

**课 程 实 验 报 告**

**课程名称： 自然语言处理实验**

**专业班级： 计算机科学与技术202207**

**学 号： U202215561**

**姓 名： 瞿明睿**

**指导教师： 魏巍**

**报告日期： 2025年 5 月 5 日**

**计算机科学与技术学院**

目 录

[1 中文分词实现 2](#_Toc198125476)

[1.1 实验简介 2](#_Toc198125477)

[1.2 字典实现 2](#_Toc198125478)

[1.3 深度学习实现 3](#_Toc198125479)

[1.3.1基于BiLSTM+CRF 3](#_Toc198125480)

[1.3.2基于网络做的基础优化 4](#_Toc198125481)

[1.3.3 网络参数的调整 5](#_Toc198125482)

[1.3.4 bert预训练引入（实现的效果不好） 6](#_Toc198125483)

[1.4 实验结果 8](#_Toc198125484)

[附录A 中文分词实现的源程序 9](#_Toc198125485)

# 1 中文分词实现

## 1.1 实验简介

中文分词指将汉字序列切分成单个词语的过程。在以英文为代表的拉丁语系中，单词之间是以空格作为自然分界符的。而中文只有字、句和段能通过明显的分界符来划界，唯独词没有一个形式上的分界符，这给中文的处理带来了独有的困难，因此中文分词技术得到了广泛研究。

实验要求实现基于词典的分词方法，Bi-LSTM+CRF模型以及进行优化，同时最后对比不同算法分词效果和性能。

## 1.2 字典实现

基于字典的分词器，我们根据实验任务书，完成补全几个基础算法，从整个算法上来看，实现的主要是一个Tokenizer类，包含三种匹配方法，如下

1.fmm\_split正向最大匹配

匹配逻辑：从文本的开头向后扫描。在当前位置时尝试匹配词典中存在的最长词语（不超过最大长度）。如果找不到匹配，则缩短尝试匹配的子串长度，直到找到匹配或子串长度为1（单字成词）。将匹配到的词语加入结果列表，并移动扫描的指针。

2.rmm\_split逆向最大匹配

匹配逻辑：在当前位置，尝试匹配词典中存在的最长词语。如果找不到匹配，则缩短尝试匹配的子串长度，直到找到匹配或子串长度为1。将匹配到的词语插入到结果列表的前端（保持正确的顺序），并移动扫描指针。

3.bimm\_split双向最大匹配

匹配逻辑：分别调用fmm+rmm得到两种分词结果。

同时算法中需要比较结果，如果两种方法分出的词数不同，选择词数较少的结果。如果词数相同，则两种结果中单字词的数量较少的结果。

通过输出分词结果提交后能得到这样的分词方法准确度是在78%-79%之间,具体准确度如下图

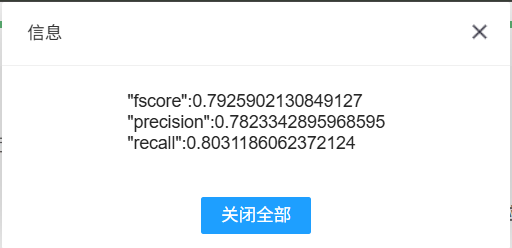


图1 fmm验证

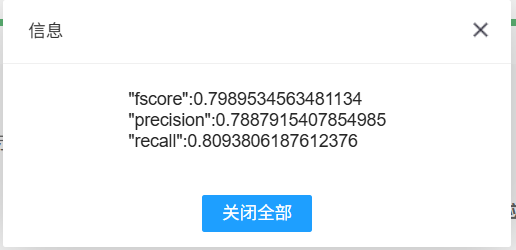


图2 rmm验证

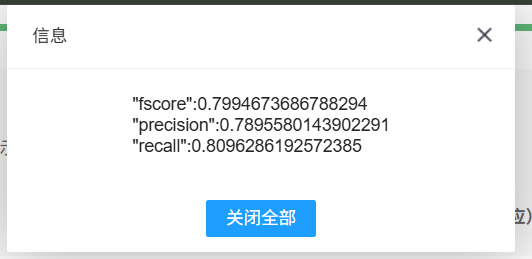


图3 bimm验证

## 1.3 深度学习实现

主要基于实验包中提供的BiLSTM+CRF代码修改和调优

实验环境基于python3.7 pytorch1.7.1 transformers4.0.0软件

实验硬件基于 cuda11.0 2080ti 11G

### 1.3.1基于BiLSTM+CRF

模型结构分为Embedding层+BiLSTM层+CRF层，大概的功能如下

Embedding层将输入的字符映射为稠密向量。

BiLSTM层构建双向LSTM。从两个方向处理输入序列，从而更好地捕捉上下文信息。LSTM的输出会经过一个线性层映射到标签空间的大小。

CRF层接收BiLSTM的输出并计算给定标签序列的条件概率。在训练时计算损失函数；在推断时使用维特比算法解码出最优的标签序列。

同时在分词任务中使用BMES标签

B: Begin (开始)

M: Middle (中间)

E: End (结束)

S: Single (成词)

CRF层中的num\_tags参数4对应这个标签体系的大小。

数据加载与预处理中，将一个批次的数据进行填充以使序列长度一致，并创建掩码在CRF计算中忽略填充部分。数据预先处理成字符ID序列和对应的标签ID序列，并使用pickle保存。

基础的网络实现是很标准的BiLSTM+CRF序列标注任务的标准流程：

字符嵌入 => BiLSTM提取特征 => 线性层映射到标签空间 => CRF解码得到最优标签序列。

通过执行默认参数的训练，我们可以得到训练出来的结果大概是84%左右的准确率可以参看下图：

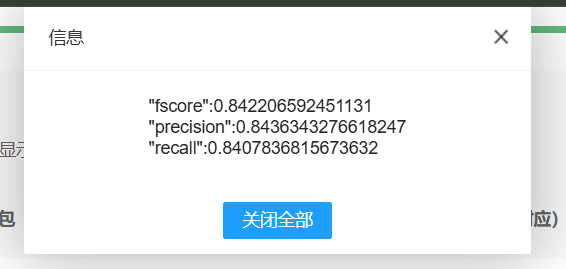


图4 原始网络性能

### 1.3.2基于网络做的基础优化

通过分析需要分词的任务文本，我们可以发现其中包含中文，英文和数字。但是其中数字和英文的部分很少，我们可以针对这一点对模型做一些小的修改

在数据处理中引入字符类型特征，定义了字符类型映射char\_type\_map。在处理数据时为每个字符确定其类型，并将这些类型信息与字、标签一同保存。在CWS模型的初始化函数增加对应需要的参数。同时添加一个新的嵌入层用于学习字符类型的表示。

此外另外添加参数，BiLSTM层设置为num\_layers=2，并加上dropout=0.2，在BiLSTM层与CRF层之间添加一个额外的线性层，ReLU和Dropout层，进一步提取和整合特征。

通过以上的小调整我们可以获得一些些的性能提升如下图

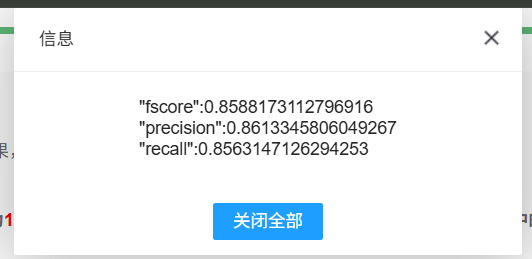


图5 性能提升

我们可以看到大概是有2%左右的提升但是不多，因为并没有做更多的调整。

### 1.3.3 网络参数的调整

我们可以将原先的run.py的

parser.add\_argument('--embedding\_dim', type=int, default=200)

parser.add\_argument('--hidden\_dim', type=int, default=300)

三行进行调整，此时也有效果如下图：

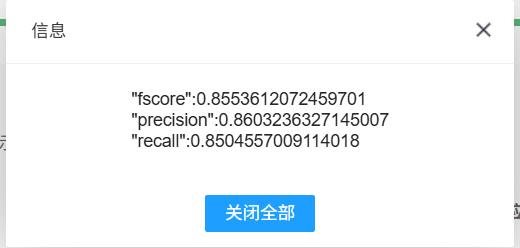


图6 参数调整效果

那么此时我更加激进的给出参数

parser.add\_argument('--embedding\_dim', type=int, default=400)

parser.add\_argument('--hidden\_dim', type=int, default=400)

得到结果如

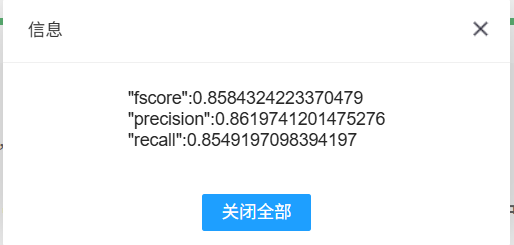


图7 网络参数效果

说明即使在这个基础上，网络能学到的东西并没有增加和层数类似的比例，也说明一律的扩大网络规模不一定能够有效提升网络性能。

同时我们加上基于网络的优化，embedding给到500，隐藏层更到1024可以得到87%正确率，是更好的结果

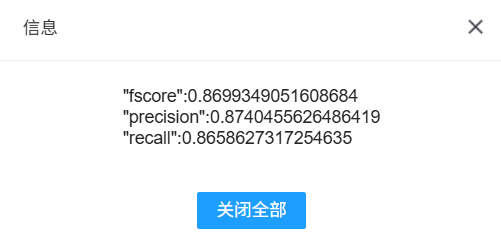


图8 综合调整效果

### 1.3.4 bert预训练引入（实现的效果不好）

我们可以引入预训练的bert模型，比如我们可以使用bert-base-chinese来为输入序列中的每个字符（或子词）生成富含上下文信息的向量表示。BERT的强大之处在于它能够根据上下文理解字符的含义。

我们可以搭建两个模型，直接使用BERT或者构建一个BertBiLSTMCRF

无论是训练还是推理，原本的字符序列都需要通过BERT的BertTokenizerFast

Tokenizer会将输入的文本转换成BERT模型能够理解的格式，主要包括：

输入文本中每个token对应的input\_ids。BERT有自己的词汇表，一个中文字符通常对应一个token，但也可能出现子词切分。特殊token如句子开头和 `句子结尾会被添加。

attention\_mask二进制掩码，指示哪些token是真实的输入，哪些是为了批处理而填充的padding token。

由于BERT的tokenization可能改变原始字符序列的长度，原始的字符级别标签需要与BERT的token对齐，

在训练脚本中调用方法返回一个列表，其中每个元素是当前BERT token对应的原始字符在输入中的索引。利用这个word\_ids，将原始CWS标签映射到每个BERT token上。对于特殊token，赋予一个默认的CWS标签（如'O'的ID）。

对于padding token，设置为-100，这样在计算损失时会被忽略。

在这个BERT处理基础上，我们来构建两个模型

BertCWS:模型直接在BERT的输出之上连接一个CRF层

我们使用一个Dropout层和一个线性层，将BERT的输出映射到CWS标签空间。加入一个CRF层用于序列标注。获取BERT的输出，并通过Dropout和线性层得到发射分数。

BertBiLSTMCRF:模型在BERT和CRF之间增加了一个BiLSTM层，并且可以选择性地整合NER特征。

BERT层处理 input\_ids和attention\_mask得到BERT的token级特征。

当我们需要整合NER特征，训练时传入ner\_labels设置ner\_tagset\_size和 ner\_embedding\_dim> 0，ner\_labels通过\_embeddings层转换成NER嵌入向量。这些NER嵌入向量与BERT的特征维度上拼接。拼接后的特征序列输入到BiLSTM层，然后是线性层和CRF层。

利用Dropout层和一个线性层，将LSTM的输出映射到CWS标签空间。最后加上一个CRF层

拼接后的特征通过 BiLSTM 层。最后通过Dropout和线性层得到发射分数。

实验最后得到的结果并不好，正确率只有76%左右。

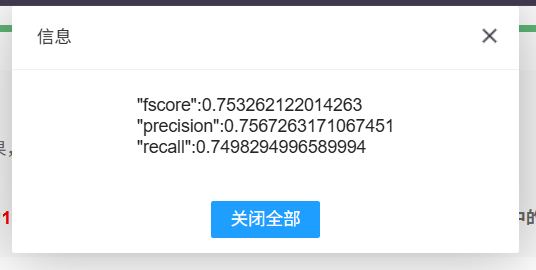


图9 BERT不好的效果

这里探讨一下为什么不好，因为实际上这样的流水线的实现并不好，我一开始并没有多思考这个问题， NER模型的预测准确率直接影响特征质量。如果NER训练的结果本身性能一般，其错误预测会成为CWS模型的噪声输入，反而降低性能。同时由于NER标签是较高层次的语义信息。虽然它们隐含了词边界，但对于CWS任务的直接指导性可能不如更底层的字符类型特征。所以这样一个类似流水线的结构会有错误累积的问题 ，NER预测=>CWS模型使用，错误会在NER模型阶段累积并传递下去。

## 1.4 实验结果

我们可以看到在于字典的实现，三种算法的结果准确度是类似的，后我们利用BiLSTM+CRF，通过细微调整网络结构和参数获得不少的性能提升，相比字典算法是一种很大的进步。

但是在引入BERT这样的预训练模型，我实现的效果并不好，获得了和字典差不多的结果，我分析了原因，但是实际上是因为实现的过程并不优美。

在BERT引入的时， BertBiLSTMCRF : LSTM层数为 args.bert\_lstm\_layers (默认为1)，隐藏层维度为 args.bert\_lstm\_hidden\_dim (256)，所以每层每个方向128。

尽管BERT提供了强大的初始嵌入，但后续的BiLSTM层在 BertBiLSTMCRF 中容量较小。这样设置是由原因的，英文如果在设置和优化过后的BiLSTM一样的隐藏层等参数，可以看到训练10epoch的时间相当多，大概是10小时，类似的我也使用了transformers模型去训练，但是效果依然不好同时使用时间相当多，我放着电脑跑了8-9小时，效果很一般，这样让我意识到：模型复杂，我们需要更细致的超参数搜索，特别是学习率、BiLSTM部分的维度/层数、Dropout率以及NER嵌入维度，其次复杂模型更难训练和调优，任何一个环节（如NER特征的质量、BERT与后续模块的衔接、超参数）出现问题都可能影响整体性能

# 

# 附录A 中文分词实现的源程序

import torch

import torch.nn as nn

import torch.nn.functional as F

from torchcrf import CRF

from torch.nn.utils.rnn import pack\_padded\_sequence, pad\_packed\_sequence

class CWS(nn.Module):

    """Chinese Word Segmentation Model using BiLSTM-CRF"""

    def \_\_init\_\_(self, vocab\_size, tag2id, embedding\_dim, hidden\_dim, char\_type\_vocab\_size, char\_type\_embedding\_dim=20):

        super(CWS, self).\_\_init\_\_()

        self.embedding\_dim = embedding\_dim

        self.hidden\_dim = hidden\_dim

        self.vocab\_size = vocab\_size

        self.tag2id = tag2id

        self.tagset\_size = len(tag2id)

        self.char\_type\_embedding\_dim = char\_type\_embedding\_dim

        self.word\_embeds = nn.Embedding(vocab\_size + 1, embedding\_dim)

        self.char\_type\_embeds = nn.Embedding(char\_type\_vocab\_size, char\_type\_embedding\_dim)

        lstm\_input\_dim = embedding\_dim + char\_type\_embedding\_dim

        self.lstm = nn.LSTM(lstm\_input\_dim, hidden\_dim // 2, num\_layers=2,

                            bidirectional=True, batch\_first=True, dropout=0.2)

        self.dropout = nn.Dropout(0.5)

        self.linear = nn.Linear(hidden\_dim, hidden\_dim)

        self.relu = nn.ReLU()

        self.hidden2tag = nn.Linear(hidden\_dim, self.tagset\_size)

        self.crf = CRF(self.tagset\_size, batch\_first=True)

    def init\_hidden(self, batch\_size, device):

        """初始化LSTM的隐藏状态

        Args:

            batch\_size: 当前batch的大小

            device: 设备类型 (cpu/cuda)

        Returns:

            (h\_0, c\_0): 初始化后的隐藏状态和细胞状态

        """

        return (

            torch.zeros(4, batch\_size, self.hidden\_dim // 2, device=device),

            torch.zeros(4, batch\_size, self.hidden\_dim // 2, device=device)

        )

    def \_get\_lstm\_features(self, sentence, char\_types, length): # 添加 char\_types 参数

        batch\_size, seq\_len = sentence.size(0), sentence.size(1)

        embeds = self.word\_embeds(sentence) # No need for view/reshape if batch\_first=True

        type\_embeds = self.char\_type\_embeds(char\_types)

        combined\_embeds = torch.cat((embeds, type\_embeds), dim=2)

        packed\_embeds = pack\_padded\_sequence(combined\_embeds, length, batch\_first=True, enforce\_sorted=False) # enforce\_sorted=False if dataloader doesn't guarantee sorting

        self.hidden = self.init\_hidden(batch\_size, sentence.device)

        lstm\_out, self.hidden = self.lstm(packed\_embeds, self.hidden)

        lstm\_out, \_ = pad\_packed\_sequence(lstm\_out, batch\_first=True, total\_length=seq\_len) # Ensure output length matches input seq\_len

        lstm\_out = self.dropout(lstm\_out)

        lstm\_out = self.linear(lstm\_out)

        lstm\_out = self.relu(lstm\_out)

        lstm\_feats = self.hidden2tag(lstm\_out)

        return lstm\_feats

    def forward(self, sentence, tags, char\_types, mask, length):

        emissions = self.\_get\_lstm\_features(sentence, char\_types, length)

        mask = mask.bool()

        loss = -self.crf(emissions, tags, mask=mask, reduction='mean')

        return loss

    def infer(self, sentence, char\_types, mask, length):

        emissions = self.\_get\_lstm\_features(sentence, char\_types, length)

        mask = mask.bool()

        return self.crf.decode(emissions, mask=mask)