实验报告：序列标注编程作业

1.实验简介

本实验任务是设计并实现一个基于Transformer模型的命名实体识别（NER）系统。通过训练语料(train.txt和train_TAG.txt)，模型将学习识别文本中的实体并为其分配正确的标签。使用发展集(dev.txt和dev_TAG.txt)对模型进行调优，最终在测试集(test.txt)上评估模型性能。实验结果将展示模型在识别人名、地点、组织等实体方面的准确性。实验报告将包含模型设计、训练细节、性能曲线和部分代码实现。

2.数据准备

1. **标签集统计**：首先从训练标签文件`train_TAG.txt`中统计出所有唯一的标签，并将它们存储在一个集合中。然后，将这个集合转换为列表，并将这个列表保存到文件`tag_set.txt`中。此外，还添加了一个'pad'标签，用于表示padding tokens。最后一共有10个标签：

```
# 序列标注的标签集合
['I_LOC', 'B_T', 'B_ORG', 'B_PER', 'I_T', 'I_PER', 'I_ORG', 'B_LOC', 'O', 'pad']
```

2. **标签到索引的映射**：接着，我创建了一个从标签到索引的映射字典`label2idx`，这在数据编码时用于将标签转换为模型可以理解的数字索引。

```
# 标签到索引的映射
label2idx = {label: idx for idx, label in enumerate(tag_set)}
```

3. **数据集类定义**：定义了一个名为NERDataset的PyTorchDataset类，用于处理和封装文本数据和对应的标签，使其能够被模型读取和训练。这个类将文本和标签转换为BERT模型所需的格式，使用tokenizer对文本进行编码，生成模型所需的input_ids和attention_mask。设置padding='max_length'和truncation=True以确保所有文本序列长度一致，使用max_length参数指定最大长度。return_tensors='pt'指定返回的格式为PyTorch张量。

   对于数据集中的每一个句子，NERDataset类在调用其__getitem__方法时会返回一个字典，其中包含以下键对应的值：

   1. 'input_ids': 一个一维的PyTorch张量，包含了句子经过BERT分词器处理后的token ID序列。如果原始句子的token数量少于max_len，剩余的位置将用padding token的ID填充。下面是一个例子：

      ```
      # 第一个句子的 input_ids 值
      tensor([ 101,  782, 3696, 5381,  122, 3299,  122, 3189, 6380, 2945,  517, 5294,
              5276, 3198, 2845,  518, 2845, 6887,  117, 5401, 1744, 1290, 2209, 6125,
              ......
               809, 3297, 7770, 5279, 2497, 2772, 2970, 6818, 3297, 7770, 5279, 2497,
              5310, 3338, 3315, 2399, 4638,  769, 3211,  511,  102,    0,    0,    0,
                 0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
              ......
                 0,    0,    0,    0,    0,    0,    0,    0])
      ```

      101 通常是BERT使用的特殊的开始标记[CLS] token的ID，0 通常用作[PAD] token的ID，其他大部分的数字，每个数字代表一个token的ID。这些token是从原始文本序列经过分词器转换而来的，可能包括单词、标点符号或特殊字符。

   2. 'attention_mask': 一个一维的PyTorch张量，用于指示句子中哪些token是实际的单词（而非padding token）。在这个张量中，实际单词对应的位置是1，而padding token的位置是0。这有助于模型在计算注意力权重时忽略padding token。

   3. 'labels': 一个一维的PyTorch张量，包含了与句子中每个token对应的标签索引。这些索引是根据label2idx映射生成的。如果句子的token数量少于max_len，剩余的位置将用'pad'标签的索引填充，这通常表示这些位置不参与模型的损失计算。

   这个字典格式的输出是BERT模型以及大多数现代Transformer模型所期望的输入格式，可以被直接用于模型训练和预测。通过这种方式，NERDataset类使得每个句子在输入模型之前都经过了适当的转换和封装。

4. **数据读取与处理**：读取训练数据和发展集数据的文本和标签文件，将它们分别存储在列表中。

5. **文本分割**：定义了一个split_text函数，用于将长文本分割成多个短文本，以适应模型的最大序列长度限制。

6. **数据集封装**：使用split_train_texts和split_train_tags创建训练数据集，使用split_dev_texts和split_dev_tags创建发展数据集，并使用BertTokenizer进行预处理。

```
tokenizer = BertTokenizer.from_pretrained('./bert_tokenizer')
train_dataset = NERDataset(split_train_texts, split_train_tags, tokenizer, max_len=128,
label2idx=label2idx)
dev_dataset = NERDataset(split_dev_texts, split_dev_tags, tokenizer, max_len=128,
label2idx=label2idx)
```

3.加载BERT预训练模型

BERT和GPT都是基于Transformer架构的模型，但它们在设计理念和应用场景上有所不同。对于序列标注任务，通常BERT更为合适，原因如下：

1. **双向上下文**：BERT模型设计时就考虑了双向上下文，即在训练时同时考虑了单词的左侧和右侧上下文。这对于序列标注任务非常重要，因为实体的标注通常依赖于对整个实体的上下文理解。
2. **任务适应性**：BERT在设计时就考虑到了多种下游任务的适应性，包括序列标注。它的预训练目标使得模型能够学习到丰富的语言表示，有助于提高序列标注的性能。
3. **迁移学习**：由于BERT的预训练特性，它可以很容易地迁移到特定的序列标注任务上。在实际应用中，通常只需要在顶层添加一个简单的线性层或全连接层，并对模型进行微调即可。

相比之下，GPT模型主要设计用于生成任务，如文本生成、条件文本生成等。它的特点是单向性，即在训练时只考虑单词的左侧上下文，这在文本生成任务中是有利的，因为它能够捕捉到文本的流畅性和连贯性。然而，这种单向性在序列标注任务中可能是一个缺点，因为它无法像BERT那样同时捕捉到单词的左右两侧上下文。

尽管如此，GPT及其变体（如GPT-2）仍然可以用于序列标注任务，尤其是当任务需要生成或预测序列的标签时。但在大多数情况下，BERT由于其双向上下文理解能力，仍然是序列标注任务的首选模型。

3.1BERT模型的结构

以下是 `BertForTokenClassification` 模型的简化结构描述，它基本涵盖了BERT模型的主要组成部分：

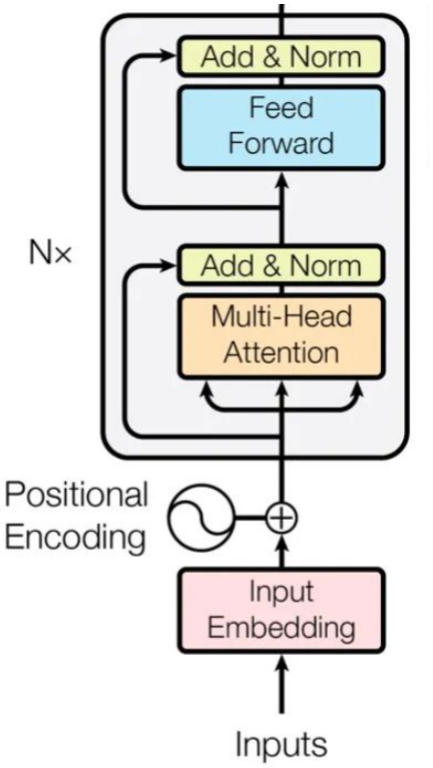
```
BertForTokenClassification
|
|--- BertModel          # BERT的基本模型结构
|    |
|    |--- Embeddings     # 包括词嵌入、位置嵌入、片段嵌入
|    |
|    |--- Encoder      # 多层Transformer编码器
|         |--- Layer_0, Layer_1, ..., Layer_n (12个相同的Transformer编码器层)
|              |--- Multi-Head Attention
|              |--- Add & Norm
|              |--- Feed Forward Network
|              |--- Add & Norm
|   |--- Dropout
|--- Classification Head  # 用于分类的头部，针对每个token预测分类标签
     |--- Linear Layer (可能有多层，本实验只有一层)
     |--- Activation Function (如ReLU)
     |--- Output Layer (针对每个token预测标签的分类，输出维度与标签种类数相同)
```

这个结构主要包括两个部分：

1. **BertModel**：这是BERT模型的核心结构，它基于Transformer架构，旨在处理各种NLP任务。在这个结构中，词嵌入、位置嵌入和片段嵌入共同作用，为模型提供了丰富的输入表示。这些嵌入层把单词转换成向量形式，使模型能够理解文本的语义和语法信息。
   - **Embeddings**：
     - **词嵌入(word_embeddings)**：将每个单词映射到一个高维空间，使得语义相近的单词在这个空间中也相近。该模型中对应的嵌入矩阵是`Embedding(21128, 768, padding_idx=0)`，此矩阵的大小为 21128 x 768，意味着它包含 21128 个不同词汇的嵌入，每个词的嵌入是一个 768 维的向量。

- **位置嵌入(position_embeddings)**：由于Transformer是不关注顺序的，位置嵌入为模型提供了词语在句子中的相对或绝对位置信息。在模型中表现为一个`Embedding(512, 768)`嵌入矩阵。
- **片段嵌入(token_type_embeddings)**：对于处理多个句子的任务（如问答任务），片段嵌入标识句子的界限和归属，帮助模型处理跨句子的语义关系。每种类型的句子都有一个独特的嵌入向量，这里的大小是`Embedding(2, 768)`，意味着有两种不同的句子类型可以被模型区分。
- **LayerNorm**: 层归一化是一种在深度神经网络中常用的技术，用于稳定神经网络的训练。它对每个样本的所有特征进行归一化，确保训练过程中特征分布的均一性。这里的 `LayerNorm` 是应用于嵌入向量的，有助于改进训练动态和模型性能。
- **dropout**: Dropout 是一种正则化技术，通过随机丢弃网络中的一部分激活（这里的概率为 0.1），来防止模型过拟合。这在训练复杂的神经网络时尤其有用，可以增加模型的泛化能力。

- **Encoder**：由12个Transformer编码层组成，每一层都有两个主要子结构：
  - **多头注意力机制**：这是模型的核心，通过对输入数据的不同子空间进行并行注意，模型可以从多个角度理解信息，这对于捕捉语言的复杂性非常关键。
  - **前馈神经网络**：每个多头注意力层后面都跟着一个前馈神经网络，这个网络对每个位置的输出进行独立处理，增加了模型的表达能力。 该模型中前馈神经网络为两个线性层：`Linear(in_features=768, out_features=3072, bias=True)`、`Linear(in_features=3072, out_features=768, bias=True)`
- **Dropout**：用于防止模型过拟合，通过随机地丢弃一部分神经元的输出来增加模型的泛化能力。



2. **Classification Head**：
   - **线性层**：这一部分通常包含多个线性层，用于从Encoder的输出中提取特征并进行最终的分类决策。
   - **激活函数**：介于线性层之间，用于增加非线性，帮助模型学习更复杂的模式。

BertModel 是传统的 BERT 模型部分，**Classification Head** 是针对特定任务（在这个情况下是 token 级别的分类任务）添加的额外结构。因此，可以将 `BertForTokenClassification` 理解为基于传统 BERT 模型（`BertModel`），通过添加一个特定的分类头部（`Classification Head`）来适应特定的 NLP 任务，这种设计增加了 BERT 模型在不同任务上的适用性和灵活性。

3.2结构参数及超参数

以下是模型的几个关键结构参数：

1. **Transformer类型**: 使用的是BERT模型，它基于Transformer编码器层堆叠而成。
2. **堆叠层数**: BERT模型的主体是一个由多个相同的层`BertLayer`组成的模块列表`ModuleList`。该BERT模型包含12层，每层都执行相同的操作。
3. **输入token数**: 模型的输入是通过`BertTokenizer`分词得到的token序列。最大序列长度由`max_len`参数决定，这个值被设置为128。这意味着输入序列中的token数量不能超过128个。
4. **词嵌入维度**：词的嵌入是一个 768 维的向量。
5. **隐藏层维度**: BERT模型中的隐藏层维度是768，这是在`word_embeddings`、`query`、`key`、`value`、`dense`等组件中使用的维度。
6. **输出维度**: `classifier`是一个线性层，其`in_features`是768，这与BERT模型的隐藏层维度相匹配。`out_features`是10，这表明模型的输出是10个类别的概率分布。
7. **激活函数**: 在`BertIntermediate`部分使用了GELU（Gaussian Error Linear Unit）激活函数。
8. **Dropout**: 在多个地方使用了dropout（p=0.1），用于正则化模型，减少过拟合。
9. **预训练权重**: 模型使用了预训练权重（`from_pretrained('./bert_model')`），这意味着模型在被用于特定任务之前已经在大量数据上进行了预训练。

以下是模型的超参数：

1. **批次大小**: `batchsize=128` 在文本分割函数中使用，意味着如果单个文本的token数超过128，文本将被分割成多个批次。
2. **训练批次大小**: `batch_size=32` 用于训练和评估，表示每个批次包含32个样本。
3. **学习率**: `lr=5e-5` 是优化算法中的步长，用于调整模型权重。
4. **训练轮数**: `epoch_num = 10` 表示整个训练集将被遍历10次。

3.3模型微调及发展集评估

- **模型微调**

在训练过程中，我通过tqdm实现了批次数据的可视化进度显示，并通过将数据传送到GPU来加速计算。我使用了自动混合精度来优化性能，通过梯度置零和梯度缩放来管理梯度，避免了在训练中可能出现的梯度消失问题。整个训练流程通过计算损失并进行反向传播和参数更新，确保了模型按照任务需求进行有效的学习和优化，从而提高了模型在特定自然语言处理任务上的表现和准确性。

代码实现：

```
    # 使用tqdm来包装data_loader，显示每个batch的进度
    for batch in tqdm(data_loader, desc="Training", leave=False):
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        optimizer.zero_grad()  # 将梯度置零放在循环开始处，以避免潜在的优化问题

        # 自动管理混合精度的上下文
        with autocast():
            outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
```

```
            loss = outputs.loss

        # 使用梯度缩放进行反向传播
        scaler.scale(loss).backward()
        scaler.step(optimizer)  # 使用scaler来更新模型参数
        scaler.update()  # 更新缩放器
```

- **发展集评估**

我一共执行了10个epochs，在每个周期中，模型通过train_loader提供的训练数据进行学习，使用optimizer调整模型参数来最小化损失函数，该损失由train_model函数计算并返回。随后，模型在发展集（dev_dataset）上进行评估，以验证其在未见数据上的性能。evaluate_model函数计算模型在开发集上的准确率。最后，每个周期结束时，代码打印出当前周期的训练损失和验证准确率。

代码实现：

```
def evaluate_model(model, dev_dataset, device):
    '''在发展集上计算正确率'''
    ......
    with torch.no_grad():  # 禁用梯度计算
        for batch_idx, batch in enumerate(dev_loader):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)  # 仍需要labels来进行准确率计算
            # 移除labels参数，因为我们不希望在评估时进行损失计算或后向传播
            outputs = model(input_ids, attention_mask=attention_mask)

            prob = F.softmax(outputs.logits, dim=-1)  # 得到每个位置上每个类别的概率
            # 获取最高概率的类别索引，即预测类别
            preds = torch.argmax(prob, dim=-1)  # torch.Size([32, 128])
            # 计算正确预测的数量
            ......
    accuracy = total_correct / total_samples
    return accuracy
```

3.4损失曲线和性能变化曲线

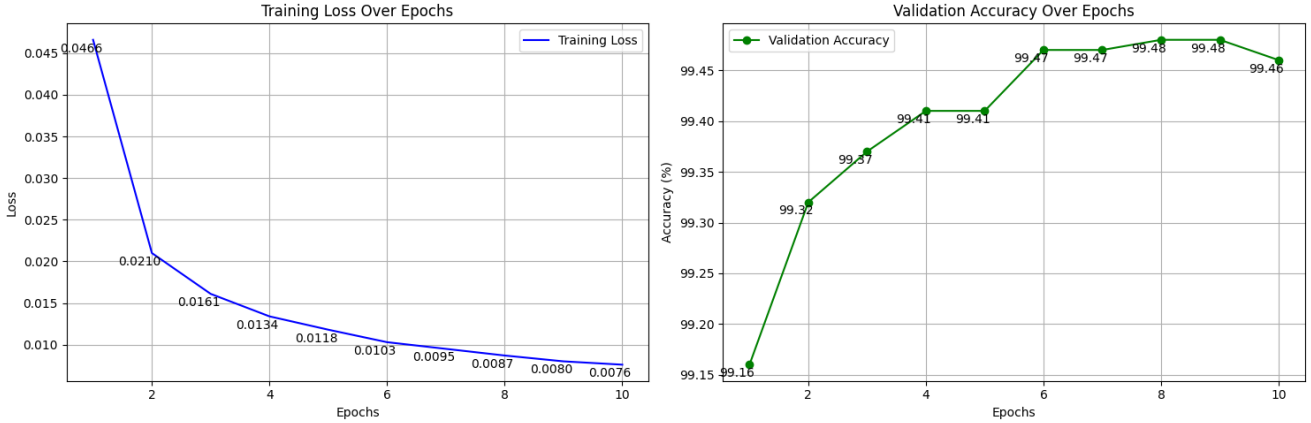
本次实验一共训练了10个epoch，在每个周期中，首先调用train_model函数训练模型，并计算训练损失。接着，使用evaluate_model函数在发展集上评估模型的准确率。训练和验证的结果会在控制台上打印出来，并存储在列表中用于后续的可视化分析。

代码实现：

```
for epoch in range(10):
    loss = train_model(model, train_loader, optimizer, device)
    validation_accuracy = evaluate_model(model, dev_dataset, device)

    print(f"Epoch {epoch+1}, Train Loss: {loss:.4f}, Validation Accuracy:
{validation_accuracy*100:.2f}%")

    # 收集每个epoch的数据
    train_losses.append(loss)
    validation_accuracies.append(validation_accuracy)
```

随后，创建一个包含两个子图的画布。第一个子图用于绘制每个训练周期后的训练损失曲线，显示模型训练过程中损失如何变化。第二个子图展示了每个周期的验证准确率，这有助于观察模型在未见数据上的表现是否有改进或过拟合的迹象。

两个曲线如下：



从训练损失和验证准确率的数据来看，模型的训练表现出了以下几个特点：

1. **损失持续下降**：训练损失值从初始的0.0466逐步降低到0.0076，这表明随着训练的进行，模型在训练数据集上的误差逐渐减小。损失函数值的降低意味着模型对于训练数据的拟合度在提高，学习到了更多的关于训练数据的知识和模式。
2. **准确率高且稳定提升**：验证准确率从99.16%开始，并且在训练过程中稳步上升至99.48%，最后一步略微下降至99.46%。这一趋势表明模型不仅在训练集上有很好的表现，而且在未直接用于训练的验证集上也能泛化得很好。达到接近99.5%的准确率说明模型在预测任务上已经相当精确。
3. **收敛迹象**：虽然训练损失持续减少，但减少的速度逐渐放缓，且验证准确率在最后几次迭代中几乎不再增长，这可能意味着模型开始接近其最佳性能，进一步训练可能不会带来显著的性能提升，同时还有可能导致过拟合的风险。验证准确率的小幅波动也暗示模型可能正处在或已过最优状态。

3.5保存、重新加载模型参数

- **保存参数**

```
# 下载并保存模型到指定目录
save_directory1 = './trains/train1/bert_tokenizer/'
save_directory2 = './trains/train1/bert_model/'
tokenizer.save_pretrained(save_directory1)
model.save_pretrained(save_directory2)
```

- **加载参数**

```
# 指定之前保存的目录路径
saved_tokenizer_directory = './trains/train1/bert_tokenizer/'
saved_model_directory = './trains/train1/bert_model/'
# 重新加载分词器
tokenizer = BertTokenizer.from_pretrained(saved_tokenizer_directory)
# 重新加载模型
model = BertForTokenClassification.from_pretrained(saved_model_directory)
```

4.尝试复现BERT模型

我在GPT的帮助下实现了一个基于BERT的文本分类模型。模型的结构和我加载的预训练模型结构完全一致，主要结构包括：

1. **BertSelfAttention**: BERT模型中的自注意力机制，用于计算注意力分数。
2. **BertSelfOutput**: 自注意力的输出层，包括残差连接和LayerNorm归一化。
3. **BertIntermediate**: BertSelfOutput之后的前馈神经网络部分。
4. **BertOutput**: BertIntermediate之后的输出层，包括残差连接和LayerNorm归一化。
5. **BertLayer**: BERT模型中的一层，包括自注意力机制和前馈神经网络部分。
6. **BertEncoder**: BERT模型中的多层编码器，由多个BertLayer组成。
7. **BertEmbeddings**: BERT模型中的嵌入层，用于将输入的token序列转换为嵌入向量。
8. **BertModel**: 完整的BERT模型，包括嵌入层和多层编码器。
9. **BertForClassification**: 基于BERT的文本分类模型，包括完整的BERT模型、Dropout层和分类器。

4.1整体结构

- **BertForClassification**：
  - **BertModel**：
    - **BertEmbeddings**：嵌入层，将输入的token序列转换为嵌入向量。这个部分除了可以使用自己构建的词向量，也可以使用BERT预训练模型的词向量

    代码实现：

    ```
            # self.embeddings = BertEmbeddings(vocab_size, hidden_size, max_position_embeddings)
            self.embeddings =
    BertForTokenClassification.from_pretrained('./bert_model').bert.embeddings
    ```

    - **BertEncoder**：多层编码器，包括多个BertLayer。
      - **BertLayer**：一层编码器，包括自注意力机制和前馈神经网络部分。
        - **BertSelfAttention**：多头自注意力机制，计算注意力分数。
        - **BertSelfOutput**：自注意力的输出层，包括残差连接和LayerNorm归一化。

- **BertIntermediate**：前馈神经网络部分。
- **BertOutput**：输出层，包括残差连接和LayerNorm归一化。

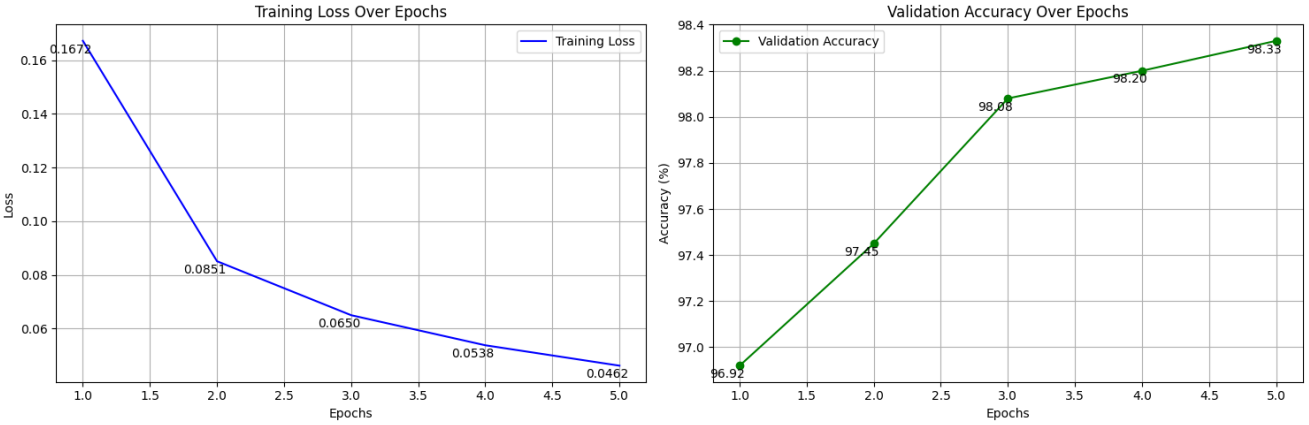
- **BERTClassifier**: 分类任务头，是一个隐藏层，可以实现分类任务

具体代码保存在`BertForClassification.py`文件中

4.2参数设置

```
vocab_size = 21128  # 词汇表大小
hidden_size = 768  # 隐藏层大小
max_position_embeddings = 512  # 序列的最大长度
num_hidden_layers = 4  # Transformer 层的数量 原模型为12层 这里减少一点参数
num_attention_heads = 8  # 注意力头的数量
num_labels = len(tag_set)  # 标签数量
```

4.3训练结果



在我自己搭建的BERT模型上，我减少了Transformer的层数，并且只进行了5轮训练，正确率从96.92%提升到了98.33%。正确率从一开始就很高的原因可能是：

1. 我使用了BERT模型的词嵌入进行训练，该词嵌入性能很强；
2. 大多数标签结果都是o，比较容易学习到一个很高的准确率。

验证集正确率到达98%以后提升缓慢，这可能有以下的原因：

- 模型开始过拟合；
- 模型结构还有待加强。

我们可以尝试增加Transformer的层数，增加模型的复杂度进行改进。

5.BERT+CRF

- **什么是CRF?**

  条件随机场（简称CRF）是一种统计建模方法，主要用于序列数据的标注和分割。在自然语言处理中，CRF常用于命名实体识别、词性标注、语义角色标注等任务。

CRF的主要作用是考虑序列数据中各个元素之间的依赖关系。在一些任务中，如命名实体识别，一个词的标签可能会受到其前后词的标签的影响。例如，在英文中，“New York”通常被标记为一个地名，而不是将“New”和“York”分别标记。这就需要模型能够考虑到序列中的依赖关系，而CRF就是为此设计的。

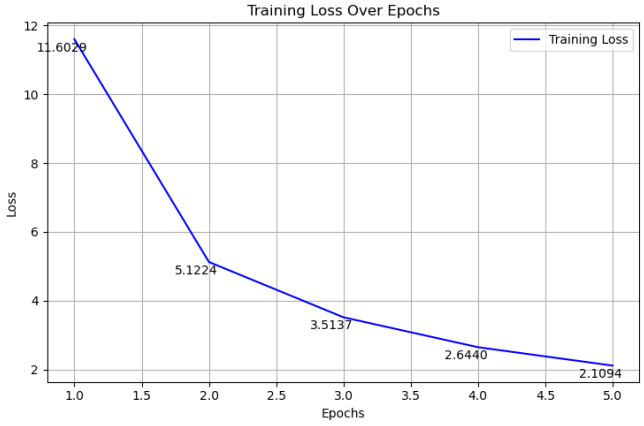
CRF通过在全局范围内进行归一化，并考虑整个序列的标签分布，使得标签之间的转移能够得到合理的建模。这使得CRF在处理序列标注问题时，相比于只考虑局部特征的模型（如隐马尔科夫模型），能够获得更好的性能。总的来说，CRF的作用是在序列标注任务中，考虑并利用标签之间的依赖关系，以提高模型的性能。

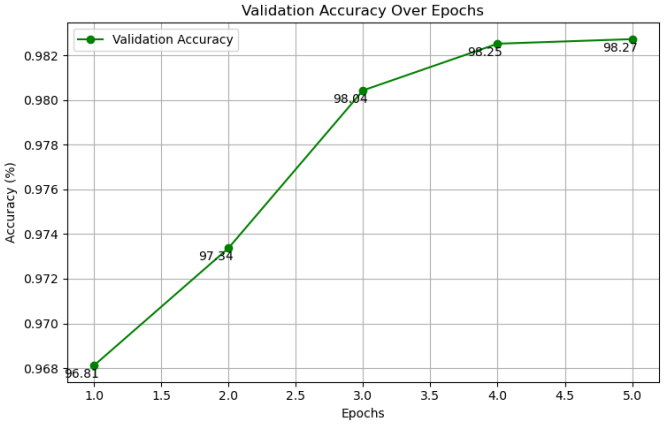
- **实现代码**

```
class BertModelWithCRF(nn.Module):
    def __init__(self, vocab_size, max_position_embeddings, \
            num_hidden_layers, hidden_size, num_attention_heads, num_labels):
        super(BertModelWithCRF, self).__init__()

        self.bert = BertModel(vocab_size, hidden_size, max_position_embeddings,
num_hidden_layers, num_attention_heads)
......
        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(hidden_size, num_labels)
        # CRF layer
        self.crf = CRF(num_labels, batch_first=True) # 设置为True 表示输入的第一个维度是batch_size
(batch_size, sequence_length, num_labels)
......

    def forward(self, input_ids, attention_mask, labels=None):
......
```

CRF的实现很简单，我在自己构建的BERT模型最后多加了一层CRF

- **训练结果**





BERT模型后面加上CRF层，理论上应该能提高模型的性能，特别是在序列标注任务中。但是，我发现模型的性能没有明显提升，个人认为，这可能是因为BERT模型已经能够很好地捕捉到数据的特征，那么添加CRF层可能不会带来显著的性能提升。

6.提交文件

```
2021213346
|
|--- 2021213346.pdf     # 说明文档
|
```

```
|--- 2021213346.txt     # 输出文件
|
|--- code      # 代码
|    |--- BERT.ipynb    # 调用BERT预训练模型进行训练微调的代码
|    |--- BertForClassification.py      # 自己实现的BERT模型和BERT+CRF的代码
|    |--- load_bert.ipynb   # 加载模型参数 输出2021213346.txt文件的代码
|    |--- tag_set.txt   # 保存了所有的标签
|    |--- BERT_2.ipynb  # 训练、测试自己的BERT模型
|    |--- CRF.ipynb     # 训练BERT+CRF模型
```

参考资料

本次实验无参考的论文、网站、代码链接。

本次实验使用了大模型进行辅助。通过大模型，我了解了调用BERT预训练模型的大致方法，以及tokenizer的使用方法。通过询问加快模型训练的方法，我学习了自动管理混合精度的使用方法。还了解了保存、加载模型参数的方法。除此之外的数据处理、环境调试、模型训练、发展集测试和test标签生成均由自己完成。在代码调试的过程中会使用大模型分析错因，但是具体的代码修改由自己完成。