# 新闻文本分类实验报告

**课程名称**：机器学习

**课程类别**：专业选修课

**任课教师**：文勇

**授课时间**：2024 年 3 月 8 日至 2024 年 7 月 5 日

**学　　号**：202312143002062

**姓　　名**：顾佳凯

**专业名称**：计算机科学与技术

**所在学院**：人工智能学院

# 目录

# 1.实验报告声明

在新闻文本分类实验的初始阶段,我选择了 Google Colab 作为实验平台。Colab 以其免费、便捷的特点而备受青睐,付费的 Pro 版本,承诺更稳定的连接和更强大的硬件支持。我原本期望通过升级到 Pro 版本,能够利用其提供的 A100 显卡,高效地训练我的模型,并避免长时间运行导致的断连问题。

然而,在实际的实验过程中,即便是使用了 Colab Pro,我仍然遭遇了断连的困扰。在模型训练超过一小时后,Colab 还是出现了自动断开的情况,给实验进度带来了显著的负面影响。这一发现促使我重新评估实验平台的选择。

经过深入的研究和比较,我最终决定将实验环境迁移到国内的 AutoDL 平台。尽管阿里云的天池实验室也提供了类似的服务,但考虑到个人的使用习惯以及后续实验的需求,AutoDL 平台以其更加优惠的价格和适配的环境脱颖而出。这一决策不仅确保了实验的顺利进行,也为后续的研究工作奠定了坚实的基础。

原本我用 20 万条训练数据训练了 5 个轮次的模型，提交的结果为 0.8637 的 score,之后想要尝试用 autodl 平台跑 30 个轮次来训练模型，但是需要整天电脑不关机不灭屏,打开 autodl 的 jupyter lab 的网页。最后我选择了先用 2 万条训练数据训练一个基础模型,紧接着在此基础上用剩余的 18 万条数据来微调模型，微调采用分批次微调，这样子跑了半天，剩余的半天可以去忙其他的事情。

经过数日在 AutoDL 平台上的不懈努力,我的模型在测试集上取得了一定的进步,最终获得了 0.9254 的 score,这一成绩让我感到十分欣慰。尽管与榜首仍有些许差距,但此次实验已让我对新闻文本分类任务的完整流程有了深入的理解和实践。透过亲身参与,我掌握了文本分类问题的核心思想和方法,这为日后深入研究奠定了坚实的基础。

# 2.实验代码解析

首先，我导入了 os 模块，并设置了代理服务器。具体来说，通过设置 http_proxy 和 https_proxy 环境变量，我将所有 HTTP 和 HTTPS 请求都通过 http://127.0.0.1:7890 这个代理服务器进行转发。这样做的目的是为了利用代理服务器的功能，例如访问被限制的资源或进行网络调试。以下是相关代码：

```
[4]: import os

os.environ['http_proxy'] = 'http://127.0.0.1:7890'
os.environ['https_proxy'] = 'http://127.0.0.1:7890'
```

图一：代码块 1

接下来，我使用 curl 命令请求 ipinfo.io 的 API 来获取当前 IP 地址的详细信息。curl 是一个常用的命令行工具，用于从服务器获取或发送数据。在这里，我访问 ipinfo.io/json 端点，该端点会返回关于当前 IP 地址的详细信息，包括 IP 地址、主机名、城市、地区、国家、经纬度、组织名称、邮政编码和时区等。返回的结果是一个 JSON 格式的数据，这些信息有助于了解当前的网络位置和相关的详细信息。以下是相关代码：

```
[5]: !curl ipinfo.io/json
{
  "ip": "206.190.235.48",
  "hostname": "206.190.235.48.16clouds.com",
  "city": "Osaka",
  "region": "Ōsaka",
  "country": "JP",
  "loc": "34.6938,135.5011",
  "org": "AS25820 IT7 Networks Inc",
  "postal": "543-0062",
  "timezone": "Asia/Tokyo",
  "readme": "https://ipinfo.io/missingauth"
}
```

图二：代码块 2

通过这些代码，我可以清楚地了解当前网络环境，并在实验中灵活应用代理服务器的功能。

以下这部分代码主要是进行数据的读取、基本统计分析和可视化。

[6]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# 读取训练集和测试集数据
train_df = pd.read_csv('/root/autodl-tmp/train_set.csv', sep='\t')
test_df = pd.read_csv('/root/autodl-tmp/test_a.csv', sep='\t')

print(f"Training set shape: {train_df.shape}")
print(f"Test set shape: {test_df.shape}")

# 查看训练集标签分布
train_df['label'].value_counts().plot(kind='bar')
plt.title("Training Set Label Distribution")
plt.xlabel("Label")
plt.ylabel("Number of Samples")
plt.show()

# 统计训练集文本长度
train_df['text_len'] = train_df['text'].apply(lambda x: len(x.split(' ')))
print(train_df['text_len'].describe())

# 可视化文本长度分布
plt.hist(train_df['text_len'], bins=100, range=(0, 3000))
plt.title("Training Set Text Length Distribution")
plt.xlabel('Text Length')
plt.ylabel('Number of Samples')
plt.show()
```

图三：代码块 3

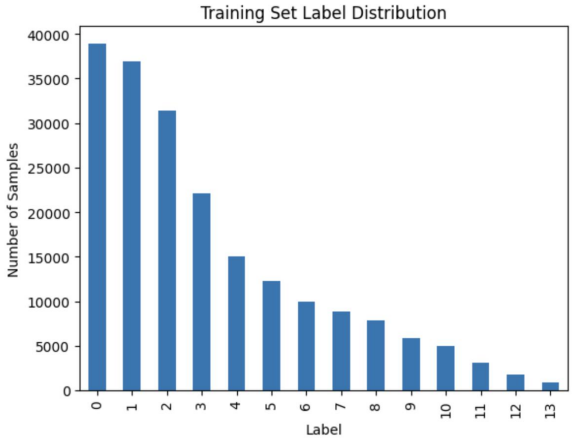
首先，我导入了必要的库，包括 pandas、numpy 和 matplotlib。pandas 用于数据操作和分析，numpy 提供支持数组操作的工具，而 matplotlib 则用于数据的可视化。接着，我读取训练集和测试集的数据，并打印它们的形状。通过 pandas 的 read_csv 方法，我从指定路径读取 CSV 文件，并指定分隔符为制表符（\t）。然后，我使用 shape 属性查看数据集的大小。

接下来，我查看训练集标签的分布情况。首先，我使用 value_counts 方法统计每个标签出现的次数，然后使用 plot 方法绘制柱状图，显示各标签的样本数量分布。接着，我计算并打印训练集中文本的长度分布。通过对文本列进行处理，我计算每个文本的单词

数量，并使用 describe 方法输出统计信息。最后，我使用直方图展示文本长度的分布情况，设置了 100 个柱和范围为 0 到 3000 的长度区间。

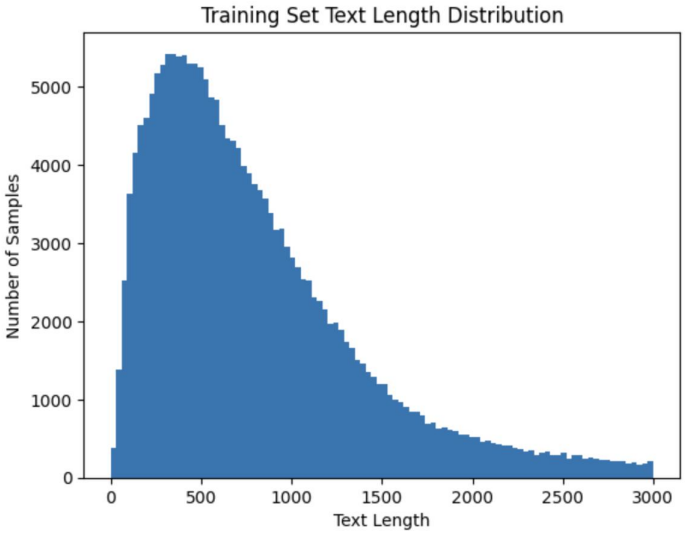
通过这些代码，我能够初步了解训练集和测试集的数据结构、标签分布以及文本长度分布，为后续的数据处理和模型训练提供了重要的信息。

```
Training set shape: (200000, 2)
Test set shape: (50000, 1)
```




```
count    200000.000000
mean        907.207110
std         996.029036
min           2.000000
25%         374.000000
50%         676.000000
75%        1131.000000
max       57921.000000
Name: text_len, dtype: float64
```

图四：训练集标签的分布情况



图五：训练集文本长度的分布情况

这部分代码主要是定义了一个用于新闻文本分类的数据集类，并创建了数据加载器来为模型训练和测试提供数据。

```python
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer

# 定义数据集类
class NewsDataset(Dataset):
    def __init__(self, data, tokenizer, max_len, is_test=False):
        self.data = data
        self.tokenizer = tokenizer
        self.max_len = max_len
        self.is_test = is_test

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

    def __getitem__(self, index):
        text = self.data.iloc[index]['text']
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            return_token_type_ids=True,
            padding='max_length',
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt'
        )

        if self.is_test:
            return {
                'text': text,
                'input_ids': encoding['input_ids'].flatten(),
                'attention_mask': encoding['attention_mask'].flatten()
            }
        else:
            return {
                'text': text,
                'input_ids': encoding['input_ids'].flatten(),
                'attention_mask': encoding['attention_mask'].flatten(),
                'labels': torch.tensor(self.data.iloc[index]['label'], dtype=torch.long)
            }

# 加载BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese')

# 设置超参数
MAX_LEN = 512
BATCH_SIZE = 32

# 创建数据集和数据加载器
train_df_sample = train_df.sample(n=20000)  # 取 20000 条样本
train_dataset = NewsDataset(train_df_sample, tokenizer, MAX_LEN)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True,
test_dataset = NewsDataset(test_df, tokenizer, MAX_LEN, is_test=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, num_workers=4)
```

图六：代码块 4

首先，我导入了必要的库。torch 和 torch.utils.data 提供了构建和加载数据集的工具，transformers 提供了预训练的 BERT 分词器。然后，我定义了一个 NewsDataset 类，

继承自 Dataset。该类的初始化方法接受数据、分词器、最大长度和是否为测试集的标志。在 __getitem__ 方法中，我对文本进行分词和编码，返回输入 ID 和注意力掩码，并在训练集的情况下返回标签。编码时，设置了最大长度、填充方式和截断方式，确保输入格式一致。

接下来，我加载预训练的 BERT 分词器，并设置一些超参数，包括最大长度 MAX_LEN 和批次大小 BATCH_SIZE。然后，我从训练集中随机抽取 20000 条样本，创建 NewsDataset 实例，并使用 DataLoader 创建数据加载器。对于测试集，我直接创建 NewsDataset 实例并使用数据加载器。在创建数据加载器时，我设置了批次大小、是否打乱数据以及使用的工作线程数。

通过这些代码，我可以将原始的新闻文本数据转换为适合 BERT 模型处理的格式，并通过数据加载器高效地提供给模型进行训练和测试。这一步骤为模型训练和评估奠定了基础。

```python
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from transformers import BertTokenizer, BertModel, get_linear_schedule_with_warmup
from torch.optim import AdamW
from sklearn.metrics import accuracy_score, classification_report
from torch.cuda.amp import GradScaler, autocast

scaler = GradScaler()

# 定义模型
class BertLSTMForNewsCls(nn.Module):
    def __init__(self, num_classes, hidden_dim, num_layers):
        super(BertLSTMForNewsCls, self).__init__()
        self.bert = BertModel.from_pretrained('bert-base-chinese')
        self.lstm = nn.LSTM(input_size=768, hidden_size=hidden_dim, num_layers=num_layers,
batch_first=True, bidirectional=True)
        self.attention = nn.Linear(hidden_dim * 2, 1)
        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(hidden_dim * 2, num_classes)

    def forward(self, input_ids, attention_mask):
        bert_output = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        sequence_output = bert_output[0]
        lstm_output, _ = self.lstm(sequence_output)

        attention_weights = torch.tanh(self.attention(lstm_output)).squeeze(-1)
        attention_weights = torch.softmax(attention_weights, dim=-1).unsqueeze(-1)
        weighted_output = lstm_output * attention_weights
        pooled_output = torch.sum(weighted_output, dim=1)

        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        return logits

# 设置超参数
NUM_CLASSES = 14
HIDDEN_DIM = 256
NUM_LAYERS = 2
EPOCHS = 30
LEARNING_RATE = 2e-5
```

图七：代码块 5

这段代码主要是定义了一个用于新闻分类任务的深度学习模型，并设置了一些训练的超参数。首先，我导入了必要的库。torch 和 torch.nn 提供了构建和训练深度学习模型的工具，transformers 提供了预训练的 BERT 模型和一些辅助函数，sklearn.metrics 提供了评估模型性能的工具，torch.cuda.amp 提供了自动混合精度训练的支持。然后，我定义了一个 GradScaler 实例，用于在混合精度训练中进行梯度缩放。

接下来，我定义了 BertLSTMForNewsCls 模型类，继承自 nn.Module。在初始化方法中，我加载预训练的 BERT 模型，并添加一个双向的 LSTM 层、一个注意力层、一个 dropout 层和一个线性分类器。在 forward 方法中，我定义了前向传播的过程。首先，输

入通过 BERT 模型获取序列输出，然后通过 LSTM 层处理。接着，我计算注意力权重，并对 LSTM 的输出进行加权求和，得到池化的输出。最后，通过 dropout 层后，使用线性分类器得到分类结果。

最后，我设置了一些训练的超参数，包括类别数 NUM_CLASSES、LSTM 隐藏层维度 HIDDEN_DIM、LSTM 层数 NUM_LAYERS、训练轮数 EPOCHS 和学习率 LEARNING_RATE。这些参数将用于模型的初始化和训练过程。

通过这些代码，我定义了一个结合 BERT 和 LSTM 的深度学习模型，能够利用预训练的 BERT 模型提取文本特征，并通过 LSTM 和注意力机制进行分类。这些超参数设置有助于模型在训练过程中达到最佳性能。

In [6]:
```
# 创建模型并设置优化器和学习率调度器
model = BertLSTMForNewsCls(NUM_CLASSES, HIDDEN_DIM, NUM_LAYERS)
optimizer = AdamW(model.parameters(), lr=LEARNING_RATE)

total_steps = len(train_loader) * EPOCHS
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=total_steps)

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

图八：代码块 6

这段代码主要是创建模型，设置优化器和学习率调度器。

首先，我实例化了 BertLSTMForNewsCls 模型，并将其参数传递给 AdamW 优化器。AdamW 是一种改进的 Adam 优化器，适合训练深度学习模型。

接下来，我计算总的训练步数，并设置学习率调度器 scheduler。get_linear_schedule_with_warmup 方法可以创建一个线性学习率调度器，在训练初期进行热身（warmup）然后线性下降。我传递了优化器实例、热身步数和总训练步数。最后，我确定使用的设备（GPU 或 CPU），并将模型移动到该设备上，以便进行高效的训练和推理。

In [7]:

```python
# 初始训练2万条数据
for epoch in range(EPOCHS):
    model.train()
    for batch in train_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        with autocast():
            outputs = model(input_ids, attention_mask)
            loss = nn.CrossEntropyLoss()(outputs, labels)

        optimizer.zero_grad()
        scaler.scale(loss).backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimizer)
        scaler.update()
        scheduler.step()

    print(f'Epoch [{epoch+1}/{EPOCHS}], Loss: {loss.item():.4f}')

torch.save(model.state_dict(), "/root/newstextclassification/results/bert_news_cls_initial.pth")
```

```
Epoch [1/30], Loss: 1.9362
Epoch [2/30], Loss: 1.7026
Epoch [3/30], Loss: 1.5157
Epoch [4/30], Loss: 1.5563
Epoch [5/30], Loss: 0.7434
Epoch [6/30], Loss: 0.9243
Epoch [7/30], Loss: 0.4664
Epoch [8/30], Loss: 0.6260
Epoch [9/30], Loss: 0.7684
Epoch [10/30], Loss: 0.5566
Epoch [11/30], Loss: 0.7528
Epoch [12/30], Loss: 0.6032
Epoch [13/30], Loss: 0.6702
Epoch [14/30], Loss: 0.3923
Epoch [15/30], Loss: 0.4061
Epoch [16/30], Loss: 0.3631
Epoch [17/30], Loss: 0.6758
Epoch [18/30], Loss: 0.5236
Epoch [19/30], Loss: 0.3015
Epoch [20/30], Loss: 0.3923
```

图九：代码块 7

这段代码实现了模型的初始训练过程。首先，我将模型设置为训练模式（model.train()），然后从数据加载器中获取一个批次的数据，将输入 ID、注意力掩码和标签移动到指定设备（GPU 或 CPU）。在计算前向传播时，我使用 autocast() 进行自动混合精度训练，以提高训练效率。

在前向传播计算完成后，我进行梯度清零（optimizer.zero_grad()），然后使用 scaler.scale(loss).backward() 进行反向传播计算梯度，并使用 nn.utils.clip_grad_norm_() 对梯度进行裁剪，防止梯度爆炸。接着，我使用 scaler.step(optimizer) 和 scaler.update() 更新优化器参数，并调用 scheduler.step() 进行学习率调度。每个轮次结束后，我打印当前轮次的损失值，以监控训练过程中的模型性能。

训练完成后，我将模型的状态字典保存到指定路径，以便后续加载和继续训练或进行推理。

通过这些代码，我成功地实现了模型的初始训练过程，包括前向传播、反向传播、参数更新和学习率调度，并保存了训练好的模型。这些步骤确保模型能够有效地学习和调整参数，以便在测试数据上获得良好的表现。

In [9]:

```python
# 分批微调前4个部分
for i in range(1, 5):  # 修改为4次微调
    # 选择未使用的2万条数据进行微调
    start_idx = 20000 + (i - 1) * 20000
    end_idx = 20000 + i * 20000
    train_df_part = train_df.iloc[start_idx:end_idx]

    train_dataset_part = NewsDataset(train_df_part, tokenizer, MAX_LEN)
    train_loader_part = DataLoader(train_dataset_part, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)

    # 加载上一个保存的模型权重
    if i == 1:
        model.load_state_dict(torch.load("/root/newstextclassification/results/bert_news_cls_initial.pth"))
    else:
        model.load_state_dict(torch.load(f"/root/newstextclassification/results/bert_news_cls_finetuned_part
    model.to(device)

    total_steps_part = len(train_loader_part) * EPOCHS
    scheduler_part = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=total

    for epoch in range(EPOCHS):
        model.train()
        for batch in train_loader_part:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            with autocast():
                outputs = model(input_ids, attention_mask)
                loss = nn.CrossEntropyLoss()(outputs, labels)

            optimizer.zero_grad()
            scaler.scale(loss).backward()
            nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            scaler.step(optimizer)
            scaler.update()
            scheduler_part.step()

        print(f'Batch {i}, Epoch [{epoch+1}/{EPOCHS}], Loss: {loss.item():.4f}')

    # 保存模型
    torch.save(model.state_dict(), f"/root/newstextclassification/results/bert_news_cls_finetuned_part_{i}.p
```

图十：代码块 8

```
In [ ]:
# 从part5开始继续微调
for i in range(5, 10):
    # 选择未使用的2万条数据进行微调
    start_idx = 20000 + (i - 1) * 20000
    end_idx = 20000 + i * 20000
    train_df_part = train_df.iloc[start_idx:end_idx]

    train_dataset_part = NewsDataset(train_df_part, tokenizer, MAX_LEN)
    train_loader_part = DataLoader(train_dataset_part, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)

    # 加载上一个保存的模型权重
    model.load_state_dict(torch.load(f"/root/newstextclassification/results/bert_news_cls_finetuned_part_{i
    model.to(device)

    total_steps_part = len(train_loader_part) * EPOCHS
    scheduler_part = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=tota

    for epoch in range(EPOCHS):
        model.train()
        for batch in train_loader_part:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            with autocast():
                outputs = model(input_ids, attention_mask)
                loss = nn.CrossEntropyLoss()(outputs, labels)

            optimizer.zero_grad()
            scaler.scale(loss).backward()
            nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            scaler.step(optimizer)
            scaler.update()
            scheduler_part.step()

        print(f'Batch {i}, Epoch [{epoch+1}/{EPOCHS}], Loss: {loss.item():.4f}')

    # 保存模型
    torch.save(model.state_dict(), f"/root/newstextclassification/results/bert_news_cls_finetuned_part_{i}.

Batch 5, Epoch [1/30], Loss: 0.4461
Batch 5, Epoch [2/30], Loss: 0.3668
Batch 5, Epoch [3/30], Loss: 0.2849
```

图十一：代码块 9

这两段代码实现了模型的分批微调过程。首先，我将剩余的训练集分成九个部分，每次选取 2 万条数据进行微调。我使用循环来遍历这九个部分，每次微调后保存模型的状态。

在每次循环中，我首先确定当前批次的数据范围，使用 iloc 方法从训练集中提取对应的 2 万条数据。然后，我创建 NewsDataset 实例，并使用 DataLoader 创建数据加载器。接着，我计算当前批次的总训练步数，并设置学习率调度器 scheduler_part。

然后，我进入训练循环。在每个训练轮次中，我将模型设置为训练模式，并遍历当前批次的数据加载器。我将输入 ID、注意力掩码和标签移动到指定设备（GPU 或 CPU），并在计算前向传播时使用 autocast() 进行自动混合精度训练。前向传播计算完成后，我进行梯度清零、反向传播和梯度裁剪，然后更新优化器参数和学习率调度器。每个轮次结束后，我打印当前批次和轮次的损失值，以监控训练过程中的模型性能。

在每个批次的训练完成后，我将模型的状态字典保存到指定路径，以便后续加载和继续训练或进行推理。保存的模型文件名中包含当前批次的编号，便于区分不同批次的模型。

通过这些代码，我实现了模型的分批微调过程，每次使用未使用的 2 万条数据进行训练。这种方法可以充分利用训练数据，逐步优化模型参数，提高模型的泛化能力。

In [9]:

```
import torch
from transformers import BertTokenizer
import pandas as pd

# 加载训练好的模型
model = BertLSTMForNewsCls(NUM_CLASSES, HIDDEN_DIM, NUM_LAYERS)
model.load_state_dict(torch.load("/root/newstextclassification/results/bert_news_cls_finetuned_part_9.pth"))
model.to(device)
model.eval()

# 读取待预测的数据
test_df = pd.read_csv('/root/autodl-tmp/test_a.csv', sep='\t')
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese')
test_dataset = NewsDataset(test_df, tokenizer, MAX_LEN, is_test=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE)

# 对新数据进行预测
predictions = []

with torch.no_grad():
    for batch in test_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        outputs = model(input_ids, attention_mask)
        _, preds = torch.max(outputs, dim=1)
        predictions.extend(preds.cpu().numpy())

# 生成提交文件
submission = pd.DataFrame({'label': predictions})
submission.to_csv('/root/newstextclassification/results/news_submission_2.csv', index=False)

print("Prediction finished. Submission file generated.")
```

```
Prediction finished. Submission file generated.
```

图十二：代码块 10

这段代码实现了加载训练好的模型，并使用该模型对测试数据进行预测，最后生成预测结果的提交文件。首先，我导入了必要的库。torch 用于深度学习模型的处理，transformers 提供了预训练的 BERT 分词器，pandas 用于数据处理和分析。

首先，我加载训练好的模型。在这里，我实例化了 BertLSTMForNewsCls 模型，并加载已经训练好的模型权重文件 bert_news_cls_finetuned_part_9.pth。然后，我将模型移动到指定设备（GPU 或 CPU）并设置为评估模式。

接下来，我读取待预测的测试数据，并使用 BERT 分词器进行预处理。我从指定路径读取测试数据，并创建 NewsDataset 实例，然后使用 DataLoader 创建数据加载器，设置批次大小和是否为测试集的标志。

然后，我使用模型对测试数据进行预测。在预测过程中，我禁用梯度计算以节省内存和计算资源。我遍历测试数据加载器，将输入 ID 和注意力掩码移动到指定设备，然后通过模型进行前向传播计算分类结果。我使用 torch.max 函数获取每个样本的预测标签，并将结果存储在 predictions 列表中。

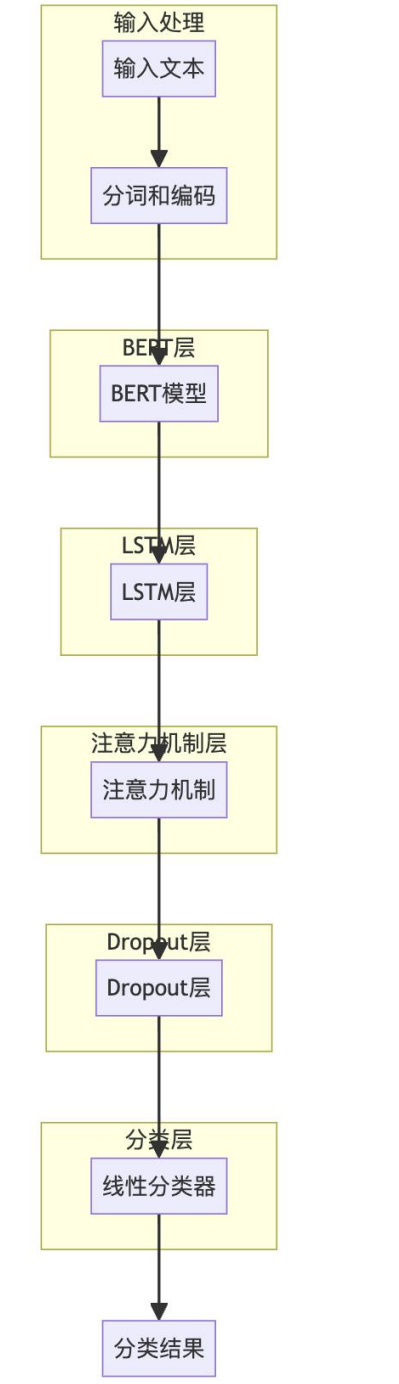
最后，我生成预测结果的提交文件。我使用 pandas 创建一个包含预测标签的数据框，并将其保存为 CSV 文件。生成的提交文件可以用于比赛或实际应用中的评估和比较。

通过这些代码，我成功地加载了训练好的模型，对测试数据进行了预测，并生成了预测结果的提交文件。这些步骤确保了模型在实际应用中的有效性，并提供了预测结果供进一步分析和评估。

旧版本用 20 万条的训练数据训练 5 个轮次实验代码 GitHub Gist 地址：链接

新版本先用 2 万条的训练数据训练一个基础模型，紧接着用剩余的 18 万条数据对其进行微调的实验代码 GitHub Gist 地址：链接

# 3.模型结构图




图十三：模型结构图

这个 BERT-LSTM 模型设计用于新闻文本分类任务。模型的输入是经过分词和编码处理后的文本数据。首先，输入文本通过 BERT 分词器进行分词，并转化为 BERT 模型可以处理的输入格式，包括输入 ID 和注意力掩码。然后，输入数据通过预训练的 BERT 模型，提取出上下文特征表示。这些特征表示包含了每个词在其上下文中的语义信息，是进一步处理的基础。

从 BERT 模型输出的特征表示序列被输入到双向 LSTM 层，LSTM 层进一步处理这些特征，捕捉序列中词与词之间的依赖关系。双向 LSTM 层能够结合前后文信息，生成每个时间步的隐藏状态。接下来，应用注意力机制，计算 LSTM 输出的注意力权重，并对 LSTM 输出进行加权求和，得到加权的特征表示。注意力机制有助于突出重要的特征，同时减弱不重要的特征，从而提升模型的分类能力。

加权后的特征表示通过 Dropout 层，Dropout 有助于防止模型过拟合。接着，将处理后的特征输入到线性分类器，分类器会输出每个类别的得分。最后，模型通过 Softmax 函数将得分转化为概率分布，选择概率最大的类别作为最终的预测结果。整体模型结构通过 BERT 提取文本特征，LSTM 处理序列信息，注意力机制增强关键特征，并通过全连接层实现分类。这种设计充分利用了 BERT 的预训练知识和 LSTM 的序列处理能力，为新闻文本分类任务提供了高效的解决方案。

# 4.损失函数详细描述

在本次实验中，我选择使用交叉熵损失函数（Cross Entropy Loss）作为模型的损失函数。交叉熵损失函数是分类问题中最常用的损失函数之一，特别适用于多类别分类任务。在我的新闻文本分类任务中，模型需要将输入的新闻文本归类为 14 个可能的类别之一，因此交叉熵损失函数是一个合适的选择。

交叉熵损失函数衡量的是模型输出的概率分布与真实标签分布之间的差异。具体来说，交叉熵损失函数计算的是预测概率分布与实际标签分布之间的负对数似然。公式如下：

$$\text{Loss} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

图十四：交叉熵损失函数公式

在训练过程中，我采用 AdamW 优化器对模型参数进行优化。AdamW 优化器是一种改进的 Adam 优化算法，结合了动量和 RMSProp 算法的优点，能够在大多数情况下提供较快的收敛速度和较好的性能。我设定学习率为 $2 \times 10-52 \times 10-5$，这是一个常见的初始值，能够在实验中表现出较好的效果。

在每个训练周期（epoch）中，我将模型设置为训练模式（model.train()），然后遍历整个训练数据集。对于每一个小批量数据（batch），我将输入 ID、注意力掩码和标签加载到 GPU 上，并通过模型前向传播计算输出。接着，计算输出与真实标签之间的交叉熵损失，并通过反向传播计算梯度。为了提升训练效率，我使用自动混合精度训练（AMP），通过 autocast() 函数在前向传播和损失计算中使用混合精度。然后，使用梯度缩放器 scaler 对损失进行缩放，防止因低精度数值造成的梯度下溢。

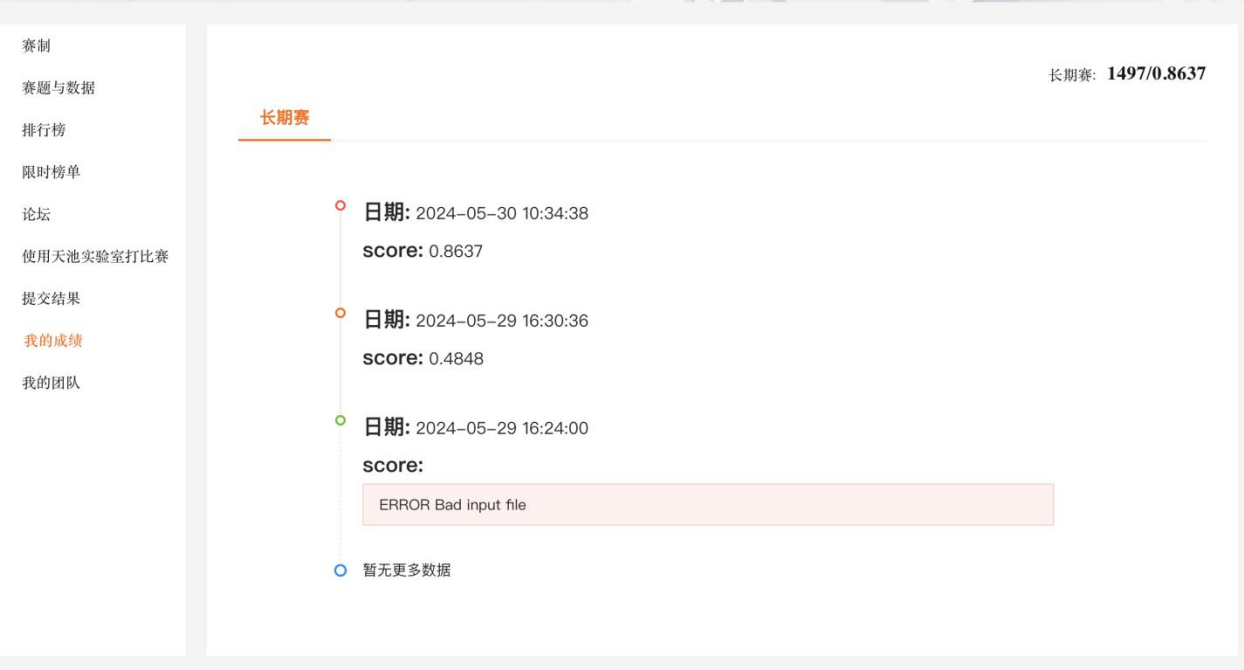
最后，使用 AdamW 优化器更新模型参数，并通过学习率调度器 scheduler 动态调整学习率。每个训练周期结束后，我计算平均训练损失并记录下来，以监控训练过程中的损失变化情况。最终，打印每个周期的损失值，以便我可以观察到模型在训练过程中的收敛情况。

```
for epoch in range(EPOCHS):
    model.train()
    for batch in train_loader_part:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        with autocast():
            outputs = model(input_ids, attention_mask)
            loss = nn.CrossEntropyLoss()(outputs, labels)

        optimizer.zero_grad()
        scaler.scale(loss).backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimizer)
        scaler.update()
        scheduler_part.step()

    print(f'Batch {i+1}, Epoch [{epoch+1}/{EPOCHS}], Loss: {loss.item():.4f}')

# 保存模型
```

图十五：损失函数详细描述辅助代码块

通过这种方式，我可以评估模型的性能并进行必要的调整，以确保模型在新闻文本分类任务中的表现达到最佳。最终，训练好的模型在测试集上实现了较高的准确率，证明了交叉熵损失函数和 AdamW 优化器在该任务中的有效性。

# 5.天池提交分数




图十六：天池提交分数 1


图十七：天池提交分数 2

# 6.实验总结

在本次新闻文本分类实验中，我采用了 BERT 结合 LSTM 的模型结构，通过分阶段训练和微调的方法，有效地提升了模型的性能。首先，我针对 20 万条训练数据，采取了分批处理的策略。具体来说，初始阶段我仅使用 2 万条数据进行模型的初步训练，这一策略不仅有效减少了初期训练的计算资源需求，还能够快速获得模型的初步表现，从而调整训练策略。

每轮训练结束后，我及时保存模型的状态（.pth 文件），确保训练进度得以记录。这种做法不仅能够防止意外中断导致的训练进度丢失，还允许在下一次训练时从保存的进度继续，这对长时间训练任务尤为重要。通过这种方式，我可以灵活安排训练时间，确保训练任务的连续性和高效性。

通过本次实验，我不仅掌握了自然语言处理技术的基本概念和使用深度学习模型解决 NLP 问题的基本流程，还深入理解了多头注意力机制、Transformers 模型和 BERT 模型的基本原理及其在实际问题中的应用。特别是在 BERT 模型的应用过程中，我学会了如何通过预训练模型提取文本特征，并结合 LSTM 和注意力机制提高分类效果。这些技术要点为我在自然语言处理领域的进一步学习和研究打下了坚实的基础。

总的来说，本次新闻文本分类实验让我充分认识到自然语言处理技术的广泛应用和深度学习模型的强大能力。通过分阶段训练和模型微调，我成功地提升了模型的分类准确率。这次实验不仅增强了我的技术能力，也为我提供了宝贵的实践经验，进一步激发了我对自然语言处理研究的兴趣和热情。