姓名：向建宇

学号：1801210687

**找一段英文，采用Seq2Seq和Attention机制翻译为中文。**

**一、Seq2Seq**

seq2seq模型是以编码（Encode）和解码（Decode）为代表的架构方式，seq2seq模型是根据输入序列X来生成输出序列Y，在翻译，文本自动摘要和机器人自动问答以及一些回归预测任务上有着广泛的运用。以encode和decode为代表的seq2seq模型，encode意思是将输入序列转化成一个固定长度的向量，decode意思是将输入的固定长度向量解码成输出序列。其中编码解码的方式可以是RNN,CNN等。

#### 

#### 上图为seq2seq的encode和decode结构，采用CNN/LSTM模型。在RNN中，当前时间的隐藏状态是由上一时间的状态和当前时间的输入x共同决定的，即

#### 

**【编码阶段】**

　　得到各个隐藏层的输出然后汇总，生成语义向量

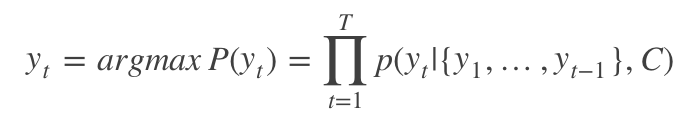
#### https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114153851702-282150839.png

#### 也可以将最后的一层隐藏层的输出作为语义向量C

#### https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114215439671-1735495691.png

【解码阶段】

这个阶段，我们要根据给定的语义向量C和输出序列y1,y2,…yt−1来预测下一个输出的单词yt，即

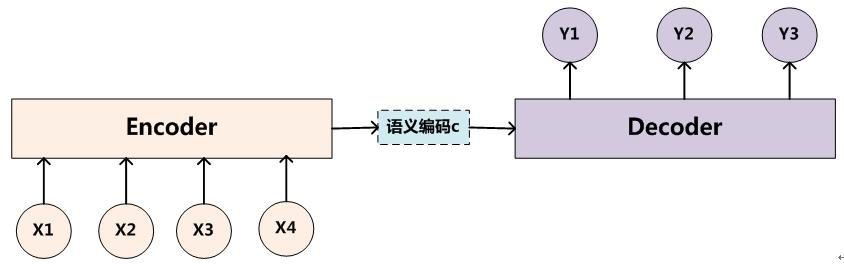


也可以写做

https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114215727640-1466593219.png

其中g（）代表的是非线性激活函数。在RNN中可写成 yt=g(yt−1,ht,C) ，其中h为隐藏层的输出。

以上就是seq2seq的编码解码阶段，seq2seq模型的抽象框架可描述为下图：



#### 二、Attention机制在seq2seq模型中的运用

由于encoder-decoder模型在编码和解码阶段始终由一个不变的语义向量C来联系着，编码器要将整个序列的信息压缩进一个固定长度的向量中去。这就造成了 （1）语义向量无法完全表示整个序列的信息，（2）最开始输入的序列容易被后输入的序列给覆盖掉，会丢失许多细节信息。在长序列上表现的尤为明显。

　　Attention模型的引入：

　　相比于之前的encoder-decoder模型，attention模型最大的区别就在于它不在要求编码器将所有输入信息都编码进一个固定长度的向量之中。相反，此时编码器需要将输入编码成一个向量的序列，而在解码的时候，每一步都会选择性的从向量序列中挑选一个子集进行进一步处理。这样，在产生每一个输出的时候，都能够做到充分利用输入序列携带的信息。而且这种方法在翻译任务中取得了非常不错的成果。

　　下图为seq2seq模型加入了Attention注意力机制

#### 

【seq2seq的attention解码过程】

 　　现在定义条件概率：

#### https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114215826546-1545822377.png

#### 上式 si表示解码器 i 时刻的隐藏状态。计算公式为：

#### https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114215916452-2068769218.png

注意这里的条件概率与每个目标输出  yi相对应的内容向量  ci有关。在sea2seq模型中，只有一个语义向量C。‘s’为隐藏层输出，相当于上面提到的h。

　　关键问题是语义向量 C 怎么得到？

　　ci是由编码时的隐藏向量序列(h1,…,hTx)按权重相加得到的。

#### https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114220010515-807350655.png

将隐藏向量序列按权重相加，表示在生成第j个输出的时候的注意力分配是不同的。αij的值越高，表示第i个输出在第j个输入上分配的注意力越多，在生成第i个输出的时候受第j个输入的影响也就越大。

　　这意味着在生成每个单词Yi的时候，原先都是相同的中间语义表示C会替换成根据当前生成单词而不断变化的Ci。理解AM模型的关键就是这里，即由固定的中间语义表示C换成了根据当前输出单词来调整成加入注意力模型的变化的Ci

　　如何得到 αij的权重值？

　　由第i-1个输出隐藏状态 si−1和输入中各个隐藏状态共同决定的，即：

#### https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114220058827-1559404546.png

#### si−1先跟每个h分别计算得到一个数值，然后使用softmax函数得到i时刻的输出在Tx个输入隐藏状态中的注意力分配向量。这个分配向量也就是计算ci的权重。

#### 

图4 显示的是Attention模型在计算αij的概率分配过程。

对于采用RNN的Decoder来说，如果要生成yi单词，在时刻i，我们是可以知道在生成Yi之前的隐层节点i时刻的输出值Hi的，而我们的目的是要计算生成Yi时的输入句子单词“Tom”、“Chase”、“Jerry”对Yi来说的注意力分配概率分布，那么可以用i时刻的隐层节点状态Hi去一一和输入句子中每个单词对应的RNN隐层节点状态hj进行对比，即通过函数F(hj,Hi)来获得目标单词Yi和每个输入单词对应的对齐可能性，这个F函数在不同论文里可能会采取不同的方法，然后函数F的输出经过Softmax进行归一化就得到了符合概率分布取值区间的注意力分配概率分布数值。图4显示的是当输出单词为“汤姆”时刻对应的输入句子单词的对齐概率。绝大多数AM模型都是采取上述的计算框架来计算注意力分配概率分布信息，区别只是在F的定义上可能有所不同。

公式汇总：

#### https://images2017.cnblogs.com/blog/1212984/201711/1212984-20171114220153140-2145437293.png

#### 三、训练数据

#### 1.参数设置。

|  |
| --- |
| # 假设输入数据已经用9.2.1小节中的方法转换成了单词编号的格式。SRC\_TRAIN\_DATA = "./train.en" # 源语言输入文件。TRG\_TRAIN\_DATA = "./train.zh" # 目标语言输入文件。CHECKPOINT\_PATH = "./seq2seq\_ckpt" # checkpoint保存路径。HIDDEN\_SIZE = 1024 # LSTM的隐藏层规模。NUM\_LAYERS = 2 # 深层循环神经网络中LSTM结构的层数。SRC\_VOCAB\_SIZE = 10000 # 源语言词汇表大小。TRG\_VOCAB\_SIZE = 4000 # 目标语言词汇表大小。BATCH\_SIZE = 100 # 训练数据batch的大小。NUM\_EPOCH = 5 # 使用训练数据的轮数。KEEP\_PROB = 0.8 # 节点不被dropout的概率。MAX\_GRAD\_NORM = 5 # 用于控制梯度膨胀的梯度大小上限。SHARE\_EMB\_AND\_SOFTMAX = True # 在Softmax层和词向量层之间共享参数。MAX\_LEN = 50 # 限定句子的最大单词数量。SOS\_ID = 1 # 目标语言词汇表中<sos>的ID。 |

#### 2.读取训练数据并创建Dataset。

|  |
| --- |
| # 使用Dataset从一个文件中读取一个语言的数据。# 数据的格式为每行一句话，单词已经转化为单词编号。def MakeDataset(file\_path):dataset = tf.data.TextLineDataset(file\_path)# 根据空格将单词编号切分开并放入一个一维向量。dataset = dataset.map(lambda string: tf.string\_split([string]).values)# 将字符串形式的单词编号转化为整数。dataset = dataset.map(lambda string: tf.string\_to\_number(string, tf.int32))# 统计每个句子的单词数量，并与句子内容一起放入Dataset中。dataset = dataset.map(lambda x: (x, tf.size(x)))return dataset# 从源语言文件src\_path和目标语言文件trg\_path中分别读取数据，并进行填充和# batching操作。def MakeSrcTrgDataset(src\_path, trg\_path, batch\_size):# 首先分别读取源语言数据和目标语言数据。src\_data = MakeDataset(src\_path)trg\_data = MakeDataset(trg\_path)# 通过zip操作将两个Dataset合并为一个Dataset。现在每个Dataset中每一项数据ds# 由4个张量组成：# ds[0][0]是源句子# ds[0][1]是源句子长度# ds[1][0]是目标句子# ds[1][1]是目标句子长度dataset = tf.data.Dataset.zip((src\_data, trg\_data))# 删除内容为空（只包含<EOS>）的句子和长度过长的句子。def FilterLength(src\_tuple, trg\_tuple):((src\_input, src\_len), (trg\_label, trg\_len)) = (src\_tuple, trg\_tuple)src\_len\_ok = tf.logical\_and(tf.greater(src\_len, 1), tf.less\_equal(src\_len, MAX\_LEN))trg\_len\_ok = tf.logical\_and(tf.greater(trg\_len, 1), tf.less\_equal(trg\_len, MAX\_LEN))return tf.logical\_and(src\_len\_ok, trg\_len\_ok)dataset = dataset.filter(FilterLength)# 从图9-5可知，解码器需要两种格式的目标句子：# 1.解码器的输入(trg\_input)，形式如同"<sos> X Y Z"# 2.解码器的目标输出(trg\_label)，形式如同"X Y Z <eos>"# 上面从文件中读到的目标句子是"X Y Z <eos>"的形式，我们需要从中生成"<sos> X Y Z"# 形式并加入到Dataset中。def MakeTrgInput(src\_tuple, trg\_tuple):((src\_input, src\_len), (trg\_label, trg\_len)) = (src\_tuple, trg\_tuple)trg\_input = tf.concat([[SOS\_ID], trg\_label[:-1]], axis=0)return ((src\_input, src\_len), (trg\_input, trg\_label, trg\_len))dataset = dataset.map(MakeTrgInput)# 随机打乱训练数据。dataset = dataset.shuffle(10000)# 规定填充后输出的数据维度。padded\_shapes = ((tf.TensorShape([None]), # 源句子是长度未知的向量tf.TensorShape([])), # 源句子长度是单个数字(tf.TensorShape([None]), # 目标句子（解码器输入）是长度未知的向量tf.TensorShape([None]), # 目标句子（解码器目标输出）是长度未知的向量tf.TensorShape([]))) # 目标句子长度是单个数字# 调用padded\_batch方法进行batching操作。batched\_dataset = dataset.padded\_batch(batch\_size, padded\_shapes)return batched\_dataset |

#### 3.定义翻译模型。

|  |
| --- |
| # 定义NMTModel类来描述模型。class NMTModel(object):# 在模型的初始化函数中定义模型要用到的变量。def \_\_init\_\_(self):# 定义编码器和解码器所使用的LSTM结构。self.enc\_cell = tf.nn.rnn\_cell.MultiRNNCell([tf.nn.rnn\_cell.BasicLSTMCell(HIDDEN\_SIZE)for \_ in range(NUM\_LAYERS)])self.dec\_cell = tf.nn.rnn\_cell.MultiRNNCell([tf.nn.rnn\_cell.BasicLSTMCell(HIDDEN\_SIZE)for \_ in range(NUM\_LAYERS)])# 为源语言和目标语言分别定义词向量。self.src\_embedding = tf.get\_variable("src\_emb", [SRC\_VOCAB\_SIZE, HIDDEN\_SIZE])self.trg\_embedding = tf.get\_variable("trg\_emb", [TRG\_VOCAB\_SIZE, HIDDEN\_SIZE])# 定义softmax层的变量if SHARE\_EMB\_AND\_SOFTMAX:self.softmax\_weight = tf.transpose(self.trg\_embedding)else:self.softmax\_weight = tf.get\_variable("weight", [HIDDEN\_SIZE, TRG\_VOCAB\_SIZE])self.softmax\_bias = tf.get\_variable("softmax\_bias", [TRG\_VOCAB\_SIZE])# 在forward函数中定义模型的前向计算图。# src\_input, src\_size, trg\_input, trg\_label, trg\_size分别是上面# MakeSrcTrgDataset函数产生的五种张量。def forward(self, src\_input, src\_size, trg\_input, trg\_label, trg\_size):batch\_size = tf.shape(src\_input)[0]# 将输入和输出单词编号转为词向量。src\_emb = tf.nn.embedding\_lookup(self.src\_embedding, src\_input)trg\_emb = tf.nn.embedding\_lookup(self.trg\_embedding, trg\_input)# 在词向量上进行dropout。src\_emb = tf.nn.dropout(src\_emb, KEEP\_PROB)trg\_emb = tf.nn.dropout(trg\_emb, KEEP\_PROB)# 使用dynamic\_rnn构造编码器。# 编码器读取源句子每个位置的词向量，输出最后一步的隐藏状态enc\_state。# 因为编码器是一个双层LSTM，因此enc\_state是一个包含两个LSTMStateTuple类# 张量的tuple，每个LSTMStateTuple对应编码器中的一层。# enc\_outputs是顶层LSTM在每一步的输出，它的维度是[batch\_size,# max\_time, HIDDEN\_SIZE]。Seq2Seq模型中不需要用到enc\_outputs，而# 后面介绍的attention模型会用到它。with tf.variable\_scope("encoder"):enc\_outputs, enc\_state = tf.nn.dynamic\_rnn(self.enc\_cell, src\_emb, src\_size, dtype=tf.float32)# 使用dyanmic\_rnn构造解码器。# 解码器读取目标句子每个位置的词向量，输出的dec\_outputs为每一步# 顶层LSTM的输出。dec\_outputs的维度是 [batch\_size, max\_time,# HIDDEN\_SIZE]。# initial\_state=enc\_state表示用编码器的输出来初始化第一步的隐藏状态。with tf.variable\_scope("decoder"):dec\_outputs, \_ = tf.nn.dynamic\_rnn(self.dec\_cell, trg\_emb, trg\_size, initial\_state=enc\_state)# 计算解码器每一步的log perplexity。这一步与语言模型代码相同。output = tf.reshape(dec\_outputs, [-1, HIDDEN\_SIZE])logits = tf.matmul(output, self.softmax\_weight) + self.softmax\_biasloss = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=tf.reshape(trg\_label, [-1]), logits=logits)# 在计算平均损失时，需要将填充位置的权重设置为0，以避免无效位置的预测干扰# 模型的训练。label\_weights = tf.sequence\_mask(trg\_size, maxlen=tf.shape(trg\_label)[1], dtype=tf.float32)label\_weights = tf.reshape(label\_weights, [-1])cost = tf.reduce\_sum(loss \* label\_weights)cost\_per\_token = cost / tf.reduce\_sum(label\_weights)# 定义反向传播操作。反向操作的实现与语言模型代码相同。trainable\_variables = tf.trainable\_variables()# 控制梯度大小，定义优化方法和训练步骤。grads = tf.gradients(cost / tf.to\_float(batch\_size),trainable\_variables)grads, \_ = tf.clip\_by\_global\_norm(grads, MAX\_GRAD\_NORM)optimizer = tf.train.GradientDescentOptimizer(learning\_rate=1.0)train\_op = optimizer.apply\_gradients(zip(grads, trainable\_variables))return cost\_per\_token, train\_op |

#### 4.训练过程和主函数。

|  |
| --- |
| # 使用给定的模型model上训练一个epoch，并返回全局步数。# 每训练200步便保存一个checkpoint。def run\_epoch(session, cost\_op, train\_op, saver, step):# 训练一个epoch。# 重复训练步骤直至遍历完Dataset中所有数据。while True:try:# 运行train\_op并计算损失值。训练数据在main()函数中以Dataset方式提供。cost, \_ = session.run([cost\_op, train\_op])if step % 10 == 0:print("After %d steps, per token cost is %.3f" % (step, cost))# 每200步保存一个checkpoint。if step % 200 == 0:saver.save(session, CHECKPOINT\_PATH, global\_step=step)step += 1except tf.errors.OutOfRangeError:breakreturn stepdef main():# 定义初始化函数。initializer = tf.random\_uniform\_initializer(-0.05, 0.05)# 定义训练用的循环神经网络模型。with tf.variable\_scope("nmt\_model", reuse=None,initializer=initializer):train\_model = NMTModel()# 定义输入数据。data = MakeSrcTrgDataset(SRC\_TRAIN\_DATA, TRG\_TRAIN\_DATA, BATCH\_SIZE)iterator = data.make\_initializable\_iterator()(src, src\_size), (trg\_input, trg\_label, trg\_size) = iterator.get\_next()# 定义前向计算图。输入数据以张量形式提供给forward函数。cost\_op, train\_op = train\_model.forward(src, src\_size, trg\_input,trg\_label, trg\_size)# 训练模型。saver = tf.train.Saver()step = 0with tf.Session() as sess:tf.global\_variables\_initializer().run()for i in range(NUM\_EPOCH):print("In iteration: %d" % (i + 1))sess.run(iterator.initializer)step = run\_epoch(sess, cost\_op, train\_op, saver, step)if \_\_name\_\_ == "\_\_main\_\_":main() |

#### 四、测试翻译

#### 1.参数设置

|  |
| --- |
| # 读取checkpoint的路径。9000表示是训练程序在第9000步保存的checkpoint。CHECKPOINT\_PATH = "./seq2seq\_ckpt-9000"# 模型参数。必须与训练时的模型参数保持一致。HIDDEN\_SIZE = 1024 # LSTM的隐藏层规模。NUM\_LAYERS = 2 # 深层循环神经网络中LSTM结构的层数。SRC\_VOCAB\_SIZE = 10000 # 源语言词汇表大小。TRG\_VOCAB\_SIZE = 4000 # 目标语言词汇表大小。SHARE\_EMB\_AND\_SOFTMAX = True # 在Softmax层和词向量层之间共享参数。# 词汇表文件SRC\_VOCAB = "./en.vocab"TRG\_VOCAB = "./zh.vocab"# 词汇表中<sos>和<eos>的ID。在解码过程中需要用<sos>作为第一步的输入，并将检查# 是否是<eos>，因此需要知道这两个符号的ID。SOS\_ID = 1EOS\_ID = 2 |

#### 2.定义NMT模型和解码步骤。

|  |
| --- |
| # 定义NMTModel类来描述模型。class NMTModel(object):# 在模型的初始化函数中定义模型要用到的变量。def \_\_init\_\_(self):# 定义编码器和解码器所使用的LSTM结构。self.enc\_cell = tf.nn.rnn\_cell.MultiRNNCell([tf.nn.rnn\_cell.BasicLSTMCell(HIDDEN\_SIZE)for \_ in range(NUM\_LAYERS)])self.dec\_cell = tf.nn.rnn\_cell.MultiRNNCell([tf.nn.rnn\_cell.BasicLSTMCell(HIDDEN\_SIZE)for \_ in range(NUM\_LAYERS)])# 为源语言和目标语言分别定义词向量。self.src\_embedding = tf.get\_variable("src\_emb", [SRC\_VOCAB\_SIZE, HIDDEN\_SIZE])self.trg\_embedding = tf.get\_variable("trg\_emb", [TRG\_VOCAB\_SIZE, HIDDEN\_SIZE])# 定义softmax层的变量if SHARE\_EMB\_AND\_SOFTMAX:self.softmax\_weight = tf.transpose(self.trg\_embedding)else:self.softmax\_weight = tf.get\_variable("weight", [HIDDEN\_SIZE, TRG\_VOCAB\_SIZE])self.softmax\_bias = tf.get\_variable("softmax\_bias", [TRG\_VOCAB\_SIZE])def inference(self, src\_input):# 虽然输入只有一个句子，但因为dynamic\_rnn要求输入是batch的形式，因此这里# 将输入句子整理为大小为1的batch。src\_size = tf.convert\_to\_tensor([len(src\_input)], dtype=tf.int32)src\_input = tf.convert\_to\_tensor([src\_input], dtype=tf.int32)src\_emb = tf.nn.embedding\_lookup(self.src\_embedding, src\_input)# 使用dynamic\_rnn构造编码器。这一步与训练时相同。with tf.variable\_scope("encoder"):enc\_outputs, enc\_state = tf.nn.dynamic\_rnn(self.enc\_cell, src\_emb, src\_size, dtype=tf.float32)# 设置解码的最大步数。这是为了避免在极端情况出现无限循环的问题。MAX\_DEC\_LEN=100with tf.variable\_scope("decoder/rnn/multi\_rnn\_cell"):# 使用一个变长的TensorArray来存储生成的句子。init\_array = tf.TensorArray(dtype=tf.int32, size=0,dynamic\_size=True, clear\_after\_read=False)# 填入第一个单词<sos>作为解码器的输入。init\_array = init\_array.write(0, SOS\_ID)# 构建初始的循环状态。循环状态包含循环神经网络的隐藏状态，保存生成句子的# TensorArray，以及记录解码步数的一个整数step。init\_loop\_var = (enc\_state, init\_array, 0)# tf.while\_loop的循环条件：# 循环直到解码器输出<eos>，或者达到最大步数为止。def continue\_loop\_condition(state, trg\_ids, step):return tf.reduce\_all(tf.logical\_and(tf.not\_equal(trg\_ids.read(step), EOS\_ID),tf.less(step, MAX\_DEC\_LEN-1)))def loop\_body(state, trg\_ids, step):# 读取最后一步输出的单词，并读取其词向量。trg\_input = [trg\_ids.read(step)]trg\_emb = tf.nn.embedding\_lookup(self.trg\_embedding,trg\_input)# 这里不使用dynamic\_rnn，而是直接调用dec\_cell向前计算一步。dec\_outputs, next\_state = self.dec\_cell.call(state=state, inputs=trg\_emb)# 计算每个可能的输出单词对应的logit，并选取logit值最大的单词作为# 这一步的而输出。output = tf.reshape(dec\_outputs, [-1, HIDDEN\_SIZE])logits = (tf.matmul(output, self.softmax\_weight)+ self.softmax\_bias)next\_id = tf.argmax(logits, axis=1, output\_type=tf.int32)# 将这一步输出的单词写入循环状态的trg\_ids中。trg\_ids = trg\_ids.write(step+1, next\_id[0])return next\_state, trg\_ids, step+1# 执行tf.while\_loop，返回最终状态。state, trg\_ids, step = tf.while\_loop(continue\_loop\_condition, loop\_body, init\_loop\_var)return trg\_ids.stack() |

#### 3.翻译一个测试句子。

|  |
| --- |
| def main():# 定义训练用的循环神经网络模型。with tf.variable\_scope("nmt\_model", reuse=None):model = NMTModel()# 定义个测试句子。test\_en\_text = "This is a test . <eos>"print(test\_en\_text)# 根据英文词汇表，将测试句子转为单词ID。with codecs.open(SRC\_VOCAB, "r", "utf-8") as f\_vocab:src\_vocab = [w.strip() for w in f\_vocab.readlines()]src\_id\_dict = dict((src\_vocab[x], x) for x in range(len(src\_vocab)))test\_en\_ids = [(src\_id\_dict[token] if token in src\_id\_dict else src\_id\_dict['<unk>'])for token in test\_en\_text.split()]print(test\_en\_ids)# 建立解码所需的计算图。output\_op = model.inference(test\_en\_ids)sess = tf.Session()saver = tf.train.Saver()saver.restore(sess, CHECKPOINT\_PATH)# 读取翻译结果。output\_ids = sess.run(output\_op)print(output\_ids)# 根据中文词汇表，将翻译结果转换为中文文字。with codecs.open(TRG\_VOCAB, "r", "utf-8") as f\_vocab:trg\_vocab = [w.strip() for w in f\_vocab.readlines()]output\_text = ''.join([trg\_vocab[x] for x in output\_ids])# 输出翻译结果。print(output\_text.encode('utf8').decode(sys.stdout.encoding))sess.close()if \_\_name\_\_ == "\_\_main\_\_":main() |

#### 五、结果截图

#### 在intel上训练了9000轮

#### 预测基本正确

#### 