大作业报告

2023214364 罗一夫

我本次选题为中文分词算法，其中传统匹配算法部分共实现了 3 种算法，不包括神经网络的机器学习算法部分共实现了 2 种算法，深度神经网络部分共实现了 1 种算法。本次实验的配置环境为 win11，python3.7.1，tensorflow-gpu2.2.0，Keras2.3.1.下面给出文件清单：

数据集：

icwb2-data 文件夹

实验一：

base.py --定义匹配算法基类

tire.py --定义树状词典搜索

fmm.py --正向最大匹配

rmm.py --反向最大匹配

chunk.py --定义一段文本的分割属性，用于 mmseg

mmseg.py --mmseg 最大匹配

实验二：

data_process.py --定义几种应用工具，如生成样本标注等

hmm.py --隐马尔科夫模型

unigram_model.py --N-Gram 算法

segment.py --实验一与实验二的接口，直接运行该文件即可查看实验一、二的结果

实验三：

BiLSTM_crf.py --BiLSTM_crf 模型，直接运行即可查看实验三的结果

Save model 文件夹 --存放训练好的模型

sgns.context……bz2 --开源的中文字向量集

生成分词文件：

test_result 文件夹 --traditional 代表实验一，ML_without_nn 代表实验二，ML_with_nn 代表实验三，jieba 代表 jieba 库分词

UI 界面：

Ui_demo.py --直接运行即可

一、 传统匹配算法

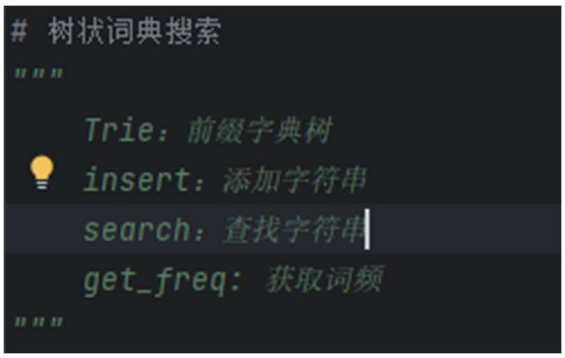
在这个部分，我一共尝试使用了 3 种算法，分别是正向最大匹配 fmm 算法、反向最大匹配 rmm 算法以及 mmseg 算法，它们都属于基于词典的传统中文分词算法。

首先讲一下 fmm 算法的基本思想与代码实现。用一句话概括这种算法，那就是从左到右，用贪心方法匹配出当前位置开始长度最长的词。具体来说，这是一个逐渐迭代的过程，首先根据文本序列得到可能的最大词长，然后从文本开始处选择恰好为最大词长的片段，查看该片段是否匹配

词典中的词汇。假如匹配成功则切分，假如不成功则从序列右侧减少一个字符，查看减少后的序列是否匹配词典中的词汇，以此重复直到匹配出第一个词语；然后，我们再对去除匹配成功词语后的文本进行新一轮的上述重复操作，直到文本全部被切分完成。rmm 算法的思路很类似，只不过是将初始位置变成了文本末尾，并且减少字符是从左到右减少，在此不再赘述。

显而易见，fmm 与 rmm 算法的优缺点都很显著：优点是思想简单，实现难度较低，而且只要词典足够大，算法正确率是有一定保障的；缺点是要求词典规模不能过小，否则无法精准处理未收录词汇，搜索内存要求较高，对歧义词处理不够准确，而且贪心算法会导致遗漏“词中有词”的现象。

为了实现 fmm 算法（rmm 算法高度类似，不再赘述，详情可参见源码），我首先定义了 tire 类（用于树状搜索字典，减少内存）和基类，其中 tire 类的几个基本功能函数如下所示：



然后根据 fmm 算法的思想编写算法，直接嵌套 for 循环即可，核心代码如下：

```python
def cut(self, sentence: str):
    if sentence is None or len(sentence) == 0:
        return []

    index = 0
    text_size = len(sentence)
    while text_size > index:
        word = ''
        for size in range(self.trie.max_word_len+index, index, -1):
            word = sentence[index:size]
            if self.trie.search(word):
                index = size - 1
                break
        index = index + 1
        yield word
```

为了更好地解决 fmm 算法与 rmm 算法中的歧义问题，我另外尝试了 mpseg 算法，下面讲一下它的基本思想与代码实现。mmseg 算法的内核很简单，就是在最大匹配算法的基础上，引入了 4 条消歧规则，来应用于一段最多 3 个词的文本，从而最大程度上地消除歧义问题。这 4 条规则分别是：最大匹配、最大平均词汇长度、最小词长方差和最大单字自由度。显而易见，相比于 fmm 算法与 rmm 算法，mmseg 算法添加了消除分歧的辅助技术，从理论上来说应该有更好的正确率，我们待会进行实验来验证一下；但同样的，mmseg 算法仍然没有脱离词典的限制，其性能与前期准备的词典仍然有很大的关系。

为了实现 mmseg 算法，我首先定义了 chunk 类，用于代表一段文本的一种分割方式，以及几种属性（平均值、方差等），用于应用上述的 4 条消歧规则；之后应用 4 条消歧规则进行评分，选取评分最高的即可，核心代码如下：

```
def get_chunks(self, sentence: str) -> list:
    result = []

    def iter_chunk(sub_sentence, num, tmp_seg_words):
        if (len(sub_sentence) == 0 or num == 0) and tmp_seg_words:
            result.append(Chunk(tmp_seg_words, [self.trie.get_freq(x) for x in tmp_seg_words]))
        else:
            match_words = self.match_all(sub_sentence)
            for word in match_words:
                iter_chunk(sub_sentence[len(word):], num - 1, tmp_seg_words + [word])

    iter_chunk(sentence, num=self.chunk_num, tmp_seg_words=[])
    return result

1 个用法
def match_all(self, sentence: str) -> list:
    words = []
    for i in range(len(sentence)):
        word = sentence[0:i+1]
        if self.trie.search(word):
            words.append(word)
    if len(words) == 0:
        words.append(sentence[0])
    return words

def cut(self, sentence: str):
    if sentence is None or len(sentence) == 0:
        return []

    while sentence:
        chunks = self.get_chunks(sentence)
        word = max(chunks).words[0]
        sentence = sentence[len(word):]
        yield word
```

介绍完基本原理与核心代码，接下来我们利用数据集做一些测试，并比较 3 种方法的性能，性能指标定为 f 参数，衡量工具直接采用数据集自带的 score 脚本。为了增强参考性，我也用 jieba 对测试文本进行分词，jieba 分词模式采用默认的精确模式。

	as	cityu	msr	pku
fmm	0.887	0.872	0.937	0.874
rmm	0.889	0.874	0.935	0.876
mmseg	0.890	0.875	0.936	0.877
jieba	0.738	0.742	0.813	0.818

通过观察实验数据，可以发现 3 个结论：

（1） 几种最大匹配方法效果相近，区别不大

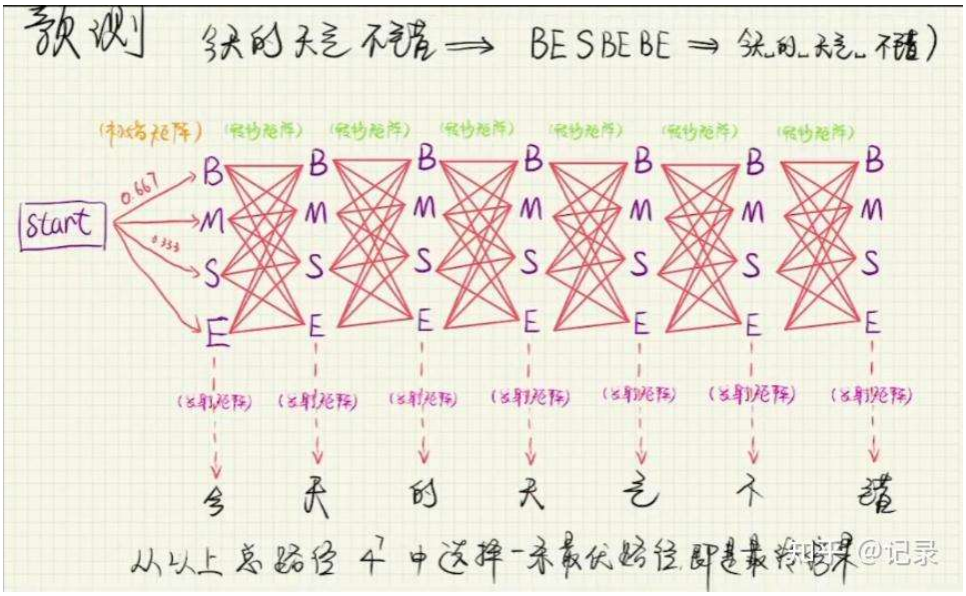
（2） 几种最大匹配方法均优于 jieba

（3） 无论是最大匹配亦或 jieba，其在繁体文本上的性能远远不如其在繁体字上的性能，f1 指数大概有 10%的下降。我猜测这可能涉及到繁体字的文件，无论是训练集、字典亦或测试集，均有更多生僻字和不规范字的出现，从而导致分词算法性能下降。

二、 不含神经网络的机器学习算法

在本部分我采用了两种算法，分别是基于隐马尔可夫模型的 HMM 算法和基于贝叶斯原理的 N-Gram 算法。下面分别对这两种算法进行简单介绍。

从纯数学上讲，隐马尔可夫模型是一个关于时序的概率模型，描述由一个隐藏的马尔可夫链随机生成不可观测的状态序列，再由各个状态生成一个观测而产生观测序列的过程。具体到我们的中文分词问题上，我们可以认为待切分的文本是一个观测序列，而每一段文本都对应着一段状态序列。这样一来，我们就把中文分词的问题，转化成了数学上的求解 HMM 中概率最大的状态序列问题。下图是一个简单的将中文分词转化为概率最大状态序列问题的示意图：



更具体的来说，我们的算法步骤如下：首先遍历数据集每个单词，根据单词特性，为每个单词中的字符打上状态标签，即“S”，“B”，“E”或“M”，从而构建出打上标签的序列（数据集），该部分核心代码如下：

```
# 将原始分词数据集处理成BMES标记的数据集
1个用法
def generate_corpus_status(self, encoding="utf-8"):
    with open(self.data_path, 'r', encoding=encoding) as f:
        for line in f:
            l = []
            for word in line.strip().split():
                if len(word) == 1:
                    l.append((word[0], 'S'))
                    continue
                for i in range(len(word)):
                    if i == 0:
                        l.append((word[i], 'B'))
                    elif i == len(word) - 1:
                        l.append((word[i], 'E'))
                    else:
                        l.append((word[i], 'M'))
            self.corpus.append(l)
    return self.corpus
```

接下来，我们根据上述的标签序列构建索引字典，包括状态集合与其索引相互转换的字典，观测集合与其索引相互转化的字典。有了这种字符与数字之间的联系后，我们就可以依据极大似然估计估计 HMM 的初始矩阵π，

转移概率矩阵 A 和输出概率矩阵 B，这样我们就完成了 HMM 模型的搭建。这一部分的核心代码如下：

```
# 统计频率，计算A，B，Pi参数
for seq in self.idxed_corpus:
    for i in range(len(seq)):
        obsv_cur, hide_cur = seq[i]

        if (i == 0):
            self.Pi[hide_cur] += 1
        else:
            obsv_pre, hide_pre = seq[i - 1]
            self.A[hide_cur, hide_pre] += 1

        self.B[obsv_cur, hide_cur] += 1

# 平滑
if smooth == 'add1':
    self.A += 1
    self.B += 1
    self.Pi += 1

    self.Pi /= self.Pi.sum()
    self.A /= self.A.sum(axis=1)[:, None]
    self.B /= self.B.sum(axis=1)[:, None]

return self.A, self.B, self.Pi
```

搭建好 HMM 模型后，我们就可以采用 Viterbi 算法来解决我们转化出来的“求解 HMM 中概率最大的状态序列问题”，这是一个经典的动态规划算法，可以抽象为数学上的求解最短路径，在此我不多做赘述，核心代码如下：

```
def _veterbi(self, obsv_seq):
    # 初始化
    len_seq = len(obsv_seq)
    f = np.zeros([len_seq, self.num_hide])
    f_arg = np.zeros( shape: [len_seq, self.num_hide], dtype=int)
    for i in range(0, self.num_hide):
        f[0, i] = self.Pi[i] * self.B[obsv_seq[0], i]
        f_arg[0, i] = 0
    # 动态规划求解
    for i in range(1, len_seq):
        for j in range(self.num_hide):
            fs = [f[i-1, k] * self.A[j, k] * self.B[obsv_seq[i], j] for k in range(self.num_hide)]
            f[i, j] = max(fs)
            f_arg[i, j] = np.argmax(fs)
    # 反向求解概率最大的隐藏序列
    hidden_seq = [0] * len_seq
    z = np.argmax(f[len_seq-1, self.num_hide-1])
    hidden_seq[len_seq-1] = z
    for i in reversed(range(1, len_seq)):
        z = f_arg[i, z]
        hidden_seq[i-1] = z
    return hidden_seq
```

以上就是关于 HMM 算法的简单介绍，下面我再简单介绍 N-Gram 算法。

N-Gram 模型对文本数据有一个非常强的假设，那就是当前词语只与它前面的一个词有关，而与其他词完全相互独立。如此一来，整个句子的出现概率就可以简单地计算为各个词语出现概率的乘积，而各个词的概率是可以通过语料中的统计计算得到的。构建词表的核心代码如下：

```
def generate_vocab_dict(self, encoding='utf-8'):
    """
    统计语料库，获取词表
    :param encoding: 编码方式
    :return: 词表字典
    """
    with open(self.data_path, 'r', encoding=encoding) as f:
        print("Start generate vocab dict...")

        for line in f.readlines():
            for word in line.strip().split():
                self.vocab_dict[word] = self.vocab_dict.get(word, 0) + 1
        count = len(self.vocab_dict)
        self.vocab_dict['_total_'] = count
        print("Finished. Total number of words: {0}".format(count))

    return self.vocab_dict
```

构建好词表后，我们直接进行+1 平滑，然后选择其作为我们的模型参数即可。之后，我们对能够产生子句对的标点符号（如顿号、逗号）进行统计，这是我们将为切分句子初步切分为断句的基础。

切分好短句后，利用上一步搭建好的模型，我们就可以用单词乘积的方法对子句进行概率计算，选取概率最大的即可。值得注意的是，为了提高分词的精度，我特意对一些特殊的表达规则进行匹配，例如表示日期，数量等词语规则进行单独处理，从而增加一些通用用语的分词准确率。

同样的，在实验中我们利用数据集做一些测试，并统计 f1 参数。

	as	cityu	msr	pku
hmm	0.78	0.875	0.786	0.788
Unigram	0.884	0.873	0.932	0.920
jieba	0.738	0.742	0.813	0.818

观察实验数据，我们仍然可以发现 hmm 与 Unigram 均优于 jieba 的分词效果。不过这次，在简体中文数据集 msr 和 pku 上，Unigram 体现出的性能要明显好于 hmm，其 f1 参数要高出 15%；然而在繁体中文数据集上，cityu 上两者的性能相差无几，而 as 上仍然是 Unigram 大幅领先。

综合来看，hmm 的性能不尽如人意，不仅比不过 Unigram，也比不过上一节的最大匹配。我尝试分析其中的原因，我认为最大的可能性在于 hmm 模型缺少了词典的辅助，缺少了足够的辅助信息，这让它天然在性能上要弱于最大匹配；而相比于 Unigram，hmm 模型对与一开始的输出矩阵、转移矩阵与初始状态更为敏感，这也决定了假如没有好的初始化，hmm 的效果将不尽如人意。

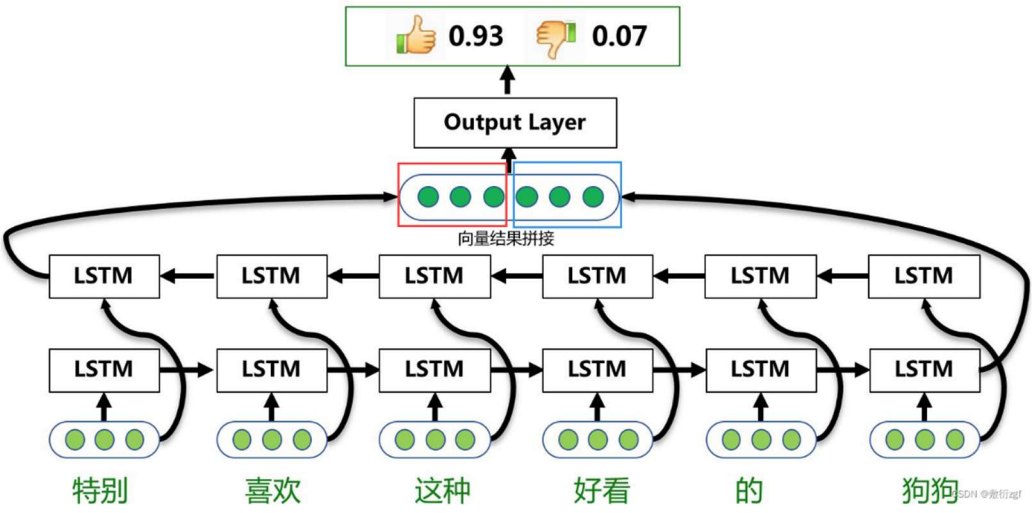
三、 基于深度学习神经网络的算法

本部分我采用了 Bi_LSTM+CRF 来实现。

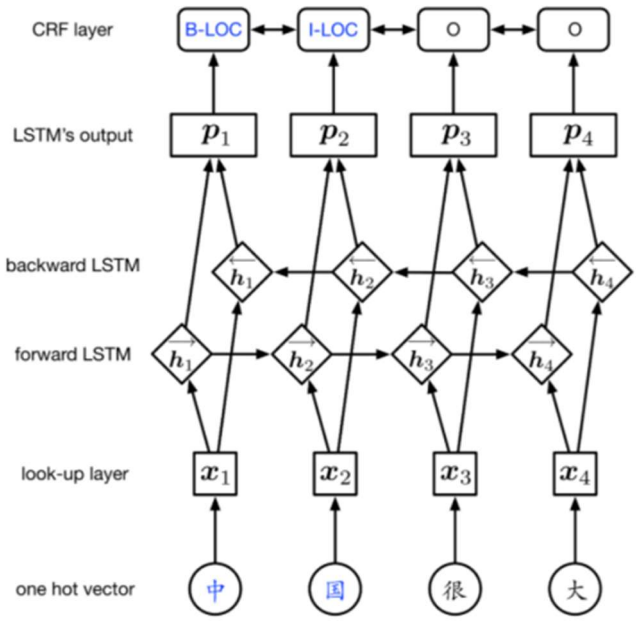
在对中文文本的标签处理上，我直接继承了第二部分 HMM 方法的“BMES”标注方法，只不过这次我采用了深度循环神经网络 RNN 来帮助我完成测试文本的标记。总的来说，思路很简单，就是用训练文本（简单处

理便可获得标注作为标签）来训练神经网络，然后用训好的网络来预测测试文本，通过预测的标签来进行中文分词；但是模型搭建较为复杂，我在下面会进行详细介绍。在本次神经网络的搭建与训练上，我采用了 keras 来实现。

首先从理论上来简单介绍一下我所采用的模型。Bi_LSTM 是双向的 LSTM，它融合了两组学习方向相反的 LSTM 层，然后对其输出进行拼接。所以从理论上来说，其能够实现当前词标签预测既包含历史信息、又包含未来信息，从而让词语标注更加准确。Bi_LSTM 示意图如下所示：



然而，人们在研究中很快就发现，虽然 Bi_LSTM 能够考虑到上下文信息，但也恰恰因为它的这个特点，该模型会导致出现“标签偏置”的问题。标签偏置，指的是模型没有正确理解上下文的潜在信息，反而让预测结果朝着完全相反的方向发展。为了减少这种情况发生的概率，我在网上查阅到，可以选择向模型引入 CRF 层，对 BiLSTM 最后预测的标签增加一些约束规则，从而降低标签偏置的概率。综上，我设计的模型整体示意图如下所示：



总的来说，模型分为输入层、双向 LSTM 层和 CRF 层。输入层负责将一句话的 n 个字做 one-hot 得到稀疏向量，然后通过网上开源的中文字向量集来得到稠密字向量（详见 Embedding/Chinese-Word-Vectors）；双向 LSTM 层则负责迭代训练中文分词，设置了 dropout 以保证随机性，并且在最后通过全连接层输出一个 4 维向量（对应 4 个标签）；CRF 层则是负责构建一个 6*6 的状态转移矩阵，输出的是句子中每个词的标签。

下面简单讲一下核心代码。首先是数据预处理，涉及对数据集转化的字符序列与标签序列进行映射以及进行 padding：padding 过程比较直接，直接设置最大长度，多的截掉短的补齐。核心代码如下：

```python
def process_data(char_list, label_list, vocab, chunk_tags, MAXLEN):
    vocab2idx = {char: idx for idx, char in enumerate(vocab)}
    # 注意 <UNK>
    x = [[vocab2idx.get(char, 1) for char in s] for s in char_list]
    y_chunk = [[chunk_tags.index(label) for label in s] for s in label_list]

    # padding
    x = pad_sequences(x, maxlen=MAXLEN, value=0)
    y_chunk = pad_sequences(y_chunk, maxlen=MAXLEN, value=-1)

    # one_hot编码:
    y_chunk = to_categorical(y_chunk, len(chunk_tags))
    return x, y_chunk
```

加载数据的代码比较简单，很多地方与 HMM 相似，我不在此赘述；另外，由于直接采用开源的中文字向量集，因此词向量直接加载向量机模型然后得出即可；但在词嵌入模块中，一个值得注意的点是，我们额外构建了一个字对应词向量的大矩阵，用于后面的嵌入，具体代码如下：

```python
def make_embeddings_matrix(word2vec_model, vocab):
    char2vec_dict = {}  # 字对词向量
    # vocab2idx = {char: idx for idx, char in enumerate(vocab)}
    for char, vector in zip(word2vec_model.vocab, word2vec_model.vectors):
        char2vec_dict[char] = vector
    embeddings_matrix = np.zeros((len(vocab), EMBED_DIM))
    for i in tqdm(range(2, len(vocab))):
        char = vocab[i]
        if char in char2vec_dict.keys():  # 如果char在词向量列表中，更新权重；否则，赋值为全0
            char_vector = char2vec_dict[char]
            embeddings_matrix[i] = char_vector
    return embeddings_matrix
```

接下来是我们完全从 0 到 1 的 Bilstm_crf 模型构建，全部基于 keras 完成，包含输入层、遮掩层、嵌入层、双向 LSTM 层、Dropout 层、全连接层和 crf 层。结合源代码一开始给出的超参数，我们可以在这个部分对网络结构进行调整，从而探索不同参数与不同优化器对模型效果的影响，详细代码如下：

```
# 构造模型
# 输入层
inputs = Input(shape=(maxlen,), dtype='int32')
# 遮掩层
x = Masking(mask_value=0)(inputs)
# 嵌入层
x = Embedding(len(vocab), EMBED_DIM, weights=[embeddings_matrix], input_length=maxlen, trainable=True)(x)
# 双向LSTM层
x = Bidirectional(LSTM(BiRNN_UNITS // 2, return_sequences=True))(x)
# Dropout: 增加随机性
x = Dropout(0.5)(x)
# 全连接层
x = TimeDistributed(Dense(len(chunk_tags)))(x)
# 输出层，即crf层
outputs = CRF(len(chunk_tags))(x)

# print("Device name:", tensorflow.test.is_gpu_available())
model = Model(inputs = inputs, outputs = outputs)
# 打印模型参数
model.summary()
# 可以试验不同的损失函数
# SGD = keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
# SGDM = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.0, nesterov=False)
# RMSprop = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)
model.compile(optimizer='adam', loss=crf_loss, metrics=[crf_viterbi_accuracy])
```

构建好模型后，我们直接用 keras 的 fit（）函数和 predict（）函数进行训练与测试即可。值得注意的是，由于在之前的 read_file（）中我们已经完成了对测试文本的删除空格，因此我们直接利用“test_gold.utf8”读取测试集即可，不需要再重复读取“test.utf8”，两者最终处理后的数据是一样的。

最后的预测文本存储比较简单，在此不再赘述。有一个点需要提到，我在使用 Tensorflow 框架时，虽然 cuda、cudnn 等环境全部按要求配置完成，但在运行模型加载到 GPU 时总会出现“tensorflow.python.framework.errors_impl.InternalError: Blas GEMM launch failed”的报错，即使升级 Tensorflow 与 keras 版本也无济于事。我在源文件开头的注释

```
import tensorflow as tf

# tensorflow一直无法调用GPU，网上说加上下面这段可以解决，但在我的电脑上没有作用
# physical_devices = tf.config.list_physical_devices('GPU')
# for device in physical_devices:
#     tf.config.experimental.set_memory_growth(device, True)
```

是我在网上查到的解决办法，但在我的电脑上并不起作用。因此，处于 cpu 训练时间过长的考虑，以及可能的调参实验，我的 epoch 仅仅设定为 2，但效果还算不错。加入日后可以搬到 cuda 上跑的话，可以增大 epoch 到十的量级。经过试验，效果最好的参数如下所示，我之后会详细讲述调参过程中各个参数对最终效果的影响：

```
# 超参
BiRNN_UNITS = 200
BATCH_SIZE = 16  # 32、64
EMBED_DIM = 300
EPOCHS = 2 # 因为没法调用GPU所以次数较小，有GPU的情况下可以增大到10次的数量级
optimizer = 'Adam'
```

同样，我们先来看看最优超参下的训练效果：

	as	cityu	msr	pku
Bi_LSTM_crf	0.883	1.00	0.923	0.890
jieba	0.738	0.742	0.813	0.818

可以看到虽然跑的 epoch 很少，但神经网络已经取得了不错的效果，f1 指标已经超出了 jieba，尤其是在 cityu 数据集上得到了 1.00 的高分。我分析这是因为数据集本身的样本已经给的足够多，导致训练几个 epoch，loss 就已经趋于收敛，而打印出来的 loss 也印证了我的观点，在效果最好的 cityu 数据集上，如下图所示，仅仅 2 个 epoch，loss 就已经小于 0.000001 了。

```
47216/47717 [============================>.] - ETA: 4s - loss: 1.1091e-05 - crf_viterbi_accuracy: 0.9992
47232/47717 [============================>.] - ETA: 4s - loss: 8.7112e-06 - crf_viterbi_accuracy: 0.9992
47248/47717 [============================>.] - ETA: 4s - loss: 5.6610e-06 - crf_viterbi_accuracy: 0.9992
47264/47717 [============================>.] - ETA: 3s - loss: 2.6365e-06 - crf_viterbi_accuracy: 0.9992
47280/47717 [============================>.] - ETA: 3s - loss: -3.9598e-07 - crf_viterbi_accuracy: 0.9992
```

接下来讨论一下调参变量对实验的影响。除去网络结构前后牵连影响较大，本次实验共有 3 个可以直接调整不加任何改动的参数：是否对字向

量参数进行训练 train=true/false，BATCH_SIZE =16/32/64 以及优化器=Adam \SGD\RMSprop。考虑到实验耗时，本部分实验我均是在 pku 数据集上，进行 EPOCHS=2 次迭代完成。

首先来看 batch_size，当 train = true 且优化器=Adam 时，我们依次调整 BATCH_SIZE =16/64，得到的数据如下：

BATCH_SIZE	16	64
f1	0.890	0.832

可以发现随着 BATCH_SIZE 大小的增加，f1 指标在减小。仔细想一想这也很合理，适当的 BATCH_SIZE 可以既避免了梯度震荡，又避免了陷入局部极小值，同时又体现出“使用更大的方差来试探最优点的质量”这一强调随机性的思想，因此效果好不足为奇。

我们固定 BATCH_SIZE =16，再来看看 train 的影响。同样设置优化器为 Adam，我们依次调整 train 为 true\false，这时可以通过打印出来的模型信息得到，需要训练的参数：不需要训练的参数=1:5，相比于之前训练参数量大幅下降，因此我猜想模型的效果一定会随之减弱。果然，下面的数据支持了我的猜想：

train	true	false
f1	0.89	0.831

最后我们再来看看优化器的影响。我们固定 BATCH_SIZE =16，train=true，依次调整优化器为 Adam\SGD\RMSprop，来看看他们的效果如何：

优化器	Adam	SGD	RMSprop

f1	0.890	0.42	0.896

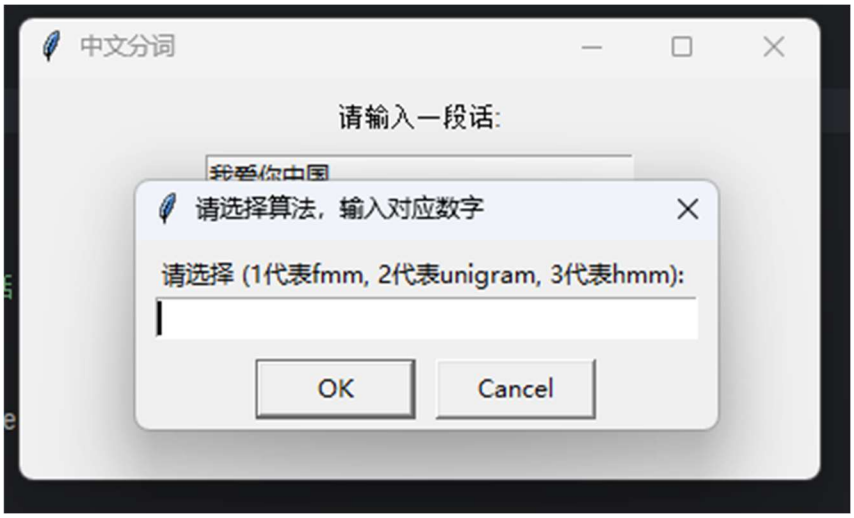
可以看到 Adam 和 RMSprop 效果相近，均大幅领先于 SGD；而 SGD 的发挥可谓糟糕透顶。经过网上查阅资料得知，RMSprop 经常被用于训练 RNN，并且一般来说保持其默认参数就可以让 RNN 获得一个不错的性能；而 SGD 作为随机梯度下降优化器，其对学习率和函数形状都很敏感，有可能是本次学习率设置不太适合的原因。

四、 UI 设计

最后附上几张 UI 界面的截图，直接运行 ui_demo.py 即可。

运行后会弹出界面，需要用户输入待分词语句，并选择算法：

点击选择算法，会让用户输入数字来选择，1 代表前向最大匹配，2 代表 N-Gram，3 代表 hmm，如下所示：



输入数字后点击 OK，下方输出框便会给出结果，如下图: