度为 hidden_dim * 2,因为要拼接正向和反向的隐藏状态;若是单向 RNN，输入维度就是 hidden_dim。
- Dropout 层定义：self.dropout = nn.Dropout(dropout) 定义了一个 Dropout 层，用于在训练阶段随机将一些神经元的输出置为 0，以增加模型的泛化能力，在测试阶段该层不起作用，其 dropout 参数与 RNN 层中的 dropout 取值一致（设为 0.1）。

```python
# 定义模型参数
VOCAB_SIZE = tokenizer.vocab_size
EMBEDDING_DIM = 128
HIDDEN_DIM = 256
OUTPUT_DIM = 10
N_LAYERS = 3
BIDIRECTIONAL = True
DROPOUT = 0.1

# 初始化模型
model = RNN(VOCAB_SIZE, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM, N_LAYERS, BIDIRECTIONAL, DROPOUT)
```

日志文件中输出模型的结构如下：

```
2024-12-08 21:06:22,079 - INFO - Model:
RNN(
  (embedding): Embedding(50257, 128)
  (rnn): RNN(128, 256, num_layers=3, batch_first=True, dropout=0.1, bidirectional=True)
  (fc): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.1, inplace=False)
)
```

2. **前向传播方法（forward）：**
   - **词嵌入操作：**embedded = self.embedding(input_ids) 将输入的 input_ids（文本经过编码后的索引张量）通过之前定义的词嵌入层，得到对应的词向量表示 embedded。

- **序列打包：**packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, attention_mask.sum(1).cpu(), batch_first=True, enforce_sorted=False) 对经过词嵌入后的**变长序列数据进行打包处理**。由于输入的文本长度可能不一致，经过填充后存在变长情况，这个操作会根据 **attention_mask（注意力掩码，用于指示哪些是真实文本部分，哪些是填充部分）**计算每个样本的有效长度，并将序列数据**按照有效长度打包**，方便后续高效地输入到 RNN 层中进行计算，同时设置 batch_first=True 符合数据组织习惯，**enforce_sorted=False 表示输入的序列不需要提前按照长度排序**（如果设为 True 则需要手动排序）。
- **RNN 层计算与解包：**packed_output, hidden = self.rnn(packed_embedded) 将打包后的序列数据传入 RNN 层进行计算，得到 packed_output（打包后的输出序列）和 hidden（最终的隐藏状态）。然后通过 output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output, batch_first=True) 对输出序列进行解包填充操作，恢复成固定长度的张量形式 output 以及对应的长度信息 output_lengths，使其能够继续后续的处理。
- **隐藏状态处理与 Dropout ：**对于 RNN 层输出的隐藏状态 hidden，根据是否双向连接进行不同的处理。
  - 如果是双向 RNN，则通过 hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)) 将最后两个时间步（正向最后一个和反向最后一个）的隐藏状态在指定维度（dim=1）上进行拼接，并经过 Dropout 层；
  - 若是单向 RNN，则直接通过 hidden = self.dropout(hidden[-1,:,:]) 取最后一个时间步的隐藏状态并经过 Dropout 层。这样处理后的隐藏状态 hidden 包含了经过 RNN 层学习到的文本特征信息以及适当的正则化处理。
- **全连接层输出：**最后通过 return self.fc(hidden) 将处理后的隐藏状态传入全连接层 self.fc，得到最终的模型输出结果，这个结果可以是对应各个分类类别的得分或者概率（取决于后续的处理方式，如经过 Softmax 等激活函数），完成整个模型的前向传播过程。

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

# 如果有多个GPU，使用DataParallel
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)

# 输出模型结构
logger.info("Model:\n{}".format(model))

# 定义优化器和损失函数
optimizer = Adam(model.parameters(), lr=1e-4, betas=(0.9, 0.999), weight_decay=1e-5)

criterion = torch.nn.CrossEntropyLoss()
```

3.3 Transformer 模型

1. 嵌入层

功能：将输入的 **token 索引转换为固定维度的向量表示（即嵌入）**。这是从离散的词汇索引到连续向量空间的映射，使得模型能够处理和学习这些向量之间的关系。

过程：

- 输入是一批文本序列，每个序列由一系列 token 组成，token 已经被转换为词汇表中的索引。
- 嵌入层根据这些索引查找对应的嵌入向量，生成形状为 (batch_size, seq_len, embedding_dim) 的张量。这里 batch_size 是批次大小，seq_len 是序列长度，而 embedding_dim 则是每个 token 的嵌入维度。

2. 位置编码

功能：添加位置信息到嵌入向量中，**帮助模型理解序列中各个 token 的相对或绝对位置。**因为 Transformer 架构本身不包含对顺序的理解能力，所以位置编码是必要的。

过程：

- 使用正弦和余弦函数构造位置编码，其频率随位置的不同而变化。
- 这些位置编码与嵌入向量相加，使得每个 token 的表示不仅包含其内容信息，还包含了它在序列中的位置信息。

```python
class TransformerClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers, n_heads, dropout):
        super(TransformerClassifier, self).__init__()
        # 嵌入层，将输入的token索引转换为向量表示
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # Transformer编码器
        self.transformer = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(d_model=hidden_dim, nhead=n_heads, dim_feedforward=hidden_dim * 4, dr
            num_layers=n_layers
        )
        # 全连接层用于分类
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input_ids, attention_mask):
        # 获取嵌入后的向量表示，形状为(batch_size, seq_len, embedding_dim)
        embedded = self.embedding(input_ids)
        # 调整维度顺序以匹配Transformer输入要求，变为(seq_len, batch_size, embedding_dim)
        embedded = embedded.permute(1, 0, 2)
        # 对嵌入向量添加位置编码
        seq_len = embedded.size(0)
        position_encoding = self.get_position_encoding(seq_len, embedded.size(2)).unsqueeze(1).to(embedded.d
        embedded += position_encoding
        # 通过Transformer编码器
        transformer_output = self.transformer(embedded)
        # 取平均池化，形状变为(batch_size, hidden_dim)
        pooled_output = transformer_output.mean(dim=0)
        # 经过Dropout和全连接层得到最终输出
        output = self.fc(self.dropout(pooled_output))
        return output
```

3. Transformer 编码器

功能：对输入序列进行深度特征提取，捕捉序列中 token 之间的复杂依赖关系。

结构：

- 包含多个堆叠的 Transformer 层，每一层由一个多头自注意力机制（Multi-Head Attention）和一个前馈神经网络（Feed-Forward Network，FFN）组成。
- **多头自注意力机制：**允许模型在同一层内同时关注多个不同子空间中的信息，从而更好地捕捉 token 间的长距离依赖关系。
- **前馈神经网络：**用于进一步变换特征，通常包括两个线性变换层和中间的激活函数（如 ReLU），以增加模型的非线性表达能力。

过程：

- 输入经过嵌入和位置编码后，送入 Transformer 编码器，逐层处理以生成更高级别的特征表示。每一层都会更新 token 的表示，使其更加抽象且富含上下文信息。

4. 池化操作

功能：从 Transformer 编码器的输出中提取一个固定大小的特征向量，以便后续用于分类任务。

过程：

- 在这个模型中，采用的是**平均池化**（mean pooling），即将序列中所有 token 的最终表示取平均值。这样可以得到一个形状为 (batch_size, hidden_dim) 的张量，其中 hidden_dim 是 Transformer 编码器的隐藏状态维度。
- 平均池化是一种简单有效的方法，它假设每个 token 的贡献是平等的，并且可以很好地概括整个序列的信息。

```
def get_position_encoding(self, seq_len, embedding_dim):
    """
    简单的位置编码函数参考Transformer原论文实现方式
    """
    position_encoding = torch.zeros(seq_len, embedding_dim)
    div_term = torch.exp(torch.arange(0, embedding_dim, 2).float() * (-torch.log(torch.tensor(10000.0))
    position_encoding[:, 0::2] = torch.sin(torch.arange(0, seq_len).unsqueeze(1).float() * div_term)
    position_encoding[:, 1::2] = torch.cos(torch.arange(0, seq_len).unsqueeze(1).float() * div_term)
    return position_encoding
```

日志输出的模型结构如下：

```
2024-12-09 09:17:52,199 - INFO - Model:
TransformerClassifier(
  (embedding): Embedding(50257, 768)
  (transformer): TransformerEncoder(
    (layers): ModuleList(
      (0-1): 2 x TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
        )
        (linear1): Linear(in_features=768, out_features=3072, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=3072, out_features=768, bias=True)
        (norm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
  )
  (fc): Linear(in_features=768, out_features=10, bias=True)
  (dropout): Dropout(p=0.1, inplace=False)
)
```

详细流程如下：首先通过**嵌入层将输入的 token 索引转换为 768 维的词嵌入向量**，然后经过 2 层 Transformer 编码器，**每层包含多头注意力机制和前馈神经网络**，分别对输入进行特征提取。多头注意力机制的输入和输出维度为 768，并通过线性层将注意力机制的输出映射回 768 维。前馈神经网络包含**两个线性层**，输入 768 维，中间层 3072 维，输出 768 维，中间使用 Dropout（概率为 0.1）防止过拟合。每层还使用了归一化层（LayerNorm）对输入进行归一化，维度为 768，epsilon 为 1e-5。最后，通过全连接层将 Transformer 编码器的输出映射到 10

个类别，用于分类任务。模型中还使用了 Dropout（概率为 0.1）来防止过拟合。

说明：只使用了一层 Transformer 编码层，论文中使用 12 层，因此效果并不理想，但由于在 A4000 上跑一个 epoch 就要 1 小时，所以没有再继续增加层试验

3.4 Bert 模型

```
# 加载BERT模型
model = BertForSequenceClassification.from_pretrained('bert-base-chinese', num_labels=10)
```

1. 编码器

功能：编码器是 BERT 模型的核心部分，负责从输入序列中提取特征。它通过多头**自注意力机制和前馈神经网络**来捕捉序列中的上下文信息。

过程：

- 输入是经过嵌入层处理后的张量，形状为 (batch_size, seq_len, embedding_dim)。
- 编码器由 **12 层** BertLayer 组成，每层包含多头自注意力机制和前馈神经网络。
- 多头自注意力机制通过线性层将输入映射为**查询、键和值**，并使用 Dropout 防止过拟合。注意力机制的输出通过线性层和 LayerNorm 进行归一化。
- 前馈神经网络包含两个线性层，输入 768 维，中间层 3072 维，输出 768 维，中间使用 GELU 激活函数。前馈神经网络的输出同样通过 LayerNorm 和 Dropout 进行归一化和防止过拟合。
- 每一层的输出作为下一层的输入，逐步提取序列的高阶特征。

2. 池化层

功能：池化层用于从编码器的输出中提取一个固定维度的向量表示，通常用于分类任务。

过程：

- 输入是编码器的输出，形状为 (batch_size, seq_len, hidden_dim)。
- 池化层通过一个线性层将编码器的输出映射为 768 维，并通过 Tanh 激活

函数进行非线性变换，得到一个固定维度的向量表示。

- 这个向量表示了整个序列的全局特征，适合用于分类任务。

3. 分类器

功能：分类器将池化层的输出映射到具体的分类结果，通常是一个全连接层。

过程：

- 输入是池化层的输出，形状为 (batch_size, hidden_dim)。
- 分类器通过一个全连接层将池化层的输出映射到 10 个类别，用于分类任务。
- 模型中还使用了 Dropout（概率为 0.1）来防止过拟合。

4. 损失函数

功能：损失函数用于衡量模型预测结果与真实标签之间的差异，并通过反向传播更新模型参数。

过程：

- 输入是分类器的**输出（logits）和真实标签（labels）。**
- 根据任务类型（回归、单标签分类或多标签分类），选择合适的损失函数（如 MSELoss、CrossEntropyLoss 或 BCEWithLogitsLoss）。
- 计算损失并进行反向传播，更新模型参数。

日志输出的模型结构如下：

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(21128, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=10, bias=True)
)
```

这个模型结构是一个基于 BERT 的序列分类模型，名为 BertForSequenceClassification。它主要由 **BERT 模型和分类器组成**，BERT 模型负责对输入序列进行编码，提取特征，而分类器则将这些特征映射到具体的分类结果。

首先，BERT 模型的嵌入层将输入序列的 **token 索引、位置信息和 token 类型信息**转换为 768 维的向量。嵌入层包括三个部分：**词嵌入（word_embeddings）、位**

置嵌入（position_embeddings）和 token 类型嵌入（token_type_embeddings）。这些嵌入向量通过 LayerNorm 进行归一化，并通过 Dropout（概率为 0.1）防止过拟合。嵌入层的输出作为 BERT 编码器的输入。

接下来，BERT 编码器由 **12 层 BertLayer** 组成，每层包含**多头自注意力机制和前馈神经网络**。多头自注意力机制通过线性层将输入映射为**查询（query）、键（key）和值（value）**，并使用 Dropout（概率为 0.1）防止过拟合。注意力机制的输出通过线性层和 LayerNorm 进行归一化，确保特征的稳定性。前馈神经网络包含两个线性层，输入 768 维，中间层 3072 维，输出 768 维，中间使用 **GELU 激活函数进行非线性变换**。前馈神经网络的输出同样通过 LayerNorm 和 Dropout 进行归一化和防止过拟合。每一层的输出作为下一层的输入，逐步提取序列的高阶特征。

BERT 模型的池化层（pooler）将编码器的输出进行进一步处理。池化层通过一个线性层将编码器的**输出映射为 768 维**，并通过 Tanh 激活函数进行非线性变换，得到一个固定维度的向量表示。这个向量表示了整个序列的全局特征，适合用于分类任务。

最后，分类器通过一个**全连接层（classifier）将池化层的输出映射到 10 个类别**，用于分类任务。模型中还使用了 Dropout（概率为 0.1）来防止过拟合。整体结构能够捕捉序列的全局依赖关系，适用于文本分类、情感分析等任务，能够处理变长序列并输出多个类别的概率。此外还打印出其中的部分层的参数和维度进行查看：

```
# 输出模型结构
logger.info("Model:\n{}".format(model))

# 打印模型每层参数
for name, param in model.named_parameters():
    logger.info(f"Layer: {name}")
    logger.info(f"Parameter Shape: {param.shape}")
    logger.info(f"Requires Grad: {param.requires_grad}")
    logger.info("-" * 50)
```

```
2024-12-16 13:59:07,622 - INFO - Layer: bert.encoder.layer.9.attention.output.dense.weight
2024-12-16 13:59:07,622 - INFO - Parameter Shape: torch.Size([768, 768])
2024-12-16 13:59:07,622 - INFO - Requires Grad: False
2024-12-16 13:59:07,636 - INFO - ------------------------------------------------
2024-12-16 13:59:07,637 - INFO - Layer: bert.encoder.layer.9.attention.output.dense.bias
2024-12-16 13:59:07,637 - INFO - Parameter Shape: torch.Size([768])
2024-12-16 13:59:07,637 - INFO - Requires Grad: False
2024-12-16 13:59:07,637 - INFO - ------------------------------------------------
2024-12-16 13:59:07,637 - INFO - Layer: bert.encoder.layer.9.attention.output.LayerNorm.weight
2024-12-16 13:59:07,637 - INFO - Parameter Shape: torch.Size([768])
2024-12-16 13:59:07,638 - INFO - Requires Grad: False
2024-12-16 13:59:07,638 - INFO - ------------------------------------------------
2024-12-16 13:59:07,638 - INFO - Layer: bert.encoder.layer.9.attention.output.LayerNorm.bias
2024-12-16 13:59:07,638 - INFO - Parameter Shape: torch.Size([768])
2024-12-16 13:59:07,638 - INFO - Requires Grad: False
```

截取部分日志信息，BERT 模型中第 9 层编码器的注意力机制输出部分的参数信息。具体来说，列出了几个关键层的权重和偏置参数的形状以及**无需进行梯度更新。**

```
2024-12-16 13:59:07,697 - INFO - Layer: fc.weight
2024-12-16 13:59:07,697 - INFO - Parameter Shape: torch.Size([10, 768])
2024-12-16 13:59:07,697 - INFO - Requires Grad: True
2024-12-16 13:59:07,697 - INFO - ------------------------------------------------
2024-12-16 13:59:07,697 - INFO - Layer: fc.bias
2024-12-16 13:59:07,697 - INFO - Parameter Shape: torch.Size([10])
2024-12-16 13:59:07,697 - INFO - Requires Grad: True
2024-12-16 13:59:07,698 - INFO - ------------------------------------------------
2024-12-16 13:59:07,710 - INFO -
```

```python
# 冻结 BERT 参数
for param in model.bert.parameters():
    param.requires_grad = False
```

以上采用**预训练**的方法，只有最后的 fc 层参数允许更新。

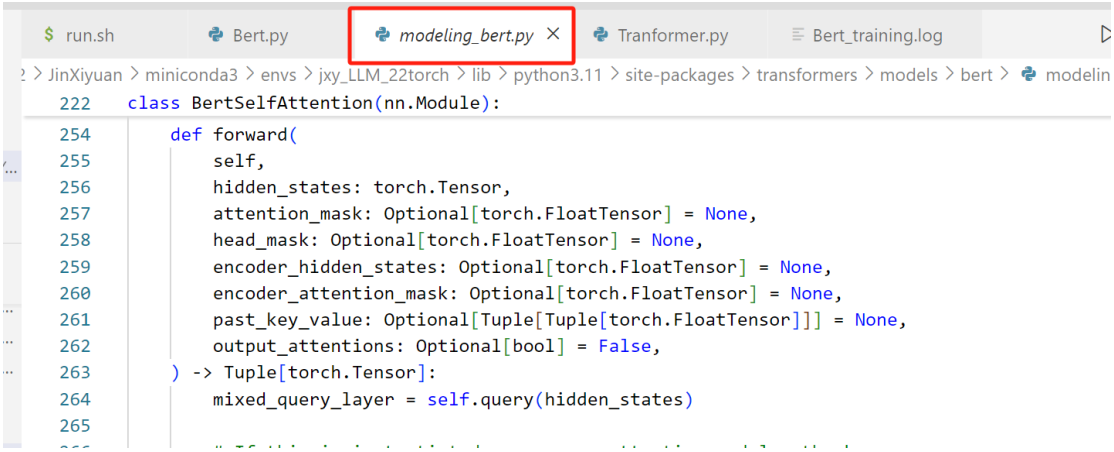
若使用**微调**，则不将 bert 主干冻结，本实验使用全量微调。

说明：刚开始实验时搞混了微调和预训练的定义，后来在助教的帮忙下解决，如果使用预训练的话，测试集最后准确率约 85.57%，但使用微调之后，准确率能提高到 94.52%

通过 bert 预训练和微调的对比可以发现**微调具有强大优势**，通过在特定任务上进一步训练预训练模型，使其能够将预训练阶段学到的知识迁移到目标任务中。但是由于微调要更新所有参数，所以所花的时间也会更长。

此外，在 Hugging Face 的 Transformers 库中，BertModel 和其他以 BertFor... 开头的类都是基于 BERT（Bidirectional Encoder Representations from Transformers）预训练模型的不同变体。它们的区别主要在于**每个类针对特定的**

自然语言处理任务进行了微调（fine-tuning），或者提供了特定任务所需的输出格式。



自行针对其中的几个模型进行了简要探究

1. BertModel

- **用途**：这是一个基础的 BERT 模型，通常用于**特征提取或作为其他任务的起点。**
- **输出**：它返回两个主要输出：
  - **最后一层的隐藏状态（hidden states）**：这是每个输入 token 的表示，可以用于下游任务。
  - **池化输出（pooled output）**：这是通过一个线性变换和激活函数从 [CLS] token 的隐藏状态得到的向量，通常用于句子级别的表示。
- **特点**：不包含任何特定任务的头部（head），因此需要用户自行添加或连接到其他模块以完成特定任务。

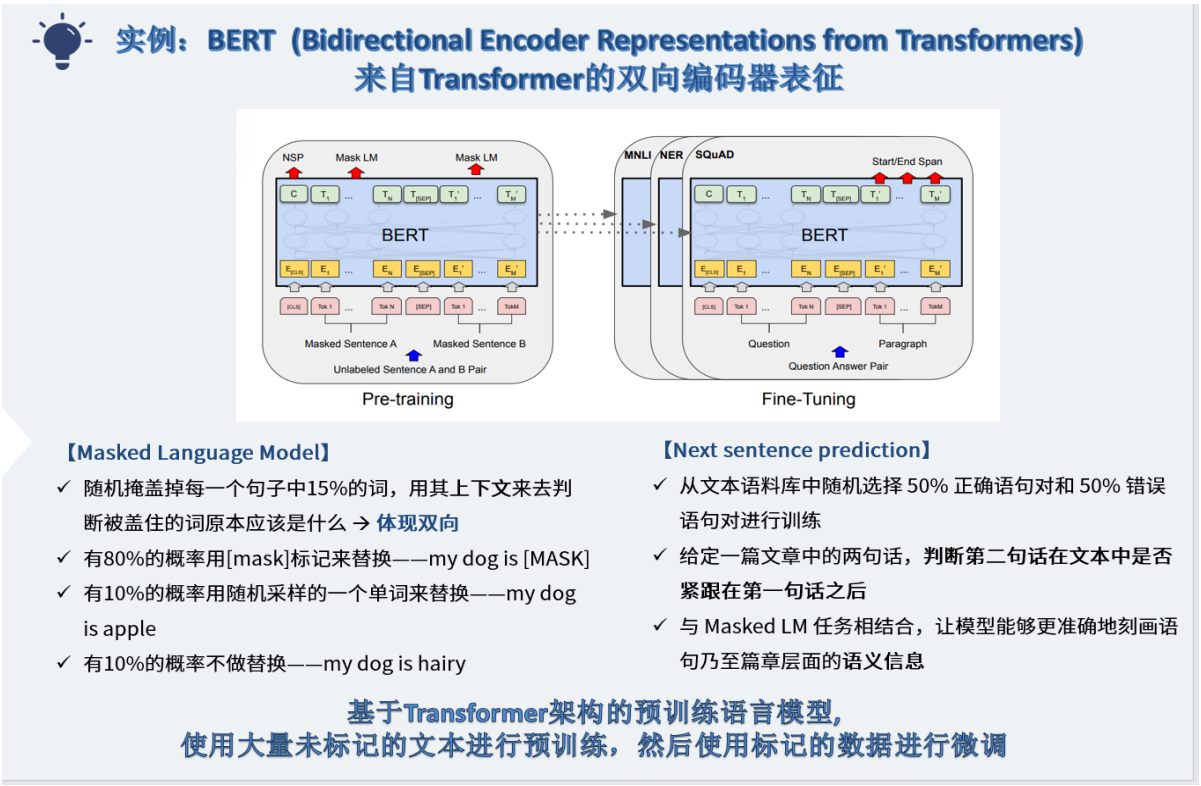
2. BertForSequenceClassification

- **用途**：专门用于**文本分类任务，如情感分析、意图识别等**。
- **结构**：在 BertModel 的基础上增加了一个分类头部（classification head），包括：
  - **全连接层（Dense Layer）**：将 BertModel 输出的 [CLS] token 表示映射到指定数量的类别。
  - **激活函数（Activation Function）**：例如 Softmax 或 Sigmoid，用于生成概率分布。

- **输出**：直接输出分类结果（logits），可以直接用于计算损失函数（如交叉熵损失）并进行预测。

3. BertForTokenClassification
   - **用途**：用于**序列标注任务，如命名实体识别（NER）、词性标注等。**
   - **结构**：同样基于 BertModel，但它为每个 token 添加了分类头部，包括：
     - **全连接层（Dense Layer）**：将每个 token 的隐藏状态映射到标签空间。
     - **激活函数（Activation Function）**：通常使用 Softmax 来生成每个 token 的标签概率分布。
   - **输出**：对于序列中的每个 token 输出其对应的标签（logits），适用于多标签分类任务。
4. BertForQuestionAnswering
   - **用途**：专门用于问答任务，如**抽取式问答（Extractive Question Answering）**，其中模型需要从给定的上下文中找到问题的答案。
   - **结构**：在 BertModel 的基础上增加了两个额外的输出层，分别用于预测答案的起始位置和结束位置：
     - **起始位置预测层（Start Position Prediction Layer）**：预测答案在上下文中的起始 token。
     - **结束位置预测层（End Position Prediction Layer）**：预测答案在上下文中的结束 token。
   - **输出**：给出上下文中每个 token 作为答案起始和结束位置的概率分布，从而可以确定最有可能的答案片段。

结合之前的课堂算法讲解，进一步认识到 Bert 系列模型功能的强大，



BERT 的**双向编码器架构**允许它同时考虑词的左右上下文信息，使得对句子的理解更加准确和全面。这种特性对于诸如问答系统、情感分析、命名实体识别等复杂任务至关重要，因为它能够捕捉到词语在不同语境下的细微差别。此外，Hugging Face 的 **Transformers 库提供了简单易用的接口**，让我们可以轻松加载预训练的 BERT 模型，并快速微调以适应自己的应用场景，极大地降低了开发门槛。

3.5 模型训练验证

通过定义模型、损失函数和优化器，使用**早停机制训练模型**，并在训练过程中记录和绘制**训练损失和验证损失的变化曲**线。这种设置和流程有助于有效地训练时间序列预测模型，并在验证集上监控模型的性能，防止过拟合。

1. 记录指标的列表初始化

首先创建了四个空列表 train_losses、train_accuracies、dev_losses、dev_accuracies，它们分别用于记录训练过程中每个 epoch 的训练损失、训练准确率以及验证过程中的验证损失、验证准确率。这些记录的数据后续可以用于分

析模型训练过程中的收敛情况以及判断模型是否出现过拟合等问题，例如通过观察训练损失和验证损失的变化趋势来评估模型的学习效果。

2.定义训练单个 epoch 的函数 train_epoch

- **训练模式设置与变量初始化**：定义了 train_epoch 函数，它接受模型、数据加载器、优化器、损失函数、设备（GPU 或 CPU）以及当前 epoch 数作为参数。函数内首先通过 model.train() 将模型设置为训练模式，这会启用一些在训练过程中需要的模块（如 Dropout 层等），同时初始化用于累计损失、正确预测数量和总样本数量的变量 total_loss、total_correct、total_samples，它们将在遍历数据批次的过程中不断更新。

```
# 使用 tqdm 显示进度条
for i, batch in enumerate(tqdm(data_loader, desc=f"Training Epoch {epoch + 1}", leave=False)):
    optimizer.zero_grad()

    input_ids = batch['input_ids'].to(device)
    attention_mask = batch['attention_mask'].to(device)
    labels = batch['labels'].to(device)
```

- **数据批次遍历与训练操作**：使用 tqdm 对数据加载器进行包装，这样在训练过程中会显示一个进度条，清晰展示训练进度，进度条的描述信息包含当前的训练轮次。对于每个批次的数据，先调用 optimizer.zero_grad() 来清零优化器的梯度，这是因为在反向传播时，梯度是累加的，每次反向传播前都需要将之前的梯度清零，避免影响本次的梯度计算。然后将批次中的 input_ids、attention_mask 和 labels 这三个数据张量移动到指定的设备（device）上，以确保数据和模型在同一设备上进行计算。

```
    # 前向传播
    outputs = model(input_ids, attention_mask)
    loss = criterion(outputs, labels)
```

```
    # 反向传播和优化
    loss.backward()
    optimizer.step()

    total_loss += loss.item()
```

- **前向传播、损失计算与反向传播优化**：接着进行前向传播，将处理好的输入数据传入模型得到输出 outputs，再根据模型输出和真实标签 labels 使用预先定义的损失函数 criterion 计算损失值 loss。之后调用 loss.backward() 进行反向传播，计算得到每个模型参数相对于损失的梯度，最后通过 optimizer.step() 根据计算出的梯度来更新模型的参数，完成

一次模型参数的优化调整过程。在这个过程中，还会累计该批次的损失值（通过 total_loss += loss.item()），用于后续计算平均损失。

- **准确率计算与指标记录：**通过 torch.max 函数获取模型输出中概率最大的类别索引作为预测结果 predictions，然后将预测结果与真实标签进行比较，统计正确预测的数量（通过 (predictions == labels).sum().item()）并累加到 total_correct 变量中，同时累计该批次的样本数量到 total_samples 变量。当遍历完整个训练数据加载器的所有批次后，计算该 epoch 的平均损失（avg_loss = total_loss / len(data_loader)）和准确率（accuracy = total_correct / total_samples），并将它们分别添加到 train_losses 和 train_accuracies 列表中，最后返回平均损失和准确率值，方便在主训练循环中进行记录或进一步分析。

```
        # 计算准确率
        _, predictions = torch.max(outputs, dim=1)
        correct = (predictions == labels).sum().item()
        total_correct += correct
        total_samples += labels.size(0)

    avg_loss = total_loss / len(data_loader)
    accuracy = total_correct / total_samples
    train_losses.append(avg_loss)
    train_accuracies.append(accuracy)
    return avg_loss, accuracy
```

3.定义模型评估函数 evaluate

- **函数初始化与模式设置：**
  - 函数 evaluate 用于在给定的数据加载器（可以是验证集或测试集的数据加载器）上评估模型的性能。首先将模型设置为评估模式（model.eval()），这会关闭一些在训练过程中启用但在评估时不需要的操作，例如 Dropout 层将不会随机丢弃神经元，保证模型在固定的参数状态下进行预测。然后初始化一些变量，包括**累计损失 total_loss、累计正确预测数量 total_correct、累计样本数量 total_samples，以及用于存储所有真实标签和预测结果的列表 all_labels 和 all_preds**。

```
def evaluate(model, data_loader, device, save_conf_matrix=False, save_dir='results'):
    model = model.eval()
    total_loss = 0
    total_correct = 0
    total_samples = 0
    all_labels = []
    all_preds = []
```

- **无梯度计算的前向传播与指标计算：**
  - 使用 torch.no_grad() 上下文管理器，确保在这个范围内进行的计算不会产生梯度，因为在评估阶段不需要进行反向传播更新模型参数。然后遍历数据加载器中的每个批次，将批次数据移动到指定设备后进行前向传播得到输出，接着根据模型输出和真实标签计算损失）并累计损失。
  - 同时，通过比较模型预测结果和真实标签计算正确预测的数量并累计以及将真实标签和预测结果分别添加到对应的列表中
- **综合评估指标计算与记录：**
  - 在遍历完所有批次后，计算模型在该数据集上的准确、平均损，并通过 sklearn.metrics 中的相关函数计算更详细的评估指标，如加权平均的精确率（precision_score）、召回率（recall_score）、F1 - score（f1_score），还会计算混淆矩阵（confusion_matrix）。将这些指标的值通过日志记录下来，方便查看模型的性能表现。

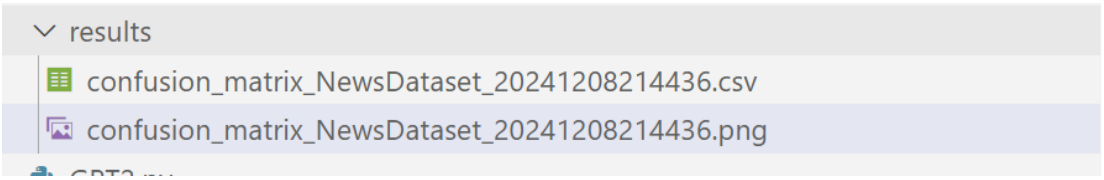
```
    logger.info(f"Precision: {precision:.4f}")
    logger.info(f"Recall: {recall:.4f}")
    logger.info(f"F1 - score: {f1:.4f}")
    logger.info(f"Confusion Matrix: \n{conf_matrix}")
```

- **混淆矩阵可视化与保存（可选）：**
  - 如果 save_conf_matrix 参数设置为 True，则会进行混淆矩阵的可视化和保存操作。首先检查保存目录是否存在，**若不存在则创建该目录。然后根据当前时间生成时间戳**，结合数据集名称等信息构造出用于保存混淆矩阵图像文件和 csv 文件的文件名。接着使用 matplotlib 库绘制混淆矩阵的可视化图像，设置图像大小、颜色映射、标题、坐标轴标签等信息，并在矩阵的每个格子中添加对应的数值文本（根据数值与阈值的比较设置文本颜色），最后保存图像并关闭绘图对象，同时还会将混淆矩阵数据以 csv 格式保存到对应的文件中。最后，将平均损失

和准确率添加到对应的记录列表 dev_losses 和 dev_accuracies 中，并返回准确率和分类报告。

```
if save_conf_matrix:
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    timestamp = datetime.datetime.now().strftime('%Y%m%d%H%M%S')
    img_file_name = os.path.join(save_dir, f'confusion_matrix_{data_loader.dataset.__class__.__name__}_{timestamp}.png')
    csv_file_name = os.path.join(save_dir, f'confusion_matrix_{data_loader.dataset.__class__.__name__}_{timestamp}.csv')

    plt.figure(figsize=(8, 8))
    plt.imshow(conf_matrix, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Confusion Matrix')
    plt.colorbar()
    tick_marks = np.arange(len(set(all_labels)))
    plt.xticks(tick_marks, tick_marks)
    plt.yticks(tick_marks, tick_marks)
```



4.主训练与验证循环及早停机制实现

- **训练轮次与相关变量初始化：**
  - 设定总的训练轮次 EPOCHS 为 10。初始化与验证集性能相关的几个重要变量，包括最佳验证准确率 best_dev_accuracy 初始值设为 0.0，早停机制的耐心值 patience 设为 3，用于记录验证集指标连续未提升次数的 no_improvement_count 初始化为 0，以及用于保存最佳模型状态字典的 best_model_state_dict 初始设为 None。

```
# 训练和评估模型，添加日志记录和进度条
EPOCHS = 10
best_dev_accuracy = 0.0
patience = 3
no_improvement_count = 0
best_model_state_dict = None
```

- **多轮训练与验证过程：**
  - 通过循环进行多轮训练，每轮训练开始时先记录日志表明开始训练当前轮次。接着调用 train_epoch 函数进行该轮的训练操作，完成一个 epoch 的训练。之后每隔 2 个 epoch 进行一次模型验证。当满足验证条件时，记录日志表示开始评估当前轮次，然后调用 evaluate 函数对模型在验证集上进行评估，传入相关参数

并获取验证准确率 dev_accuracy 和验证集分类报告 dev_report，记录验证准确率等相关日志信息。

```
for epoch in range(EPOCHS):
    logger.info(f"\n开始训练第 {epoch + 1} 轮：")

    # 训练阶段
    train_epoch(model, train_loader, optimizer, criterion, device, epoch)

    # 每隔 2 个 epoch 进行验证
    if (epoch + 1) % 2 == 0:
        logger.info(f"开始评估第 {epoch + 1} 轮：")
        dev_accuracy, dev_report = evaluate(model, dev_loader, device, save_conf_matrix=True)
        logger.info(f"第 {epoch + 1} 轮验证准确率：{dev_accuracy:.4f}")
        logger.info(f"验证集分类报告：\n{dev_report}")
```

- **早停机制检查与处理：**
  - 在每次验证后，进行早停机制的检查。如果当前验证准确率 dev_accuracy 高于之前记录的最佳验证准确率 best_dev_accuracy，说明模型在验证集上的性能有提升，则更新 best_dev_accuracy 为当前验证准确率的值，将 no_improvement_count 重置为 0，并保存当前模型的状态字典到 best_model_state_dict 中；若验证准确率没有提升，no_improvement_count 加 1，当 no_improvement_count 达到设定的 patience 值（即连续 3 次验证集指标未提升）时，判定模型在验证集上出现性能停滞，可能存在过拟合情况，此时记录日志提示提前停止训练，结束整个训练循环。

```
    # 早停机制检查
    if dev_accuracy > best_dev_accuracy:
        best_dev_accuracy = dev_accuracy
        no_improvement_count = 0
        # 保存当前最好模型的状态字典
        best_model_state_dict = model.state_dict()
    else:
        no_improvement_count += 1
        if no_improvement_count >= patience:
            logger.info(f"验证集指标连续 {patience} 次未提升，提前停止训练。")
            break
```

- **保存最好模型：**
  - 在训练完成后，如果 best_model_state_dict 不为 None，说明在训练过程中**找到了在验证集上表现最好的模型**。此时将该模型的状态字典保存到文件 best_trained_model.pth 中，并通过日志记录保存的路径信

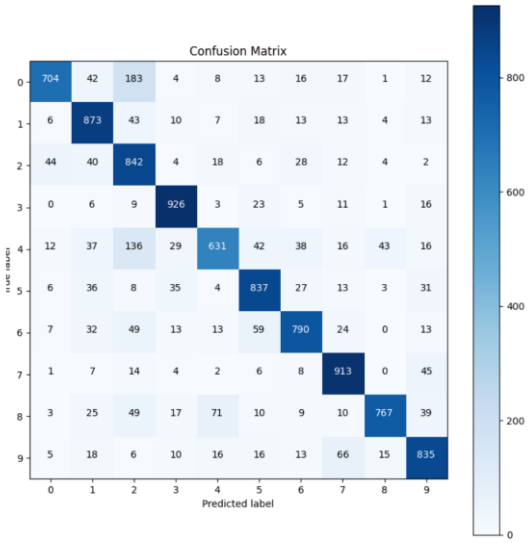
息。这样可以在后续需要时加载这个最佳模型进行进一步的应用或分析。

```
# 保存最好的模型
if best_model_state_dict is not None:
    model_path = "best_trained_model.pth"
    torch.save(best_model_state_dict, model_path)
    logger.info(f"最好的模型已保存至 {model_path}")
```

3.6 分类结果展示

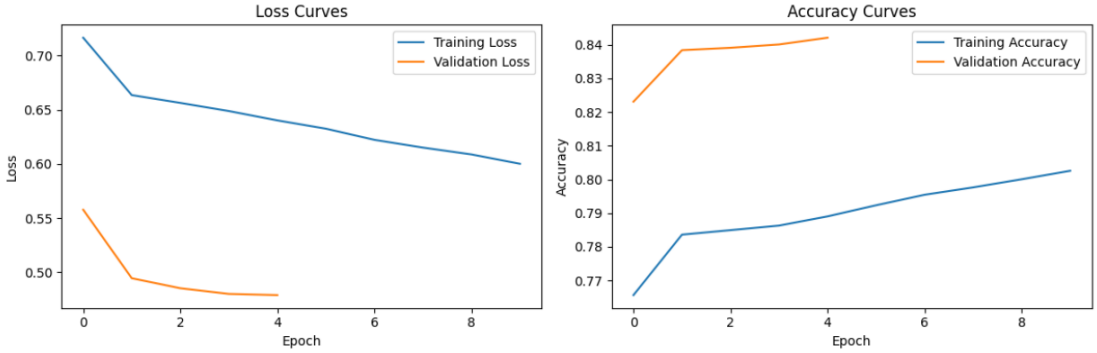
1. RNN 模型

测试集上准确率 88.74%



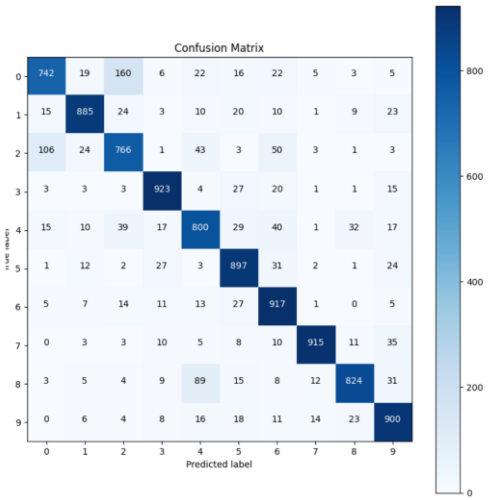
2024-12-09 01:48:20,506 - INFO - 测试集准确率: 0.8874
2024-12-09 01:48:20,508 - INFO - 测试集分类报告:

	precision	recall	f1-score	support
0	0.8936	0.8570	0.8749	1000
1	0.9384	0.8990	0.9183	1000
2	0.7991	0.8590	0.8280	1000
3	0.9340	0.9340	0.9340	1000
4	0.8585	0.8010	0.8288	1000
5	0.8377	0.9190	0.8765	1000
6	0.9007	0.8440	0.8715	1000
7	0.9453	0.9330	0.9391	1000
8	0.9025	0.9160	0.9092	1000
9	0.8778	0.9120	0.8946	1000
accuracy			0.8874	10000
macro avg	0.8888	0.8874	0.8875	10000
weighted avg	0.8888	0.8874	0.8875	10000



2. Transformer 模型

测试集上准确率 91.18%

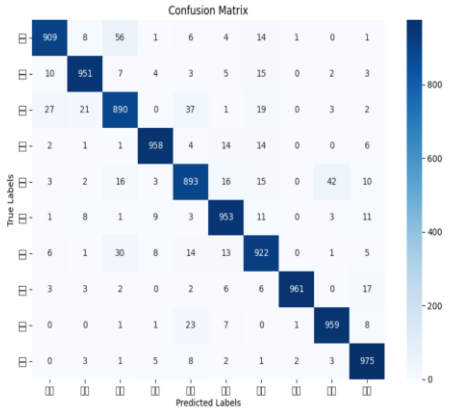


2024-12-09 19:10:20,794 - INFO - 测试集准确率: 0.9118
2024-12-09 19:10:20,795 - INFO - 测试集分类报告:

	precision	recall	f1-score	support
0	0.8698	0.8550	0.8623	1000
1	0.8562	0.9110	0.8828	1000
2	0.8231	0.7540	0.7871	1000
3	0.9489	0.9290	0.9389	1000
4	0.7850	0.8290	0.8064	1000
5	0.8515	0.8600	0.8557	1000
6	0.8308	0.8690	0.8495	1000
7	0.9533	0.9390	0.9461	1000
8	0.9171	0.8960	0.9064	1000
9	0.8902	0.8760	0.8831	1000
accuracy			0.9018	10000
macro avg	0.9026	0.9018	0.9018	10000
weighted avg	0.9026	0.9018	0.9018	10000

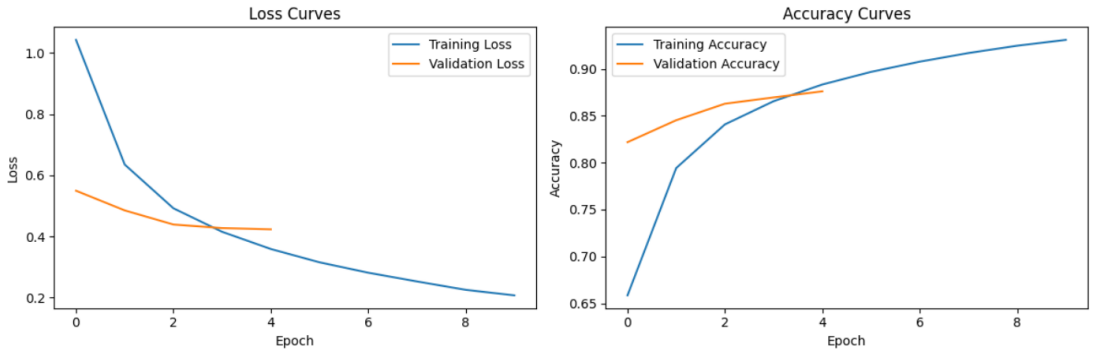
3. Bert 模型（微调）

测试集上准确率 94.52%



2024-12-16 19:10:20,794 - INFO - 测试集准确率: 0.9452
2024-12-16 19:10:20,795 - INFO - 测试集分类报告:

	precision	recall	f1-score	support
0	0.9298	0.9550	0.9423	1000
1	0.8962	0.9710	0.9328	1000
2	0.8831	0.9040	0.8934	1000
3	0.9889	0.9890	0.9889	1000
4	0.8850	0.9290	0.9064	1000
5	0.9115	0.9300	0.9207	1000
6	0.8808	0.9690	0.9225	1000
7	0.9833	0.9890	0.9861	1000
8	0.9571	0.9460	0.9514	1000
9	0.9302	0.9760	0.9526	1000
accuracy			0.9452	10000
macro avg	0.9426	0.9418	0.9418	10000
weighted avg	0.9426	0.9418	0.9418	10000



可以看到 Bert 模型在文本分类任务中表现最好，原因总结如下

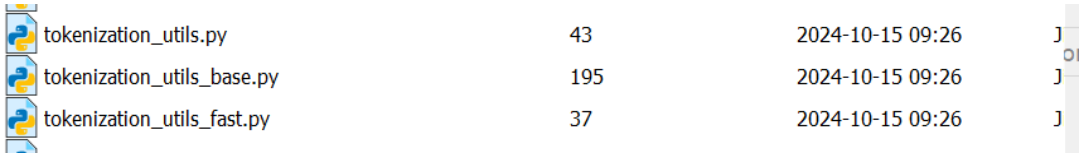
特性	BERT	RNN	Transformer
上下文建模	双向上下文建模，同时捕捉左右上下文信息。	单向建模，只能从前到后或从后到前处理文本。	多头自注意力机制，但 BERT 的双向设计更优。
预训练与迁移学习	基于大规模语料库预训练，微调后适应性强。	需要从头训练，缺乏预训练优势。	可以预训练，但 BERT 的预训练策略更成熟。
注意力机制	多头自注意力机制，并行处理，捕捉长距离依赖。	顺序模型，依赖时间步递归计算，难以并行化。	多头自注意力机制，但 BERT 的双向设计更优。
计算效率	并行计算，效率高。	顺序计算，效率低，处理长序列时易出现梯度问题。	并行计算，效率较高，但 BERT 的双向设计更优。
任务适应性	多种预训练任务（如 MLM 和 NSP），适应性强。	需要针对特定任务定制化设计。	可以适应多种任务，但 BERT 的预训练策略更成熟。
实验验证	在多个基准测试中表现优异，广泛验证。	在某些任务中表现良好，但不如 BERT。	在某些任务中表现良好，但 BERT 的双向设计更优。
社区支持	开源社区提供了丰富的实现和预训练模型，便于快速应用。	社区支持较少，实现和优化相对复杂。	社区支持较多，但 BERT 的预训练模型更丰富。
文本分类表现	通常表现最好，得益于双向上下文和预训练。	表现不如 BERT，受限于单向建模和计算效率。	表现较好，但 BERT 的双向设计和预训练策略使其更优。

四、问题及解决

问题一：无法加载名为的分词器（tokenizer）

```
Traceback (most recent call last):
  File "/mnt/nfsData9/JinXiyuan/6_LLM/SimMTM(org)/SimMTM_Forecasting/1/1.py", line 23, in <module>
    tokenizer = BertTokenizer.from_pretrained('bert-base-chinese')
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/data/JinXiyuan/miniconda3/envs/jxy_LLM_22torch/lib/python3.11/site-packages/transformers/tokenization_utils_base.py", li
ne 2094, in from_pretrained
    raise EnvironmentError(
OSError: Can't load tokenizer for 'bert-base-chinese'. If you were trying to load it from 'https://huggingface.co/models', make s
ure you don't have a local directory with the same name. Otherwise, make sure 'bert-base-chinese' is the correct path to a direct
ory containing all relevant files for a BertTokenizer tokenizer.
```

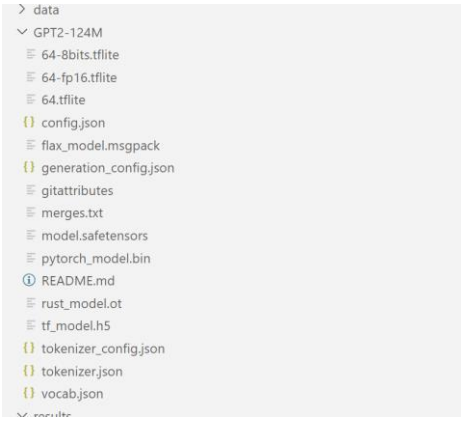
解决方法：定位到相关文件，发现之前下载过同名的本地模型，选择GPT2Tokenizer



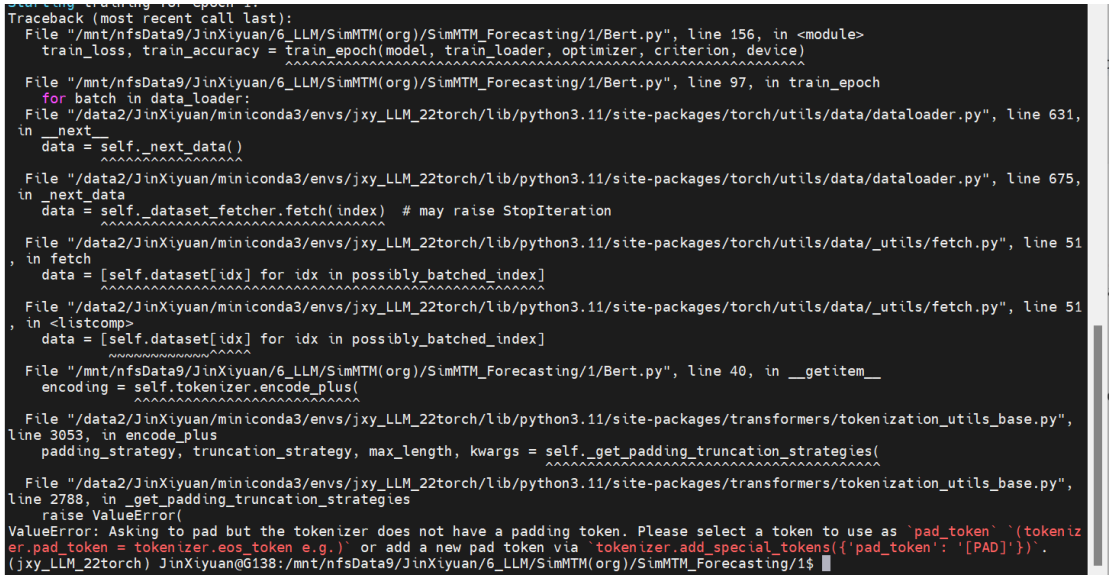

```python
# 使用GPT-2的Tokenizer进行分词
gpt2_config = GPT2Config.from_pretrained('./GPT2-124M')
gpt2_config.num_hidden_layers = 256
gpt2_config.output_attentions = True
gpt2_config.output_hidden_states = True
tokenizer = GPT2Tokenizer.from_pretrained('./GPT2-124M', trust_remote_code=True, local_files_only=True)

# 设置pad_token
```

将所需文件配置到目录下，在代码中加载



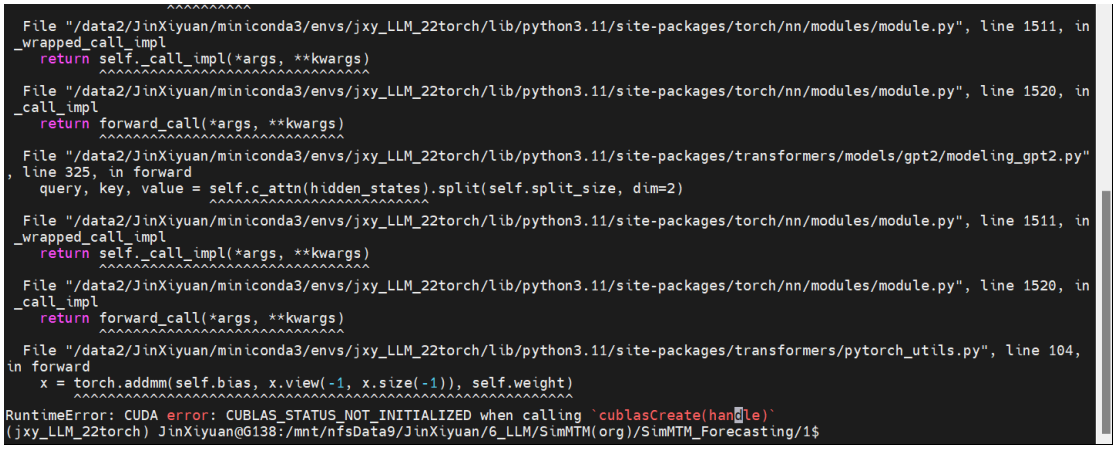
问题二：提示 tokenizer 没有设置 pad_token，但在数据处理过程中需要进行填充（padding）操作。



解决方法： 在初始化 tokenizer 后，添加一个新的 pad_token。文本数据通常需要进行填充（padding）以确保所有输入序列的长度一致。填充操作通常使用一个特殊的填充标记（pad token）来填充较短的序列，使其与最长序列的长度相同。

```
gpt2_config = GPT2Config.from_pretrained('./GPT2-124M')
gpt2_config.num_hidden_layers = 256
gpt2_config.output_attentions = True
gpt2_config.output_hidden_states = True
tokenizer = GPT2Tokenizer.from_pretrained('./GPT2-124M', trust_remote_code=True,
                                          local_files_only=True)
# 检查并添加填充令牌
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})
print("\nGPT2 tokenizer loaded\n")
```

问题三： 报错信息为 CUDA_STATUS_NOT_INITIALIZED when calling 'cublasCreate(handle)'。这表明在调用 cublasCreate(handle)函数时，CUDA 状态未初始化。



解决方法：检查 GPU 驱动，确保 GPU 驱动是最新的并且与 CUDA 版本兼容，确保使用的 PyTorch 版本与 CUDA 版本兼容。接下来检查下 Python 代码中，是否有不正确的 CUDA 调用。

run.sh 脚本文件配置如下：

```
$ run.sh
1   #!/bin/bash
2
3   export CUDA_VISIBLE_DEVICES=0
4
```

使用 **nvidia - smi 命令**来查看 GPU 的使用情况，发现有其他程序占用了大量 GPU 资源，可以尝试关闭这些程序后再运行代码。

```
Sun Dec  8 22:22:36 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.86       Driver Version: 470.86       CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA RTX A4000    Off  | 00000000:43:00.0 Off |                  Off |
| 41%   48C    P2    36W / 140W |   4825MiB / 16117MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA RTX A4000    Off  | 00000000:44:00.0 Off |                  Off |
| 41%   49C    P2    39W / 140W |   2763MiB / 16117MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  NVIDIA RTX A4000    Off  | 00000000:45:00.0 Off |                  Off |
| 41%   47C    P2    36W / 140W |   2763MiB / 16117MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  NVIDIA RTX A4000    Off  | 00000000:46:00.0 Off |                  Off |
| 41%   48C    P2    37W / 140W |  12682MiB / 16117MiB |     17%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      2625      G   /usr/lib/xorg/Xorg                  4MiB |
|    0   N/A  N/A    116006      G   ...ersions/Base75689/SC2_x64      121MiB |
|    0   N/A  N/A    189784      G   ...ersions/Base75689/SC2_x64      121MiB |
|    0   N/A  N/A   3206108      C   python3                          1009MiB |
|    0   N/A  N/A   3995447      G   ...ersions/Base75689/SC2_x64      121MiB |
|    0   N/A  N/A   4031387      C   python3                           371MiB |
|    0   N/A  N/A   4069237      C   python                           2947MiB |
|    0   N/A  N/A   4097917      G   ...ersions/Base75689/SC2_x64      121MiB |
|    1   N/A  N/A      2625      G   /usr/lib/xorg/Xorg                  4MiB |
|    1   N/A  N/A   4069237      C   python                           2755MiB |
|    2   N/A  N/A      2625      G   /usr/lib/xorg/Xorg                  4MiB |
|    2   N/A  N/A   4069237      C   python                           2755MiB |
|    3   N/A  N/A      2625      G   /usr/lib/xorg/Xorg                  4MiB |
|    3   N/A  N/A   3163983      C   python3                          2243MiB |
|    3   N/A  N/A   3167555      C   python3                          3133MiB |
|    3   N/A  N/A   3171961      C   python3                          4563MiB |
|    3   N/A  N/A   4069237      C   python                           2735MiB |
+-----------------------------------------------------------------------------+
```

问题四：使用 bert 模型之后输出分类结果很差

解决方法：一步一步排查错误，先从数据部分开始检查，print 出数据的类别分布、输入数据维度以及在训练过程中的数据维度，发现都没有问题

```
# 打印训练集、验证集和测试集的类别分布
# def print_class_distribution(data, name):
#     class_counts = data['label'].value_counts()
#     logger.info(f"{name} 类别分布: \n{class_counts}")

# print_class_distribution(train_data, "训练集")
# print_class_distribution(dev_data, "验证集")
# print_class_distribution(test_data, "测试集")
```

```
    )
    # 打印输入数据的维度
    # labels shape: ()
    # Input IDs shape: torch.Size([1, 128])
    # Attention Mask shape: torch.Size([1, 128])
    return {
```

```
    input_ids = batch['input_ids'].to(device)
    attention_mask = batch['attention_mask'].to(device)
    labels = batch['labels'].to(device)
    # Input IDs shape: torch.Size([16, 128])
    # Attention Mask shape: torch.Size([16, 128])
    # Labels shape: torch.Size([16])
```

询问助教后发现微调时忘记冻结 bert 参数，增加以下代码解决

```
# 冻结 BERT 参数
for param in model.bert.parameters():
    param.requires_grad = False
```

此外还有一些在服务器运行过程中的报错，例如 to.device（）忘记添加、传入的类型不匹配、数据精度问题等等，都在上网查询资料后解决。

五、总结及收获

我的心得体会如下：

1. 自然语言处理任务的进一步理解:。自然语言处理领域汇聚了众多不同类型的技术，从传统的机器学习算法到新兴的深度学习架构，每一种技术都有其独特的优势。例如，在数据预处理阶段，采用先进的词向量表示方法（如 Word2Vec）可以将文本中的语义信息有效地转化为计算机能够理解的向量形式。而在模型构建环节，将不同的神经网络架构进行融合，可以充分发挥它们各自在处理序列数据和提取局部特征方面的长处，进而提高模型的整体性能。这种跨技术的融合与创新不仅能够突破单一技术的局限性，还能为解决复杂的自然语言处理问题提供全新的思路和方法。

2. 模型搭建与训练：在实验中，自己设计并搭建神经网络模型是一个挑战且有意思的过程。从数据预处理，包括将句子转换为向量、进行 embedding 操作，到构建网络结构，每一步都需要精心设计和调试。同时，训练模型需要考虑很多超参数的设置，如学习率、批大小、迭代次数等，这些参数的选择直接影响模型的训练效果和收敛速度。在寻找合适的预训练模型和工具包方面，我体会到开源资源的丰富性和重要性，可以帮助我们提升模型的效果。合理利用这些资源可以大大节省模型训练的时间和成本，但同时也需要对这些工具和模型有深入的了解，才能正确地应用到自己的项目中。

3. 实践能力的提高:在实验过程中，不可避免地会遇到各种各样的问题，例如模型过拟合、训练不收敛、数据预处理错误等。这些问题需要通过仔细观察模型的训练过程、分析损失函数和准确率的变化趋势、检查数据的质量和处理流程等方法来解决。每一次解决问题的过程都是一次学习和成长的机会，让我学会了如何从复杂的现象中找到问题的根源，并运用所学知识和经验找到有效的解决方案。这种问题解决能力的提升不仅对自然语言处理实验有帮助，也将对我今后从事其他相关领域的研究和开发工作产生积极的影响。