深度学习 word2vec 笔记之基础篇

by 北流浪子(2014-05-25)

博客地址：http://blog.csdn.net/mytestmy/article/details/26969149

基础篇：http://blog.csdn.net/mytestmy/article/details/26961315

一．前言

伴随着深度学习的大红大紫，只要是在自己的成果里打上 deep learning 字样，总会有人去看。深度学习可以称为当今机器学习领域的当之无愧的巨星，也特别得到工业界的青睐。

在各种大举深度学习大旗的公司中，Google 公司无疑是旗举得最高的，口号喊得最响亮的那一个。Google 正好也是互联网界璀璨巨星，与深度学习的联姻，就像影视巨星刘德华和林志玲的结合那么光彩夺目。

巨星联姻产生的成果自然是天生的宠儿。2013 年末，Google 发布的 word2vec 工具引起了一帮人的热捧，互联网界大量 google 公司的粉丝们兴奋了，从而 google 公司的股票开始大涨，如今直逼苹果公司。

在大量赞叹 word2vec 的微博或者短文中，几乎都认为它是深度学习在自然语言领域的一项了不起的应用，各种欢呼“深度学习在自然语言领域开始发力了”。

互联网界很多公司也开始跟进，使用 word2vec 产出了不少成果。身为一个互联网民工，有必要对这种炙手可热的技术进行一定程度的理解。

好在 word2vec 也算是比较简单的，只是一个简单三层神经网络。在浏览了多位大牛的博客，随笔和笔记后，整理成自己的博文，或者说抄出来自己的博文。

二．背景知识

2.1 词向量

自然语言处理（NLP）相关任务中，要将自然语言交给机器学习中的算法来处理，通常需要首先将语言数学化，因为机器不是人，机器只认数学符号。向量是人把自然界的东西抽象出来交给机器处理的东西，基本上可以说向量是人对机器输入的主要方式了。

词向量就是用来将语言中的词进行数学化的一种方式，顾名思义，词向量就是把一个词表示成一个向量。

主要有两种表示方式，下面分别介绍，主要参考了@皮果提在知乎上的问答，也就是参考文献【2】。

2.1.1 One-Hot Representation

一种最简单的词向量方式是 one-hot representation，就是用一个很长的向量

来表示一个词，向量的长度为词典的大小，向量的分量只有一个 1，其他全为 0，1 的位置对应该词在词典中的位置。举个例子，

“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 ...]

每个词都是茫茫 0 海中的一个 1。

这种 One-hot Representation 如果采用稀疏方式存储，会是非常的简洁：也就是给每个词分配一个数字 ID。比如刚才的例子中，话筒记为 3，麦克记为 8（假设从 0 开始记）。如果要编程实现的话，用 Hash 表给每个词分配一个编号就可以了。这么简洁的表示方法配合上最大熵、SVM、CRF 等等算法已经很好地完成了 NLP 领域的各种主流任务。

但这种词表示有两个缺点：（1）容易受维数灾难的困扰，尤其是将其用于 Deep Learning 的一些算法时；（2）不能很好地刻画词与词之间的相似性（术语好像叫做“词汇鸿沟”）：任意两个词之间都是孤立的。光从这两个向量中看不出两个词是否有关系，哪怕是话筒和麦克这样的同义词也不能幸免于难。

所以会寻求发展，用另外的方式表示，就是下面这种。

2.1.2 Distributed Representation

另一种就是 Distributed Representation 这种表示，它最早是 Hinton 于 1986 年提出的，可以克服 one-hot representation 的缺点。其基本想法是直接用一个普通的向量表示一个词，这种向量一般长成这个样子：[0.792, −0.177, −0.107, 0.109, −0.542, ...]，也就是普通的向量表示形式。维度以 50 维和 100 维比较常见。

当然一个词怎么表示成这么样的一个向量是要经过一番训练的，训练方法较多，word2vec 是其中一种，在后面会提到，这里先说它的意义。还要注意的是每个词在不同的语料库和不同的训练方法下，得到的词向量可能是不一样的。

词向量一般维数不高，很少有人闲着没事训练的时候定义一个 10000 维以上的维数，所以用起来维数灾难的机会现对于 one-hot representation 表示就大大减少了。

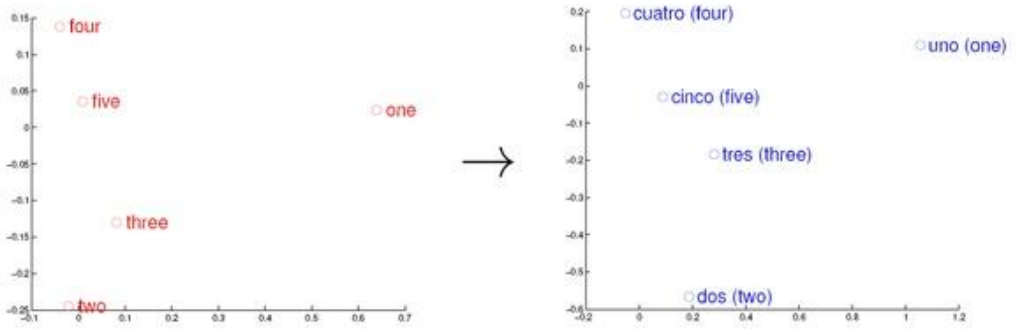
由于是用向量表示，而且用较好的训练算法得到的词向量的向量一般是有空间上的意义的，也就是说，将所有这些向量放在一起形成一个词向量空间，而每一向量则为该空间中的一个点，在这个空间上的词向量之间的距离度量也可以表示对应的两个词之间的“距离”。所谓两个词之间的“距离”，就是这两个词之间的语法，语义之间的相似性。

一个比较爽的应用方法是，得到词向量后，假如对于某个词 A，想找出这个词最相似的词，这个场景对人来说都不轻松，毕竟比较主观，但是对于建立好词向量后的情况，对计算机来说，只要拿这个词的词向量跟其他词的词向量一一计算欧式距离或者 cos 距离，得到距离最小的那个词，就是它最相似的。

这样的特性使得词向量很有意义，自然就会吸引比较多的人去研究，前有 Bengio 发表在 JMLR 上的论文《A Neural Probabilistic Language Model》，又有 Hinton 的层次化 Log-Bilinear 模型，还有 google 的 Tomas Mikolov 团队搞的 word2vec，等等。

词向量在机器翻译领域的一个应用，就是 google 的 Tomas Mikolov 团队开发了一种词典和术语表的自动生成技术，该技术通过向量空间，把一种语言转变成另一种语言，实验中对英语和西班牙语间的翻译准确率高达 90%。

介绍算法工作原理的时候举了一个例子：考虑英语和西班牙语两种语言，通过训练分别得到它们对应的词向量空间 E 和 S。从英语中取出五个词 one，two，three，four，five，设其在 E 中对应的词向量分别为 v1，v2，v3，v4，v5，为方便作图，利用主成分分析（PCA）降维，得到相应的二维向量 u1，u2，u3，u4，u5，在二维平面上将这五个点描出来，如下图左图所示。类似地，在西班牙语中取出（与 one，two，three，four，five 对应的） uno，dos，tres，cuatro，cinco，设其在 S 中对应的词向量分别为 s1，s2，s3，s4，s5，用 PCA 降维后的二维向量分别为 t1，t2，t3，t4，t5，将它们在二维平面上描出来（可能还需作适当的旋转），如下图右图所示：



观察左、右两幅图，容易发现：五个词在两个向量空间中的相对位置差不多，这说明两种不同语言对应向量空间的结构之间具有相似性，从而进一步说明了在词向量空间中利用距离刻画词之间相似性的合理性。

2.2 语言模型

2.2.1 基本概念

语言模型其实就是看一句话是不是正常人说出来的。这玩意很有用，比如机器翻译、语音识别得到若干候选之后，可以利用语言模型挑一个尽量靠谱的结果。在 NLP 的其它任务里也都能用到。

语言模型形式化的描述就是给定一个 T 个词的字符串 s，看它是自然语言的概率 P(w1,w2,…,wt)。w1 到 wT 依次表示这句话中的各个词。有个很简单的推论是：

$$p(s) = p(w_1, w_2, \cdots w_T) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)\cdots p(w_t|w_1, w_2, \cdots w_{T-1}) \quad (1)$$

上面那个概率表示的意义是：第一个词确定后，看后面的词在前面的词出现的情况下出现的概率。如一句话“大家喜欢吃苹果”，总共四个词“大家”，“喜欢”，“吃”，“苹果”，怎么分词现在不讨论，总之词已经分好，就这四个。那么这句话是一个自然语言的概率是：

P(大家，喜欢，吃，苹果)=p(大家)p(喜欢|大家)p(吃|大家,喜欢)p(苹果|大家,喜欢,吃)

p(大家)表示“大家”这个词在语料库里面出现的概率；

p(喜欢|大家)表示“喜欢”这个词出现在“大家”后面的概率;

p(吃|大家，喜欢)表示“吃”这个词出现在“大家喜欢”后面的概率；

p(苹果|大家,喜欢,吃)表示“苹果”这个词出现在“大家喜欢吃”后面的概率。

把这些概率连乘起来，得到的就是这句话平时出现的概率。

如果这个概率特别低，说明这句话不常出现，那么就不算是一句自然语言，因为在语料库里面很少出现。如果出现的概率高，就说明是一句自然语言。

看到了上面的计算，看有多麻烦：只有四个词的一句话，需要计算的是 p(大家)，p(喜欢|大家)，p(吃|大家,喜欢)，p(苹果|大家,喜欢,吃)这四个概率，这四个概率还要预先计算好，考虑词的数量，成千上万个，再考虑组合数，p(吃|大家,喜欢)这个有“大家”、“喜欢”和“吃”的组合，总共会上亿种情况吧；再考虑 p(苹果|大家,喜欢,吃)这个概率，总共也会超过万亿种。

从上面的情况看来，计算起来是非常麻烦的，一般都用偷懒的方式。

为了表示简单，上面的公式（1）用下面的方式表示

$$p(s) = p(w_1, w_2, \cdots w_T) = \prod_{i=1}^{T} p(w_i|Context_i)$$

其中，如果 Contexti 是空的话，就是它自己 p(w)，另外如“吃”的 Context 就是“大家”、“喜欢”，其余的对号入座。

符号搞清楚了，就看怎么偷懒了。

2.2.2 N-gram 模型

接下来说怎么计算$p(w_i|Context_i)$，上面看的是跟据这句话前面的所有词来计算，那么$p(w_i|Context_i)$就得计算很多了，比如就得把语料库里面 p(苹果|大家,喜欢,吃)这种情况全部统计一遍，那么为了计算这句话的概率，就上面那个例子，都得扫描四次语料库。这样一句话有多少个词就得扫描多少趟，语料库一般都比较大，越大的语料库越能提供准确的判断。这样的计算速度在真正使用的时候是万万不可接受的，线上扫描一篇文章是不是一推乱七八糟的没有序列的文字都得扫描很久，这样的应用根本没人考虑。

最好的办法就是直接把所有的$p(w_i|Context_i)$提前算好了，那么根据排列组上面的来算，对于一个只有四个词的语料库，总共就有 4!+3!+2!+1!个情况要计算，那就是 24 个情况要计算；换成 1000 个词的语料库，就是$\sum_{i=1}^{1000} i!$个情况需要统计，对于计算机来说，计算这些东西简直是开玩笑。

这就诞生了很多偷懒的方法，N-gram 模型是其中之一了。N-gram 什么情况呢？上面的 context 都是这句话中这个词前面的所有词作为条件的概率，N-gram 就是只管这个词前面的 n-1 个词，加上它自己，总共 n 个词，计算$p(w_i|Context_i)$只考虑用这 n 个词来算，换成数学的公式来表示，就是

$$\mathrm{p}(w_i|Context_i) = \mathrm{p}(w_i|w_{i-n+1}, w_{i-n+2}, \cdots, w_{i-1})$$

这里如果 n 取得比较小的话，就比较省事了，当然也要看到 n 取得太小，会特别影响效果的，有可能计算出来的那个概率很不准。怎么平衡这个效果和计算就是大牛们的事情了，据大牛们的核算，n 取 2 效果都还凑合，n 取 3 就相当不错了，n 取 4 就顶不住了。看下面的一些数据，假设词表中词的个数 |V| = 20,000 词，那么有下面的一些数据。

n	所有可能的**n-gram**的个数
2 (bigrams)	**400,000,000**
3 (trigrams)	**8,000,000,000,000**
4 (4-grams)	**1.6 x 10^{17}**

照图中的数据看去，取 n=3 是目前计算能力的上限了。在实践中用的最多的就是 bigram 和 trigram 了，而且效果也基本够了。

N-gram 模型也会有写问题，总结如下：

1、n 不能取太大，取大了语料库经常不足，所以基本是用降级的方法

2、无法建模出词之间的相似度，就是有两个词经常出现在同一个 context 后面，但是模型是没法体现这个相似性的。

3、有些 n 元组（n 个词的组合，跟顺序有关的）在语料库里面没有出现过，对应出来的条件概率就是 0，这样一整句话的概率都是 0 了，这是不对的，解决的方法主要是两种：平滑法（基本上是分子分母都加一个数）和回退法（利用 n-1 的元组的概率去代替 n 元组的概率）

2.2.3N-pos 模型

当然学术是无止境的，有些大牛觉得这还不行，因为第 i 个词很多情况下是条件依赖于它前面的词的语法功能的，所以又弄出来一个 n-pos 模型，n-pos 模型也是用来计算$p(w_i|Context_i)$的，但是有所改变，先对词按照词性(Part-of-Speech，POS)进行了分类，具体的数学表达是

$$\mathrm{p}(w_i|Context_i) = \mathrm{p}(w_i|\mathrm{c}(w_{i-n+1}), \mathrm{c}(w_{i-n+2}), \cdots, \mathrm{c}(w_{i-1}))$$

其中 c 是类别映射函数，功能是把 V 个词映射到 K 个类别（1=<K<=V）。这样搞的话，原来的 V 个词本来有V^n种 n 元组减少到了$\mathrm{V} \times \mathrm{K}^{n-1}$种。

其他的模型还很多，不一一介绍了。

2.2.4 模型的问题与目标

如果是原始的直接统计语料库的语言模型，那是没有参数的，所有的概率直接统计就得到了。但现实往往会带一些参数，所有语言模型也能使用极大似然作为目标函数来建立模型。下面就讨论这个。

假设语料库是一个由 T 个词组成的词序列 s（这里可以保留疑问的，因为从很多资料看来是不管什么多少篇文档，也不管句子什么的，整个语料库就是一长串词连起来的，或许可以根据情况拆成句子什么的，这里就往简单里说），其中有 V 个词，则可以构建下面的极大似然函数

$$\mathrm{L} = \prod_{i=1}^{T} p(w_i | Context_i)$$

另外，做一下对数似然

$$\mathrm{l} = \log \mathrm{L} = \frac{1}{V} \sum_{i=1}^{T} logp(w_i | Context_i)$$

对数似然还有些人称为交叉熵，这里不纠结也不介绍。

上面的问题跟正常的情况不太符合，来看看下一种表达。假设语料库是有 S 个句子组成的一个句子序列（顺序不重要），同样是有 V 个词，似然函数就会构建成下面的样子

$$\mathrm{L} = \prod_{j}^{\mathrm{S}} \left(\prod_{i_j=1}^{T_j} p\left(w_{i_j} \middle| Context_{i_j} \right) \right)$$

对数似然就会是下面的样子

$$\mathrm{l} = \log \mathrm{L} = \frac{1}{V} \sum_{j=1}^{\mathrm{S}} \left(\sum_{i_j=1}^{T_j} logp\left(w_{i_j} \middle| Context_{i_j} \right) \right)$$

有意向的同学可以扩展到有文档的样子，这里就不介绍了。

为啥要注意这个问题呢？原因有多种，计算$p(w_i | Context_i)$这个东西的参数是主要的原因。

为啥会有参数呢？在计算$p(w_i | Context_i)$这个东西的过程中，有非常多的方法被开发出来了，如上面的平滑法，回退法上面的，但这些都是硬统计一下基本就完了；这就带来一些需要求的参数，如平滑法中使用的分子分母分别加上的常数是什么？

这还不够，假如用的是 trigram，还得存储一个巨大的元组与概率的映射（如

果不存储，就得再进行使用的时候实际统计，那太慢了），存这个东西可需要很大的内存，对计算机是个大难题。

这都难不倒大牛们，他们考虑的工作是利用函数来拟合计算$p(w_i|Context_i)$，换句话说，$p(w_i|Context_i)$不是根据语料库统计出来的，而是直接把 context 和 wi 代到一个函数里面计算出来的，这样在使用的时候就不用去查那个巨大的映射集了（或者取语料库里面统计这个概率）。用数学的方法描述就是

$$p(w_i|Context_i) = \mathrm{f}(w_i, Context_i; \theta)$$

这样的工作也体现了科学家们的价值——这帮人终于有点东西可以忙了。

那么探索这个函数的具体形式就是主要的工作了，也是后面 word2vec 的工作的主要内容。函数的形式实在太多了，线性的还好，非线性真叫一个多，高维非线性的就更多了。

探索一个函数的具体形式的术语叫做拟合。

然后就有人提出了用神经网络来拟合这个函数，就有了各种方法，word2vec 是其中的一种。

致谢

多位 Google 公司的研究员无私公开的资料。

多位博主的博客资料。

参考文献

[1]http://techblog.youdao.com/?p=915Deep Learning 实战之 word2vec，网易有道的 pdf

[2] http://www.zhihu.com/question/21714667/answer/19433618@皮果提在知乎上的问答

[3]http://www.zhihu.com/question/21661274/answer/19331979 @杨超在知乎上的问答《Word2Vec 的一些理解》

[4] 第五章 n-gram 语言模型百度文库上的一个资料

[5] 主题：统计自然语言处理的数学基础百度文库上的一个

深度学习 word2vec 笔记之算法篇

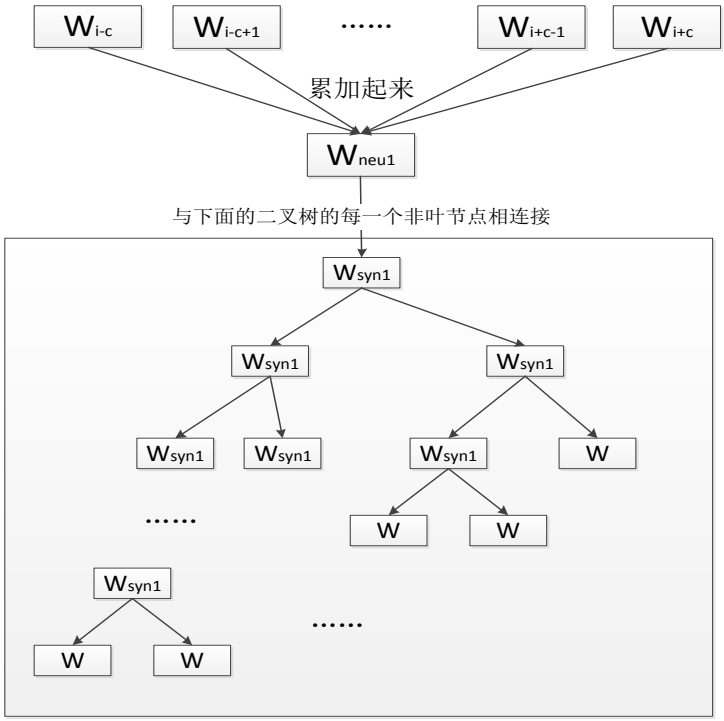
在看 word2vec 的资料的时候，经常会被叫去看那几篇论文，而那几篇论文也没有系统地说明 word2vec 的具体原理和算法，这样看资料就没有得到应有的效果。

为了节省看无用资料的时间，就整理了一个笔记，希望能帮助各位尽快理解 word2vec 的基本原理，避免浪费时间。

当然如果已经了解了，就随便看看得了。

一． CBOW 加层次的网络结构与使用说明

Word2vec 总共有两种类型，每种类型有两个策略，总共 4 种。这里先说最常用的一种。这种的网络结构如下图。



其中第一层，也就是最上面的那一层可以称为输入层。输入的是若干个词的词向量(词向量的意思就是把一个词表示成一个向量的形式表达，后面会介绍)。中间那个层可以成为隐层，是输入的若干个词向量的累加和，注意是向量的累加和，结果是一个向量。

第三层是方框里面的那个二叉树，可以称之为输出层，隐层的那个节点要跟输出层的那个二叉树的所有非叶节点链接的，线太多画不过来了。第三层的这个二叉树是一个霍夫曼树，每个非叶节点也是一个向量，但是这个向量不代表某个词，代表某一类别的词；每个叶子节点代表一个词向量，为了简单只用一个 w 表示，没有下标。另外要注意的是，输入的几个词向量其实跟这个霍夫曼树中的

某几个叶子节点是一样的，当然输入的那几个词跟它们最终输出的到的那个词未必是同一个词，而且基本不会是同一个词，只是这几个词跟输出的那个词往往有语义上的关系。

还有要注意的是，这个霍夫曼树的所有叶子节点就代表了语料库里面的所有词，而且是每个叶子节点对应一个词，不重复。

这个网络结构的功能是为了完成一个的事情——判断一句话是否是自然语言。怎么判断呢？使用的是概率，就是计算一下这句话的“一列词的组合”的概率的连乘（联合概率）是多少，如果比较低，那么就可以认为不是一句自然语言，如果概率高，就是一句正常的话。这个其实也是语言模型的目标。前面说的“一列词的组合”其实包括了一个词跟它的上下文的联合起来的概率，一种普通的情况就是每一个词跟它前面所有的词的组合的概率的连乘，这个后面介绍。

对于上面的那个网络结构来说，网络训练完成后，假如给定一句话 s，这句话由词 w1,w2,w3,…,wT 组成，就可以利用计算这句话是自然语言的概率了，计算的公式是下面的公式

$$\mathrm{p(s)} = \mathrm{p}(w_1, w_2, \cdots w_T) = \prod_{i=1}^{T} p(w_i|Context_i)$$

其中的 Context 表示的是该词的上下文，也就是这个词的前面和后面各若干个词，这个“若干”（后面简称 c）一般是随机的，也就是一般会从 1 到 5 之间的一个随机数；每个$p(w_i|Context_i)$代表的意义是前后的 c 个词分别是那几个的情况下，出现该词的概率。举个例子就是：“大家喜欢吃好吃的苹果”这句话总共 6 个词，假设对“吃”这个词来说 c 随机抽到 2，则“吃”这个词的 context 是“大家”、“喜欢”、“好吃”和“的”，总共四个词，这四个词的顺序可以乱，这是 word2vec 的一个特点。

计算$p(w_i|Context_i)$的时候都要用到上面的那个网络，具体计算的方法用例子说明，假设就是计算“吃”这个词的在“大家”、“喜欢”、“好吃”和“的”这四个词作为上下文的条件概率，又假设“吃”这个词在霍夫曼树中是的最右边那一个叶子节点（在下面的图中的节点 R），那么从根节点到到达它就有两个非叶节点，根节点对应的词向量命名为 A，根节点的右孩子节点对应的词向量命名为 B，另外再假设“大家”、“喜欢”、“好吃”和“的”这四个词的词向量的和为 C，则

$$p\left(\text{吃}\middle|Context_{\text{吃}}\right) = (1 - \sigma(A \cdot C)) \cdot (1 - \sigma(B \cdot C))$$

其中$\sigma(x) = 1/(1 + e^{-x})$，是 sigmoid 公式。

要注意的是，如果“吃”这个词在非叶节点 B 的左孩子节点（假设称为 E）的右边的那个叶子节点（在下面的图中的节点 S），也就是在图中右边的三个叶子的中间那个，则有

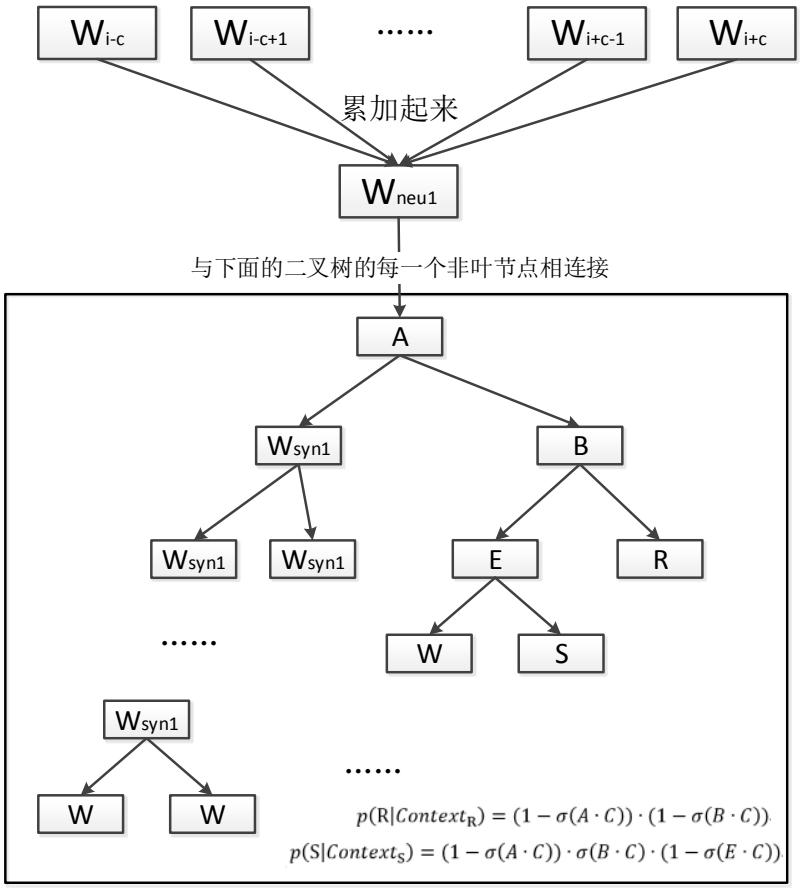
$$p\left(\text{吃}\middle|Context_{\text{吃}}\right) = (1 - \sigma(A \cdot C)) \cdot \sigma(B \cdot C) \cdot (1 - \sigma(E \cdot C))$$

上面的那句话的每个词都计算$p(w_i|Context_i)$后连乘起来得到联合概率，这个概率如果大于某个阈值，就认为是正常的话；否则就认为不是自然语言，要排除掉。

对于这个神经网络的描述索然无味，因为主角也不是这个概率，这个神经网络最重要的是输出层的那个霍夫曼树的叶子节点上的那些向量，那些向量被称为词向量，词向量就是另外一篇博文里面介绍的，是个好东西。

怎么得到这些词向量更加是一个重要的过程，也是 word2vec 这整个算法最重要的东西，后面会认真介绍。

至于霍夫曼树，其实是一个优化的解法，后面再提。



二．优化目标与解问题

2.1 从霍夫曼树到条件概率的计算

前面已经提过语言模型的目标就是判断一句话是否是正常的，至于怎么判断则需要计算很多条件概率如$p(w_i|Context_i)$，然后还要把这些条件概率连乘起来得到联合概率。这样就带来了问题了——怎么去计算$p(w_i|Context_i)$，有很多办法的，后面的章节会介绍。这里的 word2vec 的计算这个条件概率的方法是利用神经网络的能量函数，因为在能量模型中，能量函数的功能是把神经网络的状态转化为概率表示，这在另外一篇博文 RBM 里面有提到，具体要看 hinton 的论文

来了解了。能量模型有个特别大的好处，就是能拟合所有的指数族的分布。那么，如果认为这些条件概率是符合某个指数族的分布的话，是可以用能量模型去拟合的。总之 word2vec 就认为$p(w_i|Context_i)$这个条件概率可以用能量模型来表示了。

既然是能量模型，那么就需要能量函数，word2vec 定义了一个非常简单的能量函数

$$\mathrm{E(A,C)} = -(A \cdot C)$$

其中 A 可以认为是某个词的词向量，C 是这个词的上下文的词向量的和（向量的和），基本上就可以认为 C 代表 Context；中间的点号表示两个向量的内积。

然后根据能量模型（这个模型假设了温度一直是 1，所以能量函数没有分母了），就可以表示出词 A 的在上下文词向量 C 下的概率来了

$$\mathrm{p(A|C)} = \frac{e^{-E(A,C)}}{\sum_{v=1}^{V} e^{-E(w_v,C)}} \qquad (2.1.2)$$

其中 V 表示语料库里面的的词的个数，这个定义的意思是在上下文 C 出现的情况下，中间这个词是 A 的概率，为了计算这个概率，肯定得把语料库里面所有的词的能量都算一次，然后再根据词 A 的能量，那个比值就是出现 A 的概率。这种计算概率的方式倒是能量模型里面特有的，这个定义在论文《Hierarchical Probabilistic Neural Network Language Model》里面，这里拿来改了个形式。

这个概率其实并不好统计，为了算一个词的的概率，得算上这种上下文的情况下所有词的能量，然后还计算指数值再加和。注意那个分母，对语料库里面的每个词，分母都要算上能量函数，而且再加和，假如有 V 个词汇，整个语料库有 W 个词，那么一轮迭代中光计算分母就有 W*V*D 个乘法，如果词向量维度是 D 的话。比如，语料库有 100000000 个词，词汇量是 10000，计算 100 维的词向量，一轮迭代要10^{14}次乘法，计算机计算能力一般是10^{9}每秒，然后一轮迭代就要跑 100000 秒，大约 27 小时，一天多吧。1000 轮迭代就三年了。

这时候科学家们的作用又体现了，假如把语料库的所有词分成两类，分别称为 G 类和 H 类，每类一半，其中词 A 属于 G 类，那么下面的式子就可以成立了

$$\mathrm{p(A|C)} = \mathrm{p(A|G,C)p(G|C)} \qquad (2.1.3)$$

这个式子的的含义算明确的了，词 A 在上下文 C 的条件下出现的概率，与后面的这个概率相等——在上下文 C 的条件下出现了 G 类词，同时在上下文为 C，并且应该出现的词是 G 类词的条件下，词 A 出现的概率。

列出这么一个式子在论文《Hierarchical Probabilistic Neural Network Language Model》里面也有个证明的，看原始的情况

$$\mathrm{P(Y = y|X = x) = P(Y = y|D = d(y),X)P(D = d(y)|X = x)}$$

其中 d 是一个映射函数，把 Y 里面的元素映射到词的类别 D 里面的元素。还有个证明

$$\mathrm{P(Y|X)} = \sum_i P(Y, D = \mathrm{i|X})$$
$$= \sum_i P(Y|D = i, X)P(D = i|X)$$
$$= P(Y|D = d(Y), X)P(D = d(Y)|X)$$

式子（2.1.3）说明了一个问题，计算一个词 A 在上下文 C 的情况下出现的概率，可以先对语料库中的词分成两簇，然后能节省计算。现在来展示一下怎么节省计算，假设 G，H 这两类的簇就用 G 和 H 表示（G 和 H 也是一个词向量，意思就是 G 表示了其中一簇的词，H 表示了另外一簇的词，G 和 H 只是一个代表，也不是什么簇中心的说法。其实如果情况极端点，只有两个词，看下面的式子就完全没问题了。在多个词的情况下，就可以认为词被分成了两团，G 和 H 个表示一团的词，计算概率什么的都以一整团为单位），那么式子(2.3)中的 p(G|C) 可以用下面的式子计算

$$\mathrm{p(G|C)} = \frac{e^{-E(G,C)}}{e^{-E(G,C)} + e^{-E(H,C)}} = \frac{1}{1 + e^{-(-(H-G)\cdot C)}} = \frac{1}{1 + e^{-E(H-G,C)}}$$

也就是说，可以不用关心这两个簇用什么表示，只要利用一个 F=H-G 的类词向量的一个向量就可以计算 P(G|C)了，所以这一步是很节省时间的。再看另外一步

$$\mathrm{p(A|G, C)} = \frac{e^{-E(A,C)}}{\sum_{W \in G} e^{-E(W,C)}}$$

由于在 G 内的词数量只有 V/2 个，也就是说计算分母的时候只要计算 V/2 个词的能量就可以了。这已经省了一半的计算量了，可惜科学家们是贪得无厌的，所以还要继续省，怎么来呢？把 G 类词再分成两个簇 GG，GH，A 在 GH 里面，然后

$$\mathrm{p(A|G, C)} = \mathrm{p(A|GH, G, C)p(GH|G, C)}$$

同样有

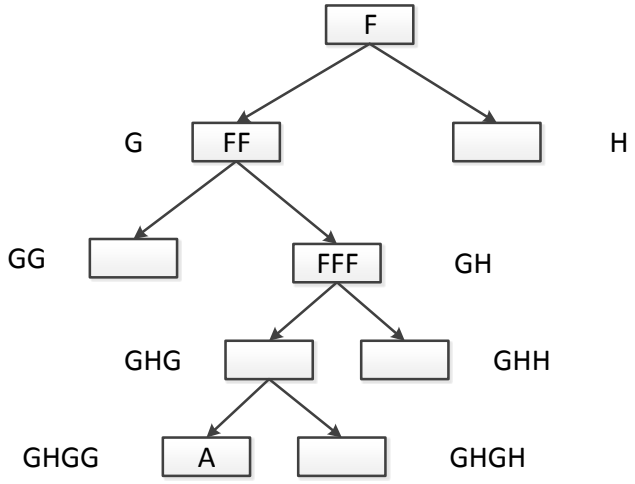
$$\mathrm{p(GH|G, C)} = \frac{1}{1 + e^{-E(GG-GH,C)}}$$

和

$$\mathrm{p(A|GH, G, C)} = \frac{e^{-E(A,C)}}{\sum_{W \in GH} e^{-E(W,C)}}$$

同样可以把 GG-GH 用一个类词向量表达，这时候

$$\mathrm{p(A|C)} = \mathrm{p(A|GH, G, C)p(GH|G, C)p(G|C)}$$



继续下去假设继续分到 GHG 簇的时候只剩两个词了，再分两簇为 GHGG 和 GHGH，其中的簇 GHGG 就只有一个词 A，那么p(A|C)可以用下面的式子算

$$p(A|C) = p(A|GHGG, GHG, GH, G, C)p(GHGG|GHG, GH, G, C)p(GHG|GH, G, C)p(GH|G, C)p(G|C)$$

其中 p(A|GHGG,GHG,GH,G)是 1，因为只有一个单词，代到公式（2.2）就可以得到，那么就有

$$p(A|C) = p(GHGG|GHG, GH, G, C)p(GHG|GH, G, C)p(GH|G, C)p(G|C)$$

也就是

$$p(A|C) = \frac{1}{1+e^{-E(GHH-GHG,C)}} \cdot \frac{1}{1+e^{-E(GG-GH,C)}} \cdot \frac{1}{1+e^{-E(H-G,C)}}$$

假设再令 FFF=GHH-GHG，FF=GG-GH，F=H-G，那么 p(A|C)只要算这三个词与上下文 C 的能量函数了，确实比原来的要节省很多计算的。

对于上面的霍夫曼树来说假设 G 表示向右，H 表示向左，那么 A 就是从右边开始数的第二个叶子节点，就是图中右边的三个 W 的中间那个。那么 F，FF，FFF 就是这个叶子节点路径上的三个非叶节点。

但是一个词总是会一会向左，一会向右的，也就是在根节点那里，一会是 p(G|C)那么 F=H-G，一会又是 p(H|C)那么 F=G-H，如果 F 在每个节点都是唯一一个值，就可以直接用一次词向量表示这个非叶节点了。这下难不倒科学家的，令 F 一直是等于 H-G，那么一直有

$$p(H|C) = \frac{1}{1+e^{-E(F,C)}}$$

并且有 p(G|C)=1-p(H|C)。

这样每个非叶节点就可以用唯一一个词向量表示了。

看到这里，总该明白为啥 p(A|C)要这么算了吧。再换种情况，上面的概率$p\left(\text{吃}\middle|Context_{\text{吃}}\right)$这个概率的计算方法是不是也是同样的道理？

总结下来，$p(w_i|Context_i)$可以用下面的公式计算了

$$\mathrm{p(w|Context)} = \prod_{k=1}^{K} p(d_k|q_k, C) = \prod_{k=1}^{K}((\sigma(q_k \cdot C))^{1-d_k} \cdot (1-\sigma(q_k \cdot C))^{d_k})$$

其中 C 表示上下文的词向量累加后的向量，qk 表示从根节点下来到叶子节点的路径上的那些非叶节点，dk 就是编码了，也可以说是分类，因为在霍夫曼树的每个非叶节点都只有两个孩子节点，那可以认为当 wi 在这个节点的左子树的叶子节点上时 dk=0，否则 dk=1。这样的话每个词都可以用一组霍夫曼编码来表示，就有了上面的那个式子中间的那个 dk，整个p(w|Context)就可以用霍夫曼树上的若干个非叶节点和词 w 的霍夫曼编码来计算了。

看到这务必想明白，因为开始要讨论怎么训练了。

2.1.1 霍夫曼树

上面输出层的二叉树是霍夫曼树，其实并没有要求是霍夫曼树，随便一个不太离谱的二叉树都可以的，但是用霍夫曼树能达到最优的计算效果。

根据之前的讨论，已经知道了语料库里面每个词都要从根节点下来，一直走到叶子节点，每经过一个非叶节点，就要计算一个 sigmoid 函数。

随便乱分也能达到效果，但是信息熵理论给出了最优的方案——霍夫曼树。具体可以查看其它资料。

2.2 目标函数

假设语料库是有 S 个句子组成的一个句子序列（顺序不重要），整个语料库有 V 个词，似然函数就会构建成下面的样子

$$\mathrm{L(\theta)} = \prod_{j}^{S}\left(\prod_{i_j=1}^{T_j} p\left(w_{i_j}\middle|Context_{i_j}\right)\right)$$

(2.2.1)

其中T_j表示第 j 个句子的词个数，极大似然要对整个语料库去做的。对数似然就会是下面的样子

$$\mathrm{l(\theta)} = \mathrm{logL(\theta)} = \frac{1}{V}\sum_{j=1}^{S}\left(\sum_{i_j=1}^{T_j} logp\left(w_{i_j}\middle|Context_{i_j}\right)\right)$$

(2.2.2)

如果前面有个 1/V，对数似然还有些人称为交叉熵，这个具体也不了解，就不介绍了；不用 1/V 的话，就是正常的极大似然的样子。

有意向的同学可以扩展到有文档的样子，这里就不介绍了。

但是对于 word2vec 来说，上面的似然函数得改改，变成下面的样子

$$L(\theta) = \prod_{j}^{S}\left(\prod_{i_j=1}^{T_j} p\left(w_{i_j}\middle|Context_{i_j}\right)\right)$$

$$= \prod_{j}^{S}\left(\prod_{i_j=1}^{T_j}\left(\prod_{k_{ij}=1}^{K_{ij}}\left(\left(\sigma(q_{k_{ij}}\cdot C_{i_j})\right)^{1-d_{k_{ij}}}\cdot\left(1-\sigma(q_{k_{ij}}\cdot C_{i_j})\right)^{d_{k_{ij}}}\right)\right)\right)$$

其中的 Cij 表示上下文相加的那个词向量。对数似然就是下面的

$$l(\theta) = \log L(\theta) = \sum_{j=1}^{S}\left(\sum_{i_j=1}^{T_j}\left(\sum_{k_{ij}=1}^{K_{ij}}\log\left(\left(\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)^{1-d_{k_{ij}}}\cdot\left(1-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)^{d_{k_{ij}}}\right)\right)\right)$$

$$= \sum_{j=1}^{S}\left(\sum_{i_j=1}^{T_j}\left(\sum_{k_{ij}=1}^{K_{ij}}\left[\left(1-d_{k_{ij}}\right)log\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right.\right.\right.$$

$$\left.\left.\left.+\, d_{k_{ij}}log\left(1-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\right]\right)\right)$$

这里就不要 1/V 了。

这个看起来应该比较熟悉了，很像二分类的概率输出的逻辑回归——logistic regression 模型。没错了，word2vec 就是这么考虑的，把在霍夫曼树向左的情况，也就是 dk=0 的情况认为是正类，向右就认为是负类（这里的正负类只表示两种类别之一）。这样每当出现了一个上下文 C 和一个词在左子树的情况，就认为得到了一个正类样本，否则就是一个负类样本，每个样本的属于正类的概率都可以用上面的参数算出来，就是$\sigma(q_{i_{jk}}\cdot Context_{i_j})$，如果是向右的话，就用$1-\sigma(q_{i_{jk}}\cdot Context_{i_j})$计算其概率。注意每个词可以产生多个样本，因为从霍夫曼树的根节点开始，每个叶子节点都产生一个样本，这个样本的 label（也就是属于正类或者负类标志）可以用霍夫曼编码来产生，前面说过了，向左的霍夫曼编码 dk=0，所以很自然地可以用 1-dk 表示每个样本 label。

在这里，霍夫曼编码也变成了一个重要的东西了。

这样就好多了，问题到这也该清楚了，上面那个l(θ)就是对数似然，然后负对数似然$f = -l(\theta)$就是需要最小化的目标函数了。

2.3 解法

解法选用的是 SGD，博文《在线学习算法 FTRL》中说过 SGD 算法的一些情况。具体说来就是对每一个样本都进行迭代，但是每个样本只影响其相关的参数，跟它无关的参数不影响。对于上面来说，第 j 个样本的第 ij 个词的负对数似然是

$$f_{i_j} = p\left(w_{i_j}\middle|C_{i_j}\right) = -\sum_{k_{ij}=1}^{K_{ij}}\left[\left(1-d_{k_{ij}}\right)log\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)+d_{k_{ij}}log\left(1-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\right]$$

第 j 个样本的第 ij 个词的在遇到第 kij 个非叶节点时的负对数似然是

$$f_{k_{ij}} = -\left(1-d_{k_{ij}}\right)log\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)-d_{k_{ij}}log\left(1-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)$$

计算$f_{k_{ij}}$的梯度，注意参数包括$q_{k_{ij}}$和C_{i_j}，其中C_{i_j}的梯度是用来计算w_{i_j}的时候用到。另外需要注意的是logσ(x)的梯度是$1-\sigma(x)$，$\log(1-\sigma(x))$的梯度是$-\sigma(x)$，

$$Fq\left(q_{k_{ij}}\right)=\frac{\partial f_{k_{ij}}}{\partial q_{k_{ij}}} = -\left(1-d_{k_{ij}}\right)\cdot\left(1-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\cdot C_{i_j}-d_{k_{ij}}\cdot\left(-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\cdot C_{i_j} = -\left(1-d_{k_{ij}}-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\cdot C_{i_j}$$

和

$$Fc\left(q_{k_{ij}}\right)=\frac{\partial f_{k_{ij}}}{\partial C_{i_j}} = -\left(1-d_{k_{ij}}\right)\cdot\left(1-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\cdot q_{k_{ij}}-d_{k_{ij}}\cdot\left(-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\cdot q_{k_{ij}} = -\left(1-d_{k_{ij}}-\sigma\left(q_{k_{ij}}\cdot C_{i_j}\right)\right)\cdot q_{k_{ij}}$$

上面的 Fq 和 Fc 只是简写，有了梯度就可以对每个参数进行迭代了

$$q_{k_{ij}}^{n+1} = q_{k_{ij}}^{n} - \eta Fq\left(q_{k_{ij}}^{n}\right)$$

同时，每个词的词向量也可以进行迭代了

$$\mathrm{w}_I^{n+1} = \mathrm{w}_I^{n} - \eta\sum_{k_{ij}=1}^{K_{ij}} Fc\left(q_{k_{ij}}^{n}\right)$$

注意第二个迭代的 wI 是代表所有的输入词的，也就是假如输入了 4 个词，这四个词都要根据这个方式进行迭代（注意是上下文的词才是输入词，才根据梯度更新了，但是 wij 这个词本身不更新的，就是轮到它自己在中间的时候就不更新了）。第二个迭代式确实不好理解，因为这里的意思是所有非叶节点上的对上下文的梯度全部加和就得到了这个词的上下文的梯度，看起来这个就是 BP 神经网络的误差反向传播。

论文《Hierarchical Probabilistic Neural Network Language Model》和《Three New Graphical Models for Statistical Language Modelling》中看起来也是这么样的解释，人家都是 Context 的几个词首尾连接得到的一个向量，对这个长向量有一

个梯度，或者一个超大的 V*m 矩阵（m 是词向量的维度），对这个矩阵每个元素有一个梯度，这些梯度自然也包括了输入词的梯度。

对于这个迭代式，csdn 有一位网名为@LJZLJZ1994 的同学给出了他的解释，说得很好，直接原话引用：就是用对 C 的偏导迭代我觉得可以这么理解，这个关于 C 的偏导既是关于 hidden layer 的偏导（就是 2c 个向量相加平均），同时也是关于 input layer 里各个词向量的偏导（不太准确，还要再乘上一个常数），这是因为 hidden 上的 activation function 是线性的，而且 input 到 hidden 的线性变换的各个权重都是(1/2c)，考虑一下求偏导的链式法则就行了。

非常感谢这位热心的同学，就以此作为该疑问的答案，如有其它解释，请告知，本人再综合。

2.4 代码中的 trick

如前文，文字的 c 表示左右各取多少个词（注意跟代码中的不同），代码中 b 是一个从 0 到 window-1 的一个数，是对每个词都随机生成的，而这个 window 就是用户自己输入的一个变量，默认值是 5（这里得换换概念了，代码中的 c 代表当前处理到了那个词的下标，实际的随机变量是 b）。代码实际实现的时候是换了一种方法首先生成一个 0 到 window-1 的一个数 b，然后训练的词（假设是第 i 个词，代码中的变量是 sentence_position）的窗口是从第 i-(window-b)个词开始到第 i+(window-b)个词结束，中间用 a != window 这个判断跳过了这个词他自己，也就是更新词向量的时候只更新上下文相关的几个输入词，这个词本身是不更新的。要注意的是每个词的 b 都不一样的，都是随机生成的，这就意味着连窗口大小都是随机的。

如果有人看过代码，就会发现，其中的$q_{k_{ij}}$在代码中用矩阵 syn1 表示，C_{i_j}在代码中用 neu1 表示。叶子节点里面的每个词向量在代码中用 syn0 表示，利用下标移动去读取。

核心代码如下，其中的 vocab[word].code[d]就表示$d_{k_{ij}}$，其他就是迭代过程，代码是写得相当简洁啊。

```
413       for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
414         c = sentence_position - window + a;
415         if (c < 0) continue;
416         if (c >= sentence_length) continue;
417         last_word = sen[c];
418         if (last_word == -1) continue;
419         for (c = 0; c < layer1_size; c++) neu1[c] += syn0[c + last_word * layer1_size];
420       }
421       if (hs) for (d = 0; d < vocab[word].codelen; d++) {
422         f = 0;
423         l2 = vocab[word].point[d] * layer1_size;
424         // Propagate hidden -> output
425         for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1[c + l2];
426         if (f <= -MAX_EXP) continue;
427         else if (f >= MAX_EXP) continue;
428         else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
429         // 'g' is the gradient multiplied by the learning rate
430         g = (1 - vocab[word].code[d] - f) * alpha;
431         // Propagate errors output -> hidden
432         for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
433         // Learn weights hidden -> output
434         for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * neu1[c];
435       }
```

代码中的 419 行就是计算 Cij，425-428 行就是计算 f，也就是$\sigma\left(q_{k_{ij}} \cdot C_{i_j}\right)$的值，432 行就是累积 Cij 的误差，434 就是更新$q_{k_{ij}}^{n+1}$。

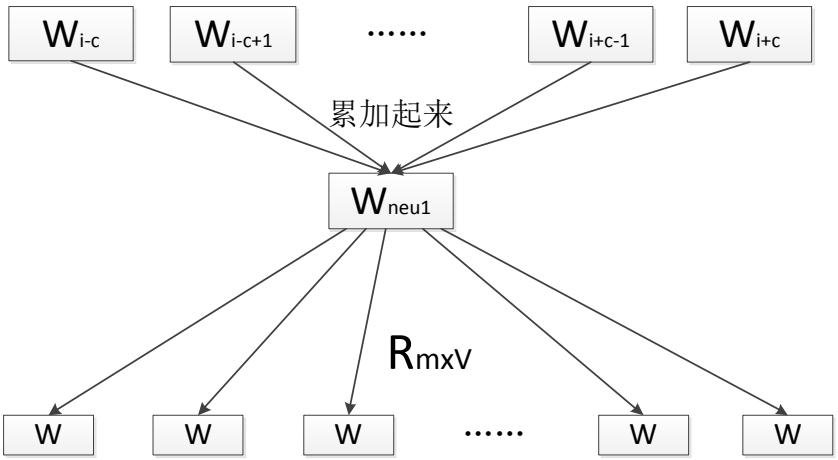
```
457       // hidden -> in
458       for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
459         c = sentence_position - window + a;
460         if (c < 0) continue;
461         if (c >= sentence_length) continue;
462         last_word = sen[c];
463         if (last_word == -1) continue;
464         for (c = 0; c < layer1_size; c++) syn0[c + last_word * layer1_size] += neu1e[c];
465       }
```

注意上面，对每个输入词都进行了更新，更新的幅度就是 432 行中误差累计的结果。

三． CBOW 加抽样的网络结构与使用说明

3.1 网络结构与使用说明

网络结构如下



如（二）中，中间的那个隐层是把上下文累加起来的一个词向量，然后有一个矩阵 R，是在训练过程中用到的临时矩阵，这个矩阵连接隐层与所有输出节点，但是这个矩阵在使用这个网络的时候不怎么用得到，这里也是没弄清楚的一个地方。每个输出节点代表一个词向量。

同样如（二）中的例子，计算$p\left(吃\middle|Context_{吃}\right)$这么一个概率，这里的计算方法就简单多了，就是随机从语料库里面抽取 c 个词，这里假设 c=3，抽中了 D，E，F 这三个词，又假设“吃”这个词的词向量是 A，那么就计算“吃”这个词的概率就用下面的公式

$$p\left(吃\middle|Context_{吃}\right)=\sigma(\mathrm{A}\cdot\mathrm{C})\cdot(1-\sigma(\mathrm{D}\cdot\mathrm{C}))\cdot(1-\sigma(\mathrm{E}\cdot\mathrm{C}))\cdot(1-\sigma(\mathrm{F}\cdot\mathrm{C}))$$

同样如（二）中，那句话的每个词都计算$p(w_i|Context_i)$后连乘起来得到联合概率，这个概率如果大于某个阈值，就认为是正常的话；否则就认为不是自然语言，要排除掉。

这里只是说明这个网络是怎么样的例子，真正重要的始终是那些词向量。

四．CBOW 加抽样的优化目标与解问题

4.1 抽样方法的意义与目标函数

为啥要抽样呢？目的跟（二）中的霍夫曼树其实是一样的，都是为了节省计算量，这个计算量就是式(2.1.2)中计算 p(A|C)的概率，因为这个概率实在不好算。论文《Distributed Representations ofWords and Phrasesand their Compositionality》中提到有一个叫 NCE 的方法可以来代替上面的那个 hierarchical softmax 方法（就是使用霍夫曼树的方法），但是由于 word2vec 只关心怎么学到高质量的词向量，所以就用了一种简单的 NCE 方法，称为 NEG，方法的本质就是在第 j 个句子的第 ij 个词 wij 处使用下面的式子代替$logp\left(w_{i_j}\middle|Context_{i_j}\right)$

$$\log\sigma\left(w_{i_j}\cdot C_{i_j}\right)+\sum_{k=1}^{K}E_{w_k\sim p_V(w)}\log\left(1-\sigma\left(w_k\cdot C_{i_j}\right)\right)$$

其中 E 下面的那个下标的意思是 wk 是符合某个分布的，在这里$p_V(w)$表示词频的分布。

这个式子的第二项是求 K 个期望，这 K 个期望中的每一个期望，都是在该上下文的情况下不出现这个词的期望。这里出现一个特别大的偷懒，就是认为这个期望只要抽取一个样本就能计算出来，当然，如果说遍历完整个语料库，其实还可以认为是抽取了多次实验的，因为每次抽取词的时候，是按照词频的分布来抽样的，如果抽样的次数足够多，在总体的结果上，这里计算的期望还是接近这个期望的真实值的，这个可以参考博文中 RBM 中计算梯度的时候的那个蒙特卡洛抽样来理解。

在这里，从代码上体现来看，就只用一个样本来估算这个期望的，所有式子被简化成了下面的形式

$$\log\sigma\left(w_{i_j}\cdot C_{i_j}\right)+\sum_{k=1}^{K}\log\left(1-\sigma\left(w_k\cdot C_{i_j}\right)\right)$$

用这个式子取代(2.2.2)中的$logp\left(w_{i_j}\middle|Context_{i_j}\right)$，就能得到 CBOW 加抽样的目标函数（去掉 1/V 的），这个目标函数也是极其像 logistic regression 的目标函数，其中 wij 是正类样本，wk 是负类样本。

为了统一表示，正类样本设置一个 label 为 1，负类样本设置 label 为 0，每个样本的负对数似然都变成下面的方式

$$f_w = -\text{label}\cdot\log\sigma\left(\text{w}\cdot C_{i_j}\right) - (1-\text{label})\log\left(1-\sigma\left(w\cdot C_{i_j}\right)\right)$$

4.2CBOW 加抽样方法的解法

解法还是用 SGD，所以对一个词 wij 来说，这个词本身是一个正类样本，同时对这个词，还随机抽取了 k 个负类样本，那么每个词在训练的时候都有 k+1 个样本，所以要做 k+1 次 SGD。

对于每个样本求两个梯度

$$\begin{aligned}\text{Fw(w)} = \frac{\partial f_w}{\partial w} &= -\text{label}\cdot\left(1-\sigma\left(\text{w}\cdot C_{i_j}\right)\right)\cdot C_{i_j} + (1-label)\cdot\sigma\left(\text{w}\cdot C_{i_j}\right)\cdot C_{i_j}\\ &= -\left(label-\sigma\left(\text{w}\cdot C_{i_j}\right)\right)\cdot C_{i_j}\end{aligned}$$

和

$$\begin{aligned}\text{Fc(w)} = \frac{\partial f_w}{\partial C_{i_j}} &= -\text{label}\cdot\left(1-\sigma\left(\text{w}\cdot C_{i_j}\right)\right)\cdot w + (1-label)\cdot\sigma\left(\text{w}\cdot C_{i_j}\right)\cdot w\\ &= -\left(label-\sigma\left(\text{w}\cdot C_{i_j}\right)\right)\cdot w\end{aligned}$$

两个梯度都有这么相似的形式实在太好了，然后开始迭代，代码里面有个奇怪的地方，就是一开的网络结构中的那个 V*m 的矩阵，用来保存的是每次抽中的负类样本的词向量，每一行代表一个词向量，刚好是所有词的词向量。每次抽样抽中一个词后，拿去进行迭代的（就是计算梯度什么的），就是这个矩阵中对应的那个词向量，而且这个矩阵还参与更新，就是第一个梯度实际更新的是这个矩阵里面的词向量。但总的看来，这个矩阵在迭代计算完了就丢弃了，因为最终更新的词向量，每次只有一个，就是公式一直出现的那个词 wij，最终输出的，也是输入里面的词向量（在图中是最下面的输出层的那些词向量），当然这个矩阵可以认为是隐藏层跟输出层的连接矩阵。

$$\text{R}_w^{n+1} = \text{R}_w^n - \eta\text{Fw}(\text{R}_w^n)$$

其中 w 代表每个样本对应的词向量，包括 wij 和抽中的词向量，注意更新的是 R 这个连接矩阵。词向量的更新公式如下

$$\text{w}_I^{n+1} = \text{w}_I^n - \eta\left(Fc\left(\text{R}_{w_I}^n\right) + \sum_{k=1}^{K} Fc\left(\text{R}_{w_k}^n\right)\right)$$

注意每个梯度的值的计算都是拿连接矩阵 R 中对应的那一行来算的，这样看起来可以避免每次都更新输出层的词向量，可能是怕搞乱了吧。

4.3CBOW 加抽样方法代码中的 trick

随机数是自己产生的，代码里面自己写了随机数程序。

每个词都要执行 negative 次抽样的，如果遇到了当前词（就是 label 为 1 的词）就提前退出抽样，这个步骤就提前完成了，这个也是一个比较费解的 trick。

其中前面说的 R 矩阵在代码中用 syn1neg 表示，C_{i_j} 在代码中用 neu1 表示，同样的，叶子节点里面的每个词向量在代码中用 syn0 表示，利用下标移动去读取。

```
436       // NEGATIVE SAMPLING
437       if (negative > 0) for (d = 0; d < negative + 1; d++) {
438         if (d == 0) {
439           target = word;
440           label = 1;
441         } else {
442           next_random = next_random * (unsigned long long)25214903917 + 11;
443           target = table[(next_random >> 16) % table_size];
444           if (target == 0) target = next_random % (vocab_size - 1) + 1;
445           if (target == word) continue;
446           label = 0;
447         }
448         l2 = target * layer1_size;
449         f = 0;
450         for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1neg[c + l2];
451         if (f > MAX_EXP) g = (label - 1) * alpha;
452         else if (f < -MAX_EXP) g = (label - 0) * alpha;
453         else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
454         for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
455         for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * neu1[c];
456       }
```

其中 442-446 就是抽样的代码了，是作者自己写的模块，然后 label 这个变量跟上文的 label 意思是一样的，f 就表示$\sigma\left(w\cdot C_{i_j}\right)$的值，syn1neg 就保存了矩阵 R 每一行的值，neu1e 还是累积误差，直到一轮抽样完了后再更新输入层的词向量。。

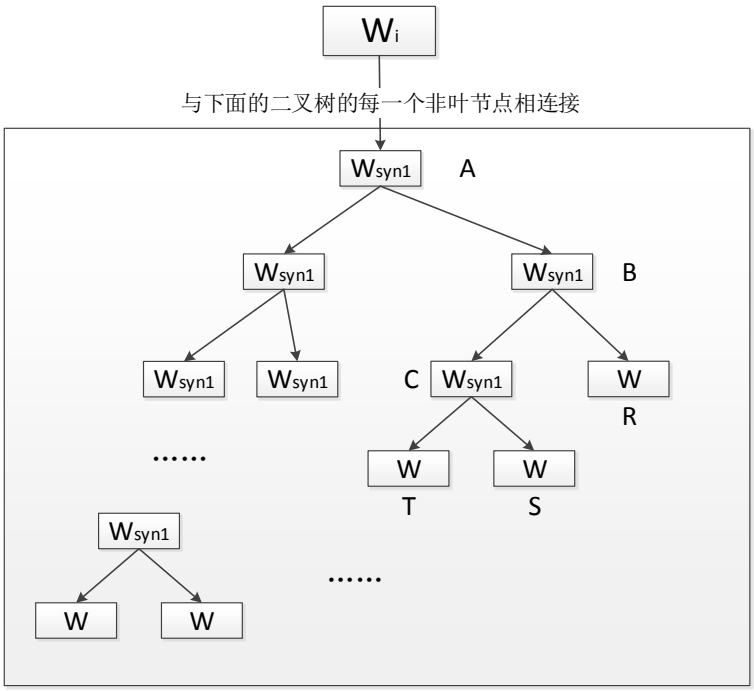
对输入层的更新模块和上面的（二）中一样的，都是更新所有的输入词。

```
457       // hidden -> in
458       for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
459         c = sentence_position - window + a;
460         if (c < 0) continue;
461         if (c >= sentence_length) continue;
462         last_word = sen[c];
463         if (last_word == -1) continue;
464         for (c = 0; c < layer1_size; c++) syn0[c + last_word * layer1_size] += neu1e[c];
465       }
```

五．Skip-gram 加层次的优化目标与解问题

5.1 网络结构与使用说明

网络结构如下图



其中的 Wi 是相应的词，词 Wi 与 huffman 树直接连接，没有隐藏层的。使用方法依然与 cbow 加层次的相似。

在判断“大家喜欢吃好吃的苹果”这句话是否自然语言的时候，是这么来的，同样比如计算到了“吃”这个词，同样随机抽到的 c=2，对吃这个词需要计算的东西比较多，总共要计算的概率是 p(大家|吃)，p(喜欢|吃)，p(好吃|吃)和 p(的|吃)总共四个，在计算 p(大家|吃)这个概率的时候，要用到上面图中的二叉树，假设“大家”这个词在 huffman 树根节点的右孩子的最左边的那个节点（途中的节点 T），就是图中右数第三个叶子节点；再假设“吃”这个词在 huffman 树的节点 S。再假设从根节点开始到这个叶子节点的路径上的三个非叶节点分别为 A，B，C（从高到低排列的），“吃”这个词的词向量设为 D，那么 p(大家|吃)这个概率可以用下面的公式算概率

$$p(\text{大家}|\text{吃}) = (1-\sigma(A\cdot D))\cdot\sigma(B\cdot D)\cdot\sigma(C\cdot D)$$

同样的方法计算 p(喜欢|吃)，p(好吃|吃)和 p(的|吃)，再把这四个概率连乘，得到了“吃”这个词的上下文概率，注意这只是一个词的概率。

把一整句话的所有词的概率都计算出来后进行连乘，得到的就是这句话是自然语言的概率。这个概率如果大于某个阈值，就认为是正常的话；否则就认为不是自然语言，要排除掉。

再声明一下，这里只是说明这个网络是怎么样的例子，真正重要的始终是那些词向量。

5.2 目标函数

假设语料库是有 S 个句子组成的一个句子序列（顺序不重要），整个语料库

有 V 个词，似然函数就会构建成下面的样子

$$L(\theta) = \prod_{j}^{S}\left(\prod_{i_j=1}^{T_j}\left(\prod_{-c_{ij}<u_{ij}<c_{ij},j\neq 0} p(w_{u_{ij}+i_j}|w_{i_j})\right)\right) \quad (5.2.1)$$

其中T_j表示第 j 个句子的词个数，$w_{u_{ij}+i_j}$表示词w_{i_j}左右的各c_{ij}个词的其中一个，注意c_{ij}对每个w_{i_j}都不一样的。极大似然要对整个语料库去做的。

但是，Google 这公司，为了代码风格的统一，用了一个 trick，我举例说明吧。

对于一开始的那句话：大家喜欢吃好吃的苹果，总共 6 个词，假设每次的 cij 都抽到了 2，按照上面的公式中的$\prod_{i_j=1}^{T_j}\left(\prod_{-c_{ij}<u_{ij}<c_{ij},j\neq 0} p(w_{u_{ij}+i_j}|w_{i_j})\right)$部分按每个词作为条件$w_{i_j}$展开，看到得到什么吧。

大家：P(喜欢|大家)*p(吃|大家)

喜欢：p(大家|喜欢)*p(吃|喜欢)*p(好吃|喜欢)

吃：p(大家|吃)*p(喜欢|吃)*p(好吃|吃)*p(的|吃)

好吃：p(喜欢|好吃)*p(吃|好吃)*p(的|好吃)*p(苹果|好吃)

的：p(吃|的)*p(好吃|的)*p(苹果|的)

苹果：p(好吃|苹果)*p(的|苹果)

把结果重新组合一下，得到下面的组合方式。

大家：p(大家|喜欢)*p(大家|吃)

喜欢：P(喜欢|大家)*p(喜欢|吃)*p(喜欢|好吃)

吃：p(吃|大家)*p(吃|喜欢)*p(吃|好吃)*p(吃|的)

好吃：p(好吃|喜欢)*p(好吃|吃)*p(好吃|的)*p(好吃|苹果)

的：p(的|吃)　　*p(的|好吃)*p(的|苹果)

苹果：p(苹果|好吃)*p(苹果|的)

不证明，不推导，直接得到下面的结论：

$$\prod_{i_j=1}^{T_j}\left(\prod_{-c_{ij}<u_{ij}<c_{ij},j\neq 0} p(w_{u_{ij}+i_j}|w_{i_j})\right) = \prod_{i_j=1}^{T_j}\left(\prod_{-c_{ij}<u_{ij}<c_{ij},j\neq 0} p(w_{i_j}|w_{u_{ij}+i_j})\right)$$

这个是网易有道的那个资料给出的结论，这里是变成了本文的方式来描述，具体参考网易有道的原文。

这个变化倒是莫名其妙的，不过也能理解，Google 公司的代码是要求很优美的，这样做能把代码极大地统一起来。

总对数似然就会是下面的样子

$$l(\theta) = \log L(\theta) = \frac{1}{V}\sum_{j=1}^{S}\left(\sum_{i_j=1}^{T_j}\left(\sum_{-c_{ij}<u_{ij}<c_{ij},j\neq 0} \log p(w_{i_j}|w_{u_{ij}+i_j})\right)\right) \quad (5.2.2)$$

其中的 V 表示整个语料库的词没有去重的总个数，整个目标函数也可以叫交

叉熵，但是这里对这也不感兴趣，一般去掉。

这就涉及到计算某个词的概率了，如$p(w_{i_j}|w_{u_{ij}+i_j})$，这个概率变了，条件变成上下文的词，计算条件概率的词变成了输入中要考察的那个词。当然，计算方法没有变，前面介绍的 huffman 树的计算方法到这里是一摸一样的。

$$\mathrm{p(w|I)} = \prod_{k=1}^{K} p(d_k|q_k, \mathrm{I}) = \prod_{k=1}^{K}((\sigma(q_k \cdot \mathrm{I}))^{1-d_k} \cdot (1-\sigma(q_k \cdot I))^{d_k})$$

其中 I 表示上下文的词，也就是（5.1）的例子中的那个词“大家”，也就是节点 T 就表示例子中的“大家”；当前词假设是“吃”，在 huffman 树上的节点是 S，qk 表示从根节点下来到“吃”这个词所在的叶子节点的路径上的非叶节点，dk 就是编码了，也可以说是分类，当 w 在某个节点如 qk 的左子树的叶子节点上时 dk=0，否则 dk=1。

用这个式子代替掉上面的(5.2.1)中的似然函数中的$p(w_{i_j}|w_{u_{ij}+i_j})$，当然每个变量都对号入座，就能得到总的似然函数了。

再对这个式子求个对数，得到

$$log\mathrm{p(w|I)} = \sum_{k=1}^{K}\big((1-d_k)log\sigma(q_k \cdot \mathrm{I}) + d_k \cdot (1-\sigma(q_k \cdot I))\big)$$

再利用这个式子替换掉(5.2.2)中的log$p(w_{i_j}|w_{u_{ij}+i_j})$就能得到总的对数似然函数，也就是目标函数，剩下的就是怎么解了。

可以注意的是，计算每个词（例中的“吃”）上下文概率的时候都要计算好几个条件概率的（例子中 p(吃|大家)，p(吃|喜欢)，p(吃|好吃)和 p(吃|的)），这每个条件概率又是需要在 huffman 树上走好几个非叶节点的，每走到一个非叶节点，都需要计算一个$(1-d_k)log\sigma(q_k \cdot \mathrm{I}) + d_k \cdot (1-\sigma(q_k \cdot I))$的。可以看到，走到每一个非叶节点，在总的对数似然函数中，都类似 logistic regression 的一个样本，为了方便描述，就用样本和 label 的方式在称呼这些东西。

跟一般的 logistic regression 一样，每走到一个非叶节点，如果是向左走的，就定义 label 为 1，为正样本；否则 label 就是 0，是负样本。这样 label=1-dk，每一个非叶节点都为整个问题产生了一个样本。

5.3 解法

解法选用的是 SGD，在处理每个样本时，对总目标函数的贡献是

$$\mathrm{lf} = p(d_k|q_k, \mathrm{I}) = -(1-d_k)log\sigma(q_k \cdot \mathrm{I}) - d_k \cdot (1-\sigma(q_k \cdot I))$$

计算梯度

$$Fq(q_k) = \frac{\partial \mathrm{lf}}{\partial q_k} = -(1-d_k) \cdot \big(1-\sigma(q_k \cdot I)\big) \cdot I - d_k \cdot \big(-\sigma(q_k \cdot I)\big) \cdot I$$
$$= -\big(1-d_k-\sigma(q_k \cdot I)\big) \cdot I$$

和

$$Fi(q_k) = \frac{\partial lf}{\partial I} = -(1-d_k)\cdot\big(1-\sigma(q_k\cdot I)\big)\cdot q_k - d_k\cdot\big(-\sigma(q_k\cdot I)\big)\cdot q_k$$
$$= -\big(1-d_k-\sigma(q_k\cdot I)\big)\cdot q_k$$

更新

$$q_k^{n+1} = q_k^n - \eta \mathrm{Fq}(q_k^n)$$
$$I^{n+1} = I^n - \eta \mathrm{Fi}(q_k^n)$$

5.4 代码中的 trick

对输入词 I 的更新是走完整个 huffman 树后对整个误差一起计算的，这个误差保存在 neu1e 这个数组里面。

```
475         // HIERARCHICAL SOFTMAX
476         if (hs) for (d = 0; d < vocab[word].codelen; d++) {
477           f = 0;
478           l2 = vocab[word].point[d] * layer1_size;
479           // Propagate hidden -> output
480           for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1[c + l2];
481           if (f <= -MAX_EXP) continue;
482           else if (f >= MAX_EXP) continue;
483           else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
484           // 'g' is the gradient multiplied by the learning rate
485           g = (1 - vocab[word].code[d] - f) * alpha;
486           // Propagate errors output -> hidden
487           for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
488           // Learn weights hidden -> output
489           for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * syn0[c + l1];
490         }
```

其中 480-483 行是计算$\sigma(q_k\cdot I)$的值，保存在 f 中，vocab[word].code[d]表示的就是 dk 的值，word = sen[sentence_position]，表示的是当前词，neu1e 就保存了从根节点到叶子节点的路径上的所有非叶节点的累积误差。

```
512           // Learn weights input -> hidden
513           for (c = 0; c < layer1_size; c++) syn0[c + l1] += neu1e[c];
```

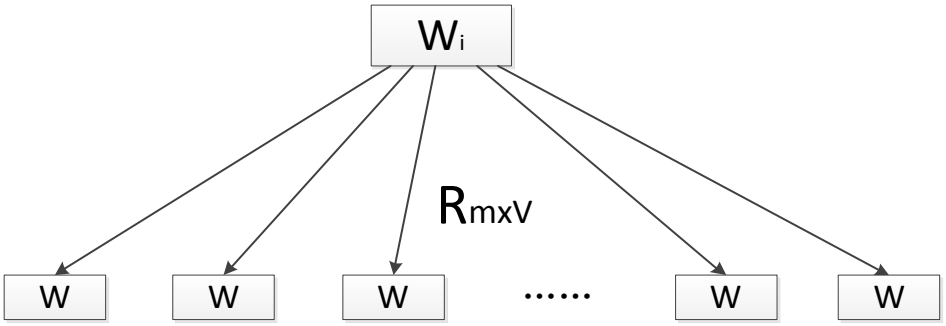
这个误差反向传播就简单多了，每次都是对一个词进行更新的，就是 p(w|I)中的那个 I。

六．Skip-gram 加抽样的优化目标与解问题

这个就简单说说吧，不清楚的看上面的。

6.1 网络结构与使用说明

网络结构如下



使用说明就不说的，同样是抽样的。

如（四）中，有一个矩阵 R，是在训练过程中用到的临时矩阵，这个矩阵连接隐层与所有输出节点，但是这个矩阵在使用这个网络的时候不怎么用得到，这里也是没弄清楚的一个地方。每个输出节点代表一个词向量。

同样如（五）中的例子，计算$p(\text{大家}|\text{吃})$这么一个概率，这里的计算方法就简单多了，就是随机从语料库里面抽取 c 个词，这里假设 c=3，抽中了 D，E，F 这三个词，又假设“吃”这个词的词向量是 A，那么就计算“吃”这个词的概率就用下面的公式

$$p\left(\text{吃}\middle|Context_{\text{吃}}\right)=\sigma(\mathrm{A}\cdot\mathrm{C})\cdot(1-\sigma(\mathrm{D}\cdot\mathrm{C}))\cdot(1-\sigma(\mathrm{E}\cdot\mathrm{C}))\cdot(1-\sigma(\mathrm{F}\cdot\mathrm{C}))$$

同样如（五）中，那句话的每个词都计算与上下文几个词的概率（p(大家|吃)，p(喜欢|吃)，p(好吃|吃)和 p(的|吃)）后连乘起来得到词“吃”的概率，所有词的概率计算出来再连乘就是整句话是自然语言的概率了。剩下的同上。

6.2 目标函数与解法

似然函数跟(5.2.1)一样的，对数似然函数跟(5.2.2)是一样的，但是计算 $\log p(w_{u_{ij}+i_j}|w_{i_j})$ 也就是 logp(w|I) 的的时候不一样，论文《Distributed Representations of Words and Phrases and their Compositionality》中认为 logp(w|I) 可以用下面的式子

$$\log\sigma(w\cdot I)+\sum_{k=1}^{K}E_{w\sim p_V(w)}\left[log\left(1-\sigma\left(w_k\cdot C_{i_j}\right)\right)\right]$$

代替。

同样可以和（四）中一样，设定正负样本的概念。为了统一表示，正类样本设置一个 label 为 1，负类样本设置 label 为 0，每个样本的负对数似然都变成下面的方式

$$f_w=-\text{label}\cdot\log\sigma(\mathrm{w}\cdot I)-(1-\text{label})\log\left(1-\sigma(w\cdot I)\right)$$

梯度

$$\begin{aligned}\mathrm{Fw(w)}=\frac{\partial f_w}{\partial w}&=-\text{label}\cdot\left(1-\sigma(\mathrm{w}\cdot I)\right)\cdot C_{i_j}+(1-label)\cdot\sigma(\mathrm{w}\cdot I)\cdot I\\&=-\left(label-\sigma(\mathrm{w}\cdot I)\right)\cdot I\end{aligned}$$

和

$$\begin{aligned}\mathrm{Fc(w)}=\frac{\partial f_w}{\partial I}&=-\text{label}\cdot\left(1-\sigma(\mathrm{w}\cdot I)\right)\cdot w+(1-label)\cdot\sigma(\mathrm{w}\cdot I)\cdot w\\&=-\left(label-\sigma(\mathrm{w}\cdot I)\right)\cdot w\end{aligned}$$

更新

$$\mathrm{R}_w^{n+1}=\mathrm{R}_w^n-\eta\mathrm{Fw}(\mathrm{R}_w^n)$$

$$\mathrm{I}^{n+1} = \mathrm{I}^{n} - \eta\left(Fc(\mathrm{R}^{n}_{\mathrm{I}^{n}}) + \sum_{k=1}^{K} Fc\left(\mathrm{R}^{n}_{w_k}\right)\right)$$

6.3 代码

还是每个词都要执行 negative 次抽样的，如果遇到了当前词（就是 label 为 1 的词）就提前退出抽样。

```
491         // NEGATIVE SAMPLING
492         if (negative > 0) for (d = 0; d < negative + 1; d++) {
493           if (d == 0) {
494             target = word;
495             label = 1;
496           } else {
497             next_random = next_random * (unsigned long long)25214903917 + 11;
498             target = table[(next_random >> 16) % table_size];
499             if (target == 0) target = next_random % (vocab_size - 1) + 1;
500             if (target == word) continue;
501             label = 0;
502           }
503           l2 = target * layer1_size;
504           f = 0;
505           for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1neg[c + l2];
506           if (f > MAX_EXP) g = (label - 1) * alpha;
507           else if (f < -MAX_EXP) g = (label - 0) * alpha;
508           else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
509           for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
510           for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * syn0[c + l1];
511         }
```

其中 493-502 就是抽样的代码，505-508 是计算$\sigma(\mathrm{w}\cdot I)$的值，保存在 f 中，syn1neg 就是保存了矩阵 R 中的每一行的值。而 neu1e 还是累积这误差，直到一轮抽样完了后再更新输入层的词向量。

```
512         // Learn weights input -> hidden
513         for (c = 0; c < layer1_size; c++) syn0[c + l1] += neu1e[c];
```

更新输入层还是一样。

七．一些总结

从代码看来，word2vec 的作者 Mikolov 是个比较实在的人，那种方法效果好就用哪种，也不纠结非常严格的理论证明，代码中的 trick 也是很实用的，可以参考到其他地方使用。

致谢

多位 Google 公司的研究员无私公开的资料。

多位博主的博客资料。

参考文献

[1] http://techblog.youdao.com/?p=915 Deep Learning 实战之 word2vec，网易有道的 pdf

[2]http://www.zhihu.com/question/21661274/answer/19331979 @杨超在知乎上的问答《Word2Vec 的一些理解》

[3]http://xiaoquanzi.net/?p=156hisen 博客的博文

[4] Hierarchical probabilistic neural network language model.Frederic Morin and Yoshua Bengio.

[5] Distributed Representations ofWords and Phrases and their CompositionalityT. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean.

[6] A neural probabilistic language modelY. Bengio, R. Ducharme, P. Vincent.

[7] Linguistic Regularities in Continuous Space Word Representations. Tomas Mikolov,Wen-tau Yih,Geoffrey Zweig

[8] Efficient Estimation of Word Representations in Vector Space.Tomas Mikolov,Kai Chen,Greg Corrado,Jeffrey Dean.

深度学习 word2vec 笔记之应用篇

好不容易学了一个深度学习的算法，大家是否比较爽了？但是回头想想，学这个是为了什么？吹牛皮吗？写论文吗？参加竞赛拿奖吗？

不管哪个原因，都显得有点校园思维了。

站在企业的层面，这样的方式显然是不符合要求的，如果只是学会了，公式推通了，但是没有在工作中应用上，那会被老大认为这是没有产出的。没有产出就相当于没有干活，没有干活的话就……呃……不说了。

下面就给大家弄些例子，说说在互联网广告这一块的应用吧。

一．对广告主的辅助

1.1 基本概念

互联网广告的广告主其实往往有他们的困惑，他们不知道自己的目标人群在哪里。所谓目标人群，就是广告主想向他们投广告的那帮人。就像互联网广告的一个大牛的一句名言——我知道互联网广告有一半是浪费的，问题是我不知道是哪一半。

这个困惑就给媒体带来一个义务——要帮助广告主定向他们的目标人群。

对于普通的广告主来说，比如说一个化妆品广告的广告主，它的目标人群很明显就是年轻的女性。注意关键词“年轻”和“女性”，这是决定媒体这边能否赚到钱的关键词。要知道对于媒体来说，广告主是它们的客户，满足客户的要求，客户就给它们钱，不满足客户的要求，就没有人为媒体买单；没有人为媒体买单，媒体就没有钱养它们的员工和机器，也弄不来新闻和互联网的其他内容，那样媒体公司就垮了……

那么在媒体这边，需要做的的工作就很明确了——满足它们的客户（也就是广告主）的需求。怎么满足呢？这工作说容易也容易，说简单也简单，就是把喜欢这个广告主的广告的人找出来，然后帮这个广告主把他们的广告投放给这些人，让这些人看到这个广告主的广告。

这个工作带来的问题就真多了，媒体又不是什么神人，比如说一个新闻网站，浏览这个网站的每天有 100 万人，这个新闻网站的员工不可能一个个去访问他们的用户（浏览这个网站的人），整天问他们你喜不喜欢化妆品啊，喜不喜欢体育啊之类的问题。

那怎么办呢？媒体的员工只好猜了，但是哪怕是猜都很费劲，想想都头疼，一百万人啊，一个个猜也得吃力不讨好啊。这时候计算机的作用就来了，用计算机猜嘛，而且不一定需要全部瞎猜的，因为用户如果注册了的话，还有一些用户的个人信息可以参考的。一般的网站注册的时候都要求提供年龄性别之类的个人

信息，有时候要要求写一些个人的兴趣什么的标签。这个时候这些数据就用上大用处了。

网站可以把注册用户的个人信息保存下来，然后提供广告主选择。如上面的那个化妆品的广告主，它就可以跟媒体提它的要求——我要向年轻的女性投放广告。媒体这个时候就可以提供一些条件给这个广告主选择，如媒体说我有很多用户，18 到 80 岁的都有，然后男性女性用户都有。广告主就可以根据这些条件选择自己的目标用户，如选择了 18 到 30 岁的女性用户作为目标人群。选中了目标人群后，广告主和媒体就可以谈价钱了，谈好了价钱广告主就下单，然后媒体就帮广告主投广告，然后媒体的钱就赚到了。

1.2 兴趣挖掘的必要性

上面多次提到的“目标人群”，就是广告主最关心的事情。客户最关心的事情自然也是媒体最关心的事情。所以媒体会尽力帮助它们的客户去定向它们的目标人群。

一般所谓的定向也不是媒体亲自有一个人来跟广告主谈的，是媒体建立好一个页面，这个页面上有一些选项，比如年龄，性别，地域什么的，都是条件。广告主在上面把自己的目标人群符合的条件输入，然后下单购买向这些人投放广告的机会。

媒体为了更好地赚钱，肯定是愿意把这个页面上的条件做得更加丰富一点，让更多的广告主觉得这个网站的用户里面有它们的目标人群，从而让更多的广告主愿意过来下单。

广告主的定向其实有粗细之分的，有些广告主粗放点，它们有钱，选的定向条件比较宽，就说女性的用户，全部都投放；有些就定向得比较窄，比如说，北京的 20 到 25 岁的女性，并且要喜欢羽毛球的用户。对于定向宽的广告主好处理，问题就是这些定向窄的广告主，它们还希望知道用户的兴趣所在，这就麻烦了。

为啥麻烦呢？一个用户的兴趣鬼才知道呢。就算当面问，人家也不乐意回答，何况就凭借一点点东西瞎猜。但是为了赚钱，瞎猜也得上的了，工业界为了赚这个钱，诞生了整整一个行业——数据挖掘，甚至在学术界还有一个更加生猛的名字——机器学习。学术界的那个名字和解释都是相当大气的：让机器学会像人一样思考。工业界就务实一点，只是对数据内容本身做一个挖掘，获取到啥呢？一般就是用户的兴趣啊，爱好啊什么的。这些东西供谁使用呢？暂时看来只有广告主愿意为这些掏钱，其他的就有些媒体做来让自己推荐的内容不至于让用户那么反感而已。

上面有个名词“数据”，没错了，这个词是互联网广告业，甚至是数据挖掘行业的核心的东西。所谓数据，这里简单点说就可以认为是用户的年龄、性别、地域等用户的基本属性；复杂点说可以说是用户兴趣、爱好，浏览记录等；更高级

的有用户的交易数据（当然这个高级的数据很少媒体能搞得到）等。

解释完“数据”这个词，结合一下广告这个场景，就可以得到活在媒体公司里面的互联网广告行业数据挖掘工程师的工作是什么了。他们的工作就是：根据用户自身的基本属性和用户流量的网页记录以及内容，想方设法让计算机猜出用户的兴趣爱好。用户的兴趣爱好“挖掘”出来后，就可以作为定向条件放到上面说的那个网页上面供广告主选择了。这事情整好了，广告投了有人点击，公司的钱就赚到了；没整好，广告没人点击，广告主不乐意下单了，公司就赚不到钱……怎么着？炒这些工程师的鱿鱼去。

上面可以看到了，辅助广告主定位它们的目标人群是很重要的。

经过一番的探索，word2vec 在互联网广告上面也是可以辅助广告主定向他们的目标人群的，下面就讲讲这个算法在互联网广告的应用吧。

1.3 利用 word2vec 给广告主推荐用户

为了用上 word2vec，把场景转换到一个新闻媒体如 A 公司。

在 A 公司的多个页面中，电商公司 B 有他们的一个主页，专门介绍他们公司一些产品促销，抢购和发布会什么的。

公司 A 目前有很多用户的浏览数据，如用户 u 浏览了公司 A 的页面 a1，a2，a3 等。

把这些数据处理一下，整合成 word2vec 能处理的数据，如下

U1 a1,a2,a3……

U2 a2,a3,a5,……

U3 a1,a3,a6,……

其中 u1，u2，u3 表示不同的用户，后面的一串表示这些用户的浏览记录，如 U1 a1,a2,a3 表示用户 u1 先浏览了页面 a1，再浏览 a2，然后浏览了 a3,……

这些数据还不符合 word2vec 的输入数据格式，把第一列去掉，变成下面的样子

a1,a2,a3……

a2,a3,a5,……

a1,a3,a6,……

这些数据就可以作为 word2vec 的输入数据了。

就把这些数据作为 word2vec 的训练数据，词向量维度为 3，进行训练，完成后得到下面的输出

A1 (0.3,-0.5,0.1)

A2 (0.1,0.4,0.2)

A3 (-0.3,0.7,0.8)

……

An (0.7,-0.1,0.3)

就得到了每个页面的向量。

这些向量有啥意义呢？其实单个向量的意义不大，只是用这些向量可以计算一个东西——距离，这个距离是页面之间的距离，如页面 a1 和 a2 可以用欧式距离或者 cos 距离计算公式来计算一个距离，这个距离是有意义的，表示的是两个网页在用户浏览的过程中的相似程度（也可以认为是这两个页面的距离越近，被同一个人浏览的概率越大）。注意这个距离的绝对值本身也是没有意义的，但是这个距离的相对大小是有意义的，意思就是说，假设页面 a1 跟 a2、a3、a4 的距离分别是 0.3、0.4、0.5，这 0.3、0.4、0.5 没啥意义，但是相对来说，页面 a2 与 a1 的相似程度就要比 a3 和 a4 要大。

那么这里就有玄机了，如果页面 a1 是电商公司 B 的主页，页面 a2、a3、a4 与 a1 的距离在所有页面里面是最小的，其他都比这三个距离要大，那么就可以认为同一个用户 u 浏览 a1 的同时，浏览 a2、a3、a4 的概率也比较大，那么反过来，一个用户经常浏览 a2、a3、a4，那么浏览 a1 的概率是不是也比较大呢？从实验看来可以这么认为的。同时还可以得到一个推论，就是用户可能会喜欢 a1 这个页面对应的广告主的广告。

这个在实验中实际上也出现过的。这里模拟一个例子吧，如 a1 是匹克体育用品公司在媒体公司 A 上的官网，a2 是湖人队比赛数据页，a3 是热火队的灌水讨论区，a4 是小牛队的球员讨论区。这个结果看起来是相当激动人心的。

根据这样的一个结果，就可以在广告主下单的那个页面上增加一个条件——经常浏览的相似页面推荐，功能就是——在广告主过来选条件的时候，可以选择那些经常浏览跟自己主页相似的页面的用户。举个例子就是，当匹克体育用品公司来下单的时候，页面上给它推荐了几个经常浏览页面的粉丝：湖人队比赛数据页，热火队的灌水讨论区，小牛队的球员讨论区。意思是说，目标人群中包括了经常浏览这三个页面的人。

这个功能上线后是获得过很多广告主的好评的。

这样 word2vec 这个算法在这里就有了第一种用途。

二．对 CTR 预估模型的帮助

根据另一篇博文《互联网广告综述之点击率系统》，里面需要计算的用户对某广告的 CTR。在实际操作的时候，这个事情也是困难重重的，其中有一个冷启动问题很难解决。冷启动问题就是一个广告是新上线的，之前没有任何的历史投放数据，这样的广告由于数据不足，点击率模型经常不怎么凑效。

但是这个问题可以使用同类型广告点击率来缓解，意思就是拿一个同行的广告的各种特征作为这个广告的特征，对这个新广告的点击率进行预估。

同行往往太粗糙，那么怎么办呢？可以就利用跟这个广告主比较相似的广告的点击率来预估一下这个广告的点击率。

上面说过，可以得到每个页面的词向量。这里的方法比较简单，如在媒体公司 A 上面有 1000 个广告主，它们的主页分别是 a1、a2、……、a1000。

根据上面的方法，得到了这 1000 个词向量，然后运行 kmean 或者其他聚类算法，把这 1000 个广告主聚成 100 个簇，然后每个簇里面的广告主看成是一个。

这里可以模拟一个例子，聚类完成后，某个簇 c 里面包含了几个广告主的主页，分别是京东商城，天猫，唯品会，当当，聚美优品，1 号店，蘑菇街，卓越，亚马逊，淘宝这 10 个，这 10 个的目标人群看起来基本是一致的。

这里的看成是一个簇是有意义的，比如说第一个簇 c1，c1 这个簇里面的所有历史投放数据和实时数据可以做特征，来预估这个流量对这个簇的 ctr。得到这个 ctr 后，就很有用了，如果某广告投放数据比较充分，就直接预估这个广告的 ctr；如果某广告的历史投放数据很少，就用这个广告主所在的簇的 ctr 来代替这个广告，认为对簇的 ctr 就是这个广告的 ctr，这样能让一个新广告也能得到相对靠谱的预估 ctr，保证不至于乱投一番。

三．一些总结

如何应用好一个算法，确实是很多算法工程师的一个重大课题。

数据挖掘算法工程师经常要面对的一个难题就是：这个算法怎么用到我们的数据上面来？有不少同学会认为是：我到了公司，就发明一个很牛逼的算法，把公司的原来的问题解决掉，然后大大增加了效果，获得了领导的好评。这个天真烂漫的想法就不评价了，免得被说打击人。互联网企业里面的真实情况是算法工程师面对那一团乱遭的数据，得想尽办法去把数据整合成能用的格式。

拿上面的（1.3）中的例子，那个把数据组合成 a1,a2,a3……这样一行行的，然后进入 word2vec 去进行训练是最难想到的而且是最核心的东西，虽然明着说是 word2vec 这个算法厉害，实际上面是“把数据整合成合适的方式交给 word2vec 进行训练”这个想法重要，因为尝试了很多想法，做了很多实验才能想到这样的一招的。

还有数据的整合其实也费了很多功夫的，比如说媒体有些用户是一些机器的账号，人家乱搞的，要想办法排除掉的，而“想办法排除”这么简单一句话，真正要做的工作真是多多的有。

哪怕结果都训练出来了，怎么解释这个结果是好的？这个问题也是得想了一段时间的，后来是实验发现了利用词向量的距离来评价相似性这个东西最靠谱，然后才用上的。

一个数据挖掘的过程其实不简单，这个博客也没办法一一体现做的过程里面的那些各种折腾，各种不顺畅。

数据挖掘工程师经常要面对的另一个难题就是：明明理论上推得杠杠的，算法性能也是杠杠的，但是对于互联网广告的效果，怎么就那么不咸不淡的呢？

这个问题真没有什么统一的答案，这种现象多了去了。经常遇到的原因有：数据本身处理的方式不对和算法不合适。

所谓数据本身处理的方式，可以参看博文《互联网广告综述之点击率特征工程》，里面说的那些方法不是从哪本书上面看到的，是经过比较长时间实践，然后各种折腾，各种特征取舍，各种胡思乱想，各种坑踩出来的。可能志在学术的人看起来都简单，实际上课本那些东西，学生们吹起牛皮来不眨眼的那些东西，一跟真实应用场景结合起来就各种坑要踩的了。

拿上面的（二）中的例子来看。方法简单得不得了，但是可以想象一下，word2vec 牛逼啊，kmeans 牛逼啊，第一次聚类出来的结果也不过如此。后来又加入了每个广告主的行业和地域作为特征，而且这个加特征，就是直接把行业和地域处理一下，连接到广告主的词向量后面的。如 a1 的词向量是(0.3,-0.5,0.1)，然后假设只有两个行业，体育和化妆品，处理成二值特征，占据第 4 和 5 两个 index，第 4 个特征为 1，第 5 个特征为 0 表示体育类广告主，反过来，第 4 个特征为 0，第 5 个特征为 1 表示化妆品；再对地域的下标做了一下处理，成为二值特征，比如说占据了 6 到 10 这 5 个位置（假设第 6 个位置为 1，其余 7 到 10 为 0 表示北京；第 7 个位置为 1，其余为 0 表示广东，以此类推）。

经过了上面的处理，再用 kmeans 进行聚类，从聚类后一个个簇去看，结果看起来才顺眼了很多。上面的行业和地域特征的加入，也是用了比较多的经验的，不是凭空乱整出来的一个吹牛皮的东西，当然谁有更好的方法，也可以提出来试试看。另外还希望大家注意关键字“一个个簇去看”，这个工作真是费时费力，比较辛苦的。

以上举了一些例子，也把互联网广告的数据挖掘算法工程师的一些工作中的成功和不成功的地方都说出来了，基本上算是实话实说，希望对大家有点帮助吧。有过类似经历的人能看懂，没啥兴趣的就呵呵吧。

致谢

多位同事提供的建议与指导。

多位 google 研究员有关 word2vec 的资料。