Deep Learning 实战之 word2vec

邓澍军、陆光明、夏龙
网易有道
2014.02.27

目录

前言：

Deep Learning 已经很火了，本文作者算是后知后觉者，主要原因是作者的目前工作是广告点击率预测，而之前听说 Deep Learning 最大的突破还是在图像语音领域，而在 NLP 和在线广告点击预测方面的突破还不够大。但后来听说 Google 开源的 word2vec 还挺有意思，能够把词映射到 K 维向量空间，甚至词与词之间 的向量操作还能和语义相对应。如果换个思路，把词当做 feature，那么 word2vec 就可以把 feature 映射到 K 维向量空间，应该可以为现有模型提供更多的有用信息，基于这个出发点，作者对 word2vec 的相关代码和算法做了相关调研，本文就是作者关于 word2vec 调研的总结，也是作为自己以后备用。存在疏漏之处，欢迎大家反馈：shujun_deng@163.com。

本文所有讨论都是基于 word2vec 以下版本的代码：

```
URL: http://word2vec.googlecode.com/svn/trunk
Repository Root: http://word2vec.googlecode.com/svn
Repository UUID: c84ef02e-58a5-4c83-e53e-41fc32d635eb
Revision: 37
Node Kind: directory
Schedule: normal
Last Changed Author: tmikolov@google.com
Last Changed Rev: 37
Last Changed Date: 2013-12-19 08:16:00 +0800 (Thu, 19 Dec 2013)
```

一、什么是 word2vec？

word2vec 是 Google 在 2013 年年中开源的一款将词表征为实数值向量的高效工具，采用的模型有 CBOW（Continuous Bag-Of-Words，即连续的词袋模型）和 Skip-Gram 两种。word2vec 代码链接为：https://code.google.com/p/word2vec/，遵循 Apache License 2.0 开源协议，是一种对商业应用友好的许可，当然需要充分尊重原作者的著作权。

word2vec 一般被外界认为是一个 Deep Learning（深度学习）的模型，究其原因，可能和 word2vec 的作者 Tomas Mikolov 的 Deep Learning 背景以及 word2vec 是一种神经网络模型相关，但我们谨慎认为该模型层次较浅，严格来说还不能算是深层模型。当然如果 word2vec 上层再套一层与具体应用相关的输出层，比如 Softmax，此时更像是一个深层模型。

word2vec 通过训练，可以把对文本内容的处理简化为 *K* 维向量空间中的向量运算，而向量空间上的相似度可以用来表示文本语义上的相似度。因此，word2vec 输出的词向量可以被用来做很多 NLP 相关的工作，比如聚类、找同义词、词性分析等等。而 word2vec 被人广为传颂的地方是其向量的加法组合运算（Additive Compositionality），官网上的例子是：vector('Paris') - vector('France') + vector('Italy') ≈vector('Rome')，vector('king') - vector('man') + vector('woman') ≈ vector('queen')。但我们认为这个多少有点被过度炒作了，很多其他降维或主题模型在一定程度也能达到类似效果，而且 word2vec 也只是少量的例子完美符合这种加减法操作，并不是所有的 case 都满足。

word2vec 大受欢迎的另一个原因是其高效性，Mikolov 在论文[2]中指出一个优化的单机版本一天可训练上千亿词。

二、快速入门

1. 代码下载：http://word2vec.googlecode.com/svn/trunk/
2. 针对个人需求修改 makefile 文件，比如作者使用的 linux 系统就需要把 makefile 编译选项中的-Ofast 要更改为-O2 或者-g（调试时用），同时删除编译器无法辨认的-march=native 和-Wno-unused-result 选项。有些系统可能还需要修改相关的 c 语言头文件，具体网上搜搜应该可以解决。
3. 运行“make”编译 word2vec 工具。
4. 运行 demo 脚本：./demo-word.sh

   demo-word.sh 中的代码如下，主要工作为：

   1） 编译（make）

   2） 下载训练数据 text8，如果不存在。text8 中为一些空格隔开的英文单词，但不含标点符号，一共有 1600 多万个单词。

   3） 训练，大概一个小时左右，取决于机器配置

   4） 调用 distance，查找最近的词

```
make
if [ ! -e text8 ]; then
  wget http://mattmahoney.net/dc/text8.zip -O text8.gz
  gzip -d text8.gz -f
fi
time ./word2vec -train text8 -output vectors.bin -cbow 0 -size 200 -window 5 -negative 0 -hs 1 -sample 1e-3 -threads 12 -binary 1
./distance vectors.bin
```

训练完毕后，输入 china 就会找出与其最相近的词：

```
Enter word or sentence (EXIT to break): china

Word: china  Position in vocabulary: 486

                                              Word       Cosine distance
------------------------------------------------------------------------
                                            taiwan              0.768188
                                             japan              0.652825
                                             macau              0.614888
                                             korea              0.614887
                                               prc              0.613579
                                           beijing              0.605946
                                            taipei              0.592367
                                          thailand              0.577905
                                          cambodia              0.575681
                                         singapore              0.569950
                                          republic              0.567597
                                          mongolia              0.554642
                                           chinese              0.551576
```

5. 也可以自己执行其他相关的程序，比如作者比较推崇的向量加减法操作，在命令行执行./word-analogy vectors.bin 即可。相关结果如下：

```
Enter three words (EXIT to break): france paris china

Word: france  Position in vocabulary: 303

Word: paris  Position in vocabulary: 1055

Word: china  Position in vocabulary: 486

                                              Word              Distance
------------------------------------------------------------------------
                                           beijing              0.494238
                                             tokyo              0.441599
                                          shanghai              0.421356
```

当然效果的好坏是取决于你语料大小和质量以及训练相关参数的。

6. 模型的输出 vectors.bin 是每个词及其对应的浮点数向量，只是默认为二进制的，不好查看，如果想要查看文本形式，请在训练时把 binary 选项设置为 0。

7. 也可以./demo-phrases.sh 进行短语训练，这样 los angeles 就会被当成一个整体来看待。distance 的相关测试如下：

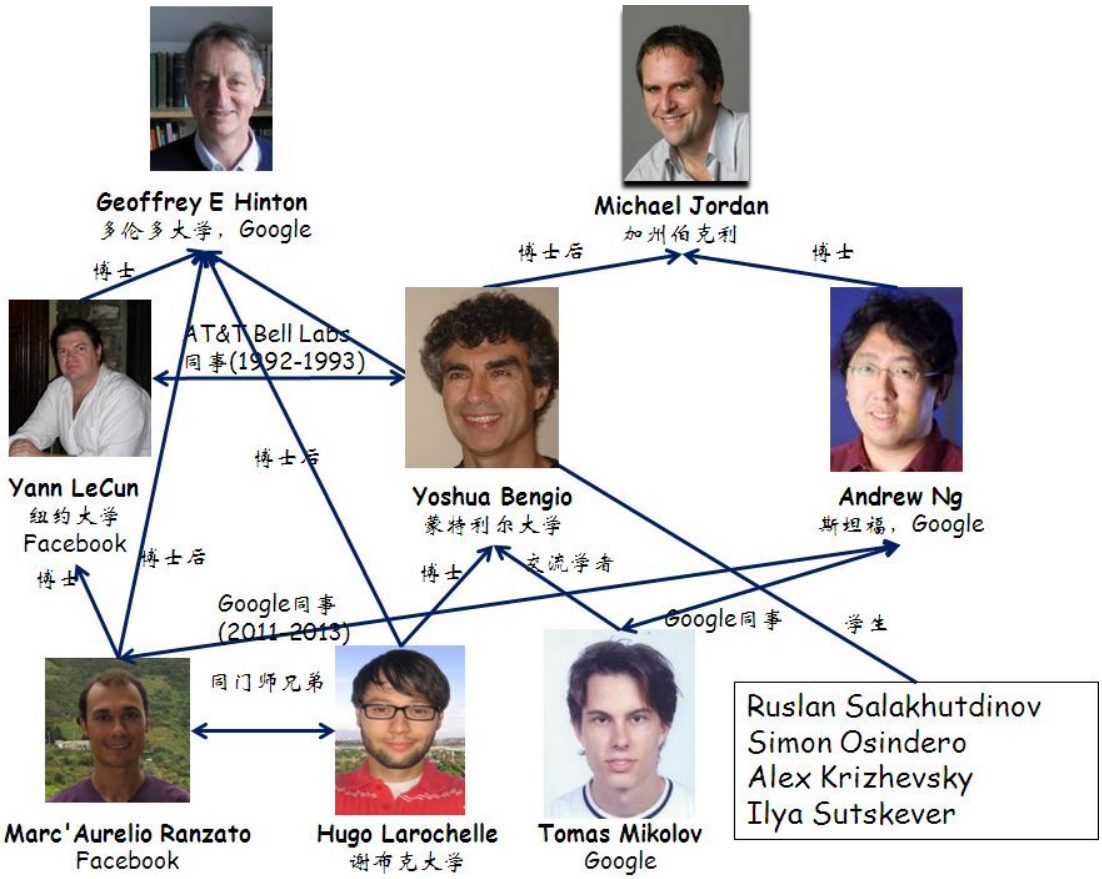
```
Enter word or sentence (EXIT to break): los_angeles

Word: los_angeles  Position in vocabulary: 1680

                                              Word       Cosine distance
------------------------------------------------------------------------
                                     san_francisco              0.614459
                                         san_diego              0.609573
                                        california              0.607403
                                            lakers              0.575140
                                           seattle              0.565795
                                             miami              0.561132
                                           oakland              0.550734
```

8. 如果是中文的语料，则需要调用相应的分词工具将预料中的词用空格隔开。

三、作者八卦



感觉上 Deep Learning 圈子很小，一般离不开 Hinton，Bengio，Lecun，Ng 等人，Tomas Mikolov 也不例外，他的工作很多参考了 Bengio 的工作，而且曾在

Bengio 实验室呆过一段时间，也一起发表过 paper。而且相关的语言模型也被这几个大牛翻江倒海搞了好多年。

还有一个 Deep Learning 相关开源工具 SENNA 的作者 Ronan Collobert 则是 Samy Bengio（也是张栋的博士生导师）的博士生，而 Samy Bengio 则是 Yoshua Bengio 的亲弟弟。

四、背景知识

4.1 词向量

1. One-hot Representation

NLP 相关任务中最常见的第一步是创建一个词表库并把每个词顺序编号。这实际就是词表示方法中的 One-hot Representation，这种方法把每个词顺序编号，每个词就是一个很长的向量，向量的维度等于词表大小，只有对应位置上的数字为 1，其他都为 0。当然在实际应用中，一般采用稀疏编码存储，主要采用词的编号。

这种表示方法一个最大的问题是无法捕捉词与词之间的相似度，就算是近义词也无法从词向量中看出任何关系。此外这种表示方法还容易发生维数灾难，尤其是在 Deep Learning 相关的一些应用中。

2. Distributed Representation

Distributed representation 最早由 Hinton 在 1986 年提出[8]。其基本思想是通过训练将每个词映射成 K 维实数向量（K 一般为模型中的超参数），通过词之间的距离（比如 cosine 相似度、欧氏距离等）来判断它们之间的语义相似度。而 word2vec 使用的就是这种 Distributed representation 的词向量表示方式。

4.2 统计语言模型

传统的统计语言模型是表示语言基本单位（一般为句子）的概率分布函数，这个概率分布也就是该语言的生成模型。一般语言模型可以使用各个词语条件概率的形式表示：

$p(s) = p(w_1^T) = p(w_1, w_2, \dots, w_T) = \prod_{t=1}^{T} p(w_t|\text{Context})$

其中 Context 即为上下文，根据对 Context 不同的划分方法，可以分为五大类：

（1）上下文无关模型（Context=NULL）

该模型仅仅考虑当前词本身的概率，不考虑该词所对应的上下文环境。这是一种最简单，易于实现，但没有多大实际应用价值的统计语言模型。

$p(w_t|\text{Context}) = p(w_t) = \frac{N_{w_t}}{N}$

这个模型不考虑任何上下文信息，仅仅依赖于训练文本中的词频统计。它是 n-gram 模型中当 n=1 的特殊情形，所以有时也称作 Unigram Model（一元文法统计模型）。实际应用中，常被应用到一些商用语音识别系统中。

（2）*n*-gram 模型（Context= $w_{t-n+1}, w_{t-n+2},\dots, w_{t-1}$）

n=1 时，就是上面所说的上下文无关模型，这里 *n*-gram 一般认为是 N>=2 是的上下文相关模型。当 *n*=2 时，也称为 Bigram 语言模型，直观的想，在自然语言中 “白色汽车”的概率比“白色飞翔”的概率要大很多，也就是 *p*(汽车|白色)> *p*(飞翔|白色)。*n*>2 也类似，只是往前看 *n*-1 个词而不是一个词。

一般 *n*-gram 模型优化的目标是最大 log 似然，即：

$\prod_{t=1}^{T} p_t(w_t|w_{t-n+1}, w_{t-n+2},\dots, w_{t-1})\log p_m(w_t|w_{t-n+1}, w_{t-n+2},\dots, w_{t-1})$

n-gram 模型的优点包含了前 N-1 个词所能提供的全部信息，这些信息对当前词出现具有很强的约束力。同时因为只看 N-1 个词而不是所有词也使得模型的效率较高。

n-gram 语言模型也存在一些问题：

1. *n*-gram 语言模型无法建模更远的关系，语料的不足使得无法训练更高阶的语言模型。大部分研究或工作都是使用 Trigram，就算使用高阶的模型，其统计到的概率可信度就大打折扣，还有一些比较小的问题采用 Bigram。

2. 这种模型无法建模出词之间的相似度，有时候两个具有某种相似性的词，如果一个词经常出现在某段词之后，那么也许另一个词出现在这段词后面的概率也比较大。比如“白色的汽车”经常出现，那完全可以认为“白色的轿车”也可能经常出现。

3. 训练语料里面有些 *n* 元组没有出现过，其对应的条件概率就是 0，导致计算一整句话的概率为 0。解决这个问题有两种常用方法：

方法一为平滑法。最简单的方法是把每个 n 元组的出现次数加 1，那么原来出现 *k* 次的某个 *n* 元组就会记为 *k*+1 次，原来出现 0 次的 *n* 元组就会记为出现 1 次。这种也称为 Laplace 平滑。当然还有很多更复杂的其他平滑方法，其本质都是将模型变为贝叶斯模型，通过引入先验分布打破似然一统天下的局面。而引入先验方法的不同也就产生了很多不同的平滑方法。

方法二是回退法。有点像决策树中的后剪枝方法，即如果 *n* 元的概率不到，那就往上回退一步，用 *n*-1 元的概率乘上一个权重来模拟。

（3）*n*-pos 模型（Context= $c(w_{t-n+1)}, c(w_{t-n+2}),\dots, c(w_{t-1})$）

严格来说 *n*-pos 只是 *n*-gram 的一种衍生模型。*n*-gram 模型假定第 t 个词出现概率条件依赖它前 N-1 个词，而现实中很多词出现的概率是条件依赖于它前面词的语法功能的。*n*-pos 模型就是基于这种假设的模型，它将词按照其语法功能进行分类，由这些词类决定下一个词出现的概率。这样的词类称为词性（Part-of-Speech，简称为 POS）。*n*-pos 模型中的每个词的条件概率表示为：

$$p(s) = p(w_1^T) = p(w_1, w_2,\dots, w_T)=\prod_{t=1}^{T} p(w_t|c(w_{t-n+1)}, c(w_{t-n+2}),\dots, c(w_{t-1}))$$

c 为类别映射函数，即把 T 个词映射到 K 个类别（1=<K<=T）。实际上 n-Pos 使用了一种聚类的思想，使得原来 *n*-gram 中$w_{t-n+1}, w_{t-n+2},\dots, w_{t-1}$中的可能为 T^{N-1} 减少到$c(w_{t-n+1)}, c(w_{t-n+2}),\dots, c(w_{t-1})$中的 K^{N-1}，同时这种减少还采用了语义有意义的类别。

当然 *n*-pos 模型还有很多变种，具体可以参考论文[9]。

（4）基于决策树的语言模型

上面提到的上下文无关语言模型、*n*-gram 语言模型、*n*-pos 语言模型等等，

都可以以统计决策树的形式表示出来。而统计决策树中每个结点的决策规则是一个上下文相关的问题。这些问题可以是“前一个词时 w 吗？”“前一个词属于类别 c_i 吗？”。当然基于决策树的语言模型还可以更灵活一些，可以是一些“前一个词是动词？”，“后面有介词吗？”之类的复杂语法语义问题。

基于决策树的语言模型优点是：分布数不是预先固定好的，而是根据训练预料库中的实际情况确定，更为灵活。缺点是：构造统计决策树的问题很困难，且时空开销很大。

（5）最大熵模型

最大熵原理是 E.T. Jayness 于上世纪 50 年代提出的，其基本思想是：对一个随机事件的概率分布进行预测时，在满足全部已知的条件下对未知的情况不做任何主观假设。从信息论的角度来说就是：在只掌握关于未知分布的部分知识时，应当选取符合这些知识但又能使得熵最大的概率分布。

$$p(w|Context) = \frac{e^{\sum_i \lambda_i f_i(context,w)}}{Z(Context)}$$

其中λ_i是参数，$Z(Context)$为归一化因子，因为采用的是这种 Softmax 形式，所以最大熵模型有时候也称为指数模型。

（6）自适应语言模型

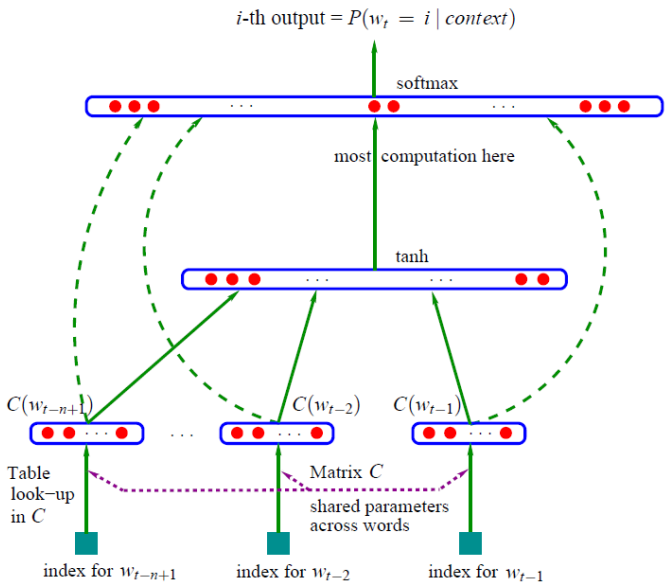
前面的模型概率分布都是预先从训练语料库中估算好的，属于静态语言模型。而自适应语言模型类似是 Online Learning 的过程，即根据少量新数据动态调整模型，属于动态模型。在自然语言中，经常出现这样现象：某些在文本中通常很少出现的词，在某一局部文本中突然大量地出现。能够根据词在局部文本中出现的情况动态地调整语言模型中的概率分布数据的语言模型成为动态、自适应或者基于缓存的语言模型。通常的做法是将静态模型与动态模型通过参数融合到一起，这种混合模型可以有效地避免数据稀疏的问题。

还有一种主题相关的自适应语言模型，直观的例子为：专门针对体育相关内容训练一个语言模型，同时保留所有语料训练的整体语言模型，当新来的数据属于体育类别时，其应该使用的模型就是体育相关主题模型和整体语言模型相融合的混合模型。

4.3 NNLM

NNLM 是 Neural Network Language Model 的缩写，即神经网络语言模型。神经网络语言模型方面最值得阅读的文章是 Deep Learning 二号任务 Bengio 的《A Neural Probabilistic Language Model》，JMLR 2003。

NNLM 采用的是 Distributed Representation，即每个词被表示为一个浮点向量。其模型图如下：



目标是要学到一个好的模型：

$f(w_t, w_{t-1}, \ldots, w_{t-n+2}, w_{t-n+1}) = p(w_t | w_1^{t-1})$

需要满足的约束为：

$f(w_t, w_{t-1}, \ldots, w_{t-n+2}, w_{t-n+1}) > 0$

$$\sum_{i=1}^{|V|} f(i, w_{t-1}, \ldots, w_{t-n+2}, w_{t-n+1}) = 1$$

上图中，每个是输入词都被映射为一个向量，该映射用 C 表示，所以 $C(w_{t-1})$ 即为 w_{t-1} 的词向量。g 为一个前馈或递归神经网络，其输出是一个向量，向量中的第 i 个元素表示概率 $p(w_t = i | w_1^{t-1})$。训练的目标依然是最大似然加正则项，即：

$$\text{Max Likelihood} = \max \frac{1}{T} \sum_t log f(w_t, w_{t-1}, \ldots, w_{t-n+2}, w_{t-n+1}; \theta) + R(\theta)$$

其中θ为参数，$R(\theta)$ 为正则项，输出层采用 softmax 函数：

$$p(w_t | w_{t-1}, \ldots, w_{t-n+2}, w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}$$

其中 y_i 是每个输出词 i 的未归一化 log 概率，计算如下：

$$y = b + Wx + U\tanh(d + Hx)$$

其中 b，W，U，d 和 H 都是参数，x 为输入，需要主要的是，一般的神经网络输入是不需要优化，而在这里，$x = (C(w_{t-1}), C(w_{t-2}), \ldots, C(w_{t-n+1}))$，也是需要优化的参数。在图中，如果下层原始输入 x 不直接连到输出的话，可令 b=0，W=0。

如果采用随机梯度算法的话，梯度更新的法则为：

$$\theta \leftarrow \theta + \epsilon \frac{\partial \log p(w_t | w_{t-1}, \ldots, w_{t-n+2}, w_{t-n+1})}{\partial \theta}$$

其中ϵ为 learning rate。需要注意的是，一般神经网络的输入层只是一个输入值，而在这里，输入层 x 也是参数（存在 C 中），也是需要优化的。优化结束之后，词向量有了，语言模型也有了。

这个 Softmax 模型使得概率取值为(0,1)，因此不会出现概率为 0 的情况，也就是自带平滑，无需传统 n-gram 模型中那些复杂的平滑算法。Bengio 在

APNews 数据集上做的对比实验也表明他的模型效果比精心设计平滑算法的普通 n-gram 算法要好 10% 到 20%。

4.4 其他 NNLM

请参考 licstar 的《Deep Learning in NLP （一）词向量和语言模型》一文[10]，这里不再累述

2.2 C&W 的 SENNA
2.3 M&H 的 HLBL
2.4 Mikolov 的 RNNLM
2.5 Huang 的语义强化

4.5 Log-Linear 模型

Log-linear 也是 Word2vec 所用模型的前身。Log-linear 模型有以下成分组成：

1. 一个输入集合 *X*；
2. 一个标注集合 *Y*，*Y* 是有限的；
3. 一个正整数 *K* 指定模型中向量的维数；
4. 一个映射函数 f：$X \times Y \rightarrow \mathbb{R}^K$，即将任意的（x, y）对映射到一个 K 维实数向量；
5. 一个 K 维的实数参数向量 $v \in \mathbb{R}^K$。

对任意的 x，y，模型定义的条件概率为：

$$p(y \mid x; v) = \frac{e^{v \cdot f(x,y)}}{\sum_{y' \in Y} e^{v \cdot f(x,y')}}$$

为何叫 log-linear？原因是分子部分取完 log 后是 *v* 中个元素的线性表达式，例如 $v_1+v_3+v_6$。Michael Collins 在《Log-Linear Models》给出的一个 *K*=8 的例子为：

$$
\begin{aligned}
f_1(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model} \\ 0 & \text{otherwise} \end{cases} \\
f_2(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model and } w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases} \\
f_3(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any}, w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases} \\
f_4(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any} \\ 0 & \text{otherwise} \end{cases} \\
f_5(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-1} \text{ is an adjective} \\ 0 & \text{otherwise} \end{cases} \\
f_6(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-1} \text{ ends in ``ical''} \\ 0 & \text{otherwise} \end{cases} \\
f_7(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, \text{``model'' is not in } w_1, \ldots w_{i-1} \\ 0 & \text{otherwise} \end{cases} \\
f_8(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, \text{``grammatical'' is in } w_1, \ldots w_{i-1} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

其中 *x* 是一个文字序列：$w_1w_2\ldots w_{i-1}$，y 为一个词。

可以看出，各种 *n* 元语言关系及其变种都可以作为映射函数，当然 Log-linear 模型的分母需要遍历所有的词汇，看起来计算量巨大，但因为绝大部分词汇都是属于 otherwise 分支，即为 0，所以计算上也是可以接受的。

4.6 Log-Bilinear 模型

Log-bilinear[13]是 Hinton 提出的一个模型，和 Log-linear 的区别在于映射函数部分，之前 *f*(*x*,*y*)对于一个具体的输入数据都是直接映射到一个 *K* 维实数向量，而 Log-bilinear 模型中则是直接采用和 *y* 对应的向量 v_y。因为输入的 *v* 和 v_y 都是变量，所以这个模型就变为取 log 后成双线性了，因此成为 Log-bilinear。这样带来的问题是分母上的计算变得密集，当然这难不倒聪明的大佬们，改进的方法则是引入分类或聚类的思想，不直接做多分类，而是每次做二分类。这就是 4.6 节中的层次化 Log-bilinear 模型。

4.6 层次化 Log-Bilinear 模型

Hinton 进一步结合 Bengio 的层次化概率语言模型[14]提出了层次化 Log-bilinear 模型[13]。这种模型的特征是：

1. 叶子结点为词（word）的二叉树，而内部结点虽然也有对应的向量，但并不对应到具体的词。
2. 每个结点上都要进行决策。

N 元层次化 Log-bilinear 模型中由上下文预测下一个词为 *w* 的公式为：

$p(w_n=w|w_{1:n-1}) = \prod_{i=1}^{K} p(d_i|q_i, w_{1:n-1})$

其中 d_i 是 w 编码中第 i 位的值，q_i 是那个编码对应路径上的第 i 个结点。

$$p(d_i|q_i, w_{1:n-1}) = \sigma({v_{context}}^T \cdot q_i)$$

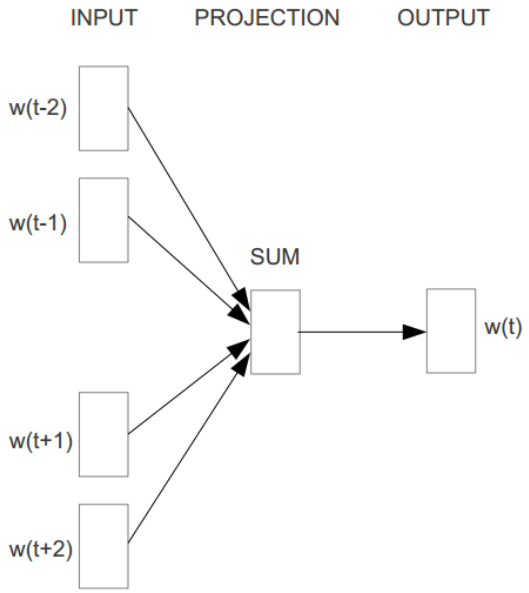
其中 $v_{context}$ 是对应上下文的向量，比如下一章的 Skip-gram 采用的是输入词 input 对应的向量，CBOW 中是 $w_{1:n-1}$ 对应 *n-1* 个向量之和，当然你还能带权，你还能加 bias，还可以搞个牛逼的映射函数，一个词还能多个编码。此外，如何编码，自然的方法是根据语义分类聚类之类，简单点 Huffman 编码，也就是 word2vec 的做法，还有在每个结点预测的时候，除了用 likelihood（对应伯努利概率分布），还可以用方差（对应高斯分布）等。总之这个地方，心有多大，舞台就有多大，哪天你搞出一个更好的 xxx-gram 一点也不稀奇。

Word2vec 提出的新的 Log-Bilinear 模型包括 CBOW 和 Skip-gram 两种，具体的介绍请参考下一章。

五、模型

5.1 CBOW

CBOW 是 Continuous Bag-of-Words Model 的缩写，是一种与前向 NNLM 类似的模型，不同点在于 CBOW 去掉了最耗时的非线性隐层且所有词共享隐层。如下图所示。可以看出，CBOW 模型是预测 $P(w_t|w_{t-k},w_{t-(k-1)}...,w_{t-1},w_{t+1},w_{t+2}...,w_{t+k})$。



从输入层到隐层所进行的操作实际就是上下文向量的加和，具体的代码如下。其中 *sentence_position* 为当前 word 在句子中的下标。以一个具体的句子 A B C D 为例，第一次进入到下面代码时当前 word 为 A，*sentence_position* 为 0。b 是一个随机生成的 0 到 window-1 的词，整个窗口的大小为（2*window + 1 – 2*b），相当于左右各看 window-b 个词。可以看出随着窗口的从左往右滑动，其大小也是随机的 3（b=window-1）到 2*window+1(b=0)之间随机变通，即随机值 b 的大小决定了当前窗口的大小。代码中的 neu1 即为隐层向量，也就是上下文（窗口内除自己之外的词）对应 vector 之和。

```
// in -> hidden
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
  c = sentence_position - window + a;
  if (c < 0) continue;
  if (c >= sentence_length) continue;
  last_word = sen[c];
  if (last_word == -1) continue;
  for (c = 0; c < layer1_size; c++) neu1[c] += syn0[c + last_word * layer1_size];
}
```

CBOW 有两种可选的算法：层次 Softmax 和 Negative Sampling。这一部分 Mikolov 并没有在论文中进行阐述，下面的相关讨论均来自于本文作者对 word2vec 源码的分析。这种层次 Softmax 算法结合了 Huffman 编码，每个词 w 都可以从树的根结点沿着唯一一条路径被访问到。这种 Huffman 编码用于神经网络语言模型并不是 word2vec 首创，作者在他之前的论文[4]和[5]中就有提到。假设 n(w, j)为这条路径上的第 j 个结点，且 L(w)为这条路径的长度，注意 j 从 1 开始编码，即 n(w, 1)=root，n(w, L(w)) = w。对于第 j 个结点，层次 Softmax 定义的 Label 为 1 - code[j]，这里其实也可以把 Label 定义为 code[j]，得到的向量也差不多，而输出 f 为：

$$f = \sigma(neu1^T \cdot syn1)$$

Loss 为负的 Log 似然，即：

$$Loss = -Likelihood = -(1 - code[j])logf - code[j]\log(1 - f)$$

那么梯度为:

$$\text{Gradient} = \frac{\partial Loss}{\partial neu1} = -(1 - code[j]) \cdot (1 - f) \cdot syn1 + code[j] \cdot f \cdot syn1$$
$$= -(1 - code[j] - f) \cdot syn1$$
$$\text{Gradient} = \frac{\partial Loss}{\partial syn1} = -(1 - code[j]) \cdot (1 - f) \cdot neu1 + code[j] \cdot f \cdot neu1$$
$$= -(1 - code[j] - f) \cdot neu1$$

需要注意的是，word2vec 源码中的 g 实际为负梯度中公共的部分与 Learning rate alpha 的乘积，Mikolov 的注释有点坑爹，也许可以改为：'g' is the error multiplied by the learning rate。

```
if (hs) for (d = 0; d < vocab[word].codelen; d++) {
  f = 0;
  l2 = vocab[word].point[d] * layer1_size;
  // Propagate hidden -> output
  for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1[c + l2];
  if (f <= -MAX_EXP) continue;
  else if (f >= MAX_EXP) continue;
  else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
  // 'g' is the gradient multiplied by the learning rate
  g = (1 - vocab[word].code[d] - f) * alpha;
  // Propagate errors output -> hidden
  for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
  // Learn weights hidden -> output
  for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * neu1[c];
}
```

CBOW 的第二种算法是 Negative Sampling，这种方法的原理说简单很简单，就是随机搞点负例，说复杂也很复杂，Mikolov 硬是扯上了一个理论比较复杂的 Noise Contrastive Estimation(NCE)，感兴趣的不妨仔细研究一下。

Negative Sampling 在源代码中就是随机生成 negative 个（也有可能少一点，如果随机时候撞上了原来的 word）负例，原来的 word 为正例，Label 为 1，其他

随机生成的 Label 为 0，输出 f 仍为：

$$f = \sigma(neu1^T \cdot syn1)$$

Loss 为负的 Log 似然(因采用随机梯度下降，这里只看一个 word 中的一层)，即：

$$Loss = -Log\ Likelihood = -label \cdot logf - (1 - label) \cdot log(1 - f)$$

那么梯度为:

$$\text{Gradient}_{neu1} = \frac{\partial Loss}{\partial neu1} = -label \cdot (1 - f) \cdot syn1 + (1 - label) \cdot f \cdot syn1$$
$$= -(label - f) \cdot syn1$$

$$\text{Gradient}_{syn1} = \frac{\partial Loss}{\partial syn1} = -label \cdot (1 - f) \cdot neu1 + (1 - label) \cdot f \cdot neu1$$
$$= -(label - f) \cdot neu1$$

同样注意代码中 g 并非梯度，可以看做是一个乘了 learning rate 的 error（label 与输出 f 的差）。

```
// NEGATIVE SAMPLING
if (negative > 0) for (d = 0; d < negative + 1; d++) {
  if (d == 0) {
    target = word;
    label = 1;
  } else {
    next_random = next_random * (unsigned long long)25214903917 + 11;
    target = table[(next_random >> 16) % table_size];
    if (target == 0) target = next_random % (vocab_size - 1) + 1;
    if (target == word) continue;
    label = 0;
  }
  l2 = target * layer1_size;
  f = 0;
  for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1neg[c + l2];
  if (f > MAX_EXP) g = (label - 1) * alpha;
  else if (f < -MAX_EXP) g = (label - 0) * alpha;
  else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
  for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
  for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * neu1[c];
}
```

由隐层到输入层的梯度传播则更加简单，因为隐层为输入层各变量的加和，因此输入层的梯度即为隐层梯度（注意每次循环 neu1e 都被置零了）。

```
// hidden -> in
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
  c = sentence_position - window + a;
  if (c < 0) continue;
  if (c >= sentence_length) continue;
  last_word = sen[c];
  if (last_word == -1) continue;
  for (c = 0; c < layer1_size; c++) syn0[c + last_word * layer1_size] += neu1e[c];
}
```

5.2 Skip-Gram

Skip-Gram 模型的图与 CBOW 正好方向相反，从图中看应该 Skip-Gram 应该预测概率 $p(w_i \mid w_t)$，其中$t - c \le i \le t + c$且$i \ne t$，c 是决定上下文窗口大小的常数，c 越大则需要考虑的 pair 就越多，一般能够带来更精确的结果，但是训练时间也会增加。假设存在一个 $w_1, w_2, w_3, \ldots, w_T$ 的词组序列，Skip-gram 的目标是最大化：

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j\neq 0} log\ p(w_{t+j}|w_t)$$

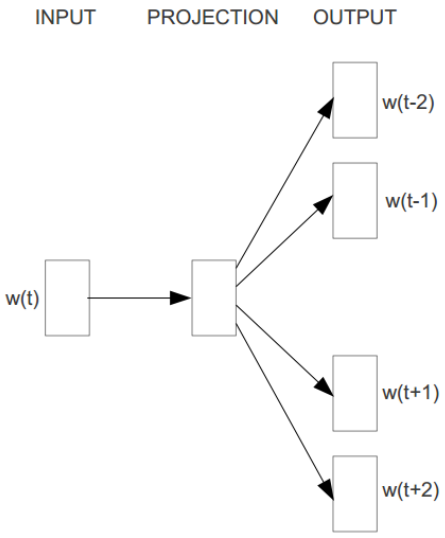
基本的 Skip-Gram 模型定义 p($w_O|w_I$)为：

$$p(w_O|\ w_I) = \frac{e^{v'_{w_O}{}^T v_{w_I}}}{\sum_{w=1}^{W} e^{v'_w{}^T v_{w_I}}}$$

从公式不能看出，Skip-gram 是一个对称的模型，如果 w_t 为中心词时 w_k 在其窗口内，则 w_t 也必然在以 w_k 为中心词的同样大小窗口内，也就是：

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j\neq 0} log\ p(w_{t+j}|w_t) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j\neq 0} log\ p(w_t|w_{t+j})$$

同时，Skip-gram 中的每个词向量表征了上下文的分布。Skip-gram 中的 skip 是指在一定窗口内的词两两都会计算概率，就算他们之间隔着一些词，这样的好处是“白色汽车”和“白色的汽车”很容易被识别为相同的短语。



与 CBOW 类似，Skip-Gram 也有两种可选的算法：层次 Softmax 和 Negative Sampling。层次 Softmax 算法也结合了 huffman 编码，每个词 w 都可以从树的根结点沿着唯一一条路径被访问到。假设 n(w, j)为这条路径上的第 j 个结点，且 L(w)为这条路径的长度，注意 j 从 1 开始编码，即 n(w, 1)=root，n(w, L(w)) = w。层次 Softmax 定义的概率 p(w|w_I)为：

$$\mathrm{p(w}|w_I) = \prod_{j=1}^{L(w)-1} \sigma([\![n(w,j+1) = ch(n(w,j))]\!] \cdot {v'_{n(w,j)}}^T v_I)$$

其中:

$$[\![x]\!] = \begin{cases} 1, & if\ x\ is\ true \\ -1, & else \end{cases}$$

ch(*n*(*w*,*j*))既可以是 *n*(*w*,*j*)的左子结点也可以是 *n*(*w*,*j*)的右子结点，word2vec 源代码中采用的是左子结点（Label 为 1-code[j]），其实此处改为右子结点也是可以的。

$Loss_{pair}$为负的 Log 似然(因采用随机梯度下降，这里只看一个 pair)，即：

$$Loss_{pair} = -Log\ Likelihood_{pair}$$
$$= -log\big(\mathrm{p(w|}w_I\mathrm{)}\big)$$
$$= -\sum_{j=1}^{L(w)-1} \log(\sigma([\![n(w,j+1) = ch(n(w,j))]\!] \cdot {v'_{n(w,j)}}^T v_I))$$

对 LR 似然函数比较熟悉的读者应该能看出该公式与 CBOW 中的对应 Loss 公式其实是很类似的，唯一不同是把$\sigma(neu1 \cdot syn1)$变成了$\sigma({v'_{n(w,j)}}^T v_I)$。这里顺着论文[2]的公式推导一下梯度。因为采用随机梯度下降，每次只要 *w* 的一层，假设为第 j 层，那么对应的该层 loss 为：

$$\text{Loss=}-\text{Log Likelihood=}-\log\Big(\sigma\big([\![n(w,j+1) = ch\big(n(w,j)\big)]\!] \cdot {v'_{n(w,j)}}^T v_I\big)\Big)$$

1）如果$[\![n(w,j+1) = ch\big(n(w,j)\big)]\!]$为 true，即当前结点为左子结点，那么

$$\text{Loss=}-\log\Big(\sigma\big({v'_{n(w,j)}}^T v_I\big)\Big)$$

那么梯度为:

$$\text{Gradient}_{v'_{n(w,j)}} = \frac{\partial Loss}{\partial v'_{n(w,j)}} = -\Big(1 - \sigma\big({v'_{n(w,j)}}^T v_I\big)\Big) \cdot v_I$$

$$\text{Gradient}_{v_I} = \frac{\partial Loss}{\partial v_I} = -\Big(1 - \sigma\big({v'_{n(w,j)}}^T v_I\big)\Big) \cdot v'_{n(w,j)}$$

2）如果$[\![n(w,j+1) = ch\big(n(w,j)\big)]\!]$为 false，即当前结点为右子结点，那么

$$\text{Loss=}-\log\Big(\sigma\big(-{v'_{n(w,j)}}^T v_I\big)\Big) = -\log\Big(1 - \sigma\big({v'_{n(w,j)}}^T v_I\big)\Big)$$

那么梯度为:

$$\text{Gradient}_{v'_{n(w,j)}} = \frac{\partial Loss}{\partial v'_{n(w,j)}} = \sigma\big({v'_{n(w,j)}}^T v_I\big) \cdot v_I$$

$$\text{Gradient}_{v_I} = \frac{\partial Loss}{\partial v_I} = \sigma\big({v'_{n(w,j)}}^T v_I\big) \cdot v'_{n(w,j)}$$

3）合并 1）和 2）得：

$$\text{Gradient}_{v'_{n(w,j)}} = \frac{\partial Loss}{\partial v'_{n(w,j)}} = -\Big(1 - \text{code[j]} - \sigma\big({v'_{n(w,j)}}^T v_I\big)\Big) \cdot v_I$$

$$\text{Gradient}_{v_I} = \frac{\partial Loss}{\partial v_I} = -\Big(1 - \text{code[j]} - \sigma\big({v'_{n(w,j)}}^T v_I\big)\Big) \cdot v'_{n(w,j)}$$

此处相关的代码如下图所示，注意 *g* 就是梯度中的公共部分$\Big(1 - \text{code[j]} - \sigma\big({v'_{n(w,j)}}^T v_I\big)\Big)$与 learning rate alpha 的乘积。

```
// HIERARCHICAL SOFTMAX
if (hs) for (d = 0; d < vocab[word].codelen; d++) {
  f = 0;
  l2 = vocab[word].point[d] * layer1_size;
  // Propagate hidden -> output
  for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1[c + l2];
  if (f <= -MAX_EXP) continue;
  else if (f >= MAX_EXP) continue;
  else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
  // 'g' is the gradient multiplied by the learning rate
  g = (1 - vocab[word].code[d] - f) * alpha;
  // Propagate errors output -> hidden
  for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
  // Learn weights hidden -> output
  for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * syn0[c + l1];
}
```

Negative Sampling 和隐层往输入层传播梯度部分与 CBOW 区别不大，这里不再赘述。相关代码如下：

```
// NEGATIVE SAMPLING
if (negative > 0) for (d = 0; d < negative + 1; d++) {
  if (d == 0) {
    target = word;
    label = 1;
  } else {
    next_random = next_random * (unsigned long long)25214903917 + 11;
    target = table[(next_random >> 16) % table_size];
    if (target == 0) target = next_random % (vocab_size - 1) + 1;
    if (target == word) continue;
    label = 0;
  }
  l2 = target * layer1_size;
  f = 0;
  for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1neg[c + l2];
  if (f > MAX_EXP) g = (label - 1) * alpha;
  else if (f < -MAX_EXP) g = (label - 0) * alpha;
  else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
  for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
  for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * syn0[c + l1];
}
```

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

5.3 为什么要使用 Hierarchical Softmax 或 Negative Sampling

前面提到到 skip-gram 中的条件概率为：

$$p(w_O|w_I) = \frac{e^{v'^{T}_{w_O} v_{w_I}}}{\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}}$$

这其实是一个多分类的 logistic regression，即 softmax 模型，对应的 label 是 One-hot representation，只有当前词对应的位置为 1，其他为 0。

普通的方法是 $p(w_o|w_i)$ 的分母要对所有词汇表里的单词求和，这使得计算梯度很耗时。

另外一种方法是只更新当前 w_o、w_i 两个词的向量而不更新其他词对应的向量，也就是不管归一化项，这种方法也会使得优化收敛的很慢。

Hierarchical Softmax 则是介于两者之间的一种方法，使用的办法其实是借助了分类的概念。假设我们是把所有的词都作为输出，那么“桔子”、“汽车”都是

混在一起。而 Hierarchical Softmax 则是把这些词按照类别进行区分的。对于二叉树来说，则是使用二分类近似原来的多分类。例如给定 w_i，先让模型判断 w_o 是不是名词，再判断是不是食物名，再判断是不是水果，再判断是不是“桔子”。虽然 word2vec 论文里，作者是使用哈夫曼编码构造的一连串两分类。但是在训练过程中，模型会赋予这些抽象的中间结点一个合适的向量，这个向量代表了它对应的所有子结点。因为真正的单词公用了这些抽象结点的向量，所以 Hierarchical Softmax 方法和原始问题并不是等价的，但是这种近似并不会显著带来性能上的损失同时又使得模型的求解规模显著上升。

Negative Sampling 也是用二分类近似多分类，区别在于使用的是 one-versus-one 的方式近似，即采样一些负例，调整模型参数使得模型可以区分正例和负例。换一个角度来看，就是 Negative Sampling 有点懒，他不想把分母中的所有词都算一次，就稍微选几个算算，选多少，就是模型中负例的个数，怎么选，一般就需要按照词频对应的概率分布来随机抽样了。

六、Tricks

总体来说 word2vec 的作者 Mikolov 是个实用主义者，什么好用就用什么，并不完全追求其理论解释，这个其实也是整个 Deep Learning 研究领域的一贯作风了，所以不能看到 RBM 之类的求解算法中有无数并无严格理论证明的 trick。除了 Huffman 编码，Negative Sampling 等方法没有严格理论证明外，在 word2vec 的实现代码中 Mikolov 也用了不少 trick，这也许给初学者阅读代码带来一定困难，本节主要讲解这些 trick，只对原理感兴趣的同学可以跳过。

6.1 指数运算

由于 word2vec 大量采用 logistic 公式，所以训练开始之前预先算好这些指数值，采用查表的方式得到一个粗略的指数值。虽然有一定误差，但是能够较大幅度提升性能。

代码中的 EXP_TABLE_SIZE 为常数 1000，当然这个值越大，精度越高，同时内存占用也会越多。MAX_EXP 为常数 6。循环内的代码实质为：

expTable[i] = exp((i -500)/ 500 * 6)　　即 e^-6 ~ e^6

expTable[i] = 1/(1+e^6) ~ 1/(1+e^-6)　　即 0.01 ~ 1 的样子。

相当于把横坐标从-6 到 6 等分为 1000 份，每个等分点上（1001 个）记录相应的 logistic 函数值方便以后的查询。感觉这里有个 bug，第 1001 个等分点（下标为 EXP_TABLE_SIZE）并未赋值，是对应内存上的一个随机值。

```
expTable = (real *)malloc((EXP_TABLE_SIZE + 1) * sizeof(real));
for (i = 0; i < EXP_TABLE_SIZE; i++) {
  expTable[i] = exp((i / (real)EXP_TABLE_SIZE * 2 - 1) * MAX_EXP); // Precompute the exp() table
  expTable[i] = expTable[i] / (expTable[i] + 1);                   // Precompute f(x) = x / (x + 1)
}
```

相关查询的代码如下所示。前面三行保证 f 的取值范围为[-MAX_EXP, MAX_EXP]，这样(f + MAX_EXP)/MAX_EXP/2 的范围为[0,1]，那下面的 expTable 的下标取值范围为[0, EXP_TABLE_SIZE]。一旦取值为 EXP_TABLE_SIZE，就会引发上面所说的 bug。

```
for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1neg[c + l2];
if (f > MAX_EXP) g = (label - 1) * alpha;
else if (f < -MAX_EXP) g = (label - 0) * alpha;
else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
```

6.2 按 word 分布随机抽样

word2vec 中的 Negative Sampling 需要随机生成一些负例，通过随机找到一个词和当前 word 组成 pair 生成负例。那么随机找词的时候就要参考每个词的词频，也就是需要根据词的分布进行抽样。这个问题实际是经典的 Categorical Distribution Sampling 问题。关于 Categorical Distribution 的基本知识及其抽样可以参考维基百科相关资料：

http://en.wikipedia.org/wiki/Categorical_distribution，
http://en.wikipedia.org/wiki/Categorical_distribution#Sampling

Categorical Distribution Sampling 的基本问题是：已知一些枚举变量（比如 index，或者 term 之类）及其对应的概率，如果根据他们的概率分布快速进行抽样。举例来说：已知字符 a，b，c 出现概率分别为 1/2, 1/3, 1/6，设计一个算法能够随机抽样 a，b，c 使得他们满足前面的出现概率。

维基百科上提到两种抽样方法，而 word2vec 中的实现采用了一种近似但更高效的离散方法，具体描述如下：

方法一

把类别映射到一条线段上，线段上的点为前面所有类别的概率之和，使用时依次扫描直到改点概率与随机数匹配。

事先准备：

1. 计算每个类别或词的未归一化分布值（对于词来即词频）；
2. 把上面计算的值加和，利用这个和进行概率归一化；
3. 引入类别或词的一定顺序，比如词的下标；
4. 计算 CDF（累积分布函数），即把每个点上的值修改为其前面所有类别或词的归一化概率之和。

使用时：

1. 随机抽取一个 0 到 1 之间的随机数；
2. 定位 CDF 中小于或等于该随机数的最大值，该操作如果使用二分查找可以在 O(log(k))时间内完成；
3. 返回 CDF 对应的类别或词。

该方法使用时的时间复杂度 O(log(k))，空间复杂度 O(k) ，k 为类别数或词数。

方法二

方法二稍有些区别，他每次抽样 n 个，如果使用时是一个一个抽的话，可以对这 n 个进行循环，当然 n 越大，随机性越好，n 与最终的抽样次数尽量匹配比较好。

```
r = 1;
s = 0;
for (i = 1; i <= k; i++) {
```

v = 根据二项分布 binomial(n, p[i] / r)进行抽样得到的整数。//期望值应该是 n*p[i]/r
//产生 v 个下标为 i 的样本
for (j = 1; j <= v; j++)
z[s++] = i;
n = n – v; //下次需要继续生成 n 个样本
r = r – p[i]; //剩余样本的概率和
}
随机重新排列（shuffle）z 中的左右样本；
返回 z。

这种方法每次可抽 n 个，随机抽样时的时间复杂度 O(1)，空间复杂度 O(n)。

方法三

方法三即 word2vec 实现方法。

1）类似于方法一，先将概率以 CDF 形式排列在一条线段上，以字符 a，b，c 出现概率分别为 1/2, 1/3, 1/6 为例，线段如下，左端点为 0，右端点为 1，中间分割点分别为 1/2,(1/2+1/3),(1/2+1/3+1/6)

|_____a_____|_____b_____|____c____|

2） 讲线段划分为 m 段，0 对应左端点，即上面的概率 0，m 对应右端点，即上面的概率 1，与上面的线段做一次映射，那么就知道 0-m 中任意整数所对应的字符了。

|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|

word2vec 中的具体代码如下，需要注意的是概率大小不是与词频正比，而是与词频的 power 次方成正比。

```
void InitUnigramTable() {
  int a, i;
  long long train_words_pow = 0;
  real d1, power = 0.75;
  table = (int *)malloc(table_size * sizeof(int));
  for (a = 0; a < vocab_size; a++) train_words_pow += pow(vocab[a].cn, power);
  i = 0;
  d1 = pow(vocab[i].cn, power) / (real)train_words_pow;
  for (a = 0; a < table_size; a++) {
    table[a] = i;
    if (a / (real)table_size > d1) {
      i++;
      d1 += pow(vocab[i].cn, power) / (real)train_words_pow;
    }
    if (i >= vocab_size) i = vocab_size - 1;
  }
}
```

该方法不是特别精准，可以无限次抽样，随机抽样时的时间复杂度 O(1)，空间复杂度 O(m)，m 为离散区间的段数，段数越多越精准，但空间复杂度也越高。

其他方法

其实该问题还有很多其他抽样方法，感兴趣的同学不妨调研一下。同时这种抽样其实和计数排序还有非常密切联系，尤其是方法三，只是把计数排序中的概率或者计数映射到了 0-m。

6.3 哈希编码

这个比较简单，哈希冲突解决采用的是线性探测的开放定址法。相关代码如下：

```
// Returns hash value of a word
int GetWordHash(char *word) {
  unsigned long long a, hash = 0;
  for (a = 0; a < strlen(word); a++) hash = hash * 257 + word[a];
  hash = hash % vocab_hash_size;
  return hash;
}

// Returns position of a word in the vocabulary; if the word is not found, returns -1
int SearchVocab(char *word) {
  unsigned int hash = GetWordHash(word);
  while (1) {
    if (vocab_hash[hash] == -1) return -1;
    if (!strcmp(word, vocab[vocab_hash[hash]].word)) return vocab_hash[hash];
    hash = (hash + 1) % vocab_hash_size;
  }
  return -1;
}
```

6.4 随机数

作者自己编写了随机数生成模块，方法比较简单，就是每次乘以一个很大的数字再加 11 然后取模再归一化。

```
real ran = (sqrt(vocab[word].cn / (sample * train_words)) + 1) * (sample * train_words) / vocab[word].cn;
next_random = next_random * (unsigned long long)25214903917 + 11;
if (ran < (next_random & 0xFFFF) / (real)65536) continue;
```

6.5 回车符

这个 trick 其实不太优美，这里提一下的原始是这块可能有一点小 bug。word2vec 中将训练数据中的回车符都替换成了</s>这样的字符，而且数据读取之前首先占据了 index 0。

```
AddWordToVocab((char *)"</s>");
```

在排序的时候对改词又进行了保护，使得该词排在最前面。

```
// Sort the vocabulary and keep </s> at the first position
qsort(&vocab[1], vocab_size - 1, sizeof(struct vocab_word), VocabCompare);
```

但是在删除长尾词时，如果下标为 a 的词词频小于 min_count(可设置，默认为 5)，则删除最后一个字符。如果回车换行符过少，少于 min_count，这里会导致多删除一个词频大于等于 min_count 的词，当然这样的小 bug 无伤大雅。

```
for (a = 0; a < size; a++) {
  // Words occuring less than min_count times will be discarded from the vocab
  if (vocab[a].cn < min_count) {
    vocab_size--;
    free(vocab[vocab_size].word);
  } else {
    // Hash will be re-computed, as after the sorting it is not actual
    hash=GetWordHash(vocab[a].word);
    while (vocab_hash[hash] != -1) hash = (hash + 1) % vocab_hash_size;
    vocab_hash[hash] = a;
    train_words += vocab[a].cn;
  }
}
```

6.6 高频词亚采样

这里的亚采样是指 Sub-Sampling，不知道这样翻译是否准确。Mikolov 在论文[2]指出这种亚采样能够带来 2 到 10 倍的性能提升，并能够提升低频词的表示精度。这个策略论文和代码也有不一致的地方，对 mikolov 来说，肯定是哪个好就用哪个，所以这种地方都是可以结合应用进行调整的地方。论文中每个词 w_i 被丢弃的概率为：

$$\mathrm{p}(w_i) = 1 - \sqrt{\frac{sample}{freq(w_i)}}$$

sample是一个可以设置的参数，demo-word.sh 中是 10-3，freq(w_i)则为 wi 的词频。具体的实现代码如下。

```
real ran = (sqrt(vocab[word].cn / (sample * train_words)) + 1) * (sample * train_words) / vocab[word].cn;
next_random = next_random * (unsigned long long)25214903917 + 11;
if (ran < (next_random & 0xFFFF) / (real)65536) continue;
```

从具体代码可以看出，wi 被丢弃的概率为：

$$\mathrm{p}(w_i) = 1 - \left(\sqrt{\frac{sample}{freq(w_i)}} + \frac{sample}{freq(w_i)}\right)$$

七、分布式实现

Jeffrey Dean 在论文《Large Scale Distributed Deep Networks》[9]中介绍了异步 SGD 训练大规模深度网络的方法。我们借鉴该论文的思想实现了分布式版本的 word2vec。

方法很简单，将训练数据分成若干块交由不同 worker，worker 用自己的数据计算 mini-batch 梯度Δv_i（输入向量的梯度）和Δv_o（输出向量的梯度），将其传到 server，server 收到梯度后立即更新所有相关的词向量。server 总是保存一份最新的参数，worker 参数可能是陈旧的，worker 每隔一段时间会到 server 拉取最新的参数。server 端每隔一段时间保存参数到文件。

其中数据分割直接使用 cowork 提供的机制，server 使用 odis.rpc2，实现了传递梯度（push）、拉取参数（pull）等 rpc 调用。为了防止 server 负载过高，server 只允许少量 worker 同时传递梯度或拉取参数，使用信号量实现，worker 在每次传递或拉取调用钱都要先调用一个获取许可的 rpc 方法（acquire）。最初实现的

acquire 方法是阻塞的，128 worker 的处理速度大概是~100k words/s 。进一步优化时发现，这里其实可以使用非阻塞的，所以实现了 tryAcquire 方法，worker 如果拿到了许可，那么 push 或 pull，否则马上返回继续处理自己的数据，累计梯度，下次再来试。这种方法显著提高了训练速度，达到了~1000k words/s。

为了方便调试和性能调优，有必要在 server 端 log 中打印出有用的统计信息，例如我会在 server 端打印出 rpc 方法被调用的次数，网络开销，每秒处理单词数，平均每次参数更新需要请求 ticket 的次数。这其中的许多信息需要 worker 定时向 server 发送，由 server 汇总后输出。

八、总结

总结一下 word2vec 高效率的原因，本文作者认为主要有以下几点：

1. 去掉了费时的非线性隐层；
2. Huffman 编码相当于做了一定聚类，不需要统计所有词对；
3. Negative Sampling；
4. 随机梯度算法；
5. 只过一遍数据，不需要反复迭代；
6. 编程实现中的一些 trick，比如指数运算的预计算，高频词亚采样等。

word2vec 可调整的超参数有很多，其中比较重要的有（排名不分先后）：

1. –size：向量维数
2. –window：上下文窗口大小
3. –sample：高频词亚采样的阈值
4. –hs：是否采用层次 softmax
5. –negative：负例数目
6. –min-count：被截断的低频词阈值
7. –alpha：开始的 learning rate
8. –cbow：使用 CBOW 算法

Mikolov 关于超参数的建议如下：

1. 模型架构：Skip-gram 更慢一些，但是对低频词效果更好；对应的 CBOW 则速度更快一些。
2. 训练算法：层次 softmax 对低频词效果更好；对应的 negative sampling 对高频词效果更好，向量维度较低时效果更好。
3. 高频词亚采样：对大数据集合可以同时提高精度和速度，sample 的取值在 1e-3 到 1e-5 之间效果最佳。
4. 词向量的维度：一般维度越高越好，但并不总是这样。
5. 窗口大小：Skip-gram 一般 10 左右，CBOW 一般 5 左右。

在未来应用方面，word2vec 除了在 NLP，在别的领域稍加改造也是可以很好利用的，尤其 word2vec 中所用的模型和策略非常值得借鉴。

参考代码

总体来说，还是 google 自己的代码最权威，除非感觉阅读还有障碍。

google 代码：http://word2vec.googlecode.com/svn/trunk/

400 行 C++11 代码：https://github.com/jdeng/word2vec
Python 版本：http://radimrehurek.com/gensim/models/word2vec.html
java 版本：https://github.com/ansjsun/word2vec_java
并行 java 版本：https://github.com/siegfang/word2vec
CUDA 版本：https://github.com/whatupbiatch/cuda-word2vec

参考文献

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.
[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.
[3] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In Proceedings of NAACL HLT, 2013.
[4] Tomas Mikolov, Stefan Kombrink, Lukas Burget, Jan Cernocky, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In Acoustics, Speech and Signal Processing (ICASSP), 2011, IEEE International Conference on, pages 5528–5531. IEEE, 2011.
[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. ICLR Workshop, 2013.
[6] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In Proceedings of the international workshop on artificial intelligence and statistics, pages 246–252, 2005.
[7] Andriy Mnih and Geoffrey E Hinton. A scalable hierarchical distributed language model. Advances in neural information processing systems, 21:1081–1088, 2009.
[8] Hinton, Geoffrey E. "Learning distributed representations of concepts." Proceedings of the eighth annual conference of the cognitive science society. 1986.
[9] R. Rosenfeld, "Two decades of statistical language modeling: where do we go from here?", Proceedings of the IEEE, 88(8), 1270-1288, 2000.
[10] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. "Large Scale Distributed Deep Networks". Proceedings of NIPS, 2012.
[11] http://licstar.net/archives/328
[12] http://www.cs.columbia.edu/~mcollins/loglinear.pdf
[13] A. Mnih and G. Hinton. Three new graphical models for statistical language modelling. Proceedings of the 24th international conference on Machine learning, pages 641–648, 2007
[14] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In Robert G. Cowell and Zoubin Ghahramani, editors, AISTATS'05, pages 246–252, 2005.