你什么意思 做题笔记：

1.数据预处理（data_process)：

```
for fi in file_list:
        with open(fi, encoding='utf8') as file_input:
            all_texts += [remove_tags(" ".join(file_input.readlines()))]

    return all_labels, all_texts
```

`utf8`是最通用、最兼容的字符编码方式,主要用于确保文件正确读取和写入，避免乱码和编码错误。

`file_input.readlines()` 读取文件所有行，返回列表。

`" ".join(file_input.readlines())` 连接所有行，转换为一个字符串，去掉 `\n`。

`remove_tags(...)` 去除 HTML 标签，确保纯文本。

`ll_texts += [...]`追加清理后的文本到 `all_texts`。

文本编码：

```
# 进行文本编码
def encode_texts(texts,tokenizer,max_length=512):

    encoded = tokenizer(texts, # 输入文本列表
    add_special_tokens=True, # 添加特殊标记 [CLS] 和 [SEP]
    max_length=max_length, # 限制最大长度
    padding='max_length',  # 短文本用 `0` 填充到最大长度
    truncation=True,  # 长文本截断
    return_tensors='np', # 返回 NumPy 数组
    return_attention_mask=True) # 生成注意力掩码

    # 返回文本的token ID和attention mask（用于标识padding部分）
    return encoded['input_ids'],encoded['attention_mask']
```

作用：将文本转换为BERT可以处理的张量形式，转换为BERT需要的token ID，并生成对应的attention mask。

`max_length=512`：BERT 的最大输入长度，超出这个长度的文本会被**截断**。

`padding='max_length'`:若文本长度 < `max_length`，用 `0` 填充。如果输入的文本长度小于模型规定的最大长度（通常是 512），则会进行 **填充**，即在句子后面填充特殊符号 `[PAD]`，直到达到指定的最大长度。

`truncation=True`:如果输入的文本长度超过了最大长度，则会 **截断** 多余的部分，通常会截掉从序列末尾开始的部分。

`return_attention_mask=True`:生成的 **attention mask** 是一个与输入序列相同长度的二进制向量。值为 1 表示该位置的 token 是有效的（即它是实际的文本数据），值为 0 表示该位置是填充部分。模型会根据 attention mask 来避免计算填充部分。

2.Transformer.train:

训练集 (`shuffle=True`):

在训练过程中，我们希望每个 epoch 都以不同的顺序看到数据，这样可以防止模型过拟合某些数据顺序，提高泛化能力。

Transformer分类模型

```
# Transformer 分类模型
class TransformerClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, num_heads=4, num_layers=2,
num_classes=2):
        # 初始化 Transformer 分类器
        # - vocab_size: 词汇表大小
        # - embed_dim: 词向量的维度（embedding 维度）
        # - num_heads: 多头自注意力的头数
        # - num_layers: Transformer 编码器的层数
        # - num_classes: 分类类别数（默认为二分类任务）
        super().__init__()
        # 词嵌入层：将输入的单词索引转换为 `embed_dim` 维度的向量
        self.embedding = nn.Embedding(vocab_size, embed_dim)

        # 定义 Transformer 编码器层
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embed_dim,     # 词向量维度（必须和 embedding 维度一致）
            nhead=num_heads,       # 多头自注意力头数
            dim_feedforward=256    # 前馈神经网络的隐藏层维度（默认 256）
            )

        # 堆叠多个Transformer编码器层
        self.transformer = nn.TransformerEncoder(
            encoder_layer,            # 使用前面预定义的编码器层
            num_layers=num_layers     # 指定Transformer层的数量
            )

        # 分类层（全连接层）
        self.fc = nn.Linear(embed_dim, num_classes)

    def forward(self, x,attention_mask):
        x = self.embedding(x)  # 将输入形状(batch, seq_len)转化为(batch, seq_len,
embed_dim)
        x = x.permute(1, 0, 2)  # Transformer 需要 (seq_len, batch, embed_dim)
        padding_mask = attention_mask == 0  # 转换为布尔型，Padding 的地方是 True
        x = self.transformer(x, src_key_padding_mask=padding_mask)
        x = x.mean(dim=0)  # 取均值，将(seq_len, batch, embed_dim)变为(batch,
embed_dim) ，符合全连接层的输入需要
        return self.fc(x)
```

`src_key_padding_mask` 是 `nn.TransformerEncoder` 的参数，它：

- **True 代表填充位置，需要被屏蔽**
- **False 代表真实 token，正常计算注意力**

3.超参数、损失函数和优化器的设定：

```
# 超参数
vocab_size = 30522  # BERT 词表大小（可以改小）
model = TransformerClassifier(vocab_size).to(device)

# 损失函数和优化器
criterion = nn.CrossEntropyLoss()  # 交叉熵损失函数
optimizer = optim.AdamW(model.parameters(), lr=2e-4)
```

```
nn.CrossEntropyLoss()
```

数学公式：

$$H(y,y) = -\sum yilog(yi)$$

Pytorch会自动处理softmax将未归一化模型的输出分数转化为概率分布，再计算交叉熵损失，所以无需额外添加softmax。

4.训练模型：

```
# 训练模型
def train_model(model, train_loader, criterion, optimizer, epochs=5):
    model.train() # 设定模型为训练模式
    for epoch in range(epochs):
        total_loss = 0   # 用于累加该 epoch 的总损失
        correct = 0      # 用于统计预测正确的样本数
        total = 0        # 用于统计总样本数，以计算准确率

        for batch in train_loader:
            optimizer.zero_grad()  # 清空梯度
            input_ids, attention_mask,labels = [x.to(device) for x in batch]
            outputs = model(input_ids,attention_mask=attention_mask)
            loss = criterion(outputs, labels)
            loss.backward() # 反向传播
            optimizer.step() # 更新梯度

            total_loss += loss.item() # .item将tensor转换为普通python类型
            predictions = torch.argmax(outputs, dim=1) # 找到张量中最大值所在的索引，
获取预测类别
            correct += (predictions == labels).sum().item()  # 统计预测正确的样本数
            total += labels.size(0)  # 总样本数

        print(f"Epoch {epoch+1}: Loss = {total_loss/len(train_loader):.4f},
Accuracy = {correct/total:.4f}")

train_model(model, train_loader, criterion, optimizer)
```

`train()` 设置训练模式

循环 `epoch` 和 `batch` 训练

`zero_grad()` **清空梯度**

前向传播 `outputs = model(input)`

计算损失 `loss = criterion(outputs, labels)`

反向传播 `loss.backward()`

优化器更新 `optimizer.step()`

计算准确率

打印 `Loss` **和** `Accuracy`

5.测试模型：

```
# 评估模型
def evaluate_model(model, test_loader):
    model.eval() # 设定模型为评估模式
    correct = 0  # 用于统计预测正确的样本数
    total = 0    # 用于统计总样本数，以计算准确率
    with torch.no_grad():
        for batch in test_loader:
            input_ids, attention_mask,labels = [x.to(device) for x in batch]
            outputs = model(input_ids,attention_mask=attention_mask)
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)

    print(f"Test Accuracy: {correct/total:.4f}")
```

测试部分大体思路与训练部分类似

model.train()和model.eval()的区别：

`train()` 用于训练，会开启 `Dropout`，`BatchNorm` 更新统计信息。

`eval()` 用于推理，会关闭 `Dropout`，`BatchNorm` 使用全局统计信息。

另外，测试部分中使用 `torch.no_grad()`：彻底关闭梯度计算，减少计算量和显存占用，因为测试阶段不需要更新梯度。

6.BERT.train.py:

```
# 定义模型
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
num_labels=2)
model.to(device)
for param in model.bert.parameters():
    param.requires_grad = False  # 冻结 BERT 主干网络(将所有参数的梯度设置为不更新)

for param in model.bert.encoder.layer[-4:].parameters():
    param.requires_grad = True  # 只训练最后 4 层
```

这里冻结了BERT主干网络的大部分参数，只对最后4层进行微调，从而加速训练，刚开始我也尝试了不使用冻结，训练速度实在太慢了。

BERT 的前几层提取的是通用语言特征（比如词语的语法关系），而后几层捕捉到的更多是高层语义信息，与具体任务更相关。

只微调最后4层可以保留预训练的特征提取能力，加快训练速度。

```
# 混合精度训练
scaler = GradScaler()

def train_model(model, train_loader, criterion, optimizer, epochs=3):
    model.train() # 训练模式
    for epoch in range(epochs):
        # 初始化loss、正确预测数和总样本数
        total_loss = 0
        correct = 0
        total = 0
        for batch in train_loader:
            optimizer.zero_grad()  # 梯度清空
            input_ids, attention_mask, labels = [x.to(device) for x in batch]

            # 启用 autocast() 进行混合精度训练
            with autocast():
                outputs = model(input_ids, attention_mask=attention_mask)
                loss = criterion(outputs.logits, labels)

            scaler.scale(loss).backward() # GradScaler进行损失缩放
            scaler.step(optimizer)  # scaler.step(optimizer) 先反缩放梯度，再更新参数。
            scaler.update()  # 动态调整损失缩放系数，确保 FP16 训练稳定。

            total_loss += loss.item()
            predictions = torch.argmax(outputs.logits, dim=1)  # 取 num_labels 维度上的最大值索引，得到最终预测类别
            correct += (predictions == labels).sum().item()
            total += labels.size(0)

        print(f"Epoch {epoch+1}: Loss = {total_loss/len(train_loader):.4f}, Accuracy = {correct/total:.4f}")

train_model(model, train_loader, criterion, optimizer)
```

混合精度训练：

混合精度训练是一种同时使用 16-bit（FP16）和 32-bit（FP32）浮点数进行深度学习训练的方法，目的是加快计算速度，减少显存占用，同时保持模型精度。

在 PyTorch 的 **AMP（Automatic Mixed Precision）** 训练中：

1. **大部分计算使用 FP16**（单精度浮点数）（减少显存占用，加快计算）。
2. **关键运算使用 FP32**（半精度浮点数）（如 `Softmax`、`LayerNorm`，防止数值不稳定）。

FP32（单精度）：精度高，占用 4 字节，计算较慢，由于指数位有 8 bit，可以表示 `±3.4 × 10^38`，所以更能容忍梯度的剧烈变化，不容易溢出。

FP16（半精度）：精度低，占用 2 字节，计算更快，但由于其指数位只有 5 bit，能表示的最大数值大约是 `6.5 × 10^4`。当梯度值超过这个范围时，就会溢出。

PyTorch 提供了两个核心工具：

1. `torch.cuda.amp.autocast()`：自动选择 FP16/FP32 计算。
2. `torch.cuda.amp.GradScaler()`：动态调整梯度，防止梯度溢出。

(1) `autocast()` 自动选择计算精度

(2) scaler = GradScaler(): `GradScaler()`：防止梯度溢出

(3) `scaler.scale(loss).backward()`：缩放损失

(4) `scaler.step(optimizer)`：更新梯度

(5) `scaler.update()`：动态调整缩放因子

Attention_mask：

对于 BERT 这类 NLP 预训练模型，输入通常是批量化的句子，但每个句子的长度不同，所以必须填充（padding）到相同的长度。

所以要使用 `attention_mask` 标识 哪些是有效的 token，哪些是 padding。

`attention_mask=1`：正常计算注意力。

`attention_mask=0`：屏蔽 padding，即不计算注意力。

7.运行结果：

Transformer:

```
Epoch 1: Loss = 0.4847, Accuracy = 0.7528
Epoch 2: Loss = 0.3411, Accuracy = 0.8534
Epoch 3: Loss = 0.2837, Accuracy = 0.8823
Epoch 4: Loss = 0.2445, Accuracy = 0.9008
Epoch 5: Loss = 0.2107, Accuracy = 0.9176
Test Accuracy: 0.8628
```

Bert:

```
Epoch 1: Loss = 0.2471, Accuracy = 0.9005
Epoch 2: Loss = 0.1798, Accuracy = 0.9311
Epoch 3: Loss = 0.1397, Accuracy = 0.9476
Test Accuracy: 0.9316
```