[**解读tensorflow之rnn 的示例 ptb\_word\_lm.py**](http://www.cnblogs.com/welhzh/p/6739370.html)

这两天想搞清楚用tensorflow来实现rnn/lstm如何做，但是google了半天，发现tf在rnn方面的实现代码或者教程都太少了，仅有的几个教程讲的又过于简单。没办法，只能亲自动手一步步研究官方给出的代码了。

本文研究的代码主体来自官方源码[ptb-word-lm](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/models/rnn/ptb/ptb_word_lm.py)。但是，如果你直接运行这个代码，可以看到warning：

WARNING:tensorflow:: Using a concatenated state is slower and will soon be deprecated. Use state\_is\_tuple=True.

于是根据这个warning，找到了一个相关的issue：<https://github.com/tensorflow/tensorflow/issues/2695>  
回答中有人给出了对应的[修改](https://github.com/jihunchoi/tensorflow/blob/ptb_use_state_tuple/tensorflow/models/rnn/ptb/ptb_word_lm.py)，加入了state\_is\_tuple=True，笔者就是基于这段代码学习的。

**代码结构**

tf的代码看多了之后就知道其实官方代码的这个结构并不好：

1. graph的构建和训练部分放在了一个文件中，至少也应该分开成model.py和train.py两个文件，model.py中只有一个PTBModel类
2. graph的构建部分全部放在了PTBModel类的constructor中

恰好看到了一篇专门讲如何构建tensorflow模型代码的[blog](http://danijar.com/structuring-your-tensorflow-models/)，值得学习，来重构自己的代码吧。

**值得学习的地方**

虽说官方给出的代码结构上有点小缺陷，但是毕竟都是大神们写出来的，值得我们学习的地方很多，来总结一下：

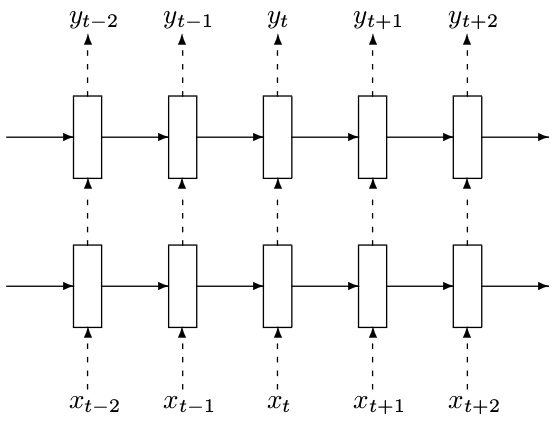
**(1)** 设置is\_training这个标志  
这个很有必要，因为training阶段和valid/test阶段参数设置上会有小小的区别，比如test时不进行dropout  
**(2)** 将必要的各类参数都写在config类中独立管理  
这个的好处就是各类参数的配置工作和model类解耦了，不需要将大量的参数设置写在model中，那样可读性不仅差，还不容易看清究竟设置了哪些超参数

**placeholder**

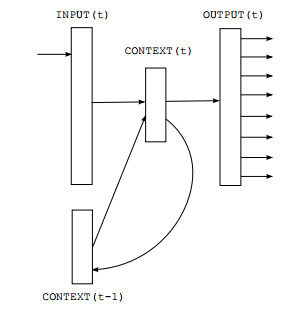
两个，分别命名为self.\_input\_data和self.\_target，只是注意一下，由于我们现在要训练的模型是language model，也就是给一个word，预测最有可能的下一个word，因此可以看出来，input和output是同型的。并且，placeholder只 存储一个batch的data，input接收的是个word在vocabulary中对应的index【后续会将index转成dense embedding】,每次接收一个seq长度的words，那么，input shape=[batch\_size, num\_steps]

**定义cell**

在很多用到rnn的paper中我们会看到类似的图：



这其中的每个小长方形就表示一个cell。每个cell中又是一个略复杂的结构，如下图：



图中的context就是一个cell结构，可以看到它接受的输入有input(t)，context(t- 1)，然后输出output(t)，比如像我们这个任务中，用到多层堆叠的rnn cell的话，也就是当前层的cell的output还要作为下一层cell的输入，因此可推出每个cell的输入和输出的shape是一样。如果输入的 shape=(None, n)，加上context(t-1)同时作为输入部分，因此可以知道**W**的shape=(2n, n)。

说了这么多，其实我只是想表达一个重点，就是

**别小看那一个小小的cell，它并不是只有1个neuron unit，而是n个hidden units**

因此，我们注意到tensorflow中定义一个cell（BasicRNNCell/BasicLSTMCell/GRUCell/RNNCell/LSTMCell）结构的时候需要提供的一个参数就是hidden\_units\_size。

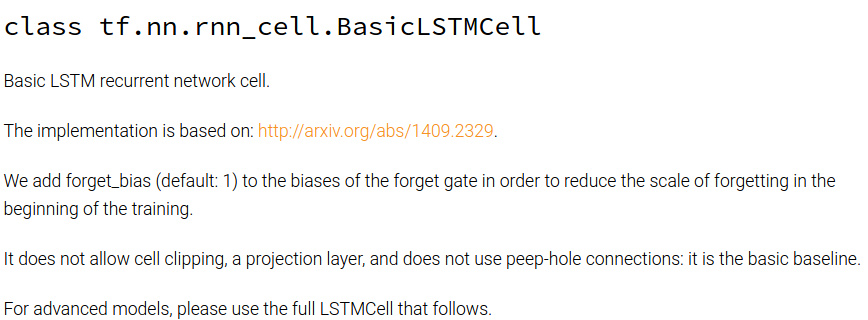
弄明白这个之后，再看tensorflow中定义cell的代码就无比简单了：

|  |  |
| --- | --- |
| 1  2  3  4  5 | lstm\_cell = tf.nn.rnn\_cell.BasicLSTMCell(size, forget\_bias=0.0, state\_is\_tuple=True)  if is\_training and config.keep\_prob < 1:  lstm\_cell = tf.nn.rnn\_cell.DropoutWrapper(  lstm\_cell, output\_keep\_prob=config.keep\_prob)  cell = tf.nn.rnn\_cell.MultiRNNCell([lstm\_cell] \* config.num\_layers, state\_is\_tuple=True) |

首先，定义一个最小的cell单元，也就是小长方形，BasicLSTMCell。

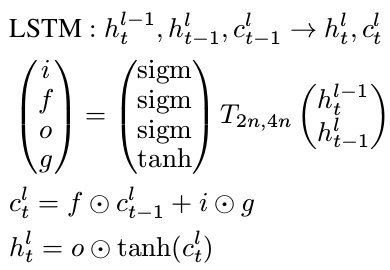
**问题1：为什么是BasicLSTMCell**

你肯定会问，这个类和LSTMCell有什么区别呢？good question，文档给出的解释是这样的：



划一下重点就是倒数第二句话，意思是说这个类没有实现clipping，projection layer，peep-hole等一些lstm的高级变种，仅作为一个基本的basicline结构存在，如果要使用这些高级variant要用LSTMCell这个类。  
因为我们现在只是想搭建一个基本的lstm-language model模型，能够训练出一定的结果就行了，因此现阶段BasicLSTMCell够用。这就是为什么这里用的是BasicLSTMCell这个类而不是别的什么。

**问题2：state\_is\_tuple=True是什么**



(此图偷自[recurrent neural network regularization](http://arxiv.org/pdf/1409.2329.pdf))  
可以看到，每个lstm cell在t时刻都会产生两个内部状态Ct和ht，都是在t-1时刻计算要用到的。这两个状态在tensorflow中都要记录，记住这个就好理解了。

来看官方对这个的解释：

http://images2015.cnblogs.com/blog/604152/201704/604152-20170420155849602-1090326120.jpg

意思是说，如果state\_is\_tuple=True，那么上面我们讲到的状态Ct和ht就是分开记录，放在一个tuple中，如果这个参数没有设定或设置成False，两个状态就按 列连接起来，成为[batch, 2n](n是hidden units个数)返回。官方说这种形式马上就要被deprecated了，所有我们在使用LSTM的时候要加上state\_is\_tuple=True

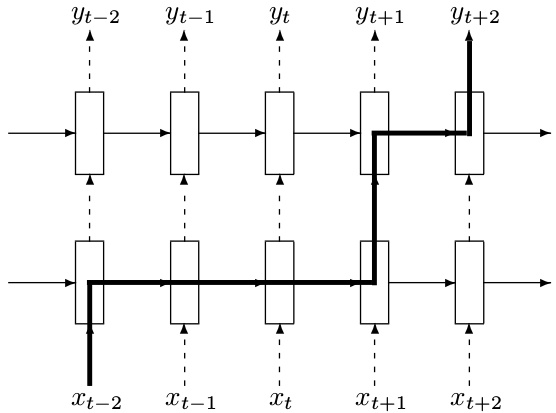
**问题3：forget\_bias是什么**

暂时还没管这个参数的含义

**DropoutWrapper**

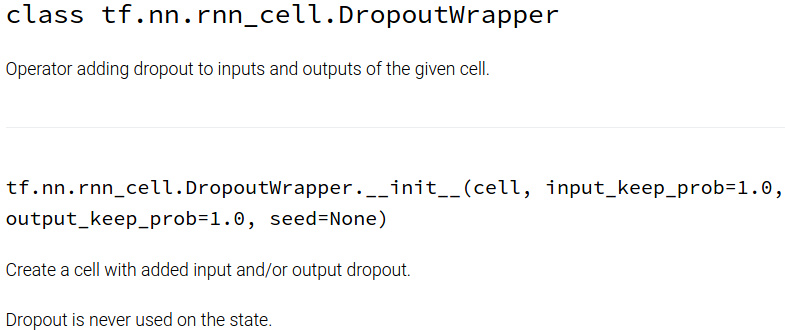
dropout是一种非常efficient的regularization方法，在rnn中如何使用dropout和cnn不同，推荐大家去把[recurrent neural network regularization](http://arxiv.org/pdf/1409.2329.pdf)看一遍。我在这里仅讲结论:

　　对于rnn的部分不进行dropout，也就是说从t-1时候的状态传递到t时刻进行计算时，这个中间不进行memory的dropout；仅在同一个t时刻中，多层cell之间传递信息的时候进行dropout。



上图中，xt-2时刻的输入首先传入第一层cell，这个过程有dropout，但是从 t-2 时刻的第一层cell传到 t-1, t, t+1的第一层cell这个中间都不进行dropout。再从t+1时候的第一层cell向同一时刻内后续的cell传递时，这之间又有dropout了。

因此，我们在代码中定义完cell之后，在cell外部包裹上dropout，这个类叫DropoutWrapper，这样我们的cell就有了dropout功能！



可以从官方文档中看到，它有input\_keep\_prob和output\_keep\_prob，也就是说裹上这个DropoutWrapper之 后，如果我希望是input传入这个cell时dropout掉一部分input信息的话，就设置input\_keep\_prob，那么传入到cell的 就是部分input；如果我希望这个cell的output只部分作为下一层cell的input的话，就定义output\_keep\_prob。不要太 方便。  
根据Zaremba在paper中的描述，这里应该给cell设置output\_keep\_prob。

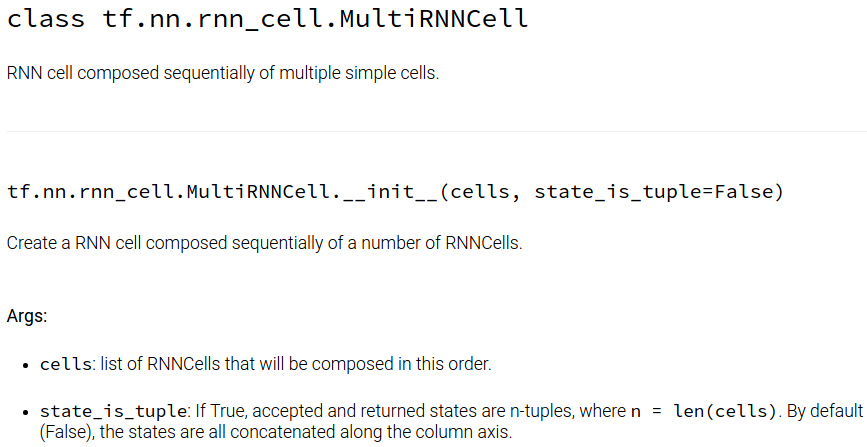
|  |  |
| --- | --- |
| 1  2  3 | if is\_training and config.keep\_prob < 1:  lstm\_cell = tf.nn.rnn\_cell.DropoutWrapper(  lstm\_cell, output\_keep\_prob=config.keep\_prob) |

**Stack MultiCell**

现在我们定义了一个lstm cell，这个cell仅是整个图中的一个小长方形，我们希望整个网络能更deep的话，应该stack多个这样的lstm cell，tensorflow给我们提供了MultiRNNCell（注意：multi只有这一个类，并没有MultiLSTMCell之类的），因此 堆叠多层只生成这个类即可。

|  |  |
| --- | --- |
| 1 | cell = tf.nn.rnn\_cell.MultiRNNCell([lstm\_cell] \* config.num\_layers, state\_is\_tuple=True) |

我们还是看看官方文档:



我们可以从描述中看出，tensorflow并不是简单的堆叠了多个single cell，而是将这些cell stack之后当成了一个完整的独立的cell，每个小cell的中间状态还是保存下来了，按n\_tuple存储，但是输出output只用最后那个 cell的输出。

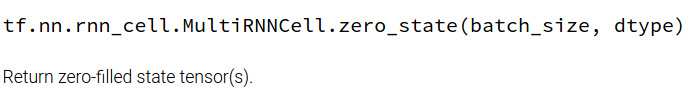
这样，我们就定义好了每个t时刻的整体cell，接下来只要每个时刻传入不同的输入，再在时间上展开，就能得到上图多个时间上unroll graph。

**initial states**

接下来就需要给我们的multi lstm cell进行状态初始化。怎么做呢？Zaremba已经告诉我们了

We initialize the hidden states to zero. We then use the  
final hidden states of the current minibatch as the initial hidden state of the subsequent minibatch  
(successive minibatches sequentially traverse the training set).

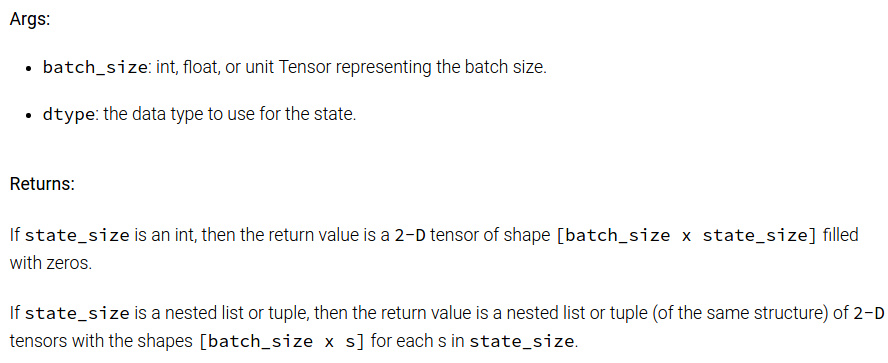
也就是初始时全部赋值为0状态。



那么就需要有一个self.\_initial\_state来保存我们生成的全0状态，最后直接调用MultiRNNCell的zero\_state()方法即可。

|  |  |
| --- | --- |
| 1 | self.\_initial\_state = cell.zero\_state(batch\_size, tf.float32) |

注意：这里传入的是batch\_size，我一开始没看懂为什么，那就看文档的解释吧:



state\_size是我们在定义MultiRNNCell的时就设置好了的，只是我们的输入input shape=[batch\_size, num\_steps]，我们刚刚定义好的cell会依次接收num\_steps个输入然后产生最后的state（n-tuple，n表示堆叠的层数）但是 一个batch内有batch\_size这样多个seq，因此就需要[batch\_size，s]来存储整个batch每个seq的状态。

**embedding input**

我们预处理了数据之后得到的是一个二维array，每个位置的元素表示这个word在vocabulary中的index。  
但是传入graph的数据不能讲word用index来表示，这样词和词之间的关系就没法刻画了。我们需要将word用dense vector表示，这也就是广为人知的word embedding。  
paper中并没有使用预训练的word embedding，所有的embedding都是随机初始化，然后在训练过程中不断更新embedding矩阵的值。

|  |  |
| --- | --- |
| 1  2  3 | with tf.device("/cpu:0"):  embedding = tf.get\_variable("embedding", [vocab\_size, size])  inputs = tf.nn.embedding\_lookup(embedding, self.\_input\_data) |

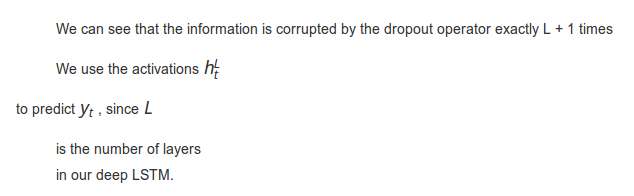
首先要明确几点：

1. 既然我们要在训练过程中不断更新embedding矩阵，那么embedding必须是tf.Variable并且trainable=True（default）
2. 目前tensorflow对于lookup embedding的操作只能再cpu上进行
3. embedding 矩阵的大小是多少：每个word都需要有对应的embedding vector，总共就是vocab\_size那么多个embedding，每个word embed成多少维的vector呢？因为我们input embedding后的结果就直接输入给了第一层cell，刚才我们知道cell的hidden units size，因此这个embedding dim要和hidden units size对应上（这也才能和内部的各种门的W和b完美相乘）。因此，我们就确定下来embedding matrix shape=**[vocab\_size, hidden\_units\_size]**

最后生成真正的inputs节点，也就是从embedding\_lookup之后得到的结果，这个tensor的shape=[batch\_size, num\_stemps, size](http://lan2720.github.io/2016/07/16/%E8%A7%A3%E8%AF%BBtensorflow%E4%B9%8Brnn/size%E8%A1%A8%E7%A4%BAhidden%20units%20size)

**input data dropout**

刚才我们定义了每个cell的输出要wrap一个dropout，但是根据paper中讲到的，



cell的层数一共定义了L层，为什么dropout要进行L+1次呢？就是因为输入这个地方要进行1次dropout。比如，我们设置cell的 hidden units size=200的话，input embbeding dim=200维度较高，dropout一部分，防止overfitting。

|  |  |
| --- | --- |
| 1  2 | if is\_training and config.keep\_prob < 1:  inputs = tf.nn.dropout(inputs, config.keep\_prob) |

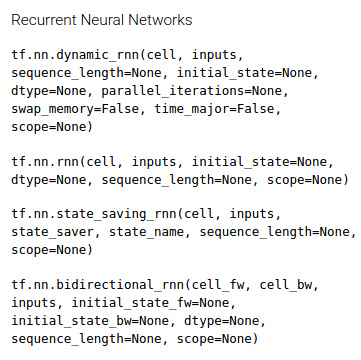
和上面的DropoutWrapper一样，都是在is\_training and config.keep\_prob < 1的条件下才进行dropout。  
由于这个仅对tensor进行dropout（而非rnn\_cell进行wrap），因此调用的是tf.nn.dropout。

**RNN循环起来！**

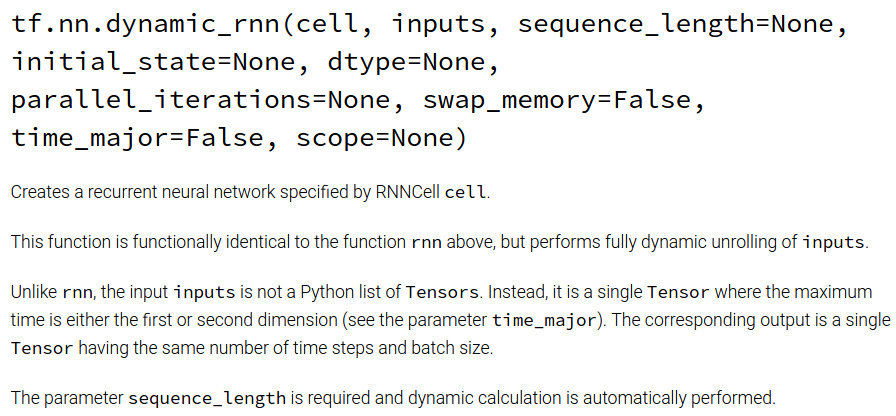
到上面这一步，我们的基本单元multi cell和inputs算是全部准备好啦，接下来就是在time上进行recurrent，得到num\_steps每一时刻的output和states。  
那么很自然的我们可以猜测output的shape=[batch\_size, num\_steps, size]，states的shape=[batch\_size, n(LSTMStateTuple)]【state是整个seq输入完之后得到的每层的state

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | outputs = []  state = self.\_initial\_state  with tf.variable\_scope("RNN"):  for time\_step in range(num\_steps):  if time\_step > 0: tf.get\_variable\_scope().reuse\_variables()  (cell\_output, state) = cell(inputs[:, time\_step, :], state)  outputs.append(cell\_output) |

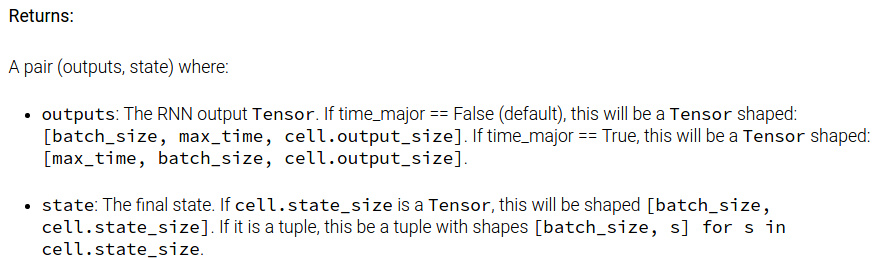
以上这是官方给出的代码，个人觉得不是太好。怎么办，查文档。



可以看到，有四个函数可以用来构建rnn，我们一个个的讲。  
（1） dynamic rnn



这个方法给rnn()很类似，只是它的inputs不是list of tensors，而是一整个tensor，num\_steps是inputs的一个维度。这个方法的输出是一个pair，



由于我们preprocessing之后得到的input shape=[batch\_size, num\_steps, size]因此，time\_major=False。  
最后的到的这个pair的shape正如我们猜测的输出是一样的。

sequence\_length: (optional) An int32/int64 vector sized [batch\_size].表示的是batch中每行sequence的长度。

调用方法是：

|  |  |
| --- | --- |
| 1 | outputs, state = tf.nn.dynamic\_rnn(cell, inputs, sequence\_length=..., initial\_state=state) |

state是final state，如果有n layer，则是final state也有n个元素，对应每一层的state。

（2）tf.nn.rnn  
这个函数和dynamic\_rnn的区别就在于，这个需要的inputs是a list of tensor，这个list的长度是num\_steps，也就是将每一个时刻的输入切分出来了，tensor的shape=[batch\_size, input\_size]【这里的input每一个都是word embedding，因此input\_size=hidden\_units\_size】

除了输出inputs是list之外，输出稍有差别。



可以看到，输出也是一个长度为T(num\_steps)的list，每一个output对应一个t时刻的input(batch\_size, hidden\_units\_size)，output shape=[batch\_size, hidden\_units\_size]

（3）state\_saving\_rnn  
这个方法可以接收一个state saver对象，这是和以上两个方法不同之处，另外其inputs和outputs也都是list of tensors。

（4）bidirectional\_rnn  
等研究bi-rnn网络的时候再讲。

以上介绍了四种rnn的构建方式，这里选择dynamic\_rnn.因为inputs中的第2个维度已经是num\_steps了。

**得到output之后传到下一层softmax layer**

既然我们用的是dynamic\_rnn，那么outputs shape=[batch\_size, num\_steps, size]，而接下来需要将output传入到softmax层，softmax层并没有显式地使用tf.nn.softmax函数，而是只是计算了 wx+b得到logits（实际上是一样的，softmax函数仅仅只是将logits再rescale到0-1之间）

**计算loss**

得到logits后，用到了nn.seq2seq.sequence\_loss\_by\_example函数来计算“所谓的softmax层”的 loss。这个loss是整个batch上累加的loss，需要除上batch\_size，得到平均下来的loss，也就是self.\_cost。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | loss = tf.nn.seq2seq.sequence\_loss\_by\_example(  [logits],  [tf.reshape(self.\_targets, [-1])],  [tf.ones([batch\_size \* num\_steps])])  self.\_cost = cost = tf.reduce\_sum(loss) / batch\_size  self.\_final\_state = state |

**求导，定义train\_op**

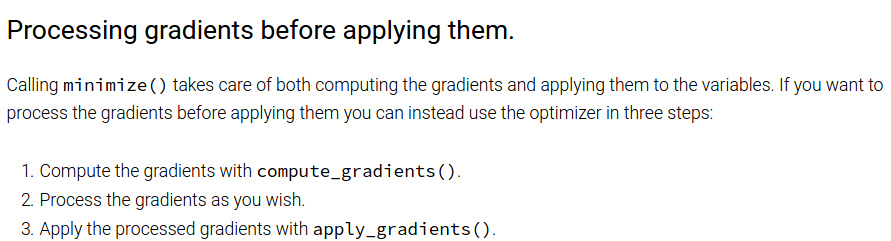
如果is\_training=False，也就是仅valid or test的话，计算出loss这一步也就终止了。之所以要求导，就是train的过程。所以这个地方对is\_training进行一个判断。

|  |  |
| --- | --- |
| 1  2 | if not is\_training:  return |

如果想在训练过程中调节learning rate的话，生成一个lr的variable，但是trainable=False，也就是不进行求导。

|  |  |
| --- | --- |
| 1 | self.\_lr = tf.Variable(0.0, trainable=False) |

gradient在backpropagate过程中，很容易出现vanish&explode现象，尤其是rnn这种back很多个time step的结构。  
因此都要使用clip来对gradient值进行调节。  
既然要调节了就不能简单的调用optimizer.minimize(loss)，而是需要显式的计算gradients，然后进行clip，将clip后的gradient进行apply。  
官方文档说明了这种操作：



并给出了一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | # Create an optimizer.  opt = GradientDescentOptimizer(learning\_rate=0.1)  # Compute the gradients for a list of variables.  grads\_and\_vars = opt.compute\_gradients(loss, <list of variables>)  # grads\_and\_vars is a list of tuples (gradient, variable). Do whatever you  # need to the 'gradient' part, for example cap them, etc.  capped\_grads\_and\_vars = [(MyCapper(gv[0]), gv[1]) for gv in grads\_and\_vars]  # Ask the optimizer to apply the capped gradients.  opt.apply\_gradients(capped\_grads\_and\_vars) |

模仿这个代码，我们可以写出如下的伪代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | optimizer = tf.train.AdamOptimizer(learning\_rate=self.\_lr)  # gradients: return A list of sum(dy/dx) for each x in xs.  grads = optimizer.gradients(self.\_cost, <list of variables>)  clipped\_grads = tf.clip\_by\_global\_norm(grads, config.max\_grad\_norm)  # accept: List of (gradient, variable) pairs, so zip() is needed  self.\_train\_op = optimizer.apply\_gradients(zip(grads, <list of variables>)) |

可以看到，此时就差一个<list of variables>不知道了，也就是需要对哪些variables进行求导。  
答案是：trainable variables  
因此，我们得到

|  |  |
| --- | --- |
| 1 | tvars = tf.trainable\_variables() |

用tvars带入上面的代码中即可。

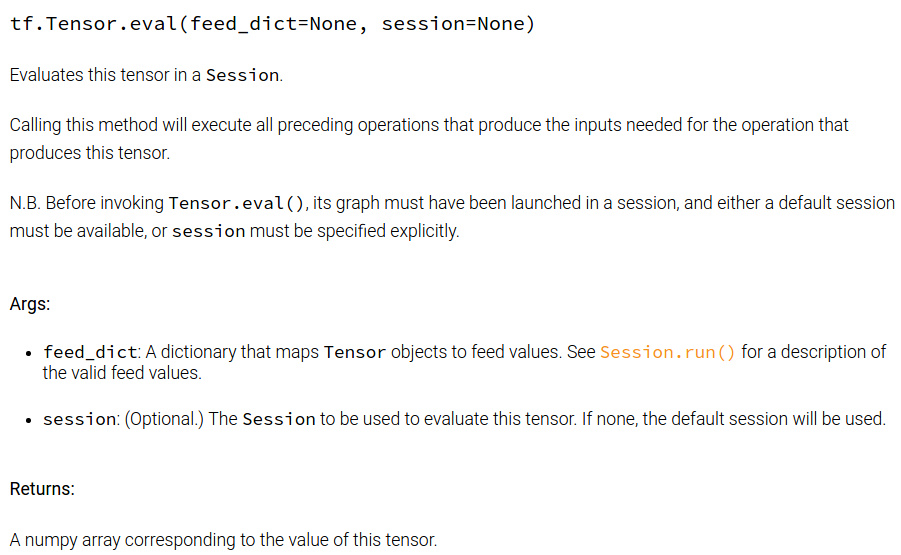
**how to change Variable value**

使用tf.assign(ref, value)函数。ref应该是个variable node，这个assign是个operation，因此需要在sess.run()中进行才能生效。这样之后再调用ref的值就发现改变成新值了。  
在这个模型中用于改变learning rate这个variable的值。

|  |  |
| --- | --- |
| 1  2 | def assign\_lr(self, session, lr\_value):  session.run(tf.assign(self.lr, lr\_value)) |

**run\_epoch()**

**Tensor.eval()**



比如定义了一个tensor x,x.eval(feed\_dict={xxx})就可以得到x的值，而不用sess.run(x, feed\_dict={xxx})。返回值是一个numpy array。