用 TensorFlow 实现智能机器人的原理及如何实现一个对话机器人

作者/分享人：[李嘉璇](http://gitbook.cn/m/mazi/author/58874614cfe9f7c43f52d2ff)

[查看本场Chat](http://gitbook.cn/m/mazi/activity/592b116672cd3248715a03ac)

第一部分讲解自然语言处理的原理，第二部分讲解聊天机器人的实现原理，解决方案及挑战，最后以seq2seq+Attention机制讲解模型结构。第三部分讲解如何从0开始训练一个聊天机器人。

一、自然语言处理的原理

人工智能理解自然语言的原理，要回答这个问题，首先需要界定下这个问题的含义，一种是狭义的说如何用计算机来处理和分析自然语言；另一种是则是广义地理解关于“人工智能”、“自然语言”和“理解”的含义。

我们先从广义层面探讨。弄清楚这几个名词。

自然语言：就是人类社会中发明和演变的用于沟通和交流的语言。而人工智能在对事物（不仅仅是自然语言）的理解，往往包含两个层次：一是研究内容；二是方法论。研究内容上主要是现在流行的研究课题，例如知识图谱、CV、语音识别、NLP等。方法论是指实现人工智能的方法，主要有三种：符号主义、联结主义、行为主义。符号主义是用数理逻辑的推理来模拟人的思维智能，例如专家系统的实现。联结主义对人脑的仿生学研究，最常见的就是神经网络模型。行为主义重点在可预测的人类行为上，认为人类通过与外界环境的交互而得到自适应性，涉及的算法有遗传算法、强化学习等。

现有的NLP主要是以规则和统计相结合来处理的。它的规则一面偏向于符号主义的视角；而统计一面偏向于挖掘一般规律，属于归纳，目前用的方法，比如将自然语言用词向量的方法表征，然后接入神经网络中进行训练，也就是联结主义的思想。

理解：关于机器是否能真正理解语言一直有争论。先抛开这个问题，我们看看人类对语言的理解是怎么样的。实际上，人类对理解这个事情也做的不一定好。比如，南北方对“豆腐脑”的认知是不同的，两人交谈可能就会对同一物体的理解不同。因此，理解是需要由相似的生活经历、共同话题、上下文、会话的环境、双方的知识等很多因素决定的。既然对于人类来说，真正能理解对方，需要这么多的隐性因素，那对于机器来说，我们最好就不要关心机器是否真正能理解问题的含义本身，而是尽可能地让机器关注上述因素，来模拟人的智能。

狭义的层面是我们工程师研究的主要方向。

也就是将自然语言理解看成是用计算机来处理和分析自然语言，它涉及到语言学（词、词性、语法）和计算机学科（模型/算法）的范畴。

从语言学上来看，研究的方向包括词干提取、词性还原、分词、词性标注、命名实体识别、词性消歧、句法分析、篇章分析等等。这属于研究的基础范畴，在这些基础的研究内容之上，面向的是具体的文本处理应用，如，机器翻译、文本摘要、情感分类、问答系统、聊天机器人等。

在计算机算法的研究方面，一般是以规则和统计相结合的方法，也就是理性主义和经验主义相结合。自然语言本质上还是符号系统，因此有一定的规则可寻，但是它的复杂性又决定了没有规则可以既不相互冲突又能覆盖所有的语言现象。后来大规模语料库的完善和统计机器学习方法流行起来后，就省去了很多人工编制规则的负担，使模型生成自动生成特征。

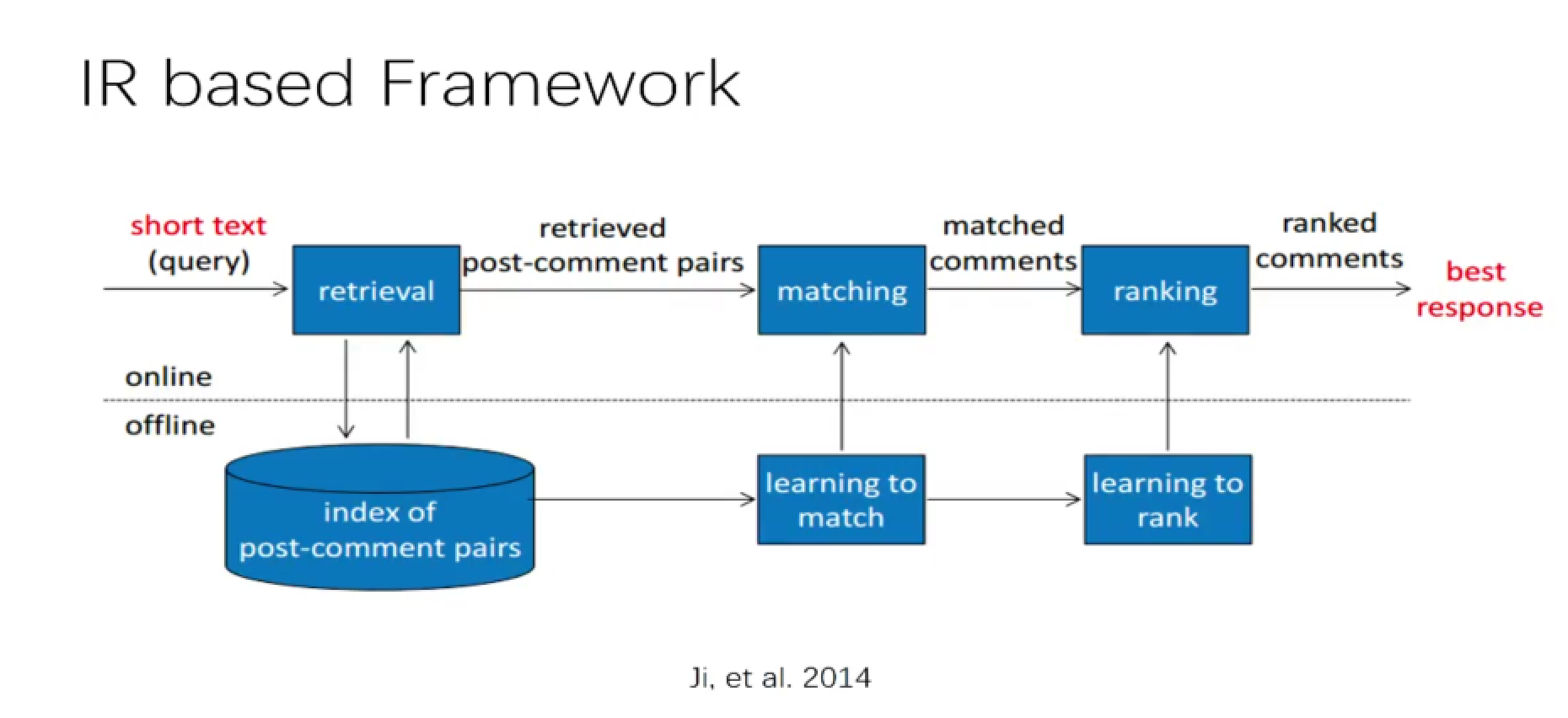
所以，我们研究的NLP就是使用数理和机器学习的方法对语言进行建模。可以说，NLP不是达到真正的自然语言理解，而是把语言当成是一种计算任务。

二、聊天机器人的实现原理，解决方案及挑战

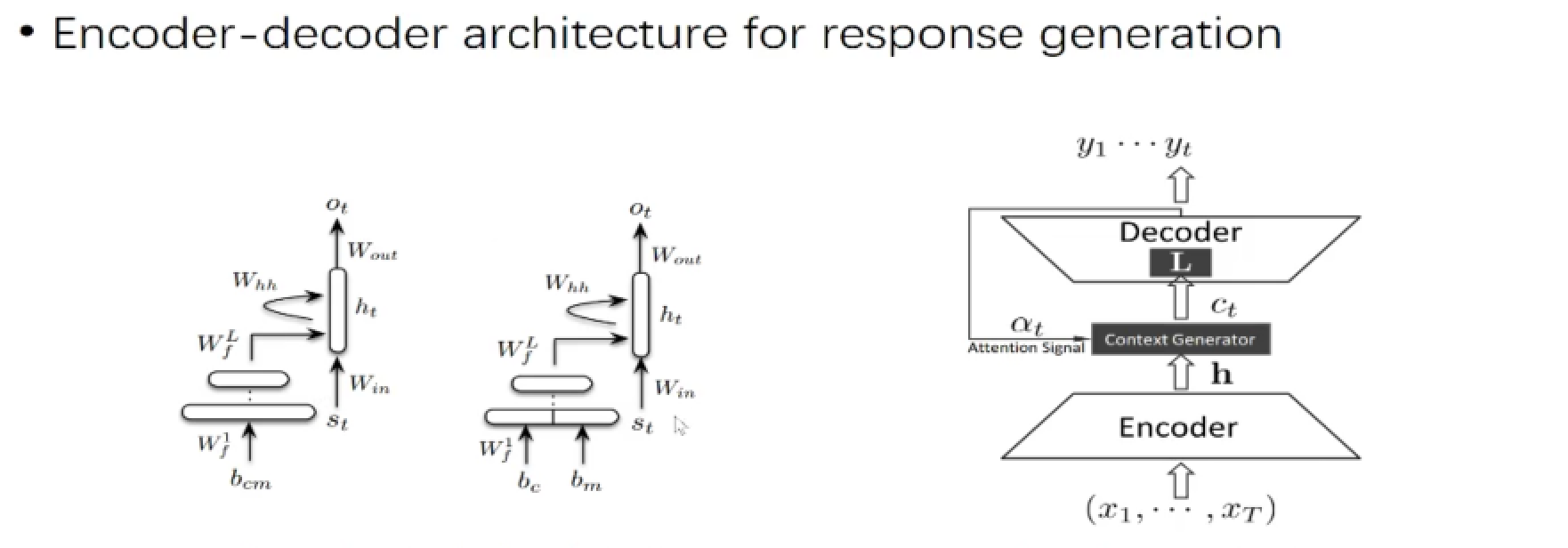
我们从聊天机器人的分类和实现原理分别说起。

目前聊天机器人根据对话的产生方式，可以分为基于检索的模型（Retrieval-Based Models）和生成式模型（Generative Models）。

基于检索的模型有一个预先定义的回答集，我们需要设计一些启发式规则，这些规则能够根据输入的问句及上下文，挑选出合适的回答。



生成式模型不依赖预先定义的回答集，而是根据输入的问句及上下文，产生一个新的回答。



聊天机器人的这两条技术路线，从长远的角度看目前技术还都还处在山底，两种技术路线的异同和优势如下：

基于检索的模型的优势：

* 答句可读性好
* 答句多样性强
* 出现不相关的答句，容易分析、定位bug

但是它的劣势在于：需要对候选的结果做排序，进行选择

基于生成式模型的优势：

* 端到端的训练，比较容易实现
* 避免维护一个大的Q-A数据集
* 不需要对每一个模块额外进行调优，避免了各个模块之间的误差级联效应

但是它的劣势在于：难以保证生成的结果是可读的，多样的。

因此，上述方法共同面临的挑战有：

1. 如何利用前几轮对话的信息，应用到当轮对话当中
2. 合并现有的知识库的内容进来
3. 能否做到个性化，千人千面。

这有点类似于我们的信息检索系统，既希望在垂直领域做得更好；也希望对不同的人的query有不同的排序偏好。

从应用目的的角度区分，可以分为目标驱动（Goal Driven），应用于客服助理等，在封闭话题场景中；无目标驱动（Non-Goal Driven），应用在开放话题的场景下，这是可谈论的主题是不限的，但是需要机器人有一定的基础常识。

尽管目前工业界应用的大多数是基于检索的模型，属于目标驱动的，例如：阿里小蜜，应用的话题领域比较窄，稍微将话题扩大一点，它就会不着边际回复或者文不对题。如下图：



一个开放话题场景下的生成式模型应该是最智能、符合我们预期的聊天机器人。因此总结来看:

智能聊天机器人的目标：

1. 和人类能够进行持续的沟通
2. 对不同的提问能够给出合适的回答
3. 考虑到人类不同个性化的差异性，给出差异性的回答（例如，同一个问题，对男女老少不同群体的回答应该略有差异）

那么对于一个智能机器人来说，它的聊天的功能在其中应该处于什么位置？首先，聊天应该是一个基础模块；其次，聊天应该和完成任务的模块有很好的协作；最后，聊天应该使机器人看上去像您的朋友，而不是您的代理或者助手。

从上述角度来说，现在有一些经常与聊天机器人混淆的概念，也是一些聊天系统的周边产品：

1. QA 问答系统：是回答事实问题（例如珠峰有多高）以及非事实问题（例如why, how, opinion等观点性问题）的领域机器人。
2. Dialog system 对话系统：这种大多是目标驱动的，但是近几年都也在慢慢接受聊天机器人功能
3. Online customer service 在线客服：例如淘宝小蜜，它在多数情况下像一个自动的FAQ。

因此，尽管聊天系统都是针对文本理解的大方向，但目标不同决定了技术路线会有所偏重，但聊天功能是一个基础功能。

智能聊天机器人可以从上面的周边系统研究领域的数据集有所借鉴：

* 非事实问题的问答
* 社区型问答系统（例如百度知道等，对问题和答案间有较强的匹配；并且一个问题多个答案时有评分、排序）
* 从在线系统中挖掘一些好的QA corpus

那如何来评价一个聊天机器人的好坏？最重要的是问句和答句的相关性，也就是本质是：短文本相关度计算。但要注意：

1. 相似性和相关性是不同的。用于相似性计算的各种方法并不适用于相关性。我们需要建立一套短文本相关性计算方法。
2. 相关性计算有一些在早期的聊天机器人的构建中延续下来的方法：

* 词语共现的统计
* 基于机器翻译的相关度计算
* 主题模型（LDA)的相似度计算

目前在聊天机器人上使用的深度学习方法有如下这些：

1. Word2vec, Glove
2. CNN, LSTM, GRU
3. Seq2Seq
4. Attention mechanism
5. Deep Reinforcement Learning

其中，深度强化学习是把“如何得到一个足够让人满意的生成式结果”量化成Reward。这些因素包括：

* 是否容易被回答（一个好的生成式的结果应该是更容易让人接下去的）
* 一个好的回复应该让对话前进，避免重复性
* 保证语义的连贯性。用生成式的结果反推回query的可能性有多大，能保证在语义上是和谐的、连贯的。

那么如何实现在智能聊天机器人的第三个目标，差异化回答呢？也就是如何在对话里如何引入个性化（personality）信息？

首先预训练一个人的向量，让生成的结果贴近人的特点。通过word embedding将人的信息作为一个词向量，在输入部分增加一组人的向量；在生成回答的时候，考虑不同的人应该选什么样的词语。

目前，我们要做一个真正智能的自动聊天机器人，仍然面临一些挑战：

* 缺乏公共的训练数据集，目前使用国外的数据集较多
  + Ubuntu dialog corpus（subset of Ubuntu Corpus）
  + Reddit dataset（可读性和质量都不错）
  + Corpora from Chinese SNS(微博）
* 测试集还不统一
* 评估度量：度量很难设计（目前是从机器翻译和自动摘要的BLEU、ROUGE里借鉴的，但面临问题是否能刻画聊天机器人的好坏，并且指导聊天机器人的技术朝着正向的方向发展）
* 聊天机器人的一般对话和任务导向的对话之间如何能够平滑切换
  + 这种平滑切换对用户的体验非常重要
  + 切换的技术需要依赖情绪分析及上下文分析
  + 用户不需要给出明确的反馈。例如，我前一句说鹿晗好，后一句说不喜欢韩范，需要聊天机器人能正确识别
* 仍然存在的问题
  + 句子级、片段级的语义建模还没有词语的建模（word embedding）那么好
  + 生成式仍然会产生一些安全回答
  + 在知识的表示和引入上还需要努力

最后有一些可以期待的结论和前景：

可以认为智能聊天机器人是一个ALL-NLP问题，深度学习是促进了chatbot的发展，但还不够，需要和传统机器学习相结合。目前我们仍在探索一些方法论及测试的评价体系。并且希望聊天机器人能够主动找人说话，例如它说“你昨天跟我说什么……你生日快到啦……”等等，这个可以用规则实现，需要把问题定义清楚。个性化的聊天机器人的实现，着力点可能更偏工程。聊天机器人的研究可能在2015年左右才开始蓬勃兴起，所以大家有很大的机会和挑战。

一些聊天机器人的参考文献：

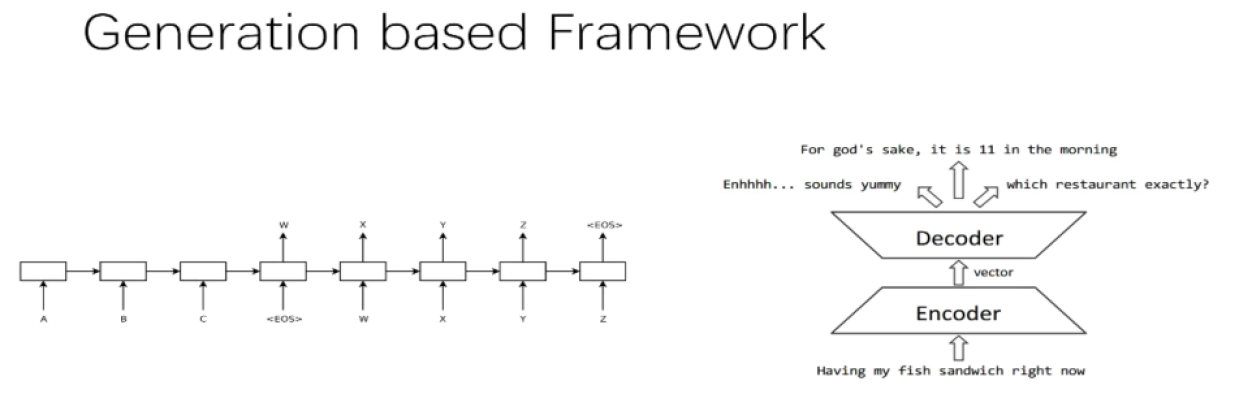
* Neural Responding Machine for Short-Text Conversation (2015-03)
* A Neural Conversational Model (2015-06)
* A Neural Network Approach to Context-Sensitive Generation of Conversational Responses (2015-06)
* The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue - Systems (2015-06)
* Building End-To-End Dialogue Systems Using Generative Hierarchical Neural Network Models (2015-07)
* A Diversity-Promoting Objective Function for Neural Conversation Models (2015-10)
* Attention with Intention for a Neural Network Conversation Model (2015-10)
* Improved Deep Learning Baselines for Ubuntu Corpus Dialogs (2015-10)
* A Survey of Available Corpora for Building Data-Driven Dialogue Systems (2015-12)
* Incorporating Copying Mechanism in Sequence-to-Sequence Learning (2016-03)
* A Persona-Based Neural Conversation Model (2016-03)
* How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation (2016-03)

下面是讲解智能聊天机器人的Seq2Seq+Attention机制。

而生成式模型现在主要研究方法是Sequence to Sequence+Attention的模型，以及最新出的GANSeq模型。（参考论文：Sequence to Sequence Learning with Neural Networks）

seq2seq模型是一个翻译模型，主要是把一个序列翻译成另一个序列。它的基本思想是用两个RNNLM，一个作为编码器，另一个作为解码器，组成RNN编码器-解码器。

在文本处理领域，我们常用编码器-解码器（encoder-decoder）框架。如下图：

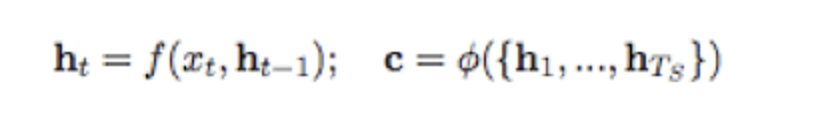


在这个模型中，每一时间的输入和输出是不同的。我们的目的是把现有序列ABC作为输入，映射成WXYZ作为输出。编码器对输入的ABC进行编码，生成中间语义编码C，然后解码器对中间语义编码C进行解码，在每个i时刻，结合已经生成的WXY的历史信息生成Z。但是，这个框架有一个缺点，就是生成的句子中每一个词采用的中间语义编码是相同的，都是C。因此，在句子比较短的时候，还能比较贴切，句子长时，就明显不合语义了。

它的过程分为这两步：

**（1）编码（Encode）**

ht是由当前的输入xt和上一次的隐藏层的输出ht-1，经过非线性变换得到的。向量c通常为RNN中的最后一个隐节点（h，Hidden state），或是多个隐节点的加权总和（在注意力机制里）。

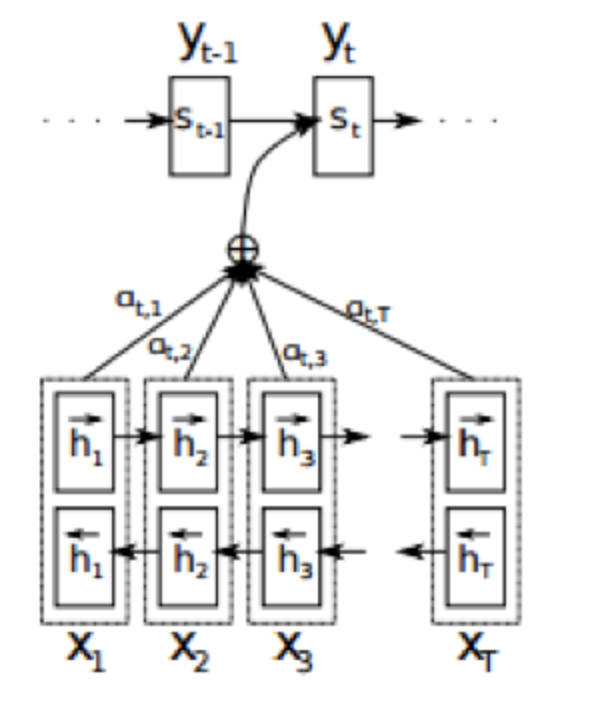


**（2）解码（Decode）**

解码的过程常常使用贪心算法或者集束搜索，来返回对应概率最大的词汇。

在实际实现聊天系统的时候，一般编码器和解码器都采用RNN模型以及RNN模型的改进模型LSTM。当句子长度超过30以后，LSTM模型的效果会急剧下降，一般此时会引入Attention模型，对长句子来说能够明显提升系统效果。

Attention机制是认知心理学层面的一个概念，它是指当人在做一件事情的时候，会专注地做这件事而忽略周围的其他事。例如，人在专注地看这本书，会忽略旁边人说话的声音。这种机制应用在聊天机器人、机器翻译等领域，就把源句子中对生成句子重要的关键词的权重提高，产生出更准确的应答。 增加了Attention模型的编码器-解码器框架如下图所示。Seq2Seq中的编码器被替换为一个双向循环网络（bidirectional RNN），源序列x＝(x1,x2,…,xt)分别被正向与反向地输入了模型中，进而得到了正反两层隐节点，语境向量c则由RNN中的隐节点h通过不同的权重a加权而成。



三、动手实现一个智能机器人

本次代码基于TF 1.1版本实现，使用双向LSTM+Attention模型。分为6步：

1）库引入及超参数定义

**import** numpy **as** np #matrix math

**import** tensorflow **as** tf #machine learningt

**import** helpers #for formatting data into batches and generating random sequence data

tf.reset\_default\_graph() #Clears the default graph stack and resets the global default graph.

sess = tf.InteractiveSession()

PAD = 0

EOS = 1

vocab\_size = 10

input\_embedding\_size = 20 #character length

encoder\_hidden\_units = 20 #num neurons

decoder\_hidden\_units = encoder\_hidden\_units \* 2

#input placehodlers

encoder\_inputs = tf.placeholder(shape=(**None**, **None**), dtype=tf.int32, name='encoder\_inputs')

encoder\_inputs\_length = tf.placeholder(shape=(**None**,), dtype=tf.int32, name='encoder\_inputs\_length')

decoder\_targets = tf.placeholder(shape=(**None**, **None**), dtype=tf.int32, name='decoder\_targets')

2）输入文本的向量表示

embeddings = tf.Variable(tf.random\_uniform([vocab\_size, input\_embedding\_size], -1.0, 1.0), dtype=tf.float32)

#this thing could get huge in a real world application

encoder\_inputs\_embedded = tf.nn.embedding\_lookup(embeddings, encoder\_inputs)

3）Encoder

from tensorflow.contrib.rnn import LSTMCell, LSTMStateTuple

encoder\_cell = LSTMCell(encoder\_hidden\_units)

((encoder\_fw\_outputs,

encoder\_bw\_outputs),

(encoder\_fw\_final\_state,

encoder\_bw\_final\_state)) = (

tf.nn.bidirectional\_dynamic\_rnn(cell\_fw=encoder\_cell,

cell\_bw=encoder\_cell,

inputs=encoder\_inputs\_embedded,

sequence\_length=encoder\_inputs\_length,

dtype=tf.float64, time\_major=True)

)

encoder\_outputs = tf.concat((encoder\_fw\_outputs, encoder\_bw\_outputs), 2)

encoder\_final\_state\_c = tf.concat(

(encoder\_fw\_final\_state.c, encoder\_bw\_final\_state.c), 1)

encoder\_final\_state\_h = tf.concat(

(encoder\_fw\_final\_state.h, encoder\_bw\_final\_state.h), 1)

encoder\_final\_state = LSTMStateTuple(

c=encoder\_final\_state\_c,

h=encoder\_final\_state\_h

)

4）Decoder

decoder\_cell = LSTMCell(decoder\_hidden\_units)

encoder\_max\_time, batch\_size = tf.unstack(tf.shape(encoder\_inputs))

decoder\_lengths = encoder\_inputs\_length + 3

#weights

W = tf.Variable(tf.random\_uniform([decoder\_hidden\_units, vocab\_size], -1, 1), dtype=tf.float32)

#bias

b = tf.Variable(tf.zeros([vocab\_size]), dtype=tf.float32)

**assert** EOS == 1 **and** PAD == 0

eos\_time\_slice = tf.ones([batch\_size], dtype=tf.int32, name='EOS')

pad\_time\_slice = tf.zeros([batch\_size], dtype=tf.int32, name='PAD')

#retrieves rows of the params tensor. The behavior is similar to using indexing with arrays in numpy

eos\_step\_embedded = tf.nn.embedding\_lookup(embeddings, eos\_time\_slice)

pad\_step\_embedded = tf.nn.embedding\_lookup(embeddings, pad\_time\_slice)

**def** **loop\_fn\_initial**():

initial\_elements\_finished = (0 >= decoder\_lengths) # all False at the initial step

#end of sentence

initial\_input = eos\_step\_embedded

#last time steps cell state

initial\_cell\_state = encoder\_final\_state

#none

initial\_cell\_output = **None**

#none

initial\_loop\_state = **None** # we don't need to pass any additional information

**return** (initial\_elements\_finished,

initial\_input,

initial\_cell\_state,

initial\_cell\_output,

initial\_loop\_state)

**def** **loop\_fn\_transition**(time, previous\_output, previous\_state, previous\_loop\_state):

**def** **get\_next\_input**():

output\_logits = tf.add(tf.matmul(previous\_output, W), b)

#Returns the index with the largest value across axes of a tensor.

prediction = tf.argmax(output\_logits, axis=1)

#embed prediction for the next input

next\_input = tf.nn.embedding\_lookup(embeddings, prediction)

**return** next\_input

elements\_finished = (time >= decoder\_lengths) # this operation produces boolean tensor of [batch\_size]

# defining if corresponding sequence has ended

#Computes the "logical and" of elements across dimensions of a tensor.

finished = tf.reduce\_all(elements\_finished) # -> boolean scalar

#Return either fn1() or fn2() based on the boolean predicate pred.

input = tf.cond(finished, **lambda**: pad\_step\_embedded, get\_next\_input)

#set previous to current

state = previous\_state

output = previous\_output

loop\_state = **None**

**return** (elements\_finished,

input,

state,

output,

loop\_state)

**def** **loop\_fn**(time, previous\_output, previous\_state, previous\_loop\_state):

**if** previous\_state **is** **None**: # time == 0

**assert** previous\_output **is** **None** **and** previous\_state **is** **None**

**return** loop\_fn\_initial()

**else**:

**return** loop\_fn\_transition(time, previous\_output, previous\_state, previous\_loop\_state)

decoder\_outputs\_ta, decoder\_final\_state, \_ = tf.nn.raw\_rnn(decoder\_cell, loop\_fn)

decoder\_outputs = decoder\_outputs\_ta.stack()

decoder\_max\_steps, decoder\_batch\_size, decoder\_dim = tf.unstack(tf.shape(decoder\_outputs))

#flettened output tensor

decoder\_outputs\_flat = tf.reshape(decoder\_outputs, (-1, decoder\_dim))

#pass flattened tensor through decoder

decoder\_logits\_flat = tf.add(tf.matmul(decoder\_outputs\_flat, W), b)

#prediction vals

decoder\_logits = tf.reshape(decoder\_logits\_flat, (decoder\_max\_steps, decoder\_batch\_size, vocab\_size))

decoder\_prediction = tf.argmax(decoder\_logits, 2)

5）Optimizer

stepwise\_cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(

labels=tf.one\_hot(decoder\_targets, depth=vocab\_size, dtype=tf.float32),

logits=decoder\_logits,

)

#loss function

loss = tf.reduce\_mean(stepwise\_cross\_entropy)

#train it

train\_op = tf.train.AdamOptimizer().minimize(loss)

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

6）Training on the toy task

batch\_size = 100

batches = helpers.random\_sequences(length\_from=3, length\_to=8,

vocab\_lower=2, vocab\_upper=10,

batch\_size=batch\_size)

print('head of the batch:')

**for** seq **in** next(batches)[:10]:

print(seq)

**def** **next\_feed**():

batch = next(batches)

encoder\_inputs\_, encoder\_input\_lengths\_ = helpers.batch(batch)

decoder\_targets\_, \_ = helpers.batch(

[(sequence) + [EOS] + [PAD] \* 2 **for** sequence **in** batch]

)

**return** {

encoder\_inputs: encoder\_inputs\_,

encoder\_inputs\_length: encoder\_input\_lengths\_,

decoder\_targets: decoder\_targets\_,

}

loss\_track = []

max\_batches = 3001

batches\_in\_epoch = 1000

**try**:

**for** batch **in** range(max\_batches):

fd = next\_feed()

\_, l = sess.run([train\_op, loss], fd)

loss\_track.append(l)

**if** batch == 0 **or** batch % batches\_in\_epoch == 0:

print('batch {}'.format(batch))

print(' minibatch loss: {}'.format(sess.run(loss, fd)))

predict\_ = sess.run(decoder\_prediction, fd)

**for** i, (inp, pred) **in** enumerate(zip(fd[encoder\_inputs].T, predict\_.T)):

print(' sample {}:'.format(i + 1))

print(' input > {}'.format(inp))

print('predicted > {}'.format(pred))

**if** i >= 2:

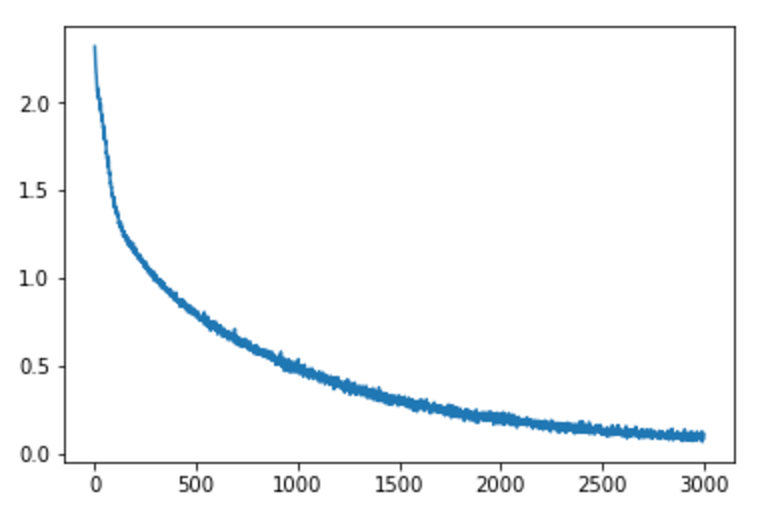
**break**

print()

**except** KeyboardInterrupt:

print(' training interrupted')

在30000次迭代后，loss值降到了0.0869。



大家可以尝试一下，需要源码的话，我稍后放在github上。