tensorflow笔记系列：
（一）tensorflow笔记：流程，概念和简单代码注释
（二）tensorflow笔记：多层CNN代码分析
（三）tensorflow笔记：多层LSTM代码分析
（四）tensorflow笔记：常用函数说明
（五）tensorflow笔记：模型的保存与训练过程可视化
（六）tensorflow笔记：使用tf来实现word2vec

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，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，而是只挑选了几个片段简略的介绍了一下。我当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，发现这份文件只能在命令行里面行，需要输入参数，例如

```
python ptb_word_lm.py --data_path=/tmp/simple-examples/data/ --model small
```

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

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
```

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
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
```

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

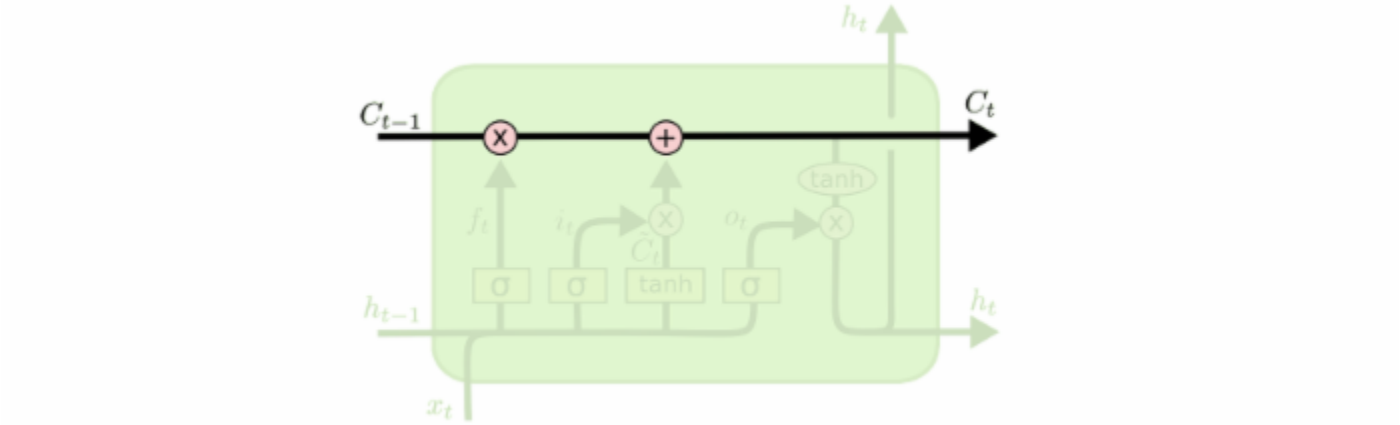
```
self.batch_size = batch_size = config.batch_size
self.num_steps = num_steps = config.num_steps
size = config.hidden_size       # 隐藏层规模
vocab_size = config.vocab_size  # 词典规模

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，
```

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
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)
```

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
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。

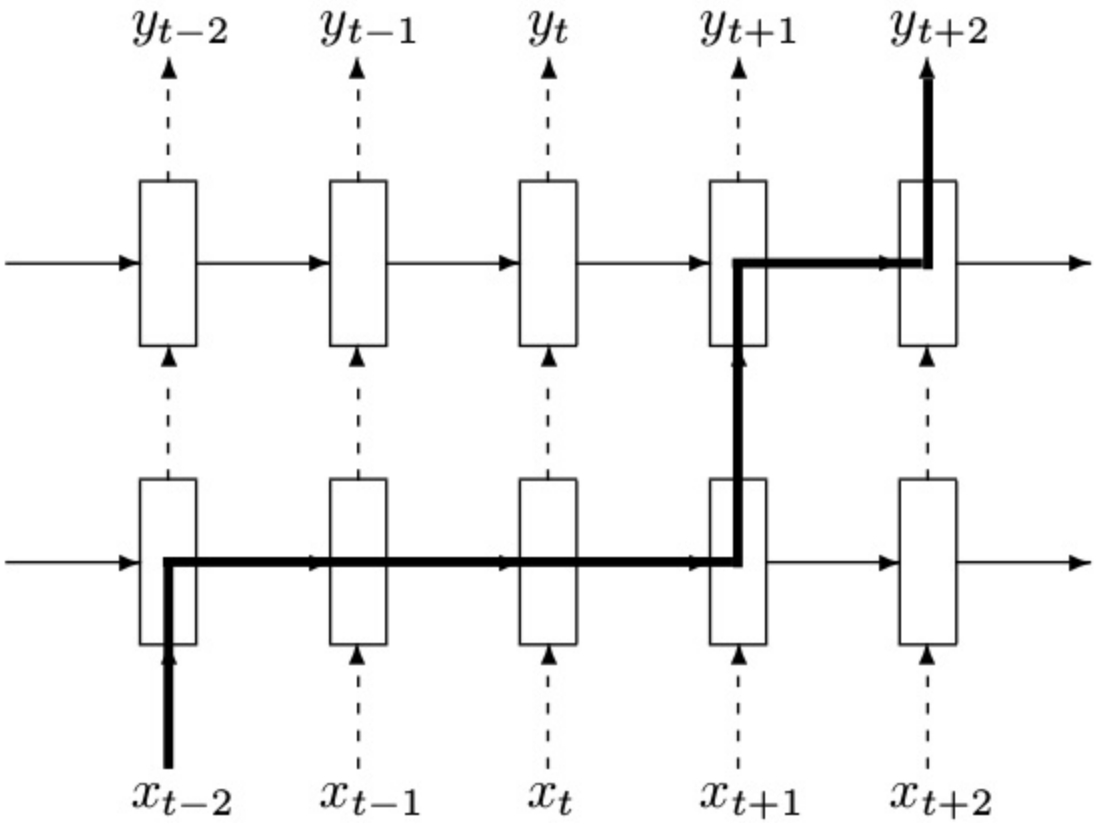


那么当state_is_tuple=True的时候，state是元组形式，state=(c,h)。如果是False，那么state是一个和h拼接起来的张量，state=tf.concat(1,[c,h])。**在运行时，则返回2值，一个是h，还有一个state。**

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)
```

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(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中用dropout的方法和cnn不同，推荐大家去把recurrent neural network regularization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x_{t-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c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())
```

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)
```

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
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
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
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
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]]
```

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])
```

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
```

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

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？见这里各个变量的具体shape我注释中标明了。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，首先须要求梯度

```
self._lr = tf.Variable(0.0, trainable=False)  # lr 指的是 learning_rate
tvars = tf.trainable_variables()
```

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tvar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)
```

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):
```

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ten
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
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

$$g_i = \sum_{j=0}^{len(y)} \frac{\partial y_j}{\partial x_i}$$

$$g = [g_0, g_1, \cdots, g_{len(x)}]$$

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):
```

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

$$L_c^i = \frac{L_t^i * N_c}{\max(N_c, N_g)}$$
$$N_g = \sqrt{\sum_i (L_t^i)^2}$$

其中，
L_c^i和L_g^i代表t_list[i]和list_clipped[i]，
N_c和N_g代表clip_norm 和 global_norm的值。
其实也可以看到其实N_g就是t_list的L2模。上式也可以进一步写作

$$L_c^i = \begin{cases} L_t^i, (N_g <= N_c) \\ L_t^i * \frac{N_c}{N_g}, (N_g > N_c) \end{cases}$$
$$N_g = \sqrt{\sum_i (L_t^i)^2}$$

也就是说，当t_list的L2模大于指定的N_c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)
```

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)
```

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)
```

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)
```

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ind

        # 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

        self._initial_state = cell.zero_state(batch_size, data_type()) # 参数初始化,rnn_cell.RNNCell.ze

        with tf.device("/cpu:0"):
            embedding = tf.get_variable(
                "embedding", [vocab_size, size], dtype=data_type()) # vocab size * hidden size, 将单词转
            # 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_si
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
            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=Fal
            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()
```