

# 自然语言处理 中文分词实验报告

姓名：TRY

学号：

专业：计算机科学与技术

时间：2020/11/27

## 一、研究内容




本次任务是使用 $BiLSTM + CRF$ 来实现**中文分词**模型，并在SIGHAN Microsoft Research数据集上进行中文分词的**训练和测试**。

**中文分词**是中文文本处理的一个基础步骤，也是中文人机自然语言交互的基础模块。不同于英文的是，中文句子中没有词的界限，因此在进行中文自然语言处理时，通常需要先进行分词，分词效果将直接影响词性、句法树等模块的效果。当然分词只是一个工具，场景不同，要求也不同。在人机自然语言交互中，成熟的中文分词算法能够达到更好的自然语言处理效果，帮助计算机理解复杂的中文语言。

而在本次实验中，**中文分词**实际上就是**序列标注**问题，即通过 $BiLSTM + CRF$ 模型对数据集中的每一个字打上B,M,E,S的标签，并通过合理有效的标签组合来实现分词。其中，B for "begin"，表示词组的开头的字；M for "middle"，表示词组的中间的字；E for "end"，表示词组最后的字；S for "single"，表示单独成词的字。例如：

	结果
原句子	我周末去中山大学玩
分词结果	我/周末/去/中山大学/玩
标签结果	S BE S BMME S

**训练和测试**基于下面的3个数据集：

名称	修改日期	类型
 msr_test.utf8 <b>测试集</b>	2005/11/18 22:01	UTF8 文件
 msr_test_gold.utf8 <b>测试集groundtruth</b>	2005/8/15 7:47	UTF8 文件
 msr_training.utf8 <b>训练集</b>	2005/6/30 4:14	UTF8 文件

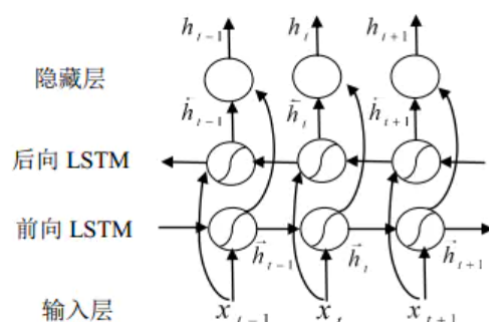
即通过多次迭代后得到的model模型对测试集 msr\_test.utf8 进行预测，并与真实测试集 msr\_test\_gold.utf8 的结果进行比较，计算测试集所有句子**F1**分数的平均值。

## 二、研究方案

$BiLSTM + CRF$ 是NLP序列标注问题的经典模型。本次实验使用了`keras`深度学习框架来实现。

### 2.1 BiLSTM

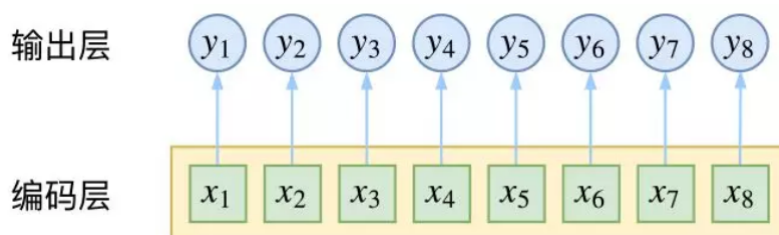
- *BiLSTM*是双向的*LSTM*，融合了两组学习方向相反（一个按句子顺序，一个按句子逆序）的*LSTM*层，能够在理论上实现当前词即包含历史信息、又包含未来信息，更有利于对当前词进行标注。*BiLSTM*在时间上的展开图如下所示：



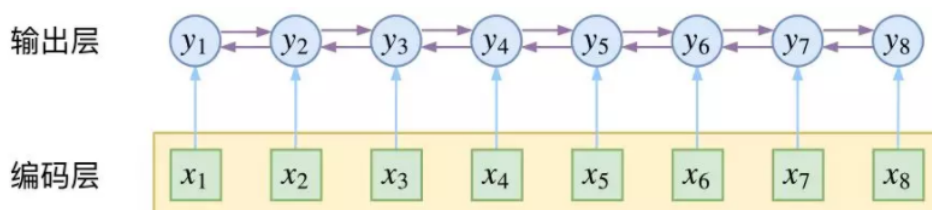
- 若输入句子由120个词组成，每个词由300维的词向量表示，则模型对应的输入是  $(120, 300)$ ，经过*BiLSTM*后隐层向量变为T1  $(120, 100)$ ，其中100为模型中*BiLSTM*的输出维度。如果不使用CRF层，则可以在模型最后加上一个全连接层用于分类。设分词任务的目标标签为 B (Begin)、M (Middle)、E (End)、S (Single)，则模型最终输出维度为  $(120, 4)$  的向量。对于每个词对应的4个浮点值，分别表示对应B\M\E\S的概率，最后取概率大的标签作为预测label。通过大量的已标注数据和模型不断迭代优化，这种方式能够学习出不错的分词模型。
- 然而，虽然依赖于神经网络强大的非线性拟合能力，理论上已经能学习出不错的模型。但是，上述模型只考虑了标签上的上下文信息。对于序列标注任务来说，当前位置的标签 $L_t$ 与前一个位置 $L_{t-1}$ 、后一个位置 $L_{t+1}$ 都有潜在的关系。因此，不能保证标签每次都是预测正确的，会出现**标签偏置**的问题。例如：“我/S 喜/B 欢/E 你/S”被标注为“我/S 喜/B 欢/B 你/S”，由分词的标注规则可知，B标签后只能接M和E，出现了标签偏置问题。
- 对于上述“标签偏置”问题，NLP领域的学者**引入了CRF层**，增加一些约束规则，降低标签偏置的概率。

## 2.2 CRF

- 当我们设计标签时，如用 B,M,E,S 的4个标签来做字标注法的分词，目标输出序列本身会有一些上下文关联，比如 S 后面就不能接 M 和 E，第一个词应该是以 B 或者 o 开头等等。*BiLSTM*的逐标签 softmax 并没有考虑这种输出层面的上下文关联，所以它意味着把这些关联放到了编码层面，希望模型能自己学到这些内容，但有时会“强模型所难”（如下图）。

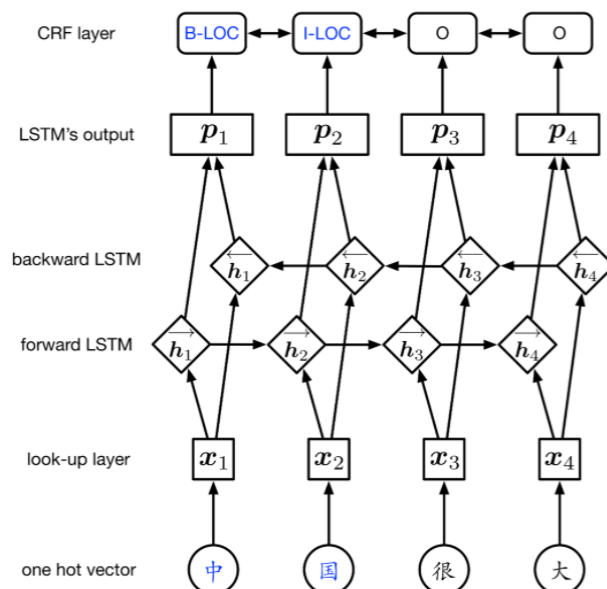


- CRF层将输出层面的关联分离出来，对*BiLSTM*最后预测的标签添加一些约束，来保证预测的标签是合法的。在训练数据训练过程中，这些约束可以通过CRF层自动学习到。



## 2.3 BiLSTM + CRF

- 完整的BiLSTM + CRF的网络结构图如下：



- 输入层：** 比如句子  $x$  有  $n$  个字。首先将  $n$  个字做one-hot得到稀疏向量，再通过look-up table得到  $n$  个单词的  $d$  维稠密词向量。在实验中，可直接通过查找**预训练好的中文字向量集**来得到稠密字向量。因此，经过输入层后，每一个字都由word embedding来构成的向量。
- 双向LSTM层：** 共有  $n * s$  个LSTM模型，用来迭代训练中文分词，得到  $n$  个  $m$  维向量，在设置dropout之后 通过一个全连接层(线性层  $y = w * x + b$ ) 降维到  $k$  维向量( $k$ 也就是标注集的标签数)其中  $p_{ij}$  都视作将字  $x_i$  分类到第  $j$  个标签的打分值。
- CRF层：** 给句子添加起始位和结束位可以构建一个  $(k + 2) * (k + 2)$  的状态转移矩阵，输出的是句子  $x$  中每个单元的标签 (B\I\O\ES)。

## 三、核心代码讲解

以下讲解各部分的核心代码，详细见代码注释。

### 3.1 数据预处理

- 读文件 readfile() 函数：** 实现从utf8文件中读取内容，并获得整个数据集的字列表 `char` 和标签列表 `label`。
  - 其中，调用了get\_char函数和get\_label函数，获得每行的char和label。

```
1 def read_file(file):
2     char, content, label = [], [], []
3     maxlen = 0
4
5     for i in range(len(file)): # 记得加range!!
6         line = file.loc[i,0] # 用loc来访问dataframe
7         line = line.strip('\n') # 去掉换行符
8         line = line.strip(' ') # 去掉开头和结尾的空格
9
10        char_list = get_char(line) # 获得字列表
11        label_list = get_label(line) # 获得标签列表
12        maxlen = max(maxlen, len(char_list))
```

```

13         if len(char_list) != len(label_list):
14             continue # 由于数据集本身问题,要删掉有问题的样本(在训练集中有26个
                        # 样本;测试集中无)
15         char.extend(char_list) # 每一个单元是1个字
16         content.append(char_list) # 每一个单元是一行里面的各个字(分
                        # 好)
17         label.append(label_list) # 每一个单元是一行里面打好标签的结果
                        # (含标点)
18         return char, content, label, maxlen # word是单列表, content和label是双
                        # 层列表
19
20 # 将句子转换成字序列
21 def get_char(sentence):
22     char_list = []
23     sentence = ''.join(sentence.split(' ')) # 去掉空格
24     for i in sentence:
25         char_list.append(i)
26     return char_list
27
28 # 将句子转成BMES序列
29 def get_label(sentence):
30     result = []
31     word_list = sentence.split(' ') # 两个空格来分隔一个词
32     for i in range(len(word_list)):
33         if len(word_list[i]) == 1:
34             result.append('S')
35         elif len(word_list[i]) == 2:
36             result.append('B')
37             result.append('E')
38         else:
39             temp = len(word_list[i]) - 2
40             result.append('B')
41             result.extend('M' * temp)
42             result.append('E')
43     return result

```

- **加工数据函数 process\_data()**，主要涉及对 char 序列和 label 序列进行padding操作。首先，构建 vocab2idx 字典，形成字到序号的映射；并对整个数据集的 char\_list 和 label\_list 中每一个句子的 char 和 label 进行序号的映射，根据最大长度 MAXLEN 进行padding。最后，再对标签列表进行归一化操作。
  - padding调用 pad\_sequences() 函数进行实现，具体操作为：大于 MAXLEN 的进行截断，小于 MAXLEN 的进行padding。且padding是默认的left\_padding。
  - char 列表的默认padding值为0，label 列表的默认padding值为-1。分别表示 <PAD> 和 'E'。
  - 归一化操作具体调用 to\_categorical() 函数实现。

```

1 # process data: padding
2 def process_data(char_list, label_list, vocab, chunk_tags, MAXLEN):
3     vocab2idx = {char: idx for idx, char in enumerate(vocab)}
4     # get every char of every word, map to idx in vocab, set to <UNK>
    if not in vocab
5     x = [[vocab2idx.get(char, 1) for char in s] for s in char_list]
6     # map label to idx

```

```

7     y_chunk = [[chunk_tags.index(label) for label in s] for s in
label_list]
8     # padding of x, default is 0(symbolizes <PAD>). padding
includes:over->cutoff, less->padding. default: left_padding
9     x = pad_sequences(x, maxlen=MAXLEN, value=0)
10    # padding of y_chunk
11    y_chunk = pad_sequences(y_chunk, maxlen=MAXLEN, value=-1)
12    # one_hot:
13    y_chunk = to_categorical(y_chunk, len(chunk_tags))
14    return x, y_chunk

```

- **加载数据 load\_data() 函数：**调用 read\_file() 函数，得到训练集和测试集的字符列表和标签列表，且字符列表有单列表形式的 train\_char，也有双层列表形式的 train\_content。然后再利用 train\_char 和 test\_char 构建的词表 vocab 来对数据集与测试集进行加工，调用 process\_data() 函数完成。
  - 其中，train\_char 是用来和 test\_char 一起形成整个数据集的词表（用来除重使用的）。
  - 并且得到词表 vocab 之后，需要加上两个特殊词 <PAD> 和 <UNK>，形成完整的词表。
  - 在处理测试集的时候，需要得到测试集的最大句子长度 MAXLEN，来作为加工数据中 padding 的长度依据。
    - 实际上，可以取其他的长度作为 MAXLEN。但由于本次实验要求输出测试集的预测结果，因此取测试集的最大句子长度作为 MAXLEN。
  - **注意：**实际上，一开始我只将训练集的词表作为了整个数据集的词表，因此引入了 <PAD> 和 <UNK> 两个符号，其中 <UNK> 表示在测试集中存在但在训练集中不存在的字。但在本次实验的后序调参过程中，我发现将测试集和训练集所有的字一起加起来作为整个数据集的词表的效果更好，因此对此处的 load\_data 进行了修改，因此实际上 <UNK> 在此处已经没有作用了。

```

1  def load_data():
2      chunk_tags = ['S', 'B', 'M', 'E']
3      train_char, train_content, train_label, _ = read_file(train_set)
4      test_char, test_content, test_label, maxlen = read_file(test_set)
5
6      vocab = list(set(train_char + test_char)) # 合并，构成大词表
7      special_chars = ['<PAD>', '<UNK>'] #特殊词表示：PAD表示padding，UNK表示词表中没有
8      vocab = special_chars + vocab
9      # save initial config data
10     with open(SAVE_PATH, 'wb') as f:
11         pickle.dump((train_char, chunk_tags), f)
12     # process data: padding
13     print('maxlen is %d' % maxlen)
14     train_x, train_y = process_data(train_content, train_label, vocab,
chunk_tags, maxlen)
15     test_x, test_y = process_data(test_content, test_label, vocab,
chunk_tags, maxlen)
16     return train_x, train_y, test_x, test_y, vocab, chunk_tags, maxlen,
test_content

```

## 3.2 词嵌入

- **读取预训练的词向量集：**这里，调用了gensim库中的 `KeyedVectors.load_word2vec_format` 函数对词向量集进行读取。

```
1 word2vec_model_path = 'sgns.context.word-character.char1-1.bz2' #词向量位置
2 word2vec_model = KeyedVectors.load_word2vec_format(word2vec_model_path,
  binary=False, unicode_errors='ignore')
```

- **构造整个词表的大词向量矩阵：**这里需要构建一个字对词向量的大矩阵，用于后面的embedding层。且构建的顺序就是每一个字在vocab中的顺序。
  - 如果这个字不在词向量列表中，则将其赋值为全0。

```
1 def make_embeddings_matrix(word2vec_model, vocab):
2     char2vec_dict = {} # 字对词向量
3     vocab2idx = {char: idx for idx, char in enumerate(vocab)}
4     for char, vector in zip(word2vec_model.vocab,
5 word2vec_model.vectors):
6         char2vec_dict[char] = vector
7         embeddings_matrix = np.zeros((len(vocab), EMBED_DIM)) # form huge
8         matrix
9         for i in tqdm(range(2, len(vocab))):
10             char = vocab[i]
11             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（默认）
12                 char_vector = char2vec_dict[char]
13                 embeddings_matrix[i] = char_vector
14     return embeddings_matrix
```

### 3.3 构建BiLSTM + CRF模型

- **构建BiLSTM + CRF模型：**调用 `keras.models` 和 `keras.layers` 中的各个函数进行实现。其中，包括的层有：Input输入层，Masking屏蔽层，Embedding嵌入层（包含加载预训练词向量的操作），Bi-LSTM层，Dropout层（防止过拟合），TimeDistributed全连接层，CRF层。并调用了 `model.summary()` 函数和 `model.compile()` 函数，前者输出model的各项参数信息；后者 compile模型，参数可指定目标函数类型，如 `adam`，`SGD`，`SGDM`，`RMSprop` 等等。

```
1 train_x, train_y, test_x, test_y, vocab, chunk_tags, maxlen,
  test_content = load_data()
2 embeddings_matrix = make_embeddings_matrix(word2vec_model, vocab)
3 # input layer
4 inputs = Input(shape=(maxlen, ), dtype='int32')
5 # masking layer 屏蔽层
6 x = Masking(mask_value=0)(inputs)
7 # embedding layer: map the word to it's weights(with embedding-matrix)
8 x = Embedding(len(vocab), EMBED_DIM, weights=[embeddings_matrix],
  input_length=maxlen, trainable=True)(x)
9 # Bi-LSTM layer
10 x = Bidirectional(LSTM(BiRNN_UNITS // 2, return_sequences=True))(x)
11 # Dropout: 正则化, 防止过拟合.argument means percentage
12 x = Dropout(0.5)(x)
13 # 一维展开, 全连接
```

```

14 x = TimeDistributed(Dense(len(chunk_tags)))(x)
15 # output layer
16 outputs = CRF(len(chunk_tags))(x)
17 # model
18 model = Model(inputs=inputs, outputs=outputs)
19 # print arguments of each layer
20 model.summary()
21 # target_function: includes optimizer, function_type, metrics
22 SGD = keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0,
23 nesterov=False)
23 SGDM = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.0,
24 nesterov=False)
24 RMSprop = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)
25 model.compile(optimizer=RMSprop, loss=crf_loss, metrics=
    [crf_viterbi_accuracy])

```

### 3.4 训练和测试

- **训练函数**：调用fit函数是实现训练。

```

1 # train
2 model.fit(train_x, train_y, batch_size=BATCH_SIZE, epochs=EPOCHS,
3 verbose=1, validation_split=0.1)
4 score = model.evaluate(test_x, test_y, batch_size=BATCH_SIZE)
5 print(score)
6 model.save_weights('model.h5')

```

- **测试函数**：调用predict函数实现预测，并对预测出来的各标签的概率取最大值，得到各个字对应的标签。

```

1 # test
2 model.load_weights('model.h5')
3 test_predict = model.predict(test_x)
4
5 test_predict = [[np.argmax(char) for char in sample] for sample in
6 test_predict] # get the max label_id
7 test_predict_tag = [[chunk_tags[i] for i in sample] for sample in
8 test_predict] # get the label of predic
9 test_gold = [[np.argmax(char) for char in sample] for sample in test_y]
10 # get the label_id
11 test_gold_tag = [[chunk_tags[i] for i in sample] for sample in
12 test_gold] # get the label of real

```

- **计算测试集的F1函数**：计算每个句子的F1值，并对所有的F1值求均值。

- **注意**：由于本次实验F1的定义为

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

$precision$  = 分对词数 / 预测分词结果的总词数

$recall$  = 分对词数 / 正确分词结果的总词数



因此，**不可以使用** `sklearn.metrics` 库中的 `f1_score` 函数进行计算。因为 `f1_score` 函数是对每一个label进行统计的，而本次实验是要对每一个分好的词进行统计，两者要求不同。（实际上，我也对 `f1_score` 进行了计算。发现 `f1_score` 的 macro 版本计算出来的值和我自己设计的函数计算的值相近，大概差1个百分点。）

```
1  _, test_content, _, _ = read_file(test_set)
2  f_sum = 0    # 各个句子的f值之和
3  for i in range(len(test_predict_tag)):
4      correct_word_num = 0    # 分对词数
5      predict_word_num = 0    # 预测分词结果的总词数
6      gold_word_num = 0      # 实际分词结果总词数
7      predict_sample = test_predict_tag[i]
8      gold_sample = test_gold_tag[i]
9      s_len = len(test_content[i])    # the real length of the sentence
10     flag = False    # true: inside a word; false: outside a word
11     for j in range(len(predict_sample) - s_len, len(predict_sample)):
12         if gold_sample[j] == 'S' or gold_sample[j] == 'E' or j ==
len(predict_sample) - 1:    # update gold_word_num
13             gold_word_num += 1
14             if predict_sample[j] == 'S' or predict_sample[j] == 'E' or j ==
len(predict_sample) - 1:    # update predict_word_num
15                 predict_word_num += 1
16                 if gold_sample[j] != predict_sample[j]:
17                     flag = False
18                     continue
19                 elif gold_sample[j] == predict_sample[j] and (gold_sample[j] ==
'S' or (gold_sample[j] == 'E' and flag is True)):
20                     correct_word_num += 1
21                     flag = False
22                 elif gold_sample[j] == predict_sample[j] and gold_sample[j] ==
'B':
23                     flag = True    # inside the word: start
24             precision = float(correct_word_num) / float(predict_word_num)
25             recall = float(correct_word_num) / float(gold_word_num)
26             if precision == 0 and recall == 0: f1 = 0
27             else: f1 = 2 * precision * recall / (precision + recall)
28             f_sum += f1
29     print(f_sum / len(test_predict_tag))
```

## 四、实验步骤设计

本次实验使用 `keras` 深度学习框架进行编写，具体步骤如下：

1. 安装 `keras 2.3.1` 版本和 `tensorflow 2.2` 版本，并安装 `keras contrib` 库和 `gensim` 库。
  - `keras` 和 `tensorflow` 版本一定要对应且不能过高！否则会出现很多奇奇怪怪的报错。
  - `keras contrib` 库是 `keras` 的一个扩展库，包含了封装好的 CRF 层。
  - `gensim` 用于读取预训练好的词向量。
2. 定义超参量 `BIRNN_UNITS`, `BATCH_SIZE`, `EMBED_DIM`, `EPOCHS` 等。
3. 利用 `pandas` 库，读取训练集和测试集的 utf8 文件。
4. 利用 `gensim` 库，读取预训练好的词向量文件。
5. 调用 `load_data()` 函数，读取训练集和测试集的字列表和标签列表，并进行 padding 处理和 one-hot 处理，实现“数据预处理”。



6. 根据预训练的词向量，构建大的词向量矩阵，下标对应 vocab 的顺序，实现“词嵌入”。
7. 构建Bi-LSTM+CRF的model，对训练集进行迭代训练。
8. 利用上步得到的model，对测试集进行预测，并计算F1值。
9. 重复上述过程调参，获得最佳参数。

## 五、实验结果

本次实验的调参变量有：是否对词向量进行训练train=true/false，BATCH\_SIZE =16/32/64，优化器= adam, SGD, SGDM, RMSprop。

且由于keras训练速度较慢，使用cuda或者纯cpu下都需要近半小时完成一次迭代，所以每种参数组合只迭代了5次进行比较。

### 5.1 是否对预训练好的词向量进行训练更新

- 在Bi-LSTM+CRF的模型中，构建Embedding层的代码如下，其中，trainable表示是否训练词向量参数。

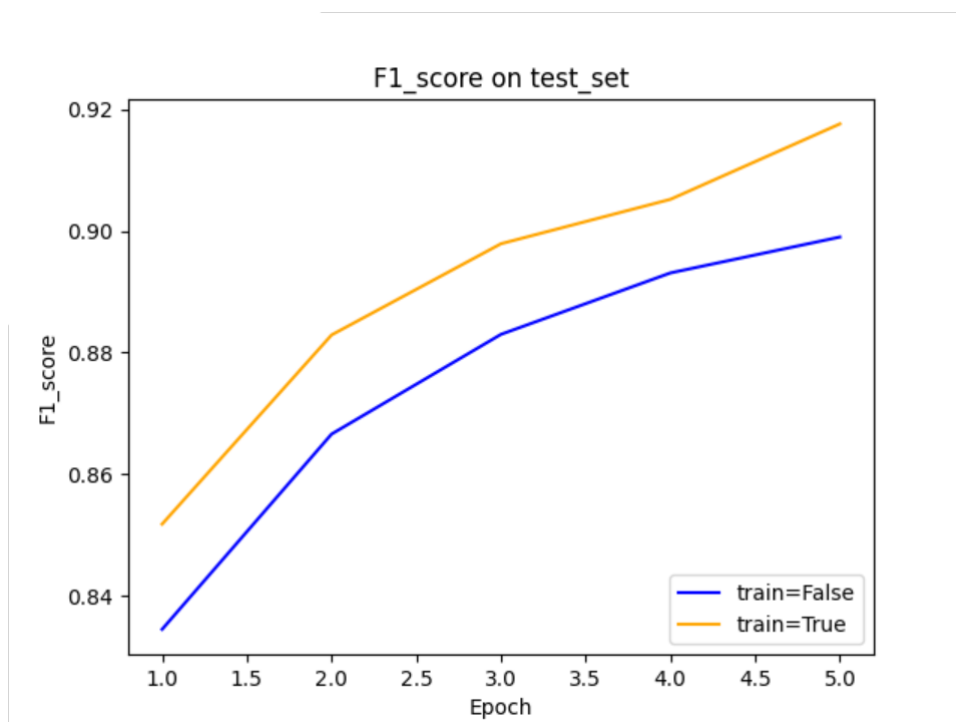
```
1 # embedding layer: map the word to it's weights(with embedding-matrix)
2 x = Embedding(len(vocab), EMBED_DIM, weights=[embeddings_matrix],
    input_length=maxlen, trainable=True)(x)
```

- trainable=False表示不对预训练的词向量进行训练，而trainable=True表示对预训练的词向量参数进行后续的训练更新。发现当不对词向量参数进行训练时，模型参数如下：

```
-----
Total params: 1,875,948
Trainable params: 321,648
Non-trainable params: 1,554,300
```

训练参数：非训练参数=1:5。

- 当优化器为 adam，BATCH\_SIZE=64的情况下，结果如下：



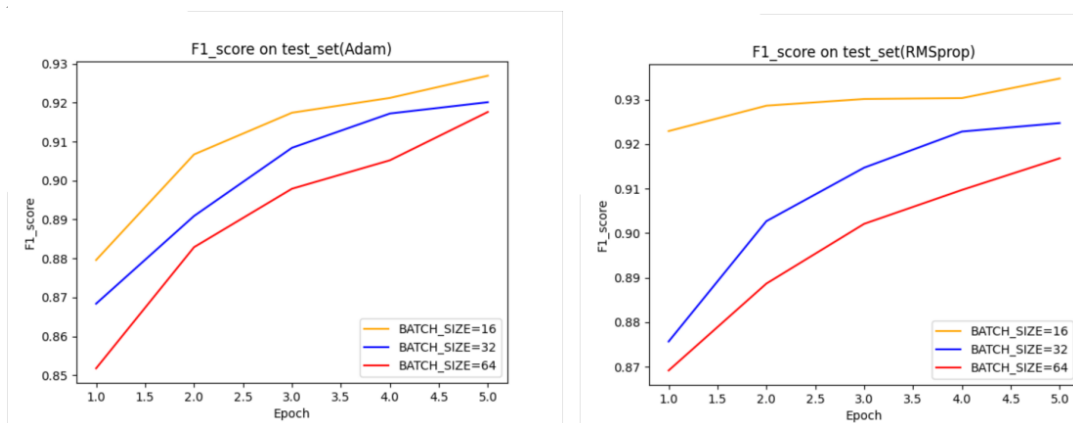
可以看出，train=True的效果明显优于train=False的效果，即训练词向量的效果要优于不训练的效果。

## 5.2 BATCH\_SIZE

- 在训练的代码部分，调用了 `model.fit` 函数，可以成批训练数据，参数为 `batch_size`。

```
1 model.fit(train_x, train_y, batch_size=BATCH_SIZE, epochs=1, verbose=1,
  validation_split=0.1)
```

- 因此，我对批数据的大小 `BATCH_SIZE` 进行了调参，分别取16,32,64进行了实验。当优化器为 'adam' 或者 'RMSprop'，train=true时，结果如下：



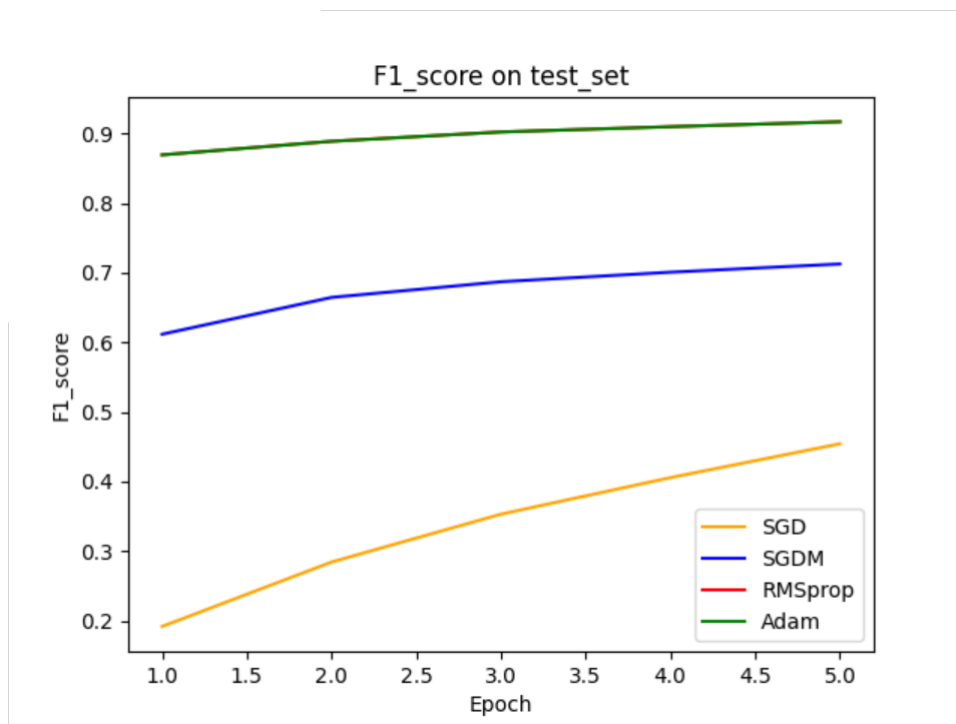
**结论：**在这两个优化器中，共同规律是随着 `BATCH_SIZE` 的大小的增加，`F1_score` 的大小在逐渐减小，因此，`BATCH_SIZE=16`的效果最好。

## 5.3 优化器的选择

- 在代码中，优化器实际是作为“目标函数”：

```
1 SGD = keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0,
  nesterov=False)
2 SGDM = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.0,
  nesterov=False)
3 RMSprop = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)
4 # model.compile(optimizer=RMSprop, loss=crf_loss, metrics=
  [crf_viterbi_accuracy])
5 # model.compile(optimizer=SGD, loss=crf_loss, metrics=
  [crf_viterbi_accuracy])
6 # model.compile(optimizer=SGDM, loss=crf_loss, metrics=
  [crf_viterbi_accuracy])
7 model.compile(optimizer='adam', loss=crf_loss, metrics=
  [crf_viterbi_accuracy])
```

- 因此，我对优化器进行了调参，分别取了 'adam', 'SGD', 'SGDM', 'RMSprop' 进行实验。当 `BATCH_SIZE=64`，train=true时，结果如下：



**结论：**可以发现，SGD 优化器在本次实验中表现非常不好，SGDM 表现一般，RMSprop 和 Adam 的表现非常好（红色线和绿色线重合）。

- 经过查询资料，发现：
  - SGD 是随机梯度下降优化器，SGDM 是 SGD 的动量版本；
  - RMSprop 通常是训练循环神经网络RNN的不错选择，且建议使用优化器的默认参数。
  - Adam 本质上是 RMSprop 与动量 momentum 的结合。

## 5.4 最佳参数

因此，本次实验使用`keras`框架时的最佳参数为：

参数	最佳取值
BATCH_SIZE	16
Optimizer	Adam
BiRNN_UNITS	200 (=100+100)
Epoch	5
F1_score	0.9302

此时，F1的值为0.9302，分词结果实例如下：

句子编号	类别	结果
3	正确分词	海运业雄踞全球之首，按吨位计占世界总数的17%。
	实际分词	海运业雄踞全球之首，按吨位计占世界总数的17%。
6	正确分词	十几年来，改革开放的中国经济高速发展，远东在崛起。
	实际分词	十几年来，改革开放的中国经济高速发展，远东在崛起。
21	正确分词	近几年来，兼从事社会工作及社会保障问题研究。
	实际分词	近几年来，兼从事社会工作及社会保障问题研究。

结果保存为文件：`msr_test_predict.txt`和`msr_test_predict.utf8`两种格式。

## 六、加分项

在本次实验中，加分项为：使用预训练词向量，模型超参数组合，选择不同的目标函数等。（详见第五部分的实验结果）