

目录

- [1. 什么是NLP](#)
- [2. NLP主要研究方向](#)
- [3. NLP的发展](#)
- [4. NLP任务的一般步骤](#)
- [5. 我的NLP启蒙读本](#)
- [6. NLP、CV，选哪个？](#)

1. 什么是NLP

自然语言处理 (Natural Language Processing) 是人工智能 (AI) 的一个子领域。自然语言处理是研究在人与人交互中以及在人与计算机交互中的语言问题的一门学科。为了建设和完善语言模型，自然语言处理建立计算框架，提出相应的方法来不断的完善设计各种实用系统，并探讨这些实用系统的评测方法。

2. NLP主要研究方向

1. **信息抽取**: 从给定文本中抽取重要的信息，比如时间、地点、人物、事件、原因、结果、数字、日期、货币、专有名词等等。通俗说来，就是要了解谁在什么时候、什么原因、对谁、做了什么事、有什么结果。
2. **文本生成**: 机器像人一样使用自然语言进行表达和写作。依据输入的不同，文本生成技术主要包括数据到文本生成和文本到文本生成。数据到文本生成是指将包含键值对的数据转化为自然语言文本；文本到文本生成对输入文本进行转化和处理从而产生新的文本。
3. **问答系统**: 对一个自然语言表达的问题，由问答系统给出一个精准的答案。需要对自然语言查询语句进行某种程度的语义分析，包括实体链接、关系识别，形成逻辑表达式，然后到知识库中查找可能的候选答案并通过一个排序机制找出最佳的答案。
4. **对话系统**: 系统通过一系列的对话，跟用户进行聊天、回答、完成某一项任务。涉及到用户意图理解、通用聊天引擎、问答引擎、对话管理等技术。此外，为了体现上下文相关，要具备多轮对话能力。
5. **文本挖掘**: 包括文本聚类、分类、情感分析以及对挖掘的信息和知识的可视化、交互式的表达界面。目前主流的技术都是基于统计机器学习的。
6. **语音识别和生成**: 语音识别是将输入计算机的语音符号识别转换成书面语表示。语音生成又称文语转换、语音合成，它是指将书面文本自动转换成对应的语音表征。
7. **信息过滤**: 通过计算机系统自动识别和过滤符合特定条件的文档信息。通常指网络有害信息的自动识别和过滤，主要用于信息安全和防护，网络内容管理等。
8. **舆情分析**: 是指收集和处理海量信息，自动化地对网络舆情进行分析，以实现及时应对网络舆情的目的。
9. **信息检索**: 对大规模的文档进行索引。可简单对文档中的词汇，赋之以不同的权重来建立索引，也可建立更加深层的索引。在查询的时候，对输入的查询表达式比如一个检索词或者一个句子进行分析，然后在索引里面查找匹配的候选文档，再根据一个排序机制把候选文档排序，最后输出排序得分最高的文档。
10. **机器翻译**: 把输入的源语言文本通过自动翻译获得另外一种语言的文本。机器翻译从最早的基于规则的方法到二十年前的基于统计的方法，再到今天的基于神经网络（编码-解码）的方法，逐渐形成了一套比较严谨的方法体系。

3. NLP的发展

1. 1950年前：图灵测试

1950年前阿兰·图灵图灵测试：人和机器进行交流，如果人无法判断自己交流的对象是人还是机器，就说明这个机器具有智能。

2. 1950-1970：主流：基于规则形式语言理论

乔姆斯基，根据数学中的公理化方法研究自然语言，采用代数和集合论把形式语言定义为符号的序列。他试图使用有限的规则描述无限的语言现象，发现人类普遍的语言机制，建立所谓的普遍语法。

3. 1970-至今：主流：基于统计

谷歌、微软、IBM，20世纪70年代，弗里德里克·贾里尼克及其领导的IBM华生实验室将语音识别率从70%提升到90%。

1988年，IBM的彼得·布朗提出了基于统计的机器翻译方法。

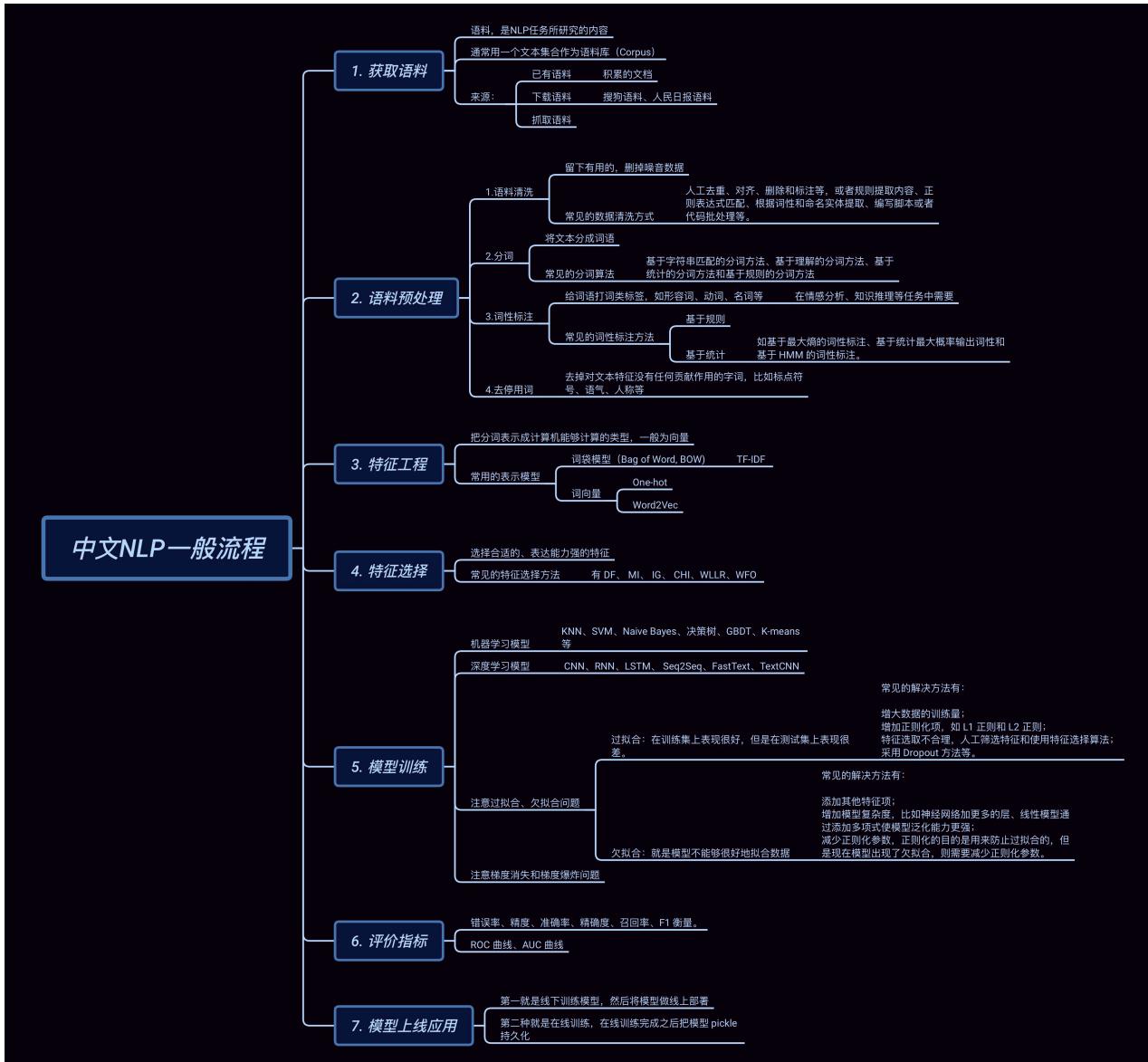
2005年，Google机器翻译打败基于规则的Sys Tran。

4. 2010年以后：逆袭：机器学习

AlphaGo先后战胜李世石、柯洁等，掀起人工智能热潮。深度学习、人工神经网络成为热词。领域：语音识别、图像识别、机器翻译、自动驾驶、智能家居。

4. NLP任务的一般步骤

下面图片看不清楚的，可以百度脑图查看，[点击链接](#)



5. 我的NLP启蒙读本

[《数学之美》--吴军](#)

6. NLP、CV，选哪个？

NLP: 自然语言处理，数据是文本。

CV: 计算机视觉，数据是图像。

两者属于不同的领域，在遇到这个问题的时候，我也是犹豫了很久，想了很多，于是乎得出一个结论：
都是利用深度学习去解决现实世界存在的问题，离开了CV，NLP存活不了；离开了NLP，CV存活不了。
两者就像兄弟姐妹一样，整个“家庭”不能分割但个体又存在差异！

NLP/CV属于两个不同的研究领域，都是很好的领域，可以根据自己的爱好作出适合自己的选择，人工智能是一个多学科交叉的领域，需要的不仅仅是单方面的能力，而是多方面的能力。对于每个人来说都有自己的侧重点，毕竟人的精力是有限的。**只要在自己擅长的领域里持续深耕，我相信都会有所成就！**

这里提供一些参考资料给大家阅读阅读，做出适合自己的选择：

- [一文看尽2018全年AI技术大突破：NLP跨过分水岭、CV研究效果惊人](#)

- [《数学之美》--吴军](#)
 - [BERT时代与后时代的NLP](#)
-

作者: [@mantchs](#)

GitHub: <https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论! 共同完善此项目! 群号: 【541954936】  加入QQ群

目录

- [1. 什么是词嵌入\(Word Embedding\)](#)
- [2. 离散表示](#)
 - [2.1 One-hot表示](#)
 - [2.2 词袋模型](#)
 - [2.3 TF-IDF](#)
 - [2.4 n-gram模型](#)
 - [2.5 离散表示存在的问题](#)
- [3. 分布式表示](#)
 - [3.1 共现矩阵](#)
- [4. 神经网络表示](#)
 - [4.1 NNLM](#)
 - [4.2 Word2Vec](#)
 - [4.3 sense2vec](#)
- [5. 词嵌入为何不采用one-hot向量](#)
- [6. Word2Vec代码实现](#)

1. 什么是词嵌入(Word Embedding)

自然语言是一套用来表达含义的复杂系统。在这套系统中，词是表义的基本单元。顾名思义，词向量是用来表示词的向量，也可被认为是词的特征向量或表征。把词映射为实数域向量的技术也叫词嵌入（**word embedding**）。近年来，词嵌入已逐渐成为自然语言处理的基础知识。

在NLP(自然语言处理)领域，文本表示是第一步，也是很重要的一步，通俗来说就是把人类的语言符号转化为机器能够进行计算的数字，因为普通的文本语言机器是看不懂的，必须通过转化来表征对应文本。早期是基于规则的方法进行转化，而现代的方法是基于统计机器学习的方法。

数据决定了机器学习的上限,而算法只是尽可能逼近这个上限，在本文中数据指的就是文本表示，所以，弄懂文本表示的发展历程，对于NLP学习者来说是必不可少的。接下来开始我们的发展历程。文本表示分为离散表示和分布式表示：

2. 离散表示

2.1 One-hot表示

One-hot简称读热向量编码，也是特征工程中最常用的方法。其步骤如下：

1. 构造文本分词后的字典，每个分词是一个比特值，比特值为0或者1。
2. 每个分词的文本表示为该分词的比特位为1，其余位为0的矩阵表示。

例如：John likes to watch movies. Mary likes too

John also likes to watch football games.

以上两句可以构造一个词典，{"John": 1, "likes": 2, "to": 3, "watch": 4, "movies": 5, "also": 6, "football": 7, "games": 8, "Mary": 9, "too": 10}

每个词典索引对应着比特位。那么利用One-hot表示为：

John: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

likes: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]等等，以此类推。

One-hot表示文本信息的缺点：

- 随着语料库的增加，数据特征的维度会越来越大，产生一个维度很高，又很稀疏的矩阵。
- 这种表示方法的分词顺序和在句子中的顺序是无关的，不能保留词与词之间的关系信息。

2.2 词袋模型

词袋模型(Bag-of-words model)，像是句子或是文件这样的文字可以用一个袋子装着这些词的方式表现，这种表现方式不考虑文法以及词的顺序。

文档的向量表示可以直接将各词的词向量表示加和。例如：

John likes to watch movies. Mary likes too

John also likes to watch football games.

以上两句可以构造一个词典， {"John": 1, "likes": 2, "to": 3, "watch": 4, "movies": 5, "also": 6, "football": 7, "games": 8, "Mary": 9, "too": 10}

那么第一句的向量表示为：[1,2,1,1,1,0,0,0,1,1]，其中的2表示**likes**在该句中出现了2次，依次类推。

词袋模型同样有一下缺点：

- 词向量化后，词与词之间是有大小关系的，不一定词出现的越多，权重越大。
- 词与词之间是没有顺序关系的。

2.3 TF-IDF

TF-IDF (term frequency-inverse document frequency) 是一种用于信息检索与数据挖掘的常用加权技术。TF意思是词频(Term Frequency)，IDF意思是逆文本频率指数(Inverse Document Frequency)。

字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。一个词语在一篇文章中出现次数越多，同时在所有文档中出现次数越少，越能够代表该文章。

$$TF_w = \frac{\text{在某一类中词条 } w \text{ 出现的次数}}{\text{该类中所有的词条数目}}$$

$$IDF = \log\left(\frac{\text{语料库的文档总数}}{\text{包含词条 } w \text{ 的文档总数} + 1}\right)$$

分母之所以加1，是为了避免分母为0。

那么， $TF - IDF = TF * IDF$ ，从这个公式可以看出，当w在文档中出现的次数增大时，而TF-IDF的值是减小的，所以也就体现了以上所说的了。

缺点：还是没有把词与词之间的关系顺序表达出来。

2.4 n-gram模型

n-gram模型为了保持词的顺序，做了一个滑窗的操作，这里的n表示的就是滑窗的大小，例如2-gram模型，也就是把2个词当做一组来处理，然后向后移动一个词的长度，再次组成另一组词，把这些生成一个字典，按照词袋模型的方式进行编码得到结果。该模型考虑了词的顺序。

例如：

John likes to watch movies. Mary likes too

John also likes to watch football games.

以上两句可以构造一个词典，{"John likes": 1, "likes to": 2, "to watch": 3, "watch movies": 4, "Mary likes": 5, "likes too": 6, "John also": 7, "also likes": 8, "watch football": 9, "football games": 10}

那么第一句的向量表示为：[1, 1, 1, 1, 1, 1, 0, 0, 0, 0]，其中第一个1表示**John likes**在该句中出现了1次，依次类推。

缺点：随着n的大小增加，词表会成指数型膨胀，会越来越大。

2.5 离散表示存在的问题

由于存在以下的问题，对于一般的NLP问题，是可以使用离散表示文本信息来解决问题的，但对于要求精度较高的场景就不适合了。

- 无法衡量词向量之间的关系。
- 词表的维度随着语料库的增长而膨胀。
- n-gram词序列随语料库增长呈指数型膨胀，更加快。
- 离散数据来表示文本会带来数据稀疏问题，导致丢失了信息，与我们生活中理解的信息是不一样的。

3. 分布式表示

科学家们为了提高模型的精度，又发明出了分布式的表示文本信息的方法，这就是这一节需要介绍的。

用一个词附近的其它词来表示该词，这是现代统计自然语言处理中最有创见的想法之一。当初科学家发明这种方法是基于人的语言表达，认为一个词是由这个词的周边词汇一起来构成精确的语义信息。就好比，物以类聚人以群分，如果你想了解一个人，可以通过他周围的人进行了解，因为周围人都有一些共同点才能聚集起来。

3.1 共现矩阵

共现矩阵顾名思义就是共同出现的意思，词文档的共现矩阵主要用于发现主题(topic)，用于主题模型，如LSA。

局域窗中的word-word共现矩阵可以挖掘语法和语义信息，例如：

- I like deep learning.
- I like NLP.
- I enjoy flying

有以上三句话，设置滑窗为2，可以得到一个词典：{"I like","like deep","deep learning","like NLP","I enjoy","enjoy flying","I like"}。

我们可以得到一个共现矩阵(对称矩阵)：

中间的每个格子表示的是行和列组成的词组在词典中共同出现的次数，也就体现了共现的特性。

存在的问题：

- 向量维数随着词典大小线性增长。
- 存储整个词典的空间消耗非常大。
- 一些模型如文本分类模型会面临稀疏性问题。
- 模型会不稳定，每新增一份语料进来，稳定性就会变化。

4. 神经网络表示

4.1 NNLM

NNLM (Neural Network Language model)，神经网络语言模型是03年提出来的，通过训练得到中间产物--词向量矩阵，这就是我们要得到的文本表示向量矩阵。

NNLM说的是定义一个前向窗口大小，其实和上面提到的窗口是一个意思。把这个窗口中最后一个词当做y，把之前的词当做输入x，通俗来说就是预测这个窗口中最后一个词出现概率的模型。

以下是NNLM的网络结构图：

- input层是一个前向词的输入，是经过one-hot编码的词向量表示形式，具有 $V \times 1$ 的矩阵。
- C矩阵是投影矩阵，也就是稠密词向量表示，在神经网络中是w参数矩阵，该矩阵的大小为 $D \times V$ ，正好与input层进行全连接(相乘)得到 $D \times 1$ 的矩阵，采用线性映射将one-hot表示投影到稠密D维表示。
- output层(softmax)自然是前向窗中需要预测的词。
- 通过BP + SGD得到最优的C投影矩阵，这就是NNLM的中间产物，也是我们所求的文本表示矩阵，通过NNLM将稀疏矩阵投影到稠密向量矩阵中。

4.2 Word2Vec

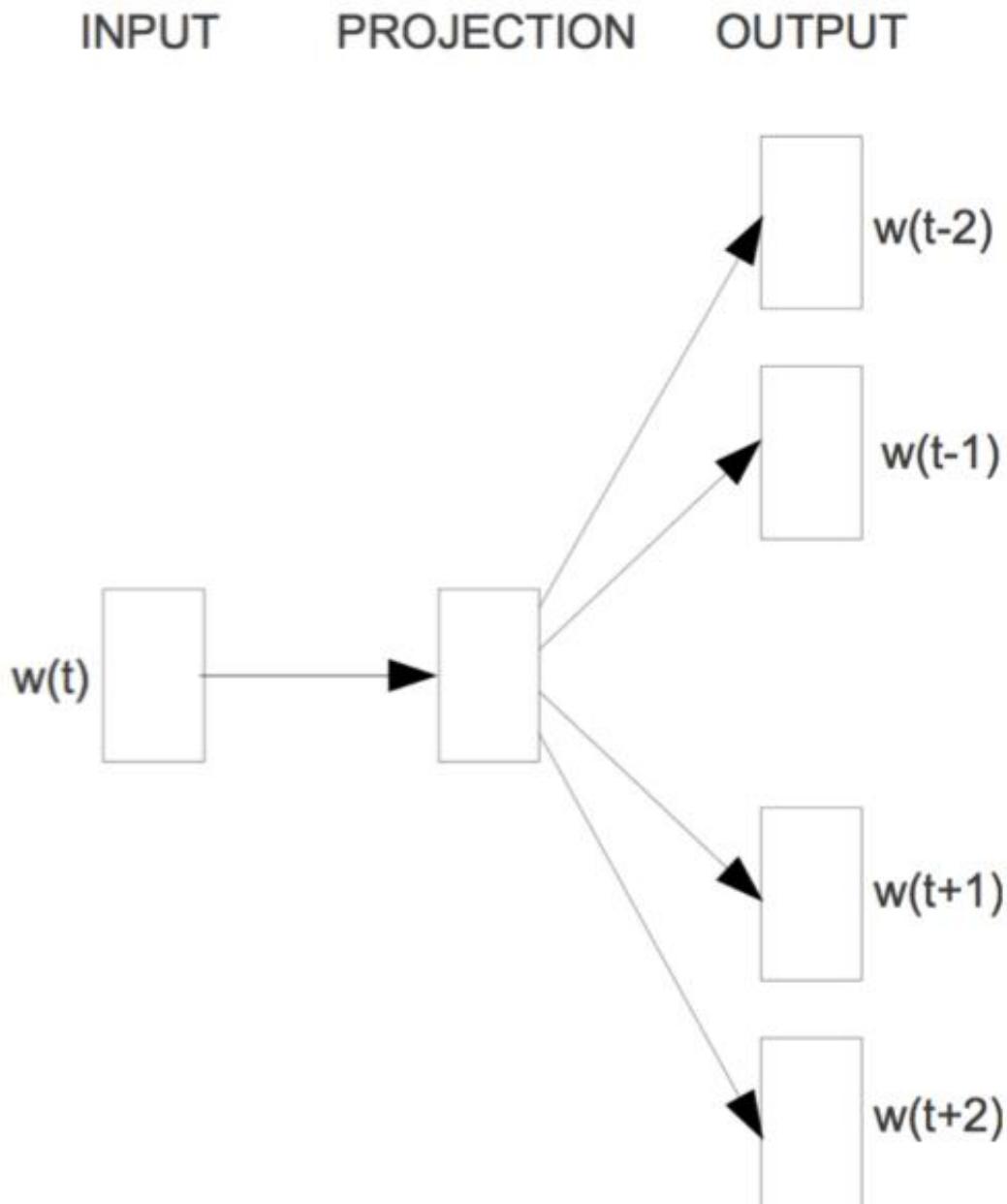
谷歌2013年提出的Word2Vec是目前最常用的词嵌入模型之一。Word2Vec实际是一种浅层的神经网络模型，它有两种网络结构，分别是**CBOW (Continues Bag of Words)** 连续词袋和**Skip-gram**。Word2Vec和上面的NNLM很类似，但比NNLM简单。

CBOW

CBOW获得中间词两边的上下文，然后用周围的词去预测中间的词，把中间词当做y，把窗口中的其它词当做x输入，x输入是经过one-hot编码过的，然后通过一个隐层进行求和操作，最后通过激活函数softmax，可以计算出每个单词的生成概率，接下来的任务就是训练神经网络的权重，使得语料库中所有单词的整体生成概率最大化，而求得的权重矩阵就是文本表示词向量的结果。

Skip-gram：

Skip-gram是通过当前词来预测窗口中上下文词出现的概率模型，把当前词当做x，把窗口中其它词当做y，依然是通过一个隐层接一个Softmax激活函数来预测其它词的概率。如下图所示：



优化方法：

- 层次Softmax：至此还没有结束，因为如果单单只是接一个softmax激活函数，计算量还是很大的，有多少词就会有多少维的权重矩阵，所以这里就提出层次Softmax(Hierarchical Softmax)，使用Huffman Tree来编码输出层的词典，相当于平铺到各个叶子节点上，瞬间把维度降低到了树的深度，可以看如下图所示。这棵Tree把出现频率高的词放到靠近根节点的叶子节点处，每一次只要做二分类计算，计算路径上所有非叶子节点词向量的贡献即可。

哈夫曼树(Huffman Tree)：给定N个权值作为N个叶子结点，构造一棵二叉树，若该树的带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

- 负例采样(Negative Sampling)：这种优化方式做的事情是，在正确单词以外的负样本中进行采样，最终目的是为了减少负样本的数量，达到减少计算量效果。将词典中的每一个词对应一条线段，所有词组成了[0, 1]间的部分，如下图所示，然后每次随机生成一个[1, M-1]间的整数，看落在哪个词对应的部分上就选择哪个词，最后会得到一个负样本集合。

Word2Vec存在的问题

- 对每个local context window单独训练，没有利用包含在global co-currence矩阵中的统计信息。
- 对多义词无法很好的表示和处理，因为使用了唯一的词向量

4.3 sense2vec

word2vec模型的问题在于词语的多义性。比如duck这个单词常见的含义有水禽或者下蹲，但对于word2vec模型来说，它倾向于将所有概念做归一化平滑处理，得到一个最终的表现形式。

5. 词嵌入为何不采用one-hot向量

虽然one-hot词向量构造起来很容易，但通常并不是一个好选择。一个主要的原因是，one-hot词向量无法准确表达不同词之间的相似度，如我们常常使用的余弦相似度。由于任何两个不同词的one-hot向量的余弦相似度都为0，多个不同词之间的相似度难以通过onehot向量准确地体现出来。

word2vec工具的提出正是为了解决上面这个问题。它将每个词表示成一个定长的向量，并使得这些向量能较好地表达不同词之间的相似和类比关系。

6. Word2Vec代码实现

数据下载

中文维基百科的打包文件地址为链接: https://pan.baidu.com/s/1H-wulve0d_fvczvy3EOKMQ 提取码: uqua

百度网盘加速下载地址: <https://www.baiduwp.com/?m=index>

[Word2Vec训练维基百科文章代码](#)

作者:[@mantchs](#)

GitHub:<https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论！共同完善此项目！群号：【541954936】



目录

- [1. 什么是fastText](#)
- [2. n-gram表示单词](#)
- [3. fastText模型架构](#)
- [4. fastText核心思想](#)
- [5. 输出分类的效果](#)
- [6. fastText与Word2Vec的不同](#)
- [7. 代码实现](#)
- [8. 参考文献](#)

1. 什么是fastText

英语单词通常有其内部结构和形成方式。例如，我们可以从“dog”“dogs”和“dogcatcher”的字面上推测它们的关系。这些词都有同一个词根“dog”，但使用不同的后缀来改变词的含义。而且，这个关联可以推广至其他词汇。

在word2vec中，我们并没有直接利用构词学中的信息。无论是在跳字模型还是连续词袋模型中，我们都将形态不同的单词用不同的向量来表示。例如，“**dog**”和“**dogs**”分别用两个不同的向量表示，而模型中并未直接表达这两个向量之间的关系。鉴于此，**fastText**提出了子词嵌入(**subword embedding**)的方法，从而试图将构词信息引入**word2vec**中的**CBOW**。

这里有一点需要特别注意，一般情况下，使用fastText进行文本分类的同时也会产生词的embedding，即embedding是fastText分类的产物。除非你决定使用预训练的embedding来训练fastText分类模型，这另当别论。

2. n-gram表示单词

word2vec把语料库中的每个单词当成原子的，它会为每个单词生成一个向量。这忽略了单词内部的形态特征，比如：“book” 和“books”，“阿里巴巴”和“阿里”，这两个例子中，两个单词都有较多公共字符，即它们的内部形态类似，但是在传统的word2vec中，这种单词内部形态信息因为它们被转换成不同的id丢失了。

为了克服这个问题，**fastText**使用了字符级别的**n-grams**来表示一个单词。对于单词“book”，假设n的取值为3，则它的trigram有：

“**<bo**”，“**bo**”，“**oo**”，“**ok**”，“**ok>**”

其中，<表示前缀，>表示后缀。于是，我们可以用这些trigram来表示“book”这个单词，进一步，我们可以用这4个trigram的向量叠加来表示“apple”的词向量。

这带来两点好处：

1. 对于低频词生成的词向量效果会更好。因为它们的n-gram可以和其它词共享。
2. 对于训练词库之外的单词，仍然可以构建它们的词向量。我们可以叠加它们的字符级n-gram向量。

3. fastText模型架构

之前提到过，fastText模型架构和word2vec的CBOW模型架构非常相似。下面是fastText模型架构图：

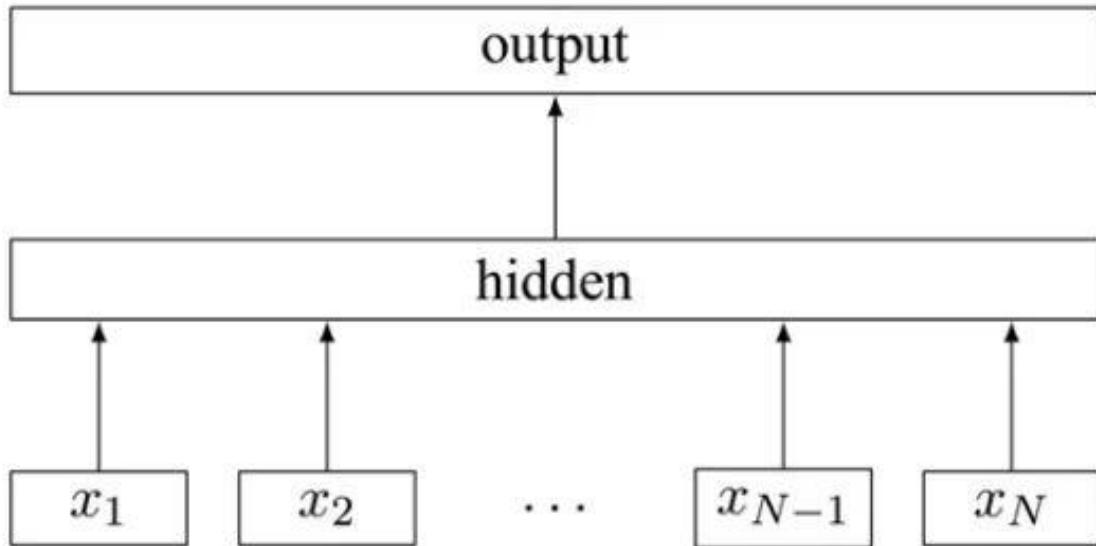


Figure 1: Model architecture of fastText for a sentence with N ngram features x_1, \dots, x_N . The features are embedded and averaged to form the hidden variable.

注意：此架构图没有展示词向量的训练过程。可以看到，和CBOW一样，fastText模型也只有三层：输入层、隐含层、输出层（Hierarchical Softmax），输入都是多个经向量表示的单词，输出都是一个特定的target，隐含层都是对多个词向量的叠加平均。

不同的是，

- CBOW的输入是目标单词的上下文，fastText的输入是多个单词及其n-gram特征，这些特征用来表示单个文档；
- CBOW的输入单词被one-hot编码过，fastText的输入特征是被embedding过；
- CBOW的输出是目标词汇，fastText的输出是文档对应的类标。

值得注意的是，fastText在输入时，将单词的字符级别的n-gram向量作为额外的特征；在输出时，fastText采用了分层Softmax，大大降低了模型训练时间。这两个知识点在前文中已经讲过，这里不再赘述。

fastText相关公式的推导和CBOW非常类似，这里也不展开了。

4. fastText核心思想

现在抛开那些不是很讨人喜欢的公式推导，来想一想fastText文本分类的核心思想是什么？

仔细观察模型的后半部分，即从隐含层输出到输出层输出，会发现它就是一个softmax线性多类别分类器，分类器的输入是一个用来表征当前文档的向量；

模型的前半部分，即从输入层输入到隐含层输出部分，主要在做一件事情：生成用来表征文档的向量。那么它是如何做的呢？**叠加构成这篇文档的所有词及n-gram的词向量，然后取平均。**叠加词向量背后的思想就是传统的词袋法，即将文档看成一个由词构成的集合。

于是**fastText**的核心思想就是：**将整篇文档的词及n-gram向量叠加平均得到文档向量，然后使用文档向量做softmax多分类。**这中间涉及到两个技巧：字符级n-gram特征的引入以及分层Softmax分类。

5. 输出分类的效果

还有个问题，就是为何**fastText**的分类效果常常不输于传统的非线性分类器？

假设我们有两段文本：

肚子 饿了 我 要 吃饭

肚子 饿了 我 要 吃东西

这两段文本意思几乎一模一样，如果要分类，肯定要分到同一个类中去。但在传统的分类器中，用来表征这两段文本的向量可能差距非常大。传统的文本分类中，你需要计算出每个词的权重，比如TF-IDF值，“吃饭”和“吃东西”算出的TF-IDF值相差可能会比较大，其它词类似，于是，VSM（向量空间模型）中用来表征这两段文本的文本向量差别可能比较大。

但是**fastText**就不一样了，它是用单词的**embedding**叠加获得的文档向量，词向量的重要特点就是向量的距离可以用来衡量单词间的语义相似程度，于是，在**fastText**模型中，这两段文本的向量应该是非常相似的，于是，它们很大概率会被分到同一个类中。

使用词**embedding**而非词本身作为特征，这是**fastText**效果好的一个原因；另一个原因就是字符级n-gram特征的引入对分类效果会有一些提升。

6. fastText与Word2Vec的不同

有意思的是，**fastText**和**Word2Vec**的作者是同一个人。

相同点：

- 图模型结构很像，都是采用**embedding**向量的形式，得到word的隐向量表达。
- 都采用很多相似的优化方法，比如使用Hierarchical softmax优化训练和预测中的打分速度。

之前一直不明白**fasttext**用层次softmax时叶子节点是啥，CBOW很清楚，它的叶子节点是词和词频，后来看了源码才知道，其实**fasttext**叶子节点里是类标和类标的频数。

	Word2Vec	fastText
输入	one-hot形式的单词的向量	embedding过的单词的词向量和n-gram向量
输出	对应的是每一个term,计算某term概率最大	对应的是分类的标签。

本质不同，体现在softmax的使用：

word2vec的目的是得到词向量，该词向量最终是在输入层得到的，输出层对应的h-softmax也会生成一系列的向量，但是最终都被抛弃，不会使用。

fastText则充分利用了h-softmax的分类功能，遍历分类树的所有叶节点，找到概率最大的label

fastText优点：

1. **适合大型数据+高效的训练速度：**能够训练模型“在使用标准多核CPU的情况下10分钟内处理超过10亿个词汇”
2. **支持多语言表达：**利用其语言形态结构，fastText能够被设计用来支持包括英语、德语、西班牙语、法语以及捷克语等多种语言。FastText的性能要比时下流行的word2vec工具明显好上不少，也比其他目前最先进的词态词汇表征要好。
3. **专注于文本分类，**在许多标准问题上实现当下最好的表现（例如文本倾向性分析或标签预测）。

7. 代码实现

清华文本分类数据集下载：<https://thunlp.oss-cn-qingdao.aliyuncs.com/THUCNews.zip>

[新闻文本分类代码](#)

8. 参考文献

[fastText原理及实践](#)

作者：[@mantchs](#)

GitHub：<https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论！共同完善此项目！群号：【541954936】 加入QQ群

目录

- [1. 说说GloVe](#)
- [2. GloVe的实现步骤](#)
 - [2.1 构建共现矩阵](#)
 - [2.2 词向量和共现矩阵的近似关系](#)
 - [2.3 构造损失函数](#)
 - [2.4 训练GloVe模型](#)
- [3. GloVe与LSA、Word2Vec的比较](#)
- [4. 代码实现](#)
- [5. 参考文献](#)

1. 说说GloVe

正如GloVe论文的标题而言，**GloVe**的全称叫**Global Vectors for Word Representation**，它是一个基于全局词频统计（count-based & overall statistics）的词表征（word representation）工具，它可以把一个单词表达成一个由实数组成的向量，这些向量捕捉到了单词之间一些语义特性，比如相似性（similarity）、类比性（analogy）等。我们通过对向量的运算，比如欧几里得距离或者cosine相似度，可以计算出两个单词之间的语义相似性。

2. GloVe的实现步骤

2.1 构建共现矩阵

什么是共现矩阵？

共现矩阵顾名思义就是共同出现的意思，词文档的共现矩阵主要用于发现主题(topic)，用于主题模型，如LSA。

局域窗中的word-word共现矩阵可以挖掘语法和语义信息，例如：

- I like deep learning.
- I like NLP.
- I enjoy flying

有以上三句话，设置滑窗为2，可以得到一个词典：{"I like", "like deep", "deep learning", "like NLP", "I enjoy", "enjoy flying", "I like"}。

我们可以得到一个共现矩阵(对称矩阵)：

中间的每个格子表示的是行和列组成的词组在词典中共同出现的次数，也就体现了共现的特性。

GloVe的共现矩阵

根据语料库 (corpus) 构建一个共现矩阵 (Co-occurrence Matrix) X , 矩阵中的每一个元素 X_{ij} 代表单词 i 和上下文单词 j 在特定大小的上下文窗口 (context window) 内共同出现的次数。一般而言, 这个次数的最小单位是1, 但是GloVe不这么认为: 它根据两个单词在上下文窗口的距离 d , 提出了一个衰减函数 (decreasing weighting) : $\text{decay}=1/d$ 用于计算权重, 也就是说距离越远的两个单词所占总计数 (total count) 的权重越小。

2.2 词向量和共现矩阵的近似关系

构建词向量 (Word Vector) 和共现矩阵 (Co-occurrence Matrix) 之间的近似关系, 论文的作者提出以下的公式可以近似地表达两者之间的关系:

$$w_i^T \tilde{w}_j + b_i + \tilde{b}_j = \log(X_{ij})$$

其中, $w_i^T \tilde{w}_j$ 是我们最终要求解的词向量; b_i 分别是两个词向量的bias term。当然你对这个公式一定有非常多的疑问, 比如它到底是怎么来的, 为什么要使用这个公式, 为什么要构造两个词向量 $w_i^T \tilde{w}_j$? 请参考文末的参考文献。

2.3 构造损失函数

有了2.2的公式之后我们就可以构造它的loss function了:

$$J = \sum_{i,j=1}^V f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij}))^2$$

这个loss function的基本形式就是最简单的mean square loss, 只不过在此基础上加了一个权重函数 $f(X_{ij})$, 那么这个函数起了什么作用, 为什么要添加这个函数呢? 我们知道在一个语料库中, 肯定存在很多单词他们在一起出现的次数是很多的 (frequent co-occurrences), 那么我们希望:

- 这些单词的权重要大于那些很少在一起出现的单词 (rare co-occurrences), 所以这个函数要是非递减函数 (non-decreasing);
- 但我们也希望这个权重过大 (overweighted), 当到达一定程度之后应该不再增加;
- 如果两个单词没有在一起出现, 也就是 $X_{ij} = 0$, 那么他们应该不参与到 loss function 的计算当中去, 也就是 $f(x)$ 要满足 $f(0)=0$ 。

满足以上三个条件的函数有很多, 论文作者采用了如下形式的分段函数:

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

这个函数图像如下所示:

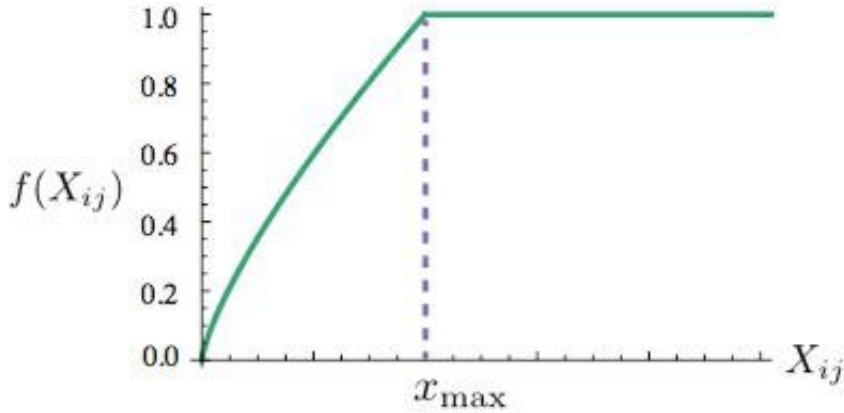


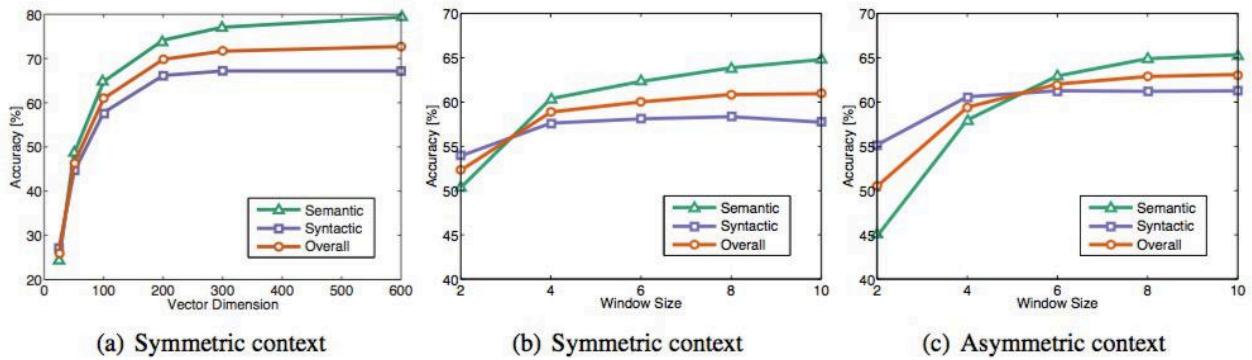
Figure 1: Weighting function f with $\alpha = 3/4$.

2.4 训练GloVe模型

虽然很多人声称GloVe是一种无监督（unsupervised learning）的学习方式（因为它确实不需要人工标注label），但其实它还是有label的，这个label就是以上公式中的 $\log(X_{ij})$ ，而公式中的向量 w 和 \tilde{w} 就是要不断更新/学习的参数，所以本质上它的训练方式跟监督学习的训练方法没什么不一样，都是基于梯度下降的。

具体地，这篇论文里的实验是这么做的：采用了AdaGrad的梯度下降算法，对矩阵 X 中的所有非零元素进行随机采样，学习率（learning rate）设为0.05，在vector size小于300的情况下迭代了50次，其他大小的vectors上迭代了100次，直至收敛。最终学习得到的是两个vector是 w 和 \tilde{w} ，因为 X 是对称的（symmetric），所以从原理上讲 w 和 \tilde{w} 也是对称的，他们唯一的区别是初始化的值不一样，而导致最终的值不一样。

所以这两者其实是等价的，都可以当成最终的结果来使用。但是为了提高鲁棒性，我们最终会选择两者之和 $w + \tilde{w}$ 作为最终的vector（两者的初始化不同相当于加了不同的随机噪声，所以能提高鲁棒性）。在训练了400亿个token组成的语料后，得到的实验结果如下图所示：



这个图一共采用了三个指标：语义准确度，语法准确度以及总体准确度。那么我们不难发现Vector Dimension在300时能达到最佳，而context Windows size大致在6到10之间。

3. GloVe与LSA、Word2Vec的比较

LSA (Latent Semantic Analysis) 是一种比较早的count-based的词向量表征工具，它也是基于co-occurrence matrix的，只不过采用了基于奇异值分解 (SVD) 的矩阵分解技术对大矩阵进行降维，而我们知道SVD的复杂度是很高的，所以它的计算代价比较大。还有一点是它对所有单词的统计权重都是一致的。而这些缺点在GloVe中被一一克服了。

而word2vec最大的缺点则是没有充分利用所有的语料，所以GloVe其实是把两者的优点结合了起来。从这篇论文给出的实验结果来看，GloVe的性能是远超LSA和word2vec的，但网上也有人说GloVe和word2vec实际表现其实差不多。

4. 代码实现

生成词向量

下载GitHub项目：<https://github.com/stanfordnlp/GloVe/archive/master.zip>

解压后，进入目录执行

make

进行编译操作。

然后执行 sh demo.sh 进行训练并生成词向量文件：vectors.txt和vectors.bin

[GloVe代码实现](#)

5. 参考文献

- [GloVe详解](#)
- [NLP从词袋到Word2Vec的文本表示](#)

作者：[@mantchs](#)

GitHub：<https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论！共同完善此项目！群号：【541954936】  加入QQ群

目录

- [1. 什么是textRNN](#)
 - [1.1 textRNN的原理](#)
- [2. textRNN网络结构](#)
 - [2.1 structure 1](#)
 - [2.2 structure 2](#)
 - [2.3 总结](#)
- [3. 什么是textCNN](#)
 - [3.1 一维卷积层](#)
 - [3.2 时序最大池化层](#)
 - [3.3 textCNN模型](#)
- [4. 代码实现](#)
- [5. 参考文献](#)

1. 什么是textRNN

textRNN指的是利用**RNN**循环神经网络解决文本分类问题，文本分类是自然语言处理的一个基本任务，试图推断出给定文本(句子、文档等)的标签或标签集合。

文本分类的应用非常广泛，如：

- 垃圾邮件分类：2分类问题，判断邮件是否为垃圾邮件
- 情感分析：2分类问题：判断文本情感是积极还是消极；多分类问题：判断文本情感属于{非常消极，消极，中立，积极，非常积极}中的哪一类。
- 新闻主题分类：判断一段新闻属于哪个类别，如财经、体育、娱乐等。根据类别标签的数量，可以是2分类也可以是多分类。
- 自动问答系统中的问句分类
- 社区问答系统中的问题分类：多标签多分类(对一段文本进行多分类，该文本可能有多个标签)，如 知乎看山杯
- 让AI做法官：基于案件事实描述文本的罚金等级分类(多分类)和法条分类(多标签多分类)
- 判断新闻是否为机器人所写：2分类

1.1 textRNN的原理

在一些自然语言处理任务中，当对序列进行处理时，我们一般会采用循环神经网络RNN，尤其是它的一些变种，如LSTM(更常用)，GRU。当然我们也可以把RNN运用到文本分类任务中。

这里的文本可以一个句子，文档(短文本，若干句子)或篇章(长文本)，因此每段文本的长度都不尽相同。在对文本进行分类时，我们一般会指定一个固定的输入序列/文本长度：该长度可以是最长文本/序列的长度，此时其他所有文本/序列都要进行填充以达到该长度；该长度也可以是训练集中所有文本/序列长度的均值，此时对于过长的文本/序列需要进行截断，过短的文本则进行填充。总之，要使得训练集中所有的文本/序列长度相同，该长度除之前提到的设置外，也可以是其他任意合理的数值。在测试时，也需要对

测试集中的文本/序列做同样的处理。

首先我们需要对文本进行分词，然后指定一个序列长度n（大于n的截断，小于n的填充），并使用词嵌入得到每个词固定维度的向量表示。对于每一个输入文本/序列，我们可以在RNN的每一个时间步长上输入文本中一个单词的向量表示，计算当前时间步长上的隐藏状态，然后用于当前时间步骤的输出以及传递给下一个时间步长并和下一个单词的词向量一起作为RNN单元输入，然后再计算下一个时间步长上RNN的隐藏状态，以此重复...直到处理完输入文本中的每一个单词，由于输入文本的长度为n，所以要经历n个时间步长。

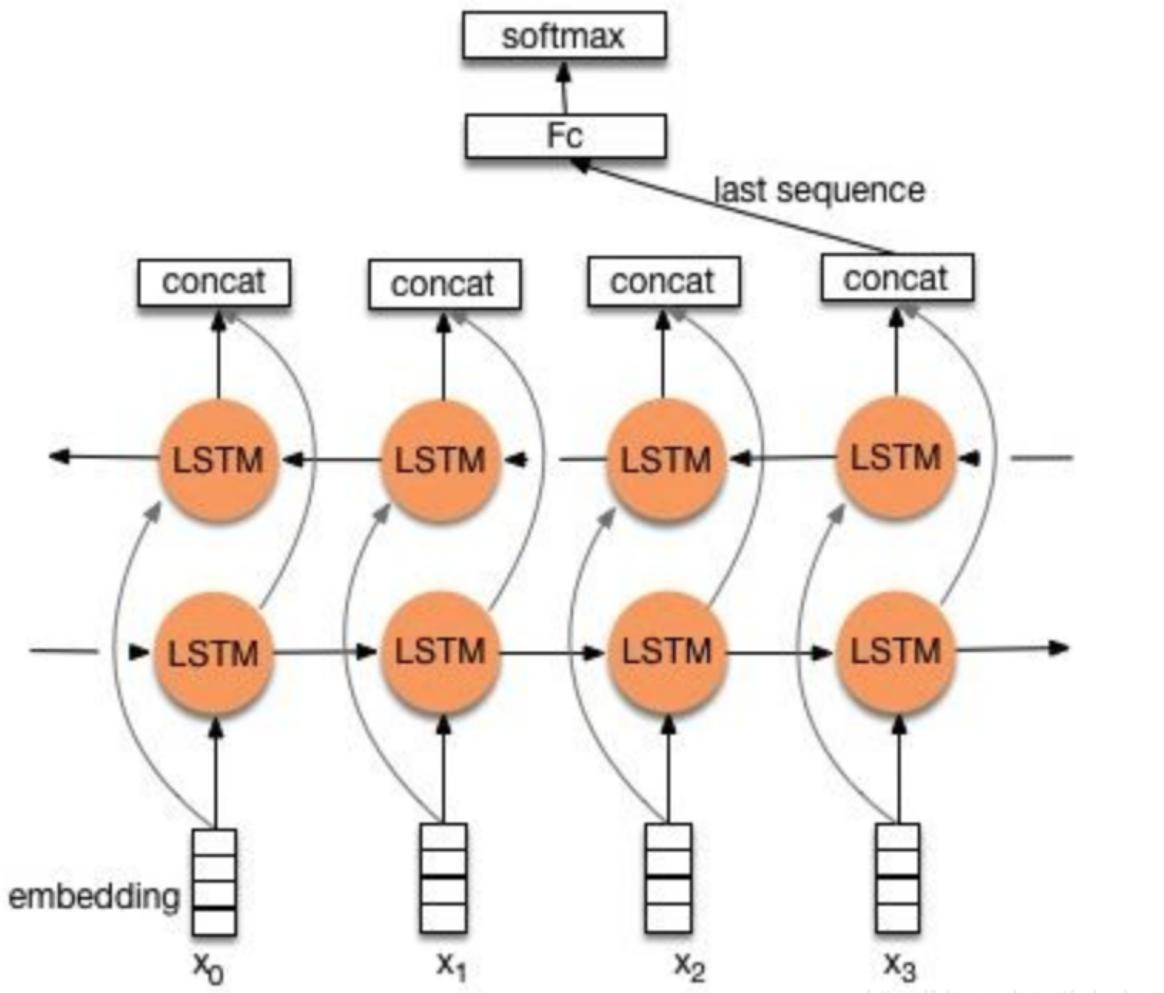
基于RNN的文本分类模型非常灵活，有多种多样的结构。接下来，我们主要介绍两种典型的结构。

2. textRNN网络结构

2.1 structure 1

流程：embedding--->BiLSTM--->concat final output/average all output----->softmax layer

结构图如下图所示：



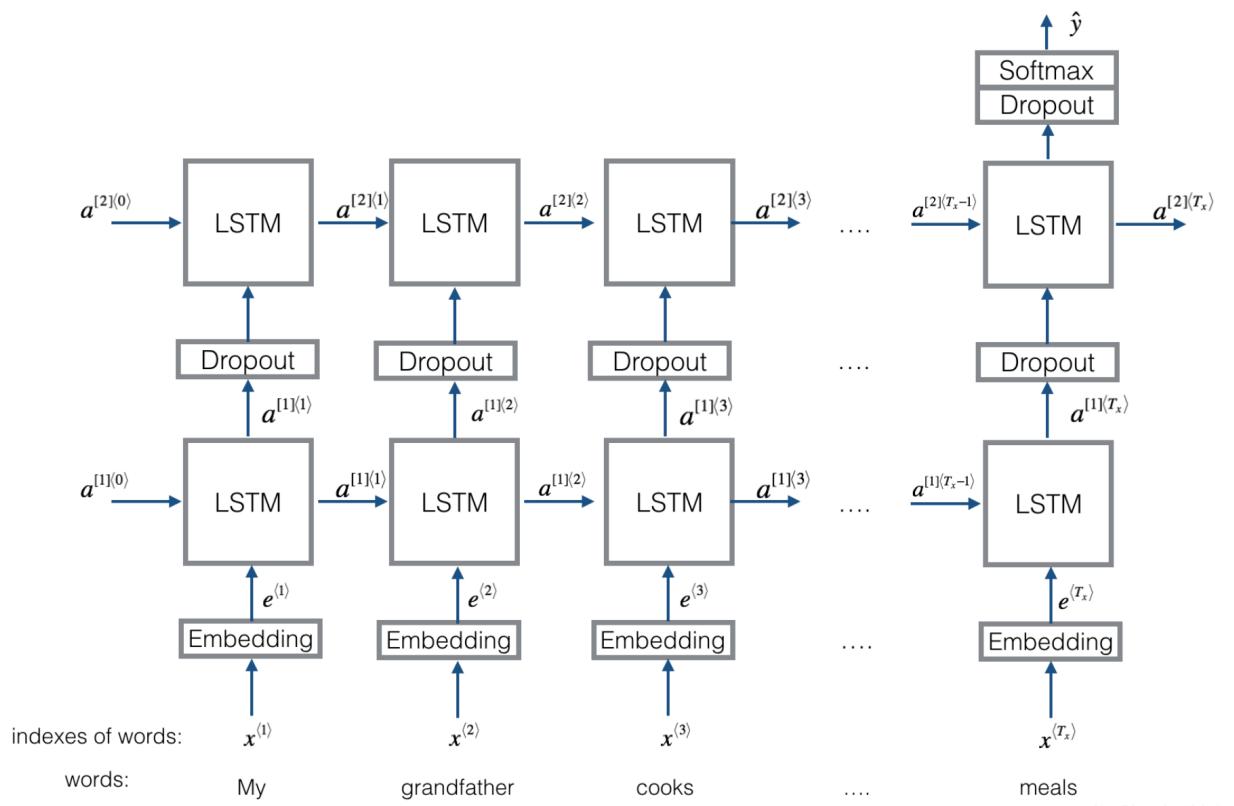
一般取前向/反向LSTM在最后一个时间步长上隐藏状态，然后进行拼接，在经过一个softmax层(输出层使用softmax激活函数)进行一个多分类；或者取前向/反向LSTM在每一个时间步长上的隐藏状态，对每一个时间步长上的两个隐藏状态进行拼接，然后对所有时间步长上拼接后的隐藏状态取均值，再经过一个softmax层(输出层使用softmax激活函数)进行一个多分类(2分类的话使用sigmoid激活函数)。

上述结构也可以添加**dropout/L2正则化或BatchNormalization** 来防止过拟合以及加速模型训练。

2.2 structure 2

流程：embedding-->BiLSTM---->(dropout)-->concat ouput--->UniLSTM--->(dropout)-->softmax layer

结构图如下图所示：



与之前结构不同的是，在双向LSTM(上图不太准确，底层应该是一个双向LSTM)的基础上又堆叠了一个单向的LSTM。把双向LSTM在每一个时间步长上的两个隐藏状态进行拼接，作为上层单向LSTM每一个时间步长上的一个输入，最后取上层单向LSTM最后一个时间步长上的隐藏状态，再经过一个softmax层(输出层使用softmax激活函数，2分类的话则使用sigmoid)进行一个多分类。

2.3 总结

TextRNN的结构非常灵活，可以任意改变。比如把LSTM单元替换为GRU单元，把双向改为单向，添加**dropout**或**BatchNormalization**以及再多堆叠一层等等。TextRNN在文本分类任务上的效果非常好，与TextCNN不相上下，但RNN的训练速度相对偏慢，一般2层就已经足够多了。

3. 什么是textCNN

在“卷积神经网络”中我们探究了如何使用二维卷积神经网络来处理二维图像数据。在之前的语言模型和文本分类任务中，我们将文本数据看作是只有一个维度的时间序列，并很自然地使用循环神经网络来表征这样的数据。其实，我们也可以将文本当作一维图像，从而可以用一维卷积神经网络来捕捉临近词之间的关联。本节将介绍将卷积神经网络应用到文本分析的开创性工作之一：**textCNN**。

3.1 一维卷积层

在介绍模型前我们先来解释一维卷积层的工作原理。与二维卷积层一样，一维卷积层使用一维的互相关运算。在一维互相关运算中，卷积窗口从输入数组的最左方开始，按从左往右的顺序，依次在输入数组上滑动。当卷积窗口滑动到某一位置时，窗口中的输入子数组与核数组按元素相乘并求和，得到输出数组中相应位置的元素。如下图所示，输入是一个宽为7的一维数组，核数组的宽为2。可以看到输出的宽度为 $7 - 2 + 1 = 6$ ，且第一个元素是由输入的最左边的宽为2的子数组与核数组按元素相乘后再相加得到的： $0 \times 1 + 1 \times 2 = 2$ 。

输入							*	核		=	输出					
0	1	2	3	4	5	6		1	2		2	5	8	11	14	17

多输入通道的一维互相关运算也与多输入通道的二维互相关运算类似：在每个通道上，将核与相应的输入做一维互相关运算，并将通道之间的结果相加得到输出结果。下图展示了含3个输入通道的一维互相关运算，其中阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$ 。

输入								*	核			=	输出					
2	3	4	5	6	7	8		-1	-3		2	8	14	20	26	32		
1	2	3	4	5	6	7		3	4									
0	1	2	3	4	5	6		1	2									

由二维互相关运算的定义可知，多输入通道的一维互相关运算可以看作单输入通道的二维互相关运算。如下图所示，我们也可以将上图中多输入通道的一维互相关运算以等价的单输入通道的二维互相关运算呈现。这里核的高等于输入的高。下图的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$ 。

输入							*	核		=	输出					
2	3	4	5	6	7	8		-1	-3		2	8	14	20	26	32
1	2	3	4	5	6	7		3	4							
0	1	2	3	4	5	6		1	2							

以上都是输出都只有一个通道。我们在“多输入通道和多输出通道”一节中介绍了如何在二维卷积层中指定多个输出通道。类似地，我们也可以在一维卷积层指定多个输出通道，从而拓展卷积层中的模型参数。

3.2 时序最大池化层

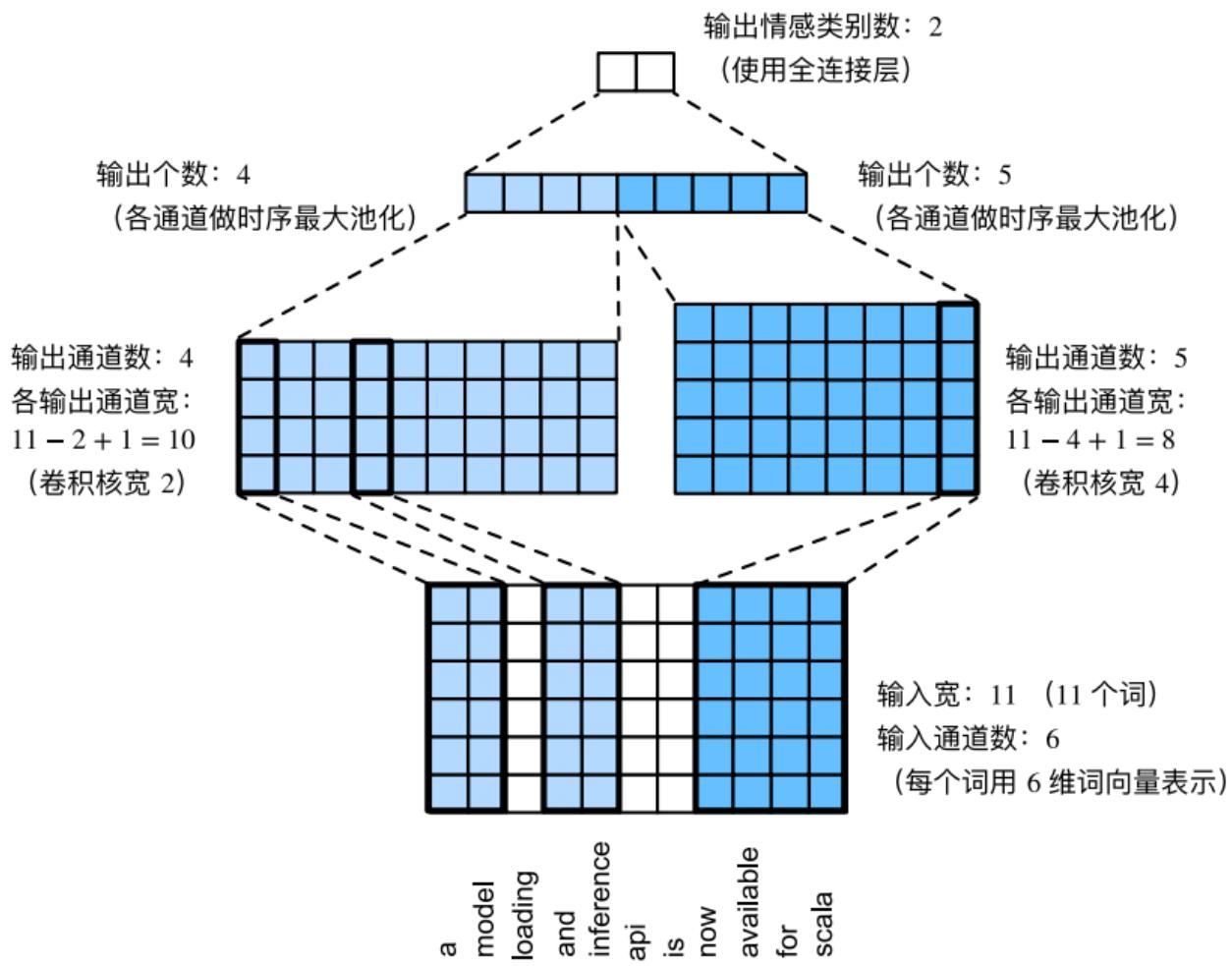
类似地，我们有一维池化层。textCNN中使用的时序最大池化（max-over-time pooling）层实际上对应一维全局最大池化层：假设输入包含多个通道，各通道由不同时间步上的数值组成，各通道的输出即该通道所有时间步中最大的数值。因此，时序最大池化层的输入在各个通道上的时间步数可以不同。为提升计算性能，我们常常将不同长度的时序样本组成一个小批量，并通过在较短序列后附加特殊字符（如0）令批量中各时序样本长度相同。这些人为添加的特殊字符当然是无意义的。由于时序最大池化的主要目的是抓取时序中最重要的特征，它通常能使模型不受人为添加字符的影响。

3.3 textCNN模型

textCNN模型主要使用了一维卷积层和时序最大池化层。假设输入的文本序列由 n 个词组成，每个词用 d 维的词向量表示。那么输入样本的宽为 n ，高为1，输入通道数为 d 。textCNN的计算主要分为以下几步：

1. 定义多个一维卷积核，并使用这些卷积核对输入分别做卷积计算。宽度不同的卷积核可能会捕捉到不同个数的相邻词的相关性。
2. 对输出的所有通道分别做时序最大池化，再将这些通道的池化输出值连结为向量。
3. 通过全连接层将连结后的向量变换为有关各类别的输出。这一步可以使用丢弃层应对过拟合。

下图用一个例子解释了textCNN的设计。这里的输入是一个有11个词的句子，每个词用6维词向量表示。因此输入序列的宽为11，输入通道数为6。给定2个一维卷积核，核宽分别为2和4，输出通道数分别设为4和5。因此，一维卷积计算后，4个输出通道的宽为 $11 - 2 + 1 = 10$ ，而其他5个通道的宽为 $11 - 4 + 1 = 8$ 。尽管每个通道的宽不同，我们依然可以对各个通道做时序最大池化，并将9个通道的池化输出连结成一个9维向量。最终，使用全连接将9维向量变换为2维输出，即正面情感和负面情感的预测。



4. 代码实现

清华新闻分类数据集下载: <https://www.lanzous.com/i5t0lsd>

- [textRNN实现新闻分类](#)
- [textCNN实现新闻分类](#)

5. 参考文献

- [基于TextRNN的文本分类原理](#)
- [动手学深度学习](#)

作者: [@mantchs](#)

GitHub: <https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论! 共同完善此项目! 群号: 【541954936】 [加入QQ群](#)

目录

- [1. 什么是seq2seq](#)
- [2. 编码器](#)
- [3. 解码器](#)
- [4. 训练模型](#)
- [5. seq2seq模型预测](#)
 - [5.1 贪婪搜索](#)
 - [5.2 穷举搜索](#)
 - [5.3 束搜索](#)
- [6. Bleu得分](#)
- [7. 代码实现](#)
- [8. 参考文献](#)

1. 什么是seq2seq

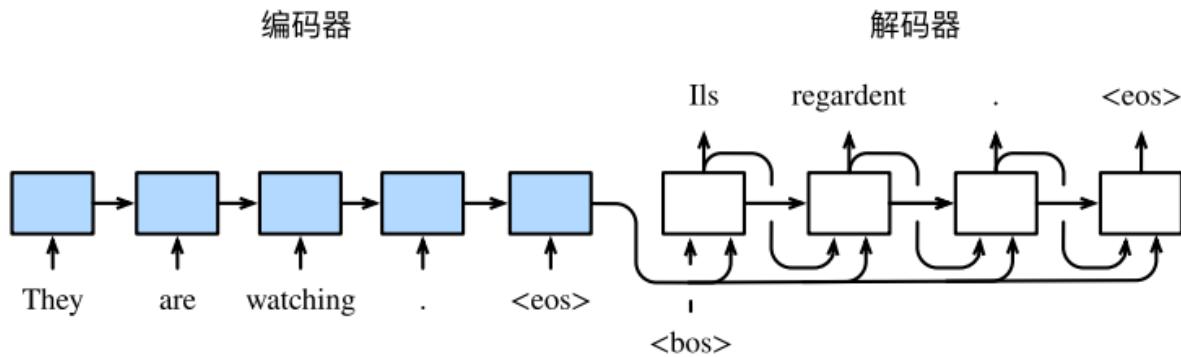
在自然语言处理的很多应用中，输入和输出都可以是不定长序列。以机器翻译为例，输入可以是一段不定长的英语文本序列，输出可以是一段不定长的法语文本序列，例如：

英语输入：“They”、“are”、“watching”、“”

法语输出：“Ils”、“regardent”、“”

当输入和输出都是不定长序列时，我们可以使用编码器—解码器（encoder-decoder）或者seq2seq模型。序列到序列模型，简称**seq2seq**模型。这两个模型本质上都用到了两个循环神经网络，分别叫做编码器和解码器。编码器用来分析输入序列，解码器用来生成输出序列。两个循环神经网络是共同训练的。

下图描述了使用编码器—解码器将上述英语句子翻译成法语句子的一种方法。在训练数据集中，我们可以在每个句子后附上特殊符号“<eos>”（end of sequence）以表示序列的终止。编码器每个时间步的输入依次为英语句子中的单词、标点和特殊符号“<eos>”。下图中使用了编码器在最终时间步的隐藏状态作为输入句子的表征或编码信息。解码器在各个时间步中使用输入句子的编码信息和上个时间步的输出以及隐藏状态作为输入。我们希望解码器在各个时间步能正确依次输出翻译后的法语单词、标点和特殊符号“<eos>”。需要注意的是，解码器在最初时间步的输入用到了一个表示序列开始的特殊符号“”（beginning of sequence）。



2. 编码器

编码器的作用是把一个不定长的输入序列转换成一个定长的背景变量 c ，并在该背景变量中编码输入序列信息。常用的编码器是循环神经网络。

让我们考虑批量大小为1的时序数据样本。假设输入序列是 x_1, \dots, x_T ，例如 x_i 是输入句子中的第 i 个词。在时间步 t ，循环神经网络将输入 x_t 的特征向量 x_t 和上个时间步的隐藏状态 h_{t-1} 变换为当前时间步的隐藏状态 h_t 。我们可以用函数 f 表达循环神经网络隐藏层的变换：

$$h_t = f(x_t, h_{t-1})$$

接下来，编码器通过自定义函数 q 将各个时间步的隐藏状态变换为背景变量：

$$c = q(h_1, \dots, h_T)$$

例如，当选择 $q(h_1, \dots, h_T) = h_T$ 时，背景变量是输入序列最终时间步的隐藏状态 h_T 。

以上描述的编码器是一个单向的循环神经网络，每个时间步的隐藏状态只取决于该时间步及之前的输入子序列。我们也可以使用双向循环神经网络构造编码器。在这种情况下，编码器每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入），并编码了整个序列的信息。

3. 解码器

刚刚已经介绍，编码器输出的背景变量 c 编码了整个输入序列 x_1, \dots, x_T 的信息。给定训练样本中的输出序列 $y_1, y_2, \dots, y_{T'}$ ，对每个时间步 t' （符号与输入序列或编码器的时间步 t 有区别），解码器输出 $y_{t'}$ 的条件概率将基于之前的输出序列 $y_1, \dots, y_{t'-1}$ 和背景变量 c ，即：

$$P(y_{t'} | y_1, \dots, y_{t'-1}, c)$$

为此，我们可以使用另一个循环神经网络作为解码器。在输出序列的时间步 t' ，解码器将上一时间步的输出 $y_{t'-1}$ 以及背景变量 c 作为输入，并将它们与上一时间步的隐藏状态 $s_{t'-1}$ 变换为当前时间步的隐藏状态 $s_{t'}$ 。因此，我们可以用函数 g 表达解码器隐藏层的变换：

$$s_{t'} = g(y_{t'-1}, c, s_{t'-1})$$

有了解码器的隐藏状态后，我们可以使用自定义的输出层和softmax运算来计算

$P(y_{t'} | y_1, \dots, y_{t'-1}, c)$ ，例如，基于当XQ前时间步的解码器隐藏状态 $s_{t'}$ 、上一时间步的输出 $s_{t'-1}$ 以及背景变量 c 来计算当前时间步输出 $y_{t'}$ 的概率分布。

4. 训练模型

根据最大似然估计，我们可以最大化输出序列基于输入序列的条件概率：

$$\begin{aligned} P(y_1, \dots, y_{t'-1} | x_1, \dots, x_T) &= \prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, x_1, \dots, x_T) \\ &= \prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, c) \end{aligned}$$

并得到该输出序列的损失：

$$-\log P(y_1, \dots, y_{t'-1} | x_1, \dots, x_T) = -\sum_{t'=1}^{T'} \log P(y_{t'} | y_1, \dots, y_{t'-1}, c)$$

在模型训练中，所有输出序列损失的均值通常作为需要最小化的损失函数。在上图所描述的模型预测中，我们需要将解码器在上一个时间步的输出作为当前时间步的输入。与此不同，在训练中我们也可以将标签序列（训练集的真实输出序列）在上一个时间步的标签作为解码器在当前时间步的输入。这叫作强制教学（teacher forcing）。

5. seq2seq模型预测

以上介绍了如何训练输入和输出均为不定长序列的编码器—解码器。本节我们介绍如何使用编码器—解码器来预测不定长的序列。

在准备训练数据集时，我们通常会在样本的输入序列和输出序列后面分别附上一个特殊符号“<eos>”表示序列的终止。我们在接下来的讨论中也将沿用上一节的全部数学符号。为了便于讨论，假设解码器的输出是一段文本序列。设输出文本词典Y（包含特殊符号“<eos>”）的大小为 $|Y|$ ，输出序列的最大长度为 T' 。所有可能的输出序列一共有 $O(|y|^{T'})$ 种。这些输出序列中所有特殊符号“<eos>”后面的子序列将被舍弃。

5.1 贪婪搜索

贪婪搜索 (greedy search)。对于输出序列任一时间步 t' , 我们从 $|Y|$ 个词中搜索出条件概率最大的词:

$$y_{t'} = \operatorname{argmax}_{y \in Y} P(y | y_1, \dots, y_{t'-1}, c)$$

作为输出。一旦搜索出“`<eos>`”符号, 或者输出序列长度已经达到了最大长度 T' , 便完成输出。我们在描述解码器时提到, 基于输入序列生成输出序列的条件概率是

$$\prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, c)。我们将该条件概率最大的输出序列称为最优$$

输出序列。而贪婪搜索的主要问题是不能保证得到最优输出序列。

下面来看一个例子。假设输出词典里面有“`A`”“`B`”“`C`”和“`<eos>`”这4个词。下图中每个时间步下的4个数字分别代表了该时间步生成“`A`”“`B`”“`C`”和“`<eos>`”这4个词的条件概率。在每个时间步, 贪婪搜索选取条件概率最大的词。因此, 图10.9中将生成输出序列“`A`”“`B`”“`C`”“`<eos>`”。该输出序列的条件概率是 $0.5 \times 0.4 \times 0.6 = 0.048$ 。

时间步	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<code><eos></code>	0.1	0.2	0.2	0.6

接下来, 观察下面演示的例子。与上图中不同, 在时间步2中选取了条件概率第二大的词“`C`”。由于时间步3所基于的时间步1和2的输出子序列由上图中的“`A`”“`B`”变为了下图中的“`A`”“`C`”, 下图中时间步3生成各个词的条件概率发生了变化。我们选取条件概率最大的词“`B`”。此时时间步4所基于的前3个时间步的输出子序列为“`A`”“`C`”“`B`”, 与上图中的“`A`”“`B`”“`C`”不同。因此, 下图中时间步4生成各个词的条件概率也与上图中的不同。我们发现, 此时的输出序列“`A`”“`C`”“`B`”“`<eos>`”的条件概率是 $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$, 大于贪婪搜索得到的输出序列的条件概率。因此, 贪婪搜索得到的输出序列“`A`”“`B`”“`C`”“`<eos>`”并非最优输出序列。

时间步	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<code><eos></code>	0.1	0.2	0.1	0.6

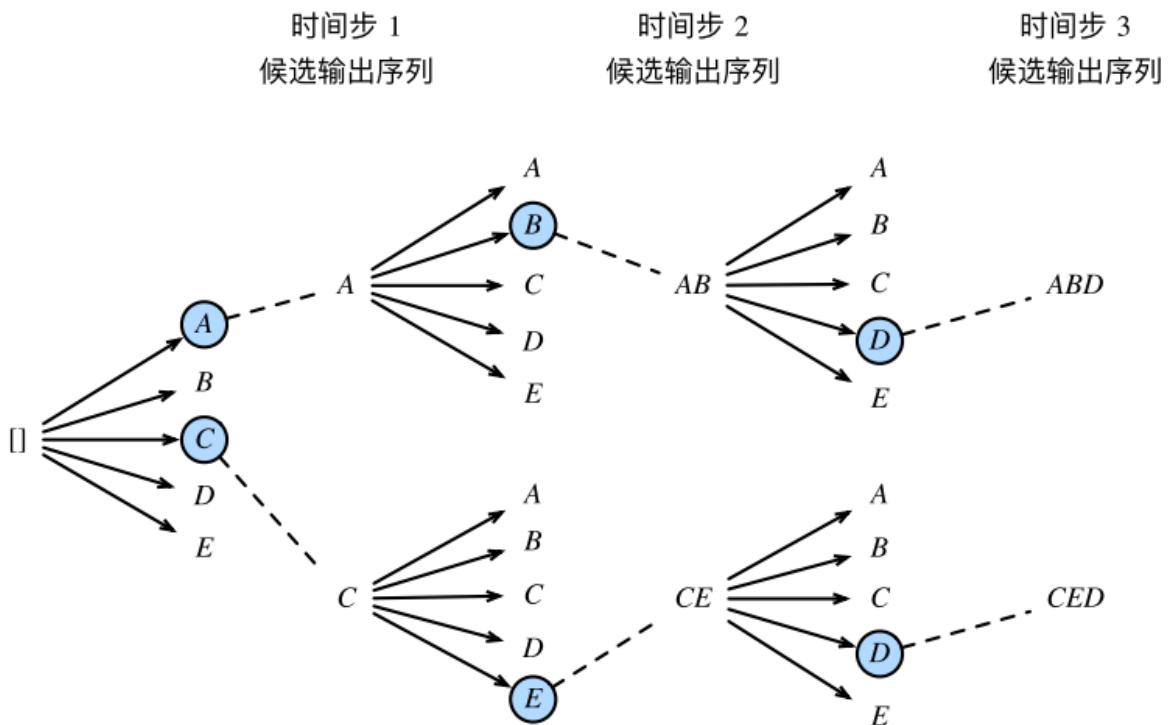
5.2 穷举搜索

如果目标是得到最优输出序列，我们可以考虑穷举搜索（exhaustive search）：穷举所有可能的输出序列，输出条件概率最大的序列。

虽然穷举搜索可以得到最优输出序列，但它的计算开销 $O(|y|^{T'})$ 很容易过大。例如，当 $|Y| = 10000$ 且 $T' = 10$ 时，我们将评估 $10000^{10} = 10^{40}$ 个序列：这几乎不可能完成。而贪婪搜索的计算开销是 $O(|y|^{T'})$ ，通常显著小于穷举搜索的计算开销。例如，当 $|Y| = 10000$ 且 $T' = 10$ 时，我们只需评估 $10000 * 10 = 10^5$ 个序列。

5.3 束搜索

束搜索（beam search）是对贪婪搜索的一个改进算法。它有一个束宽（beam size）超参数。我们将它设为 k 。在时间步 1 时，选取当前时间步条件概率最大的 k 个词，分别组成 k 个候选输出序列的首词。在之后的每个时间步，基于上个时间步的 k 个候选输出序列，从 $k |Y|$ 个可能的输出序列中选取条件概率最大的 k 个，作为该时间步的候选输出序列。最终，我们从各个时间步的候选输出序列中筛选出包含特殊符号“<eos>”的序列，并将它们中所有特殊符号“<eos>”后面的子序列舍弃，得到最终候选输出序列的集合。



束宽为2，输出序列最大长度为3。候选输出序列有A、C、AB、CE、ABD和CED。我们将根据这6个序列得出最终候选输出序列的集合。在最终候选输出序列的集合中，我们取以下分数最高的序列作为输出序列：

$$\frac{1}{L^\alpha} \log P(y_1, \dots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^{T'} \log P(y_{t'} | y_1, \dots, y_{t'-1}, c)$$

其中 L 为最终候选序列长度， α 一般可选为0.75。分母上的 L^α 是为了惩罚较长序列在以上分数中较多的对数相加项。分析可知，束搜索的计算开销为 $O(k|y|^{T'})$ 。这介于贪婪搜索和穷举搜索的计算开销之间。此外，贪婪搜索可看作是束宽为1的束搜索。束搜索通过灵活的束宽 k 来权衡计算开销和搜索质量。

6. Bleu得分

评价机器翻译结果通常使用BLEU (Bilingual Evaluation Understudy) (双语评估替补)。对于模型预测序列中任意的子序列，BLEU考察这个子序列是否出现在标签序列中。

具体来说，设词数为 n 的子序列的精度为 p_n 。它是预测序列与标签序列匹配词数为 n 的子序列的数量与预测序列中词数为 n 的子序列的数量之比。举个例子，假设标签序列为A、B、C、D、E、F，预测序列为A、B、B、C、D，那么：

$$P1 = \frac{\text{预测序列中的1元词组在标签序列是否存在个数}}{\text{预测序列1元词组的个数之和}}$$

预测序列一元词组：A/B/C/D，都在标签序列里存在，所以 $P1=4/5$ ，以此类推， $p2 = 3/4$, $p3 = 1/3$, $p4 = 0$ 。设 len_{label} , len_{pred} 分别为标签序列和预测序列的词数，那么，BLEU的定义为：

$$\exp(\min(0, 1 - \frac{len_{label}}{len_{pred}})) \prod_{n=1}^k p_n^{\frac{1}{2^n}}$$

其中 k 是我们希望匹配的子序列的最大词数。可以看到当预测序列和标签序列完全一致时，BLEU为1。

因为匹配较长子序列比匹配较短子序列更难，BLEU对匹配较长子序列的精度赋予了更大权重。例如，当 p_n 固定在0.5时，随着 n 的增大，

$$0.5^{\frac{1}{2}} \approx 0.7, 0.5^{\frac{1}{4}} \approx 0.84, 0.5^{\frac{1}{8}} \approx 0.92, 0.5^{\frac{1}{16}} \approx 0.96$$

另外，模型预测较短序列往往得到较高 p_n 值。因此，上式中连乘项前面的系数是为了惩罚较短的输出而设的。举个例子，当 $k = 2$ 时，假设标签序列为A、B、C、D、E、F，而预测序列为A、B。虽然 $p1 = p2 = 1$ ，但惩罚系数 $\exp(1-6/2) \approx 0.14$ ，因此BLEU也接近0.14。

7. 代码实现

[TensorFlow seq2seq的基本实现](#)

8. 参考文献

[动手学深度学习](#)

作者: [@mantchs](#)

GitHub: <https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论! 共同完善此项目! 群号: 【541954936】  加入QQ群

目录

- [1. 什么是Attention机制](#)
- [2. 编解码器中的Attention](#)
 - [2.1 计算背景变量](#)
 - [2.2 更新隐藏状态](#)
- [3. Attention本质](#)
 - [3.1 机器翻译说明Attention](#)
 - [3.2 注意力分配概率计算](#)
 - [3.3 Attention的物理含义](#)
- [4. Self-Attention模型](#)
- [5. 发展](#)
- [6. 代码实现](#)
- [7. 参考文献](#)

1. 什么是Attention机制

在“编码器—解码器（seq2seq）”一节里，解码器在各个时间步依赖相同的背景变量来获取输入序列信息。当编码器为循环神经网络时，背景变量来自它最终时间步的隐藏状态。

现在，让我们再次思考那一节提到的翻译例子：输入为英语序列“They”“are”“watching”“.”，输出为法语序列“Its”“regardent”“.”。不难想到，解码器在生成输出序列中的每一个词时可能只需利用输入序列某一部分的信息。例如，在输出序列的时间步1，解码器可以主要依赖“They”“are”的信息来生成“Its”，在时间步2则主要使用来自“watching”的编码信息生成“regardent”，最后在时间步3则直接映射句号“.”。这看上去就像是在解码器的每一时间步对输入序列中不同时间步的表征或编码信息分配不同的注意力一样。这也是注意力机制的由来。

仍然以循环神经网络为例，注意力机制通过对编码器所有时间步的隐藏状态做加权平均来得到背景变量。解码器在每一时间步调整这些权重，即注意力权重，从而能够在不同时间步分别关注输入序列中的不同部分并编码进相应时间步的背景变量。

在注意力机制中，解码器的每一时间步将使用可变的背景变量。记 $c_{t'}$ 是解码器在时间步 t' 的背景变量，那么解码器在该时间步的隐藏状态可以改写为：

$$s_{t'} = g(y_{t'-1}, c_{t'}, s_{t'-1})$$

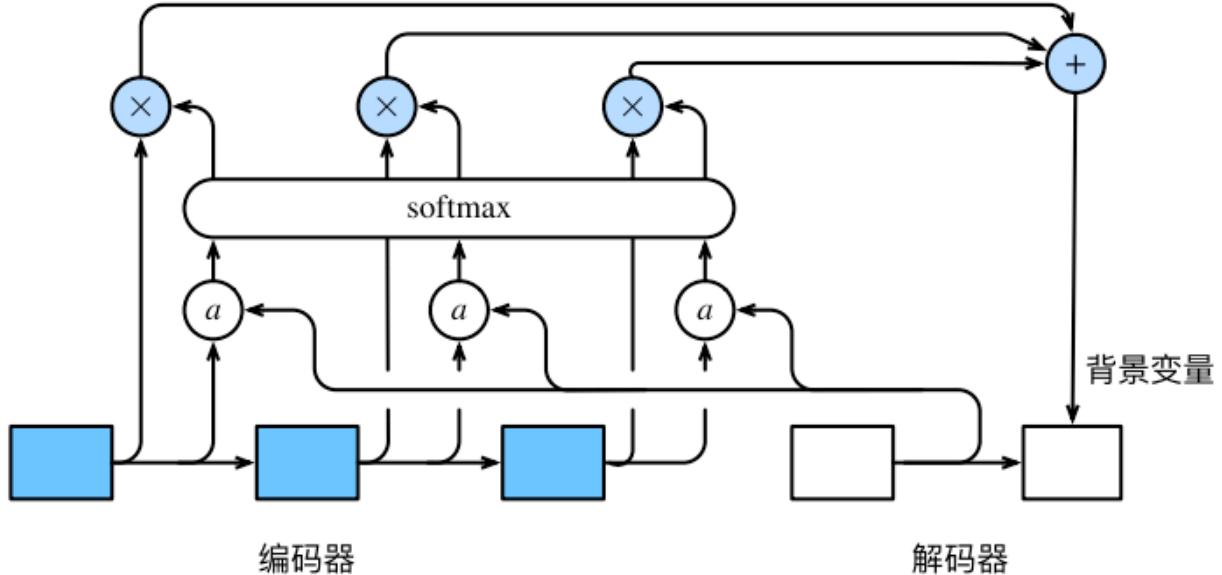
这里的关键是如何计算背景变量 $c_{t'}$ 和如何利用它来更新隐藏状态 $s_{t'}$ 。下面将分别描述这两个关键点。

2. 编解码器中的Attention

2.1 计算背景变量

我们先描述第一个关键点，即计算背景变量。下图描绘了注意力机制如何为解码器在时间步 2 计算背景变量。

1. 函数 a 根据解码器在时间步 1 的隐藏状态和编码器在各个时间步的隐藏状态计算softmax运算的输入。
2. softmax运算输出概率分布并对编码器各个时间步的隐藏状态做加权平均，从而得到背景变量。



令编码器在时间步 t 的隐藏状态为 h_t ，且总时间步数为 T 。那么解码器在时间步 t' 的背景变量为所有编码器隐藏状态的加权平均：

$$c_{t'} = \sum_{t=1}^T \alpha_{t' t} h_t$$

矢量化计算背景变量

我们还可以对注意力机制采用更高效的矢量化计算。我们先定义，在上面的例子中，查询项为解码器的隐藏状态，键项和值项均为编码器的隐藏状态。

广义上，注意力机制的输入包括查询项以及一一对应的键项和值项，其中值项是需要加权平均的一组项。在加权平均中，值项的权重来自查询项以及与该值项对应的键项的计算。

让我们考虑一个常见的简单情形，即编码器和解码器的隐藏单元个数均为 h ，且函数 $a(s, h) = s^T h$ 。假设我们希望根据解码器单个隐藏状态 $s_{t'-1}$ 和编码器所有隐藏状态 $h_t, t = 1, \dots, T$ 来计算背景向量 $c_{t'}$ 。我们可以将查询项矩阵 Q 设为 $s_{t'-1}^T$ ，并令键项矩阵 K 和值项矩阵 V 相同且第 t 行均为 h_t^T 。此时，我们只需要通过矢量化计算：

$$\text{softmax}(QK^T)V$$

即可算出转置后的背景向量 $c_{t'}^T$ 。当查询项矩阵 Q 的行数为 n 时，上式将得到 n 行的输出矩阵。输出矩阵与查询项矩阵在相同行上一一对应。

2.2 更新隐藏状态

现在我们描述第二个关键点，即更新隐藏状态。以门控循环单元为例，在解码器中我们可以对门控循环单元（GRU）中门控循环单元的设计稍作修改，从而变换上一时间步 $t'-1$ 的输出 $y_{t'-1}$ 、隐藏状态 $s_{t'-1}$ 和当前时间步 t' 的含注意力机制的背景变量 $c_{t'}$ 。解码器在时间步: t' 的隐藏状态为：

$$s_{t'} = z_{t'} \odot s_{t'-1} + (1 - z_{t'}) \odot \tilde{s}_{t'}$$

其中的重置门、更新门和候选隐藏状态分别为：

$$r_{t'} = \sigma(W_{yr}y_{t'-1} + W_{sr}s_{t'-1} + W_{cr}c_{t'} + b_r)$$

$$z_{t'} = \sigma(W_{yz}y_{t'-1} + W_{sz}s_{t'-1} + W_{cz}c_{t'} + b_z)$$

$$\tilde{s}_{t'} = \tanh(W_{ys}y_{t'-1} + W_{ss}(s_{t'-1} \odot r_{t'}) + W_{cs}c_{t'} + b_s)$$

其中含下标的 W 和 b 分别为门控循环单元的权重参数和偏差参数。

3. Attention本质

3.1 机器翻译说明Attention

本节先以机器翻译作为例子讲解最常见的Soft Attention模型的基本原理，之后抛离Encoder-Decoder框架抽象出了注意力机制的本质思想。

如果拿机器翻译来解释这个Encoder-Decoder框架更好理解，比如输入的是英文句子：Tom chase Jerry，Encoder-Decoder框架逐步生成中文单词：“汤姆”，“追逐”，“杰瑞”。

在翻译“杰瑞”这个中文单词的时候，模型里面的每个英文单词对于翻译目标单词“杰瑞”贡献是相同的，很明显这里不太合理，显然“Jerry”对于翻译成“杰瑞”更重要，但是模型是无法体现这一点的，这就是为何说它没有引入注意力的原因。

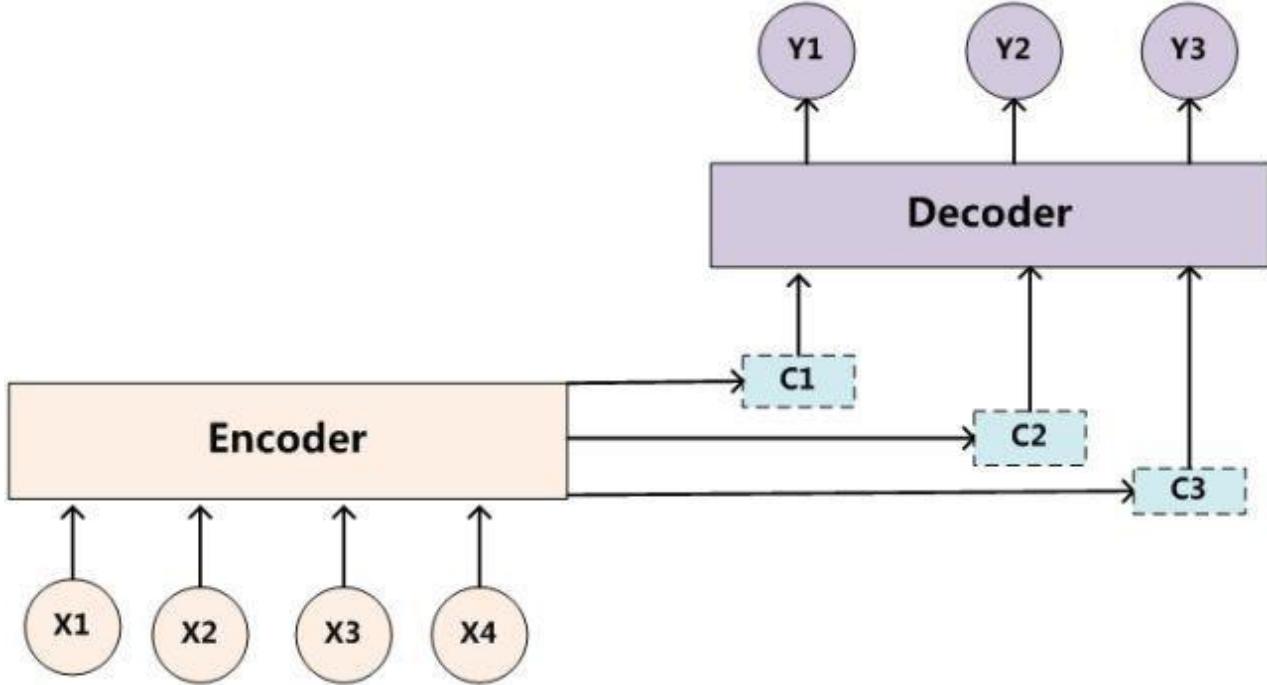
没有引入注意力的模型在输入句子比较短的时候问题不大，但是如果输入句子比较长，此时所有语义完全通过一个中间语义向量来表示，单词自身的信息已经消失，可想而知会丢失很多细节信息，这也是为何要引入注意力模型的重要原因。

上面的例子中，如果引入Attention模型的话，应该在翻译“杰瑞”的时候，体现出英文单词对于翻译当前中文单词不同的影响程度，比如给出类似下面一个概率分布值：

(Tom,0.3) (Chase,0.2) (Jerry,0.5)

每个英文单词的概率代表了翻译当前单词“杰瑞”时，注意力分配模型分配给不同英文单词的注意力大小。这对于正确翻译目标语单词肯定是有帮助的，因为引入了新的信息。

同理，目标句子中的每个单词都应该学会其对应的源语句子中单词的注意力分配概率信息。这意味着在生成每个单词 y_i 的时候，原先都是相同的中间语义表示 C 会被替换成根据当前生成单词而不断变化的 C_i 。理解Attention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 C_i 。增加了注意力模型的Encoder-Decoder框架理解起来如下图所示。



每个 C_i 可能对应着不同的源语句子单词的注意力分配概率分布，比如对于上面的英汉翻译来说，其对应的信息可能如下：

$$C_{\text{汤姆}} = g(0.6 * f2(\text{tom}), 0.2 * f2(\text{chase}), 0.2 * f2(\text{jerry}))$$

$$C(\text{追逐}) = g(0.2 * f2(\text{tom}), 0.7 * f2(\text{chase}), 0.1 * f2(\text{jerry}))$$

$$C(\text{杰瑞}) = g(0.3 * f2(\text{tom}), 0.2 * f2(\text{chase}), 0.5 * f2(\text{jerry}))$$

其中， $f2$ 函数代表Encoder对输入英文单词的某种变换函数，比如如果Encoder是用的RNN模型的话，这个 $f2$ 函数的结果往往是某个时刻输入 x_i 后隐层节点的状态值； g 代表Encoder根据单词的中间表示合成整个句子中间语义表示的变换函数，一般的做法中， g 函数就是对构成元素加权求和，即下列公式：

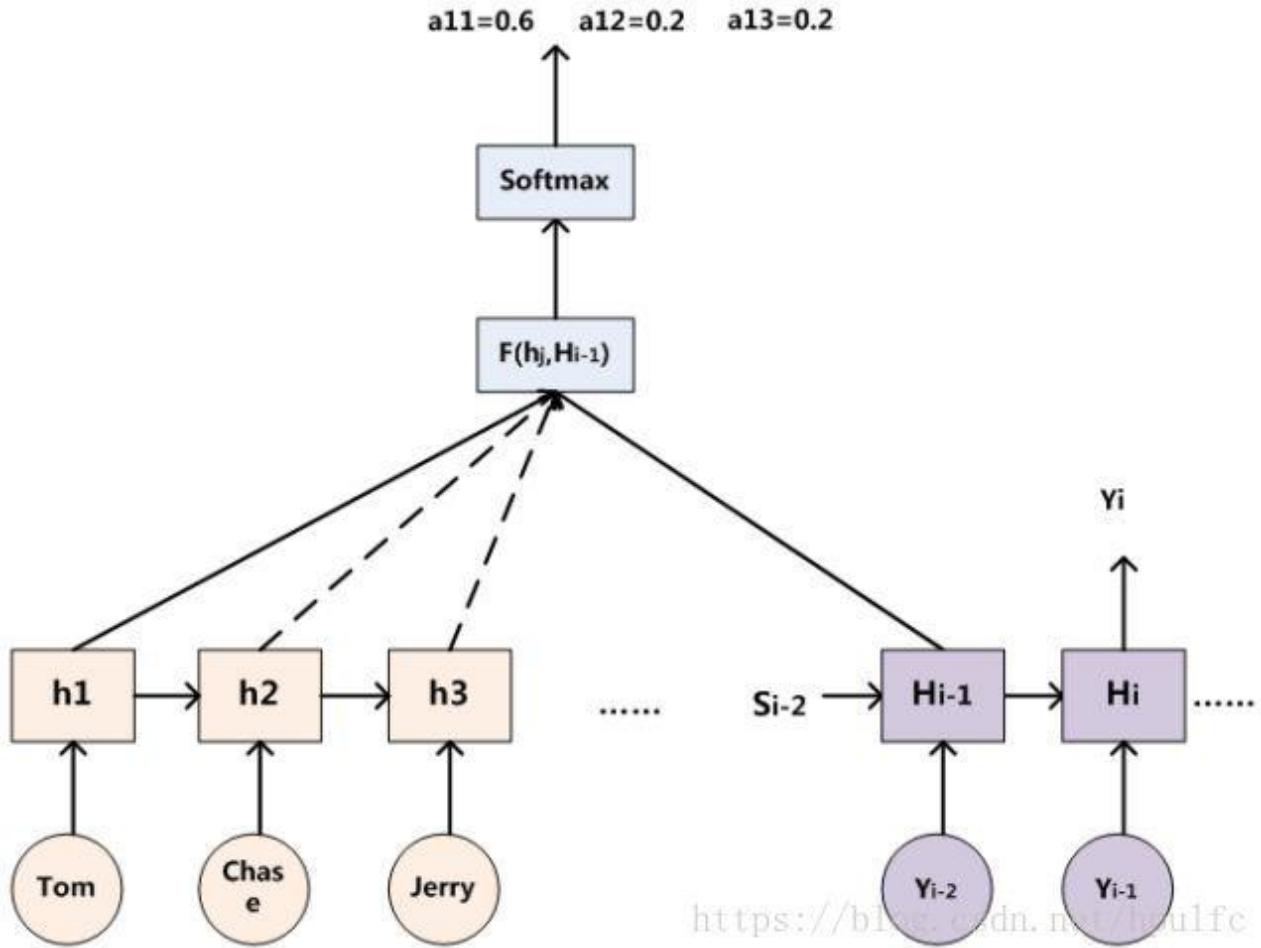
$$C_i = \sum_{j=1}^{L_x} a_{ij} h_j$$

其中， L_x 代表输入句子Source的长度， a_{ij} 代表在Target输出第*i*个单词时Source输入句子中第*j*个单词的注意力分配系数，而 h_j 则是Source输入句子中第*j*个单词的语义编码。假设下标*i*就是上面例子所说的“汤姆”，那么 L_x 就是3， $h_1=f(\text{Tom})$ ， $h_2=f(\text{Chase})$ ， $h_3=f(\text{Jerry})$ 分别是输入句子每个单词的语义编码，对应的注意力模型权值则分别是0.6,0.2,0.2，所以 g 函数本质上就是个加权求和函数。

3.2 注意力分配概率计算

这里还有一个问题：生成目标句子某个单词，比如“汤姆”的时候，如何知道Attention模型所需要的输入句子单词注意力分配概率分布值呢？就是说“汤姆”对应的输入句子Source中各个单词的概率分布： $(\text{Tom}, 0.6), (\text{Chase}, 0.2), (\text{Jerry}, 0.2)$ 是如何得到的呢？

对于采用RNN的Decoder来说，在时刻*i*，如果要生成 y_i 单词，我们是可以知道Target在生成 y_i 之前的时刻*i-1*时，隐层节点*i-1*时刻的输出值 H_{i-1} 的，而我们的目的是要计算生成 y_i 时输入句子中的单词“Tom”、“Chase”、“Jerry”对 y_i 来说的注意力分配概率分布，那么可以用Target输出句子*i-1*时刻的隐层节点状态 H_{i-1} 去一一和输入句子Source中每个单词对应的RNN隐层节点状态 h_j 进行对比，即通过函数 $F(h_j, H_{i-1})$ 来获得目标单词 y_i 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。



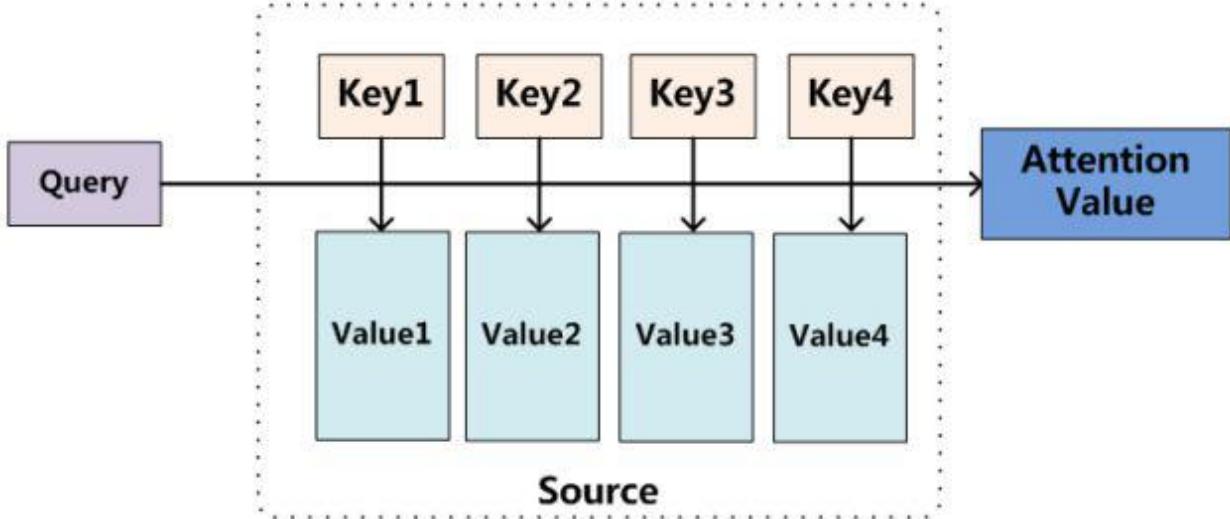
<https://bluelink.bj.bjtu.edu.cn/~hulfc/>

3.3 Attention的物理含义

一般在自然语言处理应用里会把Attention模型看作是输出Target句子中某个单词和输入Source句子每个单词的对齐模型，这是非常有道理的。

目标句子生成的每个单词对应输入句子单词的概率分布可以理解为输入句子单词和这个目标生成单词的对齐概率，这在机器翻译语境下是非常直观的：传统的统计机器翻译一般在做的过程中会专门有一个短语对齐的步骤，而注意力模型其实起的是相同的作用。

如果把Attention机制从上文讲述例子中的Encoder-Decoder框架中剥离，并进一步做抽象，可以更容易看懂Attention机制的本质思想。



我们可以这样来看待Attention机制（参考图9）：将Source中的构成元素想象成是由一系列的 $\langle \text{Key}, \text{Value} \rangle$ 数据对构成，此时给定Target中的某个元素Query，通过计算Query和各个Key的相似性或者相关性，得到每个Key对应Value的权重系数，然后对Value进行加权求和，即得到了最终的Attention数值。所以本质上Attention机制是对Source中元素的Value值进行加权求和，而Query和Key用来计算对应Value的权重系数。即可以将其本质思想改写为如下公式：

$$\text{Attention}(\text{Query}, \text{Source}) = \sum_{i=1}^{L_s} \text{Similarity}(\text{Query}, \text{key}_i) * \text{Value}_i$$

其中， $L_s = |\text{Source}|$ 代表Source的长度，公式含义即如上所述。上文所举的机器翻译的例子里，因为在计算Attention的过程中，Source中的Key和Value合二为一，指向的是同一个东西，也即输入句子中每个单词对应的语义编码，所以可能不容易看出这种能够体现本质思想的结构。

至于Attention机制的具体计算过程，如果对目前大多数方法进行抽象的话，可以将其归纳为两个过程：第一个过程是根据Query和Key计算权重系数，第二个过程根据权重系数对Value进行加权求和。而第一个过程又可以细分为两个阶段：第一个阶段根据Query和Key计算两者的相似性或者相关性；第二个阶段对第一阶段的原始分值进行归一化处理；

4. Self-Attention模型

Self Attention也经常被称为intra Attention（内部Attention），最近一年也获得了比较广泛的使用，比如Google最新的机器翻译模型内部大量采用了Self Attention模型。

在一般任务的Encoder-Decoder框架中，输入Source和输出Target内容是不一样的，比如对于英-中机器翻译来说，Source是英文句子，Target是对应的翻译出的中文句子，Attention机制发生在Target的元素Query和Source中的所有元素之间。而**Self Attention**顾名思义，指的不是Target和Source之间的Attention机制，而是Source内部元素之间或者Target内部元素之间发生的Attention机制，也可以理解为**Target=Source**这种特殊情况下的注意力计算机制。其具体计算过程是一样的，只是计算对象发生了变化而已，所以此处不再赘述其计算过程细节。

很明显，引入Self Attention后会更容易捕获句子中长距离的相互依赖的特征，因为如果是RNN或者LSTM，需要依次序序列计算，对于远距离的相互依赖的特征，要经过若干时间步步骤的信息累积才能将两者联系起来，而距离越远，有效捕获的可能性越小。

但是Self Attention在计算过程中会直接将句子中任意两个单词的联系通过一个计算步骤直接联系起来，所以远距离依赖特征之间的距离被极大缩短，有利于有效地利用这些特征。除此外，Self Attention对于增加计算的并行性也有直接帮助作用。这是为何Self Attention逐渐被广泛使用的主要原因。

5. 发展

本质上，注意力机制能够为表征中较有价值的部分分配较多的计算资源。这个有趣的想法自提出后得到了快速发展，特别是启发了依靠注意力机制来编码输入序列并解码出输出序列的**变换器**

(Transformer) 模型的设计。变换器抛弃了卷积神经网络和循环神经网络的架构。它在计算效率上比基于循环神经网络的编码器—解码器模型通常更具明显优势。含注意力机制的变换器的编码结构在后来的**BERT**预训练模型中得以应用并令后者大放异彩：微调后的模型在多达11项自然语言处理任务中取得了当时最先进的结果。不久后，同样是基于变换器设计的**GPT-2模型**于新收集的语料数据集预训练后，在7个未参与训练的语言模型数据集上均取得了当时最先进的结果。除了自然语言处理领域，注意力机制还被广泛用于图像分类、自动图像描述、唇语解读以及语音识别。

6. 代码实现

[注意力模型实现中英文机器翻译](#)

1. 数据预处理

首先先下载本目录的数据和代码，并执行 [datautil.py](#)，生成中、英文字典。

2. 执行 [train.ipynb](#)，训练时间会比较长。
3. 测试模型，运行[test.py](#)文件。

7. 参考文献

[动手学深度学习](#)

[注意力机制的基本思想和实现原理](#)

作者:[@mantchs](#)

GitHub:<https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论！共同完善此项目！群号:【541954936】  加入QQ群

目录

- [1. 什么是Transformer](#)
- [2. Transformer结构](#)
 - [2.1 总体结构](#)
 - [2.2 Encoder层结构](#)
 - [2.3 Decoder层结构](#)
 - [2.4 动态流程图](#)
- [3. Transformer为什么需要进行Multi-head Attention](#)
- [4. Transformer相比于RNN/LSTM, 有什么优势? 为什么?](#)
- [5. 为什么说Transformer可以代替seq2seq?](#)
- [6. 代码实现](#)
- [7. 参考文献](#)

1. 什么是Transformer

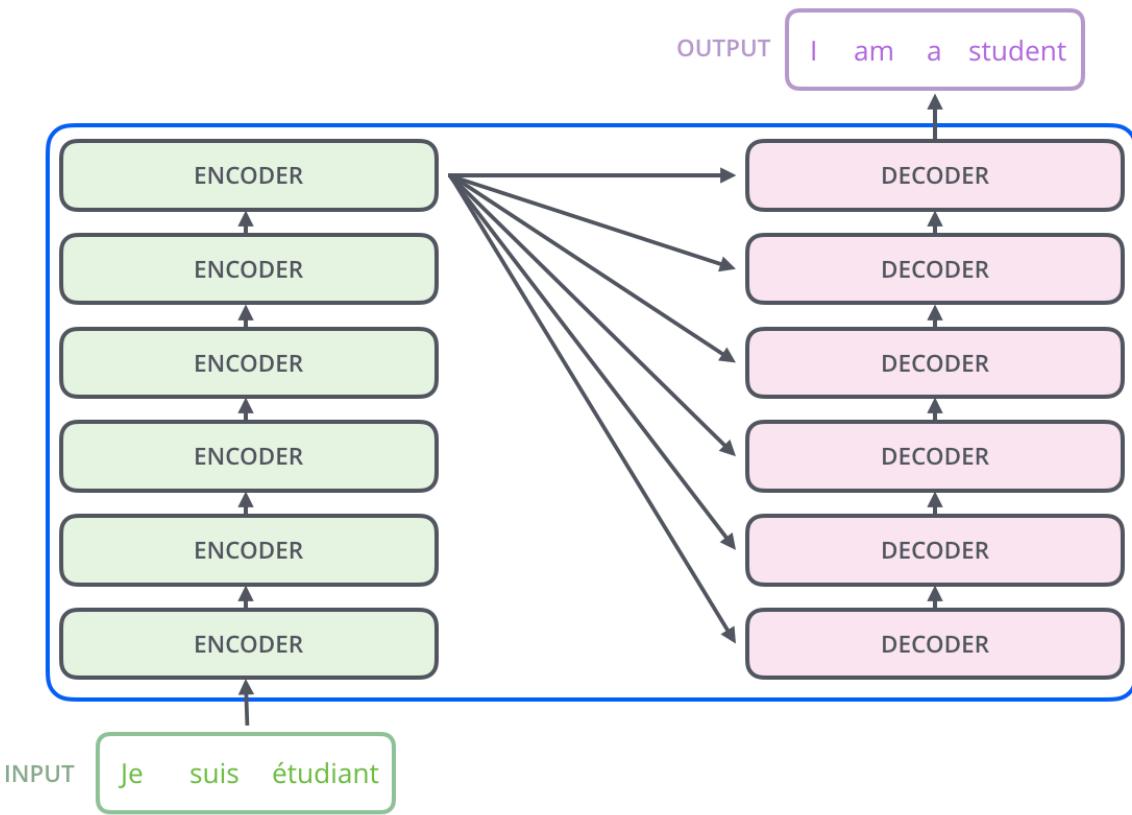
《Attention Is All You Need》是一篇Google提出的将Attention思想发挥到极致的论文。这篇论文中提出一个全新的模型，叫 Transformer，抛弃了以往深度学习任务里面使用到的 CNN 和 RNN。目前大热的Bert就是基于Transformer构建的，这个模型广泛应用于NLP领域，例如机器翻译，问答系统，文本摘要和语音识别等等方向。

2. Transformer结构

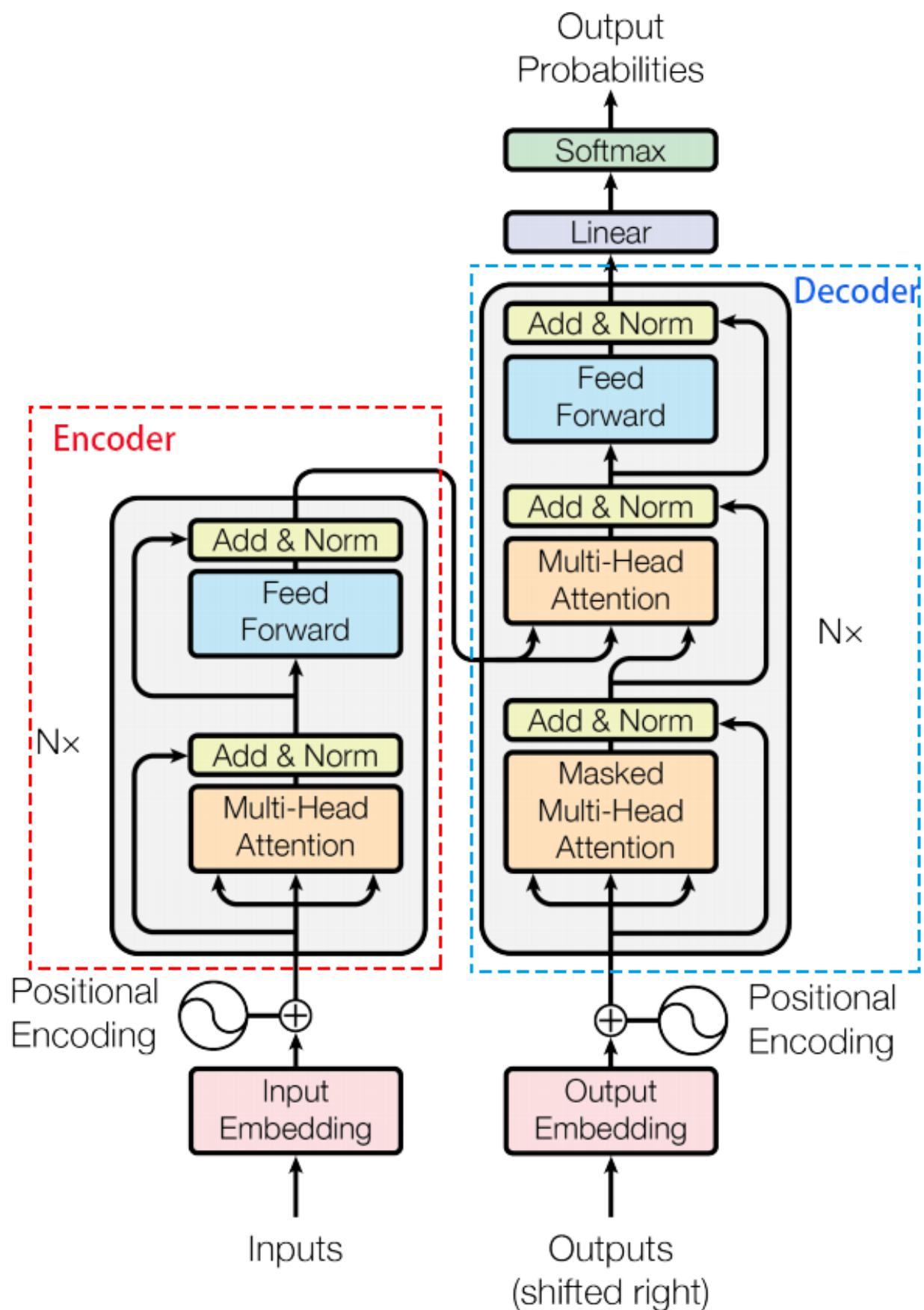
2.1 总体结构

Transformer的结构和Attention模型一样，Transformer模型中也采用了 encoder-decoder 架构。但其结构相比于Attention更加复杂，论文中encoder层由6个encoder堆叠在一起，decoder层也一样。

不了解Attention模型的，可以回顾之前的文章：[Attention](#)



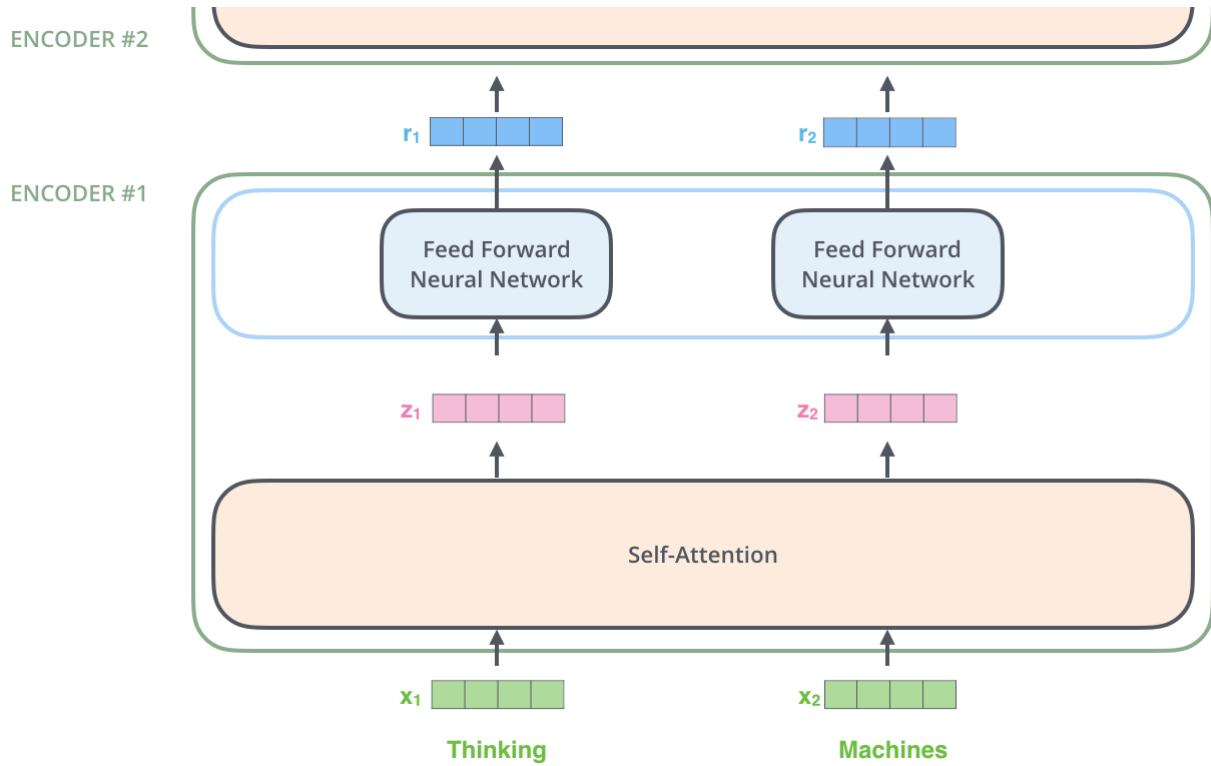
每一个encoder和decoder的内部结构如下图：



- encoder, 包含两层，一个self-attention层和一个前馈神经网络，self-attention能帮助当前节点不仅仅只关注当前的词，从而能获取到上下文的语义。
- decoder也包含encoder提到的两层网络，但是在这两层中间还有一层attention层，帮助当前节点获取到当前需要关注的重点内容。

2.2 Encoder层结构

首先，模型需要对输入的数据进行一个embedding操作，也可以理解为类似w2c的操作，embedding结束之后，输入到encoder层，self-attention处理完数据后把数据送给前馈神经网络，前馈神经网络的计算可以并行，得到的输出会输入到下一个encoder。



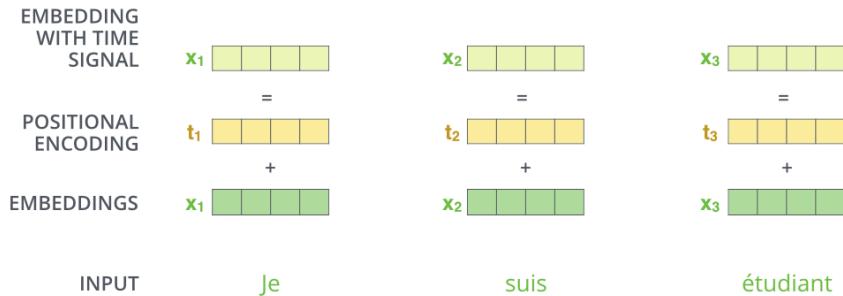
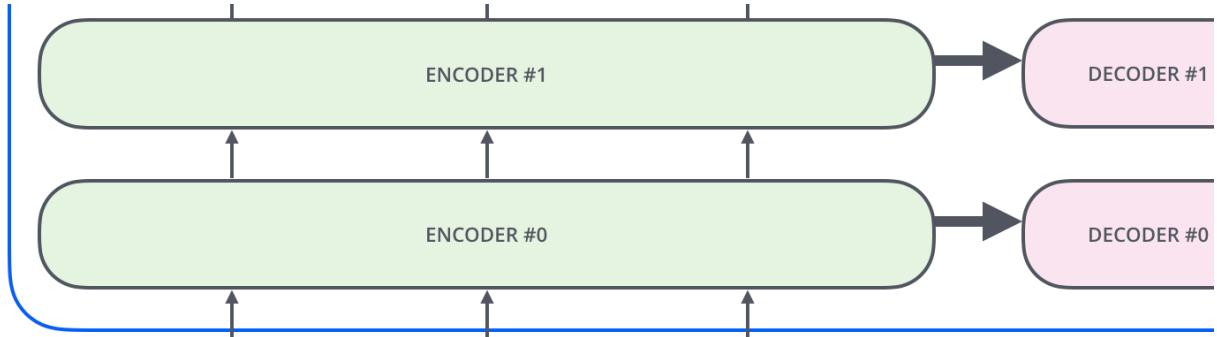
2.2.1 Positional Encoding

transformer模型中缺少一种解释输入序列中单词顺序的方法，它跟序列模型还不不一样。为了处理这个问题，transformer给encoder层和decoder层的输入添加了一个额外的向量Positional Encoding，维度和embedding的维度一样，这个向量采用了一种很独特的方法来让模型学习到这个值，这个向量能决定当前词的位置，或者说在一个句子中不同的词之间的距离。这个位置向量的具体计算方法有很多种，论文中的计算方法如下：

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$
$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

其中pos是指当前词在句子中的位置，i是指向量中每个值的index，可以看出，在偶数位置，使用正弦编码，在奇数位置，使用余弦编码。

最后把这个Positional Encoding与embedding的值相加，作为输入送到下一层。



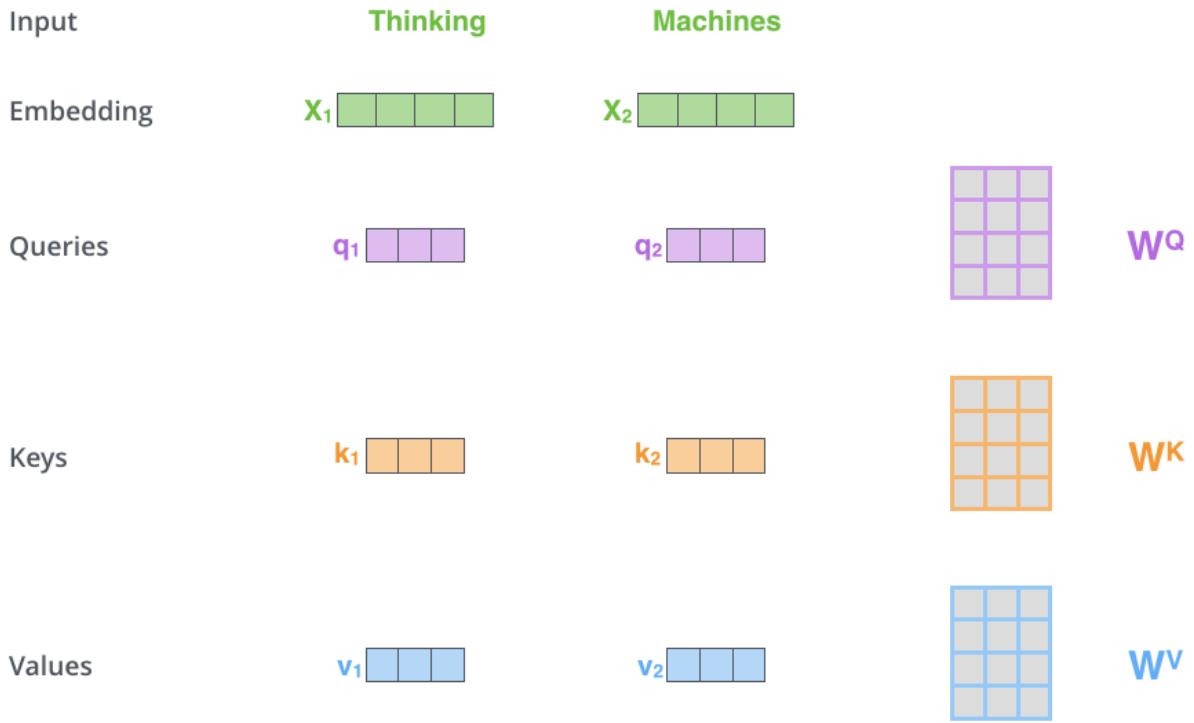
2.2.2 Self-Attention

接下来我们详细看一下self-attention，其思想和attention类似，但是self-attention是Transformer用来将其他相关单词的“理解”转换成我们正在处理的单词的一种思路，我们看个例子：

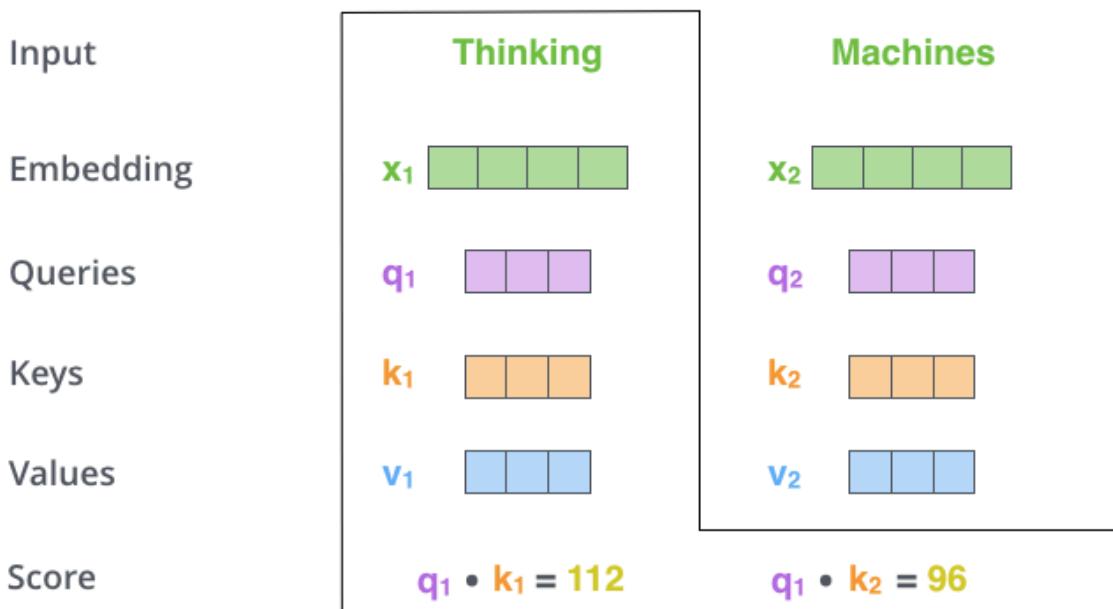
The animal didn't cross the street because it was too tired

这里的it到底代表的是animal还是street呢，对于我们来说能很简单的判断出来，但是对于机器来说，是很难判断的，self-attention就能够让机器把it和animal联系起来，接下来我们看下详细的处理过程。

- 首先，self-attention会计算出三个新的向量，在论文中，向量的维度是512维，我们把这三个向量分别称为Query、Key、Value，这三个向量是用embedding向量与一个矩阵相乘得到的结果，这个矩阵是随机初始化的，维度为(64, 512) 注意第二个维度需要和embedding的维度一样，其值在BP的过程中会一直进行更新，得到的这三个向量的维度是64。



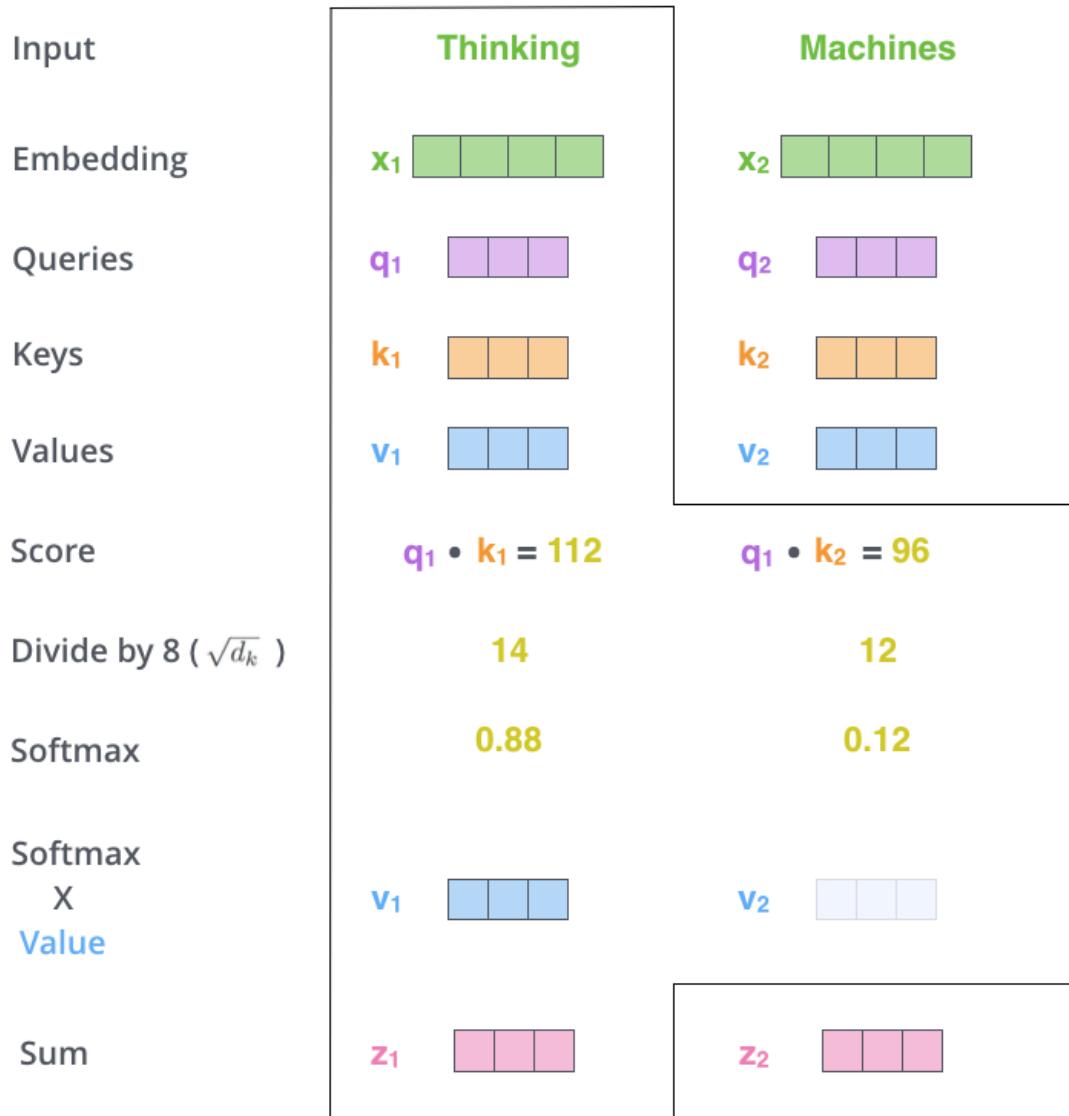
2. 计算self-attention的分数值，该分数值决定了当我们在某个位置encode一个词时，对输入句子的其他部分的关注程度。这个分数值的计算方法是Query与Key做点积，以下图为例，首先我们需要针对Thinking这个词，计算出其他词对于该词的一个分数值，首先是针对于自己本身即 $q_1 \cdot k_1$ ，然后是针对于第二个词即 $q_1 \cdot k_2$ 。



3. 接下来，把点积的结果除以一个常数，这里我们除以8，这个值一般是采用上文提到的矩阵的第一个维度的开方即64的开方8，当然也可以选择其他的值，然后把得到的结果做一个softmax的计算。得到的结果即是每个词对于当前位置的词的相关性大小，当然，当前位置的词相关性肯定会很大。

Input	Thinking		Machines	
Embedding	x_1	[]	x_2	[]
Queries	q_1	[]	q_2	[]
Keys	k_1	[]	k_2	[]
Values	v_1	[]	v_2	[]
Score		$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$	
Divide by 8 ($\sqrt{d_k}$)		14	12	
Softmax		0.88	0.12	

4. 下一步就是把Value和softmax得到的值进行相乘，并相加，得到的结果即是self-attention在当前节点的值。



在实际的应用场景，为了提高计算速度，我们采用的是矩阵的方式，直接计算出Query, Key, Value的矩阵，然后把embedding的值与三个矩阵直接相乘，把得到的新矩阵 Q 与 K 相乘，乘以一个常数，做 softmax 操作，最后乘上 V 矩阵。

这种通过 **query** 和 **key** 的相似性程度来确定 **value** 的权重分布的方法被称为**scaled dot-product attention**。

$$X \times W^Q = Q$$

A diagram illustrating the multiplication of matrix X by weight matrix W^Q to produce matrix Q . Matrix X is a green 4x4 grid. Weight matrix W^Q is a purple 4x4 grid. The result, matrix Q , is also a purple 4x4 grid.

$$X \times W^K = K$$

A diagram illustrating the multiplication of matrix X by weight matrix W^K to produce matrix K . Matrix X is a green 4x4 grid. Weight matrix W^K is an orange 4x4 grid. The result, matrix K , is an orange 4x4 grid.

$$X \times W^V = V$$

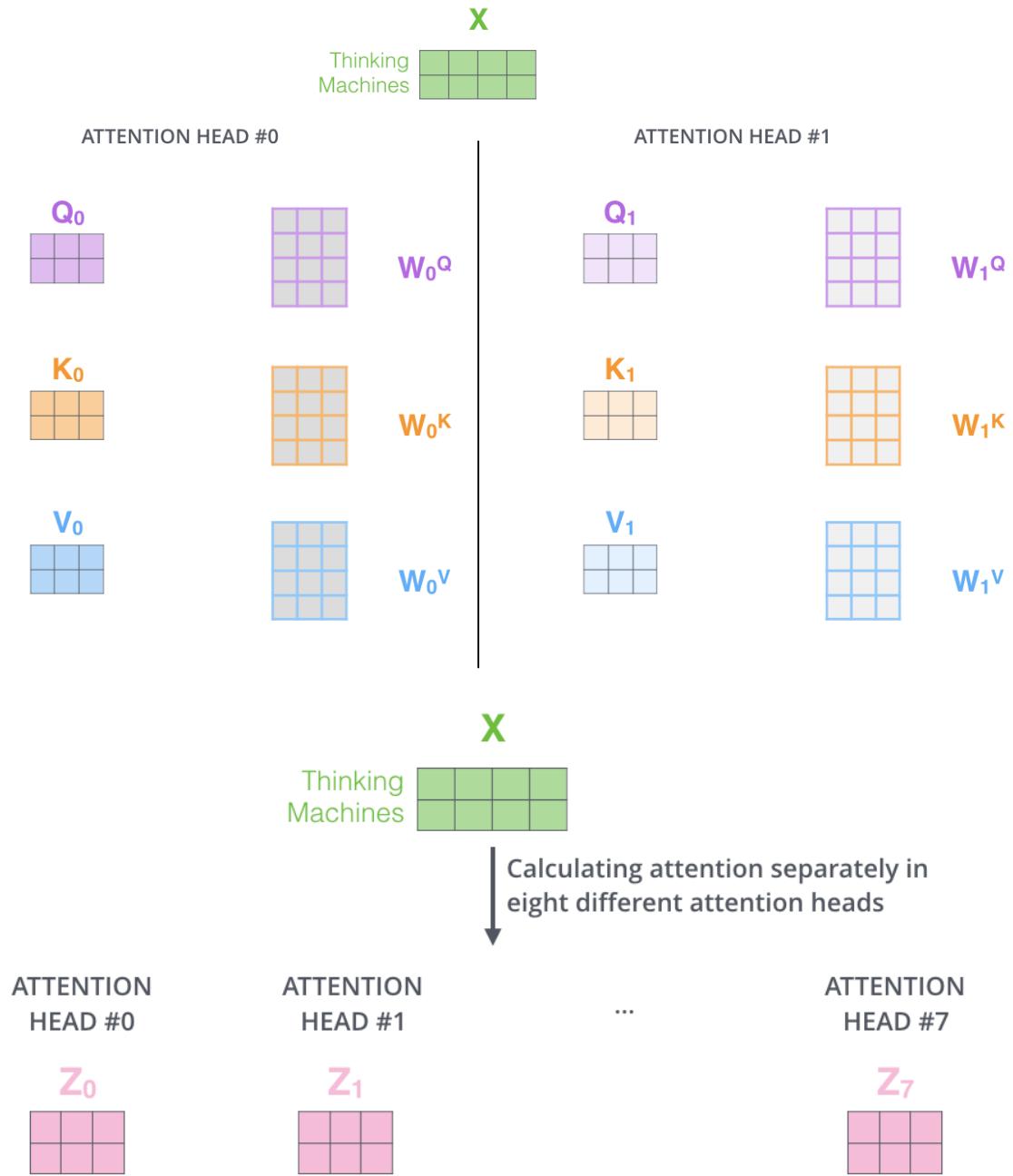
A diagram illustrating the multiplication of matrix X by weight matrix W^V to produce matrix V . Matrix X is a green 4x4 grid. Weight matrix W^V is a blue 4x4 grid. The result, matrix V , is a blue 4x4 grid.

$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) = Z$$

A diagram illustrating the computation of attention weights Z . It shows the softmax function applied to the product of matrices Q and K^T , scaled by $\frac{1}{\sqrt{d_k}}$. The result is matrix Z .

2.2.3 Multi-Headed Attention

这篇论文更牛逼的地方是给self-attention加入了另外一个机制，被称为“multi-headed” attention，该机制理解起来很简单，就是说不仅仅只初始化一组Q、K、V的矩阵，而是初始化多组，transformer是使用了8组，所以最后得到的结果是8个矩阵。



2.2.4 Layer normalization

在transformer中，每一个子层（self-attention, Feed Forward Neural Network）之后都会接一个残缺模块，并且有一个Layer normalization。

Normalization有很多种，但是它们都有一个共同的目的，那就是把输入转化成均值为0方差为1的数据。我们在把数据送入激活函数之前进行normalization（归一化），因为我们不希望输入数据落在激活函数的饱和区。

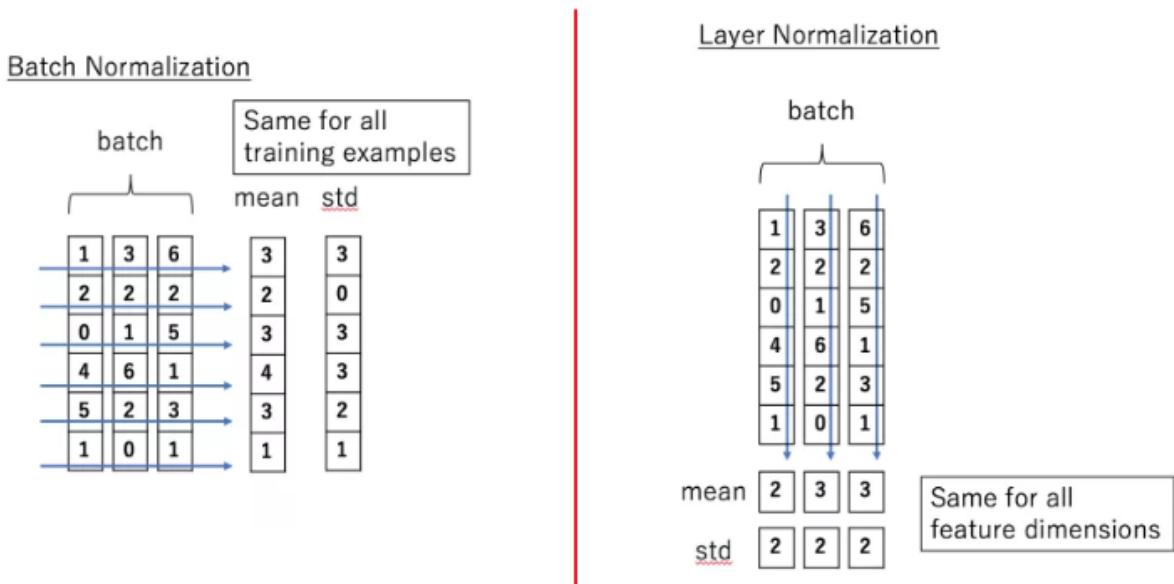
Batch Normalization

BN的主要思想就是：在每一层的每一批数据上进行归一化。我们可能会对输入数据进行归一化，但是经过该网络层的作用后，我们的数据已经不再是归一化的了。随着这种情况的发展，数据的偏差越来越大，我的反向传播需要考虑到这些大的偏差，这就迫使我们只能使用较小的学习率来防止梯度消失或者梯度爆炸。**BN的具体做法就是对每一小批数据，在批这个方向上做归一化。**

Layer normalization

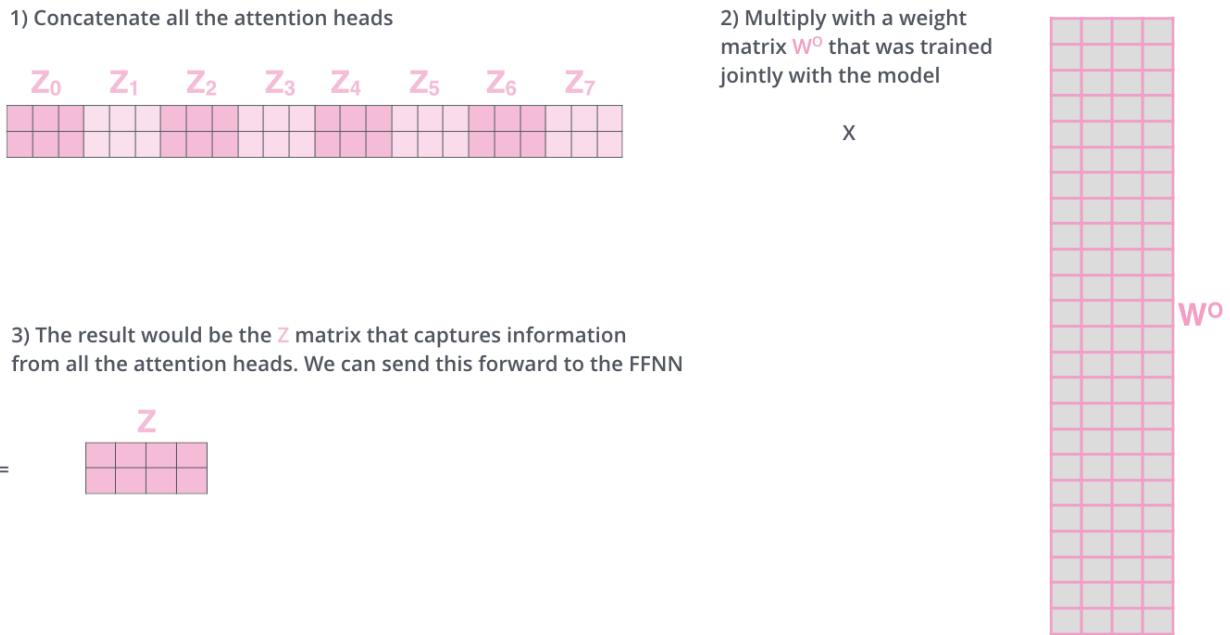
它也是归一化数据的一种方式，不过LN是在每一个样本上计算均值和方差，而不是BN那种在批方向计算均值和方差！公式如下：

$$LN(x_i) = \alpha * \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}} + \beta$$



2.2.5 Feed Forward Neural Network

这给我们留下了一个小的挑战，前馈神经网络没法输入8个矩阵呀，这该怎么办呢？所以我们需要一种方式，把8个矩阵降为1个，首先，我们把8个矩阵连在一起，这样会得到一个大的矩阵，再随机初始化一个矩阵和这个组合好的矩阵相乘，最后得到一个最终的矩阵。



2.3 Decoder层结构

根据上面的总体结构图可以看出，decoder部分其实和encoder部分大同小异，刚开始也是先添加一个位置向量Positional Encoding，方法和 2.2.1 节一样，接下来接的是masked multi-head attention，这里的mask也是transformer一个很关键的技术，下面我们会进行一一介绍。

其余的层结构与Encoder一样，请参考Encoder层结构。

2.3.1 masked multi-head attention

mask 表示掩码，它对某些值进行掩盖，使其在参数更新时不产生效果。Transformer 模型里面涉及两种 mask，分别是 padding mask 和 sequence mask。其中，padding mask 在所有的 scaled dot-product attention 里面都需要用到，而 sequence mask 只有在 decoder 的 self-attention 里面用到。

1. padding mask

什么是 padding mask 呢？因为每个批次输入序列长度是不一样的也就是说，我们要对输入序列进行对齐。具体来说，就是给在较短的序列后面填充 0。但是如果输入的序列太长，则是截取左边的内容，把多余的直接舍弃。因为这些填充的位置，其实是没什么意义的，所以我们的attention机制不应该把注意力放在这些位置上，所以我们需要进行一些处理。

具体的做法是，把这些位置的值加上一个非常大的负数(负无穷)，这样的话，经过 softmax，这些位置的概率就会接近0！

而我们的 padding mask 实际上是一个张量，每个值都是一个 Boolean，值为 false 的地方就是我们要进行处理的地方。

2. Sequence mask

文章前面也提到，sequence mask 是为了使得 decoder 不能看见未来的信息。也就是对于一个序列，在 time_step 为 t 的时刻，我们的解码输出应该只能依赖于 t 时刻之前的输出，而不能依赖 t 之后的输出。因此我们需要想一个办法，把 t 之后的信息给隐藏起来。

那么具体怎么做呢？也很简单：产生一个上三角矩阵，上三角的值全为0。把这个矩阵作用在每一个序列上，就可以达到我们的目的。

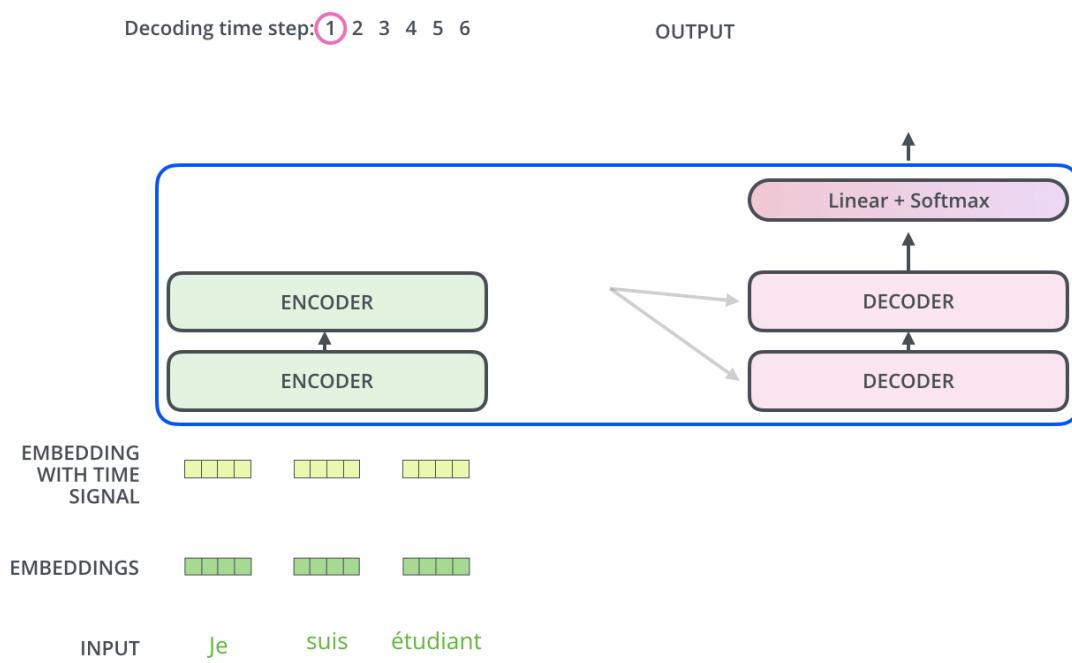
- 对于 decoder 的 self-attention，里面使用到的 scaled dot-product attention，同时需要padding mask 和 sequence mask 作为 attn_mask，具体实现就是两个mask相加作为attn_mask。
- 其他情况，attn_mask一律等于 padding mask。

2.3.2 Output层

当decoder层全部执行完毕后，怎么把得到的向量映射为我们需要的词呢，很简单，只需要在结尾再添加一个全连接层和softmax层，假如我们的词典是1w个词，那最终softmax会输入1w个词的概率，概率值最大的对应的词就是我们最终的结果。

2.4 动态流程图

编码器通过处理输入序列开启工作。顶端编码器的输出之后会变转化为一个包含向量K（键向量）和V（值向量）的注意力向量集，这是并行化操作。这些向量将被每个解码器用于自身的“编码-解码注意力层”，而这些层可以帮助解码器关注输入序列哪些位置合适：



在完成编码阶段后，则开始解码阶段。解码阶段的每个步骤都会输出一个输出序列（在这个例子里，是英语翻译的句子）的元素。

接下来的步骤重复了这个过程，直到到达一个特殊的终止符号，它表示transformer的解码器已经完成了它的输出。每个步骤的输出在下一个时间步被提供给低端解码器，并且就像编码器之前做的那样，这些解码器会输出它们的解码结果。

[解码器动态图请点击](#)

3. Transformer为什么需要进行Multi-head Attention

原论文中说到进行Multi-head Attention的原因是将模型分为多个头，形成多个子空间，可以让模型去关注不同方面的信息，最后再将各个方面信息综合起来。其实直观上也可以想到，如果自己设计这样的一个模型，必然也不会只做一次attention，多次attention综合的结果至少能够起到增强模型的作用，也可以类比CNN中同时使用多个卷积核的作用，直观上讲，多头的注意力有助于网络捕捉到更丰富的特征/信息。

4. Transformer相比于RNN/LSTM，有什么优势？为什么？

1. RNN系列的模型，并行计算能力很差。RNN并行计算的问题就出在这里，因为 T 时刻的计算依赖 $T-1$ 时刻的隐层计算结果，而 $T-1$ 时刻的计算依赖 $T-2$ 时刻的隐层计算结果，如此下去就形成了所谓的序列依赖关系。
2. Transformer的特征抽取能力比RNN系列的模型要好。

具体实验对比可以参考：[放弃幻想，全面拥抱Transformer：自然语言处理三大特征抽取器（CNN/RNN/TF）比较](#)

但是值得注意的是，并不是说Transformer就能够完全替代RNN系列的模型了，任何模型都有其适用范围，同样的，RNN系列模型在很多任务上还是首选，熟悉各种模型的内部原理，知其然且知其所以然，才能遇到新任务时，快速分析这时候该用什么样的模型，该怎么做好。

5. 为什么说Transformer可以代替seq2seq？

seq2seq缺点：这里用代替这个词略显不妥当，seq2seq虽已老，但始终还是有其用武之地，seq2seq最大的问题在于将Encoder端的所有信息压缩到一个固定长度的向量中，并将其作为Decoder端首个隐藏状态的输入，来预测Decoder端第一个单词(token)的隐藏状态。在输入序列比较长的时候，这样做显然会损失Encoder端的很多信息，而且这样一股脑的把该固定向量送入Decoder端，Decoder端不能够关注到其想要关注的信息。

Transformer优点：transformer不但对seq2seq模型这两点缺点有了实质性的改进(多头交互式attention模块)，而且还引入了self-attention模块，让源序列和目标序列首先“自关联”起来，这样的话，源序列和目标序列自身的embedding表示所蕴含的信息更加丰富，而且后续的FFN层也增强了模型的表达能力，并且Transformer并行计算的能力是远远超过seq2seq系列的模型，因此我认为这是transformer优于seq2seq模型的地方。

6. 代码实现

地址：<https://github.com/Kyubyong/transformer>

7. 参考文献

- [Transformer模型详解](#)
 - [图解Transformer \(完整版\)](#)
 - [关于Transformer的若干问题整理记录](#)
-

作者：[@mantchs](#)

GitHub：<https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论！共同完善此项目！群号：【541954936】[!\[\]\(7c2b9810f9235b80f896ccb0dcbb3827_img.jpg\) 加入QQ群](#)

目录

- [1. 什么是BERT](#)
- [2. 从Word Embedding到Bert模型的发展](#)
 - [2.1 图像的预训练](#)
 - [2.2 Word Embedding](#)
 - [2.3 ELMO](#)
 - [2.4 GPT](#)
 - [2.5 BERT](#)
- [3. BERT的评价](#)
- [4. 代码实现](#)
- [5. 参考文献](#)

1. 什么是BERT

BERT的全称是**Bidirectional Encoder Representation from Transformers**, 是Google2018年提出的预训练模型, 即双向Transformer的Encoder, 因为decoder是不能获要预测的信息的。模型的主要创新点都在pre-train方法上, 即用了Masked LM和Next Sentence Prediction两种方法分别捕捉词语和句子级别的representation。

Bert最近很火, 应该是最近最火爆的AI进展, 网上的评价很高, 那么Bert值得这么高的评价吗? 我个人判断是值得。那为什么会有这么高的评价呢? 是因为它有重大的理论或者模型创新吗? 其实并没有, 从模型创新角度看一般, 创新不算大。但是架不住效果太好了, 基本刷新了很多NLP的任务的最好性能, 有些任务还被刷爆了, 这个才是关键。另外一点是Bert具备广泛的通用性, 就是说绝大部分NLP任务都可以采用类似的两阶段模式直接去提升效果, 这个第二关键。客观的说, 把Bert当做最近两年NLP重大进展的集大成者更符合事实。

2. 从Word Embedding到Bert模型的发展

2.1 图像的预训练

自从深度学习火起来后, 预训练过程就是做图像或者视频领域的一种比较常规的做法, 有比较长的历史了, 而且这种做法很有效, 能明显促进应用的效果。

预训练在图像领域的应用

预训练（say, imagenet）在图像领域广泛使用

1. 训练数据小，不足以训练复杂网络
2. 加快训练速度
3. 参数初始化，先找到好的初始点，有利于优化

Frozen

Fine-Tuning



那么图像领域怎么做预训练呢，上图展示了这个过程，

1. 我们设计好网络结构以后，对于图像来说一般是CNN的多层叠加网络结构，可以先用某个训练集合比如训练集合A或者训练集合B对这个网络进行预先训练，在A任务上或者B任务上学会网络参数，然后存起来以备后用。
2. 假设我们面临第三个任务C，网络结构采取相同的网络结构，在比较浅的几层CNN结构，网络参数初始化的时候可以加载A任务或者B任务学习好的参数，其它CNN高层参数仍然随机初始化。
3. 之后我们用C任务的训练数据来训练网络，此时有两种做法：
一种是浅层加载的参数在训练C任务过程中不动，这种方法被称为“Frozen”；
另一种是底层网络参数尽管被初始化了，在C任务训练过程中仍然随着训练的进程不断改变，这种一般叫“Fine-Tuning”，顾名思义，就是更好地把参数进行调整使得更适应当前的C任务。

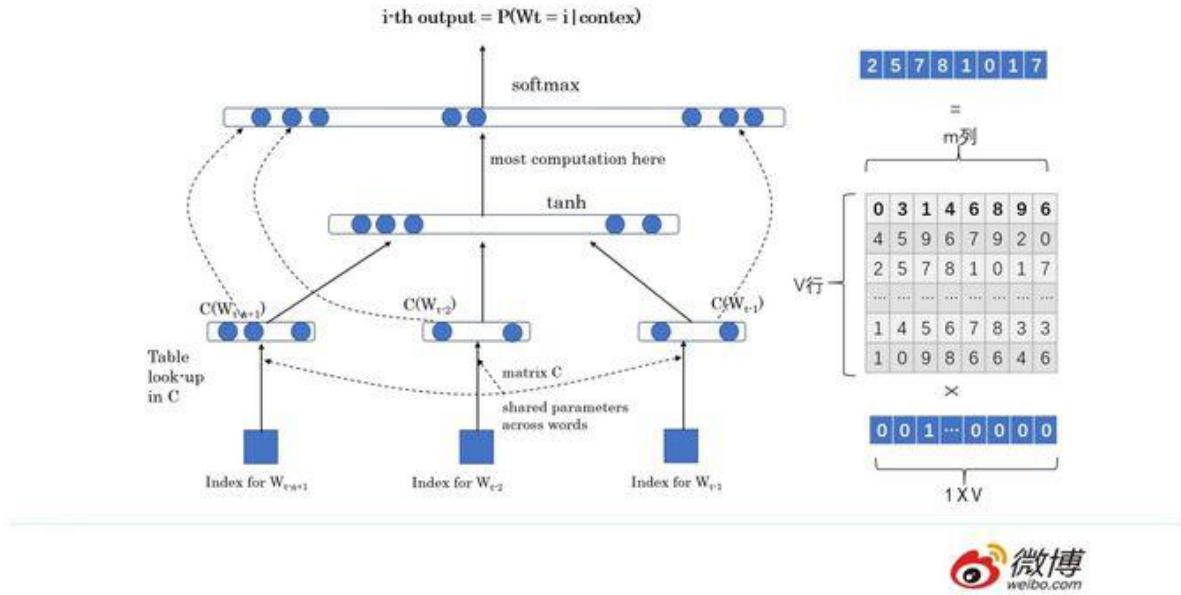
一般图像或者视频领域要做预训练一般都这么做。这样做的优点是：如果手头任务C的训练集合数据量较少的话，利用预训练出来的参数来训练任务C，加个预训练过程也能极大加快任务训练的收敛速度，所以这种预训练方式是老少皆宜的解决方案，另外疗效又好，所以在做图像处理领域很快就流行开来。

为什么预训练可行

对于层级的CNN结构来说，不同层级的神经元学习到了不同类型的图像特征，由底向上特征形成层级结构，所以预训练好的网络参数，尤其是底层的网络参数抽取出特征跟具体任务越无关，越具备任务的通用性，所以这是为何一般用底层预训练好的参数初始化新任务网络参数的原因。而高层特征跟任务关联较大，实际可以不用使用，或者采用Fine-tuning用新数据集合清洗掉高层无关的特征抽取器。

2.2 Word Embedding

神经网络语言模型 (NNLM)



神经网络语言模型(NNLM)的思路。先说训练过程。学习任务是输入某个句中单词 $W_t = \text{(Bert)}$ 前面句子的 $t-1$ 个单词，要求网络正确预测单词 Bert，即最大化：

$$P(W_t = \text{"Bert"} | W_1, W_2, \dots, W_{(t-1)}; \theta)$$

前面任意单词 W_i 用Onehot编码（比如：0001000）作为原始单词输入，之后乘以矩阵 Q 后获得向量 $C(W_i)$ ，每个单词的 $C(W_i)$ 拼接，上接隐层，然后接softmax去预测后面应该后续接哪个单词。这个 $C(W_i)$ 是什么？这其实就是单词对应的Word Embedding值，那个矩阵 Q 包含 V 行， V 代表词典大小，每一行内容代表对应单词的Word embedding值。只不过 Q 的内容也是网络参数，需要学习获得，训练刚开始用随机值初始化矩阵 Q ，当这个网络训练好之后，矩阵 Q 的内容被正确赋值，每一行代表一个单词对应的Word embedding值。所以你看，通过这个网络学习语言模型任务，这个网络不仅自己能够根据上文预测后接单词是什么，同时获得一个副产品，就是那个矩阵 Q ，这就是单词的Word Embedding。

2013年最火的用语言模型做Word Embedding的工具是Word2Vec，后来又出了Glove，Word2Vec。对于这两个模型不熟悉的可以参考我之前的文章，这里不再赘述：

- [Word2Vec](#)
- [GloVe](#)

上面这种模型做法就是18年之前NLP领域里面采用预训练的典型做法，之前说过，Word Embedding其实对于很多下游NLP任务是有帮助的，只是帮助没有大到闪瞎忘记戴墨镜的围观群众的双眼而已。那么新问题来了，为什么这样训练及使用Word Embedding的效果没有期待中那么好呢？答案很简单，因为Word Embedding有问题呗。这貌似是个比较弱智的答案，关键是Word Embedding存在什么问题？这其实是个好问题。

这片在Word Embedding头上笼罩了好几年的乌云是什么？是多义词问题。我们知道，多义词是自然语言中经常出现的现象，也是语言灵活性和高效性的一种体现。多义词对Word Embedding来说有什么负面影响？如上图所示，比如多义词Bank，有两个常用含义，但是Word Embedding在对bank这个单词进行编码的时候，是区分不开这两个含义的，因为它们尽管上下文环境中出现的单词不同，但是在用语言模型训练的时候，不论什么上下文的句子经过word2vec，都是预测相同的单词bank，而同一个单词占

的是同一行的参数空间，这导致两种不同的上下文信息都会编码到相同的word embedding空间里去。所以word embedding无法区分多义词的不同语义，这就是它的一个比较严重的问题。

有没有简单优美的解决方案呢？ELMO提供了一种简洁优雅的解决方案。

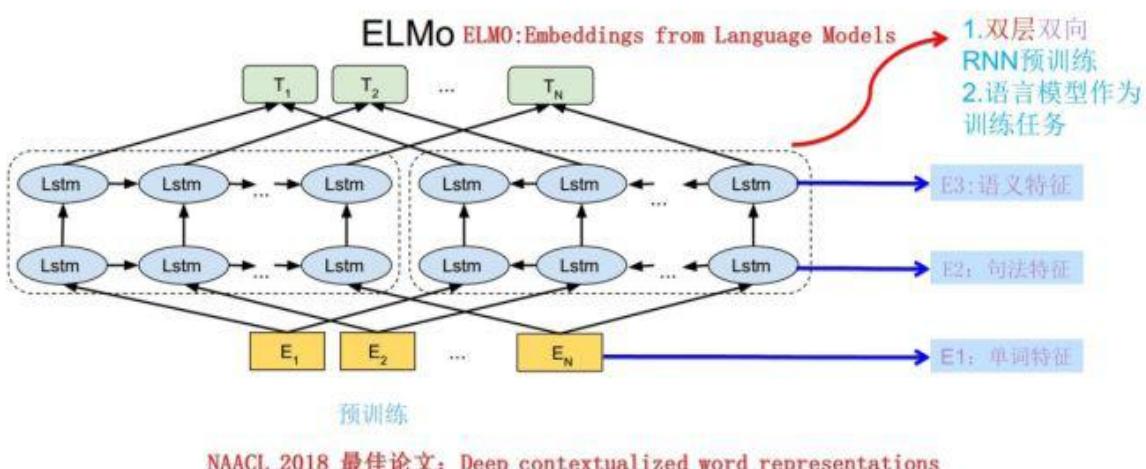
2.3 ELMO

ELMO是“Embedding from Language Models”的简称，其实这个名字并没有反应它的本质思想，提出ELMO的论文题目：“Deep contextualized word representation”更能体现其精髓，而精髓在哪里？在deep contextualized这个短语，一个是deep，一个是context，其中context更关键。

在此之前的Word Embedding本质上是个静态的方式，所谓静态指的是训练好之后每个单词的表达就固定住了，以后使用的时候，不论新句子上下文单词是什么，这个单词的Word Embedding不会跟着上下文场景的变化而改变，所以对于比如Bank这个词，它事先学好的Word Embedding中混合了几种语义，在应用中来了个新句子，即使从上下文中（比如句子包含money等词）明显可以看出它代表的是“银行”的含义，但是对应的Word Embedding内容也不会变，它还是混合了多种语义。这是为何说它是静态的，这也是问题所在。

ELMO的本质思想是：我事先用语言模型学好一个单词的Word Embedding，此时多义词无法区分，不过这没关系。在我实际使用Word Embedding的时候，单词已经具备了特定的上下文了，这个时候我可以根据上下文单词的语义去调整单词的Word Embedding表示，这样经过调整后的Word Embedding更能表达在这个上下文中的具体含义，自然也就解决了多义词的问题了。所以ELMO本身是个根据当前上下文对Word Embedding动态调整的思路。

从WE到ELMO：基于上下文的Embedding



ELMO采用了典型的两阶段过程，第一个阶段是利用语言模型进行预训练；第二个阶段是在做下游任务时，从预训练网络中提取对应单词的网络各层的Word Embedding作为新特征补充到下游任务中。

上图展示的是其预训练过程，它的网络结构采用了双层双向LSTM，目前语言模型训练的任务目标是根据单词 W_i 的上下文去正确预测单词 W_i ， W_i 之前的单词序列Context-before称为上文，之后的单词序列Context-after称为下文。

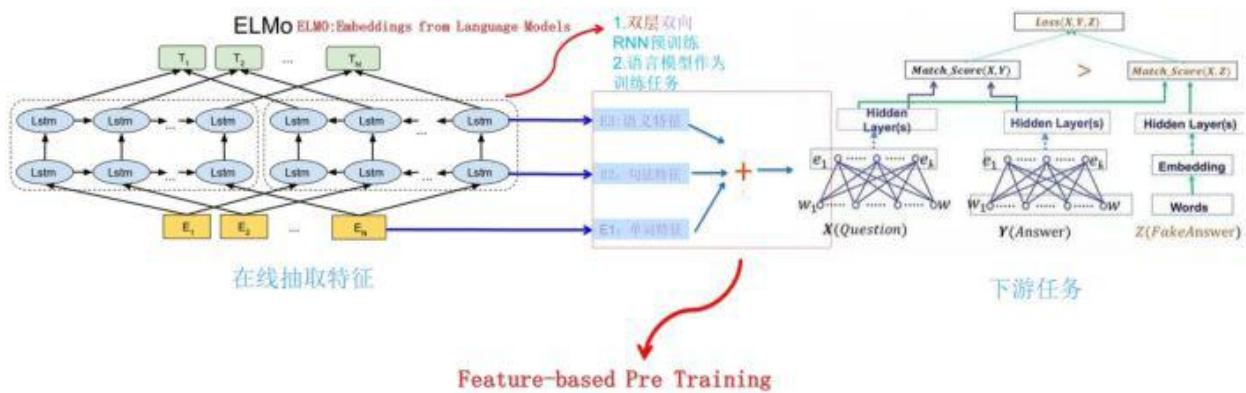
图中左端的前向双层LSTM代表正方向编码器，输入的是从左到右顺序的除了预测单词外 \mathbf{W}_i 的上文 Context-before；右端的逆向双层LSTM代表反方向编码器，输入的是从右到左的逆序的句子下文 Context-after；每个编码器的深度都是两层LSTM叠加。

这个网络结构其实在NLP中是很常用的。使用这个网络结构利用大量语料做语言模型任务就能预先训练好这个网络，如果训练好这个网络后，输入一个新句子 S_{new} ，句子中每个单词都能得到对应的三个 Embedding：

- 最底层是单词的Word Embedding；
- 往上走是第一层双向LSTM中对应单词位置的Embedding，这层编码单词的句法信息更多一些；
- 再往上走是第二层LSTM中对应单词位置的Embedding，这层编码单词的语义信息更多一些。

也就是说，ELMO的预训练过程不仅仅学会单词的Word Embedding，还学会了一个双层双向的LSTM网络结构，而这两者后面都有用。

ELMO：训练好之后如何使用？



上面介绍的是ELMO的第一阶段：预训练阶段。那么预训练好网络结构后，如何给下游任务使用呢？上图展示了下游任务的使用过程，比如我们的下游任务仍然是QA问题：

1. 此时对于问句X，我们可以先将句子X作为预训练好的ELMO网络的输入，这样句子X中每个单词在ELMO网络中都能获得对应的三个Embedding；
2. 之后给予这三个Embedding中的每一个Embedding一个权重a，这个权重可以学习得来，根据各自权重累加求和，将三个Embedding整合成一个；
3. 然后将整合后的这个Embedding作为X句在自己任务的那个网络结构中对应单词的输入，以此作为补充的新特征给下游任务使用。对于上图所示下游任务QA中的回答句子Y来说也是如此处理。

因为ELMO给下游提供的是每个单词的特征形式，所以这一类预训练的方法被称为“Feature-based Pre-Training”。

前面我们提到静态Word Embedding无法解决多义词的问题，那么ELMO引入上下文动态调整单词的embedding后多义词问题解决了吗？解决了，而且比我们期待的解决得还要好。对于Glove训练出的Word Embedding来说，多义词比如play，根据它的embedding找出的最接近的其它单词大多数集中在体育领域，这很明显是因为训练数据中包含play的句子中体育领域的数量明显占优导致；而使用ELMO，根据上下文动态调整后的embedding不仅能够找出对应的“演出”的相同语义的句子，而且还可以保证找

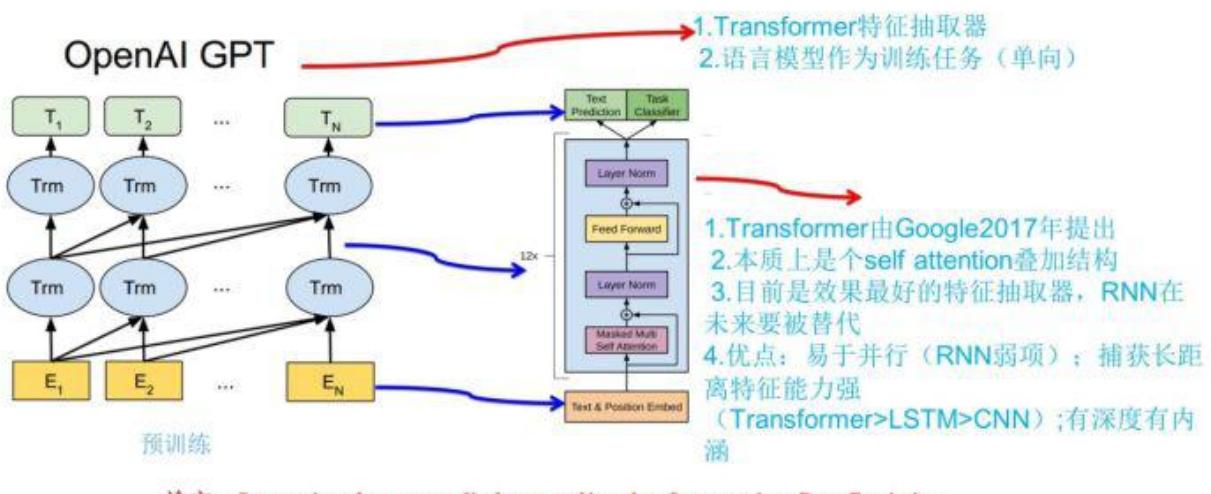
出的句子中的play对应的词性也是相同的，这是超出期待之处。之所以会这样，是因为我们上面提到过，第一层LSTM编码了很多句法信息，这在这里起到了重要作用。

ELMO有什么值得改进的缺点呢？

- 首先，一个非常明显的缺点在特征抽取器选择方面，ELMO使用了LSTM而不是新贵Transformer，Transformer是谷歌在17年做机器翻译任务的“Attention is all you need”的论文中提出的，引起了相当大的反响，很多研究已经证明了Transformer提取特征的能力是要远强于LSTM的。如果ELMO采取Transformer作为特征提取器，那么估计Bert的反响远不如现在的这种火爆场面。
- 另外一点，ELMO采取双向拼接这种融合特征的能力可能比Bert一体化的融合特征方式弱，但是，这只不过是一种从道理推断产生的怀疑，目前并没有具体实验说明这一点。

2.4 GPT

从WE到GPT：Pretrain+Finetune两阶段过程



GPT是“Generative Pre-Training”的简称，从名字看其含义是指的生成式的预训练。GPT也采用两阶段过程，第一个阶段是利用语言模型进行预训练，第二阶段通过Fine-tuning的模式解决下游任务。

上图展示了GPT的预训练过程，其实和ELMO是类似的，主要不同在于两点：

- 首先，特征抽取器不是用的RNN，而是用的Transformer，上面提到过它的特征抽取能力要强于RNN，这个选择很明显是很明智的；
- 其次，GPT的预训练虽然仍然是以语言模型作为目标任务，但是采用的是单向的语言模型，所谓“单向”的含义是指：语言模型训练的任务目标是根据 W_i 单词的上下文去正确预测单词 W_i ， W_i 之前的单词序列Context-before称为上文，之后的单词序列Context-after称为下文。

如果对Transformer模型不太了解的，可以参考我写的文章：[Transformer](#)

ELMO在做语言模型预训练的时候，预测单词 W_i 同时使用了上文和下文，而GPT则只采用Context-before这个单词的上文来进行预测，而抛开了下文。这个选择现在看不是个太好的选择，原因很简单，它没有把单词的下文融合进来，这限制了其在更多应用场景的效果，比如阅读理解这种任务，在做任务的时候是可以允许同时看到上文和下文一起做决策的。如果预训练时候不把单词的下文嵌入到Word

Embedding中，是很吃亏的，白白丢掉了很多信息。

2.5 BERT

Bert采用和GPT完全相同的两阶段模型，首先是语言模型预训练；其次是使用Fine-Tuning模式解决下游任务。和GPT的主要不同在于在预训练阶段采用了类似ELMO的双向语言模型，即双向的Transformer，当然另外一点是语言模型的数据规模要比GPT大。所以这里Bert的预训练过程不必多讲了。模型结构如下：

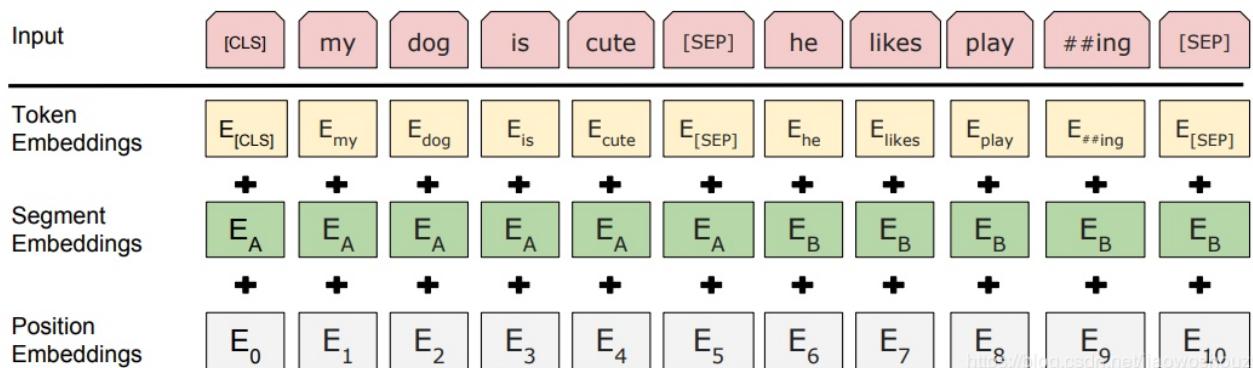
对比OpenAI GPT(Generative pre-trained transformer)，BERT是双向的Transformer block连接；就像单向rnn和双向rnn的区别，直觉上来讲效果会好一些。

对比ELMo，虽然都是“双向”，但目标函数其实是不同的。ELMo是分别以 $P(w_i|w_1, \dots, w_{i-1})$ 和 $P(w_i|w_{i+1}, \dots, w_n)$ 作为目标函数，独立训练处两个representation然后拼接，而BERT则是以 $P(w_i|w_1, \dots, w_{i-1}, w_{i+1}, \dots, w_n)$ 作为目标函数训练LM。

BERT预训练模型分为以下三个步骤： **Embedding**、**Masked LM**、**Next Sentence Prediction**

2.5.1 Embedding

这里的Embedding由三种Embedding求和而成：



- Token Embeddings是词向量，第一个单词是CLS标志，可以用于之后的分类任务
- Segment Embeddings用来区别两种句子，因为预训练不光做LM还要做以两个句子为输入的分类任务
- Position Embeddings和之前文章中的Transformer不一样，不是三角函数而是学习出来的

2.5.2 Masked LM

MLM可以理解为完形填空，作者会随机mask每一个句子中15%的词，用其上下文来做预测，例如： my dog is hairy → my dog is [MASK]

此处将hairy进行了mask处理，然后采用非监督学习的方法预测mask位置的词是什么，但是该方法有一个问题，因为是mask15%的词，其数量已经很高了，这样就会导致某些词在fine-tuning阶段从未见过，为了解决这个问题，作者做了如下的处理：

80%是采用[mask]， my dog is hairy → my dog is [MASK]

10%是随机取一个词来代替mask的词， my dog is hairy -> my dog is apple

10%保持不变， my dog is hairy -> my dog is hairy

注意：这里的10%是15%需要mask中的10%

那么为啥要以一定的概率使用随机词呢？这是因为transformer要保持对每个输入token分布式的表征，否则Transformer很可能会记住这个[MASK]就是“hairy”。至于使用随机词带来的负面影响，文章中解释说，所有其他的token(即非“hairy”的token)共享 $15\% * 10\% = 1.5\%$ 的概率，其影响是可以忽略不计的。Transformer全局的可视，又增加了信息的获取，但是不让模型获取全量信息。

2.5.3 Next Sentence Prediction

选择一些句子对A与B，其中50%的数据B是A的下一条句子，剩余50%的数据B是语料库中随机选择的，学习其中的相关性，添加这样的预训练的目的是目前很多NLP的任务比如QA和NLI都需要理解两个句子之间的关系，从而能让预训练的模型更好的适应这样的任务。

个人理解：

- Bert先是用Mask来提高视野范围的信息获取量，增加duplicate再随机Mask，这样跟RNN类方法依次训练预测没什么区别了除了mask不同位置外；
- 全局视野极大地降低了学习的难度，然后再用A+B/C来作为样本，这样每条样本都有50%的概率看到一半左右的噪声；
- 但直接学习Mask A+B/C是没法学习的，因为不知道哪些是噪声，所以又加上next_sentence预测任务，与MLM同时进行训练，这样用next来辅助模型对噪声/非噪声的辨识，用MLM来完成语义的大部分的学习。

3. BERT的评价

总结下BERT的主要贡献：

- 引入了Masked LM，使用双向LM做模型预训练。
- 为预训练引入了新目标NSP，它可以学习句子与句子间的关系。
- 进一步验证了更大的模型效果更好：12 -> 24层。
- 为下游任务引入了很通用的求解框架，不再为任务做模型定制。
- 刷新了多项NLP任务的记录，引爆了NLP无监督预训练技术。

BERT优点

- Transformer Encoder因为有Self-attention机制，因此BERT自带双向功能。
- 因为双向功能以及多层Self-attention机制的影响，使得BERT必须使用Cloze版的语言模型Masked-LM来完成token级别的预训练。
- 为了获取比词更高级别的句子级别的语义表征，BERT加入了Next Sentence Prediction来和Masked-LM一起做联合训练。
- 为了适配多任务下的迁移学习，BERT设计了更通用的输入层和输出层。
- 微调成本小。

BERT缺点

- task1的随机遮挡策略略显粗犷，推荐阅读《Data Noising As Smoothing In Neural Network

Language Models》。

- [MASK]标记在实际预测中不会出现，训练时用过多[MASK]影响模型表现。每个batch只有15%的token被预测，所以BERT收敛得比left-to-right模型要慢（它们会预测每个token）。
- BERT对硬件资源的消耗巨大（大模型需要16个tpu，历时四天；更大的模型需要64个tpu，历时四天）。

评价

Bert是NLP里里程碑式的工作，对于后面NLP的研究和工业应用会产生长久的影响，这点毫无疑问。但是从上文介绍也可以看出，从模型或者方法角度看，Bert借鉴了ELMO，GPT及CBOW，主要提出了Masked 语言模型及Next Sentence Prediction，但是这里Next Sentence Prediction基本不影响大局，而Masked LM明显借鉴了CBOW的思想。所以说Bert的模型没什么大的创新，更像最近几年NLP重要进展的集大成者，这点如果你看懂了上文估计也没有太大异议，如果你有大的异议，杠精这个大帽子我随时准备戴给你。如果归纳一下这些进展就是：

- 首先是两阶段模型，第一阶段双向语言模型预训练，这里注意要用双向而不是单向，第二阶段采用具体任务Fine-tuning或者做特征集成；
- 第二是特征抽取要用Transformer作为特征提取器而不是RNN或者CNN；
- 第三，双向语言模型可以采取CBOW的方法去做（当然我觉得这个是个细节问题，不算太关键，前两个因素比较关键）。

Bert最大的亮点在于效果好及普适性强，几乎所有NLP任务都可以套用Bert这种两阶段解决思路，而且效果应该会有明显提升。可以预见的是，未来一段时间在NLP应用领域，Transformer将占据主导地位，而且这种两阶段预训练方法也会主导各种应用。

4. 代码实现

[bert中文分类实践](#)

5. 参考文献

- [【NLP】Google BERT详解](#)
- [从Word Embedding到Bert模型—自然语言处理中的预训练技术发展史](#)
- [一文读懂BERT\(原理篇\)](#)

作者:[@mantchs](#)

GitHub:<https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论！共同完善此项目！群号：【541954936】  [加入QQ群](#)

目录

- [1. 什么是XLNet](#)
- [2. 自回归语言模型 \(Autoregressive LM\)](#)
- [3. 自编码语言模型 \(Autoencoder LM\)](#)
- [4. XLNet模型](#)
 - [4.1 排列语言建模 \(Permutation Language Modeling\)](#)
 - [4.2 Transformer XL](#)
- [5. XLNet与BERT比较](#)
- [6. 代码实现](#)
- [7. 参考文献](#)

1. 什么是XLNet

XLNet 是一个类似 BERT 的模型，而不是完全不同的模型。总之，**XLNet是一种通用的自回归预训练方法**。它是CMU和Google Brain团队在2019年6月份发布的模型，最终，XLNet 在 20 个任务上超过了 BERT 的表现，并在 18 个任务上取得了当前最佳效果 (state-of-the-art)，包括机器问答、自然语言推断、情感分析和文档排序。

作者表示，BERT 这样基于去噪自编码器的预训练模型可以很好地建模双向语境信息，性能优于基于自回归语言模型的预训练方法。然而，由于需要 mask 一部分输入，BERT 忽略了被 mask 位置之间的依赖关系，因此出现预训练和微调效果的差异 (pretrain-finetune discrepancy)。

基于这些优缺点，该研究提出了一种泛化的自回归预训练模型 XLNet。XLNet 可以：

1. 通过最大化所有可能的因式分解顺序的对数似然，学习双向语境信息；
2. 用自回归本身的特点克服 BERT 的缺点；
3. 此外，XLNet 还融合了当前最优自回归模型 Transformer-XL 的思路。

2. 自回归语言模型 (Autoregressive LM)

在ELMO / BERT出来之前，大家通常讲的语言模型其实是根据上文内容预测下一个可能跟随的单词，就是常说的自左向右的语言模型任务，或者反过来也行，就是根据下文预测前面的单词，这种类型的LM被称为自回归语言模型。GPT 就是典型的自回归语言模型。ELMO尽管看上去利用了上文，也利用了下文，但是本质上仍然是自回归LM，这个跟模型具体怎么实现有关系。ELMO是做了两个方向（从左到右以及从右到左两个方向的语言模型），但是是分别有两个方向的自回归LM，然后把LSTM的两个方向的隐节点状态拼接到一起，来体现双向语言模型这个事情的。所以其实是两个自回归语言模型的拼接，本质上仍然是自回归语言模型。

自回归语言模型有优点有缺点：

缺点是只能利用上文或者下文的信息，不能同时利用上文和下文的信息，当然，貌似ELMO这种双向都做，然后拼接看上去能够解决这个问题，因为融合模式过于简单，所以效果其实并不是太好。

优点其实跟下游NLP任务有关，比如生成类NLP任务，比如文本摘要，机器翻译等，在实际生成内容的时候，就是从左向右的，自回归语言模型天然匹配这个过程。而Bert这种DAE模式，在生成类NLP任务中，就面临训练过程和应用过程不一致的问题，导致生成类的NLP任务到目前为止都做不太好。

3. 自编码语言模型（Autoencoder LM）

自回归语言模型只能根据上文预测下一个单词，或者反过来，只能根据下文预测前面一个单词。相比而言，Bert通过在输入X中随机Mask掉一部分单词，然后预训练过程的主要任务之一是根据上下文单词来预测这些被Mask掉的单词，如果你对Denoising Autoencoder比较熟悉的话，会看出，这确实是典型的DAE的思路。那些被Mask掉的单词就是在输入侧加入的所谓噪音。类似Bert这种预训练模式，被称为DAE LM。

这种DAE LM的优缺点正好和自回归LM反过来，它能比较自然地融入双向语言模型，同时看到被预测单词的上文和下文，这是好处。缺点是啥呢？主要在输入侧引入[Mask]标记，导致预训练阶段和Fine-tuning阶段不一致的问题，因为Fine-tuning阶段是看不到[Mask]标记的。DAE吗，就要引入噪音，[Mask]标记就是引入噪音的手段，这个正常。

XLNet的出发点就是：能否融合自回归LM和DAE LM两者的优势。就是说如果站在自回归LM的角度，如何引入和双向语言模型等价的效果；如果站在DAE LM的角度看，它本身是融入双向语言模型的，如何抛掉表面的那个[Mask]标记，让预训练和Fine-tuning保持一致。当然，XLNet还讲到了一个Bert被Mask单词之间相互独立的问题。

4. XLNet模型

4.1 排列语言建模（Permutation Language Modeling）

Bert的自编码语言模型也有对应的缺点，就是XLNet在文中指出的：

1. 第一个预训练阶段因为采取引入[Mask]标记来Mask掉部分单词的训练模式，而Fine-tuning阶段是看不到这种被强行加入的Mask标记的，所以两个阶段存在使用模式不一致的情形，这可能会带来一定的性能损失；
2. 另外一个是，Bert在第一个预训练阶段，假设句子中多个单词被Mask掉，这些被Mask掉的单词之间没有任何关系，是条件独立的，而有时候这些单词之间是有关系的。

上面两点是XLNet在第一个预训练阶段，相对Bert来说要解决的两个问题。

其实思路也比较简洁，可以这么思考：XLNet仍然遵循两阶段的过程，第一个阶段是语言模型预训练阶段；第二阶段是任务数据Fine-tuning阶段。它主要希望改动第一个阶段，就是说不像Bert那种带Mask符号的Denoising-autoencoder的模式，而是采用自回归LM的模式。就是说，看上去输入句子X仍然是自左向右的输入，看到 T_i 单词的上文Context_before，来预测 T_i 这个单词。但是又希望在Context_before里，不仅仅看到上文单词，也能看到 T_i 单词后面的下文Context_after里的下文单词，这样的话，Bert里面预训练阶段引入的Mask符号就不需要了，于是在预训练阶段，看上去是个标准的从左向右过程，Fine-tuning当然也是这个过程，于是两个环节就统一起来。当然，这是目标。剩下是怎么做到这一点的问题。

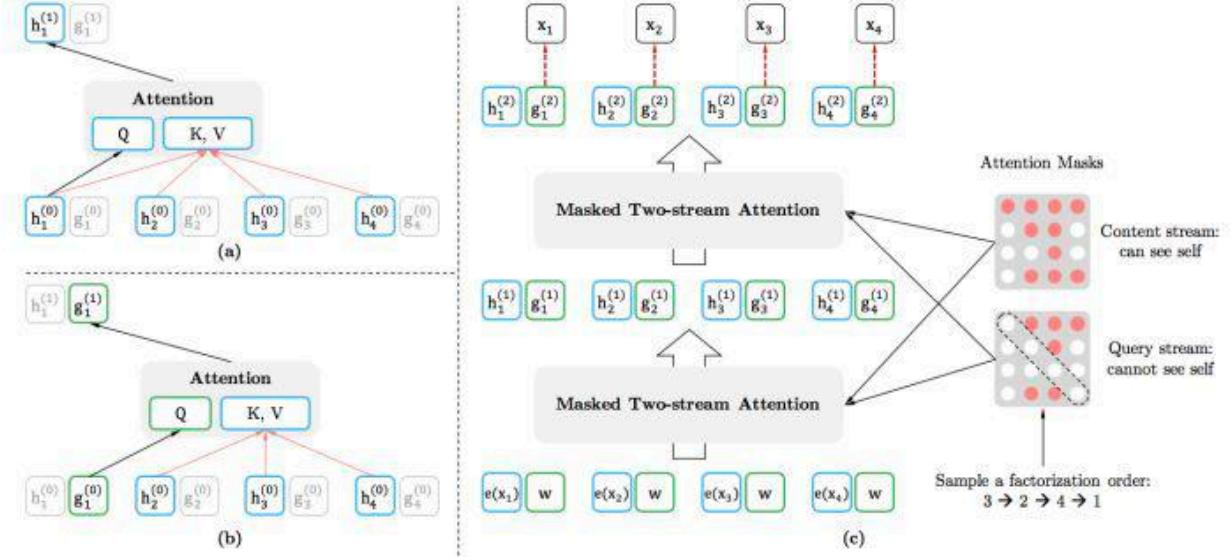


Figure 2: (a): Content stream attention, which is the same as the standard self-attention. (b): Query stream attention, which does not have access information about the content x_{z_t} . (c): Overview of the permutation language modeling training with two-stream attention.

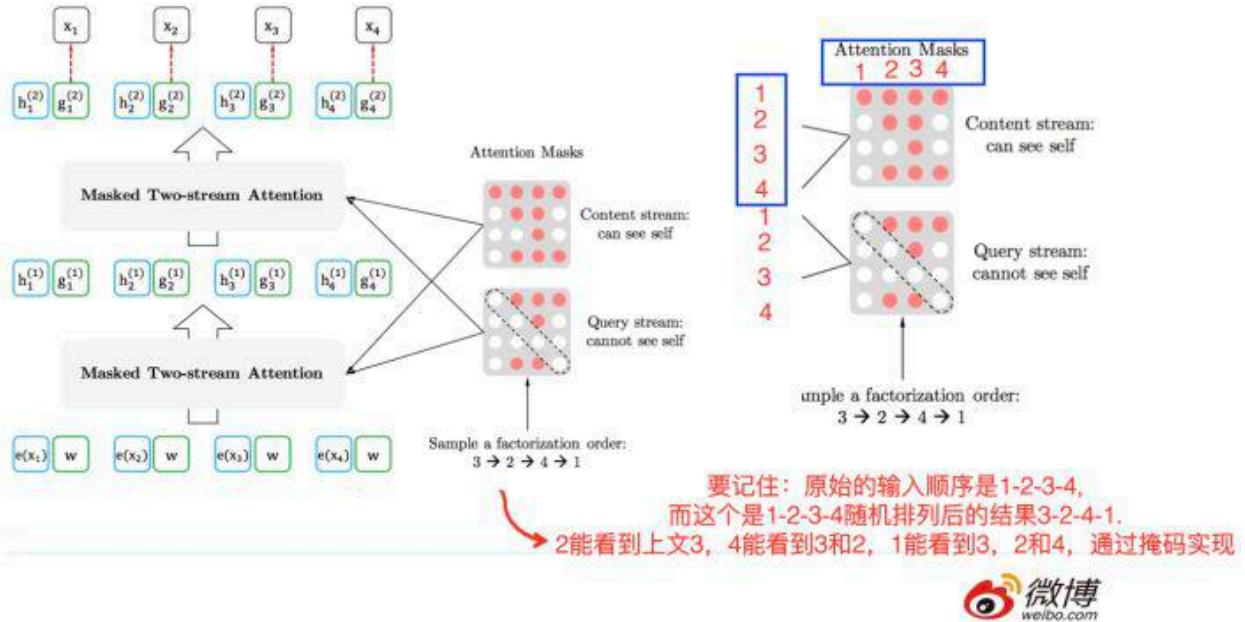
首先，需要强调一点，尽管上面讲的是把句子X的单词排列组合后，再随机抽取例子作为输入，但是，实际上你是不能这么做的，因为Fine-tuning阶段你不可能也去排列组合原始输入。所以，就必须让预训练阶段的输入部分，看上去仍然是 x_1, x_2, x_3, x_4 这个输入顺序，但是可以在Transformer部分做些工作，来达成我们希望的目标。

具体而言，XLNet采取了Attention掩码的机制，你可以理解为，当前的输入句子是X，要预测的单词 T_i 是第*i*个单词，前面1到*i*-1个单词，在输入部分观察，并没发生变化，该是谁还是谁。但是在Transformer内部，通过Attention掩码，从X的输入单词里面，也就是 T_i 的上文和下文单词中，随机选择*i*-1个，放到 T_i 的上文位置中，把其它单词的输入通过Attention掩码隐藏掉，于是就能够达成我们期望的目标（当然这个所谓放到 T_i 的上文位置，只是一种形象的说法，其实在内部，就是通过Attention Mask，把其它没有被选到的单词Mask掉，不让它们在预测单词 T_i 的时候发生作用，如此而已。看着就类似于把这些被选中的单词放到了上文Context_before的位置了）。

具体实现的时候，XLNet是用“双流自注意力模型”实现的，细节可以参考论文，但是基本思想就如上所述，双流自注意力机制只是实现这个思想的具体方式，理论上，你可以想出其它具体实现方式来实现这个基本思想，也能达成让 T_i 看到下文单词的目标。

这里简单说下“双流自注意力机制”，一个是内容流自注意力，其实就是标准的Transformer的计算过程；主要是引入了Query流自注意力，这个是干嘛的呢？其实这就是用来代替Bert的那个[Mask]标记的，因为XLNet希望抛掉[Mask]标记符号，但是比如知道上文单词 x_1, x_2 ，要预测单词 x_3 ，此时在 x_3 对应位置的Transformer最高层去预测这个单词，但是输入侧不能看到要预测的单词 x_3 ，Bert其实是直接引入[Mask]标记来覆盖掉单词 x_3 的内容的，等于说[Mask]是个通用的占位符号。而XLNet因为要抛掉[Mask]标记，但是又不能看到 x_3 的输入，于是Query流，就直接忽略掉 x_3 输入了，只保留这个位置信息，用参数w来代表位置的embedding编码。其实XLNet只是扔了表面的[Mask]占位符号，内部还是引入Query流来忽略掉被Mask的这个单词。和Bert比，只是实现方式不同而已。

双流注意力及Attention Mask机制



上面讲的Permutation Language Model是XLNet的主要理论创新，所以介绍的比较多，从模型角度讲，这个创新还是挺有意思的，因为它开启了自回归语言模型如何引入下文的一个思路，相信对于后续工作会有启发。当然，XLNet不仅仅做了这些，它还引入了其它的因素，也算是一个当前有效技术的集成体。感觉XLNet就是Bert、GPT 2.0和Transformer XL的综合体变身：

1. 首先，它通过PLM(Permutation Language Model)预训练目标，吸收了Bert的双向语言模型；
2. 然后，GPT2.0的核心其实是更多更高质量的预训练数据，这个明显也被XLNet吸收进来了；
3. 再然后，Transformer XL的主要思想也被吸收进来，它的主要目标是解决Transformer对于长文档NLP应用不够友好的问题。

4.2 Transformer XL

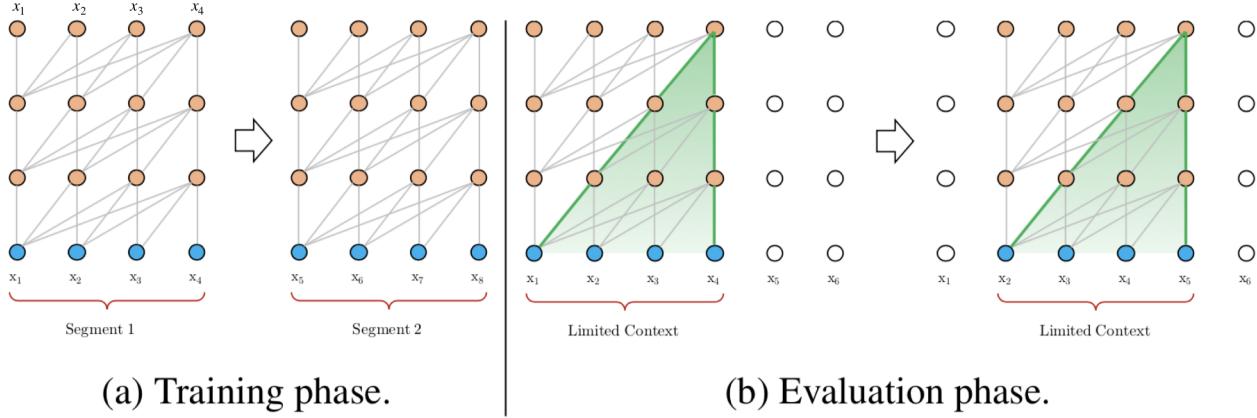
目前在NLP领域中，处理语言建模问题有两种最先进的架构：RNN和Transformer。RNN按照序列顺序逐个学习输入的单词或字符之间的关系，而Transformer则接收一整段序列，然后使用self-attention机制来学习它们之间的依赖关系。这两种架构目前来看都取得了令人瞩目的成就，但它们都局限在捕捉长期依赖性上。

为了解决这一问题，CMU联合Google Brain在2019年1月推出的一篇新论文《Transformer-XL: Attentive Language Models beyond a Fixed-Length Context》同时结合了RNN序列建模和Transformer自注意力机制的优点，在输入数据的每个段上使用Transformer的注意力模块，并使用循环机制来学习连续段之间的依赖关系。

4.2.1 vanilla Transformer

为何要提这个模型？因为Transformer-XL是基于这个模型进行的改进。

AI-Rfou等人基于Transformer提出了一种训练语言模型的方法，来根据之前的字符预测片段中的下一个字符。例如，它使用 x_1, x_2, \dots, x_{n-1} 预测字符 x_n ，而在 x_n 之后的序列则被mask掉。论文中使用64层模型，并仅限于处理512个字符这种相对较短的输入，因此它将输入分成段，并分别从每个段中进行学习，如下图所示。在测试阶段如需处理较长的输入，该模型会在每一步中将输入向右移动一个字符，以此实现对单个字符的预测。



该模型在常用的数据集如enwik8和text8上的表现比RNN模型要好，但它仍有以下缺点：

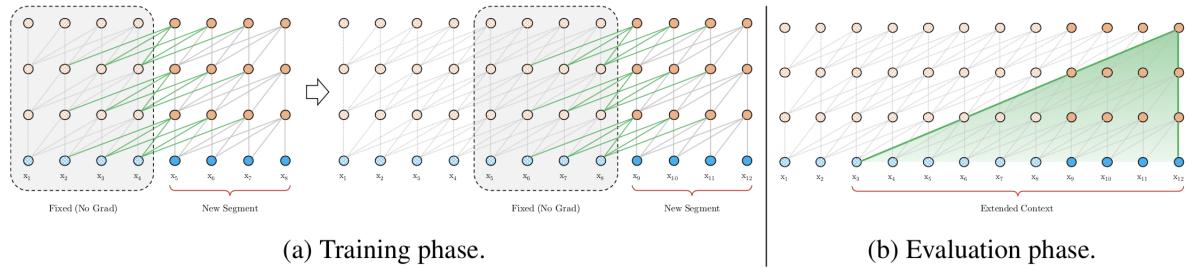
- **上下文长度受限：**字符之间的最大依赖距离受输入长度的限制，模型看不到出现在几个句子之前的单词。
- **上下文碎片：**对于长度超过512个字符的文本，都是从头开始单独训练的。段与段之间没有上下文依赖性，会让训练效率低下，也会影响模型的性能。
- **推理速度慢：**在测试阶段，每次预测下一个单词，都需要重新构建一遍上下文，并从头开始计算，这样的计算速度非常慢。

4.2.2 Transformer XL

Transformer-XL架构在vanilla Transformer的基础上引入了两点创新：循环机制（Recurrence Mechanism）和相对位置编码（Relative Positional Encoding），以克服vanilla Transformer的缺点。与vanilla Transformer相比，Transformer-XL的另一个优势是它可以被用于单词级和字符级的语言建模。

1. 引入循环机制

与vanilla Transformer的基本思路一样，Transformer-XL仍然是使用分段的方式进行建模，但其与vanilla Transformer的本质不同是在于引入了段与段之间的循环机制，使得当前段在建模的时候能够利用之前段的信息来实现长期依赖性。如下图所示：



在训练阶段，处理后面的段时，每个隐藏层都会接收两个输入：

- 该段的前面隐藏层的输出，与vanilla Transformer相同（上图的灰色线）。

- 前面段的隐藏层的输出（上图的绿色线），可以使模型创建长期依赖关系。

这两个输入会被拼接，然后用于计算当前段的Key和Value矩阵。

该方法可以利用前面更多段的信息，测试阶段也可以获得更长的依赖。在测试阶段，与vanilla Transformer相比，其速度也会更快。在vanilla Transformer中，一次只能前进一个step，并且需要重新构建段，并全部从头开始计算；而在Transformer-XL中，每次可以前进一整个段，并利用之前段的数据来预测当前段的输出。

2. 相对位置编码

在Transformer中，一个重要的地方在于其考虑了序列的位置信息。在分段的情况下，如果仅仅对于每个段仍直接使用Transformer中的位置编码，即每个不同段在同一个位置上的表示使用相同的位置编码，就会出现问题。比如，第 $i-2$ 段和第 $i-1$ 段的第一个位置将具有相同的位置编码，但它们对于第 i 段的建模重要性显然并不相同（例如第 $i-2$ 段中的第一个位置重要性可能要低一些）。因此，需要对这种位置进行区分。

论文对于这个问题，提出了一种新的位置编码的方式，即会根据词之间的相对距离而非像Transformer中的绝对位置进行编码。从另一个角度来解读公式的话，可以将attention的计算分为如下四个部分：

- 基于内容的“寻址”，即没有添加原始位置编码的原始分数。
- 基于内容的位置偏置，即相对于当前内容的位置偏差。
- 全局的内容偏置，用于衡量key的重要性。
- 全局的位置偏置，根据query和key之间的距离调整重要性。

详细公式见：[Transformer-XL解读（论文 + PyTorch源码）](#)

5. XLNet与BERT比较

尽管看上去，XLNet在预训练机制引入的Permutation Language Model这种新的预训练目标，和Bert采用Mask标记这种方式，有很大不同。其实你深入思考一下，你会发现，两者本质是类似的。

区别主要在于：

- Bert是直接在输入端显示地通过引入Mask标记，在输入侧隐藏掉一部分单词，让这些单词在预测的时候不发挥作用，要求利用上下文中其它单词去预测某个被Mask掉的单词；
- 而XLNet则抛弃掉输入侧的Mask标记，通过Attention Mask机制，在Transformer内部随机Mask掉一部分单词（这个被Mask掉的单词比例跟当前单词在句子中的位置有关系，位置越靠前，被Mask掉的比例越高，位置越靠后，被Mask掉的比例越低），让这些被Mask掉的单词在预测某个单词的时候不发生作用。

所以，本质上两者并没什么太大的不同，只是Mask的位置，Bert更表面化一些，XLNet则把这个过程隐藏在了Transformer内部而已。这样，就可以抛掉表面的[Mask]标记，解决它所说的预训练里带有[Mask]标记导致的和Fine-tuning过程不一致的问题。至于说XLNet说的，Bert里面被Mask掉单词的相互独立问题，也就是说，在预测某个被Mask单词的时候，其它被Mask单词不起作用，这个问题，你深入思考一下，其实是不重要的，因为XLNet在内部Attention Mask的时候，也会Mask掉一定比例的上下文单词，只要有一部分被Mask掉的单词，其实就面临这个问题。而如果训练数据足够大，其实不靠当前这个例子，靠其它例子，也能弥补被Mask单词直接的相互关系问题，因为总有其它例子能够学会这些单词的相互依赖关系。

当然，XLNet这种改造，维持了表面看上去的自回归语言模型的从左向右的模式，这个Bert做不到，这个有明显的好处，就是对于生成类的任务，能够在维持表面从左向右的生成过程前提下，模型里隐含了上下文的信息。所以看上去，XLNet貌似应该对于生成类型的NLP任务，会比Bert有明显优势。另外，因为XLNet还引入了Transformer XL的机制，所以对于长文档输入类型的NLP任务，也会比Bert有明显优势。

6. 代码实现

[中文XLNet预训练模型](#)

7. 参考文献

- [XLNet原理解读](#)
- [XLNet:运行机制及和Bert的异同比较](#)
- [Transformer-XL解读_\(论文+PyTorch源码\)](#)

作者:[@mantchs](#)

GitHub:<https://github.com/NLP-LOVE/ML-NLP>

欢迎大家加入讨论！共同完善此项目！群号:【541954936】[!\[\]\(56d456ce23fbc6d691f6e596f1c162cb_img.jpg\) 加入QQ群](#)