**基于LSTM语言模型的实现**

**1 相关知识**

**1.1 语言模型**

语言模型是计算单词序列的概率的模型，即通过计算一段文本的概率来判断这段文本是否通顺合理，对于语言中常见的句子，一个好的语言模型应得出相对较高的概率，而对与不合语法的句子，计算出的概率应接近零。语言模型在信息检索、机器翻译、语音识别等任务有着重要的作用。语言模型分为统计语言模型和神经网络语言模型。

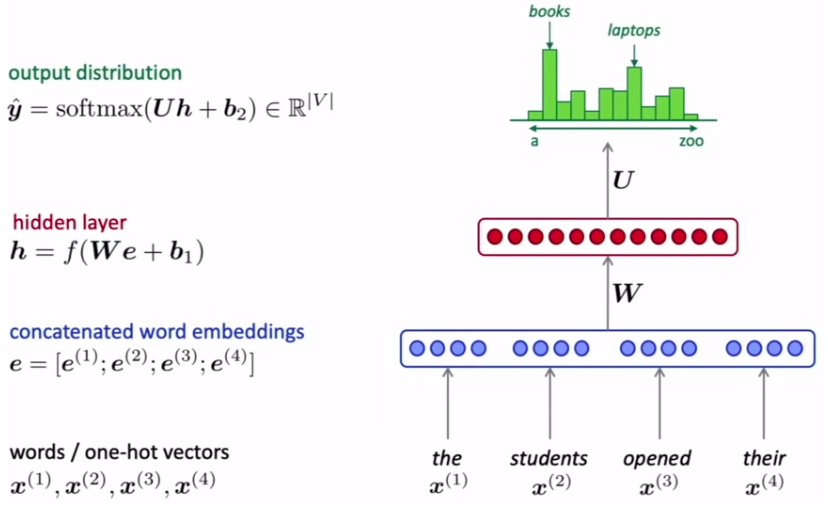
统计语言模型是从概率统计角度出发，解决自然语言上下文相关的特性的数学模型。假设一段文本S由,,,...,组成，若想知道S在文本中出现的可能性，就可以用下面的公式计算其联合概率：

如果文本较长，则此概率计算非常困安，于是基于马尔科夫假设，提出了N-gram语言模型：当前词只和它前面的n个词有关，与更前面的词无关，则上面公式为：

N-gram语言模型训练一般采用极大似然估计的方法：

当文本中出现意义相近但是没有在训练文本中出现的单词或者单词组的时候，这时候这个词出现的概率接近为零，可通过平滑方法解决这种矛盾。同时，模型依赖于固定单词组合，需要完全的模式匹配，导致模型的泛化能力差。

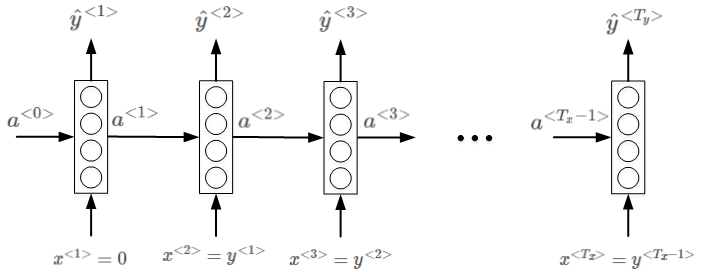
神经网络语言模型通过多层神经网络对输入和输出进行建模，模型的参数通过网络中神经元之间连接的权重进行体现。最早出现的是基于前馈神经网络的语言模型，如下图所示，每次从语料库中滑动N个单词，输入层将单词映射词向量并进行拼接输入到隐藏层，输出层使用softmax函数得到词典大小的向量，表示词典中每个词作为下一个词的可能性。神经网络模型解决了N-gram模型当n较大时会发生的数据稀疏问题。



前馈神经网络并没有考虑数据之间的关联性，网络的输出只和当前时刻网络的输入相关。在处理很多序列型数据时，例如文本、语音及视频等，这些数据具有时序上的关联性，即某一时刻网络的输出除了与当前时刻输入相关，还与之前某一个时刻或者某几个时刻相关。前馈神经网络没有记忆能力，因此并不能处理好这种关联性。

**1.2 循环神经网络（RNN）**

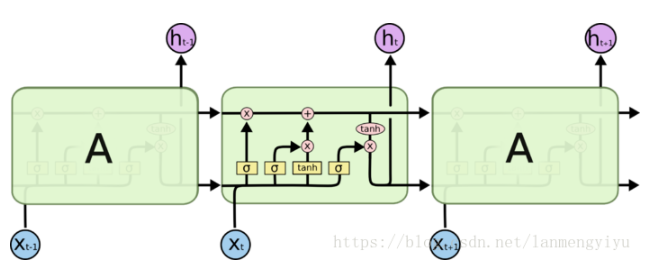
循环神经网络的模型解决了固定长度输入的问题。在RNN中，当前的输出不仅和当前的输入有关，还和上一时刻的输出有关，因此可以十分方便处理序列数据，而这个性质正是语言模型所需要的：预测的当前词要和前面的所有词有关，而不仅仅是前面的N-1个词。准备好训练集并对语料库进行分词处理后，可构建相应的RNN模型，RNN模型结构如图所示。



其中，每个时刻的输入为对应单词的词向量，第一个输出为语句的第一个单词的概率，第二个输出表示在第一个单词基础上出现第二个单词的概率，即条件概率，以此类推，最后输出出现<EOS>的条件概率。对语料库中的每条语句进行训练，最终可得到基于循环神经网络的语言模型。

**1.3 长短期记忆网络（LSTM）**

RNN模型中每个元素受其周围附近的影响，而当语句中某个单词与它距离较远的某个单词存在很大的依赖关系时，RNN容易出现梯度消失，无法捕捉它们之间的依赖。LSTM是一种特殊的RNN，能够学习到长期依赖。LSTM模型结构如图所示。



LSTM为了能记住长期状态，使用3个门来保护和控制细胞状态。LSTM第一步决定我们将要从细胞状态中扔掉哪些信息，该决定由遗忘门决定。在语言模型中，若遇到一个新的主语，则需要把旧的主语遗忘掉。下一步是决定将新的信息存储到细胞状态中，由输入门做决定。在语言模型中，可用于把新的主语更新到细胞中，来取代已经遗忘的主语。最后由输出门根据目前的细胞状态决定最后的输出。在语言模型中，若刚刚输出一个主语，则下一步可能需要输出与动词相关的信息，可能是主语是单数还是负数，以便于接下来选择正确的动词形式。

**1.4 语言模型的评估**

困惑度（PPL）被用来评估语言模型，它主要是根据每个词来估计一句话出现的概率，并用句子长度作归一化，公式为：

S代表句子，N是句子长度。当句子通顺合理时，PPL越小，而PPL越高，则表示句子越不连贯和混乱。

**2 语言模型的实现**

**2.1 数据集及开发环境**

项目使用PTB数据集，并基于Tensorflow框架，在windows操作系统、Python3.6的开发环境下来训练语言模型。

**2.2 项目实现**

（1）首先定义用来处理输入数据的模块ptb\_reader.py，用来读取PTB的三个文件：ptb.train.txt、ptb.valid.txt、ptb.test.txt。读取数据集后，提取语料库中的单词，并将每个单词映射到整数编号。最后，根据batch\_size和numsteps参数对数据集进行小批次训练的处理，生成成对的输入数据与目标数据。关键代码及注释如下：

#读取文件

def \_read\_words(filename):

with gfile.GFile(filename, "rb") as f:

return f.read().decode("utf-8").replace("\n", "<eos>").split()

#构造词典

def \_build\_vocab(filename):

data = \_read\_words(filename)

#统计单词出现的个数

counter = collections.Counter(data)

#按照单词的个数降序排列，count\_pairs是一个列表[(单词，count),...]

count\_pairs = sorted(counter.items(), key=lambda x: -x[1])

#提取所有单词

words, \_ = list(zip(\*count\_pairs))

#word\_to\_id {"单词":序号,...}

word\_to\_id = dict(zip(words, range(len(words))))

return word\_to\_id

#文件转成词表中的id号

def \_file\_to\_word\_ids(filename, word\_to\_id):

data = \_read\_words(filename)

return [word\_to\_id[word] for word in data]

#shutil.rmtree 递归删除目录

def save\_obj(obj, name):

if not os.path.exists('obj'):

os.makedirs('obj')

else:

shutil.rmtree('obj')

os.makedirs('obj')

with open('obj/'+ name + '.pkl', 'wb') as f:

pickle.dump(obj, f,protocol=2)

def load\_obj(name):

with open('obj/' + name + '.pkl', 'rb') as f:

return pickle.load(f)

#得到训练集、验证集、测试集

def ptb\_raw\_data(data\_path=None):

#连接两个路径

train\_path = os.path.join(data\_path, "ptb.train.txt")

valid\_path = os.path.join(data\_path, "ptb.valid.txt")

test\_path = os.path.join(data\_path, "ptb.test.txt")

word\_to\_id = \_build\_vocab(train\_path)

save\_obj(word\_to\_id,"word2id")

train\_data = \_file\_to\_word\_ids(train\_path, word\_to\_id)

valid\_data = \_file\_to\_word\_ids(valid\_path, word\_to\_id)

test\_data = \_file\_to\_word\_ids(test\_path, word\_to\_id)

vocabulary = len(word\_to\_id)

return train\_data, valid\_data, test\_data, vocabulary

def ptb\_test\_data(test\_path=None):

word\_to\_id = load\_obj("word2id")

test\_data = \_file\_to\_word\_ids(test\_path, word\_to\_id)

vocabulary = len(word\_to\_id)

return test\_data

#根据batch\_size和num\_steps参数分成批数据

def ptb\_iterator(raw\_data, batch\_size, num\_steps):

raw\_data = np.array(raw\_data, dtype=np.int32)

data\_len = len(raw\_data)

print(data\_len)

batch\_len = data\_len // batch\_size

data = np.zeros([batch\_size, batch\_len], dtype=np.int32)

for i in range(batch\_size):

data[i] = raw\_data[batch\_len \* i:batch\_len \* (i + 1)]

epoch\_size = (batch\_len - 1) // num\_steps

if epoch\_size == 0:

raise ValueError("epoch\_size == 0, decrease batch\_size or num\_steps")

#数组切片

for i in range(epoch\_size):

x = data[:, i\*num\_steps:(i+1)\*num\_steps]

y = data[:, i\*num\_steps+1:(i+1)\*num\_steps+1]

yield (x, y)

（2）定义LSTM语言模型的类LSTM，初始化方法为\_\_init\_\_，包含参数is\_training和config。当is\_training为True时，构造训练模型，否则为验证模型或测试模型。通过config参数读取batch\_size、numsteps、vocab\_size等参数到模型。

在初始化方法中，创建并词嵌入embedding，初始化行数设为词汇表数vocab\_size，单词向量表达维数为hiddens\_dimession，与LSTM单元内含节点数一致。tf.nn.embedding\_lookup函数查询单词对应的向量表达获得inputs。通过tf.contrib.rnn.BasicLSTMCell设置默认LSTM单元,隐含节点数为hidden\_dimession，在训练状态下，lstm输入和输出接接Dropout层，在输出层使用tf.contrib.rnn.DropoutWrappe函数实现，在输入层使用tf.nn.dropout。

循环神经网络一般在不同循环体结构中使用dropout，而不在同一层的循环结构体之间使用。通过使用RNN堆叠函数tf.contrib.rnn.BasicLSTMCell使得lstm\_cell堆叠，堆叠次数为num\_layers。设置网络的初始零状态，堆叠的LSTM读取单词，结合存储状态state计算下一单词出现的概率分布，每次读取单词，保存并更新状态。

定义输出outputs，tf.variable\_scope设名为LanguageModel。为了控制训练过程，限制梯度在反向传播时可以展开的步数为一个固定的值num\_steps。从第二次循环开始,使用tf.get\_variable().reuse\_variables()。每次循环，传入inputs和state到堆叠的LSTM单元，得到输出outputs和更新后的state。

将outputs内容用tf.concat串接到一起，并使用reshape将其转为一个一维向量。定义权重和偏置，tf.matmul输出outputs乘权重加偏置得到最后的输出logits。通过tf.contrib.legacy\_seq2seq.ssequence\_loss\_by\_example计算logits和targets偏差得到损失loss。定义平均误差cost，汇总batch的总误差，再计算到平均每个样本的误差。

模型在训练状态时，定义学习速率，设置为不可训练。tf.trianable\_variables获取全部可训练参数tvars,针对cost，计算tvars梯度，当梯度过大时，训练难以收敛。使用tf.clip\_by\_gloabal\_norm设置梯度最大范数，起正则化效果。定义优化器GradientDescent，并创建训练操作train\_op，用apply\_gradients将剪切过的梯度应用到所有可训练参数tvars。，定义通过外部控制学习速率的函数，通过执行lr\_update修改学习速率。关键代码如下：

class LSTM(object):

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

# Initialize the parameter values from the config.

self.batch\_size = config.batch\_size

self.num\_steps = config.num\_steps

dropout\_probability = config.keep\_prob

hidden\_dimension = config.hidden\_size

vocab\_size = config.vocab\_size

batch\_size = self.batch\_size

num\_steps = self.num\_steps

#placeholder()函数是在神经网络构建graph的时候在模型中的占位，数据类型，数据形状

self.input\_data = tf.placeholder(tf.int32, shape = [batch\_size, num\_steps])

# Convert input matrix to embedding matrix.变量名称"embedding",变量维度，uniform 均匀分布

#tf.get\_variable 根据变量的名称来获取变量或者创建变量

embedding = tf.get\_variable("embedding", [vocab\_size, hidden\_dimension],

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

#选取一个张量中对应索引的元素

input\_vector\_embedding = tf.nn.embedding\_lookup(embedding, self.input\_data)

lstm\_cell = tf.contrib.rnn.BasicLSTMCell(hidden\_dimension)

if is\_training:

lstm\_cell = tf.contrib.rnn.DropoutWrapper(lstm\_cell, output\_keep\_prob = dropout\_probability)

input\_vector\_embedding = tf.nn.dropout(input\_vector\_embedding, config.keep\_prob)

# Stacking of RNN with two layers of it.

stacked\_lstm\_cell = tf.contrib.rnn.MultiRNNCell([lstm\_cell] \* 2)

#生成全0的初始状态

self.initial\_state = stacked\_lstm\_cell.zero\_state(batch\_size, tf.float32)

output\_vector\_embedding = []

hidden\_cell\_states = []

current\_state = self.initial\_state

# Recurrent unit for simulating RNN.

with tf.variable\_scope("LangaugeModel",

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

for iter\_ in range(num\_steps):

if (iter\_ > 0):

tf.get\_variable\_scope().reuse\_variables()

current\_cell\_input = input\_vector\_embedding[:, iter\_, :]

(current\_cell\_output, current\_state) = stacked\_lstm\_cell(current\_cell\_input, current\_state)

output\_vector\_embedding.append(current\_cell\_output)

hidden\_cell\_states.append(current\_state)

self.final\_state = hidden\_cell\_states[-1]

output = tf.reshape(tf.concat(output\_vector\_embedding, 1), [-1, hidden\_dimension])

self.targets = tf.placeholder(tf.int32, [batch\_size, num\_steps])

targets = tf.reshape(self.targets, [-1])

#weight = tf.get\_variable("weight", [HIDDEN\_SIZE, VOCAB\_SIZE])

#bias = tf.get\_variable("bias", [VOCAB\_SIZE])

#logits = tf.matmul(output, weight) + bias

#weights = tf.get\_variable("weights",[hidden\_dimension,vocab\_size])

weights=tf.transpose(embedding)

bias=tf.get\_variable("bias",[vocab\_size],dtype=tf.float32)

logits = tf.matmul(output, weights)+bias

loss = tf.contrib.legacy\_seq2seq.sequence\_loss\_by\_example([logits], [targets], [tf.ones([batch\_size \* num\_steps])], vocab\_size)

self.cost = tf.reduce\_sum(loss)/ batch\_size

#反向传播更新参数

if is\_training:

#获取所有可训练的向量

self.lr=tf.Variable(0.0,trainable=False)

tvars = tf.trainable\_variables()

grads, \_ = tf.clip\_by\_global\_norm(tf.gradients(self.cost, tvars), 5)

optimizer = tf.train.GradientDescentOptimizer(self.lr)

#optimizer = tf.train.AdamOptimizer(0.9)

#optimizer = tf.train.RMSPropOptimizer(0.9)

self.train\_op = optimizer.apply\_gradients(zip(grads, tvars)) self.new\_lr=tf.placeholder(tf.float32,shape=[],name="new\_learning\_rate")

self.lr\_update=tf.assign(self.lr,self.new\_lr)

else:

self.train\_op = tf.no\_op()

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

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

（3）定义模型设置Config类，包括网络的参数设置。关键代码如下：

class Config(object):

batch\_size = 20 #批尺寸

num\_steps = 35 # number of unrolled time steps 展开的时间步数

hidden\_size = 450 # number of blocks in an LSTM cell 神经元个数

vocab\_size = 10000

max\_grad\_norm = 5 # maximum gradient for clipping 梯度裁剪

init\_scale = 0.05 # scale between -0.1 and 0.1 for all random initialization 权重初始

keep\_prob = 0.5 # dropout probability 让某个神经元以p的概率失效

num\_layers = 2 # number of LSTM layers

learning\_rate = 1.0

lr\_decay = 0.8 #学习率衰减因子

（4）定义训练函数run\_epoch。初始化损失total\_cost和iterations为零，执行model.initial\_state初始化模型状态，创建输出结果字典表，包括cost、final\_state。训练循环，次数为num\_epoches，每次循环，生成训练feed\_dict训练网络，得到cost和state，累加cost到total\_cost，累加num\_steps到iterations，每训练100批数据，则展示当前epoch进度和perplexity，定义test\_epoch函数用于模型测试。关键代码为：

def run\_epoch(sess,model,data):

epoch\_size=((len(data)//model.batch\_size) - 1)//model.num\_steps

saver=tf.train.Saver()

#初始化模型参数

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

total\_cost=0

iterations=0

for step,(x, y) in enumerate(ptb\_reader.ptb\_iterator(data, model.batch\_size, model.num\_steps)):

cost, state, \_ = sess.run([model.cost, model.final\_state, model.train\_op],

feed\_dict={model.input\_data: x,model.targets: y,model.initial\_state: state})

total\_cost += cost

iterations += model.num\_steps

perplexity = np.exp(total\_cost / iterations)

if step % 100 == 0:

progress = (step \*1.0/ epoch\_size) \* 100

print("%.1f%% Perplexity: %.3f (Cost: %.3f) " % (progress, perplexity, cost))

save\_path=saver.save(sess,"./saved\_model\_rnn/lstm-model.ckpt")

return (total\_cost / iterations), perplexity

（5）编写main函数用于训练模型和测试模型。创建默认Graph和Session，Graph定义了计算方式，Session用来执行Graph，保存中间结果和给Graph分配资源。使用LSTM模型类来创建训练、验证和测试模型，其中训练和验证模型使用配置为train\_config，测试模型使用的是eval\_config。根据num\_epoches，循环内执行训练和验证，并输出训练、验证和测试的perplexity。关键代码为：

def main(\_):

train\_config = Config()

eval\_config = Config()

eval\_config.num\_steps = 1

eval\_config.batch\_size=1

num\_epochs = 10

#if not FLAGS.test:

#print(FLAGS.test)

train\_data, valid\_data, test\_data, \_ = ptb\_reader.ptb\_raw\_data("../data")

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

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

train\_model = LSTM(is\_training=True, config=train\_config)

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

valid\_model=LSTM(is\_training=False,config=train\_config)

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

test\_model=LSTM(is\_training=False,config=eval\_config)

if not os.path.exists('saved\_model\_rnn'):

os.makedirs('saved\_model\_rnn')

else:

shutil.rmtree('saved\_model\_rnn')

os.makedirs('saved\_model\_rnn')

session.run(tf.global\_variables\_initializer())

for i in range(num\_epochs):

lr\_decay=train\_config.lr\_decay\*\*max(i+1-4,0.0)

train\_model.assign\_lr(session,train\_config.lr\_decay\*lr\_decay)

train\_cost, train\_perp = run\_epoch(session, train\_model, train\_data)

print("Epoch: %i Training Perplexity: %.3f (Cost: %.3f)" % (i + 1, train\_perp, train\_cost))

valid\_cost, valid\_perp = run\_epoch(session, valid\_model, valid\_data)

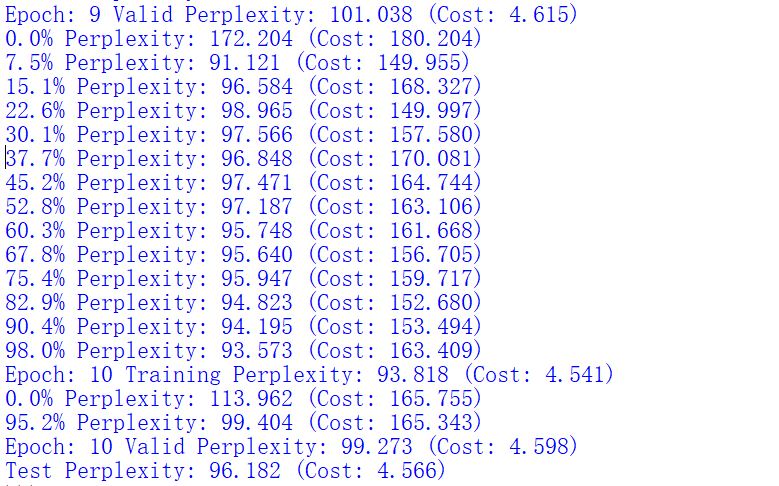
print("Epoch: %i Valid Perplexity: %.3f (Cost: %.3f)" % (i + 1, valid\_perp, valid\_cost))

test\_cost, test\_perp = test\_epoch(session, test\_model, test\_data)

print("Test Perplexity: %.3f (Cost: %.3f)" % (test\_perp, test\_cost))

**2.3 程序截图**

程序中将num\_epoches设置为10，得到的结果如图所示。



**3 结论**

本项目在PTB数据集使用Tensorflow实现了基于LSTM的语言模型。首先，对数据集进行处理，将数据集中的每个单词映射成整数，并切分成批数据。然后设计网络模型，定义输入、输出和训练过程。最后，根据配置和轮数进行训练。在实现过程中，了解到了语言模型、循环神经网络及长短期记忆网络的相关知识，并且对网络中的梯度裁剪、优化器、学习率更新、参数更新有了进一步的认识。但通过模型的评价指标来看，依然需要进一步调整算法或参数以实现更好的效果。