使用sequence-to-sequence生成对联

本节的内容将会带领读者用神经网络来做序列生成。该技术也可以用在其他方面，例如在智能对话场景中，根据用户输入的内容作为上文，自动生成下文，我们日常使用的机器翻译，甚至于音符的生成，把有顺序的音符序列作为输入生成一段新的音乐。本节将主要是以对对联为例，对对联也可以看作是将表示上联的字序列映射到表示下联的字序列。

9.1 sequence-to-sequence原理

对于传统的DNN，只能用来解决固定维度的问题，然而对于序列问题来说长度是未知的，固定长度成了其最关键的限制。本节将会介绍一种seq2seq的结构来解决序列生成的问题，该结构分为encoder与decoder两个部分，encoder用于编码，提取输入序列的上下文信息，decoder用于解码，把上下文信息解析出来并生成新的序列。

本节提到的seq2seq并没有采用RNN而是基于Long Short-Term Memory（LSTM）的，对于序列问题来说输入和相应输出之间存在相当大的时间延迟，而LSTM能有效的长期保存上文信息，让生成的序列不仅仅只关注当前内容。注意，encoder阶段读取的输入数据需要对序列做一个反向操作，从序列的尾部开始读取数据，这个小技巧能让输入的值更接近输出值，对于该模型的效果来说有很重要的意义，例如输入“ABC”生成“XYZ”，那么此时应该按顺序输入的是“CBA”。图8.1即seq2seq的结构，标记位<EOS>表示的是end-of-sentence，即当遇到<EOS>标记位时即会停止生成新的序列。

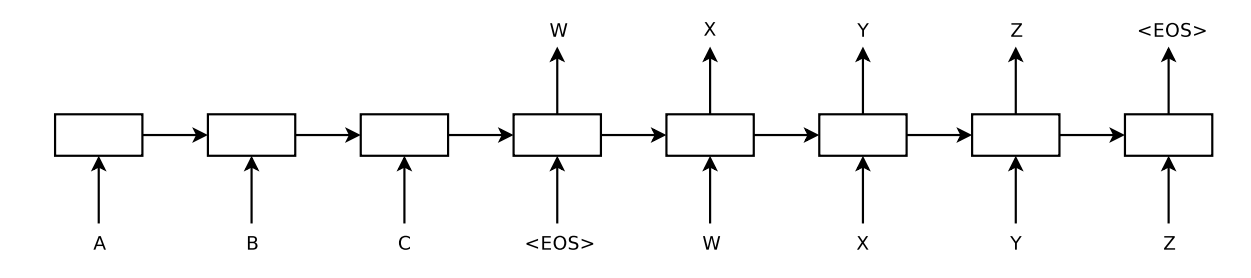


图 8.1 seq2seq结构

对于该模型来说，其损失函数其实就是一个条件概率 其中 是输入序列， 是生成序列，输入序列的长度与生成序列的长度可能是不一样的。encdoer首先根据输入序列计算出encoder最后一个LSTM的hidden state，然后把这个hidden state作为decoder阶段的第一个LSTM的初始化hidden state，所以其最终的损失函数为：

在这个等式中，每一个 值表示就是词典中每一个字的概率分布，因此我们要做的就是最小化该值。

9.2 使用keras实现sequence-to-sequence

本节将会采用keras来搭建一个seq2seq的模型，并以对对联来作为验证，其中输入序列是上联，由模型来自动生成下联。

数据包括三部分，训练数据，验证数据与词典。词典中除了常用的字与标点符号以外，还有上文提到的<EOS>，这里为了区分输入序列与输出序列，引入了两个标记位<EOS>与</EOS>。

首先读取词典，初始化一个字典，key是字，value是字在字典中的index。

|  |
| --- |
| voc = {} with open('couplet/vocabs', encoding='utf-8')as vocab:  for index, line in enumerate(vocab.readlines()):  voc[line.strip()] = index |

定义下文会用到的超参数，batch\_size表示的是每一轮训练数据的个数，epochs表示的是训练多少轮，latent\_dim表示的是LSTM输出的维度，max\_encoder\_seq\_length是上联的最大长度，max\_decoder\_seq\_length是下联的最大长度，因为需要加上两个标记位，所以下联的最大长度是上联的最大长度加2。

|  |
| --- |
| batch\_size = 128  epochs = 50  latent\_dim = 256  max\_encoder\_seq\_length = 7 max\_decoder\_seq\_length = 9 |

接下来读取输入数据，这里的训练数据一共有接近50万条，如果一次性把数据加载到内存可能会出现内存不够的问题，因此我们这里采用generator的方法，每次只读取一个batch\_size的数据。这里的count是用来计量读取的数据条数，每当读取的数据条数等于batch\_size的时候，就会把当前读取的数据yield即generator每一轮获得的数据。encoder\_input\_data表示的是encoder输入序列的one-hot编码，decoder\_input\_data表示的是decoder输入序列的one-hot编码，decoder\_target\_data表示的是decoder输出序列的one-hot编码，读者也可以自行将此处的one-hot编码替换成效果更好的word embedding。

|  |
| --- |
| def get\_data():  while True:  data = pd.read\_csv('couplet/train.csv')  count = 0  for line in data.values:  X1 = []  X2 = []  y = []  count += 1  encoder\_input\_data= np.zeros((max\_encoder\_seq\_length,  len(voc)), dtype='float16')  decoder\_input\_data = np.zeros((max\_decoder\_seq\_length,  len(voc)), dtype='float16')  decoder\_target\_data = np.zeros((max\_decoder\_seq\_length,  len(voc)), dtype='float16')  for t, char in enumerate(str(line[0]).split()[::-1]):  encoder\_input\_data[t, voc[char]] = 1  for t, char in enumerate(str('<EOS> ' + line[1] + ' </EOS>').split()):  decoder\_input\_data[t, voc[char]] = 1  if t > 0:  decoder\_target\_data[t - 1, voc[char]] = 1   X1.append(encoder\_input\_data)  X2.append(decoder\_input\_data)  y.append(decoder\_target\_data)   if count == batch\_size:  count = 0  yield ({'input\_1': np.array(X1),  'input\_2': np.array(X2)},  {'dense\_1': np.array(y)}) |

先看encoder，首先确认输入的张量的维度，其中None表示的任意长度，由于我们采用的是one-hot编码所以第二个维度是len(voc)，LSTM的参数return\_state表示的是是否返回隐层的状态，这里设置为True，最后保存好隐层的状态即encoder\_states。

|  |
| --- |
| encoder\_inputs = Input(shape=(None, len(voc))) encoder\_outputs, state\_h, state\_c = LSTM(latent\_dim, return\_state=True)(encoder\_inputs) encoder\_states = [state\_h, state\_c] |

decoder和encoder类似，首先定义decoder的输入维度，LSTM多了一个参数return\_sequences，该参数表示的是是否返回整个序列，若为True则返回整个序列，否则仅返回输出序列的最后一个输出，此处需要输出每一个输出序列，因此设置为True，LSTM的输入值除了decoder\_inputs外还包括encoder阶段的initial\_state，最后把LSTM的输出值经过一层全连接层，并添加softmax激活函数来得到最后的每个字的概率值

|  |
| --- |
| decoder\_inputs = Input(shape=(None, len(voc))) decoder\_lstm = LSTM(latent\_dim, return\_sequences=True, return\_state=True) decoder\_outputs, \_, \_ = decoder\_lstm(decoder\_inputs,  initial\_state=encoder\_states) decoder\_dense = Dense(len(voc), activation='softmax') decoder\_outputs = decoder\_dense(decoder\_outputs) |

模型由于存在两个输入，因此无法使用Sequential API，这里采用functional API得到model，设置优化方法为adam，损失函数是categorical\_crossentropy，这里要特别注意，由于数据是采用generator生成的，因此不能使用fit方法而是是用fit\_generator方法，第一个参数是一个generator，也即上文定义的get\_data方法，steps\_per\_epoch参数表示的是要学习多少个batch\_size，该参数的值一般等于总的数据条数除以batch\_size的大小，这里总的数据量是50多万条，每个batch\_size是128，因此该处取了4096这个值，epochs即训练的轮数。

|  |
| --- |
| model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)  model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['acc']) model.fit\_generator(get\_data(batch\_size),  steps\_per\_epoch=4096,  epochs=epochs) |

模型的训练阶段到此即全部完成，接下来是推断阶段。训练阶段的模型encoder与decoder是连接在一起的，推断阶段此处把encoder与decoder区分开，两者的内部结构是一样。依旧采用functional API，根据训练阶段或得到的输入输出定义出encoder的模型并保存下来。

|  |
| --- |
| encoder\_model = Model(encoder\_inputs, encoder\_states)  encoder\_model.save('encoder\_model.h5') |

然后是decoder模型，首先定义输入值即LSTM的hidden state，然后定义decoder阶段的LSTM，并把LSTM的输出值作为全连接层的输入值并得到最终的结果，根据decoder的输入输出值得到decoder阶段的model，并保存下来。

|  |
| --- |
| decoder\_state\_input\_h = Input(shape=(latent\_dim,)) decoder\_state\_input\_c = Input(shape=(latent\_dim,)) decoder\_states\_inputs = [decoder\_state\_input\_h, decoder\_state\_input\_c]  decoder\_outputs, state\_h, state\_c = decoder\_lstm(decoder\_inputs,  initial\_state=decoder\_states\_inputs) decoder\_states = [state\_h, state\_c] decoder\_outputs = decoder\_dense(decoder\_outputs)  decoder\_model = Model(  [decoder\_inputs] + decoder\_states\_inputs,  [decoder\_outputs] + decoder\_states)  decoder\_model.save('decoder\_model.h5') |

模型准备就绪之后即可进行推断了，如果模型做了持久化，可调用load\_model方法读取模型。reverse\_target\_char\_index是词典的一个index到字的映射，和上文的voc的映射刚好相反，目的是为了根据decoder输出的index在词典中找到相应的字。

|  |
| --- |
| encoder\_model = load\_model('encoder\_model.h5') decoder\_model = load\_model('decoder\_model.h5')  reverse\_target\_char\_index = dict((i, char) for char, i in voc.items()) |

推断的主体逻辑如下，首先对输入句子进行one-hot编码，根据encoder得到hidden state，初始化target\_seq为<EOS>的one-hot编码，并作为decoder的第一个节点的输入值，然后开启一个循环，循环的结束条件是是生成的序列长度大于等于输入序列的长度，循环的内部首先根据decoder的输出值，在reverse\_target\_char\_index中找出对应的字，拼接成新的输出序列，并把该字的one-hot编码与上个节点输出的hidden state作为下一个节点的输入值，以此类推，直到循环结束。

|  |
| --- |
| def decode\_sequence(sentence):  input\_seq = np.zeros((1, 32, len(voc)), dtype='float32')  seq\_len = len(sentence)  for index, word in enumerate(sentence[::-1]):  input\_seq[:, index, voc[word]] = 1  states\_value = encoder\_model.predict(input\_seq)   target\_seq = np.zeros((1, 1, len(voc)))  target\_seq[0, 0, voc['<EOS>']] = 1.   stop\_condition = False  decoded\_sentence = ''  while not stop\_condition:  output\_tokens, h, c = decoder\_model.predict(  [target\_seq] + states\_value)   sampled\_token\_index = np.argmax(output\_tokens[0, -1, :])  sampled\_char = reverse\_target\_char\_index[sampled\_token\_index]  decoded\_sentence += sampled\_char   if len(decoded\_sentence) >= seq\_len:  stop\_condition = True   target\_seq = np.zeros((1, 1, len(voc)))  target\_seq[0, 0, sampled\_token\_index] = 1.   states\_value = [h, c]   return decoded\_sentence |

9.3 小结

本节主要讲解了sequence-to-sequence原理，采用Keras实现了其模型并在对对联这个例子上进行了实验，其关键在于构建编码器和解码器，并且要认识到训练阶段与推断阶段工作的流程是不一样的。对于文本序列来说，输入可以是字级的，也可以是单词级的，需要根据具体的场景来确定，读者也可以根据单词级来对上文的代码进行修改。