# 基于循环神经网络的语言建模

**计硕2004 赵宁宁 2001880**

**一、概念介绍**

**1、自然语言处理**

自然语言处理是是数据科学里的一个分支，它的主要覆盖的内容是：以一种智能与高效的方式，对文本数据进行系统化分析、理解与信息提取的过程。通过使用NLP以及它的组件，我们可以管理非常大块的文本数据，或者执行大量的自动化任务，并且解决各式各样的问题，如自动摘要，机器翻译，命名实体识别，关系提取，情感分析，语音识别，以及主题分割等等。

**2、语言模型**

语言模型是通过语料来计算出某个句子出现的概率。对于一段由w1,w2,…,wn这些词组成的句子，想要判断这条语句是否是正常语句，可以通过求这段文字的概率来表示其存在的可能性 。例如P(How are you)>P(You how are)。

**2.1传统的语言模型**

传统的语言模型为统计语言模型。语言模型中的词是有顺序的，判断这条语句是否合理，关键是看这些词的排列顺序是否正确。因此统计语言模型的基本思想就是计算其条件概率，由链式法则可以得到：



在统计语言模型中，采用极大似然估计来计算每个词出现的条件概率，即：



其中，表示子序列在训练集中出现的次数。然而对于任意长的自然语言语句，根据极大似然估计直接计算 显然不现实。为了解决这个问题，我们引入马尔可夫假设，即假设当前词出现的概率只依赖于前n-1个词，可以得到：



称为n-1阶马尔科夫近似，也称为n元语言模型，它解决了参数太多难以训练的问题，但仍存在着稀疏性和泛化能力弱的缺点。因为n元语言模型只能对文本中出现的单词或者单词组进行建模，当新的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，传统离散模型无法正确计算这些训练样本中未出现的单词的应有概率，他们都会被赋予0概率预测值。这是非常不符合语言规律的事情。

除了对未出现的单词本身进行预测非常困难之外，离散模型还依赖于固定单词组合，需要完全的模式匹配，否则也无法正确输出单词组出现的概率。

**2.2神经网络语言模型**

传统语言模型的上述几个内在缺陷使得人们开始把目光转向神经网络模型，期望深度学习技术能够自动化地学习代表语法和语义的特征，解决稀疏性问题，并提高泛化能力。

2.2.1 前馈神经网络

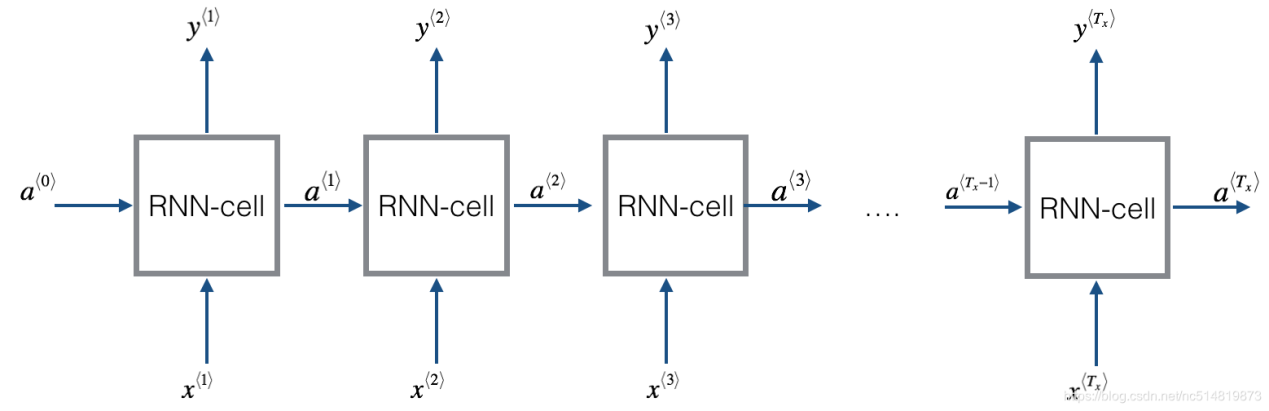
Bengio等人提出了前馈神经网络模型，其主要思想为先给每个词在连续空间中赋予一个向量(词向量)，再通过神经网络去学习这种分布式表征。利用神经网络去建模当前词出现的概率与其前 n-1 个词之间的约束关系。通过使用词向量的映射，前馈神经网络模型能解决稀疏性的问题。一些在训练集中没有遇到过单词由于其与上下文同时出现的关系，在词向量的空间中会与相类似的单词处于相近的位置，从而降低出现接近于0的条件概率的问题。

该模型在实际应用过程中表现出了一定的泛化能力，但是该模型没有明确地对超出观察窗口的上下文信息进行处理。为了解决固定窗口问题，采用循环神经网络模型RNN。

2.2.2循环神经网络

前馈神经网络模型假设元素之间是相互独立的，输入与输出也是相互独立的，然而在经常一起出现的单词以后也经常出现的概率会更高，并且当前应该出现的词通常是由前面一段文字决定的，而循环神经网络的结构能利用文字的这种上下文序列关系，从而有利于对文字建模。这一点相比前馈神经网络模型更接近人脑对文字的处理模型。

循环神经网络的结构如下，RNN先设隐藏层的初始状态为a(0)，然后读入第一个词x(1),预测下一个词，根据预测结果通过优化目标函数更新网络参数，然后再读入下一个词，根据这两个词预测第三个词，依次类推，不管预测第几个词，都会用到全部上文信息。



然而RNN受到短期记忆的影响。如果序列很长，他们将很难将信息从较早的时间步传送到后面的时间步。因此，如果你尝试处理一段文本进行预测，RNN可能会遗漏开头的重要信息。而 LSTM 可以改善长距离依赖的问题，不需要特别复杂的调试超参数就可以记住长期的信息。

2.2.3 长短期记忆（LSTM）

长短期记忆（Long short-term memory, LSTM）是一种特殊的RNN，主要是为了解决长依赖问题。LSTM主要分为三步：

1、通过忘记门决定需要丢弃哪些信息；

2、通过输入门决定添加哪些新的信息；

3、通过输出门得到最终该RNN单元的输出。

**二、基于循环神经网络的语言建模**

**1、数据处理**

项目用到的数据集是PTB数据集，PTB文本数据集是语言模型学习中目前最广泛的数据集，需要用到ptb.test.txt，ptb.train.txt，ptb.valid.txt三个数据文件。

PTBInput用于处理数据。定义语言模型处理输入数据的一些参数，用 reader.ptb\_producer 读取数据和标签。num\_steps是LSTM的展开步数，#epoch\_size即为每个epoch内需要多少轮训练的迭代。用reader.ptb\_producer获取特征数据input\_data，以及label数据targets。

class PTBInput(object):

def \_\_init\_\_(self, config, data, name=None):

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

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

self.input\_data, self.targets = reader.ptb\_producer(

data, batch\_size, num\_steps, name=name)

**2、语言模型**

class PTBModel(object):

1）def \_\_init\_\_(self, is\_training, config, input\_) 包括三个参数，训练标记，配置参数以及输入数据的实例;把这几个变量读取到本地，hidden\_size 是隐藏层的节点数，vocab\_size 是词汇表的大小；

def \_\_init\_\_(self, is\_training, # 训练标记

config, # 配置参数

input\_): # PTBInput类的实例input\_

self.\_input = input\_

batch\_size = input\_.batch\_size

num\_steps = input\_.num\_steps

size = config.hidden\_size # LSTM的节点数

vocab\_size = config.vocab\_size # 词汇表的大小

2）def lstm\_cell()，设定基本的 LSTM 单元，用的是tf.contrib.rnn.BasicLSTMCell；

如果 if is\_training and config.keep\_prob < 1 这个条件的话，在 LSTM 单元后面可以加一个 dropout 层；

再用 tf.contrib.rnn.MultiRNNCell 把多层的 LSTM 堆加到一起；

用 cell.zero\_state 将 LSTM 的初始状态设置为0；

def lstm\_cell(): # 定义默认的LSTM单元

return tf.contrib.rnn.BasicLSTMCell(

size, # 隐含节点数

forget\_bias=0.0, # 忘记门的bias

state\_is\_tuple=True) # 代表接受和返回的state将是2-tuple的形式

attn\_cell = lstm\_cell # 不加括号表示调用的是函数，加括号表示调用的是函数的结果

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

def attn\_cell():

return tf.contrib.rnn.DropoutWrapper(

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

)

cell = tf.contrib.rnn.MultiRNNCell(

[attn\_cell() for \_ in range(config.num\_layers)],

state\_is\_tuple=True)

self.\_initial\_state = cell.zero\_state(batch\_size, tf.float32) # 设置LSTM单元的初始化状态为0

3）接下来是 embedding 矩阵，行数是词汇表的大小，列数是每个单词的向量表达的维度，在训练过程中，它可以被优化和更新；

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

embedding = tf.get\_variable( # 初始化词向量embedding矩阵

"embedding", [vocab\_size, size], dtype=tf.float32)

inputs = tf.nn.embedding\_lookup(embedding, input\_.input\_data) # 查询单词的向量表达获得inputs

if is\_training and config.keep\_prob < 1: # 如果为向量状态则再添加一层Dropout层

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

4）接下来定义输出，限制一下反向传播时可以展开的步数，将 inputs 和 state 传到 LSTM，然后把输出结果添加到 outputs 的列表里；

然后将输出的内容串到一起，接下来 softmax 层，接着要定义损失函数 loss,加和整个 batch 的误差，再平均到每个样本的误差，并且保留最终的状态，如果不是训练状态就直接返回；

outputs = []

state = self.\_initial\_state

# 为了控制训练过程，我们会限制梯度在反向传播时可以展开的步数为一个固定的值，而这个步数也就是num\_steps

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

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

if time\_step > 0: # 第二次循环开始，使用tf.get\_variable\_scope().reuse\_variables()设置复用变量

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

# 每次循环，传入inputs和state到堆叠的LSTM单元，得到输出的cell\_output和更新后的state

# inputs有三个维度，1：batch中第几个样本，2：样本中第几个单词，3：单词的向量维度

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

state) # inputs[:, time\_step, :]表示所有样本的第time\_step个单词

outputs.append(cell\_output) # 添加到输出列表outputs

# 将output内容用tf.concat串接到一起，并使用tf.reshape将其转为一个很长的一维向量

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

# 定义权重

softmax\_w = tf.get\_variable(

"softmax\_w", [size, vocab\_size], dtype=tf.float32)

# 定义偏置

softmax\_b = tf.get\_variable(

"softmax\_b", [vocab\_size], dtype=tf.float32)

# 输出乘上权重并加上偏置得到logits，即网络最后的输出

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

# 使用如下函数计算输出logits和targets的偏差

loss = tf.contrib.legacy\_seq2seq.sequence\_loss\_by\_example(

[logits],

[tf.reshape(input\_.targets, [-1])],

[tf.ones([batch\_size \* num\_steps], dtype=tf.float32)])

self.\_cost = cost = tf.reduce\_sum(loss) / batch\_size # 汇总batch的总误差，再计算到平均每个样本的误差cost

self.\_final\_state = state # 保留最终的状态为final\_state

if not is\_training:

return

5) 接下来是定义学习速率，根据前面的 cost 计算一下梯度，并将梯度的最大范数设置好，相当于正则化的作用，可以防止梯度爆炸；

这个学习速率还可以更新，将其传入给 \_new\_lr，再执行 \_lr\_update 完成修改

self.\_lr = tf.Variable(0.0, trainable=False) # 定义学习速率并设为不可训练的

tvars = tf.trainable\_variables() # 获取全部可训练的参数

# 针对前面得到的cost，计算tvars的梯度，并用tf.clip\_by\_global\_norm设置梯度的最大范数max\_grad\_norm

# 这就是Gradient Clipping的方法，控制梯度的最大范数，防止梯度爆炸，某种程度上起到正则化的效果

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

config.max\_grad\_norm)

optimizer = tf.train.GradientDescentOptimizer(self.\_lr) # 定义优化器

# 创建训练操作\_train\_op，用optimizer.apply\_gradients将clip过的梯度应用到所有可训练参数tvars上

self.\_train\_op = optimizer.apply\_gradients(zip(grads, tvars),

global\_step=tf.contrib.framework.get\_or\_create\_global\_step()) # 生成全局统一的训练步数

# 设置一个名为\_new\_lr的placeholder用以控制学习速率

self.\_new\_lr = tf.placeholder(

tf.float32, shape=[], name="new\_learning\_rate")

self.\_lr\_update = tf.assign(self.\_lr, self.\_new\_lr) # 将\_new\_lr的值赋给当前的学习速率\_lr

# 用以外部控制学习速率

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

session.run(self.\_lr\_update, feed\_dict={self.\_new\_lr: lr\_value}) # 执行\_lr\_update操作完成对学习速率的修改

**3、模型参数设置**

小型模型：

class SmallConfig(object):

init\_scale = 0.1 # 网络中权重的初始scale

learning\_rate = 1.0 # 学习速率的初始值

max\_grad\_norm = 5 # 前面提到的梯度的最大范数

num\_layers = 2 # LSTM可以堆叠的层数

num\_steps = 20 # LSTM梯度反向传播的展开步数

hidden\_size = 200 # LSTM内隐含节点数

max\_epoch = 4 # 初始学习速率可训练的epoch数

max\_max\_epoch = 13 # 总共可训练的epoch数

keep\_prob = 1.0 # dropout保留节点的比例

lr\_decay = 0.5 # 学习速率的衰减速度

batch\_size = 20 # 每个batch中样本数

vocab\_size = 10000 # 词汇表大小

中等模型：

class MediumConfig(object):

init\_scale = 0.1 # 减小，小一些有利于温和的训练

learning\_rate = 1.0

max\_grad\_norm = 5

num\_layers = 2

num\_steps = 35 # 增大

hidden\_size = 650 # 增大

max\_epoch = 6 # 增大

max\_max\_epoch = 39 # 增大

keep\_prob = 0.5 # 减小，之前设为1即没有dropout

lr\_decay = 0.8 # 学习迭代次数增大，因此学习速率的衰减次数减小

batch\_size = 20

vocab\_size = 10000

大型模型：

class MediumConfig(object):

init\_scale = 0.04 # 减小

learning\_rate = 1.0

max\_grad\_norm = 10 # 增大

num\_layers = 2

num\_steps = 35

hidden\_size = 1500 # 增大

max\_epoch = 14 # 增大

max\_max\_epoch = 55 # 增大

keep\_prob = 0.35 # 减小，之前设为1即没有dropout

lr\_decay = 1 / 1.15 # 学习迭代次数增大，因此学习速率的衰减次数减小

batch\_size = 20

vocab\_size = 10000

测试用设置

class TestConfig(object):

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

5、训练

接着定义训练一个epoch数据的函数run\_epoch。记录当前时间，初始化损失costs和迭代数iters，并执行model.initial\_state来初始化状态并获得初始状态。

接着创建输出结果的字典表fetches，其中包括cost和final\_state，如果有评测操作eval\_op，也一并加入到fetches。

进入循环，次数为epoch\_size。每次循环中，生成训练用的feed\_dict。将全部LSTM单元的state加入feed\_dict中，然后传入feed\_dict并执行fetches对网络进行一次训练，拿到cost和state。我们每完成约10%的epoch，就进行一次结果的展示，一次展示当前的epoch的进度。最后返回perplexity作为函数结果。和训练速度最后返回perplexity作为函数结果。perplexity，是语言模型中用来比较模型性能的重要指标，越低代表模型输出的概率分布在预测样本上越好。

def run\_epoch(session, model, eval\_op=None, verbose=False):

start\_time = time.time() # 记录当前时间

costs = 0.0 # 初始化损失costs

iters = 0 # 初始化迭代数

state = session.run(model.initial\_state) # 执行初始化状态并获得初始状态

feches = { # 输出结果的字典表

"cost": model.cost,

"final\_state": model.final\_state

}

if eval\_op is not None: # 如果有评测操作，也加入feches

feches["eval\_op"] = eval\_op

for step in range(model.input.epoch\_size): # 训练循环，次数为epoch\_size

feed\_dict = {}

for i, (c, h) in enumerate(model.initial\_state): # 生成训练用的feed\_dict，将全部LSTM单元的state加入feed\_dict

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

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

# 执行feches对网络进行一次训练，得到cost和state

vals = session.run(feches, feed\_dict)

cost = vals["cost"]

state = vals["final\_state"]

costs += cost # 累加cost到costs

iters += model.input.num\_steps # 累加num\_steps到iters

if verbose and step % (model.input.epoch\_size // 10) == 10: # 每完成约10%的epoch，就进行一次结果的展示

# 一次展示当前epoch的进度，perplexity和训练速度

# perplexity(即平均cost的自然常数指数，是语言模型中用来比较模型性能的重要指标，越低表示模型输出的概率分布在预测样本上越好)

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

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

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

return np.exp(costs / iters) # 返回perplexity作为函数结果

raw\_data = reader.ptb\_raw\_data('E:\研究生上课资料及作业\语言分析与机器翻译\data') # 直接读取解压后的数据

train\_data, valid\_data, test\_data, \_ = raw\_data # 得到训练数据，验证数据和测试数据

config = SmallConfig() # 定义训练模型的配置为SmallConfig

eval\_config = SmallConfig() # 测试配置需和训练配置一致

eval\_config.batch\_size = 1 # 将测试配置的batch\_size和num\_steps修改为1

eval\_config.num\_steps = 1

# 创建默认的Graph

with tf.Graph().as\_default():

# 设置参数的初始化器，令参数范围在[-init\_scale,init\_scale]之间

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

'''使用PTBInput和PTBModel创建一个用来训练的模型m,以及用来验证的模型mvalid和测试的模型mtest，

其中训练和验证模型直接使用前面的config，测试模型使用前面的测试配置eval\_config'''

# 训练模型m

with tf.name\_scope("Train"):

train\_input = PTBInput(config=config, data=train\_data, name="TrainInput")

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

m = PTBModel(is\_training=True, config=config, input\_=train\_input)

# 验证模型mvalid

with tf.name\_scope("Valid"):

valid\_input = PTBInput(config=config, data=valid\_data, name="ValidInput")

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

mvalid = PTBModel(is\_training=False, config=config, input\_=valid\_input)

# 测试模型mtest

with tf.name\_scope("Test"):

test\_input = PTBInput(config=eval\_config, data=test\_data, name="TestInput")

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

mtest = PTBModel(is\_training=False, config=eval\_config, input\_=test\_input)

sv = tf.train.Supervisor() # 创建训练的管理器

with sv.managed\_session() as session: # 使用sv.managed\_session()创建默认session

for i in range(config.max\_max\_epoch): # 执行多个epoch数据的循环

lr\_decay = config.lr\_decay \*\* max(i + 1 - config.max\_max\_epoch, 0.0) # 计算累计的学习速率衰减值(只计算超过max\_epoch的轮数)

m.assign\_lr(session, config.learning\_rate \* lr\_decay) # 将初始学习速率呈上累计的衰减，并更新学习速率

'''循环内执行一个epoch的训练和验证并输出当前的学习速率，训练和验证集上的perplexity'''

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

train\_perplexity = run\_epoch(session, m, eval\_op=m.train\_op, verbose=True)

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

valid\_perplexity = run\_epoch(session, mvalid)

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

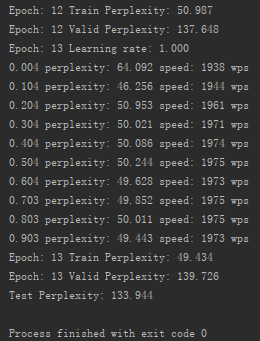
'''完成全部训练后，计算并输出模型在测试集上的perplexity'''

test\_perplexity = run\_epoch(session, mtest)

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

**三、运行结果**

运行得到SmallConfig小型模型的最后结果：



**四、总结**

通过这门课程，我接触到了自然语言处理这门学科，学习如何让机器能够理解人类的语言并进行各种灵活的应用，通过本次大作业，我也更加深刻的了解了语言模型及循环神经网络的相关知识，感谢老师幽默风趣的教导，让我受益良多。