目录

[**一、 综述 1**](#_Toc505792360)

[**二、 相关知识点梳理 2**](#_Toc505792361)

[**2.1. 语言模型 2**](#_Toc505792362)

[**2.2. RNN简介 2**](#_Toc505792363)

[**2.3. 基于Encoder-Decoder的Seq2seq模型 7**](#_Toc505792364)

[**2.4. 双向RNN 9**](#_Toc505792365)

[**2.5. Attention机制 10**](#_Toc505792366)

[**2.6. Word Embedding 13**](#_Toc505792367)

[**2.7. Beam-search算法 14**](#_Toc505792368)

[**2.8. 评估指标 14**](#_Toc505792369)

[**（1）PPL(Perplexity) 14**](#_Toc505792370)

[**（2）BLEU(Bilingual Evaluation Understudy) 15**](#_Toc505792371)

[**三、 用TensorFlow来构建 17**](#_Toc505792372)

[**3.1数据预处理 17**](#_Toc505792373)

[**3.2相关API简介 19**](#_Toc505792374)

[**（1） tf.data.Dataset模块 19**](#_Toc505792375)

[**（2） tf.nn模块 21**](#_Toc505792376)

[**（3） tf.contrib.seq2seq模块 22**](#_Toc505792377)

[**3.3项目概览 26**](#_Toc505792378)

[**3.4使用 27**](#_Toc505792379)

[**四、 训练 27**](#_Toc505792380)

[**4.1概要 27**](#_Toc505792381)

[**4.2调参 28**](#_Toc505792382)

[**五、 结果 30**](#_Toc505792383)

[**六、 总结 34**](#_Toc505792384)

**用Sequnece to sequence模型构建简易Chatbot**

1. **综述**

目前应用于构建Chatbot的技术主要可以分为于两大类：基于检索的技术和基于深度学习生成模型的技术

检索式模型由于采用人工制作的回复库，基于检索式方法不会有语法错误。与此同时，缺点也是显而易见的，检索式模型不能处理没出现过的情况，即回复库中没有出现过的句式模型不会产生。同样，这些模型不能重新利用提上下文中的实体信息，如先前对话中提到过的名字。综上，检索式模型可以用在需要正确回答问题的场合，对答案的语法和准确性要求比较高。

生成式对话模型从原理上讲更“聪明“些。它们可以重新提及输入中的实体并带给你一种正和你对话的感觉。然而，这类模型很可能会犯语法错误（特别是输入一个长句时），而且通常要求大量的训练数据。综上，生成式对话模型可以用在要求不那么精确的对话中。比如游戏的NPC交谈，比如一般的生活类对话场景。

生成对话模型是目前比较前沿的对话系统构建技术。Sequence to sequence是进行序列到序列的建模一个比较成功的应用，其中一个重要的应用方向就是构建对话系统。

1. **相关知识点梳理**
2. **语言模型**

语言模型可以计算任何句子的概率。例如，“I love you”的概率是多少？用数学语言表述，N元语言模型(N-gram model)根据一个词语的前N−1个词语，来计算这个词语的概率。如果我们有了一个2元模型（Bi-gram Model），“I love you”的概率就等于P(I)×P(love|I)×P(you|love)。这样的计算方式基于马尔可夫假设，即：第n个词出现与前n-1个词相关，而与其他任何词不相关。

1. **RNN简介**

循环网络（RNN，recurrent neural network）是一类用于处理序列数据的神经网络。不像其他神经网络如CNN等，只接受定长的输入向量，一般的RNN不仅可以处理定长的序列，也可以处理变长的序列。

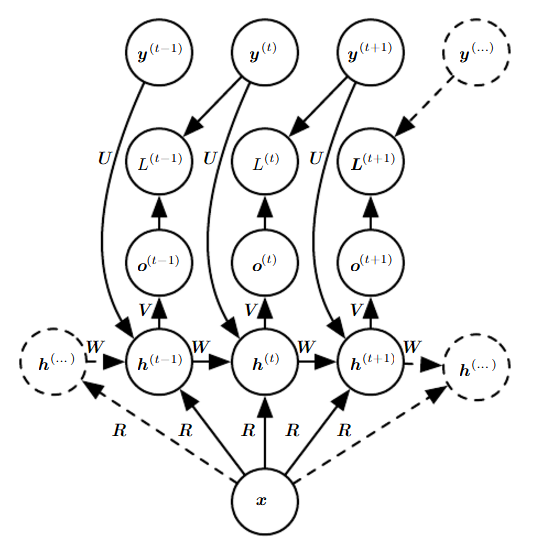


图 2.1

RNN处理定长序列的情况如上图所示，这种类型的RNN将固定长度的向量x映射到训练Y上，这里的y(t)同时作为t时刻的输入和t-1时刻的输出目标。X在每个时间步作为额外的输入项，通过与参数R相乘得到xTR作用于每个隐藏单元h(t)。此时的xTR可以视为隐藏单元的偏置项。这种模型可以视为采用了非条件模型的参数W，此时偏置为输入x的函数。项这一类RNN可以用做图注等任务，单个图像作为输入，输出描述图像的词序列。

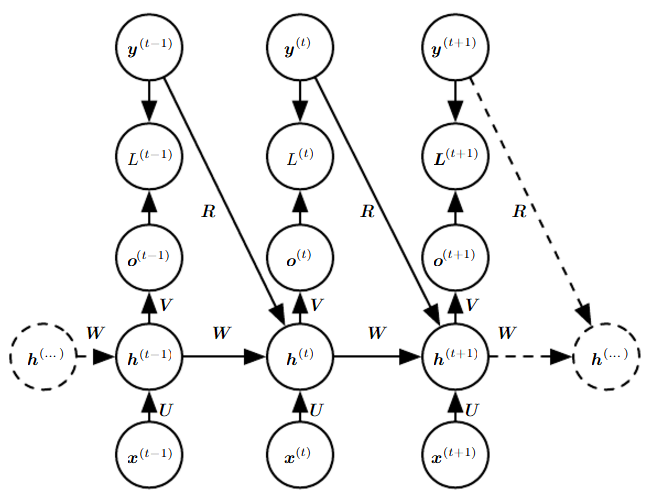


图 2.2

应用于自然语言处理的RNN更多的是如上图所示的可以处理变长序列的RNN。该模型可以代表关于y序列的任意概率分布，这里的x是一个可变长的序列，y是一个定长序列，序列的长度由隐藏单元的个数决定（这种限制可以通过Encoder-Decoder结构的建模方式来消除）。

使用RNN进行序列建模所面临的的一个重要挑战就是长期依赖问题。其根本问题还是由RNN模型的基本原理造成的，在经过多个阶段的传播后，RNN的梯度倾向于爆炸（很少，但对优化过程影响较大）或消失（大部分情况）。从公式上来说，就是RNN涉及到相同函数的多次嵌套组合，如果不考虑非线性激活，RNN的‘循环’递推过程可以简化为：



前一个时间步的状态是后一个时间步的输入，那么第t个时间步的状态可以简单描述为初始状态经过t步参数化（t次W相乘）得到：



而如果W可以被特征分解为以下形式：



其中Q为正交矩阵，RNN的循环性可以进一步简化为：



这样，随着t增大时，如果幅值Ʌ小于1，那么特征值会逐渐衰减到0，如果幅值Ʌ大于1，那么特征值会激增。对于梯度消失来说，长期相互作用的梯度幅值相比短期相互作用的幅值会变得指数小，这会导致有着长期依赖关系的信号很容易被短期相关性产生的最小波动而影响，使得学习长期依赖变得困难。这就是RNN所面临的长期依赖问题。

长短期记忆（LSTM，long short term memory）就是被设计用来解决上述长期依赖问题的RNN变种。LSTM引入自循环，以此产生梯度可以长时间持续流动的路径，这个自循环结构被称为“LSTM Cell”。和普通的RNN不同的是，LSTM拥有很多控制信息的“门”。下面将引用[Christopher Olah 的博文](https://link.jianshu.com/?t=http://colah.github.io/posts/2015-08-Understanding-LSTMs/)详细阐述LSTM的相关原理。

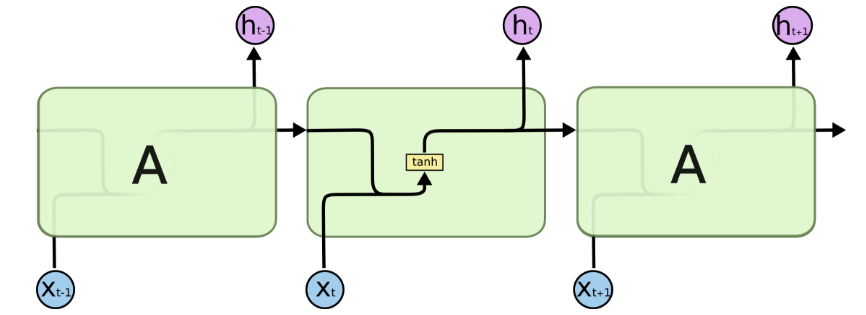


图2.3

普通的RNN的结构如上图所示，每个RNN重复模块只有一个很简单的结构。在输入和状态经过仿射变换之后施加一个非线性变换（激活函数，如tanh）。而LSTM的结构如下图所示：

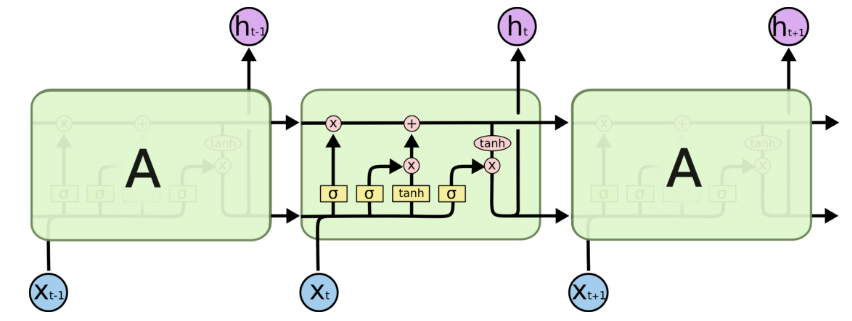


图 2.4

LSTM同样也有着和普通RNN一样的链式结构，但在每个重复模块中LSTM增加了许多“门”，这里的门通常是sigmoid非线性激活函数，sigmoid输出的值在0到1之间，0表示不允许任何信息通过，1表示任何信息都可以通过。通过门可以控制信息的流动。

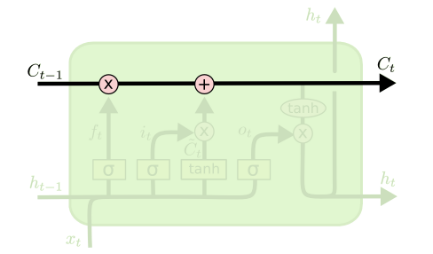


图 2.5

如上图所示，细胞状态类似于传送带。直接在整个链上运行，只有一些少量的线性交互。信息在上面流传保持不变会很容易。下面跟着Christopher Olah的图来一步步了解这其中的流程。

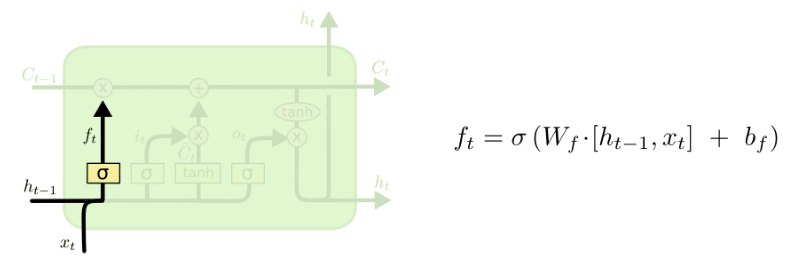


图 2.6

（1）首先，要确定什么样的信息被“忘记”。上个时刻的细胞状态C(t-1)在“传送带”上传递到当前时刻细胞状态C(t)（LSTM细胞状态自循环）的信息由遗忘门控制。上一个时刻的隐藏状态h(t-1)（这个隐藏状态由LSTM细胞状态经过非线性激活得到，后续说明）和当前时刻的输入x(t)，经过权重仿射变换后产生的输出经过sigmoid非线性激活产生0到1之间的门控权重，由上图所示（值得注意的是这里的h(t-1)和x(t)通过拼接操作后共用了一个权重变量，这也和TensorFlow在LSTM实现中的做法相同，而许多文献中的公式会把h(t-1)和x(t)的权重区分开，但是效果是一样的）。

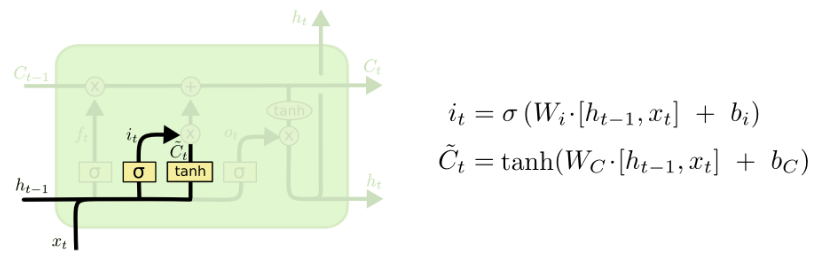


图 2.7

（2）接下来需要确定什么样的信息被保留在细胞状态中。这由两部分组成：a）由输入门产生0到1之间的输出控制信息保留的权重。b）由一个非线性激活tanh对h(t-1)和x(t)进行仿射变换产生激活输出（这一步类似普通RNN的操作）。二者相乘产生当前时刻细胞状态的输入（被保留的信息）。

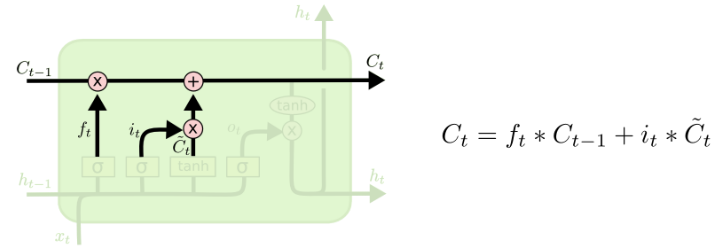


图 2.8

（3）这一步更新LSTM细胞状态，上一个时刻的细胞状态C(t-1)和当前时刻的遗忘门相乘产生“历史信息”，当前时刻的激活输入和输入门相乘产生“当前信息”，两部分信息相加产生当前时刻的细胞状态C(t)

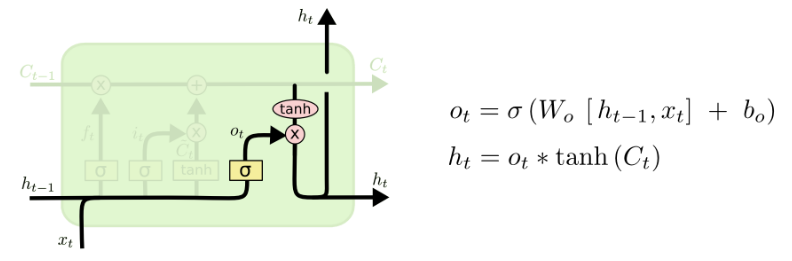


图 2.9

（4）最后要确定的是输出什么值，也就是产生当前时刻的隐藏状态h(t)，输出也是由一个“门”来控制的，对当前时刻细胞状态C(t)进行非线性激活，再和输出门权重相乘得到最终的隐藏状态h(t)

以上就是LSTM的整体架构，LSTM相比于普通RNN更容易学习长期依赖。LSTM之后也有许多其他的研究者提出一些变种，如GRU等。也有相关研究表明，向LSTM的遗忘门加入1的偏置可以让LSTM和其最佳变种一样健壮。

1. **基于Encoder-Decoder的Seq2seq模型**

Seq2seq是2014年先后分别由Yoshua Bengio团队的《Learning Phrase Representation using RNN Encoder-Decoder for Statistical Machine Translation》和Google Brain团队的《Sequence to Sequence Learning with Neural Networks》两篇论文提出，两个团队不约而同的都采用了Encoder-Decoder架构的设计思路，致力于解决变长序列到变长序列的映射问题。关于Encoder-Decoder，一个通俗易懂的架构图如下所示：

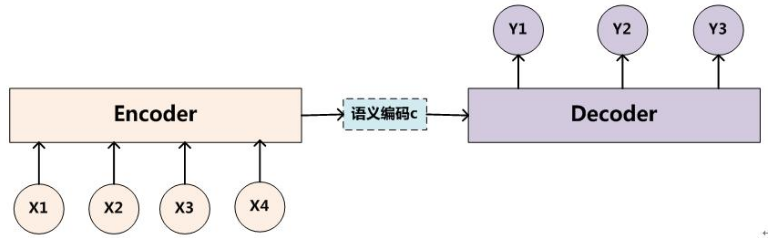


图 2.10

Seq2seq设计的初衷就是为了解决机器翻译中的变长序列映射问题。传统的DNN只能应用于输入和输出都是固定维度向量的问题，这样的局限性导致DNN不能够很好的解决输入输出向量维度事先不能预知的问题，比如机器翻译、语音识别等。Seq2seq的思路非常清晰明朗，既然 RNN擅长处理序列数据，那么将RNN构造成Encoder-Decoder架构的形式不就可以很好的解决序列到序列的映射问题了么。于是作者提出了如下的Seq2seq架构：



图 2.11

如图所示，模型接受一个序列输入“ABC”，编码-解码操作产生一个序列输出“WXYZ”。<EOS>（End Of Sequence）用作模型预测的定界符，是用户指定的特殊字符，不包含在所要训练的数据词汇表中。当模型解码遇到<EOS>就不再继续进行预测。所以依据论文中的描述，最终输入的序列为“ABC<EOS>”，输出序列为“WXYZ<EOS>”。

（注：对于这一点一开始比较费解，因为从Google NMT项目的源码中得到的答案是输入序列并没有做<EOS>的Padding操作，输入是未经Padding的原始串‘ABC’。在训练阶段，target input被Padding为“<EOS>WXYZ”作为每个时间步Decoder的输入, target output被Padding为“WXYZ<EOS>”作为优化目标输出（Label）。在用已有的模型进行推断时，<EOS>是做为整个解码操作的初始输入，加上Encoder的final\_state一同作为Decoder Cell的初始输入进行解码操作，所以如果把Encoder的finalstate记为Decoder的state\_0，第一次解码输出记做output\_0，第一次解码状态记做state\_1，以此类推，那么整个解码流程的输入序列是(<EOS>，state\_0)->(output0, state\_1)->(output\_1，state\_2)->……直到output\_n为<EOS>。这里的<EOS>并没有参与到Encoder的编码流程，这样的实现方式和文中描述略有不同，特此记录。）

Seq2seq的工作流程分为编码阶段和解码阶段。在编码阶段，处于编码结构的LSTM通过计算得到一个固定维度大小的特征表示v（LSTM的最终状态或由注意力机制引入的所有状态的加权平均）。在解码阶段，处于解码器结构中的LSTM以v作为初始状态，对下一时刻的序列元素进行预测，每个时刻可能出现的概率最大的元素将被选择（此处也可以引入BeamSearch），公式如下：



其中，每个分布是由一个Softmax分类器产生，类别的个数为整个词汇表的大小。由此可见，处于解码结构的LSTM其实就是一个RNN语言模型，在每次时刻t，LSTM根据前t-1个词来预测第t个词。

此外，作者还在文中提到，在实际的实现中，他们使用的方法和图中描述的结构有以下不同：

1. 采用了不同的LSTM单元，一个负责输入，一个负责输出（Encoder-Decoder），因为这样以较小的计算代价增加了模型的参数数量，并且可以更自然地采用并行计算来提升训练速度。
2. 每个LSTM使用了多层结构，这样相较于浅层LSTM可以明显提升训练效果。
3. 把输入序列进行翻转操作可以明显提升效果。一种可能的解释是这样引入了更多的短期依赖比如a,b,c->e,f,g，进行原序列翻转变为c,b,a->e,f,g，这样a和e的距离更近了。虽然这样并没有改变源串和目标川元素之间的平均距离，但是源串和目标串之间开头的几个元素间的距离更近了，因此他们的最小时间间隔（minimal time tag）变得更小。

初始的Seq2seq模型在翻译问题上取得了较好的效果，但这个初始的模型还有很多优化提升的空间，比如使用双向RNN，注意力机制等。

1. **双向RNN**

2.1节描述的RNN，存在一个因果结构，即当前时刻t的状态只能从过去的输入信息x(1),x(2),…,x(t-1)以及当前时刻的输入x(t)捕获信息，或者过去时刻的某些y也会对当时刻的状态产生影响（图2.2所示）。然而在某些应用中如语音识别中，由于协同发音，当前时刻的声音可能取决于前后附近的几个音素；文本处理中，一句话的词与附近的几个词都存在语义依赖关系，我们可能需要在遥远的未来寻找信息来区分他们，seq2seq也是如出一辙。双向循环网络（双向RNN就是为满足这种需求而发明），它在需要双向信息的应用中应用的非常成功，如手写识别，语音识别以及生物信息学等。

双向RNN顾名思义，就是在时间上从两个方向移动的RNN序列模型，一个RNN从序列开始移动，另一个RNN从序列末尾开始移动。如下图所示：

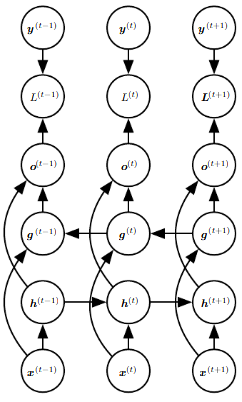


图 2.12

其中，h(t)表示通过时间向后移动的子RNN的状态，g(t)表示通过时间向前移动的子RNN的状态，每个时刻的输出o(t)同时接收前向和后向的状态信息，这允许输出o(t)能够同时计算依赖于过去和未来信息且对时刻t最敏感的表示，而不必指定固定大小的窗口（这是前馈网络、卷积网络和具有固定大小先行缓冲器的常规RNN所必须要做的）。这个想法也能够很容易的扩展到二维输入，如图像等，由4个RNN组成，每一个RNN沿着上下左右其中一个方向计算，如果RNN能够学习到承载长期信息，那么o(i,j)就能计算一个捕捉到大多局部信息但仍依赖长期输入的表示。

1. **Attention机制**

Attention机制最初是由Bahdanau提出（[Bahdanau, et al. 2015](https://arxiv.org/pdf/1409.0473.pdf)），其背后的动机来源于对Seq2seq模型瓶颈的猜想。基本的基于Encoder-Decoder架构的seq2seq模型首先把源序列映射到一个固定长度的向量（语义向量），然后在Decode阶段对这个中间语义向量再进行解码，解码的过程可以表示为求解如下的条件概率：



其中，y={y1,y2,…,yt},C是一个固定长度的语义向量。当源序列长度较小时，模型有可能还会取得不错的效果，但是当源序列长度逐渐增大时，这个固定长度的向量很难把源序列的所有信息都表示出来，尤其是当输入序列的长度大于训练时源序列的长度时，随着长度的增加，模型的性能急剧恶化。

上述基础seq2seq是没有Attention机制的，之所以这么说，是因为在进行解码推断阶段，无论是进行哪个单词的预测，模型所使用的语义编码都是一样的，即相同的C。这意味着源序列中的每个单词对解码输出任何单词的作用都是一样的（如果Encoder使用的是RNN，那么理论上是源序列中时间上越靠后的元素对语义编码的贡献越大，并非等权，这也是所谓的长短期依赖的作用影响。这也是Google将源序列逆置会得到更好的翻译效果的原因，增加了开始序列元素的对语义编码的权重）。Attention机制提出了一个更加合理的方法来解决上述问题，新的架构充分利用了Encoder阶段各个时刻的编码信息（隐藏状态），架构图如图2.13所示。在新的架构中，解码阶段的条件概率函数如下所示：



其中，Si为Decoder第i时刻的隐藏层状态：



Ci为不同的语义编码向量，对应于不同的语义注意力概率分布：



本质上是对源序列的隐藏状态进行加权求和，其中权值由下式计算：





a是一个对齐模型，用来评估目标序列i时刻和源序列j时刻附近输入的匹配程度，作者这里使用的是一个参数化的前馈网络，并且能够和模型的其他参数进行联合训练（个人理解是这里对齐模型的参数直接参与到整个模型的前向和反向传播，并没有独立的损失函数）。这个概率分布依赖于源序列j时刻的隐藏状态（或输出）和目标序列i-1时刻的隐藏状态。注意力分布计算架构如图2.14所示，右边的序列表示目标序列，Hi这里指的就是上述的si。



图 2.13

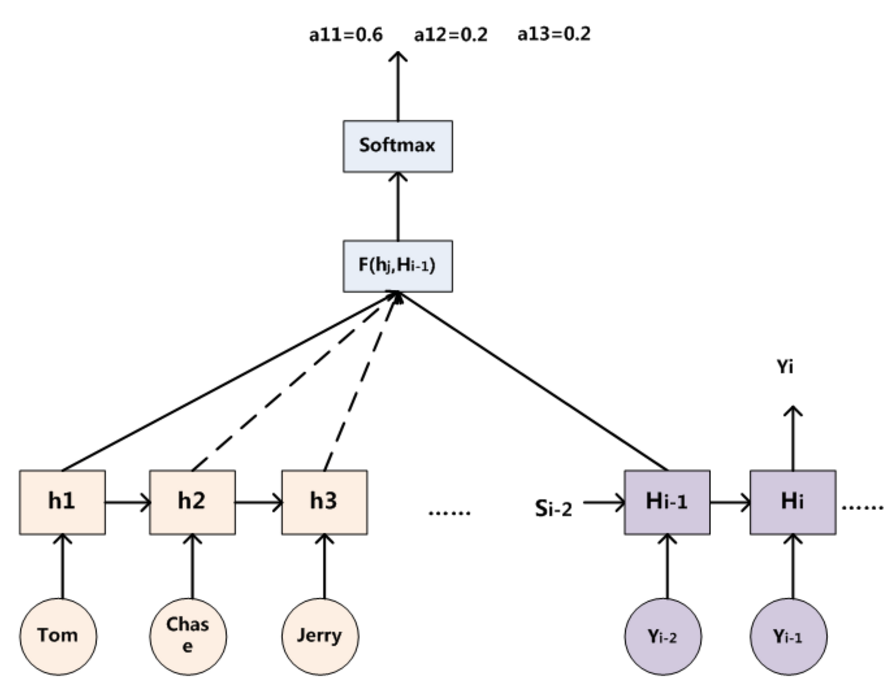


图 2.14

上述Attention模型就是经常提起的Soft Attention Model，最主要的思想就是摒弃了Encoder产生固定的语义向量的做法，而是根据不同的注意力分布产生源序列的不同语义编码。这个注意力分布可以理解为源序列和目标序列单词之间的对齐概率分布，这可以通俗的理解为：在生成目标序列某个单词的时候，源序列的每个单词到底对当前步骤产生多大的影响。关于Attention机制通俗的说明可以参考<http://blog.csdn.net/jdbc/article/details/50586099>，更加形象的可视化解释可以参考<https://distill.pub/2016/augmented-rnns/>

1. **Word Embedding**

Word Embedding中文译名为词嵌入，可以简单的理解为一种将词进行向量化表示。目前所有的建模技术都是基于数字量化的，所以如何将抽象的文本表示为计算机能够计算的数字形式，这就需要进行Word Embedding。Word Embedding是一种词的表示方式，是一种基于神经网络语言模型的词的分布式表示。

词的表示当然也可以表示成传统的one-hot形式，比如：

“话筒”表示为 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 ...]  
“麦克”表示为 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 ...]

每个词都是这样的一个个稀疏向量，这样的缺点非常明显，那就是存储和计算的代价都比较高，并且词与词之间没有任何语义上的关联信息。Word Embedding使用更低纬度的浮点型向量对词进行分布式表示，形式如下：

“话筒”表示为 [0.1123 0.1344 0.1535 0.1234 0.0534…]  
“麦克”表示为 [0.1423 0.1233 0.3423 0.0342 0.0534…]

因为Word Embedding是使用神经网络语言模型生成的，所以也进一步考虑了词与词之间的语义联系，理论上对词的表示更加合理，对模型的训练也有一定的好处。

1. **Beam-search算法**

Beam-search算法只在模型的推断阶段使用，本质上是一种解码选择策略。Beam-search在解码的每一时间步都采用贪心策略选取下一步解码所要选择的序列元素。Beam-search算法需要指定beam width或者beam size，就是每一步解码所要选择候选词的个数。假设我们的词汇表中只有I love NJUPT 3个词，beam width为2，详细的算法过程如下（结合图2.11）：

（1）根据语义向量，第一次Decode的输出分布为I 0.5, love 0.3, NJUPT 0.2

（2）选择概率最大的两个词最为下一步Decode的候选输入，这里选择 I，love

（3）把I作为输入，结合上一步的隐藏状态进行Decode，得到概率分布I 0.1, love 0.6, NJUPT 0.3。把love作为下一步解码的输入得到概率分布I 0.2, love 0.2, NJUPT 0.6。

（4）此时计算所有可能序列的概率:

’I,I’=0.5\*0.1=0.05 ‘I,love’=0.5\*0.6=0.3 ‘I, NJUPT’=0.5\*0.3=0.15

‘love,I’=0.3\*0.2=0.06 ‘love,love’=0.3\*0.2=0.06 ‘love,NJUPT’=0.3\*0.6=0.18

最后选取概率最大的两个序列‘I，love’和‘love，NJUPT’

（5）继续根据（4）算出的序列计算下一步解码的输出，计算概率，选择概率最大的序列。如此循环解码直到遇到结束符或者达到最大解码长度结束算法过程。

上述过程也可以理解为构造了一个三叉树，三叉树的每个路径上的节点构成一个最终序列，树的每个节点的子节点构成整个此空间的概率分布。计算每条路径的概率积，最终取最大的beam width个序列。

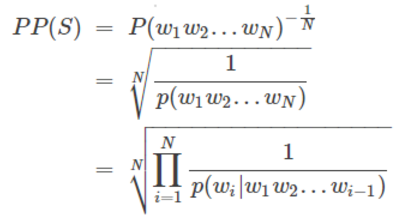
应用Beam-search算法，从最终的beam width个序列中随机选取一个，可以使得chatbot的每次对相同的问题产生不同的回复。

1. **评估指标**

在神经机器翻译领域，对机器翻译的结果进行人工评估是高代价且无法复用的。于是人们引入了很多机器自动评估的方法。对于对话系统而言，和机器翻译有诸多相似之处，评估算法也可以引入在机器翻译领域普遍使用的一些评估指标。常用的评估指标列举如下：

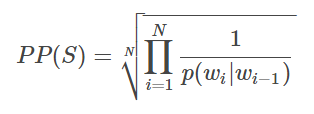
**（1）PPL(Perplexity)**

Perplexity是一种衡量自然语言处理领域（NLP）中，语言模型的好坏的指标。其公式定义如下：



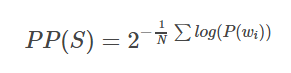
其中S代表一个句子，Wi表示句子S中的第i个词，N为S的长度。

对于Bi-gram模型来说，其公式如下：



其中，对于第一个词，就是p(w1|w0)，w0表示START，是一个占位符，表示句子的起始。

另一种表达：



其中，N是句子S的长度，P(wi)是第i个词的概率（如果是bigram就应该是第i个bigram的概率）

Perplexity越小越好，相应的，就是我们见过的句子出现的概率越大越好。对于perplexity的影响因素有：

1. 训练数据集越大，PPL会下降得更低，1billion dataset和10万dataset训练效果是很不一样的；

2. 数据中的标点会对模型的PPL产生很大影响，一个句号能让PPL波动几十，标点的预测总是不稳定；

3. 预测语句中的“的，了”等词也对PPL有很大影响，可能“我借你的书”比“我借你书”的指标值小几十，但从语义上分析有没有这些停用词并不能完全代表句子生成的好坏。

**（2）BLEU(Bilingual Evaluation Understudy)**

<https://wenku.baidu.com/view/891482f5e2bd960591c677d4.html>

BLEU即双语互译质量评估辅助工具，是由IBM的研究人员提出的一种一种自动评估机器翻译，且独立于语言的方法，方法和人工评估是高度正相关的。每次运行只有少量的边缘代价。

BLEU方法的主要思想就是机器的翻译结果和人工翻译的结果越接近，那么机器翻译的效果就越好。基于这个思想，其最重要的评估依据就是机器译文和参考译文之间的相似度。那么如何度量这个相似度，就是BLEU计算方法的关键。

作者提出的相似度计算方式是统计N-gram单词相同的个数，即对待评价译文和参考译文的“n-单位片段（n-gram）”进行比较，并计算出匹配片段的个数。这些匹配片段与它们在文字中存在的位置无关。匹配片段数越多，则待评价译文质量越好。值得注意的是，如果有三个参考译文，那么此方法将待评价译文和三个参考译文进行比较，待评价译文中单词出现在三个参考译文中的个数除以待评价译文中总单词个数，便得到原始的精确度（precision）计算结果。

然而这样的简单的精确度计算存在一些问题，例如：

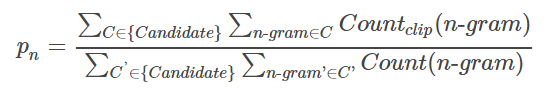
待评价译文 : the the the the the the the

参考译文 1: The cat is on the mat.

参考译文2: There is a cat on the mat.

问题很明显，当某个参考译文中的某个单词匹配完以后，这个单词就不应该再继续计数匹配了。按照上述的计算方法，并不能体现待评价译文和单句参考译文的相似程度。于是作者提出修正过的n-单位精确度即：参考译文中如果一个单词片段已经被匹配，那么这个片段就不能再次被匹配，并且一个单词片段只能取一个参考译文中出现次数的最大值，比如7个the分别在参考译文1 和 2中出现2和1次，所以取2而不是两者相加的3。

利用以上方法，每一个句子都可以得到一个modified n-gram recision，一个句子不能代表文本翻译的水平高低，于是把一段话或者所有翻译句子的结果综合起来可以得到pn。即：



#### 另外还要考虑的是待测译文翻译不完全不完整的情况，这个问题在机器翻译中是不能忽略的，而简单的pn值不能反映这个问题。于是引入 BP值(Brevity Penalty)。作者指定当待评价译文同任意一个参考译文长度相等或超过参考译文长度时，BP值为1，当待评价译文的长度较短时，则用一个算法得出BP值。以c来表示待评价译文的长度，r来表示参考译文的文字长度，则：

#### 

#### 基于以上讨论，最终的BLEU值的计算公式为：

#### 

#### N通常取4，wn=1/4，这就是很多论文里面的一个经典指标Bleu4。BLEU值的取值范围是从0到1的数值。只有译文同参考译文完全一致的时候才会有评分为1的情况。值得注意的一点是，使用越多的参考译文，BLEU评分值就越高。BLEU值越高越好。

1. **用TensorFlow来构建**

## 3.1数据预处理

根据Seq2seq论文中所提到的方法，首先要对带训练数据进行格式化处理，包括分词，构建词汇表，构建完整的训练集、验证集和测试集。

实验采用的是小黄鸡聊天对话数据集。原始文本格式如下：

E  
M 呵呵  
M 是王若猫的。  
E  
M 不是  
M 那是什么？  
E  
M 怎么了  
M 我很难过，安慰我~  
E

（1）首先处理成如下格式：

questions.txt文件

呵呵  
不是  
怎么了

answers.txt文件

是王若猫的。  
那是什么？  
我很难过，安慰我~

也就是把原始文件拆分成question.txt和answers.txt两个文件，每行一句问或答，两个文件的问答要逐行一一对应。

（2）分词：

分词采用的是结巴分词工具，安装使用命令pip install jieba。分别对questions.txt文件和answers.txt文件逐行切词，词汇之间以空格隔开，格式如下：

questions:

呵呵  
不是  
怎么 了  
开心 点哈 , 一切 都 会 好 起来

answers:

是 王若 猫 的 。  
那 是 什么 ？  
我 很 难过 ， 安慰 我 ~  
嗯 会 的

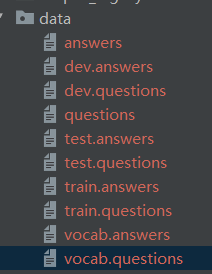
（3）构建词汇表

根据切好词的构建词汇表，词汇按词频从高到低排序存放，每个词占一行。读取的时候以行号作为词的id。需要注意的是，词汇表的前3个词约定为特殊字符<unk>，<s>，</s>，分别表示未知词汇，句子开始，句子结尾。格式如下：

<unk>  
<s>  
</s>  
   
你  
我  
,  
了

（4）拆分数据集

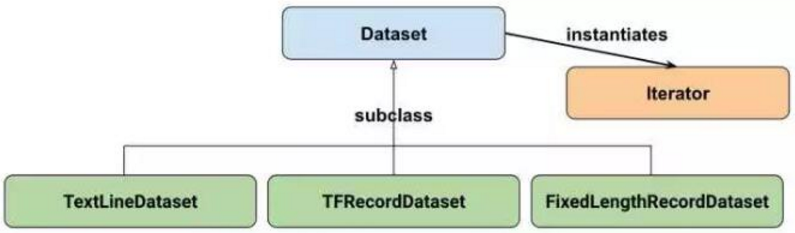
按比例把分好词的questions和answers拆分为训练集、验证集和测试集。整个数据目录如下：



## 3.2相关API简介

以下API说明基于TensorFlow 1.5.0，由于TensorFlow版本迭代改动较大，其他版本可能不适用。

1. **tf.data.Dataset模块**



官方API给出的描述是：A Dataset can be used to represent an input pipeline as a collection of elements (nested structures of tensors) and a "logical plan" of transformations that act on those elements.

Dataset可以视作元素的有序列表，并且支持一系列常用的数据序列操作，比如数据混洗，剪切，筛选，遍历等。其中单个“元素”可以是向量，也可以是字符串、图片，甚至是tuple或者dict。在数据集上执行一系列逻辑操作之后最终需要创建一个Iterator实例才能在实际数据中进行迭代。

a）Dataset实例创建：

Dataset基类提供了创建Dataset实例的一些静态方法：

**Dataset.from\_tensors：**由指定的tensor创建Dataset

**Dataset.from\_tensor\_slices：**从tensor切片中创建Dataset，例如:

dataset = tf.data.Dataset.from\_tensor\_slices(np.random.uniform(size=(5, 2)))这会以第一个维度切分输入的数据，创建一个5个元素的shape为(2,)的Dataset。

dataset = tf.data.Dataset.from\_tensor\_slices(

{

"a": np.array([1.0, 2.0, 3.0, 4.0, 5.0]),

"b": np.random.uniform(size=(5, 2))

}

)

这会创建一个5个元素，每个元素为一个字典如第一个元素为{"a":1.0, "b":[0.1,0.9]}

dataset = tf.data.Dataset.from\_tensor\_slices(

(np.array([1.0, 2.0, 3.0, 4.0, 5.0]), np.random.uniform(size=(5, 2)))

)

同样的，这会创建一个5个元素的元祖Dataset。

**Dataset.from\_generator：**从一个generator创建Dataset

**Dataset.zip：**把多个Dataset压缩为一个Dataset操作

**Dataset.concatenate:**把多个Dataset拼接为一个Dataset操作

(更多方法见API Doc)

此外，还可以通过继承Dataset的子类创建：

**tf.data.TextLineDataset()：**这个函数的输入是一个文件的列表，输出是一个dataset。dataset中的每一个元素就对应了文件中的一行。可以使用这个函数来读入CSV文件。

**tf.data.FixedLengthRecordDataset()：**这个函数的输入是一个文件的列表和一个record\_bytes，之后dataset的每一个元素就是文件中固定字节数record\_bytes的内容。通常用来读取以二进制形式保存的文件，如CIFAR10数据集就是这种形式。

**tf.data.TFRecordDataset()：**顾名思义，这个函数是用来读TFRecord文件的，dataset中的每一个元素就是一个TFExample。

b）Iterator实例的创建：

初始化Iterator实例的方法有两个：make\_initializable\_iterator和make\_one\_shot\_iterator。make\_initializable\_iterator所创建的Iterator实例支持重新初始化操作，这在实际的多epoch训练中是很有必要的，当数据迭代完一轮之后，通过执行Iterator的initializer属性来重新初始化迭代器来进行新的一轮训练。而通过make\_one\_shot\_iterator方法创建的Iterator实例不支持重新初始化操作，只能在目标数据集上迭代一次。

需要进一步说明的是，initializable\_iterator还可以从Placeholder来创建，例如：

with np.load("/var/data/training\_data.npy") as data:

features = data["features"]

labels = data["labels"]

features\_placeholder = tf.placeholder(features.dtype, features.shape)

labels\_placeholder = tf.placeholder(labels.dtype, labels.shape)

dataset = tf.data.Dataset.from\_tensor\_slices((features\_placeholder, labels\_placeholder))

iterator = dataset.make\_initializable\_iterator()

sess.run(iterator.initializer, feed\_dict={features\_placeholder: features,

labels\_placeholder: labels})

这里指定了两个placeholder，分别代表features和labels数据，这样方便我们在使用的过程中灵活的替换数据。同时，placeholder还可以避免较大的数组被保存在计算图中，因为当我们执行tf.data.Dataset.from\_tensor\_slices(array)的时候，实际上TensorFlow是把array作为tf.constants保存在了计算图中，如果array很大，那么会导致计算图很大，这时使用placeholder可以避免这一问题。本项目中的Iterator模块的InferIterator就使用了上述操作，这样在聊天循环中可以每次输入一句话构建feed\_dict并重新运行iterator.initializer初始化Iterator把数据输入模型。

此外，还有两个比较复杂的Iterator类型：reinitializable iterator、feedable iterator。这两个并不常用，这里只列举出来，详细信息见官方API Doc.

c）Dataset逻辑操作（Transformation）

**map：**map接收一个函数，Dataset中的每个元素都会被当作这个函数的输入，并将函数返回值作为新的Dataset，如我们可以对dataset中每个元素的值加1：

dataset = tf.data.Dataset.from\_tensor\_slices(np.array([1.0, 2.0, 3.0, 4.0, 5.0]))

dataset = dataset.map(lambda x: x + 1) # 2.0, 3.0, 4.0, 5.0, 6.0

**batch：**batch就是将多个元素组合成batch，如下面的程序将dataset中的每个元素组成了大小为32的batch：dataset = dataset.batch(32)

**shuffle：**shuffle的功能为打乱dataset中的元素，它有一个参数buffersize，表示打乱时使用的buffer的大小：dataset = dataset.shuffle(buffer\_size=10000)

**repeat：**repeat的功能就是将整个序列重复多次，主要用来处理机器学习中的epoch，假设原先的数据是一个epoch，使用repeat(5)就可以将之变成5个epoch：

dataset = dataset.repeat(5)

如果直接调用repeat()的话，生成的序列就会无限重复下去，没有结束，因此也不会抛出tf.errors.OutOfRangeError异常：

dataset = dataset.repeat()

（本项目中使用make\_initializable\_iterator创建可重新初始化Iterator实例，在每次迭代完一遍数据之后抛出tf.errors.OutOfRangeError异常执行相关操作，再重新初始化Iterator进行下一轮迭代）

1. **tf.nn模块**

Encoder部分的实现只需要使用tf.nn模块的一些API即可，整个编码架构如下：



说明：

**Input Data：**是经过词汇表映射得到的Id序列，一次输入一个mini-batch，所以shape为（batchsize，max\_length）；

**Embedding：**对输入数据使用tf.nn.embedding\_lookup函数进行Embedding操作，得到Embedded Input。如果是time\_major的格式，shape为(max\_time,batch\_size,embedding\_size),否则Embedded Input 的shape为(batch\_size,max\_time,embedding\_size)。

**MultiRNNCell：**接收Embedded Input，每个时间步输出一个Output和state，前一个时间步的state输出作为下一个时间步的state输入，直到达到max\_time。

上述Encoder架构的整个编码过程是通过tf.nn.dynamic\_rnn()接口完成的，接口输出每个时间步的rnn输出Outputs和最后一个时间步的隐藏单元状态final\_state

1. **tf.contrib.seq2seq模块**

seq2seq模块不仅包含了实现基本seq2seq的功能API，还包含了一些常用的改进方法封装，包括注意力机制、BeamSearch等。基本的原理已在2.4节的sequence to sequence模型中详细阐述，这里不再赘述。从TensorFlow具体的API实现的角度看，解码的框架如下：



说明：

**Decoder Input：**与encoder的一样，也是序列元素对应的id。

**Embedding：**视情况而定需不需要与encoder的embedding不同，比如在翻译中，源语言与目标语言的词向量空间就不一样，但是像文本摘要这种都是基于一种语言的，encoder与decoder的embedding matrix是可以共用的。

**Output Layer：**与encoder仅输出hidden state不同，decoder需要输出每个时刻词典中各token的概率，因此还需要一个dense layer将hidden state向量转换为维度等于vocabulary\_size的向量，然后再将dense layer输出的logits经过softmax层得到最终的token概率。

Decoder的初始Input均为特殊字符<EOS>，下一步的输入由Decoder所处的阶段决定。如果是训练阶段，那么下一步的输入依次为Target Input的下个时间步的元素。如果是推断阶段，那么下一步的输入为上一步MultiRNNCell的输出经过逻辑处理再经过采样得到。具体来说就是：

**Inference阶段**，decoder的输出是未知的，对于生成['W', 'X', 'Y', 'Z', '<EOS>']序列，是在decoder输出token 'W'之后，再将'W'作为输入，结合此时的hidden state，推断出下一个token 'X'，以此类推直到输出为<EOS>或达到最长序列长度之后终止。

而在**Training阶段**，decoder应该输出的序列是已知的，不管最终output的结果是什么，都将已知Target Input序列中的token依次输入。train的阶段如果也将输出的结果再作为输入，一旦前面的一步错了，都会放大误差，导致训练过程更不稳定。

**API：**

TensorFlow的seq2seq接口，大致可以分为3类：数据接口（各种Helper）、dynamic\_rnn相关接口（包括Decoder接口）、Wrapper接口。其中数据接口主要负责为dynamic\_rnn提供动态数据输入（每一个时间步的输入）；dynamic\_rnn相关的接口主要实现了rnn的主要计算流程；Wrapper接口封装了一些额外的算法，如Attention等。项目所用到的接口说明如下：

**BasicDecoder：**BasicDecoder是实现Decoder最为关键的一个接口，整个流程的主要操作是靠调用BasicDecoder的step函数进行的，step接收上个时间步的output和state输出当前时间步的output和state。

参数：def \_\_init\_\_(self, cell, helper, initial\_state, output\_layer=None)

cell：MultiRNNCell实例

helper：对应不同操作的Helper实例。Helper为解码过程提供数据输入。

initial\_state：Decoder初始状态，这里即为Encoder的final\_state，类型要一致，也就是说如果encoder的final state是tuple类型(如LSTM的包含了cell state与hidden state)，那么这里的输入也必须是tuple。直接将encoder的final\_state作为这个参数输入即可。

output\_layer：对应于框架图中的output layer，这里使用的是tensorflow.python.layers.core import Dense，在训练阶段不是必须的，在推断阶段用于逻辑输出。

**BeamSearchDecoder：**顾名思义，这个Decoder对BeamSearch算法进行了封装。

参数：def \_\_init\_\_(self, cell, embedding, start\_tokens, end\_token, initial\_state,

beam\_width, output\_layer=None, length\_penalty\_weight=0.0):

cell：MultiRNNCell实例

embedding：decoder的embedding矩阵

start\_tokens：初始指定的标志着句子开始的特殊字符id，shape：（batch\_size,）

end\_token：初始指定的标志着句子结束的特殊字符id，是一个标量

initial\_state：Decoder初始状态，这里即为经过tile\_batch 操作的Encoder的

final\_state，shape：（[batch\_size \* beam\_width, ...]）

beam\_width：每个beamsearch时间步的选择概率值的候选个数

output\_layer：同BasicDecoder

length\_penalty\_weight：序列长度的惩罚因子，是一个标量。计算score的时候用到（log\_probs / length\_penality\_），其中length\_penality\_= [(5+sequence\_lengths)/6]\*\*penalty\_factor

值得注意的是BeamSearchDecoder没有helper参数，这部分操作直接放在了内部函数\_beam\_search\_step()里面做了。

**TrainingHelper：**用于训练阶段的Helper，next\_inputs()函数返回的下一时间步输入为已知target inputs输入序列的一组元素。

参数：def \_\_init\_\_(self, inputs, sequence\_length, time\_major=False, name=None)

inputs：decoder的输入，经过embedding操作的decoder target inputs序列，time\_major=True时shape： (max\_time,batch\_size,embedding\_size)，time\_major=True=False时，shape：(batch\_size,max\_time, embedding\_size)

sequence\_length：一个mini-batch的输入序列长度，shape：(batch\_size,)

time\_major：指定输入格式，见inputs说明。

name：name scope.

**GreedyEmbeddingHelper：**用于推断阶段的Helper，next\_inputs函数返回的下个时间步的输入为上个时间步解码操作的sample id。

参数：def \_\_init\_\_(self, embedding, start\_tokens, end\_token)

embedding：decoder的embedding矩阵

start\_tokens：初始指定的标志着句子开始的特殊字符id，shape：（batch\_size,）

end\_token：初始指定的标志着句子结束的特殊字符id，是一个标量

**dynamic\_decode：** 在Decoder对象上循环调用step()函数进行解码。内部通过一个control\_flow\_ops.while\_loop操作实现循环解码，知道所有的序列都遇到了<EOS>或者达到最大解码长度，一个mini-batch的解码操作完成。

参数：

decoder：Decoder实例

output\_time\_major：指定输出shape，同以上time\_major释义

impute\_finished：如果设置为True，对于解码过程中标记为finished的序列，每个时间步的output为0序列，state则直接复制finished之前最后一个步骤的state。如果为False，则已经finished的序列仍然正常执行解码，上个时间步的输出做为下个时间步的输入，上个时间步的state作为下个时间不得输入state。

maximum\_iterations：指定最大解码步数（建议设为最大目标序列长度的2倍）

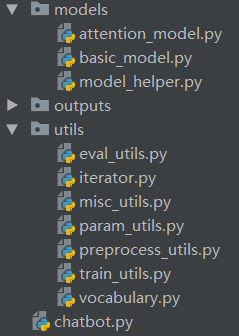
parallel\_iterations：tf.while\_loop并行执行的iteration数量。

swap\_memory：tf.while\_loop参数，是否允许CPU-GPU内存交换

scope：name scope

## 3.3项目概览

项目结构如下：



说明：

**models：**

**basic\_model.py：**定义了seq2seq model基础架构，包含Base和BasicModel两个类，BasicModel继承Base实现了build\_encoder和\_create\_decoder\_cell两个抽象方法

**attention\_model.py：**继承basic\_model，重写了\_create\_decoder\_cell方法，加入AttentionMechanism。

**model\_helper.py：**model创建所需的基础方法封装。

**utils：**

**eval\_utils.py：**模型评估计算方法封装

**iterator.py：**数据迭代器封装，作为model的参数传入。

**misc\_utils.py：**各种各样的杂项操作封装

**param\_utils.py：**Python参数解析操作封装

**preprocess\_util.py：**数据预处理封装

**train\_utils.py：**训练所需的辅助方法封装

**vocabulary.py：**词汇表封装，作为model的参数传入

**chatbot.py：**封装了模型训练、推断、对话的总体流程

## 3.4使用

**shell进入项目顶级目录，执行：**

python chatbot.py –mode train即可开始训练

4.1将结合某些参数简要说明。具体参数可以通过python chatbot.py –help来获取，或者查看param\_utils.py源码。

1. **训练**

## 4.1概要

数据集被划分为训练集、验证集和测试集。数据集一共200000条数据，其中训练集198000条数据，验证集和测试集各1000条数据。batch-size设为32（显存充足可以尝试64或更大来提升训练速度），一共训练了40个epoch。详细参数如下（json配置）：

{

"data\_dir": "data", # 数据目录，存放各数据集，数据集文件名为

# ”前缀名.后缀名”的形式

"src\_suffix": "questions", # 源数据集文件名后缀

"tgt\_suffix": "answers", # 目标数据集文件名后缀

"train\_prefix": "train", # 训练集文件名前缀

"dev\_prefix": "dev", # 验证集文件名前缀

"test\_prefix": "test", # 测试集文件名前缀

"vocab\_prefix": "vocab", # 词汇表文件名前缀

"embed\_prefix": null, # 词嵌入文件名前缀，此处未使用预训练词嵌入

"out\_dir": "outputs", # 输出文件目录

"embedding\_size": 128, # 词嵌入向量大小

"unit\_type": "lstm", # RNN单元类型，此处使用LSTM

"num\_units": 256, # RNN隐藏单元个数

"forget\_bias": 1.0, # RNN偏置

"dropout": 0.2, # Dropout值，默认值为0.2

"encoder\_type": "bi", # Encoder类型，此处采用双向RNN

"num\_encoder\_layers": 2, # Encoder RNN层数

"num\_encoder\_residual\_layers": 0, # Encoder 残差层数，此处未使用

"num\_decoder\_layers": 2, # Decoder RNN层数

"num\_decoder\_residual\_layers": 0, # Decoder 残差层数

"time\_major": true, # 是否使用time major数据输入格式

"attention\_option": "scaled\_luong", # 注意力机制类型

"output\_attention": true, # 在解码的每一步输出attention值

"pass\_hidden\_state": true, # Attention state作为Decoder输入状态

"num\_train\_steps": 375000, #训练总步数，一步定义为一个mini-batch

"optimizer": "sgd", # 使用随机梯度下降

"learning\_rate": 1.0, # 学习率设置为1.0

"batch\_size": 32, # 由于显存限制，batch\_size设置为32

"num\_buckets": 10, # 把数据按长度投射到10个bucket

"var\_initializer": "uniform", # 使用uniform变量初始化

"init\_weight": 0.1, # 初始化权重0.1

"max\_gradient\_norm": 5.0, # 梯度截断阈值，默认为5.0

"decay\_scheme": "luong234", # 使用luong234学习率衰减策略

"src\_max\_len": 50, # 训练集源序列长度阈值，超过50将被过滤

"tgt\_max\_len": 50, # 训练集目标序列长度阈值

"src\_max\_len\_infer": null, # 推断源序列长度阈值

"tgt\_max\_len\_infer": null, # 推断目标序列长度阈值

"infer\_batch\_size": 32, # 推断batch\_size

"beam\_width": 5, # 用于beam\_search

"length\_penalty\_weight": 0, # 用于beam\_search

"sos": "<s>", # 目标序列起始字符

"eos": "</s>", # 目标序列结束字符

"steps\_per\_stats": 100, # 训练阶段每100步进行一次数据统计，log输出

"share\_vocab": false, # 源序列和目标序列是否使用同一词汇表

"random\_seed": null, # 随机种子

"num\_keep\_ckpts": 5 # 最大检查点数量

}

## 4.2调参

**1.LSTM隐层单元数：**一般默认为128，小数据量的情况下，100以下的隐层单元数也能的到很好的训练效果。隐层单元数越多，网络的容量越大，容易造成过拟合现象。大数据量情况下，Google NMT项目给出的建议是512或1024。

**2.LSTM Cell初始化器的设置：**对于单向LSTM来说，使用orthogonal初始化器一般来说会得到比较好的验证集误差，采用0初始化的效果一般比较差。初始化的不同所带来的差异在使用双向LSTM的时候会比较小。（但是实验中在使用orthogonal初始化的时候容易造成loss为Nan的情况，可以通过适当的减小学习率来解决）

**3.LSTM Cell的遗忘门偏置：**一般来说建议设置为默认的1.0，这在大多数情况下都能得到较好的效果，但是不是一定的，对于具体的任务可以尝试调整bias，尝试更大的bias。（实验的Forget Bias都设置为默认的1.0）

**4.双向LSTM：**单向LSTM只能根据过去的信息来预测后续的信息，而双向LSTM综合了过去和未来的信息，理论上可以获得更好的上下文信息。在实践中也确实如此，大多数情况下，双向LSTM的效果要比单项LSTM的效果好。双向LSTM是把两个方向的LSTM的状态信息综合输出，综合的方式有拼接（concat）和求和（sum）两种，实验用的是Sum方式。

**5.Word Embedding：**词嵌入的一般方式有两种：one-hot向量形式和预训练的词向量。词向量预训练采用word2vec，使用预训练的词向量在理论上比one-hot向量更能标识词语词之间的特征关联信息。小数据量下，预训练的词向量也有助于加快收敛速度

**6.Embedding Size：**即词向量维度大小，根据数据量的大小选择适当的size，可以从128开始调整，实验中使用的是200。

**7.Batch Size：**对于大量的数据，batch size一般采用32,64,128,256,512的序列尝试。但是在实验过程中发现，小数据量的情况下（10000为基准），batch size为1的时候，模型能够的到很好的收敛，并且所产生的生成序列效果较好，而在batch size大于1的时候，小数据量的情况下生成序列往往不佳。一种解释是batch size为1的时候，相当于起到了很好的正则作用（相当于对网络权重施加了更大的噪声，提升网络泛化能力）。个人的猜想可能是数据量较小的原因，也许在数据量极大的情况下较大的batchsize效果也会很好(Google NMT项目建议batchsize为128，前提是训练数据)

**8.优化器：**比较主流的优化器有SGD，RMSprop，Adam。Adam优化器能带来更快的收敛速度，RMSprop效果类似，SGD训练速度较慢。但是有建议在模型收敛的情况下，SGD的训练效果一般好于Adam，这点未做验证，为了优化速度，实际选用Adam优化器的情况居多。

**9.梯度截断（clip gradient）：**RNN网络容易出现梯度爆炸的现象，构建网络模型的时候一般需要进行梯度截断操作，来防止梯度爆炸对网络训练带来的负面影响。clip\_gradient的值一般设置为5。

**10.学习率：**从实验结果观察来看，学习率可以从0.001开始适当的缩小。可以采用手动方式调整，当训练loss达到一定的值不再缩小或者不降反升时，可以尝试停止训练，手动减小学习率再重新开始训练。减小的幅度可以视具体的情况而定，可以尝试每次除以2递减，也可以除以10递减等等，视具体效果而定。除了手动设置学习率外，也有一些自动的学习率衰减策略（decay scheme），如luong5, luong10, luong234等。

**11.正则化：**尝试加入dropout策略来提升网络泛化能力，dropout值可以设置略小一些，比如设置为0.2。前面提到batchsize设置为1也是一种类似正则化的策略（小数据量而言，在大数据量情况下由于训练速度太慢基本不可行）

**12.Attention机制：**attention机制的应用理论上可以使得序列模型能够在较长的序列学习上收到良好的效果。

1. **结果**

从训练集采样解码结果：

# Decoding sentence 151972

src: 你妹 我 擦

ref: 人家 错 了

bot: 人家 错 了

tim: 15.1437s

# Decoding sentence 177624

src: 你 又 吃 同类 啊 ? 真坏

ref: 你 都 把 动物 世界 吃遍 了

bot: 你 都 把 动物 世界 吃遍 了

tim: 15.2689s

# Decoding sentence 190115

src: 你 期待 爱情 麽

ref: = 。 =

bot: = 。 =

tim: 15.5499s

# Decoding sentence 2162

src: 蚯蚓 吃 吗

ref: 越 恶心 越 喜欢

bot: 越 恶心 越 喜欢

tim: 15.8068s

# Decoding sentence 62248

src: 无聊 就 数羊

ref: = 。 =

bot: = =

tim: 15.8970s

# Decoding sentence 92168

src: 怎么

ref: 白岑 刚才 不 小心 笨死了

bot: = =

tim: 16.1284s

# Decoding sentence 129006

src: 你 拿 去 孵小 黒 鸡

ref: 抱住 ！

bot: 抱住 ！

tim: 16.2156s

# Decoding sentence 55327

src: 居然 承认 了

ref: 我 承认 什么 了 啊 ?

bot: = =

tim: 16.2985s

# Decoding sentence 19690

src: 你 哪个 幼儿园 毕业 的

ref: 你 要是 毕业 了 我 就 毕业 了 。

bot: 你 要是 毕业 了 我 就 毕业 了 。

tim: 16.6249s

# Decoding sentence 31099

src: 你 下蛋 没

ref: 我 就 是从 蛋里 出来 的 ~ 你 好笨 哦 ~ ~ ( ￣ ▽ ￣ )

bot: 你 猜 ？

tim: 17.0538s

# Decoding sentence 50945

src: 怎么 认识 的

ref: 天下 谁 不 知道 啊 〜 我 主人 最好 的 朋友 〜 也 是 主人 以后 唯一 可能 喜欢 的 人 。

bot: 不 重要 的 人 啦 ~

tim: 17.1516s

# Decoding sentence 99848

src: 你 大爷

ref: 在 土里 吧 ！

bot: 嗻

tim: 17.2604s

# Decoding sentence 165753

src: 你 是 在 说 谢峰 吗

ref: 废话 … 她 不 就是 那个 爱好 打麻将 的 吗 ！

bot: 废话 … 她 不 就是 那个 爱好 打麻将 的 爱好 吗

tim: 11.5659s

# Decoding sentence 48119

src: 小 … 小黄 … 小通 … 小通 … 小通 … 呼叫

ref: 诶 。 我 在 我 在 。

bot: 臣 在 〜

tim: 9.7380s

测试集上的推断结果：

sentence 103

src: 给 你 看

ref: 我 看

bot: = 。 =

sentence 104

src: 对 , 你 可耻

ref: 没错 ! 李浩 就是 猪头 白痴 山炮

bot: = 。 =

sentence 105

src: 不玩

ref: 不玩 学习 去

bot: 好 吧 ， 我 在 呢

sentence 106

src: 你 怎么 不 学习

ref: 等到 夜深人静 的 时候 我 再 悄悄地 给 你 说 一个 秘密 ^ \_ ^

bot: 我 一直 在 思念 我 的 娜娜 女王 。 。

sentence 108

src: 你 是 火鸡 吗

ref: 对 呀 ， 我 是 鸡娃 ！ 比 忘 了 ？

bot: 我 是 卖报 的 小 行家

sentence 109

src: 轻轻地 我 将 离开 你

ref: 请 将 眼角 的 泪 拭去

bot: 吻吻 , 宝贝 kiss kiss , kiss you ,

sentence 111

src: 你 才 放屁

ref: 没有 这是 事实 人人 网上 的 哥哥 姐姐 告诉 我 的

bot: 你 人参 公鸡 ！

sentence 112

src: 谁 告诉 你 的 , 说

ref: 冬瓜 说 的 。 。 他 告诉 我 的

bot: 是 你 这辈子 <unk> 的 人 啊 ！

sentence 113

src: 春天 在 哪里 啊 春天 在 哪里

ref: 在 我 的 眼睛 里 ！ ！ ！ 哈哈哈 ！ ！ ！

bot: 春天 在 那 小朋友 的 眼睛 里 ， 春天 在 寻找 的 眼睛 里

sentence 115

src: 你 是 战斗机

ref: 有 瀑布 大 吗 ？

bot: 毒舌攻 （ ≧ ∇ ≦ ） 〜 〜 〜

sentence 116

src: 你 说话 我 听不懂

ref: 他 也 在 想 你

bot: = =

sentence 117

src: 你 自己 说

ref: 我 爱 主任

bot: @ 小通 ( \* \* \* \* \* \* ) 你好 无聊 啊

sentence 118

src: 因为 你 又 犯病 了

ref: 恩 啊 恩 啊

bot: 呜呜 … 主人 你 发现 了 ？

sentence 119

src: 好 吧 , 你 是 挺 可爱 的

ref: 我 不是 ， 你 才 是

bot: 谢谢 夸奖

sentence 120

src: 你 几点 睡 的

ref: 我 有 一颗 永不 眠 的 耳 粑 粑 〜

bot: 你 不 完 ! 我 睡不着 !

sentence 121

src: 你 几点 起 的

ref: 马上

bot: 嘿嘿嘿 嘿嘿 嘿嘿 嘿嘿 嘿嘿 嘿嘿 你 说 呢 ～ ～ ～

sentence 93

src: 别 呵 了

ref: 傻笑 什么

bot: 呵呵 呵呵

sentence 96

src: 还是 小 傻 鸡

ref: 诶 。 就是 我

bot: = =

sentence 97

src: 不会 的 问题 可以 问 你 吗

ref: 爱过

bot: 问 吧 ， 我 不会 告诉 你 我 是 雷锋 的 。

其中，src是测试集的问句，ref是测试集的答句，bot是模型推断的结果。从以上结果可以看出，在大数据集的情况下，测试集上推断的表现勉强能接受，但是仍然有很多不自然的地方，还有一些文不对题的情况。可行的解决方法是集训扩大语料库，调整训练参数在更大的数据集上训练。

1. **总结**

以上的工作是对seq2seq模型相关理论的汇总及实现的简要说明。在本项工作中，根据已有的关于seq2seq和chatbot的相关理论，以及Google NMT开源项目，构建了一个简单的基于深度学习技术的聊天机器人。

Seq2seq自提出就受到了广泛的关注，基于初始的seq2seq模型，后续有很多工作都对其提出了改进，在以上的汇总信息中也有所体现。其中双向RNN的引入对训练效果的提升较大，Attention机制的引入能够改善长序列的推断效果。

通过在小黄鸡聊天数据集上的训练，结合其他人的一些经验，本文还对seq2seq训练过程中的一些技巧进行了简单的罗列，不能保证完全正确，但至少在我训练的过程中确实有些效果。

以上的工作只是介绍了基于seq2seq的chatbot的简单实现。仍有许多可以提升的地方，例如：

1. 引入记忆网络，显式的记住某些特征，把记忆长期保留。在推断的时候直接在已有的答案空间搜索，就像人类在回答某些问题时能够熟练地从记忆中获取答案一样。
2. 使用知识图谱，让模型具备从外部获取信息的能力，就像语音助手可以熊网络搜索信息来回答一样。
3. 尝试新的更复杂的建模技术，如多轮对话控制、情感控制、语义理解方面的新的研究成果等。