之前讲过了tensorflow中CNN的示例代码，现在我们来看RNN的代码。不过好像官方只给了LSTM的代码。那么我们就来看LSTM吧。LSTM的具体原理就不讲了，可以参见[深度学习笔记(五)：LSTM](http://blog.csdn.net/u014595019/article/details/52605693)，讲的非常清楚。

坦白说，这份写LSTM的代码有点难，倒不是说LSTM的原理有多难，而是这份代码中使用了大量tf提供的现成的操作函数。在精简了代码的同时，也增加了初学者阅读的难度。很多函数的用法我是去看源码，然后自己写示例代码才搞懂的。当然如果能把整份代码搞清楚的话，掌握这么多操作函数还是非常有用的。

这份代码并没有完整的出现在tf给出的示例中[见这里](https://www.tensorflow.org/versions/r0.11/tutorials/recurrent/index.html#recurrent-neural-networks)，而是只挑选了几个片段简略的介绍了一下。我当时看完之后简直是一头雾水。后来在github找到了这份代码的[完整文件](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/models/rnn/ptb/ptb_word_lm.py)，发现这份文件只能在命令行里面运行，需要输入参数，例如

python ptb\_word\_lm.py --data\_path=/tmp/simple-examples/data/ --model small

* 1

后来我改写了一下，使之可以直接运行。当然，运行之前需要先手动下载数据集，数据集的地址在[这里](http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz)

分段讲解

总的来看，这份代码主要由三步分组成。   
第一部分，是PTBModel,也是最核心的部分，负责tf中模型的构建和各种操作(op)的定义。   
第二部分，是run\_epoch函数，负责将所有文本内容分批喂给模型（PTBModel）训练。   
第三部分，就是main函数了，负责将第二部分的run\_epoch运行多遍，也就是说，文本中的每个内容都会被重复多次的输入到模型中进行训练。随着训练的进行，会适当的进行一些参数的调整。   
下面就按照这几部分来分开讲一下。我在后面提供了完整的代码，所以可以将完整代码和分段讲解对照着看。

参数设置

在构建模型和训练之前，我们首先需要设置一些参数。tf中可以使用tf.flags来进行全局的参数设置

flags = tf.flags

logging = tf.logging

flags.DEFINE\_string( # 定义变量 model的值为small, 后面的是注释

"model", "small",

"A type of model. Possible options are: small, medium, large.")

flags.DEFINE\_string("data\_path", #定义下载好的数据的存放位置

'/home/multiangle/download/simple-examples/data/',

"data\_path")

flags.DEFINE\_bool("use\_fp16", False, # 是否使用 float16格式？

"Train using 16-bit floats instead of 32bit floats")

FLAGS = flags.FLAGS # 可以使用FLAGS.model来调用变量 model的值。

def data\_type():

return tf.float16 if FLAGS.use\_fp16 else tf.float32

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

细心的人可能会注意到上面有行代码定义了model的值为small.这个是什么意思呢？其实在后面的完整代码部分可以看到，作者在其中定义了几个参数类，分别有small,medium,large和test这4种参数。如果model的值为small，则会调用SmallConfig，其他同样。在SmallConfig中，有如下几个参数：

init\_scale = 0.1 # 相关参数的初始值为随机均匀分布，范围是[-init\_scale,+init\_scale]

learning\_rate = 1.0 # 学习速率,在文本循环次数超过max\_epoch以后会逐渐降低

max\_grad\_norm = 5 # 用于控制梯度膨胀，如果梯度向量的L2模超过max\_grad\_norm，则等比例缩小

num\_layers = 2 # lstm层数

num\_steps = 20 # 单个数据中，序列的长度。

hidden\_size = 200 # 隐藏层中单元数目

max\_epoch = 4 # epoch<max\_epoch时，lr\_decay值=1,epoch>max\_epoch时,lr\_decay逐渐减小

max\_max\_epoch = 13 # 指的是整个文本循环次数。

keep\_prob = 1.0 # 用于dropout.每批数据输入时神经网络中的每个单元会以1-keep\_prob的概率不工作，可以防止过拟合

lr\_decay = 0.5 # 学习速率衰减

batch\_size = 20 # 每批数据的规模，每批有20个。

vocab\_size = 10000 # 词典规模，总共10K个词

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

其他的几个参数类中，参数类型都是一样的，只是参数的值各有所不同。

PTBModel

这个可以说是核心部分了。而具体来说，又可以分成几个小部分：**多层LSTM结构的构建，输入预处理，LSTM的循环，损失函数计算，梯度计算和修剪**

LSTM结构

self.batch\_size = batch\_size = config.batch\_size

self.num\_steps = num\_steps = config.num\_steps

size = config.hidden\_size # 隐藏层规模

vocab\_size = config.vocab\_size # 词典规模

self.\_input\_data = tf.placeholder(tf.int32, [batch\_size, num\_steps]) # 输入

self.\_targets = tf.placeholder(tf.int32, [batch\_size, num\_steps]) # 预期输出，两者都是index序列，长度为num\_step

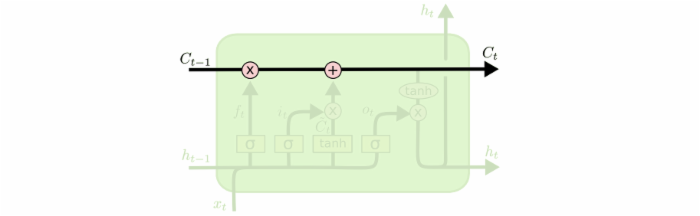
* 1
* 2
* 3
* 4
* 5
* 6
* 7

首先引进参数，然后定义2个占位符，分别表示输入和预期输出。注意此时不论是input还是target都是用词典id来表示单词的。

lstm\_cell = tf.nn.rnn\_cell.BasicLSTMCell(size, forget\_bias=0.0, state\_is\_tuple=True)

* 1

首先使用tf.nn.rnn\_cell.BasicLSTMCell定义单个基本的LSTM单元。这里的size其实就是hidden\_size。   
从源码中可以看到，在LSTM单元中，有2个状态值，分别是c和h，分别对应于下图中的c和h。其中h在作为当前时间段的输出的同时，也是下一时间段的输入的一部分。



那么当state\_is\_tuple=True的时候，state是元组形式，state=(c,h)。如果是False，那么state是一个由c和h拼接起来的张量，state=tf.concat(1,[c,h])。**在运行时，则返回2值，一个是h，还有一个state。**

DropoutWrapper

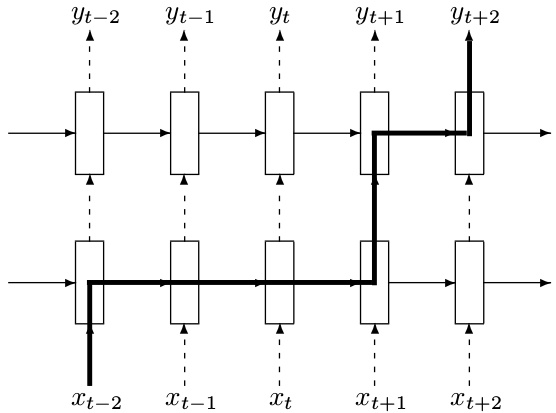
if is\_training and config.keep\_prob < 1: # 在外面包裹一层dropout

lstm\_cell = tf.nn.rnn\_cell.DropoutWrapper(

lstm\_cell, output\_keep\_prob=config.keep\_prob)

* 1
* 2
* 3

我们在这里使用了dropout方法。**所谓dropout,就是指网络中每个单元在每次有数据流入时以一定的概率(keep prob)正常工作，否则输出0值**。这是是一种有效的正则化方法，可以有效防止过拟合。在rnn中使用dropout的方法和cnn不同，推荐大家去把[recurrent neural network regularization](http://arxiv.org/pdf/1409.2329.pdf)看一遍。   
在rnn中进行dropout时，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，如下图所示



上图中，t-2时刻的输入xt−2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。

在使用tf.nn.rnn\_cell.DropoutWrapper时，同样有一些参数，例如input\_keep\_prob,output\_keep\_prob等，分别控制输入和输出的dropout概率，很好理解。

多层LSTM结构和状态初始化

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

# 参数初始化,rnn\_cell.RNNCell.zero\_stat

self.\_initial\_state = cell.zero\_state(batch\_size, data\_type())

* 1
* 2
* 3
* 4

在这个示例中，我们使用了2层的LSTM网络。也就是说，前一层的LSTM的输出作为后一层的输入。使用tf.nn.rnn\_cell.MultiRNNCell可以实现这个功能。这个基本没什么好说的，state\_is\_tuple用法也跟之前的类似。构造完多层LSTM以后，使用zero\_state即可对各种状态进行初始化。

输入预处理

with tf.device("/cpu:0"):

embedding = tf.get\_variable(

# vocab size \* hidden size, 将单词转成embedding描述

"embedding", [vocab\_size, size], dtype=data\_type())

# 将输入seq用embedding表示, shape=[batch, steps, hidden\_size]

inputs = tf.nn.embedding\_lookup(embedding, self.\_input\_data)

if is\_training and config.keep\_prob < 1:

inputs = tf.nn.dropout(inputs, config.keep\_prob)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

之前有提到过，输入模型的input和target都是用词典id表示的。例如一个句子，“我/是/学生”，这三个词在词典中的序号分别是0,5,3，那么上面的句子就是[0,5,3]。显然这个是不能直接用的，我们要把词典id转化成向量,也就是embedding形式。可能有些人已经听到过这种描述了。实现的方法很简单。

第一步，构建一个矩阵，就叫embedding好了，尺寸为[vocab\_size, embedding\_size]，分别表示词典中单词数目，以及要转化成的向量的维度。一般来说，向量维度越高，能够表现的信息也就越丰富。

第二步，使用tf.nn.embedding\_lookup(embedding,input\_ids) 假设input\_ids的长度为len，那么返回的张量尺寸就为[len,embedding\_size]。举个栗子

# 示例代码

import tensorflow as tf

import numpy as np

sess = tf.InteractiveSession()

embedding = tf.Variable(np.identity(5,dtype=np.int32))

input\_ids = tf.placeholder(dtype=tf.int32,shape=[None])

input\_embedding = tf.nn.embedding\_lookup(embedding,input\_ids)

sess.run(tf.initialize\_all\_variables())

print(sess.run(embedding))

#[[1 0 0 0 0]

# [0 1 0 0 0]

# [0 0 1 0 0]

# [0 0 0 1 0]

# [0 0 0 0 1]]

print(sess.run(input\_embedding,feed\_dict={input\_ids:[1,2,3,0,3,2,1]}))

#[[0 1 0 0 0]

# [0 0 1 0 0]

# [0 0 0 1 0]

# [1 0 0 0 0]

# [0 0 0 1 0]

# [0 0 1 0 0]

# [0 1 0 0 0]]

* 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

第三步，如果keep\_prob<1， 那么还需要对输入进行dropout。不过这边跟rnn的dropout又有所不同，这边使用tf.nn.dropout。

LSTM循环

现在，多层lstm单元已经定义完毕，输入也已经经过预处理了。那么现在要做的就是将数据输入lstm进行训练了。其实很简单，只要按照文本顺序依次向cell输入数据就好了。lstm上一时间段的状态会自动参与到当前时间段的输出和状态的计算当中。

outputs = []

state = self.\_initial\_state # state 表示 各个batch中的状态

with tf.variable\_scope("RNN"):

for time\_step in range(num\_steps):

if time\_step > 0: tf.get\_variable\_scope().reuse\_variables()

# cell\_out: [batch, hidden\_size]

(cell\_output, state) = cell(inputs[:, time\_step, :], state) # 按照顺序向cell输入文本数据

outputs.append(cell\_output) # output: shape[num\_steps][batch,hidden\_size]

# 把之前的list展开，成[batch, hidden\_size\*num\_steps],然后 reshape, 成[batch\*numsteps, hidden\_size]

output = tf.reshape(tf.concat(1, outputs), [-1, size])

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

这边要注意，tf.get\_variable\_scope().reuse\_variables()这行代码不可少，不然会报错，应该是因为同一命名域(variable\_scope)内不允许存在多个同一名字的变量的原因。

损失函数计算

# softmax\_w , shape=[hidden\_size, vocab\_size], 用于将distributed表示的单词转化为one-hot表示

softmax\_w = tf.get\_variable(

"softmax\_w", [size, vocab\_size], dtype=data\_type())

softmax\_b = tf.get\_variable("softmax\_b", [vocab\_size], dtype=data\_type())

# [batch\*numsteps, vocab\_size] 从隐藏语义转化成完全表示

logits = tf.matmul(output, softmax\_w) + softmax\_b

# loss , shape=[batch\*num\_steps]

# 带权重的交叉熵计算

loss = tf.nn.seq2seq.sequence\_loss\_by\_example(

[logits], # output [batch\*numsteps, vocab\_size]

[tf.reshape(self.\_targets, [-1])], # target, [batch\_size, num\_steps] 然后展开成一维【列表】

[tf.ones([batch\_size \* num\_steps], dtype=data\_type())]) # weight

self.\_cost = cost = tf.reduce\_sum(loss) / batch\_size # 计算得到平均每批batch的误差

self.\_final\_state = state

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

上面代码的上半部分主要用来将多层lstm单元的输出转化成one-hot表示的向量。关于one-hot presentation和distributed presentation的区别，可以参考[这里](http://blog.csdn.net/u014595019/article/details/51884529#t0)

代码的下半部分，正式开始计算损失函数。这里使用了tf提供的现成的交叉熵计算函数，tf.nn.seq2seq.sequence\_loss\_by\_example。不知道交叉熵是什么？见[这里](http://blog.csdn.net/u014595019/article/details/52562159#t7)各个变量的具体shape我都在注释中标明了。注意其中的self.\_targets是词典id表示的。这个函数的具体实现方式不明。我曾经想自己手写一个交叉熵，不过好像tf不支持对张量中单个元素的操作。

梯度计算

之前已经计算得到了每批数据的平均误差。那么下一步，就是根据误差来进行参数修正了。当然，首先必须要求梯度

self.\_lr = tf.Variable(0.0, trainable=False) # lr 指的是 learning\_rate

tvars = tf.trainable\_variables()

* 1
* 2

**通过tf.trainable\_variables 可以得到整个模型中所有trainable=True的Variable**。实际得到的tvars是一个列表，里面存有所有可以进行训练的变量。

grads, \_ = tf.clip\_by\_global\_norm(tf.gradients(cost, tvars),

config.max\_grad\_norm)

* 1
* 2

这一行代码其实使用了两个函数，tf.gradients 和 tf.clip\_by\_global\_norm。 我们一个一个来。

**tf.gradients**   
用来计算导数。该函数的定义如下所示

def gradients(ys,

xs,

grad\_ys=None,

name="gradients",

colocate\_gradients\_with\_ops=False,

gate\_gradients=False,

aggregation\_method=None):

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

虽然可选参数很多，但是最常使用的还是ys和xs。根据说明得知，ys和xs都可以是一个tensor或者tensor列表。而计算完成以后，该函数会返回一个长为len(xs)的tensor列表，列表中的每个tensor是ys中每个值对xs[i]求导之和。如果用数学公式表示的话，那么 g = tf.gradients(y,x)可以表示成 

gi=∑j=0len(y)∂yj∂xig=[g0,g1,...,glen(x)]

梯度修剪

**tf.clip\_by\_global\_norm**   
修正梯度值，用于**控制梯度爆炸的问题**。梯度爆炸和梯度弥散的原因一样，都是因为链式法则求导的关系，导致梯度的指数级衰减。为了避免梯度爆炸，需要对梯度进行修剪。   
先来看这个函数的定义：

def clip\_by\_global\_norm(t\_list, clip\_norm, use\_norm=None, name=None):

* 1

**输入参数**中：t\_list为待修剪的张量, clip\_norm 表示修剪比例(clipping ratio).

函数**返回2个参数**： list\_clipped，修剪后的张量，以及global\_norm，一个中间计算量。当然如果你之前已经计算出了global\_norm值，你可以在use\_norm选项直接指定global\_norm的值。

那么具体**如何计算**呢？根据源码中的说明，可以得到   
list\_clipped[i]=t\_list[i] \* clip\_norm / max(global\_norm, clip\_norm),其中   
global\_norm = sqrt(sum([l2norm(t)\*\*2 for t in t\_list]))

如果你更熟悉数学公式，则可以写作 

Lic=Lit∗Ncmax(Nc,Ng)Ng=∑i(Lit)2−−−−−−−√

其中，   
Lic和Lig代表t\_list[i]和list\_clipped[i]，   
Nc和Ng代表clip\_norm 和 global\_norm的值。   
其实也可以看到其实Ng就是t\_list的L2模。上式也可以进一步写作 

Lic={Lit,(Ng<=Nc)Lit∗NcNg,(Ng>Nc)Ng=∑i(Lit)2−−−−−−−√

也就是说，当t\_list的L2模大于指定的Nc时，就会对t\_list做等比例缩放

优化参数

之前的代码已经求得了合适的梯度，现在需要使用这些梯度来更新参数的值了。

# 梯度下降优化，指定学习速率

optimizer = tf.train.GradientDescentOptimizer(self.\_lr)

# optimizer = tf.train.AdamOptimizer()

# optimizer = tf.train.GradientDescentOptimizer(0.5)

self.\_train\_op = optimizer.apply\_gradients(zip(grads, tvars)) # 将梯度应用于变量

# self.\_train\_op = optimizer.minimize(grads)

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

这一部分就比较自由了，tf提供了很多种优化器，例如最常用的梯度下降优化（GradientDescentOptimizer）也可以使用AdamOptimizer。这里使用的是梯度优化。值得注意的是，这里使用了optimizer.apply\_gradients来将求得的梯度用于参数修正，而不是之前简单的optimizer.minimize(cost)

还有一点，要留心一下self.\_train\_op，只有该操作被模型执行，才能对参数进行优化。如果没有执行该操作，则参数就不会被优化。

run\_epoch

这就是我之前讲的第二部分，主要功能是将所有文档分成多个批次交给模型去训练，同时记录模型返回的cost,state等记录，并阶段性的将结果输出。

def run\_epoch(session, model, data, eval\_op, verbose=False):

"""Runs the model on the given data."""

# epoch\_size 表示批次总数。也就是说，需要向session喂这么多批数据

epoch\_size = ((len(data) // model.batch\_size) - 1) // model.num\_steps # // 表示整数除法

start\_time = time.time()

costs = 0.0

iters = 0

state = session.run(model.initial\_state)

for step, (x, y) in enumerate(reader.ptb\_iterator(data, model.batch\_size,

model.num\_steps)):

fetches = [model.cost, model.final\_state, eval\_op] # 要获取的值

feed\_dict = {} # 设定input和target的值

feed\_dict[model.input\_data] = x

feed\_dict[model.targets] = y

for i, (c, h) in enumerate(model.initial\_state):

feed\_dict[c] = state[i].c

feed\_dict[h] = state[i].h

cost, state, \_ = session.run(fetches, feed\_dict) # 运行session,获得cost和state

costs += cost # 将 cost 累积

iters += model.num\_steps

if verbose and step % (epoch\_size // 10) == 10: # 也就是每个epoch要输出10个perplexity值

print("%.3f perplexity: %.3f speed: %.0f wps" %

(step \* 1.0 / epoch\_size, np.exp(costs / iters),

iters \* model.batch\_size / (time.time() - start\_time)))

return np.exp(costs / iters)

* 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

基本没什么其他的，就是要注意传入的eval\_op。在训练阶段，会往其中传入train\_op，这样模型就会自动进行优化；而在交叉检验和测试阶段，传入的是tf.no\_op，此时模型就不会优化。

main函数

这里略去了数据读取和参数读取的代码，只贴了最关键的一部分。

with tf.Graph().as\_default(), tf.Session() as session:

# 定义如何对参数变量初始化

initializer = tf.random\_uniform\_initializer(-config.init\_scale,

config.init\_scale)

with tf.variable\_scope("model", reuse=None,initializer=initializer):

m = PTBModel(is\_training=True, config=config)

with tf.variable\_scope("model", reuse=True,initializer=initializer):

mvalid = PTBModel(is\_training=False, config=config)

mtest = PTBModel(is\_training=False, config=eval\_config)

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

注意这里定义了3个模型，对于训练模型，is\_trainable=True; 而对于交叉检验和测试模型，is\_trainable=False

summary\_writer = tf.train.SummaryWriter('/tmp/lstm\_logs',session.graph)

tf.initialize\_all\_variables().run() # 对参数变量初始化

for i in range(config.max\_max\_epoch): # 所有文本要重复多次进入模型训练

# learning rate 衰减

# 在 遍数小于max epoch时， lr\_decay = 1 ; > max\_epoch时， lr\_decay = 0.5^(i-max\_epoch)

lr\_decay = config.lr\_decay \*\* max(i - config.max\_epoch, 0.0)

m.assign\_lr(session, config.learning\_rate \* lr\_decay) # 设置learning rate

print("Epoch: %d Learning rate: %.3f" % (i + 1, session.run(m.lr)))

train\_perplexity = run\_epoch(session, m, train\_data, m.train\_op,verbose=True) # 训练困惑度

print("Epoch: %d Train Perplexity: %.3f" % (i + 1, train\_perplexity))

valid\_perplexity = run\_epoch(session, mvalid, valid\_data, tf.no\_op()) # 检验困惑度

print("Epoch: %d Valid Perplexity: %.3f" % (i + 1, valid\_perplexity))

test\_perplexity = run\_epoch(session, mtest, test\_data, tf.no\_op()) # 测试困惑度

print("Test Perplexity: %.3f" % test\_perplexity)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

注意上面train\_perplexity操作中传入了m.train\_op，表示要进行优化，而在valid\_perplexity和test\_perplexity中均传入了tf.no\_op，表示不进行优化。

完整代码和注释

# Copyright 2015 The TensorFlow Authors. All Rights Reserved.

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

# ==============================================================================

"""Example / benchmark for building a PTB LSTM model.

Trains the model described in:

(Zaremba, et. al.) Recurrent Neural Network Regularization

http://arxiv.org/abs/1409.2329

There are 3 supported model configurations:

===========================================

| config | epochs | train | valid | test

===========================================

| small | 13 | 37.99 | 121.39 | 115.91

| medium | 39 | 48.45 | 86.16 | 82.07

| large | 55 | 37.87 | 82.62 | 78.29

The exact results may vary depending on the random initialization.

The hyperparameters used in the model:

- init\_scale - the initial scale of the weights

- learning\_rate - the initial value of the learning rate

- max\_grad\_norm - the maximum permissible norm of the gradient

- num\_layers - the number of LSTM layers

- num\_steps - the number of unrolled steps of LSTM

- hidden\_size - the number of LSTM units

- max\_epoch - the number of epochs trained with the initial learning rate

- max\_max\_epoch - the total number of epochs for training

- keep\_prob - the probability of keeping weights in the dropout layer

- lr\_decay - the decay of the learning rate for each epoch after "max\_epoch"

- batch\_size - the batch size

The data required for this example is in the data/ dir of the

PTB dataset from Tomas Mikolov's webpage:

$ wget http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz

$ tar xvf simple-examples.tgz

To run:

$ python ptb\_word\_lm.py --data\_path=simple-examples/data/

"""

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import time

import numpy as np

import tensorflow as tf

from tensorflow.models.rnn.ptb import reader

flags = tf.flags

logging = tf.logging

flags.DEFINE\_string(

"model", "small",

"A type of model. Possible options are: small, medium, large.")

flags.DEFINE\_string("data\_path", '/home/multiangle/download/simple-examples/data/', "data\_path")

flags.DEFINE\_bool("use\_fp16", False,

"Train using 16-bit floats instead of 32bit floats")

FLAGS = flags.FLAGS

def data\_type():

return tf.float16 if FLAGS.use\_fp16 else tf.float32

class PTBModel(object):

"""The PTB model."""

def \_\_init\_\_(self, is\_training, config):

"""

:param is\_training: 是否要进行训练.如果is\_training=False,则不会进行参数的修正。

"""

self.batch\_size = batch\_size = config.batch\_size

self.num\_steps = num\_steps = config.num\_steps

size = config.hidden\_size

vocab\_size = config.vocab\_size

self.\_input\_data = tf.placeholder(tf.int32, [batch\_size, num\_steps]) # 输入

self.\_targets = tf.placeholder(tf.int32, [batch\_size, num\_steps]) # 预期输出，两者都是index序列，长度为num\_step

# Slightly better results can be obtained with forget gate biases

# initialized to 1 but the hyperparameters of the model would need to be

# different than reported in the paper.

lstm\_cell = tf.nn.rnn\_cell.BasicLSTMCell(size, forget\_bias=0.0, state\_is\_tuple=True)

if is\_training and config.keep\_prob < 1: # 在外面包裹一层dropout

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) # 多层lstm cell 堆叠起来

self.\_initial\_state = cell.zero\_state(batch\_size, data\_type()) # 参数初始化,rnn\_cell.RNNCell.zero\_state

with tf.device("/cpu:0"):

embedding = tf.get\_variable(

"embedding", [vocab\_size, size], dtype=data\_type()) # vocab size \* hidden size, 将单词转成embedding描述

# 将输入seq用embedding表示, shape=[batch, steps, hidden\_size]

inputs = tf.nn.embedding\_lookup(embedding, self.\_input\_data)

if is\_training and config.keep\_prob < 1:

inputs = tf.nn.dropout(inputs, config.keep\_prob)

# Simplified version of tensorflow.models.rnn.rnn.py's rnn().

# This builds an unrolled LSTM for tutorial purposes only.

# In general, use the rnn() or state\_saving\_rnn() from rnn.py.

#

# The alternative version of the code below is:

#

# inputs = [tf.squeeze(input\_, [1])

# for input\_ in tf.split(1, num\_steps, inputs)]

# outputs, state = tf.nn.rnn(cell, inputs, initial\_state=self.\_initial\_state)

outputs = []

state = self.\_initial\_state # state 表示 各个batch中的状态

with tf.variable\_scope("RNN"):

for time\_step in range(num\_steps):

if time\_step > 0: tf.get\_variable\_scope().reuse\_variables()

# cell\_out: [batch, hidden\_size]

(cell\_output, state) = cell(inputs[:, time\_step, :], state)

outputs.append(cell\_output) # output: shape[num\_steps][batch,hidden\_size]

# 把之前的list展开，成[batch, hidden\_size\*num\_steps],然后 reshape, 成[batch\*numsteps, hidden\_size]

output = tf.reshape(tf.concat(1, outputs), [-1, size])

# softmax\_w , shape=[hidden\_size, vocab\_size], 用于将distributed表示的单词转化为one-hot表示

softmax\_w = tf.get\_variable(

"softmax\_w", [size, vocab\_size], dtype=data\_type())

softmax\_b = tf.get\_variable("softmax\_b", [vocab\_size], dtype=data\_type())

# [batch\*numsteps, vocab\_size] 从隐藏语义转化成完全表示

logits = tf.matmul(output, softmax\_w) + softmax\_b

# loss , shape=[batch\*num\_steps]

# 带权重的交叉熵计算

loss = tf.nn.seq2seq.sequence\_loss\_by\_example(

[logits], # output [batch\*numsteps, vocab\_size]

[tf.reshape(self.\_targets, [-1])], # target, [batch\_size, num\_steps] 然后展开成一维【列表】

[tf.ones([batch\_size \* num\_steps], dtype=data\_type())]) # weight

self.\_cost = cost = tf.reduce\_sum(loss) / batch\_size # 计算得到平均每批batch的误差

self.\_final\_state = state

if not is\_training: # 如果没有训练，则不需要更新state的值。

return

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

tvars = tf.trainable\_variables()

# clip\_by\_global\_norm: 梯度衰减，具体算法为t\_list[i] \* clip\_norm / max(global\_norm, clip\_norm)

# 这里gradients求导，ys和xs都是张量

# 返回一个长为len(xs)的张量，其中的每个元素都是\grad{\frac{dy}{dx}}

# clip\_by\_global\_norm 用于控制梯度膨胀,前两个参数t\_list, global\_norm, 则

# t\_list[i] \* clip\_norm / max(global\_norm, clip\_norm)

# 其中 global\_norm = sqrt(sum([l2norm(t)\*\*2 for t in t\_list]))

grads, \_ = tf.clip\_by\_global\_norm(tf.gradients(cost, tvars),

config.max\_grad\_norm)

# 梯度下降优化，指定学习速率

optimizer = tf.train.GradientDescentOptimizer(self.\_lr)

# optimizer = tf.train.AdamOptimizer()

# optimizer = tf.train.GradientDescentOptimizer(0.5)

self.\_train\_op = optimizer.apply\_gradients(zip(grads, tvars)) # 将梯度应用于变量

self.\_new\_lr = tf.placeholder(

tf.float32, shape=[], name="new\_learning\_rate") # 用于外部向graph输入新的 lr值

self.\_lr\_update = tf.assign(self.\_lr, self.\_new\_lr) # 使用new\_lr来更新lr的值

def assign\_lr(self, session, lr\_value):

# 使用 session 来调用 lr\_update 操作

session.run(self.\_lr\_update, feed\_dict={self.\_new\_lr: lr\_value})

@property

def input\_data(self):

return self.\_input\_data

@property

def targets(self):

return self.\_targets

@property

def initial\_state(self):

return self.\_initial\_state

@property

def cost(self):

return self.\_cost

@property

def final\_state(self):

return self.\_final\_state

@property

def lr(self):

return self.\_lr

@property

def train\_op(self):

return self.\_train\_op

class SmallConfig(object):

"""Small config."""

init\_scale = 0.1 #

learning\_rate = 1.0 # 学习速率

max\_grad\_norm = 5 # 用于控制梯度膨胀，

num\_layers = 2 # lstm层数

num\_steps = 20 # 单个数据中，序列的长度。

hidden\_size = 200 # 隐藏层规模

max\_epoch = 4 # epoch<max\_epoch时，lr\_decay值=1,epoch>max\_epoch时,lr\_decay逐渐减小

max\_max\_epoch = 13 # 指的是整个文本循环13遍。

keep\_prob = 1.0

lr\_decay = 0.5 # 学习速率衰减

batch\_size = 20 # 每批数据的规模，每批有20个。

vocab\_size = 10000 # 词典规模，总共10K个词

class MediumConfig(object):

"""Medium config."""

init\_scale = 0.05

learning\_rate = 1.0

max\_grad\_norm = 5

num\_layers = 2

num\_steps = 35

hidden\_size = 650

max\_epoch = 6

max\_max\_epoch = 39

keep\_prob = 0.5

lr\_decay = 0.8

batch\_size = 20

vocab\_size = 10000

class LargeConfig(object):

"""Large config."""

init\_scale = 0.04

learning\_rate = 1.0

max\_grad\_norm = 10

num\_layers = 2

num\_steps = 35

hidden\_size = 1500

max\_epoch = 14

max\_max\_epoch = 55

keep\_prob = 0.35

lr\_decay = 1 / 1.15

batch\_size = 20

vocab\_size = 10000

class TestConfig(object):

"""Tiny config, for testing."""

init\_scale = 0.1

learning\_rate = 1.0

max\_grad\_norm = 1

num\_layers = 1

num\_steps = 2

hidden\_size = 2

max\_epoch = 1

max\_max\_epoch = 1

keep\_prob = 1.0

lr\_decay = 0.5

batch\_size = 20

vocab\_size = 10000

def run\_epoch(session, model, data, eval\_op, verbose=False):

"""Runs the model on the given data."""

# epoch\_size 表示批次总数。也就是说，需要向session喂这么多次数据

epoch\_size = ((len(data) // model.batch\_size) - 1) // model.num\_steps # // 表示整数除法

start\_time = time.time()

costs = 0.0

iters = 0

state = session.run(model.initial\_state)

for step, (x, y) in enumerate(reader.ptb\_iterator(data, model.batch\_size,

model.num\_steps)):

fetches = [model.cost, model.final\_state, eval\_op] # 要进行的操作，注意训练时和其他时候eval\_op的区别

feed\_dict = {} # 设定input和target的值

feed\_dict[model.input\_data] = x

feed\_dict[model.targets] = y

for i, (c, h) in enumerate(model.initial\_state):

feed\_dict[c] = state[i].c # 这部分有什么用？看不懂

feed\_dict[h] = state[i].h

cost, state, \_ = session.run(fetches, feed\_dict) # 运行session,获得cost和state

costs += cost # 将 cost 累积

iters += model.num\_steps

if verbose and step % (epoch\_size // 10) == 10: # 也就是每个epoch要输出10个perplexity值

print("%.3f perplexity: %.3f speed: %.0f wps" %

(step \* 1.0 / epoch\_size, np.exp(costs / iters),

iters \* model.batch\_size / (time.time() - start\_time)))

return np.exp(costs / iters)

def get\_config():

if FLAGS.model == "small":

return SmallConfig()

elif FLAGS.model == "medium":

return MediumConfig()

elif FLAGS.model == "large":

return LargeConfig()

elif FLAGS.model == "test":

return TestConfig()

else:

raise ValueError("Invalid model: %s", FLAGS.model)

# def main(\_):

if \_\_name\_\_=='\_\_main\_\_':

if not FLAGS.data\_path:

raise ValueError("Must set --data\_path to PTB data directory")

print(FLAGS.data\_path)

raw\_data = reader.ptb\_raw\_data(FLAGS.data\_path) # 获取原始数据

train\_data, valid\_data, test\_data, \_ = raw\_data

config = get\_config()

eval\_config = get\_config()

eval\_config.batch\_size = 1

eval\_config.num\_steps = 1

with tf.Graph().as\_default(), tf.Session() as session:

initializer = tf.random\_uniform\_initializer(-config.init\_scale, # 定义如何对参数变量初始化

config.init\_scale)

with tf.variable\_scope("model", reuse=None,initializer=initializer):

m = PTBModel(is\_training=True, config=config) # 训练模型， is\_trainable=True

with tf.variable\_scope("model", reuse=True,initializer=initializer):

mvalid = PTBModel(is\_training=False, config=config) # 交叉检验和测试模型，is\_trainable=False

mtest = PTBModel(is\_training=False, config=eval\_config)

summary\_writer = tf.train.SummaryWriter('/tmp/lstm\_logs',session.graph)

tf.initialize\_all\_variables().run() # 对参数变量初始化

for i in range(config.max\_max\_epoch): # 所有文本要重复多次进入模型训练

# learning rate 衰减

# 在 遍数小于max epoch时， lr\_decay = 1 ; > max\_epoch时， lr\_decay = 0.5^(i-max\_epoch)

lr\_decay = config.lr\_decay \*\* max(i - config.max\_epoch, 0.0)

m.assign\_lr(session, config.learning\_rate \* lr\_decay) # 设置learning rate

print("Epoch: %d Learning rate: %.3f" % (i + 1, session.run(m.lr)))

train\_perplexity = run\_epoch(session, m, train\_data, m.train\_op,verbose=True) # 训练困惑度

print("Epoch: %d Train Perplexity: %.3f" % (i + 1, train\_perplexity))

valid\_perplexity = run\_epoch(session, mvalid, valid\_data, tf.no\_op()) # 检验困惑度

print("Epoch: %d Valid Perplexity: %.3f" % (i + 1, valid\_perplexity))

test\_perplexity = run\_epoch(session, mtest, test\_data, tf.no\_op()) # 测试困惑度

print("Test Perplexity: %.3f" % test\_perplexity)

# if \_\_name\_\_ == "\_\_main\_\_":

# tf.app.run()