北京大学

自然语言处理导论
实验报告

虞梦夏
1600014403
中国语言文学系
2018年12月

目录

LSTM实现中文分词

虞梦夏

1 实验目标

编程实现一个LSTM模型来完成中文自动切词任务。

2 实验数据

- 来自Sighan-2004 中文切词国际比赛的标准数据。
- 训练数据为86924 个句子，大小为16M。
- 测试数据为3985 句子，大小为0.6M。
- 测试数据的答案为test.answer.txt，用于计算系统的Precision，Recall，F-score。

3 实验环境

Windows 10 64位

4 实验过程

本实验使用tensorflow库，采取双层双向LSTM模型进行序列标注。

4.1 数据预处理

观察训练集，可见词边界以词后空格标记。
读入训练集，采用“BSME”4标签，根据空格为每个字符贴上标签，标点符号标为“S”。

```python
def to_tagged_texts(raw_text):
    '''
    训练集去掉空格，打上标签
    采用4标签。tags = {"S": 1, "B": 2, "M": 3, "E": 4}
    :param raw_text: list.带空格的训练集文本
    :return: list.每个字符后打上标签的文本
    '''
    tagged = []
    for line in raw_text:
        output_line = ""
        for i in range(len(line)):
            if line[i] not in [" ", "\r", "\n"]:
                # 句子长度大于60，则按标点符号分割
                if len(output_line) > 60 and line[i] in Config.punctuations:
                    output_line += f"{line[i]}/S "
                    tagged.append(output_line)
                    # tagged.append(output_line)
                    output_line = ""

                # 否则按换行符分割
                elif i == 0:  # 行首
                    if line[i + 1] == ' ':
                        output_line += f"{line[i]}/S "
                    else:
                        output_line += f"{line[i]}/B "
                elif line[i - 1] == ' ':  # 不在行首且前面是空格
                    if line[i + 1] in [' ', '\r','\n']:  # 行尾或者后面是空格
                        output_line += f"{line[i]}/S "
                    else:
                        output_line += f"{line[i]}/B "
                elif i == len(line) - 1 or line[i + 1] == ' ':  # 前面不是空格，行尾或者后面是空格
                    output_line += f"{line[i]}/E "
                else:
                    output_line += f"{line[i]}/M "
        if output_line != "":
            tagged.append(output_line)
    return tagged
```

分离字符和标签，通过字典结构将其转化为一一对应的索引值。

```python
def split(tagged):
    '''
    分离出标签和训练数据，生成字符字典
    :param tagged: list. 带标签的训练文本
    :return: 2darray：train_X and train_Y. dictionary: char_id
    '''
    train_X = []
    train_Y = []
    char_id = {}
    for sent in tagged:
        x = []
        y = []
        word_units = sent.split(" ")
        for unit in word_units:
            if len(unit.split("/")) == 2:
                char, tag = unit.split("/")
                if char not in char_id:
                    char_id[char] = len(char_id) + 1  # 索引值从1开始

                x.append(char_id[char])
                y.append(Config.tags[tag])

        train_X.append(x)
        train_Y.append(y)
    return np.array(train_X), np.array(train_Y), char_id
```

将生成的字典保存为json文件。将训练数据和标签保存为npy文件。

4.2 神经网络框架

4.2.1 输入层

读入数据，由于使用定长序列RNN，要先把序列补齐（padding）到固定的长度，这里选取了70作为序列长度（以自然标点和换行符切分句子后，测试集的最长序列不超过70）。

```
def padding(data_x, maxlen):
    padded_x = []
    for line in data_x:
        if len(line) <= maxlen:
            padded_x.append(line + [0 for _ in range(maxlen - len(line))])
        else:
            padded_x.append(line[:maxlen])
    return np.array(padded_x, dtype=np.int32)
```

实例化Dataset和Iterator，为后续训练构造好分批次的数据，详见代码。

4.2.2 字向量层

通过tensorflow的查表功能，将每个字符用64维的向量表示。

```
# embedding layer
with tf.variable_scope('embedding'):
    embedding_matrix = tf.Variable(tf.random_normal([FLAGS.dict_size, FLAGS.embedding_size], -1.0, 1.0))
    inputs = tf.nn.embedding_lookup(embedding_matrix, x)
```

4.2.3 LSTM层

采用双层双向LSTM。以tf.nn.rnn_cell.LSTMCell()作为LSTM基本单元，在此基础上构造前向、后向LSTM，第一层输出拼接在一起之后作为下一层的输入。

```
# LSTM layer
keep_prob = tf.placeholder(tf.float32, [])
cell_fw = [lstm_cell(FLAGS.num_units, keep_prob) for _ in range(FLAGS.num_layer)]
cell_bw = [lstm_cell(FLAGS.num_units, keep_prob) for _ in range(FLAGS.num_layer)]
inputs = tf.unstack(inputs, FLAGS.time_step, axis=1)
output, _, _ = tf.contrib.rnn.stack_bidirectional_rnn(cell_fw, cell_bw, inputs=inputs, dtype=tf.float32)
output = tf.stack(output, axis=1)  # output变形回来 shape=(batch_size, time_step, 2,  output_size)
print('Output', output)
output = tf.reshape(output, [-1, FLAGS.num_units * 2])  # shape = (batch内总字数, 2倍units数)
print('Output Reshape', output)
```

4.2.4 输出层

经过线性变换得到最后一维为标签数的向量，表示属于各个标签的得分。

```
# output layer
with tf.variable_scope('outputs'):
    w = weight([FLAGS.num_units * 2, FLAGS.category_num])
    b = bias([FLAGS.category_num])
    y = tf.matmul(output, w) + b  # shape = (batch内总字数, 类别数)
    y_scores = tf.reshape(y, [-1, FLAGS.time_step, FLAGS.category_num])
    print('y scores', y_scores)
```

4.2.5 标签推断

用条件随机场(CRF)方法求出最大可能的路径，训练模式下学习概率转移矩阵，并求对数似然。

```
# decode and count loss
else:
    with tf.variable_scope('tag_inf'):
        log_likelihood, transition_params = tf.contrib.crf.crf_log_likelihood(y_scores, y_label,seq_lens)
    y_predict, _= tf.contrib.crf.crf_decode(y_scores, transition_params,seq_lens)

    with tf.variable_scope('loss'):
        loss_loglikelihood = tf.reduce_mean(-log_likelihood)
        tf.summary.scalar('loss', loss_loglikelihood)

    accuracy = count_accuracy(y_predict, y_label)
```

在预测模式下直接用学到的概率转移矩阵进行标签推断。

```
# inference
if not FLAGS.train:
    with tf.variable_scope('tag_inf'):
        transition_params = tf.get_variable('transitions', shape=[FLAGS.category_num,FLAGS.category_num])
    y_predict,_ = tf.contrib.crf.crf_decode(y_scores, transition_params,seq_lens)
    print('y predict', y_predict)
```

4.2.6 反向传播训练

以CRF中的对数似然作为损失函数，采用Adam优化方法反向传播调整各层的参数。

```
train = tf.train.AdamOptimizer(FLAGS.learning_rate).minimize(loss_loglikelihood, global_step=global_step)
```

最后用tf.train.Saver()保存训练好的模型。

4.3 预测分词结果

4.3.1 数据预处理

将测试集按标点符号和换行符切碎，用之前保存的字典将字符转成索引值，OOV一律用"字典大小-1"表示。

```
def testsetprocess(test_txt):
    '''
    测试集预处理
    :param test_txt: 测试集文本
    :return: 切碎的文本和转换为索引值的array
    '''
    with open(test_txt, 'r', encoding='utf-8') as f:
        text = f.readlines()

    testtext = []
    for line in text:
        # 长度小于50则按换行符切割
        if len(line) < 50:
            testtext.append(line)
        # 否则按标点符号切割
        else:
            sent = ""
            for char in line:
                sent += char
                if char in line[:-5] and char in Config.punctuations:
                    sent += "\t"  # '\t'为切割标记
            testtext.append(sent)
    short_text = []
    for str in testtext:
            short_text.extend(str.split("\t"))
    test_X = np.array(test_label(short_text))
    short_text = np.array(short_text)

    # 保存数据
    np.save('../data/test_X.npy', test_X)
    np.save('../data/test_text.npy', short_text)
    print('Test Data Saved')
```

4.3.2 预测标签

加载之前保存的模型，把测试数据做padding之后扔进模型里，batch size和epoch设为1，跑。生成的标签序列保存为npy文件。

4.3.3 生成分词结果

切碎的测试集文本和预测的标签序列按位置对照，标点符号后面一律打上空格，预测为“S”和“E”的字符后面打上空格，即为预测的分词结果。

```python
def generate_text(test_text, test_Y, outputfile):
    # tags = {"S": 1, "B": 2, "M": 3, "E": 4}
    text = []
    for i in range(len(test_text)):
        line = ""
        for j in range(len(test_text[i])):
            # add a strong rule: cut when come across punctuations
            if test_text[i][j] in Config.punctuations:
                line += f'{test_text[i][j]} '
            elif test_Y[i][j] in [1,4,0]:
                line += f'{test_text[i][j]} ' # cut
            else:
                # test_Y[i][j] in [2,3]:
                line += f'{test_text[i][j]}' # do not cut

        text.append(line)
    # 保存预测结果
    with open(outputfile,'w', encoding='utf-8') as f:
        f.writelines(text)
```

5 实验结果

通过score脚本评测分词效果。

某次结果保存在"./data/score4.txt"中，命令行日志保存在"./data/log_output.txt"中。F score为0.936，可见效果一般。

```
=== SUMMARY:
=== TOTAL INSERTIONS:  2643
=== TOTAL DELETIONS:   1963
=== TOTAL SUBSTITUTIONS:   4557
=== TOTAL NCHANGE: 9163
=== TOTAL TRUE WORD COUNT:   106873
=== TOTAL TEST WORD COUNT:   107553
=== TOTAL TRUE WORDS RECALL:    0.939
=== TOTAL TEST WORDS PRECISION:   0.933
=== F MEASURE: 0.936
=== OOV Rate:  0.026
=== OOV Recall Rate:    0.667
=== IV Recall Rate: 0.946
### test_output4.txt    2643 1963 4557 9163 106873 107553 0.939    0.933    0.936    0.026    0.667    0.946
```

当时的参数选择：

```python
flags.DEFINE_integer('train_batch_size', 256, 'train_batch_size')
flags.DEFINE_integer('dev_batch_size', 256, 'dev batch size')
flags.DEFINE_integer('test_batch_size', 1, 'test batch size')
flags.DEFINE_integer('dict_size', 6000, 'dict_size')
flags.DEFINE_integer('category_num', 5, 'category number')
flags.DEFINE_float('learning_rate', 0.001, 'learning_rate')
flags.DEFINE_integer('num_units', 64, 'the number of units in LSTM cell')
flags.DEFINE_integer('num_layer', 2, 'num_layers')
flags.DEFINE_integer('time_step', 70, 'timestep_size')
flags.DEFINE_integer('epoch_num', 30, 'epoch_num')
flags.DEFINE_integer('epochs_per_dev', 2, 'epoch per dev')
flags.DEFINE_integer('epochs_per_save', 2, 'epoch per save')
flags.DEFINE_integer('steps_per_print', 50, 'steps per print')
flags.DEFINE_integer('steps_per_summary', 50, 'steps per summary')
flags.DEFINE_integer('embedding_size', 64, 'embedding size')
flags.DEFINE_string('summaries_dir', '..\\summaries\\', 'summaries dir')
flags.DEFINE_string('checkpoint_dir', '..\\ckpt\\model.ckpt', 'checkpoint dir')
flags.DEFINE_float('keep_prob', 0.5, 'keep prob dropout')
flags.DEFINE_boolean('train', False, 'train or test')
```

观察发现主要问题是切分粒度过细。对4字以上多字词识别较好，但部分2字词和3字词被错误地断开了。

6 实验总结

本实验实现了一个粗糙的LSTM自动中文分词任务。虽然学习tf的过程很痛苦，但是受益匪浅。

项目代码和文件已上传至GitHub。地址：`https://github.com/xyliaaaaa/nlpws`

6.1 模型调优

1、Adam比Adadelta收敛快很多。

2、batchsize试过128到400，256左右效果最好。

3、timestep很影响训练速度。32或者50会比我选的70快很多，但是测试集分割之后的序列长度最大是69。训练和测试时间步长不相等的情况没有测试。

4、防止过拟合：设置LSTM的keep_prob（本文只尝试了0.5），训练集准确率还在提高但验证集准确率不再提高时终止训练，等等。

6.2 可能的改进

1、在设备允许的条件下grid search或者手动尝试更多的参数组合。诸如LSTM的层数、unit数、embedding维度等等。损失函数、优化方法也还可以进行选择。

2、Ensemble model？

3、加入Attention机制？

这都有待实验验证。