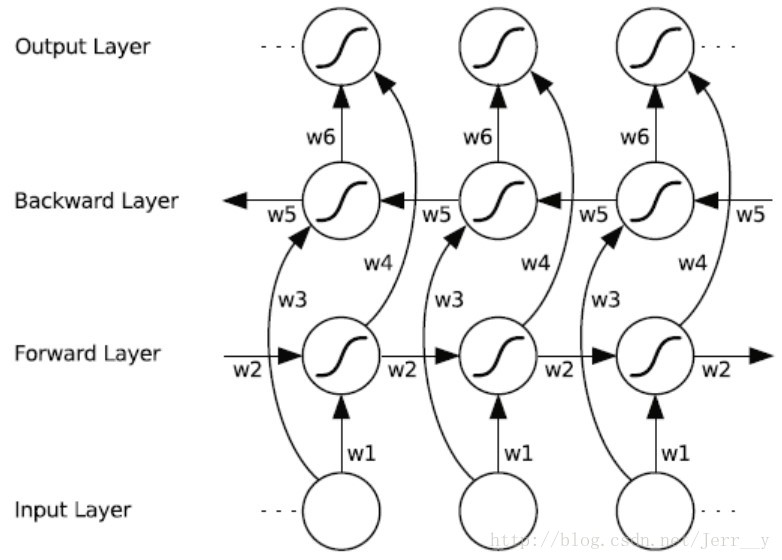
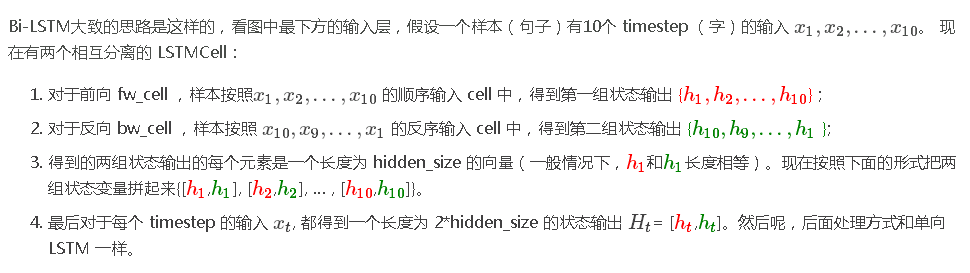
Bi-LSTM

我们知道 RNN 是可以学习到文本上下文之间的联系的，输入是上文，输出是下文，但这样的结果是模型可以根据上文推出下文，而如果输入下文，想要推出上文就没有那么简单了，为了弥补这个缺陷，我们可以让模型从两个方向来学习，这就构成了双向 RNN。在某些任务中，双向 RNN 的表现比单向 RNN 要好，本文要实现的文本分词就是其中之一。不过本文使用的模型不是简单的双向 RNN，而是 RNN 的变种 — LSTM。

Bi-LSTM 的基本原理，输入层的数据会经过向前和向后两个方向推算，最后输出的隐含状态再进行 concat，再作为下一层的输入，原理其实和 LSTM 是类似的，就是多了双向计算和 concat 过程。





数据处理

本文的训练和测试数据使用的是已经做好序列标注的中文文本数据。序列标注，就是给一个汉语句子作为输入，以“BEMS”组成的序列串作为输出，然后再进行切词，进而得到输入句子的划分。其中，B 代表该字是词语中的起始字，M 代表是词语中的中间字，E 代表是词语中的结束字，S 则代表是单字成词。

这里的原始数据样例如下：

人/b 们/e 常/s 说/s 生/b 活/e 是/s 一/s 部/s 教/b 科/m 书/e

这里一个字对应一个标注，我们首先需要对数据进行预处理，预处理的流程如下：

将句子切分

将句子的的标点符号去掉

将每个字及对应的标注切分

去掉长度为 0 的无效句子

首先我们将句子切分开来并去掉标点符号，代码实现如下：

# Read origin data

text = open('data/data.txt', encoding='utf-8').read()

# Get split sentences

sentences = re.split('[，。！？、‘’“”]/[bems]', text)

# Filter sentences whose length is 0

sentences = list(filter(lambda x: x.strip(), sentences))

# Strip sentences

sentences = list(map(lambda x: x.strip(), sentences))

这样我们就可以将句子切分开来并做好了清洗，接下来我们还需要把每个句子中的字及标注转为 Numpy 数组，便于下一步制作词表和数据集，代码实现如下：

import re

# To numpy array

words, labels = [], []

print('Start creating words and labels...')

for sentence in sentences:

groups = re.findall('(.)/(.)', sentence)

arrays = np.asarray(groups)

words.append(arrays[:, 0])

labels.append(arrays[:, 1])

print('Words Length', len(words), 'Labels Length', len(labels))

print('Words Example', words[0])

print('Labels Example', labels[0])

这里我们利用正则 re 库的 findall() 方法将字及标注分开，并分别添加到 words 和 labels 数组中，运行效果如下：

Words Length 321533 Labels Length 321533

Words Example ['人' '们' '常' '说' '生' '活' '是' '一' '部' '教' '科' '书']

Labels Example ['b' 'e' 's' 's' 'b' 'e' 's' 's' 's' 'b' 'm' 'e']

接下来我们有了这些数据就要开始制作词表了，词表制作起来无非就是输入词表和输出词表的不重复的正逆对应，制作词表的目的就是将输入的文字或标注转为 index，同时还能反向根据 index 获取对应的文字或标注，所以我们这里需要制作 word2id、id2word、tag2id、id2tag 四个字典。

为了解决 OOV 问题，我们还需要将无效字符也进行标注，这里我们统一取 0。制作时我们借助于 pandas 库的 Series 进行了去重和转换，另外还限制了每一句的最大长度，这里设置为 32，如果大于32，则截断，否则进行 padding，代码如下：

from itertools import chain

import pandas as pd

import numpy as np

# Merge all words

all\_words = list(chain(\*words))

# All words to Series

all\_words\_sr = pd.Series(all\_words)

# Get value count, index changed to set

all\_words\_counts = all\_words\_sr.value\_counts()

# Get words set

all\_words\_set = all\_words\_counts.index

# Get words ids

all\_words\_ids = range(1, len(all\_words\_set) + 1)

# Dict to transform

word2id = pd.Series(all\_words\_ids, index=all\_words\_set)

id2word = pd.Series(all\_words\_set, index=all\_words\_ids)

# Tag set and ids

tags\_set = ['x', 's', 'b', 'm', 'e']

tags\_ids = range(len(tags\_set))

# Dict to transform

tag2id = pd.Series(tags\_ids, index=tags\_set)

id2tag = pd.Series(tags\_set, index=tag2id)

max\_length = 32

def x\_transform(words):

ids = list(word2id[words])

if len(ids) >= max\_length:

ids = ids[:max\_length]

ids.extend([0] \* (max\_length - len(ids)))

return ids

def y\_transform(tags):

ids = list(tag2id[tags])

if len(ids) >= max\_length:

ids = ids[:max\_length]

ids.extend([0] \* (max\_length - len(ids)))

return ids

print('Starting transform...')

data\_x = list(map(lambda x: x\_transform(x), words))

data\_y = list(map(lambda y: y\_transform(y), labels))

data\_x = np.asarray(data\_x)

data\_y = np.asarray(data\_y)

这样我们就完成了 word2id、id2word、tag2id、id2tag 四个字典的制作，并制作好了 Numpy 数组类型的 data\_x 和 data\_y，这里 data\_x 和 data\_y 单句示例如下：

Data X Example: [8, 43, 320, 88, 36, 198, 7, 2, 41, 163, 124, 245, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data Y Example: [2, 4, 1, 1, 2, 4, 1, 1, 1, 2, 3, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

可以看到数据的 x 部分，原始文字和标注结果都转化成了词表中的 index，同时不够 32 个字符就以 0 补全。

接下来我们将其保存成 pickle 文件，以备训练和测试使用：

print('Starting pickle to file...')

with open(join(path, 'data.pkl'), 'wb') as f:

pickle.dump(data\_x, f)

pickle.dump(data\_y, f)

pickle.dump(word2id, f)

pickle.dump(id2word, f)

pickle.dump(tag2id, f)

pickle.dump(id2tag, f)

print('Pickle finished')

好，现在数据预处理部分就完成了。

构造模型

接下来我们就需要利用 pickle 文件中的数据来构建模型了，首先进行 pickle 文件的读取，然后将数据分为训练集、开发集、测试集，详细流程不再赘述，赋值为如下变量：

# Load data

data\_x, data\_y, word2id, id2word, tag2id, id2tag = load\_data()

# Split data

train\_x, train\_y, dev\_x, dev\_y, test\_x, test\_y = get\_data(data\_x, data\_y)

接下来我们使用 TensorFlow 自带的 Dataset 数据结构构造输入输出，利用 Dataset 我们可以构造一个 iterator 迭代器，每调用一次 get\_next() 方法，我们就可以得到一个 batch，这里 Dataset 的初始化我们使用 from\_tensor\_slices() 方法，然后调用其 batch() 方法来初始化每个数据集的 batch\_size，接着初始化同一个 iterator，并绑定到三个数据集上声明为三个 initializer，这样每调用 initializer，就会将 iterator 切换到对应的数据集上，代码实现如下：

# Train and dev dataset

train\_dataset = tf.data.Dataset.from\_tensor\_slices((train\_x, train\_y))

train\_dataset = train\_dataset.batch(FLAGS.train\_batch\_size)

dev\_dataset = tf.data.Dataset.from\_tensor\_slices((dev\_x, dev\_y))

dev\_dataset = dev\_dataset.batch(FLAGS.dev\_batch\_size)

test\_dataset = tf.data.Dataset.from\_tensor\_slices((test\_x, test\_y))

test\_dataset = test\_dataset.batch(FLAGS.test\_batch\_size)

# A reinitializable iterator

iterator = tf.data.Iterator.from\_structure(train\_dataset.output\_types, train\_dataset.output\_shapes)

train\_initializer = iterator.make\_initializer(train\_dataset)

dev\_initializer = iterator.make\_initializer(dev\_dataset)

test\_initializer = iterator.make\_initializer(test\_dataset)

有了 Dataset 的 iterator，我们只需要调用一次 get\_next() 方法即可得到 x 和 y\_label 了，就不需要使用 placeholder 来声明了，代码如下：

# Input Layer

with tf.variable\_scope('inputs'):

x, y\_label = iterator.get\_next()

接下来我们需要实现 embedding 层，调用 TensorFlow 的 embedding\_lookup 即可实现，这里没有使用 Pre Train 的 embedding，代码实现如下：

# Embedding Layer

with tf.variable\_scope('embedding'):

embedding = tf.Variable(tf.random\_normal([vocab\_size, FLAGS.embedding\_size]), dtype=tf.float32)

inputs = tf.nn.embedding\_lookup(embedding, x)

接下来我们就需要实现双向 LSTM 了，这里我们要构造一个 2 层的 Bi-LSTM 网络，实现的时候我们首先需要声明 LSTM Cell 的列表，然后调用 stack\_bidirectional\_rnn() 方法即可：

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)

这个方法内部是首先对每一层的 LSTM 进行正反向计算，然后对输出隐层进行 concat，然后输入下一层再进行计算，这里值得注意的地方是，我们不能把 LSTM Cell 提前组合成 MultiRNNCell 再调用 bidirectional\_dynamic\_rnn() 进行计算，这样相当于只有最后一层才进行 concat，是错误的。

现在我们得到的 output 就是 Bi-LSTM 的最后输出结果了。

接下来我们需要对输出结果进行一下 stack() 操作转化为一个 Tensor，然后将其 reshape() 一下，转化为 [-1, num\_units \* 2] 的 shape：

output = tf.stack(output, axis=1)

output = tf.reshape(output, [-1, FLAGS.num\_units \* 2])

这样我们再经过一层全连接网络将维度进行转换：

# 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

y\_predict = tf.cast(tf.argmax(y, axis=1), tf.int32)

print('Output Y', y\_predict)

这样得到的最后的 y\_predict 即为预测结果，shape 为 [batch\_size]，即每一句都得到了一个最可能的结果标注。

接下来我们需要计算一下准确率和 Loss，准确率其实就是比较 y\_predict 和 y\_label 的相似度，Loss 即为二者交叉熵：

# Reshape y\_label

y\_label\_reshape = tf.cast(tf.reshape(y\_label, [-1]), tf.int32)

# Prediction

correct\_prediction = tf.equal(y\_predict, y\_label\_reshape)

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

# Loss

cross\_entropy = tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y\_label\_reshape, logits=tf.cast(y, tf.float32)))

# Train

train = tf.train.AdamOptimizer(FLAGS.learning\_rate).minimize(cross\_entropy, global\_step=global\_step)

这里计算交叉熵使用的是 sparse\_softmax\_cross\_entropy\_with\_logits() 方法，Optimizer 使用的是 Adam。

最后指定训练过程和测试过程即可，训练过程如下：

for epoch in range(FLAGS.epoch\_num):

tf.train.global\_step(sess, global\_step\_tensor=global\_step)

# Train

sess.run(train\_initializer)

for step in range(int(train\_steps)):

smrs, loss, acc, gstep, \_ = sess.run([summaries, cross\_entropy, accuracy, global\_step, train], feed\_dict={keep\_prob: FLAGS.keep\_prob})

# Print log

if step % FLAGS.steps\_per\_print == 0:

print('Global Step', gstep, 'Step', step, 'Train Loss', loss, 'Accuracy', acc)

if epoch % FLAGS.epochs\_per\_dev == 0:

# Dev

sess.run(dev\_initializer)

for step in range(int(dev\_steps)):

if step % FLAGS.steps\_per\_print == 0:

print('Dev Accuracy', sess.run(accuracy, feed\_dict={keep\_prob: 1}), 'Step', step)

这里训练时首先调用了 train\_initializer，将 iterator 指向训练数据，这样每调用一次 get\_next()，x 和 y\_label 就会被赋值为训练数据的一个 batch，接下来打印输出了 Loss，Accuracy 等内容。另外对于开发集来说，每次进行验证的时候也需要重新调用 dev\_initializer，这样 iterator 会再次指向开发集，这样每调用一次 get\_next()，x 和 y\_label 就会被赋值为开发集的一个 batch，然后进行验证。

对于测试来说，我们可以计算其准确率，然后将测试的结果输出出来，代码实现如下：

sess.run(test\_initializer)

for step in range(int(test\_steps)):

x\_results, y\_predict\_results, acc = sess.run([x, y\_predict, accuracy], feed\_dict={keep\_prob: 1})

print('Test step', step, 'Accuracy', acc)

y\_predict\_results = np.reshape(y\_predict\_results, x\_results.shape)

for i in range(len(x\_results)):

x\_result, y\_predict\_result = list(filter(lambda x: x, x\_results[i])), list(

filter(lambda x: x, y\_predict\_results[i]))

x\_text, y\_predict\_text = ''.join(id2word[x\_result].values), ''.join(id2tag[y\_predict\_result].values)

print(x\_text, y\_predict\_text)

这里打印输出了当前测试的准确率，然后得到了测试结果，然后再结合词表将测试的真正结果打印出来即可。

运行结果

在训练过程中，我们需要构建模型图，然后调用训练部分的代码进行训练，输出结果类似如下：

Global Step 0 Step 0 Train Loss 1.67181 Accuracy 0.1475

Global Step 100 Step 100 Train Loss 0.210423 Accuracy 0.928125

Global Step 200 Step 200 Train Loss 0.208561 Accuracy 0.920625

Global Step 300 Step 300 Train Loss 0.185281 Accuracy 0.939375

Global Step 400 Step 400 Train Loss 0.186069 Accuracy 0.938125

Global Step 500 Step 500 Train Loss 0.165667 Accuracy 0.94375

Global Step 600 Step 600 Train Loss 0.201692 Accuracy 0.9275

Global Step 700 Step 700 Train Loss 0.13299 Accuracy 0.954375

...

随着训练的进行，准确率可以达到 96% 左右。

在测试阶段，输出了当前模型的准确率及真实测试输出结果，输出结果类似如下：

Test step 0 Accuracy 0.946125

据新华社北京７月９日电连日来 sbmebebmmesbes

董新辉为自己此生不能侍奉母亲而难过 bmesbebebebmmesbe

...

可见测试准确率在 95% 左右，对于测试数据，此处还输出了每句话的序列标注结果，如第一行结果中，“据”字对应的标注就是 s，代表单字成词，“新”字对应的标注是 b，代表词的起始，“华”字对应标注是 m，代表词的中间，“社”字对应的标注是 e，代表结束，这样 “据”、“新华社” 就可以被分成两个词了，可见还是有一定效果的。

**参考来源**

* [TensorFlow入门 双端 LSTM 实现序列标注](http://blog.csdn.net/jerr__y/article/details/70471066)
* [基于双向LSTM的seq2seq字标注](http://spaces.ac.cn/archives/3924/)
* [TensorFlow全新的数据读取方式：Dataset API入门教程](https://www.leiphone.com/news/201711/zV7yM5W1dFrzs8W5.html)
* [TensorFlow Importing Data](https://www.tensorflow.org/programmers_guide/datasets)