[**从头实现深度学习的对话系统--新版本tf seq2seq API构建chatbot**](http://blog.csdn.net/liuchonge/article/details/79021938)

标签： [tensorflow](http://www.csdn.net/tag/tensorflow)[seq2seq](http://www.csdn.net/tag/seq2seq)[attention](http://www.csdn.net/tag/attention)[beam\_search](http://www.csdn.net/tag/beam_search)[chatbot](http://www.csdn.net/tag/chatbot)

2018-01-10 11:43 311人阅读 [评论](http://blog.csdn.net/liuchonge/article/details/79021938#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/liuchonge/article/details/79021938#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

深度学习（52） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg nlp（41） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg RNN循环神经网络（10） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg TensorFlow（27） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg QA（23） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。 <http://blog.csdn.net/liuchonge/article/details/79021938>

目录[(?)](http://blog.csdn.net/liuchonge/article/details/79021938" \o "系统根据文章中H1到H6标签自动生成文章目录)[[+]](http://blog.csdn.net/liuchonge/article/details/79021938)

上篇文章我们使用tf.contrib.legacy\_seq2seq下的API构建了一个简单的chatbot对话系统，但是我们已经说过，这部分代码是1.0版本之前所提供的API，将来会被弃用，而且API接口并不灵活，在实际使用过程中还会存在版本不同导致的各种个样的错误。所以我们有必要学习一下新版本的API，这里先来说一下二者的不同：

* 新版本都是用dynamic\_rnn来构造RNN模型，这样就避免了数据长度不同所带来的困扰，不需要再使用model\_with\_buckets这种方法来构建模型，使得我们数据处理和模型代码都简洁很多
* 新版本将Attention、Decoder等几个主要的功能都分别进行封装，直接调用相应的Wapper函数进行封装即可，调用起来更加灵活方便，而且只需要写几个简单的函数既可以自定义的各个模块以满足我们个性化的需求。
* 实现了beam\_search功能，可直接调用

这次我们先来看如何直接使用新版本API构造对话系统，然后等下一篇文章在分析一些主要文件和函数的源码实现。本文代码可以再我的github中找到：[seq2seq\_chatbot\_new](https://github.com/lc222/seq2seq_chatbot_new" \t "_blank)。欢迎fork和star~~

数据处理

仍然沿用之前的代码，不过createBatch函数可以变得简单而又整洁，原因是新版本API我们在定义输入的placeholder是不需要在定义为seq\_len\*batch\_size这样的列表，直接定义一个batch\_size\*seq\_len的tensor即可。所以数据处理部分的代码也可以简化为得到一个嵌套列表的形式即可。这里我们重新定义Batch类，使其包含四个元素分别为encoder\_inputs、encoder\_inputs\_length、decoder\_targets、decoder\_targets\_length，前两项是PAD之后的源序列及每个序列的长度，后两项为PAD之后的目的序列和每个序列的长度。这里encoder\_inputs\_length和decoder\_targets\_length是为了动态编解码时表示序列长度的作用，下面给出修改了的Batch类和createBatch函数，其他函数都没有发生变化。

class Batch:

def \_\_init\_\_(self):

self.encoder\_inputs = [] #嵌套列表，每个元素都是一个句子中每个单词都id

self.encoder\_inputs\_length = [] #一维列表，每个元素对应上面每个句子的长度

self.decoder\_targets = []

self.decoder\_targets\_length = []

def createBatch(samples):

'''

根据给出的samples（就是一个batch的数据），进行padding并构造成placeholder所需要的数据形式

:param samples: 一个batch的样本数据，列表，每个元素都是[question， answer]的形式，id

:return: 处理完之后可以直接传入feed\_dict的数据格式

'''

batch = Batch()

#获取每个样本的长度，并保存在source\_length和target\_length中

batch.encoder\_inputs\_length = [len(sample[0]) for sample in samples]

batch.decoder\_targets\_length = [len(sample[1]) for sample in samples]

#获得一个batch样本中最大的序列长度

max\_source\_length = max(batch.encoder\_inputs\_length)

max\_target\_length = max(batch.decoder\_targets\_length)

#将每个样本进行PAD至最大长度

for sample in samples:

#将source进行反序并PAD值本batch的最大长度

source = list(reversed(sample[0]))

pad = [padToken] \* (max\_source\_length - len(source))

batch.encoder\_inputs.append(pad + source)

#将target进行PAD，并添加END符号

target = sample[1]

pad = [padToken] \* (max\_target\_length - len(target))

batch.decoder\_targets.append(target + pad)

return batch

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35

模型构建

这一部分代码主要是从tensorflow官网给出的nmt例子的代码简化而来，实现了最基本的attention和beam\_search等功能，同时有将nmt代码中繁杂的代码逻辑进行简化，将不必要的代码都清除，是的代码的易读性提高。这里参考nmt中所提到的构建train、eval、inference三个图进行模型构建，好处在于（下面部分翻译自[nmt官方文档](https://github.com/tensorflow/nmt" \l "building-training-eval-and-inference-graphs" \t "_blank)）：

* inference图往往与train和eval结构存在较大差异（没有decoder输入和目标，需要使用贪婪或者beam\_search进行decode，batch\_size也不同等等），所以往往需要单独进行构建
* eval图也会得到简化，因为其不需要进行反向传播，只需要得到一个loss和acc值
* 数据可以分别进行feed，简化数据操作
* 变量重用变得简单，因为train、eval存在一些公用变量和代码块，就不需要我们重复定义，使代码简化
* 只需要在train时不断保存模型参数，然后在eval和infer的时候restore参数即可

以上，所以我们构建了train、eval、infer三个函数来实现上面的功能。在看代码之前我们先来简单说一下新版API几个主要的模块以及相互之间的调用关系。tf.contrib.seq2seq文件夹下面主要有下面6个文件，除了loss文件和之前的sequence\_loss函数没有很大区别，这里不介绍之外，其他几个文件都会简单的说一下，这里主要介绍函数和类的功能，源码会放在下篇文章中介绍。

* decoder
* basic\_decoder
* helper
* attention\_wrapper
* beam\_search\_decoder
* loss

**BasicDecoder类和dynamic\_decode**

decoder和basic\_decoder文件可以放在一起看，decoder文件中定义了Decoder抽象类和dynamic\_decode函数，dynamic\_decode可以视为整个解码过程的入口，需要传入的参数就是Decoder的一个实例，他会动态的调用Decoder的step函数按步执行decode，可以理解为Decoder类定义了单步解码（根据输入求出输出，并将该输出当做下一时刻输入），而dynamic\_decode则会调用control\_flow\_ops.while\_loop这个函数来循环执行直到输出结束编码过程。而basic\_decoder文件定义了一个基本的Decoder类实例BasicDecoder，看一下其初始化函数：

def \_\_init\_\_(self, cell, helper, initial\_state, output\_layer=None):

需要传入的参数就是cell类型、helper类型、初始化状态（encoder的最后一个隐层状态）、输出层（输出映射层，将rnn\_size转化为vocab\_size维），需要注意的就是前面两个，下面分别介绍：

**cell类型（Attention类型）**

cell类型就是RNNCell，也就是decode阶段的神经元，可以使简单的RNN、GRU、LSTM（也可以加上dropout、并使用MultiRNNCell进行堆叠成多层），也可以是加上了Attention功能之后的RNNcell。这就引入了attention\_wrapper文件中定义的几种attention机制（BahdanauAttention、 LuongAttention、 BahdanauMonotonicAttention、 LuongMonotonicAttention）和将attention机制封装到RNNCell上面的方法AttentionWrapper。其实很简单，就跟dropoutwrapper、outputwrapper一样，我们只需要在原本RNNCell的基础上在封装一层attention即可。代码如下所示：

# 分为三步，第一步是定义attention机制，第二步是定义要是用的基础的RNNCell，第三步是使用AttentionWrapper进行封装

#定义要使用的attention机制。

attention\_mechanism = tf.contrib.seq2seq.BahdanauAttention(num\_units=self.rnn\_size, memory=encoder\_outputs, memory\_sequence\_length=encoder\_inputs\_length)

#attention\_mechanism = tf.contrib.seq2seq.LuongAttention(num\_units=self.rnn\_size, memory=encoder\_outputs, memory\_sequence\_length=encoder\_inputs\_length)

# 定义decoder阶段要是用的LSTMCell，然后为其封装attention wrapper

decoder\_cell = self.\_create\_rnn\_cell()

decoder\_cell = tf.contrib.seq2seq.AttentionWrapper(cell=decoder\_cell, attention\_mechanism=attention\_mechanism, attention\_layer\_size=self.rnn\_size, name='Attention\_Wrapper')

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**helper类型**

helper其实就是decode阶段如何根据预测结果得到下一时刻的输入，比如训练过程中应该直接使用上一时刻的真实值作为下一时刻输入，预测过程中可以使用贪婪的方法选择概率最大的那个值作为下一时刻等等。所以Helper也就可以大致分为训练时helper和预测时helper两种。官网给出了下面几种Helper类：

* “Helper”：最基本的抽象类
* “TrainingHelper”：训练过程中最常使用的Helper，下一时刻输入就是上一时刻target的真实值
* “GreedyEmbeddingHelper”：预测阶段最常使用的Helper，下一时刻输入是上一时刻概率最大的单词通过embedding之后的向量
* “SampleEmbeddingHelper”：预测时helper，继承自GreedyEmbeddingHelper，下一时刻输入是上一时刻通过某种概率分布采样而来在经过embedding之后的向量
* “CustomHelper”：最简单的helper，一般用户自定义helper时会基于此，需要用户自己定义如何根据输出得到下一时刻输入
* “ScheduledEmbeddingTrainingHelper”：训练时Helper，继承自TrainingHelper，添加了广义伯努利分布，对id的embedding向量进行sampling
* “ScheduledOutputTrainingHelper”：训练时Helper，继承自TrainingHelper，直接对输出进行采样
* “InferenceHelper”：CustomHelper的特例，只用于预测的helper，也需要用户自定义如何得到下一时刻输入

所以了解cell和helper类之后我们就可以很轻松的构建decode阶段的模型，以train阶段为例：

#分为四步，第一步是定义cell类型，第二步是定义helper类型，第三步是定义BasicDecoder类实例，第四步是调用dynamic\_decode函数进行解码

cell = \*\*\*(上面的代码)

training\_helper = tf.contrib.seq2seq.TrainingHelper(inputs=decoder\_inputs\_embedded,

sequence\_length=self.decoder\_targets\_length,

time\_major=False, name='training\_helper')

training\_decoder = tf.contrib.seq2seq.BasicDecoder(cell=decoder\_cell, helper=training\_helper,

initial\_state=decoder\_initial\_state, output\_layer=output\_layer)

decoder\_outputs, \_, \_ = tf.contrib.seq2seq.dynamic\_decode(decoder=training\_decoder, impute\_finished=True,

maximum\_iterations=self.max\_target\_sequence\_length)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

**beam search decoder类**

到这，基本上就可以构建一个完整的seq2seq模型了，但是上面的文件中还有beam\_search\_decoder.py文件没有介绍，也就是我们常用的beam\_search方法，下面也简单说一下。该文件定义了BeamSearchDecoder类，其实是一个Decoder的实例，跟BasicDecoder在一个等级上，但是二者又存在着不同，因为BasicDecoder需要指定helper参数，也就是指定decode阶段如何根据上一时刻输出获得下一时刻输入。但是BeamSearchDecoder不需要，因为其在内部实现了beam\_search的功能，也就包含了helper的效果，不需要再额外定义。所以BeamSearchDecoder的调用方法如下所示：

#分为三步，第一步是定义cell，第二步是定义BeamSearchDecoder，第三步是调用dynamic\_decode函数进行解码

cell = \*\*\*(上面代码)

inference\_decoder = tf.contrib.seq2seq.BeamSearchDecoder(cell=decoder\_cell, embedding=embedding,

start\_tokens=start\_tokens, end\_token=end\_token,

initial\_state=decoder\_initial\_state,

beam\_width=self.beam\_size,

output\_layer=output\_layer)

decoder\_outputs, \_, \_ = tf.contrib.seq2seq.dynamic\_decode(decoder=inference\_decoder,

maximum\_iterations=self.max\_target\_sequence\_length)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

OK，接下来切入正题，看一下model部分代码：

class Seq2SeqModel():

def \_\_init\_\_(self, rnn\_size, num\_layers, embedding\_size, learning\_rate, word\_to\_idx, mode, use\_attention,

beam\_search, beam\_size, max\_gradient\_norm=5.0):

self.learing\_rate = learning\_rate

self.embedding\_size = embedding\_size

self.rnn\_size = rnn\_size

self.num\_layers = num\_layers

self.word\_to\_idx = word\_to\_idx

self.vocab\_size = len(self.word\_to\_idx)

self.mode = mode

self.use\_attention = use\_attention

self.beam\_search = beam\_search

self.beam\_size = beam\_size

self.max\_gradient\_norm = max\_gradient\_norm

#执行模型构建部分的代码

self.build\_model()

def \_create\_rnn\_cell(self, single=False):

def single\_rnn\_cell():

# 创建单个cell，这里需要注意的是一定要使用一个single\_rnn\_cell的函数，不然直接把cell放在MultiRNNCell

# 的列表中最终模型会发生错误

single\_cell = tf.contrib.rnn.LSTMCell(self.rnn\_size)

#添加dropout

cell = tf.contrib.rnn.DropoutWrapper(single\_cell, output\_keep\_prob=self.keep\_prob\_placeholder)

return cell

#列表中每个元素都是调用single\_rnn\_cell函数

cell = tf.contrib.rnn.MultiRNNCell([single\_rnn\_cell() for \_ in range(self.num\_layers)])

return cell

def build\_model(self):

print('building model... ...')

#=================================1, 定义模型的placeholder

self.encoder\_inputs = tf.placeholder(tf.int32, [None, None], name='encoder\_inputs')

self.encoder\_inputs\_length = tf.placeholder(tf.int32, [None], name='encoder\_inputs\_length')

self.batch\_size = tf.placeholder(tf.int32, [], name='batch\_size')

self.keep\_prob\_placeholder = tf.placeholder(tf.float32, name='keep\_prob\_placeholder')

self.decoder\_targets = tf.placeholder(tf.int32, [None, None], name='decoder\_targets')

self.decoder\_targets\_length = tf.placeholder(tf.int32, [None], name='decoder\_targets\_length')

# 根据目标序列长度，选出其中最大值，然后使用该值构建序列长度的mask标志。用一个sequence\_mask的例子来说明起作用

# tf.sequence\_mask([1, 3, 2], 5)

# [[True, False, False, False, False],

# [True, True, True, False, False],

# [True, True, False, False, False]]

self.max\_target\_sequence\_length = tf.reduce\_max(self.decoder\_targets\_length, name='max\_target\_len')

self.mask = tf.sequence\_mask(self.decoder\_targets\_length, self.max\_target\_sequence\_length, dtype=tf.float32, name='masks')

#=================================2, 定义模型的encoder部分

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

#创建LSTMCell，两层+dropout

encoder\_cell = self.\_create\_rnn\_cell()

#构建embedding矩阵,encoder和decoder公用该词向量矩阵

embedding = tf.get\_variable('embedding', [self.vocab\_size, self.embedding\_size])

encoder\_inputs\_embedded = tf.nn.embedding\_lookup(embedding, self.encoder\_inputs)

# 使用dynamic\_rnn构建LSTM模型，将输入编码成隐层向量。

# encoder\_outputs用于attention，batch\_size\*encoder\_inputs\_length\*rnn\_size,

# encoder\_state用于decoder的初始化状态，batch\_size\*rnn\_szie

encoder\_outputs, encoder\_state = tf.nn.dynamic\_rnn(encoder\_cell, encoder\_inputs\_embedded,

sequence\_length=self.encoder\_inputs\_length,

dtype=tf.float32)

# =================================3, 定义模型的decoder部分

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

encoder\_inputs\_length = self.encoder\_inputs\_length

if self.beam\_search:

# 如果使用beam\_search，则需要将encoder的输出进行tile\_batch，其实就是复制beam\_size份。

print("use beamsearch decoding..")

encoder\_outputs = tf.contrib.seq2seq.tile\_batch(encoder\_outputs, multiplier=self.beam\_size)

encoder\_state = nest.map\_structure(lambda s: tf.contrib.seq2seq.tile\_batch(s, self.beam\_size), encoder\_state)

encoder\_inputs\_length = tf.contrib.seq2seq.tile\_batch(self.encoder\_inputs\_length, multiplier=self.beam\_size)

#定义要使用的attention机制。

attention\_mechanism = tf.contrib.seq2seq.BahdanauAttention(num\_units=self.rnn\_size, memory=encoder\_outputs,

memory\_sequence\_length=encoder\_inputs\_length)

#attention\_mechanism = tf.contrib.seq2seq.LuongAttention(num\_units=self.rnn\_size, memory=encoder\_outputs, memory\_sequence\_length=encoder\_inputs\_length)

# 定义decoder阶段要是用的LSTMCell，然后为其封装attention wrapper

decoder\_cell = self.\_create\_rnn\_cell()

decoder\_cell = tf.contrib.seq2seq.AttentionWrapper(cell=decoder\_cell, attention\_mechanism=attention\_mechanism,

attention\_layer\_size=self.rnn\_size, name='Attention\_Wrapper')

#如果使用beam\_seach则batch\_size = self.batch\_size \* self.beam\_size。因为之前已经复制过一次

batch\_size = self.batch\_size if not self.beam\_search else self.batch\_size \* self.beam\_size

#定义decoder阶段的初始化状态，直接使用encoder阶段的最后一个隐层状态进行赋值

decoder\_initial\_state = decoder\_cell.zero\_state(batch\_size=batch\_size, dtype=tf.float32).clone(cell\_state=encoder\_state)

output\_layer = tf.layers.Dense(self.vocab\_size, kernel\_initializer=tf.truncated\_normal\_initializer(mean=0.0, stddev=0.1))

if self.mode == 'train':

# 定义decoder阶段的输入，其实就是在decoder的target开始处添加一个<go>,并删除结尾处的<end>,并进行embedding。

# decoder\_inputs\_embedded的shape为[batch\_size, decoder\_targets\_length, embedding\_size]

ending = tf.strided\_slice(self.decoder\_targets, [0, 0], [self.batch\_size, -1], [1, 1])

decoder\_input = tf.concat([tf.fill([self.batch\_size, 1], self.word\_to\_idx['<go>']), ending], 1)

decoder\_inputs\_embedded = tf.nn.embedding\_lookup(embedding, decoder\_input)

#训练阶段，使用TrainingHelper+BasicDecoder的组合，这一般是固定的，当然也可以自己定义Helper类，实现自己的功能

training\_helper = tf.contrib.seq2seq.TrainingHelper(inputs=decoder\_inputs\_embedded,

sequence\_length=self.decoder\_targets\_length,

time\_major=False, name='training\_helper')

training\_decoder = tf.contrib.seq2seq.BasicDecoder(cell=decoder\_cell, helper=training\_helper,

initial\_state=decoder\_initial\_state, output\_layer=output\_layer)

#调用dynamic\_decode进行解码，decoder\_outputs是一个namedtuple，里面包含两项(rnn\_outputs, sample\_id)

# rnn\_output: [batch\_size, decoder\_targets\_length, vocab\_size]，保存decode每个时刻每个单词的概率，可以用来计算loss

# sample\_id: [batch\_size], tf.int32，保存最终的编码结果。可以表示最后的答案

decoder\_outputs, \_, \_ = tf.contrib.seq2seq.dynamic\_decode(decoder=training\_decoder,

impute\_finished=True,

maximum\_iterations=self.max\_target\_sequence\_length)

# 根据输出计算loss和梯度，并定义进行更新的AdamOptimizer和train\_op

self.decoder\_logits\_train = tf.identity(decoder\_outputs.rnn\_output)

self.decoder\_predict\_train = tf.argmax(self.decoder\_logits\_train, axis=-1, name='decoder\_pred\_train')

# 使用sequence\_loss计算loss，这里需要传入之前定义的mask标志

self.loss = tf.contrib.seq2seq.sequence\_loss(logits=self.decoder\_logits\_train,

targets=self.decoder\_targets, weights=self.mask)

# Training summary for the current batch\_loss

tf.summary.scalar('loss', self.loss)

self.summary\_op = tf.summary.merge\_all()

optimizer = tf.train.AdamOptimizer(self.learing\_rate)

trainable\_params = tf.trainable\_variables()

gradients = tf.gradients(self.loss, trainable\_params)

clip\_gradients, \_ = tf.clip\_by\_global\_norm(gradients, self.max\_gradient\_norm)

self.train\_op = optimizer.apply\_gradients(zip(clip\_gradients, trainable\_params))

elif self.mode == 'decode':

start\_tokens = tf.ones([self.batch\_size, ], tf.int32) \* self.word\_to\_idx['<go>']

end\_token = self.word\_to\_idx['<eos>']

# decoder阶段根据是否使用beam\_search决定不同的组合，

# 如果使用则直接调用BeamSearchDecoder（里面已经实现了helper类）

# 如果不使用则调用GreedyEmbeddingHelper+BasicDecoder的组合进行贪婪式解码

if self.beam\_search:

inference\_decoder = tf.contrib.seq2seq.BeamSearchDecoder(cell=decoder\_cell, embedding=embedding,

start\_tokens=start\_tokens, end\_token=end\_token,

initial\_state=decoder\_initial\_state,

beam\_width=self.beam\_size,

output\_layer=output\_layer)

else:

decoding\_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(embedding=embedding,

start\_tokens=start\_tokens, end\_token=end\_token)

inference\_decoder = tf.contrib.seq2seq.BasicDecoder(cell=decoder\_cell, helper=decoding\_helper,

initial\_state=decoder\_initial\_state,

output\_layer=output\_layer)

decoder\_outputs, \_, \_ = tf.contrib.seq2seq.dynamic\_decode(decoder=inference\_decoder,

maximum\_iterations=10)

# 调用dynamic\_decode进行解码，decoder\_outputs是一个namedtuple，

# 对于不使用beam\_search的时候，它里面包含两项(rnn\_outputs, sample\_id)

# rnn\_output: [batch\_size, decoder\_targets\_length, vocab\_size]

# sample\_id: [batch\_size, decoder\_targets\_length], tf.int32

# 对于使用beam\_search的时候，它里面包含两项(predicted\_ids, beam\_search\_decoder\_output)

# predicted\_ids: [batch\_size, decoder\_targets\_length, beam\_size],保存输出结果

# beam\_search\_decoder\_output: BeamSearchDecoderOutput instance namedtuple(scores, predicted\_ids, parent\_ids)

# 所以对应只需要返回predicted\_ids或者sample\_id即可翻译成最终的结果

if self.beam\_search:

self.decoder\_predict\_decode = decoder\_outputs.predicted\_ids

else:

self.decoder\_predict\_decode = tf.expand\_dims(decoder\_outputs.sample\_id, -1)

# =================================4, 保存模型

self.saver = tf.train.Saver(tf.all\_variables())

def train(self, sess, batch):

#对于训练阶段，需要执行self.train\_op, self.loss, self.summary\_op三个op，并传入相应的数据

feed\_dict = {self.encoder\_inputs: batch.encoder\_inputs,

self.encoder\_inputs\_length: batch.encoder\_inputs\_length,

self.decoder\_targets: batch.decoder\_targets,

self.decoder\_targets\_length: batch.decoder\_targets\_length,

self.keep\_prob\_placeholder: 0.5,

self.batch\_size: len(batch.encoder\_inputs)}

\_, loss, summary = sess.run([self.train\_op, self.loss, self.summary\_op], feed\_dict=feed\_dict)

return loss, summary

def eval(self, sess, batch):

# 对于eval阶段，不需要反向传播，所以只执行self.loss, self.summary\_op两个op，并传入相应的数据

feed\_dict = {self.encoder\_inputs: batch.encoder\_inputs,

self.encoder\_inputs\_length: batch.encoder\_inputs\_length,

self.decoder\_targets: batch.decoder\_targets,

self.decoder\_targets\_length: batch.decoder\_targets\_length,

self.keep\_prob\_placeholder: 1.0,

self.batch\_size: len(batch.encoder\_inputs)}

loss, summary = sess.run([self.loss, self.summary\_op], feed\_dict=feed\_dict)

return loss, summary

def infer(self, sess, batch):

#infer阶段只需要运行最后的结果，不需要计算loss，所以feed\_dict只需要传入encoder\_input相应的数据即可

feed\_dict = {self.encoder\_inputs: batch.encoder\_inputs,

self.encoder\_inputs\_length: batch.encoder\_inputs\_length,

self.keep\_prob\_placeholder: 1.0,

self.batch\_size: len(batch.encoder\_inputs)}

predict = sess.run([self.decoder\_predict\_decode], feed\_dict=feed\_dict)

return predict

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186

训练 && 预测

模型构建好之后，剩下的工作就很简单了，训练的话其实就是一个简单的循环，每个epoch都重新shuffle数据并产生batches的数据并传入模型调用train函数进行训练。时不时打印结果并保存模型参数，这里如果大家有eval数据集可以添加上相应的代码，比如每个100步评价一次等~~

预测的话跟训练步骤是一样的，先倒入模型参数，再将输入的句子转化成batch，接下来调用infer函数即可。这里主要说一下如何从infer函数返回结果predicted\_ids得到我们想要的字符串结果。首先来讲predicted\_ids是一个batch\_size\*decode\_length\*beam\_size维度的列表，我这里每次只预测一个结果所以batch\_size为1。我们最终想要beam\_szie个长度为decode\_length的字符串（如果字符串中有，则长度会变短），所以需要对predicted\_ids进行转化并将id转换为其对应的word。使用下面这个函数即可：

def predict\_ids\_to\_seq(predict\_ids, id2word, beam\_szie):

'''

将beam\_search返回的结果转化为字符串

:param predict\_ids: 列表，长度为batch\_size，每个元素都是decode\_len\*beam\_size的数组

:param id2word: vocab字典

:return:

'''

for single\_predict in predict\_ids:

for i in range(beam\_szie):

predict\_list = np.ndarray.tolist(single\_predict[:, :, i])

predict\_seq = []

for idx in predict\_list[0]:

predict\_seq.append(id2word[idx])

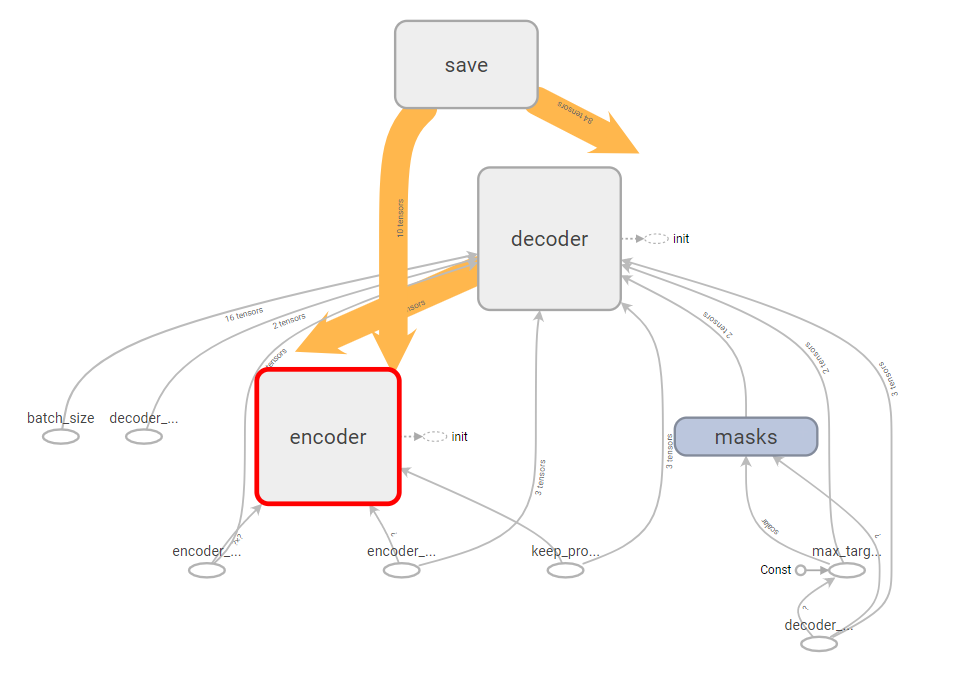
print(predict\_seq)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

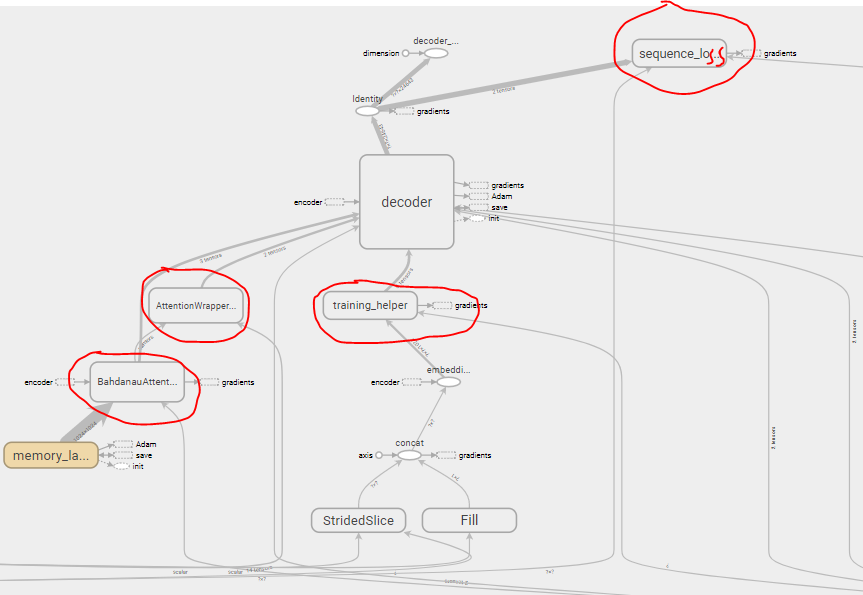
结果

训练过程中发现，相比老版本的API而言，训练速度变慢了很多，大概降低了4,5倍左右，具体的原因还没有搞清楚，不知道大家有没有遇到这种情况，求指教~~导致我的模型现在还没训练结束==不过也有一些简单的结果可以看看：

先看看模型的结构图：



decoder的内部细节：



训练的loss曲线：



回答，现在模型训练的时间还太短，貌似还没有学习到什么有营养的回答，机器太慢也是没有办法，这里beam\_size取5，如下图所示，等过几天模型训练好了再来更新吧：

